Многопоточность в .NET: когда производительности не хватает

DOTNEXT

Евгений Пешков

e-mail: peshkov@kontur.ru
telegram/twitter: @epeshk

О чём будем говорить?

- Многопоточность и асинхронность в .NET
- Кишки примитивов синхронизации и коллекций
- Когда они не справляются с нагрузкой
- Что делать?

1. Task.Delay & TimerQueue

Polling & long polling

Сервер – выполняет долгие операции, клиент – ждёт их.

Polling:

- Клиент периодически спрашивает сервер про результат

Long polling:

- Клиент отправляет запрос с *большим таймаутом*, а сервер отвечает по завершению операции
  - \+ Меньший объём трафика
  - \+ Клиент узнаёт о результате быстрее

Задача: timeout

Хотим подождать Task с таймаутом:

```
await SendAsync();
```

Решение

```
var sendTask = SendAsync();
var delayTask = Task.Delay(timeout);
var task = await Task.WhenAny(sendTask, delayTask);

if (task == delayTask)
    return Timeout;
```

Проблемы при long polling

- Большие таймауты
- Много параллельных запросов
- => Высокая загрузка CPU

It's WinDbg time!

```
~*e!clrstack
```

```
System.Threading.Monitor.Enter(System.Object)
System.Threading.TimerQueueTimer.Change(...)
System.Threading.Timer.TimerSetup(...)
System.Threading.Timer..ctor(...)
System.Threading.Tasks.Task.Delay(...)
```

Lock convoy

- Много потоков пытаются захватить один lock
- Под lock'ом выполняется мало кода

- Время тратится не на выполнение кода, а на синхронизацию потоков
- Блокируются потоки из тредпула – они не бесконечны

TimerQueue

- Управляет таймерами в .NET приложении
- Таймеры используются в:
  - `Task.Delay`
  - `CancellationToken.CancelAfter`
  - `HttpClient`

TimerQueue under the hood

- Global state (per-appdomain):
  - Double linked list of TimerQueueTimer
  - Lock object
- Routine, запускающая callbacks таймеров
- Таймеры не упорядочены по времени срабатывания

- Добавление таймера: O(1) + lock
- Удаление таймера: O(1) + lock
- Запуск таймеров: O(N) + lock

.NET Core fix

- Lock sharding
  - Environment.ProcessorCount TimerQueue's
- Separate queues for short/long-living timers
- Short timer: time <= 1/3 second

https://github.com/dotnet/coreclr/pull/14527

.NET Framework 4.8

netfx-port-consider

App.config

```
<AppContextSwitchOverrides value=
"Switch.System.Threading.UseNetCoreTimer=true"/>
```

https://github.com/Microsoft/dotnet-framework-early-access/blob/master/release-notes/NET48/dotnet-48-changes.md
https://github.com/dotnet/coreclr/labels/netfx-port-consider

Task.Delay: можно ли ещё эффективнее?

- BinaryHeap
- Добавление таймера: O(log(N)) + lock
- Запуск таймеров: K*O(log(N)) + lock
- Удаление таймера: O(N) + lock

Task.Delay: выводы

- Подводные камни везде - даже в самых используемых вещах
- Проводите нагрузочное тестирование
- Переходите на Core, получайте багфиксы (и новые баги) первыми :)

2. SemaphoreSlim

Задача: серверный троттлинг

Требуется ограничить число параллельно обрабатываемых запросов на сервере

Semaphore

- Примитив синхронизации, под который могут пройти N воркеров одновременно
- Остальные – ждут в очереди

```
var semaphore = new SemaphoreSlim(N);
...
await semaphore.WaitAsync();
await HandleRequestAsync(request);
semaphore.Release();
```

Задача: серверный троттлинг

Усложнение:

запросы нужно обрабатывать в LIFO порядке

- SemaphoreSlim
- ConcurrentStack
- TaskCompletionSource

Failure

- Высоконагруженное приложение
- Потребление CPU приложением периодически возрастает до 100%
- Проблема воспроизводилась каждый день на случайном сервере
- Снять дамп получилось не сразу

```
procdump64 -c 70 -ma <pid> -accepteula
```

Lock convoy

~*e!clrstack

- WaitAsync – 150 threads
- Release – 900 threads

```
System.Threading.Monitor.Enter(...)
System.Threading.SemaphoreSlim.WaitAsync(...)
Throttling.ThrottlingProvider.ThrottleAsync(...)
Http.Server.HandleThrottlingAsync(...)
```

