МАССИВЫ И КОЛЛЕКЦИИ

Массивы хороший инструмент группировки данных. Однако, массивы хранят фиксированное количество объектов, а иногда заранее не неизвестно, сколько потребуется объектов. И в этом случае намного удобнее применять коллекции. Еще один плюс коллекций состоит в том, что некоторые из них реализует стандартные структуры данных, например, стек, очередь, словарь, которые могут пригодиться для решения различных специальных задач.

Большая часть классов коллекций содержится в пространствах имен System.Collections (простые необобщенные классы коллекций), System.Collections.Generic (обобщенные или типизированные классы коллекций) и System.Collections.Specialized (специальные классы коллекций). Также для обеспечения параллельного выполнения задач и многопоточного доступа применяются классы коллекций из пространства имен System.Collections.Concurrent.

В этой главе рассмотрим использование массивов и обобщенных коллекций.

Массивы

Синтаксис массивов в C#

```
type[ ] name; // правильно
type name[ ]; // неправильно в C#
type[4] name; // также неправильно C#
```

Для объявления двумерных массивов, используются пустые индексы через запятую.

```
int[,] grid;
```

Для доступа к элементам массива используются индексы в квадратных скобках. Нумерация начинается с нуля. Если размерность массива больше одного, то индексы перечисляются через запятую.

```
long[] row; int[,] grid;
… …

row[3]; grid[1,2];
```

В C# индексы массива автоматически проверяются на удовлетворение размерности, при выходе за интервал выдаётся исключение IndexOutOfRangeException. Для проверки размерности можно использовать свойство массива Length и метод GetLength.

Сравним массивы с коллекциями. Коллекции гибче массивов, могут хранить различные элементы и имеют переменную длину. Но в связи с этим работа с коллекциями медленнее.

```
ArrayList flexible = new ArrayList( );
flexible.Add("one"); // Добавили строку...
flexible.Add(99); // Добавили int
```

Создание коллекции только для чтения

```
ArrayList flexible = new ArrayList( ); ...
ArrayList noWrite = ArrayList.ReadOnly(flexible);
noWrite[0] = 42; // Исключение при выполнении
```

Объявление массива не создает его, так как массив – это ссылочный тип. При объявлении массива можно не знать его размерность, но при создании знать размер необходимо. Память для массива выделяется последовательно.

```
long[] row = new long[4];
int[,] grid = new int[2,3];
```

Можно использовать список инициализации при создании массива. Необходимо объявить все элементы, в качестве элементов можно использовать выражения. Те же правила справедливы для многомерных массивов: необходимо определить все строки и в каждой строке все элементы.

```
int[,] data = new int[2,3]{
{2,3,4},
{5,6,7} }
```

Размерность массива можно задавать как константами, так и вычисляемыми значениями. Единственное ограничение при вычисляемой длине, нельзя использовать список инициализации.

```
string s = Console.ReadLine();

int size = int.Parse(s);
```

```
long[] row = new long[size];
```
При копировании переменной массива, не создается новый массив, создается новая ссылка на тот же массив.
```
long [] row = new long[4];
long [] copy = row;
row[0]++; // изменит copy[0]
```
Рассмотрим некоторые свойства и методы класса System.Array.
· Свойство Rank – размерность массива.
· Свойство Length – общая длина массива, для многомерных массивов количество всех ячеек.

System.Array – класс, от которого неявно наследуют все массивы в C#.
· Sort – сортировка массива, поддержка интерфейса IComparable.
· Clear – очистка массива, все элементы устанавливаются в NULL.
·Clone – создаёт копию массива, копирует все элементы. Этот метод не следит за значениями в элементах. Если там ссылки на объекты, то они просто скопируются, новых объектов создано не будет.
· GetLength – по номеру размерности получаем длину массива в этой размерности.
· IndexOf – ищет значение в массиве, если нашел возвращает индекс первого вхождения, иначе -1.
При возвращении массива из метода его размеры не задаются, скобки остаются пустыми. Если задать, получим ошибку при компиляции. Также с многомерными массивами.
```
static int[,] CreateArray( ) {
string s1 = System.Console.ReadLine( );
int rows = int.Parse(s1);
string s2 = System.Console.ReadLine( );
int cols = int.Parse(s2);
return new int[rows,cols];
}
```
При передаче массива в метод в качестве параметра, новый массив не создается, передается ссылка на тот же массив. Все действия с массивом в

