C# Professional

Работа контекста синхронизации с async await.

Работа контекста синхронизации в async await.

План урока

1) Использование async await в WPF.
2) Класс SynchronizationContext – контекст синхронизации.
3) Продолжения оператора await.
4) Управление ожиданием.
5) Асинхронные лямбда выражения.
6) Модификатор async для void.

Использование async await в WPF

Использование ключевых слов async await упрощает асинхронное программирование в приложениях, написанных по шаблону WPF.

Сложность асинхронного кода для приложений WPF была всегда в том, чтобы обращаться к элементам управления из потока пользовательского интерфейса.

Приходилось либо постоянно разными способами передавать данные из одного потока в другой, либо блокировать поток пользовательского интерфейса на время выполнения. Трудностей в этом не было, но код становился грязным, тяжелым к рассмотрению и изменениям.

Использование ключевых слов async await делают работу с WPF достаточно простой. Асинхронный код выглядит как синхронный.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
        var operationResult = Operation();
        txtResult.Text = operationResult;
}
```

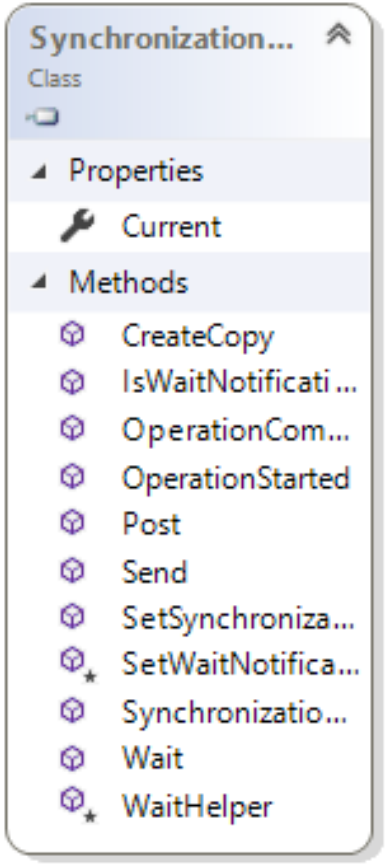
```
private async void Button_Click(object sender, RoutedEventArgs e)
{
        var operationResult = await OperationAsync();
        txtResult.Text = operationResult;
}
```

SynchronizationContext

SynchronizationContext – базовый класс для создания контекстов синхронизации.

Служит для обеспечения единого механизма распространения контекста синхронизации в различных моделях синхронизации.

SynchronizationContext позволяет расширять себя и предоставлять свои собственные реализации методов.



Основной функционал класса SynchronizationContext

Свойства:

- Current *(static)* – отдает контекст синхронизации прикрепленный к текущему потоку.

Методы:

- CreateCopy *(virtual)* – при переопределении в производном классе создает копию контекста синхронизации.
- Send *(virtual)* – при переопределении в производном классе отправляет синхронное сообщение в контекст синхронизации.
- Post *(virtual)* - при переопределении в производном классе отправляет асинхронное сообщение в контекст синхронизации.
- SetSynchronizationContext *(static)* – задает текущий контекст синхронизации.
- OperationStarted *(virtual)* – метод
- OperationCompleted *(virtual)* – метод

Использование контекста синхронизации

1) Создание экземпляра класса SynchronizationContext или производного от него класса.

2) Регистрация этого контекста синхронизации с помощью метода SetSynchronizationContext.

Когда вы захотите обратится к контексту синхронизации, вы должны запросить его с помощью статического свойства Current. С помощью метода Post или Send вы можете отправить сообщение в контекст синхронизации.

Post vs Send

В чем отличия этих двух методов?

Стандартная реализация метода Send:

```
public virtual void Send(SendOrPostCallback callback, object state)
{
        callback(state);
}
```

Стандартная реализация метода Post:

```
public virtual void Post(SendOrPostCallback callback, object state)
{
        ThreadPool.QueueUserWorkItem(new WaitCallback(callback.Invoke, state);
}
```

Метод Post производит асинхронный посыл сообщения. Он не ждет окончания работы делегата для собственного завершения, в отличии от метода Send.

При переопределении рекомендуется оставлять для метода Send синхронный посыл сообщения, для метода Post – асинхронный.

Продолжения оператора await

Оператор await имеет установленные правила для выполнения своих продолжений. Он выполняет продолжение в контексте одного из заранее определенных механизмов. **Всегда выбирается только один из них.**

Поиск механизма, который может выполнить продолжение оператора await идет в следующем порядке (по умолчанию):

- **Контекст синхронизации**. Оператор await в начале пытается захватить контекст синхронизации. Если он захватил его, то продолжение будет отправлено на выполнение в контекст синхронизации.
- **Планировщик задач**. Оператор await пытается получить текущего планировщика задач. Если он его получает, то продолжение будет отправлено на выполнение через планировщика задач.
- **Пул потоков**. Оператор await выполняет продолжение в контексте пула потоков, если предыдущие варианты не сработали или если было указано отказаться от окружения вызывающего потока.

Управление ожиданием

Есть открытый API для управления ожиданием. Для управления ожиданием используется метод **ConfigureAwait**(bool continueOnCapturedContext).

*Метод позволяет вам указать, нужно ли выполнятся продолжениям в захваченном контексте синхронизации класса **SynchronizationContext** или производных от него классов.*

- Нужно указать значение true для параметра **continueOnCapturedContext**, чтобы разрешить выполнение продолжения в захваченном контексте синхронизации.

- Нужно указать значение false для параметра **continueOnCapturedContext**, чтобы запретить выполнение продолжения в захваченном контексте синхронизации. Продолжение будет выполнено в пуле потоков.

Если вы не вызываете явно этот метод, то по умолчанию оператор await всегда будет пытаться захватить контекст синхронизации и выполнить продолжение в нем. Все равно, что вы вызвали метод ConfigureAwait и передали значение true.

Асинхронные лямбда выражения

Лямбда выражения могут быть асинхронными. На них накладываются все правила и ограничения асинхронных методов.

Создание асинхронных лямбда выражений:

НЕПРАВИЛЬНЫЙ ВАРИАНТ

```
private async Task MethodAsync()
{
        // ....
        Func<Task> func = () =>
        {
                await DoSomethingAsync();
        };
        await func.Invoke();
}
```

ПРАВИЛЬНЫЙ ВАРИАНТ

```
private async Task MethodAsync()
{
        // ....
        Func<Task> func = async () =>
        {
                await DoSomethingAsync();
        };
        await func.Invoke();
}
```

Чтобы сделать лямбда выражение асинхронным – добавьте модификатор async перед указанием формальных параметров лямбда выражения. Пример: async () => { ... }

Модификатор async для void

Для асинхронных методов с возвращаемым значением void существует потенциальное взаимодействие с контекстом синхронизации. Взаимодействие описано внутри строителя асинхронных методов AsyncVoidMethodBuilder.

Если контекст синхронизации будет захвачен произойдет следующее:

- При запуске конечного автомата через метод Start строителя AsyncVoidMethodBuilder будет вызван метод OperationStarted на захваченном контексте синхронизации.
- Если конечный автомат завершает работу с необработанным исключением (ловит его методом SetException), то исключение будет проброшено в захваченный контекст синхронизации.
- По завершению работы конечного автомата (успешном или провальном) будет вызван метод OperationCompleted на захваченном контексте синхронизации.

Если контекст синхронизации не будет захвачен, то вызовов методов OperationStarted и OperationCompleted не произойдет. А возникшее необработанное исключение будет выброшено через ThreadPool.

Делегаты с возвращаемыми значениями void

Будьте аккуратны при работе с асинхронными лямбда-выражениями. Вы можете не заметив использовать асинхронное лямбда-выражение с типом возвращаемого значения void.

Из-за этого, вы можете потерять преимущества использования задач (ожидание, результат, статус, отлов исключения) и получить непредсказуемый проброс исключения.

```
Action action = async () =>
{
        await Task.Run();
}

Func<Task> func = async () =>
{
        await Task.Run();
}
```

```
private async void Lambda1()
{
        await Task.Run();
}

private async Task Lambda2()
{
        await Task.Run();
}
```

Q&A

После урока обязательно

Повторите этот урок в видео формате на ITVDN.com

Проверьте как Вы усвоили данный материал на TestProvider.com

Информационный видеосервис для разработчиков программного обеспечения

ITVDN

IT VIDEO DEVELOPERS NETWORK

TestProvider PROMETRIC TEST CENTER CyberBionic systematics Microsoft Partner Silver Learning Windows Azure Cloud Partner EBA European Business Association