Vol.2015-IOT-30 No.1
2015/7/3

実験用仮想サーバに適したSensuプラグインの開発

堀内 晨彦[1,a)]　最所 圭三[1,b)]

概要：Sensu とは，Ruby 製の監視フレームワークである．監視対象の自動登録，構成管理を前提とした設定など，クラウドに適したアーキテクチャとなっている．我々の研究室では，分散 Web システムの研究のため数十台の仮想サーバを運用しており，Sensu を用いて監視しようとしたが，監視オーバーヘッドの大きさや，サーバ停止時でも監視を継続しようとする問題が発生した．本稿では，これらの問題を解決するために開発した Go 言語を用いた監視のためのプラグイン，およびサーバ停止時に自動的に監視対象から削除するための手法について報告する．

1. はじめに

我々は，クラウド環境において負荷量に応じて動的に仮想キャッシュサーバ数を増減させることで，応答性を確保しつつ運用コストを低減する分散 Web システムの実現を目指している．図 1 に示すように，キャッシュサーバと大元のサーバ (オリジンサーバ) の負荷状況を監視し，負荷量に応じてクラウド環境で提供される仮想キャッシュサーバを起動・停止させ，それらにリクエストを振り分ける機能を拡張した拡張ロードバランサを開発している [1].

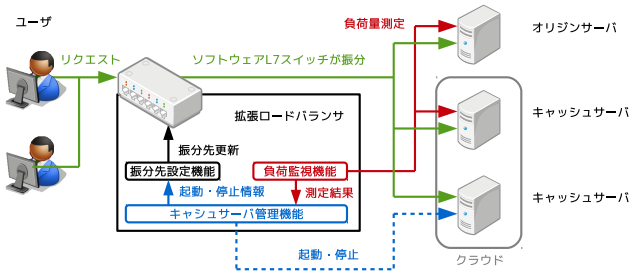

図 1　分散 Web システムの概要

分散 Web システムの実験環境においては，多くの仮想サーバと仮想クライアントを用いて実験を行うため，それらの監視やメトリクスの収集が必要である．このため，オープンソースソフトウェアとして開発されている監視フレームワーク「Sensu」を利用して，仮想サーバの監視やメトリクスの収集を行うことにした．しかし，実際に使ってみると，満足にメトリクスを収集できない事象が発生した．更に，オートスケールの際にサーバが停止しても，それを検知して自動的に監視対象から削除する機能がないため，不必要な警告を発し続けるという問題が発生した．そのため，以下の要件を満足する Sensu プラグインを Go 言語を用いて開発することにした．

- 過負荷時の Web サービスへの影響を避けるために，監視のオーバーヘッドを小さくする．
- サーバを停止した際にそれを検知し，そのサーバを監視対象から削除する．

本稿では，開発したプラグインの概要，実装および評価について述べる．

[1] 香川大学
[a)] s14g481@stmail.eng.kagawa-u.ac.jp
[b)] sai@eng.kagawa-u.ac.jp

2. Sensu の概要

本節では，Sensu の特徴とプラグイン，基本的な構成について述べる．

2.1 Sensu の特徴

「Sensu[2]」はオープンソースソフトウェアとして開発されている監視フレームワークで，サーバの監視やメトリクスの収集を行う．2011 年に Sean Porter 氏によって開発が開始され，オブジェクト指向スクリプト言語「Ruby[3]」で記述されている．

従来の監視システムである「Nagios[4]」は，クラウドなどのサーバ構成の変更が頻繁に行われる環境において，監視対象を自動的に追加することができない，追加するためにはシステムの再起動が必要であるなどの問題があった．Sensu はそれらの問題が解決されたシステムであり，以下の特徴を持つ．

- **クライアント (エージェント) の自動登録**
  監視対象のサーバに Sensu のエージェントをインストールすることで，エージェントの初回起動時に自動的に監視対象として登録される．

- **構成管理ツールとの高い親和性**
  設定は JSON 形式で記述し分割も可能である．このため，Chef[5] などの構成管理ツールを使用して容易に生成できる．
- **スケールアウトが容易なアーキテクチャ**
  監視対象のサーバ数が増加した場合でも，Sensu の監視サーバや，通信やデータの保存に用いるソフトウェアを容易にスケールアウトすることができ，処理性能の向上や冗長化が可能である．
- **ダッシュボードなどと連携可能な API**
  標準で RESTful API を提供しており，ダッシュボードなどのフロントエンドと連携しやすい．

