

عنوان: مقایسه بین حلقه های تکرار (foreach و for و Lambda ForEach)

نویسنده: مسعود پاکدل

تاریخ: ۲۳:۴۰ ۱۳۹۲/۰۳/۰۵

آدرس: www.dotnettips.info

برچسب‌ها: C#, foreach, Performance

به حلقه‌های تکرار زیر دقت کنید.

#1 حلقه for با استفاده از متغیر Count لیست

```
var ListOfNumber = new List<int>() { 100, 200, 300 , 400 , 500 };
for ( int i = 0 ; i < ListOfNumber.Count ; i++ )
{
    Console.WriteLine( ListOfNumber[i] );
}
```

#2 حلقه for با استفاده از متغیر یا مقدار صریح

```
var ListOfNumber = new List<int>() { 100, 200, 300 , 400 , 500 };
for ( int i = 0 ; i < 5 ; i++ )
{
    Console.WriteLine( ListOfNumber[i] );
}
```

#3 foreach ساده که احتمالا خیلی از شماها از اون استفاده می‌کنید.

```
var ListOfNumber = new List<int>() { 100, 200, 300 , 400 , 500 };
foreach ( var number in ListOfNumber )
{
    Console.WriteLine( number );
}
```

#4 Lambda ForEach که مورد علاقه بعضی‌ها از جمله خود من است.

```
var ListOfNumber = new List<int>() { 100, 200, 300 , 400 , 500 };
ListOfNumber.ForEach( number =>
{
    Console.WriteLine( number );
});
```

به نظر شما حلقه‌های بالا از نظر کارایی چه تفاوتی با هم دارند؟

تمام حلقه‌های بالا یک خروجی رو چاپ خواهند کرد ولی اگر فکر می‌کنید که هیچ تفاوتی ندارند سخت در اشتباه هستید.

هر 4 حلقه تکرار بالا رو در 21 حالت مختلف با شریط یکسان در یک سیستم تست کردیم و نتایج زیر حاصل شد.(منظور از نتایج مدت زمان اجرای هر حلقه است)

تعداد تکرار	#1 for با استفاده از متغیر Count لیست	#2 for-استفاده از متغیر	#3 foreach	#4 Lambda ForEach
1000	0.000008	0.000007	0.000014	0.000012
2000	0.000014	0.000013	0.000026	0.000022
3000	0.000019	0.000016	0.000036	0.000028
4000	0.000024	0.000022	0.000047	0.000035
5000	0.000029	0.000025	0.000058	0.000043
10,000	0.000059	0.000047	0.000117	0.000081

#4 Lambda ForEach	#3 foreach	#2-for استفاده از متغیر	#1 for با استفاده از متغیر Count لیست	تعداد تکرار
0.000161	0.000225	0.000093	0.000128	20,000
0.000233	0.000336	0.000141	0.000157	30,000
0.000310	0.000442	0.000180	0.000221	40,000
0.000307	0.000553	0.000236	0.000263	50,000
0.000773	0.001103	0.000443	0.000530	100,000
0.001531	0.002194	0.000879	0.001070	200,000
0.002308	0.003281	0.001345	0.001641	300,000
0.003083	0.004388	0.001783	0.002233	400,000
0.003873	0.005521	0.002244	0.002615	500,000
0.007767	0.011072	0.004520	0.005303	1,000,000
0.015536	0.022127	0.009074	0.010543	2,000,000
0.023268	0.033186	0.013569	0.015738	3,000,000
0.031188	0.044335	0.018113	0.021039	4000,000
0.038793	0.055521	0.022593	0.026280	5000,000
0.078482	0.111517	0.046090	0.052528	10,000,000

بررسی نتایج :

سریع ترین حلقه تکرار حلقه for با استفاده از متغیر معمولی به عنوان تعداد تکرار حلقه است. رتبه دوم برای حلقه for همراه با استفاده از خاصیت Count لیست مورد نظر بوده است. دلیلش هم اینه که سرعت دستیابی کامپایلر به متغیرهای معمولی حتی تا 3 برابر سریع تر از دسترسی به متد get خاصیت هاست. مهم ترین نکته این است که Lambda ForEach عملکردی بسیار بهتری نسبت به foreach معمولی داره.

