Peter Varhol is werkzaam bij Compuware en is lid van de senior development staff.

Geheugen optimaliseren in .NET-toepassingen

CLR HAALT GEHEUGEN TERUG UIT TOEPASSINGEN

Met Microsoft .NET Framework en CLR (Common Language Runtime) komt er voorgoed verandering in de manier waarop ontwikkelaars toepassingen maken voor het Windows-platform. Ontwikkelaars hoeven zich namelijk niet langer bezig te houden met het beheren van het toepassingsgeheugen. Dit saaie en foutgevoelige werk wordt voortaan geheel automatisch door .NET Framework verzorgd.

Maar zoals altijd geldt bij het schrijven van toepassingen: niets voor niets. Er is veel rekenkracht nodig om niet-gebruikt geheugen te identificeren en te verzamelen en het vervolgens weer bij het beschikbare geheugen te voegen. Toepassingsgeheugen dat slecht wordt beheerd, kan op den duur leiden tot subtiele, moeilijk te traceren fouten. Dergelijke fouten hebben niet alleen performanceproblemen tot gevolg, maar beperken ook de schaalbaarheid en betrouwbaarheid van de toepassing. De meeste ontwikkelaars zijn nog niet bekend met deze geheugenproblemen. Het ontbreekt ontwikkelaars veelal aan de ervaring en de kennis om te kunnen vaststellen dat er sprake is van een geheugenprobleem bij een toepassing en ze weten evenmin wat ze er überhaupt aan kunnen doen. Maar er zijn wel degelijk oplossingen. Het is gewoonweg zaak om te leren waar en waarom de problemen zich kunnen voordoen. Goede tools, die ontwikkelaars ondersteunen bij dit leerproces en ze helpen een goede toepassingsarchitectuur te ontwerpen, zijn onontbeerlijk voor het maken van snelle en betrouwbare .NET-toepassingen.

Werken met de garbage collector

Het mechanisme waarmee de CLR geheugen terughaalt uit toepassingen, wordt de garbage collector genoemd. De garbage collector van .NET begint met roots (basisobjectreferenties die niet in een ander object zijn ingebed) en traceert referenties naar andere objecten op de heap. De garbage collector maakt graphs van alle objecten die vanuit deze rootcomponenten te bereiken zijn. Van objecten die niet in deze graph zijn terug te vinden, wordt aangenomen dat ze niet meer worden gebruikt. Deze objecten worden weer bij de heap gevoegd. De heap wordt doorzocht op objecten die niet in de graphs zijn opgenomen. Vervolgens worden de desbetreffende adressen gemarkeerd en gevolgd tot de volledige heap of een specifiek deel ervan is doorlopen. Om efficiëntieredenen maakt de garbage collector ook gebruik van zogenoemde generations bij het terughalen van geheugen. Er zijn in totaal drie generations: 0, 1 en 2. Wanneer objecten voor het eerst worden toegewezen als een toepassing wordt gestart, markeert de garbage collector dit gedeelte van de heap als generation 0. Nieuw aangemaakte objecten worden altijd toegewezen aan generation 0. De garbage collector controleert deze generation eerst, zodat meer geheugen kan worden teruggehaald en bovendien sneller dan wanneer al het heapgeheugen tegelijkertijd zou worden benaderd.
Als er referenties naar het object zijn wanneer de garbage collector nogmaals wordt uitgevoerd, wordt het object verplaatst naar generation 1. Zodoende hoeft er altijd maar een gedeelte van de geheugenheap te worden gecontroleerd, wat de performance van de garbage collector ten goede komt. Daarnaast wordt een object dat niet door de garbage collector wordt teruggehaald uit generation 1, verplaatst naar generation 2. Wanneer de garbage collector weer wordt uitgevoerd, worden achtereenvolgens de generations 0, 1 en 2 van het heapgeheugen gecontroleerd. Als met het controleren van generation 0 voldoende geheugen kan worden teruggehaald, wordt de uitvoering van de garbage collector gestaakt. Als er meer geheugen moet worden teruggehaald, wordt de garbage collector ook nog uitgevoerd voor generation 1 en tot slot voor generation 2. In de praktijk zijn objecten in generation 2 meestal langdurig van aard. Deze objecten worden dan ook pas teruggehaald wanneer de toepassing wordt afgesloten.

