Оглавление

Предисловие 15

Об авторе 19
Контактная информация 19

Благодарности 20

От издательства 21

Введение во второе издание 22

Введение 24
Цель этой книги 24
В чем смысл выбора управляемого кода 27
Работает ли управляемый код медленнее нативного? 29
Стоит ли овчинка выделки? 31
Я что, теряю контроль? 31
Работа с CLR, а не против нее 32
Уровни оптимизации 32
Коварная соблазнительность простоты 34
Хронология совершенствования производительности среды .NET 36
.NET Core 38
Учебный исходный код 39

Глава 1. Измерение производительности и инструменты 41
Выбор предмета измерения 41
Преждевременная оптимизация 43
Сравнение усредненных и процентных показателей 44
Эталонное тестирование 46

Полезные инструменты ... 47
Visual Studio ... 49
Профилирование центрального процессора ... 51
Профилирование с помощью командной строки ... 54
Счетчики производительности ... 56
ETW-события ... 64
PerfView ... 67
Интерфейс и представления данных в PerfView ... 68
Профилировщик CLR Profiler ... 73
Анализатор производительности Windows Performance Analyzer ... 76
WinDbg ... 78
CLR MD ... 83
Анализаторы IL ... 87
MeasureIt ... 88
BenchmarkDotNet ... 89
Оснащение кода инструментами ... 91
Утилиты SysInternals ... 92
База данных ... 94
Другие инструменты ... 94
Издержки измерений ... 94
Резюме ... 95

Глава 2. Управление памятью ... 96
Выделение памяти ... 96
Операция сборки мусора ... 99
Параметры конфигурации ... 105
Сравнение сборки мусора в режиме рабочей станции и в режиме сервера ... 105
Сборка мусора в фоновом режиме ... 107
Режимы задержки ... 108
Большие объекты ... 110
Дополнительные параметры ... 111
Советы по повышению производительности ... 113
Сокращайте размеры выделяемой памяти ... 113
Самое важное правило ... 114

Сокращайте время существования объекта 115
Сбалансируйте выделение 116
Сократите количество ссылок между объектами 116
Избегайте закреплений 117
Избегайте финализаторов 118
Избегайте выделения больших объектов 120
Избегайте копирования буферов 121
Объединяйте долгоживущие и большие объекты в пулы 124
Сокращайте степень фрагментации кучи больших объектов 131
При определенных обстоятельствах выполняйте принудительную полную сборку мусора 131
Уплотняйте кучу больших объектов по требованию 133
Получайте уведомление о намечающейся сборке мусора 133
Применяйте для кэширования слабые ссылки 137
Динамически выделяйте память в стеке 144
Исследование памяти и сборки мусора 145
Счетчики производительности 145
События ETW 147
Как выглядит куча памяти моего приложения 148
Сколько времени занимает сборка мусора 151
Где именно происходит выделение памяти 155
Что за объекты находятся в куче 158
Где именно допущена утечка памяти 165
Каков размер моих объектов 170
Каким объектам выделена память в LOH 173
Какие объекты были закреплены 175
Где происходит фрагментация 177
Фрагментация виртуальной памяти 180
В каком поколении находится объект 182
Какие объекты выжили в поколении gen 0 182
Откуда был сделан явный вызов метода GC.Collect 185
Какие слабые ссылки имеются в моем процессе 186
Какие финализируемые объекты имеются в куче 186
Резюме 187

Глава 3. JIT-компиляция 189
Преимущества JIT-компиляции 190
JIT в действии 191
JIT-оптимизации 193
Сокращение времени JIT-компиляции и запуска 194
Оптимизация JIT-компиляции с помощью профилирования (Multicore JIT) 197
Когда следует применять NGEN 197
.NET Native 200
Настраиваемая предварительная подготовка 201
Когда JIT-компиляция не может составить конкуренцию 202
Исследование поведения JIT-компилятора 203
Счетчики производительности 203
ETW-события 204
Какой код подвергся JIT-компиляции 204
На какие методы и модули затрачивается больше всего времени при JIT-компиляции 207
Исследование кода, полученного после JIT-компиляции 208
Резюме 209

Глава 4. Асинхронное программирование 210
Пул потоков 212
Библиотека распараллеливания задач 213
Отмена задачи 217
Обработка исключений 218
Дочерние задачи 222
Среда TPL Dataflow 223
Параллельно выполняемые циклы 229
Советы по повышению производительности 232
Избегайте использования блокировок 232
Избегайте конвоев при блокировке и диспетчеризации 233
Использование объектов Tasks для неблокирующего ввода-вывода 233
async и await 238
О структуре программы 240
Правильно используйте таймеры 242

