عنوان: **مدیریت حافظه در JavaScript**
نویسنده: م عسکری
تاریخ: ۱۳۹۲/۰۵/۰۸ ۲۲:۰
آدرس: www.dotnettips.info
گروه‌ها: 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);
```

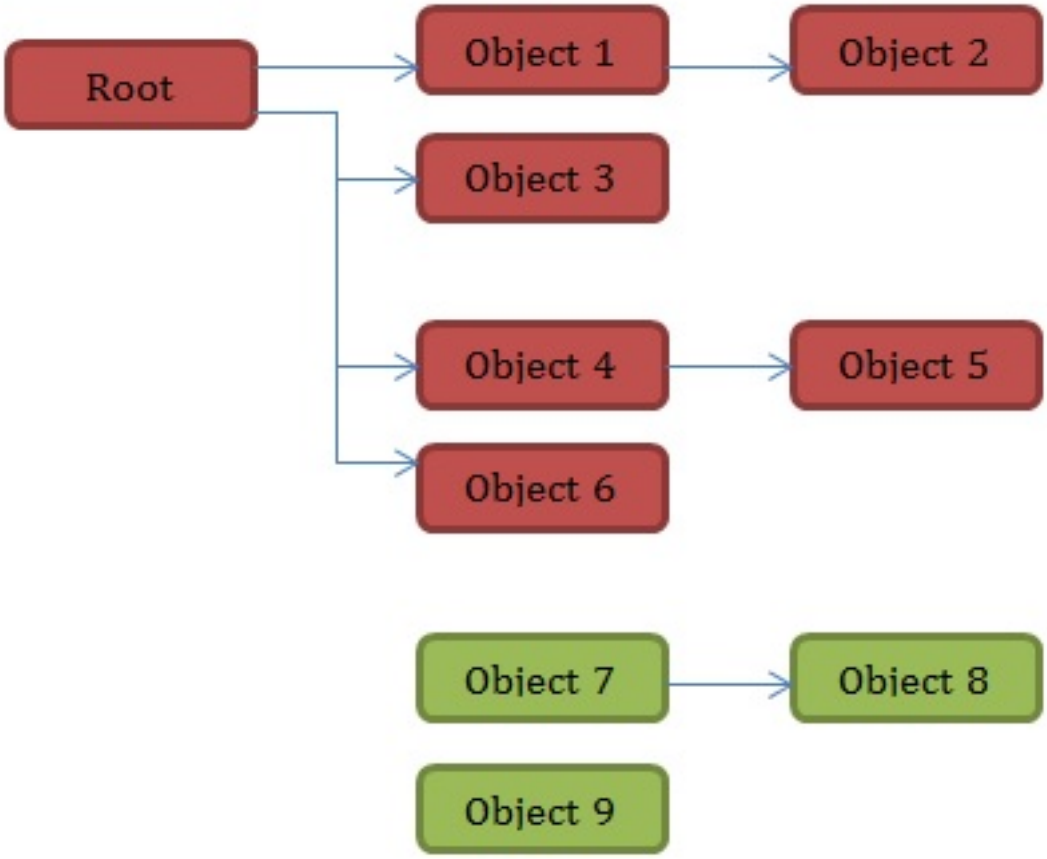
پس از اجرای این کد، جدولی مانند زیر در GC ایجاد می‌شود که به صورت زیر مقدار طی اجرای برنامه مقدار دهی می‌شود:

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.

