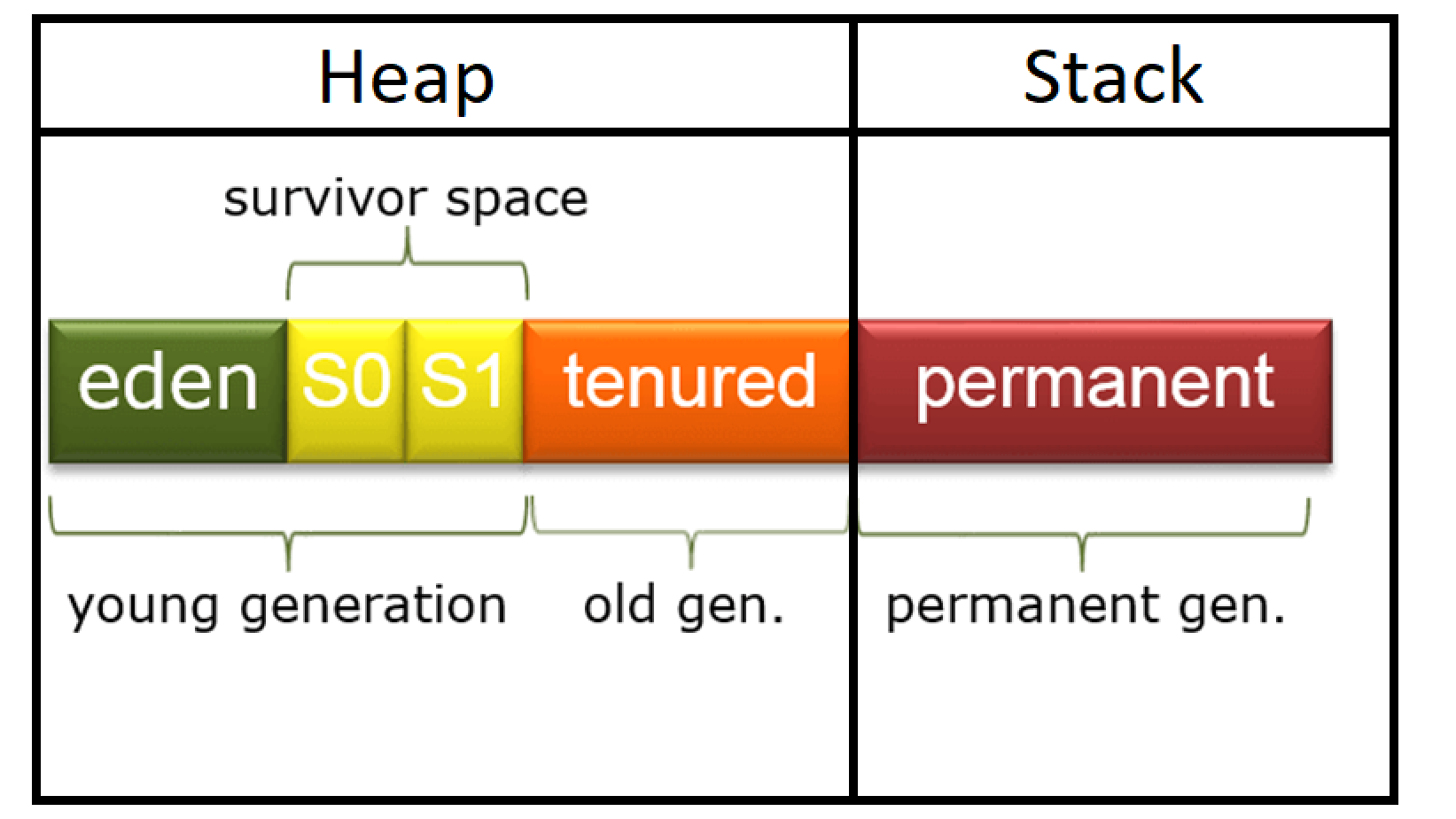
**Сборщик мусора Garbage Collection**



|  |  |
| --- | --- |
| **Различия** | |
| **Heap** | **Stack** |
| Используется всеми частями приложения | Используется только одним потоком исполнения программы |
| Создается новый объект | Размещается ссылка на объект  (также хранятся примитивы) |
| Объекты доступны из любого места программы | Память не доступна для других потоков |
| Если память полностью занята, то бросается исключение java.lang.OutOfMemoryError | Если память полностью занята, то бросается исключение **java.lang.StackOverflowError** |
| Памяти больше, но работает медленно | Памяти меньше, но работает намного быстрее из-за простоты (LIFO) |

# В чем разница между Permanent Generation и Metaspace в Java 8?

Основное отличие с точки зрения пользователя заключается в том, что **Metaspace по умолчанию автоматически увеличивает свой размер** (до того, что предоставляет базовая ОС), в то время как PermGen всегда имеет фиксированный максимальный размер. Вы можете установить фиксированный максимум для Metaspace с параметрами JVM, но вы не можете сделать PermGen автоматически увеличивающимся.

* **Stack**
  + **Permanent Generation** (Постоянное поколение) — используемая JVM память для хранения метаинформации; классы, методы, данные о созданных объектах и т.д.

При каждом создании объекта JVM будет сохранять некоторый набор данных об объекте в области Permanent Generation. Соответственно, чем больше создается в программе объектов, тем больше требуется «пространства» в Permanent Generation.

Размер Permanent Generation можно задать двумя параметрами виртуальной машины JVM:

-**XX:PermSize** – минимальный размер выделяемой памяти для Permanent Generation;

-**XX:MaxPermSize** – максимальный размер выделяемой памяти для Permanent Generation.

Для «больших» Java-приложений можно при запуске определить одинаковые значения данных параметров, чтобы Permanent Generation была создана с максимальным размером. Это может увеличить производительность, поскольку динамическое изменение размера Permanent Generation является «дорогостоящей» (трудоёмкой) операцией. Определение одинаковых значений этих параметров может избавить JVM от выполнения дополнительных операций, связанных с проверкой необходимости изменения размера Permanent Generation.

* **Heap**
  + **Tenured (Old) Generation** (Постоянное поколение) - хранит долгоживущие объекты. Когда данная область памяти заполняется, выполняется полная сборка мусора (full, major collection).
  + **New Generation** (Новое поколение)   
    - **Eden Space** - в этой области выделяется память под все **создаваемые** программой объекты. Жизненный цикл большей части объектов, к которым относятся итераторы, объекты внутри методов и т.п., недолгий.
    - **Survivor Space** - здесь хранятся перемещенные из Eden Space объекты после первой сборки мусора. Объекты, пережившие несколько сборок мусора, перемещаются в следующую сборку Tenured Generation

При создании нового объекта, когда используется оператор 'new', например byte[] data = new byte[1024], этот объект создаётся в сегменте Eden Space. Кроме, собственно данных для массива байт, создается также ссылка (указатель) на эти данные. Если места в сегменте Eden Space уже нет, то JVM выполняет сборку мусора. При сборке мусора объекты, на которые имеются ссылки, не удаляются, а перемещаются из одной области в другую. Так, объекты со ссылками перемещаются из Eden Space в Survivor Space, а объекты без ссылок удаляются.

