66.59
Рейтинг

SimbirSoft
Лидер в разработке современных ИТ-решений на заказ

Подписаться

SSul
1 июл 2024 в 09:59

Глубокое погружение в CancellationToken: эффективное управление отменой в .NET

Средний 14 мин 16K

Блог компании SimbirSoft, Программирование*, .NET*, C#*

Обзор

Привет, Хабр! Меня зовут Давид, я C#-разработчик в SimbirSoft. В современном программировании эффективное управление ресурсами и контроль за выполнением задач становятся ключевыми аспектами для создания надежных и масштабируемых приложений. В C# одним из важнейших инструментов для достижения этих целей является механизм Cancellation Token. Эта концепция позволяет разработчикам изящно и безопасно управлять долгосрочными или ресурсоемкими операциями, обеспечивая возможность их отмены по требованию.

В этой статье мы погрузимся в мир Cancellation Token в C#, исследуя его роль, принципы работы и практическое применение. Мы обсудим, как эта технология позволяет разработчикам улучшить производительность и надежность их приложений, а также — как избежать распространенных ошибок при работе с асинхронными операциями. Подробно рассмотрим примеры кода, демонстрирующие использование Cancellation Token в различных сценариях, что сделает наше обсуждение не только теоретически насыщенным, но и довольно практичным для программистов любого уровня.

Основное назначение Cancellation Token

Cancellation Token в C# — это эффективный инструмент для управления операциями, которые могут затянуться или требуют асинхронного выполнения. В современном программировании, где время реакции и гибкость приложений играют ключевую роль, наличие механизма для остановки и корректировки длительных процессов становится не просто полезным, а критически важным.

Cancellation Token предоставляет разработчикам элегантный способ для контроля над такими задачами, обеспечивая создание отзывчивых и адаптивных приложений.

> **Рассмотрим практический пример:** пользователь начинает загрузку большого файла, но решает ее отменить до завершения. Без механизма Cancellation Token процесс загрузки продолжался бы, потребляя ценные ресурсы и снижая общую производительность системы. Аналогичная ситуация возникает при необходимости изменения параметров обработки данных: без возможности быстрой остановки текущей задачи приложение теряет в гибкости и скорости реагирования.

В контексте многопоточных приложений управление Cancellation Token приобретает еще большую значимость. Оно позволяет безопасно координировать работу множества параллельных задач, избегая проблем, связанных с блокировками и «мертвыми» состояниями потоков. Это не только улучшает производительность, но и повышает общую надежность приложений, делая их более устойчивыми к изменчивым условиям эксплуатации.

Принцип работы Cancellation Token

Cancellation Token в C# реализуется через два основных типа: структуру `CancellationToken` и класс `CancellationTokenSource`.

Объект `CancellationTokenSource` выступает в роли командного центра. Он предоставляет get-only свойство Token, через которое можно получить связанный объект `CancellationToken`. Управление процессом отмены осуществляется через методы `Cancel()` и `CancelAfter()`. Когда наступает момент остановить операцию, `CancellationTokenSource` действует как регулятор, инициируя процесс отмены через вызов метода `Cancel()`. Это приводит к тому, что все связанные с этим источником токены отправляют сигналы об отмене своим асинхронным задачам. Аналогичным образом действует метод `CancelAfter()`, который выполняет то же действие через указанное в параметрах время.

`CancellationToken`, в свою очередь, представляет собой запрос на отмену операции. Его можно передавать асинхронным методам через параметры, что позволяет им отслеживать статус отмены [1] [6].

```
CancellationTokenSource cts = new();
CancellationToken token = cts.Token;

Task myTask = Task.Run(() => DoWork(token), token);

cts.Cancel();
// cts.CancelAfter(1000);
```

Здесь следует уточнить механизм работы CancellationToken по замечанию @aegoroff. Несмотря на то, что токен представляет собой структуру, по семантике он ведёт себя как класс. Сделано это было разработчиками следующим образом: в `CancellationToken` есть единственное поле

```
private CancellationTokenSource m_source;
```

Исходники

Это поле используется в проверке на равенство токенов и при копировании, так что копия токена ведёт себя как оригинал и тем самым неотличима от него. В частности, если оригинал токена отменён, то его копия — тоже.

Причина же такой нестандартной конструкции кроется в эффективности работы с памятью: если бы `CancellationToken` был реализован как класс, каждый раз при создании нового токена

происходила бы аллокация памяти в куче. Однако для структуры в большинстве случаев аллокации происходят в стеке (если токен является локальной переменной, а в куче только тогда, когда структура является частью другого объекта, хранящегося в куче), что, во-первых, быстрее само по себе, а во-вторых, позволяет серьезно уменьшить нагрузку на `Garbage Collector`. В основном именно по этим причинам `CancellationToken` был создан таким, какой он есть сейчас (и каким мы его любим).

