ИНФОРМАТИКА И ИНФОРМАЦИОННЫЕ ПРОЦЕССЫ/INFORMATICS AND INFORMATION PROCESSES

DOI: https://doi.org/10.60797/IRJ.2025.157.40

ОПТИМИЗАЦИЯ АСИНХРОННЫХ ОПЕРАЦИЙ В .NET

Научная статья

Гибадуллин Р.Ф.[1,] *, Гашигуллин Д.А.[2]
[1] ORCID : 0000-0001-9359-911X;
[1, 2] Казанский национальный исследовательский технический университет им. А.Н. Туполева – КАИ, Казань, Российская Федерация

* Корреспондирующий автор (landwatersun[at]mail.ru)

Аннотация

Статья посвящена принципам управления состояниями асинхронных операций на платформе .NET с целью минимизации аллокаций в управляемой памяти. Проведён анализ типичных сценариев использования асинхронного программирования в клиент-серверных приложениях. Основное внимание уделено следующим аспектам: синхронное завершение асинхронных функций, стратегии кеширования задач Task и Task<T>, применение типов ValueTask и ValueTask<T>, реализации интерфейса IValueTaskSource<T>. Приведены результаты тестирования различных типов асинхронных методов на примере вычисления функции Аккермана. Статья предназначена для разработчиков программного обеспечения, работающих над созданием высокопроизводительных и отзывчивых .NET-приложений, в которых критически важна оптимизация ресурсоёмких асинхронных операций.

Ключевые слова: платформа .NET, асинхронное программирование, параллельное программирование, Task, ValueTask, IValueTaskSource, кеширование задач, синхронное завершение, оптимизация памяти, производительность.

OPTIMISING ASYNCHRONOUS OPERATIONS IN .NET

Research article

Gibadullin R.F.[1,] *, Gashigullin D.A.[2]
[1] ORCID : 0000-0001-9359-911X;
[1, 2] Kazan National Research Technical University named after A.N. Tupolev – KAI, Kazan, Russian Federation

* Corresponding author (landwatersun[at]mail.ru)

Abstract

The article is devoted to the principles of state management of asynchronous operations on the .NET platform in order to minimise allocations in managed memory. Typical scenarios of using asynchronous programming in client-server applications are analysed. The focus is on the following aspects: synchronous termination of asynchronous functions, Task and Task<T> caching strategies, application of ValueTask and ValueTask<T> types, IValueTaskSource<T> interface implementations. The results of testing different types of asynchronous methods on the example of calculating the Ackerman function are given. The paper is intended for software developers working on creating high-performance and responsive .NET applications where optimisation of resource-intensive asynchronous operations is critical.

Keywords: .NET platform, asynchronous programming, parallel programming, Task, ValueTask, IValueTaskSource, task caching, synchronous termination, memory optimisation, performance.

Введение

Принцип асинхронного программирования состоит в том, что длительно выполняющиеся (или потенциально длительно выполняющиеся) функции реализуются асинхронным образом. Он отличается от традиционного подхода синхронной реализации длительно выполняющихся функций с последующим их вызовом в новом потоке или в задаче для введения параллелизма по мере необходимости.

Асинхронный подход обеспечивает:

- параллельное выполнение операций ввода-вывода без связывания потоков;
- уменьшение количества кода в рабочих потоках обогащенных клиентских приложений (созвучно с понятием «*толстый*» клиент).

Это приводит к двум сценариям использования асинхронного программирования.

Первый сценарий касается серверных приложений, которые обрабатывают множество параллельных операций ввода-вывода. В таких приложениях важна не безопасность потоков (разделяемое состояние минимально), а эффективность их использования, чтобы поток, обрабатывающий клиентские запросы, не простаивал на сетевых операциях.

Второй сценарий упрощает поддержку потокобезопасности в обогащенных клиентских приложениях, для которых в целях упрощения программы проводится рефакторинг крупных методов в методы меньших размеров, получая в результате цепочки методов, которые вызывают друг друга (графы вызовов).

Если любая операция в графе *синхронных* вызовов длительная, то весь граф запускается в рабочем потоке (или рабочих потоках) для сохранения отзывчивости пользовательского интерфейса. Так реализуется *крупномодульный параллелизм*.

Мелкомодульный параллелизм — последовательность небольших параллельных операций, между которыми выполнение возвращается в главный поток пользовательского интерфейса [1]. Например, методы, отвечающие за

отражение промежуточных результатов, не требуют значительного времени и могут выполняться в основном потоке, что упрощает потокобезопасность.

Целесообразность применения асинхронного программирования несомненна при работе с операциями ввода-вывода и трудоемкими по времени вычислениями [2], [3]. Однако, чрезмерная асинхронность может снизить производительность из-за накладных расходов связанных с выделением объектов-обещаний в управляемой памяти (куче).

Целью исследования является разработка принципов асинхронного программирования, обеспечивающих эффективное управление состояниями асинхронных операций при снижении накладных расходов на управление памятью. Основное внимание уделяется следующим аспектам:

- синхронное завершение асинхронной функции;
- кеширование *Task*/*Task*<*T*>;
- применение структуры *ValueTask*/*ValueTask*<*T*>;
- реализация *IValueTaskSource*<*T*>.

Синхронное завершение асинхронной функции

Возврат из асинхронной функции может произойти перед организацией ожидания. Рассмотрим следующий код, который обеспечивает кеширование в процессе загрузки веб-страниц:

```
async void Main()
{
  string html = await GetWebPageAsync("https://csharpcooking.github.io");
  html.Length.Dump("Страница загружена");
  // Попробуем снова. В этот раз должно быть мгновенно:
  html = await GetWebPageAsync("https://csharpcooking.github.io");
  html.Dump("Страница загружена");
}
static Dictionary<string, string> _cache = new Dictionary<string, string>();
async Task<string> GetWebPageAsync(string uri)
{
  string html;
  if (_cache.TryGetValue(uri, out html)) return html;
  return _cache[uri] = await new WebClient().DownloadStringTaskAsync(uri);
}
```

При ожидании задачи компилятор оптимизирует код, проверяя свойство *IsCompleted*. Если задача уже завершена (например, при наличии данных в кеше), выполнение происходит с возвратом завершённого экземпляра задачи без создания продолжения. Это называется *синхронным завершением*. В противном случае создается продолжение для асинхронного выполнения. Такой подход позволяет избежать накладных расходов на асинхронность, когда она не требуется, ускоряя выполнение кода, когда данные доступны немедленно.

В представленном примере ожидание асинхронной функции, которая завершается синхронно, все равно связано с небольшими накладными расходами (компилятор все равно добавляет код для управления состоянием метода и возможным продолжением) — примерно 20 наносекунд на современных компьютерах. Напротив, переход в пул потоков вызывает переключение контекста — возможно одну или две микросекунды, а переход в цикл обработки сообщений пользовательского интерфейса — минимум в десять раз больше (и еще больше, если пользовательский интерфейс занят) [3].

Асинхронное программирование в C# предоставляет интересную возможность: создавать асинхронные методы, которые фактически никогда не используют ключевое слово *await*. Например, можно написать метод вида:

```
async Task<string> Foo() { return "abc"; }
```

Хотя компилятор и выдаст предупреждение об отсутствии *await*, такой код вполне допустим и работоспособен. Альтернативный способ достижения того же результата заключается в использовании метода *Task.FromResult*, который возвращает уже завершенную задачу. В этом случае метод может выглядеть так:

```
Task<string> Foo() { return Task.FromResult("abc"); }
```

Этот вариант не требует использования ключевого слова *async*. Оба подхода позволяют сохранить согласованность интерфейса и обеспечивают гибкость при реализации асинхронных интерфейсов. Они особенно полезны, когда необходимо работать с асинхронным кодом в преимущественно синхронных сценариях. Важно отметить, что в обоих случаях возвращается сигнализированная (завершенная) задача. Когда же метод помечен как *async* и использует *await*,

компилятор автоматически генерирует состояние машины (state machine) для управления асинхронными операциями. Это включает в себя создание нескольких объектов и структур для управления жизненным циклом задачи, что влечет за собой дополнительные накладные расходы.

