Убийцы производительности

Евгений Пешков

@epeshkblog

О чём будем говорить?

- Как менялись подходы к написанию кода на .NET
- Причины низкой производительности
- Проблемы со сторонними библиотеками
- Приёмы ускорения кода

.NET Framework era

- Закрытый исходный код (доступны только reference source)
- Нет обратной связи между пользователями и разработчиками
- Нехватка примитивов для написания производительного кода
- Неоптимальный код стандартной библиотеки: аллокации и блокировки

- Ради производительности требовалось переписывать «стандартный» код

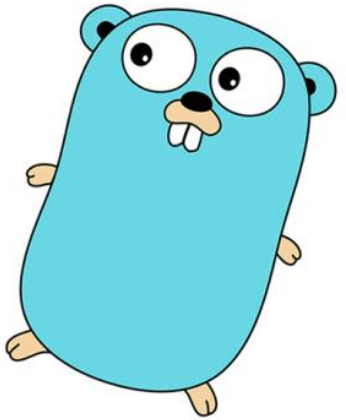
.NET Framework era: timers

```
await Task.Delay(time);


lock (timerQueue) {
  timerQueue.Enqueue(timer);
}
```

Глобальная блокировка – убийца производительности


TechEmpower




.net

Улучшения .NET

Фичи языка: ref, stackalloc, value tuples

Оптимизации на уровне IL. Редко, но бывают
Пример: сложение строк

Автоматические оптимизации при JIT-компиляции,
ускорение сборщика мусора, микрооптимизации

Новые API, ручные микрооптимизации,
большие оптимизации,
более эффективные алгоритмы

.NET Core & Modern .NET era

- Основная сложность – улучшение производительности с сохранением обратной совместимости и простоты разработки

- .NET Core 2.0 – ConcurrentQueue без аллокаций
- .NET Core 2.1 – Span<T>, SocketsHttpHandler, IValueTaskSource, Utf8Formatter
- .NET Core 3.0 – System.Text.Json
- .NET 5 – Source generators, CollectionsMarshal, GC.AllocateUninitializedArray
- .NET 6 – Interpolated string handlers, Json source generator
- .NET 7 – Regex+Span
- .NET 8 – Frozen collections, inline arrays

Пирамида перформанса

BCL

Новые API, ручные микрооптимизации,
большие оптимизации,
более эффективные алгоритмы

Сторонние библиотеки

`*.cs`

Пользовательский код

Проблемы сторонних библиотек

- Разработка начата во времена .NET Framework
- Непроизводительные абстракции
- Необходимость поддержки разных рантаймов
- Разные потребности разработчика и пользователей
- Недостаток разработчиков, времени и денег на улучшения

В итоге…

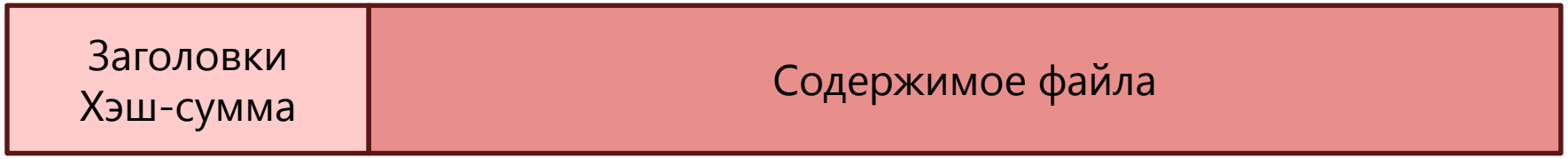
- Часто нужна доработка библиотеки под высокую нагрузку
- Иногда проще переписать с нуля
- Но всегда приятно оптимизировать под общий случай

Сторонние библиотеки на DotNext

- MongoDB.Driver
  - Точечная переработка драйвера MongoDB для многократного увеличения производительности
- SMBLibrary
  - Делаем zero-allocation код на примере оптимизации крупной библиотеки
- Amazon.S3, Minio
  - Приемы экономии памяти в .NET
- MediatR
  - MediatR не нужен
- RestSharp
  - Клиентский HTTP в .NET: От WebRequest до SocketsHttpHandler

