Impala用户权限隔离探究

上文讲述了Hiveserver2实现用户权限隔离的方案，通过使用代理的方式可以使得平台端使用一个可代理账号代理任意用户执行任务提交和HDFS操作，这种特性使得用户的操作类似于用户单独使用一样，但是impala实现方式不一样，在启动impala的时候配置指定使用的keytab和principal账号，之后所有的数据操作都是使用该账号执行的，由于impala的计算框架由自己实现，目前不必依赖于YARN，因此无需代理用户提交任何任务。但是作为数据平台支持不同用户执行操作时，需要设置impala配置的kerberos用户（假设这个用户名为impala）为超级账号（类似于root账号），这样才能无需担心运行时出现数据访问错误。但是所有的操作都使用超级账号执行存在如下弊端：

- 暴露HDFS超级账号，存在可能出现安全问题的隐患，一般运维同学不会给予这个权限。
- 所有数据读写都是通过超级账号执行，即使用户自己写入的表的数据该用户也没有权限访问（例如通过HDFS客户端下载）。
- impala上写入的数据无法被其他系统使用，例如hive执行的时候使用的是被代理用户访问的，impala写入的数据则是超级账号写入的，当然没有权限。

Impala使用实例

假设启动impala的时候配置了如下的kerberos信息：

```
-principal=impala/_HOST@HADOOP.HZ.NETEASE.COM
-be_principal=impala/_HOST@HADOOP.HZ.NETEASE.COM
-keytab_file=/home/impala/impala/impala-kudu-2.8/keytab/impala.keytab
-krb5_conf=/etc/krb5.conf
```

通过impala-shell连接到impalad，使用的是当前shell下认证的kerberos用户，当前认证的用户是nrpt，连接impalad，执行查询发现会出现错误：

```
> klist
Ticket cache: FILE:/tmp/krb5cc_1025
Default principal: nrpt/dev@HADOOP.HZ.NETEASE.COM

Valid starting       Expires              Service principal
02/13/2017 15:39:54  02/14/2017 13:39:54  krbtgt/HADOOP.HZ.NETEASE.COM@HADOOP.HZ.NETEASE.COM
     renew until 02/14/2017 15:39:54

> ./shell.sh
Starting Impala Shell without Kerberos authentication
Error connecting: TTransportException, TSocket read 0 bytes
Kerberos ticket found in the credentials cache, retrying the connection with a secure transport.
Connected to db-87.photo.163.org:21000
Server version: impalad version 2.8.0-cdh5-INTERNAL RELEASE (build 17a10dc48aa740344702618ca87c69010d4e8f45)
***********************************************************************************
Welcome to the Impala shell.
(Impala Shell v2.8.0-cdh5-INTERNAL (746917f) built on Wed Jan  4 09:33:31 CST 2017)

To see more tips, run the TIP command.
***********************************************************************************
[db-87.photo.163.org:21000] > select * from foodmart.store;
Query: select * from foodmart.store
Query submitted at: 2017-02-13 15:40:06 (Coordinator: http://db-87.photo.163.org:25000)
ERROR: IllegalStateException: Create execute request failed: TQueryCtx(...)
```

