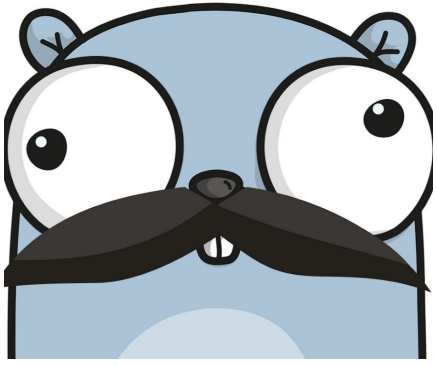
Оптимизации в разработке ПО



Автор: Денис Лимарев

Введение

Версия Go - 1.17.3

Архитектура - darwin/amd64

Процессор: 2,3 GHz 8-ядерный процессор Intel Core i9

Память: 16 ГБ DDR4

ОС: macOS Catalina 10.15.7

GOMAXPROCS: 16, GOGC=100

Бенчмарки: https://github.com/peakle/meetup/tree/master/go-optimizations

Оптимизации без доказательств

- Правильное использование make
- Избегайте defer в цикле
- Отправка запросов без контекста
- Использование буферизированных каналов
- Использование буферов записи (strings.Builder, bytes.Buffer, bufio)
- Компиляция регулярок в init функции

Оптимизации с бенчмарками

- Итерация по массиву
- Итерация по элементам массива/слайса по индексу
- Переиспользование горутин
- Использование sync.Pool
- Использование atomic пакета

Итерация по массиву

Дано: массив, элементы массива структура размером 96 байт

Необходимо: проитерироваться по элементам и сделать какие-то вычисления

```go
for _, v := range hugeArray {
    sum += v.h
}
```

```go
for _, v := range &hugeArray {
    sum += v.h
}
```

Данные для структуры размером 96 байт

Размер массива	Копирование массива (ns/op)	Массив по указателю (ns/op)	Разница
64	246	172	143,02%
128	472	344	137,21%
256	1145	662	172,96%
512	2396	1411	169,81%
1024	4743	3021	157,00%
2048	10390	5740	181,01%

Данные для структуры размером ~2кб

Размер массива	Копирование массива (ns/op)	Массив по указателю (ns/op)	Разница
64	5370	2006	267,70%
128	12996	4069	319,39%
256	26459	10788	245,26%
512	54588	23956	227,87%
1024	112733	44393	253,94%
2048	240037	93362	257,10%

Итоги: итерация по массиву

Как проверку автоматизировать: линтер go-critic

Ссылка на проблему: https://github.com/golang/go/issues/15812

Вывод: При итерации по массиву большого размера, лучше использовать указатель на массив, чтобы избежать его копирования

Итерация по элементам слайса

Дано: слайс, где элементы структура размером 96 байт

Необходимо: проитерироваться по элементам и сделать какие-то вычисления

```
myStruct struct {
    h     uint64
    cache [64]byte
    body  []byte
}
```

Версия с копированием структуры

```
for _, hs := range hugeSlice {
    sum += hs.h
}
```

Версии без копирование структуры

```
for ii := range hugeSlice {
    sum += hugeSlice[ii].h
}
```

```
for ii := range hugeSlice {
    sum = (&hugeSlice[ii]).h
}
```

Данные для структуры размером 96 байт

Количество элементов	По значению (ns/op)	По индексу (ns/op)	Разница
1000	3143	1354	232,13%
10000	32117	13642	235,43%
100000	473056	205053	230,70%
1000000	7777976	6050619	128,55%

Данные для структуры размером ~2кб

Количество элементов	По значению (ns/op)	По индексу (ns/op)	Разница
1000	44208	1284	3442,99%
10000	1386485	20471	6772,92%
100000	17403605	280727	6199,48%
1000000	175878184	14379364	1223,13%

Итоги: итерация по элементам слайса

Как проверку автоматизировать: линтер go-critic

Вывод: при итерации по слайсу эффективнее использовать индексы вместо копирования элементов в цикле.

