**LESSON 4**

**Stack memory**

Стековая память в Java работает по схеме LIFO (Последний-зашел-Первый-вышел). Всякий раз, когда вызывается метод, в памяти стека создается новый блок, который содержит примитивы и ссылки на другие объекты в методе. Как только метод заканчивает работу, блок также перестает использоваться, тем самым предоставляя доступ для следующего метода.

Размер стековой памяти намного меньше объема памяти в куче.

**Hype memory**

Java Heap (куча) используется Java Runtime для выделения памяти под объекты и JRE классы. Создание нового объекта также происходит в куче.

Здесь работает сборщик мусора: освобождает память путем удаления объектов, на которые нет каких-либо ссылок.

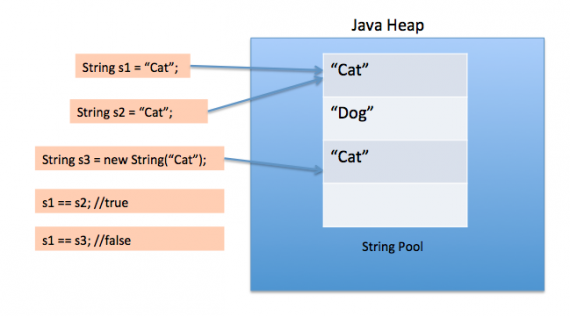
Любой объект, созданный в куче, имеет глобальный доступ и на него могут ссылаться с любой части приложения.

*Разница между Stack и Heap памятью в Java*

|  |  |
| --- | --- |
| **Stack** | **Heap** |
| Стек используется только одним потоком исполнения программы. | Куча используется всеми частями приложения. |
| Память стека содержит только локальные переменные примитивных типов и ссылки на объекты в куче. | Всякий раз, когда создается объект, он всегда хранится в куче, а в памяти стека содержится ссылка на него. |
| Стековая память не может быть доступна для других потоков. | Объекты в куче доступны с любой точки программы. |
| Управление памятью в стеке осуществляется по схеме LIFO. |  |
| Стековая память существует лишь какое-то время работы программы. | Память в куче живет с самого начала до конца работы программы |
| Для стека определить размер памяти можно с помощью опции -Xss . | Мы можем использовать -Xms и -Xmx опции JVM, чтобы определить начальный и максимальный размер памяти в куче. |
| Если память стека полностью занята, то Java Runtime бросает *java.lang.StackOverflowError* | Если память кучи заполнена, то бросается исключение *java.lang.OutOfMemoryError: Java Heap Space* |
| Размер памяти стека намного меньше памяти в куче.  Из-за простоты распределения памяти (LIFO), стековая память работает намного быстрее кучи. |  |

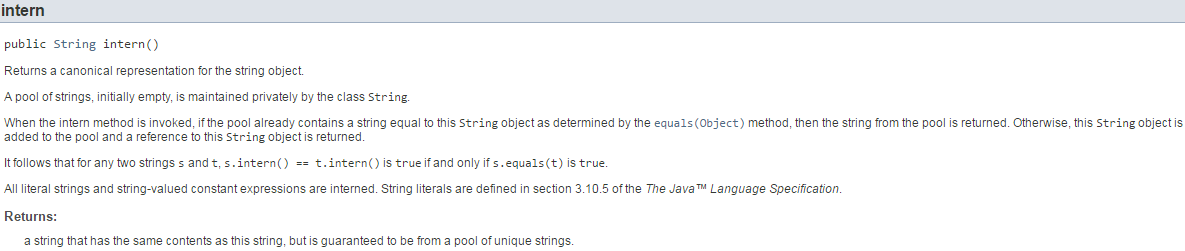
**String Pool**

Пул строк – это набор строк, который хранится в памяти Java heap. Мы знаем, что String это специальный класс в Java, и мы можем создавать объекты этого класса, используя оператор new точно так же, как и создавать объекты, предоставляя значение строки в двойных кавычках.