可以看出，虽然当前认证的用户是nrpt并且该表的owner以及数据路径的owner都是nrpt，但是使用impala查询仍然失败，这是什么原因呢，查看impalad的错误日志发现数据访问权限的错误：

```
Caused by: com.cloudera.impala.catalog.TableLoadingException: Failed to load metadata for table: store
CAUSED BY: CatalogException: Failed to retrieve file descriptors from path hdfs://hz-cluster2/user/nrpt/hive-server/foodmart.db/st
CAUSED BY: AccessControlException: Permission denied: user=impala, access=EXECUTE, inode="/user/nrpt/hive-server":nrpt:hdfs:drwxrw
        at org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.checkFsPermission(FSPermissionChecker.java:271)
        at org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.check(FSPermissionChecker.java:257)
        at org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.checkTraverse(FSPermissionChecker.java:208)
        at org.apache.hadoop.hdfs.server.namenode.FSPermissionChecker.checkPermission(FSPermissionChecker.java:171)
        at org.apache.hadoop.hdfs.server.namenode.FSNamesystem.checkPermission(FSNamesystem.java:5911)
        at org.apache.hadoop.hdfs.server.namenode.FSNamesystem.getFileInfo(FSNamesystem.java:3698)
        at org.apache.hadoop.hdfs.server.namenode.NameNodeRpcServer.getFileInfo(NameNodeRpcServer.java:803)
```

```
        at org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolServerSideTranslatorPB.getFileInfo(ClientNamenodeProtocolServer
        at org.apache.hadoop.hdfs.protocol.proto.ClientNamenodeProtocolProtos$ClientNamenodeProtocol$2.callBlockingMethod(ClientNa
        at org.apache.hadoop.ipc.ProtobufRpcEngine$Server$ProtoBufRpcInvoker.call(ProtobufRpcEngine.java:585)
        at org.apache.hadoop.ipc.RPC$Server.call(RPC.java:928)
        at org.apache.hadoop.ipc.Server$Handler$1.run(Server.java:2013)
        at org.apache.hadoop.ipc.Server$Handler$1.run(Server.java:2009)
        at java.security.AccessController.doPrivileged(Native Method)
        at javax.security.auth.Subject.doAs(Subject.java:415)
        at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1614)
        at org.apache.hadoop.ipc.Server$Handler.run(Server.java:2007)
```

从日志中可以看出，该路径为查询表的数据存储路径，但是访问用户为impala（也就是impala启动的时候配置的kerberos用户），而不是真正的客户端用户nrpt，这就验证了在impala中所有的数据访问都是通过impala配置的用户进行的，进而导致上面提到的后两个问题：普通用户的数据需要为impala用户开通读权限;在impala中数据的写入也是impala用户写入的，普通用户也没有权限访问。

分析

较之与hive和spark，impala不需要向yarn提交任务（不保证后面的版本扔不依赖于yarn），不需要涉及yarn队列等权限，所以对于impala只需要保证数据访问可代理就可以了，使用代理访问可以做到任意的客户端用户访问impala时使用自己的账号操作，而不是全部由impala完成。既然问题确定，那首先来分析一下需要从哪里入手，从需求上来看其实很明确，仿照hive的做法，只需要在执行数据访问操作的时候使用一个可代理账号执行和被代理账号执行doAs访问数据就可以了，那么首先来找一下哪里会访问数据。

Impala由三个组件组成：statestored、catalogd和impalad，statestored作为一个订阅-发布服务实现消息的订阅和通知，它不会解析任何的消息，因此也就不会执行任何的数据读写操作；catalogd负责元数据操作，用户指定的所有的DDL操作都需要通过catalogd实现持久化；impalad负责处理查询，包括查询解析和查询执行。从以上的组件功能可以看出，只有catalogd和impalad服务需要代理访问的支持。

impala代码的实现有两部分：JAVA和C++，JAVA部分实现了元数据访问和查询解析等逻辑，C++部分实现查询的执行和statestore的逻辑，从代码结构可以看出无论是在C++还是JAVA代码中都需要执行数据访问，下面首先看一下这两种语言分别都是如何实现代理访问的。

代理访问实例

下面以C++和JAVA分别实现以代理的方式访问HDFS的逻辑，JAVA本身提供的有较易用的接口，C++接口本身不支持代理方式，需要作出一定的修改，下面测试代码实现如下逻辑：列出目录下的所有文件并且在该目录下创建一个空文件，然后读取文件的数据，最后查看新文件的属主是被代理用户。

