Golang в действии: Как нам удается писать highload приложение на (не?)подходящем языке

Даниил Подольский
CTO inCaller.org

inCaller

О чем этот доклад

- Highload - довольно “мутный” термин
  - Кое кто даже утверждает, что его изобрел Олег Бунин самолично
- Дмитрий Завалишин определяет highload как “упереться во все ограничения сразу”
  - И в этой формулировке термин имеет смысл, но довольно неглубокий

inCaller

О чем этот доклад

- с точки зрения автора доклада для highload характерны три вещи
  - недостаток ресурсов, требующий производить оптимизации
  - недостаток ресурсов, требующий производить горизонтальное масштабирование
  - высокая сложность проекта
    - простые проекты просто масштабируются и просто оптимизируются

inCaller

О чем этот доклад

- вышеперечисленное означает, что highload проект дорогой
  - у вас дорогая инфраструктура
  - у вас дорогие специалисты
  - у вас дорогие простои и ошибки

О чем этот доклад

- таким образом, язык, однозначно подходящий для создания highload проекта должен обладать следующими свойствами:
  - обеспечивать разумный уровень потребления ресурсов
    - и контроль над потреблением!
  - обеспечивать разумный уровень загрузки мозгов
  - предоставлять средства анализа проблем в процессе эксплуатации
    - debuger не подойдет
  - обеспечивать масштабируемость

inCaller

Уничтожим интригу

- С нашей точки зрения Go этим требованиям соответствует
- Но могло бы быть и лучше
  - Особенно сильно могло бы быть лучше у нас в голове
- Именно мы своими неумелыми действиями часто выпячиваем недостатки Go как языка для highload и не пользуемся его достоинствами
- Но некоторые вещи должны нам исправить создатели языка
  - Мы требуем этого!

inCaller

Что плохо в Go как в языке
Питонизмы

- Не знаю, чей это термин, но он довольно точен, и означает:
  “попытка использования паттернов языка с динамической типизацией в языке со статической типизацией”
- Мы успели натащить в проект довольно много этой дряни, прежде чем одумались
- Попробовали бы мы сделать это в Java!

inCaller

Что плохо в Go как в языке
Питонизмы: unmarshaling

- Сервер inCaller обменивается с клиентом сообщения
- Сообщения сериализованные - требуется десериализация
- Есть соблазн сделать универсальную структуру, в которую десереализуются сообщения всех типов

inCaller

Что плохо в Go как в языке
Питонизмы: unmarshaling

- Не используйте универсальную структуру в своем коде!
  - Копируйте значения в структуру конкретного типа
  - Ну или проводите unmarshaling в два этапа, если ваш десериализатор это позволяет

inCaller

Что плохо в Go как в языке
Питонизмы: marshaling

- Marshaler игнорирует приватные поля в структурах
- А публичные поля нет способа выставить правильно в соответствии с типом

inCaller

Что плохо в Go как в языке
Питонизмы: marshaling

- Используйте кастомный Marshaler
- Да, для каждого типа свой кастомный Marshaler

inCaller

Что плохо в Go как в языке
Питонизмы: строки вместо переменных

- Для метрик мы используем Prometheus
- Репорт метрики в Прометее выглядит примерно так:

```
SomeCounterVec.WithLabelValues(
   “someLabel”, “anotherLabel”,
).Add(1)
```

inCaller

Что плохо в Go как в языке
Питонизмы: строки вместо переменных

```
type CounterVec struct {
    vec prometheus.CounterVec
    lvs []string{
        “someLabel”, “anotherLabel”,
    }
}
func (m *CounterVec) Add(v float64) {
    m.vec.WithLabelValues(m.lvs…).Add(v)
}
```

inCaller

Что плохо в Go как в языке
Питонизмы: []interface{}

- Есть соблазн сделать код универсальным

```
func SomeFunc(s []interface{}) {
    ss := s.([]string)
    …
}
```

- Не работает!

