Go로 사용자 행동 데이터 분석하기

Luft: Build OLAP DB in Go

Geon Kim / GDG Golang Korea

Index

- Introduction
- Behavioral Analytics
- Why Go?
- Retrospect
- Q&A

Introduction

Introduction

Geon Kim

- GDG Golang Korea Organizer
- Gopher (Go 1.6 ~)
- AB180 - Query Engine Team
- GitHub: @KimMachineGun

Behavioral Analytics

Behavioral Analytics

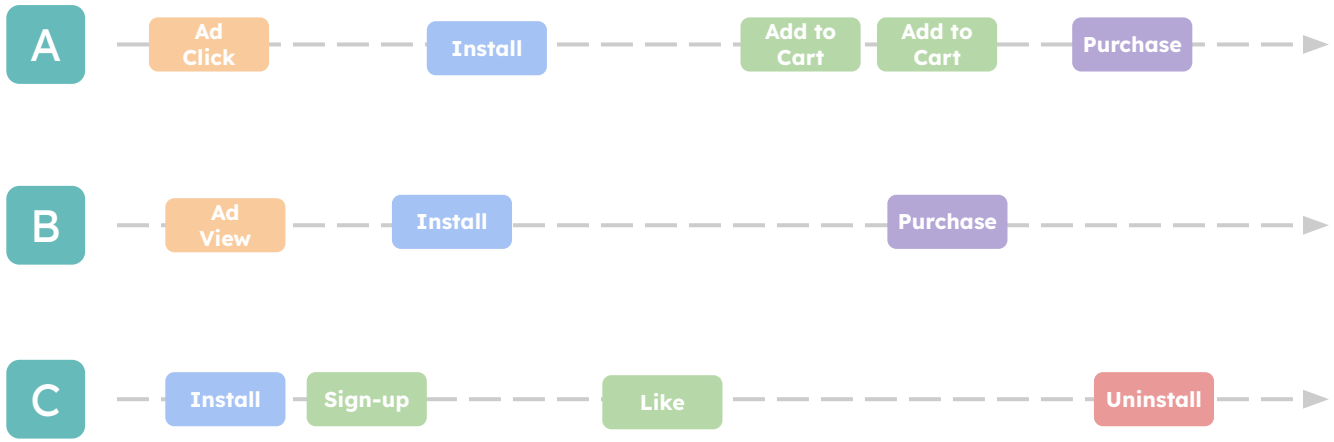
What Users Do

- Retention Rate
- Churn Rate
- Funnel Conversion Rate
- Active Users
- …

Behavioral Analytics

Technical Aspects

"사용자별 이벤트를 모아 분석하는 것"



