Программа: цикл на 100000 итераций, в котором в предварительно созданную Map<Integer, String> складываются ключ = {индекс} и значение = {"value" + индекс}

Задание по JIT:

Запустить программу с опциями -XX:+PrintCompilation, -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining и проанализировать информацию в консоли

1) -XX:+PrintCompilation будет выводить в стандартный поток вывода логирование для каждого метода и цикла:

time id attr level methodName (size) deopt

```
370  412     n 0       java.lang.ClassLoader::defineClass2 (native)   (static)
370  410       3       java.security.SecureClassLoader$CodeSourceKey::hashCode (21 bytes)
371  416       3       java.lang.ClassLoader::getNamedPackage (73 bytes)
371  417       3       jdk.internal.module.SystemModuleFinders$SystemModuleReader::release (10 bytes)
371  418       3       jdk.internal.jimage.ImageReader::releaseByteBuffer (5 bytes)
371  420   !   3       jdk.internal.jimage.BasicImageReader::readBuffer (232 bytes)
372  419       3       jdk.internal.jimage.BasicImageReader::releaseByteBuffer (16 bytes)
373  411   !   3       java.lang.ClassLoader::checkCerts (195 bytes)
374  413       3       java.lang.ClassLoader::addClass (9 bytes)
374  414  s    3       java.util.Vector::addElement (24 bytes)
```

1. **time** — это время со старта JVM
2. **id** — id задачи
3. **attr** – набор атрибутов с доп информацией
   - % — OSR (on-stack replacement)
   - s — метод является синхронизированным (synchronized)
   - ! — метод содержит обработчик исключений
   - b — компиляция произошла в блокирующем режиме
   - n — скомпилированный метод является оберткой нативного метода
4. **level** – уровень компиляции
   - — многоуровневая компиляция выключена
   - 0 — интерпретируемый код
   - 1 — C1 с полной оптимизацией (без профилирования)
   - 2 — C1 с учетом количества вызовов методов и итераций циклов
   - 3 — C1 с профилированием
   - 4 — C2 (более оптимизированный код)
5. **methodName** – название скомпилированного метода
6. **size** – размер скомпилированного байт-кода
7. **deopt** – название деоптимизации (made zombie и т.п.)

2) -XX:+UnlockDiagnosticVMOptions – разблокирует диагностические параметры JVM

3) -XX:+PrintInlining – будет выводить в стандартный поток вывода, какие методы становятся встроенными. Должен использоваться вместе с параметром **XX:+UnlockDiagnosticVMOptions**

```
1017  477       1       com.sun.tools.javac.util.SharedNameTable$NameImpl::getIndex (5 bytes)
1017  474       3       java.lang.String::toCharArray (25 bytes)
                            @ 1   java.lang.String::isLatin1 (19 bytes)
                            @ 11   java.lang.StringLatin1::toChars (16 bytes)
                              @ 11   java.lang.StringLatin1::inflate (34 bytes)   callee is too large
                            @ 21   java.lang.StringUTF16::toChars (18 bytes)
                              @ 13   java.lang.StringUTF16::getChars (42 bytes)   callee is too large
```

Задание по GC: *Запустить приложение создающее много объектов с разными GC, посмотреть в jvisualvm как заполняются объекты в разных областях памяти(heap)*