JAVA版实现

```
import java.io.IOException;
import java.security.PrivilegedExceptionAction;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.security.UserGroupInformation;


public class FileProxyOperation {
    public static void main(String[] args) throws IOException, InterruptedException {
        if(args.length < 2) {
            System.err.println("./FileProxyOperation proxy_user, location");
            return;
        }
        String proxyUser = args[0];
        final String location = args[1];

        final Configuration conf = new Configuration();
        conf.setBoolean("hadoop.security.authorization", true);
        conf.set("hadoop.security.authentication", "kerberos");
        UserGroupInformation.setConfiguration(conf);
```

```
        UserGroupInformation ugi = UserGroupInformation.createProxyUser(proxyUser, UserGroupInformation.getLoginUser());
        System.out.println("Current kerberos user : " + ugi);
        final Path path = new Path(location);
        ugi.doAs(new PrivilegedExceptionAction<Void>() {
            public Void run() throws Exception {
                FileSystem fs = path.getFileSystem(conf);
                for(FileStatus file : fs.listStatus(path)) {
                    System.out.println("File : " + file.getPath().toString());
                }

                String newFile = location + "/_test_file_";
                FSDataOutputStream ops = fs.create(new Path(newFile));
                String write_data = "Hello, World!";
                ops.write(write_data.getBytes());
                ops.flush();
                ops.close();
                FSDataInputStream ips = fs.open(new Path(newFile));
                byte[] read_data = new byte[256];
                ips.seek(0);
                ips.read(read_data);
                String in_data = new String(read_data);
                assert(write_data.equals(in_data));
                System.out.println("Read data : " + in_data + ", length " + in_data.length());
                ips.close();
                return null;
            }
        });

    }
}
```

编译，执行程序，编译过程需要依赖于hadoop的一些包，至少UserGroupInformation等类存在于hadoop-common下，一般使用hadoop classpath命令来生成hadoop依赖的jar和配置文件。

```
> klist
Ticket cache: FILE:/tmp/krb5cc_1025
Default principal: hive/app-20.photo.163.org@HADOOP.HZ.NETEASE.COM

Valid starting       Expires              Service principal
02/13/2017 16:24:07  02/14/2017 02:24:07  krbtgt/HADOOP.HZ.NETEASE.COM@HADOOP.HZ.NETEASE.COM
     renew until 02/14/2017 16:24:07
> java -cp .:$CLASSPATH FileProxyOperation nrpt hdfs://hz-cluster2/user/nrpt/rdd1
Current kerberos user : nrpt (auth:PROXY) via hive/app-20.photo.163.org@HADOOP.HZ.NETEASE.COM (auth:KERBEROS)
File : hdfs://hz-cluster2/user/nrpt/rdd1/_SUCCESS
File : hdfs://hz-cluster2/user/nrpt/rdd1/_test_file_
File : hdfs://hz-cluster2/user/nrpt/rdd1/part-00000
File : hdfs://hz-cluster2/user/nrpt/rdd1/part-00001
Read data : Hello, World!, length 13
```

然后通过HDFS客户端查看新创建的文件的权限：

```
> ./bin/hadoop fs -ls hdfs://hz-cluster2/user/nrpt/rdd1
Found 4 items
-rw-r--r--   3 nrpt hdfs      0 2016-11-18 17:34 hdfs://hz-cluster2/user/nrpt/rdd1/_SUCCESS
-rw-r-----   3 nrpt hdfs     13 2017-02-14 14:32 hdfs://hz-cluster2/user/nrpt/rdd1/_test_file_
-rw-r--r--   3 nrpt hdfs     10 2016-11-18 17:34 hdfs://hz-cluster2/user/nrpt/rdd1/part-00000
-rw-r--r--   3 nrpt hdfs     11 2016-11-18 17:34 hdfs://hz-cluster2/user/nrpt/rdd1/part-00001
```