Кооперативная отмена

Первое, что необходимо понимать о Cancellation Token — то, что он работает на основе кооперативной отмены. Это означает, что задача, которую нужно отменить, сама должна регулярно проверять состояние токена и корректно реагировать на запрос отмены.

Когда дело доходит до отмены задачи, система не вмешивается автоматически, как некий всемогущий регулятор — задача должна «сотрудничать», чтобы отмена была безопасной и эффективной. Это похоже на водителя, который следит за дорожными знаками и сигналами светофора, чтобы знать, когда пора остановиться или изменить маршрут. Если задача игнорирует эти «дорожные знаки» отмены, она продолжит свое выполнение, даже если токен уже подает сигнал о необходимости остановки.

Этот подход требует от разработчиков осознанного и аккуратного внедрения логики проверки и реагирования на Cancellation Token в свои асинхронные методы. Это означает, что каждая задача должна быть написана с учетом потенциальной необходимости ее безопасной и эффективной отмены [7].

Исключения

Еще одним важным пунктом при работе с Cancellation Token в C# является понимание того, как обрабатывать исключения при отмене операций. В основном здесь мы сталкиваемся с двумя типами исключений: `OperationCanceledException` и `TaskCanceledException`.

Рассмотрим их подробнее:

1. `OperationCanceledException` вызывается при явной отмене операции через метод `ThrowIfCancellationRequested()`.

```
async static void DoWorkWithOperationCancel(CancellationToken cancellationToken)
{
    try
    {
        for (int i = 1; i != 10; i++)
        {
            cancellationToken.ThrowIfCancellationRequested();

            Thread.Sleep(200);

            Console.WriteLine($"Work {i} done.");
        }
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Operation was canceled");
    }
}
```

2. `TaskCanceledException` чаще всего возникает во время запроса на отмену в ходе выполнения встроенного метода, который изначально поддерживает возможность отмены операций (например, `Task.Delay()`).

Этот тип исключения, являясь наследником OperationCanceledException, выступает как более конкретный сигнал о том, что задача, выполняемая в рамках `Task`, была отменена [6]. Тем не

менее, некоторые API вызывают OperationCanceledException, даже если имеют дело с отмененными задачами, поэтому для стабильной и надежной работы рекомендуется всегда обрабатывать общий родительский тип исключения OperationCanceledException. [7]

```
async static void DoWorkWithTaskCancel(CancellationToken cancellationToken)
{
    try
    {
        for (int i = 1; i != 10; i++)
        {
            await Task.Delay(200, cancellationToken);

            Console.WriteLine($"Work {i} done.");
        }
    }
    catch (TaskCanceledException)
    {
        Console.WriteLine("Task was canceled");
    }
}
```

В случаях, когда не происходит внутренней обработки исключения, задача может перейти в состояние `TaskStatus.Canceled`, если ее выполнение было прервано, или `TaskStatus.RanToCompletion`, если задача завершилась до того, как был подан сигнал об отмене, или если вместо генерации исключения построена другая логика отмены операции. Это помогает программистам понимать текущее состояние задач и правильно обрабатывать различные сценарии завершения операций.

Реализация в WebAPI

В контексте разработки Web API на платформе ASP.NET Core обработка исключений играет ключевую роль в создании надежных и устойчивых к ошибкам приложений. Среди разнообразных подходов к обработке исключений можно выделить `Middleware` и `Filter` как наиболее эффективные инструменты для управления ошибками, связанными с отменой задач. Рассмотрим их подробнее:

1. `Middleware:` является ключевым компонентом в архитектуре Web API, обеспечивающим гибкую и мощную систему для обработки запросов. Этот инструмент служит центральным узлом в конвейере обработки запросов, позволяя разработчикам внедрять разнообразные функции, включая логирование, управление кэшем, динамическую модификацию ответов и многое другое. В данном случае нам интересна его способность перехватывать и обрабатывать исключения. Создание специального класса `Middleware` с методом `Invoke()` и использование конструкции `try-catch` для обработки исключений `OperationCanceledException` и `TaskCanceledException` позволяет централизованно управлять ошибками отмены задач в приложении [3]. Пример кода демонстрирует, как логировать данные события и минимизировать их влияние на работу приложения.

```
namespace WebApp.Middleware;

public class TaskCancellationHandlingMiddleware(RequestDelegate next, ILogger<TaskCancellationH
{
    public async Task Invoke(HttpContext context)
    {
        try
        {
            await next(context);
        }
        catch (Exception _) when (_ is OperationCanceledException or TaskCanceledException)
        {
            logger.LogInformation("Task canceled");
        }
```

```
    }
}
```

