**Сборщик мусора(garbage collector)**

**Сборка мусора** — это процесс восстановления заполненной памяти среды выполнения путем уничтожения неиспользуемых объектов.

В Java сборка мусора происходит автоматически в течение всего времени работы программы. Это устраняет необходимость выделения памяти и, следовательно, позволяет избежать утечек.

Сборка мусора в Java — это процесс, с помощью которого программы Java автоматически управляют памятью. Java-программы компилируются в байт-код, который запускается на виртуальной машине Java (JVM).

Когда Java-программы выполняются на JVM, объекты создаются в куче, которая представляет собой часть памяти, выделенную для них.

Пока Java-приложение работает, в нем создаются и запускаются новые объекты. В конце концов, некоторые объекты перестают быть нужны. Можно сказать, что в любой момент времени память кучи состоит из двух типов объектов.

**Живые** — эти объекты используются, на них ссылаются откуда-то еще.

**Мертвые** — эти объекты больше нигде не используются, ссылок на них нет.

Сборщик мусора находит эти неиспользуемые объекты и удаляет их, чтобы освободить память.

**Основная цель сборки мусора** — освободить память кучи, уничтожив объекты, которые не содержат ссылку. Когда на объект нет ссылки, предполагается, что он мертв и больше не нужен. Таким образом, память, занятая объектом, может быть восстановлена.

Есть несколько способов убрать ссылки на объект и сделать его кандидатом на сборку мусора. Вот некоторые из них.

1. Сделать ссылку нулевой

Student student = new Student();

student = null;

1. Назначить ссылку другому объекту

Student studentOne = new Student();

Student studentTwo = new Student();

studentOne = studentTwo; // Т

1. Использовать анонимный объект

register(new Student());

**Источники для сбора мусора**

Сборщики мусора работают с концепцией корней сбора мусора (GC Roots) для идентификации живых и мертвых объектов.

Примеры таких корней.

Классы, загружаемые системным загрузчиком классов (не пользовательские загрузчики классов).

Живые потоки.

Локальные переменные и параметры выполняемых в данный момент методов.

Локальные переменные и параметры методов JNI.

Глобальная ссылка на JNI.

Объекты, применяемые в качестве монитора для синхронизации.

Объекты, удерживаемые из сборки мусора JVM для своих целей.

**Этапы сборки мусора**

1. Пометка объектов как живых

На этом этапе GC (сборщик мусора) идентифицирует все живые объекты в памяти путем обхода графа объектов.

Когда GC посещает объект, то помечает его как доступный и, следовательно, живой. Все объекты, недоступные из корней GC, рассматриваются как кандидаты на сбор мусора.

1. Зачистка мертвых объектов

После фазы разметки пространство памяти занято либо живыми (посещенными), либо мертвыми (не посещенными) объектами. Фаза зачистки освобождает фрагменты памяти, которые содержат эти мертвые объекты.

1. Мертвые объекты, которые были удалены во время предыдущей фазы, не обязательно находились рядом друг с другом. Поэтому вы рискуете получить фрагментированное пространство памяти.

Память можно уплотнить, когда сборщик мусора удалит мертвые объекты. Оставшиеся будут располагаться в непрерывном блоке в начале кучи.

Процесс уплотнения облегчает последовательное выделение памяти для новых объектов.

**Сбор мусора поколениям**

Сборщики мусора в Java реализуют стратегию сбора мусора поколений, которая классифицирует объекты по возрасту.

Область памяти кучи в JVM разделена на три секции:

1)Молодое поколение

Вновь созданные объекты начинаются в молодом поколении. Молодое поколение далее подразделяется на две категории.

Пространство Эдема — все новые объекты начинают здесь, и им выделяется начальная память.

Пространства выживших (FromSpace и ToSpace) — объекты перемещаются сюда из Эдема после того, как пережили один цикл сборки мусора.

Процесс, когда объекты собираются в мусор из молодого поколения, называется малым событием сборки мусора.

Когда пространство Эдема заполнено объектами, выполняется малая сборка мусора. Все мертвые объекты удаляются, а все живые — перемещаются в одно из оставшихся двух пространств. Малая GC также проверяет объекты в пространстве выживших и перемещает их в другое (следующее) пространство выживших.

Возьмем в качестве примера следующую последовательность.

В Эдеме есть объекты обоих типов (живые и мертвые).

Происходит малая GC — все мертвые объекты удаляются из Эдема. Все живые объекты перемещаются в пространство-1 (FromSpace). Эдем и пространство-2 теперь пусты.

Новые объекты создаются и добавляются в Эдем. Некоторые объекты в Эдеме и пространстве-1 становятся мертвыми.

Происходит малая GC — все мертвые объекты удаляются из Эдема и пространства-1. Все живые объекты перемещаются в пространство-2 (ToSpace). Эдем и пространство-2 снова пусты.