Если наш метод *GetWebPageAsync* вызывается из потока пользовательского интерфейса, то он является неявно безопасным к потокам в том смысле, что его можно было бы вызвать несколько раз подряд (инициируя тем самым множество параллельных загрузок) без необходимости в каком-либо блокировании с целью защиты кеша. Однако если бы последовательность обращений относилась к одному и тому же URI, то инициировалось бы множество избыточных загрузок, которые все в конечном итоге обновляли бы одну и ту же запись в кеше. Хоть это и не ошибка, но более эффективно было бы сделать так, чтобы последующие обращения к тому же самому URI взамен (асинхронно) ожидали результата выполняющегося запроса.

Существует простой способ достичь указанной цели, не прибегая к блокировкам или сигнализирующим конструкциям. Вместо кеша строк мы создаем объект-обещания (*Task<string>*) — кеш «будущего»:

```
static Dictionary<string, Task<string>> _cache = new Dictionary<string, Task<string>>();
Task<string> GetWebPageAsync(string uri)
{
 Task<string> downloadTask;
 if (_cache.TryGetValue(uri, out downloadTask)) return downloadTask;
 return _cache[uri] = new WebClient().DownloadStringTaskAsync(uri);
}
```

Обратите внимание, что мы не помечаем метод как *async*, поскольку напрямую возвращаем объект задачи, полученный в результате вызова метода класса *WebClient*.

Теперь при повторяющихся вызовах метода *GetWebPageAsync* с тем же самым URI мы гарантируем получение одного и того же объекта *Task<string>*. Это обеспечивает и дополнительное преимущество минимизации нагрузки на сборщик мусора [4], [5]. И если задача завершена, то ожидание не требует больших затрат благодаря описанной выше оптимизации, которую предпринимает компилятор.

Чтобы сделать код безопасным к потокам без защиты со стороны контекста синхронизации, необходимо блокировать все тело метода [6]:

```
lock (_cache)
{
 Task<string> downloadTask;
 if (_cache.TryGetValue(uri, out downloadTask)) return downloadTask;
 return _cache[uri] = new WebClient().DownloadStringTaskAsync(uri);
}
```

В данном решении мы производим блокировку не на время загрузки страницы (это нанесло бы ущерб параллелизму), а на небольшой промежуток времени, пока проверяется кеш и при необходимости запускается новая задача, которая обновляет кеш.

Кеширование Task

Одной из ключевых концепций в асинхронном программировании является использование класса *Task*, который представляет собой асинхронную операцию. Однако, несмотря на все свои преимущества [3], *Task* имеет потенциальную слабую сторону, особенно когда создается большое количество его экземпляров. Когда асинхронные методы выполняются часто и создают множество объектов *Task*, это может привести к значительным накладным расходам. Каждый объект *Task* должен быть создан и размещен в управляемой куче, что требует дополнительных затрат ресурсов на его управление и последующую уборку сборщиком мусора.

Среда выполнения .NET предоставляет механизмы для снижения числа создаваемых объектов в управляемой памяти. Когда метод завершается синхронно, нет необходимости создавать новый объект *Task* [7]. Вместо этого можно использовать уже существующий экземпляр, что значительно уменьшает накладные расходы.

Для иллюстрации рассмотрим следующий пример:

```
public async Task WriteAsync(byte value)
{
 if (_bufferedCount == _buffer.Length)
 {
  await FlushAsync();
 }
 _buffer[_bufferedCount++] = value;
}
```

В данном примере, если метод завершается синхронно, ему не нужно возвращать новый *Task*, так как возвращаемое значение отсутствует. В таких случаях платформа .NET использует кешированный необобщенный *Task*, который возвращает пустое значение (эквивалент *void* в синхронных методах). Этот кешированный синглтон доступен через свойство *Task.CompletedTask*. (Слово «синглтон» (от англ. «singleton») в программировании обозначает паттерн проектирования, который ограничивает создание экземпляра класса одним объектом.)

Таким образом, если в методе *WriteAsync* буфер заполнен, вызывается метод *FlushAsync*, который является асинхронным и возвращает *Task*. Однако, если в буфере достаточно свободного пространства, операция записи выполняется синхронно, и новый объект *Task* не создается.

Или, например, представим код, в котором метод *NextValueAsync* имеет возвращаемый объект типа *Task<bool>*:

```
static Random r = new Random();
static int _random;
public static async Task Main(string[] args)
{
  List<Task<bool>> taskList = new List<Task<bool>>();
  for (int i = 0; i < 1000; i++)
  {
    taskList.Add(NextValueAsync());
  }
  // Ожидаем завершения всех задач
  await Task.WhenAll(taskList);
  // Группируем задачи по результату и проверяем эквивалентность экземпляров
  var groupedTasks = taskList.GroupBy(task => task.Result);
  foreach (var group in groupedTasks)
  {
    var firstTask = group.First();
    bool allSame = group.All(task => object.ReferenceEquals(task, firstTask));
    Console.WriteLine(allSame
        ? [__rj__content__placeholder__]quot;Все задачи с результатом {group.Key} являются одним и тем
же экземпляром."
        : [__rj__content__placeholder__]quot;Не все задачи с результатом {group.Key} являются одним и тем
же экземпляром.");
  }
}
static async Task<bool> NextValueAsync()
{
  await RandomWithAsyncDelay(false);
  return _random > 0;
}
static async Task RandomWithAsyncDelay(bool setAsync)
{
  if (setAsync) await Task.Delay(1000);
  _random = r.Next(0, 2);
}
```

Поскольку есть только два возможных результата типа *bool*(*true* и *false*), то существует только два возможных объекта *Task*, которые нужны для представления этих результатов. В сценарии синхронного завершения среда .NET обеспечивает кеширование этих объектов, возвращая их с соответствующим значением без выделения памяти. Только в случае асинхронного завершения (достигается вызовом *RandomWithAsyncDelay* с параметром *setAsync* равным *true*) методу понадобится создать новый *Task*, потому что его нужно будет вернуть до того, как станет известен результат операции. Вывод программы при различных значениях параметра *setAsync* представлен на рисунках 1 и 2.

