DmitryNBoyko 10 февраля 2020 в 12:14

Ускоряемся в Entity Framework Core

.NET, C#

Из песочницы

Не будь жадиной!

При выборке данных выбирать нужно ровно столько сколько нужно за один раз. Никогда не извлекайте все данные из таблицы!

Неправильно:

```
using var ctx = new EFCoreTestContext(optionsBuilder.Options);
// Мы возвращаем колонку ID с сервера, но никогда не используем и это неправильно!
ctx.FederalDistricts.Select(x=> new { x.ID, x.Name, x.ShortName }).ToList();
```

Правильно:

```
using var ctx = new EFCoreTestContext(optionsBuilder.Options);
// Мы не возвращаем колонку ID с сервера и это правильно!
ctx.FederalDistricts.Select(x=> new { x.Name, x.ShortName }).ToList();
ctx.FederalDistricts.Select(x => new MyClass { Name = x.Name, ShortName = x.ShortName }).ToList();
```

Неправильно:

```
var blogs = context.Blog.ToList(); // Тут вы скопировали ВСЮ таблицу в память. Зачем?
// Чтобы выбрать лишь некоторые записи?
var somePost = blogs.FirstOrDefault(x=>x.Title.StartWidth(“Hello world!”));
```

Правильно:

```
var somePost = context.Blog.FirstOrDefault(x=>x.Title.StartWidth(“Hello world!”));
```

Встроенная проверка данных может быть выполнена, когда запрос вернул какие-то записи.

Неправильно:

```
var blogs = context.Blogs.Where(blog => StandardizeUrl(blog.Url).Contains("dotnet")).ToList();

public static string StandardizeUrl(string url)
{
    url = url.ToLower();
    if (!url.StartsWith("http://"))
    {
        url = string.Concat("http://", url);
    }
    return url;
}
```

Правильно:

```
var blogs = context.Blogs.AsEnumerable().Where(blog => StandardizeUrl(blog.Url).Contains("dotnet")).ToList();
```

```
//Еще правильней так
var blogs = context.Blogs.Where(blog => blog.Contains("dotnet"))
    .OrderByDescending(blog => blog.Rating)
    .Select(blog => new
    {
        Id = blog.BlogId,
        Url = StandardizeUrl(blog.Url)
    })
    .ToList();
```

Вау, вау, вау, разогнался.

Самое время немного освежить знания по методам LINQ.

Давайте рассмотрим отличия между *ToList AsEnumerable* AsQueryable

*И*так, *ToList*

- Выполняет запрос немедленно.
- Используйте .*ToList()* для форсирования получения данных и выхода из режима поздней загрузки (lazy loading), так что этот метод полезен перед тем как вы пройдетесь по данным.

AsEnumerable

- Выполнение с задержкой (lazy loading)
- Принимает параметр: *Func <TSource, bool>*
- Загружает каждую запись в память приложения и управляет фильтрует его (в том числе **Where/Take/Skip** приведут к тому, что, например запрос select * from Table1,
- загрузит результирующий набор в память, затем выберет первые N элементов)
- В этом случает отрабатывает схема: *Linq-to-SQL + Linq-to-Object*.
- Используйте *IEnumerable* для получения списка из базы данных в режиме поздней загрузки (lazy loading).

AsQueryable

- Выполнение с задержкой (lazy loading)
- Может быть перезагружен:

```
AsQueryable(IEnumerable) или AsQueryable<TElement>(IEnumerable<TElement>)
```

- Преобразует Expression в T-SQL (с учетом специфики провайдера), удаленное исполняет запрос и возвращает результат в память приложения.
- Вот почему DbSet (в Entity Framework) также наследуется от AsQueryable чтобы получать эффективные запросы.
- Не загружает каждую запись, например если **Take(5)** это сгенерирует запрос вида **«select top 5 * SQL»** в фоновом режиме. Это означает, что этот подход более дружественный для SQL базы данных, и дает более скоростной результат.Так что *AsQueryable()* обычно работает быстрее, чем *AsEnumerable()* так как сначала генерирует T-SQL включающий в себя все условия Linq определённые вами.
- *И*спользуйте *AsQueryable* если хотите запрос к базе данных который может быть улучшен перед запуском на стороне сервера.