نظرات خوانندگان

نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۵/۰۲ ۰:۵۵

مورد اول رو زیاد متوجه نشدم آیا هر شی جدا کانتر میخوره؟اگر آره چرا در خط سوم شده ۲ و اگه کلا فقط شماره میندازه آیا به هر شی اشاره داره.کلا نتونستم هیچ ارتباطی بین کد و جدول پیدا کنم

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

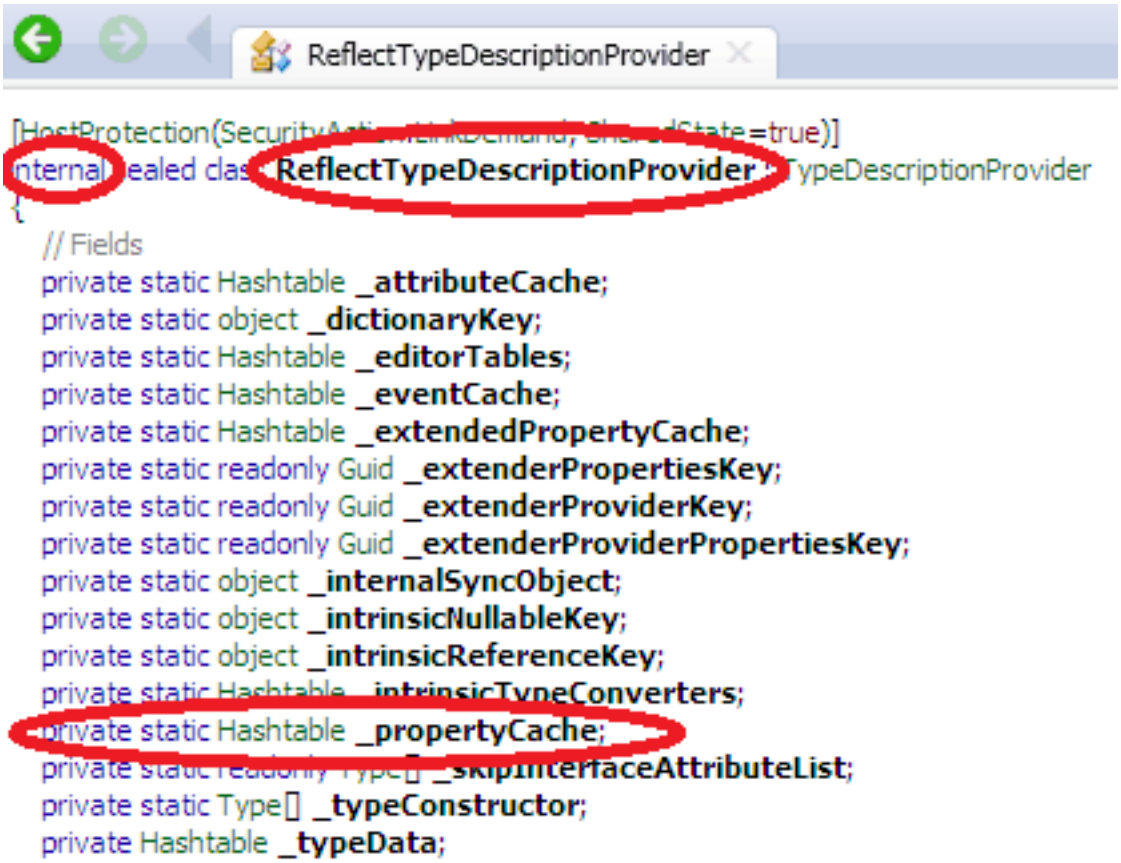
در WPF، زیر ساخت‌های ComponentModel توسط کلاسی به نام PropertyDescriptor ، منابع Binding موجود در قسمت‌های مختلف برنامه را در جدولی عمومی ذخیره و نگهداری می‌کند. هدف از آن، مطلع بودن از