Оглавление

Настройка адаптивного и статического определения размеров кучи

Настройка параллельного уборщика мусора сводится к времени паузы и соблюдению баланса между общим размером кучи и размерами старого/молодого поколения.

При этом приходится учитывать два компромиссных решения. Во-первых, действует классический компромисс между затратами времени и затратами памяти. Большая куча потребляет больше памяти на машине, но благодаря потреблению этой памяти (по крайней мере до определенной степени) приложение начинает работать более эффективно.

Второй компромисс относится к времени выполнения уборки мусора. Число полных пауз можно сократить, увеличивая размер кучи, но это может негативно отразиться на среднем времени отклика, потому что уборка мусора занимает больше времени. Аналогичным образом паузы полной уборки мусора можно сократить, выделяя для нового поколения бóльшую часть кучи, чем для старого, но в свою очередь это приведет к повышению частоты уборки мусора в старом поколении.

Последствия этих компромиссов представлены на рис. 6.3. На графике изображена максимальная производительность REST-сервера при разных размерах кучи. При малом размере кучи — 256 Мбайт — сервер проводит значительное время за уборкой мусора (36% общего времени); в результате производительность ограничивается. С увеличением размера кучи производительность быстро возрастает — пока размер кучи не достигнет 1500 Мбайт. После этого скорость роста производительности замедляется: приложение на этой стадии уже не привязано к уборке мусора (за которой проводится всего 6%). Начинает действовать закон убывающей отдачи: приложение может использовать дополнительную память для повышения производительности, но выигрыш становится менее заметным.

После достижения порога 4500 Мбайт производительность начинает слегка убывать. На этой стадии приложение достигло второй балансной точки: дополнительная память удлиняет циклы уборки мусора, а более долгие циклы — несмотря на их меньшую частоту — могут сократить общую производительность.

Данные на графике были получены при отключении адаптивного определения размеров в JVM; минимальному и максимальному размеру кучи были присвоены одинаковые значения. Вы можете провести эксперименты в любом приложении и определить оптимальные размеры для кучи и поколений, но чаще бывает проще поручить эти решения JVM (что обычно и происходит, так как адаптивное определение размеров включено по умолчанию).

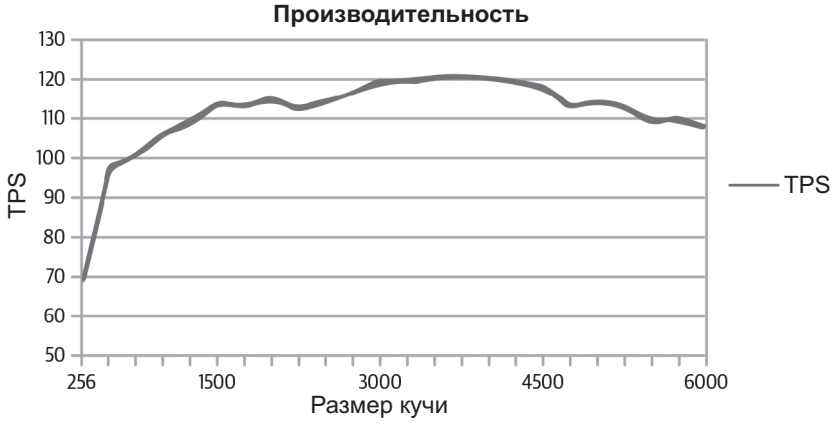

Рис. 6.3. Производительность при разных размерах кучи

Адаптивное определение размеров в параллельном уборщике изменяет размер кучи (и поколений) для соблюдения целей по продолжительности пауз. Эти цели задаются следующими флагами: `-XX:MaxGCPauseMillis=`*N* и `-XX:GCTimeRatio=`*N*.

Флаг `MaxGCPauseMillis` задает максимальную продолжительность паузы, приемлемую для приложения. Появляется искушение присвоить ему значение 0 или какую-нибудь малую величину (скажем, 50 мс). Учтите, что эта цель относится как к малой, так и к полной уборке мусора. При выборе очень малого значения у приложения будет очень маленькое старое поколение: такое, которое может быть очищено за 50 мс. В результате JVM будет очень, очень часто проводить полную уборку мусора, и производительность будет просто катастрофической. Будьте реалистом: используйте величину, которая может быть достигнута на практике. По умолчанию флаг не устанавливается.

Флаг `GCTimeRatio` задает время, которое приложение может проводить за уборкой мусора (относительно времени выполнения потоков приложения). Это соотношение, так что над значением *N* стоит немного поразмыслить. Оно используется в следующей формуле для определения процента времени, в течение которого (в идеале) должны выполняться потоки приложения:

$$ЦельПоПроизводительности = 1 - \frac{1}{(1 + GCTimeRatio)}.$$

Значение `GCTimeRatio` по умолчанию равно 99. Подставляя это значение в формулу, мы получаем 0,99 — это означает, что в соответствии с целью 99% времени приложение должно проводить за своими вычислениями, только 1% — за уборкой мусора. Но вас не должно сбивать с толку сочетание чисел в этом конкретном

случае. Если значение `GCTimeRatio` равно 95, это не означает, что уборка мусора должна занимать до 5% времени — на нее должно расходоваться до 1,94% времени.

