Иерархичные структуры данных и промежуточные расчеты

Традиционной, в плане изучения программирования в школах, университетах, курсах повышения квалификации и так далее – является архитектура, в которой мы явно управляем ходом выполнения программы. Это начинается с фактически процедурного турбо паскаля и идеологически продолжается в «рабочих» для бизнес-программирования языках (C# и Java). Каждое действие программы задается явно, т.е. если надо что-то пересчитать после добавления нового элемента в коллекцию, то так и пишут: сервис X пересчитай мне коллекцию с помощью метода Z. Это – **push** модель, мы толкаем/принуждаем систему сделать определенные действия.

Более интересной является **pull** модель, когда объекты сами говорят о том, что с ними следует сделать. При той же ситуации, что описана выше, надо только добавить объект в коллекцию, а дальше уже коллекция сама может сказать сервису X пересчитать себя с помощью метода Z. Система реагирует на события внутри себя.

Да, сначала все это может показаться несколько сумбурно и непонятно, однако на следующих примерах, я надеюсь, будут видны и плюсы pull модели и ее отличия от традиционного подхода.

Постоянные читатели блога уже знают, что в последнее время я проявляю повышенный интерес к событийной модели построения приложения. На мой взгляд, она более интересна и компактна в реализации, особенно если использовать Reactive Extensions (Rx).

Немного о Rx

Rx – это библиотека позволяющая создавать программы основанные на асинхронной модели или на событийной модели данных, используя observable (не могу никак подобрать определение русское) коллекции и операторы в стиле LINQ, на самом деле в формате fluent интерфейса.

Данная библиотека разработана в недрах MS Lab и активно ей продвигается как для десктоп платформ, так и для WP7. Лучше всего устанавливать библиотеки для работы с помощью менеджера компонентов NuGet.

Сам Rx Framework является набором методов расширений для базовых интерфейсов

- IObservable<T>
- IObserver<T>

Базовыми элементом Rx является класс Subject<T> реализующий оба интерфейса. Управление потоком событий происходит с помощью 3-х методов:

- OnNext – выдать новое значение в поток;
- OnComplete – указать на завершение потока;
- OnError – указать на ошибку при выполнении;

Все оказывается логично и пока что не очень сложно.

*Самое главное, что нужно для себя понять, так это что работа с Rx подразумевает работу именно с **потоком** данных/событий. Это не коллекция объектов, не какой-либо статичный набор, это именно поток данных, который идет на обработку с некоторой скоростью.*

Если хочется самостоятельно покопаться с Rx то хорошо начать с цикла статей у Ли Кэмпбела

Вводные данные

Работать будем со структурой, которую наверно каждый в своей жизни хоть раз да написал. Иерархия классов с указанием на родителя и коллекцию дочерних элементов того же типа. При этом всегда надо было осуществлять какие-либо суммарные подсчеты. У всех такое было)

В таком случае классическая структура класса не требует дополнительных пояснений.

```
public class CategoryClassic {
    private readonly List<CategoryClassic> items = new List<CategoryClassic>();

    public CategoryClassic Parent { get; private set; }
    public string Name { get; set; }

    public ReadOnlyCollection<CategoryClassic> Items {
        get { return items.AsReadOnly(); }
    }

    public int DebtCount { get; set; }
    public int CreditCount { get; set; }
    public decimal DebtSummary { get; set; }
    public decimal CreditSummary { get; set; }


    public void Add(CategoryClassic category) {
        if (items.Contains(category)) return;

        items.Add(category);
        category.Parent = this;
    }

    public void Remove(CategoryClassic category) {
        category.Parent = null;
        items.Remove(category);
    }

    public int CountAllDescendants() {
        var count = items.Count;

        items.ForEach(i => count = count + i.CountAllDescendants());

        return count;
    }
}
```

Все предельно просто. Коллекция элементов только для чтения, так как при добавлении и удалении необходимы дополнительные действия. 4 поля для подсчета сумм. Так же присутствует служебный метод для подсчета всех потомков у данного элемента.

От этого классического примера будем отталкиваться.

Задача

Надо посчитать суммы по DebtCount и другим полям для каждого уровня. Оценить затраты при единовременном составлении дерева, а так же при добавлении элементов в дерево.

Классика жанра

Классический подход при решении данной задачи состоит в написании метода или сервиса, который подсчитает рекурсивно нужные суммы для каждого уровня.

