ГЛАВА 5

Понятие и оптимизация производительности в .NET Compact Framework

В этой главе:

- Это должен знать каждый разработчик.
- Понятие CLR-движка для мобильных устройств.
- Статистика производительности в .NET Compact Framework.
- Программное измерение производительности.
- Рекомендации по повышению производительности.

Несмотря на значительную положительную динамику на аппаратном уровне и на уровне операционной системы, мобильные устройства по-прежнему ограничены по своим базовым возможностям, и это еще более поднимает значимость вопросов производительности при разработке программ для мобильных устройств. Кроме того, если учесть, что мобильные устройства питаются от батарей, старая пословица «чем меньше — тем лучше» становится жизненно важной из соображений не только производительности, но и экономии электроэнергии.

В данной главе обсуждаются принципы написания высокопроизводительного кода. Залогом достижения высокой производительности являются постановка этой цели уже на ранней стадии разработки, понимание того, как общеязыковая исполнительная среда (Common Language Runtime, CLR) управляет памятью, отказ от такой практики кодирования, при которой код после выполнения оставляет в памяти ненужный мусор. Важно также понимать, что именно делает среда, когда выполняется приложение, и как добиться того, чтобы среда делала как можно меньше.

Это должен знать каждый разработчик

В некоторых коллективах разработчиков считается, что процессы надо стремиться делать как можно более быстрыми, используя для оптимизации каждую возможность, и основным показателем при выборе конструкторских решений должна стать высокая производительность, а все остальное учитывать не

обязательно. Это неправильно. Просто код должен выполняться «достаточно быстро». Наша задача — установить, что для приложения означает «достаточно быстро».

Например, если приложению нужно три секунды для выполнения определенного действия (например, для подключения к серверу и проверки полномочий на вход), и пользователи этим вполне довольны, то тратить время на то, чтобы ускорить эту операцию, означает неразумно расходовать ресурсы. В этой ситуации требуемый уровень производительности определяется критерием приемлемости для пользователя. Соответствие требуемого уровня производительности критерию приемлемости для пользователя должно быть единственной движущей силой при определении того, как быстро должен выполняться код. Если вам необходимы дополнительные факторы для определения быстродействия кода, примите во внимание следующие вопросы: Насколько быстро работала предыдущая версия? Насколько быстро работают аналогичные продукты конкурентов? Еще раз повторимся: единственным критерием, который имеет смысл учитывать при принятии решения о необходимости оптимизировать код, является удовлетворенность пользователей скоростью его работы. Если ваш продукт быстрее, чем тот, который пользователи применяли ранее (включая аспекты, выполняемые вручную), и он не менее быстр, чем другие подобные приложения, то в его дальнейшей оптимизации действительно нет необходимости.

Когда вы пишете спецификации функциональных возможностей, следует включить в них конкретные минимальные и идеальные требования к производительности, избегая характеристик типа «эта подсистема должна быть быстрой». К несчастью, многие разработчики, написав огромное количество кода, только в последнюю минуту обнаруживают, что их код недостаточно быстрый, и только после этого пытаются оптимизировать его в надежде добиться прироста производительности. Требования к производительности должны быть установлены уже на ранней стадии разработки, чтобы впоследствии избежать неприятных сюрпризов. Частью цикла тестирования должен быть регулярный замер производительности подсистем, чтобы убедиться, что она соответствует спецификациям, и изменения кода не сказались на производительности отрицательно.

Высокопроизводительный код не создается «задним числом» — производительность закладывается в приложение при конструировании. Например, рассмотрим подсистему, которая наполняет список содержимым. Вы реализуете подсистему, а затем обнаруживаете, что загрузка 10000 позиций занимает недопустимо много времени. Следует ли после этого для оптимизации кода переделывать отдельные методы или лучше изменить всю конструкцию? Процесс загрузки 10000 позиций на ограниченном в ресурсах устройстве будет работать медленно независимо от того, как написан код. Даже если вы готовы на потерю производительности такого решения, имеет ли смысл требовать от пользователя, чтобы он прокручивал тысячи позиций на устройстве, экран которого вмещает не более десятка позиций? Очевидно, решение состоит в том, что данную подсистему изначально следует конструировать с прицелом на высокую производительность: загружайте по 100 или 200 позиций за раз, подгружайте

следующие, пока пользователь прокручивает список, предложите кнопки для перемещения на предыдущую и следующую страницы, наконец, разбейте данные по алфавиту, по категориям или по какому-нибудь иному признаку.

Лучше в первую очередь сосредоточиться не на реальной (например, времени загрузки формы), на воспринимаемой производительности (например, на времени, которое проходит от нажатия кнопки до появления указателя в виде песочных часов, символизирующих ожидание). Конструируйте свои приложения таким образом, чтобы пользователи постоянно получали отклик на свои действия. Удивительно, насколько настойчиво пользователи утверждают, что одно приложение быстрее другого, если «более быстрое» предложение предоставляет постоянный визуальный отклик (при том, что обоим приложениям на выполнение некоего действия требуется совершенно одинаковое время). Например, можно показать на экране индикатор процесса или указатель ожидания, можно также выводить в строке состояния некие промежуточные сообщения, чтобы информировать пользователя о состоянии дел, а не заставлять его вглядываться в пустой экран в ожидании завершения операции. Промежуточный отклик важен и для некоторых действий должен быть заранее предусмотрен в проекте. Например, ситуация, когда результаты поиска появляются на экране только через 10 секунд после нажатия кнопки поиска, является совершенно неприемлемой. Однако если каждые 2 секунды пользователь будет видеть промежуточные результаты, то весь поиск может занять даже 15 секунд и все равно восприниматься пользователем как быстрый.

Итак, вы получили базовые рекомендации, которые применимы к любому проекту по разработке программного обеспечения. Еще один совет — необходимо всегда понимать характеристики рабочей платформы. Это особенно верно в отношении Microsoft .NET Compact Framework. Почти любой вариант оптимизации, который придет вам в голову, имеет смысл использовать, если вы понимаете, как в .NET Compact Framework функционирует общеязыковая исполнительная среда.

Понятие CLR-движка для мобильных устройств

Разработчикам программного обеспечения обычно приходится идти на компромисс между быстродействием кода и экономным использованием памяти. Некоторые конструкторские решения могут обеспечивать высокую производительность, требуя для этого много памяти (например, для кэширования результатов в памяти), другие могут быть более медленными, но расходовать очень мало памяти (благодаря, например, обращению к процессору для вычисления неких результатов каждый раз, когда они запрашиваются). Под управлением CLR для мобильных устройств компромисс память-скорость работает не всегда. Требуется писать код, который быстро работает и в то же время экономно расходует память. Чтобы понять, почему, нужно разобраться во внутреннем функционировании сборщика мусора (Garbage Collector, GC) и оперативного (Just In Time, JIT) компилятора, известного также как JIT-компилятор, или джиттер.

JIT-компилятор

Когда вы компилируете код на Microsoft Visual Basic или C#, получаемые исполняемые файлы (EXE- или DLL-файлы) содержат не машинные коды (команды процессора), а коды на промежуточном языке (Intermediate Language, IL). Во время выполнения JIT-компилятор продолжает компиляцию: каждый метод транслируется с языка IL в машинные коды, которые затем выполняются. Важно понимать, что JIT-компиляция происходит по методам и только тогда, когда метод должен выполняться. При вызове метода делается проверка, имеется ли машинный код для данного метода. Если он имеется, то он выполняется; если машинного кода нет, то JIT-компилятор компилирует IL-код в машинный код, который затем ассоциируется с хранящейся в кэше для данного метода записью, затем машинный код, естественно, выполняется.

Каждый акт JIT-компиляции метода сказывается на производительности. Чтобы снизить степень этого влияния, попробуйте уменьшить глубину вызовов методов, объем кода методов или рекурсию, так как JIT-компилятор лучше всего работает с короткими фрагментами кода. В CLR для настольных компьютеров это происходит только один раз, так как сгенерированный машинный код метода связан с методом в течение всего времени выполнения приложения. Это отличается от того, что делает JIT-компилятор для мобильных устройств: машинный код, сгенерированный JIT-компилятором, во время выполнения в некоторых обстоятельствах, таких как серьезный дефицит памяти, может быть удален. Это явление известно под названием *пичинга* кода. Когда имеет место пичинг, удар по производительности оказывается гораздо сильнее, чем при простой JIT-компиляции одного метода. Дополнительную информацию о пичинге см. в разделе «Сборка мусора» далее в этой главе.

Еще одним отличием от настольного варианта является то, что в CLR для мобильных устройств образы неуправляемого кода не поддерживаются. Другими словами, вы не можете использовать программу Ngen из SDK для создания при установке образов неуправляемого кода, что означало бы отсутствие влияния JIT-компилятора на производительность во время выполнения. Так сделано потому, что образ неуправляемого кода в три-четыре раза больше, чем управляемая сборка, так что ограничения на размер будут очень строгими, если учесть, что одни только библиотеки .NET Compact Framework версии 2.0 занимают почти 5 Мбайт.

Подстановка

JIT-компилятор поддерживает механизм оптимизации, известный как *подстановка тел методов*. Это означает, что тела некоторых методов могут подставляться в вызывающий метод. В результате тело вызывающего метода увеличивается благодаря включению в него тела подставляемого метода, и вызов последнего больше не требуется. Все это происходит на уровне машинного кода после JIT-компиляции кода на промежуточном языке. Проиллюстрировать влияние подстановки на уровне управляемого кода можно с помощью следующих двух методов:

```
public int CallingMethod()
{
    // код, выполняющий некое действие A

    this.SomeOtherMethod();

    // код, выполняющий некое действие C
}

private void SomeOtherMethod()
{
    // код, выполняющий некое действие B
}
```

Во время выполнения эти методы становятся единым методом, то есть метод `SomeOtherMethod` подставляется, переставая существовать отдельно:

```
public int CallingMethod()
{
    // код, выполняющий некое действие A

    // код, выполняющий некое действие B

    // код, выполняющий некое действие C
}
```

JIT-компилятор для мобильных устройств обеспечивает подстановку только самых базовых методов — в действительности подстановка выполняется только для простых методов доступа, то есть свойств. Может ли метод подставляться, определяется правилами[1] Обратите внимание, что подстановка тела метода — это нюанс внутренней реализации, который может измениться, так что вам не стоит конструировать приложение, специально ориентируясь на подстановку. Тем не менее, может быть полезно делать наиболее чувствительные в плане производительности методы как можно более простыми, чтобы у них было больше шансов на подстановку. В то же время существуют методы, которые никогда не подставляются — это виртуальные методы.

