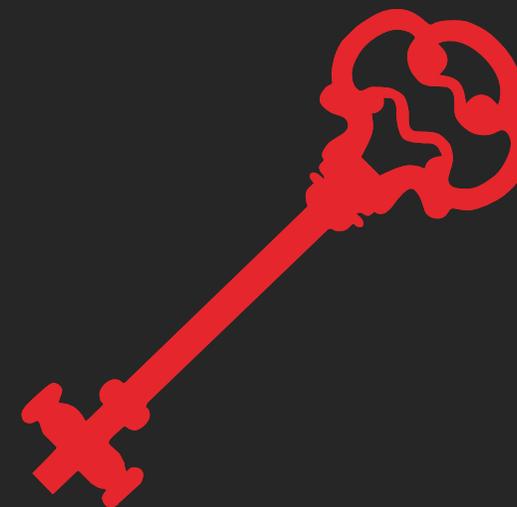
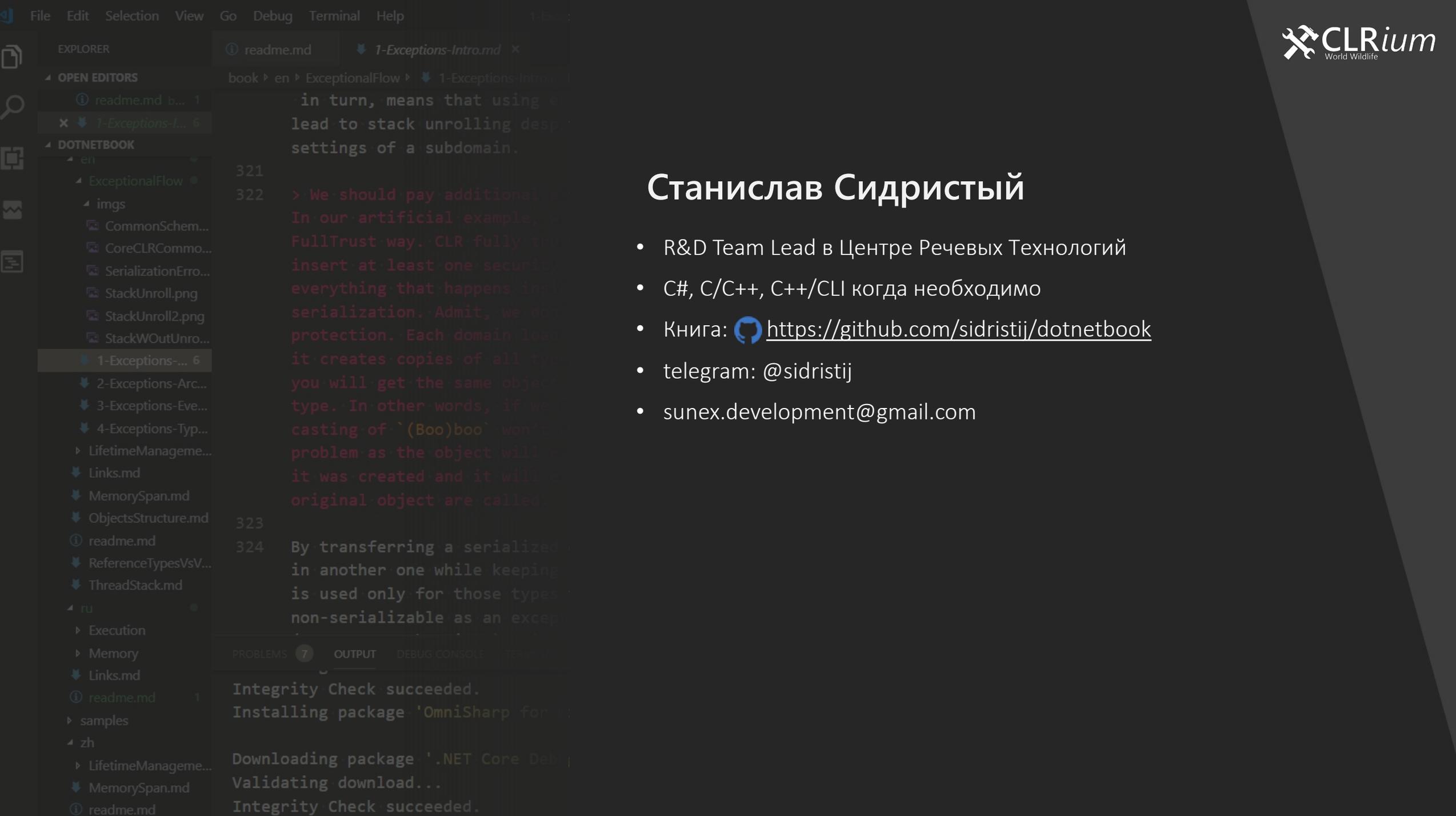


