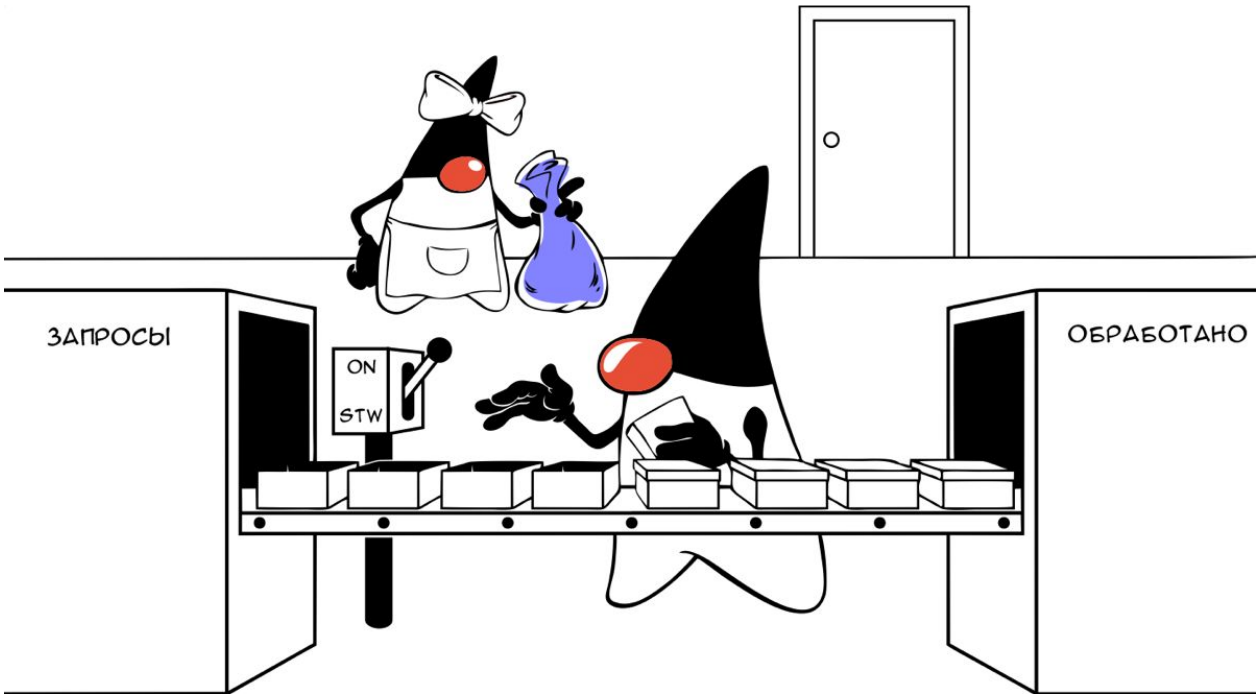
Core Java - 5

GC

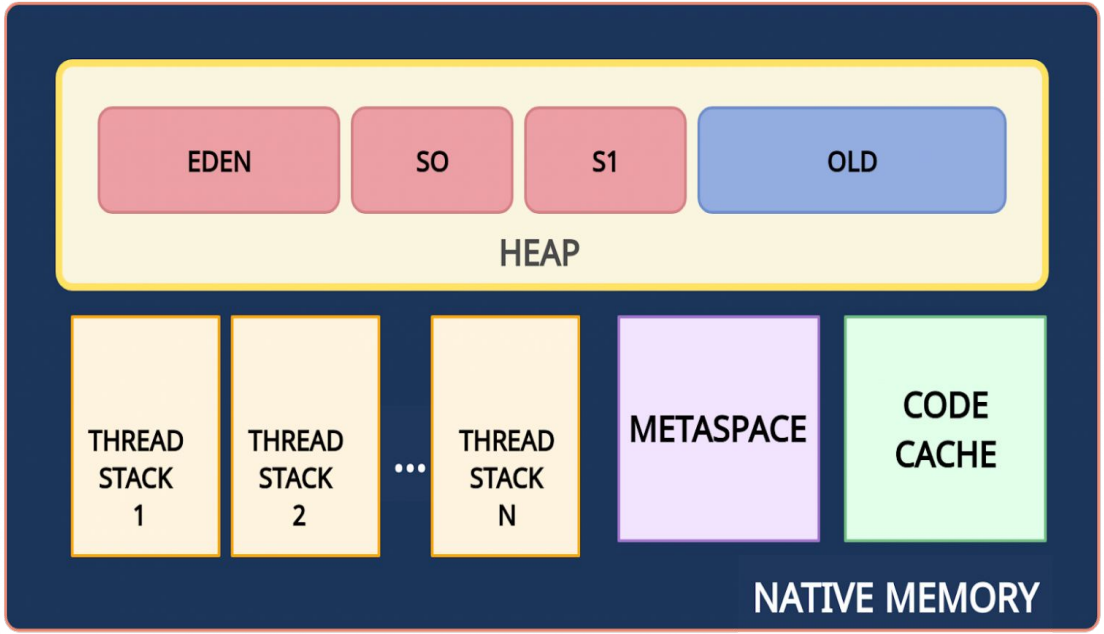


GC

Преимущества	Недостатки
● Ускорение разработки ● Защита от утечек памяти ● Нет необходимости отслеживать жизненный цикл объекта	● Потребление дополнительных ресурсов ● Утечки памяти :) ● Stop-the-word паузы

*** Утечки памяти** — это класс ошибок, когда приложению не удается освободить **память**, если она больше не нужна.

Память в Java



Native Memory — вся доступная системная память.

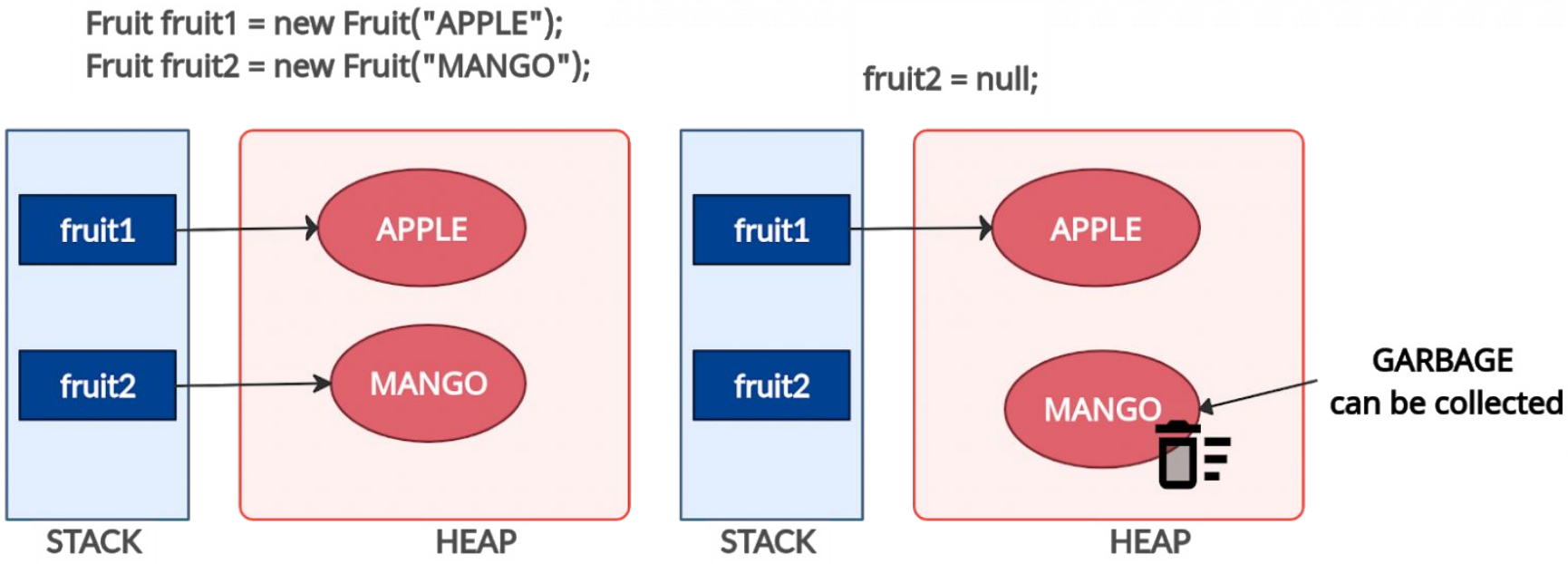
Heap (куча) — часть native memory, выделенная для кучи. Здесь JVM хранит объекты. Это общее пространство для всех потоков приложения. Размер этой области памяти настраивается с помощью параметра -Xms (минимальный размер) и -Xmx (максимальный размер).

Stack (стек) — используется для хранения локальных переменных и стека вызовов метода. Для каждого потока выделяется свой стек.

Metaspace (метаданные) — в этой памяти хранятся метаданные классов и статические переменные. Это пространство также является общими для всех. Так как metaspace является частью native memory, то его размер зависит от платформы. Верхний предел объема памяти, используемой для metaspace, можно настроить с помощью флага MaxMetaspaceSize.