Виртуальные методы

СОВЕТ

Виртуальные методы — это методы, помеченные ключевым словом virtual в C# и Overridable в Visual Basic. Они являются одним из основных строительных блоков, которые подчеркивают элегантную конструкцию иерархий объектов, обеспечивающих полиморфизм, то есть способность метода в унаследованном классе переопределять поведение метода базового класса. Способность метода переопределяться влечет за собой определенные издержки, именно поэтому она по умолчанию не поддерживается.

[1] Для подстановки метод не должен превышать по объему 16 байт IL-кода, не должен иметь ветвлений (обычно инструкций if), локальных переменных, обработчиков событий, 32-разрядных аргументов с плавающей точкой и возвращаемого значения. Кроме того, если метод имеет более одного аргумента, доступ к аргументам должен производиться от младшего к старшему (как они видны в IL-коде).

В случае JIT-компилятора для мобильных устройств вызовы виртуальных методов обходятся примерно на 40 % дороже[1], чем вызовы невиртуальных! Хотя мы не призываем конструировать ваши решения, опираясь на этот факт, здесь важно отметить, что виртуальные методы подставляться не могут. Будьте особенно осторожными с определением *виртуальных свойств*, потому что фактически свойства — это методы, несмотря на любые внешние различия. Например, рассмотрим два метода, которые идентичны за исключением того, что один вызывает виртуальное свойство, а другой идентичное ему невиртуальное:

```
private int myVar = 1;
public int MyProperty
{
    get { return myVar; }
    set { myVar = value; }
}

private int myVVar = 1;
public virtual int MyVirtualProperty
{
    get { return myVVar; }
    set { myVVar = value; }
}

public void Test1()
{
    int total=0;
    for (int i = 0; i < 1000000; i++)
    {
        total += this.MyProperty;
    }

    MessageBox.Show(total.ToString());
}

public void Test2()
{
    int total =0;
    for (int i = 0; i < 1000000; i++)
    {
        total += this.MyVirtualProperty;
    }

    MessageBox.Show(total.ToString());
}
```

Если выполнить этот код, выяснится, что метод `Test2` менее производительный, чем `Test1`. На КПК под управлением Microsoft Windows Mobile 2003 Standard Edition выполнение метода `Test1` занимает 240 мс в режиме компиляции `debug` и 45 мс в режиме `release`, а выполнение метода `Test2` — соответственно

[1] Мобильный JIT-компилятор не использует виртуальную таблицу, а это означает, что виртуальные методы при первом вызове нужно интерпретировать, а не просто искать в таблице.

320 и 190 мс. Если код собран в режиме `debug`, то разница в производительности будет меньше разницы в режиме `release` (хотя режим `release` в целом обеспечивает более высокое быстродействие, чем `debug`). В режиме `debug` разница в производительности вызвана тем, что виртуальный вызов по своей природе более медленный, а в режиме `release` эта разница увеличивается, потому что от подстановки выигрывает только невиртуальное свойство.

Сборщик мусора

Сборщик мусора отвечает за выделение ресурсов объектам и освобождение объектов, когда на них больше нет ссылок. Одним из самых серьезных притязаний, касающихся программирования в управляемой среде, является притязание на то, что разработчику больше не нужно заботиться об управлении памятью. Однако это не совсем верно: хотя об управлении памятью за вас позаботятся, приложение наверняка будет низкопроизводительным, если конструировать его без учета использования памяти. Поэтому об управлении памятью думать все-таки нужно, но не совсем так, как в случае неуправляемого кода. Прежде чем анализировать использование памяти приложением, полезно будет пояснить роль сборщика мусора в отношении производительности.

УПРАВЛЕНИЕ ПАМЯТЬЮ В WINDOWS CE И WINDOWS MOBILE

Если CLR выполняет свои обязанности, то вы, как разработчик управляемого кода, не должны беспокоиться об управлении памятью. После этих слов самое время отметить несколько полезных моментов, с которыми иногда приходится иметь дело разработчикам управляемого кода. Под управлением Windows CE может одновременно работать только 32 процесса, а если вы представляете себе, сколько процессов работает по умолчанию на устройстве под управлением Windows Mobile, вы поймете, что этого предела достичь очень легко. Кроме того, каждый процесс получает только 32 Мбайт виртуального адресного пространства, так что если вы, например, загружаете в память приложения большие растровые изображения, то не удивляйтесь, если вам не хватит памяти (не смотря даже на то, что свободная память у самого устройства еще есть). Ограничения на количество процессов и виртуальное адресное пространство — это проблемы, о которых разработчики управляемых и неуправляемых кодов должны знать и которые они должны учитывать при конструировании. Обратите внимание, что в Windows Embedded CE 6.0 оба эти ограничения сняты, но пока еще не существует версии Windows Mobile, сделанной на основе Windows Embedded CE 6.0

Обратите также внимание, что когда устройству под управлением Windows Mobile не хватает памяти, оно посылает приложениям сообщение WM_HIBERNATE, начиная с приложения, которое неактивно дольше всех, и заканчивая тогда, когда ресурсов становится достаточно. Когда приложение получает это сообщение, оно должно освободить все ресурсы, кроме совершенно необходимых. Если после отправки сообщений WM_HIBERNATE системе все равно не хватает памяти, то она начинает закрывать приложения сначала путем отправки сообщения WM_CLOSE, а затем при необходимости и путем принудительного завершения процессов, и точно так же прекращает эту деятельность, когда ресурсов становится достаточно. Когда управляемое приложение получает сообщение WM_HIBERNATE, выполняется полная сборка мусора. Если ваше приложение может дополнительно освободить активные ссылки, это следует делать путем обработки события Microsoft.

WindowsCE.MobileDevice.Hibernate (появилось в версии 2.0), после чего и происходит полная сборка мусора.

Если вас интересует управление памятью в Windows CE, прочитайте статью под названием «Windows CE .NET Advanced Memory Management» на сайте Microsoft MSDN (msdn2.microsoft.com/en-us/library/ms836325.aspx). Если вы хотите глубже разобраться во внутреннем устройстве среды CLR и ее модели расходов в Windows CE, читайте блог под названием «.Net Compact Framework Advanced Memory Management» на сайте Mike Zintel (blogs.msdn.com/mikezintel/archive/2004/12/08/278153.aspx).

Когда создается новый объект, то для хранения его содержимого требуется блок в оперативной памяти. Эта область называется кучей, и каждый процесс имеет свою кучу. В коде есть ссылка на начало объекта в куче. Эта ссылка, известная также как дескриптор, или указатель, хранится в стеке и имеет длину 4 байта в 32-разрядных системах, таких как Windows CE. В неуправляемых средах, когда программе нужно новое место в памяти для хранения объекта, необходимо проделать определенную работу для отыскания в куче подходящего адреса непрерывного блока памяти достаточного для объекта размера. Когда объект больше не нужен, специально написанный разработчиком код должен явно освободить память, расположенную по ссылке. Как же это делается в управляемом коде?

Выделение объектам памяти под управлением сборщика мусора является очень быстрой операцией, обычно более быстрой, чем в неуправляемых средах, так как сборщик мусора предварительно выделяет часть кучи и по мере создания новых объектов продолжает наращивать ее сегментами по 64 Кбайт[1]. Другими словами, память уже выделена, и каждый раз, когда требуется создать новый объект, внутренний указатель перемещается на следующий доступный адрес, подготовленный для нового объекта.

Как же освобождаются объекты в управляемом коде? Освобождение объектов известно под названием *сборки мусора* (подробности см. далее). Сборка мусора — операция не дешевая, даже в полной версии платформы, где она отличается от таковой в мобильной версии. Сборка мусора выполняется в специальном программном потоке, который запускается, когда это необходимо, при этом любая другая потоковая деятельность приостанавливается. Несколько упрощая, можно сказать, что на время сборки мусора приложение «замораживается». Обычно сборка мусора занимает несколько миллисекунд, но от того, насколько часто она выполняется, может зависеть производительность всего приложения.

Сборку мусора могут инициировать шесть ситуаций:

1. После последней сборки мусора в куче выделен суммарный объем в 1 Мбайт (в версии 1.0 было 750 Кбайт).
2. Код делает вызов `GC.Collect`.
3. Приложение переходит в фоновый режим.

[1] Если объекту требуется более 64 Кбайт, то для него создается собственный сегмент нужного размера.

4. Происходит ошибка выделения памяти для управляемого объекта.
5. Подсистема `System.Drawing` получает ошибку нехватки памяти при попытке выделения памяти под неуправляемый ресурс.
6. Приложение получает сообщение `WM_HIBERNATE`.

ВНИМАНИЕ

Важно понять и безоговорочно принять положение о том, что вызова GC.Collect в окончательном коде лучше избежать. Он обходится очень дорого и, возможно, не даст того эффекта, на который вы надеетесь. Это столь же справедливо для .NET Compact Framework, как и для полной версии .NET Framework. В общем случае система знает, когда ей лучше заняться сборкой мусора, и поскольку каждая сборка мусора влечет за собой замораживание программных потоков приложения и блуждание по куче — даже при отсутствии недостижимых для сборки объектов, — то, возможно, никакой памяти вы фактически не освобождаете, а просто увеличиваете количество приостановок приложения.

