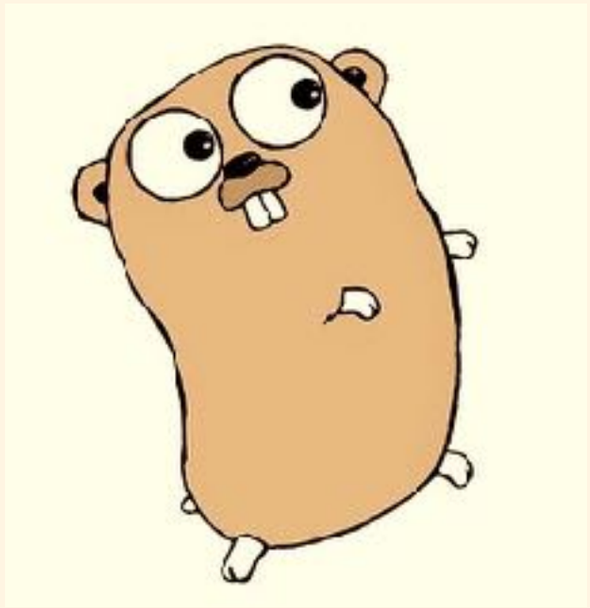
Go のバイナリを眺める

そしてバイナリファイルから Go を学ぶ



go build ?

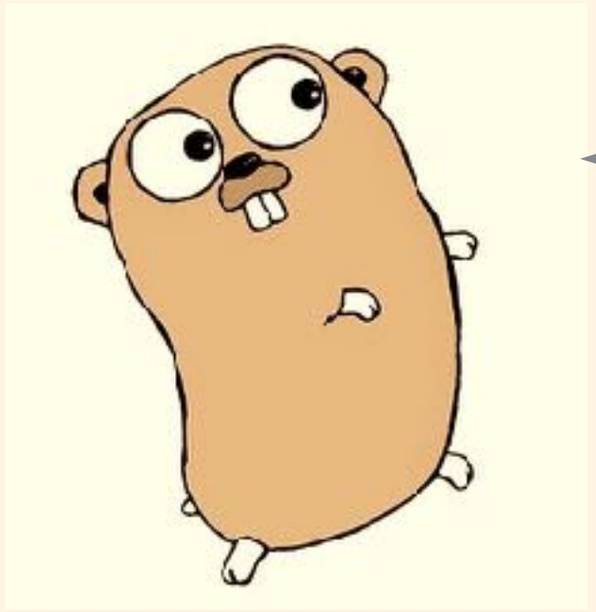
ldflags ?

.gopclntab ?

```
hello.go
1   package main
2
3   import (
4       "fmt"
5   )
6
7   func main() {
8       fmt.Println("Hello World")
9   }
10
```

```
> go version
go version go1.12.5 darwin/amd64
```

yu fujioka : CypherTec Inc.



whoami

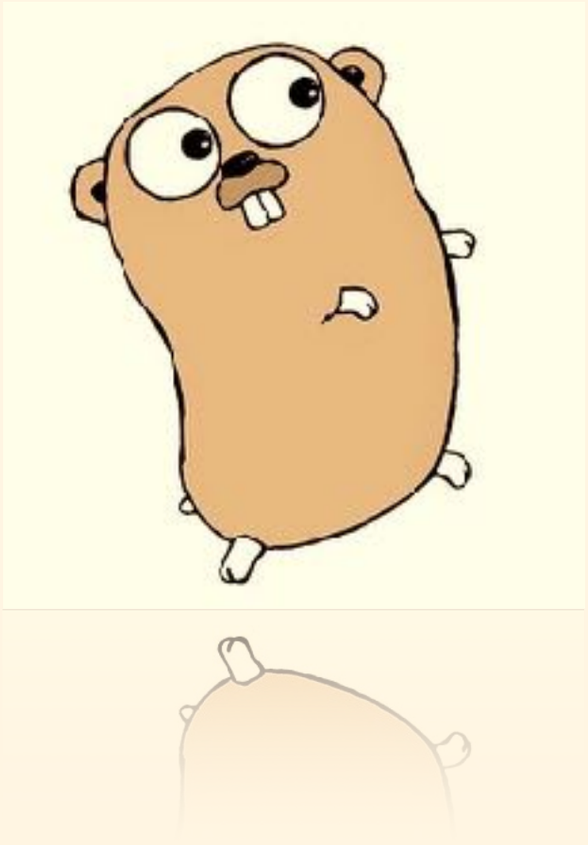
yu fujioka:

所属：CypherTec Inc.

