20180321_Hadoop 不同节点间的通信机制

总结如下：
1- 不同节点间通过RPC机制进行通信
2- 自己实现了一套节点通信机制，称为Hadoop IPC（Inter-Process Communication，进程间通信）
3- 这是一种简洁，低消耗的通信机制，可以精确控制进程间通信中如连接、超时、缓存等细节。
4- Hadoop IPC机制的实现使用了Java动态代理，Java NIO等技术。

原文链接： http://blog.csdn.net/workformywork/article/details/17391725

后面代码未整理！！！

在Hadoop中，不同节点间使用RPC机制进行通信，在Java中比较典型的RPC是RMI(Remote Method Invocation)的调用方式，虽然Hadoop使用Java语言实现的，但是Hadoop并没有使用RMI实现RPC，而是实现了一套自己独有的节点通信机制，称为Hadoop IPC（Inter-Process Communication，进程间通信）。这是一种简洁，低消耗的通信机制，可以精确控制进程间通信中如连接、超时、缓存等细节。Hadoop IPC机制的实现使用了Java动态代理，Java NIO等技术。关于Java动态代理技术可以参考博文Java动态代理，Java NIO技术可以参考博文Java NIO。

如何使用Hadoop IPC

与Hadoop IPC相关的几个类在org.apache.hadoop.ipc包中，共7个类文件。根据Hadoop提供的IPC机制，下面就来着手开发一个使用Hadoop IPC实现客户端调用服务器端方法的示例，这个示例源于《Hadoop技术内幕：深入解析Hadoop Common和HDFS架构设计与实现原理》这本书，功能是返回服务器端的一个文件信息，文件信息的类为IPCFileStatus：

[java] view plain copy

```
1. package org.hadoopinternal.ipc;
2.
3. import java.io.DataInput;
4. import java.io.DataOutput;
5. import java.io.IOException;
6. import java.util.Date;
7.
8. import org.apache.hadoop.io.Text;
9. import org.apache.hadoop.io.Writable;
10. import org.apache.hadoop.io.WritableFactories;
11. import org.apache.hadoop.io.WritableFactory;
12.
```

```
13. public class IPCFileStatus implements Writable {
14.     private String filename;
15.     private long time;
16. 
17.     public IPCFileStatus() {
18.     }
19. 
20.     public IPCFileStatus(String filename) {
21.         this.filename=filename;
22.         this.time=(new Date()).getTime();
23.     }
24. 
25.     public String getFilename() {
26.         return filename;
27.     }
28. 
29.     public void setFilename(String filename) {
30.         this.filename = filename;
31.     }
32. 
33.     public long getTime() {
34.         return time;
35.     }
36. 
37.     public void setTime(long time) {
38.         this.time = time;
39.     }
40. 
41.     public String toString() {
42.         return "File: "+filename+" Create at "+(new Date(time));
43.     }
44. 
45.     @Override
46.     public void readFields(DataInput in) throws IOException {
47.         this.filename = Text.readString(in);
48.         this.time = in.readLong();
49.     }
50. 
51.     @Override
52.     public void write(DataOutput out) throws IOException {
53.         Text.writeString(out, filename);
54.         out.writeLong(time);
55.     }
56. }
```

由于IPCFileStatus类的对象需要从服务器端传到客户端，所以就需要进行序列化，Writable接口就是Hadoop定义的一个序列化接口，与Java中的Serializable接口类似，但是强化了Serializable的功能。

由于客户端要调用服务器的方法，所以客户端需要知道服务器有哪些方法可以调用，在IPC中使用的是定义公共接口的方法，**如定义一个IPC接口，客户端和服务器端都知道这个接口，客户端通过IPC获取到一个服务器端这个实现了接口的引用，待要调用服务器的方法时，直接使用这个引用来调用方法，这样就可以调用服务器的方法了。**定义一个服务器端和客户端共有的接口IPCQueryStatus：

[java] view plain copy

```
1. package org.hadoopinternal.ipc;
2.
3. import org.apache.hadoop.ipc.VersionedProtocol;
4.
5. public interface IPCQueryStatus extends VersionedProtocol {
6.     IPCFileStatus getFileStatus(String filename);
7. }
```

在接口IPCQueryStatus中，定义了一个getFileStatus(String filename)方法根据文件名得到一个IPCFileStatus对象，注意到IPCQueryStatus接口继承自接口org.apache.hadoop.ipc.VersionedProtocol接口，VersionedProtocol接口是Hadoop IPC接口必须继承的一个接口，它定义了一个方法getProtocolVersion()，用于返回服务器端的接口实现的版本号，有两个参数，分别是协议接口对应的接口名称protocol和客户端期望服务器的版本号clientVersion，主要作用是检查通信双方的接口是否一致，VersionedProtocol的代码如下：

