Основные принципы построения архитектуры .NET. Сравнительная характеристика технологий .NET и Java, а также промежуточного языка IL и байт-кода Java.

Основные принципы .NET:

1) Переносимость программ между аппаратурой и ОС благодаря промежуточному коду и динамической компиляции

2) Модульное программирование на основе сборок с версионностью и полной информацией о типах

3) Автоматическое управление памятью на основе сборки мусора

4) Объектная модель с ссылочными и скалярными типами в основе сборки мусора

5) Открытость и расширяемость, поддержка многих языков.

Подобно Java, .NET создает промежуточный язык(IL). .NET поддерживает динамическую компиляцию. Структурные типы данных в .NET могут размещаться как на стеке, так и в динамической памяти(управляемой куче). .NET поддерживает создание как многомерных массивов(прямоугольных), так и нерегулярных массивов(массивы массивов, зубчатые, ступенчатые). В Java все методы объектов вызываются как виртуальные методы, в то время как .NET поддерживает как виртуальные, так и не виртуальные методы. Для возврата результатов подпрограмм Java использует возвращаемое значение, .NET – возвращаемое значение, а также ref и out параметры. Для работы приложения на Java необходимо наличие JRE, для работы .NET требуется наличие CLR.

Байт-код Java был создан для интерпретации, в то время как IL-код необходим для динамической компиляции JIT-компилятором. IL-код поддерживает несколько ЯП, в то время как байт-код был спроектирован специально под Java.

Сборки (assembly) в платформе .NET. Проблема версионности сборок и ее решение. Строго именованные и нестрого именованные сборки. Глобальный кэш сборок (GAC).

Среда CLR работает со сборками. Сборка обеспечивает логическую группировку одного или нескольких управляемых модулей или файлов ресурсов. Файлы, которые входят в сборку, общедоступные экспортируемые типы, реализованные в файлах сборки, а также относящиеся к сборке файлы ресурсов или данных описываются в манифесте. Манифест представляет собой набор таблиц метаданных.

Манифест содержит имя сборки, сведения о версии, региональные стандарты, флаги, алгоритм хеширования, открытый ключ издателя, по одной записи для каждого PE-файла и файла ресурсов, входящих в состав сборки, по одной записи для каждого ресурса, включенного в сборку, записи для всех всех открытых типов, экспортируемых модулями сборки.

CLR поддерживает два вида сборок: с нестрогими именами и со строгими именами. Сборки со строгими именами подписаны при помощи пары ключей, уникально идентифицирующей издателя сборки. Эта пара ключей позволяет уникально идентифицировать сборку, обеспечивать её безопасность, управлять её версиями.

У сборки со строгим именем четыре атрибута, уникально её идентифицирующих: имя файла, номер версии, идентификатор регионального стандарта и открытый ключ. Для отличия сборки, созданной одной компанией, от сборок, созданными другими компаниями применяются стандартные криптографические технологии. Первое — пара ключей, открытый ключ связывается со сборкой. Второе — посчитанный хеш-код всего содержимого PE-файла с манифестом. Хеш-код подписывается закрытым ключом, полученная подпись записывается в зарезервированный раздел PE-файла. Комбинация имени файла, версии сборки, региональных стандартов и значения открытого ключа составляют строгое имя сборки.

Сборки, развертываемые в том же каталоге, что и приложение, называются сборками с закрытым развертыванием, так как файлы сборки не используются совместно другими приложениями.

Глобальный кэш сборок — место, где располагаются совместно используемые сборки. Это общеизвестный каталог, который CLR автоматически проверяет при обнаружении ссылки на сборку. В GAC попадают только сборки со строгими именами. Главное преимущество — при использовании специального инструмента для установки сборки в GAC, для сборок с одинаковыми названиями создаются отдельные папки, что исключает перезапись файлов.

Объектная модель в платформе .NET и языке C#. Общая система типов данных в платформе .NET. Ссылочные и скалярные (value-type) типы данных. Упаковка и распаковка скалярных типов данных в платформе .NET.

В CLR каждый объект прямо или косвенно является производным от System.Object. Благодаря этому любой объект любого типа гарантированно имеет минимальный набор методов. Методы класса System.Object:

```
public class Object
{
    public Object();
    ~Object(); // virtual void Finalize();

    public static bool Equals(Object objA, Object objB);
    public static bool ReferenceEquals(Object objA, Object objB);
    public virtual bool Equals(Object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
    protected Object MemberwiseClone();
}
```

Equals – возвращает true, если два объекта имеют одинаковые значения

GetHashCode – возвращает хеш-код для значения данного объекта.

ToString – по умолчанию возвращает полное имя типа(this.GetType().FullName).

GetType – возвращает экземпляр объекта, производного от Type, который идентифицирует тип объекта, вызвавшего GetType.

MemberwiseClone – создает новый экземпляр типа и присваивает полям нового объекта соответствующие значения объекта this. Возвращается ссылка на созданный экземпляр

Finalize – виртуальный метод, вызываемый когда уборщик мусора определяет, что объект является мусором, но до возвращения занятой памяти объектом в кучу.

CLR требует, чтобы все объекты создавались оператором new.

Типы данных, которые поддерживаются компилятором напрямую, называются примитивными; у них существуют прямые аналоги в библиотеке классов .NET FCL. Например типу данных int языка C# соответствует System.Int32.

CLR поддерживает две разновидности типов: ссылочные(reference types) и значимые(value types).

Память для ссылочных типов всегда выделяется из управляемой кучи; каждый объект, размещаемый в куче, содержит дополнительные члены(указатель на объект-тип, индекс блока сихронизации), подлежащие инициализации; незанятые полезной информацией байты объекта обнуляются; размещение объекта в управляемой куче со временем инициирует сборку мусора. К ссылочным типам относятся interface, class, delegate, record.

Экземпляры значимых типов обычно размещаются в стеке потока. В представляющей экземпляр переменной нет указателя на экземпляр; поля экземпляра размещаются в самой

переменной. К значимым типам относятся структуры и перечисления(struct, enum). Все значимые типы должны быть производными от System.ValueType. Все перечисления являются производными от System.Enum, производного от System.ValueType.

Отличия значимых и ссылочных типов:

1) Объекты значимого типа существуют в двух формах: упакованной и неупакованной. Ссылочные типы бывают только в упакованной форме.

2) Значимые типы являются производными от System.ValueType.

3) В объявлении значимого типа или ссылочного типа нельзя указывать значимый тип в качестве базового класса. Методы значимого типа неявно являются запечатанными.

4) Переменные ссылочного типа содержат адреса объектов в куче, значение по умолчанию — null. В переменной значимого типа всегда содержится некоторое значение соответствующего типа, при инициализации всем членам присваивается 0.

5) Когда переменной значимого типа присваивается другая переменная значимого типа, выполняется копирование всех её полей. Когда переменная ссылочного типа присваивается переменная ссылочного типа, копируется только её адрес.

6) Несколько переменных ссылочного типа могут ссылаться на один объект в куче. Каждая переменная значимого типа имеет собственную копию данных объекта.

7) Так как неупакованные значимые типы не размещаются в куче, отведенная для них память освобождается сразу после возвращения управления методом, в котором описан экземпляр этого типа.

Для преобразования значимого типа в ссылочный служит упаковка. При упаковке происходит следующее:

1) В управляемой куче выделяется память. Её объем определяется размером значимого типа и двумя дополнительными членами.

2) Поля значимого типа копируются в память, только что выделенную в куче.

3) Возвращается адрес объекта. Этот адрес является ссылкой на объект, то есть значимый тип превращается в ссылочный.

Распаковка происходит в два этапа: сначала извлекается адрес полей из упакованного объекта, затем значения полей копируются из кучи в экземпляр значимого типа, находящийся в стеке. Также при упаковке если переменная содержащая ссылку на упакованный значимый тип, равна null, генерируется NullReferenceException; если ссылка указывает на объект, не являющийся упакованный значением требуемого значимого типа, генерируется исключение InvalidCastException.

Утилизация динамической памяти. Модель с явным освобождением памяти.

Это наиболее распространенная модель. В распоряжении программиста есть две процедуры или два оператора, с помощью которых он может соответственно запрашивать и освобождать блоки памяти.

В модели с ручным освобождением памяти система не следит за наличием или отсутствием ссылок на объекты. Программист должен сам заботиться об уничтожении ненужных объектов и о возвращении их памяти системе.

Когда программа создает объект менеджер памяти просматривает список имеющихся свободных блоков в поисках блока, подходящего по размеру. Как только такой блок найден, он изымается из списка свободных блоков и его адрес возвращается программе. После уничтожения программой объекта менеджер памяти добавляет освобожденную память в список свободных блоков. Список свободных блоков является двусвязным. После добавления в него освобождаемого блока памяти система выполняет дефрагментацию, сливая смежные свободные блоки в один.

Достоинства:

1) Детерминизм — временные задержки на выделение и освобождение памяти предсказуемы.

Недостатки:

1) Ненадежность и подверженность ошибкам. Очень трудно поддерживать соответствие операторв delete операторам new, поэтому выделенная память может вообще никогда не освобождаться, происходит утечка памяти.

2) Зависание ссылок. Суть в том, что в программе остаются ссылки на уничтоженные объекты, что может привести к нарушению целостности данных.

3) Фрагментация. Выделенные блоки памяти перемежаются занятыми блоками при интенсивном выделении памяти.

Утилизация динамической памяти. Модель со счётчиками ссылок.

В модели со счетчиком ссылок с каждым объектом ассоциируется целочисленный счетчик ссылок. Обычно он хранится в одном из полей объекта. При создании объекта счетчик устанавливается в нулевое значение, а потом увеличивается на единицу при создании каждой новой ссылки на объект. При пропадании каждой ссылки значение счетчика уменьшается на единицу, и когда оно становится равным нулю, объект уничтожается.

Увеличение и уменьшении счетчика происходит с помощью методов AddRef и Release. AddRef вызывается при любом копировании ссылки, а также при её передаче в качестве параметра подпрограммы. Release вызывается при пропадании или обнулении ссылки.

Недостатки:

1) Накладные расходы на копирование ссылок.

2) Счетчики ссылок не учитывают возможности циклических связей между объектами. Для решения этой проблемы ссылки делятся на два вида: сильные и слабые. Сильные влияют на счетчик ссылок, слабые — нет. При уничтожении объекта слабые ссылки обнуляются. Для доступа к объекту слабую ссылку надо превратить в сильную. Из-за такой реализации увеличивается потреблении памяти и замедляется доступ к объектам.

3) Возможные ошибке в выборе между сильными и слабыми ссылками.

Достоинства:

1) Детерминизм

Утилизация динамической памяти. Модель с иерархией владения.

Модель с иерархией владения основана на том, что при создании любого объекта ему назначается объект-владелец, отвечающий за уничтожение подчиненных объектов Создав объект и назначив ему владельца, можно не заботиться, что ссылки на него пропадут. Этот объект будет обязательно уничтожен при удалении владельца. Владелец отвечает за уничтожение подчиненных объектов.

Объект можно уничтожить принудительно, даже если у него есть владелец. При этом объект либо изымается из списка подчиненных объектов своего владельца, либо помечается как уничтоженный для предотвращения повторного удаления.

Объект может быть создан без владельца, и тогда он требует явного уничтожения.

Модель с иерархией владения не избавляет полностью от необходимости явно освобождать память, однако значительно сокращает риск утечек памяти. Эта модель также не решает проблему фрагментации памяти, но позволяет более успешно бороться с зависшими указателями, например, путем рассылки сообщений об уничтожении объектов по иерархии. Обрабатывая эти сообщения, объекты-получатели могут обнулять сохраненные ссылки на уничтожаемые объекты.

Утилизация динамической памяти. Модель с владеющими ссылками.

Модель памяти, применяемая в ЯП Rust. Основные правила:

1) У каждого значения есть переменная, называемая владельцем

2) Одновременно может быть только один владелец

3) Когда владелец уходит за пределы области видимости, значение удаляется.

В этой модели память автоматически возвращается, как только владеющая памятью переменная выходит из области видимости.

Возникает ошибка двойного освобождения, когда обе переменных выходят из области видимости, они обе будут пытаться освободить одну и ту же память в куче:

let s1 = String::from(“hello”);

let s2 = s1;

Для борьбы с этим после строки let s2 = s1; s1 считается более недействительной. Следовательно, когда s1 выходит за пределы области видимости, не нужно ничего освобождать. Поскольку первая переменная аннулируется, вместо копирования происходит перемещение(s1 перемещен в s2).

Можно создавать невладеющие ссылки, которые в реализации являются ссылками на владеющие.

Утилизация динамической памяти. Модель с автоматической сборкой мусора.

Модель с автоматической сборкой мусора предусматривает лишь возможность создавать объекты, но не уничтожать их. Система сама следит за тем, на какие объекты ещё имеются ссылки, а на какие уже нет. Когда объекты становятся недостижимы через имеющиеся в программе ссылки, их память автоматически возвращается системе.

Эта работа периодически выполняется сборщиком мусора и происходит в две фазы. Сначала сборщик мусора находит все достижимые по ссылкам объекты и помечает их. Затем он перемещает их в адресном пространстве для устранения фрагментации.

Обход графа достижимых объектов начинается с корней, к которым относятся все глобальные ссылки и ссылки в стеках имеющихся программных потоков. Сборщик мусора также выясняет, где внутри объектов имеются ссылки на другие объекты. Следуя по этим ссылками, он обходит все цепочки объектов и выясняет, какие блоки памяти стали свободными. После этого достижимые по ссылкам объекты перемещаются для устранения фрагментации.

Достоинства:

1) Решение проблем утечек памяти, фрагментации памяти, зависших указателей.

2) Скорость выделения памяти.

Недостатки:

1) Недетерминизм

2) Возможные задержки в работе программы.

3) Легальные утечки памяти. Такой вид утечек памяти характерен для программного кода, в котором одними объектами регистрируются обработчики событий в других объектах, в результате чего ассоциированные объекты остаются в памяти.

Утилизация динамической памяти. Модель с автоматической сборкой мусора и явным освобождением памяти.

