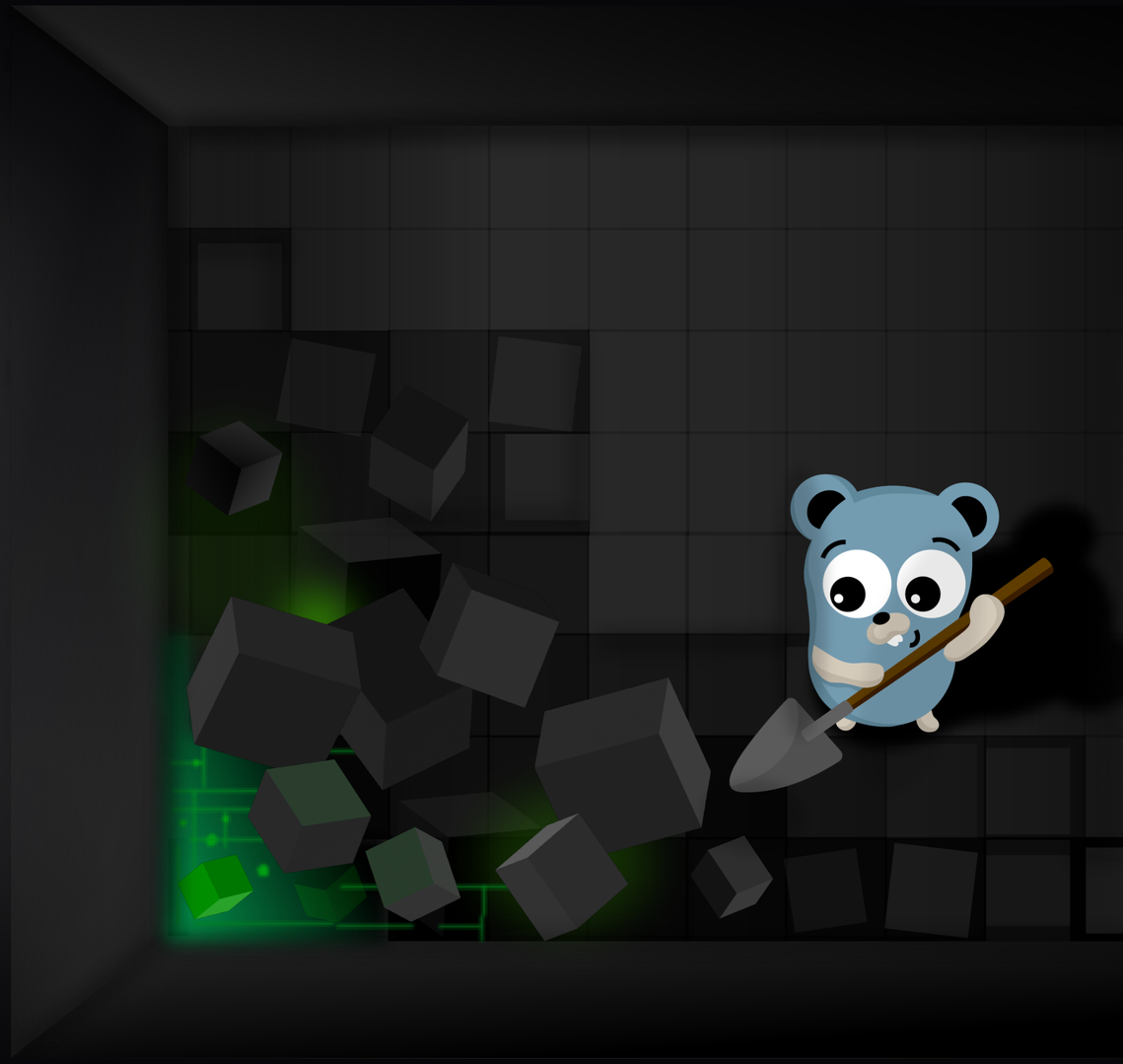
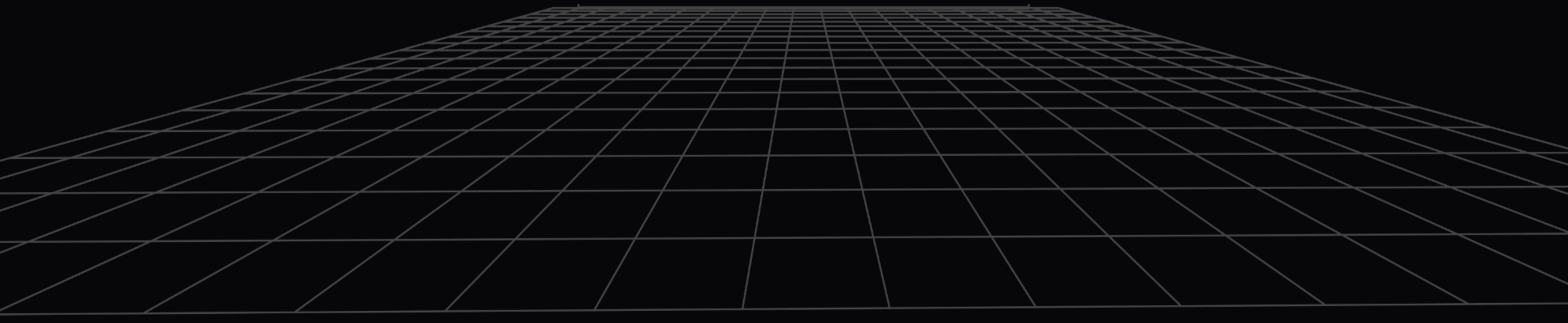
Глубокий Go

СБОРЩИК МУСОРА



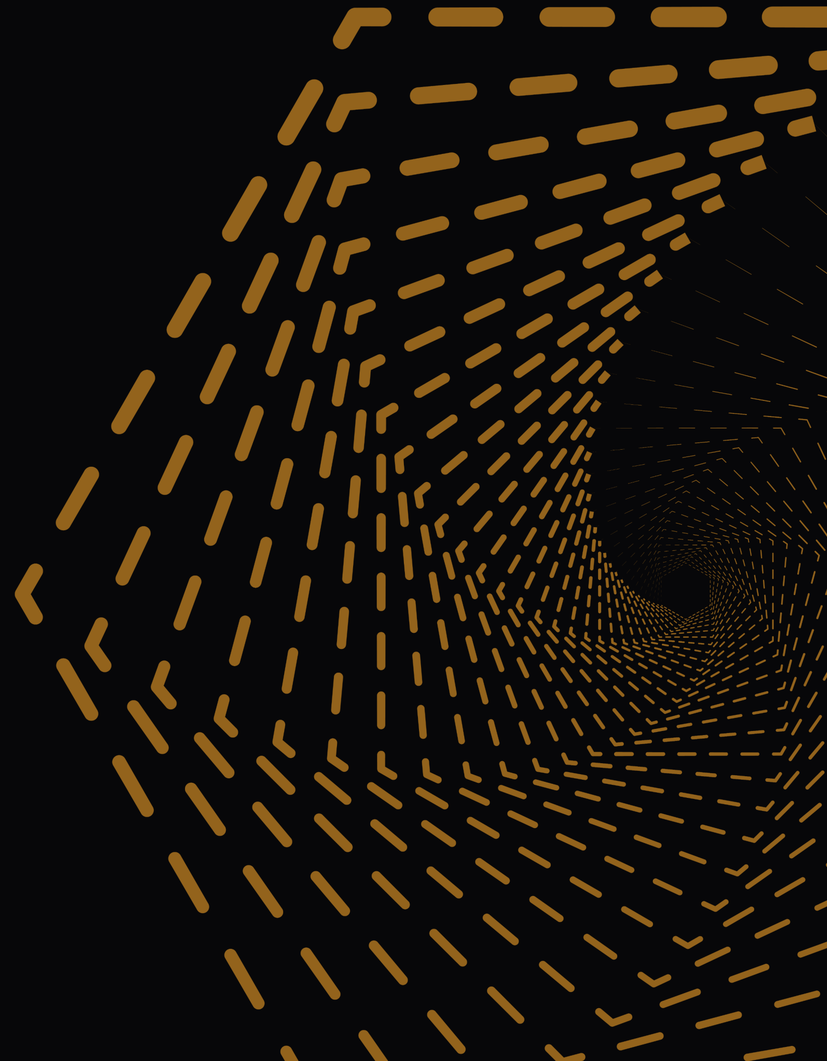
Balun.Courses

ПРОВЕРЬ ЗАПИСЬ



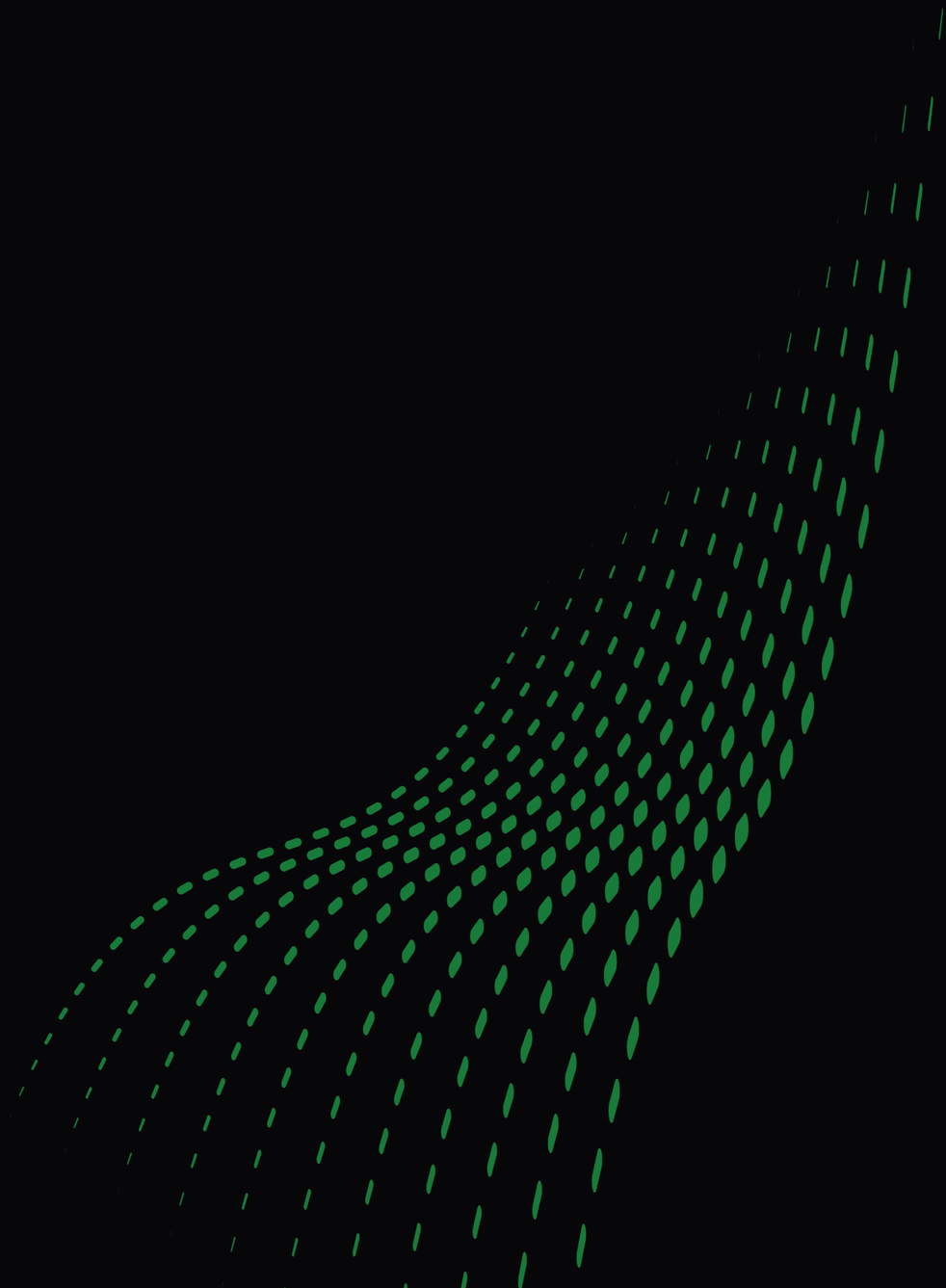
ПРАВИЛА ЗАНЯТИЯ

1. вопросы в чате можно задавать **в любое время**
2. вопросы голосом задаем **по поднятой руке в Zoom**
3. ответы на вопросы будут **в запланированных местах**



ПЛАН ЛЕКЦИИ

1. Сборка мусора
2. Подсчет ссылок
3. Трейсинг
4. Устройство GC в Go
5. Finalizers



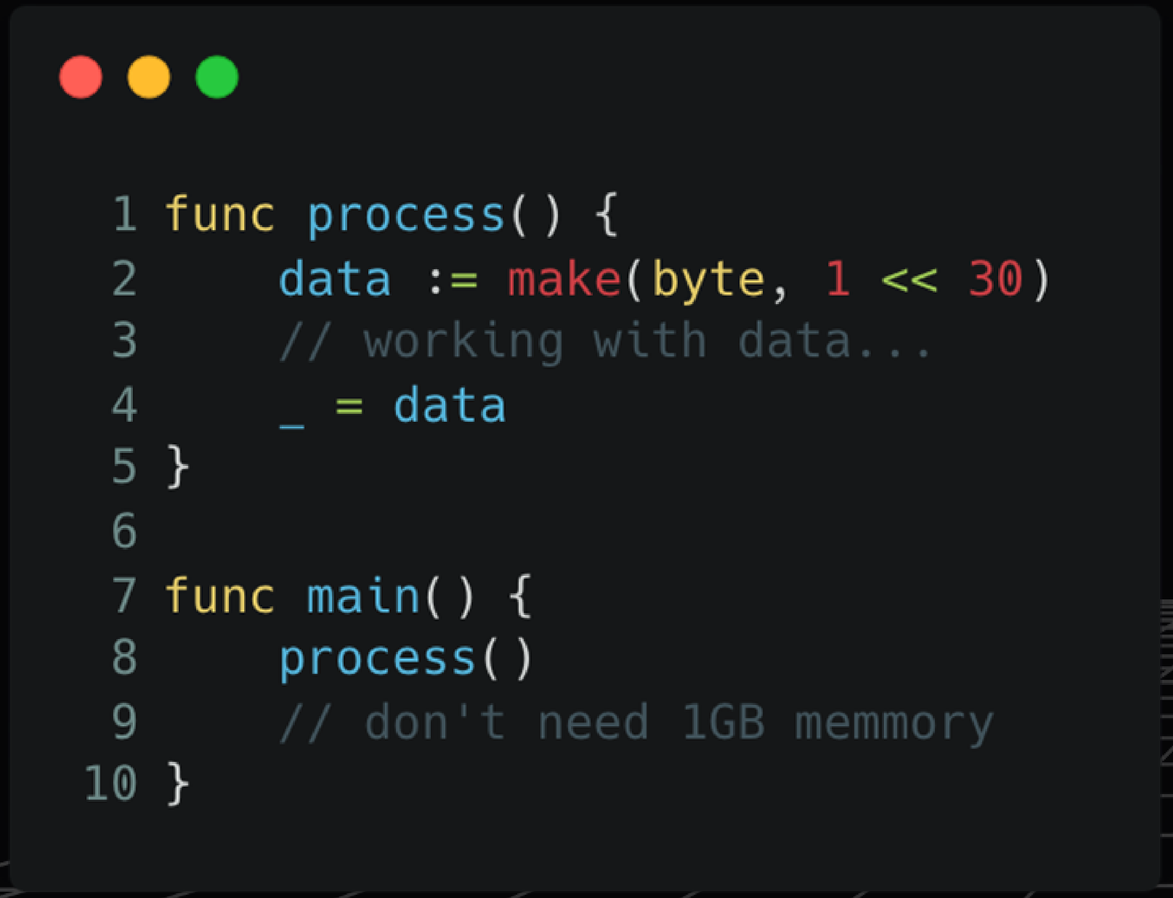
СБОРКА МУСОРА

Terminal: question

ЧТО ТАКОЕ МУСОР?

МУСОР — ЭТО

Неиспользуемая память программы,
которую следует освободить
для дальнейшего использования



```go
func process() {
    data := make(byte, 1 << 30)
    // working with data...
    _ = data
}

func main() {
    process()
    // don't need 1GB memmory
}
```

Terminal: question + ⌄ ⚙ –

ВСЕГДА ЛИ НУЖНО СОБИРАТЬ МУСОР?

ИНОГДА МОЖНО
НЕ ДУМАТЬ О СБОРКЕ МУСОРА

Например, если ваше приложение работает несколько минут или часов и ему хватает памяти для его работы, то мусор можно не собирать

РУЧНОЙ РЕЖИМ

Мусор можно собирать руками, когда вы самостоятельно освобождаете те объекты, которые вам не нужны

Terminal: question + ⌄ ⚙ –

МОЖНО ЛИ СОБИРАТЬ МУСОР АВТОМАТИЧЕСКИ?