Проще сначала определить минимальный процент времени, в течение которого приложение должно выполнять свою работу (скажем, 95%), а затем вычислить значение `GCTimeRatio` по следующей формуле:

$$GCTimeRatio = \frac{Производительность}{(1 - Производительность)}.$$

Если цель по производительности составляет 95% (0,95), то согласно формуле значение `GCTimeRatio` равно 19.

JVM использует эти два флага для задания размера кучи в границах, установленных исходным (`-Xms`) и максимальным (`-Xmx`) размером кучи. Флаг `MaxGCPauseMillis` обладает более высоким приоритетом: если он установлен, то размеры молодого и старого поколения регулируются до тех пор, пока не будет выполнена цель по продолжительности пауз. Когда это произойдет, общий размер кучи увеличивается до выполнения цели по соотношению времени. После достижения обеих целей JVM пытается сократить размер кучи для достижения наименьшего возможного размера, удовлетворяющего обеим целям.

Так как цель по продолжительности пауз по умолчанию не устанавливается, автоматическое определение размера кучи обычно приводит к тому, что размер кучи (и поколений) увеличивается до достижения цели `GCTimeRatio`. Однако на практике значение этого флага по умолчанию оказывается слишком оптимистичным. Конечно, ваш опыт может быть другим, но мне обычно попадались приложения, которые проводят за уборкой мусора от 3% до 6% и при этом работают вполне прилично. Иногда мне приходится работать над приложениями в средах с серьезной нехваткой памяти; такие приложения проводят от 10 до 15% времени за уборкой мусора. Уборка мусора серьезно влияет на производительность таких приложений, но общие цели производительности при этом выполняются. Таким образом, оптимальные настройки изменяются в зависимости от целей приложений. При отсутствии других целей я начинаю с соотношения 19 (5% времени расходуется на уборку мусора).

В табл. 6.1 перечислены эффекты динамической настройки в приложении с небольшим размером кучи и незначительной уборкой мусора (на примере REST-сервера с акциями, использующего малое количество объектов с долгим сроком жизни).

По умолчанию минимальный размер кучи составляет 64 Мбайт, а максимальный — 2 Гбайт (так как машина оснащена 8 Гбайт физической памяти). В этом случае `GCTimeRatio` работает так, как ожидалось: размер кучи динамически меняется до 649 Мбайт — точки, в которой приложение проводит около 1% общего времени за уборкой мусора.

Таблица 6.1. Эффект динамической настройки уборки мусора

Параметры уборки мусора	Конечный размер кучи	Процент времени на уборку мусора	OPS
По умолчанию	649 Мбайт	0,9%	9,2
`MaxGCPauseMillis=50ms`	560 Мбайт	1,0%	9,2
`Xms=Xmx=2048m`	2 Гбайт	0,04%	9,2

Настройка флага `MaxGCPauseMillis` в данном случае начинает сокращать размер кучи для соблюдения цели по продолжительности пауз. Так как объем работы у уборщика мусора в этом примере невелик, приложение проводит всего 1% общего времени за уборкой мусора при сохранении той же производительности 9,2 OPS.

Наконец, следует заметить, что больше — не всегда значит лучше. Полная куча на 2 Гбайт означает, что приложение сможет тратить меньше времени за уборкой мусора, но в данном случае уборка мусора не является доминирующим фактором производительности, так что производительность не возрастает. Как обычно, время, потраченное на оптимизацию не той части приложения, не окупается.

Если изменить то же приложение так, чтобы последние 50 запросов каждого пользователя сохранялись в глобальном кэше (например, в кэше JPA), уборщику мусора придется выполнять больше работы. В табл. 6.2 представлены некоторые соотношения для такой ситуации.

Таблица 6.2. Влияние заполнения кучи на динамическую настройку уборки мусора

Параметры уборки мусора	Конечный размер кучи	Процент времени на уборку мусора	OPS
По умолчанию	1,7 Гбайт	9,3%	8,4
`MaxGCPauseMillis=50ms`	588 Мбайт	15,1%	7,9
`Xms=Xmx=2048m`	2 Гбайт	5,1%	9,0
`Xmx=3560M; MaxGCRatio=19`	2,1 Гбайт	8,8%	9,0

В тесте, который проводит значительное время за уборкой мусора, поведение уборки мусора отличается. JVM никогда не сможет достичь цели по производительности 1% в этом тесте; она пытается по возможности приблизиться к цели по умолчанию и неплохо справляется со своей задачей, используя 1,7 Гбайт памяти.

Нереалистичная цель по продолжительности паузы ухудшает поведение приложения. Чтобы достичь времени уборки мусора в 50 мс, куча удерживается на уровне 588 Мбайт, но это означает, что уборки мусора становятся слишком частыми. Как следствие, производительность значительно ухудшается. В этом сценарии для достижения лучшей производительности следует приказать JVM использовать всю кучу, установив исходный и максимальный размеры равными 2 Гбайт.

