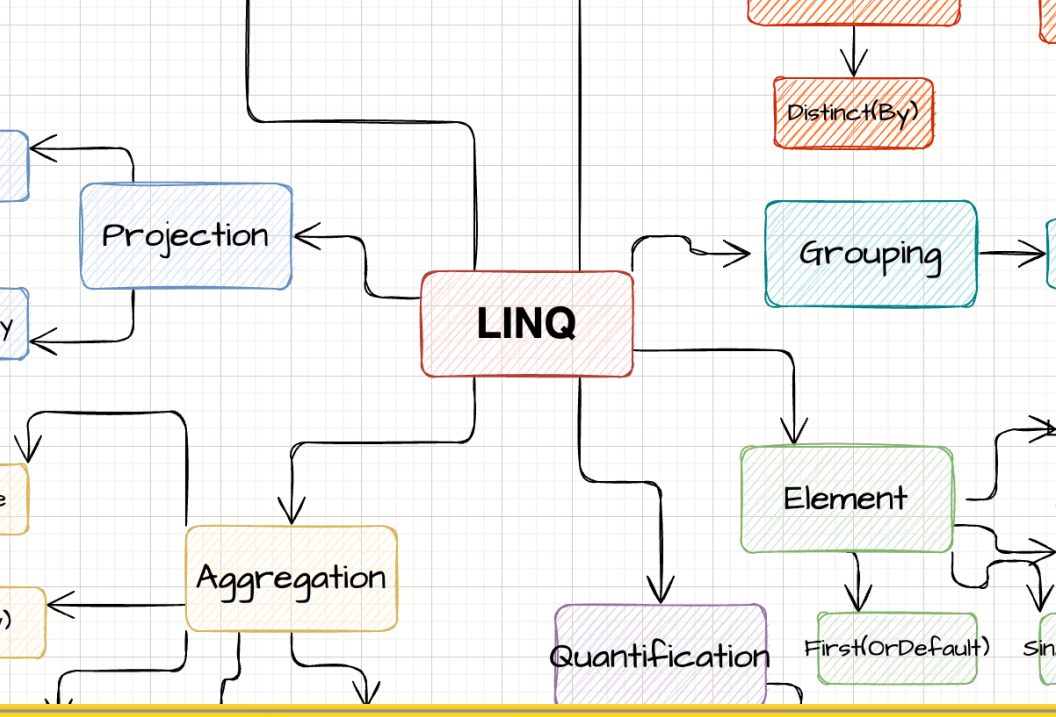


LINQ

Explicado com desenhos

Por Steven Giesel

Tradução Jefferson Rocha

Prólogo

Esta pequena obra foi criada com o objetivo de dar aos iniciantes uma introdução visual e simples ao LINQ. Segundo o lema: "Imagens falam mais que mil palavras".
Qual é o objetivo deste pequeno livro? Ele deve permitir que você use os métodos do LINQ corretos na situação certa. Vou começar cada vez com um pequeno desenho, que será completado por uma pequena explicação, incluindo um exemplo de código. Vou abordar menos coisas como "LINQ-ToObject". Além disso, tocarei brevemente em coisas como **IQueryable** e **IEnumerable**.

Sumário

O que é LINQ?

LINQ é a abreviação de “Language-Integrated Query” (linguagem integrada de consultas) e é o nome de um conjunto de tecnologias baseadas na integração de recursos de consulta diretamente no C#1.
O objetivo é ter uma maneira uniforme e estruturada de operar em enumerações. As consultas LINQ retornam sempre o resultado como novos objetos. Isso garante que a enumeração original não será modificada. Isso é muito importante lembrar. Todas as consultas LINQ retornam uma nova enumeração em vez de excluir, atualizar ou adicionar novos itens ao determinado.
Além disso, existem maneiras de transformar consultas LINQ em sintaxe SQL ou usar LINQ para passar por um documento XML. O tipo básico em que **todas** as consultas LINQ operam é **IEnumerable**.