Модель с автоматической сборкой мусора и явным освобождением памяти сочетает в себе два качества:

1) Быстрая автоматическая сборка мусора

2) Безопасное принудительное освобождение памяти

Наличие сборки мусора означает, что система следит за потерей ссылок на объекты и устраняет утечку памяти. Наличие безопасного принудительного освобождения памяти означает, что программист вправе уничтожить объект, при этом память объекта возвращается системе, а все имеющиеся на него ссылки становятся недействительными.

В такой системе на каждое машинное слово отводится два дополнительных бита, называемых битами тегов. Значения этих битов показывают, свободно ли машинное слово или занято, и если занято, то хранится ли в нем указатель или скалярное значение. Этими битами управляет аппаратура и ОС. Созданием объектов занимается система, которая размещает в памяти объекты и создает ссылки на них. При уничтожении объектов теги памяти устанавливаются в состояние, запрещающее доступ. Попытка обратиться к свободной памяти по зависшему указателю приводит к аппаратному прерыванию. Поскольку вся память помечена тегами, сборщику мусора нет необходимости анализировать информацию о типах, чтобы разобраться, где внутри объектов располагаются ссылки на другие объекты, а также ему почти не нужно тратить время на поиск недостижимых объектов.

Основные принципы:

1) Выделение динамической памяти выполняется оператором new. Выделенная память автоматически инициализируется нулями и всегда привязывается к типу созданного в памяти объекта

2) Уничтожение объекта выполняется автоматически при пропадании всех ссылок на объект Для дефрагментации периодически выполняется сборка мусора.

3) Объекты можно уничтожать принудительно. В результате этого все ссылки становятся недействительными, попытка доступа к объекту приводит к исключительной ситуации.

Механизм сборки мусора в платформе .NET. Построение графа достижимых объектов сборщиком мусора в платформе .NET. Поколения объектов.

Механизм уборки мусора представляет возможность уничтожения объектов, которые перестали быть нужными приложению.

CLR использует алгоритм отслеживания ссылок. Все переменные ссылочных типов называются корнями. Когда среда CLR запускает уборку мусора, она сначала приостанавливает все программные потоки в процессе. Затем CLR переходит к этапу сборки мусора, называемому маркировкой. CLR перебирает все объекты в куче, задавая биту в поле индекса блока синхронизации значение 0. Это означает, что все объекты могут быть удалены. Затем CLR проверяет все активные корни и объекты, на которые они ссылаются. Если корень содержит null, CLR игнорирует его и переходит к следующему корню.

Если корень ссылается на объект, в поле индекса блока синхронизации устанавливается бит — это и есть признак маркировки. После маркировки объекта CLR проверяет все корни в этом объекте и маркирует объекты, на которые они ссылаются.

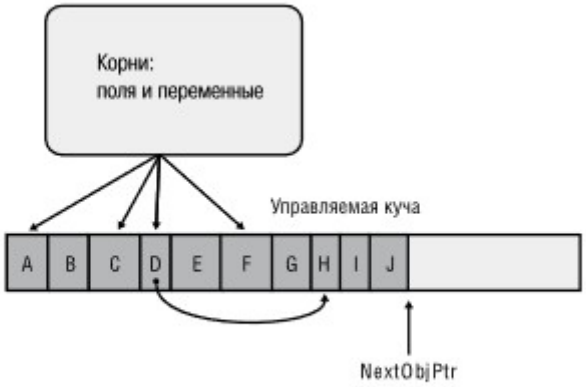


Рис. 21.2. Управляемая куча до уборки мусора

Маркированные объекты переживают сборку мусора.

Теперь начинается фаза уборки мусора, называемая сжатием. В этой фазе CLR перемещает вниз все немусорные объекты, чтобы они занимали смежный блок памяти. В фазе сжатия CLR вычитает из каждого корня количество байт, на которое объект был сдвинут вниз в памяти, гарантируя, что каждый корень будет ссылаться на тот же объект, что и прежде. После завершения фазы сжатия возобновляется выполнение потоков приложения.

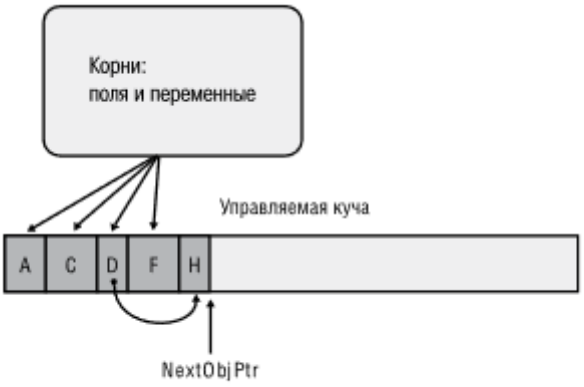


Рис. 21.3. Управляемая куча после уборки мусора

Сразу после инициализации в управляемой куче нет объектов. Создаваемые в куче объекты составляют поколение 0 или же к поколению 0 относятся только что созданные объекты, которых не касался уборщик мусора

Рис. 21.4. Вид кучи сразу после инициализации: все объекты в ней относятся к поколению 0, уборка мусора еще не выполнялась

При инициализации CLR выбирает пороговый размер для поколения 0. Если в результате выделения памяти для нового объекта размер поколения 0 превышает пороговое значение, должна начаться уборка мусора.

Объекты, пережившие уборку мусора, становятся поколением 1. Объекты из поколения 1 были проверены уборщиком мусора один раз.

Рис. 21.5. Вид кучи после одной уборки мусора: выжившие объекты из поколения 0 переходят в поколение 1, поколение 0 пустует

После уборки мусора объектов в поколении 0 не остается. Туда помещаются новые объекты.

Рис. 21.6. В поколении 0 появились новые объекты, в поколении 1 — мусор

Начиная уборку мусора, уборщик определяет, сколько памяти занято поколением 1. Пока поколение 1 занимает намного меньше отведенной памяти, поэтому уборщик проверяет только поколение 0. Объекты из поколения 0, пережившие уборку мусора, переходят в поколение 1.

Рис. 21.7. Вид кучи после двух операций уборки мусора: выжившие объекты из поколения 0 переходят в поколение 1 (увеличивая его размер), поколение 0 пустует

Допустим, что поколение 1 выросло до таких размеров, что все его объекты в совокупности превысили пороговое значение. Поколение 0 также заполняется и начинается уборка мусора. Поэтому теперь уборщик мусора проверяет все объекты поколений 1 и 0.

Рис. 21.11. Вид кучи после четырех операций уборки мусора: выжившие объекты из поколения 1 переходят в поколение 2, выжившие объекты из поколения 0 переходят в поколение 1, поколение 0 снова пусто

Все выжившие объекты поколения 0 теперь находятся в поколении 1, а все выжившие объекты поколения 1 — в поколении 2.

Управляемая куча поддерживает только три поколения: 0, 1 и 2. При инициализации CLR устанавливается пороговое значение для всех трех поколений. В процессе работы пороговое значение адаптируется. Например, если после обработки поколения 0 уборщик обнаруживает множество выживших объектов, порог для поколения 0 можно поднять.

Также CLR делит объекты на большие и малые. Любые объекты размеров 85000 байт и более считаются большими. Для них память выделяется в отдельной части адресного пространства, к ним не применяется сжатие, они всегда считаются частью поколения 2.

Существует два основных режима уборки мусора:

1) Режим рабочей станции. Оптимизирован для минимизации времени приостановки потоков приложения.

2) Режим сервера. Управляемая куча разбивается на несколько разделов — по одному на процессор. Изначально уборщик использует один поток на один процессор. Каждый поток выполняется в собственном разделе одновременно с другими потоками.

Также существуют два подрежима: параллельный и непараллельный. В параллельном режиме у уборщика мусора есть дополнительный фоновый поток, выполняющий пометку объектов во время работы приложения.

Завершение объектов в платформе .NET. Метод Finalize и проблема недетерминизма при его вызове. Список завершаемых объектов (finalization queue) и очередь завершения (freachable queue).

Если тип, использующий системный ресурс, будет уничтожен в ходе уборки мусора, занимаемая объектом память вернется в управляемую кучу; однако системный ресурс, о котором сборщику мусора ничего не известно, будет потерян. CLR поддерживает механизм финализации, позволяющий объекту выполнить корректную очистку, прежде чем уборщик мусора освободит занятую им память.

Когда уборщик определяет, что объект подлежит уничтожению, он вызывает метод Finalize этого объекта(если он переопределен).

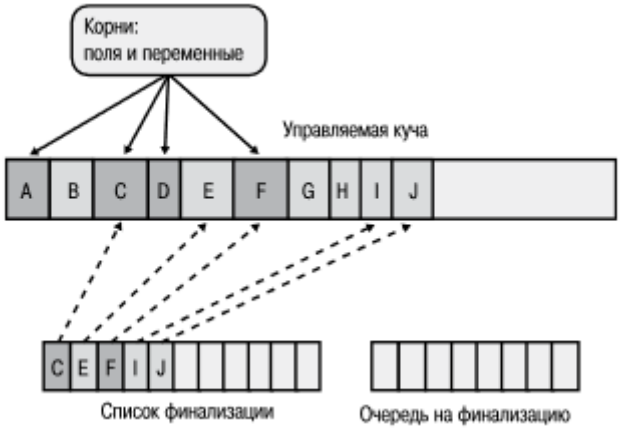
```
internal sealed class SomeType {
        // Метод финализации
        ~SomeType() {
        }
}
```

Код в теле метода генерируется в блок try, а вызов метода base.Finalize – в блок finally.

Память объектов, имеющих метод Finalize, не может быть освобождена немедленно. Финализируемый объект должен пережить уборку мусора, поэтому он переводится в другое поколение.

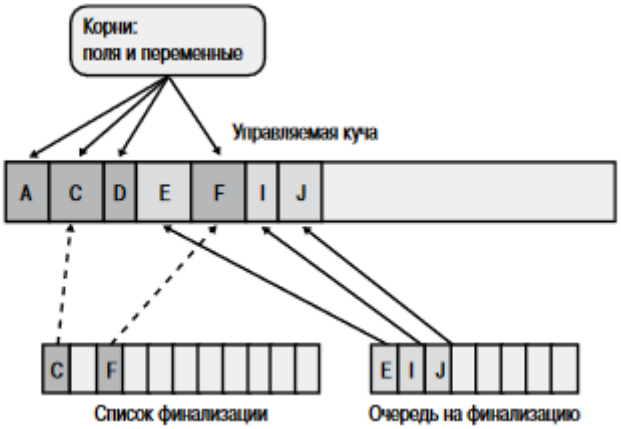
Разработчик не знает, в какой именно момент будет выполнен метод Finalize, и не может управлять его выполнением. CLR не дает никаких гарантий относительно порядка вызова методов Finalize. Для вызова методов Finalize CLR использует специальный высокоприоритетный поток. Если метод Finalize блокируется, специальный поток не сможет вызвать методы Finalize. Если метод Finalize выдает необработанное исключение, процесс завершается; перехватить такое исключение невозможно. Также не гарантии, что метод Finalize вообще будет вызван, так как при завершении программы каждому методу Finalize дается лишь 2 секунды на выполнение, а всем методам вместе — 40 секунд на выполнение.

Если в типе объекта определен метод финализации, непосредственно перед вызовом конструктора экземпляра типа указатель на объект помещается в список финализации. Каждая запись списка указывает на объект, для которого нужно вызвать метод финализации, прежде чем освободить занятую им память.



Сначала уборщик мусора определяет мусорные объекты. Уборщик сканирует список

финализации в поисках указателей на эти объекты. Обнаружив указатель, он извлекает его из списка финализации и добавляет в конец очереди на финализацию. Каждый указатель в этой очереди идентифицирует объект, готовый к вызову своего метода финализации.



Если объект недоступен, уборщик считает его мусором. Далее, когда уборщик перемещает ссылку на объект из списка финализации в очередь на финализацию, объект перестает считаться мусором, а это означает, что занятую им память освобождать нельзя.

Вызванный снова уборщик обнаруживает, что финализированные объекты стали мусором, так как ни корни приложения, ни очередь на финализацию больше на них не указывают. Память, занятая этими объектами, освобождается. Для освобождения памяти, занятой объектами, требующими финализации, уборку мусора нужно выполнить минимум дважды, так как объекты переходят в следующее поколение.

Модель детерминированного освобождения ресурсов в платформе .NET. Интерфейс IDisposable и его совместное использование с методом Finalize.

Для решения проблемы непредсказуемого вызова метода Finalize в среде .NET используется детерминированное завершение жизни объектов через интерфейс IDisposable:

```
public interface IDisposable {
        void Dispose();
}
```

Вызов Dispose позволяет управлять тем, когда произойдет освобождение системных ресурсов. Вызов Dispose не удаляет управляемый объект из управляемой кучи.

Проблемы IDisposable:

1) После вызова Dispose в программе могут оставаться ссылки на объект, находящийся уже в некорректном состоянии. Программе никто не запрещает обращаться по этим физически доступными, но логически зависшими ссылками и вызывать у некорректного объекта различные методы.

2) Метод Dispose может вызываться повторно, в том числе рекурсивно

3) В программе с несколькими вычислительными потоками может происходить асинхронный вызов метода Dispose для одного и того же объекта.

Для решения проблем метода Dispose было предписано следующее:

1) Определять в объекте булевский флаг, позволяющий выяснить, работал ли в объекте код завершения, и игнорировать повторные вызовы метода Dispose, проверяя упомянутый булевский флаг

2) В программах с несколькими вычислительными потоками блокировать объект внутри метода Dispose на время работы кода завершения

3) В начале public-методов проверять, что объект уже находится в завершенном состоянии, и в этом случае создавать исключение класса ObjectDisposedException

```
public class LogFile : Object, IDisposable
{
    private StreamWriter writer;
    private bool disposed;

    public LogFile(string filePath)
    {
        writer = new StreamWriter(filePath, append: true);
    }

    public void Dispose()
    {
        if (!disposed)
        {
            writer.Close();
            disposed = true;
        }
    }

    public void Write(string str)
    {
        if (disposed)
            throw new ObjectDisposedException(this.ToString());

        writer.Write(str);
    }
}
```

```
LogFile f = new LogFile("Log.txt");
try
{
    f.Write("Ключ на старт");
    f.Write("Протяжка-1");
    f.Write("Продувка");
}
finally
{
    f.Dispose();
}

// Короткий эквивалент в C# с оператором using:

using (var f = new LogFile("Log.txt"))
{
    f.Write("Ключ на старт");
    f.Write("Протяжка-1");
    f.Write("Продувка");
}
```

Если класс одновременно реализует метод Finalize и интерфейс IDisposable, то в методе Dispose нужно отменять вызов метода Finalize путем вызова метода GC.SuppressFinalize.

