[Главная](http://www.sergechel.info/ru)

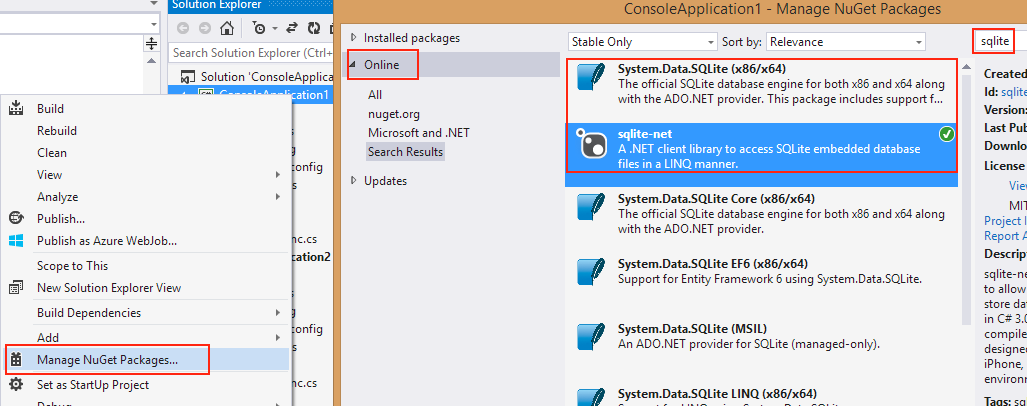
[Использование SQLite в C#: Часть 4 – Проект SQLite.Net](http://www.sergechel.info/ru/content/using-sqllite-with-c-sharp-part-4-sqlite-net)

ср, 2015-05-20 22:04 | SergeChel

**SQLite.Net** – это любительский проект с открытым программным кодом, ставящий своей целью реализацию работы с движком локально хранимых баз данных SQLite посредством обработки данных на уровне их объектных моделей. В идеологии SQLite.Net каждая модель данных – это отдельный объект, со своими свойствами и методами, низкоуровневую же часть работы с данными SQLite.Net берет на себя.

Логика работы SQLite.Net реализована в файлах с исходным кодом *SQLite.cs* и *SQLiteAsync.cs*, которые добавляются непосредственно в состав разрабатываемого проекта. Библиотеки движка SQLite также должны быть включены в состав проекта, либо дистрибутив SQLite должен быть установлен в целевой системе.

Простейший способ включить файлы SQLite.Net в состав своего проекта – воспользоваться *менеджером пакетов NuGet*, входящим в состав Visual Studio. Для этого необходимо открыть контекстное меню проекта в *Solution Explorer*, выбрать пункт *Manage NuGet Packages…*, далее в открывшемся окне NuGet переключиться в раздел *Online/All* и выполнить поиск по ключевому слову *sqlite*. Из результатов поиска вам необходимо выбрать и установить пакет *sqlite-net*, а также пакет *System.Data.SQLite (x86/x64)* в том случае, если вы ходите сразу добавить библиотеки SQLite в свой проект.

[](http://www.sergechel.info/sites/default/files/pubs/sqlite/sqlite-net-add.png)

Также можно скачать файлы *SQLite.cs* и *SQLiteAsync.cs* с ресурса <https://github.com/praeclarum/sqlite-net> и вручную добавить в состав разрабатываемого проекта.

Работа с данными

Первым делом, необходимо добавить в начало программного кода своего проекта директиву:

using SQLite;

Центральным классом SQLite.Net является класс *SQLiteConnection*. Для создания нового объекта этого класса используется конструктор с несколькими параметрами:

var db = new SQLiteConnection("filename.db", true);

// do your work here

db.Dispose();

По окончании работы желательно вызывать метод *Dispose()*. На случай исключительных ситуаций необходимо использовать конструкции *try…catch…finally* для обеспечения обязательного вызова этого метода. Если есть возможность, то лучше всего использовать оператор *using()*:

using(var db = new SQLiteConnection("filename.db", true))

{

// do your work here

}

В качестве входящего параметра при создании объекта класса SQLiteConnection передается имя файла базы данных, а также способ хранения даты и времени *storeDateTimeAsTicks*. Если в данном параметре передано значение *true*, то дата и время будут храниться в виде тиков, если передано значение *false* или параметр не указан, то значение будут храниться в текстовом виде.

