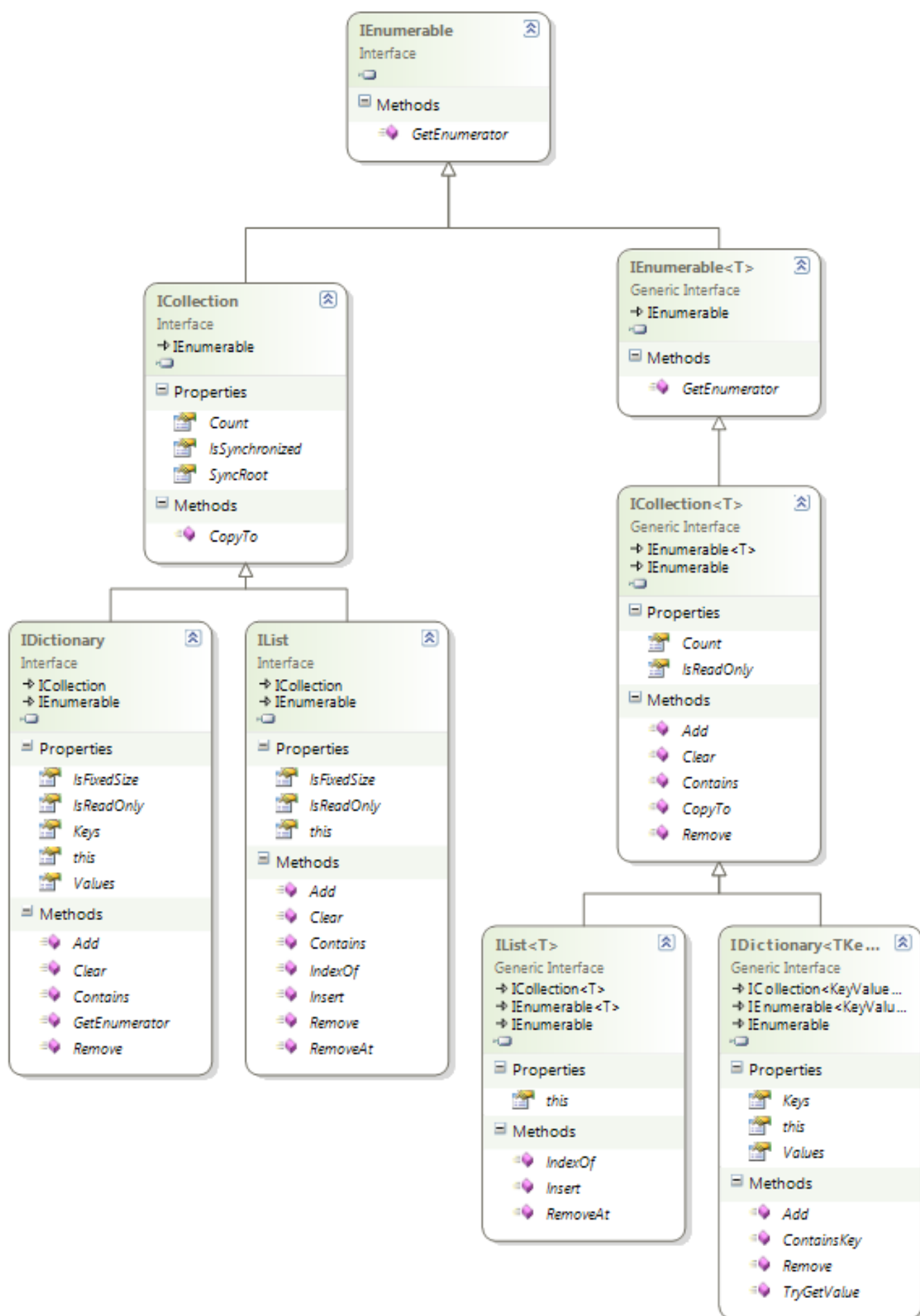


در این مقاله می‌خواهیم نحوه ساخت اشیایی با خصوصیات Enumerable را بررسی کنیم. بررسی ویژگی این اشیاء دارای اهمیت است حداقل به این دلیل که پایه یکی از قابلیت مهم زبانی سی‌شارپ یعنی LINQ هستند. برای یافتن پیش‌زمینه‌ای در این موضوع خواندن این مقاله‌های بسیار خوب ([۱](#) و [۲](#)) نیز توصیه می‌شود.