Наконец, последняя строка таблицы показывает, что происходит, если для кучи определены разумные размеры, а для цели по продолжительности пауз выбран реалистичный уровень 5%. Сама JVM определила, что оптимальный размер кучи составляет приблизительно 2 Гбайт, и достигла такой же производительности, как в варианте с ручной настройкой.

РЕЗЮМЕ

- Динамическая настройка кучи — первый шаг определения ее размеров. В широком спектре применений ничего больше не понадобится, а динамические настройки минимизируют потребление памяти JVM.
- Статическая настройка размера кучи позволяет добиться максимальной возможной производительности. Размеры, определяемые JVM для разумного набора целей производительности, станут хорошей отправной точкой для такой настройки.

Уборщик мусора G1

Уборщик мусора G1 работает с отдельными областями кучи. Каждая область (по умолчанию их около 2048) может принадлежать старому или новому поколению, и области поколений не обязаны занимать смежную область в памяти. Идея определения областей в старом поколении основана на том, что когда конкурентно работающие фоновые потоки ищут объекты, на которые нет ссылок, некоторые области будут содержать больше мусора, чем другие. Фактическая уборка мусора все равно требует остановки потоков приложений, но алгоритм G1 может сосредоточиться на областях, которые в основном состоят из мусора, и проводить лишь незначительное время за очисткой этих областей. Именно от такого подхода — очистки только областей, занятых в основном мусором, — происходит название алгоритма G1: «Garbage First», то есть «сначала мусор».

Это не относится к областям молодого поколения: в ходе уборки мусора в молодом поколении все молодое поколение либо освобождается, либо повышается (переводится в область выживших или в старое поколение). При этом молодое

поколение определяется в областях — отчасти из-за того, что заранее определенные области существенно упрощают изменение размеров поколений.

Алгоритм G1 называется *конкурентным*, потому что пометка свободных объектов в старом поколении происходит конкурентно с работой потоков приложения (то есть они продолжают работать). Тем не менее речь не идет о полной конкурентности, потому что пометка и сжатие молодого поколения требуют остановки всех потоков приложения, а сжатие старого поколения также происходит во время остановки потоков приложения.

Уборка мусора G1 состоит из четырех логических операций:

- Уборка в молодом поколении.
- Фоновый конкурентный цикл пометки.
- Смешанная уборка.
- При необходимости — полная уборка мусора.

Рассмотрим все эти операции поочередно, начиная с уборки G1 в молодом поколении (рис. 6.4).

Каждый квадратик на диаграмме представляет область уборки мусора G1. Данные в каждой области представляются черным цветом, а буква обозначает поколение, к которому принадлежит область: [E] — Эдем (Eden), [O] — старое поколение (Old generation), [S] — область выживших (Survivor space). Пустые области не принадлежат поколениям; алгоритм G1 произвольно использует их для того поколения, которое сочтет нужным.

Рис. 6.4. Уборка мусора G1 в молодом поколении

Уборка G1 в молодом поколении инициируется при заполнении Эдема (в данном примере после заполнения четырех областей). После уборки мусора Эдем

остается пустым (хотя для него выделены области, которые начнут заполняться данными при дальнейшей работе приложения). По крайней мере одна область выделена под область выживших (частично заполненная в данном примере), и часть данных была перемещена в старое поколение.

В журнале процесса данные уборки мусора G1 несколько отличаются от других уборщиков. Пример для JDK 8 был получен при помощи `PrintGCDetails`, но для G1 выводится более подробная информация. В приведенных примерах представлены лишь самые важные строки.

Стандартная уборка мусора в молодом поколении:

```
23.430: [GC pause (young), 0.23094400 secs]
...
   [Eden: 1286M(1286M)->0B(1212M)
    Survivors: 78M->152M Heap: 1454M(4096M)->242M(4096M)]
   [Times: user=0.85 sys=0.05, real=0.23 secs]
```

Уборка в молодом поколении заняла 0,23 секунды реального времени, за которые потоки уборки мусора потребили 0,85 секунды процессорного времени. 1286 Мбайт объектов были перемещены из Эдема (которому был адаптивно назначен размер 1212 Мбайт); 74 Мбайт из них были перемещены в область выживших (она увеличилась в размерах с 78 Мбайт до 152 Мбайт), а остальные были освобождены. Мы знаем, что они были освобождены, наблюдая за тем, что общее заполнение кучи снизилось до 1212 Мбайт. В общем случае некоторые объекты из области выживших могли быть перемещены в старое поколение, а если область выживших была переполнена, то некоторые объекты из Эдема могли быть переведены непосредственно в старое поколение — в этих случаях размер старого поколения увеличится.

