Лекция. Массивы и коллекции, структуры данных

МДК 01.04. Системное программирование

План лекции:

1. Понятие структур данных и их классификация
2. Массивы в .NET: особенности и использование
3. Коллекции в .NET: List, Dictionary<TKey, TValue>, HashSet
4. Специализированные коллекции: Queue, Stack, LinkedList
5. Производительность операций с различными структурами данных
6. Практические рекомендации по выбору структур данных

---

1. Понятие структур данных и их классификация

Структура данных - это способ организации, хранения и управления данными, который обеспечивает эффективный доступ и модификацию.

Классификация структур данных:

Линейные структуры:

- Элементы располагаются последовательно
- Примеры: массивы, списки, очереди, стеки

Неллинейные структуры:

- Элементы имеют иерархические или сетевые связи
- Примеры: деревья, графы, хеш-таблицы

```
// Пример различных структур данных
public class DataStructuresOverview
{
    // Линейная структура - массив
    int[] linearArray = new int[] { 1, 2, 3, 4, 5 };

    // Нелинейная структура - дерево (упрощенное представление)
    public class TreeNode
    {
        public int Value { get; set; }
        public List<TreeNode> Children { get; set; } = new List<TreeNode>();
    }
}
```

По типу доступа к элементам:

- **Прямой доступ** (массивы) - доступ по индексу за O(1)

- **Последовательный доступ** (связные списки) - необходимо пройти предыдущие элементы
- **Ассоциативный доступ** (словари) - доступ по ключу

---

2. Массивы в .NET: особенности и использование

Массив - структура данных фиксированного размера, содержащая элементы одного типа.

Особенности массивов в C#:

- Фиксированный размер после создания
- Быстрый доступ по индексу
- Элементы располагаются в памяти последовательно

```
public class ArrayExamples
{
    public void DemonstrateArrays()
    {
        // Создание массивов разными способами
        int[] numbers1 = new int[5]; // Массив из 5 нулей
        int[] numbers2 = new int[] { 1, 2, 3, 4, 5 };
        int[] numbers3 = { 10, 20, 30, 40, 50 };

        // Многомерные массивы
        int[,] matrix = new int[3, 3]
        {
            { 1, 2, 3 },
            { 4, 5, 6 },
            { 7, 8, 9 }
        };

        // Зубчатый массив (массив массивов)
        int[][] jaggedArray = new int[3][];
        jaggedArray[0] = new int[] { 1, 2 };
        jaggedArray[1] = new int[] { 3, 4, 5 };
        jaggedArray[2] = new int[] { 6 };

        // Операции с массивами
        numbers1[0] = 100; // Запись
        int value = numbers2[2]; // Чтение

        // Копирование массивов
        int[] copy = new int[numbers2.Length];
        Array.Copy(numbers2, copy, numbers2.Length);

        // Сортировка
        int[] unsorted = { 5, 3, 1, 4, 2 };
        Array.Sort(unsorted);

        // Поиск
        int index = Array.BinarySearch(unsorted, 3);
```

```
    }
}
```

Преимущества массивов:

- Высокая производительность при доступе по индексу
- Минимальные накладные расходы памяти
- Кэш-дружественность (элементы в памяти рядом)

Недостатки:

- Фиксированный размер
- Дорогая вставка/удаление в середину
- Необходимость знать размер заранее

---

3. Коллекции в .NET: List, Dictionary<TKey, TValue>, HashSet

List - динамический массив

```
public class ListExamples
{
    public void DemonstrateList()
    {
        List<string> fruits = new List<string>();

        // Добавление элементов
        fruits.Add("Apple");
        fruits.Add("Banana");
        fruits.AddRange(new[] { "Orange", "Grape" });

        // Вставка в середину
        fruits.Insert(1, "Mango");

        // Удаление элементов
        fruits.Remove("Banana");
        fruits.RemoveAt(0);

        // Поиск
        bool hasApple = fruits.Contains("Apple");
        int index = fruits.IndexOf("Orange");

        // Capacity и Count
        Console.WriteLine($"Capacity: {fruits.Capacity}, Count: {fruits.Count}");

        // Итерация
        foreach (string fruit in fruits)
        {
            Console.WriteLine(fruit);
        }
```

```
        // Фильтрация с помощью LINQ
        var longFruits = fruits.Where(f => f.Length > 5).ToList();
    }
}
```