Переиспользование горутин

Дано: большое количество поступающих задач, которые нужно выполнять параллельно

Необходимо: ограничить параллельность процесса, снизить затраты на создание горутин, уменьшить нагрузку на GC

Актуальность: серверы, отложенная обработка задач, ограниченные по памяти задачи

Пример сырой горутины

```
for j := 0; j < b.N; j++ {
    go func(num int64) {
        atomic.AddInt64(&counter, dummyProcess(num))
        wg.Done()
    }(int64(j))
}
```

Пример горутины с семафорой

```
for j := 0; j < b.N; j++ {
    sema <- struct{}{}
    go func(num int64) {
        atomic.AddInt64(&counter, dummyProcess(num))
        <-sema
        wg.Done()
    }(int64(j))
}
```

Пример переиспользуемой горутины

```
go func() {
    select {
    case task := <-ch:
        processFunc(task)
    case <-shutDownCtx.Done():
        return
    case <-timer.C:
        return
    }
}()
```

Данные для миллиона задач и размера пула 500к

Название	Время на операцию (ns/op)	Память (Мб)	Циклов GC
Сырые	270	126	44
Семафоры	473	82	31
Переиспользуемые	679	46	9

Примечание: количество сырых горутин = количеству тасок = 1 миллион

Итоги: переиспользование горутин

Библиотеки:

- Реализация в fasthttp сервере
- Реализация от Delivery Club
- Реализация в свободной библиотеке

Вывод: переиспользование горутин может помочь:

- сократить время на создание новых горутин
- уменьшить нагрузку на GC, сократить размер кучи
- ограничить параллельность без значительной потери производительности

Использование sync.Pool

Дано: нагруженный процесс обработки с частым создание объектов структур

Необходимо: оптимизировать создание громоздких объектов, убрать нагрузку на GC

Пример обычного создания объекта

```
go func() {
    h = &hugeStruct{body: make([]byte, 0, mediumArraySize)}
    h = dummyPointer(h)
    wg.Done()
}()
```

Пример создания через sync.Pool

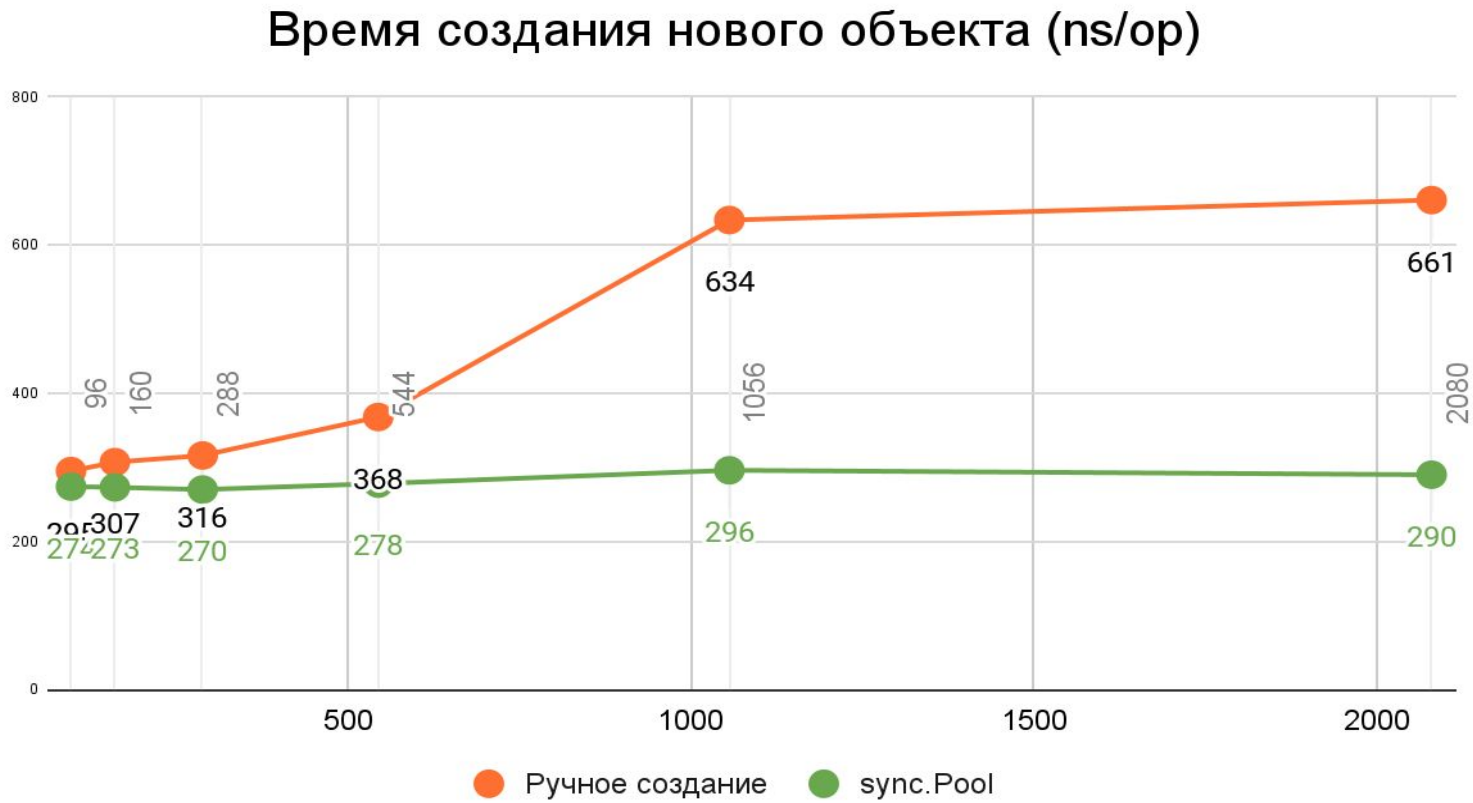
```
go func() {
    h = get()
    h = dummyPointer(h)
    wg.Done()

    put(h)
}()
```