«Слабые» ссылки (тип данных WeakReference) и их применение.

Слабая ссылка — специфический вид ссылок на динамически создаваемые объекты в системах со сборкой мусора. Типичная ссылка на объект является очень детерминированной: пока имеется ссылка на объект, он будет продолжать существовать. Сборщик мусора не учитывает связь ссылки и объекта в куче при выявлении объектов, подлежащих удалению Слабая ссылка позволяет работать с объектом как и строгая ссылка, но при необходимости объект будет удален, даже при наличии слабой ссылки на него.

Порядок использования:

1) Когда требуется сохранить слабую ссылку, создается экземпляр класса WeakReference, которому передается обычная ссылка на целевой объект.

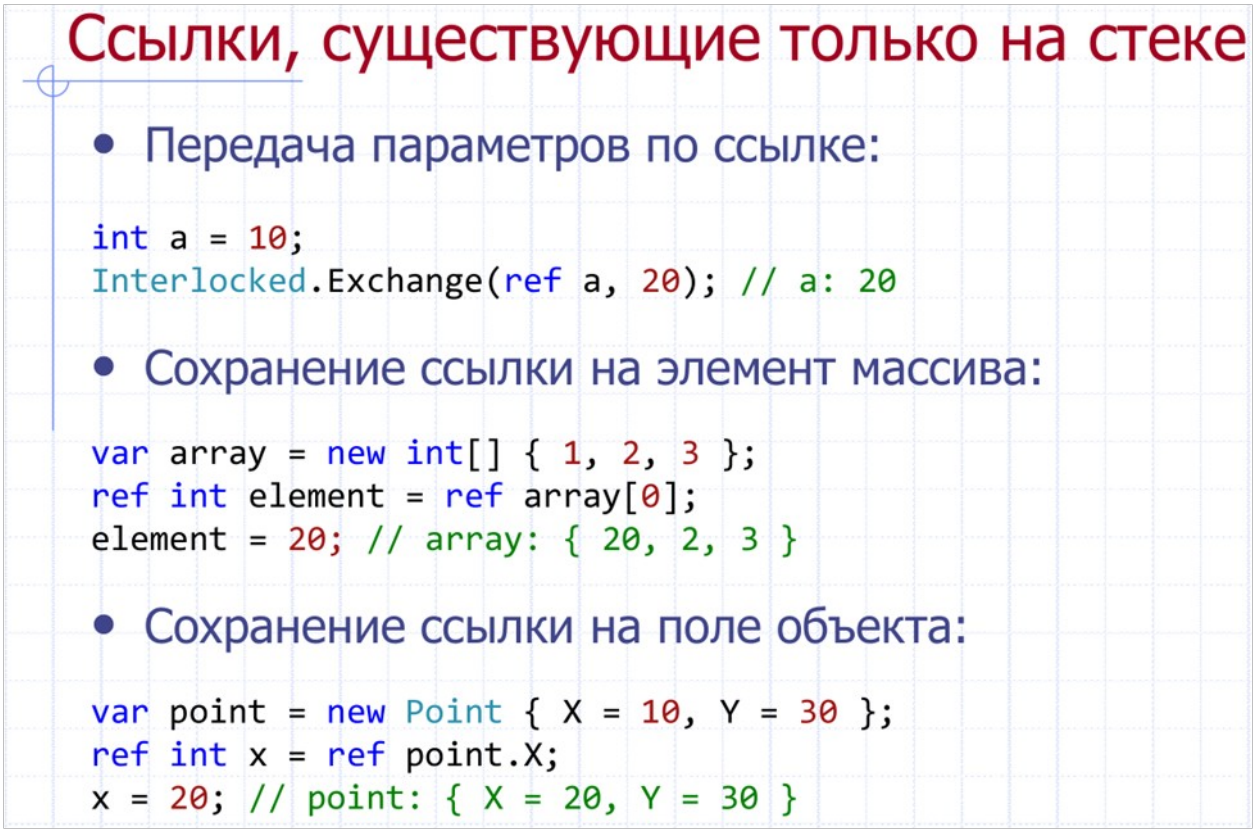
2) Когда требуется воспользоваться слабой ссылкой, вызывается свойство Target, которое возвращает сильную ссылку на объект, если он ещё существует, или нулевой указатель, если объект уже удален сборщиком мусора

3) Если возвращается обычная ссылка, она далее используется для доступа к объекту обычным образом. По завершении её использования, объект снова становится доступным для сборки мусора. То есть полученная ссылка была удалена, для повторного использования необходимо получить новую ссылку и проверить её на равенство нулевому указателю.

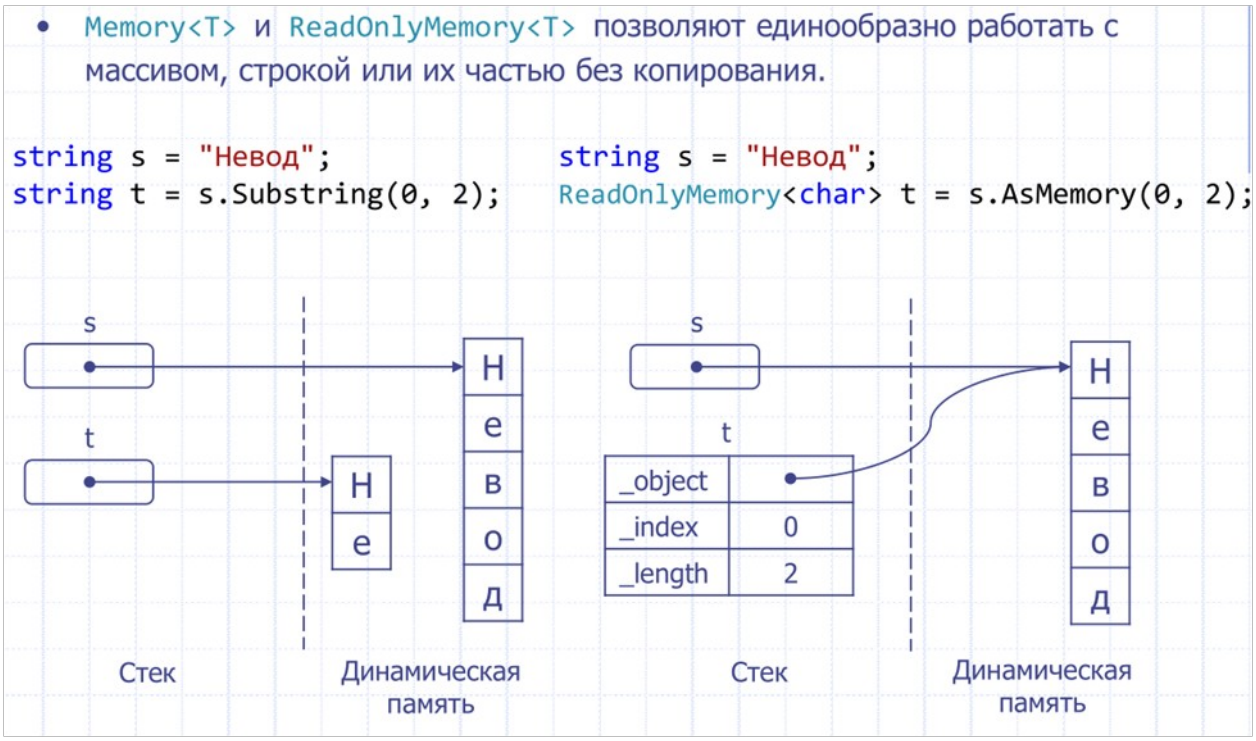
4) Если возвращается нулевая ссылка, это означает, что объект к моменту обращения уже был удален сборщиком мусора.

Ссылки, существующие только на стеке и избежание мусора с помощью типа данных Memory<T>.

Использование модификатора ref позволяет указать, что значение в метод передается по ссылке. Также можно создать ссылку, которая будет указывать на элемент массива или на поле объекта.



Тип данных Memory<T> представляет собой непрерывную область памяти на стеке, но при этом не является структурой с модификатором ref, что позволяет переменным этого типа данных располагаться как на стеке, так и в управляемой куче. Тип данных ReadOnlyMemory<T> позволяет работать с объектом как с неизменяемой областью памяти.



В первом случае метод Substring() создает новый объект типа String и возвращает его ссылку в управляемой кучи. Во втором случае переменная t будет указывать на ту же область памяти в управляемой куче, не прибегая к созданию нового объекта, благодаря использованию типа ReadOnlyMemory<T>.

Избежание мусора с помощью типа данных Span<T>. Оператор stackalloc. Объекты, существующие только на стеке (типы данных ref struct).

public readonly ref struct Span<T>

Span<T> - это структура с модификатором ref, которая может располагаться на стеке и не может располагаться в управляемой куче.

Из-за невозможности расположения типа Span<T> в управляемой куче возникают следующие ограничения:

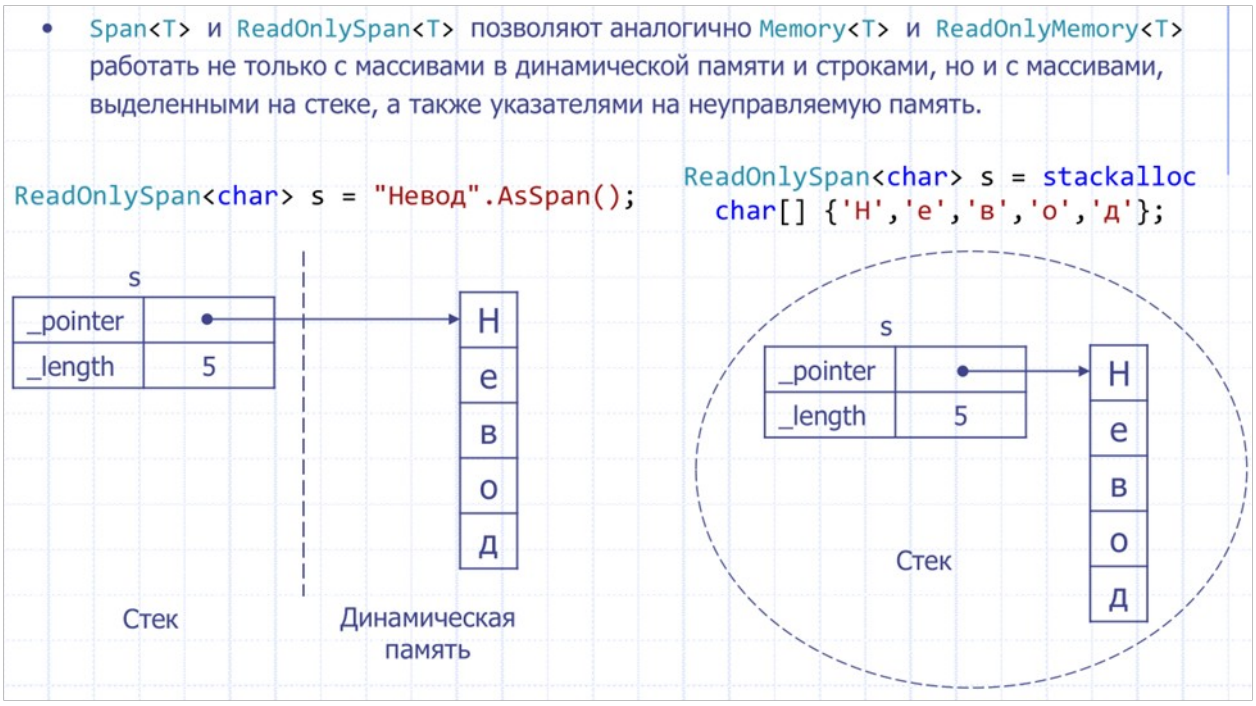
1) Экземпляр типа не может быть упакован

2) Не может быть присвоен переменной типа Object, dynamic, а также любому типу-интерфейсу.

3) Не может быть полем в ссылочном типа.

4) Не поддерживает методы Equals, GetHashCode. При их вызове происходит генерация исключения NotSupportedException.

Span<T> представляет собой непрерывную область памяти. Часто используется для хранения элементов массива или части массива. Может указывать как на память в управляемой куче, так и на память на стеке.



Управляемый массив вместо кучи можно разместить в стеке потока. Для этого используется инструкция stackalloc языка C#. Она позволяет создавать одномерные массивы элементов значимого типа с нулевой нижней границей. При этом значимый тип не должен содержать никаких полей ссылочного типа. Выделенная в стеке память автоматически освобождается после завершения метода. Благодаря использованию stackalloc повышается производительность, так как для выделенная память не требует сборки мусора.

```
  private static void StackallocDemo() {
    unsafe {
      const Int32 width = 20;
      Char* pc = stackalloc Char[width]; // В стеке выделяется
                                         // память под массив

      String s = "Jeffrey Richter";      // 15 символов

      for (Int32 index = 0; index < width; index++) {
        pc[width - index - 1] =
          (index < s.Length) ? s[index] : '.';
      }


    // Следующая инструкция выводит на экран ".....rethciR yerffeJ"
    Console.WriteLine(new String(pc, 0, width));
  }
}
```

Делегаты в платформе .NET и механизм их работы. Анонимные делегаты. События в языке C# и их отличия от делегатов.

Делегат — механизм поддержки функций обратного вызова.

internal delegate void Feedback(Int32 value);

Делегат Feedback задает сигнатуру метода обратного вызова. Данный делегат определяет метод, принимающий один параметр типа Int32 и возвращающий значение void.

private static void Counter(Int32 from, Int32 to, Feedback fb);

Данный метод принимает параметр fb, который является ссылкой на делегат Feedback.

