СПОСОБЫ ПАРАЛЛЕЛЬНОГО ВЫПОЛНЕНИЯ ЗАДАЧ В C#

Thread

BackgroundWorker

Task

Thread. Особенности и применение.

Потоки удобны в применении, когда требуется:

- Выполнять операции в фоновом режиме без необходимости в ожидании их завершения
- Ускорить выполнение операции путем ее распараллеливания по потокам
- Задать потоку имя, чтобы было удобнее искать его в отладчике
- Поставить повышенный или пониженный приоритет потоку

Thread. Основные свойства.

Пространство имен - System.Threading;

- **CurrentContext** позволяет получить контекст, в котором выполняется поток
- **CurrentThread** возвращает ссылку на выполняемый поток
- **IsAlive** указывает, работает ли поток в текущий момент
- **IsBackground** указывает, является ли поток фоновым
- **Name** содержит имя потока
- **Priority** хранит приоритет потока - значение перечисления ThreadPriority
- **ThreadState** возвращает состояние потока - одно из значений перечисления ThreadState

Thread. Некоторые методы.

- **Sleep** останавливает поток на определенное количество миллисекунд
- **Abort** уведомляет среду CLR о том, что надо прекратить поток, однако прекращение работы потока происходит не сразу, а только тогда, когда это становится возможно. Для проверки завершенности потока следует опрашивать его свойство ThreadState
- **Interrupt** прерывает поток на некоторое время
- **Join** блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
- **Resume** возобновляет работу ранее приостановленного потока
- **Start** запускает поток
- **Suspend** приостанавливает поток

Thread. Создание и запуск.

Для запуска нового потока нам надо определить задачу в приложении, которую будет выполнять данный поток. Для этого мы можем ввести добавить новый метод, производящий какие-либо действия.

```
private void calcLog() //метод для потока
{
    double logValue = 0;
    for (int i = 1; i < 100000000; i++)
    {
        logValue = Math.Log10(i);
    }
    MessageBox.Show("All.");
}
```

Thread. Создание и запуск.

Для создания нового потока используется делегат **ThreadStart**, который получает в качестве параметра метод, который мы определили выше.

Чтобы запустить поток, вызывается метод **Start**.

```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    Thread thread = new Thread(new ThreadStart(calcLog)); //создаем поток
    thread.Start(); //запускаем поток
}
```

