

Kubernetesで実践するクラウドネイティブDevOps

7章

2020/08/11

磯野 史弥

目次(1/2)

1. Kubectlコマンドの活用

- シェルエイリアス
- kubectlコマンドの省略形
- kubectlの自動補完
- ヘルプの表示
- JSONデータとjqの活用

2. リソースを使った作業

- 命令的なコマンドを使うべきではない場合
- リソースマニフェストの生成

3. コンテナを使った作業

- コンテナでのコマンド実行
- BusyBoxコマンドの使用
- kubesquashによるライブデバッグ

目次(2/2)

4. ContextとNamespace

- kubectlとkubens
- kube-ps1

5. kubernetesのシェルとツール

- kube-shell
- Click
- kubed-sh
- Stern
- kubespys
- 独自のクライアントツール

シェルエイリアス

- kubectlコマンドのシェルエイリアスを作成することで、作業を楽にすることができる

```
alias k=kubectl
```

- aliasの組み方はGoogleのエンジニアであるAhmet Alp Balkan氏のブログ上に一例が示されているので、参考にしてみてください。
 - <https://ahmet.im/blog/kubectl-aliases/>

kubectlコマンドの省略形

- kubectlコマンドは、flagの指定とリソースタイプの指定について、多くについて短縮形をサポートしている
- 例):フラグの指定の短縮系

```
kubectl get pods --namespace kube-system  
kubectl get pods -n kube-system
```

- 例): リソースタイプの省略形(※)

(※)

`kubectl(oc) api-resources` コマンドでそのクラスターで利用できるすべてのリソースタイプの短縮形を参照できる

kubectlの自動補完(1/2)

- bash/zshを利用している場合、kubectlコマンドの自動補完機能を使える
- kubectlコマンドを途中まで入力してTabキーを押すことで、コマンドの残りの部分が保管される(※)
- kubetctlコマンドを途中まで入力して1回TABキーを入力することで、コマンドの残り部分が保管される

```
kubectl cl<TAB> # kubectl cluster-infoと表示される
```

(※)

自動補完機能が有効になっていない場合、 `kubectl completion -h` コマンドを入力して表示される指示に従い有効にすることができる。

また、シェルエイリアスを設定した場合にも自動補完が聞かないため、追加設定をおこなう必要がある。

kubectlの自動補完(2/2)

- TABキーを2回入力することで利用できるすべてのコマンド・フラグを表示することができる
 - Pod、Deployment、Namespace等のオブジェクト名も自動補完される

```
kubectl <TAB><TAB> # 利用できるすべてのコマンドが表示される
kubectl get pods --<TAB><TAB> # 利用できるすべてのフラグが表示される
kubectl -n kube-system describe pods <TAB><TAB>
# Namespace:kube-system に構築されているすべてのPodを表示する
```

ヘルプの表示

- `kubectl -h` コマンドを入力することで利用できるコマンドと概要レベルの説明が表示される。
- `kubectl <COMMAND> -h` コマンドを入力することで、利用できるすべてのオプションや使用例を含めた詳細なドキュメンテーションを表示できる。
- `kubectctl explain <リソース名>` を入力することで、Kubernetesオブジェクトに関するヘルプ情報を参照することができる。

```
kubectl explain <リソース名>.<フィールド名> # 特定のフィールドに関する詳細情報を表示  
kubectl explain <リソース名> --recursive #特定のリソースにおけるフィールドを階層構造で表示
```


JSONデータとjqの活用(1/2)

- kubectl get podsコマンドにおいて、`-o json` フラグを入力することで、JSON形式で情報を出力することが出来る
 - ただしPodの数が増えてくると、出力が大量になり見づらい
- 1行程度の簡単なフィルタリングを行う際にはJSONPathを使うことができる
 - JSONPathはXPathと同等の手法でJSON形式のドキュメントから検索をおこなうためのツール。

```
kubectctl get pods -o=jsonpath='{.items[0].metadata.name}'  
# items配列の最初に格納されているPodの名前を表示する
```

JSONデータとjqの活用(2/2)

- JSONファイルに対して、より複雑なクエリおよびフィルタリングする際にはjqを利用することが好ましい
 - jqはこちらからダウンロードすることが出来る。
<https://stedolan.github.io/jq/>
 - jqクエリを試行錯誤する際には、以下のサイトを活用するとよい
<https://jqplay.org>

```
kubectl get pods -o json --all-namespaces | jq '.items |  
group_by(.spec.nodeName) | map({"nodeName": .[0].spec.nodeName,  
"count": length }) | sort_by(.count) | reverse'  
# 各ノードで実行しているPodの数を調べるためのjqクエリ
```

