Доклад по теме «Управление памятью и сборщик мусора в .NET и Rotor 2.0»

Память

Память — это один из самых важных ресурсов компьютера. Так как современные языки программирования не обязывают программиста работать напрямую с физическими ячейками памяти, на компилятор языка программирования возлагается ответственность за обеспечение доступа к физической памяти, ее распределение и утилизацию.

(В качестве ресурса могут выступать самые разные логические и физические единицы: обычные переменные примитивного типа, массивы, структуры, объекты, файлы и т.д.) Со всеми этими объектами необходимо работать и, следовательно, обеспечить выделение памяти под связанные с ними переменные в программах.

Для этого компилятор должен последовательно выполнить следующие задачи:

- выделить память под переменную;
- инициализировать выделенную память некоторым начальным значением;
- предоставить программисту возможность использования этой памяти;
- как только память перестает использоваться, необходимо ее освободить(возможно, предварительно очистив)
- наконец, необходимо обеспечить возможность последующего повторного использования освобожденной памяти

Проблемы управления памятью

• Необходимо различать уничтожение памяти (уничтожение объекта/уничтожение путей доступа) и утилизацию памяти (сборка мусора)
• Проблема отслеживания различных путей доступа к структуре (различные указатели на одно и то же место, передача параметром в процедуру и т.д.)
=> утилизация памяти обычно проблематична

Фазы управления памятью:

- Начальное распределение памяти
  – методы учета свободной памяти
- Утилизация памяти
  – Простая (перемещение указателя стека)
  – Сложная (сборка мусора)
- Уплотнение и повторное использование
  – память либо сразу пригодна к повторному использованию, либо должна быть уплотнена для создания больших блоков свободной памяти

Сборщик мусора — это такой способ управления ресурсами, обычно — оперативной памятью, выделяемой в куче. Суть проста — программист просит **сборщик мусора** выделить ему кусок памяти, а тот уже сам определяет, когда он станет не нужен и может быть освобожден.

Размер сегментов, выделенных сборщиком мусора, зависит от реализации и может быть изменен в любое время, в том числе при периодических обновлениях. Приложение не должно делать никаких допущений относительно размера определенного сегмента, полагаться на него или пытаться настроить объем памяти, доступный для выделения сегментов.

Некоторые свойства сборки мусора:

• Реализация сборки мусора должна использовать как можно меньший объем рабочей памяти (т.к. сам факт вызова сборки мусора означает недостаток памяти)
• Одно из стандартных решений – использование алгоритма с обращением указателей
- Затраты на сборку мусора обратно пропорциональны объему высвобожденной памяти!
- Если сборка мусора освободила слишком мало памяти, то имеет смысл прекратить исполнение программы

Условия для сборки мусора

Сборка мусора возникает при выполнении одного из следующих условий:

- Недостаточно физической памяти в системе.
- Память, используемая объектами, выделенными в управляемой куче, превышает допустимый порог. Этот порог непрерывно корректируется во время выполнения процесса.
- вызывается метод GC.Collect. Практически во всех случаях вызов этого метода не потребуется, так как сборщик мусора работает непрерывно. Этот метод в основном используется для уникальных ситуаций и тестирования.

Сборка мусора

- производится маркировка активных элементов;

  она начинается с так называемых корневых объектов, список которых хранится в JIT-компиляторе .NET и предоставляется сборщику мусора.

- По окончании маркировки все активные элементы

сдвигаются к началу кучи путем простого копирования памяти.

Так как эта операция компрометирует все указатели, сборщик мусора также исправляет все ссылки, используемые программой.

Замечание:

Реально алгоритм сборки мусора, используемый в .NET, существенно сложнее, так как включает в себя такие оптимизации как слабые ссылки, отдельную кучу для крупных объектов, сборку мусора в многопоточных приложениях и т.д.

