ORACLE
Cloud Infrastructure

Oracle Cloud Infrastructure上のKubernetes

概要および手動デプロイメント・ガイド

ORACLE WHITE PAPER | 2018年2月

ORACLE

免責事項

下記事項は、弊社の一般的な製品の方向性に関する概要を説明するものです。また、情報提供を唯一の目的とするものであり、いかなる契約にも組み込むことはできません。以下の事項は、マテリアルやコード、機能を提供することをコミットメント（確約）するものではないため、購買決定を行う際の判断材料になさらないで下さい。オラクルの製品に関して記載されている機能の開発、リリース、および時期については、弊社の裁量により決定されます。

目次

対象読者

本書は、Oracle Cloud InfrastructureにデプロイしたKubernetesの動作について概要を知りたい方、またはKubernetesに関する経験は十分にあり、高度な構成のクラスタを作成するための基本的なデプロイメント・プロセスを知りたい方を対象としています。

前提として、ユーザーはmacOSが稼働しているコンピュータを使用しており、UNIXのコマンドおよびファイル・システムについて理解していることとします。

本書をお使いになるには、Oracle Cloud Infrastructureの基本を十分に理解している必要があります。詳細は、https://docs.us-phoenix-1.oraclecloud.com/を参照してください。このプラットフォームを初めてお使いになる場合には、特にhttps://docs.us-phoenix-1.oraclecloud.com/Content/GSG/Reference/overviewworkflow.htmのチュートリアルをお読みになることをお薦めします。

はじめに

Kubernetesは、現在特に広く使用されているコンテナ・オーケストレーション・ツールです。まだ歴史が浅く、急成長の過程にあるオープンソース・プロジェクトですが、適切にデプロイすれば、本番環境でコンテナのワークロードを実行する信頼性の高いツールになります。

このドキュメントでは、Oracle Cloud Infrastructureで可用性の高い安全なKubernetesクラスタをデプロイするための最初の手順をご紹介します。本書の指示に従って作成したクラスタでも十分お役に立つはずですが、ここに示した以上のクラスタを構成したい場合には、各種のカスタマイズ・オプションと、Kubernetesへの更新に対応するために、追加のマテリアルとホワイトペーパーが必要になります。

Kubernetesは驚くほど動きのはやいプロジェクトで、リリースもバグの修正も頻繁に行われています。したがって、本書ではKubernetesバージョン1.6（本書執筆時点での安定版）のデプロイを扱うものとします。本書で作成するクラスタをアップグレードするプロセスについては、今後のドキュメントで扱います。

Kubernetesの概要

この項では、Kubernetesの概要について説明します。Kubernetesについてすでにご存じの方は、この項をスキップしていただいてかまいません。

Kubernetesは、現在特に広く使用されているコンテナ・オーケストレーション・ツールで、急成長中のオープンソース・コミュニティによって管理されています。Kubernetesプロジェクトは、長い間多数のコンテナを扱ってきたGoogleから生まれました。コンテナの管理性を向上させるために、コンテナ・オーケストレーションのシステムを開発する必要があったためです。

Kubernetesは、長年にわたってコンテナを使用してきたなかでGoogleが学んだ教訓を1つのツールにまとめたもので、コンテナ・オーケストレーションがシンプルになり、テクノロジー業界のさまざまなユース・ケースに適応します。2015年7月にオープンソース化されたばかりで、その機能は今も発展を続けています。問題点や新機能のリクエストは、公開のGitHubプロジェクトで追跡されており、新しいメジャー・バージョンはおよそ2か月ごとにリリースされます。

コンテナは、アプリケーションをインストールする際の依存性の欠落、あるいは特定のOSバージョンにアプリケーションをデプロイする際のトラブルといった問題を解決できるよう設計されています。コンテナ・オーケストレータは、このようなアプリケーションをスケーリングすることによって問題を解決しようとします。

コンテナ・イメージ

アプリケーションは、依存性とともに*コンテナ・イメージ*に保存されます。Dockerなどのコンテナ・エンジンによって実行されるとコンテナ・イメージはコンテナとして実行されます。アプリケーションにこのようなコンテナ・イメージを作成するプロセスを*コンテナ化*と言います。コンテナ化は、同じマシン上にデプロイされているアプリケーションどうしの対話が不十分な場合に役に立ちます。コンテナには一定レベルの分離機能があり、完全なマルチテナントではないものの、同じ物理ホスト上で他のアプリケーションが実行されている場合に別のアプリケーションが問題を起こすのを防ぐことができます。たとえば、アプリケーションの依存性（Java Runtime Environmentの特定のバージョンなど）を、アプリケーションとともにコンテナにバンドルすれば、Javaアプリケーションのデプロイが簡素になります。また、アプリケーションがLinuxで実行される場合、コンテナはLinux OSのフレーバーとバージョンを抽象します。OS上にあるアプリケーションの実行に必要な依存性をすべてコンテナにバンドルすることもできます。そのため、コンテナ化したアプリケーションは、Oracle LinuxでもUbuntu上と同じように動作します。

アプリケーションのデプロイメント

アプリケーションとその依存性をコンテナ・イメージにバンドルしたら、次のステップでは、必要としているすべてのグループにそのアプリケーションを配布します。多くのチームでこのアプリケーションを使用する必要があり、チームごとの使用スケールは時間とともに変化すると仮定しましょう。従来のデータ・センターであれば、このアプリケーションをスケーリングするには、次の四半期、場合によっては次の会計年度全期に関して全社的に、このアプリケーションの実行に必要になるリソースを予測する分析が必要になるところです。このような物理リソースを管理するIT組織は、業務上の必要性を満たすために新しい機器を注文する必要があるかもしれません。一方クラウドなら、このアプリケーションを実行する新しいリソースをオンデマンドで、またさまざまな規模で獲得できます（つまり、物理的なマシン1台ではなく仮想マシンでコア数を減らせます）。しかし、そのリソースに対するアプリケーションのデプロイメントを管理し、その管理を通じて業務上のニーズに対応することは依然として必要です。コンテナ・オーケストレーションは、この問題をシンプルに解決します。アプリケーションのコンテナ化によって、アプリケーションをさまざまな環境に簡単にデプロイでき、必要に応じてアプリケーションの管理とスケーリングが可能になります。

コンテナ・オーケストレーション機能

Kubernetesには、コンテナ化されたアプリケーションのオーケストレーション・タスクを実行する機能があります。

宣言的な管理

Kubernetesは、*命令的*ではなく*宣言的*な形での管理を前提に設計されています。命令的な管理の例が`apt-get install`コマンドを介したプログラムのインストールです。命令によって、特定の場所にアプリケーションをインストールします。プログラムの更新が必要になった場合も、命令的に`apt-get`を指定してアプリケーションを更新します。

宣言的な管理とは、*必要な状態*に基づいた管理です。*Deployment*は、Kubernetesでアプリケーションの状態を保持しておくためのコンポーネントです。DeploymentはJSONまたはYAMLで定義され、コンテナ・イメージのバージョンや、Kubernetesクラスタに存在すべきコンテナの数、アプリケーションを適切にデプロイするためにKubernetesが必要とするその他のプロパティなど、コンテナ・イメージに関する各種の情報を保持します。必要なアプリケーション・インスタンスの数を指定すれば、Kubernetesがリソース間で必要な数のコンテナを作成して管理します。なんらかの理由でコンテナに問題が生じた場合（たとえば、コンテナが稼働している仮想マシンがオフラインになった場合）は、Kubernetesが自動的に正常なノードで新しいコンテナを作成し、Deploymentで宣言された状態を維持します。Kubernetesは、実際の状態と必要な状態を継続的に比較してチェックし、2つの状態が合致するよう努めます。

宣言的なモデルは、必要なことをすべてシステムに指示する必要があるという点で、命令的なモデルより有利です。宣言的なモデルでは、必要なシステム状態の定義が存在するかぎり、Kubernetesがその指示を自動的に行います。

ローリング・アップグレードとロールバック

KubernetesのDeploymentを介してアプリケーションをデプロイするもうひとつのメリットが、組込みのローリング・アップグレードおよびロールバック機能です。たとえば、Deploymentの状態を定義するYAMLファイルが新しいバージョンのコンテナ・イメージによって更新されるときには、Kubernetesがこの変更を認識し、古いバージョンのインスタンスのシャットダウンを開始すると同時に、更新後のバージョンで新しいインスタンスを作成します。Kubernetesはこのローリング・アップグレードを実行すると同時に、実行中のコンテナ・インスタンスにリクエストを送り続けるので、通常は停止時間が発生しません。同様に、アップグレードで問題が発生した場合にも、Kubernetesが必要に応じてロールバックを実行します。

