Elastic MapReduce

Container-Based EMR

Product Documentation

Contents

Container-Based EMR Overview

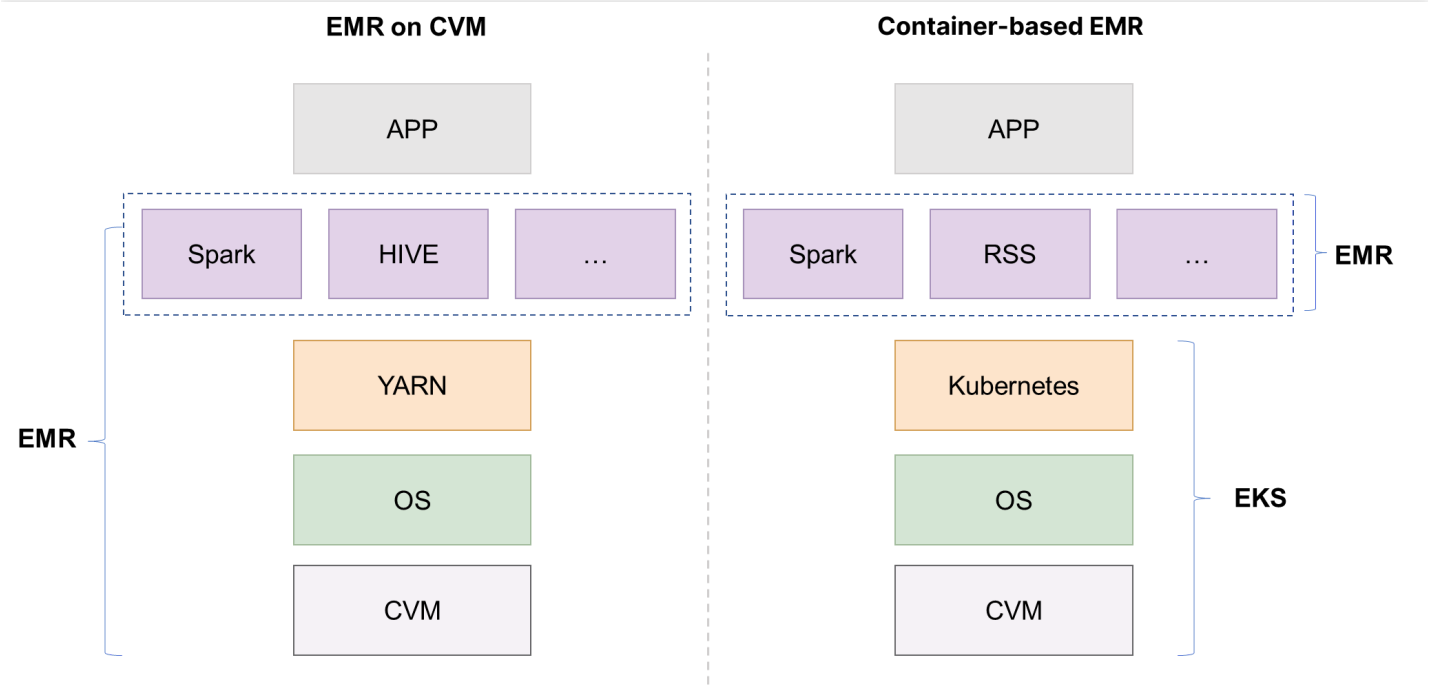
Last updated：2023-05-08 15:22:21

Unavailable for Purchase:
The container-based EMR has been unavailable for purchase from March 10, 2023 for feature updates, and existing clusters are not affected. The new edition will be available for beta testing soon. Please stay tuned.

Container-based EMR offers a new way of deploying open-source big data components solely based on a container service. For example, you can deploy big data components to the cloud-native EKS and leverage its strengths in container application management to reduce the resource Ops costs and quickly create a cluster to run big data jobs.

Deployments

EMR provides open-source big data component deployments on CVM or EKS to meet different user needs.