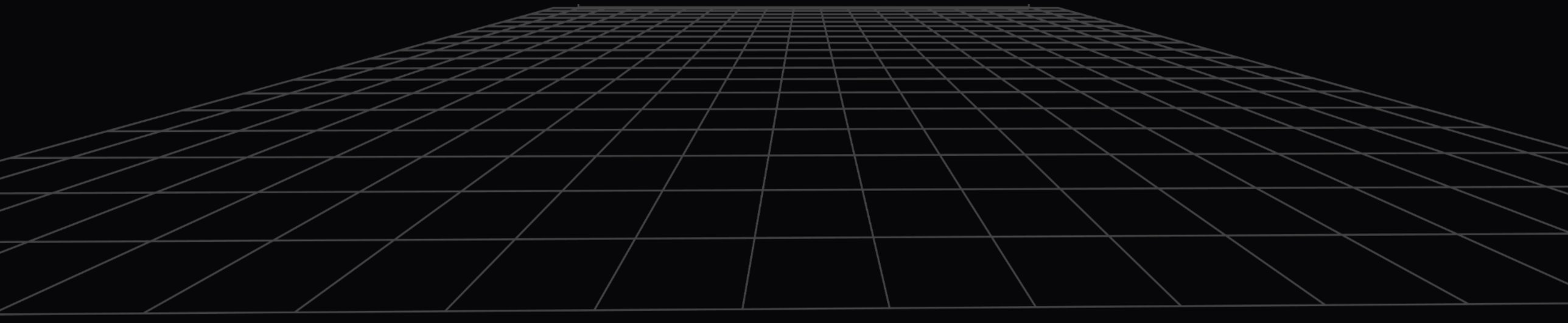
АВТОМАТИЧЕСКИЙ РЕЖИМ

Подсчет ссылок

Трассирование

FAQ

Сборка мусора



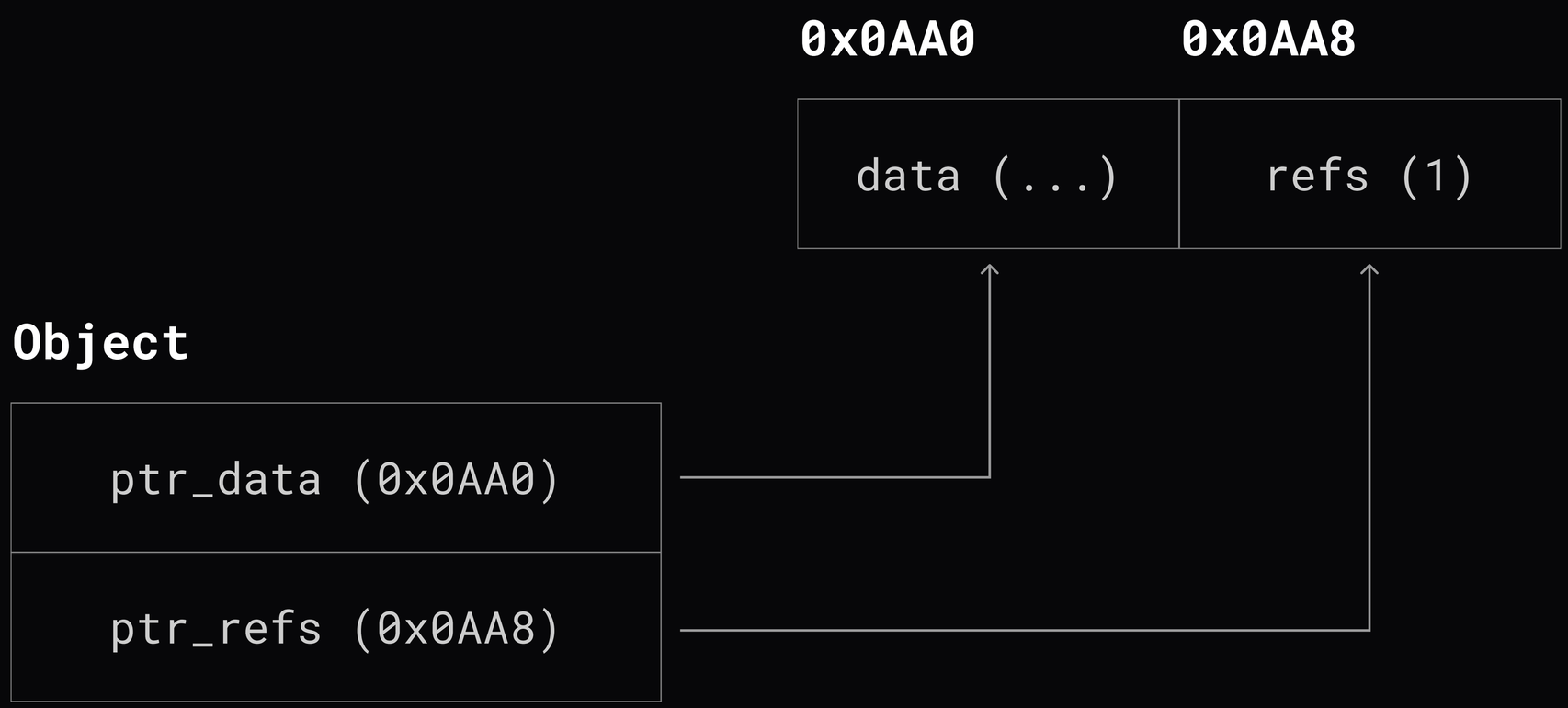
ПОДСЧЕТ ССЫЛОК

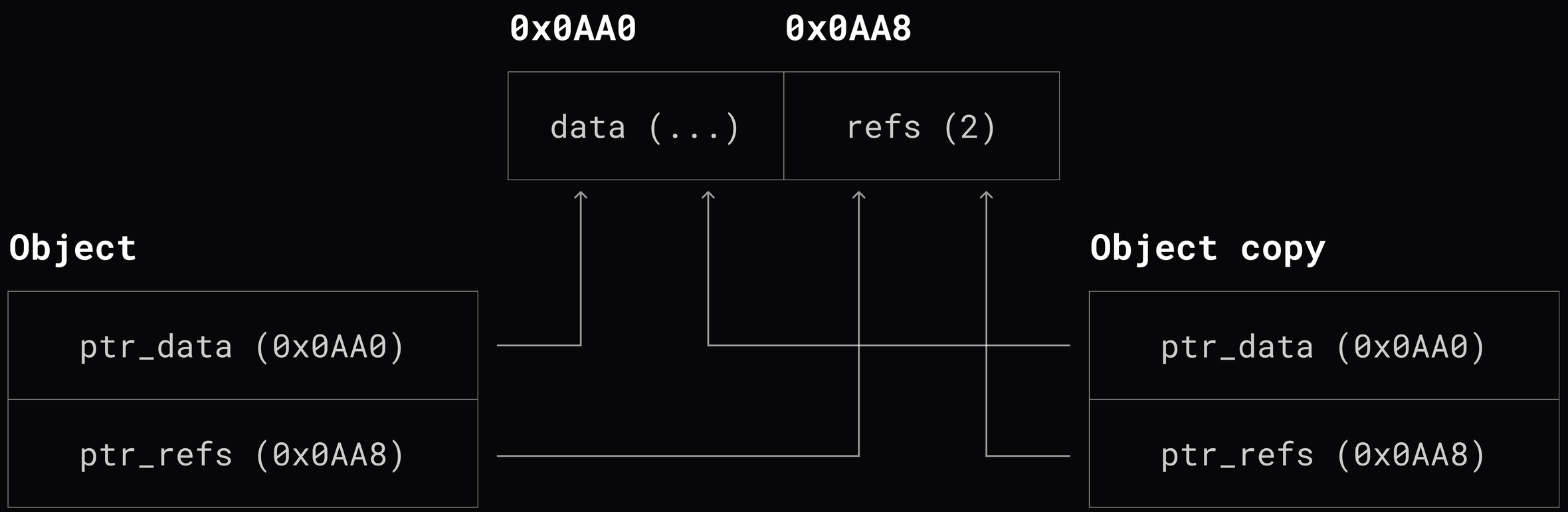
Terminal: deep_go

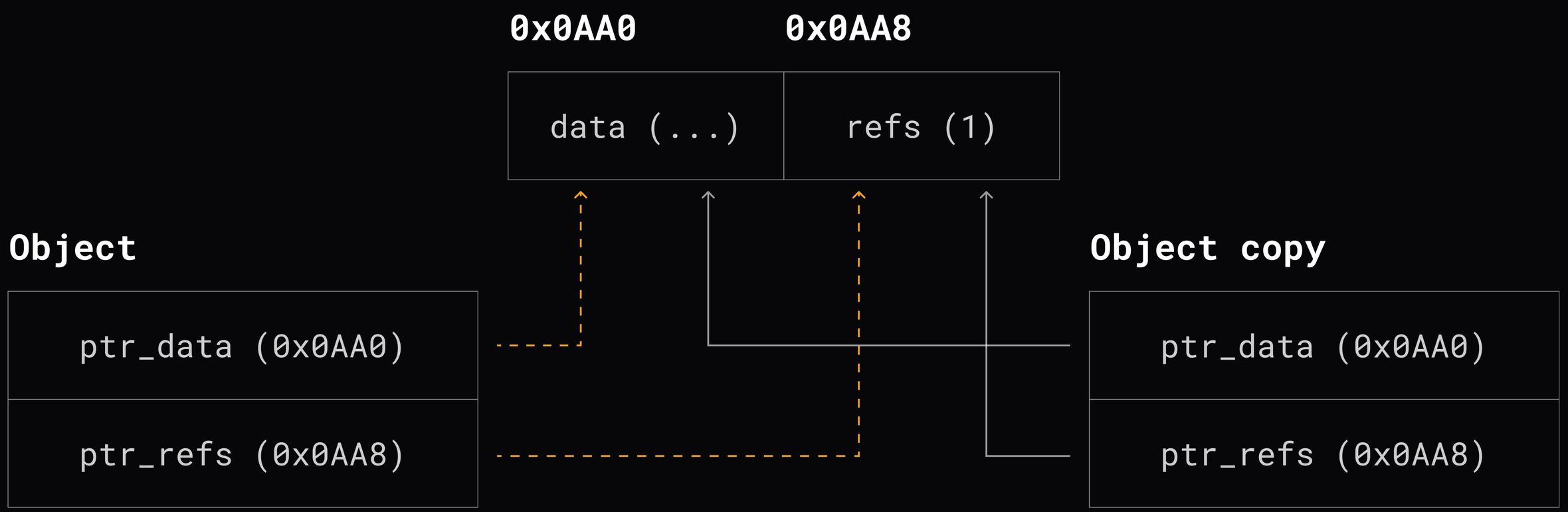
ПОДСЧЕТ ССЫЛОК

Каждый объект содержит счётчик количества ссылок на него, используемых другими объектами

> i Когда этот счётчик уменьшается до нуля, это означает, что объект стал недоступным, и его можно освобождать


0x0AA0
0x0AA8
data (...)
refs (1)
Object
ptr_data (0x0AA0)
ptr_refs (0x0AA8)


0x0AA0
0x0AA8
data (...)
refs (2)
Object
ptr_data (0x0AA0)
ptr_refs (0x0AA8)
Object copy
ptr_data (0x0AA0)
ptr_refs (0x0AA8)


0x0AA0
0x0AA8
data (...)
refs (1)
Object
ptr_data (0x0AA0)
ptr_refs (0x0AA8)
Object copy
ptr_data (0x0AA0)
ptr_refs (0x0AA8)


0x0AA0
0x0AA8
data (...)
refs (0)
Object
ptr_data (0x0AA0)
ptr_refs (0x0AA8)
Object copy
ptr_data (0x0AA0)
ptr_refs (0x0AA8)

Terminal: question + ⌄ ⚙ –

КАКИЕ ПРЕИМУЩЕСТВА И НЕДОСТАТКИ У ТАКОГО ПОДХОДА?

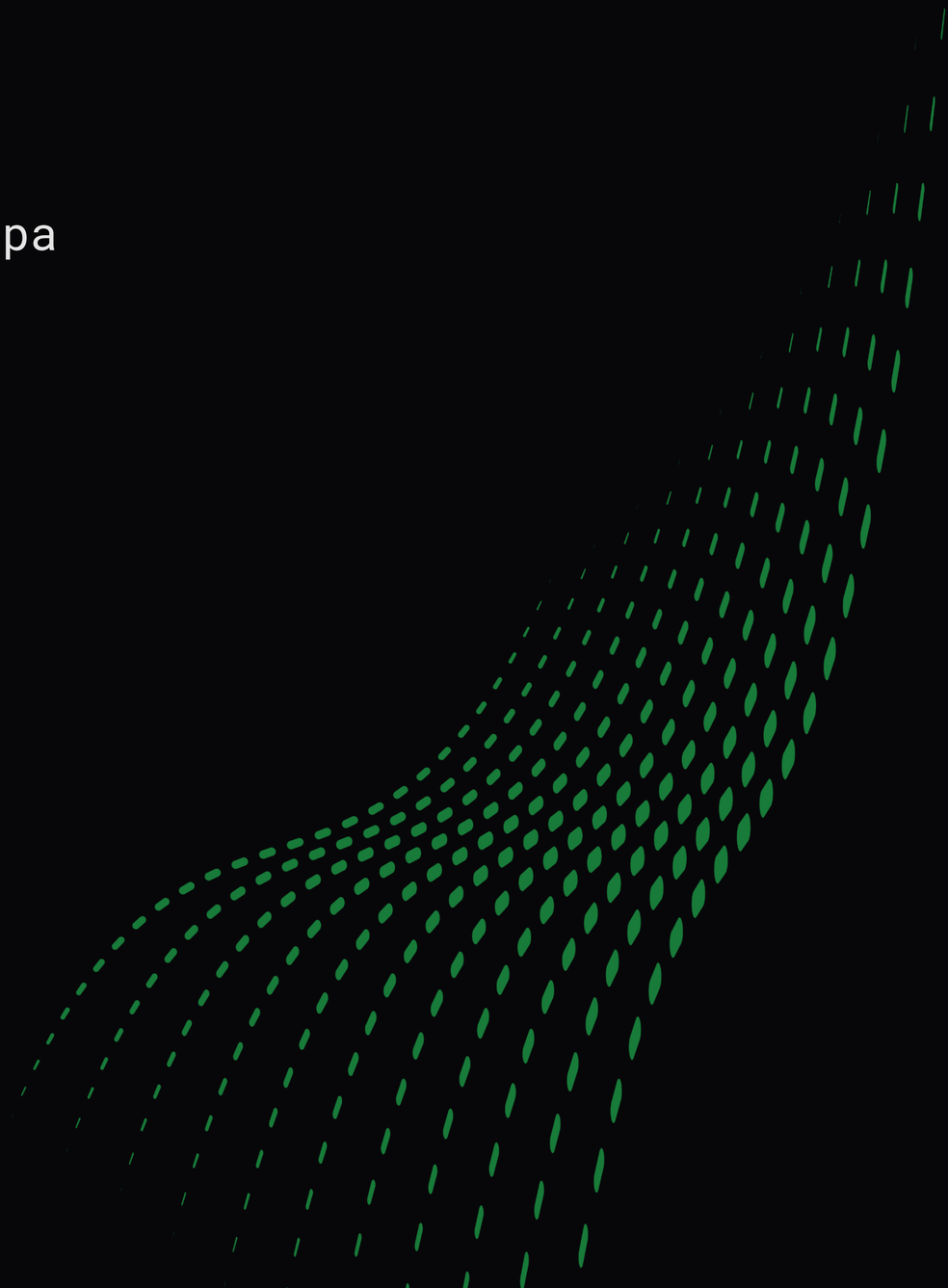
ПОДСЧЕТ ССЫЛОК

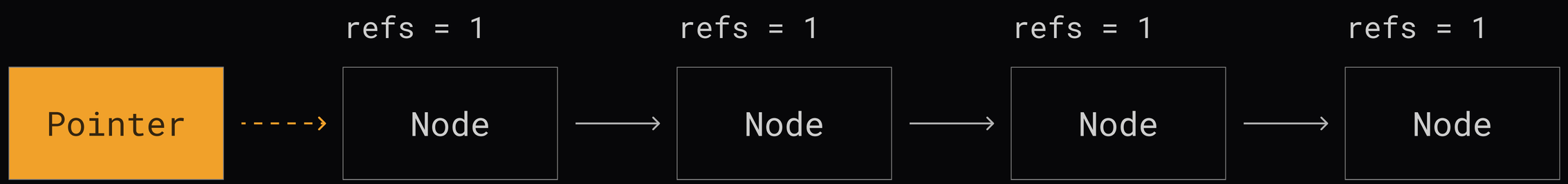
+ не нужно запускать никаких процессов сборки мусора

- делаем лишнюю работу по увеличению и уменьшению счетчиков ссылок

- нужна синхронизация работы со счетчиком ссылок

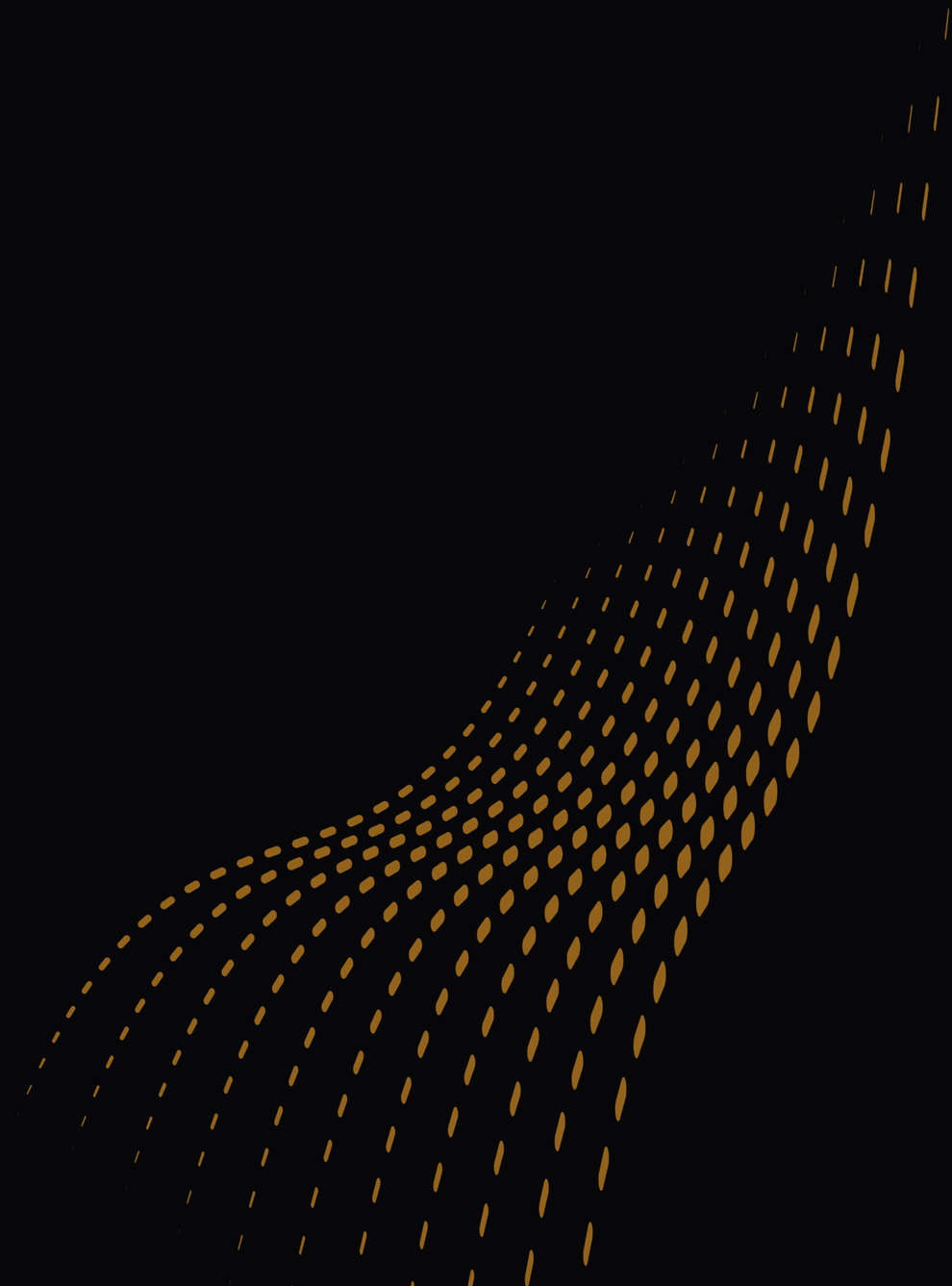
- дополнительная память на хранение счетчика