مواردی است که نیاز دارند توسط مکانیزم‌هایی مانند INotifyPropertyChanged و DependencyProperty ها، اطلاعات اشیاء متصل 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:

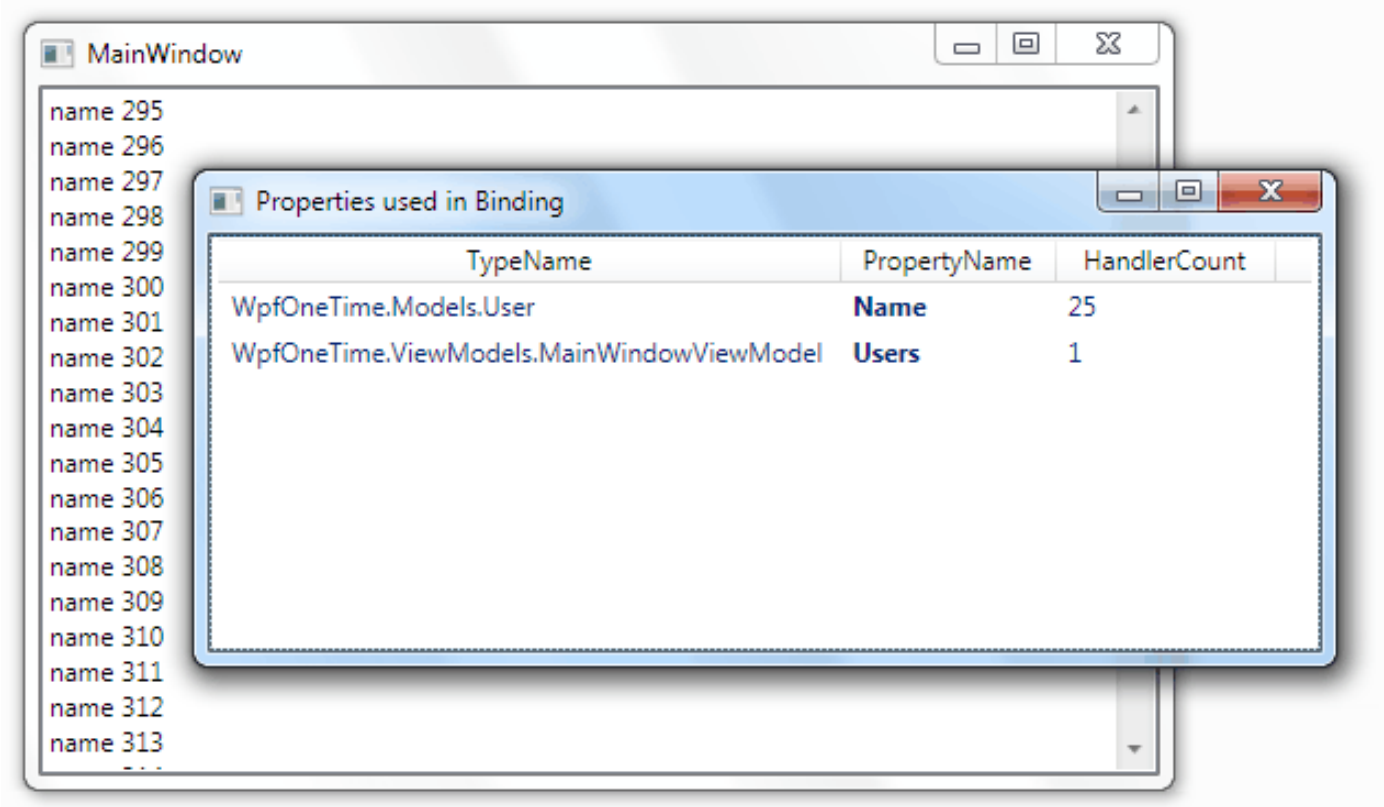
```
using WpfOneTime.Models;
using System.Collections.Generic;

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>
```

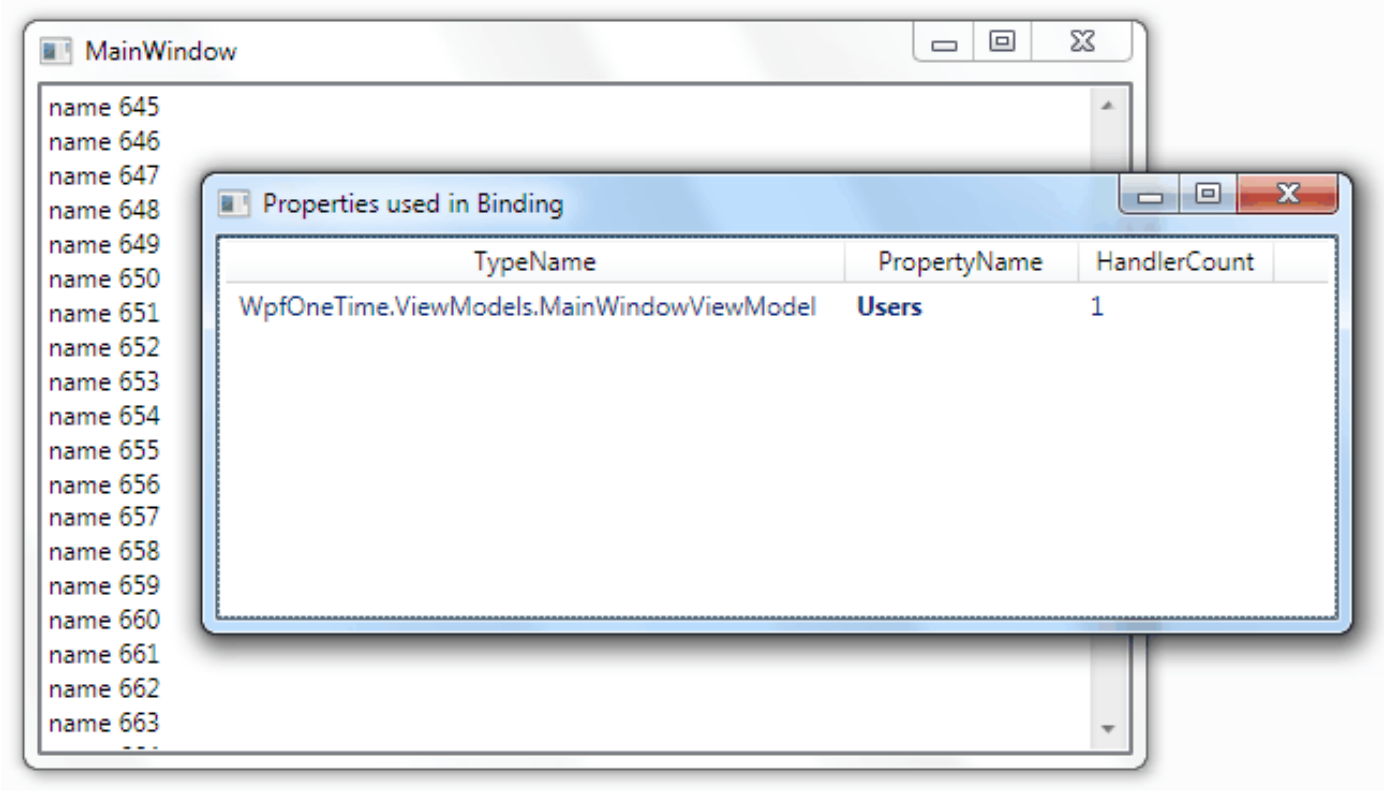
همه چیز در آن معمولی به نظر می‌رسد. ابتدا به ViewModel برنامه دسترسی یافته و DataContext را با آن مقدار دهی می‌کنیم. سپس اطلاعات این لیست را توسط یک ListBox نمایش خواهیم داد.