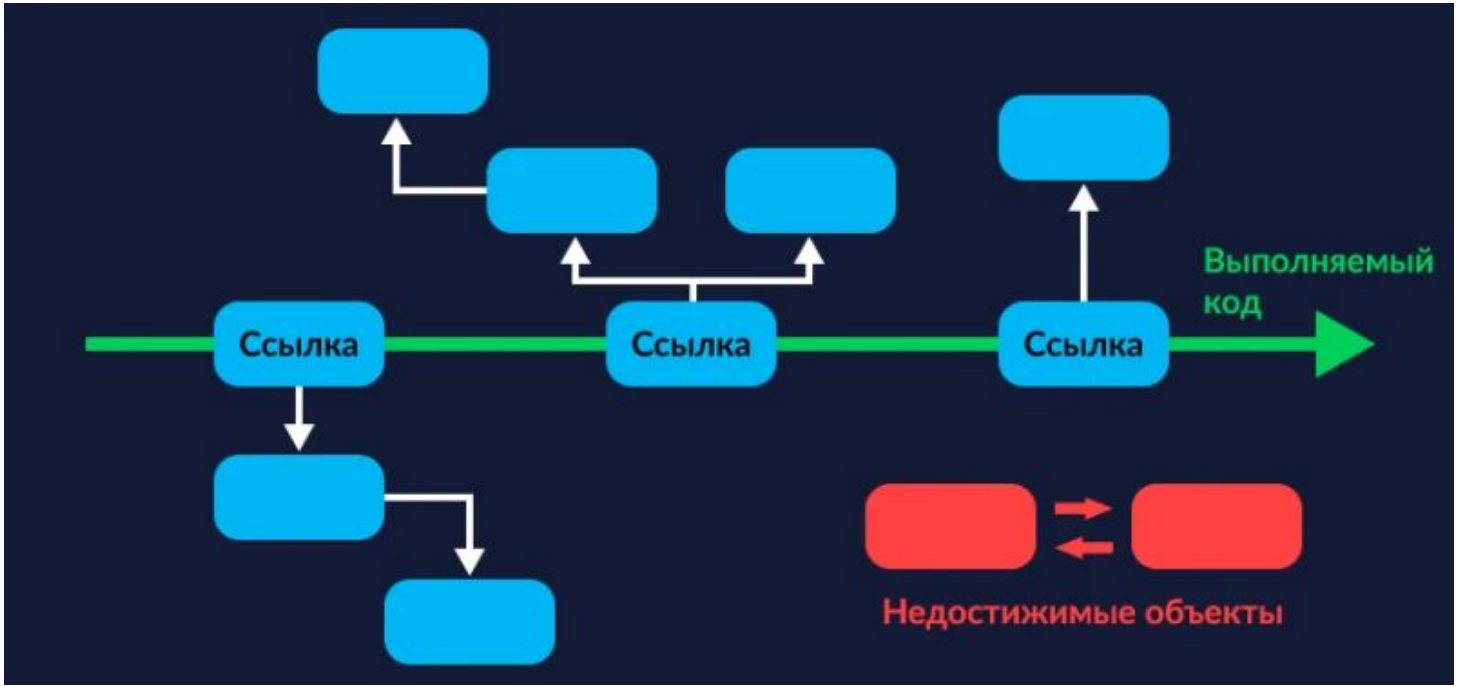
CodeCache (кэш кода) — JIT-компилятор компилирует часто исполняемый код, преобразует его в нативный машинный код и кеширует для более быстрого выполнения. Это тоже часть native memory.

Что такое мусор



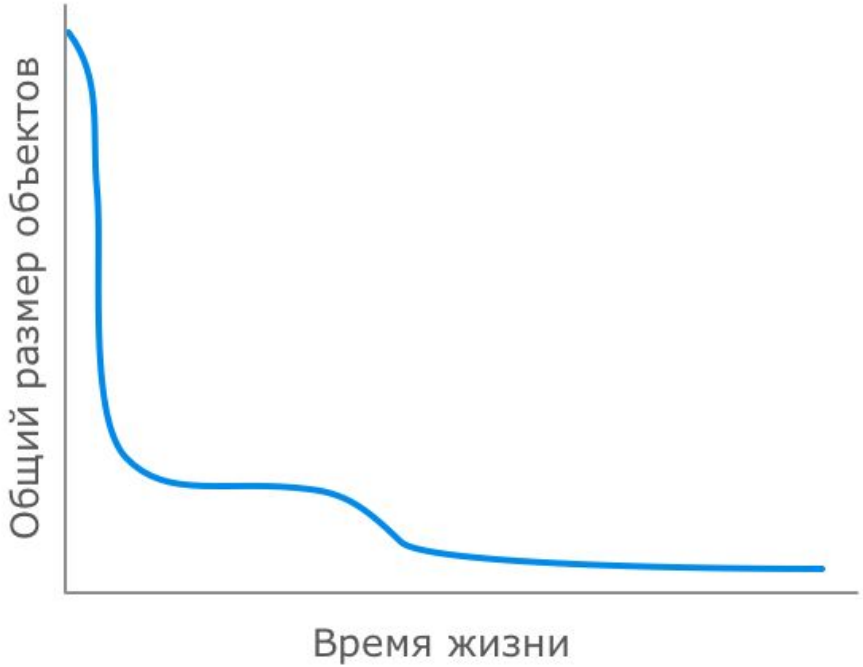
Проблемы сборки мусора

Достижимые и недостижимые объекты



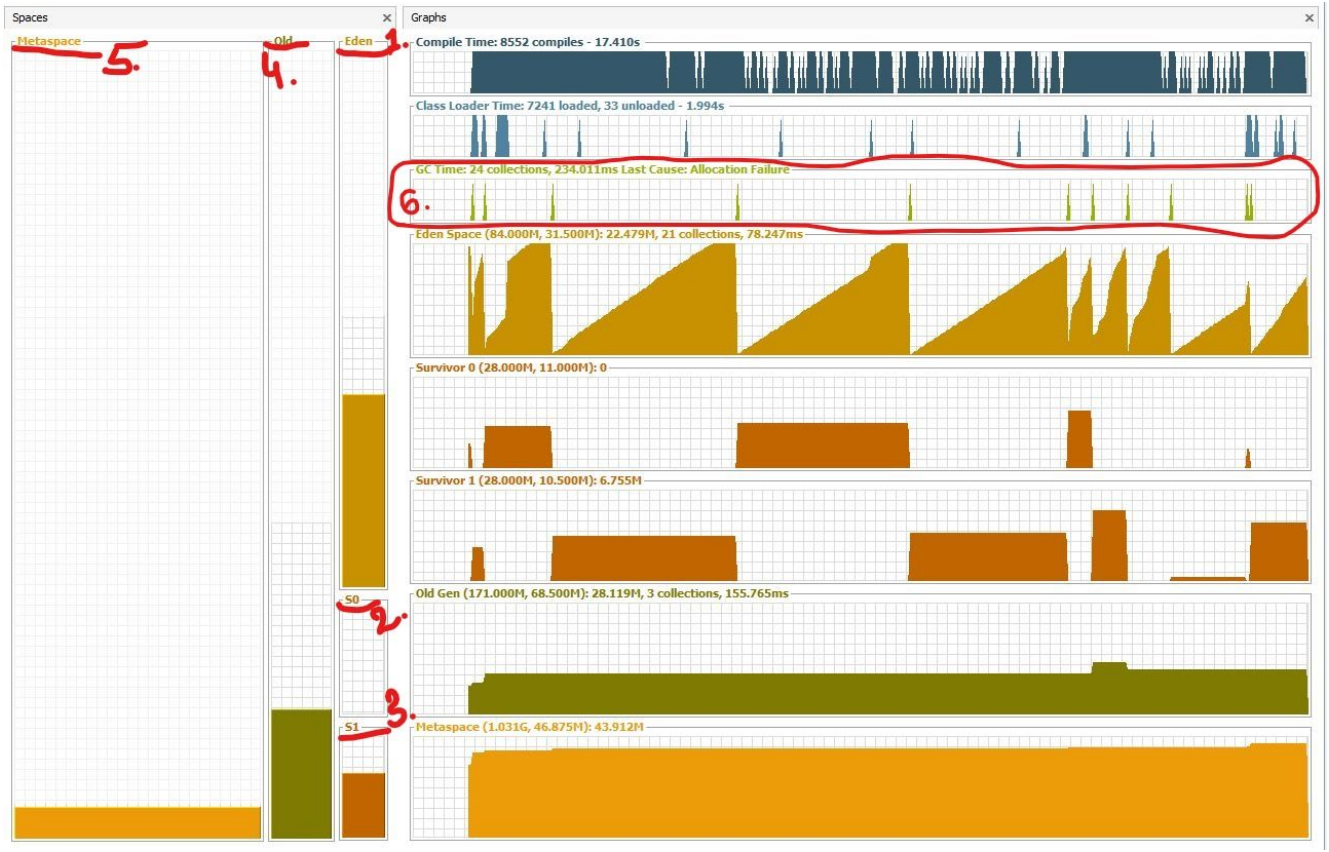
* Объект считается **достижимым**, если на него ссылается другой достижимый объект. Получается такая “цепочка достижимости”. Она начинается при запуске программы и тянется в течение всего времени ее работы.

Теория о поколениях



Подавляющее большинство объектов создаются на очень короткое время, они становятся ненужными практически сразу после их первого использования.

Теория о поколениях



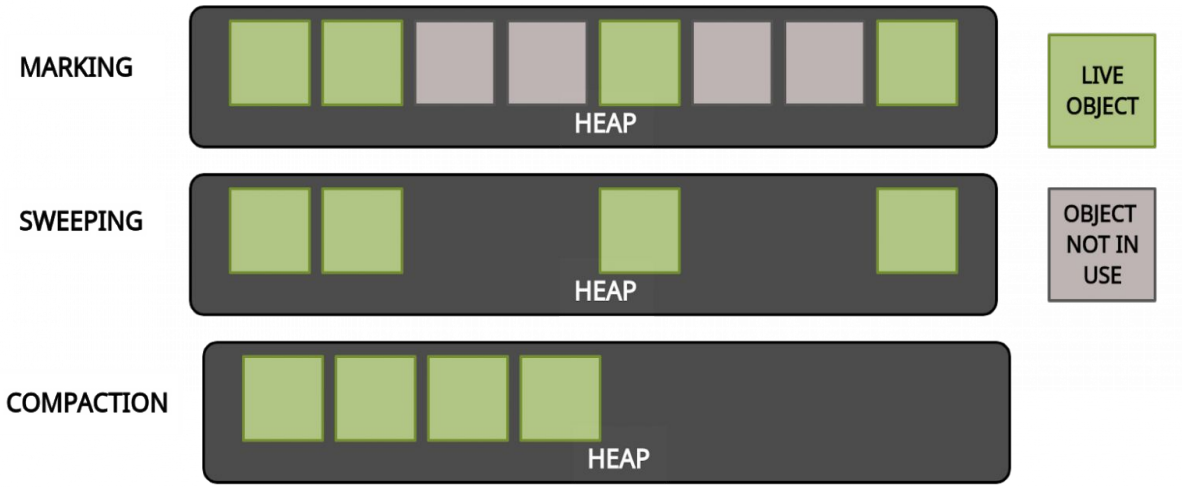
Категории объектов по сроку жизни

- Краткосрочные - используются сразу после создания
- Со средним временем жизни - переходящие по методам
- Долгожители - существующие все время выполнения

Информация о сборщике мусора

-verbose:gc	Включает режим логирования сборок мусора в stdout.
-Xloggc:filename	Указывает имя файла, в который должна логироваться информация о сборках мусора. Имеет приоритет над *-verbose:gc*.
-XX:+PrintGCTimeStamps	Добавляет к информации о сборках временные метки (в виде количества секунд, прошедших с начала работы программы).
-XX:+PrintGCDetails	Включает расширенный вывод информации о сборках мусора.
-XX:+PrintFlagsFinal	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.