Регистрация `Middleware` в конвейере обработки запросов гарантирует, что каждый запрос будет проверен на наличие этих исключений.

```
app.UseMiddleware<TaskCancellationHandlingMiddleware>();
```

2. `Filter.` Использование фильтров исключений предоставляет мощный механизм для обработки ошибок на уровне контроллеров и действий. Создание класса, наследующего от `ExceptionFilterAttribute` , и переопределение метода `OnException` позволяет определить специфическую логику обработки для исключений, возникающих в результате отмены задач. Этот подход не только упрощает обработку исключений, но и позволяет настраивать отправляемые клиенту HTTP-ответы, что улучшает взаимодействие с пользователем при возникновении ошибок [3].

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

namespace WebApp.Filters;

public class TaskcanceledExceptionFilter(ILoggerFactory loggerFactory) : ExceptionFilterAttribu
{
    private readonly ILogger _logger = loggerFactory.CreateLogger<TaskcanceledExceptionFilter>(

    public override void OnException(ExceptionContext context)
    {
        if (context.Exception is OperationCanceledException or TaskCanceledException)
        {
            _logger.LogInformation("Request was canceled");
            context.ExceptionHandled = true;
            context.Result = new StatusCodeResult(400);
        }
    }
}
```

Регистрация фильтра исключений в конфигурации контроллеров обеспечивает его применение ко всем действиям в рамках приложения.

```
builder.Services.AddControllers(opt => opt.Filters.Add<TaskcanceledExceptionFilter>());
```

Условия применения Cancellation Token

В данном разделе нашего обзора рассмотрим сценарии, где применение этого инструмента оказывается не только полезным, но и необходимым, а также случаи, в которых его использование может быть излишним или нежелательным.

Сценарии, в которых Cancellation Token становится вашим союзником:

1. **Длительные операции.** Если ваше приложение выполняет операции, которые занимают значительное время (например, загрузка данных, обработка больших объемов информации), тогда использование Cancellation Token позволяет предоставить пользователю контроль над этими процессами и сохранить ресурсы для более важных задач.

2. **Асинхронные и многопоточные операции.** В среде, где несколько задач выполняются параллельно, Cancellation Token обеспечивает эластичность управления, позволяя оперативно останавливать отдельные задачи по требованию, тем самым улучшая отклик приложения и предотвращая потенциальные проблемы с производительностью.

3. **Операции с возможностью отмены пользователем.** В интерактивных приложениях, где пользователи имеют возможность отменить текущие операции (например, загрузку файла или выполнение поискового запроса), наличие механизма отмены через Cancellation Token является ключевым для создания гибкого и отзывчивого пользовательского интерфейса.

Случаи, когда Cancellation Token может оказаться не на своем месте:

1. **Краткосрочные или простые операции.** Для очень быстрых или простых операций внедрение данного инструмента может быть излишним. В таких случаях добавление механизма отмены может привести к ненужному усложнению кода и снижению производительности.

2. **Операции, где отмена неприемлема.** Также не следует применять этот механизм в таких типах операций, как критические задачи обновления или записи данных, где отмена может привести к неконсистентному или поврежденному состоянию.

3. **Задачи с обязательным выполнением.** В случаях, когда задача должна быть доведена до конца без возможности отмены (как важные финансовые транзакции или операции с критически важными данными), применение Cancellation Token не только нежелательно, но и потенциально опасно.

Шаблоны проектирования и архитектурные принципы

Использование Cancellation Token рекомендуется так же в некоторых паттернах проектирования, включая некоторые классические паттерны GoF (Command, State, Strategy), архитектурные стили (Produser-Consumer) и корпоративные шаблоны Фаулера (Unit of work, Repository):

1. **Command**

Паттерн Command инкапсулирует запросы в виде объектов, позволяя гибко ими управлять. Cancellation Token в этом случае можно использовать для отмены уже запущенных команд, особенно если они выполняют длительные операции [2] [5]. Это добавляет дополнительный уровень контроля над выполнением команд.

> **Пример:** в приложении для редактирования изображений пользователь начинает процесс фильтрации изображения, но решает отменить операцию. Каждая команда фильтрации поддерживает отмену через Cancellation Token.