リソースを使った作業

- Kubernetesには宣言的なYAMLマニフェストを適用する宣言的(declarative)なコマンドと、リソースを直接的に作成または変更する命令的(imperative)なコマンドとの二種類が存在する。
- 宣言的コマンド
 - `kubectl apply`
- 命令的コマンド
 - `kubectl create`
 - `kubectl delete`
 - `kubectl edit`

命令的コマンドを使うべきではない場合(1/3)

- 命令的なコマンドの大きな問題は、唯一の信頼できる情報源(source of truth)が存在しないことである。
 - 命令的コマンドを実行したとたんに、クラスタの情報は管理しているマニフェストファイルと同期が取れなくなってしまう。
- 次に誰かがマニフェストファイルを適用する際に、命令的に行った変更は上書きされて失われ、想定外の影響を及ぼす可能性がある

命令的コマンドを使うべきではない場合(2/3)

ケーススタディ

- 管理しているサービスの負荷が突然高騰したため、運用保守の担当者はリクエストに応答するサーバーの数を10に増やした。
 - その際、デプロイされているReplicaSetを直接変更し、バージョン管理されている設定ファイルの変更をおこなわなかった。
- リリース担当者は、ReplicaSetの数が増えられていることに気づかずシステムのリリース作業をおこなってしまった。
- 結果、リリース後Podのレプリカ数は10→5に変更されてしまい、システムは過負荷状態となり障害が発生してしまった。

命令的コマンドを使うべきではない場合(3/3)

ケーススタディ

- 上述の例より以下の二つを教訓として得ることができる。
 - i. リソース変更時には、常にバージョン管理下のYAMLマニフェストを変更し `kubectl apply` コマンドで適用する。
 - ii. マニフェストファイルを適用する前には、 `kubectl diff` コマンドで新旧設定の差分を確認する。

```
kubectl diff -f deployment.yaml
- replicas : 10
+ replicas : 5
```

リソースマニフェストの生成

- YAMLマニフェストファイルを1から作る際には、命令的なコマンドを利用することで時間を大幅に節約することができる。

```
kubectl create --image=cloudfnated/demo:hello  
--dry-run -o yaml  
# 作成されるリソース(deployment)のマニフェストをyaml形式で出力する
```

- クラスタ内の既存のリソースに関するマニフェストファイルを作成する際には、以下のコマンドを入力する。

```
kubectl get deployments demo -o yaml --export > deployment.yaml  
# statusやmetadata.resourceVersionといったクラスタ固有のフィールドを  
# 取り除いたマニフェストを出力するコマンド(※)
```

(※)

代替となるコマンドが提示されないまま、

Kubernetes1.18で削除される予定だったが、削除されていない。

コンテナを使った作業(1/3)

- Kubernetesでは、コンテナが標準出力および標準エラー出力のストリームに書き込む情報はすべてログとしてみなされる。
 - また、コンテナアプリケーションを作成するときには、標準出力・標準エラー出力をストリーム上に出力するのがベストプラクティスとなる。
- 本番の分散型アプリケーションでは、複数のサービスからログを集約し永続的なデータベースに保存して、クエリ及びグラフ化を行えるようにする必要がある。
 - 詳細は15章で紹介。

コンテナを使った作業(2/3)

- `kubectl logs` コマンドにPodの名前を指定することで、特定のコンテナが出力するログメッセージを確認できる。

```
kubectl logs --tail=20 <pod名> # 指定したPodの最新20行を表示するコマンド  
kubectl logs <pod名> --follow # 指定したPodのログ出力をターミナルに随時表示させるコマンド(短縮系:-f)  
kubectl logs --container <container名> # 指定したcontainerのログを表示するコマンド(短縮形:-c)
```

- コンテナの出力を直接したい場合には、`kubectl attach` コマンドを利用する。
 - 本コマンドは様々なサイトに用途がわからないと記載されている。

```
kubectl attach <Pod名> -c <container名>
```

コンテナでのコマンド実行(3/3)