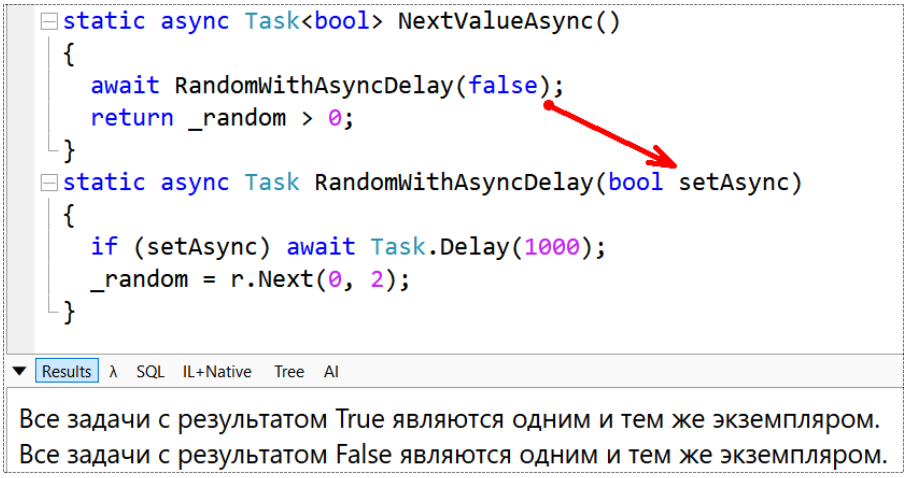

Рисунок 1 - Вывод программы при *setAsync = false*
DOI: https://doi.org/10.60797/IRJ.2025.157.40.1

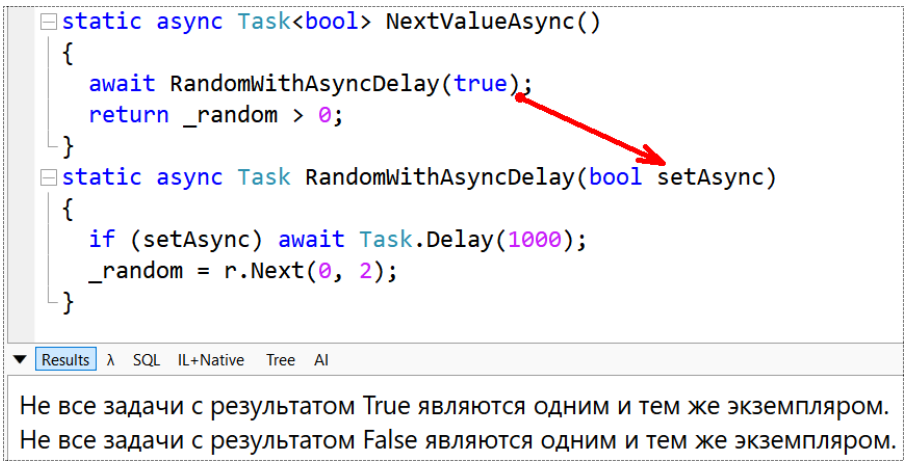

Рисунок 2 - Вывод программы при *setAsync = true*
DOI: https://doi.org/10.60797/IRJ.2025.157.40.2

При использовании *Task<int>* в качестве типа возвращаемого объекта асинхронного метода наблюдается иная ситуация. Кеширование всех возможных значений *Task<int>* потребовало бы сотни гигабайт памяти, так как *Int32* представляет собой 32-битное целое число со знаком, которое может принимать около 4,3 миллиарда уникальных значений (в диапазоне от -2147483648 до 2147483647). Поэтому среда выполнения предоставляет ограниченный кеш для *Task<int>*, покрывающий только небольшой набор значений: значения *Task<int>* кешируются для диапазона значений от -1 до 9 (для подтверждения и получения дополнительной информации можно обратиться к исходному коду в репозитории dotnet/runtime на GitHub, конкретно к файлу «Task.cs» [8]). Это означает, что если метод возвращает значение в этом диапазоне, то будет использована кешированная задача, а не создана новая.

Множество методов библиотеки стремятся сгладить ситуацию за счет использования собственного кеша. В частности, в .NET Framework 4.5 метод *MemoryStream.ReadAsync* всегда выполняется синхронно, поскольку он считывает данные из памяти (данный метод можно найти в исходном коде .NET на GitHub [9]).

Применение структуры ValueTask

.NET Core 2.0 ввел новый тип *ValueTask<TResult>* [10], доступный через NuGet пакет *System.Threading.Tasks.Extensions*, чтобы решить проблему создания ненужных объектов *Task<TResult>* при синхронных операциях. (NuGet [11] — это менеджер пакетов для платформы .NET, который позволяет разработчикам добавлять сторонние библиотеки в свои проекты, управлять зависимостями и обновлениями библиотек.) *ValueTask<TResult>* позволяет оборачивать как *TResult*, так и *Task<TResult>*. При синхронном и успешном выполнении асинхронного метода, структура *ValueTask<TResult>* возвращается без размещения объекта в куче. Только при асинхронном выполнении создается объект *Task<TResult>*, который затем оборачивается в *ValueTask<TResult>*. Если асинхронный метод завершается с исключением, для представления этого исключения используется объект *Task<TResult>*, который затем оборачивается в *ValueTask<TResult>*. Это позволяет *ValueTask<TResult>* оставаться компактной структурой, которая не требует отдельного поля для хранения исключения. Пример:

```
public async ValueTask<int> ExampleMethodAsync()
{
 try
 {
  // Выполнение асинхронной операции
  int result = await SomeAsyncOperation();
  return result;
 }
 catch (Exception ex)
 {
  // Если возникает исключение, оно будет упаковано в Task<int>
  return new ValueTask<int>(Task.FromException<int>(ex));
 }
}
```

В этом примере, если *SomeAsyncOperation* завершается синхронно и успешно, *ValueTask<int>* просто вернет результат. Если возникает исключение, создается *Task<int>* с этим исключением, и *ValueTask<int>* оборачивает его, сохраняя компактность структуры. Применение конструкции *await Task.FromException<int>(ex)* в завершении метода привело бы к созданию дополнительного кода, необходимого для управления продолжением выполнения программы. В отличие от этого, использование конструкции *new ValueTask<int>(Task.FromException<int>(ex))* является более производительным решением. Оно позволяет сразу вернуть экземпляр структуры *ValueTask<int>*, снижая накладные расходы. Такой подход оптимизирует работу программы за счет исключения ненужных операций, связанных с асинхронным управлением.

Однако при разработке высокопроизводительных сервисов по-прежнему важно минимизировать любое выделение памяти, включая размещения в куче объектов, которые связаны с асинхронными операциями. В .NET Core 2.1 тип *ValueTask<TResult>* был усовершенствован для поддержки пула объектов и их повторного использования. Вместо того чтобы просто оборачивать *TResult* или *Task<TResult>*, был введен новый интерфейс *IValueTaskSource<TResult>* [12], [13], и *ValueTask<TResult>* был дополнен возможностью оборачивать его. *IValueTaskSource<TResult>* обеспечивает основную функциональность, необходимую для представления асинхронной операции в *ValueTask<TResult>*.

Основные методы *IValueTaskSource<TResult>*:

- *GetStatus(short token)*: этот метод используется для проверки статуса асинхронной операции. Он возвращает значение типа *ValueTaskSourceStatus*, которое указывает, завершена ли операция, находится ли она в ожидании или завершилась с ошибкой.

- *OnCompleted(Action<object> continuation, object state, short token, ValueTaskSourceOnCompletedFlags flags)*: этот метод регистрирует обратный вызов (*callback*), который будет вызван при завершении асинхронной операции.

- *GetResult(short token)*: этот метод используется для получения результата операции или выброса исключения, если операция завершилась с ошибкой.

Интерфейс *IValueTaskSource<TResult>* позволяет создавать асинхронные операции, которые могут быть связаны с пулом объектов. Это позволяет многократно использовать один и тот же объект для выполнения разных асинхронных операций, избегая избыточных аллокаций памяти и снижая нагрузку на сборщик мусора. В отличие от *IValueTaskSource<TResult>* тип *Task<TResult>* разработан так, чтобы быть безопасным и предсказуемым в многопоточных сценариях. Как только задача завершена, её состояние становится неизменным, и она может быть безопасно использована многократно (например, вызов *await* может быть выполнен несколько раз). Это означает, что после завершения задачи *Task<TResult>* она больше не может быть изменена или использована для новой операции.

