




Lesson 16

18.07.2022




Лямбда представляет собой набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызвать в различных местах программы.

Основу лямбда-выражения составляет лямбда-оператор, который представляет стрелку \rightarrow . Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список параметров выражения, а правая собственно представляет тело лямбда-выражения, где выполняются все действия.

$$(x, y) \rightarrow x + y;$$

Лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в функциональном интерфейсе. При этом важно, что функциональный интерфейс должен содержать только один единственный метод без реализации.

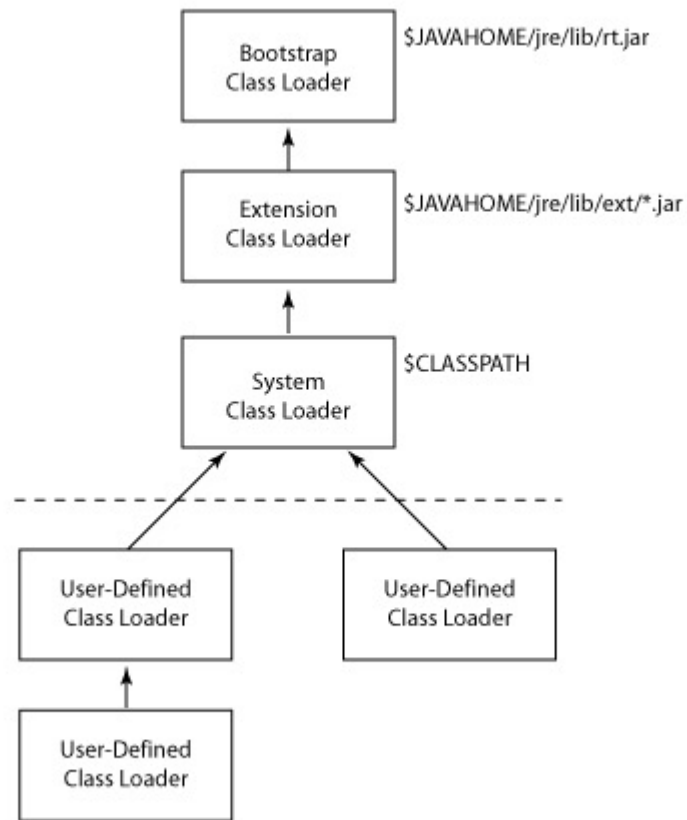



По факту лямбда-выражения являются в некотором роде сокращенной формой внутренних анонимных классов, которые ранее применялись в Java.

- Отложенное выполнение (deferred execution) лямбда-выражения определяется один раз в одном месте программы, вызываются при необходимости, любое количество раз и в произвольном месте программы.
- Параметры лямбда-выражения должны соответствовать по типу параметрам метода функционального интерфейса.
- *Конечные лямбда-выражения* не обязаны возвращать какое-либо значение.
- *Блочные лямбда-выражения* обрамляются фигурными скобками. В блочных лямбда-выражениях можно использовать внутренние вложенные блоки, циклы, конструкции if, switch, создавать переменные и т.д. Если блочное лямбда-выражение должно возвращать значение, то явным образом применяется оператор return.
- *Передача лямбда-выражения в качестве параметра метода.*



