MSK DOT NET

Span, Memory and Pipelines, the APIs you always missed

Microsoft Most Valuable Professional

Raffaele Rialdi - Senior Software Architect

@raffaeler
raffaeler@vevy.com

Who am I?

- Raffaele Rialdi, Senior Software Architect in Vevy Europe – Italy
  - @raffaeler also known as "Raf"
- Consultant in many industries
  - Manufacturing, racing, healthcare, financial, …
- Speaker and Trainer around the globe (development and security)
  - Italy, Romania, Bulgaria, Russia (Moscow, St Petersburg and Novosibirsk), USA, …
- And proud member of the great Microsoft MVP family since 2003

Agenda

- Using value type by reference is the key
- Our new best friends: Span<T> and Memory<T>
- Going unsafe
- The new memory allocation primitives
- Pipelines: a better way to manage streams of data
- Realtime processing with Span, Memory and Pipelines

A long dated problem

- Value-based vs Reference-based languages
- .NET is value-based but splits the type system in value and reference types

	Reference Types	Value Types
Allocation	heap (GC involved) ⚠	stack (no GC)
What is copied	just the reference	the whole data ⚠

- C# expanded the ability to work with references on value types
  - Started with C#7 and continued in C#8

Less GC, more performance, still safe

- C# 7.x widened the reference paradigm to avoid copies
  - the "in" modifier, meaning "readonly ref"
  - using "ref" when returning values
  - declaring local "ref" or "readonly ref" local variables

- The new «ref struct» and «readonly ref struct» ensure at compile time that instances <u>will only live on the stack</u> (no GC involved)

- Using ref is like viewing memory without owning it
  - You can both read and write it, provided it is not readonly

```
public readonly ref partial struct Span<T>
```

Span<T> and ReadOnlySpan<T>

- They are both "ref readonly struct"
  - The compiler ensure they only live on the stack, not hitting the GC at all
- It is a "view" over a contiguous region of memory
  - Every change on the view is effectively made on the memory being viewed

```
Span<byte> span = new byte[]
     { 0, 2, 4, 6, 8, 10, 12, 14, 16 }.AsSpan();
int total = 0;
foreach (byte item in span.Slice(3, 5))
    total += item;
Debug.Assert(total == 50);
```

```
string hello = "Hello, world!";
ReadOnlySpan<char> span1 = hello;
ReadOnlySpan<char> span2 = span1.Slice(7, 5);

Debug.Assert(span2.ToString() == "world");
Debug.Assert(span2 != "world");
```

- Designed to easily wrap any array

0	1	2	3	4	5	6	7	8	9	A	B	C	D
	0		1		2		3		4		5		6
00	48	00	65	00	6C	00	6C	00	6F	00	20	00	2C
	H		e		l		l		o				,

Example: A no-GC version of the string.Trim()

```
string test = "   Hello, World! ";
Trim(test).ToArray()
```

Span<T>

- is allocated **on the stack**
- cannot be stored as a class member
- does not involve any heap allocation
- does not impact on **GC**
- is a view on **managed** or **native** memory

An immutable view over a string

```
ReadOnlySpan<char> Trim(ReadOnlySpan<char> source)
{
    if (source.IsEmpty) return source;
    int start = 0, end = source.Length - 1;
    char startChar = source[start]
    char endChar = source[end];
    while ((start < end) &&
           (startChar == ' ' || endChar == ' '))
    {
        if (startChar == ' ') start++;
        if (endChar == ' ') end--;
        startChar = source[start];
        endChar = source[end];
    }
    return source.Slice(start, end - start + 1);
}
```

A new immutable view over a string

Span<T> limitations

- ref struct are allowed only in ref structs
- can't declare ref struct in async methods, but … look at the example!
- As it is a ref struct, can't survive the stack unwind in local functions

```
private async Task SomeAsyncFunc()
{
    var memory = new Memory<byte>(new byte[100]);
    await Task.Delay(1);

    // Not allowed in async methods
    //var span = memory.Span;
    //ref var a = ref MyLocalFunc1();

    MyLocalFunction() = 99;
    ref byte MyLocalFunction() => ref memory.Span[1];
}
```

```
public class SomeClass
{
    Span<byte> span;
}
```

```
public ref struct SomeStruct
{
    Span<byte> span;
}
```

