ГЛАВА 7

Время жизни объекта

Одним из достоинств модели управляемого исполнения .NET является то, что среда выполнения способна автоматизировать большую часть работы по управлению памятью в вашем приложении. Я показал множество примеров создания объектов с помощью ключевого слова `new`, но ни в одном из них явно не освобождал память, занятую этими объектами.

В большинстве случаев не нужно предпринимать каких-либо действий для освобождения памяти. Среда выполнения предоставляет *сборщик мусора* (GC), механизм, который автоматически обнаруживает, когда объекты больше не используются, и восстанавливает память, которую они занимали, чтобы ее можно было использовать для новых объектов[1]. Однако существуют определенные схемы использования, которые могут вызвать проблемы с производительностью или даже полностью свести на нет работу сборщика мусора, поэтому полезно понимать, как он работает. Это особенно важно для долго выполняющихся процессов, которые могут работать в течение нескольких дней (процессы с коротким сроком службы вполне могут пережить несколько утечек памяти).

Сборщик мусора разработан для эффективного управления памятью, но память — не единственный ограниченный ресурс, с которым вам, скорее всего, придется иметь дело. Некоторые вещи занимают небольшой объем памяти в CLR, но при этом являются достаточно затратными, например подключение к базе данных или дескриптор из API операционной системы. Сборщик мусора не всегда хорошо с этим справляется, поэтому я объясню работу `IDisposable`, интерфейса, предназначенного для работы с тем, что должно быть освобождено более срочно, чем память.

Значимые типы часто имеют совершенно разные правила, определяющие их время жизни — например, некоторые значения локальных переменных существуют только до тех пор, пока работает содержащий их метод. Тем не

[1] В этой главе аббревиатура GC используется для обозначения как *механизма сборки мусора*, так и *сборки мусора*, которую делает сборщик мусора.

менее иногда значимые типы действуют как ссылочные типы и в конечном итоге управляются сборщиком мусора. Я поясню, как это может оказаться полезным, и я расскажу о механизме *упаковки*, который делает это возможным.

Сборка мусора

CLR поддерживает работу *кучи* — сервиса, который предоставляет память для объектов и значений, временем жизни которых управляет сборщик мусора. Каждый раз, когда вы создаете экземпляр класса с помощью `new` или создаете новый объект массива, CLR выделяет новый блок кучи. Когда нужно освободить этот блок, решает сборщик мусора.

Блок кучи содержит все нестатические поля объекта или все элементы, если это массив. CLR также добавляет заголовок, который вашей программе непосредственно не виден. Он включает в себя указатель на структуру, описывающую тип объекта, и предназначен для поддержки операций, которые зависят от реального типа объекта. Например, если вы вызываете `GetType` для ссылки, среда выполнения использует этот указатель для определения типа. (Тип часто не полностью определяется статическим типом ссылки, которая может быть указателем на тип интерфейса или базовый класс фактического типа.) Он также используется для определения, какой метод использовать при вызове виртуального метода или члена интерфейса. CLR также использует заголовок, чтобы узнать, насколько велик блок кучи, — заголовок не включает размер блока, потому что среда выполнения может рассчитать его исходя из типа объекта. (Большинство типов имеют фиксированный размер. Есть только два исключения — строки и массивы, которые CLR обрабатывает как особые случаи.) Заголовок содержит еще одно поле, которое используется для ряда задач, включая многопоточную синхронизацию и генерацию хеш-кода по умолчанию. Заголовки блоков кучи — это просто деталь реализации, так что другие реализации CLI могут выбирать иные стратегии. Тем не менее будет полезно узнать об издержках. В 32-битной системе заголовок имеет длину 8 байт, а если вы работаете в 64-битном процессе, то он занимает 16. Таким образом, объект, содержащий только одно поле типа `double` (8-байтовый тип), будет использовать 16 байт в 32-битном процессе и 24 байта в 64-битном процессе.

Хотя объекты (т. е. экземпляры класса) всегда располагаются в куче, экземпляры значимых типов ведут себя по-другому: некоторые располагаются в куче, а некоторые нет. Например, CLR хранит ряд локальных переменных

значимых типов в стеке, но если значение находится в поле экземпляра класса (а экземпляр класса располагается в куче), то это значение будет расположено внутри этого объекта в куче[2]. А в некоторых случаях значение будет занимать целый блок кучи.