خوب؛ اکنون اگر اطلاعات HashTable داخلی سیستم Binding را در مورد View فوق بررسی کنیم به شکل زیر خواهیم رسید:



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

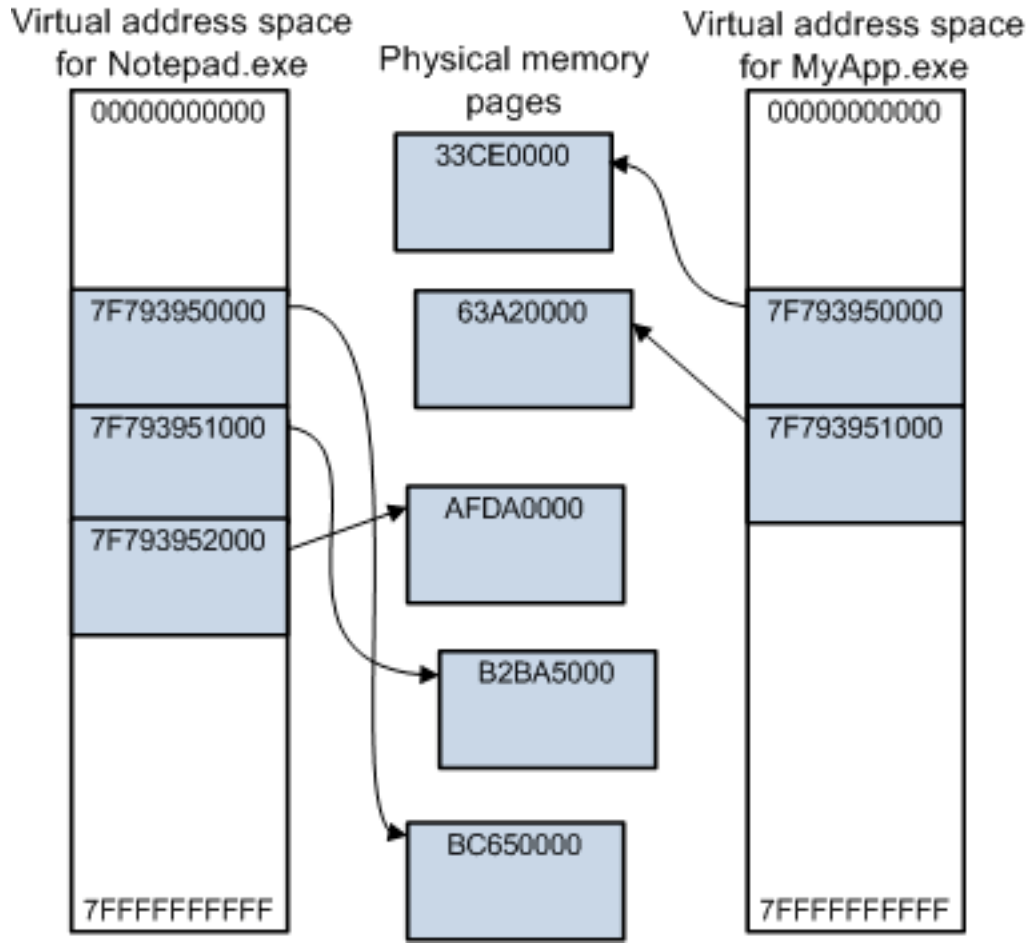
عنوان: **آشنایی با ذخیره سازی در حافظه**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۳/۰۹/۳۰ ۱۰:۲۵
آدرس: www.dotnettips.info
گروه‌ها: IIS, Memory Manegement, Operating System

آشنایی با Virtual Address spaces

فضای آدرس‌دهی مجازی: موقعی که یک پردازشگر در مکانی از حافظه عمل خواندن و نوشتن را آغاز می‌کند، از آدرس‌های مجازی بهره می‌برد. بخشی از عملیات خواندن و نوشتن، تبدیل آدرس‌های مجازی به آدرس‌های فیزیکی در حافظه است. این عمل سه مزیت دارد:

آدرس‌های مجازی به صورت پیوسته و پشت سر هم هستند و آدرس دهی بسیار راحت است ولی داده‌ها بر روی یک حافظه به صورت متصل به هم یا پیوسته ذخیره یا خوانده نمی‌شوند و کار آدرس دهی مشکل است. پس یکی از مزایای داشتن آدرس دهی مجازی پشت سر هم قرار گرفتن آدرس هاست.