Процесс сборки мусора

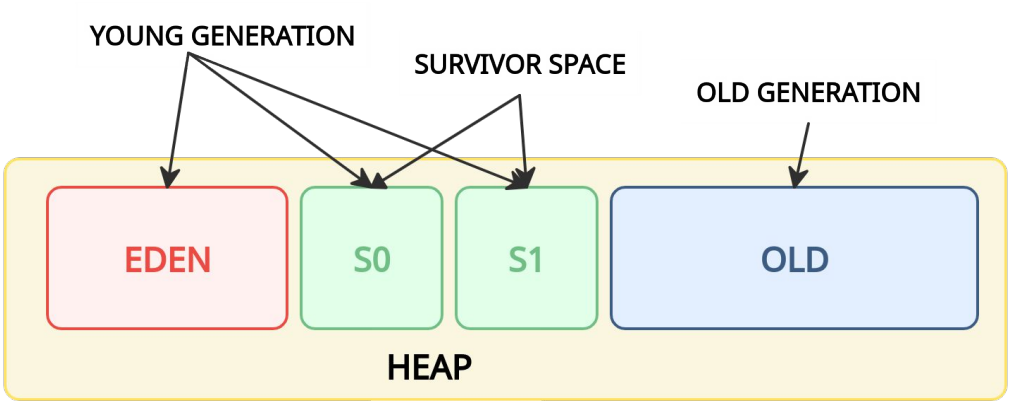


1. Mark (маркировка). На первом этапе GC сканирует все объекты и помечает живые (объекты, которые все еще используются). На этом шаге выполнение программы приостанавливается. Поэтому этот шаг также называется "Stop the World" .

2. Sweep (очистка). На этом шаге освобождается память, занятая объектами, не отмеченными на предыдущем шаге.

3. Compact (уплотнение). Объекты, пережившие очистку, перемещаются в единый непрерывный блок памяти. Это уменьшает фрагментацию кучи и позволяет проще и быстрее размещать новые объекты.

Поколения объектов

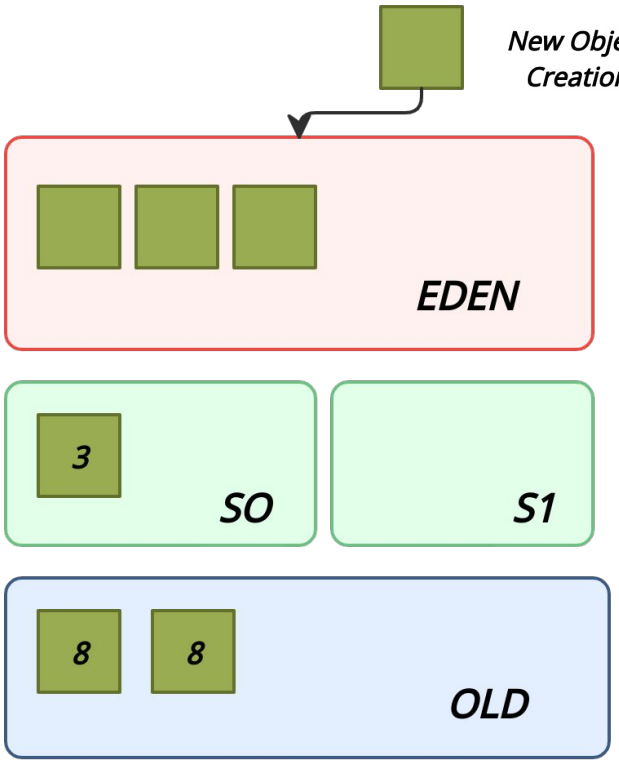


Young Generation (молодое поколение). Здесь создаются новые объекты. Область young generation разделена на три части раздела: Eden (Эдем), S0 и S1 (Survivor Space — область для выживших).

Old Generation (старое поколение). Здесь хранятся давно живущие объекты.

Процессы сборки мусора разделяют на **малую сборку (minor GC)**, затрагивающую только молодое поколение, и **полную сборку (full GC)**, которая может затрагивать оба поколения

Сборка мусора, использующая поколения



1. Новые объекты создаются в области Eden. Области Survivor (S0, S1) на данный момент пустые.

2. Когда область Eden заполняется, происходит минорная сборка мусора (Minor GC). Minor GC — это процесс, при котором операции mark и sweep выполняются для young generation (молодого поколения).

3. После Minor GC живые объекты перемещаются в одну из областей Survivor (например, S0). Мертвые объекты полностью удаляются.

4. По мере работы приложения пространство Eden заполняется новыми объектами. При очередном Minor GC области young generation и S0 очищаются. На этот раз выжившие объекты перемещаются в область S1, и их возраст увеличивается (отметка о том, что они пережили сборку мусора).

5. При следующем Minor GC процесс повторяется. Однако на этот раз области Survivor меняются местами. Живые объекты перемещаются в S0 и у них увеличивается возраст. Области Eden и S1 очищаются.

6. Объекты между областями Survivor копируются определенное количество раз (пока не переживут определенное количество Minor GC) или пока там достаточно места. Затем эти объекты копируются в область Old.

7. Major GC. При Major GC этапы mark и sweep выполняются для Old Generation. Major GC работает медленнее по сравнению с Minor GC, поскольку старое поколение в основном состоит из живых объектов.

Сборка мусора, использующая поколения

Преимущества использования поколений

Minor GC происходит в меньшей части кучи (~ 2/3 от кучи). Этап маркировки эффективен, потому что область небольшая и состоит в основном из мертвых объектов.

Недостатки использования поколений

В каждый момент времени одно из пространств Survivor (S0 или S1) пустое и не используется.

Epsilon GC

Epsilon GC разработан для ситуаций, когда сборка мусора не требуется. Он не выполняет сборку мусора, а использует TLAB (thread-local allocation buffers, локальные буферы выделения потока) для выделения новых объектов — небольших буферов памяти, запрашиваемых отдельными потоками из кучи. Огромные объекты, не помещающиеся в буфер, запрашивают блоки памяти специально для себя.

Когда Epsilon GC исчерпывает ресурсы, генерируется ошибка OutOfMemoryError, и процесс завершается.

К преимуществам Epsilon GC относятся меньшие требования к ресурсам и более быстрое выделение памяти для приложений, которые создают все необходимые им объекты при запуске или запускают недолговечные приложения, не использующие всю выделенную память.

Epsilon GC также может помочь проанализировать требования к ресурсам, которые другие сборщики мусора добавляют в ваше приложение.

Достоинства: Очень быстрый.

Минусы: не очищает объекты :)

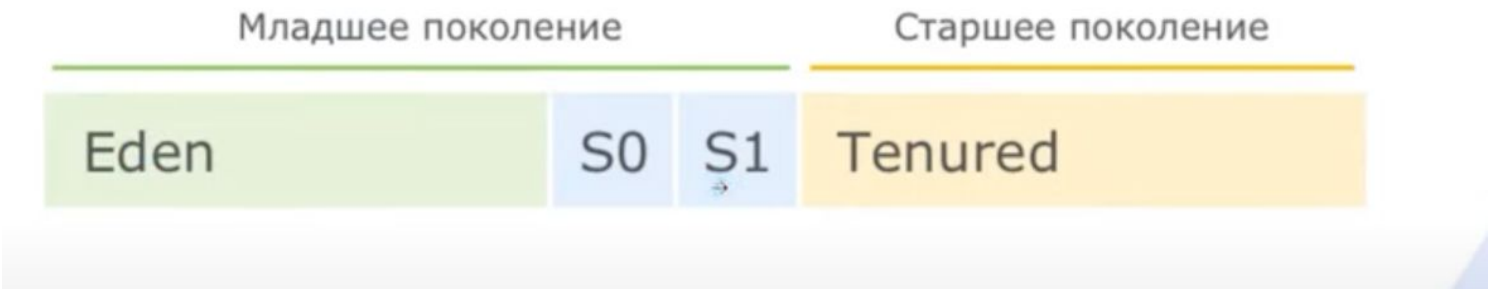
Serial GC

Serial GC — это сборщик мусора в виртуальной машине Java, который использовался с самого начала существования Java. Он полезен для программ с маленькой кучей и работающих на менее мощных машинах.

Основным преимуществом этого сборщика мусора являются его низкие требования к ресурсам, поэтому для выполнения сборки тут достаточно маломощного процессора.

Главным недостатком Serial GC являются длительные паузы при сборке мусора, особенно если речь идет о больших объемах данных. Сплошное Stop-the-word

Serial GC. Принцип работы



Распределение памяти - младшее поколение втрое меньше старшего поколения.

