

Коллекции и структуры данных

27.01.2024

Аналогичные данные часто можно обрабатывать более эффективно при хранении и управлении ими как коллекции. Вы можете использовать класс [System.Array](#) или классы в пространствах имен [System.Collections](#), [System.Collections.Generic](#), [System.Collections.Concurrent](#) и [System.Collections.Immutable](#) для добавления, удаления и изменения отдельных элементов или диапазона элементов в коллекции.

Существует два основных типа коллекций; универсальные коллекции и не универсальные коллекции. Универсальные коллекции являются типобезопасными во время компиляции. Из-за этого универсальные коллекции обычно обеспечивают более высокую производительность. Универсальные коллекции принимают параметр типа при их создании. Им не требуется приведение к типу [Object](#) при добавлении или удалении элементов из коллекции. Кроме того, большинство универсальных коллекций поддерживаются в приложениях Магазина Windows. Коллекции, не являющиеся универсальными, такие как [Object](#), требуют приведения типов, и большинство из них не поддерживаются для разработки приложений Windows Store. Однако в более старом коде могут отображаться не универсальные коллекции.

В .NET Framework 4 и более поздних версиях коллекции в [System.Collections.Concurrent](#) пространстве имен предоставляют эффективные потокобезопасные операции для доступа к элементам коллекции из нескольких потоков. Неизменяемые классы коллекций в [System.Collections.Immutable](#) пространстве имен ([накет NuGet](#)) по сути являются потокобезопасными, так как операции выполняются в копии исходной коллекции, а исходная коллекция не может быть изменена.

Общие функции коллекции

Все коллекции предоставляют методы для добавления, удаления или поиска элементов в коллекции. Кроме того, все коллекции, которые напрямую или косвенно реализуют [ICollection](#) интерфейс или [ICollection<T>](#) интерфейс совместно используют следующие функции:

- **Возможность перечисления коллекции**

Коллекции .NET либо реализуют [System.Collections.IEnumerable](#) , либо [System.Collections.Generic.IEnumerable<T>](#) позволяют выполнять итерацию коллекции. Перечислитель может рассматриваться как перемещаемый указатель на любой элемент в коллекции. The [foreach, in](#) оператор и [For Each...Next оператор](#) используют предоставляемый методом [GetEnumerator](#) перечислитель и упрощают управление перечислителем. Кроме того, любая коллекция, реализующая [System.Collections.Generic.IEnumerable<T>](#) , считается *запрашиваемым типом* и может запрашиваться с помощью LINQ. Запросы LINQ предоставляют общий шаблон для доступа к данным. Они обычно более краткие и читаемые, чем стандартные `foreach` циклы, и позволяют фильтровать, упорядочивать и группировать. Запросы LINQ также могут повысить производительность. Дополнительные сведения см. в

статьях [LINQ to Objects \(C#\)](#), [LINQ to Objects \(Visual Basic\)](#), [Parallel LINQ \(PLINQ\)](#), [Introduction to LINQ Querys \(C#\)](#) и [Basic Query Operations \(Visual Basic\)](#).

- **Возможность копирования содержимого коллекции в массив**

Все коллекции можно скопировать в массив с помощью `CopyTo` метода. Однако порядок элементов в новом массиве основан на последовательности, в которой перечислитель возвращает их. Результирующий массив всегда одномерный с нижней границей нуля.

Кроме того, многие классы коллекций содержат следующие функции:

- **Свойства вместимости и количества**

Емкость коллекции — это количество элементов, которые он может содержать. Количество коллекции — это число элементов, которые она действительно содержит. Некоторые коллекции скрывают объем, количество или то и другое.

Большинство коллекций автоматически расширяют емкость при достижении текущей емкости. Память перераспределена, и элементы копируются из старой коллекции в новую. Эта конструкция сокращает код, необходимый для использования коллекции. Однако на производительность коллекции может негативно сказаться. Например, для `List<T>`, если значение `Count` меньше `Capacity`, добавление элемента является операцией $O(1)$. Если емкость должна быть увеличена для размещения нового элемента, добавление элемента становится операцией $O(n)$, где n находится `Count`. Лучший способ избежать низкой производительности, вызванной несколькими перемещениями, заключается в том, чтобы задать начальную емкость для оценки размера коллекции.

