Оглавление

Основы управления памятью

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 с 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. Причем если с кодом все более-менее ясно, то данные можно поделить на несколько больших секций:

- **Thread stack** - 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 85K 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

Управление памятью

Пара слов перед стартом

Когда я разговаривал с различными людьми и рассказывал, как работает Garbage Collector (для меня по началу это было большим и странным увлечением), то весь рассказ умещался максимум минут на 15. После чего, мне задавали один вопрос: «а зачем это знать? Ведь работает как-то и работает». После чего в голове начиналась путаница: с одной стороны я понимал, что они в большинстве случаев правы. И рассказывал про то самое меньшинство случаев, где эти знания прекрасно себя чувствуют и используются. Но поскольу таких случаев было всё-таки меньшинство, в глазах собеседника оставалось некоторое чувство недоверия.

На уровне тех знаний, которые нам давали раньше в немногочисленных источниках, люди, которых собеседуют на позицию разработчика обычно говорят: есть три поколения, пара хипов больших и малых объектов. Еще максимум можно услышать - про наличие неких сегментов и таблицы карт. Но обычно дальше поколений и хипов люди не уходят. И все почему? Ведь **вовсе не потому**, что чего-то не знают, а потому, что действительно **не понятно, зачем это знать**. Ведь та информация, которая нам давалась, выглядела как рекламный буклет к чему-то большому, закрытому. Ну знаем мы про три поколения, ну и что?.. Всё это, согласитесь, какое-то эфемерное.

Сейчас же, когда Microsoft открыли исходники, я ожидал нескольких бенефитов от этого. Первый бенефит - это то, что сообщество накинется и начнет какие-то баги исправлять. И оно накинулось! И исправило все грамматические ошибки в комментариях. Много запятых исправлено и опечаток. Иногда даже написано, что, например, свойство `IsEnabled` возвращает признак того, что что-то включено. Можно даже подать на членство в .NET Foundation, опираясь на множесто вот таких вот комментариев (и, понятное дело, не получить). Сейчас же можно: дорога открыта для граммар-наци. Второй бенефит, который ожидался - это предложение нового и полезного функционала в готовом виде. Этот бенефит, насколько я знаю, время от времени также срабатывает, но это очень редкие кейсы. Например, один разработчик очень ускорил получение символа по индексу в строке. Как выяснилось, ранее это работало не очень эффективно.

Наш рассказ про менеджмент памяти будет идти от общего к частному. Т.е. для начала мы посмотрим на алгоритмы с высоты птичьего полёта, не сильно вдаваясь в подробности. Ведь если мы начнем сразу с подробностей, придётся делать отсылки к будущим главам, а оттуда - обратно - в ранние главы. Это крайне не удобно как для написания, так и в чтении. Напротив, сделав вводную, мы поймем все основы. А потом - начнем погружаться в детали.

Введение в управление памятью

Мы пишем разные программы: консольные, сервисы, web-сервисы и другие. Все они работают примерно одинаково. Но есть очень важное отличие - это стиль работы с памятью. Консольные приложения, скорее всего, работают в рамках базовой, выделенной при старте приложения, памяти. Такое приложение всю или частично ее использует и больше ничего не запросит: запустилось и вышло. Иногда речь идет о сервисах, которые долго работают, перерабатывают память постоянно. И делают это не по изолированным запросам, в отличие от сервисов ASP.NET и WCF (которые мы вызвали, из базы что-то достали и забыли). А именно как какой-то расчётный сервис: есть поток данных на вход, с которыми сервис работает и так может работать очень долго, выделяя и освобождая память. И это уже совершенно другой стиль расхода памяти: ведь в этом случае память необходимо контролировать, смотреть как она расходуется, течет или не течет.

А если это ASP.NET, то это уже третий способ управления памятью. Надо понимать, что нас вызвал внешний код, мы отработаем достаточно быстро и исчезнем. Отсюда, если мы во время запроса выделяем некоторую память, можно сделать всё так, чтобы не волноваться по поводу её освобождения: ведь метод завершит свою работу и все объекты потеряют свои корни: локальные переменные метода, обрабатывающего запрос.

Как же этим всем управлять? С точки зрения разработки Garbage Collector'а, с точки зрения системы менеджмента памяти у нас есть совершенно разные стили и мы должны в них идеально хорошо работать. У нас же может быть машина, на которой запустилось консольное приложение, а есть машина, на которой приложение забирает 256 Гб. Эти системы помимо различия в объёме пожираемой памяти также отличаются по ряду других признаков: например, в стиле её выделения и освобождения путём обнуления ссылок (или их ненужности. при выходе из метода локальные переменные более не нужны, но они не обнуляются). Поэтому, для начала надо как-то классифицировать эту память: а от

классификации памяти танцевать в сторону оптимизации её выделения и освобождения в зависимости от того, с каким классом памяти мы в данный момент работаем.

Возможные классификации памяти исходя из логики

Как можно классифицировать память? Чисто интуитивно можно разделять выделяемые участки памяти исходя из размеров объекта, который выделяется. Например, понятно, что если мы говорим о больших структурах данных, то управлять ими надо совершенно по-другому, нежели маленькими: потому что они тяжелые и их трудно перемещать при надобности. А маленькие, соотвественно, занимают мало места и из-за того, что они образуют группы, перемещать легко. Однако из-за того что их намного больше, ими тяжелее управлять: знать о положении в памяти каждого из них. А значит, для них без всякой статистики и так понятно, что должен быть другой подход.

Если разделять по времени жизни, то тут тоже возникают идеи. Например, если объекты короткоживущие, то, возможно, к ним надо чаще присматриваться, чтобы побыстрее от них избавляться (желательно, сразу, как только они стали не нужны). Если объекты долгоживущие, то можно уже посмотреть на статистику. Например, можно пофантазировать и решить, что эту область памяти анализировать на предмет ненужных объектов можно и пореже: ведь большие объекты редко создают траффик в памяти. А если смотреть редко, это сокращает время на собрку мусора сумме, но увеличивает - каждый вызов GC.

Или же по типу данных. Можно легко предположить, что все типы, которые отнаследованы от типа `Attribute` или в зоне `Reflection`, будут жить почти всегда вечно. Или строки, которые представляют собой массив символов: к ним тоже может быть свой подход.

Видов может быть сколько угодно много и в зависимости от классификаций может оказаться, что управление памятью для конкретной классификации может быть более эффективно, если учитывать её особенности.

Когда создавали архитектуру нашего GC, то выбрали первые два вида классификаций: размер и время жизни (хотя, если присмотреться к делению типов на классы и структуры,

то можно подумать, что классификации на самом деле три. Однако, различие свойств классов и структур можно свести к размеру и времени жизни).

Как это работает у нас. Обоснование выбора архитекторов

Если мы с вами будем досконально разбираться, почему были выбраны именно эти два алгоритма управления памятью: Sweep и Compact, нам для этого придётся рассматривать десятки алгоритмов управления памятью, которые существуют в мире: начиная обычными словарями, заканчивая очень сложными lock-free структурами. Вместо этого, оставив голову мыслям о полезном, мы просто *обоснуем* выбор и тем самым *поймём*, почему выбор был сделан именно таким. Мы более не смотрим в рекламный буклет ракеты-носителя: у нас на руках полный набор документации.

Я выбрал формат рассуждения чтобы вы почувствовали себя архитекторами платформы и сами пришли к тем же самым выводам, к каким пришли реальные архитекторы в штаб-квартире Microsoft в Рэдмонде.

Определимся с терминологией: менеджмент памяти - это структура данных и ряд алгоритмов, которые позволяют "выделять" память и отдавать её внешнему потребителю и освобождать её, регистрируя как свободный участок. Т.е. если взять, например, какой-то массив байт (линейный кусок памяти), написать алгоритмы разметки массива на объекты .NET (запросили новый объект: мы подсчитали его размер, пометили у себя что этот вот кусок и есть новый объект, отдали указатель на объект внешней стороне) и алгоритмы освобождения памяти (когда нам говорят, что объект более не нужен, а потому память из-под него можно выдать кому-то другому).

Исходя из классификации выделяемых объектов на основании их размера можно разделить места под выделение памяти на два больших раздела: на место с объектами размером ниже определенного порога и на место с размером выше этого порога и посмотреть, какую разницу можно внести в управление этими группами (исходя из их размера) и что из этого выйдет. Рассмотрим каждую категорию в отдельности.

Если рассматривать вопросы **управления условно** "*маленьких*" объектов, то можно заметить, что если придерживаться идеи сохранения информации о каждом объекте, нам

будет очень дорого поддерживать структуры данных управления памятью, которые будут хранить в себе ссылки на каждый такой объект. В конечном счёте может оказаться, что для того, чтобы хранить информацию об одном объекте понадобится столько же памяти, сколько занимает сам объект. Вместо этого стоит подумать: если при сборке мусора мы будем помечать достижимые объекты обходом графа объектов (понять это легко, зная, откуда начинать обход графа), а линейный проход по куче нам понадобится *только* для идентификации всех остальных, т.е. мусорных объектов, так ли нам необходимо в алгоритмах менеджмента памяти хранить информацию о каждом объекте? Ответ очевиден: надобности в этом нет никакой. Ведь если мы будем размещать объекты друг за другом и при этом сделать возможным узнать размер каждого из них, сделать итератор кучи очень просто:

```
var current = memory_start;


while(current < memory_end)
{
    var size = current.typeInfo.size;
    current += size;
}
```

А значит, можно попробовать исходить из того, что такую информацию мы хранить не должны: пройти кучу мы можем линейно, зная размер каждого объекта и смещая указатель каждый раз на размер очередного объекта.

В куче нет дополнительных структур данных, которые хранят указатели на каждый объект, которым управляет куча.

Однако, тем не менее, когда память нам более не нужна, мы должны её освобождать. А при освобождении памяти нам становится трудно полагаться на линейное прохождение кучи: это долго и не эффективно. Как следствие, мы приходим к мысли, что надо как-то хранить информацию о свободных участках памяти.

В куче есть списки свободных участков памяти: набор указателей на их начала + размер.

Если, как мы решили, хранить информацию о свободных участках, и при этом при освобождении памяти из под объектов эти участки оказались слишком малы для размещения в них чего-либо полезного, то во-первых мы приходим к той-же проблеме хранения информации о свободных участках, с которой столкнулись при рассмотрении занятых: хранить информацию о таких малышах может оказаться слишком дорого. Это снова звучит расточительно, согласитесь: не всегда выпадает удача освобождения группы объектов, следующих друг за другом. Обычно они освобождаются в хаотичном порядке, образуя небольшие просветы свободной памяти, где сложно выделить что-либо ещё. Но всё-таки в отличии от занятых участков, которые нам нет надобности линейно искать, искать свободные участки нам необходимо потому что при выделении памяти они нам снова могут понадобиться. А потому возникает вполне естественное желание уменьшить фрагментацию и сжать кучу, переместив все занятые участки на места свободных, образовав тем самым большую зону свободного участка, где можно совершенно спокойно выделять память.

Отсюда рождается идея алгоритма сжатия кучи Compacting.

Но, подождите, скажите вы. Ведь эта операция может быть очень тяжёлой. Представьте только, что вы освободили объект в самом начале кучи. И что, скажете вы, надо двигать вообще всё?? Ну конечно, можно пофантазировать на тему векторных инструкций CPU, которыми можно воспользоваться для копирования огромного занятого участка памяти. Но это ведь только начало работы. Надо ещё исправить все указатели с полей объектов на объекты, которые подверглись передвижениям. Эта операция может занять дичайше длительное время. Нет, надо исходить из чего-то другого. Например, разделив весь отрезок памяти кучи на сектора и работать с ними по отдельности. Если работать отдельно в каждом секторе (для предсказуемости времени работы алгоритмов и масштабирования этой предсказмуемости - желательно, фиксированных размеров), идея сжатия уже не кажется такой уж тяжёлой: достаточно сжать отдельно взятый сектор и тогда можно даже начать рассуждать о времени, которое необходимо для сжатия одного такого сектора.

Теперь осталось понять, на основании чего делить на сектора. Тут надо обратиться ко второй классификации, которая введена на платформе: разделение памяти, исходя из времени жизни отдельных её элементов.

Деление простое: если учесть, что выделять память мы будем по мере возрастания адресов, то первые выделенные объекты (в младших адресах) становятся самыми старыми, а те, что находятся в старших адресах - самыми молодыми. Далее, проявив смекалку, можно прийти к выводам, что в приложениях объекты делятся на две группы: те, что создали для долгой жизни и те, которые были созданы жить очень мало. Например, для временного хранения указателей на другие объекты в виде коллекции. Или те же DTO объекты. Соответственно, время от времени сжимая кучу мы получаем ряд долгоживущих объектов - в младших адресах и ряд короткоживущих - в старших.

Таким образом мы получили *поколения*.

Разделив память на поколения, мы получаем возможность реже заглядывать за сборкой мусора в объекты старшего поколения, которых становится всё больше и больше.

Но возникает еще один вопрос: если мы будем иметь всего два поколения, мы получим проблемы:

- Либо мы будем стараться, чтобы GC отрабатывал маскимально быстро: тогда размер *младшего поколения мы будем стараться делать минимальных размеров*. Как результат - недавно созданные объекты при вызове GC будут случайно уходить в старшее поколение (если GC сработал "прям вот сейчас, во время яростного выделения памяти под множество объектов"), хотя если бы он сработал чуть позже, они бы остались в младшем, где были бы собраны за короткие сроки.
- Либо, чтобы минимизировать такое случайное "проваливание", мы *увеличим размер младшего поколения*. Однако, в этом случае GC на младшем поколении будет работать достаточно долго, замедляя и подтормаживая тем самым всё приложение.

Выход - введение "среднего" поколения. Подросткового. Суть его введения сводится к получению баланса между *получением минимального по размеру младшего поколения* и *максимально-стабильного старшего поколения*, где лучше ничего не трогать. Это - зона, где судьба объектов еще не решена. Первое (не забываем, что мы считаем с нуля) поколение создается также небольшим, но чуть крупнее, чем младшее и потому GC туда заглядывает реже. Он тем самым дает возможность объектам, которые

находятся во временном, "подростковом" поколении, не уйти в старшее поколение, которое собирать крайне тяжело.

Так мы получили идею трёх поколений.

Следующий слой оптимизации - попытка отказаться от сжатия. Ведь если его не делать, мы избавляемся от огромного пласта работы. Вернемся к вопросу свободных участков.

Если после того, как мы израсходовали всю доступную в куче память и был вызван GC, возникает естественное желание отказаться от сжатия в пользу дальнейшего выделения памяти внутри освободившихся участков, если их размер достаточен для размещения некоторого количества объектов. Тут мы приходим к идее второго алгоритма освобождения памяти в GC, который называется `Sweep`: память не сжимаем, для размещения новых объектов используем пустоты от освобожденных объектов.

Так мы описали и обосновали все основы алгоритмов GC.

Далее спускаться мы не будем, иначе я не оставлю себе почвы для размышлений. Замечу только, что мы рассмотрели все предпосылки и выводы к существующим алгоритмам менеджмента памяти.

Как это работает у нас

Теперь мы зайдём с другой стороны. Я буду выполировывать факты так, что даже если у вас плохая память на вычитке текста, вы всё равно запомните, как работает менеджмент памяти в .NET.

Итак, мы знаем, что у нас существует два способа классифицировать память: исходя из времени жизни сущностей и исходя из их размера. Подумаем, что мы имеем, исходя из размеров объектов. Если у нас объекты имеют большие размеры, то нам не выгодно часто делать `Compact`. Потому что в этом случае мы все объекты перетаскиваем на освободившиеся участки. То есть копируем их. А если объект огромен, то копировать дорого и каждый раз прибегая к сжатию кучи можно сильно просесть по производительности. В данной ситуации удобен только `Sweep`. Об этом способе мы подробно поговорим позже, но если совсем коротко, то память освобождается и свободный кусок сохраняется в список свободных участков и дальше переиспользуется при

выделении объектов. Вызов Compact может быть выгоден в редких случаях: когда *есть понимание*, что из-за Sweep и траффика буферов большого размера существует некоторая фрагментация в зоне крупных объектов (Large Objects Heap. Давайте уже начнём называть вещи своими именами). И ручной вызов GC с указанием метода сбора мусора Compact в этом случае может нам помочь. Однако, давайте не будем углубляться: у нас для этого есть очень много времени.

Остановимся лишь на том, что в LOH метод сбора мусора Sweep имеет абсолютное преимущество перед Sweep

А если объекты маленькие, то наоборот: хоть и не всегда, но удобен Compact. Например, у нас была куча объектов и мы потеряли ссылку на объекты через одного, и получается, что занятые участки и свободные чередуются, например по 24 байта. И может так оказаться, что такими маленькими участками мы воспользоваться не сможем потому что дальше мы будем аллоцировать более крупные объекты. Возникает дилемма: либо наращивать кучу либо избавиться от фрагментации. Поэтому тут, возможно, стоит сжать кучу. Однако, если объекты ушли на покой группой, то нам в данном случае по-прежнему выгоден Sweep, поскольку тот не сжимает кучу, а использует для дальнейшей аллокации освободившееся место.

Отсюда можно сделать простой вывод: в обоих кучах память управляется одинаково. Но в LOH в отличии от SOH отключен автоматический вызов Compact. Он доступен только для прямого вызова.

Тут возникает проблема: у нас есть хип маленьких объектов и хип больших. Они, соответственно, разделены. Но по факту мы всегда аллоцируем маленькие объекты. Их в любом случае будет больше: возможно, миллионы. GC проходит через разные стадии: стадия планирования, стадия маркировки, сбора мусора. На 200гб памяти, если так получится, любая из стадий будет очень дорогой, а потому память надо как-то дополнительно сегментировать, чтобы оптимизировать работу с ними.

Поэтому, второй тип сегментации работает исходя из времени жизни объектов. Куча растет у нас в одном направлении. Аллокация идет с младших адресов к старшим. Соответственно, при выделении памяти берется указатель на первый свободный участок и затем он сдвигается на размер выделенного объекта и всё: куча растет в одном

направлении. Отсюда, используя наши ранние рассуждения можно сделать вывод о том, как легко поделить на три поколения. Нулевое поколение - место, где объекты аллоцируются. Первое поколение - это место, где объекты не были собраны Garbage Collector, но в них пока еще нет уверенности: мы хотим созранить стабильность во втором поколении и даём объектам еще один шанс быть собранными. И, соответственно, второе поколение, где объекты в идеале будут находиться без сборки мусора.

Как я уже говорил, если объекты живут долго, то туда можно реже заглядывать, чтобы запускать GC. Поэтому, если мы сделаем нулевое поколение определенных, малых и заранее известных размеров, мы сможем обходить его за *гарантированно короткий* промежуток времени. И Microsoft гарантирует, что нулевое поколение будет собираться за какие-то определенные миллисекунды. Т.е. GC быстро что-то сделал и дальше пошел, а никто даже и не заметил, что он отработал.

LOH имеет другую структуру. Сюда попадает все, что больше или равно 85 тысячам байт. Цифра странная, но нам полезно ее знать: её можно использовать, например, чтобы определить размер для аллоцируемого массива, чтобы он не ушел в кучу больших объектов. Если аллоцируете массив int-ов, то нужно грубо поделить на четыре и получится примерно 20 тысяч int-ов, которые туда прекрасно лягут и не уйдут в LOH. Также интересно, что в LOH уходят массивы double от тысячи элементов и выше.

Проверить это все очень легко. Можно написать такой код:

```
var arr1 = new double[999];  // -> Gen 0

var arr2 = new double[1000]; // -> Gen 2
```

и первый массив пойдет в нулевое, а второй - во второе поколение. Какие еще классификации типов существуют для GC? У нас есть две основных, которые мы только что рассмотрели. Однако, хоть эта классификация в общем смысле нам и не доступна, она нам доступна в узком смысле: классификация по типу. Все мы знаем про интернированные строки. Если мы предполагаем, что какая-то строка будет часто встречаться, то ее стоит интернировать, тогда мы сэкономим на памяти.

Как они хранятся? Если строка интернирована, то она хранится как обычная строка в куче. Но ее надо как-то найти, чтобы проверить, что точно такая же строка уже существует. Куча иногда может достигать нескольких сотен гигабайт и поиск будет очень дорогим

решением. Поэтому интернированные строки хранятся отдельно (предполагается, что их будет не так много).

У каждого домена при этом (системного или с базовым типом BaseDomain) есть внутренние таблицы, которая нам не доступны. Среди прочих существует Large Heap Handle Table. Существует два типа внутренних массивов, основанных на bucket-ах. Это массив статиков. Имеются ввиду статические члены классов, которые хранятся в массивах. У каждого домена есть ссылка на массив статиков. И дальше ячейками являются ссылки на значения. И еще одна - pinning handles. Это таблица запиненных элементов. Для тех случаев, когда вы пинуете объект в памяти, можете сделать это двумя путями. Первый - это через API, а второй - через ключевое слово fixed в C#. Это два совершенно разных механизма.

Соответственно, исходя из времени жизни, из-за большого количества объектов, SOH разделен на части, чтобы им было проще управлять. Заполнение SOH идет линейно, поэтому старые объекты живут в младших адресах, свежие - в старших. Старые объекты, как правило, живут долго. Чем дольше объект существует, тем больше вероятность того, что он будет существовать все время работы приложения. Поэтому существует три поколения. Нулевое поколение - это между временем создания объекта и ближайшим GC. Объектов не успевает накопится слишком много и GC успевает их быстро убрать, не залезая в остальную кучу.

Первое поколение живет между первым и вторым GC. Соответственно, для тех, кто не ушел во второе. Оптимизация какая: нулевое собирать быстро, первое - чуть подольше. Это последняя возможность GC собрать объект, прежде чем он ушел во второе, огромное, поколение. Если объект ушел во второе поколение, то, скорее всего, он будет жить долго. Туда можно редко обращаться. А первое - для объекта, который случайно ушел в первое поколение, но на самом деле он короткоживущий. Это такая оптимизация, чтобы его во втором не ловить. И второе поколение для тех, кто решил пожить подольше и если GC туда пришел, то он там останется работать надолго.

Оранжевые кубики - это руты. Руты - это точки, относительно которых, если обходить граф, то гарантированно вы обойдете все объекты, которыми пользуется программа. Если бы фаза маркировки работала на всех поколениях, то она бы работала долго. Поэтому она работает максимально на самых младших. Если мы решили, что собираем нулевое

поколение, то она только там и будет работать. Поэтому есть три типа ссылок. Первый тип - ссылка внутри одного поколения. Например, если мы с рута пришли в нулевое поколение, а дальше у нас ссылка из этого объекта, но опять в нулевое поколение. Это внутренняя ссылка. Второй тип ссылок - это ссылка из более старшего поколения в более младшее поколение. Older-to-younger. Он характеризуется тем, что объект, на который мы ссылаемся из первого поколения нет имеет ссылки с рута в нулевом поколении. Это значит, что если мы будем маркировать только нулевое поколение, чтобы на нем GC отработал, то мы пропустим этот объект. Мы должны знать о существовании ссылки с более старшего поколения.

Третий вид ссылок - это ссылка из младшего в старшее поколение. Для нас она не важна. Если мы собираем нулевое поколение - она не имеет значение. При сборке более старших поколений, младшие тоже собираются. Почему? Если собираем, например, первое - оно больше, крупнее. И, поскольку, нулевое поколение собирается намного чаще, то есть высокая степень вероятности, что после сборки первого будет собрано и нулевое. И GC опять запустится. Для того, чтобы два раза не ходить за одним и тем же, пересобираются и более младшие поколения. В этом случае нам нужно знать ссылку из младшего в старшие? Нам это без разницы, она и так есть. При сборке второго поколения та же самая ситуация. С точки зрения фазы маркировки нам важны ссылки внутри поколения и ссылки с более старших на наше поколение.

Если мы находимся в нулевом поколении - как узнать о том, что на нас есть ссылка с более старшего поколения. Есть механизм, который в начале изучения начинает немного пугать. Есть такой код (см. слайд) 35:19.