これらの特徴は，Amazon EC2[6] をはじめとするクラウド環境での利用に適したものとなっている．

2.2 Sensu プラグイン

Sensu では，監視やメトリクスの収集，Event のハンドリングに使用するスクリプトをプラグインと呼び，監視対象のサーバ上で動作する Check と，監視サーバ上で動作する Handler がある．有志によって開発されているプラグイン集が，Community Plugin[*1] として公式に公開されている．

- **Check**
  監視やメトリクスの測定を行うプラグイン．その結果は Server によって，Event と呼ぶ JSON 形式のデータとして Handler に渡される．Event は Check と監視対象の情報を持ち，これによりどの監視対象の結果であるのかを識別する．監視結果は終了ステータスによって `0(OK)`，`1(WARNING)`，`2(CRITICAL)` と区別される．
- **Handler**
  Event を処理するプラグイン．監視対象のサーバに異常が生じたときなどに管理者にメールを送信したり，Event 中のメトリクスを外部のデータベースに保存するため形式に変換するなどの処理を行う．

2.3 Sensu の構成

図 2 に示すように，Sensu は Server，Client，API の 3 つのコンポーネントによって構成されている．Server と Client 間の通信にはメッセージ指向ミドルウェアの「RabbitMQ[7]」を，データの保存には標準で Key-Value ストアの「Redis[8]」を用いている．

- **Server**
  監視サーバ上に置かれるコンポーネントで，各 Client に対して Check の実行を指示し，その結果の収集と Handler の実行を行う．
- **Client**
  監視対象となるサーバ上にインストールするエージェントで，Check を実行し，その結果を Server に送信する．Client 単体で定期的に監視を実行することも可能である (Standalone)．
- **API**
  Server が Redis に保存したデータに対して RESTful API を提供する．この API を利用することで，Client の一覧や，現在発生中の Event などを取得できる．

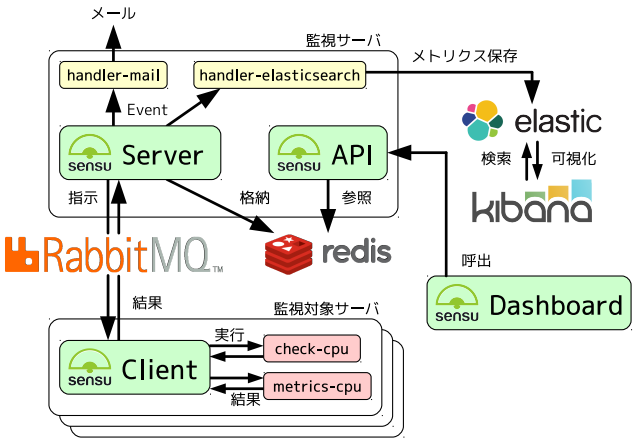

図 2 Sensu のアーキテクチャ

3. Sensu によるサーバ監視

Sensu では，Check と Client に監視項目を決定するためのグループ情報が設定されており，その組合せによって実行する Check が決定される．このグループは「Web サーバ」や「ファイルサーバ」など，監視対象の役割に基づいて設定することが一般的である．本節では，分散 Web システムの実験環境で用いている設定と，収集したメトリクスの可視化について述べる．

3.1 Server と Client の設定

Sensu では，Client が実行する Check の設定をそれぞれの Client ではなく，Server 上で一括して記述する．図 3 に 60 秒間隔で Web サービスの死活監視を行う`"check-http"`の設定例を示す．`"command"`で Client が実行するスクリプトを，`"interval"`でその間隔を設定する．先に説明したように Check は監視対象のサーバ上に置かれているため，アクセス先は同一のサーバ上の Web サービス (`localhost`) になる．`"subscribers"`では，この Check を実行する Client のグループ名 (ここでは`"http"`) を設定する．Server の設定では他にも RabbitMQ や Redis の接続先，API のポートや認証についても記述する．

図 4 に，この`"check-http"`を実行する Client の設定例を示す．`"name"`と`"address"`で自身のホスト名と IP アドレスを，`"subscriptions"`で所属するグループを設定する．Client の設定で RabbitMQ の接続先を記述し，コネク

[*1] https://github.com/sensu/sensu-community-plugins

ションを確立することで Server に登録され，監視が開始される．Server では Check の"subscribers"と各 Client の"subscriptions"を突き合わせ，実行する Check を決定する．