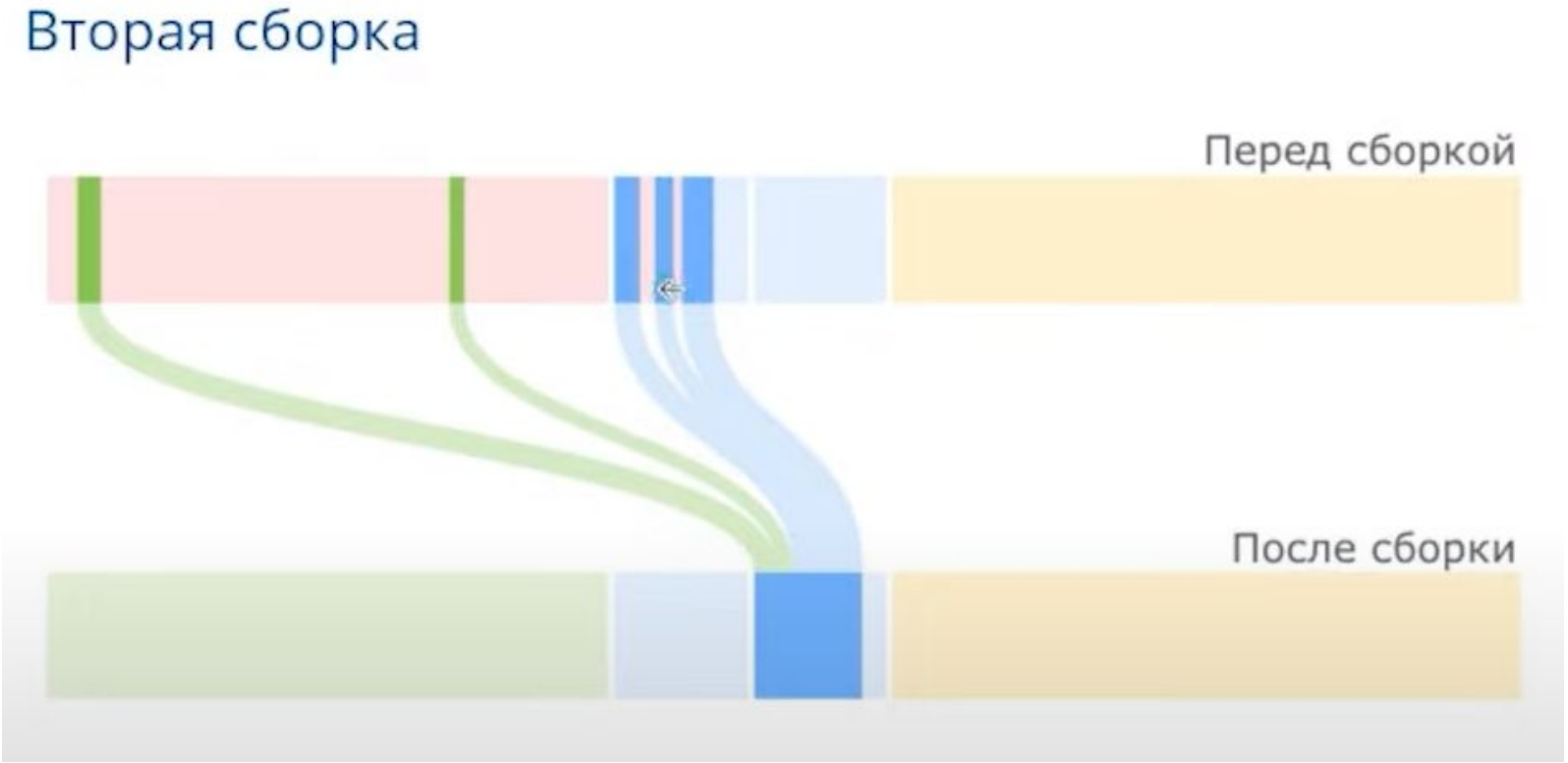
Этот сборщик мусора делит кучу на регионы, куда входят Eden и Survivor. Регион Eden — это пул, из которого изначально выделяется память для большинства объектов. Survivor — это пул, содержащий объекты, пережившие сборку мусора в регионе Eden. По мере заполнения кучи объекты перемещаются между регионами Eden и Survivor.

JVM постоянно отслеживает перемещение объектов в регионы Survivor и выбирает подходящий порог количества таких перемещений, после чего объекты перемещаются в регион старшего поколения (Tenured).

Когда в регионе Tenured не хватает места, в дело вступает полная сборка мусора, работающая с объектами обоих поколений.

Малая сборка

Малая сборка



Полная сборка

Выполняется, когда в старшем поколении не хватает места

Совмещается с малой сборкой

В старшем поколении происходит сборка типа Mark-Sweep-Compact

Parallel GC

Параллельный сборщик мусора (Parallel CG) похож на последовательный конструктор. Он включает параллельную обработку некоторых задач и возможность автоматической настройки параметров производительности.

Parallel GC — это сборщик мусора в виртуальной машине Java, основанный на идеях Serial GC, но с добавлением параллелизма и интеллекта. Если компьютер имеет более одного ядра процессора, старая версия JVM автоматически выбирает Parallel GC.

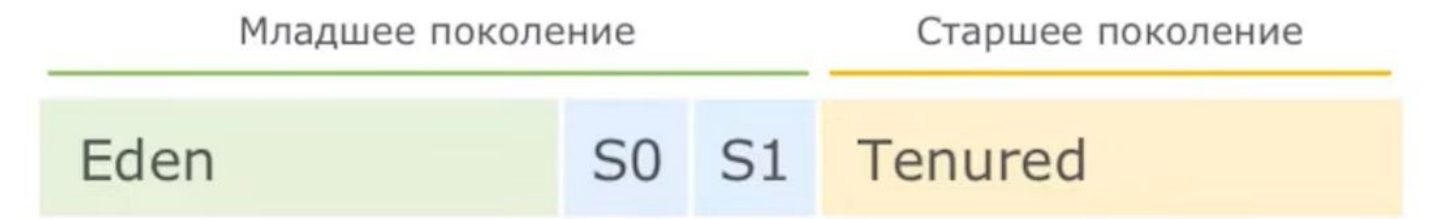
Parallel GC обеспечивает автоматическую настройку параметров производительности и меньшее время паузы для сборки, но есть один незначительный недостаток в виде некоторой фрагментации памяти. Он подходит для большинства приложений, но для более сложных программ лучше выбрать более расширенные реализации сборщиков мусора.

Плюсы: Во многих случаях быстрее, чем Serial GC. Имеет хорошую скорость работы.

Минусы: потребляет больше ресурсов, и паузы могут быть довольно длинными, но мы мы можем настроить максимальную длительность паузы Stop-The-World.

Parallel GC. Принцип работы

Устройство памяти полностью аналогично Serial GC



Куча здесь разделена на те же регионы, что и в Serial GC — Eden, Survivor 0, Survivor 1 и Old Gen (Tenured). Однако в сборке мусора параллельно участвуют несколько потоков, и сборщик может подстраиваться под требуемые параметры производительности.

У каждого потока-коллектора есть область памяти, которую нужно очистить. Parallel GC также имеет настройки, направленные на достижение требуемой эффективности сборки мусора. Сборщик использует статистику предыдущих сборок мусора для настройки параметров производительности в будущих сборках.

Малая сборка

Полная сборка

Аналогична Serial GC, однако может использовать многопоточный режим на этапе уплотнения (compact)

В таком случае каждый поток уплотняет объекты в своей области старшего поколения. Это ускоряет стадию, но приводит к фрагментации памяти.

Concurrent mark sweep (CMS)

Сборщик мусора Concurrent Mark Sweep (CMS) направлен на уменьшение максимальной длительности пауз за счет выполнения некоторых задач по сбору мусора одновременно с потоками приложения. Этот сборщик мусора подходит для управления большими объемами данных в памяти.

Concurrent Mark Sweep (CMS) — это альтернатива Parallel GC в виртуальной машине Java (JVM). Он предназначен для приложений, где требуется доступ к нескольким ядрам процессора и которые чувствительны к паузам Stop-The-World.

Одним из преимуществ CMS является его направленность на минимизацию времени простоя, что имеет решающее значение для многих приложений. Однако он требует жертвы в виде ресурсов процессора и общей пропускной способностью. Его особенностью является то, что он разносит во времени малые и полные сборки мусора, чтобы они совместно не создавали значительных пауз во время работы приложения.

Кроме того, CMS не сжимает объекты в старом поколении, что приводит к фрагментации. Долгие паузы из-за возможных сбоев параллельного режима могут стать неприятным сюрпризом (хотя они случаются нечасто). При наличии достаточного количества памяти CMS удается избежать таких пауз.

Плюсы: Быстрый. Имеет небольшие паузы Stop-The-World.

Минусы: потребляет больше памяти, при недостатке памяти некоторые паузы могут быть длинными. Не очень хорош, если приложение создает много объектов.

Concurrent mark sweep (CMS). Принцип работы

CMS выполняет этапы сборки мусора параллельно с основной программой, что позволяет ей работать без остановки. Он использует ту же организацию памяти, что и сборщики Serial и Parallel, но не ждет заполнения области Tenured перед запуском чистки старого поколения. Вместо этого он работает в фоновом режиме и пытается сохранить компактность региона Tenured.

Concurrent Mark Sweep начинается с начальной фазы маркировки, которая ненадолго останавливает основные потоки приложения и помечает все объекты, доступные из root.

Затем основные потоки приложения возобновляют работу, и CMS начинает поиск всех активных объектов, доступных по ссылкам из отмеченных root-объектов.

После маркировки всех живых объектов сборщик в несколько параллельных потоков очищает память от мертвых объектов.

Старшая сборка

Выполняется постоянно в фоновом режиме, не дожидаясь заполнения старшего поколения.

Называется старшей, так как не совмещена с младшей сборкой.

Допускает наличие “плавающего мусора” - неиспользуемых объектов, которые не были найдены в процессе сборки.

