Computação Paralela e Sistemas Distribuídos
Tutorial – Como montar um cluster

Autores: Frederico Oliveira Freitas
Luan Felipe Ribeiro Santos
Talles Souza Silva

Introdução

Um cluster é um termo em inglês que significa aglomerar, um cluster nada mais é que uma combinação de vários computadores que pode ser aplicado em vários contextos. É um conjunto de computadores que são combinados para aumentar o processamento assim executando uma tarefa em menos tempo.

No trabalho a seguir vamos estudar alguns tipos de clusters como banco de dados distribuídos, processamentos distribuídos e kubernetes.

Banco de dados distribuído

Primeiro, abordaremos como o DBInputFormat interage com os bancos de dados. O DBInputFormat usa o JDBC para conectar-se a fontes de dados. Como o JDBC é amplamente implementado, o DBInputFormat pode trabalhar com MySQL, PostgreSQL e vários outros sistemas de banco de dados. Fornecedores de banco de dados individuais fornecem drivers JDBC para permitir que aplicativos de terceiros (como o Hadoop) se conectem a seus bancos de dados. Links para drivers populares estão listados na seção de recursos no final deste post.

Para começar a usar o DBInputFormat para conectar-se ao seu banco de dados, você precisará baixar o driver de banco de dados apropriado da lista na seção de recursos (veja o final deste post) e soltá-lo no diretório ***$ HADOOP_HOME / lib /*** no seu ***Hadoop TaskTracker*** e na máquina onde você inicia seus trabalhos.

Lendo tabelas com DBInputFormat

O DBInputFormat é uma classe InputFormat que permite ler dados de um banco de dados. Um InputFormat é a formalização do Hadoop de uma fonte de dados; pode significar arquivos formatados de uma maneira particular, dados lidos de um banco de dados, etc. DBInputFormat fornece um método simples de varrer tabelas inteiras a partir de um banco de dados, assim como os meios de ler consultas SQL arbitrárias executadas no banco de dados. A maioria das consultas é suportada, sujeita a algumas limitações discutidas no final deste artigo.

Configurando o trabalho

Para usar o DBInputFormat, você precisará configurar seu trabalho. O exemplo a seguir mostra como se conectar a um banco de dados MySQL e carregar de uma tabela:

```
CREATE TABLE employees (
  employee_id INTEGER NOT NULL PRIMARY KEY,
  name VARCHAR(32) NOT NULL);
```

Listagem 1: esquema de tabela de exemplo

```
JobConf conf = new JobConf(getConf(), MyDriver.class);
conf.setInputFormat(DBInputFormat.class);
DBConfiguration.configureDB(conf,
    “com.mysql.jdbc.Driver”,
    “jdbc:mysql://localhost/mydatabase”);
String [] fields = { “employee_id”, "name" };
DBInputFormat.setInput(conf, MyRecord.class, “employees”,
    null /* conditions */,  “employee_id”, fields);
// set Mapper, etc., and call JobClient.runJob(conf);
```

Listagem 2: Código Java para configurar uma tarefa MapReduce com DBInputFormat

Este código de exemplo se conectará ao ***mydatabase*** no localhost e lerá os dois campos da tabela ***employees*** .

As chamadas ***configureDB ()*** e ***setInput ()*** configuram o DBInputFormat. A primeira chamada especifica a implementação do driver JDBC a ser usada e a qual banco de dados se conectar. A segunda chamada especifica quais dados serão carregados do banco de dados. A classe **MyRecord** é a classe na qual os dados serão lidos em Java e "***employees***" é o nome da tabela a ser lida. O parâmetro "***employee_id***" especifica a chave primária da tabela, usada para ordenar os resultados. A seção “Limitações do InputFormat” abaixo explica por que isso é necessário. Finalmente, a matriz de campos lista quais colunas da tabela devem ser lidas. Uma definição sobrecarregada de ***setInput ()*** permite que você especifique uma consulta SQL arbitrária para ler, em vez disso.

Depois de chamar o ***configureDB ()*** e ***setInput ()***, você deve configurar o restante do trabalho como de costume, definindo as classes ***Mapper*** e ***Reducer***, especificando quaisquer outras fontes de dados para ler (por exemplo, conjuntos de dados no HDFS) e outros parâmetros específicos do trabalho.

Recuperando os dados

O método ***setInput ()*** usado no exemplo acima tomou como parâmetro o nome de uma classe que conterá o conteúdo de uma linha. Você precisará escrever uma implementação da interface ***DBWritable*** para permitir que o ***DBInputFormat*** preencha sua classe com campos da tabela. ***DBWritable*** é uma interface de adaptador que permite que os dados sejam lidos e gravados usando o mecanismo de serialização interna do Hadoop e usando chamadas JDBC. Depois que os dados forem lidos em sua classe personalizada, você poderá ler os campos da classe no mapeador.

