

КАК СТАТЬ АВТОРОМ



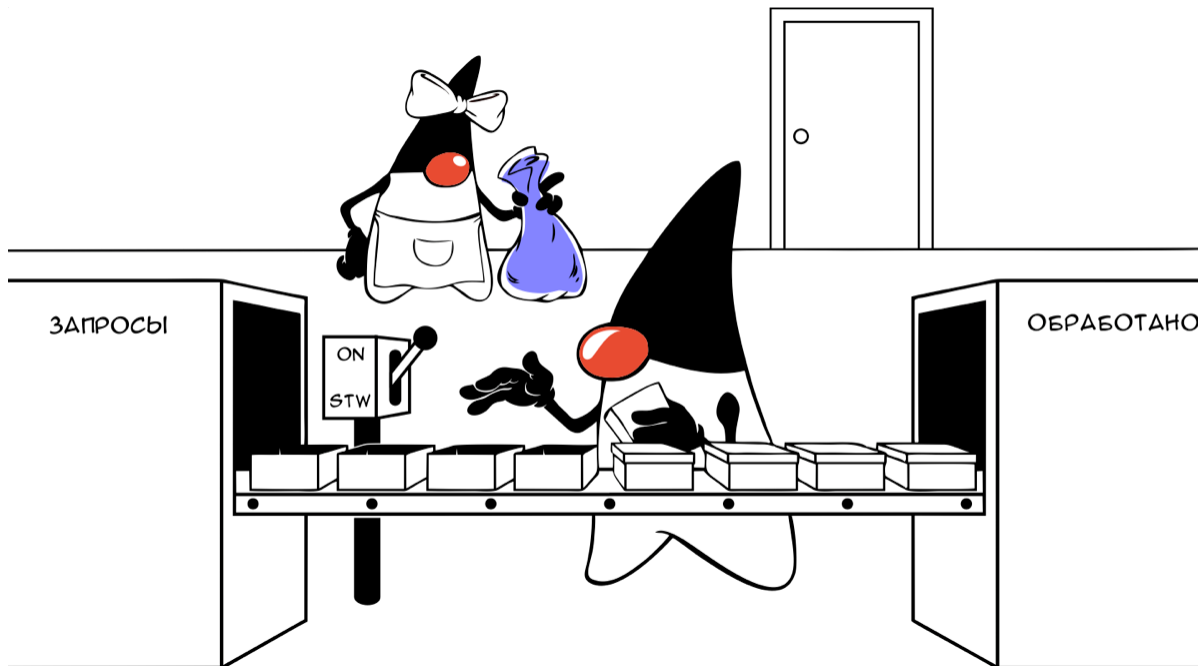
alygin 27 октября 2015 в 15:58

Дюк, вынеси мусор! — 1. Введение

Java*

Из песочницы

Tutorial



Наверняка вы уже читали не один обзор механизмов сборки мусора в Java и настройка таких опций, как *Xmx* и *Xms*, превратилась для вас в обычную рутину. Но действительно ли вы в деталях понимаете, что происходит под капотом вашей виртуальной машины в тот момент, когда приходит время избавиться от ненужных объектов в памяти и ваш идеально оптимизированный метод начинает выполняться в несколько раз дольше положенного? И знаете ли вы, какие возможности предоставляют вам последние версии Java для оптимизации ответственной работы по сборке мусора, зачастую сильно влияющей на производительность вашего приложения?

Попробуем в нескольких статьях пройти путь от описания базовых идей, лежащих в основе всех сборщиков мусора, до разбора алгоритмов работы и возможностей тонкой настройки различных сборщиков Java HotSpot VM (вы ведь знаете, что таких сборщиков четыре?). И самое главное, рассмотрим, каким образом эти знания можно использовать на практике.

Следует сразу оговориться, что все сказанное ниже относится к виртуальной машине HotSpot. Так что если вы встречаете в тексте упоминание JVM, то речь идет именно об этой реализации. Но базовые принципы распространяются и на виртуальные машины других поставщиков, хотя в некоторых деталях они могут отличаться.

А оно мне надо?