Кстати, такие сценарии работы GC можно проверять таким способом: мы создаем объект, делаем GC.Collect(), отправляем его в первое поколение. Мы знаем, что он туда уйдет. Дальше создаем ссылку и тем самым фактически создаем ссылку из старшего поколения в младшее. Если дома захочется с чем-то поиграть, то с помощью метода GC.Collect() можно смоделировать такие ситуации.

Что мы здесь видим? У нас есть x, GC.Collect(), инстанс класса Foo уходит в поколение один из нулевого. И дальше x.field присваиваем new Boo(). Это значит, что объект типа Foo

начинает ссылаться на новый инстанс объекта типа Boo. То есть из первого в нулевого. Это у нас older-to-younger link.

Что происходит в .NET. В месте присваивания, джиттер, то есть там не просто присваивание, а присваивание с проверкой. Эта техника называется Write Barrier и Remembered Set. В .NET она называется немного по-другому. Если у нас звезды сошлись, что нужно запомнить эту ссылку, то мы запоминаем ее во внутренней структуре джита.

Что это за условие? Значение - это ссылка на экземпляр .NET класса. Точка присваивания находится в управляемой куче и имеет более старшее поколение, чем адрес присваиваемого объекта. Когда мы делаем вот так (см. слайд 37:29), джиттер дополнительно проверяет, что слева поколение старше, чем справа. Если старше, то он запоминает адреса во внутренних структурах, убеждается, что с левой части на правую есть ссылка. Это нужно для дальнейшей сборки мусора. На фазе маркировки отмечается выбранное поколение и если у нас есть ссылки в Remembered Set, если мы запомнили, что где-то сохраняли ссылку из первого в нулевое поколение, они тоже становятся корнями, чтобы пройти маркировку. Но хип получается в итоге огромный и становится страшновато.

Поэтому используется более оптимизированный способ. Он называется механизм карточного стола. Знания об этом механизме не так распространены. Как он работает? Если взять адресное пространство всего огромного хипа, весь кусок памяти, то сбоку есть карточный стол. Это, грубо говоря, массив чисел. Где каждый бит массива отвечает за определенный диапазон памяти. Если бит выставлен в единицу, значит в этом диапазоне памяти есть ссылка на младшее поколение. То есть это массив признаков ссылок на младшее поколение. См. слайд - 40:12

У нас есть память, внизу карточный стол. У нас появилась ссылка, слева на право. Это значит, что бит должен быть выставлен в единицу. Потому что от более старшего поколения пошла ссылка в более младшее. Каждый бит карточного стола отвечает за 128 байт в x86 и за 256 байт в x64. Это, по сути, 32 машинных слова. Машинное слово - это то, с чем работает процессор.

Если учесть, что каждый пустой объект, максимально маленький (например, new Object) занимает четыре машинных слова в среднем, то получается, что один бит карточного стола перекрывает десять объектов. И если хотя бы с одного из них есть ссылка в младшее

поколение, то GC должен при обходе в фазе маркировки зайти по этому адресу и просмотреть все десять объектов и найти те, которые ссылаются на младшее поколение. Один байт перекрывает уже 1,2 КБ оперативной памяти. Или 80 объектов. Четыре байта - 320 объектов. Это x86 архитектура. Получается жирновато, если ссылка появилась.

Как это работает. Можно посмотреть код, который будет вызван при присваивании (см. слайд 43:24) по этому адресу на github. Там ассемблеровский код, он достаточно простой, разобраться в нем легко. Есть много комментариев, гораздо больше, чем кода.

Реализация сильно зависит от особенностей. У нас есть Workstation GC, есть серверный GC. У Workstation есть две версии: до и после роста кучи. Как следствие, перемещается gen_1 gen_0. И Server GC: есть несколько хипов для SOH и несколько для LOH. Там свои реализации этих методов присваивания, потому что придется параметризовать для какой кучи идет вызов. А так он просто генерирует ставку для новой кучи и все хорошо. Плюс две реализации под x64. Если смотреть базовую, то будет примерно так (см. слайд 44:36). Регистр RCX - это адрес filed. Адрес таргета, куда мы присваиваем. RDX - это ссылка на объект. Когда мы присваивали, должна быть составлена эта инструкция и больше ничего. Но на самом деле нет. Присвоили и дальше этим кодом мы проверяем, находится ли правая часть присваивания внутри эфемерного сегмента (gen_0, gen_1). И если находится, то мы берем карточный стол, делим на 2048, получаем адрес ячейки и если флаг не выставлен, то выставить.

Здесь есть две особенности. Первая - почему просто не выставить? Это будет очень долго. Операция записи намного дольше, чем чтения. Чтение происходит из кеша, а чтобы записать надо записать кроме кеша еще и в оперативную память. Поэтому их проверяем. Интересна процедура выставления флага. Он выставляется сразу же 0FF. То есть мы выставляем не один флаг, а сразу группой. Почему? Когда мы будем дальше проверять ссылку из старшего поколения в младшее, нам побитово будет долго проверять. Проще сразу словами делать проверку. Вместо того, чтобы смотреть, на каком бите ссылка и какие 2 КБ смотреть, все работает проще, система оперирует более значительными диапазонами.

Код, получается, проверяет только поколение object, но не target. Target не интересует. Мы проверяем только то, что мы попали в нулевое или первое поколение. В любом этом случае выставляется бит в карточном столе. Когда мы проверяем нулевое поколение, будем

проходится по карточному столу, который относится и к первому, и ко второму поколению. Нас устроит, что биты выставлены. Если первое поколение будем просматривать с нулевым, собирать там мусор и у первого и второго, то мы будем просматривать карточный стол второго поколения. Там тоже эти биты будут выставлены. Поэтому мы левую часть смотрим, и не имеет значения первого или второго поколения. Дальше фильтрация уже идет на стадии проверки.

Однако, карточный стол в случае большого хипа (у LOH он может быть феерических размеров) тоже будет огромным. И по нему точно так же будет идти сканирование. Мы собираем нулевое поколение и должны уложиться в несколько миллисекунд, а у нас хип в несколько сотен ГБ. Например, это сервер. Карточный стол придется сканировать весь, а это долго. Выход - двухуровневый карточный стол. Называется это Cards Bundle Table. Это еще один массив, где бит отвечает за 32 слова карт. Этот массив оперирует огромными диапазонами. Получается, 1 бит Cards Bundle Table отвечает за 128 Кб на x86 и за 256 Кб на x64. Одна ячейка - 4 байта, это 8 Мб карт целевой памяти.

Когда мы будем делать GC, собирая нулевое поколение, надо посмотреть, что с первого и второго есть что-то полезное и далее мы уходим в Cards Bundle Table. Сканиурем, и если где-то не ноль, то переходим на карточный стол, в соответствующий его диапазон и ищем ненулевую ячейку. И потом уже переходим в нужный диапазон памяти и сканиурем объекты, которые там находятся, в поисках ссылки со старшего на младшее поколение. И только тогда мы эту ссылку добавляем в руты и маркируем все объекты, на которые эта ссылка ведет.

Есть маленькая оптимизация для Windows. Эта система построена, в первую очередь, на архитектуре x86, где используются механизмы вирутализации памяти процессора, которая основана на страницах памяти. Она поделена на зоны, на странички. Можно выставить флаг MEM_WRITE_WATCH. Это означает, что если кто-то будет писать по заданному диапазону, то можно подписаться на обновления. Мы сделали массив и если туда кто-то будет писать, у нас будет дергаться метод из winapi. Почему мы не видим этого когда в ассемблеровском методе расстановки карт? Когда мы записываем, выставляя карты, Windows получает нотификацию от страницы, куда мы пишем, и исходя из этого проставляет бит в Cards Bundle Table.

Базовый вывод, который можно сделать уже сейчас, основываясь на карточных столах, что если вы хотите, чтобы GC протекал быстро и как по маслу, не стоит делать ссылок из старших поколений. Не надо делать вечноживущие массивы, аллоцировать объекты и ссылки на них складывать в эти древние массивы. Но если вы так делаете, надо контролировать, чтобы эти массивы располагались рядом, чтобы их аллоцировать друг за другом. Не распределять их по памяти. Самое неудачное, что можно сделать - это иметь кучу объектов старшего поколения и по какой-то причине выставить ссылки на младшее поколение, но не группой, а вразброс через всю память. Это самый тяжелый сценарий, потому что карточный стол будет забит единицами. А GC, собирая нулевое поколение, будет вынужден проходить все второе, все первое и искать, что там добавлено. Если вы делаете ссылку из старших поколений в младшие, то необходимо эти ссылки группировать: массив, который ссылается на объект младшего поколения. Поскольку это массив, который ссылается на объекты младшего поколения, все ячейки рядом и в карточном столе в идеале это будет просто единица. Дальше мы будем изучать более подробно.

Стек потока

Ссылка на обсуждение

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". Причём 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, разберём элементарный пример кода:

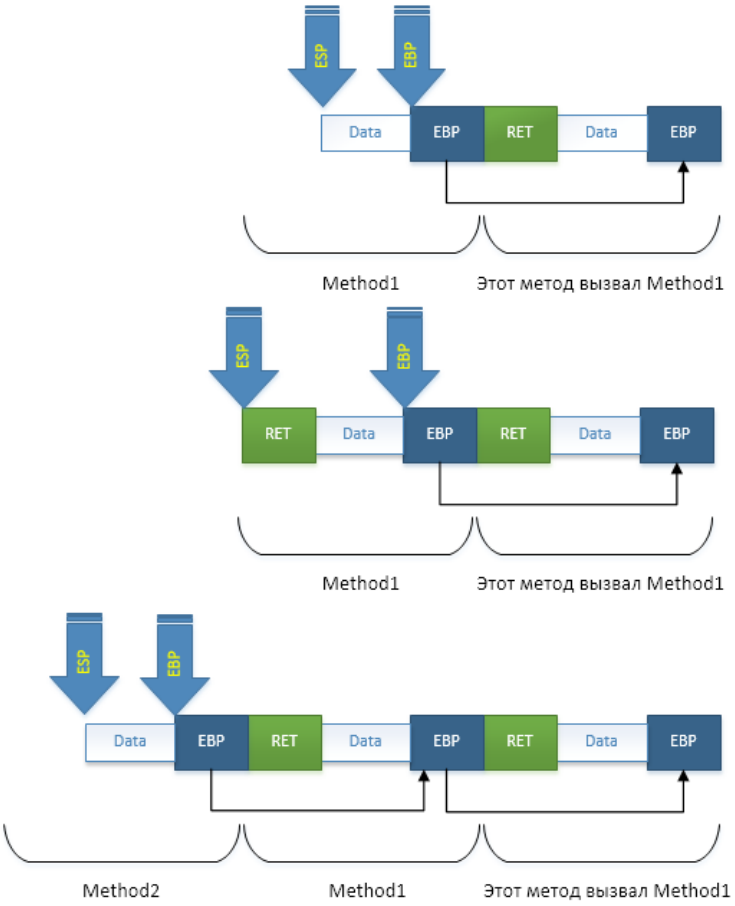
```
void Method1()
{
    Method2(123);
}

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 передаётся указатель на тот объект, с которым работает метод. Т.е. указатель `this`. Так что в этих (почти всех) случаях для регистров остаётся 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 подсчёте 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 создаём 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 её 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 "растёт", 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 её 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 - своё - методов… А работа при этом часто идёт 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 ещё 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 начнётся в методе `Method1`. Однако если ничего не случится, то `Method3` завершит свою работу, управление перейдёт 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 живёт 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 кладётся 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 ещё раз глянем на структуру стека потока, которая по своей сути - обычный массив. Диапазон памяти, в котором строятся фреймы, организован так, что он растёт с конца в начало. Т.е. более поздние фреймы располагаются по более ранним адресам. Так же, 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 даёт 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 отдаёте, 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 определённого 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 дошёл. 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 вернёт 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:

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

Ну и наверняка что-то ещё, о чем мы можем не знать. Да и знать нам всего для примера нет никакой надобности: в промышленное использование данный код не пойдёт, 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();
}
```

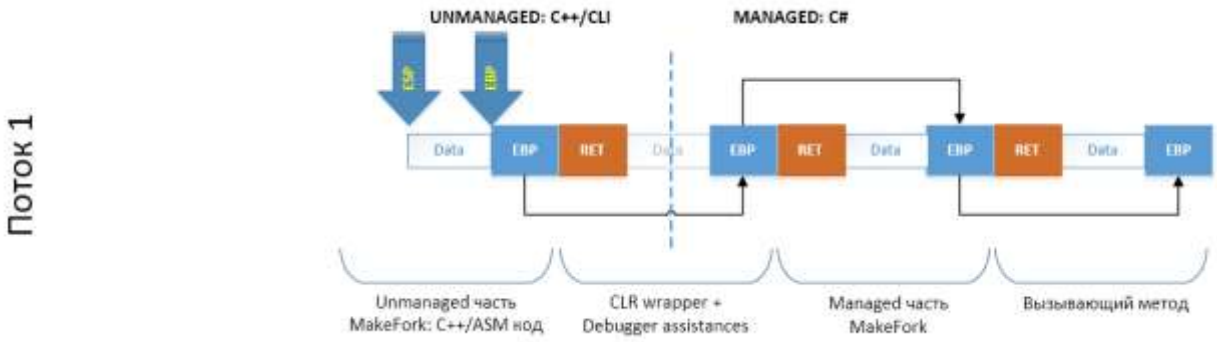
Когда вызовется MakeFork(), что мы ожидаем с точки зрения стек трейсов? Что в родительском потоке все останется без изменений, а дочерний будет взят из пула потоков (для скорости), в нем будет сымитирован вызов метода `MakeFork` вместе с его локальными переменными, а код продолжит выполнение не с начала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

```
// Parent Thread
```

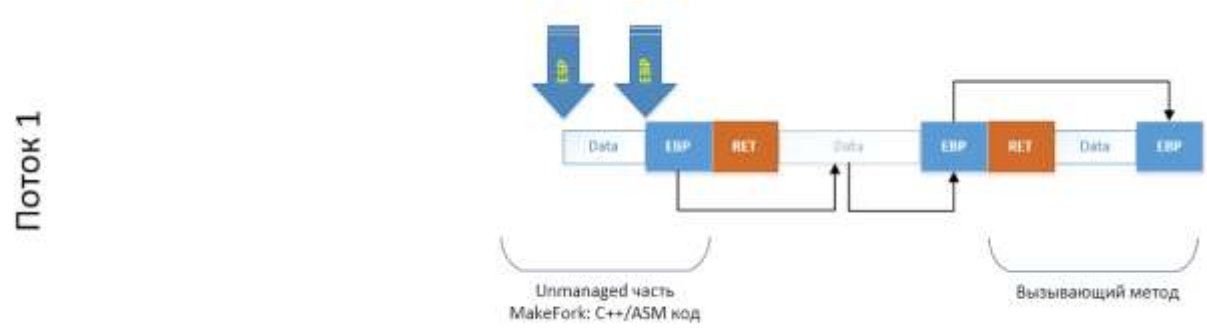
```
RootMethod -> MakeFork


// 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). Причём, 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.

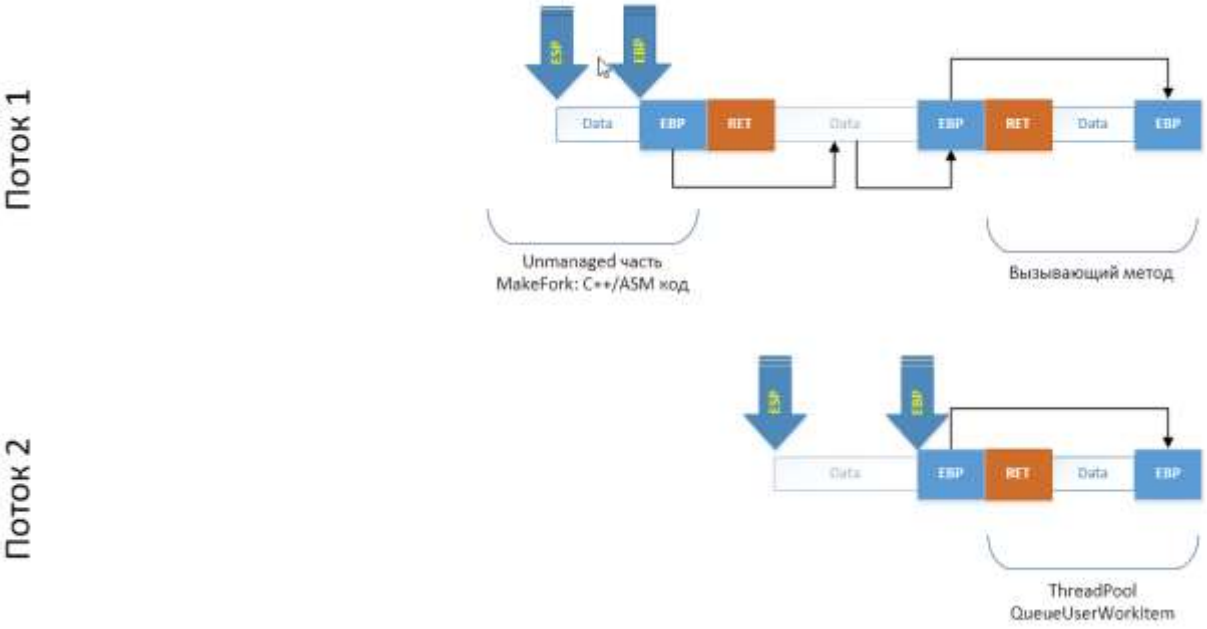
Итак, как будет выглядеть стек потока, когда наш код вызовет MakeFork, который вызовет CloneThread, который уйдёт в unmanaged мир CLI/C++ и вызовет метод клонирование (саму реализацию) - там? Давайте посмотрим на схему (ещё раз напомню, что стек растёт от старших адресов к младшим. Справа налево):



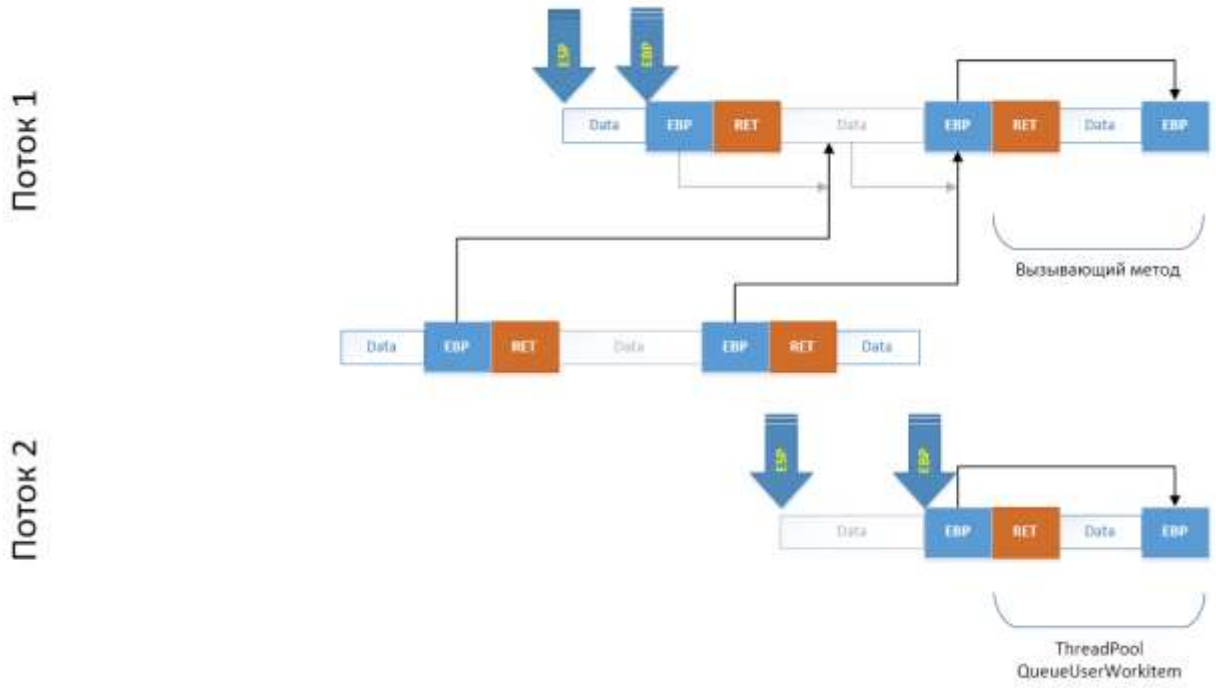
Ну а для того чтобы не тащить всю простыню со схемы на схему, упростим, отбросив то, что нам не нужно:



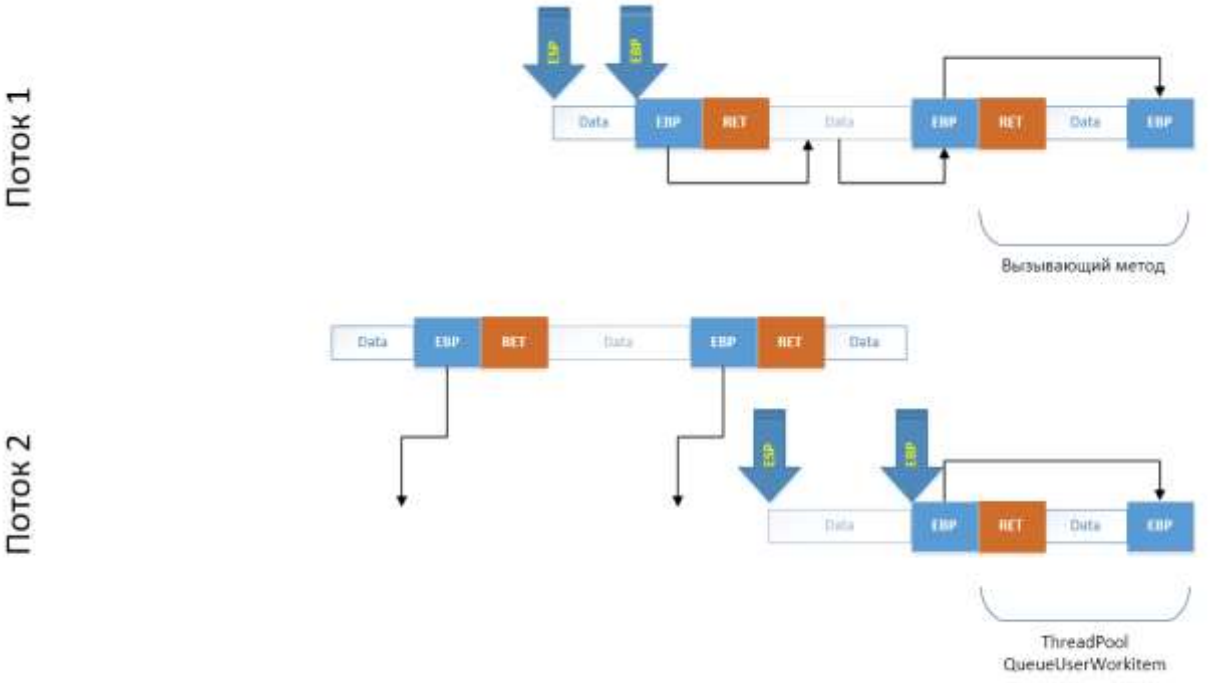
Когда мы создадим поток либо возьмём готовый из пула потоков, в нашей схеме появляется ещё один стек, пока ещё ничем не проинициализированный:



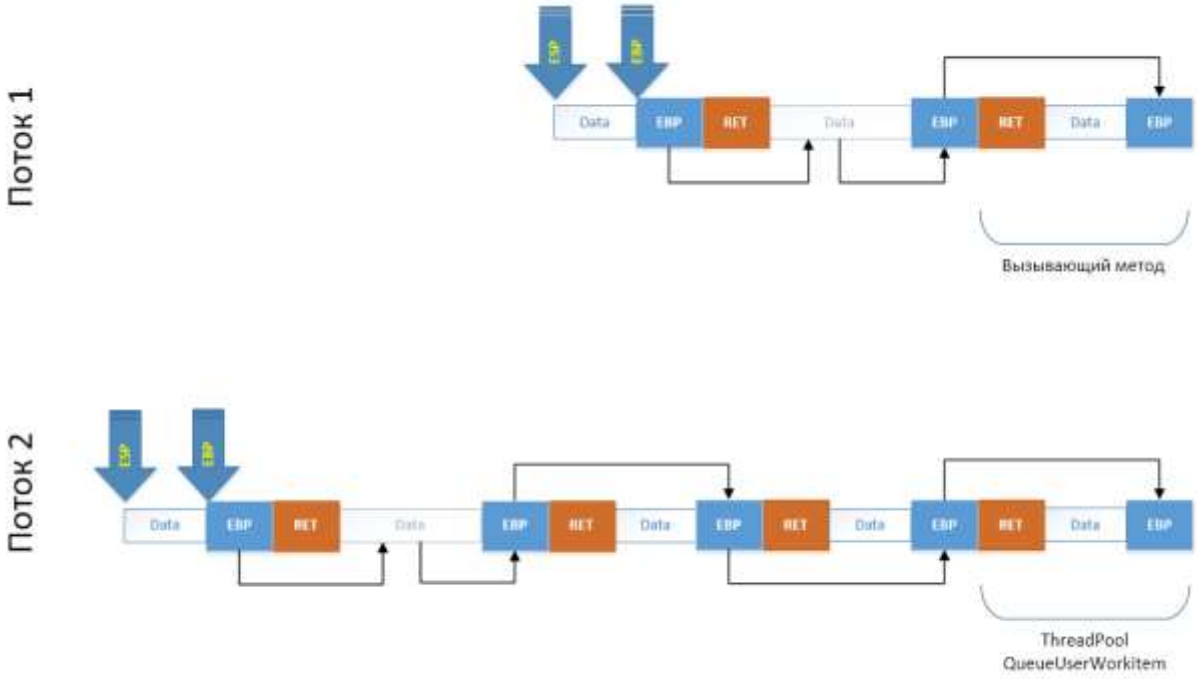
Теперь наша задача - сымитировать запуск метода `Fork.CloneThread()` в новом потоке. Для этого мы должны в конец его стека потока дописать серию кадров: как будто из делегата, переданного ThreadPool'у был вызван `Fork.CloneThread()`, из которого через враппер C++ кода managed обёрткой был вызван CLI/C++ 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 приведёт к возврату в реальный wrapper CLI/C++ вызова, который вернёт 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 стека необходимо у себя рисовать из родительского потока в дочерний. Метод создания потока оборачивает в делегат вызов unmanaged метода, передаёт 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 пойдёт чуть ниже. Также, чтобы вы понимали, о чем идёт 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 обёртку 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 произойдёт 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 достаём это число из стека и проверяем: там лежит `0`? Если да, мы находимся в том же самом потоке: а значит у нас ещё 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 ещё мы, а значит надо подготовить данные для дочернего потока. Чтобы более не спускаться на уровень ассемблера, работать мы будем со структурой ранее сохранённых регистров. Достанем из неё значение регистра EBP: он по сути является полем "Next" в односвязном списке кардов стека. Перейдя по адресу, который там содержится, мы очутимся в кадре метода, который нас вызвал. Если и там возьмём первое поле и перейдём по тому адресу, то окажемся в ещё более раннем кадре. Так мы сможем дойти до managed части `CloneThread`: ведь мы сохранили адрес переменной в её стековом кадре, а значит, прекрасно знаем, где остановиться. Этой задачей и занимается цикл, приведённый 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 ещё один переход по односвязному списку: `*(int *)curptr`. После чего создаём 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);
```

