עוד על איסוף זבל ב-.NET

נכתב ע"י סשה גולדשטיין

הקדמה

במאמר הקודם על איסוף זבל שהתפרסם בגיליון חודש אוגוסט, עברנו על המרכיבים העיקריים של אוסף הזבל ב-.NET, לרבות: מבנה הערימה המנוהלת, הקצאת זיכרון, הקשר בין אוסף הזבל לבין מערכת ההפעלה, פרגמנטציה של הזיכרון, דורות, וניהול אובייקטים גדולים. במאמר זה נמשיך לסקור את מנגנון איסוף הזבל, ונתבונן במספר נושאים מתקדמים ומורכבים יותר המשפיעים על הביצועים של יישומי .NET ומאפשרים לשנות את התנהגותם.

כדי להפיק את המירב ממאמר זה, כדאי לקרוא קודם את המאמר הקודם כדי לקבל רקע כללי על איסוף הזבל ב-.NET. ועל המנגנונים שנדונים במאמר הנוכחי.

ניהול אובייקטים גדולים

כפי שראינו במאמר הקודם, .NET מנהלת שלושה דורות של אובייקטים הממוקמים בשלושה אזורים נפרדים של הערימה המנוהלת. אובייקטים חדשים נוצרים בדור 0, ומועברים לדורות ותיקים יותר כאשר הם שורדים סיבוב אחד לפחות של איסוף זבל. כיוון שדור 0 נשמר באזור זיכרון קטן יחסית (בר-השוואה לגודל זיכרון המטמון של המעבד, כלומר 1-8MB), לא ברור מה עולה בגורלם של אובייקטים גדולים יותר.

יתר על כן, כאשר אובייקטים עוברים בין דורות, וכאשר אוסף הזבל מאחה את הערימה על ידי קיבוץ אובייקטים חיים יחד, יש צורך בהעתקת הזיכרון של האובייקט. למנגנון זה שתי עלויות עיקריות: ראשית, עלות ההעתקה עצמה, שעבור אובייקטים שאינם נכנסים בזיכרון המטמון יכולה להיות משמעותית מאוד (רוחב הפס לזיכרון במחשבים מודרניים נמדד במספר חד-ספרתי של ג'יגה-בייט לשנייה בלבד); שנית, במהלך ההעתקה יש צורך בעצירת חוטים אחרים של התוכנית, כיוון שלא ניתן לספוג את האפשרות שבה חוט אחר של התוכנית ישנה את הזיכרון של אובייקט שכרגע מועתק על ידי אוסף הזבל.

לפיכך, אובייקטים גדולים מנוהלים באזור נפרד של הערימה המנוהלת, המכונה ערימת האובייקטים הגדולים (Large Object Heap). אוסף הזבל של .NET. לעולם אינו מבצע העתקה ואיחוי מחדש של אזורי זיכרון המאוכלסים באובייקטים גדולים. הדבר משפיע גם על אלגוריתם הקצאת הזיכרון עבור אובייקטים גדולים: לא זו בלבד שהם מוקצים באזור זיכרון נפרד, אלא שכדי להקצות את האובייקט יש לחפש בלוקים

פנויים בערימת האובייקטים הגדולים. בגרסת NET. הנוכחית, הבלוקים הפנויים מנוהלים במספר רשימות מקושרות, בדומה למנגנוני הקצאת הזיכרון של הספרייה הסטנדרטית של שפת C ושל ה- Low Fragmentation Heap במערכת ההפעלה.

אם כן, לצד היתרונות הברורים שבהמנעות מהעתקה ואיחוי מחדש של אובייקטים גדולים, ישנם גם מספר חסרונות משמעותיים. החיסרון הראשון נובע מכך שהקצאה ושחרור תכופים של אובייקטים גדולים עשויים לגרום לפרגמנטציה של ערימת האובייקטים הגדולים, תוך בזבוז של מקום רב בבלוקים פנויים. אולם החיסרון המרכזי קשור במבנה הדורות של אוסף הזבל. אובייקטים גדולים לא משויכים לדורות 0 ו-1, אלא נכנסים עם יצירתם לדור 2, השמור לאובייקטים ותיקים. לכן, אוסף הזבל בוחן את האובייקטים הגדולים רק כאשר מתבצע איסוף זבל מלא (של דור 2), או כאשר חסר מקום בערימת האובייקטים הגדולים. בשני המקרים, מדובר על תהליך יקר שאינו מביא לידי ביטוי את משך החיים של האובייקטים, שהוא היתרון המרכזי של מודל הדורות.