Резонный вопрос. Далеко не любой программе для бесперебойной работы требуется тонкая настройка сборщика мусора. Очень часто выделения ей необходимого объема памяти оказывается достаточным. В конце концов, редкий пользователь заметит, что отклик программы время от времени занимает на сотню-другую миллисекунд дольше обычного.

Но возможно, объемы используемой вашей программой памяти таковы, что ее очистка занимает секунды, а то и десятки секунд. Или ваш сервис связан жестким SLA, и вы не можете позволить себе раскидываться десятками миллисекунд направо и налево. Или же любознательность не позволяет вам просто так закрывать глаза на то, что ваша программа что-то делает в своих недрах, а вы не знаете что. В этих случаях давайте разбираться.

Разделяй и властвуй

Прежде чем приступить непосредственно к решению вопросов очистки наших Авгиевых конюшен, давайте разберемся с их общим устройством и определимся, на чем конкретно нам хотелось бы сосредоточиться.

JVM разделяет используемую ею память на две области: *куча (heap)*, в которой хранятся данные приложения, и *не-куча (non-heap)*, в которой хранится код программы и другие вспомогательные данные.

Если ваше приложение при работе самостоятельно не генерирует новые классы и не занимается постоянной подгрузкой / выгрузкой классов, то состояние non-heap в долгосрочной перспективе будет близким к статичному и мало поддающимся оптимизации. В связи с этим, механизмы функционирования области non-heap мы здесь рассматривать не будем, а сосредоточимся на той области, где наши усилия принесут наибольшую выгоду.

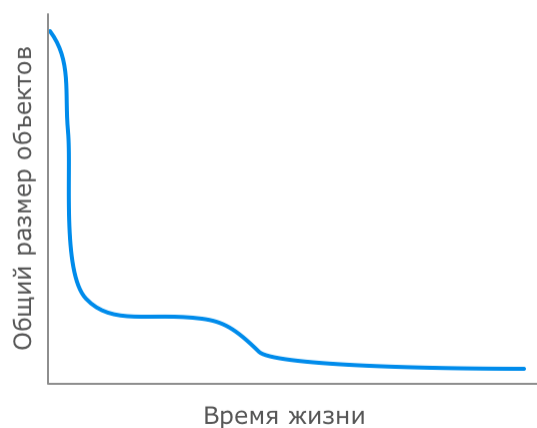
Все объекты, которые явно или неявно создаются Java-приложением, размещаются в куче. Над оптимизацией размещения объектов и алгоритмами их обработки разработчики

языков с автоматической сборкой мусора бьются с первого дня их создания. И как минимум в ближайшем будущем эта битва будет продолжаться, ведь объемы обрабатываемых данных растут, а требования к сборке мусора у различных приложений сильно отличаются, что делает создание единого идеального сборщика не самым тривиальным делом. Наше же дело — следить за развитием ситуации и стараться извлекать из имеющихся инструментов как можно больше пользы.

Из поколения в поколение

Преследуя свои цели (которые могут варьироваться и которые мы обязательно рассмотрим ниже), различные сборщики мусора используют разные подходы к организации памяти и ее очистке, но их объединяет общая черта — все они опираются на *слабую гипотезу о поколениях*. В общем виде, гипотеза о поколениях гласит, что вероятность смерти как функция от возраста снижается очень быстро. Ее приложение к сборке мусора в частности означает, что подавляющее большинство объектов живут крайне недолго. По людским меркам, большинство даже в детский сад не пойдут. Также это означает, что чем дольше прожил объект, тем выше вероятность того, что он будет жить и дальше.

Большинство приложений имеют распределение времен жизни объектов, схематично описываемое примерно такой кривой:



Подавляющее большинство объектов создаются на очень короткое время, они становятся ненужными практически сразу после их первого использования. Итераторы, локальные переменные методов, результаты боксинга и прочие временные объекты, которые зачастую создаются неявно, попадают именно в эту категорию, образуя пик в самом начале графика.

Далее идут объекты, создаваемые для выполнения более-менее долгих вычислений. Их жизнь чуть разнообразнее — они обычно гуляют по различным методам, трансформируясь и обогащаясь в процессе, но после этого становятся ненужными и превращаются в мусор. Благодаря таким объектам возникает небольшой бугорок на графике следом за пиком временных объектов.