ロード・バランサ

Kubernetes内のコンテナとしてデプロイされたアプリケーションへの接続を管理するために、Kubernetes Serviceコンポーネントが Kubernetesクラスタ内にソフトウェア定義のロード・バランサを提供します。たとえば、Javaアプリケーションの3つのインスタンスのデプロイメントに、Kubernetes Serviceの1つのポイントからアクセスすることもできます。そのため、1つのインスタンスが使用できなくなった場合や、ローリング・アップグレードが実行される場合でも、このアプリケーションへのアクセスはそのまま続行できます。

こうしたKubernetesの基本コンポーネントを利用することで、リソースのプール間でコンテナ化された多数のアプリケーションのオーケストレーションが円滑に進みます。必要に応じてコンテナ・オーケストレーションを最適に利用する方法については、Kubernetesのドキュメント（https://kubernetes.io/docs）を参照してください。

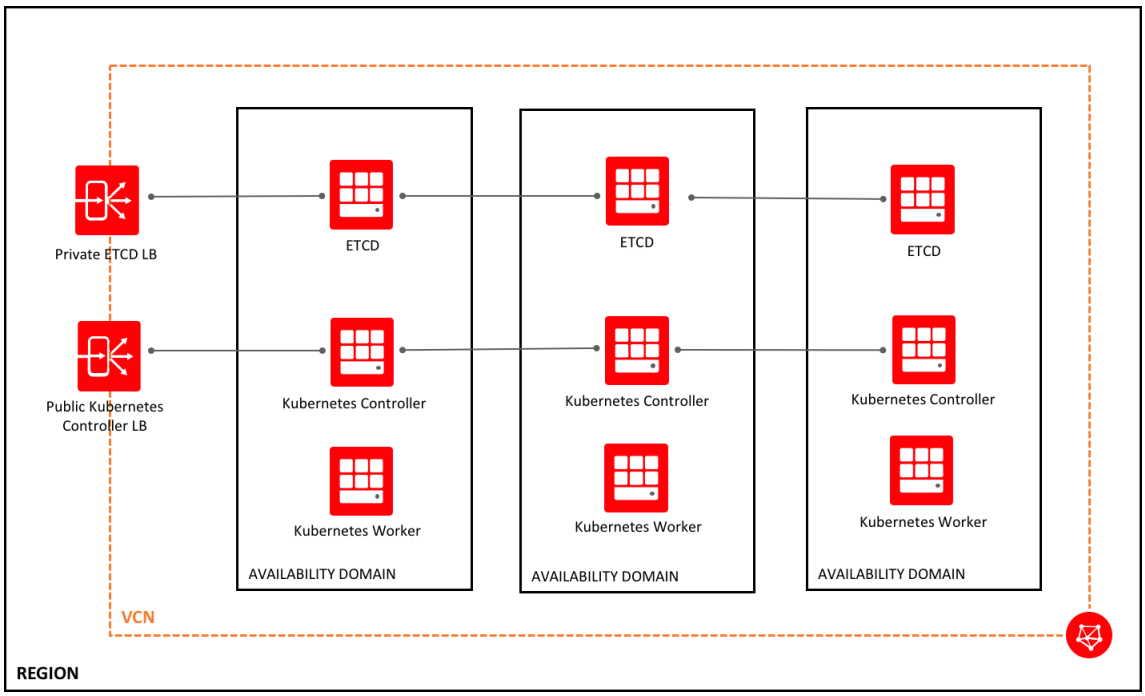
Oracle Cloud InfrastructureにおけるKubernetes手動デプロイメントのクラスタ・アーキテクチャ

この項では、実行中のKubernetesクラスタを構成するコンポーネントを紹介し、クラスタを構成しているコンピュート・リソース間でそのコンポーネントをデプロイする方法について説明します。ここで説明するアーキテクチャは、「Oracle Cloud InfrastructureへのKubernetes手動デプロイメント・ガイド」（次の項）でデプロイします。

Kubernetesのクラスタは、etcd、Kubernetesマスター（コントローラ）、Kubernetesワーカー（ノード）という3つのコンポーネントで構成されています。このガイドでは、以下のアーキテクチャで高可用性（HA）のKubernetesクラスタを作成する方法を説明します。

- 3つのetcdノード（1つのリージョンで3つの可用性ドメインにわたる）
- 3つのKubernetesマスター（コントローラ）（1つのリージョンで3つの可用性ドメインにわたる）
- 3つのKubernetesワーカー（ノード）（1つのリージョンで3つの可用性ドメインにわたる）

このガイドの手順に従って生成されるインフラストラクチャは、次の図のようになります。etcdクラスタは、Kubernetesクラスタの別々のコンピュート・リソース・セット上で稼働するように構成されます。



各コンポーネントの詳細は以下のとおりです。

etcd

etcdは、CoreOSによって作成されるキー/値ストアです。Kubernetesの状態情報がetcdクラスタに格納されます。Kubernetesを介してetcdクラスタを実行することと混同しないようにしてください。そうではなく、このetcdクラスタを利用してKubernetes自体が実行されます。

このガイドで、etcdクラスタは、Kubernetesクラスタの別々のコンピュート・リソース・セット上で稼働するように構成されます。etcdを別々のコンピュート・リソース上で実行することで、etcdと、Kubernetesクラスタの各コンポーネントの間で確実な分離が保証されます。

Kubernetesマスター

Kubernetesマスター（コントローラ）は、クラスタのAPIサーバー、コントローラ・マネージャ、およびスケジューラ・コンポーネントを実行するマシン（仮想または物理）です。

Kubernetesワーカー

Kubernetesワーカー（ノード）は、KubernetesクラスタのKubernetesコンポーネントを実行するマシン（仮想または物理）です。ワーカーは、Kubernetesがコンテナ（ポッド）をスケジュールするリソースです。

Oracle Cloud InfrastructureへのKubernetes手動デプロイメント・ガイド

このガイドでは、Oracle Cloud InfrastructureでKubernetesを実行する際に必要なすべてのコンポーネントと機能をデプロイして構成する方法を説明します。「クリーン・スレート」環境から始めると想定し、ガイドを通じてOracle Cloud Infrastructureで必要なすべてのリソースを作成することとします。

説明するタスクは以下のとおりです。

1. Oracle Cloud Infrastructureリソースを作成します。
2. Kubernetesコンポーネントの証明書を生成します。
3. HA etcdクラスタを起動します。
4. コンポーネントを認証するトークンと構成ファイルを生成します。
5. HA Kubernetesマスター（Kubernetesコントロール・プレーン）を起動します。
6. ワーカーを追加します。
7. リモート・アクセスを構成します。
8. Kube-DNSをデプロイします。
9. スモーク・テストを実行します。

ステップ1：Oracle Cloud Infrastructureリソースの作成

このガイドでは、Kubernetesクラスタを作成するために必要なOracle Cloud Infrastructureリソースの作成方法については*説明しません*。必要なリソースはリストして示しますが、Oracle Cloud Infrastructureコンソール、CLI、API、SDK、あるいはTerraformプロバイダを使用してご自身で作成してください。手順については、Oracle Cloud Infrastructureのドキュメントを参照してください。

ネットワーキング

このガイドで示すKubernetesクラスタを作成するには、次のNetworkingコンポーネントが必要です。Networkingコンポーネントの作成については、Oracle Cloud Infrastructureドキュメントの Networkingの項を参照してください。

VCN

次の値をもつ1つの仮想クラウド・ネットワーク（VCN）が必要です。

VCN名	CIDR	説明
k8sOCI.oraclevcn.com	10.0.0.0/16	Kubernetesクラスタのネットワーク・リソースのホスティングに使用されるVCN

サブネット

1つのVCNと、可用性ドメインごとに1つ以上のサブネット（合計3つ）のサブネットが必要です。本番構成では、可用性ドメインごとにetcd、マスター、ワーカーのサブネットを1つずつ作成することをお薦めします。合計でサブネットは9つになります。後で削除するテストまたは学習用のデプロイメントであれば、作成するサブネットは3つで十分です（可用性ドメインごとに1つ）。可用性ドメインが3つのリージョンのクラスタで推奨される構成の値を以下に示します。

