# مدیریت حافظه در .NET

در این مقاله سعی شده است چگونگی ساخته شدن Objectها توسط برنامه ها , چگونگی مدیریت طول عمر اشیا در .NET و چگونگی آزاد شدن حافظه های گرفته شده توسط Collector Garbage شرح داده شود.

## درک مبانی کار Garbage Collector:

هر برنامه به نحوی از منابع مشخصی استفاده میکند. این منابع میتوانند فایلها، بافرهای حافظه، فضاهای صفحه نمایش، ارتباطات شبکه ای، منابع بانک اطلاعاتی و مانند اینها باشند. در حقیقت در یک محیط شیی گرا هر نوع داده تعریف شده در برنامه معرف یک سری منابع مربوط به آن برنامه هستند. برای استفاده از هر نوع از این داده ها لازم است که برای ارایه آن نوع مقداری حافظه تخصیص داده شود. موارد زیر برای دسترسی به یک منبع مورد نیاز است:

- تخصیص حافظه برای نوع داده ای که منبع مورد نظر را ارایه میدهد. این تخصیص حافظه با استفاده از دستور newobj در زبان IL صورت میگیرد که این دستور از ترجمه دستور new در زبانهایی مثل# C و Visual Basic و دیگر زبانهای برنامه نویسی ایجاد میشود.
- مقداردهی اولیه حافظه برای تنظیم حالت آغازین(Initial state) منابع و قابل استفاده کردن آن. توابع Constructor در این نوع داده ها مسئول این تنظیمات برای ایجاد این حالت آغازین هستند.
- استفاده از منابع با دسترسی به اعضای موجود در نوع داده.
- از بین بردن حالت کلی منابع برای پاک کردن آن.
- آزادسازی حافظه. Garbage Collector مسئول مطلق این مرحله به شمار می رود.

این نمونه به ظاهر ساده یکی از ریشه های اصلی خطاهای ایجاد شده در برنامه نویسی به شمار میرود. مواقع زیادی پیش می آید که برنامه نویس آزادسازی یک حافظه را وقتی دیگر مورد نیاز نیست فراموش می کند. مواقع زیادی پیش می آید که برنامه نویس از یک حافظه که قبلا آزاد شده استفاده کند.

این دو باگ برنامه ها از اکثر آنها بدتراند زیرا معمولا برنامه نویس نمیتواند ترتیب یا زمان به وجود آمدن این خطاها را پیش بینی کند. برای دیگر باگها شما میتوانید با مشاهده رفتار اشتباه یک برنامه آن را به سادگی تصحیح کنید. اما این دو باگ موجب نشت منابع (Leak Resource) (مصرف بیجای حافظه) و از بین رفتن پایداری اشیا میشوند که کارایی برنامه را در زمانهای مختلف تغییر میدهد. برای کمک به یک برنامه نویس

برای تشخیص این نوع خطاها ابزارهای ویژه ای مانند Windows Task Manager و System Control Monitor ActiveX و NuMega Bounds Checker و ... طراحی شده اند.

یک مدیریت منبع مناسب بسیار مشکل و خسته کننده است. این مورد تمرکز برنامه نویس را بر روی مطلب اصلی از بین میبرد. به همین دلیل نیاز به یک مکانیسم که مدیریت حافظه را به بهترین نحو انجام دهد در این زمینه به وضوح احساس میشد. در پلتفرم .NET این امر توسط Collector Garbage انجام میشود.

Garbage Collection کاملا برنامه نویس را از کنترل استفاده از حافظه و بررسی زمان آزادسازی آن راحت میکند. اگرچه Garbage Collector درمورد منابع ارائه شده توسط نوع داده در حافظه هیچ چیز نمیداند، یعنی Collector Garbage نمیداند چه طور میتواند مرحله 4 از موارد بالا را انجام دهد: از بین بردن حالت کلی منابع برای پاک کردن آن. برنامه نویس باید کدهای مربوط به این قسمت را انجام دهد چون او میداند باید چه گونه حافظه را به درستی و کاملا آزاد کند. البته Garbage Collector میتواند در این زمینه نیز قسمتهایی از کار را برای برنامه نویس انجام دهد

البته، بیشتر نوع داده ها، مانند Int32، Point، Rectangle، String، ArrayList و SerializationInfo از منابعی استفاده می کنند که احتیاجی به نوع ویژه ای از آزادسازی حافظه ندارند. برای مثال منابع یک شئی از نوع Point به راحتی و با نابود کردن فیلدهای X و Y در حافظه شیی آزاد میشود.

از طرف دیگر، یک نوع داده که منابع مدیریت نشده ای را ارائه میدهد، مانند یک فایل، یک ارتباط بانک اطلاعاتی، یک سوکت، یک Bitmap، یک آیکون و مانند اینها همیشه به اجرای مقداری کد ویژه برای آزاد کردن حافظه گرفته شده نیاز دارند.

CLR نیاز دارد که حافظه تمام منابع از یک heap مخصوص که managed heap نامیده میشود تخصیص داده شود. این heap شبیه heap زمان اجرای C است و فقط از یک لحاظ متفاوت است و آن این است که در این heap شما هیچ وقت حافظه تخصیص داده شده را آزاد نمیکنید. در حقیقت اشیا موجود در این heap وقتی دیگر نیازی به آنها نباشد آزاد میشوند. این مورد این سوال را ایجاد میکند که چگونه heap managed متوجه میشود که دیگر نیازی به یک شیی خاص نیست؟

چندین الگوریتم از Collector Garbage در حال حاضر در مرحله آزمایش هستند و هر کدام از این الگوریتم ها برای یک محیط خاص و نیز برای کسب بهترین راندمان بهینه سازی شده اند. در این مقاله روی الگوریتم Garbage Collector استفاده شده در Microsoft .NET Framework CLR متمرکز شده است.

