Постоянные подписчики

При разработке сервисов для небольшой программы вы, скорее всего, не думаете о многопоточности, о том, что какой-либо метод сервиса будут запускать сразу несколько экземпляров класса и много еще о чем обычно не думаете. Так же как и я. В небольших программах, которые, скорее всего, будут исполняться в один поток, не надо развитой системы событий. Можно просто передать делегат, который будет вызван после окончания работы сервиса.

Может быть обстоятельства, а может быть что-то уложилось в голове, но описанный выше подход больше не работает в программах, которые я пишу. Слишком много действий одновременно может делать пользователь и начинка программы, чтобы оставаться в рамках модели «один сервис – один потребитель». Однако с новым подходом меняется сложность и ответственность кода, и на этом пути поджидают многие сложности и опасности.

Новый подход, как вы наверно догадались, будет состоять в построении системы на событиях (event). При первом взгляде не видно никаких проблем при использовании событий, но это только на первый взгляд. Не уверен, что можно правильно проранжировать опасности, но, пожалуй, перечислю их в той последовательности, в которой с ними сталкиваешься.

Повторно подписались. Самая частая ошибка, которая вводит в ступор людей, это когда они обнаруживают:

- кратные прибавки в вычислениях
- двойные записи в базе данных
- прочие казусы вычислений и отображений данных.

Неявной проблемой, с которой сталкиваются не скоро при отсутствии проблем с вычислениями является скорость работы приложения. При двойной, тройной и так далее подписках программа начинает делать одни и те же действия без практической необходимости, что может серьезно снизить производительность.

Забыли отписаться. Бывает так, что повторной подписки не происходит, но и отписаться от события забыли. В этом случае могут возникнуть утечки памяти, так как сервис держит класс и не дает сборщику мусора пометить объект на удаление.

К слову, не всегда бывает удобно отписываться и хочется, чтобы подписка была «слабой», как WeakReference. Тогда, если на объект нет внешних ссылок, кроме слабых подписок, то сборщик мусора пометит его на удаление, подписка освободится, память освободится и будет небольшое программистское счастье. Собственно, об этом и пойдет речь, как реализовать и использовать в своем приложении слабую подписку на события. Попутно я планирую раскрыть еще несколько рабочих моментов при использовании событий и сервисов.

Эволюция применения

Прежде чем перейти к сути, я бы хотел немного остановиться на том, как может идти развитие проекта, какие проблемы могут вставать на пути. Начнем с самого простого. Допустим, есть какой-то сервис, который делает сложный расчет в течение нескольких секунд. Для того, чтобы не

блокировать выполнение других задач, например, обновление UI, будем запускать расчет в отдельном потоке, передавая параметры для расчета и метод для обратного вызова.

Самым простым примером может послужить сервис следующего вида:

```
public class SomeStraightCalcService {
    public void LongRunningMethodAsync(string val, Action callback) {
        ThreadPool.QueueUserWorkItem(LongRunningMethod, new {val, callback});
    }

    private void LongRunningMethod(dynamic obj) {
        Thread.Sleep(2000);
        obj.callback();
    }
}
```

Создаем новый поток с помощью пула потоков ThreadPool, вызвав метод QueueUserWorkItem. Это наверно самый простой способ для создания нового потока, при этом забота о создании, утилизации и работе потока перекладывается на систему. С помощью этого метода мы сводим вероятность натворить безобразий с потоками практически к нулю. Далее идет эмуляция долгих расчетов и вызов обратной функции, которая подскажет вызывающему классу, что работа готова.

Использование сервиса для демонстрационных целей сделаем таким:

```
internal class NonBlockingResource {
    public Stopwatch Watch1 = new Stopwatch();
    public Stopwatch Watch2 = new Stopwatch();

    public NonBlockingResource() {
        var service = new SomeStraightCalcService();
        Watch1.Start();
        service.LongRunningMethodAsync("", Ends1);

        Watch2.Start();
        service.LongRunningMethodAsync("", Ends2);
    }

    private void Ends1() {
        Watch1.Stop();
    }
    private void Ends2() {
        Watch2.Stop();
    }
}
```