پس هر گاه قصد اجرای حلقه ForEach رو برای لیست دارید و سرعت اجرا هم براتون اهمیت داره بهتره که از Lambda ForEach استفاده کنید. حالا به کد زیر دقت کنید:

```
int[] arrayOfNumbers = new int[] { 100 , 200 , 300 , 400 , 500 };
Array.ForEach<int>( arrayOfNumbers, ( int counter ) => { Console.WriteLine( counter ); } );
```

من همون حلقه بالا رو به صورت آرایه پیاده سازی کردم و برای اجرای حلقه از دستور Array.ForEach که عملکردی مشابه با List.ForEach داره استفاده کردم که نتیجه به دست اومده نشون داد که Array.ForEach از نظر سرعت به مراتب از foreach معمولی کندتر عمل می کنه. دلیلش هم اینه که کامپایلر هنگام کار با آرایه ها و اجرای اون ها به صورت حلقه، کد IL خاصی رو تولید می کنه که مخصوص کار با آرایه هاست و سرعت اون به مراتب از سرعت کد IL تولید شده برای IEnumerator ها پایین تره.

نظرات خوانندگان

نویسنده: قاسم کشاورز حداد
تاریخ: ۱۹:۲۹ ۱۳۹۲/۰۳/۰۶

خیلی برام جالب بود، ممنون از مطلب

نویسنده: محمد رعیت پیشه
تاریخ: ۲۳:۵۱ ۱۳۹۲/۰۳/۰۶

ممنون از مطلبتون.
فقط در صورت امکان توضیحی هم درباره نحوه تست کردن چنین دستوراتی بدید.

نویسنده: شاهین کیاست
تاریخ: ۸:۴۸ ۱۳۹۲/۰۳/۰۷

یک روش ساده :

```
Stopwatch sw = Stopwatch.StartNew();
var ListOfNumber = new List<int>() { 100, 200, 300 , 400 , 500 };
for ( int i = 0 ; i < ListOfNumber.Count ; i++ )
{
    Console.WriteLine( ListOfNumber[i] );
}
sw.Stop();
Console.WriteLine("Total time (ms): {0}", (long) sw.ElapsedMilliseconds);
```

نویسنده: مصطفی عسگری
تاریخ: ۲۳:۳ ۱۳۹۲/۰۳/۰۷

جالب بود که این روش از foreach سریعتر عمل میکنه

نویسنده: یوسف نژاد
تاریخ: ۰:۳۳ ۱۳۹۲/۰۳/۰۸

یا استفاده از [Microbenchmark](#) برای دریافت نتایج دقیقتر.

نویسنده: یوسف نژاد
تاریخ: ۰:۴۱ ۱۳۹۲/۰۳/۰۸

متد ForEach در کلاس List از حلقه for معمولی استفاده میکنه و نه foreach:

```
public void ForEach(Action<T> action)
{
    if (action == null)
        ThrowHelper.ThrowArgumentNullException(ExceptionArgument.match);
    for (int index = 0; index < this._size; ++index)
        action(this._items[index]);
}
```

متد Array.ForEach هم از روشی مشابه استفاده کرده:

```
public static void ForEach<T>(T[] array, Action<T> action)
```

```
{
    if (array == null)
        throw new ArgumentNullException("array");
    if (action == null)
        throw new ArgumentNullException("action");
    for (int index = 0; index < array.Length; ++index)
        action(array[index]);
}
```

foreach به دلیل استفاده از اشیای درون IEnumerable و در نتیجه اجرای دستورات بیشتر در هر حلقه کندتر عمل میکند. اما! اگر هدف تنها بررسی سرعت اجرای حلقه‌های اشاره شده باشد متدهای بالا نتیجه درستی نشان نخواهد داد، چون عملیات انجام شده در حلقه‌های نشان داده شده با هم دقیقاً یکسان نیست. بهتره که به عملیات ثابت و مستقل از متغیرهای درگیر استفاده بشه تا نتایج دقیقتری بدست بیاد. مثلاً به چیزی مثل اکشن زیر:

```
() => { int a = 1; }
```