Когда выполняется сборка мусора, сборщик распознает мертвые объекты и помечает занимаемое ими в куче пространство как свободное. Если эти объекты имеют финализатор, они переносятся в другую очередь, где их метод финализации выполняется программным потоком финализатора, и при следующем сеансе сборки мусора занимаемая ими память освобождается[1]. Этот процесс известен как *простая сборка мусора*.

В дополнение к описанным действиям в качестве составляющей части сборки мусора в зависимости от степени фрагментации кучи может происходить ее сжатие. *Сжатие* — это перемещение всех живых объектов в непрерывный блок в начале кучи, в то время как неиспользуемые блоки памяти в конце кучи освобождаются для операционной системы. Такая сборка мусора называется *сборкой со сжатием*.

И наконец, помимо описанных действий как часть сборки мусора может производиться пичинг кода (см. раздел «JIT-компилятор» ранее в этой главе). В результате, исключая методы текущего стека вызова, все остальные методы при следующем вызове должны опять обрабатываться JIT-компилятором. Это называется *полной сборкой мусора*. Полная сборка мусора инициируется при возникновении любой из четырех последних ситуаций из приведенного ранее списка. При полной сборке мусора также уплотняется куча, размер которой во всех остальных случаях сохраняется на уровне 1 Мбайт (если этот размер был достигнут).

Что же это все значит? Большинство ситуаций, в которых выполняется сборка мусора, находятся вне вашего контроля, но теперь, по крайней мере, вы понимаете, когда выполняется сборка мусора и что именно при этом происходит.

[1] Финализатор — это метод, который будучи реализованным для объекта, выполняется перед «ликвидацией» объекта сборщиком мусора. Его цель — освобождение всех ресурсов, которые может занимать объект, кроме управляемых ссылок. Финализатор почти всегда реализуется в соответствии с эталоном освобождения, описываемым далее в этой главе.

Вы ничего не сможете сделать, если ваше приложение перейдет в фоновый режим или из-за работы других приложений на устройстве возникнет общий дефицит памяти. Однако и в этих условиях вам придется расплачиваться за перекомпилируемые джиттером методы вашего приложения. Кроме этих двух ситуаций, остальных вы можете избежать, не выделяя помять под объекты. Чем больше объектов вы размещаете в памяти, тем больше мусора создается и тем больше сеансов сборки мусора должно выполняться, что увеличивает вероятность накладных расходов на перекомпиляцию и суммарные задержки от деятельности сборщика мусора. Эти суммарные задержки прямо пропорциональны количеству объектов в приложении, так как для выявления живых и мертвых объектов необходимо перебрать кучу. На мобильной платформе больше чем на любой другой сохраняет свою справедливость основополагающий принцип: чем короче код, тем он быстрее. В одном из следующих разделов вы узнаете, как использовать счетчики производительности для того, чтобы определить, не вызваны ли проблемы производительности сборкой мусора.

Улучшения в версии 2.0 по сравнению с версией 1.0

Рисунки 1.10 и 1.11 в главе 1 иллюстрируют некоторые подвижки в производительности разных версий платформы.

Подводя итог, можно отметить, что в любом аспекте версия 2.0 .NET Compact Framework быстрее, чем версия 1.0. Команда разработчиков вложила много усилий в переработку и движка, и библиотек, как и в их оптимизацию. JIT-компилятор полностью переписан с прицелом на производительность, алгоритм сборщика мусора тщательно проработан, скорость вызова методов увеличена, базовые сценарии (такие, как вызов веб-служб и доступ к данным) оптимизированы. Под управлением версии 2.0 одно и то же приложение будет не только быстрее выполняться, но и гораздо быстрее запускаться. Короче говоря, одной из главных побудительных мотивов перехода с версии 1.0 на версию 2.0 является не только дополнительная функциональность, но и то, что приложения смогут выполняться быстрее (см. главу 1).

Статистика производительности в .NET Compact Framework

После того как вы поняли, что ваше приложение в том или ином аспекте имеет проблемы производительности, на следующем шаге нужно определить, в какой части кода возникает эта проблема. В управляемой среде это не всегда просто, потому что определенную работу вместо вас делает движок (как описано в предыдущем разделе), к тому же значительные объемы кода входят в библиотеки среды и находятся вне пределов вашего контроля. Например, вы можете создать единственный объект среды, но этот объект может создать еще 10 объектов от вашего имени. Именно по этой причине .NET Compact Framework может

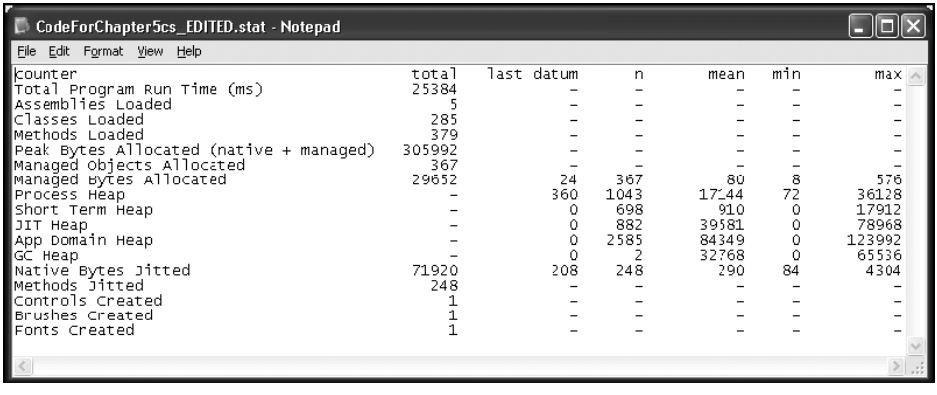
генерировать счетчики производительности, показания которых способны помочь при решении проблем.

Счетчики производительности поддерживаются и в .NET Compact Framework 1.0, но в версии 2.0 они значительно улучшены: счетчиков стало гораздо больше, их можно создавать для нескольких одновременно выполняющихся приложений, они могут обновляться, пока работает приложение (в то время как в версии 1.0 требовалось дождаться корректного завершения приложения). Наконец, в версии 2.0 имеется программа для просмотра этих данных — удаленный монитор производительности, описанный далее в этой главе.

Активизация счетчиков производительности

Счетчики производительности активизируются в реестре, во многом аналогично файлам журналов, которые мы обсуждали в главе 4 в разделе под названием «Файлы журналов». Воспользуйтесь на устройстве удаленным редактором реестра (как было показано в главе 4), найдите ключ `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETCompactFramework` и создайте под ним ключ `PerfMonitor`. Под этим новым ключом создайте значение `Counters` типа `DWORD` и присвойте ему 1. Обратите внимание, что это повлияет на производительность вашего приложения, поэтому при проведении других тестов производительности или в окончательном варианте приложения этот счетчик следует отключить, присвоив 0 значению `Counters`.

Обратите внимание, что при подключении к устройству удаленного монитора производительности, как это описано далее, включать и выключать счетчики можно просто установкой или сбрасыванием флажков. Вернитесь к рис. 4.15 в главе 4 и обратите внимание на флажок Generate.stat files, позволяющий включить режим создания файлов статистики.



Counter	total	last datum	n	mean	min	max
Total Program Run Time (ms)	25384	-	-	-	-	-
Assemblies Loaded	5	-	-	-	-	-
Classes Loaded	285	-	-	-	-	-
Methods Loaded	379	-	-	-	-	-
Peak Bytes Allocated (native + managed)	305992	-	-	-	-	-
Managed Objects Allocated	367	-	-	-	-	-
Managed Bytes Allocated	29652	24	367	80	8	576
Process Heap	-	360	1043	17.44	72	36128
Short Term Heap	-	0	698	910	0	17912
JIT Heap	-	0	882	39581	0	78968
App Domain Heap	-	0	2585	84349	0	123992
GC Heap	-	0	2	32768	0	65536
Native Bytes Jitted	71920	208	248	290	84	4304
Methods Jitted	248	-	-	-	-	-
Controls Created	1	-	-	-	-	-
Brushes Created	1	-	-	-	-	-
Fonts Created	1	-	-	-	-	-

Рис. 5.1. STAT-файл в Блокноте, иллюстрирующий некие случайные счетчики

Просмотр данных

После активизации счетчиков при каждом запуске приложения на устройстве в корневом каталоге устройства создается файл *<название_приложения>*.stat.

Когда приложение завершится, можно скопировать STAT-файл с устройства на рабочий компьютер и открыть его в Блокноте или другом текстовом редакторе. Вы увидите семь столбцов. Первый столбец — это список названий счетчиков, всего их 63. Остальные шесть столбцов показывают данные для каждого счетчика. Не ко всем счетчикам применимы все столбцы, поэтому местами вы видите прочерки. Вот эти шесть столбцов: total (итого), last datum (последнее значение), n (значение замера), mean (среднее), min (минимум) и max (максимум). На рис. 5.1 показан отредактированный STAT-файл, из которого были удалены некоторые счетчики.

STAT-файл можно также открыть в удаленном мониторе производительности, выбрав пункт Open в меню File (см. далее), но для начала нужно разобраться с назначением всех счетчиков.

Описание счетчиков производительности

63 счетчика в STAT-файле можно разделить на 10 категорий. На деле при просмотре в удаленном мониторе производительности они уже разделены на 10 категорий, и каждый имеет полезное описание. Категории и счетчики обсуждаются в следующих разделах. Один из способов понять назначение этих счетчиков — получить STAT-файл для вашего приложения и во время чтения следующих подразделов изучать реальные данные.

Загрузчик

В табл. 5.1 приведены описания шести счетчиков CLR-загрузчика. Обычно у вас будет загружен один домен приложений, и вы можете подсчитать, сколько сборок загружено, если проанализируете проект. Количество классов и методов оставалось бы загадкой, если бы не счетчики Classes Loaded и Methods Loaded. Они являются хорошими индикаторами объема приложения. Чем больше эти цифры, тем больше метаданных исполнительной среде необходимо держать в памяти. Используйте эту информацию совместно с данными журнала загрузчика, обсуждавшимся в главе 4.