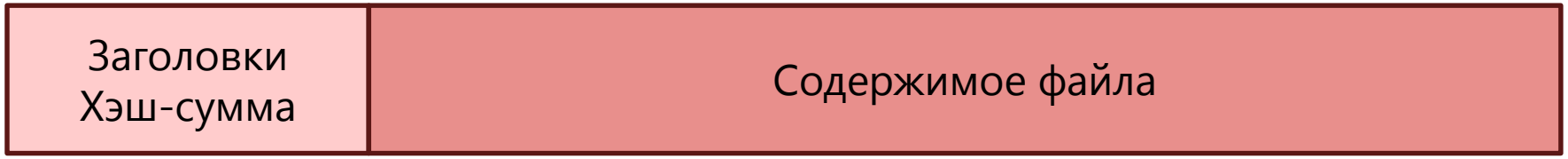
SMBLibrary

- Библиотека для передачи файлов по протоколу SMB
- Полезна при переходе на Linux
- SMB-пакет упрощённо:



SMBLibrary: недостатки

- Синхронный API для сетевых операций
- Огромные количества аллокаций



SMBLibrary

```
void Send(byte[] content)
{
  var lengthWithHeaders = HeadersLength + content.Length;
  var packet = new byte[lengthWithHeaders]; // big alloc
  SetHeaders(packet);
  SetContent(packet, content); // content copy
  socket.Send(packet);
}
```

SMBLibrary: pooling

```
void Send(byte[] content)
{
  var lengthWithHeaders = HeadersLength + content.Length;
  var packet = pool.Rent(lengthWithHeaders); // reuse array
  SetHeaders(packet);
  SetContent(packet, content); // content copy
  socket.Send(packet, 0, lengthWithHeaders);
  pool.Return(packet);
}
```

Недостатки пулинга

- Отслеживать lifetime объектов, взятых из пула – сложная работа
- Пул – то же, что и аллокатор, управление памятью становится ручным
- Реализация пула может убить производительность – иногда лучше положиться на GC

ArrayPool

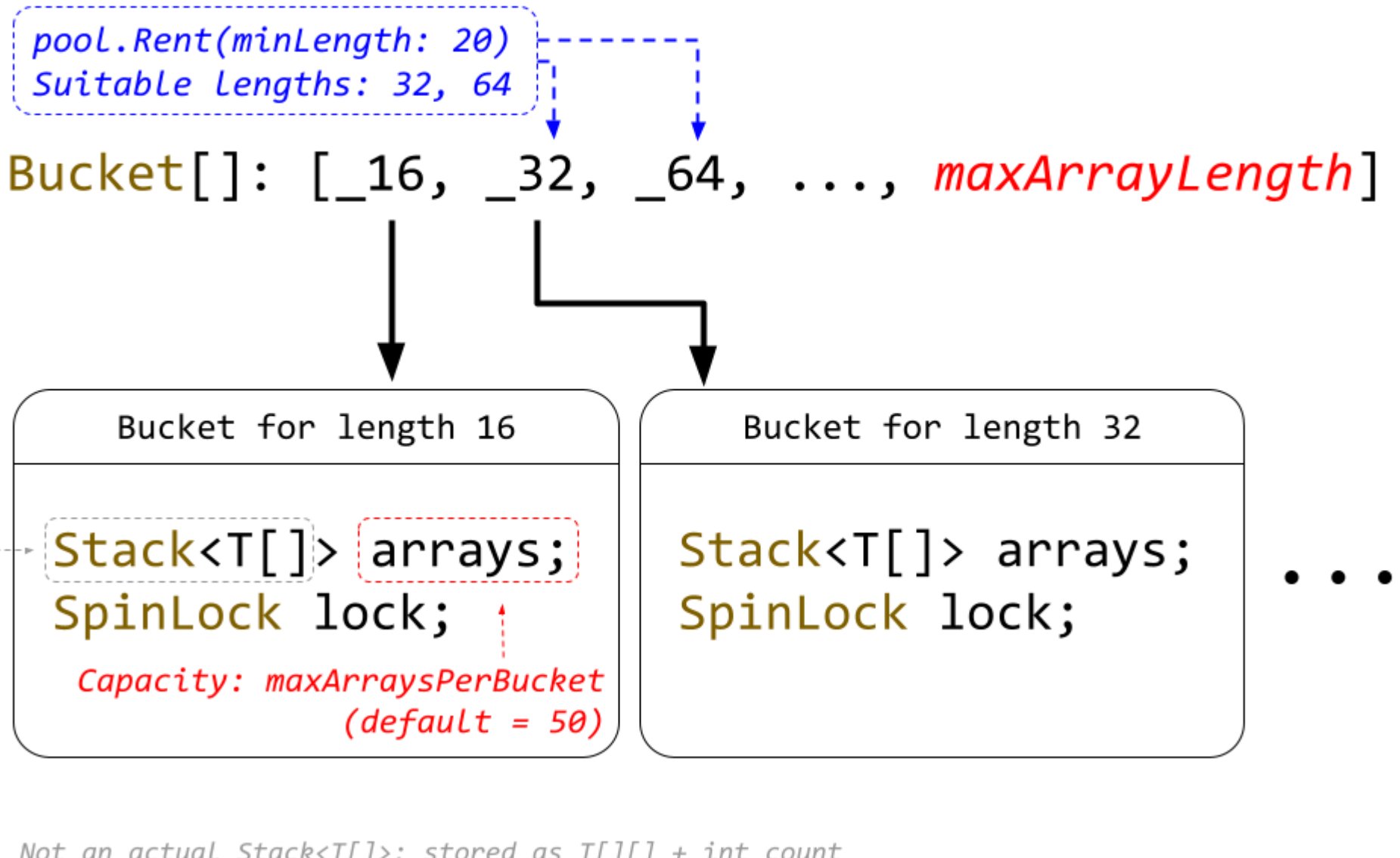
Готовые реализации ArrayPool:

- `ArrayPool<T>.Shared`
- `ArrayPool<T>.Create(maxLength, maxArraysPerBucket)`

Pool	Mean	Allocated
new []	138 ms	2146306.76 KB
Create	230 ms	2.88 KB
Shared	33 ms	2.53 KB

https://habr.com/ru/articles/723420/

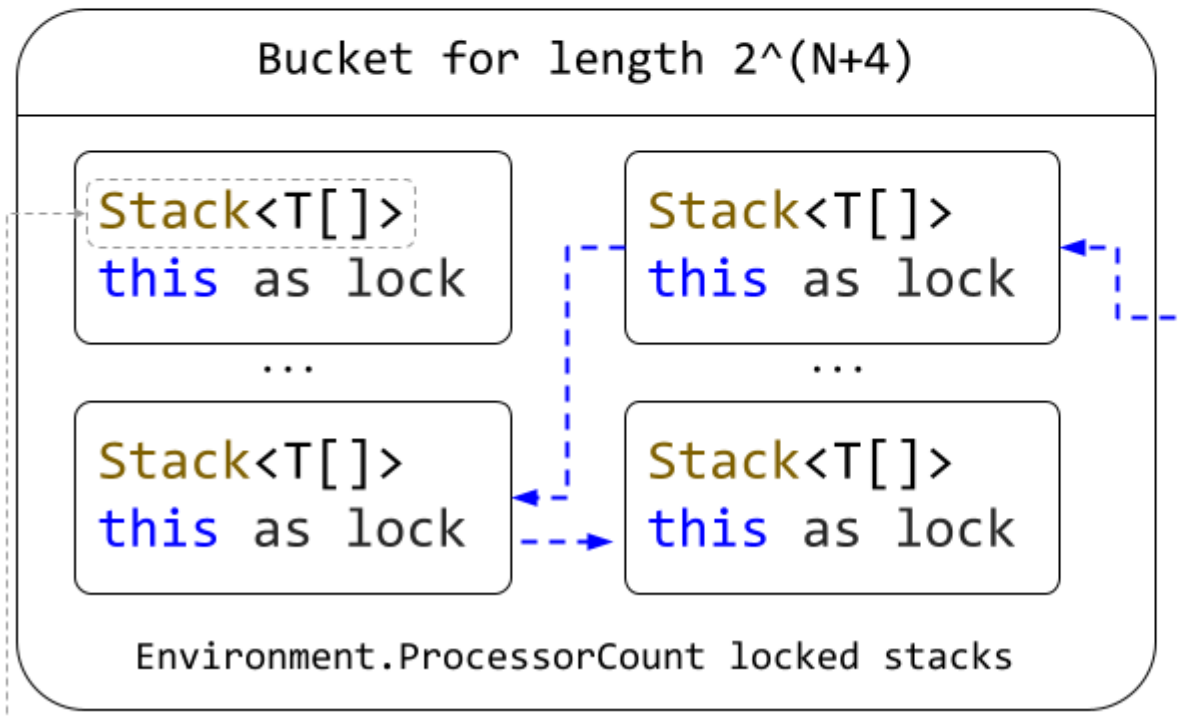
ConfigurableArrayPool<T>



ArrayPool<T>.Shared

[ThreadStatic] T[][]: Thread local cache
Single slot per length

PerCoreLockedStacks[]: [_16, _32, _64, ..., 1024*1024*1024]



Not an actual Stack<T[]>: stored as T[][] + int count

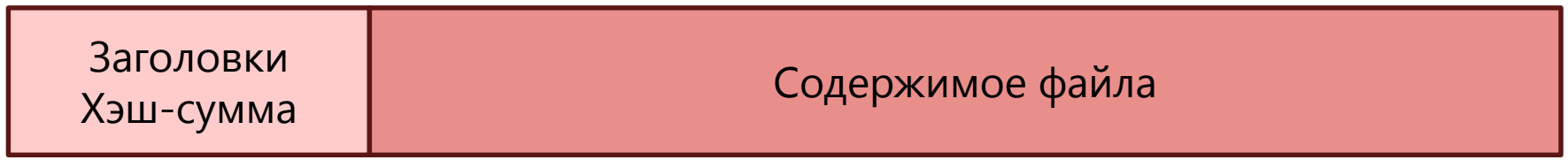
.Rent(minLength: 13)

Pooled array length: 16
Bucket: 0

1. *Look into TLS*
2. *Look into shared pool, starting from stack ProcessorId % ProcessorCount*
3. *If empty, round-robin through the other stacks*

SMBLibrary: pooling

```
void Send(byte[] content)
{
  var lengthWithHeaders = HeadersLength + content.Length;
  var packet = pool.Rent(lengthWithHeaders); // reuse array
  SetHeaders(packet);
  SetContent(packet, content); // content copy
  socket.Send(packet, 0, lengthWithHeaders);
  pool.Return(packet);
}
```



Scatter-gather IO

```
void Send(byte[] content)
{
  var headers = new byte[HeadersLength]; // small alloc
  SetHeaders(headers);
  // no copy
  socket.Send(new List<ArraySegment<byte>>{
    headers,
    content
  });
}
```

Scatter-gather IO: Socket

```
socket.Send(new List<ArraySegment<byte>>{
    headers,
    content
  });
```

