Управление памятью и сборщиком мусора в Go

Нина Пакшина

13/07/2023

Области памяти

СТЕК

КУЧА

Где сохранится переменная: в стеке или в куче?

From a correctness standpoint,
you don't need to know

(c) Golang FAQ

…не забивай свою красивую голову такой ерундой…

Что попадает в стек?

- Локальные переменные, объявленные внутри функции, такие как переменные базовых типов данных
- Аргументы функции
- Возвращаемые значения функции

Escape analysis

оптимизация компилятора, которая позволяет определить, следует ли выделить память для объекта на куче или же можно использовать стек для его хранения

go run -gcflags=-m main.go

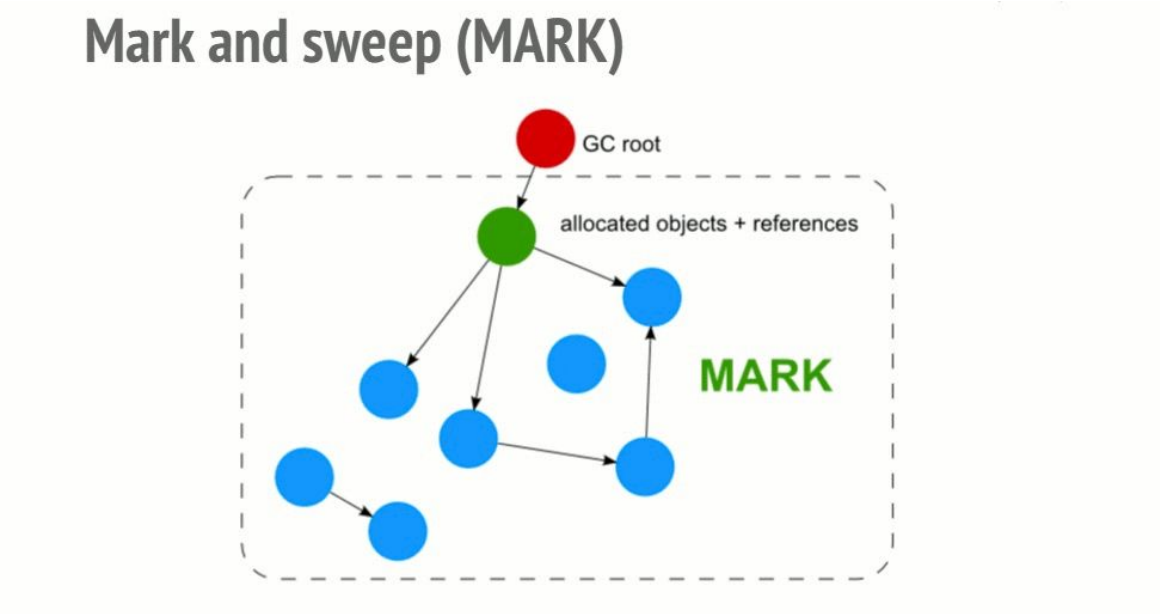
Углубиться

Сборщик мусора в Go

Mark & Sweep

[Описание]

[Реализация]