Behavioral Analytics

Limitations of Off-the-shelf OLAP Systems

BigQuery

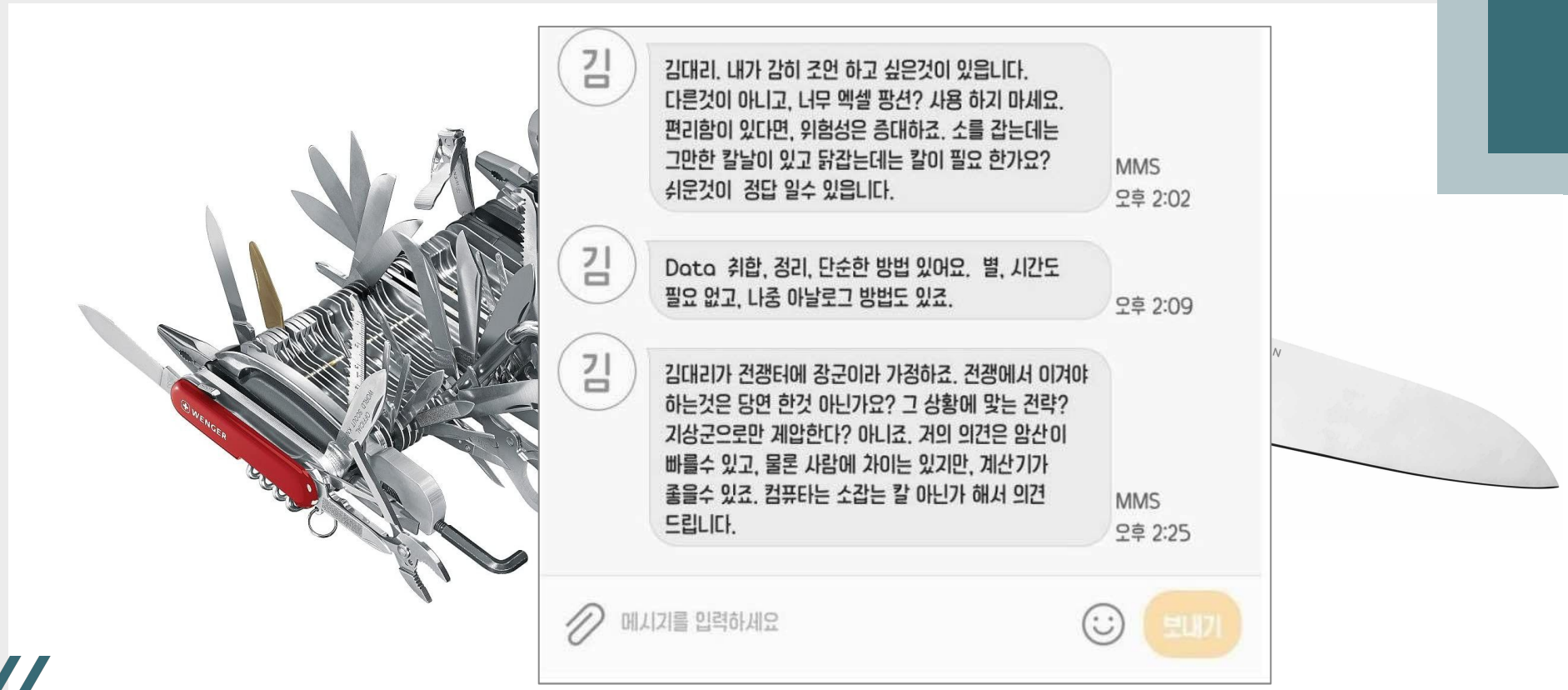
Behavioral Analytics

What They Are vs. What I Need

vs.

Behavioral Analytics

What They Are vs. What I Need



Behavioral Analytics

Luft; Airbridge's Purpose-built OLAP DB written in **Go**

"사용자 행동 데이터 분석에 특화된 자체개발 OLAP 데이터베이스"

- Fast
- Highly Available
- Scalable
- Easy to Operate
- Easy to Use

Full Version:
abit.ly/ab180-luft

Behavioral Analytics

Luft; Airbridge's Purpose-built OLAP DB written in **Go**

"사용자 행동 데이터 분석에 특화된 자체개발 OLAP 데이터베이스"

- **Fast: 사용자 행동 데이터 분석에 최적화 된 형태로 데이터를 저장 및 핸들링함**
- Highly Available: 노드 장애에 대비하여 데이터를 복제하여 저장함
- Scalable: 런타임에 동적으로 클러스터 멤버십을 관리할 수 있어 확장에 용이함
- Easy to Operate: 단순한 컴포넌트로 구성된 클러스터이기에 운영에 용이함
- Easy to Use: 사용자 행동 데이터 분석에 특화된 쿼리 인터페이스를 제공함

Full Version:
abit.ly/ab180-luft

Behavioral Analytics

Luft; Airbridge's Purpose-built OLAP DB written in **Go**

"사용자 행동 데이터 분석에 특화된 자체개발 OLAP 데이터베이스"

- Fast: 사용자 행동 데이터 분석에 최적화 된 형태로 데이터를 저장 및 핸들링함
- **Highly Available: 노드 장애에 대비하여 데이터를 복제하여 저장함**
- Scalable: 런타임에 동적으로 클러스터 멤버십을 관리할 수 있어 확장에 용이함
- Easy to Operate: 단순한 컴포넌트로 구성된 클러스터이기에 운영에 용이함
- Easy to Use: 사용자 행동 데이터 분석에 특화된 쿼리 인터페이스를 제공함