بهتره تو این تستها مشخصات دقیق سخت افزاری هم ارائه بشه تا مقایسه‌ها بهتر انجام بگیره. با این شرح با روشی که در مطلب [Microbenchmark](#) آورده شده آزمایشات رو دوباره انجام دادم و برای تعداد تکرار 100 میلیون اختلاف تمام حلقه‌ها در حد چند میلی ثانیه بود که کاملاً قابل صرفنظره! نتایج برای حالات مختلف موجود تفاوت‌های زیادی داشت اما در نسخه ریلیز نهایتاً نتایج کلی این بود که حلقه for معمولی از همه سریعتر، سپس Array.ForEach و بعد متد ForEach در کلاس List و در نهایت از همه کندتر حلقه foreach بود. من آزمایشات روی یک سیستم با پردازنده 4 هسته ای با کلاک 3.4 گیگاهرتز (AMD Phenom II 965) با ویندوز 7 و 32 بیتی با رم 4 گیگ (3.25 گیگ قابل استفاده) انجام دادم. متأسفانه تعداد تکرار بیشتر خطای OutOfMemory میداد. **نکته:** اجرای تستهای این چینی برای آزمایش کارایی و سرعت به شدت تحت تاثیر عوامل جانبی هستند. مثل میزان منابع در دسترس سخت افزاری، نوع سیستم عامل، برنامه‌ها و سرویس‌های در حال اجرا، و مهمتر از همه نوع نسخه بیلد شده از برنامه تستر (دیبگ یا ریلیز) و محل اجرای تست (منظور اجرا در محیط دیباگ ویزوال استودیو یا اجرای مستقل برنامه) و ... (همونطور که آقای نصیری هم مطلبی مرتبط رو به اشتراک گذاشتند [^](#))

نویسنده: مسعود م. پاکدل
تاریخ: ۱۴۰۲/۰۳/۰۸

در ابتدا بهتر عنوان کنم که در کل 2 نوع برنامه نویسی وجود داره. برنامه نویسی که می‌خواد برنامه درست کار کنه و برنامه نویسی که می‌خواد برنامه درست نوشته بشه. در این جا هدف اصلی ما نوشتن برنامه به صورت درست هستش. دلیل اینکه foreach کندتر از Lambda ForEach عمل می‌کنه همان طور که جناب یوسف نژاد عنوان کردند به خاطر اجرای دستورات بیشتر در هر تکرار است. مثل کد زیر:

```
long Sum(List<int> intList)
{
    long result = 0;
    foreach (int i in intList)
        result += i;
    return result;
}
```

کامپایلر برای انجام کامپایل، کدهای بالا رو تبدیل به کدهای قابل فهم زیر می‌کنه:

```
long Sum(List<int> intList)
{
    long result = 0;
    List<T>.Enumerator enumerator = intList.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            int i = enumerator.Current;
            result += i;
        }
    }
}
```

```

}
finally
{
    enumerator.Dispose();
}
return result;
}

```

همانطور که می بینید از دو دستور enumerator.MoveNext و enumerator.Current در هر تکرار دایره استفاده می شه در حالی که List.ForEach فقط نیاز به یک فراخوانی در هر تکرار دارد.

در مورد Array.ForEach هم این نکته رو اضافه کنم که Array.ForEach فقط برای آرایه های یک بعدی استفاده میشه و کامپایلر هنگام کار با آرایه ها کد IEnumerator رو که در بالا توضیح دادم تولید نمی کنه در نتیجه در حلقه foreach برای آرایه ها هیچ فراخوانی متدی صورت نمی گیرد در حالی Array.ForEach نیاز به فراخوانی delegate تعریف شده در ForEach به ازای هر تکرار دارد.

آزمایشات بالا هم در یک سیستم DELL Inspiron 9400 با Core Duo T2400 و 2 GB RAM انجام شده است. این آزمایشات رو اگر در هر سیستم دیگر با هر Config اجرا کنید نتیجه کلی تغییر نخواهد کرد و فقط از نظر زمان اجرا تفاوت خواهیم داشت نه در نتیجه کلی.

نویسنده:

یوسف نژاد

تاریخ:

۱۳۹۲/۰۳/۱۲ ۱۲:۳۳

"این آزمایشات رو اگر در هر سیستم دیگر با هر Config اجرا کنید نتیجه کلی تغییر نخواهد کرد و فقط از نظر زمان اجرا تفاوت خواهیم داشت نه در نتیجه کلی."