Если количество используемой Eden Space памяти превышает некоторый заданный объем, то Garbage Collection может выполнить быструю (minor collection) сборку мусора. По сравнению с полной сборкой мусора данный процесс занимает немного времени, и затрагивает только область Eden Space — устаревшие объекты без ссылок удаляются, а выжившие перемещаются в область Survivor Space.

**Размер сегмента Heap можно определить двумя параметрами:** Xms (минимум) и -Xmx (максимум).

**Способы поиска мусора**

**Reference counting** (подсчет ссылок) **Tracing** (отслеживание)

**Reference counting** - у каждого объекта счетчик ссылок. Когда он равен нулю, объект считается мусором. Проблема такого подхода в том, что могут быть цикличные ссылки у объектов друг на друга, в то время как они фактически мусор и не используются программой.

**Tracing** - объект считается не мусором, если до него можно добраться с корневых точек (**GC Root**: локальные переменные и параметры методов, java-потоки, статичные переменные, ссылки из JNI).

-----------------------------------------------------------------------------------------

**GC Root** – корневая точка. Типы:

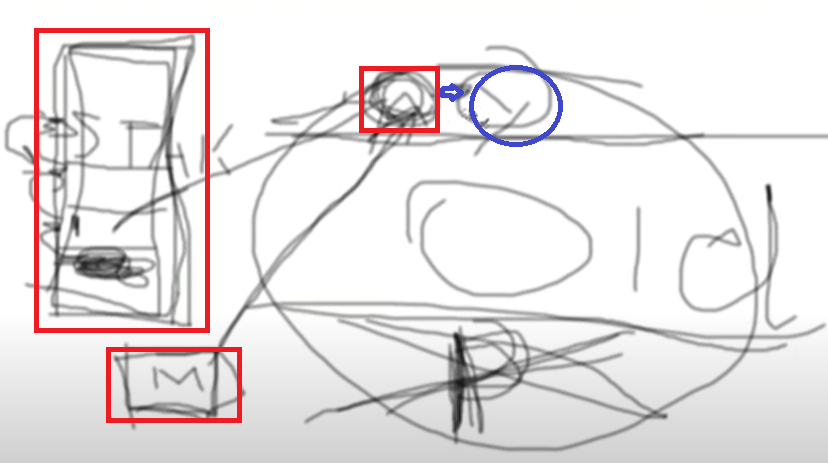
1. Основной Java поток
2. Локальные переменные в основном методе
3. Статические переменные основного класса

Таким образом, простое java-приложение будет иметь следующие корневые точки:

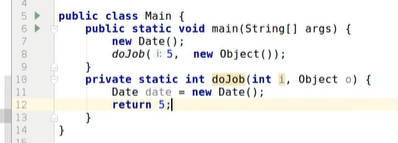
1. Параметры main метода и локальные переменные внутри main метода.
2. Поток, который выполняет main.
3. Статические переменные основного класса, внутри которого находится main метод.

-------------------------------------------------------------------------------------------------------------------------

**GC Root** – живые части программы – Stack, Mata Space (Permanent Space) – и если через них мы можем добраться до объекта, то он считается живым



Например, в ситуации:

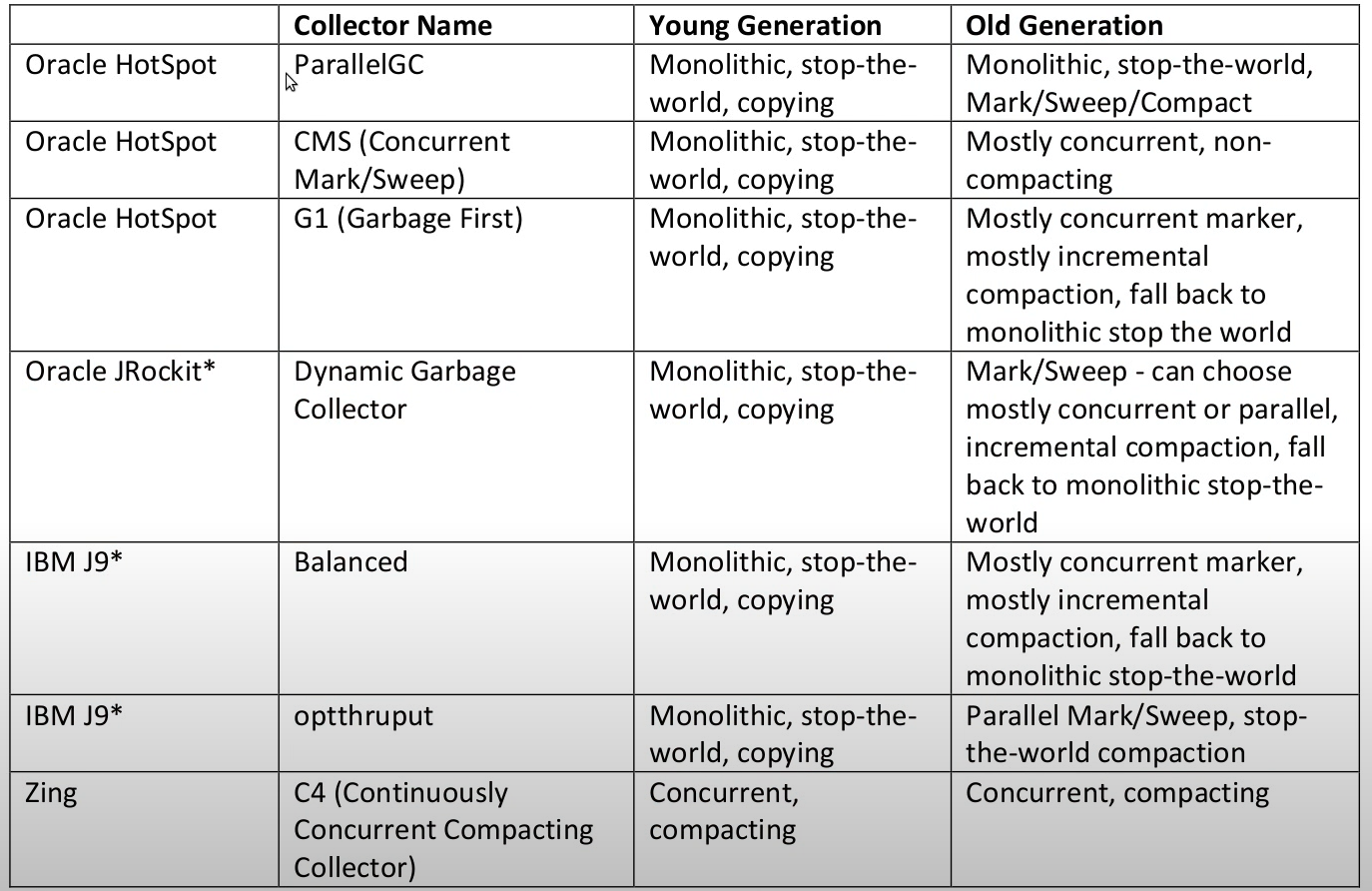


до объекта date мы добраться не можем и этот объект уже подлежит чистке **GC**

**Принципы работы 4 сборщиков HotSpot VM (одна из JVM)**

Виды сборщиков:

1. Serial
2. Parallel (Parallel Compacting)
3. Concurent Mark Sweep (CMS)
4. Garbage-First (G1) и т. д.



**Serial Collector**

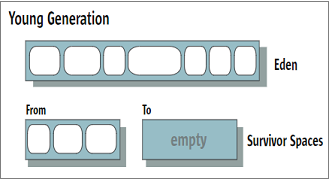
Самый простой. По умолчанию используется, например, в консольных приложениях.

