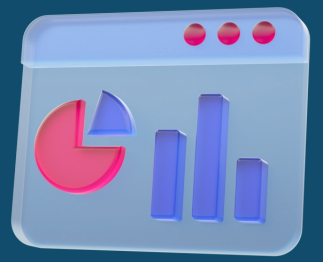
OZON tech

PGO: практика и маленькие хитрости использования

Макс Ривейро, ведущий разработчик в Ozon

Языковая платформа Go в Ozon

Библиотеки
на Go



Сборка проектов
на Go

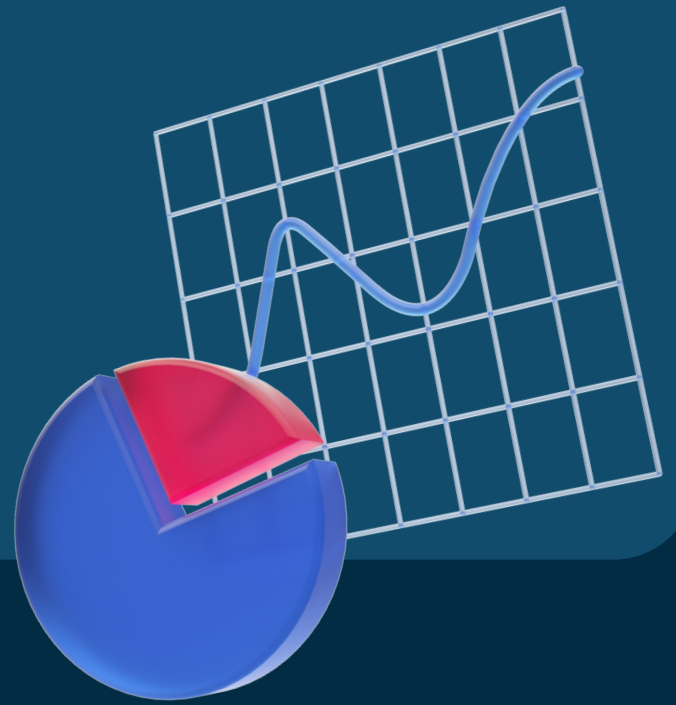
Поиск
узких мест

Ускорение
и оптимизация

А что такое PGO?

PGO

это Profile Guided Optimization,
т.е. оптимизация программы с использованием профилей (статистики) исполнения программы



Для кого этот доклад?

- Использовали PGO в Go?

Для кого этот доклад?

- Использовали PGO в Go?
- А в других языках?

Для кого этот доклад?

- Использовали PGO в Go?
- А в других языках?
- А кто занимался профилированием в Go?

Для кого этот доклад?

- Использовали PGO в Go?
- А в других языках?
- А кто занимался профилированием в Go?
- Бенчмарки в **_test.go**?

Для кого этот доклад?

- Использовали PGO в Go?
- А в других языках?
- А кто занимался профилированием в Go?
- Бенчмарки в **_test.go**?
- Вы точно знаете, как компилировать в Go!

Для кого этот доклад?

- Использовали PGO в Go?
- А в других языках?
- А кто занимался профилированием в Go?
- Бенчмарки в **_test.go**?
- Вы точно знаете, как компилировать в Go!

Значит, этот доклад для вас!

AGENDA

01 Что такое PGO?

02 Как использовать?

03 Несколько хитростей при использовании!

04 Какие результаты можно получить и так ли они значительны?

ГОРЬКАЯ ПРАВДА

PGO — не «серебряная пуля», а лишь ещё один инструмент, который вы можете использовать в своих проектах

Что мы можем получить?

«As of Go 1.22, benchmarks for a representative set of Go programs show that building with PGO improves performance by around 2-14%.»

https://go.dev/doc/pgo

А что внутри?

