6 Использование среды .NET Framework

В предыдущей главе рассматривались общие приемы и трудности программирования на .NET, в особенности те, которые имеют отношение к специфике языка. В этой главе речь пойдет о том, на что стоит обратить внимание при использовании обширной библиотеки кода, которая поставляется со средой .NET. Рассмотреть все многообразие подсистем и классов, являющихся частью .NET Framework, не представляется возможным, но целью этой главы является обеспечение вас инструментарием, необходимым для исследования приемов повышения производительности, и предоставление сведений о самых распространенных шаблонах, применения которых следует избегать.

Среда .NET Framework создавалась с прицелом на самую широкую аудиторию (фактически под всех разработчиков из всех областей) и задумывалась как универсальная среда, предоставляющая стабильный, правильный, надежный код, который способен справиться с множеством ситуаций. В ней как таковой не делается упор на исключительную производительность, и во внутренних циклах вашего кода найдется немало моментов, требующих доработки. Среда .NET должна работать для всех и везде, выстраивая предположения о вызывающем коде. Зачастую особое значение придается правильности и надежности, а не скорости и эффективности. Но в своем коде нередко можно добиться более впечатляющих успехов, осмыслив собственные ограничения и допущения и соответствующим образом адаптировав код. Это утверждение не дает вам права на переписывание класса `string` в новом проекте, но осведомляет вас об ограничениях среды в сочетании с критическими областями производительности вашего собственного кода.

Чтобы обойти недостатки среды .NET Framework или любой библиотеки стороннего производителя, может понадобиться проявить смекалку. Вот некоторые из возможных подходов.

- Воспользуйтесь альтернативным API с меньшими издержками.
- Переконструируйте свое приложение, чтобы реже вызывать API.
- Заново реализуйте некоторые API, добившись от них более высокой производительности.
- Для выполнения тех же задач перейдите к взаимодействию с API конкретной системы, предположив, что издержки на маршализацию будут ниже.

Разберитесь с каждым вызываемым API

Ведущий принцип этой главы: **необходимо понимать код, выполняющийся при каждом вызове API**.

Говорить о контроле производительности равнозначно тому, чтобы утверждать, что вы знаете код, который выполняется на каждом критическом пути программы. Непонятной библиотеки сторонних разработчиков во внутреннем цикле вашей программы быть не должно — это будет означать потерю контроля.

Доступ к исходному коду каждого вызываемого метода у вас будет не всегда (хотя всегда будет доступ к коду на уровне ассемблера!), но обычно все Windows API снабжаются качественной документацией. В среде .NET можно воспользоваться одним из многочисленных инструментов просмотра IL-кода, позволяющих увидеть, что делает эта среда (такая простота изучения не распространяется на саму среду CLR, которая, несмотря на свою доступность в качестве части ядра .NET Core, написана в основном на компактном, изобилующем макросами машинном коде).

Нужно привыкать к исследованию кода среды на предмет обнаружения чего-либо вам незнакомого. Чем более производительность важна для вас, тем больше вопросов вы должны задавать по поводу реализации сторонних API. Не забывайте, что ваша привередливость должна быть прямо пропорциональна требующейся скорости работы приложения.

Далее в главе рассматриваются несколько общих областей, к которым следует отнестись внимательно, а также некоторые конкретные общие классы, используемые каждой программой.

Множество API для решения одних и тех же задач

Временами встречаются ситуации, в которых можно выбирать, какой из множества API для решения одних и тех же задач использовать. Наглядным примером может послужить разбор XML. Конечно, разбирать XML — это никогда не быстрая работа, но, в зависимости от вашего сценария, некоторые из имеющихся вариантов решения этой задачи могут вам подойти лучше других. В среде .NET есть по крайней мере девять различных средств разбора XML:

- `XmlTextReader`;
- `XmlValidatingReader`;
- `XDocument`;
- `XmlDocument`;
- `XPathNavigator`;
- `XPathDocument`;
- `LINQ-to-XML`;
- `DataContractSerializer`;
- `XmlSerializer`.

Какой из них взять, зависит от таких факторов, как простота использования, продуктивность, целесообразность применения для данной задачи и производительность. Парсер `XmlTextReader` работает очень быстро, но он однонаправленный и не выполняет проверку. `XmlDocument` очень удобен, поскольку обладает полностью загруженной моделью объекта, но относится к самым медленным.