Full Version:
abit.ly/ab180-luft

Behavioral Analytics

Luft; Airbridge's Purpose-built OLAP DB written in **Go**

"사용자 행동 데이터 분석에 특화된 자체개발 OLAP 데이터베이스"

- Fast: 사용자 행동 데이터 분석에 최적화 된 형태로 데이터를 저장 및 핸들링함
- Highly Available: 노드 장애에 대비하여 데이터를 복제하여 저장함
- **Scalable: 런타임에 동적으로 클러스터 멤버십을 관리할 수 있어 확장에 용이함**
- Easy to Operate: 단순한 컴포넌트로 구성된 클러스터이기에 운영에 용이함
- Easy to Use: 사용자 행동 데이터 분석에 특화된 쿼리 인터페이스를 제공함

Full Version:
abit.ly/ab180-luft

Behavioral Analytics

Luft; Airbridge's Purpose-built OLAP DB written in **Go**

"사용자 행동 데이터 분석에 특화된 자체개발 OLAP 데이터베이스"

- Fast: 사용자 행동 데이터 분석에 최적화 된 형태로 데이터를 저장 및 핸들링함
- Highly Available: 노드 장애에 대비하여 데이터를 복제하여 저장함
- Scalable: 런타임에 동적으로 클러스터 멤버십을 관리할 수 있어 확장에 용이함
- **Easy to Operate: 단순한 컴포넌트로 구성된 클러스터이기에 운영에 용이함**
- Easy to Use: 사용자 행동 데이터 분석에 특화된 쿼리 인터페이스를 제공함

Full Version:
abit.ly/ab180-luft

Behavioral Analytics

Luft; Airbridge's Purpose-built OLAP DB written in Go

"사용자 행동 데이터 분석에 특화된 자체개발 OLAP 데이터베이스"

- Fast: 사용자 행동 데이터 분석에 최적화 된 형태로 데이터를 저장 및 핸들링함
- Highly Available: 노드 장애에 대비하여 데이터를 복제하여 저장함
- Scalable: 런타임에 동적으로 클러스터 멤버십을 관리할 수 있어 확장에 용이함
- Easy to Operate: 단순한 컴포넌트로 구성된 클러스터이기에 운영에 용이함
- **Easy to Use: 사용자 행동 데이터 분석에 특화된 쿼리 인터페이스를 제공함**

Full Version:
abit.ly/ab180-luft

Behavioral Analytics

Luft; Airbridge's Purpose-built OLAP DB written in **Go**

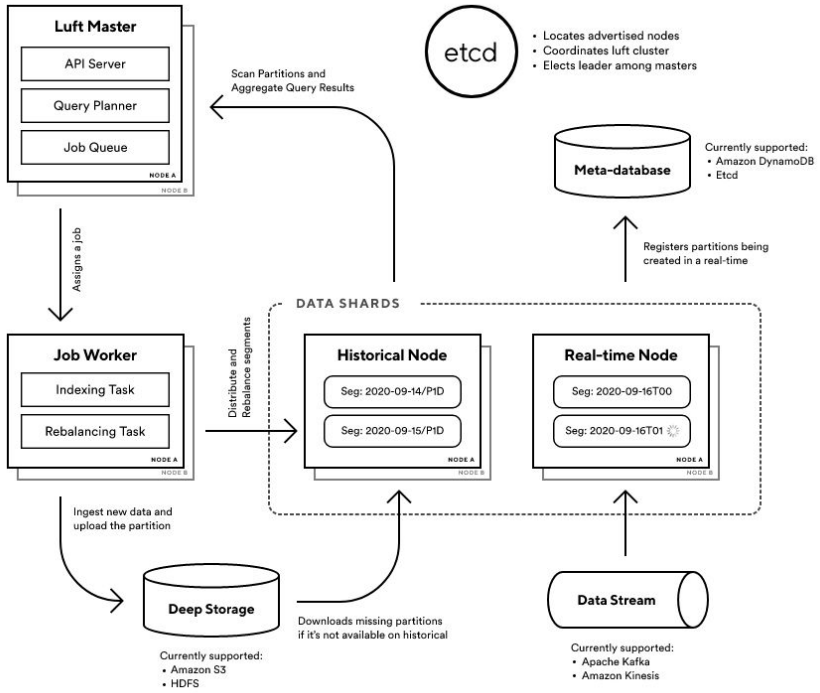
"사용자 행동 데이터 분석에 특화된 자체개발 OLAP 데이터베이스"