В JDK 11 аналогичный журнал выглядит так:

```
[23.200s][info   ][gc,start     ] GC(10) Pause Young (Normal)
                                           (G1 Evacuation Pause)
[23.200s][info   ][gc,task      ] GC(10) Using 4 workers of 4 for evacuation
[23.430s][info   ][gc,phases    ] GC(10)   Pre Evacuate Collection Set: 0.0ms
[23.430s][info   ][gc,phases    ] GC(10)   Evacuate Collection Set: 230.3ms
[23.430s][info   ][gc,phases    ] GC(10)   Post Evacuate Collection Set: 0.5ms
[23.430s][info   ][gc,phases    ] GC(10)   Other: 0.1ms
[23.430s][info   ][gc,heap      ] GC(10) Eden regions: 643->606(606)
[23.430s][info   ][gc,heap      ] GC(10) Survivor regions: 39->76(76)
[23.430s][info   ][gc,heap      ] GC(10) Old regions: 67->75
[23.430s][info   ][gc,heap      ] GC(10) Humongous regions: 0->0
[23.430s][info   ][gc,metaspace ] GC(10) Metaspace: 18407K->18407K(1067008K)
[23.430s][info   ][gc           ] GC(10) Pause Young (Normal)
                                           (G1 Evacuation Pause)
                                           1454M(4096M)->242M(4096M) 230.104ms
[23.430s][info   ][gc,cpu       ] GC(10) User=0.85s Sys=0.05s Real=0.23s
```

Конкурентный цикл уборки мусора G1 начинается и завершается так, как показано на рис. 6.5.

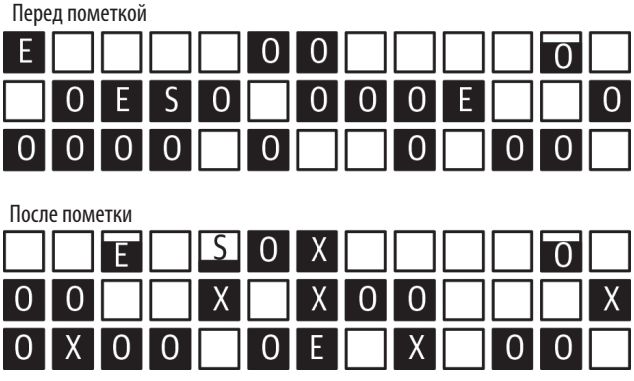

Рис. 6.5. Конкурентная уборка мусора G1

На диаграмме представлены три важных момента, на которые следует обратить внимание. Во-первых, молодое поколение изменило заполнение; в конкурентном цикле будет по крайней мере одна (а возможно, и более) уборка мусора в молодом поколении. А следовательно, области Эдема перед циклом пометки были полностью освобождены и началось выделение новых областей Эдема.

Во-вторых, некоторые области теперь снабжены пометкой X. Эти области принадлежат старому поколению (обратите внимание: они все еще содержат данные) — области, которые, как было определено в ходе пометки, состоят в основном из мусора.

Наконец, заметим, что старое поколение (состоящее из областей с пометкой O или X) в действительности обладает большим заполнением после завершения цикла. Это связано с тем, что уборки в молодом поколении, происходящие в цикле пометки, переместили данные в старое поколение. Кроме того, цикл пометки не освобождает никаких данных в старом поколении; он просто идентифицирует области, которые состоят в основном из мусора. Данные из этих областей будут освобождены в более позднем цикле.

Конкурентный цикл уборки мусора G1 состоит из нескольких фаз. В одних фазах все потоки приложения приостанавливаются, в других нет. Первая фаза называется *исходной пометкой* (в JDK 8) или *запуском конкурентной обработки* (в JDK 11). В этой фазе останавливаются все потоки приложений — отчасти потому, что в ней также выполняется уборка мусора в молодом поколении и она подготавливает следующие фазы цикла.

В JDK 8 это выглядит так:

```
50.541: [GC pause (G1 Evacuation pause) (young) (initial-mark), 0.27767100 secs]
    ... много других данных ...
    [Eden: 1220M(1220M)->0B(1220M)
     Survivors: 144M->144M Heap: 3242M(4096M)->2093M(4096M)]
    [Times: user=1.02 sys=0.04, real=0.28 secs]
```

То же в JDK 11:

```
[50.261s][info   ][gc,start      ] GC(11) Pause Young (Concurrent Start)
                                          (G1 Evacuation Pause)
[50.261s][info   ][gc,task       ] GC(11) Using 4 workers of 4 for evacuation
[50.541s][info   ][gc,phases     ] GC(11)   Pre Evacuate Collection Set: 0.1ms
[50.541s][info   ][gc,phases     ] GC(11)   Evacuate Collection Set: 25.9ms
[50.541s][info   ][gc,phases     ] GC(11)   Post Evacuate Collection Set: 1.7ms
[50.541s][info   ][gc,phases     ] GC(11)   Other: 0.2ms
[50.541s][info   ][gc,heap       ] GC(11) Eden regions: 1220->0(1220)
[50.541s][info   ][gc,heap       ] GC(11) Survivor regions: 144->144(144)
[50.541s][info   ][gc,heap       ] GC(11) Old regions: 1875->1946
[50.541s][info   ][gc,heap       ] GC(11) Humongous regions: 3->3
[50.541s][info   ][gc,metaspace  ] GC(11) Metaspace: 52261K->52261K(1099776K)
[50.541s][info   ][gc            ] GC(11) Pause Young (Concurrent Start)
                                          (G1 Evacuation Pause)
                                          1220M->0B(1220M) 280.055ms
[50.541s][info   ][gc,cpu        ] GC(11) User=1.02s Sys=0.04s Real=0.28s
```

