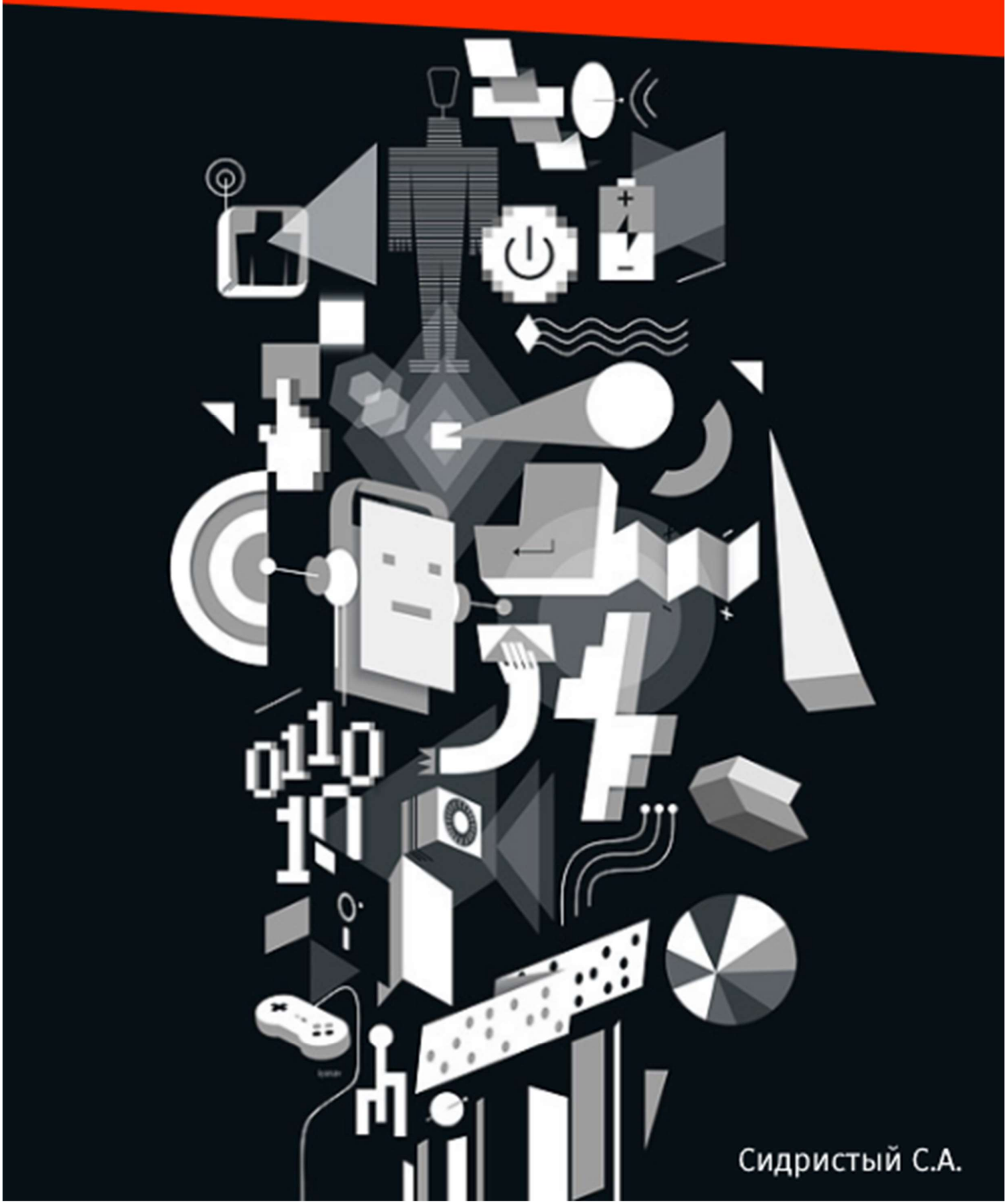
# CLR Book: под капотом .NET Framework



Сидристый С.А.

# CLR book:
# под капотом .NET Framework

*Сидристый Станислав*

# Оглавление

# Исключения

В нашем разговоре о потоке исполнения команд различными подсистемами пришло время поговорить про исключения или, скорее, исключительные ситуации. И прежде чем продолжить стоит совсем немного остановиться именно на самом определении. Что такое исключительная ситуация?

Исключительной называют такую ситуацию, которая делает исполнение дальнейшего или текущего кода абсолютно не корректным. Не таким как задумывалось, проектировалось. Переводит состояние приложения в целом или же его отдельной части (например, объекта) в состояние нарушенной целостности. Т.е. что-то экстраординарное, исключительное.

Почему же это так важно - определить терминологию? Работа с терминологией очень важна, т.к. она держит нас в рамках. Если не следовать терминологии можно уйти далеко от созданного проектировщиками концепта и получить множество неоднозначных ситуаций. А чтобы закрепить понимание вопроса давайте обратимся к примерам:

```
struct Number
{
    public static Number Parse(string source)
    {
        // ...
        if(!parsed)
        {
            throw new ParsingException();
        }
        // ...
    }

    public static bool TryParse(string source, out Number result)
    {
       // ..
       return parsed;
    }
}
```

Этот пример кажется немного странным: и это не просто так. Для того чтобы показать исключительность проблем, возникающих в данном коде я сделал его несколько утрированным. Для начала посмотрим на метод `Parse`. Почему он должен выбрасывать исключение?

Он принимает в качестве параметра строку, а в качестве результата - некоторое число, которое является значимым типом. По этому числу мы никак не можем определить, является ли оно результатом корректных вычислений или же нет: оно просто есть. Другими словами, в интерфейсе метода отсутствует возможность сообщить о проблеме;

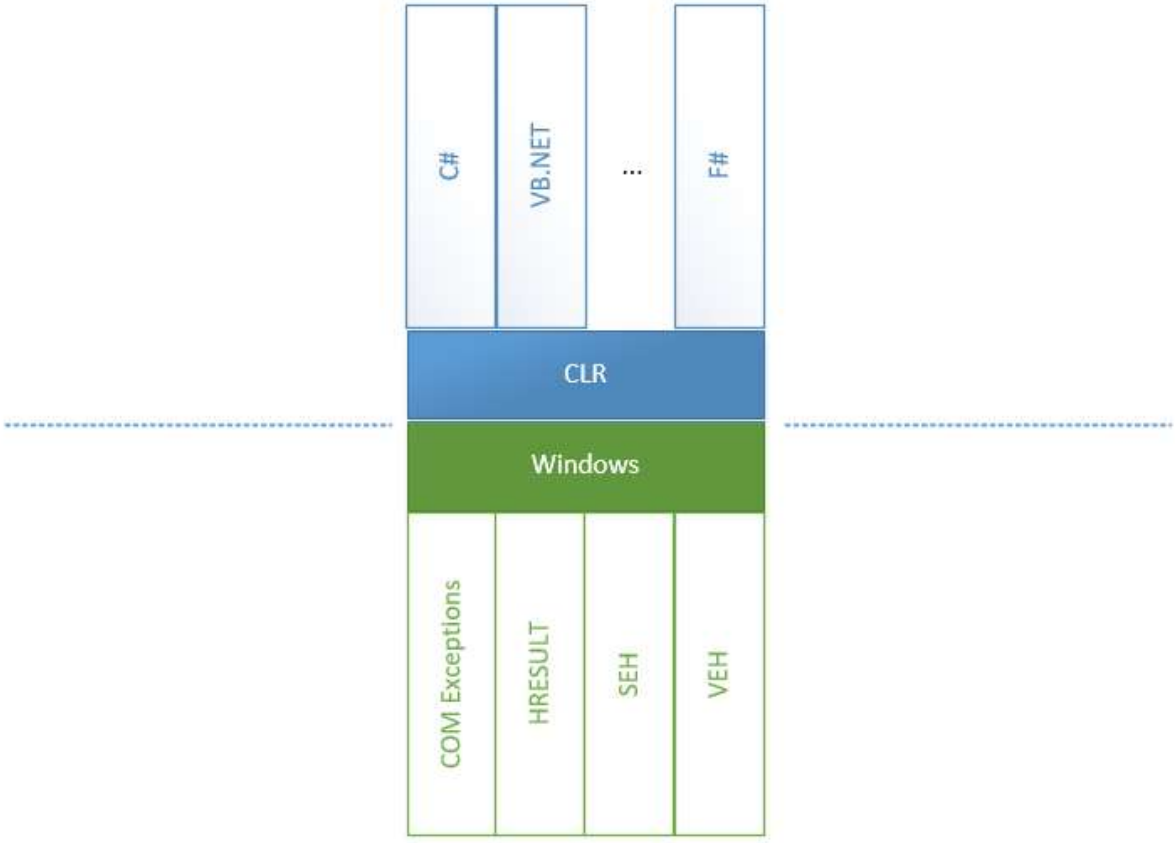
- С другой стороны метод, принимая строку подразумевает что ее для него корректно подготовили: там нет лишних символов и строка содержит некоторое число. Если это не так, то возникает проблема предусловий к методу: тот код, который вызвал наш метод отдал не корректные данные.

Получается, что для данного метода ситуация получения строки с не корректными данными является исключительной: метод не может вернуть корректного значения, но и вернуть абы что он не может. А потому единственный выход - бросить исключение.

Второй вариант метода обладает каналом сигнализации о наличии проблем с входными данными: возвращаемое значение тут `boolean` и является признаком успешности выполнения метода. Сигнализировать о каких-либо проблемах при помощи механизма исключений данный метод не имеет ни малейшего повода: все виды проблем легко уместятся в возвращаемое значение `false`.

## Общая картина

Обработка исключительных ситуаций может показаться вопросом достаточно элементарным: ведь все что нам необходимо сделать - это установить `try-catch` блоки и ждать соответствующего события. Однако вопрос кажется элементарным только благодаря огромной работе, проделанной командами CLR и CoreCLR чтобы унифицировать все ошибки, которые лезут в CLR со всех щелей - из самых разных источников. Чтобы иметь представление, о чем мы будем далее вести беседу, давайте взглянем на диаграмму:




На этой схеме мы видим, что в большом .NET Framework существует по сути два мира: все что относится к CLR и все, что находится за ней: все возможные ошибки, возникающие в Windows и прочем unsafe мире:

- Structured Exception Handling (SEH) - структурированная обработка исключений - стандарт платформы Windows для обработки исключений. Во время вызовов unsafe методов и последующем выбросе исключений происходит конвертация исключений unsafe <-> CLR в обе стороны: из unsafe в CLR и обратно, т.к. CLR может вызвать unsafe метод, а тот в свою очередь - CLR метод.
- Vectored Exception Handling (VEH) - по своей сути является корнем SEH, позволяя вставлять свои обработчики в точку выброса исключения. Используется в частности для установки `FirstChanceException`.
- COM+ исключения - когда источником проблемы является некоторый COM компонент, то прослойка между COM и .NET методом должна сконвертировать COM ошибку в исключение .NET

- И, наконец, обёртки для HRESULT. Введены для конвертации модели WinAPI (код ошибки - в возвращаемом значении, а возвращаемые значения - через параметры метода) в модель исключений: для .NET стардартом является именно исключительная ситуация

С другой стороны, поверх CLI располагаются языки программирования, каждый из которых частично или же полностью - предлагает функционал по обработке исключений конечному пользователю языка. Так, например, языки VB.NET и F# до недавнего времени обладали более богатым функционалом по части обработки исключительных ситуаций, предлагая функционал фильтров, которых не существовало в языке C#.

## Коды возврата vs. исключение

Стоит отдельно отметить модель работы с ошибками в приложении через коды возврата. Эта идея - просто вернуть ошибку - очень проста и понятна. Мало того, если отталкиваться от отношения к исключениям как к оператору `goto`, то коды возврата становятся намного более корректной реализацией: ведь в этом случае пользователь метода видит и возможность самой ошибки, а также может сразу понять, какие ошибки возможны. Но давайте не будет гадать на кофейной гуще, что лучше и для чего, а лучше обсудим проблематику выбора академически.

Представим, что все методы обладают интерфейсом для получения ошибки. Тогда все наши методы выглядели бы как-то так:

```
public bool TryParseInteger(string source, out int result);

public DialogBoxResult OpenDialogBox(...);
public WebServiceResult IWebService.GetClientsList(...);

public class DialogBoxResult : ResultBase { ... }
public class WebServiceResult : ResultBase { ... }
```

А их использование выглядело бы как-то так:

```
public ShowClientsResult ShowClients(string group)
{
    if(!TryParseInteger(group, out var clientsGroupId))
        return new ShowClientsResult
           { Reason = ShowClientsResult.Reason.ParsingFailed };
```

```
    var webResult = _service.GetClientsList(clientsGroupId);
    if(!webResult.Successful)
    {
        return new ShowClientsResult {
                Reason = ShowClientsResult.Reason.ServiceFailed,
                WebServiceResult = webResult
        };
    }

    var dialogResult = _dialogsService.OpenDialogBox(webResult.Result);
    if(!dialogResult.Successful)
    {
        return new ShowClientsResult {
            Reason = ShowClientsResult.Reason.DialogOpeningFailed,
            DialogServiceResult = dialogResult
        };
    }

    return ShowClientsResult.Success();
}
```

Возможно, вам покажется что этот код перегружен обработкой ошибок. Однако я попрошу вас не согласиться с такой позицией: все что здесь происходит - это эмуляция работы механизма выброса и обработки исключений.

Как любой метод может сообщить о возникшей проблеме? Посредством некоторого интерфейса сообщения об ошибки. Например, в методе `TryParseInteger` интерфейсом является возвращаемое значение: если все хорошо, метод вернет `true`. Если все плохо, вернет `false`. Однако данный способ обладает и минусом: реальное значение возвращается через `out int result` параметр. Минус подхода с одной стороны состоит в том что возвращаемое значение чисто логически и по восприятию является более "возвращаемым значением" чем `out` параметр. А с другой - сам факт ошибки нам не всегда интересен. Ведь если строка для парсинга пришла из сервиса, который сгенерировал эту строку, значит проверять на ошибки парсинг нет необходимости: там всегда будет лежать правильная, пригодная для разбора строка. С другой стороны, если взять другую реализацию метода:

```
public int ParseInt(string source);
```

То возникает резонный вопрос: если парсинг все-таки будет содержать ошибку по неизвестной нам причине, что делать методу тогда? Вернуть ноль? Будет не корректно: нуля в строке не было. Тогда становится ясно что мы имеем конфликт

интересов: первый вариант многословен, а второй - не содержит канала сигнализации об ошибках. Однако, можно легко прийти к выводу, когда надо работать с кодами возвратов, а когда - с исключениями:

Код возврата необходимо внедрять тогда, кода факт ошибки является нормой поведения. Например, в алгоритме парсинга текста ошибки в тексте являются нормой поведения тогда как в алгоритме работы с разобранной строкой получение от парсера ошибки может являться критичным или, другими словами, чем-то исключительным.

## Блоки Try-Catch-Finally коротко

Блок `try` создает секцию, от которой программист ожидает возникновения критических ситуаций, которые с точки зрения внешнего кода являются нормой поведения. Т.е. другими словами если мы работаем с некоторым кодом, который в рамках своих правил считает внутреннее состояние более не консистентным и в связи с этим выбрасывает исключение, то внешняя система, которая обладает более широким видением той же ситуации возникшее исключение может перехватить блоком `catch` тем самым нормализовав исполнение кода приложения. А потому, *перехватом исключений вы легализуете их наличие на данном участке кода*. Это, на мой взгляд, очень важная мысль, которая обосновывает запрет на перехват всех типов исключений `try-catch(Exception ex){ ... }` *на всякий случай*.

Это вовсе не означает что перехватывать исключения идеологически плохо: я всего лишь хочу сказать о необходимости перехватывать то и только то, что вы ожидаете от конкретного участка кода и ничего больше. Например, вы не можете ожидать все типы исключений, которые наследуется от `ArgumentException` или же получение `NullReferenceException` поскольку это означает что проблема чаще всего не в вызываемом коде, а *в вашем*. Зато вполне корректно ожидать что желаемый файл открыть вы не сможете. Даже если на 200% уверены, что сможете, не забудьте сделать проверку.

Третий блок - `finally` - также не должен нуждаться в представлении. Этот блок срабатывает для всех случаев работы блоков `try-catch`. Кроме некоторых достаточно редких *особых* ситуаций, этот блок отрабатывает *всегда*. Для чего введена такая гарантия исполнения? Для зачистки тех ресурсов и тех групп объектов, которые были выделены или же захвачены в блоке `try` и при этом являются зоной его ответственности.

Этот блок очень часто используется без блока `catch`, когда нам не важно, какая ошибка уронила алгоритм, но важно очистить все выделенные для этого конкретно алгоритма ресурсы. Простой пример: для алгоритма копирования файла необходимо: два открытых файла и участок памяти под кэш-буфер копирования. Память мы выделить смогли, один файл открыть смогли, а вот со вторым возникли какие-то проблемы. Чтобы запечатать все в одну "транзакцию", мы помещаем все три операции в единый `try` блок (как вариант реализации), с очисткой ресурсов - в `finally`. Пример может показаться упрощенным, но тут главное - показать суть.

Чего не хватает в языке программирования C#, так это блока `fault`, суть которого - срабатывать всегда, когда произошла любая ошибка. Т.е. тот же `finally`, только на стероидах. Если бы такое было, мы бы смогли как классический пример делать единую точку входа в логгирование исключительных ситуаций:

```
try {
    //...
} fault exception
{
    _logger.Warn(exception);
}
```

Также, о чем хотелось бы упомянуть во вводной части - это фильтры исключительных ситуаций. Для платформы .NET это новшеством не является, однако является таковым для разработчиков на языке программирования C#: фильтрация исключительных ситуаций появилась у нас только в шестой версии языка. Фильтры призваны нормализовать ситуацию, когда есть единый тип исключения, который объединяет в себе несколько видов ошибок. И в то время как мы хотим отрабатывать на конкретный сценарий, вынуждены перехватывать всю группу и

фильтровать её - уже после перехвата. Я, конечно же, имею ввиду код следующего вида:

```
try {
    //...
}
catch (ParserException exception)
{
    switch(exception.ErrorCode)
    {
        case ErrorCode.MissingModifier:
            // ...
            break;
        case ErrorCode.MissingBracket:
            // ...
            break;
        default:
            throw;
    }
}
```

Так вот теперь мы можем переписать этот код нормально:

```
try {
    //...
}
catch (ParserException exception) when (exception.ErrorCode ==
ErrorCode.MissingModifier)
{
    // ...
}
catch (ParserException exception) when (exception.ErrorCode ==
ErrorCode.MissingBracket)
{
    // ...
}
```

И вопрос улучшения тут вовсе не в отсутствии конструкции `switch`. Новая конструкция как по мне лучше по нескольким пунктам:

- фильтруя по when мы перехватываем ровно то что хотим поймать и не более того. Это правильно идеологически;
- в новом виде код стал более читаем. Просматривая взглядом, мозг более легко находит определения ошибок, т.к. изначально он их ищет не в `switch-case`, а в `catch`;
- и менее явное, но также очень важное: предварительное сравнение идет ДО входа в catch блок. А это значит, что работа такой конструкции для случая промаха мимо всех условий будет идти намного быстрее чем `switch` с перевыбросом исключения.

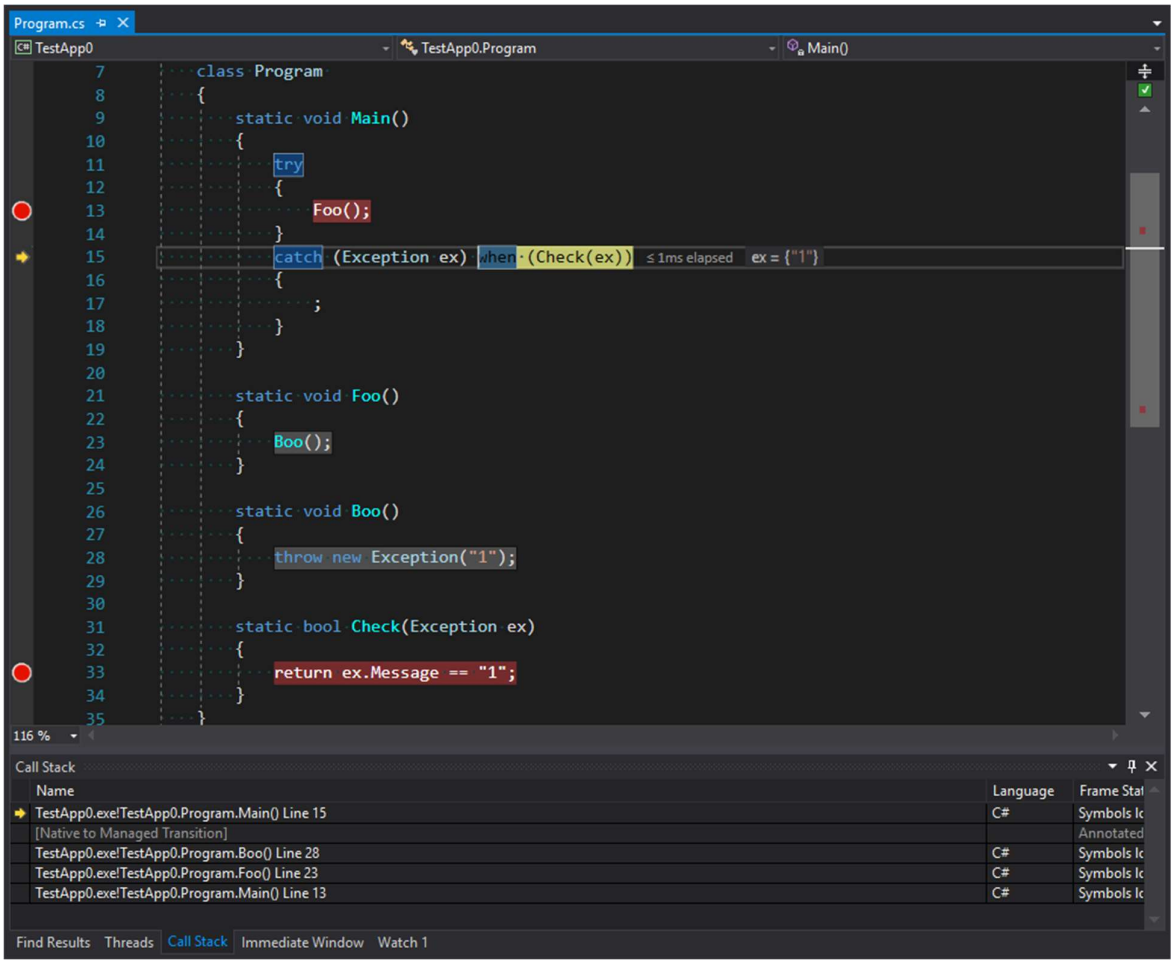
Особенностью исполнения кода по уверениям многих источников является то, что код фильтрации происходит *до* того как произойдет развертка стека. Это можно наблюдать в ситуациях, когда между местом выброса исключения и местом проверки на фильтрацию нет никаких других вызовов кроме обычных:

```
static void Main()
{
    try
    {
        Foo();
    }
    catch (Exception ex) when (Check(ex))
    {
        ;
    }
}

static void Foo()
{
    Boo();
}

static void Boo()
{
    throw new Exception("1");
}

static bool Check(Exception ex)
{
    return ex.Message == "1";
}
```




Как видно на изображении трассировка стека содержит не только первый вызов `Main` как место отлова исключительной ситуации, но и весь стек до точки выброса исключения плюс повторный вход в `Main` через некоторый неуправляемый код. Можно предположить, что этот код и есть код выброса исключений, который просто находится в стадии фильтрации и выбора конечного обработчика. Однако стоит отметить что *не все вызовы позволяют работать без раскрутки стека*. На мой скромный взгляд, внешняя унифицированность платформы порождает излишнее к ней доверие. Например, вызов методов между доменами с точки зрения кода выглядит абсолютно прозрачно. Тем не менее работа вызовов методов происходит совсем по другим законам. О них мы и поговорим в следующей части.

## Сериализация

Давайте начнем несколько издалека и посмотрим на результаты работы следующего кода (я добавил проброс вызова через границу между доменами приложения):

```
    class Program
    {
        static void Main()
```

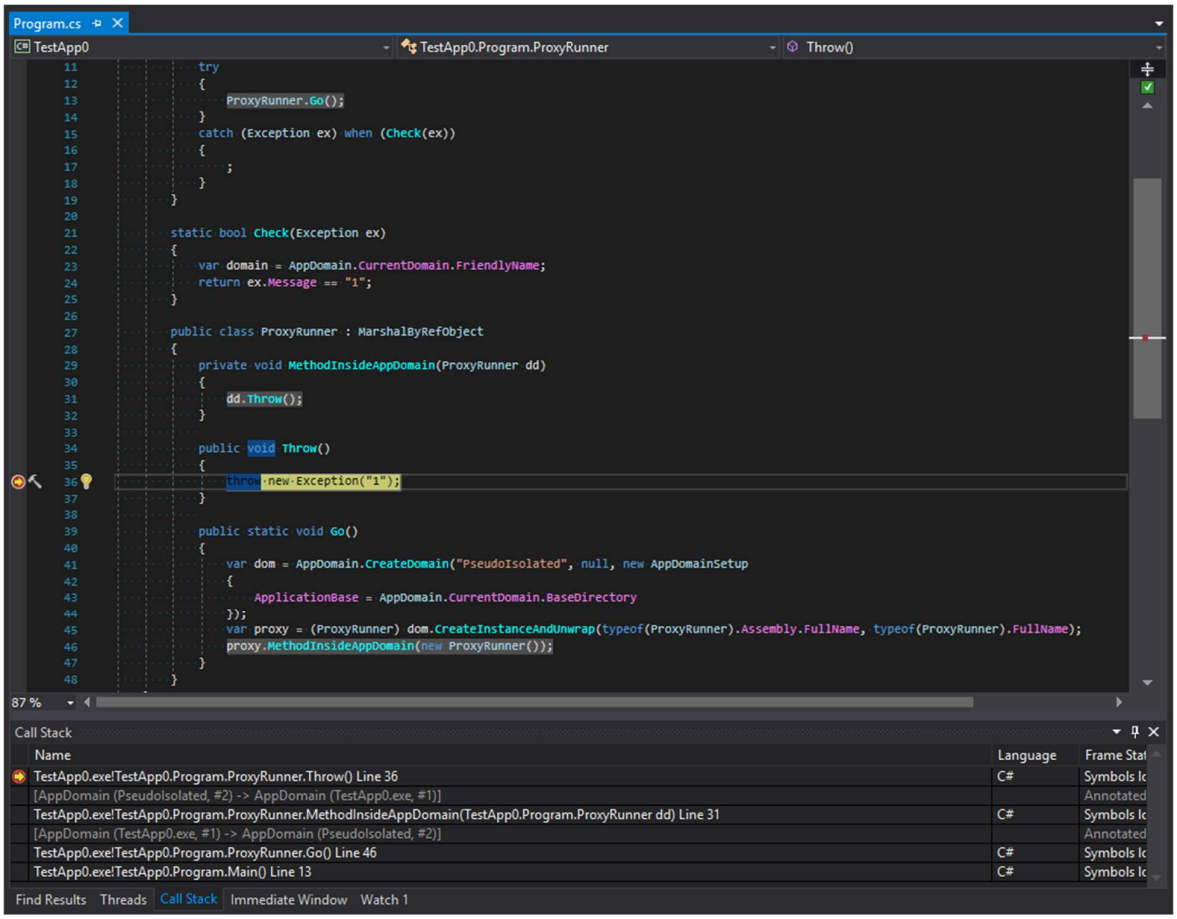
```
        {
            try
            {
                ProxyRunner.Go();
            }
            catch (Exception ex) when (Check(ex))
            {
                ;
            }
        }

        static bool Check(Exception ex)
        {
            var domain = AppDomain.CurrentDomain.FriendlyName; // -> TestApp.exe
            return ex.Message == "1";
        }

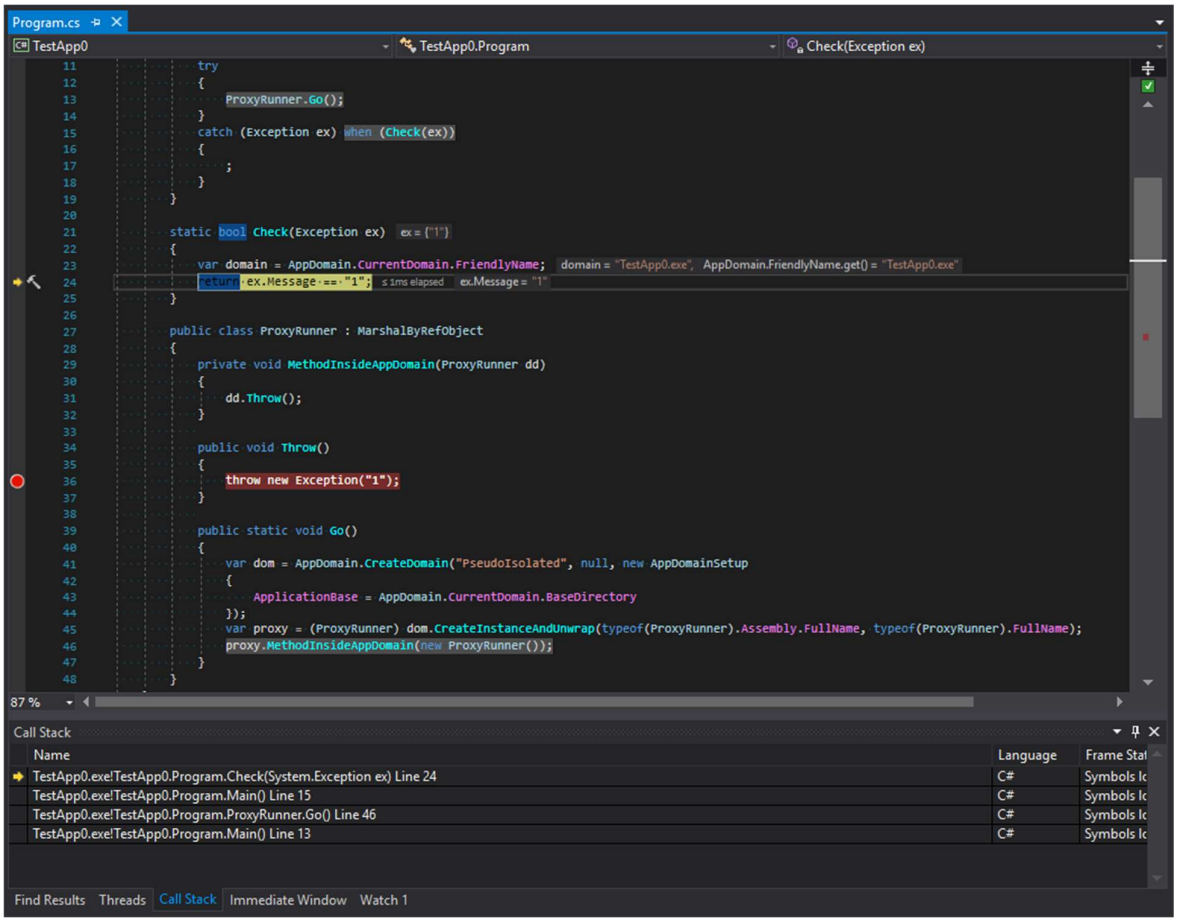
        public class ProxyRunner : MarshalByRefObject
        {
            private void MethodInsideAppDomain()
            {
                throw new Exception("1");
            }

            public static void Go()
            {
                var   dom   =   AppDomain.CreateDomain("PseudoIsolated",   null,   new
AppDomainSetup
                {
                    ApplicationBase = AppDomain.CurrentDomain.BaseDirectory
                });
                var               proxy               =               (ProxyRunner)
dom.CreateInstanceAndUnwrap(typeof(ProxyRunner).Assembly.FullName,
typeof(ProxyRunner).FullName);
                proxy.MethodInsideAppDomain();
            }
        }
    }
```

Если обратить внимание на размотку стека, то станет ясно что в данном случае она происходит еще до того, как мы попадаем в фильтр. Взглянем на скриншоты. Первый взят до того, как генерируется исключение:




А второй - после:




Изучим трассировку вызовов до и после попадания в фильтр исключений. Что же здесь происходит? Здесь мы видим, что разработчики платформы сделали некоторую с первого взгляда защиту дочернего домена. Трассировка обрезана по крайний метод в цепочке вызовов, после которого идет переход в другой домен. Но на самом деле, как по мне так это выглядит несколько странно. Чтобы понять, почему так происходит, вспомним основное правило для типов, организующих взаимодействие между доменами. Эти типы должны наследовать `MarshalByRefObject` плюс - быть сериализуемыми. Однако, как бы ни был строг C#, типы исключений могут быть какими угодно. А что это значит? Это значит, что могут быть ситуации, когда исключительная ситуация внутри дочернего домена может привести к ее перехвату в родительском домене. И если у объекта данных исключительной ситуации есть какие-либо опасные методы с точки зрения безопасности, они могут быть вызваны в родительском домене. Чтобы такого избежать, исключение сериализуется, проходит через границу доменов

приложений и возникает вновь - с новым стеком. Давайте проверим эту стройную теорию:

```
[StructLayout(LayoutKind.Explicit)]
class Cast
{
    [FieldOffset(0)]
    public Exception Exception;

    [FieldOffset(0)]
    public object obj;
}

static void Main()
{
    try
    {
        ProxyRunner.Go();
        Console.ReadKey();
    }
    catch (RuntimeWrappedException ex) when (ex.WrappedException is Program)
    {
        ;
    }
}

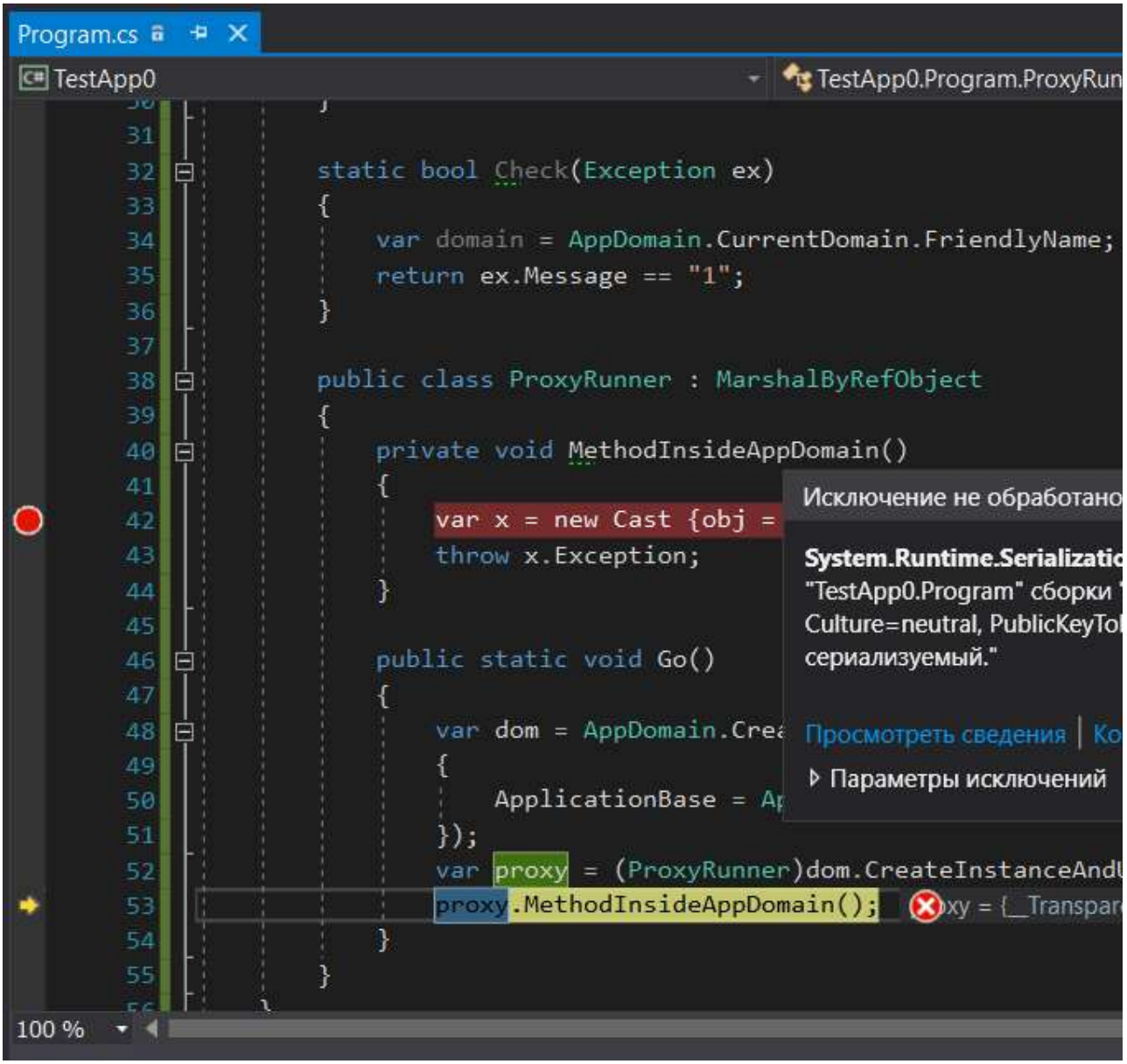
static bool Check(Exception ex)
{
    var domain = AppDomain.CurrentDomain.FriendlyName; // -> TestApp.exe
    return ex.Message == "1";
}

public class ProxyRunner : MarshalByRefObject
{
    private void MethodInsideAppDomain()
    {
        var x = new Cast {obj = new Program()};
        throw x.Exception;
    }

    public static void Go()
    {
        var dom = AppDomain.CreateDomain("PseudoIsolated", null, new AppDomainSetup
        {
            ApplicationBase = AppDomain.CurrentDomain.BaseDirectory
        });
        var                                  proxy                                    =
(ProxyRunner)dom.CreateInstanceAndUnwrap(typeof(ProxyRunner).Assembly.FullName,
typeof(ProxyRunner).FullName);
        proxy.MethodInsideAppDomain();
    }
}
```

В данном примере для того чтобы выбросить исключение любого типа из C# кода (я не хочу никого мучать вставками на MSIL) был проделан трюк с приведением типа к не сопоставимому: чтобы мы бросили исключение любого типа, а транслятор C#

думал бы что мы используем тип `Exception`. Мы создаем экземпляр типа `Program` - гарантированно не сериализуемого и бросаем исключение с ним в виде полезной нагрузки. Хорошие новости заключаются в том, что вы получите обертку над не-Exception исключениями `RuntimeWrappedException`, который внутри себя сохранит экземпляр нашего объекта типа `Program` и в C# перехватить такое исключение мы сможем. Однако есть и плохая новость, которая подтверждает наше предположение: вызов `proxy.MethodInsideAppDomain();` приведет к исключению `SerializationException`:



Т.е. проброс между доменами такого исключения не возможен, т.к. его нет возможности сериализовать. А это в свою очередь значит, что оборачивание вызовов методов, находящихся в других доменах фильтрами исключений все равно приведет к развертке стека несмотря на то что при `FullTrust` настройках дочернего домена сериализация казалось бы не нужна.

> Стоит дополнительно обратить внимание на причину, по которой сериализация между доменами так необходима. В нашем синтетическом примере мы создаем дочерний домен, который не имеет никаких настроек. А это значит, что он работает в FullTrust. Т.е. CLR полностью доверяет его содержимому как себе и никаких дополнительных проверок делать не будет. Но как только вы выставите хоть одну настройку безопасности, полная доверенность пропадет и CLR начнет контролировать все что происходит внутри этого дочернего домена. Так вот когда домен полностью доверенный, сериализация по идее не нужна. Нам нет необходимости как-то защищаться, согласитесь. Но сериализация создана не только для защиты. Каждый домен грузит в себя все необходимые собрки по второму разу, создавая их копии. Тем самым создавая копии всех типов и всех таблиц виртуальных методов. Передавая объект по ссылке из домена в домен вы получите, конечно, тот же объект. Но у него будут чужие таблицы виртуальных методов и как результат - этот объект не сможет быть приведен к другому типу. Другими словами, если вы создали экземпляр типа `Boo`, то получив его в другом домене приведение типа `(Boo)boo` не сработает. А сериализация и десериализация решает проблему: объект будет существовать одновременно в двух доменах. Там где он был создан - со всеми своими данными и в доменах использования - в виде прокси-объекта, обеспечивающего вызов методов оригинального объекта.

Передавая серализуемый объект между доменами, вы получите в другом домене полную копию объекта из первого сохранив некоторую разграниченность по памяти. Разграниченность тоже мнимая. Она - только для тех типов, которые не находятся в `Shared AppDomain`. Т.е., например, если в качестве исключения бросить

что-нибудь несериализуемое, но из `Shared AppDomain`, то ошибки сериализации не будет (можно попробовать вместо `Program`бросить `Action`). Однако раскрутка стека при этом все равно произойдет: оба случая должны работать стандартно. Чтобы никого не путать.

# Архитектура исключительной ситуации

Наверное, один из самых важных вопросов, который касается нашей темы - это вопрос построения архитектуры исключений в вашем приложении. Этот вопрос интересен по многим причинам. Как по мне так основная - это видимая простота, с которой не всегда очевидно, что делать. Это свойство присуще всем базовым конструкциям, которые используются повсеместно: это и `IEnumerable`, и `IDisposable` и `IObservable` и прочие-прочие. С одной стороны, своей простотой они манят, вовлекают в использование себя в самых разных ситуациях. А с другой стороны, они полны омутов и бродов, из которых, не зная, как иной раз и не выбраться вовсе. И, возможно, глядя на будущий объем у вас созрел вопрос: так что же такого в исключительных ситуациях?

Но для того чтобы прийти к каким-то выводам относительно построения архитектуры классов исключительных ситуаций мы должны с вами скопить некоторый опыт относительно их классификации. Ведь только поняв, с чем мы будем иметь дело, как и в каких ситуациях программист должен выбирать тип ошибки, а в каких - делать выбор относительно перехвата или пропуска исключений, можно понять как можно построить систему типов таким образом чтобы это стало очевидно для пользователя вашего кода. А потому, попробуем классифицировать исключительные ситуации (не сами типы исключений, а именно ситуации) по различным признакам.

## По теоретической возможности перехвата проектируемого исключения

По теоретическому перехвату исключения можно легко поделить на два вида: на те, которые перехватывать будут точно и на те, которые с высокой степенью вероятности перехватывать не будут. Почему *с высокой степенью вероятности*? Потому что всегда найдется тот, кто попробует перехватить, хотя это и не нужно было совершенно делать.

Давайте для начала раскроем особенности первой группы: исключения, которые должны и будут перехватывать.

Когда мы вводим исключение такого типа, то одной стороны мы сообщаем внешней подсистеме что мы вошли в положение, когда дальнейшие действия в рамках наших данных не имеют смысла. А с другой имеем в виду, что ничего глобального сломано не было и если нас убрать, то ничего не поменяется, а потому это исключение может быть легко перехвачено чтобы поправить ситуацию. Это свойство очень важно: именно оно определяет критичность ошибки и уверенность в том что если перехватить исключение и просто очистить ресурсы, то можно спокойно выполнять код дальше.

Вторая группа как бы это ни звучало странно - отвечает за исключения, которые перехватывать не нужно. Они могут быть использованы только для записи в журнал ошибок, но не для того чтобы можно было как-то поправить ситуацию. Самый простой пример - это исключения группы `ArgumentException` и `NullReferenceException`. Ведь в нормальной ситуации вы не должны, например, перехватывать исключение `ArgumentNullException` потому что источником проблемы тут будете являться именно вы, а не кто либо еще. Если вы перехватываете данное исключение, то тем самым вы допускаете что вы ошиблись и отдали методу то, что отдавать ему было нельзя:

```
void SomeMethod(object argument)
{
    try {
        AnotherMethod(argument);
    } catch (ArgumentNullException exception)
    {
        // Log it
    }
}
```

В этом методе мы пробуем перехватить `ArgumentNullException`. Но на мой взгляд его перехват выглядит очень странным: прокинуть корректные аргументы методу - полностью наша забота. Было бы не корректно среагировать постфактум: в такой ситуации самое правильное что только можно сделать - это проверить передаваемые данные заранее, до вызова метода или же что еще лучше - построить

код таким образом чтобы получение неправильных параметров было бы попросту не возможным.

Еще одна группа - это исключения фатальных ошибок. Если сломан некий кэш и работа подсистемы в любом случае будет не корректной? Тогда это - фатальная ошибка и ближайший по стеку код ее перехватывать гарантированно не станет:

```
T GetFromCacheOrCalculate()
{
    try {
        if(_cache.TryGetValue(Key, out var result))
        {
            return result;
        } else {
            T res = Strategy(Key);
            _cache[Key] = res;
            return res;
        }
    } cache (CacheCorreptedException exception)
    {
        RecreateCache();
        return GetFromCacheOrCalculate();
    }
}
```

И пусть `CacheCorreptedException` - это исключение, означающее "кэш на жестком диске не консистентен". Тогда получается, что если причина такой ошибки фатальна для подсистемы кэширования (например, отсутствуют права доступа к файлу кэша), то дальнейший код если не сможет пересоздать кэш командой `RecreateCache`, а потому факт перехвата этого исключения является ошибкой сам по себе.

