Про .NET
Производительность

Numerics

DateTime/DateTimeOffset

```
[Benchmark]
public DateTimeOffset GetFutureTime() // -81% (2.0846 ns → 0.4045 ns)
    => _dt + _ts;

private DateTime _dt = DateTime.UtcNow;
private TimeSpan _ts = TimeSpan.FromMinutes(25);
```

```
using System;
public class C {

    ulong Foo1(int x)
    {
        if ((uint)x < 10)
        {
            return (ulong)x << 60;
        }
        GC.KeepAlive(this);
        return 0;
    }

    ulong Foo2(int x)
    {
        if ((uint)x < 10)
        {
            return (ulong)(uint)x << 60;
        }
        GC.KeepAlive(this);
        return 0;
    }
}
```

```
; Core CLR 9.0.24.52809 on x64

C..ctor()
    L0000: ret

C.Foo1(Int32)
    L0000: cmp edx, 0xa
    L0003: jae short L000d
    L0005: movsxd rax, edx
    L0008: shl rax, 0x3c
    L000c: ret
    L000d: xor eax, eax
    L000f: ret

C.Foo2(Int32)
    L0000: cmp edx, 0xa
    L0003: jae short L000c
    L0005: mov eax, edx
    L0007: shl rax, 0x3c
    L000b: ret
    L000c: xor eax, eax
    L000e: ret
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static uint Div96By32(ref Buf12 bufNum, uint den)
        {
            if (X86.X86Base.IsSupported)
            {
                uint remainder = 0;

                if (bufNum.U2 != 0)
                    goto Div3Word;
                if (bufNum.U1 >= den)
                    goto Div2Word;

                remainder = bufNum.U1;
                bufNum.U1 = 0;
                goto Div1Word;
Div3Word:
                (bufNum.U2, remainder) = X86.X86Base.DivRem(bufNum.U2, remainder, den);
Div2Word:
                (bufNum.U1, remainder) = X86.X86Base.DivRem(bufNum.U1, remainder, den);
Div1Word:
                (bufNum.U0, remainder) = X86.X86Base.DivRem(bufNum.U0, remainder, den);
                return remainder;
            }
```

decimal/UInt128

```
[Benchmark]
public decimal DivideDecimal() // -27% (10.666 ns → 7.761 ns)
    ⇒ _n_decimal / _d_decimal;

[Benchmark]
public UInt128 DivideUInt128() // -3% (10.885 ns → 10.612 ns)
    ⇒ _n_uint128 / _d_uint128;

private decimal _n_decimal = 9.87654321m;
private decimal _d_decimal = 1.23456789m;

private UInt128 _n_uint128 = new UInt128(123, 456);
private UInt128 _d_uint128 = new UInt128(0, 789);
```

Collections

```
372             public bool MoveNext()
373             {
374 -               bool retval;
375 -               if (_version != _stack._version) throw new
    InvalidOperationException(SR.InvalidOperation_EnumFailedVersion);
376 -               if (_index == -2)
377 -               {  // First call to enumerator.
378 -                   _index = _stack._size - 1;
379 -                   retval = (_index >= 0);
380 -                   if (retval)
381 -                       _currentElement = _stack._array[_index];
382 -                   return retval;
383 -               }
384 -               if (_index == -1)
385 -               {  // End of enumeration.
386 -                   return false;
387                 }
388
389 -               retval = (--_index >= 0);
390 -               if (retval)
391 -                   _currentElement = _stack._array[_index];
392 -               else
393 -                   _currentElement = default;
394 -               return retval;
395 -           }
```

```
public bool MoveNext()
{
    if (_version != _stack._version)
    {
        ThrowInvalidVersion();
    }

    T[] array = _stack._array;
    int index = _index - 1;
    if ((uint)index < (uint)array.Length)
    {
        Debug.Assert(index < _stack.Count);
        _currentElement = array[index];
        _index = index;
        return true;
    }

    _currentElement = default;
    _index = -1;
    return false;
}
```

