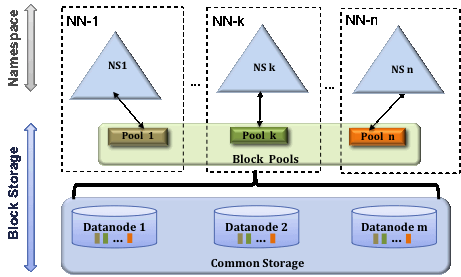
Hadoop ViewFs

ViewFs(View File System)，视图文件系统提供管理多个Hadoop文件系统命名空间，典型的使用是用来管理Federation 集群中多NameNode的命名空间文件系统。在HDFS Federation模式下：



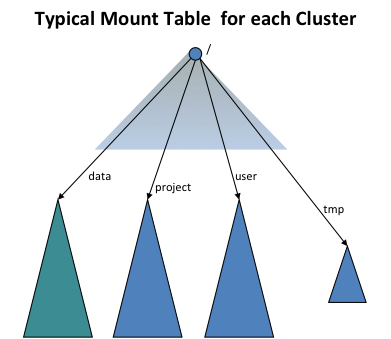
有多个NameNode，对应多个NameSpace，命名空间之间相互独立，因此其管理的文件也是相互独立，不能共享。

在访问Federation集群中的文件时，要指定文件及其所在的NameSpace(即指定NameNode)，例如访问访问ns1上的文件目录fool，路径为：

*hdfs://namenodeOfNS1:port/foo/bar*

从HDFS和HBase的使用者角度来看，仅使用单NameNode上管理的数据时是没有问题的，但是考虑到HDFS上的计算类应用，比如YARN/MR应用程序，可能会涉及到跨NN的数据读写，必须显式的指定全URL，为了解决这种麻烦，需要使用ViewFS为用户提供统一的全局HDFS访问入口。

在Federation中，借鉴Linux提供client-side mount table，其通过一层新的文件系统viewfs实现，实际上提供一种映射关系，将一个全局（逻辑）目录映射到某个具体的NN(物理)目录上：



在core-site.xml中的配置示例如下：

*<xi:include href="mount-table.xml" />*

*<property>*

*<name>fs.defaultFS</name>*

*<value>viewfs://brqviewfs</value>*

*<final>true</final>*

*</property>*

其中brqviewfs是整个HDFS集群的名称，mount-table.xml配置了全局（逻辑）目录与具体NN(物理)目录的映射关系，可以类比Linux挂载点。例如有两个NN，分别是brqrzt和brqrzt1，其中brqrzt管理/data目录，brqrzt1管理/project,/user和/tmp目录，则mount-table.xml的配置内容如下：