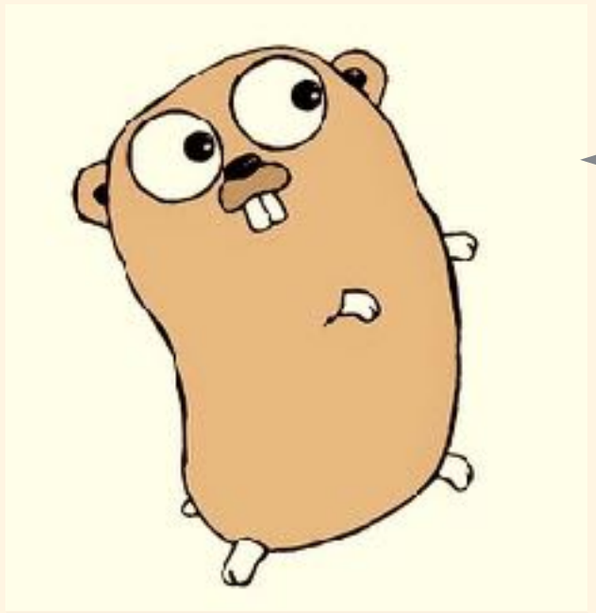
- Software Engineer 兼 Security Engineer
- C++ での製品開発やセキュリティ診断業務を担当
- Go、TypeScript などを勉強中
- 転職を機に、2015年に東京から徳島県へ引っ越し

- サテライトオフィスの導入を考えている企業の方
- サテライトオフィスに興味のあるエンジニアの方

お気軽にお声掛けください。



go build で生成される実行
ファイルを見てみよう



Let's Build

hello.go

```go
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello World")
}
```

hello.c

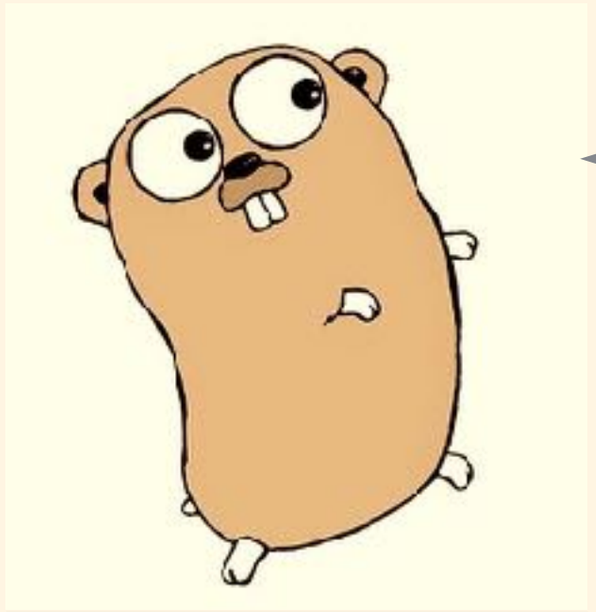
```c
#include <stdio.h>

int main(){
    printf("Hello World\n");
}
```

Go はクロスコンパイルが可能であり、ビルド環境によって生成されるバイナリが大きく異なります。
作業端末は Mac ですが、Mac の実行ファイルである Mach-O はちょっと個性的なので、今回は env を指定して Alpine 向けのバイナリを作成します。

```
env GOOS=linux GOARCH=amd64 go build hello.go
```

比較用の C プログラムは GCC でコンパイルします。

```
gcc -o c-hello hello.c
```



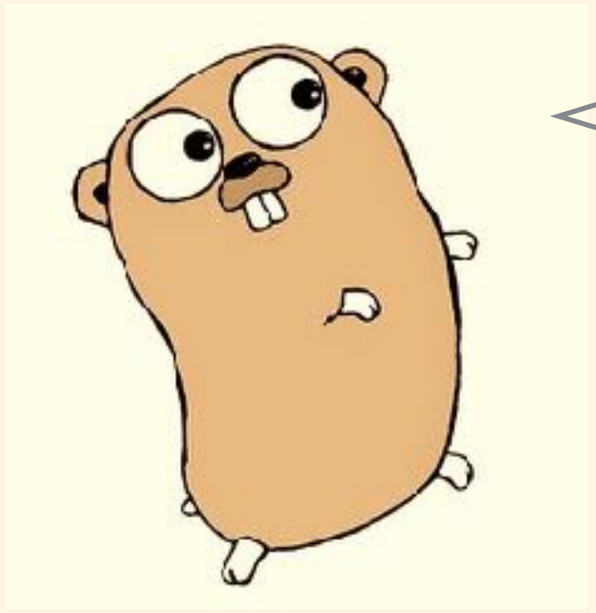
この差

```
~/G/r/g/hello >
~/G/r/g/hello > ls -ltr | tail -2
-rwxr-xr-x  1 fujioka  staff  2001712  6 22 15:05 hello
-rwxr-xr-x  1 fujioka  staff     8432  6 22 15:05 c-hello
```