Dictionary<TKey, TValue> - хеш-таблица

```
public class DictionaryExamples
{
    public void DemonstrateDictionary()
    {
        Dictionary<string, int> ages = new Dictionary<string, int>();

        // Добавление элементов
        ages.Add("Alice", 25);
        ages["Bob"] = 30; // Альтернативный синтаксис
        ages["Charlie"] = 35;

        // Проверка существования ключа
        if (ages.ContainsKey("Alice"))
        {
            int age = ages["Alice"];
        }

        // Безопасное получение значения
        if (ages.TryGetValue("David", out int davidAge))
        {
            Console.WriteLine($"David's age: {davidAge}");
        }

        // Удаление
        ages.Remove("Bob");

        // Итерация по парам ключ-значение
        foreach (KeyValuePair<string, int> pair in ages)
        {
            Console.WriteLine($"{pair.Key}: {pair.Value}");
        }

        // Использование пользовательского компаратора
        Dictionary<string, int> caseInsensitive =
            new Dictionary<string, int>(StringComparer.OrdinalIgnoreCase);
    }
}
```

HashSet - множество уникальных элементов

```
public class HashSetExamples
{
    public void DemonstrateHashSet()
    {
        HashSet<int> set1 = new HashSet<int> { 1, 2, 3, 4, 5 };
        HashSet<int> set2 = new HashSet<int> { 4, 5, 6, 7, 8 };

        // Проверка принадлежности
        bool containsThree = set1.Contains(3); // true

        // Добавление (игнорирует дубликаты)
        set1.Add(6); // true (успешно добавлен)
        set1.Add(1); // false (уже существует)

        // Операции с множествами
        var union = new HashSet<int>(set1);
        union.UnionWith(set2); // {1, 2, 3, 4, 5, 6, 7, 8}

        var intersection = new HashSet<int>(set1);
        intersection.IntersectWith(set2); // {4, 5, 6}

        var difference = new HashSet<int>(set1);
        difference.ExceptWith(set2); // {1, 2, 3}

        // Проверка подмножества
        bool isSubset = set1.IsSubsetOf(union); // true
    }
}
```

---

4. Специализированные коллекции: Queue, Stack, LinkedList

Queue - очередь (FIFO)

```
public class QueueExamples
{
    public void DemonstrateQueue()
    {
        Queue<string> queue = new Queue<string>();

        // Добавление в очередь
        queue.Enqueue("First");
        queue.Enqueue("Second");
        queue.Enqueue("Third");

        // Просмотр первого элемента без удаления
        string first = queue.Peek(); // "First"

        // Извлечение из очереди
        while (queue.Count > 0)
```

```
        {
            string item = queue.Dequeue();
            Console.WriteLine($"Processing: {item}");
        }

        // Практический пример: обработка задач
        Queue<Action> tasks = new Queue<Action>();
        tasks.Enqueue(() => Console.WriteLine("Task 1"));
        tasks.Enqueue(() => Console.WriteLine("Task 2"));

        while (tasks.Count > 0)
        {
            tasks.Dequeue()(); // Выполнение задачи
        }
    }
}
```

Stack - стек (LIFO)

```
public class StackExamples
{
    public void DemonstrateStack()
    {
        Stack<string> stack = new Stack<string>();

        // Добавление в стек
        stack.Push("First");
        stack.Push("Second");
        stack.Push("Third");

        // Просмотр верхнего элемента
        string top = stack.Peek(); // "Third"

        // Извлечение из стека
        while (stack.Count > 0)
        {
            string item = stack.Pop();
            Console.WriteLine($"Popped: {item}");
        }

        // Практический пример: отмена действий
        Stack<Action> undoStack = new Stack<Action>();

        // Пользователь выполняет действия
        undoStack.Push(() => Console.WriteLine("Undo: Delete text"));
        undoStack.Push(() => Console.WriteLine("Undo: Format document"));

        // Отмена последних действий
        if (undoStack.Count > 0)
        {
            undoStack.Pop()(); // Отмена форматирования
```

```
        }
    }
}
```