*<?xml version="1.0"?>*

*<configuration xmlns:xi="http://www.w3.org/2001/XInclude">*

*<property>*

*<name>fs.viewfs.mounttable.brqviewfs.link./data</name>*

*<value>hdfs://brqrzt/data</value>*

*</property>*

*<property>*

*<name>fs.viewfs.mounttable.brqviewfs.link./user</name>*

*<value>hdfs://brqrzt1/user</value>*

*</property>*

*<property>*

*<name>fs.viewfs.mounttable.brqviewfs.link./project</name>*

*<value>hdfs://brqrzt1/project</value>*

*</property>*

*<property>*

*<name>fs.viewfs.mounttable.brqviewfs.link./tmp</name>*

*<value>hdfs://brqrzt1/tmp</value>*

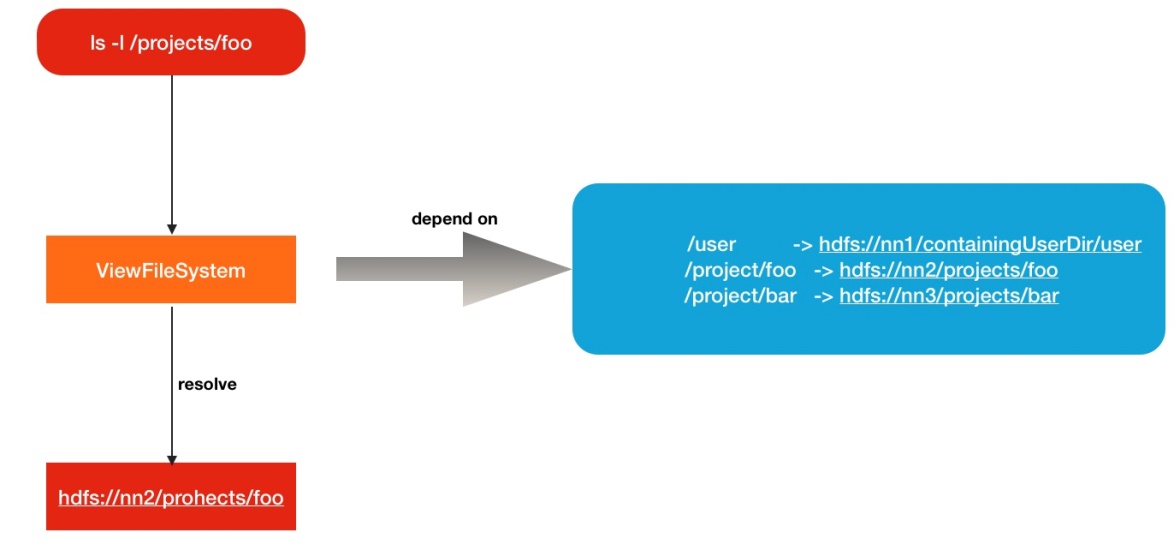
*</property>*

*</configuration>*

经过上面的配置后，可以向访问单NN一样访问访问Federation模式下的HDFS文件。

# ViewFileSystem解析流程

ViewFs的核心流程是路径的路由解析，下面是一个简单的原理图:



当访问/project/foo目录时，基于mount table找到真实的集群的目录。下图是相关的类：



## 目录挂载点

挂载点是路由解析的核心，其定义如下：

*static class MountPoint<T> {*

*String src; //源路径*

*INodeLink<T> target; //目标目录的挂载Link，可以执行*

*MountPoint(String srcPath, INodeLink<T> mountLink) {*

*src = srcPath;*

*target = mountLink;*

*}*

*}*

其中InodeLink指向目标目录，源码如下：

*static class INodeLink<T> extends INode<T> {*

*final boolean isMergeLink; // true if MergeLink*

*final URI[] targetDirLinkList; //可以指向执行多个路径*

*final T targetFileSystem;*

*final String fullPath*

*}*

在不同集群间有相同名称目录的情况，可以配置一对多的目录，示例如下：

*fs.viewfs.mounttable.default.linkMerge./user= hdfs://nnUser1/user,hdfs://nnUser1/user*

