

Garbage Collection

Попов Матвей

План

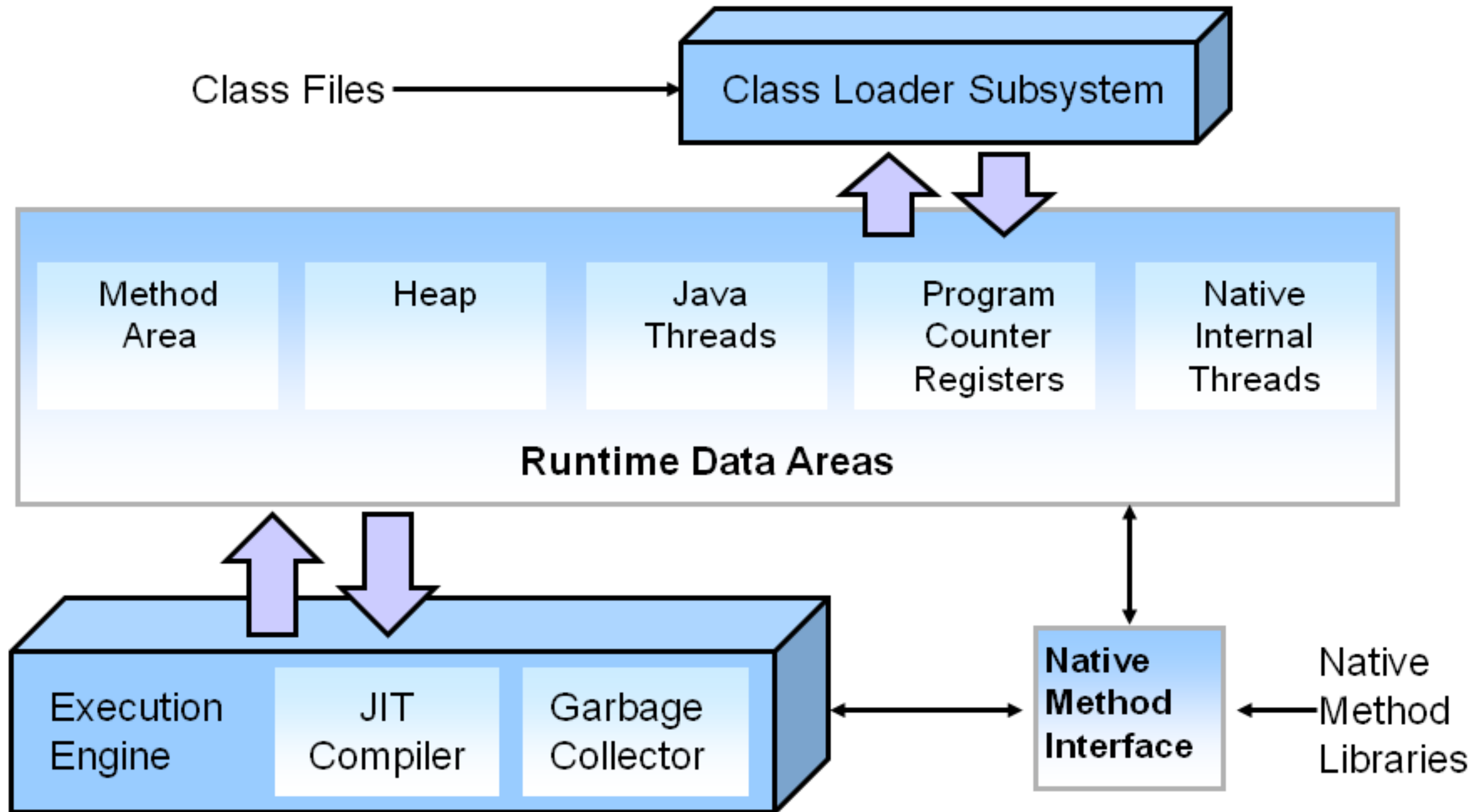
- Что такое Garbage Collection?
- Почему существуют несколько GC?
- Какие существуют GC в JDK?
- В чем их различие?
- Как мониторить за работой GC?
- А что там в других языках?

Что такое Garbage Collection?

Сборка мусора

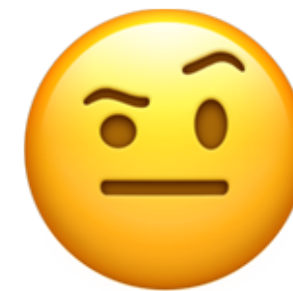
Процесс **автоматического** управления памятью.
Освобождение памяти выполняется автоматически
специальным компонентом JVM - Garbage Collector.

HotSpot JVM: Architecture



А оно мне надо?

Резонный вопрос



- Далеко не любой программе требуется тонкая настройка GC
- Редко кто заметит что отклик поменялся на десяток ms
- А что если очистка занимает секунды?
- Как можно работать с тем, чего не знаешь?

Разделяй и властвуй

**JVM разделяет память на две
области**

- Куча(heap) - в которой хранятся данные приложения
- Не-куча(non-heap) - в которой хранится код программы и вспомогательные данные

- Состояние non-hear в целом статично
- Non-hear глобально слабо поддается оптимизациям
- Механизмы функционирования non-hear не будем рассматривать

Из поколения в поколение

- У разных GC разные цели
- Но их обычно объединяет одна слабая гипотеза

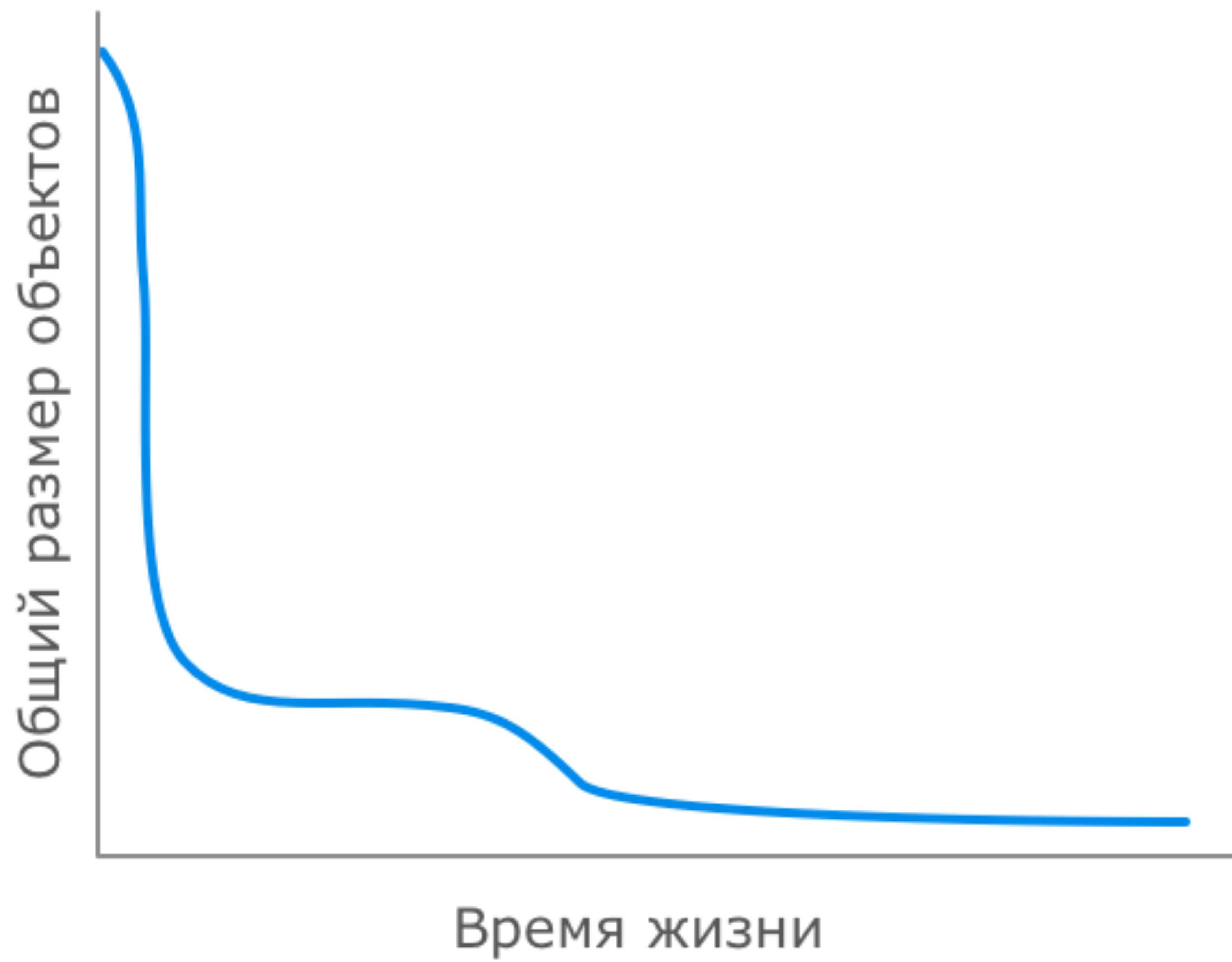
Слабая гипотеза о поколениях

Вероятность **смерти** объекта, как функция от возраста, **снижается** очень быстро.