O exemplo a seguir fornece uma implementação ***DBWritable*** que contém um registro da tabela ***employees***, conforme descrito acima:

```
class MyRecord implements Writable, DBWritable {
  long id;
  String name;

  public void readFields(DataInput in) throws IOException {
    this.id = in.readLong();
    this.name = Text.readString(in);
  }

  public void readFields(ResultSet resultSet)
      throws SQLException {
    this.id = resultSet.getLong(1);
    this.name = resultSet.getString(2);
  }

  public void write(DataOutput out) throws IOException {
    out.writeLong(this.id);
    Text.writeString(out, this.name);
  }

  public void write(PreparedStatement stmt) throws SQLException {
    stmt.setLong(1, this.id);
    stmt.setString(2, this.name);
  }
}
```

Listagem 3: Implementação de DBWritable para registros da tabela employees

Um objeto ***java.sql.ResultSet*** representa os dados retornados de uma instrução SQL. Ele contém um cursor que representa uma única linha dos resultados. Esta linha conterá os campos especificados na chamada ***setInput ()*** . No método ***readFields ()*** do ***MyRecord***, lemos os dois campos do ***ResultSet***. Os métodos ***readFields ()*** e ***write ()*** que operam nos objetos ***java.io.DataInput*** e ***DataOutput*** fazem parte da interface gravável usada pelo Hadoop para empacotar dados entre mapeadores e redutores, ou empacotar resultados em ***SequenceFiles.***

Usando os dados em um mapeador

O mapeador então recebe uma instância da sua implementação ***DBWritable*** como seu valor de entrada. A chave de entrada é um id de linha fornecido pelo banco de dados; você provavelmente descartará esse valor.

```
public class MyMapper extends MapReduceBase
    implements Mapper<LongWritable, MyRecord, LongWritable, Text> {
  public void map(LongWritable key, MyRecord val,
      OutputCollector<LongWritable, Text> output, Reporter reporter) throws IOException {
    // Use val.id, val.name here
    output.collect(new LongWritable(val.id), new Text(val.name));
  }
}
```

Listagem 4: Exemplo de mapeador usando um DBWritable personalizado

Escrevendo resultados de volta ao banco de dados

Uma classe complementar, ***DBOutputFormat***, permitirá gravar os resultados em um banco de dados. Ao configurar o job, chame ***conf.setOutputFormat (DBOutputFormat.class)***; e, em seguida, chame ***DBConfiguration.configureDB ()*** como antes.

O método ***DBOutputFormat.setOutput ()*** define como os resultados serão gravados no banco de dados. Seus três argumentos são o objeto JobConf para a tarefa, uma sequência que define o nome da tabela para a qual gravar e uma matriz de seqüências definindo os campos da tabela a serem preenchidos. por exemplo, ***DBOutputFormat.setOutput*** (***job, "employees", "employee_id", "name"***); .

A mesma implementação ***DBWritable*** que você criou anteriormente será suficiente para injetar registros de volta no banco de dados. O método ***write (PreparedStatement stmt)*** será invocado em cada instância do DBWritable que você passar para o OutputCollector do redutor. Ao final da redução, esses objetos PreparedStatement serão transformados em instruções INSERT para serem executados no banco de dados SQL.

Limitações do InputFormat

O JDBC permite que aplicativos gerem consultas SQL que são executadas no banco de dados; os resultados são então retornados ao aplicativo de chamada. Tenha em mente que você estará interagindo com seu banco de dados através de consultas SQL repetidas. Assim sendo:

O Hadoop pode precisar executar a mesma consulta várias vezes. Ele precisará retornar os mesmos resultados a cada vez. Portanto, quaisquer atualizações simultâneas em seu banco de dados, etc, não devem afetar a consulta que está sendo executada pelo seu trabalho MapReduce. Isso pode ser feito não permitindo gravações na tabela enquanto a tarefa MapReduce é executada, restringindo a consulta do ***MapReduce*** por meio de uma cláusula como ***" insert_date < yesterday "*** ou despejando os dados em uma tabela temporária no banco de dados antes de iniciar seu processo MapReduce.

Para paralelizar o processamento de registros do banco de dados, o Hadoop executará consultas SQL que usam cláusulas **ORDER BY , LIMIT e OFFSET** para selecionar intervalos fora das tabelas. Seus resultados, portanto, precisam ser ordenados por uma ou mais chaves (PRIMARY, como a do exemplo, ou UNIQUE).

Para definir o número de tarefas do mapa, o DBInputFormat precisa saber quantos registros ele irá ler. Portanto, se você estiver escrevendo uma consulta SQL arbitrária no banco de dados, precisará fornecer uma segunda consulta que retorne o número de linhas que a primeira consulta retornará (por exemplo, usando COUNT e GROUP BY).

Com essas restrições em mente, ainda há muita flexibilidade disponível para você. Você pode carregar em massa tabelas inteiras no HDFS ou selecionar grandes intervalos de dados. Por exemplo, se você deseja ler registros de uma tabela que também está sendo preenchida por outra origem simultaneamente, pode configurar essa tabela para anexar um campo de registro de data e hora a cada registro. Antes de fazer a leitura em massa, escolha o registro de data e hora atual e, em seguida, selecione todos os registros com registros de data e hora anteriores a esse. Novos registros sendo alimentados pelo outro gravador terão registros de data e hora posteriores e não afetarão o trabalho MapReduce.

Por fim, tenha cuidado para entender os gargalos no seu pipeline de processamento de dados. Lançar uma tarefa MapReduce com 100 mapeadores executando consultas em um servidor de banco de dados pode sobrecarregar o servidor ou sua conexão de rede. Nesse caso, você obterá menos paralelismo do que teoricamente possível, devido à inanição, busca de disco e outras penalidades de desempenho.

