Всестороннее изучение структур данных с использованием C# 2.0

Часть I: Введение в структуры данных

Эта статья начинает серию из шести статей, которые фокусируют свое внимание на структурах данных и их использовании в разработке приложений. Мы рассмотрим как встроенные структуры данных, присутствующие в.NET Framework, так и базовые структуры данных, которые мы построим сами. Эта первая часть служит введением в структуры данных, давая определения структур данных, анализируя насколько эффективными являются структуры данных и почему такой анализ важен. В этой статье мы также изучим две наиболее часто используемые структуры данных, присутствующие в .NET Framework: Массив (Array) и Список (List).

1. Введение

Добро пожаловать в первую часть серии из шести статей об использовании структур данных в .NET 2.0. Версия 2.0 .NET Framework добавляет новые структуры данных к Библиотеке Базовых Классов (Base Class Library), наряду с новыми возможностями, такими как параметризованные типы (Generics), что делает создание структур данных, обеспечивающих типовую безопастность, гораздо легче чем в версии 1.x. Эта серия статей освещает эти новые структуры данных .NET Framework и изучает новые языковые возможности.

На протяжении этой серии статей мы изучим разнообразные структуры данных, некоторые из которых включены в .NET Framework's Base Class Library, а другие мы построим сами. Если вы не знакомы с этим термином, то «структуры данных» это классы, которые используются для организации данных и обеспечивают различные операции над этими данными. Вероятно, наиболее хорошо известная и широко используемая структура данных – это массив, который содержит непрерывный набор данных, доступаться к которым можно по индексу.

Перед тем как перейти к содержимому этой статьи давайте взглянем на то, о чем мы будем говорить в шести статьях этой серии, так чтобы вы знали, что вас ждет впереди.

В первой части, мы посмотрим на то, почему структуры данных так важны и их влияние на производительность алгоритма. Чтобы определить влияние структур данных на производительность нам необходимо исследовать то, как различные операции, выполняемые над структурой данных, могут быть подвергнуты точному анализу. В конце мы обратим наше внимание на две похожих структуры данных, присутствующие в .NET Framework: Массив (Array) и Список (List). Очень велики шансы того, что вы уже использовали эти структуры данных в выших предыдущих проектах. В этой же статье мы изучим не только операции, которые они предоставляют, но и эффективность этих операций.

Во второй части мы изучим «двоюродных братьев» Списка (List) – Очередь (Queue) и Стек (Stack). Так же как и Список (List), и Очередь (Queue) и Стек (Stack) хранят набор данных и являются структурами данных присутствующими в .NET Framework Base Class Library. В отличие от Списка (List), из которого вы можете извлекать элементы в любом порядке, Очереди и Стеки позволяют доступаться к элементам только в предопределенном порядке. Мы также

изучим некоторые приложения Очередей и Стеков, а также то, как эти классы реализованы в .NET Framework. После изучения Очередей и Стеков мы взглянем на Хэш-таблицы, которые обеспечивают непосредственный доступ к элементам, так же как и ArrayList, но хранят данные, индексированные по строковому ключу.

В то время, как массивы и списки идеально подходят для непосредственного доступа к содержимому, при работе с большими объемами данных эти структуры данных часто являются недостаточно оптимальными кандидатами в том случае, если требуется производить поиск в этих данных. В части III мы изучим структуру данных двоичного дерева поиска, которая и была спроектирована для того, чтобы улучшить время поиска в наборе данных. Несмотря на улучшенное время поиска в двоичном дереве, существует также и несколько недостатков. В части IV мы рассмотрим SkipLists, который является неким совмещением (mix) двоичных деревьев и связанных списков, и которые предназначены для разрешения некоторых проблем присущих двоичным деревьям.

В части V мы уделим внимание структурам данных, которые могут быть использованы для представления графов. Граф это набор узлов и набор ребер, соединяющих различные узлы. Например, карту можно визуализировать как граф, в котором узлы представляют собой города, а дороги между городами – ребра между узлами. Многие проблемы реального мира могут быть абстрактно определены в терминах графов, делая таким образом графы часто используемой структурой данных.

И наконец, в части VI мы рассмотрим структуры данных, которые представляют множества и несовместимые множества (disjoint sets). Множество это набор неупорядоченных данных. Несовместымые множества – это набор множеств, которые не имеют между собой общих элементов. И множество, и несовместимые множества имеют много применений в программах ежедневного использования, которые мы детально и рассмотрим в нашей заключительной части.