新たな監視項目を追加する場合は，Server の設定に Check の定義を追加し，実行するスクリプトを監視対象サーバに配置する．必要であれば，Check の"subscribers"や Client の"subscription"にグループを追加する．設定を変更した Server と Client のプログラムを再起動することで，監視が開始される．

```
"checks": {
  "check-http": {
    "command": "/etc/sensu/plugins/check-http
                -u http://localhost/",
    "interval": 60,
    "subscribers": ["http"]
  }
}
```

図 3 Sensu Server の設定例

```
"client": {
  "name": "air",
  "address": "air.eng.kagawa-u.ac.jp",
  "subscriptions": ["http"]
}
```

図 4 Sensu Client の設定例

3.2 Kibana によるメトリクスの可視化

分散 Web システムの実験環境では収集したメトリクスを可視化するために Kibana を用いている．「Kibana[11]」は Elastic 社 [10] によって開発されている可視化ツールで，Elasticsearch に保存されたデータを Web ブラウザ上でヒストグラムや表などの形式で可視化することができる．「Elasticsearch[9]」は同じく Elastic 社によって開発されている Java 製の全文検索エンジンで，ほぼ全ての操作を RESTful API 経由で行えるため，他のソフトウェアと連携し易いという特徴がある．

図 5 に Community Plugin を用いて収集したメトリクスを Kibana を用いて可視化した結果を示す．赤線が CPU 使用率，黄線がメモリ使用率，緑線がディスク使用率である．対象のサーバは CPU を 2 コア，メモリを 2GB 割り当てた仮想マシン上の Ubuntu Server 14.04 であり，監視に必要な Sensu Server，RabbitMQ，Redis，Elasticsearch，Kibana が動作している．ここでは断続的に過負荷になる実験用 Web サーバではなく，常に負荷がかかっている監視サーバの負荷状況を可視化している．

図 5 より，CPU 使用率が 20%前後を上下し，メモリ使用率が 40%を越えていることが分かる．更に，矢印で示しているように，一部ではメトリクスの測定が行えず，グラフの値が 0 になっている点もある．このように，CPU やメモリに余裕があってもメトリクスを測定できない場合があった．Community Plugins の 9 割近くが Ruby で記述されており，Ruby はインタプリタで処理されるため比較的負荷が大きいことが原因であると考えられる．

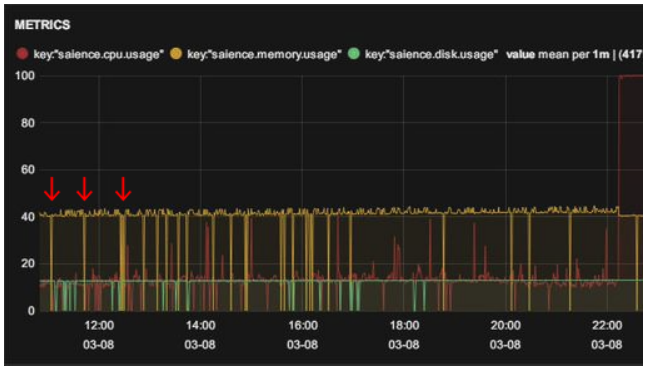

図 5 Community Plugin 使用時のメトリクス

4. 監視オーバーヘッドの抑制手法

分散 Web システムの実験では，Web サーバに対して大量のアクセスを行うため，サーバが過負荷になることが多い．そのような状態でも Sensu による監視を継続でき，なおかつ Web サーバのパフォーマンスへの影響を抑えなければならない．この問題を解決するために，コンパイラ言語である Go 言語を用いて Check を開発することにした．

4.1 Go 言語の概要

Go 言語 [12] は 2009 年に Google によって開発が開始されたオープンソースのプログラミング言語である．Linux，Mac OS X，Windows 上で動作する．我々は動作が高速である，パッケージが豊富であるということを考え，Go 言語でプラグインを開発することにした．Go 言語は以下のような特徴を持つ．

- **シンプルな構文**
  シンタックスは C 言語に類似している．構造体と関数を用いて，オブジェクトを記述することも可能である．
- **高速な動作**
  動作は C 言語に迫る速さである．また，バイナリのみでデプロイ可能で，ライブラリなどをインストールする必要がない．
- **並行処理の記述が用意**
  goroutine と名付けられた軽量スレッドにより，並行処理を容易に記述できる．goroutine 間の通信には，channel と呼ばれるメッセージパッシングを用いる．
- **豊富なパッケージ**
  今回開発するプラグインに必要な HTTP などのネットワーク，JSON などのフォーマットを標準でサポー