Что это значит?

**Подавляющее большинство
объектов живут крайне недолго**

**Также это означает, что чем дольше
прожил объект, тем выше вероятность
того, что он будет жить и дальше**



Почему это так?

- Часто объекты создаются на очень короткое время(итераторы, локальные переменные, результаты боксинга)
- Далее идут объекты для более-менее долгих вычислений
- Объекты-старожилы - постоянные данные программы

**Тут возникает идея
разделения объектов**

- Младшее поколение(young generation)
- Старшее поколение(old generation)

**Процесс сборки тоже
разделяется**

- Малая сборка(minor GC)
- Полная сборка(full GC)

Важные факты:

- Разделение памяти не просто условное, она физически разделена на регионы
- Объекты по мере выживания переходят в следующие поколения
- В старшем поколении объект может прожить до конца работы приложения

**Вам быстро, дешево или
качественно?**

- Хочется иметь сборщик, который как можно быстрее избавлялся от мусора
- Работа сборщика мусора не бесплатная
- Она оплачивается ресурсами и задержками программы

**Существуют факторы
эффективности ГС**

1. Максимальная задержка

Максимальное время, на которое сборщик приостанавливает выполнение программы для выполнения одной сборки(stop-the-world)

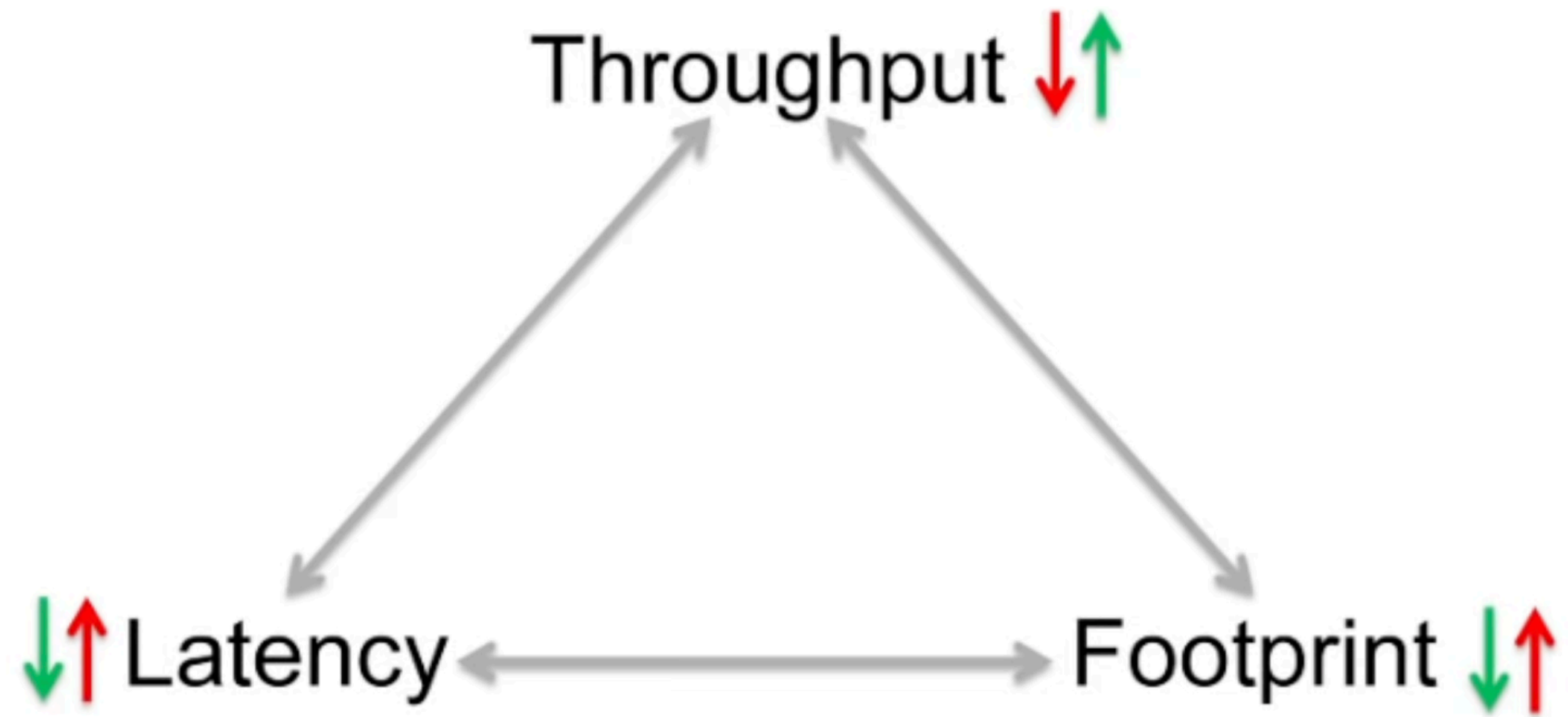
2. Пропускная способность

Отношение общего времени работы программы к общему времени простоя, вызванного сборкой мусора, на длительном промежутке времени

3. Потребляемые ресурсы

Объем ресурсов процессора и/или дополнительной памяти, потребляемых сборщиком

Оптимизируйте 2 из 3



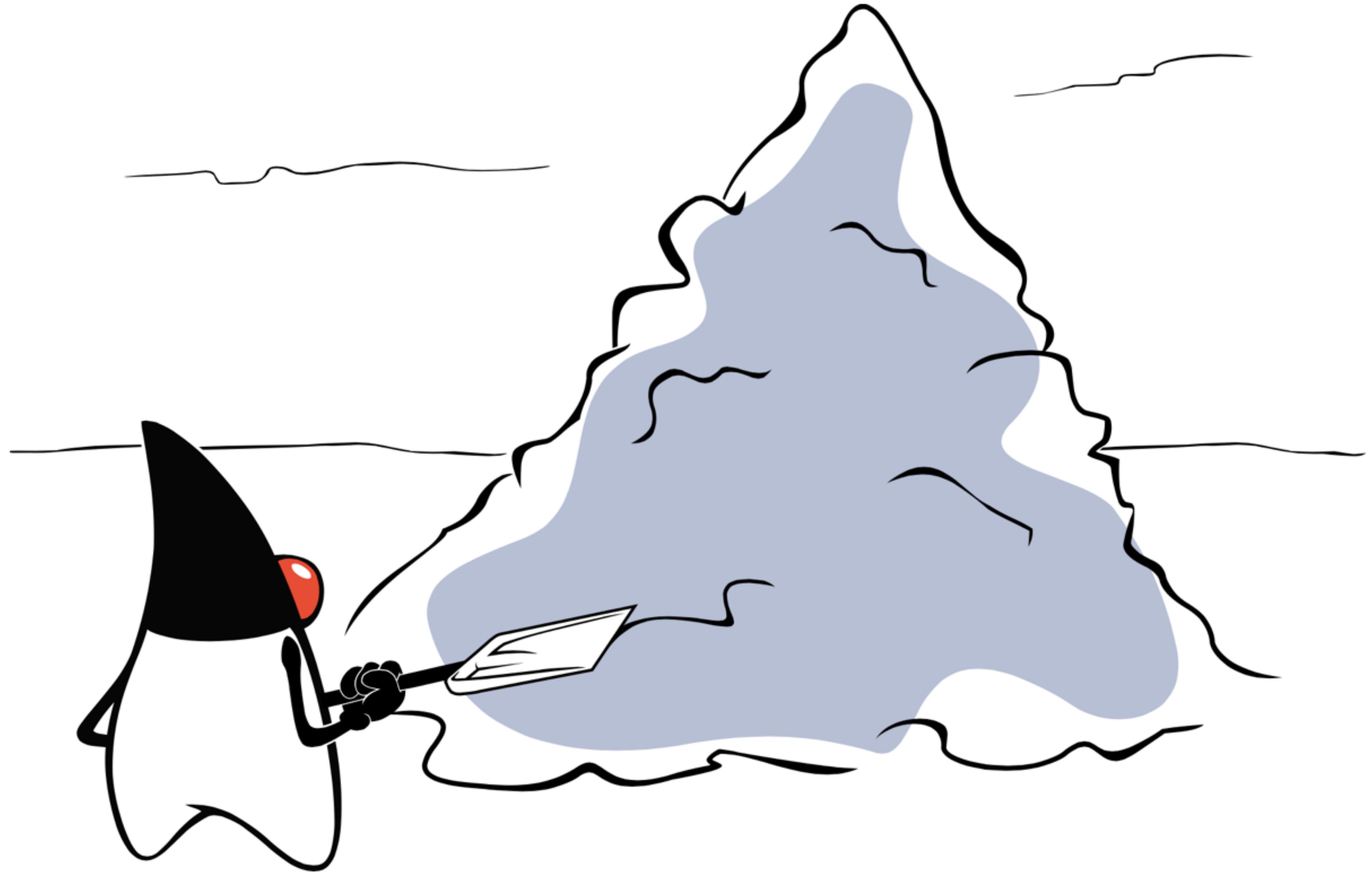
Как искать мусор?