Если вы обращаетесь к чему-то через переменную ссылочного типа, то вы обращаетесь к чему-то в куче. Важно уточнить, что именно я имею в виду под переменной ссылочного типа, потому что, к сожалению, терминология здесь немного запутанная: C# использует термин *ссылка* для описания двух совершенно разных вещей. В контексте этого обсуждения ссылка — это то, что можно сохранить в переменной типа, который является производным от `object`, но не от `ValueType`. В это число не входит каждый аргумент метода вида `in-`, `out-` или `ref-`, переменная ссылочного типа или возвращаемое значение. Хотя все это своего рода ссылки, аргумент `ref int` является ссылкой на значимый тип, а это не то же самое, что ссылочный тип. (CLR фактически использует другой термин для механизма, который поддерживает `ref`, `in` и `out`: он называет это *управляемыми указателями*, тем самым давая понять, что они довольно сильно отличаются от ссылок на объекты.)

Модель управляемого выполнения, используемая C# (и всеми языками .NET), означает, что CLR знает о каждом блоке кучи, который создает ваш код, а также о каждом поле, переменной и элементе массива, в которых ваша программа хранит ссылки. Эта информация позволяет среде выполнения в любой момент определить, какие объекты *достижимы*, т. е. к которым программа может получить доступ, чтобы использовать их поля и другие элементы. Если объект недостижим, то программа по определению никогда не сможет использовать его снова. Чтобы проиллюстрировать, как CLR определяет достижимость, я написал простой метод, показанный в листинге 7.1, который выбирает веб-страницы с веб-сайта моего работодателя.

Листинг 7.1. Использование и выброс объектов

```
public static string WriteUrl(string relativeUri)
{
    var baseUri = new Uri("https://endjin.com/");
    var fullUri = new Uri(baseUri, relativeUri);
    var w = new WebClient();
    return w.DownloadString(fullUri);
}
```

[2] Значимые типы, определенные с помощью `ref struct`, являются исключением: они всегда находятся в стеке (см. главу 18).

Обычно тип `WebClient`, показанный в листинге 7.1, не используют. В большинстве случаев вы бы задействовали более свежий `HttpClient`. Я сознательно не использую здесь `HttpClient`, потому что он содержит только асинхронные методы, что делает время жизни переменных не таким очевидным.

CLR анализирует способ использования локальных переменных и аргументов метода. Например, хотя аргумент `relativeUri` находится в области доступности всего метода, мы используем его лишь один раз в качестве аргумента при создании второго `Uri`. Переменная считается *действительной* (live) от точки, где она получает значение, до точки, в которой она в последний раз используется. Аргументы метода являются действительными от начала метода и до последнего использования, кроме тех случае, когда они не используются вообще и, следовательно, никогда не являются действительными. Локальные переменные становятся действительными позже; `baseUri` становится действительной после того, как ей было присвоено начальное значение, а затем перестает быть таковой с последним использованием, что в этом примере происходит в той же точке, что и `relativeUri`. Действительность является важным свойством при определении того, используется ли конкретный объект.

Чтобы оценить роль, которую играет действительность (liveness), предположим, что, когда листинг 7.1 достигает строки, которая создает `WebClient`, у CLR недостаточно свободной памяти для хранения нового объекта. В этот момент он может запросить больше памяти у ОС, но у него также есть возможность попытаться освободить память от объектов, которые больше не используются, а это означает, что нашей программе не потребуется больше памяти, чем она уже использует. В следующем разделе описывается процесс, который использует CLR, когда обрабатывает второй вариант[3].

Определение достижимости

CLR начинает с определения всех корневых ссылок в вашей программе. Корень — это место хранения, такое как локальная переменная, содержащее ссылку, которая, как известно CLR, была инициализирована. Ваша программа может использовать ее в будущем без посредничества какой-либо

[3] CLR не всегда ждет, пока закончится память. Подробности чуть позже. Пока важным является то, что время от времени он будет пытаться освободить место.

другой ссылки на объект. Не все места хранения считаются корневыми. Если объект содержит поле экземпляра какого-то ссылочного типа, то это поле не является корневым, потому что перед тем, как вы сможете его использовать, вам нужно получить ссылку на содержащий объект, который, возможно, недостижим. Однако статическое поле ссылочного типа является корневой ссылкой, потому что программа может прочитать значение в этом поле в любое время — единственная ситуация, в которой это поле станет недоступным в будущем, — это когда компонент, определяющий тип, выгружен, что в большинстве случаев случается лишь при выходе из программы.

