ADO.NET Entity Framework

Каркас сущностей ADO.NET (ADO.NET Entity Framework) — объектно-реляционный каркас отображения, основанный на .NET 3.5. В главе 27 мы продемонстрировали объектно-реляционное отображение посредством LINQ to SQL. LINQ to SQL предоставляет простое средство отображения для отношений ассоциации и наследования. ADO.NET Entity Framework предлагает намного больше опций для отражения ассоциации и наследования. Другое отличие между LINQ to SQL и ADO.NET Entity Framework состоит в том, что ADO.NET Entity Framework представляет собой модель на основе поставщиков, позволяющую подключаться к ней другим поставщикам баз данных.

В настоящем приложении рассматриваются следующие темы:

- каркас сущностей ADO.NET;
- уровни каркаса сущностей;
- сущности;
- объектные контексты;
- отношения;
- запросы объектов;
- обновления;
- LINQ to Entities.

Настоящее приложение основано на версии Beta 3 этого каркаса, которая вышла несколькими месяцами позже продукта .NET 3.5, поэтому некоторые имена методов и классов могут отличаться от тех, что вы встретите здесь.

В этом приложении используются базы данных Books, Formula 1 и Northwind. Вы можете загрузить базу Northwind с `msdn.microsoft.com`; базы Books и Formula 1 входят в состав кода примеров для книги.

Обзор ADO.NET Entity Framework

ADO.NET Entity Framework обеспечивает отображение схемы реляционной базы данных на объекты. Реляционные базы данных и объектно-ориентированные языки определяют ассоциации по-разному. Например, база примеров Microsoft Northwind со-

держит таблицы Customers и Orders. Чтобы получить доступ ко всем строкам Orders определенного заказчика, вы должны составить SQL-оператор соединения. В объектно-ориентированных языках принято определять классы Customer и Order и обращаться к заказам заказчика через свойство Orders класса Customer.

Со времен .NET 1.0 для объектно-реляционного отображения можно было использовать класс DataSet и типизированные множества данных. Множества данных (datasets) очень похожи на структуры баз данных, содержащие классы DataTable, DataRow, DataColumn и DataRelation. ADO.NET Entity Framework обеспечивает поддержку явно определенных сущностных классов, которые полностью независимы от структуры базы данных и отображают их на таблицы и ассоциации базы данных. Использование в приложении объектов защищает приложение от изменений в базе данных.

ADO.NET Entity Framework использует Entity SQL для определения запросов к хранилищу на основе сущностей. LINQ to Entities дает возможность применять синтаксис LINQ для опроса данных.

Объектный контекст сохраняет знание об измененных сущностях, чтобы знать, когда сущности должны быть записаны обратно в хранилище.

Пространства имен, содержащие классы ADO.NET Entity Framework, перечислены в табл. A.1.

Таблица A.1. Пространства имен, содержащие классы ADO.NET Entity Framework

Пространство имен	Описание
System.Data	Это главное пространство имен ADO.NET. Для ADO.NET Entity Framework это пространство содержит классы исключений, относящихся к сущностям, например, MappingException и QueryException.
System.Data.Common	Это пространство имен содержит классы, разделяемые поставщиками данных .NET. Класс DbProviderServices — абстрактный базовый класс, который может быть реализован поставщиком ADO.NET Entity Framework.
System.Data.Common.CommandTrees	Это пространство имен содержит классы для построения дерева выражений.
System.Data.Entity.Design	Это пространство имен содержит классы, используемые дизайнером для создания файлов сущностной модели данных — Entity Data Model (EDM).
System.Data.EntityClient	Это пространство имен специфицирует классы .NET Framework Data Provider для доступа к Entity Framewrok. EntityConnection, EntityCommand и EntityDataReader могут применяться для доступа к Entity Framework.
System.Data.Objects	Пространство имен содержит классы для опроса и обновления баз данных. Класс ObjectContext инкапсулирует соединение с базой данных и служит шлюзом для методов создания, чтения, обновления и удаления. Класс ObjectQuery представляет запрос к хранилищу, а CompiledQuery — кэшированный запрос.
System.Data.Objects.DataClasses	Пространство имен, содержащее классы и интерфейсы, необходимые сущностям.

Уровни каркаса сущностей

ADO.NET Entity Framework предоставляет несколько уровней для отображения таблиц базы данных на объекты. Вы можете начать со схемы базы данных и использовать шаблон элемента Visual Studio для создания полного отображения. Вы можете также начать проектирование сущностных классов в дизайнере и отобразить их на базу данных, где таблицы и ассоциации между ними могут иметь очень разную структуру.

Уровни, которые должны быть определены:

- логический — этот уровень определяет реляционные данные;
- концептуальный — этот уровень определяет классы .NET;
- отображения — этот уровень определяет отображение классов .NET на реляционные таблицы и ассоциации.

Начнем с простой схемы базы данных, показанной на рис. А.1, с таблицами `Books` и `Authors`, а также таблицу ассоциации `BookAuthors`, которая отображает авторов на книги.

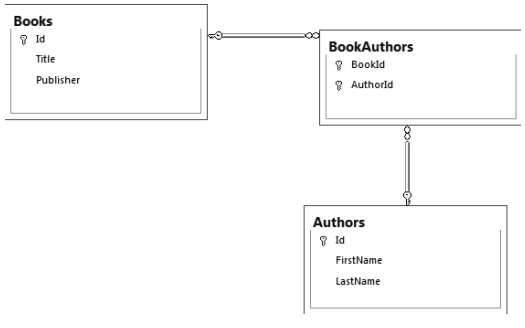

Рис. А.1. Пример схемы базы данных

Логический уровень

Логический уровень определен на языке SSDL (Store Schema Definition Language — язык определения схемы хранилища) и определяет структуру таблиц базы данных и их отношений.

Следующий код использует SSDL для описания трех таблиц: `Books`, `Authors` и `BookAuthors`. Элемент `EntityContainer` описывает все таблицы в виде элементов `EntitySet`, а ассоциации — в виде элементов `AssociationSet`. Части таблицы определяются элементом `EntityType`. В `EntityType` по имени `Books` вы можете видеть столбцы `Id`, `Title` и `Publisher`, определенные элементом `Property`. Элемент `Property` содержит атрибуты XML для определения типа данных. Элемент `Key` определяет ключ таблицы.

```
<Schema Namespace="BookEntities.Store" Alias="Self"
   ProviderManifestToken="09.00.3054"
   xmlns="http://schemas.microsoft.com/ado/2006/04/edm/ssdl">
  <EntityContainer Name="dbo">
    <EntitySet Name="Authors" EntityType="Wrox.ProCSharp.Entities.Store.Authors" />
    <EntitySet Name="BookAuthors"
      EntityType=" Wrox.ProCSharp.Entities.Store.BookAuthors" />
    <EntitySet Name="Books" EntityType=" Wrox.ProCSharp.Entities.Store.Books" />
```

```
    <AssociationSet Name="FK_BookAuthors_Authors"
        Association=" Wrox.ProCSharp.Entities.Store.FK_BookAuthors_Authors">
      <End Role="Authors" EntitySet="Authors" />
      <End Role="BookAuthors" EntitySet="BookAuthors" />
    </AssociationSet>
    <AssociationSet Name="FK_BookAuthors_Books"
        Association="BookDemoEntities.Store.FK_BookAuthors_Books">
      <End Role="Books" EntitySet="Books" />
      <End Role="BookAuthors" EntitySet="BookAuthors" />
    </AssociationSet>
  </EntityContainer>
  <EntityType Name="Authors">
    <Key> <PropertyRef Name="Id" /> </Key>
    <Property Name="Id" Type="int" Nullable="false" StoreGeneratedPattern="Identity" />
    <Property Name="FirstName" Type="nvarchar" Nullable="false" MaxLength="50" />
    <Property Name="LastName" Type="nvarchar" Nullable="false" MaxLength="50" />
  </EntityType>
  <EntityType Name="BookAuthors">
    <Key> <PropertyRef Name="BookId" /> <PropertyRef Name="AuthorId" /> </Key>
    <Property Name="BookId" Type="int" Nullable="false" />
    <Property Name="AuthorId" Type="int" Nullable="false" />
  </EntityType>
  <EntityType Name="Books">
    <Key> <PropertyRef Name="Id" /> </Key>
    <Property Name="Id" Type="int" Nullable="false" StoreGeneratedPattern="Identity" />
    <Property Name="Title" Type="nvarchar" Nullable="false" MaxLength="50" />
    <Property Name="Publisher" Type="nvarchar" Nullable="false" MaxLength="50" />
  </EntityType>
  <Association Name="FK_BookAuthors_Authors">
    <End Role="Authors"
       Type=" Wrox.ProCSharp.Entities.Store.Authors" Multiplicity="1" />
    <End Role="BookAuthors"
       Type=" Wrox.ProCSharp.Entities.Store.BookAuthors"
       Multiplicity="*" />
    <ReferentialConstraint>
    <Principal Role="Authors"> <PropertyRef Name="Id" /> </Principal>
    <Dependent Role="BookAuthors"> <PropertyRef Name="AuthorId" /> </Dependent>
    </ReferentialConstraint>
  </Association>
  <Association Name="FK_BookAuthors_Books">
    <End Role="Books" Type=" Wrox.ProCSharp.Entities.Store.Books" Multiplicity="1" />
    <End Role="BookAuthors" Type=" Wrox.ProCSharp.Entities.Store.BookAuthors"
       Multiplicity="*" />
    <ReferentialConstraint>
      <Principal Role="Books"> <PropertyRef Name="Id" /> </Principal>
      <Dependent Role="BookAuthors"> <PropertyRef Name="BookId" /> </Dependent>
    </ReferentialConstraint>
  </Association>
</Schema>
```