Deployment	Description
EMR on CVM	EMR deploys the open-source big data components on CVM based on user needs and starts the installed services. In addition, the EMR console allows for Ops operations on cluster and component services to facilitate big data job execution.
Container-based EMR	EMR deploys the big data components in the resources provided by EKS, and the component services run in the container. You can run Spark jobs directly in the container cluster and associate them with RSS clusters to improve stability.

Strengths

Strength	Description
Reduced costs	Container-based EMR is serverless and out-of-the-box with a high resource utilization. Spark clusters automatically create Pod resources based on job needs and release them after the jobs end, saving costs.
Easy Ops	Container-based EMR is deployed based on EKS, a fully managed Kubernetes service. In contrast to CVM, it can quickly recover abnormal component services. Spark clusters automatically adjust Pod resources, simplifying node resource Ops.
Elastic scaling	Container-based EMR allows you to adjust the number of containers. It relies on EKS's unlimited resources and proprietary lightweight virtualization technology. It can implement the quick scaling of Pod resources to support jobs involving a large data volume.

Operation Guide
Creating Clusters

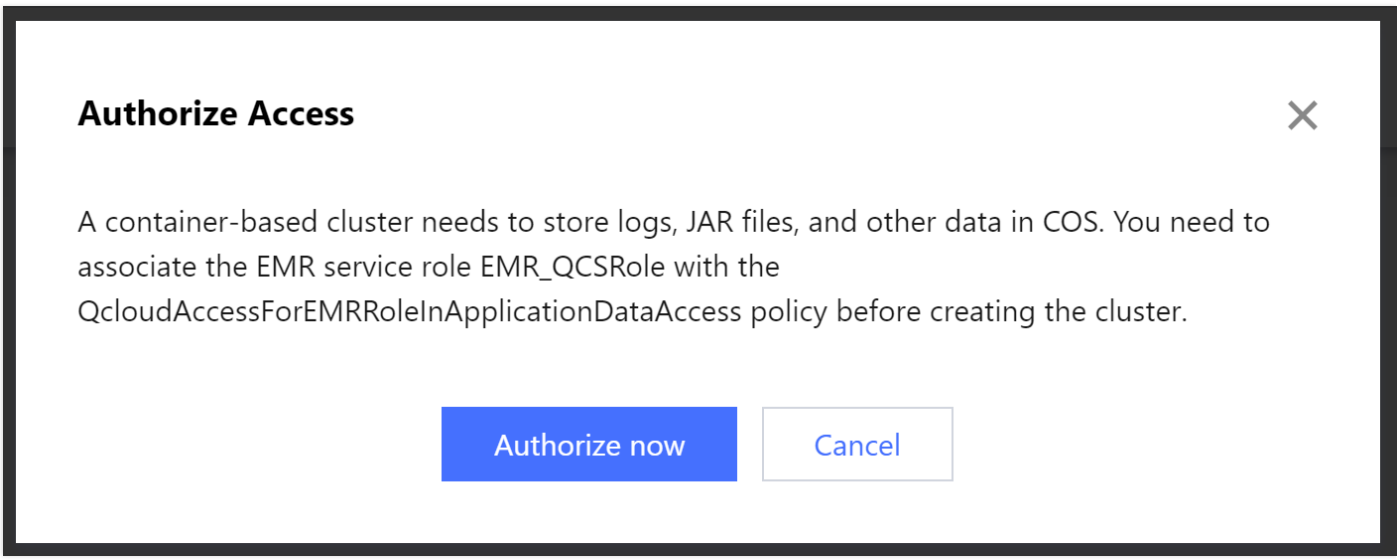
Last updated：2023-02-01 16:06:22

Overview

This document describes how to create a container-based EMR cluster in the EMR console.

