Основы платформы Microsoft .NET

Тема:

Управление памятью

Переменные величины и ссылки 1
Выделение памяти 2
Удаление объектов из памяти 4
  Дефрагментация кучи 4
  Поколения объектов 5
Деструкторы 7
Явное освобождение ресурсов 8
Слабые ссылки 8
Управление сборщиком мусора программным путем 9
Литература 9

Переменные величины и ссылки

Для того чтобы сделать возможным написание программ на любых языках программирования, MS .NET Framework предоставляет систему общих типов (*Common Type System – CTS*). Более подробно с CTS рассказано в разделе "Общая система типов".

CTS позволяет создавать переменные двух типов: величины и ссылки.

Под *величинами* понимаются такие переменные, которые непосредственно хранят данные. Причем каждая копия переменной содержит свою копию данных и все операции, совершаемые с одной переменной, не влияют на содержимое другой переменной. Рассмотрим небольшой пример:

```
using System;
class VarTest
{
  public static void Main()
  {
    int nVar1 = 10;
    int nVar2 = nVar1;
    nVar1 = 30;
    Console.WriteLine("nVar1={0}, nVar2={1}", nVar1, nVar2);
  }
}
```

Результат работы: **nVar1**=30, **nVar2**=10

Как видно из примера, переменная **nVar2** хранит свою копию данных, которая создается в момент операции присвоения (**nVar2** = **nVar1**), а операции над переменной **nVar1** (значение этой переменной было изменено) никак не влияют на значение переменной **nVar2**.

Стоит отметить, что переменные величины всегда содержат какое-либо значение, даже если им его еще не присвоили (не инициализировали), а только объявили переменную (однако предсказать значение неинициализированной переменной в большинстве случаев невозможно).

Ссылки содержат только указание на данные, т.е. в них хранятся адреса памяти, где расположены сами данные в памяти. При объявлении ссылки память для хранения данных не выделяется. Для того чтобы память была выделена, необходимо воспользоваться оператором **new**, о работе которого будет рассказано ниже.

Кроме того, в программе может существовать несколько ссылок на одни и те же данные. И если данные будут изменены с использованием одной из ссылок, то естественно, что другая ссылка, указывающая на эту же область памяти, будет ссылаться уже на измененные данные.

Выделение памяти

Давайте рассмотрим, каким образом в MS .NET Framework происходит выделение памяти для переменных величин и ссылок.

Как уже отмечалось ранее, для использования переменных величин достаточно их только объявить, после чего можно приступить к использованию данной переменной, например, присвоить ей некоторое начальное значение. Все переменные величины создаются в стеке программы.

В данном разделе нас интересует, как ведет себя среда MS .NET Framework при работе со ссылками. Поэтому если дальше в тексте отсутствует явное указание типа переменной, то будет подразумеваться, что это ссылка.

Для работы со ссылкой необходимо либо настроить ссылку на уже существующую область памяти, либо воспользоваться оператором **new,** чтобы выделить новый свободный участок памяти для хранения переменной указанного типа. Например, динамическое создание массива из десяти элементов целого типа происходит следующим образом:

```
int[] Data = new int[10];
```

После того, как выделенный участок памяти больше не требуется, его необходимо освободить для возможного повторного использования позже. Оперативная память компьютера, в которой можно выделять память для хранения объектов, называют *динамически распределяемой областью памяти* или *кучей*. Рассмотрим, каким образом реализованы алгоритмы работы с кучей в MS .NET Framework.

В языке C++ выделение памяти происходило следующим образом. Программа запрашивала у системы необходимое количество памяти. Система просматривала кучу в поисках необходимого непрерывного участка и выделяла его программе.

В данном подходе есть один значительный недостаток. В процессе активного использования (выделения и освобождения) памяти из кучи, память становится *фрагментированной*, т.е. появляется достаточно много небольших свободных участков. И может случиться так, что при очередном запросе памяти из кучи не найдется непрерывного свободного блока нужного размера, т.к. свободная память не является непрерывной.

В MS .NET Framework имеется специальный компонент *Garbage Collector (сборщик мусора)*, который отвечает за механизмы выделения и освобождения памяти в куче.

Механизм выделения памяти в куче следующий:

- Программе в момент ее запуска системой выделяется некоторая область памяти (куча). На начало непрерывного свободного участка кучи указывает системная ссылка (будем именовать эту переменную далее как *ссылка свободной памяти - ССП*),
- В момент, когда программа запрашивает память из кучи, используя оператор **new**, Garbage Collector проверяет, достаточно ли памяти в куче (объем свободной памяти определяется количеством байтов между **ССП** и концом кучи). Если памяти достаточно, то объект располагается в памяти, начиная с адреса, на который указывает **ССП**, а сама ссылка перемещается на новое начало свободной памяти,
- При освобождении памяти – и это важно отметить – новые свободные участки памяти не становятся сразу же для повторного использования - вновь запрашиваемые из кучи блоки памяти будет выделяться из начала свободного участка, на который указывает **ССП**.

