目录

Hue简介

Hue (Hadoop User Experience)是一个开源的Apache Hadoop UI系统，由Cloudera Desktop演化而来，最后Cloudera公司将其贡献给Apache基金会的Hadoop社区，其基于Python Web框架Django实现。Hue为KMR集群提供了图形化用户界面，便于用户配置、使用以及查看KMR集群。

HUE功能

1.访问HDFS和文件浏览；

2.通过web调试和开发hive以及数据结果展示；

3.solr查询、结果展示、报表生成；

4.通过web调试和开发impala交互式SQL Query；

5.spark调试和开发；

6.Pig开发和调试；

7.oozie任务的开发、监控和工作流协调调度；

8.Hbase数据查询和修改、数据展示；

9.Hive的元数据（metastore）查询；

10.MapReduce任务进度查看，日志追踪；

11.创建和提交MapReduce，Streaming，Java job任务；

12.Sqoop2的开发和调试；

13.Zookeeper的浏览和编辑；

14.数据库（MySQL、PostgreSQL、SQlite，Oracle）的查询和展示。

相关连接

官网：http://gethue.com/

Github：https://github.com/cloudera/hue

Reviews：https://review.cloudera.org

登陆Hue控制台

在使用Hue组件管理工作流时，首先登陆Ambari控制台，然后再登陆Hue控制台页面。

操作步骤

1.在KMR集群详情页中的选择Ambari控制台。

2.在Ambari控制台Service界面中选择**Hue**，单击上边**Quick Links**下拉菜单，然后单击**Hue WEB-UI**即可进入Hue的Web页面。

3.首次登录会创建用户，请牢记首次登录的用户名和密码，如果忘记请联系金山云的开发人员重置密码。

HDFS 文件浏览

通过Hue的Web页面可方便查看HDFS中的文件及文件夹，以及对其进行创建、下载、上传、复制、修改及删除等操作。

操作步骤

1、在 Hue 控制台左侧，选择 **Browsers>Files** 进入 HDFS 文件浏览。

2、进入 **File Browser** ，可以对文件进行创建、下载、上传、复制、修改及删除、修改权限等操作。

Hive SQL 查询

Hive操作

HUE的beeswax app提供友好方便的Hive查询功能，能够选择不同的Hive数据库，编写HQL语句，提交查询任务，并且能够在界面下方看到查询作业运行的日志。在得到结果后，还提供进行简单的图表分析能力。

新建数据库

1.页面新建

页面单击 **Query>Editor>Hive**。

2.输入框输入语句

```
create database if not exists db2;
```

注意

在新建数据库时，可能会出现一下错误：

```
Execution Error, return code 1 from org.apache.hadoop.hive.ql.exec.DDLTask. MetaException(message:java.security.AccessControlException: Permission denied: user=hive, access=WRITE, inode="/":hdfs:hdfs:drwxr-xr-x
```

出现错误原因如下：

对于使用非hdfs用户进行 xxx 操作导致类似错误，我们不建议用户使用root用户进行类似操作，应切换到所使用服务对用的用户下进行操作，以减少不必要的问题。

对于第一次使用Hive/HBase导致此错误，进行以下操作：

```
su  -hdfs

#If error come with Hive

hdfs dfs -mkdir -p /apps/hive

hdfs dfs -chown hive:hdfs /apps/hive/

#If error come with HBase

hdfs dfs -mkdir -p /apps/hbase

hdfs dfs -chown hbase:hdfs /apps/hbase
```

新建表

1.在hive输入框输入语句。

```
CREATE TABLE IF NOT EXISTS user_test (

user_id string ,

seller_id string ,

product_id string ,

time string

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' ;
```

2.向hdfs上传文档文件，其中导入文档内容如下。

```
0001    zhangsan    99  98  100 school1 class1
0002    lisi    59  89  79  school1 class1
0003    wangwu  89  99  100 school3 class1
0004    zhangsan2   99  98  100 school1 class1
0005    lisi2   59  89  79  school2 class1
0006    wangwu2 89  99  100 school3 class1
```

