# 下面的示例演示如何将元素归到不同的块区中，块区表示连续键的子序列。

## 例如，假设您有下面的键值对序列：

| **键** | **值** |
| --- | --- |
| A | We |
| A | think |
| A | that |
| B | Linq |
| C | is |
| A | really |
| B | cool |
| B | ! |

## 将按照下面的顺序创建以下各组：

1. We, think, that
2. Linq
3. is
4. really
5. cool, !

## 此解决方案作为扩展方法实现，扩展方法是线程安全的且以流的方式返回其结果。

也就是说，它在移动通过源序列的过程中生成自己的各个组。与 **group** 或 **orderby** 运算符不同，它可以在整个序列读取完毕之前就开始将组返回给调用方。

线程安全性是通过在迭代源序列的过程中创建每个组或块区的副本来实现的，如源代码注释中所述。如果源序列包含一个由连续项组成的很大的序列，则公共语言运行时可能会引发 [OutOfMemoryException](https://msdn.microsoft.com/zh-cn/library/system.outofmemoryexception.aspx)。

下面的示例显示了此扩展方法及使用它的客户端代码。

using System;

using System.Collections.Generic;

using System.Linq;

namespace ChunkIt

{

// Static class to contain the extension methods.

public static class MyExtensions

{

public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector)

{

return source.ChunkBy(keySelector, EqualityComparer<TKey>.Default);

}

public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector, IEqualityComparer<TKey> comparer)

{

// Flag to signal end of source sequence.

const bool noMoreSourceElements = true;

// Auto-generated iterator for the source array.

var enumerator = source.GetEnumerator();

// Move to the first element in the source sequence.

if (!enumerator.MoveNext()) yield break;

// Iterate through source sequence and create a copy of each Chunk.

// On each pass, the iterator advances to the first element of the next "Chunk"

// in the source sequence. This loop corresponds to the outer foreach loop that

// executes the query.

Chunk<TKey, TSource> current = null;

while (true)

{

// Get the key for the current Chunk. The source iterator will churn through

// the source sequence until it finds an element with a key that doesn't match.

var key = keySelector(enumerator.Current);

// Make a new Chunk (group) object that initially has one GroupItem, which is a copy of the current source element.

current = new Chunk<TKey, TSource>(key, enumerator, value => comparer.Equals(key, keySelector(value)));

// Return the Chunk. A Chunk is an IGrouping<TKey,TSource>, which is the return value of the ChunkBy method.

// At this point the Chunk only has the first element in its source sequence. The remaining elements will be

// returned only when the client code foreach's over this chunk. See Chunk.GetEnumerator for more info.

yield return current;

// Check to see whether (a) the chunk has made a copy of all its source elements or

// (b) the iterator has reached the end of the source sequence. If the caller uses an inner

// foreach loop to iterate the chunk items, and that loop ran to completion,

// then the Chunk.GetEnumerator method will already have made

// copies of all chunk items before we get here. If the Chunk.GetEnumerator loop did not

// enumerate all elements in the chunk, we need to do it here to avoid corrupting the iterator

// for clients that may be calling us on a separate thread.

if (current.CopyAllChunkElements() == noMoreSourceElements)

{

yield break;

}

}

}

// A Chunk is a contiguous group of one or more source elements that have the same key. A Chunk

// has a key and a list of ChunkItem objects, which are copies of the elements in the source sequence.

class Chunk<TKey, TSource> : IGrouping<TKey, TSource>

{

// INVARIANT: DoneCopyingChunk == true ||

// (predicate != null && predicate(enumerator.Current) && current.Value == enumerator.Current)

// A Chunk has a linked list of ChunkItems, which represent the elements in the current chunk. Each ChunkItem

// has a reference to the next ChunkItem in the list.

class ChunkItem

{

public ChunkItem(TSource value)

{

Value = value;

}

public readonly TSource Value;

public ChunkItem Next = null;

}

// The value that is used to determine matching elements

private readonly TKey key;

// Stores a reference to the enumerator for the source sequence

private IEnumerator<TSource> enumerator;

// A reference to the predicate that is used to compare keys.

private Func<TSource, bool> predicate;

// Stores the contents of the first source element that

// belongs with this chunk.

private readonly ChunkItem head;

// End of the list. It is repositioned each time a new

// ChunkItem is added.

private ChunkItem tail;

// Flag to indicate the source iterator has reached the end of the source sequence.

internal bool isLastSourceElement = false;

// Private object for thread syncronization

private object m\_Lock;

// REQUIRES: enumerator != null && predicate != null

public Chunk(TKey key, IEnumerator<TSource> enumerator, Func<TSource, bool> predicate)

{

this.key = key;

this.enumerator = enumerator;

this.predicate = predicate;

// A Chunk always contains at least one element.

head = new ChunkItem(enumerator.Current);

// The end and beginning are the same until the list contains > 1 elements.

tail = head;

m\_Lock = new object();

}

// Indicates that all chunk elements have been copied to the list of ChunkItems,

// and the source enumerator is either at the end, or else on an element with a new key.