Какие ресурсы потребляет GC

Физическая память

Процессорное время

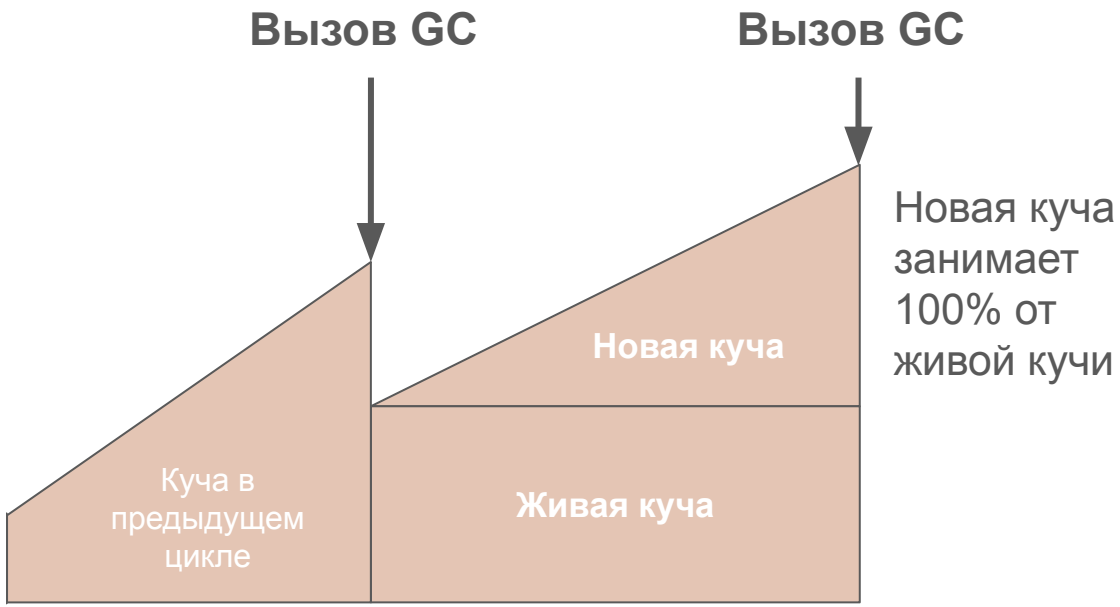
Stop the world

Ваше приложение полностью остановлено,

когда работает сборка мусора

В Go не полный Stop the world

Живая и новая память



Управление сборщиком мусора

`GOGC` - процент новой необработанной памяти кучи от живой памяти, при достижении которого будет запущена сборка мусора.

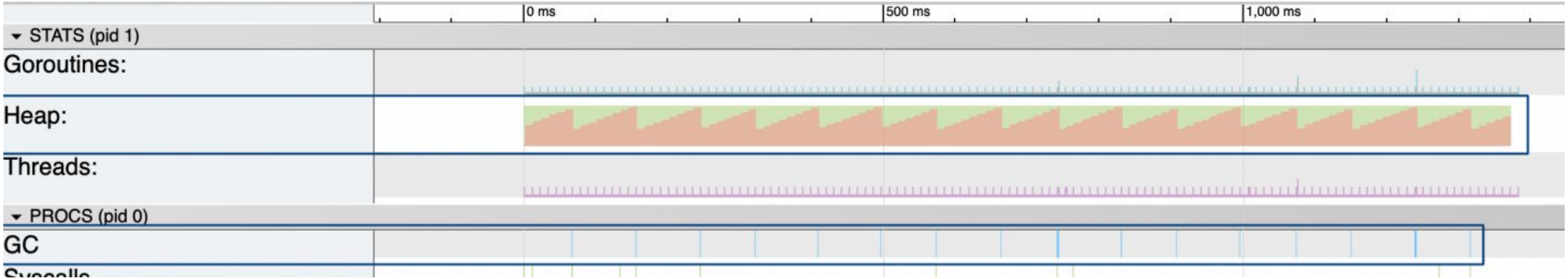
`GOGC` = 100:

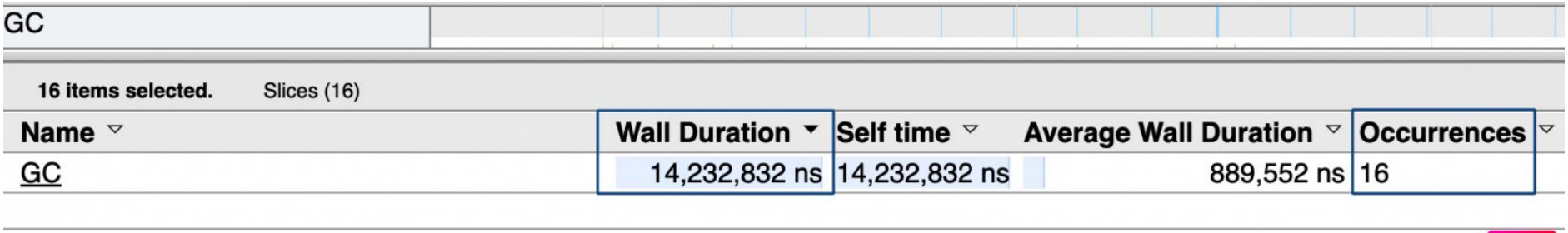
Сборка мусора будет запущена, когда объем новой памяти достигнет 100% от объема живой памяти кучи.