2. Анализ производительности структур данных

Размышляя над определенным приложением или частной проблемой программирования, многие разработчики (включая и меня) большей частью интересуются написанием алгоритма для решения проблемы или добавлением интересных опций к приложению для улучшения взаимодействия пользователя с приложением. Очень редко, если вообще когда-либо, вы услышите о ком-нибудь, кто был бы в восторге от того, какую структуру данных он использует. Однако, структуры данных используемые для определенного алгоритма могут существенно повлиять на его производительность. Наиболее часто упоминаемый пример это поиск элемента в структуре данных. В неотсортированном массиве время обработки пропорционально количеству элементов в массиве. А в двоичных деревьях поиска или SkipLists, требуемое время логарифмически пропорционально числу элементов. При поиске в достаточно большом объеме данных выбранная структура данных может существенно повлиять на производительность приложения, что может быть даже визуально измерено в секундах или даже минутах.

Так как структура данных используемая алгоритмом может существенно повлиять на производительность алгоритма очень важно, чтобы существовал точный метод, с помощью которого можно было бы сравнивать эффективность различных структур данных. Как разработчики использующие структуру данных мы прежде всего интересуемся тем, как изменяется производительность структуры данных с увеличением количества хранимых в ней элементов. То есть, как влияет на время выполнения операций над структурой данных при добавлении нового элемента к этой структуре данных?

Рассмотрим следующий сценарий: представим, что вам поставлена задача написать программу, которая будет получать в качестве входных данных массив строк, содержащих имена файлов. Задача вашей программы состоит в определении того, содержит ли массив строк имена файлов с заданным расширением. Один из подходов будет состоять в том, чтобы просканировать массив и установить некий флаг, как только будет встречен, например, XML файл. Код мог бы выглядеть следующим образом:

```
public bool DoesExtensionExist(string [] fileNames, string extension)
{
      int i = 0;
      for (i = 0; i < fileNames.Length; i++)
         if (String.Compare(Path.GetExtension(fileNames[i]), extension, true) == 0)
            return true;

      return false;   // Если мы достигли это строки, то мы не нашли файла с заданым
                      // расширением
   }
}
```

Здесь мы видим, что в худшем случае — когда в массиве нет файла с заданным расширением или такой файл есть, но он является последним файлом в списке — мы должны просмотреть каждый элемент массива в точности один раз. Для анализа эффективности сортировки массива, мы должны задать себе следующий вопрос: "Предположим, что я имею массив с n элементами. Если я добавлю еще один элемент, то есть массив будет содержать $n + 1$ элемент, каково будет новое время выполнения?" (Термин "время выполнения" несмотря на его имя, не измеряет абсолютное время, требуемое для выполнения программы, он скорее относится к числу шагов, которые программа должна выполнить, чтобы завершить требуемую задачу. Работая с массивами, рассматриваемые шаги представляют собой количество доступов к элементам массива, которые необходимо сделать.) Так как поиск данного значения в массиве требует посещения, потенциально, каждого элемента массива, то если мы имеем $n + 1$ элементов в массиве, нам необходимо выполнить $n + 1$ проверок. То есть, время, требуемое для поиска в массиве линейно пропорционально числу элементов в массиве.

Вид анализа описанный здесь называется *асимптотическим анализом* (*asymptotic analysis*), так как он исследует как изменяется эффективность структуры данных при приближении размера структуры данных к бесконечности. Нотация, используемая в асимптотическом анализе, называется нотацией «*большого ‘O’*». Для того, чтобы описать производительность поиска в неотсортированном массиве с помощью нотации большого ‘O’ следует написать $O(n)$. Большая буква O в обозначении и является тем ключевым элементом, из которого и произошла терминология «*большого ‘O’*», а буква n указывает, что число шагов требуемых для поиска в массиве растет линейно с ростом размера массива.

Более методический способ вычисления асимптотического времени выполнения блока кода состоит в следовании этих простых шагов:

1. Определить шаги, составляющие время выполнения алгоритма. Как было сказано ранее, применительно к массивам, обычно рассматриваемые шаги это операции доступа к массиву для чтения и записи. Для других структур данных эти шаги могут отличаться. Обычно вы рассматриваете шаги, в которые вовлечена сама структура данных, но не атомарные операции, выполняемые компьютером. То есть, относительно блока кода приведенного выше, я проанализировал его время выполнения, принимая во внимание только количество доступов к массиву, не беспокоясь при этом о времени необходимом для создания и инициализации переменных или проверки того, равны ли две строки.
2. Найдите строки кода, которые выполняют интересующие вас шаги. Поставьте 1 в конце каждой такой строки.