Старшая сборка. Этапы

1. Остановка основных потоков приложения и пометки всех объектов, напрямую доступных из корней.
2. Приложение возобновляет работу, а сборщик параллельно с ним производит поиск всех живых объектов, доступных по ссылкам из тех самых помеченных корневых объектов.
3. Повторная остановка, поиск упущенных объектов
4. Удаление мусора в фоновом режиме

Старшая сборка. Этапы

G1

Garbage-First (G1) считается альтернативой CMS, особенно для серверных приложений, работающих на многопроцессорных серверах и управляющих большими наборами данных.

Сборщик мусора G1 преобразует память в несколько регионов одинакового размера, за исключением огромных регионов (которые создаются путем слияния обычных регионов для размещения массивных объектов). Регионы не обязательно должны быть организованы в ряд и могут менять принадлежность к своему поколению.

Сборщик мусора G1 считается более точным, чем сборщик CMS, в прогнозировании размеров пауз и лучше распределяет сбор мусора во времени, чтобы предотвратить длительные простои приложений, особенно при больших размерах кучи. Он также не фрагментирует память, как сборщик CMS.

Однако сборщику G1 требуется больше ресурсов процессора для работы параллельно с основной программой, что снижает пропускную способность приложения.

Плюсы: работает лучше, чем CMS. Имеет более короткие паузы.

Минусы: потребляет больше ресурсов процессора. Также он потребляет больше памяти, если у нас много довольно больших объектов (более 500 КБ), потому что он помещает такие объекты в один регион (1-32 МБ).

G1. Принцип работы

Небольшие чистки выполняются периодически для младшего поколения и перемещения объектов в регионы Survivor или апгрейда их до старшего поколения с переводом в Tenured.

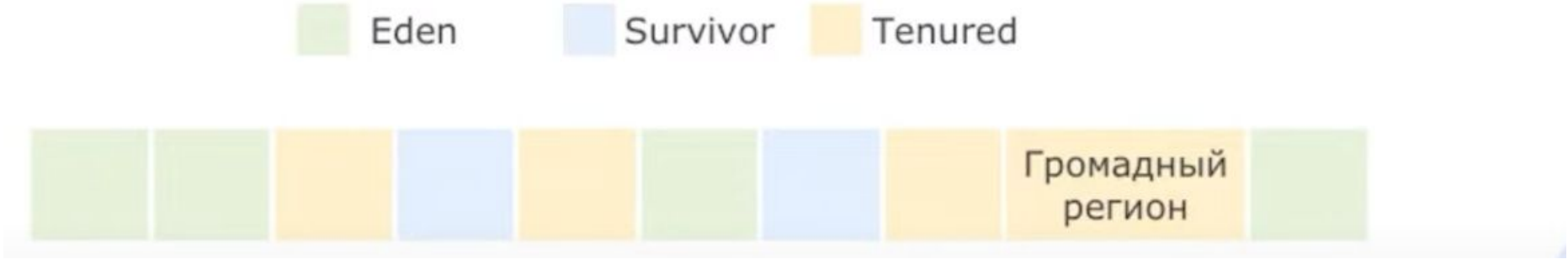
Очистка выполняется только в тех регионах, где нужно избежать превышения желаемого времени. Сборщик сам прогнозирует и выбирает для очистки регионы с наибольшим количеством мусора.

Полная очистка используют цикл маркировки для создания списка живых объектов, который работает параллельно с основным приложением. После цикла маркировки G1 переключается на запуск смешанных чисток, которые добавляют регионы старшего поколения к набору регионов младшего поколения, подлежащих очистке.

Для получения списка живых объектов используется алгоритм Snapshot-At-The-Beginning (SATB), то есть в список живых попадают все объекты, которые были таковыми на момент начала работы алгоритма, плюс все объекты, созданные за время его выполнения, то есть допускается наличие “плавающего мусора”

G1. Принцип работы

Куча разбита на регионы разных поколений, которых не более 2048



Малая сборка

Использует те же принципы, что и Parallel/CMS

Производится только на некоторых регионах, а не на всем поколении

Сборщик выбирает регионы, в которых по его мнению находится наибольшее количество мусора

Большая (смешанная) сборка

Начинается с цикла пометки:

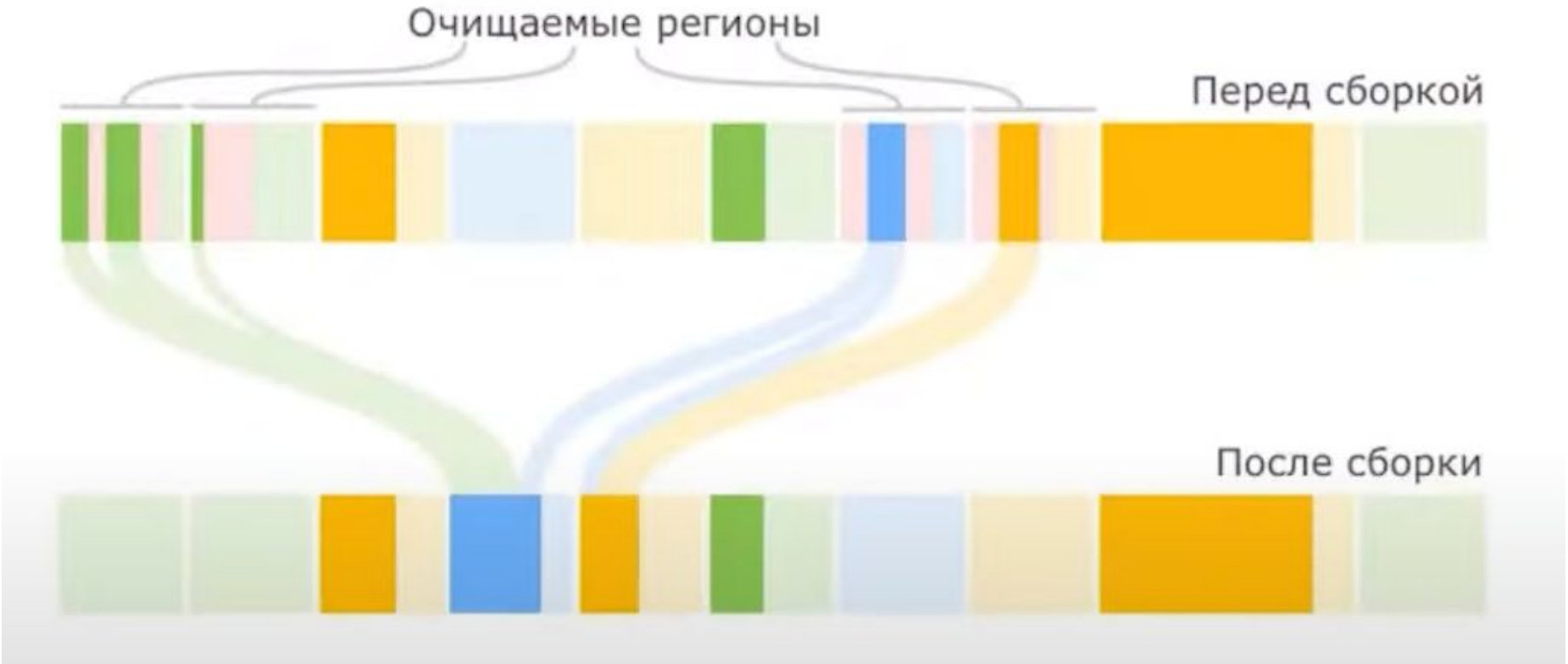
1. Initial mark. Пометка корней (с остановкой основного приложения) с использованием информации, полученной из малых сборок
2. Concurrent marking. Пометка всех живых объектов в куче в нескольких потоках, параллельно с работой основного приложения
3. Remark. Дополнительный поиск неучтенных ранее живых объектов (с остановкой основного приложения)
4. Cleanup. Очистка вспомогательных структур учета ссылок на объекты и поиск пустых регионов, которые уже можно использовать для размещения новых объектов

Большая (смешанная) сборка

После пометки, во время проведения малых сборок к ним добавляется несколько регионов старого поколения (поэтому сборка смешанная) до тех пор, пока размер старшего поколения не достигнет желаемого, при этом не допуская превышения максимального времени сборки.

После достижения этого порога G1 снова переходит в режим малых сборок.

Большая (смешанная) сборка



ZGC

ZGC может поддерживать паузы на уровне менее миллисекунды даже при работе с огромными объемами данных. ZGC — это сборщик мусора, разработанный Oracle для Java, который предназначен для обеспечения высокой пропускной способности и низкой задержки при обработке больших куч (до 16 ТБ).

ZGC основан на принципах виртуальной памяти и использует разные цвета маркировки для отслеживания состояния объектов во время сборки мусора.

Плюсы: паузы менее миллисекунды, даже в больших кучах, что очень полезно для приложений, требующих короткого времени обработки запросов. Он работает с очень большими кучами с хорошей пропускной способностью. ZGC может сжимать динамическую память во время сборки мусора.

Минусы: Высокая загрузка процессоров и ощутимые требования к производительности, что может замедлить время запуска приложений.

Виртуальная память

Разные страницы виртуальной памяти могут указывать на одну и ту же страницу физической памяти.

Это, в частности, означает, что один и тот же объект в физической памяти может иметь несколько разных виртуальных адресов.

Цветные указатели ZGC

Каждый указатель в ZGC содержит доп. метаданные, помимо обычного адреса объекта. Комбинация этих флагов определяет состояние указателя, которое при описании ZGC называется его “цветом”

Барьеры

Еще одной особенностью GC является использование барьеров во время конкурентных фаз сборки мусора (когда сборщик работает одновременно с приложением, не останавливая его работу)

Например:

При работе приложение обращается к объекту, который нужно перенести в другую область памяти. В таком случае, перед тем как обратиться к нему, основной поток переносит его в новый регион.

