**35. Стратегии на управление на памет и събиране на ‘боклук’ в .NET среда. Алгоритъм за “ събиране**

**на боклук**”

**Как работи garbage collector?**

Вече беше споменато, че ако добавянето на нов обект би довело до препълване на хийпа, трябва да се

осъществи почистване на паметта. В този момент, CLR стартира системата за почистване на паметта, т.нар. garbage collector. **Всъщност това е опростено обяснение. Garbage collector се**

**стартира когато Поколение 0 се запълни. Поколенията се разглеждат в следващата секция.**

Първото нещо, което трябва да се направи, за да може системата за почистване на паметта да започне работа, това е да се приспят всички нишки на приложението, изпълняващи управляван код. по време на събирането на отпадъци е твърде вероятно обектите да се преместят на нови адреси в динамичната памет, нишките не трябва да могат да достъпват и модифицират обекти докато трае почистването. CLR изчаква всички нишки да достигнат в безопасно състояние, след което ги приспива. Съществуват няколко механизма, чрез които CLR може да приспи дадена нишка. Причината за тези различни механизми е

стремежът да се намали колкото се може повече натоварването и нишките да останат активни възможно най-дълго.

**Освобождаване на неизползваните обекти**

След като всички управлявани нишки на приложението са безопасно "приспани", garbage collector проверява дали в managed heap има обекти, които вече не се използват от приложението. Ако такива обекти съществуват, заетата от тях памет се освобождава. След приключване на

работата по събиране на отпадъци се възобновява работата на всички

нишки и приложението продължава своето изпълнение. Както вероятно се досещате, откриването на ненужните обекти и освобождаването на ресурсите, заети от тях, не е проста задача. В тази

секция накратко ще опишем алгоритъмът, който .NET garbage collector използва за нейното решаване.

За да установи кои обекти подлежат на унищожение, garbage collector построява граф на всички обекти, достъпни от нишките на приложението в дадения момент. Всички обекти от динамичната памет, които не са част от графа се считат за отпадъци и подлежат на унищожаване. Възниква въпросът как garbage collector може да знае кои обекти са достъпни и кои не? **Корените на приложението** са точката, от която

системата за почистване на паметта започва своята работа.

**Корени на приложението**

Всяко приложение има набор от корени (**application roots**). Корените представляват области от паметта, които сочат към обекти от managed heap, или са установени на **null**. Например всички глобални и статични

променливи, съдържащи референции към обекти се считат за корени на приложението. Всички локални променливи или параметри в стека към момента, в който се изпълнява garbage collector, които сочат към обекти, също принадлежат към корените. Регистрите на процесора, съдържащи указатели към обекти, също са част от корените. Към корените на приложението спада и Freachable queue (за Freachable queue по-подробно ще стане дума в секцията за финализация на обекти в настоящата глава.

Засега просто приемете че тази опашка е част от вътрешните структури, поддържани от CLR и се счита за един от корените на приложението). Когато JIT компилаторът компилира IL инструкциите на даден метод в

процесорни инструкции, той също съставя и вътрешна таблица, съдър- жаща корените за съответния метод. Тази таблица е достъпна за garbage collector. Ако се случи garbage collector да започне работа, когато методът се изпълнява, той ще използва тази таблица, за да определи кои са

корените на приложението към този момент. Освен това се обхожда и стекът на извикванията за съответната нишка и се определят корените зa всички извикващи методи (като се използват техните вътрешни таблици). Към получения набор от корени, естествено, се включват и тези, намира-

щи се в глобални и статични променливи.Трябва да се помни, че не е задължително даден обект да излезе от обхват за да бъде считан за отпадък. JIT компилаторът може да определи\_\_

кога този обект се достъпва от кода за последен път и веднага след това

го изключва от вътрешната таблица на корените, с което той става

кандидат за почистване от garbage collector. Изключение правят случаите,

когато кодът е компилиран с **/debug** опция, която предотвратява почист-

ването на обекти, които са в обхват. Това се прави за улеснение на

процеса на дебъгване – все пак при трасиране на кода бихме искали да

можем да следим състоянието на всички обекти, които са в обхват в

дадения момент.

**Алгоритъмът за почистване на паметта**

Когато garbage collector започва своята работа, той предполага че всички

обекти в managed heap са отпадъци, т.е. че никой от корените не сочи

към обект от паметта. След това, системата за почистване на паметта

започва да обхожда корените на приложението и да строи граф на

обектите, достъпни от тях.

Нека разгледаме примера, показан на следващата фигура. Ако глобална

променлива сочи към обект A от managed heap, то A ще се добави към

графа. Ако A съдържа указател към C, а той от своя страна към обектите

D и F, всички те също стават част от графа. Така garbage collector обхожда

рекурсивно в дълбочина всички обекти, достъпни от глобалната промен-

лива A:

Когато приключи с построяването на този клон от графа, garbage collector

преминава към следващия корен и обхожда всички достъпни от него

обекти. В нашия случай към графа ще бъде добавен обект E. Ако по време

на работата garbage collector се опита да добави към графа обект, който

вече е бил добавен, той спира обхождането на тази част от клона. Това се

прави с две цели:

- значително се увеличава производителността, тъй като не се преми-

нава през даден набор от обекти повече от веднъж;

- предотвратява се попадането в безкраен цикъл, ако съществуват

циклично свързани обекти (например A сочи към B, B към C, C към D

и D обратно към A).

След обхождането на всички корени на приложението, Графът съдържа

всички обекти, които по някакъв начин са достъпни от приложението. В

посочения на фигурата пример, това са обектите A, C, D, E и F.

Всички обекти, които не са част от този граф, не са достъпни и следова-

телно се считат за отпадъци. В нашия пример това са обектите B, G, H и I.

След идентифицирането на достъпните от приложението обекти, garbage

collector преминава през хийпа, търсейки последователни блокове от

отпадъци, които вече се смятат за свободно пространство. Когато такава

област се намери, всички обекти, намиращи се над нея се придвижват

надолу в паметта, като се използва стандартната функция **memcpy(…)**.

Крайният резултат е, че всички обекти, оцелели при преминаването на

garbage collector, се разполагат в долната част на хийпа, а **NextObjPtr** се

установява непосредствено след последния обект. Фигурата показва със-

тоянието на динамичната памет след приключване на работата на garbage

collector.