לפיכך, תוכנית שמבצעת הקצאות רבות של אובייקטים גדולים וזמניים, עשויה למצוא את עצמה מבלה זמן רב באיסוף זבל מלא (של דור 2 וערימת האובייקטים הגדולים). תהליכים אלה עשויים לגזול עשרות אחוזים מזמן הריצה של התוכנית - במקרים חריגים, ראינו מערכות שמבלות 70% מזמן הריצה שלהם באיסוף זבל. תוצאות כאלה אמנם חריגות, אך מסוכנות ביותר והן אחת הסיבות למוניטין הרע של איסוף הזבל בשפות מנוהלות.

כדי להימנע מהעלות הגדולה של איסוף זבל מלא הקשור באובייקטים גדולים וזמניים, לעתים קרובות יש טעם למחזר אובייקטים גדולים במקום להקצות אותם מחדש. למשל, תשתית תקשורת המנהלת אוסף גדול של מערכים השומרים הודעות נכנסות ויוצאות, תנהל לעתים קרובות בריכה (Pool) של מערכים בגדלים טיפוסיים, ותמחזר את הקצאות הזיכרון כדי להימנע מעלות איסוף הזבל. אם זה נשמע לכם מעוות, ומושך את השטיח מתחת לרגליהן של הסביבות המנוהלות, אתם צודקים - אך זהו המחיר של דרישות ביצועים גבוהות עם נוחות פיתוח הגבוהה לאין שיעור בסביבה בעלת איסוף זבל.

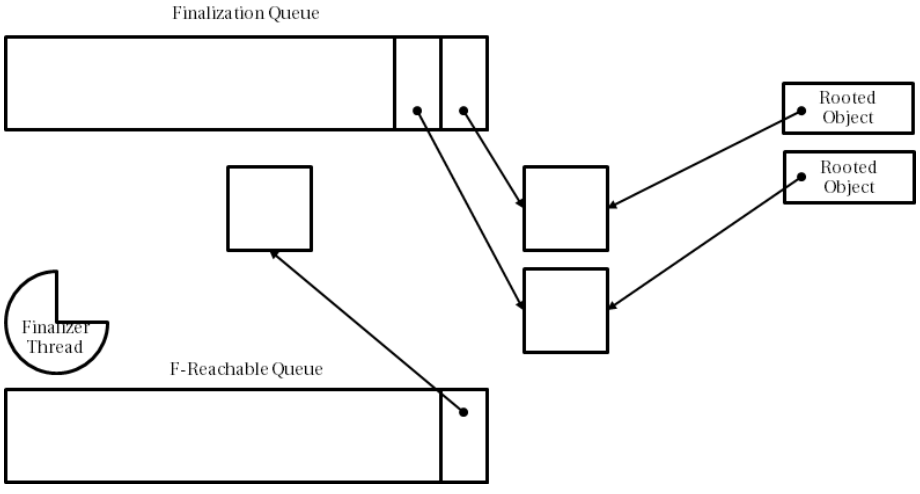
Finalization

בשפות לא-מנוהלות (כמו C++) ניתן לשייך לכל אובייקט מתודה שתופעל כאשר האובייקט מושמד (destructor, או "הורס"). כיוון שזמן השמדת האובייקט הוא דטרמיניסטי בשפות אלה, גם הנקודה בזמן שבה מופעל ההורס היא דטרמיניסטית. לעומת זאת, ב-NET. זמן השמדת האובייקט מוכתב על ידי אוסף הזבל, ואינו ניתן לצפייה מראש (בקלות) על ידי המפתח. לכן גם היכולת להריץ קוד באופן אוטומטי כאשר האובייקט מושמד כפופה לאי-דטרמיניזם זה.

גם אובייקטים ב-.NET. יכולים לבקש הרצה של מתודה כאשר הם עומדים להימחק. מתודה זו נקראת finalizer, ואוסף הזבל מבטיח שהיא תופעל לאחר שהתוכנית אינה משתמשת יותר באובייקט, אך מצד שני לפני שהזיכרון שלו משתחרר. זוהי ההזדמנות האחרונה עבור האובייקט לפנות משאבים לא-מנוהלים: חיבור למסד נתונים, קובץ פתוח, קישור רשת, ועוד.