Загрузка классов в Java





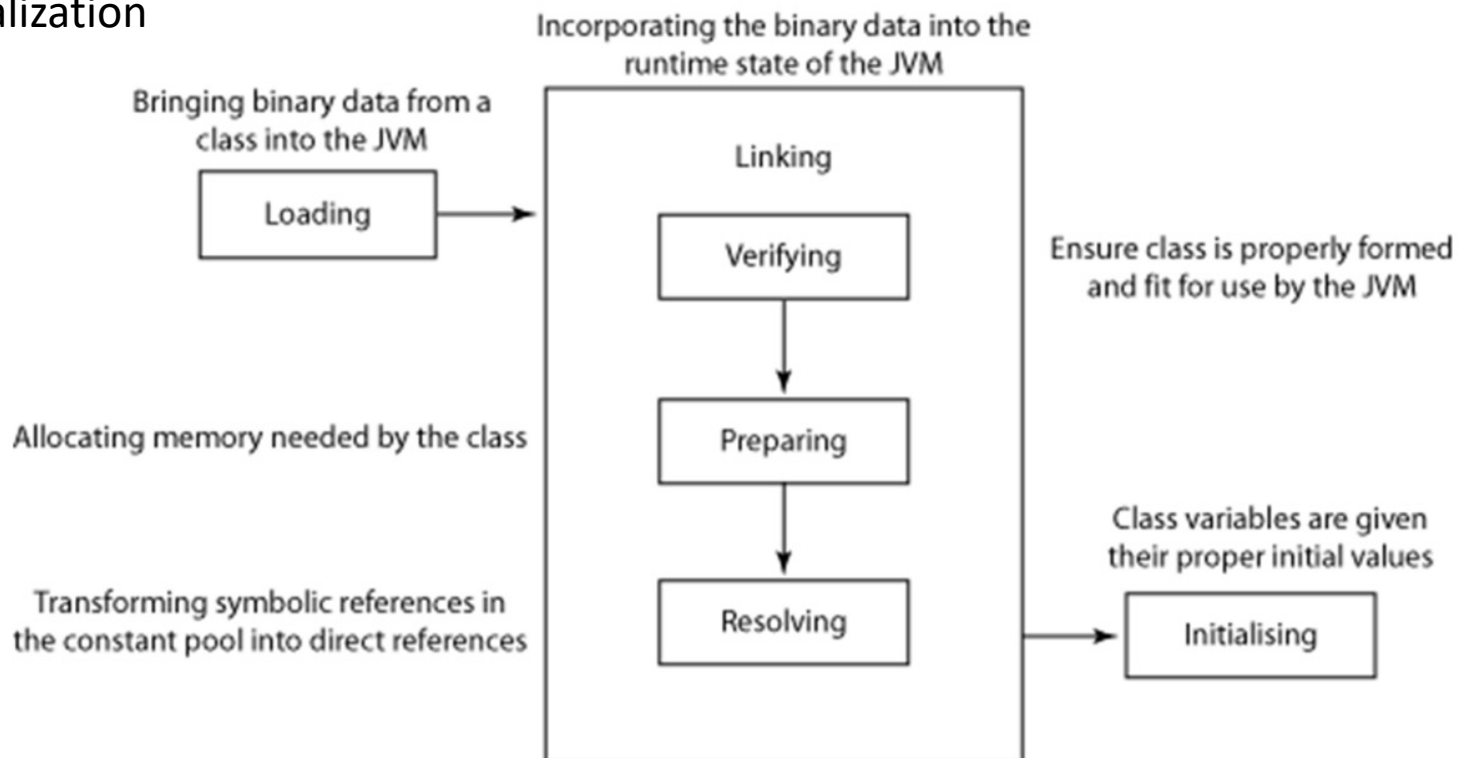
Bootstrap — реализован на уровне JVM и не имеет обратной связи со средой исполнения. Данным загрузчиком загружаются классы из директории `$JAVA_HOME/lib`. Т.е. всеми любимый `rt.jar` загружается именно базовым загрузчиком. Поэтому, попытка получения загрузчика у классов `java.*` всегда заканчивается `null`'ом. Это объясняется тем, что все базовые классы загружены базовым загрузчиком, доступа к которому из управляемой среды нет.


System Classloader — системный загрузчик, реализованный уже на уровне JRE. В Sun JRE — это класс `sun.misc.Launcher$AppClassLoader`. Этим загрузчиком загружаются классы, пути к которым указаны в переменной окружения `CLASSPATH`.

Extension Classloader — загрузчик расширений. Данный загрузчик загружает классы из директории `$JAVA_HOME/lib/ext`

Процесс загрузки класса

1. Loading
2. Linking
3. Initialization





Loading – на этой фазе происходит поиск и физическая загрузка файла класса в определенном источнике (в зависимости от загрузчика). Этот процесс определяет базовое представление класса в памяти. На этом этапе такие понятия как методы, поля и т.д. пока не известны.

Linking – процесс, который может быть разбит на 3 части:

1. Bytecode verification – происходит несколько проверок байт-кода на соответствие ряду зачастую нетривиальных требований определенных в спецификации JVM.

2. Class preparation – на этом этапе происходит подготовки структуры данных, отображающей поля, методы и реализованные интерфейсы, которые определены в классе.

3. Resolving – разрешение всех классов, которые ссылаются на текущий класс.

Initialization – происходит выполнение статических инициализаторов определенных в классе. Таким образом, статические поля инициализируются стандартными значениями.

Java Memory Model

Heap

PermGen
(Method
Area)