[1] Definição: learn.microsoft.com/pt-BR/dotnet/csharp/programming-guide/concepts/linq/

IEnumerable

O tipo básico em que todas as consultas LINQ operam é o IEnumerable. Sem entrar em muitos detalhes, é fundamental entender que **IEnumerable** não representa uma lista "materializada". Chamamos isso de "avaliação preguiçosa" ("lazy evaluation"). Isso significa que, no momento de executar as consultas LINQ, não obtemos os resultados reais. Somente quando enumeramos através da enumeração ou chamamos operações como **ToList** ou **Count** é que realmente "criamos" / "materializamos" o objeto.
Agora você verá um pequeno trecho. Não se preocupe se você não entender isso agora. Tome isso como uma motivação para entender completamente isso depois de ler o pequeno livro:

```
var list = new List<int>();
list.Add(1);
list.Add(2);

var evenNumbers = list.Where(n => n % 2 == 0);

list.Add(4);
Console.WriteLine($"Números pares na lista:
{evenNumbers.Count()}");
```

Criamos a enumeração depois que a lista contém 2 elementos. Depois adicionamos outro número à própria lista. Então, quantos números pares temos na enumeração? A resposta é: **2**. A razão é que nós materializamos a enumeração ao chamar **Count** e não no momento de criar a enumeração. Então quando chamamos **Count** temos dois elementos, que são números pares (2 e 4). Sempre tenha isso em mente.

Outro tipo sempre associado ao LINQ é IQueryable. IQueryable é basicamente IEnumerable com algo a mais e é exatamente essa parte do "algo a mais" que o torna tão único. Para isso, vou apenas listar os pontos principais aqui e fazer referência à minha postagem no blog:

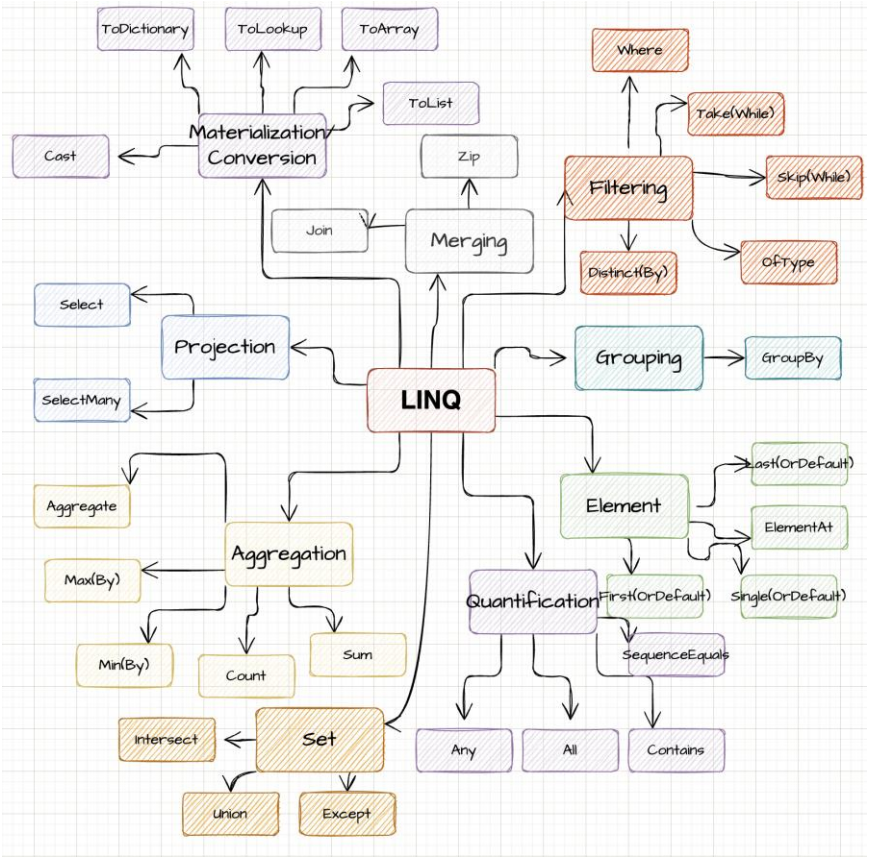
IEnumerable vs IQueryable - Qual a diferença? no qual eu descrevo a diferença em detalhes.

- Tanto IEnumerable quanto IQueryable são “coleções de encaminhamento” (“forward collections”) - elas não são materializadas imediatamente
- A consulta de dados do banco de dados IEnumerable carregará os dados na memória para depois filtrar no cliente
- A consulta de dados do banco de dados IQueryable irá filtrar primeiro e depois enviar os dados filtrados para o cliente
- IQueryable é adequado para consultar dados que não estão em memória
- Pode haver cenários em que o provedor de consulta subjacente não pode traduzir sua expressão para algo significativo, então você deve mudar para IEnumerable

Mindmap

O LINQ tem muitas operações em seu conjunto de ferramentas para você. Assim, podemos agrupá-los em diferentes categorias. A próxima imagem mostrará uma visão geral aproximada para que você possa obter uma imagem mental. Eu também aconselharia voltar a essa imagem várias vezes para ver onde você está.