SemaphoreSlim under the hood

- State:
  - currentCount (remaining capacity)
  - Waiters queue (double linked list)
- Sync object

Try to implement lock-free semaphore

- Операции WaitAsync, Release – неблокирующие
- Порядок – LIFO

LifoSemaphore

- State:
  - currentCount (remaining count)
  - Waiters (ConcurrentStack<TaskCompletionSource>)

LifoSemaphore: WaitAsync

```
var decrementedCount = Interlocked.Decrement(ref currentCount);

if (decrementedCount >= 0)
    return Task.CompletedTask;

var waiter = new TaskCompletionSource<bool>();
waiters.Push(waiter);

return waiter.Task;
```

LifoSemaphore: Release

```
var countBefore = Interlocked.Increment(ref currentCount) - 1;

if (countBefore < 0)
{
    if (waiters.TryPop(out var waiter))
        waiter.TrySetResult(true);
}
```

Работает?

TaskCompletionSource continuations

```
var tcs = new TaskCompletionSource<bool>();

/* Task 1 */

Console.WriteLine(1);
tcs.TrySetResult(true);
Console.WriteLine(2);

/* Task 2 */
await tcs.Task;
Thread.Sleep(-1);
```

TaskCompletionSource continuations

```
var tcs = new TaskCompletionSource<bool>(
  TaskCreationOptions.RunContinuationsAsynchronously);

/* OR */

Task.Run(() => tcs.TrySetResult(true));
```

Race condition

```
using static Interlocked;
using Waiter = TaskCompletionSource<bool>;
```

```
Task WaitAsync() {
  var decrementedCount =
    Decrement(ref currentCount);

  if (decrementedCount >= 0)
    return Task.CompletedTask;

  var waiter = new Waiter();
  waiters.Push(waiter);

  return waiter.Task;
}
```

```
void Release() {
  var countBefore =
    Increment(ref currentCount) - 1;

  if (countBefore < 0)
  {
    if (waiters.TryPop(out var waiter))
      waiter.TrySetResult(true);
  }
}
```

Race condition

```
using static Interlocked;
using Waiter = TaskCompletionSource<bool>;
```

```
Task WaitAsync() {
  var decrementedCount =
    Decrement(ref currentCount);

  if (decrementedCount >= 0)
    return Task.CompletedTask;

  var waiter = new Waiter();
  waiters.Push(waiter);

  return waiter.Task;
}
```

```
void Release() {
  var countBefore =
    Increment(ref currentCount) - 1;

  if (countBefore < 0)
  {
    if (waiters.TryPop(out var waiter))
      waiter.TrySetResult(true);
  }
}
```

LifoSemaphore: Release

```
var countBefore = Interlocked.Increment(ref currentCount) - 1;

if (countBefore < 0)
{
    Waiter waiter;

    var spinner = new SpinWait();

    while (!waiters.TryPop(out waiter))
        spinner.SpinOnce();

    waiter.TrySetResult(true);
}
```

LifoSemaphore: CoreRT

`LowLevelLifoSemaphore`

- Синхронный
- На Windows использует в качестве стека Windows IO Completion port

https://github.com/dotnet/corert/blob/master/src/System.Private.CoreLib/src/System/Threading/LowLevelLifoSemaphore.cs

...

Здесь будет сравнение разных семафоров

¯_(ツ)_/¯

Выводы

- Не надейтесь, что кишки фреймворка выживут под вашей нагрузкой
- Проще решать конкретную задачу, чем общий случай
- Нагрузочное тестирование помогает не всегда
- Опасайтесь блокировок

3. (A)sync IO

Async IO

```
System.Threading.Monitor.Enter(System.Object)
System.Threading.PinnableBufferCache.Restock(...)
System.Threading.PinnableBufferCache.Allocate(...)
System.Threading.Overlapped..ctor(...)
System.Net.HttpRequestStream...
```

Overlapped

OVERLAPPED structure – contains information used in asynchronous (or *overlapped*) input and output (I/O).

https://docs.microsoft.com/en-us/windows/desktop/api/minwinbase/ns-minwinbase-overlapped