- Fast: 사용자 행동 데이터 분석에 최적화 된 형태로 데이터를 저장 및 핸들링함
- Highly Available: 노드 장애에 대비하여 데이터를 복제하여 저장함
- Scalable: 런타임에 동적으로 클러스터 멤버십을 관리할 수 있어 확장에 용이함
- Easy to Operate: 단순한 컴포넌트로 구성된 클러스터이기에 운영에 용이함
- Easy to Use: 사용자 행동 데이터 분석에 특화된 쿼리 인터페이스를 제공함

Full Version:
abit.ly/ab180-luft

Behavioral Analytics

Luft; Airbridge's Purpose-built OLAP DB written in **Go**



- **Master Node:**
  쿼리 API 서빙, 분산 쿼리 실행, 클러스터 코디네이팅 등을 함.

- **Shard Node:**
  디스크로부터 인덱싱 된 데이터를 읽고, 쿼리를 처리함.

- **Deep Storage:**
  인덱싱 된 데이터를 안전하게 백업해둠.
  필요에 따라 Shard Node가 로컬 디스크에 캐싱함.

Full Version:
abit.ly/ab180-luft

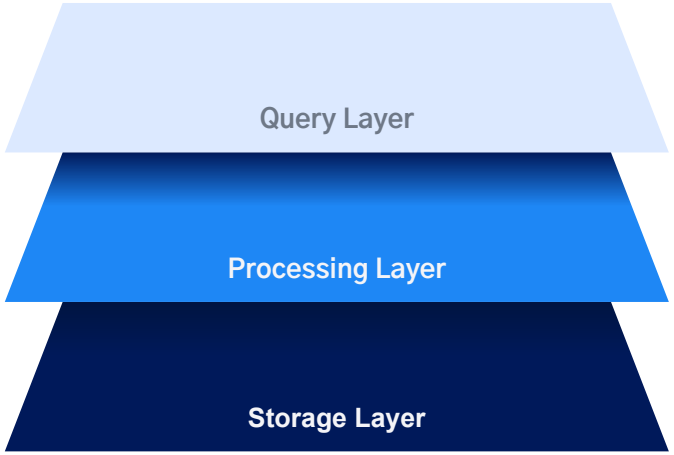
Why Go?

Why Go?

- Easy to Use/Learn
- Fast
- Concurrent/Parallel Execution
- Rich Ecosystem (esp. Distributed System)
- I love it..

Why Go?

All Layers Written in Go



- **Query Layer: (feat. ab180/luft)**
  쿼리 API 제공, 쿼리 플래닝, 쿼리 최적화, 쿼리 실행 등을 함.
  gin-gonic/gin, json-iterator/go, gorm
- **Processing Layer: (feat. ab180/lrmr)**
  분산 맵리듀스 프레임워크를 통해 쿼리를 분산 처리함.
  planetscale/vtprotobuf, tinylib/msgp, sourcegraph/conc, samber/lo
- **Storage Layer: (feat. ab180/ziegel)**
  Columnar 포맷을 사용해 데이터 효과적으로 압축, 쿼리함
  standard libraries...

Full Version:
abit.ly/ziegel-cgo-to-go

Why Go?