Таблица 5.1. Счетчики загрузчика

Счетчик	Описание
Total Program Run Time (ms)	Время, прошедшее после вызова CLR
App Domains Created	Количество доменов приложений, созданных в процессе
App Domains Unloaded	Количество доменов приложений, выгруженных из процесса
Assemblies Loaded	Количество сборок, загруженных для всех доменов приложений
Classes Loaded	Количество классов, загруженных для всех доменов приложений
Methods Loaded	Общее количество методов, загруженных для всех доменов приложений

Обобщенные типы

В табл. 5.2 приведены описания счетчиков для обобщенных типов приложения. Не удивляйтесь, если вы не используете обобщенные типы (их поддержка появилась лишь в версии 2.0), но все равно видите ненулевые значения этих

счетчиков — обобщенные типы могут быть побочным эффектом вызова в приложении необобщенных методов. Полезно также вспомнить, что обобщенные методы не обязательно должны принадлежать к обобщенным типам! Дополнительную информацию по реализации обобщенных типов в .NET Compact Framework читайте в блоге Романа Батокова (Roman Batoukov) по адресу blogs.msdn.com/romanbat/archive/2005/01/06/348114.aspx.

Таблица 5.2. Счетчики обобщенных типов

Счетчик	Описание
Closed Types Loaded per Definition	Максимальное количество уникальных обобщенных типов, созданных для данного определения с учетом всех доменов приложений
Closed Methods Loaded per Definition	Максимальное количество уникальных обобщенных методов, созданных для данного определения с учетом всех доменов приложений
Closed Types Loaded	Количество уникальных обобщенных типов, загруженных для всех доменов приложений
Open Types Loaded	Количество открытых обобщенных типов, созданных для всех доменов приложений. Открытые типы обычно создаются только в ходе отражения
Closed Methods Loaded	Количество уникальных обобщенных методов, загруженных для всех доменов приложений
Open Methods Loaded	Количество открытых обобщенных методов, созданных для всех доменов приложений. Открытые методы обычно создаются только в ходе отражения

Блокировки и программные потоки

В табл. 5.3 приведены описания потоковых счетчиков. О программных потоках рассказывается в главе 11.

Таблица 5.3. Счетчики блокировок и программных потоков

Счетчик	Описание
Threads in Thread Pool	Текущее количество программных потоков в пуле потоков
Pending Timers	Текущее количество таймеров, ожидающих запуска
Scheduled Timers	Текущее количество запущенных или запланированных для запуска таймеров
Timers Delayed by Thread Pool Limit	Количество таймеров, приостановленных из-за ограниченности пула потоков
Work Items Queued	Количество заданий в очереди пула потоков
Uncontested Monitor.Enter Calls	Количество неконкурентных вызовов метода Monitor.Enter
Contested Monitor.Enter Calls	Количество вызовов метода Monitor.Enter, заблокированных из-за конкуренции

В этой категории есть два важных счетчика. Первый — счетчик Threads in Thread Pool. Если значение этого счетчика не меньше максимального количества потоков, которые может содержать пул потоков (по умолчанию 25), в приложении могут возникать задержки. Анализируйте это число совместно со значением счетчика Work Items Queued для того, чтобы понять, каково отношение

количества заданий, которые вы ставите в очередь, к количеству доступных для их обработки программных потоков. Счетчики, связанные с таймерами, относятся к таймеру `System.Threading.Timer`, а не к `System.Windows.Forms.Timer`, и упоминаются здесь, так как они тоже используют потоки из пула потоков.

Еще один важный счетчик — Contested Monitor.Enter Calls. Каждый раз, когда вы явно вызываете метод `System.Threading.Monitor.Enter` или неявно используете ключевое слово `lock` в C# (или `SyncLock` в Visual Basic), вы защищаете область от одновременного входа более чем одного потока (подробнее см. в главе 11). Это само по себе незначительно влияет на производительность, но может сильно затормозить работу приложения, если поток натыкается на область, где уже выполняется другой поток, из-за чего первому потоку приходится ждать. Именно для выявления подобных случаев и служит данный счетчик. Такое поведение обусловлено конструкцией большинства приложений, и если значение счетчика отличается от того, которое вы ожидали, то, возможно, вам следует переделать конструкцию. Подробности см. в главе 11.

Сборщик мусора

В табл. 5.4 приведены описания большого количества счетчиков сборщика мусора. Если вы прочитаете раздел «Сборщик мусора», размещенный ранее в этой главе, то назначение счетчиков будет вам понятно. Обратите внимание, что вместо значения счетчиков GC Latency Time, Garbage Collections (GC), GC Compactions и Code Pitchings составляют статистику сборщика мусора.

Таблица 5.4. Счетчики сборщика мусора

Счетчик	Описание
Peak Bytes Allocated (native + managed)	Максимальное количество байтов, используемых CLR, включая управляемую и неуправляемую память
Bytes Collected By GC	Количество байтов, собранных сборщиком мусора
Managed Bytes In Use After GC	Количество живых объектов после последней сборки мусора
Total Bytes In Use After GC	Количество используемых байтов управляемой и неуправляемой памяти после последней сборки мусора
GC Latency Time (ms)	Общее время (в миллисекундах), потраченное сборщиком мусора на сборку объектов и сжатие кучи
Managed Bytes Allocated	Количество байтов, выделенных в памяти в результате сборки мусора
Managed Objects Allocated	Количество объектов, для которых выделена память сборщиком мусора
Managed Strings Objects Allocated	Количество строковых объектов, для которых выделена память сборщиком мусора
Bytes of Strings Objects Allocated	Количество байтов, выделенных сборщиком мусора в памяти для строковых объектов
Garbage Collections (GC)	Количество запусков сборщика мусора
GC Compactions	Количество операций сжатия кучи, выполненных сборщиком мусора
Code Pitchings	Количество выполненных сборщиком мусора операций пичинга кода, скомпилированного JIT-компилятором

продолжение

Таблица 5.4 (продолжение)

Счетчик	Описание
Calls to GC.Collect	Количество вызовов в приложении метода GC.Collect
Pinned Objects	Количество неперемещаемых объектов, встреченных при сборке мусора
Objects Moved by Compactor	Количество объектов, перемещенных сборщиком мусора в ходе сжатия
Objects Not Moved by Compactor	Количество объектов, неперемещенных сборщиком мусора в ходе сжатия
Objects Finalized	Количество объектов, для которых запускался финализатор
Boxed Value Types	Количество значимых типов, которые были упакованы

Обратите внимание, что большое значение счетчика Boxed Value Types может означать проблему производительности, так как операции упаковки и распаковки очень дорогостоящие. Обычно упаковка имеет место при использовании типов из пространства имен `System.Collections` (см. раздел «Советы и трюки» далее в этой главе).

УПАКОВКА

Значимым типам, таким как целые, булевы, перечислимые и структурные типы, память выделяется в стеке. Такое выделение памяти делает значимые типы привлекательными с точки зрения производительности, так как им не нужно выделять или освобождать память в куче, следовательно, сборщик мусора им не требуется. Значимые типы могут быть более дорогостоящими при передаче их по значению в методах, но эта проблема легко решается путем объявления сигнатур методов для передачи их по ссылке.

Значимые типы могут также применяться везде, где ожидается ссылочный тип. Такое двойственное использование делает структуру хорошим выбором с точки зрения производительности и не нарушает базового объектно-ориентированного принципа — все является объектом. Однако имеются издержки: когда вы используете значимый тип как ссылочный, происходит операция упаковки, в результате которой создается реальный ссылочный тип (естественно, в куче, то есть он становится кандидатом для сборки мусора), являющийся эквивалентом значимого типа. Обратное преобразование к значимому типу называют распаковкой. Упаковка и распаковка — операции дорогостоящие.

Если в вашем приложении значимый тип в течение своей жизни часто подвергается упаковке, то все преимущества от использования значимых типов теряются, и лучше задействовать ссылочный тип. Для выявления случаев упаковки значимого типа требуется анализировать код. Ищите реализации интерфейса из значимого типа и т. п.

В качестве исторического отступления: название boxing (упаковка) происходит от IL-инструкции box, предназначенной для фактического приведения значимого типа к ссылочному. В качестве экстремального подхода можно сделать дамп IL-кода из ваших сборок и поискать там по ключевому слову box, чтобы найти места, где типы подвергаются упаковке.

Вам следует стремиться к низкому значению счетчика Objects Finalized, так как объект с финализатором остается в памяти дольше, что также приводит к снижению производительности, поскольку для обработки очереди финализации нужен отдельный программный поток (см. раздел «Сборщик мусора»). Используйте финализатор только тогда, когда объект непосредственно удерживает неуправляемые ресурсы, и даже тогда его следует реализовать вместе с методом `Dispose`, как описано далее.

ПРИМЕЧАНИЕ

В версии .NET Compact Framework 3.5 в дополнение к прочим журналам, описанным в главе 4, появится журнал финализатора.

