СТАТТІ · 30 листопада 2018, 13:00 45805 Игорь Пикуль, Software Architect в GR8 Tech

Принципы работы Garbage collection

В этой статье вспомним, что такое Garbage collection (GC), зачем он нужен вообще и какие проблемы решает. Детально рассмотрим режимы работы GC в .NET, поймем, как работает каждый из них, их особенности и различия. Затронем специфику применения некоторых режимов GC в .NET.

Изучим вопрос мониторинга работы GC, какие доступны для этого инструменты и как ими пользоваться.

Введение

Вообще, откуда взялась эта тема? Она появилась из-за поведения наших сервисов, в том числе и на production. Мы увидели, что некоторые приложения начали отнимать 30% CPU. Не могли понять, почему это происходит — ведь по коду все было хорошо. Провели анализ метрик, о которых поговорим позже, и выяснили, что GC потребляет на сборку мусора порядка 30%. И тут возник вопрос — что же с этим делать. Появилось поле для оптимизации. И мы добились хороших результатов, когда после всевозможных манипуляций снизили потребление CPU до 10%, до 5%. Как этого можно добиться, я расскажу ниже.

Когда я задался вопросом и начал готовить эту статью, мне было интересно, а когда у нас появился первый язык, который уже поддерживал сборку мусора. Я даже немного удивился, потому что это был 1964 год. 50 лет назад люди уже задумывались о том, что разработчиков нужно освобождать от занятий с памятью. Это был язык APL. Из языков, которые поддерживают сборку мусора, можно назвать Erlang (1990 год), Eifel, Smalltalk (1972 год), конечно же, C# и любой современный язык, который выходит сейчас, например Go. Это уже must have.

Интересный факт: по исследованиям, разработчики, которые занимаются написанием кода на языках, не поддерживающих сборку мусора, 40% своего продуктивного времени тратят на операции по работе с управлением памятью, что довольно много и, скорее всего, не всегда будет понятно менеджменту.

Что такое Garbage Collection

GC (Garbage Collection — сборка мусора) — высокоуровневая абстракция, которая избавляет разработчиков от необходимости заботиться об освобождении управляемой памяти.

Давайте вспомним основные тезисы по сборке мусора. В .NET сборка мусора основана на трассировке.

Существует понятие корневых элементов приложения. *Корневым элементом* (root) называется ячейка в памяти, в которой содержится ссылка на размещаемый в куче объект. Строго говоря, корневыми могут называться такие элементы:

— Ссылки на глобальные объекты (хотя в C# они не разрешены, но CIL-код позволяет размещать глобальные объекты).
— Ссылки на любые статические объекты или статические поля.
— Ссылки на локальные объекты в пределах кодовой базы приложения.
— Ссылки на передаваемые методу параметры объекта.
— Ссылки на объект, ожидающий финализации.
— Любые регистры центрального процессора, которые ссылаются на объект.

Во время процесса сборки мусора исполняющая среда будет исследовать объекты в куче, чтобы определить, являются ли они по-прежнему достижимыми (т. е. корневыми) для приложения. Для этого среда CLR будет создавать *графы объектов*, представляющие все достижимые для приложения объекты. Кроме того, следует иметь в виду, что сборщик мусора никогда не будет создавать граф для одного и того же объекта дважды, избавляя от необходимости выполнения подсчета циклических ссылок, который характерен для программирования в среде COM.

Фазы сборки мусора:

1. Маркировка (mark phase).
2. Чистка (sweep phase).
3. Сжатие (compact phase).

Поколения объектов: нулевое, первое, второе поколение.

Нулевое и первое поколения еще называют эфемерными поколениями. Они нужны для ускорения отклика нашего приложения.

Для работы приложения CLR инициализирует 2 сегмента виртуального адресного пространства — Small object heap (объекты до 85 КБ) и Large object heap (объекты свыше 85 КБ, в некоторых случаях массивы и связанные списки (linked list), не достигшие данного размера).

Конфигурирование GC довольно простое, что отображено на следующем рисунке:

```
<configuration>
  <runtime>
    <gcConcurrent enabled="truelfalse"/>
<gcServer enabled="truelfalse"/>
  </runtime>
</configuration>
```

Рисунок 1. App.config

Конфигурировать режимы работы GC можно путем добавления в app.config секции, показанной на слайде выше, с помощью параметров gcConcurrent, gcServer.