Stack<T>

```
public int SumDirect() // -77% (12.636 ns → 2.964 ns)
{
    int sum = 0;
    foreach (int item in _direct) sum += item;
    return sum;
}

public int SumEnumerable() // -65% (15.408 ns → 5.339 ns)
{
    int sum = 0;
    foreach (int item in _enumerable) sum += item;
    return sum;
}

private Stack<int> _direct = new Stack<int>(Enumerable.Range(0, 10));
private IEnumerable<int> _enumerable = new Stack<int>(Enumerable.Range(0, 10));
```

```
443             public bool MoveNext()
444             {
445 -               if (_version != _q._version)
    ThrowHelper.ThrowInvalidOperationException_InvalidOperation_EnumFailedVersio
446 -
447 -               if (_index == -2)
448 -                   return false;
449 -
450 -               _index++;
451 -
452 -               if (_index == _q._size)
453                 {
454 -                   // We've run past the last element
455 -                   _index = -2;
456 -                   _currentElement = default;
457 -                   return false;
458                 }
459
460 -               // Cache some fields in locals to decrease code size
461 -               T[] array = _q._array;
462 -               uint capacity = (uint)array.Length;
463
464 -               // _index represents the 0-based index into the queue, howev
465 -               // doesn't have to start from 0 and it may not even be store
466 -
467 -               uint arrayIndex = (uint)(_q._head + _index); // this is the
    backing array
468 -               if (arrayIndex >= capacity)
469                 {
470 -                   // NOTE: Originally we were using the modulo operator he
471 -                   // on Intel processors it has a very high instruction la
472 -                   // was slowing down the loop quite a bit.
473 -                   // Replacing it with simple comparison/subtraction opera
474 -                   // the average foreach loop by 2x.
475 -
476 -                   arrayIndex -= capacity; // wrap around if needed
477 -               }
478
479 -               _currentElement = array[arrayIndex];
480 -               return true;
481 -           }
```

```
public bool MoveNext()
{
    if (_version != _queue._version)
    {
        ThrowHelper.ThrowInvalidOperationException_InvalidOperation_EnumFailedVersion();
    }

    Queue<T> q = _queue;
    int size = q._size;

    int offset = _i + 1;
    if ((uint)offset < (uint)size)
    {
        _i = offset;

        T[] array = q._array;
        int index = q._head + offset;
        if ((uint)index < (uint)array.Length)
        {
            _currentElement = array[index];
        }
        else
        {
            // The index has wrapped around the end of the array. Shift the index and then
            // get the current element. It is tempting to dedup this dereferencing with that
            // in the if block above, but the if block above avoids a bounds check for the
            // accesses that are in that portion, whereas these still incur it.
            index -= array.Length;
            _currentElement = array[index];
        }

        return true;
    }

    _i = -2;
    _currentElement = default;
    return false;
}
```

Queue<T>

```
public int SumDirect() // -67% (17.428 ns → 5.812 ns)
{
    int sum = 0;
    foreach (int item in _direct) sum += item;
    return sum;
}

public int SumEnumerable() // -72% (18.224 ns → 5.083 ns)
{
    int sum = 0;
    foreach (int item in _enumerable) sum += item;
    return sum;
}

[GlobalSetup]
public void Setup()
{
    _direct = new Queue<int>(Enumerable.Range(0, 10));
    for (int i = 0; i < 5; i++)
      _direct.Enqueue(_direct.Dequeue());
      _enumerable = _direct;
}

private Queue<int> _direct;
private IEnumerable<int> _enumerable;
```

```
903             public bool MoveNext()
904             {
905 -               switch (_state)
906 -               {
907 -                   case StateUninitialized:
908 -                       _buckets = _dictionary._tables._buckets;
909 -                       _i = -1;
910 -                       goto case StateOuterloop;
911 -
912 -                   case StateOuterloop:
913 -                       ConcurrentDictionary<TKey, TValue>.VolatileNode[]? buckets = _buckets;
914 -                       Debug.Assert(buckets is not null);
915 -
916 -                       int i = ++_i;
917 -                       if ((uint)i < (uint)buckets.Length)
918 -                       {
919 -                           _node = buckets[i]._node;
920 -                           _state = StateInnerLoop;
921 -                           goto case StateInnerLoop;
922 -                       }
923 -                       goto default;
924
925 -                   case StateInnerLoop:
926 -                       if (_node is Node node)
927 -                       {
928 -                           Current = new KeyValuePair<TKey, TValue>(node._key, node._value);
929 -                           _node = node._next;
930 -                           return true;
931 -                       }
932 -                       goto case StateOuterloop;
933
934 -                   default:
935 -                       _state = StateDone;
936                         return false;
937             }
938         }
939     }
```

```
890             public bool MoveNext()
891             {
                    Comment on line R891 Resolved
892 +               while (true)
893 +               {
894 +                   if (_node is Node node)
895 +                   {
896 +                       Current = new KeyValuePair<TKey, TValue>(node._key, node._value);
897 +                       _node = node._next;
898 +                       return true;
899 +                   }
900
901 +                   ConcurrentDictionary<TKey, TValue>.VolatileNode[]? buckets = _buckets ??= _dictionary._tables._buckets;
902 +                   Debug.Assert(buckets is not null);
903
904 +                   int i = _i + 1;
905 +                   if ((uint)i >= (uint)buckets.Length)
906 +                   {
907                         return false;
908 +                   }
909 +
910 +                   _node = buckets[i]._node;
911 +                   _i = i;
912 +               }
913 +           }
914 +       }
```