Limitações do OutputFormat

O DBOutputFormat grava no banco de dados gerando um conjunto de instruções INSERT em cada redutor. O método close () do redutor, em seguida, os executa em uma transação em massa. Realizar um grande número deles a partir de várias tarefas de redução simultaneamente pode anular um banco de dados. Se você deseja exportar um volume muito grande de dados, talvez seja melhor gerar as instruções INSERT em um arquivo de texto e, em seguida, usar uma ferramenta de importação de dados em massa fornecida pelo seu banco de dados para fazer a importação do banco de dados.

Processamento Distribuído

A ideia do cluster de processamento distribuído é ligar vários computadores como se fosse um só para formar um supercomputador. A tarefa ou processamento é dividido por um computador "mestre" e lançado o processo dividido entre outros computadores "escravos". Isso gera um tempo menor para execução de tal processo.

Supomos que um processo de 10Gb leve 30 minutos para ser executado. Digamos que o cluster tem 6 computadores. O tempo do processo será dividido por 6 (30/6= 5 minutos), ou seja, o cluster resolveria isto em 5 minutos.

O sistema operacional utilizado é o Linux (não informado a distribuição), o cluster foi formado por 4 computadores sendo um "mestre" e três escravos. Abaixo segue um passo a passo de como vai ser configurado e os comandos utilizados.

Foi usado o API MPICH-1.2.6 para a formação do cluster pode ser encontrada no endereço www.mcs.anl.gov/mpi/mpich/download.html obtendo o arquivo **mpich-1.2.6.tar.gz**.

Este arquivo, por default, se encontra no diretório **/home/user/Download** que deve ser descompactado com o comando, em modo root, **tar –zxvf mpich-1.2.6.tar.gz**. Nesta implementação, o arquivo foi movido para o diretório **/usr/local**, Ibidem é a escolha mais apropriada, com o comando **mv mpich-1.2.6 /usr/local**.

Dentro do diretório mpich-1.2.6, foi dado o comando de configuração do ambiente em diretório **fixo ./configure –prefix=/usr/local |& tee c.log**. Em seguida o comando de finalização **make install**.

Para testar se a instalação foi bem sucedida, dentro do diretório **/usr/local/examples** há um código escrito em linguagem C. Para gerar o seu executável é necessário o comando make cpi. O comando mpirun –np 1 cpi foi realizado mostrando cujo retorno deve ser o valor de PI.

No mestre deve ser verificado se o arquivo machines.sun4 existe no diretório **/usr/local/share**. Caso não deve se criar com o comando **#vi /usr/local/share/machines.sun4**. Este arquivo é uma lista de hosts responsável pelo comando tstmachines sun4 que testa as conexões descrevendo possíveis problemas encontrados.

Ainda no mestre, o arquivo **/usr/local/share/machine.LINUX** foi editado. Os nomes **localhost.localdomainhost** foram comentados **#localhost.localdomainhot,** e substituídos por:

master:8

slave1:4

slave2:4

slave3:4

PRÉ DEFINIÇÕES

Os IPs definidos para esta implementação são da Classe C compreendendo a 256 endereços por rede. O grupo de domínio possui o nome **UNIFEG.CLUSTER.RM**.

Master representa o nome da máquina mestre e SlaveX o nome para as máquinas escravas onde X compreende ao número da máquina.

master – 99.99.99.1

slave1 – 99.99.99.2

slave2 - 99.99.99.3

slave3 - 99.99.99.4

Foi criado uma interface, em todas as máquinas, chamada **eth2** com IP estático 99.99.99.X 255.255.255.0, para a rede do grupo.

Foi instalado, em todas as máquinas, os serviços **rsh**, **ssh** e o compilador **gcc** com os comandos:

#yum install rsh-server
#yum install openssh-server
#yum install gcc

Após instalação foram habilitados os serviços **rsh, rlogin, rexec** e **nfs** no utiliátior ntsysv para serem acrescentados, nas ultimas linhas, no arquivo securetty:

 # vi /etc/securetty
(...)
rsh
rlogin

ARQUIVO HOSTS

O arquivo Hosts, localizado no diretório **/etc**, é utilizado pelo sistema operacional GNU/Linux com a finalidade de relacionar nomes a endereços IPs, ou seja, a transposição do nome de uma máquina DNS em um endereço IP.Sua assinatura corresponde a sequência: . A ordem não faz diferença.

Para esta implementação foi definido (em todas as máquinas):

#vi /etc/hosts

127.0.0.1	master	localhost.localdomain	localhost
99.99.99.1	master	master.unifeg.cluster.rm	
99.99.99.2	slave1	slave1.unifeg.cluster.rm	
99.99.99.3	slave2	slave2.unifeg.cluster.rm	
99.99.99.4	salve3	slave3.unifeg.cluster.rm	

ARQUIVO HOSTS.EQUIV

Este arquiEo é responsável por bloquear computadores e usuários de acesso aos serviços “r*” (rsh, rexec, rcp, etc) sem a necessidade de fornecer senhas. A biblioteca MPI utiliza o protocolo rsh para conexão remota entre as máquinas. Sua assinatura corresponde a .

Foi utilizado (em todas as máquinas):

#vi /etc/hosts.equiv
master
slave1
slave2
slave3

ARQUIVO .RHOSTS

Este arquivo é usado para dar acesso de confiança entre máquinas permitindo que um usuário específico tenha acesso a conta de usuário.