Os próximos capítulos são organizados exatamente por essas categorias.



O verdadeiro poder do LINQ surge quando você combina várias operações. Após a explicação dos operadores LINQ, você encontrará alguns exemplos do mundo real em que várias operações LINQ são usadas em uma instrução.

```
IEnumerable<BlogPost> allBlogPosts = await GetAllBlogPosts();

var publishedBlogPosts = allBlogPosts
    .Where(bp => bp.IsPublished)
    .OrderByDescending(bp => bp.PublishDate)
    .Skip(pageSize * (page - 1))
    .Take(pageSize)
    .ToList();
```

Filtros

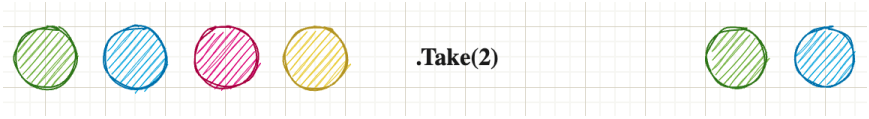
O capítulo a seguir descreve como alguém pode usar LINQ para filtrar a enumeração com base na operação fornecida.

Where

Com **Where** podemos filtrar uma determinada lista com base em nossa condição. O método aceita um Predicado. Isso significa que definimos uma função de filtro que é aplicada objeto por objeto. Se o filtro for avaliado como **verdadeiro**, o elemento será retornado na nova enumeração.

```
var list = new List<int>();
list.Add(1);
list.Add(2);

// Retorna os números pares
// Resultado: [ 2 ]
var evenNumbers = list.Where(n => n % 2 == 0);
```

Take



Take nos permite "pegar" a quantidade dada de elementos. Se tivermos menos elementos na matriz do que queremos, Take() retornará apenas os objetos restantes.

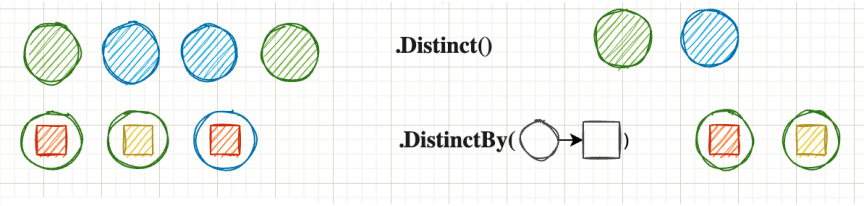
```
var list = new List<int>();
list.Add(1);
list.Add(2);

// Result: [ 1 ]
var takeOne = list.Take(1);

// Result: [ 1, 2 ]
var takeOneHundred = list.Take(100);
```

Skip

Com **Skip** nós "pulamos" a quantidade dada de elementos. Se pularmos mais elementos do que nossa lista contém, obteremos uma enumeração vazia de volta. Take e Skip juntos podem ser muito poderosos para coisas como paginação.

```
var list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);


// Result: [ 2, 3 ]
var skipOne = list.Skip(1);

// Result: [ ]
var skipOneHundred = list.Take(100);

// Result: [ 2 ]
var pageTwo = list.Skip(1).Take(1);
```

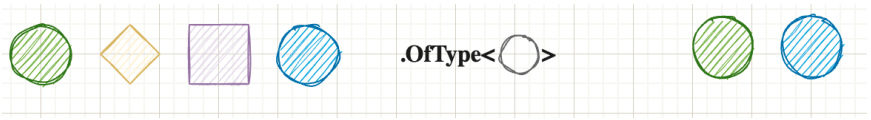
Distinct(By)



Distinct retorna um novo enumerável onde todas as duplicatas são removidas, mais ou menos como em um Set. Tenha cuidado para que, para enumerações de referências (reference type), o padrão é verificar a igualdade de referência, o que pode levar a resultados diferentes do esperado. O conjunto de resultados pode ser do mesmo tamanaho ou menor que o original.
DistinctBy funciona de maneira semelhante a **Distinct**, mas em vez do nível do objeto em si, podemos definir uma projeção para uma propriedade onde queremos ter um conjunto de resultados distinto.

```
var people = new List<Person>
{
    new Person("Steven", 31),
    new Person("Katarina", 29),
    new Person("Nils", 31)
};

// [
//     Person { Name = Steven, Age = 31 },
//     Person { Name = Katarina, Age = 29 }
// ]
var uniqueAgedPeople = people.DistinctBy(p => p.Age);

record Person(string Name, int Age);
```