زمانی که یک پروسه مقداردهی اولیه(Initialize) میشود، CLR یک قسمت پیوسته از آدرس حافظه را برای آن اختصاص میدهد این آدرس فضای حافظه managed heap نامیده میشود. این heap همچنین یک اشاره گر مخصوص هم دارد که ما از این به بعد آن را NextObjPtr می نامیم. این اشاره گر مکان قرار گیری شیی بعدی را در heap مشخص میکند. در ابتدا این اشاره گر به آدرس ابتدای فضای گرفته شده برای managed heap اشاره میکند.

دستور newobj در زبان IL باعث ایجاد یک شیی جدید میشود. بیشتر زبانها از جمله# C و Visual Basic برای درج این دستور در کد IL عملگر new را در برنامه ارائه میدهند. این دستور IL باعث میشود که CLR مراحل زیر را انجام دهد:

1. محاسبه تعداد بایتهای مورد نیاز برای این نوع داده
2. اضافه کردن بایتهای مورد نیاز برای overhead شیی. هر شیی دو فیلد overhead دارد: یک اشاره گر به جدول تابع و یک SyncBlockIndex. در سیستمهای 32بیتی، هر کدام از این فیلدها 32 بیت هستند، که 8 بایت را به هر شیی اضافه می کند. در سیستم های 64 بیتی، هر کدام از این فیلدها 64 بیت است که 16 بایت را برای هر شیی اضافه می کند.
3. سپس CLR چک میکند که حافظه مورد نیاز برای شیی جدید در managed heap موجود باشد. اگر فضای کافی موجود باشد این شیی در آدرسی که NextObjPtr به آن اشاره میکند ایجاد میشود. تابع constructor شیی مذکور فراخوانی میشود (اشاره گر NextObjPtr به عنوان پارامتر this به constructor فرستاده میشود) و دستور newobj آدرس شیی ایجاد شده را برمیگرداند. درست قبل از اینکه آدرس برگردانده شود، NextObjPtr به بعد از شیی ایجاد شده پیشروی میکند و مثل قبل آدرسی که باید شیی بعدی در آن قرار گیرد را در خود نگه میدارد.

اگر یک شیی جدید ایجاد شود این شیی دقیقا در جایی که NextObjPtr به آن اشاره میکند قرار میگیرد.

در عوض اجازه دهید تخصیص حافظه را در heap زمان اجرای C بررسی کنیم. در یک heap زمان اجرای C تخصیص حافظه برای یک شی به حرکت در میان ساختارهای داده از یک لیست پیوندی نیاز دارد. زمانی که یک بلاک حافظه با اندازه لازم پیدا شد این بلاک حافظه تقسیم میشود و شیی مذکور در آن ایجاد میشود و اشاره گرهای موجود در لیست پیوندی برای نگه داری در آن شیی تغییر داده میشوند. برای managed heap تخصیص حافظه برای یک شیی به معنای اضافه کردن یک مقدار به اشاره گر است. در حقیقت تخصیص حافظه به یک شیی در managed heap تقریبا به سرعت ایجاد یک متغییر در stack است! به علاوه در بیشتر heapها مانند heap زمان اجرای C حافظه در جایی اختصاص داده میشود که فضای خالی کافی یافت شود. بنابراین اگر چند شیی بلافاصله بعد از هم در برنامه ایجاد شوند، ممکن است این اشیا چندین مگابایت آدرس حافظه با هم فاصله داشته باشند ولی در managed heap ایجاد چند شیی بلافاصله بعد از هم باعث قرار گرفتن ترتیبی این اشیا در حافظه میشود.

در بیشتر برنامه ها وقتی برای یک شیی حافظه در نظر گرفته میشود که یا بخواهد با یک شیی دیگر ارتباط قوی داشته باشد یا بخواهد چندین بار در یک قطعه کد استفاده شود. برای مثال معمولا وقتی یک حافظه برای شیی BinaryWriter ایجاد شد بلافاصله بعد از آن یک حافظه برای FileStream گرفته شود. سپس برنامه از BinaryWriter استفاده میکند که در حقیقت به صورت درونی از شیی FileStream هم استفاده میکند.

در یک محیط کنترل شده به وسیله Garbage Collector برای اشیای جدید به صورت متوالی فضا در نظر گرفته میشود که این عمل موجب افزایش راندمان بدلیل موقعیت ارجاعها میشود. به ویژه این مورد به این معنی است که مجموعه کارهای پروسه شما کمتر شده و این نیز متشابه قرار گرفتن اشیای مورد استفاده توسط برنامه در Cache CPU است.

تا کنون اینگونه به نظر میرسید که managed heap بسیار برتر از heap زمان اجرای C است و این نیز به دلیل سادگی پیاده سازی و سرعت آن است. اما نکته دیگری که اینجا باید در نظر گرفته شود این است که managed heap این توانایی ها را به این دلیل به دست می آورد که یک فرض بزرگ انجام میدهد و آن فرض این است که فضای آدرس و حافظه بینهایت هستند. به وضوح این فرض کمی خنده دار به نظر میرسد و مسلما managed heap باید یک مکانیسم ویژه ای را به کار برد تا بتواند این فرض را انجام دهد. این مکانیسم Garbage Collector نامیده میشود ، که در ادامه طرز کار آن شرح داده میشود.

