Снижаем нагрузку на GC ускоряя работу с памятью

Aleksandr Ivanov
alexandr.ivanov@yadro.com

О себе

Александр Иванов

Старший инженер-программист, Тимлид

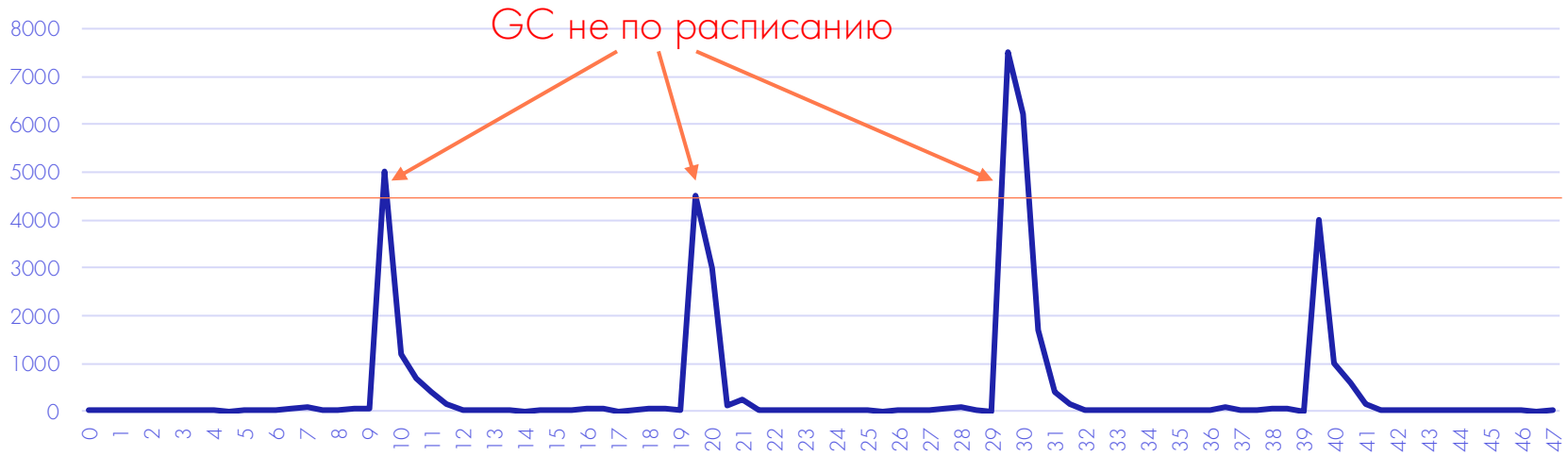
ООО «ЯДРО ЦЕНТР ТЕХНОЛОГИЙ МОБИЛЬНОЙ СВЯЗИ»

Более 20-ти лет программировал мультимедиа на C/C++ под разные платформы и операционные системы.

2 года назад перешёл на Go, так как мне поручили ускорить написанное на Go приложение и мне показалось, что я справлюсь.

Какую проблему решаем

- Характерной чертой ускоряемого приложения были редкие пиковые вычислительные нагрузки требующие память при получении пачки новых данных, которые нужно было очень быстро обработать чтобы опередить конкурентов работающих с теми же данными.



