عنوان: **مدیریت حافظه در JavaScript**
نویسنده: م منفرد
تاریخ: ۱۳۹۲/۰۵/۰۸ ۲۲:۰
آدرس: www.dotnettips.info
برچسب‌ها: JavaScript, Memory Manegement, GC

مدیریت حافظه در JavaScript همانند مدل مدیریت حافظه در .NET می‌باشد. حافظه وقتی مورد نیاز است تخصیص پیدا می‌کند و وقتی دیگر مورد نیاز نیست آزاد می‌شود. این پروسه در CLR به نام جمع آوری زباله یا Garbage Collector یا GC مشهور است. تفاوت عمده فی مابین مدیریت حافظه در .NET با مدیریت حافظه در JavaScript این است که مدیریت حافظه در .NET توسط CLR واحد انجام می‌شود. یعنی پیاده سازی واحدی از GC وجود دارد و شما می‌توانید از نوع فعالیت آن اطمینان حاصل نمایید ولی در JavaScript با توجه به اینکه موتورهای اجرایی مختلفی برای اجرای آن وجود دارد، در سرورها و مرورگرهای مختلف پیاده سازی‌های متفاوتی برای آن وجود دارد. اطلاع از نحوه کار GC می‌تواند به درک ما از JavaScript کمک کرده تا بتوانیم کدهای بهتری در این زبان تولید کنیم.

در این مقاله به بررسی دو الگوریتم عمده GC در JavaScript می‌پردازیم.

### 1. مدل Reference Counting Garbage Collector

در این مدل از جمع آوری زباله، به ازای ایجاد هر آبجکت در حافظه و یا هر تخصیصی در حافظه، شمارشگری با عنوان reference counter در نظر گرفته می‌شود. هر زمان که به این آبجکت یا حافظه تخصیصی دسترسی ایجاد شود و یا reference داده شود، یک واحد به شمارشگر آن اضافه و هر وقت که رفرنس به حافظه یا آبجکت دیگر مورد استفاده نداشت یا از دسترس خارج شد، یک واحد از شمارشگر آن کاسته می‌شود. این مدل که سریعترین، ساده‌ترین و کم سربارترین مدل GC می‌باشد، وقتی شمارشگر رفرنس حافظه به صفر رسید، حافظه ومنابع سیستم تخصیصی به آن آبجکت آزاد شده و آماده استفاده مجدد می‌شودبه عنوان نمونه به کد زیر دقت کنید:

```
var object1='GC test object 1';
function Test1(){
   var object2='GC test object 2';
   alert (object1+'-' + object2);
}
alert (object1);
```

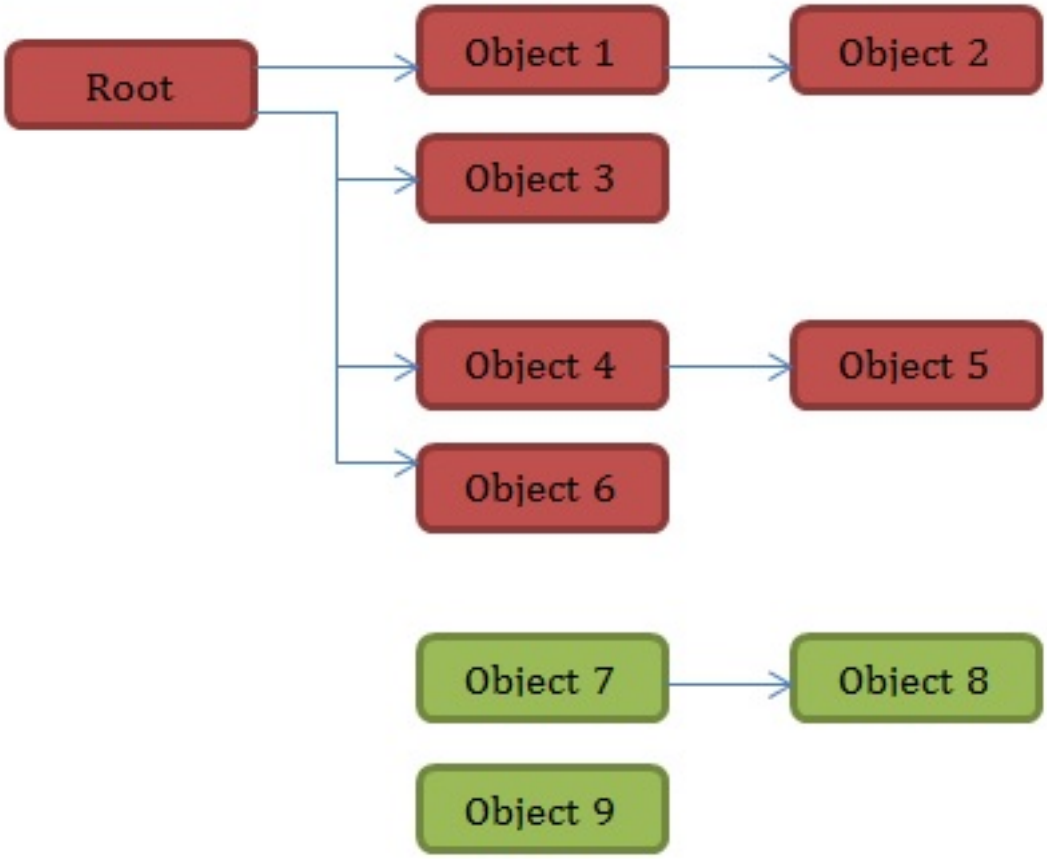
پس از اجرای این کد، جدولی مانند زیر در GC ایجاد می‌شود که به صورت زیر مقدار طی اجرای برنامه مقدار دهی می‌شود:

| Reference Counter End Program | Reference Counter Line 6 | Reference Counter Line 5 | Reference Counter Line 3 | Reference Counter Line 1 | Object |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 1 | object1 |
| 0 | 0 | 0 | 1 | - | object2 |

همانطور که در جدول فوق مشخص است، وقتی از متقیر استفاده می‌شود، reference count آن زیاد و وقتی دیگر مورد استفاده ندارد یکی کم می‌شود. وقتی مقدار reference count به صفر برسد، متقیر از حافظه حذف شده و منابع سیستمی آزاد می‌شود. این مدل که در مرورگرهای قدیمی مورد استفاده قرار گرفته است، در صورتی که دو آبجکت به یکدیگر ارجاع داشته باشند، reference counter آن صفر نشده، حافظه و منابع تخصیصی آنها آزاد نمی‌شود و احتمال ایجاد نشت حافظه زیاد می‌شود.

### 2. مدل Mark-and-Sweep

در این مدل از مدیریت حافظه، برای آبجکت‌های ایجادی در حافظه،GC درخت ارجاعات ایجاد کرده و دقیقا مشخص می‌کند زمانی که یک آبجکت در دسترس نباشد و یا دیگر نیازی به آن نباشد، آن را از حافظه حذف می‌کند. مانند شکل زیر:




در این صورت هنگامی که آبجکت دیگر واقعا مورد نیاز نباشد از حافظه حذف می‌شود. یعنی اگر دو آبجکت به یکدیگر نیز ارجاع داشته باشند هنگامی که دیگر مورد استفاده قرار نگیرند حذف شده و امکان ایجاد نشت حافظه به حداقل می‌رسد.
تفاوت عمده بین GC در Javascript و GC در CLR این است که در زبان‌های مبتنی بر .NET شما می‌توانید به صورت مستقیم GC را صدا زده تا عمل جمع آوری زباله انجام پذیرد ولی در JavaScript هر زمان که نیاز به حافظه بیشتر باشد (و یا در یک زمانبندی مشخص) عمل جمع آوری زباله انجام شده و از طریق کد قابل فراخوانی نمی‌باشد.

عنوان: **نگاهی به درون سیستم Binding در WPF و یافتن مواردی که هنوز در حافظه‌اند**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۵/۲۰ ۱۴:۵
آدرس: www.dotnettips.info
گروه‌ها: WPF, Performance, Memory Manegement, GC