可以看出新创建的文件都是是被代理用户(nrpt)创建的，也就实现了我们代理访问数据的需求。

C++版实现

HDFS（Hadoop）是由JAVA开发的，周边的一些工具也都是使用JAVA开发，大家都是使用JAVA客户端，所以很少见到使用C++访问HDFS的实例，不过HDFS提供的有C/C++客户端，但是C++认证Kerberos怎么做？怎么让它实现代理访问呢？带着这个问题查看HDFS的C语言客户端代码实现发现其实客户端的实现都是通过JNI的方式调用JAVA接口实现的，提供的典型API如下：

```
/**
 * hdfsConnectAsUser - Connect to a hdfs file system as a specific user
 * Connect to the hdfs.
 * @param nn   The NameNode.  See hdfsBuilderSetNameNode for details.
 * @param port The port on which the server is listening.
 * @param user the user name (this is hadoop domain user). Or NULL is equivelant to hhdfsConnect(host, port)
 * @return Returns a handle to the filesystem or NULL on error.
 * @deprecated Use hdfsBuilderConnect instead.
 */
 hdfsFS hdfsConnectAsUser(const char* nn, tPort port, const char *user);

/**
 * hdfsOpenFile - Open a hdfs file in given mode.
 * @deprecated    Use the hdfsStreamBuilder functions instead.
 * This function does not support setting block sizes bigger than 2 GB.
 *
 * @param fs The configured filesystem handle.
 * @param path The full path to the file.
 * @param flags - an | of bits/fcntl.h file flags - supported flags are O_RDONLY, O_WRONLY (meaning create or overwrite i.e., impl
 * O_WRONLY|O_APPEND. Other flags are generally ignored other than (O_RDWR || (O_EXCL & O_CREAT)) which return NULL and set errno
 * @param bufferSize Size of buffer for read/write - pass 0 if you want
 * to use the default configured values.
 * @param replication Block replication - pass 0 if you want to use
 * the default configured values.
 * @param blocksize Size of block - pass 0 if you want to use the
 * default configured values.  Note that if you want a block size bigger
 * than 2 GB, you must use the hdfsStreamBuilder API rather than this
 * deprecated function.
 * @return Returns the handle to the open file or NULL on error.
 */
hdfsFile hdfsOpenFile(hdfsFS fs, const char* path, int flags,
                      int bufferSize, short replication, tSize blocksize);

/**
 * hdfsRead - Read data from an open file.
 * @param fs The configured filesystem handle.
 * @param file The file handle.
 * @param buffer The buffer to copy read bytes into.
 * @param length The length of the buffer.
 * @return      On success, a positive number indicating how many bytes
 *              were read.
 *              On end-of-file, 0.
 *              On error, -1.  Errno will be set to the error code.
 *              Just like the POSIX read function, hdfsRead will return -1
 *              and set errno to EINTR if data is temporarily unavailable,
 *              but we are not yet at the end of the file.
 */
tSize hdfsRead(hdfsFS fs, hdfsFile file, void* buffer, tSize length);
```

这些接口类似于Linux文件系统提供的系统调用接口，通过Open方法中的flags参数确定以可读或者可写的方式打开文件，这些接口中主要使用了两个数据结构：hdfsFS 和 hdfsFile ，前者对应着HDFS的FileSystem，后者对应着Java API的FSDataInputStream或者FSDataOutputStream对象（依赖于是以可读还是可写方式打开），在hdfsConnectAsUser接口中有user参数，但是该参数并不是使用代理的方式访问HDFS，通过对C语言接口实现的了解，如果在C接口中实现代理访问的功能，至少需要在每一个接口中都添加被代理的用户，或者上面那两个数据结构中存在被代理用户的信息，前一种方案改动代价太大，需要将所有的接口实现重新撸一遍，后面一种方案需要依赖于现有的FileSystem的实现。在这感觉走投无路的时候查看了FileSystem的创建过程，参考HDFS C语言接口的实现，发现它调用的是FileSystem.get(final URI uri, final Configuration conf, final String user)方法，实现如下：

```
  public static FileSystem get(final URI uri, final Configuration conf,
        final String user) throws IOException, InterruptedException {
    String ticketCachePath =
      conf.get(CommonConfigurationKeys.KERBEROS_TICKET_CACHE_PATH);
    UserGroupInformation ugi =
        UserGroupInformation.getBestUGI(ticketCachePath, user);
    return ugi.doAs(new PrivilegedExceptionAction<FileSystem>() {
```

```
      @Override
      public FileSystem run() throws IOException {
        return get(uri, conf);
      }
    });
  }
```