Строка объявления делегата заставляет компилятор создать полное определение класса, которое выглядит примерно так:

```
internal class Feedback : System.MulticastDelegate {
  // Конструктор
  public Feedback(Object object, IntPtr method);

  // Метод, прототип которого задан в исходном тексте
  public virtual void Invoke(Int32 value);

  // Методы, обеспечивающие асинхронный обратный вызов
  public virtual IAsyncResult BeginInvoke(Int32 value,
    AsyncCallback callback, Object object);
  public virtual void EndInvoke(IAsyncResult result);
}
```

Любые типы делегатов — это потомки класса MulticastDelegate, от которого они наследуют поля, свойства и методы. Самые важные поля:

1) _target – если делегат является оболочкой статического метода, это поле содержит null. Если делегат является оболочкой экземплярного метода, поле ссылается на объект, с которым будет работать метод обратного вызова. Поле указывает на значение, которые нужно передать параметру this экземплярного метода.

2) _methodPtr – внутреннее целочисленное значение, используемое CLR для идентификации метода обратного вызова.

3) _invocationList – это поле обычно имеет значение null. Оно может ссылаться на массив делегатов при построении из них цепочки.

Любой делегат — лишь обертка для метода и обрабатываемого этим методом объекта.

Feedback fbStatic = new Feedback(Program.FeedbackToConsole);

Feedback fbInstance = new Feedback(new Program().FeedbackToFile);

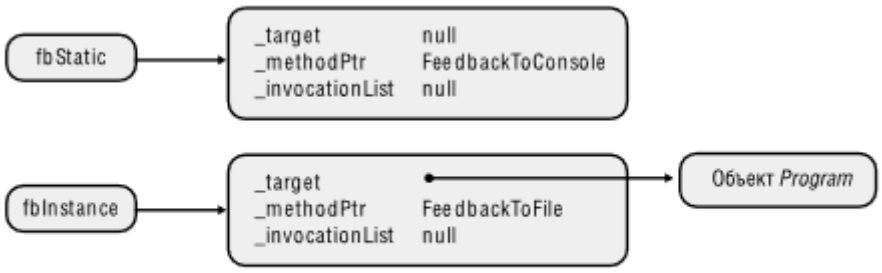


Рис. 17.2. Верхняя переменная ссылается на делегата статического метода, нижняя — на делегата экземплярного метода

Для вызова метода обратного вызова компилятор генерирует код:

fb.Invoke(val);

Цепочкой делегатов называется коллекция делегатов, дающая возможность вызывать все методы, представленные этими делегатами.

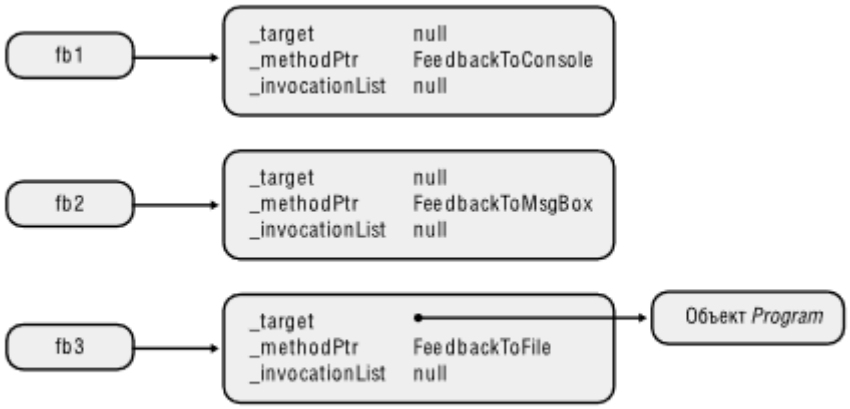


Рис. 17.3. Начальное состояние делегатов, на которые ссылаются переменные fb1, fb2 и fb3

Для добавления в цепочку делегатов используется открытый статический метод Combine класса Delegate:

Feedback fbChain = null;

fbChain = (Feedback) Delegate.Combine(fbChain, fb1);

При выполнении этой строки метод Combine видит, что мы пытаемся объединить значение null с переменной fb1. В итоге он возвращает значение в переменную fb1, а затем заставляет сослаться на делегата, на которого уже ссылается переменная fb1.

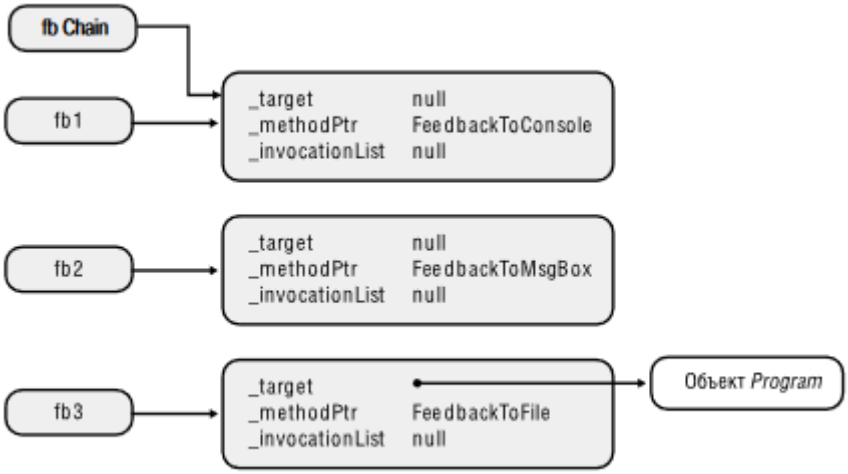


Рис. 17.4. Состояние делегатов после добавления в цепочку нового члена

Добавим в цепочку ещё одного делегата.

fbChain = (Feedback) Delegate.Combine(fbChain, fb2);

Метод Combine видит, что переменная fbChain уже ссылается на делегата, поэтому он создает нового делегата, который присваивается своим закрытым полям _target и _methodPtr некоторые значение, которые в данном случае не важны. Поле _invocationList инициализируется ссылкой на массив делегатов. Первому элементу массива присваивается ссылка на делегат, служащий оболочкой метода FeedbackToConsole, второму — FeedBackToMsgBox. Переменной fbChain присваивается ссылка на вновь созданный делегат

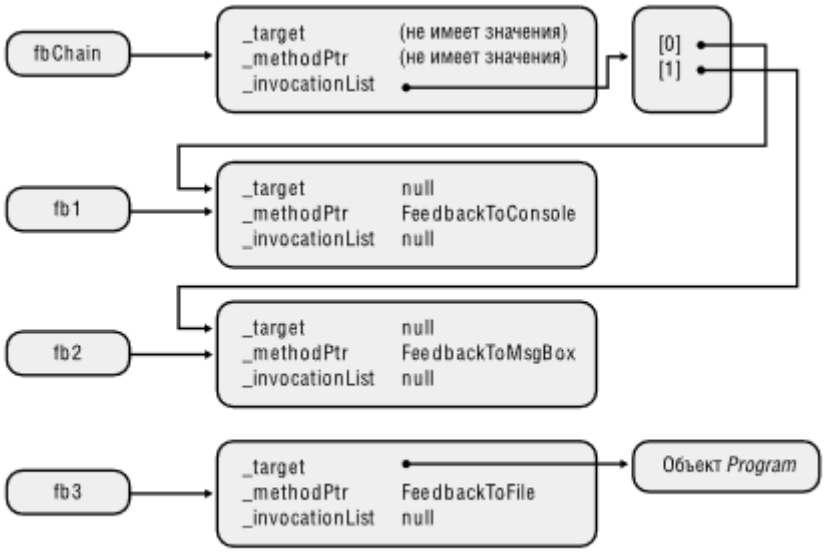


Рис. 17.5. Делегаты после вставки в цепочку второго члена

fbChain = (Feedback) Delegate.Combine(fbChain, fb3);

В этом случае все повторяется. Первому и второму элементам массива присваиваются ссылки на те же делегаты, на которые ссылался предыдущий делегат. Третий элемент массива становится ссылкой на делегата, служащего оболочкой метода FeedbackToFile. fbChain получает ссылку на вновь созданный делегат. Ранее созданный делегат подлежит обработке уборщика мусора.

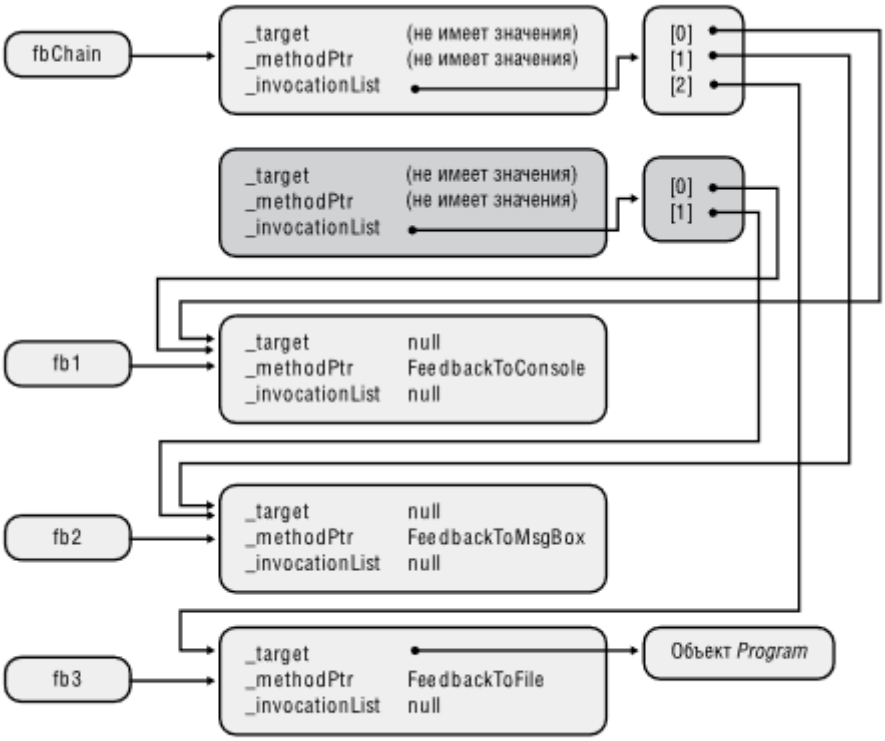


Рис. 17.6. Окончательный вид цепочки делегатов

При вызове Invoke для цепочки делегатов, происходит выполнение цикла, перебирающего все элементы массива, вызывая для них метод, оболочкой которого служит данный делегат.

Для удаления делегатов из цепочки применяется статический метод Remove объекта Delegate.

FbChain = (Feedback) Delegate.Remove(fbChain, new Feedback(FeedbackToMsgBox));

Метод Remove сканирует массив делегатов. Он ищет делегат, поля _target и _methodPtr которого совпадают с соответствующими полями второго аргумента. При обнаружении совпадения, если в массиве осталось более одного элемента, создается новый делегат — создается массив _invocationList, который инициализируется ссылкой на все элементы исходного массива, за исключением удаляемого, - после чего возвращается ссылка на нового делегата.

ThreadPool.QueueUserWorkItem(obj => Console.WriteLine(obj), 5);

Первый аргумент метода представляет собой фрагмент кода. Формально он называется лямбда выражением. Обнаружив лямбда выражение, компилятор автоматически определяет в классе новый закрытый метод. Этот метод называется анонимной фунцией.

Объявление события

public event EventHandler<NewMailEventArgs> NewMail;

Это компилируется в

1) Закрытое поле делегатами

private EventHandler<NewMailEventArgs> NewMail = null;

2) Открытый метод для регистрации на получение уведомлений о события

3) Открытый метод для отмены регистрации.

Для события можно переопределить методы добавления и удаления обработчиков

```
public event ListUpdateEvent Updated
{
    add { updated += value; }
    remove { updated -= value; }
}

list.Updated += HandleEvent;
```

Событию запрещено присваивание значения снаружи объекта. Запрещено вызывать обработчики события снаружи объекта.

Средства многопоточного программирования в платформе .NET. Самостоятельные потоки. Фоновые потоки.

Каждый поток состоит из нескольких частей:

1) Объект ядра потока.

2) Блок окружения потока. Это место в памяти, инициализированное в пользовательском режиме. Этот блок занимает одну страницу памяти. Он содержит заголовок цепочки обработки исключения.

3) Стек пользовательского режима

4) Стек режима ядра

5) Уведомления о создании и завершении потоков.

Для создания выделенного потока потребуется экземпляр класса System.Threading.Thread, для получения которого следует передать конструктору имя метода. Сигнатура этого метода должна совпадать с сигнатурой делегата ParametrizedThreadStart:

delegate void ParametrizedThreadStart(Object obj);

Создания объекта Thread является достаточно простой операцией, так как при этом физический поток в ОС не появляется. Для создания физического потока, призванного осуществить метод обратного вызова, следует воспользоваться методом Start класса Thread, передав в него объект, которые будет аргументом метода обратного вызова.

```
class Program
{
    public static void Main()
    {
        Thread t1 = new Thread(DoWork);
        t1.Start("X");
        Thread t2 = new Thread(DoWork, maxStackSize: 16 * 1024);
        t2.Start("o");
        Console.ReadLine();
    }

    public static void DoWork(object obj)
    {
        int threadId = Thread.CurrentThread.ManagedThreadId;
        Console.WriteLine("\nПоток с номером {0}", threadId);
        for (int i = 0; i < 1000; ++i)
        {
            Console.Write(" {0}", obj);
            Thread.Sleep(10);
        }
    }
}
```

В CLR все потоки делятся на активные и фоновые. При завершении активных потоков в процессе CLR принудительно завершает также все запущенные на этот момент фоновые

потоки. При этом завершение фоновых потоков происходит немедленно и без появления исключений.