## 挂载点的解析和存放

在ViewFileSystem的初始化操作中，挂载点的解析与存放是一个非常重要的过程，器质性在下面的变量中进行：

*InodeTree<FileSystem> fsState; // the fs state; ie the mount table*

ViewFileSystem的initialize的实现源码如下：

*public void initialize(final URI theUri, final Configuration conf)*

*throws IOException {*

*super.initialize(theUri, conf);*

*setConf(conf);*

*config = conf;*

*//根据配置构建client side mount table.*

*final String authority = theUri.getAuthority();*

*try {*

*myUri = new URI(FsConstants.VIEWFS\_SCHEME, authority, "/", null, null);*

*//传入conf信息，进行fsState初始化*

*fsState = new InodeTree<FileSystem>(conf, authority) {*

*......*

*};*

*workingDir = this.getHomeDirectory();*

*} catch (URISyntaxException e) {*

*throw new IOException("URISyntax exception: " + theUri);*

*}*

*}*

InodeTree的构造方法如下：

*protected InodeTree(final Configuration config, final String viewName)*

*throws UnsupportedFileSystemException, URISyntaxException,*

*FileAlreadyExistsException, IOException {*

*……*

*//viewFs的Home dir fs.viewfs.mounttable.<viewFS>.homedir*

*homedirPrefix = ConfigUtil.getHomeDirValue(config, vName);*

*//根view root，InodeDir为 ”/”*

*root = new INodeDir<T>("/", UserGroupInformation.getCurrentUser());*

*root.InodeDirFs = getTargetFileSystem(root);*

*root.isRoot = true;*

*final String mtPrefix = Constants.CONFIG\_VIEWFS\_PREFIX + "." + vName + ".";*

*final String linkPrefix = Constants.CONFIG\_VIEWFS\_LINK + ".";*

*final String linkMergePrefix = Constants.CONFIG\_VIEWFS\_LINK\_MERGE + ".";*

*boolean gotMountTableEntry = false;*

*final UserGroupInformation ugi = UserGroupInformation.getCurrentUser();*

*for (Entry<String, String> si : config) {*

*//判断配置的前缀是否为：fs.viewfs.mounttable.<vName>*

*final String key = si.getKey();*

*if (key.startsWith(mtPrefix)) {*

*//根据配置，获取目标映射的真实路径，可能为多个，以”,”隔开*

*String src = key.substring(mtPrefix.length());*

*if (src.startsWith(linkPrefix)) {*

*src = src.substring(linkPrefix.length());*

*} else if (src.startsWith(linkMergePrefix)) { // A merge link*

*isMergeLink = true;*

*src = src.substring(linkMergePrefix.length());*

*} else if (src.startsWith(Constants.CONFIG\_VIEWFS\_HOMEDIR)) {*

*// ignore - we set home dir from config*

*continue;*

*}…..*

*final String target = si.getValue(); // link or merge link*

*createLink(src, target, isMergeLink, ugi);*

*}*

*}*

*}*

真正实现挂载点的关系存储其实是在createLink方法中，该方法源码如下：

*private void createLink(final String src, final String target,*

*final boolean isLinkMerge, final UserGroupInformation aUgi)*

*throws URISyntaxException, IOException,*

*FileAlreadyExistsException, UnsupportedFileSystemException {*

*// 校验src路径为绝对路径*

*final Path srcPath = new Path(src);*

*if (!srcPath.isAbsoluteAndSchemeAuthorityNull()) {*

*throw new IOException("ViewFs:Non absolute mount name in config:" + src);*

*}*

*//将待添加的路径按照"/"分隔符进行拆分*

*final String[] srcPaths = breakIntoPathComponents(src);*

*//设置当前节点为根节点*

*INodeDir<T> curInode = root;*

*int i;*

*………*

在createLink中，生成InodeDir对象，设置并当前curlNode为根节点，InodeDir类定义如下：

*static class INodeDir<T> extends INode<T> {*

*final Map<String,INode<T>> children = new HashMap<String,INode<T>>();*

*T InodeDirFs = null; // file system of this internal directory of mountT*

*boolean isRoot = false;*

*final String fullPath*

*}*

InodeDir是父子关系的对象，并且每个目录会有对应自身的文件系统，孩子中可能会有InodeDir或是Inode的子类，路径按照”/”进行划分，ViewFileSystem是按照书写结构的存放方式进行挂载点的存储的，如下所示，从最相近的目录树寻找：