GOGC или debug.SetGCPercent

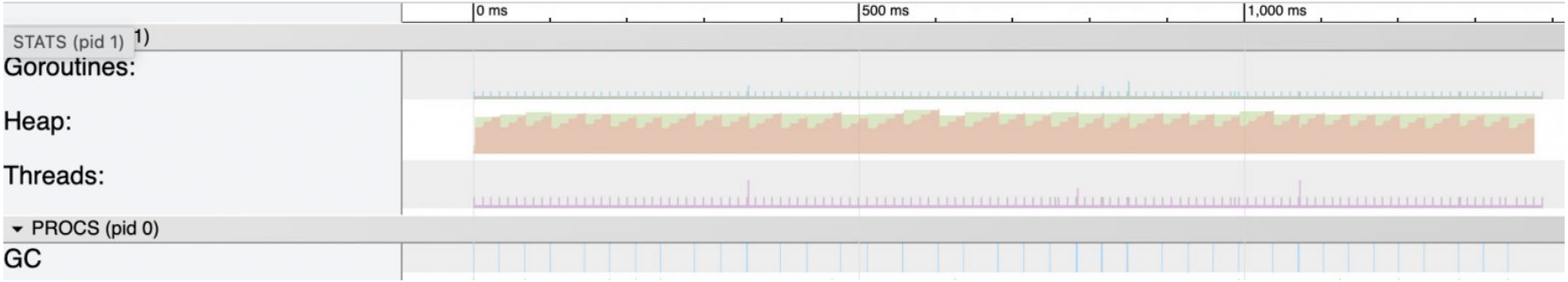
*Есть еще runtime.GC()

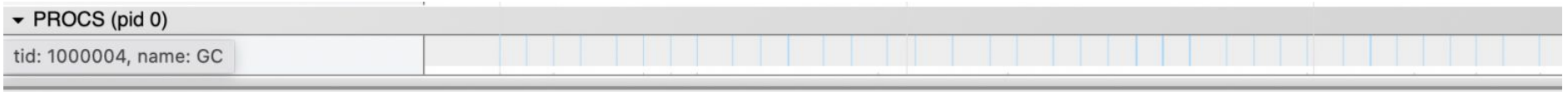
GOGC = 100


0 ms
500 ms
1,000 ms
STATS (pid 1)
Goroutines:
Heap:
Threads:
PROCS (pid 0)
GC


GC
16 items selected.
Slices (16)
Name
Wall Duration
Self time
Average Wall Duration
Occurrences
GC
14,232,832 ns
14,232,832 ns
889,552 ns
16

