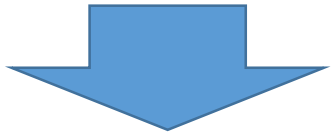
クラウドネイティブアプリケーションとRACK

自己紹介

- 名前：　金子雄大（@tktk8924）
- 仕事：　RACK開発

クラウドの登場で大きく変わったもの

- 人が手作業で行ってきたサーバ調達、セットアップがプログラムから簡単にできるようになった
- システムの利用状況に応じた自動スケールイン/アウトが可能になった
- システム（サーバ）は壊れたら同じものを新しく作るという発想に変わった
- …



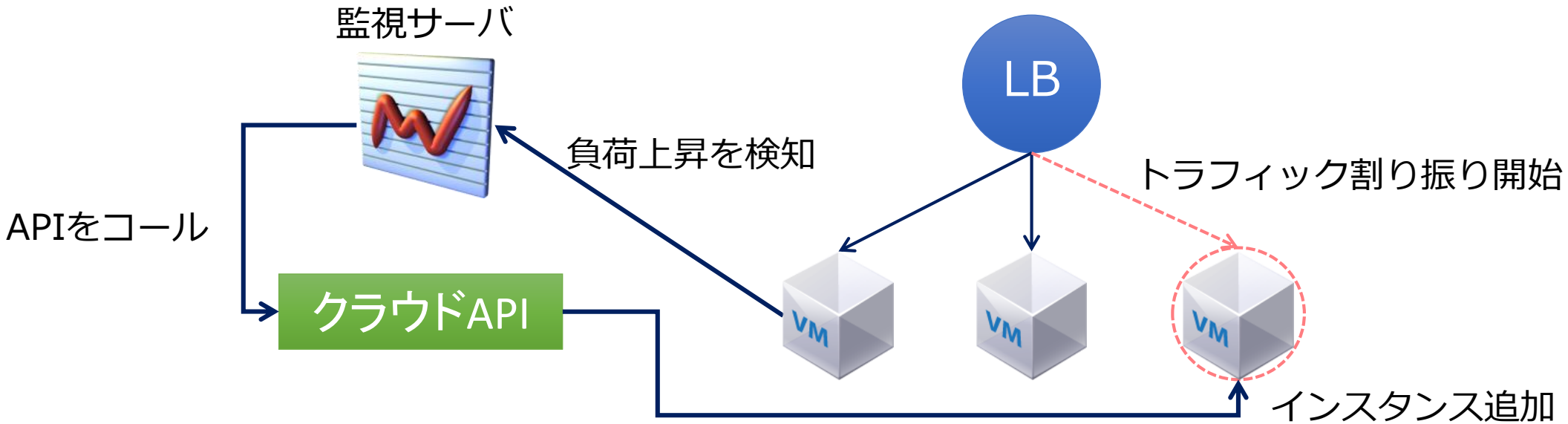
クラウドAPIの存在がこれらを可能にしている

クラウドAPIを活用したシステム
クラウドを前提として設計されたシステム

＝ クラウドネイティブ

クラウドネイティブなシステム

- インフラをプログラムから操作できるという特性を活かしたシステム
- Ex：Webオートスケールシステム
  - LB配下に最初は２台のWebサーバを配置
  - 監視サーバがWebサーバの負荷を監視
  - Webサーバの負荷が増えてきたらもう一台Webサーバを追加
  - 1台壊れても新しいサーバを用意して台数を維持



実は大きく変わってないもの

- 「アプリ」と「インフラ」は分離したまま
  - アプリはアプリで開発され、デプロイやオートスケールといったインフラ側の仕組みは別でスクリプトやレシピが開発されている
  - アプリ内部からクラウドが操作されることはない（アプリはVMを増やしたり消したりしない）
- なぜか？
  - クラウドAPIおよびSDKはアプリから使われることを想定していない
  - あくまでインフラ（VM、NW、Storage）を操作するもの
  - 追加のVMは作れても、そのVMとアプリレベルで連携はできない