3. Для каждой строки в конце которой стоит 1, проверьте находится ли она в цикле. Если это так, то замените 1 на 1 умноженное на максимальное количество повторений цикла. Если вы имеете два или более вложенных цикла продолжите умножение для каждого из циклов.
4. Найдите самое большое выражение, которое вы записали. Это и есть время выполнения.

Давайте применим эти шаги к блоку кода приведенному выше. Мы уже определили, что интересующие нас шаги это операции доступа к массиву. Переходя к шагу 2 отметим, что есть только одна строка, в которой производится доступ к массиву `fileNames`: в параметре метода `String.Compare()`, поэтому поставим 1 в конце этой строки. Теперь применияя шаг 3 заметим, что доступ к массиву `fileNames` в методе `String.Compare()` происходит внутри цикла, который выполняется по крайней мере n раз (где n это размер массива). Поэтому уберем 1 в конце строки и заменим ее на n. Это наибольшее значение, поэтому время выполнения обозначается $O(n)$.

$O(n)$, или линейное время представляет собой одно из многочисленных возможных асимптотических времен выполнения. Другие примеры включают $O(\log_2 n)$, $O(n \log_2 n)$, $O(n^2)$, $O(2^n)$, и так далее. Не вдаваясь в математических деталей большого O, чем меньше выражение внутри скобок для больших значений n, тем лучше производительность структуры данных. Например, операция которая выполняется за $O(\log n)$ более эффективна, чем операция которая выполняется за $O(n)$ так как $\log n < n$.

На протяжении всей серии статей каждый раз когда мы будем изучать новую структуру данных и операции над ней, мы обязательно будем вычислять ее асимптотическое время выполнения и сравнивать его с временем выполнения аналогичных операция для других структур данных.

2.1. Асимптотическое время выполнения и алгоритмы реального мира

Асимптотическое время выполнения алгоритма измеряет производительность алгоритма в терминах количества шагов, которые должен выполнить алгоритм, приближаясь к бесконечности. Когда говорится, что время выполнения одного алгоритма больше времени выполнения другого алгоритма, то математически это означает, что существует некоторое количество шагов, таких что, как только это количество будет превышено, то алгоритму с большим временем выполнения всегда потребуется больше времени для выполнения, чем алгоритму с меньшим временем выполнения. Однако для случая с малым количеством шагов, алгоритм с асимптотически большим временем выполнения может работать быстрее, чем алгоритм с меньшим временем выполнения.

Например, существует бесчисленное количество алгоритмов сортировки массива, которые имеют различное время выполнения. Один из наиболее простых алгоритмов сортировки – пузырьковая сортировка – который использует пару вложенных циклов `for` для сортировки элементов массива. Пузырьковая сортировка показывает время выполнения $O(n^2)$ из-за наличия двух вложенных циклов `for`. Альтернативный алгоритм сортировки – сортировка слиянием, который делит массив на две половины и рекурсивно сортирует каждую половину. Время выполнения для сортировки слиянием составляет $O(n \log_2 n)$. Асимптотически сортировка слиянием намного более эффективна, чем пузырьковая сортировка, но для маленьких массивов пузырьковая сортировка может быть более эффективной. При сортировке слиянием расходуются ресурсы не только при выполнении рекурсивных функциональных запросов, но и при переобъединении сортированых половин массива, тогда как пузырковая сортировка выполняется с помощью простых циклов для квадратичного массива, обмен значениями между элементами массива происходит по мере необходимости. В целом, сортировка слиянием должна выполнить меньше шагов, но шаги, которые должна выполнить сортировка слиянием

более дорогие, чем шаги, выполняемые пузырьков ой сортировкой. Для больших массивов эта дополнительная плата за шаг незначительна, но для маленьких массивов пузырьковая сортировка может действительно быть более эффективной.

Асимптотический анализ определенно имеет свое место, так как асимптотическое время выполнения двух алгоритмов может показать, как один алгоритм превосходит другой, когда оба алгоритма работают над достаточно большими объемами данных. Использование только асимптотического анализа для суждения об эффективности алгоритма является, однако, безрассудным, так как действительное время выполнения различных алгоритмов зависит от специфичных факторов реализации, таких как количество данных «закачиваемых» в алгоритм. Принимая решение, какую структуру данных нужно использовать в проектах из реальной жизни, рассмотрите асимптотическое время выполнения, но также внимательно профилируйте ваше приложение, чтобы удостовериться в реальном влиянии на производительность выбранной вами структуры данных.