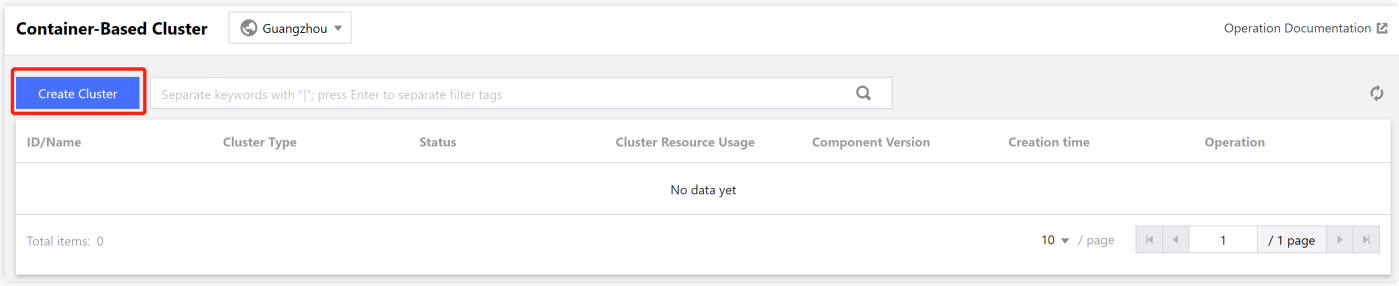
Prerequisites

1. You have completed the role authorization. For more information, see Role Authorization.
2. You have completed the COS authorization. When you create a cluster, if COS access has not been granted, the following prompt will be displayed. Click **Authorize now** to authorize COS. After successful COS authorization, you won't need to authorize COS again when creating clusters in the future.



Directions

1. Log in to the EMR console and click **Create Cluster** on the container-based cluster list page.



2. On the **Create Cluster** page, configure the following items:

Field Name	Description
Cluster Name	1. The name can contain 6–36 letters, digits, hyphens, and underscores. 2. A random cluster name will be generated by default.
Region	A region is the physical location of an IDC. Currently supported regions include Beijing, Shanghai, Guangzhou, and Nanjing.
Cluster Type	Currently, Spark and RSS cluster types are supported.
Component Version	Components and their version information under the selected cluster type.
Container Type	If you want to select an EKS cluster but there is none in your region, a hidden EKS cluster (counted against the quota) will be automatically created to expand EMR compute resources.
Container Network	Set a network dedicated for the hidden EKS cluster. If you have selected a container network for this EKS cluster, it is bound and cannot be changed.
Specification Configuration	Currently, resource specifications can be configured only for RSS clusters. You can configure the Pod specification of the `Coordinator` and `Shuffle Server` roles as needed. Note that once specified, the data disk type, disk size and quantity, CPU type and range, and memory range cannot be modified.
COS Bucket	1. Select an existing bucket or create a new one in the COS console. 2. Before using COS, you need to grant COS read/write permissions to the EMR cluster first.

3. Click **Create**. Then, you can view the newly created cluster in the EMR console.

Managing Cluster

Cluster Management Overview

Last updated：2023-02-01 15:51:18

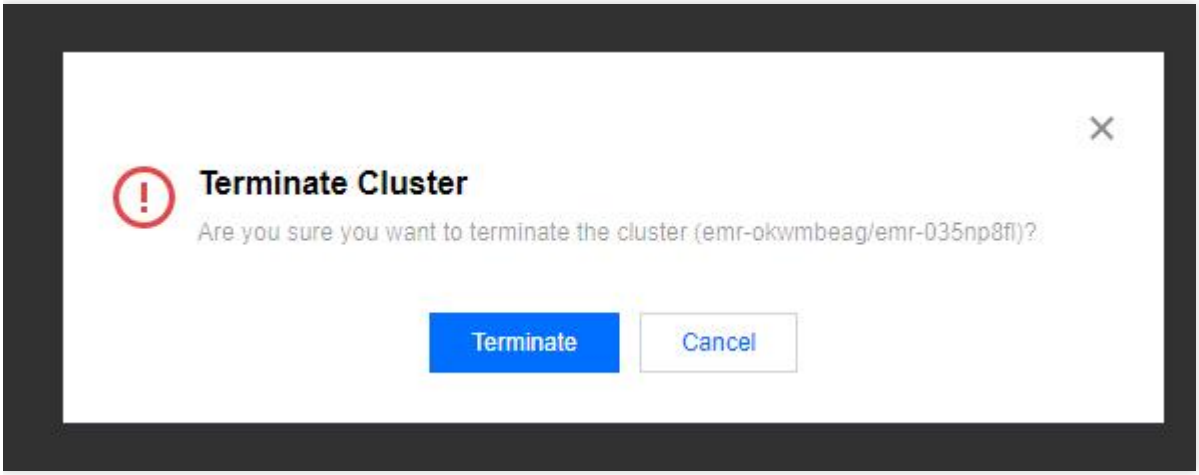
This document describes how to view the information of a container-based EMR cluster in the console.