زمانی که یک برنامه عملگر new را فراخوانی میکند ممکن است فضای خالی کافی برای شیی مورد نظر وجود نداشته باشد. heap این موضوع را با اضافه کردن حجم مورد نیاز به آدرس موجود در NextObjPtr متوجه میشود. اگر نتیجه از فضای در نظر گرفته شده برای برنامه تجاوز کرد heap پر شده است و Garbage Collector باید آغاز به کار کند.

مهم: مطالبی که ذکر شد در حقیقیت صورت ساده شده مسئله بود. در واقعیت یک Garbage Collection زمانی رخ می دهد که نسل صفر کامل شود. بعضی Garbage Collector ها از نسل ها استفاده می کنند که یک مکانیسم به شمار می رود و هدف اصلی آن افزایش کارایی است. ایده اصلی به این صورت است که اشیای تازه ایجاد شده نسل صفر به شمار می روند و اشیایی قدیمی تر در طول عمر برنامه در نسل های بالاتر قرار می گیرند. جداسازی اشیا و دسته بندی انها به نسل های مختلف می تواند به Garbage Collector اجازه دهد اشیایی موجود در نسل خاصی را به جای تمام اشیا مورد بررسی قرار دهد. در بخش های بعدی نسل ها با جزئیات تمام شرح داده می شوند. اما تا ان مرحله فرض می شود که Garbage collector وقتی رخ می دهد که heap پر شود.

## الگوریتم Garbage Collection:

Garbage Collection بررسی می کند که ایا در heap شیی وجود دارد که دیگر توسط برنامه استفاده نشود. اگر چنین اشیای در برنامه موجود باشند حافظه گرفته شده توسط این اشیا آزاد میشود (اگر هیچ حافظه ای برای اشیای جدید در heap موجود نباشد خطای OutOfMemoryException توسط عملگر new رخ میدهد). اما چگونه Garbage Collector تشخیص میدهد که آیا برنامه یک متغیر را نیاز دارد یا خیر؟ همانطور که ممکن است تصور کنید این سوال پاسخ ساده ای ندارد. هر برنامه دارای یک مجموعه از rootها است. یک root اشاره گری است به یک نوع داده ارجاعی. این اشاره گر یا به یک نوع داده ارجاعی در managed heap اشاره میکند یا با مقدار null مقدار دهی شده است.برای مثال تمام متغییرهای استاتیک و یا عمومی(Global Variables) یک root به شمار میروند . به علاوه هر متغیر محلی که از نوع ارجاع باشد و یا پارامترهای توابع در stack نیز یک root به شمار میروند. در نهایت، درون یک تابع، یک ثبات CPU که به یک شیی از نوع ارجاع اشاره کند نیز یک root به شمار میرود.

زمانی که کامپایلر JIT یک کد IL را کامپایل میکند علاوه بر تولید کدهای Native یک جدول داخلی نیز تشکیل میدهد. منطقا هر ردیف از این جدول یک محدوده از بایتهای آفست را در دستورات محلی CPU برای تابع نشان میدهند و برای هر کدام از این محدوده ها یک مجموعه از آدرسهای حافظه یا ثباتهای CPU را که محتوی rootها هستند مشخص میکند. برای مثال جدول ممکن است مانند جدول زیر باشد:

Sample of a JIT compiler-produced table showing mapping of native code offsets to a method's roots
Starting Byte Offset Ending Byte Offset Roots
0x00000000 0x00000020 this, arg1, arg2, ECX ,EDX
0x00000021 0x00000122 this, arg2, fs, EBX
0x00000123 0x00000145 fs

اگر یک Garbage Collector زمانی که کدی بین آفست x000000210 و x000001220 در حال اجرا است آغاز شود، Garbage Collector میداند که پارامترهای this و arg2 و متغییرهای محلی fs و ثبات EBX همه root هستند و به اشیایی درون heap اشاره میکنند که نباید زباله تلقی شوند. به علاوه Garbage Collector میتواند بین stack حرکت کند و rootها را برای تمام توابع فراخوانی شده با امتحان کردن جدول داخلی هر کدام از این توابع مشخص کند. Garbage Collector وسیله دیگری را برای بدست آوردن مجموعه rootهای نگه داری شده توسط متغیرهای ارجاعی استاتیک و عمومی به کار میبرد.

نکته: در جدول بالا توجه کنید که آرگومان arg1 تابع بعد از دستورات CPU در آفست x000000200 دیگر به چیزی اشاره نمیکند و این امر بدین معنی است که شیئی که arg1 به آن اشاره میکند هر زمان بعد از اجرای این دستورات میتواند توسط Garbage Collector جمع آوری شود (البته فرض بر اینکه هیچ شیئی دیگری در برنامه به شیئی مورد ارجاع توسط arg1 اشاره نمیکند). به عبارت دیگر به محض اینکه یک شیئی غیر قابل دسترسی باشد برای جمع آوری شدن توسط Garbage Collector داوطلب میشود و به همین علت باقی ماندن اشیا تا پایان یک متد توسط Garbage Collector تضمین نمیشود.