Na documentação do mpichdevido a um possível erro de permissão a falta do arquivo machine.sun4, é recomendado a criação deste documento no diretório /home. A sua criação em /home e no /root sem maiores detalhes. Nesta implementação foi criado com o comando vi /root/.rhosts o arquivo com o conteúdo master e slave1 em linhas distintas. Em seguida, ainda para Group e Lusk (2012, p.71), foi dada permissão de leitura/gravação com o comando chmod og-rwx .rhosts (ou chmod 600 .rhosts) e copiado para outro diretório com o comando cp /root/.rhosts /home.

Foi utilizado (em todas as máquinas):

#vi /root/.rhosts
master
slave1
slave2
slave3

#chmod og-rwx .rhosts
#cp /root/.rhosts /home

PASTAS DE COMPARTILHAMENTO

Por fim, é montado unidade de compartilhamento em /home e /usr/local nas máquinas. No mestre acrescenta-se as linhas:

 # vi /etc/exports
/home *(rw,no_root_squash)
/usr/local *(rw,no_root_squash)

Nos escravos, o arquivo **fstab** é o responsável por montar pastas de compartilhamento em rede gerenciado pelo serviço **nfs.**

 # vi /etc/fstab

#caminho do servidor	ponto de montagem	tipo-fs	opções
master:/home/unifeg	/home/unifeg	nfs	exec,dev,suid,rw 1 1
master:/usr/local	/usr/local	nfs	exec,dev,suid,rw 1 1

TESTE

Certifique-se de que o arquivo machines.sun4 não existir você deverá cria-lo com o comando:

#vi /usr/local/share/machines.sun4
Dentro deste arquivo digite:
master
slave1
slave2
slave3

Dentro da pasta mpich-1.2.6, execute #tstmachines sun4. É correto não aparecer erros. Este arquivo é apenas para testar a conexão remota com os hosts. Caso não houver erros, o cluster está pronto.

Basta entrar no diretório /usr/local/examples e digitar mpirun –np N cpi. Onde N é o número de processadores ou máquinas que temos, neste caso 3. É um cluster heterogêneo onde os computadores têm arquiteturas diferentes.

Kubernetes

O Kubernetes é um sistema de containers em escala, é open source e desenvolvido ativamente por uma comunidade em todo mundo.

O Kubeadm atomatiza a instalação e a configuração de componentes do Kubernetes tais como o servidor de API, o Controller Manager, e o Kube DNS. Contudo, ele não cria usuários ou lida com a instalação de dependências no nível do sistema operacional e sua configuração. Para essas tarefas preliminares, é possível utilizar uma ferramenta de gerência de configuração como o Ansible ou o SaltStack. A utilização dessas ferramentas torna a criação de clusters adicionais ou a recriação de clusters existentes muito mais simples e menos propensa a erros.

Abaixo segue um passo a passo como configurar um cluster Kubernetes a partir do zero usando o Ansible e o Kubeadm

Objetivos

O cluster irá incluir os seguintes recursos físicos:

- Um nó master

O nó master (um node no Kubernetes refere-se a um servidor) é responsável por gerenciar o estado do cluster. Ele roda o Etcd, que armazena dados de cluster entre componentes que fazem o scheduling de cargas de trabalho para nodes de trabalho.

- Dois nós slaves (worker)

Nós slaves são os servidores onde suas cargas de trabalho (aplicações e serviços de containers) irão executar. Um worker continuará a executar sua carga de trabalho uma vez que estejam atribuídos a ela, mesmo se o master for desativado quando o scheduling estiver concluído. A capacidade de um cluster pode ser aumentada adicionando workers.

Pré-requisitos

- Um par de chaves SSH em sua máquina Linux/macOS/BSD local.
- Três servidores rodando CentOS 7 com pelo menos 1GB de RAM. Você deve ser capaz de fazer SSH em cada servidor como usuário root com o seu par de chaves SSH. Certifique-se também de adicionar sua chave pública à conta do usuário do centOS no nó master. Ansible instalado em sua máquina local.
- Familiaridade com o playbooks do Ansible. Para uma revisão, verifique Configuration Management 101: Writing Ansible Playbooks.
- Conhecimento de como lançar um container a partir de uma imagem Docker.

Passo 1 — Configurando o Diretório da Área de Trabalho e o Arquivo de Inventário Ansible

Nesse passo, você vai criar um diretório em sua máquina local que irá servir como sua área de trabalho. Você configurará o Ansible localmente para que ele possa se comunicar e executar comandos em seus servidores remotos. Depois disso pronto, você irá criar um arquivo hosts contendo informações de inventário tais como os endereços IP de seus servidores e os grupos aos quais cada servidor pertence.

Dos seus três servidores, um será o master com um IP exibido como master_ip. Os outros dois servidores serão workers e terão os IPs ***worker_1_ip*** e ***worker_2_ip***.

Crie um diretório chamado ***~/kube-cluster*** no diretório home de sua máquina local e faça um cd para dentro dele:

```
$ mkdir ~/kube-cluster
$ cd ~/kube-cluster
```

Esse diretório será sua área de trabalho para o restante desse passo a passo e conterá todos os seus playbooks de Ansible. Ele também será o diretório no qual você irá executar todos os comandos locais.