Caching of OverlappedData

.NET 4.5.1

OverlappedDataCache

- ConcurrentStack based
- Lock free

.NET 4.5.2

PinnableBufferCache

- Two parts:
  - Lock free for Gen2 buffers
  - List+global lock for young buffers

(optimizations to avoid pinning in younger generations)

PinnableBufferCache

Lock convoy:

- Если закончились буферы
- При возврате объектов в пул

Voodoo code

```
Environment.SetEnvironmentVariable(
    "PinnableBufferCache_System.Threading.OverlappedData_MinCount",
    "10000");

new Overlapped().GetHashCode();

for (int i = 0; i < 3; i++)
    GC.Collect(GC.MaxGeneration, GCCollectionMode.Forced);
```

.NET Core

- NativeOverlapped for OverlappedData allocated on unmanaged heap
- Pinning is unnecessary
- PinnableBufferCache removed

https://github.com/dotnet/coreclr/pull/18360

Выводы

- Не все оптимизации одинаково полезны
- В этот раз – просто повезло
- И снова .NET Core (:

4. Key-value collections

Concurrent collections

- ConcurrentStack<T>
- ConcurrentQueue<T>
- ConcurrentDictionary<TKey, TValue>

ConcurrentDictionary

Применения:

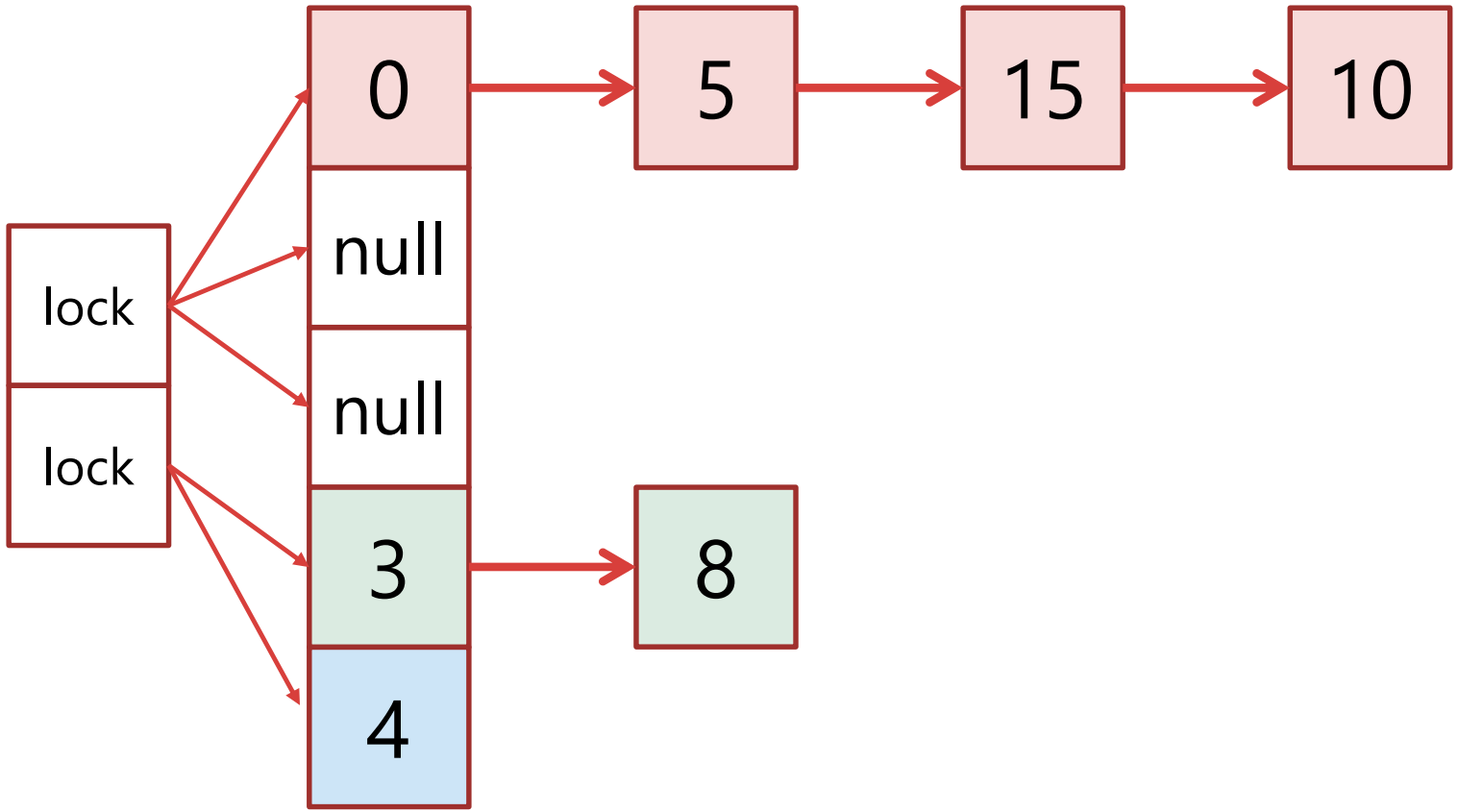
- Кэш
- Индекс

Плюсы:

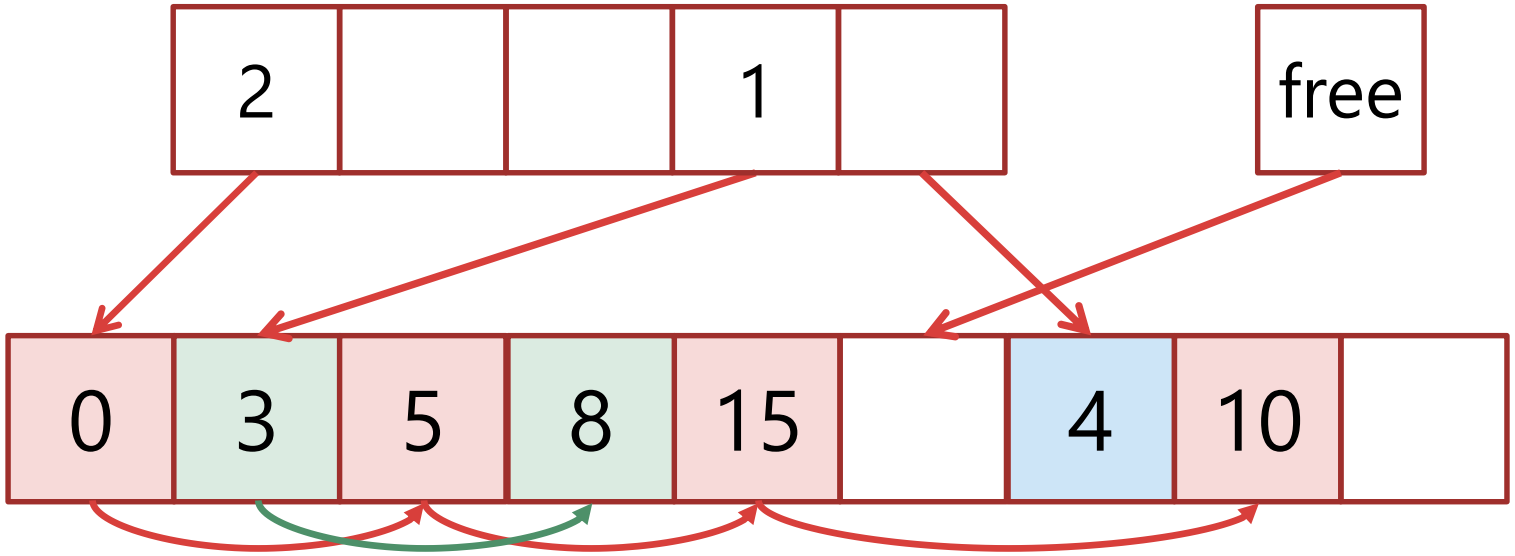
- Входит в стандартную библиотеку
- Удобные операции (TryAdd/TryUpdate/AddOrUpdate)
- Lock-free чтения
- Lock-free enumeration

.NET Dictionaries: хэш-таблица

int bucket = key.GetHashCode() % buckets.Length



.NET Dictionaries: хэш-таблица



ConcurrentDictionary: memory overhead

`struct Dictionary.Entry`

- `key`
- `value`
- **`int`** `hashCode`
- **`int`** `next`

8 bytes overhead

`class` `ConcurrentDictionary.Node`

- `key`
- `value`
- **`int`** `hashCode`
- **`object`** `next`

28 bytes overhead

ConcurrentDictionary: GC overhead

```
ConcurrentDictionary<Guid, Guid>
GC.Collect(2)
```

Count	Mean
100_000	4.9 ms
1_000_000	61.9 ms
10_000_000	474.1 ms

```
Dictionary<Guid, Guid>
GC.Collect(2)
```

Count	Mean
100_000	69.15 us
1_000_000	66.75 us
10_000_000	67.59 us

ConcurrentDictionary: простые решения

- Ограничение на размер
- TTL
- Sharding

Задача: индекс

- Нужно хранить in-memory индекс <Guid, Guid>
- В индексе $> 10^6$ элементов
- Постоянно происходят чтения из нескольких потоков, записи – редкие
- Нужно уметь перечислять все элементы в коллекции

Свой ConcurrentDictionary?!