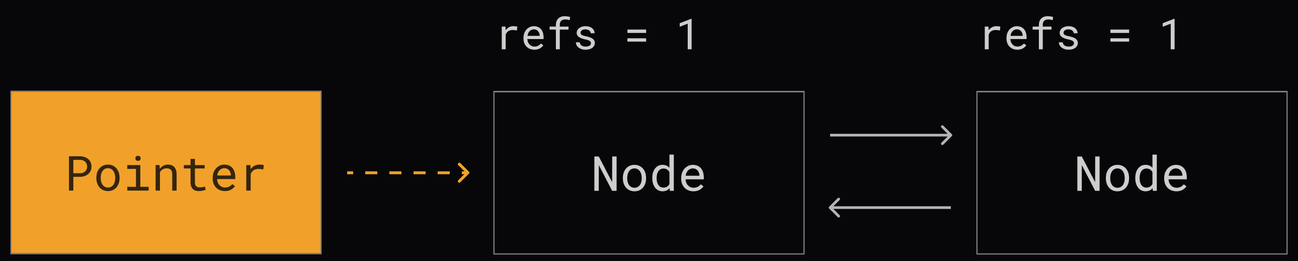


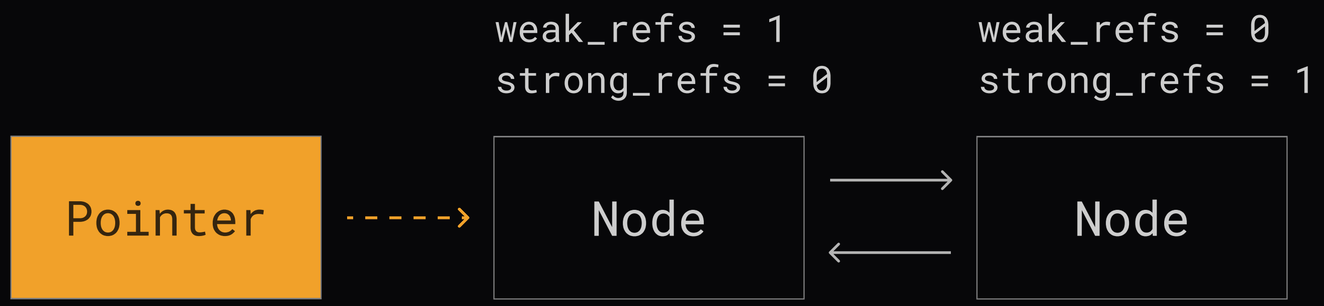

refs = 1
refs = 1
refs = 1
refs = 1
Pointer
Node
Node
Node
Node

ПОДСЧЕТ ССЫЛОК

- синхронное освобождение последовательности элементов, например множества элементов связного списка

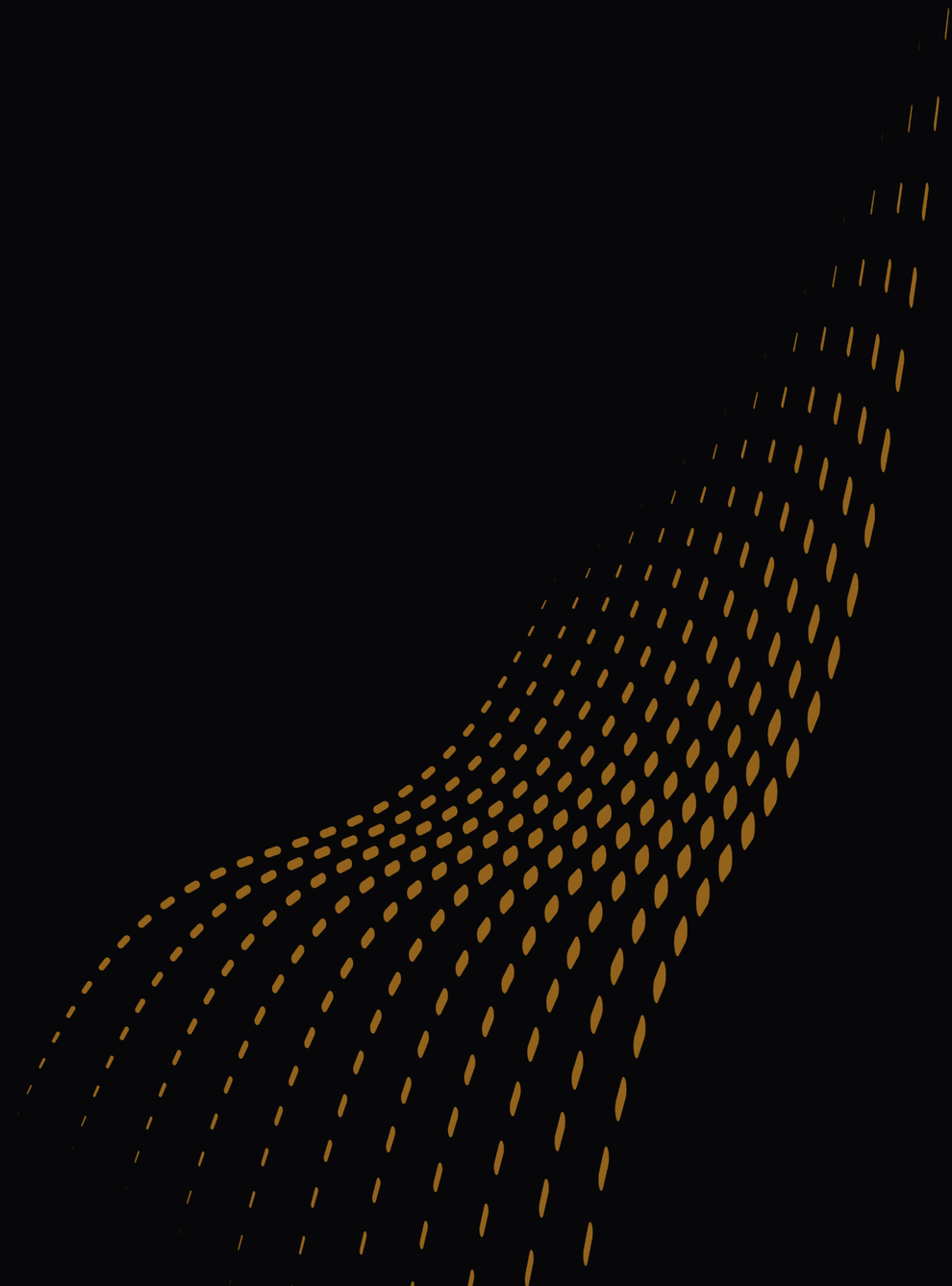



refs = 1
refs = 1
Pointer
Node
Node


weak_refs = 1
strong_refs = 0
weak_refs = 0
strong_refs = 1
Pointer
Node
Node

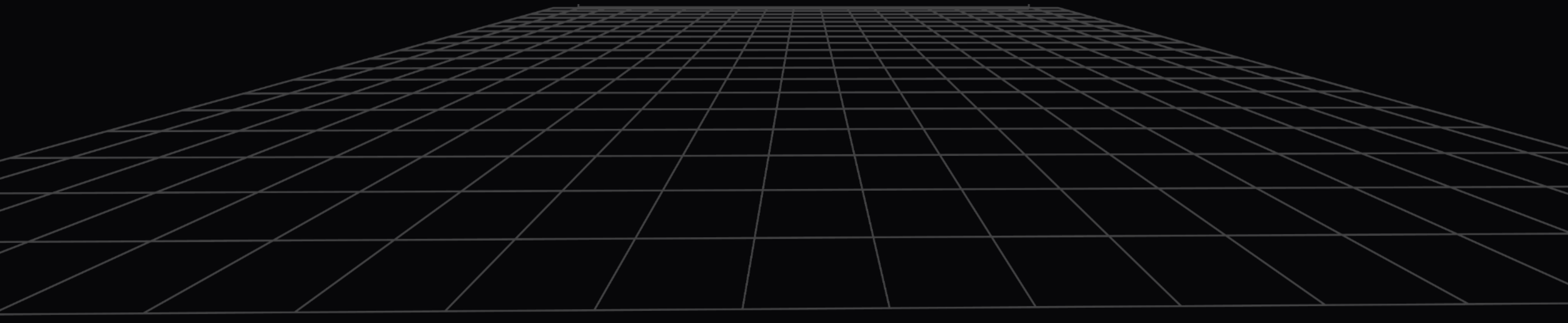
ПОДСЧЕТ ССЫЛОК

- проблемы циклических ссылок (абстрагировать их не получится, программисту придется знать о них и ими управлять)



FAQ

Подсчет ссылок



ТРЕЙСИНГ

Terminal: deep_go

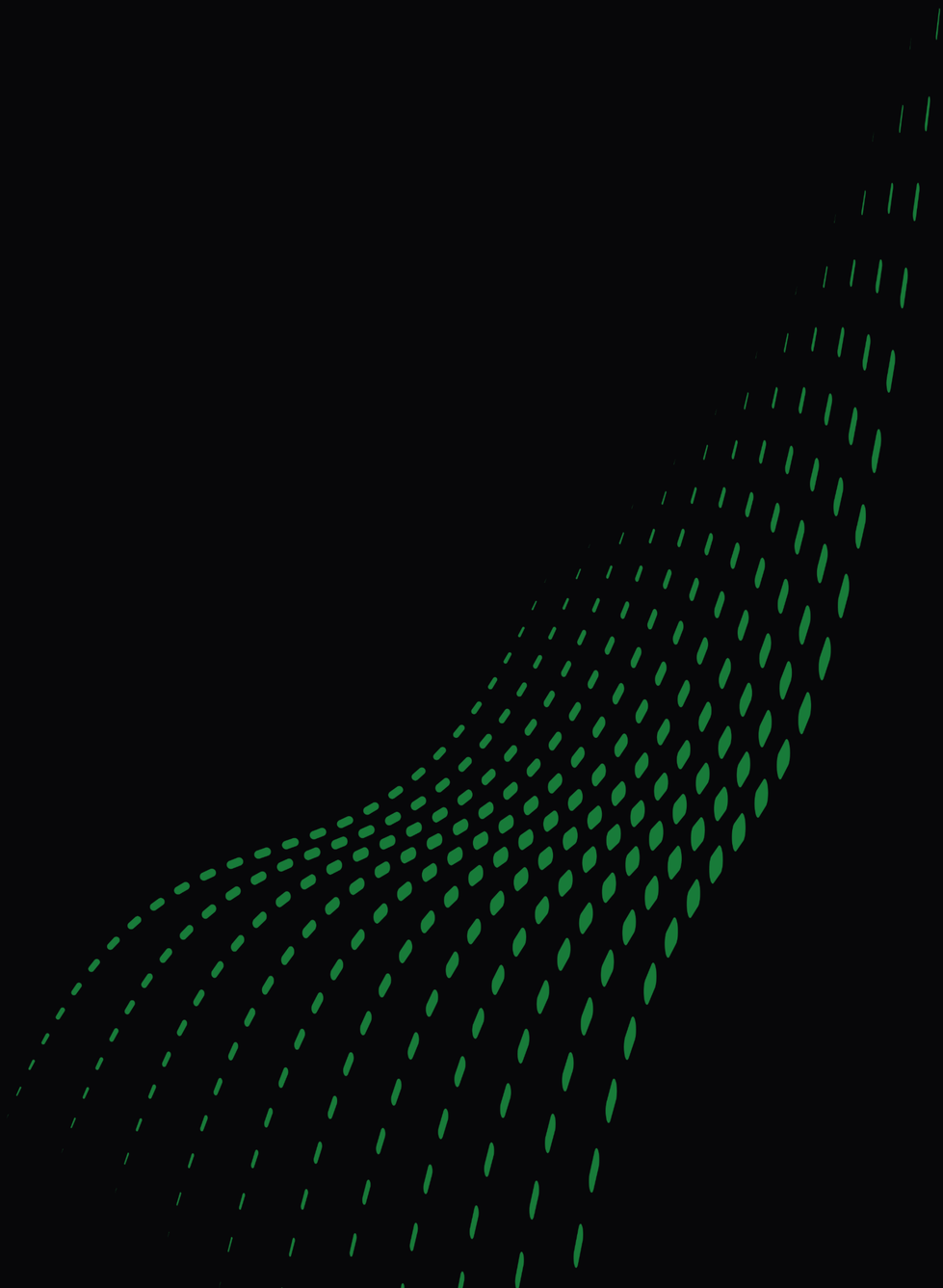
ТРЕЙСИНГ

В отличии от подсчета ссылок, трейсинг занимается противоположным – он ищет живые объекты.

Останавливаются все потоки, без синхронизации обходится граф ссылок, помечаются все используемые объекты, после чего непомеченные объекты удаляются

САМЫЙ ПРИМИТИВНЫЙ СЦЕНАРИЙ РАБОТЫ

1. Останавливаются все потоки (мутаторы)
2. Без синхронизации обходится граф ссылок (можно параллельно)
3. Помечаются все используемые объекты
4. После чего непомеченные объекты удаляются



№1

Остановить все потоки

Terminal: question + ⌄ ⚙ –

КАК ОСТАНОВИТЬ ВСЕ ПОТОКИ?

Можно вставить точки в код, благодаря которым поток может понять, что нужно остановиться, запретить страницу для чтения при помощи **mprotect**

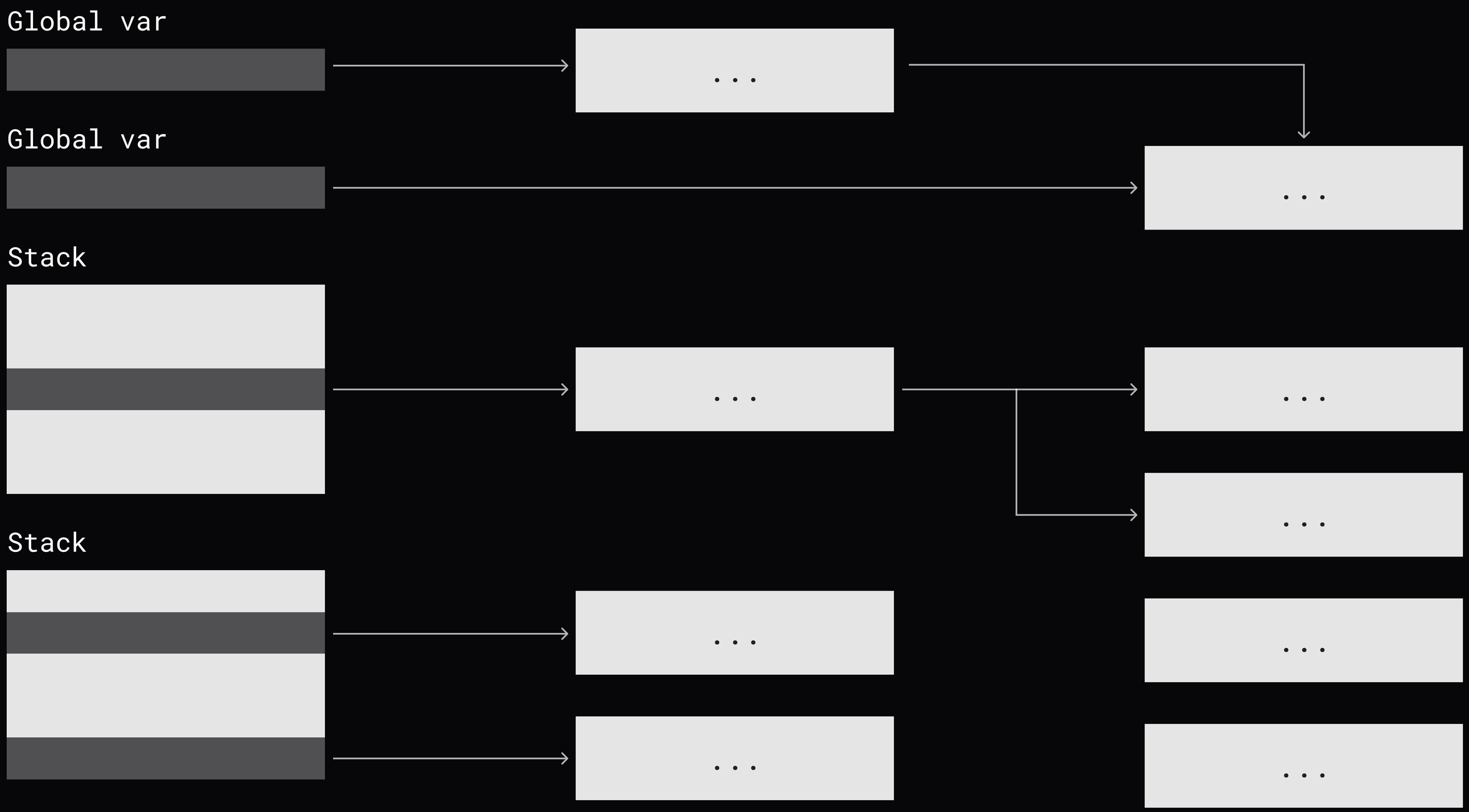
№2

Обойти граф ссылок - фаза mark

Terminal: question + ⌄ ⚙ –

ОТКУДА НАЧНЕМ ОБХОД ГРАФА?

Начнем обход графа со стеков потоков и с глобальных переменных, формируя root set для обхода


Global var
Global var
Stack
Stack

Terminal: question + ⌄ ⚙ –

КАК БУДЕМ ОБХОДИТЬ ГРАФ?

Делать обход графа можно поиском в глубину или ширину,
но какой тогда выбрать вариант?

ПРИ ОБХОДЕ КРАСИМ ГРАФ В ТРИ ЦВЕТА

1. **Черный** – исследованный объект (объект посещен и все ссылки которые идут из него исследованы)

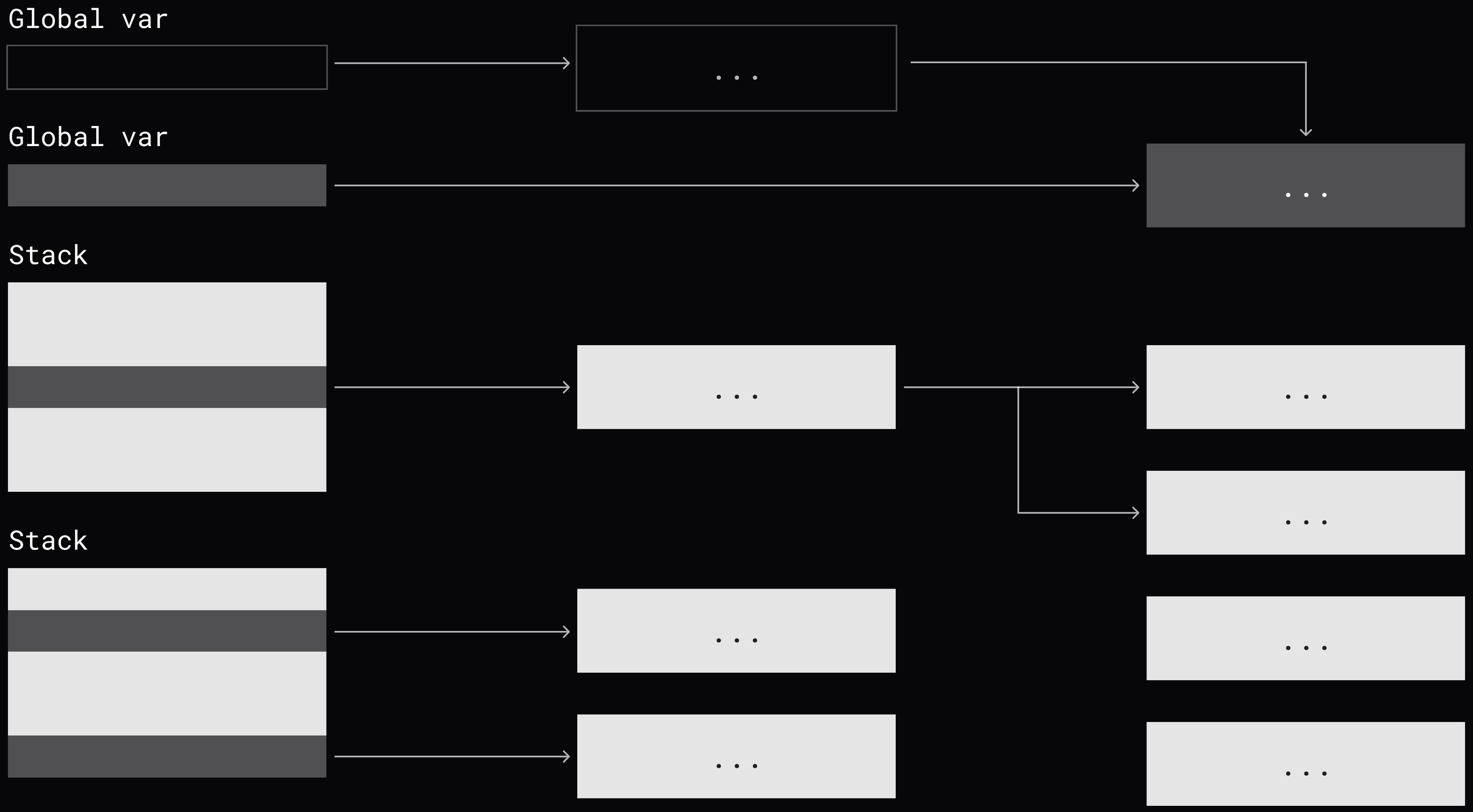
2. **Серый** – ожидающий исследования объект (узнали про объект, но его еще не исследовали, не все ссылкам от него прошлись)