可以看到该函数会通过getBestUGI函数获取ugi并且使用该ugi创建FileSystem，之后FileSystem的的操作都是使用该ugi作为当前用户，那么只需要在创建FileSystem的时候能够创建出代理的ugi就可以使用代理的方式操作HDFS了，但是遗憾的是getBestUGI函数并没有考虑代理方式：

```
  public static UserGroupInformation getBestUGI(
      String ticketCachePath, String user) throws IOException {
    if (ticketCachePath != null) {
      return getUGIFromTicketCache(ticketCachePath, user);
    } else if (user == null) {
      return getCurrentUser();
    } else {
      return createRemoteUser(user);
    }
  }
```

从该函数的实现可以看到，它首先会检查cache是否存在，通常情况下这个cache不会被传递（从get函数中可以看到是CommonConfigurationKeys.KERBEROS_TICKET_CACHE_PATH配置项配置的），那么如果user被传递时调用了createRemoteUser函数，该函数只会创建一个SIMPLE方式认证的普通用户，因此使用这种方式去连接kerberos认证的HDFS显然会出现错误，那么这里我们是否可以利用该函数创建代理的ugi呢？当然是可以的，既然需要的参数都有了，那么在user不为null的情况下判断是否当前认证的方式是KERBEROS，如果是则创建代理用户，修改之后的getBestUGI为：

```
  public static UserGroupInformation getBestUGI(
      String ticketCachePath, String user) throws IOException {
    if (ticketCachePath != null) {
      return getUGIFromTicketCache(ticketCachePath, user);
    } else if (user == null) {
      return getCurrentUser();
    } else if (isAuthenticationMethodEnabled(AuthenticationMethod.KERBEROS)) {
      return createProxyUser(user, getCurrentUser());
    } else {
      return createRemoteUser(user);
    }
  }
```

这样就可以很简单的在创建FileSystem的时候使用被代理的ugi，为了验证只在FileSystem中设置kerberos ugi是否能够成功执行其他的操作，可以将上面的JAVA代码进行简单的修改，只使用proxy的方式创建出FileSystem，然后使用这个对象执行其余的操作（不用再doAs函数中），修改之后的代码如下：

```
//上面代码保持不变
final Path path = new Path(location);
FileSystem fs = ugi.doAs( new PrivilegedExceptionAction<FileSystem>() {
    public FileSystem run() throws Exception {
        FileSystem fs = path.getFileSystem( conf);
        return fs ;
    }
});
for(FileStatus file : fs .listStatus(path)) {
    System. out.println("File : " + file .getPath().toString());
}
//下面代码保持不变
```

修改之后的代码和改动之前运行结果一致，并不会出现权限之类的问题，说明只需要在创建FileSystem的时候使用的ugi会在之后该对象的所有操作中被使用，这可以说是一个超级惊喜，毕竟对于HDFS C客户端的修改就不需要加任何的被代理用户参数了，只需要修改getBestUGI函数使其可以创建被代理用户的ugi，那么C客户端返回的hdfsFS对象也关联被代理用户身份的，当然是用该对象执行的其余所有文件操作都是使用的该被代理用户，所以HDFS的C接口不需要做任何修改，只需要在getBestUGI函数中做以上小小的修改就可以了（新增一个判断条件和一行代码）。

既然修改hadoop-common（UserGroupInformation这个类在common包下）源代码就需要重新编译整个Hadoop源码，impala使用的是cloudera的hadoop发行版，所以需要在Cloudera-Hadoop源码中checkout对应版本的代码，然后重新编译，编译过程还是比较非主流的：首先需要到hadoop-

maven-plugins目录下执行`mvn install`，然后再跑到根目录下执行`mvn package -Pdist,native -DskipTests -Dtar`编译，当然编译之前可以把不需要的module给注释了（hadoop-common-project之后的module可以都给注释了，包括HDFS/YARN/MAPREDUCE等）。