Создаем таймеры, запускаем долгоиграющие методы, затем смотрим результат и сравниваем. Здесь вообще ничего сложного нет. Средой для проверки будут тестовые методы

```
[TestMethod]
public void SimpleLongRuning() {
    var someTestClass = new NonBlockingResource();

    Thread.Sleep(5000);
    var elapsedMs1 = someTestClass.Watch1.ElapsedMilliseconds;
    var elapsedMs2 = someTestClass.Watch2.ElapsedMilliseconds;
    Assert.Fail("1st call = {0}ms,  2nd call = {1}ms", elapsedMs1, elapsedMs2);
}
```

Для вывода информации запорем тест, и в сообщении выведем нужную нам информацию. У меня получилось примерно так: *1st call = 2005ms, 2nd call = 2004ms*

Далее обычно идет осознание того факта, что вычисления портятся либо данный подход натыкается на использование ресурса, к которому желательно обращаться монопольно.

Теперь вспоминаются начальные знания по многопоточности и на сцену выходит оператор **lock**.

Пусть монопольным ресурсом будет такой класс:

```
public class SingleAccess {
    private readonly object locker = new object();

    public void Enter() {
        lock (locker) {
            Thread.Sleep(2000);
        }
    }
}
```

Тогда сервис можно переписать таким образом:

```
public class SomeStraightCalcServiceWithBlockingResource {
    private readonly SingleAccess singleAccess = new SingleAccess();

    public void LongRunningMethodAsync(string val, Action callback) {
        ThreadPool.QueueUserWorkItem(LongRunningMethod, new {val, callback});
    }

    private void LongRunningMethod(dynamic obj) {
        singleAccess.Enter();
        obj.callback();
    }
}
```

По сути, несильно отличается от первоначального примера. Класс, использующий сервис, по своей структуре не поменяется вообще, только другой экземпляр сервиса будет создавать. Тестовый метод:

```
[TestMethod]
public void LongRunningWithBlockingResource() {
    var someTestClass = new WithBlockingResource();

    Thread.Sleep(5000);
    var elapsedMs1 = someTestClass.Watch1.ElapsedMilliseconds;
    var elapsedMs2 = someTestClass.Watch2.ElapsedMilliseconds;
    Assert.Fail("1st call = {0}ms,  2nd call = {1}ms", elapsedMs1, elapsedMs2);
}
```

И результаты вполне ожидаемые: *1st call = 4006ms, 2nd call = 2022ms*

Дальнейшее развитие ситуации может происходить таким образом, что запрос к блокирующему ресурсу можно выполнить один раз и оповестить об окончании вычислений всех интересующихся. Т.е. идет первый вызов к сервису, начинает выполняться долгоиграющий код. Затем, пока первый запрос все еще выполняется, какая-то часть программы тоже захочет получить эти же данные и делает второй запрос. В системе ,построенной на обратных вызовах придется в общем случае

делать еще один такой же долгий запрос, как только первый пройдет. Либо делать хитрое кеширование и еще что-нибудь.

Более логично будет сказать системе, что как только ты завершишь обрабатывать запрос, выдай результаты обоим потребителям, а не делай все то же самое еще раз. Т.е. подписаться на событие. В самом простом варианте это выглядит так (в рабочем коде не забываем про double check lock):

```
public class CalcServiceWithBlockingResource {
    private readonly SingleAccess singleAccess = new SingleAccess();
    private bool inAction;

    public event Action Ends;

    public void LongRunningMethodAsync(string val) {
        if(inAction) return;

        inAction = true;
        ThreadPool.QueueUserWorkItem(LongRunningMethod, val);
    }

    private void LongRunningMethod(object obj) {
        singleAccess.Enter();
        if(Ends != null)
            Ends();
        inAction = false;
    }
}
```

В качестве простой проверки используется булева переменная inAction для того, чтобы не запускать два раза расчет, если он уже идет.