Enumerableها

اشیاء Enumerable یا به عبارت دیگر اشیایی که اینترفیس IEnumerable را پیاده‌سازی می‌کنند، دامنه گسترده‌ای از Collection‌های [CLI](#) را شامل می‌شوند. همانطور که در نمودار زیر نیز می‌توانید مشاهده کنید IEnumerable (از نوع غیر Generic آن) در بالای سلسله مراتب اینترفیس‌های Collection‌های CLI قرار دارد:



درخت اینترفیس‌های Collection ها در CLI [منبع](#)

IEnumerator ها همچنین دارای اهمیت دیگری نیز هستند؛ قابلیت‌های LINQ که از دات‌نت ۳.۵ به دات‌نت اضافه شدند به‌عنوان Extension های این اینترفیس تعریف شده‌اند و پیاده‌سازی Linq to Objects را می‌توانید در کلاس استاتیک System.Linq.Enumerable در System.Core مشاهده کنید. (می‌توانید برای دیدن آن را با ILDasm یا Reflector باز کنید یا پیاده‌سازی آزاد آن در پروژه Mono را [اینجا](#) مشاهده کنید که برای شناخت بیشتر LINQ واقعاً مفید است).

همچنین این Enumerable ها هستند که foreach را امکان‌پذیر می‌کنند. به عبارتی دیگر هر شی‌ای که قرار باشد در (var x) foreach (in object) قرار بگیرد و بدین طریق اشیاء درونی‌اش را برای پیمایش یا عملی خاص قرار دهد باید Enumerable باشد.

همانطور که قبلاً هم اشاره شد IEnumerator از نوع غیر Generic در بالای نمودار Collection ها قرار دارد و حتی IEnumerable از نوع Generic نیز باید آن را پشتیبانی کند. این موضوع به احتمال به این دلیل در طراحی لحاظ شد که مهاجرت به NET 2.0 قابلیت‌های Generic را افزوده بود ساده‌تر کند. IEnumerable همچنین قابلیت covariance که از قابلیت‌های جدید C# 4.0 هست را دارا است (در اصل IEnumerator دارای Generic از نوع [out](#) است).

Enumerable ها همانطور که از اسم اینترفیس IEnumerable انتظار می‌رود اشیایی هستند که می‌توانند یک شی Enumerator که IEnumerator را پیاده‌سازی کرده‌است را از خود ارائه دهند. پس طبیعی است برای فهم و درک دلیل وجودی Enumerable باید Enumerator را بررسی کنیم.

Enumerator ها

Enumerator شی است که در یک پیمایش یا به‌عبارت دیگر گذر از روی تک‌تک اعضا ایجاد می‌شود که با حفظ موقعیت فعلی و پیمایش امکان ادامه پیمایش را برای ما فراهم می‌آورد. اگر بخواهید آن را در حقیقت بازسازی کنید شی Enumerator به‌مانند کاغذ یا جسمی است که بین صفحات یک کتاب قرار می‌دهید که مکانی که در آن قرار دارید را گم نکنید؛ در این مثال، Enumerable همان کتاب است که قابلیت این را دارد که برای پیمایش به وسیله قرار دادن یک جسم در وسط آن را دارد.

حال برای اینکه دید بهتری از رابطه بین Enumerable و Enumerator از نظر برنامه‌نویسی به این موضوع پیدا کنیم یک کد نمونه عملی را بررسی می‌کنیم.

در اینجا نمونه ساده و خوانایی از استفاده از یک List برای پیمایش تمامی اعداد قرار دارد:

```
List<int> list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);
foreach (int i in list)
{
    Console.WriteLine(i);
}
```

همانطور که قبلاً اشاره foreach نیاز به یک Enumerable دارد و List هم با پیاده‌سازی IList که گسترشی از IEnumerable هست نیز یک نوع Enumerable هست. اگر این کد را Compile کنیم و IL آن را بررسی کنیم متوجه می‌شویم که CLI در اصل چنین کدی را برای اجرا می‌بینید:

```
List<int> list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);
IEnumerator<int> listIterator = list.GetEnumerator();
while (listIterator.MoveNext())
{
    Console.WriteLine(listIterator.Current);
}
```

```

}
listIterator.Dispose();

```

(می‌توان از using استفاده نمود که Dispose را خود انجام دهد که اینجا برای سادگی استفاده نشده‌است).