Это `BitArray` особый случай, его емкость совпадает с его длиной, которая совпадает с его числом.

- **Согласованная нижняя граница**

Нижняя граница коллекции является индексом своего первого элемента. Все индексированные коллекции в `System.Collections` пространствах имен имеют нижнюю границу нуля, то есть они индексируются 0. `Array` имеет нижнюю границу, равную нулю, по умолчанию, но при создании экземпляра класса `Array` можно `Array.CreateInstance` определить другую нижнюю границу.

- **Синхронизация для доступа из нескольких потоков (`System.Collections` только для классов).**

Необобщенные типы коллекций в пространстве имен `System.Collections` обеспечивают некоторую безопасность потоков с помощью синхронизации; обычно это достигается через члены `SyncRoot` и `IsSynchronized`. Эти коллекции по умолчанию не являются потокобезопасными. Если требуется масштабируемый и эффективный многопоточковый доступ к коллекции, используйте один из классов в `System.Collections.Concurrent` пространстве имен или рассмотрите возможность использования неизменяемой коллекции. Дополнительные сведения см. в разделе [Thread-Safe Коллекции](#).

Выбор коллекции

Как правило, следует использовать универсальные коллекции. В следующей таблице описаны некоторые распространенные сценарии сбора и классы коллекций, которые можно использовать для этих сценариев. Если вы не знакомы с универсальными коллекциями, следующая таблица поможет выбрать универсальную коллекцию, которая лучше всего подходит для вашей задачи:


 Развернуть таблицу

Я хочу...	Параметры общей коллекции	Настройки нестандартной коллекции	Параметры потокобезопасных или неизменяемых коллекций
Хранение элементов в виде пар "ключ-значение" для быстрого поиска по ключу	<code>Dictionary<TKey,TValue></code>	<code>Hashtable</code> (Коллекция пар "ключ-значение", упорядоченных на основе хэш-кода ключа.)	<code>ConcurrentDictionary<TKey,TValue></code> <code>ReadOnlyDictionary<TKey,TValue></code> <code>ImmutableDictionary<TKey,TValue></code>
Доступ к элементам по индексу	<code>List<T></code>	<code>Array</code> <code>ArrayList</code>	<code>ImmutableList<T></code> <code>ImmutableArray</code>
Используйте метод очереди первым пришёл — первым вышел (FIFO)	<code>Queue<T></code>	<code>Queue</code>	<code>ConcurrentQueue<T></code> <code>ImmutableQueue<T></code>
Используйте данные Last-In-First-Out по принципу LIFO (последним пришел - первым ушел)	<code>Stack<T></code>	<code>Stack</code>	<code>ConcurrentStack<T></code> <code>ImmutableStack<T></code>
Доступ к элементам последовательно	<code>LinkedList<T></code>	Нет рекомендаций	Нет рекомендаций
Получение уведомлений при удалении или добавлении элементов в коллекцию. (реализует <code>INotifyPropertyChanged</code> и <code>INotifyCollectionChanged</code>)	<code>ObservableCollection<T></code>	Нет рекомендаций	Нет рекомендаций
Отсортированная коллекция	<code>SortedList<TKey,TValue></code>	<code>SortedList</code>	<code>ImmutableSortedDictionary<TKey,TValue></code> <code>ImmutableSortedSet<T></code>

Я хочу...	Параметры общей коллекции	Настройки нестандартной коллекции	Параметры потокобезопасных или неизменяемых коллекций
Набор математических функций	<code>HashSet<T></code>	Нет рекомендаций	<code>ImmutableHashSet<T></code>
	<code>SortedSet<T></code>		<code>ImmutableSortedSet<T></code>

Алгоритмическая сложность коллекций

При выборе [класса коллекции](#) стоит рассмотреть потенциальные компромиссы в производительности. Используйте следующую таблицу для ссылки на то, как различные типы изменяемых коллекций сравниваются в алгоритмической сложности с соответствующими неизменяемыми аналогами. Часто неизменяемые типы коллекций являются менее производительными, но обеспечивают неизменяемость - что часто является допустимым относительным преимуществом.