با وجود این زمانی که یک برنامه زمانی که در حالت debug اجرا شده باشد و یا ویژگی System.Diagnostics.DebuggableAttribute به اسمبلی برنامه اظافه شده باشد و یا اینکه پارامتر isJITOptimizeDisabled با مقدار true در constructor برنامه تنظیم شده باشد، کامپایلر JIT طول عمر تمام متغیرها را، چه از نوع ارجاعی و چه از نوع مقدار، تا پایان محدوده شان افزایش میدهد که معمولا همان پایان تابع است( کامپایلر C# مایکروسافت یک سوییچ خط فرمان به نام /debug را ارائه میدهد که باعث اضافه شدن DebuggableAttribute به اسمبلی میشود و نیز پارامتر isJITOptimizeDisabled را نیز true میکند). این افزایش طول عمر از جمع آوری شدن متغیرها توسط Garbage Collector در محدوده اجرایی آنها در طول برنامه جلوگیری میکند و این عمل فقط در زمان debug یک برنامه مفید واقع میشود.

زمانی که Garbage Collector شروع به کار میکند، فرض میکند که تمام اشیای موجود در heap زباله هستند. به عبارت دیگر فرض میکند که هیچ کدام از rootهای برنامه به هیچ شیی در heap اشاره نمیکند. سپس Garbage Collector شروع به حرکت در میان rootهای برنامه میکند و یک گراف از تمام rootهای قابل دسترسی تشکیل میدهد. برای مثال Garbage Collector ممکن است یک متغیر عمومی را که به یک شیی در heap اشاره میکند موقعیت یابی کند.

زمانی که این بخش از گراف کامل شد Garbage Collector، rootهای بعدی را چک میکند و مجددا اشیا را بررسی میکند. در طول اینکه Garbage Collector از یک شیی به یک شیی دیگر منتقل میشود، اگر سعی کند که یک شیی تکراری را به گراف اضافه کند، Garbage Collector حرکت در آن مسیر را متوقف میکند. این نوع رفتار دو هدف را دنبال میکند: اول اینکه چون Garbage Collector از هیچ شیی دو بار عبور نمیکند راندمان برنامه را به شکل قابل توجهی افزایش میدهد. دوم اینکه هرچقدر هم در برنامه لیستهای پیوندی دایره ای از اشیا موجود باشند ، Garbage Collector در حلقه های بینهایت نمی ماند.

زمانی که تمام rootها بررسی شدند، گراف Garbage Collector محتوی تمام اشیایی است که به نحوی از طریق rootهای برنامه قابل دسترسی می باشند و هر شیی که در گراف نباشد به این معنی است که توسط برنامه قابل دسترسی نیست و یک زباله محسوب میشود. بعد از این Garbage Collector به صورت خطی heap را طی میکند و دنبال بلاکهای پیوسته از زباله های مشخص شده توسط گراف میگردد (که هم اکنون فضای خالی محسوب میشوند). اگر بلاکهای کوچکی پیدا شوند Garbage Collector این بلاکها را به همان حال قبلی رها میکند.

اگر یک بلاک پیوسته وسیع توسط Garbage Collector یافت شد، در این حال، Garbage Collector اشیای غیر زباله را به سمت پایین حافظه heap شیفت میدهد (و این کار با تابع استاندارد memcopy انجام میشود) و به این طریق heap را فشرده میکند. طبیعتا حرکت دادن اشیا به سمت پایین در heap باعث نامعتبر شدن تمام اشاره گرهای موجود برای آن اشیا میشود. به علاوه، اگر شیی محتوی اشاره گری به شیی دیگری بود Garbage Collector مسئول تصحیح این اشاره گرها میشود. بعد از این که این عمل فشرده سازی روی heap انجام شد NextObjPtr به آخرین شیی غیر زباله اشاره میکند.

Garbage Collector یک افزایش بازدهی قابل توجهی را ایجاد میکند. اما به یاد داشته باشید زمانی Garbage Collector شروع به کار میکند که نسل صفر کامل شود و تا آن زمان managed heap به صورت قابل توجهی سریعتر از heap زمان اجرای C است. در نهایت Garbage Collector مربوط به CLR روش بهینه سازی را ارائه می دهد که راندمان کاری Garbage Collector را مقدار زیادی افزایش می دهد.

کد زیر نشان میدهد که چگونه به اشیا حافظه تخصیص داده میشود و آنها مدیریت میشوند:

```
Class App
}
static void Main()
}
 //ArrayList object created in heap, a is now a root
```

```
ArrayList a = new ArrayList;()

 //Create 10000 objects in the heap
for(Int32 x=0;x&lt;10000;x++
}
a.Add(new Object()); // Object created in heap
{

 //Right now, a is a root (on the thread's stack). So a is
 //reachable and the 10000 objects it refers to
 //are reachable.
Console.WriteLine(a.Length;(

 //After a.Length returns, a isn't referred to in the code
 //and is no longer a root. If another thread were to start
 //a garbage collection before the result of a.Length were
 //passed to WriteLine, the 10001 objects would have their
 //memory reclaimed.

Console.WriteLine("End Of Method;("
{
{
```

**نکته: اگر فکر میکنید که** Garbage Collector **یک تکنولوژی با ارزش محسوب میشود، ممکن است تعجب کنید که چرا در** ANSI C++ **قرار نمی گیرد. دلیل این مورد این است که** Garbage Collector **احتیاج دارد که** root**های موجود در برنامه را تعیین هویت کند و نیز باید بتواند تمام اشاره گرهای اشیا را پیدا کند. مشکل با** C++ **مدیریت نشده این است که این برنامه تغییر نوع یک اشاره گر را از یک نوع به یک نوع دیگر مجاز میداند و هیچ راهی برای فهمیدن این که این اشاره گر به چه چیز اشاره میکند وجود ندارد. در** CLR، managed heap **همیشه میداند که نوع واقعی یک شیی چیست و از اطلاعات** metadata **برای مشخص کردن اینکه کدام عضوها از یک شی به اشیای دیگر اشاره می کنند استفاده می کند.**