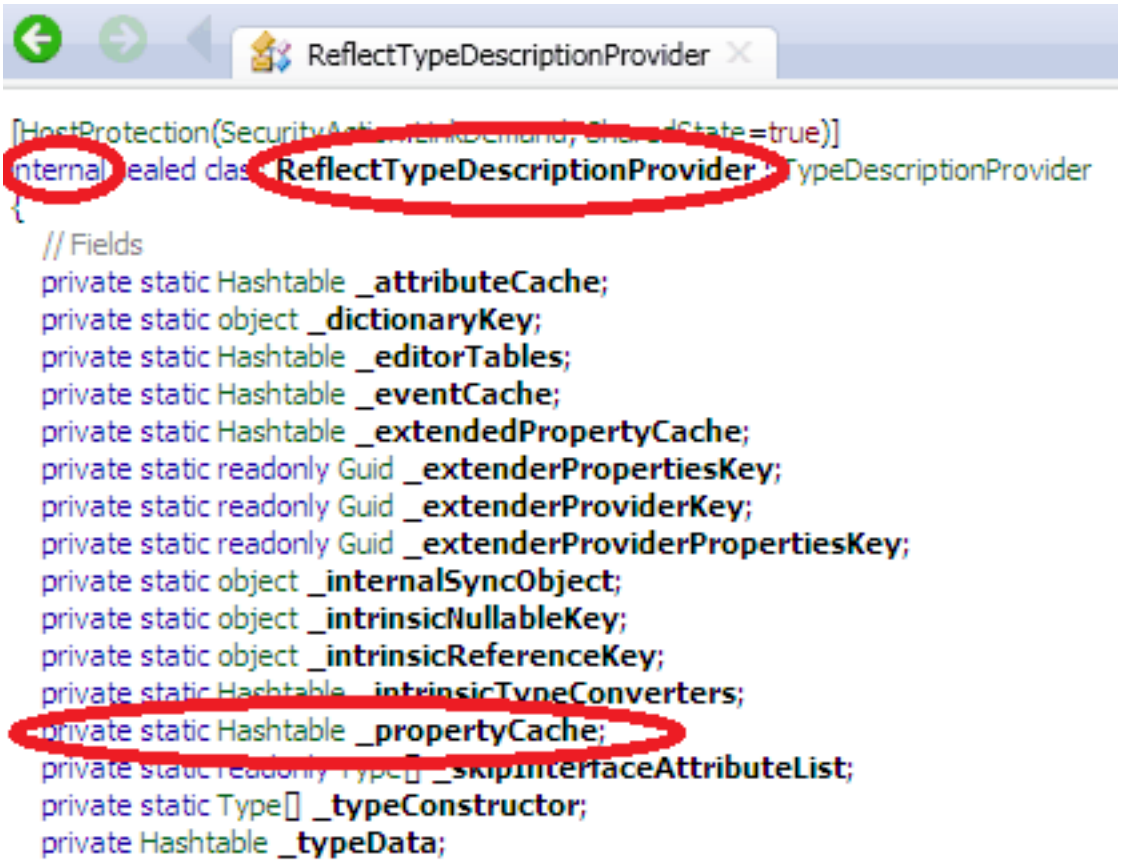
در WPF، زیر ساخت‌های ComponentModel توسط کلاسی به نام PropertyDescriptor ، منابع Binding موجود در قسمت‌های مختلف برنامه را در جدولی عمومی ذخیره و نگهداری می‌کند. هدف از آن، مطلع بودن از مواردی است که نیاز دارند توسط مکانیزم‌هایی مانند INotifyPropertyChanged و DependencyProperty ها، اطلاعات اشیاء متصل را به روز کنند.
در این سیستم، کلیه اتصالاتی که Mode آن‌ها به OneTime تنظیم نشده است، به صورت اجباری دارای یک valueChangedHandlers متصل توسط سیستم PropertyDescriptor خواهند بود و در حافظه زنده نگه داشته می‌شوند؛ تا بتوان در صورت نیاز، توسط سیستم binding اطلاعات آن‌ها را به روز کرد.
همین مساله سبب می‌شود تا اگر قرار نیست خاصیتی برای نمونه توسط مکانیزم INotifyPropertyChanged اطلاعات UI را به روز کند (یک خاصیت معمولی دات نتی است) و همچنین حالت اتصال آن به OneTime نیز تنظیم نشده، سبب مصرف حافظه بیش از حد برنامه شود.
اطلاعات بیشتر
A memory leak may occur when you use data binding in Windows Presentation Foundation

راه حل آن هم ساده است. برای اینکه valueChangedHandler ایی به خاصیت ساده‌ای که قرار نیست بعدها UI را به روز کند، متصل نشود، حالت اتصال آن‌را باید به OneTime تنظیم کرد.