Таким образом, выделение памяти происходит последовательно. В отличие от механизма, используемого в C++, нет никаких затрат времени на поиск подходящего участка памяти. Очевидно, что два объекта, создаваемые в программе последовательно, будут расположены в памяти друг за другом. Такой механизм выделения памяти работает очень быстро, до тех пор, пока имеется свободная память. Очевидно, что должен существовать механизм *дефрагментации (уплотнения)*[1] свободных участков памяти для ее повторного использования, т.к. размер кучи, в любом случае, является ограниченным (рассмотрение механизма дефрагментации будет выполнено в пункте "Дефрагментация кучи").

---

[1] Дефрагментация (уплотнение) кучи – это объединение всех свободных блоков памяти в один большой непрерывный блок путем сдвига используемых блоков в начало кучи.

Удаление объектов из памяти

В языке программирования C++ при выделении памяти с помощью оператора **new** необходимо обязательно вызвать оператор **delete** для освобождения занятой памяти. Невыполнение этого требования (ошибка в программе) может привести к ряду серьезных ошибок в работе приложения. А проблемы могут быть следующими.

Если в программе выделенный участок памяти не освобождается с помощью оператора **delete** (хотя этот участок больше не используется), то при длительной работе программы вся свободная память будет исчерпана и приложение не сможет нормально продолжать работать. Данная проблема называется "*утечкой памяти*" (участки памяти, которые не используются в программе, но и не возвращены в кучу для повторного использования, обычно именуются в профессиональной литературе как *мусор*). Найти подобную ошибку в программе достаточно сложно.

Другая проблема, связанная с работой с памятью, выделенной из кучи, состоит в попытке использовании участка памяти после его освобождения. После освобождения блока памяти все существующие ссылки на этот участок сохраняются, и вся ответственность за их использование лежит на программисте. Попытка обратиться к освобожденному участку с помощью «старых» ссылок может привести к непредсказуемым последствиям.

Для решения описанных выше проблем в Microsoft .NET используется следующий механизм управления памятью в куче. От программиста не требуется применение оператора освобождения памяти – память освобождается *автоматически*, как только на нее перестают существовать ссылки!!!

Дефрагментация кучи

С целью повторного использования освобождаемой памяти Garbage Collector периодически производит дефрагментацию кучи. Этот процесс состоит из нескольких этапов.

В начале Garbage Collector анализирует все имеющиеся в программе ссылки и запоминает адреса блоков памяти в куче, на которые они ссылаются.

На следующем шаге просматриваются все блоки в куче и проверяется, требуются ли они еще программе, т.е. входят ли они в список адресов, полученных на предыдущем этапе. Если в куче находится блок памяти, который можно освободить (*мусор*), то все размещенные после него объекты копируются на место пустого. Если освобождается блок

памяти в конце кучи, то вместо копирования корректируется значение ссылки **ССП** так, чтобы она указывала на первый свободный байт в куче.

Так как после дефрагментации, проведенной на предыдущем этапе, блоки памяти, на которые указывают используемые в программе ссылки, теперь хранятся по другим адресам в памяти, то Garbage Collector производит изменение значений ссылок, настраивая их на новые адреса.

Основной недостаток описанного выше подхода состоит в том, что в момент дефрагментации может понадобиться копировать большие объемы памяти. Поэтому в MS .NET Framework имеется две кучи, одна из которых предназначена для хранения небольших объектов и управляется с помощью механизма, описанного выше, а вторая предназначена специально для больших объектов. В механизме управления второй кучей дефрагментируется только часть кучи, что существенно ускоряет работу Garbage Collector.

Поколения объектов

При управлении кучей для больших объектов в MS .NET используется специально-разработанный алгоритм для определения возможного времени существования объектов. Знание такой информации позволяет сократить количество объектов, рассматриваемых при дефрагментации памяти, т.к. наибольший объем неиспользуемой памяти (мусора) возникает, конечно, в результате прекращения времени действия кратковременно существующих объектов.

Рассмотрим, каким образом реализован подобный подход. Прежде всего, для всех объектов, создаваемых в куче, определяется время существования, которое подразделяется на три разных уровня (*поколения*). Всего таких выделяется поколений три, каждому из которых присваивается числовой индекс 0, 1 или 2 (данные уровни можно трактовать как краткое, среднее или долгое время существования соответственно). Сразу после создания объекта в куче ему присваивается номер поколения 0. Допустим в куче были созданы объекты (A-E). Через некоторое время объекты C и E больше не используются и память, выделенную для их хранения, можно освободить. Состояние кучи показано на рис. 3.1.