Thread
1..N

YoungGeneration

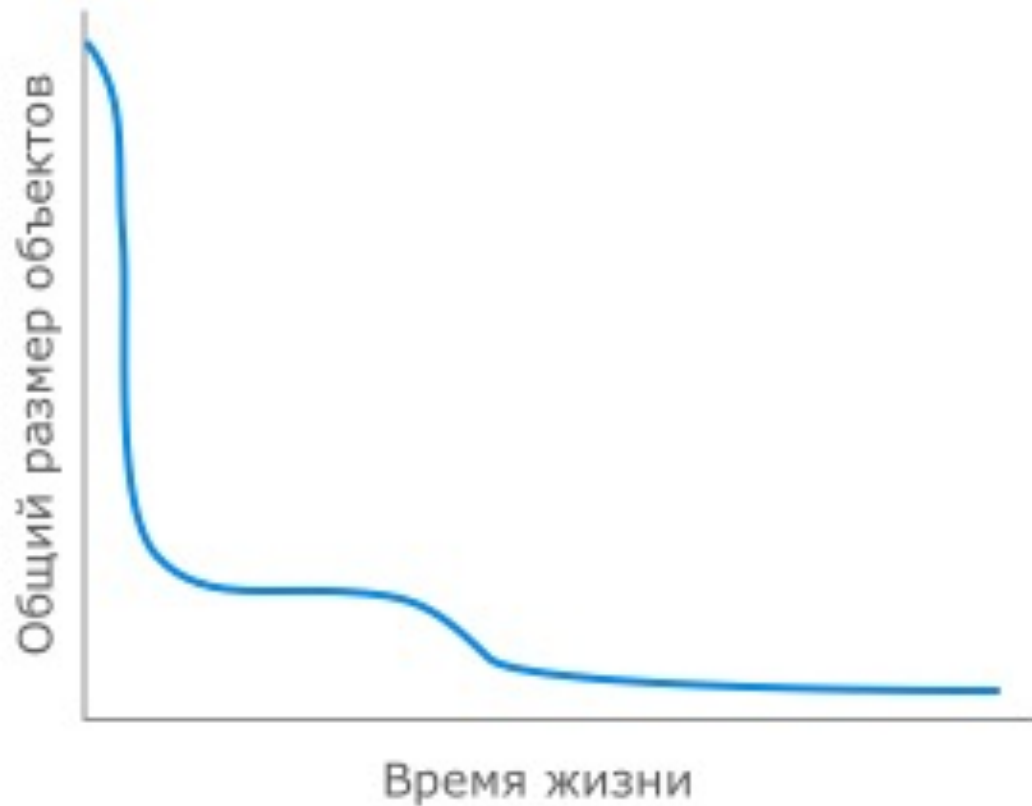
Old/Tenured
Generation


EdenSpace

FromSpace
(Survivor1)

ToSpace
(Survivor2)

Гипотеза о поколениях



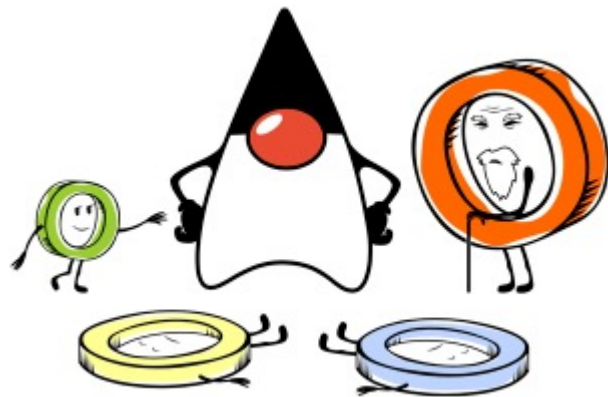


Подавляющее большинство объектов создаются **на очень короткое время**, они становятся ненужными практически сразу после их первого использования. **Итераторы, локальные переменные методов, результаты боксинга и прочие временные объекты**, которые зачастую создаются неявно, попадают именно в эту категорию,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

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.





Эффективности работы сборщика мусора :

- ✓ **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.



Garbage Collector

Сборщик мусора Garbage Collector выполняет всего две задачи, связанные с поиском мусора и его очисткой. Для обнаружения мусора существует два подхода

Reference counting – учет ссылок;

Tracing – трассировка.



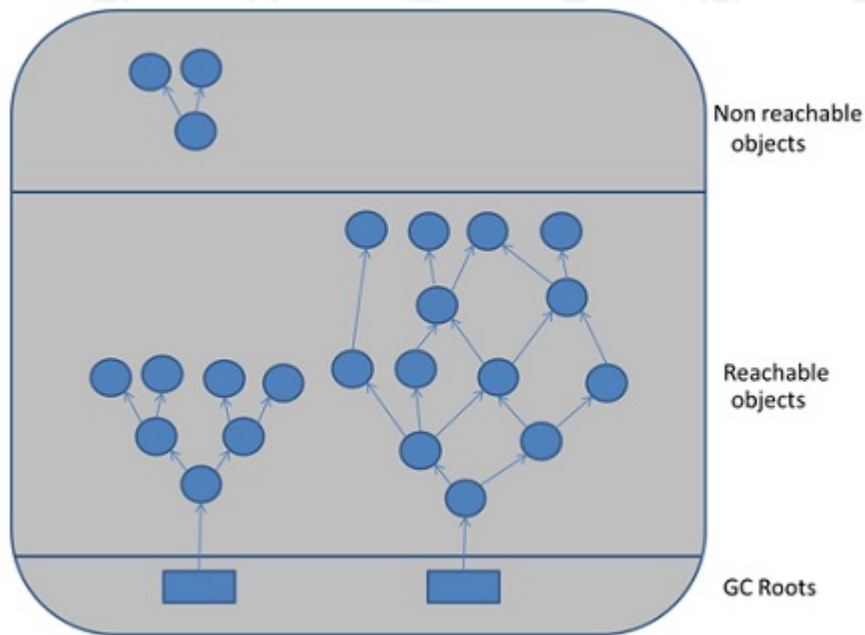
Reference counting


Суть подхода «Reference counting» связана с тем, что каждый объект имеет счетчик, который хранит информацию о количестве указывающих на него ссылок. При уничтожении ссылки счетчик уменьшается. При нулевом значении счетчика объект можно считать мусором.

Главным недостатком данного подхода является сложность обеспечения точности счетчика и «невозможность» выявлять циклические зависимости. Так, например, два объекта могут ссылаться друг на друга, но ни на один из них нет внешней ссылки. Это сопровождается утечками памяти. В этой связи данный подход не получил распространения.

Tracing

Главная идея «Tracing» связана с тем, что до «живого» объекта можно добраться из корневых точек (GC Root). Всё, что доступно из «живого» объекта, также является «живым». Если представить все объекты и ссылки между ними как дерево, то необходимо пройти от корневых узлов GC Roots по всем узлам. При этом узлы, до которых нельзя добраться, являются мусором.