**Для этого есть корни сборки
мусора(GC roots)**

GC roots:

- Классы, загружаемые системным загрузчиком классов
- Регистры живых потоков
- Локальные переменные и параметры функций
- Переменный и параметры JNI
- Объекты, применяемые в качестве монитора синхронизации
- Объекты, удерживаемые из сборки мусора JVM для своих целей

Какие есть GC?



**Java HotSpot VM предоставляет
7 сборщиков мусора**

1. Serial (последовательный)
2. Parallel (параллельный)
3. Concurrent Mark Sweep (CMS)
4. Garbage-First (G1)
5. Epsilon GC
6. ZGC
7. Shenandoah GC

Начнем с простого

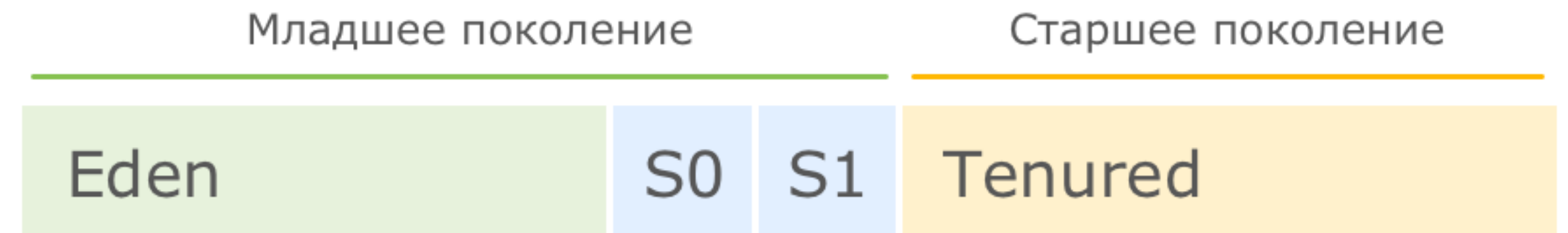
Serial GC и Parallel GC

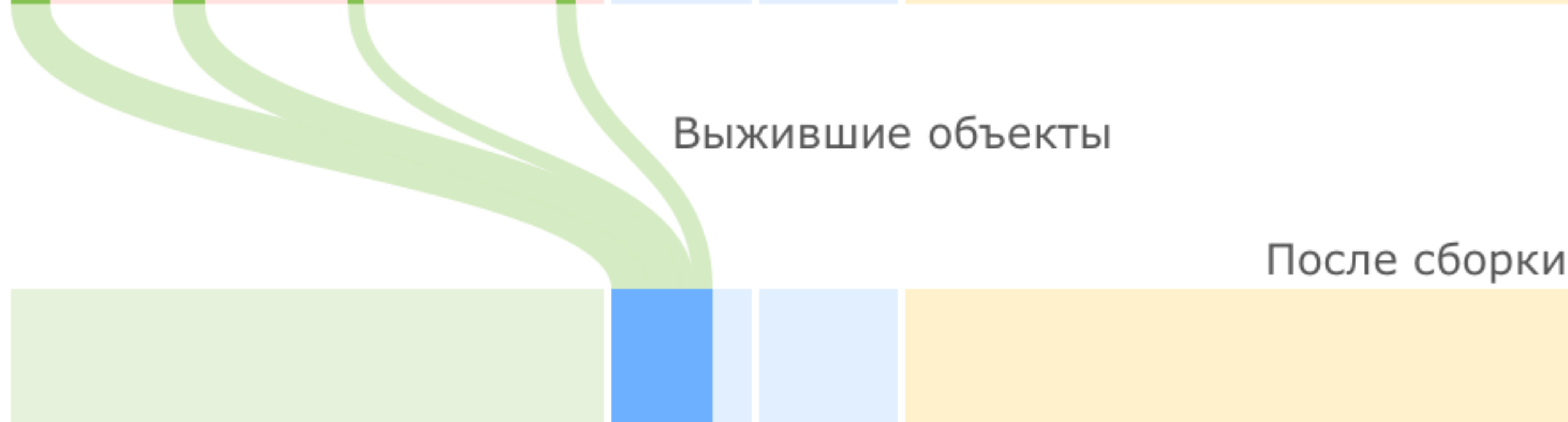
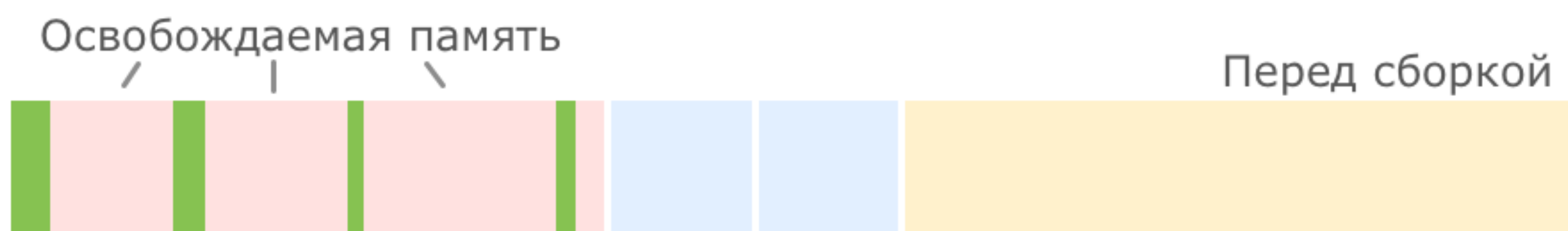
Serial GC

- Последовательный сборщик
- Самый просто сборщик
- Самый старый сборщик
- Используется когда heap очень мал
- Используется когда мало ресурсов
- Включается `-XX:+UseSerialGC`

Принципы работы

- Куча разбивается на 4 региона
- Три относятся к младшему: Eden, S0, S1
- Один к старшему: Tenured





Перед сборкой



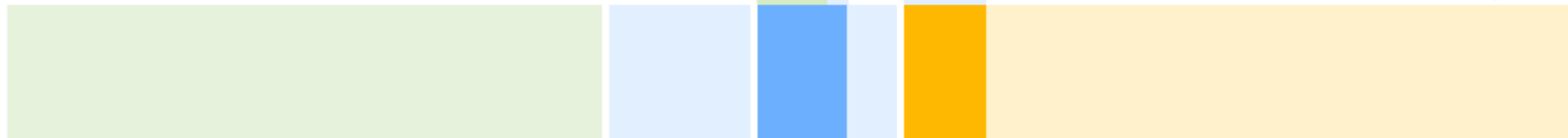
После сборки



Перед сборкой



После сборки



Перед сборкой



После сборки



Достоинства и недостатки

- Непритязательность по части ресурсов компьютера
- Долгие паузы на сборку при заметных объемах памяти
- Можно тонко настраивать каждый регион heap
- Выполняет работу в одном потоке