برنامه از آدرس‌های مجازی برای دسترسی به بافر حافظه استفاده می‌کند که بزرگتر از حافظه فیزیکی موجود هست. موقعی که نیاز به حافظه بیشتر باشد و حافظه سیستم کوچکتر یا کمتر از تقاضا باشد، مدیر حافظه، صفحات حافظه فیزیکی را به صورت یک فایل (عموما 4 کیلوبیی) بر روی دیسک سخت ذخیره می‌کند و صفحات داده‌ها در موقع نیاز بین حافظه فیزیکی و دیسک سخت جابجا می‌شود.
هر پردازشی که بر روی آدرس‌های مجازی کار می‌کند ایزوله شده است. یعنی یک پروسه هیچ گاه نمیتواند به آدرس‌های یک پروسه دیگر دسترسی داشته باشد و باعث تخریب داده‌های آن شود.

به محدوده شروع آدرس‌های مجازی تا پایان آن محدوده، فضای آدرس‌دهی مجازی گویند. هر پروسه ای که در مد کاربر آغاز میشود از یک فضای آدرس خصوصی یا مختص به خود استفاده می‌کند. برای سیستم‌های 32 بیتی این فضا میتواند دو گیگ باشد که از آدرس 0x00000000 شروع می‌شود و تا 0x7FFFFFFF ادامه پیدا می‌کند و برای یک سیستم 64 بیتی تا 8 ترابایت می‌باشد که از آدرس 0x000'00000000 تا آدرس 0x7FF'FFFFFFFF ادامه می‌یابد. گاهی اوقات به محدوده آدرس‌های مجازی، حافظه مجازی می‌گویند. شکل زیر اصلی‌ترین خصوصیات فضای آدرس‌های مجازی را نشان می‌دهد:



در شکل بالا دو پروسه 64 بیتی به نام‌های *notepad.exe* و *myapp.exe* قرار دارند که هر کدام فضای آدرس‌های مجازی خودشان را دارند و از آدرس 0x000'0000000 شروع و تا آدرس 0x7FF'FFFFFFFF ادامه میابند. هر قسمت شامل یک صفحه 4 کیلویی از حافظه مجازی یا فیزیکی است. به برنامه نوت‌پد دقت کنید که از سه صفحه پشت سر هم یا پیوسته تشکیل شده که آدرس شروع آن 0x7F7'93950000 می باشد ولی در حافظه فیزیکی خبری از پیوسته بودن دیده نمی‌شود و حتما این نکته را متوجه شدید که هر دو پروسه از یک آدرس شروع استفاده کرده‌اند، ولی به آدرسی متفاوت از حافظه فیزیکی نگاشت شده اند.