トしている．それ以外にも，インターネット上で多くのパッケージが公開されている．

4.2 Go 言語による Check 開発

Sensu の Check の要件として，以下の機能が必要とされている．

(1) WARNING と CRITICAL を発生させる閾値をコマンドライン引数で設定できる．

(2) 結果と閾値を比較し，終了ステータスを変更する．

Ruby を用いて Check を開発する際は，公式に提供されている sensu-plugin/check/cli ライブラリを用いることで，これらの機能を簡単に実装することができる．しかし，Go 言語では Check の開発をサポートするパッケージは提供されていない．そこで，(1) については POSIX/GNU 形式のフラグ機能を提供する github.com/spf13/pflag を，(2) については標準で提供されている os.Exit() 関数を用いて実現することにした．

図 6 に Go 言語で記述した Check の例を示す．warn と crit のフラグで閾値を指定し，測定値が warn を超えた場合は終了ステータスを 1，crit を超えた場合は 2 としている．

今回の開発では，CPU などのシステム情報を監視する Check と，HTTP の死活監視を行う Check を開発した．Community Plugin との互換性を考え，そこで用いられているアルゴリズムに準じて実装することにした．

```
func main() {
  var (
    warn  int
    crit  int
  )

  pflag.IntVarP(&warn, "warn", "w", 80, "WARN")
  pflag.IntVarP(&crit, "crit", "c", 90, "CRIT")
  pflag.Parse()

  usage := cpuUsage()

  switch {
  case usage > float64(crit):
    fmt.Printf("CRITICAL: %.0f%%\n", usage)
    os.Exit(2)
  case usage > float64(warn):
    fmt.Printf("WARNING: %.0f%%\n", usage)
    os.Exit(1)
  default:
    fmt.Printf("OK: %.0f%%\n", usage)
    os.Exit(0)
  }
}
```

図 6 Go 言語製プラグインの例

- **CPU**
  Linux の OS 情報が書き込まれている /proc/stat ファイルを指定した間隔で 2 度読み込む．各システム時間の差分の合計に対するアイドル時間の割合を CPU 使用率として用いる．
- **メモリ**
  メモリの使用状況を表示する free コマンドから，メモリ使用率を計算する．ただし，Linux がページキャッシュとして利用している buffers/cache は空き容量とする．
- **ディスク**
  ディスクの使用状況を表示する df コマンドを用いる．各ファイルシステムの空き領域と使用領域をそれぞれ合計し，その割合をディスク使用率とする．
- **HTTP**
  指定された URL に HTTP アクセスを行い，そのステータスコードを用いて判定を行う．400 系は WARNING，500 系は CRITICAL とする．リダイレクトを許容しない場合は，300 系も WARNING となる．

CPU，メモリ，ディスクについては，同様の方法を用いてメトリクスの測定を行う Check も開発した．合わせて，以下の方法を用いてネットワークインタフェースの通信量を測定する Check も開発した．

- **通信量**
  /sys/class/net/*/statistics に保存されている各ネットワークインタフェースの通信量を送受信ごとに測定する．クライアントでは受信量が多く，サーバでは送信量が多い傾向がある．

メトリクスを測定する Check の場合は終了ステータスは関係なく，図 7 に示すようにスキーマ，測定値，UNIX 時間を出力する．この出力を Handler が処理し，Elasticsearch などの全文検索エンジンに転送する．

```
$ ./metrics-cpu
example.cpu.usage 6.532663 1432631642
```

図 7 メトリクスの出力例

4.3 Community Plugin との比較

前節で述べた Go 言語で実装したプラグインと，従来の Community Plugin を比較するための実験を行った．コマンドの実行時間を測定する time コマンドを用いて，各 Check を 10 回連続して実行した際の時間を測定し，プログラム自体の処理時間である user と，OS での処理時間である system の合計した値を表 1 に示す．なお，実験の対象として CPU を 2 コア，メモリを 4GB 割り当てた仮想マシン上の Ubuntu Server 14.04 を用いた．