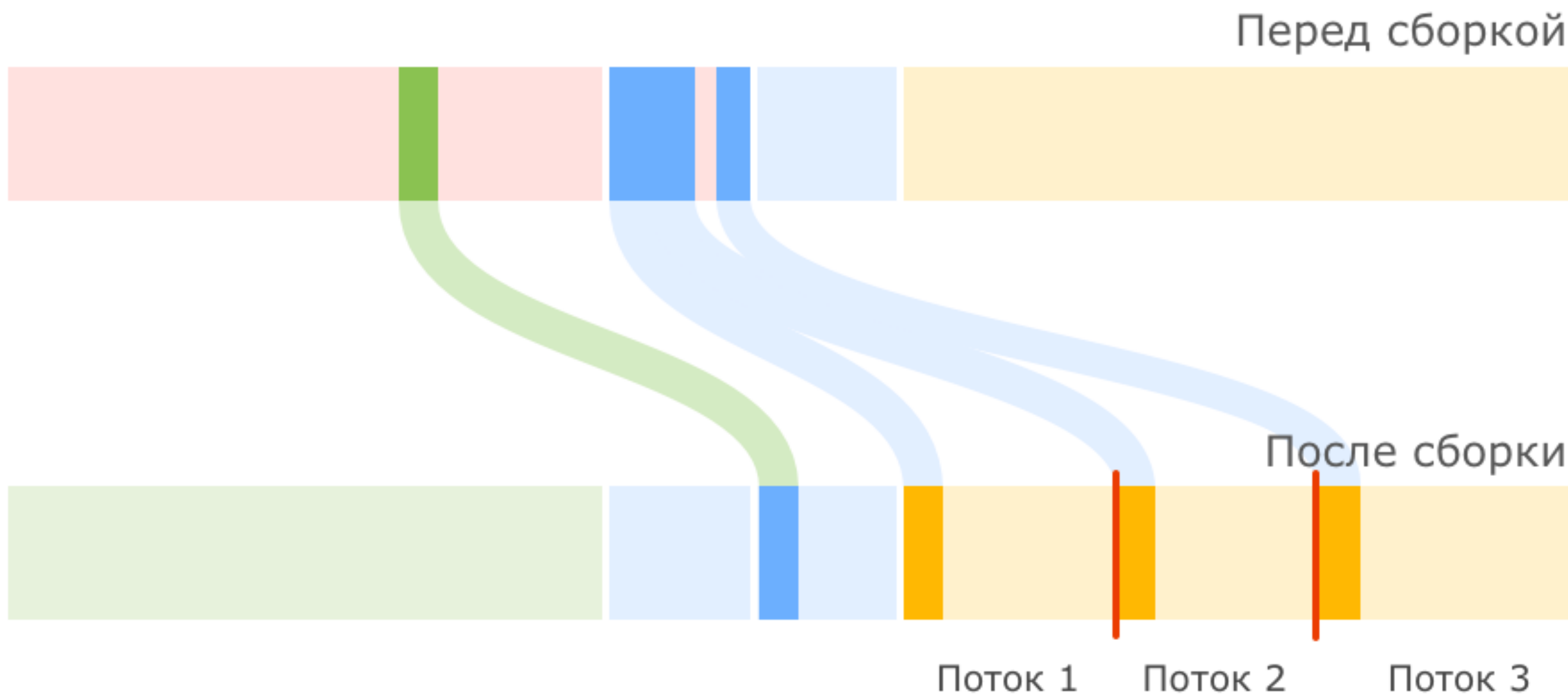
Когда использовать?

- Когда вашему приложению не требуется большой размер кучи
- Oracle указывает до 100МБ
- Приложение не чувствительно к коротким остановкам
- Доступно только одно ядро процессора

Parallel GC

- Используется те же самые подходы к организации heap что и Serial GC
- Сборкой мусора занимаются несколько потоков параллельно
- Сборщик может самостоятельно подстраиваться под требуемые параметры производительности
- Включается `-XX:+UseParallelGC`

Принципы работы



Преимущества и недостатки

- Есть настройки, ориентированные на достижение необходимой эффективности GC
- Оставляет возможность настройки размера регионов
- Паузы обычно короче чем в Serial GC
- Будет присутствовать небольшая фрагментация памяти
- Нет скрытых накладных ресурсов

**Переходим к более сложным
сборщикам**

CMS GC и G1 GC

«Mostly concurrent collectors»

Что это значит?

- Часть своей работы выполняют параллельно с основными потоками приложения
- Потоки GC конкурируют за ресурсы процессора с основными потоками
- Есть свои преимущества и недостатки

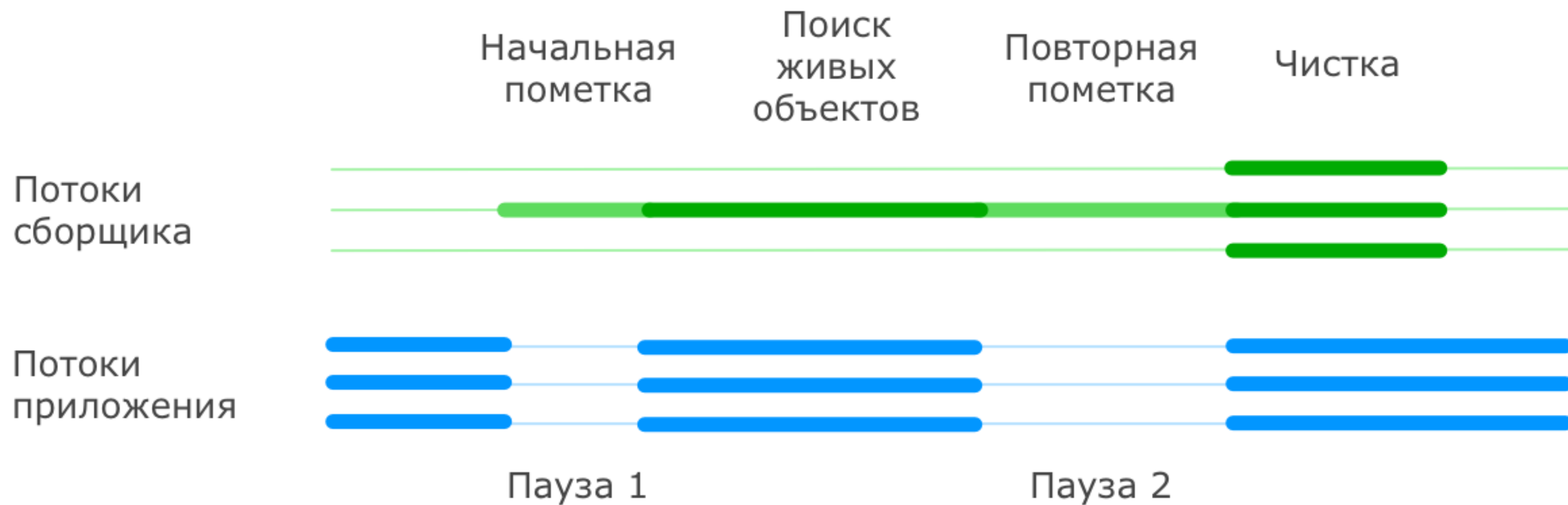
CMS GC

Concurrent Mark Sweep GC

- Появился вместе с Parallel GC
- Также раньше существовала альтернатива Incremental GC
- Предназначен для приложений чувствительным к STW паузам
- Предназначен для приложений имеющих доступ к нескольким ядрам
- Включается `-XX:+UseCMSGC`

Принципы работы

- Использует такие же принципы малой сборки мусора
- Отличается полной сборкой мусора
- В CMS она называется major GC
- Major не затрагивает объекты младшего поколения
- Всегда работает в фоне
- Не дожидается заполнения старшего региона



Ситуации STW

- Малая сборка мусора
- Начальная фаза поиска живых объектов при старшей сборке(initial mark pause)
- Фаза дополнения набора живых объектов при старшей сборке(remark pause)
- Когда сборщик не успевает очистить Tenured до того как она закончится(concurrent mode failure)

Достоинства и недостатки

- Ориентированность на минимизацию времени простоя
- Жертвует ресурсами процессора
- Жертвует общей пропускной способностью
- Происходит фрагментация Tenured
- Возможны сбои конкурентного режима
- Нужно выделять больше памяти(на 20%)

Когда использовать?

- Приложениям, использующий большой объем долгоживущих данных
- Но лучше G1 GC

G1 GC

Garbage First GC

- Не является продолжением Serial/Parallel/CMS
- Использует существенно отличающийся подход к очистке памяти
- Позиционируется как сборщик для приложений с большими кучами
- Включается `-XX:+UseG1GC`

Принципы работы

- Память разбивается на множество регионов одинаково размера
- Разделение регионов логическое
- Существуют громадные(humongous) регионы



Как устроена малая сборка мусора?

- Выполняются для очистки и переноса
- Над переносом трудятся несколько потоков
- Очистка выполняется не на всем поколении, а только на части регионов
- Сборщик выбирает регионы, где скопилось наибольшее количество мусора

Цикл пометки:

- Initial Mark
- Concurrent Marking
- Remark
- Cleanup

Очищаемые регионы

Перед сборкой



После сборки



Ситуации STW:

- Процессы переноса объектов между поколениями
- Короткая фаза начальной пометки корней в рамках цикла пометки
- Более длинная пауза в конце фазы remark и в начале фазы cleanup

Достоинства и недостатки

- Более аккуратно предсказывает размеры пауз
- Не фрагментирует память
- Использует не мало ресурсов процессора
- Страдает пропускная способность приложения

ZGC

Зачем еще один?

Цели при проектировании:

- Поддерживать паузы STW на уровне меньше 1мс
- Сделать так, чтобы паузы не увеличивались с ростом кучи, объектов или корневых ссылок
- Поддерживать кучи размером до 16ТБ

Что же получилось?