В следующем примере кода иллюстрируется идиома освобождения, которой вам необходимо следовать и которая применима как в версии .NET Compact Framework, так и в полной версии. Обратите внимание на комментарии и на вывод, представленный следом за этим примером:

```
class NativeResourceHolder: IDisposable
{
    private bool alreadyDisposed = false;

    // Этот метод выполняет очистку.
    // Вызывается только из финализатора(true) ИЛИ из метода Dispose(false).
    // Это защищенный виртуальный метод, поэтому в подклассе
    // он может быть перекрыт.
    protected virtual void Dispose(bool calledFromFinalizer)
    {
        if (this.alreadyDisposed)
        {
            return;
        }
        this.alreadyDisposed = true;

        if (!calledFromFinalizer)
        {
            // Если вы храните другие ссылки на IDisposable,
            // вызовите для них метод Dispose.
            // Не делайте этого из финализатора, поскольку
            // они могут оказаться уже освобожденными.
        }

        // Всегда освобождайте неуправляемые ресурсы, такие как описатели
    }

    public void Dispose()
    {
        this.Dispose(false);
        // Получаем rid финализатора, чтобы избежать ситуации,
        // когда объект остается для внешнего цикла сборщика мусора!
        GC.SuppressFinalize(this);
    }
```

продолжение ⇩

```
    ~NativeResourceHolder()
    {
        this.Dispose(true);
    }
}
```

Из вашего вызывающего кода всегда нужно вызывать метод `Dispose()` класса, если объект более не используется. Если вы не сделаете этого, а положитесь на финализатор, это отрицательно скажется на производительности. Реализация финализатора для класса рассчитана только на тот крайний случай, когда разработчик забывает вызвать в своем коде метод `Dispose()` для объекта. Дополнительную информацию по этому поводу см. в статье «Implementing a Dispose Method» на сайте Microsoft MSDN (msdn2.microsoft.com/en-us/library/fs2xkftw.aspx).

В завершение посмотрите на счетчики сборщика мусора, и если значение счетчика Managed String Objects Allocated больше, чем вы ожидаете, изучите свой код на предмет возможной оптимизации путем замены строк типом `System.Text.StringBuilder` (о том, как и зачем это делать, рассказывается в разделе «Советы и трюки» далее в этой главе).

Память

В табл. 5.5 приведены описания счетчиков памяти. Два счетчика, на значения которых может повлиять ваше приложение, это JIT Heap (машинное представление всех управляемых методов, скомпилированных JIT-компилятором) и GC Heap (память, выделенная под все управляемые объекты). Остальные три счетчика кучи могут выражать объем вашего приложения, так что принцип «чем код короче, тем он быстрее» применим всегда.

Таблица 5.5. Счетчики памяти

Счетчик	Описание
GC Heap	Количество байтов, используемое кучей сборщика мусора
Short Term Heap	Текущее количество байтов, используемое кучей CLR краткосрочно
JIT Heap	Количество байтов, используемое кучей JIT-компилятора
Process Heap	Текущее количество байтов, используемое кучей CLR по умолчанию
App Domain Heap	Количество байтов, используемое кучей домена приложений CLR

JIT

В табл. 5.6 приведены описания счетчиков JIT-компилятора. Чем больше значение счетчиков пичинга, тем выше накладные расходы приложения на повторную JIT-компиляцию этих методов исполнительной средой при следующем вызове. Если приложение при выполнении не было переведено в фоновый режим, это значение должно быть равно 0. Обратите также внимание, что показания счетчиков JIT-компилятора и счетчиков памяти надо рассматривать совместно, чтобы получить полную картину характеристик приложения во время его выполнения.

Таблица 5.6. Счетчики JIT-компилятора

Счетчик	Описание
Method Pitch Latency Time (ms)	Общее время (в миллисекундах), затраченное методами пичинга, сгенерированными JIT-компилятором
Bytes Pitched	Количество байтов сгенерированного JIT-компилятором неуправляемого кода, который подвергается пичингу
Native Bytes Jitted	Количество байтов сгенерированного JIT-компилятором неуправляемого кода
Methods Jitted	Количество сгенерированных JIT-компилятором методов
Methods Pitched	Количество сгенерированных JIT-компилятором методов, которые подвергаются пичингу

Исключения

В табл. 5.7 представлен единственный счетчик для исключений. Как упоминалось в главе 4, запуск исключения — дорогостоящая операция. Исключения следует запускать только в чрезвычайных обстоятельствах. Если этот счетчик имеет высокое значение, то у вашего приложения проблемы в отношении не только производительности, но и общей конструкции.

Таблица 5.7. Счетчик для исключений

Счетчик	Описание
Exception Thrown	Количество запущенных управляемых исключений

Взаимодействие с платформой

В табл. 5.8 представлены счетчики взаимодействия с платформой. Значения этих счетчиков не должны вас удивлять, если вы знакомы с платформенными сервисами вызова (PInvoke) и их использованием в коде. Если они вас удивляют, следует разобраться. Пересечение границы между управляемым и неуправляемым кодом влияет на производительность, причем это влияние усиливается, когда необходим сложный маршалинг. Если вы контролируете неуправляемые коды или можете ввести неуправляемого посредника, старайтесь вместо многословных делать компактные вызовы. Конечно, предварительно нужно убедиться, что причиной проблемы производительности является взаимодействие с платформой. Наконец, объедините информацию счетчиков взаимодействия с платформой и журнала взаимодействия с платформой, как описано в главах 4 и 14.

Таблица 5.8. Счетчики взаимодействия с платформой

Счетчик	Описание
Platform Invoke Calls	Количество PInvoke-вызовов неуправляемого кода из управляемого кода, исключая внутренние PInvoke-вызовы CLR
COM Calls Using a vtable	Количество вызовов неуправляемого кода из управляемого кода, выполненных с помощью виртуальной таблицы COM-модели взаимодействия
COM Calls Using IDispatch	Количество вызовов неуправляемого кода из управляемого кода, выполненных с помощью интерфейса IDispatch COM-модели взаимодействия

продолжение

Таблица 5.8 (продолжение)

Счетчик	Описание
Complex Marshaling	Количество объектов, переданных путем маршалинга из управляемого кода в неуправляемый, что подразумевает необходимость копирования или преобразования данных
Runtime Callable Wrappers	Общее количество вызываемых во время выполнения COM-упаковщиков, которые были созданы

Сеть

Пара счетчиков, представленных в табл. 5.9, не требуют пояснений. Объединяйте информацию этих счетчиков и сетевого журнала, описанного в главах 4 и 8.

Таблица 5.9. Сетевые счетчики

Счетчик	Описание
Socket Bytes Sent	Общее количество байтов, отправленных через сокеты
Socket Bytes Received	Общее количество байтов, полученных через сокеты

Объекты оконных форм

В табл. 5.10 представлены счетчики объектов оконных форм. Когда вы создаете приложение с единственной формой и без элементов управления — то есть проект, предлагаемый по умолчанию (без кода) — то создается один элемент управления (форма) с одной кистью, одним шрифтом и ничего более. Если вы видите в приложении большие значения у разнообразных счетчиков объектов оконных форм, попробуйте повторно использовать на своих формах такие объекты, как `Font` и `Brush`.

Таблица 5.10. Счетчики объектов оконных форм

Счетчик	Описание
Controls Created	Общее количество элементов управления, созданных приложением
Brushes Created	Общее количество кистей, созданных приложением
Pens Created	Общее количество перьев, созданных приложением
Bitmaps Created	Общее количество растровых образов, созданных приложением
Regions Created	Общее количество областей, созданных приложением
Fonts Created	Общее количество шрифтов, созданных приложением
Graphics Created (FromImage)	Общее количество графических объектов, созданных методом FromImage
Graphics Created (CreateGraphics)	Общее количество графических объектов, созданных методом CreateGraphics

Удаленный монитор производительности

Файл удаленного монитора производительности (Remote Performance Monitor, RPM) называется `NetCFRPM.exe`. Он устанавливается на жесткий диск при установке пакета .NET Compact Framework SP1.

СОВЕТ

Даже если пакет SP1 не установлен на целевом устройстве, для подключения к нему все равно можно использовать удаленный монитор производительности.

Для открытия STAT-файлов в удаленном мониторе производительности его не обязательно подключать к устройству. Достаточно просто скопировать файл с устройства на рабочий компьютер и уже затем открыть его.

Ранее в этой главе уже упоминалось диалоговое окно `Logging Options` (см. рис. 4.15 в главе 4). Возможно, вы помните, что это окно можно использовать для удаленного изменения соответствующих ключей реестра, которые управляют протоколированием и счетчиками производительности. Для того чтобы это сделать, удаленный монитор производительности нужно подключить к устройству (см. далее).

Еще одна функция RPM — сбор статистики о производительности во время выполнения приложения и при желании ее просмотр в программе PerfMon на настольном компьютере. Опять-таки для этого удаленный монитор производительности нужно подключить к устройству.

Подключение к устройству

Для подключения удаленного монитора производительности к устройству нужно скопировать с рабочего компьютера в папку `Windows` мобильного устройства файлы `Netcfrtl.dll` и `Netcflaunch.exe`. Оба файла по умолчанию находятся в каталоге `C:\Program Files\Microsoft Visual Studio 8\SmartDevices\SDK\CompactFramework\2.0\v2.0\WindowsCE\wce400\armv4`. (Для устройств под управлением Windows CE версии 5.0 и Windows Mobile версии 5.0 или более поздних измените папку `wce400` на `wce500`.) При первом запуске RPM вы можете увидеть на устройстве приглашение подсистемы безопасности, касающееся файла `Netcfrtl.dll`; естественно, вам следует принять приглашение. На устройствах под управлением Windows Mobile 5.0 вам может дополнительно понадобиться подключить устройство через Microsoft ActiveSync с использованием программы настройки настольного компьютера Rapiconfig, которую можно применять для запуска подготовительных фрагментов кода на языке XML (Extensible Markup Language — расширяемый язык разметки). В любом случае XML-код для подготовки устройства таков:

```
<wap-provisioningdoc>
    - <characteristic type="Metabase">
        - <characteristic type="RAPI\Windows\netcfrtl.dll\*">
            <parm name="rw-access" value="3" />
            <parm name="access-role" value="152" />
            <!-- 152 maps to "CARRIER_TPS | USER_AUTH | MANAGER"-->
        </characteristic>
    </characteristic>
</wap-provisioningdoc>
```

Дополнительную информацию о подготовке устройств, в том числе с помощью управляемого кода и сборки `Microsoft.WindowsMobile.Configuration`, можно найти в главе 17.