Алгоритм выделения памяти в .NET

- Все ресурсы выделяются из управляемой кучи
- Стековый механизм выделения памяти
- Если для создания объекта не хватает памяти, то производится сборка мусора:
  - Производится маркировка активных элементов (список корневых объектов хранится в JIT-компиляторе и предоставляется сборщику мусора)
  - Активные элементы сдвигаются "вниз" путем копирования памяти
  - Так как все указатели могли измениться, сборщикмусора исправляет все ссылки

Управление кучей

• Куча - это блок памяти, части которого выделяются и освобождаются способом, не подчиняющимся какой-либо структуре
• Куча требуется в тех языках, где выделение и освобождение памяти требуется в произвольных местах программы
• Серьезные проблемы выделения, утилизации, уплотнения и повторного использования памяти
• Самая сложная часть управления кучей – это сборка мусора

Стековый механизм

- имеется один указатель
- на следующее свободное место в куче, который после помещения в кучу очередного
- объекта увеличивается на его размер. Понятно, что в какой-то момент указатель кучи
- может выйти за пределы доступного адресного пространства — в этот момент начинает
- работу алгоритм сборки мусора. В целях оптимизации процесса сборка мусора чаще всего
- ограничивается обходом нулевого поколения – чаще всего этого оказывается достаточно.

Перемножение матриц не вызывает сбора мусора C#

```
using System;
/// <summary>
/// Класс, представляющий матрицу
/// </summary>
class Matrix
{
double[,] matrix;
int rows, columns;
// Не вызывается до закрытия приложения
~Matrix()
{
Console.WriteLine("Finalize");
}
public Matrix(int sizeA, int sizeB)
{
rows = sizeA;
columns = sizeB;
matrix = new double[sizeA, sizeB];
}
// Индексатор для установки/получения элементов внутреннего массива
public double this[int i, int j]
{
set { matrix[i,j] = value; }
get { return matrix[i,j]; }
}
// Возвращает число строк в матрице
public int Rows
{
get { return rows; }
}
// Возвращает число столбцов в матрице
public int Columns
{
get { return rows; }
}
}
```

```
/// <summary>
/// Пример перемножения матриц
/// </summary>
class MatMulTest
{
[STAThread]
static void Main(string[] args)
{
int i, size, loopCounter;
Matrix MatrixA, MatrixB, MatrixC;
size = 200;
MatrixA = new Matrix(size,size);
MatrixB = new Matrix(size,size);
MatrixC = new Matrix(size,size);
/* Инициализируем матрицы случайными значениями */
for (i=0; i<size; i++)
{
for (int j=0; j<size; j++)
{
MatrixA [i,j]= (i + j) * 10;
MatrixB [i,j]= (i + j) * 20;
}
}
loopCounter = 1000;
for (i=0; i < loopCounter; i++) Matmul(MatrixA,
MatrixB, MatrixC);

Console.WriteLine("Done.");
Console.ReadLine();
}
// Подпрограмма перемножения матриц
public static void Matmul(Matrix A, Matrix B, Matrix C)
{
int i, j, k, size;
double tmp;
size = A.Rows;
for (i=0; i<size; i++)
{
for (j=0; j<size; j++)
{
tmp = C[i,j];
for (k=0; k<size; k++)
{
tmp += A[i,k] * B[k,j];
}
C[i,j] = tmp;
}
}
}
}
```

Здесь определен класс Matrix, в котором объявляется двухмерный массив для хранения данных матрицы. Метод Main создает три экземпляра этого класса с размерностью по 200×200 (каждый объект занимает примерно 313 Кб). Ссылка на каждую из этих матриц передается по значению методу Matmul (по значению передаются сами ссылки, а не реальные объекты), который затем перемножает матрицы A и B, а результат сохраняет в матрице C.

Для большего интереса метод Matmul вызывается в цикле тысячу раз. Иными словами, эти объекты используются для выполнения тысячи «разных» перемножений матриц и ни разу не инициируются сборки мусора. Следить за числом операций сбора мусора можно при помощи счетчиков производительности памяти, предоставляемых CLR.

Однако при вычислениях с использованием более крупных блоков памяти сбор мусора окажется совершенно неизбежен, как только будет запрошено большее пространство, чем есть в наличии. В таких ситуациях можно прибегнуть к альтернативе, например выделить критичные к быстродействию участки в неуправляемый код и вызывать их из управляемого C#-кода. Но P/Invoke или вызовы .NET interop сопряжены с некоторыми издержками периода выполнения, поэтому такой способ следует использовать в последнюю очередь или в том случае, когда гранулярность операций достаточно груба, чтобы оправдать затраты на вызов.