Crie um arquivo chamado ***~/kube-cluster/hosts*** usando o ***vi*** ou o seu editor de textos favorito:

```
$ vi ~/kube-cluster/hosts
```

Pressione ***i*** para inserir o seguinte texto ao arquivo, que irá especificar informações sobre a estrutura lógica do cluster:

~/kube-cluster/hosts

```
[masters]
master ansible_host=master_ip ansible_user=root

[workers]
worker1 ansible_host=worker_1_ip ansible_user=root
worker2 ansible_host=worker_2_ip ansible_user=root
```

Você deve se lembrar de que arquivos de inventário no Ansible são utilizados para especificar informações de servidor tais como endereços IP, usuários remotos, e agrupamentos de servidores para tratar como uma unidade única para a execução de comandos. O ***~/kube-cluster/hosts*** será o seu arquivo de inventário e você adicionou dois grupos Ansible a ele (masters e workers) especificando a estrutura lógica do seu cluster.

No grupo masters, existe uma entrada de servidor chamada "master" que lista o IP do node master (***master_ip***) e especifica que o Ansible deve executar comandos remotos como root.

De maneira similar, no grupo workers, existem duas entradas para os servidores workers (***worker_1_ip e worker_2_ip***) que também especificam o ansible_user como root.

Tendo configurado o inventário do servidor com grupos, vamos passar a instalar dependências no nível do sistema operacional e a criar definições de configuração.

Passo 2 — Criando um Usuário Não-Root em Todos os Servidores Remotos

Nesta seção você irá criar um usuário não-root com privilégios sudo em todos os servidores para que você possa fazer SSH manualmente neles como um usuário sem privilégios. Isso pode ser útil se, por exemplo, você gostaria de ver informações do sistema com comandos como ***top/htop***, ver a lista de containers em execução, ou alterar arquivos de configuração de propriedade do root. Estas operações são rotineiramente executadas durante a manutenção de um cluster, e a utilização de um usuário que não seja root para tarefas desse tipo minimiza o risco de modificação ou exclusão de arquivos importantes ou a realização não intencional de operações perigosas.

Crie um arquivo chamado ***~/kube-cluster/initial.yml*** na área de trabalho:

```
$ vi ~/kube-cluster/initial.yml
```

A seguir, adicione o seguinte play ao arquivo para criar um usuário não-root com privilégios sudo em todos os servidores. Um play no Ansible é uma coleção de passos a serem realizados que visam servidores e grupos específicos. O seguinte play irá criar um usuário sudo não-root:

~/kube-cluster/initial.yml

```
- hosts: all
  become: yes
  tasks:
    - name: create the 'centos' user
      user: name=centos append=yes state=present createhome=yes shell=/bin/bash

    - name: allow 'centos' to have passwordless sudo
      lineinfile:
        dest: /etc/sudoers
        line: 'centos ALL=(ALL) NOPASSWD: ALL'
        validate: 'visudo -cf %s'

    - name: set up authorized keys for the centos user
      authorized_key: user=centos key="{{item}}"
      with_file:
        - ~/.ssh/id_rsa.pub
```

Aqui está um detalhamento do que este playbook faz:

- Cria um usuário não-root ***centos***.
- Configura o arquivo ***sudoers*** para permitir o usuário centos executar comandos ***sudo*** sem uma solicitação de senha.
- Adiciona a chave pública em sua máquina local (normalmente ***~/.ssh/id_rsa.pub***) para a lista de chaves autorizadas do usuário remoto centos. Isto o permitirá fazer SSH para dentro de cada servidor como usuário centos.

Salve e feche o arquivo depois que tiver adicionado o texto.

Em seguida, rode o playbook localmente executando:

```
$ ansible-playbook -i hosts ~/kube-cluster/initial.yml
```

O comando será concluído dentro de dois a cinco minutos. Na conclusão, você verá uma saída semelhante à seguinte:

```
Output
PLAY [all] ****

TASK [Gathering Facts] ****
ok: [master]
ok: [worker1]
ok: [worker2]

TASK [create the 'centos' user] ****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [allow 'centos' user to have passwordless sudo] ****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [set up authorized keys for the centos user] ****
changed: [worker1] => (item=ssh-rsa AAAAB3...)
changed: [worker2] => (item=ssh-rsa AAAAB3...)
changed: [master] => (item=ssh-rsa AAAAB3...)

PLAY RECAP ****
master                     : ok=5    changed=4    unreachable=0    failed=0
worker1                    : ok=5    changed=4    unreachable=0    failed=0
worker2                    : ok=5    changed=4    unreachable=0    failed=0
```

Agora que a configuração preliminar está completa, você pode passar para a instalação de dependências específicas do Kubernetes.

Passo 3 — Instalando as Dependências do Kubernetes

Nesta seção, você irá instalar os pacotes no nível do sistema operacional necessários pelo Kubernetes com o gerenciador de pacotes yum do CentOS. Esses pacotes são:

- ***Docker*** - um runtime de container. Este é o componente que executa seus containers. Suporte a outros runtimes como o rkt está em desenvolvimento ativo no Kubernetes.
- ***kubeadm*** - uma ferramenta CLI que irá instalar e configurar os vários componentes de um cluster de uma maneira padrão.
- ***kubelet*** - um serviço/programa de sistema que roda em todos os nodes e lida com operações no nível do node.
- ***kubectl*** - uma ferramenta CLI usada para emitir comandos para o cluster através de seu servidor de API.