Использование данного сервиса будет немного отличаться от предыдущих примеров, но не кардинальным образом, на мой взгляд.

```
internal class EventWithBlockingResource {
    public Stopwatch Watch1 = new Stopwatch();
    public Stopwatch Watch2 = new Stopwatch();
    private readonly CalcServiceWithBlockingResource service;

    public EventWithBlockingResource() {
        service = new CalcServiceWithBlockingResource();
        service.Ends += Ends1;
        service.Ends += Ends2;

        Watch1.Start();
        service.LongRunningMethodAsync("");

        Watch2.Start();
        service.LongRunningMethodAsync("");
    }

    private void Ends1() {
        Watch1.Stop();
        service.Ends -= Ends1;
    }

    private void Ends2() {
        Watch2.Stop();
        service.Ends -= Ends2;
    }
```

```
}
```

Вместо передачи метода для обратного вызова, подписываемся на событие об окончании работы метода.

```
[TestMethod]
public void EventBasedWithSameBlockingResource() {
    var someTestClass = new EventWithBlockingResource();

    Thread.Sleep(5000);
    var elapsedMs1 = someTestClass.Watch1.ElapsedMilliseconds;
    var elapsedMs2 = someTestClass.Watch2.ElapsedMilliseconds;
    Assert.Fail("1st call = {0}ms,  2nd call = {1}ms", elapsedMs1, elapsedMs2);
}
```

Результат: *1st call = 2000ms, 2nd call = 2000ms*

«Слабые» делегаты

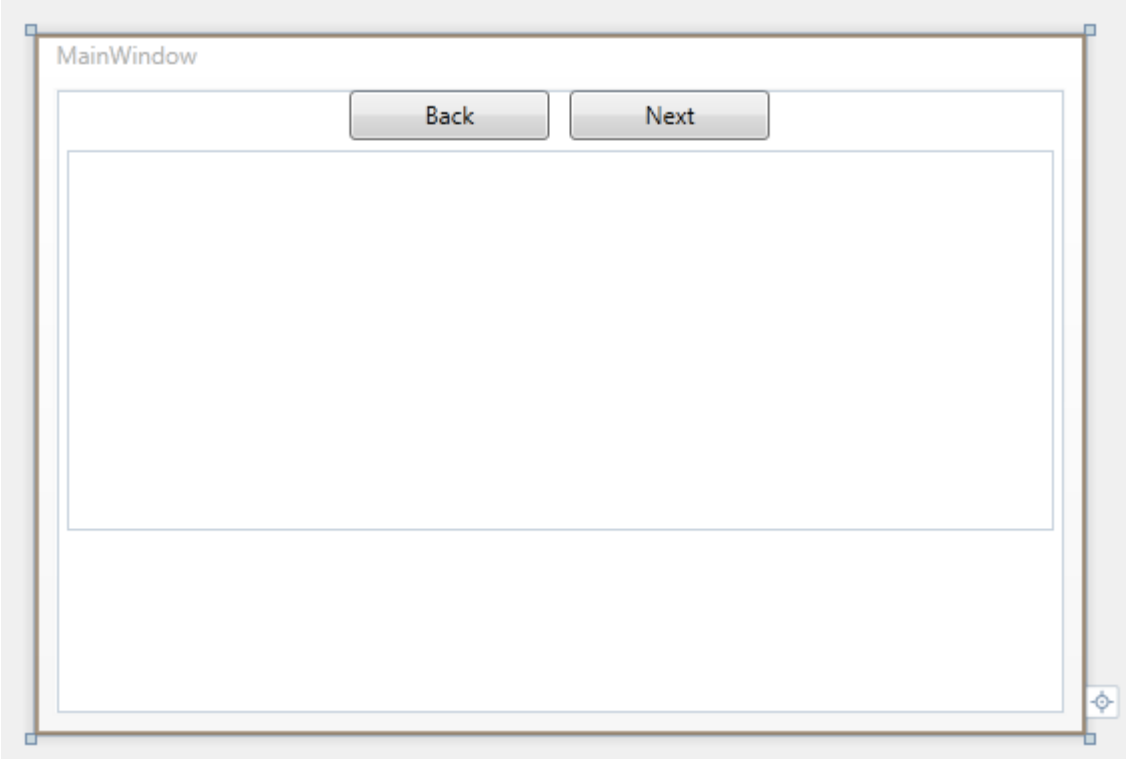
На этом вводную часть можно закончить и перейти к «слабым» делегатам, и показу их работы на примере приложения с несколькими экранами, которые участвуют в подписке на события. В данном примере будет наглядно видно как утекает память, если не отписываться от событий.

