# Linqその効果的な使い方

えムナウ（児玉宏之）
http://mnow.jp/
http://mnow.wankuma.com/
http://blogs.wankuma.com/mnow/
http://www.ailight.jp/blog/mnow/





# アジェンダ

- はじめに
- Linq to Object をながめてみる
- Linq to Object の正体
- Linq to SQL の使いどころ
- まとめ

# はじめに

- Linq のセッションや勉強会、Webでの情報も結構出てきていて、そろそろ飽きてきている方もいると思います。
- そろそろ仕事でも使ってみようとしている方も多いと思います。
- そこで今日はLinq to Object・Linq to SQLの全体像と使うときの注意点を見ていきたいと思います。

# Linq to Object をながめてみる

- Linq はあらかじめ データベースのSQL文のようなクエリ式の構文が用意されています。
- C# は、メソッドベースでしかサポートされていない機能もありますが、Visual Basic ではクエリ式の構文が用意されています。
- いっぱいあるので組み合わせて使うととても便利です。

# Linq to Object をながめてみる

| IEnumerableメソッド | C# のクエリ式の構文 | Visual Basic のクエリ式の構文 |
|---|---|---|
| Cast | from type i in numbers | From … As … |
| GroupBy | group … by | Group … By … Into … |
| | group … by … into … | |
| GroupJoin | join … in … on … equals … into … | Group Join … In … On … |
| Join | join … in … on … equals … | From x In , y In  Where x.a = y.a |
| | | Join … [As …]In … On … |
| | let … = … | Let … = … |
| OrderBy | orderby … | Order By … |
| OrderByDescending | orderby … descending | Order By … Descending |
| Select | select | Select |
| SelectMany | 複数の from 句。 | 複数の From 句。 |
| ThenBy | orderby …, … | Order By …, … |
| ThenByDescending | orderby …, … descending | Order By …, … Descending |
| Where | where | Where |

# Linq to Object をながめてみる

| IEnumerableメソッド | C# のクエリ式の構文 | Visual Basic のクエリ式の構文 |
|---|---|---|
| All | 該当なし | Aggregate … In … Into All(…) |
| Any | 該当なし | Aggregate … In … Into Any() |
| Average | 該当なし | Aggregate … In … Into Average() |
| Count | 該当なし | Aggregate … In … Into Count() |
| Distinct | 該当なし | Distinct |
| LongCount | 該当なし | Aggregate … In … Into LongCount() |
| Max | 該当なし | Aggregate … In … Into Max() |
| Min | 該当なし | Aggregate … In … Into Min() |
| Skip | 該当なし | Skip |
| SkipWhile | 該当なし | Skip While |
| Sum | 該当なし | Aggregate … In … Into Sum() |
| Take | 該当なし | Take |
| TakeWhile | 該当なし | Take While |

## Linq to Object をながめてみる

- C# って言語のサポートが少ない
  - 不便だよねぇ
  - どうせだったらフルサポートすればいいのに
  - Visual Basic っていいなぁ

Visual Basic に乗り換えちゃおうか

# ちょっと待って

# Linq to Object をながめてみる

```
string[] strings =
{
    "A penny saved is a penny earned.",
    "The early bird catches the worm.",
    "The pen is mightier than the sword.",
    "My name is M-now."
};
```

```
var earlyBirdQuery =
    from sentence in strings
    let words = sentence.Split(' ', '.')
    from word in words
    where !string.IsNullOrEmpty(word)
    let w = word.ToLower()
    orderby w
    select w;

foreach (var v in earlyBirdQuery.Distinct())
    Console.WriteLine(v);
```

# そんなに困らない

# Linq to Object をながめてみる

```
double[] value = {1.2, 3.6, 2.1, 10.5, 4.8, 6.3};
var calcQuery =
    from v in value
    orderby v
    select v;
var cutQuery = calcQuery.Skip(1).Take(calcQuery.Count() - 2);
Console.WriteLine("Count=" + cutQuery.Count());
Console.WriteLine("Sum=" + cutQuery.Sum());
Console.WriteLine("Average=" + cutQuery.Average());
Console.WriteLine("Max=" + cutQuery.Max());
Console.WriteLine("Mix=" + cutQuery.Min());
```