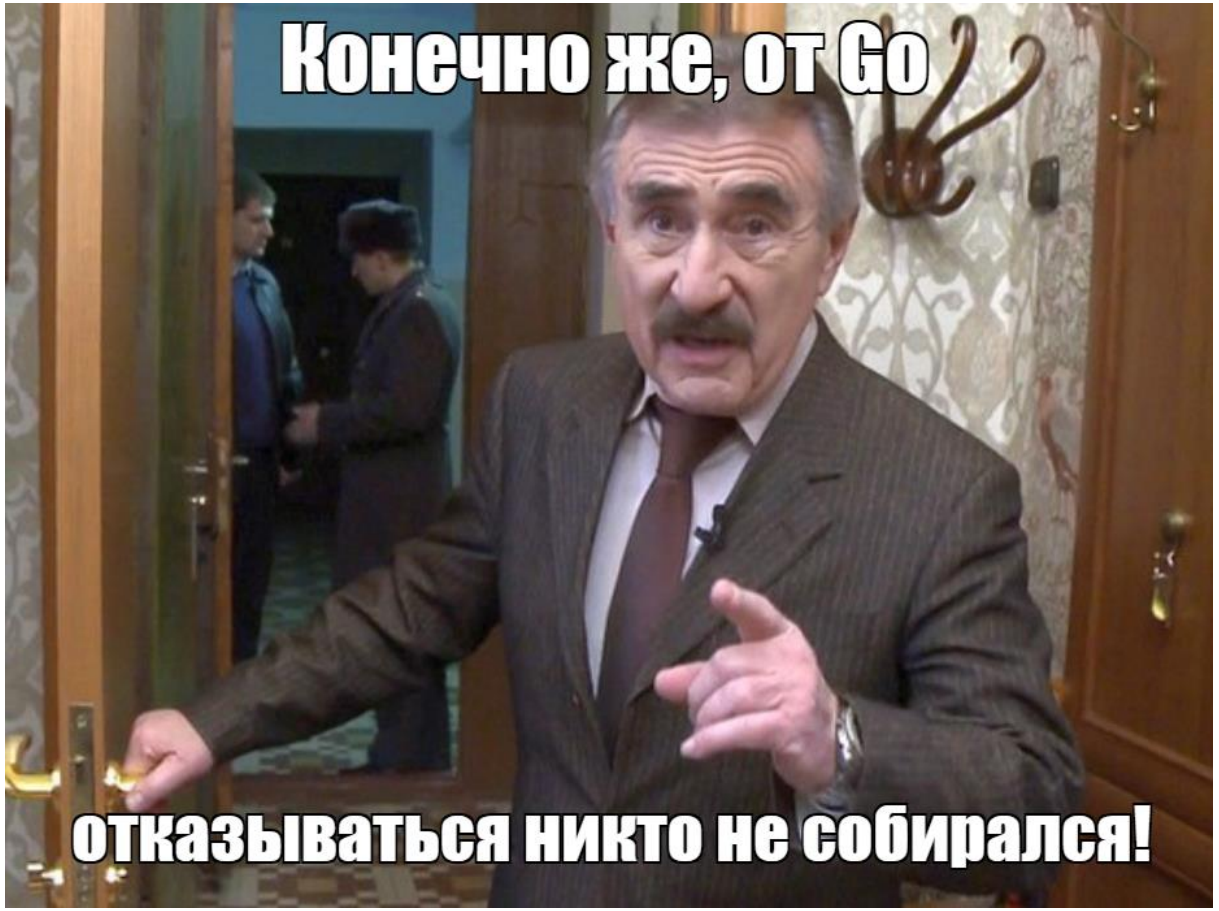
Какую проблему решаем

Профилирование показало, что дело было не в какой-то медленной time-critical функции, а в том, что в кульминационный момент из-за нехватки памяти запускался Garbage Collector и на время сервисных работ по очистке памяти приложение замирало.