Локальные переменные и аргументы метода более интересны. Иногда они являются корневыми, а иногда нет. Это зависит от того, какая именно часть метода выполняется в данный момент. Локальная переменная или аргумент могут быть корневыми, только если в данный момент поток выполнения находится внутри области, в которой эта переменная или аргумент являются действительными. Таким образом, в листинге 7.1 `baseUri` становится корневой ссылкой только после того, как ей присвоено начальное значение, и до вызова создания второго `Uri`, что является довольно узким интервалом. Переменная `fullUri` остается корневой ссылкой чуть дольше, потому что становится действительной после получения начального значения и продолжает работать во время построения `WebClient` в следующей строке; ее действительность завершается только после вызова `DownloadString`.

Когда последнее использование переменной является аргументом в вызове метода или конструктора, она перестает существовать с началом вызова этого метода. В этот момент вызываемый метод перехватывает инициативу — в начале действительны его собственные аргументы (за исключением тех, которые он не использует). Тем не менее они обычно перестают быть таковыми до возврата метода. Это означает, что в листинге 7.1 объект, на который ссылается `fullUri`, может перестать быть доступным через корневые ссылки, прежде чем произойдет возврат из `DownloadString`.

Поскольку набор динамических переменных изменяется по мере выполнения программы, набор корневых ссылок также претерпевает изменения, поэтому CLR нужна возможность сформировать моментальный снимок соответствующего состояния программы. Детали не задокументированы, но сборщик мусора может приостановить все потоки, в которых выполня-

ется управляемый код, если это необходимо для обеспечения правильного поведения.

Действительные переменные и статические поля — не единственные элементы, которые могут быть корневыми. Временные объекты, созданные в результате вычисления выражений, должны оставаться действительными столько времени, сколько необходимо для завершения вычислений. Поэтому вполне возможно существование корневых ссылок, которые не соответствуют напрямую никаким именованным объектам в вашем коде. Есть и другие типы корневых ссылок. Например, класс `GCHandle` позволяет явно создавать новые корневые ссылки, что может быть полезно в сценариях взаимодействия, где некий неуправляемый код нуждается в доступе к определенному объекту. Существуют ситуации, в которых корневые ссылки создаются неявно. Взаимодействие с COM-объектами может создавать корневые ссылки без явного использования `GCHandle` — если CLR необходимо создать оболочку COM для одного из ваших объектов .NET, эта оболочка фактически будет корневой ссылкой. Вызовы в неуправляемый код могут также включать передачу указателей на память в куче, что будет означать, что соответствующий блок кучи должен рассматриваться как достижимый на время вызова. Спецификация CLI не содержит полного списка способов появления корневых ссылок, но общий принцип заключается в том, что они будут там, где важно обеспечить достижимость используемых объектов.

Составив полный список текущих корневых ссылок для всех потоков, сборщик мусора определяет, какие объекты могут быть доступны по этим ссылкам. Он просматривает каждую ссылку по очереди, и, если она не равна `null`, сборщик знает, что объект, на который она ссылается, достижим. Среди них могут содержаться дубликаты, так как несколько корневых ссылок могут ссылаться на один и тот же объект. В связи с этим сборщик мусора отслеживает, какие объекты он уже встречал. Для каждого нового обнаруженного объекта сборщик добавляет все поля экземпляра ссылочного типа из этого объекта в список ссылок для оценки, опять же, избавляясь от дубликатов. (Это включает в себя скрытые поля, сгенерированные компилятором, такие как поля для автоматических свойств, которые я описал в главе 3.) Это означает, что если объект достижим, это относится и ко всем объектам, на которые он ссылается. Сборщик мусора повторяет этот процесс, пока у него не закончатся ссылки для проверки. Любые объекты, которые он не обозначил доступными, считаются недоступными, потому что сборщик просто делает то, что делает программа: использует только те объекты, которые

доступны прямо или косвенно через ее переменные, временное локальное хранилище, статические поля и другие корневые ссылки.

Вернемся к листингу 7.1. К чему все это приведет, если CLR решит запустить сборку мусора при создании `WebClient`? Переменная `fullUri` все еще действительная, поэтому `Uri`, к которой она относится, достижима, но `baseUri` больше не действительна. Мы передали копию базовой `Uri` в конструктор для второй `Uri`, и если бы она содержала копию ссылки, то не имело бы никакого значения, что `baseUri` не является действительной; если есть способ добраться до объекта от корневой ссылки, то объект достижим. Но как видим, вторая `Uri` копии не содержит, поэтому первая `Uri`, выделенная в примере, будет считаться недоступной, а CLR будет вправе восстановить занимаемую ей память.