Подберите подходящий размер пула потоков ... 244
Не прерывайте потоки ... 245
Не меняйте приоритет потоков ... 245
Синхронизация потоков и блокировки ... 246
Нужно ли вообще заботиться о производительности? ... 246
А нужна ли вообще блокировка? ... 247
Порядок предпочтения синхронизации ... 249
Модели памяти ... 249
Использование volatile при необходимости ... 251
Использование Monitor (lock) ... 252
Использование методов Interlocked ... 255
Асинхронные блокировки ... 258
Другие механизмы блокировки ... 260
Конкурентность и коллекции ... 261
Копирование ресурса для каждого потока ... 264
Исследование потоков и конфликтов ... 265
Счетчики производительности ... 265
ETW-события ... 266
Получение информации о потоках ... 267
Визуализация задач и потоков с помощью Visual Studio ... 268
Использование PerfView для обнаружения конфликта блокировок ... 269
Где потоки блокируются на вводе-выводе ... 270
Резюме ... 270

Глава 5. Общие подходы к написанию кода и классов ... 272
Классы и структуры ... 272
Исключение из правил: изменяемая структура для хранения иерархии полей ... 274
Виртуальные методы и запечатанные классы ... 276
Свойства ... 276
Переопределение Equals и GetHashCode для структур ... 277
Потоковая безопасность ... 279
Кортежи ... 279
Диспетчеризация интерфейсов ... 280

Избегайте упаковки....281
Возвращения по ссылке (ref) и локальные значения....282
for или foreach....287
Приведение типов....289
P/Invoke....291
Делегаты....293
Исключения....295
dynamic....297
Отражение....299
Генерация кода....301
Создание шаблонов....301
Создание делегата....302
Аргументы метода....303
Оптимизация....305
Подведение итогов....305
Предварительная обработка....306
Исследование проблем производительности....307
Счетчики производительности....307
ETW-события....307
Поиск инструкций упаковки....308
Обнаружение исключений первого шанса....310
Резюме....311

Глава 6. Использование среды .NET Framework....313
Разберитесь с каждым вызываемым API....314
Множество API для решения одних и тех же задач....314
Коллекции....315
Какие коллекции лучше не использовать....315
Массивы....316
Сравнение ступенчатых и многомерных массивов....317
Обобщенные коллекции....319
Коллекции для многопоточной среды....321
Коллекции для работы с битами....323
Исходный объем....324

Сравнение ключей 325
Сортировка 326
Создание собственных типов коллекций 326
Строки 327
Сравнение строк 327
ToUpper и ToLower 328
Объединение 328
Форматирование 329
ToString 330
Избегайте разбора строк 330
Подстроки 331
Избегайте использования API, выдающих исключения при обычных обстоятельствах 331
Избегайте использования API, выделяющих память из кучи больших объектов 332
Применение ленивой инициализации 332
Удивительно высокие издержки от использования перечислений 334
Учет времени 335
Регулярные выражения 337
LINQ 339
Чтение и запись файлов 343
Оптимизация настроек HTTP и сетевых соединений 344
SIMD 347
Исследование причин возникновения проблем с производительностью 348
Резюме 349

Глава 7. Счетчики производительности 351
Использование существующих счетчиков 352
Создание пользовательского счетчика 352
Счетчики усредненных показателей 353
Счетчики мгновенных показателей 354
Дельта-счетчики 354
Процентные счетчики 354
Резюме 355

Глава 8. ETW-события 356
Определение событий 357
Потребление пользовательских событий в PerfView 360
Создание собственного слушателя ETW-событий 362
Получение подробных данных об EventSource 367
Потребление событий CLR и системы 368
Пользовательские аналитические расширения PerfView 370
Резюме 373

Глава 9. Безопасность и анализ кода 374
Представление об операционной системе, API и оборудовании 374
Ограничение использования API в определенных областях кода 375
Пользовательские правила FxCop 375
.NET Compiler Code Analyzers 382
Выполняйте централизацию и абстрагирование сложного и важного для повышения производительности кода 391
Изолируйте неуправляемый и небезопасный код 391
Отдавайте приоритет ясности кода, а не получению высокой производительности, пока нет веских причин для обратного 392
Резюме 393

Глава 10. Формирование команды, нацеленной на достижение высокой производительности 394
Выявление областей, требующих особо высокой производительности 394
Эффективное тестирование 395
Инфраструктура и автоматизация для оценки производительности 396
Доверяйте только конкретным числовым показателям 398
Эффективная система просмотра кода 399
Обучение 400
Резюме 401

Приложение А. Начало работы над повышением производительности приложения 402
Определение метрик 402
Анализ использования центрального процессора 402

Анализ использования памяти 403
Анализ JIT-компиляции 404
Анализ производительности в асинхронном режиме 404

Приложение Б. Увеличение производительности на более высоком уровне 406
ASP.NET 406
ADO.NET 407
WPF 408

Приложение В. Нотация «“O” большое» 409
«O» большое 409
Самые распространенные алгоритмы и их сложность 412
Сортировка 412
Графы 412
Поиск 413
Особый случай 413

Приложение Г. Библиография 414
Ценные источники информации 414
Люди и блоги 414

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>`.