inCaller

Что плохо в Go как в языке
Питонизмы: []interface{}

```
func SomeFunc(s []interface{}) {
    ss := make([]string, len(s))
    for i, str := range s {
        ss[i] = str.(string)
    }
    …
}
```

Что плохо в Go как в языке
error processing boilerplate

```
func SomeFunc(s []interface{}) error {
    ss := make([]string, len(s))
    for i, str := range s {
        ss[i], err = str.(string)
        if err != nil {
            return err
        }
    }
    …
}
```

inCaller

Что плохо в Go как в языке
error processing boilerplate

У Гоферов вырабатывается избирательное зрение

```
func SomeFunc(s []interface{}) error {
    ss := make([]string, len(s))
    for i, str := range s {
        ss[i], err = str.(string)
        if err != nil {
            return err
        }
    }
    …
}
```

inCaller

Что плохо в Go как в языке
error processing boilerplate

- Живите с этим
  - Ну или, как мы, держите в команде человека с незамутненным взглядом

inCaller

Что плохо в Go как в языке no generics

- `[]interface{}` - это как раз от отсутствия генериков
  - Всем лень копипастить и файдреплейсить
- Есть кододогенерация
- Но мы ей не пользуемся
  - Неудачный первый опыт – и компайлер, и рантайм репортят ошибки не там, где они сделаны

inCaller

Что плохо в Go как в языке no exceptions

- Кстати, о error processing boilerplate
- Чем нам `panic()` не исключение?
  - Его можно поймать только на выходе из функции
  - `recover()` возвращает `interface{}`
  - И не возвращает stacktrace

Что плохо в Go как в языке no exceptions

- Попытка использовать `panic()` как exception приводит к такому усложнению кода, что лучше уж error processing boilerplate

Что плохо в Go как в языке recover() не возвращает stacktrace

```
type Error struct {
    err  error
    st   string
    msg  string
    line string
}
```

inCaller

Что плохо в Go как в языке recover() не возвращает stacktrace

```
type Error struct {
    err  error
    st   string
    msg  string
    line string
}
```

inCaller

Что плохо в Go как в языке recover() не возвращает stacktrace

```
func NewError(dbg bool, err error, msg string, params ...interface{}) *Error {
  st := ""
  if dbg {
    buf := make([]byte, stackBufSize)
    n := runtime.Stack(buf, false)
    st = string(buf[:n])
  }
  return &Error{
    line: GetCallerString(1),
    msg:  Ternary(len(params) > 0, fmt.Sprintf(msg, params...), msg).(string),
    err:  err,
    st:   st,
  }
}
```

inCaller

Что плохо в Go как в языке no __FILE__, no __LINE__

```
func GetCallerString(stackBack int) string {
	fileName, line, funcName := GetCaller(stackBack + 1)
	return fmt.Sprintf("%s:%d:%s", fileName, line, funcName)
}
func GetCaller(stackBack int) (string, int, string) {
	pc, file, line, ok := runtime.Caller(stackBack + 1)
	if !ok {
		return "UNKNOWN", 0, "UNKNOWN"
	}
	if li := strings.LastIndex(file, "/"); li > 0 {
		file = file[li+1:]
	}
	return file, line, runtime.FuncForPC(pc).Name()
}
```

inCaller

Что плохо в Go как в языке
no ternary operator

```
func Ternary(c bool, t interface{}, f interface{}) interface{} {
	if c {
		return t
	}
	return f
}
```

inCaller

Что плохо в Go как в языке мощный switch

Go's switch is more general than C's. The expressions need not be constants or even integers, the cases are evaluated top to bottom until a match is found, and if the switch has no expression it switches on true. It's therefore possible—and idiomatic—to write an if-else-if-else chain as a switch.

Что плохо в Go как в языке мощный switch

животные делятся на:

а) принадлежащих Императору,

б) набальзамированных,

в) прирученных,

г) молочных поросят,

д) сирен,

е) сказочных,

ж) бродячих собак,

з) включённых в эту классификацию,

и) бегающих как сумасшедшие,

к) бесчисленных,

л) нарисованных тончайшей кистью из верблюжьей шерсти,

м) прочих,

н) разбивших цветочную вазу,

о) похожих издали на мух.