- Lock-free reads
- Lock-free enumeration
- Small memory overhead
- Small GC overhead
- Single writer/many readers
- No LOH if possible

Dictionary

```
int[] buckets;
Dictionary.Entry entries[];

Dictionary.Entry:
```

- key, value, hashCode
- int next

Dictionary

- Потокобезопасен на чтение, если нет писателей
- Что может пойти не так, если будут записи?
  - При Resize увидели buckets и entries разных версий
  - Прочитали мусор вместо Dictionary.Entry
  - Перешли по индексам-ссылкам в мусор
  - Поток зациклился при переходе по индексам-ссылкам

Dictionary

- Потокобезопасен на чтение, если нет писателей
- Что может пойти не так, если будут записи?
  - При Resize увидели buckets и entries разных версий
  - Поток зациклился при переходе по индексам-ссылкам
  - Прочитали мусор вместо Dictionary.Entry
  - Перешли по индексам-ссылкам в мусор

Tricks from .NET Framework 1.1

`System.Collections.Hashtable`

"Hashtable is thread safe for use by **multiple reader threads** and a **single writing thread**. It is thread safe for multi-thread use when only one of the threads perform write (update) operations" – MSDN

Dictionary

- Потокобезопасен на чтение, если нет писателей
- Что может пойти не так, если будут записи?
  - При Resize увидели buckets и entries разных версий
  - Поток зациклился при переходе по индексам-ссылкам
  - Прочитали мусор вместо Dictionary.Entry
  - Перешли по индексам-ссылкам в мусор

Tricks from .NET Framework 1.1

Dictionary.Entry слишком большой – его нельзя прочитать атомарно

sizeof(DictionaryEntry) == sizeof(key) + sizeof(value) + 8

Или всё же можно?

Clean reading

```
bool writing;
int version;


while (true)
{
    int version = this.version;
    bucket = buckets[index];
    if (this.writing || version != this.version)
        continue;
    break;
}
```

Dictionary

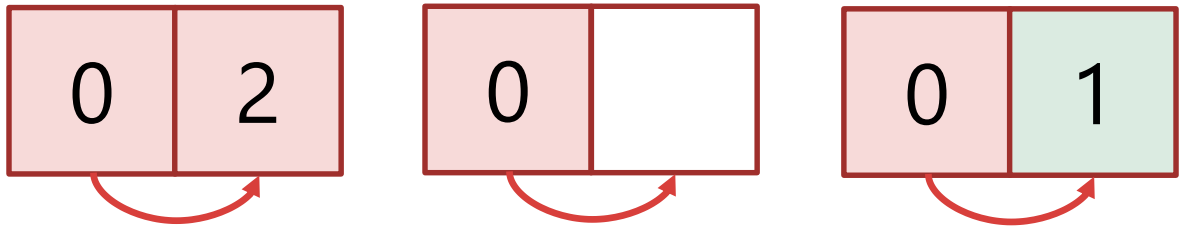
- Потокобезопасен на чтение, если нет писателей
- Что может пойти не так, если будут записи?
  - При Resize увидели buckets и entries разных версий
  - Поток зациклился при переходе по индексам-ссылкам
  - Прочитали мусор вместо Dictionary.Entry
  - Перешли по индексам-ссылкам в мусор

Trick from .NET Framework 1.1

Как переходить по индексным ссылкам?