ConcurrentDictionary<K,V>

```
public int EnumerateInts() // -83% (3,432.3 ns → 594.9 ns)
{
    int sum = 0;
    foreach (var kvp in _ints) sum += kvp.Value;
    return sum;
}

public int EnumerateStrings() // -84% (8,306.4 ns → 1,309.1 ns)
{
    int sum = 0;
    foreach (var kvp in _strings) sum += kvp.Value.Length;
    return sum;
}

private ConcurrentDictionary<int, int> _ints
    = new ConcurrentDictionary<int, int>(
        Enumerable.Range(0, 1000).ToDictionary(i => i, i => i));
private ConcurrentDictionary<string, string> _strings
    = new ConcurrentDictionary<string, string>(
        Enumerable.Range(0, 1000).ToDictionary(i => i.ToString(), i => i.ToString()));
```

ConcurrentDictionary<K,V>

```
[Benchmark]
public void ClearAndAdd() // -54% (104.87 us → 48.16 us) | -66% (585.66 KB → 198.09 KB)
{
    _data.Clear();
    for (int i = 0; i < 4096; i++)
        _data.TryAdd(i, i);
}

private ConcurrentDictionary<int, int> _data = new(concurrencyLevel: 1, capacity: 4096);
```

```
412 +                   // TODO: Replace with just key.GetHashCode once https://github.com/dotnet/runtime/issues/117521 is resolved.
413 +                   uint hashCode = (uint)EqualityComparer<TKey>.Default.GetHashCode(key);
```

> Comment on line **R413** Resolved

> Comment on line **R413** Resolved

```
414                     int i = GetBucket(hashCode);
415                     Entry[]? entries = _entries;
416                     uint collisionCount = 0;
```

+1 -1 ☐ Viewed ···

```
112             {
113             }
114
115 +           public override bool Equals(string? x, string? y) => string.Equals(x, y, StringComparison.OrdinalIgnoreCase);
116
117             public override int GetHashCode(string? obj)
118             {
```

Dictionary<string, T>

```
public int Get() // -43% (47.81 ns → 27.18 ns)
  ⇒  _data["f98XAGjWBmEFWKPGX3paQ96hpY3i9u"] +
     _data["D1NG4i5T1oSWV1q114e3uvTNc3N9Sn"] +
     _data["B2Ewzn8tBJZZi8bPtSMxTYwo871Bwi"] +
     _data["fzJwk2Lf3gXNLDcecKf68qHyM1aMAB"] +
     _data["avv4rMrP7y8tWy2ZfK8Ck79TPZyiHz"];

[GlobalSetup]
public void Setup() {
    var generator = new Random(42);
    for (int i = 0; i < Count; i++) {
        var key = new string(generator.GetItems<char>(Base58, Length));
        _data.Add(key, i);
    }
}

[Params(30)] public int Length { get; set; }
[Params(1000)] public int Count { get; set; }
private const string Base58 = "123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz";
private Dictionary<string, int> _data = [];
```

/FrozenDictionary.cs

Viewed

ionary<TKey, TValue>

```
122                 // the Equals/GetHashCode methods to be devirtualized and possibly
    inlined.
123                 if (typeof(TKey).IsValueType && ReferenceEquals(comparer,
    EqualityComparer<TKey>.Default))
124                 {
125 + #if NET
126 +                   if (DenseIntegralFrozenDictionary.TryCreate(source, out
      FrozenDictionary<TKey, TValue>? result))
127 +                   {
128 +                       return result;
129 +                   }
130 + #endif
131 +
132                     if (source.Count <=
    Constants.MaxItemsInSmallValueTypeFrozenCollection)
133                     {
134                         // If the key is a something we know we can efficiently
    compare, use a specialized implementation
```