- Не всегда эффективнее копирования
- Использует `ArraySegment` вместо `ReadOnlyMemory`
- Копирует список переданных буферов
- Зависит от платформы

- Доработка API запланирована на .NET 9

mgravell/Pipelines.Sockets.Unofficial

- Адаптер для работы с сокетами через API System.IO.Pipelines
- Используется в StackExchange.Redis
- Поддерживает scatter-gather IO

Scatter-gather IO: Chunked array

- ArrayPool содержит массивы разных размеров
- Последствия неоднородности размеров:
  - Фрагментация памяти
  - Неэффективное использование памяти
- Решение: разделить контент на массивы одинаковых размеров
- LOH – не абсолютное зло, фрагментация – зло

RecyclableMemoryStream

`Microsoft.IO.RecyclableMemoryStream`

- Готовая альтернатива MemoryStream
- Контент хранится в виде chunked array

Выводы

- Пулинг – не всегда эффективен
- Меньше взаимодействия между потоками – лучше
- Большие контенты можно разделить на фрагменты

Логирование и производительность

Мониторинг должен минимально нагружать программу

- Работа со строками
- Аллокации в heap
- Синхронизация потоков
- Ввод-вывод

Компоненты логера

Фасад –> Ядро –> Аппендер

- Фасад – внешний интерфейс
- Ядро – конфигурация, enrichment событий
- Аппендер – запись событий

Throughput

Пропускная способность: тысяч лог-сообщений в секунду

	Log4j 2	NLog	Serilog	manual
File	3200	1650	1250	5700
Console	50	46	1.7	140
Console (to file)	380	340	300	5300