В Dictionary можно перескочить на другой bucket

1. Read 0
2. Delete 2
3. Add 1
4. Read 1



Решение из Hashtable: отказаться от buckets и ссылок

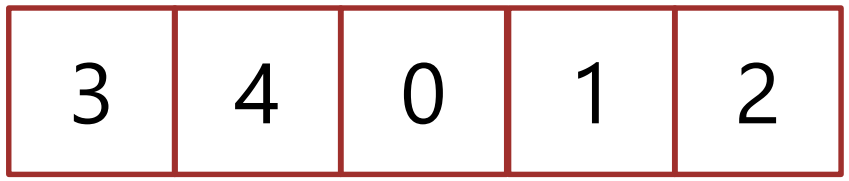
Обработка коллизий

- Элементы хранятся в массиве
- Для каждого элемента научимся вычислять порядок обхода

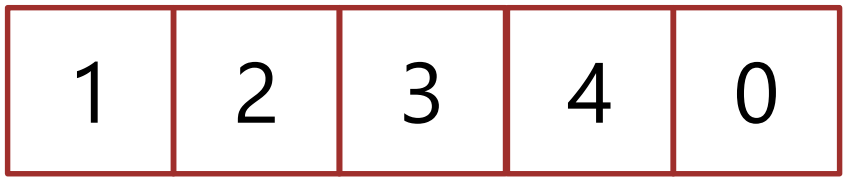
- Запись:
  - В первую свободную ячейку в порядке обхода
  - Если свободных ячеек нет – resize
- Чтение:
  - Ищем элемент в порядке обхода, пока не найдём пустую ячейку

Последовательный порядок обхода

Для элемента с hash = 2



Для элемента с hash = 4



Trick from .NET Framework 1.1

"**Enumerating** through a collection is intrinsically **not a thread safe** procedure. Even when a collection is synchronized, other threads can still modify the collection, which causes the enumerator to throw an exception." – MSDN

Trick from .NET Framework 1.1

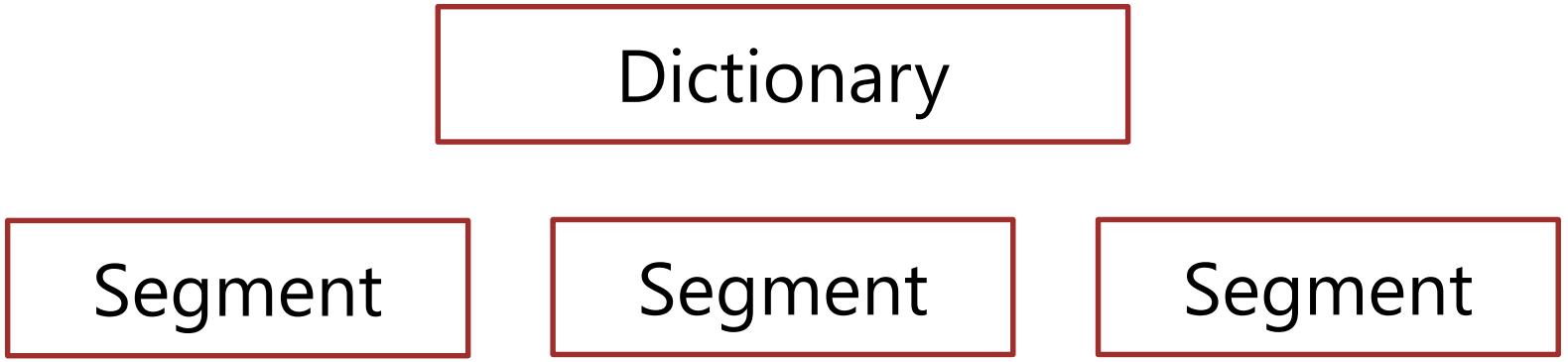
- Clean reading during enumeration
  - Elements may appears twice
- Clean reading of entire buckets

Свой ConcurrentDictionary?!

- Lock-free reads
- Lock-free enumeration
- Small memory overhead
- Small GC overhead
- Single writer/many readers
- No LOH if possible

Avoid LOH: sharding

- CustomDictionary => CustomDictionarySegment
- Массив с элементами в сегменте – маленький
- Можно сделать блокировки сегментов на запись – почти ConcurrentDictionary



...

Здесь будет сравнение разных dictionary

¯_(ツ)_/¯

Выводы

- .NET не идеален
- Ничего не идеально
- Проводите тестирование
- Знайте, как работают стандартные классы
- Изобретайте велосипеды
- Тестируйте велосипеды

LINKS

ВОПРОСЫ

DOTNEXT

Евгений Пешков

e-mail: peshkov@kontur.ru
telegram/twitter/vk: @epeshk