点击Tables后加号标志，在SOURCE后导入文档（下列中导入的为txt文档），单击**Next**。 您也可以直接输入SQL语句进行表创建，如图所示。

3.表创建完成后页面如下，可看到表结构，表状态等相应信息，在右上侧output中可以看到历史作业及其状态信息，日志信息也可在上方栏目中查看。

查询

1.单击 **Query** 下拉菜单，依次选择 **Editor>Hive**。

2.在输入框内输入相应SQL语句，进行数据库搜索。在下方Results中可以看到搜索结果。

结果可视化

在下图中单击①处，可选择展示图类型，此例中选择Pie饼状图，其结果在右方显示。

Oozie 任务调度

工作流数据

Hue的任务是基于工作流的调度，我们创建一个包含 Hive script 脚本的工作流，其中具体脚本内容如下所示：

```
create database if not exists hive_db;
show databases;
use hive_db;
show tables;
create table if not exists hive_test (a int, b string);
show tables;
insert into hive_test select 1, "test";
select * from hive_test;
```

将上面脚本内容保存到hive_test.sql脚本文件中，并上传到hdfs目录/tmp/目录下;另外Hive工作流还需要一个hive-site.xml 配置文件，此配置文件路径在：/usr/hdp/2.6.1.0-129/hive/conf/hive-site.xml下，同时将改配置文件上传到hdfs目录/tmp/目录下。

创建工作流

1.在Hue页面的上方，选择Workflow，具体如下图所示。

2.在工作流编辑页面中拖一个 Hive Script。

3.选择上传到hdfs上的hive_test.sql脚本文件和hive-site.xml 配置文件。

4.单击 **Add** 后，还需在 **FILES** 中指定 **hive script** 文件。

5.单击右上角**保存**，然后单击**执行**，运行 workflow。

创建定时任务

Hue是支持定时任务调度的，有点类似于crontab执行命令，该定时任务支持的调度粒度可以到分钟级别。

1.在Hue页面的上方，选择Schedule，具体如下图所示。

2.选择一个创建好的工作流，然后选择需要调度的时间、时间间隔、时区、调度任务的开始时间及结束时间，然后单击保存和执行，具体如下所示。

3.最后在Workflows、Schedulers 的监控页面可以查看任务调度执行情况。

MapReduce简介

MapReduce是一种编程模型，用于大规模数据集（大于1TB）的并行运算。概念"Map（映射）"和"Reduce（归约）"，是它们的主要思想，都是从函数式编程语言里借来的，还有从矢量编程语言里借来的特性。它极大地方便了编程人员在不会分布式并行编程的情况下，将自己的程序运行在分布式系统上。 当前的软件实现是指定一个Map（映射）函数，用来把一组键值对映射成一组新的键值对，指定并发的Reduce（归约）函数，用来保证所有映射的键值对中的每一个共享相同的键组。

使用Maven工程来管理MR作业

当您的工程规模越来越大时，会变得非常复杂，不易管理。我们采用类似Maven这样的软件项目管理工具来进行管理。其操作步骤如下。

1.确保本地安装好Maven。

2.在IDE打开,并编辑pom.xml文件，在dependencies内添加如下内容。

```
<dependency>
     <groupId>org.apache.hadoop</groupId>
     <artifactId>hadoop-mapreduce-client-common</artifactId>
     <version>2.6.0</version>
 </dependency>
 <dependency>
     <groupId>org.apache.hadoop</groupId>
     <artifactId>hadoop-common</artifactId>
     <version>2.6.0</version>
 </dependency>
```

3.Wrodcount实例代码(以下代码来源于Hadoop官网)。

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount2 {

    public static class TokenizerMapper
            extends Mapper<Object, Text, Text, IntWritable>{

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context
        ) throws IOException, InterruptedException {
```

```
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }

    public static class IntSumReducer
            extends Reducer<Text,IntWritable,Text,IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values,
                           Context context
        ) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount2.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

4.编译并打包上传。 在工程目录下执行以下命令，您在工程目录的target目录下看到一个wordcountv2-1.0-SNAPSHOT.jar，将jar包上传到服务器上。

```
mvn clean package -DskipTests
```

5.作业输入输出。 hadoop作业的输入和输出文件，可以放在HDFS上，也可以选择放在KS3上。

（1）使用HDFS

将输入文件放到HDFS上，假设输入文件为TWILIGHT.txt。首先su hdfs用户下。

```
hadoop dfs -mkdir -p /user/hadoop/examples/input
hadoop dfs -put TWILIGHT.txt /user/hadoop/examples/input
```