И, наконец, объекты-старожилы, переживающие почти всех — это постоянные данные программы, загружаемые часто в самом начале и проживающие долгую и счастливую жизнь до остановки приложения.

Конечно, каждое приложение по-своему уникально, поэтому в каждом конкретном случае этот график будет варьироваться, изменять пропорции, на нем будут появляться аномалии, но чаще всего форма именно такая. Запомните этот график, он нам еще пригодится при выполнении оптимизаций.

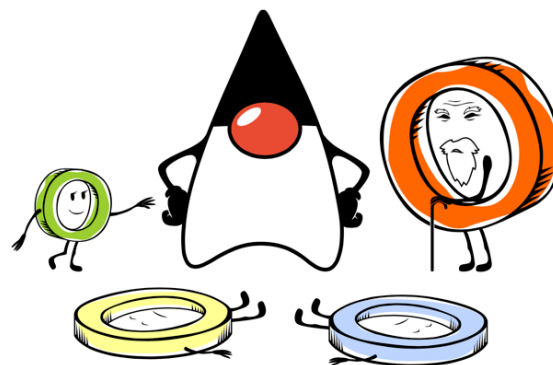
Все это навело разработчиков на мысль, что в первую очередь необходимо сосредотачиваться на очистке тех объектов, которые были созданы совсем недавно. Именно среди них чаще всего находится большее число тех, кто уже отжил свое, и именно здесь можно получить максимум эффекта при минимуме трудозатрат.

Вот тут и возникает идея разделения объектов на *младшее поколение (young generation)* и *старшее поколение (old generation)*. В соответствии с этим

разделением и процессы сборки мусора разделяются на *малую сборку (minor GC)*,

затрагивающую только младшее поколение, и

полную сборку (full GC), которая может затрагивать оба поколения. Малые сборки выполняются достаточно часто и удаляют основную часть мертвых объектов. Полные сборки выполняются тогда, когда текущий объем выделенной программе памяти близок к исчерпанию и малой сборкой уже не обойтись.



При этом разделение объектов по поколениям не просто условное, они физически размещаются в разных регионах памяти. Объекты из младшего поколения по мере выживания в сборках мусора переходят в старшее поколение. В старшем поколении объект может прожить до окончания работы приложения, либо будет удален в процессе одной из полных сборок мусора.

Вам быстро, дешево или качественно?

Интуитивно понятно, что желательно иметь сборщик мусора, который как можно быстрее избавлялся бы от ненужных объектов, расчищая дорогу молодым и обеспечивая тихое и спокойное существование долгожителям. Но работа сборщика мусора не бесплатная, она оплачивается ресурсами компьютера и задержками в выполнении программы. Поэтому прежде чем двигаться дальше, давайте разберемся с критериями, используемыми при оценке сборщиков.

Традиционно, при определении эффективности работы сборщика мусора учитываются следующие факторы:

- Максимальная задержка — максимальное время, на которое сборщик приостанавливает выполнение программы для выполнения одной сборки. Такие остановки называются *stop-the-world* (или *STW*).
- Пропускная способность — отношение общего времени работы программы к общему времени простоя, вызванного сборкой мусора, на длительном промежутке времени.
- Потребляемые ресурсы — объем ресурсов процессора и/или дополнительной памяти, потребляемых сборщиком.

Понятно, что добиться улучшения всех трех параметров одновременно практически невозможно. Уменьшение максимального времени задержки приводит к учащению сборки мусора, уменьшая пропускную способность. Либо приходится использовать более ухищренные алгоритмы для сохранения пропускной способности, что чаще всего увеличивает потребление ресурсов. И так далее.

Поэтому при настройке сборщиков мусора разработчики обычно фокусируются на оптимизации одного или двух параметров, стараясь сильно не ухудшать остальные, но жертвуя ими в случае необходимости.

Memento Mori

*Господи, дай мне места для размещения того, что пока еще нужно,
Дай мне смелости удалить то, что больше не пригодится,*

И дай мне мудрости, чтобы отличить одно от другого.