Что плохо в Go runtime
SSL

- Медленный SSL handshake
  - 250rps против 500rps на nginx (читай OpenSSL)
- Жрущий память SSL
  - Примерно на 13KB на коннект в сранении с таким же, но без шифрования
  - Видимо, копия сертификата и ключа у каждого коннекта своя
- nginx – не выход: увеличивает потребление сокетов втрое
  - А сокеты – они дорогие

inCaller

Что плохо в Go runtime нереентерабельный RWMutex

- Делаем RLock()
- Определяем, что требуется update
- Делаем Lock()
- Perfect deadlock!

inCaller

Что плохо в Go runtime нереентерабельный RWMutex

- Делаем RLock()
- Определяем, что требуется update
- Отпускаем RUnlock()
- Делаем Lock()
- Определяем, что update все еще требуется
- Апдейтим
- Отпускаем Unlock()

inCaller

Что плохо в Go runtime
бесконтрольные goroutines

- Каждая gouroutine существует как отдельная сущность
  - Это ясно показывает stacktrace
- Но эта сущность недоступна программисту
  - Даже ID для записи в лог взять негде
  - Не говоря уже об имени

inCaller

Что плохо в Go runtime
бесконтрольные goroutines

- Мы передаем имя и id горутины параметрами в функцию
  - Иначе разобраться потом в логах невозможно
- А еще мы передаем туда `continue int`, который изображает `bool`, который апдейтим и читаем `atomic`
- Рекомендованный способ - передавать не int, а канал
- Но внутри у канала даже не atomic, а mutex

inCaller

Что плохо в Go runtime
каналы не имеют сигналинга

- Кстати о каналах - на них нет сигналов!
- Узнать о том, что канал закрыт, можно только почитав из него
- А если пописать в закрытый канал – будет panic
- Поэтому читатель не должен закрывать никогда – это должен делать писатель
- Но что если у нас один читатель и много писателей?

inCaller

Что плохо в Go runtime
каналы не имеют сигналинга

- Кстати о каналах - на них нет сигналов!
- Узнать о том, что канал закрыт, можно только почитав из него
- А если пописать в закрытый канал – будет panic
- Поэтому читатель не должен закрывать никогда – это должен делать писатель
- Но что если у нас один читатель и много писателей?

inCaller

Что плохо в Go runtime
каналы не имеют сигналинга

- Отдельный канал, из которого писатели читают в select
- Когда читатель завершается, он этот отдельный канал закрывает, и писатели узнают об этом, получив при чтении ошибку
- Если только они не заблокировались в записи в первый, полезный канал
- Поэтому писать надо тоже в select
- И это прямой путь в callback hell

inCaller

Что плохо в Go runtime
no drop privileges

- Обычно демон запускается под root, открывает все привилегированные ресурсы – например, порты меньше 1024 – и делает себе set uid, чтобы не работать из-под root.
- И у нас даже есть syscall.Setuid()
- Но он не работает, как минимум на linux
  - Потому, что к моменту, когда мы можем вызвать syscall.Setuid(), несколько threads уже запущены, и на них действие вызова распространиться не может

Что плохо в Go runtime
no drop privileges

- Запускаемся под root
- Открываем все нужные порты
- Получаем дескрипторы сокетов
- Запускаем себя же с помощью exec.Command, передав ему правильный uid и файловые дескрипторы сокетов в качестве параметров

inCaller

Так почему же Go?

- Статическая типизация. Она все же есть
- Более менее внятная обработка ошибок. Помните, чем все кончилось в Java?
- Компиляция. Она быстрая
- Сборка мусора. Она работает.
- Скорость. Go быстр, как ни странно.
- Green threads AKA gouroutines. Можно запускать их миллионами.

inCaller

Спасибо!
Вопросы?

inCaller