// the tail of the linked list is set to null in the CopyNextChunkElement method if the

// key of the next element does not match the current chunk's key, or there are no more elements in the source.

private bool DoneCopyingChunk { get { return tail == null; } }

// Adds one ChunkItem to the current group

// REQUIRES: !DoneCopyingChunk && lock(this)

private void CopyNextChunkElement()

{

// Try to advance the iterator on the source sequence.

// If MoveNext returns false we are at the end, and isLastSourceElement is set to true

isLastSourceElement = !enumerator.MoveNext();

// If we are (a) at the end of the source, or (b) at the end of the current chunk

// then null out the enumerator and predicate for reuse with the next chunk.

if (isLastSourceElement || !predicate(enumerator.Current))

{

enumerator = null;

predicate = null;

}

else

{

tail.Next = new ChunkItem(enumerator.Current);

}

// tail will be null if we are at the end of the chunk elements

// This check is made in DoneCopyingChunk.

tail = tail.Next;

}

// Called after the end of the last chunk was reached. It first checks whether

// there are more elements in the source sequence. If there are, it

// Returns true if enumerator for this chunk was exhausted.

internal bool CopyAllChunkElements()

{

while (true)

{

lock (m\_Lock)

{

if (DoneCopyingChunk)

{

// If isLastSourceElement is false,

// it signals to the outer iterator

// to continue iterating.

return isLastSourceElement;

}

else

{

CopyNextChunkElement();

}

}

}

}

public TKey Key { get { return key; } }

// Invoked by the inner foreach loop. This method stays just one step ahead

// of the client requests. It adds the next element of the chunk only after

// the clients requests the last element in the list so far.

public IEnumerator<TSource> GetEnumerator()

{

//Specify the initial element to enumerate.

ChunkItem current = head;

// There should always be at least one ChunkItem in a Chunk.

while (current != null)

{

// Yield the current item in the list.

yield return current.Value;

// Copy the next item from the source sequence,

// if we are at the end of our local list.

lock (m\_Lock)

{

if (current == tail)

{

CopyNextChunkElement();

}

}

// Move to the next ChunkItem in the list.

current = current.Next;

}

}

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

}

}

// A simple named type is used for easier viewing in the debugger. Anonymous types

// work just as well with the ChunkBy operator.

public class KeyValPair

{

public string Key { get; set; }

public string Value { get; set; }

}

class Program

{

// The source sequence.

public static IEnumerable<KeyValPair> list;

// Query variable declared as class member to be available

// on different threads.

static IEnumerable<IGrouping<string, KeyValPair>> query;

static void Main(string[] args)

{

// Initialize the source sequence with an array initializer.

list = new[]

{

new KeyValPair{ Key = "A", Value = "We" },

new KeyValPair{ Key = "A", Value = "Think" },

new KeyValPair{ Key = "A", Value = "That" },

new KeyValPair{ Key = "B", Value = "Linq" },

new KeyValPair{ Key = "C", Value = "Is" },

new KeyValPair{ Key = "A", Value = "Really" },

new KeyValPair{ Key = "B", Value = "Cool" },

new KeyValPair{ Key = "B", Value = "!" }

};

// Create the query by using our user-defined query operator.

query = list.ChunkBy(p => p.Key);

// ChunkBy returns IGrouping objects, therefore a nested

// foreach loop is required to access the elements in each "chunk".

foreach (var item in query)

{

Console.WriteLine("Group key = {0}", item.Key);

foreach (var inner in item)

{

Console.WriteLine("\t{0}", inner.Value);

}

}

Console.WriteLine("Press any key to exit");

Console.ReadKey();

}

}

}

若要在您的项目中使用此扩展方法，请将 MyExtensions 静态类复制到一个新的或现有源代码文件中，并在必要时，为该类所在的命名空间添加一条 **using** 指令。

## 参考

<https://learn.microsoft.com/en-us/dotnet/csharp/linq/get-started/introduction-to-linq-queries?redirectedfrom=MSDN>