クラウドをもっとApplication Centricに

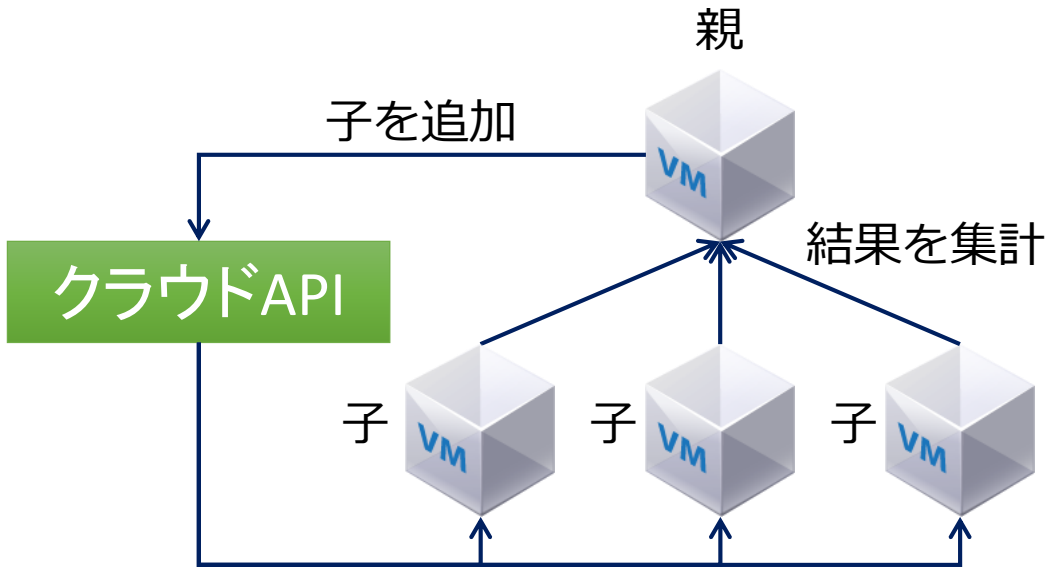
- Developer-FriendlyなAPI、SDKを用意する
  - VM、NW、Storageといったインフラを、オブジェクトとして操作できるようにする（単なるJSONデータのやり取りにしない）
  - アプリはこのオブジェクトを作成することで実行リソースを動的に確保する
  - オブジェクトの実体はVMとする
  - オブジェクト間に関係性を持たせてやる（ex. グループ、親子関係）
  - オブジェクト間で簡単にデータや状態の受け渡しができるようにする

アプリがクラウド上で自律的にスケールする新しいモデルが生まれる

クラウドネイティブアプリケーション

クラウドネイティブアプリケーション

- アプリ自らクラウドAPIをコールし、自律的にスケールする
- 実装例
  - 親はあるロジックに基づき複数の子を作成し、処理を依頼
  - 子は依頼された処理を行い、親に結果を返す
  - 親は全ての子の出力を集計した結果を出力する