Данный подход, обеспечивающий выявление циклических ссылок, используется в виртуальной машине HotSpot VM. Теперь, осталось понять, а что представляет из себя корневая точка (GC Root)? «Источники» говорят, что существуют следующие типы корневых точек :

- Основной Java поток.
- Локальные переменные в основном методе.
- Статические переменные основного класса.



В Java существует 4 типа ссылок на объекты:

- strong reference
- soft reference
- weak reference
- phantom reference



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.

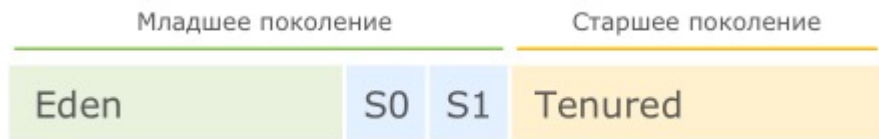
Concurrent Mark Sweep (CMS) — нацелен на снижение максимальных задержек путем выполнения части работ по сборке мусора параллельно с основными потоками приложения. Подходит для работы с относительно большими объемами данных в памяти.

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

Serial GC

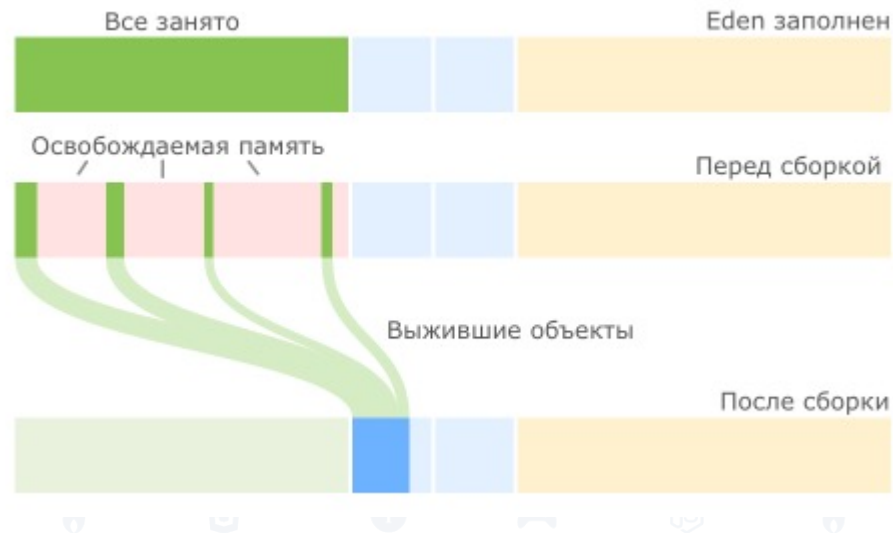
Использование Serial GC включается опцией **-XX:+UseSerialGC**.

При использовании данного сборщика куча разбивается на четыре региона, три из которых относятся к младшему поколению (Eden, Survivor 0 и Survivor 1), а один (Tenured) — к старшему:



Среднестатистический объект начинает свою жизнь в регионе Eden. Именно сюда его помещает JVM в момент создания. Но со временем может оказаться так, что места для вновь создаваемого объекта в Eden нет, в таких случаях запускается малая сборка мусора.

Первым делом такая сборка **находит и удаляет мертвые объекты** из Eden. Оставшиеся **живые объекты переносятся в пустой регион Survivor**. Один из двух регионов Survivor всегда пустой, именно он выбирается для переноса объектов из Eden:



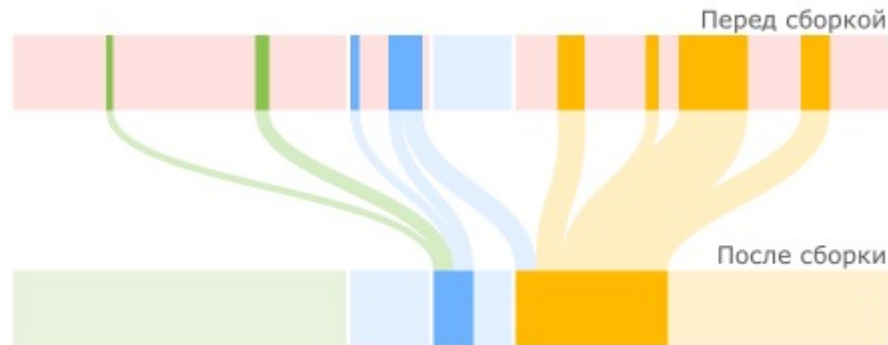
После малой сборки регион Eden полностью опорожнен и может быть использован для размещения новых объектов. Но рано или поздно наше приложение опять займет всю область Eden и JVM снова попытается провести малую сборку, на этот раз очищая Eden и частично занятый Survivor 0, после чего перенося все выжившие объекты в пустой регион Survivor 1:



JVM постоянно следит за тем, как долго объекты перемещаются между Survivor 0 и Survivor 1, и выбирает подходящий порог для количества таких перемещений, после которого объекты перемещаются в Tenured, то есть переходят в старшее поколение. Если регион Survivor оказывается заполненным, то объекты из него также отправляются в Tenured:



В случае, когда места для новых объектов не хватает уже в Tenured, в дело вступает полная сборка мусора, работающая с объектами из обоих поколений. При этом старшее поколение не делится на подрегионы по аналогии с младшим, а представляет собой один большой кусок памяти, поэтому после удаления мертвых объектов из Tenured производится не перенос данных (переносить уже некуда), а их уплотнение, то есть размещение последовательно, без фрагментации. Такой механизм очистки называется Mark-Sweep-Compact по названию его шагов (позметить выжившие объекты, очистить память от мертвых объектов, уплотнить выжившие объекты).