Одним из важных результатов определения достижимости является то, что сборщик мусора не смущают циклические ссылки. Это одна из причин, по которой .NET использует сборку мусора вместо подсчета ссылок (еще один популярный подход для автоматизации управления памятью). Если у вас есть два объекта, которые ссылаются друг на друга, схема подсчета ссылок решит, что оба объекта используются, поскольку каждый из них упоминается как минимум один раз. Но объекты могут быть недоступны — если нет других ссылок на них, приложение не может их использовать. Подсчет ссылок не способен этого обнаружить, поэтому он может стать причиной утечек памяти. Но для схемы, использующей сборщик мусора CLR, тот факт, что они ссылаются друг на друга, не имеет значения — сборщик мусора никогда не доберется ни до одного из них, поэтому он правильно определит, что они больше не используются.

Случайное нарушение работы сборщика мусора

Хотя сборщик мусора может обнаружить пути, которыми программа может добраться до объекта, он не может доказать, что это обязательно произойдет. Взгляните на поразительно идиотский кусок кода в листинге 7.2. Хотя вы никогда не написали бы такой плохой код, в нем содержится распространенная ошибка. Эта проблема обычно вызывается более изощренными способами, но сначала я хочу указать на нее в более очевидном примере. Показав, как этот код предотвращает освобождение неиспользуемых объектов сборщиком мусора, я опишу не такой простой, но более реалистичный сценарий, при котором часто возникает такая же проблема.

Код суммирует числа от 1 до 100 000, а затем отображает их среднее арифметическое. Первая ошибка здесь в том, что даже не нужно делать сложение в цикле, потому что есть простое и хорошо известное решение в виде формулы: `n * (n +1) / 2`, где `n` в нашем случае равно 100 000. Помимо этой математической оплошности код делает нечто еще более глупое: создает список, содержащий каждое добавляемое число, однако все, что он делает с этим списком, — это извлекает его свойство `Count` для вычисления среднего значения в конце. Но что еще хуже, перед тем, как поместить его в список, код преобразует каждое число в строку, но по сути никогда не использует эти строки.

Листинг 7.2. Крайне неэффективный кусок кода

```
static void Main(string[] args)
{
    var numbers = new List<string>();
    long total = 0;
    for (int i = 1; i < 100_000; ++i)
    {
        numbers.Add(i.ToString());
        total += i;
    }
    Console.WriteLine("Total: {0}, average: {1}",
        total, total / numbers.Count);
}
```

Очевидно, что это надуманный пример, и хотел бы я сказать, что никогда не встречал подобную нелепицу в реальных программах. К сожалению, в жизни я сталкивался с такими примерами, и они были гораздо более запутаны — когда вы встречаете такое, то требуется около получаса, чтобы понять, правда ли этот код такой ошеломляюще бесполезный. Но я это пишу не для того, чтобы пожаловаться на стандарты разработки ПО. Цель этого примера — показать, как вы можете столкнуться с пределом возможностей сборщика мусора.

Предположим, что цикл в листинге 7.2 работает какое-то время — возможно, он находится на своей 90 000-й итерации и пытается добавить запись в список `numbers`. Предположим, что `List<string>` задействовал всю свою свободную емкость, и поэтому метод `Add` должен будет выделить новый, еще больший внутренний массив. В этот момент CLR может принять решение запустить сборку мусора, чтобы выяснить, можно ли освободить место. Что же произойдет?

Листинг 7.2 создает три вида объектов: создает `List<string>` в начале цикла, создает новую `string`, каждый проход цикла вызывая `ToString ()` для `int`, и, что не так очевидно, `List<string>` определяет `string[]` для хранения ссылок на эти строки. Поскольку мы продолжаем добавлять новые элементы, ему придется выделять все большие и большие массивы. (Этот массив является деталью реализации `List<string>`, поэтому напрямую его не увидеть.) Таким образом, вопрос в том, от каких из этих объектов сборщик мусора может избавиться, чтобы освободить место для большего массива при вызове `Add`?