Как и при обычной уборке мусора в молодом поколении, потоки приложения были остановлены (на 0,28 секунды), а молодое поколение было очищено (так что Эдем начинается с размера 0). 71 Мбайт данных был перемещен из молодого поколения в старое. В JDK 8 понять это не так просто (приходится вычислять результат 2093 – 3242 + 1220); в выводе JDK 11 этот факт лучше виден.

С другой стороны, в выводе JDK 11 присутствует ряд моментов, которые еще не упоминались. Во-первых, размеры областей задаются в областях, а не в мегабайтах. О размерах мы еще поговорим в этой главе, но в этом примере размер одной области составляет 1 Мбайт. Кроме того, в выводе JDK 11 упоминается новое пространство: *огромные* (humongous) области. Они являются частью старого поколения и также будут более подробно рассмотрены в этой главе.

Сообщение исходной пометки (или запуска конкурентной обработки) указывает, что цикл фоновой конкурентной обработки начался. Так как исходная пометка в фазе цикла пометки также требует остановки всех потоков приложения, уборщик G1 пользуется циклом уборки в молодом поколении для выполнения этой работы. Последствия от добавления фазы исходной пометки в уборку мусора в молодом поколении не столь значительны: она увеличивает затраты процессорного времени на 20% по сравнению с предыдущей уборкой (то есть обычной уборкой в молодом поколении), хотя пауза увеличилась совсем незначительно.

(К счастью, на машине хватало свободных ресурсов процессора для параллельных потоков G1, иначе пауза получилась бы более продолжительной.)

Затем уборщик G1 сканирует корневую область:

```
50.819: [GC concurrent-root-region-scan-start]
51.408: [GC concurrent-root-region-scan-end, 0.5890230]

[50.819s][info ][gc             ] GC(20) Concurrent Cycle
[50.819s][info ][gc,marking     ] GC(20) Concurrent Clear Claimed Marks
[50.828s][info ][gc,marking     ] GC(20) Concurrent Clear Claimed Marks 0.008ms
[50.828s][info ][gc,marking     ] GC(20) Concurrent Scan Root Regions
[51.408s][info ][gc,marking     ] GC(20) Concurrent Scan Root Regions 589.023ms
```

Процесс занимает 0,58 секунды, но он не останавливает потоки приложения; используются только фоновые потоки. Тем не менее эта фаза не может быть прервана уборкой мусора в молодом поколении, поэтому наличие доступных ресурсов процессора для этих фоновых потоков критично. Если молодое поколение окажется заполненным во время сканирования корневой области, уборка в молодом поколении (которая останавливает все потоки приложения) должна дождаться завершения сканирования корневой области. По сути это означает, что для уборки в молодом поколении потребуется пауза большей длительности, чем обычно. Эта ситуация представлена в журнале уборки мусора так:

```
350.994: [GC pause (young)
  351.093: [GC concurrent-root-region-scan-end, 0.6100090]
  351.093: [GC concurrent-mark-start],
  0.37559600 secs]

[350.384s][info][gc,marking   ] GC(50) Concurrent Scan Root Regions
[350.384s][info][gc,marking   ] GC(50) Concurrent Scan Root Regions 610.364ms
[350.994s][info][gc,marking   ] GC(50) Concurrent Mark (350.994s)
[350.994s][info][gc,marking   ] GC(50) Concurrent Mark From Roots
[350.994s][info][gc,task      ] GC(50) Using 1 workers of 1 for marking
[350.994s][info][gc,start     ] GC(51) Pause Young (Normal) (G1 Evacuation
                                       Pause)
```

Пауза уборки мусора начинается до конца сканирования корневой области. В JDK 8 чередование вывода в журнале указывает на то, что уборке в молодом поколении пришлось сделать паузу для завершения сканирования корневой области, прежде чем она смогла продолжить работу. В JDK 11 обнаружить этот факт немного сложнее: для этого нужно заметить, что временная метка конца сканирования корневой области в точности совпадает с меткой начала следующей уборки мусора в молодом поколении.

В любом случае невозможно точно определить, на какое время была отложена уборка в молодом поколении. Величина задержки не обязательно составила все 610 мс в данном примере; какое-то время (до того, как было заполнено моло-

дое поколение) работа продолжалась. Но в этом случае по временным меткам видно, что потоки приложения ожидали лишние 100 мс — вот почему продолжительность паузы при уборке молодого поколения приблизительно на 100 мс больше средней продолжительности других пауз в этом журнале. (Если такое происходит часто, это признак того, что уборку мусора G1 необходимо лучше настроить, как описано в следующем разделе.)