Проблема повторного использования *ValueTask<TResult>* хорошо прослеживается на примере следующего кода:

```
public class DatabaseReadOperation : IValueTaskSource<string>, IDisposable
{
 private static readonly ObjectPool<DatabaseReadOperation> _pool = new
DefaultObjectPool<DatabaseReadOperation>(new DefaultPooledObjectPolicy<DatabaseReadOperation>());
 private ManualResetEventSlim _completion = new ManualResetEventSlim(false);
 private string _result;
 private short _token;
 public DatabaseReadOperation() { }
 public static DatabaseReadOperation Rent(string result)
 {
  var objPool = _pool.Get();
  objPool._result = result;
  // objPool._token = (short)Environment.TickCount;
  objPool._completion.Reset();
  return objPool;
 }
 public ValueTask<string> StartobjPoolAsync()
 {
  return new ValueTask<string>(this, _token);
 }
 public void Complete()
 {
  _completion.Set();
 }
```

```
 public string GetResult(short token)
 {
  if (token != _token)
  {
   throw new InvalidOperationException("Invalid token");
  }
  return _result;
 }
 public ValueTaskSourceStatus GetStatus(short token)
 {
  if (token != _token)
  {
   throw new InvalidOperationException("Invalid token");
  }
  return _completion.IsSet ? ValueTaskSourceStatus.Succeeded : ValueTaskSourceStatus.Pending;
 }
 public void OnCompleted(Action<object> continuation, object state, short token,
ValueTaskSourceOnCompletedFlags flags)
 {
  if (token != _token)
  {
   throw new InvalidOperationException("Invalid token");
  }
  ThreadPool.QueueUserWorkItem(_ =>
  {
   _completion.Wait();
   continuation(state);
  });
 }
 public void Dispose()
 {
  // Возвращаем объект в пул после завершения
  _pool.Return(this);
 }
}
public class Program
{
 public static async void Main(string[] args)
 {
  // Первый запрос к базе данных
  var objPool = DatabaseReadOperation.Rent("Результат #1");
  ValueTask<string> firstValueTask = objPool.StartobjPoolAsync();
  // Обработка запроса
  ThreadPool.QueueUserWorkItem(async _ =>
  {
   string result = await firstValueTask;
   Console.WriteLine([__rj__content__placeholder__]quot;firstValueTask.Result: {result}");
  });
  // Повторная обработка того же ValueTask (проблема)
  ThreadPool.QueueUserWorkItem(async _ =>
  {
   Thread.Sleep(100); // Симуляция задержки
   string result = await firstValueTask;
   Console.WriteLine([__rj__content__placeholder__]quot;firstValueTask.Result: {result}");
  });
  Thread.Sleep(10); // Симуляция работы сервера
  objPool.Complete(); // Завершаем операцию
  objPool.Dispose(); // Возвращаем объект в пул
  Thread.Sleep(10); // Симуляция задержки до следующего запроса
                    // Второй запрос к базе данных, используя повторно тот же объект из пула
  objPool = DatabaseReadOperation.Rent("Результат #2");
  ValueTask<string> secondValueTask = objPool.StartobjPoolAsync();
  // Обработка второго запроса
  ThreadPool.QueueUserWorkItem(async _ =>
  {       Thread.Sleep(100); // Симуляция задержки
   string result = await secondValueTask;
   Console.WriteLine([__rj__content__placeholder__]quot;secondValueTask.Result: {result}");
  });
  objPool.Complete(); // Завершаем операцию
  objPool.Dispose(); // Возвращаем объект в пул
 }
}
```

Вывод программы:
firstValueTask.Result: Результат #1
firstValueTask.Result: Результат #2
secondValueTask.Result: Результат #2
Структурно представим выполнение вышеуказанного кода:

1. Первый запрос к базе данных:

1.1. Создается объект *objPool* типа *DatabaseReadOperation*, который арендуется из пула (*Microsoft.Extensions.ObjectPool.ObjectPool<T>*) и используется для первой асинхронной операции. Ему присваивается результат "Результат #1".

1.2. Запускаются два потока, которые ожидают завершения этой операции. Первый поток немедленно выполняет *await firstValueTask*, получает результат и завершает выполнение.

1.3. Второй поток ожидает завершения *firstValueTask* через 100 миллисекунд. Это ожидание выполняется уже после того, как первый поток завершил выполнение и объект *objPool* возвращен в пул для повторного использования (см. пункт 2.1 далее).

2. Второй запрос к базе данных:

2.1. Тот же объект *objPool* повторно арендуется для выполнения второго запроса, и ему присваивается новый результат "Результат #2".

2.2. Запускается еще один поток, который асинхронно ожидает завершения второй операции.

Таким образом, второй поток, который ожидает завершения *firstValueTask* через 100 мс. после первого, фактически работает с объектом, который уже был переиспользован для другой операции. Это приводит к ситуации, когда второй поток, ожидающий результат первого запроса, на самом деле получает результат второго запроса, что является ошибочным поведением.

В представленном коде закомментированная строка:

objPool.token = (short)Environment.TickCount;

играет важную роль в предотвращении проблемы повторного использования экземпляра *ValueTask<TResult>*. Эта строка отвечает за обновление токена, который используется для проверки корректности выполнения асинхронной операции. Токен (*_token*) используется для обеспечения уникальности операции. Каждый раз, когда объект арендуется из пула для новой операции, этот токен должен обновляться. В противном случае, если токен останется прежним, все связанные с этим объектом *ValueTask* будут указывать на принадлежность текущей операции, даже если объект уже был возвращен в пул и повторно использован для другой операции. Если каждый раз при аренде объекта из пула вы обновляете токен (например, с использованием *Environment.TickCount* или другого уникального значения), вы гарантируете, что старые *ValueTask*, связанные с этим объектом, больше не будут действительны. Это предотвращает ситуацию, когда второй поток получает результат, предназначенный для новой операции, что мы видели в предыдущем примере.

Таким образом, возможность повторного использования экземпляра *ValueTask<TResult>* без негативных последствий определяется правильной реализацией интерфейса *IValueTaskSource<TResult>*. Важнейшую роль в этом процессе играет корректное управление токенами, которые обеспечивают уникальность и правильную идентификацию каждой асинхронной операции. При правильном использовании *IValueTaskSource<TResult>* и соответствующей реализации методов интерфейса можно добиться значительного улучшения производительности, минимизируя накладные расходы на память и избегая ошибок, связанных с многократным ожиданием на одном и том же экземпляре *ValueTask<TResult>*.

Когда *ValueTask<TResult>* был представлен в .NET Core 2.0, основной акцент был сделан на оптимизации сценариев с синхронным завершением операций, чтобы избежать лишнего выделения памяти под объект *Task<TResult>*, если результат уже был доступен. Это объясняло отсутствие необходимости в необобщенном классе *ValueTask*, поскольку для синхронного выполнения можно было просто использовать синглтон *Task.CompletedTask*, что не требовало создания новых объектов.

Однако с развитием технологий и появлением требования исключить выделение памяти даже при асинхронных завершениях операций, необходимость в необобщенном *ValueTask* вновь стала актуальной. Так в .NET Core 2.1 был представлен необобщенный *ValueTask*. Это дало возможность управлять асинхронными операциями с минимальными накладными расходами, аналогично обобщенным версиям, но с пустым возвращаемым значением. Освобождение от выделения в куче при асинхронном завершении с использованием необобщенного *ValueTask* достигается за счет использования необобщенного интерфейса *IValueTaskSource*, который позволяет реализовать логику асинхронной операции так, чтобы управлять её завершением без необходимости создания нового объекта *Task* в куче.