В Serial Garbage Collection **область памяти делится на две части** - «young generation» и «old generation».

|  |  |
| --- | --- |
| **Типы сборки мусора** | |
| Minor GC | mark-sweep-compact |
| частый и быстрый c областью памяти «young generation» | редкий и более длительный c областью памяти «old generation» |

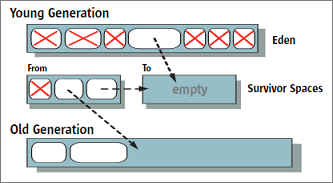
**Алгоритм работы minor GC**

Область памяти «**young generation**», представленная на следующем рисунке, разделена на две части, одна из которых **Survior** также разделена на 2 части (**From**, **To**). Тут на начальном этапе постепенно заполняются области памяти – **Eden, From**



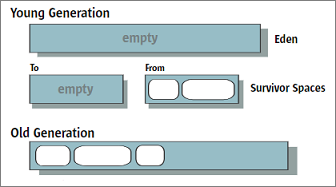
После того, как **Eden** забился, происходят следующее шаги:

* приложение приостанавливается на начало сборки мусора;
* «живые» объекты из **Eden** перемещаются в область памяти «**To**»;
* «живые» объекты из «**From**» перемещаются в «**To**» или в «**old generation**», если они достаточно «старые»;
* **Eden** и «**From**» очищаются от мусора;
* «**To**» и «**From**» меняются местами;
* приложение возобновляет работу.



Все мертвые объекты, которые помечены крестиком и до которых достучаться нельзя, будут уничтожены

В результате сборки мусора картинка области памяти изменится и будет выглядеть следующим образом:



**Алгоритм работы mark-sweep-compact**

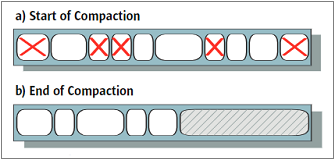
Алгоритм «**mark-sweep-compact**» связан с очисткой и уплотнением области памяти «**old generation**».

Тут работает концепция – **Stop The World**

**Stop The World** – останавливается выполнение всей виртуальной машины Java и начинает выполняться алгоритм **mark-sweep-compact**

Это все происходит в 3 этапа:

1. **Mark** – помечаем те объекты, которые нужно удалить
2. **Sweep** – удаляем эти объекты
3. **Compact** – складываем оставшиеся объекты в одну кучу (делаем дефрагментацию)

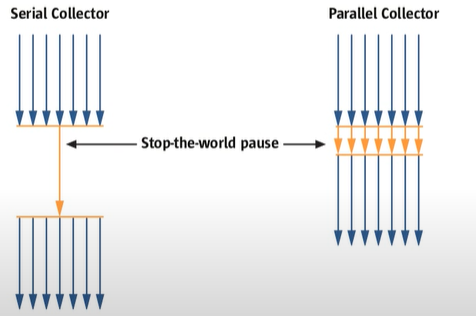


**Вывод:** такой GC устраивает, только тогда, когда мы особо не заметим зависание программы на сек или пол сек (например, то же десктопное приложение). Но если мы такое используем на сервере, на которые приходит примерно 1000 запросов каждую сек, то зависание программы на сек из-за GC может привести к печальным последствиям и Serial коллектор уже не подойдет

**Parallel Collector**

Этот коллектор ускоряет только работу **Young Generation**

По сути, он ее делает только параллельно



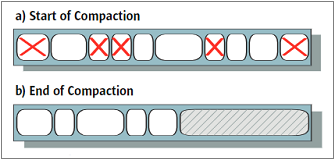
Отличие **Parallel** коллектора от **Serial** заключается в том, что он использует несколько потоков для сканирования и сжатия кучи во время паузы **Stop The World**. Это классно делать, когда компьютер с несколькими ядрами на процессоре.

Также у этого коллектора есть возможность автоматической подстройки под требуемые параметры производительности и меньшие паузы на время сборок.

**Parallel Compacting Collector**

Этот коллектор уже умеет работать с областью **old generation**

Память **old generation** разбивается на участки. Коллектор каждый участок в отдельном потоке начинает чистить. Соответственно чистка происходит быстрее, так как это все работает параллельно.



Но даже при таком подходе у нас все равно могут быть большие паузы между чистками.

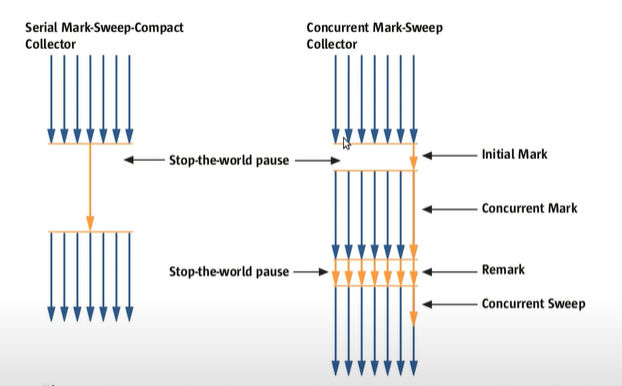
**Concurrent Mark-Sweep (CMS) Collector**