Пример использования AsQueryable в простейшеем случае:

```
public IEnumerable<EmailView> GetEmails(out int totalRecords, Guid? deviceWorkGroupID,
                DateTime? timeStart, DateTime? timeEnd, string search, int? confirmStateID, int? stateTypeID, int? limitOf
fset, int? limitRowCount, string orderBy, bool desc)
        {
            var r = new List<EmailView>();

            using (var db = new GJobEntities())
            {
                var query = db.Emails.AsQueryable();

                if (timeStart != null && timeEnd != null)
                {
                    query = query.Where(p => p.Created >= timeStart && p.Created <= timeEnd);
                }

                if (stateTypeID != null && stateTypeID > -1)
                {
                    query = query.Where(p => p.EmailStates.OrderByDescending(x => x.AtTime).FirstOrDefault().EmailStateTyp
eID == stateTypeID);
                }


                if (confirmStateID != null && confirmStateID > -1)
                {
                    var boolValue = confirmStateID == 1 ? true : false;
                    query = query.Where(p => p.IsConfirmed == boolValue);
                }

                if (!string.IsNullOrEmpty(search))
                {
                    search = search.ToLower();
                    query = query.Where(p => (p.Subject + " " + p.CopiesEmails + " " + p.ToEmails + " " + p.FromEmail + "
 " + p.Body)
                                             .ToLower().Contains(search));
                }

                if (deviceWorkGroupID != Guid.Empty)
                {
                    query = query.Where(x => x.SCEmails.FirstOrDefault().SupportCall.Device.DeviceWorkGroupDevices.FirstOr
Default(p => p.DeviceWorkGroupID == deviceWorkGroupID) != null);
                }

                totalRecords = query.Count();
                query = query.OrderByDescending(p => p.Created);
                if (limitOffset.HasValue)
                {
                    query = query.Skip(limitOffset.Value).Take(limitRowCount.Value);
                }
                var items = query.ToList(); // Получаем все отфильтрованные записи

                foreach (var item in items)
                {
                    var n = new EmailView
                    {
                        ID = item.ID,
                        SentTime = item.SentTime,
                        IsConfirmed = item.IsConfirmed,
                        Number = item.Number,
                        Subject = item.Subject,
                        IsDeleted = item.IsDeleted,
                        ToEmails = item.ToEmails,
                        Created = item.Created,
                        CopiesEmails = item.CopiesEmails,
                        FromEmail = item.FromEmail,
                    };

                    // Другой код для заполнения класса-представления

                    r.Add(n);
                }
            }
```

```
            return r;
        }
```

Волшебство простого чтения

Если вам не нужно менять данные, только отобразить используйте *.AsNoTracking()* метод.

Медленная выборка

```
var blogs = context.Blogs.ToList();
```

Быстрая выборка (только на чтение)

```
var blogs = context.Blogs.AsNoTracking().ToList();
```

Чувствую, вы немного уже размялись?

Типы загрузки связанных данных

Для тех, кто забыл, что такое *lazy loading*.

Ленивая загрузка *(Lazy loading)* означает, что связанные данные прозрачно загружаются из базы данных при обращении к свойству навигации. Подробнее читаем тут .

И заодно, напомню о других типах загрузки связанных данных.

Активная загрузка *(Eager loading)* означает, что связанные данные загружаются из базы данных как часть первоначального запроса.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
            .ThenInclude(post => post.Author)
                .ThenInclude(author => author.Photo)
        .Include(blog => blog.Owner)
            .ThenInclude(owner => owner.Photo)
        .ToList();
}
```

Внимание! Начиная с версии EF Core 3.0.0, каждое Include будет вызывать добавление дополнительного *JOIN* к запросам SQL, создаваемым реляционными поставщиками, тогда как предыдущие версии генерировали дополнительные запросы SQL. Это может значительно изменить производительность ваших запросов, в лучшую или в худшую сторону. В частности, запросы LINQ с чрезвычайно большим числом операторов включения могут быть разбиты на несколько отдельных запросов LINQ.

Явная загрузка *(Explicit loading)* означает, что связанные данные явно загружаются из базы данных позднее.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    var goodPosts = context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Where(p => p.Rating > 3)
        .ToList();
}
```

Рывок и прорыв! Двигаемся дальше?

Готовы ускориться еще больше?

Чтобы резко ускориться при выборке сложно структурированных и даже ненормализованных данных из реляционной базы данных есть два способа сделать это: используйте индексированные представления (1) или что еще лучше – предварительно подготовленные(вычисленные) данные в простой плоской форме для отображения (2).

(1) **Индексированное представление** в контексте MS SQL Server

Индексированное представление имеет уникальный кластеризованный индекс. Уникальный кластерный индекс хранится в SQL Server и обновляется, как и любой другой кластерный индекс. Индексированное представление является более значительным по сравнению со стандартными представлениями, которые включают сложную обработку большого количества строк, например, агрегирование большого количества данных или объединение множества строк.

Если на такие представления часто ссылаются в запросах, мы можем повысить производительность, создав уникальный кластеризованный индекс для представления. Для стандартного представления набор результатов не сохраняется в базе данных, вместо этого набор результатов вычисляется для каждого запроса, но в случае кластеризованного индекса набор результатов сохраняется в базе данных точно так же, как таблица с кластеризованным индексом. Запросы, которые специально не используют индексированное представление, могут даже выиграть от существования кластеризованного индекса из представления.

Представление индекса имеет определенную стоимость в виде производительности. Если мы создаем индексированное представление, каждый раз, когда мы изменяем данные в базовых таблицах, SQL Server должен поддерживать не только записи индекса в этих таблицах, но также и записи индекса в представлении. В редакциях SQL Server для разработчиков и предприятий оптимизатор может использовать индексы представлений для оптимизации запросов, которые не указывают индексированное представление. Однако в других выпусках SQL Server запрос должен включать индексированное представление и указывать подсказку NOEXPAND, чтобы получить преимущество от индекса в представлении.