3. Любимая каждым, однородная структура данных с линейным, непосредственным доступом: массив

Массивы это одна из наиболее простых и наиболее широко используемых структур данных в компьютерных программах. Массивы в любом языке программирования имеют несколько общих свойств:

- Содержимое массива хранится в непрерывном блоке памяти.
- Все элементы массива должны быть одного типа или одного производного типа; поэтому массивы называют однородными структурами данных.
- К элементам массива можно доступаться непосредственно. Если вы используете массив и знаете, что вам нужно доступиться к *i*-му элементу, то вы просто используете вот такую строку кода: *arrayName*[*i*].

Наиболее общие операции, выполняемые с массивом это:

- Распределение массива
- Доступ к элементам массива

В C#, когда объявляется массив (или любая другая переменная ссылочного типа), она имеет значение `null`. То есть, следующая строка кода просто создает переменную с именем `booleanArray` равную `null`:

```
bool [] booleanArray;
```

Перед тем как начать использовать массив мы должны создать экземпляр массива, который может хранить заданное число элементов. Это достигается с помощью следующего синтаксиса:

```
booleanArray = new bool[10];
```

Или в более общем виде:

```
arrayName = new arrayType[allocationSize];
```

Эта операция распределяет непрерывный блок памяти в CLR-managed heap достаточно блоьшой для того, чтобы хранить *allocationSize* число элементов типа *arrayType*s. Если *arrayType* является типом значений, то будет создано *allocationSize* неупакованных значений

типа *arrayType*. Если *arrayType* является ссылочным типом, то создается *allocationSize* ссылок на элементы типа *arrayType*.

Чтобы помочь понять как .NET Framework хранит массивы внутри, рассмотрим следующий пример:

```
bool [] booleanArray;
FileInfo [] files;

booleanArray = new bool[10];
files = new FileInfo[10];
```

Здесь `booleanArray` это массив значений `System.Boolean`, в то время как `files` это массив ссылочных значений типа `System.IO.FileInfo`. Рисунок 1 показывает изображение CLR-managed кучи после исполнения этих четырех строк кода.

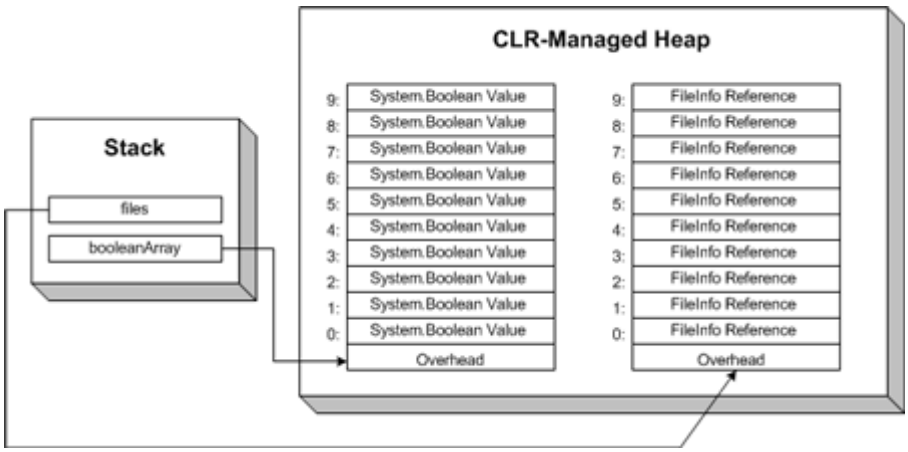

Рисунок 1. Содержимое массивов располагается смежно в управляемой (managed) куче.

Нужно помнить, что десять элементов в массиве files это ссылки (*references*) на экземпляры `FileInfo`. Рисунок 2 поясняет это, показывая расположение памяти после того, как мы выполним присваивание некоторым элементам массива `files` значений экземпляров `FileInfo`.

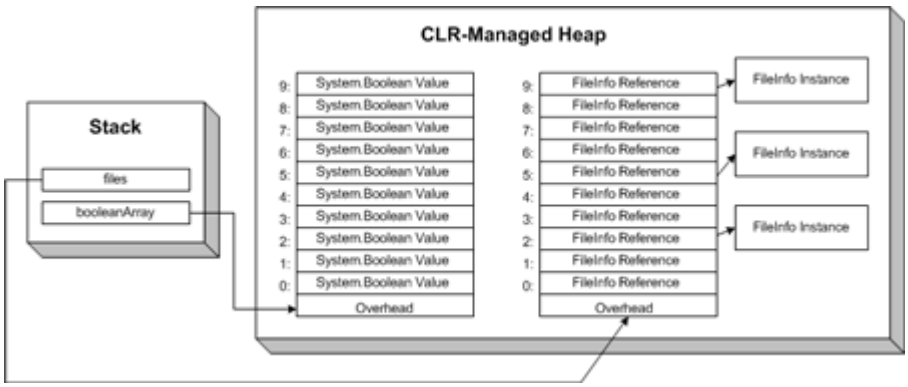

Рисунок 2. Содержимое массива расположено рядом в управляемой (managed) куче.