Ситуации STW

С этим сборщиком все достаточно просто, так как вся его работа — это один сплошной STW. В начале каждой сборки мусора работа основных потоков приложения останавливается и возобновляется только после окончания сборки. Причем всю работу по очистке Serial GC выполняет не торопясь, в одном потоке, последовательно, за что и удостоился своего имени.



Достоинства и недостатки

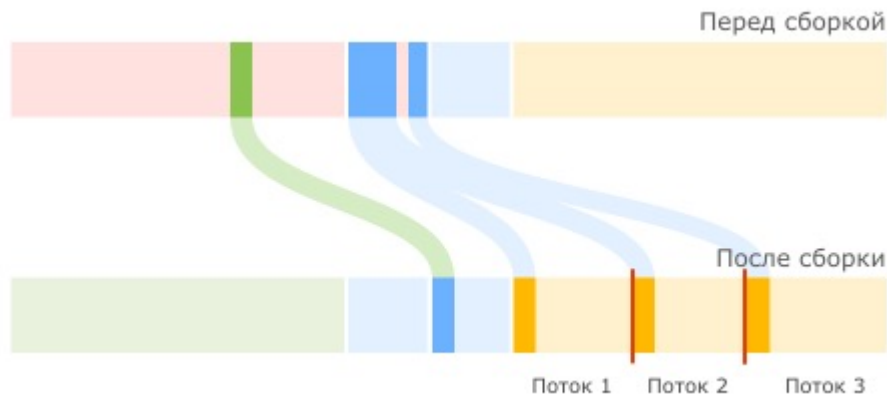
Основное достоинство данного сборщика очевидно — это неприязнительность по части ресурсов компьютера. Так как всю работу он выполняет последовательно в одном потоке, никаких заметных оверхедов и негативных побочных эффектов у него нет.


Главный недостаток тоже понятен — это долгие паузы на сборку мусора при заметных объемах данных.

Parallel GC

Параллельный сборщик включается опцией **-XX:+UseParallelGC**.

При подключении параллельного сборщика используются те же самые подходы к организации кучи, что и в случае с Serial GC — она делится на такие же регионы Eden, Survivor 0, Survivor 1 и Old Gen, функционирующие по тому же принципу. Но есть два принципиальных отличия в работе с этими регионами: во-первых, сборкой мусора занимаются **несколько потоков параллельно**; во-вторых, данный сборщик может **самостоятельно подстраиваться под требуемые параметры производительности**. Давайте разберемся, как это устроено.





По умолчанию и малая и полная сборка задействуют многопоточность. Малая пользуется ею при переносе объектов в старшее поколение, а полная — при уплотнении данных в старшем поколении.

Каждый поток сборщика получает свой участок памяти в регионе Old Gen, так называемый *буфер повышения (promotion buffer)*, куда только он может переносить данные, чтобы не мешать другим потокам.

Ситуации STW

Как и в случае с последовательным сборщиком, на время операций по очистке памяти все основные потоки приложения останавливаются. Разница только в том, что пауза, как правило, короче за счет выполнения части работ в параллельном режиме.



Достоинства и недостатки

Бесспорным плюсом данного сборщика на фоне Serial GC является возможность автоматической подстройки под требуемые параметры производительности и меньшие паузы на время сборок. При наличии нескольких процессорных ядер выигрыш в скорости будет практически во всех приложениях.

Определенная фрагментация памяти, конечно, является минусом, но вряд ли она будет существенной для большинства приложений, так как сборщиком используется относительно небольшое количество потоков.

В целом, Parallel GC — это простой, понятный и эффективный сборщик, подходящий для большинства приложений. У него нет скрытых накладных расходов, мы всегда можем поменять его настройки и ясно увидеть результат этих изменений.



Сборщик CMS (расшифровывается как Concurrent Mark Sweep)

Использование CMS GC включается опцией **-XX:+UseConcMarkSweepGC**

Мы уже встречали слова Mark и Sweep при рассмотрении последовательного и параллельного сборщиков. Они обозначали два шага в процессе сборки мусора в старшем поколении: пометку выживших объектов и удаление мертвых объектов. Сборщик CMS получил свое название благодаря тому, что выполняет указанные шаги параллельно с работой основной программы.