Этот подход приемлем как к разбору XML, так и к другим ситуациям с выбором API: не все варианты будут равноценны и рациональны с точки зрения достижения высокой производительности. Какие-то будут работать быстрее, но потреблять больше памяти. Какие-то обойдутся весьма скромным объемом памяти, но не позволят выполнять определенные операции. Придется определить набор нужных вам качеств и измерить производительность, чтобы выявить тот API, который обеспечивает разумный баланс функциональности и производительности. Необходимо создать прототипы для всех вариантов и провести их профилирование путем запуска на тестовых данных.

Коллекции

В .NET предоставляются свыше 20 встроенных типов коллекций, включая обобщенные версии многих популярных структур данных и версии, предназначенные для параллельной работы. Многим программам понадобится только лишь использование комбинации из существующих коллекций, а потребность в создании своих собственных будет возникать крайне редко.

Выбор коллекций зависит от множества факторов, включая семантическое значение предоставляемого API (помещение и извлечение, постановка в очередь и удаление из нее, добавление и удаление и т. д.), лежащий в основе механизм хранения и локальность кэша, скорость проведения различных операций с коллекцией, таких как `Add` и `Remove`, и необходимость синхронизации доступа к коллекции (или ее отсутствие). Все эти факторы могут существенно повлиять на производительность вашей программы.

Какие коллекции лучше не использовать

Некоторые коллекции все еще присутствуют в среде .NET Framework, но только из соображений обратной совместимости. Их никогда не следует использовать в новом коде. К их числу относятся:

- `ArrayList`;
- `Hashtable`;
- `Queue`;
- `SortedList`;
- `Stack`;
- `ListDictionary`;
- `HybridDictionary`.

Причинами, по которым нужно избегать их применения, являются преобразования типов и упаковка. В этих коллекциях хранятся ссылки на экземпляры `Object`, поэтому вам обязательно потребуется приведение к фактическому типу объекта.

Еще более вредна упаковка. Предположим, что нужно воспользоваться коллекцией `ArrayList`, состоящей из значимых типов Int32. Каждое значение будет в индивидуальном порядке упаковано и сохранено в куче. Вместо перебора последовательного массива памяти для доступа к каждому целочисленному значению каждая ссылка в массиве потребует разыменования указателя, обращения к куче (возможно, подрывая локальность), а затем операции распаковки для получения внутреннего значения. Это ужасно. Вместо этого лучше воспользоваться массивом, не меняющим свой размер, или одним из классов обобщенной коллекции.

В ранних версиях .NET присутствовало несколько коллекций, ориентированных на строковые значения, которые теперь в силу эффективности обобщенных коллекций устарели. В их числе `StringCollection`, `StringDictionary`, `NameValueCollection` и `OrderedDictionary`. Их использование не обязательно приведет к возникновению проблем с производительностью как таковой, но нет никакой необходимости даже брать их в расчет, пока не придется воспользоваться существующим API, требующим их применения.

Массивы

Самой простой и, наверное, наиболее часто используемой коллекцией является старый добрый массив `Array`. Массивы являются идеальной коллекцией в силу своей компактности, задействования одного последовательного блока памяти, улучшающего локальность кэша процессора при обращении к нескольким элементам (при условии использования значимых типов, ведь ссылочные типы будут по-прежнему приводить к переходам к другим местам кучи).

Обращение к ним занимает константное время, а их копирование выполняется быстро. Но изменение их размера будет означать выделение памяти под новый массив и копирование старых значений в новый объект. В качестве надстройки над массивами создаются многие более сложные структуры данных.

Общее требование — возможность передачи сегмента из состава массива. Одним из способов является копирование требуемых значений в новый массив нужного размера, но это влечет за собой дополнительные расходы ресурсов центрального процессора и памяти. Лучше заиметь структуру, которая просто ссылается на соответствующую часть массива. И хорошо, что в среде .NET уже есть такие структуры — `ArraySegment<T>` и `Span<T>`:

```
byte[] fileContents = File.ReadAllBytes("foo.bin");
ArraySegment <byte> header = new ArraySegment <byte >(fileContents,
                                                      1,
                                                      24);
ProcessHeader(header);
```

Многие API .NET, способные работать с массивами, зачастую будут иметь версию, непосредственно принимающую либо `ArraySegment<T>`, либо ссылку на

массив, смещение и количество элементов. При создании собственных API, работающих с массивами, разумно будет предусмотреть разработку версии, совместимой с `ArraySegment<T>`.

Похожими свойствами обладает и структура `Span<T>`, но она может представлять подсегменты, собранные из нескольких типов непрерывной памяти (она подробно рассмотрена в главе 2).