Этапы сборки

Mark:

1. Pause Mark Start
2. Concurrent Map
3. Pause Mark End

Relocate:

1. Concurrent Prepare for Relocate
2. Pause Relocate Start
3. Concurrent Relocate
4. Concurrent Remap

Поиск живых объектов. Шаг 1

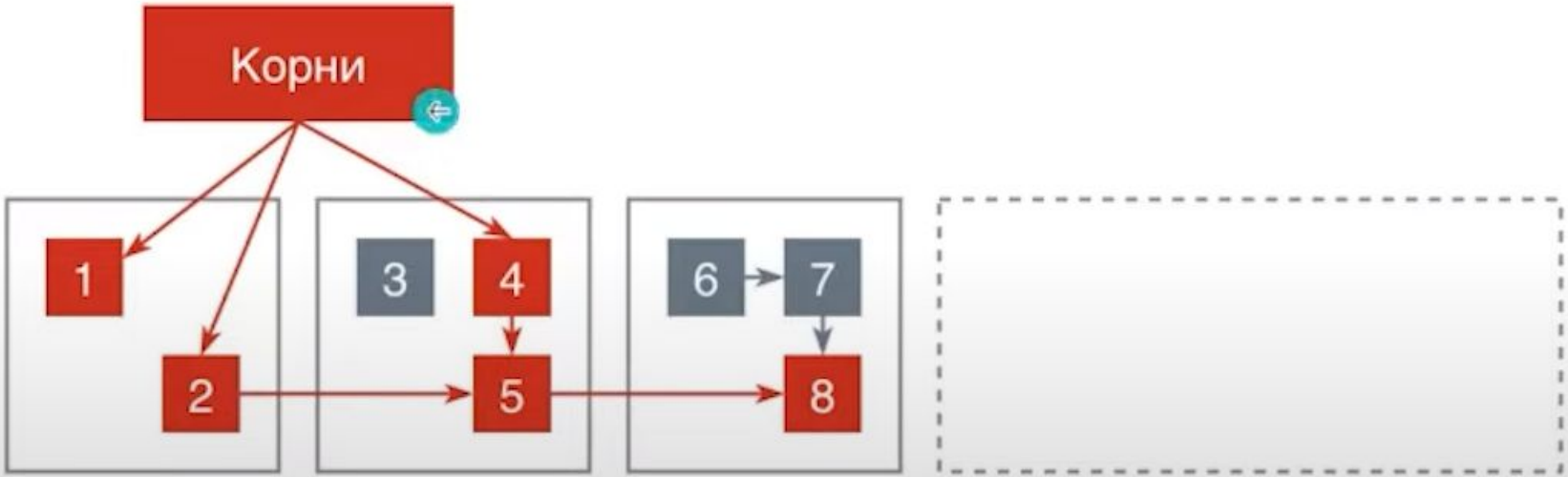
Поиск живых объектов. Шаг 2

Сборщик продолжает поиск живых объектов, доступных из корней, но уже в конкурентном режиме.

Так как во время этой фазы приложение работает и может создавать новые объекты, в этот период активно используются барьеры, которые красят все указатели, по которым в это время происходит доступ к объектам.

Поиск живых объектов. Шаг 3

STW, обработка особых кейсов. В частности, soft- и weak-references



Перемещение. Шаг 1

Перемещение. Шаг 2

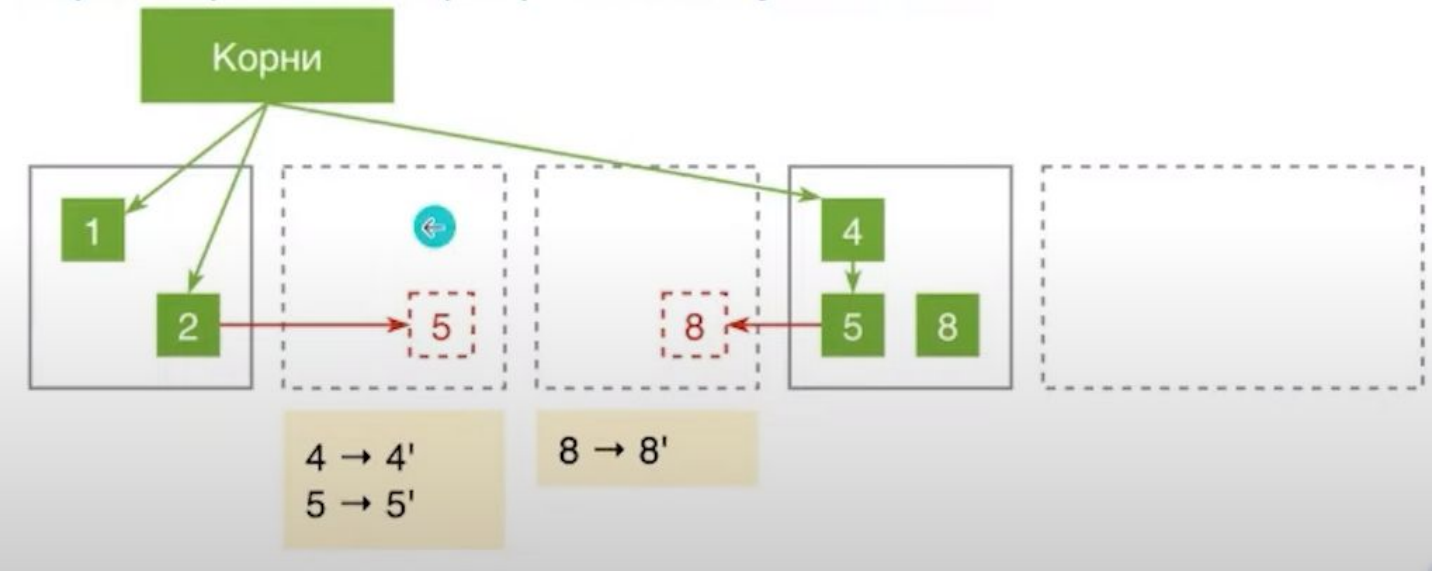
Перемещение начинается с объектов, достижимых из корней, и производится в рамках паузы SWT. Объект переносится в новый блок памяти, корневой указатель на него красится и сборщик запоминает соответствие старого и нового адреса в специальной таблице переадресации

Перемещение. Шаг 3

Concurrent Relocate - предыдущий шаг распространяется на всю оставшуюся кучу в конкурентном режиме. При этом активно используются барьеры: они не только раскрашивают указатели, но и физически перемещают объекты, если обнаружили доступ к объекту, который подлежит перемещению

Перемещение. Шаг 3

Если в течение этой фазы приложение попыталось получить объект 5 по устаревшему указателю из объекта 4, то барьер корректно перенаправит и перекрасит этот указатель



Shenandoah GC

Shenandoah GC — еще один сборщик мусора с короткими паузами независимо от размера кучи. Этот сборщик мусора разработан компанией Red Hat. Он предназначен для минимизации времени, затрачиваемого приложением на сборку мусора.

Как и ZGC, это параллельный сборщик, что означает, что он работает во время работы приложения, сводя к минимуму паузы.

Shenandoah GC использует “указатели пересылки” для перемещения объектов во время сборки мусора. Также он имеет метод под названием “устранение барьера нагрузки” для повышения производительности.

Плюсы: Shenandoah GC может достичь короткого времени паузы, часто менее 10 мс, даже для массивных куч. Хорошая пропускная способность.

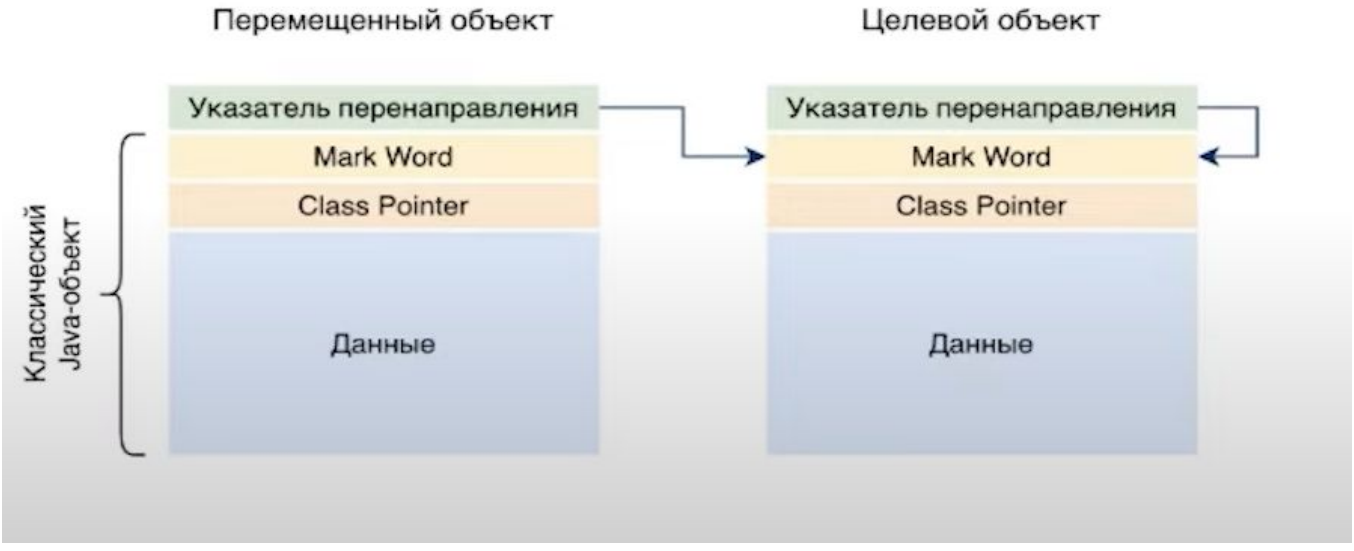
Минусы: высокая загрузка процессора и сложность в работе при больших нагрузках.

Shenandoah GC. Принцип работы

Shenandoah имеет много общего с ZGC. Рассмотрим их отличия

Указатели Брукса

Расходы на хранение каждого объекта увеличиваются и перед упомянутым заголовком добавляется еще указатель перенаправления.