*// 根据src路径，生成INodeDir，即client mount table，最后一个component不处理*

*for (i = 1; i < srcPaths.length-1; i++) {*

*//获取当前的子段字符串*

*final String iPath = srcPaths[i];*

*//从当前的目录Inode中进行查找*

*INode<T> nextInode = curInode.resolveInternal(iPath);*

*//如果没有查到，意味着当前的节点中没有此路径下对应的信息*

*if (nextInode == null) {*

*//新增此路径对应的targetFileSystem信息*

*INodeDir<T> newDir = curInode.addDir(iPath, aUgi);*

*newDir.InodeDirFs = getTargetFileSystem(newDir);*

*//并以此作为下个节点，即为找到的目标节点*

*nextInode = newDir;*

*}*

*if (nextInode instanceof INodeLink) { //此节点已经是InodeLink信息，抛异常*

*// Error - expected a dir but got a link*

*throw new FileAlreadyExistsException("Path " + nextInode.fullPath +*

*" already exists as link");*

*} else {*

*//如果是Inode目录，则将子目录作为当前目录，往下寻找*

*assert(nextInode instanceof INodeDir);*

*curInode = (INodeDir<T>) nextInode;*

*}*

*}*

找到最后一层的目录树后，加入新的URL的Link关联信息

*//处理最后一个path component，作为最底层目录，在该目录下添加InodeLink连接，两种情况：link不存在和Link已经存在*

*String iPath = srcPaths[i];// last component*

*if (curInode.resolveInternal(iPath) != null) {*

*// directory/link already exists*

*StringBuilder strB = new StringBuilder(srcPaths[0]);*

*for (int j = 1; j <= i; ++j) {*

*strB.append('/').append(srcPaths[j]);*

*}*

*throw new FileAlreadyExistsException("Path " + strB +*

*" already exists as dir; cannot create link here");*

*}*

*//创建新的link*

*final INodeLink<T> newLink;*

*final String fullPath = curInode.fullPath + (curInode == root ? "" : "/")*

*+ iPath;*

*if (isLinkMerge) { // Target is list of URIs*

*String[] targetsList = StringUtils.getStrings(target);*

*URI[] targetsListURI = new URI[targetsList.length];*

*int k = 0;*

*for (String itarget : targetsList) {*

*targetsListURI[k++] = new URI(itarget);*

*}*

*newLink = new INodeLink<T>(fullPath, aUgi,*

*getTargetFileSystem(targetsListURI), targetsListURI);*

*} else {*

*newLink = new INodeLink<T>(fullPath, aUgi,*

*getTargetFileSystem(new URI(target)), new URI(target));*

*}*

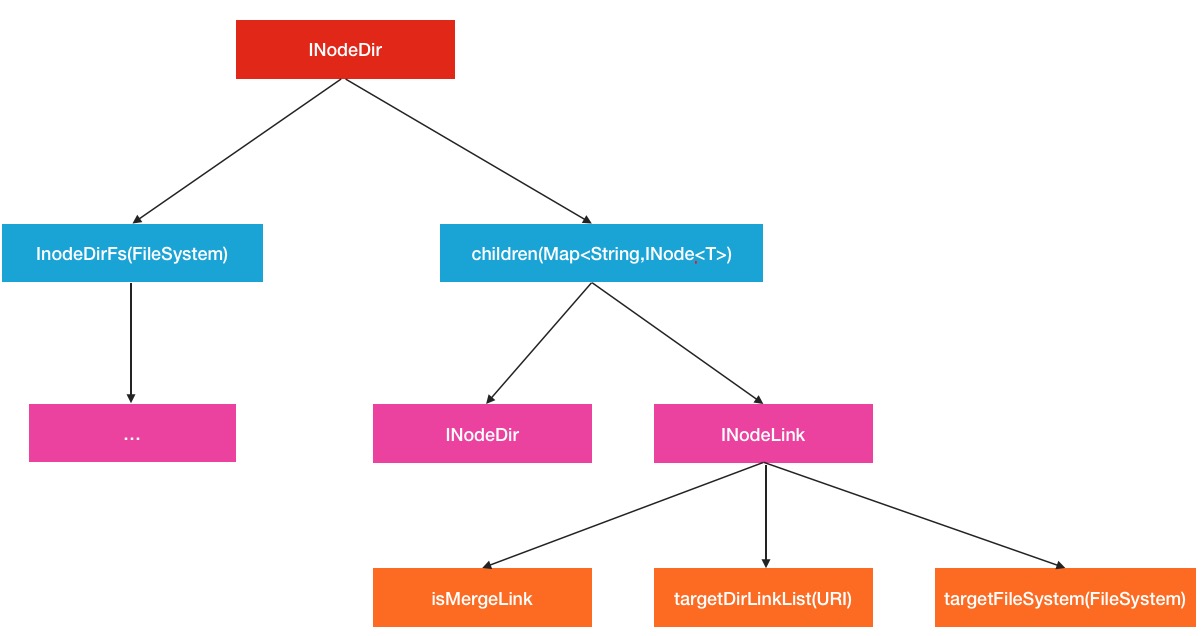
*curInode.addLink(iPath, newLink);*

*//将新的link添加到mountPoints中*

*mountPoints.add(new MountPoint<T>(src, newLink));*

*}*

根据上面的分析，最终的targetFileSystem和具体信息都在InodeLink中，所有的挂载目录点都被以key字符串被树形的拆开存放。另一种说法，在ViewFileSystem中输入viewFileSystem中配置的查询路径，会被逐层解析到对应的InodeDir，最终取出对应的InodeLink，存储模型图如下：