Наша переменная `numbers` остается действительной до последнего оператора программы, а мы смотрим на более раннюю точку кода, поэтому объект `List<string>`, на который она указывает, достижим. Массив `string[]`, который она в данное время использует, также должен быть достижимым: он выделяет более новый, более крупный объект, но ему нужно будет скопировать содержимое старого объекта в новый, поэтому список все еще должен хранить в одном из своих полей ссылку на этот текущий массив. Поскольку этот массив все еще достижим, то каждая строка, на которую ссылается массив, также будет достижима. Наша программа уже создала 90 тысяч строк, и сборщик обнаружит их все, начав с нашей переменной `numbers` и просматривая поля объекта `List<string>`, на который она ссылается, а затем просматривая каждый элемент в массиве, на который ссылается одно из `private` полей списка.

Единственные выделенные элементы, которые сборщик мусора может собрать, — это старые массивы `string[]`, которые `List<string>` создавал, когда список был меньше, и на которые он больше не имеет ссылки. К тому времени, когда мы добавим 90 тысяч элементов, список, вероятно, несколько раз изменится в размерах. Таким образом, в зависимости от того, когда сборщик мусора последний раз запускался, он, вероятно, сможет найти несколько таких неиспользуемых массивов. Но интереснее то, чего он не может освободить.

Программа никогда не использует 90 тысяч строк, которые создает, поэтому в идеале мы хотели бы, чтобы сборщик мусора освободил занимаемую ими память, так как они будут занимать несколько мегабайтов. Из-за того что программа такая короткая, легко увидеть, что эти строки не используются. Но сборщик не будет об этом знать; он основывает свои решения на достижимости и, начав с переменной `numbers`, правильно определяет, что все 90 тысяч строк достижимы. Как известно сборщику мусора, вполне

возможно, что свойство `Count` списка, которое мы используем после завершения цикла, будет обращаться к содержимому списка. Мы с вами знаем, что это не так, потому что в этом нет необходимости, но лишь потому, что мы знаем, что означает свойство `Count`. Чтобы сборщик мог сделать вывод, что программа никогда не будет использовать какие-либо элементы списка прямо или косвенно, ему необходимо знать, что `List<string>` делает внутри своих методов `Add` и `Count`. Это будет означать анализ с уровнем детализации, намного превышающим описанные мной механизмы, что может сделать сборку мусора гораздо более затратной. Более того, даже при серьезном повышении сложности, необходимом для определения того, какие достижимые объекты этот пример никогда не будет использовать, в более реалистичных сценариях сборщик вряд ли сможет делать более достоверные прогнозы, чем те, которые основаны на достижимости.

Например, гораздо более вероятно, что вы столкнетесь с этой проблемой в кэше. Если вы напишите класс, который кэширует данные, которые затратно получать или рассчитывать, то представьте себе, что произойдет, если ваш код только добавляет элементы в кэш и никогда не удаляет их. Все кэшированные данные будут доступны до тех пор, пока доступен сам объект кэша. Проблема заключается в том, что ваш кэш будет занимать все больше и больше места, и если на вашем компьютере недостаточно памяти для хранения всех фрагментов данных, которые ваша программа, возможно, захочет использовать, в конечном итоге он исчерпает память.

Наивный разработчик будет жаловаться, что это проблемы сборщика мусора — смысл сборщика мусора заключается в том, чтобы не думать об управлении памятью, так почему же у меня вдруг заканчивается память? Но, конечно, проблема в том, что у сборщика мусора нет возможности узнать, какие объекты можно безопасно удалить. Он не экстрасенс и не способен точно предсказать, какие кэшированные элементы могут понадобиться вашей программе в будущем — если код выполняется на сервере, то будущее использование кэша может зависеть от того, какие запросы получает сервер, чего сборщик мусора точно не может предсказать. Поэтому хотя можно предположить, что управление памятью действует достаточно умно, чтобы проанализировать что-то столь простое, как в листинге 7.2, в целом это не та проблема, которую способен решить сборщик мусора. Таким образом, если вы добавляете объекты в коллекции и сохраняете доступность этих коллекций, сборщик мусора будет считать все в этих коллекциях достижимым. Удаление элементов — это ваша задача.

Коллекции не единственный способ, которым вы можете обмануть сборщик мусора. Как я покажу в главе 9, существует распространенный сценарий, при котором неосторожное использование событий может вызвать утечку памяти. В более общем смысле, если ваша программа делает объект достижимым, сборщик мусора не может определить, собираетесь ли вы снова использовать этот объект, поэтому он должен быть сохранен.