Также в конструкторе можно задать набор опций *SQLiteOpenFlags*, наиболее интересными из которых являются следующие:

* *ReadOnly* – только чтение данных;
* *ReadWrite* – чтение/запись данных;
* *Create* – если файл с указанным именем отсутствует, то создавать новый файл данных.

По умолчанию конструктор использует опции *SQLiteOpenFlags.ReadWrite | SQLiteOpenFlags.Create*. Таким образом, если необходимо открыть файл с данными только для чтения, то можно использовать конструктор:

var db = new SQLiteConnection("filename.db", SQLiteOpenFlags.ReadOnly, true);

Создание структуры базы данных

Для наглядности создадим следующий класс:

class Person

{

[PrimaryKey, AutoIncrement, Unique]

public int Id { get; set; }

[MaxLength(30), NotNull]

public string FirstName { get; set; }

[MaxLength(30), NotNull]

public string LastName { get; set; }

[NotNull]

public DateTime BirthDate { get; set; }

[Ignore]

public string FullName {

get

{

return string.Format(

"{0} {1}",

LastName,

FirstName

);

}

}

public override string ToString()

{

return string.Format(

"{0}: {1} {2}",

Id,

FullName,

BirthDate.ToString("dd-MM-yyyy")

);

}

}

Свойства класса могут содержать предопределенные атрибуты, которые являются инструкциями для создания структуры таблицы в будущей базе данных. Наиболее существенными атрибутами являются:

* *[PrimaryKey]* – целочисленное значение будет использоваться для хранения значений первичного ключа;
* *[AutoIncrement]* – целочисленное значение будет автоматически увеличиваться при добавлении новых записей в таблицу;
* *[Unique]* – для данных значений будет выполняться контроль уникальности;
* *[MaxLength]* – задает ограничение для максимальной длины сохраняемого текстового значения;
* *[Indexed]* – значения в данной колонке будут проиндексированы;
* *[NotNull]* – для значений данной колонки будет выполняться контроль на уникальность;
* *[Ignore]* – свойство класса с данным атрибутом будет игнорироваться при работе с базой данных.

Для создания новой таблицы в базе данных используется метод *CreateTable()* класса SQLiteConnection, например:

db.CreateTable<Person>();

или

db.CreateTable(typeof(Person));

Для уничтожения таблицы используется метод *DropTable()*:

db.DropTable<Person>();

Добавление данных

Для добавления данных в таблицу используется метод *Insert()* класса SQLiteConnection:

var person = new Person {

FirstName = "John",

LastName = "Doe",

BirthDate = DateTime.Parse("01.12.1980")

};

db.Insert(person);

Метод принимает в качестве входящего параметра непосредственно добавляемый объект, а возвращает автоматически созданное значение первичного ключа.

Также можно добавить сразу набор записей, передав какой-либо набор данных с интерфейсом *IEnumerable*, например, *List*:

var people = new List<Person>();

people.Add(new Person {

FirstName = "Elena",

LastName = "Petrova",

BirthDate = DateTime.Parse("11.06.1991")

});

db.InsertAll(people);

В результате выполнения данного метода возвращается количество добавленных записей.

Также стоит отметить метод *InsertOrReplace()*. Этот метод при добавлении нового объекта в базу данных проверяет наличие совпадений его свойств, помеченных атрибутом *[Unique]*, с уже существующими записями и, в случае обнаружения совпадений, удаляет их. В результате своего выполнения метод возвращает количество замененных таким образом объектов.

Изменение данных

Для обновления данных используется метод *Update()* класса SQLiteConnection. В качестве параметра в него передается обновляемый объект:

db.Update(person);

По аналогии с внесением данных, также можно обновить сразу несколько объектов, передав их список в метод *UpdateAll()*:

db.UpdateAll(people);

Удаление данных

Удалить запись из таблицы можно вызовом метода *Delete<>()* класса SQLiteConnection, в качестве параметра передается идентификатор удаляемого объекта:

db.Delete<Person>(onePerson.Id);

Полностью удалить данные из таблицы можно методом *DeleteAll<>()*:

db.DeleteAll<Person>();

Работа с данными

Простейший способ считать отдельную запись из базы данных – использовать метод *Get<>()* класса SQLiteConnection:

var person = db.Get<Person>(0);

В качестве входящего параметра метод принимает идентификатор записи, в результате своего выполнения возвращает данные сразу в виде объекта.