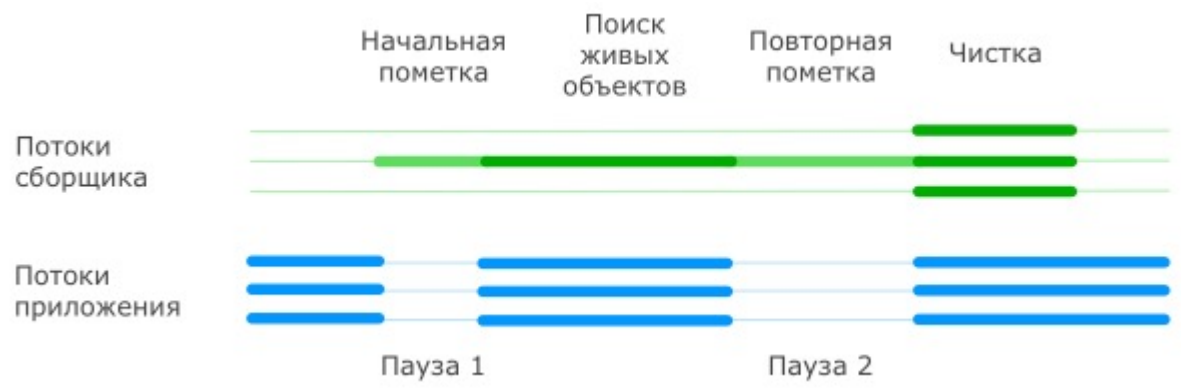
При этом CMS GC использует ту же самую организацию памяти, что и уже рассмотренные Serial / Parallel GC: регионы Eden + Survivor 0 + Survivor 1 + Tenured и такие же принципы малой сборки мусора. Отличия начинаются только тогда, когда дело доходит до полной сборки. В случае CMS ее называют **старшей (major) сборкой**, а не полной, так как она не затрагивает объекты младшего поколения. В результате, малая и старшая сборки здесь всегда разделены



Начинается она с остановки основных потоков приложения и пометки всех объектов, напрямую доступных из корней. После этого приложение возобновляет свою работу, а сборщик параллельно с ним производит поиск всех живых объектов, доступных по ссылкам из тех самых помеченных корневых объектов (эту часть он делает в одном или в нескольких потоках).

Естественно, за время такого поиска ситуация в куче может поменяться, и не вся информация, собранная во время поиска живых объектов, оказывается актуальной. Поэтому сборщик еще раз приостанавливает работу приложения и просматривает кучу для поиска живых объектов, ускользнувших от него за время первого прохода.

После того, как живые объекты помечены, работа основных потоков приложения возобновляется, а сборщик производит очистку памяти от мертвых объектов в нескольких параллельных потоках. При этом следует иметь в виду, что после очистки не производится упаковка объектов в старшем поколении, так как делать это при работающем приложении весьма затруднительно.





Ситуации STW

Из всего сказанного выше следует, что при обычной сборке мусора у CMS GC существуют следующие ситуации, приводящие к STW: Малая сборка мусора. Эта пауза ничем не отличается от аналогичной паузы в Parallel GC.

Начальная фаза поиска живых объектов при старшей сборке (так называемая *initial mark pause*). Эта пауза обычно очень короткая.

Фаза дополнения набора живых объектов при старшей сборке (известная также как *remark pause*). Она обычно длиннее начальной фазы поиска.


В случае же возникновения сбоя конкурентного режима пауза может затянуться на достаточно длительное время.



Достоинства и недостатки

Достоинством данного сборщика по сравнению с рассмотренными ранее Serial / Parallel GC является его ориентированность на минимизацию времен простоя, что является критическим фактором для многих приложений. Но для выполнения этой задачи приходится жертвовать ресурсами процессора и зачастую общей пропускной способностью.

Вспомним еще, что данный сборщик не уплотняет объекты в старшем поколении, что приводит к фрагментации Tenured. Этот факт в совокупности с наличием плавающего мусора приводит к необходимости выделять приложению (конкретно — старшему поколению) больше памяти, чем потребовалось бы для других сборщиков (Oracle советует на 20% больше).



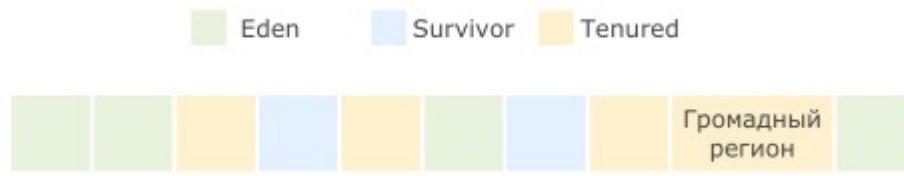
G1 GC


G1 — самый молодой в составе сборщиков мусора виртуальной машины HotSpot. Он изначально позиционировался как сборщик для приложений с большими кучами (от 4 ГБ и выше), для которых важно сохранять время отклика небольшим и предсказуемым, пусть даже за счет уменьшения пропускной способности.

G1 включается опцией Java **-XX:+UseG1GC**.


Первое, что бросается в глаза при рассмотрении G1 — это изменение подхода к организации кучи. Здесь память разбивается на множество регионов одинакового размера. Размер этих регионов зависит от общего размера кучи и по умолчанию выбирается так, чтобы их было не больше 2048, обычно получается от 1 до 32 МБ. Исключение составляют только так называемые *громадные (humongous) регионы*, которые создаются объединением обычных регионов для размещения очень больших объектов.