Разработка собственного пула потоков
Для получения дополнительного контроля исполнения





Станислав Сидристый

- R&D Team Lead в Центре Речевых Технологий
- C#, C/C++, C++/CLI когда необходимо
- Книга: <https://github.com/sidristij/dotnetbook>
- telegram: @sidristij
- sunex.development@gmail.com



и целого мира мало

Первый акт — это презентация основных действующих лиц, зов к странствию, встреча с наставником, принятие зова

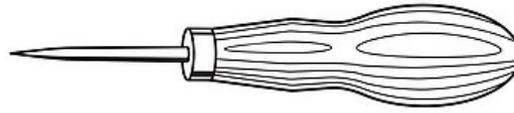
kwanini?

どうして?

zakaj?

為什麼?

ทำไม?



ngoba?

hvorfor?

Зачем? Why? Warum?

чаро?

vim li cas?



perché?

چرا؟

kodél?

왜요?

нигә?

Тест

```
private static int TestRegularPool()
{
    var sw = Stopwatch.StartNew();
    var @event = new CountdownEvent(count);

    for (var i = 0; i < count; i++)
    {
        ThreadPool.QueueUserWorkItem((x) =>
        {
            x.Signal();
        }, @event, false);
    }

    @event.Wait();
    sw.Stop();
    return (int)sw.ElapsedMilliseconds;
}
```

~112,1 мсек

```
private static int TestSmartPool(SmartThreadPool<object> pool)
{
    var sw = Stopwatch.StartNew();
    var @event = new CountdownEvent(count);

    for (var i = 0; i < count; i++)
    {
        pool.Enqueue((state) =>
        {
            ((CountdownEvent)state).Signal();
        }, @event, false);
    }

    @event.Wait();
    sw.Stop();
    return (int)sw.ElapsedMilliseconds;
}
```

~80,5 мсек

- 28%

Threads (user/all): 17/18

Show system threads

- ▶ **Main Thread** • 1 739 ms
- ▶ **Thread #6** • 1 580 ms
 - ▶ **78,53%** Dispatch • 1 241 ms • System.Threading.ThreadPoolWorkQueue.Dispatch
 - ▶ **71,51%** Dequeue • 1 130 ms • System.Threading.ThreadPoolWorkQueue.Dequeue(ThreadPoolWorkQueueThreadLocals, ref B
 - ▶ **69,39%** TryDequeue • 1 096 ms • System.Collections.Concurrent.ConcurrentQueueSegment`1.TryDequeue(out T)
 - ▶ **41,59%** SpinOnceCore • 657 ms • System.Threading.SpinWait.SpinOnceCore(Int32)
 - ▶ **0,40%** [Garbage collection] • 6 ms
 - ▶ **2,13%** [Garbage collection] • 34 ms
 - ▶ **4,65%** <TestRegularPool>b_7_0 • 74 ms • Demo.Program+<>c.<TestRegularPool>b_7_0(Object)
 - ▶ **0,57%** [Garbage collection] • 9 ms
 - ▶ **0,34%** EnsureThreadRequested • 5 ms • System.Threading.ThreadPoolWorkQueue.EnsureThreadRequested
 - ▶ **0,23%** IsInstanceOfClass • 4 ms • System.Runtime.CompilerServices.CastHelpers.IsInstanceOfClass(Void*, Object)
 - ▶ **14,44%** PerformWaitCallback • 228 ms • System.Threading.ThreadPoolWaitCallback.PerformWaitCallback
 - ▶ **7,03%** [Native or optimized code] • 111 ms
- ▶ **Thread #9** • 1 407 ms
- ▶ **Thread #13** • 1 600 ms
- ▶ **Thread #12** • 1 583 ms
- ▶ **Thread #11** • 1 570 ms
- ▶ **Thread #14** • 1 615 ms
- ▶ **Thread #8** • 1 611 ms
- ▶ **Thread #7** • 1 601 ms
- ▶ **Thread #5** • 1 548 ms
- ▶ **Thread #15** • 757 ms
- ▶ **Thread #10** • 1 412 ms
- ▶ **Thread #4** • 1 290 ms