— Молитва сборщиков мусора

Еще один важный вопрос, который хотелось бы разобрать прежде, чем двигаться дальше, это определение самого понятия мусора, то есть *мертвых* объектов.

Как мы уже выяснили выше, путь большинства объектов от момента создания и исполнения своего предназначения до момента превращения в мусор, достаточно короток. Но существуют факторы, которые могут задержать его в мире живых чуть дольше, чем нам того хотелось бы.

Все мы знаем, что считать объект живым просто по факту наличия на него ссылок из других объектов нельзя. В противном случае рецепт бессмертия в JVM был бы до безобразия прост и заключался бы в наличии взаимных ссылок хотя бы у двух объектов друг на друга, а в общем случае — в наличии цикла в графе связанности объектов. При таком подходе и ограниченном объеме памяти более-менее серьезная программа долго не проработала бы, поэтому с отслеживанием циклов в графах объектов JVM справляется хорошо.

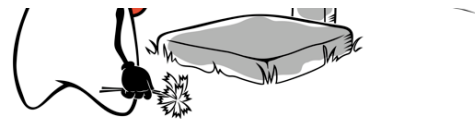
Но и просто сказать, что объект мертв и может быть удален только на основании того, что в программе не осталось ссылающихся на него (напрямую или опосредованно) еще используемых объектов, нельзя, так как разделение объектов на поколения вносит свои коррективы.

Рассмотрим такую ситуацию: У нас есть молодой объект А и ссылающийся на него объект В, уже заслуживший место в старшем поколении. В какой-то момент времени оба этих объекта стали нам не нужны и мы обнулили все имеющиеся у нас ссылки на них. Очевидно, объект А можно было бы удалить в ближайшую малую сборку мусора, но для того, чтобы получить это знание, сборщику пришлось бы просмотреть всё старшее поколение и понять, что объект В ссылающийся на А, тоже является мусором, а следовательно их оба можно утилизировать. Но анализ старшего поколения не входит в план малой сборки, так как является относительно дорогой процедурой, поэтому объект А во время малой сборки будет считаться живым.

Таким образом, чаще всего для целей малой сборки



мусора объект считается мертвым и подлежащим утилизации, если до него невозможно добраться по ссылкам ни из объектов старшего поколения, ни из так называемых *корней* (*roots*), к каковым относятся ссылки из стеков потоков, статические члены классов и т. п. При полной сборке мусора могут анализироваться оба поколения, поэтому здесь сборщик может плясать только от корней.



Кстати, время от момента, когда объект стал нам не нужен, до момента его фактического удаления из памяти называется *проворством* (*promptness*) и иногда рассматривается как дополнительный фактор оценки эффективности сборщика.

Под микроскопом

Итак, мы уже получили базовые представления о том, чем занимаются сборщики мусора и по каким критериям их можно оценивать. Теперь хотелось бы разобраться, каким образом можно заглянуть внутрь виртуальной машины, чтобы у нас была возможность наблюдать за работой ее скрытых механизмов.

Инструменты мониторинга памяти и процессов сборки мусора целесообразно разделить на две группы:

- внутренние, являющиеся частью той программы, которую мы мониторим,
- внешние, подключаемые к процессу исследуемой программы извне.

Проблема с инструментами мониторинга памяти в том, что они самим фактом наблюдения за памятью и сборками мусора, как в квантовой механике или в психологии, влияют на поведение подопытного. Ниже я приведу пример такого изменения поведения, а пока просто нужно запомнить, что какой бы инструмент вы ни использовали, следует проверить его калибровку хотя бы на простом примере: запустите программу, которая ничего не делает, и помониторьте ее.

Внутренние инструменты

Что касается внутренних инструментов мониторинга, то здесь мы можем либо попросить JVM выводить информацию о производимых сборках с различным уровнем детализации (в `stdout` или в лог-файл), либо самостоятельно обращаться к MXBean'ам, возвращающим информацию о состоянии памяти и о выполняемых сборках мусора, и обрабатывать ее как

нам вздумается.