Допустим, предшествующий XML-код находится в файле Rpmprovision.xml, тогда его можно запустить на устройстве следующей командой:

```
rapiconfig rpmprovision.xml
```

После подключения удаленного монитора производительности вы конфигурируете реестр при помощи диалогового окна Logging Options, которое открывается из меню Devices, и что еще более важно — собираете текущие значения счетчиков, как описано далее.

ПРИМЕЧАНИЕ

Удаленный монитор производительности, поставляемый с версией .NET Compact Framework 2.0 SP1, может подключаться только к реальному устройству, а не к эмулятору. В следующей версии RPM этот недостаток предполагается устранить, она также будет иметь другие улучшения.

Сбор текущих значений счетчиков

В меню File выберите команду Live Counters, чтобы открыть диалоговое окно Live Counters. Свое устройство вы должны увидеть в раскрывающемся списке Device (при условии, что вы успешно выполнили рекомендации предыдущего раздела), как показано на рис. 5.2.

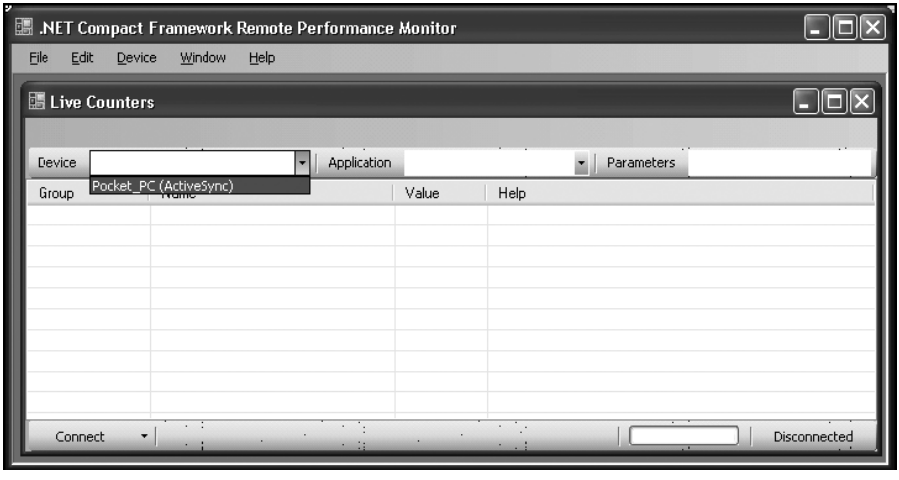


Рис. 5.2. Диалоговое окно Live Counters удаленного монитора производительности позволяет убедиться, что устройство подготовлено, хотя сам монитор еще не подключен к приложению

После выбора устройства в списке Device введите полный путь к приложению на устройстве в поле раскрывающегося списка Application. Если ваше приложение принимает аргументы командной строки, введите их в поле Parameters. Когда все будет готово, щелкните на кнопке Connect. Вы должны увидеть нечто подобное показанному на рис. 5.3.

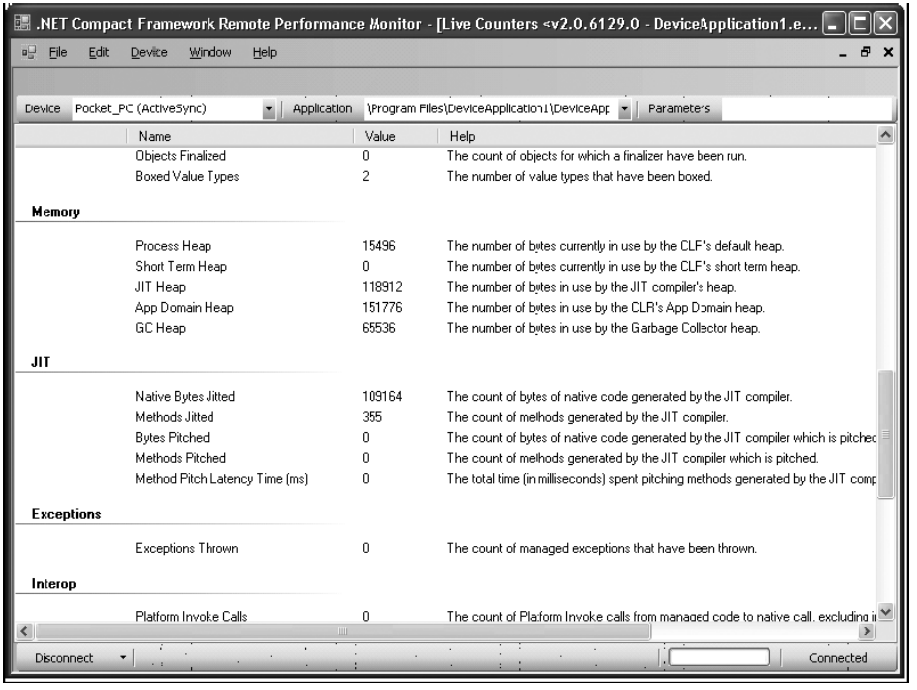


Рис. 5.3. Текущие значения счетчиков в RPM при запущенном приложении

Здесь в отличие от открытого статичного STAT-файла значения счетчиков при выполнении приложения обновляются. Например, если вы прокрутите окно до верха и посмотрите на счетчик Total Program Run Time, то увидите, что его значение постоянно увеличивается. Текущие значения хороши для наблюдения за их изменением при выполнении приложения; например, нажмите кнопку и посмотрите, какие счетчики изменились и на сколько. Если вы хотите сосредоточиться на определенных счетчиках и, к примеру, увидеть их в графическом виде, можете воспользоваться программой PerfMon, как описано в следующем разделе.

Использование монитора производительности

Запустите программу PerfMon (нажмите клавишу с логотипом Windows совместно с клавишей R) для открытия консоли управления с оснасткой измерения производительности. Щелкните правой кнопкой мыши в поле графика и выберите команду Add Counters. В списке Performance Object вы должны увидеть счетчики производительности для .NET Compact Framework, как показано на рис. 5.4.

Мы добавили два счетчика: Total Program Run Time и Managed Bytes Allocated. Вы, естественно, ожидаете, что первый счетчик будет линейно увеличиваться, а второй, начав с некоторого значения, будет увеличиваться в зависимости от особенностей вашего приложения. В примере приложения у нас есть кнопка, которая выделяет память для нескольких строк, поэтому при нажатии этой

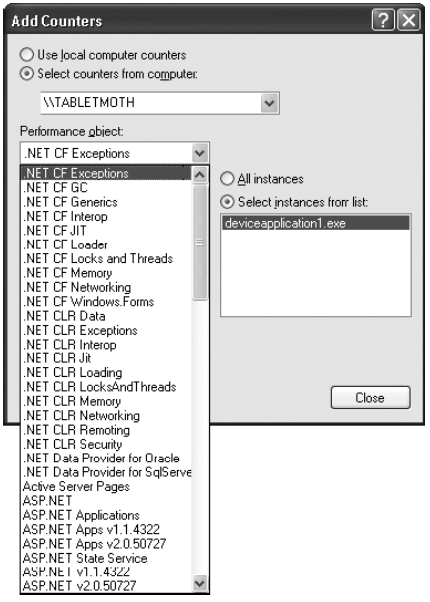


Рис. 5.4. Добавление счетчиков производительности для .NET Compact Framework

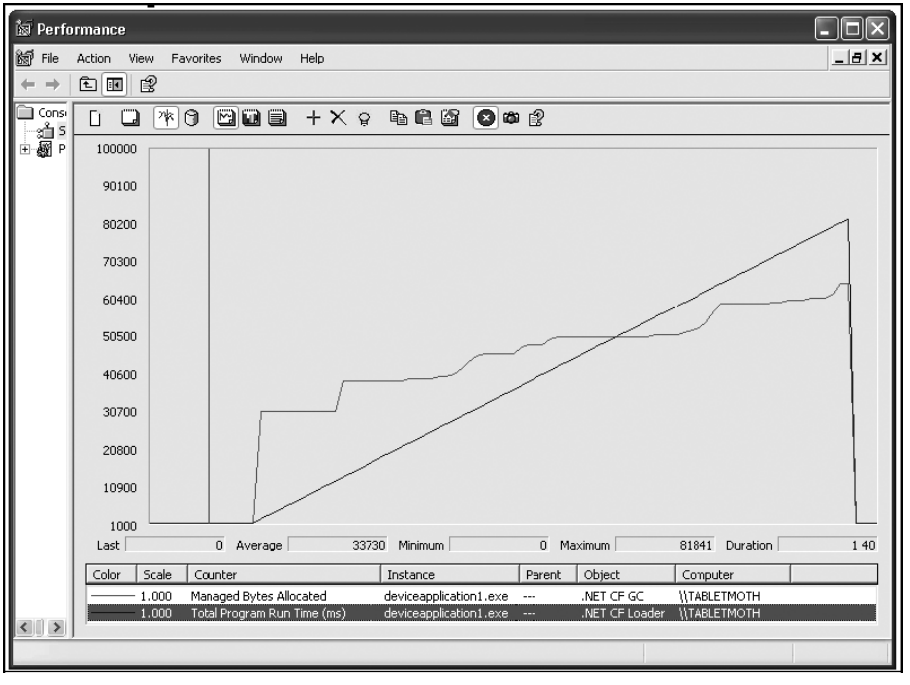


Рис. 5.5. График изменения счетчиков Total Program Run Time и Managed Bytes Allocated

кнопки показания второго счетчика должны немного увеличиться. На рис. 5.5 показан полный график: оба счетчика имеют значение 0 перед запуском приложения, потом их значения возрастают и в конце (когда приложение завершается) оба значения опять становятся нулевыми.

Программное измерение производительности

Данная глава начинается с обсуждения характеристик движка исполнительной среды .NET Compact Framework и их влияния на производительность приложения. В частности, теперь вы знаете, что использование большого объема памяти отрицательно влияет на производительность, и что чем больше кода, тем приложение медленнее. Иногда может просто потребоваться измерить, сколько времени занимает выполнение определенного метода и, например, оптимизировать производительность на микроуровне данного конкретного алгоритма.