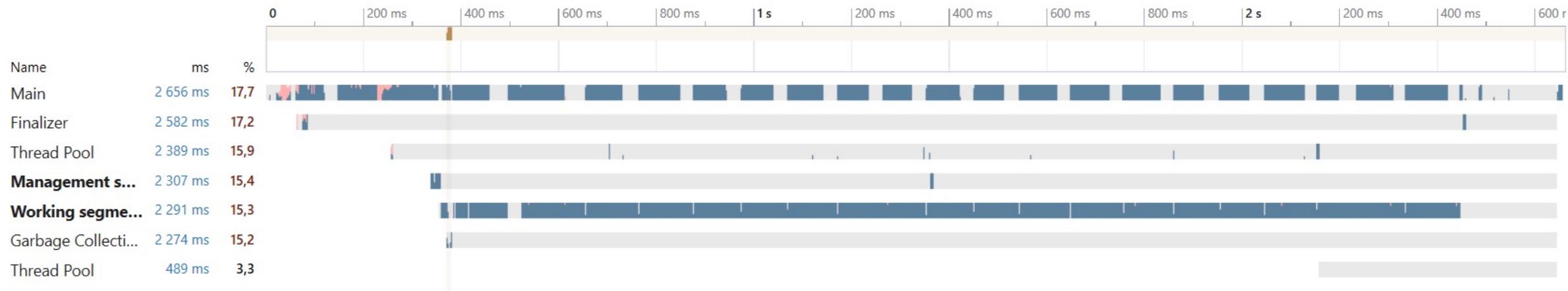
Threads (user/all): 17/18

- ▶ **Main Thread** •
- ▶ **Thread #6** • 1 580
- ▶ **Thread #9** • 1 407
- ▶ **Thread #13** • 1 600
- ▶ **Thread #12** • 1 583
- ▶ **Thread #11** • 1 570
- ▶ **Thread #14** • 1 615
- ▶ **Thread #8** • 1 611
- ▶ **Thread #7** • 1 601
- ▶ **Thread #5** • 1 548
- ▶ **Thread #15** • 757
- ▶ **Thread #10** • 1 412
- ▶ **Thread #4** • 1 290
- ▶ **Thread #16** • 1 290
- ▶ **Thread #17** • 1 010
- ▶ **Thread #18** • 1 010
- ▶ **Finalizer Thread**

ThreadPool



SmartThreadPool*



история про серийное
убийство



Решаемые проблемы

- Контроль lifetime потока:
 - Пользовательская оболочка с OnStarted, OnStopping для потока
 - OnRun(PoolWork item) => item.Run(_connection);
для проброса состояния из оболочки потока
- Внешнее управление количеством потоков
- Повышение скорости за счёт уменьшения борьбы за очередь
- Сдерживание «страха» перед большой очередью задач
- Авторазблокирование

Пользовательская логика

```
protected override void OnStarted()
{
    _connection = ConnectionProvider.Create();
}

protected override Task OnRun(PoolWork poolWork)
{
    return poolWork.Run(_connection);
}

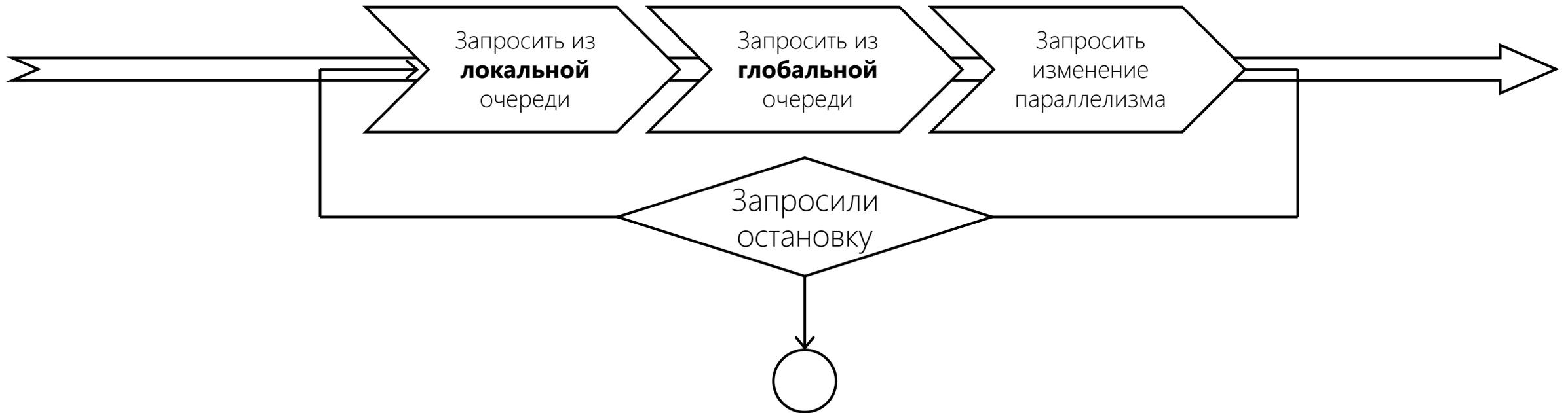
protected override void OnStopping()
{
    _connection.Dispose();
}
```



