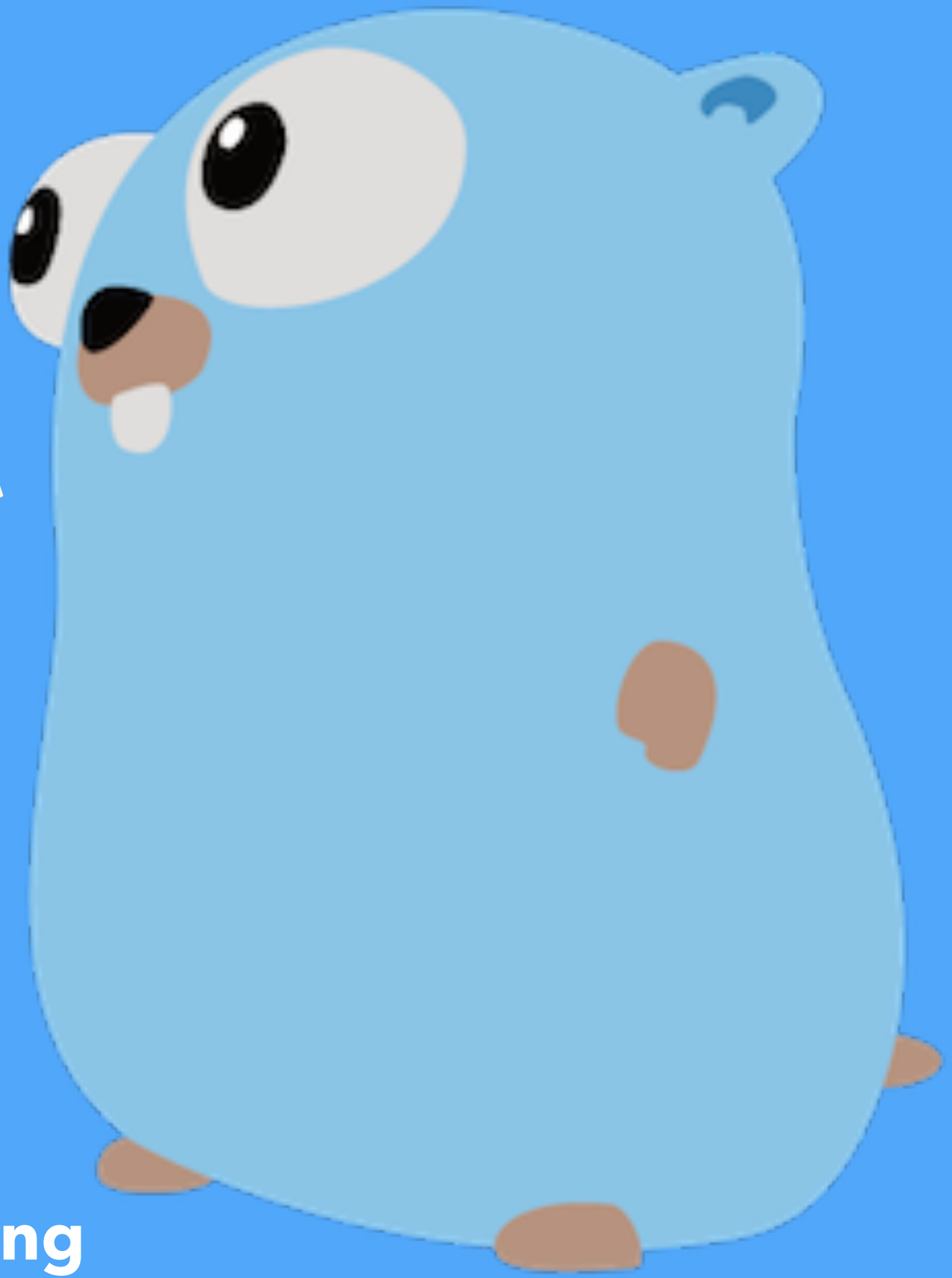
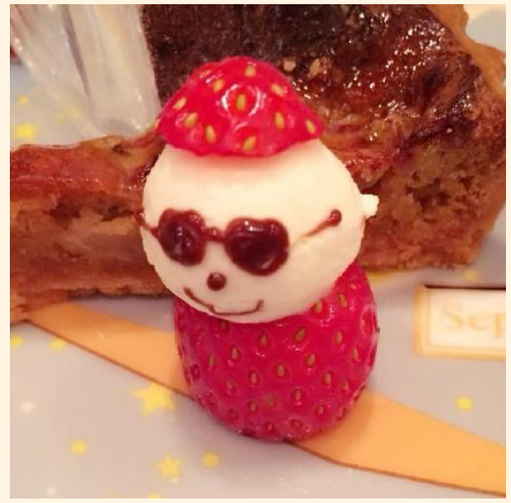
Gopherに逆らうとどうなるのか