Режим рабочей станции

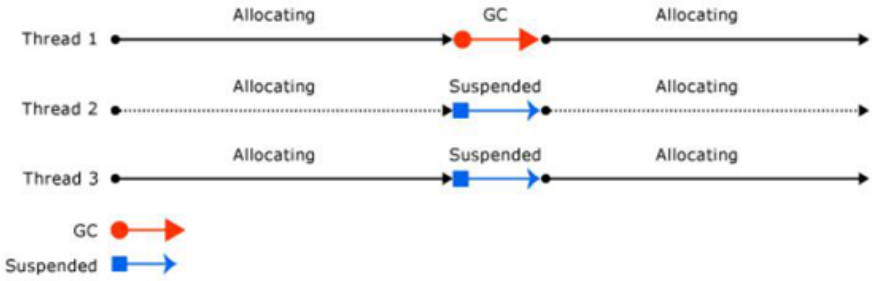

Рисунок 2. Процесс сборки мусора в режиме рабочей станции

Если мы откроем любую книгу по .NET, любую статью по .NET, где у нас описано, как работает Garbage Collection, обычно это звучит так: работает приложение, не хватает памяти для того, чтобы выделить следующий объект, и происходит запуск GC. При этом все активные потоки приложения приостанавливаются. Это самый простой процесс сборки мусора — workstation non-concurrent mode.

Недостатком этого подхода является то, что во время сборки мусора приложение не занимается ничем другим, кроме сборки мусора. Можно ли этого избежать и как-то повысить отклик нашего приложения?

Идея, как повысить производительность приложения, довольно проста: если нулевое и первое поколения собираются очень быстро, то почему бы их не очищать отдельно от второго поколения. Возможно ли так сделать, чтобы при сборке второго поколения, наше приложение и дальше продолжало аллоцировать объекты? Да, возможно.

Параллельная сборка мусора

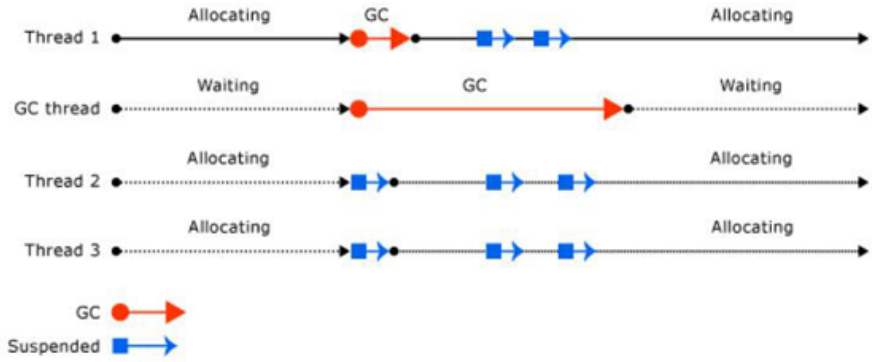

Рисунок 3. Параллельная сборка мусора

Для этого существует режим параллельной сборки мусора (workstation concurrent GC).

Параллельная сборка мусора в .NET 1.0–3.5

До выхода .NET 4.0 очистка неиспользуемых объектов проводилась с применением техники *параллельной сборки мусора*. В этой модели, при выполнении сбора мусора эфемерных объектов, сборщик мусора временно приостанавливал все активные *потоки* внутри текущего процесса, чтобы

приложение не могло получить доступ к управляемой куче вплоть до завершения процесса сборки мусора.

По завершении цикла сборки мусора приостановленным потокам разрешалось снова продолжить работу. К счастью, в .NET 3.5 сборщик мусора был хорошо оптимизирован, и потому связанные с ним короткие перерывы в работе с приложением редко становились заметными.

Как и оптимизация, параллельная сборка мусора позволяла проводить очистку объектов, которые не были обнаружены ни в одном из эфемерных поколений, в отдельном потоке. Это сокращало (но не устраняло) необходимость в приостановке активных потоков исполняющей средой .NET. Тем более, параллельная сборка мусора позволяла размещать объекты в куче во время сборки объектов неэфемерных поколений.