Таким образом, в любое время одно из пространств для выживших всегда пусто. Когда выжившие объекты достигают определенного порога перемещения по пространствам выживших, они переходят в старшее поколение

2) Старшее поколение

Объекты-долгожители в конечном итоге переходят из молодого поколения в старшее. Оно также известно как штатное поколение и содержит объекты, которые долгое время оставались в пространствах выживших.

Пороговое значение срока службы объекта определяет, сколько циклов сборки мусора он может пережить, прежде чем будет перемещен в старшее поколение.

Процесс, когда объекты отправляются в мусор из старшего поколения, называется основным событием сборки мусора.

Для установки начального и максимального размера памяти кучи вы можете воспользоваться флагами -Xms и -Xmx.

Поскольку Java задействует сборку мусора по поколениям, то чем больше событий сборки мусора переживает объект, тем дальше он продвигается в куче. Он начинает в молодом поколении и в конечном итоге заканчивает в штатном поколении, если проживет достаточно долго.

Чтобы понять продвижение объектов между пространствами и поколениями, рассмотрим следующий пример.

Когда объект создается, он сначала помещается в пространство Эдема молодого поколения. Как только происходит малая сборка мусора, живые объекты из Эдема перемещаются в пространство FromSpace. Когда происходит следующая малая сборка мусора, живые объекты как из Эдема, так и из пространства перемещаются в пространство ToSpace.

Этот цикл продолжается определенное количество раз. Если объект все еще “в строю” после этого момента, следующий цикл сборки мусора переместит его в пространство старшего поколения.

3) Постоянное поколение

Метаданные, такие как классы и методы, хранятся в постоянном поколении. JVM заполняет его во время выполнения на основе классов, используемых приложением. Классы, которые больше не используются, могут переходить из постоянного поколения в мусор.

Для установки начального и максимального размера постоянного поколения вы можете воспользоваться флагами -XX:PermGen и -XX:MaxPermGen.

4)Мета-пространство

Начиная с Java 8, на смену пространству постоянного поколения (PermGen) приходит пространство памяти MetaSpace. Реализация отличается от PermGen — это пространство кучи теперь изменяется автоматически.

Это позволяет избежать проблемы нехватки памяти у приложений, которая возникает из-за ограниченного размера пространства PermGen в куче. Память мета-пространства может быть собрана как мусор, и классы, которые больше не используются, будут автоматически очищены, когда мета-пространство достигнет максимального размера.

Типы сборщиков мусора

У виртуальной машины Java есть восемь типов сборщиков мусора. Рассмотрим каждый из них в деталях.

1. Серийный GC

Это самая простая реализация GC. Она предназначена для небольших приложений, работающих в однопоточных средах. Все события сборки мусора выполняются последовательно в одном потоке. Уплотнение выполняется после каждой сборки мусора.

Запуск сборщика приводит к событию “остановки мира”, когда все приложение приостанавливает работу. Поскольку на время сборки мусора все приложение замораживается, не следует прибегать к такому в реальной жизни, если требуется, чтобы задержки были минимальными.

1. Параллельный GC

Параллельный сборщик мусора предназначен для приложений со средними и большими наборами данных, которые выполняются на многопроцессорном или многопоточном оборудовании. Это реализация GC по умолчанию, и она также известна как сборщик пропускной способности.

Несколько потоков предназначаются для малой сборки мусора в молодом поколении. Единственный поток занят основной сборкой мусора в старшем поколении.

Запуск параллельного GC также вызывает “остановку мира”, и приложение зависает. Такое больше подходит для многопоточной среды, когда требуется завершить много задач и допустимы длительные паузы, например при выполнении пакетного задания.

1. Старый параллельный GC

Это версия Parallel GC по умолчанию, начиная с Java 7u4. Это то же самое, что и параллельный GC, за исключением того, что в нем применяются несколько потоков как для молодого поколения, так и для старшего поколения.

1. CMS (Параллельная пометка и зачистка) GC

Также известен как параллельный сборщик низких пауз. Для малой сборки мусора задействуются несколько потоков, и происходит это через такой же алгоритм, как в параллельном сборщике. Основная сборка мусора многопоточна, как и в старом параллельном GC, но CMS работает одновременно с процессами приложений, чтобы свести к минимуму события “остановки мира”.

Из-за этого сборщик CMS потребляет больше ресурсов процессора, чем другие сборщики. Если у вас есть возможность выделить больше ЦП для повышения производительности, то CMS предпочтительнее, чем простой параллельный сборщик. В CMS GC не выполняется уплотнение.

1. G1 (Мусор — первым) GC

G1GC был задуман как замена CMS и разрабатывался для многопоточных приложений, которые характеризуются крупным размером кучи (более 4 ГБ). Он параллелен и конкурентен, как CMS, но “под капотом” работает совершенно иначе, чем старые сборщики мусора.

