СБЕРБАНК ТЕХНОЛОГИИ

JIT компиляция

Узнаем

СБЕРБАНК ТЕХНОЛОГИИ

- Как работает JIT компиляция
- Как работает JIT компиляция в HotSpot’s?
- Мониторинг JIT компиляции в HotSpot’s

Типы компиляций

СБЕРБАНК ТЕХНОЛОГИИ

- **JIT**-компиляция (англ. **Just-in-time** compilation, компиляция «на лету»), динамическая компиляция (англ. dynamic translation) — технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы

Static vs DINAMIC

СБЕРБАНК ТЕХНОЛОГИИ

- Статическая компиляция
  - “ahead-of-time”(AOT) compilation
  - Исходный код -> Машинный код
  - Компиляция происходит до выполнения программы
- Динамическая компиляция
  - “just-in-time” (JIT) compilation
  - Исходный код → Байткод → Интерпретатор + JITted executable
  - Компиляция происходит во время выполнения программы

Статическая компиляция

СБЕРБАНК ТЕХНОЛОГИИ

- Плюсы

можно использовать длительный по времени и сложный анализ и оптимизацию

- Минусы

статической информации часто не хватает для качественной оптимизации

сложно использовать профилирующую информацию

нельзя использовать специфические особенности платформы (“железа”)

Динамическая компиляция

СБЕРБАНК ТЕХНОЛОГИИ

- Плюсы

  используются агрессивные оптимистичные оптимизаций за счет широкого использования информации профилирования (сбор характеристик работы программы)

  оптимизация с учетом особенностей конкретной платформы(железа) на которой выполняется программа

- Минусы

  во время выполнения программы часть вычислительных ресурсов системы необходимо потратить на компиляцию

  задержки при старте программы

  пиковая производительность может быть ниже (в некоторых ситуациях)

JIT компиляция в Java может быть выполнена:

- непосредственно перед выполнением кода программы
- когда JVM решит что это нужно
- в некоторых случаях никогда

Жизненный цикл

СБЕРБАНК ТЕХНОЛОГИИ

Интерпретатор -> Профилирование программы -> Динамическая компиляция -> Деоптимизация

- Динамическая компиляция выполняется асинхронно
- Замена кода на оптимизированный происходит “на лету”

СБЕРБАНК ТЕХНОЛОГИИ

Способы оптимизации

Optimizations in HotSpot JVM

СБЕРБАНК ТЕХНОЛОГИИ

§ compiler tacticsdelayed compilationtiered compilationon-stack replacementdelayed reoptimizationprogram dependence graph rep.static single assignment rep.§ proof-based techniquesexact type inferencememory value inferencememory value trackingconstant foldingreassociationoperator strength reductionnull check eliminationtype test strength reductiontype test eliminationalgebraic simplificationcommon subexpressioneliminationinteger range typing§ flow-sensitive rewritesconditional constant propagationdominating test detectionflow-carried type narrowingdead code elimination§ language-specific techniquesclass hierarchy analysisdevirtualizationsymbolic constant propagationautobox eliminationescape analysislock elisionlock fusionde-reflection§ speculative (profile-based) techniquesoptimistic nullness assertionsoptimistic type assertionsoptimistic type strengtheningoptimistic array lengthstrengtheninguntaken branch pruningoptimistic N-morphic inliningbranch frequency predictioncall frequency prediction§ memory and placement transformationexpression hoistingexpression sinkingredundant store eliminationadjacent store fusioncard-mark eliminationmerge-point splitting§ loop transformationsloop unrollingloop peelingsafepoint eliminationiteration range splittingrange check eliminationloop vectorization§ global code shapinginlining (graph integration)global code motionheat-based code layoutswitch balancingthrow inlining§ control flow graph transformationlocal code schedulinglocal code bundlingdelay slot fillinggraph-coloring register allocationlinear scan register allocationlive range splittingcopy coalescingconstant splittingcopy removaladdress mode matchinginstruction peepholingDFA-based code generator

Inligning - Подстановка кода метода вместо вызова метода

- Не тратятся вычислительные ресурсы на вызов виртуального метода
- JVM может значительно более качественно оптимизировать получившийся после сделанного inlining ‘а код (основная причина)