Фоновая сборка мусора

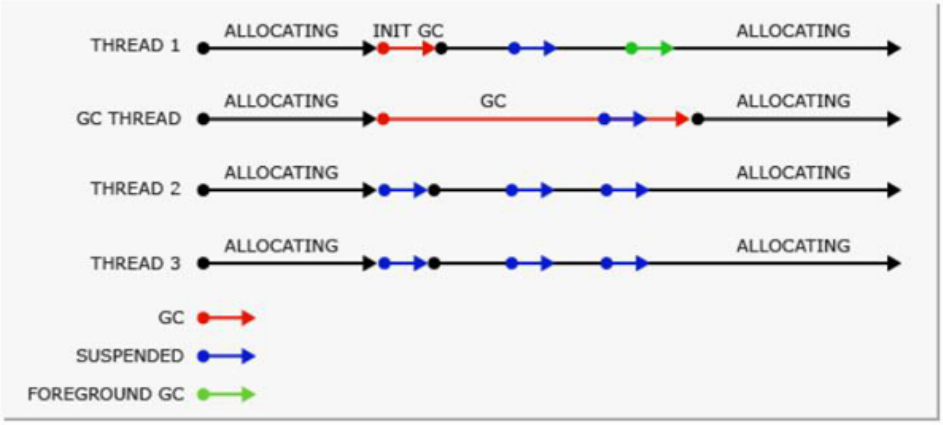

Рисунок 4. Фоновая сборка мусора

В .NET 4.0 сборщик мусора по-другому решает вопрос о приостановке потоков и очистке объектов в управляемой куче, используя при этом технику *фоновой сборки мусора*. Несмотря на ее название, это вовсе не означает, что вся сборка мусора теперь происходит в дополнительных фоновых потоках выполнения. На самом деле, в случае фоновой сборки мусора для объектов, не относящихся к эфемерному поколению, исполняющая среда .NET теперь может проводить сборку мусора объектов эфемерного поколения в отдельном фоновом потоке.

Механизм сборки мусора в .NET 4.0 был улучшен так, чтобы на приостановку потока, связанного с деталями сбора мусора, требовалось меньше времени. Благодаря этим изменениям процесс очистки неиспользуемых объектов поколения 0 и 1 стал оптимальным. Он позволяет получать более высокий уровень производительности приложений.

Давайте представим, что у нас на хосте, где наше приложение запустится, есть один процессор. В таком случае, что бы мы ни делали, мы все равно запустимся в режиме рабочей станции. Вы можете делать с флажками что угодно, но на одном процессоре не хватит одновременных потоков, которые могут запуститься для того, чтобы обслуживать другой режим.

Режим сервера

Особенности работы GC в режиме сервера.

- Сборка выполняется в нескольких выделенных потоках, выполняемых с приоритетом THREAD_PRIORITY_HIGHEST .
- Для каждого процессора предоставляется куча и выделенный поток, выполняющий сборку мусора, и сборка куч выполняется одновременно. Каждая куча содержит кучу небольших объектов и кучу больших объектов, и все кучи доступны из пользовательского кода. Объекты из различных куч могут ссылаться друг на друга.
- Так как несколько потоков сборки мусора работают совместно, для кучи одного и того же размера сборка мусора сервера выполняется быстрее сборки мусора рабочей станции.
- В сборке мусора сервера часто используются сегменты большего размера. Однако обратите внимание, что это только обобщение: размер сегмента зависит от реализации и может изменяться. При настройке приложения не следует делать никаких предположений относительно размера сегментов, выделенных сборщиком мусора.
- Сборка мусора сервера может оказаться ресурсоемкой операцией. Например, если на компьютере с 4 процессорами выполняется 12 процессов, в каждом из которых применяется сборка мусора сервера, будут использоваться 48 выделенных потоков сборки мусора. В случае высокой загрузки памяти, если все процессы запускают сборку мусора, сборщику мусора понадобится выполнить планирование работы 48 потоков.

При запуске сотен экземпляров приложения рассмотрите возможность использования сборки мусора рабочей станции с отключенной параллельной сборкой мусора. Это уменьшит число переключений контекста, что может повысить быстродействие.