次の値を使用します。

サブネット名	CIDR	可用性ドメイン	説明
publicETCDSubnetAD1.sub	10.0.20.0/24	AD1	AD1でetcdホストに使用されるサブネット
publicETCDSubnetAD2.sub	10.0.21.0/24	AD2	AD2でetcdホストに使用されるサブネット
publicETCDSubnetAD3.sub	10.0.22.0/24	AD3	AD3でetcdホストに使用されるサブネット
publicK8SMasterSubnetAD1.sub	10.0.30.0/24	AD1	AD1でKubernetesに使用されるサブネット
publicK8SMasterSubnetAD2.sub	10.0.31.0/24	AD2	AD2でKubernetesに使用されるサブネット
publicK8SMasterSubnetAD3.sub	10.0.32.0/24	AD3	AD3でKubernetesに使用されるサブネット
publicK8SWorkerSubnetAD1.sub	10.0.40.0/24	AD1	AD1でホストKubernetesワーカーに使用されるサブネット
publicK8SWorkerSubnetAD2.sub	10.0.41.0/24	AD2	AD2でホストKubernetesワーカーに使用されるサブネット
publicK8SWorkerSubnetAD3.sub	10.0.42.0/24	AD3	AD3でホストKubernetesワーカーに使用されるサブネット

セキュリティ・リスト

本番構成では、次のセキュリティ・リストが必要になります。

- etcd_security_list
- k8sMaster_security_list
- k8sWorker_security_list

各セキュリティ・リストで推奨されるセキュリティ・ルールの一覧は、「付録A：セキュリティ・ルール」を参照してください。

インターネット・ゲートウェイとルート表

お使いになる構成では、Kubernetesクラスタがインターネット・ゲートウェイを通じてインターネットにアクセスできるように、1つのインターネット・ゲートウェイと、1つのルート表ルールが必要です。

ルート表ルールは、宛先CIDRブロックを`0.0.0.0/0`、ターゲット・タイプを`internet gateway`とします。ターゲットは、Kubernetesクラスタで使用するインターネット・ゲートウェイです。

ロード・バランサ

推奨されるクラスタ構成には、2つのロード・バランサが必要です。etcdノード用にプライベート・ロード・バランサを1つ、Kubernetesマスター用にパブリック・ロード・バランサを1つ作成します。次の表をロード・バランサの情報で埋めながら、ガイドを読み進めてください。

このガイドでは、Kubernetesマスターのロード・バランサのパブリックアドレスを`loadbalancer_public_ip`と表します。次の値を使用します。

ロード・バランサ名	ロード・バランサ・タイプ	サブネット1	サブネット2	パブリックIP	プライベートIP	コンピュート・リソースのバック・エンド
lb-etcd	プライベート		該当せず			Etcd1、Etcd2、Etcd3
lb-k8smaster	パブリック					KubeM1、KubeM2、KubeM3

コンピュート

このガイドで使用するOracle Cloud Infrastructureのリソースを追跡しやすいように、次の表を埋めてください。コンピュート・リソースには、任意のComputeインスタンス・シェイプを使用できます。このテクノロジーについて調べながらこのガイドを読み進める場合には、VM.Standard1.1やVM.Standard1.2のように小さなVMシェイプを選択することをお薦めします。

Computeインスタンスの作成については、Oracle Cloud InfrastructureドキュメントのComputeの項を参照してください。

Computeインスタンスの名前	Computeインスタンスのシェイプ	可用性ドメイン	サブネット	プライベートIPアドレス	パブリックIPアドレス	Kubernetesクラスタにおけるロール
Etcd1		AD1				etcdノード
Etcd2		AD2				etcdノード
Etcd3		AD3				etcdノード
KubeM1		AD1				Kubernetesマスター
KubeM2		AD2				Kubernetesマスター
KubeM3		AD3				Kubernetesマスター
KubeW1		AD1				Kubernetesワーカー
KubeW2		AD2				Kubernetesワーカー
KubeW3		AD3				Kubernetesワーカー

このガイドでは、前述の表の値として以下の名前を使用します。

Computeインスタンスの名前	プライベートIPアドレス	パブリックIPアドレス
Etcd1	etcd1_private_ip	etcd1_public_ip
Etcd2	etcd2_private_ip	etcd2_public_ip
Etcd3	etcd3_private_ip	etcd3_public_ip
KubeM1	kubem1_private_ip	kubem2_public_ip
KubeM2	kubem2_private_ip	kubem2_public_ip

Computeインスタンスの名前	プライベートIPアドレス	パブリックIPアドレス
KubeM3	kubem3_private_ip	kubem3_public_ip
KubeW1	kubew1_private_ip	kubew1_public_ip
KubeW2	kubew2_private_ip	kubew2_public_ip
KubeW3	kubew3_private_ip	kubew3_public_ip

ステップ2：認証局の設定とTLS証明書の作成

このステップには、CFSSLのインストール、証明書の作成、証明書ファイルのホストへのコピーという3パートがあります。

CFSSLのインストール

Kubernetesクラスタで使用する証明書を生成するには、CloudFlareのTLS証明書生成ツール、CFSSLを使用します。ローカル・コンピュータ（macOSが稼働している）で、この項のコマンドを実行してください。

1. curlを使用してcfsslパッケージをダウンロードします。
```
curl -O https://pkg.cfssl.org/R1.2/cfssl_darwin-amd64
```
2. ディレクトリに対する権限を変更してcfssl実行可能ファイルを作成します。
```
chmod +x cfssl_darwin-amd64
```
3. cfsslを`/usr/local/bin`に移動してパスに追加します。
```
sudo mv cfssl_darwin-amd64 /usr/local/bin/cfssl
```
4. curlを使用して`cfssljson`バイナリをダウンロードします。
```
curl -O https://pkg.cfssl.org/R1.2/cfssljson_darwin-amd64
```
5. ディレクトリに対する権限を変更して`cfssljson`実行可能ファイルを作成します。
```
chmod +x cfssljson_darwin-amd64
```
6. `cfssljson`をパスに移動します。
```
sudo mv cfssljson_darwin-amd64 /usr/local/bin/cfssljson
```

証明書の作成

この項では、クラスタに必要な証明書の設定について説明します。ローカル・コンピュータ（macOSが稼働している）で、この項のコマンドを実行してください。

1. 認証局（CA）を記述した構成ファイルを作成します。このファイルは、クラスタに関連するすべての鍵の認証に使用されます。

```
cat > ca-config.json <<EOF
{
  "signing": {
    "default": {
      "expiry": "8760h"
    },
    "profiles":
      { "kubernetes":
      {
        "usages": ["signing", "key encipherment", "server auth", "client
auth"],
        "expiry": "8760h"
      }
    }
  }
}
```

2. CA証明書の署名リクエストを作成します。

```
cat > ca-csr.json <<EOF
{
  "CN": "Kubernetes",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "Kubernetes",
      "OU": "CA",
      "ST": "Oregon"
    }
  ]
}
EOF
```

3. CA証明書と秘密鍵を生成します。

```
cfssl gencert -initca ca-csr.json | cfssljson -bare ca
```

次のファイルが作成されます。

- ca-key.pem
- ca.pem

4. Kubernetesワーカー・ノードごとに、クライアントおよびサーバーTLC証明書の署名リクエストを作成します。`$(instance)`は、扱っている個々のKubernetesワーカーの名前に置き換えてください（kubeW1、kubeW2、kubeW3など）。

```
for instance in worker-0 worker-1 worker-2; do
cat > ${instance}-csr.json <<EOF
{
  "CN": "system:node:${instance}",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "system:nodes",
      "OU": "Kubernetes The Hard
      Way", "ST": "Oregon"
    }
  ]
}
EOF
```

5. Kubernetesワーカー・ノードごとに、証明書と秘密鍵を生成します。次のコマンドをワーカーごとに1回ずつ実行します。`$(instance)`は、扱っている個々のKubernetesワーカーの名前に置き換えてください（kubeW1、kubeW2、kubeW3など）。`$(EXTERNAL_IP)`および`${INTERNAL_IP)`は、扱っているワーカーのパブリック（外部）IPアドレスおよびプライベート（内部）IPアドレスに置き換えます。

```
cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -hostname=${instance},${EXTERNAL_IP},${INTERNAL_IP} \
  -profile=kubernetes \
  ${instance}-csr.json | cfssljson -bare
${instance} done
```

次のファイルが作成されます。

- kubeW1-key.pem
- kubeW1.pem
- kubeW2-key.pem
- kubeW2.pem
- kubeW3-key.pem
- kubeW3.pem