FrozenDictionary<K,V>

```
[Arguments(HttpStatusCode.OK)]
public string Get(HttpStatusCode status) // -82% (1.5585 ns → 0.2779 ns)
  ⇒ s_statusDescriptions[status];

private static readonly FrozenDictionary<HttpStatusCode, string> s_statusDescriptions =
    Enum.GetValues<HttpStatusCode>().Distinct()
        .ToFrozenDictionary(status ⇒ status, status ⇒ status.ToString());
```

LINQ

```
152 +
153 +            public override bool Contains(TSource value)
154 +            {
155 +                Func<TSource, bool> predicate = _predicate;
156 +
157 +                foreach (TSource item in _source)
158 +                {
159 +                    if (predicate(item) && EqualityComparer<TSource>.Default.Equals(item, value))
160 +                    {
161 +                        return true;
162 +                    }
163 +                }
164 +
165 +                return false;
166 +            }
```

Contains

```
public bool AppendContains() // -96% (1,960.48 ns → 82.81 ns) | 88 B → 56 B (0.64)
  ⇒ _source.Append(100).Contains(999);
public bool ConcatContains() // -96% (1,969.36 ns → 83.63 ns) | 88 B → 56 B (0.64)
  ⇒ _source.Concat(_source).Contains(999);
public bool DistinctContains() // -99% (8,925.57 ns → 85.67 ns) | 58656 B → 64 B (0.001)
  ⇒ _source.Distinct().Contains(999);
public bool OrderByContains() // -99% (8,406.01 ns → 84.50 ns) | 12280 B → 88 B (0.007)
  ⇒ _source.OrderBy(x ⇒ x).Contains(999);
public bool ReverseContains() // -50% (171.65 ns → 85.45 ns) | 4072 B → 48 B (0.01)
  ⇒ _source.Reverse().Contains(999);
public bool UnionContains() // -99% (8,973.47 ns → 83.49 ns) | 58664 B → 72 B (0.001)
  ⇒ _source.Union(_source).Contains(999);
public bool SelectManyContains() // -95% (1,978.40 ns → 91.69 ns) | 192 B → 128 B (0.67)
  ⇒ _source.SelectMany(x ⇒ _source).Contains(999);
public bool WhereSelectContains() // -85% (1,732.03 ns → 258.25 ns) | 104 B → 104 B (1.00)
  ⇒ _source.Where(x ⇒ true).Select(x ⇒ x).Contains(999);
private IEnumerable<int> _source = Enumerable.Range(0, 1000).ToArray();
```

```
using IEnumerator<TSource> e = source.GetEnumerator();
if (e.MoveNext())
{
    // Fill the reservoir with the first takeCount elements from the source.
    // If we can't fill it, just return what we get.
    reservoir = new List<TSource>(Math.Min(takeCount, 4)) { e.Current };
    while (reservoir.Count < takeCount)
    {
        if (!e.MoveNext())
        {
            totalElementCount = reservoir.Count;
            goto ReturnReservoir;
        }

        reservoir.Add(e.Current);
    }

    // For each subsequent element in the source, randomly replace an element in the
    // reservoir with a decreasing probability.
    long i = takeCount;
    while (e.MoveNext())
    {
        i++;
        long r = Random.Shared.NextInt64(i);
        if (r < takeCount)
        {
            reservoir[(int)r] = e.Current;
        }
    }

    totalElementCount = i;
}
```

Shuffle.Take

```
// +3% (2.783 us → 2.860 us) | -95% (4232 B → 192 B)
public List<int> ShuffleTakeManual()
  ⇒ ShuffleManual(_source).Take(10).ToList();

public List<int> ShuffleTakeLinq()
  ⇒ _source.Shuffle().Take(10).ToList();

private static IEnumerable<int> ShuffleManual(IEnumerable<int> source)
{
    int[] arr = source.ToArray();
    Random.Shared.Shuffle(arr);
    foreach (var item in arr)
        yield return item;
}

private IEnumerable<int> _source = Enumerable.Range(1, 1000).ToList();
```

```
// Otherwise, given totalCount items of which equalCount match the target, we're going to sample
// _takeCount items, and we need to determine how likely it is that at least one of those sampled
// items is one of the equalCount items. For that, we'll use hypergeometric distribution to determine
// the probability of drawing zero matches in our sample, at which point the chance of getting at least
// one match is the inverse.
double probOfDrawingZeroMatches = 1;
for (long i = 0; i < _takeCount; i++)
{
    probOfDrawingZeroMatches *=
        (double)(totalCount - equalCount - i) / // number of non-matching items left to draw from
        (totalCount - i); // number of items left to draw from
}

return Random.Shared.NextDouble() > probOfDrawingZeroMatches;
```