# 困らないよ～～

# Linq to Object をながめてみる

| Enumerableメソッド | 機能 |
|---|---|
| Concat | 2つのIEnumerableの連結 |
| Contains | 要素がIEnumerableに格納されているかどうかを判断 |
| Except | 1つめのIEnumerableから2つめのIEnumerableの要素を削除 |
| Intersect | 2つのIEnumerableの積集合 |
| OfType<Type> | Typeで指定された型の物だけ抜き出す |
| Range | 指定範囲の整数のIEnumerableを作成 |
| Repeat | 一つの要素を繰り返し作成 |
| Reverse | IEnumerableの要素の順番を反転 |
| SequenceEqual | 2つのIEnumerableが等しいか比較 |
| Union | 2つのIEnumerableの和集合 |

# Linq to Object をながめてみる

| Enumerableメソッド | 機能 |
|---|---|
| DefaultIfEmpty | IEnumerableが空でなければそのまま、空ならデフォルト値 |
| ElementAt | インデックス位置にある要素 |
| ElementAtOrDefault | インデックス位置にある要素、空ならデフォルト値 |
| Empty | 空のIEnumerable |
| First | 先頭の要素 |
| FirstOrDefault | 先頭の要素、空ならデフォルト値 |
| Last | 最後の要素 |
| LastOrDefault | 最後の要素、空ならデフォルト値 |
| Single | 要素が一つか確認し取り出す |
| SingleOrDefault | 要素が一つか確認し取り出す、空ならデフォルト値 |
| ToArray | Arrayに変換する |
| ToDictionary | Dictionaryに変換する |
| ToList | Listに変換する |
| ToLookup | Lookupに変換する |

# Linq to Object をながめてみる

- デフォルト値って？
  - 数値の場合は0です。
  - bool は false です。
  - 参照型は null です。

# Linq to Object をながめてみる

- For ループの置き換え
  - でも foreach でダンプしちゃつまんないけどね

```
delegate T Y<T>(Y<T> y);

Y<Func<Func<Func<int, int>, Func<int, int>>, Func<int, int>>> Y =
        y => f => x => f(y(y)(f))(x);

Func<Func<int, int>, Func<int, int>> g =
        f => x => x == 0 ? 1 : x * f(x - 1);
var fact = from i in Enumerable.Range(1, 10) select Y(Y)(g)(i);
foreach (var i in fact) Console.WriteLine(i);
```

# Linq to Object をながめてみる

- まとめ
  - データベースのSQL文のようなクエリ式の構文が用意
  - Visual Basic は、言語サポートがしっかりしている、C# はそんなに困らない程度のサポート
  - 色々と組み合わせて使うととても便利

# Linq to Object の正体

- どちらが簡単で分かりやすくてなおかつ速いのでしょうか？

A)

```
var accounts = from a in al
               where a.ZipCode == "168-0064"
               select new { Name = a.Name, ZipCode = a.ZipCode };
foreach (var account in accounts)
    Console.WriteLine(account.Name + "(" + account.ZipCode + ")");
```

B)

```
foreach (var a in al)
    if (a.ZipCode == "168-0064")
        Console.WriteLine(a.Name + "(" + a.ZipCode + ")");
```

# Linq to Object の正体

- どちらが速いでしょうか？

A)
```
static int[] cal1(int[] arr)
{
    int sum = 0;
    int count = 0;
    int max = int.MinValue;
    int min = int.MaxValue;
    foreach (int a in arr)
    {
        sum += a;
        count++;
        if (max < a) max = a;
        if (min > a) min = a;
    }
    return new int[] { sum, count, max, min };
}
```

B)
```
static int[] cal2(int[] arr)
{
    return new int[] {
        arr.Sum(),
        arr.Count(),
        arr.Max(),
        arr.Min()
    };
}
```

# Linq to Object の正体

- Linq to Object は、IEnumerable<T> によるパイプラインという見方もできます。
- IEnumerable<T> ですから、Linq to Object のメソッドは、一般に foreach を内部で使っていますので、あまり高速性能を発揮できません。