Go : 2,001KB
C　: 8KB

Go はシングルバイナリで動作する（ランタイムのインストールが不要）という大きな特徴があり、そのために実行ファイルが大きくなるのは容易に想像できる。

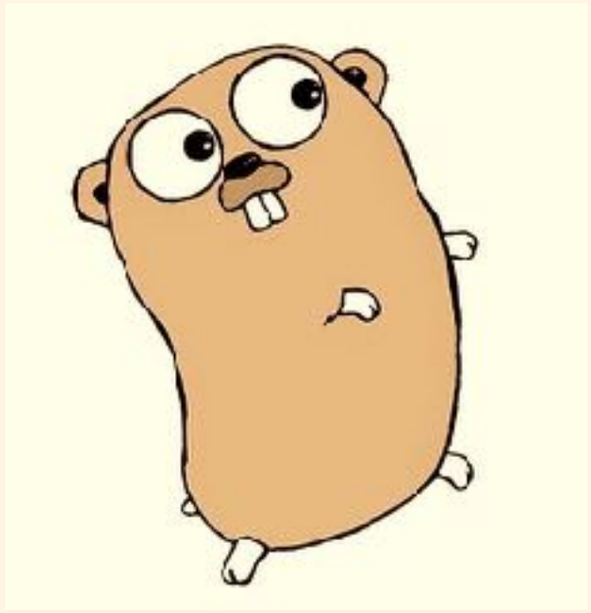
続いてこのバイナリの中を解析していきます。



readelf -S hello

[Nr]	Name	Size (Decimal)
[1]	.text	549569
[2]	.rodata	305887
[3]	.shstrtab	403
[4]	.typelink	3024
[5]	.itablink	72
[6]	.gosymtab	0
[7]	.gopclntab	462504
[8]	.noptrdata	51868
[9]	.data	27920
[10]	.bss	112464
[11]	.noptrbss	10072
[12]	.zdebug_abbrev	276
[13]	.zdebug_line	88847
[14]	.zdebug_frame	23937
[15]	.zdebug_pubname	7664
[16]	.zdebug_pubtypes	12401
[17]	.debug_gdb_script	64
[18]	.zdebug_info	185121
[19]	.zdebug_loc	81482
[20]	.zdebug_ranges	30208
[21]	.note.go.buildid	100
[22]	.symtab	78624
[23]	.strtab	79888

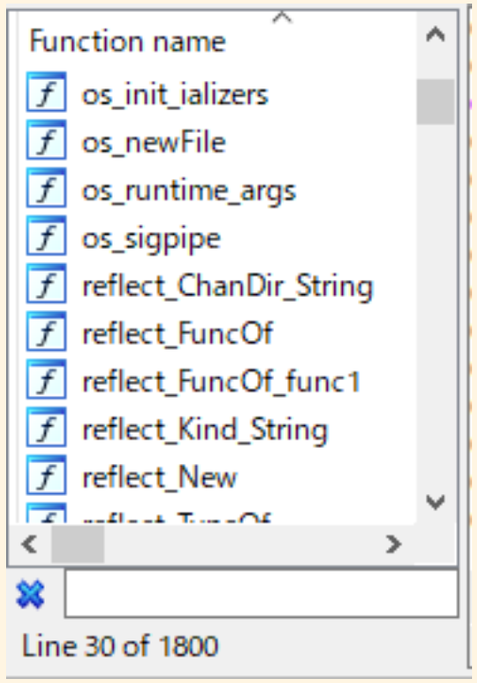
- 一番大きいのは .text セクション。
  これは実行コードが格納されている領域なので
  そもそもの実行コードの量が多いと言える。
- それに伴い .rodata (定数を格納する領域)も大きい
- .gopclntab、.note.go.buildid はなんだろう
- .gosymtab は使われてなさそう。
- デバッグ用の情報が多い