Go 言語製プラグインと Community Plugin を比較する

と，処理時間を約 47〜92%削減できていることが分かる．また，Sensu は同じ監視間隔の Check を複数同時に実行するため，効果はより大きいと思われる．

次に，第 3.2 節で述べた実験と同じ環境において，Go 言語製プラグインを使用してメトリクスを収集した結果を図 8 に示す．赤線が CPU 使用率，黄線がメモリ使用率，緑線がディスク使用率である．

Community Plugin を使用した場合と比較して，Go 言語製プラグインを使用した場合では，CPU 使用率は 10%以下，メモリ使用率も 40%以下を維持している．メトリクスの測定の失敗もなく，安定していることが分かる．

表 1 Community Plugin と Go 言語製プラグインの比較

	Community Plugin	Go 言語
CPU	0.79s (`check-cpu.rb`)	0.06s
メモリ	0.15s (`check-memory-pcnt.sh`)	0.07s
ディスク	0.77s (`check-disk.rb`)	0.06s
HTTP	1.51s (`check-http.rb`)	0.41s

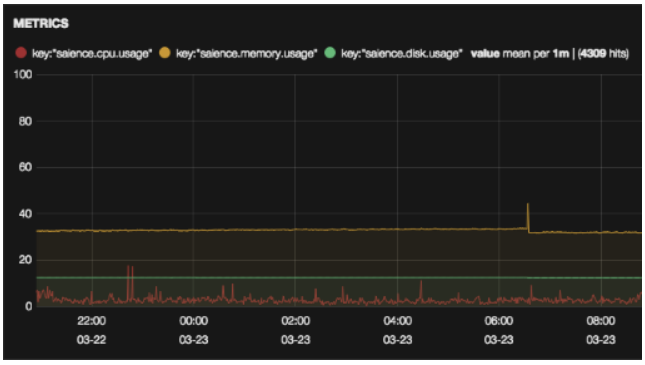

図 8 Go 言語製プラグイン使用時のメトリクス

5. VM 停止による監視対象からの削除

分散 Web システムでは，Web サーバや仮想キャッシュサーバ群の負荷に応じて，仮想キャッシュサーバの台数を動的に増減させる．負荷が増加すると新たな仮想キャッシュサーバを起動 (スケールアウト)，負荷が減少すると起動している仮想キャッシュサーバを停止 (スケールイン) する．各 Web サーバには Sensu をインストールしているが，スケールインを行った際に `keepalive`(死活監視) の Event が発生していた．`keepalive` は標準で設定されている Check であり，設定を解除することができない．そこで本節では，この `keepalive` の Event をトリガーにして，その Client を Server の監視対象から削除する手法について述べる．

5.1 Amazon EC2 を対象とした Handler

Community Plugin では，`ec2_node.rb` と呼ばれる Amazon EC2 用の Handler が提供されている．

Amazon EC2 では，負荷に応じてインスタンスの起動・停止を行う Auto Scaling 機能が提供されている．スケールアウトを行う場合は指定した設定でインスタンスを起動し，スケールインを行う場合はインスタンスを削除する．

Amazon EC2 において Sensu を使用している場合，スケールインを行いインスタンスが削除された際に Event が発生することになる．`ec2_node.rb` では，`keepalive`Event が発生した Client が Amazon EC2 上に存在しているかを確認し，存在していない場合は Sensu API を用いてその Client を Server の監視対象から削除する．

しかし，`ec2_node.rb` は Amazon EC2 環境でしか動作せず，他のパブリッククラウドサービスやプライベートクラウドサービスにおいては利用することができない．

5.2 監視対象を削除する Delete Handler の設計

前節で述べた問題を解決するため，Amazon EC2 以外のクラウド環境でも利用可能な，`keepalive` の Event をトリガーにして，その Client を Server の監視対象から削除する「Delete Handler」と名付けた Handler を開発することにした．Delete Handler の動作手順は以下のようになる．

(1) `keepalive` の Event データを受け取る．

(2) Event の終了ステータスが設定した値であるかどうかを確認する．

(3) 対象の Client が設定した`"subscriptions"`に属しているかどうか確認する．

(4) (2) と (3) の条件を満たす場合，Sensu API を用いて対象 Client を Server の監視対象から削除する．

`ec2_node.rb` と異なり，`WARNING` と `CRITICAL` のいずれの段階で削除するかを設定をできる，Client が特定の `"subscriptions"`に属する場合に削除できるなど，より汎用的に使えるように工夫している．