Начнем с виртуальной памяти

Виртуальная память

Метод управления **памятью** компьютера, позволяющий выполнять программы, требующие **больше** оперативной памяти, чем **имеется** в компьютере, путем автоматического **перемещения** частей программы между **основной** памятью и **вторичным** хранилищем.

Зачем нам эта информация?

В контексте ZGC

- Разные виртуальные страницы могут указывать на одну физическую
- Создание таких двойников не приводит к увеличению потребления физической памяти
- Один и тот же объект в физической памяти может иметь несколько виртуальных адресов
- ZGC активно эксплуатирует эту особенность

**Теперь разберемся с
указателями на объекты**

**ZGC использует подход, называемый
раскрашиванием указателей**

Что это значит?

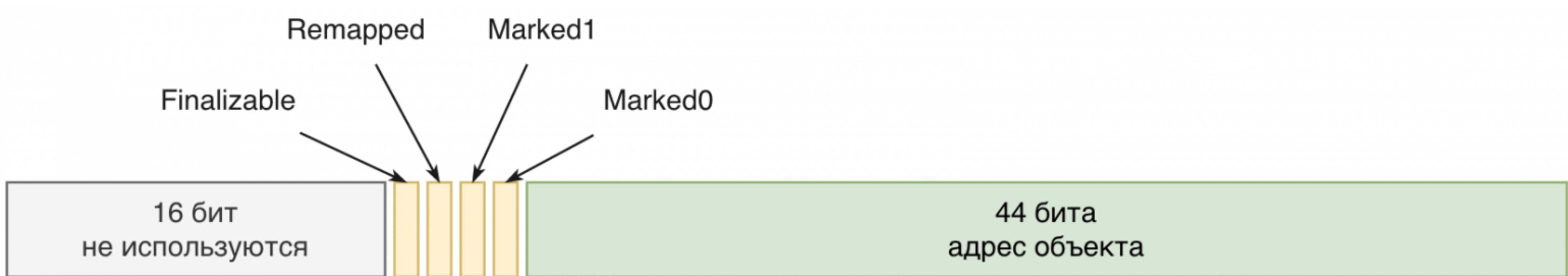
- Каждый 64 битный указатель содержит не только адрес памяти
- Указатель дополнительно содержит другие метаданные

Из чего состоит указатель?

- Под адрес в указателе выделяется от 42 до 44 младших бит
- Еще 4 бита выделены под метаданные
- Остальные 16 битов всегда равны 0 и никогда не используются

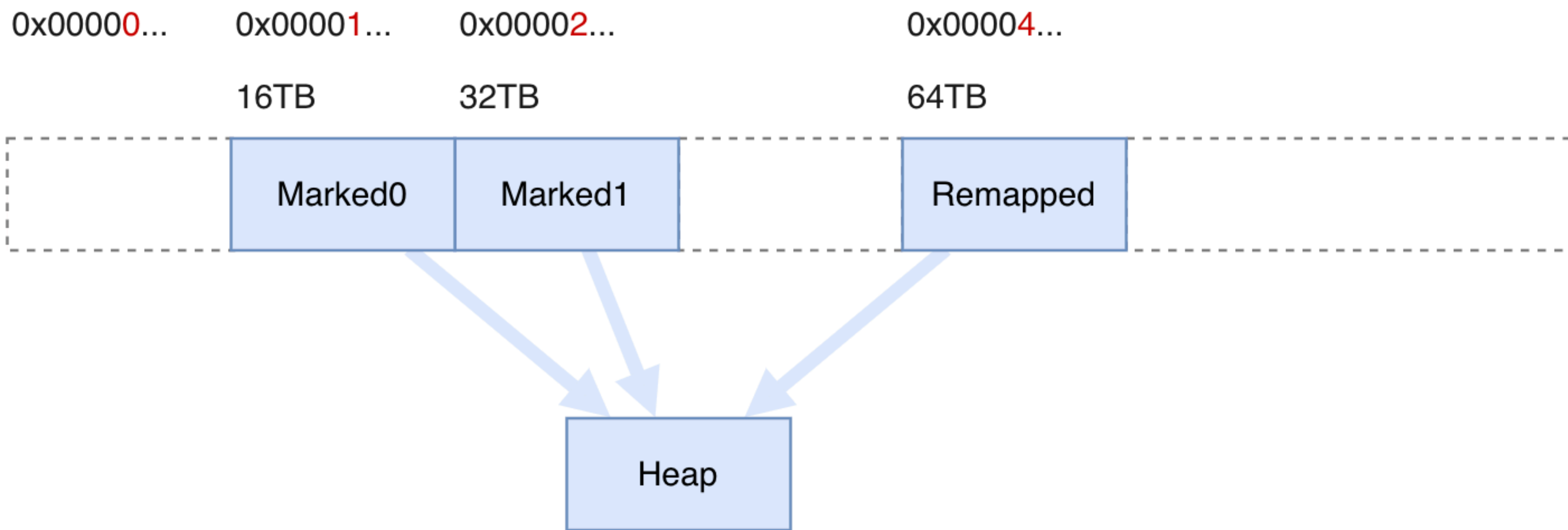
Что за метаданные?

- Marked0(0001) и Marked1(0010) - используются для пометки указателей на разных фазах сборки
- Remapped(0100) - указатель помечается этим битом в случае, если адрес в указателе является окончательным и не должен модифицироваться в рамках текущего цикла сборки
- Finalizable(1000) - этим битом помечаются объекты, достижимые только из финализатора
- Комбинация этих флагов определяет состояние указателя, которое при описании ZGC называется его «цветом».



**Теперь объединим это с
виртуальной памятью**

- Нулевой адрес(первые 44 бита) с каким-то «красочным» битом будет представлять собой начало 16-терабайтной области в виртуальной памяти
- Все эти области проецируются на одну и ту же область физической памяти - кучу JVM



И что это нам дает?

**Устанавливая или снимая какой-либо
из **битов** метаданных мы переносимся
в **другую** область виртуальной памяти**

Но при этом **получаем** виртуальный **указатель** на тот же самый объект, на который **ссылались** до перекрашивания

Удобно



**Теперь переходим к
алгоритму сборки?**

Нет

Барьеры

Барьер

Просто **функция**, которая принимает указатель на объект в памяти, **анализирует** цвет этого указателя, в зависимости от цвета **выполняет** какие-либо **действия** с этим указателем или даже с самим объектом, на который он ссылается, после чего **возвращает** актуальное значение указателя, которое необходимо **использовать** для доступа к объекту.

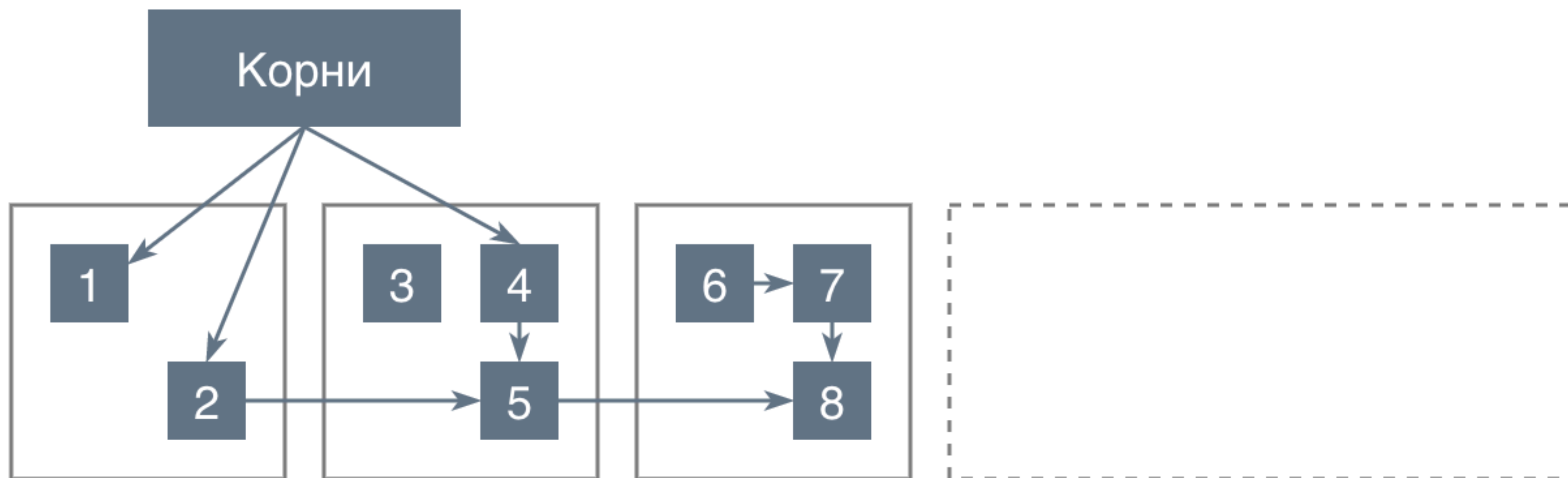
Важные особенности

- Используется во время конкурентных фаз сборки мусора
- Выполняется также в рамках основных потоков приложения