Viewing the cluster information

1. After successfully creating a cluster, log in to the container-based EMR console, click the **ID/Name** of the target cluster on the **Cluster list** page, or select **Details** in the **Operation** column in the **Cluster list**.
2. **Cluster Info** records the basic information of the EMR cluster, such as **Region Info**, **Namespace**, **Component Version**, **Container/Cluster Type**, **Container Network**, **COS**, **Bucket Name**, **Custom Service Role**, and **Resource Usage**.
3. If you need refined authorization, you can set a custom service role for accessing Tencent Cloud resources during the execution of big data jobs. You should select **Tencent Cloud Product Service** as the service role type and **Elastic MapReduce** as the service supporting the role.

Terminating a cluster

When you no longer need a container-based EMR cluster, you can log in to the container-based EMR console and select **Terminate** in the **Operation** column in the **Cluster list**. Then, in the **Terminate Cluster** pop-up window, confirm the information of the cluster to be terminated and click **Terminate**.



Deleting a cluster

When a container-based EMR cluster fails to be created, you can log in to the container-based EMR console and select **Delete** in the **Operation** column in the **Cluster list**. Then, in the **Delete Cluster** pop-up window, confirm the information of the cluster to be deleted and click **Confirm**.

Managing Spark Job

Last updated：2022-08-17 16:26:17

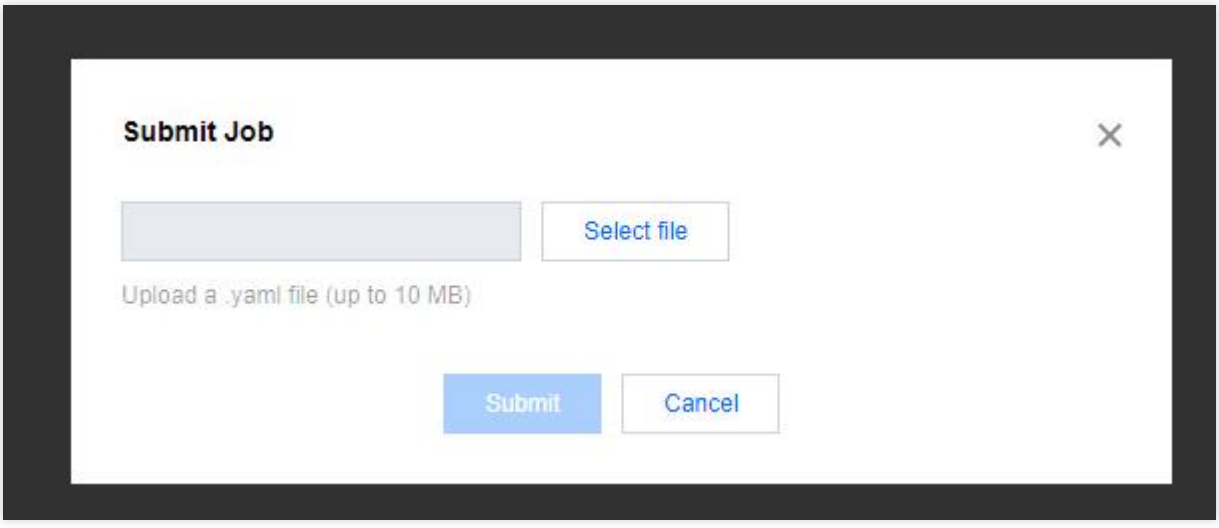
Feature Overview

Container-based EMR clusters allow you to submit Spark jobs and view job information in the console.