Главный поток является основным. Если он завершается, то для завершения программы

```
class Program
{
    public static void Main()
    {
        Thread t1 = new Thread(DoWork) { IsBackground = true };
        t1.Start("X");
        Thread t2 = new Thread(DoWork) { IsBackground = true };
        t2.Start("o");
        Console.ReadLine();
    }

    public static void DoWork(object obj)
    {
        int threadId = Thread.CurrentThread.ManagedThreadId;
        Console.WriteLine("\nПоток с номером {0}", threadId);
        for (int i = 0; i < 1000; ++i)
        {
            Console.Write(" {0}", obj);
            Thread.Sleep(10);
        }
    }
}
```

необходимо, чтобы все остальные основные потоки тоже завершились.

Свойство IsBackground отвечает за то, является ли поток фоновым или нет.

Средства многопоточного программирования в платформе .NET. Пул потоков. Бесконечная блокировка из-за исчерпания пула потоков.

Создание и уничтожение потока занимает изрядное время. Кроме того, при наличии множества потоков впустую расходуется память и снижается производительность, ведь ОС приходится планировать исполнение потоков и выполнять переключение контекста. Среда CLR способна управлять собственным пулом потоков, то есть набором готовых потоков, доступных для использования приложениями.

При инициализации CLR пул потоков пуст. В его внутренней реализации поддерживается очередь запросов на выполнение операций. Для выполнения приложением асинхронной операции вызывается метод, размещающий запрос в очереди пула потоков. Код пула извлекает записи из очереди и распределяет их среди потоков из пула. Если пул пуст, создается новый поток. По завершении исполнения поток не уничтожается, а возвращается в пул и ожидает следующего запроса. Если приложение создает очередь запросов быстрее, чем поток из пула их обслуживает, создаются дополнительные потоки. Через некоторое время бездействия поток пробуждается и самоуничтожается, освобождая ресурсы.

Для добавления в очередь пула потоков асинхронных вычислительных операций обычно вызывают один из следующих методов класса ThreadPool:

static Boolean QueueUserWorkItem(WaitCallback callBack);

static Boolean QueueUserWorkItem(WaitCallback callBack, Object state);

delegate void WaitCallback(Object state);

```
public static void Main()
{
    ThreadPool.QueueUserWorkItem(DoWork, "o");
    ThreadPool.QueueUserWorkItem(DoWork, "X");
    Console.ReadLine();
}

public static void DoWork(object obj)
{
    int threadId = Thread.CurrentThread.ManagedThreadId;
    Console.WriteLine("\nРаботает поток с номером {0}", threadId);
    for (int i = 0; i < 1000; ++i)
    {
        Console.Write(" {0}", obj);
        Thread.Sleep(10);
    }
}
```

CLR позволяет указать максимально возможное количество потоков, создаваемых пулом. Задание верхнего предела может привести к зависанию или взаимной блокировке. Представим очередь из 1000 рабочих элементов, заблокированную сигнальным событием элемента под номером 1001. Если верхний предел для количества потоков равен 1000, этот новый поток исполнен не будет, а значит вся тысяча потоков навсегда окажется заблокированной. Класс System.Threading.ThreadPool предлагает несколько статических

методов для управления количеством потоков в пуле: GetMaxThreads, SetMaxThreads, GetMinThreads, SetMinThreads, GetAvaibleThreads.

```
static volatile int CompletedTaskNumber = 0;
static void Main(string[] args) {
    ThreadPool.GetMinThreads(out int workerThreads, out _);
    ThreadPool.SetMaxThreads(workerThreads, workerThreads);
    int n = workerThreads + 1; // если n = workerThreads, то нет блокировки
    ThreadPool.QueueUserWorkItem(Do, n);
    while (CompletedTaskNumber != n) Thread.Yield();
    Console.ReadLine();
}
static void Do(object value) {
    int current = (int) value;
    if (current > 1) {
        int next = current - 1;
        Console.WriteLine($"Задача {current} ожидает задачу {next}");
        ThreadPool.QueueUserWorkItem(Do, next);
        while (CompletedTaskNumber != next) Thread.Yield();
    }
    Console.WriteLine($"Задача {current} завершается");
    CompletedTaskNumber = current;
}
```

Проблема синхронизации потоков при работе с общими данными. Синхронизация с помощью мьютекса. Оператор lock.

Синхронизация представляет собой акт координации параллельно выполняемых действий с целью получения предсказуемых результатов. Синхронизация особенно важна, когда множество потоков получают доступ к одним и тем же данным. Проблема синхронизации заключается в общих данных, которые пытаются изменить несколько потоков, что приводит к получению непредсказуемых значений.

Мьютекс предоставляет взаимно исключающую блокировку. Он функционирует аналогично объекту Semaphore со значением счетчика 1.

```
public sealed class Mutex : WaitHandle {
  public Mutex();
  public void ReleaseMutex();
}
```

Объекты Mutex сохраняют информацию о том, какие потоки ими владеют. Для этого они запрашивают идентификатор потока. Если поток вызывает метод ReleaseMutex, объект Mutex сначала убеждается, что это именно владеющий им поток. Если это не так, состояние объекта Mutex не меняется, а метод ReleaseMutex генерирует исключение. Объекты Mutex управляют рекурсивным счетчиком, указывающим, сколько раз поток-владелец уже владел объектом. Если поток владеет мьютексом в настоящий момент и ожидает его ещё раз, рекурсивный счетчик увеличивается на единицу, и потоку разрешается продолжить выполнение. После того, как значение счетчика достигнет 0, владельцем мьютекса может стать другой поток.

```
internal class SomeClass : IDisposable {
  private readonly Mutex m_lock = new Mutex();

  public void Method1() {
    m_lock.WaitOne();
    // Делаем что-то...
    Method2(); // Метод Method2, рекурсивно получающий право на блокировку
    m_lock.ReleaseMutex();
  }

  public void Method2() {
    m_lock.WaitOne();
    // Делаем что-то...
    m_lock.ReleaseMutex();
  }

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

В C# появился упрощенный синтаксис в виде ключевого слова lock, позволяющее в одном и том же методе устанавливать блокировку, что-то делать, а затем снимать блокировку.

```
private void SomeMethod() {
  lock (this) {
    // Этот код имеет эксклюзивный доступ к данным...
  }
}
```

Приведенный фрагмент эквивалентен коду:

```
private void SomeMethod() {
  Boolean lockTaken = false;
  try {
    //
    Monitor.Enter(this, ref lockTaken);
    // Этот код имеет монопольный доступ к данным...
  }
  finally {
    if (lockTaken) Monitor.Exit(this);
  }
}
```

Переменная lockTaken типа Boolean решает проблему, когда поток вошел в блок try и был прерван до вызова метода Monitor.Enter. После этого вызывается блок finally, но его код не должен снимать блокировку. Если метод Monitor.Enter успешно получает блокировку, переменной lockTaken присваивается значение true. Блок finally по значению этой перемнной определяет, нужно ли вызывать метод Monitor.Exit.

Атомарные (Interlocked-) операции. Реализация мьютекса с помощью Interlocked.CompareExchange. Переменные с модификатором volatile.

Инструкция является атомарной, если она выполняется как единая, неделимая команда. Строгая атомарность препятствует любой попытке вытеснения. В C# просто чтение или присвоение значения полю в 32 бита или менее является атомарным. Операции с большими полями не атомарны, так как являются комбинацией более чем одной операции чтения/записи.

Если поток A читает 64-битное значение, в то время как поток B обновляет его, поток A может получить битовую комбинацию из старого и нового значений.

Одно из решений проблемы — обернуть неатомарные операции в блокировку. Блокировка фактически моделирует атомарность. Однако класс Interlocked предлагает более простое и быстрое решение для простых атомарных операций.

```
public static class Interlocked {
  // Возвращает (++location)

  public static Int32 Increment(ref Int32 location);

  // Возвращает (--location)
  public static Int32 Decrement(ref Int32 location);

  // Возвращает (location += value)
  // ПРИМЕЧАНИЕ. Значение может быть отрицательным,
  // что позволяет выполнить вычитание
  public static Int32 Add(ref Int32 location, Int32 value);

  // Int32 old = location; location = value; возвращает old;
  public static Int32 Exchange(ref Int32 location, Int32 value);

  // Int32 old = location1;
  // если (location1 == comparand) location = value;
  // возвращает old;
  public static Int32 CompareExchange(ref Int32 location,
    Int32 value, Int32 comparand);
  ...
}
```

Использование Interlocked более эффективно чем lock, так как при этом в принципе отсутствует блокировка — и соответствующие накладные расходы на временную приостановку потока.

Реализация мьютекса с помощью Interlocked.CompareExchange

```
public class Mutex
{
    private Thread thread;

    public void Lock()
    {
        Thread t = Thread.CurrentThread;
        while (Interlocked.CompareExchange(ref thread, t, null) != null)
            Thread.Yield();
        Thread.MemoryBarrier();
    }

    public void Unlock()
    {
        Thread t = Thread.CurrentThread;
        if (Interlocked.CompareExchange(ref thread, null, t) != t)
            throw new SynchronizationLockException();
        Thread.MemoryBarrier();
    }
}
```

```
class Unsafe
{
  static bool endIsNigh;
  static bool repented;

  static void Main()
  {
    // Запустить поток, ждущий изменения флага в цикле...
    new Thread(Wait).Start();
    Thread.Sleep(1000); // Дадим секунду на «прогрев»!
    repented = true;
    endIsNigh = true;
    Console.WriteLine("Понеслась...");
  }

  static void Wait()
  {
    while (!endIsNigh) // Крутимся в ожидании изменения значения endIsNigh
      ;

    Console.WriteLine("Готово, " + repented);
  }
}
```

Из-за возможных оптимизаций компилятора в данном коде могут произойти неприятные ситуации.

1) Компилятор может вынести проверку !endIsNigh один раз перед циклом, из-за чего цикл может начать выполняться бесконечно.

2) Метода Wait может вывести на консоль комбинацию Готово false, что происходит из-за того, что присвоение переменных в исходном коде может происходить не в том порядке, в котором вы записали.

Поля repented и endIsNigh могут кэшироваться в регистрах CPU для повышения производительности и записываться назад в память с некоторой задержкой. И порядок, в каком регистры записываются в память, необязательно совпадает с порядком обновления полей.

Кэширования и оптимизации компилятора обходятся, используя статические методы Thread.VolatileRead и Thread.VolatileWrite для чтения и записи полей. VolatileRead – это способ «читать последнее значение»; VolatileWrite означает «немедленно записать в память». Того же эффекта позволяет достичь объявлением полей с модификатором volatile.

Мониторы в платформе .NET. Ожидание выполнения условий с помощью методов Wait и Pulse.

Мониторы — одна из конструкций синхронизации потоков.

Класс Monitor является статическим, его методы принимают ссылки на любой объект из кучи. Управление полями эти методы осуществляют в блоке синхронизации заданного объекта. Блок синхронизации содержит поле для объекта ядра, идентификатор потока-владельца, счетчик рекурсии и счетчик ожидающих потоков.

```
public static class Monitor {
  public static void Enter(Object obj);
  public static void Exit(Object obj);

  // Можно также указать время блокирования (требуется редко):
  public static Boolean TryEnter(Object obj, Int32 millisecondsTimeout);

  // Аргумент lockTaken будет рассмотрен позднее
  public static void Enter(Object obj, ref Boolean lockTaken);
  public static void TryEnter(
    Object obj, Int32 millisecondsTimeout, ref Boolean lockTaken);
}
```

В C# появился упрощенный синтаксис в виде ключевого слова lock, позволяющее в одном и том же методе устанавливать блокировку, что-то делать, а затем снимать блокировку.

```
private void SomeMethod() {
  lock (this) {
    // Этот код имеет эксклюзивный доступ к данным...
  }
}
```

Приведенный фрагмент эквивалентен коду:

```
private void SomeMethod() {
  Boolean lockTaken = false;
  try {
    //
    Monitor.Enter(this, ref lockTaken);
    // Этот код имеет монопольный доступ к данным...
  }
  finally {
    if (lockTaken) Monitor.Exit(this);
  }
}
```

Переменная lockTaken типа Boolean решает проблему, когда поток вошел в блок try и был прерван до вызова метода Monitor.Enter. После этого вызывается блок finally, но его код не должен снимать блокировку. Если метод Monitor.Enter успешно получает блокировку, переменной lockTaken присваивается значение true. Блок finally по значению этой перемнной определяет, нужно ли вызывать метод Monitor.Exit.

Назначение Wait и Pulse – обеспечить простой сигнальный механизм; Wait блокирует, пока не получено уведомление от другого потока, Pulse реализует это уведомление.

Чтобы сигнализация сработала Wait должен выполняться перед Pulse. Если pulse выполнится первым, его сигнал будет потерен, и вызванный после него Wait должен будет ожидать следующего сигнала или остаться навсегда заблокированным.

Например, если **x** объявлен следующим образом:

```
class Test
{
  // Любой объект ссылочного типа может быть объектом синхронизации
  object x = new object();
}
```

то следующий код заблокирует поток на вызове **Monitor.Wait**:

```
lock (x)
  Monitor.Wait(x);
```

А этот код (если он выполнен позже в другом потоке) освободит блокированный поток:

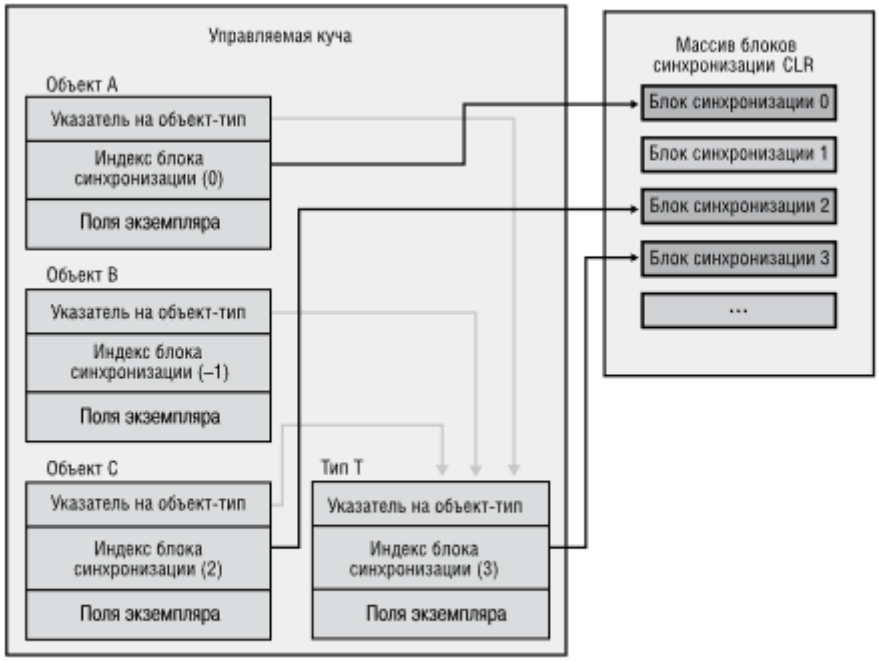
```
lock (x)
  Monitor.Pulse(x);
```

Для вызова Wait или Pulse необходимо определить объект синхронизации. Если два потока используют один и тот же объект, они способны посигналить друг другу. Объект синхронизации должен быть заблокирован перед вызовом Wait или Pulse.

Блоки синхронизации в объектах. Логическая реализация методов Wait и Pulse/PulseAll в классе Monitor.

С каждым объектом в куче может быть связана структура данных, называемая блоком синхронизации. Этот блок содержит поле для объекта ядра, идентификатора потока-владельца, счетчика рекурсии и счетчика ожидающих потоков. Во время инициализации CLR выделяется массив блоков синхронизации. При создании объекта в куче с ним связывается два дополнительных служебных поля. Первое — указатель на объект-тип — содержит адрес этого объекта в памяти. Второе поле содержит индекс блока синхронизации, то есть индекс в массиве таких блоков.

В момент конструирования объекта этому индексу присваивается значение -1, что означает отсутствие ссылок на блок синхронизации . При вызове метода Monitor.Enter CLR обнаруживает в массиве свободный блок синхронизации и присваивает ссылку на него объекту. Метод Exit проверяет наличие потоков, ожидающих блока синхронизации. Если таких потоков не обнаруживается, метод возвращает индексу значение -1, означающее, что блоки синхронизации свободны и могут быть связаны с какими-нибудь объектами.



ис. 30.1. Индекс блоков синхронизации объектов в куче (включая объекты-типы) может ссылаться на запись в массиве блоков синхронизации CLR

Чтобы выполнить переключение блокировки, Wait временно освобождает или отключает базовый lock на время ожидания, чтобы другой поток, который будет вызывать Pulse, тоже мог получить блокировку. Метод Wait можно представить в виде следующего псевдокода:

Monitor.Exit(x);

// Ожидание вызова pulse для x;

Monitor.Enter(x);

Следовательно Wait может заблокировать поток дважды: один раз при ожидании Pulse, и ещё раз — при восстановлении эксклюзивной блокировки. Pulse также не полностью разблокирует ожидающий поток: только когда сигнализирующий поток покидает конструкцию lock, ожидающий поток действительно может идти дальше.

Вызвать Wait на одном и том же объекте могут сразу несколько потоков — в этом случае за

объектом синхронизации образуется очередь ожидания, waiting queue. Каждый Pulse освобождает один поток из головы очереди ожидания, после чего он переходи к ready queue(очередь ожидания на lock) для переустановки блокировки.

Класс Monitor предоставляет также метод PulseAll, освобождающий всю очередь или пул потоков. Потоки стартуют на сразу, а в определенной последовательности, так как каждый Wait пытается переустановить одну и ту же блокировку. Так что PulseAll просто перемещает потоки из waiting queue в ready queue.

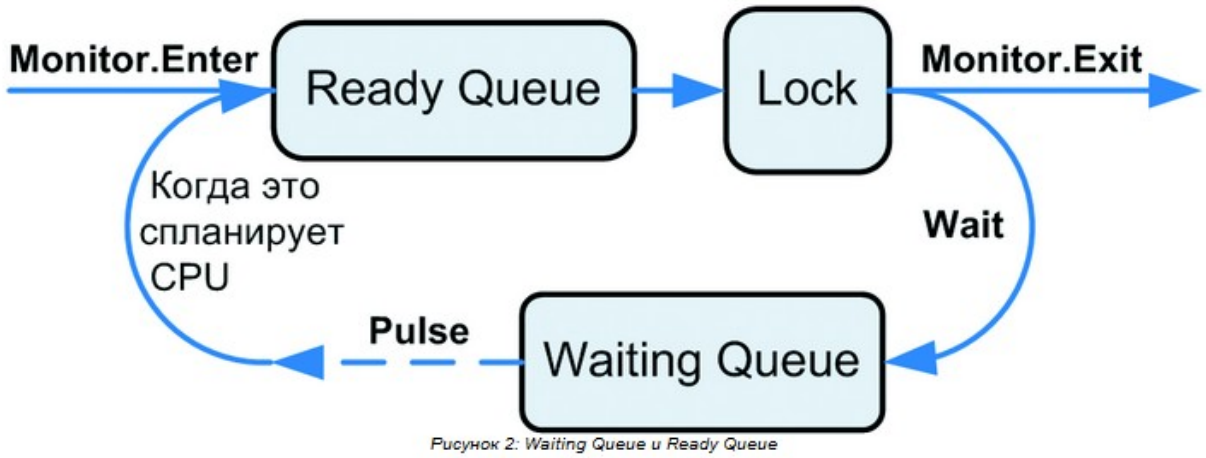


Рисунок 2: Waiting Queue и Ready Queue

http://rsdn.org/article/dotnet/CSThreading2.xml

Локальные данные потока (Thread-Local Storage – TLS) на основе атрибута ThreadStatic, типов данных ThreadLocal<T> и LocalDataStoreSlot.

Иногда данные должны хранится изолированно, гарантируя тем самым, что каждый поток имеет их отдельную копию. Решением является локальное хранилище потока.

Простейший подход к реализации локального хранилища предусматривает пометку статического поля с помощью атрибута [ThreadStatic]:

[ThreadStatic] static int _x;

После этого каждый поток будет видеть отдельную копию _x. Атрибут [ThreadStatic] не работает с полями экземпляра(он просто ничего не делает). Инициализация такого поля происходит только один раз, когда выполняется конструктор класса и, следовательно, влияет только на один поток.
Если необходимо работать с полями экземпляра или начать с нестандартного значения более подходящим вариантом является ThreadLocal<T>. Он представляет локальное хранилище потока как для статических полей, так и для полей экземпляра, и позволяет указывать стандартные значения. Для получения или установки значения, локального для потока, применяется свойство Value объекта _x.

static ThreadLocal<int> _x = new ThreadLocal<int>(() => 3);

Тип LocalDataStoreSlot представляет собой ячейка памяти для хранения локальных данных.
Недостатки:

1) Нет проверки типов во время компиляции

2) Необходимость приведения к нужному типу.

Достоинство:

1) Удобно использовать в случае, когда недостаточно информации для выделения статических полей во время компиляции.

Для работы с типом LocalDateStoreSlot используются следующие методы:
Thread.AllocateDataSlot() - создание слота, Thread.GetData(slot) – чтение из изолированного хранилища данных потока, Thread.SetData(slot, value) – запись в него.

Одна и та же ячейка может использоваться во всех потоках, но они по-прежнему будут получать отдельные значения.

Области (домены) приложений (AppDomain). Создание и использование объектов в разных доменах приложений.

В ходе инициализации CLR создает домен приложений, представляющий собой логический контейнер для набора сборок. Первый из созданных доменов называют основным, он уничтожается только при завершении Windows-процесса. Домены удобны благодаря нескольким свойствам:

1) Объекты, созданные одним доменом приложений, недоступны для кода других доменов. Код другого домена может получить доступ к объекту только используя семантику продвижения по ссылке или по значению.

2) Домены приложений можно выгружать. Можно заставить CLR выгрузить домен приложений со всеми содержащимися в нем в данный момент сборками.

3) Домены приложений можно индивидуально защищать. При создании можно назначить набор разрешений, определяющий права запущенных в нем сборок.

4) Домены приложений можно индивидуально настраивать.

У каждого домена есть собственная куча загрузчика, ведущая учет обращений к типам с момента создания домена.

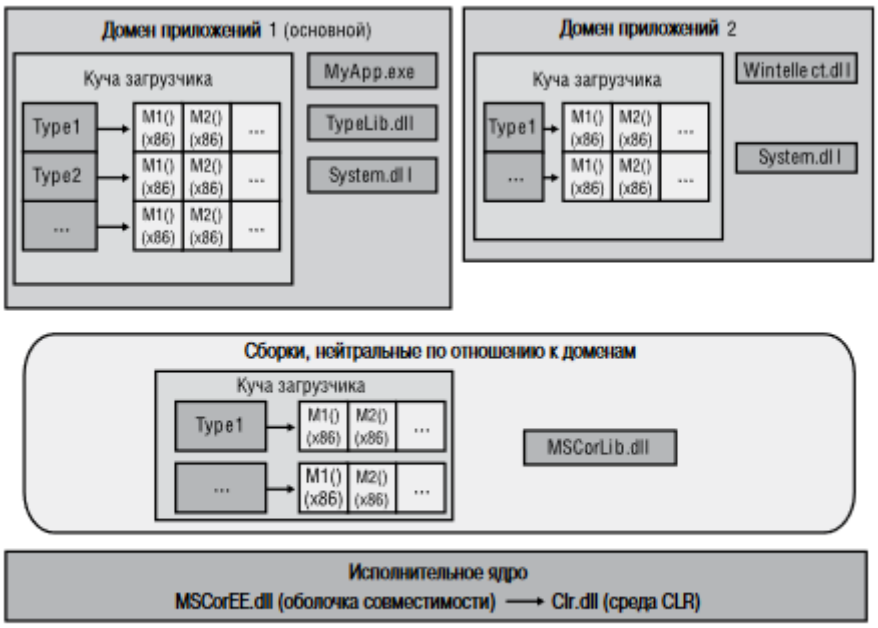


Рис. 22.1. Windows-процесс, являющийся хостом для CLR и двух доменов приложений

Некоторые сборки предназначены для совместного использования разными доменами. Например, MSCorLib.dll. Эта сборка автоматически загружается при инициализации CLR, и домены приложений совместно используют её типы. Сборки, загруженные без привязки к доменам, нельзя выгружать до завершения процесса.

Код, расположенный в одном домене приложений, способен взаимодействовать с типами и объектами другого домена.

```
// Определяем локальную переменную, ссылающуюся на домен
AppDomain ad2 = null;
```

1) Междоменное взаимодействие с продвижением по ссылке

```
// ПРИМЕР 1. Доступ к объектам другого домена приложений
// с продвижением по ссылке
Console.WriteLine("{0}Demo #1", Environment.NewLine);

// Создаем новый домен (с теми же параметрами защиты и конфигурирования)
ad2 = AppDomain.CreateDomain("AD #2", null, null);
MarshalByRefType mbrt = null;

// Загружаем нашу сборку в новый домен, конструируем объект
// и продвигаем его обратно в наш домен
// (в действительности мы получаем ссылку на представитель)
mbrt = (MarshalByRefType)
  ad2.CreateInstanceAndUnwrap(exeAssembly, "MarshalByRefType");

Console.WriteLine("Type={0}", mbrt.GetType()); // CLR неверно
                                               // определяет тип

// Убеждаемся, что получили ссылку на объект-представитель
Console.WriteLine(
   "Is proxy={0}", RemotingServices.IsTransparentProxy(mbrt));

 // Все выглядит так, как будто мы вызываем метод экземпляра
 // MarshalByRefType, но на самом деле мы вызываем метод типа
 // представителя. Именно представитель переносит поток в тот домен,
 // в котором находится объект, и вызывает метод для реального объекта
 mbrt.SomeMethod();

 // Выгружаем новый домен
 AppDomain.Unload(ad2);
 // mbrt ссылается на правильный объект-представитель;
 // объект-представитель ссылается на неправильный домен

 try {
   // Вызываем метод, определенный в типе представителя
   // Поскольку домен приложений неправильный, появляется исключение
   mbrt.SomeMethod();
   Console.WriteLine("Successful call.");
 }
 catch (AppDomainUnloadedException) {
   Console.WriteLine("Failed call.");
 }

// Экземпляры допускают продвижение по ссылке через границы доменов
public sealed class MarshalByRefType : MarshalByRefObject {
  public MarshalByRefType() {
  Console.WriteLine("{0} ctor running in {1}",
    this.GetType().ToString(), Thread.GetDomain().FriendlyName);
  }

  public void SomeMethod() {
    Console.WriteLine("Executing in " + Thread.GetDomain().FriendlyName);
  }

  public MarshalByValType MethodWithReturn() {
    Console.WriteLine("Executing in " + Thread.GetDomain().FriendlyName);
    MarshalByValType t = new MarshalByValType();
    return t;
  }
```

CreateDomain создает новый домен в процессе. Чтобы получить экземпляр объекта в новом домене, надо сначала загрузить туда сборку, а затем создать экземпляр определенного в этой сборке типа. Эти операции выполняет метод CreateInstanceAndUnwrap. Метод заставляет вызывающий поток перейти из текущего домена в новый. Теперь поток загружает указанную

сборку в новый домен. Далее поток вызывает конструктор MarshalByRefType без параметров и возвращается обратно в основной домен, чтобы метод CreateInstanceAndUnwrap могу вернуть ссылку на новый объект MarshalByRefType.

Непосредственно перед возвращением ссылки происходят ещё некоторые действия. Обнаружив, что метод CreateInstanceAndUnwrap выполняет продвижение объекта типа, производного от MarshalByRefObject, CLR выполняет продвижение по ссылке в другой домен.

Когда домену-источнику нужно передать ссылку на объект в целевой домен приложения или вернуть её обратно, CLR определяет в куче загрузчика этого домена тип представителя(proxy). Его поля указывают, который из доменов владеет реальным объектом и как найти этот объект в домене-владельце.

После определения этого типа в целевом домене метод CreateInstanceAndUnwrpa создает экземпляр типа представителя, инициализирует его поля таким образом, чтобы они указывали на домен-источник и реалиный объект и возвращает в целевой домен ссылку на объект представитель.

При вызове метода у объекта-представителя информация из его полей используется для направления вызывающего потока из основного домена в новый. Далее в домене ищется реальный объект, после чего вызывается метод.

2) Междоменное взаимодействие с продвижением по значению