К несчастью, в отличие от полной версии .NET Framework версия .NET Compact Framework не имеет инструментов профилирования. Это означает, что вы не сможете собрать информацию о том, как долго выполнялся каждый метод, сколько раз он вызывался, как долго работала каждая инструкция кода и т. д. — всю эту информацию можно получить только с помощью программного обеспечения настольного компьютера. Как отмечалось в главе 4 по поводу модульного тестирования, код можно запустить на настольном компьютере, если предположить, что этот код не зависит от специфических особенностей мобильных устройств или условий их эксплуатации. Хотя в большинстве случаев это не так, эту возможность всегда надо иметь в виду.

Все, что можно в этом случае сделать, — измерять время выполнения того или иного метода вручную, добавив для этого специальный код и не забыв удалить его в окончательном варианте. Традиционно для измерения времени используется счетчик тиков, доступный через свойство `Environment.TickCount`, при этом возвращается количество миллисекунд с момента старта системы. Вы можете считать значения этого свойства до и после выполнения некой операции, и разность этих значений покажет, сколько времени прошло. Это демонстрируется в следующем примере:

```
private void SomeMethodB()
{
    int start = Environment.TickCount;

    // некая длительная операция
    Thread.Sleep(2000);

    int end = Environment.TickCount;

    int millis = end - start;

    MessageBox.Show(millis.ToString());
}
```

В полной версии .NET Framework 2.0 имеется класс `Stopwatch` из пространства имен `System.Diagnostics`. Вы можете использовать его для точного измерения затраченного времени. Класс `Stopwatch` реализует оболочку для высокоточных неуправляемых прикладных программных интерфейсов `QueryPerformanceCounter` и `QueryPerformanceFrequency` таймера. Если они на вашей платформе недоступны, `Stopwatch` переходит к резервному варианту — использованию свойства `Environment.TickCount`. Хотя класс `Stopwatch` на .NET Compact Framework 2.0 не реализован, вы можете загрузить его реализацию с блога Даниеля Мот (Daniel Moth) по адресу: www.danielmoth.com/Blog/2004/12/stopwatch.html. Обратите внимание, что эта общедоступная реализация не содержит механизма отката для вызова `TickCount`, в то же время версия .NET Compact Framework 3.5 содержит полную реализацию класса `Stopwatch`.

Вот пример использования класса `Stopwatch`:

```
private void SomeMethodA()
{
    Stopwatch sw = new Stopwatch();
    sw.Start();

    // некая длительная операция
    Thread.Sleep(2000);

    sw.Stop();

    long millis = sw.ElapsedMilliseconds;

    MessageBox.Show(millis.ToString());
}
```

Хотя мы обсуждаем программное измерение времени, в некоторых обстоятельствах вам может понадобиться также программно измерить расход памяти. Следующая строка кода позволяет определить количество байтов, которое считается выделенным приложению:

```
long bytesInUseByManagedObjects = GC.GetTotalMemory(false);
```

Вы можете также использовать PInvoke-вызов неуправляемого метода `GlobalMemoryStatus`, который находит информацию о текущем расходовании системой как физической, так и виртуальной памяти. Это показано в следующем примере:

```
private void ShowMemory()
{
    MemoryStatus ms = new MemoryStatus();
    GlobalMemoryStatus(ms);

    string result =
        "Memory Load % = " + ms.MemoryLoad +
        "\r\nTotal Physical (KB) = " + ms.TotalPhysical / 1024 +
        "\r\nAvailable Physical (KB) = " + ms.AvailPhysical / 1024 +
        "\r\nTotal Virtual = (KB) " + ms.TotalVirtual / 1024+
        "\r\nAvailable Virtual = (KB) " + ms.AvailVirtual / 1024;

    MessageBox.Show(result);
}
```

```
[DllImport("coredll.dll")]
public static extern void GlobalMemoryStatus(MemoryStatus lpBuffer);

public class MemoryStatus
{
    public int Length;
    public int MemoryLoad;
    public int TotalPhysical;
    public int AvailPhysical;
    public int TotalPageFile;
    public int AvailPageFile;
    public int TotalVirtual;
    public int AvailVirtual;

    public MemoryStatus()
    {
        Length = Marshal.SizeOf(this);
    }
}
```

При программном проведении измерений нужно учитывать несколько факторов. Попытаемся резюмировать здесь некоторые из них.

Всегда собирайте ваш проект в режиме `release` с использованием всех возможных вариантов оптимизации, которые ваш язык программирования позволяет задействовать в окне свойств проекта. Выполняйте код непосредственно на устройстве, а не на эмуляторе и не через Microsoft Visual Studio. Убедитесь, что на устройстве не запущены другие приложения. Закройте все сетевые подключения и подключение к Интернету (если они не требуются при измерениях). Целью всего этого является создание условий, которые можно будет воспроизвести при последующих тестах и на которые не окажут нежелательного влияния другие внешние факторы. Никогда не делайте один замер; измерьте требуемую операцию многократно и возьмите среднее значение. Отбросьте первое измерение, чтобы компенсировать время JIT-компиляции кода. Убедитесь в том, что вы выводите результаты либо на экран, либо в файл и не делайте стандартной ошибки, включая в измерения время работы механизма входа в систему!

Наконец, нацеливайтесь на секунды, а не на миллисекунды. Несмотря на то что выполнение операции менее чем за секунду вполне возможно, вообще говоря, полагаться на это не следует. Учитывая, что поведение сборщика мусора конструктивно не детерминировано, надежда, что выполнение некой операции займет менее секунды, может не оправдаться, так что не ждите и не обещайте этого. Если вам требуются детерминированные измерения в диапазоне до секунды, подумайте об использовании в решении неуправляемого кода.

Рекомендации по повышению производительности

Большинство разработчиков пытаются отыскать некий магический знак, позволяющий писать быстрый код. Вынужден повториться — его не существует. Производительность конструктивно встраивается в приложение, а микрооптимизация

почти никогда не дает значительного эффекта. Всегда производите измерения перед тем, как пытаться что-то оптимизировать, и всегда замеряйте результат оптимизации, чтобы убедиться, дала она эффект в вашем конкретном случае или нет. В данной главе параллельно с обсуждением прочих важных вопросов приводятся советы относительно производительности, а также описываются некоторые полезные приемы и трюки, так что, пожалуйста, прочитайте всю главу, чтобы познакомиться со всеми ими.

Дополнительную информацию о производительности вообще (а не конкретно производительности мобильных устройств), предоставленную командой эталонов и правил компании *Microsoft*, можно найти на сайте MSDN (`msdn2.microsoft.com/en-us/library/ms998530.aspx`).

В следующем разделе обобщаются рекомендации, данные непосредственно командой разработки .NET Compact Framework. Пожалуйста, не следуйте этим рекомендациям вслепую. Во многих случаях оптимизация производительности входит в противоречие с принципами написания простого для расширения и сопровождения кода; прибегайте к оптимизации только тогда, когда в этом есть необходимость.

Советы и трюки

Один из описанных ранее счетчиков производительности подсчитывает количество запускаемых исключений. Запуск новых исключений — операция дорогостоящая и по возможности запуска исключений надо избегать. Кроме того, если вы пишете на языке Visual Basic, не используйте устаревшую конструкцию `On Error GoTo/Resume Next`, которая также является очень дорогостоящей, даже когда реально исключение и не запускается; вместо нее эффективнее применять конструкцию `try...catch...finally`.

Сокращение времени запуска

Загрузка приложений в версии .NET Compact Framework 2.0 происходит быстрее, чем в версии 1.0, и визуальный конструктор Visual Studio 2005 генерирует более оптимальный код компоновки формы. Вы также можете самостоятельно написать оптимальный код и для начальной загрузки, и для создания, и для размещения элементов управления на форме. Везде, где возможно, используйте две пары методов: `SuspendLayout/ResumeLayout` и `BeginUpdate/EndUpdate`.

Этот совет помогает сохранить реактивность пользовательского интерфейса. Другой способ создания чувствительного пользовательского интерфейса заключается в загрузке любых данных и выполнении прочих дорогостоящих операций в фоновом программном потоке. Вы узнаете об этом больше в главе 11. Ликвидировать эффект мерцания пользовательского интерфейса при нестандартной прорисовке экрана можно путем двойной буферизации, описываемой в главе 12.

Добавляйте к вашим сборкам криптографическую подпись только в том случае, если они размещаются в глобальном кэше сборок (Global Assembly Cache, GAC), и размещайте их там только в случае необходимости (за дополнительной информацией о том, как разместить сборку в GAC, обращайтесь к главе 6).

Загрузка приложения с криптографической подписью требует, чтобы исполнительная среда верифицировала сборку, что является недешевой операцией. Windows Mobile поддерживает также некоторые собственные схемы проверки хеш-суммы исполняемого файла при запуске. Это означает, что чем больше станет ваша сборка, тем дольше она будет загружаться. Одним из трюков, позволяющих сохранить малый размер сборки, является удаление встроенных ресурсов и загрузка их через файловую систему. Соответствующий пример есть в главе 11.