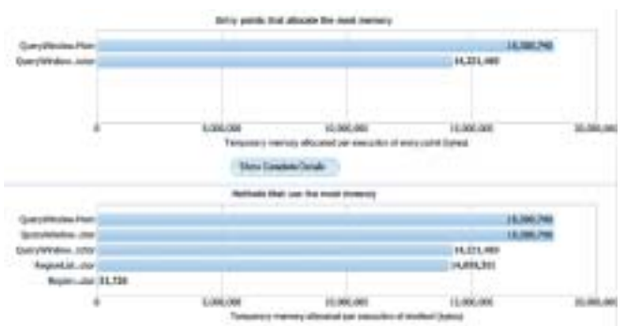
De garbage collector en optimalisatie

De garbage collector zelf vergt veel rekenkracht. Het traceren van objectreferenties en het comprimeren van geheugen kost meer tijd dan het handmatig terughalen van geheugen naar de heap. Dit betekent echter niet dat je voor een optimale performance beter geen gebruik kunt maken van managed code. Er is namelijk wel degelijk sprake van performanceverbeteringen wat geheugentoewijzing betreft. Bovendien is er bij managed code geen sprake van de traditionele geheugenmanagementfouten die bij native code een rol spelen. Deze voordelen zijn reden genoeg om te profiteren van managed toepassingen. Maar het is van essentieel belang dat je goed op de hoogte bent van de werking van de garbage collector als je met .NET Framework werkt en dat je goed weet hoe je geheugenbeheerstrategieën kunt benutten om de performance van jouw toepassingen te verbeteren. De activiteiten van de garbage collector lijken relatief eenvoudig en weldoordacht. Maar als het op de details aankomt, kunnen er verschillende problemen ontstaan. Kleine implementatiewijzigingen in een toepassing kunnen leiden tot aanzienlijke performanceverschillen. Een paar schijnbaar onschadelijke constructs kunnen de toepassing enorm vertragen of leiden tot 'objectlekkage'; waardoor de objecten ook aanwezig blijven als ze niet meer worden gebruikt.
Het maken van te veel objecten is een probleem dat vaak voorkomt. Aangezien de allocatie van nieuw geheugen met het .NET Framework erg snel verloopt, zou je gemakkelijk vergeten dat met slechts één coderegel een groot aantal toewijzingen kan worden gestart. Het probleem treedt op wanneer deze objecten moeten worden opgehaald. De garbage collector beïnvloedt de performance in negatieve zin en het ophalen van een groot aantal onnodige objecten maakt de performance er niet beter op. Dit probleem speelt meestal wanneer met instructies die op basis van code zijn gegenereerd, tijdelijke objecten worden gemaakt om bewerkingen uit te voeren. Er is een groot aantal .NET-klassen dat tijdelijke objecten maakt voor retourwaarden, voor tijdelijke strings en voor bijbehorende klassen zoals enumerators die een noodzakelijk, maar tijdelijk doel hebben. Natuurlijk kan er - ook als de garbage collector wordt gebruikt - nog sprake zijn van objectlekkage. Wanneer er per ongeluk een referentie wordt gemaakt of wanneer deze niet op de

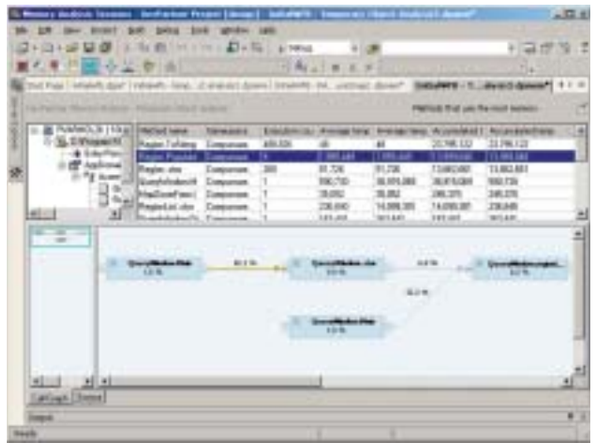
juiste wijze wordt verwijderd, kan er naar een object worden geschreven terwijl de toepassing het object niet meer gebruikt. Als dit object nog op de een of andere manier met een rootstructuur is verbonden, wordt het niet opgehaald door de garbage collector, ondanks het feit dat het object niet meer door de toepassing wordt gebruikt.
Dan zijn er ook nog objecten die onterecht langdurig in stand worden gehouden. Omdat de garbage collector automatisch werkt, zou je bijna vergeten dat het geheugen wordt beheerd volgens vooraf gedefinieerde regels. Als een object lang genoeg wordt gebruikt om naar generation 2 te worden verplaatst, kan het gemakkelijk gebeuren dat het pas wordt opgehaald wanneer de toepassing wordt afgesloten. Hierdoor ontstaan twee problemen. Ten eerste geldt dat hoe meer heapgeheugen er is, hoe meer tijd de garbage collector nodig heeft en hoe trager de toepassing wordt. Ten tweede genereert de toepassing een fout vanwege onvoldoende geheugen als de toepassing lang genoeg wordt uitgevoerd. Al met al betekent dit dat geheugenbeheer in het .NET Framework niet alleen een goed begrip van de toepassing vereist, maar ook van de manier waarop het Framework bewerkingen uitvoert. En zelfs wanneer aan deze voorwaarden is voldaan, is het niet altijd mogelijk om de beste beslissing te nemen. In plaats daarvan behelst het ontwikkelen van toepassingen het voortdurend afwegen van strategieën voor de implementatie van functies en moeten factoren zoals efficiency, implementatiegemak en beheersbaarheid worden overwogen.