Crie um arquivo chamado ***~/kube-cluster/kube-dependencies.yml*** na área de trabalho:

```
$ vi ~/kube-cluster/kube-dependencies.yml
```

Adicione os seguintes plays ao arquivo para instalar esses pacotes em seus servidores:

~/kube-cluster/kube-dependencies.yml

```
- hosts: all
  become: yes
  tasks:
   - name: install Docker
     yum:
       name: docker
       state: present
       update_cache: true

   - name: start Docker
     service:
       name: docker
       state: started

   - name: disable SELinux
     command: setenforce 0

   - name: disable SELinux on reboot
     selinux:
       state: disabled

   - name: ensure net.bridge.bridge-nf-call-ip6tables is set to 1
     sysctl:
      name: net.bridge.bridge-nf-call-ip6tables
      value: 1
      state: present
```

```
  - name: add Kubernetes' YUM repository
    yum_repository:
     name: Kubernetes
     description: Kubernetes YUM repository
     baseurl: https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
     gpgkey: https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.goog
     gpgcheck: yes

  - name: install kubelet
    yum:
       name: kubelet
       state: present
       update_cache: true

  - name: install kubeadm
    yum:
       name: kubeadm
       state: present

  - name: start kubelet
    service:
      name: kubelet
      enabled: yes
      state: started
```

```
  - name: start kubelet
    service:
      name: kubelet
      enabled: yes
      state: started

- hosts: master
  become: yes
  tasks:
   - name: install kubectl
     yum:
        name: kubectl
        state: present
```

O primeiro play no playbook faz o seguinte:

- Instala o Docker, o runtime de container.
- Inicia o serviço do Docker.
- Desativa o SELinux, uma vez que ele ainda não é totalmente suportado pelo Kubernetes.
- Define alguns valores ***sysctl*** relacionados ao netfilter, necessários para o trabalho em rede. Isso permitirá que o Kubernetes defina regras de iptables para receber tráfego de rede IPv4 e IPv6 em bridge nos nodes.
- Adiciona o repositório YUM do Kubernetes às listas de repositórios de seus servidores remotos.

- Instala ***kubelet*** e ***kubeadm***.

O segundo play consiste de uma única tarefa que instala o ***kubectl*** no seu node master.

Salve e feche o arquivo quando você tiver terminado.

A seguir, execute o playbook:

```
$ ansible-playbook -i hosts ~/kube-cluster/kube-dependencies.yml
```

Na conclusão, você verá uma saída semelhante à seguinte:

```
Output
PLAY [all] ****

TASK [Gathering Facts] ****
ok: [worker1]
ok: [worker2]
ok: [master]

TASK [install Docker] ****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [disable SELinux] ****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [disable SELinux on reboot] ****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [ensure net.bridge.bridge-nf-call-ip6tables is set to 1] ****
changed: [master]
changed: [worker1]
changed: [worker2]
```

```
TASK [ensure net.bridge.bridge-nf-call-iptables is set to 1] ****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [start Docker] ****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [add Kubernetes' YUM repository] *****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [install kubelet] *****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [install kubeadm] *****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [start kubelet] ****
changed: [master]
changed: [worker1]
changed: [worker2]
```

```
PLAY [master] *****

TASK [Gathering Facts] *****
ok: [master]

TASK [install kubectl] ******
ok: [master]

PLAY RECAP ****
master                     : ok=9    changed=5    unreachable=0    failed=0
worker1                    : ok=7    changed=5    unreachable=0    failed=0
worker2                    : ok=7    changed=5    unreachable=0    failed=0
```

Após a execução, o Docker, o ***kubeadm*** e o ***kubelet*** estarão instalados em todos os seus servidores remotos. O ***kubectl*** não é um componente obrigatório e somente é necessário para a execução de comandos de cluster. A instalação dele somente no node master faz sentido nesse contexto, uma vez que você irá executar comandos ***kubectl*** somente a partir do master. Contudo, observe que os comandos ***kubectl*** podem ser executados a partir de quaisquer nodes worker ou a partir de qualquer máquina onde ele possa ser instalado e configurado para apontar para um cluster.

Todas as dependências de sistema agora estão instaladas. Vamos configurar o node master e inicializar o cluster.

Passo 4 — Configurando o Nó Master

Nesta seção, você irá configurar o nó master. Antes da criação de quaisquer playbooks, contudo, vale a pena cobrir alguns conceitos como Pods e Plugins de Rede do Pod, uma vez que seu cluster incluirá ambos.

Um pod é uma unidade atômica que executa um ou mais containers. Esses containers compartilham recursos tais como volumes de arquivo e interfaces de rede em comum. Os pods são a unidade básica de scheduling no Kubernetes: todos os containers em um pod têm a garantia de serem executados no mesmo node no qual foi feito o scheduling do pod.

Cada pod tem seu próprio endereço IP, e um pod em um node deve ser capaz de acessar um pod em outro node utilizando o IP do pod. Os containers em um único nó podem se comunicar facilmente através de uma interface local. Contudo, a comunicação entre pods é mais complicada e requer um componente de rede separado que possa encaminhar o tráfego de maneira transparente de um pod em um node para um pod em outro nó.

Essa funcionalidade é fornecida pelos plugins de rede para pods. Para este cluster vamos utilizar o Flannel, uma opção estável e de bom desempenho.