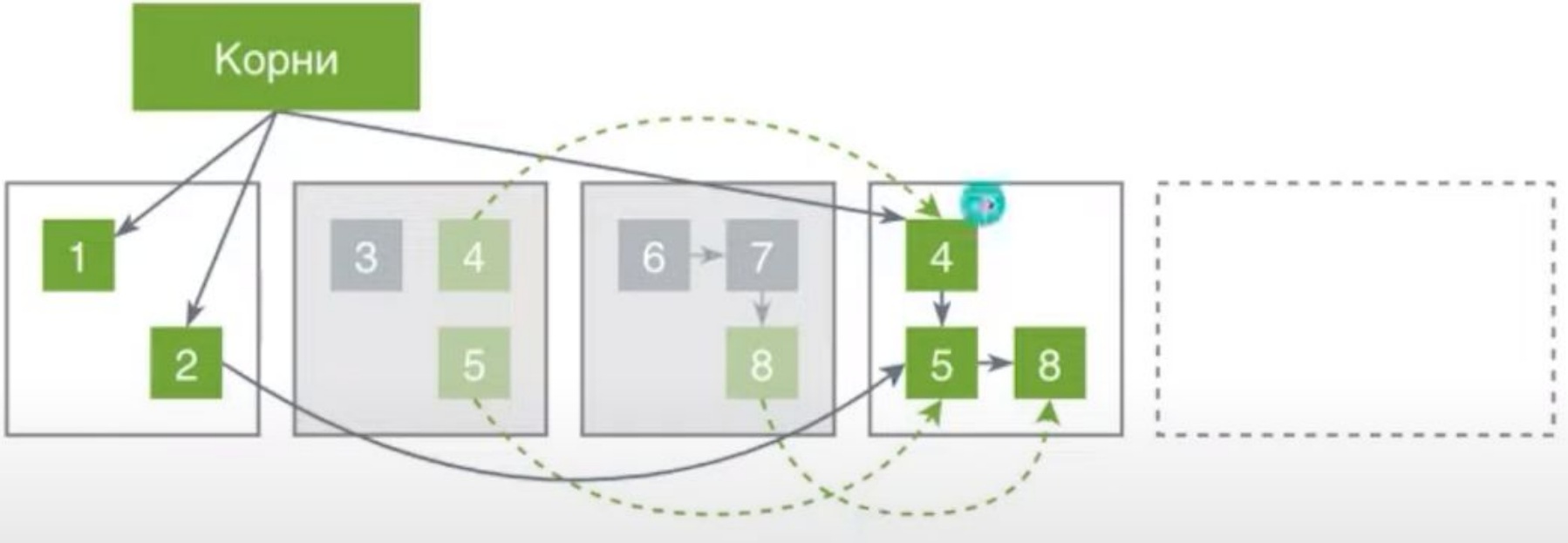


Организация кучи

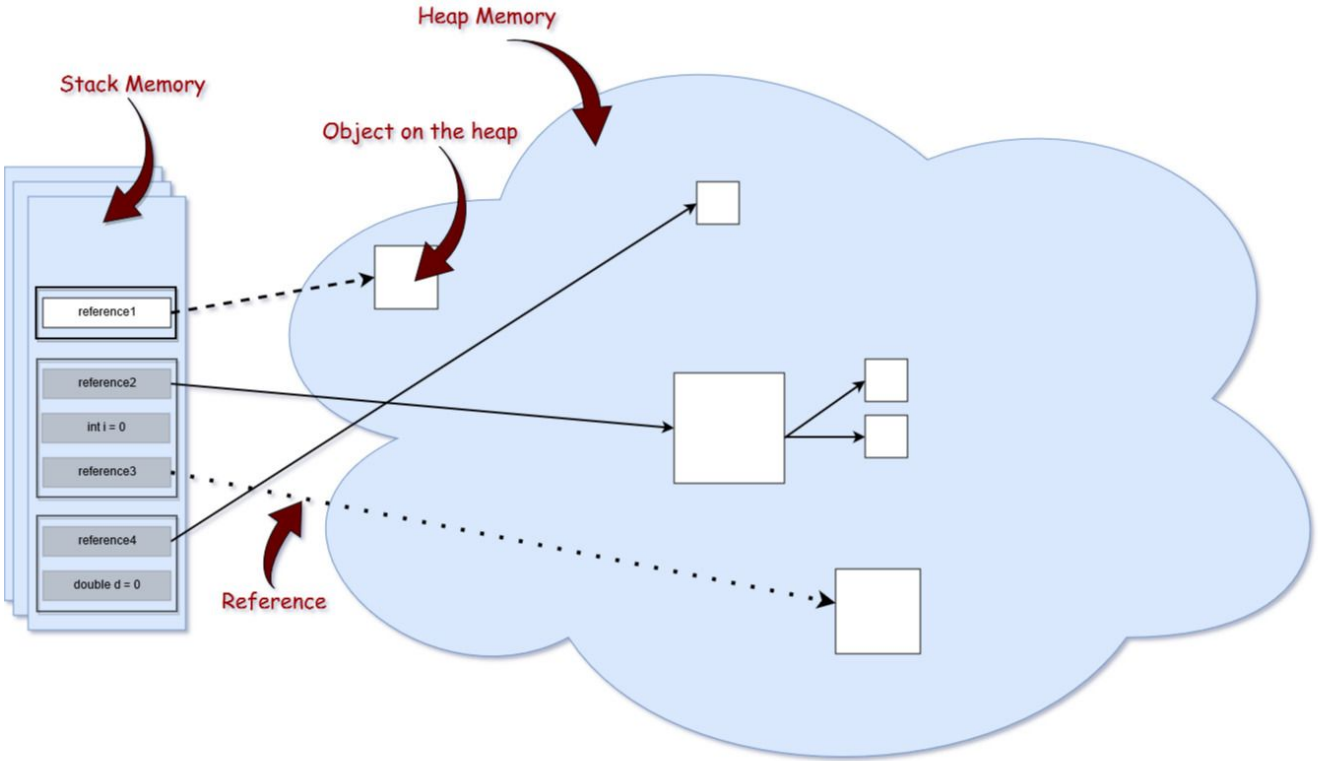
Shenandoah ганизует кучу, разбивая ее на большое количество регионов равных размеров. В каждый момент времени регион может:

- размещать живые объекты и не подлежать очистке
- размещать живые объекты и подлежать очистке
- быть забронированным под перемещение живых объектов из очищаемых регионов
- не использоваться

Перемещение объектов



Типы ссылок в Java



Stack

Стековая память отвечает за хранение ссылок на объекты кучи и за хранение типов значений (также известных в Java как примитивные типы), которые содержат само значение, а не ссылку на объект из кучи.

Кроме того, переменные в стеке имеют определенную видимость, также называемую областью видимости. Используются только объекты из активной области. Например, предполагая, что у нас нет никаких глобальных переменных (полей) области видимости, а только локальные переменные, если компилятор выполняет тело метода, он может получить доступ только к объектам из стека, которые находятся внутри тела метода. Он не может получить доступ к другим локальным переменным, так как они не входят в область видимости. Когда метод завершается и возвращается, верхняя часть стека выталкивается, и активная область видимости изменяется.

Возможно, вы заметили, что на картинке выше отображено несколько стеков памяти. Это связано с тем, что стековая память в Java выделяется для каждого потока. Следовательно, каждый раз, когда поток создается и запускается, он имеет свою собственную стековую память и не может получить доступ к стековой памяти другого потока.

Heap

Эта часть памяти хранит в памяти фактические объекты, на которые ссылаются переменные из стека. Например, давайте проанализируем, что происходит в следующей строке кода:

```
StringBuilder builder = new StringBuilder();
```

Ключевое слово `new` несет ответственность за обеспечение того, достаточно ли свободного места на куче, создавая объект типа StringBuilder в памяти и обращаясь к нему через «builder» ссылки, которая попадает в стек.

Для каждого запущенного процесса JVM существует только одна область памяти в куче. Следовательно, это общая часть памяти независимо от того, сколько потоков выполняется. На самом деле структура кучи немного отличается от того, что показано на картинке выше. Сама куча разделена на несколько частей, что облегчает процесс сборки мусора.

Максимальные размеры стека и кучи не определены заранее - это зависит от работающей JVM машины. Позже в этой статье мы рассмотрим некоторые конфигурации JVM, которые позволят нам явно указать их размер для запускаемого приложения.

Типы ссылок

В языке Java используются разные типы ссылок: сильные, слабые, мягкие и фантомные ссылки. Разница между типами ссылок заключается в том, что объекты в куче, на которые они ссылаются, имеют право на сборку мусора по различным критериям. Рассмотрим подробнее каждую из них.

Сильные ссылки

Это самые популярные ссылочные типы, к которым мы все привыкли. В приведенном выше примере со StringBuilder мы фактически храним сильную ссылку на объект из кучи. Объект в куче не удаляется сборщиком мусора, пока на него указывает сильная ссылка или если он явно доступен через цепочку сильных ссылок.

Слабые ссылки

Попросту говоря, слабая ссылка на объект из кучи, скорее всего, не сохранится после следующего процесса сборки мусора. Слабая ссылка создается следующим образом:

```
WeakReference<StringBuilder> reference = new WeakReference<>(new StringBuilder());
```

Мягкие ссылки

Эти типы ссылок используются для более чувствительных к памяти сценариев, поскольку они будут собираться сборщиком мусора только тогда, когда вашему приложению не хватает памяти. Следовательно, пока нет критической необходимости в освобождении некоторого места, сборщик мусора не будет касаться легко доступных объектов. Java гарантирует, что все объекты, на которые имеются мягкие ссылки, будут очищены до того, как будет выдано исключение `OutOfMemoryError`. В документации Javadocs говорится, что *«все мягкие ссылки на мягко достижимые объекты гарантированно очищены до того, как виртуальная машина выдаст OutOfMemoryError».*

Подобно слабым ссылкам, мягкая ссылка создается следующим образом:

```
SoftReference<StringBuilder> reference = new SoftReference<>(new StringBuilder());
```

Фантомные ссылки

Используется для планирования посмертных действий по очистке, поскольку мы точно знаем, что объекты больше не живы. Используется только с очередью ссылок, поскольку `.get()`метод таких ссылок всегда будет возвращаться `null`. Эти типы ссылок считаются предпочтительными для финализаторов.

```
ReferenceQueue<StringBuilder> queue = new ReferenceQueue<>();
PhantomReference<StringBuilder> reference = new PhantomReference<>(new StringBuilder(),
queue);
```

ReferenceQueue — это место, куда помещаются ссылки на объекты для освобождение памяти.