این مطلب لزوما صحیح نیست. یک بنچمارک میتونه تو مجموعه سخت افزارهای مختلف، نتایج کاملا متفاوتی داشته باشه. مثلا سوالی در همین زمینه آقای [شهرز جعفری](#) تو [StackOverflow](#) پرسیدن که در جوابش دو نفر نتایج متفاوتی ارائه دادن. معمولا برای بیان نتایج تستهای بنچمارک ابتدا مشخصات سخت افزاری ارائه میشه مخصوصا وقتی که نتایج دقیق (و نه کلی) نشون داده میشه. مثل همین نتایج دقیق زمانهای اجرای حلقه ها.

نکته ای که من درکامنتم اشاره کردم صرفا درباره تست "سرعت اجرای" انواع حلقه ها بود که ممکنه با تست کارایی حلقه ها در اجرای یک کد خاص فرق داشته باشه.

نکته دیگه هم اینکه نمیدونم که آیا شما از همون متد Console.WriteLine در حلقه ها برای اجرای تستون استفاده کردین یا نه. فقط باید بگم که به خاطر مسائل و مشکلات مختلفی که استفاده از این متد به همراه دایره، به نظر من بکارگیری اون تو این جور تست ها اصلا مناسب نیست و باعث دور شدن زیاد نتایج از واقعیت میشه. مثلا من تست کردم و هر دفعه یه نتیجه ای می داد که نمیشه بر اساس اون نتیجه گیری کرد.

مورد دیگه ای هم که باید اضافه کنم اینه که بهتر بود شما کد کامل تست خودتون رو هم برای داندلود میذاشتین تا دیگران هم بتونن استفاده کنن. اینجوری خیلی بهتر میشه نتایج مختلف رو با هم مقایسه کرد. این مسئله برای تستای بنچمارک نسبتا رایج هست. مثل کد زیر که من آماده کردم:

```

static void Main(string[] args)
{
    //Action<int> func = Console.WriteLine;
    Action<int> func = number => number++;
    do
    {
        try
        {
            Console.Write("Iteration: ");
            var iterations = Convert.ToInt32(Console.ReadLine());
            Console.Write("Loop Type (for:0, foreach:1, List.ForEach:2, Array.ForEach:3): ");
            var loopType = Console.ReadLine();
            switch (loopType)
            {
                case "0":
                    Console.WriteLine("FOR loop test for {0} iterations", iterations.ToString("0,0"));

```

```

        TestFor(iterations, func);
        break;
    case "1":
        Console.WriteLine("FOREACH loop test for {0} iterations", iterations.ToString("0,0"));
        TestForEach(iterations, func);
        break;
    case "2":
        Console.WriteLine("LIST.FOREACH test for {0} iterations", iterations.ToString("0,0"));
        TestListForEach(iterations, func);
        break;
    case "3":
        Console.WriteLine("ARRAY.FOREACH test for {0} iterations", iterations.ToString("0,0"));
        TestArrayForEach(iterations, func);
        break;
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex);
}
Console.Write("Continue?(Y/N)");
Console.WriteLine("");
} while (Console.ReadKey(true).Key != ConsoleKey.N);

Console.WriteLine("Press any key to exit");
Console.ReadKey();
}

static void TestFor(int iterations, Action<int> func)
{
    StartupTest(func);

    var watch = Stopwatch.StartNew();
    for (int i = 0; i < iterations; i++)
    {
        func(i);
    }
    watch.Stop();
    ShowResults("for loop test: ", watch);
}

static void TestForEach(int iterations, Action<int> func)
{
    StartupTest(func);
    var list = Enumerable.Range(0, iterations);

    var watch = Stopwatch.StartNew();
    foreach (var item in list)
    {
        func(item);
    }
    watch.Stop();
    ShowResults("foreach loop test: ", watch);
}

static void TestListForEach(int iterations, Action<int> func)
{
    StartupTest(func);
    var list = Enumerable.Range(0, iterations).ToList();

    var watch = Stopwatch.StartNew();
    list.ForEach(func);
    watch.Stop();
    ShowResults("list.ForEach test: ", watch);
}

static void TestArrayForEach(int iterations, Action<int> func)
{
    StartupTest(func);
    var array = Enumerable.Range(0, iterations).ToArray();

    var watch = Stopwatch.StartNew();
    Array.ForEach(array, func);
    watch.Stop();
    ShowResults("Array.ForEach test: ", watch);
}

static void StartupTest(Action<int> func)
{
    // clean up