6. 管理者のロール・ベース・アクセス制御（RBAC）証明書の署名リクエストを作成します。管理者クライアント証明書は、管理ロールを通じてAPIサーバー（マスター）に接続するときに使用されます。それによって、KubernetesのネイティブRBACのもとで特定の権限が許可されます。

```
cat > admin-csr.json <<EOF
{
  "CN": "admin",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "system:masters",
      "OU": "Cluster",
      "ST": "Oregon"
    }
  ]
}
EOF
```

7. 管理者クライアント証明書と秘密鍵を生成します。

```
cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  admin-csr.json | cfssljson -bare admin
```

次のファイルが作成されます。

- admin-key.pem
- admin.pem

8. kube-proxyクライアント証明書の署名リクエストを作成します。この証明書セットは、kube-proxyがKubernetesマスターに接続する際に使用されます。

```
cat > kube-proxy-csr.json <<EOF
{
  "CN": "system:kube-proxy",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "system:node-proxier",
      "OU": "Cluster",
```

```
      "ST": "Oregon"
    }
  ]
}
EOF
```

9. kube-proxyクライアント証明書と秘密鍵を生成します。

```
cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  kube-proxy-csr.json | cfssljson -bare kube-proxy
```

次のファイルが作成されます。

- kube-proxy-key.pem
- kube-proxy.pem

10. Kubernetesサーバー証明書を作成します。`kubeMn_private_ip`はお使いのマスターのプライベートIPアドレスに、`loadbalancer_public_ip`はロード・バランサのIPアドレスに、それぞれ置き換えてください。

```
cat > kubernetes-csr.json <<EOF
{
  "CN": "Kubernetes",
  "key": [
    "10.32.0.1",
     “kubeM1_private_ip",
     “kubeM2_private_ip” ,
     “kubeM3_private_ip” ,
    "loadbalancer_public_ip",
    "127.0.0.1",
    "kubernetes.default"
  ],
  "key":
    { "algo":
    "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Portland",
      "O": "Kubernetes",
      "OU": "Cluster",
      "ST": "Oregon"
    }
  ]
}
```

11. Kubernetesサーバー証明書と秘密鍵を生成します。

```
cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
kubernetes-csr.json | cfssljson -bare kubernetes
```

次のファイルが作成されます。

- kubernetes-key.pem
- kubernetes.pem

12. 以下の手順で、etcd証明書を作成します。

A. 次のプロジェクトをダウンロードし、手順に従って各etcdノードの証明書を作成します。**config**にある**req-csr.pem**ファイルには、必ずetcdノードのプライベートIPアドレスを入力してください。

https://github.com/coreos/etcd/tree/v3.2.1/hack/tls-setup

B. **cert**ディレクトリから**etcd-certs**という独自のディレクトリにetcd証明書をコピーします。このパスは後から必要になるので、ディレクトリは覚えやすい場所にしてください（**~/etcd-certs**など）。

C. `rename`コマンドを使用して、etcd用であることがわかるようにetcdの認証局の名前を変更します。

```
brew install rename
rename 's/ca/etcd-ca/' *
```

証明書ファイルのホストへのコピー

1. 次のようなスクリプトを使用して、必要な証明書ファイルをホストにコピーします。*node*_public_IPは、ワーカー、マスター、およびetcdノードのパブリックIPアドレスに置き換えてください。

ファイル名：`copyCAs.sh`

```
for host in 'kubeW1_public_IP' 'kubeW2_public_IP' 'kubeW3_public_IP'; do
  scp -i ~/.ssh/id_rsa ca.pem kube-proxy.pem kube-proxy-key.pem
ubuntu@${host}:~/
done
for host in 'kubeM1_public_IP' 'kubeM2_public_IP' 'kubeM3_public_IP' ;
  do scp -i ~/.ssh/id_rsa ca.pem ca-key.pem kubernetes-key.pem
  kubernetes.pem
ubuntu@${host}:~/
done
for host in 'etcd1_public_IP' 'etcd2_public_IP' 'etcd3_public_IP' ;
```

```
done
```

2. 個々のワーカー証明書を、対応するワーカーにコピーします。ワーカーごとに、etcd証明書、kubeconfigファイル、ca.pemファイル、および独自の証明書がすべて必要です。

ファイルがワーカーにコピーされると、各ワーカーには次のようなファイルが含まれるようになります。

```
bootstrap.kubeconfig etcd1.csr      etcd1.pem etcd2-key.pem etcd3.csr      etcd3.pem etcd-ca-key.pem
kube-proxy-key.pem    kube-proxy.pem  kubew2-csr.json kubew2.kubeconfig proxy1.csr      proxy1.pem

ca.pem              etcd1-key.pem etcd2.csr etcd2.pem       etcd3-key.pem etcd-ca.csr etcd-ca.pem
kube-proxy.kubeconfig kubew2.csr      kubew2-key.pem  kubew2.pem         proxy1-key.pem
```

これらのファイルをまとめるために、etcd証明書を保管する**etcd-certs**というディレクトリを作成します。このディレクトリを作成し、適切なファイルを移動するには、次のコマンドを使用します。

```
mkdir etcd-certs
mv *.* etcd-
```

ステップ3：HA etcdクラスタの起動

このステップには、Dockerのインストール、etcdディレクトリの作成、etcdクラスタのプロビジョニングという3パートがあります。この項のコマンドは、別途の指示がないかぎり、etcdノードで実行してください。

全ノードでのDockerのインストール

このガイドでは、**get.docker.com**で公開されているDockerインストール・スクリプトを使用します。Dockerのインストールには、好みに応じて任意の方法をお使いください。

1. curlを使用してスクリプトをダウンロードします。

```
curl -fsSL get.docker.com -o get-docker.sh
```

2. 次のコマンドを実行してDockerをインストールします。

```
sh get-docker.sh
```

etcdディレクトリの作成

次のコマンドを実行して、etcdがデータの格納に使用するディレクトリを作成します。

```
sudo mkdir /var/lib/etcd
```

注意：etcdクラスタの作成中に問題があり、etcdサーバーの再デプロイが必要になった場合には、このディレクトリを削除する必要があります。そうしないと、etcdは古い構成で起動しようとします。削除するには、次のコマンドを使用してください：`sudo rm -rf /var/lib/etcd`

etcdクラスタのプロビジョニング

1. 各etcdノードに次の変数を設定します。`NAME_n`および`HOST_n`は、`CLUSTER`変数に必要な情報を提供する変数です。`HOST_n`変数については、`etcdn_private_ip`をお使いのノードのプライベートIPアドレスに置き換えてください。

```
NAME_1=etcd1
NAME_2=etcd2
NAME_3=etcd3
HOST_1= etcd1_private_ip
HOST_2= etcd2_private_ip
HOST_3= etcd3_private_ip

CLUSTER=${NAME_1}=https://${HOST_1}:2380,${NAME_2}=https://${HOST_2}:2380,
${NAME_3}=https://${HOST_3}:2380
DATA_DIR=/var/lib/etcd
ETCD_VERSION=latest
CLUSTER_STATE=new
THIS_IP=$(curl http://169.254.169.254/opc/v1/vnics/0/privateIp)
THIS_NAME=$(hostname)
```

2. iptableを削除します。

   KubernetesとetcdIは、設定中にiptableに変更を加えます。このステップでは、必要に応じてetcdで設定できるように、iptableを完全に削除します。

   注意：この手順にはセキュリティ上のリスクが伴います。この手順を実行する前に、ネットワーク・リソースのセキュリティ・ルールが十分に確立していることを確認してください。

```
sudo iptables -F
```

3. 次のコードを各ノードでそのまま実行して、etcdコンテナを起動します。

```
sudo docker run -d -p 2379:2379 -p 2380:2380 --volume=${DATA_DIR}:/etcd-
data  --volume=/home/ubuntu/etcd-certs:/etc/etcd  --net=host  --name  etcd
quay.io/coreos/etcd:${ETCD_VERSION} /usr/local/bin/etcd --name
${THIS_NAME}  --cert-file=/etc/etcd/${THIS_NAME}.pem  --key-
file=/etc/etcd/${THIS_NAME}-key.pem  --peer-cert-
file=/etc/etcd/${THIS_NAME}.pem  --peer-key-file=/etc/etcd/${THIS_NAME}-
key.pem  --trusted-ca-file=/etc/etcd/etcd-ca.pem  --peer-trusted-ca-
file=/etc/etcd/etcd-ca.pem  --peer-client-cert-auth  --client-cert-auth  -
-initial-advertise-peer-urls https://${THIS_IP}:2380  --listen-peer-
urls  --listen-client-urls  --advertise-client-urls  --initial-
cluster-
token etcd-cluster-0  --initial-cluster ${CLUSTER} --initial-cluster-
state new             --data-dir=/etcd-data
```