GOGC = 10

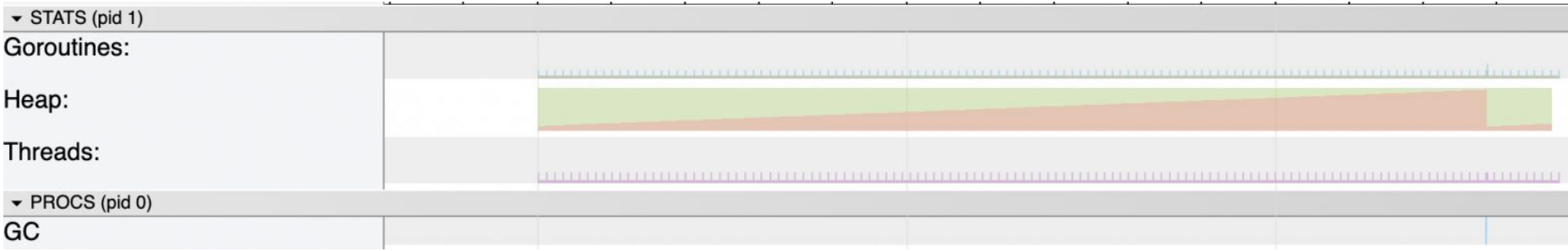


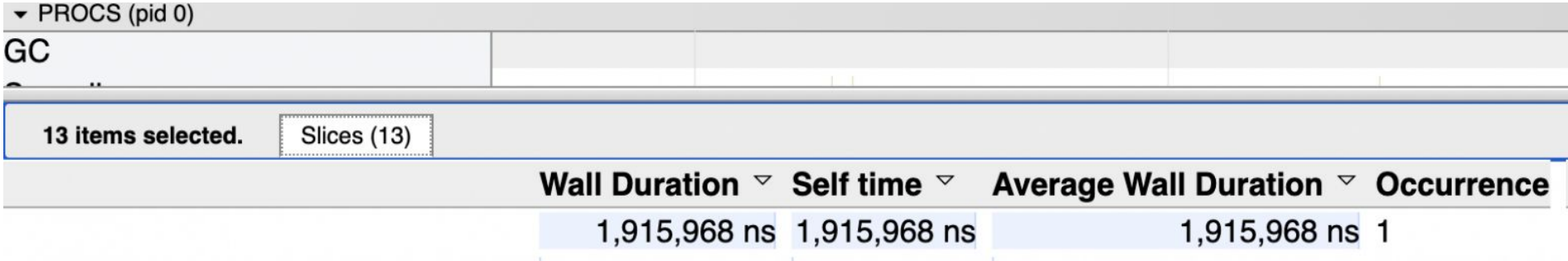


38 items selected. Slices (38)

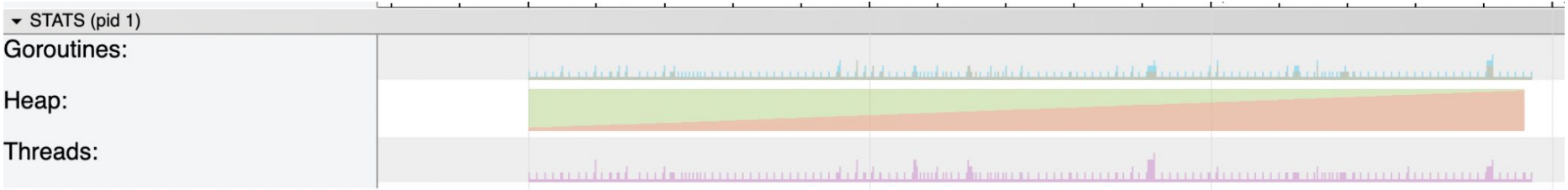
Name	Wall Duration	Self time	Average Wall Duration	Occurrences
GC	27,882,384 ns	27,882,384 ns	733,747 ns	38