Ещё 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 попадёт обратно в этот метод, который, завершив работу, освободит поток и вернёт 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 идём 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 подойдёт.

```
    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, восстанавливаем и их самих.

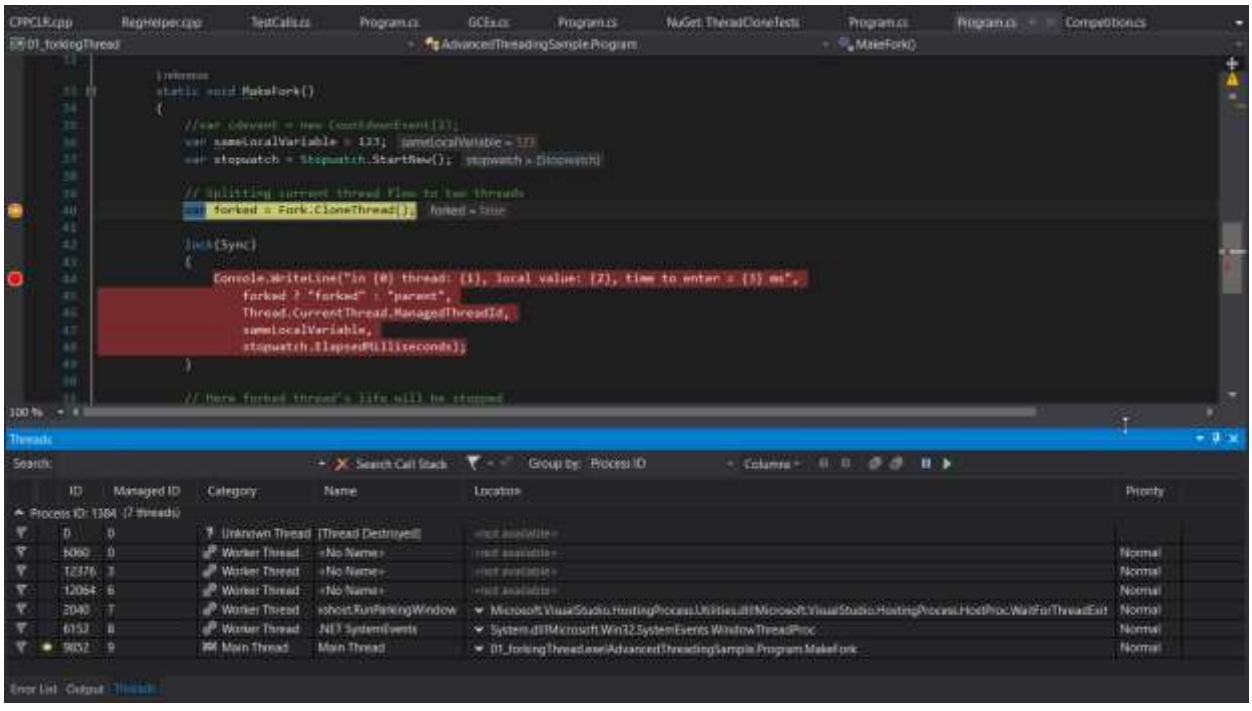
```
    // restore registers, push 1 for Fork() and jmp
    _asm {
        push copy.EAX
        push copy.EBX
        push copy.ECX
        push copy.EDX
        push copy.ESI
        push copy.EDI
        pop EDI
        pop ESI
        pop EDX
        pop ECX
        pop EBX
        pop EAX
```

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 ещё тут. Когда мы туда попадём, то ForkImpl распознает смену потока и осуществит выход в метод MakeFork, который,

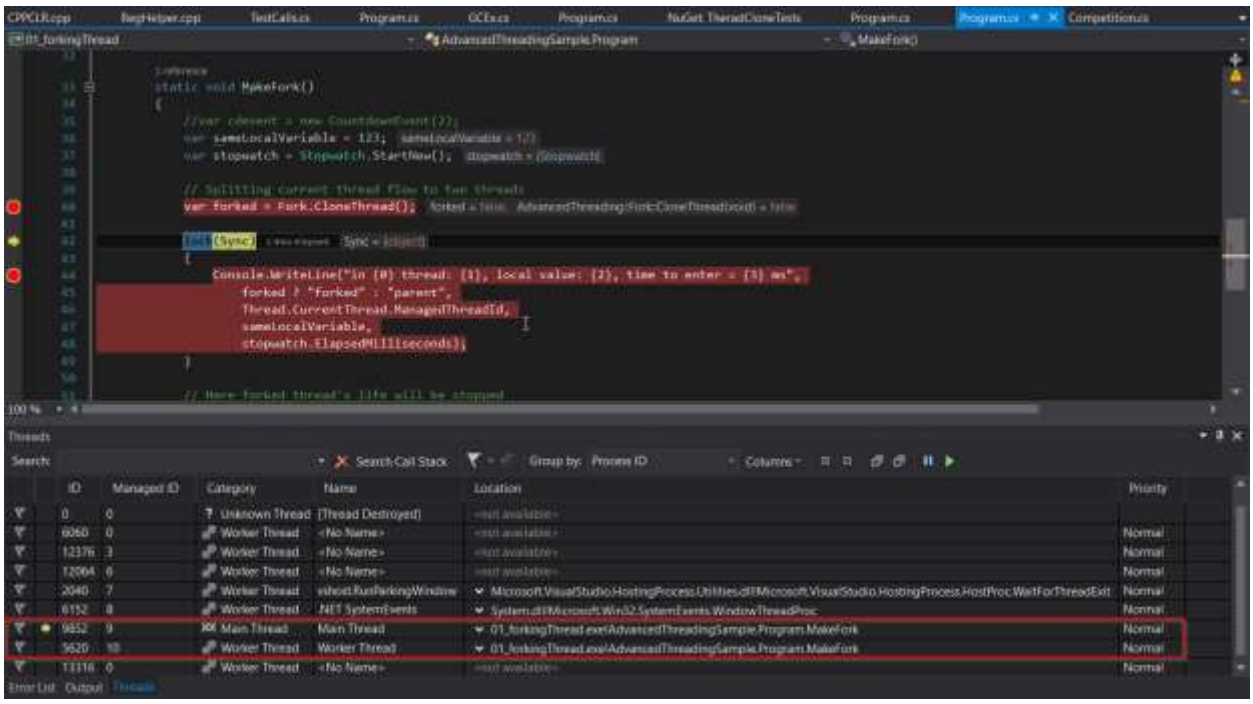
завершив работу, попадёт в точку RestorePointAfterClonnedExited, т.к. немного ранее мы сымтировали вызов MakeFork из этой точки. Восстановив регистры до состояния "только что вызваны из TheadPool", мы завершаем работу, отдавая поток в пул потоков.

```
        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 ещё более низкий уровень, то узнаем или же вспомним, что память на самом деле является виртуальной и что она поделена на страницы объёмом 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 её частью и также состоит из страниц объёмом 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 уйдёт 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: тёмная сторона `stackalloc` - unsafe код. Тот результат, что он возвращает не является managed указателем: значение - обычный указатель на участок не защищённой памяти. Причём если по этому адресу сделать запись уже после того, как метод завершил работу, вы начнёте писать в локальные переменные некоторого метода или же вообще перетрёте 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 начнём.

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 найдём 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 объёмы 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 ещё 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 ещё 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 всё-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 её в куче. Причём, 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 своё свободное время интересной задачкой. Однако понимание его структуры даёт 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.

Время жизни сущностей

Один из вопросов, которые могут очень сильно влиять на производительность наших приложений - это политика выбора типа сущности: класс или структура, и контроль за их временем жизни. Ведь архитекторы платформы не просто так выделили для нас эти два типа данных. Это деление в первую очередь обусловлено возможностями оптимизации приложений, архитектура которых учитывает особенности обоих групп типов.

Как уже было сказано в главе Ссылочные и значимые типы данных, огромным преимуществом значимых типов данных является то, что их не надо аллоцировать. Т.е. другими словами если кто-то располагает экземпляр значимого типа в локальных переменных или параметрах метода, то расположение идёт на стеке (не выделяя дополнительной памяти в куче). Эта операция - та, о которой вам надо мечтать, т.к. именно она максимально быстра и эффективна. Если же структура располагается в полях ссылочного типа (класса), то под нее операция выделения памяти также не вызывается: ведь она является структурной частью этого ссылочного типа. Однако всё гораздо сложнее с ссылочными типами. Ведь, если речь идет о них, то мы имеем целый набор сложностей при выделении памяти под их экземпляры. Причём, что самое печальное, а может быть даже обидное - от нас почти никак не зависит, на какой из алгоритмов выделения памяти мы напоремся: на самый быстрый вариант из четырёх или же на самый тяжеловесный.

Ссылочные типы

Общий обзор

Для целостности картины при дальнейшем чтении рассмотрим особенности времени жизни экземпляров ссылочных типов данных. Ссылочные типы обладают следующими свойствами в вопросе времени собственной жизни:

- У ссылочных типов в отличии от значимых детерменированное начало жизни. Другими словами они порождаются тогда и только тогда, когда кто-либо запросил их создание;
- Однако, они имеют недетерменированный освобождение: мы не знаем, когда произойдет освобождение памяти из под них. Мы не можем вызвать GC для

конкретного экземпляра даже для случая с Large Objects Heap, где эта операция могла бы быть вполне уместна;

Эти два свойства дают нам немного пищи для размышлений:

- экземпляры классов уничтожаются в случайное время в неопределенно отдаленном будущем;
- их уничтожение обуславливается утерей ссылок на них;
- поэтому с одной стороны это значит, что операция освобождения последней ссылки на объект превращается в детерменированную операцию "удаления" объекта из зоны видимости приложения. Он ещё есть, существует, но недосягаем для всего остального приложения;
- однако, с другой стороны мы далеко не всегда в курсе, какое именно обнуление ссылки будет последним, что лишает нас свойства детерменированности в обнулении последней ссылки.

Еще одним очень важным свойством является наличие виртуального метода финализации объекта. Этот метод вызывается во время срабатывания сборщика мусора: т.е. в неопределенном будущем. И необходим данный метод для одного: корректного закрытия неуправляемых ресурсов, которыми владеет объект в тех и только тех случаях, когда что-то пошло не так (например, было выброшено исключение) и программа более не сможет самостоятено это сделать (код, который отвечает за освобождение данных ресурсов никогда более не вызовется вследствие срабатывания исключительной ситуации). И, поскольку время вызова данного метода ровно как и освобождение памяти из под объекта от нас не зависят, его вызов также не является детерменированным. Мало того, он является асинхронным, т.к. осуществяется в отдельном потоке во время исполнения приложения. Это важно помнить, т.к. если, например, ваше приложение имеет логику повторной попытки работы с ресурсом и если произошла какая-то ошибка (например, `ThreadAbortException`), в результате которой ресурсы "повисли" в очереди на финализацию, то это значит, что вы не сможете открыть этот ресурс (например, файл), пока не отработает очередь на финализацию, в которой этот ресурс будет освобождён.

Однако, там где есть неопределенность, программисту всегда хочется внести определенность и как результат, возник интерфейс `IDisposable`, речь о котором пойдет чуть позже, в следующей главе. Я сейчас могу сказать лишь одно: он реализуется если

необходимо, чтобы внешний код мог самостоятельно отдать команду на освобождение ресурсов объекта. Т.е. детерменированно сообщить объекту, что он более не нужен.

В защиту текущего подхода

Мы никогда не задумывались (а может только я?) над тем, что было бы, будь всё по-другому: если бы память освобождалась детерменированно. Текущий подход с автоматической памятью, когда мы не задумываемся, где выделять объекты и когда их освобождать нам не всегда нравится: ведь бывают случаи, когда готовых к освобождению объектов накапливается слишком много и их освобождение тормозит всё приложение. Однако, в защиту текущего подхода давайте немного отвлечемся на сценарии, о которых иногда начинаешь задумываться, мечтая сменить текущий набор алгоритмов:

- если вместо того чтобы освобождать объекты по срабатыванию GC мы будем освобождать их с потерей последней ссылки, что произойдёт? Вот наш код присваивает некоторой переменной `null`. Тогда получается, что на каждом присвоении необходимо проверять, идёт ли присвоение `null` или какой-либо другой рабочей ссылки. Если да, то надо понять, последняя ли это была ссылка. Каждый раз считать входящие с кучи ссылки, перебирая все объекты SOH/LOH - дорого. Значит, надо чтобы каждый объект считал все входящие ссылки сам: инкрементируя и декрементируя счётчик на каждой операции. Это - дополнительное место + дополнительные действия. Плюс ко всему получается, что мы уже не можем сжимать кучу: после каждого присваивания это делать слишком дорого: подходит только метод `Sweep`. Как мы видим, уже на поверхности всплывает очень много проблем, не говоря уже о подробностях;
- если ввести оператор `delete`, чтобы как в C++ освобождать объекты по требованию, дополнительно воскрешает деструктор, как средство детерменированного освобождения памяти: ведь если мы освобождаем объект оператором `delete`, необходимо таким же образом освобождать те объекты, которыми этот объект владеет. Значит, необходим метод, который будет вызываться при разрушении объекта: деструктор экземпляра типа. Это приведет к увеличению сложности разработки и удорожанию сопровождения программ: утечки будут постоянно. Плюс ко всему прочему возникнет путаница при освобождении памяти: мы лишаемся

возможности освобождать её в последней точке использования. Т.е. теперь мы должны это делать в строго отведенном месте.

- если вводить смешанный алгоритм: в целом чтобы работало как сейчас, но чтобы был оператор `delete`. Например, вы мне скажете, вам захочется освобождать массивы данных, которые были использованы под скачивание изображений ровно в определенный момент. Потому что если наше приложение качает изображения друг за другом и при этом они достаточно быстро становятся не нужны, то мы вхолостую выделяем кучу памяти, которая быстро копится и приводит к вызову GC. Это особенно актуально для мобильных приложений на Xamarin и элемента управления "виртуальный список", где при быстром скролле изображений они в больших количествах грузятся, а потом становятся ненужными. Если удалять их сразу, то не будет ситуации с большим GC, который испортит анимацию прокрутки. Однако, тут возникнут сложности для GC. При ручном освобождении памяти, последняя в свою очередь станет фрагментирована и может перестать вмещать в себя те массивы данных, которые вы запросите под следующие изображения. Как следствие - всё равно произойдет GC. Если блоки памяти с ручным управлением располагать в LOH, то ручное освобождение хорошо "ляжет" на его алгоритмы. Однако, всё также будет приводить к фрагментации и дальнейшему срабатыванию полного GC. Единственно верное решение - использовать пул массивов и Span/Memory для доступа к поддиапазону индексов. Но тогда зачем вводить `delete`?

Тогда получается, что текущее решение - прекрасно и надо просто научиться им правильно пользоваться. Этим мы чуть позже и займемся.

Предварительные выводы

Из всего сказанного можно увидеть, что у любого объекта есть некоторое время его существования. Это может показаться тривиальной мыслью, которая лежит на поверхности, но не все так однозначно:

- важно понимать, как, когда и при каких иных условиях *объект создается*. Ведь его создание занимает некоторое не всегда короткое время. И вы не можете заранее угадать, по какому алгоритму он будет создан: простым переносом указателя в

случае наличия места в allocation context, вследствие необходимости расширения или переноса allocation context, необходимости сжатия эфимерного сегмента или же необходимости создания нового эфимерного сегмента с его полным структурированием;

- также стоит понимать, насколько "популярным" будет объект *во время его жизни* и как долго он будет существовать: какое количество иных объектов будет на него ссылаться и как долго. Этот фактор влияет как на сборку мусора, фрагментацию кучи, время создания других объектов и что самое интересное - на время обхода графа объектов в фазе маркировки достижимых объектов сборщиком мусора;
- а также, что логично и очень важно: как объект будет *достигать* состояния освобождения (состояние выброшенности звучит грустно). Это значит, будет ли осуществляться детерменированное его разрушение или нет. Например, при помощи `IDisposable.Dispose`
- и освобождаться - быть подхваченным Garbage Collector'ом с дальнейшей возможностью вызова финализатора.

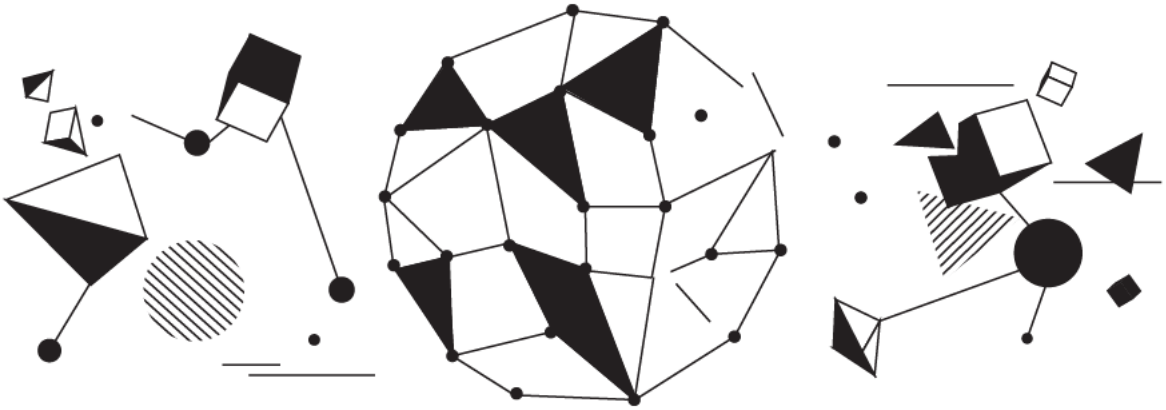
Каждый из этих этапов определяет производительность приложения и логичность его архитектуры. Рассмотрим каждый этап в отдельности.

Создание объекта

Все начинается с операции запроса памяти к подсистеме управления памятью:

```
var x = new A();
```

Эта казалось бы самая простая операция платформы .NET кроет в себе огромный пласт работы, который будет описан в отдельной главе.



Шаблон Disposable (Disposable Design Principle)

Ссылка на обсуждение

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 всё 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 и делает так, чтобы мы вообще не задумывались о том, как её чистить, то становится не совсем понятно, зачем её вообще заниматься этим вопросом.

Однако есть нюансы. Существует некоторое заблуждение, что `IDisposable` сделан, чтобы освобождать неуправляемые ресурсы. И это только часть правды. Чтобы одномоментно понять, что это не так, достаточно вспомнить примеры неуправляемых ресурсов. Является ли неуправляемым класс `File`? Нет. Может быть, `DbContext`? И опять же - нет. Неуправляемый ресурс - это то, что не входит в систему типов .NET. То, что не было создано платформой, и находящееся вне её 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 (вы только просите, а открывает его всё-таки оперционная система). Т.е. все управляющие структуры (такие как координаты файла на файловой системе, его фрагменты в случае фрагментации и прочая служебная информация, номера цилиндра, 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 передаёт приложению дескриптор неуправляемого ресурса, но дополнительно сохраняет его в таблице открытых дескрипторов процесса. Сохраняя при этом 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 узнаёт. 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 выделяете слишком много памяти и не будете её 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 придётся делать запрос администраторам на `iisreset` либо ручное закрытие ресурсов при помощи специализированного ПО. Таким образом, решение этой проблемы может стать не тривиальной задачей на удалённом 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 создаёте экземпляр сущности, чтобы быстренько с ней поработать в рамках одного метода, либо в рамках времени жизни экземпляра этой 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 освобождать именно тогда, когда необходимо нам: 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 пойдём 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 создаём экземпляр некоторого ресурса, который должен быть освобождён: 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 случай 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 освобождён в любом случае. Ведь `DisposableResource` у нас - управляемый, а значит, .NET CLR о нём 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 произойдёт в этом случае?

Если мы используем неуправляемые ресурсы, получается, что у нас опять же два варианта: когда всё хорошо и метод Dispose вызвался (тогда всё 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 уйдёт в рекурсию и в некоторой точке вылетит по `StackOverflowException`, то те ресурсы, которые были захвачены и не были освобождены, забудутся рантаймом .NET. 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 неопределённое 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 идёт не только об исключениях, обрушающих поток приложения. Тут уже речь идёт вообще о любых исключениях, которые приведут к тому, что алгоритм не дойдёт 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 нужно собрать. Максимум что он сделает - при очередном проходе поймёт, что на граф объектов, содержащих наш объект типа `FileWrapper`, потеряна последняя ссылка и память перетрётся 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 всё 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 произойдёт.

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 ещё 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 серьёзная проблема, которая не даст ему работать так, как задумали мы. Если мы вспомним, как отрабатывает процесс сборки мусора, то заметим одну деталь. При сборке мусора GC *в первую очередь* финализирует всё, 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 уйдём на "последнюю милю". Тем не менее, более высокоуровневый объект может пытаться работать с объектом, который хранит неуправляемый ресурс - в своём 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 кеширования 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 приведёт 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 унаследуете 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 скомпилированы и будут безопасными с точки зрения вызова при нехватке ресурсов.