Все массивы в .NET позволяют читать и писать их элементы. Синтаксис для доступа к массиву выглядит следующим образом:

```
// Читать элемент массива
bool b = booleanArray[7];
```

```
// Писать элемент в массив
booleanArray[0] = false;
```

Время выполнения операции доступа к элементу массива обозначается через *O*(1), так как оно постоянно. То есть, независимо от того, как много элементов хранится в массиве, необходимо одинаковое время для доступа к любому элементу. Это постоянное время доступа возможно только потому что элементы массива хранятся непрерывно, следовательно для доступа к элементу массива требуется знать только начальное положение массива в памяти, размер каждого элемента, и номер элемента к которому производится доступ.

Поймите, что в каждое обращение к элементу массива ведет к CLR проверне, чтобы гарантировать, что запрашиваемій индекс находистя в пределах границ массива. Если индекс выходит за пределы массива, то генерируется `IndexOutOfRangeException`. Эта проверка помогает убедиться, что вы пытаетесь обратится за пределы массива, к другой памяти. Тем не менее, эта проверка не затрагивает ассимптомическую прожолжительность доступа к массиву, потому что время выполнения проверки не увеличивается вместе с увеличением размера массива.

При работе с массивами вам может понадобиться изменить число элементов, которые он может содержать. Чтобы сделать это вам необходимо создать новый экземпляр массива заданного размера и скопировать содержимое старого массива в новый, с измененным размером. Этот процесс может быть совершен с помощью следующего кода:

```
// Создать массив целых чисел из трех элементов
int [] fib = new int[3];
fib[0] = 1;
fib[1] = 1;
fib[2] = 2;

// Новый масив, состоящий из 10 элементов
int [] temp = new int[10];

// Копировать массив fib в temp
fib.CopyTo(temp, 0);

// Назначаем fibу temp
fib = temp;
```

После последней строки кода, `fib` ссылается на 10-элементный массив с элементами типа `Int32`. Элементы с 3-го по 9-й массива `fib` будут содержать значение по умолчанию типа `Int32` равное 0.

Массивы это замечательные структуры данных для хранения наборов однородных данных, к которым требуется только непосредственный доступ. Поиск в неотсортированном массиве имеет линейное время выполнения. В то время как это вполне приемлемо при работе с малыми массивами, или при осуществлении небольшого числа операций поиска, когда же ваше приложение хранит большие массивы, в которых, к тому же, часто производятся операции поиска, то существуют другие структуры данных, более приспособленные для таких задач. Мы рассмотрим некоторые из таких структур данных в последующих статьях этой серии. Помните, что если вы производите поиск в массиве по какому-либо свойству и массив *отсортирован* по этому свойству, то вы можете использовать для поиска алгоритм, называемый бинарным (двоичным) поиском, время выполнения которого пропорционально *O*(log *n*), который наравне со временем поиска в бинарных деревьях поиска. На самом деле класс `Array` содержит статический метод `BinarySearch()`.

4. Создание эффективных, повторно используемых структур данных, обеспечивающих типовую безопастность

При создании структуры данных для решения частной задачи очень часто внутренняя структура данных может быть «подогнана» под специфику проблемы. Например, представим, что вы работаете над списочным приложением. Одной из сущностей этой системы будет служащий, поэтому вам нужно создать класс `Employee` с соответствующими свойствами и методами. Чтобы представить набор служащих вы можете использовать массив элементов типа `Employee`, но возможно вам понадобится некоторая дополнительная функциональность не присутствующая в массиве, или вы просто не хотите тратить время на написание кода по отслеживанию емкости массива и изменению его размера при необходимости. Один из подходов будет состоять в создании специализированной структуры данных которая использует внутренний массив элементов типа `Employee`, и предлагающей методы для расширения базовой функциональности массива, такие как автоматическое изменение размера, поиск в массиве определенного объекта типа `Employee`, и так далее.

Эта структура данных покажет себя очень удачной в приложении, так что вы захотите повторно использовать ее в других приложениях. Однако эта структура данных не открыта для повторного использования, так как она сильно связана со списочным приложением и поддерживает только элементы типа `Employee` (или типов производных от `Employee`). Одна из возможностей сделать структуру данных более гибкой это организовать работу с внутренним массивом элементов типа `object`, в противоположность элементам типа `Employee`. Так как все типы в .NET Framework являются производными от типа `object`, то такая структура данных сможет хранить любые типы данных. Это поможет сделать вашу структуру данных повторно используемой в других приложениях и сценариях.