（2）使用KS3

```
hadoop dfs -mkdir -p ks3://kmrtest9/wordcount/lib
hadoop dfs -mkdir -p ks3://kmrtest9/wordcount/input
hadoop dfs -put wordcountv2-1.0-SNAPSHOT.jar ks3://kmrtest9/wordcount/lib
hadoop dfs -put TWILIGHT.txt ks3://kmrtest9/wordcount/input
```

作业提交方法

（1）输入输出在HDFS上：

```
hadoop jar wordcountv2-1.0-SNAPSHOT.jar WordCount2 /user/hadoop/examples/input/ /user/hadoop/examples/output
```

（2）输入输出在KS3上：

```
hadoop jar wordcountv2-1.0-SNAPSHOT.jar WordCount2 ks3://kmrtest9/wordcount/input/ ks3://kmrtest9/wordcount/output
```

Hadoop Streaming 构建MR作业

1.Hadoop Streaming允许用户使用可执行的命令或者脚本作为mapper和reducer。以下用几个示例说明Hadoop Streaming如何使用。 详细可参考Hadoop官网Hadoop Streaming。

2.使用python脚本作为mapper和reducer。

mapper.py

```
#!/usr/bin/env python
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print "%s\t%s" % (word, 1)
```

reducer.py

```
#!/usr/bin/env python
from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    try:
        count = int(count)
    except ValueError:
        continue
    if current_word == word:
        current_count += count
    else:
        if current_word:
            print "%s\t%s" % (current_word, current_count)
        current_count = count
        current_word = word

if word == current_word:
    print "%s\t%s" % (current_word, current_count)
```

3.命令行执行streaming作业。

```
chmod +x mapper.py
chmod +x reducer.py
hadoop jar /usr/hdp/2.6.1.0-129/hadoop-mapreduce/hadoop-streaming-2.7.3.2.6.1.0-129.jar -input /user/hadoop/examples/input/ -output
 /user/hadoop/examples/output2 -mapper mapper.py -reducer reducer.py -file mapper.py  -file reducer.py
```

Presto简介

Presto是一种分布式、交互式的SQL查询引擎，支持的数据量可以达到PB级别，解决商业数仓的交互式分析和处理速度的问题。可以实现对Hive、Cassandra、关系型数据库以及特有数据存储的在线实时查询，延时可达到亚秒级。

Presto整合mysql

1.在kmr集群所有节点Presto的/etc/presto/catalog目录下创建mysql.properties，并加入以下内容：

```
connector.name=mysql
connection-url=jdbc:mysql://$host:$port
connection-user=ambari
connection-password=ambari
```

其中$host为集群master节点的IP，$port为您的端口号。

2.在Ambari控制台界面中 Services>Presto>Service Actions重启。

然后切换到 presto-client 文件夹中，并且使用 Presto 连接 Mysql：

```
/usr/lib/presto/bin/presto-cli --server kmr-4014e7ad-gn-bdf258f3-master-1-001.ksc.com:8285 --catalog mysql
```

其中 --catalog 参数表示要操纵的数据库类型，执行成功后即可进入 Presto 的界面，并且直接进入指定的数据库。可以使用 Mysql 来查看数据库中的表：

```
#查看所有MySQL数据库
presto> SHOW SCHEMAS FROM mysql;
       Schema
--------------------
 ambari
 hive
 hue
```

```
 information_schema
 performance_schema
 test
(6 rows)

Query 20210202_100436_00008_ujqvm, FINISHED, 4 nodes
Splits: 53 total, 53 done (100.00%)
0:00 [6 rows, 83B] [30 rows/s, 417B/s]
# 查看某个MySQL数据库下的所有表
presto> SHOW TABLES FROM hive;
           Table
---------------------------
 aux_table
 bucketing_cols
 cds
 columns_v2
 compaction_queue
 completed_compactions
 completed_txn_components
 database_params

#查看某个表的所有列
presto> SHOW COLUMNS FROM hive.db_privs;
     Column     |     Type      | Extra | Comment
----------------+---------------+-------+---------
 db_grant_id    | bigint        |       |
 create_time    | integer       |       |
 db_id          | bigint        |       |
 grant_option   | smallint      |       |
 grantor        | varchar(128)  |       |
 grantor_type   | varchar(128)  |       |
 principal_name | varchar(128)  |       |
 principal_type | varchar(128)  |       |
 db_priv        | varchar(128)  |       |
(9 rows)
```