методе сохранятся для вызывающего метода. Если нужно избежать этого, необходимо передавать копию массива при помощи метода
```
Array.Copy
```
При вызове приложения из командной строки можно использовать строку параметров, которые разделяются пробелом. Например:
```
C:\> pkzip –add –rec –path=relative c:\code *.cs
```
Тогда если pkzip написан на C#, получим массив
```
string[ ] args = {
"-add",
"-rec",
"-path=relative", "c:\\code",
"*.cs"
};
```
Этот массив получим при запуске метода Main.
```
class PKZip
{
static void Main(string[] args) {
}
}
```

Для прохода по массиву можно использовать цикл foreach. Тогда не нужен счетчик, проверка длины массива и обращение к элементу.
Две следующие конструкции эквивалентны
```
for (int i = 0; i < args.Length; i++) {
System.Console.WriteLine(args[i]);
}
```
и

```
foreach (string arg in args) {
System.Console.WriteLine(arg);
}
```
Также foreach можно использовать и для многомерных массивов
```
int[,] numbers = { {0,1,2}, {3,4,5} };
foreach (int number in numbers) {
System.Console.WriteLine(number);
}
```

Списки – List<T>

Класс List<T> является самым простым из классов коллекций. Его можно использовать практически так же, как массив, ссылаясь на существующий в коллекции List<T> элемент с использованием обычной для массивов системы записи с квадратными скобками и индексом элемента. Можно добавить элемент к концу коллекции List<T>, воспользовавшись имеющимся в ее классе методом Add, которому предоставляется добавляемый элемент. Размер коллекции увеличивается List<T> автоматически.

```
List<int> list = new List<int>();
list.Add(3);
list.Add(1);
list[0]=list[1]+3;
```

Указывать размер коллекции List<T> при ее создании не обязательно. Коллекция может изменять свои размеры по мере добавления (или удаления) элементов. Но надо иметь в виду, что на физическое добавление элементов уходит время процессора, и при необходимости нужно указать начальный размер. Но если он будет превышен, то в силу необходимости коллекция List<T> просто расширится. Если первую строку примера изменить на

```
List<int> list = new List<int>(2);
```

то результат будет один и тот же, но последний вариант отработает быстрее. В этом случае команда Add – это просто инициализация очередного элемента в конце списка.

Для удаления из коллекции List<T> указанного элемента можно воспользоваться методом Remove. Элементы коллекции List<T> автоматически перестроятся, закрывая образовавшееся пустое место. С помощью метода RemoveAt можно также удалить элемент, указав его позицию в коллекции List<T>. Можно вставить элемент в середину коллекции List<T>, воспользовавшись для этого методом Insert. При этом размер коллекции List<T> также изменится автоматически. Размер коллекции, т.е. количество инициализированных элементов, можно получить через свойство (только на чтение) Count, а емкость коллекции – через свойство (чтение и запись) Capacity.

Приведем ряд методов класса List<T>:

· Add(T) – добавляет объект в конец списка List<T>.
· AddRange(IEnumerable<T>) – добавляет элементы указанной коллекции в конец списка List<T>.
· Clear()– удаляет все элементы из коллекции List<T>.
· Contains(T) – определяет, входит ли элемент в коллекцию List<T>.
· ConvertAll<TOutput>(Converter<T, TOutput>) – преобразует элементы текущего списка List<T> в другой тип и возвращает список преобразованных элементов.
· Exists(Predicate<T>) – определяет, содержит ли List<T> элементы, удовлетворяющие условиям указанного предиката.
· Find(Predicate<T>) – выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает первое найденное вхождение в пределах всего списка List<T>.
· FindAll(Predicate<T>) – извлекает все элементы, удовлетворяющие условиям указанного предиката.
· FindIndex(Predicate<T>) – выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает отсчитываемый от нуля индекс первого найденного вхождения в пределах всего списка List<T>.
· IndexOf(T) – осуществляет поиск указанного объекта и возвращает отсчитываемый от нуля индекс первого вхождения, найденного в пределах всего списка List<T>.
· Insert(Int32, T) – вставляет элемент в коллекцию List<T> по указанному индексу.
· Remove(T) – удаляет первое вхождение указанного объекта из коллекции List<T>.