Не удивительно, что .NET Framework уже содержит структуры данных, которые обеспечивают эту функциональность — класс `System.Collections.ArrayList`. ArrayList поддерживает внутренний массив элементов типа `object` и обеспечивает автоматическое изменение размера массива, при увеличении количества элементов добавляемых к ArrayList. Так как ArrayList использует массив элементов типа object, разработчики могут добавлять в него данные любого типа — строки, целые числа, объекты типа `FileInfo`, экземпляры `Form`, и прочее.

Хотя ArrayList предоставляет дополнительную гибкость в сравнении со стандартным массивом, эта гибкость дается за счет производительности. Так как ArrayList хранит массив элементов типа `objects`, то при чтении значения из ArrayList вам необходимо явно приводить тип элемента к типу переменной в которой будет храниться читаемое значение. Вспомните, что массив значений — таких как `System.Int32`, `System.Double`, `System.Boolean`, и так далее — хранится непрерывным блоком в управляемой куче и в неупакованном (unboxed) виде. Однако внутренний массив структуры ArrayList представляет собой массив ссылок. Поэтому если у вас есть ArrayList хранящий только значения, то каждый элемент ArrayList представляет собой ссылку на упакованное значение, как это показано на Рисунке 3.

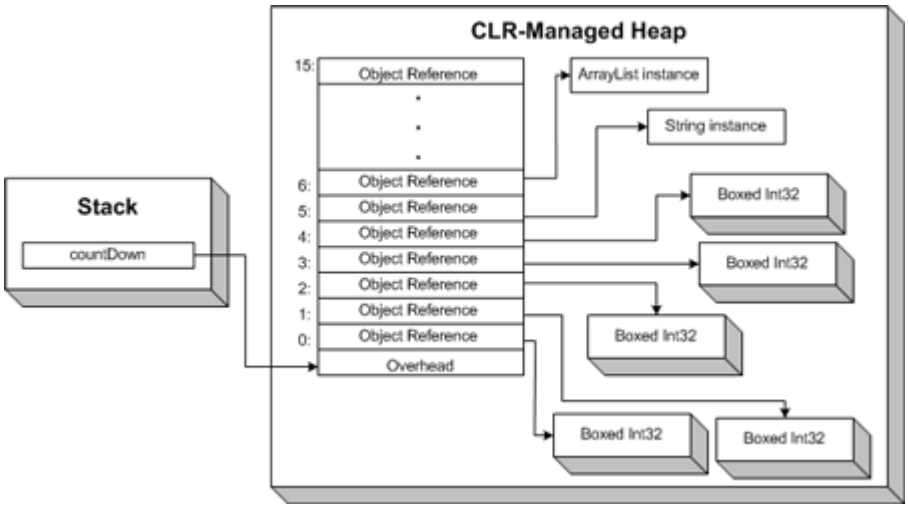

Рисунок 3. ArrayList содержит непрерывный блок ссылок на объекты

Упаковка и распаковка, наряду с дополнительным уровнем разыменования (indirection), которое получается за счет использования значений в ArrayList, может повлиять на производительность вашего приложения, если оно будет использовать большие ArrayLists и производить много операций чтения и записи. Как показано на Рисунке 3, то же самое распределение памяти выполняется для ссылок как в случае ArrayLists так и в случае обычных массивов.

Работа с массивом элементов типа `object` также привносит потенциальную возможность ошибки, которая не может быть обнаружена до времени выполнения. Разработчик может намереваться добавлять в ArrayLists только элементы одного определенного типа, но так как ArrayLists позволяет добавлять элементы любого типа, то добавление элемента неправильного типа не сможет быть определено на стадии компиляции. Более того, такая ошибка не проявиться вплоть до времени выполнения, означая тем самым, что ошибка не будет найдена до тестирования или, в худшем случае, вплоть до реального использования приложения.