Presto整合hive

1.在kmr集群所有节点Presto的/etc/presto/catalog目录下创建hive.properties，并加入以下内容：

```
connector.name=hive-hadoop2
#【host】 为存放元数据地址
hive.metastore.uri=thrift://【host】:9083
hive.config.resources=/usr/hdp/2.6.1.0-129/hadoop/etc/hadoop/core-site.xml,/usr/hdp/2.6.1.0-129/hadoop/etc/hadoop/hdfs-site.xml
```

2.在kmr集群所有节点Presto的/etc/jvm.config中加入hdfs用户名 -DHADOOP_USER_NAME=hdfs。

3.在Ambari控制台界面中 Services>Presto>Service Actions 重启。

4.在Master-1启动Hive MetaStore。

```
nohup hive --service metastore -p 9083 >/dev/null &
```

5.切换到 presto-client 文件夹中，并且使用 Presto 连接 Hive。

```
/usr/lib/presto/bin/presto-cli --server kmr-4014e7ad-gn-bdf258f3-master-1-001.ksc.com:8285 --catalog hive
```

其中 --catalog 参数表示要操纵的数据库类型，执行成功后即可进入 Presto 的界面，并且直接进入指定的数据库。可以使用 Hive 来查看数据库中的表。

```
presto> show catalogs;
 Catalog
---------
 hive
 mysql
 system
 tpch
(4 rows)

#查看所有数据库
presto> show SCHEMAS FROM hive;
       Schema
--------------------
 db2
 default
 information_schema
 test
 test1
(5 rows)
```

```
# 查看某个Hive数据库下的所有表
presto> show tables FROM hive.db2;
  Table
---------
 stu
 user_01
(2 rows)

#查看某个表中所有数据
presto> select * from hive.db2.stu;
 id |   name    | math | english | pe  | school  | class
----+-----------+------+---------+-----+---------+--------
  1 | zhangsan  |   99 |      98 | 100 | school1 | class1
  2 | lisi      |   59 |      89 |  79 | school1 | class1
  3 | wangwu    |   89 |      99 | 100 | school3 | class1
  4 | zhangsan2 |   99 |      98 | 100 | school1 | class1
  5 | lisi2     |   59 |      89 |  79 | school2 | class1
  6 | wangwu2   |   89 |      99 | 100 | school3 | class1
(6 rows)
```

Storm简介

Storm 是一个实时的、分布式的、可靠的流式数据处理系统。它的工作就是委派各种组件分别独立的处理一些简单任务。在 Storm 集群中处理输入流的是 Spout 组件，而 Spout 又把读取的数据传递给叫Bolt的组件。Bolt组件会对收到的数据元组进行处理，也有可能传递给下一个Bolt。我们可以把 Storm集群想象成一个由Bolt 组件组成的链条集合，数据在这些链条上传输，而Bolt作为链条上的节点来对数据进行处理。

Storm 保证每个消息都会得到处理，而且处理速度非常快，在一个小集群中，每秒可以处理数以百万计的消息。Storm 的处理速度非常惊人：经测试，每个节点每秒可以处理 100 万个数据元组。其主要应用领域有实时分析、在线机器学习、持续计算、分布式 RPC（远过程调用协议，一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。）、ETL（数据抽取、转换和加载）等。

Storm 和 Hadoop 集群表面看上去很类似，但是 Hadoop 上面运行的是 MapReduce Jobs，而在Storm上运行的是拓扑 Topology，这两者之间是非常不一样的，关键区别是：MapReduce 最终会结束，而一个 Topology永远会运行（除非你手动 kill 掉），换句话说，Storm 是面向实时数据分析，而 Hadoop 面向的是离线数据分析。

Storm实践

WordCount程序

1.创建maven工程,导入依赖。

```
<dependency>
    <groupId>org.apache.storm</groupId>
    <artifactId>storm-core</artifactId>
    <version>1.1.0</version>
</dependency>
```