Сбор мусора не должен мешать разработке высокопроизводительного кода для научных расчетов. Его задача — исключить проблемы с управлением памятью, с которыми иначе вам пришлось бы иметь дело самостоятельно.

Благодаря Rotor(архив от Microsoft под длинным названием Shared Source Common Language Infrastructure (CLI) Implementation Beta (кодовое название) с shared source кодом .NET) для разработчиков появились возможности:

- посмотреть на реализации сборки мусора, JIT-компиляции, протоколов безопасности, организацию среды и систем виртуальных объектов.
- технологию локализации невизуальных компонент
- технологию сборки сложных проектов
- использование UnmanagedApi4MetaData
- более глубокоее понимание работы определенных функции в .NET (Reflection, Remouting, IL)

Поколения

Куча организована в виде поколений, что позволяет ей обрабатывать долгоживущие и короткоживущие объекты. Сборка мусора в основном сводится к уничтожению короткоживущих объектов, которые обычно занимают только небольшую часть кучи. В куче существует три поколения объектов.

Поколение 0. Это самое молодое поколение содержит короткоживущие объекты. Примером короткоживущего объекта является временная переменная. Сборка мусора чаще всего выполняется в этом поколении.

Вновь распределенные объекты образуют новое поколение объектов и неявно являются сборками поколения 0, если они не являются большими объектами, в противном случае они попадают в кучу больших объектов в сборке поколения 2.

Большинство объектов уничтожаются при сборке мусора для поколения 0 и не доживают до следующего поколения.

Большинство объектов уничтожаются при сборке мусора для поколения 0 и не доживают до следующего поколения.

Поколение 1. Это поколение содержит коротко живущие объекты и служит буфером между короткоживущими и долгоживущими объектами.

Поколение 2. Это поколение содержит долгоживущие объекты. Примером долгоживущих объектов служит объект в серверном приложении, содержащий статические данные, которые существуют в течение длительности процесса

Сборки мусора выполняются для конкретных поколений при выполнении соответствующих условий. Сборка поколения означает сбор объектов в этом поколении и во всех соответствующих младших поколениях. Сборка мусора поколения 2 также называется полной сборкой мусора, так как она уничтожает все объекты во всех поколениях (то есть все объекты в управляемой куче).

Выживание и переходы

Объекты, которые не уничтожаются при сборке мусора, называются выжившими объектами и переходят в следующее поколение. Объекты, пережившие сборку мусора для поколения 0, переходят в поколение 1, объекты, пережившие сборку мусора для поколения 1, переходят в поколение 2, а объекты, пережившие сборку мусора для поколения 2, остаются в поколении 2.

Когда сборщик мусора обнаруживает высокую долю выживания в поколении, он повышает порог распределений для этого поколения, чтобы при следующей сборке мусора освобождалась заметная часть занятой памяти. В среде CLR непрерывно контролируется равновесие двух приоритетов: не позволить рабочему набору приложения стать слишком большим и не позволить сборке мусора занимать слишком много времени.

Процесс сборки мусора

Сборка мусора состоит из следующих этапов:

Этап маркировки, выполняющий поиск всех используемых объектов и составляющий их перечень.

Этап перемещения, обновляющий ссылки на сжимаемые объекты.

Этап сжатия, освобождающий пространство, занятое неиспользуемыми объектами и сжимающий выжившие объекты. На этапе сжатия объекты, пережившие сборку мусора, перемещаются к более старому концу сегмента.

Так как сборки поколения 2 могут занимать несколько сегментов, объекты, перешедшие в поколение 2, могут быть перемещены в более старый сегмент. Выжившие объекты поколений 1 и 2 могут быть перемещены в другой сегмент, так как они перешли в поколение 2.

Как правило, куча больших объектов не сжимается, поскольку копирование больших объектов приводит к снижению производительности. Однако начиная с .NET Framework 4.5.1 можно использовать
свойство GCSettings.LargeObjectHeapCompactionMode для сжатия большой кучи объектов по требованию.