 [Развернуть таблицу](#)


Изменяемый	Амортизированный	Худший случай	Неизменный	Сложность
<code>Stack<T>.Push</code>	$O(1)$	$O(n)$	<code>ImmutableStack<T>.Push</code>	$O(1)$
<code>Queue<T>.Enqueue</code>	$O(1)$	$O(n)$	<code>ImmutableQueue<T>.Enqueue</code>	$O(1)$
<code>List<T>.Add</code>	$O(1)$	$O(n)$	<code>ImmutableList<T>.Add</code>	$O(\log n)$
<code>List<T>.Item[Int32]</code>	$O(1)$	$O(1)$	<code>ImmutableList<T>.Item[Int32]</code>	$O(\log n)$
<code>List<T>.Enumerator</code>	$O(n)$	$O(n)$	<code>ImmutableList<T>.Enumerator</code>	$O(n)$
<code>HashSet<T>.Add</code> Поиск	$O(1)$	$O(n)$	<code>ImmutableHashSet<T>.Add</code>	$O(\log n)$
<code>SortedSet<T>.Add</code>	$O(\log n)$	$O(n)$	<code>ImmutableSortedSet<T>.Add</code>	$O(\log n)$
<code>Dictionary<T>.Add</code>	$O(1)$	$O(n)$	<code>ImmutableDictionary<T>.Add</code>	$O(\log n)$
<code>Dictionary<T></code> Поиск	$O(1)$	$O(1)$ — или строго $O(n)$	<code>ImmutableDictionary<T></code> Поиск	$O(\log n)$
<code>SortedDictionary<T>.Add</code>	$O(\log n)$	$O(n \log n)$	<code>ImmutableSortedDictionary<T>.Add</code>	$O(\log n)$

Перечисление `List<T>` можно эффективно осуществить с помощью цикла `for` или цикла `foreach`. Однако `ImmutableList<T>`, выполняется плохо в цикле `for` из-за временной сложности $O(\log n)$ для своего индексатора. Перечисление `ImmutableList<T>` с помощью цикла `foreach` является эффективным, потому что `ImmutableList<T>` использует двоичное дерево для хранения своих данных, вместо массива, который использует `List<T>`. Массив можно быстро индексировать в, в то

время как двоичное дерево должно пройти вниз до тех пор, пока узел с нужным индексом не найден.

Кроме того, `SortedSet<T>` и `ImmutableSortedSet<T>` имеют одинаковую сложность, так как оба используют двоичные деревья. Существенное различие заключается в том, что `ImmutableSortedSet<T>` использует неизменяемое двоичное дерево. Так как `ImmutableSortedSet<T>` также предлагает [System.Collections.Immutable.ImmutableSortedSet<T>.Builder](#) класс, который позволяет мутации, вы можете иметь как неизменяемость, так и производительность.

Связанные статьи

 Развернуть таблицу

Название	Описание
Выбор класса коллекции	Описывает различные коллекции и помогает выбрать один из них для вашего сценария.
Часто используемые типы коллекций	Описывает часто используемые универсальные и не универсальные типы коллекций, такие как System.Array , System.Collections.Generic.List<T> и System.Collections.Generic.Dictionary<TKey,TValue> .
Когда следует использовать универсальные коллекции	Описывает использование универсальных типов коллекций.
Сравнения и сортировки в коллекциях	Описывает использование сравнений равенства и сортировки сравнений в коллекциях.
Отсортированные типы коллекций	Описывает производительность и характеристики отсортированных коллекций.
Типы коллекций хеш-таблиц и словарей данных	Описывает функции универсальных и не универсальных типов словарей на основе хэша.
Thread-Safe коллекций	Описывает типы коллекций, такие как System.Collections.Concurrent.BlockingCollection<T> и System.Collections.Concurrent.ConcurrentBag<T> поддерживающие безопасный и эффективный одновременный доступ из нескольких потоков.
System.Collections.Immutable	Представляет неизменяемые коллекции и предоставляет ссылки на типы коллекций.

Справка

- [System.Array](#)
- [System.Collections](#)
- [System.Collections.Concurrent](#)
- [System.Collections.Generic](#)
- [System.Collections.Specialized](#)

- [System.Linq](#)
- [System.Collections.Immutable](#)