Строки, XML-разметка и данные

Нет такого приложения, которое не использовало бы строки. Строки — это устойчивые объекты, так что если у вас есть методы, которые производят множественные конкатенации и другие изменяющие операции, это означает, что ваш код фактически создает и копирует множество строковых объектов. Используйте вместо строкового типа тип `System.Text.StringBuilder`. Этот совет применим также и к миру настольных компьютеров, но в случае его применения в среде .NET Compact Framework он дает весьма впечатляющий результат, как показано в следующих примерах кода для устройства, работающего под управлением Windows Mobile 2003 Standard Edition:

```
// Занимает примерно 5 минут (или 300 000 секунд)
public static void UseString()
{
    string result = string.Empty;
    for (int i = 0; i < 10000; i++)
    {
        result += "strings are immutable " +
            "but I still use them as if they are not";
    }
}

// Занимает примерно 0,2 секунды (или 230 миллисекунд)
public static void UseStringBuilder()
{
    string result = string.Empty;
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < 10000; i++)
    {
        sb.Append("strings are immutable ").Append(
            "but I still use them as if they are not");
    }

    result = sb.ToString();
}
```

Язык XML используется в современных приложениях так же часто, как строковые операции. Язык XML хорош тем, что для него существует много утилит и он широко распространен, но это — «многословный» формат. Если вы загружаете документы размером более 64 Кбайт, то вместо весьма популярного типа `System.Xml.XmlDocument` лучше задействовать `System.Xml.XmlReader`. При загрузке XML применяйте также такие полезные в плане производительности

свойства, как `IgnoreWhitespace`, что может дать значительную экономию времени. Другие традиционные советы: используйте короткие имена элементов и атрибутов; старайтесь делать XML-код как можно более лаконичным.

Рекомендации относительно доступа к данным можно найти в главе 3. В частности, если вы новичок в SQL Server 2005 Compact Edition, научитесь применять тип `System.Data.SqlServerCe.SqlCeResultset`, призванный обеспечить эффективность приложений, использующих локальную, а не удаленную базу данных. Именно поэтому этот класс в SQL Server 2005 поддерживается только в версии Compact Edition.

Математические вычисления

Если ваше приложение должно производить сложные математические вычисления, старайтесь обходиться 32-разрядными числами, поскольку 64-разрядные не подходят для оптимизации, при которой JIT-компилятор для хранения переменных использует регистры процессора. Математические вычисления с плавающей точкой на ARM-устройствах выполняются медленно, так как эти устройства не имеют блока для выполнения операций с плавающей точкой (Floating-Point Unit, FPU), поэтому независимо от того, какой код вы используете, управляемый или нет, быстрым он не будет. При всем этом, поскольку .NET Compact Framework применяется на других процессорах в составе Microsoft XNA[1], значительная работа по оптимизации платформы для операций с плавающей точкой уже проделана, так что будущие версии должны быть лучше.

Наконец, десятичный тип очень медленный даже на настольной платформе, потому что не имеет прямого соответствия типу в языке IL. Так что не стоит ждать, что он будет быстрым в .NET Compact Framework, и используйте десятичный тип только тогда, когда это совершенно необходимо.

Отражение

Отражение — сложная и нетривиальная тема, которой можно посвятить целую книгу. Мы сопротивляемся соблазну вдаваться в излишние подробности и вместо этого резюмируем основные пункты, к которым вам следует вернуться, когда потребуется использовать отражение в приложении.

Даже на настольном компьютере отражение является дорогостоящей операцией. На мобильных устройствах ваш код может выполняться в 10 или 100 раз медленнее, чем при выполнении вызовов с ранним связыванием, к тому же увеличивается количество расходуемой приложением памяти. Приемы, позволяющие избежать отражения на мобильных устройствах, те же, что на настольном компьютере или сервере.

---

[1] XNA Framework — это набор библиотек управляемого кода для разработки игр на PC и Microsoft Xbox 360. На начальных этапах его реализации в качестве основы использовалась среда .NET Compact Framework, и все, что было сделано с тех пор полезного, является обратно переносимым на устройства, работающие под управлением .NET Compact Framework для Windows CE.

Отражение обычно касается динамического создания типов и иногда его можно избежать с помощью фабрик классов. После создания типа вы вызываете по нему члены типа, чего иногда можно избежать путем использования интерфейсов (для примера посмотрите код для этой главы на сайте поддержки данной книги). Помните также об атрибутах, особенно о настраиваемых. Атрибуты не дороги до тех пор, пока их не вызовут, что всегда делается путем отражения, чего опять-таки можно иногда избежать, используя вместо них интерфейсы (для примера посмотрите код для этой главы на сайте поддержки данной книги).

Интересно отметить, что отражение имеет место, когда код вызывает веб-службы. Если вы установили, что вызовы веб-служб являются узким местом вашего приложения, можете потратить время на создание решения с нестандартной сериализацией (двоичной), хотя в этом случае узкое место обычно возникает из-за сетевых проблем, а не из-за самого мобильного устройства. Еще один совет — при использовании веб-служб создайте для них единственный экземпляр объекта прокси и задействуйте его во всем приложении вместо того, чтобы каждый раз создавать его вновь.

Коллекции

Коллекции в той или иной форме применяются в приложениях очень часто. Обычно при использовании коллекций (например `ArrayList`) для хранения значимых типов, таких как целые, требуется упаковка. Это идеальный случай для использования обобщенных типов и особенно коллекций (таких, как `List<T>`) из пространства имен `System.Collections.Generic`. Обобщенные коллекции имеют дополнительное преимущество в виде сильной типизации и, следовательно, за счет проверок на этапе компиляции могут сократить количество исключений времени выполнения.

```
public static void UseArrayList()
{
    ArrayList a1 = new ArrayList(100000);
    ArrayList a2 = new ArrayList(100000);

    for (int i = 0; i < 100000; i++)
    {
        a2.Add(i * i); // упаковка
    }

    for (int i = 0; i < 100000; i++)
    {
        int j = (int)a2[i]; // распаковка
        a1.Add(j); // упаковка
    }
    // занимает примерно 600 миллисекунд
}

public static void UseGenerics()
{
    List<int> a1 = new List<int>(100000);
    List<int> a2 = new List<int>(100000);
```

продолжение ⇘

```
    for (int i = 0; i < 100000; i++)
    {
        a2.Add(i * i);
    }

    for (int i = 0; i < 100000; i++)
    {
        int j = a2[i];
        a1.Add(j);
    }
    // занимает примерно 75 миллисекунд
}
```

В предшествующем коде быстрее выполняется метод с обобщенным типом. Тем не менее сравнение кода с обобщенным типом и кода, в котором массив используется непосредственно, показывает, что код с массивом все же быстрее. Это неудивительно, поскольку все другие коллекции упаковывают именно массив. В качестве упражнения запустите два предыдущих метода и следующий метод, оценив, насколько они отличаются по быстродействию. Задействуйте класс `Stopwatch` и также изучите счетчики производительности для каждого случая.

```
public static void UseArray()
{
    int[] a1 = new int[100000];
    int[] a2 = new int[100000];
    for (int i = 0; i < 100000; i++)
    {
        a2[i] = i * i;
    }
    for (int i = 0; i < 100000; i++)
    {
        int j = a2[i];
        a1[i] = j;
    }
        // занимает примерно 30 миллисекунд
}
```

Естественно, характеристики разных способов доступа отличаются, поэтому всегда тестируйте именно ваш вариант. Еще раз повторим, принесение в жертву производительности кода в угоду его элегантности допустимо только при необходимости.

Еще один совет в отношении коллекций — пользуйтесь стандартным циклом `for` вместо `foreach`, так как последний задействует отражение. Всегда уточняйте ваши коллекции с помощью конструктора, который принимает объем памяти в качестве параметра; в противном случае при заполнении всего доступного объема придется выделять новый буфер большего размера и копировать в него все существующие позиции. Вы должны суметь сразу правильно определить размер буфера.

Перекрытие метода System.Object

Отражение — операция дорогостоящая. Иногда отражение может происходить не непосредственно в вашем коде, а опосредованно. Рассмотрим метод `Object.ToString()`: его поддерживает каждый объект, но если вы не перекроете его в своем коде, при вызове запустится его базовая реализация. Это — случай виртуального вызова, но стоимость виртуального вызова в смысле производительности мала по сравнению со стоимостью применяемой по умолчанию реализации метода `ToString`, в которой используется отражение. Возьмите себе на заметку, что надо перекрывать метод `ToString` в своих классах, если вы собираетесь его вызывать.

Аналогично, для значимых типов необходимо перекрывать методы `Equals` и `GetHashCode`. Эти методы являются источником серьезных проблем производительности, и их перекрытые версии нужно усовершенствовать в отношении ваших значимых типов. Эти методы являются виртуальными, в них используется отражение, к тому же они требуют упаковки, поскольку для вызова методов значимый тип должен быть упакован. Перекрытие этих двух методов в вашем значимом типе (то есть в вашей структуре) устранит упаковку, виртуальный вызов, а также отражение (если вы сможете реализовать эти методы таким образом, чтобы отражение не использовалось).

Напутствие

Пожалуйста, не применяйте к вашему коду вслепую все рекомендации из этой главы. В первую очередь проведите измерения, чтобы понять, имеются ли проблемы производительности. Если имеются, наметьте лучший способ оптимизации проблемного кода, и только после этого подумайте о возможности последовать одной из приводимых рекомендаций. После внесения изменений снова замерьте производительность, чтобы понять, дала ли оптимизация эффект (который и является критерием сохранения внесенных изменений). Некоторые из приводимых рекомендаций противоречат принципам конструирования хороших приложений, которыми ради производительности можно жертвовать только в случае необходимости.

Заключение

Принципы оптимизации производительности для разных платформ в общем одни и те же. Советы по производительности, касающиеся среды .NET для настольных компьютеров, применимы также и для мира мобильных устройств. В этой главе представлены некоторые базовые рекомендации по оптимизации производительности и выделены приемы, применимые к мобильным устройствам, для которых характерна ограниченность в ресурсах. Для написания эффективного кода чрезвычайно важно понимание общеязыковой среды времени выполнения (CLR) мобильных устройств. В этой главе показано, как использовать счетчики производительности и удаленный монитор производительности.

Примеры кодов этой главы призваны помочь вам научиться программно измерять расходование памяти или скорость выполнения операций.

Высокая производительность должна быть в числе технических требований к проекту, причем не менее важных, чем требования относительно функциональности. Эта глава дает вам необходимые для оптимизации кода и написания приложений знания и инструменты, хорошо приспособленные для мобильной платформы: интерфейс этих приложений всегда реактивен, делают они свою работу быстро и не расходуют ресурсов, которые могли бы потребоваться другим приложениям.