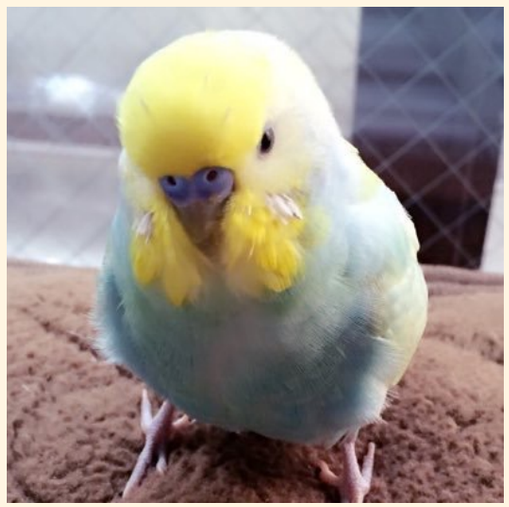
timakin / @　timakin



Go Conference 2016 Spring

自己紹介





- **高橋誠二 / ちまきん**
- Github: timakin
- Twitter: @__timakin__
- Gmail: timaki.st@gmail.com
- hatena: timakin.log
- **プライベートでGoで作ったCLIツール**
  - timakin/ts - teckstack (**テック系の情報収集CLI**)
  - timakin/octop - octopatrol (github tracking CLI)

アジェンダ

- 「go-lxcを通して実感したGolangの良さ」が正式題です。
- LXC周りの話します。
- CLIツールをGoで書き直したら高速化した話しようと思ったけど時間なさげなので今度...

何が起こったのか

- Dockerクローンを作ってた
- YAPC::Asia Tokyo 2015で「Gitの作り方」というLTに感化されて、"実装が難しそうだ"と思うものをプチ実装してみようと考えた。
- 手元でちゃんと中身わかってそうでわかってないツール...
- Dockerだね！

何が起こったのか

- p8952/bockerというshell100行の実装がある
- これを**Ruby**でやったら可読性もあがっていいのでは。（何よくなかった😇）
- DockerというかLXCをラップしたtimakin/Oceanusというのを書きました。

LXC(Linux Containers)の概要

- コンテナ仮想化に必要なchrootやcgroup、namespaceの分割等を扱うAPIを提供する技術です。

- 仮想マシンを作るのではなく、ホストOSのリソースを分割、管理することで隔離空間を作り出します。ハードウェアのシミュレーションが必要ないため、仮想化によるオーバーヘッドが少ないです。

- v0.8以前のDockerはLXC経由で一部リソース管理をしていました。

- Dockerはリソース分割もそうですが、アプリケーションのデプロイ環境やイメージのバージョニングに主眼を置いている点で、差異があるものだという印象です。

go-lxcの概要

- Go Bindings for LXC (Linux Containers)
- LXD (REST APIを提供するデーモンを備えており、ネットワーク経由での操作が可能なコンテナ技術。LXCの進化版的なもの)にも導入されています。
- 他にもruby-lxcやpython2-lxc等ありますが、開発は止まってます...

go-lxcのAPIの例

- コンテナの基本操作
  - 新規作成：func NewContainer
  - コンテナ起動：func (*Container) Start
  - コンテナ内でのコマンド実行：func (*Container) Execute
  - コンテナ破棄：func (*Container) Destroy
  - コンテナ停止：func (*Container) Shutdown
  - コンテナー覧：func Containers

go-lxcのAPIの例

- コンテナごとの独自設定、snapshot、プロセスのデーモン化
  - func (*Container) SetMemorySwapLimit
  - func (*Container) CreateSnapshot
  - func (*Container) Daemonize

go-lxcのコードリーディング

```
// go-lxc/container.go
func (c *Container) Start() error {
   if err := c.makeSure(isNotRunning); err != nil {
      return err
   }

   c.mu.Lock()
   defer c.mu.Unlock()

   if !bool(C.go_lxc_start(c.container, 0, nil)) {
      return ErrStartFailed
   }
   return nil
}
```