# Linq to Object の正体

- 例えば右のメソッドは foreach 何回ループするでしょうか？
- Sum、Count、Max、Minが別々のパイプラインです。
- ４回の foreach を内部で行なうことになります。

```
static int[] cal2(int[] arr)
{
    return new int[] {
        arr.Sum(),
        arr.Count(),
        arr.Max(),
        arr.Min()
    };
}
```

# Linq to Object をながめてみる

- 同じ Linq の foreach を何回もやるんであれば ToArray しておくのがお勧め

10000 x 1000 ループで、

当社比なんと52倍

# Linq to Object の正体

- まとめ
  - Linq to Object は、IEnumerable<T> によるパイプライン
  - 便利に使うことができる反面、メソッドの内部で使用する foreach の回数に注意を払わないと、効率が悪い場合もある
  - 同じ Linq の foreach を何回もやるんであれば ToArray

## Linq to SQL の使いどころ

- ＜N氏談＞

すくなくとも今のところ大規模案件で**使う気はない**です。

大規模案件で**使う気はない**です。

**使う気はないです**

**使う気はない**

# Linq to SQL の使いどころ

- Linq は Query ですが Linq to SQL の母体になる DataContext は Dataset より進化しています。
  - Row は INotifyPropertyChanging や INotifyPropertyChanged を実装したObjectです。
  - 当然 Insert Update Delete ストアド も使えます。
  - あらかじめ必要な partial method が仕込まれています。
  - 同時実行制御で競合の解決がサポートされています。

# Linq to SQL の使いどころ

- データの一部を表示する場合
  - コンボボックスに表示するだけなのに、Dataset DataTable DataRow みたいにフルセットのインスタンスなんか必要なの？
  - コンボボックスに表示するだけなのに、一々ストアド作んなきゃいけないの？
  - Linq to SQL なら 匿名型を使って軽いコンボボックス用のインスタンスを作れる
  - Linq to SQL なら ストアドに対応することもできる

# Linq to SQL の使いどころ

- たったこれだけのソースで、WPFのコンボボックスの選択可能なデータがセットできます。

```
using (NWDataContext context = new NWDataContext())
{
    var customerComboBoxQuery =
        from customer in context.Customers
        select new { ID = customer.CustomerID, Name = customer.CompanyName };
    this.comboBox1.ItemsSource = customerComboBoxQuery;
    this.comboBox1.SelectedValuePath = "ID";
    this.comboBox1.DisplayMemberPath = "Name";
}
```

# Linq to SQL の使いどころ

- 同時実行制御がすごいことになっています
  - Timestamp 列のあるなしで自動的に最適な楽観的同時実行制御のコードを作ってくれます。
  - 同時実行制御で競合の解決がサポートされています。

# Linq to SQL の使いどころ

- タイムスタンプと同時実行制御
  - タイムスタンプがないテーブル
    WHERE ([ID] = @p0) AND ([TEXT1] = @p1) AND ([TEXT2] = @p2)
  - タイムスタンプがあるテーブル
    WHERE ([ID] = @p0) AND ([timestamp] = @p1)
    timestampの読み直しもきちんと行われています。
  - （Datasetでもちゃんとやってます）

# Linq to SQL の使いどころ

- 同時実行制御で競合の解決
  - 競合している行単位で競合の解決
    - 最新を取得してユーザーには再入力をしてもらうことになります。

```
conflict.Resolve(Data.Linq.RefreshMode.OverwriteCurrentValues)
```

    - 元の値とユーザーが変更した値で更新が行われ競合を発生させた他者の変更は上書きされます。

```
conflict.Resolve(Data.Linq.RefreshMode.KeepCurrentValues)
```

    - 他者の変更とユーザーの変更をマージします。両方が変更した列はユーザーの変更を優先します。

```
conflict.Resolve(Data.Linq.RefreshMode.KeepChanges)
```

# Linq to SQL の使いどころ

- 同時実行制御で競合の解決
  - 競合している行ごとの列単位で競合の解決
    - 最新を取得してユーザーには再入力をしてもらうことになります。

member.Resolve(Data.Linq.RefreshMode.OverwriteCurrentValues)