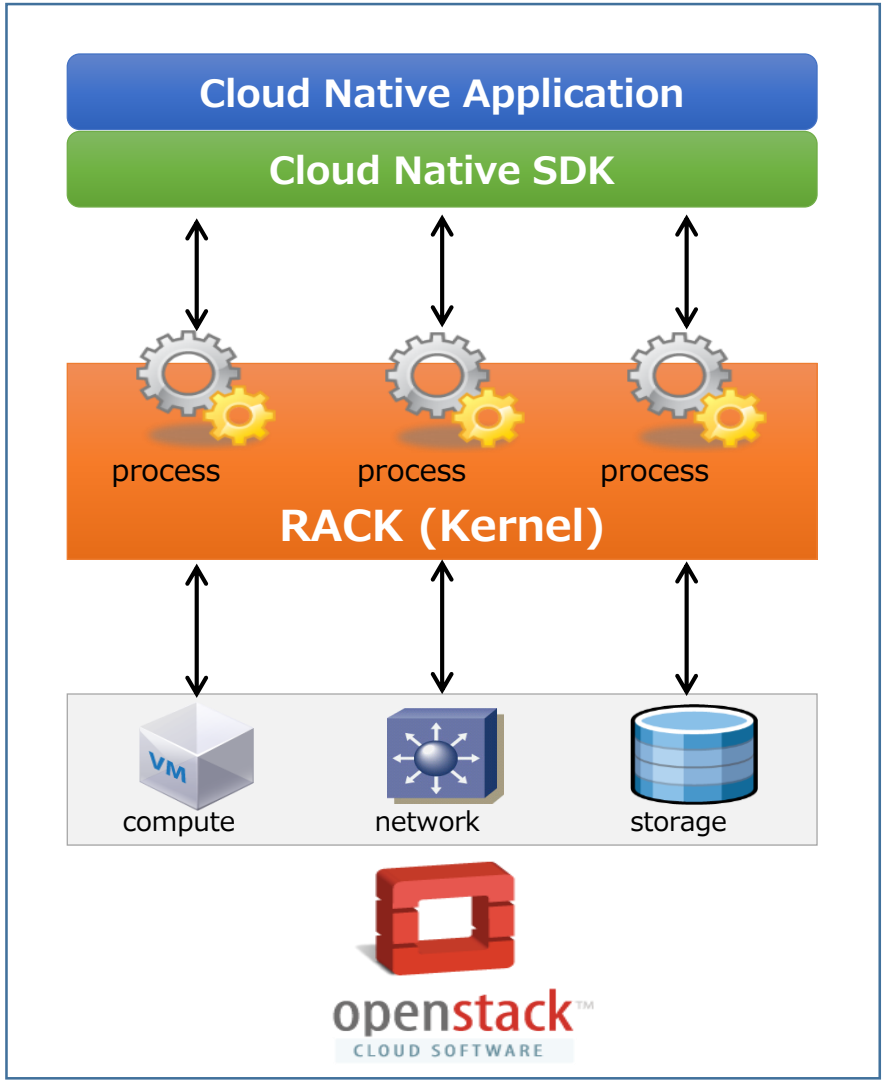


ポイント

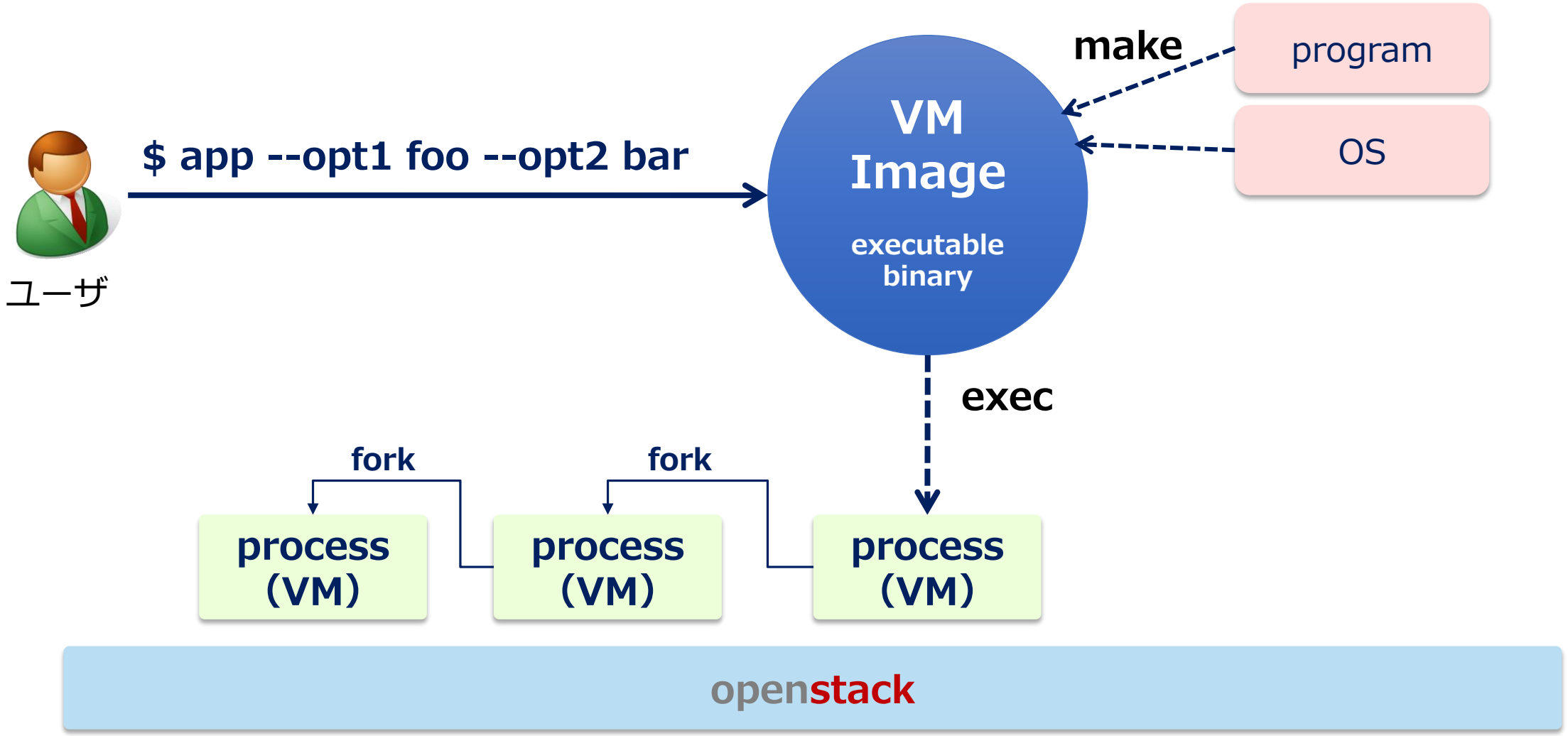
- 監視サーバもレシピもクラスタソフトも必要なく、アプリで完結
- クラウド上のほぼ無制限のリソースを使える
- 開発者はクラウドを意識せず、スケーラブルなアプリが開発できる
- 負荷ではなく、アプリケーションロジックに基づいたスケールができる
  - 処理データ量に応じてスケールする
  - 処理単位ごとにオブジェクトを追加する
  - など