## По фактическому перехвату исключительной ситуации

Еще один вопрос, который останавливает наш полет мысли в программировании алгоритмов - это понимание: стоит ли перехватывать те или иные исключения или же стоит пропустить их сквозь себя кому-то более понимающему. Переводя на язык терминов вопрос, который нам надо решить - разграничить зоны ответственности. Давайте рассмотрим следующий код:

```
namespace JetFinance.Strategies
{
    public class WildStrategy : StrategyBase
    {
        private Random random =  new Random();

        public void PlayRussianRoulette()
        {
            if(DateTime.Now.Second == (random.Next() % 60))
```

```
            {
                throw new StrategyException();
            }
        }
    }

    public class StrategyException : Exception { /* .. */ }
}

namespace JetFinance.Investments
{
    public class WildInvestment
    {
        WildStrategy _strategy;

        public WildInvestment(WildStrategy strategy)
        {
            _strategy = strategy;
        }

        public void DoSomethingWild()
        {
            ?try?
            {
                _strategy.PlayRussianRoulette();
            }
            catch(StrategyException exception)
            {
            }
        }
    }
}

using JetFinance.Strategies;
using JetFinance.Investments;

void Main()
{
    var foo = new WildStrategy();
    var boo = new WildInvestment(foo);

    ?try?
    {
        boo.DoSomethingWild();
    }
    catch(StrategyException exception)
    {
    }
}
```

Какая из двух предложенных стратегий является более корректной? Зона ответственности - это очень важно. Изначально может показаться, что поскольку работа `WildInvestment` и его консистентность целиком и полностью зависит от `WildStrategy`, то если `WildInvestment` просто проигнорирует данное исключение, оно уйдет в уровень повыше и делать ничего более не надо. Однако, прошу заметить что существует чисто архитектурная проблема: метод `Main` ловит

исключение из архитектурно одного слоя, вызывая метод архитектурно - другого. Как это выглядит с точки зрения использования? Да в общем так и выглядит:

- заботу об этом исключении просто перевесили на нас;
- у пользователя данного класса нет уверенности что это исключение прокинуто через ряд методов до нас специально
- мы начинаем тянуть лишние зависимости, от которых мы избавились, вызывая промежуточный слой.

Однако, из данного вывода следует другой: `catch` мы должны ставить в методе `DoSomethingWild`. И это для нас несколько странно: `WildInvestment` вроде как жестко зависим от кого-то. Т.е. если `PlayRussianRoulette` отработать не смог, то и `DoSomethingWild` тоже: кодов возврата тот не имеет, а сыграть в рулетку он обязан. Что же делать в такой казалось бы безвыходной ситуации? Ответ на самом деле прост: находясь в другом слое, `DoSomethingWild` должен выбросить собственное исключение, которое относится к этому слою и обернуть исходное как оригинальный источник проблемы - в `InnerException`:

```
namespace JetFinance.Strategies
{
    pubilc class WildStrategy
    {
        private Random random =  new Random();

        public void PlayRussianRoulette()
        {
            if(DateTime.Now.Second == (random.Next() % 60))
            {
                throw new StrategyException();
            }
        }
    }

    public class StrategyException : Exception { /* .. */ }
}

namespace JetFinance.Investments
{
    public class WildInvestment
    {
        WildStrategy _strategy;

        public WildInvestment(WildStrategy strategy)
        {
            _strategy = strategy;
        }
```

```
        public void DoSomethingWild()
        {
            try
            {
                _strategy.PlayRussianRoulette();
            }
            catch(StrategyException exception)
            {
                throw new FailedInvestmentException("Oops", exception);
            }
        }
    }

    public class InvestmentException : Exception { /* .. */ }

    public class FailedInvestmentException : Exception { /* .. */ }
}

using JetFinance.Investments;

void Main()
{
    var foo = new WildStrategy();
    var boo = new WildInvestment(foo);

    try
    {
        boo.DoSomethingWild();
    }
    catch(FailedInvestmentException exception)
    {
    }
}
```

Обернув исключение другим мы по сути переводим проблематику из одного слоя приложения в другой, сделав его работу более предсказуемой с точки зрения пользователя этого класса: метода `Main`.

## По вопросам переиспользования

Очень часто перед нами встает непростая задача: с одной стороны нам лень создавать новый тип исключения, а когда мы все-таки решаемся, не всегда ясно от чего отталкиваться: какой тип взять за основу как базовый. А ведь именно эти решения и определяют всю архитектуру исключительных ситуаций. Давайте пробежимся по популярым решениям и сделаем некоторые выводы.

При выборе типа исключений можно попробовать взять уже существующее решение: найти исключение с похожим смыслом в названии и использовать его. Например, если нам отдали через параметр какую-либо сущность, которая нас почему-то не устраивает, мы можем выбросить `InvalidArgumentException`, указав

причину ошибки - в Message. Этот сценарий выглядит хорошо, особенно с учетом того что `InvalidArgumentException` находится в группе исключений, которые не подлежат обязательному перехвату. Но плохим будет выбор `InvalidDataException` если вы работаете с какими-либо данными. Просто потому что этот тип находится в зоне `System.IO`, а это врядли то, чем вы занимаетесь. Т.е. получается что найти существующий тип потому что лениво делать свой - практически всегда будет не правильным подходом. Исключений, которые созданы для общего круга задач почти не существует. Практически все из них созданы под конкретные ситуации и их переиспользование будет грубым нарушением архитектуры исключительных ситуаций. Мало того, получив исключение определенного типа (например, тот же `System.IO.InvalidDataException`), пользователь будет запутан: с одной стороны он увидит источник проблемы в `System.IO` как пространство имен исключения, а с другой - совершенно другое пространство имен точки выброса. Плюс ко всему, задумавшись о правилах выброса этого исключения зайдет на referencesource.microsoft.com и найдет все места его выброса:

- `internal class System.IO.Compression.Inflater`

И поймет что ~~просто у кого-то кривые руки~~ выбор типа исключения его запутал, поскольку метод, выбросивший исключение компрессией не занимался.

Также в целях упрощения переиспользования можно просто взять и создать какое-то одно исключение, объявив у него поле `ErrorCode` с кодом ошибки и жить себе припеваючи. Казалось бы: хорошее решение. Бросаете везде одно и то же исключение, выставив код, ловите всего-навсего одним `catch` повышая тем самым стабильность приложения: и делать более ничего не надо. Однако, прошу не согласиться с такой позицией. Действуя таким образом по всему приложению вы с одной стороны, конечно, упрощаете себе жизнь. Но с другой - вы отбрасываете возможность ловить подгруппу исключений, объединенных некоторой общей особенностью. Как это сделано, например, с `ArgumentException`, который под собой объединяет целую группу исключений путем наследования. Второй серьезный минус - чрезмерно большие и нечитаемые простыни кода, который будет организовывать фильтрацию по коду ошибки. А вот если взять другую ситуацию: когда конечному пользователю конкретизация ошибки не должна быть важна, введение обобщающего типа плюс код ошибки выглядит уже куда более правильным применением:

```
public class ParserException
```

```
{
    public ParserError ErrorCode { get; }

    public ParserException(ParserError errorCode)
    {
        ErrorCode = errorCode;
    }

    public override string Message
    {
        get {
            return
Resources.GetResource($"{nameof(ParserException)}{Enum.GetName(typeof(ParserError),
ErrorCode)}");
        }
    }
}

public enum ParserError
{
    MissingModifier,
    MissingBracket,
    // ...
}

// Usage
throw new ParserException(ParserError.MissingModifier);
```

Коду, который защищает вызов парсера почти всегда безразлично, по какой причине был завален парсинг: ему важен сам факт ошибки. Однако, если это все-таки станет важно, пользователь всегда сможет вычленить код ошибки из свойства `ErrorCode`. Для этого вовсе не обязательно искать нужные слова по подстроке в `Message`.

Если отталкиваться от игнорирования вопросов переиспользования, то можно создать по типу исключения под каждую ситуацию. С одной стороны это выглядит логично: один тип ошибки - один тип исключения. Однако, тут, как и во всем, главное не переусердствовать: имея по типу исключительных операций на каждую точку выброса вы порождаете тем самым проблемы для перехвата: код вызывающего метода будет перегружен блоками `catch`. Ведь ему надо обработать все типы исключений, которые вы хотите ему отдать. Другой минус - чисто архитектурный. Если вы не используете наследования, то тем самым дезориентируете пользователя этих исключений: между ними может быть много общего, а перехватывать их приходится по отдельности.

Тем не менее существуют хорошие сценарии для введения отдельных типов для конкретных ситуаций. Например, когда поломка происходит не для всей сущности в целом, а для конкретного метода. Тогда этот тип должен быть в иерархии

наследования находиться в таком месте чтобы не возникало мысли его перехватить заодно с чем-то еще: например, выделив его через отдельную ветвь наследования.

Дополнительно, если объединить эти два подхода, можно получить очень мощный инструментарий по работе с группой ошибок: можно ввести обобщающий абстрактный тип, от которого унаследовать конкретные частные ситуации. Базовый класс (наш обобщающий тип) необходимо снабдить абстрактным свойством, хранящем код ошибки, а наследники переопределяя это свойство будут этот код ошибки уточнять:

```
public abstract class ParserException
{
    public abstract ParserError ErrorCode { get; }

    public override string Message
    {
        get {
            return
Resources.GetResource($"{nameof(ParserException)}{Enum.GetName(typeof(ParserError),
ErrorCode)}");
        }
    }
}

public enum ParserError
{
    MissingModifier,
    MissingBracket
}

public class MissingModifierParserException : ParserException
{
    public override ParserError ErrorCode { get; } => ParserError.MissingModifier;
}

public class MissingBracketParserException : ParserException
{
    public override ParserError ErrorCode { get; } => ParserError.MissingBracket;
}

// Usage
throw new MissingModifierParserException(ParserError.MissingModifier);
```

Какие замечательные свойства мы получим при таком подходе?

- с одной стороны мы сохранили перехват исключения по базовому типу;
- с другой стороны, перехватив исключение по базовому типу сохранилась возможность узнать конкретную ситуацию;

- и плюс ко всему можно перехватить по конкретному типу, а не по базовому, не пользуясь плоской структурой классов.

Как по мне так очень удобный вариант.

## По отношению к единой группе поведенческих ситуаций

Какие же выводы можно сделать, основываясь на ранее описанных рассуждениях? Давайте попробуем их сформулировать:

Для начала давайте определимся, что имеется ввиду под ситуациями. Когда мы говорим про классы и объекты, то мы привыкли в первую очередь оперировать сущностями с некоторым внутренним состоянием над которыми можно осуществлять действия. Получается, что тем самым мы нашли первый тип поведенческой ситуации: действия над некоторой сущностью. Далее, если посмотреть на граф объектов как-бы со стороны, можно заметить, что он логически объединен в функциональные группы: первая занимается кэшированием, вторая - работа с базами данных, третья осуществляет математические расчеты. Через все эти функциональные группы могут идти слои: слой логгирования различных внутренних состояний, журналирование процессов, трассировка вызовов методов. Слои могут быть более охватывающие: объединяющие в себе несколько функциональных групп. Например, слой модели, слой контроллеров, слой представления. Эти группы могут находиться как в одной сборке, так и в совершенно разных, но каждая из них может создавать свои исключительные ситуации.

Получается, что если рассуждать таким образом, то можно построить некоторую иерархию типов исключительных ситуаций, основываясь на принадлежности типа той или иной группе или слою создавая тем самым возможность перехватывающему исключения коду легкую смысловую навигацию в этой иерархии типов.

Давайте рассмотрим код:

```
namespace JetFinance
{
```

```
    namespace FinancialPipe
    {
        namespace Services
        {
            namespace XmlParserService
            {
            }

            namespace JsonCompilerService
            {
            }

            namespace TransactionalPostman
            {
            }
        }
    }

    namespace Accounting
    {
        /* ... */
    }
}
```

На что это похоже? Как по мне, пространства имен - прекрасная возможность естественной группировки типов исключений по их поведенческим ситуациям: все, что принадлежит определенным группам там и должно находиться, включая исключения. Мало того, когда вы получите определенное исключение, то помимо названия его типа вы увидете и его пространство имен, что четко определит его принадлежность. Помните пример плохого переиспользования типа `InvalidDataException`, который на самом деле определен в пространстве имен `System.IO`? Его принадлежность данному пространству имен означает что по сути исключение этого типа может быть выброшено из классов, находящихся в пространстве имен `System.IO` либо в более вложенном. Но само исключение при этом было выброшено совершенно из другого места, запутывая исследователя возникшей проблемы. Сосредотачивая типы исключений по тем же пространствам имен, что и типы, эти исключения выбрасывающие, вы с одной стороны сохраняете архитектуру типов консистентной, а с другой - облегчаете понимание причин произошедшего конечным разработчиком.

Каков второй путь группировки на уровне кода? Наследование:

```
public abstract class LoggerExceptionBase : Exception
{
    protected LoggerException(..);
}
```

```
public class IOLoggerException : LoggerExceptionBase
{
    internal IOLoggerException(..);
}

public class ConfigLoggerException : LoggerExceptionBase
{
    internal ConfigLoggerException(..);
}
```

Причем, если в случае с обычными сущностями приложения наследование означает наследование поведения и данных, объединяя типы по принадлежности к *единой группе сущностей*, то в случае исключений наследование означает принадлежность к *единой группе ситуаций*, поскольку суть исключения - не сущность, а проблематика.

Объединяя оба метода группировки, можно сделать некоторые выводы:

- внутри сборки (`Assembly`) должен присутствовать базовый тип исключений, которые данная сборка выбрасывает. Этот тип исключений должен находиться в корневом для сборки пространстве имен. Это будет первый слой группировки;
- далее внутри самой сборки может быть одно или несколько различных пространств имен. Каждое из них делит сборку на некоторые функциональные зоны, тем самым определяя группы ситуаций, которые в данной сборке возникают. Это могут быть зоны контроллеров, сущностей баз данных, алгоритмов обработки данных и прочих. Для нас эти пространства имен - группировка типов по функциональной принадлежности, а с точки зрения исключений - группировка по проблемным зонам этой же сборки;
- наследование исключений может идти только от типов в этом же пространстве имен либо в более корневом. Это гарантирует однозначное понимание ситуации конечным пользователем и отсутствие перехвата *левых* исключений при перехвате по базовому типу. Согласитесь: было бы странно получить `global::Finiki.Logistics.OhMyException`, имея `catch(global::Legacy.LoggerExeption exception)`, зато абсолютно гармонично выглядит следующий код:

```
namespace JetFinance.FinancialPipe
{
    namespace Services.XmlParserService
    {
        public class XmlParserServiceException : FinancialPipeExceptionBase
        {
            // ..
        }

        public class Parser
        {
            public void Parse(string input)
            {
                // ..
            }
        }
    }

    public abstract class FinancialPipeExceptionBase : Exception
    {

    }
}

using JetFinance.FinancialPipe;
using JetFinance.FinancialPipe.Services.XmlParserService;

var parser = new Parser();

try {
    parser.Parse();
}
catch (XmlParserServiceException exception)
{
    // Something wrong in parser
}
catch (FinancialPipeExceptionBase exception)
{
    // Something else wrong. Looks critical because we don't know real reason
}
```

Заметьте, что тут происходит: мы как пользовательский код вызываем некий библиотечный метод, который, насколько мы знаем, может при некоторых обстоятельствах выбросить исключение XmlParserServiceException. И, насколько мы знаем, это исключение находится в пространстве имен, наследуя JetFinance.FinancialPipe.FinancialPipeExceptionBase, что говорит о возможном упущении других исключений: это сейчас микросервис XmlParserService создает только одно исключение, но в будущем могут появиться и другие. И поскольку у нас есть конвенция в создании типов исключений, мы точно знаем от кого это новое исключение будет наследоваться и заранее ставим

обобщающий catch не затрагивая при этом ничего лишнего: то что не попало в зону нашей ответственности пролетит мимо.

Как же построить иерархию типов?

- Для начала необходимо сделать базовый класс для домена. Назовем его доменным базовым классом. Домен в данном случае - это обобщающее некоторое количество сборок слово, которое объединяет их по некоторому глобальному признаку: логгирование, бизнес-логика, UI. Т.е. максимально крупные функциональные зоны приложения;
- Далее необходимо ввести дополнительный базовый класс для исключений, которые перехватывать необходимо: от него будут наследоваться все исключение, которые будут перехватываться ключевым словом `catch`;
- Все исключения которые обозначают фатальные ошибки – наследовать напрямую от доменного базового класса. Тем самым вы отделили их от перехватываемых архитектурно;
- Разделить домен на функциональные зоны по пространствам имен и объявить базовый тип исключений, которые будут выбрасываться из каждой функциональной зоны. Тут стоит дополнительно орудовать здравым смыслом: если приложение имеет большую вложенность пространств имен, то делать по базовому типу для каждого уровня вложенности, конечно, не стоит. Однако, если на каком-то уровне вложенности происходит ветвление: одна группа исключений ушла в одно подпространство имен, а другая - в другое, то тут, конечно, стоит ввести два базовых типа для каждой подгруппы;
- Частные исключения наследовать от типов исключений функциональных зон
- Если группа частных исключений может быть объединена, объединить их еще одним базовым типом: так вы упрощаете их перехват;
- Если предполагается что группа будет чаще перехватываться по своему базовому классу, ввести Mixed Mode с ErrorCode.

## По источнику ошибки

Еще одним поводом для объединения исключений в некоторую группу может выступать источник ошибки. Например, если вы разрабатываете библиотеку классов, то группами источников могут стать:

- Вызов unsafe кода, который отработал с ошибкой. Данную ситуацию следует обработать следующим образом: обернуть исключение либо код ошибки в собственный тип исключения, а полученные данные об ошибке (например, оригинальный код ошибки) сохранить в публичном свойстве исключения;
- Вызов кода из внешних зависимостей, который вызвал исключения, которые наша библиотека перехватить не может, т.к. они не входят в ее зону ответственности. Сюда могут входить исключения из методов тех сущностей, которые были приняты в качестве параметров текущего метода или же конструктора того класса, метод которого вызвал внешнюю зависимость. Как пример, метод нашего класса вызвал метод другого класса, экземпляр которого был получен через параметры метода. Если исключение говорит о том что источником проблемы были мы сами - генерируем наше собственное исключение с сохранением оригинального - в `InnerExcepton`. Если же мы понимаем что проблема именно в работе внешней зависимости - пропускаем исключение насквозь как принадлежащее к группе внешних непоконтрольных зависимостей;
- Наш собственный код, который был случайным образом введен в не консистентное состояние. Хорошим примером может стать парсинг текста. Внешних зависимостей нет, ухода в `unsafe` нет, а ошибка парсинга есть.

## События об исключительных ситуациях

В общем случае мы не всегда знаем о тех исключениях, которые произойдут в наших программах потому что практически всегда мы используем что-то, что написано другими людьми и что находится в других подсистемах и библиотеках. Мало того что возможны самые разные ситуации в вашем собственном коде, в коде других библиотек, так еще и существует множество проблем, связанных с исполнением кода в изолированных доменах. И как раз в этом случае было бы крайне полезно уметь получать данные о работе изолированного кода. Ведь вполне реальной может быть ситуация, когда сторонний код перехватывает все без исключения ошибки, заглушив их `fault` блоком:

```
try {
    // ...
} catch {
    // do nothing, just to make code call more safe
}
```

В такой ситуации может оказаться что выполнение кода уже не так безопасно как выглядит, но сообщений о том что произошли какие-то проблемы мы не имеем. Второй вариант - когда приложение глушит некоторое, пусть даже легальное, исключение. А результат - следующее исключение в случайном месте вызовет падение приложения в некотором будущем от случайной казалось бы ошибки. Тут хотелось бы иметь представление, какая была предыстория этой ошибки. Каков ход событий привел к такой ситуации. И один из способов сделать это возможным - использовать дополнительные события, которые относятся к исключительным ситуациям: `AppDomain.FirstChanceException` и `AppDomain.UnhandledException`.

Фактически, когда вы "бросаете исключение", то вызывается обычный метод некоторой внутренней подсистемы `Throw`, который внутри себя проделывает следующие операции:

- Вызывает `AppDomain.FirstChanceException`
- Ищет в цепочке обработчиков подходящий по фильтрам
- Вызывает обработчик предварительно откатив стек на нужный кадр

- Если обработчик найден не был, вызывается `AppDomain.UnhandledException`, обрушивая поток, в котором произошло исключение.

Сразу следует оговориться, ответив на мучающий многие умы вопрос: есть ли возможность как-то отменить исключение, возникшее в неконтролируемом коде, который исполняется в изолированном домене, не обрушивая тем самым поток, в котором это исключение было выброшено? Ответ лаконичен и прост: нет. Если исключение не перехватывается на всем диапазоне вызванных методов, оно не может быть обработано в принципе. Иначе возникает странная ситуация: если мы при помощи `AppDomain.FirstChanceException` обрабатываем (некий синтетический `catch`) исключение, то на какой кадр должен откатиться стек потока? Как это задать в рамках правил .NET CLR? Никак. Это просто не возможно. Единственное что мы можем сделать - запротоколировать полученную информацию для будущих исследований.

Второе, о чем следует рассказать "на берегу" - это почему эти события введены не у `Thread`, а у `AppDomain`. Ведь если следовать логике, исключения возникают где? В потоке исполнения команд. Т.е. фактически у `Thread`. Так почему же проблемы возникают у домена? Ответ очень прост: для каких ситуаций создавались `AppDomain.FirstChanceException` и `AppDomain.UnhandledException`? Помимо всего прочего - для создания песочниц для плагинов. Т.е. для ситуаций, когда есть некий `AppDomain`, который настроен на PartialTrust. Внутри этого AppDomain может происходить что угодно: там в любой момент могут создаваться потоки, или использоваться уже существующие из ThreadPool. Тогда получается что мы, будучи находясь снаружи от этого процесса (не мы писали тот код) не можем никак подписаться на события внутренних потоков. Просто потому что мы понятия не имеем что там за потоки были созданы. Зато мы гарантированно имеем `AppDomain`, который организует песочницу и ссылка на который у нас есть.

Итак, по факту нам предоставляется два краевых события: что-то произошло, чего не предполагалось (`FirstChanceExecption`) и "все плохо", никто не обработал исключительную ситуацию: она оказалась не предусмотренной. А потому поток исполнения команд не имеет смысла и он (`Thread`) будет отгружен.

Что можно получить, имея данные события и почему плохо что разработчики обходят эти события стороной?

## AppDomain.FirstChanceException

Это событие по своей сути носит чисто информационный характер и не может быть "обработано". Его задача - уведомить вас что в рамках данного домена произошло исключение и оно после обработки события начнет обрабатываться кодом приложения. Его исполнение несет за собой пару особенностей, о которых необходимо помнить во время проектирования обработчика.

Но давайте для начала посмотрим на простой синтетический пример его обработки:

```
void Main()
{
    var counter = 0;

    AppDomain.CurrentDomain.FirstChanceException += (_, args) => {
        Console.WriteLine(args.Exception.Message);
        if(++counter == 1) {
            throw new ArgumentOutOfRangeException();
        }
    };

    throw new Exception("Hello!");
}
```

Чем примечателен данный код? Где бы некий код ни сгенерировал бы исключение первое что произойдет - это его логгирование в консоль. Т.е. даже если вы забудете или не сможете предусмотреть обработку некоторого типа исключения оно все равно появится в журнале событий, которое вы организуете. Второе - несколько странное условие выброса внутреннего исключения. Все дело в том что внутри обработчика `FirstChanceException` вы не можете просто взять и бросить еще одно исключение. Скорее даже так: внутри обработчика FirstChanceException вы *не имеете возможности* бросить хоть какое-либо исключение. Если вы так сделаете, возможны два варианта событий. При первом, если бы не было условия `if(++counter == 1)`, мы бы получили бесконечный выброс `FirstChanceException` для все новых и новых `ArgumentOutOfRangeException`. А что это значит? Это значит, что на определенном этапе мы бы получили `StackOverflowException`: `throw new Exception("Hello!")` вызывает CLR метод Throw, который

вызывает `FirstChanceException`, который вызывает `Throw` уже для `ArgumentOutOfRangeException` и далее - по рекурсии. Второй вариант - мы защитились по глубине рекурсии при помощи условия по `counter`. Т.е. в данном случае мы бросаем исключение только один раз. Результат более чем неожиданный: мы получим исключительную ситуацию, которая фактически отрабатывает внутри инструкции `Throw`. А что подходит более всего для данного типа ошибки? Согласно ECMA-335 если инструкция была введена в исключительное положение, должно быть выброшено `ExecutionEngineException`! А эту исключительную ситуацию мы обработать никак не в состоянии. Она приводит к полному вылету из приложения. Какие же варианты безопасной обработки у нас есть?

Первое, что приходит в голову - это выставить `try-catch` блок на весь код обработчика `FirstChanceException`:

```
void Main()
{
    var fceStarted = false;
    var sync = new object();
    EventHandler<FirstChanceExceptionEventArgs> handler;
    handler = new EventHandler<FirstChanceExceptionEventArgs>((_, args) =>
    {
        lock (sync)
        {
            if (fceStarted)
            {
                // Этот код по сути - заглушка, призванная уведомить что исключение по
своей сути - родилось не в основном коде приложения,
                // а в try блоке ниже.
                Console.WriteLine($"FirstChanceException inside FirstChanceException
({args.Exception.GetType().FullName})");
                return;
            }
            fceStarted = true;

            try
            {
                // не безопасное логгирование куда угодно. Например, в БД
                Console.WriteLine(args.Exception.Message);
                throw new ArgumentOutOfRangeException();
            }
            catch (Exception exception)
            {
                // это логгирование должно быть максимально безопасным
                Console.WriteLine("Success");
            }
            finally
            {
```

```
                fceStarted = false;
            }
        }
    });
    AppDomain.CurrentDomain.FirstChanceException += handler;

    try
    {
        throw new Exception("Hello!");
    } finally {
        AppDomain.CurrentDomain.FirstChanceException -= handler;
    }
}

OUTPUT:

Hello!
Specified argument was out of the range of valid values.
FirstChanceException inside FirstChanceException (System.ArgumentOutOfRangeException)
Success

!Exception: Hello!
```

Т.е. с одной стороны у нас есть код обработки события FirstChanceException, а с другой - дополнительный код обработки исключений в самом FirstChanceException. Однако методики логгирования обоих ситуаций должны отличаться. Если логгирование обработки события может идти как угодно, то обработка ошибки логики обработки FirstChanceException должно идти без исключительных ситуаций в принципе. Второе, что вы наверняка заметили - это синхронизация между потоками. Тут может возникнуть вопрос: зачем она тут если любое исключение рождено в каком-либо потоке а значит FirstChanceException по идее должен быть потокобезопасным. Однако, все не так жизнерадостно. FirstChanceException у нас возникает у AppDomain. А это значит, что он возникает для любого потока, стартованного в определенном домене. Т.е. если у нас есть домен, внутри которого стартовано несколько потоков, то FirstChanceException могут идти в параллель. А это значит, что нам необходимо как-то защитить себя синхронизацией: например при помощи lock.

Второй способ - попробовать увести обработку в соседний поток, принадлежащий другому домену приложений. Однако тут стоит оговориться что при такой реализации мы должны построить выделенный домен именно под эту задачу чтобы не получилось так что этот домен могут положить другие потоки, которые являются рабочими:

```
static void  Main()
{
    using (ApplicationLogger.Go(AppDomain.CurrentDomain))
    {
        throw new Exception("Hello!");
    }
}

public class ApplicationLogger : MarshalByRefObject
{
    ConcurrentQueue<Exception> queue = new ConcurrentQueue<Exception>();
    CancellationTokenSource cancellation;
    ManualResetEvent @event;

    public void LogFCE(Exception message)
    {
        queue.Enqueue(message);
    }

    private void StartThread()
    {
        cancellation = new CancellationTokenSource();
        @event = new ManualResetEvent(false);
        var thread = new Thread(() =>
        {
            while (!cancellation.IsCancellationRequested)
            {
                if (queue.TryDequeue(out var exception))
                {
                    Console.WriteLine(exception.Message);
                }
                Thread.Yield();
            }
            @event.Set();
        });
        thread.Start();
    }

    private void StopAndWait()
    {
        cancellation.Cancel();
        @event.WaitOne();
    }

    public static IDisposable Go(AppDomain observable)
    {
        var dom = AppDomain.CreateDomain("ApplicationLogger", null, new AppDomainSetup
        {
            ApplicationBase = AppDomain.CurrentDomain.BaseDirectory,
        });

        var                                  proxy                                  =
(ApplicationLogger)dom.CreateInstanceAndUnwrap(typeof(ApplicationLogger).Assembly.Ful
lName, typeof(ApplicationLogger).FullName);
        proxy.StartThread();

        var subscription = new EventHandler<FirstChanceExceptionEventArgs>((_, args)
=>
        {
            proxy.LogFCE(args.Exception);
        });
```

```
        observable.FirstChanceException += subscription;

        return new Subscription(() => {
            observable.FirstChanceException -= subscription;
            proxy.StopAndWait();
        });
    }

    private class Subscription : IDisposable
    {
        Action act;
        public Subscription (Action act) {
            this.act = act;
        }
        public void Dispose()
        {
            act();
        }
    }
}
```

В данном случае обработка `FirstChanceException` происходит максимально безопасно: в соседнем потоке, принадлежащим соседнему домену. Ошибки обработки сообщения при этом не могут обрушить рабочие потоки приложения. Плюс отдельно можно послушать UnhandledException домена логгирования сообщений: фатальные ошибки при логгировании не обрушат все приложение.

## AppDomain.UnhandledException

Второе сообщение, которое мы можем перехватить и которое касается обработки исключительных ситуаций - это `AppDomain.UnhandledException`. Это сообщение - очень плохая новость для нас поскольку обозначает что не нашлось никого кто смог бы найти способ обработки возникшей ошибки в некотором потоке. Также, если произошла такая ситуация, все что мы можем сделать - это "разгрести" последствия такой ошибки. Т.е. каким-либо образом зачистить ресурсы, принадлежащие только этому потоку если таковые создавались. Однако, еще более лучшая ситуация - обрабатывать исключения, находясь в "корне" потоков не заваливая поток. Т.е. по сути ставить `try-catch`. Давайте попробуем рассмотреть целесообразность такого поведения.

Пусть мы имеем библиотеку, которой необходимо создавать потоки и осуществлять какую-то логику в этих потоках. Мы, как пользователи этой библиотеки интересуемся только гарантией вызовов API а также получением сообщений об

ошибках. Если библиотека будет рушить потоки не нотифицируя об этом, нам это мало чем может помочь. Мало того обрушение потока приведет к сообщению `AppDomain.UnhandledException`, в котором нет информации о том, какой конкретно поток лег на бок. Если же речь идет о нашем коде, обрушивающийся поток нам тоже вряд-ли будет полезным. Во всяком случае необходимости в этом я не встречал. Наша задача - обработать ошибки правильно, отдать информацию об их возникновении в журнал ошибок и корректно завершить работу потока. Т.е. по сути обернуть метод, с которого стартует поток в `try-catch`:

```
    ThreadPool.QueueUserWorkitem(_ => {
        using(Disposables aggregator = ...){
            try {
                // do work here, plus:
                aggregator.Add(subscriptions);
                aggregator.Add(dependantResources);
            } catch (Exception ex)
            {
                logger.Error(ex, "Unhandled exception");
            }
        }
    });
```

В такой схеме мы получим то что надо: с одной стороны мы не обрушим поток. С другой - корректно очистим локальные ресурсы если они были созданы. Ну и в довесок - организуем журналирование полученной ошибки. Но постойте, скажете вы. Как-то вы лихо соскочили с вопроса события `AppDomain.UnhandledException`. Неужели оно совсем не нужно? Нужно. Но только для того чтобы сообщить что мы забыли обернуть какие-то потоки в `try-catch` со всей необходимой логикой. Именно со всей: с логгированием и очисткой ресурсов. Иначе это будет совершенно не правильно: брать и гасить все исключения, как будто их и не было вовсе.

## CLR Exceptions

Существует ряд исключительных ситуаций, которые скажем так... Несколько более исключительны чем другие. Причем если попытаться их классифицировать, то как и было сказано в самом начале главы, есть исключения родом из самого .NET приложения, а есть исключения родом из unsafe мира. Их в свою очередь можно разделить на две подкатегории: исключительные ситуации ядра CLR (которое по своей сути - unsafe) и любой unsafe код внешних библиотек.

[todo]

*ThreadAbortException*

Вообще, это может показаться не очевидным, но существует четыре типа Thread Abort.

- Грубый вариант ThreadAbort, который, отрабатывая не может быть никак остановлен и который не запускает обработчиков исключительных ситуаций вообще включая секции `finally`
- Вызов метода Thread.Abort() на текущем потоке
- Асинхронное исключение ThreadAbortException, вызванное из другого потока
- Если во время выгрузки AppDomain существуют потоки, в рамках которых запущены методы, скомпилированные для этого домена, будет произведен ThreadAbort тех потоков, в которых эти методы запущены

> Стоит заметить что ThreadAbortException довольно часто используется в *большом* .NET Framework, однако его не существует на CoreCLR, .NET Core или же под Windows 8 "Modern app profile". Попробуем узнать, почему.

Итак, если мы имеем дело с не принципиальным типом обрыва потока, когда мы еще можем с ним что-то сделать (т.е. второй, третий и четвертый вариант), виртуальная машина при возникновении такого исключения начинает идти по всем обработчикам исключительных ситуаций и искать как обычно те, тип исключения которых является тем, что было выброшено либо более базовым. В нашем случае это три типа: `ThreadAbortException`, `Exception` и `object`(помним что Exception - это по своей сути - хранилище данных и тип исключения может быть любым. Даже `int`).

Отрабатывая *все* подходящие `catch` блоки виртуальная машина пробрасыват `ThreadAbortException` дальше по цепочке обработки исключений попутно входя во все `finally` блоки. В целом, ситуации в двух примерах, описанных ниже абсолютно одинаковые:

```
var thread = new Thread(() =>
{
    try {
        // ...
    } catch (Exception ex)
    {
        // ...
    }
});
thread.Start();
//...
thread.Abort();

var thread = new Thread(() =>
{
    try {
        // ...
    } catch (Exception ex)
    {
        // ...
        if(ex is ThreadAbortException)
        {
            throw;
        }
    }
});
thread.Start();
//...
thread.Abort();
```

Конечно же, всегда возникнет ситуация, когда возникающий ThreadAbort может быть нами вполне ожидаем. Тогда может возникнуть понятное желание его все-таки обработать. Как раз для таких случаев был разработан и открыт метод `Thread.ResetAbort()`, который делает именно то, что нам нужно: останавливает сквозной проброс исключения через всю цепочку обработчиков, делая его обработанным:

```
void Main()
{
    var barrier = new Barrier(2);

    var thread = new Thread(() =>
    {
        try {
            barrier.SignalAndWait();  // Breakpoint #1
            Thread.Sleep(TimeSpan.FromSeconds(30));
        }
        catch (ThreadAbortException exception)
        {
            "Resetting abort".Dump();
            Thread.ResetAbort();
        }
```

```
        "Catched successfully".Dump();
        barrier.SignalAndWait();     // Breakpoint #2
    });

    thread.Start();
    barrier.SignalAndWait();         // Breakpoint #1

    thread.Abort();
    barrier.SignalAndWait();         // Breakpoint #2
}

Output:
Resetting abort
Catched successfully
```

Однако реально ли стоит этим пользоваться? И стоит ли обижаться на разработчиков CoreCLR что там этот код попросту выпилен? Представьте что вы - пользователь кода, который по вашему мнению "повис" и у вас возникло непреодолимое желание вызвать для него `ThreadAbortException`. Когда вы хотите оборвать жизнь потока все чего вы хотите - чтобы он действительно завершил свою работу. Мало того, редкий алгоритм просто обрывает поток и бросает его, уходя к своим делам. Обычно внешний алгоритм решает дождаться корректного завершения операций. Или же наоборот: может решить что поток более уже ничего делать не будет, декрементирует некие внутренние счетчики и более не будет завязываться на то что есть какая-то многопоточная обработка какого-либо кода. Тут в общем не скажешь, что хуже. Я даже так вам скажу: отработав много лет программистом я до сих пор не могу вам дать прекрасный способ его вызова и обработки. Сами посудите: вы бросаете `ThreadAbort` не *прямо сейчас* а в любом случае спустя некоторое время после осознания безвыходности ситуации. Т.е. вы можете как попасть по обработчику `ThreadAbortException` так и промахнуться мимо него: "зависший код" мог оказаться вовсе не зависшим, а попросту долго работающим. И как раз в тот момент, когда вы хотели оборвать его жизнь, он мог вырваться из ожидания и корректно продолжить работу. Т.е. без лишней лирики выйти из блока `try-catch(ThreadAbortException) { Thread.ResetAbort(); }`. Что мы получим? Оборванный поток, который ни в чем не виноват. Шла уборщица, выдернула провод, сеть пропала. Метод ожидал таймаута, уборщица вернула провод, все заработало, но ваш контролирующий код не дождался и убил поток. Хорошо? Нет. Как-то можно защититься? Нет. Но вернемся к навящивой идее

легализации `Thread.Abort()`: мы кинули кувалдой в поток и ожидаем что он с вероятностью 100% оборвется, но этого может не произойти. Во-первых становится не понятно как его оборвать в таком случае. Ведь тут все может быть намного сложнее: в подвисшем потоке может быть такая логика, которая перехватывает `ThreadAbortException`, останавливает его при помощи `ResetAbort`, однако продолжает висеть из-за сломанной логики. Что тогда? Делать безусловный `thread.Interrupt()`? Попахивает попыткой обойти ошибку в логике программы грубыми методами. Плюс, я вам гарантирую что у вас поплывут утечки: `thread.Interrupt()` не будет заниматься вызовом `catch` и `finally`, а это значит что при всем опыте и сноровке очистить ресурсы вы не сможете: ваш поток просто исчезнет, а находясь в соседнем потоке вы можете не знать ссылок на все ресурсы, которые были заняты умирающим потоком. Также прошу заметить что в случае промаха `ThreadAbortException` мимо `catch(ThreadAbortException) { Thread.ResetAbort(); }` у вас точно также потекут ресурсы.

После того что вы прочитали чуть выше я надеюсь, вы остались в некотором состоянии запутанности и желания перечитать абзац. И это будет совершенно правильная мысль: это будет доказательством того что пользоваться `Thread.Abort()` попросту нельзя. Как и нельзя пользоваться `thread.Interrupt();`. Оба метода приводят к неконтролируемому поведению вашего приложения. По своей сути они нарушают принцип целостности: основной принцип .NET Framework.

Однако, чтобы понять для каких целей этот метод введен в эксплуатацию достаточно посмотреть исходные коды .NET Framework и найти места использования `Thread.ResetAbort()`. Ведь именно его наличие по сути легализует `thread.Abort()`.

**Класс ISAPIRuntime** ISAPIRuntime.cs

```
try {
    // ...
}
catch(Exception e) {
```

```
    try {
        WebBaseEvent.RaiseRuntimeError(e, this);
    } catch {}

    // Have we called HSE_REQ_DONE_WITH_SESSION?  If so, don't re-throw.
    if (wr != null && wr.Ecb == IntPtr.Zero) {
        if (pHttpCompletion != IntPtr.Zero) {
            UnsafeNativeMethods.SetDoneWithSessionCalled(pHttpCompletion);
        }
        // if this is a thread abort exception, cancel the abort
        if (e is ThreadAbortException) {
            Thread.ResetAbort();
        }
        // IMPORTANT: if this thread is being aborted because of an AppDomain.Unload,
        // the CLR will still throw an AppDomainUnloadedException. The native caller
        // must special case COR_E_APPDOMAINUNLOADED(0x80131014) and not
        // call HSE_REQ_DONE_WITH_SESSION more than once.
        return 0;
    }

    // re-throw if we have not called HSE_REQ_DONE_WITH_SESSION
    throw;
}
```

В данном примере происходит вызов некоторого внешнего кода и если тот был завершен не корректно: с `ThreadAbortException`, то при определенных условиях помечаем поток как более не прерываемый. Т.е. по сути обрабатываем `ThreadAbort`. Почему в данном конкретно случае мы обрываем `Thread.Abort`? Потому что в данном случае мы имеем дело с серверным кодом, а он в свою очередь вне зависимости от наших ошибок вернуть корректные коды ошибок вызывающей стороне. Обрыв потока привел бы к тому что сервер не смог бы вернуть нужный код ошибки пользователю, а это совершенно не правильно. Также оставлен комментарий о `Thread.Abort()`во время `AppDomin.Unload()`, что является экстримальной ситуацией для ThreadAbort поскольку такой процесс не остановить и даже если вы сделаете `Thread.ResetAbort`. Это хоть и остановит сам Abortion, но не остановит выгрузку потока с доменом, в котором он находится: поток же не может исполнять инструкции кода, загруженного в домен, который отгружен.

**Класс HttpContext** HttpContext.cs

```
internal void InvokeCancellableCallback(WaitCallback callback, Object state) {
    // ...

    try {
        BeginCancellablePeriod();  // request can be cancelled from this point
        try {
            callback(state);
        }
```

```
        finally {
            EndCancellablePeriod();  // request can be cancelled until this point
        }
        WaitForExceptionIfCancelled();  // wait outside of finally
    }
    catch (ThreadAbortException e) {
        if (e.ExceptionState != null &&
            e.ExceptionState is HttpApplication.CancelModuleException &&
            ((HttpApplication.CancelModuleException)e.ExceptionState).Timeout) {

            Thread.ResetAbort();
            PerfCounters.IncrementCounter(AppPerfCounter.REQUESTS_TIMED_OUT);

            throw new HttpException(SR.GetString(SR.Request_timed_out),
                                    null, WebEventCodes.RuntimeErrorRequestAbort);
        }
    }
}
```

Здесь приведен прекрасный пример перехода от неуправляемого асинхронного исключения `ThreadAbortException` к управляемому `HttpException` с логгированием ситуации в журнал счетчиков производительности.

**Класс HttpApplication** [HttpApplication.cs]

```
internal Exception ExecuteStep(IExecutionStep step, ref bool completedSynchronously)
{
   Exception error = null;

   try {
       try {

       // ...

       }
       catch (Exception e) {
           error = e;

           // ...

           // This might force ThreadAbortException to be thrown
           // automatically, because we consumed an exception that was
           // hiding ThreadAbortException behind it

           if (e is ThreadAbortException &&
               ((Thread.CurrentThread.ThreadState & ThreadState.AbortRequested) ==
0)) {
               //   Response.End   from   a   COM+   component   that   re-throws
ThreadAbortException
               // It is not a real ThreadAbort
               // VSWhidbey 178556
               error = null;
               _stepManager.CompleteRequest();
           }
       }
       catch {
           // ignore non-Exception objects that could be thrown
       }
   }
```

```
    catch (ThreadAbortException e) {
        // ThreadAbortException could be masked as another one
        // the try-catch above consumes all exceptions, only
        // ThreadAbortException can filter up here because it gets
        // auto rethrown if no other exception is thrown on catch
        if (e.ExceptionState != null && e.ExceptionState is CancelModuleException) {
            // one of ours (Response.End or timeout) -- cancel abort

            // ...

            Thread.ResetAbort();
        }
    }
}
```

Здесь описывается очень интересный случай: когда мы ждем не настоящий `ThreadAbort` (мне вот в некотором смысле жалко команду CLR и .NET Framework. Сколько не стандартных ситуаций им приходится обрабатывать, подумать страшно). Обработка ситуации идет в два этапа: внутренним обработчиком мы ловим `ThreadAbortException` но при этом проверяем наш поток на флаг реальной прерываемости. Если поток не помечен как прерывающийся, то на самом деле это не настоящий ThreadAbortException. Такие ситуации мы должны обработать соответствующим образом: спокойно поймать исключение и обработать его. Если же мы получаем настоящий ThreadAbort, то он уйдет во внешний `catch` поскольку `ThreadAbortException` должен войти во все подходящие обработчики. Если он удовлетворяет необходимым условиям, он также будет обработан путем очистки флага `ThreadState.AbortRequested`методом `Thread.ResetAbort()`.

Если говорить про примеры самого вызова `Thread.Abort()`, то все примеры кода в .NET Framework написаны так что могут быть переписаны без его использования. Для наглядности приведу только один:

**Класс QueuePathDialog** QueuePathDialog.cs

```
protected override void OnHandleCreated(EventArgs e)
{
    if (!populateThreadRan)
    {
        populateThreadRan = true;
        populateThread = new Thread(new ThreadStart(this.PopulateThread));
        populateThread.Start();
    }

    base.OnHandleCreated(e);
}
```

```
protected override void OnFormClosing(FormClosingEventArgs e)
{
    this.closed = true;

    if (populateThread != null)
    {
        populateThread.Abort();
    }

    base.OnFormClosing(e);
}

private void PopulateThread()
{
    try
    {
        IEnumerator messageQueues = MessageQueue.GetMessageQueueEnumerator();
        bool locate = true;
        while (locate)
        {
            // ...
            this.BeginInvoke(new FinishPopulateDelegate(this.OnPopulateTreeview), new
object[] { queues });
        }
    }
    catch
    {
        if (!this.closed)
            this.BeginInvoke(new ShowErrorDelegate(this.OnShowError), null);
    }

    if (!this.closed)
        this.BeginInvoke(new SelectQueueDelegate(this.OnSelectQueue), new object[] {
this.selectedQueue, 0 });
}
```

**TheradAbortException во время AppDomain.Unload**

Попробуем отгрузить AppDomain во время исполнения кода, который в него загружен. Для этого искусственно создадим не вполне нормальную ситуацию, но достаточно интересную с точки зрения исполнения кода. В данном примере у нас два потока: один создан для того чтобы получить в нем ThreadAbortException, а другой - основной. В основном мы создаем новый домен, в котором запускаем новый поток. Задача этого потока - уйти в основной домен. Чтобы методы дочернего домена остались бы только в Stack Trace. После этого основной домен отгружает дочерний:

```
class Program : MarshalByRefObject
{
    static void Main()
    {
```

```
        try
        {
            var domain = ApplicationLogger.Go(new Program());
            Thread.Sleep(300);
            AppDomain.Unload(domain);

        } catch (ThreadAbortException exception)
        {
            Console.WriteLine("Main AppDomain aborted too, {0}", exception.Message);
        }
    }

    public void InsideMainAppDomain()
    {
        try
        {
            Console.WriteLine($"InsideMainAppDomain()          called          inside
{AppDomain.CurrentDomain.FriendlyName} domain");

            // AppDomain.Unload will be called while this Sleep
            Thread.Sleep(-1);
        }
        catch (ThreadAbortException exception)
        {
            Console.WriteLine("Subdomain aborted, {0}", exception.Message);

            // This sleep to allow user to see console contents
            Thread.Sleep(-1);
        }
    }

    public class ApplicationLogger : MarshalByRefObject
    {
        private void StartThread(Program pro)
        {
            var thread = new Thread(() =>
            {
                pro.InsideMainAppDomain();
            });
            thread.Start();
        }

        public static AppDomain Go(Program pro)
        {
            var   dom   =   AppDomain.CreateDomain("ApplicationLogger",   null,   new
AppDomainSetup
            {
                ApplicationBase = AppDomain.CurrentDomain.BaseDirectory,
            });

            var                                proxy                                =
(ApplicationLogger)dom.CreateInstanceAndUnwrap(typeof(ApplicationLogger).Assembly.Ful
lName, typeof(ApplicationLogger).FullName);
            proxy.StartThread(pro);

            return dom;
        }
    }

}
```

Происходит крайне интересная вещь. Код выгрузки домена помимо самой выгрузки ищет вызванные в этом домене методы, которые еще не завершили работу в том числе в глубине стека вызова методов и вызывает `ThreadAbortException` в этих потоках. Это важно, хоть и не очевидно: если домен отгружен, нельзя осуществить возврат в метод, из которого был вызван метод основного домена, но который находится в отгружаемом. Т.е. другими словами AppDomain.Unload может выбрасывать потоки, исполняющие в данный момент код из других доменов. Прервать `Thread.Abort` в таком случае не получится: исполнять код выгруженного домена вы не сможете, а значит `Thread.Abort` завершит свое дело, даже если вы вызовите `Thread.ResetAbort`.