همانطور که می‌بینیم یک Enumerator برای Enumerable ما (یعنی List) ایجاد شد و پس از آن با پرسش این موضوع که آیا این پیمایش امکان ادامه دارد، کل اعضا پیموده‌شده و عمل مورد نظر ما بر آن‌ها انجام شده‌است.

خب، تا اینجا کار با خصوصیات و اهمیت Enumeratorها و Enumerableها آشنا شدیم، حال نوبت به آن می‌رسد که بررسی کنیم آن‌ها را چگونه می‌سازند و بعد از آن با کاربردهای فراتری از آن‌ها نسبت به پیمایش یک List آشنا شویم.

ساخت Enumeratorها و Enumerableها

همانطور که اشاره شد ایجاد اشیاء Enumerable به اشیاء Enumerator مربوط است، پس ما در یک قطعه کد که پیمایش از روی یک آرایه را فراهم می‌آورد ایجاد هر دوی آن‌ها و رابطه بینشان را بررسی می‌کنیم.

```

public class ArrayEnumerable<T> : IEnumerable<T>
{
    private T[] _array;
    public ArrayEnumerable(T[] array)
    {
        _array = array;
    }

    public IEnumerator<T> GetEnumerator()
    {
        return new ArrayEnumerator<T>(_array);
    }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

public class ArrayEnumerator<T> : IEnumerator<T>
{
    private T[] _array;
    public ArrayEnumerator(T[] array)
    {
        _array = array;
    }

    public int index = -1;

    public T Current { get { return _array[index]; } }

    object System.Collections.IEnumerator.Current { get { return this.Current; } }

    public bool MoveNext()
    {
        index++;
        return index < _array.Length;
    }

    public void Reset()
    {
        index = 0;
    }

    public void Dispose() { }
}

```

[ادامه](#)

نظرات خوانندگان

نویسنده: مرتضی

تاریخ: ۱۳۹۱/۰۵/۱۷ ۱۹:۲۰

درخت اینترفیس‌های Collection ها در سی‌شارپ منبع: <http://www.mbaldinger.com/post/NET-Collection-Interface-Hierarchy.aspx>

بجای سی‌شارپ به دانت نت تغییرش بدید
درخت اینترفیس‌های Collection ها در دانت نت

نویسنده: ابراهیم بیاگوی

تاریخ: ۱۳۹۱/۰۵/۱۷ ۱۹:۲۹

من شخصاً اطمینان ندارم که [همهٔ زبان‌های CLI](#) از همین Collection ها استفاده کنند و البته این نمودار با Syntax سی‌شارپ بود
به همین دلیل سی‌شارپ نوشته بودم با این حال آن را به Collection های CLI تبدیل کردم.

در مطلب قبل متوجه شدیم که Enumerable و Enumerator چه چیزی هستند و آن‌ها را چگونه می‌سازند. در انتهای آن مطلب نیز قطعه کدی وجود داشت که در آن دیدیم چگونه یک شیء Enumerable می‌تواند در عملیاتی نسبتاً پیچیده یک شیء Enumerator ایجاد کند.

حال می‌خواهیم قابلیت زبانی‌ای را بررسی کنیم که در اصل مشابه همین کاری که ما انجام دادیم یعنی ایجاد شیء جداگانه‌ی Enumerator و برگرداندن یک نمونه از آن در زمانی که ما GetEnumerator را از Enumerable مان فراخوانی می‌کنیم را انجام می‌دهد.

yield و نحوه پیاده‌سازی آن

در اینجا قطعه کدی قرار دارد که در اصل جایگزین دو کلاسی است که در انتهای مطلب قبل قرار داشت که به کمک قابلیت yield آن را بازنویسی کرده‌ایم:

```
public class ArrayEnumerable<T> : IEnumerable<T>
{
    T[] _array;
    public ArrayEnumerable(T[] array)
    {
        _array = array;
    }

    public IEnumerator<T> GetEnumerator()
    {
        int index = 0;
        while (index < _array.Length)
        {
            yield return _array[index];
            index++;
        }
        yield break;
    }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

yield break در اینجا مانند return در یک تابع/متد با نوع خروجی void اضافی است و فقط برای آشنایی با syntax دومی که yield در سی‌شارپ پشتیبانی می‌کند قرار داده شده است)

همانطور که می‌بینیم کد قبلی ما به مقدار بسیاری ساده‌تر و خواناتر شد و برای فهم آن کافی است که مفهوم yield را بدانیم.

yield به معنای برآوردن یا ارائه کردن کلید واژه‌ای است که می‌توان آن را اینگونه تصور کرد که با هر بار صدا زده شدن کد را متوقف می‌کند و نتیجه‌ای را برمی‌گرداند و با درخواست ما برای ادامه کار (با MoveNext) کار خود را از همان جای متوقف شده ادامه می‌دهد.

حالا اگر کمی دقیق‌تر باشیم سوالی که باید برای ما پیش بیاید این است که آیا [CLR](#) خود yield را پشتیبانی می‌کند؟ این قطعه کدی است که با کمک بازگردانی مجدد همین کلاس به زبان سی‌شارپ دیده می‌شود:

```
public class ArrayEnumerable<T> : IEnumerable<T>, IEnumerable
{
    // Fields
    private T[] _array;
```

```
// Methods
public ArrayEnumerable(T[] array)
{
    this._array = array;
}

public IEnumerator<T> GetEnumerator()
{
    return new <GetEnumerator>d__0(0);
}

IEnumerator IEnumerable.GetEnumerator()
{
    return this.GetEnumerator();
}

// Nested Types
[CompilerGenerated]
private sealed class <GetEnumerator>d__0 : IEnumerator<T>, IEnumerator, IDisposable
{
    // Fields
    private int <>1__state;
    private T <>2__current;
    public ArrayEnumerable<T> <>4__this;
    public int <index>5__1;

    // Methods
    [DebuggerHidden]
    public <GetEnumerator>d__0(int <>1__state)
    {
        this.<>1__state = <>1__state;
    }

    private bool MoveNext()
    {
        switch (this.<>1__state)
        {
            case 0:
                this.<>1__state = -1;
                this.<index>5__1 = 0;
                while (this.<index>5__1 < ArrayEnumerable<T>._array.Length)
                {
                    this.<>2__current = ArrayEnumerable<T>._array[this.<index>5__1];
                    this.<>1__state = 1;
                    return true;
                }
                Label_0050:
                this.<>1__state = -1;
                this.<index>5__1++;
                break;

            case 1:
                goto Label_0050;
        }
        return false;
    }

    [DebuggerHidden]
    void IEnumerator.Reset()
    {
        throw new NotSupportedException();
    }

    void IDisposable.Dispose()
    {
    }

    // Properties
    T IEnumerator<T>.Current
    {
        [DebuggerHidden]
        get
        {
            return this.<>2__current;
        }
    }

    object IEnumerator.Current
    {
        [DebuggerHidden]

```

```

        get
        {
            return this.<>2__current;
        }
    }
}

```

(توجه: برای خواندن این کد، <...ها را نادیده بگیرید، اینها هیچ وظیفه خاصی ندارند و کار خاصی نمی‌کنند)
این کد را که البته چندان خوانا نیست اگر با کد انتهای مطلب قبل مقایسه کنید متوجه می‌شوید که دارای اشتراک‌هایی است. در آن مثال نیز شیء Enumerable یک شیء جداگانه بود (در اینجا یک [کلاس درونی](#) است) که هنگامی که GetEnumerator را صدا می‌زدیم نمونه‌ای از آن ایجاد می‌شد و بازگردانیده می‌شد.

در این کد کامپایلر وضعیت‌های مختلفی که برای توقف و ادامه کار MoveNext که مهم‌ترین بخش کد هست را با کمک ترکیبی از switch case و goto پیاده‌سازی کرده‌است که با کمی دقت می‌توانید متوجه منطق آن شوید :

ممکن است به نظرتان برسد که این قطعه کد از نظر (حداقل نامگذاری) در سی‌شارپ صحیح نیست. اینگونه نامگذاری‌ها که از نظر CLR (و زبان IL) درست ولی از نظر زبان سطح بالا نادرست هستند باعث می‌شوند که از هرگونه برخورد نامی احتمالی با نام‌های معتبر تعریف شده توسط کاربر جلوگیری شود.

احتمالاً اگر پیش‌زمینه نسبت به این مطلب داشته باشید با خود خواهید گفت که «این که واضح بود، اصلاً وظیفه ماشین در سطح پایین نیست که چنین عملی را پشتیبانی کند». واضح‌بودن این موضوع برای شما شاید به این دلیل باشد که پیاده‌سازی yield را قبلاً جای دیگری ندیده‌اید. برای درک این مطلب در اینجا نحوه پیاده‌سازی yield را در پایتون بررسی می‌کنیم.