Libraries

- [json-iterator/go](#)
  - `encoding/json` 패키지와 호환되며 최적화된 **json** 인(디)코딩이 가능함
- [tinylib/msgp](#)
  - **MessagePack** 포맷을 인(디코딩)하기 위한 패키지로 독보적인 성능을 보여줌
- [sourcegraph/conc](#)
  - 동시성 프로그래밍을 더 쉽고 안전하게 할 수 있도록 도와주는 헬퍼 패키지
- [samber/lo](#)
  - 제네릭을 사용하여 **Lodash** 스타일로 코드를 잘성할 수 있도록 도와주는 헬퍼 패키지

Retrospect

Retrospect

Is Go Fast Enough? (⚠️Unpopular Opinions)

“YES”

- Abstraction costs, but worth it
- Rarely, Dark Arts is the answer, but mostly not
- Channels and Goroutines are not free
- Disk and Network IO are still expensive
- **Proper designs outweigh language overheads**

Retrospect

Dos and Don’ts (v1)

- **Don’t** Use Empty Interface
- **Don’t** Use Dark Arts
- **Do** Use pprof
- **Do** Use Sync.Pool
- **Do** Use Latest Go Version

Don’t

Use Empty Interface

“interface{} says nothing. — Rob Pike”

- 일반적인 상황에서 empty interface를 사용하는 것은 성능에 큰 영향을 미치진 않지만, 처리할 데이터 양이 매우 많은 경우엔 문제가 될 수 있음.
- interface type ↔ concrete type는 컴파일러가 최적화하기 어렵게 만들고, 형 변환 과정에서 할당과 복사를 자주 하게 됨.
- 성능이 아니더라도 empty interface는 유지보수성에도 영향을 미침.

Don’t

Use Dark Arts

“With the unsafe package there are no guarantees. — Rob Pike”

- 초기에 퍼포먼스 최적화를 위해 굉장히 많은 곳에서 unsafe 패키지를 사용했고, 많은 버그를 겪었음.
- Go에서 segfault와 buffer overflow를 겪는다는건... 정말 슬픈일임.
- 거의 모든 상황에서 unsafe와 같은 것을 사용한 최적화보다 유지보수하기 쉬운 코드가 훨씬 가치가 있음.
  - **대부분의 성능 저하는 Go가 아니라 내가 짠 비효율적인 로직에서 발생함.**
- 쉽게 테스트할 수 있고, 코드 수정이 잦지 않은 정말 일부 영역에서만 사용하는 것이 좋음.
  - **현재는 unsafe를 사용하는 함수에는 ‘Unsafe’라는 prefix를 강제하는 방식으로 관리하고 있음.**

Do

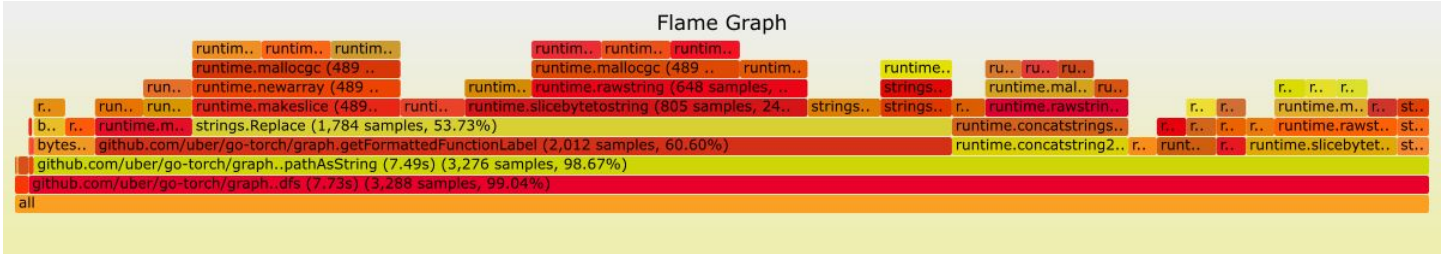
Use pprof

- 새로운 기능을 개발할 때 마다 pprof를 통해서 cpu, memory profiling을 주기적으로 함.
- pprof 자체가 성능을 개선시켜 주는 것은 아니지만 주기적인 pprof를 통해 정말 많은 인사이트를 얻을 수 있었음.