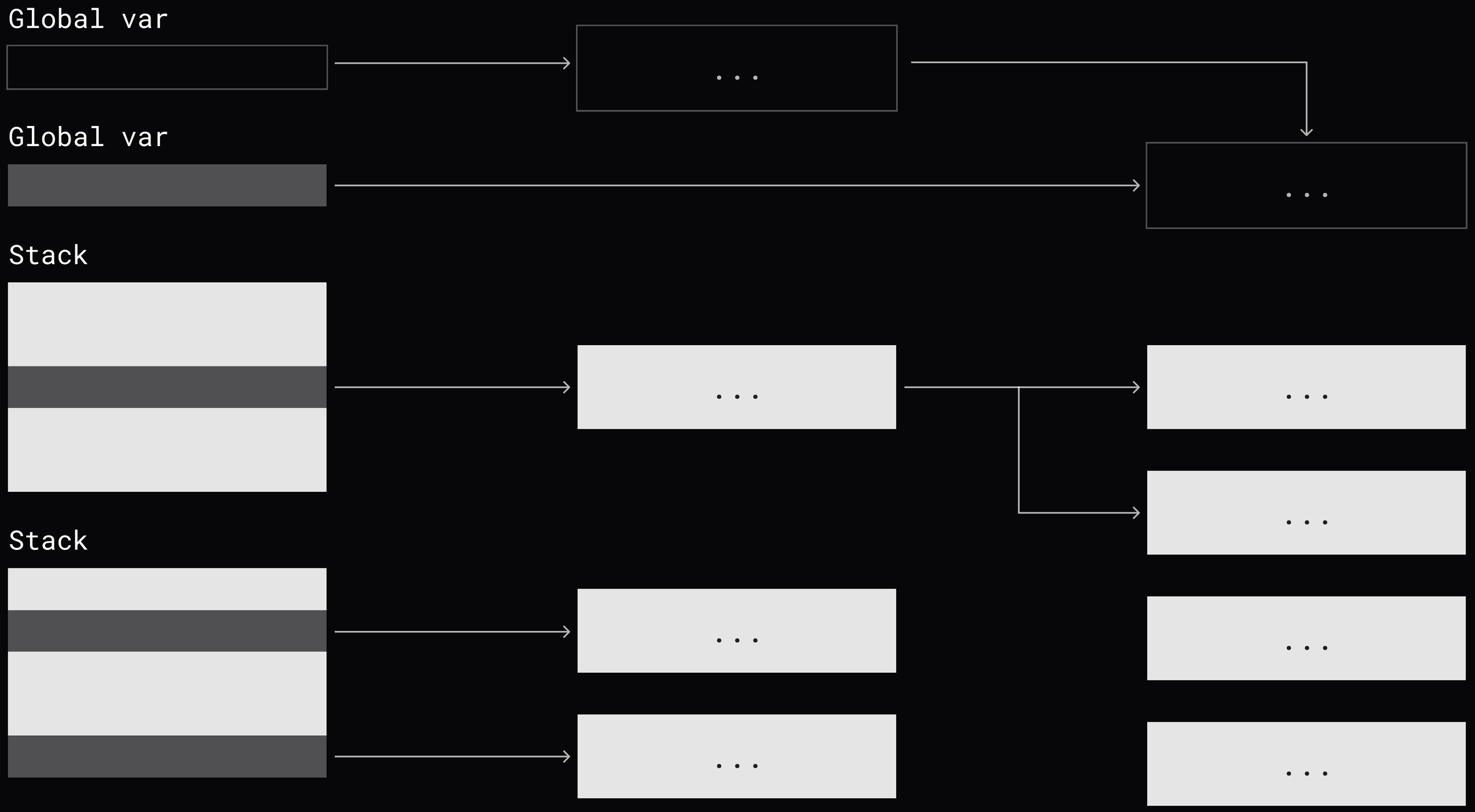
3. **Белый** – неисследованный объект (не знаем еще про объект)

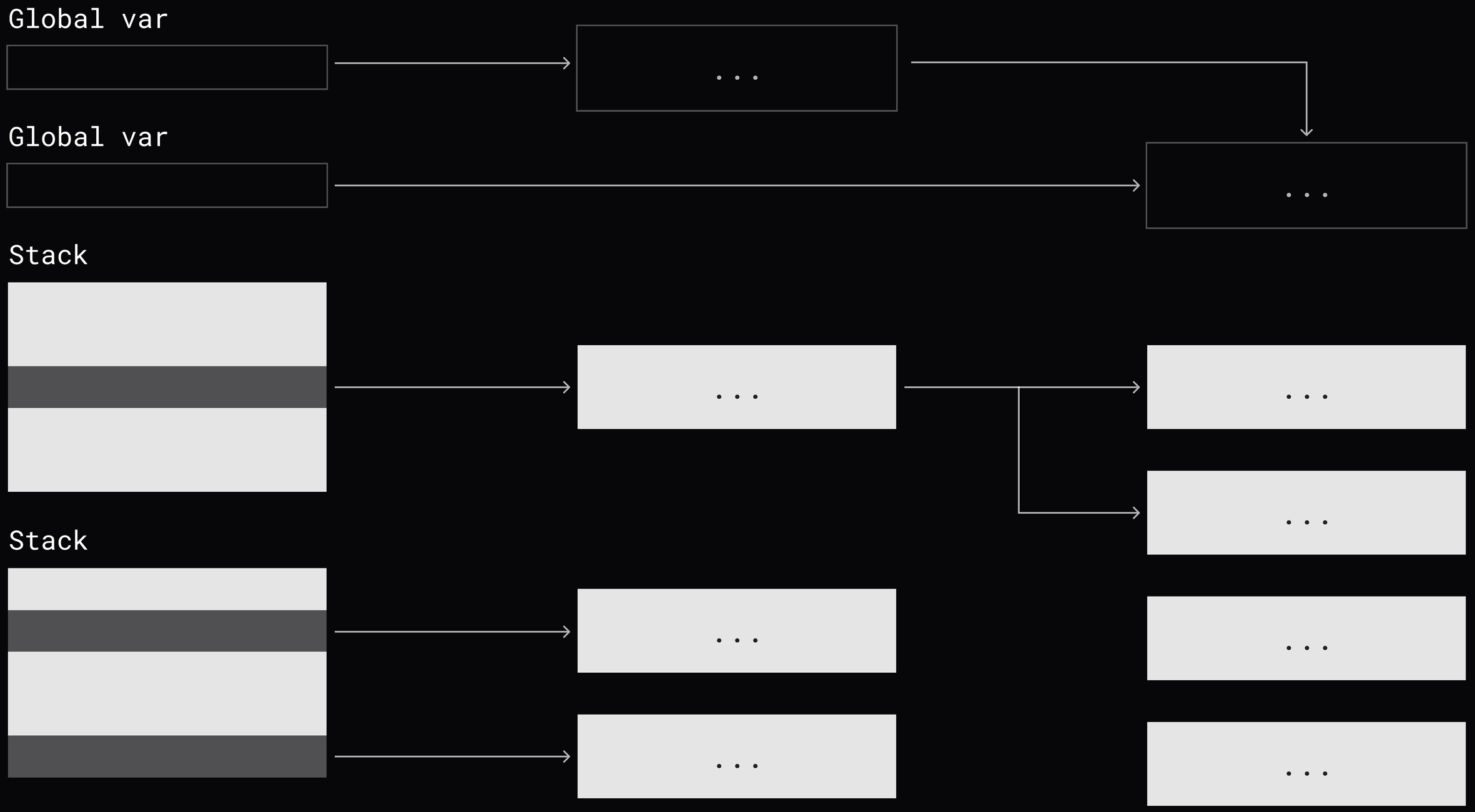
> **Tree color invariant** – не бывает ссылок из черного объекта в белый

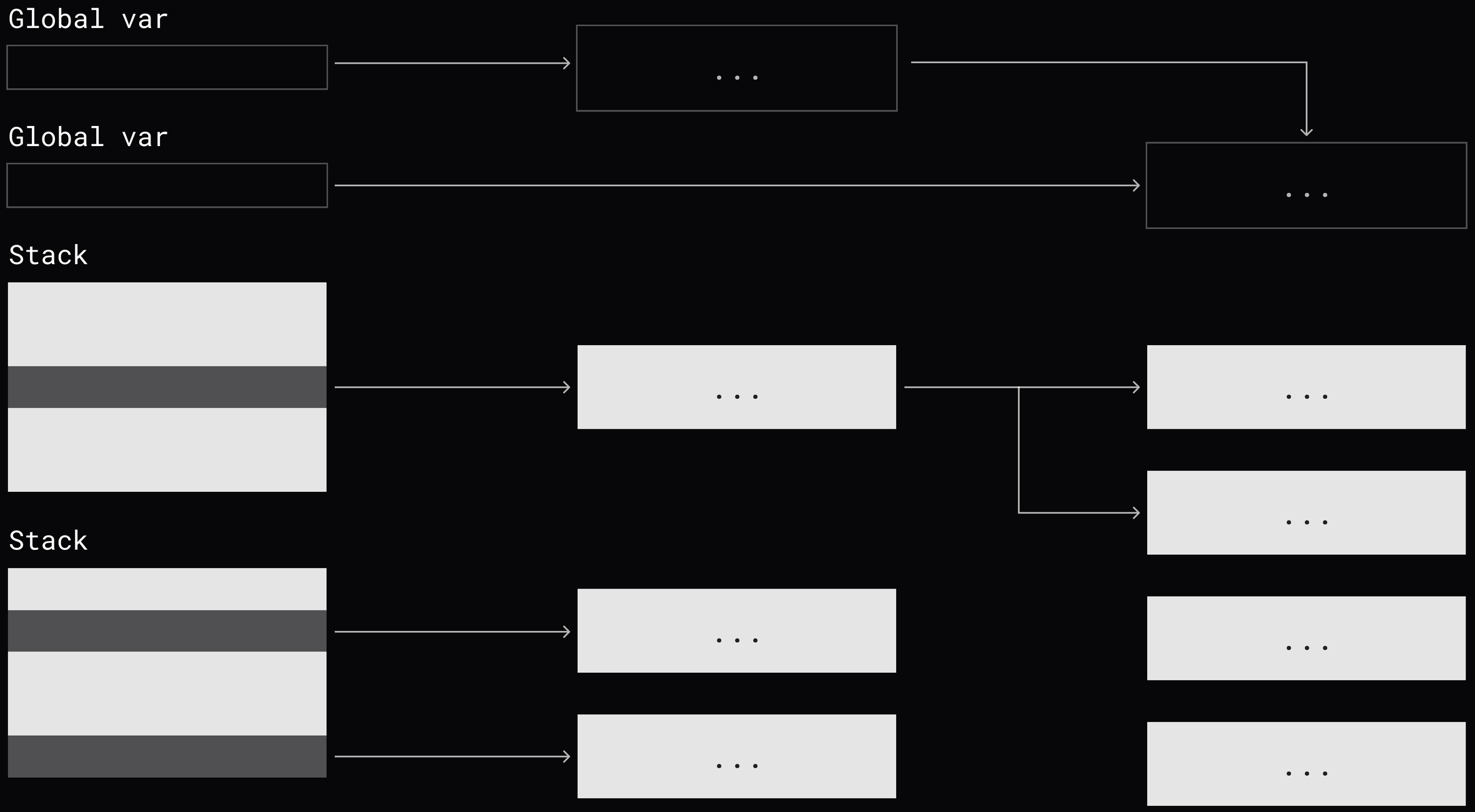

Global var
Global var
Stack
Stack

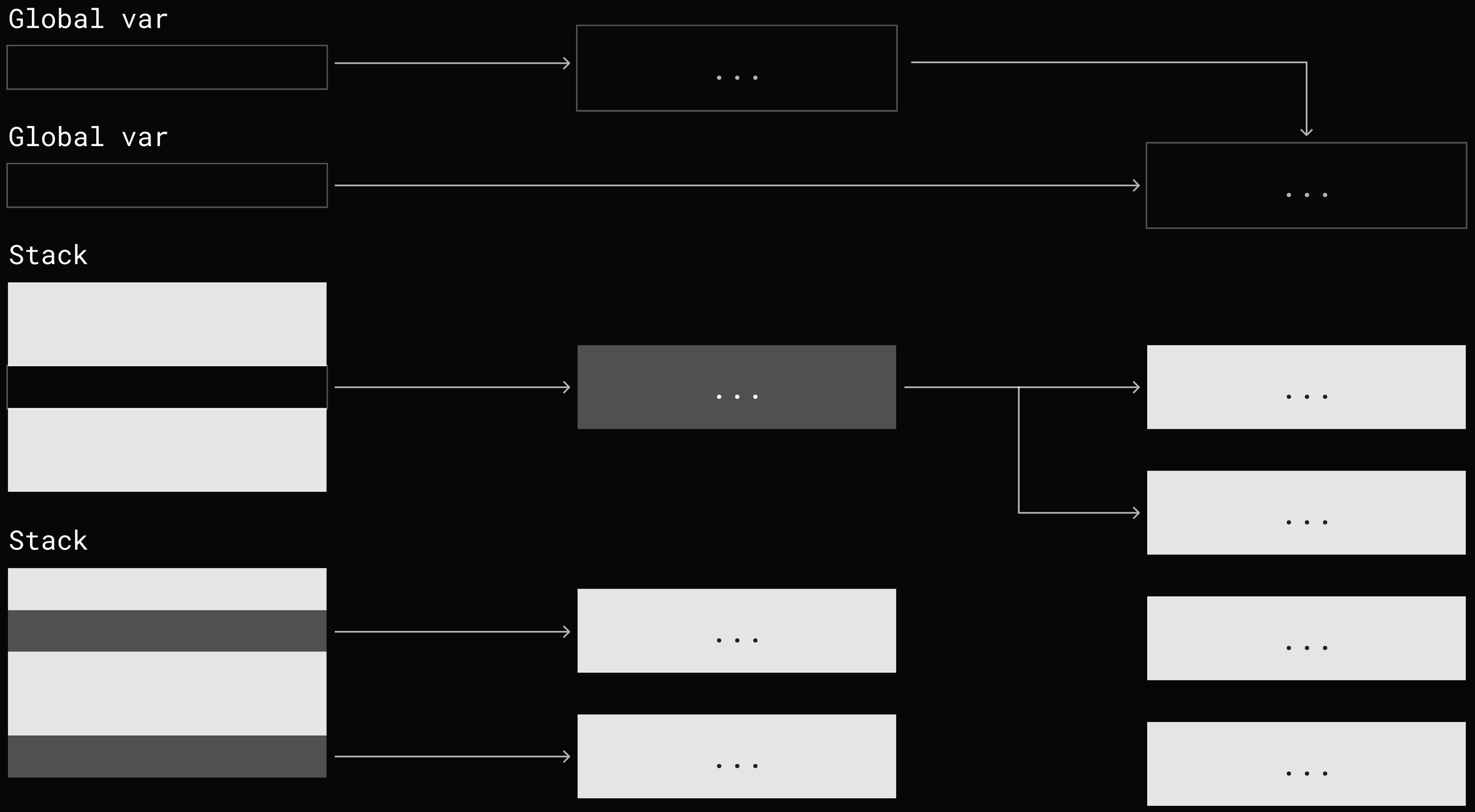

Global var
...
Global var
...
Stack
...
...
...
Stack
...
...
...
...

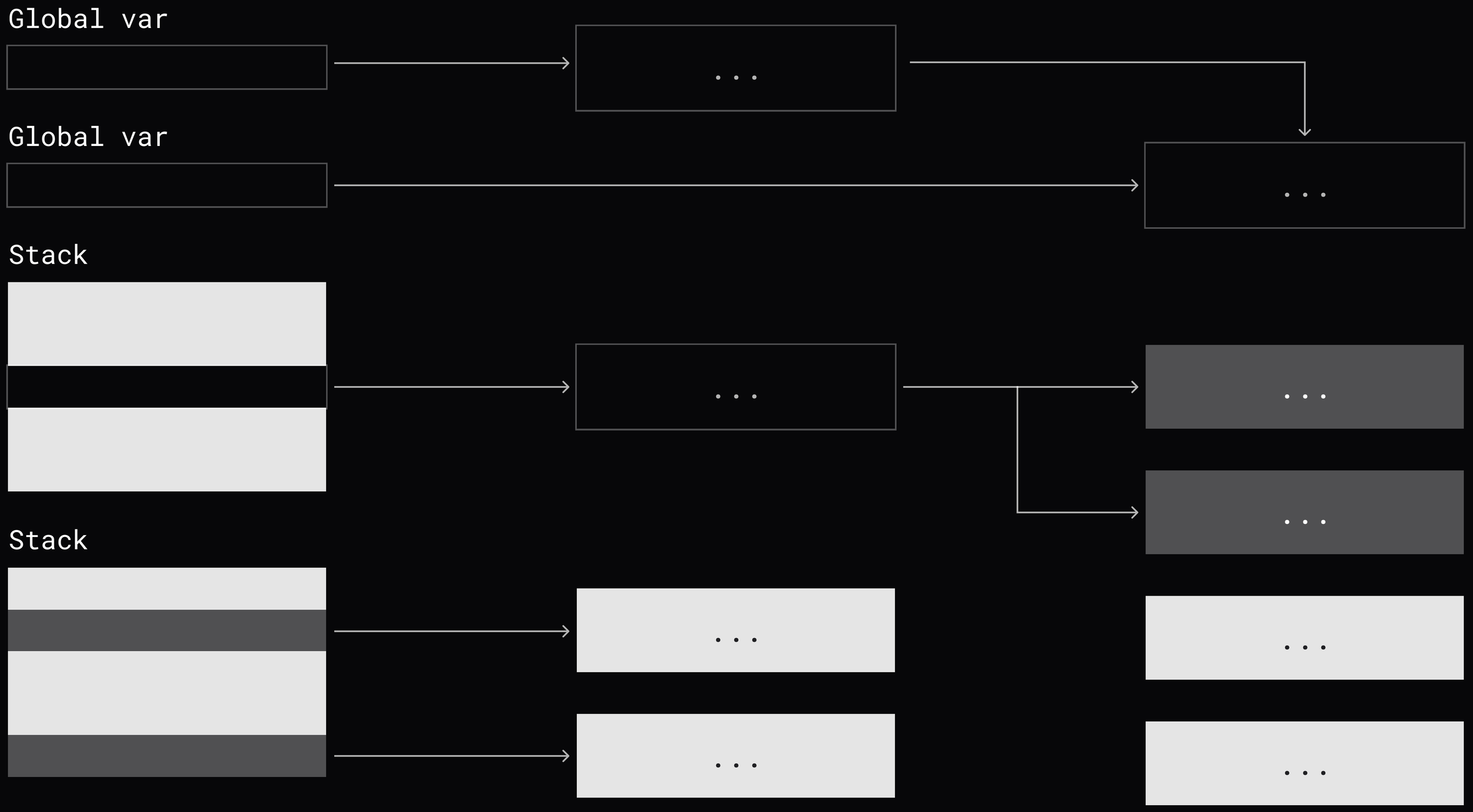

Global var
...
Global var
...
Stack
...
...
...
Stack
...
...
...
...