После сканирования корневой области уборщик G1 входит в фазу конкурентной пометки. Это происходит полностью в фоновом режиме; в начале и в конце выводятся сообщения:

```
111.382: [GC concurrent-mark-start]
....
120.905: [GC concurrent-mark-end, 9.5225160 sec]
```

```
[111.382s][info][gc,marking   ] GC(20) Concurrent Mark (111.382s)
[111.382s][info][gc,marking   ] GC(20) Concurrent Mark From Roots
...
[120.905s][info][gc,marking   ] GC(20) Concurrent Mark From Roots 9521.994ms
[120.910s][info][gc,marking   ] GC(20) Concurrent Preclean
[120.910s][info][gc,marking   ] GC(20) Concurrent Preclean 0.522ms
[120.910s][info][gc,marking   ] GC(20) Concurrent Mark (111.382s, 120.910s)
                                       9522.516ms
```

Конкурентная пометка может быть прервана, так что уборки в молодом поколении могут происходить в этой фазе (на месте многоточий будет длинный вывод уборки мусора).

Также обратите внимание на то, что в примере JDK 11 вывод включает тот же идентификатор уборки мусора (20), как у записи, в которой произошло сканирование корневой области. Операция разбивается с большей детализацией, чем система ведения журнала JDK: в JDK все фоновое сканирование считается одной операцией. Мы разбиваем обсуждение на более мелкие логические операции, потому что, например, сканирование корневой области может создать паузу, тогда как конкурентная пометка — нет.

За фазой пометки следует фаза повторной пометки и нормальная фаза очистки:

```
120.910: [GC remark 120.959:
        [GC ref-PRC, 0.0000890 secs], 0.0718990 secs]
  [Times: user=0.23 sys=0.01, real=0.08 secs]
120.985: [GC cleanup 3510M->3434M(4096M), 0.0111040 secs]
  [Times: user=0.04 sys=0.00, real=0.01 secs]
```

```
[120.909s][info][gc,start     ] GC(20) Pause Remark
[120.909s][info][gc,stringtable] GC(20) Cleaned string and symbol table,
                                        strings: 1369 processed, 0 removed,
                                        symbols: 17173 processed, 0 removed
[120.985s][info][gc            ] GC(20) Pause Remark 2283M->862M(3666M) 80.412ms
[120.985s][info][gc,cpu        ] GC(20) User=0.23s Sys=0.01s Real=0.08s
```

В этих фазах потоки приложений останавливаются, хотя обычно на непродолжительное время. Затем конкурентно происходит дополнительная фаза очистки:

```
120.996: [GC concurrent-cleanup-start]
120.996: [GC concurrent-cleanup-end, 0.0004520]

[120.878s][info][gc,start      ] GC(20) Pause Cleanup
[120.879s][info][gc            ] GC(20) Pause Cleanup 1313M->1313M(3666M) 1.192ms
[120.879s][info][gc,cpu        ] GC(20) User=0.00s Sys=0.00s Real=0.00s
[120.879s][info][gc,marking    ] GC(20) Concurrent Cleanup for Next Mark
[120.996s][info][gc,marking    ] GC(20) Concurrent Cleanup for Next Mark
                                        117.168ms
[120.996s][info][gc            ] GC(20) Concurrent Cycle 70,177.506ms
```

В этом случае нормальный фоновый цикл пометки G1 завершен — по крайней мере в том, что касается уборки мусора. Однако пока еще почти ничего не было реально освобождено. Небольшой объем памяти был освобожден в фазе очистки, но к настоящему моменту уборщик мусора G1 всего лишь выявил старые области, которые состоят в основном из мусора и могут быть освобождены (на рис. 6.5 они помечены знаком X).

Теперь уборщик мусора G1 выполняет серию смешанных уборок мусора. Они называются «смешанными» (mixed), потому что выполняется нормальная уборка мусора в молодом поколении, но также в ней задействованы некоторые помеченные области из фонового сканирования. Эффект смешанной уборки мусора показан на рис. 6.6.

Рис. 6.6. Смешанная уборка, выполняемая уборщиком мусора G1

Как обычно в молодом поколении, уборка мусора G1 полностью опустошила Эдем и отрегулировала области выживших. Кроме того, была проведена уборка в двух помеченных регионах. Было заранее известно, что эти области в основ-

ном содержат мусор, так что большая их часть была освобождена. Все «живые» данные в этих областях были перемещены в другую область (по аналогии с тем, как «живые» данные перемещались из молодого поколения в области старого поколения). Так уборка мусора G1 производит сжатие старого поколения — такое перемещение объектов фактически сжимает кучу по ходу уборки мусора G1.