**سؤال: در یک برنامه بزرگ که هم اکنون مشغول به کار است، چطور می‌توان این مسایل را ردیابی کرد؟**

برای دستیابی به اطلاعات کش Binding در WPF، باید به Reflection متوسل شد. به این ترتیب در برنامه جاری، در کلاس PropertyDescriptor به دنبال یک کلاس خصوصی تو در توی دیگری به نام ReflectTypeDescriptionProvider خواهیم گشت (این اطلاعات از طریق مراجعه به سورس دات نت و یا حتی برنامه‌های ILSpy و Reflector قابل استخراج است) و سپس در این کلاس خصوصی داخلی، فیلد خصوصی propertyCache آن‌را که از نوع HashTable است استخراج می‌کنیم:

```
var reflectTypeDescriptionProvider =
typeof(PropertyDescriptor).Module.GetType("System.ComponentModel.ReflectTypeDescriptionProvider");
var propertyCacheField = reflectTypeDescriptionProvider.GetField("_propertyCache",
BindingFlags.Static | BindingFlags.NonPublic);
```




اکنون به لیست داخلی Binding نگهداری شونده توسط WPF دسترسی پیدا کرده‌ایم. در این لیست به دنبال مواردی خواهیم گشت که فیلد valueChangedHandlers به آن‌ها متصل شده است و در حال گوش فرا دادن به سیستم binding هستند (سورس کامل و طولانی این مبحث را در پروژه پیوست شده می‌توانید ملاحظه کنید).

**یک مثال: تعریف یک کلاس ساده، اتصال آن و سپس بررسی اطلاعات درونی سیستم Binding**

فرض کنید یک کلاس مدل ساده به نحو ذیل تعریف شده است:

```
namespace WpfOneTime.Models
{
    public class User
    {
        public string Name { set; get; }
    }
}
```

سپس این کلاس به صورت یک List، توسط ViewModel برنامه در اختیار View متناظر با آن قرار می‌گیرد:

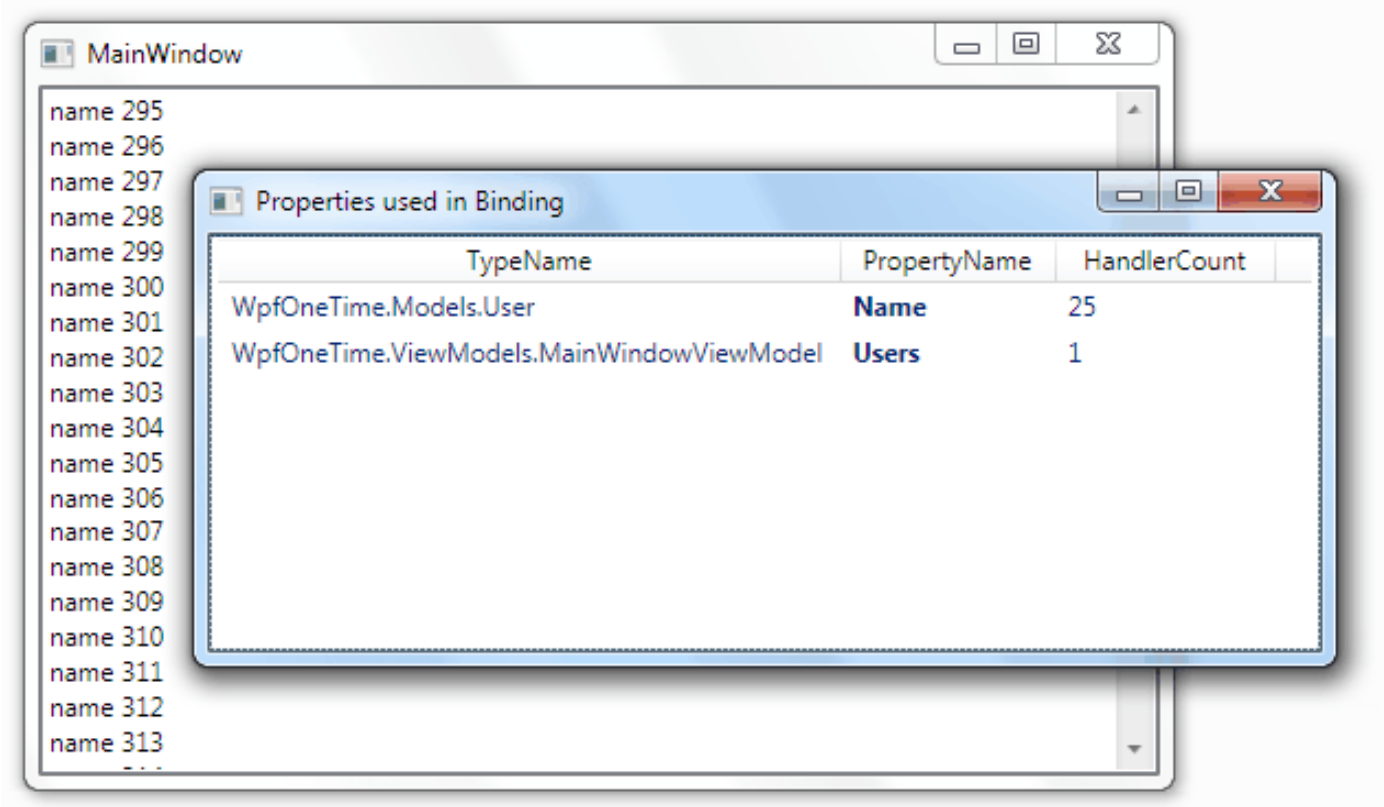
```
using WpfOneTime.Models;
using System.Collections.Generic;

namespace WpfOneTime.ViewModels
{
    public class MainWindowViewModel
    {
        public IList<User> Users { set; get; }

        public MainWindowViewModel()
        {
            Users = new List<User>();
```

```
            for (int i = 0; i < 1000; i++)
            {
                Users.Add(new User { Name = "name " + i });
            }
        }
    }
}
```