ConsoleAppender

Разные потребности разработчиков и пользователей библиотеки:

- Предназначен для просмотра логов глазами при отладке
- Но часто используется в продакшене (контейнеры)
- Реальная консоль – медленная, рендерит текст

Цветные логи

- ConsoleAppender из Serilog раскрашивает логи
- Hello from task 10, n: 1234
- На Windows это отдельное, медленное API

- Исправляем: используем escape последовательности для раскраски 1.7 –> 16
- Убираем раскраску полностью 16 –> 37

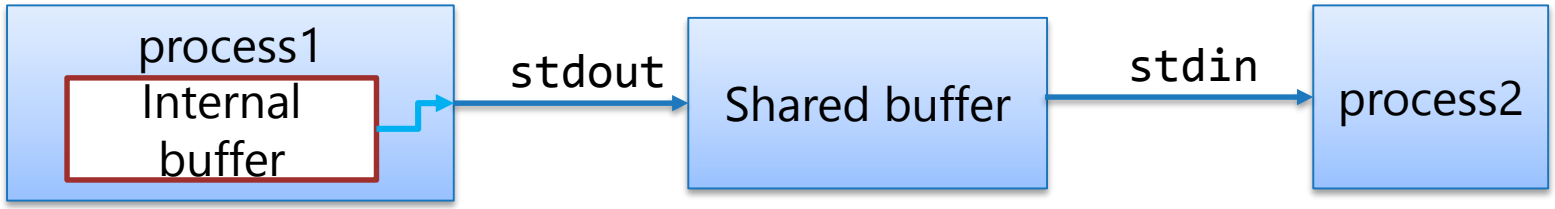
Буферизация

- Объединение нескольких записей в одну IO-операцию (вывод на консоль)
- Стандарт – flush после каждой записи
- Касается и файловых логов

- 37 –> 110

Другая сторона консоли

```
./process1 | process2
```



Другая сторона консоли

```
./process1 | process2
```

Вывод логов на реальную консоль – вредит производительности
При тестовом запуске в IDE

Throughput vs latency

- Throughput – сколько мегабайт или событий можно прокачать через логер за секунду

- Latency – время добавления события в логер.

```
log.Log("Message");
```

Throughput vs latency

- Но зачем миллионы лог-записей в секунду?

- Latency – генерация лог-записи должна минимально влиять на обработку запроса
- Малая пропускная способность – следствие нерационального использования ресурсов системы

Аппендер: sync или async

- Синхронный – вызывается в потоке, который инициировал логирование, сам выполняет запись события
- Асинхронный – помещает log событие в очередь. Снижает latency, в ущерб гарантиям доставки

	Sync	**Async**	**No log**
RPS	*550*	*590*	*640*
p50	1.9	2.1	1.9
p90	6.3	3.8	3.5
p95	6.9	3.8	3.5
p99	8.8	3.9	3.6
p999	10.7	4.3	4.0

Фасад –> Ядро –> Async Appender ~ Thread –> Appender

Минимизация latency

Фасад –> Ядро –> Async Appender ~ Thread –> Appender

Для минимизации latency требуется оптимизировать код, выполняющийся в потоке, в котором началось логирование

Serilog messageTemplate

```
logger.Information(
  $"A: {a}, B: {b}");
```

```
logger.Information(
  "A: {a}, B: {b}", a, b);
```

- Serilog – структурный логгер
- Параметры log event должны быть именованными

Template cache

- Парсинг message template – медленный
- Кэш `Dictionary<string, Template>`

- Если вместо message template использовать уникальные строки – будет cache thrashing

- А в чём ещё проблема такого кэша?