Global var
...
Global var
...
Stack
...
...
...
Stack
...
...
...
...


Global var
Global var
Stack
Stack


Global var
Global var
Stack
Stack


Global var
...
Global var
...
Stack
...
...
...
Stack
...
...
...
...


Global var
Global var
Stack
Stack

ПО ИТОГУ ГРАФ ПРИОБРЕТЕТ ТАКОЙ ВИД:

Terminal: question + ⌄ ⚙ –

ЗАЧЕМ НУЖНО КРАСИТЬ ГРАФ В ТРИ ЦВЕТА?

Использование раскраски вершин графа в три цвета – типичный способ поиска циклов в графе

№3

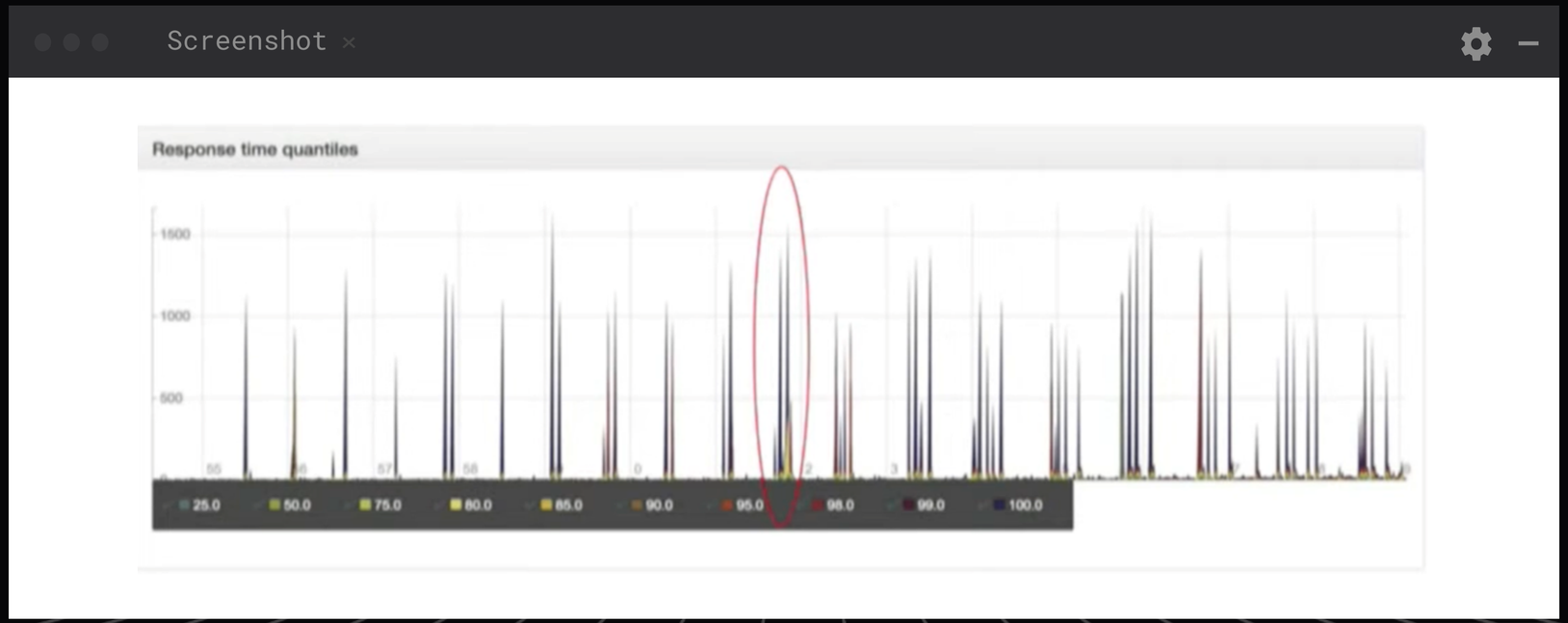
Дальше есть две альтернативы:

1. **Sweep** (подмести) — освобождаем то, что белого цвета
2. **Copying** (перенести) — переносим живые объекты в новое место,
   а старое освобождаем, чтобы не было фрагментации

Так работает простой **STW** (Stop-The-World) сборщик мусора

— чем больше куча, тем больше паузы!

КАК УМЕНЬШИТЬ ПАУЗУ?

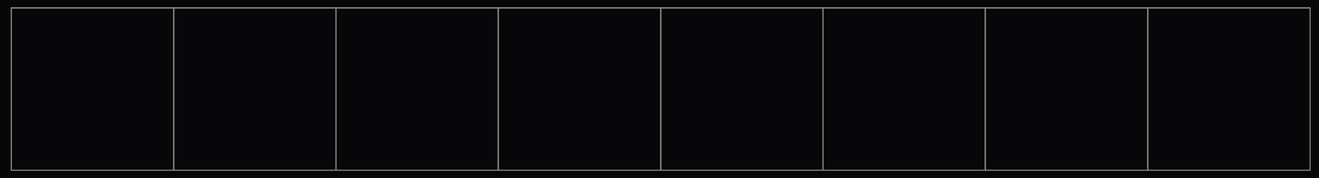


ПЕРЕРЫВ 5 МИНУТ

Можно обходить не всю кучу, а часть (поколения), так как большинство объектов удаляются «молодыми». Хорошо бы отделить объекты по временим жизни – те, которые живут долго, будем их обходить редко, а молодые будем обходить чаще

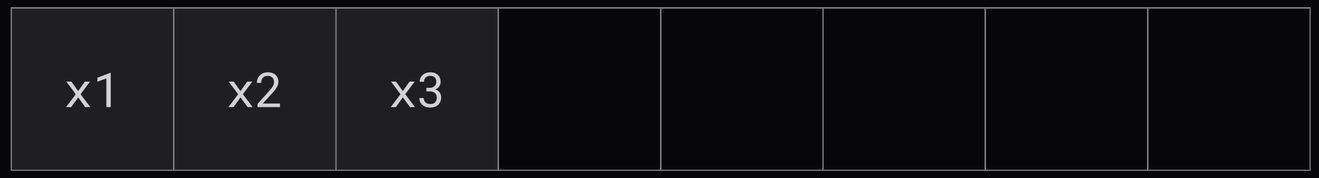
#1 generation

#2 generation



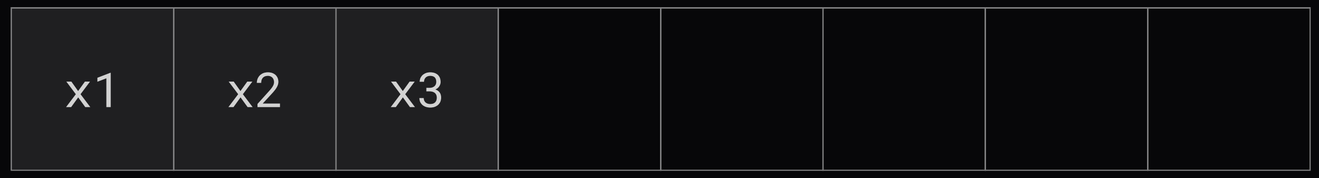
#1 generation

#2 generation



#1 generation

#2 generation



#1 generation

#2 generation

Terminal: question + ⌄ ⚙ –

ПОДХОД С ПОКОЛЕНИЯМИ МОЖЕТ МИНИМИЗИРОВАТЬ ПАУЗЫ, НО КАК ОТ НИХ ИЗБАВИТЬСЯ СОВСЕМ?

Terminal: deep_go

CONCURRENT GC

Можно запустить поток GC одновременно, который выполняет сбор мусора вместе с мутаторами

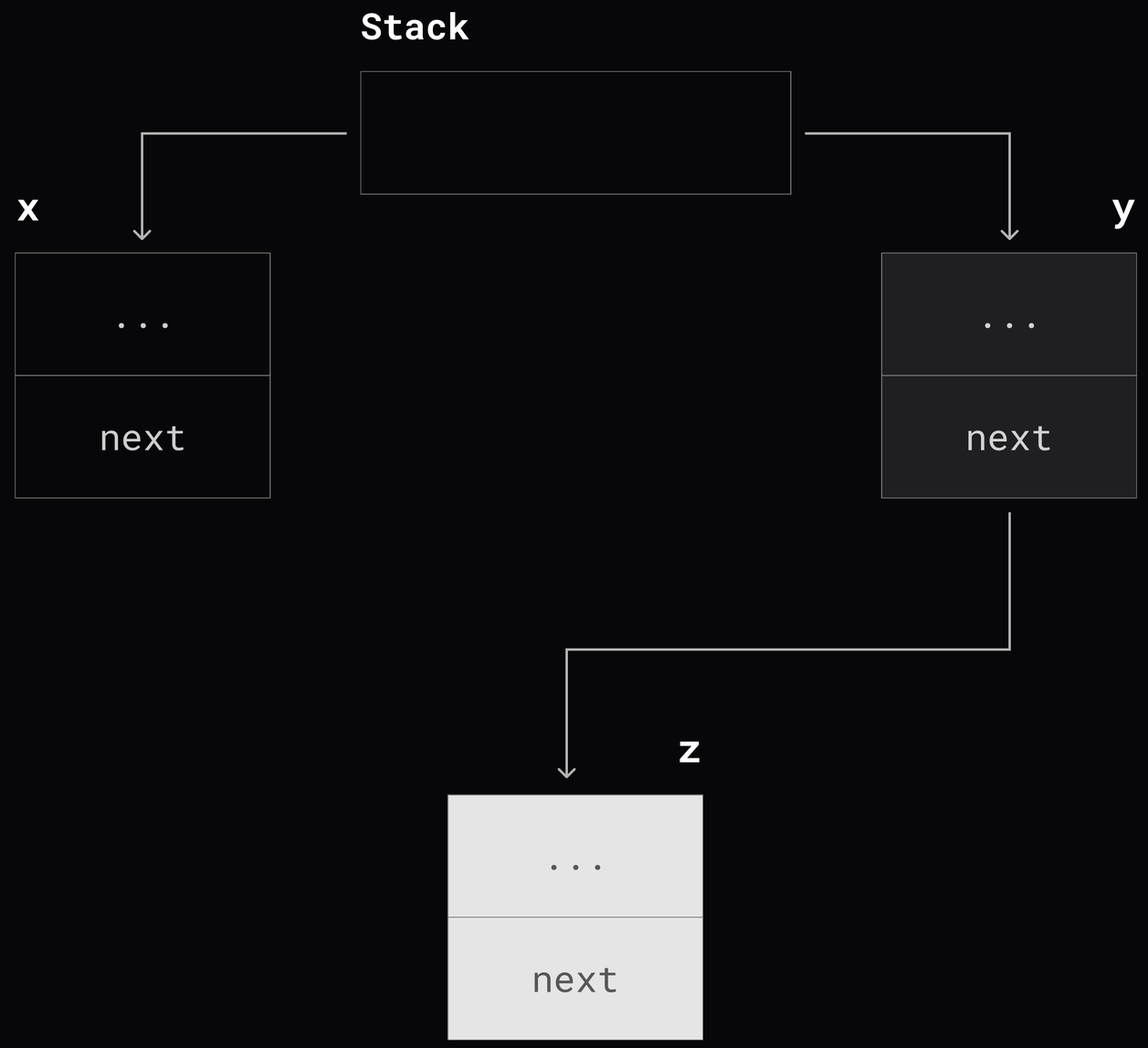
ВАЖНО НЕ ПУТАТЬ CONCURRENT GC С PARALLEL GC

Потому что Parallel GC - это сборка мусора со STW,
но потоки для сборке мусора работают параллельно

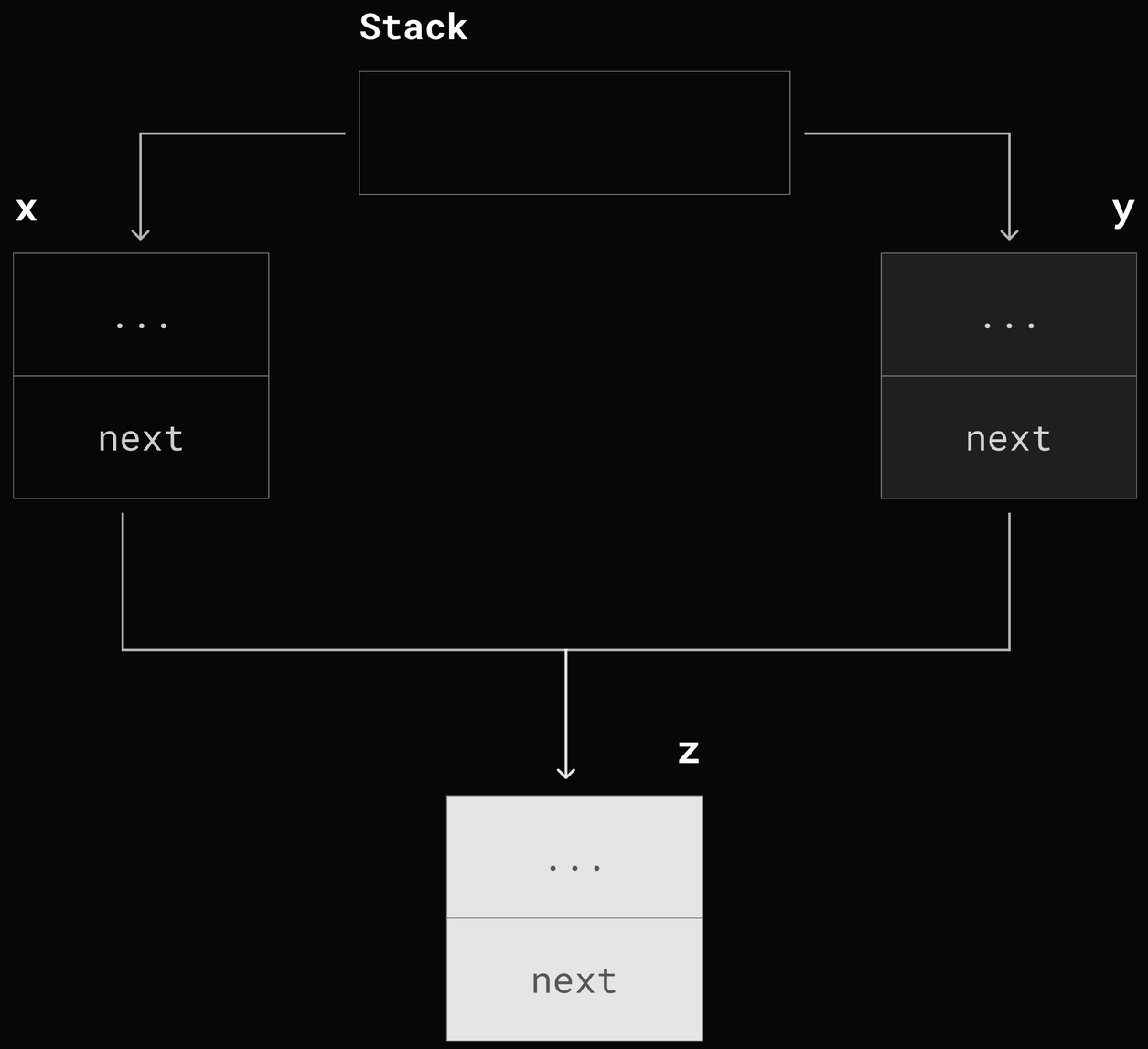
КАКИЕ ПРОБЛЕМЫ МОГУТ ВОЗНИКНУТЬ

во время конкурентного запуска сборщика мусора с кодом приложения?

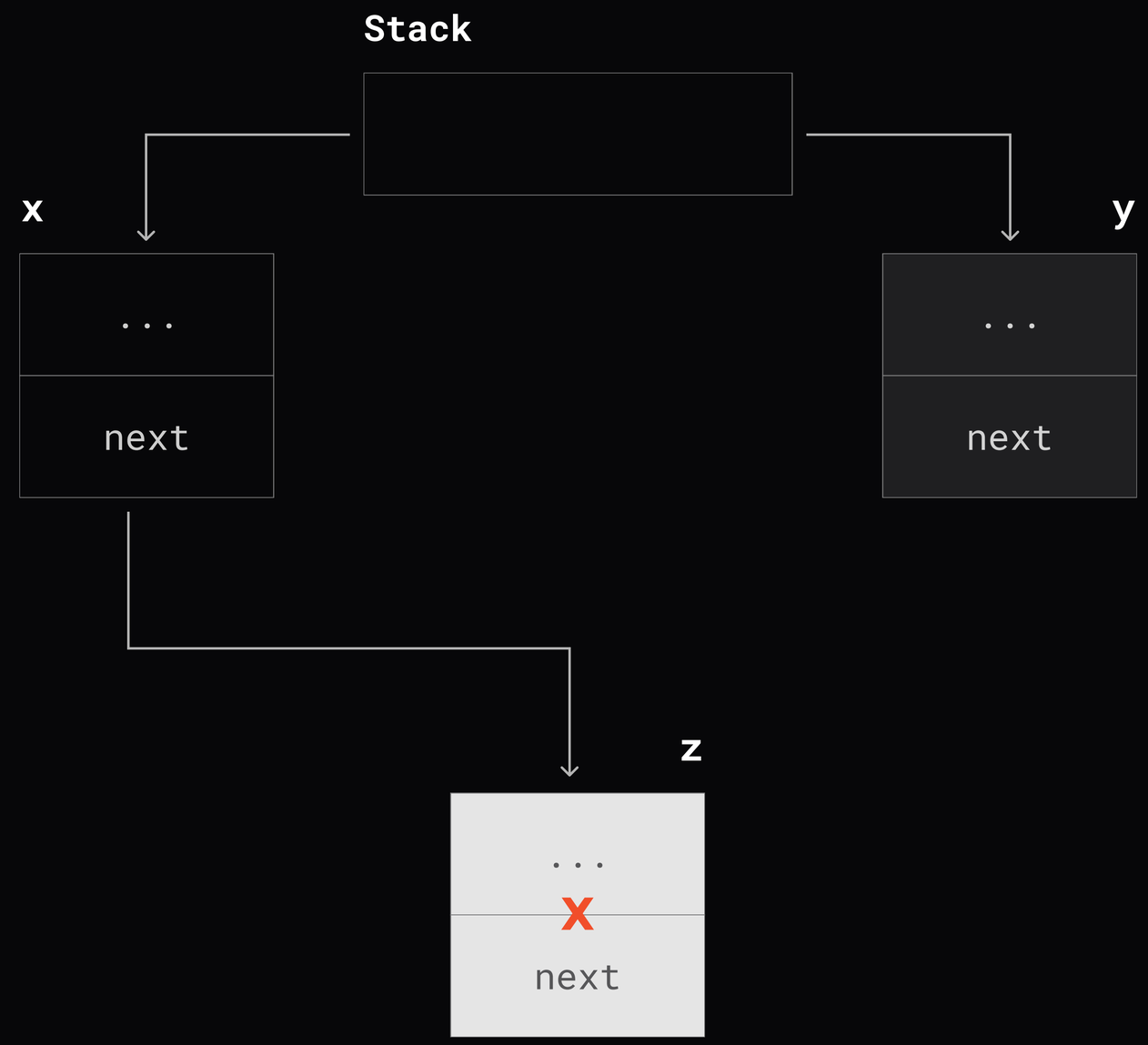
```
1 [...]
2 x := &node{}
3 y := &node{}
4
5 y.next = &node{}
6 [...]
```