تعاریف View برنامه نیز به نحو زیر است:

```
<Window x:Class="WpfOneTime.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:ViewModels="clr-namespace:WpfOneTime.ViewModels"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <ViewModels:MainWindowViewModel x:Key="vmMainWindowViewModel" />
    </Window.Resources>
    <Grid DataContext="{Binding Source={StaticResource vmMainWindowViewModel}}">
        <ListBox ItemsSource="{Binding Users}">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding Name}" />
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>
</Window>
```

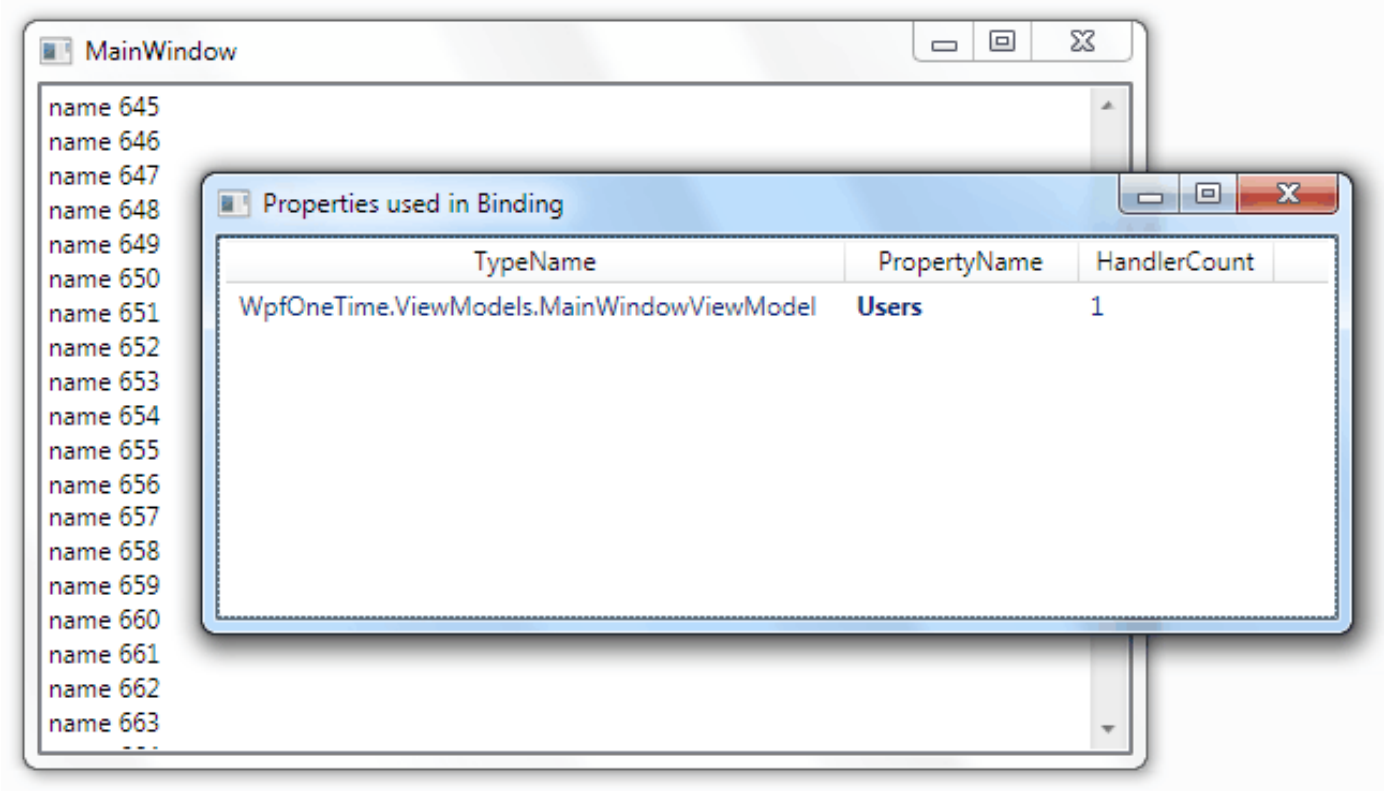
همه چیز در آن معمولی به نظر می‌رسد. ابتدا به ViewModel برنامه دسترسی یافته و DataContext را با آن مقدار دهی می‌کنیم. سپس اطلاعات این لیست را توسط یک ListBox نمایش خواهیم داد.
خوب؛ اکنون اگر اطلاعات HashTable داخلی سیستم Binding را در مورد View فوق بررسی کنیم به شکل زیر خواهیم رسید:



بله. تعداد زیادی خاصیت Name زنده و موجود در حافظه باقی هستند که تحت ردیابی سیستم Binding می‌باشند.
در ادامه، نکته‌ی ابتدای بحث را جهت تعیین حالت Binding به OneTime ، به View فوق اعمال می‌کنیم (یک سطر ذیل باید تغییر کند):

```
<TextBlock Text="{Binding Name, Mode=OneTime}" />
```

در این حالت اگر نگاهی به سیستم ردیابی WPF داشته باشیم، دیگر خبری از اشیاء زنده دارای خاصیت Name در حال ردیابی نیست:




به این ترتیب می‌توان در لیست‌های طولانی، به مصرف حافظه کمتری در برنامه WPF خود رسید.
بدیهی است این نکته را تنها در مواردی می‌توان اعمال کرد که نیاز به به‌روز رسانی‌های ثانویه اطلاعات UI در کدهای برنامه وجود ندارند.

### چطور از این نکته برای پروفایل یک برنامه موجود استفاده کنیم؟

کدهای برنامه را از انتهای بحث دریافت کنید. سپس دو فایل ReflectPropertyDescriptorWindow.xaml و ReflectPropertyDescriptorWindow.xaml.cs آن‌را به پروژه خود اضافه نمائید و در سازنده پنجره اصلی برنامه، کد ذیل را فراخوانی نمائید:

```
new ReflectPropertyDescriptorWindow().Show();
```

کمی با برنامه کار کرده و منتظر شوید تا لیست نهایی اطلاعات داخلی Binding ظاهر شود. سپس مواردی را که دارای HandlerCount بالا هستند، مدنظر قرار داده و بررسی نمائید که آیا واقعا این اشیاء نیاز به valueChangedHandler متصل دارند یا خیر؟ آیا قرار است بعدها UI را از طریق تغییر مقدار خاصیت آن‌ها به روز نمائیم یا خیر. اگر خیر، تنها کافی است نکته Mode=OneTime را به این Bindingها اعمال نمائیم.

**دریافت کدهای کامل پروژه این مطلب**

WpfOneTime.zip

## نظرات خوانندگان

نویسنده: سیما
تاریخ: ۱۳۹۳/۰۳/۲۳ ۱۹:۲

سلام،
می‌خواستم بدونم به چه شکل میتوانم متوجه شوم کدام قسمت از برنامه من موجب افزایش مصرف رم شده است؟ برای مثال برنامه من بعد گذشت 1 دقیقه از اجرای آن مصرف رم معادل 5MB دارم ولی پس از گذشت 10 دقیقه به 1GB میرسد.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۳/۲۳ ۱۹:۱۷

از برنامه‌های Profiler باید استفاده کنید؛ مانند:

- ابزارهای توکار VS.NET
- New Memory Usage Tool for WPF and Win32 Applications
- Windows Performance Toolkit
- dotMemory
- ANTS Memory Profiler