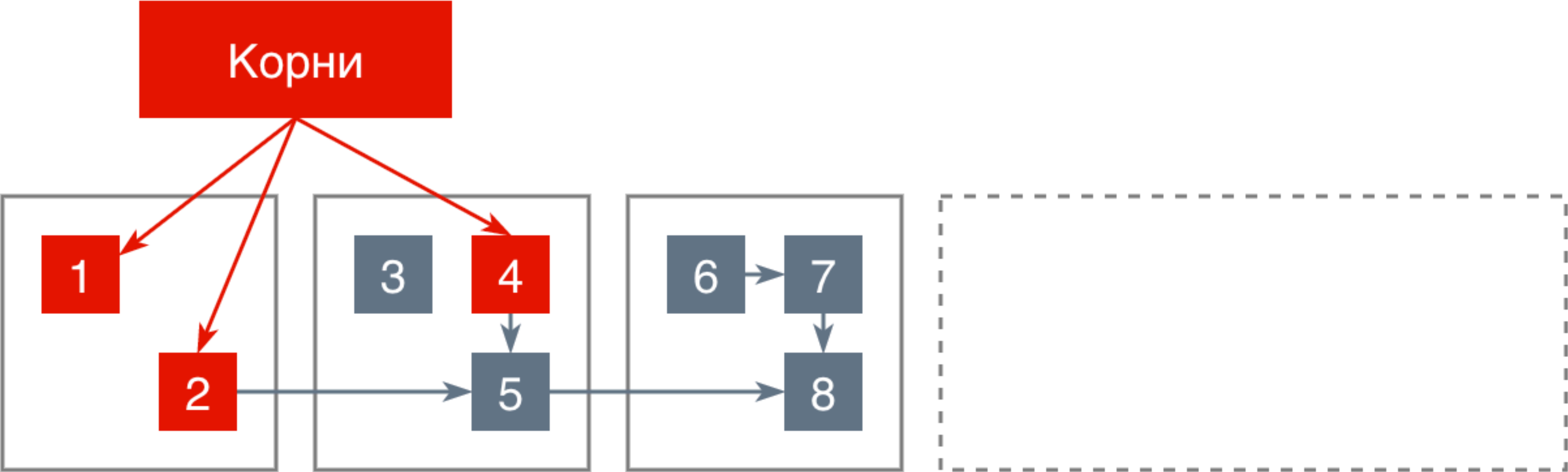
Как работает ZGC?

Этап 1: Поиск живых объектов

Начальное состояние кучи



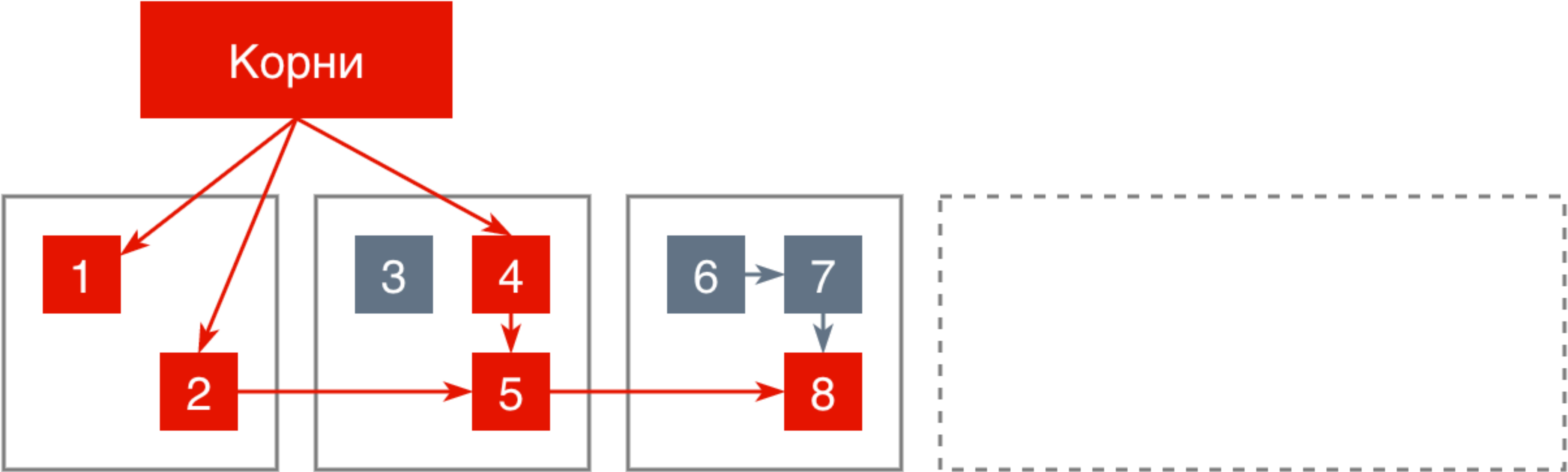
Фаза Pause Mark Start



Фаза Concurrent Map

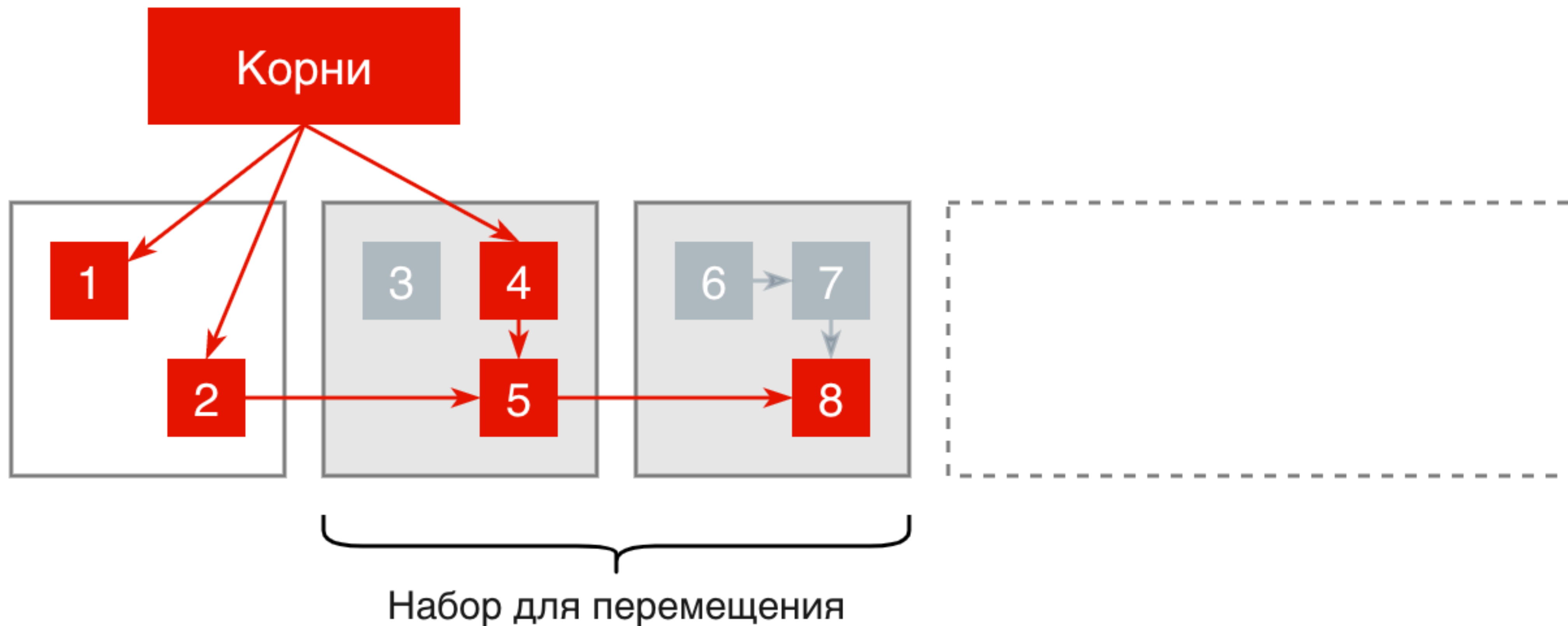
- Продолжается поиск живых объектов, но в concurrent режиме
- В этот период активно используются барьеры
- Барьеры красят все указатели, по которым в это время происходит доступ к объектам