> Note：
> A job should be submitted as a YAML file of up to 10 MB in size.

Directions

1. Log in to the container-based EMR console and click the **ID/Name** of the target cluster in the **Cluster list** to enter the cluster details page.
2. On the cluster details page, click **Job Management** to submit and query jobs.
3. You can submit YAML job files through CRD in the EMR console after compiling the files.
4. Click **Submit Job** above the **Job List** to pop up the **Submit Job** window. Then, select the job file to be submitted and click **Confirm**.



5. Click **Details** in the **Job List** to enter the Spark HistoryServer UI to view the job details.
6. Click **Delete** in the **Job List**. Then, in the **Delete job** pop-up window, confirm the information of the job to be deleted and click **Confirm**.

Sample Job

The process of submitting a Spark job through CRD is as follows:

1. Write a Spark program.
2. Compile and package the program into a JAR package and put the package in the COS or HDFS file system, or write a Dockerfile to create an image for the package.
3. Write a YAML file and submit it in the console.

The following describes four sample Spark jobs:

Sample 1. Using a Spark JAR package

Below is a sample YAML job file:

```
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
 name: test1
spec:
 hadoopConf:
   "fs.cosn.userinfo.secretId":"$SecretId"
   "fs.cosn.userinfo.secretKey":"$SecretKey"
 type: Scala
 mode: cluster
 mainClass: org.apache.spark.examples.SparkPi
 mainApplicationFile: "local:///opt/spark/examples/jars/spark-examples_2.12-3.2.
0.jar"
```

For more information on the parameters used in this sample, visit GitHub.

- `apiVersion` and `kind` are the resource version and type in K8s, which cannot be changed here.
- `metadata.name` defines the job name, which is `test1` here and can be customized.
- `spec.hadoopConf` defines the configuration information of Hadoop. Interacting with COS requires configuring the key information, which can be obtained on the Manage API Key page. The `$SecretId` and `$Secretkey` in the code should be replaced with your actual `SecretId` and `Secretkey` .
- `type` defines the type of the Spark program, which can be Java, Scala, Python, or R. It is Scala here and can be selected as needed.
- `mode` defines the deployment mode of `sparkApplication` , which can be cluster or client. It is cluster here and can be selected as needed.
- `driver` and `executor` define the Spark driver and executor respectively. They are automatically generated on the backend as follows by default:

```
driver:
cores: 1
memory: 512m
executor:
cores: 1
instances: 2
memory: 512m
```

You can customize the `driver` and `executor` parameters and add them to the sample YAML job file 1. Then, the custom parameters will overwrite the default parameters. Below is a sample:

```
driver:
   cores: 1
   coreLimit: "1200m"
   memory: "512m"
 executor:
   cores: 1
   instances: 1
   memory: "512m"
```

Sample 2. Compiling and packaging a Spark program and putting the JAR package in COS (recommended)

The following sample shows the complete process of compiling a Spark program, packaging it into a JAR package, and writing and submitting a YAML job file.

1. Prepare for development.
   You need to have a COS bucket for this job, which can be the bucket you selected when creating the cluster or a new bucket created in the same region as the previously selected bucket.

2. Create a project with Maven.
   You need to create a project and then compile, package, and upload it. Maven is recommended because it can help you manage project dependency more easily.