01

Трансформация CPU-профиля во взвешенный граф вызовов

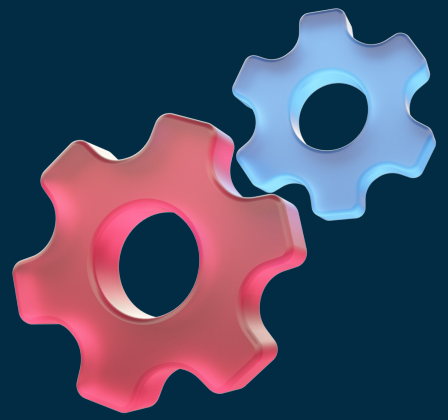
02

Корректировка бюджетов на inlining для «горячих» вызовов

03

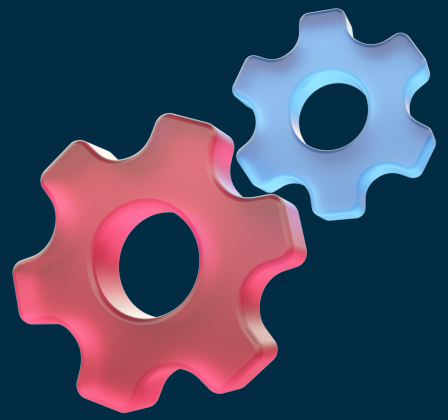
Непосредственно inlining

Коротко: что такое inlining?



Оптимизация, которая заключается в подстановке тела функции в место её вызова

Коротко: что такое inlining?



Оптимизация, которая заключается в подстановке тела функции в место её вызова

- Улучшение производительности
- Уменьшение накладных расходов на вызов
- Улучшение оптимизации
- Увеличение размера исполняемого файла

Примеры результатов оптимизации

- Применение PGO к библиотеке и «микро»-бенчмарки
- Изменение потребления CPU на уровне сервиса после PGO

«Микро»-бенчмарки

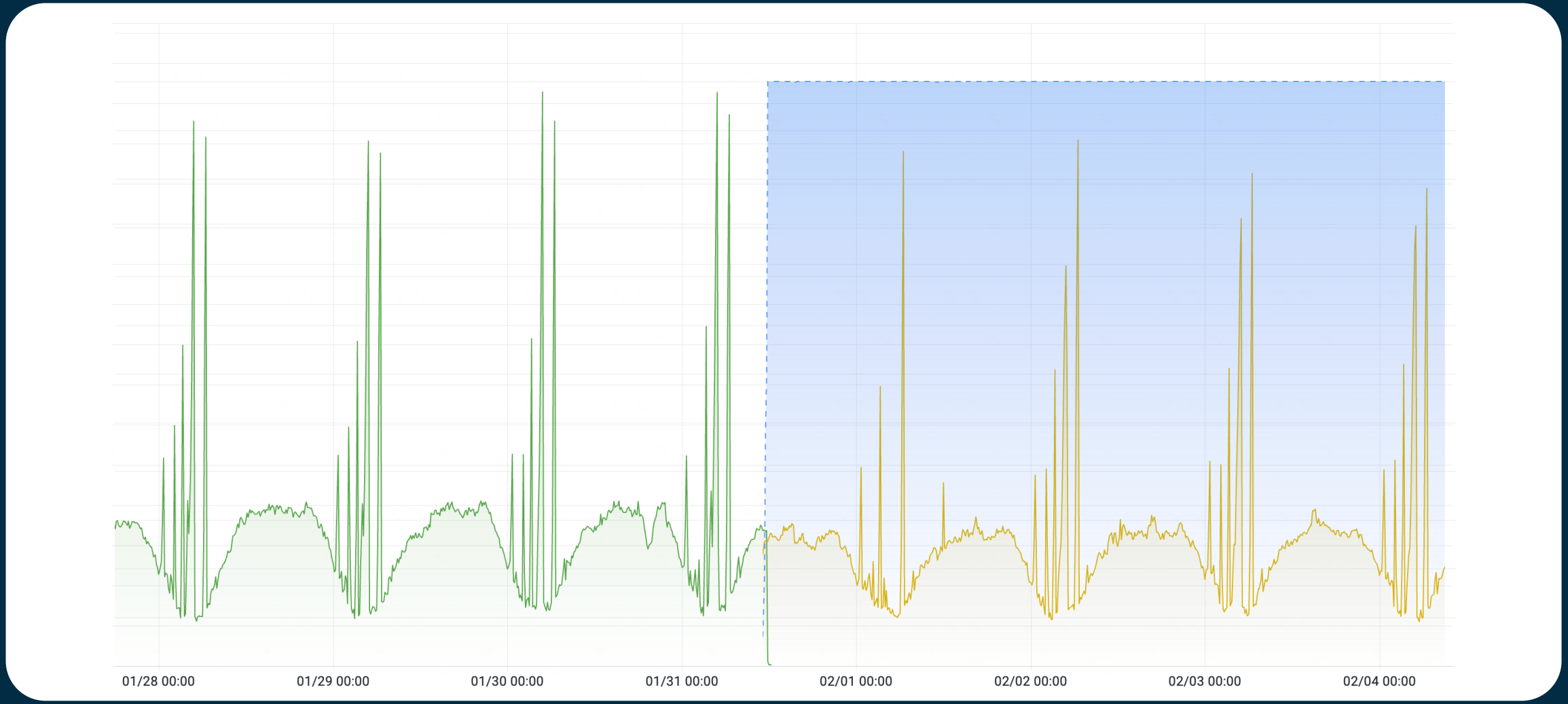
```
_Trace_ServerUnaryInterceptor-8                56.8ns ± 0%   50.5ns ± 1%  -11.06%
MatchEnlarged/client_and_handler_match-8       20.6ns ± 3%   18.6ns ± 3%   -9.91%
_HeaderSanitizerUnaryInterceptor-8             6.89ns ± 0%   6.23ns ± 0%   -9.57%
_HeaderSanitizerStreamInterceptor-8            33.0ns ± 0%   30.1ns ± 0%   -8.64%

Limiter_UnaryServerInterceptor-8               20.0ns ± 0%   20.9ns ± 0%   +4.73%
MatchEnlarged/client_and_handler_wildcard-8    21.8ns ± 0%   23.1ns ± 0%   +5.67%
MatchRealistic/client_and_handler_match-8      22.4ns ± 2%   25.1ns ± 8%  +12.15%
MatchEnlarged/handler_wildcard-8               32.1ns ± 1%   36.8ns ± 0%  +14.93%
```

Потребление CPU

На примере двух сервисов в Production-кластере Kubernetes:

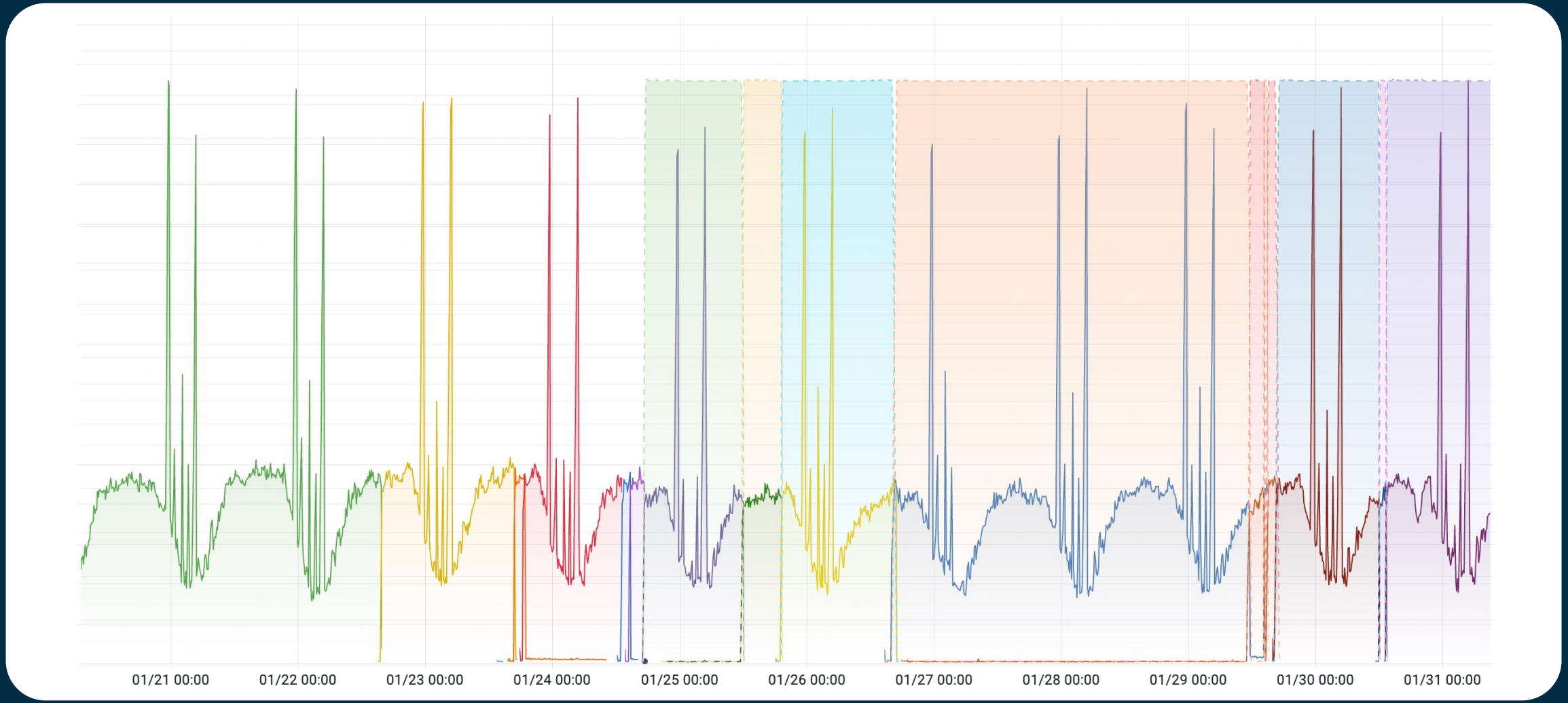
Потребление CPU

На примере двух сервисов в Production-кластере Kubernetes:



Потребление CPU

На примере двух сервисов в Production-кластере Kubernetes:



И как же использовать PGO?

01 Соберите вашу программу на Go 1.20+

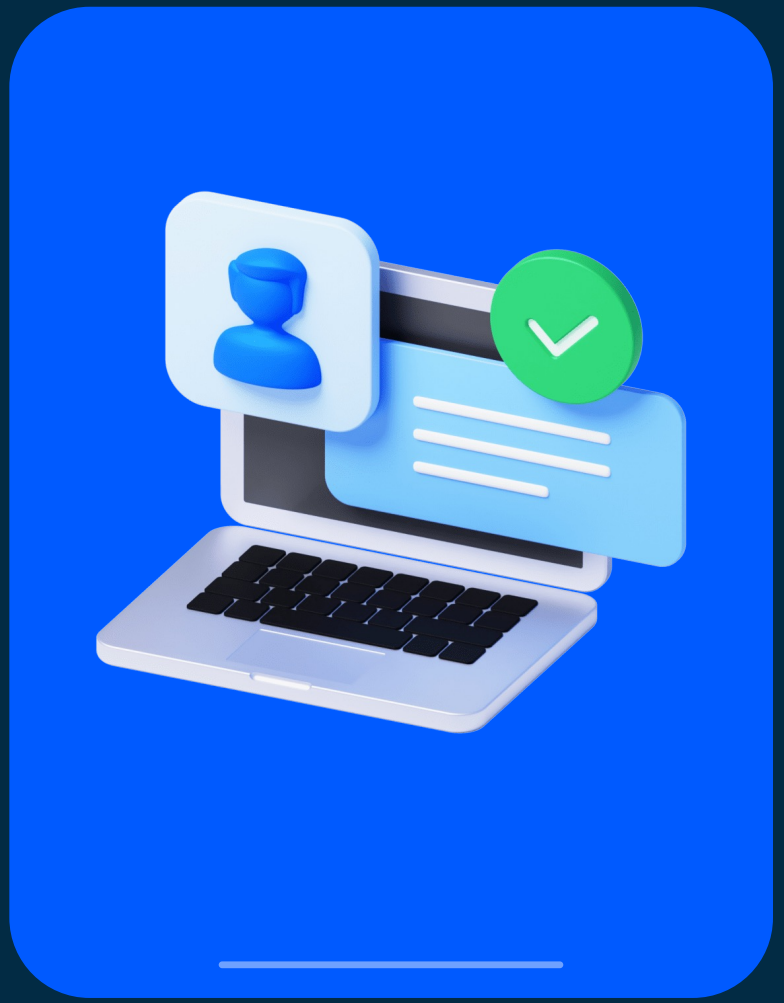
02 Соберите pprof-профиль по CPU:
- **runtime/pprof** или **net/http/pprof**

03 Соберите вашу программу с PGO:
- флаг **-pgo**

04 ...

05 Возможный profit!

Практика



- Включение PGO
- Место для файла профиля
- «А мы включили PGO?»
- Объединение нескольких профилей
- Как понять, что было встроено?
- Дебаг-флаги — Логирование
- Дебаг-флаги — Бюджет

Флаг компилятора

1.20

Флаг **-pgo** при сборке или запуске, значение — полный путь до файла

1.21 и дальше

У флага теперь значение **auto**, а это значит, что компилятор автоматически возьмёт **default.pgo** при его наличии

Место для default.pgo

Корректное место для **default.pgo** — рядом с каждым **main.go**, а не в корне модуля!

А включен ли PGO?

```
go version -m golangci-lint | grep pgo
```

```
build -pgo=/Users/kavu/golangci-lint/cmd/golangci-lint/default.pgo
```

Несколько профилей? Объединим!

Можно объединить несколько профилей!

```
go tool pprof -proto *.pgo > total.pgo
```

```
121K Feb 20 16:55 all.pgo
72K Feb 20 16:55 all2.pgo
71K Feb 20 16:55 all3.pgo
74K Feb 20 16:55 all4.pgo
73K Feb 20 16:55 all5.pgo
222K Feb 20 16:08 basic.pgo
220K Feb 20 16:23 basic2.pgo
221K Feb 20 16:19 optimized.pgo

598K Feb 20 18:28 total.pgo
```

Большой CPU-профиль — дольше сборка

1 профиль — 35 секунд

100 профилей — 202 секунды

600 профилей — 22 минуты

go tool pgopack — golang/go #58102, proposal о создании промежуточного формата для препроцессинга профилей, с целью ускорения компиляции

Как понять, что было оптимизировано?

- Сравнить CPU-профиль «до» и «после»
- Дебаг-флаги в **gcflags** специально для PGO

Сравним CPU-профили: до

pprof VIEW SAMPLE REFINE CONFIG DOWNLOAD pgo-presentation cpu

Flat	Flat%	Sum%	Cum	Cum%	Name	Inlined?
2.86s	20.41%	20.41%	3.58s	25.55%	fmt.(*fmt).fmtInteger	
2.20s	15.70%	36.12%	7.54s	53.82%	fmt.(*pp).doPrintf	
1.41s	10.06%	46.18%	1.95s	13.92%	runtime.mallocgc	
1.39s	9.92%	56.10%	1.39s	9.92%	runtime.kevent	
0.87s	6.21%	62.31%	0.87s	6.21%	runtime.madvise	
0.85s	6.07%	68.38%	4.80s	34.26%	fmt.(*pp).printArg	

Сравним CPU-профили: после

pprof VIEW SAMPLE REFINE CONFIG DOWNLOAD pgo-presentation cpu

Flat	Flat%	Sum%	Cum	Cum%	Name	Inlined?
2.65s	**19.57%**	**19.57%**	**3.54s**	**26.14%**	**fmt.(*fmt).fmtInteger**	**(inline)**
1.96s	**14.48%**	**34.05%**	**7.12s**	**52.58%**	**fmt.(*pp).doPrintf**	**(inline)**
1.33s	9.82%	43.87%	1.33s	9.82%	runtime.kevent	
1.24s	9.16%	53.03%	1.96s	14.48%	runtime.mallocgc	
1.05s	7.75%	60.78%	1.05s	7.75%	runtime.madvise	
0.75s	5.54%	66.32%	4.53s	33.46%	fmt.(*pp).printArg	

Кстати, сравнить может и сам pprof

pprof VIEW SAMPLE REFINE CONFIG DOWNLOAD unknown cpu

Flat	Flat%	Sum%	Cum	Cum%	Name
0.03s	0.00%	0.00%	-9.79s	-0.97%	honnef.co/go/tools/go/ir.makeWrapper
0.02s	0.00%	0.00%	-6.01s	-0.59%	sort.Sort
0.02s	0.00%	0.01%	-10.80s	-1.07%	runtime.mcall
0.01s	0.00%	0.01%	-4.29s	-0.42%	runtime.preemptM
0.01s	0.00%	0.01%	-3.79s	-0.37%	honnef.co/go/tools/go/ir.(*builder).addr
0.01s	0.00%	0.01%	-20.52s	-2.03%	honnef.co/go/tools/go/ir.(*Function).finishBody
0.01s	0.00%	0.01%	-13.62s	-1.35%	honnef.co/go/tools/analysis/code.Preorder
0.01s	0.00%	0.01%	-7.42s	-0.73%	go/types.(*Checker).stmt
0.01s	0.00%	0.01%	-5.33s	-0.53%	go/types.(*Checker).expr

```
go tool pprof -http=:3000 -base=default.pgo pgo-optimized.pprof
```

Дебаг-флаги

```
❯ go tool compile -d help
usage: -d arg[,arg]* and arg is <key>[=<value>]

<key> is one of:

abiwrap         print information about ABI wrapper generation
append          print information about append compilation
checkptr        instrument unsafe pointer conversions
              0: instrumentation disabled
              1: conversions involving unsafe.Pointer are instrumented
              2: conversions to unsafe.Pointer force heap allocation
closure         print information about closure compilation
defer           print information about defer compilation
```

Дебаг-флаги для PGO

```
❯ go tool compile -d help | grep pgo
        pgodebug            debug profile-guided optimizations
        pgodevirtualize     enable profile-guided devirtualization
        pgoinline           enable profile-guided inlining
        pgoinlinebudget     inline budget for hot functions
```

Логирование решений

```
-gcflags='-d=pgodebug=2,pgodevirtualize=3,pgoinline=1'
```

```
./main.go:24:7: PGO devirtualize considering call i.Foo()
./main.go:24:7: call main.foo:1: no hot callee
hot-callsite-thres-from-CDF=0.11223344556677892
hot-node enabled increased budget=2000 for func=main.Concrete.Foo
hot-node enabled increased budget=2000 for func=main.callIt
hot-budget check allows inlining for call main.callIt (cost 105) at ./main.g
```

Но можно пойти чуть дальше…

Graphviz для hot-paths

```
-gcflags='-d=pgodebug=3'
```

```
digraph G {
  forcelabels=true;
  "fmt.Fprintf" [color=black, style=solid, label="fmt.Fprintf,inl_cost=1923"];
  "main.foo" [color=black, style=solid, label="main.foo"];
  "main.Concrete.Foo" [color=black, style=solid, label="main.Concrete.Foo"];
  "os.Create" [color=black, style=solid, label="os.Create,inl_cost=72"];
  "os.(*File).Close" [color=black, style=solid, label="os.(*File).Close,inl_cost=67"];
  "main.callIt" [color=black, style=solid, label="main.callIt"];
  "main.main" [color=black, style=solid, label="main.main"];
  edge [color=black, style=solid];
  "main.Concrete.Foo" -> "fmt.Fprintf" [label="9.09"];
  edge [color=black, style=solid];
```

Graphviz для hot-paths

Дебаг не только для main

Мы не кладём весь код в пакет main и логично посмотреть оптимизации и для других пакетов, в том числе для модулей из стандартной библиотеки

Для std-пакетов менее подробный вывод

```
-gcflags='-d=pgodebug=3' -gcflags='std="-d=pgodebug=2"'
```

```
❯ go build -o golangci-lint
  -gcflags='-d=pgodebug=3’
  -gcflags='std="-d=pgodebug=2”’
  ./cmd/golangci-lint 2>&1 | wc –l

    7738
```

Включим оптимизации только для runtime

```
-gcflags='runtime=-d=pgoinline=1,pgodebug=2'
```

```
❯ go build -o golangci-lint
  -gcflags='runtime=-d=pgoinline=1,pgodebug=2’
  ./cmd/golangci-lint 2>&1 | wc –l

     514
```

Для всех пакетов максимальный вывод

```
-gcflags='all="-d=pgodebug=3"'
```

```
❯ go build -o golangci-lint
  -gcflags='all="-d=pgodebug=3”’
  ./cmd/golangci-lint 2>&1 | wc –l

  443940
```

Вывод будет очень большой, **не рекомендую**,
потому что понять там будет что-то крайне сложно!