Реализация в моем случае в виде сервиса, так как именно это я вытащил из недр текущего проекта на работе.

```
public class CategoryClassicCalcService {
     public void Calculate(CategoryClassic category) {
         if(category.Items.Count == 0) return;

         category.Items.ToList().ForEach(Calculate);

         category.CreditCount = category.Items.Sum(i => i.CreditCount);
         category.CreditSummary = category.Items.Sum(i => i.CreditSummary);

         category.DebtCount = category.Items.Sum(i => i.DebtCount);
         category.DebtSummary = category.Items.Sum(i => i.DebtSummary);
     }
}
```

Простые тесты приводить не буду, рассмотрим те, где появляются упомянутые ранее NBuilder и FluentAssertions.

Лирическое отступление о NBuilder

Когда надо создать иерархию объектов, то еще можно ее создать из четырех, пяти, максимум 10 элементов. Но узнав о **NBuilder**, мой максимум снизился до двух элементов. Сейчас я расскажу почему, и как повторить это в домашних условиях.

Кроме того, что NBuilder умеет хорошо строить коллекции объектов, он умеет строить иерархии, и это действительно просто. Нам понадобится использовать шаблонный класс **HierarchySpec**, в котором мы укажем следующие вещи:

- Метод для добавления нового элемента в иерархию
- Глубину иерархии
- Минимальное количество потомков на узел
- Максимальное количество потомков на узел
- Метод именования, если надо
- Количество корневых узлов

Данный класс описывает правила, по которым надо будет строить иерархию. Для построения самой иерархии воспользуемся методом **BuildHierarchy**.

```
[Test]
public void ShouldSummarizeByHierarchy3Level() {
    // Arrange
    var hierarchySpec = new HierarchySpec<CategoryClassic> {
           AddMethod = (parent, child) => parent.Add(child),
           Depth = 3,
           MinimumChildren = 1,
```

```
            MaximumChildren = 1,
            NamingMethod = (item, index) => item.Name = "Category " + index,
            NumberOfRoots = 1
                                                        };
    var root = Builder<CategoryClassic>.CreateListOfSize(5)
                        .BuildHierarchy(hierarchySpec).First();
    var service = new CategoryClassicCalcService();

    root.CreditCount.Should().Be(1);

    // Act
    service.Calculate(root);

    // Assert
    root.CreditCount.Should().Be(0 + 0 + 0 + 4);
}
```

Все же есть некоторые нюансы в построении иерархии. Количество созданных элементов в методе **CreateListOfSize** должно быть не меньше, чем потребуется для построения иерархии.

По самому тесту пояснений особых думаю не надо делать. Создаем определение на 3х уровневую иерархию по одному элементу на уровень. Затем строим иерархию, считаем и проверяем результат.

Результаты нагрузочного тестирования

Увеличив глубину иерархии до 6, а так же зафиксировав количество дочерних элементов для узла в размере 6 элементов, можно провести замеры по скорости создания такого массива данных и расчета сумм. Для большей глубины и количества элементов возникает StackOverflow.

Замер начинается до объявления спецификации на построение иерархии и останавливается сразу после пересчета полей.

Элементов: 55 986
Время построения (ms): 8 552
Время добавления и пересчета (ticks): 128 909

Хорошо, просто запомним эти цифры, пока что это ни о чем не говорит. Хотя конечно пересчет длителен, но другого не ожидалось.

Построение на событиях

Подумав о том, что пересчет всей структуры не самое хорошее мероприятие при добавлении нового листового элемента, приходим к решению, что при добавлении элемента будем говорить пересчитать себя и сообщить дальше по цепочке. Таким образом будут затронуты только связанные родительские узлы.

Реализация значительно разрастется, что и говорить. Начнем с того, что внутренней коллекцией будет ObservableCollection и подпишемся на изменение этой коллекции в конструкторе. Когда это событие возникнет, надо пересчитать суммы и сказать родителю, если он есть. Но лучше сделать заглушку для делегата, чтобы не вставлять лишнюю проверку.

```
public event Action SummaryChanged;

public CategoryNotify() {
    SummaryChanged = () => { };
```

```
    items = new ObservableCollection<CategoryNotify>();
    items.CollectionChanged += UpdateSummary;
}
```

При изменении коллекции пересчитываем суммы и рапортуем выше с помощью метода SummaryChanged :

```
private void UpdateSummary(object sender, NotifyCollectionChangedEventArgs e) {
     UpdateSummary();
}

private void UpdateSummary() {
    CreditCount = items.Sum(i => i.CreditCount);
    DebtCount = items.Sum(i => i.DebtCount);

    CreditSummary = items.Sum(i => i.CreditSummary);
    DebtSummary = items.Sum(i => i.DebtSummary);
    Calls++;
    SummaryChanged();
}
```

Так хитро сделано для удобства подписки родителей.