· RemoveAll(Predicate<T>) – удаляет все элементы, удовлетворяющие условиям указанного предиката.
· RemoveAt(Int32) – удаляет элемент списка List<T> с указанным индексом.
· Sort() – сортирует элементы во всем списке List<T> с помощью функции сравнения по умолчанию.
· ToArray() – копирует элементы списка List<T> в новый массив.

Многие методы коллекции List<T> содержатся и в других коллекциях.

Двухсвязные списки – LinkedList<T>

Класс коллекций LinkedList<T> реализует двусвязный список. В каждом элементе списка содержится значение элемента со ссылкой на следующий элемент списка (свойство Next) и его предыдущий элемент (свойство Previous).

В классе LinkedList<T> записи, присущие массивам, не поддерживаются. Вставка элементов осуществляется отличным от List<T> способом. Можно воспользоваться методом AddFirst для вставки элемента в начало списка с перемещением предыдущего первого элемента дальше по списку и установки в качестве значения его свойства Previous ссылки на новый элемент. Аналогично этому для вставки элемента в конец списка можно воспользоваться методом AddLast. Для вставки элемента перед указанным элементом списка или после него можно воспользоваться методами AddBefore и AddAfter.

Первый элемент коллекции LinkedList<T> можно найти, запросив значение свойства First, а свойство Last даст ссылку на последний элемент списка. Для последовательного обхода элементов связного списка можно приступить к этой операции с одного конца и пошагово применять ссылки из свойства Next или Previous, пока не будет найден элемент, у которого это свойство имеет значение null. Конечно же, лучше воспользоваться инструкцией foreach, которая выполнит последовательный обход элементов вперед по списку LinkedList<T>-объекта, автоматически остановившись в конце.

Удаление элемента из коллекции LinkedList<T> осуществляется с помощью методов Remove, RemoveFirst и RemoveLast.

Преимущество связного списка проявляется в том, что операция вставки элемента в середину выполняется очень быстро. Это происходит за счет того, что только ссылки Next (следующий) предыдущего элемента и Previous (предыдущий) следующего элемента должны быть изменены так, чтобы указывать на вставляемый элемент. В классе List<T> при вставке нового элемента все последующие должны быть сдвинуты.

Естественно, у связных списков есть и свои недостатки. Так, например, все элементы таких списков доступны лишь друг за другом. Поэтому для нахождения элемента, находящегося в середине или конце списка, требуется довольно много времени. Связный список не может просто хранить элементы внутри себя. Вместе с каждым из них ему необходимо иметь информацию о следующем и предыдущем элементах. Поэтому LinkedList<T> содержит элементы типа LinkedListNode<T>. С помощью класса LinkedListNode<T> появляется возможность обратиться к предыдущему и последующему элементам списка. Класс LinkedListNode<T> определяет свойства List, Next, Previous и Value. Свойство List возвращает объект LinkedList<T>, ассоциированный с узлом. Свойства Next и Previous предназначены для итераций по списку и для доступа к следующему и предыдущему элементам. Свойство Value типа T возвращает элемент, соответствующий узлу.

Если в простом списке List<T> каждый элемент представляет объект типа T, то в LinkedList<T> каждый узел помимо T содержит также объект класса LinkedListNode<T>. Этот класс имеет следующие свойства:

· Value –значение узла, представленное типом T.
· Next – ссылка на следующий элемент типа LinkedListNode<T> в списке. Если следующий элемент отсутствует, то имеет значение null.
· Previous – ссылка на предыдущий элемент типа LinkedListNode<T> в списке. Если предыдущий элемент отсутствует, то имеет значение null.