Crie um playbook Ansible chamado ***master.yml*** em sua máquina local:

```
$ vi ~/kube-cluster/master.yml
```

Adicione o seguinte play ao arquivo para inicializar o cluster e instalar o Flannel:

~/kube-cluster/master.yml

```
- hosts: master
  become: yes
  tasks:
    - name: initialize the cluster
      shell: kubeadm init --pod-network-cidr=10.244.0.0/16 >> cluster_initialized.txt
      args:
        chdir: $HOME
        creates: cluster_initialized.txt

    - name: create .kube directory
      become: yes
      become_user: centos
      file:
        path: $HOME/.kube
        state: directory
        mode: 0755

    - name: copy admin.conf to user's kube config
      copy:
        src: /etc/kubernetes/admin.conf
        dest: /home/centos/.kube/config
        remote_src: yes
        owner: centos
```

```
- name: install Pod network
  become: yes
  become_user: centos
  shell: kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/v0.9.1/Documentation/
  args:
    chdir: $HOME
    creates: pod_network_setup.txt
```

Aqui está um detalhamento deste play:

- A primeira tarefa inicializa o cluster executando ***kubeadm init***. A passagem do argumento ***--pod-network-cidr=10.244.0.0/16*** especifica a sub-rede privada que os IPs do pod serão atribuídos. O Flannel utiliza a sub-rede acima por padrão; estamos dizendo ao ***kubeadm*** para utilizar a mesma sub-rede.
- A segunda tarefa cria um diretório ***.kube*** em ***/home/centos***. Este diretório irá manter as informações de configuração tais como os arquivos de chaves do admin, que são requeridas para conectar no cluster, e o endereço da API do cluster.
- A terceira tarefa copia o arquivo ***/etc/kubernetes/admin.conf*** que foi gerado a partir do ***kubeadm init*** para o diretório home do seu usuário não-root centos. Isso irá permitir que você utilize o ***kubectl*** para acessar o cluster recém-criado.
- A última tarefa executa ***kubectl apply*** para instalar o ***Flannel. kubectl apply -f descriptor.[yml|json***] é a sintaxe para dizer ao ***kubectl*** para criar os objetos descritos no arquivo ***descriptor.[yml|json].*** O arquivo ***kube-flannel.yml*** contém as descrições dos objetos requeridos para a configuração do Flannel no cluster.

Salve e feche o arquivo quando você tiver terminado.

Execute o playbook:

```
$ ansible-playbook -i hosts ~/kube-cluster/master.yml
```

Na conclusão, você verá uma saída semelhante à seguinte:

```
Output

PLAY [master] ****

TASK [Gathering Facts] ****
ok: [master]

TASK [initialize the cluster] ****
changed: [master]

TASK [create .kube directory] ****
changed: [master]

TASK [copy admin.conf to user's kube config] *****
changed: [master]

TASK [install Pod network] *****
changed: [master]

PLAY RECAP ****
master                     : ok=5    changed=4    unreachable=0    failed=0
```

Para verificar o status do node master, faça SSH nele com o seguinte comando:

```
$ ssh centos@master_ip
```

Uma vez dentro do nó master, execute:

```
$ kubectl get nodes
```

Agora você verá a seguinte saída:

```
Output
NAME      STATUS    ROLES     AGE       VERSION
master    Ready     master    1d        v1.10.1
```

A saída informa que o nó master concluiu todas as tarefas de inicialização e está em um estado Ready do qual pode começar a aceitar nó worker e executar tarefas enviadas ao Servidor de API. Agora você pode adicionar os workers a partir de sua máquina local.

Agora você verá a seguinte saída:

```
Output
NAME      STATUS    ROLES     AGE       VERSION
master    Ready     master    1d        v1.10.1
```

A saída informa que o node master concluiu todas as tarefas de inicialização e está em um estado Ready do qual pode começar a aceitar nodes worker e executar tarefas enviadas ao Servidor de API. Agora você pode adicionar os workers a partir de sua máquina local.

Passo 5 — Configurando os Nós Worker

A adição de workers ao cluster envolve a execução de um único comando em cada um. Este comando inclui as informações necessárias sobre o cluster, tais como o endereço IP e a porta do Servidor de API do master, e um token seguro. Somentes os nodes que passam no token seguro estarão aptos a ingressar no cluster.

Navegue de volta para a sua área de trabalho e crie um playbook chamado ***workers.yml:***

```
$ vi ~/kube-cluster/workers.yml
```

Adicione o seguinte texto ao arquivo para adicionar os workers ao cluster:

~/kube-cluster/workers.yml

```
- hosts: master
  become: yes
  gather_facts: false
  tasks:
    - name: get join command
      shell: kubeadm token create --print-join-command
      register: join_command_raw

    - name: set join command
      set_fact:
        join_command: "{{ join_command_raw.stdout_lines[0] }}"


- hosts: workers
  become: yes
  tasks:
    - name: join cluster
      shell: "{{ hostvars['master'].join_command }} >> node_joined.txt"
      args:
        chdir: $HOME
        creates: node_joined.txt
```

Aqui está o que o playbook faz:

- O primeiro play obtém o comando de junção que precisa ser executado nos nodes workers. Este comando estará no seguinte formato: ***kubeadm join --token <token><master-ip>:<master-port> --discovery-token-ca-cert-hash sha256:<hash***>. Assim que obtiver o comando real com os valores apropriados de token e hash, a tarefa define isso como um fact para que o próximo play possa acessar essa informação.
- O segundo play tem uma única tarefa que executa o comando de junção em todos os nodes worker. Na conclusão desta tarefa, os dois nodes worker farão parte do cluster.