Фаза Pause Mark End

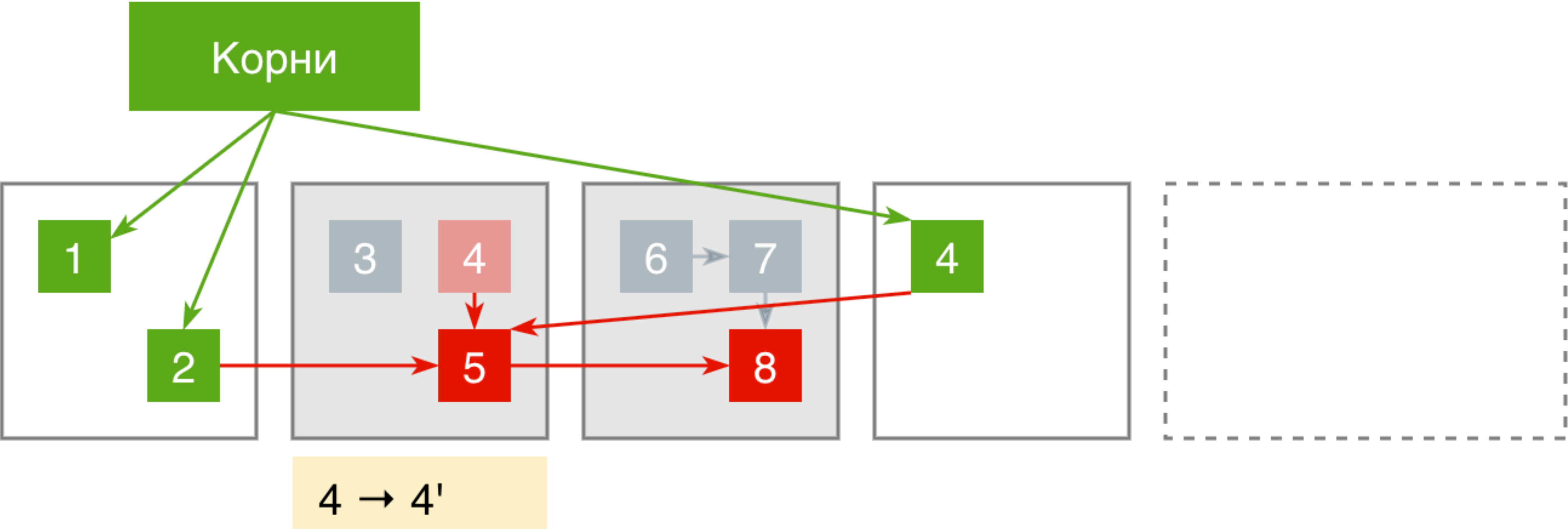


Этап 2: Перемещение

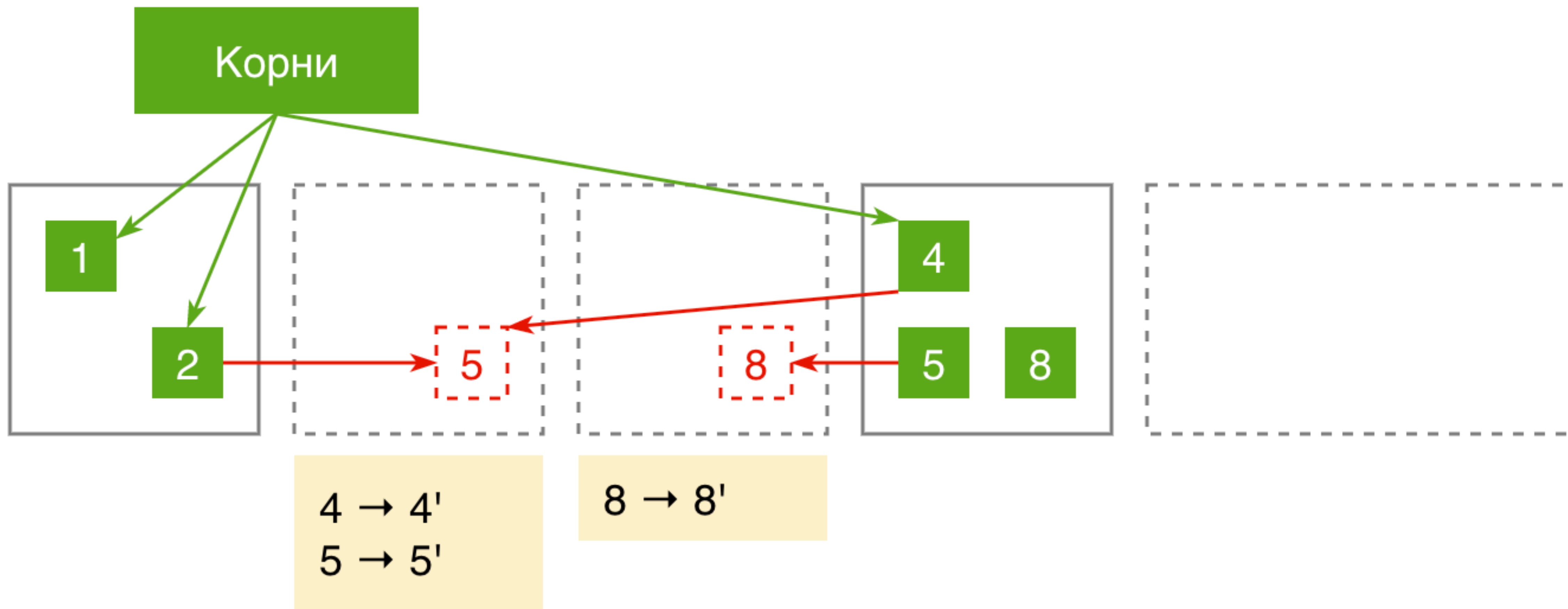
Фаза Concurrent Prepare for Relocate

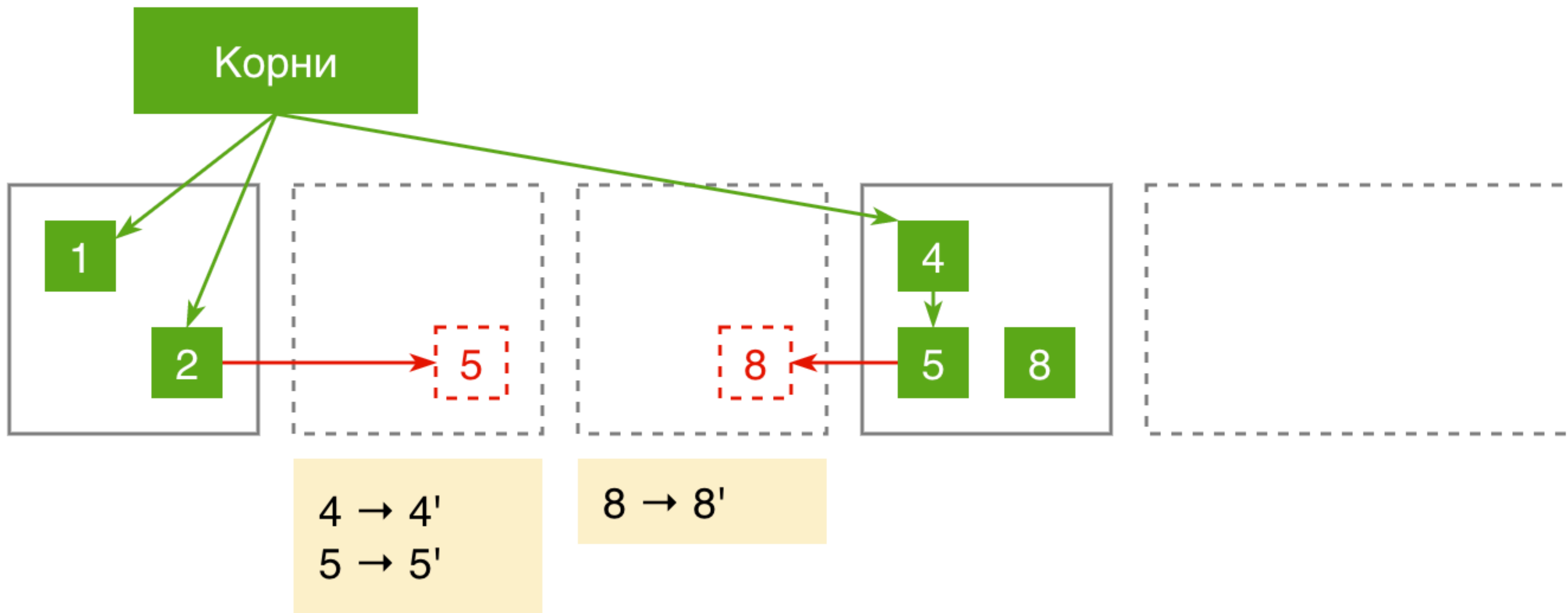


Фаза Pause Relocate Start



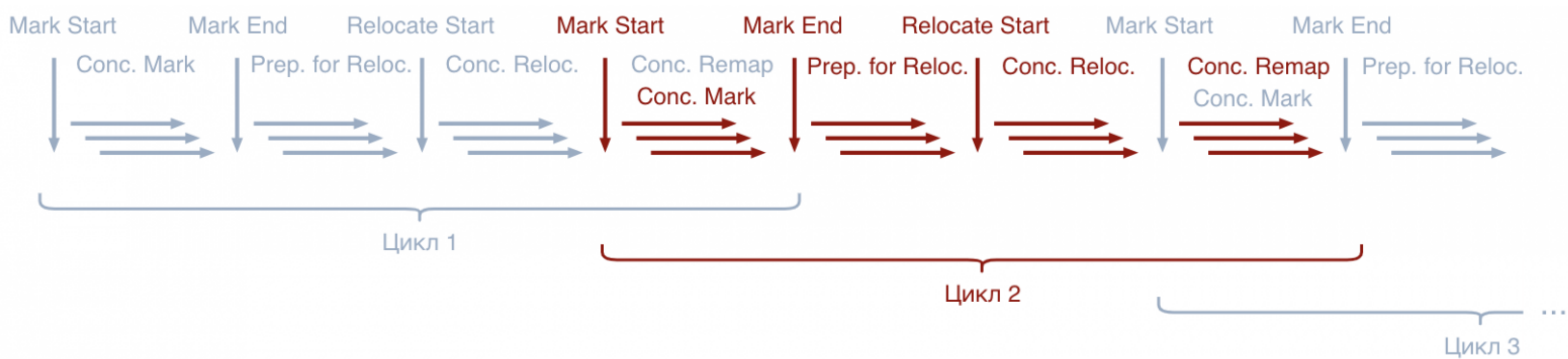
Фаза Concurrent Relocate



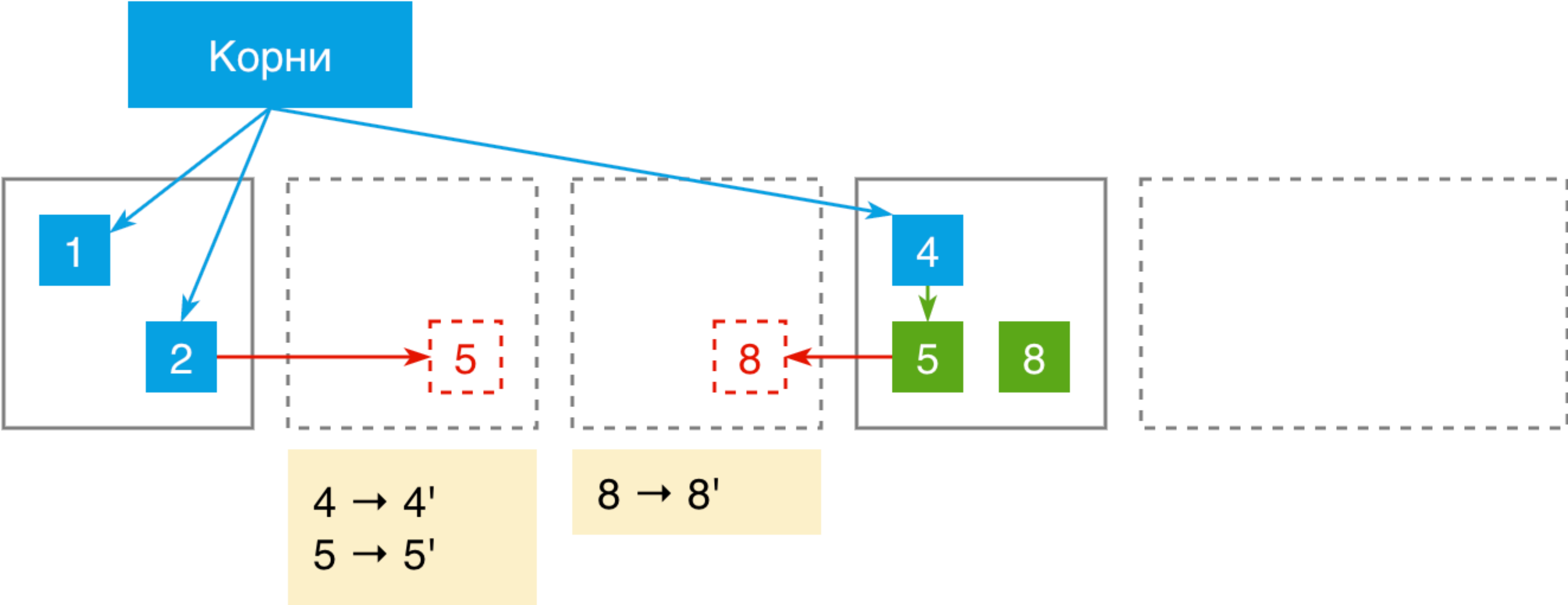


Фаза Concurrent Remap

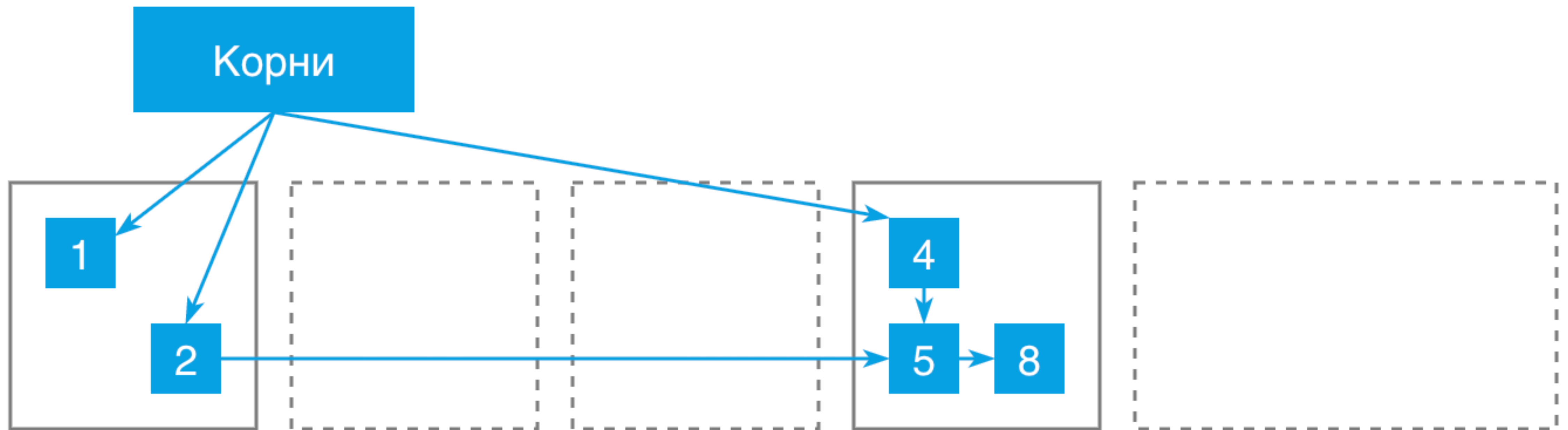
- Все объекты уже оказываются перемещенными в целевые области памяти
- Остались зависшие указатели на объекты в освобожденных регионах памяти
- Чтобы все исправить, нужно совершить еще один обход графа объектов
- Эта фаза совмещается с фазой Concurrent Mark следующего цикла сборки



Фаза Pause Mark Start следующего цикла сборки



Итог



Достоинства и недостатки

- Позволяет добиться субмиллисекундных пауз
- Даже на очень больших кучах
- Может страдать пропускная способность приложения
- Барьеры совсем не бесплатные
- Требуется большое количество свободных регионов памяти

Epsilon GC

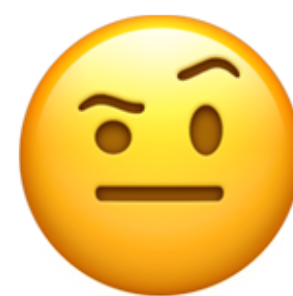
**Можно описать коротко - Epsilon
GC вообще не собирает мусор**

Это как?



- В JVM для размещения новых объектов используются *TLAB'ы* (*thread-local allocation buffers*)
- Для так называемых *громоздких объектов* (*humongous objects*), не помещающихся в буфер, в куче запрашиваются блоки памяти специально под них
- Когда потоку не удастся получить очередной буффер, то Epsilon GC просто генерирует `OutOfMemoryError` и завершает процесс.

Достоинства и недостатки



- Отказ от сборки означает отказ от больших накладных расходов
- Может использоваться в приложениях, которые не мусорят
- Подходит коротко живущим приложения
- Можно использовать для анализа накладных ресурсов других GC

Мы на финишной прямой!



Shenandoah GC

Вопросы?

Материалы будут в README

Спасибо за внимание!