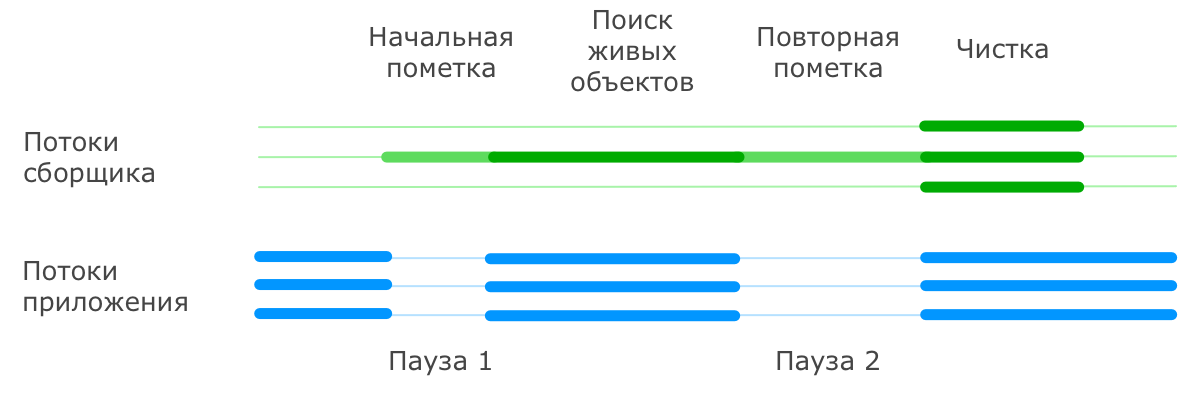
Появился в Java 7

**Young Generation** он выполняет также, как и **Parallel Compacting Collector**

**Old** **Generation** коллектор уже разбивает на несколько фаз

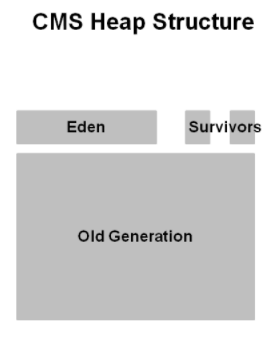
1. **Initial Mark** – происходит остановка основного приложения (**Stop The World** при этом происходит быстро), во время этой остановки **помечаются объекты, которые доступны из корней** – то есть находятся **КОРНЕВЫЕ ОБЪЕКТЫ!!!!**
2. **Concurrent Mark** – CG параллельно с работой программы **производит поиск всех ЖИВЫХ ОБЪЕКТОВ**, доступных по ссылкам из тех самых помеченных корневых объектов – ***обходится граф достижимых объектов*** – все это делается в одном или нескольких потоках
3. **Remark** – ***помечает те ЖИВЫЕ ОБЪЕКТЫ***, которые могли ускользнуть от GC во время предыдущей фазы (**Stop The World** при этом происходит быстро)
4. **Concurrent Sweep** – работа основных потоков приложения возобновляется - сборщик производит очистку памяти от **МЕРТВЫХ ОБЪЕКТОВ** в нескольких параллельных потоках (При этом могут в живых остаться объекты, которые на этот момент таковыми уже не являются – ПЛАВАЮЩИЙ МУСОР (floating garbage). Такие объеты будут удалены при следующей сборки)
5. **Resetting** - подготовьтесь к следующему параллельному сбору, очистив структуры данных



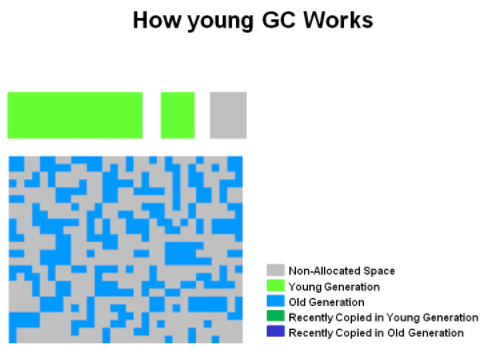
Тут получается 2-а **Stop The World**, но они минимальны и при этом все работает параллельно. Такой коллектор уже может свободно использоваться на web-приложениях!!!!

**Обзор шагов по сборке мусора**

1. **Структура кучи для CMS Collector**

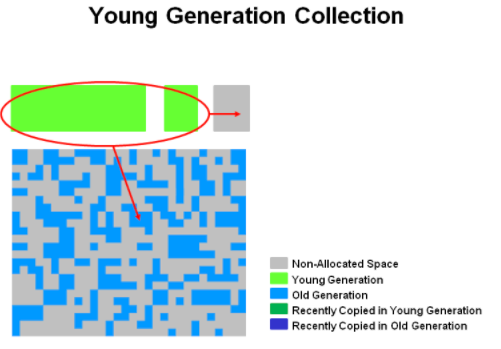
Куча разделена на три части

Молодое поколение разделено на Эдем и два пространства выживших. Старое поколение - одно сплошное пространство. Сбор объектов выполняется на месте. Никакое уплотнение не выполняется, если нет полного GC.

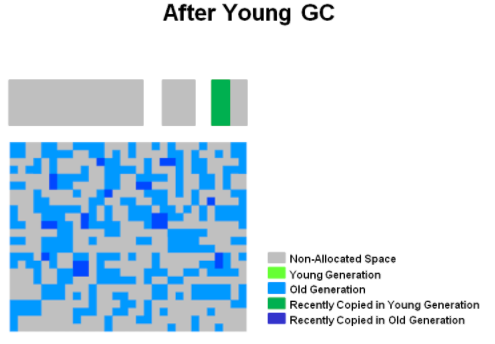
1. **Как работает Young GC в CMS**

Молодое поколение окрашено в светло-зеленый цвет, а старое поколение - в синий. Вот как могла бы выглядеть CMS, если бы ваше приложение работало какое-то время. Объекты разбросаны по территории старого поколения.

В CMS объекты старого поколения освобождаются на месте. Они не перемещаются. Пространство не уплотняется, если нет полного сборщика мусора.

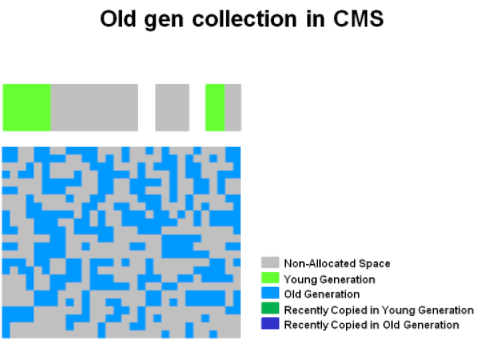
1. **Коллекция молодого поколения**

Живые объекты копируются из пространства Эдема и пространства выжившего в другое пространство выжившего. Любые старые объекты, достигшие порога старения, переходят в старое поколение.

1. **После молодого GC**

После молодого GC пространство Эдема очищается, а одно из ячеек выживших очищается.

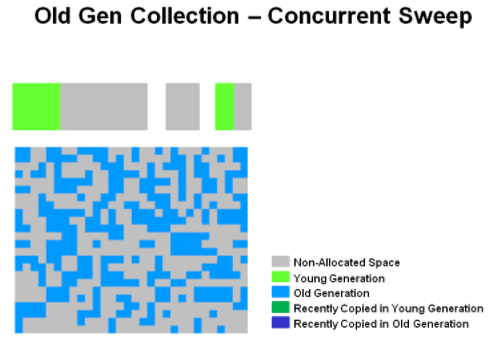
Недавно продвинутые объекты показаны на диаграмме темно-синим цветом. Зеленые объекты - это уцелевшие объекты молодого поколения, которые еще не были преобразованы в старое поколение.

1. **Коллекция старого поколения с CMS**

Происходят две остановки мировых событий: начальная отметка и замечание. Когда старое поколение достигает определенной заполняемости, запускается CMS.

1) Начальная отметка - это фаза короткой паузы, на которой отмечены активные (доступные) объекты. (2) Параллельная маркировка находит живые объекты, пока приложение продолжает работать. Наконец, на этапе (3) замечаний обнаруживаются объекты, которые были пропущены во время (2) одновременной маркировки на предыдущем этапе.