2.创建数据流入源(spout)。

```
/**
 * 读取外部的文件，将一行一行的数据发送给下游的bolt
 * 类似于hadoop MapReduce中的inputformat
 */
public class ReadFileSpout extends BaseRichSpout {
    private SpoutOutputCollector collector;
    private BufferedReader bufferedReader;

    /**
     * 初始化方法，类似于这个类的构造器，只被运行一次
     * 一般用来打开数据连接，打开网络连接。
     *
     * @param conf      传入的是storm集群的配置文件和用户自定义配置文件，一般不用。
     * @param context   上下文对象，一般不用
     * @param collector 数据输出的收集器，spout类将数据发送给collector，由collector发送给storm框架。
     */
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        try {
            bufferedReader = new BufferedReader(new FileReader(new File("//data//test//wordcount.txt")));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        this.collector = collector;
    }
    /**
```

```
     * 下一个tuple，tuple是数据传送的基本单位。
     * 后台有个while循环一直调用该方法，每调用一次，就发送一个tuple出去
     */
    public void nextTuple() {
        String line = null;
        try {
            line = bufferedReader.readLine();
            if (line!=null){
                collector.emit(Arrays.asList(line));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    /**
     * 声明发出的数据是什么
     *
     * @param declarer
     */
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("juzi"));
    }
}
```

3.splitBolt进行单词切割。

```
/**
 * 输入：一行数据
 * 计算：对一行数据进行切割
 * 输出：单词及单词出现的次数
 */
public class SplitBolt extends BaseRichBolt{
    private  OutputCollector collector;
    /**
     * 初始化方法，只被运行一次。
     * @param stormConf 配置文件
     * @param context 上下文对象，一般不用
     * @param collector 数据收集器
     */
    @Override
    public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
            this.collector = collector;
    }
    /**
     * 执行业务逻辑的方法
     * @param input 获取上游的数据
     */
    @Override
    public void execute(Tuple input) {
        // 获取上游的句子
        String juzi = input.getStringByField("juzi");
        // 对句子进行切割
        String[] words = juzi.split(" ");
        // 发送数据
        for (String word : words) {
            // 需要发送单词及单词出现的次数，共两个字段
            collector.emit(Arrays.asList(word,"1"));
        }
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word","num"));
    }
}
```

4.单词计数

```
/**
 * 输入：单词及单词出现的次数
 * 输出：打印在控制台
 * 负责统计每个单词出现的次数
 * 类似于hadoop MapReduce中的reduce函数
 */
public class WordCountBolt extends BaseRichBolt {
    private Map<String, Integer> wordCountMap = new HashMap<String, Integer>();

    /**
     * 初始化方法
     *
     * @param stormConf 集群及用户自定义的配置文件
     * @param context   上下文对象
     * @param collector 数据收集器
     */
```

```
    @Override
    public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
        // 由于wordcount是最后一个bolt，所有不需要自定义OutputCollector collector，并赋值。
    }
    @Override
    public void execute(Tuple input) {
        //获取单词出现的信息（单词、次数）
        String word = input.getStringByField("word");
        String num = input.getStringByField("num");
        // 定义map记录单词出现的次数
        // 开始计数
        if (wordCountMap.containsKey(word)) {
            Integer integer = wordCountMap.get(word);
            wordCountMap.put(word, integer + Integer.parseInt(num));
        } else {
            wordCountMap.put(word, Integer.parseInt(num));
        }
        // 打印整个map
        System.out.println(wordCountMap);
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // 不发送数据，所以不用实现。
    }
}
```