```
// ПРИМЕР 2. Доступ к объектам другого домена
// с продвижением по значению
Console.WriteLine("{0}Demo #2", Environment.NewLine);

// Создаем новый домен (с такими же параметрами защиты
// и конфигурирования, как в текущем)
ad2 = AppDomain.CreateDomain("AD #2", null, null);

// Загружаем нашу сборку в новый домен, конструируем объект
// и продвигаем его обратно в наш домен
// (в действительности мы получаем ссылку на представитель)
mbrt = (MarshalByRefType)
  ad2.CreateInstanceAndUnwrap(exeAssembly, "MarshalByRefType");

// Метод возвращает КОПИЮ возвращенного объекта;
// продвижение объекта происходило по значению, а не по ссылке
MarshalByValType mbvt = mbrt.MethodWithReturn();

// Убеждаемся, что мы НЕ получили ссылку на объект-представитель
Console.WriteLine(
  "Is proxy={0}", RemotingServices.IsTransparentProxy(mbvt));

// Кажется, что мы вызываем метод экземпляра MarshalByRefType,
// и это на самом деле так
Console.WriteLine("Returned object created " + mbvt.ToString());

// Выгружаем новый домен
AppDomain.Unload(ad2);
// mbrt ссылается на действительный объект;
```

```
// Экземпляры допускают продвижение по значению через границы доменов
[Serializable]
public sealed class MarshalByValType : Object {
  private DateTime m_creationTime = DateTime.Now;
  // ПРИМЕЧАНИЕ: DateTime помечен атрибутом [Serializable]

  public MarshalByValType() {
    Console.WriteLine("{0} ctor running in {1}, Created on {2:D}",
      this.GetType().ToString(),
      Thread.GetDomain().FriendlyName,
      m_creationTime);
  }

  public override String ToString() {
    return m_creationTime.ToLongDateString();
  }
}
```

```
// выгрузка домена не имеет никакого эффекта

try {
  // Вызываем метод объекта; исключение не генерируется
  Console.WriteLine("Returned object created " + mbvt.ToString());
  Console.WriteLine("Successful call.");
}
catch (AppDomainUnloadedException) {
  Console.WriteLine("Failed call.");
}
```

Тип MarshalByValType не является производным от MarshalByRefObject, следовательно CLR не может определить тип представителя для создания его экземпляра.

Однако благодаря наличию атрибута [Serializable] можно выполнить продвижение по значению.

Когда исходному домену нужно передать ссылку на объект в целевой домен CLR сериализует экземплярные поля объекта в байтовый массив, который затем копируется. После этого байтовый массив десериализуется в целевом домене. Далее создается экземпляр типа и используются значения из байтового массива для инициализации полей объекта, чтобы они полностью совпадали со значениями исходного объекта. Иначе говоря создается точная копия.

На этом этапе объекты в исходном и целевом доменах существуют независимо друг от друга. При вызове методов переходов между доменами не происходит.

3) Междоменное взаимодействие без продвижения.

```
// Экземпляры не допускают продвижение между доменами
// [Serializable]
public sealed class NonMarshalableType : Object {
  public NonMarshalableType() {
    Console.WriteLine("Executing in " + Thread.GetDomain().FriendlyName);
  }
}
```

В случае, когда объект не является производным от MarshableByRefObject и не помечен атрибутом [Serializable], объект не может быть продвинут по ссылке или по значению. То есть способа передать объект через границы домена нет.

Асинхронные делегаты в платформе .NET.

Асинхронные делегаты позволяют вызывать методы, на которые эти делегаты указывают, в асинхронном режиме. Для этого используется пара методов BeginInvoke/EndInvoke.

Метод BeginInvoke принимает, как минимум, два параметра. Первый параметр представляет делегат SystemAsyncCallback. AsyncCallback указывает на метод, который будет выполняться в результате завершения работы асинхронного делегата. Второй параметр представляет объект, с помощью которого мы можем передать дополнительную информацию в метод завершения, указанный в предыдущем параметре. BeginInvoke запускает делегат в пуле потоков и возвращает объект с интерфейсом IasyncResult, который служит для ожидания окончания выполнения.

Объект IasyncResult потом передается в метод EndInvoke, который дожидается окончания выполнения делегата.

```
class Program
{
    static void Main()
    {
        Action action = Do;
        IAsyncResult result = action.BeginInvoke(WorkCompleted, null);
        Console.WriteLine("Ожидание окончания выполнения делегата.");
        action.EndInvoke(result);
    }

    static void Do()
    {
        Console.WriteLine("Выполняется метод Do.");
        Thread.Sleep(1000);
    }

    static void WorkCompleted(object state)
    {
        Console.WriteLine("Работа выполнена.");
    }
}
```

Недостатки:

1) Нет стандартных средств возврата результата работы

2) Нет стандартных средств возврата информации об ошибке

3) Нет стандартных средств ожидания завершения

4) Нет стандартных средств досрочного прерывания работы

5) Нет стандартных средств ограничения степени параллелизма

6) Нет планирования и распределения процессорного времени

7) Нет возможности асинхронного выполнения в контексте одного потока

Параллельные и асинхронные задачи на основе класса Task. Асинхронные методы (async) и средства ожидания их выполнения (await) в языке C#.

Вызвать метод QueueUserWorkItem класса ThreadPool для запуска асинхронных вычислительных операций очень просто. Однако у этого метода имеются недостатки: отсутствие встроенного механизма, позволяющего узнать о завершении операции и получить возвращаемое значение. Для обхода этих ограничений был введено понятие заданий(tasks).

ThreadPool.QueueUserWorkIten(ComputeBoundOp, 5); // Вызов QueueUserWorkItem

new Task(ComputeBoundOp, 5).Start(); // Аналог

Task.Run(() => ComputeBoundOp(5)); // Еще один аналог.

```
public class Task : IAsyncResult, IDisposable
{
    public Task(Action action);
    public Task(Action<object> action, object state,
        CancellationToken cancellationToken, TaskCreationOptions creationOptions);
    public TaskCreationOptions CreationOptions { get; }
    public AggregateException Exception { get; }
    public bool IsCompleted { get; }
    public bool IsCanceled { get; }
    public object AsyncState { get; }
    public bool IsCompletedSuccessfully { get; }
    public int Id { get; }
    public bool IsFaulted { get; }
    public TaskStatus Status { get; }
    public void Start();
    public void Start(TaskScheduler scheduler);
    public void Wait();
    public bool Wait(int millisecondsTimeout, CancellationToken cancellationToken);
    public void RunSynchronously();
    public TaskAwaiter GetAwaiter();
    public ConfiguredTaskAwaitable ConfigureAwait(bool continueOnCapturedContext);
    public Task ContinueWith(Action<Task> continuationAction);
}

public class Task<TResult> : Task
{
    public TResult Result { get; }
}
```


Конструктору можно также передать структуру CancellationToken, позволяющую отменить объект Task до его выполнения. Можно передавать флаги из перечисления TaskCreationOption, управляющие способами выполнения заданий.
Можно дождаться завершения задачи и после этого получить результат выполнения. Для этого создается объект Task<Tresult> и в качестве универсального аргумента передается тип результата, возвращаемого вычислительной операцией.

Task<Int32> t = new Task<Int32>(n => Sum((Int32)n), 1000000);

t.Start();

t.Wait();

Console.WriteLine(“The Sum is: “ + t.Result);

Если вычислительное задание генерирует необработанное исключение, оно поглощается и сохраняется в коллекции, а потоку пула разрешается вернуться в пул. Затем при вызове метода Wait или свойства Result эти члены вбросят исключение System.AggregateException.

Тип AggregateException инкапсулирует коллекцию исключений. Он содержит свойство InnerExceptions, возвращающее объект ReadOnlyCollection<Exception>, элементы которого

будут ссылаться на объект порождаемых исключений.

Можно ожидать завершения не только одного задания, но и массива объектов Task. Для этого в одноименном классе существуют методы WaitAny, ожидающий завершения одного из заданий и возвращающий индекс завершенного задания. WaitAll ожидает выполнения всех объектов Task в массиве. Возвращает true либо false.

Узнать о завершении задания можно, инициировав выполнение следующего задания:

```
// Создание объекта Task с отложенным запуском
Task<Int32> t = Task.Run(() => Sum(CancellationToken.None, 10000));

// Метод ContinueWith возвращает объект Task, но обычно
// он не используется
Task cwt = t.ContinueWith(task => Console.WriteLine(
  "The sum is: " + task.Result));
```

Как только задание, выполняющее метод Sum, завершится, оно инициирует выполнение следующего задания, которые выведет результат.

```
private static async Task<String> IssueClientRequestAsync(String serverName,
   String message)
{
      using (var pipe = new NamedPipeClientStream(serverName, "PipeName",
      PipeDirection.InOut, PipeOptions.Asynchronous | PipeOptions.WriteThrough)
      {
          pipe.Connect(); // Прежде чем задавать ReadMode, необходимо
          pipe.ReadMode = PipeTransmissionMode.Message; // вызвать Connect

          // Асинхронная отправка данных серверу
         Byte[] request = Encoding.UTF8.GetBytes(message);
          await pipe.WriteAsync(request, 0, request.Length);

         // Асинхронное чтение ответа сервера
         Byte[] response = new Byte[1000];
         Int32 bytesRead = await pipe.ReadAsync(response, 0, response.Length);
         return Encoding.UTF8.GetString(response, 0, bytesRead);
      }  // Закрытие канала
}
```

Когда метод помечается ключевым словом async, компилятор преобразует код метода в тип, реализующий конечный автомат. Это позволяет потоку выполнить часть кода в конечном автомате, а затем вернуть управление без выполнения всего метода до завершения. Также компилятор генерирует код, создающий объект Task в начале выполнения конечного автомата.

```
static async Task OutputPageTextAsync(string url)
{
    var httpClient = new HttpClient();
    string result = await httpClient.GetStringAsync(url);
    Console.WriteLine(result);
}
```

```
static Task OutputPageTextAsync_Compiled(string url)
{
    var _stateMachine = new _OutputPageTextAsync();
    _stateMachine._builder = AsyncTaskMethodBuilder.Create();
    _stateMachine.url = url;
    _stateMachine._state = -1;
    _stateMachine._builder.Start(ref _stateMachine);
    return _stateMachine._builder.Task;
}
```

При компиляции OutputPageTextAsync компилятор преобразует код метода в структуру конечного автомата с возможностью приостановки и продолжения выполнения. В скомпилированном методе создается экземпляр конечного автомата и его инициализация, создается builder, возвращающий объект Task, инициализируется местонахождение и копируются аргументы в поля конечного автомата. Дальше начинается выполнение конечного автомата и возвращается задание конечного автомата.

```
public sealed class _OutputPageTextAsync : IAsyncStateMachine
{
    public string url;
    private HttpClient httpClient;
    private string result;

    // Состояние конечного автомата
    public int _state;
    // Хранит объект Task, возвращаемый из метода OutputPageText
    public AsyncTaskMethodBuilder _builder;
    // Объект, который служит для ожидания подзадач
    private TaskAwaiter<string> _awaiter;

    public void MoveNext()
    {
        ...
    }

    public void SetStateMachine(IAsyncStateMachine stateMachine)
    {
    }
}
```


На каждый тип Awaiter создается по одному пол. В любой момент времени важно только одно из этих полей. В нем хранится ссылка на последний выполненный экземпляр await, который завершается асинхронно.

```
public void MoveNext()
{
  try
  {
    TaskAwaiter<string> awaiter;
    if (_state != 0)
    {
      // Код, запускаемый до await:
      httpClient = new HttpClient();
      // Код, соответствующий оператору await:
      awaiter = httpClient.GetStringAsync(url).GetAwaiter();
      if (!awaiter.IsCompleted)
      {
        _state = 0;
        _awaiter = awaiter;
        _OutputPageText stateMachine = this;
        // В очередь к задаче ставится вызов этого же метода (MoveNext):
        _builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine);
        return;
      }
    }
    else
    {
      awaiter = _awaiter;
      _awaiter = new TaskAwaiter<string>();
      _state = -1;
    }
    result = awaiter.GetResult();
    // Код, запускаемый после оператора await:
    Console.WriteLine(result);
  }
  catch (Exception exception)
  {
    _state = -2;
    httpClient = null;
    result = null;
    // Завершение задачи с ошибкой
    _builder.SetException(exception);
    return;
  }
  _state = -2;
  httpClient = null;
  result = null;
  // Успешное завершение задачи:
  _builder.SetResult();
}
```

До объекта awaiter располагается код, запускаемый до await. После него идет код, соответствующий оператору await.

Проверка if(!awaiter.IsCompleted) нужна, чтобы просмотреть, выполнился ли метод GetStringAsync. Если проверка успешна то состояние устанавливается в 0, сохраняется

объект ожидания и по завершению GetStringAsync вызывается метод снова метод MoveNext(), который уже сохраняет результат и выполняет код, идущий после оператора await. Если проверка не прошла, метод успел завершится, сохраняем результат и продолжаем выполнение.

Итераторы в языке C# и платформе .NET: интерфейсы IEnumerator и IEnumerable, оператор foreach, оператор yield return. Асинхронные итераторы. Интерфейсы IAsyncEnumerable и IasyncEnumerator.

Итератор — прием программирования для единообразной обработки элементов коллекций независимо от их разновидности и реализации(массив, связный список, словарь, множество).

Оператор foreach языка C# компилятором транслируется в цикл while, использующий итератор:

```
foreach (int a in numbers)
{
    Console.WriteLine(a);
}

IEnumerator it = numbers.GetEnumerator();
while (it.MoveNext())
{
    int a = (int)it.Current;
    Console.WriteLine(a);
}
```

Интерфейс IEnumerable имеет метод, возвращающий ссылку на другой интерфейс-перечислитель:

```
public interface IEnumerable
{
        IEnumerator GetEnumerator();
}
```

Интерфейс IEnumerator определяет функционал для перебора внутренних объектов в контейнере:

```
public interface IEnumerator
{
        bool MoveNext();
        object Current { get; }
        void Reset();
}
```

Метод MoveNext перемещает указатель на текущий элемент на следующую позицию в последовательности. Если последовательность ещё не закончилась, то возвращается true. Если же последовательность закончилась, возвращается false.

Свойство Current возвращает объект в последовательности, на который указывает указатель.

Метод Reset() сбрасывает указатель позиции в начальное положение.

Для реализации итератора можно либо реализовать интерфейс IEnumerable, либо тип может просто содержать метд GetEnumerable.

Ключевое слово yield используется для указания возвращаемого значения или значений. При подходе к оператору yield return текущее положение сохраняется. При следующем вызове итератора выполнение возобновляется с этого места.

Асинхронные итераторы — это асинхронные методы на основе оператора yield return.

Асинхронное итерирование с ожиданием выполняется с помощью оператора await foreach.