Концептуальный уровень

Концептуальный уровень определяет классы .NET. Этот уровень создается на языке CSDL (Conceptual Schema Definition Language — язык концептуального определения схемы).

На рис. A.2 показаны сущности `Author` и `Book`, определенные в дизайнере ADO.NET Entity Data Model Designer.

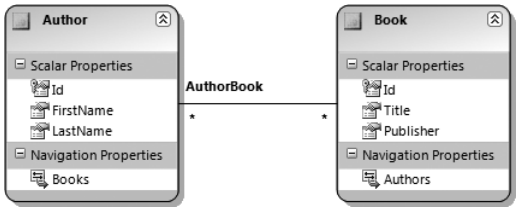

Рис. А.2. Сущности `Author` *и* `Book`

Ниже приведено содержимое CSDL, определяющее сущностные типы `Book` и `Author`. Оно было создано на основе базы данных `Books`.

```
<Schema Namespace="BookEntities" Alias="Self"
   xmlns="http://schemas.microsoft.com/ado/2006/04/edm">
  <EntityContainer Name="BookEntities">
    <EntitySet Name="Authors" EntityType="Wrox.ProCSharp.Entities.Author" />
    <EntitySet Name="Books" EntityType="Wrox.ProCSharp.Entities.Book" />
    <AssociationSet Name="BookAuthors"
       Association=" Wrox.ProCSharp.Entities.BookAuthors">
      <End Role="Authors" EntitySet="Authors" />
      <End Role="Books" EntitySet="Books" />
    </AssociationSet>
  </EntityContainer>
  <EntityType Name="Author">
    <Key>
      <PropertyRef Name="Id" />
    </Key>
    <Property Name="Id" Type="Int32" Nullable="false" />
    <Property Name="FirstName" Type="String" Nullable="false" MaxLength="50" />
    <Property Name="LastName" Type="String" Nullable="false" MaxLength="50" />
    <NavigationProperty Name="Books" Relationship="BookDemoEntities
      .BookAuthors" FromRole="Authors" ToRole="Books" />
  </EntityType>
  <EntityType Name="Book">
    <Key>
      <PropertyRef Name="Id" />
    </Key>
    <Property Name="Id" Type="Int32" Nullable="false" />
    <Property Name="Title" Type="String" Nullable="false" MaxLength="50" />
    <Property Name="Publisher" Type="String" Nullable="false" MaxLength="50" />
    <NavigationProperty Name="Authors"
       Relationship=" Wrox.ProCSharp.Entities.BookAuthors" FromRole="Books"
       ToRole="Authors" />
  </EntityType>
  <Association Name="BookAuthors">
    <End Type=" Wrox.ProCSharp.Entities.Author" Role="Authors" Multiplicity="*" />
    <End Type=" Wrox.ProCSharp.Entities.Book" Role="Books" Multiplicity="*" />
  </Association>
</Schema>
```

Сущность определена элементом `EntityType`, содержащим элементы `Key`, `Property` и `NavigationProperty` для описания свойств созданного класса. Элемент `Property` содержит атрибуты для описания имени и типа свойств .NET классов, сгенерированных дизайнером. Элемент `Assotiation` подключает типы `Author` и `Book`. Конструкция `Multiplicity="*"` означает, что `Author` (автор) может написать много `Books` (книг), а одна книга может быть написана несколькими авторами.

Уровень отображения

Уровень отображения отображает определение типа сущности с CSDL на SSDL, используя язык MSL (Mapping Specification Language — язык спецификации отображения). Следующая спецификация включает элемент `Mapping`, содержащий элемент `EntityTypeMapping` для ссылки на тип `Book` языка CSDL и определяет `MappingFragment` для ссылки на таблицу `Authors` из SSDL. `ScalarProperty` отображает свойство класса .NET с атрибутом `Name` на столбец таблицы базы данных с атрибутом `ColumnName`.

```
<Mapping Space="C-S" xmlns="urn:schemas-microsoft-com:windows:storage:mapping:CS">
  <EntityContainerMapping StorageEntityContainer="dbo"
     CdmEntityContainer="BookEntities">
    <EntitySetMapping Name="Authors">
      <EntityTypeMapping TypeName="IsTypeOf(Wrox.ProCSharp.Entities.Author)">
        <MappingFragment StoreEntitySet="Authors">
          <ScalarProperty Name="LastName" ColumnName="LastName" />
          <ScalarProperty Name="FirstName" ColumnName="FirstName" />
          <ScalarProperty Name="Id" ColumnName="Id" />
        </MappingFragment>
      </EntityTypeMapping>
    </EntitySetMapping>
    <EntitySetMapping Name="Books">
      <EntityTypeMapping TypeName="IsTypeOf(Wrox.ProCSharp.Entities.Book)">
        <MappingFragment StoreEntitySet="Books">
          <ScalarProperty Name="Publisher" ColumnName="Publisher" />
          <ScalarProperty Name="Title" ColumnName="Title" />
          <ScalarProperty Name="Id" ColumnName="Id" />
        </MappingFragment>
      </EntityTypeMapping>
    </EntitySetMapping>
    <AssociationSetMapping Name="AuthorBook"
       TypeName=" Wrox.ProCSharp.Entities.AuthorBook"
       StoreEntitySet="BookAuthors">
      <EndProperty Name="Book">
        <ScalarProperty Name="Id" ColumnName="BookId" />
      </EndProperty>
      <EndProperty Name="Author">
        <ScalarProperty Name="Id" ColumnName="AuthorId" />
      </EndProperty>
    </AssociationSetMapping>
  </EntityContainerMapping>
</Mapping>
```

Сущности

Сущностные классы, созданные дизайнером и созданные CSDL обычно наследуются от базового класса `EntityObject`, как видно на примере класса `Book`, приведенного ниже.