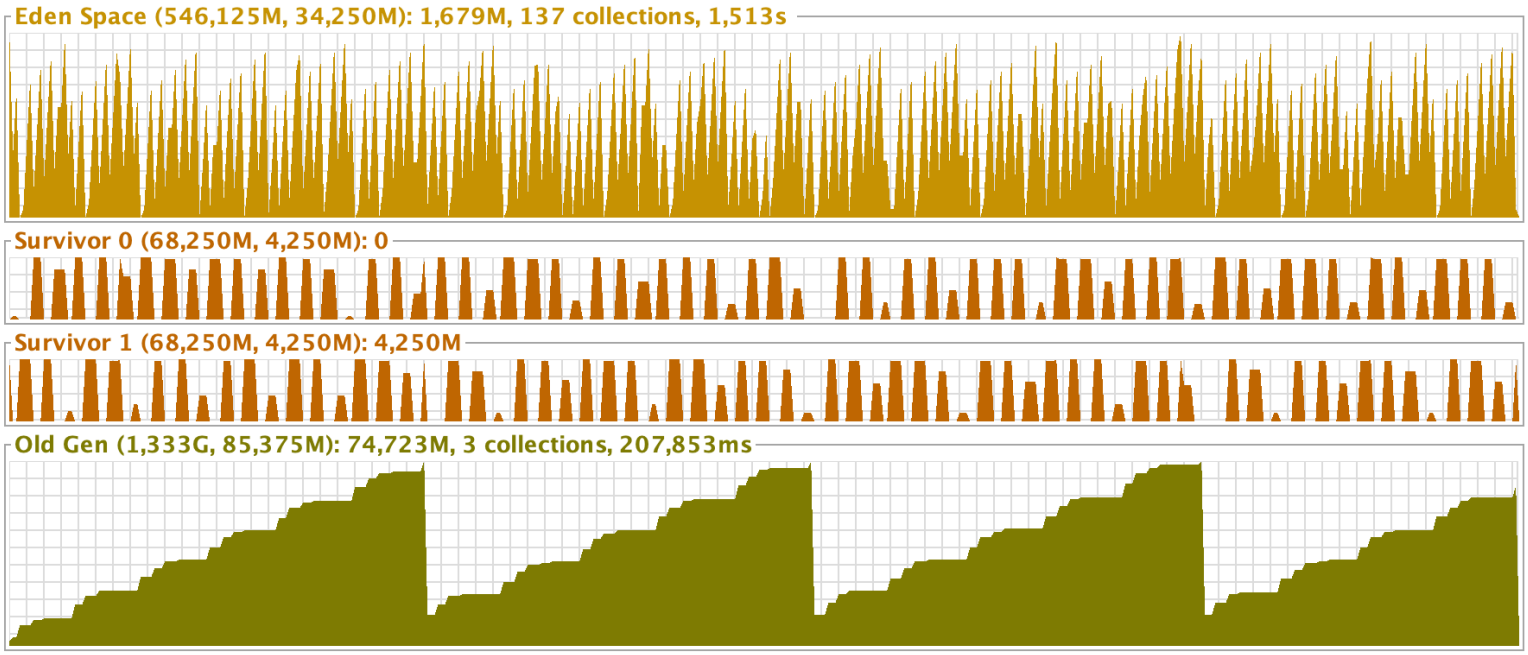
Существуют различные сборщики мусора, использующие общую стратегию Mark–Sweep–Compact:

Сборщик мусора	Описание	Преимущества / Недостатки	Когда использовать	Флаги для включения
Serial	использует один поток	+ нет оверхедов на взаимодействие потоков - длинные паузы - ручная настройка размера кучи	• однопроцессорные машины • не требуется большой размер кучи (~100 МБ) • приложение не очень чувствительно к коротким остановкам	**-XX:+UseSerialGC**
Parallel	использует несколько потоков	+ быстрая сборка мусора + автоматическая настройка - фрагментация памяти	• многопроцессорные машины • в приоритете пиковая производительность • допустимы паузы при GC в одну секунду и более • работа со средними и большими наборами данных	**-XX:+UseParallelGC**
CMS	использует несколько потоков и более умную major сборку	+ минимизация времени простоя - относительно низкая пропускная способность - фрагментация Tenured - требуется больше RAM для оптимальной работы	• многопроцессорные машины • работа с большим объемом долгоживущих данных	**-XX:+UseConcMarkSweepGC (removed in JDK-14)**
G1	использует несколько потоков и новый подход к организации кучи	+ аккуратнее предсказывает размеры пауз, чем CMS, и лучше распределяет сборки во времени - низкая пропускная способность 90%	• многопроцессорные машины • время отклика важнее пропускной способности • паузы GC должны быть меньше одной секунды.	**-XX:+UseG1GC**