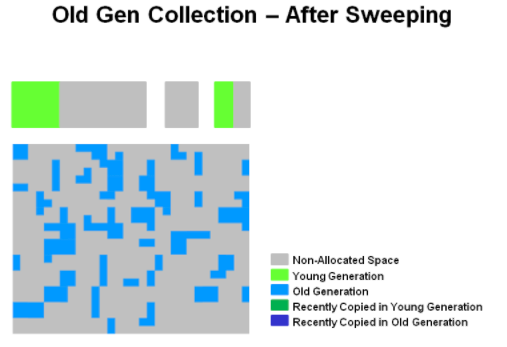
1. **Old Generation Collection - Concurrent Sweep - параллельная проверка**

Объекты, которые не были отмечены на предыдущем этапе, освобождаются на месте. Нет уплотнения.

Примечание:

немаркированные объекты == мертвые объекты

1. **Old Generation Collection - After Sweeping - после подметания**



После фазы (4) очистки вы можете увидеть, что много памяти было освобождено. Вы также заметите, что ***уплотнение не производилось***.

Наконец, коллектор CMS будет проходить этап (5) сброса и ждать следующего достижения порога GC.

### **Ситуации STW**

Из всего сказанного выше следует, что при обычной сборке мусора у CMS GC существуют следующие ситуации, приводящие к STW:

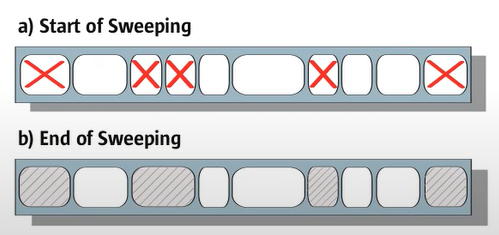
* Малая сборка мусора. Эта пауза ничем не отличается от аналогичной паузы в Parallel GC.
* Начальная фаза поиска живых объектов при старшей сборке (так называемая *initial mark pause*). Эта пауза обычно очень короткая.
* Фаза дополнения набора живых объектов при старшей сборке (известная также как *remark pause*). Она обычно длиннее начальной фазы поиска.

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

**Достоинства**

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.

**Недостатоки**

Этот коллектор не делает **compacting** – дефрагментацию.

Это плохо тем, что становится невозможно поместить большой объект со временем – например, какой-нибудь большой массив, что сказывается на работе приложения

**Garbage-First (G1)**

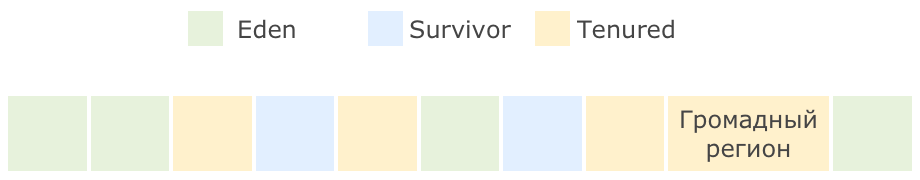
Взял все лучшее от:

1. **Parallel Compacting Collector** – идея разбития памяти на маленькие части и их параллельной очистки
2. **Concurrent Mark-Sweep (CMS) Collector** – идея 2-ух быстрых пауз **Stop The World**

<https://www.oracle.com/technetwork/tutorials/tutorials-1876574.html>

|  |  |
| --- | --- |
| Все старые сборщики мусора (serial, parallel, CMS) структурируют кучу по трем разделам: молодое поколение, старое поколение и постоянное поколение с фиксированным размером памяти. | Куча разделена на набор областей кучи одинакового размера, каждая из которых представляет собой непрерывный диапазон виртуальной памяти. Некоторым наборам регионов назначаются те же роли (eden, Survivor, old), что и в старых сборщиках, но для них нет фиксированного размера. Это обеспечивает большую гибкость в использовании памяти. |
| https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/G1GettingStarted/images/HeapStructure.png | https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/G1GettingStarted/images/slide9.png  ***Куча*** - это единое пространство памяти, разделенное на регионы |

***Куча разделена на набор областей кучи одинакового размера***, каждая из которых представляет собой непрерывный диапазон виртуальной памяти. Некоторым наборам регионов назначаются те же роли (eden, Survivor, old), что и в старых сборщиках, но для них нет фиксированного размера. Это обеспечивает большую гибкость в использовании памяти.



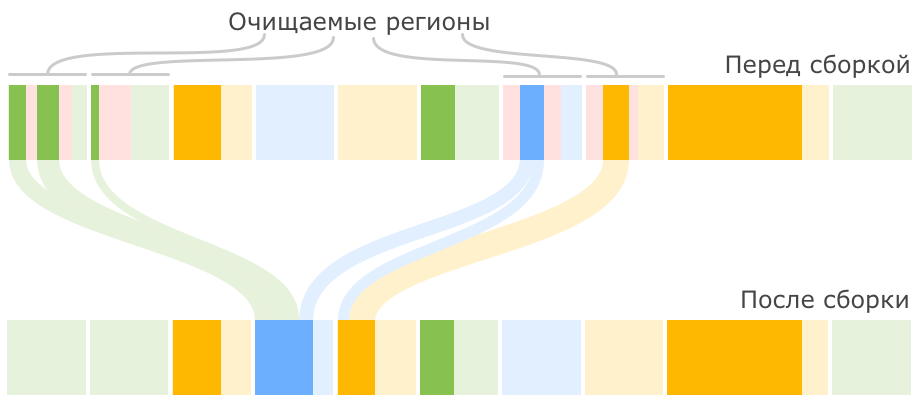
Над малыми сборками мусора трудятся несколько потоков. В прочем они такие же, как и в предыдущих рассмотренных коллекторах.

**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!!!!

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:

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 не учтенных 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. Первая часть этого шага выполняется при остановленном основном приложении.

Следует иметь в виду, что для получения списка живых объектов G1 использует 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. Это, в частности, означает, что G1 допускает наличие плавающего мусора, с которым мы познакомились при рассмотрении сборщика CMS.  
  
После окончания цикла пометки G1 переключается на выполнение **смешанных сборок**. Это значит, что при каждой сборке к набору регионов младшего поколения, подлежащих очистке, добавляется некоторое количество регионов старшего поколения. Количество таких сборок и количество очищаемых регионов старшего поколения выбирается исходя из имеющейся у сборщика статистики о предыдущих сборках таким образом, чтобы не выходить за требуемое время сборки. Как только сборщик очистил достаточно памяти, он переключается обратно в режим малых сборок.  
  
Очередной цикл пометки и, как следствие, очередные смешанные сборки будут запущены тогда, когда заполненность кучи превысит определенный порог.  
  
Смешанная сборка мусора в приведенном выше примере кучи может пройти вот так:



Может оказаться так, что в процессе очистки памяти в куче не остается свободных регионов, в которые можно было бы копировать выжившие объекты. Это приводит к возникновению ситуации *allocation (evacuation) failure*, подобие которой мы видели в CMS. В таком случае сборщик выполняет полную сборку мусора по всей куче при остановленных основных потоках приложения.  
  