Inlining

СБЕРБАНК ТЕХНОЛОГИИ

```
int addAll(int max) {
    int accum = 0;
    for (int i = 0; i < max; i++) {
        accum = add(accum, i);
    }
    return accum;
}

int add(int a, int b) {
    return a + b;
}
```

Inlining

СБЕРБАНК ТЕХНОЛОГИИ

```
int addAll(int max) {
    int accum = 0;
    for (int i = 0; i < max; i++) {
        accum = accum + i;
    }
    return accum;
}
```

Типы вызовов методов

СБЕРБАНК ТЕХНОЛОГИИ

- Monomorphic (“одна реализация метода”)
- Bimorphic (“две реализации метода”)
- Polymorphic (“много реализация метода, но можно выделить часто используемый”)
- Megamorphic (“много реализация метода”)

СБЕРБАНК ТЕХНОЛОГИИ

Inlining - наиболее эффективная оптимизация

- Эффективно оптимизирует Monomorphic и Bimorphic методы
- Оптимизирует часто используемую реализацию в Polymorphic методах
- JVM может применять Inlining в рефлексии
- JVM может применять Inlining в сгенерированном динамически байт-коде (JSR-292)

Intrinsics

СБЕРБАНК ТЕХНОЛОГИИ

- Intrinsics – замена байт кода на “native” код
- JIT компилятор знает что должен делать метод(знает контракт метода) в стандартных библиотеках java и подменяет на “native” реализацию
- Существующие intrinsics - String::equals, Math::*, System::arraycopy,Object::hashCode, Object::getClass, sun.misc.Unsafe::*

HotSpot JVM JIT compilers

- client /C1
- server/C2
- tiered mode (C1 + C2)

Client/C1

СБЕРБАНК ТЕХНОЛОГИИ

- java –client (только на 32-bit JVM)
  - быстрая генерация кода приемлемого качества
  - базовая оптимизация
  - не требуется профилирование программы
  - начало компиляции после 1500 вызовов метода

server/C2

СБЕРБАНК ТЕХНОЛОГИИ

- java -server
- высоко оптимизированный код
- агрессивные методы оптимизации использующие профилирование программы
- начало компиляции после 10000 вызовов метода

Преимущества/недостатки

СБЕРБАНК ТЕХНОЛОГИИ

- Client/C1

  + быстрый старт

  - ниже пиковая производительность

- Server/C2

  + очень быстрое выполнение для часто используемых методов (hot methods)

  - медленный старт / требуется “разогрев”

Tiered compilation

СБЕРБАНК ТЕХНОЛОГИИ

- C1 + C2

  -XX:+TieredCompilation

  Level 0 = Interpreter

  Level 1-3 = C1

  - #1: C1 w/o profiling
  - #2: C1 w/ basic profiling
  - #3: C1 w/ full profiling

  Level 4 = C2

MONITORING JIT-Compiler

СБЕРБАНК ТЕХНОЛОГИИ

- -XX:+PrintCompilation вывод информации о скомпилированных методах
- -XX:+PrintInlining вывод информации о inlining методах

-XX:+PrintCompilation

СБЕРБАНК ТЕХНОЛОГИИ

java -XX:+PrintCompilation

```
 83    1             java.lang.String::indexOf (70 bytes)
 92    2             java.lang.String::hashCode (55 bytes)
106    3             java.lang.Object::<init> (1 bytes)
115    4             java.lang.Integer::stringSize (21 bytes)
116    5             java.lang.Integer::getChars (131 bytes)
124    6             java.util.HashMap::indexFor (6 bytes)
126    7    n        java.lang.System::arraycopy (native)   (static)
```

-XX:+PrintInlining

СБЕРБАНК ТЕХНОЛОГИИ

java -XX:+PrintCompilation -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining

```
163    9          java.util.HashMap::transfer (115 bytes)
                 @ 81   java.util.HashMap::indexFor (6 bytes)   inline (hot)
166   10          java.lang.Math::min (11 bytes)
167   11          java.lang.String::length (6 bytes)
171   12          java.lang.AbstractStringBuilder::append (48 bytes)
                 @ 8   java.lang.String::length (6 bytes)   inline (hot)
                 @ 19   java.lang.AbstractStringBuilder::ensureCapacityInternal (16 bytes)   inline (hot)
                 @ 33   java.lang.String::getChars (62 bytes)   inline (hot)
                  @ 58   java.lang.System::arraycopy (0 bytes)   (intrinsic)
```

СБЕРБАНК ТЕХНОЛОГИИ

Garbage Collector

Узнаем

СБЕРБАНК ТЕХНОЛОГИИ

- Как работает Garbage Collector
- Типы Garbage Collector
- Как выбрать GC

Garbage Collector

СБЕРБАНК ТЕХНОЛОГИИ

Задачи Garbage Collector:

- Обнаруживать мусор
- Очищать память от мусора

Как Garbage Collector обнаруживает мусор?