Когда мы используем двойные кавычки для создания строки, сначала ищется строка в пуле с таким же значением, если находится, то просто возвращается ссылка, иначе создается новая строка в пуле, а затем возвращается ссылка.

Тем не менее, когда мы используем оператор new, мы принуждаем класс String создать новый объект строки, а затем мы можем использовать метод intern() для того, чтобы поместить строку в пул, или получить из пула ссылку на другой объект String с таким же значением.



*Зачем нужен пул строк?*

1. Пул строк возможен исключительно благодаря неизменяемости строк в Java.
2. Пул строк помогает экономить большой объем памяти, но с другой стороны создание строки занимает больше времени.

**PermGen**

Почти каждая JVM в настоящее время использует отдельную область памяти, которая называется Permanent Generation, для хранения информации о внутреннем представлении класса. PermGen также используется для хранения дополнительной информации, например ссылки на статических переменных.

Размер PermGen был фиксированным до Java 8.

Причина возникновения *java.lang.OutOfMemoryError: PermGen space* связана с тем, что JVM необходимо подгружать определение нового класса, но недостаточно места в PermGen для этого (слишком много классов хранится), или утечка памяти.

Размер PG можно задать двумя параметрами JVM: -XX:PermSize – задаёт минимальный, или изначальный, размер PG, и -XX:MaxPermSize – задаёт максимальный размер.

**Garbage Collector**

*За что отвечает Garbage Collector ?*

1. Обнаружение мусора
2. Очищение от мусора

*Как Garbage Collector обнаруживает мусор?*

Существует два подхода к обнаружению мусора:

1. Reference counting
2. Tracing

**Reference counting**

Суть подхода состоит в том, что каждый объект имеет счетчик. Счетчик хранит информацию о том, сколько ссылок указывает на объект. Когда ссылка уничтожается, счетчик уменьшается. Если значение счетчика равно нулю, - объект можно считать мусором и память можно очищать.

Главным минусом такого подхода является сложность обеспечения точности счетчика. Также при таком подходе сложно выявлять циклические зависимости (когда два объекта указывают друг на друга, но ни один живой объект на них не ссылается). *Это приводит к утечкам памяти.*

В общем, Reference counting редко используется из-за недостатков. Во всяком случае HotSpot VM его не использует. Поэтому мы можем отложить в памяти, что такой подход есть, и продолжить дальше.

**Tracing**

В "Tracing" главная идея состоит в мысли: "Живые объект - те до которых мы можем добраться с корневых точек (GC Root), все остальные - мусор. Все что доступно с живого объекта - также живое".

Если мы представим все объекты и ссылки между ними как дерево, то нам нужно пройти с корневых узлов по всем узлами. При этом узлы, до которых мы сможем добраться - не мусор, все остальные - мусор.

При таком подходе легко выявить циклические зависимости, - все объекты к которым невозможно добраться с корневых точек будут считаться мусором.

HotSpot VM использует именно такой подход.

*Что такое корневая точка (GC Root)?*

Литература говорит, что существует 4 типа корневых точек:

• Локальные переменные и параметры методов

• Java Потоки

• Статические переменные

• Ссылки из JNI

Самое простое java приложение будет иметь такие корневые точки:

• Локальные переменные внутри main метода, параметры main метода.

• Поток, который выполняет main.

• Статические переменные класса, внутри которого находится main метод.

*Как GC очищает память от мусора?*

Теперь мы понимаем, какой механизм используется для обнаружения мусора. Нужно разобраться как именно GC очищает память.

Рассмотрим известные человечеству подходы.

**Copying collectors**

Память делится на две части "from-space" "to-space".

Принцип работы такой:

• Объекты аллоцируются в "from-space"

• "from-space" заполняется, нужно собрать мусор

• Приложение приостанавливается

• Запускается сборщик мусора. Находятся живые объекты в "from-space" и копируются в "to-space"

• Когда все объекты скопированы "from-space" полностью очищается

• "to-space" и "from-space" меняются местами

Главный полюс такого подхода в том, что объекты плотно забивают память. Минусы подхода:

• Приложение должно остановится пока не пройдет полный цикл сборки мусора

• В худшем случае form-space и to-space должны быть одинакового размера. Это случай, когда все объекты живые.