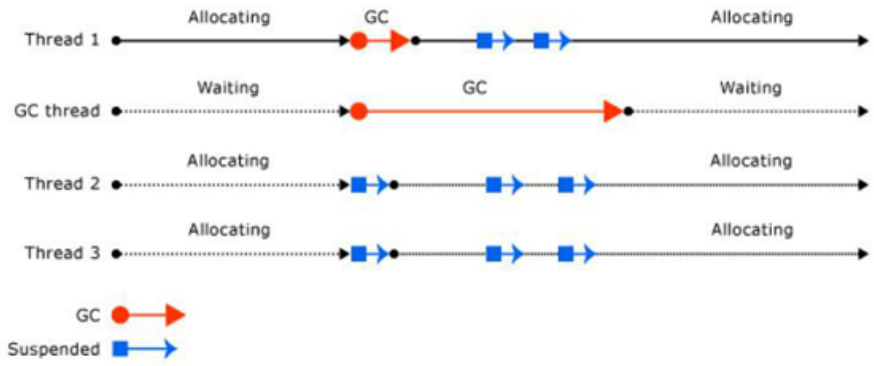

Рисунок 6. Визуализация работы Garbage Collection в режиме сервера

На рисунке 6 показана визуализация того, как все это работает в режиме сервера. Как видим, главное отличие заключается в том, что сборка мусора выполняется для каждого доступного процессора.

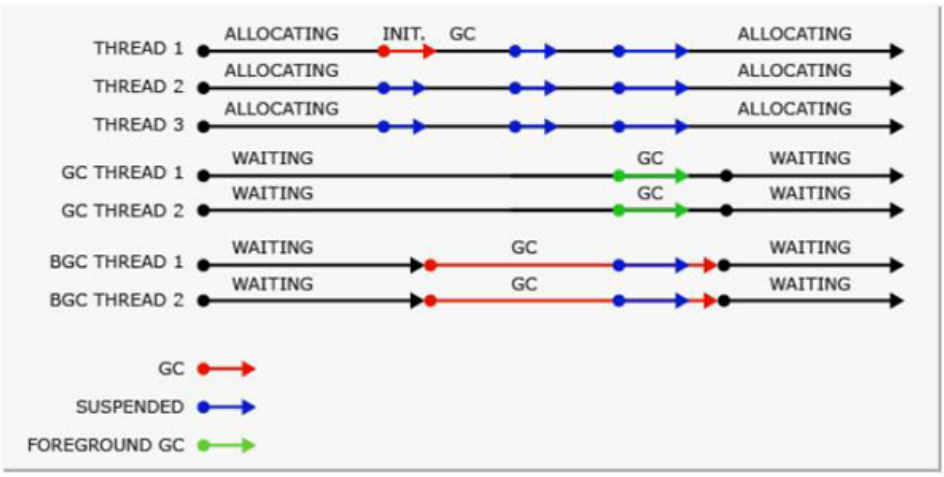

Рисунок 7. Server Background Mode

Начиная с .NET Framework 4.5, фоновая сборка мусора сервера является режимом по умолчанию для сборки мусора сервера. Этот режим функционирует аналогично фоновой сборке мусора рабочей станции, описанной выше, однако с некоторыми отличиями. Для фоновой сборки мусора рабочей станции используется один выделенный поток фоновой сборки мусора, тогда как для фоновой сборки мусора сервера используется несколько потоков — обычно по одному выделенному потоку для каждого логического процессора.

Инструменты мониторинга

GC class

Что можно сделать с помощь GC class из кода подробно описано в статье, но стоит сразу отметить, что это будет просто логирование нужной нам информации в лог, а затем анализ этой информации с помощью каких-то доступных средств. Не очень хороший способ — это не выход из ситуации.

Performance Monitor

Одним из самых мощных инструментов для обнаружения проблем с производительностью в Windows являются встроенные счетчики производительности, так называемые Performance counters. Оснастка Performance monitor — основной инструмент для управления ими.

Performance Viewer

Performance Viewer основан на трассировке событий Windows. Чуть позже поговорим о том, что это такое, зачем это нужно и что можно вообще мониторить с его помощью.

SOS Debugging Extension

SOS Debugging Extension стоит отметить, но уже мало кто использует этот инструмент.

dotMemory

Платный представитель от JetBrains. Стоит отметить, что его open source конкуренты на текущий момент мало в чем ему уступают.

Concurrency Vizualizer

Concurrency Vizualizer — расширение для Visual Studio. К мониторингу памяти относится очень косвенно. При этом оно очень информативное, так как

позволяет увидеть множество параметров по работе приложения в многопоточной среде. С помощью этой утилиты можно проанализировать, когда потоки приостанавливаются, восстанавливают свою работу и т. д.