(2) Если нужно сделать запрос, требующий отображения более трех уровней связанных таблиц в количестве три и более с повышенной **CRUD** нагрузкой, лучшим способом будет задуматься о том, чтобы периодически вычислять результирующий набор, сохранять его в таблице и использовать для отображения. Результирующая таблица, в которой будут сохраняться данные должна иметь **Primary Key и индексы по полям поиска в LINQ**.

Что насчет асинхронности?

Да! Используем ее где только можно! Вот пример:

```
public void Do()
{
    var myTask = GetFederalDistrictsAsync ();
    foreach (var item in myTask.Result)
    {
        //Ваш код
    }
}

public async Task<List<FederalDistrict>> GetFederalDistrictsAsync()
{
    var conn = configurationRoot.GetConnectionString("EFCoreTestContext");
    optionsBuilder.UseSqlServer(conn);
    using var context = new EFCoreTestContext(optionsBuilder.Options);
    return await context.FederalDistricts.ToListAsync();
}
```

И да, ничего не забыли для повышения производительности? Бууум!

```
return await context.FederalDistricts.<b>AsNoTracking()</b>.ToListAsync();
```

Внимание: метод **Do()** добавлен для демонстрационных целей только, с целью указать работоспособность метода

GetFederalDistrictsAsync(). Как правильно заметили мои коллеги тутнужен другой пример чистой асинхронности.

И давайте я его приведу на основе понятия компонента представления в ASP .NET Core:

```
// Класс компонента
public class PopularPosts : ViewComponent
    {
        private readonly IStatsRepository _statsRepository;

        public PopularPosts(IStatsRepository statsRepository)
        {
            _statsRepository = statsRepository;
        }

        public async Task<IViewComponentResult> InvokeAsync()
        {
           // Вызов нашего метода без изменений из выделенного репозитория бизнес-логики
            var federalDistricts = await _statsRepository.GetFederalDistrictsAsync();
            var model = new TablePageModel()
            {
                FederalDistricts = federalDistricts,
            };

            return View(model);
        }
    }
    // Далее

    /// <summary>
    /// Интерфейс бизнес-логики для получения хммм.... чего-либо
    /// </summary>
    public interface IStatsRepository
    {
        /// <summary>
        /// Получение списка федеральных округов и их субъектов федерации
        /// </summary>
        /// <returns></returns>
        IEnumerable<FederalDistrict> FederalDistricts();

        /// <summary>
        /// Получение списка федеральных округов и их субъектов федерации
        /// Асинхронно!!!
        /// </summary>
        /// <returns></returns>
        Task<List<FederalDistrict>> GetFederalDistrictsAsync();
    }

    /// <summary>
    /// Бизнес-логика для получения хммм.... чего-либо
    /// </summary>
    public class StatsRepository : IStatsRepository
    {
        private readonly DbContextOptionsBuilder<EFCoreTestContext>
            optionsBuilder = new DbContextOptionsBuilder<EFCoreTestContext>();
        private readonly IConfigurationRoot configurationRoot;

        public StatsRepository()
        {
            IConfigurationBuilder configurationBuilder = new ConfigurationBuilder()
                    .SetBasePath(Environment.CurrentDirectory)
                    .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true);
            configurationRoot = configurationBuilder.Build();
        }

        public async Task<List<FederalDistrict>> GetFederalDistrictsAsync()
        {
            var conn = configurationRoot.GetConnectionString("EFCoreTestContext");
            optionsBuilder.UseSqlServer(conn);
            using var context = new EFCoreTestContext(optionsBuilder.Options);
            return await context.FederalDistricts.Include(x => x.FederalSubjects).ToListAsync();
```

```
        }

        public IEnumerable<FederalDistrict> FederalDistricts()
        {
            var conn = configurationRoot.GetConnectionString("EFCoreTestContext");
            optionsBuilder.UseSqlServer(conn);

            using var ctx = new EFCoreTestContext(optionsBuilder.Options);
            return ctx.FederalDistricts.Include(x => x.FederalSubjects).ToList();
        }
    }

  // Вызов компонента происходит в данном примере на странице Home\Index
   <div id="tableContainer">
           @await Component.InvokeAsync("PopularPosts")
    </div>
// А собственно HTML с моделю по пути  Shared\Components\PopularPosts\Default.cshtml
```

Напомню, когда выполняются запросы в Entity Framework Core.

При вызове операторов LINQ вы просто создаете представление запроса в памяти. Запрос отправляется в базу данных только после обработки результатов.

Ниже приведены наиболее распространенные операции, которые приводят к отправке запроса в базу данных.

- Итерация результатов в цикле for.
- Использование оператора, например ToList, ToArray, Single, Count.
- Привязка данных результатов запроса к пользовательскому интерфейсу.

Как же организовать код EF Core с точки зрения архитектуры приложения?

(1) С точки зрения архитектуры приложения, нужно обеспечить чтобы код доступа к вашей базе данных был изолирован / отделен в четко определенном месте (в изоляции). Это позволяет найти код базы данных, который влияет на производительность.

(2) Не смешивать код доступа к вашей базе данных с другими частями приложения, такими как пользовательский интерфейс или API. Таким образом, код доступа к базе данных можно изменить, не беспокоясь о других проблемах, не связанных с базой данных.