runtime.concatstrings

/Users/jbd/go/src/runtime/string.go

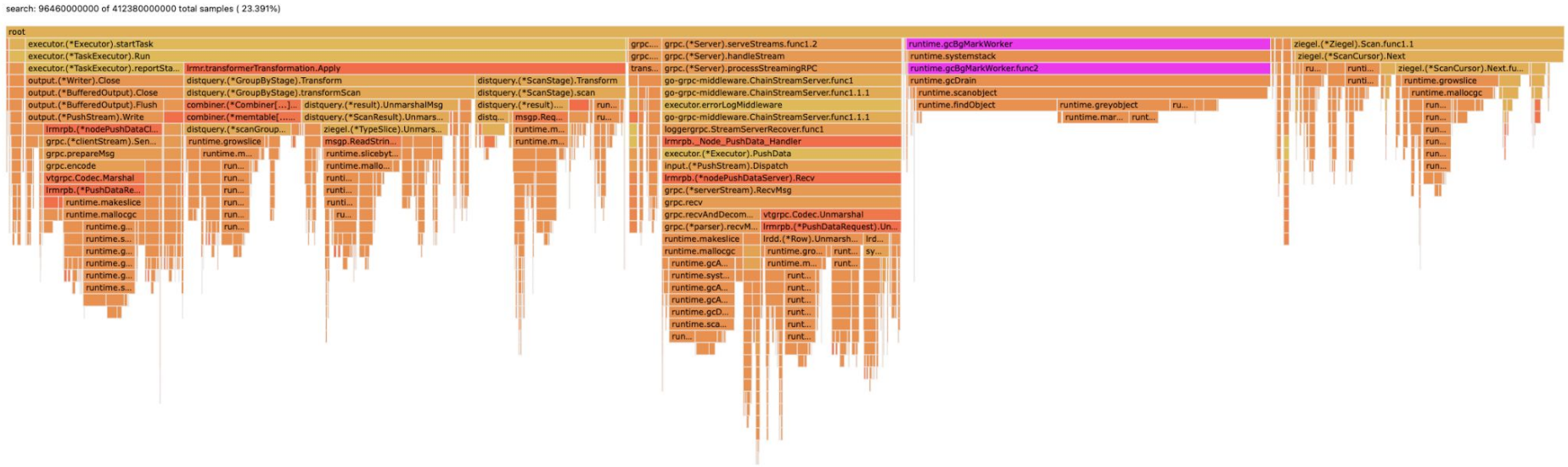
```
Total:       530ms      870ms (flat, cum) 30.63%
    15            .          . // concatstrings implements a Go string concatenation x+y+z+...
    16            .          . // The operands are passed in the slice a.
    17            .          . // If buf != nil, the compiler has determined that the result does not
    18            .          . // escape the calling function, so the string data can be stored in buf
    19            .          . // if small enough.
    20         40ms       40ms func concatstrings(buf *tmpBuf, a []string) string {
    21            .          .         idx := 0
    22            .          .         l := 0
    23            .          .         count := 0
    24         10ms       10ms         for i, x := range a {
    25            .          .                 n := len(x)
    26            .          .                 if n == 0 {
    27            .          .                         continue
    28            .          .                 }
    29         20ms       20ms                 if l+n < l {
    30            .          .                         throw("string concatenation too long")
    31            .          .                 }
    32            .          .                 l += n
    33         10ms       10ms                 count++
    34            .          .                 idx = i
    35            .          .         }
    36         10ms       10ms         if count == 0 {
    37            .          .                 return ""
    38            .          .         }
```



Do

Use sync.Pool

- 항상 문제로 제기되어온 것이지만 GC는 비쌈...
- Luft에서는 STW로 인한 latency가 문제가 아니라 background marking이 차지하는 25%의 CPU 오버헤드가 문제가 됨.
- sync.Pool을 통해 오브젝트를 재사용하고, GOGC와 GOMEMLIMIT을 잘 조절하여 GC의 발생 빈도를 줄이는 것은 큰 도움이 됨.
  - **틈새 홍보:** https://github.com/KimMachineGun/automemlimit



Do

Use Latest Go Version

- 아직 공짜 점심은 있음. (6개월마다 제공중)
- 하위 호환성을 잘 지켜주는 덕분에 큰 걱정 없이 버전을 올릴 수 있는게 Go의 큰 장점 중 하나라고 생각함.