Salve e feche o arquivo quando você tiver terminado.

Execute o playbook:

```
$ ansible-playbook -i hosts ~/kube-cluster/workers.yml
```

Na conclusão, você verá uma saída semelhante à seguinte:

```
Output
PLAY [master] ****

TASK [get join command] ****
changed: [master]

TASK [set join command] *****
ok: [master]

PLAY [workers] *****

TASK [Gathering Facts] *****
ok: [worker1]
ok: [worker2]

TASK [join cluster] *****
changed: [worker1]
changed: [worker2]

PLAY RECAP *****
master                     : ok=2    changed=1    unreachable=0    failed=0
worker1                    : ok=2    changed=1    unreachable=0    failed=0
worker2                    : ok=2    changed=1    unreachable=0    failed=0
```

Com a adição dos nodes worker, seu cluster está agora totalmente configurado e funcional, com os workers prontos para executar cargas de trabalho. Antes de fazer o scheduling de aplicações, vamos verificar se o cluster está funcionando conforme o esperado.

Step 6 — Verificando o Cluster

Às vezes, um cluster pode falhar durante a configuração porque um node está inativo ou a conectividade de rede entre o master e o worker não está funcionando corretamente. Vamos verificar o cluster e garantir que os nodes estejam operando corretamente.

Você precisará verificar o estado atual do cluster a partir do nó master para garantir que os nodes estejam prontos. Se você se desconectou do nó master, pode voltar e fazer SSH com o seguinte comando:

```
$ ssh centos@master_ip
```

Em seguida, execute o seguinte comando para obter o status do cluster:

```
$ kubectl get nodes
```

Você verá uma saída semelhante à seguinte:

```
Output
NAME      STATUS    ROLES     AGE       VERSION
master    Ready     master    1d        v1.10.1
worker1   Ready     <none>    1d        v1.10.1
worker2   Ready     <none>    1d        v1.10.1
```

Se todos os seus nodes têm o valor Ready para o STATUS, significa que eles são parte do cluster e estão prontos para executar cargas de trabalho.

Se, contudo, alguns dos nodes têm ***NotReady*** como o STATUS, isso pode significar que os nós worker ainda não concluíram sua configuração. Aguarde cerca de cinco a dez minutos antes de voltar a executar ***kubectl get*** nodes e fazer a inspeção da nova saída. Se alguns nodes ainda têm ***NotReady*** como status, talvez seja necessário verificar e executar novamente os comandos nas etapas anteriores.

Agora que seu cluster foi verificado com sucesso, vamos fazer o scheduling de um exemplo de aplicativo Nginx no cluster.

Passo 7 — Executando Uma Aplicação no Cluster

Você pode fazer o deploy de qualquer aplicação containerizada no seu cluster. Para manter as coisas familiares, vamos fazer o deploy do Nginx utilizando Deployments e Services para ver como pode ser feito o deploy dessa aplicação no cluster. Você também pode usar os comandos abaixo para outros aplicativos em container, desde que você altere o nome da imagem do Docker e quaisquer flags relevantes (tais como ***ports*** e ***volumes***).

Ainda no nó master, execute o seguinte comando para criar um deployment chamado ***nginx:***

```
$ kubectl run nginx --image=nginx --port 80
```

Um deployment é um tipo de objeto do Kubernetes que garante que há sempre um número especificado de pods em execução com base em um modelo definido, mesmo se o pod falhar durante o tempo de vida do cluster. O deployment acima irá criar um pod com um container do registro do Docker Nginx Docker Image.

A seguir, execute o seguinte comando para criar um serviço chamado ***nginx*** que irá expor o app publicamente. Ele fará isso por meio de um NodePort, um esquema que tornará o pod acessível através de uma porta arbitrária aberta em cada node do cluster:

```
$ kubectl expose deploy nginx --port 80 --target-port 80<^> --type NodePort
```

Services são outro tipo de objeto do Kubernetes que expõe serviços internos do cluster para os clientes, tanto internos quanto externos. Eles também são capazes de fazer balanceamento de solicitações para vários pods e são um componente integral no Kubernetes, interagindo frequentemente com outros componentes.

Execute o seguinte comando:

```
$ kubectl get services
```

Isso produzirá uma saída semelhante à seguinte:

```
Output
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP          PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1       <none>               443/TCP        1d
nginx        NodePort    10.109.228.209  <none>               80:nginx_port/TCP   40m
```

A partir da terceira linha da saída acima, você pode obter a porta em que o Nginx está sendo executado. O Kubernetes atribuirá uma porta aleatória maior que 30000 automaticamente, enquanto garante que a porta já não esteja vinculada a outro serviço.

Para testar se tudo está funcionando, visite ***http://worker_1_ip:nginx_port ou http://worker_2_ip:nginx_port*** através de um navegador na sua máquina local. Você verá a familiar página de boas-vindas do Nginx.

Se você quiser remover o aplicativo Nginx, primeiro exclua o serviço nginx do node master:

```
$ kubectl delete service nginx
```

Execute o seguinte para garantir que o serviço tenha sido excluído:

```
$ kubectl get services
```

Você verá a seguinte saída:

```
Output
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP          PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1       <none>               443/TCP        1d
```

Para excluir o deployment:

```
kubectl delete deployment nginx
```

Execute o seguinte para confirmar que isso funcionou:

```
$ kubectl get deployments
```

```
Output
No resources found.
```