Memory<T>

```
[DebuggerTypeProxy(typeof(MemoryDebugView<>))]
[DebuggerDisplay("{ToString(),raw}")]
public readonly struct Memory<T>
```

- Wraps a contiguous block of memory by <u>holding a reference</u> to it
  - It is <u>**not**</u> a ref struct and can survive stack unwind
  - The ***Span property*** expose a "view" of the memory hold by Memory<T>
  - Span<T> can't be converted in Memory<T> (a <u>**copy**</u> is needed)
- Rich extension methods provided in the box
  - AsSpan, AsMemory, BinarySearch, IndexOf, LastIndexOf, ...

```
var m1 = new Memory<byte>();
Debug.Assert(m1.IsEmpty);

ReadOnlyMemory<char> memStr =
    "Hello, world".AsMemory();
```

```
var blob = new byte[100];

var m2 = new Memory<byte>(blob);
var m3 = new Memory<byte>(blob, start: 10, length: 5);
var m4 = blob.AsMemory();
var m5 = blob.AsMemory(start:10);
var m6 = blob.AsMemory(start:10, length:5);
```

Span<T> on strings benchmark

- Using Benchmark.NET to measure trimming `"   Hello, world      "`

Method	Loop	Mean	Error	StdDev	Gen 0/1k Op	Allocated Memory/Op
StringTrim	1000	23.25 us	0.4561 us	0.4684 us	13.3362	56000 B
SpanTrim	1000	16.44 us	0.3164 us	0.4001 us	-	-

```
[Benchmark]
public void StringTrim()
{
    for(int i=0; i<Loop; i++)
    {
        string res = Text.Trim();
    }
}
```

```
[Benchmark]
public void SpanTrim()
{
  ReadOnlySpan<char> span = Text;
  for (int i = 0; i < Loop; i++)
  {
      ReadOnlySpan<char> res = span.Trim();
  }
}
```

Span<T> and Unsafe

Span<T> and pointers

- Span<T> can be used on unsafe, classic pointers (byte *, …)
  - Unsafe code is limited to construction, the rest is safe!
- Get some raw pointer

```
           byte* ptr = _native.ReadUnsafe();
```

```
Span<byte> spanByte = _native.ReadUnsafe();
```

- Build a Span<byte>

```
Span<byte> spanByte = new Span<byte>(ptr, sizeof(WavHeader));
```

- Or a Span<WavHeader>

```
Span<WavHeader> spanHeader = new Span<WavHeader>(ptr, 1);
```

- We just *casted a managed struct to native memory allocation*

MemoryMarshal and Unsafe helper classes

- Casting a Span<byte> to a Span<T>

```
Span<WavHeader> spanHeader = MemoryMarshal.Cast<byte, WavHeader>(spanByte);
```

- Materializing an instance of T

```
WavHeader wavheader = MemoryMarshal.Read<WavHeader>(spanByte);
```

```
WavHeader wavheader = Unsafe.Read<WavHeader>(ptr);
```

- Avoid materialization getting just a reference to T

```
ref WavHeader refWavHeader = ref MemoryMarshal.GetReference<WavHeader>(spanHeader);
```

```
ref WavHeader refwavheader = ref Unsafe.AsRef<WavHeader>(ptr);
```

NetCore source code for AsRef

```
.maxstack 1
ldarg.0
ret
```

Introducing new
memory management APIs

In the beginning …

- … we had classic allocation

```
byte[] blob = new byte[_size];

Memory<byte> memory = blob;
```

  - Memory<byte> can encapsulate and manage the ownership

- Or we could allocate on the stack using unsafe

```
byte* ptr = stackalloc byte[size];
```

ArrayPool

- ArrayPool<T> allows renting and returning chunks of memory

```
byte[] blob = ArrayPool<byte>.Shared.Rent(size);
```

- Be careful on the returned size!

```
Debug.Assert(blob2a.Length >= _size);
```

- Be careful to **return** the rented buffer

```
ArrayPool<byte>.Shared.Return(blob, clearArray:false);
```