כאשר אובייקט בעל finalizer נוצר, הוא מוכנס לתור מיוחד המכונה Finalization Queue. תור זה הכרחי כדי שאוסף הזבל לא "יאבד" את הקישור האחרון לאובייקט גם אם התוכנית אינה משתמשת בו יותר. כל עוד התוכנית משתמשת באובייקט, אוסף הזבל לא מתייחס אליו באופן מיוחד; אך כאשר הקישור האחרון מהתוכנית נעלם, אוסף הזבל יכול לזהות שהאובייקט זקוק להרצת finalizer מכיוון שהוא נמצא בתור הנ"ל. בשלב זה, אוסף הזבל מעביר את האובייקט לתור נוסף, ה-F-Reachable Queue. תור זה מכיל אובייקטים שמחכים להרצת ה-finalizer-ים שלהם, ושאין סיבה אחרת להחזקתם בחיים.

אלא שכעת יש צורך בגורמים נוספים מלבד החוטים של אוסף הזבל עצמו. הסיבה לכך פשוטה: החוטים של אוסף הזבל עסוקים מספיק במעבר על הזיכרון ואיחויו, ואין אפשרות לעכבם עוד כדי להריץ מתודות "ניקיון" שסופקו על ידי אובייקטים שונים. אך גם החוטים של התוכנית אינם יכולים להריץ מתודות אלה - תהליך איסוף הזבל הוא אסינכרוני ואינו קשור כמעט בכלל לקוד שמריצים חוטי התוכנית. מכאן נובעת הדרישה לחוט נוסף, המוקדש להרצת finalizer-ים - חוט שנקרא ה-Finalizer Thread. חוט זה מוציא אובייקטים מה-F-Reachable Queue, מפעיל את ה-finalizer-ים שלהם זה אחר זה, ולאחר מכן מנתק את הקשר עם האובייקטים כדי שיוכלו להתפנות בסבב איסוף הזבל הבא.



[הנפשות הפועלות בתהליך ה-Finalization]

העלות של המנגנון הזה גדולה, כיוון שאובייקטים שורדים זמן רב יותר. אובייקט שמבקש finalization ישרוד לפחות דור אחד נוסף בגלל הצורך להעבירו בין תורים ולהמתין לעבודה האסינכרונית של ה- Finalizer Thread.

זאת ועוד, עבודתו האסינכרונית של חוט זה מובילה לבעיות קצב ונכונות רבות, שביניהן:

- אם התוכנית מקצה אובייקטים בעלי finalizer בקצב גבוה יותר מקצב הרצת ה-finalizer-ים שלהם, נוצרת דליפת זיכרון ב-F-Reachable Queue. הדבר אפשרי בקלות כיוון שיש רק חוט אחד האחראי על הרצת finalizer-ים, לעומת חוטים רבים של התוכנית שיכולים להקצות זיכרון במקביל.
- אם finalizer של אובייקט מסוים נתקע לזמן ממושך (למשל, בגלל המתנה לפעולת רשת כגון סגירת חיבור למסד נתונים), הדבר מעכב הרצה של finalizer-ים עבור אובייקטים נוספים, ועלול לגרום למחסור במשאבים לא-מנוהלים עבור שאר התוכנית.
- כיוון שעבודת הניקיון מתבצעת בחוט נפרד, בצורה אסינכרונית, ייתכנו בעיות רגילות של תכנות מרובה-חוטים: חבק, גישה לא-מאובטחת לזיכרון משותף, ועוד. למעשה, אחת הבעיות העדינות ביותר שנתקלתי בהן בהקשר זה הייתה סיטואציה שבה ה-finalizer של אובייקט התבצע במקביל למתודה אחרת שלו!