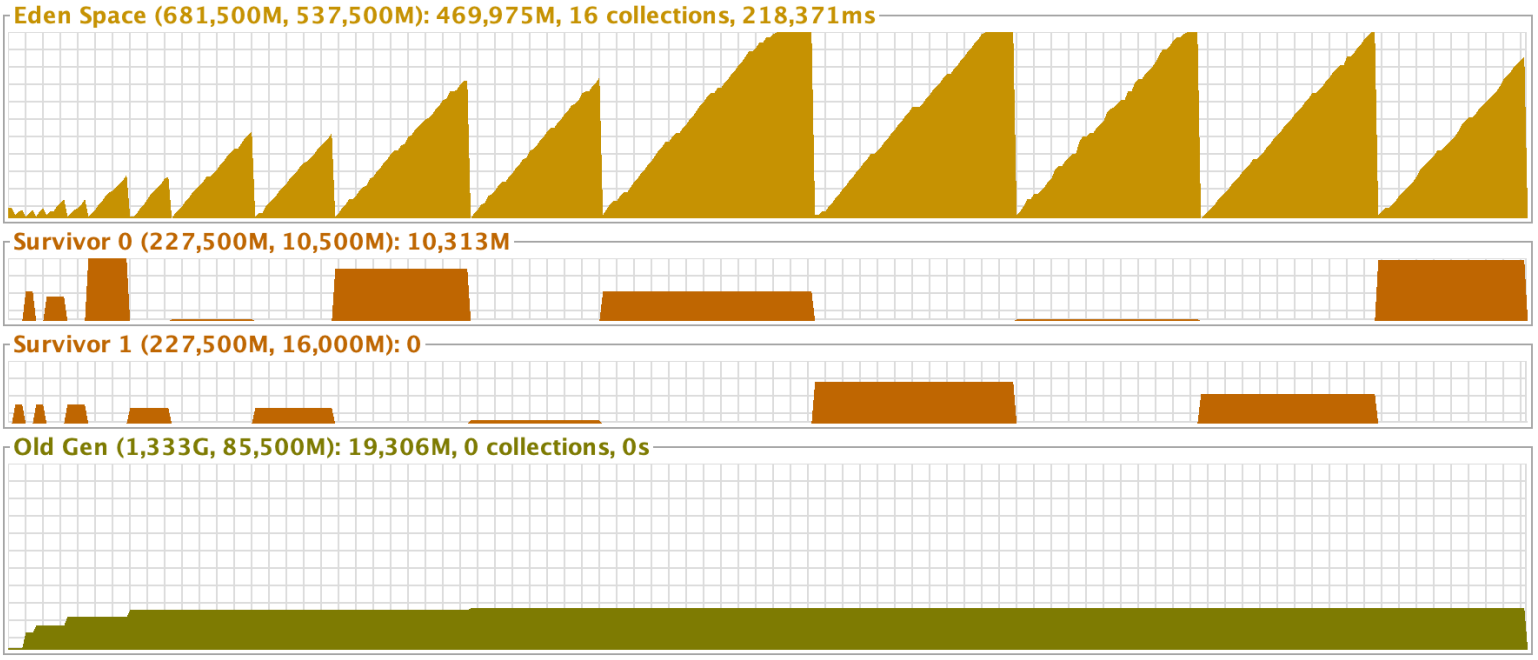
Z1	использует несколько потоков и новый подход к сборке мусора	+ субмиллисекундные паузы + время паузы не растет с увеличением кучи - низкая пропускная способность	• многопроцессорные машины • паузы GC должны быть меньше 10 мс • требуется большой размер кучи до 16 ТБ	**-XX:+UseZGC**
Epsilon	по-новому аллоцирует память и не осуществляет сборку мусора	+ быстрый процесс создания новых объектов - память не освобождается	• все объекты создаются при старте • короткоживущие приложения • анализ других сборщиков мусора	**-XX:+UseEpsilonGC**
Shenandoah	использует несколько потоков и новый подход к сборке мусора	+ короткие паузы независимо от размера кучи + множество настроек под специфику работы приложения - повышенное потребление ресурсов	• многопроцессорные машины • время отклика стоит на первом месте • нужно протестировать приложение, чтобы узнать ответ	**-XX:+UseShenandoahGC**

