HDFS 访问控制

——simple及ACL访问控制

Hadoop HDFS对文件和文件夹的权限控制模型与Posix文件系统的权限控制模型一样，每一个文件和文件夹都分配了所有者用户和所有者用户组。每个客户端访问HDFS的过程中，身份凭证由用户名和组列表两部分组成。在HDFS permission模型中，每个文件或者目录以owner,group及其他3个用户class为基本单位，权限控制分为read,write及excute。对于文件系统实体来说，权限被编码成9bit（3\*3）的数据。每个客户端访问HDFS的过程中，身份凭证由用户名和组列表两部分组成，Hadoop进行身份验证的时候，首先验证用户名，如果用户名验证不通过则验证用户组，如果用户名和用户组验证失败则身份验证失败。

这个模型可以满足大部分的安全需求，例如商店有一个经理管理者所有的销售数据，其他员工仅需要查看这部分数据（不能改变），商店之外的人不能查看数据，这个场景可以在文件中使用chmod 640命令，结果为：

-rw-r------ 1 bruce sales 22k Nov 18 10:55 sales-data

但是商店扩大了经营，有了新的请求，经理bruce没有精力管理所有的数据，因此允许diana及clark管理数据。但是Posix model中使用9 bit permission已经不能实现这个权限控制。因为文件只有单个拥有者及用户组，用户组已经用于员工查看数据权限控制。一个变通做法是让file owner成为一个虚拟账户，允许bruce,diana及clark通过sudo(类似其机制)命令来使用这个账户。但是这个做法的缺点是，增加了用户操作复杂度，每个用户都要使用不同的账户执行不同的操作。

从上面的例子可以看出permission bits不能很好的应对非常规的账户管理。为了解决这个问题，引入ACLs(Access Control List)，通过ACLs可以使多用户多群组拥有对文件系统的不同操作权限。

# 一.HDFS 访问控制类图

HDFS访问控制相关的类，如图1所示



图1：访问控制类图

如上图所示：

文件的访问控制通过FsPermissionChecker来实现，其中权限控制实体分成两个部分：

1) FsPermission

在前言中对文件系统访问控制的分析中，文件的简单控制是由Posix 9 bit permission来完成的，FsPermission类就是对其的抽象。

其中userAction是文件owner的可执行权限，groupAtion是所属组的执行权限，otherAction是其他用户的执行权限，在类中比较重要的方式是applyUmask，对FsPermission进行改变。

2) AclEntry

AclEntry中是抽象访问控制列表的权限控制实体，其中AclEntryType的作用是标记该实体是user,group,mask还是其他的用户的实体。AclEntryScope是枚举类，其中包括两项，ACCESS:当scope是该值时，文件的访问控制要检测ACL，DEFAULT:是该值的时候，说明文件的访问控制不适用ACL访问控制，因此不需要检测ACL。

3) FsPermission和AclEntry的初始化

文件的访问控制信息存储在Inode，FsPermission和AclEntry分别对应INode中的permission和feature数据项，其中permission是long类型数据。

在INode中通过getPermission和getPermissionStatus两个方法获取FsPermission，其初始化是通过内部类PermissionStatusFormat来实现，具体过程大概是将对应位的信息抽取出去。

AclEntry是通过方法getAclFeature()来实现的，AclFeature存放着AclEntry列表，在AclEntry有内部类Builder用来构建AclEntry，其中方法parseAclEntry和parseAclSpec可以解析AclEntry字符串，转成AclEntry实体。

FsPermissionChecker在文件访问控制中的作用会在下面进行详细分析。

# 二.访问控制流程

对文件的读写等操作都需要首先进行访问权限的验证，涉及到的方法较多，本处以写文件（append file）为例，说明其流程：

1) DFSclient.append(src,bufferSize,progress)为调用起始，最终调用FsNameSys

tem.appendFileInt

2) 在appendFileInt中，初始化FsPermissionChecker=FsnameSystem.getpemissionC

hecker()来实现，在初始化过程中会对file owener,superGroup及ugi赋值，其中file owner是指启动Namenode的用户，ugi是指调用DFSclient的用户（文件操作的用户）

3) pc初始化后，调用checkPathAccess(pc,src,FsAction.WRITE)，进行文件的写权限验证，最终会调用pc.check(Inode,snapshotId,access)，参数Inode是src的文件管理单元（通过Fsdirectory获取），access是FsAction.WRITE。在INode中会有文件的权限信息，在上文中已经分析（包括acl和permission）

4) 从Inode中获取acl(AclFeature)和mode(Fspermission)，首先调用

checkAccessAcl(inode,access,mode,aclEntris)，检测acl的控制权限，如果acl为空，则说明文件没有使用ACL控制，大致步骤是根据ugi中的user，从acl查找对应用户，然后检测其权限。

如果上面步骤没有确定权限，则调用checkFsPermission(inode,snapshotId,access,

mode)，根据linux文件权限的验证原则进行检测。

# 三.checkAccessAcl和checkFsPermission源码

源码内容较为简单，不再详述，如果权限不通过则抛出AccessControlException

checkAccessAcl()

1. private void checkAccessAcl(INode inode, int snapshotId, FsAction access,
2. FsPermission mode, List<AclEntry> featureEntries)
3. throws AccessControlException {
4. boolean foundMatch = false;
5. // Use owner entry from permission bits if user is owner.
6. if (user.equals(inode.getUserName(snapshotId))) { //首先验证user是否是文件所有者，如果是
7. if (mode.getUserAction().implies(access)) { // 验证其操作权限
8. return;
9. }
10. foundMatch = true;
11. }
12. // Check named user and group entries if user was not denied by owner entry.
13. if (!foundMatch) {
14. for (AclEntry entry: featureEntries) {
15. if (entry.getScope() == AclEntryScope.DEFAULT) {
16. break;
17. }
18. AclEntryType type = entry.getType();
19. String name = entry.getName();
20. if (type == AclEntryType.USER) { //根据ACL类型，如果是USER类型，则根据ACL user匹配
21. // Use named user entry with mask from permission bits applied if user
22. // matches name.
23. if (user.equals(name)) { //匹配成功，则检验其权限
24. FsAction masked = entry.getPermission().and(mode.getGroupAction());
25. if (masked.implies(access)) {
26. return;
27. }
28. foundMatch = true;
29. }
30. } else if (type == AclEntryType.GROUP) { //ACL项是group，则验证user是否是群组的一员，然后验证其 所属群组的权限
31. // Use group entry (unnamed or named) with mask from permission bits
32. // applied if user is a member and entry grants access. If user is a
33. // member of multiple groups that have entries that grant access, then
34. // it doesn't matter which is chosen, so exit early after first match.
35. String group = name == null ? inode.getGroupName(snapshotId) : name;
36. if (groups.contains(group)) {
37. FsAction masked = entry.getPermission().and(mode.getGroupAction());
38. if (masked.implies(access)) {
39. return;
40. }
41. foundMatch = true;
42. }
43. }
44. }
45. }

checkFsPermission()

1. private void checkFsPermission(INode inode, int snapshotId, FsAction access,
2. FsPermission mode) throws AccessControlException {
3. if (user.equals(inode.getUserName(snapshotId))) { //user class
4. if (mode.getUserAction().implies(access)) { return; }
5. }
6. else if (groups.contains(inode.getGroupName(snapshotId))) { //group class
7. if (mode.getGroupAction().implies(access)) { return; }
8. }
9. else { //other class
10. if (mode.getOtherAction().implies(access)) { return; }
11. }
12. throw new AccessControlException(
13. toAccessControlString(inode, snapshotId, access, mode));
14. }