אך מהי האלטרנטיבה למנגנון זה? ניקיון אוטומטי של משאבים לא יכול להיות ממומש בצורה אחרת, מסיבות שתוארו קודם. הצורך בחוט נפרד ובעיות הקצב הנלוות לו הם בלתי נמנעים אם דורשים שהמנגנון יפעל באופן אוטומטי. לחלופין, ניתן לממש באופן ידני מנגנון של ניקוי משאבים דטרמיניסטי, המבוסס על קריאה למתודה מסוימת של האובייקט. יש לכך אפילו תמיכה בתחביר השפות עצמן (C#, VB.NET ואחרות), אך הדבר בהחלט מורכב יותר ועשוי לגרום לדליפת משאבים אם המפתח לא מקפיד להפעיל ניקיון ידני זה.

```
//Deterministic (but manual) cleanup of resources - C++ style in C#
public class FileWithManualCleanup
{
    public FileWithManualCleanup(string filename, …) …
    public void Cleanup() …
}
```

יחסים בין דורות

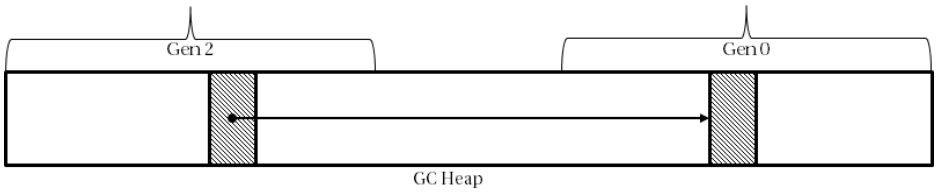
כזכור מהמאמר הקודם, מנגנון הדורות מאפשר ביצוע של איסוף זבל חלקי ויעיל במיוחד, למשל כאשר בוחנים רק אובייקטים בדור 0. מחד, אלה אובייקטים חדשים שאמורים למות מהר, ולכן איסוף הזבל הוא יעיל (מפנה אחוז ניכר מהאובייקטים הנבחנים), ומאידך דור 0 הוא אזור זיכרון קטן יחסית, שאיסוף הזבל בו לא נמשך זמן רב.

אלא שתוך כדי התיאור התעלמנו מבעיה נוקבת: כיצד אפשר לבצע איסוף זבל חלקי הבוחן רק אובייקטים של דור 0? ליתר דיוק, כיצד ניתן להתאים את מנגנון הסימון (Mark) כדי שיעבור רק אובייקטים בדור 0 ולא על אובייקטים בדורות גבוהים יותר?

הפתרון הנאיבי הוא להגביל את אלגוריתם הסימון הרקורסיבי לאובייקטים בדור 0 בלבד. כלומר, החיפוש הרקורסיבי מפסיק כאשר מגיעים לאובייקט שאינו בדור 0:

```
//Naïve - and WRONG - attempt to mark gen 0 objects only
Mark(obj):
  if gen(obj) <> 0 then return
  if visited(obj) then return
  mark obj as visited
  for each reference field of obj do
    Mark(field)
  end for
```

מדוע רעיון זה אינו עובד? כיוון שיתכנו אובייקטים בדור 2 המצביעים לאובייקטים בדור 0. אם זה המצב, לא נשים לב שהאובייקט בדור 0 נמצא בשימוש, ונמצא את עצמנו משחררים זיכרון שנמצא בשימוש.



אלא שהברירה נראית גרועה עוד יותר: לכאורה, הבעיה שגילינו כעת מחייבת מעבר מלא על כל הערימה כדי לאתר אחוז זעום של אובייקטים בדור 0 שעשויים עדיין להיות בחיים. פתרון זה יאט משמעותית את איסוף הזבל בדור 0, ויעלים לגמרי את התועלת בהפרדת האובייקטים לדורות בערימה.

כדי להתמודד עם אתגר זה, דרוש שיתוף פעולה של המהדר, ובמקרה של .NET. זהו ה- Just-In-Time Compiler המתרגם את שפת הביניים של .NET (IL) לפקודות מכונה בהתאם לארכיטקטורת המעבד עליה התוכנית רצה. הצבעה מאובייקט בדור 1 או 2 לאובייקט בדור 0 יכולה להיווצר בסוג מאוד מסוים של פקודות: השמה של מצביע לאיבר במערך או לשדה של אובייקט, כגון: a[i] = b או a.f = b - ומצבים אלה ניתנים לאיתור וטיפול על ידי המהדר.

באופן מדויק יותר, כאשר המהדר יוצר את פקודות המכונה עבור הוראות שעשויות ליצור הצבעה מדור גבוה לדור נמוך יותר, הוא מוודא שבזמן ריצה יעודכן מבנה נתונים השומר מידע על הצבעות כאלה. בהמשך, אוסף הזבל יוכל להיעזר במבנה זה כדי לאתר אובייקטים בדור 0 שצריכים להישאר בחיים כיוון שיש מצביע אליהם מדור גבוה יותר.