Запустим внутри цикла на 500 итераций программу с различными garbage коллекторами:

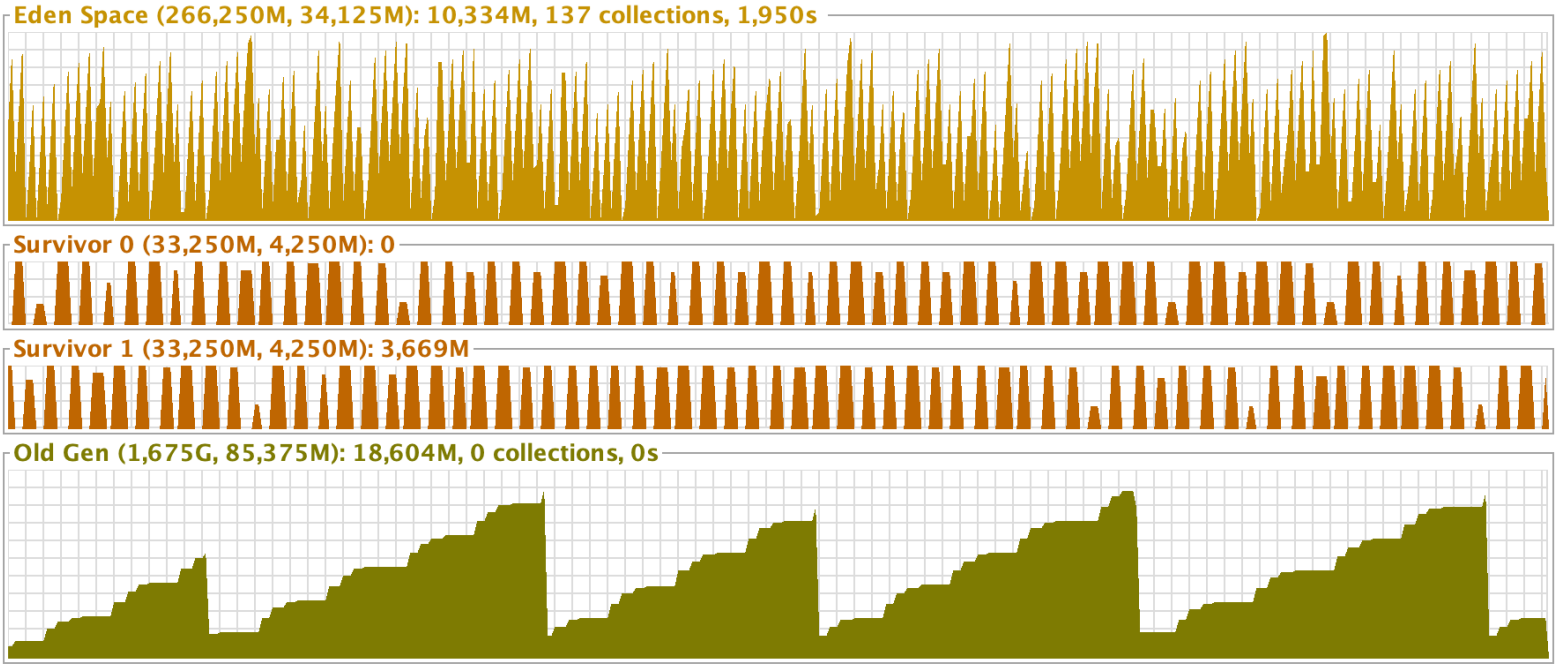