Тем не менее с небольшой помощью сборщика мусора эту проблему можно сделать менее острой.

Слабые ссылки

Хотя сборщик мусора будет проходить по обычным ссылкам в полях достижимого объекта, можно сохранить слабую ссылку. Сборщик не проходит по слабым ссылкам, поэтому, если единственный способ достичь объекта — это слабые ссылки, сборщик ведет себя так, как будто объект недоступен, и удаляет его. Слабая ссылка — это способ сообщить CLR: «Не ориентируйся на меня при сохранении этого объекта, но пока он еще кому-то нужен, я бы хотел иметь к нему доступ». В листинге 7.3 показан кэш, где используется `WeakReference<T>`.

Листинг 7.3. Использование слабых ссылок в кэше

```
public class WeakCache<TKey, TValue> where TValue : class
{
    private readonly Dictionary<TKey, WeakReference<TValue>> _cache =
        new Dictionary<TKey, WeakReference<TValue>>();

    public void Add(TKey key, TValue value)
    {
        _cache.Add(key, new WeakReference<TValue>(value));
    }

    public bool TryGetValue(TKey key, out TValue cachedItem)
    {
        WeakReference<TValue> entry;
        if (_cache.TryGetValue(key, out entry))
        {
            bool isAlive = entry.TryGetTarget(out cachedItem);
            if (!isAlive)
            {
                _cache.Remove(key);
            }
```

```
            return isAlive;
        }
        else
        {
            cachedItem = null;
            return false;
        }
    }
}
```

Этот кэш хранит все значения с помощью `WeakReference<T>`. Его метод `Add` передает объект, для которого мы хотели бы использовать слабую ссылку в качестве аргумента конструктора для нового объекта `WeakReference<T>`. Метод `TryGetValue` пытается получить значение, ранее сохраненное с помощью `Add`. Сначала проверяется, содержит ли словарь соответствующую запись. Если содержит, то значением этой записи будет `WeakReference<T>`, которую мы создали ранее. Мой код вызывает метод `TryGetTarget` слабой ссылки, который вернет `true`, если объект все еще доступен, и `false`, если нет.

Наличие не обязательно означает достижимость. Объект мог стать недоступным со времени последней сборки мусора. Или, возможно, с момента выделения объекта вообще не было сборки мусора. `TryGetTarget` заботит не то, доступен ли объект прямо сейчас, а лишь то, обнаружил ли сборщик мусора, что он подлежит освобождению.

Если объект доступен, `TryGetTarget` предоставляет его через параметр `out`, и это будет строгой ссылкой. Таким образом, если метод возвращает значение `true`, нам не нужно беспокоиться о том, что через мгновение объект может стать недоступным, — тот факт, что мы теперь сохранили эту ссылку в переменной, которую вызывающая сторона передала через аргумент `cachedItem`, сохранит и сам объект. Если `TryGetTarget` возвращает `false`, мой код удаляет соответствующую запись из словаря, поскольку она представляет объект, которого больше не существует. Это важно, потому что даже если слабая ссылка не сохранит свою цель, `WeakReference<T>` — это отдельный объект, и сборщик не может освободить его, пока я не удалю его из этого словаря. Листинг 7.4 задействует этот код, вызывая пару сборок мусора, чтобы мы могли увидеть работу сборщика в действии. (Каждый этап разделен на отдельные методы с отключенным встраиванием, потому что в противном случае JIT-компилятор в .NET Core встроит эти методы и в итоге создаст

скрытые временные переменные, которые могут сделать массив достижимым дольше, чем следовало бы, тем самым исказив результаты теста.)

Листинг 7.4. Пробуем слабый кэш

```
class Program
{
    static WeakCache<string, byte[]> cache =
             new WeakCache<string, byte[]>();
    static byte[] data = new byte[100];

    static void Main(string[] args)
    {
        AddData();
        CheckStillAvailable();

        GC.Collect();
        CheckStillAvailable();

        SetOnlyRootToNull();
        GC.Collect();
        CheckNoLongerAvailable();
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private static void AddData()
    {
        cache.Add("d", data);
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private static void CheckStillAvailable()
    {
        Console.WriteLine("Retrieval: " +
            cache.TryGetValue("d", out byte[] fromCache));
        Console.WriteLine("Same ref? " +
            object.ReferenceEquals(data, fromCache));
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private static void SetOnlyRootToNull()
    {
        data = null;
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    private static void CheckNoLongerAvailable()
```

```
    {
        byte[] fromCache;
        Console.WriteLine("Retrieval: " + cache.TryGetValue("d",
                          out fromCache));
        Console.WriteLine("Null? " + (fromCache == null));
    }
}
```