[java] view plain copy

```
1. package org.apache.hadoop.ipc;
2.
3. import java.io.IOException;
4.
5. public interface VersionedProtocol {
6.
7.   public long getProtocolVersion(String protocol,
8.                                  long clientVersion) throws IOException;
9. }
```

定义好了接口，那么在服务器端就需要有一个接口的实现类，用于实现具体的业务逻辑，下面的IPCQueryStatusImpl类实现了IPCQueryStatus接口，仅仅简单实现了IPCQueryStatus规定两个方法，代码如下：

[java] view plain copy

```
1. package org.hadoopinternal.ipc;
2.
3. import java.io.IOException;
4.
5. public class IPCQueryStatusImpl implements IPCQueryStatus {
6.     public IPCQueryStatusImpl() {}
7.     @Override
8.     public IPCFileStatus getFileStatus(String filename) {
9.         IPCFileStatus status=new IPCFileStatus(filename);
10.        System.out.println("Method getFileStatus Called, return: "+status);
11.        return status;
```

```
12.     }
13.     /**
14.      * 用于服务器与客户端，进行IPC接口版本检查，再服务器返回给客户端时调用，如果服务器端的IPC版本与客户端不一致
15.      * 那么就会抛出版本不一致的异常
16.      */
17.     @Override
18.     public long getProtocolVersion(String protocol, long clientVersion) throws IOException {
19.         System.out.println("protocol: "+protocol);
20.         System.out.println("clientVersion: "+clientVersion);
21.         return IPCQueryServer.IPC_VER;
22.     }
23. }
```

getFileStatus()方法根据参数filename创建了一个IPCFileStatus对象，getProtocolVersion()方法返回服务器端使用的接口版本。接口和实现类都完成之后就可以用客户端和服务器进行通信了。服务器端进行一些成员变量的初始化，然后使用Socket绑定IP，然后在某个端口上监听客户端的请求，代码如下：

[java] view plain copy

```
1.  package org.hadoopinternal.ipc;
2.
3.  import org.apache.hadoop.conf.Configuration;
4.  import org.apache.hadoop.ipc.RPC;
5.  import org.apache.hadoop.ipc.Server;
6.
7.  public class IPCQueryServer {
8.      public static final int IPC_PORT = 32121;
9.      public static final long IPC_VER = 5473L;
10.
11.     public static void main(String[] args) {
12.         try {
13.             Configuration conf = new Configuration();
14.             IPCQueryStatusImpl queryService=new IPCQueryStatusImpl();
15.             System.out.println(conf);
16.             Server server = RPC.getServer(queryService, "0.0.0.0", IPC_PORT, 1, true, conf);
17.             server.start();
18.
19.             System.out.println("Server ready, press any key to stop");
20.             System.in.read();
21.
22.             server.stop();
23.             System.out.println("Server stopped");
24.         } catch (Exception e) {
25.             e.printStackTrace();
26.         }
27.     }
```

```
28. }
```

在服务器端先创建一个IPCQueryStatusImpl的对象，传递到RPC.getServer()方法中。服务器端使用RPC.getServer()方法穿给创建服务器端对象server，代码中RPC.getServer()方法的几个参数说明如下：

- 第一个参数queryService标识该服务器对象对外提供的服务对象实例，即客户端所要调用的具体对象，下面客户端的代码调用的接口如此对应；
- 第二个参数"0.0.0.0"表示监绑定所有的IP地址；
- 第三个参数IPC_PORT表示监听的端口；
- 第四个参数1表示Server端的Handler实例（线程）的个数为1
- 第五个参数true表示打开调用方法日志；
- 第六个参数是Configuration对象，用于定制Server端的配置

创建Server对象之后，调用Server.start()方法开始监听客户端的请求，并根据客户端的请求提供服务。

客户端需要先获取到一个代理对象，然后才能进行方法调用，在IPC中，使用RPC.getProxy()方法获取代理对象。客户端的代码如下：

[java] view plain copy

```
1. package org.hadoopinternal.ipc;
2. 
3. import java.net.InetSocketAddress;
4. 
5. import org.apache.hadoop.conf.Configuration;
6. import org.apache.hadoop.ipc.RPC;
7. 
8. public class IPCQueryClient {
9.     public static void main(String[] args) {
10.         try {
11.             System.out.println("Interface name: "+IPCQueryStatus.class.getName());
12.             System.out.println("Interface name: 
    "+IPCQueryStatus.class.getMethod("getFileStatus", String.class).getName());
13. 
14.             InetSocketAddress addr=new InetSocketAddress("localhost", 
    IPCQueryServer.IPC_PORT);
15. 
16.             IPCQueryStatus query=(IPCQueryStatus) RPC.getProxy(IPCQueryStatus.class, 
    IPCQueryServer.IPC_VER, addr,new Configuration());
17.             IPCFileStatus status=query.getFileStatus("/tmp/testIPC");
18.             System.out.println(status);
19.             RPC.stopProxy(query);
20.         } catch (Exception e) {
21.             e.printStackTrace();
22.         }
23.     }
24. }
```