注意：次のエラーが発生した場合は、この手順のステップ1で設定した変数の一部または全部が欠落しています。すべての変数が正しく設定されているかどうか確認してください。

```
"docker: invalid reference format"
```

4. 任意のetcdノードで次のコマンドを実行して、クラスタが正常にデプロイされたことを確認します。

```
sudo docker exec etcd etcdctl --ca-file=/etc/etcd/etcd-ca.pem  --cert-
file=/etc/etcd/${THIS_NAME}.pem  --key-file=/etc/etcd/${THIS_NAME}-
key.pem cluster-health
```

5. 検証がうまく進まない場合は、etcdファイルを削除して再試行してください。

```
sudo docker stop etcd
sudo docker rm etcd
sudo rm -rf
```

ステップ4：RBACの設定

macOSが稼働しているローカル・コンピュータで、この項のコマンドを実行します。

1. kubectlをダウンロードしてインストールします。

```
curl -O https://storage.googleapis.com/kubernetes-
release/release/v1.6.0/bin/darwin/amd64/kubectl
chmod +x kubectl
sudo mv kubectl /usr/local/bin
```

2. 次のように、TLSブートストラップ・トークンを作成して配布します。

   A. トークンを生成します。

```
BOOTSTRAP_TOKEN=$(head -c 16 /dev/urandom | od -An -t x | tr -d ' ')
```

   B. トークン・ファイルを生成します。

```
cat > token.csv <<EOF
${BOOTSTRAP_TOKEN},kubelet-bootstrap,10001,"system:kubelet-bootstrap"
EOF
```

   C. 各マスターにトークンを配布します。`controller`*n*は、各Kubernetesマスターのパブリック IPアドレスに置き換えてください。

```
for host in controller0 controller1 controller2; do
  scp -i ~/.ssh/id_rsa token.csv ubuntu@${host}:~/
done
```

3. ブートストラップkubeconfigファイルを作成します。

```
kubectl config set-cluster kubernetes-the-hard-way \
  --certificate-authority=ca.pem \
  --embed-certs=true \
```

```
  --server=https://${LB_IP}:6443 \
  --kubeconfig=bootstrap.kubeconfig

kubectl config set-credentials kubelet-bootstrap \
  --token=${BOOTSTRAP_TOKEN} \
  --kubeconfig=bootstrap.kubeconfig

kubectl config set-context default \
  --cluster=kubernetes-the-hard-way \
  --user=kubelet-bootstrap \
  --kubeconfig=bootstrap.kubeconfig

kubectl config use-context default --kubeconfig=bootstrap.kubeconfig
```

4. kube-proxy kubeconfigファイルを作成します。

```
kubectl config set-cluster kubernetes-the-hard-way \
  --certificate-authority=ca.pem \
  --embed-certs=true \
  --server=https://${LB_IP}:6443 \
  --kubeconfig=kube-proxy.kubeconfig

kubectl config set-credentials kube-proxy \
  --client-certificate=kube-proxy.pem \
  --client-key=kube-proxy-key.pem \
  --embed-certs=true \
  --kubeconfig=kube-proxy.kubeconfig

kubectl config set-context default \
  --cluster=kubernetes-the-hard-way \
  --user=kube-proxy \
  --kubeconfig=kube-proxy.kubeconfig

kubectl config use-context default --kubeconfig=kube-proxy.kubeconfig
```

5. クライアントkubeconfigファイルを配布します。`kubeW`*`n`*`_public_ip`は、各ワーカー・ノードのパブリックIPアドレスに置き換えてください。

```
for host in ‘kubeW1_public_ip’ ‘kubeW2_public_ip’ ‘kubeW3_public_ip’;
  do scp -i ~/.ssh/id_rsa bootstrap.kubeconfig kube-proxy.kubeconfig
ubuntu@${host}:~/
done
```

ステップ5：HA Kubernetesコントロール・プレーンの起動

Kubernetesマスターをプロビジョニングします。マスターで次のコマンドを実行します。

1. ブートストラップ・トークンを次の場所にコピーします。

```
sudo mkdir -p /var/lib/kubernetes/
sudo mv token.csv /var/lib/kubernetes/
```

2. 「ステップ2：認証局の設定とTLS証明書の作成」で、必要な証明書をKubernetesマスターにコピーしていない場合は、ここでコピーします。必要なのは、ca.pem、ca-key.pem、kubernetes-key.pem、およびkubernetes.pemの証明書です。

   Kubernetes APIサーバー（マスター上）、kubectl（他のマシンからKubernetesを制御するために使用）、kubelet（ワーカー上）の間の通信を確保するために、ステップ2で作成したTLS証明書が使用されます。Kubernetes APIサーバーとetcdの間の通信も、ステップ2で作成したTLS証明書によって保護されます。

3. TLS証明書を、Kubernetes構成ディレクトリにコピーします。

```
sudo mv ca.pem ca-key.pem kubernetes-key.pem kubernetes.pem
/var/lib/kubernetes/
```

4. wgetを使用して、Kubernetesバイナリの公式リリースをダウンロードします。

```
wget https://storage.googleapis.com/kubernetes-
release/release/v1.7.0/bin/linux/amd64/kube-apiserver
wget https://storage.googleapis.com/kubernetes-
release/release/v1.7.0/bin/linux/amd64/kube-controller-manager
wget https://storage.googleapis.com/kubernetes-
release/release/v1.7.0/bin/linux/amd64/kube-scheduler
wget
https://storage.googleapis.com/kubernetes-
```

5. Kubernetesバイナリをインストールします。

```
chmod +x kube-apiserver kube-controller-manager kube-scheduler kubectl
sudo mv kube-apiserver kube-controller-manager kube-scheduler kubectl
/usr/bin/
```

Kubernetes APIサーバー

1. 証明書を適切にまとめておけるように、次のコマンドを使用して、すべてのetcd証明書を専用のディレクトリに配置します。

```
sudo mkdir /var/lib/etcd
sudo cp *.* /var/lib/etcd/.
```

2. マシンの内部IPアドレスを取得します。

```
INTERNAL IP=$(curl http://169.254.169.254/opc/v1/vnics/0/privateIp)
```

3. iptableを削除します。

   KubernetesとetcdはiptableにIに変更を加えます。このステップでは、必要に応じてetcdで設定できるように、iptableを完全に削除します。

注意：この手順にはセキュリティ上のリスクが伴います。この手順を実行する前に、ネットワーク・リソースのセキュリティ・ルールが十分に確立していることを確認してください。

```
sudo iptables -F
```

4. Kubernetes APIサーバーのsystemdユニット・ファイルを作成します。このファイルは、Kubernetes APIサーバーをsystemdサービスとして管理するよう、Ubuntuのsystemdに指示します。

```
cat > kube-apiserver.service <<EOF
[Unit]
Description=Kubernetes API Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStart=/usr/bin/kube-apiserver \\
  --admission-
control=NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorageClass,
ResourceQuota \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --audit-log-maxage=30 \\
  --audit-log-maxbackup=3 \\
  --audit-log-maxsize=100 \\
  --audit-log-path=/var/lib/audit.log \\
  --authorization-mode=RBAC \\
  --bind-address=${INTERNAL_IP} \\
  --client-ca-file=/var/lib/kubernetes/ca.pem \\
  --enable-swagger-ui=true \\
  --etcd-cafile=/var/lib/etcd/etcd-ca.pem \\
  --etcd-certfile=/var/lib/etcd/etcd3.pem \\
  --etcd-keyfile=/var/lib/etcd/etcd3-key.pem \\
  --etcd-servers=https://<etcd1 private ip>:2379,https://<etcd2 private
ip>:2379,https://<etcd3 private ip>:2379 \\
  --event-ttl=1h \\
  --experimental-bootstrap-token-auth \\
  --insecure-bind-address=0.0.0.0 \\
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\
  --kubelet-client-certificate=/var/lib/kubernetes/kubernetes.pem \\
  --kubelet-client-key=/var/lib/kubernetes/kubernetes-key.pem \\
  --kubelet-https=true \\
  --runtime-config=rbac.authorization.k8s.io/v1alpha1 \\
  --kubelet-preferred-address-
types=InternalIP,ExternalIP,LegacyHostIP,Hostname \\
  --service-account-key-file=/var/lib/kubernetes/ca-key.pem \\
  --service-cluster-ip-range=10.32.0.0/16 \\
  --service-node-port-range=30000-32767 \\
  --tls-cert-file=/var/lib/kubernetes/kubernetes.pem \\
  --tls-private-key-file=/var/lib/kubernetes/kubernetes-key.pem \\
  --token-auth-file=/var/lib/kubernetes/token.csv \\
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

5. kube-apiserverサービスを起動します。