Хотя G1 также действует по принципу поколений, в нем нет отдельных пространств для молодого и старшего поколений. Вместо этого каждое поколение представляет собой набор областей, что позволяет гибко изменять размер молодого поколения.

G1 разбивает кучу на набор областей одинакового размера (от 1 МБ до 32 МБ — в зависимости от размера кучи) и сканирует их в несколько потоков. Область во время выполнения программы может неоднократно становиться как старой, так и молодой.

После завершения этапа разметки G1 знает, в каких областях содержится больше всего мусора. Если пользователь заинтересован в минимизации пауз, G1 может выбрать только несколько областей. Если время паузы неважно для пользователя или предел этого времени установлен высокий, G1 пройдет по большему числу областей.

Поскольку G1 GC идентифицирует регионы с наибольшим количеством мусора и сначала выполняет сбор мусора в них, он и называется: “Мусор — первым”.

Помимо областей Эдема, Выживших и Старой памяти, в G1GC присутствуют еще два типа.

Humongous (Огромная) — для объектов большого размера (более 50% размера кучи).

Available (Доступная) — неиспользуемое или не выделенное пространство.

1. Сборщик мусора Эпсилон

Epsilon — сборщик мусора, который был выпущен как часть JDK 11. Он обрабатывает выделение памяти, но не реализует никакого реального механизма восстановления памяти. Как только доступная куча исчерпана, JVM завершает работу.

Его можно задействовать для приложений, чувствительных к сверхвысокой задержке, где разработчики точно знают объем памяти приложения или даже добиваются ситуации (почти) полной свободы от мусора. В противном случае пользоваться Epsilon GC не рекомендуется.

1. Шенандоа

Shenandoah — новый GC, выпущенный как часть JDK 12. Ключевое преимущество Shenandoah перед G1 состоит в том, что большая часть цикла сборки мусора выполняется одновременно с потоками приложений. G1 может эвакуировать области кучи только тогда, когда приложение приостановлено, а Shenandoah перемещает объекты одновременно с приложением.

Shenandoah может компактировать живые объекты, очищать мусор и освобождать оперативную память почти сразу после обнаружения свободной памяти. Поскольку все это происходит одновременно, без приостановки работы приложения, то Shenandoah более интенсивно нагружает процессор.

1. ZGC

ZGC — еще один GC, выпущенный как часть JDK 11 и улучшенный в JDK 12. Он предназначен для приложений, которые требуют низкой задержки (паузы в менее чем 10 мс) и/или задействуют очень большую кучу (несколько терабайт).

Основные цели ZGC — низкая задержка, масштабируемость и простота в применении. Для этого ZGC позволяет Java-приложению продолжать работу, пока выполняются все операции по сбору мусора. По умолчанию ZGC освобождает неиспользуемую память и возвращает ее в операционную систему.

Таким образом, ZGC привносит значительное улучшение по сравнению с другими традиционными GCS, обеспечивая чрезвычайно низкое время паузы (обычно в пределах 2 мс).

Примечание: как Shenandoah, так и ZGC планируется вывести из экспериментальной стадии в продакшен при выпуске JDK 15.

Как правильно выбрать сборщик мусора

Если у вашего приложения нет строгих требований ко времени задержки, вам стоит просто запустить приложение и предоставить выбор правильного сборщика самой JVM.

В большинстве случаев настройки по умолчанию отлично работают. При необходимости можно настроить размер кучи для повышения производительности. Если производительность по-прежнему не соответствует ожиданиям, попробуйте изменить сборщик в соответствии с требованиями вашего приложения.

Последовательный. Если в приложении небольшой набор данных (примерно до 100 МБ), и/или оно будет работать на одном процессоре без каких-либо требований к времени задержки.

Параллельный. Если приоритет — пиковая производительность приложения, и требования к времени задержки отсутствуют (или допустимы паузы в одну секунду и более).

CMS/G1. Если время отклика важнее, чем общая пропускная способность, и паузы при сборке мусора должны быть короче одной секунды.

ZGC. Если у времени отклика высокий приоритет и/или задействована очень большая куча.

Преимущества сборки мусора

У сборки мусора в Java множество преимуществ.

Прежде всего, это упрощает код. Не нужно беспокоиться о правильном назначении памяти и циклах высвобождения. Вы просто прекращаете использовать объект в коде, и память, которую он занимал, в какой-то момент автоматически восстановится.

Программистам, работающим на языках без сборки мусора (таких как C и C++), приходится реализовывать ручное управление памятью у себя в коде.

Также повышается эффективность памяти Java, поскольку сборщик мусора удаляет из памяти кучи объекты без ссылок. Это освобождает память кучи для размещения новых объектов.

Некоторые программисты выступают за ручное управление памятью вместо сборки мусора, но сборка мусора — уже стандартный компонент многих популярных языков программирования.

Для сценариев, когда сборщик мусора негативно влияет на производительность, Java предлагает множество вариантов настройки, повышающих эффективность GC.