СБЕРБАНК ТЕХНОЛОГИИ

Существует два подхода к обнаружению мусора:

- Reference counting
- Tracing

Reference counting

СБЕРБАНК
ТЕХНОЛОГИИ

Reference counting

СБЕРБАНК ТЕХНОЛОГИИ

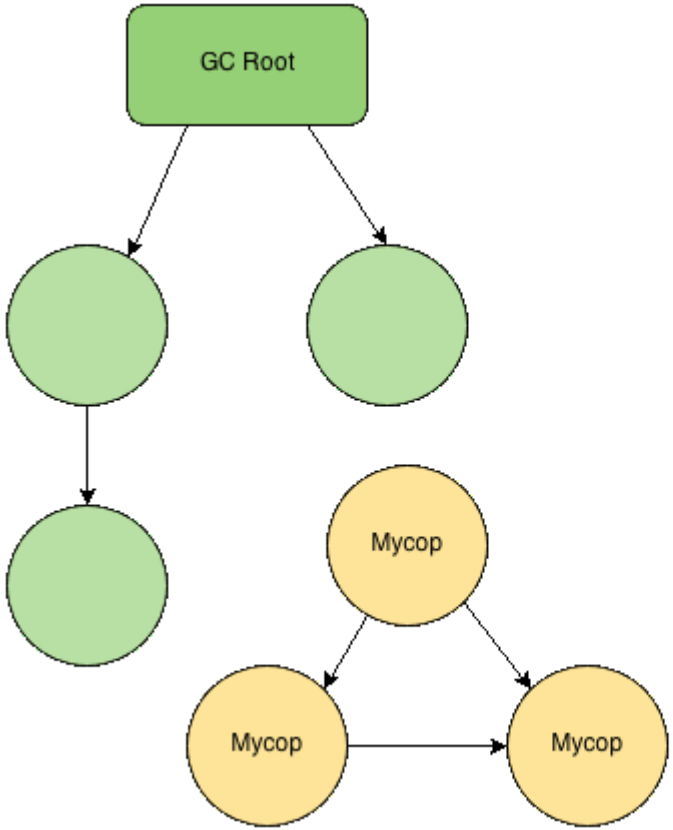
Недостатки

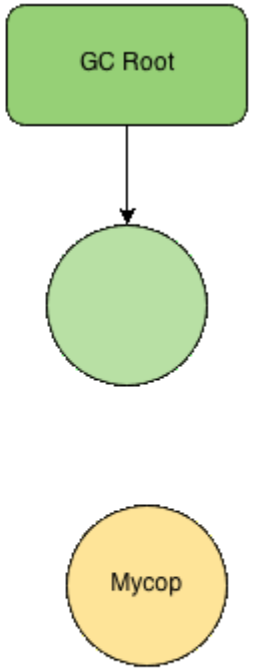
- сложность обеспечения точности счетчика
- сложно выявлять циклические зависимости (когда два объекта указывают друг на друга, но ни один живой объект на них не ссылается)

Живые объекты - те до которых мы можем добраться с корневых точек, все остальные - мусор. Все что доступно с живого объекта - также живое.

Tracing

СБЕРБАНК ТЕХНОЛОГИИ





Преимущества

- Легко обнаружить мусор
- Легко выявить циклические зависимости, - все объекты к которым не возможно добраться с корневых точек будут считаться мусором.