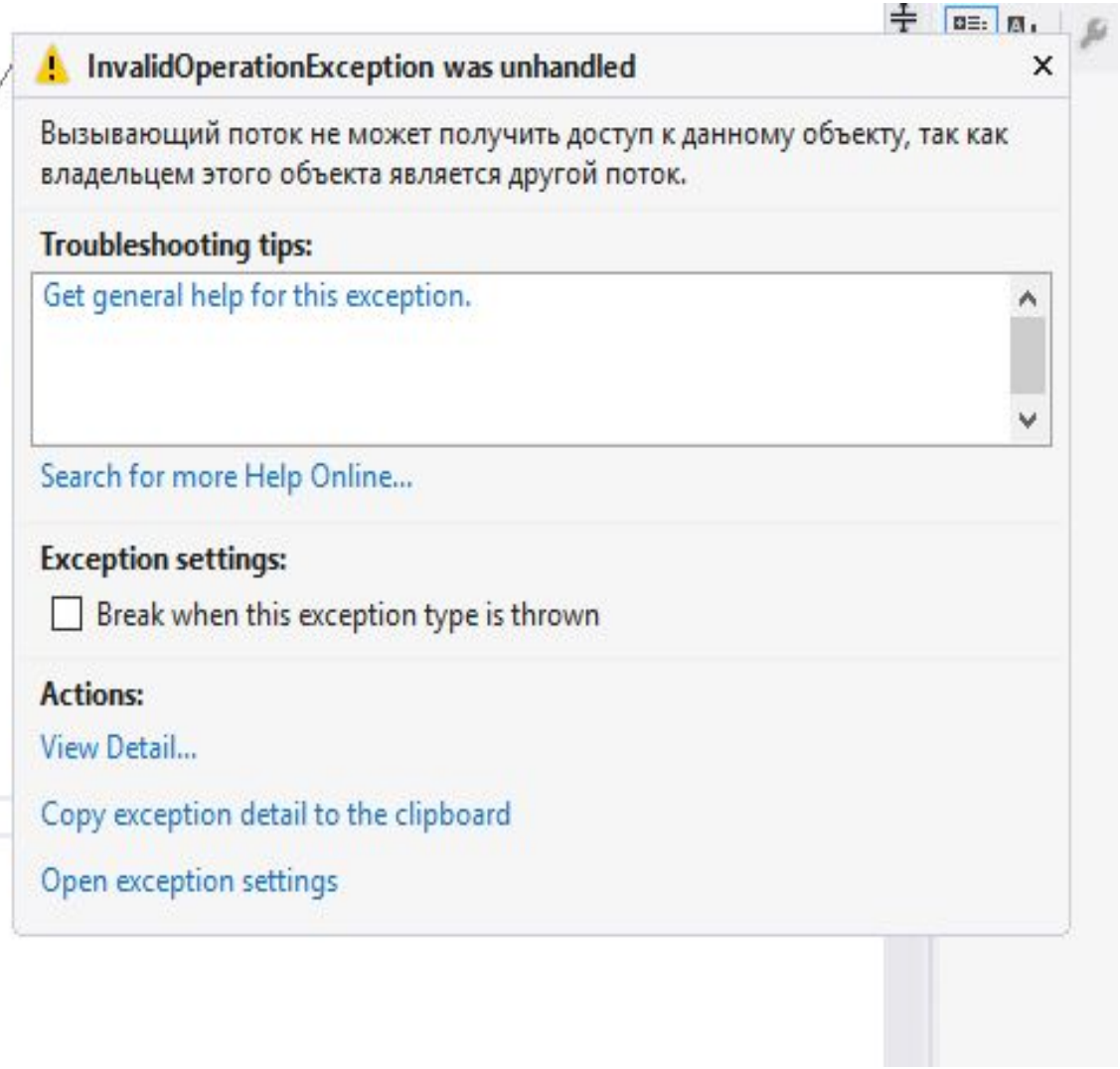
Thread. Передача параметров в поток.

Для этой цели используется делегат **ParameterizedThreadStart**. Его действие похоже на функциональность делегата **ThreadStart**.

```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    Thread myThread = new Thread(new ParameterizedThreadStart(calcLog));
    myThread.Start(120);
}

private void calcLog(object startNum)
{
    int start = (int)startNum;
    double logValue = 0;
    for (int i = start; i < 100000000; i++)
    {
        logValue = Math.Log10(i);
    }
    MessageBox.Show("All.");
}
```

Thread. Доступ к элементам окна.

```
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void Button_Click_1(object sender, RoutedEventArgs e)
    {
        Thread myThread = new Thread(new ParameterizedThreadStart(calcLog));
        myThread.Start(120);
    }

    private void calcLog(object startNum)
    {
        int start = (int)startNum;
        double logValue = 0;
        for (int i = start; i < 100000000; i++)
        {
            logValue = Math.Log10(i);
            textBoxValue.Text = logValue.ToString();
        }
        MessageBox.Show("All.");
    }
}
```



Thread. Доступ к элементам окна. Invoke.

```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    Thread myThread = new Thread(new ParameterizedThreadStart(calcLog));
    myThread.Start(120);
}

private void calcLog(object startNum)
{
    int start = (int)startNum;
    double logValue = 0;
    for (int i = start; i < 100000000; i++)
    {
        logValue = Math.Log10(i);
        textBoxValue.Dispatcher.Invoke(delegate { textBoxValue.Text = logValue.ToString(); });
    }
    MessageBox.Show("All.");
}
```

BackgroundWorker. Возможности.

BackgroundWorker – класс из пространства имен **System.ComponentModel** для управления рабочими потоками. Он обеспечивает следующие возможности:

- Флаг отмены операции для завершения потока без использования **Abort**.
- Стандартный протокол для сообщений о ходе выполнения операции, ее завершении и отмене.
- Обработка исключений в рабочем потоке.
- Возможность обновления элементов управления окна в процессе выполнения или при завершении операции.

Вывод**: можно не добавлять **try/catch** в рабочий метод и обновлять элементы управления без использования **Control.Invoke.