Все начинается с создания экземпляра моего класса кэша и последующего добавления в кэш ссылки на 100-байтовый массив. Листинг также хранит ссылку на тот же массив в статическом поле `data`, сохраняя его достижимым, пока код не вызовет `SetOnlyRootToNull`, который устанавливает его значение равным `null`. В этом примере мы пытаемся извлечь значение из кэша сразу после его добавления, а также использовать `object.ReferenceEquals` для простой проверки того, что возвращаемое значение действительно ссылается на тот же объект, который мы туда поместили. Затем я запускаю сборку мусора и пытаюсь проделать это снова. (Такой вид искусственного тестового кода является одной из немногих ситуаций, в которых потребуется это делать, — подробности см. в разделе «Принудительная сборка мусора» на с. 410.) Поскольку поле `data` по-прежнему содержит ссылку на массив, то он все еще доступен, поэтому мы ожидаем, что значение все еще будет доступно для получения из кэша. Затем я присваиваю `data` значение `null`, поэтому мой код перестает быть ответственным за достижимость массива. Единственная оставшаяся ссылка — это слабая ссылка, поэтому, когда я запускаю другую сборку мусора, я ожидаю, что массив будет освобожден и последний поиск в кэше завершится неудачей. Чтобы убедиться в этом, я проверяю как возвращаемое значение, ожидая `false`, так и значение, возвращаемое через параметр `out`, который должен быть `null`. И это именно то, что происходит, когда я запускаю программу:

```
Retrieval: True
Same ref? True
Retrieval: True
Same ref? True
Retrieval: False
Null? True
```

Позже я опишу финализацию, которая усложняет ситуацию и создает пограничное состояние, в котором объект уже был отмечен как недоступный, но еще не исчез. Объекты, находящиеся в таком состоянии, как правило, бесполезны, поэтому по умолчанию слабая ссылка будет считать ожидающие финализации объекты уже удаленными. Это называется короткой

слабой ссылкой. Если по какой-то причине вам нужно узнать, действительно ли объект исчез (а не всего лишь ожидает этого), то конструктор класса `WeakReference<T>` имеет перегрузки, часть из которых умеют создавать длинную слабую ссылку, обеспечивающую доступ к объекту даже в этом промежутке между недоступностью и окончательным удалением.

Написание кода, иллюстрирующего поведение сборщика мусора, — это шаг на кривую дорожку. Принципы работы остаются прежними, но точное поведение небольших примеров с течением времени меняется. (Мне пришлось изменить некоторые примеры из предыдущих изданий книги.) Вполне возможно, что если вы запустите эти примеры, то увидите другое поведение из-за изменений в среде выполнения после выхода книги в печать.

Освобождение памяти

До сих пор я описывал лишь то, как CLR определяет, какие объекты больше не используются, но не то, что происходит после этого. Выявив мусор, среда выполнения должна его убрать. Для маленьких и больших объектов CLR использует разные стратегии. (По умолчанию большой объект — это тот, размер которого превышает 85 тысяч байт.) Большинство выделений памяти обслуживают небольшие объекты, поэтому сначала я расскажу о них.

CLR пытается сохранить непрерывность свободного пространства кучи. Очевидно, что это легко, когда приложение запускается впервые, потому что нет ничего, кроме свободного места, и поддерживать непрерывность можно достаточно легко, выделяя память для каждого нового объекта непосредственно после последнего. Но после первой сборки мусора куча уже вряд ли будет выглядеть так аккуратно. Большинство объектов имеют короткий срок жизни и становятся недостижимыми между двумя последовательными сборками мусора. Однако некоторые из них все еще могут использоваться. Время от времени приложения создают объекты, которые задерживаются дольше. Какая бы работа ни происходила во время сборки мусора, в ней, вероятно, будут использоваться какие-то объекты, так что последние выделенные блоки кучи, скорее всего, все еще будут использоваться. Это означает, что конец кучи может выглядеть примерно так, как показано на рис. 7.1, где серые прямоугольники являются достижимыми блоками, а белые показывают блоки, которые больше не используются.