Почему это так важно? Зачем тратить так много усилий на тех, кто уйдёт в мир иной? А всё 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 сетевого диска, тот будет заблокирован удалённым хостом и отпущен либо по тайм-ауту, либо когда вы освободите ресурс, закрыв файл. Если ваше приложение вылетит в момент открытого файла, то он не будет закрыт даже после перезагрузки. Придётся ждать достаточно продолжительное время для того, чтобы удалённый хост отпустил бы его. Плюс ко всему вам нельзя допускать выброса исключений в финализаторах - это приведёт 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 ещё может быть освобождён. 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 перёд этим давайте попробуем перечислить всё, что к нам *обычно* идёт из unmanaged мира:

- Первое и самое ожидаемое, что оттуда обычно идёт, - 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 обернёт 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 всё 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 чём 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, счётчик 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

    // определённых известных значений. Обратите внимание, что метод опасен по двум причинам:

    //   - Если дескриптор отмечен как недопустимый с помощью SetHandleasInvalid, DangerousGetHandle

    //    то всё равно вернёт исходное значение дескриптора.

    //  - Возвращённый 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 ненадёжный 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.
```

```
    // Причём если возвращаемое значение = false, будет брошено исключение
    //      SafeHandleCriticalFailure,      которое      в      случае      включённого
SafeHandleCriticalFailure
    // Managed Debugger Assistant войдёт в точку останова.
    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
    protected abstract bool ReleaseHandle();



    // Работа со счётчиком 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 счётчик внешних ссылок, которые не могут быть учтены CLR. Т.е. ссылками из unsafe кода. Вручную увеличивать и уменьшать счётчик нет почти никакой необходимости: когда вы объявляете любой тип, производный от SafeHandle, как параметр unsafe метода, то при входе в метод счётчик будет увеличен, а при выходе - уменьшён. 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 счётчиком же ссылок всё проще: SafeHandle не будет собран, пока дополнительно не обнулится счётчик. Вот почему вручную менять счётчик не стоит. Либо это надо делать очень аккуратно: возвращая его, как только это становится возможным.

Второе назначение счётчика 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 счётчик 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 счётчик 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 счётчик опять же увеличится, а при выходе - уменьшится, что даст нам гарантию существования handle на всё 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 всё, что с ним связано. Наш тип значительно упрощается.

Срабатывание finalizer во время работы экземплярных 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 касаться. 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:

```
// 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 перевёл 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 серьёзная. Возможна попытка закрытия файла финализатором класса во время чтения из файла. Что практически гарантированно приведёт к ошибке. Причём поскольку в данном случае ошибка будет именно возвращена (`IntPtr == -1`), то мы этого не увидим, `_handle` будет обнулён, 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 счётчики использования в unmanaged мире, так ещё и эти счётчики автоматически увеличиваются при передаче в unmanaged методы и уменьшаются - при выходе из него.

Многопоточность

Теперь поговорим про тонкий лёд. 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 определённом состоянии, и любое действие над ним переводит его

состояние в одно из заранее определённых - 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 потокобезопасный. Есть потенциальная возможность вызова публичных методов этих типов в то время, как идёт разрушение объекта. Давайте решим эту проблему и решим, стоит ли вообще её 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 экземпляра 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 приведёт к сильному проседанию по производительности. Всё дело в том, что взятие блокировки занимает некоторое время. Это время необходимо для аллокации таблиц SyncBlockIndex, проверок на текущий поток и много чего ещё (мы рассмотрим всё это отдельно - в разделе про многопоточность). Т.е. получается, что ради "последней мили" жизни объекта мы будем платить производительностью всё время его жизни!
- Дополнительный memory traffic для объектов синхронизации
- Дополнительные шаги для обхода графа объектов при GC

Второе, и на мой взгляд, самоё важное. Мы допускаем ситуацию одновременного разрушения объекта с возможностью поработать с ним ещё разок. На что мы вообще должны надеяться в данном случае? Что не выстрелит? Ведь если сначала отработает Dispose, то дальнейшее обращение с методами объекта обязано привести к `ObjectDisposedException`. Отсюда возникает простой вывод: синхронизацию между вызовами Dispose() и остальными публичными методами типа необходимо делегировать 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 идёте 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 полшаблона 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 счётчик 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
    - Реализуют IDisposable.Dispose путём разрушения инкапсулированных объектов в порядке: сначала типы Level 0, потом типы Level 1
    - Т.к. они не имеют неуправляемых ресурсов - то не реализуют finalizer
    - Должно содержать protected свойство, дающее доступ к Level 0 типам.

Именно поэтому я с самого начала ввёл разделение на два типа: на содержащий управляемый ресурс и содержащий неуправляемый ресурс. Они должны работать совершенно по-разному.

Как ещё 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 всё не совсем так. Мы всегда можем получить ситуацию, когда мы теряем ссылку на объект и думаем, что всё хорошо: GC соберёт мусор, а вместе с ним и наш объект. Однако, выясняется, что память растёт, мы лезем в программу анализа памяти и видим, что на самом деле этот объект удерживается чем-то ещё. Всё 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 не явности захвата, программист может пропустить необходимость её 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` отдаёт 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 делегат 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 ещё, а на `Primary` потеряем ссылку, то GC **не соберёт** объект `Primary`, т.к. его будет удерживать `Secondary`. Как избежать таких неприятных ситуаций? Необходим детерминированный подход к освобождению ссылки на нас. И тут к нам на помощь приходит механизм, который прекрасно подходит для наших целей: `IDisposable`

```
// Для простоты указана упрощённая 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 приемлемо: 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 затрёт 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 пойдёт 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` чтобы ввести управление над коллекцией делегатов.

Кстати, тут возникает ещё одна неявная ситуация: может показаться, что если вы присвоите источнику событий null, то дальнейшая подписка на события приведёт 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 неясность в языке C# может запутать программиста: работа с обнулёнными 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 приведёт. 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 создаёт 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`):

```
// Создаём из лямбды делегат
Action action = () => service.SendMessageAsync(MessageType.Deploy);

// Подписываемся
button.Clicked += action;

// Создаём 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 переменных будет использование обычных приватных методов.

Защита от ThreadAbort

Когда разрабатывается библиотека для внешнего разработчика, вы никак не можете гарантировать, как она себя поведёт в чужом приложении. Иногда остаётся только догадываться, что такого с нашей библиотекой сделал чужой программист, что появился тот или иной результат её работы. Один из примеров - работа в многопоточной среде, когда вопрос целостности очистки ресурсов может встать достаточно остро. Причём, 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 `ThreadAbortException`, 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 приведёт 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;
    }
}
```

защищён 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:

1. Основным плюсом шаблона является возможность детерминированного освобождения ресурсов: тогда, когда это необходимо
2. Введение общеизвестного способа узнать, что конкретный тип требует разрушения его экземпляров в конце использования
3. 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, отдаёт тем самым команду всем, кто его будет использовать: используя меня, вы принимаете публичную оферту. Причём 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 придётся 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, встаёт 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 детерминированность, Dispose() не означает прямого разрушения объекта. Объект всё ещё 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 ещё 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. Причём она 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 идёт 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 ещё 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, чтобы их собрал Garbage Collector, вносит огромное количество путаницы 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 ещё не все вопросы, связанные с этим, казалось бы, простым шаблоном. Последним вопросом у нас идёт 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, посвящённом 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 ждёт 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 всё равно, в каком порядке мы должны финализировать ресурсы. Задерживать выгрузку домена и приложения мы не можем: наша задача всё 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 серьёзная 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 ещё один шаблон, и введён 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

Однако стоит защитить одновременный вызов `Dispose()` из двух потоков: это следует из утверждения, что Dispose() не должен вызывать ошибок

Реализация обёрток 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 приведёт к улучшенной поддержке системы типов и упрощению проектирования системы разрушения экземпляров типов через Dispose(): использующим типам не надо реализовывать финализатор.

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 пойдёт в следующей части.

Memory<T> и Span<T>

Ссылка на обсуждение

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 невозможным 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, которые наследуют 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 раздвиганием 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 экземпляров 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

```
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();
}
```

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]

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 пулинг таких буферов (ожидается частый трафик 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()
{
```

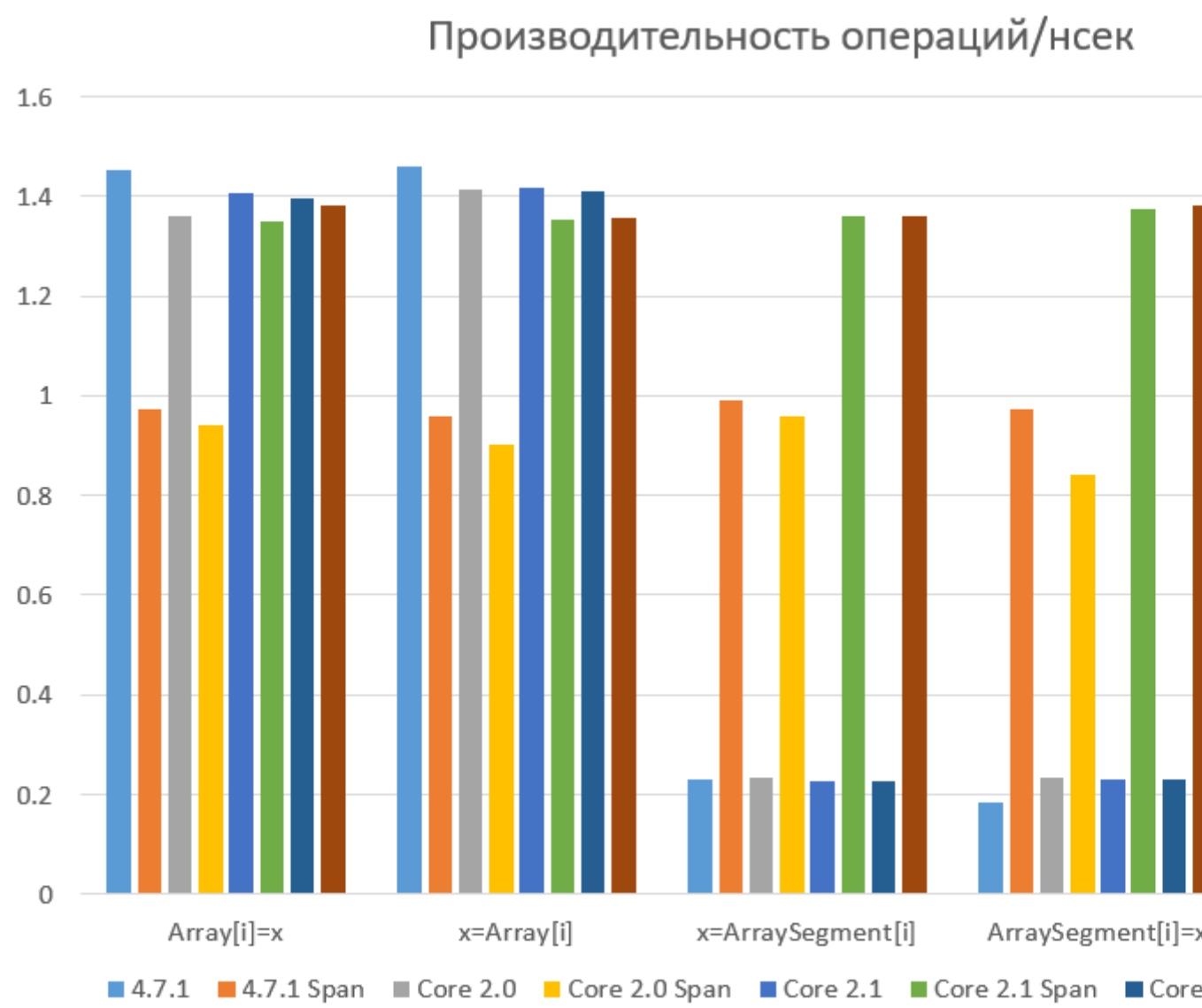
```
        var tmp = 0;
        var accessor = (IList<char>)segment;
        for (int index = 0, len = accessor.Count; index < len; index++)
        {
            tmp = accessor[index];
        }
        return tmp;
    }

    [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 почерпнуть 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.

Структура объектов в памяти

Ссылка на обсуждение

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 трёх 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 ещё. 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 займёмся.

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. Причём 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 её наполнения надо вычислять, где находятся все её 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 достаём 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 объёмы.

Но не будем отвлекаться. Итак, чтобы получить размер экземпляра класса, чей размер фиксирован, 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 (выровне
Int32	12	Boxed Int32: SyncBlk + VMT + данные
Int64	16	Boxed Int64: SyncBlk + VMT + данные
Char	12	Boxed Char: SyncBlk + VMT + данные (выровне
Double	16	Boxed Double: SyncBlk + VMT + данные
IEnumerable	0	Интерфейс не имеет размера: надо брать obj.G
List<T>	24	Не важно сколько элементов в List, занимать хранит данные он в array, который не учитыва
GenericSample<int>	12	Как видите, generics прекрасно считаются. Ра данные находятся на том же месте, что и у bo VMT + данные
GenericSample<Int64>	16	Аналогично
GenericSample<IEnumerable>	12	Аналогично

Тип или его определение	Размер	Комментарий
GenericSample<DateTime>	16	Аналогично
string	14	Это значение будет возвращено для любой размер должен считаться динамически. Од размера под пустую строку. Прошу заметить, ч по разрядности: по сути, это поле использоват
int[]{1}	24554	Для массивов в данном месте лежат совсем д размер не является фиксированным, потому е 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:

1. Если вы хотите знать, сколько займёт значимый тип как значение, используйте `sizeof(TType)`
2. 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 ещё 2 слова (`2 * sizeof(IntPtr)`). Или же взять это значение из VMT типа.
3. Расчёт выделенного объем памяти в куче представлен для следующих типов:
   i. Обычный ссылочный тип фиксированного размера: мы можем забрать размер экземпляра из VMT;
   ii. Строка, необходимо вручную считать её размер (это вообще редко когда может понадобиться, но, согласитесь, интересно)
   iii. 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   |   Term    |

  ------------------------------------------------------------------------------------------------

  |  4 / 8 байт  |  4 / 8 байт   |    4 байта     |    4 байта     |  2 байта |  2 байта |  2 байта  |

  ------------------------------------------------------------------------------------------------

  |      -1      |  0xXXXXXXXX   |        3       |        2       |     a    |     b    |   <nil>   |

  ------------------------------------------------------------------------------------------------



  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
```

Расчёты 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 рассчитать размер массивов в памяти.

Массивы

Строение массивов несколько сложнее: ведь у массивов могут быть варианты их строения:

1. Они могут хранить значимые типы, а могут хранить ссылочные
2. Массивы могут быть как одномерными, так и многомерными
3. 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 неё 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 учётом
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 расчёта 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 неопределённого размера: массив или строка. Неопределённого потому, что его размер будет определён при создании. С generic типами на самом деле все просто: для каждого конкретного generic типа создаётся 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 ещё 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 вперёд. 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 её работу не даёт 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 окунёмся 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 вернёмся 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 ещё. 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 struc
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 передаётся 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 создаёте экземпляр некоторого унаследованного типа, то в его VMT по местам, где у базового типа находятся виртуальные методы, компилятор расположит указатели на переопределённые 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 создаёт таблицу правильного размера и наполнения сразу же: в нашем примере чтобы построить таблицу для типов, которые будут делать таблицу более крупной за счёт добавления новых

методов, придётся 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. Причём, 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`. Причём 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 : IF**
-> GetValue()	SampleMethod()	RunProcess
-> SetValue()	Go()	-> GetValu
	-> GetValue()	-> SetValue
	-> SetValue()	LookToMo

VMT всех трёх 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 создаётся 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. Назовём 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 создаётся 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 ещё 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 четырёх 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 счёте станет очень много, и в силу того, что большую часть из них JIT никогда создавать не будет, т.к. наличие типов во Framework ещё 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 четырёх состояниях: состояние заглушки типа "метод ещё 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 определён 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 её выглядит следующим образом:

```
DWORD: Количество типов =
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 №
DWORD: Количество слотов для типа №1 = M
DWORD: bool: смещения могут быть отрицательными
DWORD: Слот №1
DWORD: Целевой слот №1
...
DWORD: Слот №M
DWORD: Целевой слот №M
```

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" clas

//   - 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 берётся 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 ещё 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 идёт через таблицу VMT, но из-за того, что индексы заранее известны, на каждый вызов дополнительно приходится лишь единственное разыменование указателя. 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 определённость 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 ещё 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()
```

Причём, 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 следующий 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 приведёт к вызову переопределения при вызове по интерфейсу, 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 идёт прямое нарушение этого принципа и введение ещё одного: множественного наследования. Я, честно, сильно против таких доработок, но... Я что-то ушёл в сторону.

Выделение памяти под объект

[In Progress] адаптация курса

Мы только что поговорили про GC с высоты птичьего полета. Сейчас будем уходить в подробности. В первую очередь, хочется пообщаться про алгоритм выделения памяти.

Управление памятью .NET разработчика интересует со стороны двух основных процессов: первое - это выделить память, второе - освободить. Про выделение памяти мы сейчас и поговорим.

С точки зрения процессора память выглядит несколько сложнее. Если говорит про архитектуру управления памяти, я бы сказал, что она разделена на слои. Первый слой, который видим мы, слой .NET достаточно сложный, но верхоуровневый. То есть он в любом случае опирается на что-то другое, а именно на слои управления памятью Windows, который в свою очередь запущен на каком-то процессоре. Процессор по-своему интерпретирует эту память. Таким образом, у нас фактически существует три слоя управления памятью.

Память с точки зрения процессора - это планка, физическая память, RAM. Сколько планок мы поставили, столько он и видит. Но у нас есть знания, что каждый запущенный в системе Windows процесс изолирован. И кроме себя он больше ничего не видит. Есть сам процесс, winapi и больше ничего вообще. Word не видит Exel. Exel не видит калькулятор. Все они изолированы полностью. Потому что на архитектуре Intel сделана виртуализация памяти.

Как она работает. Есть таблицы глобальных дескрипторов, которые мы не видим, есть таблицы локальных дескрипторов, которые куда-то отсылаются. И все вместе это ссылается на Page Frame.

У любого процесса есть некий диапазон памяти от нуля и до, например, 4Гб на x32 системе и до условной бесконечности на x64. Это диапазон виртуальной памяти. Процессор создает для конкретной программы некий виртуальный кусок, где есть только она. Эта область поделена на страницы: на кусочки по 4Кб. Каждый их них означает, что память в этом месте либо существует, либо ее там не существует, т.е. она физически в этом месте отсутствует полностью. Адрес есть, а памяти там нет. Это возникает в ситуациях, когда планка, например на 4Гб, вставлена в материнскую плату, а процессов у нас запущена сотня и

Windows 32х разрядный. Значит, что у каждой программы 4Гб памяти, но при этом они друг от друга изолированы.

Как 4 Гб поделить на сто и получить 4 каждому? Никак. Это значит, что у каждой программы на четырехгигабайтном участке есть места, где памяти не вообще, там только выделены кусочки адресов. Но где-то память есть и она замаплена на планку физическую. А если не замаплена - памяти нет. Если туда обратиться, что-то попытаться считать, вы получите не ноль, а AccessViolationException. Исключение, которое говорит о том, что вы пытаетесь работать с куском, который не существует, либо на который у вас нет прав. На данном уровне исключение различий не делает.

Точно также работать с Wap. (см. слайд 06:30). На изображении выделены процессы в виде столбиков. Там, где белый шум - память выделена и заполнена. А там, где пропуски - памяти нет. Выделенные области могут быть замаплены как на физическую память, так и на жесткий диск.

Если, например, программа много памяти выделила, но ей не пользуется, а соседней программе память нужна, при этом планка маленькая, то Windows неиспользуемую память отгружает на жесткий диск, а этот кусок отдает кому-то другому.

Итак, в нашем .NET приложении часть памяти используется, а часть - нет. Как это можно посмотреть? Можно запустить утилиту, например есть Procmon. Они показывают, как приложение расходует память. Там видно строчки - это как раз выделенные участки. Первый столбец - это адрес участка, второй указывает хип, третий - размер участка. И еще один столбик - это commited. Что это значит? Если программа решила, что ей нужен хип, и пытается определить какой именно ей создавать. Она резервирует память под хип, и часть ее решает закоммитить. Виртуальное адресное пространство разделено на страницы. У страниц есть три статуса. Первый - свободная страница, на которой ничего нет. Ее можно брать и использовать как нужно. Второй статус - страница зарезервирована. Это значит, что какая-то часть программы под себя эту страницу зарезервировала, но там все еще нет памяти, она там отсутствует. Но страница зарезервирована для будущих нужд. Это делается для того, чтобы в аллоцированом хипе никто другой по середине себе ничего не выделил.

Может показаться, что вы как хозяин своего приложения, как хотите, так и выделяете память. На самом деле нет. Если вы чисто в .NET - то может быть да. А вообще говоря, .NET

часто вызывает Managed code, которому в свою очередь тоже нужна память. И он не будет выделять память в хипах от .NET - SOH и LOH. Он о них не имеет понятия.

Он выделяет память в своих хипах. Если это C++, то это C++ Runtime Library. Но там есть свои механизмы построения хипов. Получается, что вы не полностью контролируете виртуальный адресный диапазон, который вам выдали. Чтобы обезопасить себя, вы сразу резервируете адресное пространство и внутри него уже начинаете аллоцировать память - коммитить ее, чтобы она стала существовать, чтобы туда можно было что-то писать.

На слайде 11:50 видны свободные места и есть непрерывный участок, частично зарезервированный, частично - закоммиченный. Это место может быть в оперативной памяти, либо находится на жестком диск, потому что им давно не пользовались. Если вы обращаетесь к куску, который находится на жестком диске, то он автоматически подгружается и становится куском из оперативной памяти. Но вам это знать уже не надо.

Зная все это, разберем как работает `var x = new A();`

Вернемся к базовым знаниям, то, что мы обычно рассказываем на собеседованиях. Когда мы общаемся с программистами, нас спрашивают, как все работает. Мы рассказываем, что есть три поколения. Как память выделяется? Указатель смотрит на свободный участок и когда делается операция new, то мы этот адрес отдаем а указатель перемещаем на размер выделенного блока. А когда происходит GC, куча сжимается и этот указатель оказывается раньше, потому что куча сжалась. Примерно так и происходит. На слайде 13:31 у нас есть закоммиченная часть - слева до вертикальной черты. Эта память существует физически. Справа от черты - это зарезервированная за хипом часть, но ее пока не существует. Дальше мы должны вспомнить о таком понятии как allocation context. Это некое скользящее окно, внутри которого в данный момент менеджер памяти выделяет память под объекты. И уже внутри него у нас есть указатели на начало свободного места. Слева уже пошли объекты.

Когда идет операция new A(), мы запоминаем этот адрес, размещаем там объект и передвигаем адрес начала свободного места, отдавая адрес в переменную x. Наш код выглядит примерно следующим образом (см.слайд 14:39). У нас есть метод Allocate(), мы попросили у него некий amount памяти. Он проверяет, есть ли у нас будущий адрес свободного места (адрес начала места плюс этот amount). Если он превысит heap size, у нас никак не получится новую память выделить, то мы бросаем OutOfMemoryException. Иначе

мы спокойно отдаем этот адрес запрашивающей стороне, обновляя весь диапазон памяти, чтобы там не было лишних данных.

Но может быть так, что allocation context у нас уже кончается. Тогда будет промах мимо этого контекста. Как действовать в этом случае? Аллокатор может действовать двумя путями. Первый - это выделить, например, по-минимуму, второй - выделить сколько-то до некоего максимума. Этот размер может быть от 1 до 8Кб. И выбирается он исходя из интенсивности выделения памяти. Если на данном потоке выделяется много памяти, то нам нет никакого смысла выделять маленький кусок. То есть allocation context расширяется исходя из текущих запросов.