Без sync.Pool

Размер структуры (байты)	Память (МБ)	Циклов GC	Скорость (ns/op)
96	6028	1881	295
160	6639	2102	307
288	7859	2491	316
544	10606	3363	368
1056	16105	5344	634
2080	27090	8723	661

С использованием sync.Pool

Размер структуры (байты)	Память (МБ)	Циклов GC	Скорость (ns/op)
96	235	67	274
160	234	79	273
288	235	67	270
544	234	70	278
1056	236	69	296
2080	238	71	290



Примечание: по оси X размер структуры

Итоги: использование sync.Pool

Плюсы	Минусы
Быстрее создание объектов	Синтаксически менее удобно
Меньше нагрузка на GC от новых объектов	Плохо подходит для короткоживущих объектов
Отлично подходит для переиспользования слайсов	Необходимо занулять объект перед возвращением
	Выше порог входа, можно исчерпать память

Где используется: fmt, fasthttp

Вывод: при частой необходимости создания объектов структур в нагруженных процессах, возможна оптимизация за счет переиспользования отработанных объектов

Использование atomic операций

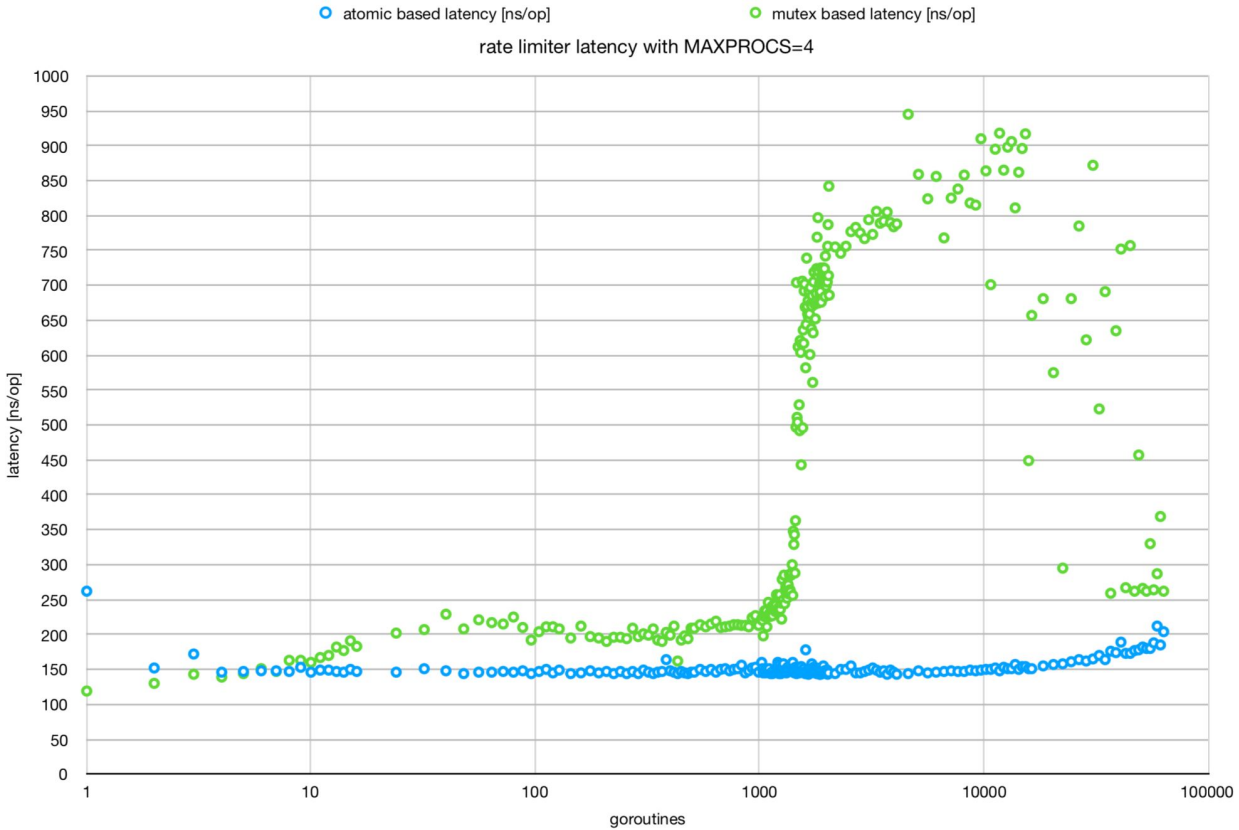
Дано: общедоступная переменная

Необходимо: эффективный алгоритм эксклюзивного доступа к данным на короткое время (> 1 ms)

```
mutexCounter := func(i int64) int64 {
    mu.Lock()
    newCounter := i * counter.c
    counter.c = newCounter
    mu.Unlock()

    return newCounter
}
```

```go
atomicCounter := func(i int64) int64 {
    var taken bool
    var newCounter int64

    for !taken {
        oldCounter := atomic.LoadInt64(&counter.c)
        newCounter = i * oldCounter

        taken = atomic.CompareAndSwapInt64(&counter.c, oldCounter, newCounter)
    }
    return newCounter
}
```

Пример ratelimit



Почему так происходит или режимы работы мьютексов

Обычный режим:

- горутины становятся в fifo очередь
- первая в очереди горутина не владеет мьютексом после разблокировки