Shuffle.Take.Contains

```
// -94% (8,042.1 ns → 474.2 ns) | -99% (12136 B → 96 B)

public bool ShuffleTakeContainsManual()
  ⇒ ShuffleManual(Enumerable.Range(1, 3000).ToList()).Take(10).Contains(2000);

public bool ShuffleTakeContainsLinq()
  ⇒ _source.Shuffle().Take(10).Contains(2000);

private static IEnumerable<int> ShuffleManual(IEnumerable<int> source)
{
    int[] arr = source.ToArray();
    Random.Shared.Shuffle(arr);
    foreach (var item in arr)
    {
        yield return item;
    }
}

private IEnumerable<int> _source = Enumerable.Range(1, 3000).ToList();
```

```
public static IEnumerable<TResult> LeftJoin<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner?, TResult> resultSelector
) ⇒
    outer
    .GroupJoin(inner, outerKeySelector, innerKeySelector, (o, inners) ⇒ (o, inners))
    .SelectMany(x ⇒ x.inners.DefaultIfEmpty(), (x, i) ⇒ resultSelector(x.o, i));
```

```
public static IEnumerable<TResult> LeftJoin<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner?, TResult> resultSelector
) {
  using (IEnumerator<TOuter> e = outer.GetEnumerator())
  {
      if (e.MoveNext())
      {
          Lookup<TKey, TInner> innerLookup = Lookup<TKey, TInner>.CreateForJoin(inner, innerKeySelector, comparer);
          do {
            TOuter item = e.Current;
            Grouping<TKey, TInner>? g = innerLookup.GetGrouping(outerKeySelector(item), create: false);
            if (g is null)
              yield return resultSelector(item, default);
            else {
              int count = g._count;
              TInner[] elements = g._elements;
              for (int i = 0; i != count; ++i)
                yield return resultSelector(item, elements[i]);
            }
          }
          while (e.MoveNext());
      }
   }
}
```

LeftJoin / RightJoin

```
// -49% (16.315 us → 8.271 us) | -44% (65.84 KB → 36.95 KB)
public void LeftJoin_Manual()
  ⇒ ManualLeftJoin(Outer, Inner, o ⇒ o, i ⇒ i, (o, i) ⇒ o + i).Count();

public int LeftJoin_Linq()
  ⇒ Outer.LeftJoin(Inner, o ⇒ o, i ⇒ i, (o, i) ⇒ o + i).Count();

private static IEnumerable<TResult> ManualLeftJoin<TOuter, TInner, TKey, TResult>(
  IEnumerable<TOuter> outer, IEnumerable<TInner> inner,
  Func<TOuter, TKey> outerKeySelector, Func<TInner, TKey> innerKeySelector,
  Func<TOuter, TInner?, TResult> resultSelector)
  ⇒ outer
      .GroupJoin(inner, outerKeySelector, innerKeySelector, (o, inners) ⇒ (o, inners))
      .SelectMany(x ⇒ x.inners.DefaultIfEmpty(), (x, i) ⇒ resultSelector(x.o, i));

private IEnumerable<int> Outer { get; } = Enumerable.Sequence(0, 1000, 2);
private IEnumerable<int> Inner { get; } = Enumerable.Sequence(0, 1000, 3);
```

Regex

My kids *love* “Frozen”. They can sing every word, re-enact every scene, and provide detailed notes on the proper sparkle of Elsa’s ice dress. I’ve seen the movie more times than I can recount, to the point where, if you’ve seen me do any live coding, you’ve probably seen my subconscious incorporate an Arendelle reference or two. After so many viewings, I began paying closer attention to the details, like how at the very beginning of the film the ice harvesters are singing a song that subtly foreshadows the story’s central conflicts, the characters’ journeys, and even the key to resolving the climax. I’m slightly ashamed to admit I didn’t comprehend this connection until viewing number ten or so, at which point I also realized I had no idea if this ice harvesting was actually “a thing” or if it was just a clever vehicle for Disney to spin a yarn. Turns out, as I subsequently researched, it’s quite real.

```
// ^hello
public int Stupid() // -29% (10.533 ns → 7.438 ns)
  ⇒ stupid_regexp.Match(s_input).Length;

// (?=^)hello
public int Clever() // -99% (905,979.231 ns → 7.460 ns)
  ⇒ clever_regexp.Count(s_input);
```

```
// \s+\S+
public int Light() // -9% (85.66 ms → 78.08 ms)
  => light_regex.Count(s_input);

// ([a-z]+ )+[A-Z]
public int Heavy() // -9% (377.27 ms → 342.36 ms)
  => heavy_regex.Count(s_input);
```

```
([a-z][0-9])+a1
```