4.1. Параметризованные типы (Generics) приходят на помощь

К счастью, проблемы типизирования и производительности, связанные с ArrayList решены в .NET Framework 2.0, благодаря параметризованным типам (*Generics*). Параметризованные типы (Generics) позволяют разработчикам создавать структуры данных с отложенным выбором типов данных. Эти типы, ассоциированные со структурой данных, могут быть выбраны разработчиком, который использует эту структуру данных. Чтобы лучше понять параметризованные типы (Generics) давайте посмотрим на пример создания коллекции, обеспечивающей типовую безопастность. В частности мы создадим класс, который поддерживает внутренний массив элементов типа, который задаст пользователь, с соответствующими методами для чтения и добавления элементов во внутренний массив.

```
public class TypeSafeList<T>
{
    T[] innerArray = new T[0];
    int currentSize = 0;
    int capacity = 0;

    public void Add(T item)
    {
        // проверить нужно ли изменить размер массива
        if (currentSize == capacity)
        {
            // изменить размер массива
```

```
            capacity = capacity == 0 ? 4 : capacity * 2;  // удвоить емкость
            T[] copy = new T[capacity];    // создать массив с новым размером
            Array.Copy(innerArray, copy, currentSize);  // копировать массив
            innerArray = copy;    // присвоить innerArray новому большему массиву
        }

        innerArray[currentSize] = item;
        currentSize++;
    }

    public T this[int index]
    {
        get
        {
            if (index < 0 || index >= currentSize)
                throw new IndexOutOfRangeException();
            return innerArray[index];
        }
        set
        {
            if (index < 0 || index >= currentSize)
                throw new IndexOutOfRangeException();
            innerArray[index] = value;
        }
    }

    public override string ToString()
    {
        string output = string.Empty;
        for (int i = 0; i < currentSize - 1; i++)
            output += innerArray[i] + ", ";

        return output + innerArray[currentSize - 1];
    }
}
```

Отметьте, что первая строка кода в определении класса, содержит идентификатор типа `T`. Этот синтаксис означает, что класс потребует от разработчика, который его использует, задания одного определенного типа. Это тип назван как `T`, хотя может использоваться любое другое имя. Этот идентификатор типа используется внутри свойств и методов класса. Например, внутренний массив типа `T`, и метод `Add()` имеющий входной параметр типа `T`, который затем будет добавлен к массиву.

Чтобы объявить переменную этого класса, разработчику необходимо задать тип `T`, вот так:

```
TypeSafeList<type> variableName;
```

Следующий отрывок кода демонстрирует создание экземпляра TypeSafeList, который хранит целые числа, и наполнение этого списка первыми 25 числами Фибоначчи.

```
TypeSafeList<int> fib = new TypeSafeList<int>();
fib.Add(1);
fib.Add(1);

for (int i = 2; i < 25; i++)
    fib.Add(fib[i - 2] + fib[i - 1]);

Console.WriteLine(fib.ToString());
```

Основные преимущества параметризованных типов (Generics) включают:

- **Обеспечение типовой безопастности (Type-safety):** разработчик, использующий класс TypeSafeList может добавлять только элементы указанного типа или типа производного от указанного. Например, при попытке добавить строку к `fib` TypeSafeList в приведенном выше примере, приведет к генерации ошибки времени компиляции.
- **Производительность:** Параметризованные типы (Generics) убирают необходимость проверки типов во время исполнения, и устраняют таким образом стоимость накладных расходов связанную с упаковкой (boxing) и распаковкой (unboxing).
- **Повторное использование:** Параметризованные типы (Generics) позволяют разорвать сильную связь между структурой данных и приложением, для которого эта структура данных была создана. Это, в свою очередь, повышает степень повторного использования структур данных.

Многие из структур данных, которые мы изучим на протяжении этой серии статей, это структуры данных, которые используют параметризованные типы (Generics) и при создании структур данных – таких как, например, двоичное дерево, которое мы построим в Части II – мы сами будем использовать параметризованные типы (Generics).

5. Список (List): однородный, динамически изменяющийся массив

Массив, как мы видели, спроектирован для хранения заданного количества элементов одного типа в виде непрерывного блока. Массивы, будучи простыми в использовании, могут быстро стать неудобством если вам понадобится часто изменять размер массива или если вы не знаете сколько элементов вам понадобится хранить во время инициализации массива. Одна из возможностей избежать изменения размера массива вручную состоит в создании структуры данных, которая служит оболочкой (wrapper) для массива, обеспечивая доступ для чтения и записи и автоматически изменяя размер массива при необходимости. Мы начали создание нашей собственной структуры данных с такими возможностями в предыдущем разделе — TypeSafeList, но в действительности нет необходимости ее реализовывать, так как .NET Framework предоставляет такой класс. Этот класс, `List`, находится в пространстве имен `System.Collections.Generics`.

Класс `List` содержит внутренний массив и предоставляет методы и свойства, которые, среди прочего, позволяют читать и писать элементы во внутренний массив. Класс `List`, как и массив, это однородная структура данных. Это означает, что вы можете хранить элементы только одинакового типа или производных типов в данном Списке (`List`). `List` использует параметризованные типы (Generics), новую возможность в версии 2.0 .NET Framework, для того, чтобы дать возможность разработчику задать во время разработки тип данных, который будет хранить `List`.