**نسلها :**

همانطور که پیشتر ذکر شد نسلها مکانیسمی درون CLR Garbage Collector به شمار میرود که هدف اصلی آن بهبود کارایی برنامه است. یک Garbage Collector که با مکانیسم نسلها کار میکند (همچنین به عنوان Garbage Collector زودگذر هم نامیده میشود) فرضهای زیر را برای کار خود در نظر میگیرد:

1)هر چه یک شیئ جدیدتر ایجاد شده باشد طول عمر کوتاهتری هم خواهد داشت.
2)هر چه یک شیئ قدیمیتر باشد طول عمر بلندتری هم خواهد داشت.
3)جمع آوری قسمتی از heap سریعتر از جمع آوری کل آن است.

مطالعات زیادی معتبر بودن این فرضیات را برای مجموعه بزرگی از برنامه های موجود تایید کرده اند و این فرضیات بر طرز پیاده سازی Garbage Collector تاثیر داشته اند. در این قسمت طرز کار این این مکانیسم شرح داده شده است.

زمانی که یک managed heap برای بار اول ایجاد میشود دارای هیچ شیئ نیست. اشیایی که به heap اظافه شوند در نسل صفر قرار میگیرند. اشیای موجود در نسل صفر اشیای تازه ایجاد شده ای هستند که تا کنون توسط Garbage Collector بررسی نشده اند. تصویر زیر یک برنامه را که تازه آغاز به کار کرده است نشان میدهد که دارای پنج شیئ است(از A تا E ). بعد از مدتی اشیای C و E غیر قابل دسترسی میشوند.

زمانی که CLR آغاز به کار میکند یک مقدار نهایی را برای نسل صفر در نظر میگیرد که به طور پیش فرض 256 کیلوبایت است(البته این مقدار مورد تغییر قرار میگیرد). بنابراین اگر شیی بخواهد ایجاد شود و در نسل صفر فضای کافی وجود نداشته باشد Garbage Collector آغز به کار میکند. اجازه دهید تصور کنیم که اشیای A تا E 256 کیلوبایت فضا اشغال کرده اند زمانی که شیی F بخواهد تشکیل شود Garbage Collector باید آغاز به کار کند. Garbage Collector تشخیص میدهد که اشیای C و E زباله محسوب میشوند و بنابراین شیی D باید فشرده شود بنابراین این شیی به کنار شیی B میرود. اشیایی که بعد از این مرحله باقی میمانند (اشیای A و B و D) وارد نسل یک میشوند. اشیای موجود در نسل یک به این معنی هستند که یک بار توسط Garbage Collector بررسی شده اند.

بعد از یک بار اجرای Garbage Collector هیچ شیی در نسل صفر باقی نمی ماند. مثل همیشه اشیایی که بعد از این ایجاد می شوند به نسل صفر اضافه میشوند. شکل زیر اجرای برنامه و به وجود آمدن اشیای F تا K را نشان میدهد. به علاوه در طول اجرای برنامه اشیای B و H و J غیر قابل استفاده شده اند و حافظه گرفته شده توسط آنها باید آزاد شود.

حال فرض کنیم با تخصیص حافظه برای شی L در نسل صفر مقدار داده های موجود در این نسل از 256کیلوبایت فراتر رود. چون نسل صفر به سرحد خود رسیده است Garbage Collector باید آغاز به کار کند. زمانی که عمل Garbage Collection آغاز میشود Garbage Collector باید تصمیم بگیرد که کدام نسل باید مورد بررسی قرار گیرد. پیشتر ذکر شد که زمانی که CLR آغاز به کار میکند برای نسل صفر 256

کیلوبایت فضا اختصاص میدهد. همچنین CLR یک سرحد نیز برای نسل یک درنظر میگیرد. فرض میکنیم این مقدار فضا شامل 2 مگابایت باشد.

زمانی که عمل Garbage Collection انجام میشود، Garbage Collector همچنین بررسی میکند که چه مقدار فضا توسط نسل یک اشغال شده است. در این حالت نسل یک فضایی کمتر از 2 مگابایت را اشغال کرده است بنابراین Garbage Collector فقط نسل صفر را مورد بررسی قرار میدهد. یک بار دیگر فرضیات Garbage Collector را که در ابتدا ذکر شد مرور کنید. اولین فرض این بود که اشیای تازه ایجاد شده دارای عمر کوتاه تری هستند. بنابراین این گونه به نظر میرسد که نسل صفر دارای بیشترین مقدار زباله باشد و جمع آوری حافظه از این نسل موجب آزاد سازی مقدار زیادی حافظه میشود. بنابراین Garbage Collector نسل یک را رها میکند و فقط به جمع آوری نسل صفر میپردازد که این عمل موجب افزایش سرعت کارکرد پروسه Garbage Collector میشود.