```

def array_iterator(array):
    length = len(array)
    index = 0
    while index < length:
        yield array[index]
        index = index + 1

```

اگر کد مفسر پایتون را برای این [generator](#) بررسی کنیم متوجه می‌شویم که پایتون دارای عملگر خاصی در سطح ماشین برای yield است:

```

>>> import dis
>>> dis.dis(array_iterator)
2          0 LOAD_GLOBAL           0 (len)
          3 LOAD_FAST              0 (array)
          6 CALL_FUNCTION          1
          9 STORE_FAST             1 (length)

3          12 LOAD_CONST           1 (0)
          15 STORE_FAST             2 (index)

4          18 SETUP_LOOP          35 (to 56)
>>         21 LOAD_FAST              2 (index)
          24 LOAD_FAST              1 (length)
          27 COMPARE_OP            0 (<)
          30 POP_JUMP_IF_FALSE    55

5          33 LOAD_FAST              0 (array)
          36 LOAD_FAST              2 (index)
          39 BINARY_SUBSCR
          40 YIELD_VALUE
          41 POP_TOP

6          42 LOAD_FAST              2 (index)
          45 LOAD_CONST           2 (1)
          48 BINARY_ADD
          49 STORE_FAST             2 (index)
          52 JUMP_ABSOLUTE        21
>>         55 POP_BLOCK
>>         56 LOAD_CONST           0 (None)
          59 RETURN_VALUE

```


همانطور که می‌بینیم پایتون دارای عملگر خاصی برای پیاده‌سازی yield بوده و به مانند سی‌شارپ از قابلیت‌های قبلی ماشین برای پیاده‌سازی yield استفاده نکرده‌است.

yield و iteratorها قابلیت‌های زیادی را در اختیار برنامه‌نویسان قرار می‌دهند. برنامه‌نویسی async یکی از این قابلیت‌هاست. پیوندهای ابتدای مقاله اول را در این زمینه مطالعه کنید (البته با ورود دات‌نت ۴.۵ شیوه دیگری نیز برای برنامه‌نویسی async ایجاد شده). از قابلیت‌های دیگر طراحی ساده یک ماشین حالت است.

کد زیر ساده‌ترین حالت یک ماشین حالت را نمایش می‌دهد که به کمک قابلیت yield ساده‌تر پیاده‌سازی شده‌است:

```
public class SimpleStateMachine : IEnumerable<bool>
{
    public IEnumerator<bool> GetEnumerator()
    {
        while (true)
        {
            yield return true;
            yield return false;
        }
    }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

(البته استفاده اینگونه از yield (در حلقه بی‌نهایت) خطرناک است و ممکن است برنامه‌تان را در اثر بی‌دقتی قفل کنید، حداقل به همین دلیل بهتر است همیشه چنین اشیائی دارای محدودیت باشند). می‌توانید از SimpleStateMachine به این شکل استفاده کنید:

```
new SimpleStateMachine().Take(20).ToList().ForEach(x => Console.WriteLine(x));
```

که ۲۰ حالت از این ماشین حالت را چاپ خواهد کرد که البته اگر Take را قرار نمی‌دادیم برنامه را قفل می‌کرد.

نظرات خوانندگان

نویسنده: متفکر

تاریخ: ۱۳۹۱/۰۵/۱۹ ۱۰:۴۲

سلام... کلاس SimpleStateMachine (برخلاف نامش) Simple نیست و حتی Error-Prone هستش. اگر کلاس رو بدین شکل در نظر بگیریم:

```
public class ReallySimple<T> : IEnumerable<T>
{
    //blah blah blah...
}
```

در این صورت می‌تونیم با استفاده از Range همون کار رو انجام بدیم:

```
Enumerable.Range(1, 20).Select(r => new ReallySimple()).ToList().ForEach(x => Console.WriteLine(x));
```

نویسنده: ابراهیم بیاگوی

تاریخ: ۱۳۹۱/۰۵/۱۹ ۱۳:۴

به Error-Prone آن در خود مقاله هم اشاره شده. SimpleStateMachine که ما قصد پیاده‌سازی داشته‌ایم تقریباً به این حالت است: [منبع](#)

که می‌توان بین حرکت‌ها یک Action قرار داد. (که هدف ما بیشتر همین هست)
به طور کلی می‌گویند که State-Machine درست‌کردن با این قابلیت [چندان درست نیست](#)، فقط امکان‌پذیر هست.