```
public class FilterCommand : ICommand
{
    public void Execute(CancellationToken cancellationToken)
    {
        // Процесс фильтрации, который регулярно проверяет cancellationToken
    }
}
```

2. **State**

Паттерн State используется для управления состоянием объекта. Cancellation Token может быть полезен для прерывания операций в одном состоянии, когда происходит переход в другое состояние [2] [5]. Это особенно важно, если операции в текущем состоянии являются длительными или асинхронными.

> **Пример:** в игре персонаж может находиться в разных состояниях (бег, прыжок, покой). Если игрок внезапно переключает состояние (например, с бега на прыжок), Cancellation Token

> используется для остановки текущего состояния.

```
public class RunningState : IState
{
    public void Enter(CancellationToken cancellationToken)
    {
        Task.Run(() =>
        {
            while (!cancellationToken.IsCancellationRequested)
            {
                // Логика бега
            }
        }, cancellationToken);
    }
}
```

3. **Strategy**

Паттерн Strategy предоставляет механизм для выбора алгоритма выполнения во время выполнения программы. Cancellation Token в этом контексте позволяет отменить выполнение текущей стратегии, если условия изменяются или если выбрана новая стратегия [2] [5]. Это особенно полезно, когда разные стратегии выполняют длительные или ресурсоемкие задачи.

> **Пример:** в приложении для анализа данных пользователь может выбрать разные стратегии анализа. Если пользователь решает изменить стратегию анализа во время ее выполнения, Cancellation Token используется для остановки текущего процесса анализа перед переключением на новую стратегию.

```
public class DataAnalyzer
{
    private IAnalysisStrategy _currentStrategy;

    public void SetStrategy(IAnalysisStrategy strategy, CancellationToken cancellationToken)
    {
        cancellationToken.Cancel(); // Отмена текущей стратегии
        _currentStrategy = strategy;
        // Запуск новой стратегии
    }
}
```

4. **Producer-Consumer**

Producer-Consumer не является отдельным паттерном, он скорее представляет собой архитектурный стиль, согласно которому одни потоки генерируют данные (producers), а другие потоки их обрабатывают (consumers). Cancellation Token позволяет безопасно и эффективно прерывать этот процесс [4], например, при закрытии приложения или когда данные больше не требуются, предотвращая таким образом утечки ресурсов и блокировки.

> **Пример:** в приложении для обработки видео Producer генерирует кадры видео, а Consumer их обрабатывает. Если пользователь отменяет обработку, Cancellation Token используется для остановки как Producer, так и Consumer.

```
public void StartProcessing(CancellationToken cancellationToken)
{
    Task producer = ProduceVideoFrames(cancellationToken);
    Task consumer = ConsumeVideoFrames(cancellationToken);
```

```
    // ...
}
```

5. **Unit of Work**

Паттерн Unit of Work группирует несколько операций в одну логическую единицу, которая должна быть выполнена целиком. Использование Cancellation Token позволяет отменить всю единицу работы, если возникает ошибка или если пользователь решает прервать процесс [3]. Это обеспечивает целостность данных и предотвращает частичное выполнение операций, что может быть критически важно в приложениях, работающих с базами данных или выполняющих сложные транзакции.

> **Пример:** в банковской системе, где несколько операций должны быть выполнены как одна транзакция, Cancellation Token используется для отмены всей транзакции, если одна из операций не может быть завершена успешно.

```
public void ProcessTransaction(UnitOfWork unitOfWork, CancellationToken cancellationToken)
{
    foreach (var operation in unitOfWork.Operations)
    {
        if (cancellationToken.IsCancellationRequested)
        {
            // Отмена остальных операций
            return;
        }
        operation.Execute();
    }
}
```

6. **Repository**

Паттерн Repository используется для абстрагирования слоя доступа к данным в приложениях. Включение Cancellation Token в операции репозитория позволяет отменять запросы к базе данных или другим хранилищам данных, особенно в сценариях с длительными запросами или операциями [3]. Это улучшает отзывчивость приложения и предотвращает замораживание интерфейса пользователя при выполнении сложных запросов.

> **Пример:** в приложении, которое позволяет пользователям выполнять сложные запросы к базе данных, Cancellation Token используется для отмены запроса, если пользователь решает, что он больше не нужен или занимает слишком много времени.