Но существует еще и многопоточность. Если она присутствует, может так получится, что несколько потоков одновременно начинают запрашивать выделения памяти. Значит, оператор new у нас должен быть потоко-безопасным, а значит и медленным. Сам по себе он быстрый, но если он выделяется без конкуренции с другими потоками, то получается, что он работает неоправданно медленно. Чтобы сделать new потоко-безопасным, надо чтобы каждый поток выделял память в каком-то своем месте. Поэтому в .NET существует по одному allocation context на каждый поток.

Отсюда можно сделать еще один вывод - не стоит делать много лишних потоков. Вы делаете очень жирный хип нулевого поколения благодаря этому.

Итак, много потоков у нас уже существует и каждый из них что-то выделяет. А еще у нас есть три поколения. Получается, что нулевое поколение, в котором идет выделение, начинает расти благодаря тому, что у нас растет количество потоков. Это плохо.

Обратимся к идее списка свободных участков.

Если мы говорим про классику, как везде heap устроен, то .NET - не исключение. Когда происходит sweep collection вместо сжатия кучи, у нас появляются свободные участки. Ими надо как-то управлять.

Есть два алгоритма менеджмента свободных участков. Первый - best-fit. Когда выделяется память, среди всех свободных участков выделяется тот, который имеет либо такой же размер, либо максимально приближенный по размеру к тому, который просили. Это best-fit.

Когда мы используем best-fit, нам нужно найти наилучший участок. Мы пробегаем по всем участкам, и запоминаем то, что мы прошли. Выполняется линейный поиск и наш участок может оказаться в конце. Но зато на выходе мы получаем минимальную фрагментацию - это здорово. Второй алгоритм - это first-fit. Это альтернатива. Мы идем вперед по списку и берем первый же участок, который нам подошел. Он может быть таким же по размеру, либо больше или существенно больше требуемого. Работает это быстро, но может сильно фрагментировать память. В .NET используется смесь этих подходов, организуется список свободных участков через корзины - бакеты. У каждого поколения есть список бакетов. Они группируют память по размеру от малого к большому. Бакет ссылается на односвязаный список свободных участков. Если посмотреть вниз, то от Head на свободный участок встали и дальше по односвязному списку можем идти дальше, перечисляя все свободные участки, которые относятся к этому бакету. Напомню, они организованы по размеру. И в данном бакете находятся участки с определенным диапазоном размеров. В следующем бакете будут свои диапазоны.

Бакет - это first-fit, среди бакетов выбираем первый, который подходит, уходим внутрь этой корзины и там, по однозвязному списку мы идем по best-fit. Таким образом мы делаем очень хорошую оптимизацию.

Что такое бакеты? Эта табличка (см. слайд 22:05) нам раскрывает глаза. Тут вспомним разницу между SOH и LOH: LOH организован только по принципу sweep collection, сжатие кучи там происходит только по запросу. Само по себе сжатие кучи там не запустится никогда. В SOH идет смесь.

Посмотрим на LOH и второе поколение, которое является самым большим. Исходя из этого можно легко понять содержимое таблицы. Нулевое и первое поколение имеют количество бакетов - 1. Потому что в нулевом и первом поколении у нас нет такого, что при sweep образуется куча свободных больших участков, куда можно разместить новые контексты. Нам этот функционал бакетов там не нужен, поэтому бакет там один.

А во втором поколении, которое может занимать гигабайты мелких объектов и в LOH, бакеты уже присутствуют, благодаря чему память там организована по рассказанной ранее структуре.

Как происходит выделение памяти в этом случае? Сначала мы сканируем бакеты, в поисках первого, который подходит по размеру участков, находящихся внутри него. Когда мы выбрали бакет, идем по односвязному списку и в нем ищем лучший из тех, которые там присутствуют - тот, который лучше всех подойдет нам по размеру. Но есть одна особенность. Может так получится, что мы просили не так много места, а имеющиеся участки слишком большие. В этом случае мы выделим сколько необходимо, а остаток вернем в список свободных участков.

Итак, мы нашли нужный участок. В этом месте лежит эмуляция объекта. Первое поле - это поле undo. Это - освободившийся участок, там лежал объект или группа объектов. Соответственно, первое поле это aSyncBlockIndex, потом - таблица виртуальных методов, потом данные. Когда этот участок стал участком свободной памяти, то первое поле стало операцией undo. Когда мы этот участок отлинкуем и сохраним ссылку на следующий участок в операции undo, чтобы можно было отменить, если что-то пойдет не так.

Дальше таблица виртуальных методов свободного участка, чтобы GC понимал, с чем он имеет дело и size - размер свободного участка.

После этого выделения мы проставляем aSyncBlockIndex. Таблица виртуальных методов становится та, что надо и вызывается конструктор. Все. Вот так работает выделение памяти.
\

Выделение памяти в SOH

Получается, что для того, чтобы выделить память сначала отрабатывает самый простой способ. Если объект влезает в allocation context, мы выделяем и отдаем. Это самый быстрый способ. Мы выходим моментально, у нас нет никаких дополнительных действий: вошли и вышли, и передвинуть указатель нам ничего не стоит.

Если не влезаем в контекст, то идет первое усложнение. Контекст необходимо увеличить. Чтобы увеличить, мы используем более продвинутый способ, он находится в JIT_NEW helper.

Мы пытаемся найти неиспользуемую часть в эфемерном сегменте. Когда у нас allocation context заканчивается, мы должны его куда-то перетащить. Это дорого. Поэтому мы сначала пытаемся его увеличить, нарастить. Если это не получилось, то надо перетащить.

Куда мы должны его переместить и какие данные у нас для этого есть?

Нужно поискать свободный участок в списке через бакеты, куда влезет наш контекст. Если участок есть, мы его передвигаем туда и пытаемся первым способом выделить там память.

Если получилось - замечательно. Если не влез, мы пытаемся подкоммитить больше зарезервированной памяти: сдвинуть вертикальную черту вправо. Место зарезервировано, но памяти там еще не существует. Нам нужно расширить кучу и мы коммитим память. Сразу же мы не делаем этого, потому что commitment - операция достаточно долгая, как и резервирование. Она может занимать сотню миллисекунд. Это происходит потому, что для ее осуществления, необходимо обратиться к Windows. А операционная система максимально не доверяет тому, что в ней находится. Он отгораживает для нас песочницу, и когда мы вызываем winapi метод, то мы не просто его вызываем, а с повышением уровня привилегий. Чтобы это безопасно это сделать, необходимо, чтобы Windows скопировал фрейм метода, который вызывается из уровня приложения в уровень ядра, чтобы случайно не подхватить чужие инфицированные данные. Потом winapi функция отрабатывает и происходит возврат. В этот момент идет обратное копирование кусочка стека и идет переход из kernel части в user space. Вся эта операция занимает много времени, поэтому GC максимально старается использовать текущее пространство. И на каждом этапе он делает GC нулевого поколения: если что-то подсжимать, возможно, allocation context обратно вырастет. Когда контекст кончился, GC пытается его раздвинуть вправо, а там уже что-то есть, значит надо поискать свободные участки. Если не получилось, делается сборка мусора, может быть свободные участки все же появятся и получится разместить среди них новые данные. Когда ничего не получилось и все забито, то приходится коммитить дальше память. А если память кончилась, то вылетает OutOfMemoryException.

Выделение памяти в LOH

В хипе больших объектов мы пытаемся найти неиспользуемую память, повторяя для каждого эфемерного сегмента LOH, потому что тут их может быть несколько, в зависимости

от того, в каком режиме работает платоформа .NET. Поскольку в LOH по умолчанию у нас нет сжатия кучи, то управление памятью тут упрощено. Сжатие кучи - это очень жирная операция, которая требует больших вычислительных процессов. Но в LOH эта операция проводится только по запросу, а значит программист исходя из знаний о работе алгоритмов собственной программы, решил, что в данное время он готов потратить неизвестно сколько времени на сжатие огромного пространства. А значит, можно сильно упростить алгоритмы по работе с LOH.

Для каждого эфемерного сегмента мы ищем участок памяти в списке свободных, пытаемся увеличить, если не получилось найти. Если не получилось увеличить - пытаемся подкоммитить зарезервированную часть. Следующим шагом запускаем GC, возможно несколько раз. И если совсем ничего не выходит - дергаем OutOfMemoryException. В данном подходе нет слова "сжатие", мы идем по простому пути.

Многопоточность. Есть такое понятие как баланс хипов. Например, GC запущен в серверном режиме. Это значит, что у него по несколько SOH и LOH и эти пары назначены к конкретному процессорному ядру. У каждого ядра есть своя пара хипов и свои треды.

Представим, что на каком-то ядре баланс нарушился. Оно начало аллоцировать слишком много памяти и перестало помещаться в текущие сегменты. Рассмотрим, что может сделать система управления памятью в данном случае.

Ей может показаться, что если данное ядро занимается очень плотной аллокацией, то можно взять и перекинуть контекст на соседнее ядро и аллоцировать там. Одно ядро стоит, а другое выжато по максимуму. Это поможет избежать таких дорогих операций, как коммитмент памяти. Но при изменении контекста, вместе с этим участком, в другое ядро переносится и поток.

Это делается не только потому, что закончилась память. Есть и другая причина. У каждого процессора есть кеш ядра. Когда приложение отрабатывает на каком-то участке памяти, весь он находится в этом кеше. Когда вы начинаете работать с другим участком и промахиваетесь мимо кеша, приходится выкачивать другой участок памяти под кеш. Если переключаться туда-сюда, будет самая худшая ситуация. Все будет работать очень-очень долго.

Когда текущий allocation context ядра заканчивается, а на соседнем ядре ничего в это время не происходит и проще туда перекинуть контекст, чем аллоцировать новый кусок памяти, то он вместе с этим контекстом вынужден туда перекинуть текущий поток. Теперь он будет работать на другом диапазоне памяти. Ему нужно, чтобы другой диапазон памяти находился в его кеше. Поэтому вместе с контекстом перемещается и текущий поток.

Какие выводы можно тут сделать. Во-первых, надо помнить как выделяется память. Во-вторых, чтобы оптимизировать работу GC, надо понимать, что вам понадобиться сколько-то объектов определенного типа. И после вызова new, заполнили память, алгоритм у вас отработал. А потом у вас создается еще один объект такого типа, например уже в цикле. И это плохо. Алгоритм отрабатывает некоторое время, в параллельных потоках в это время может быть произведена и другая аллокация. И в том же алгоритме вы можете выделять другую память. Соответственно, вы фрагментируете память по типу. Во время GC вам нужны объекты первого типа как объекты нулевого поколения, которые быстро исчезнут. Но в алгоритме вы аллоцируете вечные объекты. GC вынужден проводить фазу сжатия кучи, т.к. эти самые объекты, которые быстро ушли на покой, образовали маленькие участки свободной памяти, куда больше ничего не разместить. В противном случае GC мог бы обойтись обычным sweep: если бы участок был большой, он бы попал в список свободных участков.

Как лучше сделать. Если есть алгоритм на десять тысяч итераций, и там нужны объекты типа A, лучше выделить их сразу группой, а потом использовать. И когда GC отработает, у него получится огромный участок, где эти объекты были выделены. И он весь его отдаст в список свободных участков. Если же в вашем алгоритме вам каждый раз нужен всего лишь один экземпляр объекта типа A, то переиспользуйте его, сделайте метод init вместо конструктора. Получится, что вы вообще не фрагментируете память, а работаете на одном объекте.

Если вам нужно одновременно сто объектов делайте пул объектов, и доставайте их оттуда, чтобы не фрагментировать память короткоживущими объектами внутри алгоритма, чтобы не делать длительное сжатие кучи.

Первая книга вышла недавно. Ее написал Конрад Кокоса. Книга о менеджменте памяти объемом в тысячу страниц. Когда я ее в первый раз получил, я ожидал увидеть нечто

другое. От размера тома у меня был шок. На самом деле автор молодец. Он написал ее для любого уровня подготовки. Это значит, можно вырезать вообще все, что лишнее. Например, уровень управления памятью процессором, Windows, список инструментов, которыми удобно пользоваться для диагностики - если вы хотите почитать просто про менеджмент памяти, то можно взять вторую половину книги. Первая половина книги - это просто подготовка ко второй. А вторая половина - как раз менеджмент памяти. Взять этот менеджмент памяти. Там очень легким языком все написано, постоянно делаются отсылки к предыдущим главам.

Если книгу ужать, получится страниц 300. А это уже подъемный объем, то, что возможно прочитать.

Вторая книга - Under the Hood of .NET Memory Management - для меня уже классика. Она не переведена на русский и распространяется только через сайт Red Gate, который делает разные инструменты для рефакторинга и работы с базами данных. В том числе у них есть тулза для анализа памяти, они были вынуждены исследовать как эта память работает. Там в очень коротком стиле, достаточном для того, чтобы приятно удивить людей на собеседовании в компанию, это все описано. В районе 200 страниц занимает.

Введение в сборку мусора

[In Progress] адаптация курса

Следующий вопрос, который мы обсудим - это введение в сборку мусора.

GC работает в двух основных режимах: Workstation и Server. Связано это с особенностями поведения приложения: либо оно серверное, либо десктопное.

У десктопного приложения есть UI, со своими стандартами отклика. Стандарты отклика - около ста миллисекунд. Если после нажатия на кнопку и в течение ста миллисекунд ничего не происходит визуально, пользователь начинает думать, что программа тормозит. Исходя из этих оценок, приложение должно отрабатывать равномерно быстро без больших задержек в случайных местах. Поэтому GC работает чаще, чтобы иметь маленькую кучу нулевого поколения, на которой он может за известное время отработать.

Если GC вдруг внезапно запустился во время отрисовки кнопки, он гарантированно уйдет раньше, чем эти сто миллисекунд, не создав у пользователя ощущения, что что-то тормозит.Если речь идет о серверном режиме работы, то там несколько другие условия работы. На сервере обычно много памяти, а менеджер памяти имеет возможность развернуться достаточно широко. Как мы знаем, на каждое ядро создает по паре хипов, в процессе работы GC отрабатывает реже, а по возможности - не отрабатывает в принципе. И, тем самым, GC увеличивает скорость работы сервера в промежутках между сборкой мусора.

Прерывается он редко. Если говорить про ASP.NET (а там у нас запрос-ответ), то можно пока на одной ноде идет GC, на второй что-то делать. Каждый из этих режимов сейчас доступен в двух подрежимах: Concurrent и Non-Concurrent. Concurrent - это мифический режим. То есть когда у нас GC происходит, основной режим - когда все потоки встают и после завершения GC продолжают работу. Проще всего менять память, когда она не меняется кем-то другим. Мы может сжать кучу, поменять все указатели объектов на новые значения. Все это произойдет и при этом основное приложение не работает, а значит - нет рисков. Это Non-Concurrent. А Concurrent - это идеальный режим, когда все происходи в параллели. Но такого не бывает.

Есть статья на Хабре. Еще я смотрю видеоролики от Джуга, где Шепелев (или кто-то другой, увлеченный GC) про них рассказывает очень часто. В Java очень много видов GC. Алибаба недавно свой тулз сделал. Это там, где бывший нищим китаец, стал самым богатым человеком в мире. И у них есть свой GC. Они тоже тоже писали, что у них есть один единственный GC, который по-честному полностью Concurrent. И в ходе расскза получается, что все же полностью Concurrent не бывает.

Наш - не исключение. Я не копал и подробно рассказать не могу про Concurrent режим, но знаю, что паузы там есть. Просто они очень короткие. Такого, что он встал на паузу и до упора там нет.

Как происходит сборка мусора? Если срабатывает триггер нулевого поколения, будет собрано только оно. Это правило. Нулевое поколение состоит из новых объектов, соответственно диапазон сборки мусора короткий, алгоритмы отработают крайне быстро.

Если срабатывает триггер первого поколения, будет собрано нулевое и первое, потому что подразумевается, что первое поколение будет отрабатывать дольше. Если учесть, что нулевое отрабатывает чаще, то, скорее всего, после первого поколения, нулевое тоже будет в ближайшем будущем собрано, то можно его собрать сразу же. Поскольку первое поколение более тяжеловесное, если к нем присоединить сборку нулевого, это не сильно скажется на производительности.

Если мы дошли до второго поколения, которое может достигать феерических размеров, то тем более никакой погоды не сделает, если собрать первое и нулевое поколение вместе со вторым.

Каков порядок?

У нас есть такой график 08:43. То, что темно - это есть какие-то объекты. Светлые участки без объектов. Есть некий allocation context. Пусть, у нас будет один поток.

Что тут делает GC.

Первым делом он маркирует объекты для целевых поколений. Но здесь нет ссылки и пора мусор почистить. Дальше идет следующий этап: GC выбирает между техниками сборки мусора. Техник может быть две: sweep collection без сжатием, и техника с сжатием.

В sweep collection все недостижимые объекты нулевого поколения трактуются как свободное место. Все достижимые ообъекты становятся поколением один. То есть он у нас двигает границы. Мы промаркировали, освободили место, поколение передвинули. Compact collection все достижимые объекты поколения ноль уплотняются, занимая места недостижимых объектов. На слайде видно 10:38, что в том месте находится то, что было, а где есть свободные места. Уплотняем и сдвигаем границу поколений. Мы не копируем объекты из поколения в поколение - это дорого. Мы сдвигаем границу, указатель.

Маркировке были подвергнуты только объекты нулевого поколения, другие объекты мы не маркировали, это дорого. Поколение ноль стало пустым. Это поведение по умолчанию. После того, как граница поколений сдвинулась, достижимые объекты переместились в поколение номер один. У объектов нет признаков, в каком поколении они находятся. При смене поколения они никуда не копируются, просто меняется адрес диапазона. Когда вы проверяете GC.GetGeneration(), получаете номер поколение. Этот метод смотрит, в какой диапазон адресов попадает адрес объекта.

Поколение один выросло, как в случае sweep, так и в случае compact. Поколение два и LOH оказались не тронутыми.

GC выбирает между техниками сбора мусора. Но на самом деле может так случится, что может быть произведено выделение памяти в более старшем поколении. Если там оказался какой-то свободный участок. У нас есть три поколения, есть LOH. Срабатывает первое поколение, GC работает на первом и нулевом. Дальше GC решает, что можно переместить объект один из gen_1 в gen_2, дальше уплотнить кучу, то gen_0 получится очень большим. Если вдруг какой-то объект почему-то при сборке мусора нулевого поколения переместился во второе, то удивляться не стоит, такое может случится очень легко.Еще один вариант: закончился сегмент. Если обычной сборки мусора не достаточно для размещения allocation context, то текущий сегмент начинает резервировать все заново. Этот сегмент памяти, где размещался хип, помечается как gen_2 only. Это значит, что в текущем сегменте будет жить только второе поколение. А дальше он выделяет новый сегмент виртуальной памяти, резервирует новый кусок.

Когда приложение стартовало, он зарезервировал большой кусок памяти, но ее там физически нет. Начал коммитить. Пока он заполняет память, пытается сжать, коммитить

дальше, память растет. И вот он докоммитил до конца этого большого зарезервированного куска. Резерв закончился. Сейчас ему нужен новый зарезервированный кусок. Он аллоцирует новый. И когда их больше одного становится, этот новый кусок помечается как эфемерный. Там будет размещаться только нулевое и первое поколение. Для всех остальных будет поколение два. Когда будет три сегмента, то первые два сегмента будут использоваться под поколение два, а третий - эфемерный под нулевое и первое. Когда их будет 50, то первые 49 сегментов будут принадлежать второму поколению, а последний под нулевое и первое. 16:54

В кончившемся сегменте могли быть объекты нулевого поколения, а оно становится второго, то он должен скопировать объекты из нулевого поколения старого сегмента в первое поколение нового. Потому что все старые сегменты gen_2 only. А для нулевого поколения создаются allocation context для тредов.

Посмотрим на слайд 17:42, там видно, как все делается. Потом выделяется новый сегмент, все уплотнили. Дальше некуда, выделяется новый сегмент. В нем располагается gen_1 и gen_0, а в старом только gen_2. Если их было много, но при этом один из gen_2 сегментов старых вдруг кардинально опустел, и там освободилось много места, то GC может принять следующее решение: поскольку у нас есть кусок свободной памяти, а аллоцировать новый сегмент намного дороже, чем просто скопировать, то он просто берет и использует один из старых сегментов как эфемерный, то есть переиспользует его.

Чтобы это хорошо заработало, старые объекты gen_2 из этого сегмента должны быть убраны. Там не должно быть второго поколения.

У нас было три поколения gen_2 only, дальше он решил, что третий gen_2 only опустел и можно там разместить эфемерный сегмент вместо того, чтобы аллоцировать там новый кусок памяти от Windows. Он берет и перегоняет gen_0 туда, при этом нулевое поколение становится gen_1. Gen_2 перемещается из третьего сегмента в четвертый, который становиться gen_2 only. И остаток третьего сегмента становится gen_0. Идет такая пересортировка, так будет работать лучше. В старом эфемерном сегменте заканчивается место, а в третьем место есть, и они меняются ролями.

Как происходит сборка мусора?

Во-первых, что-то запускает сборку мусора. Дальше все управляемые потоки встают на паузу, если у нас Non-Concurren GC. Поток, который вызывал GC запускает процедуру сборки мусора, то есть GC работает в потоке, который его инициировал. Дальше выбирается поколение, которое будет очищаться. Происходит фаза маркировки для выбранных поколений. То есть мы пробегаемся по всем объектам и маркируем их. Дополнительно идем в карточный стол, ищем там ненулевые значения, трактуем их как корни. Маркируем все, что исходит от них. Дальше фаза планирования. На основе данных из фазы маркировки пытаемся понять, какой из двух алгоритмов сборки мусора будем применять - sweep или compact. Sweep из-за скорости в приоритете, но если статистика показывает, что лучше compact, то будет выбран он. В дальнейшем мы поймем, что на самом деле фаза планирования является основной. Для того, чтобы понять, какой алгоритм применить, надо фактически выполнить оба алгоритма одновременно. Последняя операция - сбор мусора - фактически это коммитмент тех вычислений, которые были сделаны на предыдущей фазе. Далее идет последний шаг - восстановить работу всех потоков.Визуально это можно увидеть на слайде 23:17.

Что вызывает GC?

Причин может быть много. Попытка аллоцировать в SOH, а там место закончился. inducted - когда мы руками запросили.

lowmemory - закончилась свободная память внутри процесса.

empty - затрудняюсь сказать что.

alloc_loh - соответственно, закончилась память в LOH.

oos_loh - это по короткому пути.

И так далее, на них не имеет смысла долго останавливаться. Что может быть. Исчерпали место в SOH. Это стадия, когда у нас allocation context закончился и GC надо его расширить. Когда контекст надо передвинуть в другое место. Прежде чем это делать, мы пытаемся скомпактить. Прежде чем мы расширяем сегмент. Потому что сегмент расширять дорого, мы попытаемся собрать мусор. Не получилось - пришлось расширять сегмент. И, наконец, прежде чем мы пытаемя новый сегмент использовать под эфемерный. Потому что новый

сегмент - это еще дороже. И нужно попытаться в рамках текущего найти какие-то свободные участки и разместить новые объекты там.

Тоже самое про LOH.

Аллокатор исчерпал место после медленного алгоритма выделения памяти, после реорганизации сегментов и после GC. Ручной вызов GC тоже бывает триггером. Их бывает три режима. Первый GC.Collect() - вызов полного GC, блокирующего при этом при вынужденного compacting на LOH. Вызов GC.Collect(int gen) - на нужном поколении. Это тоже самое, но с выбором поколения. И последний режим - GC.Collect(int gen, GCCollectionMode mode) - это вызов на необходимом боколении, блокирующий, без вынужденного compacting, с необходимым режимом. Про ручной вызов GC хочется сказать, что на самом деле если вы доходите до этой стадии, это значит, что скорее всего вы не очень понимаете, что происходит в приложении. И как hotfix, чтобы все стало хорошо, вызывается GC.Collect().