Этот класс наследуется от базового класса `EntityObject` и определяет свойства, которые инициируют изменение информации в средствах доступа `set`. Созданный класс `Book` является частичным классом, который может быть расширен в новом исходном файле, определяющем тот же класс в том же пространстве имен. Методы, вызываемые средством доступа `set`, такие как `OnTitleChanging()` и `OnTitleChanged()` также являются частичными, поэтому можно реализовать эти методы в специальном расширении класса. Свойство `Authors` использует класс `RelationshipManager` для возврата элементов `Book` для автора.

```
[EdmEntityTypeAttribute(NamespaceName="Wrox.ProCSharp.Entities", Name="Book")]
[DataContractAttribute()]
[Serializable()]
public partial class Book : global::System.Data.Objects.DataClasses.EntityObject
{
  public static Book CreateBook(int ID, string title, string publisher)
  {
    Book book = new Book();
    book.Id = ID;
    book.Title = title;
    book.Publisher = publisher;
    return book;
  }
  [EdmScalarPropertyAttribute(EntityKeyProperty=true, IsNullable=false)]
  [DataMemberAttribute()]
  public int Id
  {
    get
    {
    return this._Id;
    }
    set
    {
    this.OnIdChanging(value);
    this.ReportPropertyChanging("Id");
    this._Id = StructuralObject.SetValidValue(value);
    this.ReportPropertyChanged("Id");
    this.OnIdChanged();
    }
  }
  private int _Id;
  partial void OnIdChanging(int value);
  partial void OnIdChanged();
  [EdmScalarPropertyAttribute(IsNullable=false)]
  [DataMemberAttribute()]
  public string Title
  {
    get
    {
      return this._Title;
    }
    set
    {
      this.OnTitleChanging(value);
      this.ReportPropertyChanging("Title");
      this._Title = StructuralObject.SetValidValue(value, false, 50);
      this.ReportPropertyChanged("Title");
      this.OnTitleChanged();
    }
  }
  private string _Title;
  partial void OnTitleChanging(string value);
  partial void OnTitleChanged();
  [EdmScalarPropertyAttribute(IsNullable=false)]
  [DataMemberAttribute()]
  public string Publisher
  {
    get
    {
      return this._Publisher;
    }
```

```
    set
    {
      this.OnPublisherChanging(value);
      this.ReportPropertyChanging("Publisher");
      this._Publisher = StructuralObject.SetValidValue(value, false, 50);
      this.ReportPropertyChanged("Publisher");
      this.OnPublisherChanged();
    }
  }
  private string _Publisher;
  partial void OnPublisherChanging(string value);
  partial void OnPublisherChanged();
  [EdmRelationshipNavigationPropertyAttribute("BookDemoEntities", "AuthorBook",
      "Author")]
  [XmlIgnoreAttribute()]
  [SoapIgnoreAttribute()]
  [BrowsableAttribute(false)]
  public EntityCollection<Author> Authors
  {
    get
    {
      return ((IEntityWithRelationships)(this)).RelationshipManager.
           GetRelatedCollection<Author>("WroxProCSharp.Entities.AuthorBook",
           "Author");
    }
  }
}
```

Классы и интерфейсы, важные для сущностных классов, объясняются в табл. A.2. За исключением `INotifyPropertyChanging` и `INotifyPropertyChanged`, типы определены в пространстве имен `System.Data.Objects.DataClasses`.

Таблица A.2. Классы и интерфейсы, имеющие отношение к сущностным классам

Класс или интерфейс	Описание
`StructuralObject`	`StructuralObject` — базовый класс для классов `EntityObject` и `ComplexObject`. Этот класс реализует интерфейсы `INotifyPropertyChanging` и `INotifyPropertyChanged`.
`INotifyPropertyChanging` `INotifyPropertyChanged`	Эти интерфейсы определяют события `PropertyChanging` и `PropertyChanged`, позволяющие подписаться на информацию об изменении состояния объекта. В отличие от других классов и интерфейсов, перечисленных здесь, состоит в том, что эти интерфейсы определены в пространстве имен `System.ComponentModel`.
`EntityObject`	Этот класс наследуется от `StructuralObject` и реализует интерфейсы `IEntityWithKey`, `IEntityWithChangeTracker` и `IEntityWithRelationships`. `EntityObject` — часто используемый базовый класс для объектов, отображенных на таблицы базы данных, содержащих ключ и отношения с другими объектами.
`ComplexObject`	Этот класс может быть использован в качестве базового для сущностных объектов, не имеющих ключа. Он наследуется от `StructuralObject`, но не реализует других интерфейсов, как это делает класс `EntityObject`.
`IEntityWithKey`	Этот интерфейс определяет свойство `EntityKey`, обеспечивающее быстрый доступ к объекту.

Окончание табл. A.2

Класс или интерфейс	Описание
`IEntityWithChangeTracker`	Этот интерфейс определяет метод `SetChangeTracker()`, в котором может быть назначено средство отслеживания изменений, реализующее интерфейс `IChangeTracker`, чтобы получать информацию об изменении состояния от объекта.
`IEntityWithRelationships`	Этот интерфейс определяет доступное только на чтение свойство `RelationshipManager`, возвращающее объект `RelationshipManager`, который может быть использован для навигации между объектами.

Для сущностного класса не обязательно наследоваться от базового класса `EntityObject` или `ComplexObject`. Вместо этого сущностный класс может реализовать необходимые интерфейсы.

Сущностный класс `Book` может быть легко доступен посредством класса контекста объектов `BookEntities`. Свойство `Books` возвращает коллекцию объектов `Book`, по которым может быть выполнена итерация:

```
BookEntities data = new BookEntities();
foreach (var book in data.Books)
{
   Console.WriteLine("{0}, {1}", book.Title, book.Publisher);
}
```

Работающая программа запрашивает книги из базы данных и отображает их на консоли:

```
Professional C# 2008, Wrox Press
Beginning Visual C# 2008, Wrox Press
Working with Animation in Silverlight 1.0, Wrox Press
Professional WPF Programming, Wrox Press
```

Контекстный объект

Чтобы извлечь данные из базы, необходим класс `ObjectContext`. Этот класс определяет отображение сущностных объектов на базу данных. В ADO.NET вы можете сравнить этот класс с адаптером данных, заполняющим `DataSet`.

Класс `BookEntities`, созданный дизайнером, наследуется от базового класса `ObjectContext`. Этот класс добавляет конструкторы, принимающие строку соединения. В конструкторе по умолчанию строка соединения читается из конфигурационного файла. Можно также передавать конструктору уже открытое соединение в форме экземпляра `EntityConnection`. Если вы передадите конструктору соединение, которое не было открыто, контекст объектов откроет и закроет соединение; если вы передадите открытое соединение, вы также должны будете закрыть его.

Созданные классы определяют свойства `Books` и `Authors`, возвращающие `ObjectQuery`, и методы для добавления авторов и книг — `AddToAuthors()` и `AddToBooks()`.

```
public partial class BookEntities : ObjectContext
{
  public BookEntities() :
        base("name=BookEntities", "BookEntities") { }
  public BookEntities(string connectionString) :
        base(connectionString, "BookEntities") { }
```

```
  public BookEntities(EntityConnection connection) :
        base(connection, "BookEntities") { }
  [BrowsableAttribute(false)]
  public ObjectQuery<Author> Authors
  {
    get
    {
      if ((this._Authors == null))
      {
        this._Authors = base.CreateQuery<Author>("[Authors]");
      }
      return this._Authors;
    }
  }
  private ObjectQuery<Author> _Authors;
  [BrowsableAttribute(false)]
  public ObjectQuery<Book> Books
  {
    get
    {
    if ((this._Books == null))
    {
      this._Books = base.CreateQuery<Book>("[Books]");
    }
    return this._Books;
    }
  }
  private ObjectQuery<Book> _Books;
  public void AddToAuthors(Author author)
  {
    base.AddObject("Authors", author);
  }
  public void AddToBooks(Book book)
  {
    base.AddObject("Books", book);
  }
}
```

В случае, когда вы передаете строку соединения конструктору класса `BookEntities`, строка соединения типа `EntityConnection` определяет ключевое слово `Metadata`, которое требует трех вещей: списка с разделителями отображаемых файлов, `Provider` — для инвариантного имени поставщика для доступа к источнику данных, и строки соединения `Provider` для присвоения зависящей от поставщика строки соединения.

```
EntityConnection conn = new EntityConnection(
     "Metadata=./BookModel.csdl|./BookModel.ssdl|./BookModel.msl;" +
     "Provider=System.Data.SqlClient;" +
     "Provider connection string=\"Data Source=(local);" +
     "Initial Catalog=EntitiesDemo;Integrated Security=True\"");
```

Класс `ObjectContext` предлагает несколько услуг вызывающему коду.