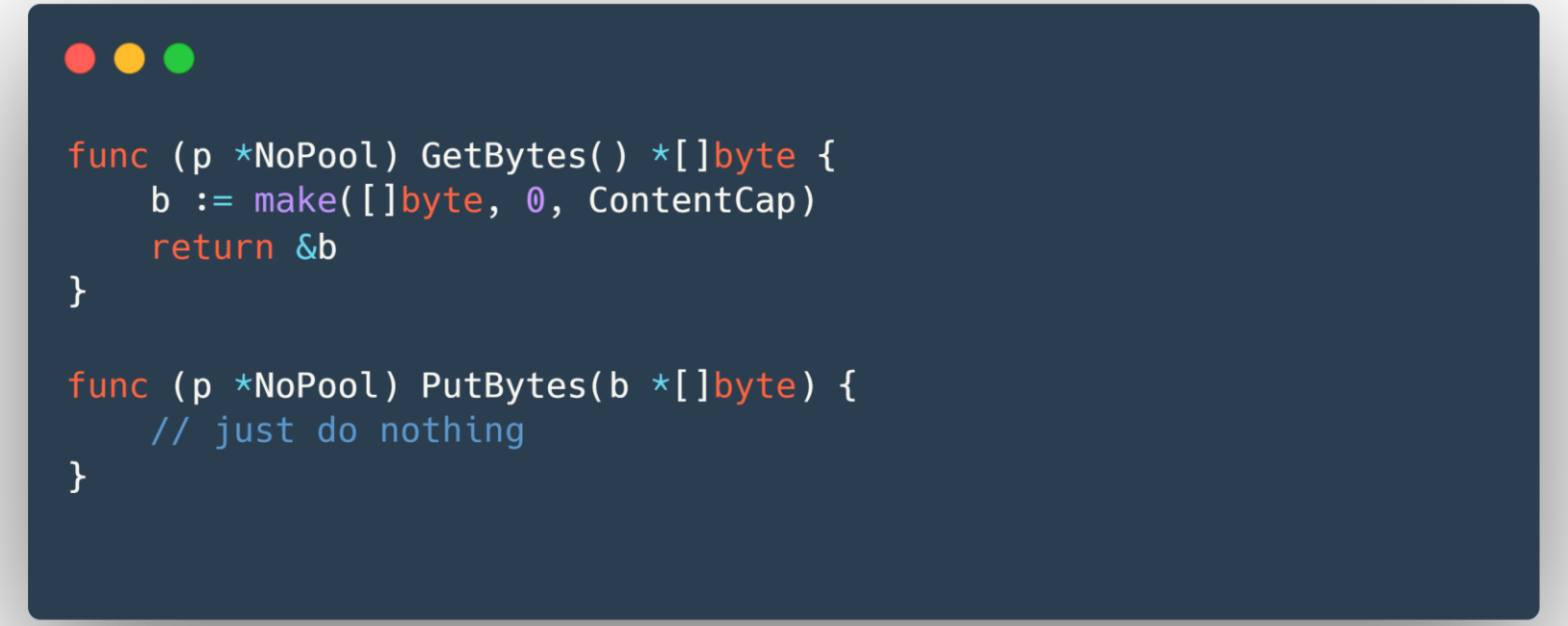
Картинка взята тут: https://weaviate.io/blog/gomemlimit-a-game-changer-for-high-memory-applications

Какую проблему решаем

Я попытался дать совет руководству проекта, но



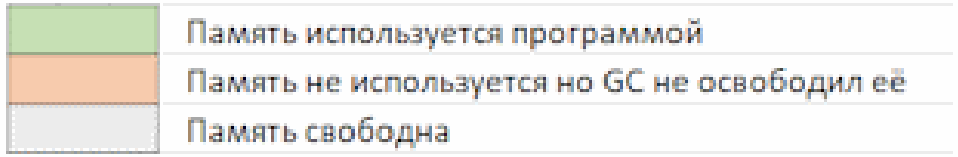
Без использования оптимизации



```go
func (p *NoPool) GetBytes() *[]byte {
    b := make([]byte, 0, ContentCap)
    return &b
}

func (p *NoPool) PutBytes(b *[]byte) {
    // just do nothing
}
```

Без использования оптимизации

Анимация работы с памятью без оптимизаций



Без использования оптимизации

Самый медленный и слабо предсказуемый с точки зрения равномерности работы программы случай, это когда память выделяется большое количество раз за короткий отрезок времени.