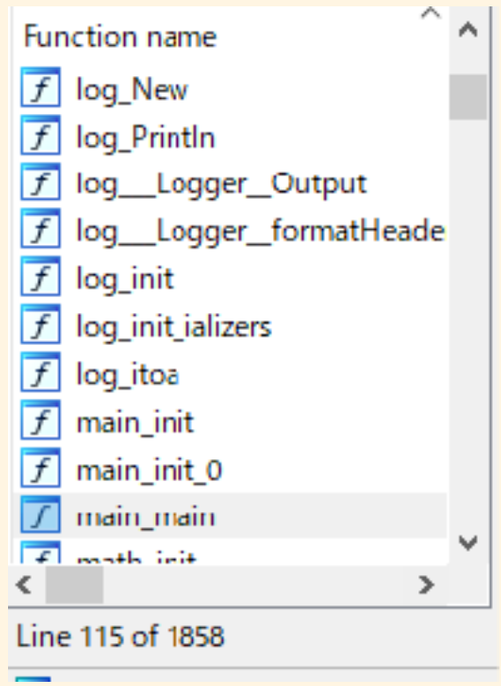
Hello World するだけの実行コードが 550KB？

[Nr]	Name	Size (Decimal)
[1]	.text	549569

`fmt.Println("Hello World")`

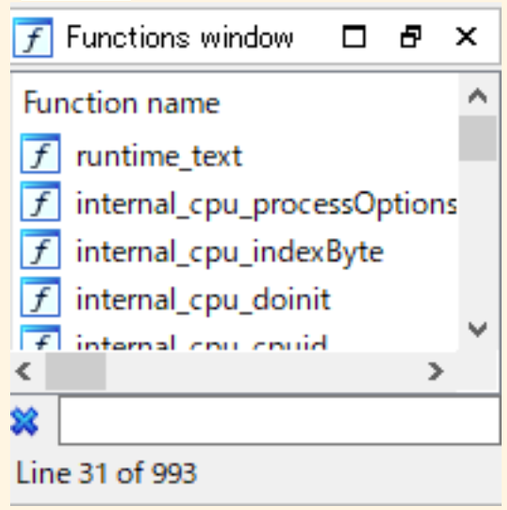


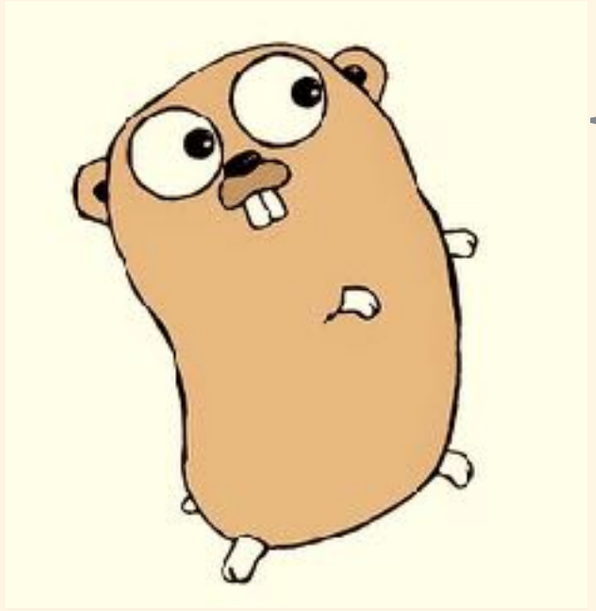
`log.Println("Hello World")`



- 高性能な逆アセンブラである IDA Pro で確認すると、
  このバイナリには 1,800 もの関数が存在した。
  関数名を見る限り、組込や標準パッケージの関数が多数
  存在している。
- もちろん標準パッケージのすべてが内包されているわけではなく、
  main の前処理や fmt の初期化などに必要な関数が詰め込まれている、という印象
- 試しに fmt を log に変えたところ、ファイルサイズは98KB 増加し、
  関数は 8 つ増えた(fmt はバイナリに残った)。
  そこで main の中で return を返すだけの空プログラムをビルドしたところ、
  サイズは 1MB 程度になり、fmt の関数もいなくなった。
  これらのことから、log 関数が内部的に fmt を利用していることが推察される。
- 【Tips】試した限り最小の Hello World は panic(“Hello World”) でした。(1,109KB)