تفاوت user mode و kernel mode

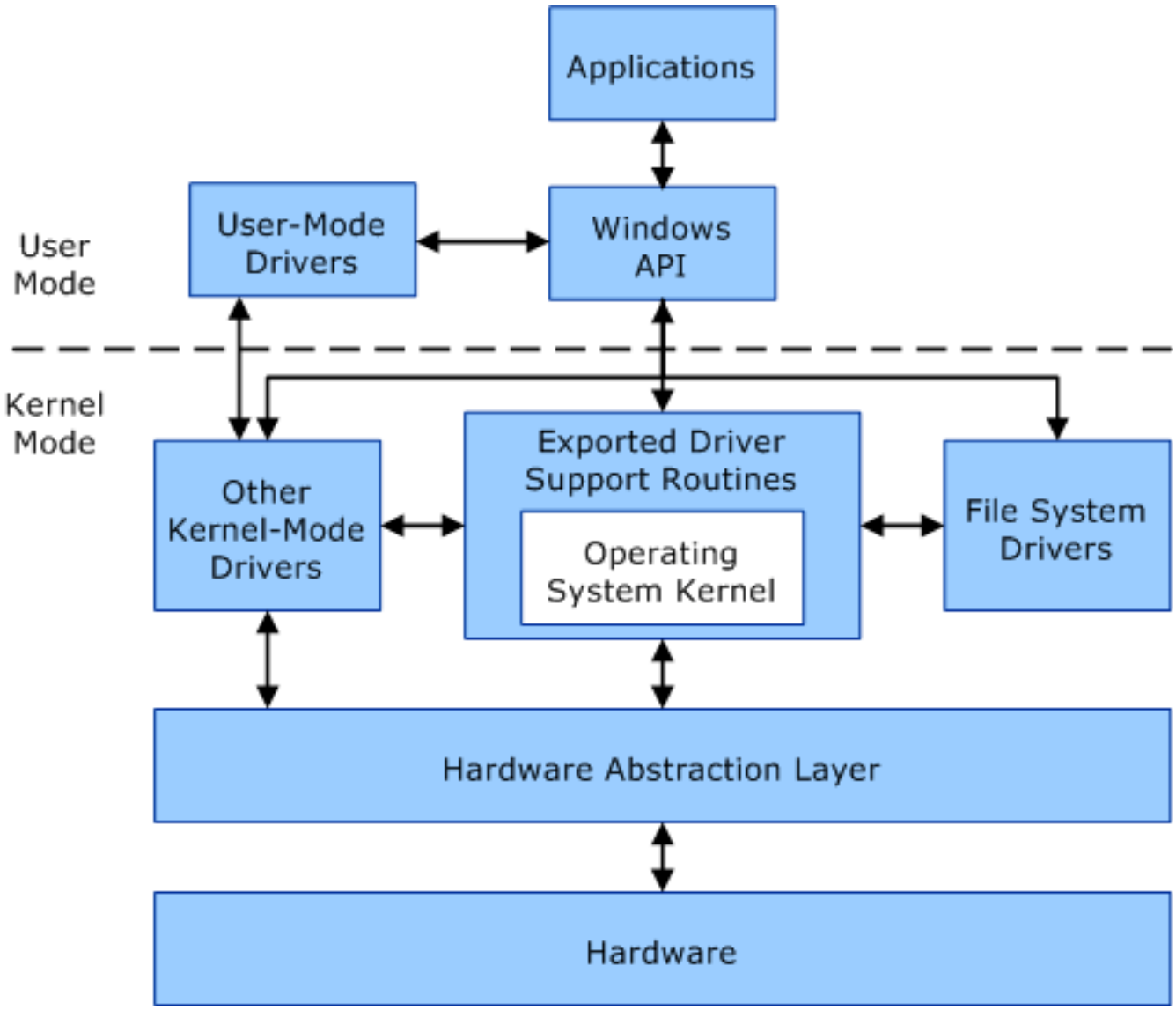
هر پردازش در سیستم بر اساس user mode مد کاربر یا kernel mode مد کرنل اجرا میشود. پردازش‌ها بر اساس هر نوع کد بین این دو بخش سوییچ می‌کنند. اپلیکیشن‌ها بر اساس مد کاربر و هسته سیستم عامل و اکثر درایورها بر اساس مد کرنل کار می‌کنند؛ ولی تعدادی از آن‌ها هم در مد کاربر.

هر برنامه یا اپلیکیشنی که اجرا می‌شود، در یک مد کاربری قرار می‌گیرد. ویندوز هم برای هر برنامه یک پروسه یا فرآیندی را ایجاد می‌کند. پروسه برای برنامه یک فضای آدرس‌دهی مجازی و یک جدول مدیریت به صورت خصوصی یا مختص همین برنامه تشکیل می‌دهد. به این ترتیب هیچ برنامه دیگری نمی‌تواند به داده‌های برنامه دیگر دسترسی داشته باشد و هر برنامه در یک محیط ایزوله شده برای خودش قرار میگیرد و این برنامه اگر به هر ترتیبی کرش کند، برنامه‌های دیگر به کار خود ادامه می‌دهند و هیچ تاثیری بر برنامه‌های دیگر نمی‌گذارند.

البته استفاده از این آدرس‌های مجازی محدودیت هایی هم دارد، چرا که بعضی از آن‌ها توسط سیستم عامل رزرو شده اند و برنامه نمی‌تواند به آن قسمت‌ها دسترسی داشته باشد و این باعث می‌شود که داده‌های برنامه از خسارت و آسیب دیدن حفظ شوند.