Без оптимизации со стороны разработчика:

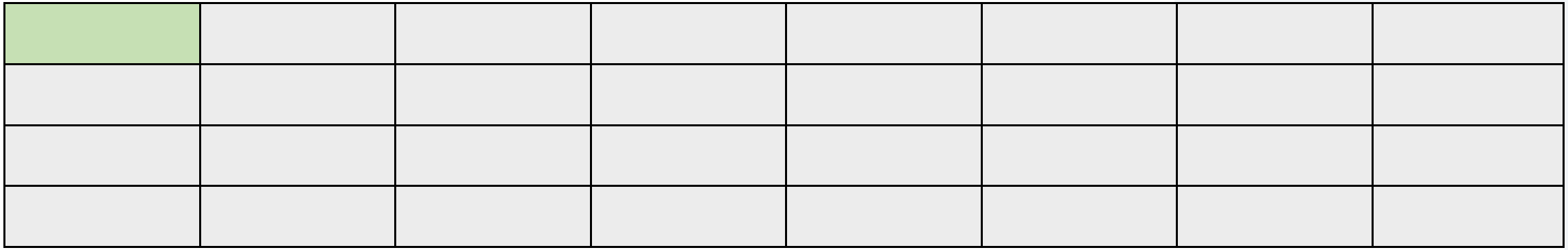
1. Память выделяется всегда новая и тратится время на поиск и выделение нужного объема непрерывной памяти
2. В моменты пиковой нагрузки GC прерывает программу чаще, так как происходит это не по расписанию, а при превышении порога занятой памяти.

Pool используя chain

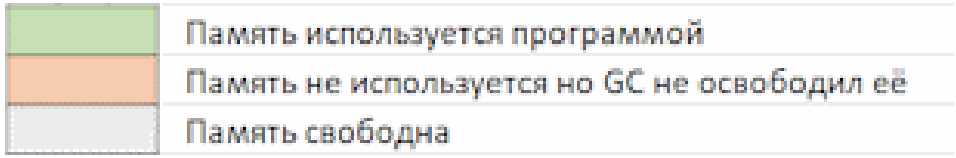
```go
var chanPool = make(chan *[]byte, BuffCount)

func (p *ChanPool) GetBytes() *[]byte {
    select {
    case b, ok := <-chanPool:
        if ok { return b }
    default:
    }
    b := make([]byte, 0, ContentCap)
    return &b
}

func (p *ChanPool) PutBytes(b *[]byte) {
    *b = (*b)[:0]
    chanPool <- b
    return
}
```

- Если известен max количества буферов
- Лёгок в реализации
- Понятен в реализации
- GC приходит только после окончания использования

Pool используя chain



Анимация работы с памятью используя chain based memory pool



Pool используя chain

То есть пул памяти на базе Go-каналов, это искомое решение, когда память выделяется и затем большое количество раз переиспользуется?

Он требует минимальных усилий со стороны разработчика, но универсальным случаем сложно считать, так как он накладывает определённые условия:

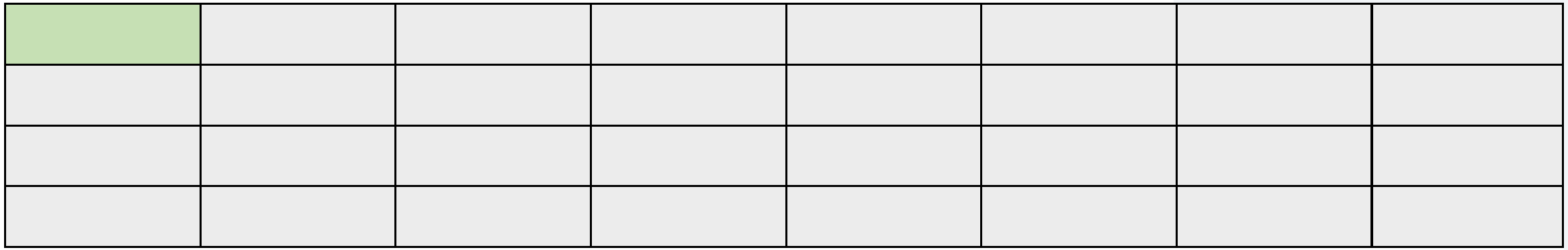
1. Желательно заранее угадать, какого количества буферов нам будет достаточно для работы (такие случаи однозначно существуют)
2. Стараться не возвращать в пул сильно разросшиеся буфера памяти так как они могут привести к неэффективному расходованию памяти.