- Отслеживает сущностные объекты, которые уже извлечены. Если объект запрашивается снова, он берется из контекста объектов.
- Хранит информацию состояния сущностей. Вы можете получить информацию о добавленных, модифицированных и удаленных объектах.
- Вы можете обновлять объекты из контекста объектов для записи изменений в лежащее в основе хранилище.

Методы и свойства класса `ObjextContext` перечислены в табл. А.3.

Таблица A.3. Методы и свойства класса `ObjextContext`

Методы и свойства `ObjectContext`	Описание
`Connection`	Возвращает объект `DbConnection`, ассоциированный с объектным контекстом.
`MetadataWorkspace`	Возвращает объект `MetadataWorkspace`, который может быть использован для чтения метаданных и информации отображения.
`QueryTimeout`	С этим свойством вы можете получать и устанавливать значение таймаута для запросов объектного контекста.
`ObjectStateManager`	Это свойство возвращает `ObjectStateManager`. Объект `ObjectStateManager` отслеживает извлеченные сущностные объекты и изменения объектов в объектном контексте.
`CreateQuery()`	Этот метод возвращает `ObjectQuery` для получения данных из хранилища. Свойства `Books` и `Authors`, показанные ранее, используют этот метод для возврата `ObjectQuery`.
`GetObjectByKey()` `TryGetObjectByKey()`	Эти методы возвращают объект по ключу — либо от диспетчера состояния объектов, либо от лежащего в основе хранилища. `GetObjectByKey()` генерирует исключение типа `ObjectNotFoundException`, если ключ не существует. `TryGetObjectByKey()` возвращает `false`.
`AddObject()`	Этот метод добавляет новый сущностный объект в объектный контекст. Этот метод вызывается методами `AddToAuthors()` и `AddToBooks()`.
`DeleteObject()`	Этот метод удаляет объект из объектного контекста.
`Detach()`	Этот метод отсоединяет сущностный объект от объектного контекста, так что его изменения с этого момента не отслеживаются.
`Attach()` `AttachTo()`	Метод `Attach()` присоединяет отсоединенный объект к хранилищу. Повторное присоединение объектов к объектному контексту требует, чтобы сущностный объект реализовывал интерфейс `IEntityWithKey`. Метод `AttachTo()` не предъявляет требования наличия ключа у объекта, но требует установки имени сущности, к которой сущностный объект должен быть присоединен.
`ApplyPropertyChanges()`	Если объект отсоединяется от объектного контекста, затем отсоединенный объект модифицируется, после чего изменения должны быть применены к объекту внутри объектного контекста, вы можете вызвать метод `ApplyPropertyChanges()` для применения изменений. Это удобно в сценарии, когда отсоединенный объект был возвращен Web-службой, изменен в клиенте и передан Web-службе в модифицированном виде.
`Refresh()`	Данные в хранилище могут измениться, пока сущностные объекты находятся внутри объектного контекста. Чтобы выполнить обновление объектов по информации хранилища, вы можете передать значение перечисления `RefreshMode`. Если значения объектов не совпадают в хранилище и объектном контексте, передача значения `ClientWins` изменяет данные в хранилище. Значение `StoreWins` изменяет данные в объектном контексте.
`SaveChanges()`	Добавление, модификация и удаление объектов из объектного контекста не изменяет объекта в лежащем в основе хранилище. Используйте метод `SaveChanges()` для сохранения изменений в хранилище.
`AcceptAllChanges()`	Этот метод изменяет состояние объектов в контексте на немодифицированное. `SaveChanges()` вызывает данный метод неявно.

Отношения

Сущностные типы `Book` и `Author` связаны между собой. Книга пишется одним или более авторами, и автор может написать одну или более книг. Отношения между ними базируются на количестве типов, с которыми они связаны, и множественности (multiplicity). Первая версия ADO.NET Entity Framework поддерживает 2 количества типов на таблицу: таблица на тип (Table per Type — TPT) и таблица на иерархию (Table per Hierarchy — TPH). Множественность может быть трех разновидностей: один к одному, один ко многим и многие ко многим.

Таблица на иерархию

При TPH существует одна таблица в базе данных, соответствующая иерархии сущностных классов. Таблица базы данных `Payments` (рис. A.3) содержит столбцы для иерархии сущностных типов. Некоторые столбцы являются общими для всех сущностей в иерархии — такие как `Id` и `Amount`. Столбец `CreditCard` используется только платежом с помощью кредитной карты.

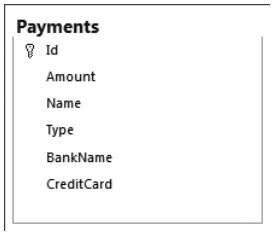

Рис. A.3. Таблица `Payments`

Сущностные классы, отображаемые на одну таблицу `Payments`, показаны на рис. A.4. `Payment` — абстрактный базовый класс, содержащий свойства, общие для всех типов в иерархии. Конкретные классы, наследуемые от `Payment` — это `CreditCardPayment`, `CashPayment` и `ChequePayment`. Класс `CreditCardPayment` в дополнение к свойствам базового класса имеет свойство `CreditCard`; класс `ChequePayment` имеет свойство `BankName`.