Разделение регионов на Eden, Survivor и Tenured в данном случае логическое, регионы одного поколения не обязаны идти подряд и даже могут менять свою принадлежность к тому или иному поколению. Пример разделения кучи на регионы может выглядеть следующим образом (количество регионов сильно приуменьшено):





Малые сборки выполняются периодически для очистки младшего поколения и переноса объектов в регионы Survivor, либо их повышения до старшего поколения с переносом в Tenured. Над переносом объектов трудятся несколько потоков, и на время этого процесса работа основного приложения останавливается. Это уже знакомый нам подход из рассмотренных ранее сборщиков, но отличие состоит в том, что очистка выполняется не на всем поколении, а только на части регионов, которые сборщик сможет очистить не превышая желаемого времени. При этом он выбирает для очистки те регионы, в которых, по его мнению, скопилось наибольшее количество мусора и очистка которых принесет наибольший результат. Отсюда как раз название Garbage First — мусор в первую очередь.

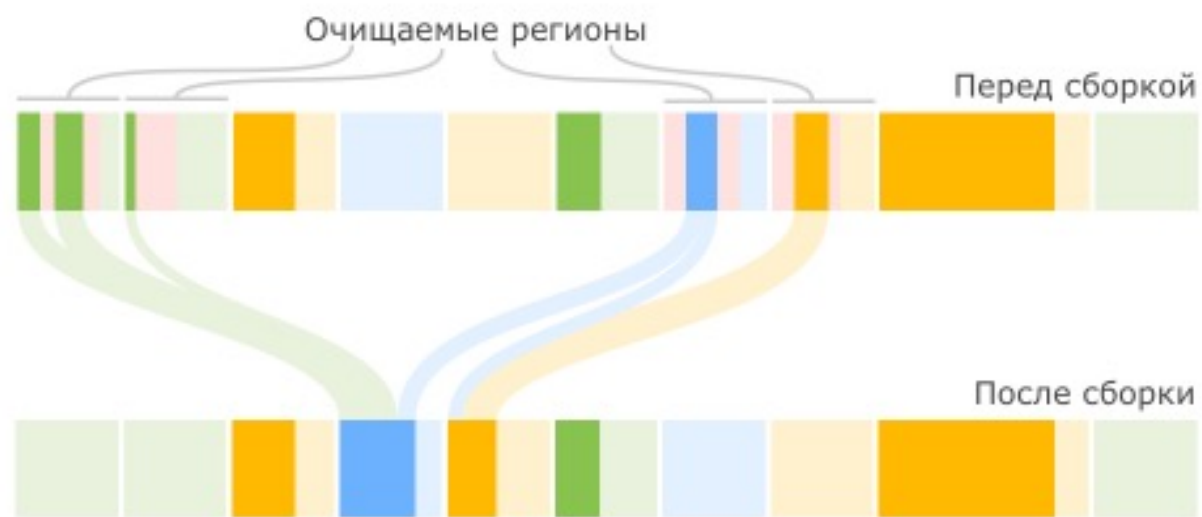


А с полной сборкой (точнее, здесь она называется *смешанной (mixed)*) все немного хитроумнее, чем в рассмотренных ранее сборщиках. В G1 существует процесс, называемый *циклом пометки (marking cycle)*, который работает параллельно с основным приложением и составляет список живых объектов. За исключением последнего пункта, этот процесс выглядит уже знакомо для нас: Initial mark. Пометка корней (с остановкой основного приложения) с использованием информации, полученной из малых сборок.

Concurrent marking. Пометка всех живых объектов в куче в нескольких потоках, параллельно с работой основного приложения.

Remark. Дополнительный поиск не учтенных ранее живых объектов (с остановкой основного приложения).

Cleanup. Очистка вспомогательных структур учета ссылок на объекты и поиск пустых регионов, которые уже можно использовать для размещения новых объектов. Первая часть этого шага выполняется при остановленном основном приложении.





Ситуации STW

Если резюмировать, то у G1 мы получаем STW в следующих случаях:


- Процессы переноса объектов между поколениями. Для минимизации таких пауз G1 использует несколько потоков.
- Короткая фаза начальной пометки корней в рамках цикла пометки.
- Более длинная пауза в конце фазы remark и в начале фазы cleanup цикла пометки.



Достоинства и недостатки

В целом считается, что сборщик G1 более аккуратно предсказывает размеры пауз, чем CMS, и лучше распределяет сборки во времени, чтобы не допустить длительных остановок приложения, особенно при больших размерах кучи. При этом он лишен и некоторых других недостатков CMS, например, он не фрагментирует память.

Расплатой за достоинства G1 являются ресурсы процессора, которые он использует для выполнения достаточно большой части своей работы параллельно с основной программой. В результате страдает пропускная способность приложения. Целевым значением пропускной способности по умолчанию для G1 является 90%. Для Parallel GC, например, это значение равно 99%. Это, конечно, не значит, что пропускная способность с G1 всегда будет почти на 10% меньше, но данную особенность следует всегда иметь в виду.




Паттерн проектирования — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ.

В отличие от готовых функций или библиотек, паттерн нельзя просто взять и скопировать в программу. Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды вашей программы.

Паттерны часто путают с алгоритмами, ведь оба понятия описывают типовые решения каких-то известных проблем. Но если алгоритм — это чёткий набор действий, то паттерн — это высокоуровневое описание решения, реализация которого может отличаться в двух разных программах.

Если привести аналогии, то алгоритм — это кулинарный рецепт с чёткими шагами, а паттерн — инженерный чертёж, на котором нарисовано **решение**, но **не** конкретные **шаги** его **реализации**.