Pool используя sync.Pool

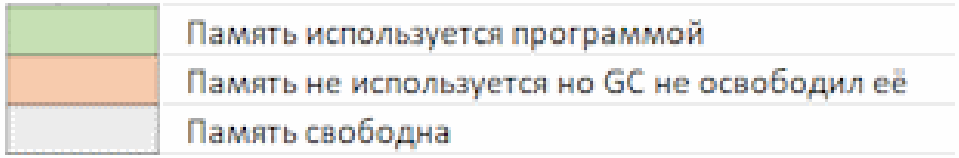
```go
var heapBuffersPool = &sync.Pool{
    New: func() interface{} {
        b := make([]byte, 0, ContentCap)
        return &b
    },
}

func (p *SyncPool) GetBytes() *[]byte {
    return heapBuffersPool.Get().(*[]byte)
}

func (p *SyncPool) PutBytes(b *[]byte) {
    *b = (*b)[:0]
    heapBuffersPool.Put(b)
}
```


- Нет ограничения по возможному количеству буферов
- GC приходит иногда
- Быстрый за счёт уменьшения распределения памяти
- Быстрый за счёт разгрузки GC

Pool используя sync.Pool



Анимация работы с памятью используя sync.Pool



Pool используя sync.Pool

Ещё один из возможных вариантов, пул памяти реализованный с помощью sync.Pool:

1. Нет ограничения по количеству буферов в пуле, не требуется предугадывать
2. Стараться не возвращать в пул сильно разросшиеся буфера памяти так как они могут привести к неэффективному расходованию памяти
3. Большие буфера, которые не вернули в пул будут подчищены GC
4. GC почистит так же и буфера, которые в пуле, но не используются на момент запуска GC

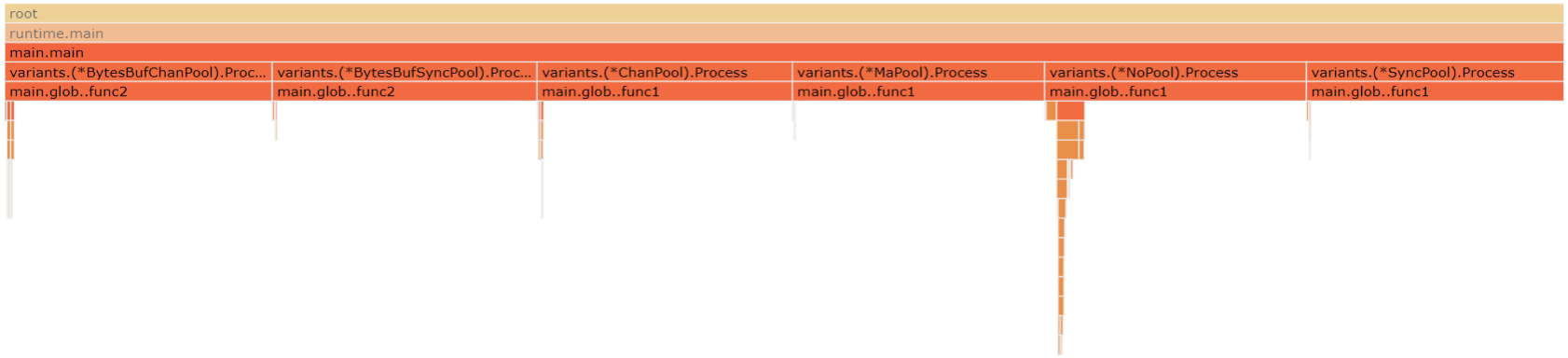
Pool используя memory arena – эксперимент

```
var arenaBuffersPool = &sync.Pool{
    New: func() interface{} {
        b := arena.MakeSlice[byte](a, 0, ContentCap)
        return &b
    },
}

func (p *ArenaPool) GetBytes() *[]byte {
    return arenaBuffersPool.Get().(*[]byte)
}

func (p *ArenaPool) PutBytes(b *[]byte) {
    *b = (*b)[:0]
    arenaBuffersPool.Put(b)
}
```