Конец цикла ≠ то, что после цикла

Начало цикла ≠ то, что после цикла

```
^(a|ab)+$
```

Начало цикла ≠ конец цикла (для циклов)

```
// (?:[\s\S])*
public int Light() // -30% (25.58 ns → 17.91 ns)
  ⇒ regex.Count(s_input);

// (?:[\d\D])*
public int Medium() // -23% (36.69 ns → 28.22 ns)
  ⇒ another_regex.Match(s_input).Length;

// (?:\n|.)*
public int Rare() // -99% (20,562,414.08 ns → 17.36 ns)
  ⇒ yet_another_regex.Count(s_input);
```

```
. = [^\n]
```

```
// \ba\b
public int CountStandaloneAs() // -10% (20.87 ms → 18.85 ms)
    ⇒ s_regex.Count(s_input);
```

```
internal static bool IsBoundary(ReadOnlySpan<char> inputSpan, int index)
{
    int indexM1 = index - 1;
    return ((uint)indexM1 < (uint)inputSpan.Length && RegexCharClass.IsBoundaryWordChar(inputSpan[indexM1])) ≠
           ((uint)index < (uint)inputSpan.Length && RegexCharClass.IsBoundaryWordChar(inputSpan[index]));
}
```

```
421 +         /// <summary>Determines whether the specified index is a boundary.</summary>",
422 +         /// <remarks>This variant is only employed when the subsequent character will separately be
      validated as a word character.</remarks>",
423 +         [MethodImpl(MethodImplOptions.AggressiveInlining)]
424 +         internal static bool IsPreWordCharBoundary(ReadOnlySpan<char> inputSpan, int index)
425 +         {
426 +             int indexMinus1 = index - 1;
427 +             return (uint)indexMinus1 >= (uint)inputSpan.Length ||
      !RegexCharClass.IsBoundaryWordChar(inputSpan[indexMinus1]);
428 +         }
429 +
430 +         /// <summary>Determines whether the specified index is a boundary.</summary>
431 +         /// <remarks>This variant is only employed when the previous character has already been validated
      as a word character.</remarks>
432 +         [MethodImpl(MethodImplOptions.AggressiveInlining)]
433 +         internal static bool IsPostWordCharBoundary(ReadOnlySpan<char> inputSpan, int index) =>
434 +             (uint)index >= (uint)inputSpan.Length || !RegexCharClass.IsBoundaryWordChar(inputSpan[index]);
435 +
```

Regex: Count vs Matches

```
// -37% (243.8 ms → 153.9 ms) | 665526880 B → -
public int MatchesCount()
    => s_regex.Matches(s_input).Count;

public int Count() // 182.2 ms → 157.4 ms | -
    => s_regex.Count(s_input);
```

Diagnostics

Stopwatch

```
public TimeSpan WithGetTimestamp() // +2% (19.08 ns → 19.45 ns)
{
    var start = Stopwatch.GetTimestamp();
    Nop();
    var end = Stopwatch.GetTimestamp();
    return Stopwatch.GetElapsedTime(start, end);
}

public TimeSpan WithStartNew() // -11% (21.48 ns → 19.07 ns)
{
    var sw = Stopwatch.StartNew();
    Nop();
    sw.Stop();
    return sw.Elapsed;
}

[MethodImpl(MethodImplOptions.NoInlining)]
private static void Nop() { }
```

Logging

```
[LoggerMessage(Level = LogLevel.Information, Message = "This happened: {Value}")]
private static partial void Oops(ILogger logger, string value);

public static void UnexpectedlyExpensive()
{
    Oops(NullLogger.Instance, $"{Guid.NewGuid()} {DateTimeOffset.UtcNow}");
}
```