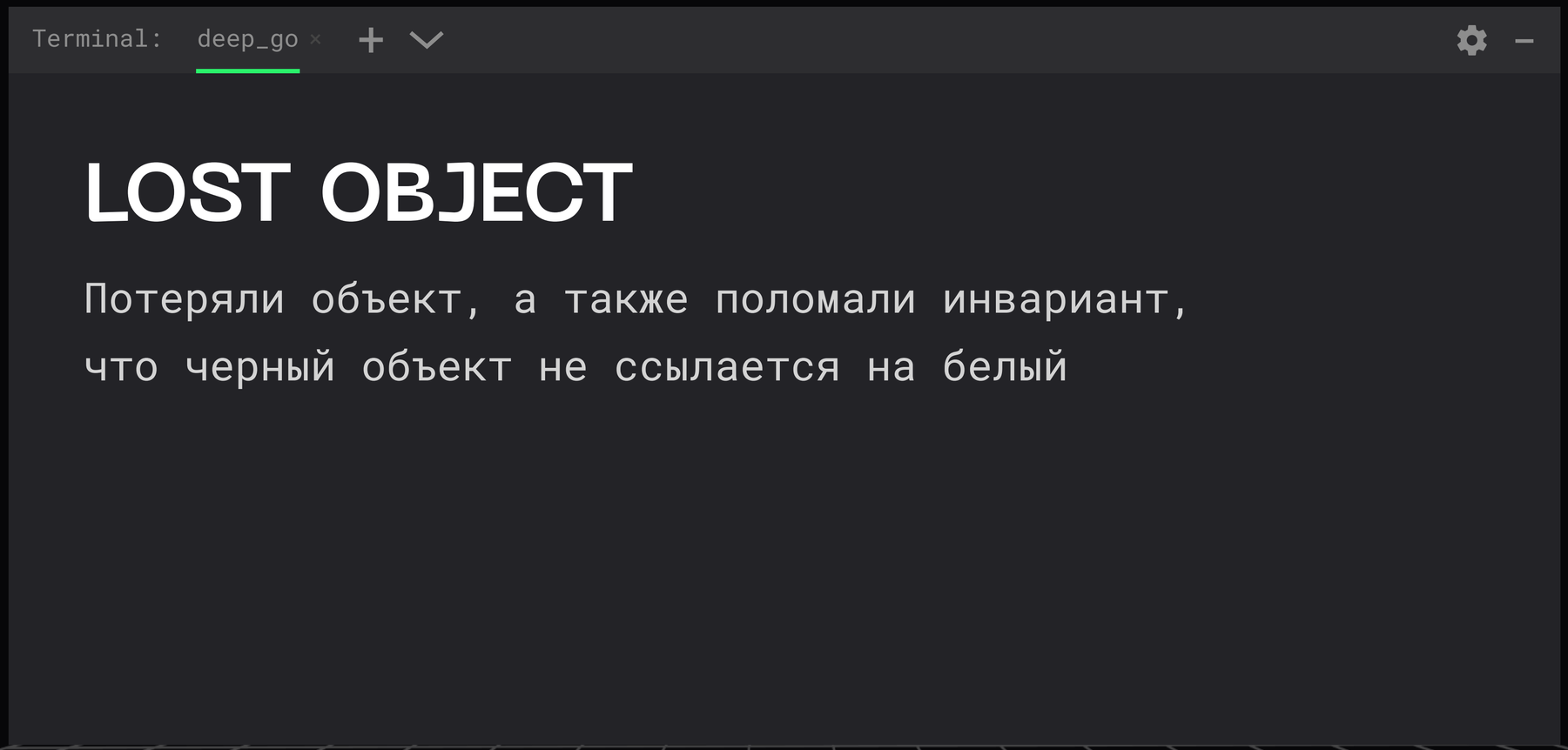


```
1 [...]
2 x := &node{}
3 y := &node{}
4
5 y.next = &node{}
6 x.next = y.next
7 [...]
```



```
1 [...]
2 x := &node{}
3 y := &node{}
4
5 y.next = &node{}
6 x.next = y.next
7 y.next = nil
8 [...]
```




Terminal:
deep_go
LOST OBJECT
Потеряли объект, а также поломали инвариант,
что черный объект не ссылается на белый

Terminal: question + ⌄ ⚙ –

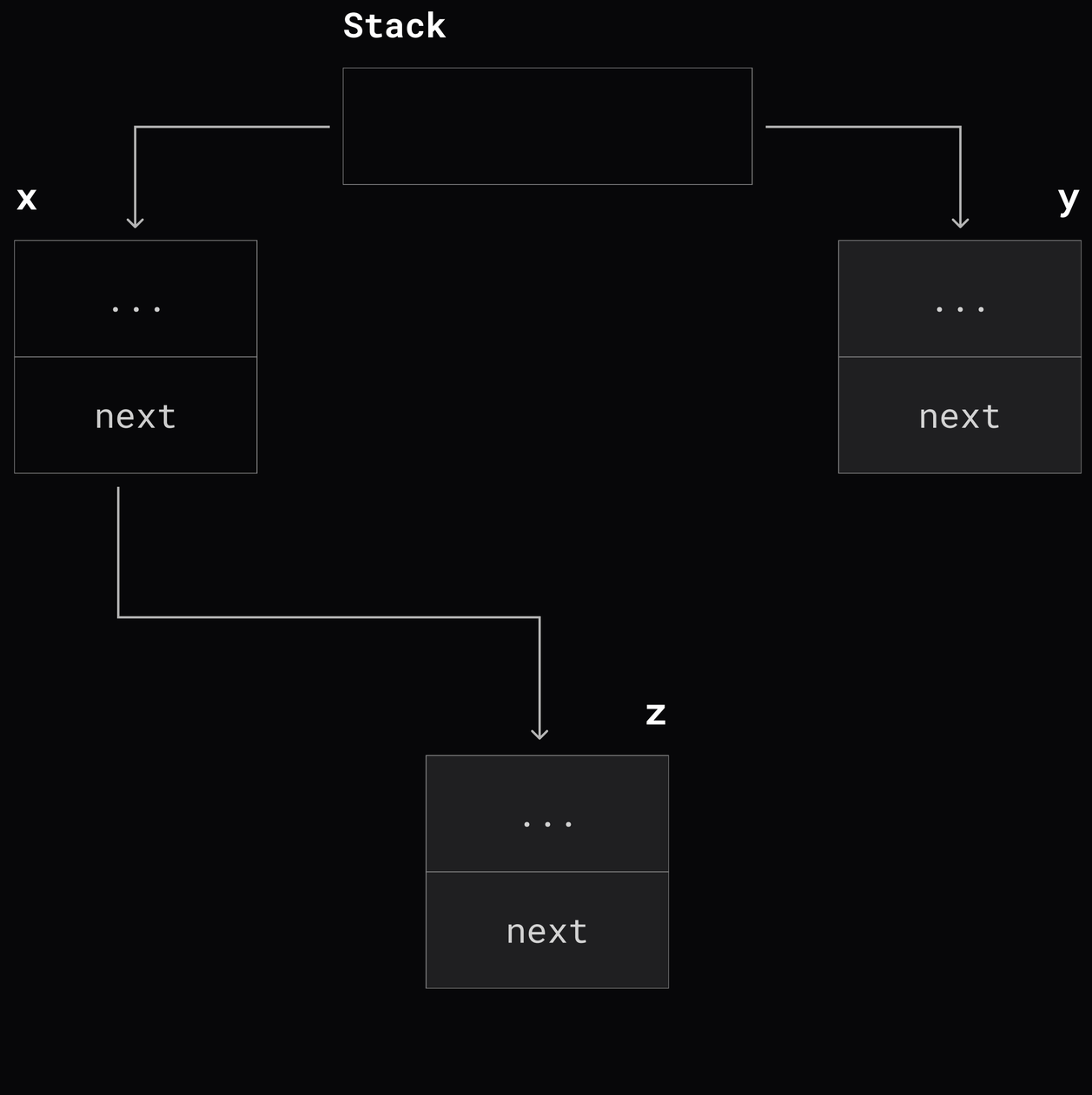
КАК МОЖНО РЕШИТЬ ТАКУЮ ПРОБЛЕМУ?

Terminal: deep_go ×

БАРЬЕР ЗАПИСИ

Если объект черный и он начинает ссылаться на новый объект, то покрасим указатель на новый цвет сразу в серый цвет (серые объекты добавляются в очередь на обход)

```
1 [...]
2 x := &node{}
3 y := &node{}
4
5 x.next = &node{}
6 [...]
```

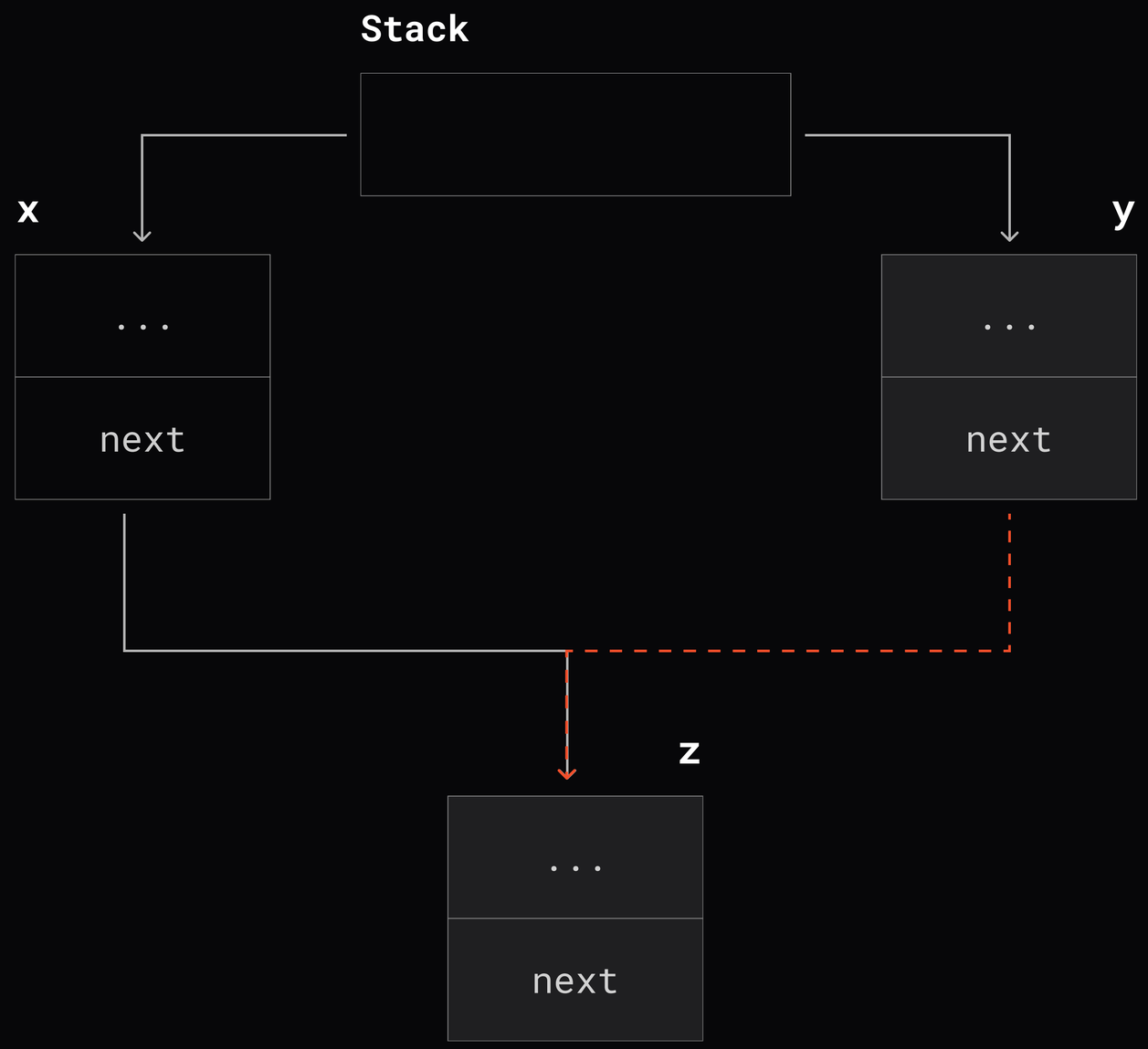


Terminal: deep_go

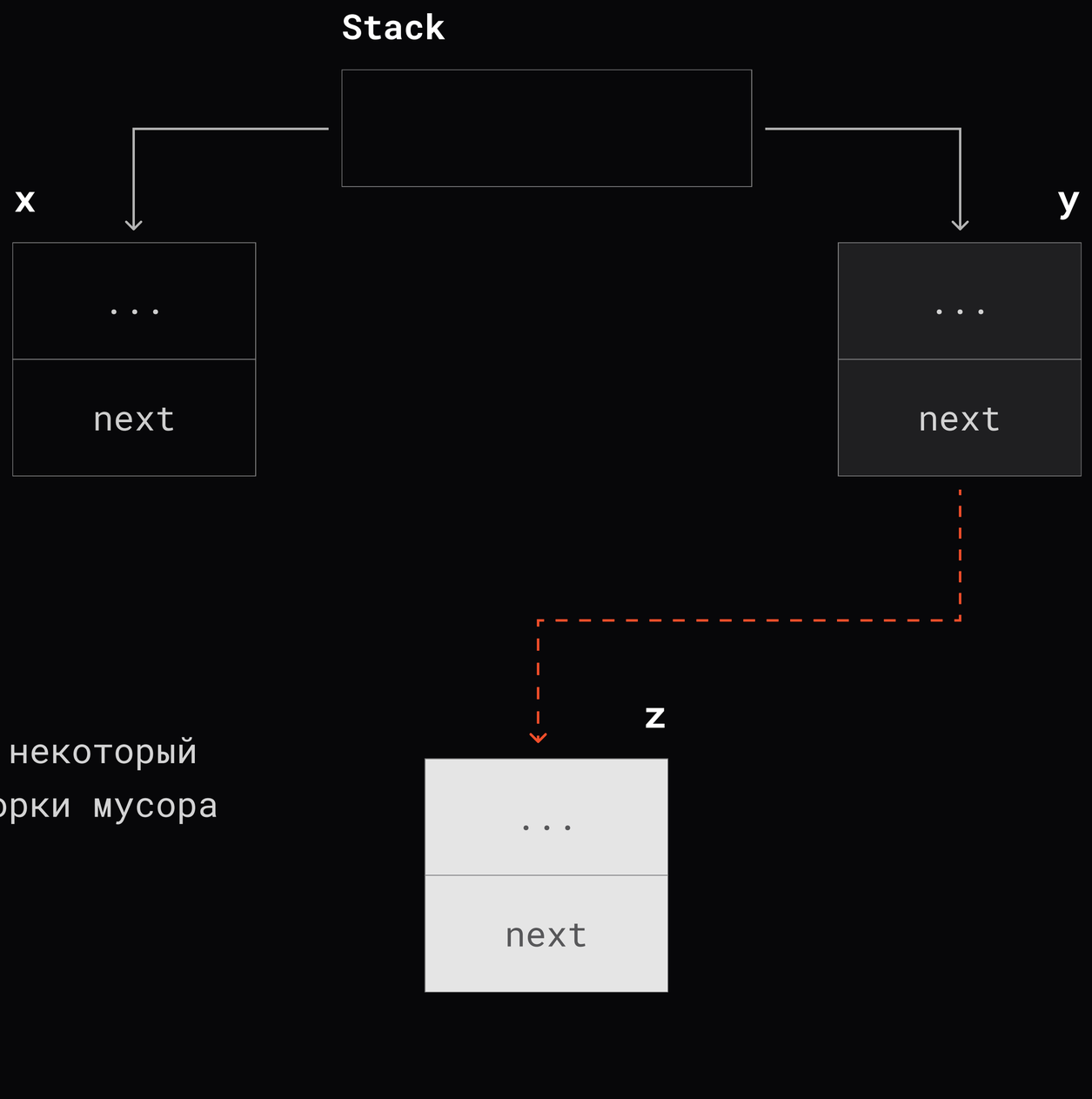
БАРЬЕР УДАЛЕНИЯ

Покрасим объект в серый цвет, на который теряется ссылка (серые объекты добавляются в очередь на обход)

```
1 [...]
2 x := &node{}
3 y := &node{}
4
5 y.next = &node{}
6 x.next = y.next
7 y.next = nil
8 [...]
```



```
1 [...]
2 x := &node{}
3 y := &node{}
4
5 y.next = &node{}
6 y.next = nil
7 [...]
```



При использовании барьера удаления – некоторый мусор может доживать до следующей сборки мусора

ДЛЯ ВКЛЮЧЕНИЯ/ОТКЛЮЧЕНИЯ БАРЬЕРОВ ЧАЩЕ ВСЕГО ТРЕБУЮТСЯ

КОРОТКИЙ STW

Terminal: question

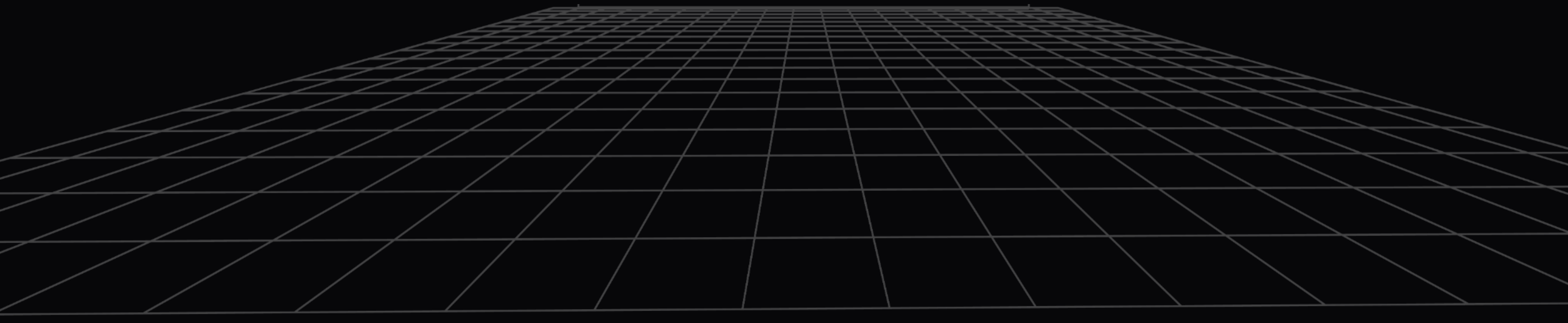
ЧТО ДЕЛАТЬ, ЕСЛИ ОЧЕРЕДЬ СЕРЫХ ОБЪЕКТОВ НЕ БУДЕТ ЗАКАНЧИВАТЬСЯ?

НУЖНО СОЗДАВАТЬ ОБЪЕКТЫ СРАЗУ ЧЕРНОГО ЦВЕТА, ИНАЧЕ ОБЪЕКТЫ МОГУТ НЕ ЗАКОНЧИТЬСЯ НИКОГДА

Если сборка мусора не выполняется, то не нужно мутатору оповещать поток сборщика мусора (то есть не нужно раскрашивать объекты в серый или черный цвет)

FAQ

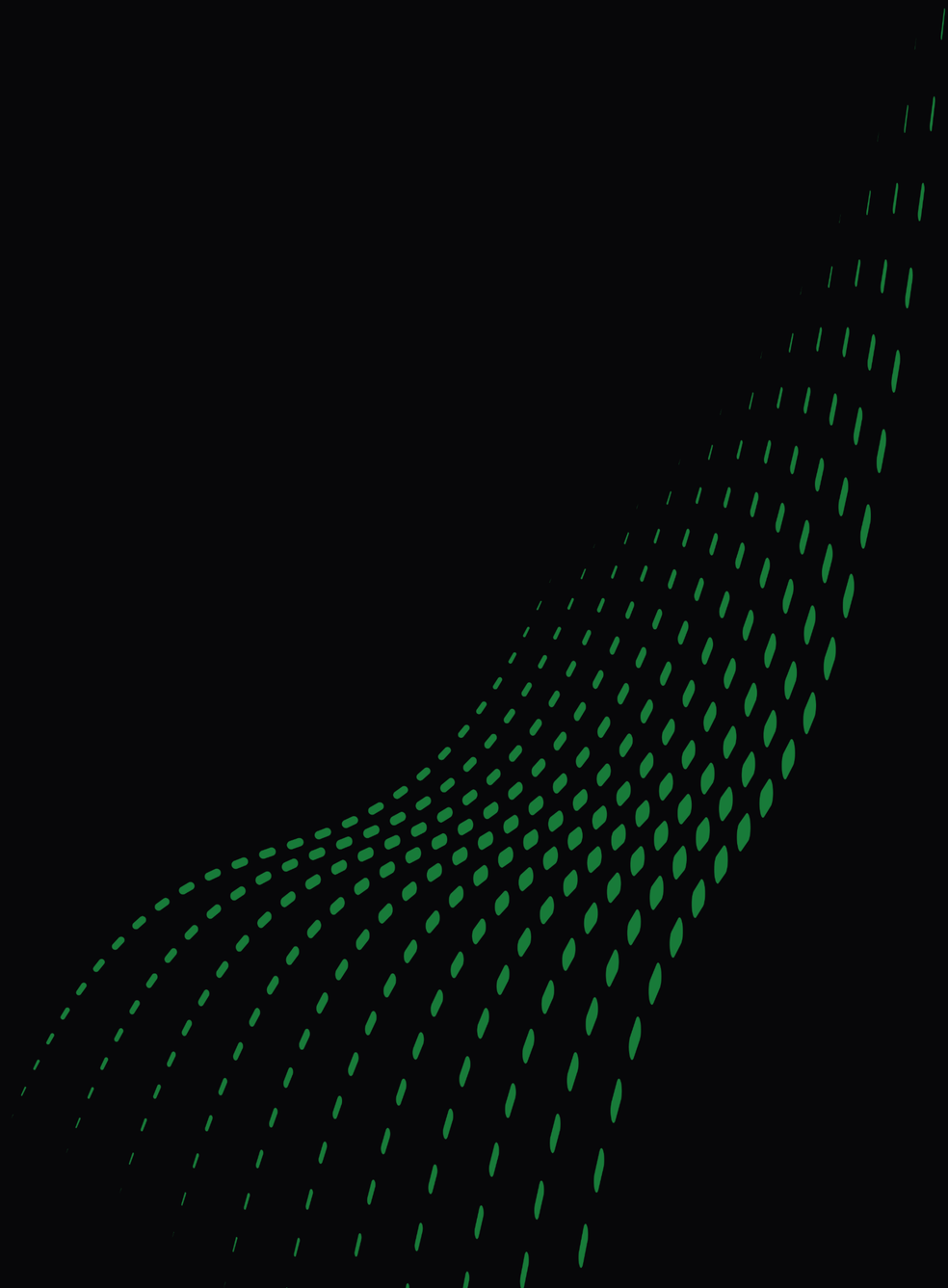
Трейсинг



УСТРОЙСТВО GC В GO

GO GC

- **Mark and Sweep** — алгоритм GC
- **Трехцветный** - алгоритм разметки
- **Конкурентный** — выполняется конкурентно с основной программой с использованием барьеров записи



В Go не стали заимствовать у Java концепцию поколений объектов.

Это связано с тем, что в Go гораздо большая роль отводится аллокациям на стеке. Короткоживущие объекты с большой вероятностью будут размещены на стеке, а не в хипе, поэтому идея о молодом поколении объектов не находит своего применения

В настоящее время фракция CPU, потребляемая воркерами GC, жёстко зафиксирована на уровне 25%. Если GC понимает, что не справляется с потоком мусора, он может слегка притормозить новые аллокации, заставляя некоторые горутины тратить часть времени на сборку мусора

> ℹ эта оптимизация называется Mark Assist и позволяет довести потребляемую GC фракцию CPU до 30%

Terminal: question + ⌄ ⚙ –

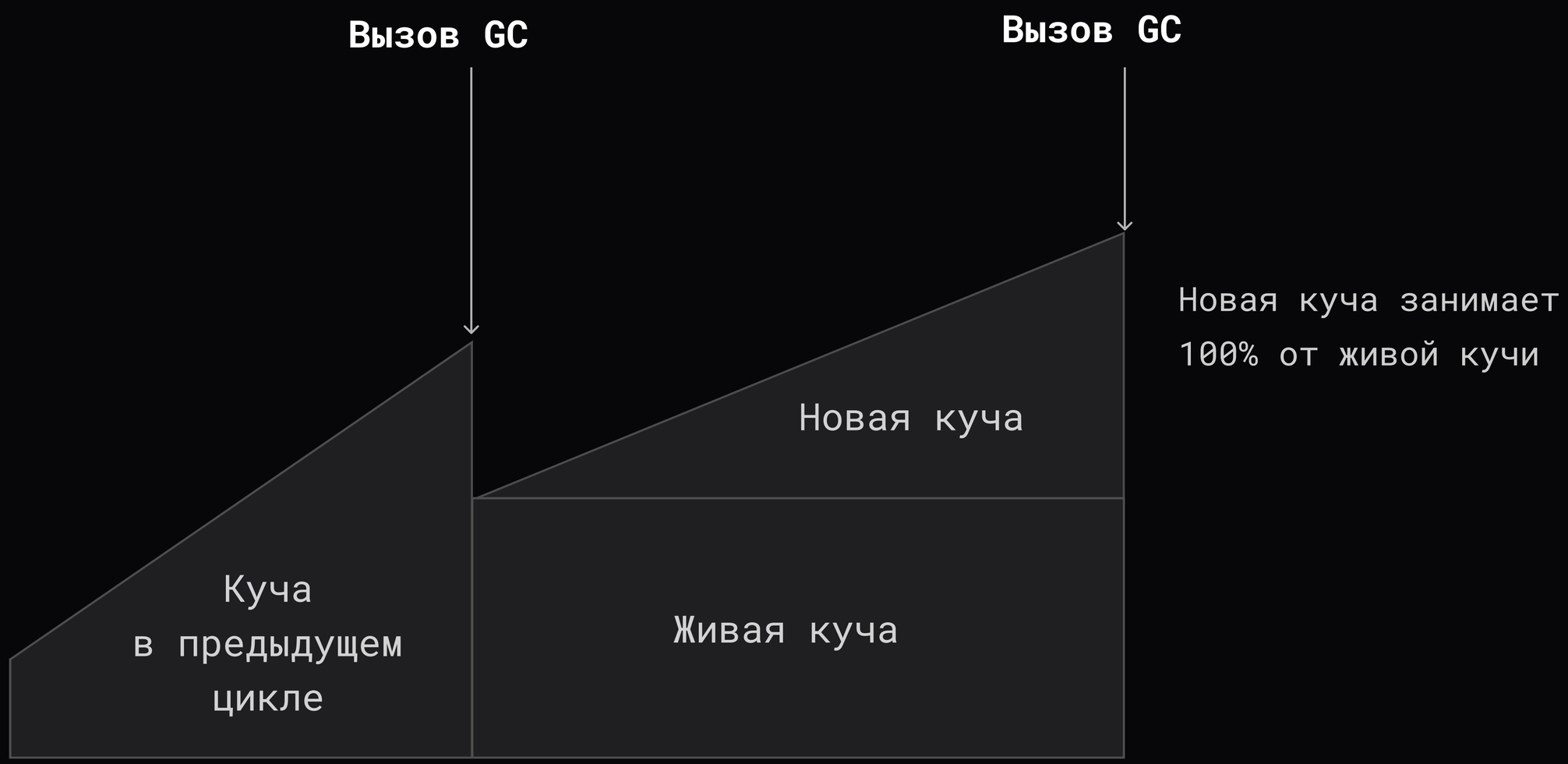
КОГДА ЗАПУСКАЕТСЯ СБОРЩИК МУСОРА В GO?

КОГДА ЗАПУСКАЕТСЯ СБОРЩИК МУСОРА В GO?

1. **Превышение динамического лимита кучи,** установленного с помощью переменной GOGC

2. **Прошло 2 минуты без GC** - если вы ничего не аллоцировали за это время, GC все равно запустится раз в 2 минуты (за это отвечает sysmon)

3. **Ручной запуск с помощью runtime.GC()** - если сделать этот вызов, когда Garbage Collector уже запущен, то по достижении фазы sweep он запустится заново


Вызов GC
Вызов GC
Новая куча занимает
100% от живой кучи
Новая куча
Куча
в предыдущем
цикле
Живая куча

EXAMPLE

Memory ballast

RSS (Resident Set Size) is used to show how much memory is allocated to that process and is in RAM. It does not include memory that is swapped out. It does include memory from shared libraries as long as the pages from those libraries are actually in memory.

VSZ (Virtual Memory Size) includes all memory that the process can access, including memory that is swapped out, memory that is allocated, but not used, and memory that is from shared libraries.

```
vladimirbalun@Vladimirs-MacBook-Pro golang_course % ps -o pid,vsz,rss \
        | awk '{if (NR == 1 || $1 == "29375") print}'
  PID      VSZ    RSS
29375 419700592   1728
```

EXAMPLE

Big allocatoins

Terminal: deep_go

LAZY ALLOCATION

After allocation, your address space is expanded immediately, but Linux does not assign actual pages of physical memory until the first write to the page

Представим, что наш процесс работает в виртуальной машине с установленным лимитом на RSS в 10 ГБ со значением GOGC = 100%

Допустим, после последнего прохода GC размер хипа составил 5.1 ГБ — вроде бы до лимита ещё далеко. Дальше берем прошлое значение и пользуемся формулой для вычисления целевого значения размера хипа для следующей сборки мусора (10.2 ГБ)

Оказывается, что оно выше, чем лимит на RSS.
Иными словами, прежде чем GC сработает
в следующий раз, процесс будет убит OOM Killer'ом!

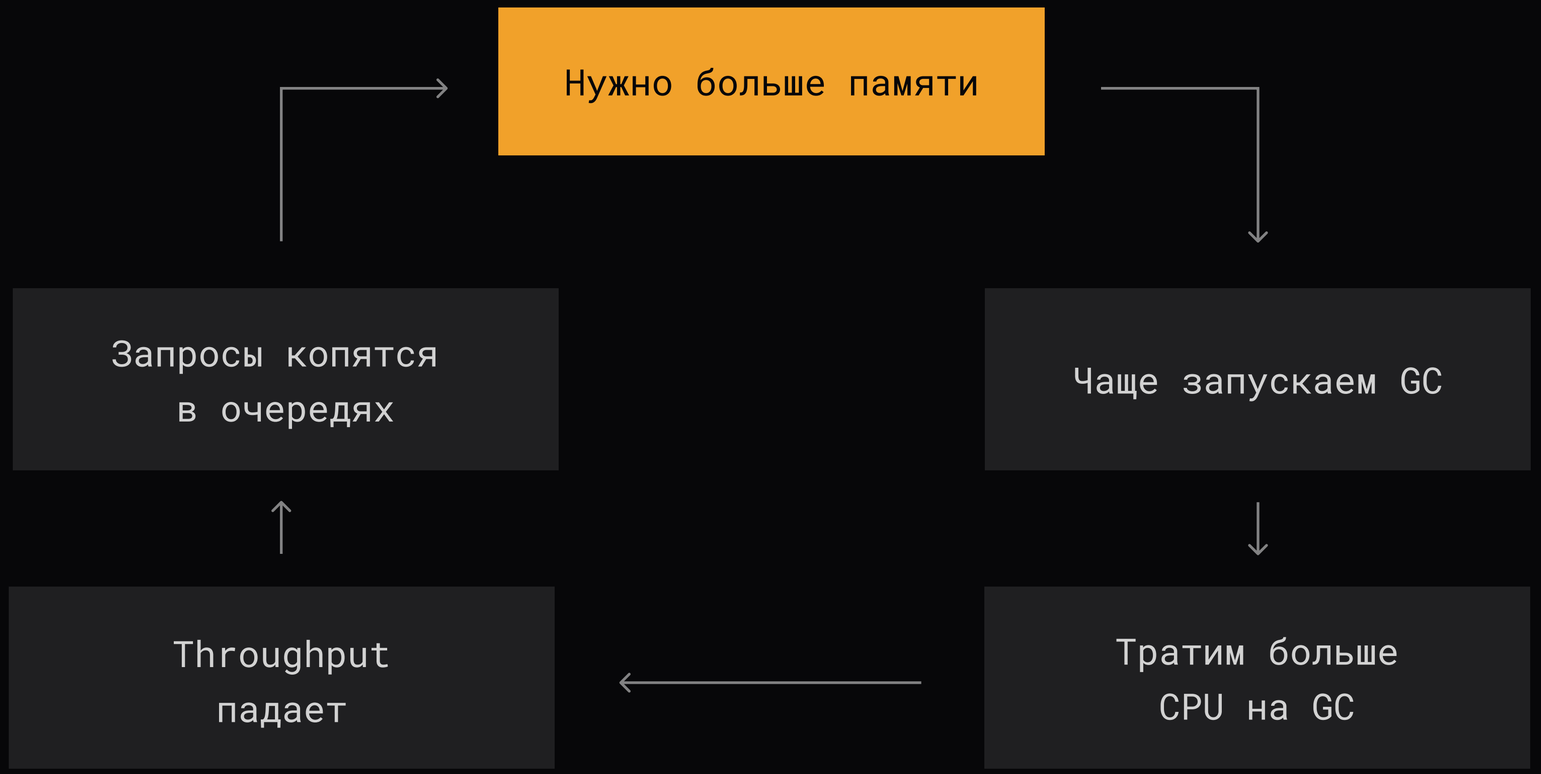
Terminal: question + ⌄ ⚙ –

ЧТО ДЕЛАТЬ В ТАКОЙ СИТУАЦИИ?

Нужно освобождать больше памяти, то есть чаще запускать GC. В Go для этого нужно выбрать более консервативное значение GOGC. **Однако ручной подбор GOGC под конкретный сервис и конкретную нагрузку — не очень весёлое занятие.**

DEATH SPIRAL

Можно перестараться с GOGC и столкнуться со спиралью смерти
(такую проблему сложнее обнаружить, чем тот же самый OOM)



Terminal: Concurrency

GOMEMLIMIT

Учитывая всю память потребляемую приложением (не только кучу) и ориентируясь на заданную верхнюю границу потребления памяти, рантайм будет чаще вызывать сборку мусора и более агрессивно возвращать память операционной системе

Выставить лимит можно при помощи переменной окружения, например так: **GOMEMLIMIT=2750MiB**

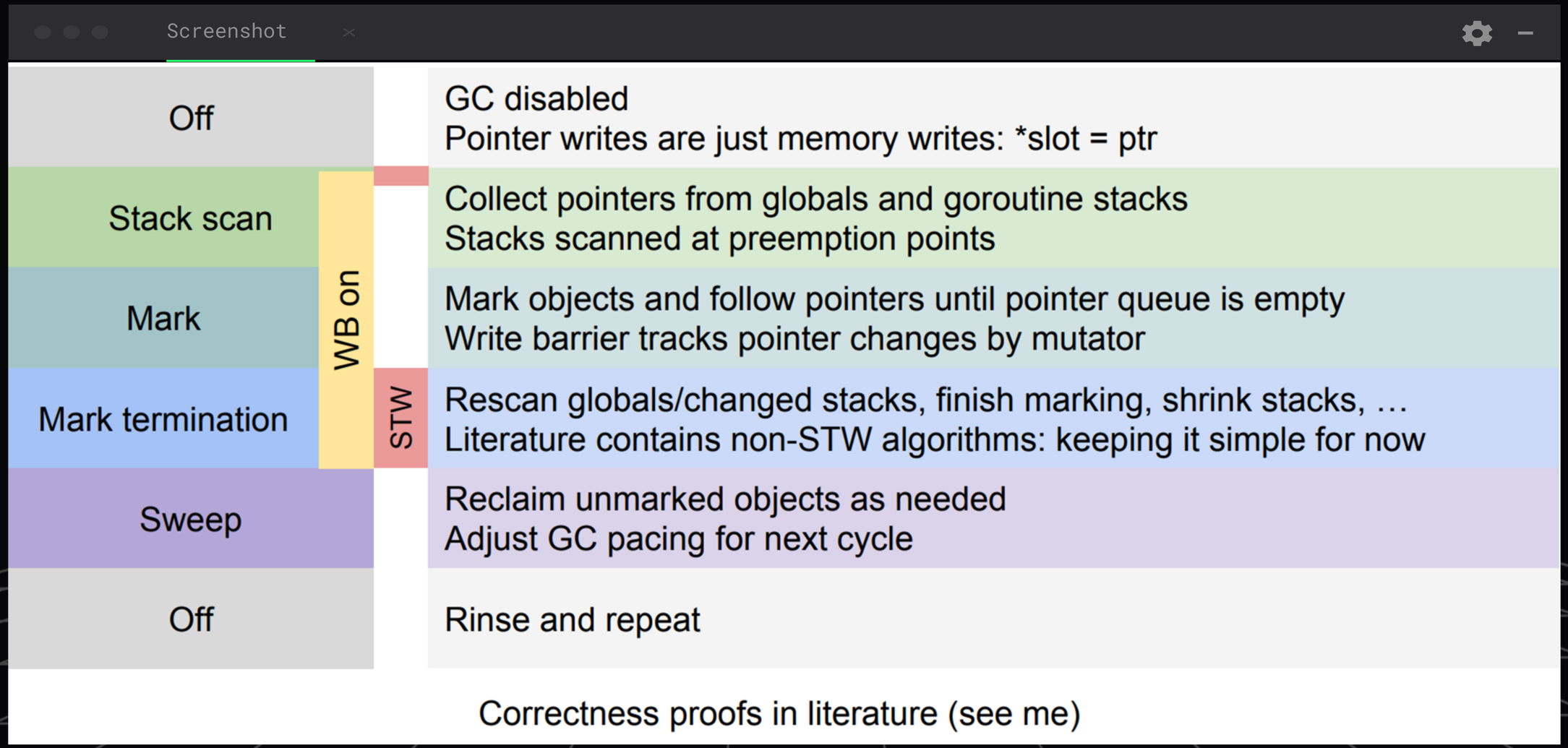
Чтобы избежать скатывания GC в спираль смерти для GOMEMLIMIT, вводится искусственное ограничение на потребление сборщиком мусора CPU — оно не превысит 50% даже в самых жёстких ситуациях, когда мусора очень много, а память почти исчерпана.