- Instead of the standard pool, we can create new ones

```
var mypool = ArrayPool<byte>.Create(
      maxArrayLength: 1024,
      maxArraysPerBucket: 10);
```

MemoryPool

- MemoryPool<T> is similar but supports the disposable pattern

```
using (IMemoryOwner<byte> blob = MemoryPool<byte>.Shared.Rent(size))
{
    Debug.Assert(blob.Memory.Length != size);
    // slicing is a good way to obtain the exact buffer size
    Memory<byte> memory = blob.Memory.Slice(0, size);
}
```

- You can create a custom pool by deriving MemoryPool<T>
  - Example on GitHub: ArrayMemoryPool<T>

Allocating on the stack

- With Span<T>, stackalloc does not require unsafe code anymore

```
Span<byte> blob = stackalloc byte[] { 0, 1, 2, 3, 4, 5 };
```

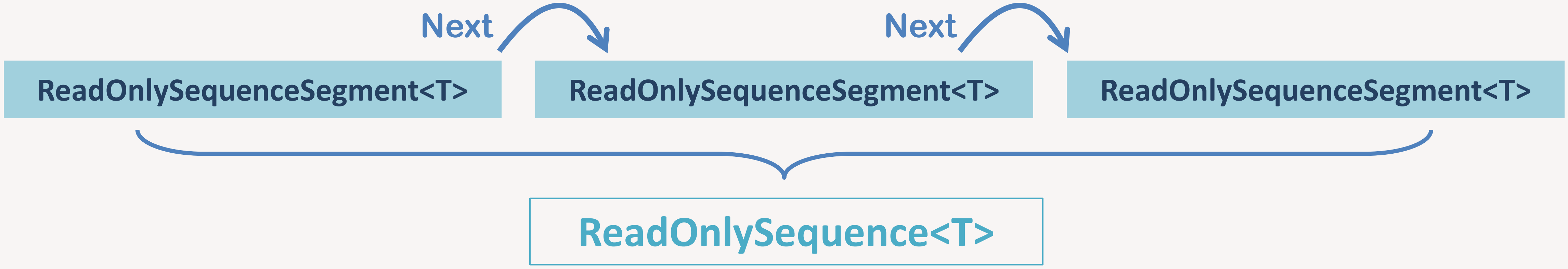
stackalloc initializer

- Avoid large buffers on the stack
  - C# stack default size is 1 MB

- This is the fastest possible allocation method for temporary buffers

ReadOnlySequence<T>

- ReadOnlySequence<T> is a linked list of memory segments/chunks
  - Each segment is made of contiguous memory
  - Segments are not (necessarily) contiguous in memory
  - Segments are exposed via Enumerator (do not implement IEnumerable<T>)
- Segments are anything deriving from ReadOnlySequenceSegment<T>
  - ReadOnlySequenceSegment<T> is abstract
  - There is no public concrete class available in corefx



The Pipeline API

Pipeline API

- Think to it as a modern Stream API
  - Conceptually mimes an (in-process) FIFO queue
  - Decouples readers from writers
  - Provides a built-in memory management for buffers
  - Leverages the power of:
    - Span<T>, Memory<T>, MemoryPool<T> and ReadOnlySequence<T>
  - Readers may decide to consume only a portion of the available buffer

- The content of the stream is always "bytes"

Writing a Pipe

- Strategy 1
  - The Pipe uses a private Pool to rent segments of memory
  - FlushAsync makes data available to the reader
  - Memory is automatically returned as soon as the data is consumed

- Strategy 2
  - The Pipe Writes an arbitrary blob of memory asynchronously

① *GetMemory strategy*

```
var pipe = new Pipe();

Memory<byte> mem = pipe.Writer.GetMemory(minSize)
int written = Encoding.UTF8.GetBytes("message",
                                      mem.Span);
pipe.Writer.Advance(written);
await pipe.Writer.FlushAsync();

pipe.Writer.Complete();
```

② *WriteAsync strategy*

```
var pipe = new Pipe();

Memory<byte> mem = Encoding.UTF8.GetBytes("message")
                                      .AsMemory();
var writeResult = await pipe.Writer.WriteAsync(mem);

pipe.Writer.Complete();
```