RACK（Real Application Centric Kernel）

- OpenStackをApplication Centricなものにするためのソフトウェア
- APIを提供するRACKサーバとSDKで構成
- アプリの実体であるVMを“プロセス”に見立てたLinuxライクなリソース管理モデル
- OpenStackのExternalプロジェクトとして開発中

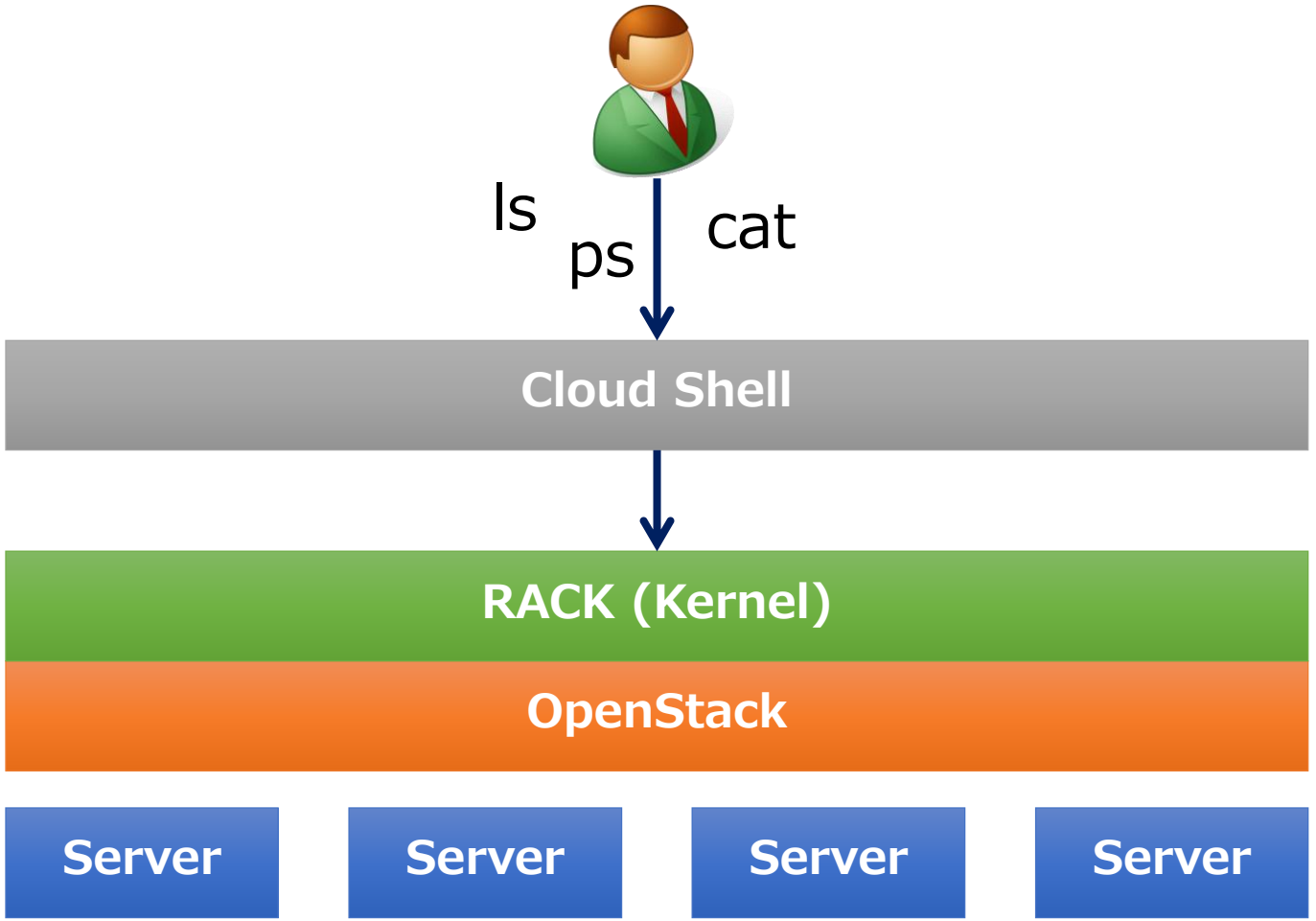


RACKのモデル


$ app --opt1 foo --opt2 bar
ユーザ
VM
Image
executable
binary
make
program
OS
exec
fork
fork
process
(VM)
process
(VM)
process
(VM)
openstack

RACKのモデル

- ユーザから見ると、OpenStackという巨大なLinuxを操作する感覚になる



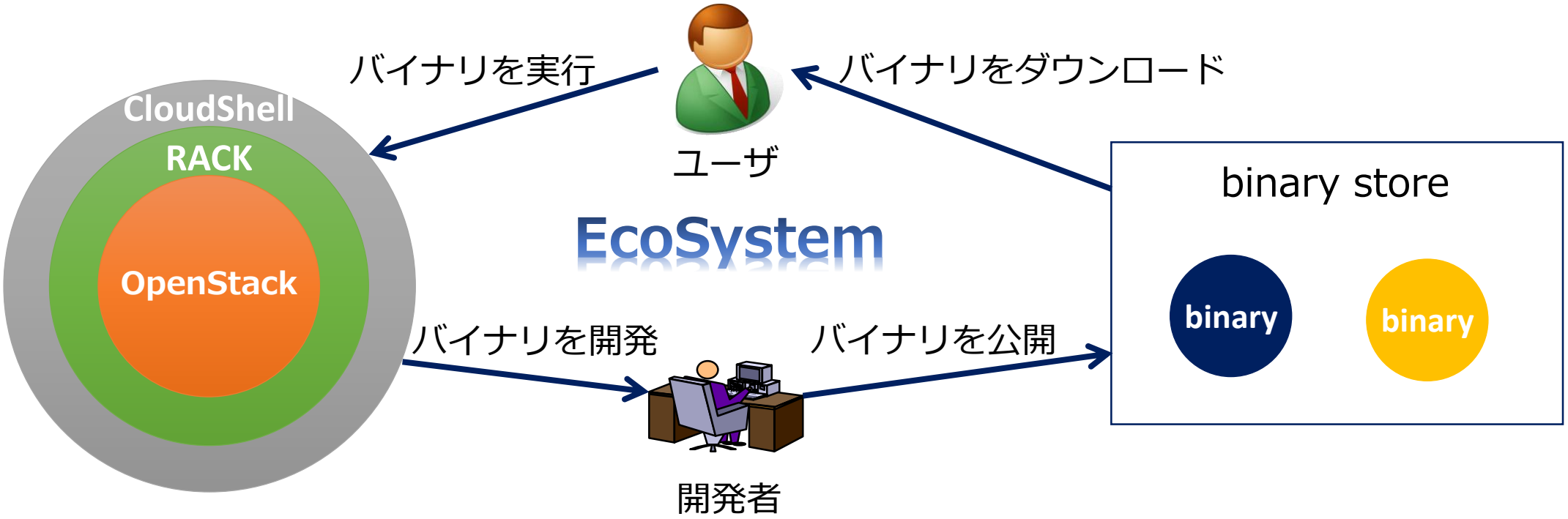
ゴールイメージ

- Linux的な世界観
- シェルからLinuxを操作するように、クラウドシェルからクラウドを操作できるようになる（クラウドを簡単に扱えるようになる）

```
# プロセス一覧
(cloud)$ ps
+-----+------+-------------------------------------+
| PID | PPID | CMD                                 |
+-----+------+-------------------------------------+
| 100 | 1    | app1 --opt1 val1 --opt2 val2        |
| 101 | 100  | app1 --opt1 val1                    |
| 200 | 1    | app2 --opt1 val1 --opt2 val2        |
+-----+------+-------------------------------------+

# 実行
(cloud)$ app3 --opt1 val1 --opt2 val2

# パイプ、リダイレクト
(cloud)$ app1 --opt1 val1 | app2 --opt1 val1 > /output/result.txt
```