BackgroundWorker. Прогресс выполнения.

Чтобы добавить отображение выполнения операции:

- Установите свойство **WorkerReportsProgress** в true.
- Периодически вызывайте **ReportProgress** из обработчика **DoWork** с указанием количества “выполненных процентов”
- Обрабатывайте событие **ProgressChanged**, запрашивая свойство **ProgressPercentage** его аргумента.

*Код в обработчике **ProgressChanged** может свободно обращаться к элементам управления UI так же, как и в **RunWorkerCompleted**. Обычно это нужно для обновления индикатора прогресса.*

BackgroundWorker. Отмена операции.

Чтобы иметь возможность отмены операции:

- Установите свойство **WorkerSupportsCancellation** в true.
- Периодически проверяйте свойство **CancellationPending** в обработчике **DoWork** – если оно установлено в true, установите свойство **Cancel** аргумента **DoWorkEventArgs** в true и сделайте return
- Для запроса отмены операции вызывайте **CancelAsync.**

BackgroundWorker. Создание и запуск.

```
BackgroundWorker bw;
bw = new BackgroundWorker();

bw.WorkerReportsProgress = true; //поддержка отображения прогресса
bw.WorkerSupportsCancellation = true; //поддержка отмены операции
bw.DoWork += bw_DoWork; //выполнение операции
bw.ProgressChanged += bw_ProgressChanged; //отображение прогресса
bw.RunWorkerCompleted += bw_RunWorkerCompleted; //завершение операции

bw.RunWorkerAsync(null);
```

BackgroundWorker. DoWork.

```
private void bw_DoWork(object sender, DoWorkEventArgs e)
{
    for (int i = 0; i <= 100; i += 20)
    {
        if (bw.CancellationPending) //если отменили операцию
        {
            e.Cancel = true;
            return;
        }
        bw.ReportProgress(i); //прогресс выполнения
        Thread.Sleep(1000);
    }
    e.Result = 123;    // будет передано в RunWorkerCompleted
}
```

```
private void bw_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    if (e.Cancelled) //если отменили
    {
        MessageBox.Show("Работа BackgroundWorker была прервана пользователем!");
    }
    else if (e.Error != null) //если были ошибки
    {
        MessageBox.Show("Worker exception: " + e.Error);
    }
    else //если все хорошо
    {
        MessageBox.Show("Работа закончена успешно. Результат - " + e.Result + ". ");
    }
}

private void bw_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
    progressBar.Value = e.ProgressPercentage; //прогресс в прогрессбар
}
```

Task. Особенности.

Пространство имен - **System.Threading.Tasks**

- Практически нет накладных расходов при создании
- Могут возвращать результат
- Поддерживают отмену выполнения задачи
- Возможно асинхронное выполнение

Task. Создание и запуск.

```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    Task task = new Task(calcLog);
    task.Start();
}

private void calcLog()
{
    double logValue = 0;
    for (int i = 1; i < 100000000; i++)
    {
        logValue = Math.Log10(i);
        textBoxValue.Dispatcher.Invoke(delegate { textBoxValue.Text = logValue.ToString(); });
    }
    MessageBox.Show("All.");
}
```

Task. Async/Await.

- Ключевое слово **async** указывает, что метод или лямбда-выражение может выполняться асинхронно. А оператор **await** позволяет остановить текущий поток, пока не завершится работа метода, помеченного как **async**.
- Эти операторы используются вместе для создания асинхронного метода. Такой метод, определенный с помощью модификатора **async** и содержащий одно или несколько выражений **await**, называется **асинхронным методом**.

Task. Async/Await.

```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    WebClient wClient = new WebClient();
    byte[] info = wClient.DownloadData("http://fias.nalog.ru/Public/Downloads/Actual/base.arj");
    MessageBox.Show("Done.");
}
//-------------------------------------------------------------------------------
private async void Button_Click_1(object sender, RoutedEventArgs e)
{
    WebClient wClient = new WebClient();
    byte[] info = await wClient.DownloadDataTaskAsync("http://fias.nalog.ru/Public/Downloads/Actual/base.arj");
    MessageBox.Show("Done.");
}
```