What's new?

- `sync.Pool`, a GC-aware tool for reusing memory, has a lower latency impact and recycles memory much more effectively than before. (Go 1.13)
- The Go runtime returns unneeded memory back to the operating system much more proactively, reducing excess memory consumption and the chance of out-of-memory errors. This reduces idle memory consumption by up to 20%. (Go 1.13 and 1.14)
- The Go runtime is able to preempt goroutines more readily in many cases, reducing stop-the-world latencies up to 90%. Watch the talk from Gophercon 2020 here. (Go 1.14)
- The Go runtime manages timers more efficiently than before, especially on machines with many CPU cores. (Go 1.14)
- Function calls that have been deferred with the `defer` statement now cost as little as a regular function call in most cases. Watch the talk from Gophercon 2020 here. (Go 1.14)
- The memory allocator's slow path scales better with CPU cores, increasing throughput up to 10% and decreasing tail latencies up to 30%, especially in highly-parallel programs. (Go 1.14 and 1.15)
- Go memory statistics are now accessible in a more granular, flexible, and efficient API, the runtime/metrics package. This reduces latency of obtaining runtime statistics by two orders of magnitude (milliseconds to microseconds). (Go 1.16)
- The Go scheduler spends up to 30% less CPU time spinning to find new work. (Go 1.17)
- Go code now follows a register-based calling convention on amd64, arm64, and ppc64, improving CPU efficiency by up to 15%. (Go 1.17 and Go 1.18)
- The Go GC's internal accounting and scheduling has been redesigned, resolving a variety of long-standing issues related to efficiency and robustness. This results in a significant decrease in application tail latency (up to 66%) for applications where goroutines stacks are a substantial portion of memory use. (Go 1.18)
- The Go GC now limits its own CPU use when the application is idle. This results in 75% lower CPU utilization during a GC cycle in very idle applications, reducing CPU spikes that can confuse job shapers. (Go 1.19)

Full version: https://go.dev/blog/go119runtime

Retrospect

Dos and Don’ts (v1.1)

- **Do** Use gcflags
- **Do** Use Empty Struct
- **Do** Use automaxprocs + automemlimit

Retrospect

Do: Use gcflags - Dos and Don'ts (v1.1)

- 빌드 시점에 **gcflags** 플래그를 통해 컴파일러의 최적화 결정을 확인할 수 있음
- 이를 통해 inlining, heap escaping 등을 직접 확인하며 최적화가 더 잘 되는 코드를 작성할 수 있음

```
go build -gcflags="-m" main.go
./int.go:7:10: new(int) does not escape
./int.go:14:2: moved to heap: m
./int.go:15:2: moved to heap: n
./int.go:23:38: func literal does not escape
```

Retrospect

Do: Use Empty Struct – Dos and Don’ts (v1.1)

- empty struct는 어떤 필드도 갖지 않는 struct로 zero-size 값을 가지고 있음
- 시그널을 보내기 위한 channel, set을 구현하기 위한 map 등에서 사용할 수 있음

```go
package main

func main() {
    sig := make(chan struct{})
    go func() {
        close(sig)
    }()
    <-sig
}
```

```go
package main

import "fmt"

func main() {
    m :=
make(map[string]struct{})
    m["a"] = struct{}{}
    m["b"] = struct{}{}
    _, ok := m["a"]
    fmt.Println(ok) // true
}
```

Retrospect

Do: Use automaxprocs, automemlimit – Dos and Don’ts (v1.1)

- 컨테이너(cgroups)의 cpu, memory quota에 맞춰 Go runtime parameter 값을 설정해줌
  - GOMAXPROCS
  - GOMEMLIMIT
- 이를 통해 (약간의) 성능 향상과 안정성 향상을 공짜로 누릴 수 있음

uber-go/automaxprocs KimMachineGun/automemlimit

Q & A

Thank you