HotSpot VM использует данный подход

Существует 4 типа корневых точек:

- Локальные переменные и параметры методов основного потока
- Java Потоки
- Статические переменные
- Ссылки из Java Native Interface

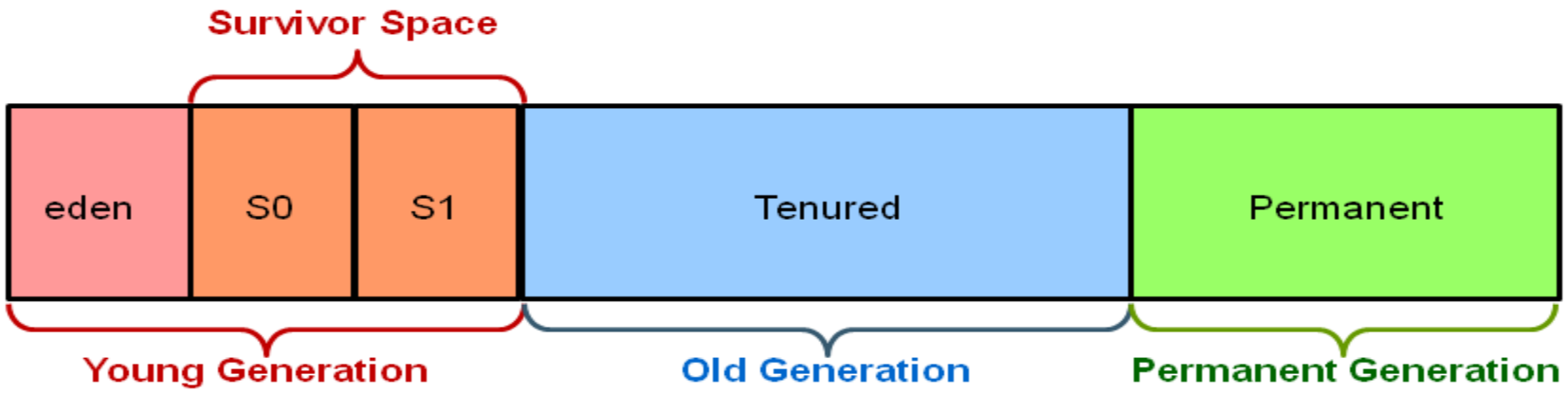
Как GC очищает память от мусора?

СБЕРБАНК ТЕХНОЛОГИИ

HotSpot VM Garbage Collector опирается на данные правила

СБЕРБАНК ТЕХНОЛОГИИ

Hotspot Heap Structure



Garbage Collector

СБЕРБАНК ТЕХНОЛОГИИ

Память делится на три пространства:

- Young generation
- Old generation
- Permanent generation (MetaSpace в Java 8 и выше)

Young generation

СБЕРБАНК ТЕХНОЛОГИИ

- 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" (minor GC).

Old generation

СБЕРБАНК ТЕХНОЛОГИИ

- Объекты которые переживают "minor garbage 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.
- Сборка мусора в "old generation" (major garbage collection) происходит не часто, но когда происходит, занимает много времени

○Permanent generation/MetaSpace

СБЕРБАНК ТЕХНОЛОГИИ

- Хранятся метаданные, классы, интернированные строки, и т.д.

Young generation

СБЕРБАНК ТЕХНОЛОГИИ

Young generation делится на:

- Eden - память, где объекты алоцируются.
- Survivor space 1,2 - объекты, которые выжили при предыдущей сборке мусора, но перед отправкой в "old generation"

minor GC

СБЕРБАНК ТЕХНОЛОГИИ

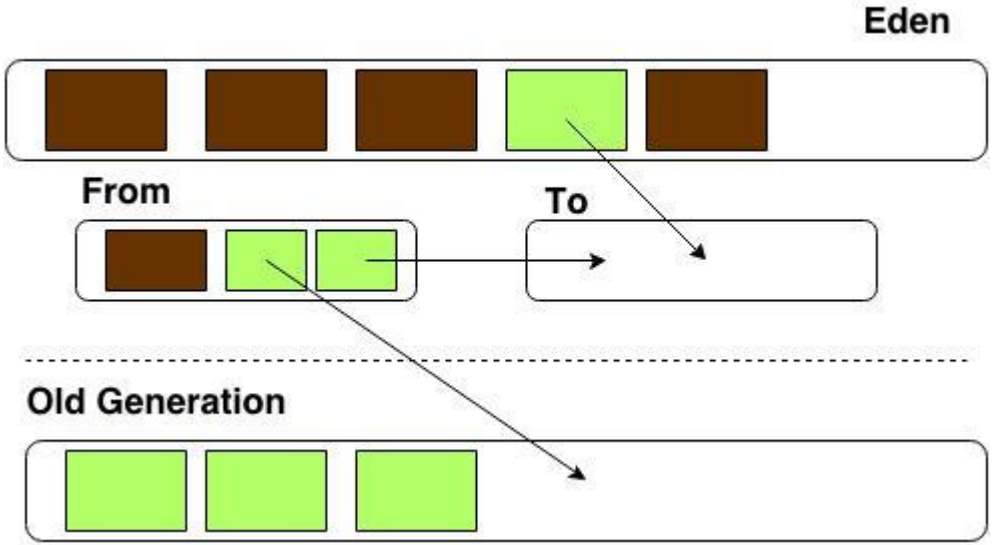
Алгоритм работы minor GC:

- Приложение приостанавливается.
- Живые объекты из "Eden" копируются в " Survivor 2".
- Живые объекты из "Survivor 1" копируются в " Survivor 2" или в "old generation", если они достаточно старые.
- "Eden" и " Survivor 1 " очищаются, так как в них остался только мусор.
- " Survivor 1 " и " Survivor 2" меняются местами
- Приложение возобновляет работу

minor GC

СБЕРБАНК ТЕХНОЛОГИИ

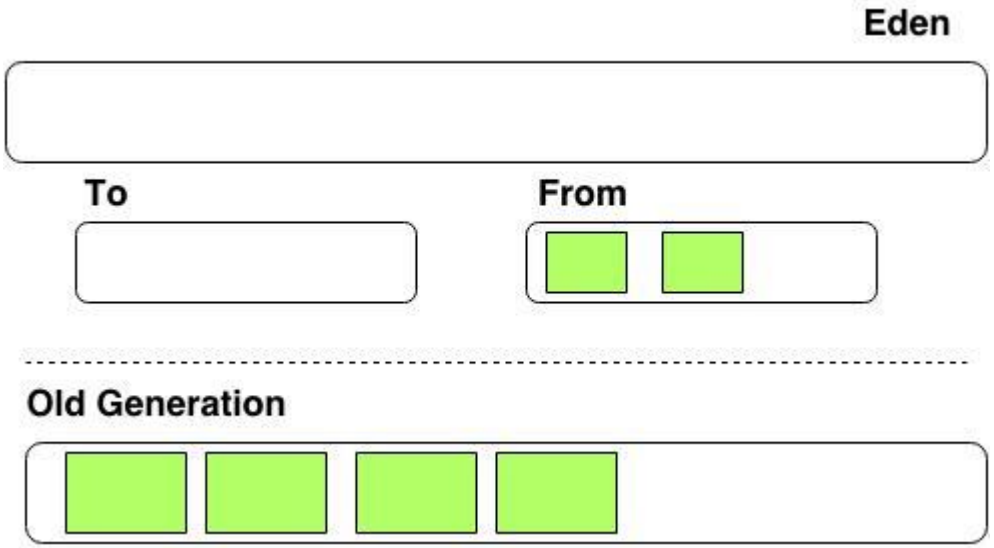
Сборка мусора



minor GC

СБЕРБАНК ТЕХНОЛОГИИ

После сборки мусора



minor GC

СБЕРБАНК ТЕХНОЛОГИИ

- Для того, что бы "minor GC" проходил быстро, нужно что бы при нем не приходилось сканировать ”old generation”
- Согласно "weak generational hypothesis" их должно быть мало
- Память в "old generation" разбивается на карты (cards). Card table - это массив, 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”.
- В итоге при "minor GC" для выявления ссылок "old-to-new" сканируется только объекты которые находятся в "грязных" картах.

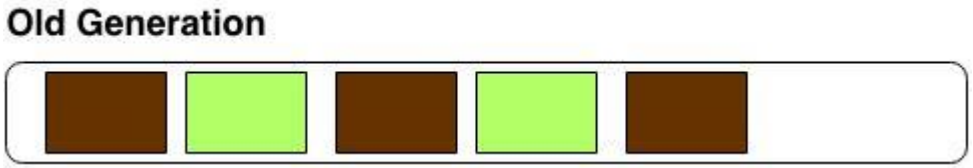
major GC

СБЕРБАНК ТЕХНОЛОГИИ

Алгоритм работы major GC:

- Приложение приостанавливается.
- Сборщик проходится по дереву объектов в "old generation", помечая живые объекты
- Все живые объекты сдвигаются к началу "old generation", мусор становится одним куском памяти, который находится сразу за последним живым объектом
- Приложение возобновляет работу