解析的逻辑是通过InodeDir的存储关系逐层的去找。其中targetFileSystem是从uri中获取，代码如下：

*public static FileSystem get(URI uri, Configuration conf) throws IOException {*

*String scheme = uri.getScheme();*

*String authority = uri.getAuthority();*

*//uri中没有配置scheme，则使用默认fs.defaultFS中的配置，该参数默认值：file:///本地文件系统*

*if (scheme == null && authority == null) { // use default FS*

*return get(conf);*

*}*

*if (scheme != null && authority == null) { // no authority*

*//no authority，则使用默认uri*

*URI defaultUri = getDefaultUri(conf);*

*if (scheme.equals(defaultUri.getScheme()) // if scheme matches default*

*&& defaultUri.getAuthority() != null) { // & default has authority*

*return get(defaultUri, conf); // return default*

*}*

*}*

*String disableCacheName = String.format("fs.%s.impl.disable.cache", scheme);*

*if (conf.getBoolean(disableCacheName, false)) {*

*//基于scheme,构建createFileSytem*

*return createFileSystem(uri, conf);*

*}*

*return CACHE.get(uri, conf);*

*}*

*private static FileSystem createFileSystem(URI uri, Configuration conf*

*) throws IOException {*

*Tracer tracer = FsTracer.get(conf);*

*TraceScope scope = tracer.newScope("FileSystem#createFileSystem");*

*scope.addKVAnnotation("scheme", uri.getScheme());*

*try {*

*Class<?> clazz = getFileSystemClass(uri.getScheme(), conf);*

*FileSystem fs = (FileSystem)ReflectionUtils.newInstance(clazz, conf);*

*fs.initialize(uri, conf);*

*return fs;*

*} finally {*

*scope.close();*

*}*

# 2.ViewFileSystem的请求处理

下面介绍ViewFileSystem如何处理客户的发来的HDFS的请求，以mkdir为例，源码如下：

*public boolean mkdirs(final Path dir, final FsPermission permission)*

*throws IOException {*

*//通过fsState对象进行解析*

*InodeTree.ResolveResult<FileSystem> res =*

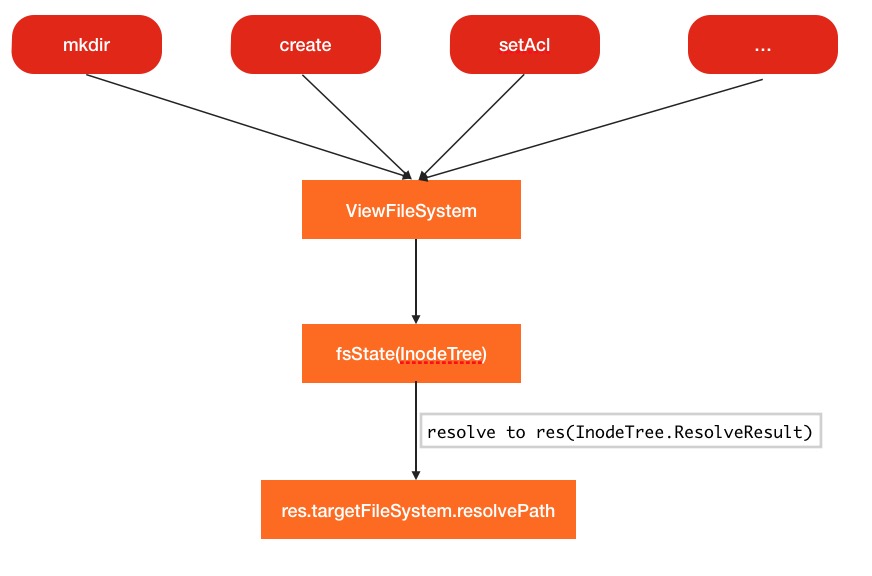
*fsState.resolve(getUriPath(dir), false);*