Performance Monitor

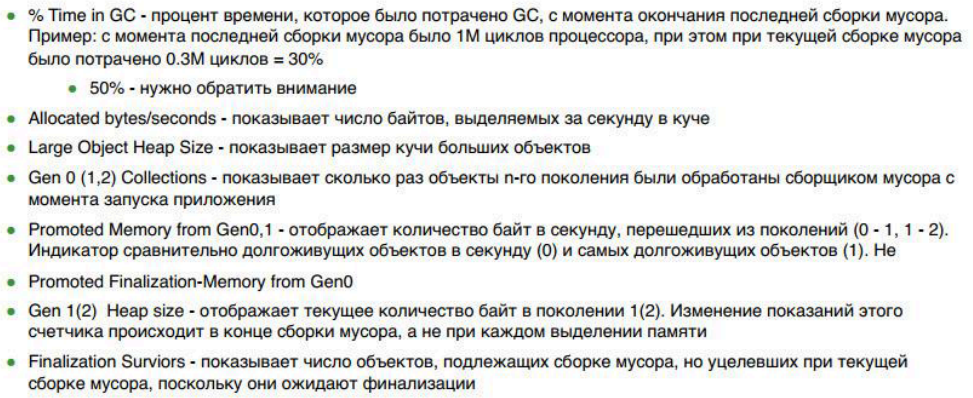

Рисунок 8. Счетчики Performance Monitor

Какие счетчики (counter) предлагает Performance Monitor? Первый счетчик, на который стоит обратить внимание — это процент времени, которое было потрачено самим GC. Этот счетчик делает замеры между двумя сборками мусора, считает циклы процессора, циклы, которые были потрачены в общем и которые были потрачены на сборку мусора. Например, если между двумя сборками прошел 1 миллион циклов процессора и при этом из них 300 тысяч потрачено на сборку мусора, то, соответственно, наше приложение 30% времени тратит просто для того, чтобы собирать мусор.

На какое значение нужно обращать внимание? Это довольно сложный вопрос. К примеру, мы получили цифру 17. Что мне с этой цифрой делать дальше? Из опыта рекомендую обращать внимание на значение 50%. Если 50% — значит половину времени мы тратим впустую. Если это время тратится еще в дата-центрах, то тратятся деньги. И с этим надо что-то делать. Если мы видим цифру в 10 %, то для того, чтобы опустить ее на 5, нужно потратить столько денег, что даже не стоит в это вкладываться.

Следующий параметр, на который стоит обращать внимание — Allocated bytes/second. Он показывает число байтов в секунду, которые мы можем аллоцировать в памяти. Можем посмотреть, какой размер занимает нулевое поколение, первое, второе поколение, сколько занимает Large Object Heap, как перетекают объекты из нулевого поколения в первое, из первого — во второе, количество выживших объектов и т. д.

Finalization Survivors — это счетчик, который показывает количество объектов, которые ушли с очереди финализации и готовы к тому, чтобы началась их чистка.

Пример, как использовать этот инструмент, показан на рисунке 9.