Сборка мусора



major GC

СБЕРБАНК ТЕХНОЛОГИИ

После сборки мусора

Параметры

СБЕРБАНК ТЕХНОЛОГИИ

Параметр	Описание
-Xms	Sets the initial heap size for when the JVM starts.
-Xmx	Sets the maximum heap size.
-Xmn	Sets the size of the Young Generation.
-XX:PermSize	Sets the starting size of the Permanent Generation. (< Java 8)
-XX:MaxPermSize	Sets the maximum size of the Permanent Generation (< Java 8)
-XX:MaxMetaspaceSize	Sets the maximum size of the Metaspace (>= Java 8)

ТИПЫ GC

СБЕРБАНК ТЕХНОЛОГИИ

- Serial GC
- Parallel GC
- CMS GC
- G1 GC

Serial GC

СБЕРБАНК ТЕХНОЛОГИИ

-XX:+UseSerialGC

- Последовательно выполняется minor GC и major GC
- Эффективен на компьютерах с небольшим количеством (один/два) процессоров
- Эффективен на компьютерах с большим количество виртуальных машин и запущенных JVM больше чем доступных процессоров

Используется для приложений не требовательных к длительным простоям вызванных сборкой мусора (~ 2 сек) и работающих на компьютерах с небольшим количеством процессоров и памяти(клиентские компьютеры)

Parallel GC

СБЕРБАНК ТЕХНОЛОГИИ

-XX:+UseParallelGC

несколько потоков по сборке young памяти, один поток для сборки old памяти

-XX:+UseParallelOldGC

несколько потоков по сборке young и old памяти

-XX:ParallelGCThreads=<desired number>

Parallel GC

СБЕРБАНК ТЕХНОЛОГИИ

- Плюсы:
  - Максимальная производительность
- Минусы:
  - Продолжительные паузы

Concurrent Mark Sweep (CMS)

СБЕРБАНК ТЕХНОЛОГИИ

-XX:+UseConcMarkSweepGC

-XX:ParallelCMSThreads=<n>

Несколько потоков по сборке мусора, сводит к минимуму перерывы из-за сборки мусора

- Плюсы:
  - Предсказуемость, Короткие паузы
- Минусы:
  - Снижение производительности

СБЕРБАНК ТЕХНОЛОГИИ

-XX:+UseG1GC

Задаваемые цели на длительность и частоту пауз
-XX:MaxGCPauseMillis=<num>
-XX:GCPauseIntervalMillis=<num>

Сборка мусора в фоновом режиме, сводит к минимуму перерывы из-за сборки мусора

- Плюсы:
  - Предсказуемость, короткие паузы
  - Подходит когда выделено очень много памяти (heap)
- Минусы:
  - Снижение производительности

G1 рекомендуется, если (не обязательно все):

- Нужна хорошая производительность
- Продолжительность пауз (<0.5-1s)
- Минимальная настройка
- Размер «кучи» >5Gb
- Занятость «кучи» >50%
- Скорость создания объектов серьезно варьируется
- Фрагментация «кучи»

G1 и CMS - **НЕ** рекомендуется, если

- Приложение уже работает надежно без серьезных проблем с производительностью “If it isnt broken, dont fix it!”
- Требуется максимально возможная производительность

ИТОГО

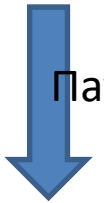
СБЕРБАНК ТЕХНОЛОГИИ

- SerialGC – последовательная сборка молодого и старого поколений
- ParallelGC – максимальная производительность, параллельная сборка молодого и старого поколений
- CMS – предсказуемость, по возможности, сборка мусора в фоновом режиме
- G1 – предсказуемость, сборка мусора в фоновом режиме, слабо подвержен фрагментации

Как выбирать GC?

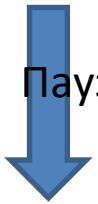
СБЕРБАНК ТЕХНОЛОГИИ

1) **ParallelGC**



Паузы не устраивают

2) **G1**



Паузы все еще не устраивают, Heap < 5Gb

3) **CMS**

Материалы

СБЕРБАНК ТЕХНОЛОГИИ

JIT

http://jeeconf.com/archive/jeeconf-2013/materials/jit-compiler

GC

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

https://ggenikus.github.io/blog/2014/05/04/gc/