*//获取目标真实文件系统进行对应的请求处理*

*return res.targetFileSystem.mkdirs(res.remainingPath, permission);*

*}*

fsState.resolve就是在InodeDir中进行逐层寻找，找到对应的文件系统后，然后把路径作为参数传入真实文件系统中，过程如下所示：



解析过程如下所示：

*ResolveResult<T> resolve(final String p, final boolean resolveLastComponent)*

*throws FileNotFoundException {*

*//将path进行split，然后逐层寻找*

*String[] path = breakIntoPathComponents(p);*

*//根"/"路径，直接返回root*

*if (path.length <= 1) { // special case for when path is "/"*

*ResolveResult<T> res =*

*new ResolveResult<T>(ResultKind.isInternalDir,*

*root.InodeDirFs, root.fullPath, SlashPath);*

*return res;*

*}*

*INodeDir<T> curInode = root;*

*for (i = 1; i < path.length - (resolveLastComponent ? 0 : 1); i++) {*

*//根据传入的path[i]，获取对应的INodeDir*

*INode<T> nextInode = curInode.resolveInternal(path[i]);*

*......*

*//如果获取到的inode为INodeLink,返回结果*

*if (nextInode instanceof INodeLink) {*

*final INodeLink<T> link = (INodeLink<T>) nextInode;*

*......*

*final ResolveResult<T> res =*

*new ResolveResult<T>(ResultKind.isExternalDir,*

*link.targetFileSystem, nextInode.fullPath, remainingPath);*

*return res;*

*} else if (nextInode instanceof INodeDir) {*

*//逐层获取*

*curInode = (INodeDir<T>) nextInode;*

*}*

*}*

*//获取到的是mout table的interal dir*

*Path remainingPath;*

*if (resolveLastComponent) {*

*remainingPath = SlashPath;*

*} else {*

*// note we have taken care of when path is "/" above*

*// for internal dirs rem-path does not start with / since the lookup*

*// that follows will do a children.get(remaningPath) and will have to*

*// strip-out the initial /*

*StringBuilder remainingPathStr = new StringBuilder("/" + path[i]);*

*for (int j = i+1; j< path.length; ++j) {*

*remainingPathStr.append('/').append(path[j]);*

*}*

*remainingPath = new Path(remainingPathStr.toString());*

*}*

*final ResolveResult<T> res =*

*new ResolveResult<T>(ResultKind.isInternalDir,*

*curInode.InodeDirFs, curInode.fullPath, remainingPath);*

*return res;*

*}*

# 3.ViewFileSystem的路径封装

ViewFileSystem作为一个视图文件系统，要保持在逻辑上完全一致，所以对返回的文件属性信息，要做一层包装和适配，如下所示：

挂载信息：

*/project/viewFsTmp -> hdfs://nn1/projects/Tmp*

前者是ViewFileSystem路径，后者是真实文件系统存放路径，在真实文件系统中假设有3个子文件：

*/projects/Tmp/child1*

*/projects/Tmp/child2*

*/projects/Tmp/child3*

在ViewFileSystem使用场景下，用hadoop fs –ls /project/viewFsTemp的命令去查看，出现的信息应该是：

*/projects/viewFsTmp/child1*

*/projects/viewFsTmp/child2*

*/projects/viewFsTmp/child3*

因为mount 信息文件路径已经变更，一切都会按照viewfs中的配置路径来，所以需要对真实返回的FileStatus做一层包装，主要是返回的路径，但是对应文件大小、修改时间等基本属性信息直接返回原结果即可。

封装的FileStatus类为ViewFsFileStatus，类如下所示：

*class ViewFsFileStatus extends FileStatus {*

*final FileStatus myFs;*

*Path modifiedPath;*

*ViewFsFileStatus(FileStatus fs, Path newPath) {*

*myFs = fs;*

*modifiedPath = newPath;*

*}*

*}*

在此类中对getPath进行重写：

*@Override*

*public Path getPath() {*

*return modifiedPath;*

*}*

对于其他的基本属性方法，直接调用原来的Status

*......*

*public long getBlockSize() {*

*return myFs.getBlockSize();*

*}*

*@Override*

*public short getReplication() {*

*return myFs.getReplication();*

*}*

*@Override*

*public long getModificationTime() {*

*return myFs.getModificationTime();*

*}*

*@Override*

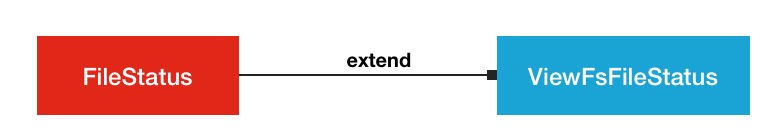
*public long getAccessTime() {*

*return myFs.getAccessTime();*

*}*

*......*

类关系图如下所示：



其中还有ViewFsLocatedFileStatus，继承LocatedFileStatus，不再介绍。

参考文献：

http://hadoop.apache.org/docs/current/api/org/apache/hadoop/fs/viewfs/ViewFs.html

http://blog.csdn.net/yuxin6866/article/details/55263140

http://blog.csdn.net/skywalker\_only/article/details/40373643

http://blog.csdn.net/androidlushangderen/article/details/51315618