Приложение будет простым насколько это возможно для демонстрации утечек в реальной жизни. Допустим, у нас есть модели представлений, которые подписываются на некий постоянный сервис, в роли которого выступит основная форма окна.

При переходе между UserControls будем замерять занятую память, предварительно вызвав сборщик мусора для прохождения по всем поколениям объектов.

```
private void CheckMemory() {
    GC.Collect();
    Message.Content = GC.GetTotalMemory(true);
}
```

Основное окно будет состоять из кнопок для создания userControls, надписи для вывода объема занятой памяти и панели для показа компонентов.



Вот такое простое окно у нас есть. Начинка у него будет состоять из:

- События
- Вызова события
- Обработчика нажатия кнопки один
- Обработчика нажатия кнопки два

Все просто!

Следуя классике жанра, событие и его вызов будут объявлены так:

```
public event EventHandler LocalEvent;

protected virtual void OnClick(EventArgs e) {
    var event = LocalEvent;
    if (event!= null)
        event (this, e);
}
```

Обработчик кнопки будет ненамного сложнее:

```
private void PrevScreenClick(object sender, RoutedEventArgs e) {
    var controlOne = new ControlOne();
    var oneModel = ((ModelOne) controlOne.DataContext);
    LocalEvent += oneModel.LinkingMethod;
    SetContent(e, controlOne);
}
```

Создаем пользовательский компонент. Получаем его контекст, подписываемся на событие из вьюмодели ModelOne. В методе SetContent будет отображение компонента, вызов обработчика события и проверка занятой памяти:

```
private void SetContent(RoutedEventArgs e, UserControl control) {
    content.Children.Clear();
    content.Children.Add(control);

    OnClick(e);

    CheckMemory();
}
```

В свою очередь модель представлений так же очень просты и не содержат никакой логики, так как это не требуется для работы демонстрационного примера.

```
public class ModelOne {
    public void LinkingMethod(object sender, EventArgs eventArgs) {}
}

public class ModelTwo {
    public void LinkingMethod(object sender, EventArgs eventArgs) {}
}
```

Создание моделей и назначение контексту пользовательского компонента будет происходить при создании самого пользовательского компонента:

```
public partial class ControlOne : UserControl {
    public ControlOne() {
        InitializeComponent();
        DataContext = new ModelOne();
    }
}
```

Думаю, что дальнейших разъяснений не требуется и можно запускать приложение и смотреть, что же получается в результате работы.

При запуске показывается, что занято 634420 байт. При нажатии на любую из кнопок создается userControl и происходит подписка на событие, после чего размер занимаемой памяти выростает до 764684 байт. При повторных нажатиях начинается непрерывный рост:

- 768204
- 768256
- 768420
- 777648

Таким образом я дощелкал до того, что стало показывать 790234 байт. Можете быстро щелкать по кнопкам или ждать в надежде, что подписка отсохнет магическим образом или как-то еще освободиться память, но этого не будет. Память, в конечном счете, будет только потребляться.

Все потому, что мы забыли отписаться от событий, созданные компоненты привязаны к основной форме через события, и сборщик мусора думает, что они требуются программе, не помечая их на удаление.

От возникшей проблемы можно избавиться, если реализовать делегаты на «слабых» ссылках. WeakReference «невидимы» для сборщика мусора и позволяют ему помечать объекты на удаление, если есть только такие «слабые» ссылки на объект.

Дабы не томить, сразу приведу полный текст для «слабого» делегата.

```
internal class WeakDelegate<TDelegate> where TDelegate : class {
    private readonly List<WeakReference> targets = new List<WeakReference>();

    public WeakDelegate() {
        if (!typeof (TDelegate).IsSubclassOf(typeof (Delegate)))
            throw new InvalidOperationException
                ("TDelegate должен быть настоящим делегатом");
    }

    public void Combine(TDelegate target) {
        if (target == null) return;
        foreach (var d in (target as Delegate).GetInvocationList())
            targets.Add(new WeakReference(d));
    }

    public void Remove(TDelegate target) {
        if (target == null) return;
        foreach (var d in (target as Delegate).GetInvocationList()) {
            var weak = targets.Find(w => d.Equals(w.Target));
            if (weak != null) targets.Remove(weak);
        }
    }

    public TDelegate Target {
        get {
            var deadRefs = new List<WeakReference>();
            Delegate combinedTarget = null;
            foreach (var weak in targets) {
                var target = (Delegate) weak.Target;
                if (target != null)
                    combinedTarget = Delegate.Combine(combinedTarget, target);
                else
                    deadRefs.Add(weak);
            }
            foreach (var weak in deadRefs) // удаление мертвых ссылок
                targets.Remove(weak);
            return combinedTarget as TDelegate;
        }
        set {
            targets.Clear();
            Combine(value);
        }
    }
}
```

Теперь можно подробнее разобрать этот код.