Опираясь на уже упомянутую статистику о предыдущих сборках, G1 может менять количество регионов, закрепленных за определенным поколением, для оптимизации будущих сборок.

### **Гиганты**

В начале рассказа о G1 я упомянул о существовании громадных регионов, в которых хранятся так называемые *громадные объекты (humongous objects)*. С точки зрения JVM **любой объект размером больше половины региона считается громадным** и обрабатывается специальным образом:

* Он никогда не перемещается между регионами.
* Он может удаляться в рамках цикла пометки или полной сборки мусора.
* В регион, занятый громадным объектом, больше никого не подселяют, даже если в нем остается свободное место.

Вообще, эти пункты иногда имеют далеко идущие последствия. Объекты большого размера, особенно короткоживущие, могут доставлять много неудобств всем типам сборщиков, так как не удаляются при малых сборках, а занимают драгоценное пространство в регионах старшего поколения (помните объекты-акселераты, обсуждавшиеся в предыдущей главе?) Но G1 оказывается более уязвимым к их негативному влиянию в силу того, что для него даже объект в несколько мегабайт (а в некоторых случаях и 500 КБ) уже является громадным. В [комментарии](http://habrahabr.ru/post/269707/#comment_8633731) к предыдущей статье как раз приводится пример такой [проблемы у Solr](https://wiki.apache.org/solr/ShawnHeisey#GC_Tuning_for_Solr).

### **Ситуации STW**

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

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

Области, определенные G1 как созревшие для рекультивации, собираются с помощью эвакуации. G1 копирует объекты из одной или нескольких областей кучи в одну область кучи, при этом одновременно сжимая и освобождая память. Эта эвакуация выполняется параллельно на мультипроцессорах, чтобы уменьшить время пауз и увеличить пропускную способность. Таким образом, **при каждой сборке мусора G1 постоянно работает над уменьшением фрагментации, работая в пределах заданного пользователем времени паузы**. Это выходит за рамки возможностей обоих предыдущих методов. Сборщик мусора CMS (Concurrent Mark Sweep) не выполняет уплотнение. Сборка мусора ParallelOld выполняет только сжатие всей кучи, что приводит к значительному времени паузы.  
  
 Важно отметить, что G1 не является коллектором в реальном времени. Он соответствует установленному целевому времени паузы с высокой вероятностью, но не с абсолютной уверенностью. **На основе данных из предыдущих сборов G1 оценивает, сколько регионов можно собрать в течение заданного пользователем целевого времени.** Таким образом, сборщик имеет достаточно точную модель стоимости сбора регионов, и он использует эту модель, чтобы определить, какие и сколько регионов нужно собрать, оставаясь в рамках целевого времени паузы.

**Сборщик мусора G1, шаг за шагом**

Сборщик G1 использует другой подход к распределению кучи. На рисунках, которые следуют ниже, рассматривается система G1 шаг за шагом.

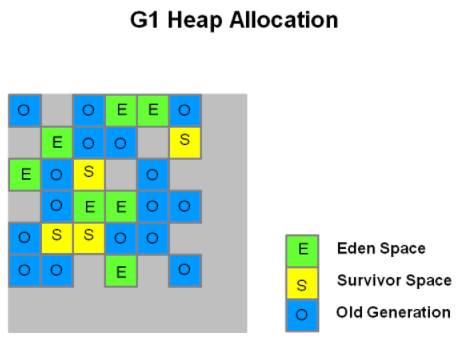
1. **Структура кучи G1**



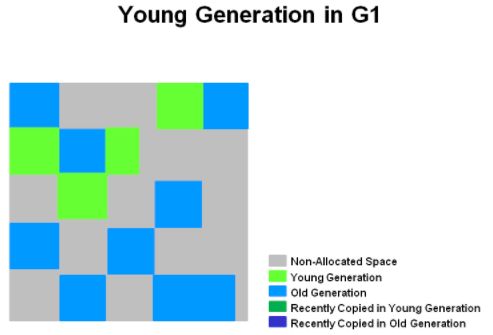
Куча - это одна область памяти, разделенная на множество областей фиксированного размера.

Размер области выбирается JVM при запуске. JVM обычно ориентирована примерно на 2000 регионов размером от 1 до 32 МБ.

1. **Выделение кучи G1**

 В действительности, эти регионы отображены в логических представлениях Эдема, Выжившего и пространств старого поколения.

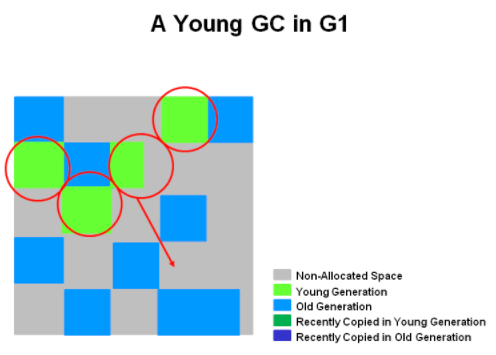
Цвета на картинке показывают, какой регион связан с какой ролью. Живые объекты эвакуируются (т. Е. Копируются или перемещаются) из одного региона в другой. Области предназначены для сбора параллельно с остановкой всех других потоков приложения или без нее.  
  
 Как показано, регионы можно разделить на районы Эдема, выжившие и старые поколения. Кроме того, существует четвертый тип объектов, известный как «Огромные регионы». Эти области предназначены для размещения объектов, размер которых составляет 50% от стандартной области или больше. Они хранятся как набор смежных регионов. Наконец, последним типом регионов будут неиспользуемые области кучи.  
  
Примечание: на момент написания этой статьи сбор огромных объектов не был оптимизирован. Поэтому вам следует избегать создания объектов такого размера.

1. **Молодое поколение в G1**

Куча разделена примерно на 2000 регионов. Минимальный размер - 1 МБ, максимальный - 32 МБ. Синие области содержат объекты старого поколения, а зеленые области содержат объекты молодого поколения.

Обратите внимание, что регионы не обязательно должны быть смежными, как в старых сборщиках мусора.

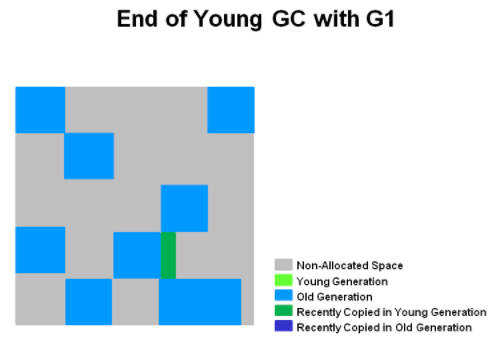
1. **Молодой GC в G1**



Живые объекты эвакуируются (т. е. копируются или перемещаются) в один или несколько уцелевших регионов. Если порог старения достигнут, некоторые объекты перемещаются в регионы старого поколения.

Это пауза остановки мира (STW). Размер Eden и размер выжившего рассчитывается для следующего молодого GC. Учетная информация сохраняется, чтобы помочь рассчитать размер. Учитываются такие вещи, как целевое время паузы.  
  