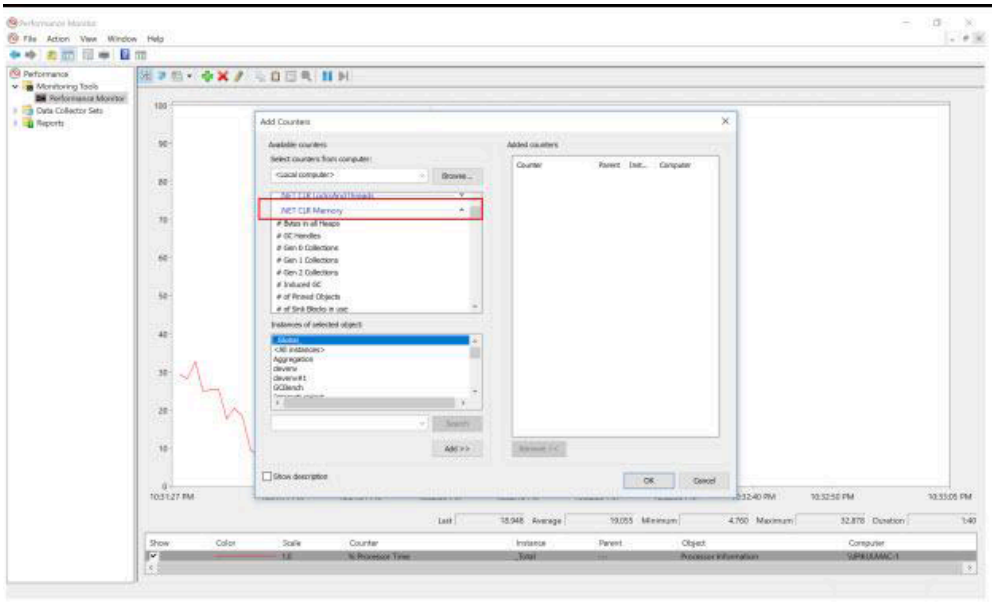

Рисунок 9. Работа с Performance Monitor

Performance Viewer

На мой взгляд, это один из лучших инструментов на текущий момент. Также радует, что производители уже начали задумываться о том, что же делать с Linux, что очень актуально для приложений, написанных под .NET Core. Уже сейчас на их сайте есть небольшой туториал, как снимать метрики с докер хостов. Надеюсь, они будут продолжать развиваться, и мы получим очень хороший инструмент.

Инструмент позволяет мониторить практически все аспекты, которые нужны разработчику для анализа: CPU, стек, есть возможность сделать дамп памяти и проанализировать его, можно посмотреть статистику по GC.

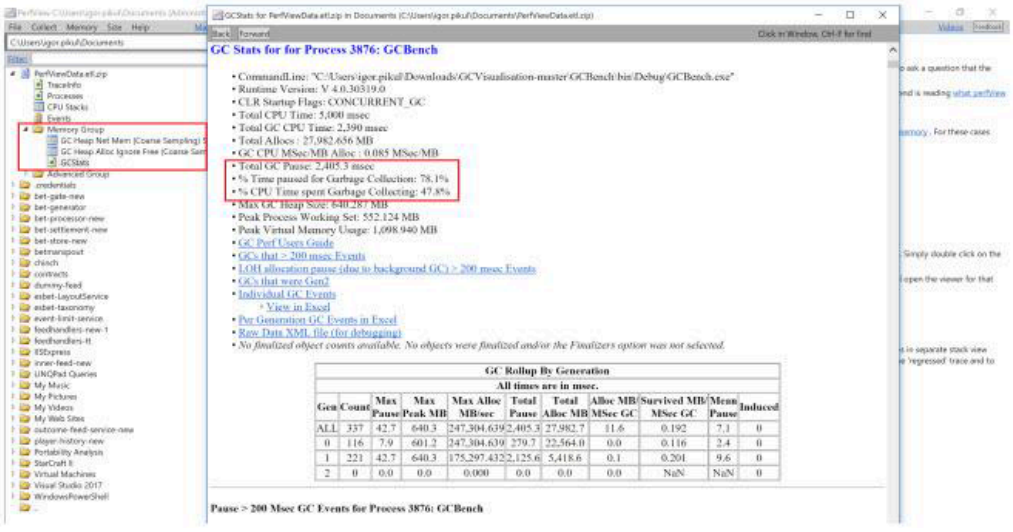

Рисунок 10. Работа с Performance Viewer

Инструмент довольно простой (см. рисунок 10): нажимаем collect и собираем нужные нам метрики. Совет для тех, кто будет использовать — не собирайте метрики долго. Сделал большую ошибку: собрал метрики за минуту и потом ждал пока распарсится минут семь, потом бросил. Должно хватить 5-10 секунд, чтобы понять, что происходит с вашим приложением и что с ним делать. Инструмент может показать все, что связано с GC. На слайде выделены данные, которые касаются GC-статистики. Показывается режим GC, в котором запущено наше приложение, время, которое было потрачено

на паузы GC, процессорное время, которое было потрачено, количество сборок мусора в каждом из поколений, минимальные паузы, пики.

События трассировки

Если посмотреть определение в MSDN или в литературе, то трассировка событий — это высокоэффективная масштабируемая система трассировки с минимальными затратами ресурсов, которая реализуется в Windows. Если немного заглянуть под капот, то очень грубо говоря, этот процесс выглядит так: мы запускаем трассировку наших приложений, это все ложится в обычные файлики, эти файлики потом парсятся, и мы исследуем, что происходит с нашим приложением.