**Выводы по ThreadAbortException**

- Это - асинхронное исключение, а значит оно может возникнуть в любой точке вашего кода (но, стоит отметить, что для этого надо постараться);
- Обычно код обрабатывает только те ошибки, которые ждет: нет доступа к файлу, ошибка парсинга строки и прочие подобные. Наличие асинхронного (в плане возникновения в любом месте кода) исключения создает ситуацию, когда try-catch могут быть не обработаны: вы же не можете быть готовым к ThreadAbort в любом месте приложения. И получается, что это исключение в любом случае породит утечки;
- Обрыв потока может также происходить из-за выгрузки какого-либо домена. Если в Stack Trace потока существуют вызовы методов отгружаемого домена, поток получит ThreadAbortException без возможности `ResetAbort`;
- В общем случае не должно возникать ситуаций, когда вам нужно вызвать Thread.Abort(), поскольку результат практически всегда - не предсказуем.
- CoreCLR более не содержит ручной вызов `Thread.Abort()`: он просто удален из класса. Но это не означает что его нет возможности получить.

### *ExecutionEngineException*

В комментарии к этому исключению стоит атрибут `Obsolete` с комментарием:

This type previously indicated an unspecified fatal error in the runtime. The runtime no longer raises this exception so this type is obsolete

И вообще-то это - неправда. Возможно, автору комментария очень бы хотелось чтобы это когда-либо стало правдой, однако чтобы продемонстрировать что это не так, достаточно вернуться к примеру исключения в `FirstChanceException`:

```
void Main()
{
    var counter = 0;

    AppDomain.CurrentDomain.FirstChanceException += (_, args) => {
        Console.WriteLine(args.Exception.Message);
        if(++counter == 1) {
            throw new ArgumentOutOfRangeException();
        }
    };

    throw new Exception("Hello!");
}
```

Результатом данного кода будет `ExecutionEngineException`, хотя ожидаемое мной поведение Unhandled Exception `ArgumentOutOfRangeException` из инструкции `throw new Exception("Hello!")`. Возможно это показалось страным разработчикам ядра и они посчитали что корректнее выбросить `ExecutionEngineException`.

Второй вполне простой путь получить `ExecutionEngineException` - это не корректно настроить маршаллинг в мир unsafe. Если вы напутаете с размерами типов, передадите больше чем надо, чем испортите, например, стек потока, ждите `ExecutionEngineException`. И это будет логичный, правильный результат: ведь в данной ситуации CLR вошла в состояние, которое она нашла не консистентным. Не понятным, как его восстанавливать. И как результат, `ExecutionEngineException`.

Отдельно стоит поговорить про диагностику `ExecutionEngineException`. Каковы могут быть причины его возникновения? Если исключение вдруг возникло в вашем коде, необходимо ответить на несколько вопросов:

- Используются ли в вашем приложении unsafe библиотеки? Вами или же может сторонними библиотеками. Попробуйте для начала выяснить, где конкретно приложение получает данную ошибку. Если код уходит в unsafe мир и получает `ExecutionEngineException` там, тогда необходимо тщательно

проверить сходимость сигнатур методов: в нашем коде и в импортируемом. Помните, что если импортируются модули написанные на Delphi и прочих вариациях языка Pascal, то аргументы должны идти в обратном порядке (настройка производится в `DllImport`: `CallingConvention.StdCall`);

- Подписаны ли вы на FirstChanceException? Возможно его код вызвал исключение. В таком случае просто оберните обработчик в `try-catch(Exception)` и обязательно сохраните в журнал ошибок происходящее;
- Может быть ваше приложение частично собрано под одну платформу, а частично - под другую. Попробуйте очистить кэш nuget пакетов, полностью пересобрать приложение с нуля: с очищенными вручную папками obj/bin
- Проблема иногда бывает в самом фреймворке. Например, в ранних версиях .NET Framework 4.0. В этом случае стоит протестировать отдельный участок кода, который вызывает ошибку - на более новой версии фреймворка;

В целом бояться этого исключения не стоит: оно возникает достаточно редко чтобы позабыть о нем до следующей радостной с ним встречи.

## Corrupted State Exceptions

После становления платформы и ее популяризации, после того как огромная масса програмистов начала мигрировать с C/C++ и MFC (Microsoft Foundation Classes) на более приятные мозгу среды разработки: сюда помимо .NET Framework можно отнести в первую очередь Qt, Java и C++ Builder, нам был дан вектор на виртуализацию исполнения кода приложения от окружающей среды. Со временем, удачно сверстанный архитектурно, .NET Framework начал занимать свою нишу. С годами, окрепнув и набирая массу, можно начинать рассуждать не с точки зрения приспособленца, а с точки зрения силы. И если раньше мы в большинстве случаев были вынуждены работать с феерическим пластом компонентов, написанных на COM/ATL/ActiveX (вы помните перетаскивание на формочки иконок COM/ActiveX компонент в Borland C++ Builder?), то теперь всем нам стало сильно легче жить. Ведь теперь соответствующие технологии *достаточно* редки чтобы волноваться и появилась возможность сделать их несколько неудобными чтобы от них начали

отказываться полностью, переходя на современный и ладный .NET Framework. Старые технологии, которые не то что существовали 5-10 лет назад, а на самом деле существуют и здравствуют и ныне, кажется нам чем-то архаичным, забытым, "кривым" и допотопным. А потому можно сделать еще один шаг к закрытию песочницы: сделать её ещё более непробиваемой, сделать её ещё более managed.

И один из этих шагов - введение понятия `Corrupted State Exceptions`, что по сути ставит ряд исключительных ситуаций вне закона. Давайте разберемся, что это за исключительные ситуации, а саму историю еще раз проследим на одной из них - `AccessViolationException`:

**Файл util.cpp** util.cpp

```
BOOL   IsProcessCorruptedStateException(DWORD   dwExceptionCode,   BOOL   fCheckForSO
/*=TRUE*/)
{
    // ...

    // If we have been asked not to include SO in the CSE check
    // and the code represent SO, then exit now.
    if ((fCheckForSO == FALSE) && (dwExceptionCode == STATUS_STACK_OVERFLOW))
    {
        return fIsCorruptedStateException;
    }

    switch(dwExceptionCode)
    {
        case STATUS_ACCESS_VIOLATION:
        case STATUS_STACK_OVERFLOW:
        case EXCEPTION_ILLEGAL_INSTRUCTION:
        case EXCEPTION_IN_PAGE_ERROR:
        case EXCEPTION_INVALID_DISPOSITION:
        case EXCEPTION_NONCONTINUABLE_EXCEPTION:
        case EXCEPTION_PRIV_INSTRUCTION:
        case STATUS_UNWIND_CONSOLIDATE:
            fIsCorruptedStateException = TRUE;
            break;
    }

    return fIsCorruptedStateException;
}
```

Рассмотрим описания наших исключительных ситуаций:

| Код ошибки | Описание |
| --- | --- |
| STATUS_ACCESS_VIOLATION | Достаточно частая ошибка попытки работы с участком памяти, на который нет прав. Память с точки зрения процесса линейная, но работать можно далеко не со всем диапазоном: только с теми "островками", которые были выделены операционной системой, а также с теми, на которые хватает прав (например, есть диапазоны, которыми владеет только операционная система или же можно только исполнять код но не читать как данные) |
| STATUS_STACK_OVERFLOW | Эта ошибка известна всем: ошибка нехватки памяти в стеке потока под вызов очередного метода |
| EXCEPTION_ILLEGAL_INSTRUCTION | Очередной код, считанный процессором из тела метода не был распознан как инструкция |
| EXCEPTION_IN_PAGE_ERROR | Поток предпринял попытку работы со страницей памяти, которой не существует |
| EXCEPTION_INVALID_DISPOSITION | Механизм обработки исключений вернул не правильный обработчик. Такое исключение никогда не должно возникать в программах, |

| Код ошибки | Описание |
| --- | --- |
| | написанных на высокоуровневых языках (например, C++) |
| EXCEPTION_NONCONTINUABLE_EXCEPTION | Поток сделал попытку продолжить исполнение программы после возникновения исключения, продолжить исполнение кода после которого не возможно. Тут имеется ввиду не блоки catch/fault/finally, а подобие фильтров исключений, которые позволяют исправить ошибку, которая привела к исключению и предпринять еще одну попытку выполнить код, приведший к ошибке |
| EXCEPTION_PRIV_INSTRUCTION | Попытка выполнить привилегированную инструкцию процессора |
| STATUS_UNWIND_CONSOLIDATE | Исключение, относящееся к размотке стека и не являющееся предметом наших обсуждений |

Прошу заметить, что по своей сути только два из них достойны перехвата: это `STATUS_ACCESS_VIOLATION` и `STATUS_STACK_OVERFLOW`. Остальные ошибки исключительны даже для исключительных ситуаций. Они скорее относятся к классу фатальных ошибок и нами рассматриваться не могут. А потому, давайте остановимся на этих двух более подробно:

### *AccessViolationException*

Получение этого исключения - одна из тех новостей, которые не хотелось бы никому получить. А когда получаешь, становится совсем не ясно что с этим делать. `AccessViolationException` - это исключение "промаха" мимо выделенного для приложения участка памяти и по своей сути выбрасывается при попытке чтения или записи в защищенную область памяти. Здесь под словом "защита" лучше понимать именно попытку работы с еще не выделенным участком памяти или же уже освобожденным. Тут, заметьте не имеется ввиду процесс выделения и освобождения памяти сборщиком мусора. Тот просто размечает уже выделенную память под свои и ваши нужды. Память - она имеет в некоторой степени слоистую структуру. Когда после слоя управления памятью сборщиком мусора идет слой управления выделением памяти библиотеками ядра CLR, а за ними - операционной системой - из пула доступных фрагментов линейного адресного пространства. Так вот когда приложение промахивается мимо своей памяти и пытается работать с невыделенным участком либо с участком, приложению не предназначенному, тогда и возникает это исключение. Когда оно возникает, вам доступно не так много вариантов для анализа:

- Если StackTrace уходит в недра CLR, вам сильно не повезло: это скорее всего ошибка ядра. Однако этот случай почти никогда не срабатывает. Из вариантов обхода - либо действовать как-то иначе либо обновить версию ядра если возможно;
- Если же StackTrace уходит в unsafe код некоторой библиотеки, тут доступны такие варианты: либо вы напутали с настройкой маршаллинга либо же в unsafe библиотеке закралась серьезная ошибка. Тщательно проверьте аргументы метода: возможно аргументы нативного метода имеют другую разрядность или же другой порядок или попросту размер. Проверьте что структуры передаются там где надо - по ссылке, а там, где надо - по значению

Чтобы перехватить такое исключение на данный момент необходимо показать JIT компилятору что это реально необходимо. В противном случае оно перехвачено

никак не будет и вы получите упавшее приложение. Однако, конечно же, его стоит перехватывать только тогда, когда вы понимаете что вы сможете его правильно обработать: его наличие может свидетельствовать о произошедшей утечке памяти если она была выделена unsafe методом между точкой его вызова и точкой выброса `AccessViolationException` и тогда хоть приложение и не будет "завалено", но его работа возможно станет не корректной: ведь перехватив поломку вызова метода вы наверняка попробуете вызвать этот метод еще раз, в будущем. А в этом случае что может пойти не так не известно никому: вы не можете знать каким образом было нарушено состояние приложения в прошлый раз. Однако, если желание перехватить такое исключение у вас сохранилось, прошу посмотреть на таблицу возможности перехвата этого исключения в различных версиях .NET Framework:

| **Версия .NET Framework** | **AccessViolationExeception** |
|---|---|
| 1.0 | NullReferenceException |
| 2.0, 3.5 | AccessViolation перехватить можно |
| 4.0+ | AccessViolation перехватить можно, но необходима настройка |
| .NET Core | AccessViolation перехватить *нельзя* |

Т.е. другими словами, если вам попалоось очень старое приложение на .NET Framework 1.0, ~~покажите его мне~~ вы получите NRE, что будет в некоторой степени обманом: вы отдали указатель со значением больше нуля, а получили NullReferenceException. Однако, на мой взгляд, такое поведение обосновано: находясь в мире управляемого кода вам меньше всего должно хотеться изучать типы ошибок неуправляемого кода и NRE - что по сути и есть "плохой указатель на объект" в мире .NET - тут вполне подходит. Однако, мир был бы прекрасен если бы

все так было просто. В реальных ситуациях пользователям решительно не хватало этого типа исключений и потому - достаточно скоро - его ввели в версии 2.0. Просуществовав несколько лет в перехватываемом варианте, исключение перестало быть перехватываемым, однако появилась специальная настройка, которая позволяет включить перехват. Такой порядок выбора в команде CLR в целом на каждом этапе выглядит достаточно обоснованным. Посудите сами:

- `1.0` Ошибка промаха мимо выделенных участков памяти должна быть именно исключительной ситуацией потому как если приложение работает с каким-либо адресом, оно его откуда-то получило. В managed мире этим местом является оператор `new`. В unmanaged мире - в целом любой участок кода может выступать точкой для возникновения такой ошибки. И хотя с точки зрения философии смысл обоих исключений диаметрально противоположен (NRE - работа с не проинициализированным указателем, AVE - работа с некорректно проинициализированным указателем), с точки зрения идеологии .NET некорректно проинициализированных указателей быть не может. Оба случая можно свести к одному и придать философский смысл: не корректно заданный указатель. А потому давайте так и сделаем: в обоих случаях будем выбрасывать `NullReferenceException`.
- `2.0` На ранних этапах существования .NET Framework оказалось что кода, который наследуется через COM библиотеки больше собственного: существует огромная кодовая база коммерческих компонент для взаимодействия с сетью, UI, БД и прочими подсистемами. А значит, вопрос получения именно `AccessViolationException` все-таки стоит: неверная диагностика проблем может сделать процесс поимки проблемы более дорогим. В .NET Framework введено исключение `AccessViolationException`.
- `4.0` .NET Framework укоренился, потеснив традиционную разработку на низкоуровневых языках программирования. Резко сокращено количество COM компонент: практически все основные задачи уже решаются в рамках самого фреймворка, а работа в unsafe кодом начинает считаться чем-то странным, неправильным. В этих условиях можно вернуться к идеологии,

введенной в фреймворк с самого начала: .NET - он только для .NET. Unsafe код - это не норма, а вынужденное состояние, а потому идеологичность наличия перехвата `AccessViolationException` идет вразрез с идеологией понятия фреймворк - как платформа (т.е. имитация полной песочницы со своими законами). Однако мы все еще находимся в реалиях платформы, на которой работаем и во многих ситуациях перехватывать это исключение все еще необходимо: вводим специальный режим перехвата: только если введена соответствующая конфигурация;

- `.NET Core` Наконец, сбылась мечта команды CLR: .NET более не предполагает законности работы с unsafe кодом, а потому существование `AccessViolationException` теперь вне закона даже на уровне конфигурации. .NET вырос настолько чтобы самостоятельно устанавливать правила. Теперь существование этого исключения в приложении приведет его к гибели, а потому любой unsafe код (т.е. сам CLR) обязан быть безопасным с точки зрения этого исключения. Если оно появляется в unsafe библиотеке, с ней просто не будут работать, а значит разработчики сторонних компонент на unsafe языках будут более аккуратными и обрабатывать его - у себя.

Вот так, на примере одного исключения можно проследить историю становления .NET Framework как платформы: от неуверенного подчинения внешним правилам до самостоятельного установления правил самой платформой.

После всего сказанного осталось раскрыть последнюю тему: как включить обработку данного исключения в `4.0+`. Итак, чтобы включить обработку исключения данного типа в конкретном методе, необходимо:

- Добавить в секцию `configuration/runtime` следующий код: `<legacyCorruptedStateExceptionsPolicy enabled="true|false"/>`
- Для каждого метода, где *необходимо* обработать `AccessViolationException`, надо добавить два атрибута: `HandleProcessCorruptedStateExceptions` и `SecurityCritical`. Эти

атрибуты позволяют включить обработку Corrupted State Exceptions, для *конкретных* методов, а не для всех вообще. Эта схема очень правильная, поскольку вы должны быть точно уверены что хотите их обрабатывать и знать, где: иногда более правильный вариант - просто завалить приложение на бок.

Для примера включения обработчика `CSE` и их примитивной обработки рассмотрим следующий код:

```
[HandleProcessCorruptedStateExceptions, SecurityCritical]
public bool TryCallNativeApi()
{
    try
    {
        // Запуск метода, который может выбросить AccessViolationException
    }
    catch (Exception e)
    {
        // Журналирование, выход
        System.Console.WriteLine(e.Message);
        return false;
    }

  return true;
}
```

### *StackOverflowException*

Последний тип исключений, о котором стоит поговорить - это ошибка переполнения стека. Она возникает тогда, когда по сути в массиве, выделенном под стек кончается память. Само строение стека мы подробно обсудили в соответствующей главе (Стек потока), а здесь без сильного углубления остановимся на самой ошибке.

Итак, когда наступает нехватка памяти под стек потока (либо уперлись в следующий занятый участок памяти и нет возможности выделить следующую страницу виртуальной памяти) или же поток уперся в разрешенный диапазон памяти, происходит попытка доступа к адресному пространству, которое называется Guard page. По сути этот диапазон - ловушка и не занимает никакой физической памяти. Вместо реального чтения или записи процессор вызывает специализированное прерывание, которое должно запросить у операционной системы новый участок

памяти под рост стека потока. В случае достижения максимально-разрешенных значений операционная система вместо выделения нового участка генерирует исключительную ситуацию с кодом `STATUS_STACK_OVERFLOW`, которая будучи проброшенной через `Structured Exception Handling` механизм в .NET обрушивает текущий поток исполнения как более не корректный.

Прошу заметить что хоть данное исключение и является `Corrupted State Exception`, перехватить его при помощи `HandleProcessCorruptedStateExceptions` не представляется возможным. Т.е. следующий код не отработает:

```
// Main.cs
[HandleProcessCorruptedStateExceptions, SecurityCritical]
static void Main()
{
    try
    {
        Recursive();
    } catch (Exception exception)
    {
        Console.WriteLine("Catched Stack Overflow!");
    }
}

static void Recursive()
{
    Recursive();
}

// app.config:
<configuration>
  <runtime>
    <legacyCorruptedStateExceptionsPolicy enabled="true"/>
  </runtime>
</configuration>
```

А не представляется возможным потому что переполнение стека может быть вызвано двумя путями: первый - это намеренный вызов рекурсивного метода, который не слишком аккуратен чтобы контролировать собственную глубину. Тут может возникнуть желание исправить ситуацию, перехватив выброс исключения. Однако, если подумать, мы тем самым легализуем данную ситуацию, позволяя ей случиться вновь, что выглядит скорее не дальновидным чем случаем проявления заботы. И вторая - случайность - получение StackOverflowException при вполне себе обычном вызове. Просто в данный момент глубина стека вызовов оказалась слишком критичной. В данном примере перехватывать исключение выглядит как что-то совсем дикое: приложение работало штатно, все шло хорошо как вдруг

легальный вызов метода при корректно работающих алгоритмах вызвал выброс исключения с дальнейшей разверткой стека до ожидающего такого поведения участка кода. Хм... Еще раз: мы ждем, что на следующем участке ничего не отработает потому что в стеке кончится память. Как по мне, это полный абсурд

# Основы управления памятью [В процессе]

Если статья находится в процессе, это значит, что прямо в эти минуты я ее правлю и возможно, часть информации не является достоверной

## Обзор

Когда вы думаете о разработке любого .NET приложения до недавних пор можно было себе позволить считать, что приложение, которое вы делаете будет всегда работать на одной и той же платформе: это операционная система Windows, запущенная поверх технологического стека Intel. Сейчас же с каждым прожитым днем мы входим в новую эпоху: платформа .NET стала поистине кроссплатформенной, пустив новые корни в сторону всех доступных настольных операционных систем. Это - прекрасное время и наш долг сейчас не потерять нить и остаться востребованными специалистами. Ведь когда toolset становится кроссплатформенным это означает что мы обязаны начать смотреть внутрь. Изучать, как работает двигатель нашей платформы. Чтобы понимать, почему тот ведет себя так или иначе на различных системах.

Подсистему управления памятью мы будем изучать по слоям. Начнем от слоя, близкого к пониманию ее работы "на пальцах" и закончим - слоем архитектуры на самом низком уровне - процессорном. Ведь чтобы до конца понимать всю проблематику работы с памятью - надо знать все, начиная от процессорных кэшей заканчивая оптимизациями работы в кучами .NET.

# Основы основ

Если взять любое приложение и попробовать грубо разделить его на две части, то получится, что любое приложение состоит фактически из двух самых важных вещей: кода, который исполняется процессором, и данных, которыми этот код оперирует в своей работе. При чем если с кодом все более-менее ясно, то данные можно поделить на несколько больших секций:

- **Thread stack** - это область памяти, которая есть у любого потока и через которую работают все вызовы всех методов, плюс там же организовано хранилище для локальных переменных методов;
- **Code Heap** - это область памяти, куда JITter складывает результаты компиляции MSIL;

- **Small Objects Heap** - это куча маленьких объектов. Как бы это не звучало, именно так это и называется. По своей сути это - хранилище объектов, размер которых не превышает 85К байт;
- **Large Objects Heap** - это куча больших объектов. Сюда попадают объекты, размеры которых превышают 85К байт;
- **TypeRefs Heap** - куча Type References - описателей типов .NET - со стороны подсистемы CLR (со стороны .NET типов выступает подсистема Reflection)
- **MethodRefs Heap** - куча Methods References - описателей методов .NET - со стороны подсистемы CLR (со стороны .NET типов выступает подсистема Reflection)
- И многие другие

## Базовая структура, платформа x86

Существует область памяти, про которую редко заходит разговор. Однако эта область является, возможно, основной в работе приложения. Самой часто используемой, достаточно ограниченной с моментальным выделением и освобождением памяти. Область эта называется "стек потока". Причем поскольку указатель на него кодируется по своей сути регистрами процессора, которые входят в контекст потока, то в рамках исполнения любого потока стек потока свой. Зачем он необходим?

Итак, разберем элементарный пример кода:

```
void Method1()
{
    Method2(123);
}

void Method2(int arg)
{
    // ...
}
```

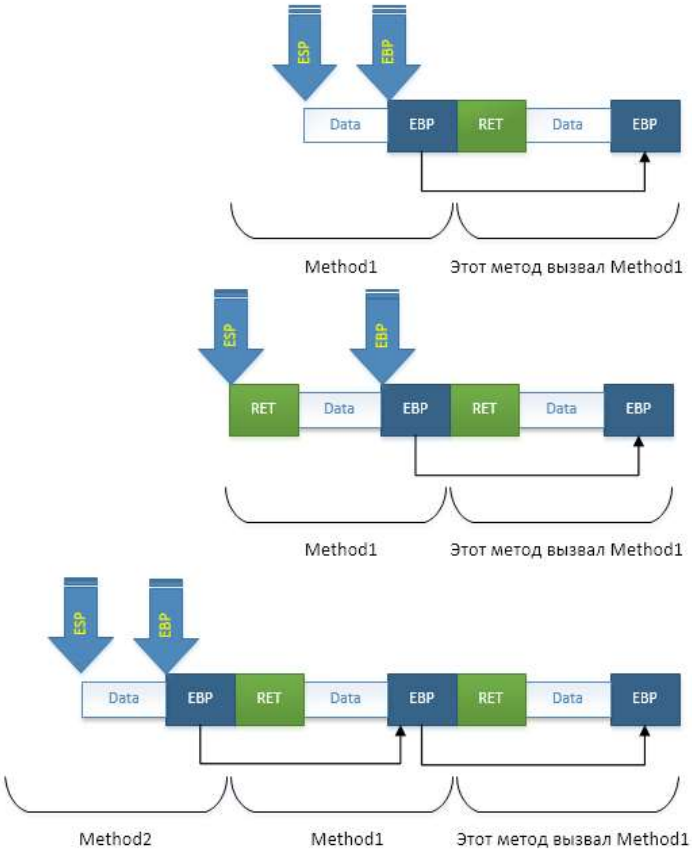
В данном коде не происходит ничего примечательного, однако не будем его пропускать, а наоборот: посмотрим на него максимально внимательно. Когда любой `Method1` вызывает любой `Method2`, то абсолютно любой такой вызов (и не только в .NET, но и в других платформах) осуществляет следующие операции:

1. Первое, что делает код, скомпилированный JIT'ом: он сохраняет параметры метода в стек (начиная с третьего). При этом первые два передаются через регистры. Тут важно помнить, что первым параметром экземплярных методов передается указатель на тот объект, с которым работает метод. Т.е. указатель `this`. Так что в этих (почти всех) случаях для регистров остается всего один параметр, а для всех остальных - стек;
2. Далее компилятор ставит инструкцию вызова метода `call`, которая помещает в стек адрес возврата из метода: адрес следующей за `call` инструкцией.

Таким образом любой метод знает, куда ему необходимо вернуться, чтобы вызывающий код смог продолжить работу;

3. После того как все параметры переданы, а метод вызван, нам надо как-то понять, как стек восстановить в случае выхода из метода, если мы не хотим заботиться о подсчете занимаемых нами в стеке байтов. Для этого мы сохраняем значение регистра EBP, который всегда хранит указатель на начало текущего кадра стека (т.е. участка, где хранится информация для конкретного вызванного метода). Сохраняя при каждом вызове значение этого регистра, мы тем самым фактически создаем односвязный список стековых кадров. Но прошу заметить, что по факту они идут чётко друг за другом, без каких-либо пробелов. Однако для упрощения освобождения памяти из-под кадра и для отладки приложения (отладчик использует эти указатели чтобы отобразить Stack Trace) строится односвязный список;
4. Последнее, что надо сделать при вызове метода, - выделить участок памяти под локальные переменные. Поскольку компилятор заранее знает сколько ее понадобится, то делает он это сразу, сдвигая указатель на вершину стека (SP/ESP/RSP) на необходимое количество байт;
5. И наконец, на пятом этапе выполняется код метода, полезные операции;
6. Когда происходит выход из метода, то вершина стека восстанавливается из EBP - места, где хранится начало текущего стекового кадра;
7. Далее, последним этапом осуществляется выход из метода через инструкцию `ret`. Она забирает со стека адрес возврата, заботливо оставленный ранее инструкцией `call` и делает `jmp` по этому адресу.

Те же самые процессы можно посмотреть на изображении:




Также замечу, что стек "растет", начиная со старших адресов и заканчивая младшими, т.е. в обратную сторону.

Глядя на все это, невольно приходишь к выводу, что если не большинство, то минимум половина всех операций, которыми занимается процессор - это обслуживание структуры программы, а не ее полезной нагрузки. Т.е. обслуживание вызовов методов, проверки типов на возможность привести один к другому, компиляцию Generic вариаций, поиск методов в таблицах интерфейсов... Особенно если мы вспомним, что большинство современного кода написано с подходом работы через интерфейсы, разбивку на множество пусть маленьких, но выполняющих каждый - свое - методов.. А работа при этом часто идет с базовыми типами и приведением типов то к интерфейсу, то к наследнику. При всех таких входящих условиях вывод о расточительности инфраструктурного кода вполне может назреть. Единственное, что я могу вам на это все сказать: компиляторы, в том числе и JIT, обладают множеством техник, позволяющим им делать более

продуктивный код. Где можно - вместо вызова метода вставляется его тело целиком, а где возможно вместо поиска метода в VSD интерфейса осуществляется его прямой вызов. Что самое грустное, инфраструктурную нагрузку очень сложно замерить: надо чтобы JITter либо какой-либо компилятор вставлял бы какие-то метрики до и после мест работы инфраструктурного кода. Т.е. до вызова метода, а внутри метода - после инициализации кадра стека. До выхода из метода, после выхода из метода. До компиляции, после компиляции. И так далее. Однако, давайте не будем о грустном, а поговорим лучше о том, что мы можем с вами сделать с полученной информацией.

## Немного про исключения на платформе x86

Если мы посмотрим внутрь кода методов, то мы заметим еще одну структуру, работающую со стеком потока. Посудите сами:

```
void Method1()
{
    try
    {
        Method2(123);
    } catch {
        // ...
    }
}

void Method2(int arg)
{
    Method3();
}

void Method3()
{
    try
    {
        //...
    } catch {
        //...
    }
}
```

Если исключение возникнет в любом из методов, вызванных из `Method3`, то управление будет возвращено в блок `catch` метода `Method3`. При этом если исключение обработано не будет, то его обработка начнется в методе `Method1`. Однако если ничего не случится, то `Method3` завершит свою работу, управление

перейдет в метод `Method2`, где также может возникнуть исключение. Однако по естественным причинам обработано оно будет не в `Method3`, а в `Method1`. Вопрос такого удобного автоматизма заключается в том, что структуры данных, образующие цепочки обработчиков исключений, также находятся в стековом кадре метода, где они объявлены. Про сами исключения мы поговорим отдельно, а здесь скажу только, что модель исключений в .NET Framework CLR и в Core CLR отличается. CoreCLR вынуждена быть разной на разных платформах, а потому модель исключений там другая и представляется в зависимости от платформы через прослойку PAL (Platform Adaption Layer) различными имплементациями. Большому .NET Framework CLR это не нужно: он живет в экосистеме платформы Windows, в которой есть уже много лет общеизвестный механизм обработки исключений, который называется SEH (Structured Exception Handling). Этот механизм используется практически всеми языками программирования (при конечной компиляции), потому что обеспечивает сквозную обработку исключений между модулями, написанными на различных языках программирования. Работает это примерно так:

1. При вхождении в блок try на стек кладется структура, которая первым полем указывает на предыдущий блок обработки исключений (например, вызывающий метод, у которого также есть try-catch), тип блока, код исключения и адрес обработчика;
2. В TEB потока (Thread Environment Block, по сути - контекст потока) меняется адрес текущей вершины цепочки обработчиков исключений на тот, что мы создали. Таким образом мы добавили в цепочку наш блок.
3. Когда try закончился, производится обратная операция: в TEB записывается старая вершина, снимая таким образом наш обработчик из цепочки;
4. Если возникает исключение, то из TEB забирается вершина и по очереди по цепочке вызываются обработчики, которые проверяют, подходит ли исключение конкретно им. Если да, выполняется блок обработки (например, catch).
5. В TEB восстанавливается тот адрес структуры SEH, который находится в стеке ДО метода, обработавшего исключение.

Как видите, совсем не сложно. Однако вся эта информация также находится в стеке.

## Совсем немного про несовершенство стека потока

Давайте немного подумаем о вопросе безопасности и возможных проблемах, которые чисто теоретически могут возникнуть. Для этого давайте еще раз глянем на структуру стека потока, которая по своей сути - обычный массив. Диапазон памяти, в котором строятся фреймы, организован так, что он растет с конца в начало. Т.е. более поздние фреймы располагаются по более ранним адресам. Также, как уже было сказано, фреймы связаны односвязным списком. Это сделано потому, что размер фрейма не является фиксированным и должен быть "считан" любым отладчиком. Процессор при этом не разграничивает фреймы между собой: любой метод по своему желанию может считать всю область памяти целиком. А если учесть при этом, что мы находимся в виртуальной памяти, которая поделена на участки, являющиеся реально выделенной памятью, то можно при помощи специальной функции WinAPI по любому адресу со стека получить диапазон выделенной памяти, в которой этот адрес находится. Ну а разобрать односвязный список - дело техники:

```
    // переменная находится в стеке
    int x;

    // Забрать информацию об участке памяти, выделенной под стек
    MEMORY_BASIC_INFORMATION *stackData = new MEMORY_BASIC_INFORMATION();
    VirtualQuery((void *)&x, stackData, sizeof(MEMORY_BASIC_INFORMATION));
```

Это дает нам возможность получить и модифицировать все данные, которые находятся в качестве локальных переменных у методов, которые нас вызвали. Если приложение никак не настраивает песочницу, в рамках которой вызываются сторонние библиотеки, расширяющие функционал приложения, то сторонняя библиотека сможет утащить данные даже если тот API, который вы ей отдаете, этого не предполагает. Методика эта может показаться вам надуманной, однако в мире C/C++, где нет такой прекрасной вещи как AppDomain с настроенными правами атака по стеку - это самое типичное, что только можно встретить из взлома приложений. Мало того, можно через рефлексию посмотреть на тип, который нам необходим, повторить его структуру у себя, и, пройдя по ссылке со стека на объект,

заменить адрес VMT на наш, перенаправив таким образом всю работу с конкретным экземпляром к нам. SEH, кстати говоря, также вовсю используется для взлома приложений. Через него вы также можете, меняя адрес обработчика исключения, заставлять ОС выполнить вредоносный код. Но вывод из всего этого очень простой: всегда настраивайте песочницу, когда хотите работать с библиотеками, расширяющими функционал вашего приложения. Я, конечно же, имею ввиду всяческие плагины, аддоны и прочие расширения.

## Большой пример: клонирование потока на платформе x86

Чтобы запомнить все, что мы прочитали до мельчайших подробностей, надо зайти к вопросу освещения какой-либо темы с нескольких сторон. Казалось бы, какой пример можно построить для стека потока? Вызвать метод из другого? Магия... Конечно же нет: это мы делаем ежедневно по много раз. Вместо этого мы склонируем поток исполнения. Т.е. сделаем так, чтобы после вызова определенного метода у нас вместо одного потока оказалось бы два: наш и новый, но продолжающий выполнять код с точки вызова метода клонирования так, как будто он сам туда дошел. А выглядеть это будет так:

```
void MakeFork()
{
    // Для уверенности что все склонировалось мы делаем локальные переменные:
    // В новом потоке их значения обязаны быть такими же как и в родительском
    var sameLocalVariable = 123;
    var sync = new object();

    // Замеряем время
    var stopwatch = Stopwatch.StartNew();

    // Клонируем поток
    var forked = Fork.CloneThread();

    // С этой точки код исполняется двумя потоками.
    // forked = true для дочернего потока, false для родительского
    lock(sync)
    {
        Console.WriteLine("in {0} thread: {1}, local value: {2}, time to enter = {3}
ms",
            forked ? "forked" : "parent",
            Thread.CurrentThread.ManagedThreadId,
            sameLocalVariable,
            stopwatch.ElapsedMilliseconds);
```

```
    }

    // При выходе из метода родительский вернет управления в метод,
    // который вызвал MakeFork(), т.е. продолжит работу как ни в чем ни бывало,
    // а дочерний завершит исполнение.
}

// Примерный вывод:
// in forked thread: 2, local value: 123, time to enter = 2 ms
// in parent thread: 1, local value: 123, time to enter = 2 ms
```

Согласитесь, концепт интересный. Конечно же, тут можно много спорить про целесообразность таких действий, но задача этого примера - поставить жирную точку в понимании работы этой структуры данных. Как же сделать клонирование? Для ответа на данный вопрос надо ответить на другой вопрос: что вообще определяет поток? А поток определяют следующие структуры и области данных:

- Набор регистров процессора. Все регистры определяют состояние потока исполнения инструкций: от адреса текущей инструкции исполнения до адресов стека потока и данных, которыми он оперирует;
- Thread Environment Block или TIB/TEB, который хранит системную информацию по потоку, включая адреса обработчиков исключений;
- Стек потока, адрес которого определяется регистрами SS:ESP;
- Платформенный контекст потока, который содержит локальные для потока данные (ссылка идет из TIB)

Ну и наверняка что-то еще, о чем мы можем не знать. Да и знать нам всего для примера нет никакой надобности: в промышленное использование данный код не пойдет, а скорее будет служить нам отличным примером, который поможет разобраться в теме. А потому он не будет учитывать всего, а только самое основное. А для того чтобы он заработал в базовом виде, нам понадобится скопировать в новый поток набор регистров (исправив SS:ESP, т.к. стек будет новым), а также подредактировать сам стек, чтобы он содержал ровно то что нам надо.

Итак. Если стек потока определяет по сути, какие методы были вызваны и какими данными они оперируют, то получается что по-идее, меняя эти структуры, можно поменять как локальные переменные методов, так и вырезать из стека вызов какого-то метода, поменять метод на другой или же добавить в любое место

цепочки свой. Хорошо, с этим определились. Теперь давайте посмотрим на некий псевдокод:

```
void RootMethod()
{
    MakeFork();
}
```

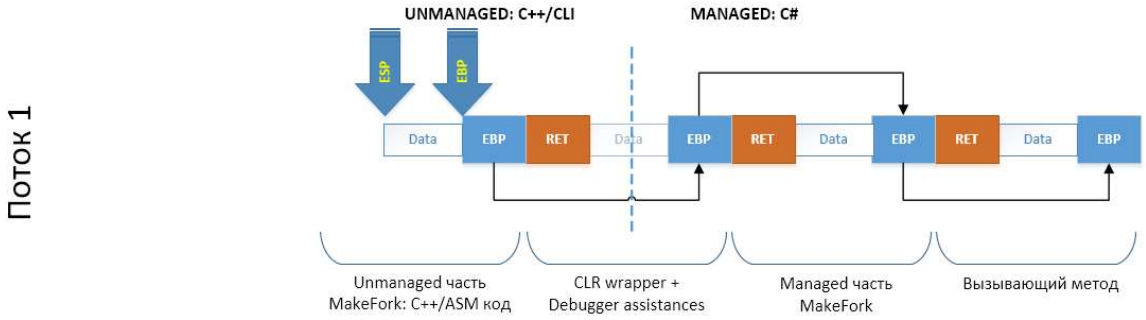
Когда вызовется MakeFork(), что мы ожидаем с точки зрения стек трейсов? Что в родительском потоке все останется без изменений, а дочерний будет взят из пула потоков (для скорости), в нем будет сымитирован вызов метода `MakeFork` вместе с его локальными переменными, а код продолжит выполнение не с начала метода, а с точки, следующей после вызова `CloneThread`. Т.е. стек трейс в наших фантазиях будет выглядеть примерно так:

```
// Parent Thread
RootMethod -> MakeFork

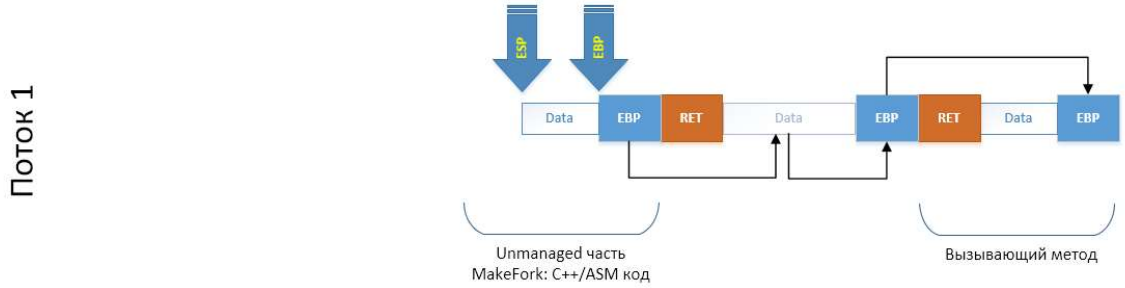
// Child Thread
ThreadPool -> MakeFork
```

Что у нас есть изначально? Есть наш поток. Также есть возможность создать новый поток либо запланировать задачу в пул потоков, выполнив там свой код. Также мы понимаем, что информация по вложенным вызовам хранится в стеке вызовов и что при желании мы можем ею манипулировать (например, используя C++/CLI). Причем, если следовать соглашениям и вписать в верхушку стека адрес возврата для инструкции ret, значение регистра EBP и выделить место под локальные (если необходимо), то можно имитировать вызов метода. Ручную запись в стек потока возможно сделать из C#, однако нам понадобятся регистры и их очень аккуратное использование, а потому без ухода в C++ нам не обойтись. Тут к нам на помощь впервые в жизни (лично у меня) приходит CLI/C++, который позволяет писать смешанный код: часть инструкций - на .NET, часть - на C++, а иногда даже уходить на уровень ассемблера. Именно то, что нам надо.

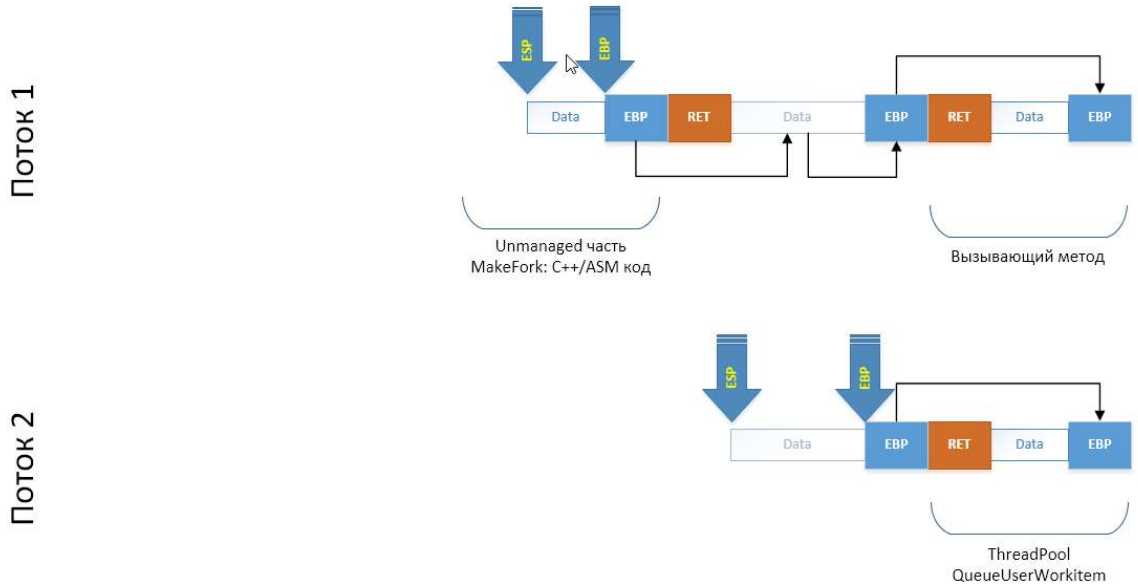
Итак, как будет выглядеть стек потока, когда наш код вызовет MakeFork, который вызовет CloneThread, который уйдет в unmanaged мир CLI/C++ и вызовет метод клонирование (саму реализацию) - там? Давайте посмотрим на схему (еще раз напомню, что стек растет от старших адресов к младшим. Справа налево):




Ну а для того чтобы не тащить всю простыню со схемы на схему, упростим, отбросив то, что нам не нужно:

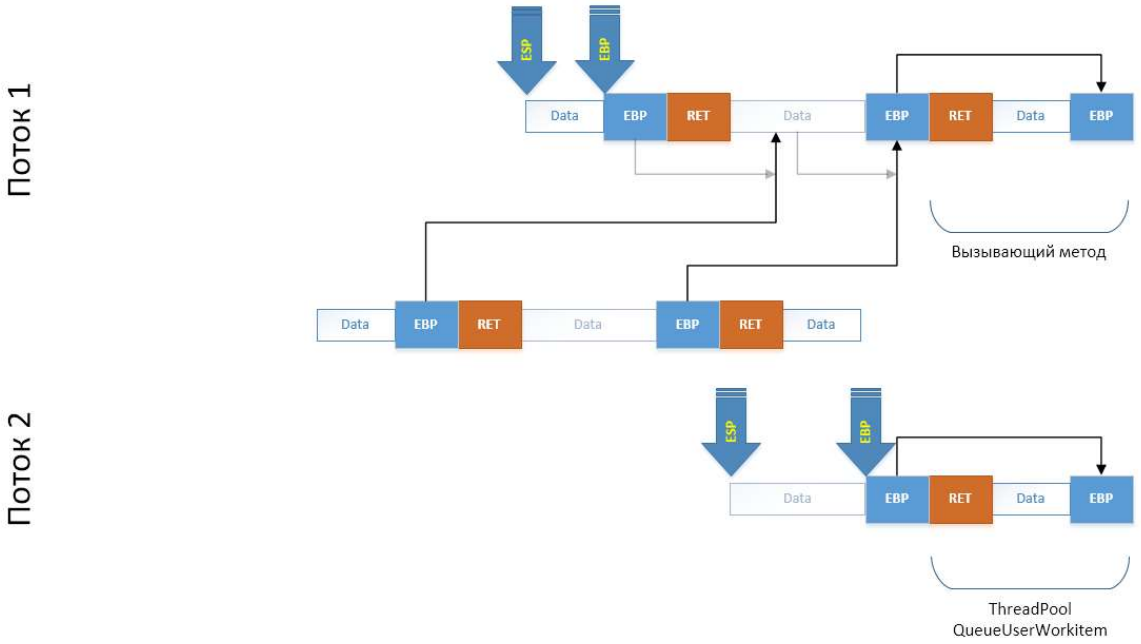



Когда мы создадим поток либо возьмем готовый из пула потоков, в нашей схеме появляется еще один стек, пока еще ничем не проинициализированный:

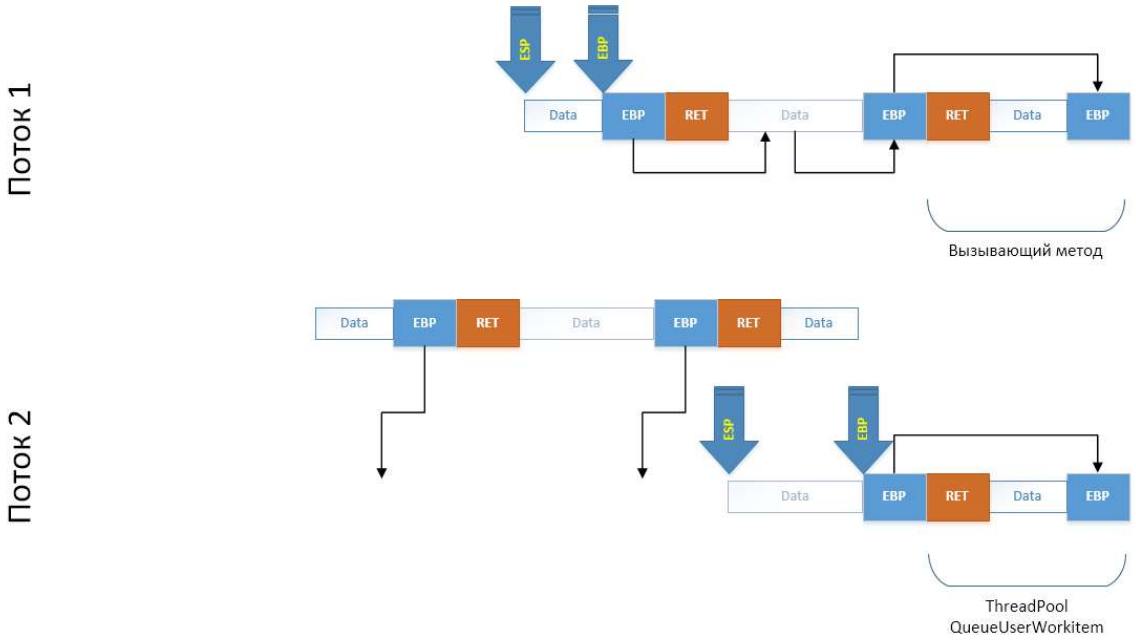



Теперь наша задача - сымитировать запуск метода `Fork.CloneThread()` в новом потоке. Для этого мы должны в конец его стека потока дописать серию кадров: как будто из делегата, переданного ThreadPool'у был вызван `Fork.CloneThread()`, из

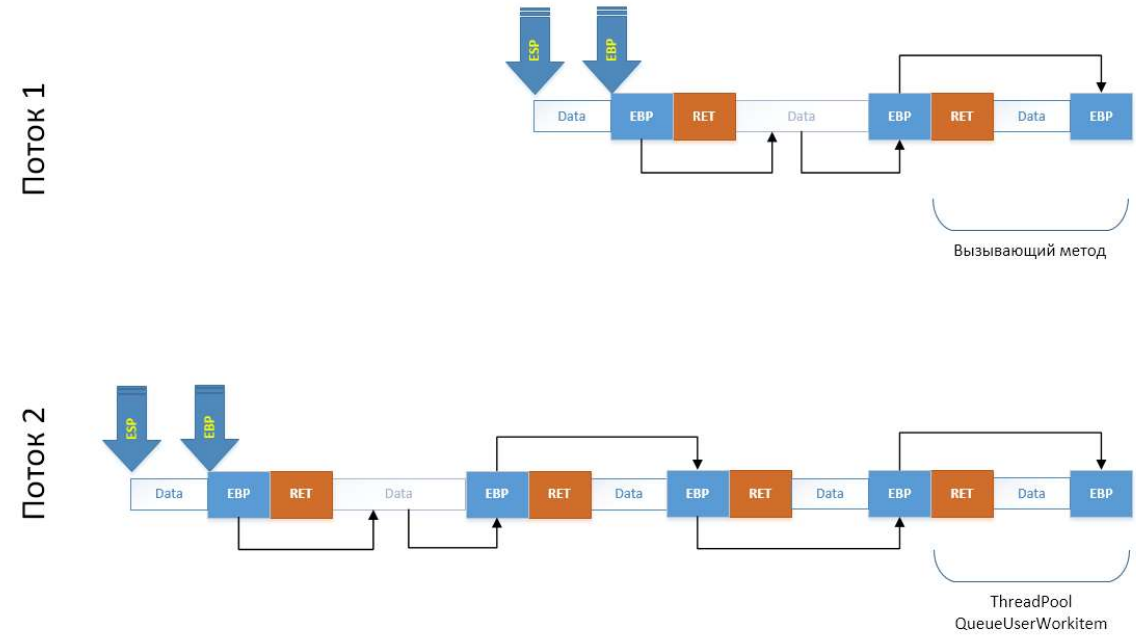
которого через враппер C++ кода managed оберткой был вызван CLI/C++ метод. Для этого мы просто скопируем необходимый участок стека в массив (замечу, что со склонированного участка на старый "смотрят" копии регистров EBP, обеспечивающих построение цепочки кадров):




Далее чтобы обеспечить целостность стека после операции копирования склонированного на предыдущем шаге участка мы заранее рассчитываем, по каким адресам будут находиться поля EBP на новом месте, и сразу же исправляем их, прямо на копии:




Последним шагом, очень аккуратно, задействуя минимальное количество регистров, копируем наш массив в конец стека дочернего потока, после чего сдвигаем регистры ESP и EBP на новые места. С точки зрения стека мы сымитировали вызов всех этих методов:




Но пока не с точки зрения кода. С точки зрения кода нам надо попасть в те методы, которые только что создали. Самое простое - просто сымитировать выход из метода:

восстановить ESP до EBP, в EBP положить то, на что он указывает и вызвать инструкцию ret, инициировав выход из якобы вызванного C++ метода клонирования потока, что приведет к возврату в реальный wrapper CLI/C++ вызова, который вернет управление в MakeFork(), но в дочернем потоке. Техника сработала.

Теперь давайте взглянем на код. Первое что мы сделаем - это возможность для CLI/C++ кода создать .NET поток. Для этого мы его должны создать в .NET:

```
extern "C" __declspec(dllexport)
void __stdcall MakeManagedThread(AdvancedThreading_Unmanaged *helper, StackInfo
*stackCopy)
{
    AdvancedThreading::Fork::MakeThread(helper, stackCopy);
}
```

На типы параметров пока не обращайте внимания. Они нужны для передачи информации о том, какой участок чтека необходимо у себя рисовать из родительского потока в дочерний. Метод создания потока оборачивает в делегает вызов unmanaged метода, передает данные и ставит делегат в очередь на обработку пулом потоков.

```
[MethodImpl(MethodImplOptions::NoInlining | MethodImplOptions::NoOptimization |
MethodImplOptions::PreserveSig)]
static void MakeThread(AdvancedThreading_Unmanaged *helper, StackInfo *stackCopy)
{
    ForkData^ data = gcnew ForkData();
    data->helper = helper;
    data->info = stackCopy;

    ThreadPool::QueueUserWorkItem(gcnew WaitCallback(&InForkedThread), data);
}

[MethodImpl(MethodImplOptions::NoInlining | MethodImplOptions::NoOptimization |
MethodImplOptions::PreserveSig)]
static void InForkedThread(Object^ state)
{
    ForkData^ data = (ForkData^)state;
    data->helper->InForkedThread(data->info);
}
```

И, наконец, сам метод клонирования (вернее, его .NET часть):

```
[MethodImpl(MethodImplOptions::NoInlining | MethodImplOptions::NoOptimization |
MethodImplOptions::PreserveSig)]
static bool CloneThread()
{
    ManualResetEvent^ resetEvent = gcnew ManualResetEvent(false);
    AdvancedThreading_Unmanaged *helper = new AdvancedThreading_Unmanaged();
    int somevalue;

    // *
    helper->stacktop = (int)(int *)&somevalue;
```

```
    int forked = helper->ForkImpl();
    if (!forked)
    {
        resetEvent->WaitOne();
    }
    else
    {
        resetEvent->Set();
    }
    return forked;
}
```

Чтобы понимать, где в цепочке кадров стека находится данный метод, мы сохраняем себе адрес стековой переменной (*). Использовать этот адрес мы будем в методе клонирования, речь о котором пойдет чуть ниже. Также, чтобы вы понимали, о чем идет речь, приведу код структуры, необходимой для хранения информации о копии стека:

```
public class StackInfo
{
public:
    // Копия значений регистров
    int EAX, EBX, ECX, EDX;
    int EDI, ESI;
    int ESP;
    int EBP;
    int EIP;
    short CS;

    // Адрес копии стека
    void *frame;

    // Размер копии
    int size;

    // Диапазоны адресов оригинального стека нужны,
    // чтобы поправить адреса на стеке если они есть на новые
    int origStackStart, origStackSize;
};
```

Работа же самого алгоритма разделена на две части: в родительском потоке мы подготавливаем данные для того, чтобы в дочернем потоке отрисовать нужные кадры стека. Вторым же этапом восстанавливаются данные в дочернем потоке, накладываясь на свой собственный стек потока исполнения, имитируя таким образом вызовы методов, которые в реальности вызваны не были.

## Метод подготовки к копированию

Описание кода я буду делать блоками. Т.е. единый код будет разбит на части, и каждая из частей будет отдельно прокомментирована. Итак, приступим. Когда

внешний код вызывает `Fork.CloneThread()`, то через внутреннюю обертку над неуправляемым кодом и через ряд дополнительных методов если код работает под отладкой (так называемые debugger assistants). Именно поэтому мы в .NET части запомнили адрес переменной в стеке: для C++ метода этот адрес является своеобразной меткой: теперь мы точно знаем, какой участок стека мы можем спокойно копировать.

```
int AdvancedThreading_Unmanaged::ForkImpl()
{
    StackInfo copy;
    StackInfo* info;
```

Первым делом, до того, как произойдет хоть какая-то операция, чтобы не получить запорченные регистры, мы их копируем локально. Также дополнительно необходимо сохранить адрес кода, куда будет сделан `goto`, когда в дочернем потоке стек будет сымитирован, и необходимо будет произвести процеруду выхода из `CloneThread` из дочернего потока. В качестве "точки выхода" мы выбираем `JmpPointOnMethodsChainCallEmulation` и не просто так: после операции сохранения этого адреса "на будущее" мы дополнительно закладываем в стек число 0.

```
    // Save ALL registers
    _asm
    {
        mov copy.EAX, EAX
        mov copy.EBX, EBX
        mov copy.ECX, ECX
        mov copy.EDX, EBX
        mov copy.EDI, EDI
        mov copy.ESI, ESI
        mov copy.EBP, EBP
        mov copy.ESP, ESP

        // Save CS:EIP for far jmp
        mov copy.CS, CS
        mov copy.EIP, offset JmpPointOnMethodsChainCallEmulation

        // Save mark for this method, from what place it was called
        push 0
    }
```

После чего, после `JmpPointOnMethodsChainCallEmulation` мы достаем это число из стека и проверяем: там лежит `0`? Если да, мы находимся в том же самом потоке: а значит у нас еще много дел, и мы переходим на `NonClonned`. Если же там не `0`, а по факту `1`, это значит, что дочерний поток закончил "дорисовку" стека потока до необходимого

состояния, положил на стек число `1` и сделал goto в эту точку (замечу, что goto он делает из другого метода). А это значит, что настало время для выхода из `CloneThread` в дочернем потоке, вызов которого был сымитирован.

```
JmpPointOnMethodsChainCallEmulation:

    _asm
    {
        pop EAX
        cmp EAX, 0
        je NonClonned

        pop EBP
        mov EAX, 1
        ret
    }
NonClonned:
```

Хорошо, мы убедились, что мы все еще мы, а значит надо подготовить данные для дочернего потока. Чтобы более не спускаться на уровень ассемблера, работать мы будем со структурой ранее сохраненных регистров. Достанем из нее значение регистра EBP: он по сути является полем "Next" в односвязном списке кадров стека. Перейдя по адресу, который там содержится, мы очутимся в кадре метода, который нас вызвал. Если и там возьмем первое поле и перейдем по тому адресу, то окажемся в еще более раннем кадре. Так мы сможем дойти до managed части `CloneThread`: ведь мы сохранили адрес переменной в ее стековом кадре, а значит прекрасно знаем, где остановиться. Этой задачей и занимается цикл, приведенный ниже.

```
    int *curptr = (int *)copy.EBP;
    int frames = 0;

    //
    //  Calculate frames count between current call and Fork.CloneTherad() call
    //
    while ((int)curptr < stacktop)
    {
        curptr = (int*)*curptr;
        frames++;
    }
```

Получив адрес начала кадра managed метода `CloneThread`, мы теперь знаем, сколько надо копировать для имитации вызова `CloneThread` из `MakeFork`. Однако поскольку нам `MakeFork` также нужен (наша задача выйти именно в него), то мы делаем дополнительно еще один переход по односвязносу списку: `*(int *)curptr`. После чего создаем массив под сохранение стека и сохраняем его простым копированием.

```
    //
    //  We need to copy stack part from our method to user code method including its
locals in stack
    //
    int localsStart = copy.EBP;                                 // our EBP points to EBP
value for parent method + saved ESI, EDI
    int localsEnd = *(int *)curptr;                            // points to end of user's
method's locals (additional leave)

    byte *arr = new byte[localsEnd - localsStart];
    memcpy(arr, (void*)localsStart, localsEnd - localsStart);
```

Еще одна задача, которую надо будет решить, - это исправление адресов переменных, которые попали на стек и при этом указывающих на стек. Для решения этой проблемы мы получаем диапазон адресов, которые нам выделила операционная система под стек потока. Сохраняем полученную информацию и запукаем вторую часть процесса клонирования, запланировав делегат в пул потоков:

```
    // Get information about stack pages
    MEMORY_BASIC_INFORMATION *stackData = new MEMORY_BASIC_INFORMATION();
    VirtualQuery((void *)copy.EBP, stackData, sizeof(MEMORY_BASIC_INFORMATION));

    // fill StackInfo structure
    info = new StackInfo(copy);
    info->origStackStart = (int)stackData->BaseAddress;
    info->origStackSize = (int)stackData->RegionSize;
    info->frame = arr;
    info->size = (localsEnd - localsStart);

    // call managed ThreadPool.QueueUserWorkitem to make fork
    MakeManagedThread(this, info);

    return 0;
}
```

## Метод восстановления из копии

Этот метод вызывается как результат работы предыдущего: нам переданы копия участка стека родительского потока, а также полный набор его регистров. Наша задача в нашем потоке, взятом из пула потоков, дорисовать все вызовы, скопированные из родительского потока таким образом, как будто мы сами их осуществили. Завершив работу, MakeFork дочернего потока попадет обратно в этот метод, который, завершив работу, освободит поток и вернет его в пул потоков.

```
void AdvancedThreading_Unmanaged::InForkedThread(StackInfo * stackCopy)
{
    StackInfo copy;
```

Первым делом мы сохраняем значения рабочих регистров, чтобы, когда `MakeFork` завершит свою работу, мы смогли их безболезненно восставновить. Чтобы в дальнейшем минимально влиять на регистры, мы выгружаем переданные нам параметры к себе на стек. Доступ к ним будет идти только через `SS:ESP`, что для нас будет предсказуемым.

```
    short CS_EIP[3];

    // Save original registers to restore
    __asm pushad

    // safe copy w-out changing registers
    for(int i = 0; i < sizeof(StackInfo); i++)
        ((byte *)&copy)[i] = ((byte *)stackCopy)[i];

    // Setup FWORD for far jmp
    *(int*)CS_EIP = copy.EIP;
    CS_EIP[2] = copy.CS;
```

Наша следующая задача - это исправить в копии стека значения `EBP`, которые образуют односвязный список кадров на их будущие новые положения. Для этого мы рассчитываем дельту между адресом нашего стека потока и родительского стека потока, дельту между копией диапазона стека родительского потока и самим родительским потоком.

```
    // calculate ranges
    int beg = (int)copy.frame;
    int size = copy.size;
    int baseFrom = (int) copy.origStackStart;
    int baseTo = baseFrom + (int)copy.origStackSize;
    int ESPr;

    __asm mov ESPr, ESP

    // target = EBP[ - locals - EBP - ret - whole stack frames copy]
    int targetToCopy = ESPr - 8 - size;

    // offset between parent stack and current stack;
    int delta_to_target = (int)targetToCopy - (int)copy.EBP;

    // offset between parent stack start and its copy;
    int delta_to_copy = (int)copy.frame - (int)copy.EBP;
```

Используя эти данные мы в цикле идем по копии стека и исправляем адреса на их будущие новые положения.

```
    // In stack copy we have many saved EPBs, which where actually one-way linked list.
    // we need to fix copy to make these pointers correct for our thread's stack.
    int ebp_cur = beg;
    while(true)
    {
        int val = *(int*)ebp_cur;

        if(baseFrom <= val && val < baseTo)
        {
```

```
            int localOffset = val + delta_to_copy;
            *(int *)ebp_cur += delta_to_target;
            ebp_cur = localOffset;
        }
        else
            break;
    }
```

Когда правка односвязного списка завершена, мы должны исправить значения регистров в их копии, чтобы, если там присутствуют ссылки на стек, они были бы исправлены. Тут на самом деле алгоритм совсем не точен. Ведь если там по некоторой случайности окажется не удачное число из диапазона адресов стека, то оно будет исправлено по ошибке. Но наша задача не для продукта концепт написать, а просто понять работу стека потока. Потому для этих целей нам данная методика подойдет.

```
    CHECKREF(EAX);
    CHECKREF(EBX);
    CHECKREF(ECX);
    CHECKREF(EDX);

    CHECKREF(ESI);
    CHECKREF(EDI);
```

Теперь, основная и самая ответственная часть. Когда мы скопируем в конец нашего стека копию диапазона родительского, все будет хорошо до момента, когда `MakeFork` в дочернем потоке захочет выйти (сделать `return`). Нам надо указать ему куда он должен выйти. Для этого мы также имитируем вызов самого 'MakeFork' из этого метода. Мы закладываем в стек адрес метки `RestorePointAfterClonnedExited`, как будто инструкция процессора `call` заложила в стек адрес возврата, а также положили текущий `EBP`, сымитировав построение односвязного списка цепочек кадров методов. После чего закладываем в стек обычной операцией `push` копию родительского стека тем самым отрисовав все методы, которые были вызваны в родительском стеке из метода `MakeFork`, включая его самого. Стек готов!

```
    // prepare for __asm nret
    __asm push offset RestorePointAfterClonnedExited
    __asm push EBP

    for(int i = (size >> 2) - 1; i >= 0; i--)
    {
        int val = ((int *)beg)[i];
        __asm push val;
    };
```

Далее поскольку мы также должны восстановить и регистры, восстанивливаем и их самих.

```
    // restore registers, push 1 for Fork() and jmp
    _asm {
        push copy.EAX
        push copy.EBX
        push copy.ECX
        push copy.EDX
        push copy.ESI
        push copy.EDI
        pop EDI
        pop ESI
        pop EDX
        pop ECX
        pop EBX
        pop EAX
```

А вот теперь самое время вспомнить тот странный код с закладыванием `0` в стек и проверки на `0`. В этом потоке мы закладываем `1` и делаем дальний jmp в код метода `ForkImpl`. Ведь по стеку мы находимся именно там, а реально все еще тут. Когда мы туда попадем, то `ForkImpl` распознает смену потока и осуществит выход в метод `MakeFork`, который, завершив работу, попадет в точку `RestorePointAfterClonnedExited`, т.к. немного ранее мы сымтировали вызов `MakeFork` из этой точки. Восстановив регистры до состояния "только что вызваны из TheadPool", мы завершаем работу, отдавая поток в пул потоков.

```
        push 1
        jmp fword ptr CS_EIP
    }

RestorePointAfterClonnedExited:

    // Restore original registers
    __asm popad
    return;
 }
```

Проверим? Это - скриншот до вызова клонирования потока:

И после:

Как мы видим, теперь вместо одного потока внутри ForkImpl мы видим два. И оба - вышли из этого метода.

## Пара слов об уровне пониже

Если мы заглянем краем глаза на еще более низкий уровень, то узнаем или же вспомним, что память на самом деле является виртуальной и что она поделена на страницы объемом 8 или 4 Кб. Каждая такая страница может физически существовать или же нет. А если она существует, то может быть отображена на файл или же реальную оперативную память. Именно этот механизм виртуализации позволяет приложениям иметь раздельную друг от друга память и обеспечивает уровни безопасности между приложением и операционной системой. При чем же здесь стек потока? Как и любая другая оперативная память приложения стек потока является ее частью и также состоит из страниц объемом 4 или 8 Кб. По краям от выделенного для стека пространства находятся две страницы, доступ к которым приводит к системному исключению, нотифицирующему операционную систему о том, что приложение пытается обратиться в невыделенный участок памяти. Внутри этого региона реально выделенными участками являются только те страницы, к которым обратилось приложение: т.е. если приложение резервирует под поток 2Мб памяти, это не значит, что они будут выделены сразу же. Отнюдь, они будут выделены по требованию: если стек потока вырастет до 1 Мб, это будет означать, что приложение получило именно 1 Мб оперативной памяти под стек.

Когда приложение резервирует память под локальные переменные, то происходят две вещи: наращивается значение регистра ESP и зануляется память под сами переменные. Поэтому, когда вы напишете рекурсивный метод, который уходит в бесконечную рекурсию, вы получите StackOverflowException: заняв всю выделенную под стек память (весь доступный регион), вы напоритесь на специальную страницу, Guard Page, доступ к которой вызовет нотификацию операционной системы, которая инициирует StackOverflow уровня ОС, которое уйдет в .NET, будет перехвачено и выбросется исключение StackOverflowException для .NET приложения.

## Выделение памяти на стеке: stackalloc

В C# существует достаточно интересное и очень редко используемое ключевое слово `stackalloc`. Оно настолько редко встречается в коде (тут я даже со словом "немного" преуменьшил. Скорее, "никогда"), что найти подходящий пример его использования достаточно трудно, а уж придумать тем более трудно: ведь если что-то редко используется, то и опыт работы с ним слишком мал. А все почему? Потому что для тех, кто наконец решается выяснить, что делает эта команда, `stackalloc` становится более пугающим чем полезным: темная сторона `stackalloc` - unsafe код. Тот результат, что он возвращает не является managed указателем: значение - обычный указатель на участок не защищенной памяти. Причем если по этому адресу сделать запись уже после того, как метод завершил работу, вы начнете писать в локальные переменные некоторого метода или же вообще перетрете адрес возврата из метода, после чего приложение закончит работу с ошибкой. Однако наша задача - проникнуть в самые уголки и разобраться, что в них скрыто. И понять, в частности, что если нам дали этот инструмент, то не просто же так, чтобы мы смогли найти секретные грабли и наступить на них со всего маху. Наоборот: нам дали этот инструмент чтобы мы смогли им воспользоваться и делать поистине быстрый софт. Я, надеюсь, вдохновил вас? Тогда начнем.

Чтобы найти правильные примеры использования этого ключевого слова надо проследовать прежде всего к его авторам: компании Microsoft и посмотреть как его используют они. Сделать это можно поискав полнотекстовым поиском по репозиторию coreclr. Помимо различных тестов самого ключевого слова мы найдем не более 25 использований этого ключевого слова по коду библиотеки. Я надеюсь что в предыдущем абзаце я достаточно сильно вас мотивировал чтобы вы не остановили чтение, увидев эту маленькую цифру, и не закрыли мой труд. Скажу честно: команда CLR куда более дальновидная и профессиональная чем команда .NET Framework. И если она что-то сделала, то это нам сильно в чем-то должно помочь. А если это не использовано в .NET Framework... Ну, тут можно

предположить, что там не все инженеры в курсе, что есть такой мощный инструмент оптимизации. Иначе бы объемы его использования были бы гораздо больше.

**Класс Interop.ReadDir**

/src/mscorlib/shared/Interop/Unix/System.Native/Interop.ReadDir.cs

```
unsafe
{
    // s_readBufferSize is zero when the native implementation does not support reading
into a buffer.
    byte* buffer = stackalloc byte[s_readBufferSize];
    InternalDirectoryEntry temp;
    int ret = ReadDirR(dir.DangerousGetHandle(), buffer, s_readBufferSize, out temp);
    // We copy data into DirectoryEntry to ensure there are no dangling references.
    outputEntry = ret == 0 ?
                new DirectoryEntry() { InodeName = GetDirectoryEntryName(temp),
InodeType = temp.InodeType } :
                default(DirectoryEntry);

    return ret;
}
```

Для чего здесь используется `stackalloc`? Как мы видим, после выделения памяти код уходит в unsafe метод для заполнения созданного буфера данными. Т.е. unsafe метод, которому необходим участок для записи, выделяется место прямо на стеке: динамически. Это отличная оптимизация, если учесть, что альтернативы: запросить участок памяти у Windows или fixed (pinned) массив .NET, который помимо нагрузки на кучу нагружает GC тем, что массив прибивается гвоздями, чтобы GC его не пододвинул во время доступа к его данным. Выделяя память на стеке мы не рискуем ничем: выделение происходит почти моментально, и мы можем совершенно спокойно заполнить его данными и выйти из метода. А вместе с выходом из метода исчезнет и stack frame метода. В общем, экономия времени значительнейшая.

Давайте рассмотрим еще один пример:

**Класс Number.Formatting::FormatDecimal**

/src/mscorlib/shared/System/Number.Formatting.cs

```
public static string FormatDecimal(decimal value, ReadOnlySpan<char> format,
NumberFormatInfo info)
{
    char fmt = ParseFormatSpecifier(format, out int digits);

    NumberBuffer number = default;
    DecimalToNumber(value, ref number);
```

```
    ValueStringBuilder sb;
    unsafe
    {
        char* stackPtr = stackalloc char[CharStackBufferSize];
        sb = new ValueStringBuilder(new Span<char>(stackPtr, CharStackBufferSize));
    }

    if (fmt != 0)
    {
        NumberToString(ref sb, ref number, fmt, digits, info, isDecimal:true);
    }
    else
    {
        NumberToStringFormat(ref sb, ref number, format, info);
    }

    return sb.ToString();
}
```

Это - пример форматирования чисел, опирающийся на еще более интересный пример класса ValueStringBuilder, работающий на основе `Span<T>`. Суть данного участка кода в том, что для того чтобы собрать текстовое представление форматированного числа максимально быстро, код не использует выделения памяти под буфер накопления символов. Этот прекрасный код выделяет память прямо в стековом кадре метода, обеспечивая тем самым отсутствие работы сборщика мусора по экземплярам StringBuilder, если бы метод работал на его основе. Плюс уменьшается время работы самого метода: выделение памяти в куче тоже время занимает. А использование типа `Span<T>` вместо голых указателей вносит чувство безопасности в работу кода, основанного на stackalloc.

Также, перед тем как перейти к выводам, стоит упомянуть, как делать нельзя. Другими словами, какой код может работать хорошо, но в один прекрасный момент выстрелит в самый не подходящий момент. Опять же, рассмотрим пример:

```
void GenerateNoise(int noiseLength)
{
    var buf = new Span(stackalloc int[noiseLength]);
    // generate noise
}
```

Код мал да удал: нельзя вот так брать и передавать размер для выделения памяти на стеке извне. Если вам так нужен заданный снаружи размер, примите сам буфер:

```
void GenerateNoise(Span<int> noiseBuf)
{
    // generate noise
}
```

Этот код гораздо информативнее, т.к. заставляет пользователя задуматься и быть аккуратным при выборе чисел. Первый вариант при неудачно сложившихся обстоятельствах может выбросить `StackOverflowException` при достаточно неглубоком положении метода в стеке потока: достаточно передать большое число в качестве параметра. Второй вариант, когда размер принимать все-таки можно - это когда этот метод вызывается в конкретных случаях и при этом вызывающий код "знает" алгоритм работы этого метода. Без знания о внутреннем устройстве метода нет конкретного понимания возможного диапазона для noiseLength и как следствие - возможны ошибки

Вторая проблема, которую я вижу: если нам случайным образом не удалось попасть в размер того буфера, который мы сами себе выделили на стеке, а терять работоспособность мы не хотим, то, конечно, можно пойти несколькими путями: либо довыделить памяти, опять же на стеке, либо выделить ее в куче. Причем скорее всего второй вариант в большинстве случаев окажется более предпочтительным (так и поступили в случае `ValueStringBuffer`), т.к. более безопасен с точки зрения получения `StackOverflowException`.

## Выводы к stackalloc

Итак, для чего же лучше всего использовать `stackalloc`?

- Для работы с неуправляемым кодом, когда необходимо заполнить неуправляемым методом некоторый буфер данных или же принять от неуправляемого метода некий буфер данных, который будет использоваться в рамках жизни тела метода;
- Для методов, которым нужен массив, но опять же на время работы самого метода. Пример с форматированием очень хороший: этот метод может вызываться слишком часто чтобы он выделял временные массивы в куче;

Использование данного аллокатора может сильно повысить производительность ваших приложений.

## Выводы к разделу

Конечно же, в общем виде нам нет надобности редактировать стек в продуктовом коде: только если захочется занять свое свободное время интересной задачкой. Однако понимание его структуры дает нам видимость простоты задачи получения данных из него и его редактирования. Т.е. если вы разработаете API для расширения функционала вашего приложения и если это API не предоставляет доступа к каким-либо данным это не значит что эти данные невозможно получить. Потому всегда проверяйте ваше приложение на устойчивость к взломам.

## Memory<T> и Span<T>

Начиная с версий .NET Core 2.0 и .NET Framework 4.5 нам стали доступны новые типы данных: это `Span` и `Memory`. Чтобы ими воспользоваться, достаточно установить nuget пакет `System.Memory`:

```
PM> Install-Package System.Memory
```

И примечательны эти типы данных тем, что специально для них была проделана огромная работа командой `CLR`чтобы реализовать их специальную поддержку в коде JIT компилятора .NET Core 2.1+, вживив их прямо в ядро. Что это за типы данных и почему на них стоит выделить целую часть книги?

Если говорить о проблематике, приведшей к появлению данного функционала, я бы выбрал три основные. И первая из них - это неуправляемый код.

Как язык, так и платформа существуют уже много лет: и все это время существовало множество средств для работы с неуправляемым кодом. Так почему же сейчас выходит очередной API для работы с неуправляемым кодом, если, по сути, он существовал уже много-много лет? Для того чтобы ответить на этот вопрос, достаточно понять, чего не хватало нам раньше.

Разработчики платформы и раньше пытались нам помочь скрасить будни разработки с использованием неуправляемых ресурсов: это и автоматические врапперы для импортируемых методов, и маршаллинг, который в большинстве случаев работает автоматически. Это также инструкция `stackalloc`, о которой говорится в главе про стек потока. Однако, как по мне если ранние разработчики с использованием языка C# приходили из мира C++ (как сделал это и я), то сейчас они приходят из более высокоуровневых языков (я, например, знаю разработчика, который пришел из JavaScript). Это означает, что люди со все большим подозрением начинают относиться к неуправляемым ресурсам и конструкциям, близким по духу к C/C++, и уж тем более - к языку Ассемблера.

Как результат такого отношения - все меньшее и меньшее содержание unsafe кода в проектах и все большее доверие к API самой платформы. Это легко проверяется, если поискать использование конструкции `stackalloc` по открытым репозиториям: оно ничтожно мало. Но если взять любой код, который его использует:

**Класс Interop.ReadDir**

/src/mscorlib/shared/Interop/Unix/System.Native/Interop.ReadDir.cs

```
unsafe
{
    // s_readBufferSize is zero when the native implementation does not support reading
into a buffer.
    byte* buffer = stackalloc byte[s_readBufferSize];
    InternalDirectoryEntry temp;
    int ret = ReadDirR(dir.DangerousGetHandle(), buffer, s_readBufferSize, out temp);
    // We copy data into DirectoryEntry to ensure there are no dangling references.
    outputEntry = ret == 0 ?
                new DirectoryEntry() { InodeName = GetDirectoryEntryName(temp),
InodeType = temp.InodeType } :
                default(DirectoryEntry);

    return ret;
}
```

Становится понятна причина непопулярности. Посмотрите, не вчитываясь в код, и ответьте для себя на один вопрос: доверяете ли вы ему? Могу предположить что ответом будет "нет". Тогда ответьте на другой: почему? Ответ будет очевиден: помимо того что мы видим слово `Dangerous`, которое как-бы намекает, что что-то может пойти не так, второй фактор, влияющий на наше отношение, - это запись `unsafe` и строчка `byte* buffer = stackalloc byte[s_readBufferSize];`, а если еще конкретнее - `byte*`. Эта запись - триггер для любого, чтобы в голове появилась мысль: "а что, по-другому сделать нельзя было что-ли?". Тогда давайте еще чуть-чуть разберемся с психоанализом: отчего может возникнуть подобная мысль? С одной стороны мы пользуемся конструкциями языка и предложенный здесь синтаксис далек от, например, C++/CLI, который позволяет делать вообще все что угодно (в том числе делать вставки на чистом Assembler), а с другой он выглядит непривычно.

Второй вопрос, который явно или неявно возникал в головах у многих разработчиков - это несовместимость типов: строки `string` и массива символов `char[]`, хотя чисто логически строка - это и есть массив символов, привести string к char[] возможности нет никакой: только создание нового объекта и

копирование содержимого строки в массив. Причем несовместимость такая введена для оптимизации строк с точки зрения хранения (readonly массивов не существует), но проблемы возникают, когда вы начинаете работать с файлами. Как прочитать? Строкой или массивом? Ведь если массивом, станет не возможным пользоваться некоторыми методами, рассчитанными на работу со строками. А если строкой? Может оказаться слишком длинной. Если речь идет о последующем парсинге, возникает вопрос выбора парсеров примитивов: далеко не всегда хочется парсить их все вручную (целые числа, числа с плавающей точкой, в разных форматах записанных). Есть же множество алгоритмов, проверенных годами, которые делают это быстро и эффективно. Но такие алгоритмы часто работают на строках, в которых кроме самого примитива ничего другого нет. Другими словами, дилемма.

Третья проблематика заключается в том, что необходимые для некоторого алгоритма данные редко лежат от начала и до самого конца считанного с некоторого источника массива. Например, если речь опять же идет о файлах или о данных, считанных с сокета, то есть некоторая уже обработанная неким алгоритмом часть, дальше идет то, что должен обработать некий наш метод, после чего идут данные, которые нам обработать еще предстоит. И этот самый метод в идеале хочет чтобы ему отдали только то что он ожидает. Например, метод парсинга целых чисел вряд-ли будет сильно благодарен, если вы отдадите ему строку с разговором о чём-либо, где в некоторой позиции будет находиться нужное число. Такой метод ожидает что вы отдадите ему число и ничего больше. Если же мы отдадим массив целиком, то возникает требование указать, например, смещение числа относительно начала массива:

```
int ParseInt(char[] input, int index)
{
    while(char.IsDigit(input[index]))
    {
        // ...
        index++;
    }
}
```

Однако, данный способ плох тем что метод получает данные, которые ему не нужны. Другими словами, *метод не введен в контекст своей задачи*. Ведь

помимо своей задачи метод решает еще и некоторую внешнюю. А это - признак плохого проектирования. Как избежать данной проблемы? Как вариант, можно воспользоваться типом `ArraySegment<T>`, суть которого - предоставить "окно" в некий массив:

```
int ParseInt(IList<char>[] input)
{
    while(char.IsDigit(input.Array[index]))
    {
        // ...
        index++;
    }
}

var arraySegment = new ArraySegment(array, from, length);
var res = ParseInt((IList<char>)arraySegment);
```

Но как по мне, так это - перебор как с точки зрения логики, так и с точки зрения просадки по производительности. `ArraySegment` - ужасно сделан и обеспечивает замедление доступа к элементам в 7 раз относительно тех же самых операций, но с массивом.

Так как же решить все эти проблемы? Как вернуть разработчиков обратно в лоно неуправляемого кода, при этом дав им механизм унифицированной и быстрой работы с разнородными источниками данных: массивами, строками и неуправляемой памятью? Необходимо дать им чувство спокойствия, что они не могут сделать ошибку случайно, по незнанию. Необходимо дать им инструмент, не уступающий в производительности нативным типам данных, но решающих перечисленные проблемы. Этим инструментом являются типы `Span<T>` и `Memory<T>`.

## Span<T>, ReadOnlySpan<T>

Тип `Span` олицетворяет собой инструмент для работы с данными части некоторого массива данных либо поддиапазона его значений. При этом позволяя как и в случае массива работать с элементами этого диапазона как на запись, так и на чтение но с одним важным ограничением: `Span<T>` вы получаете или создаете только для того, чтобы *временно* поработать с массивом. В рамках вызова группы методов, но не более того. Однако, давайте для разгона и общего понимания сравним типы

данных, для которых сделана реализация типа Span, и посмотрим на возможные сценарии для работы с ним.

Первый тип данных, о котором хочется завести речь, - это обычный массив. Для массивов работа со Span будет выглядеть следующим образом:

```
    var array = new [] {1,2,3,4,5,6};
    var span = new Span<int>(array, 1, 3);
    var position = span.BinarySearch(3);
    Console.WriteLine(span[position]);  // -> 3
```

Как мы видим в данном примере, для начала мы создаем некий массив данных. После этого мы создаем Span (или подмножество), который, ссылаясь на сам массив, разрешает доступ использующему его коду только в тот диапазон значений, который был указан при инициализации.

Тут мы видим первое свойство этого типа данных: это создание некоторого контекста. Давайте разовьем нашу идею с контекстами:

```
void Main()
{
    var array = new [] {'1','2','3','4','5','6'};
    var span = new Span<char>(array, 1, 3);
    if(TryParseInt32(span, out var res))
    {
        Console.WriteLine(res);
    }
    else
    {
        Console.WriteLine("Failed to parse");
    }
}

public bool TryParseInt32(Span<char> input, out int result)
{
    result = 0;
    for (int i = 0; i < input.Length; i++)
    {
        if(input[i] < '0' || input[i] > '9')
            return false;
    result = result * 10 + ((int)input[i] - '0');
    }
    return true;
}

-----
234
```

Как мы видим, Span<T> вводит абстракцию доступа к некоторому участку памяти как на чтение, так и на запись. Что нам это дает? Если вспомнить, на основе чего еще

может быть сделан `Span`, то мы вспомним как про неуправляемые ресурсы, так и про строки:

```
// Managed array
var array = new[] { '1', '2', '3', '4', '5', '6' };
var arrSpan = new Span<char>(array, 1, 3);
if (TryParseInt32(arrSpan, out var res1))
{
    Console.WriteLine(res1);
}

// String
var srcString = "123456";
var strSpan = srcString.AsSpan();
if (TryParseInt32(arrSpan, out var res2))
{
    Console.WriteLine(res2);
}

// void *
Span<char> buf = stackalloc char[6];
buf[0] = '1'; buf[1] = '2'; buf[2] = '3';
buf[3] = '4'; buf[4] = '5'; buf[5] = '6';

if (TryParseInt32(arrSpan, out var res3))
{
    Console.WriteLine(res3);
}

-----
234
234
234
```

Т.е., получается, что `Span<T>` - это средство унификации по работе с памятью: управляемой и неуправляемой, которое гарантирует безопасность в работе с такого рода данными во время Garbage Collection: если участки памяти с управляемыми массивами начнут двигаться, то для нас это будет безопасно.

Однако, стоит ли так сильно радоваться? Можно ли было всего этого добиться и раньше? Например, если говорить об управляемых массивах, то тут даже сомневаться не приходится: достаточно просто обернуть массив в еще один класс (например давно существующий ArraySegment), предоставив аналогичный интерфейс, и все готово. Мало того, аналогичную операцию можно проделать и со строками: они обладают необходимыми методами. Опять же, достаточно строку завернуть в точно такой же тип и предоставить методы по работе с ней. Другое дело, что для того чтобы хранить в одном типе строку, буфер или массив, придется сильно

повозиться, храня в едином экземпляре ссылки на каждый из возможных вариантов (активным, понятное дело, будет только один):

```
public readonly ref struct OurSpan<T>
{
    private T[] _array;
    private string _str;
    private T * _buffer;

    // ...
}
```

Или же если отталкиваться от архитектуры, то делать три типа, наследующих единый интерфейс. Получается, что невозможно сделать средство единого интерфейса, отличное от `Span<T>`, между этими типами данных, сохранив при этом максимальную производительность.

Далее, если продолжить рассуждения, то что такое `ref struct` в понятиях `Span`? Это именно те самые "структуры, они только на стеке", о которых мы так часто слышим на собеседованиях. А это значит, что этот тип данных может идти только через стек и не имеет права уходить в кучу. А потому `Span`, будучи ref структурой, является контекстным типом данных, обеспечивающим работу методов, но не объектов в памяти. От этого для его понимания и надо отталкиваться.

Отсюда мы можем сформулировать определение типа Span и связанного с ним readonly типа ReadOnlySpan:

> *Span - это тип данных, обеспечивающий единый интерфейс работы с разнородными типами массивов данных, а также возможность передать в другой метод подмножество этого массива таким образом, чтобы вне зависимости от глубины взятия контекста скорость доступа к исходному массиву была константной и максимально высокой.*

И действительно: если мы имеем примерно такой код:

```
public void Method1(Span<byte> buffer)
{
    buffer[0] = 0;
    Method2(buffer.Slice(1,2));
}
Method2(Span<byte> buffer)
```

```
{
    buffer[0] = 0;
    Method3(buffer.Slice(1,1));
}
Method3(Span<byte> buffer)
{
    buffer[0] = 0;
}
```

то скорость доступа к исходному буферу будет максимально высокой: вы работаете не с managed объектом, а с managed указателем. Т.е. не с .NET managed типом, а с unsafe типом, заключенным в managed оболочку.

## Span<T> на примерах

Человек так устроен, что зачастую пока он не получит определенного опыта, то конечного понимания, для чего необходим инструмент, часто не приходит. А потому, поскольку нам нужен некий опыт, давайте обратимся к примерам.

### *ValueStringBuilder*

Одним из самых алгоритмически интересных примеров является тип `ValueStringBuilder`, который прикопан где-то в недрах `mscorlib` и почему-то как и многие другие интереснейшие типы данных помечен модификатором `internal`, что означает, что, если бы не исследование исходного кода mscorlib, о таком замечательном способе оптимизации мы бы никогда не узнали.

Каков основной минус системного типа `StringBuilder`? Это конечно же его суть: как он сам, так и то, на чем он основан (а это массив символов `char[]`) - являются типами ссылочными. А это значит как минимум две вещи: мы все равно (хоть и немного) нагружаем кучу и второе - увеличиваем шансы промаха по кэшам процессора.

Еще один вопрос, который у меня возникал к `StringBuilder` - это формирование маленьких строк. Т.е. когда результирующая строка "зуб даю" будет короткой: например, менее 100 символов. Когда мы имеем достаточно короткие форматирования, к производительности возникают вопросы:

```
    $"{x} is in range [{min};{max}]"
```

Насколько эта запись хуже чем ручное формирование через `StringBuilder`? Ответ далеко не всегда очевиден: все сильно зависит от места формирования: как часто будет вызван данный метод. Ведь сначала `string.Format` выделяет память под внутренний `StringBuilder`, который создаст массив символов (SourceString.Length + args.Length * 8) и если в процессе формирования массива выяснится, что длина не была угадана, то для формирования продолжения будет создан еще один `StringBuilder`, формируя тем самым односвязный список. А в результате - необходимо будет вернуть сформированную строку: а это еще одно копирование. Транжирство и расточительство. Вот если бы избавиться от размещения в куче первого массива формируемой строки, было бы замечательно: от одной проблемы мы бы точно избавились.

Взглянем на тип из недр `mscorlib`:

**Класс ValueStringBuilder** /src/mscorlib/shared/System/Text/ValueStringBuilder

```
    internal ref struct ValueStringBuilder
    {
        // это поле будет активно если у нас слишком много символов
        private char[] _arrayToReturnToPool;
        // это поле будет основным
        private Span<char> _chars;
        private int _pos;
        // тип принимает буфер извне, делигируя выбор его размера вызывающей стороне
        public ValueStringBuilder(Span<char> initialBuffer)
        {
            _arrayToReturnToPool = null;
            _chars = initialBuffer;
            _pos = 0;
        }

        public int Length
        {
            get => _pos;
            set
            {
                int delta = value - _pos;
                if (delta > 0)
                {
                    Append('\0', delta);
                }
                else
                {
                    _pos = value;
                }
            }
        }

        // Получение строки - копирование символов из массива в массив
```

```
        public override string ToString()
        {
            var s = new string(_chars.Slice(0, _pos));
            Clear();
            return s;
        }

        // Вставка в середину сопровождается развиганием символов
        // исходной строки чтобы вставить необходимый: путем копирования
        public void Insert(int index, char value, int count)
        {
            if (_pos > _chars.Length - count)
            {
                Grow(count);
            }

            int remaining = _pos - index;
            _chars.Slice(index, remaining).CopyTo(_chars.Slice(index + count));
            _chars.Slice(index, count).Fill(value);
            _pos += count;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Append(char c)
        {
            int pos = _pos;
            if (pos < _chars.Length)
            {
                _chars[pos] = c;
                _pos = pos + 1;
            }
            else
            {
                GrowAndAppend(c);
            }
        }

        [MethodImpl(MethodImplOptions.NoInlining)]
        private void GrowAndAppend(char c)
        {
            Grow(1);
            Append(c);
        }

        // Если исходного массива, переданного конструктором не хватило
        // мы выделяем массив из пула свободных необходимого размера
        // На самом деле идеально было бы если бы алгоритм дополнительно создавал
        // дискретность в размерах массивов чтобы пул не был бы фрагментированным
        [MethodImpl(MethodImplOptions.NoInlining)]
        private void Grow(int requiredAdditionalCapacity)
        {
            Debug.Assert(requiredAdditionalCapacity > _chars.Length - _pos);

            char[]    poolArray    =    ArrayPool<char>.Shared.Rent(Math.Max(_pos    +
requiredAdditionalCapacity, _chars.Length * 2));

            _chars.CopyTo(poolArray);

            char[] toReturn = _arrayToReturnToPool;
            _chars = _arrayToReturnToPool = poolArray;
            if (toReturn != null)
```

```
            {
                ArrayPool<char>.Shared.Return(toReturn);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void Clear()
        {
            char[] toReturn = _arrayToReturnToPool;
            this = default; // for safety, to avoid using pooled array if this instance
is erroneously appended to again
            if (toReturn != null)
            {
                ArrayPool<char>.Shared.Return(toReturn);
            }
        }

        // Пропущенные методы: с ними и так все ясно
        private void AppendSlow(string s);
        public bool TryCopyTo(Span<char> destination, out int charsWritten);
        public void Append(string s);
        public void Append(char c, int count);
        public unsafe void Append(char* value, int length);
        public Span<char> AppendSpan(int length);
    }
```

Этот класс по своему функционалу сходен со своим старшим собратом `StringBuilder`, обладая при этом одной интересной и очень важной особенностью: он является значимым типом. Т.е. хранится и передается целиком по значению. А новейший модификатор типа `ref`, который приписан к сигнатуре объявления типа, говорит нам о том, что данный тип обладает дополнительным ограничением: он имеет право находиться только на стеке. Т.е. вывод его экмепляров в поля классов приведет к ошибке. К чему все эти приседания? Для ответа на этот вопрос достаточно посмотреть на класс `StringBuilder`, суть которого мы только что описали:

### Класс StringBuilder

/src/mscorlib/src/System/Text/StringBuilder.cs

```
public sealed class StringBuilder : ISerializable
{
    // A StringBuilder is internally represented as a linked list of blocks each of
which holds
    // a chunk of the string.  It turns out string as a whole can also be represented
as just a chunk,
    // so that is what we do.
    internal char[] m_ChunkChars;                // The characters in this block
    internal StringBuilder m_ChunkPrevious;      // Link to the block logically before
this block
    internal int m_ChunkLength;                    // The index in m_ChunkChars that
represent the end of the block
    internal int m_ChunkOffset;                    // The logical offset (sum of all
characters in previous blocks)
```

```
    internal int m_MaxCapacity = 0;

    // ...

    internal const int DefaultCapacity = 16;
```

`StringBuilder` - это класс, внутри которого находится ссылка на массив символов. Т.е. когда вы создаете его, то по сути создается как минимум два объекта: сам `StringBuilder` и массив символов в как минимум 16 символов (кстати именно поэтому так важно задавать предполагаемую длину строки: ее построение будет идти через генерацию односвязного списка 16-символьных массивов. Согласитесь, расточительство). Что это значит в контексте нашего разговора о типе `ValueStringBuilder`: capacity по-умолчанию отсутствует, т.к. он заимствует память извне, плюс он сам является значимым типом и заставляет пользователя расположить буфер для символов на стеке. Как итог весь экземпляр типа помещается на стек вместе с его содержимым и вопрос оптимизации здесь становится решенным. Нет выделения памяти в куче? Нет проблем с проседанием производительности по куче. Но вы мне скажите: почему тогда не пользоваться ValueStringBuilder (или его самописной версией: сам он internal и нам не доступен) всегда? Ответ такой: надо смотреть на задачу, которая вами решается. Будет ли результирующая строка известного размера? Будет ли она иметь некий известный максимум по длине? Если ответ "да" и если при этом размер строки не выходит за некоторые разумные границы, то можно использовать значимую версию `StringBuilder`. Иначе, если мы ожидаем длинные строки, переходим на использование обычной версии.

### *ValueListBuilder*

Второй тип данных, который хочется особенно - отметить - это тип `ValueListBuilder`. Создан он для ситуаций, когда необходимо на короткое время создать некоторую коллекцию элементов и тут же отдать ее в обработку некоторому алгоритму.

Согласитесь: задача очень похожа на задачу `ValueStringBuilder`. Да и решена она очень похожим образом:

**Файл [ValueListBuilder.cs]**

```
internal ref partial struct ValueListBuilder<T>
{
    private Span<T> _span;
    private T[] _arrayFromPool;
    private int _pos;

    public ValueListBuilder(Span<T> initialSpan)
    {
        _span = initialSpan;
        _arrayFromPool = null;
        _pos = 0;
    }

    public int Length { get; set; }

    public ref T this[int index] { get; }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public void Append(T item);

    public ReadOnlySpan<T> AsSpan();

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public void Dispose();

    private void Grow();
}
```

Если говорить прямо, то такие ситуации достаточно частые. Однако раньше мы решали этот вопрос другим способом: создавали `List`, заполняли его данными и теряли ссылку. Если при этом метод вызывается достаточно часто, возникает печальная ситуация: множество экземпляров класса `List` повисает в куче, а вместе с ними повисают в куче и массивы, с ними ассоциированные. Теперь эта проблема решена: дополнительных объектов создано не будет. Однако, как и в случае с `ValueStringBuilder`, решена она только для программистов Microsoft: класс имеет модификатор `internal`.

## Правила и практика использования

Для того чтобы окончательно понять суть нового типа данных, необходимо "поиграться" с ним, написав пару-тройку, а лучше - больше методов, его использующих. Однако, основные правила можно почерпнуть уже сейчас:

- Если ваш метод будет обрабатывать некоторый входящий набор данных, не меняя его размер, можно попробовать остановиться на типе `Span`. Если при этом не будет модификации этого буфера, то на типе `ReadOnlySpan`;
- Если ваш метод будет работать со строками, вычисляя какую-то статистику либо производя синтаксический разбор строки, то ваш метод *обязан* принимать `ReadOnlySpan<char>`. Именно обязан: это новое правило. Ведь если вы принимаете строку, тем самым вы заставляете кого-то сделать для вас подстроку
- Если необходимо в рамках работы метода сделать достаточно короткий массив с данными (скажем, 10Кб максимум), то вы с легкостью можете организовать такой массив при помощи `Span<TType> buf = stackalloc TType[size]`. Однако, конечно, TType должен быть только значимым типом, т.к. `stackalloc` работает только со значимыми типами.

В остальных случаях стоит присмотреться либо к `Memory` либо использовать классические типы данных.

## Как работает Span

Дополнительно хотелось бы поговорить о том, как работает `Span` и что в нем такого примечательного. А поговорить есть о чем: сам тип данных делится на две версии: для .NET Core 2.0+ и для всех остальных.

**Файл** Span.Fast.cs, .NET Core 2.0

```
public readonly ref partial struct Span<T>
{
    /// Ссылка на объект .NET или чистый указатель
    internal readonly ByReference<T> _pointer;
    /// Длина буфера данных по указателю
    private readonly int _length;
    // ...
}
```

**Файл** ??? [decompiled]

```
public ref readonly struct Span<T>
{
    private readonly System.Pinnable<T> _pinnable;
    private readonly IntPtr _byteOffset;
    private readonly int _length;
```

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

Все дело в том что *большой* .NET Framework и .NET Core 1.* не имеют специальным образом измененного сборщика мусора (в отличии от версии .NET Core 2.0+) и потому вынуждены тащить за собой дополнительный указатель: на начало буфера, с которым идет работа. Т.е., получается, что `Span` внутри себя работает с управляемыми объектами платформы .NET как с неуправляемыми. Взгляните на внутренности второго варианта структуры: там присутствует три поля. Первое поле - это ссылка на managed объект. Второе - смещение относительно начала этого объекта в байтах, чтобы получить начало буфера данных (в строках это буфер с символами `char`, в массивах - буфер с данными массива). И, наконец, третье поле - количество уложенных друг за другом элементов этого буфера.

Для примера возьмем работу `Span` для строк:

**Файл** coreclr::src/System.Private.CoreLib/shared/System/MemoryExtensions.Fast.cs

```
public static ReadOnlySpan<char> AsSpan(this string text)
{
    if (text == null)
        return default;

    return new ReadOnlySpan<char>(ref text.GetRawStringData(), text.Length);
}
```

Где `string.GetRawStringData()` выглядит следующим образом:

**Файл с определением полей**

coreclr::src/System.Private.CoreLib/src/System/String.CoreCLR.cs

**Файл с определением GetRawStringData**

coreclr::src/System.Private.CoreLib/shared/System/String.cs

```
public sealed partial class String :
    IComparable, IEnumerable, IConvertible, IEnumerable<char>,
    IComparable<string>, IEquatable<string>, ICloneable
{

    //
    // These fields map directly onto the fields in an EE StringObject.  See object.h
for the layout.
    //
    [NonSerialized] private int _stringLength;

    // For empty strings, this will be '\0' since
    // strings are both null-terminated and length prefixed
```

```
    [NonSerialized] private char _firstChar;


    internal ref char GetRawStringData() => ref _firstChar;
}
```

Т.е. получается, что метод лезет напрямую вовнутрь строки, а спецификация `ref char` позволяет отслеживать GC неуправляемую ссылку во внутрь строки, перемещая его вместе со строкой во время срабатывания GC.

Та же самая история происходит и с массивами: когда создается `Span`, то некий код внутри JIT рассчитывает смещение начала данных массива и этим смещением инициализирует `Span`. А как подсчитать смещения для строк и массивов, мы научились в главе про структуру объектов в памяти.

## Span<T> как возвращаемое значение

Несмотря на всю идиллию, связанную со `Span`, существуют хоть и логичные, но неожиданные ограничения на его возврат из метода. Если взглянуть на следующий код:

```
unsafe void Main()
{
    var x = GetSpan();
}

public Span<byte> GetSpan()
{
    Span<byte> reff = new byte[100];
    return reff;
}
```

то все выглядит крайне логично и хорошо. Однако, стоит заменить одну инструкцию другой:

```
unsafe void Main()
{
    var x = GetSpan();
}

public Span<byte> GetSpan()
{
    Span<byte> reff = stackalloc byte[100];
    return reff;
}
```

как компилятор запретит инструкцию такого вида. Но прежде чем написать, почему, я прошу вас самим догадаться, какие проблемы понесет за собой такая конструкция.

Итак, я надеюсь, что вы подумали, построили догадки и предположения, а может даже и поняли причину. Если так, главу про стек потока я по винтикам расписывал не зря. Ведь дав таким образом ссылку на локальные переменные метода, который закончил работу, вы можете вызвать другой метод, дождаться окончания его работы и через x[0.99] прочитать его локальные переменные.

Однако, к счастью, когда мы делаем попытку написать такого рода код, компилятор дает нам по рукам, выдав предупреждение: `CS8352 Cannot use local 'reff' in this context because it may expose referenced variables outside of their declaration scope` и будет прав: ведь если обойти эту ошибку, то возникает возможность, например, находясь в плагине подстроить такую ситуацию, что станет возможным украсть чужие пароли или повысить привилегии выполнения нашего плагина.

## Memory<T> и ReadOnlyMemory<T>

Визуальных отличий `Memory<T>` от `Span<T>` два. Первое - тип `Memory<T>` не содержит ограничения `ref` в заголовке типа. Т.е., другими словами, тип `Memory<T>` имеет право находиться не только на стеке, являясь либо локальной переменной, либо параметром метода, либо его возвращаемым значением, но и находиться в куче, ссылаясь оттуда на некоторые данные в памяти. Однако, эта маленькая разница создает огромную разницу в поведении и возможностях `Memory<T>` в сравнении с `Span<T>`. В отличии от `Span<T>`, который представляет собой *средство пользования* неким буфером данных для некоторых методов, тип `Memory<T>` предназначен для *хранения* информации о буфере, а не для работы с ним. Отсюда возникает разница в API:

- `Memory<T>` не содержит методов доступа к данным, которыми он заведует. Вместо этого он имеет свойство `Span` и метод `Slice`, которые возвращают рабочую лошадку - экземпляр типа `Span`.
- `Memory<T>` дополнительно содержит метод `Pin()`, предназначенный для сценариев, когда хранящийся буфер необходимо передать в `unsafe` код. При

его вызове для случаев, когда память была выделена в .NET, буфер будет закреплен (pinned) и не будет перемещаться при срабатывании GC, возвращая пользователю экземпляр структуры `MemoryHandle`, инкапсулирующей в себе понятие отрезка жизни `GCHandle`, закрепившего буфер в памяти.

Однако, для начала предлагаю познакомиться со всем набором классов. И в качестве первого из них, взглянем на саму структуру `Memory<T>` (показаны не все члены типа, а показавшиеся наиболее важными):

```
    public readonly struct Memory<T>
    {
        private readonly object _object;
        private readonly int _index, _length;

        public Memory(T[] array) { ... }
        public Memory(T[] array, int start, int length) { ... }
        internal Memory(MemoryManager<T> manager, int length) { ... }
        internal Memory(MemoryManager<T> manager, int start, int length) { ... }

        public int Length => _length & RemoveFlagsBitMask;
        public bool IsEmpty => (_length & RemoveFlagsBitMask) == 0;

        public Memory<T> Slice(int start, int length);
        public void CopyTo(Memory<T> destination) => Span.CopyTo(destination.Span);
        public        bool        TryCopyTo(Memory<T>        destination)        =>
Span.TryCopyTo(destination.Span);

        public Span<T> Span { get; }
        public unsafe MemoryHandle Pin();
    }
```

Как мы видим, структура содержит конструктор на основе массивов, но хранит данные в object. Сделано это для того чтобы дополнительно ссылаться на строки, для которых конструктор не предусмотрен, зато предусмотрен метод расширения для типа `string` `AsMemory()`, возвращающий `ReadOnlyMemory`. Однако, поскольку бинарно оба типа должны быть одинаковыми, типом поля `_object` является `Object`.

Далее мы видим два конструктора, работающих на основе `MemoryManager`. О них мы поговорим попозже. Свойства получения размера `Length` и проверки на пустое множество `IsEmpty`. Также имеется метод получения подмножества `Slice` и методы копирования `CopyTo` и `TryCopyTo`.

Подробнее же в разговоре о `Memory` хочется остановиться на двух методах этого типа: на свойстве `Span` и методе `Pin`.

## Memory<T>.Span

```
public Span<T> Span
{
    get
    {
        if (_index < 0)
        {
            return        ((MemoryManager<T>)_object).GetSpan().Slice(_index        &
RemoveFlagsBitMask, _length);
        }
        else if (typeof(T) == typeof(char) && _object is string s)
        {
            // This is dangerous, returning a writable span for a string that should
be immutable.
            // However, we need to handle the case where a ReadOnlyMemory<char> was
created from a string
            // and then cast to a Memory<T>. Such a cast can only be done with unsafe
or marshaling code,
            // in which case that's the dangerous operation performed by the dev, and
we're just following
            // suit here to make it work as best as possible.
            return  new  Span<T>(ref  Unsafe.As<char,  T>(ref  s.GetRawStringData()),
s.Length).Slice(_index, _length);
        }
        else if (_object != null)
        {
            return new Span<T>((T[])_object, _index, _length & RemoveFlagsBitMask);
        }
        else
        {
            return default;
        }
    }
}
```

А точнее, на строчки, обрабатывающие работу со строками. Ведь в них говорится о том, что если мы каким-либо образом сконвертировали `ReadOnlyMemory<T>` в `Memory<T>` (а в двоичном представлении это одно и тоже. Мало того, существует комментарий, предупреждающий что бинарно эти два типа обязаны совпадать, т.к. один из другого получается путем вызова `Unsafe.As`), то мы получаем ~~доступ в тайную комнату~~ возможность менять строки. А это крайне опасный механизм:

```
unsafe void Main()
{
    var str = "Hello!";
    ReadOnlyMemory<char> ronly = str.AsMemory();
```

```
    Memory<char> mem = (Memory<char>)Unsafe.As<ReadOnlyMemory<char>, Memory<char>>(ref
ronly);
    mem.Span[5] = '?';

    Console.WriteLine(str);
}
---
Hello?
```

Который в купе с интернированием строк может дать весьма плачевные последствия.

## Memory<T>.Pin

Второй метод, который вызывает не поддельный интерес - это метод `Pin`:

```
public unsafe MemoryHandle Pin()
{
    if (_index < 0)
    {
        return ((MemoryManager<T>)_object).Pin((_index & RemoveFlagsBitMask));
    }
    else if (typeof(T) == typeof(char) && _object is string s)
    {
        // This case can only happen if a ReadOnlyMemory<char> was created around a
string
        // and then that was cast to a Memory<char> using unsafe / marshaling code.
This needs
        // to work, however, so that code that uses a single Memory<char> field to
store either
        // a readable ReadOnlyMemory<char> or a writable Memory<char> can still be
pinned and
        // used for interop purposes.
        GCHandle handle = GCHandle.Alloc(s, GCHandleType.Pinned);
        void*  pointer  =  Unsafe.Add<T>(Unsafe.AsPointer(ref  s.GetRawStringData()),
_index);
        return new MemoryHandle(pointer, handle);
    }
    else if (_object is T[] array)
    {
        // Array is already pre-pinned
        if (_length < 0)
        {
            void*        pointer        =        Unsafe.Add<T>(Unsafe.AsPointer(ref
array.GetRawSzArrayData()), _index);
            return new MemoryHandle(pointer);
        }
        else
        {
            GCHandle handle = GCHandle.Alloc(array, GCHandleType.Pinned);
            void*        pointer        =        Unsafe.Add<T>(Unsafe.AsPointer(ref
array.GetRawSzArrayData()), _index);
            return new MemoryHandle(pointer, handle);
        }
    }
    return default;
}
```

Который также является очень важным инструментом унификации: ведь вне зависимости от типа данных, на которые ссылается `Memory<T>`, если мы захотим отдать буфер в неуправляемый код, то единственное, что нам надо сделать - вызвать метод `Pin()` и передать указатель, который будет храниться в свойстве полученной структуры в неуправляемый код:

```
void PinSample(Memory<byte> memory)
{
    using(var handle = memory.Pin())
    {
        WinApi.SomeApiMethod(handle.Pointer);
    }
}
```

И в данном коде нет никакой разницы, для чего вызван `Pin()`: для `Memory` над `T[]`, над `string` или же над буфером неуправляемой памяти. Просто для массивов и строк будет создан реальный `GCHandle.Alloc(array, GCHandleType.Pinned)`, а для неуправляемой памяти - просто ничего не произойдет.

## MemoryManager, IMemoryOwner, MemoryPool

Помимо указания полей структуры я хочу дополнительно указать на то, что существует еще два `internal`конструктора типа, работающих на основании еще одной сущности - `MemoryManager`, речь о котором зайдет несколько дальше и что не является чем-то, о чем вы, возможно, только что подумали: менеджером памяти в классическом понимании. Однако, как и `Span`, `Memory` точно также содержит в себе ссылку на объект, по которому будет производиться навигация, а также смещение и размер внутреннего буфера. Также дополнительно стоит отметить, что `Memory` может быть создан оператором `new` только на основании массива плюс методами расширения - на основании строки, массива и `ArraySegment`. Т.е. его создание на основании unmanaged памяти вручную не подразумевается. Однако, как мы видим, существует некий внутренний метод создания этой структуры на основании `MemoryManager`:

**Файл** MemoryManager.cs

```
public abstract class MemoryManager<T> : IMemoryOwner<T>, IPinnable
{
    public abstract MemoryHandle Pin(int elementIndex = 0);
    public abstract void Unpin();

    public virtual Memory<T> Memory => new Memory<T>(this, GetSpan().Length);
    public abstract Span<T> GetSpan();
    protected Memory<T> CreateMemory(int length) => new Memory<T>(this, length);
    protected Memory<T> CreateMemory(int start, int length) => new Memory<T>(this,
start, length);

    void IDisposable.Dispose()
    protected abstract void Dispose(bool disposing);
}
```

Которая инкапсулирует в себе понятие владельца участка памяти. Другими словами, если `Span` - средство работы с памятью, а `Memory` - средство хранения информации о конкретном участке, то `MemoryManager` - средство контроля его жизни, его владелец. Для примера можно взять тип `NativeMemoryManager<T>`, который хоть и написан для тестов, однако неплохо отражает суть понятия "владение":

**Файл** NativeMemoryManager.cs

```
internal sealed class NativeMemoryManager : MemoryManager<byte>
{
    private readonly int _length;
    private IntPtr _ptr;
    private int _retainedCount;
    private bool _disposed;

    public NativeMemoryManager(int length)
    {
        _length = length;
        _ptr = Marshal.AllocHGlobal(length);
    }

    public override void Pin() { ... }

    public override void Unpin()
    {
        lock (this)
        {
            if (_retainedCount > 0)
            {
                _retainedCount--;
                if (_retainedCount == 0)
                {
                    if (_disposed)
                    {
                        Marshal.FreeHGlobal(_ptr);
                        _ptr = IntPtr.Zero;
                    }
                }
            }
```

```
        }
    }

    // Другие методы
}
```

Т.е., другими словами, класс обеспечивает возможность вложенных вызовов метода `Pin()` подсчитывая тем самым образующиеся ссылки из `unsafe` мира.

Еще одной сущностью, тесно связанной с `Memory`, является `MemoryPool`, который обеспечивает пулинг экземпляров `MemoryManager` (а по факту - `IMemoryOwner`):

**Файл** MemoryPool.cs

```
public abstract class MemoryPool<T> : IDisposable
{
    public static MemoryPool<T> Shared => s_shared;

    public abstract IMemoryOwner<T> Rent(int minBufferSize = -1);

    public void Dispose() { ... }
}
```

Который предназначен для выдачи буферов необходимого размера во временное пользование. Арендуемые экземпляры, реализующие интерфейс `IMemoryOwner<T>`, имеют метод `Dispose()`, который возвращает арендованный массив обратно в пул массивов. Причем, по умолчанию вы можете пользоваться общим пулом буферов, который построен на основе `ArrayMemoryPool`:

**Файл** ArrayMemoryPool.cs

```
internal sealed partial class ArrayMemoryPool<T> : MemoryPool<T>
{
    private const int MaximumBufferSize = int.MaxValue;
    public sealed override int MaxBufferSize => MaximumBufferSize;
    public sealed override IMemoryOwner<T> Rent(int minimumBufferSize = -1)
    {
        if (minimumBufferSize == -1)
            minimumBufferSize = 1 + (4095 / Unsafe.SizeOf<T>());
        else if (((uint)minimumBufferSize) > MaximumBufferSize)

ThrowHelper.ThrowArgumentOutOfRangeException(ExceptionArgument.minimumBufferSize);

        return new ArrayMemoryPoolBuffer(minimumBufferSize);
    }
    protected sealed override void Dispose(bool disposing) { }
}
```

И на основании данной архитектуры вырисовывается следующая картина мира:

- Тип данных `Span` необходимо использовать в параметрах методов, если вы подразумеваете либо считывание данных (`ReadOnlySpan`), либо считывание или

запись (`Span`). Но не задачу его сохранения в поле класса для использования в будущем

- Если вам необходимо хранить ссылку на буфер данных из поля класса, необходимо использовать `Memory<T>` или `ReadOnlyMemory<T>` - в зависимости от целей
- `MemoryManager<T>` - это владелец буфера данных (можно не использовать: по необходимости). Необходим, когда, например, встает необходимость подсчитывать вызовы `Pin()`. Или когда необходимо обладать знаниями о том, как освобождать память
- Если `Memory` построен вокруг неуправляемого участка памяти, `Pin()` ничего не сделает. Однако, это унифицирует работу с разными типами буферов: как в случае управляемого так и в случае неуправляемого кода интерфейс взаимодействия будет одинаковым
- Каждый из типов имеет публичные конструкторы. А это значит, что вы можете пользоваться как `Span` напрямую, так и получать его экземпляр из `Memory`. Сам `Memory` вы можете создать как отдельно, так и организовать для него `IMemoryOwner` тип, который будет владеть участком памяти, на который будет ссылаться `Memory`. Частным случаем может являться любой тип, основанный на `MemoryManager`: некоторое локальное владение участком памяти (например, с подсчетом ссылок из `unsafe` мира). Если при этом необходим пуллинг таких буферов (ожидается частый траффик буферов примерно равного размера), можно воспользоваться типом `MemoryPool`.
- Если подразумевается, что вам необходимо работать с `unsafe` кодом, передавая туда некий буфер данных, стоит использовать тип `Memory`: он имеет метод `Pin()`, автоматизирующий фиксацию буфера в куче .NET, если тот был там создан.
- Если же вы имеете некий трафик буферов (например, вы решаете задачу парсинга текста программы или какого-то DSL), стоит воспользоваться типом `MemoryPool`, который можно организовать очень правильным образом, выдавая из пула буферы подходящего размера (например, немного большего

если не нашлось подходящего, но с обрезкой `originalMemory.Slice(requiredSize)` чтобы не фрагментировать пул)

## Производительность

Для того чтобы понять производительность новых типов данных, я решил воспользоваться уже ставшей стандартной библиотекой BenchmarkDotNet:

```
[Config(typeof(MultipleRuntimesConfig))]
public class SpanIndexer
{
    private const int Count = 100;
    private char[] arrayField;
    private ArraySegment<char> segment;
    private string str;

    [GlobalSetup]
    public void Setup()
    {
        str = new string(Enumerable.Repeat('a', Count).ToArray());
        arrayField = str.ToArray();
        segment = new ArraySegment<char>(arrayField);
    }

    [Benchmark(Baseline = true, OperationsPerInvoke = Count)]
    public int ArrayIndexer_Get()
    {
        var tmp = 0;
        for (int index = 0, len = arrayField.Length; index < len; index++)
        {
            tmp = arrayField[index];
        }
        return tmp;
    }

    [Benchmark(OperationsPerInvoke = Count)]
    public void ArrayIndexer_Set()
    {
        for (int index = 0, len = arrayField.Length; index < len; index++)
        {
            arrayField[index] = '0';
        }
    }

    [Benchmark(OperationsPerInvoke = Count)]
    public int ArraySegmentIndexer_Get()
    {
        var tmp = 0;
        var accessor = (IList<char>)segment;
        for (int index = 0, len = accessor.Count; index < len; index++)
        {
            tmp = accessor[index];
        }
        return tmp;
    }
```

```
[Benchmark(OperationsPerInvoke = Count)]
public void ArraySegmentIndexer_Set()
{
    var accessor = (IList<char>)segment;
    for (int index = 0, len = accessor.Count; index < len; index++)
    {
        accessor[index] = '0';
    }
}

[Benchmark(OperationsPerInvoke = Count)]
public int StringIndexer_Get()
{
    var tmp = 0;
    for (int index = 0, len = str.Length; index < len; index++)
    {
        tmp = str[index];
    }

    return tmp;
}

[Benchmark(OperationsPerInvoke = Count)]
public int SpanArrayIndexer_Get()
{
    var span = arrayField.AsSpan();
    var tmp = 0;
    for (int index = 0, len = span.Length; index < len; index++)
    {
        tmp = span[index];
    }
    return tmp;
}

[Benchmark(OperationsPerInvoke = Count)]
public int SpanArraySegmentIndexer_Get()
{
    var span = segment.AsSpan();
    var tmp = 0;
    for (int index = 0, len = span.Length; index < len; index++)
    {
        tmp = span[index];
    }
    return tmp;
}

[Benchmark(OperationsPerInvoke = Count)]
public int SpanStringIndexer_Get()
{
    var span = str.AsSpan();
    var tmp = 0;
    for (int index = 0, len = span.Length; index < len; index++)
    {
        tmp = span[index];
    }
    return tmp;
}

[Benchmark(OperationsPerInvoke = Count)]
public void SpanArrayIndexer_Set()
```

```
    {
        var span = arrayField.AsSpan();
        for (int index = 0, len = span.Length; index < len; index++)
        {
            span[index] = '0';
        }
    }

    [Benchmark(OperationsPerInvoke = Count)]
    public void SpanArraySegmentIndexer_Set()
    {
        var span = segment.AsSpan();
        for (int index = 0, len = span.Length; index < len; index++)
        {
            span[index] = '0';
        }
    }
}

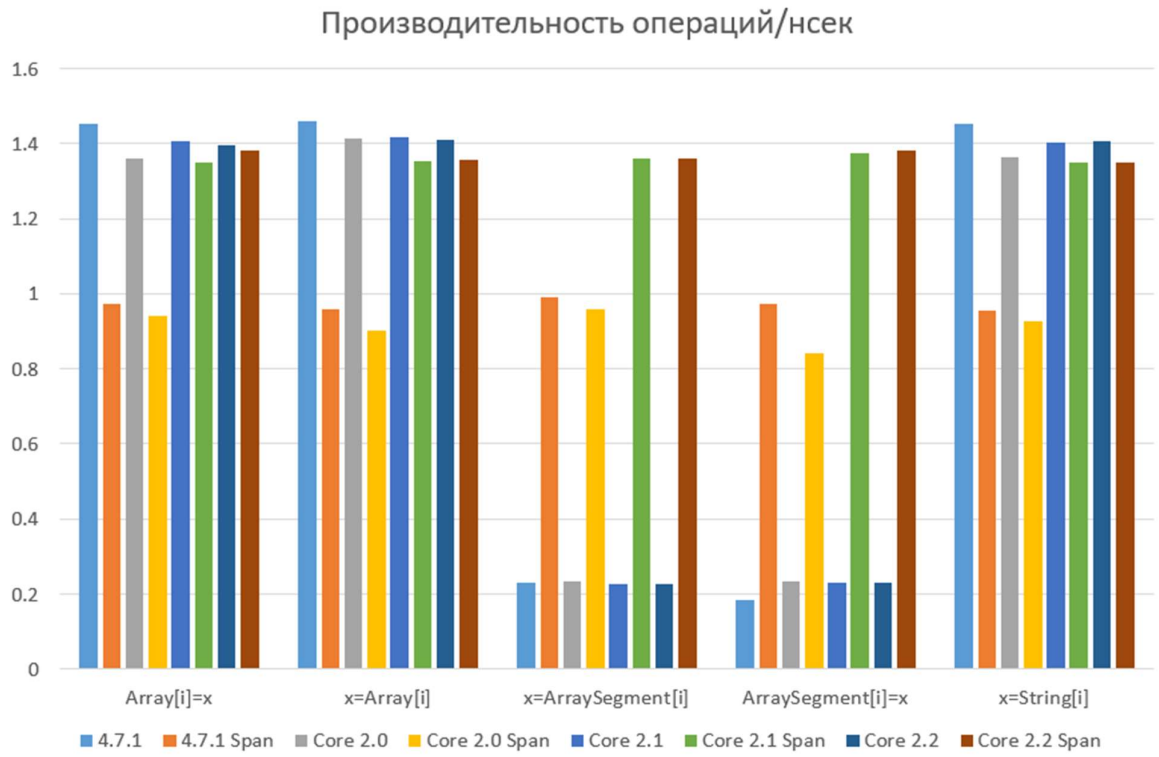
public class MultipleRuntimesConfig : ManualConfig
{
    public MultipleRuntimesConfig()
    {
        Add(Job.Default
            .With(CsProjClassicNetToolchain.Net471) // Span не поддерживается CLR
            .WithId(".NET 4.7.1"));

        Add(Job.Default
            .With(CsProjCoreToolchain.NetCoreApp20) // Span поддерживается CLR
            .WithId(".NET Core 2.0"));

        Add(Job.Default
            .With(CsProjCoreToolchain.NetCoreApp21) // Span поддерживается CLR
            .WithId(".NET Core 2.1"));

        Add(Job.Default
            .With(CsProjCoreToolchain.NetCoreApp22) // Span поддерживается CLR
            .WithId(".NET Core 2.2"));
    }
}
```

После чего изучим результаты:




Смотря на них можно подчерпнуть следующую информацию:

- ArraySegment ужасен. Но его можно сделать не таким ужасным, обернув в `Span`. Производительность вырастет при этом в 7 раз;
- Если рассматривать `.NET Framework 4.7.1` (а для 4.5 это будет аналогичным), то переход на Span заметно просадит работу с буферами данных. Примерно на 30-35%;
- Однако если посмотреть в сторону .NET Core 2.1+, то здесь производительность станет сопоставимой, а с учетом того что Span может работать на части буфера данных, создавая контекст, то вообще быстрее: ведь аналогичным функционалом обладает ArraySegment, который работает крайне медленно.

Отсюда можно сделать простые выводы по использованию этих типов данных:

- Для `.NET Framework 4.5+` и `.NET Core` их использование даст только один плюс: они быстрее их альтернативы в виде ArraySegment - на подмножестве исходного массива;
- Для `.NET Core 2.1+` их использование даст неоспоримое преимущество как перед использованием ArraySegment, так и перед любыми видами ручной реализации `Slice`

- Также, что не даст ни один способ унификации массивов - все три способа максимально производительны.

## Reference Types vs Value Types

Давайте в первую очередь поговорим про Reference Types и Value Types. И если говорить про разницу между ними и про полезность каждого из типов, то первое, о чем я бы упомянул - так это о своих мыслях об их названии. На мой скромный взгляд, если бы в русскоязычном сегменте их назвали ссылочные и значимые типы вместо проговаривания Value Types и Reference Types, то с пониманием разницы между ними все бы встало на свои места.

Очень часто при вопросе что такое ссылочные и значимые типы, люди отвечают, что ссылочные живут в куче, а значимые - в стеке. И это в корне неправильно. Это настолько маленькая часть правды, что правдой не может считаться в принципе

Также, чтобы понимать разницу между ними, давайте и будем изучать их с точки зрения разницы:

- *Значимый тип*: значением является **вся структура целиком**. Для *ссылочного типа* значением является **ссылка** на объект;
- По структуре в памяти: значимые типы содержат только те данные, которые вы указали. Ссылочные также содержат два системных поля. Первое необходимо для хранения `SyncBlockIndex`, второе - для хранения информации о типе: в том числе и о Virtual Methods Table (VMT)
- Однако, ссылочные типы можно наследовать, переопределяя методы. Значимые типы лишены такой возможности;
- Но чтобы выделить ссылочный тип, надо аллоцировать место в куче. Значимый тип *может* работать на стеке, не уходя в кучу, а может стать частью ссылочного типа. Это свойство может значительно повысить производительность для некоторых алгоритмов;

Однако, есть и общие черты:

- Оба подкласса наследуют тип object, а значит - могут выступать как его представители - на полных правах

Рассмотрим каждую особенность в отдельности.

## Копирование

Самую главную и основополагающую разницу между типами можно описать примерно так:

- Любая переменная, поле класса/структуры или же параметр метода, которые принимают ссылочный тип, на самом деле хранят в себе **ссылку** на значение;
- Тогда как любая переменная, поле класса/структуры или же параметр метода, которые принимают значимый тип, на самом деле хранят в себе именно значение. Т.е. всю структуру целиком;

Что это значит для нас? Это в частности значит, что любое присваивание или прокидывание через параметр метода вызовет копирование значения. А поменяв копию, оригинал изменен не будет. При этом если вы меняете поля ссылочного типа, изменения "получают" все, кто имеют ссылку на экземпляр типа. Давайте рассмотрим это на примере:

```
 DateTime dt = DateTime.Now;   // Здесь сначала при вызове метода будет выделено место под переменную DateTime,
                               // но заполнено оно будет нулями. Далее копируется все значение свойства Now в переменную dt
 DateTime dt2 = dt;            // Здесь значение копируется еще раз

 object obj = new object();    // Тут мы создаем объект, выделяя память в Small Object Heap, и размещаем указатель на объект в переменной obj
 object obj2 = obj;             // Тут мы копируем ссылку на этот объект. Т.е. объект - один, а ссылок - две
```

Это свойство рождает ряд двусмысленных на первый взгляд конструкций кода. Одна из них - изменение значений в коллекциях:

```
// Объявим структуру
struct ValueHolder
{
    public int Data;
}

// Создадим массив таких структур и проинициализируем поле Data = 5
var array = new [] { new ValueHolder { Data = 5 } };

// Заберем по индексу структуру и в поле Data выставим 4
array[0].Data = 4;
```

```
// Проверим значение
Console.WriteLine(array[0].Data);
```

В данном коде есть маленькая хитрость. Код выглядит так, будто мы сначала достаем экземпляр структуры, а затем у полученной копии выставляем поле Data в новое значение. А это значит, что при проверке мы снова должны получить `5`. Однако все совсем не так. Все дело в том, что в MSIL есть отдельная инструкция для выставления значения полей структур, находящихся в массивах. Она была введена для повышения производительности. И этот код отработает именно так, как и было задумано его автором: программа выведет в консоль число `4`.

Однако стоит изменить пример так:

```
// Объявим структуру
struct ValueHolder
{
    public int Data;
}

// Создадим список таких структур и проинициализируем поле Data = 5
var list = new List<ValueHolder> { new ValueHolder { Data = 5 } };

// Заберем по индексу структуру и в поле Data выставим 4
list[0].Data = 4;

// Проверим значение
Console.WriteLine(list[0].Data);
```

Так у нас ничего даже и не скомпилируется. А все потому, что когда вы пишете `list[0].Data = 4`, то сначала вы получаете именно копию структуры. Вы ведь на самом деле вызываете метод экземпляра типа List, который скрывается за доступом по индексу. И который в свою очередь забирает копию структуры из внутреннего массива (List хранит данные в массивах), которая возвращается из метода доступа по индексу - вам. После чего вы пытаетесь модифицировать копию, которая далее нигде не используется. Это - не то чтобы ошибка, но абсолютно бессмысленный код. А компилятор, зная, что люди путаются с Value Types, запрещает такое поведение. Поэтому пример должен быть переписан таким образом:

```
// Объявим структуру
struct ValueHolder
{
    public int Data;
}
```

```
// Создадим список таких структур и проинициализируем поле Data = 5
var list = new List<ValueHolder> { new ValueHolder { Data = 5 } };

// Заберем по индексу структуру и в поле Data выставим 4, после чего сохраним обратно
var copy = list[0];
copy.Data = 4;
list[0] = copy;

// Проверим значение
Console.WriteLine(list[0].Data);
```

Несмотря на кажущееся многословие, он корректен. Когда программа отработает, в консоль выведется число `4`.

Вторым примером я хочу показать вам, что вообще понимается под "значением структуры является вся структура целиком"

```
// Вариант 1
struct PersonInfo
{
    pubilc int Height;
    pubilc int Width;
    pubilc int HairColor;
}

int x = 5;
PersonInfo person;
int y = 6;

// Вариант 2
int x = 5;
int Height;
int Width;
int HairColor;
int y = 6;
```

Фактически, по расположению данных в памяти оба примера идентичны. Потому как значением структуры является вся структура в целом. Куда она попала, там под себя память и определила.

```
// Вариант 1
struct PersonInfo
{
    pubilc int Height;
    pubilc int Width;
    pubilc int HairColor;
}

class Employee
{
    public int x;
    public PersonInfo person;
    public int y;
}

// Вариант 2
class Employee
```

```
{
    public int x;
    pubilc int Height;
    pubilc int Width;
    pubilc int HairColor;
    public int y;
}
```

Эти примеры также идентичны с точки зрения положения элементов в памяти, т.к. структура просто встает туда, где ее определили: среди полей класса. Я не утверждаю, что это абсолютно идентично: ведь в структуре вы можете оперировать над ее полями при помощи методов структуры.

Если говорить про ссылочные типы, то, понятное дело, для них все обстоит иначе. Сам экземпляр находится в недосягаемом Small Object Heap (SOH) или же в Large Object Heap (LOH), а в поле класса запишется лишь значение указателя на экземпляр: 32-х или 64-разрядное число.

Последний пример, надеюсь, вас не запутает. Но мне хотелось поставить точку в этом вопросе.

```
// Вариант 1
struct PersonInfo
{
    pubilc int Height;
    pubilc int Width;
    pubilc int HairColor;
}

void Method(int x, PersonInfo person, int y);

// Вариант 2
void Method(int x, int HairColor, int Width, int Height, int y);
```

Вы меня поняли совершенно корректно: с точки зрения работы с памятью оба варианта будут работать одинаково (но не архитектурной! это вам не замена переменного числа аргументов!). Почему изменился порядок? Потому что параметры метода объявляются друг за другом и в этом порядке складываются в стек потока. Однако стек растет от старших адресов к младшим, а это значит, что порядок складывания по очереди будет отличаться от порядка складывания структуры целиком.

## Переопределяемые методы и наследование

Вторая глобальная разница между ними - это отсутствие таблицы виртуальных методов в структурах. Это означает что:

- В структурах нельзя описать `virtual` методы, а также - переопределять их;
- Структуры в принципе нельзя наследовать друг от друга. Единственный способ сделать эмуляцию наследования - расположить структуру базового типа первым полем. Тогда по смещениям они будут совпадать, поля "унаследованной" структуры будут располагаться после полей "базовой", и логически вы сделаете наследование;
- Структуры в отличии от классов можно передавать в unmanaged код. Я имею ввиду именно значение. Информация о методах, естественно, будет утеряна. Ведь структура - это просто отрезок памяти, заполненный данными без информации о типе. А это значит, что ее можно без изменений отдавать в unmanaged методы, написанные, например, на C++.

Отсутствие таблицы виртуальных методов хоть и отнимает у структур часть "магии", которую вносит понятие наследования, но и наделяет рядом преимуществ. Первое и самое главное уже было оговорено: мы можем легко и просто отдать во внешний мир (за пределы .NET Framework) экземпляр такой структуры. Это ведь просто участок памяти! Либо мы можем принять из unmanaged кода некий участок памяти и сделать приведение типа к нашей структуре, чтобы сделать более удобный доступ к ее полям. С классами такое поведение не пройдет: у классов существует два поля, которые никому не доступны: это SyncBlockIndex и адрес таблицы виртуальных методов. Если эти два поля уйдут в unmanaged код, это станет очень опасным. Ведь с любой таблицей виртуальных методов можно умеючи достучаться до любого типа и поменять его, осуществив атаку на приложение.

Давайте докажем, что это просто участок памяти без какой-либо дополнительной логики:

```
unsafe void Main()
{
    int secret = 666;
    HeightHolder hh;
    hh.Height = 5;
```

```
    WidthHolder wh;
    unsafe
    {
        // Если бы у структур была информация о типе, это приведение не смогло бы
работать:
        // CLR перед приведением типа проверила бы иерархию и, не найдя в ней
WidthHolder,
        // выбросила бы InvalidCastException. Но поскольку структура - просто участок
памяти,
        // в unsafe мире никто не мешает вам интерпретировать его какой угодно
структурой
        wh = *(WidthHolder*)&hh;
    }
    Console.WriteLine("Width: " + wh.Width);
    Console.WriteLine("Secret: " + wh.Secret);
}

struct WidthHolder
{
    public int Width;
    public int Secret;
}

struct HeightHolder
{
    public int Height;
}
```

В данном примере мы осуществляем недопустимую с точки зрения строгой типизации операцию: мы приводим один тип к несовместимому другому, который содержит одно лишнее поле. В методе `Main` мы вводим дополнительную переменную, значение которой по-идее секретно и не должно быть считано. Однако это не так. Пример уверенно выводит на экран значение переменной метода Main(), которая не находится ни в одной из структур. Тут на вашем лице должна расплыться улыбка, а в голове промелькнуть фраза "ну ни черта себе дыра в безопасности!!!"... Но на самом деле все не так очевидно. Обезопасить свой код от вызываемого unmanaged практически невозможно. Все дело в первую очередь - в структуре стека потока, о котором мы поговорим чуть позже, и по которому можно легко уйти в вызываемый код, и похимичить с локальными переменными. Защита от такого рода атак строится другими путями. Например, на рандомизации размера кадра стека или на стирании информации о регистре EBP - для усложнения восстановления стекового кадра. Но не будем слишком углубляться: это тема отдельного разговора. Единственное, о чем стоит упомянуть в рамках этого примера, это почему же при том, что переменная `secret`находится **перед** определением переменной `hh`, а в

структуре `WidthHolder` - **после** (т.е. по сути визуально - в разных местах), ее значение прекрасно считалось. А все потому, что стек растет не слева направо, а наоборот - справа налево. Т.е. переменные, объявленные первыми, будут находиться по более старшим адресам, а те, кто объявлены позднее - по более ранним.

## Поведение при вызове экземплярных методов

Оба типа данных обладают еще одной интересной особенностью, которая не лежит на поверхности, и которая может пролить еще немного света в строение обоих типов. И эта особенность связана с вызовом экземплярных методов.

```
// Пример с ссылочным типом
class FooClass
{
    private int x;

    public void ChangeTo(int val)
    {
        x = val;
    }
}

// Пример с значимым типом
struct FooStruct
{
    private int x;

    public void ChangeTo(int val)
    {
        x = val;
    }
}

FooClass klass = new FooClass();
FooStruct strukt = new FooStruct();

klass.ChangeTo(10);
strukt.ChangeTo(10);
```

Если рассуждать логически, то можно легко и просто прийти к выводу, что тело у метода компилируется одно. Т.е. нет такого, что у каждого экземпляра типа компилируется свой набор методов, который при этом совершенно идентичен набору методов других экземпляров. Однако, вызванный метод прекрасно знает для какого экземпляра он вызван. Это достигается тем, что первым параметром передается ссылка на экземпляр типа. Можно наш пример легко переписать, и это

будет совершенно идентично тому, что было написано выше (я намеренно не привожу пример с виртуальными методами. У них все по-другому):

```
// Пример с ссылочным типом
class FooClass
{
    public int x;
}

// Пример с значимым типом
struct FooStruct
{
    public int x;
}

public void ChangeTo(FooClass klass, int val)
{
    klass.x = val;
}

public void ChangeTo(ref FooStruct strukt, int val)
{
    strukt.x = val;
}

FooClass klass = new FooClass();
FooStruct strukt = new FooStruct();

ChangeTo(klass, 10);
ChangeTo(ref strukt, 10);
```

Стоит пояснить, почему я использовал ключевое слово `ref`. Если бы я его не использовал, то получилась бы ситуация, в которой я получил параметром метода **копию** структуры вместо оригинала, поменял ее, а оригинал бы остался неизменным. Мне бы пришлось возвращать измененную копию из метода вызывающей стороне (еще одно копирование), а вызывающая сторона сохранила бы это значение обратно в переменной (еще одно копирование). Вместо этого в экземплярный метод отдается указатель на структуру, по которому она и меняется. Сразу оригинал. Заметьте, что передача по указателю никак не влияет на производительность, т.к. любые операции на уровне процессора итак происходят по указателям. Т.е. ref - это из мира C#, не более того.