ゴールイメージ

- RACK（OpenStack）という仕組みの上で様々なバイナリ開発がされる
- クラウドユーザはマーケットプレイスのような場所から使用するバイナリを選択し、シェルから実行するだけ
- アプリはクラウド上で自律的に動作するので、ユーザはクラウドのことを知らなくて良い

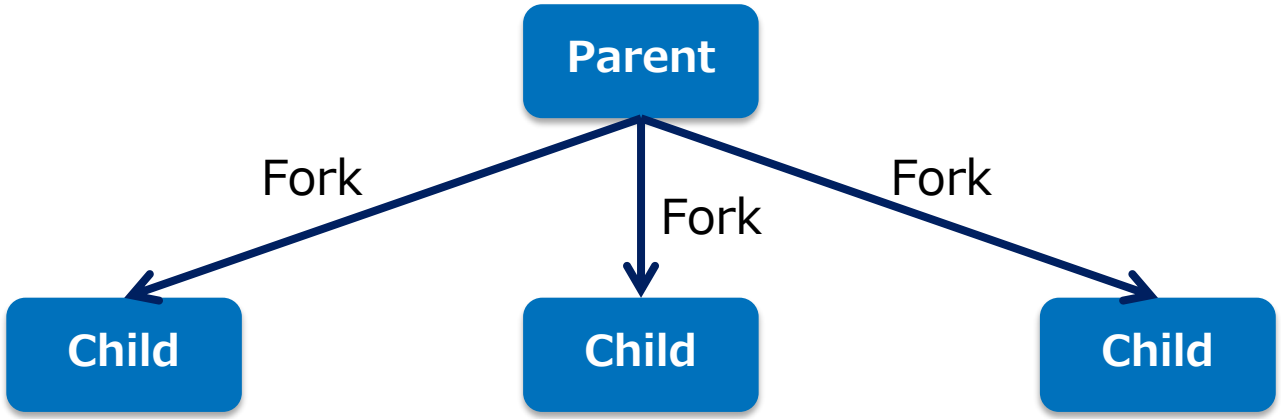


メリット

- エンドユーザ向け
  - 誰でも簡単にクラウドのパワーを享受できるようになる
  - そのインタフェースとしてLinuxライク(bashライク)なシェルを提供
- 開発者、管理者向け
  - 様々なコンポーネント（監視サーバ、スクリプト、レシピ等）を駆使することなく、クラウドネイティブなアプリが開発できる
  - 今までにない新しいタイプのアプリケーションが開発できる可能性がある