В JVM HotSpot доступны следующие опции, управляющие выводом информации о сборках мусора (это основные опции, работающие для всех сборщиков):

<code>-verbose:gc</code>	Включает режим логирования сборок мусора в stdout.
<code>-Xloggc:filename</code>	Указывает имя файла, в который должна логироваться информация о сборках мусора. Имеет приоритет над <code>-verbose:gc</code> .
<code>-XX:+PrintGCTimeStamps</code>	Добавляет к информации о сборках временные метки (в виде количества секунд, прошедших с начала работы программы).
<code>-XX:+PrintGCDetails</code>	Включает расширенный вывод информации о сборках мусора.
<code>-XX:+PrintFlagsFinal</code>	При старте приложения выводит в stdout значения всех опций, заданных явно или установленных самой JVM. Сюда же попадают опции, относящиеся к сборке мусора. Часто бывает полезно посмотреть на присвоенные им значения.

Если вы хотите собирать данные из своего приложения самостоятельно, то для этого можно использовать соответствующие MXBean'ы. Вот пример простого класса, который позволяет выводить текущее состояние различных регионов памяти, а также информацию о сборках мусора, его можно взять за основу, если хотите разработать свой собственный мониторинг:

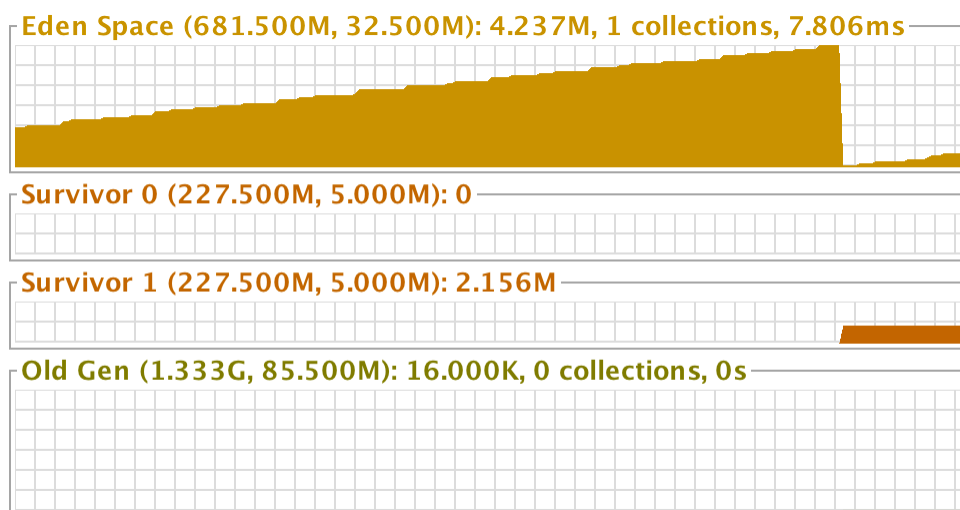
► [MemoryUtil.java](#)

Внешние инструменты

В природе существует огромное количество инструментов, позволяющих подключиться к процессу Java и в удобном виде получить информацию о состоянии памяти и процессах сборки мусора. Это и входящие в поставку JVM HotSpot утилиты VisualVM (с плагином