למרות שהמנגנון נשמע יקר, בפועל מדובר במספר פקודות מכונה בודדות:

```
; Compilation output for a.f = b, where a is in eax, b is in ecx,
; and f is at offset 0x12 from the beginning of a
cmp eax, dword ptr [GEN0_END]
jb write_through
cmp ecx, dword ptr [GEN0_END]
jg write_through
mov edx, eax
shr eax, 8
mov dword ptr [TABLE_START+edx], 1
write_through: mov dword ptr [eax+12], ecx
```

המשמעות היא שכאשר נוצר מצביע מדור 1 או 2 לדור 0, הכניסה בטבלה המתאימה ל-1024 בתים סביב הכתובת של האובייקט הוותיק מעודכנת בהתאם. כעת כאשר אוסף הזבל יבצע איסוף של דור 0, הוא יכול להישאר עם האלגוריתם הנאיבי שהוצג קודם לסימון, אלא שבנוסף עליו לבחון את הכניסות בטבלה:

```
for i = 0 to tablesize do
  if table[i] == 1 then
    p = start of 1KB range covered by table[i]
    j = 0
    while j < 1024 do
      //Assume the 1KB block always starts with an object
      //and is not in the middle of a previous object
      Mark(object at p+j)
      j += size of object at p+j
    end while
  end if
end for
```

יש לשים לב שהאלגוריתם הנ"ל שמרני מאוד במספר מובנים. ראשית, הוא מניח שכל האובייקטים בטווח של 1KB סביב האובייקט הוותיק דורשים סימון כאשר מבצעים איסוף של דור 0, בעוד שייתכן שרק אובייקט אחד מתוך הטווח באמת מחזיק מצביע לאובייקטים בדור 0. שנית, המידע בטבלה אינו מתעדכן כאשר ההצבעה מוסרת. בכל זאת, רעיון פשוט זה מספק פתרון לאתגר הסימון החלקי בעלות זיכרון נמוכה יחסית (בית אחד לכל 1024 בתים של זיכרון בדורות הגבוהים), והוא נמצא בשימוש גם ב-NET. וגם בסביבות מנוהלות אחרות, כגון JVM.

סיכום

במאמר זה סקרנו מספר נושאים מתקדמים שאוסף הזבל מטפל בהם על מנת להבטיח ביצועים גבוהים לתוכניות ב-.NET. ולשמור על נכונות מודל הדורות ושחרור המשאבים הלא-מנוהלים. אוסף הזבל הוא בהחלט המרכיב המסובך ביותר בסביבות מנוהלות כגון .NET ו-Java, והנושא נמצא עדיין במחקר אקדמי מקיף. למשל, תחומי פעילות אקדמיים פורים הם סביב צמצום עצירת חוטי התוכנית בזמן ביצוע איסוף זבל, מקבול איסוף הזבל במערכות בעלות מאות מעבדים, איסוף זבל חלקי וספקולטיבי, ועוד רבים אחרים.

למרות שקראתם כבר שני מאמרים של איסוף זבל ב-.NET., עדיין יש נושאים רבים שלא נגענו בהם. בספרי החדש Pro .NET Performance (שיצא לאור בחודש ספטמבר) יש כ-60 עמודים המוקדשים בלעדית לנושא איסוף הזבל. יש באפשרותי לספק מספר מצומצם של עותקים אלקטרוניים בחינם לקוראים שמעוניינים לכתוב ביקורת על הספר בבלוג שלהם או באתר של Amazon. לפרטים, תוכלו ליצור איתי קשר בכתובת המייל שלי: sashag@sela.co.il.

על המחבר

סשה גולדשטיין הוא ה-CTO של קבוצת סלע, חברת ייעוץ, הדרכה ומיקור חוץ בינלאומית עם מטה בישראל. סשה אוהב לנבור בקרביים של Windows וה-CLR, ומתמחה בניפוי שגיאות ומערכות בעלות ביצועים גבוהים. סשה הוא מחבר הספר Pro .NET Performance, ובין היתר מלמד במכללת סלע קורסים בנושא Windows Internals ו-CLR Internals. בזמנו הפנוי, סשה כותב בלוג על נושאי פיתוח שונים.