サンプルアプリ

- 数値解析アプリ
- 乱数生成を繰り返し、円周率の近似値を算出する
- 1台のVMでは膨大な時間がかかってしまう処理を、複数のVMに分散して処理させる

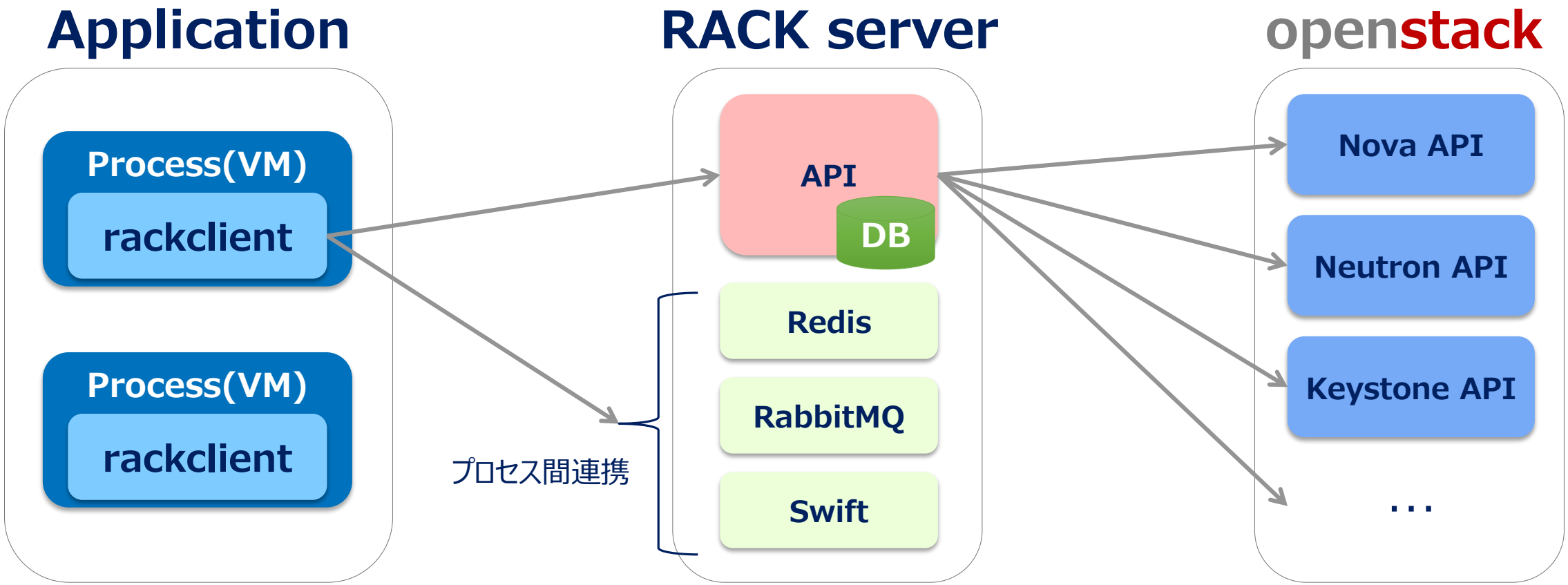


実行

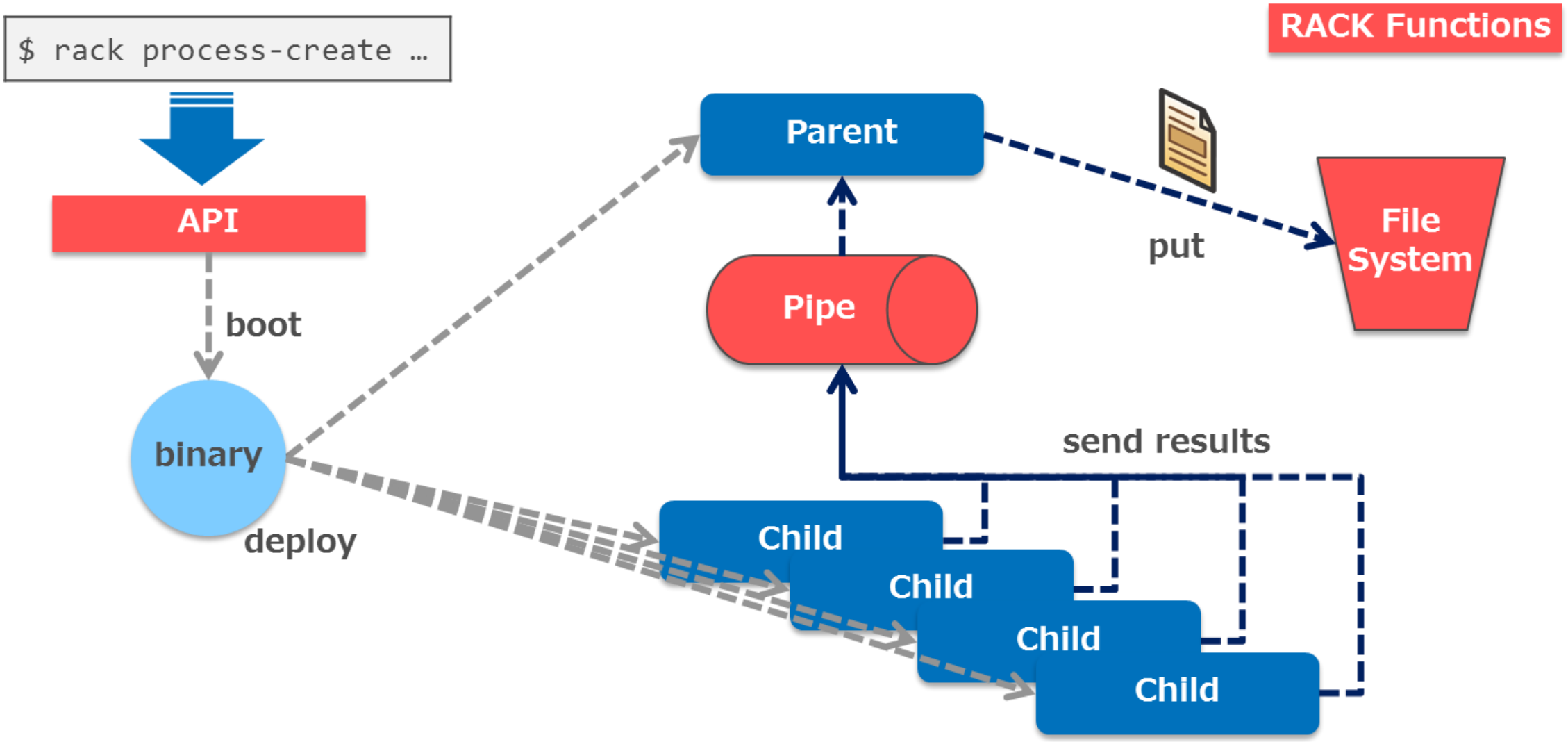
```
# rackシェルに入る
$ rack
(rack)$

# 実行
(rack)$ montecarlo --trials 100000000 --workers 10 --stdout /output/result.txt

# 結果表示（catは未実装なのでイメージ）
(rack)$ cat /output/result.txt
+----------+-------------------+
| Property | Value             |
+----------+-------------------+
| trials   | 1000000           |
| workers  | 3                 |
| points   | 785444            |
| pi       | 3.14159265359     |
| result   | 3.141776          |
| error    | 0.00018334641     |
| time     | 63.4065971375     |
+----------+-------------------+
```