Поэтому при создании экземпляра `List` вы должны задать тип данных, которые будут храниться в списке, используя синтаксис для параметризованных типов:

```
// Создать Список (List) целых чисел
List<int> myFavoriteIntegers = new List<int>();

// Создать Список (List) строк
List<string> friendsNames = new List<string>();
```

Отметьте, что тип данных, который может хранить Список (`List`) задается как в объявлении, так и в реализации `List`. При создании нового Списка (`List`) вы не должны задавать размер Списка (`List`), однако вы можете задать стартовый (начальный) размер, передавая целое число

в конструкторе, либо через свойство List's Capacity. Чтобы добавить элемент в Список (List) просто используйте метод Add(). Список (List), как и массив, позволяет непосредственно доступаться к элементам, используя их порядковый номер, или индекс. Следующий фрагмент кода показывает создание Списка (List) целых чисел, наполнение списка некоторыми начальными значениями с помощью метода Add(), а затем чтение и запись элементов Списка (List) через их порядковые номера (индексы).

```
// Создать Список (List) целых чисел
List<int> powersOf2 = new List<int>();

// Добавть 6 целых чисел в Список (List)
powersOf2.Add(1);
powersOf2.Add(2);
powersOf2.Add(4);
powersOf2.Add(8);
powersOf2.Add(16);
powersOf2.Add(32);

// Изменить второй элемент Списка, чтобы он содержал 10
powersOf2[1] = 10;

// Вычислить 2^3 + 2^4
int sum = powersOf2[2] + powersOf2[3];
```

Список (List) берет массив и оборачивает его в класс, который скрывает сложность реализации. Создавая Список (List) вам не нужно явно задавать его начальный размер. При добавлении элементов в Список (List) вам не нужно беспокоится об изменении размера структуры данных, как вы это делали с массивом. Более того, Список (List) имеет много других методов, которые обеспечивают функциональность обычного массива. Например, чтобы найти элемент в массиве вам потребуется написать цикл for для прохода по массиву (если, конечно, массив не отсортирован). С помощью Списка (List) вы просто используете метод Contains(), чтобы определить присутствует ли элемент в массиве, или IndexOf(), чтобы найти порядковый номер элемента в массиве. Класс Список (List) также поддерживает метод BinarySearch() для эффективного поиска в отсортированном массиве, а также такие методы как Find(), FindAll(), Sort(), и ConvertAll(), которые используют делегаты для выполнения операций, требующих несколько строк кода для доступа к массиву.

Асимптотическое время выполнения операций над Списком (List) такое же, как и для массива. В то время как операции для работы со Списком (List) на самом деле имеют некоторые накладные расходы, отношение между числом элементов в Списке (List) и стоимостью одной операции такое же, как и у стандартного массива.

6. Заключение

Эта статья, первая в серии из шести статей, начала наши обсуждения структур данных с определения того, почему структуры данных так важны и предоставляя методологию для определения производительности структур данных. Этот материал очень важно понимать, так как возможность уметь анализировать время выполнения для различных структур данных это основной инструмент при решении какую из структур данных использовать для решения частной проблемы разработки программного обеспечения.

После изучения того как анализировать структуры данных мы обратили наше внимание на изучение двух наиболее общих структур данных в .NET Framework Base Class Library: System.Array и System.Collections.Generics.List. Массивы предоставляют непрерывный

блок однотипных данных. Их основное преимущество состоит в том, что они обеспечивают моментальный доступ к элементам для чтения и записи. Их слабость состоит в том, что для поиска элемента в массиве требуется просмотреть (потенциально) каждый элемент массива (в случае неотсортированного массива), а также тот факт, что изменение размера массива требует написания кода.

Класс Список (`List`), служащий оболочкой для функциональности массива, предоставляет различные полезные методы. Например, метод `Add()` добавляет элемент к Списку (List) и автоматически изменяет размер массива при необходимости. Метод `IndexOf()` помогает разработчику в поиске определенного элемента в Списке (`List`). Функциональность предоставляемая Списком ничем не отличается от той, которую вы можете реализовать используя обычные массивы, но класс Список (List) дает вам возможность не беспокоится о написании такого кода самим.

В следующей статье мы сперва обратим наше внимание на двух «двоюродных братьев» Списка (**List**): на классы Стека (**Stack**) и Очереди (**Queue**). Мы также рассмотрим ассоциативные массивы, которые являются массивами индексированными строковыми ключами в противоположность целочисленному индексированию. Ассоциативные массивы предоставляются .NET Framework Base Class Library с использованием классов Хэш Таблиц (**Hashtable**) и Словарей (**Dictionary**).

Удачного программирования!

Игорь Изварин