OfType



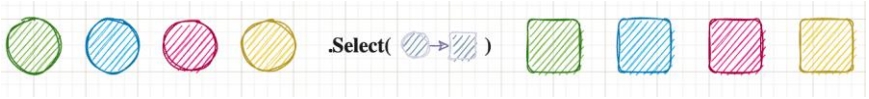
O **OfType** verifica cada elemento na enumeração se é do tipo dado (também tipos herdados contam como esse tipo) e os retorna em uma nova enumeração. Isso ajuda especialmente se tivermos matrizes não tipadas (objeto) ou quisermos uma subclasse especial da enumeração dada.

```
var fruits = new List<Fruit>
{
    new Banana(),
    new Apple()
};

// [
//     Apple { }
// ]
var apples = fruits.OfType<Apple>();

record Fruit;
record Banana : Fruit;
record Apple : Fruit;
```

Projeção

Projeção descreve a transformação de um objeto em uma nova forma. Ao usar projeções, é possível criar um novo tipo que é construído a partir do seu tipo original.

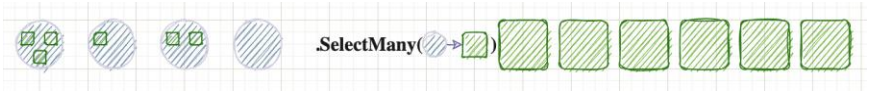
Select



Com o Select, criamos uma projeção de um item para outro. Falando de forma simples, mapeamos de um tipo dado para um tipo desejado. O resultado tem a mesma quantidade de itens que a origem.

```
var objects = new List<SourceObject>
{
    new SourceObject(1),
    new SourceObject(2),
};

// [
//     TargetObject { NumberAsString: "1" },
//     TargetObject { NumberAsString: "2" },
// ]
var targetObjects = objects.Select(o => new
TargetObject(o.ToString()));

record SourceObject(int Number);
record TargetObject(string NumberAsString);
```

SelectMany



O SelectMany é utilizado para achatamento de listas. Se você tem uma lista dentro de uma lista, podemos usá-lo criar uma representação unidimensional.

```
var recipes = new List<Recipe>
{
    new Recipe("Pizza", new() { "Tomato Sauce, Basil" }),
    new Recipe("Hot Water", new() { "Water" }),
};

// [
//     "Tomato Sauce", "Basil", "Water"
// ]
var allIngredients = recipes.SelectMany(r => r.Ingredients);

record Recipe(string Name, List<string> Ingredients);
```

Agregação

A agregação descreve o processo de reduzir a enumeração inteira a um único valor.

Count

Com o **Count**, contamos os elementos por uma dada função. Se a função for avaliada como **verdadeira**, aumentamos o contador em um.

```
var names = new[] { "Steven", "Marie", "Steven" };

// 2
var stevens = names.Count(n => n == "Steven");
```

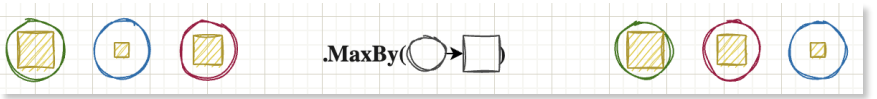
Aggregate

Aggregate, também conhecido como **reduce**, agrega/reduz **todos** os elementos em um valor escalar. Um dos melhores exemplos é a soma de uma lista. Começamos com 0 e adicionamos cada elemento até percorrermos toda a nossa enumeração. O primeiro parâmetro de agregação é o valor

inicial. Uma enumeração vazia resultará em retornar o seu valor inicial.

```
var numbers = new[] { 1, 2, 3 };

// 6
var sum = numbers.Aggregate(0, (curr, next) => curr + next);

// 6
var sumLinq = numbers.Sum();
```

Max(By)



Max(By) recupera o maior elemento. Isso também pode ser representado por uma função de agregação. Se usarmos **Max** ou **MaxBy** em uma enumeração vazia, será lançada uma exceção, informando que a sequência não contém nenhum elemento.

```
// 3
var max = new[] { 1, 2, 3 }.Max();

var people = new[]
{
    new Person("Steven", 31),
    new Person("Jean", 22)
};

// Person { Name: Steven, Age: 31 }
var oldest = people.MaxBy(p => p.Age);

record Person(string Name, int Age);
```

Claro que **Min(By)** funciona de forma semelhante. A diferença é, obviamente, que o menor valor é recuperado em vez do maior.

Quantificação

Este capítulo analisa a quantificação de elementos. Essas operações medem a quantidade de algo.

Any

Any verifica se pelo menos um elemento satisfaz sua condição. Se sim, ele retorna **verdadeiro**. Se não houver nenhum elemento que atenda à condição, ele retorna falso. Any também para o processamento imediatamente assim que encontra um elemento. Ele retorna **falso** se a enumeração dada estiver vazia.

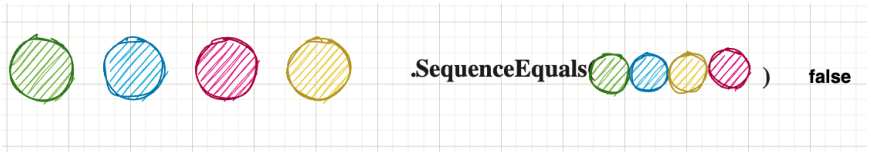
```
var fruits = new[]
{
    new Fruit("Banana", 89),
    new Fruit("Apple", 51),
};

// true
var hasDenseFood = fruits.Any(f => f.CaloriesPer100Gramm > 80);

record Fruit(string Name, int CaloriesPer100Gramm);
```

All

Como o nome sugere, **All** verifica se Todos os elementos na lista satisfazem uma certa condição. Se sim, retorna verdadeiro, caso contrário, falso. Se o **All** encontrar um elemento que não satisfaz a condição, ele pára o processamento imediatamente e retorna **falso**.

```
var fruits = new[]
{
    new Fruit("Banana", 89),
    new Fruit("Apple", 51),
};

// false
var hasDenseFood = fruits.All(f => f.CaloriesPer100Gramm > 80);

record Fruit(string Name, int CaloriesPer100Gramm);
```

SequenceEquals



SequenceEquals verifica se duas sequências são iguais. Igual significa que eles possuem a mesma quantidade de entradas dentro da enumeração, bem como todos os elementos são iguais. Ele usa o comparador de igualdade padrão. Duas listas vazias também são iguais.

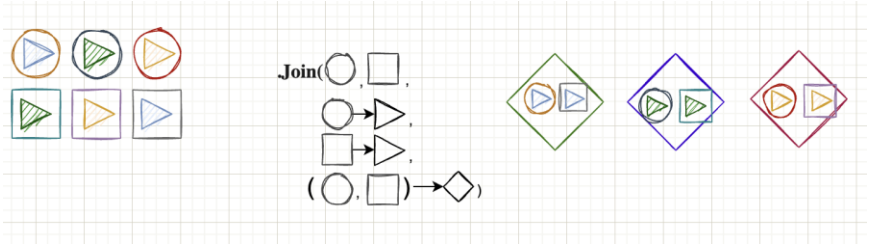
Há um segundo parâmetro opcional que permite passar um IEqualityComparer. Isso é útil se você não tem controle sobre o tipo e, portanto, não pode substituir Equals. Por padrão, os tipos de referência são comparados por suas referências entre si, o que nem sempre é o desejado.

```
var numbers = new[] { 1, 2, 3, 4 };
var moreNumbers = new[] { 1, 2, 4, 3 };

// false
var equal = numbers.SequenceEqual(moreNumbers);
```

Unificação

Este capítulo analisa operações que são responsáveis por mesclar duas ou mais enumerações em um único objeto.

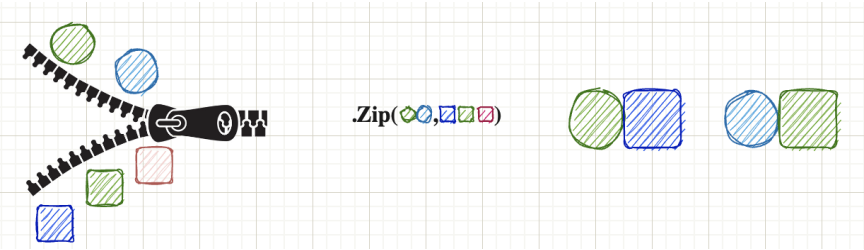
Join



Join funciona de forma semelhante a um Left-Join do **SQL**. Temos dois conjuntos que queremos unir. Os próximos dois argumentos são os "seletores de chave" de cada lista. O que o Join basicamente faz é pegar cada elemento na lista A e compará-lo com o "seletor de chave" dado contra o seletor de chave da lista B. Se ele corresponder, podemos criar um novo objeto C, que pode consistir em esses dois elementos.

```
var fruits = new[]
{
    new Fruit(1, "Banana", 89),
    new Fruit(2, "Apple", 51),
};

var classification = new[]
{
    new FruitClassification(1, "Magnesium-rich")
};

// { Name = Banana, Classification = Magnesium-rich }

var fruitWithClassification = fruits.Join(
    classification,
    f => f.FruitId, c => c.FruitId,
    (f, c) => new { f.Name, Classification = c.Classification });

foreach(var t in fruitWithClassification) Console.Write(t);

record Fruit(int FruitId, string Name, int CaloriesPer100Gramm);
record FruitClassification(int FruitId, string Classification);
```

Zip



Com o **Zip** nós "unificamos" duas listas por uma função de unificação dada. Unificamos objetos juntos até acabarmos os objetos em qualquer uma das listas. Como visto no exemplo: a primeira lista tem 2 elementos, a segunda tem 3. Portanto, o conjunto de resultados contém apenas 2 elementos.

```
var letters = new[] { "A", "B", "C", "D", "E" };
var numbers = new[] { 1, 2, 3 };
// [ "A1", "B2", "C3" ]
var merged = letters.Zip(numbers, (l, n) => l + n);
```

Elemento

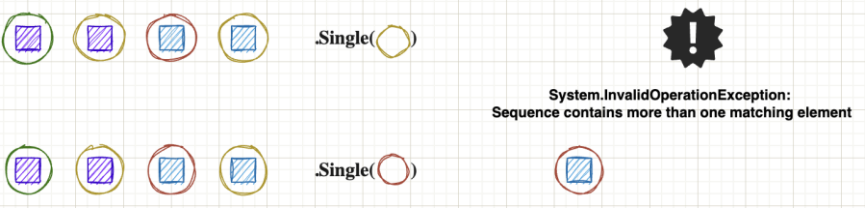
Este capítulo analisa de perto como recuperar um item específico da enumeração.

First

First retorna a primeira ocorrência de uma enumeração. Mesmo que haja outros elementos, sempre retorna imediatamente após o primeiro item encontrado. Se nenhum elemento for encontrado, ele lança uma **exceção**.

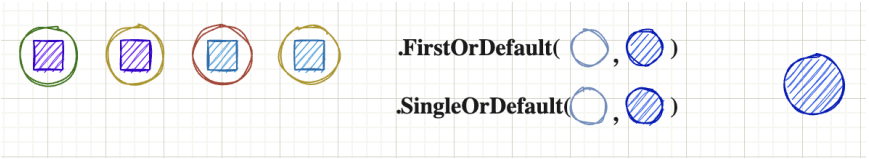
```
var people = new[]
{
    new Person("Steven", 31),
    new Person("Melissa", 32),
    new Person("Dan", 28)
};

// Person { Name: Steven, Age: 31 }
var firstOver30 = people.First(p => p.Age > 30);

record Person(string Name, int Age);
```

Single



Single não retorna imediatamente após a primeira ocorrência. A diferença para o **First** é que o **Single** garante que não há um segundo item do tipo / predicado dado. Portanto, o **Single** tem que percorrer toda a enumeração (pior caso) se conseguir encontrar outro item. Se sim, ele lança uma exceção. Se nenhum elemento for encontrado, ele lança uma **exceção**.

FirstOrDefault / SingleOrDefault

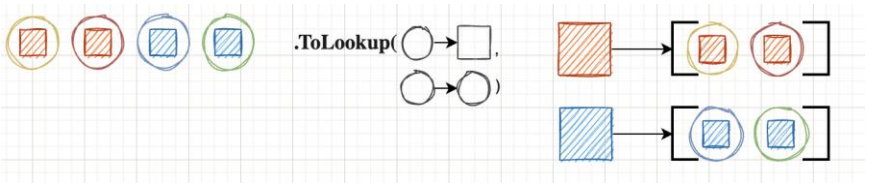



```
var people = new[]
{
    new Person("Steven", 31),
    new Person("Melissa", 32),
    new Person("Dan", 28)
};

// Person { Name: Steven, Age: 31 }
var steven = people.Single(p => p.Name == "Steven");

// This throws an exception as there are
// multiple people above 30
var above30 = people.Single(p => p.Age > 30);

record Person(string Name, int Age);
```