```
sudo mv kube-apiserver.service /etc/systemd/system/kube-apiserver.service
sudo systemctl daemon-reload
sudo systemctl enable kube-apiserver
sudo systemctl start kube-apiserver
sudo systemctl status kube-apiserver --no-pager
```

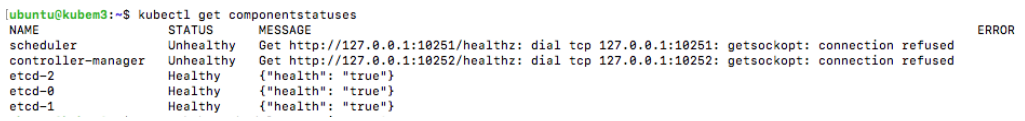
6. サービスでエラーがレポートされた場合は、次のコマンドを使用してデバッグし、必要なポートにバインドされているかどうかを確認します。

```
journalctl -xe
netstat -na | more
netstat -na | grep 6443
```

7. 次のコマンドを実行して、Kubernetes APIサーバーが稼働していることを確認します。

```
kubectl get componentstatuses
```

コマンドからの出力は、次のようになるはずです。



Kubernetesスケジューラ

1. Kubernetesスケジューラのsystemdユニット・ファイルを作成します。このファイルは、Kubernetesスケジューラをsystemdサービスとして管理するよう、Ubuntuのsystemdに指示します。

```
kube-scheduler.service
cat > kube-scheduler.service <<EOF
[Unit]
Description=Kubernetes Scheduler
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
[Service]
ExecStart=/usr/bin/kube-scheduler \\
  --leader-elect=true \\
--master=http://${INTERNAL_IP}:8080 \\
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

2. kube-schedulerサービスを起動します。

```
sudo mv kube-scheduler.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable kube-scheduler
sudo systemctl start kube-scheduler
sudo systemctl status kube-scheduler --no-pager
```

3. 次のコマンドを実行して、Kubernetesスケジューラが稼働していることを確認します。

```
kubectl get componentstatuses
```

コマンドからの出力は、次のようになるはずです。

```
[ubuntu@kubem3:~$ kubectl get componentstatuses
NAME                 STATUS      MESSAGE                                                                                     ERROR
scheduler            Healthy     ok
controller-manager   Unhealthy   Get http://127.0.0.1:10252/healthz: dial tcp 127.0.0.1:10252: getsockopt: connection refused
etcd-1               Healthy     {"health": "true"}
etcd-2               Healthy     {"health": "true"}
etcd-0               Healthy     {"health": "true"}
```

Kubernetesコントローラ・マネージャ

1. Kubernetesコントローラ・マネージャのsystemdユニット・ファイルを作成します。このファイルは、Kubernetesコントローラ・マネージャをsystemdサービスとして管理するよう、Ubuntuのsystemdに指示します。

```
cat > kube-controller-manager.service <<EOF
[Unit]
Description=Kubernetes Controller Manager
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStart=/usr/bin/kube-controller-manager \\
  --address=0.0.0.0 \\
  --allocate-node-cidrs=true \\
  --cluster-cidr=10.200.0.0/16 \\
  --cluster-name=kubernetes \\
  --cluster-signing-cert-file=/var/lib/kubernetes/ca.pem \\
  --cluster-signing-key-file=/var/lib/kubernetes/ca-key.pem \\
  --leader-elect=true \\
  --master=http://${INTERNAL_IP}:8080 \\
  --root-ca-file=/var/lib/kubernetes/ca.pem \\
  --service-account-private-key-file=/var/lib/kubernetes/ca-key.pem \\
  --service-cluster-ip-range=10.32.0.0/16 \\
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
EOF
```

2. kube-controller-managerサービスを起動します。

```
sudo mv kube-controller-manager.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable kube-controller-manager
sudo systemctl start kube-controller-manager
sudo systemctl status kube-controller-manager --no-pager
```

3. 次のコマンドを実行して、Kubernetesコントローラ・マネージャが稼働していることを確認します。

```
kubectl get componentstatuses
```

コマンドからの出力は、次のようになるはずです。

```
[ubuntu@kubem1:~$ kubectl get componentstatuses
NAME                 STATUS    MESSAGE             ERROR
controller-manager   Healthy   ok
scheduler            Healthy   ok
etcd-1               Healthy   {"health": "true"}
etcd-0               Healthy   {"health": "true"}
etcd-2               Healthy   {"health": "true"}
```

ステップ6：ワーカーの追加

このステップには、各ワーカーに対するkubeconfigファイルの生成、kube-proxy用のkubeconfigの生成、ワーカー上でのファイルの構成、複数のコンポーネントのインストールというパートがあります。

各ワーカーに対するkubeconfigの生成

各ワーカーのノード名に一致するクライアント証明書（「ステップ2：認証局の設定とTLS証明書の作成」で作成）を必ず使用してください。

次のスクリプトを使用して、各ワーカー・ノードにkubeconfigを生成します。

```
for instance in worker-0 worker-1 worker-2; do
  kubectl config set-cluster kubernetes-the-hard-way \
    --certificate-authority=ca.pem \
    --embed-certs=true \
    --server=https://${LB_IP}:6443 \
    --kubeconfig=${instance}.kubeconfig

  kubectl config set-credentials system:node:${instance} \
    --client-certificate=${instance}.pem \
    --client-key=${instance}-key.pem \
    --embed-certs=true \
    --kubeconfig=${instance}.kubeconfig

  kubectl config set-context default \
    --cluster=kubernetes-the-hard-way \
    --user=system:node:${instance} \
    --kubeconfig=${instance}.kubeconfig

  kubectl config use-context default --kubeconfig=${instance}.kubeconfig
done
```

次のファイルが作成されます。

- worker-0.kubeconfig
- worker-1.kubeconfig
- worker-2.kubeconfig

Kube-Proxy用のkubeconfigファイルの生成

次のコマンドを使用して、kube-proxyがマスターへの接続に使用するkubeconfigファイルを生成します。

```
kubectl config set-cluster kubernetes-the-hard-way \
  --certificate-authority=ca.pem \
  --embed-certs=true \
  --server=https://${LB_IP}:6443 \
  --kubeconfig=kube-proxy.kubeconfig
```

```
kubectl config set-credentials kube-proxy \
  --client-certificate=kube-proxy.pem \
  --client-key=kube-proxy-key.pem \
  --embed-certs=true \
  --kubeconfig=kube-proxy.kubeconfig
```

```
kubectl config set-context default \
  --cluster=kubernetes-the-hard-way \
  --user=kube-proxy \
  --kubeconfig=kube-proxy.kubeconfig
```

```
kubectl config use-context default --kubeconfig=kube-proxy.kubeconfig
```

ワーカー上でのファイルの構成

各ワーカーにログインし、次のファイル移動コマンドを実行します。

```
sudo mkdir /var/lib/kubernetes/
sudo mkdir /var/lib/kubelet/
sudo mkdir /var/lib/kube-proxy/
sudo mv bootstrap.kubeconfig /var/lib/kubelet
sudo mv kube-proxy.kubeconfig /var/lib/kube-
proxy sudo mv ca.pem /var/lib/kubernetes/
sudo mv $(hostname)-key.pem $(hostname).pem /var/lib/kubelet/
sudo mv $(hostname).kubeconfig /var/lib/kubelet/kubeconfig
sudo mv ca.pem /var/lib/kubernetes/
```

Flannelのインストール

1. 次のコマンドを使用してFlannelをインストールします。

   ```
   wget https://github.com/coreos/flannel/releases/download/v0.6.2/flanneld-
   amd64 -O flanneld && chmod 755 flanneld
   sudo mv flanneld /usr/bin/flanneld
   ```

2. 次の`/etc/systemd/system/flanneld.service`ファイルを作成してflannelサービスを構成します。

```
[Unit]
Description=Flanneld overlay address etcd agent
[Service]
Type=notify
EnvironmentFile=-/usr/local/bin/flanneld
ExecStart=/usr/bin/flanneld -ip-masq=true -iface 10.0.0.11 --etcd-
cafile=/home/ubuntu/etcd-ca.pem --etcd-certfile=/home/ubuntu/etcd3.pem --
etcd-keyfile=/home/ubuntu/etcd3-key.pem --etcd-
endpoints=https://10.0.0.7:2379,https://10.0.1.7:2379,https://10.0.2.6:237
9 --etcd-prefix=/kube/network
Restart=on-failure
```