Сравнение ступенчатых и многомерных массивов

В среде .NET есть два способа выделения памяти под многомерные массивы.

- Многомерные массивы — один объект с несколькими индексами:

```
const int Size = 50;

private int[,,] multiArray;

void Init()
{
    this.multiArray = new int[Size, Size, Size];
}
```

- Ступенчатые массивы — массивы массивов (то есть множество объектов), у каждого из которых один индекс:

```
private const int Size = 50;

private int[][][] jaggedArray;

public void GlobalSetup()
{
    this.jaggedArray = new int[Size][][];
    for (int i = 0; i < Size; i++)
    {
        this.jaggedArray[i] = new int[Size][];
        for (int j = 0; j < Size; j++)
        {
            this.jaggedArray[i][j] = new int[Size];
        }
    }
}
```

Выглядят они в целом равнозначно, но устроено все по-разному. Показанный ранее код инициализации не дает реальной картины этого различия, поэтому рассмотрим код, выполняющий обход всех значений и изменяющий каждую запись.

```
public void Negate_JaggedArray()
{
    for (int i = 0; i < Size; i++)
    {
        for (int j = 0; j < Size; j++)
```

```
        {
            for (int k = 0; k < Size; k++)
            {
                this.jaggedArray[i][j][k] *= -1;
            }
        }
    }
}

public void Negate_MultiArray()
{
    for (int i = 0; i < Size; i++)
    {
        for (int j = 0; j < Size; j++)
        {
            for (int k = 0; k < Size; k++)
            {
                this.multiArray[i, j, k] *= -1;
            }
        }
    }
}
```

Код выглядит очень похожим, но посмотрим на показанный далее код IL, предназначенный только для внутреннего цикла. Как ступенчатый массив он вполне оправдывает наши ожидания:

```
IL_000c: ldarg.0
IL_000d: ldfld int32[][][] ArrayPerf.ArrayPerfTest::jaggedArray
IL_0012: ldloc.0
IL_0013: ldelem.ref
IL_0014: ldloc.1
IL_0015: ldelem.ref
IL_0016: ldloc.2
IL_0017: ldelema [mscorlib]System.Int32
IL_001c: dup
IL_001d: ldind.i4
IL_001e: ldc.i4.m1
IL_001f: mul
IL_0020: stind.i4
IL_0021: ldloc.2
IL_0022: ldc.i4.1
IL_0023: add
IL_0024: stloc.2
```

Это просто серия обращений к памяти, немного математических вычислений и сохранения данных. Посмотрим для контраста на код для выполнения тех же изменений в многомерном массиве:

```
IL_000c: ldarg.0
IL_000d: ldfld int32[0..., 0..., 0...]
    ArrayPerf.ArrayPerfTest::multiArray
```

```
IL_0012: ldloc.0
IL_0013: ldloc.1
IL_0014: ldloc.2
IL_0015: call instance int32& int32[0..., 0..., 0...]::
    Address(int32 , int32 , int32)
IL_001a: dup
IL_001b: ldind.i4
IL_001c: ldc.i4.m1
IL_001d: mul
IL_001e: stind.i4
IL_001f: ldloc.2
IL_0020: ldc.i4.1
IL_0021: add
IL_0022: stloc.2
```

В основном он такой же, за исключением бросающегося в глаза вызова метода, расположенного в центральной части. Различия, которые могут быть вызваны этим обстоятельством, продемонстрированы в проекте ArrayPerf, взятом из сопровождающего книгу исходного кода.

Метод	Значение, мкс	Ошибка, мкс	Стандартное отклонение (StdDev), мкс
Negate_JaggedArray	281,0	1,7812	1,6661
Negate_MultiArray	439,6	0,2280	0,1780

Из-за этого вызова метода на обход элементов многомерного массива затрачивается намного больше времени. Теоретически характер размещения в памяти многомерных массивов и возможность последовательного размещения в ней всех значений должны дать некоторые преимущества, но из этого обстоятельства не извлекается никакой практической выгоды и, по правде говоря, оснований для применения многомерных массивов при любых обстоятельствах слишком мало.

Обобщенные коллекции

К обобщенным коллекциям относятся следующие классы:

- `Dictionary<TKey, TValue>`;
- `HashSet<T>`;
- `LinkedList<T>`;
- `List<T>`;
- `Queue<T>`;
- `SortedDictionary<TKey, TValue>`;
- `SortedList<TKey, TValue>`;
- `SortedSet<T>`;
- `Stack<T>`.