- 特定のPodに直接アクセスしたい場合、 `kubectl port-forward` コマンドでアクセスすることができる。
 - ブラウザでPodにアクセスするときに活用する。

```
kubectl port-forward <Pod名> <Host側PORT>:<Container側PORT>
```

- コンテナ内でシェルを実行する際には、 `kubectl exec` コマンドを使用する。
 - 主にトラブルシューティングの際に行う。

```
kubectl exec -it <POD名> /bin/sh -it  
# コンテナの対話型シェルをターミナルを通じて実行できるようにする  
oc rsh <POD名> # 上述のコマンドと同等の作業を行うocコマンド
```

Busyboxコマンドの使用(1/3)

- BusyBoxイメージは、一般的に利用されるUnixコマンドを豊富に搭載したイメージのことである。
 - 例): cat、echo、find、grep、kiss
 - 完全なリストは下記のURLより参照できる。

<https://busybox.net/downloads/BusyBox.html>

```
kubectl run busybox --image=busybox:1.28 --rm -it  
--restart=Never /bin/sh  
# 対話型シェルをターミナルを通じて実行できるようにする
```

Busyboxコマンドの使用(2/3)

- wgetやnslookupなどのコマンドをクラスタ内で実行して、アプリケーションが受け取る結果を確認できると便利な場合がある。

```
kubectl create deployment <Pod名> --image <イメージ名> # アプリケーションのデプロイ  
kubectl expose deployment <Pod名> --port 8888 # Deploymentを公開することでServiceを作成
```

- demo ServiceにIPアドレスとdemoというDNS名が割り当てられていること、demoに対してHTTPアクセスできることを確認する。

```
kubectl run <POD名> --image=busybox:1.28 --rm -it  
--restart=Never --command -- nslookup demo # 名前解決できるかの確認  
kubectl run <POD名> --image=busybox:1.28 --rm -it  
--restart=Never --command -- wget -q0- http://demo:8888 # HTTPアクセスできるかの確認
```

Busyboxコマンドの使用(3/3)

| オプション | 意味 |
|-----------------|----------------------------|
| --rm | 実行が完了したらPodを削除する指示 |
| -it | 対話型シェルをターミナルを通じて実行できるようにする |
| --restart=Never | コンテナが終了した場合に再起動しないように指示をする |
| --command -- | コンテナ内で実行するコマンドを指定する |

- `kubectl run busybox --image=busybox:1.28 --rm -it --restart=Never --command -`
- までもシェルエイリアスに登録しておくと便利。

BusyBoxのコンテナへの追加

- コンテナ内に対話型シェルがない場合、busyboxの実行可能コード(1MiB程度)をビルド時にコンテナ内にコピーすることで、イメージサイズを小さく保ったままデバッグ可能なコンテナを作成することができる。
 - 以下のようにDockerfileを記載することで、実現可能。

```
FROM golang:1.11-alpines AS build

WORKDIR /src/
COPY main.go go.* /src/
RUN CGO_ENABLED=0 go build -o /bin/demo

FROM scratch
COPY --from=build /bin/demo /bin/demo
COPY --from=busybox:1.28 /bin/busybox /bin/busybox
# 上行のコマンドでbusyboxの実行可能コードをコピーしている
ENTRYPOINT[ "/bin/demo" ]
```

kubesquashによるライブデバッグ

- gdbやdlv等のデバッガツールはプロセスにアタッチして、現在実行されているソースコード行の表示や、ローカル変数の値の検査、ブレイクポイントの設定、一行ずつの段階的なコード実行が可能。
 - ローカルマシンでプログラムを実行している場合には、そのプロセスに直接アクセスできる。
- デバッガをコンテナ内のプロセスに簡単にアタッチできるように設計されているのがsquashツールである。
 - Squashの紹介記事はこちら
https://qiita.com/go_vargo/items/df9084a080f66f9da0cc
 - インストール方法はGitHubのページに従うこと。
<https://github.com/solo-io/squash>
 - OpenShift上での動作もサポートしている。