Такой подход позволяет очень легко изменять размер областей, делая их больше или меньше по мере необходимости.

1. **Конец молодого GC с G1**

 Живые объекты были эвакуированы в регионы выживших или в регионы старого поколения.

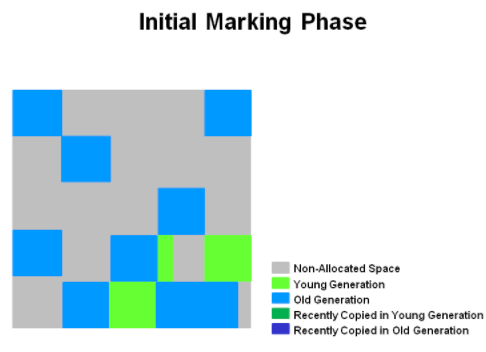
Недавно продвинутые объекты показаны синим цветом. Выжившие регионы отмечены зеленым цветом.

Подводя итог, можно сказать следующее о молодом поколении в G1:  
  
 **Куча - это единое пространство памяти, разделенное на регионы.**Память молодого поколения состоит из набора несмежных областей. Это упрощает изменение размера при необходимости.  
 Сборщики мусора молодого поколения, или молодые сборщики мусора, останавливают мировые события. Все потоки приложения остановлены для выполнения операции.  
 **Молодой сборщик мусора выполняется параллельно с использованием нескольких потоков.**  
 Живые объекты копируются в регионы нового выжившего или старого поколения.

**Фазы сбора G1 - Параллельные фазы цикла маркировки**

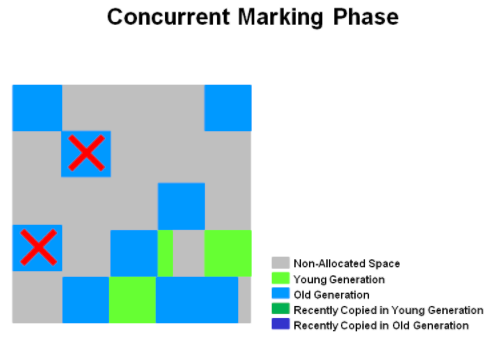
| **Phase** | **Description** |
| --- | --- |
| (1) Initial Mark *(Stop the World Event)*  Начальная отметка | Это остановка мирового события. С G1 он совмещен с нормальным молодым сборщиком мусора. Отметьте уцелевшие регионы (корневые регионы), которые могут иметь ссылки на объекты в старом поколении. |
| (2) Root Region Scanning Сканирование корневой области | Просканируйте оставшиеся в живых регионы на предмет ссылок на старое поколение. Это происходит, пока приложение продолжает работать. Фаза должна быть завершена до того, как появится молодой GC. |
| (3) Concurrent Marking Параллельная маркировка | Найдите живые объекты по всей куче. Это происходит во время работы приложения. Этот этап может быть прерван сборками мусора молодого поколения. |
| (4) Remark *(Stop the World Event)* Перемаркеровка | Завершает маркировку живого объекта в куче. Использует алгоритм, называемый моментальным снимком в начале (SATB), который намного быстрее, чем тот, который использовался в сборщике CMS. |
| (5) Cleanup *(Stop the World Event and Concurrent)* Очистка | * Осуществляет учет на живых объектах и полностью свободных регионах. (Остановите мир) * Очищает запомненные наборы (Stop the world) * Сбросьте пустые регионы и верните их в свободный список. (Concurrent) |
| (\*) Copying *(Stop the World Event)* Копирование | Это остановка, в которой мир останавливается для эвакуации или копирования живых объектов в новые неиспользуемые регионы. Это можно сделать с регионами молодого поколения, которые регистрируются как [пауза GC (молодой)]. Или вы оба |

1. **Initial Marking Phase**



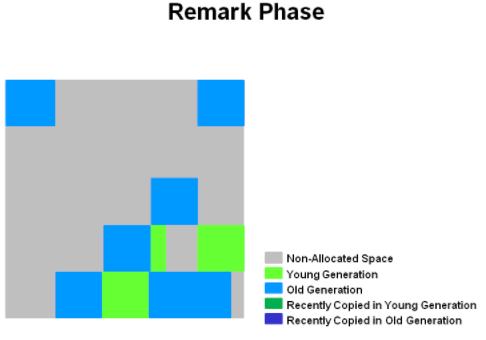
Первоначальная маркировка живого объекта совмещается со сборкой мусора молодого поколения. В журналах это отмечается как пауза сборщика мусора (молодой) (начальная отметка).

1. **Concurrent Marking Phase**

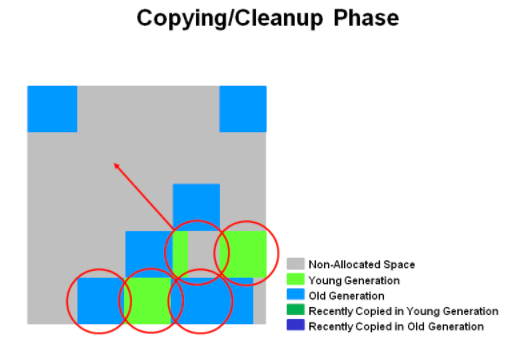


Если обнаружены пустые области (обозначенные знаком «X»), они немедленно удаляются на этапе примечания. Также вычисляется «учетная» информация, определяющая живучесть.

1. **Remark Phase**

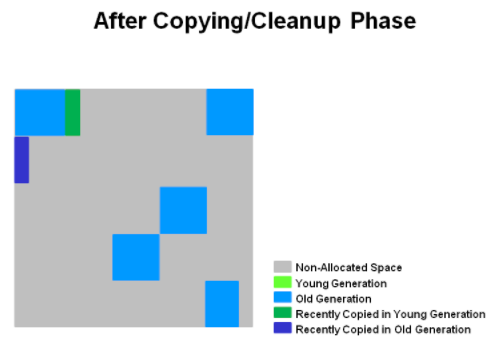
 Пустые области удаляются и восстанавливаются. Живучесть регионов теперь рассчитывается для всех регионов.

1. **Copying/Cleanup Phase**



G1 выбирает регионы с наименьшей «живостью», те регионы, которые могут быть собраны быстрее всего. Затем эти регионы собираются одновременно с молодым сборщиком мусора. В журналах это обозначается как [пауза сборщика мусора (смешанная)]. Так что и молодое, и старое поколения собираются одновременно.

1. **After Copying/Cleanup Phase**

 Выбранные области были собраны и уплотнены в темно-синюю область и темно-зеленую область, показанные на диаграмме.

**Краткое описание GC старого поколения**

Таким образом, есть несколько ключевых моментов, которые мы можем сказать о сборке мусора G1 в старом поколении.

1. **Параллельная фаза маркировки**

Информация о жизнеспособности рассчитывается одновременно во время работы приложения.

Эта информация о живучести определяет, какие регионы лучше всего восстановить во время эвакуационной паузы.

Нет фазы развёртывания *sweeping*, как в CMS.