То, что я слышу чаще всего, про вызов GC.Collect(), это приложения на технологии Xamarin. Там есть свои особенности. На Android там есть два GC: от .NET и от Java. Проблема в том, что все приходит от Java, имеет реализацию интерфейса disposable. Вкупе с непониманием, кто должен дергать dispose, там очень сильно текут ресурсы. Если не привыкнуть писать правильно. Это время от времени приводит к мысли вызвать GC.Collect().Если хочется вызвать GC.Collect() в обычном приложении, то, скорее всего, это значит, что нам не очень понятно как это все работает внутри. Почему алгоритмы приводят в необходимости вызова этого метода.

Чтобы не вызывать, необходимо посмотреть метрики от GC, построить графики. И посмотреть наложение графиков расхода памяти по разным поколениям, на график срабатывания GC, на другие графики и понять, что приводит к проседанию памяти. Возможно, вы увидите, что там что-то течет в этих графиках. Если такие места есть, то нужно смотреть внимательно в эти места и изучать там. В общем случае GC.Collect() вызывать не надо. Это, во-первых, означает, что GC потеряет свои статистики. После ручного вызова метода, GC может в том потоке, где вы только что выделяли объекты выдать вам allocation context не на 8Кб, а маленький, который замедлит дальнейшее выделение памяти. Это то, что лежит на поверхности.

На данном этапе мы рассмотрели в общих чертах все шаги в каком порядке это работает. Далее будем рассматривать конкретные шаги, подробности. Но эти подробности хороши, когда видишь картину целиком. Текущий доклад был про общую картину, как GC отрабатывает, какие режимы есть. Последующие доклады - это будет изучение вглубь. Сейчас будет маркировка, планирование, сборка мусора, то, что касается GC. После чего последним докладом по GC будет общий обзорный доклад с выводами. На основе того, что мы услышали, какие правила можно вывести, как наши алгоритмы должны быть построены, чтобы не наталкиваться на плохую работу GC.

Фаза маркировки достижимых объектов

[In Progress] адаптация курса

Мы поговорили про выделение памяти, про общую картину, теперь будем обсуждать все подробно. Фаза маркировки GC.

На этой стадии GC понимает какие поколения будут собраны. У нас есть знание, что GC является трассирующим. Как и весь алгоритм трассировки в 3D, чтобы простроить сцену, мы луч опускаем, с чем пересекся - то и объект, а куда отразился - отражение. Тоже самое и в GC. Мы к объекту идем по исходящим ссылкам и с помощью трассировки понимаем, какие объекты являются достижимыми, а какие - мусором.

Он стартует из различных корней и относительно них обходит весь граф объекта. Все, до чего он дошел - хорошо. Остальное - мертвые зоны. Если рассматривать простой сценарий, когда GC у нас не Non-Concurrent, то у нас встают сначала managed потоки, когда они встали можно делать с памятью все, что угодно. Например, сделать фазу маркировки. На этой фазе для любого адреса из группы корней в заголовке объекта устанавливается флаг pin, если объект у нас pinned. Pinning может происходить либо из таблицы хендлов у application domain, либо из ключевого слова fixed. И когда GC посреди fixed срабатывает, он понимает, что эту переменную надо запинить.

Обладая информацией о том, что у объекта есть исходящие ссылки на managed поля (это известно из таблицы виртуальных методов), он обходит все исходящие ссылки. Обход графов осуществляется путем обхода в глубину. Он сохраняет свое состояние во внутреннем стояке. При этом, при посещении каждого объекта, мы смотрим если он уже посещен, то пропускаем, если не стоит - устанавливаем флаг, как посещенному на указателе VMT. И, поскольку у нас все адреса в таблице виртуальных методов выровнены

по процессорному слову (они делятся в 32х разрядной системе на 4 без остатка, а в 64х разрядной - на 8, это сделано для того, чтобы процессор быстрее работал на этих адресах), то получается, что младшие два бита адреса не используются и равны нулю. Значит, туда можно что-то записать, ничего при этом не испортив. Главное, потом не забыть маркировку снять.

Все исходящие указатели из объекта с полей добавляются в стек адресов на обход.

В стек сначала добавляем все корни, потом начинаем цикл обхода. Забираем с вершины стека адрес, идем в таблицу виртуальных методов, там обходим информацию о исходящих ссылках на управляемые объекты. Каждую исходящую ссылку заносим обратно в стек, а объект маркируем как пройденный. Следующая итерация. Когда стек пустеет - обход завершился. Все установленные флаги стираются во время фазы планирования.

Корни.

Во-первых, корнями являются локальные переменные метода. В данном примере 06:36 есть локальная переменная path и она является, по сути, корнем. Мы можем в локальную переменную сохранить адрес объекта, но никуда больше. Чтобы доказать, что объект достижим, нам необходимо обойти весь стек потока и собрать там исходящие локальные переменные.

Это не должно быть собрано GC и может быть долго.

Локальные переменные могут храниться в двух местах.

Во-первых, в стеке потока - это структура, на основе которой происходит вызов методов. Есть поток, там крутятся методы, они друг друга вызывают. Их локальные переменные не пересекаются с локальными переменными таких же методов, которые в это же время вызываются, но в другом потоке. У каждого потока есть свой стек. Это массив, где при вызове метода выделяется некий кадр - кусок, в котором есть место под локальные переменные и некоторые параметры, с которыми метод вызывается. Когда вызывается следующий метод, он добавляется в конец. А когда метод завершает работу, они начинают с этого стека уходить. На этом стеке как раз хранятся локальные переменные метода и некоторые параметры, начиная с третьего, на сколько я помню, а первые два передаются через регистр.

Поскольку некоторые методы могут долго работать, то GC должен понимать, что эти места до определенных моментов собирать не надо. Дальше нужно смотреть по scope переменных, код на слайде 09:45. Первый - это переменная class1. Она доступна в течение жизни всего метода. Она сначала аллоцировалась и с точки зрения языка C# она живет от первой фигурной скобки до последней. Если говорить о переменной class2 с точки зрения языка C# она живет внутри блока if, от первой фигурной скобки до последней. Но есть два варианта окончания scope. Упрощенный, когда scope заканчивается с неким лексическим блоком, а есть вариант, когда scope заканчивается с последним местом его использования.

Раньше я считал магией, когда говорили, что GC может собрать объект прямо посреди метода, если вы перестали его использовать и других ссылок на него нет. Как GC понимает, что код перестал использовать переменную. Оказывается, все просто. Сбоку от описания метода лежит еще дополнительное описание scope переменных. Он примерно выглядит как на слайде 11:11. Если смотреть построчно, в первой и второе строках нет переменных. В третьей появилась переменная class1 в scope. Потом появился class2, и вот они постепенно начали исчезать. Это для частично прерываемого scope. Для полностью прерываемого scope у нас ситуация другая - слайд 11:37 . И на каком-то этапе они обе перестали использоваться. Седьмая строка - последняя, где используются обе переменные. И дальше идет операция return и после этого можно и не использовать. На самом деле здесь произойдет GC, то оба инстанса будут собраны. Но он сохраняет не просто значение переменной, но и место, где оно хранится, потому что архитектура Intel не подразумевает, что вы работаете напрямую с памятью. Надо сначала перекладывать адрес в регистр, а потом на основе регистра уже производить какие-то действия. В таблице scope хранятся именно регистры, хранящие данные. И когда доходит до варианта None, значит можно все собирать с конкретно этих регистров GC. Для Fully Interruptible на слайде 13:06 не используется class1. У нас тут картина меняется. Получается, что в третьей и четвертой строчке у нас используется регистр rax под хранение class1, потом в пятой строчке его использование пропадает. И если здесь сработает GC, то инстанс class1 может быть свободно собран.

Дальше джиттер понимает, что rax больше не используется и можно его переиспользовать еще раз уже под другую переменную. Делает это ровно в трех строках, после чего точно также дальше не используется и собирается.

Эта таблица использования 13:55 называется Eager root collection. Помимо того, что локальные переменные определяются стеком потока, они дополнительно определяются этой таблицей, которая, на самом деле добавляет нам много проблем.

Какие именно проблемы? Вот такой код 14:21, очень простой. У нас есть таймер, который срабатывает каждые сто миллисекунд, выводя на экран текущие дату и время. Он запускается, работает и дальше у нас печатается "Hello", GC.Collect() и ReadKey(). Если смотреть с точки зрения C#, то таймер используется во всем Main. Поведение в данном случае должно быть такое: напечатали "Hello", дальше, пока пользователь не нажал кнопку, мы начинаем тикать на экран текущее время. Во втором варианте, когда у нас таймер используется только в одной строчке, вызывается GC.Collect(), GC должен понять, что после первой строчки таймер уже не нужен. И мы максимум одну строчку успеем увидеть, прежде, чем GC этот таймер соберет. А может быть и вообще не увидим. Суть в том, что поведение будет разным. Это было поведение debug режима и релиза.

То есть, когда мы в релизе, думаем что все отладили и все прекрасно, запускаем на сервер, а там такое поведение и сложно понять почему оно. Получается, что поведение на релизе и на debug режиме отличается. Об этом надо помнить. И это легко проверить.

Еще один вариант такого поведения на слайде 17:12. У нас Main. Он создает экземпляр класса SomeClass, вызывает DoSomething и ждет пользователя. Дальше SomeClass, у него есть DoSomething, посреди которого срабатывает GC, делает WriteLine. И у него есть финализатор, в котором вызывается строчка финалайзера. Данный метод может быть заинлайнен. Метод Main короткий, ничего не делает и его можно оптимизировать и передвинуть наверх. При этом исчезнет scope переменной message. Scope переменной sc продлится либо до второй строчки, где DoSomething, либо до конца метода Main. Разница в том, что если DoSomething будет заинлайнен, то у него пропадет ссылка на текущий объект this. Значит, пропадет и scope его использования. А значит GC, который вызвался посреди метода DoSomething может вызвать финализатор этого класса до того, как метод этого класса закончит работу. Такое тоже возможно.

Первое - у нас есть сам класс. У него метод DoSomething, у него есть GC.Collect, есть Console.WriteLine и отсутсвует использование указателя this, потому что все это статическое, есть финализатор. И если посреди работы метода DoSomething, или даже с

использованием this, но GC сработал после последней точки его применения, тоже может быть ситуация, что финализатор вызовется до того, как этот метод завершит работу. Указатель на объект уже никому не нужен, а значит GC может совершенно спокойно экземпляр этого класса собрать. Выглядит страшно, но призываю не удивляться, если такое случиться.

Scope переменных.

Как расширить scope переменных, чтобы таймер был в обоих случаях. Простейший способ расширить - это вызвать GC.KeepAlive(). На самом деле это пустой метод, смысл в том, чтобы продлить использование переменной, ничего не делая. Ссылка на объект куда-то уходит и джиттер автоматически расширяет scope переменной до конца метода. Переменная ждет.

Еще одни корни - это pinned locals. Ключевое слово fixed. Есть два варианта пиннинга. Пиннинг - вещь очень плохая. Мы об этом поговорим на фазе планирования. Если есть возможность обходить пиннинг, лучше это делать. Если такой возможности нет, то нужно воспользоваться ключевым словом fixed, которое реального пиннинга делать не будет. Если дизассемблировать этот код в msi, то мы заметим, что эти переменные, которые обозначены 22:24 byte* array = list, вот этот list пометится как pinned. То есть мы пинуем list, его адрес помещаем в array. Но list пинуется, но только тогда, когда внутри этого fixed 22:49 срабатывает GC, если он не срабатывает между этими двумя фигурными скобками, то реального пиннинга не будет. Будет только флаг для GC, что этот массив нужно запинить, если он вдруг начнет эту область проходить. Это отличная оптимизация. Получается, что в Eager Roots collection дополнительно ставится тоже флаг, то, что переменная является pinned. Но не только у него, а у регистров тоже и у всего, где по реальному коду память будет содержать исходящие ссылки, дополнительно этот флаг так ж ставиться. Если GC срабатывает внутри фигурных скобок, то он, обходя Eager Roots collection, понимает, какие области памяти необходимо запинить и делает это, на время своей работы. Как только GC отработал и отпустил потоки, он распинивает эти объекты.

Еще одна группа корней - это Finalization Roots. Если мы сделали финализатор, и финализируемый объект ушел в очередь на финализацию, то он является естественным образом рутом для обхода объектов. Иначе получится так, что если эти объекты еще на

кого-то ссылаются, то они будут собраны GC. Чтобы этого не допустить, финализацию мы тоже обходим.

GC Internal Roots. Для обхода графа объектов используется карточный стол. Таблица маркировки старшего поколения, что есть ссылка на младшее. То есть корнями так же является карточный стол. Это причина, по которой лучше группировать места ссылок на более младшее поколение вместе. Поскольку весь карточный стол является корнем, то фаза маркировки будет проходить дольше, если на карточном столе будет много ненулевых значений. Когда мы его просматриваем в поисках ненулевых значений. Если нашли такое, то объект текущего поколения ссылается на более младшее. Мы трактуем ссылку как корень и тоже обходим.

GC Handle Roots

Последняя группа корней - это внутренние корни, в том числе таблицы handle. Внутреннике корни - это статики. Массивы статических полей всех классов - это ссылка изнутри app domain. Тут же есть ссылка на таблицу handle. Они делятся на бакеты. Бакеты группируются по принципу типа GC handdle, их может быть много. Есть GC handle с weak reference, есть com-вские, есть запинованные - всех не перечислить. Соответственно, все они могут ссылаться уже куда угодно - SOH, LOH. И эта таблица также является и таблицей корней.

Если совсем далеко не уходить, то это все. Какие можно сделать выводы?

Во-первых, не стоит делать много разных GC handle-ов. При их появлении создается дополнительная нагрузка на GC: структура handle-ов является боковой для графа объектов. Чтобы правильно все сделать GC вынужден лезть в таблицу handle, пробегать ее, перемаркировывать объекты, чтобы потом, на фазе планирования, правильно с этими объектами поступить.

Есть проблема и с карточным столом. Статики в целом - их не надо много делать. Это места, которые вечно держат все. А насчет пиннинга мы поговорим далее, так как при стадии планирования пиннинг играет важную роль. Нет таких сценариев, когда мы просто так используем пиннинг по принуждению, но то, что использовать надо с ключевым словом fixed мы точно запомним. Ну и начнем лучше проходить собеседования.

Фаза планирования

Следующая фаза, о которой хочется поговорить - это фаза планирования. Самая интересная фаза из всех. В ее ходе происходит виртуальный GC, без физического. А физический это просто коммитмент результатов вычислений на фазе планирования.

После фазы маркировки у нас нет никакой ясности, какой тип GC должен сработать: sweep или collection (?compacting). Чтобы понять, будет ли результирующая фрагментация слишком высокой, надо, во-первых, сделать и sweep, и сжатие кучи, но в виртуальном режиме. Иначе не понять, какой процесс будет лучше. Если мы можем примерно прикинуть, то механизмы пиннинг а могут помешать этим прикидкам и испортить конечный результат GC. Во время фазы планирования собирается информация, которая одновременно подходит для обоих алгоритмов, чтобы все сделать максимально быстро и выбрать лучший путь. После того, как GC понимает, что compacting, например, - это лучшее, что можно сделать, он его и делает. Но алгоритма, на самом деле, два. Я не слышал, чтобы были разговоры, чтобы в SOH работали оба алгоритма. До сих пор в моем представлении все выглядело так: в SOH идет сборка мусора путем сжатия куча, а в LOH - классический C++ алгоритм со свободными участками.

В реальности в SOH работают оба алгоритма, как и в LOH.

SOH.

Используя информацию о размерах объектов мы идем друг за другом и собираем группы свободных объектов - это plugs, и группы пропусков - gaps. То есть, когда мы пробегаем по хипу, то можем легко итерироваться. В самом начале хипа лежит первый объект. Мы переходим в таблицу виртуальных методов, проходим по указателю и по нему, среди прочих полей, лежит размер reference type. Я не очень понимаю, почему sizeof() работает только для value type, потому что чтобы сделать sizeof() reference type достаточно просто пройти по указателю и считать этот размер. Есть только одна разница - для массивов и для строк. Потому что для строк массивов надо этот sizeof посчитать, просто так не угадаешь. А для всех остальных обычных типов проблем никаких нет.

И вот так, друг за другом, мы по всем объектам пробегаемся. Понимаем, какие из них промаркированы, поэтому легко можем понять, что является plug, занятым куском, что является gap, пропуском. Это вычисляется элементарно, так как на фазе маркировки мы там

галочку поставили. Все друг за другом идущие пропуски мы маркируем как едины пропуск - группу. Все друг за другом идущие занятые участки мы маркируем как один занятый участок.

Размер и положение каждого пропуска могут быть сохранены. Дл каждого заполненного блока может быть сохранено его положение и финальное смещение. Это значит, что для каждого gap мы считаем его размер: напрмиер первый 32 байта. Для plug считаем offset, это то значение, которое было бы при запуске сжатия. Если бы мы сжимали, то offset - это то место, куда текущий plug переместился бы, насколько байт назад. Gap - это gap, тут понятно. И уже на этой стадии, если мы говорим о двух алгоритмах - sweep и collection (compacting?), можно увидеть всю информацию, которая нам нужна. В случае sweep нам нужны размеры пропусков, так как мы будем формировать списки свободных участков. Если мы делаем сжатие кучи, нам нужны только офсеты. На сладйде 07:10 это видно: вот он офсет -32, потом 32+64-96, следующий офсет -96+64-160. Так легко это все считается, поэтому все работает быстро. Чтобы это все произошло используется некий виртуальный внутренний аллокатор. Он итерирует по объектам и наращивает эту виртуальную дельту, сразу же делая виртуальный compaction этой кучи.

Полученную информацию GC хранит в последних байтах предшествующего заполненной части gap.

Был какой-то объект, на него исчезла последняя ссылка, прошел GC, информация в gap нам больше не нужна, значит мы его можем спокойно портить - писать туда то, что хотим, в том числе и результаты вычислений. Если у нас есть некий plug, то информацию мы запишем в предыдущий gap.

На слайде 08:33 есть некие plugs и gaps и некий диапазон по центру. Мы его увеличили. Получилось, что там три процессорных слова. Дальше несколько слов - это plug. И последние три слова свободные. И перед plug, перед занятым участком, этот gap в три байта мы опять увеличили. Первый байт - это gapsize, второй - relocation offset. Третий разделен на две части, первую мы не рассматриваем, а вторая - это left и right offset, которые мы рассмотрим чуть позже.

Размеры, которые мы считали, кладутся в gapsize. A relocation offset укладываются вторым байтом. Вся эта информация, которая размещена над куском памяти, вписана на места в нашем графике.

Что же такое left и right? С помощью этих двух полей образуется двоичное дерево поиска. У нас есть plugs и gaps. Если рассматривать их как единую сущность, как на слайде 10:30, то left/right offset - это offset до левой и правой ноды в двоичном дереве поиска нужных plugs.

Двоичное дерево поиска строится для того, чтобы после фазы планирования, в ходе фазы сжатия нам все адреса поменять: куча сжимается, адреса объектов меняются. У всех объектов, которые ссылались на наши, надо заменить адреса в исходящих полях, локальных переменных, во всех рутах - везде, где были ссылки на группу, которая сжимается. Чтобы это сделать быстро и строится бинарное дерево поиска сбалансированное.

Куча может быть огромная, поэтому используется структура, называемая Brick Table. Чтобы не искать по всему дереву, используется структура, схожая с карточным столом. Весь участок памяти приложения делится на большие регионы. Ячейка Brick Table - это ссылка на корень дерева двоичного поиска, которая описывает plugs и gaps внутри этого диапазона памяти.

Если у нас на слайде 13:00 диапазон памяти от 1000 до 2000, вторая ячейка Brick Table отвечает именно за него. Она указывает на некий plug в центре, который является корнем двоичного дерева поиска других plugs.

Если значение ноль, то в этом диапазоне нет никаких данных. Если значение положительное - то это адрес некоего корня. Если значение отрицательное, то этот диапазон является продолжением предыдущего диапазона. В процессе поиска нужного адреса мы делим на 0x1000, чтобы получить номер ячейки. Встаем на ячейку, дальше смотрим, что там за значение. И если ноль - то нет данных, если больше единицы - корень, меньше нуля - то там офсет на ту ячейку Brick Table, в которой записан корень.

Pinned Objects.

Чем плохи запиненые объекты?

Посомтрим на слайд 14:34. Тут есть кусок памяти, на нем - plugs, бледные участки, а яркий кусок - это запиненый объект. Что произойдет во время GC? Plug сожмется и все. А запиненый кусок останется на месте. Что произойдет, если слева от запиненого куска есть участок памяти, на который есть ссылка?

Давайте вспомним, что же делает фаза планирования? Она вычисляет plugs и gaps. И в зоне, которая предшествует plug в gap планирование записывает размер gap и смещение. Вот здесь возникает неприятная ситуация. У нас два занятых участка. Первый - обычный plug, а второй - запиненый. Мы не можем рассматривать их как единый кусок. Потому что первый будет двигаться, а второй нет. Поэтому выделяют plugs и pinned plugs.

В данном случае можно выкрутиться. Поскольку у нас все потоки стоят, мы можем испортить данные, а потом вернуть все назад. Память перед запиненым участком интерпретируется как кусок, в который можно что-то записать. Но перед тем, как этот участок портить, данные нужно куда-то сохранить, чтобы вернуть все обратно.

Этот участок в 4 байта сохраняется в pinned plug queue. Это очередь на возврат. Мы сохраняем участок, чтобы потом его вернуть обратно. Участок называется saved_pre_plug. Туда сохраняется информация о типе информации и к чему она относится. После этого все сжимается и этот участок возвращается на свое место.То есть появляется дополнительное действие.

А что, если наоборот? Сначала идет запиненый объект, а потом обычный объект, на который есть ссылка. С запиненым объектом все хорошо, он может свои данные хранить в gap. Где же хранить данные обычному объекту? Перед ним запиненный.

Почему пинят? Потому что буфер уходит в unmanaged память. Если GC, перед тем как начать работу, саспендит потоки - managed. То про обычные потоки он ничего не знает. Он не может их засапендить. А если буфер ушел в unmanaged память, то любой может с ним в параллели работать из unmanaged кода. Туда нельзя ничего записывать, в том числе и информаций plugs&gaps. В этом случае весь этот кусок помечается как pinned plug: и запиненый объект, и тот, который за ним следует. Они оба пинуются. После GC они в такой же последовательности и остаются.

Еще одна ситуация. Сначала у нас идет запиненый, потом два незапиненых объекта. Перед третьим объектом есть второй обычный объект. То есть, когда pinned plug объединяется для GC, в дальнейшем все равно воспринимается раздельно. Последний plug знает, что предшествующий объект не запинен и туда можно писать информацию о gaps и plugs. Но такой кусок тоже уходит в очередь pinned plug queue как saved_post_plug объект.

И совсем нездоровая ситуация возникает, когда у нас идет несколько объектов подряд и один из них запиненый. Это одна из частых ситуаций. В этом случае происходит совмещение обоих объектов. Для того, чтобы записать gaps и offsets, нам необходимо испортить и предыдущий объект, и следующий. А чтобы восстановить, мы должны эти куски уложить в очередь на восстановление.

Demotion.

Есть такие понятия, как Promotion и Demotion.

Promotion: GC дернули на нулевом поколении и после этого номер поколения вырос. Demotion работает наоборот. Когда у нас было первое поколение, мы сделали GC, и по какой-то причине оно стало нулевым.

После GC запиненый объект остался в поколении ноль - это суть Demotion.

Другая ситуация: есть поколение один и поколение ноль. На слайде 21:52 после GC слева осталось мало места, а под gen_0 - походит. И они оба перескочили на gen_0. Этому удивляться не стоит. Если такое произошло, надо просто искать, кто запинил объект и освобождать его.

Еще одна ситуация на слайде 22:56. Объект может свалится во второе поколение, если слева места мало и там нет смысла размещать что-то еще. С запиненными объектами может происходить все, что угодно.

LOH.

Фаза планирования в LOH имеет смысл только для compacting, чтобы понять дальнейшие действия. В обычных сценариях сборки мусора у нас только sweep, а compacting нет. И фаза планирования в таких случаях отсутствует. Sweep не использует plug&gap, планирование

ему не нужно. Он просто идет и составляет списки свободных участков. В этом плане sweep гораздо приятнее compacting.