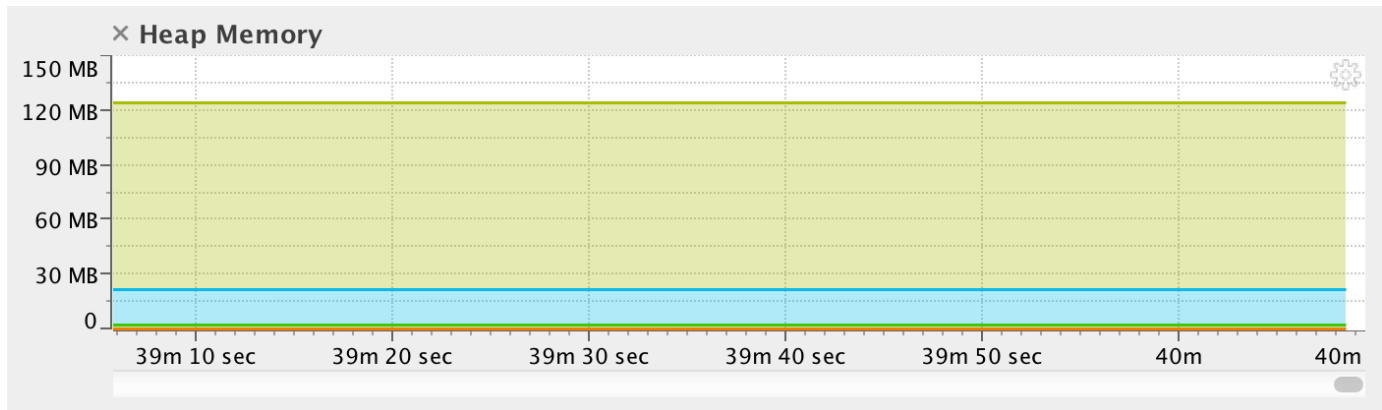
VisualGC) и Java Mission Control и различные инструменты/плагины для IDE и отдельные программы вроде JProfiler или YourKit и еще много чего.

Вы можете выбрать то, чем вам удобнее пользоваться, но как уже было сказано выше, обязательно проверьте, какое влияние оказывает ваш инструмент и его настройки на подопытное приложение. Вот пример того, как VisualVM влияет на поведение программы, весь исполняемый код которой состоит из приостановки выполнения основного потока:



Видите этот растущий график в верхней части? Это почти 8 МБ мусорных данных в минуту, привносимых мониторингом. Если вам нужно общее представление о том, как работает сборщик, либо если десяток мегабайт данных в минуту для вашей программы меньше допустимой погрешности измерений, то такое поведение инструменту можно простить. Но если вы проводите тонкую настройку и у вас каждый мегабайт на счету, то лучше выбрать что-нибудь менее прожорливое.

В идеале, ваш инструмент должен отображать график использования памяти коматозной программой как-нибудь так:



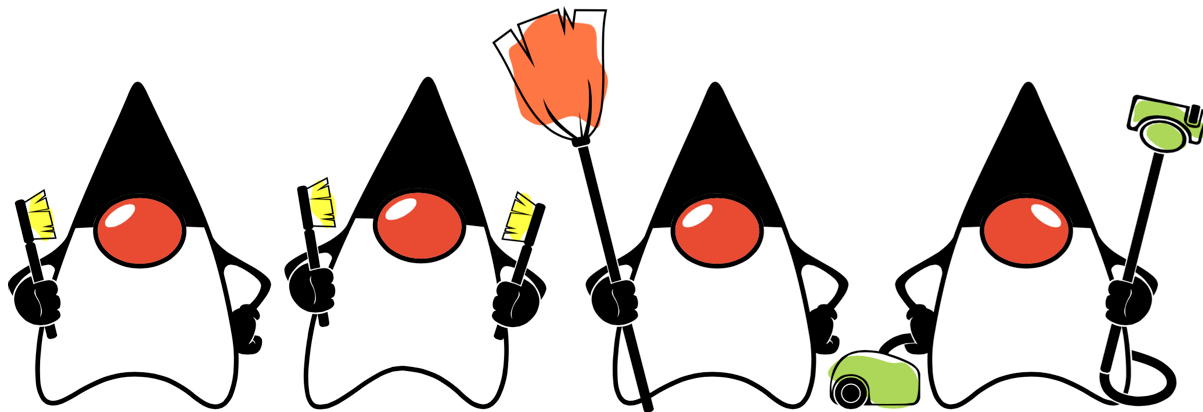
Как вариант, обратите внимание на описанные выше внутренние инструменты мониторинга, они изначально легковесные, а при необходимости добавления расширенных функций вы можете сами влиять на их прожорливость.

А можно всех посмотреть?

Ну что ж, раз вы добрались до этого места и вас не остановило даже долгое перечисление очевидных вещей в предыдущих параграфах, то вам и впрямь должно быть интересно. Давайте тогда уже взглянем на то, что же нам предоставляет HotSpot из коробки.