デモ

RACKアーキテクチャ



アプリアーキテクチャ


$ rack process-create …
API
boot
binary
deploy
Parent
Pipe
RACK Functions
put
File System
send results
Child
Child
Child
Child

プログラム

- rc.localにアプリを実行するスクリプトを追記しておく
  ex: python app.py
- 親子判定

```
from rackclient.lib import RACK_CTX   # メタデータから自身のコンテキスト情報を取得

if __name__ == '__main__':
    if not RACK_CTX.ppid:      # PPIDがNoneなら自分は親
        parent()
    else:                      # PPIDがNoneでなければ自分は子
        child()
```

※ライブラリは変更になる可能性があります

親プログラム

- オプションを基に子プロセスをforkする

```
from rackclient.lib import syscall

def parent():
    p = syscall.pipe_reader()                # パイプを生成

    trials = int(RACK_CTX.trials)            #
    workers = int(RACK_CTX.workers)          # オプションを取得
    stdout = RACK_CTX.stdout                 #

    trials_per_child = trials / workers
    args_list = [...]                        # 子プロセスに渡すオプション

    syscall.fork(args_list)                  # fork
```

※ライブラリは変更になる可能性があります

親プログラム続き

- 子プロセスの出力を集計してレポートを出力

```
points = 0
try:
    while True:
        point = p.read()        # パイプをreadして子プロセスの出力を取得する
        points += int(point)
except EndOfFile:
    p.close_reader()

report = _make_report(trials, workers, points) # シミュレーションレポート

f = syscall.fopen(stdout, "w") # ファイルopen(出力先はSwift)
f.write(report)
f.close()                      # Swiftにアップロード

syscall.kill(RACK_CTX.pid)     # 自身をKill
```

※ライブラリは変更になる可能性があります

子プログラム

- シミュレーションのワーカーとして動作する

```
def child():
    trials = int(RACK_CTX.trials)     # オプションを取得
    p = syscall.pipe_writer()         # パイプを生成

    points = 0                        #
    for i in xrange(trials):          #
        x = random.random()           # 乱数生成
        y = random.random()           #
        if (x ** 2 + y ** 2) <= 1:    #
            points += 1

    p.write(str(points))              # パイプに処理結果を書き込む
    p.close_writer()
```

※ライブラリは変更になる可能性があります

より高級なライブラリ

- 現状はlibcのような低レベルな関数群を提供するにとどまっている
- より高級でオブジェクト指向なライブラリを開発することにより、クラウドリソースを「オブジェクトのように操作」できるようにしていく

ユースケース

- RACKはあくまでKernel的な位置付けである
- このKernelの上にどのようなアプリを作ることができるかを今後模索していく必要がある

開発勉強会・ハッカソン

- 今日より一歩踏み込んだ開発勉強会、ハッカソンを企画中！
- 一緒に新しいアプリを開発しましょう！

リンク

- プロジェクトWikiページ
  - https://wiki.openstack.org/wiki/RACK
- ソースコード
  - https://github.com/stackforge/rack
  - https://github.com/stackforge/python-rackclient
- 講演動画
  - http://goo.gl/4aGrLO
  - https://www.youtube.com/watch?v=YzbFFgdLSqw&feature=youtu.be

とりあえず試す

- 前提
  - OpenStack環境があること
  - Nova v2, Neutron v2, Glance v1, Keystone v2.0, Swift v1
- RACK環境構築ツール
  - https://github.com/ctc-g/devrack