- новые прибывшие горутины имеют большой шанс захватить мьютекс

Режим голодания:

- мьютекс разблокирует первая ожидающая горутина в очереди
- новые горутины не пытаются захватить мьютекс даже если это возможно, становятся в очередь на ожидание
- нужен для возможного предотвращения задержки хвоста

Итоги: использование atomic операций

Примеры использования:

- ограничитель запросов от uber
- sync пакет
- пул воркеров
- стандартная библиотека
- прикладные задачи доступа к разделяемым переменным

Вывод: при умелом использовании CAS и других атомарных операций пакета atomic можно добиться выигрыша в производительности за счет использования низкоуровневых операций.

Разобранные оптимизации

- Итерация по элементам массива/слайса по индексу
- Итерация по указателю массива
- Переиспользование горутин
- Использование sync.Pool
- Использование atomic операций

Список литературы

- Введение в профилирование/бенчмарки Go [GO]
- Лучшие практики в fasthttp [GO]
- Линтер gocritic [GO]
- Введение в escape analysis [GO]
- Escape analysis bytes.Buffer [GO]
- Продвинутый профайлер pprof++ [GO]
- Эффективное использование структур [GO]
- Балласт памяти [GO]
- Возможные проблемы с GC [GO]
- Популярные линтеры [GO]
- Популярные советы по производительности [GO]
- Подборка оптимизаций [GO]
- Оптимизации с учетом процессора [Доступ к памяти]
- Локальность данных [Доступ к памяти]
- Неожиданные причины торможения программ [Доступ к памяти]
- История оптимизации FizzBuzz задачи [Доступ к памяти]
- Иерархия памяти [Доступ к памяти]
- Branch prediction [Доступ к памяти]

Спасибо за внимание

Список благодарностей

- Вячеслав Валявский
- Максим Поташев
- Константин Сергеев
- Алексей Смирнов
- Александр Романов
- Искандер Шарипов
- Юлия Ковалева
- Елизавета Грейм