Смешанная операция уборки мусора обычно выглядит в журнале примерно так:

```
79.826: [GC pause (mixed), 0.26161600 secs]
....
   [Eden: 1222M(1222M)->0B(1220M)
    Survivors: 142M->144M Heap: 3200M(4096M)->1964M(4096M)]
   [Times: user=1.01 sys=0.00, real=0.26 secs]

[3.800s][info][gc,start      ] GC(24) Pause Young (Mixed) (G1 Evacuation Pause)
[3.800s][info][gc,task       ] GC(24) Using 4 workers of 4 for evacuation
[3.800s][info][gc,phases     ] GC(24)   Pre Evacuate Collection Set: 0.2ms
[3.825s][info][gc,phases     ] GC(24)   Evacuate Collection Set: 250.3ms
[3.826s][info][gc,phases     ] GC(24)   Post Evacuate Collection Set: 0.3ms
[3.826s][info][gc,phases     ] GC(24)   Other: 0.4ms
[3.826s][info][gc,heap       ] GC(24) Eden regions: 1222->0(1220)
[3.826s][info][gc,heap       ] GC(24) Survivor regions: 142->144(144)
[3.826s][info][gc,heap       ] GC(24) Old regions: 1834->1820
[3.826s][info][gc,heap       ] GC(24) Humongous regions: 4->4
[3.826s][info][gc,metaspace  ] GC(24) Metaspace: 3750K->3750K(1056768K)
[3.826s][info][gc            ] GC(24) Pause Young (Mixed) (G1 Evacuation Pause)
                                          3791M->3791M(3983M) 124.390ms
[3.826s][info][gc,cpu        ] GC(24) User=1.01s Sys=0.00s Real=0.26s
[3.826s][info][gc,start      ] GC(25) Pause Young (Mixed) (G1 Evacuation Pause)
```

Следует заметить, что использование кучи сократилось на величину, превышающую 1,222 Мбайт, удаленных из Эдема. Разность (16 Мбайт) кажется небольшой, но я напомню, что часть пространства выживших была конкурентно перемещена в старое поколение; кроме того, каждая смешанная уборка мусора освобождает только часть целевых областей старого поколения.

В дальнейшем вы увидите, что очень важно позаботиться о том, что смешанные уборки мусора очищают достаточно памяти для предотвращения будущих конфликтов конкурентного доступа.

В JDK 11 первая смешанная уборка мусора снабжается пометкой `Prepared Mixed` и немедленно следует за конкурентной очисткой.

Смешанные циклы уборки мусора продолжаются до тех пор, пока (почти) во всех помеченных областях не будет проведена уборка мусора; в этой точке уборщик G1 возобновляет обычные циклы уборки молодого поколения. Со временем уборщик G1 запустит еще один конкурентный цикл для определения того, какие области в старом поколении должны быть освобождены следующими.

Хотя для смешанных циклов уборки мусора обычно указывается причина (`Mixed`), уборки в молодом поколении иногда помечаются обычным образом по схеме конкурентного цикла (например, `G1 Evacuation Pause`). Если конкурентный цикл найдет в старом поколении области, которые могут быть полностью освобождены, эти области освобождаются в ходе обычной паузы перемещения молодого поколения. С технической точки зрения это не является смешанным циклом в реализации уборщика мусора. Однако с логической точки зрения это так и есть: объекты освобождаются из молодого поколения или перемещаются в старое поколение, и в то же время мусорные объекты (вернее, области) освобождаются из старого поколения.

Если все проходит нормально, то это весь набор операций уборки мусора, которые будут присутствовать в журнале. Но нужно рассматривать некоторые аномальные ситуации.

Иногда в журнале отображается полная уборка мусора, это говорит о том, что дополнительная настройка (возможно, с включением дополнительной памяти кучи) положительно скажется на производительности приложения. Она инициируется в четырех случаях:

- *Сбой конкурентного режима* — уборщик мусора G1 запускает цикл пометки, но старое поколение заполняется до завершения цикла. В этом случае уборщик G1 отменяет цикл пометки:

```
51.408: [GC concurrent-mark-start]
65.473: [Full GC 4095M->1395M(4096M), 6.1963770 secs]
 [Times: user=7.87 sys=0.00, real=6.20 secs]
71.669: [GC concurrent-mark-abort]
```

```
[51.408][info][gc,marking     ] GC(30) Concurrent Mark From Roots
...
[65.473][info][gc             ] GC(32) Pause Full (G1 Evacuation Pause)
                                       4095M->1305M(4096M) 60,196.377
...
[71.669s][info][gc,marking     ] GC(30) Concurrent Mark From Roots 191ms
[71.669s][info][gc,marking     ] GC(30) Concurrent Mark Abort
```

  Этот сбой означает, что размер кучи следует увеличить, что фоновая обработка G1 должна начаться быстрее или что цикл необходимо настроить для ускорения выполнения (например, за счет добавления фоновых потоков). Ниже рассказано о том, как это сделать.