5.WordCountTopology类(main方法)

```
/**
 * wordcount的驱动类，用来提交任务的。
 */
public class WordCountTopology {
    public static void main(String[] args) throws InvalidTopologyException, AuthorizationException, AlreadyAliveException {
        // 通过TopologyBuilder来封装任务信息
        TopologyBuilder topologyBuilder = new TopologyBuilder();
//        设置spout，获取数据
        topologyBuilder.setSpout("readfilespout",new ReadFileSpout(),2);
//        设置splitbolt，对句子进行切割
        topologyBuilder.setBolt("splitbolt",new SplitBolt(),4).shuffleGrouping("readfilespout");
//        设置wordcountbolt，对单词进行统计
        topologyBuilder.setBolt("wordcountBolt",new WordCountBolt(),2).shuffleGrouping("splitbolt");
//        准备一个配置文件
        Config config = new Config();
//        storm中任务提交有两种方式，一种方式是本地模式，另一种是集群模式。
//        LocalCluster localCluster = new LocalCluster();
//        localCluster.submitTopology("wordcount",config,topologyBuilder.createTopology());
        //在storm集群中，worker是用来分配的资源。如果一个程序没有指定worker数，那么就会使用默认值。
        config.setNumWorkers(2);
        //提交到集群
        StormSubmitter.submitTopology("wordcount1",config,topologyBuilder.createTopology());
    }
}
```

6.指定jdk版本插件

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.6.3</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
    </configuration>
</plugin>
```

7.用 mvn clean install 命令编译，然后把target目录下生成的 storm jar包，拷到master-1机器上，执行集群管理-编译代码并提交任务。

```
bin/storm jar storm-samples-jar-with-dependencies.jar com.test.storm.MainTopology clusterStorm
```

8.在Amabari页面中StormUI里面，点进 **clusterStorm** 可以查看该拓扑运行的情况。

Kerberos

本章节将为您介绍什么是Kerberos，及Kerberos开启、使用的流程。 背景信息 集群开启Kerberos认证之后：

- 对客户端而言，在访问集群服务（如HDFS，YARN等）之前，必须先通过Kerberos认证，未经认证的客户端无法访问集群服务，只有经过认证的可信客户端才能访问集群服务、提交作业，有效防止恶意用户冒充客户端向集群提交作业的情况；

- 对服务端而言，集群的服务都是可以信任的，避免了冒充服务的情况。 开启Kerberos认证能够提升集群的安全性，但也增加了集群的使用和维护难度；
- 开启Kerberos前，用户需要对Kerberos的原理、使用有一定的了解，才能更好的使用Kerberos；
- 开启Kerberos后，提交作业的方式与没有开启Kerberos有一些区别，需要对作业进行改造，增加一些Kerberos认证的内容；
- 开启Kerberos后，由于对集群服务的访问加入了Kerberos认证机制，会带来一定的时间开销，相同作业相较于未开启Kerberos的同规格集群执行速度有所下降。 开启Kerberos 在CDP购买-硬件配置页面中，开启Kerberos身份认证。 Kerberos身份认证原理 Kerberos Principal （Kerberos主体） 每个需要使用Kerberos认证服务的用户或者服务都需要一个Kerberos principal，kerberos主体是用户或者服务的唯一标识。 启用Kerberos之后，每个访问集群服务的用户都需要证明自己是Kerberos的某个主体，然后才能使用集群服务。 注意 不但用户需要创建Kerberos principal，服务也需要，CDP会自动为集群中的服务都创建principal，而用户的principal就需要Kerberos管理员去创建了。 Kerberos Keytabs Keytab 文件包含了principal 以及该principal 的加密密钥，通过该文件，集群的服务或者用户可以不需要任何交互即被认证为合法的principal。 Kerberos协议的认证过程
- 第一阶段：KDC对client进行身份认证 KDC是kerberos的服务端程序，客户端在访问集成了kerberos的服务之前，需要先通过KDC的认证。 当通过KDC的认证之后，KDC会向客户端颁发一个TGT（Ticket Granting Ticket）。 如果把集群想象为一栋大楼，TGT就相当于该栋大楼的门禁，只有获得TGT，才能进一步访问大楼中入驻的企业。
- 第二阶段：Service对client的身份认证 当客户端获取到TGT之后，就获取到访问集群服务的资格。 但是，在访问服务之前，客户端需要携带TGT和需要访问的服务名称向KDC获取SGT（Service Granting Ticket），然后携带SGT去访问service。 Kerberos使用
  1. 在master-1节点举例，录到KMR集群master-1节点节点上，执行命令：

```
su hadoop
kinit -kt /etc/kmr/krb5/data/keytabs/hadoop.keytab hadoop/kmr-XXXXX-master-1-1.ksc.com
```

2. 认证完成即可执行hdfs相关操作

   ```
   hadoop fs -ls /
   ```

3. 查看认证的用户

   ```
   klist
   ```