ContextとNamespace

- Kubernetesクラスタが複数存在する場合、ユーザーが操作したいクラスタがどれかを判断する必要がある。
- kubectlにはContextという概念があり、特定のクラスタ・クラスタで認証するユーザー名・クラスタ内のNamespaceの組を参照する。



context

```
kubectl config get-contexts # contextの一覧を表示する
kubectl config use-context gke # contextを切り替える
kubectl config set-context <Context名> --cluster=<クラスタ名> --namespace=<Namespace名>
# 新しいコンテキストの作成
kubectl config current-context # 現在のコンテキストの表示
```


kubectxとkubens

- kubectxを使うことで高速でContextの変更をおこなうことができる。
 - OpenShift CLIでは、`oc login` が同等の用途で使える。
- Namespaceの切り替えは、kubensを使うと高速で行うことができる。
 - OpenShift CLIでは、`oc project/projects` が同等の用途で使える。
- 以下のサイトより、上述の二つのコマンドのdemoがGifで見れる。

<https://github.com/ahmetb/kubectx>

kube-ps1

- シェルとしてbash/zshを使用している場合、`kube-ps1` を使うことで現在のログインしている{Context名:Namespace名}をプロンプトに表示することができる。
 - 以下のサイトより利用イメージを確認できる。
<https://github.com/jonmosco/kube-ps1>
 - `oc` コマンドでも利用可能。

Kubernetesのシェルとツール

kube-shell

- Kubectlの自動補完機能を強化したラッパーツール。
 - ocコマンドへの対応はおこなっていない模様。
 - 詳細は以下のページを参照のこと。

<https://github.com/cloudnativelabs/kube-shell>


Click

- 一覧で表示されるリソースより番号でリソースを選択し利用できるツール。
 - 正規表現による検索もサポートしている。
 - ocコマンドへの対応はおこなっていない模様。
 - 詳細は以下のサイトを参照のこと。

<https://github.com/databricks/click>

Kubernetesのシェルとツール

Click

w:900

Kubernetesのシェルとツール

kubed-sh

- KubernetesクラスタにRuby,Python,Nodeがインストールされたpodを用意してそこでプログラムを実行するツール。
 - 以下の資料がわかりやすい
<https://qiita.com/sheepland/items/457bf350e7007d456b7f>
 - 本ツールは、以下のサイトよりダウンロードできる。
<https://kubed.sh/>
 - ocコマンドで利用できるようである。
<https://search.gocenter.io/github.com/mhausenblas/kubed-sh?tab=readme>

Kubernetesのシェルとツール

Stern

- Podの正規表現に一致するPod名 or ラベルを指定して、条件に該当するログの追従して表示する。
 - Pod内に複数のコンテナがある場合、個々のコンテナからのログメッセージにコンテナ名をプレフィックスとして追加する。
- 個別のPodが再起動されたとしてもログストリーミングを継続維持することができる。
 - 実行例は以下のページを参照のこと。
<https://qiita.com/zaki-lknr/items/189bae27d2f5320cdfd2>
 - ダウンロードは以下のページよりできる。
<https://github.com/wercker/stern>

Kubernetesのシェルとツール

kubespys

- クラスタ内の個別リソースを動的に表示するツール。
 - Kubernetesリソースの状態をリアルタイムで把握したい場合活用できる。
 - 個別のリソースの特定時点の状態を取得出来れば十分の場合、 `kubectl get` コマンドや、 `kubectl describe` コマンドを使えばよい。
 - 実行イメージは以下のサイトを確認いただきたい。
<https://github.com/pulumikubespys>
 - openshiftに対して適用できるという記載は発見できなかった。

Kubernetesのシェルとツール

独自のクライアントツール

- KubernetesAPIを呼び出すためフル機能クライアントライブラリが用意されている。
 - kubernetes用ライブラリー覧
<https://kubernetes.io/docs/reference/using-api/client-libraries/>
 - OpenShift用ライブラリー覧
<https://developers.redhat.com/openshift/REST-API-client-libraries>
- kubectl + Unixコマンド(+ jqなどのクエリ)で実現できない操作をおこなう場合などは、本ライブラリを使って実現することができる。

まとめ

- kubectlに関するドキュメントの確認方法
 - kubectl -h
 - kubectl explain
- JSONPath、jqを使ったフィルタリング
- マニフェストのひな形作成
 - `kubectl create --dry-run`
 - `kubectl get -o yaml --export`
- logファイルの確認
 - `kubectl logs -f`
 - Stern
 - `kubectl attach`

まとめ

- トラブルシュートのためのコマンド実行
 - `kubectl exec -it`
 - `BusyBox`
 - `squash`
- Context
 - `kubectx`
 - `kubens`
- Kubernetesシェル
 - `Kubectl`
 - `kube-shell`
 - `Click`
 - `kubed-sh`
 - `kubespys`