به وضوح، رها سازی اشیای نسل یک وجب افزایش سرعت و کارایی Garbage Collector میشود. با وجود این کارایی Garbage Collector بیشتر افزایش پیدا میکند چون تمام اشیای موجود در Managed Heap را بررسی نمیکند. اگر یک root یا یک شیی از این نسل به شیی از نسل قدیمی تر اشاره کند، Garbage Collector میتواند ارجاعات داخلی اشیای قدیمی تر را در نظر نگیرد، و بدین وسیله زمان مورد نیاز را برای تشکیل گرافی از اشیای قابل دسترس کاهش میدهد. البته این امر ممکن است که یک شیی قدیمی به یک شیی جدید اشاره کند. برای اطمینان از این که این شیی قدیمی نیز مورد بررسی قرار میگیرد Garbage Collector از یک مکانیسم داخلی JIT استفاده میکند به این نحو که زمانی که یک فیلد ارجاع یک شیی تغییر کرد یک بیت را تنظیم میکند. این پشتیبانی توسط JIT باعث میشود که Garbage Collector بتواند تشخیص دهد کدام از اشیای قدیمی از زمان آخرین عمل جمع آوری تا کنون تغییر کرده اند. فقط اشیای قدیمی که دارای فیلدهای تغییر کرده هستند احتیاج به بررسی شدن برای اینکه آیا به شیی از نسل صفر اشاره میکنند احتیاج دارند.

نکته: تستهای کارایی مایکروسافت نشان میدهند که عمل Garbage Collection در نسل صفر کمتر از یک میلی ثانیه در یک کامپیوتر پنتیوم با سرعت 200 مگاهرتز زمان میبرد.

یک Garbage Collector که از نسلها استفاده میکند همچنین تصور میکند که اشیایی که مدت زیادی است که در حافظه مانده اند به زودی نیز از حافظه خارج نمیشوند. بنابراین این احتمال میرود که اشیای موجود در نسل یک همچنان در طول برنامه قابل دسترس خواهند بود. بنابراین اگر Garbage Collector اشیای موجود در نسل یک را بررسی کند احتمالا مقدار زیادی متغییر غیر قابل دسترسی در برنامه نخواهد یافت و احتمالا حافظه زیادی را آزاد نخواهد کرد. بنابراین آزاد سازی نسل یک چیزی جز اتلاف وقت نخواهد بود. اگر هر زباله ای در نسل یک به وجود بیاید در همان نسل باقی خواهد ماند.

همانطور که میبینید تمام اشیای که از نسل صفر باقی مانده اند وارد نسل یک شده اند. چون Garbage Collector نسل یک را بررسی نمیکند شیی B حافظه ای را که گرفته است آزاد نمیکند با وجود اینکه از آخرین عمل Garbage Collector تا کنون این متغییر در برنامه قابل استفاده نبوده است. مجددا بعد از جمع آوری نسل صفر دارای هیچ شیی نخواهد بود و بنابراین مکانی برای قرارگیری اشیای جدید محسوب میشود.

در ادامه برنامه به کار خود ادامه میدهد و اشیای L تا O را ایجاد میکند. و در حال اجرا برنامه استفاده از اشیای G و L و M را پایان میدهد و آنها را غیر قابل دسترس میکند. بنابراین heap به صورت زیر تبدیل میشود

اجازه دهید فکر کنیم که تخصیص حافظه برای شیی P باعث تجاوز نسل صفر از سرحد خود شود و این عمل موجب اجرای مجدد Garbage Collector شود. چون تمام اشیای موجود در نسل یک کمتر از 2 مگابایت است Garbage Collector مجددا تصمیم میگیرد که فقط نسل صفر را بررسی کند و از اشیای غیر قابل دسترسی در نسل یک چشم پوشی کند(اشیای B و G). بعد از عمل جمع آوری heap به صورت زیر در می آید.

مانند قبل، اشیایی که در این مرحله از جمع آوری از نسل صفر باقی ماندند وارد نسل یک میشوند و نیز اشیایی که در این مرحله از نسل یک باقی ماندند وارد نسل دو میشوند. مثل همیشه، بلافاصله نسل صفر از اشیا خالی میشود و اشیای جدید میتوانند در این قسمت قرار گیرند. اشیای موجود در نسل دو اشیای هستند که حداقل دو بار توسط Garbage Collector مورد بررسی قرار گرفته اند. ممکن است بعد از یک یا دو بار جمع آوری نسل صفر، با انجام این عمل در نسل یک مقداری حافظه آزاد شود، اما این عمل تا زمانی که نسل یک به سرحد خود نرسد انجام نمیشود که این کار ممکن است نیاز به چندین بار اجرای جمع آوری در نسل صفر باشد.

Managed heap فقط سه نسل را پشتیبانی میکند: نسل صفر، نسل یک و نسل دو. بنابراین چیزی به نام نسل سه وجود ندارد. زمانی که CLR آغاز به کار میکند، سرحدهایی را برای هر سه نسل در نظر میگیرد. همانطور که پیشتر ذکر شد، سرحد برای نسل صفر حدود 256کیلوبایت است، سرحد برای نسل یک حدودا 2 مگابایت است و سرحد برای نسل دو حدود 10 مگابایت است. بنابراین سرحد نسلها به گونه ای انتخاب شده است که موجب افزایش بازدهی و راندمان برنامه شود. هرچه سرحد یک نسل بیشتر باشد عمل Garbage Collection کمتر روی آن نسل صورت میگیرد. و دوباره، بهبود کارایی به وجود می آید که به دلیل فرضیات اولیه است: اشیای جدید دارای طول عمر کوتاهتری هستند، اشیای قدیمی طول عمر بیشتری دارند.