В итоге, плюс в том, что память используется эффективно. Но при этом приложение должно прекращать свою работу на время сборки мусора. Также очень не эффективно используется память, так как в худшем случае "from-space" должен быть равен "to-space".

В чистом виде такой алгоритм в HotSpot VM не используется.

**Mark-and-sweep**

Алгоритм можно описать так:

• Объекты аллоцируются в памяти

• Нужно запустить GC

• Приложение приостанавливается

• Сборщик проходится по дереву объектов, помечая живые объекты

• Сборщик проходится по всей памяти, находя все не отмеченные куски памяти, сохраняя их в "free list"

• Когда новые объекты начинают аллоцироватся они аллоцируются в память доступную в "free list"

Минусы:

• Приложение не работает пока происходит сборка мусора

• Время работы зависит от размеров памяти и количества объектов

• Если не использовать "compacting" память будет использоваться не эффективно

*Какой подход используется в HotSpot VM?*

Сборщики мусора HotSpot VM используют подход "Generational Garbage Collection". Как мы увидим, этот подход позволяет использовать разные алгоритмы для разных этапов сборки мусора. Это позволяет использовать наиболее подходящий алгоритм.

Было замечено, что большинство приложений удовлетворяют двум правилам (weak generational hypothesis):

• Большинство аллоцированых объектов быстро становятся мусором.

• Существует мало связей между объектами, которые были созданы в прошлом и только что аллоцироваными объектами.

Именно на эти правила опирается подход "Generational Garbage Collection".

В HotSpot VM реализовано четыре сборщика мусора основанных на идее "Generational Garbage Collection":

• Serial GC

• Parallel GC

• CMS GC

• G1 GC

Для того, что бы разобраться с принципом работы "Generational Garbage Collection", рассмотрим "Serial GC".

Serial GC был одним из первых сборщиков мусора в HotSpot VM. Во время работы этого сборщика приложения приостанавливается и продолжает работать после прекращение сборки мусора.

Память делится на три пространства:

• Young generation. Объекты аллоцируются в этом участке. Обычно имеет сравнительно не большой размер. Очищается часто. Предполагается, что количество объектов переживших сборку будет мало (основывая на "weak generational hypothesis"). Сборку мусора в этом участке называют "minor garbage collection". В общем, "minor garbage collection" проходит часто, быстро и уничтожает кучу мусора, так как происходит на сравнительно не большом участке памяти который скорее всего содержит много мусора.

• The old generation. Объекты которые переживают "minor collection" перемещаются в участок памяти называемый "old generation". Обычно "old generation" больше чем "young generation". Заполняется этот участок сильно медленней, так как большинство объектов живут не долго. В итоге, сборка мусора в "old generation" (major garbage collection) происходит не часто, но когда происходит, занимает много времени.

• Permanent generation. Тут хранятся метаданные, классы, интернированные строки, и тд. Дальше рассматривать его не будем.

В итоге, мы знаем, что память делится на части. Нас интересует "young generation" и "old generation". Новые объекты создаются в "young generation", пережившие сборку мусора попадают в "old generation". Существует "minor GC" и "major GC".

• "minor GC" - проходит часто и быстро, в основном работает с "young generation".

• "major GC" - проходит редко и долго, в основном работает с "old generation".

minor GC

Для того, что бы "minor GC" проходил быстро, нужно что бы при нем не приходилось сканировать "old generation". Возникает вопрос: "Как выявить ссылки на объекты c "old generation" на объекты в "young generation" не сканируя "old generation""

Как мы помним, соответствуя "weak generational hypothesis" их должно быть мало, но они могут быть.

Для решения этой проблемы HotSpot VM содержит структуру "card table".

Память в "old generation" разбивается на карты (cards).