Реализация IValueTaskSource с применением ManualResetValueTaskSourceCore

Реализация интерфейса *IValueTaskSource<T>* может показаться нетривиальной задачей. Ранее при описании проблемы повторного использования экземпляра *ValueTask<TResult>* был представлен класс *DatabaseReadOperation*, реализующий интерфейс *IValueTaskSource<string>*. В данном коде поток, выполняющий *await firstValueTask*, узнает о необходимости выполнить продолжение благодаря механизму синхронизации с использованием блокирующей конструкции *ManualResetEventSlim* [14], что является минусом.

С введением *ManualResetValueTaskSourceCore<TResult>* в .NET Core 3.0 ситуация поменялась: данная изменяемая структура предоставляет встроенные механизмы для управления состоянием асинхронной задачи и обработки продолжений, причем обеспечивая данное управление без блокирующих примитивов синхронизации. Экземпляр данной структуры можно использовать в качестве поля на объекте, чтобы помочь ему реализовать интерфейс *IValueTaskSource<T>*. Представим класс *DatabaseReadOperation* с применением этой структуры.

```
public class DatabaseReadOperation : IValueTaskSource<string>, IDisposable
{
  private static readonly ObjectPool<DatabaseReadOperation> _pool = new
DefaultObjectPool<DatabaseReadOperation>(new DefaultPooledObjectPolicy<DatabaseReadOperation>());
```

```
 private ManualResetValueTaskSourceCore<string> _core; // Используем ManualResetValueTaskSourceCore
для управления состоянием и результатом
 private string _result;
 public DatabaseReadOperation() { }
 public static DatabaseReadOperation Rent(string result)
 {
  var objPool = _pool.Get();
  objPool._result = result;
  objPool._core.Reset(); // Сбрасываем состояние перед повторным использованием
  return objPool;
 }
 public ValueTask<string> StartOperationAsync()
 {
  return new ValueTask<string>(this, _core.Version);
 }
 public void Complete()
 {
  _core.SetResult(_result); // Завершаем операцию и устанавливаем результат
 }
 public string GetResult(short token)
 {
  return _core.GetResult(token); // Получаем результат операции
 }
 public ValueTaskSourceStatus GetStatus(short token)
 {
  return _core.GetStatus(token); // Получаем статус операции
 }
 public void OnCompleted(Action<object> continuation, object state, short token,
ValueTaskSourceOnCompletedFlags flags)
 {
  _core.OnCompleted(continuation, state, token, flags); // Регистрируем продолжение
 }
 public void Dispose()
 {
  // Возвращаем объект в пул после завершения
  _pool.Return(this);
 }
}
```

На рисунке 3 представлен вывод после запуска кода:

```
// Первый запрос к базе данных
var objPool = DatabaseReadOperation.Rent("Результат #1");
ValueTask<string> firstValueTask = objPool.StartOperationAsync();
// Обработка запроса
ThreadPool.QueueUserWorkItem(async _ =>
{
 string result = await firstValueTask;
 Console.WriteLine([__rj__content__placeholder__]quot;firstValueTask.Result: {result}");
});
// Повторная обработка того же ValueTask (проблема)
ThreadPool.QueueUserWorkItem(async _ =>
{
 Thread.Sleep(100); // Симуляция задержки
 string result = await firstValueTask;
 Console.WriteLine([__rj__content__placeholder__]quot;firstValueTask.Result: {result}");
});
Thread.Sleep(10); // Симуляция работы сервера
objPool.Complete(); // Завершаем операцию
objPool.Dispose(); // Возвращаем объект в пул
Thread.Sleep(10); // Симуляция задержки до следующего запроса
// Второй запрос к базе данных, используя повторно тот же объект из пула
objPool = DatabaseReadOperation.Rent("Результат #2");
ValueTask<string> secondValueTask = objPool.StartOperationAsync();
// Обработка второго запроса
ThreadPool.QueueUserWorkItem(async _ =>
{
 Thread.Sleep(100); // Симуляция задержки
 string result = await secondValueTask;
 Console.WriteLine([__rj__content__placeholder__]quot;secondValueTask.Result: {result}");
});
objPool.Complete(); // Завершаем операцию
objPool.Dispose(); // Возвращаем объект в пул
```

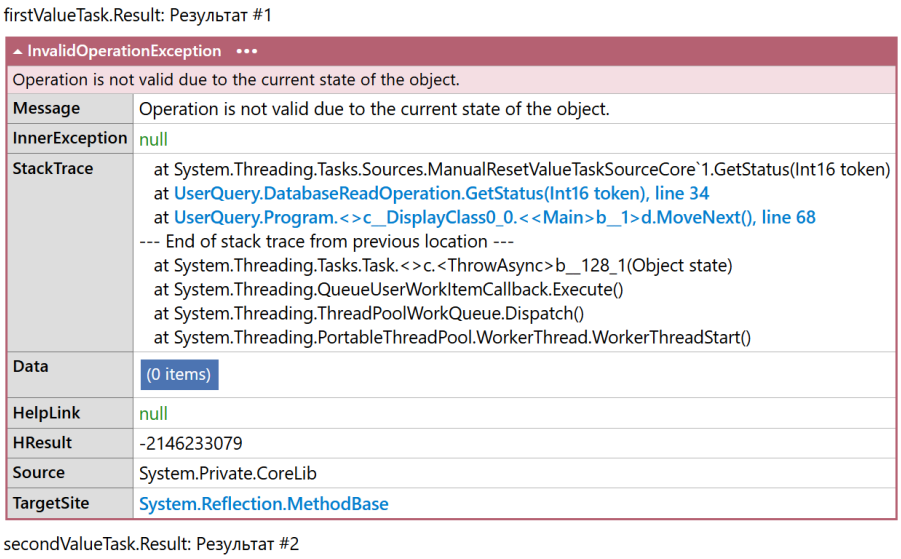

Рисунок 3 - Вывод программы с использованием ManualResetValueTaskSourceCore
DOI: https://doi.org/10.60797/IRJ.2025.157.40.3

На рисунке 3 наблюдается ошибка, потому что токен, переданный в метод *GetStatus*, больше не соответствует внутреннему состоянию объекта после его повторного использования из пула. Это объясняется следующим.

Сброс (*Reset*) очищает все внутренние состояния объекта, такие как захваченные контексты выполнения, продолжения, исключения и результат предыдущей операции. После вызова *_core.Reset* поле *_version* будет инкрементировано и свойство *_core.Version* вернет значение, которое не будет соответствовать первоначальному. Это важно для предотвращения ошибок при многократном ожидании завершения одной и той же задачи. В приведенном выше коде объект из пула был возвращен и повторно использован слишком рано, в то время, когда один из потоков все еще ожидал первоначальный объект.

Ограничения на использование ValueTask

Рассмотрим ограничения, нарушение которых может привести к некорректной работе программы, состояниям гонки и другим труднодиагностируемым ошибкам.

Первое – многократное ожидание (*await*) на одном и том же *ValueTask/ValueTask<TResult>*: поскольку внутренний объект может быть обработан и использоваться в другой операции, многократное ожидание может привести к тому, что объект уже будет занят другой задачей. Это может привести к некорректной работе программы. В отличие от *Task*, который всегда остается в завершенном состоянии и поддерживает многократные ожидания, *ValueTask* может перейти в незавершенное состояние при повторном использовании. В программе выше, где рассматривалось использование *ManualResetValueTaskSourceCore<string>* при реализации интерфейса *IValueTaskSource<string>*, представлен частный случай нарушения данного ограничения — параллельное ожидание одного и того же *ValueTask<string>*. Если один и тот же *ValueTask* ожидается в нескольких потоках одновременно, это создает условия для возникновения гонки.