`return`



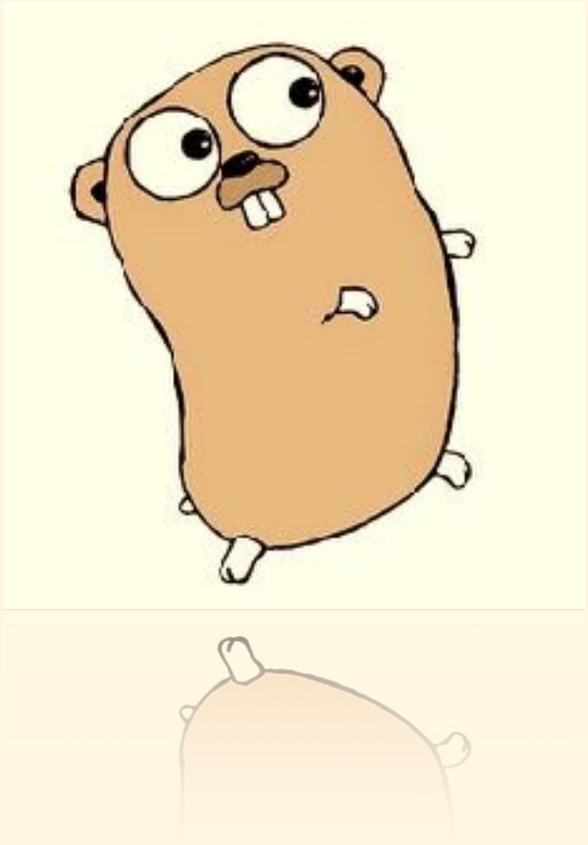


What are ...

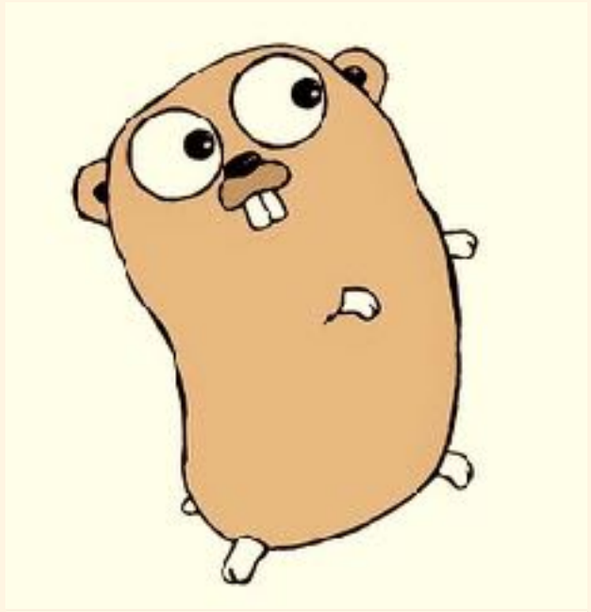
[Nr]	Name	Size (Decimal)
[7]	.gopclntab	462504
[21]	.note.go.buildid	100

```
readelf -x .gopclntab hello | tail
  0x005446c0 70755f78 38362e73 00002f75 73722f6c pu_x86.s../usr/l
  0x005446d0 6f63616c 2f43656c 6c61722f 676f2f31 ocal/Cellar/go/1
  0x005446e0 2e31322e 352f6c69 62657865 632f7372 .12.5/libexec/sr
  0x005446f0 632f696e 7465726e 616c2f63 70752f63 c/internal/cpu/c
  0x00544700 70755f78 38362e67 6f00002f 7573722f pu_x86.go../usr/
  0x00544710 6c6f6361 6c2f4365 6c6c6172 2f676f2f local/Cellar/go/
  0x00544720 312e3132 2e352f6c 69626578 65632f73 1.12.5/libexec/s
  0x00544730 72632f69 6e746572 6e616c2f 6370752f rc/internal/cpu/
  0x00544740 6370752e 676f0000                   cpu.go..


readelf -x .note.go.buildid hello

Hex dump of section '.note.go.buildid':
  0x00400f9c 04000000 53000000 04000000 476f0000 ....S.......Go..
  0x00400fac 5f30646a 474b6377 62742d4a 35315664 _0djGKcwbt-J51Vd
  0x00400fbc 7545384a 2f673373 55497a63 514a6d35 uE8J/g3sUIzcQJm5
  0x00400fcc 42624459 427a2d42 652f4370 78593257 BbDYBz-Be/CpxY2W
  0x00400fdc 6f6f5a4d 6e305844 706a2d58 674e2f49 ooZMn0XDpj-XgN/I
  0x00400fec 34455850 797a5867 58586a43 4f713059 4EXPyzXgXXjCOq0Y
  0x00400ffc 71353500                            q55.
```