Interactieve functie voor real-time geheugenanalyse

Voor een goed inzicht in het geheugen van .NET Framework is een interactieve functie voor real-time geheugenanalyse nodig waarmee afzonderlijke objecten in het geheugen gedurende langere tijd kunnen worden gevolgd. Compuware DevPartner Studio is een dergelijk product, waarmee geheugenanalyse wordt geïntegreerd in het .NET Framework. Met dit product kunnen ontwikkelaars potentiële en daadwerkelijke geheugenproblemen onderzoeken, gedetailleerde informatie vergaren over het gedrag van objecten en de effecten hiervan op het geheugen en strategieën bepalen voor een efficiënt gebruik van het geheugen in managed toepassingen. DevPartner Studio biedt drie fundamentele weergaven van .NET-geheugen: de RAM-footprint, tijdelijke objecten en geheugenlekkages. Door snapshots te maken van al deze weergaven kan je op elk gewenst moment de status van het geheugen vaststellen. Je kunt de garbage collector ook gedwongen laten uitvoeren, zodat je kunt bekijken wat de gevolgen zijn van het terughalen van geheugen en nagaan of een toepassing een objectlekkage heeft.
Door een snapshot te maken van de RAM-footprint zie je wie het geheugen heeft toegewezen, uit welke objecten het bestaat en welke componenten referenties naar het geheugen bevatten (en dus verhinderen dat het geheugen in kwestie wordt vrijgemaakt). Aan de RAM-footprint zie je mogelijk ook dat de toepassing steeds meer geheugen gaat gebruiken naarmate deze langer wordt uitgevoerd. Maar als je alleen maar kijkt naar de hoeveelheid geheugen die in gebruik is, zie je niet welke objecten worden gelekt. Wil je weten om welke objecten het gaat, dan moet je de geheugenanalyse kunnen starten en stoppen om op een willekeurig moment snapshots te maken van de geheugenstatus. Je kunt



Afbeelding 1. Overzichtsweergave van tijdelijke objecten waardoor u in één oogopslag ziet op welke objecten u zich moet richten om het geheugengebruik te beperken.



Afbeelding 2. Gedetailleerde weergave van tijdelijke objecten waardoor u kunt zien hoe objecten en methoden worden aangeroepen en hoe zij op hun beurt andere objecten aanroepen.

een analyse maken van tijdelijke objecten (de tweede fundamentele weergave van .NET-geheugen) om te controleren op ongebruikelijke of inefficiënte zaken, bijvoorbeeld wanneer er grote hoeveelheden tijdelijke objecten worden gemaakt. Met DevPartner Studio breng je nauwkeurig de objecten in kaart die het meeste geheugen toewijzen, samen met de methoden die het meeste geheugen gebruiken; zie afbeelding 1. Verder kun je inzoomen om de methoden te onderzoeken, na te gaan hoe vaak ze worden aangeroepen, wie ze aanroepen, en door wie ze worden aangeroepen; zie afbeelding 2. In een speciale grafiek wordt aangegeven wanneer en waarom de methoden worden aangeroepen.
De laatste belangrijke weergave van .NET-geheugen is de weergave van geheugenlekkages. Objecten kunnen geheugen lekken doordat referenties niet direct (of helemaal niet) worden vrijgegeven. Dit kan leiden tot een slechte performance en zelfs tot fouten in toepassingen. DevPartner Studio volgt de geheugentoewijzingen van objecten om te bepalen welke objecten het geheugen niet na verloop van tijd vrijgeven.

Geheugenanalyse

Overstappen op .NET betekent niet dat alle problemen met geheugenbeheer in één klap zijn opgelost. De meeste ontwikkelaars zijn immers niet bekend met de typische problemen die nog wel optreden, waardoor het ontwikkelen van toepassingen met .NET moeilijker en foutgevoeliger wordt dan het geheugenbeheermodel doet vermoeden. Zolang ontwikkelaars de principes van .NET-geheugenbeheer niet optimaal kunnen toepassen, is er bij toepassingen een grotere kans op een slechte performance, een beperkte schaalbaarheid en geheugenfouten. De geheugenanalyse van DevPartner Studio is een van de meest uitgebreide manieren om .NET-geheugen te analyseren. Dankzij de verschillende weergaven van geheugengebruik op overzichtsniveau, de mogelijkheid om dynamische veranderingen in het geheugen te volgen, de mogelijkheid om in te zoomen op meer gedetailleerde weergaven van afzonderlijke objecten en de beschikbaarheid van aanroepgraphs om de relaties tussen objecten te analyseren is DevPartner Studio een essentieel programma voor het ontwikkelen van snelle en betrouwbare .NET-toepassingen.

Nuttige internetadressen

http://www.compuware.com/solutions/microsoft.htm
http://www.compuware.com/products/devpartner/enterprise.htm
http://www.vsj.co.uk/dotnet/display.asp?id=284

Peter Varhol is werkzaam bij Compuware en is lid van de senior development staff en is daarnaast een gerenommeerd schrijver en columnist van technische artikelen over software-development.