Card table - это массив с однобайтной ячейкой, каждая ячейка массива соответствует куску памяти (карте) в "old generation". Когда в каком то поле объекта обновляется ссылка, то в "card table" нужная карта помечается как "грязная" (для этого нужна однобайтная ячейка). В итоге при "minor GC" для выявления ссылок "old-to-new" сканируется не весь "old-generation", а только объекты которые находятся в "грязных" картах.

"Young generation" делится на:

• Eden. Кусок памяти, где объекты алоцируются. После сборки мусора "Eden" пустой, мусор должен удалится, а выжившие объекты попасть в "Survivor space"

• Survivor space 1,2. То, что в разделе "Copying collectors" называлось "from-space" и "to-space". Тут находятся объекты, которые выжили при предыдущей сборке мусора, но перед отправкой в "old generation" им дан шанс стать мусором во время следующей сборки.

Survivor space 1 будем называть "from space", Survivor space 2 - "to space".

Алгоритм работы очень похож на "Copying collectors", отличие в том, что появился "Eden":

• Начало сборки мусора, приложение приостанавливается.

• Живые объекты из "Eden" копируются в "to space".

• Живые объекты из "from space" копируются в "to space" или в "old generation", если они достаточно старые.

• "Eden" и "from space" очищаются, так как в них остался только мусор.

• "to space" и "from space" меняются местами

• Приложение продолжает работу.

После сборки мусора:

После "minor gc" "Eden" и "to space" пустые, в "from space" лежат объекты пережившие сборку, немного долгоживущих объектов перекочевало в "old generation".

major GC

"major GC" работает по принципу "sliding compacting mark-sweep". Принцип работы похож на "Mark-and-sweep", но добавляется процедура "compacting", которая позволяет более эффективно использовать память.

Процедура заключается в перемещении живых объектов к началу "old generation space", таким образом мусор остается в конце. Для аллокации нужно иметь указатель на последний живой объекты и дальше просто аллоцировать и сдвигать указатель к концу "old generation".

• Запускается GC

• Приложение приостанавливается

• Сборщик проходится по дереву объектов в "old generation", помечая живые объекты

• Сборщик проходится по всей памяти, находя все не отмеченные куски памяти, они помечаются как мусор

• Все живые объекты сдвигаются к началу "old generation", мусор становится одним куском памяти, который находится сразу за последним живым объектом

• Приложение возобновляет свою роботу.

**Types of references**

Ссылка -  множество величин, объединенных определенной совокупностью допустимых операций.

Integer ref; // ссылка с именем ref , а типом Integer, неинициализированная

|  |  |  |  |
| --- | --- | --- | --- |
| **Strong Reference** | **Weak Reference** | **Soft Reference** | **Phantom Reference** |
| Если объект доступен через цепочку ссылок (как например объект, на который ссылается переменная buffer), то такая ссылка называется жесткой и сборщик мусора не станет уничтожать такой объект.  Минусы:  Утечка памяти (должны перенаправить на null, чтобы сборщик мусора её обработал)  Кэширование (сборщик мусора не сможет освободить память, которую занимает закэшированная ссылка) | Слабая ссылка - это ссылка, которая недостаточно сильна чтобы объект не собирался сборщиком мусора. | По своей природе данный вид ссылок очень похож на WeakReference, с одним очень существенным отличием: объекты по ссылкам уничтожаются в том случае, когда память вашей программы заполнена и появляется вероятность получить OutOfMemoryError  Рекомендуется для кэширования | Связь с объектами в этих ссылках такая слабая, что вы даже не сможете получить эти объекты - get() метод всегда будет возвращать null.  Область применения этих ссылок в отслеживании момента, когда ссылка помещается в очередь ReferenceQueue |
| StringBuffer buffer = new StringBuffer(); | WeakReference<Thing> weakThing = new WeakReference<Thing>(thing); | SoftReference<Thing> thing = new SoftReference<Thing>(new Thing()); | PhantomReference<Thing> thing = new PhantomReference<Thing>(new Thing(), queue); |