- .gopclntab 領域のバイナリを見たところ、
  コンパイル時に読み込んだ .go ファイルのパスや関数名が格納されていた。
  src/debug/gosym/pclntab.go のソースをざっと眺めたところ、プログラムカウンタとソースコードの行番号をマッピングしているように見えた。
  ソースコードのフルパスがバイナリに記録されるのは少し気になるが、
  恐らくスタックトレースなどが参照しているテーブルなのではないか。
- .note.go.buildid はコンパイル対象のファイルハッシュやコンパイラのバージョンが記録されている。
  →手入力のバージョン番号などよりは信頼できる ID なので、これはデプロイしたバイナリの管理などに使えそう。



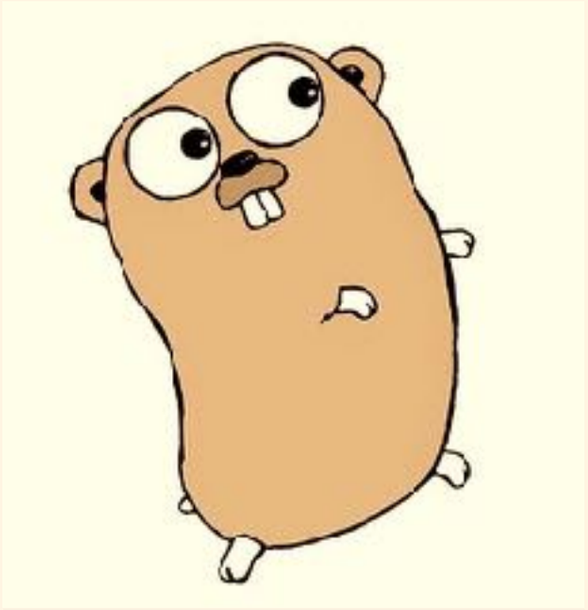
build オプションを
使ってビルドしてみよう



go help build

go help build や [Package build] を読んでみると良い。

- -tag
  - ビルドタグを付与してタグによって処理を分岐したりできる。
- go:binary-only-package と明記することでパッケージをバイナリで配布できる
  - SDK を作ったりする場合は活用できそう。
- -race
  - データ競合の検知モードを有効にする。
- -ldflags
  - ツールチェインの go tool link に引数を渡すことができる。



go build の -ldflags オプション

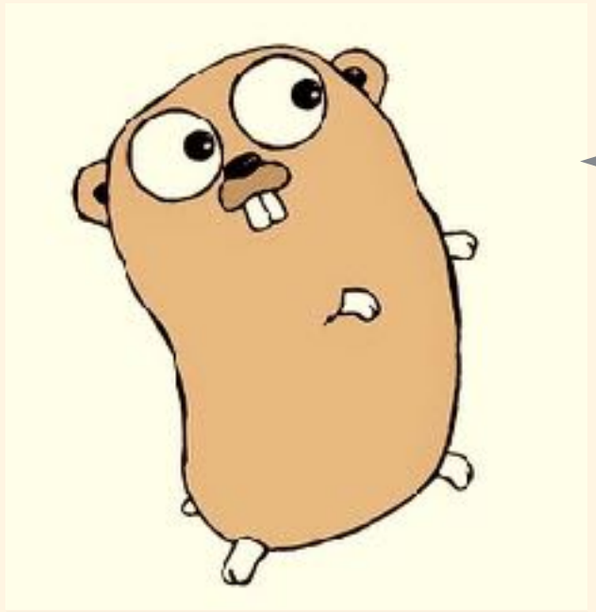
・ビルドツールチェインの go tool link に引数を渡すことができる。
・go tool link は 各パッケージのビルドオブジェクトをその依存関係とともに読み取り、それらを実行可能バイナリにバインドする。(スタティックリンク)

→ GOROOT 配下に .a という拡張子のファイルがたくさんありますが、
それらが各パッケージをビルドしたオブジェクトです。go tool compile でも作れるっぽい。

ex)
- -L dir1 -L dir2
  → 指定したディレクトリからインポートパッケージを探す
- -race
  → ライブラリの競合を検知する
- -s
  → シンボルテーブルとデバッグ情報を除去する
- -u
  → unsafe パッケージをリジェクトする
- -w
  → DWARF シンボルテーブルを除去する
- -buildid id
  → build id を指定する