Se nenhum elemento for encontrado na enumeração dada, ele retorna o padrão (**null** para tipos de referência e para tipos de valor o padrão dado, como 0 para um inteiro). Desde o .NET6, podemos passar o que o "padrão" significa para nós. Portanto, podemos ter tipos de referência não anuláveis se desejarmos ou qualquer int / float / string dado.

```
var people = new[]
{
    new Person("Steven", 31),
    new Person("Melissa", 32),
    new Person("Dan", 28)
};

// null, porque o default de reference type é null
var steven = people.FirstOrDefault(p => p.Name == "Jane");

// Nós criamos um novo objeto quando não encontramos
// nenhum com idade acima de 60 anos
// Person { Name: Some Name, Age: 62 }
var above60 = people.SingleOrDefault(
    p => p.Age > 60,
    new Person("Some Name", 62)
    );

record Person(string Name, int Age);
```

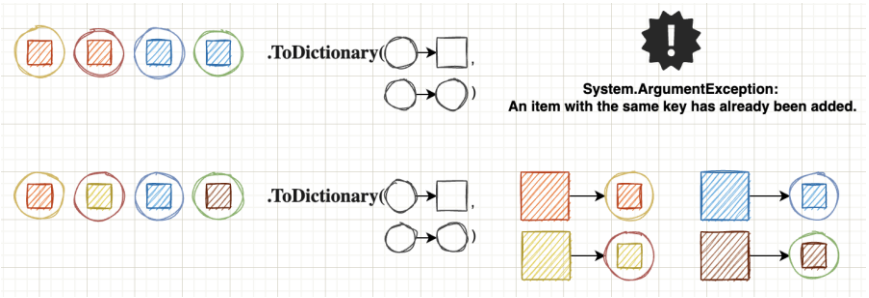
ToLookup



Este método cria um **Lookup**. Um Lookup é definido de tal forma que temos uma chave que pode apontar para uma lista de objetos (relação 1 para n). O primeiro argumento leva o "seletor de chave". O segundo seletor é o "valor". Isso pode ser o próprio objeto ou uma propriedade do próprio objeto. No final, temos uma lista de chaves distintas em que os valores compartilham essa chave exata. Um objeto LookUp é imutável. Você não pode adicionar elementos depois.

```
var products = new[]
{
    new Product("Smartphone", "Electronic"),
    new Product("PC", "Electronic"),
    new Product("Apple", "Fruit")
};

// IGrouping<string, Product>
// [
//     "Electronic": [ "Smartphone", "PC"],
//     "Apple": [ "Fruit"]
// ]
var lookup = products.ToLookup(k => k.Category, elem => elem);

record Product(string Name, string Category);
```

ToDictionary



ToDictionary funciona de forma semelhante ao ToLookup com uma diferença chave. O método ToDictionary só permite relações 1 para 1. Se dois itens compartilham a mesma chave, isso resultará em uma exceção de que a chave já está presente. Além disso, o dicionário pode ser alterado após sua criação (por exemplo, com o método Add).

```
var products = new[]
{
    new Product(1, "Smartphone"),
    new Product(2, "PC"),
    new Product(3, "Apple")
};

// IGrouping<string, Product>
// [
//     1: Product { Id: 1, Name: "Smartphone" },
//     2: Product { Id: 2, Name: "PC" },
//     3: Product { Id: 3, Name: "Apple" }
// ]
var idToProductMapping = products.ToDictionary(k => k.Id, elem =>
elem);

// Product { Id: 1, Name: "Smartphone" }
var itemWithId1 = idToProductMapping[1];

record Product(int Id, string Name);
```

ToList / ToArray

Como mencionado no início, objetos do tipo Enumerable não são avaliados diretamente, mas somente quando são materializados. Além de quantificadores como Count ou Sum, há também a possibilidade de empacotar a enumeração completa em uma coleção / array tipado (**ToArray**) ou lista (**ToList**). Com isso, criamos a enumeração na memória exatamente neste momento.

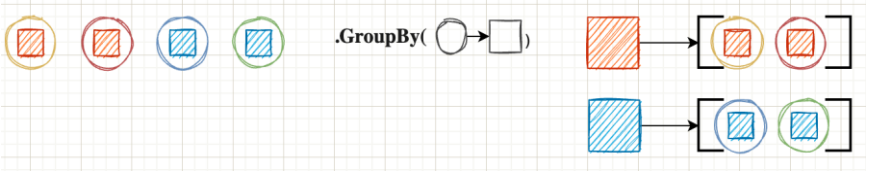
Se tomarmos o exemplo do início e chamarmos ToList diretamente, veremos que a contagem não muda mais.

```
var list = new List<int>();
list.Add(1);
list.Add(2);

var evenNumbers = list.Where(n => n % 2 == 0).ToList();

list.Add(4);
// Isso retorna apenas 1 valor pois a lista já foi materializada
Console.WriteLine($"Números pares na lista:
evenNumbers.Count()}");
```

Agrupamento

Este capítulo analisará as capacidades de agrupamento do LINQ.