好了，编译好了新的hadoop-common-xxx.jar，那就是用修改之后的jar测试一下C接口，编写测试代码：

```
#include "hdfs.h"
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(int argc, char **argv) {
    if(argc < 3) {
        fprintf(stderr, "./FileProxyOperation proxy_user, location");
        return -1;
    }
    char* proxyUser = argv[1];
    const char* path = argv[2];
    hdfsFS fs = hdfsConnectAsUser("hdfs://hz-cluster2", 0, proxyUser);
    hdfsFile writeFile = hdfsOpenFile(fs, path, O_WRONLY|O_CREAT, 0, 0, 0);
    if(!writeFile) {
        fprintf(stderr, "Failed to open %s for writing!\n", path);
        return -1;
    }
    char* buffer = "Hello, World!";
    tSize num_written_bytes = hdfsWrite(fs, writeFile, (void*)buffer, strlen(buffer));
    if (hdfsFlush(fs, writeFile)) {
        fprintf(stderr, "Failed to 'flush' %s\n", path);
        return -1;
    }
    hdfsCloseFile(fs, writeFile);

    hdfsFile readFile = hdfsOpenFile(fs, path, O_RDONLY, 0, 0, 0);
    hdfsSeek(fs, readFile, 0);
    char buf[256] = {0};
    int len = hdfsRead(fs, readFile, buf, 256);
    buf[len] = '\0';
    fprintf(stdout, "Read data : %s, length %d\n", buf, len);
    hdfsCloseFile(fs, readFile);
    return 0;
}
```

编译的时候需要注意环境的配置，首先它需要依赖于一些动态链接库，包括hdfs、jvm等，可以通过如下命令设置动态链接库查找路径：

```
export LD_LIBRARY_PATH=$JAVA_HOME/jre/lib/amd64/server/:$HADOOP_HOME/lib/native/:$LD_LIBRARY_PATH
```

然后执行源码编译：

```
gcc ./proxy_hdfs_op.c -I$HADOOP_HOME/include -L/home/hzfengyu/impala/deploy/jdk1.7.0_79/jre/lib/amd64/server/ -L$HADOOP_HOME/lib/n
```

在运行程序之前需要设置JAVA依赖的CLASSPATH，通过`export CLASSPATH=$HADOOP_HOME/bin/hadoop classpath --glob`设置，否则在运行的时候找不到对应的类，设置完成之后运行改程序发现能够实现代理的方式读写文件（执行之前需要保证hadoop classpath中的hadoop-common-xxx.jar已经被替换成修改之后新编译的版本）。运行程序：

```
> ./proxy_hdfs nrpt hdfs://hz-cluster2/user/nrpt/rdd1/_test_file_
Read data : Hello, World!, length 13
```

Impala待修改地方

搞定了两种语言的客户端代理逻辑，已是万事俱备，上面的测试能够使得JAVA和C++代码都能够通过代理的方式访问HDFS，所以对于接下来的修改首先需要找到哪里需要访问HDFS，然后在这些地方想办法获取到合适的代理用户，经过对impala代码的学习，发现如下几个地方存在访问HDFS：

操作	说明	模块	被代理用户	实现语言

metastore加载	启动时加载全部元数据，查看所有表文件和block分布信息等	catalogd	table owner	JAVA
metastore操作	运行时的metastore操作，典型场景包括创建表/新加分区等，需要读写文件	catalogd	cli user	JAVA
SQL解析	SQL的analyze过程，需要查看表文件的权限等	impalad	cli user	JAVA
SQL执行	SQL的执行过程，大部分SQL的执行都需要涉及到数据（HDFS）的读写	impalad	cli user	C++
配置文件读取	诸如权限配置文件的读取，该文件需要存储在HDFS上	impalad	无	JAVA
UDF/UDAF操作	自定义函数需要首先上传至HDFS(离线)，创建时需要将文件下载到本地	impalad	无	C++

以上操作基本上包含了impala中所有涉及HDFS访问的操作，当然还有一部分操作是访问本地文件系统的，这种操作通常不是用户级的，所以全都交给当前启动impala的账号执行即可。接下来的问题就是如何获取代理用户呢？对于metastore加载使用表的属主用户，加载表的时候可以直接获取。但是其它操作需要使用CLI user作为被代理用户，如何获取该用户呢？