図 9 に Delete Handler の設定を示す．`"status"`で削除する終了ステータス，`"subscriptions"`で削除するグループを定義する．また，Sensu API を呼び出す際に必要な `"host"`や`"port"`，BASIC 認証を通過するための`"user"`や`"password"`も合わせて設定する．

5.3 Go 言語による Handler の開発

Sensu の Handler の要件として，以下の機能が必要とされている．

(1) 標準入力 (`STDIN`) として渡される Event データをデコードし，オブジェクトとして提供する．

(2) `/etc/sensu/conf.d` に配置されている JSON の設定を読み込み，オブジェクトとして提供する．

```
{
  "delete": {
    "status": 1,
    "subscriptions": [
      "virtual",
      "container"
    ],
    "host": "127.0.0.1",
    "port": 4567,
    "user": "",
    "password": ""
  }
}
```

図 9 Delete Handler の設定

Ruby を用いて Handler を開発する際は，公式に提供されている `sensu-handler` ライブラリを用いることで，これらの機能を簡単に実装することができる．しかし，Go 言語では Handler の開発をサポートするパッケージは提供されていない．そこで，(1) については JSON のエンコード・デコード機能を提供する `encoding/json` を，(2) については設定毎に JSON の構造が異なるため，インタラクティブに読み込む `github.com/bitly/go-simplejson` を用いて実現した．再利用性を高めるため，これらの機能を `sensu/handler` パッケージとして提供し，他の Handler の開発でも利用できるようにした．

この `sensu/handler` パッケージを用いて Delete Handler を開発した．

5.4 分散 Web システムにおける動作確認

分散 Web システムにおいて，Delete Handler の動作を確認するための実験を行った．仮想環境上に 9 台の Web サーバ (`web-server-1`〜`web-server-9`) と 9 台のクライアントを構築し，分散 Web システムに対して大量のアクセスを行った．同時アクセス数を増減させることで，スケールアウトやスケールインが実行され，Web サーバの起動・停止が発生する．Web サーバが停止した際に Delete Handler が実行され，その Client が Server の監視対象から削除されているか確認する．

その結果，図 10 に示す Sensu API のログ (`/var/log/sensu/sensu-api.log`) が得られた．`/clients/web-server-1` に対して `net/http` パッケージから `DELETE` メソッドが実行されたことを示しており，Delete Handler が正常に動作したことが確認できる．

```
{
  "level": "info",
  "message": "DELETE /clients/web-server-1",
  "remote_address": "127.0.0.1",
  "request_body": "",
  "request_method": "DELETE",
  "request_uri": "/clients/cain_web-server-1",
  "timestamp": "2015-05-26T16:40:56.226768+0900",
  "user_agent": "Go 1.1 package http"
}
```

図 10 Delete Handler 動作時の Sensu API のログ

6. おわりに

以上，Go 言語を用いてプラグインを開発することで監視のオーバーヘッドを抑える手法や，サーバ停止時に自動的に監視対象から削除する手法について述べた．評価の結果，Go 言語製プラグインは Community Plugin と比較して高速に動作し，安定してメトリクスを収集できた．また，指定したグループに属するサーバがオートスケールで停止した場合に，そのサーバを Sensu の監視対象から除外できることを確認した．

謝辞 本研究は JSPS 科研費 25330082 の助成を受けた．

参考文献

[1] 堀内晨彦，最所圭三，”クラウドに適した Web システムにおけるキャッシュサーバの負荷監視および負荷分散”，信学技報，vol.114，no110，IN2014-20，pp.79-84，2014 年 6 月
[2] Sensu Core, `https://sensuapp.org/`
[3] オブジェクト指向スクリプト言語 Ruby, `https://www.ruby-lang.org/`
[4] Nagios - The Industry Standard in IT Infrastructure Monitoring, `https://www.nagios.org/`
[5] Chef - Code Can, `https://www.chef.io/`
[6] Amazon EC2 (仮想クラウドサーバー), `http://aws.amazon.com/jp/ec2/`
[7] RabbitMQ - Messaging that just works, `http://www.rabbitmq.com/`
[8] Redis, `http://redis.io/`
[9] Elasticsearch: RESTful, Distributed Search & Analytics, `https://www.elastic.co/products/elasticsearch`
[10] Elastic Revealing Insights from Data, `https://www.elastic.co/`
[11] Kibana: Explore, Visualize, Discover Data, `https://www.elastic.co/products/kibana`
[12] The Go Programming Language, `http://golang.org/`