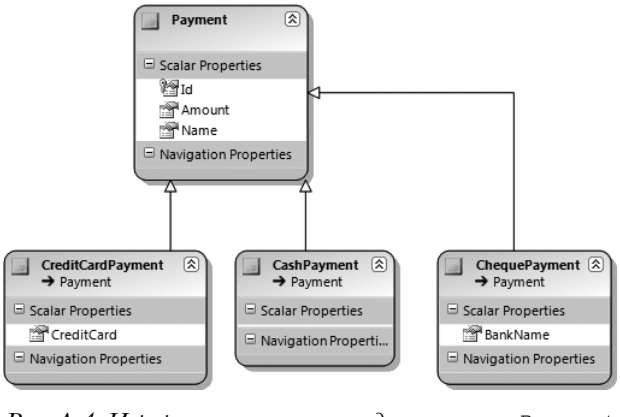

Рис. A.4. Иерархия классов, унаследованных от `Payment`

Выбор типа конкретного класса осуществляется на основе элемента `Condition`, как вы можете видеть в файле MSL. Здесь выбор типа определяется значением столбца `Type`. Возможны и другие варианты выбора типа; например, вы можете проверять, равен ли столбец `null`.

```
<Mapping Space="C-S"
  xmlns="urn:schemas-microsoft-com:windows:storage:mapping:CS">
  <EntityContainerMapping StorageEntityContainer="dbo"
     CdmEntityContainer="EntitiesDemoEntities">
    <EntitySetMapping Name="Payments">
      <EntityTypeMapping TypeName="IsTypeOf(Wrox.ProCSharp.Entities.Payment)">
        <MappingFragment StoreEntitySet="Payments">
          <ScalarProperty Name="Id" ColumnName="Id" />
          <ScalarProperty Name="Amount" ColumnName="Amount" />
          <ScalarProperty Name="Name" ColumnName="Name" />
        </MappingFragment>
      </EntityTypeMapping>
      <EntityTypeMapping
        TypeName="IsTypeOf(Wrox.ProCSharp.Entities.CashPayment)">
        <MappingFragment StoreEntitySet="Payments">
          <ScalarProperty Name="Id" ColumnName="Id" />
          <Condition ColumnName="Type" Value="CASH" />
        </MappingFragment>
      </EntityTypeMapping>
      <EntityTypeMapping
        TypeName="IsTypeOf(Wrox.ProCSharp.Entities.CreditCardPayment)">
        <MappingFragment StoreEntitySet="Payments">
          <ScalarProperty Name="Id" ColumnName="Id" />
          <ScalarProperty Name="CreditCard" ColumnName="CreditCard" />
          <Condition ColumnName="Type" Value="CREDIT" />
        </MappingFragment>
      </EntityTypeMapping>
      <EntityTypeMapping
        TypeName="IsTypeOf(Wrox.ProCSharp.Entities.ChequePayment)">
        <MappingFragment StoreEntitySet="Payments">
          <ScalarProperty Name="Id" ColumnName="Id" />
          <ScalarProperty Name="BankName" ColumnName="BankName" />
          <Condition ColumnName="Type" Value="CHEQUE" />
        </MappingFragment>
      </EntityTypeMapping>
    </EntitySetMapping>
  </EntityContainerMapping>
</Mapping>
```

Теперь можно выполнять итерацию по данным из таблицы `Payment`, при этом будут возвращаться разные типы на основе отображения:

```
PaymentEntities data = new PaymentEntities();
foreach (var p in data.Payments)
{
  Console.WriteLine("{0}, {1} - {2:C}", p.GetType().Name,
    p.Name, p.Amount);
}
```

Запуск этого приложения вернет из базы данных два объекта `CashPayment` и один объект `CreditCardPayment`:

```
CreditCardPayment, Gustav - $22.00
CashPayment, Donald - $0.50
CashPayment, Dagobert - $80,000.00
```

Таблица на тип

При TPT одна таблица отображается на один тип. База данных Northwind имеет схему с таблицами `Customers`, `Orders` и `Order Details` (рис. A.5). Таблица `Orders` отображается на таблицу `Customers` по внешнему ключу `CustomerId`; таблица `Order Details` отображается на таблицу `Orders` по внешнему ключу `OrderID`.

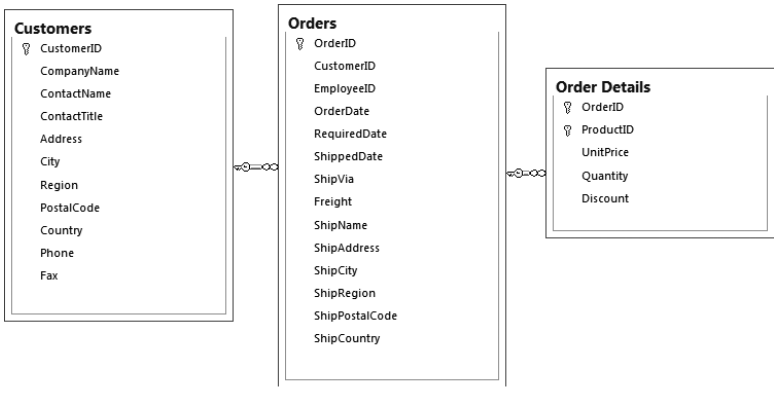

Рис. A.5. Таблицы `Customers`, `Orders` и `Order Details`

На рис. A.6 показаны сущностные типы `Customer`, `Order` и `OrderDetail`. Типы `Customer` и `Order` имеют отношение ноль или один ко многим с `Customer` и `Order`, потому что столбец `CustomerID` в таблице `Order` определен в схеме базы данных как `Nullable`.

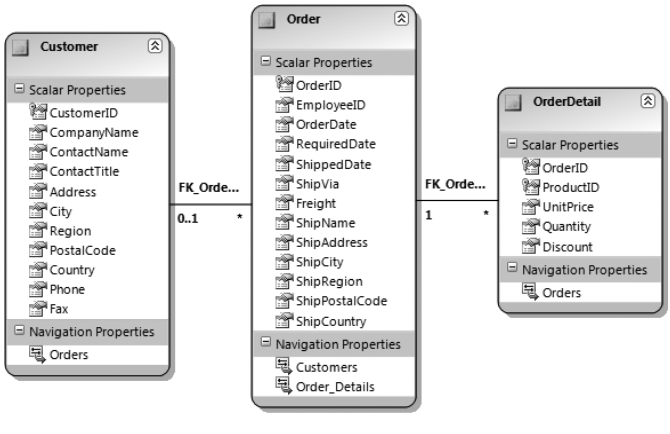

Рис. A.6. Сущностные типы `Customer`, `Order` и `OrderDetail`

Вы обращаетесь к заказчикам и их заказам за две итерации, показанные здесь. Сначала идет обращение к объектам `Customer`, и значение свойства `CompanyName` выводится на консоль. Затем идет обращение ко всем заказам через свойство `Orders` класса `Customer`. Поскольку относящиеся к заказчику заказы не загружаются в объектный контекст по умолчанию, вызывается метод `Load()` объекта `EntityCollection<Order>` на свойстве `Orders`.

```
NorthwindEntities data = new NorthwindEntities();
foreach (Customer customer in data.Customers)
{
  Console.WriteLine("{0}", customer.CompanyName);
  if (!customer.Orders.IsLoaded)
    customer.Orders.Load();
  foreach (Order order in customer.Orders)
  {
    Console.WriteLine("{0} {1:d}", order.OrderID, order.OrderDate);
  }
}
```

Для доступа к отношению "за кулисами" класс используется `RelationshipManager`. К экземпляру `RelationShipManager` можно обратиться, приведя сущностный объект к типу интерфейса `IEntityWithRelationships`. Этот интерфейс явно реализован классом `EntityObject`.

Свойство `RelationshipManager` возвращает `RelationshipManager`, ассоциированный с сущностным объектом на одном конце. Другой конец отношения определяется вызовом метода `GetRelatedCollection()`. Первый параметр `NorthwindModel.FK_Orders_Customers` — это имя отношения; второй параметр `Orders` определяет имя целевой роли.

```
RelationshipManager rm =
     ((IEntityWithRelationships)customer).RelationshipManager;
EntityCollection<Order> orders =
     rm.GetRelatedCollection<Order> (
     "NorthwindModel.FK_Orders_Customers", "Orders");
```

Загрузка отношений является отложенной. Метод `Load()` класса `EntityCollection` получает данные из хранилища. Одна из перегрузок метода `Load()` принимает перечисление `MergeOption`. Его допустимые значения перечислены в табл. А.4.

По умолчанию загрузка отношений является отложенной. Например, если вы определяете отношение между таблицами `Customers` и `Orders`, и запрашиваете заказчиков, то записи `Orders` этих заказчиков не загружаются. Срок их загрузки откладывается до того момента, когда они будут нужны. Принудительное извлечение (eager fetching) означает, что когда вы обращаетесь к записи о заказчике, его заказы также загружаются.

Таблица А.4. Значения перечисления `MergeOption`

Значение `MergeOption`	Описание
`AppendOnly`	Это значение по умолчанию. Новые сущности добавляются; имеющиеся сущности в объектном контексте не модифицируются.
`NoTracking`	`ObjectStateManager`, отслеживающий изменения в сущностных объектах, не модифицируется.
`OverwriteChanges`	Текущие значения сущностных объектов заменяются значениями из хранилища.
`PreserveChanges`	Оригинальные значения сущностных объектов в объектном контексте заменяются значениями из хранилища.