```

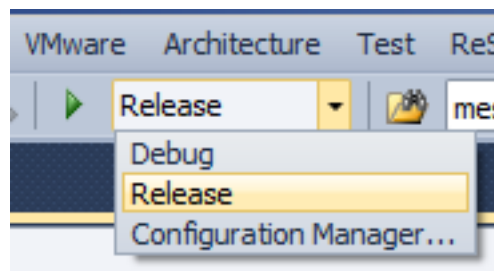
```

GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();

// warm up
func(0);
}
static void ShowResults(string description, Stopwatch watch)
{
    Console.WriteLine(description);
    Console.WriteLine(" Time Elapsed {0} ms", watch.ElapsedMilliseconds);
}

```

قبل از اجرای تست بهتره برنامه رو برای نسخه Release بیلد کنیم. ساده ترین روشش در تصویر زیر نشون داده شده:



پس از این تغییر و بیلد پروژه نتایج رو مقایسه میکنیم. نتایج اجرای این تست در همون سیستمی که قبلا تستای [StringBuilder](#) و [Microbenchmark](#) رو انجام دادم (یعنی لپ تاپ msi GE 620 با Core i7-2630QM) بصورت زیر:

```

C:\windows\system32\cmd.exe
Iteration: 100000000
Loop Type (for:0, foreach:1, List.ForEach:2, Array.ForEach:3): 0
FOR loop test for 100,000,000 iterations
for loop test: Time Elapsed 415 ms
Continue?(Y/N)
Iteration: 100000000
Loop Type (for:0, foreach:1, List.ForEach:2, Array.ForEach:3): 1
FOREACH loop test for 100,000,000 iterations
foreach loop test: Time Elapsed 1136 ms
Continue?(Y/N)
Iteration: 100000000
Loop Type (for:0, foreach:1, List.ForEach:2, Array.ForEach:3): 2
LIST.FOREACH test for 100,000,000 iterations
list.ForEach test: Time Elapsed 650 ms
Continue?(Y/N)
Iteration: 100000000
Loop Type (for:0, foreach:1, List.ForEach:2, Array.ForEach:3): 3
ARRAY.FOREACH test for 100,000,000 iterations
Array.ForEach test: Time Elapsed 460 ms

```

البته نتایج این تستها مطلق نیستن. نکاتی که در کامنت قبلی اشاره کردم از عوامل تاثیرگذار هستن. موفق باشین.

تاریخ: ۱۳:۳۸ ۱۳۹۲/۰۳/۱۲

شما هم در کل به این نتیجه رسیدید که list.ForEach از foreach loop سریعتر است. حلقه for معمولی نیز از تمام اینها سریعتر. بنابراین کار شما ناقض مطلب آقای پاکدل «نتیجه کلی تغییر نخواهد کرد و فقط از نظر زمان اجرا تفاوت خواهیم داشت نه در نتیجه کلی» نیست و مطلب ایشان برقرار است.

نویسنده: یوسف نژاد
تاریخ: ۱۳:۴۷ ۱۳۹۲/۰۳/۱۲

من نمیخواستم مطلبی رو نقض کنم فقط میخواستم بگم بهتره برای مقایسه نتایج اینجوری عمل بشه. درضمن نتایج بدست اومده من برای متد Array.ForEach با نتایج آقای پاکدل فرق میکنه. اما بحثی که اشاره کردم درست است و "یکسان بودن نتایج کلی با تغییر سخت افزار" همیشه برقرار نیست و برخی مواقع میتونه تفاوتهایی هم وجود داشته باشه. اما شاید تو این مثال کوچیک بهش برنخوریم اما در کل اینطوریست.

نویسنده: وحید نصیری
تاریخ: ۱۴:۰۰ ۱۳۹۲/۰۳/۱۲

در مورد تفاوت نتایج حاصل از بررسی کارآیی Array.ForEach، مطالبی در اینجا هست که علت رو بیشتر باز کرده (و دقیقاً در مثالهای جاری صادق هست؛ یکی با lambda است و دیگری بدون lambda):

[تفاوت کارآیی در حین استفاده از Lambdas و Method groups](#)