В обычном режиме LOH не осуществляет compacting, только по принуждению. Но в этом случае тоже без планирования. Потому что вы попросили конкретную и у него вся информация будет построена на месте, со всеми офсетами. Будут построены офсеты, plugs&gaps только для того, чтобы понять, может ли он обойтись обычным sweep. И если это возможно, то он его и сделает в хипе маленьких объектов. А в куче больших вычислять ничего не надо: что его попросят, то он и сделает. Поэтому фазы планирования там нет.

Так как LOH сделан исключительно для хранения больших данных, это позволяет упростить некоторые вопросы. Нет необходимости группировать объекты в plugs. В SOH это происходит исключительно ради производительности. Если у нас куча объектов по 32 байта, то нет смысла их рассматривать отдельно. С точки зрения GC их надо рассматривать как единое целое. Поэтому каждый объект в LOH - это plug.

Плотность объектов в нем намного ниже, значит эффективнее трансляция адресов - эффективнее работает дерево. Легче перенести большой объект, чем plug, группу больших объектов. Чтобы обеспечить хранение plugs&gaps информации между объектами в LOH резервируется дополнительное место. Мы не можем себе позволить резервировать дополнительное место под информацию по размерам gap и relocation offset в куче маленьких объектов. Это будет слишком жирно. Поскольку у нас адреса всех объектов должны быть выровнены по размеру процессорного слова, по 8 или по 4 байтам, а информация по plugs и relocation offset занимает всего 3 байта, то мы не можем этими тремя байтами манипулировать. Мы либо 4 будем использовать, либо 8. Поэтому в LOH используем оптимизацию, храним эту информацию в gap. А в LOH мы можем себе позволить дополнительную информацию слева положить, не перетирая при пининге предыдущие объекты.

Когда строится compacting в LOH, мы просто дополнительно резервируем между всеми объектами немного места для того, чтобы положить туда офсет. Когда мы запрашиваем compacting вручную, мы просчитываем эти офсеты и все сжимается.

Почему в SOH вместо sweep может быть выбран compacting.

Например, это последний GC перед OutOfMEmoryException. Последняя надежда, что получится найти немного памяти.

Compacting может быть попрошен программистом намерено. Или было выбрано все место в эфемерном сегменте. Если у нас в эфемерном сегменте нулевого и первого поколения выбрано вес место и все закоммичено, то необходимо аллоцировать новый сегмент. Это дорого по времени. Поэтому первое, что пытается GC сделать, это сжать.

Высокая степень фрагментации поколения так же может спровоцировать compacing. Это один из основных случаев. Если фрагментация слишком высокая и нет никакой возможности разместить allocation context в каких-то свободных промежутках памяти, то запускается сжатие.

Еще один вариант - процесс занял слишком много памяти. Это, в целом, понятно.

Рассмотрим фрагментацию.

Это понятие достаточно виртуальное. Что значит "слишком фрагментировано"? Ответ на вопрос простой. Если взять некий total fragmentation, который собирается во время фазы планирования (это суммарное количество памяти, которое занимают gaps) и поделить на размер поколения, то мы получим отношение. При fragmentation size в 40 тысяч байт на нулевом поколении fragmentation ratio это 50%. Это триггер. Если фрагментация вырастает, то мы делаем сжатие кучи.

Мы получили новую информацию, что если у нас есть фрагментация более 50% при определенном размере хипа, то запускается не очень приятная процедура сжатия, которого мы хотим избегать.

Избегать процесса сжатия относительно просто. У нас часто возникает ситуация, когда мы создаем объекты одного типа, при этом создаем мы их не сразу друг за другом, а на всем протяжении жизни программы, зная о том, что эти объекты будут удалены вместе. Это приведет к фрагментации. Чтобы так не случилось, можно, например, сделать пул этих объектов. В нем объекты создаются друг за другом, группой. И когда будет потеряна ссылка на пул, они вместе уничтожаться, тем самым создав единый gap. Я в целом приходил к выводам, что пулы использовать очень хорошо. Они могут помочь во многих ситуациях. Спасают от фрагментации, от попадания на карточный стол (вы достаете объект из пула уже второго поколения). Пул - метод инициализации. Туда же можно отнести боксинг, который можно решить через эмуляцию боксинга или даже пул эмуляции боксинга. Если есть какой-

то метод, который принимает object, вы изменить этого не можете и вынуждены боксить, то можно этого избежать. Фаза планирования на этом завершена.

Sweep & Collect

Мы рассмотрели практически все стадии, кроме последней. Эта фаза зачистки и сжатия.

Если на фазе планирования было решено использовать sweep, мы должны осуществить GC путем отдачи всех неиспользуемых участков на переиспользование. Когда наши allocation context, которые существуют от разных потоков, начнут заканчиваться, они будут занимать эти свободные участки между другими объектами, таким образом исключая фазу сжатия. Фаза сжатия работает дольше. После этого этапа мы имеем такую таблицу 01:27. У нас есть поколения с бакетами, которые указывают на односвязаный список свободных участков. Бакеты сгруппированы по размерам. Внутри одного бакета идет список участков примерно одного размера. Мы можем попросить у бакетов участок нужного размера, и он по принципу first fit найдет нужный бакет и через best fit найдет подходящий участок внутри бакета.

Gaps, которые меньше размера объекта игнорируются.

После этого saved_pre_plug и saved_post_plugs расставляются по местам. Потом выполняется работа по обновлению очереди на финализацию. И затем - перестроение сегментов памяти.

Например, может так случится, что после GC у нас часть страниц или сегментов больше не нужны. Зачем их держать, если можно отдать кому-то еще? Операционная система большая. Если не отдать, то все соседние приложения будут ругаться, что работают медленно. Так и есть: потому что наше приложение не освободило оперативную память под соседей.

Sweep на LOH работает по-другому. Он не использует планирование, он обходит кучу. Свободные участки объединяет с состыкованными в один. Освободившиеся участки между занятыми добавляются в список свободных участков.

Для меня было откровением, что в обоих хипах механизмы одни и те же. Просто в одном хипе у них разные приоритеты. В LOH compacting в настолько низком приоритете, что часть алгоритма просто отключена. Но в целом SOH и LOH использует одни и те же алгоритмы.

Compacting.

Если необходимо создается новый эфемерный сегмент. Если фаза планирования решила, что после сжатия места под gen_0 будет мало - создаем новый сегмент. Заменяем все ссылки на корректные. Для этого у нас есть информация по gaps&plugs, по которым легко посчитать все смещения. Перед тем, как сжимать, мы должны все ссылки поменять. Собирая gen_0 мы просматриваем его и карточный стол старшего поколения. Когда это сделано - начинаем менять ссылки в фазе сжатия до самого сжатия, чтобы не потерять информация по gaps и офсетам. Сканируем все места, производя замену. Сначала ссылки со стека, которые включают в себя все локальные переменные и параметры методов, переданные через стек, а так же eager roots collection. Потом ссылки с полей объектов, полученные через карточный стол. Затем ссылки с полей объектов SOH и LOH с этапа обхода графа. Также ссылки с pre и post plugs, потому что при сохранении туда могли попасть и ссылки. Pinned plugs queue. Ссылки с полей объектов, находящихся в finalization queue. Ссылки с handle tables.

Мы должны выполнить все эти шесть шагов. Вывод - чем сильнее связанность графа, тем дольше работает этот шаг. Нужно обойти все это очень аккуратно, чтобы не промахиваться по кешу в рамках одного участка памяти.

Далее выполняется копирование объектов на их новые места. После того, как мы поменяли все адреса, происходит сама фаза сжатия. Из последних байт gaps берется значение офсета, и на него смещаются все выжившие объекты, чтобы сжать кучу. Естественно, кроме запиненых объектов и тех, которые размещены после них.

После этого восстанавливаются все pre и post plugs участки и исправляем положение поколений (после GC нулевое поколение становится первым, первое - вторым и нулевое сдвигается так, чтобы можно было выделять память дальше).

После этого удаляется и разкоммичеивается память из-под полностью освободившихся сегментов.

Перед каждым запиненым объектом образуется свободный участок. Его надо сохранить для дальнейшего выделения памяти. То есть объект запинен. Слева от него занятое место. И между занятым местом и запиненым объектом - свободные участки. Чтобы их потом переиспользовать, они сохраняются в список свободных участков. Чтобы это сделать нужно убедиться, что есть нужный бакет, либо создать новый, и уже оттуда спустить ссылку на односвязаный список занятых участков определенного размера.

Получается, что задача sweep - создать односвязаный список на свободных участках. Сжатие - это дольше. Чтобы сработал sweep, надо группировать объекты, срок жизни которых совпадает. Это максимум, что мы можем сделать.

Чтобы не мешать GC, мы не используем GCHandle bind. По возможности, мы используем только fixed. В этом случае мы можем избежать пининга, потому что fixed - это просто напоминание GC, что если он здесь очутился, то пинить надо.

Выводы

[In Progress] адаптация курса

Давайте поговорим о выводах.

Первое: снижайте кросс-поколенческую связанность.

Проблема: для оптимизации скорости сборки мусора GC, по возможности, младшее поколение. Он старается это делать часто, чтобы уложится в какие-то миллисекунды. Чтобы сделать это, ему необходима информация о ссылках старших, с карточного стола. Если карточный стол пустой, в особенности если bundle table пустой (потому что именно он покрывает мегабайты своими ячейками), если мы везде встретим нули - это просто отличная информация и GC пролетит максимально быстро. Если на bundle table встречает не ноль, то GC идет на карточный стол, начинает анализировать его, опускается еще ниже: анализируются килобайты памяти, 320 объектов в максимуме, для того, чтобы понять, какой из этих 320 объектов ссылается на более младшее поколение. Поэтому разреженные ссылки, в хаотичном порядке - это самый худший результат.

Как одно из решений - это располагать объекты со связями в младшем поколении рядом, чтобы за них отвечала одна карта. Аллоцировать их группами, выдавая пользовательскому коду по запросу. Если мы так сделаем, то они вместе уйдут во второе поколение (например,

пул) и карточный стол будет пустой. Избегать ссылок в младшее поколение, что уже сложнее.

Нужно не допускать сильной связанности. Как следует из алгоритмов фазы сжатия, для сжатия кучи необходимо обойти дерево и проверить все ссылки, исправляя их на новые значения. Это та еще работа. При этом ссылки с карточного стола затрагивают целую группу объектов. Поэтому общая сильная связанность может привезти к проседанию при GC.

Тут советы простые. Во-первых, располагать сильно-связанные объекты рядом во втором поколении. Это отсылка к карточному столу. Во-вторых, стоит избегать лишних связей в целом. Иногда бывает желание не обращаться через две ссылки к какому-то полю, разместить эту ссылку рядом. Таким образом добавляется еще третья ссылка на объект. А сильная связанность означает, что при сжатии надо будет намного больше этих ссылок обойти и исправить. Ссылок надо делать меньше. В том числе, избегать синтаксического сахара. Зачастую образуются аллокации, которых не видно. Как их увидеть? Можно установить расширение, которое показывает в коде скрытые аллокации. Это расширение в курсе, какие конструкции языка создают лишний трафик. Если хочется оптимизировать нагруженный код, это сильно помогает.

Например, замыкания в некоторой степени зло. Они удобные, но с ними надо быть очень осторожными: замыкания начинают удерживать ссылки.

Если говорить о disposable, то вызов метода CheckDisposed() нужно поставить во все публичные места. Помимо public методов, это еще и protected методы, internal методы - все, что не private. Что самое неприятное, публичным методом является лямбда, которую вы куда-то отдали по подписке. В этот момент она стала публичным методом, теперь ее можно откуда-то дернуть. И туда тоже надо ставить CheckDisposed(). Такие вещи могут породить лишнюю связанность. Когда вы что-то забыли и объект ушел на финализацию, он, через внутренние ссылки, тянет за собой все дерево, весь граф нашего приложения. Объекты, которые должны были быть собраны в нулевом поколении, незаметно уходят во второе.

Следующий совет - мониторьте использование сегментов. Если у нас интенсивно работает приложение, может возникнуть ситуация, когда выделение новых объектов приводит к задержкам. Как это делать? С использованием утилит. Например PerfMon, Sysinternal

Utilities, dotMemory. Именно сегменты лучше смотреть их более системных утилит, типа Sysinternal. Нужно смотреть точки выделения новых сегментов, совпадают ли они с вашими просадками. Что с этим делать? Если речь идет о LOH, то если в нем идет плотный трафик буферов, то, возможно, стоит переключится на использование ArrayPool. Он для этого и был сделан. Вместо того, чтобы постоянно аллоцировать кучу массивов, которые сами по себе тяжелые элементы, используйте ArrayPool.

Если речь идет о SOH, стоит убедиться, что объекты одного времени жизни выделяются рядом, обеспечивая большую вероятность срабатывания sweep вместо collect (compact?). Если они рядом все уйдут, то пусть рядом и создадутся. Если у нас нагруженный код, внутри которого постоянно идут временные аллокации, то их лучше выделять из пула, чем через операцию new - она нагружает GC.

Еще один совет - не выделяйте память в нагруженных участках кода. Если так делать, то GC выбирает окно аллокации, не 1Кб, а 8. И если окну не хватает места, это приводит к GC и расширению заккоммиченной зоны. Плотный трафик новых объектов заставит короткоживущие объекты с других потоков быстро уйти в старшее поколение с худшими условиями сборки мусора. Если у нас плотный трафик, мы не успеваем старое освобождать, поэтому объекты, рассчитанные на существование в нулевом поколении, уходят в первое, где GC работает медленнее. Когда мы говорим "объекты, рассчитанные на существование в нулевом поколении", сражу же понимаем, что создаем объект, который будет жить не дольше секунды. Мы его создали, забили данными и практически сразу отпустили. Он рассчитан на жизнь в нулевом поколении и чем раньше его отпустить, тем лучше. Как это сделать? Не хранить ссылку. Например, у вас есть длинный метод со своей логикой. И вы метод формируете не по принципу действий друг за другом, а по принципу "похожие операции рядом". Тогда получается, что в начале идет инициализация, а внизу эти объекты используются. Возможно, это использование можно поставить повыше. Метод большой, делает кучу всего. А чем он больше, тем выше вероятность срабатывания в процессе GC. Если вы выше использование этих объектов поднимите, выше вероятность того, что эти объекты уйдут с хипа прямо посреди работы этого метода. Если объекты передержать, а GC сработает, эти объекты уйдут в первое поколение.

Полный запрет на использование замыканий в критичных участках кода. Полный запрет боксинга на критичных участках кода. Там, где необходимо создать временный объект под

хранение данных, использовать структуры. Потому что структура ложиться на стеке, ничего не аллоцирует, моментально освобождается. Освобождение структуры требует простого сдвига указателя регистра SP. И не имеет значения, сколько у вас локальных переменных, их освобождение происходит с одинаковой скоростью вне зависимости от их количества. Еще лучше использовать ref struct - это stack only структуры. Для него джиттер может делать оптимизации.

При количестве полей более двух - передавать по референсу. Если в этом случае прокидывать через параметры - будет очень жирно, будет идти копирование. А если перетащить по референсу, то будет передан только указатель, что бесплатно.

Избегайте излишних выделений памяти в LOH. Размещение массивов в этой куче приводит либо к его фрагментации, либо к утяжелению процедуры GC.

Решения. Ипсользовать разделение массивов на подмассивы и класс, который их инкапсулирует. На одном из докладов классная техника рассказывалась. Когда большие массивы в этот хип уходят - это плохо. Можно разделить большой массив на ряд маленьких, сделать массив массивов. То есть, массив, ячейки которого указывают на массивы. И все это ляжет в SOH. Работать с этим легко. Массиву нужно выставить правильную длину, например 2048. Когда мы делает доступ по индексу, вместо того, чтобы делить на 2048, мы сдвигаем на 11 бит. Тем самым делаем очень быстрый доступ к элементам внутренних массивов с эмуляцией непрерывного куска памяти. При этом такие массивы уйдут в SOH, и, если использовать ArrayPool, они лягут во второе поколение и перестанут влиять на сборку мусора.

Также стоит контролировать использование массово double, чтобы они были меньше тысячи. Где оправдано и возможно использовать стек потока?Вместо того, чтобы делать оператор new, класс использовать стек потока. Например, есть ряд сверх-короткоживущих объектов - тех, которые живут в рамках вызова метода. Такие вещи создают трафик, особенно в нагруженных местах. Использование выделения на стеке, во-первых, полностью разгрузит кучу. Выделить память на стеке - это либо либо создать локальную переменную, либо использовать ключевое слово stackalloc. В моей книге есть целый раздел, посвященный этому оператору в главе "Стек потока". Это оператор C#, который

выделяет память не в хипе, а в локальных переменных. Эта операция практически бесплатная и очень быстрая.

Чтобы изучить, как правильно использовать этот оператор, я решил, что надо обратиться к авторам. Открыл исходники и искал по тексту. 90% использования - это тесты. В других местах - очень редко. В частности, я встретил stack list и value stringbuilder. Чем плох обычный stringbuilder? Нам говорят, что для соединения строк плюс - это плохо. Что делает обычный stringbuilder? Это однозсвязный список, внутри которого есть кусочки массивов, которые формируют строку. Когда мы создаем строку, мы туда аппендим. Он заполняет кусочки по очереди. Получается, что вы избавитесь от проблемы фрагментации строками, но не до конца. В этом случае все равно много всего создается. Как избежать?

Большинство случаев использования simple stringbuilder и плюсов - это трассировка. Logger.trace() или logger.debug(). Вы формируете маленькие строчки из разнородных вещей. А value stringbuilder - это очень хорошая вещь, но есть одна проблема: его модификатор доступа internal. Но его можно обойти. Это stringbuilder на стеке. Вещь, которая строит строчку внутри локальный переменных, не аллоцируя вообще ничего. При создании экземпляр на вход ждет span - указатель на некий range памяти. Можно это написать следующим образом: Span x = stackalloc T[ ]. То есть вы среди локальных переменных выделяете память нужного размера и сохраняете в span. И дальше, на основе этого span создаете value stringbuilder, который внутри этого буфера будет строить строчку, не аллоцируя память вообще. Единсвенный случай, когда он аллоцирует память - если он не вместился. В остальных случаях все будет очень хорошо.Второй кейс - если ваш logger.info на вход принимает не строчку, а span, у вас даже строка не будет аллоцирована.

Стоит использовать span memory, где это возможно, потому что эта вещь, которая нас спасает от лишний аллокаций.Освобождайте объекты как можно раньше. Задуманные как короткоживущие объекты могут попасть в gen_1, а иногда и в gen_2. Это приводит к более тяжелому GC, который работает дольше. Поэтому необходимо освобождать ссылку на объект как можно раньше. Если длительный алгоритм содержит код, который работает с какими-то объектами, разнесенными по коду, необходимо сгруппировать, перенося использование ближе. Это увеличит вероятность того, что они будут собраны GC.

Вызывать GC.Collect() не нужно. Часто кажется, что если вызвать GC.Collect(), то это исправит ситуацию. Но намного полезнее выучить алгоритмы работы GC и посмотреть на приложение под тулзой трассировки: dotMemory и другим средством диагностики. Это покажет наиболее нагруженные участки, избавиться от лишних аллокаций, лишнего трафика. Главное, не увлекаться: преждевременная оптимизация тоже может привести к нечитаемости кода.

Еще один совет - избегайте пиннинга. Пиннинг создает кучу проблем. GC ходит вокруг запинненных объектов как по минному полю.

Избегайте финализации. Финализация вызывается не детерменированно. Она может не вызваться. Не вызванный Dispose() приводит к финализации со всеми исходящими ссылками из объекта, который держит другие объекты. Все это переносится в более старшее поколение, усложняя GC, приводя к полному GC во всех поколениях и заменой sweep на compacting.

Нужно аккуратно вызвать Dispose(). Избегать большого количества потоков. Вообще, количество потоков советуют держать в районе количества ядер. Больше нет смысла, если все они работают без ожидания.

Избегайте трафика объектов разного размера. При трафике объектов разного размера и времени жизни возникает фрагментация, как результат - повышение fragmentation ratio, срабатывание collection с изменением адресов во всех ссылающихся объектах. Поэтому решение - если предполагается трафик объектов, надо контролировать наличие лишних полей, приблизив размеры. Также проконтролировать отсутсвие манипуляций со строками. Там, где возможно, заменить readonly span на readonly memory. Освободить ссылку как можно раньше. Не обязательно обнулять, в методах достаточно поднять использование как можно выше.

Исключительные ситуации

Исключения

Ссылка на обсуждение

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 её 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.

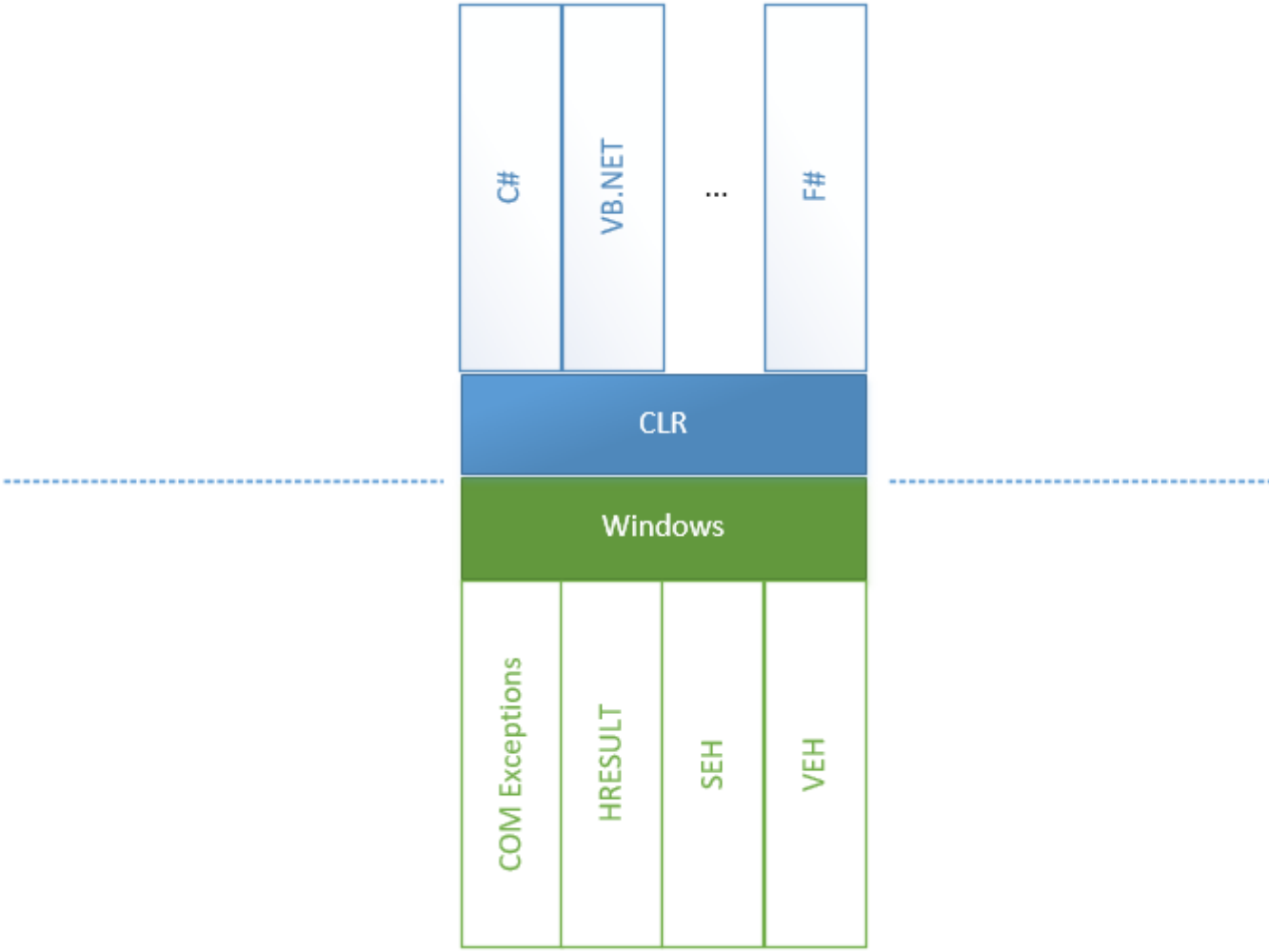
Получается, что для данного метода ситуация получения строки с не корректными данными является исключительной: метод не может вернуть корректного значения, но и вернуть абы что он не может. А потому единственный выход - бросить исключение.