Как правильно и быстро сохранять данные с помощью *SaveChanges*?

Если вставляемые записи одинаковые имеет смысл использовать одну операцию сохранения на все записи.

Неправильно

```
using(var db = new NorthwindEntities())
{
var transaction = db.Database.BeginTransaction();

try
{
    // Вставка записи 1
    var  obj1 = new Customer();
    obj1.CustomerID = "ABCDE";
    obj1.CompanyName = "Company 1";
    obj1.Country = "USA";
    db.Customers.Add(obj1);

  //Сохраняем первую запись        db.SaveChanges();

    // Вставка записи 2
    var  obj2 = new Customer();
    obj2.CustomerID = "PQRST";
    obj2.CompanyName = "Company 2";
```

```
    obj2.Country = "USA";
    db.Customers.Add(obj2);

    // Сохраняем вторую запись
    db.SaveChanges();

    transaction.Commit();
}
catch
{
    transaction.Rollback();
}
}
```

Правильно

```
using(var db = new NorthwindEntities())
{
var transaction = db.Database.BeginTransaction();

try
{
   //Вставка записи 1
    var  obj1 = new Customer();
    obj1.CustomerID = "ABCDE";
    obj1.CompanyName = "Company 1";
    obj1.Country = "USA";
    db.Customers.Add(obj1);

    // Вставка записи 2
    var  obj2 = new Customer();
    obj2.CustomerID = "PQRST";
    obj2.CompanyName = "Company 2";
    obj2.Country = "USA";
    db.Customers.Add(obj2);

   // Сохранение двух или N записей
    db.SaveChanges();

    transaction.Commit();
}
catch
{
    transaction.Rollback();
}
}
```

Всегда есть исключения из правила. Если контекст транзакции сложный, то есть состоит из нескольких независимых операций, то можно выполнять сохранение после выполнения каждой операции. А еще правильней использовать асинхронное сохранение в транзакции.

```
// Увеличение депозита его владельца
public async Task<IActionResult> AddDepositToHousehold(int householdId, DepositRequestModel model)
{
    using (var transaction = await Context.Database.BeginTransactionAsync(IsolationLevel.Snapshot))
    {
        try
        {
            // Добавить депозит в БД
            var deposit = this.Mapper.Map<Deposit>(model);
            await this.Context.Deposits.AddAsync(deposit);

            await this.Context.SaveChangesAsync();

            // Оплатить задолжности с депозита
               var debtsToPay = await this.Context.Debts.Where(d => d.HouseholdId == householdId && !d.IsPaid).OrderBy(d =
> d.DateMade).ToListAsync();
```

```
            debtsToPay.ForEach(d => d.IsPaid = true);

            await this.Context.SaveChangesAsync();

            // Увеличение баланса владельца
            var household = this.Context.Households.FirstOrDefaultAsync(h => h.Id == householdId);

            household.Balance += model.DepositAmount;

            await this.Context.SaveChangesAsync();

            transaction.Commit();
            return this.Ok();
        }
        catch
        {
            transaction.Rollback();
            return this.BadRequest();
        }
    }
}
```

Триггеры, вычисляемые поля, пользовательские функции и EF Core

Для снижения нагрузки на приложения содержащим EF Core имеет смысл применять простые вычисляемые поля и триггеры баз данных, но лучше этим не увлекаться, так как приложение может оказаться очень запутанным. А вот пользовательские функции могут быть очень полезны особенно при операциях выборки!

Параллелизм в EF Core

Если ты хочешь все запараллелить чтобы ускориться, то обломись: EF Core не поддерживает выполнение нескольких параллельных операций в одном экземпляре контекста. Следует подождать завершения одной операции, прежде чем запускать следующую. Для этого обычно нужно указать ключевое слово await в каждой асинхронной операции.

EF Core использует асинхронные запросы, которые позволяют избежать блокирования потока при выполнении запроса в базе данных. Асинхронные запросы важны для обеспечения быстрого отклика пользовательского интерфейса в толстых клиентах. Они могут также увеличить пропускную способность в веб-приложении, где можно высвободить поток для обработки других запросов. Вот пример:

```
public async Task<List<Blog>> GetBlogsAsync()
{
    using (var context = new BloggingContext())
    {
        return await context.Blogs.ToListAsync();
    }
}
```

А что вы знаете про компилированные запросы LINQ?

Если у вас есть приложение, которое многократно выполняет структурно похожие запросы в Entity Framework, вы часто можете повысить производительность, компилируя запрос один раз и выполняя его несколько раз с различными параметрами. Например, приложению может потребоваться получить всех клиентов в определенном городе; город указывается во время выполнения пользователем в форме. LINQ to Entities поддерживает использование для этой цели скомпилированных запросов.

Начиная с .NET Framework 4.5, запросы LINQ кэшируются автоматически. Тем не менее, вы все равно можете использовать скомпилированные запросы LINQ, чтобы снизить эту стоимость в последующих выполнениях, и скомпилированные запросы могут быть более эффективными, чем запросы LINQ, которые автоматически кэшируются. Обратите внимание, что запросы LINQ to Entities, которые применяют оператор Enumerable.Contains к коллекциям в памяти, не кэшируются автоматически. Также не допускается параметризация коллекций в памяти в скомпилированных запросах LINQ.