Фантомные ссылки — это безопасный способ узнать, что объект удален из памяти. Например, рассмотрим приложение, которое имеет дело с большими изображениями. Предположим, что мы хотим загрузить изображение в память, когда оно уже находится в памяти, которая готова для сборки мусора. В этом случае мы хотим подождать пока сборщик мусора убьет старое изображение и только потом загружать в память новое.

Здесь PhantomReference является гибким и безопасным выбором. Ссылка на старое изображение будет передана в ReferenceQueue после уничтожения старого объекта изображения. Получив эту ссылку, мы можем загрузить новое изображение в память.

Советы

- Чтобы минимизировать объем памяти, максимально ограничьте область видимости переменных. Помните, что каждый раз, когда выскакивает верхняя область видимости из стека, ссылки из этой области теряются, и это может сделать объекты пригодными для сбора мусора.
- Явно устанавливайте в `null` устаревшие ссылки. Это сделает объекты, на которые ссылаются, подходящими для сбора мусора.
- Избегайте финализаторов (finalizer). Они замедляют процесс и ничего не гарантируют. Фантомные ссылки предпочтительны для работы по очистке памяти.
- Не используйте сильные ссылки там, где можно применить слабые или мягкие ссылки. Наиболее распространенные ошибки памяти - это сценарии кэширования, когда данные хранятся в памяти, даже если они могут не понадобиться.
- JVisualVM также имеет функцию создания дампа кучи в определенный момент, чтобы вы могли анализировать для каждого класса, сколько памяти он занимает.
- Настройте JVM в соответствии с требованиями вашего приложения. Явно укажите размер кучи для JVM при запуске приложения. Процесс выделения памяти также является дорогостоящим, поэтому выделите разумный начальный и максимальный объем памяти для кучи. Если вы знаете его, то не имеет смысла начинать с небольшого начального размера кучи с самого начала, JVM расширит это пространство памяти. Указание параметров памяти выполняется с помощью следующих параметров:
  - Начальный размер кучи `-Xms512m`- установите начальный размер кучи на 512 мегабайт.
  - Максимальный размер кучи `-Xmx1024m`- установите максимальный размер кучи 1024 мегабайта.
  - Размер стека потоков `-Xss1m`- установите размер стека потоков равным 1 мегабайту.
  - Размер поколения `-Xmn256m`- установите размер поколения 256 мегабайт.
- Если приложение Java выдает ошибку `OutOfMemoryError`и вам нужна дополнительная информация для обнаружения утечки, запустите процесс с `-XX:HeapDumpOnOutOfMemory`параметром, который создаст файл дампа кучи, когда эта ошибка произойдет в следующий раз.

Программирование с утверждениями

Утверждение (assert) — это оператор (statement) языка программирования Java, который позволяет вам проверить свои предположения о программе.

Каждое утверждение содержит логическое выражение, которое, по вашему мнению, будет верным в момент выполнения. В противном случае, система выбросит исключение.

assert

Оператор assert имеет две формы. Первая, более простая форма? где *Expression1* – это boolean-выражение. Когда система проверяет утверждение, она вычисляет *Expression1* и если оно ложно (равно *false*), то система бросает *java.lang.AssertionError* без детализированного сообщения об ошибке.

Вторая форма утверждения – это: где:

Expression1 – логическое выражение.

Expression2 – это выражение, которое имеет значение (не может быть вызовом *void* метода).

Используйте эту версию assert-оператора для того, чтобы предоставить детализированное сообщение об ошибке. Система передает значение Expression2 в соответствующий конструктор *AssertionError* для использования строкового представления значения в качестве подробного сообщение об ошибке.

```
assert Expression1;
```

```
assert Expression1 : Expression2;
```

assert

Целью такого сообщения в случае *Expression2* об ошибке является фиксирование и сообщение сведений о причине нарушения утверждения. Сообщение должно позволять диагностировать и, в конечном счете, устранить ошибку, которая привела утверждение к сбою. Обратите внимание, что сообщение – это НЕ сообщение об ошибке для пользователя. В общем, нет необходимости делать эти сообщения понятными сами по себе, или интернационализировать их (переводить на язык пользователя).

```
public static void f(){
  int x, y;
  // некоторые вычисления
  assert x > y;
}
```

```
public static void f(){
  int x, y;
  // некоторые вычисления
  assert x > y : "x: " + x + ", y: " + y;
}
```

Советы

Не используйте утверждения для проверки аргументов публичных (public) методов.

Как правило, проверка аргумента – это часть опубликованной спецификаций (или контракта) метода, и это поведение должно соблюдаться в независимости включены или выключены утверждения. Другой проблемой использования утверждений для проверки аргументов является то, что ошибочные аргументы должны привести к соответствующем исключениям при выполнении (таким как *IllegalArgumentException*, *IndexOutOfBoundsException* или *NullPointerException*). Сбой утверждения не выдаст соответствующего исключения.

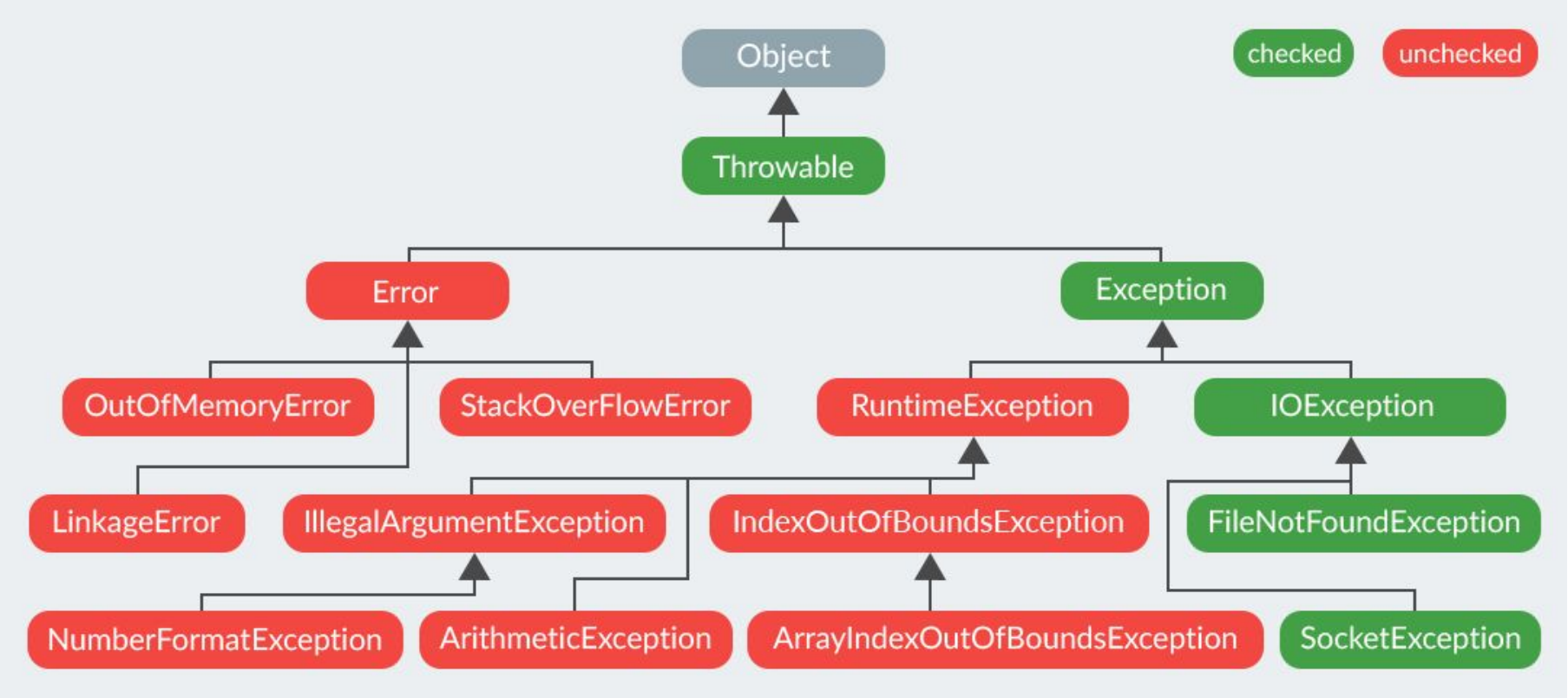
Советы

Не используйте утверждения для выполнения задачи, результат которой ожидается вашим приложением для корректной работы.

Утверждения могут быть отключены и программы не должны исходить из того, что содержащееся в утверждении логическое выражение будет вычисляться (evaluate). Нарушение этого правила может иметь тяжелые последствия. Предположим, необходимо удалить все нулевые элементами из списка имен, зная, что список содержит как минимум один или несколько таких элементов. Было бы неправильно сделать так:

```
// Не правильно! - Действие содержится в утверждении
assert names.remove(null);
// Правильно! - действие выполняется перед проверкой утверждения
boolean nullsRemoved = names.remove(null); // Работает в независимости включены ли утверждения.
assert nullsRemoved;
```

Исключения


Object
checked
unchecked
Throwable
Error
Exception
OutOfMemoryError
StackOverFlowError
RuntimeException
IOException
LinkageError
IllegalArgumentException
IndexOutOfBoundsException
FileNotFoundException
NumberFormatException
ArithmeticException
ArrayIndexOutOfBoundsException
SocketException

Исключения

- Выбрасываются явно оператором throw
- Выбрасываются вызванным методом или конструктором (непроверяемые или явно добавленные)
- Выбрасываются виртуальной машиной (только непроверяемые)

try