3. Write a WordCount program and add the following sample code:

```
import java.util.Arrays;
import java.util.regex.Pattern;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
```

```
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.sql.SparkSession;
import scala.Tuple2;
public class WordCountOnCos {
   private static final Pattern SPACE = Pattern.compile(" ");
   public static void main(String[] args){
       if (args.length < 1) {
           System.err.println("Usage: JavaWordCount <file>");
           System.exit(1);
       }
       SparkSession spark = SparkSession.builder().appName("wordCountOnCos").ge
tOrCreate();
       JavaRDD<String> lines = spark.read().textFile(args[0]).javaRDD();
       JavaRDD<String> words = lines.flatMap(s -> Arrays.<String>asList(SPACE.s
plit(s)).iterator());
       JavaPairRDD<String, Integer> ones = words.mapToPair(s -> new Tuple2(s, I
nteger.valueOf(1)));
       JavaPairRDD<String, Integer> counts = ones.reduceByKey((i1, i2) -> Integ
er.valueOf(i1.intValue() + i2.intValue()));
       counts.saveAsTextFile(args[1]);
       spark.stop();
   }
}
```

4. Run the `mvn package` command to package the entire project.

5. Upload the JAR package to the COS bucket and write a YAML file as follows:

```
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
 name: test2
spec:
 hadoopConf:
   "fs.cosn.userinfo.secretId":"$SecretId"
   "fs.cosn.userinfo.secretKey":"$SecretKey"
 type: Java
 mode: cluster
 mainClass: com.tencent.WordCountOnCos
 mainApplicationFile: "cosn://kt-test-251007880/sparkapp/jar/wordcount.jar"
 arguments:
   - "cosn://kt-test-251007880/sparkapp/input/input"
   - "cosn://kt-test-251007880/sparkapp/output"
```

Here, `arguments` is the parameters passed to the main class and indicates the input and output directories of the WordCount program. The `mainApplicationFile` and the input and output directories of the WordCount program here are examples and can be customized.

Sample 3. Compiling and packaging a Spark program into a JAR package and putting it in HDFS

Write a Spark program and package it into a JAR package as shown in sample 2. Then, upload the package to HDFS and write a YAML file as follows:

```
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
 name: test3
spec:
 hadoopConf:
   "fs.cosn.userinfo.secretId":"$SecretId"
   "fs.cosn.userinfo.secretKey":"$SecretKey"
 type: Java
 mode: cluster
 mainClass: com.tencent.WordCountOnCos
 mainApplicationFile: "hdfs://$ip:$port/sparkapp/jar/wordcount.jar"
 arguments:
   - "cosn://kt-test-251007880/sparkapp/input/input"
   - "hdfs://$ip:$port/sparkapp/output"
```

Note

If you store the JAR package in HFDS, HDFS should be in the same VPC as the container-based cluster.

Sample 4. Compiling and packaging a Spark program into a JAR package and creating an image for it

Write a Spark program and package it into a JAR package as shown in sample 2. Then, create a Dockerfile as follows:

```
FROM ccr.ccs.tencentyun.com/emr-image/spark:BaseImage
USER root
RUN mkdir -p /sparkapp
COPY jars/wordcount.jar /sparkapp
ENTRYPOINT [ "/opt/entrypoint.sh" ]
```

You need to inherit the base image `ccr.ccs.tencentyun.com/emr-image/spark:BaseImage` , which contains the JAR package required to interact with COS.

```
docker build -t ccr.ccs.tencentyun.com/emr-image/spark:wc -f ./bin/Dockerfile .
```

Write a YAML job file as follows:

```
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
 name: test4
spec:
 hadoopConf:
   "fs.cosn.userinfo.secretId":"$SecretId"
   "fs.cosn.userinfo.secretKey":"$SecretKey"
 type: Java
 mode: cluster
 mainClass: com.tencent.WordCountOnCos
 image: ccr.ccs.tencentyun.com/emr-image/spark:wc
 mainApplicationFile: "local:///sparkapp/wordcount.jar"
 arguments:
   - "cosn://kt-test-251007880/sparkapp/input/input"
   - "cosn://kt-test-251007880/sparkapp/output"
```

Here, `image` is the image you created through packaging, and `mainApplicationFile` is the path of the JAR package in the image.