- maSyncPool
- Тоже быстро
- Но размер буфера лучше фиксированный

Benchmarks:

```
goos: linux
goarch: amd64
cpu: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz

Benchmark_NoPool
Benchmark_NoPool-8            31      33464646 ns/op    156000046 B/op    200000 allocs/op
Benchmark_ChanPool
Benchmark_ChanPool-8         236       5134540 ns/op           0 B/op         0 allocs/op
Benchmark_SyncPool
Benchmark_SyncPool-8         298       4017713 ns/op           3 B/op         0 allocs/op
Benchmark_MaPool
Benchmark_MaPool-8           330       3781410 ns/op           3 B/op         0 allocs/op
```



- Замеры производительности показывают улучшение на порядок по сравнению с первоначальным вариантом без оптимизации

- Flame-граф подтверждает, что при использовании sync.Pool память используется наиболее оптимально

Нюансы

1. GC чистит буфера, которые в sync.Pool'е, но не используются на момент запуска GC
2. Неконтролируемый рост размера буферов в конечном итоге может привести к неэффективному использованию памяти
3. Возможна утечка памяти если хранить не указатели на изменяемые объекты, а их копии
4. Буфер можно вернуть даже если отдали его далеко, нужно просто завернуть в структуру ещё и указатель на пул
5. Go-рантайм часто меняется, следовательно, с каждой новой версией Go проверять выбранный подход бенчмарками