Корректировка inline budget

- Стоимость vs бюджет

01

- Базовый бюджет — 80

02

- Обход AST для получения полной стоимости функции

03

- Сравнение

04

С бюджетом по умолчанию (2000)

```
-gcflags='runtime="-d=pgodebug=2"'
```

```
hot-node enabled increased budget=2000 for func=runtime.mallocgc

hot-budget check allows inlining for call runtime.deductAssistCredit (cost 95)
  at /opt/homebrew/Cellar/go/1.22.0/libexec/src/runtime/malloc.go:1006:31
  in function runtime.mallocgc
```

С увеличенным бюджетом (3000)

```
-gcflags='runtime="-d=pgodebug=2,pgoinlinebudget=3000"'
```

```
hot-budget check allows inlining for call runtime.deductAssistCredit (cost 95)
  at /opt/homebrew/Cellar/go/1.22.0/libexec/src/runtime/malloc.go:1006:31
  in function runtime.mallocgc

hot-budget check allows inlining for call runtime.mallocgc (cost 2137)
  at /opt/homebrew/Cellar/go/1.22.0/libexec/src/runtime/iface.go:383:15
  in function runtime.convT64
```

Будущее PGO

PGO opportunities umbrella issue — golang/go #62463

01 Улучшение девиртуализации

02 Аллокация регистров

03 Размыкание циклов

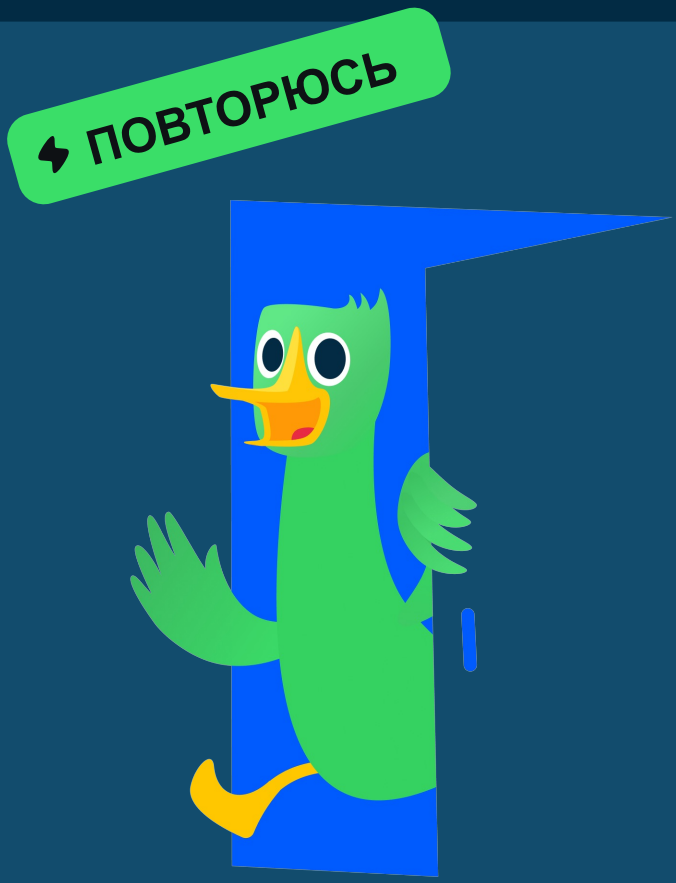
04 Порядок функций

05 alignment-циклов

Конец?

PGO — не «серебряная пуля»

Но в будущем мы можем ожидать
новых оптимизаций и улучшений тулинга, а это значит,
что выигрыш может стать больше, поэтому стоит взять
PGO в наш арсенал оптимизаций уже сейчас.



OZON tech

Спасибо за внимание!

Макс Ривейро,
ведущий разработчик в Ozon