3. flannelサービスを起動します。

```
sudo systemctl daemon-reload
sudo systemctl restart flanneld
sudo systemctl enable flanneld
sudo systemctl status flanneld --no-pager
```

最後は、サービスのステータスをチェックするコマンドで、出力は次の例のようになります。

```
ubuntu@kubew1:~$ sudo systemctl status flanneld --no-pager
  flanneld.service - Flanneld overlay address etcd agent
   Loaded: loaded (/etc/systemd/system/flanneld.service; static; vendor
preset: enabled)
   Active: active (running) since Fri 2017-09-15 17:40:06 UTC; 1s ago
 Main PID: 1904 (flanneld)
    Tasks: 10
   Memory: 10.6M
      CPU: 292ms
   CGroup: /system.slice/flanneld.service
           └─1904 /usr/bin/flanneld -ip-masq=true -iface 10.0.0.11 --
etcd- cafile=/home/ubuntu/etcd-ca.pem --etcd-
certfile=/home/ubuntu/etcd3.pem -- etcd-keyfile=/home/ubuntu/etcd3-key.pem
--etcd- endpoints=https://10.0.0.7:2379,https://10.0.1...

Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.283241 01904 ipmasq.go:47] Adding iptables rule: ! -
s 10.200.0.0/16 -d 10.200.0.0/16 -j MASQUERADE
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.285301 01904 manager.go:246] Lease acquired: 10.200.63.0/24
Sep 15 17:40:06 kubew1 systemd[1]: Started Flanneld overlay address etcd
agent.
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.285666 01904 network.go:58] Watching for L3 misses
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.285698 01904 network.go:66] Watching for new subnet leases
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.297509 01904 network.go:153] Handling initial subnet events
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.297538 01904 device.go:163] calling GetL2List()
dev.link.Index: 3
```

```
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.297617 01904 device.go:168] calling
NeighAdd: 10.0.0.9, 86:64:67:a6:29:e5
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.297682 01904 device.go:168] calling NeighAdd: 10.0.2.9,
e6:76:2b:98:c6:ab
Sep 15 17:40:06 kubew1 flanneld[1904]:
I0915 17:40:06.297728 01904 device.go:168] calling NeighAdd: 10.0.0.8,
a6:67:22:ab:2f:8b
```

Container Networking Interface（CNI）のインストール

1. CNIサービスのために次のファイルを作成します。

   /etc/systemd/system/cni-bridge.service

   ```
   [Unit]
   Requires=network.target
   Before=docker.service
   [Service]
   Type=oneshot
   ExecStart=/usr/local/bin/cni-bridge.sh
   RemainAfterExit=true
   ```

   /usr/local/bin/cni-bridge.sh

   ```
   #!/bin/bash

   set -x

   /sbin/ip link add name cni0 type bridge
   /sbin/ip addr add $(grep '^FLANNEL_SUBNET' /run/flannel/subnet.env | cut
   - d= -f2) dev cni0
   /sbin/ip link set dev cni0 up
   ```

2. 次のブロックのコマンドを実行し、CNIサービスをインストールして開始します。

   ```
   sudo su
   mkdir -p /opt/cni/bin /etc/cni/net.d
   chmod +x /usr/local/bin/cni-bridge.sh
   curl -L --
   retry 3 https://github.com/containernetworking/cni/releases/download/v0.5.
   2/cni-amd64-v0.5.2.tgz -o /tmp/cni-plugin.tar.gz
   tar zxf /tmp/cni-plugin.tar.gz -C /opt/cni/bin/
   printf '{\n    "name": "podnet",\n    "type": "flannel",\n    "delegate":
   {\n        "isDefaultGateway": true\n    }\n}\n' >/etc/cni/net.d/10-
   flannel.conf
   chmod +x /usr/local/bin/cni-bridge.sh
   systemctl enable cni-bridge && systemctl start cni-bridge
   exit
   ```

3. 次のコマンドを実行して、サービスのステータスを確認します。

   ```
   sudo systemctl status cni-bridge
   ```

出力は、次の例のようになります。

```
ubuntu@kubew3:~$ sudo systemctl status cni-bridge.service
  cni-bridge.service
   Loaded: loaded (/etc/systemd/system/cni-bridge.service; static; vendor
preset: enabled)
   Active: active (exited) since Fri 2017-09-15 18:20:25 UTC; 27s ago
  Process: 1940 ExecStart=/usr/local/bin/cni-bridge.sh (code=exited,
status=0/SUCCESS)
 Main PID: 1940 (code=exited, status=0/SUCCESS)

Sep 15 18:20:25 kubew3 systemd[1]: Starting cni-bridge.service...
Sep 15 18:20:25 kubew3 cni-bridge.sh[1940]: + /sbin/ip link add name cni0
type bridge
Sep 15 18:20:25 kubew3 cni-bridge.sh[1940]: ++ grep '^FLANNEL_SUBNET'
/run/flannel/subnet.env
Sep 15 18:20:25 kubew3 cni-bridge.sh[1940]: ++ cut -d= -f2
Sep 15 18:20:25 kubew3 cni-bridge.sh[1940]: + /sbin/ip addr add
10.200.63.1/24 dev cni0
Sep 15 18:20:25 kubew3 cni-bridge.sh[1940]: + /sbin/ip link set dev cni0
up
Sep 15 18:20:25 kubew3 systemd[1]: Started cni-bridge.service.
```

```
ubuntu@kubew1:~$ ifconfi
cni0      Link encap:Ethernet  HWaddr 32:2a:0a:be:35:a2
          inet addr:10.200.63.1  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::302a:aff:febe:35a2/64
          Scope:Link UP BROADCAST RUNNING MULTICAST
                                       MTU:1500
                                       Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
```

Dockerのインストール

1. 次のようにDockerをインストールします。

```
wget https://get.docker.com/builds/Linux/x86_64/docker-1.12.6.tgz
tar -xvf docker-1.12.6.tgz
sudo cp docker/docker* /usr/bin/
```

2. 次の`/etc/systemd/system/flanneld.service`ファイルを作成してDockerサービスを構成します。

```
 [Unit]
Description=Docker Application Container Engine
Documentation=http://docs.docker.io
After=network.target firewalld.service cni-bridge.service
Requires=cket cni-bridge.service

[Service]
ExecStart=/usr/bin/dockerd \
  --bridge=cni0 \
  --iptables=false \
  --ip-masq=false \
```

```
  --host=unix:///var/run/docker.sock \
  --insecure-registry=registry.oracledx.com \
  --log-level=error \
  --storage-driver=overlay
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

3. Dockerサービスを起動します。

```
sudo systemctl daemon-reload
sudo systemctl enable docker
sudo systemctl start docker
sudo docker version
sudo docker network inspect bridge
```

4. 次のコマンドを実行して、Dockerが適切に構成されていることを確認します。

```
sudo docker network inspect bridge
```

出力では、Dockerがcni0ブリッジを使用するよう構成されていることが示されます。

Kubeletのインストール

1. 次のコマンドを使用して、kubeletバージョン1.7.4をダウンロードします。

```
wget -q --show-progress --https-only --
timestamping https://storage.googleapis.com/kubernetes-
release/release/v1.7.4/bin/linux/amd64/kubectl
wget -q --show-progress --https-only --
timestamping https://storage.googleapis.com/kubernetes-
release/release/v1.7.4/bin/linux/amd64/kube-proxy
wget -q --show-progress --https-only --
timestamping https://storage.googleapis.com/kubernetes-
release/release/v1.7.4/bin/linux/amd64/kubelet
```

2. Kubeletをインストールします。

```
chmod +x kubectl kube-proxy kubelet
sudo mv kubectl kube-proxy kubelet /usr/bin/
```

3. 次の`/etc/systemd/system/kubelet.service`ファイルを作成します。

```
/etc/systemd/system/kubelet.service
[Unit]
Description=Kubernetes Kubelet
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=docker.service
Requires=docker.service


[Service]
ExecStart=/usr/bin/kubelet \
  --allow-privileged=true \
```

```
  --cluster-dns=10.32.0.10 \
  --cluster-domain=cluster.local \
  --container-runtime=docker \
  --image-pull-progress-deadline=2m \
  --kubeconfig=/var/lib/kubelet/kubeconfig \
  --require-kubeconfig=true \
  --network-plugin=cni \
  --pod-cidr=10.200.88.0/24 \
  --serialize-image-pulls=false \
  --register-node=true \
  --runtime-request-timeout=10m \
  --tls-cert-file=/var/lib/kubelet/kubew2.pem \
  --tls-private-key-file=/var/lib/kubelet/kubew2-key.pem \
  --hostname-override= kubew2.sub08071631352.kubevcn.oraclevcn.com \
  --v=2
Restart=on-failure
RestartSec=5