Как уже было сказано, описанные выше принципы сборки мусора являются общими для всех сборщиков. Но при этом между сборщиками существуют и заметные различия, проявляющиеся в ответах на следующие вопросы:

1. Какое количество регионов кучи используется, каково их назначение и размеры? Как эти размеры изменяются динамически?
2. Как устроен перевод объектов из младшего поколения в старшее?
3. Какие из работ по сборке мусора выполняются параллельно с работой основной программы, а какие приводят к ее остановке?
4. Каким образом сборщик мусора автоматически подстраивается под требуемые параметры производительности? Каким из них отдает приоритет?
5. Какие существуют возможности по настройке сборщика?



Java HotSpot VM предоставляет разработчикам на выбор семь различных сборщика мусора:

Serial (последовательный) — самый простой вариант для приложений с небольшим объемом данных и не требовательных к задержкам. Редко когда используется, но на слабых компьютерах может быть выбран виртуальной машиной в качестве сборщика по умолчанию.

Parallel (параллельный) — наследует подходы к сборке от последовательного сборщика, но добавляет параллелизм в некоторые операции, а также возможности по автоматической подстройке под требуемые параметры производительности.

Concurrent Mark Sweep (CMS) — нацелен на снижение максимальных задержек путем выполнения части работ по сборке мусора параллельно с основными потоками приложения. Подходит для работы с относительно большими объемами данных в памяти.

Garbage-First (G1) — создан для постепенной замены CMS, особенно в серверных приложениях, работающих на многопроцессорных серверах и оперирующих большими объемами данных.

Epsilon GC — разработан для случаев, когда сборка мусора вообще не нужна.

ZGC — пытается удерживать паузы на субмиллисекундном уровне, даже при работе с очень большими кучами.

Shenandoah GC — еще один сборщик, нацеленный на ультракороткие паузы независимо от размера кучи.

В следующих статьях мы детально рассмотрим каждый из этих сборщиков, стараясь придерживаться общего плана: краткое описание, принципы работы, ситуации STW (это stop the world, если успели забыть), способы настройки, достоинства и недостатки. Получив эти знания, мы посмотрим, что с ними делать в реальной жизни.

[Часть 2 — Сборщики Serial GC и Parallel GC →](#)

[Часть 3 — Сборщики CMS GC и G1 GC →](#)

[Часть 4 — Сборщик ZGC →](#)

[Часть 5 — Сборщик Epsilon GC →](#)

[Часть 6 — Сборщик Shenandoah GC →](#)

Теги: [java](#), [jvm](#), [hotspot](#), [gc](#), [garbage collection](#)

Хабы: [Java](#)

**47****0**

Карма Рейтинг

Андрей Лыгин [@alygin](#)

Оператор ЭВМ

[Подписаться](#)[Задонатить](#)**Комментарии 7**

igor_suhorukov
27.10.2015 в 16:37

5 за доступность для новичков! Очень легкий стиль изложения и с картинками)

+3 [Ответить](#)



alygin
27.10.2015 в 17:00

Спасибо! Продолжение будет уже более детальным и, надеюсь, интересным не только новичкам. Картинки тоже обещаю, не переключайтесь)

+3 [Ответить](#)




igor_suhorukov
27.10.2015 в 17:28



На эту тему могу посоветовать читателям публикации коллеги и про области применимости алгоритмов GC

 +2 Ответить



 **23derevo**
28.10.2015 в 01:57

Тег «java memory model» в этом посте явно лишний. Уберите его, пожалуйста.

Моделью памяти в языке Java называется совершенно другая вещь, а именно фрагмент спецификации языка Java, §17.4 JLS.

Более того, насколько я помню, с точки зрения стандарта Java SE 8 (7, 6, 5 и др.), GC нет ни в спецификации языка, ни в спецификации виртуальной машины. Он есть только в библиотеке классов в виде `System.gc()` и `Runtime.gc()`.

 +4 Ответить



 **alygin**
28.10.2015 в 02:26

Про memory model согласен, убрал из тегов.

 +2 Ответить



 **Randl**
28.10.2015 в 06:55

А что можно почитать о самих алгоритмах сборки мусора в принципе? С псевдокодом и сравнением по разным параметрам? Про [The Garbage Collection Handbook](#) слышал, но что-то мне подсказывает, что она слишком академичной окажется.