→どうやら -s フラグや -w フラグでリリースバイナリから
不要な情報を削減できそう。

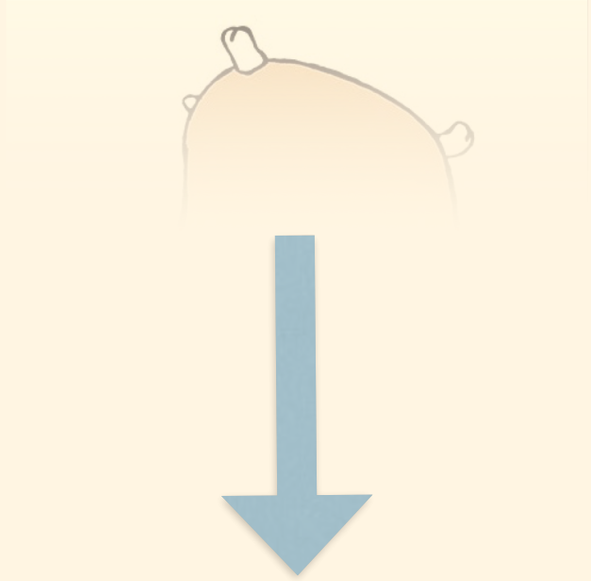
余談だが解析用のオプションも充実していた。
- -a
  → 出力結果を逆アセンブルする
- -c
  → ビルド時に関数のコールグラフを出力する
- -dumpdep
  → シンボルの依存関係グラフをダンプする
- etc...



go build -o hello-w -ldflags="-w"

DWARF を削ると

2,001KB → 1,571KB

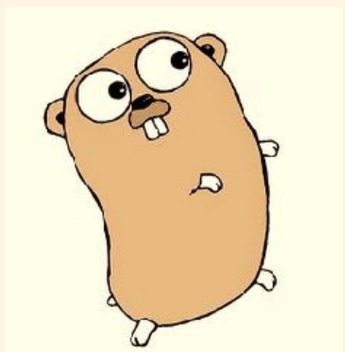


go build -o hello-sw -ldflags=”-s"

シンボルテーブルとデバッグ情報を削ると

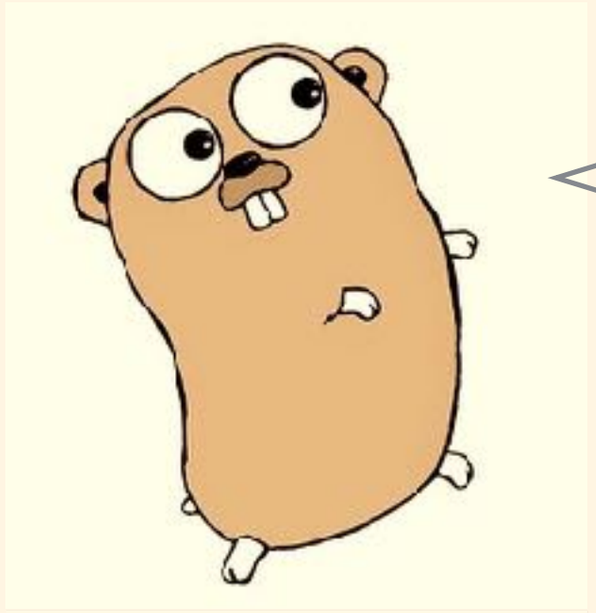
2,001KB → 1,411KB

※-w の効果は -s に含まれていた



実行ファイルによってセクションの構成は異なるので、
ダイエット効果のほどはフォーマット依存。

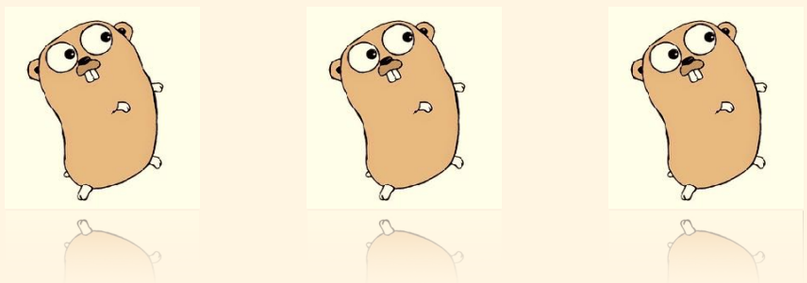
- サイズだけに着目すると、さらにUPX で圧縮をかけてバイナリを小さくするという記事も見かけたが、パフォーマンスとトレードオフになりそうなので不採用。