[Install]
WantedBy=multi-user.target
```

4. iptableを削除してから、kubeletサービスを起動します。

```
iptables -F
```

5. kubeletサービスを起動します。

```
sudo systemctl daemon-reload
sudo systemctl enable kubelet
sudo systemctl start kubelet
sudo systemctl status kubelet --no-pager
```

kube-proxyのインストール

1. kube-proxy用に、次の`/etc/systemd/system/kube-proxy.service`ファイルを作成します。

```
 [Unit]
Description=Kubernetes Kube Proxy
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStart=/usr/bin/kube-proxy \
  --cluster-cidr=10.200.0.0/16 \
  --kubeconfig=/var/lib/kube-proxy/kube-proxy.kubeconfig \
  --proxy-mode=iptables \
  --v=2
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

2. kube-proxyサービスを起動します。

```
sudo systemctl daemon-reload
sudo systemctl enable kube-
proxy sudo systemctl start
kube-proxy
```

ワーカーの削除

ワーカーの削除が必要になった場合は、次のコマンドで削除できます。このとき、`nodename`は、`kubectl get nodes`からのKubernetesワーカー・ノード名に置き換えてください。

1. ノードから、ポッドがあれば削除します。

```
kubectl drain nodename
```

2. ワーカー・ノードを削除します。

```
kubectl delete node nodename
```

注意：この操作でインスタンスは削除されません。Oracle Cloud Infrastructure Computeインスタンスを削除したい場合は、CLIまたはコンソールを介して削除する必要があります。

ステップ7：リモート・アクセスのためのkubectlの構成

kubectlは、Kubernetesクラスタの制御と管理に使用するコマンドライン・ツールです。ローカル・コンピュータにkubectlをインストールして構成すると、クラスタなどリモート・ロケーションにログインしてクラスタに管理するかわりに、お使いのコンピュータを通じて簡単にKubernetesクラスタを管理できるようになります。ローカル以外のコンピュータからKubernetesクラスタを管理したい場合は、そのコンピュータで次のステップを実行してください。

このステップを実行すると、Oracle Cloud Infrastructureでクラスタに接続できます。`${LB_IP}`をロード・バランサのIPアドレスに置き換えたうえで、ローカル・コンピュータで次のコマンドを実行します。

```
kubectl config set-cluster kubernetes-the-hard-way \
  --certificate-authority=ca.pem \
  --embed-certs=true \
  --server=https://${LB_IP}:6443

kubectl config set-credentials admin \
  --client-certificate=admin.pem \
  --client-key=admin-key.pem

kubectl config set-context kubernetes-the-hard-way \
  --cluster=kubernetes-the-hard-way \
  --user=admin

kubectl config use-context kubernetes-the-hard-way
```

ステップ8：Kube-DNSのデプロイ

1. kube-dnsクラスタ・アドオンをデプロイします。

```
kubectl create -f https://storage.googleapis.com/kubernetes-the-hard-way/kube-dns.yaml
```

出力は、次の例のようになります。

```
Service account "kube-dns" created configmap "kube-dns" created service "kube-dns" created deployment "kube-dns" created
```

2. kube-dnsデプロイメントで作成されたポッドをリストします。

```
kubectl get pods -l k8s-app=kube-dns -n kube-system
```

出力は、次の例のようになります。

```
NAME                        READY     STATUS    RESTARTS   AGE
kube-dns-3097350089-gq015   3/3       Running   0          20s
kube-dns-3097350089-q64qc   3/3       Running   0          20s
```

ステップ9：スモーク・テストの実行

この項では、クラスタが予定どおりに動作していることを確認する簡単なスモーク・テストについて説明します。

1. 次のコマンドを使用して、3つのレプリカをもつnginxデプロイメントを作成します。

```
kubectl run nginx --image=nginx --port=80 --replicas=3
```

出力は、次のようになります。

```
deployment "nginx" created
```

2. 次のコマンドを実行して、デプロイメントによって作成されたポッドを確かめ、稼働中の状態であることを確認します。

```
kubectl get pods -o wide
```

出力は、次の例のようになります。

```
NAME                    READY     STATUS    RESTARTS   AGE       AGE           AGE
nginx-158599303-btl44   1/1       Running   0          18s       10.99.49.5    k8s-worker-ad1-0
nginx-158599303-ndxtc   1/1       Running   0          18s       10.99.49.3    k8s-worker-ad2-0
nginx-158599303-r2801   1/1       Running   0          18s       10.99.49.4    k8s-worker-ad3-0
```

3. nginxデプロイメントに接続するサービスを作成します。

```
kubectl expose deployment nginx --type NodePort
```

4. 作成したサービスを表示します。Oracle Cloud Infrastructureでは現在`--type=LoadBalancer`がサポートされていないことに注意してください。

```
kubectl get service
NAME         CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes   10.21.0.1      <none>        443/TCP        1h
nginx        10.21.62.159   <nodes>       80:30050/TCP   1h
```

現時点では、クラスタにルーティングされるロード・バランサを*手動*でBMCに作成する（この例では、10.21.62.159:80）か、Oracle Cloud Infrastructureのセキュリティ・リストを介してノード・ポート（この場合、30050）を公開します。このガイドでは、後者を実行します。

5. ワーカーのセキュリティ・リストを変更し、30050へのイングレス・トラフィックを許可します。

6. nginxサービスに設定されていた`NodePort`を取得します。

```
NodePort=$(kubectl get svc nginx --output=jsonpath='{range
.spec.ports[0]}{.nodePort}')
```

7. UIから、ワーカーの1つの`worker_public_ip`値を取得します。

8. 次のcurlを使用して、nginxサービスをテストします。

```
curl http://${worker_public_ip}:${NodePort}
```

出力は、次の例のようになります。

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed
and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

付録A：セキュリティ・ルール

この付録では、次のセキュリティ・リストのリスト・ルールについて概要を示します。

- **etcd**：etcd_security_list.sl
- **Kubernetesマスター**：k8sMaster_security_list.sl
- **Kubernetesワーカー**：k8sWorker_security_list.sl

セキュリティ・リストそれぞれの特定のイングレスおよびエグレス・ルールを、次の表にまとめました。ルールはすべて、ステートフルです。

etcdセキュリティ・リストのイングレス・ルール

ソース	IPプロトコル	ソース・ポート範囲	宛先ポート範囲
10.0.0.0/16	TCP	すべて	すべて
10.0.0.0/16	TCP	すべて	22
10.0.0.0/16	TCP	すべて	2379-2380

etcdセキュリティ・リストのエグレス・ルール

宛先	IPプロトコル	ソース・ポート範囲	宛先ポート範囲
0.0.0.0/0	すべて	すべて	すべて

Kubernetesマスター・セキュリティ・リストのイングレス・ルール

宛先	IPプロトコル	ソース・ポート範囲	宛先ポート範囲
10.0.0.0/16	すべて	すべて	すべて
10.0.0.0/16	TCP	すべて	3389
10.0.0.0/16	TCP	すべて	6443
10.0.0.0/16	TCP	すべて	22

Kubernetesマスター・セキュリティ・リストのエグレス・ルール

宛先	IPプロトコル	ソース・ポート範囲	宛先ポート範囲
0.0.0.0/0	すべて	すべて	すべて

Kubernetesワーカー・セキュリティ・リストのイングレス・ルール

宛先	IPプロトコル	ソース・ポート範囲	宛先ポート範囲
10.0.0.0/16	TCP	すべて	すべて
10.0.0.0/16	TCP	すべて	22
10.0.0.0/16	UDP	すべて	30000-32767

Kubernetesワーカー・セキュリティ・リストのエグレス・ルール

宛先	IPプロトコル	ソース・ポート範囲	宛先ポート範囲
0.0.0.0/0	すべて	すべて	すべて

Oracle Corporation, World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries
Phone: +1.650.506.7000
Fax：+1.650.506.7200

CONNECT WITH US

blogs.oracle.com/oracle

facebook.com/oracle

twitter.com/oracle

oracle.com

Integrated Cloud Applications & Platform Services

Oracle Cloud Infrastructure上のKubernetes
2018年2月
著者：Kaslin Fields