```
static async IAsyncEnumerable<long> GetFibonachiList(int count)
{
    for (int i = 0; i < count; i++)
        yield return await FibonachiAsync(i);
}

static async Task WriteFibonachiList(int count)
{
    var stream = GetFibonachiList(count);
    await foreach (long x in stream)
    {
        Console.WriteLine("Fibonachi: " + x);
    }
}
```

Каждый раз, когда асинхронный итератор будет возвращать очередное число, цикл будет его получать и выводить на консоль

Коллекции со встроенными средствами синхронизации доступа. Использование класса ParallelQuery.

В пространстве имен System.Collections.Concurrent находится набор коллекций, полностью безопасных в отношении потоков.

Параллельная коллекция	Непараллельный эквивалент
ConcurrentStack<T>	Stack<T>
ConcurrentQueue<T>	Queue<T>
ConcurrentBag<T>	(отсутствует)
ConcurrentDictionary<TKey,TValue>	Dictionary<TKey,TValue>

Параллельные коллекции оптимизированы для сценариев с высокой степенью параллелизма. Тем не менее, они также могут быть полезны в ситуациях, когда требуется коллекция, безопасная к потокам.

Параллельные коллекции также отличаются от традиционных коллекций тем, что они открывают доступ к специальным методам, которые предназначены для выполнения атомарных операций, типа проверить и действовать, подобных TryPop.

Если производится перечисление параллельной коллекции, в то время как другой поток её модифицирует, то никаких исключений не возникает — взамен вы получите смесь старого и нового содержимого.

С помощью встроенного языка запросов(LINQ) можно легко фильтровать и сортировать элементы, возвращать спроецированные наборы элементов и делать многое другое. При работе с объектами все элементы в наборе данных последовательно обрабатываются одним потоком — это называется последовательным запросом(sequential query). Повысить производительность можно при помощи языка параллельных запросов(Parallel LINQ), позволяющего последовательный запрос превратить в параллельный(ParallelQuery).

Вся функциональность Parallel LINQ реализована в статическом классе System.Linq.ParallelEnumerable. В частности этот класс содержит параллельные версии всех стандартных LINQ-операторов, таких как Where, Select, SelectMany, GroupBy, Join, OrderBy, Skip, Take и т. п. Все эти методы являются методами расширения типа System.Linq.ParallelQuery<T>. Для вызова параллельных версий следует преобразовать последовательный запрос в параллельный, воспользовавшись методом AsParallel.

```
int[] numbers = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

List<int> result = numbers
    .AsParallel()
    .WithDegreeOfParallelism(4)
    .Where(n => n % 2 == 0)
    .Select(Square).ToList();

result.ForEach(x => Console.WriteLine(x));
```

Контекст синхронизации и его использование для доступа к визуальным компонентам из второстепенных потоков.

В программах с пользовательским интерфейсом только один поток создает элементы пользовательского интерфейса и выполняет цикл обработки сообщений окна. Для выполнения длительных действий без остановки обработки сообщений используются дополнительные потоки. Обращаться к элементам пользовательского интерфейса можно только из потока, который их создал. Поэтому отображение на экране результатов длительных действий необходимо выполнять только в контексте главного потока.

Контекст синхронизации позволяет выполнять действия в контексте главного потока. С помощью метода Send в очередь сообщений окна ставится специальное сообщение со ссылкой на заданный делегат. При обработке такого сообщения главным потоком вызывается делегат, содержащийся в параметрах сообщения.

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void AddItemViaSynchronizationContext(object state)
    {
        var context = (SynchronizationContext)state;
        context.Send((x) =>
        {
            ListBox1.Items.Add("text");
        }, null);
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        ThreadPool.QueueUserWorkItem(AddItemViaSynchronizationContext,
            SynchronizationContext.Current);
    }
}
```

В данном случае текст на компонент ListBox будет успешно добавляться. Если взаимодействовать с компонентом напрямую, то в результате будет получено исключение.

Взаимодействие с неуправляемым кодом. Атрибуты DllImport, StructLayout, FieldOffset.

Среда CLR спроектирована так, чтобы приложения могли состоять как из управляемых, так и из неуправляемых компонентов.

CLR поддерживает три сценария взаимодействия:

1) Управляемый код может вызывать неуправляемые функции из DLL с использованием механизма Platform Invoke. Когда неуправляемый код вызывает неуправляемую функцию, то он сначала определяет DLL-библиотеку, содержащую функцию. Затем DLL библиотека загружается в память и в ней находится адрес функции и её аргументы передаются в стек. Затем управление передается неуправляемой функции.

2) Управляемый код может использовать готовые компоненты COM.

3) Неуправляемый код может использовать управляемый тип.

[DllImport(“Kernel32”, CharSet = CharSet.Auto, SetLasError = true)]

public static extern Boolean GetVersionEx([In, Out] OSVERSIONINFO ver);

Атрибут представляет сведения, необходимые для вызова функции, экспортируемой из неуправляемой библиотеки DLL. В качестве минимального требования необходимо указать имя DLL библиотеки.

Для повышения производительности CLR дано право устанавливать порядок размещения полей типа. Например, выстроить поля таким образом, чтобы ссылки на объекты окажутся в одной группе, а поля данных и свойства — в другой.

Для того, чтобы сообщить CLR способ управления полями, в описании класса или структуры указывается атрибут StructLayoutAttrubute. Чтобы порядок устанавливался CLR нужно передать конструктору параметр LayoutKind.Auto, чтобы сохранить установленный программистом порядок — параметр LayoutKing.Sequential, а параметр LayoutKing.Explicit позволяет разместить поля в память, явно задав смещения.

Передов в конструктор атрибута параметр LayoutKind.Explicit можно явно задать смещение для всех полей. Ко всем полям можно применить атрибут FieldOffsetAttribute, путем передачи конструктору этого атрибута значения типа Int32, определяющего смещение(в байтах) первого байта поля от начала экземпляра.

Динамические объекты, тип данных dynamic и его использование в программе. Реализация своих классов динамических объектов.

Для того, чтобы облегчить разработку с коммуникацией с другими компонентами(написанных на других языках) компилятор C# предлагает помечать такие типы как динамические(dynamic).

Когда код вызывает член класса при помощи динамического выражения, компилятор создает специальный IL-код, который описывает желаемую операцию. Этот код называется полезной нагрузкой. Во время выполнения программы он определяет существующую операцию для выполнения на основе действительного типа объекта, на которые ссылается динамическое выражение.

```
internal static class DynamicDemo {
   public static void Main() {
      dynamic value;
      for (Int32 demo = 0; demo < 2; demo++) {
         value = (demo == 0) ? (dynamic) 5 : (dynamic) "A";
         value = value + value;
         M(value);
      }
   }

   private static void M(Int32 n)  { Console.WriteLine("M(Int32): " + n); }
   private static void M(String s) { Console.WriteLine("M(String): " + s); }
}
```

Компилятор генерирует код полезной нагрузки, который проверяет действительный тип переменной value во время выполнения и определяет, что должен делать оператор +.

Недостатки типа dynamic заключаются в низкой скорости работы, связанной с проверкой типа, и невозможность перегрузки процедур и функций по типу данных — только по количеству операндов.

Класс DynamicObject позволяет задавать динамические объекты. Для использования DynamicObject надо создать свой класс, унаследовав его от DynamicObject и реализовав его методы. Каждый из его методов имеет одну и ту же модель определения: все они возвращают логическое значение, показывающее, удачно ли прошла операция. В качестве первого параметра все они принимают объект связывателя или binder.

```
private void DoWriteLine(object arg)
{
    Console.WriteLine(arg);
}

public void WriteLine(object arg)
{
    Console.WriteLine("Вызван WriteLine");
}

public override bool TryInvokeMember(InvokeMemberBinder binder,
    object[] args, out object result)
{
    bool memberFound = false;
    result = null;
    if (binder.Name == "WriteLine")
    {
        Console.WriteLine("Вызван TryInvokeMember");
        DoWriteLine(args[0]);
        memberFound = true;
    }
    return memberFound;
}
```

Динамическая генерация кода на основе деревьев выражений. Преобразование лямбда-выражения на языке C# в дерево выражений. Программное создание деревьев выражений.

Дерево выражения — это дерево, кодирующее синтаксис выражения на высокоуровневом языке программирования. Для кодирования используется иерархия классов, производных от класса Expression в пространстве имен System.Linq.Expressions. Корневым узлом дерева выражения является лямбда-выражение, которое можно скомпилировать в подпрограмму с параметрами и результатом. Компилятор C# поддерживает конструирование деревьев выражений из фрагментов программного кода.

Дерево выражения можно преобразовать в делегат, вызвав метод Compile.

Неявное преобразование из лямбда-выражения в класс Expression<Tdelegate> заставляет компилятор C# генерировать код, который строит дерево выражения.

Дерево выражения — это миниатюрная DOM-модель. Каждый узел в дереве представлен типом из пространства имен System.Linq.Expressions.

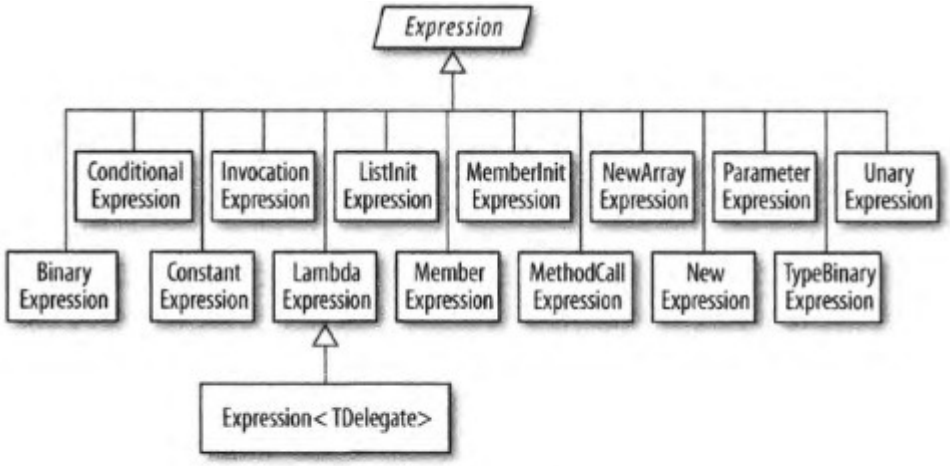


***Рис. 8.10.** Типы выражений*

Чтобы создавать деревья выражений не нужно создавать экземпляры типов узлов напрямую, взамен следует вызывать статические методы, предлагаемые классом Expression.

```
Expression<Func<string, bool>> f = s => s.Length < 5;
```

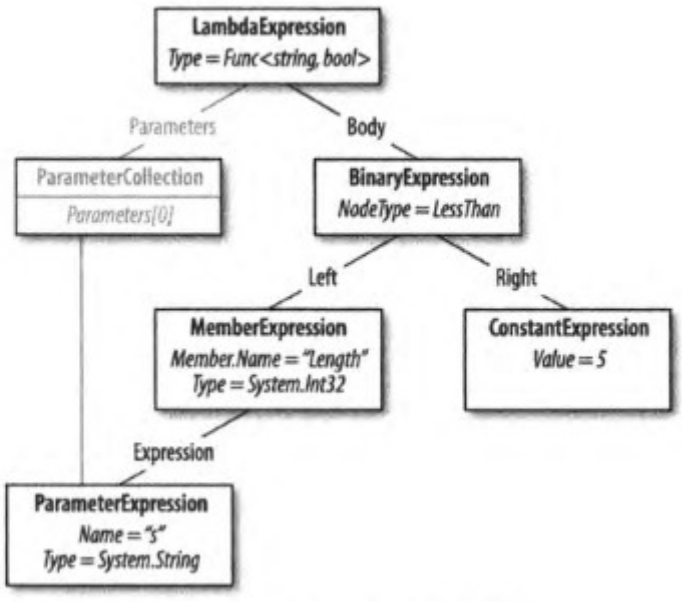


***Рис. 8.11.** Дерево выражения*

Принцип построения состоит в том, чтобы начать с нижней части дерева и работать, двигаясь вверх.

Зададим параметр лямбда-выражения:

```
ParameterExpression p = Expression.Parameter (typeof (string), "s");
```

На следующем шаге построим экземпляра MemberExpression и ConstantExpression.

```
MemberExpression stringLength = Expression.Property (p, "Length");
ConstantExpression five = Expression.Constant (5);
```

Далее создается сравнение LessThan:

```
BinaryExpression comparison = Expression.LessThan (stringLength, five);
```

На финальном шаге конструируется лямбда-выражение, которое связывает свойство Body выражения с коллекцией параметров:

```
Expression<Func<string, bool>> lambda
  = Expression.Lambda<Func<string, bool>> (comparison, p);
```

Листинг 9.7. Компиляция и выполнение дерева выражения

```
Expression firstArg = Expression.Constant(2);
Expression secondArg = Expression.Constant(3);
Expression add = Expression.Add(firstArg, secondArg);

Func<int> compiled = Expression.Lambda<Func<int>>(add).Compile();
Console.WriteLine(compiled());
```