GroupBy



GroupBy agrupa a enumeração por uma projeção / chave dada. Todos os elementos que compartilham essa chave exata são agrupados juntos. É quase idêntico ao **ToLookup** com uma diferença muito grande. **GroupBy** significa "Eu estou construindo um objeto para representar a pergunta 'como essas coisas seriam se eu as organizasse por grupo?" Chamando **ToLookup** significa "Eu quero um cache de tudo agora organizado por grupo“.

```
var products = new[]
{
    new Product("Smartphone", "Electronic"),
    new Product("PC", "Electronic"),
    new Product("Apple", "Fruit")
};

// GroupBy creates an IEnumerable<IGrouping<string, Product>>
// This is a big difference to ToLookup where we don't have
// the "wrapping" IEnumerable
// [
//     "Electronic": [ "Smartphone", "PC"],
//     "Apple": [ "Fruit"]
// ]
var lookup = products.GroupBy(k => k.Category, elem => elem);

record Product(string Name, string Category);
```

Set

Este capítulo analisa funções, que se comportam como conjuntos. Conjuntos são especialmente no sentido de que eles só contêm objetos distintos (disjuntos) neles.

Union

Union cria a união de duas listas com cada elemento distinto que está em ambas as suas listas. Ele se comporta como um conjunto, então os itens duplicados são removidos. Imagine que você tem ambas as listas juntas e chama Distinct.

```
var numbers1 = new[] { 1, 1, 2 };
var numbers2 = new[] { 2, 3, 4 };

// [ 1, 2, 3, 4 ]
var result = numbers1.Union(numbers2);
```

Intersect

Intersect funciona de maneira semelhante ao Union, mas agora verificamos quais elementos estão presentes nas lista A

E lista B. Somente os elementos presentes em ambas estarão no conjunto de resultados. Da mesma forma: somente os itens únicos estão na nova lista. Os duplicados são automaticamente removidos.

```
var numbers1 = new[] { 1, 1, 2 };
var numbers2 = new[] { 2, 3, 4 };

// [ 2 ]
var result = numbers1.Intersect(numbers2);
```

Exemplos do mundo real

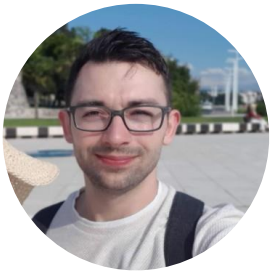
Nesta seção, você encontrará alguns exemplos da "vida real" que são mais do que apenas uma chamada de método. Consiste em exemplos executáveis do dotnetfiddle. Portanto, você pode apenas executar o exemplo ou modificá-lo à vontade.

Paginação de posts de um blog:
https://dotnetfiddle.net/hsSIPV

Colaborador mais bem pago do departamento:
https://dotnetfiddle.net/e2IfQu

Epílogo

Sobre o autor



E aí, eu sou o Steven e autor desse pequeno "livreto". Você pode me encontrar através de vários canais que eu listarei abaixo. Qualquer feedback é bem-vindo. Além disso, as novas versões virão com mais exemplos. Então, se você estiver sentindo falta de alguma coisa, que eu devo adicionar, me avise e eu atualizarei esse pequeno livro

Sobre o tradutor

Olá, eu sou o Jefferson e eu que traduzi esse livreto. Abaixo tem algumas formas de me encontrar nas internets. Qualquer feedback sobre a tradução é bem-vindo. Ficou com alguma dúvida? Posta no StackOverflow e me marca (@Jefferson-Rocha).

DEVICODE

Recursos / leituras adicionais

- Generator-Function in C# - Como o yield funciona?
- IEnumerable vs IQueryable - Qual a diferença?
- Documentação da Microsoft sobre o Enumerable

Versionamento

Versão 1.21-ptBR (2023-??-??)

- Traduzido para o português.

Versão 1.21 (2022-10-14)

- Dar nomes é difícil e por isso foi corrigida o nome de uma variável.

Versão 1.2 (2022-09-16)

- Corrigidos código e texto da introdução.

Versão 1.1 (2022-08-26)

- Corrigido links
- Ajudado método Max lançando exceção quando vazio
- Explicação sobre Aggregate para enumerados vazios.

Versão 1.0 (2022-08-25)

- Primeira pubblicação