1. **Remark Фаза**

Использует алгоритм моментального снимка в начале (SATB), который намного быстрее, чем тот, который использовался с CMS.

Осваиваются полностью пустые регионы.

1. **Фаза копирования / очистки**

Одновременно восстанавливаются молодое и старое поколение.

Регионы старшего поколения выбираются исходя из их жизнеспособности.

https://ziginsider.github.io/Garbage\_Collector\_Java/

### **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.

**Как избежать утечки памяти в JAVA?**

Утечка памяти — это ситуация, **когда в куче есть объекты, которые больше не используются, но сборщик мусора не может удалить их**, что приводит к нерациональному расходованию памяти.

**3.1 Утечки памяти из-за статических полей**

В Java **время жизни статических полей обычно совпадает со временем работы приложения.**

Мы должны быть внимательны при использовании статических переменных. Если коллекции или объекты объявлены как статические, то они остаются в памяти в течение всего срока работы приложения, тем самым блокируя ресурсы, которые можно было бы использовать в другом месте.

**Как это предотвратить?**

* Минимизировать использование статических переменных в приложении.
* При использовании синглтонов использовать реализацию с ленивый загрузкой объекта, вместо немедленной.

**3.2 Через незакрытые ресурсы**

Всякий раз, когда мы создаем новое соединение или открываем поток, JVM выделяет память для этих ресурсов. Это могут быть соединения с базой данных, входящие потоки или сессионные объекты.

Забывая закрыть эти ресурсы, вы можете заблокировать память, тем самым делая их недоступными для сборщика мусора. Это может произойти даже в случае возникновения исключения, которое не позволит программе выполнить код, отвечающий за закрытие ресурсов.

В любом случае, **открытые соединения потребляеют память** и если мы не будем корректно обрабатывать их закрытие, они могут ухудшить производительность системы и даже привести к OutOfMemoryError.

**Как это предотвратить?**

* Всегда используйте *finally* блок для закрытия ресурсов.
* Код (даже в блоке *finally*), который закрывает ресурсы, не должен иметь никаких необработанных исключений.
* При использовании версии Java 7 и выше, мы можем использовать блок *try-with-resources.*

**3.3 Неверные реализации equals() и hashCode()**

При написании новых классов очень распространенной ошибкой является некорректное написание переопределяемых методов *equals()* и *hashCode() .*

HashSet и HashMap используют эти методы во многих операциях и если они не переопределены правильно, то эти методы могут стать источником потенциальных проблем, связанных с утечкой памяти.

Примером является использование ORM, например Hibernate, который использует методы *equals()* и *hashCode()* для анализа объектов и сохранения их в кеше.

Если эти методы не переопределены, то шансы утечки памяти довольно высоки, потому что Hibernate не сможет сравнивать объекты и заполнит свой кеш их дубликатами.

**Как это предотвратить?**

* Взять за правило, при создании новых сущностей (Entity), всегда переопределять методы equals() и hashCode() .
* Не достаточно просто переопределить эти методы. Они должны быть переопределены оптимальным образом.

**3.4 Внутренние классы, которые ссылаются на внешние классы**

Не статическим внутренним классам (анонимным) для инициализации всегда требуется экземпляр внешнего класса.

Каждый нестатический внутренний класс по умолчанию имеет неявную (скрытую) ссылку на класс в котором он находится. Если мы используем этот объект внутреннего класса в нашем приложении, то даже после того, как объект внешнего класса завершает свою работу, он не будет утилизирован сборщиком мусора.

**Как это предотвратить?**

* Если внутреннему классу не нужен доступ к членам внешнего класса, подумайте о превращении его в статический класс.

**3.5 Через finalize() методы**

Использование финализаторов является еще одним потенциальным источником утечек памяти. Всякий раз, когда в классе переопределяется метод *finalize(),* объект этого класса не убирается сборщиком мусора немедленно. Вместо этого он помещается сборщиком в очередь на утилизацию, которая происходит немного позже.

Кроме того, если код, написанный в методе finalize(), переопределен неоптимально, и если очередь финализатора не может идти в ногу со сборщиком мусора Java, то рано или поздно нашему приложению суждено встретить ошибку OutOfMemoryError.

**Как это предотвратить?**

* Мы всегда должны избегать финализаторов.

**3.6 Интернированные строки**

В Java 7 пул строк претерпел значительные изменения: он был перенесен из PermGen в HeapSpace (подробнее об этом можно прочитать в статье [PermGen и Metaspace в среде Java](https://topjava.ru/blog/permgen-and-metaspace)). Но в приложениях, работающих на версии 6 и ниже, мы должны быть более внимательными при работе с большими строкам.

**Когда мы читаем большой строковый объект и вызываем у него метод intern(), то он сохраняется в пул строк, который находиться в PermGen (постоянная память) и остается там до тех пор, пока наше приложение работает**. Это блокирует память и приводит к серьезным утечкам в приложении.

**Как это предотвратить?**

* Самый простой способ решить эту проблему — обновиться до последней версии Java, поскольку пул строк был перемещен в пространство кучи, начиная с 7 версии.
* При работе с большими строками можно увеличить размер PermGen, что позволит избежать ошибки *OutOfMemoryError:*

**3.7 Использование ThreadLocals**

ThreadLocal — это механизм, который позволяет изолировать состояние (значения переменных) в определенном потоке, что делает его безопасным.

При использовании этой конструкции, каждый поток будет содержать неявную ссылку на его копию переменной *ThreadLocal* и будет хранить свою собственную копию, вместо того чтобы совместно использовать ресурс через множество потоков, так долго, сколько поток будет жить.

**Утечки памяти по причине использования ThreadLocals.**  
Предполагается что *ThreadLocal* переменные будут собраны сборщиком мусора после того, как содержащий их поток перестанет существовать. Но существует проблема с использованием *ThreadLocal* в современных серверах приложений.

Современные сервера приложений используют пул потоков для обработки запросов, вместо создания нового потока на каждый запрос. Кроме того, они используют отдельный загрузчик классов.

Поскольку пулы потоков в серверах приложений работают по принципу повторного использования потоков, они никогда не удаляются сборщиком мусора — вместо этого они повторно используются для обслуживания другого запроса.

Итак, если класс создает ThreadLocal переменную, но не удаляет ее явно, то копия этого объекта останется в рабочем потоке даже после остановки веб-приложения, тем самым не позволяя сборщику удалить этот объект.

**Как это предотвратить?**

* Хорошей практикой является очищение *ThreadLocal* переменных, когда они больше не используются. ThreadLocal предоставляет метод *remove()*, который удаляет значение переменной для текущего потока
* Не используйте ThreadLocal. set (null) для очистки значения — на самом деле оно не очищает значение, а вместо этого ищет мапу, связанную с текущим потоком, и устанавливает пару ключ-значение — текущий поток и null соответственно
* Еще лучше рассмотреть ThreadLocal как ресурс, который необходимо закрыть в блоке finally, чтобы убедиться, что он всегда будет закрыт, даже в случае исключения:

