Ментальная модель языка Go

сентября 18, 2024 · 5 минут · German Gorelkin

Перевод/заметки Mental Model for Go

Основная задача Go - мультиплексировать и абстрагировать аппаратные ресурсы, подобно операционной системе. Для этого обычно используются две основные абстракции:

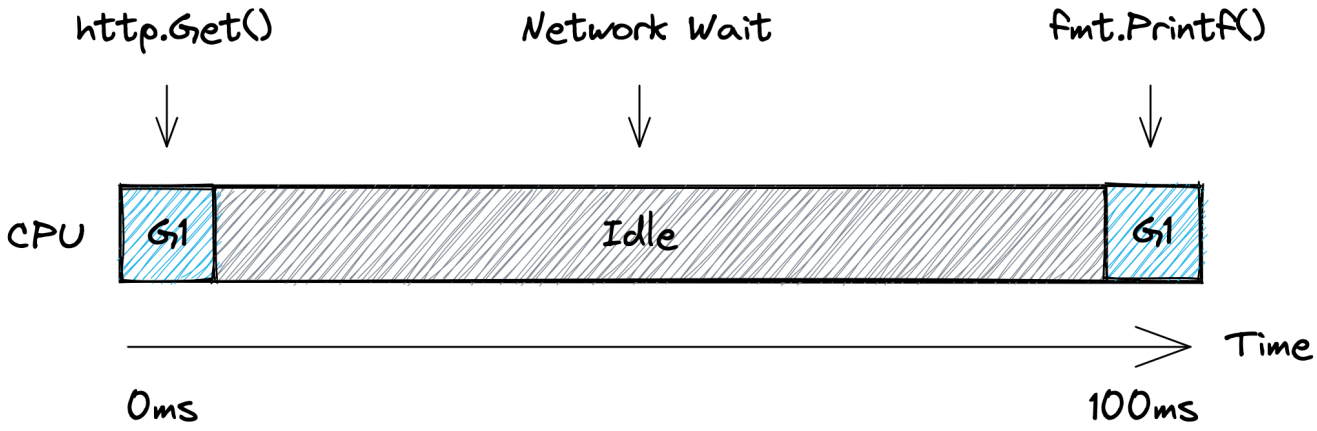
1. **Планировщик горутин(Goroutine Scheduler)**: Управляет тем, как ваш код выполняется на процессорах вашей системы.
2. **Сборщик мусора(Garbage Collector)**: Предоставляет виртуальную память, которая автоматически освобождается по мере необходимости.

Goroutine Scheduler

Сначала поговорим о планировщике на примере ниже:

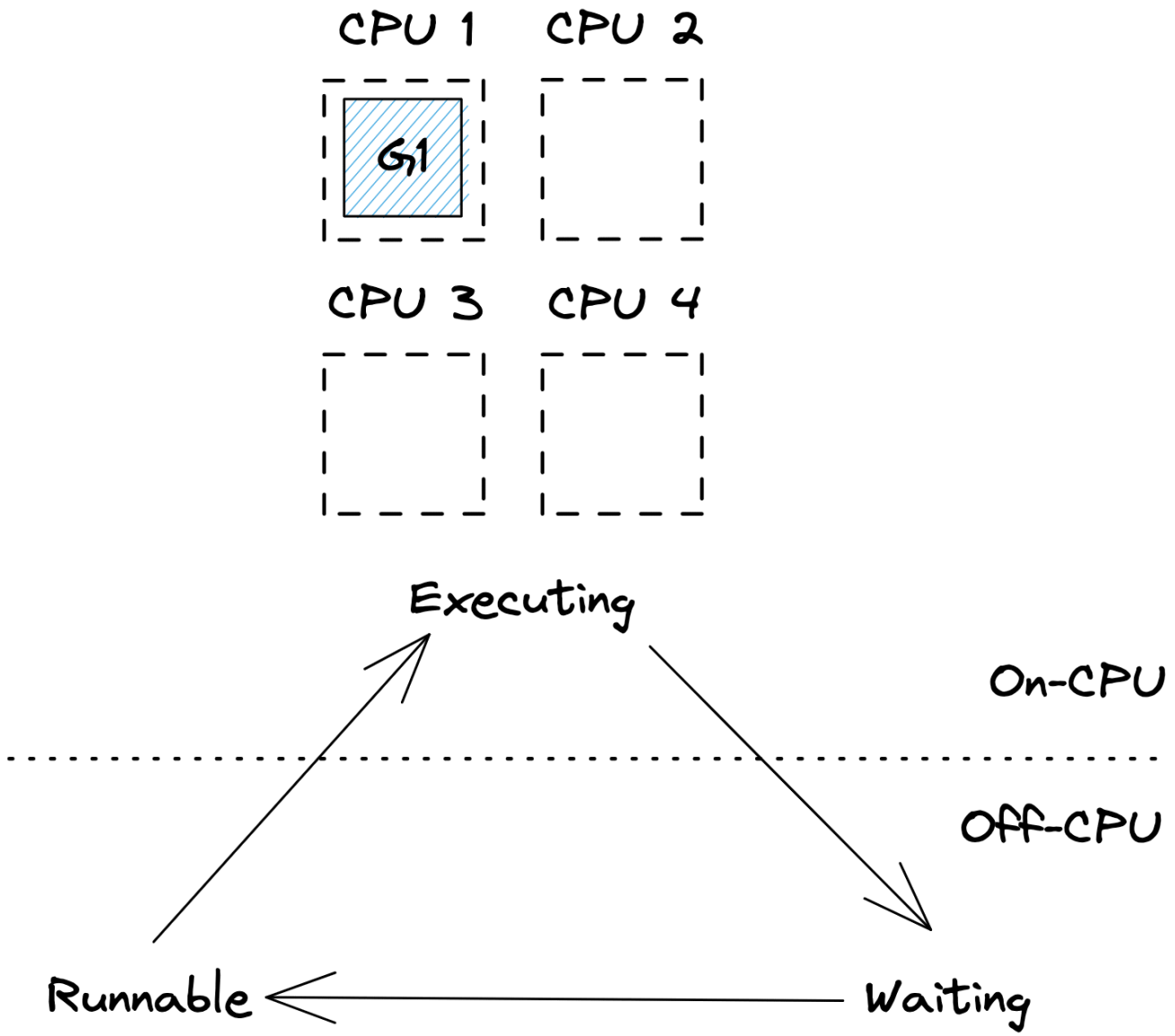
```
func main() {
    res, err := http.Get("https://example.org/")
    if err != nil {
        panic(err)
    }
    fmt.Printf("%d\n", res.StatusCode)
}
```

Здесь у нас есть одна горутина, назовем ее `G1`, которая выполняет `main` функцию. На рисунке ниже показана упрощенная схема выполнения этой функции на одном процессоре. Первоначально `G1` выполняется на CPU для подготовки http-запроса. Затем CPU простаивает(*idle*), так как горутине приходится ждать соединения с сетью. И наконец, она снова запланирована на CPU, чтобы вывести код статуса.

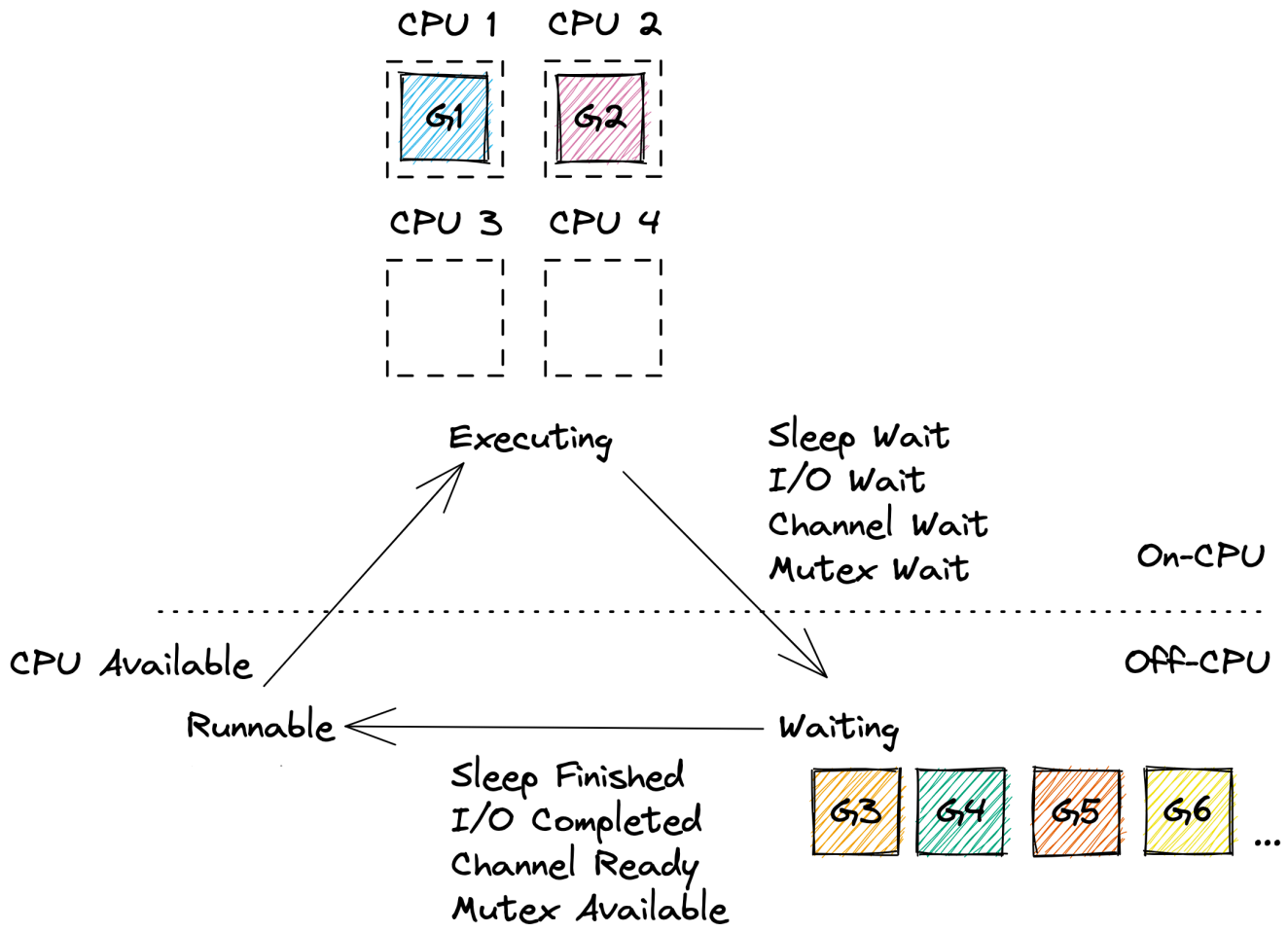


С точки зрения планировщика, приведенная выше программа выполняется так, как показано ниже. Сначала `G1` выполняется(`Executing`) на `CPU 1` . Затем горутина снимается с CPU в ожидании(`Waiting`) сети. Как только планировщик замечает, что сеть ответила, он помечает эту горутину как `Runnable` . И как только ядро процессора становится доступным, горутина снова начинает выполняться. В нашем случае все ядра доступны, поэтому `G1` может вернуться к выполнению(`Executing`) функции `fmt.Printf()` на одном из процессоров немедленно, не тратя времени на состояние `Runnable` .

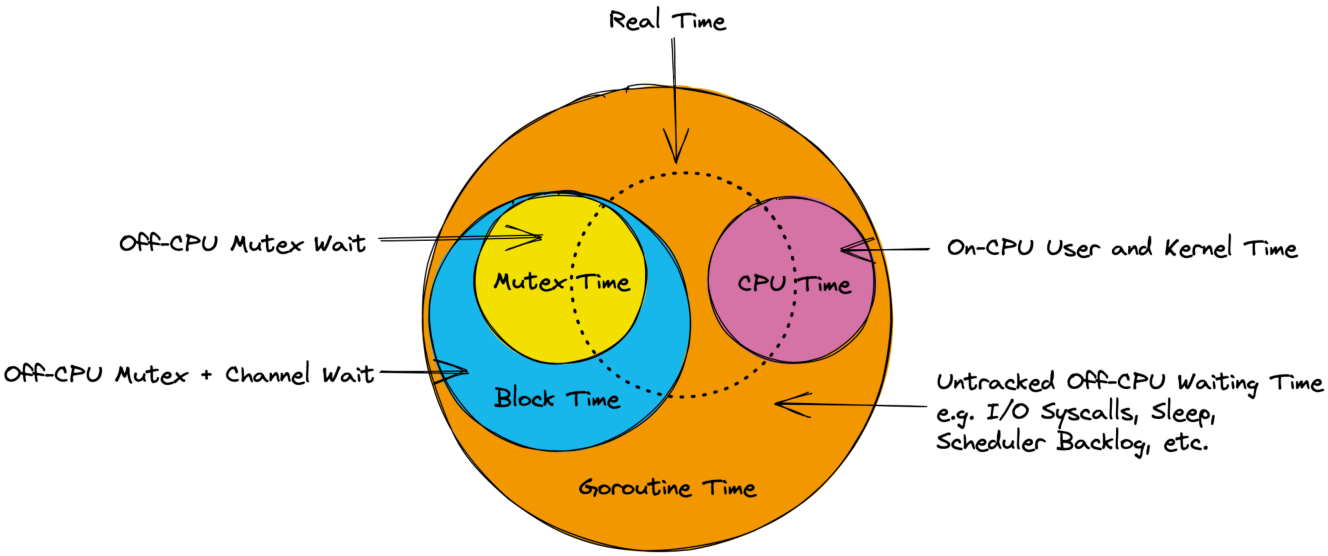
1



Чаще всего в программах на Go выполняется несколько горутин, поэтому у вас будет несколько горутин выполняться(`Executing`) на некоторых ядрах CPU, большое количество горутин будут ожидать (`Waiting`) по различным причинам, и, в идеале, ни одной горутины в состоянии `Runnable` , если только ваша программа не демонстрирует очень высокую загрузку CPU. Пример этого можно увидеть ниже.



Время измеряемое различными профилировщиками Go, - это, по сути, время, которое ваши горутины проводят в состояниях `Executing` и `Waiting` , как показано на диаграмме ниже.

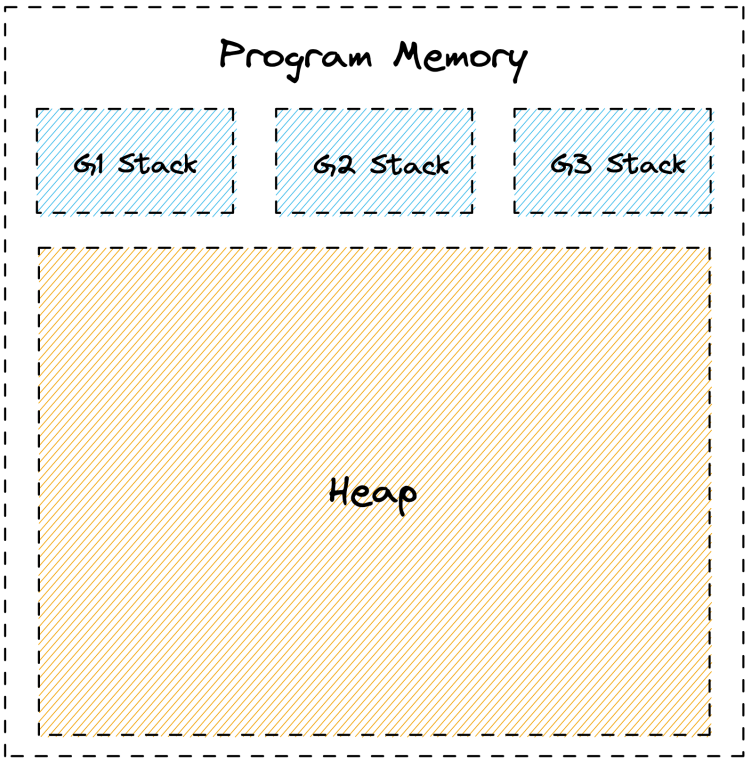


Garbage Collector

Другая важная абстракция в Go - это сборщик мусора. В таких языках, как C, программисту приходится вручную заниматься аллокацией и освобождением памяти с помощью `malloc()` и `free()`. Это дает большой контроль, но на практике оказывается очень подверженным ошибкам. Сборщик мусора сильно упрощает работу с памятью, но автоматическое управление может легко стать узким местом в производительности.

The Stack

Go может выделять память на стеке или в куче. Каждая горутина имеет свой собственный стек, который представляет собой непрерывную область памяти. Кроме того, существует большая область памяти, разделяемая между горутинами, которая называется кучей.



Когда вызывается функция, она получает свой собственный участок на стеке, называемый стековым фреймом, где она может размещать локальные переменные. Указатель стека используется для определения следующего свободного места в кадре. Когда функция возвращается, данные из последнего кадра удаляются простым перемещением указателя стека в конец предыдущего кадра. Сами данные кадра могут оставаться в стеке и перезаписываться при следующем вызове функции. Это очень просто и эффективно, поскольку Go не нужно следить за каждой переменной.

```
func main() {
    sum := 0
    sum = add(23, 42)
    fmt.Println(sum)
}

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

У нас есть функция `main()`, которая начинает работу с резервирования места в стеке для переменной `sum`. Когда вызывается функция `add()`, она получает свой собственный фрейм для хранения локальных параметров `a` и `b`. Когда функция `add()` возвращается, ее данные отбрасываются путем перемещения указателя стека обратно в конец фрейма функции `main()`, а переменная `sum` обновляется результатом. Тем временем старые значения `add()` остаются за пределами указателя стека, чтобы быть перезаписанными при следующем вызове функции.

Приведенный выше пример сильно упрощен и опускает многие детали, связанные с возвращаемыми значениями, указателями кадров, адресами возврата и инлайнингом функций. На самом деле, начиная с Go 1.17, приведенной выше программе даже не нужно место в стеке, поскольку небольшой объем данных может быть обработан компилятором с помощью регистров процессора.

На этом этапе вы можете задаться вопросом, что произойдет, если закончится место на стеке. В языках вроде C это привело бы к ошибке переполнения стека. Go же автоматически решает эту проблему, создавая копию стека, которая в два раза больше. Это позволяет горутинам начинать работу с очень маленьким стеком, обычно *2 килобайта*, и является важным свойством, делающее горутины более масштабируемыми, чем потоки операционной системы.

The Heap

Выделение памяти на стеке(Stack allocations) - это здорово, но есть много ситуаций, когда Go не может его использовать. Самая распространенная из них - возврат указателя на локальную переменную функции. Это можно увидеть в модифицированной версии нашего примера `add()`:

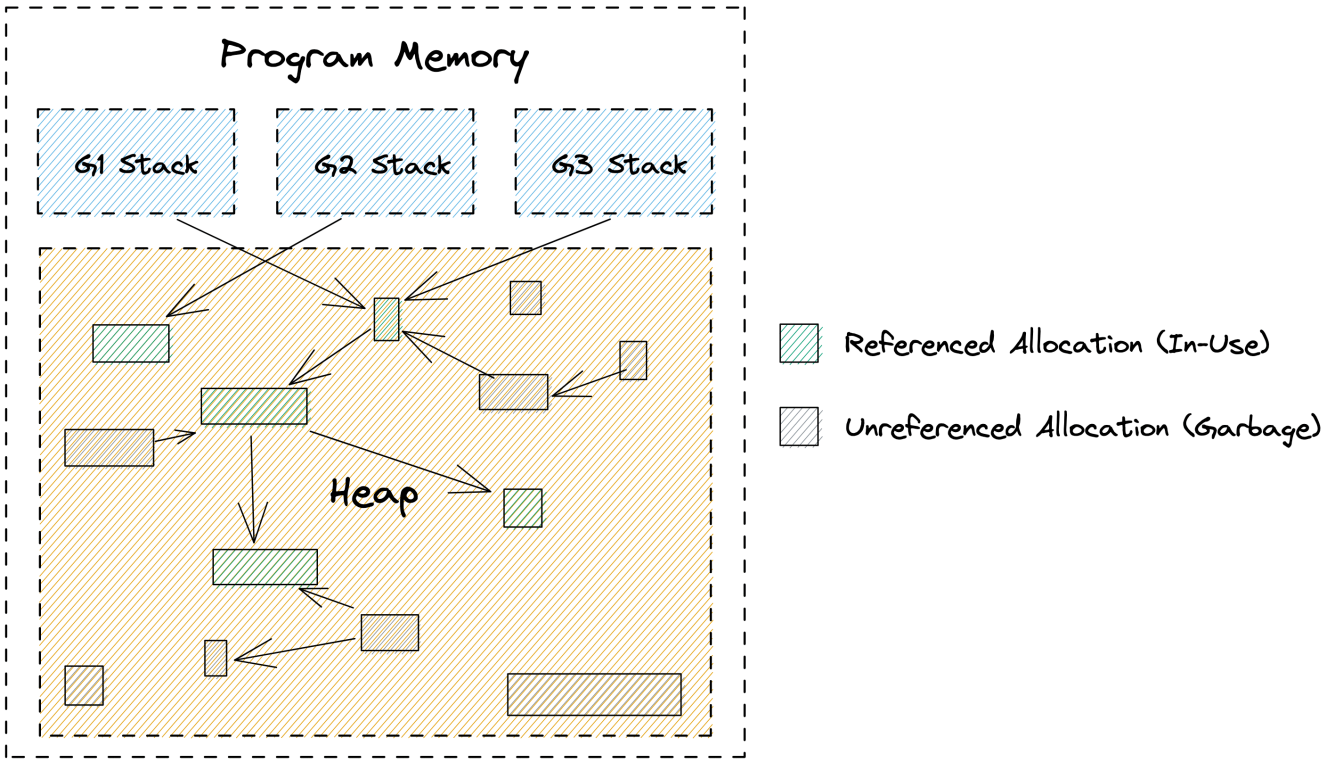
```go
func main() {
    fmt.Println(*add(23, 42))
}

func add(a, b int) *int {
    sum := a + b
    return &sum
}
```

Обычно в Go переменная `sum` внутри функции `add()` размещается на стеке. Но, как мы уже выяснили, эти данные удаляются при возврате функции `add()`. Поэтому, чтобы безопасно вернуть указатель `&sum`, Go должен выделить для него память за пределами стека. И здесь на помощь приходит куча.

Куча используется для хранения данных, которые могут пережить создающую их функцию, а также для любых данных, разделяемых между горутинами с помощью указателей. Однако в связи с этим возникает вопрос о том, как эта память освобождается. Ведь в отличие от данных на стеке, данные на куче не могут быть отброшены после возвращения создавшей их функции.

Go решает эту проблему с помощью встроенного сборщика мусора.

1



Выполнение GC связано с большим количеством дорогостоящих обходов графа и переполнения кэша. Он даже требует регулярных фаз stop-the-world, которые останавливают выполнение всей вашей программы. К счастью, в последних версиях Go это удалось сократить до долей миллисекунды, но большая часть оставшихся накладных расходов присуща любому GC. На самом деле, нередко 20-30% времени выполнения программы на Go уходит на управление памятью.

Вообще говоря, стоимость GC пропорциональна количеству аллокаций на куче, которые выполняет ваша программа. Поэтому, когда речь заходит об оптимизации накладных расходов вашей программы, связанных с памятью, мантра такова:

- **Уменьшить(Reduce)**: Постарайтесь превратить аллокации на кучи в аллокации на стека или вовсе избежать их. Минимизация количества указателей на куче также помогает.
- **Повторное использование(Reuse)**: Повторно используйте память на куче.
- **Переработка(Recycle)**: Некоторых аллокаций в куче не избежать. Позвольте GC переработать их и сосредоточиться на других проблемах.

Комментарии в Telegram-группе!

go perf gc

« ПРЕДЫДУЩАЯ
Двенадцать Добродетелей Рационалиста

СЛЕДУЮЩАЯ »
Диспетчеризация интерфейса