Много примеров можно посмотреть тут.

Не делайте больших контекстов DbContext!

В общем так, я знаю многие из вас, если не почти все — lazy f_u__c_k__e_r__s и всю базу данных вы размещаете в один контекст, особенно это свойственно для подхода Database-First. И зря вы это делаете! Ниже приведен пример как можно разделить контекст. Конечно, таблицы соединения между контекстами придется дублировать, это минус. Так или иначе если у вас в контексте более 50 таблиц лучше подумать о его разделении.

Использование группировки контекста (pooling DdContext)

Смысл пула **DbContext** состоит в том, чтобы разрешить повторное использование экземпляров **DbContext** из пула, что в некоторых случаях может привести к повышению производительности по сравнению с созданием нового экземпляра каждый раз. Это также является основной причиной создания пула соединений в ADO.NET, хотя прирост производительности для соединений будет более значительным, поскольку соединения, как правило, являются более тяжелым ресурсом.

```
using System;
using System.Diagnostics;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace Demos
{
    public class Blog
    {
        public int BlogId { get; set; }
        public string Name { get; set; }
        public string Url { get; set; }
    }

    public class BloggingContext : DbContext
    {
        public static long InstanceCount;

        public BloggingContext(DbContextOptions options)
            : base(options)
            => Interlocked.Increment(ref InstanceCount);

        public DbSet<Blog> Blogs { get; set; }
    }

    public class BlogController
    {
        private readonly BloggingContext _context;

        public BlogController(BloggingContext context) => _context = context;

        public async Task ActionAsync() => await _context.Blogs.FirstAsync();
    }

    public class Startup
    {
        private const string ConnectionString
            = @"Server=(localdb)\mssqllocaldb;Database=Demo.ContextPooling;Integrated Security=True;ConnectRetryCount=0";

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<BloggingContext>(c => c.UseSqlServer(ConnectionString));
        }
    }

    public class Program
    {
        private const int Threads = 32;
        private const int Seconds = 10;

        private static long _requestsProcessed;
```

```
private static async Task Main()
{
    var serviceCollection = new ServiceCollection();
    new Startup().ConfigureServices(serviceCollection);
    var serviceProvider = serviceCollection.BuildServiceProvider();

    SetupDatabase(serviceProvider);

    var stopwatch = new Stopwatch();

    MonitorResults(TimeSpan.FromSeconds(Seconds), stopwatch);

    await Task.WhenAll(
        Enumerable
            .Range(0, Threads)
            .Select(_ => SimulateRequestsAsync(serviceProvider, stopwatch)));
}

private static void SetupDatabase(IServiceProvider serviceProvider)
{
    using (var serviceScope = serviceProvider.CreateScope())
    {
        var context = serviceScope.ServiceProvider.GetService<BloggingContext>();

        if (context.Database.EnsureCreated())
        {
            context.Blogs.Add(new Blog { Name = "The Dog Blog", Url = "http://sample.com/dogs" });
            context.Blogs.Add(new Blog { Name = "The Cat Blog", Url = "http://sample.com/cats" });
            context.SaveChanges();
        }
    }
}
private static async Task SimulateRequestsAsync(IServiceProvider serviceProvider, Stopwatch stopwatch)
{
    while (stopwatch.IsRunning)
    {
        using (var serviceScope = serviceProvider.CreateScope())
        {
            await new BlogController(serviceScope.ServiceProvider.GetService<BloggingContext>()).ActionAsync();
        }

        Interlocked.Increment(ref _requestsProcessed);
    }
}

private static async void MonitorResults(TimeSpan duration, Stopwatch stopwatch)
{
    var lastInstanceCount = 0L;
    var lastRequestCount = 0L;
    var lastElapsed = TimeSpan.Zero;

    stopwatch.Start();

    while (stopwatch.Elapsed < duration)
    {
        await Task.Delay(TimeSpan.FromSeconds(1));

        var instanceCount = BloggingContext.InstanceCount;
        var requestCount = _requestsProcessed;
        var elapsed = stopwatch.Elapsed;
        var currentElapsed = elapsed - lastElapsed;
        var currentRequests = requestCount - lastRequestCount;

        Console.WriteLine(
            $"[{DateTime.Now:HH:mm:ss.fff}] "
            + $"Context creations/second: {instanceCount - lastInstanceCount} | "
            + $"Requests/second: {Math.Round(currentRequests / currentElapsed.TotalSeconds)}");

        lastInstanceCount = instanceCount;
        lastRequestCount = requestCount;
```

```
            lastElapsed = elapsed;
        }

        Console.WriteLine();
        Console.WriteLine($"Total context creations: {BloggingContext.InstanceCount}");
        Console.WriteLine(
            $"Requests per second:     {Math.Round(_requestsProcessed / stopwatch.Elapsed.TotalSeconds)}");

        stopwatch.Stop();
    }
```

Как избежать лишних ошибок при CRUD в EF Core?