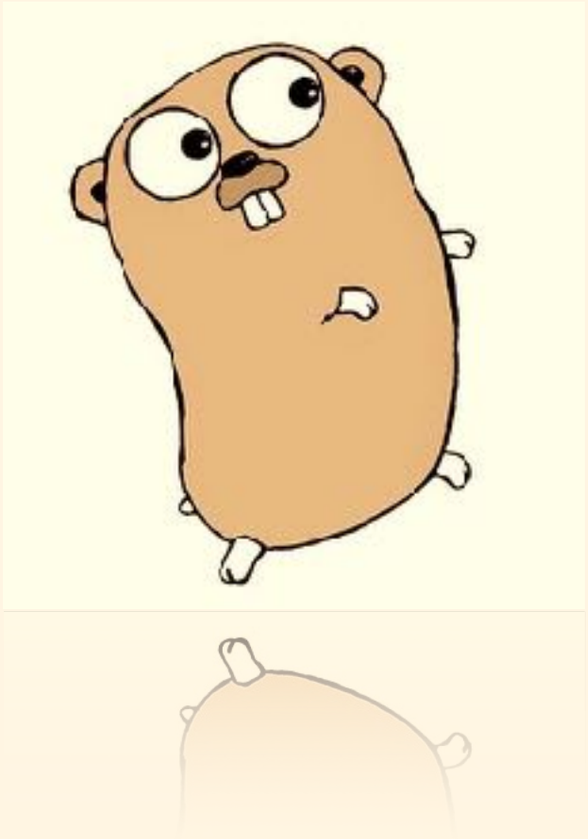
readelf -S hello-s

[Nr]	Name	Size (Decimal)
[1]	.text	549569
[2]	.rodata	305887
[3]	.shstrtab	124
[4]	.typelink	3024
[5]	.itablink	72
[6]	.gosymtab	0
[7]	.gopclntab	462504
[8]	.noptrdata	51868
[9]	.data	27920
[10]	.bss	112464
[11]	.noptrbss	10072
[12]	.note.go.buildid	100

- デバッグ情報やシンボルテーブルはセクションごとなくなり、
  セクションの数も半分程度に減りました。
- エンドユーザーに配布するようなアプリケーションであれば一般的に
  デバッグ情報は不要であり、クラッキングの糸口にも繋がるため、
  リリースビルドであれば削除(ストリップ)しておいた方が適切です。

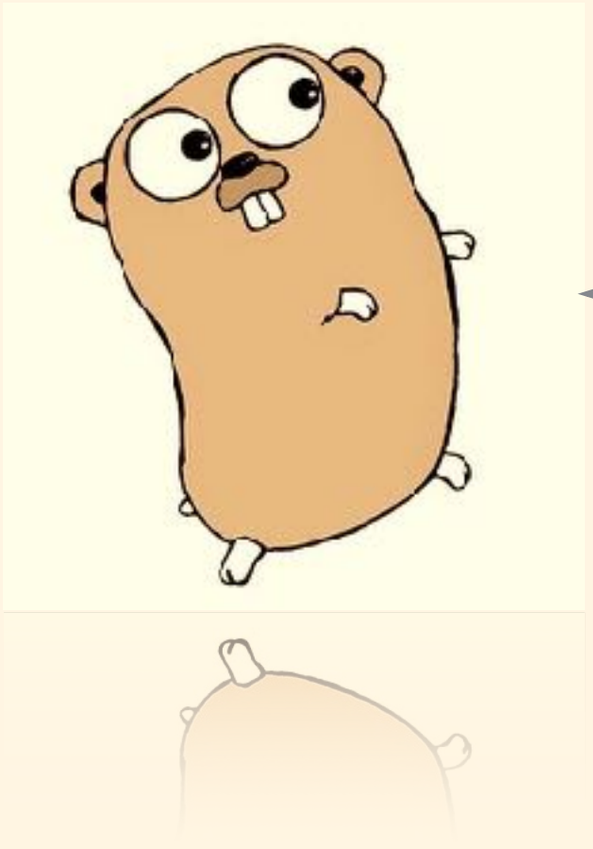


つらつら話しましたが以上です。



余談

・今回 LT のテーマを決めてから Go のビルドツールチェーンを
眺めてみましたが、リンカやコンパイラが go tool から個別に呼び出せる
機能が便利だと思いました。
→これを使った Tips が色々ありそう。
ビルド時間が長くなってきた時にパッケージを個別にビルドしておいて
ビルド時間を短縮するなど。
※おすすめの使い方があれば教えてください！
・リンカやコンパイラについても全て GoDoc から参照できるのも嬉しい。
ドキュメントが探しやすく、書式も統一されているのでわかりやすい。
Go それ自体が Go で書かれているのは良い文化。



Thanks Gopher#5