## Возможность указать положение элементов

Еще одной возможностью обоих классов типов является возможность точно указать, по какому смещению относительно начала структуры в памяти располагается то или иное поле. Это введено по нескольким причинам:

- для работы с внешними API, которые располагаются в unmanaged world, чтобы не "отбивать" до нужного поля неиспользуемыми полями
- чтобы, например, приказать компилятору расположить некоторое поле точно в самом начале типа (`[FieldOffset(0)]`). Тогда это ускорит работу с ним. А если это поле очень часто используется, то на этом можно неплохо сэкономить. Отмечу только одну важную деталь. Указанное справедливо только для значимых типов. Ведь в ссылочных по нулевому смещению располагается адрес таблицы виртуальных методов, который занимает 1 процессорное слово. Т.е. даже если вы обращаетесь к самому первому полю класса, обращение все равно будет идти по более сложной адресации (адрес + смещение). Кстати это сделано не просто так: самым часто-используемым полем класса является именно адрес таблицы виртуальных методов, т.к. именно через нее виртуальные методы и вызываются;
- вы можете установить несколько полей по одному адресу. Тогда одно и то же значение может быть интерпретировано как различные типы данных. В C++ такой тип данных называется `union`;
- также вы можете ничего не объявлять: компилятор будет размещать поля так, как ему покажется оптимальным. Т.е. конечный порядок полей может оказаться другим;

**Общие положения**

- **Auto**: Среда выполнения автоматически выбирает расположение и упаковку для всех полей класса или структуры. Структуры, определенные с помощью члена этого перечисления, не могут быть предоставлены за пределами управляемого кода. Попытка это сделать приводит к возникновению исключения;

- **Explicit**: Программист явным образом контролирует точное положение каждого поля объекта. Каждое поле должно использовать FieldOffsetAttribute для указания его точного расположения;
- **Sequential**: Члены объекта располагаются последовательно в порядке, указанном при проектировании типа. Также они располагаются в соответствии с указанным StructLayoutAttribute.Pack значением шага упаковки.

### Использование FieldOffset для пропуска неиспользуемых областей структуры

Тут, конечно же может возникнуть вопрос, почему вообще могут возникнуть поля, которые не используются вообще. Структуры, идущие из unmanaged мира, могут содержать резервные поля, которые могут быть использованы в будущих версиях библиотеки. Если в мире C/C++ принято отбивать такие пропуски путем добавления полей `reserved1, reserved2, ..`, то в .NET мы имеем прекрасную возможность просто задать смещение к началу поля при помощи атрибута `FieldOffsetAttribute` и `[StructLayout(LayoutKind.Explicit)]`:

```
[StructLayout(LayoutKind.Explicit)]
public struct SYSTEM_INFO
{
    [FieldOffset(0)] public ulong OemId;
    // 92 байта - резерв
    [FieldOffset(100)] public ulong PageSize;
    [FieldOffset(108)] public ulong ActiveProcessorMask;
    [FieldOffset(116)] public ulong NumberOfProcessors;
    [FieldOffset(124)] public ulong ProcessorType;
}
```

Прошу заметить, что пропуск - это тоже занятое, но не используемое пространство. Размер структуры будет равен `132` байта, а не `40`, как может показаться изначально.

### Union

При помощи `FieldOffsetAttribute` вы можете эмулировать такой тип из мира C/C++ как `union`. `union` – это специальный тип, который позволяет обращаться к одним и тем же данным как к разнотипным сущностям. Давайте посмотрим на примере его эмуляцию:

```
// Если прочитать RGBA.Value, мы прочитаем Int32 значение, которое будет аккумуляцией
всех остальных полей.
```

```
// Однако если мы попробуем прочитать RGBA.R, RGBA.G, RGBA.B, RGBA.Alpha, то мы прочитаем отдельные компоненты Int32 числа
[StructLayout(LayoutKind.Explicit)]
public struct RGBA
{
    [FieldOffset(0)] public uint Value;
    [FieldOffset(0)] public byte R;
    [FieldOffset(1)] public byte G;
    [FieldOffset(2)] public byte B;
    [FieldOffset(3)] public byte Alpha;
}
```

Здесь вы могли бы задуматься и сказать, что такое поведение возможно только для значимых типов, однако это не так. Как бы странно это ни звучало, но такое поведение можно воспроизвести и для ссылочных типов, перекрыв по одному адресу два ссылочных типа или же ссылочный со значимым:

```
class Program
{
    public static void Main()
    {
        Union x = new Union();
        x.Reference.Value = "Hello!";
        Console.WriteLine(x.Value.Value);
    }

    [StructLayout(LayoutKind.Explicit)]
    public class Union
    {
        public Union()
        {
            Value = new Holder<IntPtr>();
            Reference = new Holder<object>();
        }

        [FieldOffset(0)]
        public Holder<IntPtr> Value;

        [FieldOffset(0)]
        public Holder<object> Reference;
    }

    public class Holder<T>
    {
        public T Value;
    }
}
```

Заметьте что я намеренно перекрыл через Generic тип: при обычном перекрытии в момент загрузки данного типа в домен приложения будет сгенерировано исключение `TypeLoadException`. На самом деле это только внешне выглядит как брешь в безопасности приложения (особенно со стороны "плагинов" к приложению),

однако если мы попробуем запустить этот код из-под защищенного домена, то мы получим все тот же самый `TypeLoadException`.

## Разница в аллокации

Еще одним важным свойством, кардинально различающимся для обоих типов, является выделение памяти под объект/структуру. Все дело в том, что для того чтобы выделить память под объект, CLR обязана для начала ответить себе на ряд вопросов. Первый - какого размера объект? Меньше он или больше 85К байт? Если меньше, то является ли количество оставшегося места в Small Objects Heap достаточным, чтобы разместить объект? Если нет, запускается Garbage Collection, который для своей работы по сути должен сначала обойти граф объектов, а потом сжать их, переместив на освободившееся место. Если и после этой операции нет места в SOH (например, ничего не было освобождено), то инициируется процесс выделения дополнительных страниц виртуальной памяти, чтобы нарастить размер Small Objects Heap. И только после того как все срастется, выделяется место под объект, а выделенный участок памяти очищается от мусора (обнуляется), размечается SyncBlockIndex и VirtualMethodsTable, после чего ссылка на объект возвращается пользователю.

Если же выделяемый объект имеет размеры, превышающие 85K, то мы имеем дело с Large Objects Heap. Это, например, случай огромных строк и массивов. В этом случае мы должны найти максимально подходящий кусок памяти из списка освобожденных и, если таковых нет, выделить новый участок. Эти процедуры по умолчанию не быстрые, но мы предполагаем, что с объектами такого размера мы будем работать особенно осторожно и они вне контекста данной беседы

Т.е. для RefTypes мы имеем несколько случаев:

- Размер RefType < 85K, место в SOH есть: выделение памяти идет достаточно быстро;
- Размер RefType < 85K, место в SOH заканчивается: выделение памяти идет очень медленно;

- Размер RefType > 85K, выделение памяти идет относительно медленно. А с учетом того, что данные операции редки и не могут ввиду своих размеров конкурировать с ValTypes, нас это сейчас не сильно волнует.

Каков же алгоритм выделения памяти под Value Types? А нет его. Выделение памяти под Value Types не стоит абсолютно ничего. Единственное, что происходит при его "выделении" - это обнуление полей. Давайте разберемся, почему так происходит:

1. В случае объявления переменной в теле метода время на выделение места под структуру можно считать около нулевым. Ведь время на выделение места под локальные переменные почти не зависит от их количества;
2. В случае размещения ValTypes в качестве полей RefTypes просто увеличит их размер. Значимый тип размещается целиком, становясь его частью;
3. Если ValTypes передаются как параметры метода - тут, как и в случае копирования, возникнет некоторая разница - в зависимости от размера и положения параметра.

Но в любом случае это не дольше копирования из одной переменной в другую.

## Особенности выбора между class/struct

Давайте подумаем об особенностях обоих типов, об их достоинствах и недостатках и решим, где ими лучше пользоваться. Тут, конечно же, стоит вспомнить классиков, дающих утверждение, что выбор в сторону значимых типов стоит сделать, если у нас тип не планирует быть наследуемым, он не станет меняться в течении своей жизни, а его размер не превышает 16 байт. Но не все так очевидно. Чтобы сделать полноценное сравнение, нам необходимо задуматься о выборе типа с разных сторон, мысленно продумав сценарии его будущего использования. Предлагаю разделить критерии выбора на три группы:

- с точки зрения архитектуры системы типов, в которой ваш тип будет взаимодействовать;

- с точки зрения подхода вас как системного программиста: каков выбор будет оптимальным с точки зрения производительности;
- по-другому просто невозможно.

Каждая сущность, которая проектируется вами, должна в полной мере отражать ее назначение. И это касается не только её названия или интерфейса взаимодействия (методы, свойства), но даже выбор между значимым и ссылочным типом может быть сделан из архитектурных соображений. Давайте порассуждаем, почему с точки зрения архитектуры системы типов может быть выбрана структура, а не класс:

- Если наш проектируемый тип будет обладать инвариантностью по отношению к смысловой нагрузке своего состояния, то это будет значить, что его состояние полностью отражает некоторый процесс или является значением чего-либо. Другими словами, экземпляр типа полностью константен и не может быть изменен по своей сути. Мы можем создать на основе этой константы другой экземпляр типа, указав некоторое смещение, либо создать с нуля, указав его свойства. Но изменять его мы не имеем права. Я прошу заметить, что не подразумеваю структуру неизменяемым типом. Вы можете менять поля, как хотите. Мало того вы можете отдать ссылку на структуру в метод через `ref` параметр и получить измененные поля по выходу из метода. Однако, я про смысл с точки зрения архитектуры. Поясню на примерах:
  - DateTime - это структура, которая инкапсулирует в себе понятие момента времени. Она хранит эти данные в виде `uint`, однако предоставляет доступ к отдельным характеристикам момента времени. Например: год, месяц, день, час, минуты, секунды, миллисекунды и даже процессорные тики. Однако, исходя из того что она инкапсулирует, она не может быть изменяемой по своей природе. Мы не можем изменить конкретный момент времени, чтобы он стал другим. Я не могу прожить следующую минуту своей жизни в лучший день рождения своего детства. Время неизменно. Именно поэтому выбор для типа данных может стать либо

класс с readonly интерфейсом взаимодействия, который на каждое изменение свойств отдает новый экземпляр, либо структура, которая несмотря на возможность изменения полей своих экземпляров делать этого не должна: описание момента времени является *значением*. Как число. Вы же не можете залезть в структуру числа и поменять его? Если вы хотите получить другой момент времени, который является смещением относительно оригинального на один день, вы просто получаете новый экземпляр структуры;

- KeyValuePair<TKey, TValue> - это структура, инкапсулирующая в себе понятие связной пары ключ-значение. Замечу, что эта структура используется только для выдачи пользователю при перечислении содержимого словаря. Почему выбрана структура с точки зрения архитектуры? Ответ прост: потому что в рамках Dictionary ключ и значение неразделимые понятия. Да, внутри все устроено иначе. Внутри мы имеем сложную структуру, где ключ лежит отдельно от значения. Однако для внешнего пользователя, с точки зрения интерфейса взаимодействия и смысла самой структуры данных, пара ключ-значение является неразделимым понятием. Является *значением* целиком. Если мы по этому ключу расположили другое значение, это значит, что изменилась вся пара. Для внешнего наблюдателя нет отдельно ключей, а отдельно - значений, они являются единым целым. Именно поэтому структура в данном случае - идеальный вариант.
- Если наш проектируемый тип является неотъемлемой частью внешнего типа. Но при этом он структурно неотъемлем. Т.е. было бы некорректным сказать, что внешний тип ссылается на экземпляр инкапсулируемого, но совершенно корректно - что инкапсулируемый является полноправной частью внешнего вместе со всеми своими свойствами. Как правило это используется при проектировании структур, которые являются частью другой структуры.

- Как, например, если взять структуру заголовка файла, было бы нечестно дать ссылку из одного файла в другой. Мол, заголовок находится в файле `header.txt`. Это было бы уместно при вставке документа в некий другой, но не вживанием файла, а по относительной ссылке на файловой системе. Хороший пример - файл ярлыка ОС Windows. Однако если мы говорим о заголовке файла (например, о заголовке JPEG файла, в котором указаны размер изображения, методика сжатия, параметры съемки, координаты GPS и прочая метаинформация), то при проектировании типов, которые будут использованы для парсинга заголовка, будет крайне полезно использовать структуры. Ведь, описав все заголовки в структурах, вы получите в памяти абсолютно такое же положение всех полей как в файле. И через простое unsafe преобразование `*(Header *)readedBuffer` без каких-либо десериализаций - полностью заполненные структуры данных.
- При этом заметьте, ни один из примеров не обладает свойством наследования поведения чего-либо. Мало того все эти примеры также показывают, что нет абсолютно никакого смысла наследовать поведение этих сущностей. Они полностью самодостаточны, как единицы чего-либо.

Если же мы взглянем на проблематику с точки зрения эффективности работы кода, то перед нами выбор предстанет с другой стороны:

1. Структуры необходимо выбирать, если необходимо забрать из неуправляемого кода какие-то структурированные данные. Либо отдать unsafe методу структуру данных. Ссылочный тип для этого совсем не подойдет;
2. Если тип будет часто использоваться для передачи данных в вызовах методов (пусть в качестве возвращаемых значений или как параметр метода), но при этом нет никакой необходимости ссылаться на одно значение с разных мест,

то ваш выбор - структура. Как пример я могу привести кортежи. Если метод через кортеж возвращает вам несколько значений, это значит, что возвращать он будет ValueTuple, который объявлен как структура. Т.е. при возврате метод не будет выделять память в куче, а использовать он будет стек потока, выделение памяти в котором не стоит вам абсолютно ничего;

3. Если вы проектируете систему, которая создает некий больший трафик экземпляров проектируемого типа. При этом сами экземпляры имеют достаточно малый размер, а время жизни экземпляров очень короткое, то использование ссылочных типов приведет либо к использованию пула объектов, либо, если без пула, к неконтролируемому замусориванию кучи. При этом часть объектов перейдет в старшие поколения, чем вызовет проседание на GC. Использование значимых типов в таких местах (если это возможно) даст прирост производительности просто потому, что в SOH ничего не уйдет, а это разгрузит GC, и алгоритм отработает быстрее;

Совмещая все выше сказанное, могу предложить некоторые советы и замечания в использовании структур:

1. При выборе коллекций стоит избегать больших массивов, внутри которых находятся большие структуры. Это касается и тех структур данных, которые на массивах основаны (а их - большинство). Это может привести к уходу в Large Objects Heap и его фрагментации. Мало подсчитать, что, если у вашей структуры 4 поля типа byte, значит займет она 4 байта. Вовсе нет. Надо понимать, что для 32-разрядных систем каждое поле структуры будет выровнено по 4 байтам (адрес каждого поля должен делиться на 4 без остатка), а на 64-разрядных системах - по 8 байтам. Т.е. размер массива должен зависеть от размера структуры и от платформы, на которой запущено приложение. В нашем примере с 4 байтами - 85К / (от 4 до 8 байт на поле * количество полей = 4) минус размер заголовка массива: примерно до 2600 элементов на массив в зависимости от платформы (а брать понятное дело

надо в меньшую сторону). Всего-то! Не так и много! А ведь могло показаться, что магическая константа в 20,000 элементов вполне могла подойти!

2. Также стоит отдавать себе отчет, что если вы используете структуру, которая имеет некоторый достаточно большой размер, как источник данных, и размещаете ее в некотором классе как поле, и при этом, например, одна и та же копия растиражирована на тысячу экземпляров (просто потому, что вам удобно держать все под рукой), то вы тем самым увеличиваете каждый экземпляр класса на размер структуры, что в конечном счете приведет к распуханию 0-го поколения и уходу в поколение 1 или даже 2. При этом если на самом деле экземпляры класса короткоживущие, и вы рассчитываете на то, что они будут собраны GC в нулевом поколении - за 1 мс, то будете сильно разочарованы тем, что они на самом деле успели попасть в первое или даже во второе поколение. А какая, собственно, разница? Разница в том, что если поколение 0 собирается за 1 мс, то первое и второе - очень медленно, что приведет к проседаниям на пустом месте;
3. По примерно той же причине стоит избегать проброса больших структур через цепочку вызовов методов. Потому как если все начнет друг друга вызывать, то такие вызовы займут намного больше места в стеке, подводя жизнь вашего приложения к смерти через `StackOverflowException`. Вторая причина - производительность. Чем больше копирований, тем медленнее все работает;

Потому в целом выбор между типами данных - достаточно нетривиальный процесс. Зачастую это может относиться к преждевременной оптимизации, чего делать не рекомендуется. Однако, если вы знаете, что ваша ситуация попадает под выше изложенные принципы, то можете спокойно делать выбор в сторону значимого типа.

## Базовый тип - Object и возможность реализации интерфейсов. Boxing.

Мы с вами прошли, как может показаться и огонь, и воду и можем пройти любое собеседование. Возможно даже в команду .NET CLR. Но давайте не будем спешить набирать microsoft.com и искать там раздел вакансий: успеем. Давайте лучше ответим на такой вопрос. Если значимые типы не содержат ни ссылки на SyncBlockIndex, ни указателя на таблицу виртуальных методов... То, простите, как они наследуют тип `object`? Ведь по всем канонам любой тип наследует именно его. Ответ на этот вопрос к сожалению не будет вмещен в одно предложение, но даст такое понимание о нашей системе типов, что последние кусочки пазла наконец встанут на свои места.

Итак, давайте еще раз вспомним про размещение значимых типов в памяти. Везде, где бы они не находились, они вживляются в то место, где находятся. Они становятся его частью. В отличии от ссылочных типов, для которых закон твердит быть в куче малых или больших объектов, а в место установки значения - всегда ставить ссылку на место в куче, где расположился наш объект.

Так вот если задуматься, то у любого значимого типа есть методы `ToString`, `Equals` и `GetHashCode`, которые являются виртуальными, переопределяемыми, но нам не дают наследовать значимые типы, переопределяя методы. Почему? Потому что если значимые типы сделать с переопределяемыми методами, то им понадобится таблица виртуальных методов, через которую будет осуществляться роутинг вызовов. А это в свою очередь повлечет за собой проблемы проброса структур в unmanaged мир: туда уйдут лишние поля. В итоге получается, что описание методов значимых типов где-то лежат, но к ним нет прямого доступа через таблицу виртуальных методов.

Это наводит на мысль что отсутствие наследования искусственно:

- Наследование от object есть, хоть и не прямое;
- В базовом типе есть ToString, Equals и GetHashCode, которые по-своему работают в значимых типах: у этих методов свое поведение в каждом из них. А значит, что методы переопределены относительно object;

- более того, если вы сделаете приведение типа в `object`, вы все еще можете на полных правах вызывать ToString, Equals и GetHashCode.
- При вызове экземплярного метода над значимым типом не происходит копирования в метод. Т.е. вызов экземплярного метода аналогичен вызову статического метода: `Method(ref structInstance, newInternalFieldValue)`. А это ведь по сути вызов с передачей `this` за одним исключением: JIT должен собрать тело метода так, чтобы не делать дополнительного смещения на поля структуры, перепрыгивая через указатель на таблицу виртуальных методов, которой в самой структуре нет. *Для значимых типов она находится в другом месте*.

Т.е. в некотором смысле нас не то чтобы обманывают, но недоговаривают: типы сильно отличаются поведенчески, но на уровне реализации в CLR разница между ними не столь существенна. Но об этом немного позже.

Если мы напишем следующую строчку в нашей программе:

```
var obj = (object)10;
```

То мы перестанем иметь дело с числом `10`. Произойдет так называемый boxing: упаковка. Т.е. мы начнем иметь возможность работать с ним через базовый класс. А если мы получили такие возможности, это значит, что нам стала доступна VMT, через которую можно спокойно вызывать виртуальные методы ToString(), Equals и GetHashCode. Причем поскольку оригинальное значение у нас может храниться где угодно: хоть на стеке, хоть как поле класса, а приводя к типу `object` мы получаем возможность хранить ссылку на это число веки вечные, то в реальности boxing создает копию значимого типа, а не делает указатель на оригинал. Т.е. когда происходит boxing, то:

- CLR выделяет место в куче под структуру + SyncBlockIndex + VMT значимого типа (чтобы иметь возможность вызвать ToString, GetHashCode, Equals);
- копирует туда экземпляр значимого типа.

Дамы и господа. В приличном обществе такое не принято говорить, но мы получили ссылочный вариант значимого типа. Я повторю еще раз: совершив boxing структура

получила **абсолютно такой же набор системных полей, что и ссылочный тип**, став полноценным ссылочным типом. Структура стала классом. Давайте назовем это явление Кульбит Дотнетского. Мне кажется, это название будет достойным такого хитрого поворота дел.

Кстати, чтобы вы поверили в честность моих слов, достаточно разобраться, что происходит если вы используете структуру, которая реализует некий интерфейс - по этому самому интерфейсу.

```
struct Foo : IBoo
{
    int x;
    void Boo()
    {
        x = 666;
    }
}

IBoo boo = new Foo();

boo.Boo();
```

Итак, когда создается экземпляр Foo, то его значение по сути находится на стеке. После чего мы кладем эту переменную в переменную интерфейсного типа. Структуру - в переменную ссылочного типа. Происходит `boxing`. Хорошо. На выходе мы получили тип `object`. Но переменная у нас - интерфейсного типа. А это значит, что необходимо преобразование типа. Т.е. вызов, скорее, происходит как-то так:

```
IBoo boo = (IBoo)(box_to_object)new Foo();
boo.Boo();
```

Т.е. написание такого кода - это крайне не эффективно. Мало того что вы будете менять копию вместо оригинала:

```
void Main()
{
    var foo = new Foo();
    foo.a = 1;
    Console.WriteLite(foo.a);  // -> 1

    IBoo boo = foo;
    boo.Boo();                 // выглядит как изменение foo.a на 10
    Console.WriteLite(foo.a);  // -> 1
}

struct Foo : IBoo
{
    public int a;
    public void Boo()
```

```
    {
        a = 10;
    }
}

interface IBoo
{
    void Boo();
}
```

Выглядит как обман дважды. Первый раз - глядя на код мы не обязаны знать, с чем имеем дело в *чужом* коде, и видим ниже приведение к интерфейсу `IBoo`. Что фактически гарантированно наводит нас на мысль, что Foo - класс, а не структура. Далее - полное отсутствие визуального разделения на структуры и классы дает полное ощущение, что результаты модификации по интерфейсу обязаны попасть в foo, чего не происходит потому, что boo - копия foo. Что фактически вводит нас в заблуждение. На мой взгляд, такой код стоит снабжать комментариями, чтоб внешний разработчик смог бы в нем правильно разобраться.

Второе наблюдение, связанное с нашими более ранними рассуждениями, а именно с тем, что мы можем сделать приведение типа из `object` в `IBoo`. Это - еще одно доказательство, что boxed значимый тип не что-то особенное, а на самом деле ссылочный вариант значимого типа. Либо если посмотреть с другого угла - все типы в системе типов являются ссылочными. Просто со структурами мы можем работать как со значимыми, "отгружая" их значение целиком. Как бы сказали в мире C++, разыменовывая указатель на объект.

Но вы можете возразить: дескать если бы все было именно так, как я говорю, то можно было бы написать как-то так:

```
var referenceToInteger = (IInt32)10;
```

И мы получили бы не просто `object`, а типизированную ссылку на упакованный значимый тип. Но тогда бы это разрушило всю идею значимых типов, друзья. А основная идея - это целостность их значения, позволяющее делать отличные оптимизации, основываясь на их свойствах. Так не будем сидеть сложа руки! Давайте разрушим эту идею!

```
public sealed class Boxed<T>
{
    public T Value;
```

```
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public override bool Equals(object obj)
    {
        return Value.Equals(obj);
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public override string ToString()
    {
        return Value.ToString();
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public override int GetHashCode()
    {
        return Value.GetHashCode();
    }
}
```

Что мы только что получили? Мы получили абсолютно полный аналог боксинга. Но теперь у нас есть возможность менять его содержимое путем вызова его экземплярных методов. И эти изменения получат все, у кого будет ссылка на эту структуру данных.

```
var typedBoxing = new Boxed<int> { Value = 10 };
var pureBoxing = (object)10;
```

Первый вариант, согласитесь, выглядит несколько неуверенно. Вместо привычного приведения типа мы городим не пойми что. То ли дело вторая строчка. Лаконична как японский стих. Однако они на самом деле почти полностью идентичны. Разница состоит только в том, что во время обычной упаковки после выделения памяти в куче не происходит очистки памяти нулями: память сразу занимается необходимой структурой. Тогда как в первом варианте очистка есть. Только из-за этого наш вариант медленнее обычной упаковки на 10%.

Зато теперь мы можем вызывать у нашего упакованного значения какие-то методы:

```
struct Foo
{
    public int x;

    public void ChangeTo(int newx)
    {
        x = newx;
    }
}

var boxed = new Boxed<Foo> { Value = new Foo { x = 5 } };
boxed.Value.ChangeTo(10);
var unboxed = boxed.Value;
```

Мы получили новый инструмент, но пока не знаем что с ним делать. Давайте добьемся ответа рассуждениями:

- Наш тип `Boxed<T>` по сути осуществляет все то же самое, что и обычный: выделяет память в куче, отдает туда значение и позволяет его забрать, выполнив своеобразный `unbox`;
- Точно также, если потерять ссылку на упакованную структуру, GC её соберет;
- Однако у нас теперь есть возможность работы с упакованным типом: вызывать у него методы;
- Также теперь мы имеем возможность подменить экземпляр значимого типа в SOH/LOH на другой. Этого мы не могли сделать раньше: нам пришлось бы делать `unboxing`, менять структуру на другую и делать `boxing` обратно, раздав новую ссылку потребителям.

Также давайте подумаем, какая основная проблема у упаковки? Создание трафика в памяти. Трафика непонятного количества объектов, часть из которых может выжить до первого поколения, где мы получим проблемы со сборкой мусора: он там будет, его там будет много, и этого явно можно было избежать. А когда мы имеем трафик короткоживущих объектов, первое решение, которое приходит в голову, - пуллинг. Вот это будет отличным завершением Кульбита Дотнетского.

```
var pool = new Pool<Boxed<Foo>>(maxCount:1000);
var boxed = pool.Box(10);
boxed.Value=70;

// use boxed value here

pool.Free(boxed);
```

Т.е. мы получили возможность работы боксинга через пул, тем самым удалив трафик памяти по части боксинга до нуля. Шутки ради можно даже сделать, чтобы в методе финализации объекты воскрешали бы себя, засовывая обратно в пул объектов. Это пригодилось бы для ситуаций, когда `boxed` структура уходит в чужой асинхронный код и нет возможности понять, когда она стала не нужна. В этом случае она сама себя вернет в пул во время GC.

А теперь давайте сделаем выводы:

- Если упаковка - случайна и такого не должно было произойти, будьте аккуратны и не допускайте ее возникновения: она может привести к проблемам производительности;
- Если упаковка - дань требованиям архитектуры той системы, которую вы делаете, то тут могут возникнуть варианты: если трафик упакованных структур мал и не заметен, можно не обращать никакого внимания и работать через упаковку. Если же трафик становится заметным, то возможно стоит сделать пуллинг боксинга через решение, указанное выше. Да, оно дает некоторые расходы на производительности пуллинга, зато GC спокоен и не работает на износ;

Напоследок, давайте рассмотрим пример из мира совершенно не практичного кода

```
static unsafe void Main()
{
    // делаем boxed int
    object boxed = 10;

    // забираем адрес указателя на VMT
    var address = (void**)EntityPtr.ToPointerWithOffset(boxed);

    unsafe
    {
        // забираем адрес Virtual Methods Table
        var structVmt = typeof(SimpleIntHolder).TypeHandle.Value.ToPointer();

        // меняем адрес VMT целого числа, ушедшего в Heap на VMT SimpleIntHolder,
превратив Int в структуру
        *address = structVmt;
    }

    var structure = (IGetterByInterface)boxed;

    Console.WriteLine(structure.GetByInterface());
}

interface IGetterByInterface
{
    int GetByInterface();
}

struct SimpleIntHolder : IGetterByInterface
{
    public int value;

    int IGetterByInterface.GetByInterface()
    {
        return value;
    }
}
```

Этот код написан при помощи маленькой функции, которая умеет получать указатель из ссылки на объект. Библиотека находится по адресу на github. Этот код показывает, что обычный boxing превращает int в типизированный reference type. Рассмотрим его по шагам:

- Делаем boxing для целого числа
- Достаем адрес полученного объекта (по этому адресу находится VMT Int32)
- Получаем VMT структуры SimpleIntHolder
- Подменяем VMT запакованного целого числа на VMT структуры
- Делаем unbox в тип структуры
- Выводим значение поля - на экран, доставая тем самым тот Int32, который был изначально запакован.

Причем заметьте, что делаю я это намеренно через интерфейс, показав, что так тоже будет работать.

## Nullable<T>

Также стоит сказать про то, как ведет себя `boxing` для Nullable значимых типов. Это очень приятная особенность Nullable значимых типов, поскольку boxing значимого типа, который как-бы null - просто возвратит null:

```
int? x = 5;
int? y = null;

var boxedX = (object)x; // -> 5
var boxedY = (object)y; // -> null
```

Отсюда следует забавный вывод: т.к. null не имеет типа, то единственный вариант вытащить из boxing'а другой тип нежели был запакован, такой:

```
int? x = null;
var pseudoBoxed = (object)x;
double? y = (double?)pseudoBoxed;
```

Код рабочий просто потому, что с null можно сделать приведение типа куда угодно. Но зато таким кодом можно запросто удивить своих коллег и скрасить вечер забавным фактом.

## Погружаемся в boxing еще глубже

На закуску хочу вам рассказать про тип System.Enum. Не проходя по ссылке скажите, пожалуйста, является ли тип Value Type или же Reference Type? По всем канонам и логике тип просто обязан быть значимым. Ведь это обычное перечисление: набор aliases с чисел на названия в языке программирования. И мы бы закончили этот разговор словами: "вы все думаете абсолютно правильно! Так не будем терять времени и пойдем дальше!", - но нет. `System.Enum`, от которого наследуются все `enum` типы данных, определенных как в вашем коде, так и в .NET Framework, является ссылочным типом. Т.е. типом данных `class`. Мало того (!) класс этот абстрактный и наследуется от `System.ValueType`.

```
    [Serializable]
    [System.Runtime.InteropServices.ComVisible(true)]
    public abstract class Enum : ValueType, IComparable, IFormattable, IConvertible
    {
        // ...
    }
```

Значит ли это, что все перечисления выделяются в куче SOH? Значит ли это, что, используя перечисления, мы забиваем кучу, а вместе и с ней - GC? Ведь мы просто используем их. Нет, такого не может быть. Тогда, получается, что есть где-то пул перечислений, в которых они лежат, и что нам отдаются их экземпляры. И снова нет: перечисления можно использовать в структурах при маршаллинге. Перечисления - обычные числа.

А правда заключается в том, что CLR при формировании структур типа данных прямо скажем подхачивает ее, если встечается `enum` превращая класс в значимый тип:

```
// Check to see if the class is a valuetype; but we don't want to mark System.Enum
// as a ValueType. To accomplish this, the check takes advantage of the fact
// that System.ValueType and System.Enum are loaded one immediately after the
// other in that order, and so if the parent MethodTable is System.ValueType and
// the System.Enum MethodTable is unset, then we must be building System.Enum and
// so we don't mark it as a ValueType.
if(HasParent() &&
    ((g_pEnumClass != NULL && GetParentMethodTable() == g_pValueTypeClass) ||
    GetParentMethodTable() == g_pEnumClass))
{
    bmtProp->fIsValueClass = true;

    HRESULT    hr    =    GetMDImport()->GetCustomAttributeByName(bmtInternal->pType-
>GetTypeDefToken(),
```

```
g_CompilerServicesUnsafeValueTypeAttribute,
                                                        NULL, NULL);
    IfFailThrow(hr);
    if (hr == S_OK)
    {
        SetUnsafeValueClass();
    }
}
```

Зачем это делается? В частности из-за идеи наследования: чтобы сделать пользовательский `enum`, необходимо снабдить его информацией об именах его возможных значений, например. А как сделать наследование у значимых типов? Никак. Вот и придумали они сделать его ссылочным, но на этапе работы JIT превращать его в значимый. Чтобы никто не догадался.

## Что если хочется лично посмотреть как работает boxing?

В наше время для того чтобы посмотреть на реализацию упаковки значимых типов, к счастью, нет необходимости загружать дизассемблер и лезть в самые дебри не пойми чего. В наше прекрасное время у нас есть исходные тексты всего ядра платформы .NET и многие его части абсолютно идентичны между .NET Framework CLR и CoreCLR. Вы можете пройти по ссылкам ниже и посмотреть на реализации упаковки прямо в исходных кодах:

- Существует отдельная группа оптимизаций, каждая из которых используется на отдельном типе процессора:
    - *JIT_BoxFastMP_InlineGetThread* (AMD64 - многопроцессорный или Server GC, implicit Thread Local Storage)
    - *JIT_BoxFastMP* (AMD64 - многопроцессорный или Server GC)
    - *JIT_BoxFastUP* (AMD64 - однопроцессорный или Workstation GC)
    - *JIT_TrialAlloc::GenBox(..)* (x86), которая присоединяется через JitHelpers
- В общем случае JIT инлайнит вызов вспомогательной функции Compiler::impImportAndPushBox(..)
- Generic-версия, использует менее оптимизированную MethodTable::Box(..)

- И наконец вызывается CopyValueClassUnchecked(..), по коду которой прекрасно видно, почему лучше всего выбирать размер структур до 8 байт включительно.

Для сравнения чтобы сделать распаковку реализован всего один метод: *JIT_Unbox(..)*, который является оберткой вокруг *JIT_Unbox_Helper(..)*.

Также интересным фактом является то, что сам процесс распаковки не является копированием данных из кучи: распаковка является передачей указателя на кучу с проверкой совместимости типов. А вот следующий за распаковкой опкод IL определит, что будет с этим адресом: возможно, данные будут скопированы в локальную переменную, а может быть - скопированы в стек для вызова метода. Ведь если бы это было не так, мы бы имели дело с двойным копированием: сначала при копировании куда-то из кучи, а затем - при копировании в место назначения.

### Раздел вопросов по теме

## Почему .NET CLR не делает пуллинга для боксинга самостоятельно?

Если мы поговорим с любым Java разработчиком, то мы с удивлением узнаем две вещи:

- В Java все Value Types запакованы и по сути не являются значимыми. Т.е. целые числа - тоже запакованы;
- Для оптимизации числа от -128 до 127 берутся из пула объектов.

Так почему же в .NET CLR во время упаковки не происходит того же самого? Ответ прост: потому что мы можем менять содержимое упакованного значимого типа. Т.е. другими словами, мы можем сделать так:

```
object x = 1;
x.GetType().GetField("m_value",                BindingFlags.Instance                |
BindingFlags.NonPublic).SetValue(x, 138);
Console.WriteLine(x); // -> 138
```

или так (C++/CLI):

```
void ChangeValue(Object^ obj)
{
    Int32^ i = (Int32^)obj;
    *i = 138;
}
```

Т.е. если бы мы имели дело с пуллингом, то мы бы поменяли все единицы в приложении на 138! Согласитесь, ситуация не из приятных.

Вторая причина - это сама суть значимых типов в .NET. Они работают по значению, а значит - быстрее. Упаковка для нас - редкая операция и сложение упакованных чисел - это скорее что-то из фантастики и очень плохой архитектуры. Это просто не будет чем-то хоть как-то полезным, чтобы заниматься реализацией данного функционала.

## Почему при вызове метода, принимающего тип object, а по факту - значимый тип нет возможности сделать boxing на стеке, разгрузив кучу?

Потому что если произойдет упаковка значимого типа на стеке, после чего ссылка уйдет в кучу, то внутри метода ссылка может уйти в какое-либо другое место, например метод может записать пришедшую ссылку в поле какого-либо класса. Далее метод завершит работу, после чего завершит работу метод, который сделал упаковку. Как результат - с внешнего класса ссылка будет указывать на "вымерший" участок стека.

## Почему нельзя использовать в качестве поля Value Type его самого?

Иногда возникает дерзкое желание в качестве поля структуры использовать другую структуру, которая использует первую. Или же еще проще: В качестве поля структуры использовать ее саму. Не задавайте мне вопросов, почему такое может понадобиться: на самом деле не может :) Но почему так сделать нельзя? Ответ прост: когда вы используете структуру в качестве поля ее же самой либо через зависимость от другой структуры, то фактически создаете рекурсию: вы создаете

структуру бесконечных размеров. Однако, в .NET Framework есть места, в которых так сделано. Примером может служить структура `System.Char`, которая содержит саму себя:

```
public struct Char : IComparable, IConvertible
{
    // Member Variables
    internal char m_value;
    //...
}
```

Кстати говоря, так сделаны практически все примитивные типы CLR. Нам смертным так сделать невозможно, да и нет никакой надобности: в CLR это сделано, чтобы добавить примитивным типам дух ООП

До сего момента, говоря про разницу между значимыми и ссылочными типами, мы затрагивали эту тему с высоты конечного разработчика. Т.е. мы никогда не смотрели на то, как они в реальности устроены и какие приемы реализованы в них на уровне CLR. Мы смотрели, фактически, на конечный результат и рассуждали с точки зрения изучения черного ящика. Однако, чтобы понимать суть вещей глубже и чтобы отбросить в сторону последние оставшиеся мысли о какой-либо магии, происходящей внутри CLR, стоит заглянуть в самые ее потроха и изучить те алгоритмы, которые систему типов регулируют.

## Внутренняя структура экземпляров типов

Перед тем как начинать рассуждение о строении управляющих блоков системы типов давайте посмотрим на сам объект, на экземпляр любого класса. Если мы создадим в памяти экземпляр любого ссылочного типа, будь то класс или же упакованная структура, то состоять он будет всего из трех полей: SyncBlockIndex (который на самом деле не только он), указатель на описатель типа и данные. Область данных может содержать очень много полей, но, не умаляя общности, ниже в примере полагаем, что в данных содержится одно поле. Так, если представить эту структуру графически, то мы получим следующее:

**System.Object**

```
-----------------------------------------------
|  SyncBlkIndx |   VMT_Ptr    |     Data     |
-----------------------------------------------
|  4 / 8 байт  |  4 / 8 байт  |  4 / 8 байт  |
-----------------------------------------------
|  0xFFF..FFF  |  0xXXX..XXX  |      0       |
-----------------------------------------------
               ^
               | Сюда ведут ссылки на объект. Т.е. не в начало, а на VMT

Sum size = 12 (x86) | 24 (x64)
```

Т.е. фактически размер экземпляра типа зависит от конечной платформы, на которой будет работать приложение.

Далее давайте проследуем по указателю `VMT_Ptr` и посмотрим, какая структура данных лежит по этому адресу. Для всей системы типов этот указатель является самым главным: именно через него работает и наследование, и реализация интерфейсов, и приведение типов, и много чего еще. Этот указатель - отсылка в систему типов .NET CLR, паспорт объекта, по которому CLR осуществляет приведение типов, понимает объем памяти, занимаемый объектом, именно при помощи него GC так лихо обходит объект, определяя, по каким адресам лежат указатели на объекты, а по каким - просто числа. Именно через него можно узнать вообще все об объекте и заставить CLR отрабатывать его по-другому. А потому именно им и займемся.

## Структура Virtual Methods Table

Описание самой таблицы доступно по адресу в GitHub CoreCLR, и если отбросить все лишнее (а там 4381 строка), выглядит она следующим образом:

Это версия из CoreCLR. Если смотреть на структуру полей в .NET Framework, то она будет отличаться расположением полей и расположением отдельных битов системной информации из двух битовых полей `m_wFlags` и `m_wFlags2`.

```
   // Low WORD is component size for array and string types (HasComponentSize() returns
true).
   // Used for flags otherwise.
   DWORD m_dwFlags;

   // Base size of instance of this class when allocated on the heap
   DWORD m_BaseSize;

   WORD  m_wFlags2;

   // Class token if it fits into 16-bits. If this is (WORD)-1, the class token is
stored in the TokenOverflow optional member.
   WORD  m_wToken;

   // <NICE> In the normal cases we shouldn't need a full word for each of these
</NICE>
   WORD  m_wNumVirtuals;
   WORD  m_wNumInterfaces;
```

Согласитесь, выглядит несколько пугающе. Причем пугающе выглядит не то, что тут всего 6 полей (а где все остальные?), а то, что для достижения этих полей, необходимо пропустить 4,100 строк логики. Я лично ожидал тут увидеть что-то

готовое, чего не надо дополнительно вычислять. Однако, тут все совсем не так просто: поскольку методов и интерфейсов в любом типе может быть различное количество, то и сама таблица VMT получается переменного размера. А это в свою очередь означает, что для достижения ее наполнения надо вычислять, где находятся все ее оставшиеся поля. Но давайте не будем унывать и попытаемся сразу получить выгоду из того, что мы уже имеем: мы, пока что, понятия не имеем, что имеется ввиду под другими полями (разве что два последних), зато поле `m_BaseSize` выглядит заманчиво. Как подсказывает нам комментарий, это - фактический размер для экземпляра типа. Мы только что нашли `sizeof` для классов! Попробуем в бою?

Итак, чтобы получить адрес VMT мы можем пойти двумя путями: либо сложным, получив адрес объекта, а значит и VMT:

```
class Program
{
   public static unsafe void Main()
   {
       Union x = new Union();
       x.Reference.Value = "Hello!";

       // Первым полем лежит указатель на место, где лежит указатель на VMT
       // - (IntPtr*)x.Value.Value - преобразовали число в указатель (сменили тип для
компилятора)
       // - *(IntPtr*)x.Value.Value - взяли по адресу объекта адрес VMT
       // - (void *)*(IntPtr*)x.Value.Value - преобразовали в указатель
       void *vmt = (void *)*(IntPtr*)x.Value.Value;

       // вывели в консоль адрес VMT;
       Console.WriteLine((ulong)vmt);
   }

   [StructLayout(LayoutKind.Explicit)]
   public class Union
   {
       public Union()
       {
           Value = new Holder<IntPtr>();
           Reference = new Holder<object>();
       }

       [FieldOffset(0)]
       public Holder<IntPtr> Value;

       [FieldOffset(0)]
       public Holder<object> Reference;
   }

   public class Holder<T>
   {
```

```
        public T Value;
    }
}
```

Либо простым, используя .NET FCL API:

```
    var vmt = typeof(string).TypeHandle.Value;
```

Второй путь конечно же проще (хоть и дольше работает). Однако знание первого очень важно с точки зрения понимания структуры экземпляра типа. Использование второго способа добавляет чувство уверенности: если мы вызываем метод API, то вроде как пользуемся задокументированным способом работы с VMT, а если достаем через указатели, то нет. Однако не стоит забывать, что хранение `VMT *` - стандартно для практически любого ООП языка и для .NET платформы в целом: эта ссылка всегда находится на одном и том же месте, как самое часто используемое поле класса. А самое часто используемое поле класса должно идти первым, чтобы адресация была без смещения и, как результат, была быстрей. Отсюда делаем вывод, что в случае классов положение полей на скорость влиять не будет, а вот у структур - самое часто используемое поле можно поставить первым. Хотя, конечно же для абсолютного большинства .NET приложений это не даст вообще никакого эффекта: не для таких задач создавалась эта платформа.

Давайте изучим вопрос структуры типов с точки зрения размера их экземпляра. Нам же надо не просто абстрактно изучать их (это просто-напросто скучно), но дополнительно попробуем извлечь из этого такую выгоду, какую не извлечь обычным способом.

**Почему sizeof есть для Value Type, но нет для Reference Type?** На самом деле вопрос открытый т.к. никто не мешает рассчитать размер ссылочного типа. Единственное обо что можно споткнуться - это не фиксированный размер двух ссылочных типов: `Array` и `String`. А также `Generic` группы, которая зависит целиком и полностью от конкретных вариантов. Т.е. оператором `sizeof(..)` мы обойтись не смогли бы: необходимо работать с конкретными экземплярами. Однако никто не мешает команде CLR сделать метод вида `static int System.Object.SizeOf(object obj)`, который бы легко и просто возвращал бы нам то, что надо. Так почему же Microsoft не реализовала этот метод? Есть мысль, что платформа .NET в их понимании опять

же - не та платформа, где разработчик будет сильно переживать за конкретные байты. В случае чего можно просто доставить планок в материнскую плату. Тем более что большинство типов данных, которые мы реализуем, не занимают такие большие объемы.