Никогда не делайте вычисления в вставку в одном коде. Всегда разделяйте формирование/подготовку объекта и его вставку/обновление. Просто разнесите по функциям: проверку введенных данным пользователем, вычисления необходимые предварительных данных, картирование или создание объекта, и собственно CRUD операцию.

Что делать, когда совсем дела плохо с производительностью приложения?

Пиво тут точно не поможет. А вот что поможет, так это разделение чтение и записи в архитектуре приложения с последующего разнесением по сокетам этих операций. Задумайтесь об использовании Command and Query Responsibility Segregation (CQRS) pattern, а также попробуйте, разделить таблицы на вставку и чтение между двумя базами данных.

Скоростных приложений вам, друзья и коллеги!

Теги: entity framework core, оптимизация, c#.net

Хабы: .NET, C#

4,0 Карма 0,0 Рейтинг

Дмитрий Бойко @DmitryNBoyko

Full Stack Developer

Комментарии 42

lair 10 февраля 2020 в 12:37 +4

> Что насчет асинхронности?
> Да! Используем ее где только можно!

Аргументируйте. В том смысле, что докажите, вот этот код (ваш пример):

```
public void Do()
{
    var myTask = GetFederalDistrictsAsync ();
    foreach (var item in myTask.Result)
    {
        //Ваш код
    }
}

public async Task<List<FederalDistrict>> GetFederalDistrictsAsync()
{
    var conn = configurationRoot.GetConnectionString("EFCoreTestContext");
    optionsBuilder.UseSqlServer(conn);
    using var context = new EFCoreTestContext(optionsBuilder.Options);
```

```
    return await context.FederalDistricts.ToListAsync();
}
```

быстрее, чем вот этот:

```
public void Do()
{
    foreach (var item in GetFederalDistricts())
    {
        //Ваш код
    }
}

public List<FederalDistrict> GetFederalDistricts()
{
    var conn = configurationRoot.GetConnectionString("EFCoreTestContext");
    optionsBuilder.UseSqlServer(conn);
    using var context = new EFCoreTestContext(optionsBuilder.Options);
    return context.FederalDistricts.ToList();
}
```

Не говоря уже о том, что вот так:

```
var myTask = GetFederalDistrictsAsync ();
foreach (var item in myTask.Result)
```

делать без веских причин **не надо**.

kuda78 10 февраля 2020 в 13:52 −1

Так автор и не говорит, что он быстрее.
Он говорит, что операция может быть выполнена асинхронно, и на время ожидания ответа от сервера поток будет освобожден для выполнения других задач.

mayorovp 10 февраля 2020 в 13:59 0

Конкретно код автора потока не освободит, даже наоборот: займёт ещё один, пусть и на короткое время.

lair 10 февраля 2020 в 14:02 +2

> Так автор и не говорит, что он быстрее.

Заголовок статьи вроде бы "ускоряемся". Я по умолчанию предполагаю, что советы — они про ускорение.

> и на время ожидания ответа от сервера поток будет освобожден для выполнения других задач.

Не в случае кода автора.

А, главное, если моя задача в том, чтобы запросы выполнялись быстрее, как мне это поможет?

aftertherainbow 10 февраля 2020 в 14:18 +1

> А, главное, если моя задача в том, чтобы запросы выполнялись быстрее, как мне это поможет?

Я думаю автор хотел сказать, что если бы запросы *действительно* выполнялись асинхронно, то в целом приложение бы выиграло из-за более эффективного использования потоков в пуле.
То есть запросы это конечно не ускорит, но писать асинхронный код обычно полезно.

lair 10 февраля 2020 в 14:26 0

> то в целом приложение бы выиграло из-за более эффективного использования потоков в пуле

Возможно выиграло бы. Это уже зависит от приложения, его задач и характера нагрузки.

> То есть запросы это конечно не ускорит, но писать асинхронный код обычно полезно.

Мне кажется, что полезно понимать, что в этом случае происходит, и где профит, а где — потери.

kuda78 10 февраля 2020 в 14:27 0

> Не в случае кода автора.

Про метод Do я и не спорю. Автор поспешил и, возможно, не очень хорошо разбирается в магии async/await

> А, главное, если моя задача в том, чтобы запросы выполнялись быстрее, как мне это поможет?

Запрос, в общем случае выполнится медленнее, из за накладных расходов на работу с тасками.
Но в зависимости от специфики приложения, приложение в целом может начать работать быстрее.

lair 10 февраля 2020 в 14:29 +2

> Про метод Do я и не спорю. Автор поспешил и, возможно, не очень хорошо разбирается в магии async/await

Может быть, если не очень хорошо разбираешься в магии, не надо советовать ее применять "где только можно"?

> Запрос, в общем случае выполнится медленнее, из за накладных расходов на работу с тасками.

То есть эффект достигнут прямо противоположный обещанному в заголовке.

> Но в зависимости от специфики приложения, приложение в целом может начать работать быстрее.

Вот именно что "в зависимости от специфики". Это прямо противоречит "где только можно".

aftertherainbow 10 февраля 2020 в 14:02 0

Код выполняется синхронно так как блокируется поток в момент вызова .Result на таске

Matisumi 11 февраля 2020 в 15:06 0

А как было бы правильно переписать этот пример без блокировок потока?

aftertherainbow 11 февраля 2020 в 15:27 +2

```
public async Task Do()
{
    var districts = await GetFederalDistrictsAsync();
    foreach (var district in districts)
    {
    }
}
```

Примерно так. Асинхронность она как вирус — распространяется по всему проекту.