Рис. 3.1. Вид кучи после инициализации: все объекты относятся к поколению 0, сборка мусора не проводилась

Как только суммарный размер объектов поколения 0 превысит пороговое значение, определенное разработчиками среды MS .NET Framework, запускается процесс сборки мусора. Допустим, что при выделении памяти для нового объекта F первый раз запускается сборщик мусора, в процессе которого определяется, что память, выделенную для объектов C и E можно освободить, и объект D перемещается вплотную к объекту B, т.е. память дефрагментируется. Всем объектам поколения 0, "пережившим" сборку мусора, присваивается поколение 1. Состояние кучи после первой сборки мусора показано на рис. 3.2.

Рис. 3.2. Вид кучи после сбора мусора: выжившие объекты из поколения 0 переходят в поколение 1, поколение 0 пусто

Как и раньше, вновь создаваемым объектам будет присвоено поколение 0. Допустим, что были созданы объекты (F-H). В процессе работы программы выясняется, что объекты B и G больше не используются в программе. Текущее состояние кучи показано на рис. 3.3.

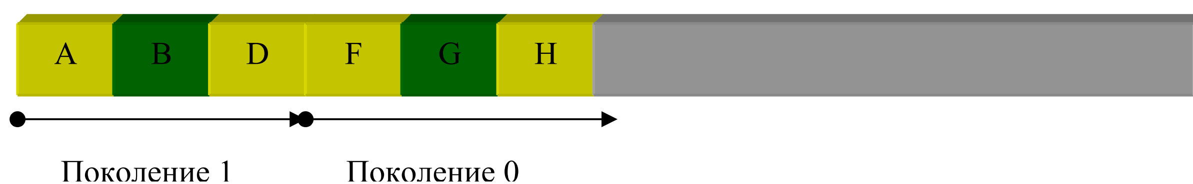

Рис. 3.3. В поколении 0 созданы новые объекты, в поколении 1 появился "мусор"

Допустим, что размер объектов поколения 0 превысил пороговое значение и снова запускается сборщик мусора. Для ускорения процесса дефрагментации обрабатываются объекты только поколения 0, несмотря на то, что в поколении 1 имеется мусор (объект B). Всем выжившим объектам присваивается поколение 1. Текущее состояние кучи показано на рис. 3.4.

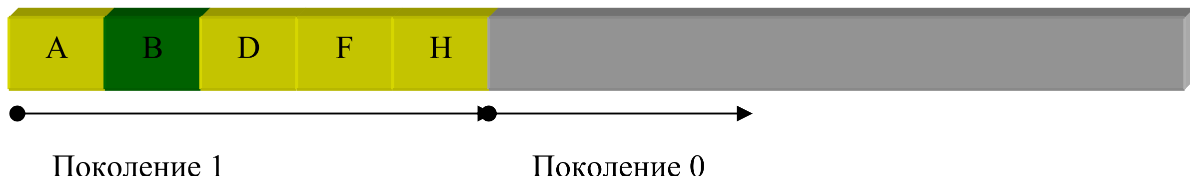

Рис. 3.4. Вид кучи после второго сбора мусора: выжившие объекты из поколения 0 переходят в поколение 1 (увеличивая его размер), поколение 0 пусто

Таким образом, размер поколения 1 медленно увеличивается. Как только суммарный размер объектов поколения 1 превысит свое пороговое значение, при очередной сборке мусора Garbage Collector будет анализировать объекты как поколения 0, так и поколения 1. Выжившие объекты поколения 1 перейдут в поколение 2, а объекты поколения 0 – в поколение 1. Аналогично поколение 2 будет просматриваться только тогда, когда суммарный объем объектов в нем превысит определенное значение для данного поколения. При этом уровень поколения выживших объектов не повышается, т.к. поколение 2 является последним.

Механизм поколений позволяет оптимизировать сбор мусора, т.к. Garbage Collector проводит его именно тогда, когда это необходимо, просматривая при этом только часть кучи.

Деструкторы

Выше было рассказано, как происходит освобождение памяти, выделенной для хранения объекта. Но в процессе своего существования объект может использовать какие-либо ресурсы (например, открытые файлы, открытые соединения с базой данных и т.п.). Корректно завершить работу с ресурсами подобного рода очень важно. Например, если не закрыть файл, то часть данных, которая находилась в буфере и предназначалась для записи в файл, может быть потеряна. Автоматически завершить работу с подобными ресурсами Garbage Collector не может – это задача программиста.