Но не будем отвлекаться. Итак, чтобы получить размер экземпляра класса, чей размер фиксированн, достаточно написать следующий код:

```
unsafe int SizeOf(Type type)
{
    MethodTable *pvmt = (MethodTable *)type.TypeHandle.Value.ToPointer();
    return pvmt->Size;
}

[StructLayout(LayoutKind.Explicit)]
public struct MethodTable
{
    [FieldOffset(4)]
    public int Size;
}

class Sample
{
    int x;
}

// ...

Console.WriteLine(SizeOf(typeof(Sample)));
```

Итак, что мы только что сделали? Первым шагом мы получили указатель на таблицу виртуальных методов. После чего мы считываем размер и получаем `12` - это сумма размеров полей `SyncBlockIndex + VMT_Ptr + поле x` для 32-разрядной платформы. Если мы поиграемся с разными типами, то получим примерно следующую таблицу для x86:

| Тип или его определение | Размер | Комментарий |
|---|---|---|
| Object | 12 | SyncBlk + VMT + пустое поле |

| Тип или его определение | Размер | Комментарий |
|---|---|---|
| Int16 | 12 | Boxed Int16: SyncBlk + VMT + данные (выровнено по 4 байта) |
| Int32 | 12 | Boxed Int32: SyncBlk + VMT + данные |
| Int64 | 16 | Boxed Int64: SyncBlk + VMT + данные |
| Char | 12 | Boxed Char: SyncBlk + VMT + данные (выровнено по 4 байта) |
| Double | 16 | Boxed Double: SyncBlk + VMT + данные |
| IEnumerable | 0 | Интерфейс не имеет размера: надо брать obj.GetType() |
| List<T> | 24 | Не важно сколько элементов в List, занимать он будет одинаково т.к. хранит данные он в array, который не учитывается |
| GenericSample<int> | 12 | Как видите, generics прекрасно считаются. Размер не поменялся, т.к. данные находятся на том же месте, |

| Тип или его определение | Размер | Комментарий |
|---|---|---|
| | | что и у boxed int. Итог: SyncBlk + VMT + данные |
| GenericSample<Int64> | 16 | Аналогично |
| GenericSample<IEnumerable> | 12 | Аналогично |
| GenericSample<DateTime> | 16 | Аналогично |
| string | 14 | Это значение будет возвращено для любой строки, т.к. реальный размер должен считаться динамически. Однако он подходит для размера под пустую строку. Прошу заметить, что размер не выровнен по разрядности: по сути это поле использоваться не должно |
| int[]{1} | 24554 | Для массивов в данном месте лежат совсем другие данные, также их размер не является фиксированным, |

| Тип или его определение | Размер | Комментарий |
|---|---|---|
| | | потому его необходимо считать отдельно |

Как видите, когда система хранит данные о размере экземпляра типа, то она фактически хранит данные для ссылочного вида этого типа (т.е. в том числе для ссылочного варианта значимого). Давайте сделаем некоторые выводы:

- Если вы хотите знать, сколько займет значимый тип как значение, используйте `sizeof(TType)`
- Если вы хотите рассчитать, чего вам будет стоить боксинг, то вы можете округлить `sizeof(TType)` в большую сторону до размера слова процессора (4 или 8 байт) и прибавить еще 2 слова (`2 * sizeof(IntPtr)`). Или же взять это значение из `VMT` типа.
- Рассчет выделеного объем памяти в куче представлен для следующих типов:
    1. Обычный ссылочный тип фиксированного размера: мы можем забрать размер экземпляра из `VMT`;
    2. Строка, необходимо вручную считать ее размер (это вообще редко когда может понадобиться, но, согласитесь, интересно)
    3. Массив, то его размер также рассчитывается отдельно: на основании размера его элементов и их количества. Эта задачка может оказаться куда более полезной: ведь именно массивы первые в очереди на попадание в `LOH`

## System.String

Про строки в вопросах практики мы поговорим отдельно: этому, относительно небольшому, классу можно выделить целую главу. А в рамках главы про строение VMT мы поговорим про строение строк на низком уровне. Для хранения строк

применяется стандарт UTF16. Это значит, что каждый символ занимает 2 байта. Дополнительно в конце каждой строки хранится null-терминатор – значение, которое идентифицирует окончание строки. Также в экземпляре хранится длина строки в виде `Int32` числа - чтобы не считать длину каждый раз, когда она понадобится (про кодировки мы поговорим отдельно). На схеме ниже представлена информация о занимаемой памяти строкой:

```
  // Для .NET Framework 3.5 и младше
  ------------------------------------------------------------------------------------------
  |  SyncBlkIndx |    VMTPtr     |  ArrayLength   |     Length     |   char   |   char    |
  ------------------------------------------------------------------------------------------
  |  4 / 8 байт  |  4 / 8 байт   |    4 байта     |    4 байта     |  2 байта |  2 байта  |
  ------------------------------------------------------------------------------------------
  |      -1      |  0xXXXXXXXX   |        3       |        2       |     a    |     b     |
  ------------------------------------------------------------------------------------------

  Term – null terminator
  Sum size = (8|16) + 2 * 4 + Count * 2 + 2 -> округлить в большую сторону по разрядности. (24 байта в примере)
  Count – количество символов в строке, не считая терминальный

  // Для .NET Framework 4 и старше
  ------------------------------------------------------------------------------------
  |  SyncBlkIndx |    VMTPtr     |     Length     |   char   |   char   |   Term    |
  ------------------------------------------------------------------------------------
  |  4 / 8 байт  |  4 / 8 байт   |    4 байта     |  2 байта |  2 байта |  2 байта  |
  ------------------------------------------------------------------------------------
  |      -1      |  0xXXXXXXXX   |        2       |     a    |     b    |   <nil>   |
  ------------------------------------------------------------------------------------
  Term - null terminator
  Sum size = (8|16) + 4 + Count * 2 + 2) -> округлить в большую сторону по разрядности. (20 байт в примере)
  Count – количество символов в строке, не считая терминальный
```

Перепишем наш метод, чтобы научить его считать размер строк:

```
unsafe int SizeOf(object obj)
{
   var majorNetVersion = Environment.Version.Major;
   var type = obj.GetType();
   var href = Union.GetRef(obj).ToInt64();
   var DWORD = sizeof(IntPtr);
   var baseSize = 3 * DWORD;

   if (type == typeof(string))
   {
       if (majorNetVersion >= 4)
       {
           var length = (int)*(int*)(href + DWORD /* skip vmt */);
```

```
            return DWORD * ((baseSize + 2 + 2 * length + (DWORD-1)) / DWORD);
        }
        Else*
        {
            // on 1.0 -> 3.5 string have additional RealLength field
            var arrlength = *(int*)(href + DWORD /* skip vmt */);
            var length = *(int*)(href + DWORD /* skip vmt */ + 4 /* skip length */);
            return DWORD * ((baseSize + 4 + 2 * length + (DWORD-1)) / DWORD);
        }
    }
    else
    if (type.BaseType == typeof(Array) || type == typeof(Array))
    {
        return ((ArrayInfo*)href)->SizeOf();
    }
    return SizeOf(type);
}
```

Где `SizeOf(type)` будет вызывать старую реализацию - для фиксированных по длине ссылочных типов.

Давайте проверим код на практике:

```
    Action<string> stringWriter = (arg) =>
    {
        Console.WriteLine($"Length of `{arg}` string: {SizeOf(arg)}");
    };

    stringWriter("a");
    stringWriter("ab");
    stringWriter("abc");
    stringWriter("abcd");
    stringWriter("abcde");
    stringWriter("abcdef");
}

-----

Length of `a` string: 16
Length of `ab` string: 20
Length of `abc` string: 20
Length of `abcd` string: 24
Length of `abcde` string: 24
Length of `abcdef` string: 28
```

Расчеты показывают, что размер строки увеличивается не линейно, а ступенчато на каждые два символа. Это происходит потому, что размер каждого символа - 2 байта, а конечный размер должен без остатка делиться на разрядность процессора (в примере x86), почему происходит соответствующее выравнивание размера строки на 2 байта. Результат нашей работы прекрасен: мы можем посчитать, во что нам обошлась та или иная строка. Последним этапом нам осталось узнать, как расчитать размер массивов в памяти.

## Массивы

Строение массивов несколько сложнее: ведь у массивов могут быть варианты их строения:

- Они могут хранить значимые типы, а могут хранить ссылочные
- Массивы могут быть как одномерными, так и многомерными
- Каждое измерение(мера) может начинаться как с `0`, так и с любого другого числа (это на мой взгляд очень спорная возможность, избавляющая программиста от необходимости в написании `arr[i - startIndex]` на уровне FCL). Сделано это, вроде как, для совместимости с другими языками, к примеру, в Pascal индексация массива может начинаться не с `0`, а с любого числа, однако мне кажется, что это лишнее.

Отсюда возникает некоторая путаность в реализации массивов и невозможность точно предсказать размер конечного массива: мало перемножить количество элементов на их размер. Хотя, конечно, для большинства случаев этого будет достаточно. Важным размер становится, когда мы боимся попасть в LOH. Однако у нас и тут возникают варианты: мы можем просто накинуть к размеру, подсчитанному "на коленке", какую-то константу сверху (например, 100), чтобы понять, перешагнули мы границу в 85000 или нет. Однако, в рамках данного раздела задача несколько другая: понять структуру типов. На нее и посмотрим:

```
  // Заголовок
  ------------------------------------------------------------------------------------
-----
  |    SBI    |  VMT_Ptr |  Total   |  Len_1   |  Len_2   | .. |  Len_N   |  Term    |
VMT_Child |
  ----------------------------------opt-------opt------------opt-------opt--------
opt-----
  |  4 / 8  |  4 / 8   |    4    |    4    |    4    |    |    4    |    4    |    4/8
|
  ------------------------------------------------------------------------------------
-----
  | 0xFF.FF | 0xXX.XX  |    ?    |    ?    |    ?    |    |    ?    | 0x00.00 | 0xXX.XX
|
  ------------------------------------------------------------------------------------
-----

  - opt: опционально
  - SBI: Sync Block Index
  - VMT_Child: присутствует, только если массив хранит данные ссылочного типа
```

```
  - Total: присутствует для оптимизации. Общее количество элементов массива с учетом всех размерностей
  - Len_2..Len_N, Term: присутствуют, только если размерность массива более 1 (регулируется битами в VMT->Flags)
```

Как мы видим, заголовок типа хранит данные об измерениях массива: их число может быть как 1, так и достаточно большим: фактически их размер ограничивается только null-терминатором, означающим, что перечисление закончено. Данный пример доступен полностью в файле GettingInstanceSize, а ниже я приведу только его самую важную часть:

```
public int SizeOf()
{
    var total = 0;
    int elementsize;

    fixed (void* entity = &MethodTable)
    {
        var arr = Union.GetObj<Array>((IntPtr)entity);
        var elementType = arr.GetType().GetElementType();

        if (elementType.IsValueType)
        {
            var typecode = Type.GetTypeCode(elementType);

            switch (typecode)
            {
                case TypeCode.Byte:
                case TypeCode.SByte:
                case TypeCode.Boolean:
                    elementsize = 1;
                    break;
                case TypeCode.Int16:
                case TypeCode.UInt16:
                case TypeCode.Char:
                    elementsize = 2;
                    break;
                case TypeCode.Int32:
                case TypeCode.UInt32:
                case TypeCode.Single:
                    elementsize = 4;
                    break;
                case TypeCode.Int64:
                case TypeCode.UInt64:
                case TypeCode.Double:
                    elementsize = 8;
                    break;
                case TypeCode.Decimal:
                    elementsize = 12;
                    break;
                default:
                    var info = (MethodTable*)elementType.TypeHandle.Value;
                    elementsize = info->Size - 2 * sizeof(IntPtr); // sync blk + vmt ptr
                    break;
            }
        }
```

```
        else
        {
            elementsize = IntPtr.Size;
        }

        // Header
        total += 3 * sizeof(IntPtr); // sync blk + vmt ptr + total length
        total += elementType.IsValueType ? 0 : sizeof(IntPtr); // MethodsTable for
refTypes
        total += IsMultidimensional ? Dimensions * sizeof(int) : 0;
    }

    // Contents
    total += (int)TotalLength * elementsize;

    // align size to IntPtr
    if ((total % sizeof(IntPtr)) != 0)
    {
        total += sizeof(IntPtr) - total % (sizeof(IntPtr));
    }
    return total;
}
```

Этот код учитывает все вариации типов массивов, и может быть использован для расчета его размера:

```
Console.WriteLine($"size of int[]{{1,2}}: {SizeOf(new int[2])}");
Console.WriteLine($"size of int[2,1]{{1,2}}: {SizeOf(new int[1,2])}");
Console.WriteLine($"size of int[2,3,4,5]{{...}}: {SizeOf(new int[2, 3, 4, 5])}");

---
size of int[]{1,2}: 20
size of int[2,1]{1,2}: 32
size of int[2,3,4,5]{...}: 512
```

## Выводы к разделу

На данном этапе мы научились нескольким достаточно важным вещам. Первое - мы разделили для себя ссылочные типы на три группы: ссылочные типы фиксированного размера, generic типы и ссылочные типы переменного размера. Также мы научились понимать структуру конечного экземпляра любого типа (про структуру VMT я пока молчу. Мы там поняли целиком пока что только одно поле: а это тоже большое достижение). Будь то ссылочный тип фиксированного размера (там все предельно просто) или неопределенного размера: массив или строка. Неопределенного потому, что его размер будет определен при создании. С generic типами на самом деле все просто: для каждого конкретного generic типа создается своя VMT, в которой будет проставлен конкретный размер.

## Таблица методов в Virtual Methods Table (VMT)

Объяснение работы Methods Table, по большей части носит академический характер: ведь в такие дебри лезть - это как самому себе могилу рыть. С одной стороны, такие закрома таят что-то будоражащее и интересное, хранят некие данные, которые еще больше раскрывают понимание о происходящем. Однако, с другой стороны, все мы понимаем, что Microsoft не будет нам давать никаких гарантий, что они оставят свой рантайм без изменений и, например, вдруг не передвинут таблицу методов на одно поле вперед. Поэтому, оговорюсь сразу:

Информация, представленная в данном разделе, дана вам исключительно для того, чтобы вы понимали, как работает приложение, основанное на CLR, и ручное вмешательство в ее работу не дает никаких гарантий. Однако, это настолько интересно, что я не могу вас отговорить. Наоборот, мой совет - поиграйтесь с этими структурами данных и, возможно, вы получите один из самых запоминающихся опытов в разработке ПО.

Ну все, предупредил. Теперь давайте окунемся в мир как говорится зазеркалья. Ведь до сих пор всё зазеркалье сводилось к знаниям структуры объектов: а её по-идее мы и так должны знать хотя бы примерно. И по своей сути эти знания зазеркальем не являются, а являются скорее входом в зазеркалье. А потому вернемся к структуре `MethodTable`, описанной в CoreCLR:

```
   // Low WORD is component size for array and string types (HasComponentSize() returns
true).
   // Used for flags otherwise.
   DWORD m_dwFlags;

   // Base size of instance of this class when allocated on the heap
   DWORD m_BaseSize;

   WORD  m_wFlags2;

   // Class token if it fits into 16-bits. If this is (WORD)-1, the class token is
stored in the TokenOverflow optional member.
   WORD  m_wToken;

   // <NICE> In the normal cases we shouldn't need a full word for each of these
</NICE>
   WORD  m_wNumVirtuals;
   WORD  m_wNumInterfaces;
```

А именно к полям `m_wNumVirtuals` и `m_wNumInterfaces`. Эти два поля определяют ответ на вопрос "сколько виртуальных методов и интерфейсов существует у типа?". В этой структуре нет никакой информации об обычных методах, полях, свойствах (которые объединяют методы). Т.е. эта структура **никак не связана с рефлексией**. По своей сути и назначению эта структура создана для работы вызова методов в CLR (и на самом деле в любом ООП: будь то Java, C++, Ruby или же что-то еще. Просто расположение полей будет несколько другим). Давайте рассмотрим код:

```
public class Sample
{
    public int _x;

    public void ChangeTo(int newValue)
    {
        _x = newValue;
    }

    public virtual int GetValue()
    {
        return _x;
    }
}

public class OverridedSample : Sample
{
    public override GetValue()
    {
        return 666;
    }
}
```

Какими бы бессмысленными не казались эти классы, они нам вполне сгодятся для описания их VMT. А для этого мы должны понять, чем отличаются базовый тип и унаследованный в вопросе методов `ChangeTo` и `GetValue`.

Метод `ChangeTo` присутствует в обоих типах: при этом его нельзя переопределять. Это значит, что он может быть переписан так, и ничего не поменяется:

```
public class Sample
{
    public int _x;

    public static void ChangeTo(Sample self, int newValue)
    {
        self._x = newValue;
    }

    // ...
}
```

```
// Либо в случае если бы он был struct
 public struct Sample
 {
     public int _x;

     public static void ChangeTo(ref Sample self, int newValue)
     {
         self._x = newValue;
     }

     // ...
 }
```

И при этом кроме архитектурного смысла ничего не изменится: поверьте, при компиляции оба варианта будут работать одинаково, т.к. у экземплярных методов `this` - это всего лишь первый параметр метода, который передается нам неявно.

Заранее поясню, почему все объяснения вокруг наследования строятся вокруг примеров на статических методах: по сути все методы - статические. И экземплярные и нет. В памяти нет поэкземплярно скомпилированных методов для каждого экземпляра класса. Это занимало бы огромное количество памяти: проще одному и тому же методу каждый раз передавать ссылку на экземпляр той структуры или класса, с которыми он работает.

Для метода `GetValue` все обстоит совершенно по-другому. Мы не можем просто взять и переопределить метод переопределением *статического* `GetValue` в унаследованном типе: новый метод получит только те участки кода, которые работают с переменной как с `OverridedSample`, а если с переменной работать как с переменной базового типа `Sample`, вызвать сможете только `GetValue` базового типа, поскольку вы понятия не имеете, какого типа на самом деле объект. Для того чтобы понимать, какого типа является переменная и, как результат, какой конкретно метод вызывается, мы можем поступить следующим образом:

```
void Main()
{
    var sample = new Sample();
    var overrided = new OverridedSample();


Console.WriteLine(sample.Virtuals[Sample.GetValuePosition].DynamicInvoke(sample));

Console.WriteLine(overrided.Virtuals[Sample.GetValuePosition].DynamicInvoke(sample));
}
```

```
public class Sample
{
    public const int GetValuePosition = 0;

    public Delegate[] Virtuals;

    public int _x;

    public Sample()
    {
        Virtuals = new Delegate[1] {
            new Func<Sample, int>(GetValue)
        };
    }

    public static void ChangeTo(Sample self, int newValue)
    {
        self._x = newValue;
    }

    public static int GetValue(Sample self)
    {
        return self._x;
    }
}

public class OverridedSample : Sample
{
    public OverridedSample() : base()
    {
        Virtuals[0] = new Func<Sample, int>(GetValue);
    }

    public static new int GetValue(Sample self)
    {
        return 666;
    }
}
```

В этом примере мы фактически строим таблицу виртуальных методов вручную, а вызовы делаем по позиции метода в этой таблице. Если вы поняли суть примера, то вы фактически поняли, как строится наследование на уровне скомпилированного кода: методы вызываются по своему индексу в таблице виртуальных методов. Просто когда вы создаете экземпляр некоторого унаследованного типа, то в его VMT по местам, где у базового типа находятся виртуальные методы, компилятор расположит указатели на переопределенные методы, скопировав из базового типа указатели на методы, которые не переопределялись. Таким образом, отличие нашего примера от реальной VMT заключается только в том, что когда компилятор строит эту таблицу, он заранее знает с чем имеет дело и создает таблицу правильного размера и наполнения сразу же: в нашем примере чтобы построить

таблицу для типов, которые будут делать таблицу более крупной за счет добавления новых методов, придется изрядно попотеть. Но наша задача заключается в другом, а потому такими извращениями мы заниматься не станем.

Второй вопрос, который возникает сразу после ответа на первый: если с методами теперь все ясно, то зачем тогда в `VMT` находятся интерфейсы? Интерфейсы, если размышлять логически, не входят в структуру прямого наследования. Они находятся как бы сбоку, указывая, что те или иные типы обязаны реализовывать некоторый набор методов. Иметь по сути некоторый протокол взаимодействия. Однако, хоть интерфейсы и находятся *сбоку* от прямого наследования, вызывать методы все равно можно. Причем, заметьте: если вы используете переменную интерфейсного типа, то за ней могут скрываться какие угодно классы, базовый тип у которых может быть разве что `System.Object`. Т.е. методы в таблице виртуальных методов, которые реализуют интерфейс могут находиться совершенно по разным местам. Как же вызов методов работает в этом случае?

## Virtual Stub Dispatch (VSD) [In Progress]

Чтобы разобраться в этом вопросе, необходимо дополнительно вспомнить, что реализовать интерфейс можно двумя путями: сделать можно либо `implicit` реализацию, либо `explicit`. Причем сделать это можно частично: часть методов сделать `implicit`, а часть - `explicit`. Эта возможность на самом деле - следствие реализации и возможно даже не является заранее продуманной: реализуя интерфейс, вы показываете явно или неявно, что в него входит. Часть методов класса может не входить в интерфейс, а методы, существующие в интерфейсе, могут не существовать в классе (они, конечно, существуют в классе, но синтаксис показывает, что архитектурно частью класса они не являются): класс и интерфейс - это, в некотором смысле, - параллельные иерархии типов. Также, в плюс к этому, интерфейс - это отдельный тип, а значит у каждого интерфейса есть собственная таблица виртуальных методов: чтобы каждый смог вызывать методы интерфейса.

Давайте взглянем на таблицу: как бы могли выглядеть VMT различных типов:

| interface IFoo | class A : IFoo | class B : IFoo |
|---|---|---|
| -> GetValue() | SampleMethod() | RunProcess() |
| -> SetValue() | Go() | -> GetValue() |
| | -> GetValue() | -> SetValue() |
| | -> SetValue() | LookToMoon() |

VMT всех трех типов содержат необходимые методы `GetValue` и `SetValue`, однако они находятся по разным индексам: они не могут везде быть по одним и тем же индексам, поскольку была бы конкуренция за индексы с другими интерфейсами класса. На самом деле для каждого интерфейса создается интерфейс - дубль - для каждой его реализации в каждом классе. Имеем 633 реализации `IDisposable` в классах FCL/BCL? Значит имеем 633 дополнительных `IDisposable` интерфейса чтобы поддержать VMT to VMT трансляцию для каждого из классов + запись в каждом классе с ссылкой на его реализацию интерфейсом. Назовем такие интерфейсы **частными интерфейсами**. Т.е. каждый класс имеет свои собственные, **частные интерфейсы**, которые являются "системными" и являются прокси типами до реального интерфейса.

Таким образом получается следующее: у интерфейсов также как и у классов есть наследование виртуальных *интерфейсных* методов, однако наследование это работает не только при наследовании одного интерфейса от другого, но и при реализации интерфейса классом. Когда класс реализует некий интерфейс, то создается дополнительный интерфейс, уточняющий какие методы интерфейса-родителя на какие методы конечного класса должны отображаться. Вызывая метод по интерфейсной переменной, вы точно также вызываете метод по индексу из массива VMT, как это делалось в случае с классами, однако для данной реализации

интерфейса вы по индексу выберите слот из *унаследованного*, невидимого интерфейса, связывающего оригинальный интерфейс `IDisposable` с нашим классом, интерфейс реализующим.

Диспетчеризация виртуальных методов через заглушки или **Virtual Stub Dispatch (VSD)** была разработана еще в 2006 году как замена таблицам виртуальных методов в интерфейсах. Основная идея этого подхода состоит в упрощении кодогенерации и последующего упрощения вызова методов, т.к. первичная реализация интерфейсов на VMT была бы очень громоздкой и требовала бы большого количества работы и времени для построения всех структур всех интерфейсов. Сам код диспетчеризации находится по сути в четырех файлах общим весом примерно в 6400 строк, и мы не строим целей понять его весь. Мы попытаемся в общих словах понять суть происходящих процессов в этом коде.

Всю логику VSD диспетчеризации можно разбить на два больших раздела: диспетчеризация и механизм заглушек (stubs), обеспечивающих кэширование адресов вызываемых методов по паре значений [тип;номер слота], которые их идентифицируют.

Для полного понимания протекающих при построении VSD процессов, давайте рассмотрим для начала их на очень высоком уровне, а затем - спустимся в самую глубь. Если говорить про механику диспетчеризации, то та откладывает их создание на потом, в силу логической параллельности иерархии интерфейсных типов, в силу того, что их в конечном счете станет очень много, и в силу того, что большую часть из них JIT никогда создавать не будет, т.к. наличие типов во Framework еще не означает, что их экземпляры будут созданы. Использование же традиционной VMT для *частных интерфейсов* создало бы ситуацию, при которой JIT пришлось бы создавать VMT для каждого *частного интерфейса* с самого начала. Т.е. создание каждого типа замедлилось бы как минимум в два раза. Основным классом, обеспечивающим диспетчеризацию, является класс `DispatchMap`, который внутри себя инкапсулирует таблицу типов интерфейсов, каждая из которых состоит из

таблицы методов, входящих в эти интерфейсы. Каждый метод может быть, в зависимости от стадии своего жизненного цикла, в четырех состояниях: состояние заглушки типа "метод еще не был ни разу вызван, его надо скомпилировать и подложить новую заглушку на место старой", состояние заглушки типа "метод должен быть каждый раз найден динамически, т.к. не может быть определен однозначно", состояние заглушки типа "метод доступен по однозначному адресу, а потому вызывается без какого либо поиска", или же полноценное тело метода.

Рассмотрим строение этих структур с точки зрения их генерирования и структур данных, необходимых для этого.

## DispatchMap

DispatchMap – это динамически строящаяся структура данных, являющаяся по своей сути основной структурой данных, на которую опирается работа интерфейсов в CLR. Структура ее выглядит следующим образом:

```
DWORD: Количество типов = N
DWORD: Тип №1
DWORD: Количество слотов для типа №1 = M1
DWORD: bool: смещения могут быть отрицательными
DWORD: Слот №1
DWORD: Целевой слот №1
...
DWORD: Слот №M1
DWORD: Целевой слот №M1
...
DWORD: Тип №N
DWORD: Количество слотов для типа №1 = MN
DWORD: bool: смещения могут быть отрицательными
DWORD: Слот №1
DWORD: Целевой слот №1
...
DWORD: Слот №MN
DWORD: Целевой слот №MN
```

Т.е. сначала записывается общее количество интерфейсов, реализуемых некоторыми типами. После чего для каждого интерфейса записывается его тип, количество реализуемых этим типом слотов (для навигации по таблице), а также для каждого слота - информация по этому слоту, а также целевой слот в *частном интерфейсе*, который содержит реализации методов для текущего типа.

Для навигации по этой структуре данных предусмотрен класс `EncodedMapIterator`, который является итератором. Т.е. никакой другой доступ, кроме как `foreach`, к `DispatchMap` не предусмотрен. Мало того номера слотов получены как разница реального номера слота и ранее закодированного номера слота. Т.е. получить номер слота в середине таблицы можно, только просмотрев всю структуру с самого начала. Это вызывает множество вопросов касательно производительности работы вызова методов через интерфейсы: ведь если у нас массив объектов, реализующих некий интерфейс, то чтобы понять, какой метод необходимо вызвать, надо просмотреть всю таблицу реализаций. Т.е. по своей сути - найти нужный. Результатом на каждом шаге итерирования будет структура `DispatchMapEntry`, которая покажет, где находится целевой метод: в текущем типе или нет, и какой слот необходимо взять у типа, чтобы получить нужный метод.

```
// DispatchMapTypeID позволяет делать относительную адресацию методов. Т.е. отвечает
на вопрос: относительно текущего типа где
// находится необходимый метод? В текущем типе или же в другом?
//
// Идентификатор типа (Type ID) используется в карте диспетчеризации и хранит внутри
себя один из следующих типов данных:
//   - специальное значение, говорящее, что это - "this" class
//   - специальное значение, показывающее, что это - тип интерфейса, не реализованный
классом
//   - индекс в InterfaceMap
class DispatchMapTypeID
{
private:
    static const UINT32 const_nFirstInterfaceIndex = 1;

    UINT32 m_typeIDVal;

    // ...
}

struct DispatchMapEntry
{
private:
    DispatchMapTypeID m_typeID;
    UINT16            m_slotNumber;
    UINT16            m_targetSlotNumber;

    enum
    {
        e_IS_VALID = 0x1
    };

    UINT16 m_flags;

    // ...
}
```

*TypeID Map*

Любой метод в адресации по интерфейсам кодируется парой <TypeId;SlotNumber>. TypeId - это, как следует из названия, идентификатор типа. Данное поле отвечает на вопросы: откуда берется этот идентификатор и каким образом его отразить на реальный тип. Класс `TypeIDMap` хранит карту типов как отражение некоторого TypeId на `MethodTable` конкретного типа, а также - дополнительно - в обратную сторону. Сделано это исключительно из соображений производительности. Построение этих хэш таблиц происходит динамически: по запросу TypeId относительно PTR_MethodTable возвращается либо FatId, либо просто Id. Это надо в некотором смысле просто помнить: FatId и Id - это просто два вида TypeId. И в некотором смысле это "указатель" на MethodTable, т.к. однозначно его идентифицирует.

**TypeId - это идентификатор MethodTable**. Он может быть двух видов: Id и FatId и по своей сути является обычным числом.

```
class TypeIDMap
{
protected:
    HashMap             m_idMap;  // Хранит map TypeID -> PTR_MethodTable
    HashMap             m_mtMap;  // Хранит map PTR_MethodTable -> TypeID1
    Crst                m_lock;
    TypeIDProvider      m_idProvider;
    BOOL                m_fUseFatIdsForUniqueness;
    UINT32              m_entryCount;

    // ...
}
```

Однако со всеми этими трудностями JIT справляется достаточно легко, вписывая вызовы конкретных методов в места их вызова по интерфейсу, когда это возможно. Если JIT понял, что ничего другого вызвано быть не может, он просто поставит вызов конкретного метода. Это - очень и очень сильная особенность JIT компилятора, который делает для нас эту прекрасную оптимизацию.

## Выводы

То, что для нас, как для программистов, на языке C# стало обыденностью и вросло корнями в наше сознание настолько, что мы даже не задумываясь понимаем, как делить приложение на классы и интерфейсы, порой реализовано так сложно для

понимания, что требуются недели анализа исходных текстов для определения всех зависимостей и логики происходящего. То, что для нас настолько обыденно в использовании, что не вызывает тени сомнения в простоте реализации, на самом деле может скрывать эти сложности реализации. Это говорит нам о том, что инженеры, воплотившие данные идеи, подходили к решению проблем с большим умом, тщательно анализируя каждый шаг.

То описание, которое здесь дано, на самом деле очень поверхностное и короткое: оно очень высокоуровневое. Даже не смотря на то, что относительно любой книги по .NET мы погрузились очень глубоко, данное описание построения VSD и VMT является очень и очень высокоуровневым. Ведь код файлов, описывающих эти две структуры данных, занимает в сумме около 20,000 строк кода. Это еще не учитывая некоторые части, отвечающие за Generics.

Однако, это позволяет нам сделать несколько выводов:

- Вызов статичных методов и экземплярных практически ничем не отличаются. А это значит, что нам не надо беспокоиться о том, что работа с экземплярными методами как-то повлияет на производительность. Производительность обоих методов абсолютно идентична при одинаковых условиях
- Вызов виртуальных методов хоть и идет через таблицу VMT, но из-за того, что индексы заранее известны, на каждый вызов дополнительно приходится лишь единственное разыменование указателя. В почти во всех случаях это ни на что не повлияет: проседание производительности (если вообще можно так выразиться) будет настолько маленьким, что им в принципе можно пренебречь
- Если говорить об интерфейсах, то тут стоит помнить о диспетчеризации и понимать, что работа через интерфейсы сильно усложняет реализацию подсистемы типов на низком уровне, приводя к ***возможным*** проседаниям в производительности, когда слишком часто, при вызове методов, отсутствует определенность в том, какой метод и какого класса вызывать у интерфейсной переменной. Однако, "интеллект" JIT компилятора позволяет в очень многих

случаях не проводить вызовы через диспетчеризацию, а напрямую вызывать метод, интерфейс реализующего

- Если вспомнить об обобщениях, то тут возникает еще один слой абстракции, который вносит сложность в поиск необходимых для вызова методов у типов, реализующих generic интерфейсы.

## Раздел вопросов по теме

Вопрос: почему если каждый класс может реализовать интерфейс, то нельзя вытащить конкретную реализацию интерфейса у объекта?

Ответ прост: это непокрытая возможность CLR при проектировании языка, CLR этот вопрос никак не ограничивает. Мало того, это с высокой долей вероятности будет добавлено в ближайших версиях C#, благо они выходят достаточно быстро. Рассмотрим пример:

```
void Main()
{
    var foo = new Foo();
    var boo = new Boo();

    ((IDisposable)foo).Dispose();
    foo.Dispose();
    ((IDisposable)boo).Dispose();
    boo.Dispose();
}

class Foo : IDisposable
{
    void IDisposable.Dispose()
    {
        Console.WriteLine("Foo.IDisposable::Dispose");
    }

    public void Dispose()
    {
        Console.WriteLine("Foo::Dispose()");
    }
}

class Boo : Foo, IDisposable
{
    void IDisposable.Dispose()
    {
        Console.WriteLine("Boo.IDisposable::Dispose");
    }

    public new void Dispose()
```

```
    {
        Console.WriteLine("Boo::Dispose()");
    }
}
```

Здесь мы вызываем четыре различных метода и результат их вызова будет таким:

```
Foo.IDisposable::Dispose
Foo::Dispose()
Boo.IDisposable::Dispose
Boo::Dispose()
```

Причем несмотря на то, что мы имеем *explicit* реализацию интерфейса в обоих классах, в классе `Boo` *explicit*реализацию интерфейса `IDisposable` для `Foo` получить не получится. Даже если мы напишем так:

```
((IDisposable)(Foo)boo).Dispose();
```

Все равно мы получим на экране все то же результат:

```
Boo.IDisposable::Dispose
```

## Что плохого в неявных и множественных реализациях интерфейсов?

В качестве примера "наследования интерфейсов", что аналогично наследованию классов, можно привести следующей код:

```
    Class Foo
        Implements IDisposable

        Public Overridable Sub DisposeImp() Implements IDisposable.Dispose
            Console.WriteLine("Foo.IDisposable::Dispose")
        End Sub

        Public Sub Dispose()
            Console.WriteLine("Foo::Dispose()")
        End Sub

    End Class

    Class Boo
        Inherits Foo
        Implements IDisposable

        Public Sub DisposeImp() Implements IDisposable.Dispose
            Console.WriteLine("Boo.IDisposable::Dispose")
        End Sub

        Public Shadows Sub Dispose()
            Console.WriteLine("Boo::Dispose()")
        End Sub

    End Class
```

```
    ''' <summary>
    ''' Неявно реализует интерфейс
    ''' </summary>
    Class Doo
        Inherits Foo

        ''' <summary>
        ''' Переопределение явной реализации
        ''' </summary>
        Public Overrides Sub DisposeImp()
            Console.WriteLine("Doo.IDisposable::Dispose")
        End Sub

        ''' <summary>
        ''' Неявное перекрытие
        ''' </summary>
        Public Sub Dispose()
            Console.WriteLine("Doo::Dispose()")
        End Sub

    End Class

    Sub Main()
        Dim foo As New Foo
        Dim boo As New Boo
        Dim doo As New Doo

        CType(foo, IDisposable).Dispose()
        foo.Dispose()
        CType(boo, IDisposable).Dispose()
        boo.Dispose()
        CType(doo, IDisposable).Dispose()
        doo.Dispose()
    End Sub
```

В нем видно, что `Doo`, наследуясь от `Foo`, неявно реализует `IDisposable`, но при этом переопределяет явную реализацию `IDisposable.Dispose`, что приведет к вызову переопределения при вызове по интерфесу, тем самым показывая "наследование интерфейсов" классов `Foo` и `Doo`.

С одной стороны, это вообще не проблема: если бы C# + CLR позволяли такие шалости, мы бы, в некотором, смысле получили нарушение консистентности в строении типов. Сами подумайте: вы сделали крутую архитектуру, все хорошо. Но кто-то почему-то вызывает методы не так, как вы задумали. Это было бы ужасно. С другой стороны, в C++ похожая возможность существует, и там не сильно жалуются на это. Почему я говорю, что это может быть добавлено в C#? Потому что не менее ужасный функционал уже обсуждается и выглядеть он должен примерно так:

```
interface IA
{
```

```
    void M() { WriteLine("IA.M"); }
}

interface IB : IA
{
    override void IA.M() { WriteLine("IB.M"); } // explicitly named
}

interface IC : IA
{
    override void M() { WriteLine("IC.M"); } // implicitly named
}
```

Почему это ужасно? Ведь на самом деле это порождает целый класс возможностей. Теперь нам не нужно будет каждый раз реализовывать какие-то методы интерфейсов, которые везде реализовывались одинаково. Звучит прекрасно. Но только звучит. Ведь интерфейс - это протокол взаимодействия. Протокол - это набор правил, рамки. В нем нельзя допускать существование реализаций. Здесь же идет прямое нарушение этого принципа и введение еще одного: множественного наследования. Я, честно, сильно против таких доработок, но... Я что-то ушел в сторону.

Сейчас, наверное, практически любой программист, который разрабатывает на платформе .NET, скажет, что ничего проще этого паттерна нет. Что это известный из известнейших шаблонов, которые применяются на платформе. Однако даже в самой простой и известнейшей проблемной области всегда найдется второе дно, а за ним еще ряд скрытых кармашков, в которые вы никогда не заглядывали. Однако, как для тех, кто смотрит тему впервые, так и для всех прочих (просто для того, чтобы каждый из вас вспомнил основы (не пропускайте эти абзацы (я слежу!))) - опишем все от самого начала и до самого конца.

## IDisposable

Если спросить, что такое IDisposable, вы наверняка ответите, что это

```
public interface IDisposable
{
    void Dispose();
}
```

Для чего же создан интерфейс? Ведь если у нас есть умный Garbage Collector, который за нас чистит всю память, делает так, чтобы мы вообще не задумывались о том, как чистить память, то становится не совсем понятно, зачем ее вообще чистить. Однако есть нюансы. Существует некоторое заблуждение, что `IDisposable` сделан, чтобы освобождать неуправляемые ресурсы. И это только часть правды. Чтобы единомоментно понять, что это не так, достаточно вспомнить примеры неуправляемых ресурсов. Является ли неуправляемым класс `File`? Нет. Может быть, `DbContext`? Опять же - нет. Неуправляемый ресурс - это то, что не входит в систему типов .NET. То, что не было создано платформой, и находящееся вне ее скоупа. Простой пример - это дескриптор открытого файла в операционной системе. Дескриптор - это некоторое число, которое однозначно идентифицирует открытый операционной системой файл. Не вами, а именно операционной системой. Т.е. все управляющие структуры (такие как координаты файла на файловой системе, его фрагменты в случае фрагментации и прочая служебная информация, номера

циллиндра, головки, сектора - в случае магнитного HDD) находятся не внутри платформы .NET, а внутри ОС. И единственным неуправляемым ресурсом, который уходит в платформу .NET, является IntPtr - число. Это число в свою очередь оборачивается FileSafeHandle, который в свою очередь оборачивается классом File. Т.е. класс File сам по себе неуправляемым ресурсом не является, но аккумулирует в себе, используя дополнительную прослойку в виде IntPtr, неуправляемый ресурс – дескриптор открытого файла. Как происходит чтение из такого файла? Через ряд методов WinAPI или ОС Linux.

Вторым примером неуправляемых ресурсов являются примитивы синхронизации в многопоточных и мультипроцессных программах. Такие как мьютексы, семафоры. Или же массивы данных, которые передаются через P/Invoke.

Стоит заметить что ОС не просто передает приложению дескриптор неуправляемого ресурса, но дополнительно сохраняет его в таблице открытых дескрипторов процесса сохраняя за собой возможность корректного закрытия этих ресурсов при завершении работы приложения. Т.е. другими словами при выходе из приложения ресурсы закрыты будут в любом случае. Однако время работы приложения может быть разным и как результат - можно получить заблокированный надолго ресурс.

Хорошо. С неуправляемыми ресурсами разобрались. Зачем же IDisposable в этих случаях? Затем, что .NET Framework понятия не имеет о том, что происходит там, где его нет. Если вы открываете файл при помощи функций ОС, .NET ничего об этом не узнает. Если вы выделите участок памяти под собственные нужды (например, при помощи VirtualAlloc), .NET также ничего об этом не узнает. А если он ничего об этом не знает, он не освободит память, которая была занята вызовом VirtualAlloc. Или не закроет файл, открытый напрямую через вызов API ОС. Последствия этого могут быть совершенно разными и непредсказуемыми. Вы можете получить OutOfMemory, если навыделяете слишком много памяти и не будете ее освобождать (а, например, по старой памяти будете просто обнулять указатель) либо заблокируете на долгое время файл на файловой шаре, если он был открыт через средства ОС, но не был закрыт. Пример с файловыми шарами особенно

хорош, потому что блокировка останется даже после закрытия соединения с сервером - на стороне IIS. А прав на освобождение блокировки у вас может не быть и придется делать запрос администраторам на `iisreset` либо ручное закрытие ресурсов при помощи специализированного ПО. Таким образом решение этой проблемы может стать не тривиальной задачей на удаленном сервере.

Во всех этих случаях необходим универсальный и узнаваемый *протокол взаимодействия* между системой типов и программистом, который однозначно будет идентифицировать те типы, которые требуют принудительного закрытия. Этот *протокол* и есть интерфейс IDisposable. И звучит это примерно так: если тип содержит реализацию интерфейса IDisposable, то после того, как вы закончите работу с его экземпляром, вы обязаны вызвать Dispose().

И ровно по этой причине есть два стандартных пути его вызова. Ведь, как правило, вы либо создаете экземпляр сущности, чтобы быстренько с ней поработать в рамках одного метода, либо в рамках времени жизни экземпляра своей сущности.

Первый вариант - это когда вы оборачиваете экземпляр в `using(...){ ... }`. Т.е. вы прямо указываете, что по окончании блока using объект должен быть уничтожен. Т.е. должен быть вызван Dispose(). Второй вариант - уничтожить его по окончании времени жизни объекта, который содержит ссылку на тот, который надо освободить. Но ведь в .NET кроме метода финализации нет ничего, что намекало бы на автоматическое уничтожение объекта. Правильно? Но финализация нам совсем не подходит по той причине, что она будет неизвестно когда вызвана. А нам надо освободать именно тогда, когда необходимо: сразу после того, как нам более не нужен, например, открытый файл. Именно поэтому мы также должны реализовать IDisposable у себя и в методе Dispose вызвать Dispose у всех, кем мы владели, чтобы освободить и их тоже. Таким образом мы соблюдаем *протокол*, и это очень важно. Ведь если кто-то начал соблюдать некий протокол, его должны соблюдать все участники процесса: иначе будут проблемы.

## Вариации реализации IDisposable

Давайте пойдем в реализациях IDisposable от простого к сложному.

Первая и самая простая реализация, которая только может прийти в голову, - это просто взять и реализовать IDisposable:

```
public class ResourceHolder : IDisposable
{
    DisposableResource _anotherResource = new DisposableResource();

    public void Dispose()
    {
        _anotherResource.Dispose();
    }
}
```

Т.е. для начала мы создаем экземпляр некоторого ресурса, который должен быть освобожден; этот ресурс и освобождается в методе Dispose(). Единственное, чего здесь нет и что делает реализацию не консистентной, - это возможность дальнейшей работы с экземпляром класса после его разрушения методом `Dispose()`:

```
public class ResourceHolder : IDisposable
{
    private DisposableResource _anotherResource = new DisposableResource();
    private bool _disposed;

    public void Dispose()
    {
        if(_disposed) return;

        _anotherResource.Dispose();
        _disposed = true;
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private void CheckDisposed()
    {
        if(_disposed) {
            throw new ObjectDisposedException();
        }
    }
}
```

Вызов CheckDisposed() необходимо вызывать первым выражением во всех публичных методах класса. Однако, если для разрушения управляемого ресурса, коим является `DisposableResource`, полученная структура класса `ResourceHolder` выглядит нормально, то для случая инкапсулирования неуправляемого ресурса - нет.

Давайте придумаем вариант с неуправляемым ресурсом.

```
public class FileWrapper : IDisposable
{
    IntPtr _handle;

    public FileWrapper(string name)
    {
        _handle = CreateFile(name, 0, 0, 0, 0, 0, IntPtr.Zero);
    }

    public void Dispose()
    {
        CloseHandle(_handle);
    }

    [DllImport("kernel32.dll", EntryPoint = "CreateFile", SetLastError = true)]
    private static extern IntPtr CreateFile(String lpFileName,
        UInt32 dwDesiredAccess, UInt32 dwShareMode,
        IntPtr lpSecurityAttributes, UInt32 dwCreationDisposition,
        UInt32 dwFlagsAndAttributes,
        IntPtr hTemplateFile);

    [DllImport("kernel32.dll", SetLastError=true)]
    private static extern bool CloseHandle(IntPtr hObject);
}
```

Так какая разница в поведении двух последних примеров? В первом варианте у нас описано взаимодействие управляемого ресурса с другим управляемым. Это означает, что в случае корректной работы программы ресурс будет освобожден в любом случае. Ведь `DisposableResource` у нас - управляемый, а значит, .NET CLR о нем прекрасно знает и, в случае некорректного поведения, освободит из-под него память. Заметьте, что я намеренно не делаю никаких предположений о том, что тип `DisposableResource` инкапсулирует. Там может быть какая угодно логика и структура. Она может содержать как управляемые, так и неуправляемые ресурсы. *Нас это волновать не должно*. Нас же не просят каждый раз декомпилировать чужие библиотеки и смотреть, какие типы что используют: управляемые или неуправляемые ресурсы. А если *наш тип* использует неуправляемый ресурс, мы не можем этого не знать. Это мы делаем в классе `FileWrapper`. Так что же произойдет в этом случае?

Если мы используем неуправляемые ресурсы, получается, что у нас опять же два варианта: когда все хорошо и метод Dispose вызвался (тогда все хорошо :) ) и когда

что-то случилось и метод Dispose отработать не смог. Сразу оговоримся, почему этого может не произойти:

- Если мы используем `using(obj) { ... }`, то во внутреннем блоке кода может возникнуть исключение, которое перехватывается блоком `finally`, который нам не видно (это синтаксический сахар C#). В этом блоке неявно вызывается Dispose. Однако есть случаи, когда этого не происходит. Например, `StackOverflowException`, который не перехватывается ни `catch`, ни `finally`. Это всегда надо учитывать. Ведь если у вас некий поток уйдет в рекурсию и в некоторой точке вылетит по `StackOverflowException`, то те ресурсы, которые были захвачены и не были освобождены, забудутся .NET'ом. Ведь он понятия не имеет, как освобождать неуправляемые ресурсы: они повиснут в памяти до тех пор, пока ОС не освободит их сама (например, при выходе из вашей программы. А иногда и неопределенное время уже после завершения работы приложения).
- Если мы вызываем Dispose() из другого Dispose(). Тогда может так получиться, что опять же мы не сможем до него дойти. И тут вопрос вовсе не в забывчивости автора приложения: мол, забыл Dispose() вызвать. Нет. Опять же, вопрос в любых исключениях. Но теперь речь идет не только об исключениях, обрушающих поток приложения. Тут уже речь идет вообще о любых исключениях, которые приведут к тому, что алгоритм не дойдет до вызова внешнего Dispose(), который вызовет наш.