- *Сбой повышения* — уборщик мусора G1 завершил цикл пометки и начал выполнение смешанных уборок мусора для очистки старых областей. Прежде чем он смог очистить достаточно места, слишком большое количество объектов было переведено из молодого поколения, и в старом поколении все равно не хватает места. В журнале за смешанной уборкой мусора немедленно следует полная уборка:

```
2226.224: [GC pause (mixed)
        2226.440: [SoftReference, 0 refs, 0.0000060 secs]
        2226.441: [WeakReference, 0 refs, 0.0000020 secs]
        2226.441: [FinalReference, 0 refs, 0.0000010 secs]
        2226.441: [PhantomReference, 0 refs, 0.0000010 secs]
        2226.441: [JNI Weak Reference, 0.0000030 secs]
                (to-space exhausted), 0.2390040 secs]
....
    [Eden: 0.0B(400.0M)->0.0B(400.0M)
     Survivors: 0.0B->0.0B Heap: 2006.4M(2048.0M)->2006.4M(2048.0M)]
    [Times: user=1.70 sys=0.04, real=0.26 secs]
2226.510: [Full GC (Allocation Failure)
        2227.519: [SoftReference, 4329 refs, 0.0005520 secs]
        2227.520: [WeakReference, 12646 refs, 0.0010510 secs]
        2227.521: [FinalReference, 7538 refs, 0.0005660 secs]
        2227.521: [PhantomReference, 168 refs, 0.0000120 secs]
        2227.521: [JNI Weak Reference, 0.0000020 secs]
                2006M->907M(2048M), 4.1615450 secs]
    [Times: user=6.76 sys=0.01, real=4.16 secs]
```

```
[2226.224s][info][gc            ] GC(26) Pause Young (Mixed)
                                           (G1 Evacuation Pause)
                                           2048M->2006M(2048M) 26.129ms
...
[2226.510s][info][gc,start      ] GC(27) Pause Full (G1 Evacuation Pause)
```

Этот сбой означает, что смешанные уборки мусора должны выполняться быстрее; каждая уборка мусора в молодом поколении должна обрабатывать больше областей в старом поколении.

- *Сбой эвакуации* — при выполнении уборки в молодом поколении не хватает места для хранения всех выживших объектов в областях выживших и в старом поколении. В журналах уборки мусора это выглядит как особая разновидность уборки мусора в молодом поколении:

```
60.238: [GC pause (young) (to-space overflow), 0.41546900 secs]
```

```
[60.238s][info][gc,start       ] GC(28) Pause Young (Concurrent Start)
                                          (G1 Evacuation Pause)
[60.238s][info][gc,task        ] GC(28) Using 4 workers of 4
                                          for evacuation
[60.238s][info][gc             ] GC(28) To-space exhausted
```

Это признак того, что куча в основном заполнена или фрагментирована. Уборщик G1 пытается компенсировать этот факт, но, скорее всего, это кончится плохо: JVM прибегнет к полной уборке мусора. Проблема проще всего решается повышением размера кучи, хотя возможные решения приводятся в разделе «Расширенная настройка», с. 225.

- *Сбой создания огромных объектов* — приложения, создающие очень большие объекты, могут инициировать другую разновидность полной уборки

мусора в уборщике G1; за дополнительной информацией (в том числе и той, как ее избежать) обращайтесь к разделу «Создание огромных объектов уборщиком мусора G1», с. 236. В JDK 8 диагностировать эту ситуацию невозможно без использования специальных параметров управления журналом, но в JDK 11 эта информация отображается в следующей записи:

```
[3023.091s][info][gc,start     ] GC(54) Pause Full (G1 Humongous Allocation)
```

- *Порог уборки мусора в метаданных* — как я уже упоминал, метапространство фактически представляет собой отдельную кучу, а уборка мусора в нем осуществляется независимо от основной кучи. Оно не чистится уборщиком мусора G1, но когда возникает необходимость в его уборке в JDK 8, уборщик G1 выполняет полную уборку мусора (непосредственно после уборки мусора в молодом поколении) с основной кучей:

```
0.0535: [GC (Metadata GC Threshold) [PSYoungGen: 34113K->20388K(291328K)]
    73838K->60121K(794112K), 0.0282912 secs]
    [Times: user=0.05 sys=0.01, real=0.03 secs]
0.0566: [Full GC (Metadata GC Threshold) [PSYoungGen: 20388K->0K(291328K)]
    [ParOldGen: 39732K->46178K(584192K)] 60121K->46178K(875520K),
    [Metaspace: 59040K->59036K(1101824K)], 0.1121237 secs]
    [Times: user=0.28 sys=0.01, real=0.11 secs]
```

В JDK 11 уборка/изменение размеров метапространства могут выполняться без полной уборки мусора.

РЕЗЮМЕ

- Уборщик мусора G1 использует несколько циклов (и фаз в конкурентном цикле). На хорошо настроенной JVM, на которой работает уборщик G1, должны происходить только циклы уборки в молодом поколении, смешанной и конкурентной уборки мусора.
- В работе алгоритма происходят небольшие паузы для конкурентных фаз G1.
- Уборщика G1 следует настроить так, чтобы по возможности предотвратить циклы полной уборки мусора.

Настройка уборщика мусора G1

Главная цель настройки уборщика G1 — гарантировать, что сбои конкурентного режима или сбоев эвакуации не потребуют полной уборки мусора. Методы, используемые для предотвращения полной уборки мусора, также могут ис-