Reading a Pipe

- Strategy 1
  - Usually used inside an infinite loop
  - The async call is ended by completing the writer
- Strategy 2
  - Used only when you need the current content (if any) without waiting
- The buffer is always a ReadOnlySequence<byte>

① *Asynchronous*

```
var result = await pipe.Reader.ReadAsync();
var buffer = result.Buffer;
if (result.IsCanceled || buffer.IsEmpty) {
    // exit
}
```

② *Synchronous*

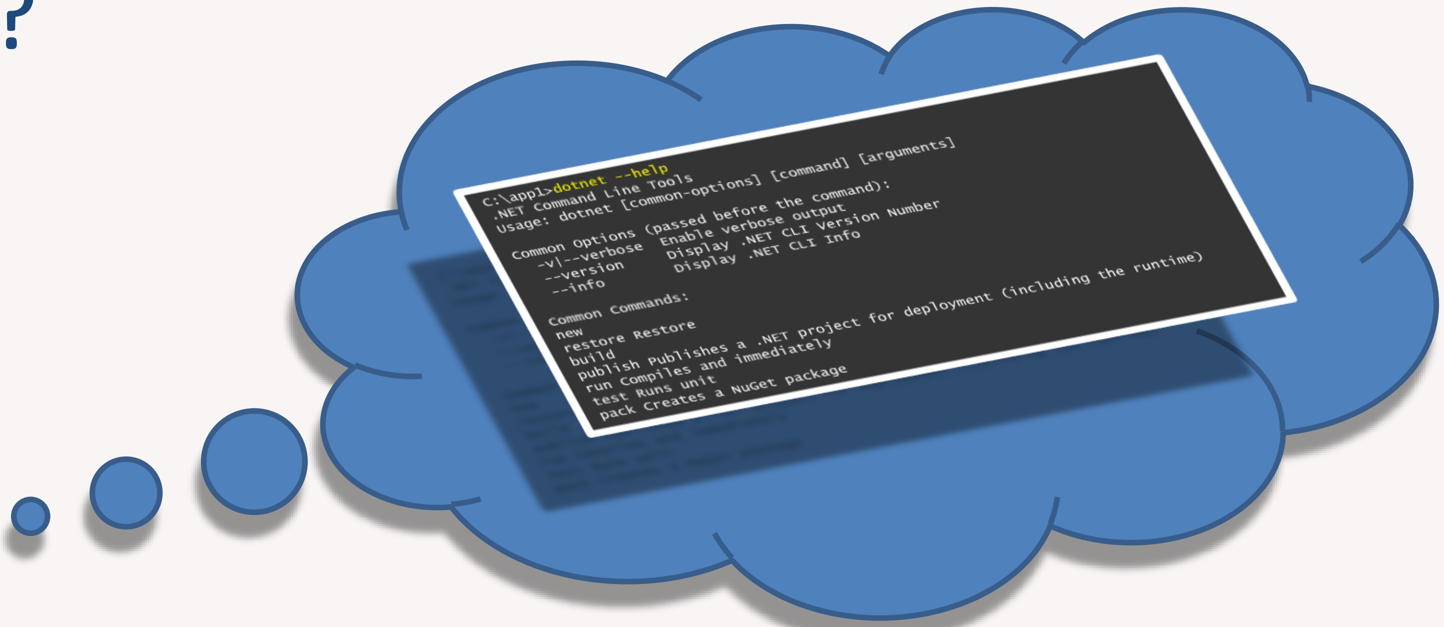
```
if(!reader.TryRead(out ReadResult result) ||
 result.IsCanceled || result.Buffer.IsEmpty) {
 // exit
}

var buffer = result.Buffer;
```

Demo on Pipelines:
read a process stdout stream

To sum up

- .NET Core is finally mature and offers modern APIs
- Try moving the hot paths from the GC to the stack with Span<T>
- Slice buffers using Span<T>
- Use memory pools to minimize the GC cost on reusable buffers
- Evaluate replacing System.IO.Stream with Pipelines

Questions?



Questions @ booth 1

Thank you!