Для более продвинутой работы с данными предназначен класс *TableQuery*, возвращаемый методом *Table<>()* класса SQLiteConnection:

var persons = db.Table<Person>();

Класс *TableQuery* реализует синтерфейс *IEnumerable*, следовательно мы может считать данные последовательным перебором:

var persons = db.Table<Person>();

foreach (var person in persons)

{

Console.WriteLine(person.ToString());

}

В классе *TableQuery* реализовано довольно много полезных методов для работы с данными, полный обзор которых выходит за рамки данной статьи, однако можно отметить ряд наиболее полезных.

Получить количество записей в таблице можно с помощью метода *Count()*:

int count = db.Table<Person>().Count();

Проверить наличие объекта в таблице можно с помощью метода *Contains<>()*:

bool isThere = db.Table<Person>().Contains<Person>(person);

Получить значение по идентификатору можно с помощью метода *ElementAt<>()*:

var person = db.Table<Person>().ElementAt<Person>(0);

Класс *TableQuery* также дает возможность получать данные с помощью запросов LINQ:

var p = from s in conn.Table<Person>()

where s.LastName.StartsWith("D")

select s;

Для того чтобы получить данные с помощью SQL-запросов, можно использовать метод *Query<>()* класса SQLiteConnection:

var persons = db.Query<Person>("SELECT \* FROM Person");

В результате выполнения запроса будет возвращен список *List<>* с выбранными данными, уже преобразованными в объекты.

Если SQL-запрос предполагает возвращение единственного значения, то можно использовать метод *ExecuteScalar<>()*:

var person = db.ExecuteScalar<Person>("SELECT \* FROM Person WHERE Id = 0");

int count = db.ExecuteScalar<int>("SELECT count(\*) FROM Person WHERE Id < 3");

Для выполнения SQL-команд, не предполагающих возврата каких-либо данных можно использовать метод *Execute()*:

db.Execute("DROP TABLE Person");

db.Execute("DELETE FROM Person WHERE Id = ?", new object[1] {0});

Транзакции

Для реализации транзакционности в простейшем случае рекомендуется использовать следующую конструкцию:

db.RunInTransaction(() =>

{

// your code here

});

Ваш код внутри этой обертки будет выполняться в рамках транзакции, в случае возникновения исключительной ситуации будет произведен откат, а по факту успешного выполнения кода транзакция будет зафиксирована.

Если же необходимо более гибкое решение, то SQLite.Net предоставляет ряд методов для реализации данной задачи. Начало транзакции запускается вызовом метода *BeginTransaction()*, а завершение транзакции методом *Commit()*. Откат транзакции выполняется с помощью метода *Rollback()*:

В процессе выполнения транзакции можно фиксировать промежуточные точки методом *SaveTransactionPoint()*:

string saved = db.SaveTransactionPoint();

С помощью метода *RollbackTo(saved)* можно откатить транзакцию до заданной точки, а с помощью метода *Release(saved)* – зафиксировать транзакцию до указанной точки.

Заключение

**SQLite.Net** позволяет разработчику быстро и комфортно реализовать работу с локально хранимыми данными за счет абстрагирования от низкоуровневой части реализации данной задачи. Данный подход может быть полезным для реализации небольших приложений, в частности мобильных, где нет нужды в сложных объектных моделях данных, а также не требуется обработка очень больших объемов данных. В противном случае лучше все же собраться с силами и работать напрямую с несравненно более богатым и гибким функционалом SQLite. Нужно четко осознать, что SQLite.Net не творит что-то чудесное внутри себя, а просто пытается автоматически формировать необходимые SQL-команды для работы с данными, при этом в сложных случаях движок вполне может принять решение загрузить в приложение ВСЕ данные из целевой таблицы и работать с ними уже внутри приложения.