تمام برنامه هایی در حالت کرنل ایجاد می‌شوند، از یک فضای آدرس مجازی استفاده می‌کنند. به این معنی که یک درایور مد کرنل نسبت به دیگر درایورها و خود سیستم عامل به هیچ عنوان در یک محیط ایزوله قرار ندارد. بنابراین ممکن است یک کرنل درایور تصادفا در یک آدرس مجازی اشتباه که میتواند متعلق به سیستم عامل یا یک درایور دیگر باشد بنویسد. یعنی اگر یک درایور کرنل کراش کند کل سیستم عامل کرش میکند.

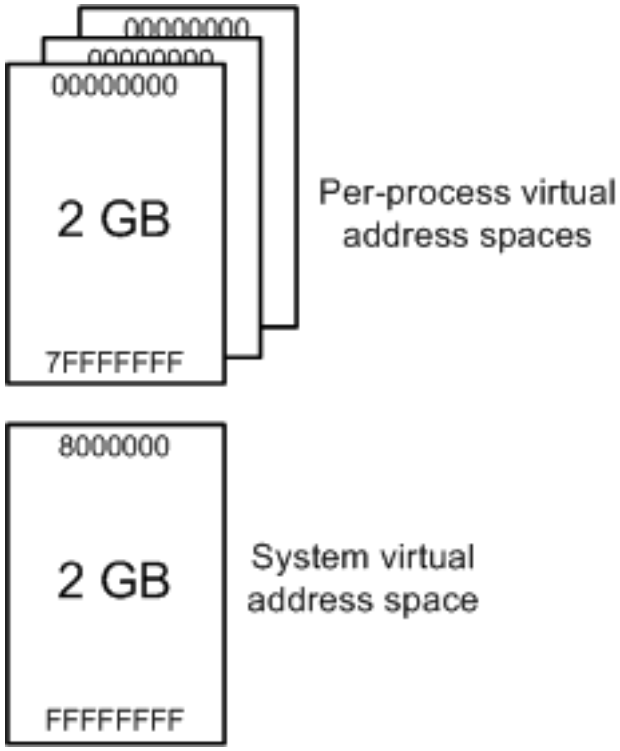
تصویر زیر به خوبی ارتباط بین مد کاربری و مد کرنل را نشان می‌دهد:



فضای کاربری و فضای سیستمی User space and system space

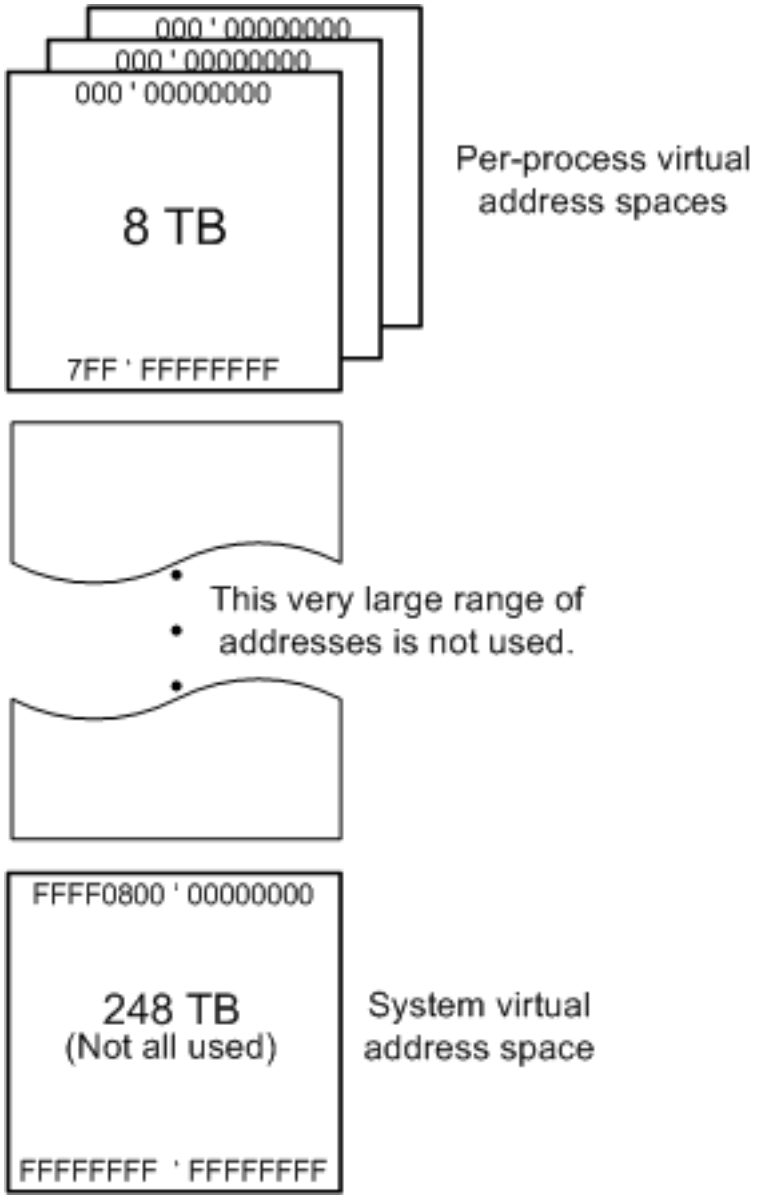
گفتیم بسیاری از پروسه‌ها در حالت user mode و پروسه‌های هسته سیستم عامل و درایورها در حالت kernel mode اجرا می‌شوند. هر پروسه مد کاربر از فضای آدرس دهی مجازی خودش استفاده می‌کند ولی در حالت کرنل همه از یک فضای آدرس دهی استفاده می‌کنند که به آن فضای سیستمی می‌گویند و برای مد کاربری می‌گویند فضای کاربری.