LinkedList - двусвязный список

```
public class LinkedListExamples
{
    public void DemonstrateLinkedList()
    {
        LinkedList<string> list = new LinkedList<string>();

        // Добавление элементов
        var firstNode = list.AddFirst("First");
        var lastNode = list.AddLast("Last");
        var middleNode = list.AddAfter(firstNode, "Middle");

        // Навигация по списку
        LinkedListNode<string> current = list.First;
        while (current != null)
        {
            Console.WriteLine(current.Value);
            current = current.Next;
        }

        // Быстрая вставка/удаление в любом месте
        list.AddBefore(middleNode, "Before Middle");
        list.Remove(lastNode);

        // Практический пример: кэш с ограниченным размером
        public class LRUCache<TKey, TValue>
        {
            private readonly int _capacity;
            private readonly Dictionary<TKey, LinkedListNode<(TKey Key, TValue
Value)>> _cache;
            private readonly LinkedList<(TKey Key, TValue Value)> _order;

            public LRUCache(int capacity)
            {
                _capacity = capacity;
                _cache = new Dictionary<TKey, LinkedListNode<(TKey, TValue)>>();
                _order = new LinkedList<(TKey, TValue)>();
            }

            public TValue Get(TKey key)
            {
                if (_cache.TryGetValue(key, out var node))
                {
                    // Перемещаем в начало (самый недавно использованный)
                    _order.Remove(node);
                    _order.AddFirst(node);
```

```
                    return node.Value.Value;
                }
                return default(TValue);
            }
        }
    }
}
```

---

5. Производительность операций с различными структурами данных

Сравнительная таблица сложности операций (Big O Notation):

Структура	Доступ	Поиск	Вставка	Удаление
Array	O(1)	O(n)	O(n)	O(n)
List	O(1)	O(n)	O(n)*	O(n)
Dictionary	O(1)	O(1)	O(1)	O(1)
HashSet	O(1)	O(1)	O(1)	O(1)
Queue	O(1)	O(n)	O(1)	O(1)
Stack	O(1)	O(n)	O(1)	O(1)
LinkedList	O(n)	O(n)	O(1)	O(1)

*Для List вставка в конец в среднем O(1), в худшем случае O(n) при увеличении Capacity

```
public class PerformanceBenchmark
{
    public void MeasurePerformance()
    {
        const int count = 100000;

        // Тест List<T>
        List<int> list = new List<int>();
        var stopwatch = System.Diagnostics.Stopwatch.StartNew();

        for (int i = 0; i < count; i++)
        {
            list.Add(i);
        }

        stopwatch.Stop();
        Console.WriteLine($"List.Add: {stopwatch.ElapsedMilliseconds} ms");

        // Тест Dictionary
        Dictionary<int, int> dict = new Dictionary<int, int>();
        stopwatch.Restart();
```

```
        for (int i = 0; i < count; i++)
        {
            dict[i] = i;
        }

        stopwatch.Stop();
        Console.WriteLine($"Dictionary.Add: {stopwatch.ElapsedMilliseconds} ms");

        // Тест поиска
        stopwatch.Restart();
        bool found = list.Contains(count / 2);
        stopwatch.Stop();
        Console.WriteLine($"List.Contains: {stopwatch.ElapsedMilliseconds} ms");

        stopwatch.Restart();
        found = dict.ContainsKey(count / 2);
        stopwatch.Stop();
        Console.WriteLine($"Dictionary.ContainsKey:
{stopwatch.ElapsedMilliseconds} ms");
    }
}
```

Факторы, влияющие на производительность:

- **Размер данных** - большие объемы требуют тщательного выбора структур
- **Паттерн доступа** - частые вставки, поиск или обход
- **Локализация данных** - массивы кэш-дружественны
- **Хеш-функция** - для словарей и множеств

---

6. Практические рекомендации по выбору структур данных

Критерии выбора:

1. По типу операций:

- **Частый поиск по ключу** → Dictionary, HashSet
- **Частые вставки/удаления в середину** → LinkedList
- **Последовательная обработка** → Queue, Stack
- **Индексный доступ** → Array, List

2. По требованиям к памяти:

- **Минимальные накладные расходы** → Array
- **Динамический размер** → List, Dictionary
- **Экономия при редких данных** → HashSet

3. По потоко-безопасности:

- **Concurrent коллекции** для многопоточности

```
public class BestPractices
{
    // Пример 1: Кэширование данных
    public class DataCache
    {
        // Для быстрого поиска по ID используем Dictionary
        private Dictionary<int, User> _userCache = new Dictionary<int, User>();

        // Для сохранения порядка добавления используем List
        private List<int> _recentlyAccessed = new List<int>();

        public User GetUser(int userId)
        {
            if (_userCache.TryGetValue(userId, out User user))
            {
                // Обновляем порядок доступа
                _recentlyAccessed.Remove(userId);
                _recentlyAccessed.Add(userId);
                return user;
            }
            return null;
        }
    }

    // Пример 2: Обработка команд
    public class CommandProcessor
    {
        // Очередь для обработки команд в порядке поступления
        private Queue<ICommand> _commandQueue = new Queue<ICommand>();

        // Стек для реализации отмены операций
        private Stack<ICommand> _undoStack = new Stack<ICommand>();

        public void ProcessCommands()
        {
            while (_commandQueue.Count > 0)
            {
                var command = _commandQueue.Dequeue();
                command.Execute();
                _undoStack.Push(command);
            }
        }

        public void UndoLastCommand()
        {
            if (_undoStack.Count > 0)
            {
                var command = _undoStack.Pop();
                command.Undo();
            }
        }
    }
```

```
    // Пример 3: Уникальные элементы с быстрым поиском
    public class UniqueItemRegistry
    {
        // HashSet для хранения уникальных элементов и быстрой проверки
существования
        private HashSet<string> _registeredItems = new HashSet<string>
(StringComparer.OrdinalIgnoreCase);

        // List для сохранения порядка регистрации
        private List<string> _registrationOrder = new List<string>();

        public bool RegisterItem(string item)
        {
            if (_registeredItems.Add(item)) // Добавляет только если элемента нет
            {
                _registrationOrder.Add(item);
                return true;
            }
            return false;
        }

        public IEnumerable<string> GetItemsInOrder()
        {
            return _registrationOrder.AsReadOnly();
        }
    }
}
```

Антипаттерны и распространенные ошибки:

```
public class CommonMistakes
{
    // ПЛОХО: Использование List для частого поиска
    public bool FindInListBad(List<string> list, string item)
    {
        return list.Contains(item); // O(n) - медленно для больших списков
    }

    // ХОРОШО: Использование HashSet для частого поиска
    public bool FindInHashSetGood(HashSet<string> set, string item)
    {
        return set.Contains(item); // O(1) - быстро
    }

    // ПЛОХО: Частая вставка в начало List
    public void InsertAtBeginningBad(List<int> list, int item)
    {
        list.Insert(0, item); // O(n) - все элементы сдвигаются
    }

    // ХОРОШО: Использование LinkedList для частых вставок в начало
```

```
    public void InsertAtBeginningGood(LinkedList<int> list, int item)
    {
        list.AddFirst(item); // O(1) - быстро
    }

    // ПЛОХО: Неправильная итерация с модификацией
    public void RemoveItemsBad(List<int> list)
    {
        foreach (var item in list) // Исключение!
        {
            if (item % 2 == 0)
                list.Remove(item); // Нельзя модифицировать коллекцию во время
итерации
        }
    }

    // ХОРОШО: Правильное удаление элементов
    public void RemoveItemsGood(List<int> list)
    {
        // Использование обратного цикла for
        for (int i = list.Count - 1; i >= 0; i--)
        {
            if (list[i] % 2 == 0)
                list.RemoveAt(i);
        }

        // Или с использованием RemoveAll
        list.RemoveAll(item => item % 2 == 0);
    }
}
```

Резюме лекции:

1. **Структуры данных** - фундаментальный инструмент программиста, правильный выбор критически важен для производительности
2. **Массивы** - лучший выбор когда известен фиксированный размер и нужен быстрый доступ по индексу
3. **List** - универсальная замена массивам с динамическим размером, но дорогая вставка в середину
4. **Dictionary/HashSet** - незаменимы для быстрого поиска, вставки и удаления по ключу
5. **Queue/Stack** - специализированные структуры для обработки данных в определенном порядке
6. **Производительность** - понимание сложности операций позволяет выбирать оптимальные структуры
7. **Практический выбор** должен основываться на анализе операций, которые будут выполняться чаще всего

Ключевой принцип: Выбирайте структуру данных исходя из операций, которые вы будете выполнять чаще всего, а не из удобства реализации. Правильный выбор структур данных - это 80% успеха в создании эффективных приложений.