Что вообще можно мониторить в .NET в среде CLR? GC, Runtime, Exceptions, Thread pool, Stack и т. д. Детально о всех метриках можно почитать здесь.

Сейчас мы рассмотрим Garbage Collection в событиях (event), и что они нам позволяют мониторить. Они нам позволяют собирать сведения, которые как раз и относятся к сборке мусора: когда она началась, когда закончилась, в каком поколении. Как долго длилась не покажут — нужно вычислять самому, и это нетривиальная задача. Нетривиальная потому, что если мы посмотрим на режим рабочей станции, когда у нас нет никаких конкурентных режимов, то там все просто: потоки остановились, приостановились, возобновились. И эту дельту мы можем словить по разнице. Когда мы вспоминаем высокоприоритетную сборку мусора, то тут уже все далеко не тривиально. Поэтому уже лучше пользоваться теми инструментами, которые у нас есть.

На GitHub есть библиотеки, которые позволяют научиться работать с данными событиями. К примеру, TraceEvent Library позволяет нам написать приложение, которое будет выполнять трассировку другого приложения. И всю эту информацию спокойно собирать, дебажить и что-то с ней делать.

На рисунке 11 показан небольшой пример, как можно запустить трассировку событий используя TraceEvent Library.

```
static void Main(string[] args)
{
    _etwSession = new TraceEventSession("GC");
    _etwSession.EnableProvider(ClrTraceEventParser.ProviderGuid,
        TraceEventLevel.Verbose,
        (ulong) ClrTraceEventParser.Keywords.GC);

    var exename = args[0];

    var process = Process.Start(exename);

    if (process == null)
        return;

    var processingTask = Task.Factory.StartNew(_ => StartProcessingEvents(process.Id), TaskCreationOptions.LongRunn

    Console.WriteLine("Visualising GC Events, press <ENTER> to exit");
    Console.ReadLine();
```

Рисунок 11. Пример кода

На рисунке 12 происходит магия в части того, как мы собираем все эти счетчики. А вот что из всего этого вышло уже отображено на рисунке 13.

```
private static void StartProcessingEvents(int processId)
{
    _etwSession.Source.Clr.RuntimeStart += data =>...

    _etwSession.Source.Clr.GCAllocationTick += data =>...

    double? gcStart = null;
    _etwSession.Source.Clr.GCStart += data =>...

    _etwSession.Source.Clr.GCStop += data =>...

    double? pauseStart = 0;
    _etwSession.Source.Clr.GCSuspendEEStop += data =>...

    _etwSession.Source.Clr.GCRestartEEStart += data =>...
    _etwSession.Source.Process();
}
```

Рисунок 12. Пример кода

Мы получили следующую информацию: когда у нас начал выполняться GC и сколько времени заняла пауза на GC, какая по счету сборка мусора, с каким поколением работал GC, в каком режиме работает наше приложение.

GC-визуализация

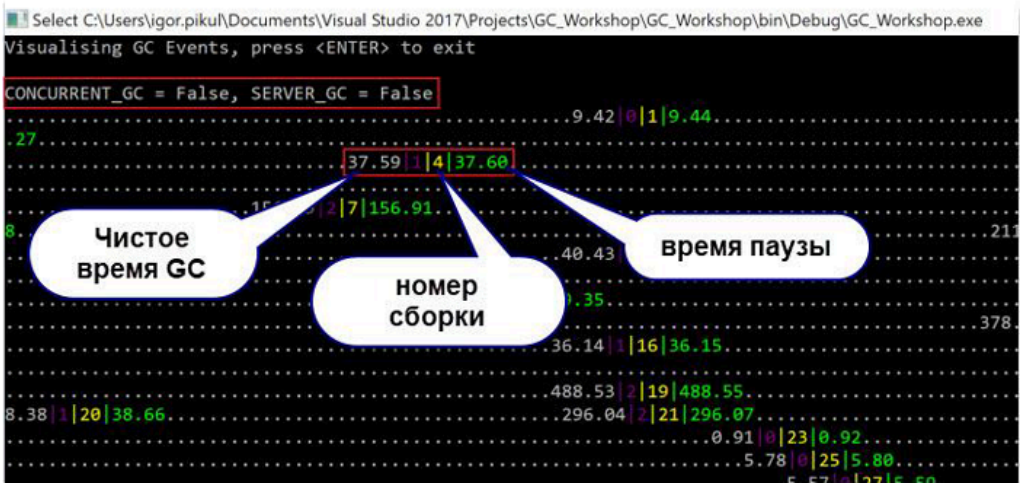

Рисунок 13. Визуализация GC

Есть довольно интересный блог, который ведет Мэт Уоррен. В нем можно найти очень много интересной и полезной информации: как работает Garbage Collection, что же происходит на самом деле «под капотом».