客户端的代码很简单，首先构造一个要请求服务器的网络地址（IP和端口），然后通过RPC.getProxy()方法获取到一个IPCQueryStatus对象，然后进行相应的方法调用。其中客户端代码中RPC.getProxy()方法的参数说明如下：

- 第一个参数是IPC接口对象，可以通过IPC接口的静态成员class直接获得。接口的静态成员class保存了该接口的java.lang.Class实例，它表示正在运行的Java应用程序中的类和接口，提供一系列与Java反射相关的重要功能；
- 第二个参数是接口版本，由于接口会根据需求不断地进行升级，形成多个版本的IPC接口，如果客户端和服务器端使用的IPC接口版本不一致，结果将是灾难性的，所以在建立IPC时，需要对IPC的双方进行版本检查；
- 第三个参数是服务器的Socket地址，用于建立IPC的底层TCP连接；
- 第四个参数是Configuration对象，用于定制IPC客户端参数；

客户端的代码编写完成之后就可以运行程序了，先启动服务器端，再运行一个客户端，就完成了一次客户端调用服务器的过程，客户端调用了服务器端IPCQueryStatusImpl对象的getFileStatus()方法，服务器端返回了方法调用结果即IPCFileStatus对象。

客户端是如何调用服务器的呢？使用了Java动态代理方式，简单分析一下客户端的调用过程。

首先通过RPC.getProxy()方法获得一个远程调用对象，其实在创建对象的过程中并没有与服务器端交互，只是在远程对象创建成功之后调用了服务器端检查服务器接口版本的方法，所以在上面的示例的运行过程中，IPCQueryStatusImpl的检查服务器版本的方法getProtocolVersion()会先调用，这次调用是在RPC.getProxy()方法调用过程中进行的，将下面的两行代码注释掉，也同样会调用getProtocolVersion()方法

[java] view plain copy

```
1. IPCFileStatus status=query.getFileStatus("/tmp/testIPC");
2. System.out.println(status);
```

RPC.getProxy()方法有9个重载方法，其中8个方法都是调用的那个有9个参数的重载方法，下面来看看这个实际调用的方法，代码如下：

[java] view plain copy

```
1. /** Construct a client-side proxy object that implements the named protocol,
2.    * talking to a server at the named address.
3.    * @see #getProxy(Class, long, InetSocketAddress, UserGroupInformation, Configuration,
   SocketFactory, int)
4.    * @param connectionRetryPolicy 如果方法执行失败，指定一种重试策略，这个参数如果为
   null，那么会在Client.ConnectionId.getConnectionId()方法中创建
5.    * @param checkVersion 是否检查服务器端接口版本
6.    * @return 返回一个客户端的代理对象
7.    * */
8.   public static VersionedProtocol getProxy(
9.       Class<? extends VersionedProtocol> protocol,
10.      long clientVersion, InetSocketAddress addr, UserGroupInformation ticket,
11.      Configuration conf, SocketFactory factory, int rpcTimeout,
12.      RetryPolicy connectionRetryPolicy,
13.      boolean checkVersion) throws IOException {
```

```
14.
15.     if (UserGroupInformation.isSecurityEnabled()) {
16.       SaslRpcServer.init(conf);
17.     }
18.     final Invoker invoker = new Invoker(protocol, addr, ticket, conf, factory,
19.         rpcTimeout, connectionRetryPolicy);
20.     //构造一个实现了protocol接口的Java动态代理对象
21.     VersionedProtocol proxy = (VersionedProtocol)Proxy.newProxyInstance(
22.         protocol.getClassLoader(), new Class[]{protocol}, invoker);
23.
24.     if (checkVersion) {
25.       checkVersion(protocol, clientVersion, proxy);//调用getProtocolVersion()
26.     }
27.     return proxy;
28.   }
```

在该方法中，先创建一个Invoker对象(暂时不讨论安全相关的代码)，然后再调用Proxy.newProxyInstance()方法创建一个动态代理对象，而Invoker类则实现了InvocationHandler接口，这样对代理对象proxy的所有方法调用都会重定向到Invoker类的invoke()方法中。动态代理对象创建成功之后，就检查服务器接口的版本，在checkVersion()方法中会调用代理对象的getProtocolVersion()方法，所以这个getProtocolVersion()方法会重定向到Invoker类的invoke()方法中。具体的调用代码在以后分析客户端的时候再来分析。

EOF

Reference

《Hadoop技术内幕：深入解析Hadoop Common和HDFS架构设计与实现原理》