Во всех таких случаях возникнет ситуация подвешенных в воздухе неуправляемых ресурсов. Ведь Garbage Collector понятия не имеет, что их надобно собрать. Максимум что он сделает - при очередном проходе поймет, что на граф объектов, содержащих наш объект типа `FileWrapper`, потеряна последняя ссылка и память перетрется теми объектами, на которые ссылки есть.

Как же защититься от подобного? Для этих случаев мы обязаны реализовать финализатор объекта. Финализатор не случайно имеет именно такое название. Это вовсе не деструктор, как может показаться изначально из-за схожести объявления

финализаторов в C# и деструкторов в C++. Финализатор, в отличии от деструктора, вызовется *гарантированно*, тогда как деструктор может и не вызваться (ровно как и `Dispose()`). Финализатор вызывается, когда запускается Garbage Collection (пока этого знания достаточно, но по факту все несколько сложнее), и предназначен для гарантированного освобождения захваченных ресурсов, если *что-то пошло не так*. И для случая освобождения неуправляемых ресурсов мы *обязаны* реализовывать финализатор. Также, повторюсь, из-за того, что финализатор вызывается при запуске GC, в общем случае вы понятия не имеете, когда это произойдет.

Давайте расширим наш код:

```
public class FileWrapper : IDisposable
{
    IntPtr _handle;

    public FileWrapper(string name)
    {
        _handle = CreateFile(name, 0, 0, 0, 0, 0, IntPtr.Zero);
    }

    public void Dispose()
    {
        InternalDispose();
        GC.SuppressFinalize(this);
    }

    private void InternalDispose()
    {
        CloseHandle(_handle);
    }

    ~FileWrapper()
    {
        InternalDispose();
    }

    /// other methods
}
```

Мы усилили пример знаниями о процессе финализации и тем самым обезопасили приложение от потери информации о ресурсах, если что-то пошло не так и Dispose() вызван не будет. Дополнительно, мы сделали вызов GC.SuppressFinalize для того, чтобы отключить финализацию экземпляра типа, если для него был вызван Dispose(). Нам же не надо дважды освобождать один и тот же ресурс? Также это стоит сделать по другой причине: мы снимаем нагрузку с очереди на финализацию, ускоряя случайный участок кода, в параллели с которым будет в случайном будущем отрабатывать финализация.

Теперь давайте еще усилим наш пример:

```
public class FileWrapper : IDisposable
{
    IntPtr _handle;
    bool _disposed;

    public FileWrapper(string name)
    {
        _handle = CreateFile(name, 0, 0, 0, 0, 0, IntPtr.Zero);
    }

    public void Dispose()
    {
        if(_disposed) return;
        _disposed = true;

        InternalDispose();
        GC.SuppressFinalize(this);
    }


    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private void CheckDisposed()
    {
        if(_disposed) {
            throw new ObjectDisposedException();
        }
    }

    private void InternalDispose()
    {
        CloseHandle(_handle);
    }

    ~FileWrapper()
    {
        InternalDispose();
    }

    /// other methods
}
```

Теперь наш пример реализации типа, инкапсулирующего неуправляемый ресурс, выглядит законченным. Повторный `Dispose()`, к сожалению, является де-факто стандартом платформы, и мы позволяем его вызвать. Замечу, что зачастую люди допускают повторный вызов `Dispose()` для того, чтобы избежать мороки с вызывающим кодом, и это не правильно. Однако пользователь вашей библиотеки с оглядкой на документацию MS может так не считать и допускать множественные вызовы Dispose(). Вызов же других публичных методов в любом случае ломает целостность объекта. Если мы разрушили объект, значит с ним работать более

нельзя. Это в свою очередь означает, что мы обязаны вставлять вызов `CheckDisposed` в начало каждого публичного метода.

Однако в этом коде существует очень серьезная проблема, которая не даст ему работать так, как задумали мы. Если мы вспоминим, как отрабатывает процесс сборки мусора, то заметим одну деталь. При сборке мусора GC *в первую очередь* финализирует все, что напрямую унаследовано от *Object*, после чего принимается за те объекты, которые реализуют *CriticalFinalizerObject*. У нас же получается, что оба класса, которые мы спроектировали, наследуют Object: и это проблема. Мы понятия не имеем, в каком порядке мы уйдем на "последнюю милю". Тем не менее, более высокоуровневый объект может пытаться работать с объектом, который хранит неуправляемый ресурс - в своем финализаторе (хотя это уже звучит как плохая идея). Тут нам бы сильно пригодился порядок финализации. И для того чтобы его задать - мы должны унаследовать наш тип, инкапсулирующий unmanaged ресурс, от `CriticalFinalizerObject`.

Вторая причина имеет более глубокие корни. Представьте себе, что вы позволили себе написать приложение, которое не сильно заботится о памяти. Аллоцирует в огромных количествах без кэширования и прочих премудростей. Однажды такое приложение завалится с OutOfMemoryException. А когда приложение падает с этим исключением, возникают особые условия исполнения кода: ему нельзя что-либо пытаться аллоцировать. Ведь это приведет к повторному исключению, даже если предыдущее было поймано. Это вовсе не обозначает, что мы не должны создавать новые экземпляры объектов. К этому исключению может привести обычный вызов метода. Например, вызов метода финализации. Напомню, что методы компилируются тогда, когда они вызываются в первый раз. И это обычное поведение. Как же уберечься от этой проблемы? Достаточно легко. Если вы отнаследуете объект от *CriticalFinalizerObject*, то *все* методы этого типа будут компилироваться сразу же, при загрузке типа в память. Мало того, если вы пометите методы атрибутом *[PrePrepareMethod]*, то они также будут предварительно

скомпилированны и будут безопасными с точки зрения вызова при нехватке ресурсов.

Почему это так важно? Зачем тратить так много усилий на тех, кто уйдет в мир иной? А все дело в том, что неуправляемые ресурсы могут повиснуть в системе очень надолго. Даже после того, как ваше приложение завершит работу. Даже после перезагрузки компьютера (если пользователь открыл в вашем приложении файл с файловой шары, тот будет заблокирован удаленным хостом и отпущен либо по таймауту, либо когда вы освободите ресурс, закрыв файл. Если ваше приложение вылетит в момент открытого файла, то он не будет закрыт даже после перезагрузки. Придется ждать достаточно продолжительное время для того, чтобы удаленный хост отпустил бы его). Плюс ко всему вам нельзя допускать выброса исключений в финализаторах - это приведет к ускоренной гибели CLR и окончательному выбросу из приложения: вызовы финализаторов не оборачиваются *try .. catch*. Т.е. освобождая ресурс, вам надо быть уверенными в том, что он еще может быть освобожден. И последний не менее интересный факт - если CLR осуществляет аварийную выгрузку домена, финализаторы типов, производных от *CriticalFinalizerObject*, также будут вызваны, в отличие от тех, кто наследовался напрямую от *Object*.

## SafeHandle / CriticalHandle / SafeBuffer / производные

У меня есть некоторое ощущение, что я для вас сейчас открою ящик Пандоры. Давайте поговорим про специальные типы: SafeHandle, CriticalHandle и их производные. И закончим уже, наконец, наш шаблон типа, предоставляющего доступ к unmanaged ресурсу. Но перед этим давайте попробуем перечислить все, что к нам *обычно* идет из unmanaged мира:

- Первое и самое ожидаемое, что оттуда обычно идет, - это дескрипторы (handles). Для разработчика .NET это может быть абсолютно пустым словом,

но это очень важная составляющая мира операционных систем. По своей сути handle - это 32-х либо 64-х разрядное число, определяющее открытую сессию взаимодействия с операционной системой. Т.е., например, открываете вы файл, чтобы с ним поработать, а в ответ от WinApi-функции получили дескриптор. После чего, используя его, можете продолжать работать именно с ним: делаете *Seek*, *Read*, *Write* операции. Второй пример: открываете сокет для работы с сетью. И опять же: операционная система отдаст вам дескриптор. В мире .NET дескрипторы хранятся в типе *IntPtr*;

- Второе - это массивы данных. Существует несколько путей работы с неуправляемыми массивами: либо работать с ним через unsafe код (ключевое слово unsafe), либо использовать SafeBuffer, который обернет буфер данных удобным .NET-классом. Замечу, что хоть первый способ быстрее (вы можете сильно оптимизировать циклы, например), то второй способ - намного безопаснее. Ведь он использует SafeHandle как основу для работы;
- Строки. Со строками все несколько проще, потому что наша задача - определить формат и кодировку строки, которую мы забираем. Далее строка копируется к нам (класс string - immutable) и мы дальше ни о чем не думаем.
- ValueTypes, которые забираются копированием и о судьбе которых думать вообще нет никакой необходимости.

SafeHandle - это специальный класс .NET CLR, который наследует CriticalFinalizerObject и который призван обернуть дескрипторы операционной системы максимально безопасно и удобно.

```
[SecurityCritical,              SecurityPermission(SecurityAction.InheritanceDemand,
UnmanagedCode=true)]
public abstract class SafeHandle : CriticalFinalizerObject, IDisposable
{
    protected IntPtr handle;        // Дескриптор, пришедший от ОС
    private int _state;             // Состояние (валидность, счетчик ссылок)
    private bool _ownsHandle;        // Флаг возможности освободить handle. Может так
получиться, что мы оборачиваем чужой handle и освобождать его не имеем права
    private bool _fullyInitialized; // Экземпляр проинициализирован

    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail)]
    protected SafeHandle(IntPtr invalidHandleValue, bool ownsHandle)
```

```
    {
    }

    // Финализатор по шаблону вызывает Dispose(false)
    [SecuritySafeCritical]
    ~SafeHandle()
    {
        Dispose(false);
    }

    // Выставление hanlde может идти как вручную, так и при помощи p/invoke Marshal - автоматически
    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
    protected void SetHandle(IntPtr handle)
    {
        this.handle = handle;
    }

    // Метод необходим для того, чтобы с IntPtr можно было бы работать напрямую. Используется
    // для определения того, удалось ли создать дескриптор, сравнив его с одим из ранее
    // определенных известных значений. Обратите внимание, что метод опасен по двум причинам:
    //   - Если дескриптор отмечен как недопустимый с помощью SetHandleasInvalid, DangerousGetHandle
    //    то все равно вернет исходное значение дескриптора.
    //  - Возвращенный дескриптор может быть переиспользован в любом месте. Это может как минимум
    //    означать, что он без обратной связи перестанет работать. В худшем случае при прямой передаче
    //    IntPtr в другое место, он может уйти в ненадежный код и стать вектором атаки на приложение
    //    через подмену ресурса на одном IntPtr
    [ResourceExposure(ResourceScope.None), ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
    public IntPtr DangerousGetHandle()
    {
        return handle;
    }

    // Ресурс закрыт (более не доступен для работы)
    public bool IsClosed {
        [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
        get { return (_state & 1) == 1; }
    }

    // Ресурс не является доступным для работы. Вы можете переопределить свойство, изменив логику.
    public abstract bool IsInvalid {
        [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
        get;
    }

    // Закрытие ресурса через шаблон Close()
    [SecurityCritical,          ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
    public void Close() {
        Dispose(true);
    }

    // Закрытие ресурса через шаблон Dispose()
```

```
    [SecuritySafeCritical,       ReliabilityContract(Consistency.WillNotCorruptState,
Cer.Success)]
    public void Dispose() {
        Dispose(true);
    }

    [SecurityCritical,           ReliabilityContract(Consistency.WillNotCorruptState,
Cer.Success)]
    protected virtual void Dispose(bool disposing)
    {
        // ...
    }

    // Вы должны вызывать этот метод всякий раз, когда понимаете, что handle более не
является рабочим.
    // Если вы этого не сделаете, можете получить утечку
    [SecurityCritical, ResourceExposure(ResourceScope.None)]
    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
    [MethodImplAttribute(MethodImplOptions.InternalCall)]
    public extern void SetHandleAsInvalid();

    // Переопределите данный метод, чтобы указать, каким образом необходимо освобождать
    // ресурс. Необходимо быть крайне осторожным при написании кода, т.к. из него
    // нельзя вызывать нескомпилированные методы, создавать новые объекты и бросать
исключения.
    // Возвращаемое значение - маркер успешности операции освобождения ресурсов.
    // Причем если возвращаемое значение = false, будет брошено исключение
    //      SafeHandleCriticalFailure,      которое      в      случае      включенного
SafeHandleCriticalFailure
    // Managed Debugger Assistant войдет в точку останова.
    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
    protected abstract bool ReleaseHandle();


    // Работа со счетчиком ссылок. Будет объяснено далее по тексту
    [SecurityCritical, ResourceExposure(ResourceScope.None)]
    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail)]
    [MethodImplAttribute(MethodImplOptions.InternalCall)]
    public extern void DangerousAddRef(ref bool success);
    public extern void DangerousRelease();
}
```

Чтобы оценить полезность группы классов, производных от SafeHandle, достаточно вспомнить, чем хороши все .NET типы: автоматизированностью уборки мусора. Т.о., оборачивая неуправляемый ресурс, SafeHandle наделяет его такими же свойствами, т.к. является управляемым. Плюс ко всему он содержит внутренний счетчик внешних ссылок, которые не могут быть учтены CLR. Т.е. ссылками из unsafe кода. Вручную увеличивать и уменьшать счетчик нет почти никакой необходимости: когда вы объявляете любой тип, производный от SafeHandle, как параметр unsafe метода, то при входе в метод счетчик будет увеличен, а при выходе - уменьшен. Это свойство введено по той причине, что когда вы перешли в unsafe код, передав туда дескриптор, то в другом потоке (если вы, конечно, работаете с одним дескриптором

из нескольких потоков) обнулив ссылку на него, получите собранный SafeHandle. Со счетчиком же ссылок все проще: SafeHandle не будет собран, пока дополнительно не обнулится счетчик. Вот почему вручную менять счетчик не стоит. Либо это надо делать очень аккуратно: возвращая его, как только это становится возможным.

Второе назначение счетчика ссылок - это задание порядка финализации `CriticalFinalizerObject`, которые друг на друга ссылаются. Если один SafeHandle-based тип ссылается на другой SafeHandle-based тип, то в конструкторе ссылающегося необходимо дополнительно увеличить счетчик ссылок, а в методе ReleaseHandle - уменьшить. Таким образом ваш объект не будет уничтожен, пока не будет уничтожен тот, на который вы сослались. Однако чтобы не путаться, стоит избегать таких ситуаций.

Давайте напишем финальный вариант нашего класса, но теперь уже с последними знаниями о SafeHandlers:

```
public class FileWrapper : IDisposable
{
    SafeFileHandle _handle;
    bool _disposed;

    public FileWrapper(string name)
    {
        _handle = CreateFile(name, 0, 0, 0, 0, 0, IntPtr.Zero);
    }

    public void Dispose()
    {
        if(_disposed) return;
        _disposed = true;
        _handle.Dispose();
    }


    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private void CheckDisposed()
    {
        if(_disposed) {
            throw new ObjectDisposedException();
        }
    }

    [DllImport("kernel32.dll", EntryPoint = "CreateFile", SetLastError = true)]
    private static extern SafeFileHandle CreateFile(String lpFileName,
        UInt32 dwDesiredAccess, UInt32 dwShareMode,
        IntPtr lpSecurityAttributes, UInt32 dwCreationDisposition,
        UInt32 dwFlagsAndAttributes,
        IntPtr hTemplateFile);
```

```
    /// other methods
}
```

Что его отличает? Зная, что если в DllImport методе в качестве возвращаемого значения установить **любой** (в том числе и свой) SafeHandle-based тип, то Marshal его корректно создаст и проинициализирует, установив счетчик использований в 1, мы ставим тип SafeFileHandle в качестве возвращаемого для функции ядра CreateFile. Получив его, мы будем при вызове ReadFile и WriteFile использовать именно его (т.к. при вызове счетчик опять же увеличится, а при выходе - уменьшится, что даст нам гарантию существования handle на все время чтения и записи в файл). Тип этот спроектирован корректно, а это значит, что он гарантированно закроет файловый дескриптор, даже когда процесс аварийно завершит свою работу. А это значит, что нам не надо реализовывать свой finalizer и все, что с ним связано. Наш тип значительно упрощается.

## Срабатывание finalizer во время работы экземплярнх методов

В процессе сборки мусора есть одна оптимизация, направленная на то чтобы как можно раньше собрать наибольшее количество объектов. Давайте рассмотрим следующий код:

```
public void SampleMethod()
{
    var obj = new object();
    obj.ToString();

    // ...
    // Если в этой точке сработает GC, obj с некоторой степенью вероятности будет собрана
    // т.к. она более не используется
    // ...

    Console.ReadLine();
}

С одной стороны код выглядит достаточно безопасно и не сразу становится ясно, почему это должно нас хоть как-то казаться. Однако достаточно вспомнить что существуют классы, оборачивающие собой неуправляемые ресурсы как сразу приходит понимание, что если класс будет спроектирован не корректно, то вполне можно получить исключение из unmanaged мира, которое будет говорить о том что handle, который был получен ранее уже не активен:

```csharp
// Пример абсолютно не правильной реализации
void Main()
{
```
```

```
    var inst = new SampleClass();
    inst.ReadData();
    // далее inst не используется
}

public sealed class SampleClass : CriticalFinalizerObject, IDisposable
{
    private IntPtr _handle;

    public SampleClass()
    {
        _handle = CreateFile("test.txt", 0, 0, IntPtr.Zero, 0, 0, IntPtr.Zero);
    }

    public void Dispose()
    {
        if (_handle != IntPtr.Zero)
        {
            CloseHandle(_handle);
            _handle = IntPtr.Zero;
        }
    }

    ~SampleClass()
    {
        Console.WriteLine("Finalizing instance.");
        Dispose();
    }

    public unsafe void ReadData()
    {
        Console.WriteLine("Calling GC.Collect...");

        // я специально перевел на локальную переменную чтобы
        // не задействовать this после GC.Collect();
        var handle = _handle;

        // Имитация полного GC.Collect
        GC.Collect();
        GC.WaitForPendingFinalizers();
        GC.Collect();

        Console.WriteLine("Finished doing something.");
        var overlapped = new NativeOverlapped();

        // Делаем не важно что
        ReadFileEx(handle, new byte[] { }, 0, ref overlapped, (a, b, c) => {;});
    }

    [DllImport("kernel32.dll", SetLastError = true, CharSet = CharSet.Auto,
BestFitMapping = false)]
    static extern IntPtr CreateFile(String lpFileName, int dwDesiredAccess, int
dwShareMode,
    IntPtr securityAttrs, int dwCreationDisposition, int dwFlagsAndAttributes, IntPtr
hTemplateFile);

    [DllImport("kernel32.dll", SetLastError = true)]
    static extern bool ReadFileEx(IntPtr hFile, [Out] byte[] lpBuffer, uint
nNumberOfBytesToRead,
    [In] ref NativeOverlapped lpOverlapped, IOCompletionCallback lpCompletionRoutine);
```

```
    [DllImport("kernel32.dll", SetLastError = true)]
    static extern bool CloseHandle(IntPtr hObject);
}
```

Согласитесь: этот код выглядит более-менее прилично. Во всяком случае явно он никак не сообщает что есть какая-то проблема. А проблема есть и при том очень серъезная. Возможна попытка закрытия файла финализатором класса во время чтения из файла. Что практически гарантированно приведет к ошибке. Причем поскольку в данном случае ошибка будет именно возвращена (`IntPtr == -1`), то мы этого не увидим, `_handle` будет обнулен, дальнейший `Dispose` не закроет файл, а мы получим утечку ресурса. Для решения этой проблемы необходимо пользоваться `SafeHandle`, `CriticalHandle`, `SafeBuffer` и их производными, которые кроме того что имеют счетчики использования в unmanaged мире, так еще и эти счетчики автоматически увеличиваются при передаче в unmanaged методы и уменьшаются - при выходе из него.

## Многопоточность

Теперь поговорим про тонкий лед. В предыдущих частях рассказа об IDisposable мы проговорили одну очень важную концепцию, которая лежит не только в основе проектирования Disposable типов, но и в проектировании любого типа: концепция целостности объекта. Это значит, что в любой момент времени объект находится в строго определенном состоянии, и любое действие над ним переводит его состояние в одно из заранее определенных - при проектировании типа этого объекта. Другими словами - никакое действие над объектом не должно иметь возможность перевести его состояние в то, которое не было определено. Из этого вытекает проблема в спроектированных ранее типах: они не потокобезопаны. Есть потенциальная возможность вызова публичных методов этих типов в то время, как идет разрушение объекта. Давайте решим эту проблему и решим, стоит ли вообще ее решать

```
public class FileWrapper : IDisposable
{
    IntPtr _handle;
    bool _disposed;
    object _disposingSync = new object();
```

```
    public FileWrapper(string name)
    {
        _handle = CreateFile(name, 0, 0, 0, 0, 0, IntPtr.Zero);
    }

    public void Seek(int position)
    {
        lock(_disposingSync)
        {
            CheckDisposed();
            // Seek API call
        }
    }

    public void Dispose()
    {
        lock(_disposingSync)
        {
            if(_disposed) return;
            _disposed = true;
        }
        InternalDispose();
        GC.SuppressFinalize(this);
    }


    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private void CheckDisposed()
    {
        lock(_disposingSync)
        {
            if(_disposed) {
                throw new ObjectDisposedException();
            }
        }
    }

    private void InternalDispose()
    {
        CloseHandle(_handle);
    }

    ~FileWrapper()
    {
        InternalDispose();
    }

    /// other methods
}
```

Установка критической секции на код проверки `_disposed` в Dispose() и по факту - установка критической секции на весь код публичных методов. Это решит нашу проблему одновременного входа в публичный метод экемпляра типа и в метод его разрушения, однако создаст таймер замедленного действия для ряда других проблем:

- Интенсивная работа с методами экземпляра типа, а также работа по созданию и разрушению объектов приведет к сильному проседанию по производительности. Все дело в том, что взятие блокировки занимает некоторое время. Это время необходимо для аллокации таблиц SyncBlockIndex, проверок на текущий поток и много чего еще (мы рассмотрим все это отдельно - в разделе про многопоточность). Т.е. получается, что ради "последней мили" жизни объекта мы будем платить производительностью все время его жизни!
- Дополнительный memory traffic для объектов синхронизации
- Дополнительные шаги для обхода графа объектов при GC

Второе, и на мой взгляд, самое важное. Мы допускаем ситуацию одновременного разрушения объекта с возможностью поработать с ним еще разок. На что мы вообще должны надеяться в данном случае? Что не выстрелит? Ведь если сначала отработает Dispose, то дальнейшее обращение с методам объекта обязано привести к `ObjectDisposedException`. Отсюда возникает простой вывод: синхронизацию между вызовами Dispose() и остальными публичными методами типа необходимо делигировать обслуживающей стороне. Т.е. тому коду, который создал экземпляр класса `FileWrapper`. Ведь только создающая сторона в курсе, что она собирается делать с экземпляром класса и когда она собирается его разрушать.

С другой стороны по требованиям к архитектуре классов, реализующих IDisposable вызов Dispose должен выкидывать только критические ошибки (такие как `OutOfMemoryException`. но не IOException, например). Это в частности значит что если Dispose вызовется более чем из одного потока одновременно, то может произойти ситуация, когда разрушение сущности будет происходить одновременно из двух потоков (проскочим проверку `if(_disposed) return;`). Тут зависит от ситуации: если освобождение ресурсов *может* идти несколько раз, то никаких дополнительных проверок не потребуется. Если же нет, необходима защита:

```
// Я намеренно не привожу весь шаблон, т.к. пример будет большим
// и не покажет сути
class Disposable : IDisposable
{
```

```
    private volatile int _disposed;

    public void Dispose()
    {
        if(Interlocked.CompareExchange(ref _disposed, 1, 0) == 0)
        {
            // dispose
        }
    }
}
```

## Два уровня Disposable Design Principle

Какой самый популярный шаблон реализации `IDisposable` можно встретить в книгах по .NET разработке и во Всемирной Паутине? Какой шаблон ждут от вас люди в компаниях, когда вы идете собеседоваться на потенциально новое место работы? Вероятнее всего этот:

```
public class Disposable : IDisposable
{
    bool _disposed;

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if(disposing)
        {
            // освобождаем управляемые ресурсы
        }
        // освобождаем неуправляемые ресурсы
    }

    protected void CheckDisposed()
    {
        if(_disposed)
        {
            throw new ObjectDisposedException();
        }
    }

    ~Disposable()
    {
        Dispose(false);
    }
}
```

Что здесь не так и почему мы ранее в этой книге никогда так не писали? На самом деле шаблон хороший и без лишних слов охватывает все жизненные ситуации. Но его использование повсеместно, на мой взгляд, не является правилом хорошего

тона: ведь реальных неуправляемых ресурсов мы в практике почти никогда не видим, и в этом случае пол-шаблона работает в холостую. Мало того, он нарушает принцип разделения ответственности. Ведь он одновременно управляет и управляемыми ресурсами и неуправляемыми. На мой скромный взгляд, это совершенно не правильно. Давайте взглянем на несколько иной подход. *Disposable Design Principle*. Если коротко, то суть в следующем:

Disposing разделяется на два уровня классов:

- Типы Level 0 напрямую инкапсулируют неуправляемые ресурсы
    - Они являются либо абстрактными, либо запакованными
    - Все методы должны быть помечены:
        - PrePrepareMethod, чтобы метод был скомпилирован вместе с загрузкой типа
        - SecuritySafeCritical, чтобы выставить защиту на вызов из кода, работающего под ограничениями
        - ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success / MayFail)] чтобы выставить CER на метод и все его дочерние вызовы
        - Могут ссылаться на типы нулевого уровня, но должны увеличивать счетчик ссылающихся объектов, чтобы гарантировать порядок выхода на "последнюю милю"
- Типы Level 1 инкапсулируют только управляемые ресурсы
    - Наследуются только от типов Level 1 либо реализуют IDisposable напрямую
    - Не имеют права наследовать типы Level 0 или CriticalFinalizerObject
    - Могут инкапсулировать управляемые типы Level 1 или Level 0
    1. Реализуют IDisposable.Dispose путем разрушения инкапсулированных объектов в порядке: сначала типы Level 0, потом типы Level 1
    2. Т.к. они не имеют неуправляемых ресурсов - то не реализуют finalizer
    3. Должно содержать protected свойство, дающее доступ к Level 0 типам.

Именно поэтому я с самого начала ввел разделение на два типа: на содержащий управляемый ресурс и содержащий неуправляемый ресурс. Они должны работать совершенно по-разному.

## Как еще используется Dispose

Идеологически IDisposable был создан для освобождения неуправляемых ресурсов. Но как и для многих других шаблонов оказалось, что он очень полезен и для других задач. Например, для освобождения ссылок на управляемые ресурсы. Звучит как-то не очень полезно: освобождать управляемые ресурсы. Ведь нам же объяснили, что управляемые ресурсы - они на то и управляемые, чтобы мы расслабились и смотрели в сторону разработчиков C/C++ с едва заметной ухмылкой. Однако все не совсем так. Мы всегда можем получить ситуацию, когда мы теряем ссылку на объект и думаем, что все хорошо: GC соберет мусор, а вместе с ним и наш объект. Однако, выясняется, что память растет, мы лезем в программу анализа памяти и видим, что на самом деле этот объект удерживается чем-то еще. Все дело в том, что как в платформе .NET, так и в архитектуре внешних классов может присутствовать логика неявного захвата ссылки на вашу сущность. После чего, ввиду неявности захвата, программист может пропустить необходимость ее освобождения и получить на выходе утечку памяти.

### Делегаты, events

Взглянем на следующий синтетический пример:

```
class Secondary
{
    Action _action;

    void SaveForUseInFuture(Action action)
    {
        _action = action;
    }

    public void CallAction()
    {
        _action();
    }
}
```

```
class Primary
{
    Secondary _foo = new Secondary();

    public void PlanSayHello()
    {
        _foo.SaveForUseInFuture(Strategy);
    }

    public void SayHello()
    {
        _foo.CallAction();
    }

    void Strategy()
    {
        Console.WriteLine("Hello!");
    }
}
```

Какая проблема здесь демонстрируется? Класс Secondary хранит делегат типа `Action` в поле `_action`, который принимается в методе `SaveForUseInFuture`. Далее в классе `Primary` метод `PlanSayHello` отдает в `Secondary` сигнатуру метода `Strategy`. Забавно, но если вы здесь будете отдавать статический метод или метод экземпляра, то сам вызов `SaveForUseInFuture` никак не изменится: просто *неявно* будет или не будет отдаваться ссылка на экземпляр класса `Primary`. Т.е. внешне выглядит что вы отдали указание, какой метод стоит вызывать. На самом деле помимо сигнатуры метода дегелат строится на основе указателя на экземпляр класса. Вызывающая сторона же должна понимать, для какого экземпляра класса она должна будет вызвать метод `Strategy`! Т.е. экземпляр класса `Secondary` неявно получил в удержание указатель на экземпляр класса `Primary`, хотя явно это не указано. Для нас это должно означать только одно: если мы отдадим указатель `_foo` куда-то еще, а на `Primary` потеряем ссылку, то GC **не соберет** объект `Primary`, т.к. его будет удерживать `Secondary`. Как избежать таких неприятных ситуаций? Необходим детерменированный подход к освобождению ссылки на нас. И тут к нам на помощь приходит механизм, который прекрасно подходит для наших целей: `IDisposable`

```
// Для простоты указана упрощенная версия реализации
class Secondary : IDisposable
{
    Action _action;

    public event Action<Secondary> OnDisposed;
```

```
    public void SaveForUseInFuture(Action action)
    {
        _action = action;
    }

    public void CallAction()
    {
        _action?.Invoke();
    }

    void Dispose()
    {
        _action = null;
        OnDisposed?.Invoke(this);
    }
}
```

Теперь пример выглядит приемлимо: если экземпляр класса передадут третьей стороне, но при этом в процессе работы будет потеряна ссылка на делегат `_action`, то мы его обнулим, а третья сторона будет извещена о разрушении экземпляра класса и затрет ссылку на него, отправив в мир иной.

Вторая опасность кода, работающего на делегатах, кроется в механизме работы `event`. Давайте посмотрим во что они разворачиваются:

```
 // закрытое поле обработчика
private Action<Secondary> _event;

// методы add/remove помечены как [MethodImpl(MethodImplOptions.Synchronized)],
// что аналогично lock(this)
public event Action<Secondary> OnDisposed {
    add { lock(this) { _event += value; } }
    remove { lock(this) { _event -= value; } }
}
```

Механизм сообщений в C# скрывает внутреннее устройство event'ов и удерживает все объекты, которые подписались на обновления через `event`. Если что-то пойдет не так, ссылка на чужой объект останется в `OnDisposed` и будет его удерживать. Получается странная ситуация: архитектурно мы имеем понятие "источник событий", которое по своей логике не должно что-либо удерживать. По факту мы имеем неявное удерживание объектов, подписавшихся на обновления. При этом не имеем возможности что-либо менять внутри этого массива делегатов: хоть сущность и является частью нас, нам на это прав не давали. Единственное что мы можем - это затереть весь список полностью, присвоив источнику событий null. Второй способ - явно реализовать методы `add`/`remove` дабы ввести управление над коллекцией делегатов.

Кстати, тут возникает еще одна неявная ситуация: может показаться, что если вы присвоите источнику событий null, то дальнейшая подписка на события приведет к `NullReferenceException`. И на мой скромный взгляд это было бы логичнее. Однако это не так: если внешний код подпишется на события после того как источник событий будет очищен, FCL создаст новый экземпляр класса Action и положит его в `OnDisposed`. Эта неявность в языке C# может запутать программиста: работа с обнуленными полями должна вызывать в вас не чувство спокойствия, а скорее тревогу. Тут же демонстрируется подход, когда излишняя расслабленность может привести программиста к утечкам памяти.

## Лямбды, замыкания

Особая опасность нас подстерегает при использовании такого синтаксического сахара как лямбды.

Я бы хотел затронуть вопрос синтаксического сахара в целом. На мой взгляд его стоит использовать достаточно аккуратно и только если вы абсолютно точно знаете, к чему это приведет. Примеры с лямбда-выражениями: замыкания, замыкания в Expressions и множество других бед, которые можно на себя навлечь.

Ну скажите себе честно: да, я знаю, что лямбда-выражение создает замыкание и тянет за собой риск утечки ресурсов. Но оно ведь такое... лаконичное... приятное... как можно удержаться и не поставить лямбду вместо выделения целого метода, который будет описан отдельно от точки использования? А вот надо на самом деле не повестись на эту провокацию, хотя и не каждый может.

Давайте рассмотрим пример:

```
button.Clicked += () => service.SendMessageAsync(MessageType.Deploy);
```

Согласитесь, эта строчка выглядит очень безопасно. Но она в себе таит большую проблему: теперь переменная `button` неявно ссылается на `service` и удерживает его. Даже если мы решим, что `service` нам более не нужен, `button` так считать не может: кнопка будет удерживать ссылку, пока сама будет жива.

Один из путей решения - воспользоваться шаблоном создания `IDisposable` из любого `Action`(`System.Reactive.Disposables`):

```
// Создаем из лямбды делегат
Action action = () => service.SendMessageAsync(MessageType.Deploy);

// Подписываемся
button.Clicked += action;
```

```
// Создаем отписку
var subscription = Disposable.Create(() => button.Clicked -= action);

// где-то, где надо отписаться
subscription.Dispose();
```

Получилось, согласитесь, несколько многословно, и при этом теряется весь смысл использования лямбда-выражений. Гораздо проще и безопаснее в плане неявных захватов переменнях будет использование обычных приватных методов.

## Защита от ThreadAbort

Когда разрабатывается библиотека для внешнего разработчика, вы никак не можете гарантировать, как она себя поведет в чужом приложении. Иногда остается только догадываться, что такого с нашей библиотекой сделал чужой программист, что появился тот или иной результат ее работы. Один из примеров - работа в многопоточной среде, когда вопрос целостности очистки ресурсов может встать достаточно остро. Причем, если при написании кода `Dispose()` метода сами мы можем дать гарантии на отсутствие исключительных ситуаций, то мы не можем гарантировать, что прямо во время работы метода `Dispose()` не вылетит `ThreadAbortException`, который отключит наш поток исполнения. Тут стоит вспомнить тот факт, что когда бросается ёThreadAbortExceptionё, то в любом случае выполняются все catch/finally блоки (в конце catch/finally ThreadAbort бросается дальше). Таким образом, чтобы что-то сделать гарантированно (гарантировав неразрывность при помощи Thread.Abort), надо обернуть критичный участок в `try { ... } finally { ... }`. В этом случае даже если бросят ThreadAbort, код будет выполнен.

```
void Dispose()
{
    if(_disposed) return;

    _someInstance.Unsubscribe(this);
    _disposed = true;
}
```

может быть оборван в любой точке при помощи `Thread.Abort`. Это в частности приведет к тому, что объект будет частично разрушен и позволит работать с собой в дальнейшем. Тогда как следующий код:

```
void Dispose()
{
    if(_disposed) return;

    // Защита от ThreadAbortException
    try {}
    finally
    {
        _someInstance.Unsubscribe(this);
        _disposed = true;
    }
}
```

защищен от такого прерывания и выполнится гарантированно и корректно, даже если `Thread.Abort` возникнет между операцией вызова метода `Unsubscribe` и исполнения его инструкций.

# Итоги

## Плюсы

Итак, мы узнали много нового про этот простейший шаблон. Давайте определим его плюсы:

- Основным плюсом шаблона является возможность детерменированного освобождения ресурсов: тогда, когда это необходимо
- Введение общеизвестного способа узнать, что конкретный тип требует разрушения его экземпляров в конце использования
- При грамотной реализации шаблона работа спроектированного типа станет безопасной с точки зрения использования сторонними компонентами, а также с точки зрения выгрузки и разрушения ресурсов при обрушении процесса (например, из-за нехватки памяти)

## Минусы

Минусов шаблона я вижу намного больше, чем плюсов:

- С одной стороны получается, что любой тип, реализующий этот шаблон, отдает тем самым команду всем, кто его будет использовать: используя меня, вы принимаете публичную оферту. Причем так неявно это сообщает, что как

и в случае публичных оферт пользователь типа не всегда в курсе, что у типа есть этот интерфейс. Приходится, например, следовать подсказкам IDE (ставить точку, набирать Dis.. и проверять, есть ли метод в отфильтрованном списке членов класса). И если Dispose замечен, реализовывать шаблон у себя. Иногда это может случиться не сразу, и тогда реализацию шаблона придется протягивать через систему типов, которая участвует в функционале. Хороший пример: а вы знали что `IEnumerator<T>` тянет за собой `IDisposable`?

- Зачастую, когда проектируется некий интерфейс, встает необходимость вставки IDisposable в систему интерфейсов типа: когда один из интерфейсов вынужден наследовать IDisposable. На мой взгляд это вносит "кривь" в те интерфейсы, которые мы спроектировали. Ведь когда проектируется интерфейс, вы прежде всего проектируете некий протокол взаимодействия. Тот набор действий, которые можно сделать *с чем-либо*, скрывающимся под интерфейсом. Метод Dispose() - метод разрушения экземпляра класса. Это входит в разрез с сущностью *протокол взаимодействия*. Это по сути - подробности реализации, которые просочились в интерфейс;
- Несмотря на детерменированность, Dispose() не означает прямого разрушения объекта. Объект все еще будет существовать после его *разрушения*. Просто в другом состоянии. И чтобы это стало правдой, вы обязаны вызывать CheckDisposed() в начале каждого публичного метода. Это выглядит как хороший такой костыль, который отдали нам со словами: "плодите и размножайте!";
- Есть еще маловероятная возможность получить тип, который реализует `IDisposable` через *explicit* реализацию. Или получить тип, реализующий IDisposable без возможности определить, кто его должен разрушать: сторона, которая выдала, или вы сами. Это породило антипаттерн множественного вызова Dispose(), который по сути позволяет разрешать разрушенный объект;
- Полная реализация сложна. Причем различна для управляемых и неуправляемых ресурсов. В этом плане попытка облегчить жизнь

разработчикам через GC выглядит немного нелепо. Можно, конечно, вводить некий тип DisposableObject, который реализует весь шаблон, отдав `virtual void Dispose()` метод для переопределения, но это не решит других проблем, связанных с шаблоном;

- Реализация метода `Dispose()` как правило идет в конце файла, тогда как `ctor` объявляется в начале. При модификации класса и вводе новых ресурсов можно легко ошибиться и забыть зарегистрировать disposing для них.
- Наконец, использование шаблона на графах объектов, которые полностью либо частично его реализуют, - та еще морока в определении порядка *разрушения* в многопоточной среде. Я прежде всего имею ввиду ситуации, когда Dispose() может начаться с разных концов графа. И в таких ситуациях лучше всего воспользоваться другими шаблонами. Например, шаблоном Lifetime.
- Желание разработчиков платформы сделать управление памятью автоматической вместе с реалиями: приложения очень часто взаимодействуют с неуправляемым кодом + необходимо контролировать освобождение ссылок на объекты дабы их собрал Garbage Collector вносит огромное количество путанницы в понимание вопросов: "как правильно реализовать шаблон? и есть ли шаблон вообще?". Возможно вызов `delete obj; delete[] arr;` был бы проще?

## Выгрузка домена и выход из приложения

Если вы сюда дошли, значит вы стали как минимум увереннее в успешности последующих собеседований. Однако, мы обсудили еще не все вопросы, связанные с этим, казалось бы, простым шаблоном. Последним вопросом у нас идет вопрос: отличается ли поведение приложения при простом GC, GC во время выгрузки домена и GC во время выхода из приложения? Процедуры `Dispose()` этот вопрос касается только если по касательной... Но `Dispose()` и финализация идут рука об

руку, и редко когда мы можем видеть реализации класса, в котором есть финализация, но нет метода `Dispose()`. Потому давайте договоримся так: саму финализацию мы опишем в разделе, посвященном финализации, а здесь лишь добавим несколько важных пунктов.

Когда выгружается домен приложения, то выгружаются как сборки, которые были загружены в домен, так и все объекты, которые были созданы в рамках выгружаемого домена. Это значит, что по сути происходит очищение (сборка GC) этих объектов, и для них будут вызваны финализаторы. Если наша логика финализатора ждет финализации других объектов, чтобы быть уничтоженным в правильном порядке, то возможно стоит обратить внимание на свойство `Environment.HasShutdownStarted`, обозначающее, что приложение в данный момент находится в состоянии выгрузки из памяти, и метод `AppDomain.CurrentDomain.IsFinalizingForUnload()`, который говорит о том, что данный домен выгружается, что и является причиной финализации. Ведь если наступили эти события, то в целом становится все равно, в каком порядке мы должны финализировать ресурсы. Задерживать выгрузку домена и приложения мы не можем: наша задача все сделать максимально быстро.

Вот так эта задача решается в рамках класса LoaderAllocatorScout

```
// Assemblies and LoaderAllocators will be cleaned up during AppDomain shutdown in
// unmanaged code
// So it is ok to skip reregistration and cleanup for finalization during appdomain
shutdown.
// We also avoid early finalization of LoaderAllocatorScout due to AD unload when the
object was inside DelayedFinalizationList.
if (!Environment.HasShutdownStarted &&
    !AppDomain.CurrentDomain.IsFinalizingForUnload())
{
    // Destroy returns false if the managed LoaderAllocator is still alive.
    if (!Destroy(m_nativeLoaderAllocator))
    {
        // Somebody might have been holding a reference on us via weak handle.
        // We will keep trying. It will be hopefully released eventually.
        GC.ReRegisterForFinalize(this);
    }
}
```

## Типичные ошибки реализации

Итак, как я вам показал, общего, универсального шаблона для реализации IDisposable не существует. Мало того, некоторая уверенность в автоматизме управления памятью заставляет людей путаться и принимать запутанные решения в реализации шаблона. Так, например, весь .NET Framework пронизан ошибками в его реализации. И чтобы не быть голословными, рассмотрим эти ошибки именно на примере .NET Framework. Все реализации доступны по ссылке: IDisposable Usages

**Класс FileEntry** cmsinterop.cs

Этот код явно написан в спешке, чтобы по-быстрому закрыть задачу. Автор явно что-то хотел сделать, но потом передумал и оставил кривое решение

```
internal class FileEntry : IDisposable
{
    // Other fields
    // ...
    [MarshalAs(UnmanagedType.SysInt)] public IntPtr HashValue;
    // ...

    ~FileEntry()
    {
        Dispose(false);
    }

    // Реализация скрыта и затрудняет вызов *правильной* версии метода
    void IDisposable.Dispose() { this.Dispose(true); }

    // Метод публичный: это серьезная ошибка, позволяющая некорректно разрушить
    // экземпляр класса. Мало того, снаружи этот метод НЕ вызывается
    public void Dispose(bool fDisposing)
    {
        if (HashValue != IntPtr.Zero)
        {
            Marshal.FreeCoTaskMem(HashValue);
            HashValue = IntPtr.Zero;
        }

        if (fDisposing)
        {
            if( MuiMapping != null)
            {
                MuiMapping.Dispose(true);
                MuiMapping = null;
            }

            System.GC.SuppressFinalize(this);
        }
    }
}
```

**Класс SemaphoreSlim** System/Threading/SemaphoreSlim.cs

Эта ошибка в топе ошибок .NET Framework касательно IDisposable: SuppressFinalize для класса, где нет финализатора. Встречается очень часто.

```
public void Dispose()
{
    Dispose(true);

    // У класса нет финализатора - нет никакой необходимости в GC.SuppressFinalize
    GC.SuppressFinalize(this);
}

// Реализация шаблона подразумевает наличие финализатора. А его нет.
// Можно было обойтись одним public virtual void Dispose()
protected virtual void Dispose(bool disposing)
{
    if (disposing)
    {
        if (m_waitHandle != null)
        {
            m_waitHandle.Close();
            m_waitHandle = null;
        }
        m_lockObj = null;
        m_asyncHead = null;
        m_asyncTail = null;
    }
}
```

**Вызов Close+Dispose** Код некоторого проекта NativeWatcher

Иногда люди вызывают и Close, и Dispose. Но это не является правильным (хотя ошибка не вызовется, так как повторный Dispose не приводит к генерации исключения). Вопрос в том, что Close - это еще один шаблон, и введен для того, чтобы людям было понятнее. Но стало непонятнее.

```
public void Dispose()
{
    if (MainForm != null)
    {
        MainForm.Close();
        MainForm.Dispose();
    }
    MainForm = null;
}
```

# Общие итоги

1. IDisposable является стандартом платформы и от качества его реализации зависит качество всего приложения. Мало того, от этого в некоторых

ситуациях зависит безопасность вашего приложения, которое может быть подвергнуто атакам через неуправляемые ресурсы

2. Реализация IDisposable должна быть максимально производительной. Особенно это касается секции финализации, которая работает в параллели со всем остальным кодом, нагружая Garbage Collector
3. При реализации IDisposable следует избегать идей синхронизации вызова Dispose() с публичными методами класса. Разрушение не может идти одновременно с использованием: это надо учитывать при проектировании типа, который будет использовать IDisposable объект
4. Однако стоит защитить одновременный вызов `Dispose()` из двух потоков: это следует из утверждения, что Dispose() не должен вызывать ошибок
5. Реализация оберток над неуправляемыми ресурсами должна идти отдельно от остальных типов. Т.е. если вы оборачиваете неуправляемый ресурс, на это должен быть выделен отдельный тип: с финализацией, унаследованный от `SafeHandle / CriticalHandle / CriticalFinalizerObject`. Это разделение ответственности приведет к улучшенной поддержке системы типов и упрощению проектирования системы разрушения экземпляров типов через Dispose(): использующим типам не надо реализовывать финализатор.
6. В целом шаблон не является удобным как в использовании, так и в поддержке кода. Возможно, следует перейти на Inversion of Control процесса разрушения состояния объектов через шаблон `Lifetime`, речь о котором пойдет в следующей части.