Matisumi 11 февраля 2020 в 16:02 0

А что тогда на самом верхнем уровне?

lair 11 февраля 2020 в 16:06 +1

Фреймворк, который умеет в асинхронию. asp.net core, например.

Matisumi 11 февраля 2020 в 16:09 0

То есть если сильно упростить — идея в том, чтобы протащить асинхронность до, условно, пользовательского отображения, и там ее отработать без «фриза» интерфейса?

lair 11 февраля 2020 в 16:11 +1

~~В каком-то смысле. И в обратную сторону — сделать пользовательский интерфейс асинхронным, чтобы сделать его более отзывчивым.~~

Все-таки, нет. Отзывчивый пользовательский интерфейс — это одна задача. Более разумное использование ресурсов (например, потоков) в серверном приложении — другая.

Matisumi 11 февраля 2020 в 16:16 0

Спасибо! Но все равно тогда до конца не понимаю. Допустим есть серверное приложение с асинхронным API — в чем тогда будет выражаться разумное использование ресурсов? Ведь это асинхронное API будет дергать некий клиент. На уровне серверного приложения разве будет выигрыш в том, что API асинхронное, а не синхронное? Ведь если я ничего не путаю, на обработку каждого запроса синхронного API все равно выделяется отдельный поток из пула?

lair 11 февраля 2020 в 16:19 +2

> На уровне серверного приложения разве будет выигрыш в том, что API асинхронное, а не синхронное?

Может быть.

> Ведь если я ничего не путаю, на обработку каждого запроса синхронного API все равно выделяется отдельный поток из пула?

В том-то и дело, что если API реализовано как асинхронное, и в какой-то момент в нем случается ожидание, не требующее потока (например, мы ждем HTTP-запрос от следующего сервера, или в БД полезли), мы можем отпустить поток, и он будет доступен для следующего запроса. И, возможно, этот запрос даже выполнится быстрее, чем время ожидания.

Это не всегда хорошо, но иногда это выигрыш.

Matisumi 11 февраля 2020 в 16:46 0

А, вот оно как. Спасибо огромное за разъяснение (к сожалению не могу поставить плюс комментарию)

aftertherainbow 11 февраля 2020 в 16:28 0

Что-то вроде такого.

```
async void button_Click(object sender, EventArgs e)
{
     await Do();
}
```

или такого

```
[HttpGet]
public async Task<IHttpActionResult> Get()
{
      await Do();
      return Ok();
}
```

DmitryNBoyko 10 февраля 2020 в 17:10 0

Душевно благодарю на указанную неточность! Исправлено, с добавлением расширенного примера использования асинхронного запроса из репозитория бизнес-логики в компонент представления ASP .NET Core. В репозитории присуствует синхронный и асинхронный варианты методов получения данных. Еще раз спасибо, за просмотр моей статьи!

lair 10 февраля 2020 в 19:37 +1

> Исправлено

Неа. Рекомендация "используем асинхронность где только можно" так и осталась.

Аргументируйте.

Quilin 11 февраля 2020 в 11:17 −1

В core вроде как вызовы .Result и .Wait() не могут приводить к дедлокам, насколько мне известно. Хотя, конечно, ничто не мешает в этом примере заавейтить результат.

Поправьте меня, если я ошибаюсь? Есть ли какие-то другие подводные камни, кроме тех что исчезли с уходом контекста синхронизации?

lair 11 февраля 2020 в 11:21 +1

Насколько я могу видеть, самое важное не поменялось (выделение мое):

> Accessing the property's get accessor **blocks the calling thread** until the asynchronous operation is complete

Ну то есть да, если коду внутри не нужно возвращаться в тот же поток, *дедлока* не будет. Но вот потоков может быть занято *больше*, что скорее плохо, чем хорошо.

mayorovp 11 февраля 2020 в 11:30 0

Что значит — не могут? Контексты синхронизации никуда не делись, просто они в ASP.NET Core не используются.

Попробуйте написать декстопное приложение на AvaloniaUI — и дедлоки при вызовах .Result и .Wait() сразу вернутся!

questor 10 февраля 2020 в 13:19 +1

Статья в принципе не особо высокого уровня, каких-то кардинально новых вещей не увидел, хотя и хотелось. (Но всё равно статье плюс, в целом сгодится хотя бы как "повторение — мать учения").

Не хватает конкретики, на сколько можно разогнаться тем или иным способом.
Например, насколько я помню, майкрософт в одном из документов приводила цифры про ускорение при помощи .AsNoTracking и там были цифры порядка 15-20% (документ 2005 — 2008 года, его уже на офсайте найти весьма проблематично), новых исследований для core я к сожалению не видел, поэтому просто полагаюсь что по-прежнему даёт довольно существенный прирост.