При добавлении нового элемента, подписываемся на его сообщения об изменении его коллекции дочерних элементов.

```
public void Add(CategoryNotify category) {
    if(items.Contains(category)) return;

    items.Add(category);
    category.Parent = this;
    category.SummaryChanged += UpdateSummary;
}
```

Ключевые моменты указаны, полный код класса можно будет посмотреть в исходниках.

Честно сказать в событиях можно запутаться на раз, и я не с первого раза написал как надо события, да и потом перечитывать не просто. Зато быстро работает при обновлении! Кстати о тестах:

Элементов: 55 986
Время построения (ms): 10 007
Время добавления и пересчета (ticks): 72

Вау! Вот это скорость пересчета, я знал, что должно быть быстрее, но чтобы настолько! А упавшая скорость построения объясняется тем, что происходит избыточный пересчет во время крупного построения иерархии.

Rx реализация

Теперь посмотрим насколько быстро будет работать иерархия построенная на тех же принципах, что в предыдущем примере, но с другими объектами оповещения. На мой взгляд получилось проще в записи.

Итак, нужен базовый объект из Rx, который будет реализовывать необходимые нам интерфейсы. Я выбрал **ReplayObject**, так как он помнит все пришедшие значения.

```
private readonly ReplaySubject<CategoryRx> addedSubject
                                                = new ReplaySubject<CategoryRx>();
```

В этом варианте обойдемся без конструктора, и без ObservableCollection.

```
public void Add(CategoryRx category) {
    if (items.Contains(category)) return;

    items.Add(category);
    category.Parent = this;
    category.addedSubject
        .Subscribe(Update);

    Update(category);
}

private void Update(CategoryRx category) {
    CreditCount = items.Sum(i => i.CreditCount);
    DebtCount = items.Sum(i => i.DebtCount);

    CreditSummary = items.Sum(i => i.CreditSummary);
    DebtSummary = items.Sum(i => i.DebtSummary);
    Calls++;
    addedSubject.OnNext(this);
}
```

Меньше кода и компактнее. Смысл тот же. При добавлении подписываемся на событие добавления нового элемента (строка 6-7), пересчитываем собственные показатели (метод Update) и говорим родителю об обновлении (последняя строка).

Метод OnNext, возбуждает событие с указанным параметром. Делегат, указанный в методе Subscribe, принимает параметр из события, которое было возбуждено методом OnNext.

Проще ведь, а?

Однако посмотрим что там со временем выполнения.

Элементов: 55 986
Время построения (ms): 14 125
Время добавления и пересчета (ticks): 182

Думал, что будет быстрее, однако, тоже неплохой в целом результат. Большие задержки видимо объясняются внутренним устройством ReplaySubject и тем, как происходит возбуждение события и его обработка. Это ведь дополнительный уровень абстракции.

Итоги

Итоги для работы с 55 986 элементами представлены в таблице ниже

Способ	Время построения (ms)	Время добавления и пересчета (ticks)
Classic	8552	128909
Event	10007	72
Rx	14125	182

Из всего этого можно сделать следующие выводы:

- Если добавления элементов не будет происходить во время работы, то лучше всего использовать рекурсивный алгоритм подсчета промежуточных результатов на каждом уровне.
- Если добавления ожидаются и немало, то следуетдля себя решить, что важнее: скорость или читабельность.

Развитие

Иерархию, которая строит модель обновления на событиях возможно получится ускорить на этапе построения, если сделать статическое свойство игнорирования пересчетов в процессе построе ния основной массы дерева. После чего рекурсивно посчитать то, что требуется и включить событийное обновление обратно. Должен получиться синтез из классического подхода и событийного. Берем плюсы каждого, но код и использование усложняется. Надо быть более внимательным.

Касательно Rx, я думаю, что можно и тут добиться ускорения построения общей структуры, если использовать правильно метод Throttle. Однако у меня не получилось, чего-то не учел. Но я еще учусь и пробую, так что как найду способ, обязательно расскажу.

Про Rx пока что рассказано совсем чуть-чуть, но большинство методов расширения еще рассмотрим позже, плюс они оставляют впечатление некоторой синтетичности.

Source Code.

Hard'n'heavy!

Violet Tape