В этом случае приложение сможет продолжить аллоцировать новую память, что с большой вероятностью приведёт к преодолению лимита (в этом смысле он является мягким, так как позволяет использовать память сверх лимита)

Terminal: question + ⌄ ⚙ –

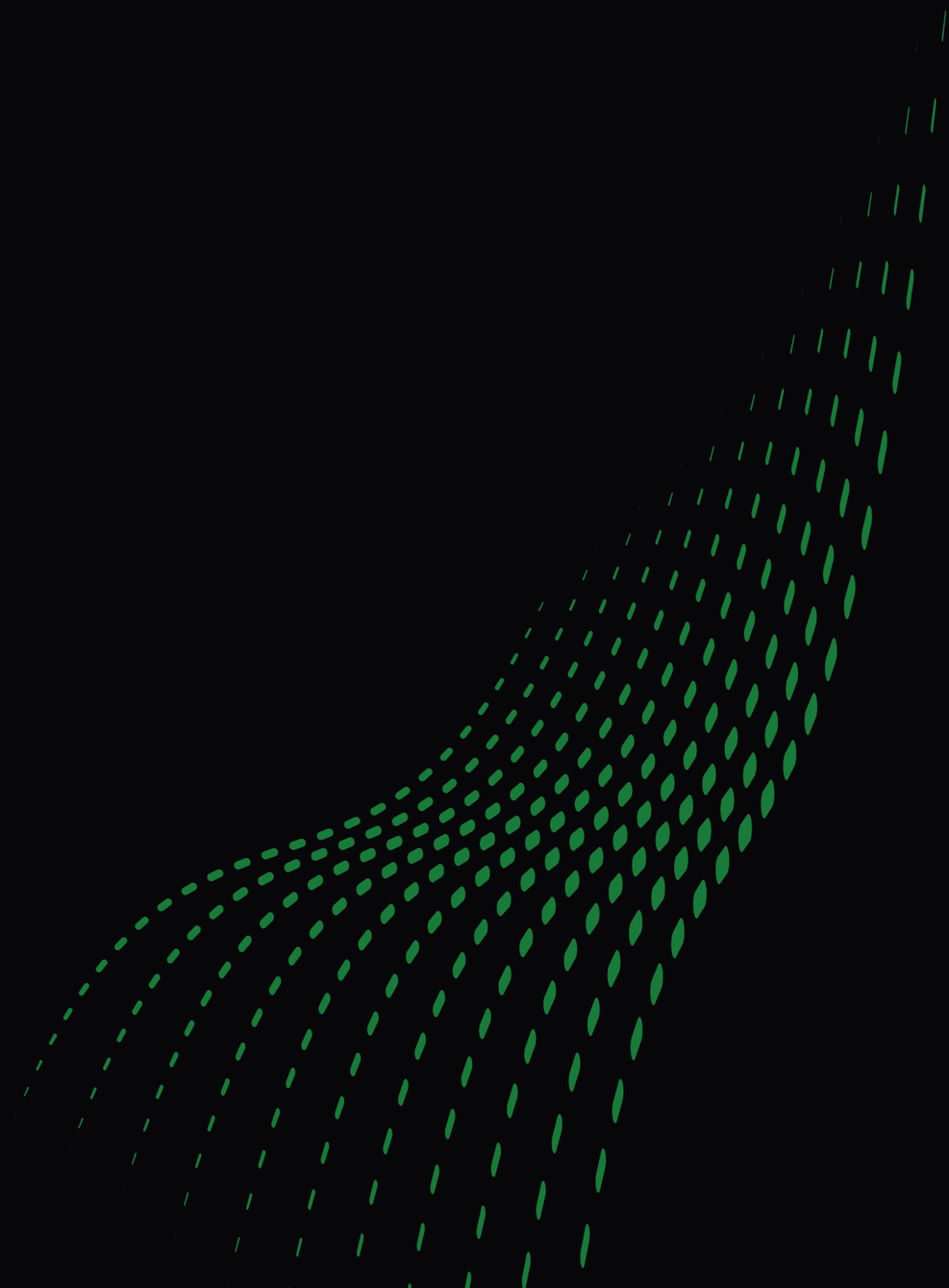
ИЗ КАКИХ ФАЗ СОСТОИТ СБОРЩИК МУСОРА?

GC ALGORITHM PHASES



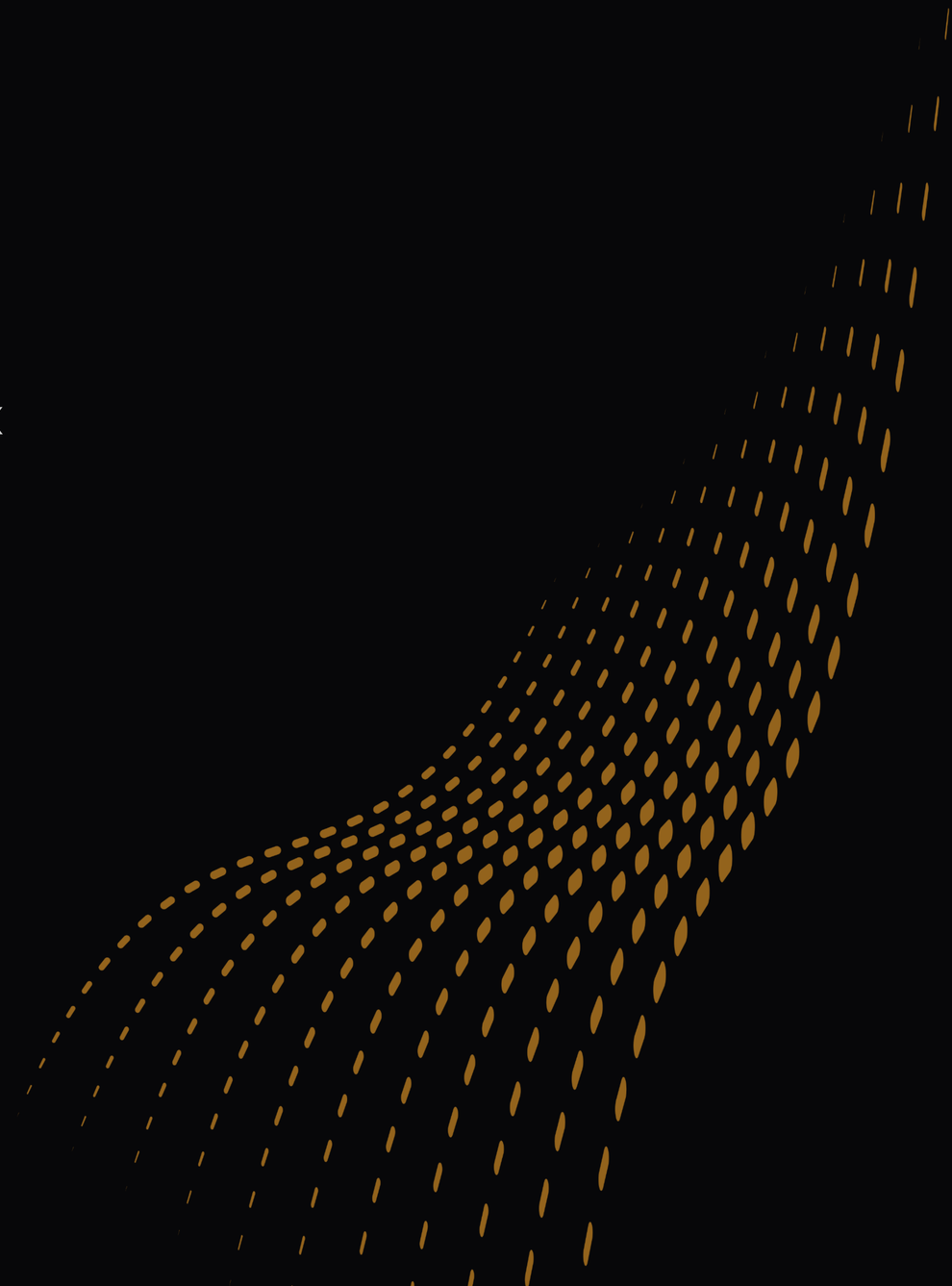
SWEEP TERMINATION

- **Stop the world**
- Завершение всех sweep фаз
- Удаление остатков мусора



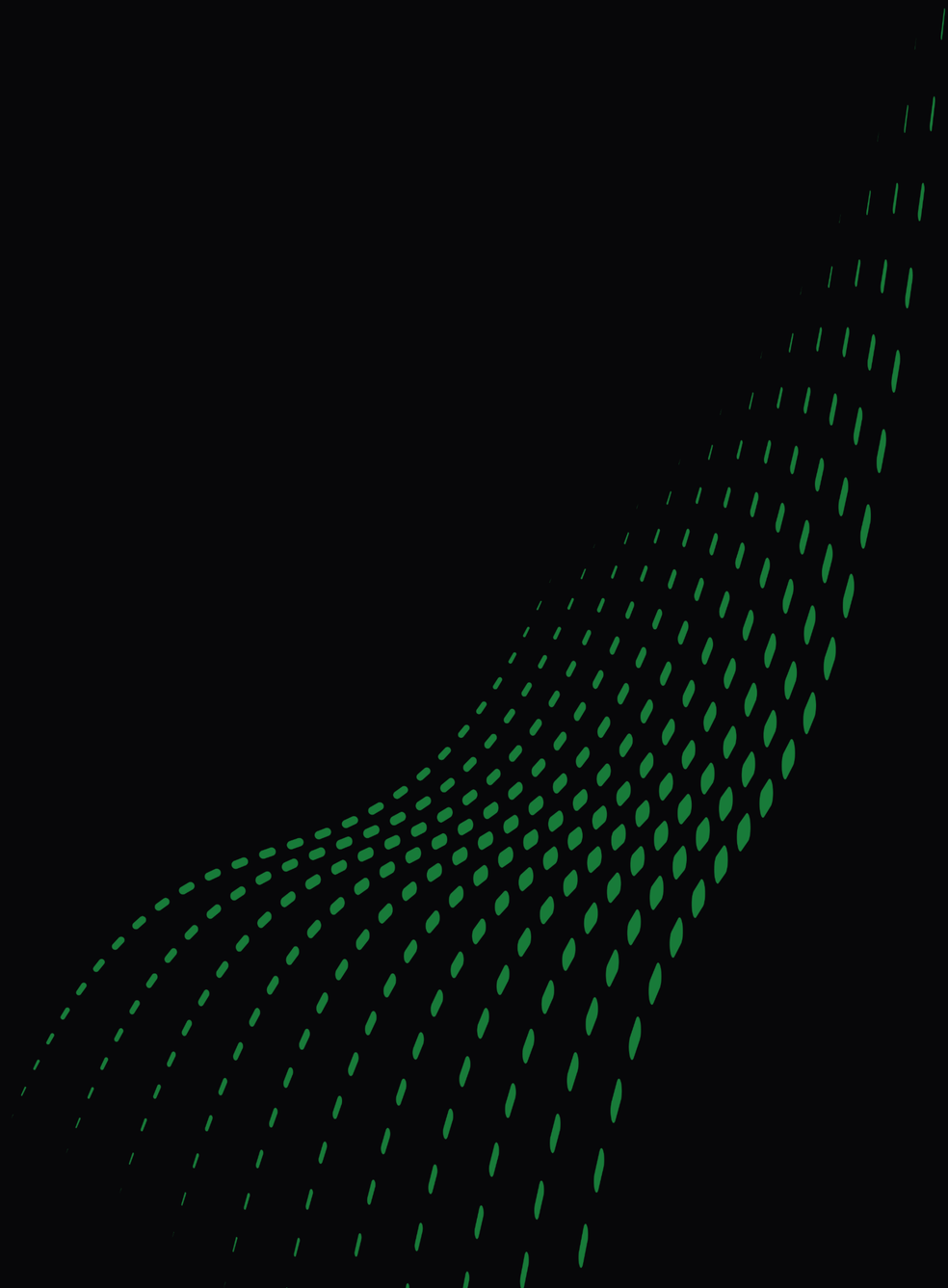
MARK

- **Включаем write barrier**
- **Start the world**
- Запускаем сканирование глобальным переменных и стеков (при сканировании стека, горутина приостанавливается)
- Раскрашиваем объекты в три цвета



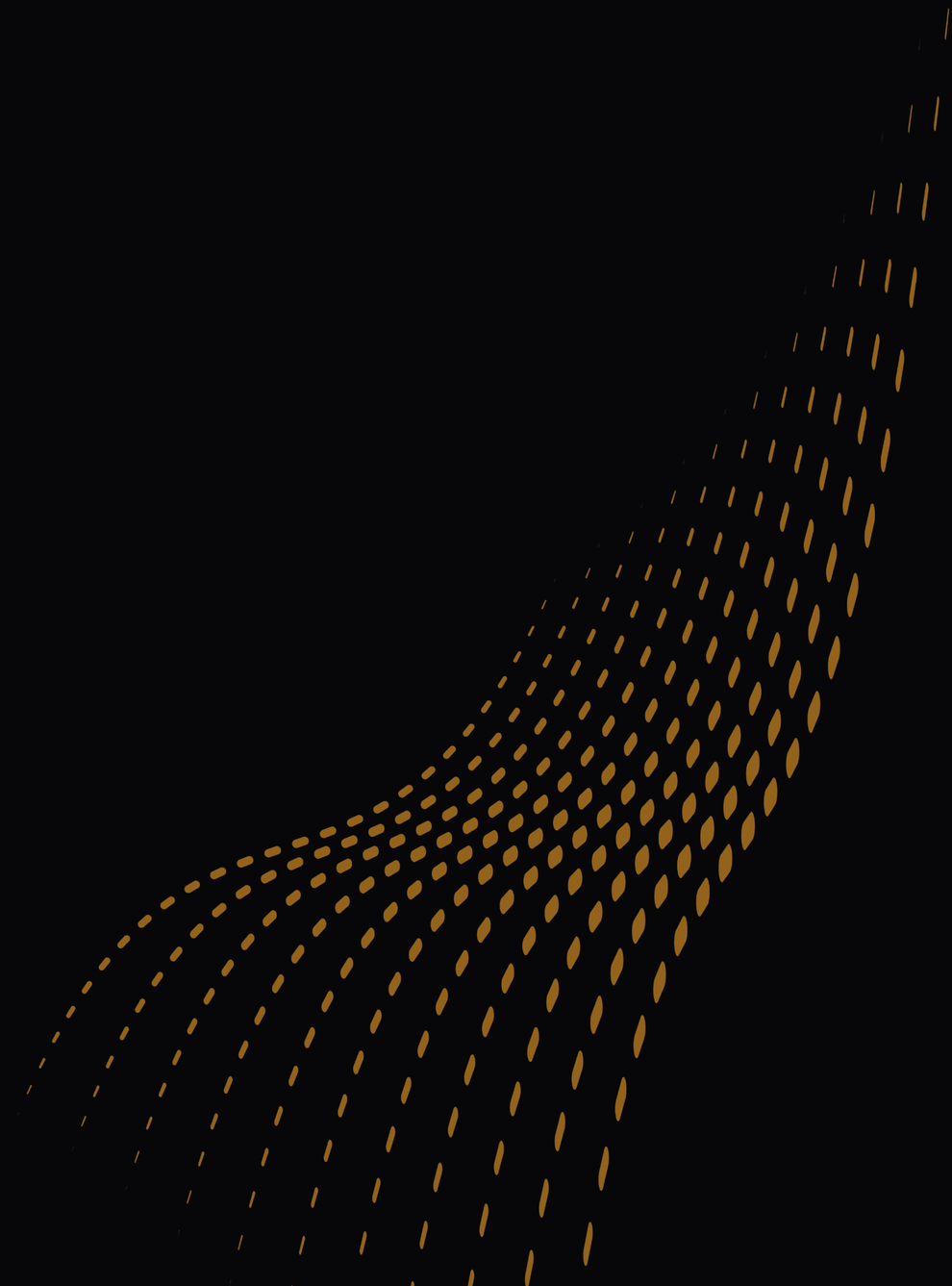
MARK TERMINATION

- **Stop the world**
- Дожидаемся обработки последних задач из очереди
- Завершаем разметку



MARK

- **Отключаем write barrier**
- **Start the world**
- Очищаем ресурсы в фоне



Terminal: question + ⌄ ⚙ –

КАК ПОНЯТЬ, В КАКОЙ СЕЙЧАС СТАДИИ НАХОДИТСЯ GC?

ТЕКУЩАЯ СТАДИЯ РАБОТЫ СБОРЩИКА МУСОРА ХРАНИТСЯ В ГЛОБАЛЬНОЙ ПЕРЕМЕННОЙ GCPHASE

Terminal: question + ⌄ ⚙ –

КТО ВЫПОЛНЯЕТ КОД СБОРКИ МУСОРА?

«M» может исполнять пользовательскую горутину,
а также может исполнять горутину GC

Terminal: question + ⌄ ⚙ –

КАК УМЕНЬШИТЬ НАГРУЗКУ НА СБОРЩИК МУСОРА?

Стараться переиспользовать выделенную память и больше аллоцировать объектов на стеке, вместо кучи

FINALIZERS

EXAMPLE

Finalizers

If f is garbage collected, a finalizer may close the file descriptor, making it invalid

```
1 func newFile(fd uintptr, name string, kind newFileKind) *File {
2     ...
3
4     runtime.SetFinalizer(f.file, (*file).close)
5     return f
6 }
```

Если вы используете биндинги на C, то по хорошему
не просить программиста явно освобождать память,
а сделать это при помощи финализатора

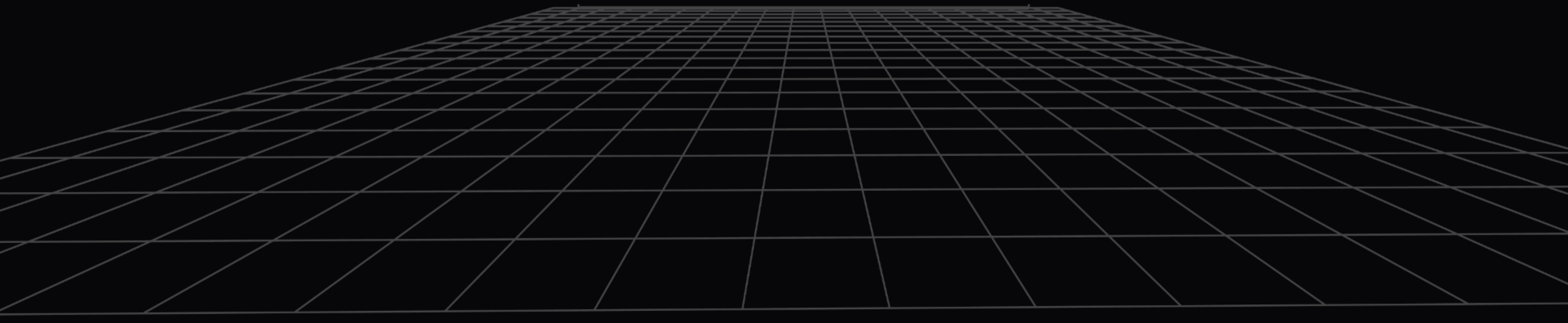
```go
type Cstr struct {
    cpointer *C.char
}

func AllocateAuto() *Cstr {
    answer := &Cstr{C.allocate()}
    runtime.SetFinalizer(answer, func(c *Cstr) {
        C.free_allocated(c.cpointer); runtime.KeepAlive(c)
    })

    return answer
}
```

FAQ

Устройство GC в Go



ПОЖАЛУЙСТА, ЗАПОЛНИ ОПРОС О ЗАНЯТИИ

Ссылка в чате и в группе участников