Garbage Collector موجود در CLR یک جمع آوری کننده با تنظیم کننده خودکار است. این بدین معنا است که Garbage Collector از رفتار برنامه شما می آموزد که چه زمانی باید عمل جمع آوری را انجام دهد. برای مثال اگر برنامه شما اشیای زیادی را ایجاد کند و از آنها برای مدت زمان کوتاهی استفاده کند، این امر ممکن است که آزاد سازی حافظه در نسل صفر مقدار زیادی حافظه را آزاد کند.حتی ممکن است تمام حافظه گرفته شده در نسل صفر آزاد شود.

اگر Garbage Collector مشاهده کند که بعد از انجام جمع آوری نسل یک تعداد محدودی از اشیا باقی ماندند، ممکن است که تصمیم بگیرد که سرحد نسل صفر را از 256 کیلوبایت به 128 کیلوبایت کاهش دهد. این کاهش در فضای معین به این معنی است که عمل جمع آوری باید در فواصل زمانی کوتاه تری رخ دهد اما فضای کمتری را بررسی کند. بنابراین کارهای پروسه شما به صورت قابل توجهی افزایش نمی یابد. اگر تمام اشیای موجود در نسل صفر زباله محسوب شوند دیگر احتیاجی به فشرده سازی حافظه توسط Garbage Collector نیست. این عمل میتواند به سادگی با آوردن اشاره گر NextObjPtr به ابتدای حافظه مورد نظر برای نسل صفر انجام شود. این عمل به سرعت حافظه را آزاد میکند!

نکته:Garbage Collector به بهترین نحو با برنامه های ASP.NET و سرویسهای وب مبتنی بر XML کار میکند. برای برنامه های تحت ASP.NET، یک تقاضا از طرف کلاینت میرسد، یک جعبه از اشیای جدید

تشکیل میشود، اشیا کارهای تعیین شده توسط کلاینت را انجام میدهند، و نتیجه به سمت کلاینت بر میگردد. در این مرحله تمام اشیای موجود برای انجام تقاضای کلاینت زباله تلقی میشوند. به بیان دیگر، هر تقاضای برنامه های تحت ASP.NET باعث ایجاد حجم زیادی از زباله میشوند. چون این اشیا اغلب بلافاصله بعد از ایجاد دیگر قابل دسترسی نیستند هر عمل جمع آوری موجب آزاد سازی مقدار زیادی از حافظه میشود. این کار مجموعه کارهای پروسه را بسیار کاهش میدهد بنابراین راندمان Garbage Collector محسوس خواهد بود.

به بیان دیگر، اگر Garbage Collector نسل صفر را مورد بررسی قرار دهد و مشاهده کند که مقدار زیادی از اشیا وارد نسل یک شدند، مقدار زیادی از حافظه توسط Garbage Collection آزاد نمیشود، بنابراین Collector Garbage سرحد نسل صفر را تا 512 کیلوبایت افزایش میدهد. در این مرحله کمتر انجام میشود اما با هر بار انجام این عمل مقدار زیادی حافظه آزاد میشود. در طول این قسمت چگونگی تغییر دینامیک سرحد نسل صفر شرح داده شد. اما علاوه بر سرحد نسل صفر سرحد نسلهای یک و دو نیز بر اساس همین الگوریتم تغییر میکنند. به این معنی که زمانی که این نسلها مورد عمل جمع آوری قرار میگیرند Garbage Collector بررسی میکند که چه مقدار فضا آزاد شده است و چه مقدار از اشیا به نسل بعد رفته اند. بر اساس نتایج این بررسیها Garbage Collector ممکن است ظرفیت این نسلها را کاهش یا افزایش دهدکه باعث افزایش سرعت اجرای برنامه میشود.

## دیگر نتایج کارایی Garbage Collector:

پیشتر در این مقاله الگوریتم کار Garbage Collector شرح داده شد. با این وجود در طول این توضیحات یک فرض بزرگ صورت گرفته بود: اینکه فقط یک ترد در حال اجرا است. اما در مدل واقعی چندین ترد به managed heap دسترسی دارند و یا حداقل اشیای قرار گرفته در managed heap رو تغییر میدهند. زمانی که یک ترد موجب اجرای عمل جمع آوری توسط Garbage Collector میشود، دیگر تردها حق دسترسی به اشیای موجود در managed heap را ندارند(این مورد شامل ارجاعهای اشیای موجود در stack هم میشود) زیرا Collector Garbage ممکن است مکان این اشیا را تغییر دهد. بنابراین وقتی Collector Garbage بخواهد عمل جمع آوری را آغاز کند، تمام تردهایی که در حال اجرای کدهای مدیریت شده هستند به حال تعلیق در می آیند.CLR دارای چندین مکانیسم نسبتا" متفاوت است که میتواند تردها را به حالت تعلیق در آورد بنابراین عمل جمع آوری میتواند به درستی اجرا شود. دلیل اینکه CLR از چندین مکانیسم استفاده میکند به حالت اجرا نگاه داشتن تردها تا حداکثر زمان ممکن و کاهش سربار کردن آنها در حافظه تا حداقل زمان ممکن است. تشریح این مکانیسمها از اهداف این مقاله خارج است اما تا این حد لازم است ذکر شود که مایکروسافت فعالیتهای زیادی را برای کاهش فشار پردازشی ناشی از Collector Garbage انجام داده است. و نیز این مکانیسمها به سرعت در حال تغییر هستند تا به بهترین کارایی خود برسند.