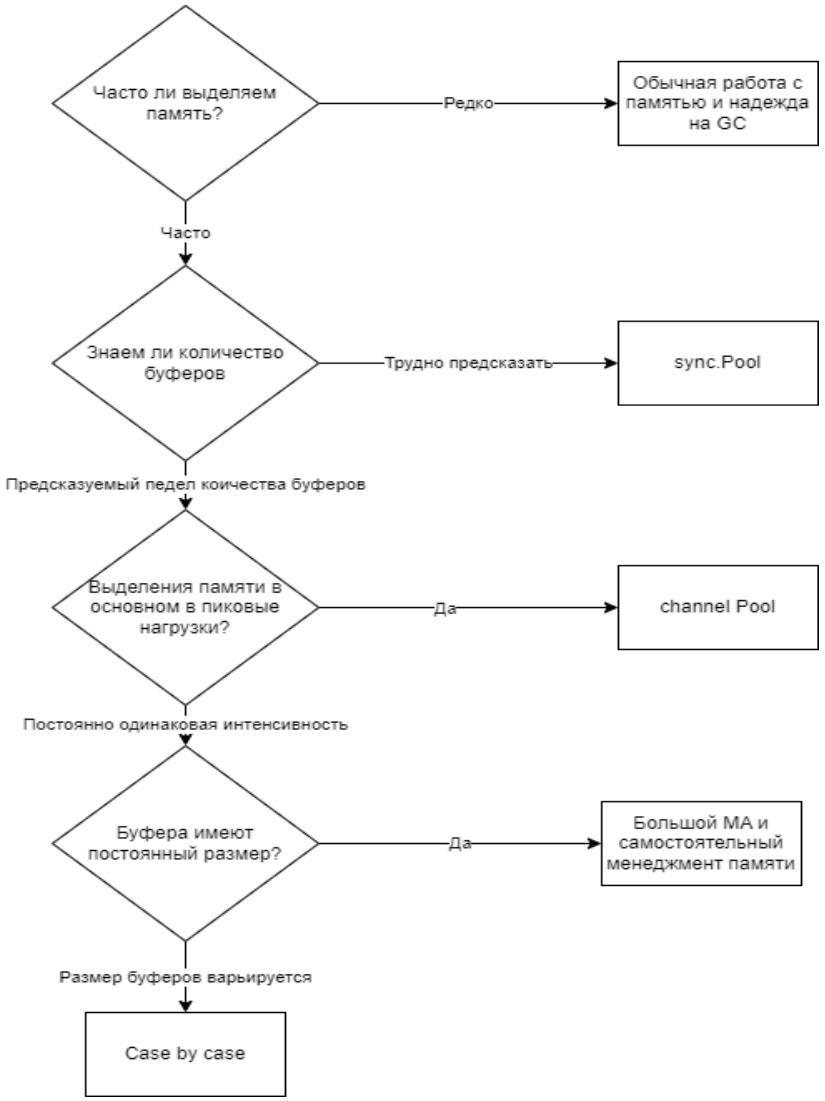
Как вернуть память в pool

```go
func (p *ChainPool) PutBytes(b *[]byte) {
    if cap(*b) > ContentCap {
        return
    }

    *b = (*b)[:0]
    arenaBuffersPool.Put(b)
}

func (p *SyncPool) PutBytes(b *[]byte) {
    if cap(*b) > ContentCap {
        return
    }

    *b = (*b)[:0]
    arenaBuffersPool.Put(b)
}
```

Не возвращать в пул большие буфера, если в процессе эксплуатации они слишком разрослись

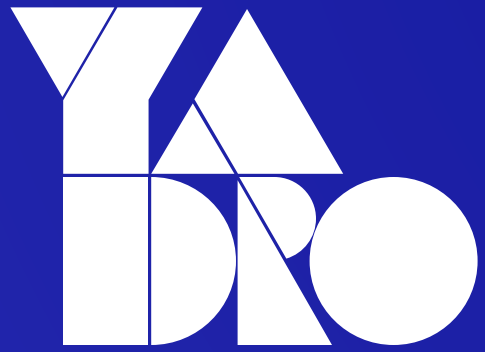
Например в пакете fmt буфера больше чем 64 килобайта не вернутся в пул и позже будут освобождены GC

Выводы


Часто ли выделяем память?
Редко
Обычная работа с памятью и надежда на GC
Часто
Знаем ли количество буферов
Трудно предсказать
sync.Pool
Предсказуемый педел коичества буферов
Выделения памяти в основном в пиковые нагрузки?
Да
channel Pool
Постоянно одинаковая интенсивность
Буфера имеют постоянный размер?
Да
Большой MA и самостоятельный менеджмент памяти
Размер буферов варьируется
Case by case

Полезные ссылки

- Сайт GoFunc gofunc.ru
- Блог компании YADRO на Хабре habr.com/ru/companies/yadro/articles
- Репозиторий с кодом gitflic.ru/project/onealexanderivanov/memory-pools
- Форматирование кода для презентаций carbon.now.sh
- Визуализация concurrency в Go habr.com/ru/articles/276255
- Объяснение работы GC habr.com/ru/companies/avito/articles/753244
- Ещё объяснение weaviate.io/blog/gomemlimit-a-game-changer-for-high-memory-applications
- Исчерпывающий гайд по GC go.dev/doc/gc-guide
- Use sync.Pool in Golang to reduce GC pressure www.sobyte.net/post/2022-06/go-sync-pool



Москва,
ул. Рочдельская, 15, стр. 13
+7 800 777-06-11

yadro.com