GOGC = 1000

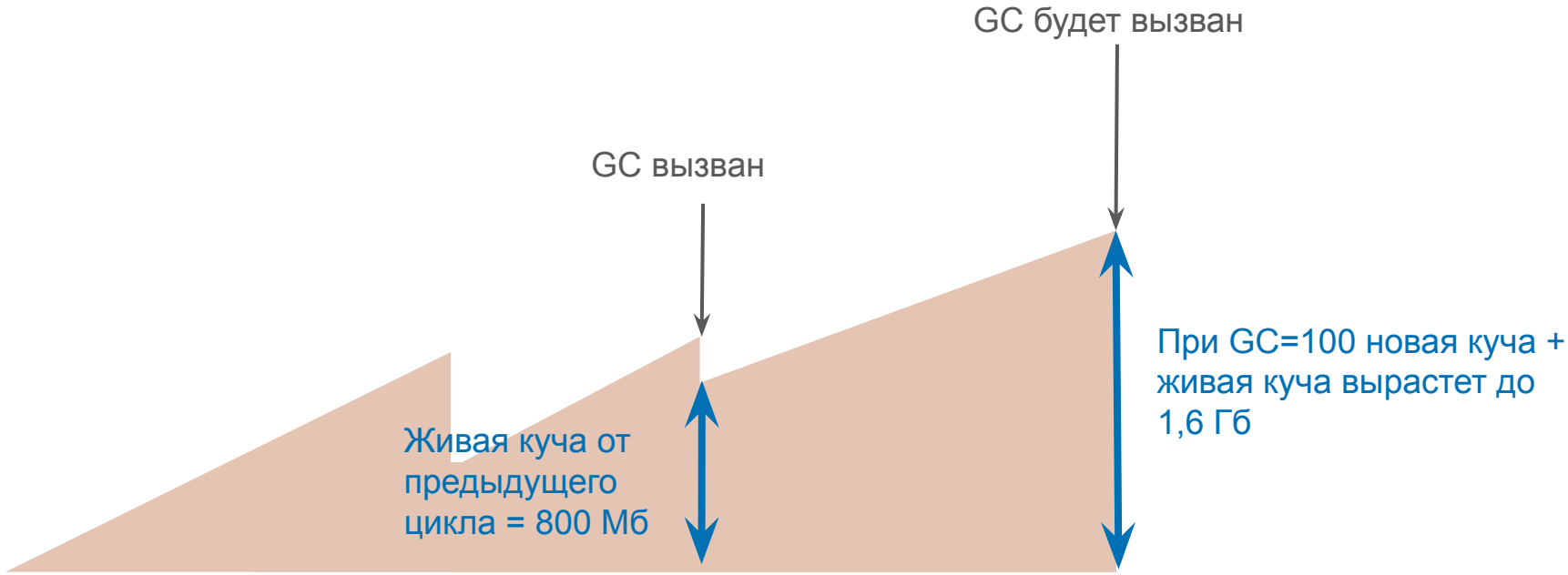

STATS (pid 1)
Goroutines:
Heap:
Threads:
PROCS (pid 0)
GC


PROCS (pid 0)
GC
13 items selected.
Slices (13)
Wall Duration
Self time
Average Wall Duration
Occurrence
1,915,968 ns
1,915,968 ns
1,915,968 ns
1

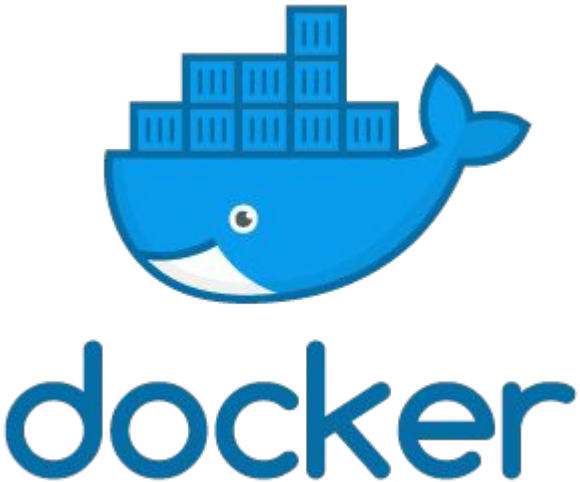
GOGC = off



Куча в реальной жизни



Запуск приложения в контейнере




```
deploy:
 resources:
   limits:
     memory: 1000M
```

Как исправить?

OOM

GOMEMLIMIT

Как оно работает?

GOMEMLIMIT задает **общее** количество памяти, которое Go runtime может использовать.

GOMEMLIMIT или debug.SetMemoryLimit

Когда общее значение памяти приближается к GOMEMLIMIT - запускается сборщик мусора

Почему не поставили по умолчанию?

А что будет при утечке данных?

GC GUIDE

Спираль смерти

По мере стремления "живой" памяти кучи к GOMEMLIMIT, GC будет запускаться все чаще и чаще.

Если не будет никаких ограничений, сборщик мусора Go в будет работать непрерывно.

Мягкое управление памятью

- Go не предоставляет 100% гарантий соблюдения ограничения памяти `GOMEMLIMIT`
- Предел использования процессорного времени: 50% с окном CPU в `2 * GOMAXPROCS` секунды

Но это значит, что на 100% OOM не избежать

Как применять GOGC и GOMEMLIMIT

- Приложение в контейнере: GOMEMLIMIT = 90-95% от доступной памяти
- Библиотека или код, требующего значительных ресурсов: динамическое управление debug.SetMemoryLimit
- Скрипт в контейнере: GOGC=off + GOMEMLIMIT

Где не применять GOGC и GOMEMLIMIT

- Приложение уже близко к предельному значению памяти своей среды
- Приложение в исполнительной среде, которой вы не управляете
- Использование памяти пропорционально входным данным: CLI-инструмент или настольное приложение

Что еще?

sync.Pool

- Повторное использование временных объектов
- Управляет объектами определенного типа
- Не гарантирует хранение данных
- Потокобезопасный

Arena

- Сам управляет памятью без использования GC, освобождается целиком
- Аллокация большого куска памяти
- Уменьшает количество вызовов GC (экономия CPU до 15%)
- Несколько Arena на приложение с различным временем жизни
- Не потокобезопасны

Ссылки

Предложение о мягком управлении памятью

Предложение о Arena

Профиль на habr

Профиль на medium.com

Галерея неудачных генераций