Также было бы неплохо вот тут проиллюстрировать хотя простым тестом:

> Если вставляемые записи одинаковые имеет смысл использовать одну операцию сохранения на все записи. Если контекст транзакции сложный, то есть состоит из нескольких независимых операций, то можно выполнять сохранение после выполнения каждой операции. А еще правильней использовать асинхронное сохранение в транзакции.

Можно было бы создать таблицу на 10-15 полей без ключей к другим таблицам и показать на сколько примерно возрастёт скорость. Даже семейство графиков построить: если таблица на 5 полей, на 10, на сто. Или померять как влияют самые типовые поля типа строк или чисел.

> А что вы знаете про компилированные запросы LINQ?

Да и вот тут пример с цифрами не помешал бы. Я не знаю, насколько большие цифры будут, мне кажется, что не особо большие — потому что в .net первый раз компиляция происходит довольно медленно, но за счёт того, что повторно код заново не компилируется — то на второй и последующие разы эффект уже не особо сильный. Ошибаюсь? Нет? Вот не знаю, хотелось бы видеть цифры.

НЛО прилетело и опубликовало эту надпись здесь

НЛО прилетело и опубликовало эту надпись здесь

mayorovp 10 февраля 2020 в 13:58 +1

> AsQueryable
> Принимает параметр: `Expression<Func <TSource, bool>`

Что я только что прочитал?

DmitryNBoyko 10 февраля 2020 в 16:21 0

Это ошибка. Спасибо, что заметили. Исправлено и дополнено примером. Еще раз благодарю!

iColt_BLR 10 февраля 2020 в 15:55 0

В 3 примере, про выполнение шарп кода в запросах, вы привели данный код как пример правильного:

```
var blogs = context.Blogs.AsEnumerable().Where(blog => StandardizeUrl(blog.Url).Contains("dotnet")).ToList();
```

Но не противоречит ли этот пример теме статьи? Ведь все данные все равно загрузятся в память.

DmitryNBoyko 10 февраля 2020 в 15:57 0

Думаю, что нет. Ключевым моментом является использование **AsEnumerable()** так как предполагается получение отфильтрованных записей только на чтение.
Соглашусь, что **StandardizeUrl(blog.Url)** может быть выполнено позже когда найдены все записи по условию **.Contains(«dotnet»)**.

lair 10 февраля 2020 в 16:00 0

> Соглашусь, что StandardizeUrl(blog.Url) может быть выполнено позже когда найдены все записи по условию .Contains(«dotnet»).

А что делать с тем печальным фактом, что до `Standardize` `Contains` находит меньше записей, чем после?

НЛО прилетело и опубликовало эту надпись здесь

Begunini 11 февраля 2020 в 13:42 0

Солидарен. Мало того, что с 3 версии EF Core выпилено дефолтное client-side evaluation и теперь такой код кинет ошибку в момент выполнения, если убрать AsEnumerable, а если не убирать, то сравнение строк на недавний момент существует лишь в виде Ordinal (не проверял 3.1 ещё), так ещё и вопрос к производительности — одно дело contains, который легко может быть выполнен на сервере и само по себе более легковесная операция, нежели преобразование целого урла к какому-то виду.

DmitryNBoyko 11 февраля 2020 в 13:44 0

Вы написали "... нежели преобразование целого урла к какому-то виду."
Урл преобразуется на стороне сервера, этот код не передается на сервер можно проверить профайлером SQL.

НЛО прилетело и опубликовало эту надпись здесь

aftertherainbow 11 февраля 2020 в 17:18 0

```
c=>c.url.ToLower().StartsWith("http://")
```

Если рассматривать общий случай, то это скорее плохой код. Зависимость от регистра определяется по коллейшену столбца в MSSQL и по умолчанию они регистронезависимые а значит преобразование к нижнему регистру тут лишнее. Ну и в целом лучше при сравнении преобразовывать к верхнему регистру потому что это быстрее.

НЛО прилетело и опубликовало эту надпись здесь

RouR 11 февраля 2020 в 22:21 −2

Транзакции, еще правильнее.

```
using(var db = new NorthwindEntities())
{

    var  obj1 = new Customer();
    obj1.CustomerID = "ABCDE";
    obj1.CompanyName = "Company 1";
    obj1.Country = "USA";

    var  obj2 = new Customer();
    obj2.CustomerID = "PQRST";
    obj2.CompanyName = "Company 2";
    obj2.Country = "USA";

    var transaction = db.Database.BeginTransaction();
    try
    {
        db.Customers.Add(obj1);
        db.Customers.Add(obj2);
        db.SaveChanges();
        transaction.Commit();
    }
    catch
    {
        transaction.Rollback();
    }
}
```

mayorovp 12 февраля 2020 в 07:05 +1

Конкретно в этом случае транзакция нафиг не нужна. SaveChanges создаст транзакцию автоматически.

RouR 12 февраля 2020 в 09:34 0

Я про то что транзакции должны быть как можно короче.

AdAbsurdum 20 февраля 2020 в 23:08 +1

> Если ты хочешь все запараллелить чтобы ускориться, то обломись

А вот и не обломись:

```
public async Task<List<Blog>> GetBlogsAsync()
{
    using (var context = new BloggingContext())
    using (var context2 = new BloggingContext())
    {
        // parallel queries
    }
}
```