Для решения подобных задач используется механизм деструкторов. *Деструктор* – это метод объекта, который вызывается *автоматически* перед уничтожением объекта (перед освобождением памяти, выделенной для хранения объекта). Следует отметить определенные неоднозначность, связанную с деструкторами в MS .NET Framework. Во всех поддерживаемых языках программирования (кроме C#) деструктором является метод в классе с зарезервированным именем **Finalize**. В C# синтаксис деструктора подобен синтаксису деструктора в C++. Деструктор – это метод, имя которого совпадает с именем класса, который не имеет параметров и не имеет возвращаемого значения. Кроме того,

перед именем деструктора указывается знак "~" (тильда). Для правильного использования деструкторов в MS .NET Framework следует учитывать следующие моменты:

- Деструктор не может быть вызван явно, его вызывает только Garbage Collector во время сборки мусора,
- Время вызова деструктора не определено (известно лишь то, что деструктор будет вызван при отсутствии ссылок на объект до освобождения памяти, выделенной для его хранения),
- Порядок вызова деструкторов не определен.

Стоит отметить, что в MS .NET Framework рекомендуется по мере возможности избегать использования деструкторов. Какие "ловушки" подстерегают программиста при использовании деструкторов и как их избежать можно прочитать в [1].

Явное освобождение ресурсов

Как было отмечено выше, время вызова деструктора не определено. Но может так случиться, что в программе необходимо явно освободить ресурсы, а не ждать, когда в ходе очередной сборки мусора будет автоматически вызван деструктор. Одним из решением данной проблемы является создание обычного метода, который будет заниматься освобождением ресурсов. Это метод, в отличие от деструктора, можно будет вызывать явно. У данного подхода есть один недостаток - пользователь класса не сразу сможет определить (ему придется внимательно изучить документацию), какой именно метод нужно вызвать для освобождения ресурсов. Поэтому в Microsoft .NET предложено стандартное решение в виде реализации специального интерфейса[1]. Например, если объект класса использует ресурсы, которые необходимо освободить сразу в тот момент, когда объект перестает использоваться, не дожидаясь вызова деструктора, то необходимо, чтобы он реализовывал интерфейс **IDisposable**, в котором описан единственный метод **Dispose**. Более подробно об использовании интерфейса **IDisposable** можно прочитать в [2, 3].

Слабые ссылки

Очень часто размер памяти, необходимый приложению, является критичным. Программа может создавать большие объекты, которые желательно удалить сразу, как только возникнет предположение, что он больше не понадобится. Но всегда есть вероятность, что пользователь может вернуться к той части программы, где этот большой

[1] При общем рассмотрении, интерфейсом можно назвать шаблон класса, т.е. класс, в котором методы только объявлены (описаны), но не реализованы (нет программного кода).

объект необходим. Вот тут может возникнуть мысль: "А что, если сборщик мусора еще не успел уничтожить объект…" Ведь есть вероятность, что сборка мусора еще не проводилась. Оказывается, что имеется возможность "воскресить" объект. Для этого необходимо воспользоваться механизмом "слабых ссылок" (weak references).

Под *слабой ссылкой* понимают специальный объект типа **WeakReference**, через который можно получить обычную ("сильную") ссылку на объект. Особенностью таких ссылок является то, что Garbage Collector их не учитывает во время сборки мусора. То есть, даже если на объект будет существовать "слабая" ссылка, объект все равно будет уничтожен, в отличие от обычной ссылки, которая бы "сохранила жизнь" объекту.

Механизм работы со слабыми ссылками следующий. Вначале необходимо создать объект класса **WeakReference** (это будет слабая ссылка), конструктору которого необходимо передать ссылку на существующий в куче объект. В ходе работы программы все обычные ссылки на этот объект могут перестать существовать, и со временем объект будет уничтожен. Чтобы попытаться "воскресить" объект необходимо воспользоваться одним из методов класса **WeakReference**, используя созданную ранее слабую ссылку. И если объект все еще находится в куче, будет получена обычная ссылка, наличие которой предотвратит уничтожение объекта сборщиком мусора.

Более подробно познакомиться с возможностями слабых ссылок можно в [1, 2]

Управление сборщиком мусора программным путем

В данном разделе много раз упоминался сборщик мусора (Garbage Collector). В Microsoft .NET Framework имеется класс GC, с помощью которого можно управлять поведением сборщика мусора. Рассмотрение этой темы выходит за рамки данного раздела, но информацию по ней можно найти в [1-3]

Литература

1. Байдачный С.С. .NET Framework. Секреты создания Windows-приложений. – М.: СОЛОН-Пресс, 2004.
2. Рихтер Д. Программирование на платформе Microsoft .NET Framework. – М.: Издательско-торговый дом "Русская Редакция", 2002.
3. Microsoft Developer Network (MSDN) (http://msdn.microsoft.com)