go-lxcのコードリーディング

```
// go-lxc/lxc-binding.c
bool go_lxc_start(struct lxc_container *c, int useinit,
char * const argv[]) {
   return c->start(c, useinit, argv);
}
```

go-lxcのコードリーディング

```
// lxc/src/lxc/lxccontainer.c
struct lxc_container *lxc_container_new(const char *name, const char
*configpath)
{
    struct lxc_container *c;
    ...

    c->start = lxcapi_start;

    ...
}

static bool lxcapi_start(struct lxc_container *c, int useinit, char *
const argv[])
{
   bool ret;
   current_config = c ? c->lxc_conf : NULL;
   ret = do_lxcapi_start(c, useinit, argv);
   current_config = NULL;
   return ret;
}
```

Dockerプチ実装過程：学習

- cgroup, chroot, snapshot作成等周りの勉強。
- 仮想環境というより隔離空間を作るイメージを持つ。
- 初期のDockerはLXC使ってたようなので、LXCの勉強も…
- Docker HubのAPIの仕様も読む…
- もちろんdockerそのものの実装とlxcのコードも読む。

Dockerプチ実装過程：実装

- e.g. pull -> run, execの流れ
  - Docker HubのAPIを叩き、images endpointを叩いて、必要なsession tokenを取得する
  - session token付きでRegistry APIを叩いて、該当タグのimage_idを取得する
  - Imageのancestry(diff履歴)を取得する
  - layer(Imageのバイナリデータ)を取得し、特定ディレクトリ配下に展開する
  - ストレージの状態の差分(ancestry)をレイヤとして重ね合わせることで、イメージを形成する。（ユニオンファイルシステム）
  - snapshotを任意のディレクトリに保存しつつ、そのイメージを元にLXCを起動、コマンド実行

Dockerプチ実装過程：実装

- e.g. pull -> run, execの流れ
  - Docker HubのAPIを叩き、images endpointを叩いて、必要なsession tokenを取得する
  - session token付きでRegistry APIを叩いて、該当タグのimage_idを取得する
  - Imageのancestry(diff履歴)を取得する
  - layer(Imageのバイナリデータ)を取得し、特定ディレクトリ配下に展開する
  - ストレージの状態の差分(ancestry)をレイヤとして重ね合わせることで、イメージを形成する。（ユニオンファイルシステム）
  - snapshotを任意のディレクトリに保存しつつ、そのイメージを元にLXCを起動、コマンド実行（ここがLXCを使う場所）

Dockerプチ実装過程：実装

- imageのロード等で外部APIを叩きに行くものの、基本的にLXCの機能で完結する。
- bockerはbtrfsによるsnapshot作成、cgroupによるリソース管理、ネットワーク設定、chrootでルート変更等をベタで実装。

結果、Golangじゃなかったから...

- まず、当然GoDocなんて親切なものはない
  - 当たり前のことですが、これが原因でAPIの仕様を把握するために直接コードを読むことになります。
- そのコードがLXCをカプセル化した、分かりやすいコードならまだしも、C拡張のコードなので死。

たとえばcontainer_start

```
static VALUE
container_start(int argc, VALUE *argv, VALUE self)
{
    int ret;
    VALUE rb_use_init, rb_daemonize, rb_close_fds, rb_args, rb_opts;
    struct start_without_gvl_args args;

    args.use_init = 0;
    args.daemonize = 1;
    args.close_fds = 0;
    args.args = NULL;
    rb_args = Qnil;

    rb_scan_args(argc, argv, "01", &rb_opts);
    if (!NIL_P(rb_opts)) {
        Check_Type(rb_opts, T_HASH);
        rb_use_init = rb_hash_aref(rb_opts, SYMBOL("use_init"));
        if (!NIL_P(rb_use_init))
```

おう、そうだな。

結果、Golangじゃなかったから...