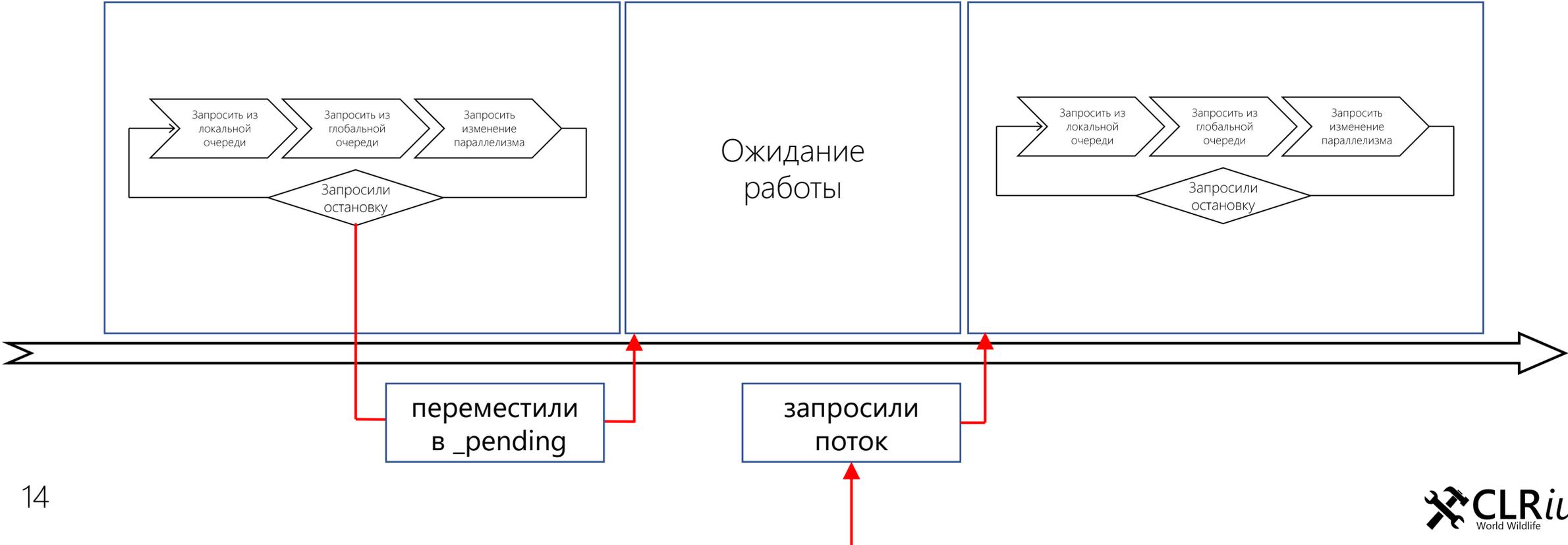
Разработка

Второй акт — это самая большая, основная часть истории. Здесь вызов, брошенный антагонистом, заставляет героя действовать. К середине второго акта он уже не может повернуть назад, как бы ему этого ни хотелось. Во второй половине второго акта многократно возрастают ставки и риски. И к концу второго акта герой терпит большое поражение, оказываясь в максимальной опасности, практически в безвыходной ситуации.

Очередь отрезков исполнения



Очередь отрезков исполнения



Код

```
131 // Initialize with parametrized async delegate
132 // </summary>
133 // 3 usages Stanislav Sidristij*
134 public void Enqueue(ExecutionUnitAsync<TPoolParameter> unit, object outer = default, bool preferLocal = true)
135 {
136     PoolWork poolWork = default;
137     poolWork.Init(unit, outer);
138     _defaultQueue.Enqueue(poolWork, preferLocal);
139 }
140
141 // <summary>
142 // Initialize with WaitHandle and non-parametrized continuation
143 // </summary>
144 // Stanislav Sidristij*
145 public void RegisterWaitForSingleObject(WaitHandle handle, ExecutionUnit unit, object outer = default, TimeSpan timeout = def
146 {
147     var wfsoState = new WaitForSingleObjectState
148     {
149         Delegate = unit,
150         Timeout = timeout,
151         Handle = handle,
152         InternalState = outer
153     };
154
155     void ExecutionUnitCallback(object args)
156     {
157         var workload = (WaitForSingleObjectState)args;
158         workload.Handle.WaitOne(workload.Timeout);
159         workload.Delegate(workload.InternalState);
160     }
161 }
```



Выводы

Выводы

- Если замечать проблемы, придёт понимание, как их решать
- Можно менять стандартные, привычные механизмы
- Решение может получиться лучше чем у профи в .NET Team
- Если не попробуешь – никогда не узнаешь.



QA