客户端代理

类似于上文提到的Hiveserver2客户端的hive.server2.proxy.user参数可以使用代理账号代理指定的任意用户执行操作，impala也存在这样的参数：impala.doas.user，对于impala客户端（hs2接口或者beeswax接口）提交的任何查询在server端都会创建一个SessionState对象保存该查询的全部上下文信息，包括所有的查询参数（覆盖默认操作），其中还有两个比较重要的参数： connected_user和do_as_user，前者保存当前连接的kerberos用户信息，后者保存被代理用户。那么如何判断kerberos用户是否有权限代理被代理用户呢？上文有介绍说hadoop代理的权限是由core-site.xml的superuser配置决定的，但是impala不依赖于该配置判断是否有代理权限，而是通过自己特有的配置：authorized_proxy_user_config，该配置指定哪些用户可以代理哪些用户执行，例如hive用户可以代理任意用户执行，impala用户可以代理nrpt和intern用户执行，则配置为`authorized_proxy_user_config=hive=*;impala=nrpt, intern`。通过这种代理的方式可以直接使用do_as_user作为查询中被代理用户。

总结

上面讨论了如何对impala中的代码做出修改以支持代理的方式执行HDFS数据的访问，总结下来，我们可以通过do_as_user和表的owner获取被代理用户，然后对C++和JAVA代码做出一定的修改（增加创建proxy ugi然后执行doAs操作），还需要对hadoop-common做出简单的修改以支持创建代理用户，对于不需要代理的操作（例如查询的时候不指定impala.doas.user或者普通用户并没有代理权限），那么就使用connected_user作为被代理用户执行操作，对于基于kerberos认证机制下的impala客户端必须指定该参数，因此整个流程可以正确的运行。

支持Ranger

但是在猛犸中还存在另外一种用户：Ranger user，这个信息是登录到平台的用户信息，例如当前使用邮箱登录，则该用户名为邮箱前缀，这个用户主要用于交给ranger检查用户权限，也就是说我们的元数据权限是平台用户级别的，但是数据权限的校验是产品账号级别的（也就是kerberos账号），平台用户对应着具体的人，kerberos账号对应着产品，这样就需要在执行查询的时候除了上面提到的两个参数之外还需要传递第三个参数：ranger_user。这三种user中ranger_user和connected_user是必须填的，当使用代理的时候do_as_user等于用户传入的impala.doas.user，使用该用户校验HDFS的访问权限，不使用代理的时候不设置这个值，使用connected_user校验HDFS的访问权限，始终使用ranger_user校验元数据访问权限。

impala本身只有这两个user信息，为了支持ranger的校验，需要在请求参数中加入ranger.user参数，不携带该参数是可以直接使用当前操作的kerberos账号的用户名作为ranger.user，为了服务端支持该参数，需要对服务端的请求参数添加ranger_user，并且对于校验元数据权限使用的用户做出对应的修改。

但是这种还是存在一个安全弊端的：对于一个普通用户（假设他的平台账号是zhangsan），它属于某一个产品（假设为nrpt），它不希望通过平台访问impala，于是他就拿到nrpt.keytab，然后在连接参数中将ranger.user设置为lisi，这样的话传递到impala的connected_user=nrpt，ranger_user=lisi，于是他可以顺利地以李四的身份做任何有权限的操作，这显然是不合理的（使用ranger做权限认证就是为了避免这种情况），如下存在几种方案：

- 为每一个平台用户创建一个kerberos账号，这样就将ranger_user和connected_user合并成一个，每一个平台用户只能拥有自己的keytab。
- 使用白名单的方式关闭除了平台服务器以外其它的客户端，这样显然是不友好的，但是可以避免出现如上的安全漏洞。
- 为每一个用户创建一个唯一的复杂的key（相当于密码），平台用户和这个key是唯一对应的，用户可以在平台上查看这个key，查询的时候传入用户名和key信息，服务端校验是否有效。

本文主要介绍了如何修改impala以支持以代理的方式隔离用户访问权限，达到和hiveserver2相似的结果，其中详细的介绍了JAVA和C++语言如何代理用户访问HDFS数据，最后介绍了impala的代理请求参数以及如何使用平台用户接入ranger做元数据权限控制，当然当前还会存在一定的风险，本文最后也给出了相应的修改方案。