- rubyの場合、Golangのように素敵にAPIが生えているわけではなくて、こう。

```
require 'lxc/lxc'
```

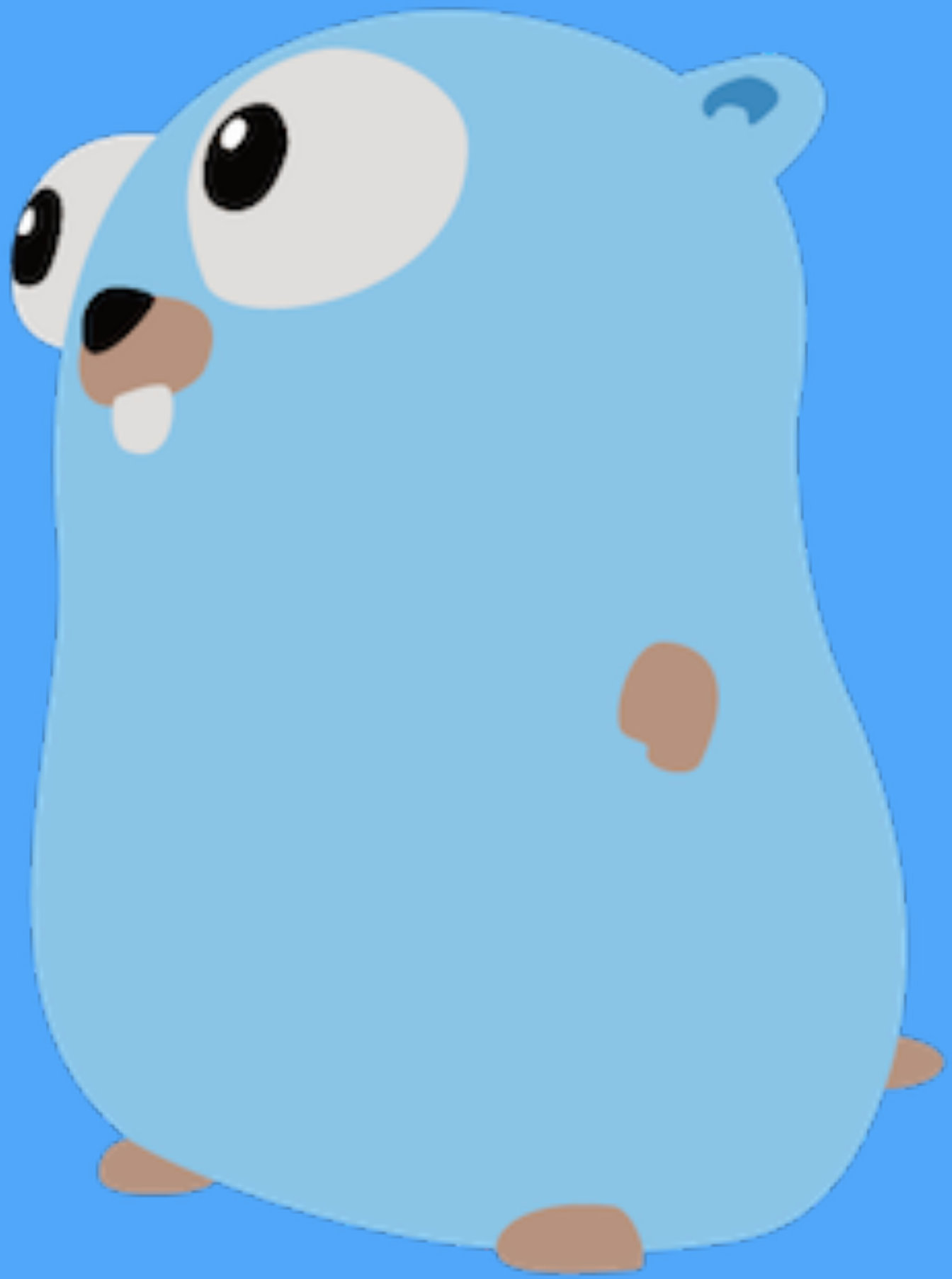
- Go以外のLXC実装はlxcを呼び出すだけで、APIとしてリーダブルだったり、利便性を考慮した単位にメソッドが分割されてなかったりします。

結果、Golangじゃなかったから...

- Golangなら
  func (*Container) IPAddressや
  func (*Container) CPUStatsなど、
  コンテナを管理するに当たって見たい情報が
  自由に見れます。

- **一方lxc/lxc-rubyは...**
  メモリもネットワーク管理もconfigのみ...
  もうちょっと労働して！

お前なんで
Rubyで
実装しよう
と思ったんだ？



まとめ

- GoのLXC周りの実装充実してます。
- APIの仕様が細分化されてて、ドキュメントも生成されてるので、サッとコンテナ仮想化のコードを読むことができます。ぜひ。
- GoでしっかりしたAPIができあがってると、それ以外の言語での実装が物足りなく感じます。逆らうのはやめましょう。

Thank you!