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 вернёт `true`. Если все плохо, вернёт `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 всё-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` создаёт 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 упрощённым, 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 логирование 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 в виду 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)
{
    // ...
}
```

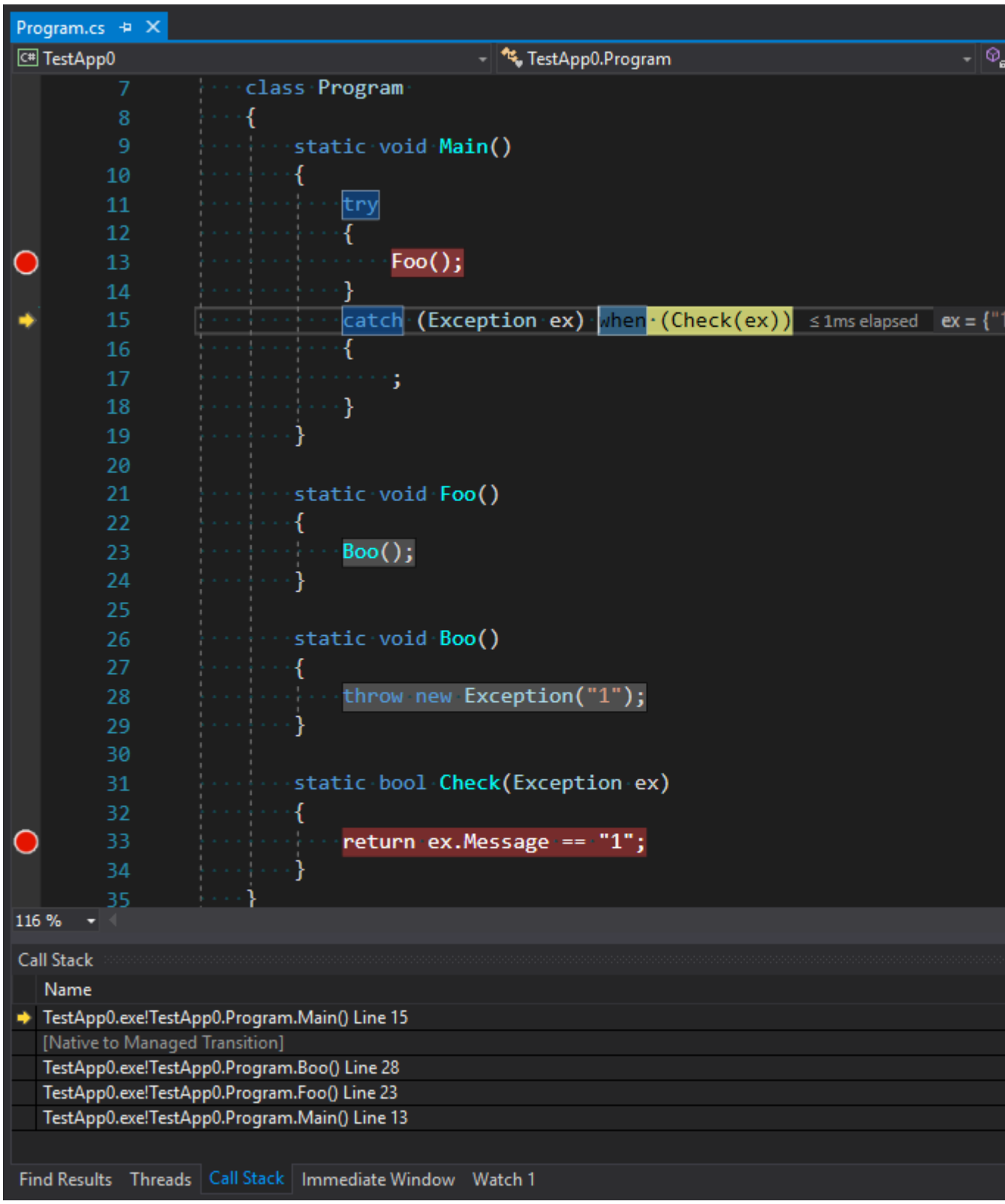
И вопрос улучшения тут вовсе не в отсутствии конструкции `switch`. Новая конструкция как по мне лучше по нескольким пунктам:

- 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 идёт 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 произойдёт развёртка 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.

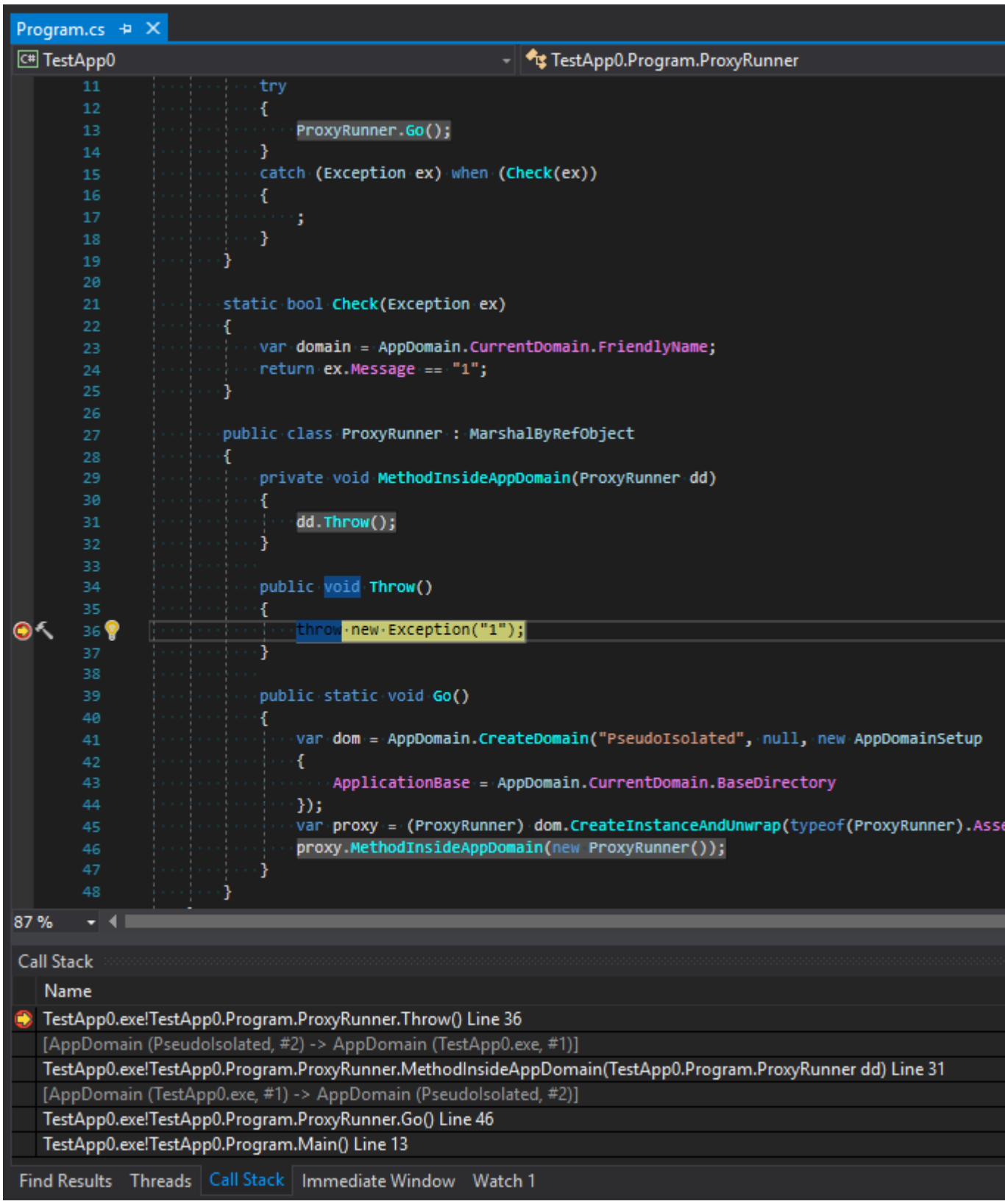
Сериализация

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

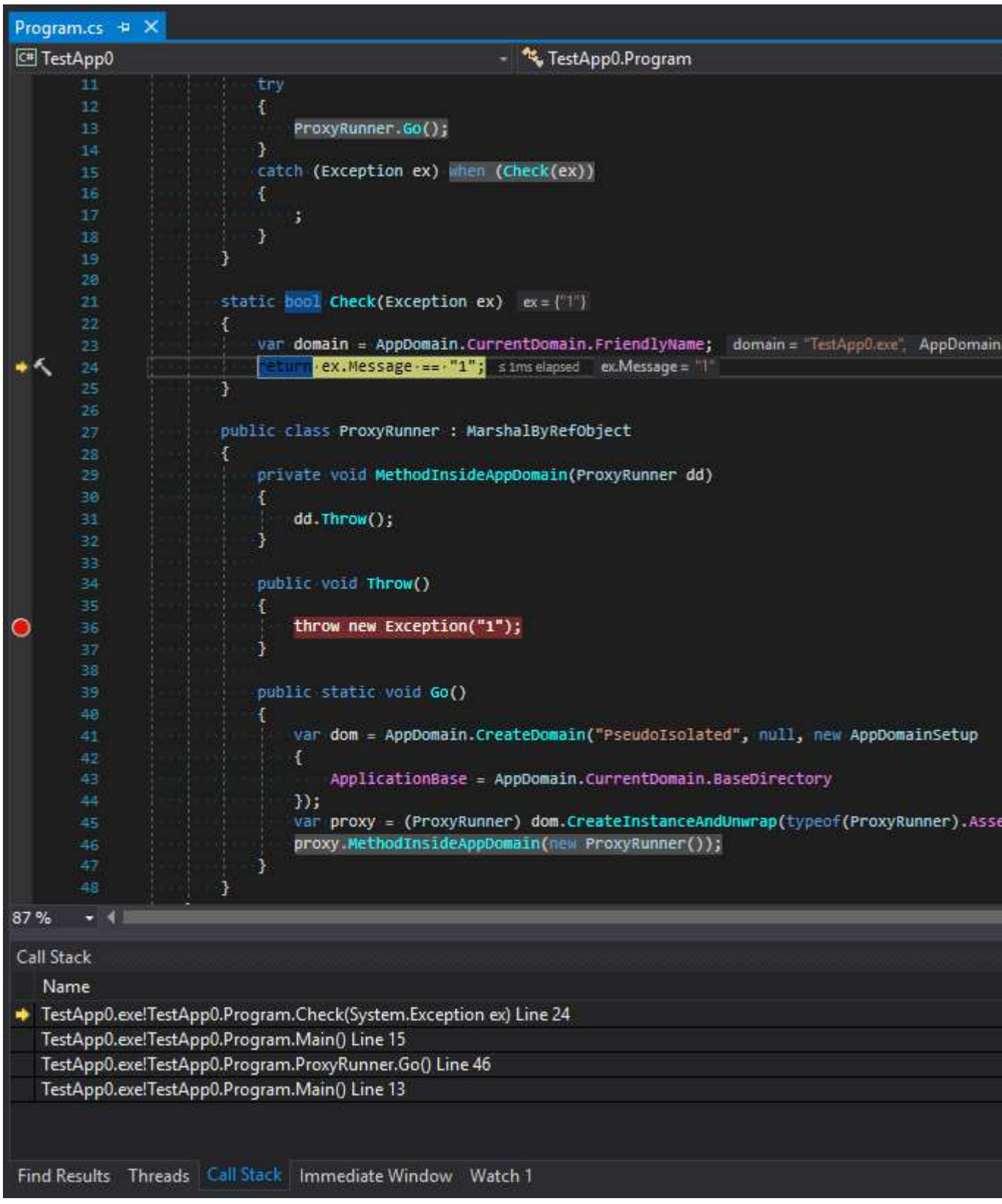
```
class Program
{
    static void Main()
    {
        try
        {
            ProxyRunner.Go();
        }
        catch (Exception ex) when (Check(ex))
        {
            ;
        }
    }

    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 ещё 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 идёт переход в другой домен. Но на самом деле, как по мне так

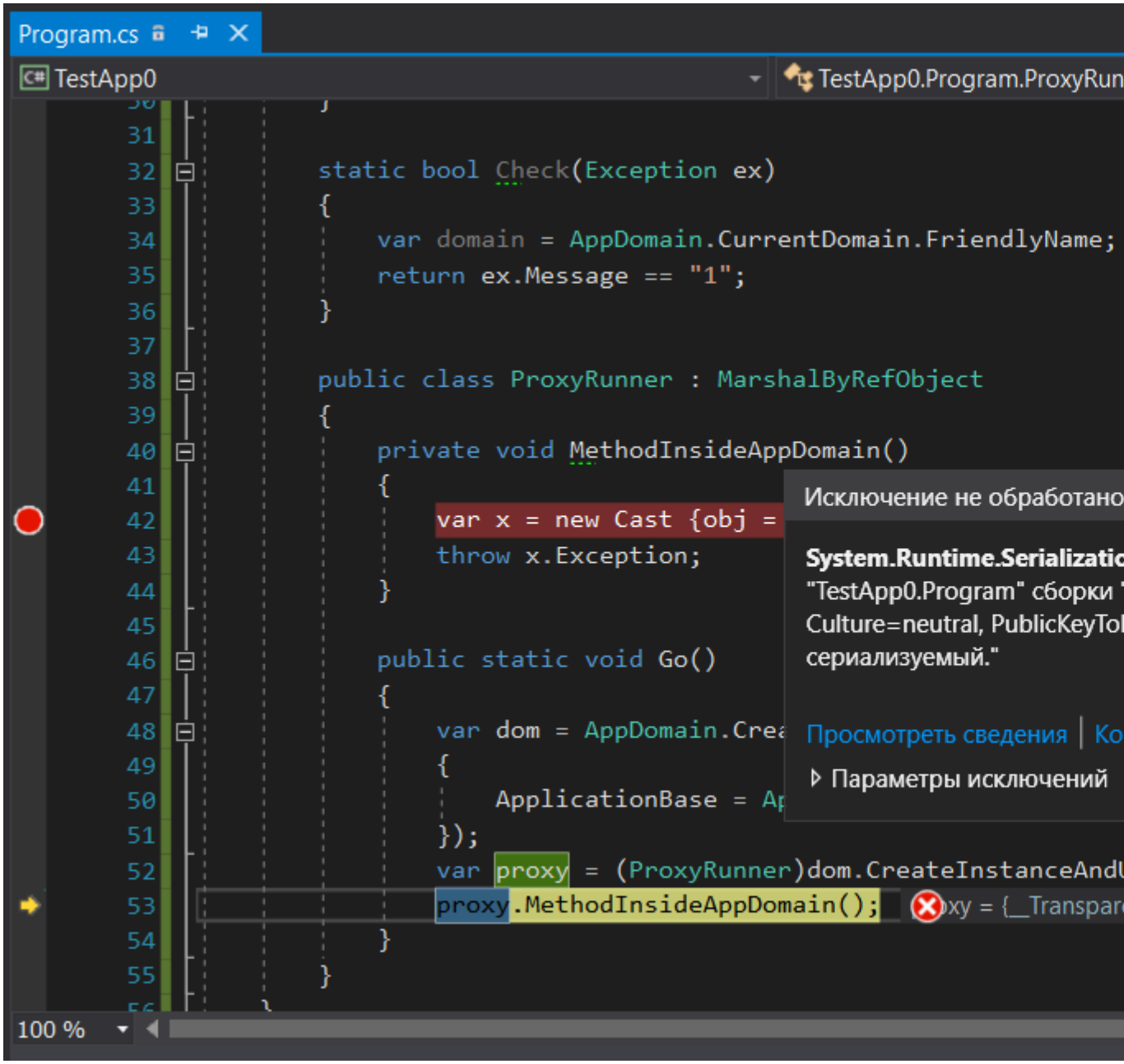
это выглядит несколько странно. Чтобы понять, почему так происходит, вспомним основное правило для типов, организующих взаимодействие между доменами. Эти типы должны наследовать `MarshalByRefObject` плюс - быть сериализуемыми. Однако, как бы ни был строг C#, типы исключений могут быть какими угодно. А что это значит? Это значит, что могут быть ситуации, когда исключительная ситуация внутри дочернего домена может привести к её 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 мучить 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 создаём экземпляр типа `Program` - гарантированно не

сериализуемого и бросаем исключение с ним в виде полезной нагрузки. Хорошие новости заключаются в том, что вы получите обёртку 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();` приведёт 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 приведёт к развёртке

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 создаём 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 пропадёт и CLR начнёт 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 сборки 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 приведён 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 сериализуемый 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 произойдёт: оба случая должны работать стандартно. Чтобы никого не путать.

Архитектура исключительной ситуации

Ссылка на обсуждение

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 найдётся 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 ещё. 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 ещё лучше - построить код таким образом чтобы получение неправильных параметров было бы попросту не возможным.

Ещё 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 её 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 жёстком 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

Ещё 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 уйдёт 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 жёстко 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 встаёт непростая задача: с одной стороны нам лень создавать новый тип исключения, а когда мы всё-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 учётом 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 определённого типа (например, тот же `System.IO.InvalidDataException`), пользователь будет запутан: с одной стороны он увидит источник проблемы в `System.IO` как пространство имён исключения, а с другой - совершенно другое пространство имён точки выброса. Плюс ко всему, задумавшись о правилах выброса этого исключения зайдёт на referencesource.microsoft.com и найдёт все места его выброса:

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

И поймёт 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, объединённых некоторой общей особенностью. Как это сделано, например, с `ArgumentException`, который

под собой объединяет, целую группу исключений путём наследования. Второй серьёзный 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 всё-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 ещё: 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 в виду 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 объединён в функциональные группы: первая занимается кэшированием, вторая - работа с базами данных, третья осуществляет математические расчёты. Через все эти функциональные группы могут идти слои: слой логирования 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 лёгкую 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 имён - прекрасная возможность естественной группировки типов исключений по их поведенческим ситуациям: все, что принадлежит определённым группам там и должно находиться, включая исключения. Мало того, когда вы получите определённое исключение, то помимо названия его типа вы увидите и его пространство имён, что чётко определит его принадлежность. Помните пример плохого переиспользования типа `InvalidDataException`, который на самом деле определён в пространстве имён `System.IO`? Его принадлежность данному пространству имён означает что по сути исключение этого типа может быть выброшено из классов, находящихся в пространстве имён `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 имён, 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(..);
}
```

Причём, 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 имён. Это будет первый слой группировки;
- далее внутри самой сборки может быть одно или несколько различных пространств имён. 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

имён - 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 имён 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 имён, наследуя `JetFinance.FinancialPipe.FinancialPipeExceptionBase`, что говорит о возможном упущении других исключений: это сейчас микросервис `XmlParserService` создаёт 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. Назовём его доменным базовым классом. Домен в данном случае - это обобщающее некоторое количество сборок слово, которое объединяет их по некоторому

глобальному признаку: логирование, 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 имён и объявить базовый тип исключений, которые будут выбрасываться из каждой функциональной зоны. Тут стоит дополнительно орудовать здравым смыслом: если приложение имеет большую вложенность пространств имён, 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 имён, 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 ещё 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

Ещё 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 её 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 не подконтрольных зависимостей;

- Наш собственный код, который был случайным образом введён 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

Ссылка на обсуждение

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 ещё 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 safer
}
```

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 привёл 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 начнёт обрабатываться кодом приложения. Его исполнение несёт 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 произойдёт - это его логирование 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 ещё 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 определённом 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 обеих ситуаций должны отличаться. Если логирование 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 определённом 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, принадлежащем соседнему домену. Ошибки обработки сообщения при этом не могут обрушить рабочие потоки приложения. Плюс отдельно можно послушать UnhandledException домена логирования сообщений: фатальные ошибки при логировании 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, ещё 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 приведёт к сообщению `AppDomain.UnhandledException`, в котором нет информации о том, какой конкретно поток лёг на бок. Если же речь идёт 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 логированием и очисткой ресурсов. Иначе это будет совершенно не правильно: брать и гасить все исключения, как будто их и не было вовсе.

CLR Exceptions

Ссылка на обсуждение

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 произведён 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 ещё можем с ним что-то сделать (т.е. второй, третий и четвёртый 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 всё-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();
        }

        "Caught 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 счётчики 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 вернёмся к навязчивой идее легализации `Thread.Abort()`: мы кинули кувалдой в поток и ожидаем что он с вероятностью 100% оборвётся, но этого может не произойти. Во-первых, становится не понятно как его оборвать в таком случае. Ведь тут все может быть намного сложнее: в повисшем 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 введён 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 завершён не корректно: с `ThreadAbortException`, то при определённых 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 привёл 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 экстремальной 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 приведён прекрасный пример перехода от неуправляемого асинхронного исключения `ThreadAbortException` к управляемому `HttpException` с логгированием ситуации в журнал счётчиков 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 ждём 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 идёт 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 уйдёт во внешний `catch` поскольку `ThreadAbortException` должен войти во все подходящие обработчики. Если он удовлетворяет необходимым условиям, он также

будет обработан путём 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 создаём 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 ещё 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 выгружаемом. 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 своё 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 ждёт: нет доступа к файлу, ошибка парсинга строки и прочие подобные. Наличие асинхронного (в плане возникновения в любом месте кода) исключения создаёт 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 удалён 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.

NullReferenceException

TODO

SecurityException

TODO

OutOfMemoryException

TODO

Corrupted State Exceptions

После становления платформы и её популяризации, после того как огромная масса программистов 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 свёрстанный 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 не удобными 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 ещё 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 разберёмся, что это за исключительные ситуации, а саму историю ещё 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 который нет прав. Память с точки зрения работать можно далеко не со всем диапа "островками", которые были выделены опе также с теми, на которые хватает прав (напр которыми владеет только операционная с только исполнять код но не читать как данн
STATUS_STACK_OVERFLOW	Эта ошибка известна всем: ошибка нехватки под вызов очередного метода

Код ошибки	**Описание**
EXCEPTION_ILLEGAL_INSTRUCTION	Очередной код, считанный процессором и распознан как инструкция
EXCEPTION_IN_PAGE_ERROR	Поток предпринял попытку работы со стра не существует
EXCEPTION_INVALID_DISPOSITION	Механизм обработки исключений вер обработчик. Такое исключение никогда н программах, написанных на высокоуровне C++)
EXCEPTION_NONCONTINUABLE_EXCEPTION	Поток сделал попытку продолжить исполне возникновения исключения, продолжить и которого не возможно. Тут имеется catch/fault/finally, а подобие фильтров позволяют исправить ошибку, которая при предпринять ещё одну попытку выполни ошибке
EXCEPTION_PRIV_INSTRUCTION	Попытка выполнить привилегированную ин
STATUS_UNWIND_CONSOLIDATE	Исключение, относящееся к размотке ст 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 защищённую область памяти. Здесь под словом "защита" лучше понимать именно попытку работы с ещё не выделенным участком памяти или же уже освобождённым. 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 идёт 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 серьёзная 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 ещё 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 нас
.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` всё-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, введённой в фреймворк с самого начала: .NET - он только для .NET. Unsafe код - это не норма, а вынужденное состояние, а потому идеологичность наличия перехвата `AccessViolationException` идёт вразрез с идеологией понятия фреймворк - как платформа (т.е. имитация полной песочницы со своими законами). Однако мы все ещё находимся в реалиях платформы, на которой работаем и во многих ситуациях перехватывать это исключение все ещё 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 приведёт 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 упёрлись в следующий занятый участок памяти и нет возможности выделить следующую страницу виртуальной памяти) или же поток упёрся в разрешённый 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 развёрткой стека до ожидающего такого поведения участка кода. Хм... Ещё раз: мы ждём, что на следующем участке ничего не отработает потому что в стеке кончится память. Как по мне, это полный абсурд.