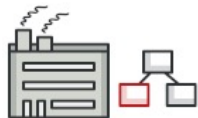
Концепцию паттернов впервые описал Кристофер Александер в книге **«Язык шаблонов. Города. Здания. Строительство»**. В книге описан «язык» для проектирования окружающей среды, единицы которого — шаблоны (или *паттерны*, что ближе к оригинальному термину *patterns*) — отвечают на архитектурные вопросы: какой высоты сделать окна, сколько этажей должно быть в здании, какую площадь в микрорайоне отвести под деревья и газоны.

Идея показалась заманчивой четвёрке авторов: Эриху Гамме, Ричарду Хелму, Ральфу Джонсону, Джону Влиссидесу. В 1995 году они написали книгу **«Приемы объектно-ориентированного проектирования. Паттерны проектирования»**, в которую вошли 23 паттерна, решающие различные проблемы объектно-ориентированного дизайна. Название книги было слишком длинным, чтобы кто-то смог всерьёз его запомнить. Поэтому вскоре все стали называть её «book by the gang of four», то есть «книга от банды четырёх», а затем и вовсе «GoF book».

- ❖ **Порождающие паттерны** беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей.
- ❖ **Структурные паттерны** показывают различные способы построения связей между объектами.
- ❖ **Поведенческие паттерны** заботятся об эффективной коммуникации между объектами.

Порождающие паттерны проектирования

Эти паттерны отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов.



Фабричный метод

Factory Method

Определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.



Абстрактная фабрика

Abstract Factory

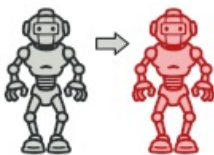
Позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



Строитель

Builder

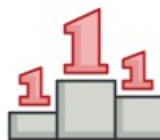
Позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.



Прототип

Prototype

Позволяет копировать объекты, не вдаваясь в подробности их реализации.



Одиночка

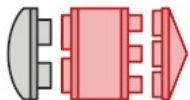
Singleton

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.



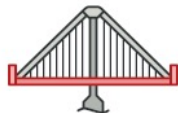
Структурные паттерны проектирования

Эти паттерны отвечают за построение удобных в поддержке иерархий классов.



Адаптер
Adapter

Позволяет объектам с несовместимыми интерфейсами работать вместе.



Мост
Bridge

Разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.



Компоновщик
Composite

Позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.



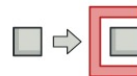
Фасад
Facade

Предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.



Декоратор
Decorator

Позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».



Заместитель
Proxy

Позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то *до* или *после* передачи вызова оригиналу.

Поведенческие паттерны проектирования

Эти паттерны решают задачи эффективного и безопасного взаимодействия между объектами программы.



**Цепочка
обязанностей**
Chain of Responsibility

Позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.



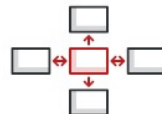
Команда
Command

превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.



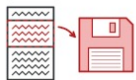
Итератор
Iterator

Даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.



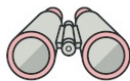
Посредник
Mediator

Позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.



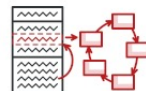
Снимок
Memento

Позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.



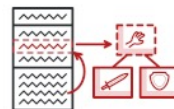
Наблюдатель
Observer

Создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.



Состояние
State

Позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.



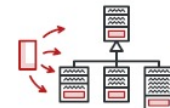
Стратегия
Strategy

Определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.



Шаблонный метод
Template Method

Определяет скелет алгоритма, переключая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.



Посетитель
Visitor

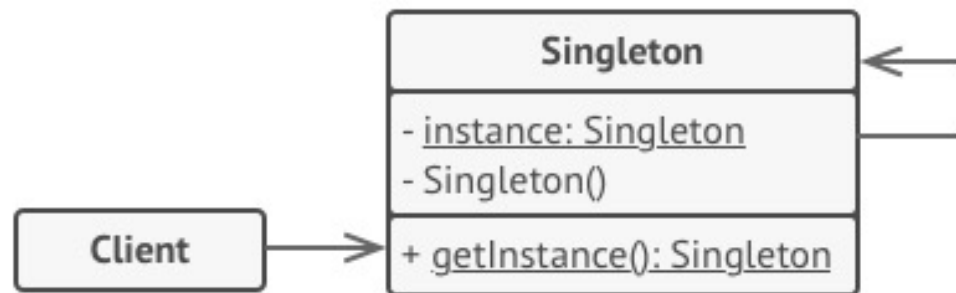
Позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.



Singleton

Гарантирует наличие единственного экземпляра класса. Чаще всего это полезно для доступа к какому-то общему ресурсу, например, базе данных.

Предоставляет глобальную точку доступа. Это не просто глобальная переменная, через которую можно достигаться к определённому объекту. Глобальные переменные не защищены от записи, поэтому любой код может подменять их значения без вашего ведома.



1 **Одиночка** определяет статический метод `getInstance`, который возвращает единственный экземпляр своего класса.

Конструктор одиночки должен быть скрыт от клиентов. Вызов метода `getInstance` должен стать единственным способом получить объект этого класса.

```
if (instance == null) {
    // Внимание, если вы пишете
    // многопоточный код, то здесь
    // нужно синхронизировать потоки.
    instance = new Singleton()
}
return instance
```