Запрос объектов

Запрашивание объектов — одна из служб, предоставляемых каркасом ADO.NET Entity Framework. Запросы могут осуществляться посредством LINQ to Entities, Entity SQL и методов Query Builder, создающих Entity SQL. LINQ to Entities рассматривается в последнем разделе настоящего приложения. Остальные два варианта рассмотрим первыми.

В следующих разделах мы будем использовать базу данных Formula 1, созданные из которой сущности вы можете видеть на рис. А.7.

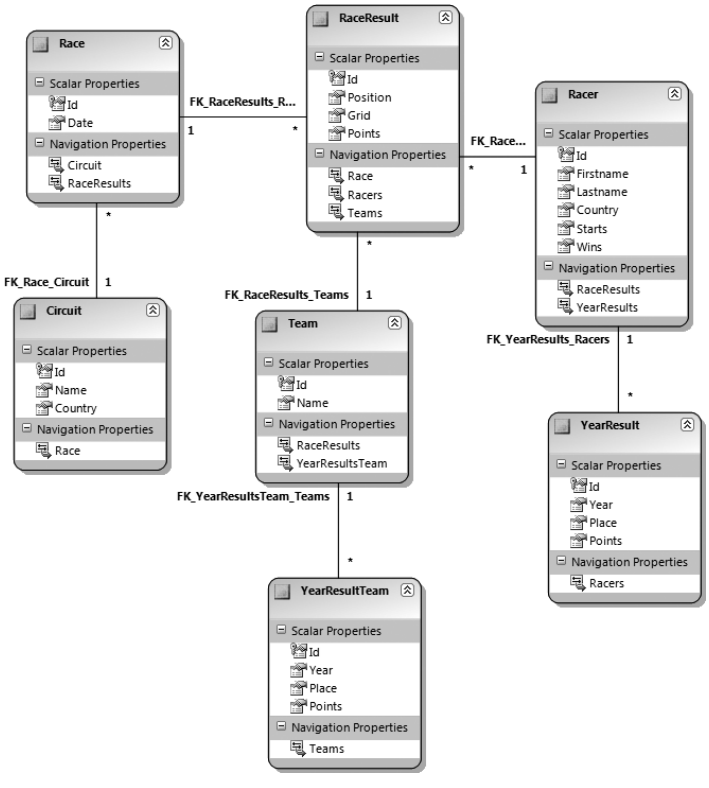

Рис. А.7. Сущности базы данных Formula 1

Запросы могут определяться посредством класса `ObjectQuery<T>`. Начнем с простого запроса для доступа ко всем сущностям `Racer`. В этом примере соединение уже открыто — передадим его объектному контексту `Formula1Entities`. Подобным образом можно извлечь сгенерированный SQL-оператор из класса `ObjectQuery<Racer>` с помощью метода `ToTraceString()`. Этот метод требует наличия открытого соединения.

```
ConnectionStringSettings connSettings =
     ConfigurationManager.ConnectionStrings["Formula1Entities"];
EntityConnection connection =
     new EntityConnection(connSettings.ConnectionString);
connection.Open();
using (Formula1Entities data = new Formula1Entities(connection))
{
  ObjectQuery<Racer> racers = data.Racers;
  Console.WriteLine(racers.CommandText);
  Console.WriteLine(racers.ToTraceString());
  connection.Close();
}
```

Оператор Entity SQL, возвращенный свойством `CommandText`, показан ниже:

```
[Racers]
```

А вот сгенерированный оператор SELECT для извлечения записей из базы данных, показанный методом ToTraceString():

```
SELECT
[Extent1].[Id] AS [Id],
[Extent1].[Firstname] AS [Firstname],
[Extent1].[Lastname] AS [Lastname],
[Extent1].[Country] AS [Country],
[Extent1].[Starts] AS [Starts],
[Extent1].[Wins] AS [Wins]
FROM [dbo].[Racers] AS [Extent1]
```

Вместо доступа к свойству Racers из объектного контекста вы можете также создать запрос методом CreateQuery():

```
ObjectQuery<Racer>racers = data.CreateQuery<Racer>("[Racers]");
```

Это подобно использованию свойства Racers, и фактически реализация свойства Racers именно таким образом создает запрос.

Теперь будет интересно отфильтровать гонщиков на основании некоторого условия. Это можно сделать с помощью метода Where() класса ObjectQuery<T>. Where() — один из методов Query Builder для создания Entity SQL. Метод требует предиката в виде строки и необязательных параметров типа ObjectParameter. Предикат, показанный здесь, специфицирует, что должны быть возвращены только гонщики из Бразилии (Brazil). it специфицирует элемент результата и страну в столбце Country. Первый параметр конструктора ObjectParameter ссылается на параметр @Country предиката, но не включает символа @.

```
string country = "Brazil";
ObjectQuery<Racer> racers = data.Racers.Where(
     "it.Country = @Country",
     new ObjectParameter("Country", country));
```

Магия состоит в том, что it становится видимым немедленно при обращении к свойству CommandText запроса. В Entity SQL конструкция SELECT VALUE it объявляет it для доступа к столбцам.

```
SELECT VALUE it
FROM (
[Racers]
) AS it
WHERE
it.Country = @Country
```

Метод ToTraceString() отображает сгенерированный SQL-оператор:

```
SELECT
[Extent1].[Id] AS [Id],
[Extent1].[Firstname] AS [Firstname],
[Extent1].[Lastname] AS [Lastname],
[Extent1].[Country] AS [Country],
[Extent1].[Starts] AS [Starts],
[Extent1].[Wins] AS [Wins]
FROM [dbo].[Racers] AS [Extent1]
WHERE [Extent1].[Country] = @Country
```

Конечно, вы можете также специфицировать полный Entity SQL:

```
string country = "Brazil";
ObjectQuery<Racer> racers = data.CreateQuery<Racer>(
   "SELECT VALUE it FROM ([Racers]) AS it WHERE it.Country = @Country",
   new ObjectParameter("Country", country));
```

Класс ObjectQuery<T> предоставляет несколько методов Query Builder, описанных в табл. A.5. Многие из этих методов очень похожи на расширяющие методы LINQ, о которых вы узнали в главе 11.

Таблица A.5. Методы Query Builder ObjectQuery<T>

Методы ObjectQuery<T>	Описание
Where()	Этот метод позволяет вам фильтровать результаты на основании условия.
Distinct()	Этот метод создает запрос с уникальными результатами.
Except()	Этот метод возвращает результат без элементов, отвечающих условию фильтра except.
GroupBy()	Этот метод создает новый запрос для группирования сущностей на основе определенного критерия.
Include()	В показанном ранее отношении, когда выполняется отложенная загрузка связанных элементов, требовался вызов метода Load() класса EntityCollection<T> для получения связанных сущностей в объектный контекст. Вместо использования метода Load() вы можете специфицировать запрос методом Include() для принудительного извлечения связанных сущностей.
OfType()	Этот метод специфицирует возврат только сущностей определенного типа. Это очень удобно для отношений TPH.
OrderBy()	Этот метод предназначен для определения порядка сортировки сущностей.
Select() SelectValue()	Эти методы возвращают проекцию результатов. Select() возвращает результирующие элементы в форме DbDataRecord; SelectValue() возвращает их в скалярном виде или в виде сложных типов, как определено обобщенным параметром TResultType.
Skip() Top()	Эти методы удобны для разбиения на страницы. Пропускайте нужное число элементов методом Skip() и берите заданное число элементов методом Top().
Intersect() Union() UnionAll()	Эти методы используются для комбинации двух запросов. Intersect() возвращает запрос, содержащий только те результаты, что доступны для обоих запросов. Union() комбинирует запросы и возвращает полный результат без дубликатов. UnionAll() также включает дубликаты.