 0 Ответить



 **alygin**
28.10.2015 в 10:42

Есть еще [Garbage Collection: Algorithms for Automatic Dynamic Memory Management](#) с псевдокодом и сравнениями. В открытом доступе можно найти ее очень плохой скан, но по нему сможете сделать заключение о степени академичности. Мне показалась вполне практичной, но я ее не целиком читал, а только выборочно интересующие меня главы.

 0 Ответить



Вы можете оставлять комментарии только к свежим постам

ПОХОЖИЕ ПУБЛИКАЦИИ

2 ноября 2015 в 16:07

Дюк, вынеси мусор! — 3. CMS и G1

◆ +29

👁 120K

🔖 366

💬 39

29 октября 2015 в 15:27

Дюк, вынеси мусор! — 2. Serial GC и Parallel GC

◆ +36

👁 101K

🔖 383

💬 22

1 апреля 2013 в 19:16

Тюнинг JVM на примере одного проекта

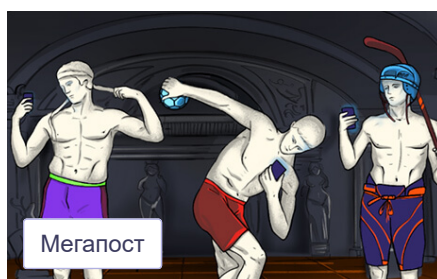
◆ +27

👁 27K

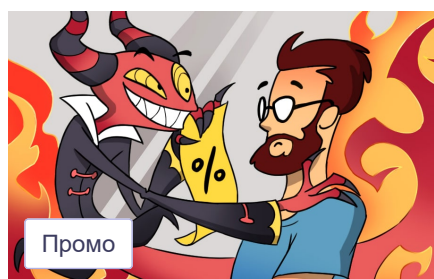
🔖 226

💬 40

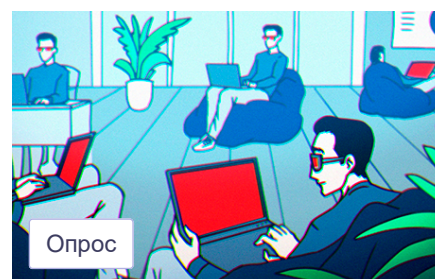
МИНУТОЧКУ ВНИМАНИЯ



Гуляем по «Спортмастеру»
ножками и в приложениях



Промокод — твой билет в
общество потребления



Собираем комп для
хабрапользователя

ВОПРОСЫ И ОТВЕТЫ

В чем проблема не корректного исполнения запроса ROOM?

Java · Простой · 1 ответ

Нужно чтобы Уроки сохранялись в объекте, в метод передать объект Lesson и сохранить его в в объекте LessonService?

Java · Простой · 1 ответ

Нужно ли закрывать bufferreader и тому подобные в Java?

Java · Простой · 1 ответ

В каком случае целесообразно изучать Java если ты php разработчик микросервисов?

Java · Простой · 3 ответа

Как соединить spring framework 5 и hibernate 6?

Java · Простой · 2 ответа

[Больше вопросов на Хабр Q&A](#)

ЛУЧШИЕ ПУБЛИКАЦИИ ЗА СУТКИ

вчера в 16:00

Новая волна ARM-процессоров. Серверы на старте

 +24

 3.6K

 14

 9 +9

вчера в 13:01

Итоги недели мобилизации. Уклониться нельзя судить. Указа не будет. Приглашаем редактировать наше письмо Мишустину

 +22

 29K

 20

 369 +369

вчера в 20:32

Как изменились батарейки FLARX: тест 2022 года

 +21

 4.5K

 6

 9 +9

вчера в 16:30

Рассуждения об asyncio.Semaphore

 +17

 1.5K

 13

 0

вчера в 15:46

Памятка для UX-дизайнера на все случаи жизни: как взяться за задачу и довести ее до

конца



 [Настройка языка](#)

[Техническая поддержка](#)

[Полная версия](#)

[Вернуться на старую версию](#)

© 2006–2022, Habr