String.GetHashCode

```
public override int GetHashCode()
{
  ulong defaultSeed = Marvin.DefaultSeed;

  return Marvin.ComputeHash32(
    ref Unsafe.As<char, byte>(ref this._firstChar),
    (uint) (this._stringLength * 2),
    (uint) defaultSeed,
    (uint) (defaultSeed >> 32));
}
```

Строковый ключ в хэш таблице

- Строки сравниваются по значению
- Хэш код от значения строки не кэшируется
- Каждый поиск в хэш-таблице – вычисление хэш-кода ключа

Строковый ключ в хэш таблице

```
logger.Information(
  "A: {a}, B: {b}", // ← CONST
  a, b);
```

Шаблоны сообщений:

- строковые литералы
- закэшированные строки из ресурсов (локализация)

Строковый ключ в хэш таблице

```
class ByReferenceStringComparer : IEqualityComparer<string>
{
  public bool Equals(string? x, string? y)
    => ReferenceEquals(x, y);

  public int GetHashCode(string obj)
    => RuntimeHelpers.GetHashCode(obj);
}
```

https://github.com/serilog/serilog/pull/1947

By-value vs by-reference cache

Items	BEFORE	AFTER
10	72 us	10 us
20	145 us	22 us
50	363 us	52 us
100	734 us	108 us
1000	750 us	120 us

Разница зависит также от длины message template

Потокобезопасность

- `Dictionary<string, Template>` – непотокобезопасный
- Блокировка на чтение и запись – убьёт производительность
- `ConcurrentDictionary` – большой граф объектов в памяти
- В Serilog используется non-generic `Hashtable` + `write lock`

ConcurrencyToolkit

```
SingleWriterDictionary<string, Template>
```

- Single Writer Multiple Readers hash map
- Optimistic concurrency
- Гранулярность изменений

Stack allocation

Stack allocation

```
public void Write<T0, T1>(
  LogEventLevel level, string messageTemplate,
  T0 propertyValue0, T1 propertyValue1)
{
  if (!this.IsEnabled(level))
    return;

  this.Write(level, messageTemplate, new object[2]
  {
    (object) propertyValue0, (object) propertyValue1
  });
}
```

Stack allocation

```
public void Write<T0, T1>(
  LogEventLevel level, string messageTemplate,
  T0 propertyValue0, T1 propertyValue1)
{
  if (!this.IsEnabled(level))
    return;

  this.Write(level, messageTemplate, new object[2]
  {
    (object) propertyValue0, (object) propertyValue1
  });
}
```

Stack allocation

```
public void Write<T0, T1>(
  LogEventLevel level, string messageTemplate,
  T0 propertyValue0, T1 propertyValue1)
{
  if (!this.IsEnabled(level))
    return;

  Span<object> props = stackalloc object[2]; // ← compilation error
  props[0] = (object) propertyValue0;
  props[1] = (object) propertyValue1;
  this.Write(level, messageTemplate, props);
}
```

Stack allocation: inline array

```
[StructLayout(LayoutKind.Sequential)]
struct InlineArrayFor2Elements
{
  object Object0;
  object Object1;

  public Span<object> AsSpan() =>
    MemoryMarshal.CreateSpan(ref Object0, 2);
}
```

Stack allocation: inline array

```
public void Write<T0, T1>(
  LogEventLevel level, string messageTemplate,
  T0 propertyValue0, T1 propertyValue1)
{
  if (!this.IsEnabled(level))
    return;

  Span<object> props = new InlineArrayFor2Elements().AsSpan();
  props[0] = (object) propertyValue0;
  props[1] = (object) propertyValue1;
  this.Write(level, messageTemplate, props);
}
```

376 -> 344 bytes

Stack allocation: inline array

```
[StructLayout(LayoutKind.Sequential)]
struct InlineArrayFor2Elements
{
  object Object0;
  object Object1;

  public Span<object> AsSpan() =>
    MemoryMarshal.CreateSpan(ref Object0, 2);
}
```

Stack allocation: C# 12 inline array

```
[System.Runtime.CompilerServices.InlineArray(2)]
struct InlineArrayFor2Elements<T>
{
  T _element0;
}
```

Stack allocation: C# 12 inline array

```
[System.Runtime.CompilerServices.InlineArray(2)]
struct InlineArrayFor2Elements<T>
{
  T _element0;
}
```

Stack allocation: const as generic param

```
struct InlineArray<T, int N> { ... }

InlineArray<object, 2> array = new();
```

Implement MVP part of const generics for CoreCLR

https://github.com/dotnet/runtime/pull/89636

Выводы

- .NET становится быстрее, но на производительность программ больше влияет то, как они написаны
- В сторонних библиотеках осталось много мест для улучшений
- Дальнейшие оптимизации .NET логеров:

t.me/epeshkblog

Вопросы

Евгений Пешков

@epeshk