Давайте рассмотрим один пример использования методов Query Builder. Здесь гонщики фильтруются методом Where() для возврата только тех из них, кто из США; метод OrderBy() специфицирует порядок сортировки по убыванию — сначала по количеству побед, а затем по числу стартов. И, наконец, только первые три гонщика попадают в результат посредством метода Top().

```
using (Formula1Entities data = new Formula1Entities())
{
  string country = "USA";
  ObjectQuery<Racer> racers = data.Racers.Where("it.Country = @Country",
     new ObjectParameter("Country", country))
     .OrderBy("it.Wins DESC, it.Starts DESC")
     .Top("3");
  foreach (var racer in racers)
  {
    Console.WriteLine("{0} {1}, wins: {2}, starts: {3}",
    racer.Firstname, racer.Lastname, racer.Wins, racer.Starts);
  }
}
```

Вот результат выполнения этого запроса:

```
Mario Andretti, wins: 12, starts: 128
Dan Gurney, wins: 4, starts: 87
Phil Hill, wins: 3, starts: 48
```

Обновления

Чтение, поиск и фильтрация данных из хранилища — только одна часть работы, которую обычно требуется выполнить в приложении, которое работает с данными. Запись изменений обратно в хранилище — другая часть, которую вам следует знать.

Следующие разделы будут посвящены таким темам:

- отслеживание объектов;
- информация об изменении;
- присоединение и отсоединение сущностей;
- сохранение изменений в сущностях.

Отслеживание объектов

Чтобы разрешить модификацию и сохранение данных, прочитанных из хранилища, сущности нужно отслеживать после того, как они были загружены. Это также требует, чтобы объектный контекст знал о загруженных из хранилища сущностях. Если несколько запросов обращаются к одним и тем же записям, объектный контекст должен возвращать уже загруженные сущности.

`ObjectStateManager` используется объектным контекстом для отслеживания загруженных в контекст сущностей.

Следующий пример демонстрирует, что на самом деле, если выполняются два разных запроса, которые возвращают одну и ту же запись из базы данных, диспетчер состояния знает об этом и не создает новой сущности. Вместо этого возвращается та же, что и в первый раз. Экземпляр `ObjectStateManager`, ассоциированный с объектным контекстом, может быть доступен через свойство `ObjectStateManager`. Класс `ObjectStateManager` определяет событие по имени `ObjectStateManagerChanged`, которое вызывается каждый раз, когда новый объект добавляется или удаляется из объектного контекста.

Здесь событию назначается метод `ObjectStateManager_ObjectStateManagerChanged` для получения информации об изменениях.

Два разных запроса используются для возврата сущностного объекта. Первый запрос получает первого гонщика из Австрии по фамилии Lauda. Второй запрос запрашивает гонщиков из Австрии, сортирует их по числу побед и берет первый результат. Фактически это будет один и тот же гонщик.

Чтобы убедиться в том, что в обоих случаях возвращается одна и та же сущность, используется метод `Object.ReferenceEquals()` для проверки того, что объектные ссылки действительно ссылаются на один и тот же экземпляр.

```
static void Tracking()
{
  using (Formula1Entities data = new Formula1Entities())
  {
    data.ObjectStateManager.ObjectStateManagerChanged +=
         ObjectStateManager_ObjectStateManagerChanged;
    Racer niki1 = data.Racers.Where(
         "it.Country='Austria' && it.Lastname='Lauda'").First();
```

```
    Racer niki2 = data.Racers.Where("it.Country='Austria'").
         OrderBy("it.Wins DESC").First();
    if (Object.ReferenceEquals(niki1, niki2))
    {
      Console.WriteLine("the same object");
    }
  }
}
static void ObjectStateManager_ObjectStateManagerChanged(object sender,
     CollectionChangeEventArgs e)
{
  Console.WriteLine("Object State change - action: {0}", e.Action);
  Racer r = e.Element as Racer;
  if (r != null)
  Console.WriteLine("Racer {0}", r.Lastname);
}
```

Запустив это приложение, вы можете видеть, что событие `ObjectStateManager Changed` объекта `ObjectStateManager` возникает только однажды, и ссылки `nuki1` и `niki2` указывают на один и тот же экземпляр:

```
Object state change - action: Add
Racer Lauda
The same object
```

Информация об изменении

Объектный контекст также в курсе изменений в сущностях. Следующий пример добавляет и модифицирует гонщика из объектного контекста и получает информацию об изменении. Сначала новый гонщик добавляется методом `AddToRacers()` класса `Formula1Entities`. Этот сгенерированный дизайнером метод вызывает метод `AddObject()` базового класса `ObjectContext`. Этот метод добавляет новую сущность с информацией `EntityState.Added`. Затем запрашивается гонщик по имени Alonso. В этом сущностном классе свойство `Starts` увеличивается, и потому сущность помечается информацией `EntityState.Modified`. "За кулисами" же `ObjectStateManager` информируется об изменении состояния объекта на основе реализации интерфейса `INotifyPropertyChanged`. Этот интерфейс реализован в базовом сущностном классе `StructuralObject`. Объект `ObjectStateManager` присоединен к событию `PropertyChanged`, и это событие инициируется с каждым изменением сущности.

Чтобы получить все добавленные или модифицированные сущностные объекты, вы можете вызвать метод `GetObjectStateEntries()` класса `ObjectStateManager` и передать значение перечисления `EntityState`, как это сделано здесь. Этот метод возвращает коллекцию объектов `ObjectStateEntry`, хранящую информацию о сущностях. Вспомогательный метод `DisplayState` выполняет итерацию по этой коллекции для получения детальной информации.

Вы можете также получить информацию состояния об отдельной сущности, передав `EntityKey` методу `GetObjectStateEntry()`. Свойство `EntityKey` доступно в сущностных объектах, реализующих интерфейс `IEntityWithKey`, а именно так обстоят дела с базовым классом `EntityObject`. Возвращенный объект `ObjectStateEntry` предоставляет метод `GetModifiedProperties()`, где вы можете прочесть все значения свойств, которые были изменены, а также получить доступ к исходной и текущей информации о свойствах посредством индексаторов `OriginalValues` и `CurrentValues`.

```
static void ChangeInformation()
{
  using (Formula1Entities data = new Formula1Entities())
  {
```

```
    Racer sebastien = new Racer()
    {
      Firstname = "Sébastien",
      Lastname = "Bourdais",
      Country = "France",
      Starts = 0
    };
    data.AddToRacers(sebastien);
    Racer fernando = data.Racers.Where("it.Lastname='Alonso'").First();
    fernando.Starts++;
    DisplayState(EntityState.Added.ToString(),
         data.ObjectStateManager.GetObjectStateEntries(
           EntityState.Added));
    DisplayState(EntityState.Modified.ToString(),
         data.ObjectStateManager.GetObjectStateEntries(
           EntityState.Modified));
    ObjectStateEntry stateOfFernando =
         data.ObjectStateManager.GetObjectStateEntry(fernando.EntityKey);
    Console.WriteLine("state of Fernando: {0}",
         stateOfFernando.State.ToString());
    foreach (string modifiedProp in stateOfFernando.GetModifiedProperties())
    {
      Console.WriteLine("modified: {0}", modifiedProp);
      Console.WriteLine("original: {0}",
      stateOfFernando.OriginalValues[modifiedProp]);
      Console.WriteLine("current: {0}",
      stateOfFernando.CurrentValues[modifiedProp]);
    }
  }
}
static void DisplayState(string state, IEnumerable<ObjectStateEntry> entries)
{
  foreach (var entry in entries)
  {
    Racer r = entry.Entity as Racer;
    if (r != null)
    {
      Console.WriteLine("{0}: {1}", state, r.Lastname);
    }
  }
}
```

При запуске приложения отображаются добавленные и модифицированные гонщики, а также измененные свойства вместе с их исходными и текущими значениями:

```
Added: Bourdais
Modified: Alonso
state of Fernando: Modified
modified: Starts
original: 95
current: 96
```

Присоединение и отсоединение сущностей

Возвращая данные сущности вызывающему коду, может быть важно отсоединить объекты от объектного контекста. Это может понадобиться, например, если сущностный объект возвращается Web-службой. Здесь, если сущностный объект изменяется на клиенте, объектный контекст не знает об изменении.