В конструкторе проверяется, чтобы переданный параметр действительно был делегатом, так как в C# нет ограничителя на шаблонный параметр вида delegate. Нельзя написать

```
internal class WeakDelegate<TDelegate> where TDelegate : delegate { .. .. }
```

Переменной класса является список слабых ссылок, для того чтобы можно было осуществлять множественную подписку.

Методы Combine и Remove будут отвечать соответственно за добавление подписчиков и удаление их. Это потребуется при создании события в итоговом классе. В этих методах у каждого делегата берется список подписчиков, так как они могут быть так же с множественной подпиской.

При добавлении каждый полученный из списка подписок оборачиваем в «слабую» ссылку. При удалении предварительно проверяем, существует ли до сих пор подписчик.

При получении самого списка подписок, мы отфильтровываем мертвые подписки и возвращаем подписки только для существующих подписчиков.

В целом все не так сложно, если внимательно проговорить про себя код и может даже нарисовать примерный ход выполнения на бумаге. После этого можно использовать «слабые» события в нашем, кхе-кхе, приложении и посмотреть, как будет меняться динамика потребления памяти. Для этого нам потребуется кое-что заменить.

Удаляем старое событие и заменяем его на ручную реализацию:

```
private readonly WeakDelegate<EventHandler> click = new WeakDelegate<EventHandler>();

public event EventHandler LocalEvent {
    add { click.Combine(value); }
    remove { click.Remove(value); }
}
```

Метод OnClick также подвергнется небольшим изменениям:

```
protected virtual void OnClick(EventArgs e) {
    var target = click.Target;
    if (target != null) target(this, e);
}
```

Не могу назвать их кардинальными, логика осталась та же самая. Только поправка на структуру делегата.

Собственно всё, можно запускать приложение, нажимать на кнопки и смотреть, как меняется потребляемая память.

При запуске показывается, что занято 634416 байт. После первого нажатия 764808, потом

- 777560
- 777744
- И далее будет все крутиться около 778000 байт с небольшими отклонениями.

Постоянного роста памяти не будет происходить, чего мы и добивались. Надеюсь, данная техника позволит вам сохранить нервные клетки, память и голосовые связки. При этом все равно не стоит забывать отписываться от событий, даже если они слабо связаны.

Послесловие

В реальных проектах вместо ThreadPool лучше использовать Task, так как они дают больший контроль над процессом.

Системы на событиях гораздо сложнее, чем можно себе представить, руководствуясь только тем что было освещено в данной статье.

Исходный код можно взять с Assembla. Статья написана по мотивам книги J.Albahari “C# in the Nutshell 4.0”

Hard’n’heavy!

Violet Tape

Как начиналась статья в версии 0.1

Несмотря на то, что во всех книжках и руководствах пишут про то, что не забывайте отписываться от событий, тем не менее, это одна из самых распространенных ошибок. Данная ошибка ведет к утечкам памяти, так как не освобождаются нужные ресурсы, к множественным подпискам, что приводит к общему замедлению программы. Причем может произойти не только банальное замедление отрисовки интерфейса, если вы подписались на обновление компонентов, но и к неправильным расчетам данных, что уже более страшно.

Отчасти можно избавиться от проблемы, или не замечать ее, если использовать делегаты типа Action получив таким образом единственность подписки. Т.е. при следующей подписке на событие ресурса, потенциально освобождается предыдущий подписчик и становится доступным для сборщика мусора. Однако такой подход не будет работать для многопоточных систем или когда нужен отзывчивый интерфейс с богатой информационной моделью.

Рассмотрим самый типичный пример кривой реализации…