Второе — использование метода *GetAwaiter().GetResult()* до завершения операции: в отличие от *Task*, который поддерживает блокирующий вызов до завершения задачи, реализация *IValueTaskSource* или *IValueTaskSource<TResult>* не обязана поддерживать блокировку до завершения операции. Поэтому вызов метода *GetResult* может привести к состояниям гонки и непредсказуемому поведению программы.

Указанные ограничения определяют важное правило: с экземплярами *ValueTask* или *ValueTask<TResult>* вы должны либо ожидать их напрямую с помощью *await* (возможно с *ConfigureAwait(false)*, указывающее, что после await выполнение кода не обязано возвращаться в исходный контекст синхронизации), либо сразу преобразовать их в *Task* с помощью метода *AsTask*, а затем больше не использовать исходный объект.

Из-за представленных ограничений, требующих соблюдения определенных рекомендаций, программисты реже прибегают к использованию *ValueTask<TResult>*. Поэтому в этом отношении, если производительность не перевешивает удобство использования, *Task<TResult>* остаются предпочтительными. Кроме того, есть небольшие затраты, связанные с возвращением *ValueTask<TResult>* вместо *Task<TResult>*. Например, по результатам тестирования реализаций функции Аккермана [15] ожидание *Task<TResult>* при определенных значениях параметров функции выполняется быстрее, чем ожидание *ValueTask<TResult>* (см. таблицу, рисунки 4 и 5 далее). Ниже представлен фрагмент кода программы [16], с помощью которого было проведено данное тестирование.

```
static void Main()
{
  BenchmarkRunner.Run<Ackermann>();
}
[IterationCount(100)]
[MemoryDiagnoser]
public class Ackermann
```

```
{
 [Params(1,2,3)]
 public int m;
 [Params(1,2,3)]
 public int n;
 [Benchmark(Baseline = true)]
 public int Baseline()
 {
  return AckermannFunc(m, n);
  int AckermannFunc(int m, int n) => (m, n) switch
  {
   (0, _) => n + 1,
   (_, 0) => AckermannFunc(m - 1, 1),
   _ => AckermannFunc(m - 1, AckermannFunc(m, n - 1)),
  };
 }
 [Benchmark]
 public ValueTask<int> ValueTask()
 {
  return AckermannFunc(m, n);
  async ValueTask<int> AckermannFunc(int m, int n) => (m, n) switch
  {
   (0, _) => n + 1,
   (_, 0) => await AckermannFunc(m - 1, 1),
   _ => await AckermannFunc(m - 1, await AckermannFunc(m, n - 1)),
  };
 }
 [Benchmark]
 public Task<int> Task()
 {
  return AckermannFunc(m, n);
  async Task<int> AckermannFunc(int m, int n) => (m, n) switch
  {
   (0, _) => n + 1,
   (_, 0) => await AckermannFunc(m - 1, 1),
   _ => await AckermannFunc(m - 1, await AckermannFunc(m, n - 1)),
  };
 }
}
```

Функция Аккермана обладает высокой степенью рекурсивной вложенности. В представленном коде в конечной точке рекурсии происходит синхронное завершение, в результате весь стек вызовов тоже разворачивается синхронно, и выполнение не уходит в асинхронность. В этом случае *ValueTask<int>* не создаёт *Task<int>*, и код остаётся полностью стековым, без выделений памяти в куче.

Результаты тестирования реализаций функции Аккермана с применением *.NET SDK 9.0.100* и библиотеки *BenchmarkDotNet v0.14.0* [17] представлены в таблице. Для тестирования была использована целевая платформа с характеристиками:

- процессор Intel Core i5-9300H (8 логических, 4 физических ядра)
- оперативная память DDR4 16 ГБ,
- операционная система Windows 11 (10.0.22631.4460),

Runtime=.NET 9.0.3 (9.0.325.11113), X64 RyuJIT AVX2.

Таблица 1 - Результаты тестирования реализаций функции Аккермана

DOI: https://doi.org/10.60797/IRJ.2025.157.40.4

Метод	Число рек. вызовов	m	n	Средн. арифм., нс.	Станд. откл., нс.	Кол-во сборок мусора в Gen0 на 1000 операций	Кол-во выделяемой памяти в куче за 1 вызов метода, байт
Baseline	4	1	1	3,506	0,0902	-	-
ValueTask				76,036	2,2531	-	-
IValueTaskSource				378,316	11,3795	0,0877	368
Task				50,700	1,0317	-	-
Baseline	6	1	2	7,313	0,0782	-	-

Метод	Число рек. вызовов	m	n	Средн. арифм., нс.	Станд. откл., нс.	Кол-во сборок мусора в Gen0 на 1000 операций	Кол-во выделяемой памяти в куче за 1 вызов метода, байт
ValueTask				106,601	6,3150	-	-
IValueTaskSource				503,147	28,4584	0,1354	568
Task				65,809	2,1491	-	-
Baseline	8	1	3	8,576	0,2664	-	-
ValueTask				136,545	3,2220	-	-
IValueTaskSource				678,346	19,3167	0,1831	768
Task				89,820	2,4432	-	-
Baseline	14	2	1	18,519	0,6279	-	-
ValueTask				270,178	7,9779	-	-
IValueTaskSource				1221,201	29,1151	0,3242	1360
Task				163,981	5,7236	-	-
Baseline	27	2	2	36,497	0,6551	-	-
ValueTask				482,350	11,3347	-	-
IValueTaskSource				3009,284	60,8933	0,6599	3376
Task				352,306	7,8184	-	-
Baseline	44	2	3	57,001	0,7175	-	-
ValueTask				814,358	28,9962	-	-
IValueTaskSource				4343,942	172,8679	1,5030	6296
Task				569,717	13,4053	0,0515	216
Baseline	106	3	1	135,360	1,1921	-	-
ValueTask				1961,375	62,9385	-	-
IValueTaskSource				10705,974	446,4759	4,0436	16937
Task				1534,464	40,6479	0,3777	1584
Baseline	541	3	2	674,350	5,9826	-	-
ValueTask				9459,547	181,8086		
IValueTaskSource				66247,500	5013,7841	21,8506	91701
Task				9222,128	378,4115	4,7455	19872
Baseline	2432	3	3	3293,043	44,9792	-	-
ValueTask				44116,709	1783,5449	-	-
IValueTaskSource				414282,242	17464,7995	96,6797	416848
Task				46579,047	1897,1332	30,3345	126864