در سیستم‌های 32 بیتی نهایتا تا 4 گیگ حافظه می‌توان به این‌ها تخصیص داد؛ 2 گیگ ابتدایی به user space و دو گیگ بعدی به system space :



در ویندوزهای 32 بیتی شما امکان تغییر این مقدار حافظه را در میان بوت دارید و می‌توانید حافظه کاربری را تا 3 گیگ مشخص کنید و یک گیگ را برای فضای سیستمی. برای اینکار می‌توانید از برنامه `bcedit` استفاده کنید.

در سیستم‌های 64 بیتی میزان حافظه‌های مجازی به صورت تئوری تا 16 اگزابایت مشخص شده است؛ ولی در عمل تنها بخش کوچکی از آن یعنی 8 ترابایت استفاده می‌شود.



کدهایی که در user mode اجرا می‌شوند فقط به فضای کاربری دسترسی دارند و دسترسی آن‌ها به فضای سیستمی به منظور جلوگیری از تخریب داده ممکن نیست. ولی در حالت کرنل می‌توان به دو فضای سیستمی و کاربری دسترسی داشت. درایورهایی که در مدکرنل نوشته شده اند باید تمام دقت خود را در زمینه نوشتن و خواندن از فضای سیستمی در حافظه به کار گیرند.
سناریوی زیر به شما نشان می‌دهد که چرا باید مراقب بود:
برنامه جهت اجرا در مد کاربر یک درخواست را برای خواندن داده‌های یک device را آماده می‌کند. سپس برنامه آدرس شروع یک بافر را برای دریافت داده، مشخص می‌کند.
وظیفه این درایور یک قطعه در مد کرنل این است که عملیات خواندن را شروع کرده و کنترل را به درخواست کننده ارسال می‌کند.
بعد device یک وقفه را به هر تردی thread که در حال اجراست ارسال می‌کند تا بگوید، عملیات خواندن پایان یافته است. این وقفه توسط ترد درایور مربوطه دریافت می‌شود.
حالا دیگر درایور نباید داده‌ها را در همان جایی که گام اول برنامه مشخص کرده است ذخیره کند. چون این آدرس که برنامه در مد کاربری مشخص کرده است، با نمونه‌ای که این فرآیند محاسبه می‌کند متفاوت است.

Paged Pool and NonPaged Pool

در فضای کاربری تمام صفحات در صورت نیاز توانایی انتقال به دیسک سخت را دارند ولی در فضای سیستمی همه بدین صورت نیستند. فضای سیستمی دو ناحیه حافظه تخصیصی پویا دارد که به نام‌های *paged pool* و *nonpaged pool* شناخته می‌شوند.
در سیستم‌های 32 بیتی *Pagedpool* توانایی 128 گیگ فضای آدرس دهی مجازی را از آدرس 0xFFFFAC00'00000000 تا آدرس

0xFFFFAC1F'FFFFFFFF و در سیستم‌های 64 بیتی توانایی 128 گیگ فضای آدرس دهی مجازی را از 0xFFFFA800'00000000 تا 0xFFFFA81F'FFFFFFFF دارد. حافظه ای که به صورت *paged pool* تخصیص شده باشد می‌تواند صفحات حافظه را بر روی دیسک سخت ذخیره کند؛ ولی حافظه ای که به صورت *nonpaged* تخصیص یافته باشد، هرگز نمی‌تواند.