1) **-XX:+UseSerialGC**



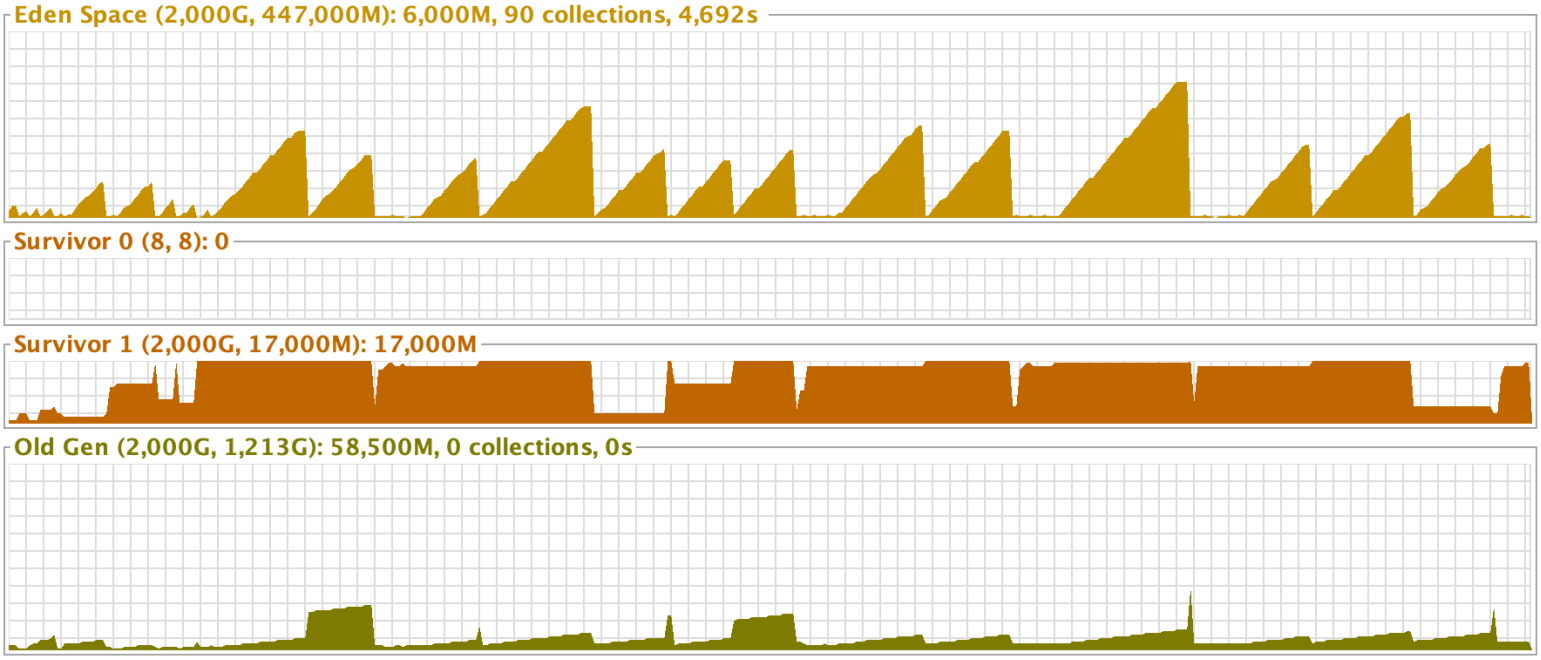
2) -XX:+UseParallelGC



3) -XX:+UseConcMarkSweepGC

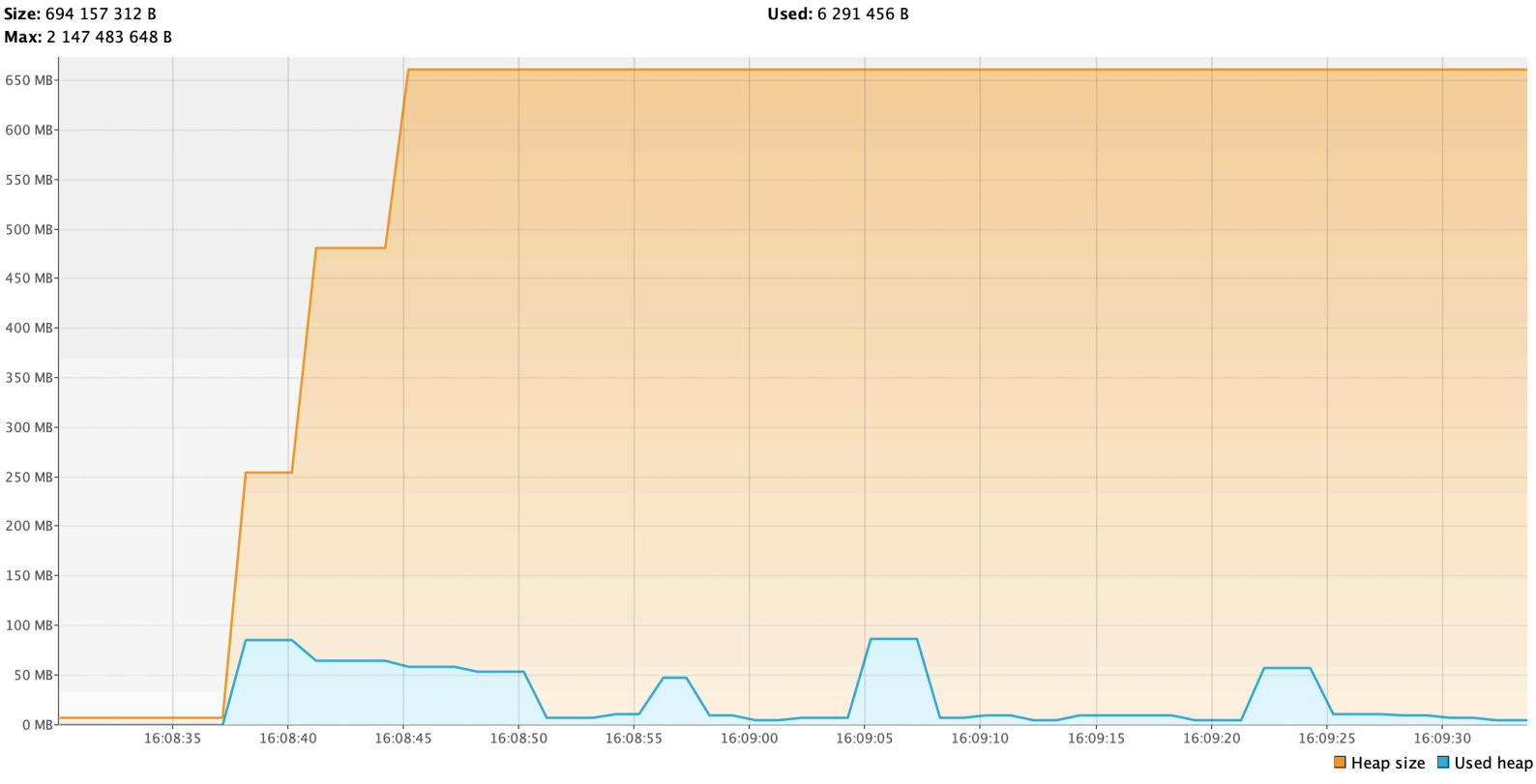


4) -XX:+UseG1GC



5) -XX:+UseZGC

Не поддерживается в Visual GC



6) -XX:+UseEpsilonGC

Не поддерживается в Visual GC