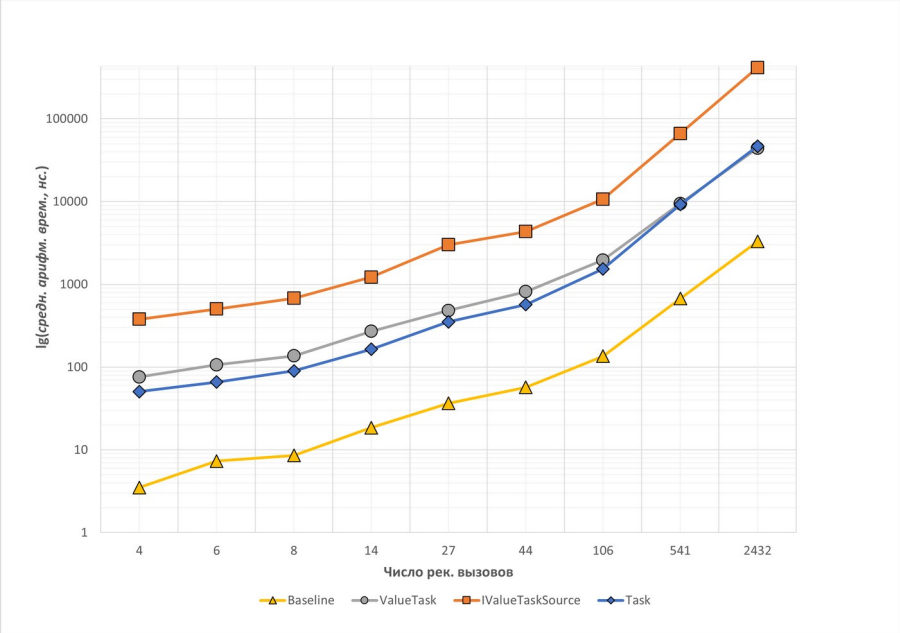

Рисунок 4 - Времена работы методов Baseline, ValueTask, IValueTaskSource, Task
DOI: https://doi.org/10.60797/IRJ.2025.157.40.5

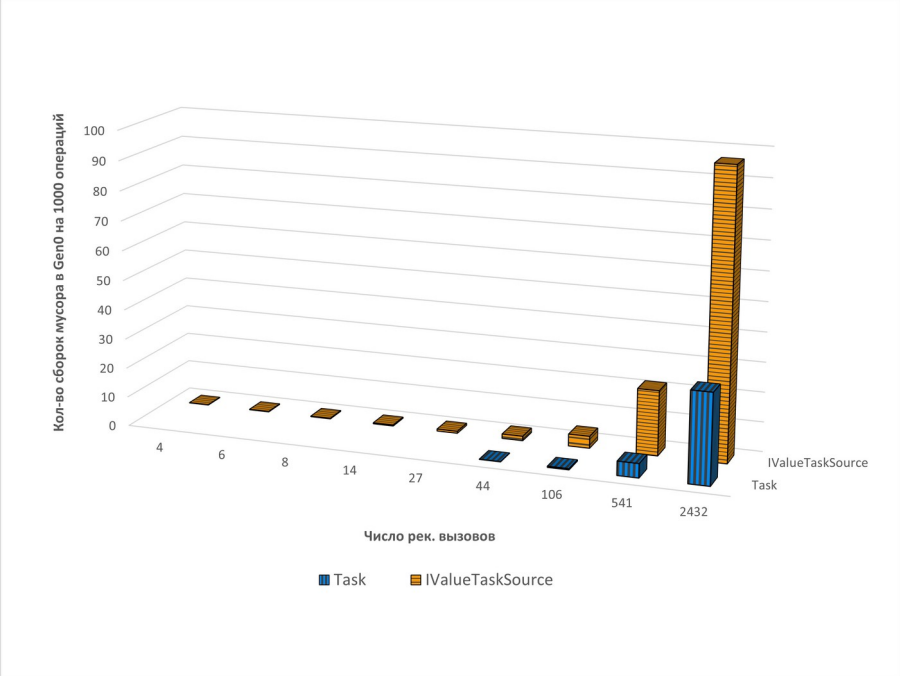

Рисунок 5 - Количество сборок мусора в ходе работы методов Task и IValueTaskSource
DOI: https://doi.org/10.60797/IRJ.2025.157.40.6

Тестирование показало, что при использовании *ValueTask<int>* не происходит выделения памяти в куче, тогда как *Task<int>* аллоцирует память при каждом *await*. Однако скорость работы с *Task<int>* при малых аллокациях выше, чем при использовании *ValueTask<int>*. Это обусловлено затратами, связанные с использованием *ValueTask<TResult>*:

- При передаче экземпляра структуры *ValueTask<TResult>* между методами происходит копирование структуры, что влечет дополнительные накладные расходы. В случае *Task<TResult>*, передается только указатель на объект в куче.

- *ValueTask<TResult>* может представлять: синхронный результат; ссылку на *Task<TResult>*; объект, реализующий интерфейс *IValueTaskSource<TResult>*. Это делает управление состоянием более сложным, поскольку метод, использующий *ValueTask<TResult>*, должен учитывать все три случая.

Адаптация функции Аккермана к интерфейсу *IValueTaskSource<T>* не дало выигрыша ни в производительности, ни в использовании управляемой памяти, так как использование глубокой рекурсии приводит к переполнению пула объектов.

Заключение

Асинхронное программирование в .NET представляет собой мощный инструмент для повышения производительности и отзывчивости приложений, особенно в сценариях, связанных с операциями ввода-вывода или длительными вычислениями.

По итогам проведенного анализа сформированы принципы оптимизации асинхронных операций:

- Выполнение предварительной проверки завершения асинхронной операции и, если результат уже доступен (например, из кеша), обеспечить синхронное завершение метода, чтобы избежать затрат на создание асинхронной инфраструктуры. Использование кешированных задач, таких как *Task.CompletedTask* или *Task.FromResult*, обеспечивает возвращение результата с минимальными накладными расходами.

- Применение методов, возвращающих *Task<bool>*, *Task<int>*, позволяет использовать кешированные экземпляры для значений, что исключает необходимость создания новых объектов.

- Использование структуры *ValueTask<T>* позволяет избежать выделения памяти в куче, оборачивая либо результат, либо объект задачи. Для использования данной структуры требуется соблюдение определенных ограничений, таких как избегание многократного ожидания одного и того же экземпляра.

- Реализация механизма управления асинхронными операциями через интерфейс *IValueTaskSource<T>* позволяет организовать пул повторно используемых объектов. Такой механизм требует ручного управления состояниями асинхронных операций, чтобы избежать ошибок.

Результаты тестирования в синхронном сценарии на примере функции Аккермана показали, что при малых аллокациях в памяти целесообразно применение *Task<T>*, так как данный тип не уступает в производительности *ValueTask<T>*. В асинхронном сценарии с возвращаемой структурой *ValueTask<TResult>* положительный результат потенциально достижим с применением интерфейса *IValueTaskSource<T>* и эффективным использованием пула объектов (т.е. не вызывая его переполнения).

Конфликт интересов

Не указан.

Рецензия

Борисов А.Н., Казанский национальный исследовательский технический университет им. А.Н. Туполева – КАИ, Казань Российская Федерация
DOI: https://doi.org/10.60797/IRJ.2025.157.40.7

Conflict of Interest

None declared.

Review

Borisov A.N., Kazan National Research Technical University named after A.N. Tupolev – KAI, Kazan Russian Federation
DOI: https://doi.org/10.60797/IRJ.2025.157.40.7

Список литературы / References

1. Гибадуллин Р.Ф. Потокобезопасные вызовы элементов управления в обогащенных клиентских приложениях / Р.Ф. Гибадуллин // Программные системы и вычислительные методы. — 2022. — № 4. — С. 1–19. — DOI: 10.7256/2454-0714.2022.4.39029.

2. Damyanov D. Using Asynchronous Programming in C# — Problems, Practical Tips and Scenarios: A Short Review / D. Damyanov, Z. Varbanov // 2024 5th International Conference on Communications, Information, Electronic and Energy Systems (CIEES). — Veliko Tarnovo, 2024. — P. 1–6. — DOI: 10.1109/CIEES62939.2024.10811272.

3. Albahari J. C# 12 in a Nutshell: The Definitive Reference / J. Albahari. — O'Reilly Media, 2023.

4. Прозорова А.П. Влияние на производительность приложения малого объема памяти и использование Garbage Collector (GC) / А.П. Прозорова, Е.В. Вершинин, А.Е. Потапов // Электронный журнал: наука, техника и образование. — 2019. — № 1. — С. 55–61.

5. Williams T. Effective .NET Memory Management: Build memory-efficient cross-platform applications using .NET Core / T. Williams. — Packt Publishing, 2024.

6. Гибадуллин Р.Ф. Неоднозначность результатов при использовании методов класса Parallel в рамках исполняющей среды. NET Framework / Р.Ф. Гибадуллин, И.В. Викторов // Программные системы и вычислительные методы. — 2023. — № 2. — С. 1–14.

7. Ashcraft A. Parallel Programming and Concurrency with C# 10 and .NET 6: A modern approach to building faster, more responsive, and asynchronous .NET applications using C# / A. Ashcraft. — Packt Publishing, 2022.

8. Исходный код класса Task в .NET Runtime // GitHub: репозиторий dotnet/runtime. — URL: https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/Threading/Tasks/Task.cs (дата обращения: 28.04.2025).

9. Исходный код класса MemoryStream в .NET Runtime // GitHub: репозиторий dotnet/runtime. — URL: https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/IO/MemoryStream.cs (дата обращения: 28.04.2025).

10. Price M.J. C# 7.1 and. NET Core 2.0—Modern Cross-Platform Development: Create powerful applications with. NET Standard 2.0, ASP. NET Core 2.0, and Entity Framework Core 2.0, using Visual Studio 2017 or Visual Studio Code / M.J. Price. — Packt Publishing Ltd, 2017.

11. Balliauw M. Pro NuGet. Apress / M. Balliauw, M., X. Decoster, X., J. Handley [et al.]. — 2012.

12. Alls J. High-Performance Programming in C# and. NET: Understand the nuts and bolts of developing robust, faster, and resilient applications in C# 10.0 and. NET 6 / J. Alls. — Packt Publishing Ltd, 2022.
13. Cleary S. Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming / S. Cleary. — O'Reilly Media, 2019.
14. Agafonov E. Multithreading with C# Cookbook / E. Agafonov. — Packt Publishing Ltd, 2016.
15. Козлов Д.А. Исследование функции Аккермана / Д.А. Козлов, Н.А. Говохин // Технические и математические науки. Студенческий научный форум. — 2018. — С. 96–110.
16. Исходный код программы AsyncAckermannBenchmark // GitHub: репозиторий CSharpCooking/AsyncAckermannBenchmark. — URL: https://github.com/CSharpCooking/AsyncAckermannBenchmark/blob/main/AsyncAckermannBenchmark/Program.cs (дата обращения: 28.04.2025).
17. Akinshin A. Pro. NET Benchmarking / A. Akinshin. — Apress, 2019.

Список литературы на английском языке / References in English

1. Gibadullin R.F. Potokobezopasnye vyzovy jelementov upravlenija v obogashhennyh klientskih prilozhenijah [Thread-safe calls of control elements in enriched client applications] / R.F. Gibadullin // Programmnye sistemy i vychislitel'nye metody [Software systems and computational methods]. — 2022. — № 4. — P. 1–19. — DOI: 10.7256/2454-0714.2022.4.39029. [in Russian]
2. Damyanov D. Using Asynchronous Programming in C# — Problems, Practical Tips and Scenarios: A Short Review / D. Damyanov, Z. Varbanov // 2024 5th International Conference on Communications, Information, Electronic and Energy Systems (CIEES). — Veliko Tarnovo, 2024. — P. 1–6. — DOI: 10.1109/CIEES62939.2024.10811272.
3. Albahari J. C# 12 in a Nutshell: The Definitive Reference / J. Albahari. — O'Reilly Media, 2023.
4. Prozorova A.P. Vlijanie na proizvoditel'nost' prilozhenija malogo ob'ema pamjati i ispol'zovanie Garbage Collector (GC) [Influence on application performance of small memory size and use of Garbage Collector (GC)]/ A.P. Prozorova, E.V. Vershinin, A.E. Potapov // Jelektronnyj zhurnal: nauka, tehnika i obrazovanie [Electronic Journal: Science, Technology and Education]. — 2019. — № 1. — P. 55–61. [in Russian]
5. Williams T. Effective .NET Memory Management: Build memory-efficient cross-platform applications using .NET Core / T. Williams. — Packt Publishing, 2024.
6. Gibadullin R.F. Neodnoznachnost' rezul'tatov pri ispol'zovanii metodov klassa Parallel v ramkah ispolnjajushhej sredy. NET Framework [Ambiguity of results when using methods of Parallel class within the executable environment .NET Framework] / R.F. Gibadullin, I.V. Viktorov // Programmnye sistemy i vychislitel'nye metody [Software Systems and Computational Methods]. — 2023. — № 2. — P. 1–14. [in Russian]
7. Ashcraft A. Parallel Programming and Concurrency with C# 10 and .NET 6: A modern approach to building faster, more responsive, and asynchronous .NET applications using C# / A. Ashcraft. — Packt Publishing, 2022.
8. Ishodnyj kod klassa Task v .NET Runtime [Task class source code in .NET Runtime] // GitHub: repozitorij dotnet/runtime [GitHub: dotnet/runtime repository]. — URL: https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/Threading/Tasks/Task.cs (accessed: 28.04.2025). [in Russian]
9. Ishodnyj kod klassa MemoryStream v .NET Runtime [MemoryStream class source code in .NET Runtime] // GitHub: repozitorij dotnet/runtime [GitHub: dotnet/runtime repository]. — URL: https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/IO/MemoryStream.cs (accessed: 28.04.2025). [in Russian]
10. Price M.J. C# 7.1 and. NET Core 2.0—Modern Cross-Platform Development: Create powerful applications with. NET Standard 2.0, ASP. NET Core 2.0, and Entity Framework Core 2.0, using Visual Studio 2017 or Visual Studio Code / M.J. Price. — Packt Publishing Ltd, 2017.
11. Balliauw M. Pro NuGet. Apress / M. Balliauw, M., X. Decoster, X., J. Handley [et al.]. — 2012.
12. Alls J. High-Performance Programming in C# and. NET: Understand the nuts and bolts of developing robust, faster, and resilient applications in C# 10.0 and. NET 6 / J. Alls. — Packt Publishing Ltd, 2022.
13. Cleary S. Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming / S. Cleary. — O'Reilly Media, 2019.
14. Agafonov E. Multithreading with C# Cookbook / E. Agafonov. — Packt Publishing Ltd, 2016.
15. Kozlov D.A. Issledovanie funkcii Akkermana [Study of the Ackerman function] / D.A. Kozlov, N.A. Govohin // Tehnicheskie i matematicheskie nauki. Studencheskij nauchnyj forum [Technical and Mathematical Sciences. Student Scientific Forum]. — 2018. — P. 96–110. [in Russian]
16. Ishodnyj kod programmy AsyncAckermannBenchmark [AsyncAckermannBenchmark source code] // GitHub: repozitorij CSharpCooking/AsyncAckermannBenchmark [GitHub: CSharpCooking/AsyncAckermannBenchmark repository]. — URL: https://github.com/CSharpCooking/AsyncAckermannBenchmark/blob/main/AsyncAckermannBenchmark/Program.cs (accessed: 28.04.2025). [in Russian]
17. Akinshin A. Pro. NET Benchmarking / A. Akinshin. — Apress, 2019.