```
public class DataRepository
{
    public async Task<List<Data>> QueryDataAsync(QueryCriteria criteria, CancellationToken can
    {
        return await dbContext.Data.Where(criteria.Filter).ToListAsync(cancellationToken);
    }
}
```

Возможные угрозы безопасности и утечки памяти при использовании Cancellation Token

Несмотря на все преимущества использования Cancellation Token, неправильное его применение может привести к серьезным проблемам, таким как угрозы безопасности и утечки памяти [7] [8]. Рассмотрим наиболее распространенные из них и способы их предотвращения:

1. **Утечки памяти из-за неправильной обработки токена отмены.**

Проблема: если Cancellation Token не проверяется должным образом в длительных или асинхронных операциях, задачи могут продолжать выполняться в фоне, даже после того как они становятся ненужными (так называемые «зомби-процессы»). Это может привести к утечкам памяти и занятию системных ресурсов.

Решение: регулярно проверяйте состояние Cancellation Token внутри асинхронных задач и корректно завершайте операции при обнаружении запроса на отмену.

2. **Блокировки и взаимоблокировки.**

Проблема: неправильная синхронизация или обработка отмены может привести к блокировкам и взаимоблокировкам, особенно в многопоточных приложениях, что угрожает стабильности и производительности приложения.

Решение: используйте конструкции синхронизации (например, `lock` в C#) и тщательно проектируйте логику отмены, чтобы избежать блокировок.

3. **Некорректное управление ресурсами.**

Проблема: отмена операции может привести к ситуации, когда ресурсы (например, файлы или сетевые соединения) не освобождаются должным образом, что может вызвать утечки ресурсов и уязвимости безопасности.

Решение: используйте конструкции `try-finally` или `using` , которые гарантируют, что ресурсы будут освобождены, даже если операция была отменена.

4. **Непредвиденное поведение и ошибки обработки исключений.**

Проблема: некорректная обработка `TaskCanceledException` и других исключений, связанных с отменой задачи, может привести к непредвиденному поведению приложения.

Решение: явно обрабатывайте исключения, связанные с отменой задач, и предусмотрите логику для безопасного завершения операции в случае ее отмены.

5. **Уязвимости при общем использовании CancellationTokenSource.**

Проблема: использование одного и того же экземпляра `CancellationTokenSource` для множества задач может привести к ситуациям, когда отмена одной задачи неожиданно влияет на другие задачи, использующие тот же токен.

Решение: внимательно отслеживайте область видимости и жизненный цикл `CancellationTokenSource` . Избегайте его общего использования там, где это может привести к нежелательным взаимодействиям между задачами.

6. **Проблемы с производительностью при частой проверке токена.**

Проблема: чрезмерно частая проверка состояния `CancellationToken` внутри интенсивных циклов может негативно сказаться на производительности приложения.

Решение: найдите баланс между необходимостью своевременной реакции на отмену и эффективностью выполнения задачи. В некоторых случаях может быть уместно использовать более длительные интервалы между проверками состояния токена.

Заключение

Подводя итоги, стоит подчеркнуть — хотя Cancellation Token является мощным инструментом для управления асинхронными операциями в C#, его неправильное использование может привести к ряду нежелательных последствий, включая утечки памяти и блокировки. Однако, придерживаясь лучших практик безопасности, внимательного управления ресурсами и аккуратной обработки

исключений, разработчики могут избежать этих проблем, значительно повышая надежность и производительность своих приложений.

Источники

1. https://mitesh1612.github.io/blog/2022/01/20/cancellation-tokens
2. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2015. — 368 с.: ил. ISBN 978-5-496-00389-6
3. Фаулер, Мартин (1963-). Шаблоны корпоративных приложений [Текст] / Мартин Фаулер при участии Дейвида Райса [и др.]. - Испр. изд. - Москва [и др.] : Вильямс, 2017. - 539, [4] с. : ил., табл.; 24 см.; ISBN 978-5-8459-1611-2 : 300 экз
4. Хоп, Грегор, Вульф, Бобби. Шаблоны интеграции корпоративных приложений. : Пер. с англ. М. : ООО “И.Д. Вильямс”, 2007. 672 с. : ил. Парал. тит. англ. ISBN 978-5-8459-1146-9 (рус.)
5. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, & Michael Stal. Pattern-Oriented Software Architecture. — Wiley, 1996, ISBN 0471958697
6. Джон Скит, C# для профессионалов. Тонкости программирования. — Вильямс, 2014. — 608 с.: ил. ISBN 978-5-8459-1909-0
7. Стивен Клири, Конкурентность в C#. Асинхронное, параллельное программирование. — Питер, 2020. — 272 с.: ил. ISBN 978-5-4461-1572-3
8. Мартин Роберт, Чистый код: создание, анализ и рефакторинг. Библиотека программиста. — Питер, 2020. — 464 с.: ил. ISBN 978-5-4461-0960-9

Спасибо за внимание!

Больше авторских материалов для backend-разработчиков от моих коллег читайте в соцсетях SimbirSoft – ВКонтакте и Telegram.

Теги: .net, cancellationtoken, cancellationtokensource, c# .net, производительность

Хабы: Блог компании SimbirSoft, Программирование, .NET, C#