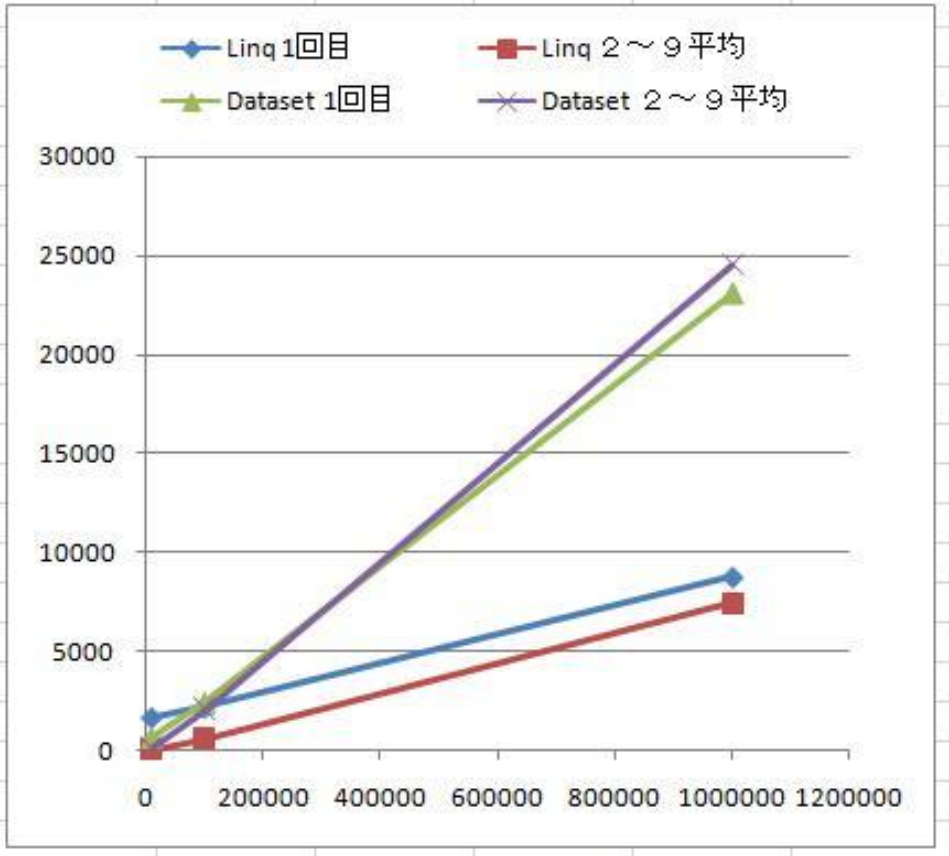
    - 元の値とユーザーが変更した値で更新が行われ競合を発生させた他者の変更は上書きされます。

member.Resolve(Data.Linq.RefreshMode.KeepCurrentValues)

# Linq to SQL の使いどころ

- 遅いんじゃないの？
  - 件数によっては速いです

| | | 10000 | 100000 | 1000000 | ctor+Open |
|---|---|---|---|---|---|
| Linq | 1回目 | 1637 | 2152 | 8764 | 619 |
| Linq | 2～9平均 | 61 | 629 | 7446 | |
| Dataset | 1回目 | 687 | 2429 | 23056 | 474 |
| Dataset | 2～9平均 | 168 | 2089 | 24489 | |



# Linq to SQL の使いどころ

- さらに速くする方法もあります
  - CompiledQuery.Compile

```
var query = CompiledQuery.Compile(
    (LinqTestDataContext db) => from a in db.Table_1 select a
    );
```

使い方：foreach (var a in query(context))

  - context.ObjectTrackingEnabled = false;
    - データの変更を追跡しない Query にみの場合に利用できます。

# Linq to SQL の使いどころ

- まとめ
  - DataContext は Dataset より進化しています。
  - データの一部を表示する場合は楽です。
  - 同時実行制御で競合の解決もできます。
  - 速くするための方法もあります。

  - なにより、長年の念願であった「Object」なんです。
  - Linq to SQL を使って自由を満喫しましょう。

## まとめ

- Linq to Object は、IEnumerable<T> によるパイプライン色々と組み合わせて使うととても便利
- Linq to SQLは、Dataset より進化していて、長年の念願であった「Object」

- 利点と欠点を理解して上手に使っていきましょう。