زمانی که CLR میخواهد Garbage Collector را اجرا کند، ابتدا تمام تردها در پروسه جاری را که در حال اجرای کدهای مدیریت شده هستند به حال تعلیق در می آورد. سپس CLR برای تعیین موقعیت هر ترد تمام اشاره گرهای دستورات در حال اجرا توسط تردها را بررسی میکند. سپس برای تعین اینکه چه کدی توسط ترد در حال اجرا بوده آدرس اشاره گر دستور با جدول ایجاد شده توسط کامپایلر JIT مقایسه میشود.

اگر دستور درحال اجرا توسط ترد در یک آفست مشخص شده به وسیله جدول مذکور باشد گفته میشود که ترد به یک نقطه امن دسترسی دارد. یک نقطه امن نقطه ای است که در آنجا میتوان بدون هیچ مشکلی ترد را به حال تعلیق در آورد تا Garbage Collector کار خود را آغاز کند.اگر اشاره گر دستور در حال اجرای ترد در روی یک آفست مشخص شده توسط جدول درونی تابع قرار نداشت، بنابراین ترد در یک نقطه امن قرار ندارد و CLR نمیتواند Garbage Collector را اجرا کند. در این حالتCLR ترد را هایجک میکند: به این معنی که CLR استک مربوط به ترد را به گونه ای تغییر میدهد که آدرس بازگشت به یک تابع خاص پیاده سازی شده درون CLR اشاره کند. سپس ترد به ادامه کار خود بازمیگردد. زمانی که متد در حال اجرا توسط ترد ادامه پیدا کند، این تابع ویژه اجرا خواهد شد و ترد به حالت تعلق درخواهدآمد.

با وجود این ممکن است در بعضی مواقع ترد از متد خود بازنگردد. بنابراین زمانی که ترد به اجرای خود ادامه میدهد، CLR 250میلی ثانیه صبر میکند. سپس دوباره بررسی میکند که آیا ترد به یک نقطه امن طبق جدول JIT رسیده است یا نه. اگر ترد به یک نقطه امن رسیده بود CLR ترد را به حالت تعلیق درمی آورد و Garbage Collector را اجرا میکند در غیر این صورت مجددا سعی میکند با تغییر Stack مربوط به ترد اجرای آن را به تابع دیگری انتقال دهد در صورت شکست مجددا CLR برای چند میلی ثانیه دیگر نیز صبر میکند. زمانی که تمام تردها به یک نقطه امن رسیدند یا اینکه با موفقیت هایجک شدند، Garbage Collector میتواند کار خود را آغاز کند. زمانی که عمل جمع آوری انجام شد تمام تردها به وضعیت قبلی خود برمیگردند و اجرای برنامه ادامه پیدا میکند. تردهای هایجک شده هم به متدهای اولیه خود بازمیگردند.

نکته: این الگوریتم یک پیچ خوردگی کوچک دارد. اگر CLR یک ترد را به حالت تعویق درآورد و دریابد که ترد در حال اجرای یک کد مدیریت نشده بود آدرس بازگشت ترد هایجک میشود و به ترد اجازه داده میشود که به اجرای خود ادامه دهد. با این وجود در این حالت به Garbage Collector اجازه داده میشود که اجرا شود در حالی که ترد مذکور در حال اجرا است. این مورد هیچ اشکالی را به وجود نمی آورد زیرا کدهای مدیریت نشده به اشیای موجود در managed heap دسترسی ندارندتا زمانی که آن اشیا پین شوند. یک شیی پین شده شی است که Garbage Collector حق حرکت دادن آن را در heap managed ندارد. اگر تردی که در حال حاضر در حال اجرای یک کد مدیریت نشده بود، شروع به اجرای یک کد مدیریت شده کند، ترد هایجک میشود و به حالت تعلیق درمی آید تا زمانی که Garbage Collection به درستی به اتمام برسد.

علاوه بر مکانیسمهای ذکر شده(نسلها، نقاط امن، و هایجک کردن)، Garbage Collector از بعضی از مکانیسمهای اضافی دیگری نیز استفاده میکند که باعث افزایش بازدهی آن میشود.

**اشیای بزرگ:**

**فقط یک نکته قابل ذکر دیگر که باعث افزایش سرعت و بازدهی بهتر میشود باقی مانده است. هر شیی که** 85000 **بایت یا بیشتر فضای حافظه را اشغال کند یک شیی بزرگ در نظر گرفته میشود. اشیای بزرگ در یک** heap **ویژه اشیای بزرگ قرار میگیرند. اشیای درون این** heap **مانند اشیای کوچک (که راجع به آنها صحبت شد)** finalize **و آزاد میشوند. با این وجود این اشیا هیچ وقت تحت فشرده سازی قرار نمیگیرند زیرا شیفت دادن** 85000 **بایت بلاک حافظه درون** heap **مقدار زیادی از زمان** CPU **را هدر میدهد. اشیای بزرگ همواره به عنوان نسل دو در نظر گرفته میشوند، بنابراین این اشیا باید فقط برای منابعی که مدت زمان زیادی در حافظه می مانند ایجاد شوند. تخصیص اشیایی که دارای طول عمر کوتاه هستند در قسمت اشیای بزرگ باعث میشود که عمل جمع آوری نسل دو سریعتر انجام شود و این مورد نیز به بازدهی و کارایی برنامه صدمه وارد میکند.**

**نویسنده :** Omid_Ahmadi

**مرجع :** www.barnamenevis.com

**منبع :** Applied Microsoft .NET Programming **از** MS Press **فصل** 21