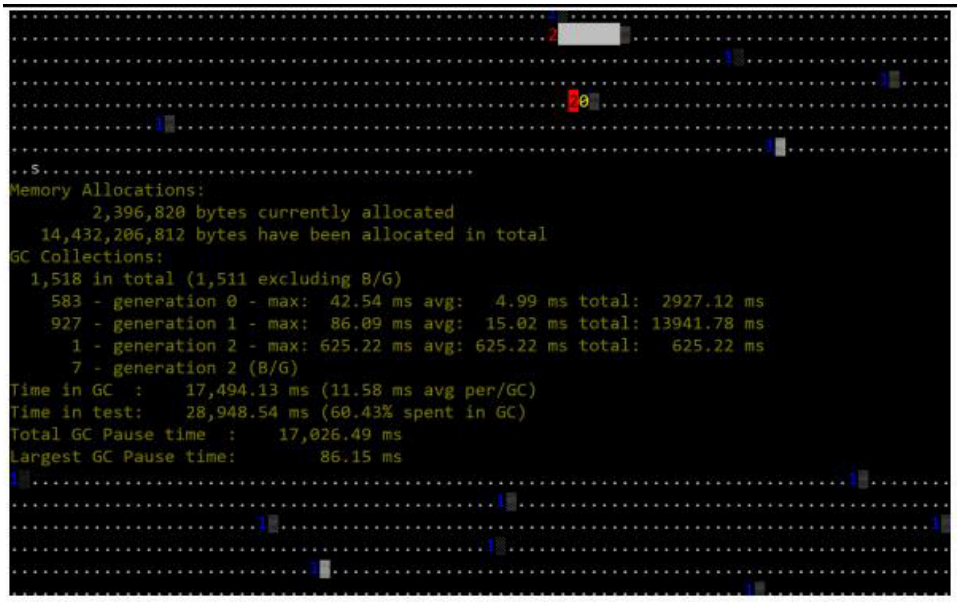

Рисунок 14. Визуализация GC от Мэта Уоррена

На рисунке 14 отображена визуализация работы GC, основанная на трассировке событий, написанная автором блога. Всем, кому интересно понять, как же работает GC, рекомендую разобраться с ним.

GC Mode	Total GC Pause (ms)	% Time paused for GC	Gen0 Count	Gen1 Count	Gen2 Count	Total Allocations (Mb)	Max GC Heap Size (Mb)
Workstation GC	46 118	35.1%	1674	1439	35	174 691	1 561
Background Workstation GC	39 798	30.4%	2109	1451	65	225 554	1 676
Server GC	9 026	9.1%	28	130	8	17 959	1 667
Background Server GC	**8 040**	**8.5%**	23	148	9	16 610	1 724

Рисунок 15. Таблица данных тестирования в разных режимах

В следующей таблице собраны метрики, полученные в результате тестирования одного и того же приложения в разных режимах. Было запущено приложение, основной задачей которого была генерация memory-трафика. Что мы видим? Серверный режим, действительно, уменьшает паузы работы GC, уменьшает количество запусков итераций сборки мусора, но это все делается за счет более интенсивного использования CPU и за счет более интенсивного потребления памяти. Об этом всегда нужно помнить. Если у нас десктопное приложение, в котором нам нужен максимальный отклик, то этот режим явно не для него.

Выводы

Каждый из нас рано или поздно сталкивается с проблемами неоптимальной работы написанного приложения, причины могут быть разные. При их анализе довольно часто мы не смотрим на то, как в таких случаях работает GC, как его работа влияет на работу приложения, оптимальный ли режим GC выбран именно для текущего приложения. А ведь многие ответы как раз и могут быть получены при таком анализе.

Полезные ссылки

- Garbage Collection