В коде примера метод `Detach()` объекта `ObjectContext` отсоединяет сущность по имени `fernando` и потому объектный контекст не знает об изменениях, выполненных

в этой сущности. Если измененный сущностный объект передается службе от клиентского приложения, он может быть вновь присоединен. Но простого присоединения его к объектному контексту может быть недостаточно, потому что это не даст информации о том, что объект был изменен. Вместо этого оригинальный объект должен быть доступен в объектном контексте. Оригинальный объект можно получить из хранилища, используя ключ с методом `GetObjectByKey()`. Если сущностный объект уже находится внутри объектного контекста, то он используется; в противном случае он заново извлекается из базы данных. Вызов метода `ApplyPropertyChanges()` принимает модифицированный сущностный объект в объектный контекст, и если в нем есть изменения, они проводятся внутри существующего сущностного объекта в объектном контексте с тем же ключом внутри объектного контекста, а `EntityState` устанавливается в `Modified`. Помните, что метод `ApplyPropertyChanges()` требует наличия объекта в объектном контексте; иначе будет добавлен новый сущностный объект с `EntityState`, равным `Added`.

```
using (Formula1Entities data = new Formula1Entities())
{
  data.ObjectStateManager.ObjectStateManagerChanged +=
       ObjectStateManager_ObjectStateManagerChanged;
  ObjectResult<Racer> racers = data.Racers.Where("it.Lastname='Alonso'");
  Racer fernando = racers.First();
  EntityKey key = fernando.EntityKey;
  data.Detach(fernando);
  // Теперь гонщик отсоединен и может изменяться
  // независимо от объектного контекста.
  fernando.Starts++;
  Racer originalObject = (Racer)data.GetObjectByKey(key);
  data.ApplyPropertyChanges(key.EntitySetName, fernando);
}
```

Сохранение изменений в сущностях

На основе всей информации об изменениях, с помощью `ObjectStateManager` все добавленные, удаленные и модифицированные сущностные объекты могут быть записаны в хранилище посредством метода `SaveChanges()` класса `ObjectContext`. Чтобы сверить изменения с объектным контекстом, вы можете присвоить метод-обработчик событию `SaveChanges` класса `ObjectContext`. Это событие инициируется перед записью данных в хранилище, так что вы можете добавить некоторую логику верификации, чтобы проверить, действительно ли должны быть выполнены изменения. `SaveChanges()` возвращает количество сущностных объектов, которые были записаны.

Но что случится, если записи в базе данных, представленные сущностными классами, изменятся после чтения записи? Ответ зависит от свойства `ConcurrencyMode`, которое устанавливается в модели. Для каждого свойства сущностного объекта вы можете установить `ConcurrencyMode` равным `Fixed` или `None`. Значение `Fixed` означает, что свойство верифицируется во время записи для определения того, не было ли оно изменено между тем. Значение `None` — принятое по умолчанию — игнорирует любые изменения. Если некоторые свойства конфигурируются в режиме `Fixed`, и данные изменяются между чтением и записью сущностных объектов, генерируется исключение `OptimisticConcurrencyException`. Вы можете обработать это исключение, вызвав метод `Refresh()` для того, чтобы перечитать актуальную информацию из базы данных в объектный контекст. Этот метод принимает два режима обновления, заданные значением перечисления `RefreshMode`: `ClientWins` или `StoreWins`. Значение `StoreWins` означает, что актуальная информация берется из базы данных и устанавливается в текущие значения сущностных объектов. `ClientWins` означает,

что информация базы данных устанавливается в исходные значения сущностных объектов, и потому значения базы данных перезаписываются при следующем вызове SaveChanges. Второй параметр метода Refresh() — это либо коллекция сущностных объектов, либо единственный сущностный объект. Вы можете изменять поведение обновления от объекта к объекту.

```
static void ChangeInformation()
{
  //...
    int changes = 0;
    try
    {
      changes = data.SaveChanges();
    }
    catch (OptimisticConcurrencyException ex)
    {
      data.Refresh(RefreshMode.ClientWins, ex.StateEntries);
      changes = data.SaveChanges();
    }
    Console.WriteLine("{0} entities changed", changes);
```

LINQ to Entities

В нескольких главах нашей книги вы встречались с LINQ для запроса информации из объектов, баз данных и XML. Конечно, LINQ также доступен и для запросов сущностей.

В LINQ to Entities источником для запроса LINQ служит ObjectQuery<T>. Поскольку ObjectQuery<T> реализует интерфейс IQueryable, выбранные расширяющие методы для запроса определены в классе Queryable из пространства имен System.Linq. Расширяющие методы, определенные этим классом, имеют параметр Expression<T>; вот почему компилятор пишет дерево выражений в сборку. Вы можете прочесть больше о деревьях выражений в главе 11. Дерево выражений затем преобразуется из класса ObjectQuery<T> в запрос SQL.

Вы можете использовать простой запрос LINQ, как показано здесь, чтобы вернуть гонщиков, выигравших более 40 гонок:

```
using (Formula1Entities data = new Formula1Entities())
{
  var racers = from r in data.Racers
               where r.Wins > 40
               orderby r.Wins descending
               select r;
  foreach (Racer r in racers)
  {
    Console.WriteLine("{0} {1}", r.Firstname, r.Lastname);
  }
}
```

Вот результат обращения к базе данных Formula 1:

```
Michael Schumacher
Alain Prost
Ayrton Senna
```

Вы можете также определить запрос LINQ для доступа к отношениям, как было показано ранее. Переменная r ссылается на гонщиков, а переменная rr — на результаты гонок. В конструкции where определен фильтр для извлечения только гонщиков из Швейцарии (Switzerland), которые заняли определенное место на подиуме.

Чтобы получить информацию о местах, занятых на подиуме, результат группируется, и вычисляется соответствующий счетчик. Сортировка выполняется на основе занятых мест.

```
using (Formula1Entities data = new Formula1Entities())
{
  var query = from r in data.Racers
              from rr in r.RaceResults
              where rr.Position <= 3 && rr.Position >= 1 &&
                  r.Country == "Switzerland"
              group r by r.Id into g
              let podium = g.Count()
              orderby podium descending
              select new { Racer = g.FirstOrDefault(), Podiums = podium };
  foreach (var r in query)
  {
    Console.WriteLine("{0} {1} {2}", r.Racer.Firstname,
    r.Racer.Lastname,r.Podiums);
  }
}
```

Запущенное приложение вернет имена трех гонщиков из Швейцарии:

```
Clay Regazzoni 28
Jo Siffert 6
Rudi Fischer 2
```

Резюме

В этой главе вы ознакомились со средствами ADO.NET Entity Framework. В отличие от LINQ to SQL, описанного в главе 27, этот каркас предлагает отображение на основе поставщиков, а значит, другие разработчики баз данных могут реализовывать свои собственные поставщики.

ADO.NET Entity Framework основан на отображении, определенном CSDL, MSL и SSDL — информации XML, описывающей сущности, отображение и схему базы данных. Используя эту технику отображения, вы можете создавать различные типы отношений, чтобы отображать сущностные классы на таблицы базы данных.

Вы увидели, как объектный контекст сохраняет знание об извлеченных и обновленных сущностях, и как изменения могут быть записаны в хранилище.

LINQ to Entities — это лишь одна грань ADO.NET Entity Framework, которая позволяет вам использовать новый синтаксис запросов для обращения к сущностям.