Используя методы класса LinkedList<T>, можно обращаться к различным элементам, как в конце, так и в начале списка:

· AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode): вставляет узел newNode в список после узла node.
· AddAfter(LinkedListNode<T> node, T value): вставляет в список новый узел со значением value после узла node.

· AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode): вставляет в список узел newNode перед узлом node.
· AddBefore(LinkedListNode<T> node, T value): вставляет в список новый узел со значением value перед узлом node.
· AddFirst(LinkedListNode<T> node): вставляет новый узел в начало списка.
· AddFirst(T value): вставляет новый узел со значением value в начало списка.
· AddLast(LinkedListNode<T> node): вставляет новый узел в конец списка.
· AddLast(T value): вставляет новый узел со значением value в конец списка.
· RemoveFirst(): удаляет первый узел из списка. После этого новым первым узлом становится узел, следующий за удаленным.
· RemoveLast(): удаляет последний узел из списка.

Словари – Dictionary<TKey, TValue>

Массив и объекты типа List<T> предоставляют способ отображения на элемент целочисленного индекса. Целочисленный индекс указывается с помощью квадратных скобок (например, [4]), и извлекается элемент по индексу 4, будучи фактически пятым. Но иногда может понадобиться реализация отображения, при котором используется другой, нецелочисленный тип, например string, double или DateTime. В других языках программирования такая организация хранения данных часто называется ассоциативным массивом. Эта функциональная возможность реализуется в классе Dictionary<TKey, TValue> путем внутреннего обслуживания двух массивов, один из которых предназначен для ключей, от которых выполняется отображение на одно из отображаемых значений. Когда в коллекцию Dictionary<TKey, TValue> вставляется пара «ключ–значение», класс автоматически отслеживает принадлежность ключа к значению, позволяя быстро и легко извлекать значение, связанное с указанным ключом. В конструкции класса Dictionary<TKey, TValue> имеется ряд важных особенностей.

В коллекции Dictionary<TKey, TValue> не могут содержаться продублированные ключи. Если для добавления уже имеющегося в массиве ключа вызывается метод Add, выдается исключение. Но для добавления пары «ключ–значение» можно воспользоваться системой записи с использованием квадратных скобок, не опасаясь при этом выдачи исключения, даже если ключ уже был добавлен: любое значение с таким же самым ключом будет переписано новым значением. Протестировать наличие в коллекции Dictionary<TKey, TValue> конкретного ключа можно с помощью метода ContainsKey.

По внутреннему устройству коллекция Dictionary<TKey, TValue> является разряженной структурой данных, работающей наиболее эффективно, когда в ее распоряжении имеется довольно большой объем памяти. По мере вставки элементов размер коллекции Dictionary<TKey, TValue> в памяти может очень быстро увеличиваться.

Когда для последовательного обхода элементов коллекции Dictionary<TKey, TValue> используется инструкция foreach, возвращается элемент KeyValuePair<TKey, TValue>. Это структура, содержащая копию элементов ключа и значения, находящихся в коллекции Dictionary<TKey, TValue>, и доступ к каждому элементу можно получить через свойства Key и Value. Эти элементы доступны только для чтения, и их нельзя использовать для изменения данных в коллекции Dictionary<TKey, TValue>.

Отметим, что есть схожая необобщенная коллекция – Hashtable, имеющая такой же функционал. Однако эта коллекция проигрывает в скорости при работе с однотипными объектами и используется, как и все необобщенные коллекции, для группировки различных объектов.

Относительно производительности рассмотренных здесь коллекций заметим, что добавление нового объекта (Add) быстрее делает List<T>, медленнее – Dictionary<TKey, TValue>. На поиск элемента уходит примерно одинаковое время, однако поиск по ключу существенно быстрее в Dictionary<TKey, TValue>. Удаление объекта медленнее делается в классе List<T>.