JGroups

作者: whitesock http://whitesock.javaeye.com

我的博客文章精选

目 录

1. SE

1.1 JGroups(1)

发表时间: 2008-06-01 关键字: jgroups

1 Overview

JGroups是一个用于建立可靠的组播通信的工具包（这里指的组播并不一定是IP Multicast，JGroups同样支持使用TCP作为传输协议）。其中可靠性是指通过适当的配置可以保证：消息在传输的过程中不会丢失；所有的接收者以相同的顺序接受所有的消息；原子性：一个消息要么被所有的接收者接收，要么不被任何一个接收者都接收。目前在JBoss Application Server Clustering，OSCache Clustering，Jetty HTTP session replication，Tomcat HTTP session replication中都使用了JGroups。

	Unreliable	Reliable
Unicast	UDP	TCP
Multicast	IP Multicast	**JGroups**

TCP和UDP是单播（Unicast）协议，也就是说：发送者和每一接收者之间是点对点传输。 如果一个发送者希望向多个接收者传输相同的数据，那么必须相应的复制多份数据。TCP是可靠的传输协议，但UDP不是可靠的，也就是说报文在传输的过程中可能丢失、重复或着乱序，报文的最大尺寸也有限制。IP Multicast可以将消息同时发送到多个接收者。由于IP Multicast是基于UDP的，因此IP Multicast是不可靠的。IP Multicast需要一个特殊的组播地址，它是一组D类IP地址，范围从224.0.0.0 到 239.255.255.255，其中有一部分地址是为特殊的目的保留的。JGroups使用UDP (IP Multicast)、TCP、JMS作为传输协议。JGroups最强大的功能之一是提供了灵活的，可定制的协议栈，以满足不同的需求。例如，如果选择使用IP Multicast作为传输协议，那么为了防止报文丢失和重复，可以在协议栈中添加NAKACK协议；为了保证报文的顺序，可以在协议栈中添加TOTAL协议，以保证FIFO的顺序；为了在组内成员发生变化时得到通知和回调，可以添加Group Membership Service (GMS) 和 FLUSH协议；Failure Detector (FD)协议用于识别组内崩溃的成员；如果新加入的成员希望获得组内其它成员维护的状态，那么可以向协议栈中添加STATE_TRANSFER协议；如果希望对传输的数据进行加密，那么可以使用CRYPT协议等等。

JGruops的主要功能有：

- 组的创建和删除。组可以跨越LANs或者WANs。
- 加入组、主动或者被动（例如当机或者网络故障）离开组。
- 在组成员改变时，组中其它成员可以得到通知。
- 向组中的单个或者多个成员发送消息。

在JGroups中JChannel类提供了主要的API ，用于连接到集群（cluster）、发送和接收消息（Message）和注册listeners等。Message包含消息头（保存地址等信息）和一个字节数组（保存希望传输的数据）。org.jgroups.Address接口及其实现类封装了地址信息，它通常包含IP地址和端口号。连接到集群中的所有实例（instance）被称为一个视图（org.jgroups.View）。通过View.getMembers()可以得到所有实例的地址。实

例只有在连接到集群后才能够发送和接收消息。以相同name调用JChannel.connect(String name)方法的所有实例会连接到同一个集群。当实例希望离开集群时，可以调用JChannel.disconnect()方法。当希望释放占有的资源时，可以调用JChannel.close()方法。JChannel.close()方法内部会调用JChannel.disconnect()方法。

通过调用JChannel.setReceiver()方法可以接收消息和得到View改变的通知。每当有实例加入或者离开集群的时候，viewAccepted(View view)方法会被调用。View.toString()方法会打印出View中所有实例的地址，以及View ID。需要注意的是，每次viewAccepted(View view)方法被调用时，view参数都不同，其View ID也会增长。View内的第一个实例被称为coordinator。Receiver接口上的getState()，setState()方法用于在实例间传递状态。新的实例通过setState()方法获得通过状态，而这个状态是通过调用集群中其它某个实例上的getState()获得的。

以下是JGruops manual中的一个简单的例子：

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.LinkedList;
import java.util.List;

import org.jgroups.JChannel;
import org.jgroups.Message;
import org.jgroups.ReceiverAdapter;
import org.jgroups.View;
import org.jgroups.util.Util;

public class SimpleChat {
        //
        private JChannel channel;
        private List<String> state = new LinkedList<String>();
        private String userName = System.getProperty("user.name", "WhiteSock");

        public void start() throws Exception {
                //
                channel = new JChannel();
                channel.setReceiver(new ReceiverAdapter() {

                        public void receive(Message msg) {
                                System.out.println(msg.getSrc() + ": " + msg.getObject());
```

```
                synchronized(state) {
                        state.add((String)msg.getObject());
                }
        }

        public void viewAccepted(View view) {
                System.out.println("view accepted: " + view);
        }

        public byte[] getState() {
                synchronized(state) {
                        try {
                                return Util.objectToByteBuffer(state);
                        }
                        catch(Exception e) {
                                e.printStackTrace();
                                return null;
                        }
                }
        }

        @SuppressWarnings("unchecked")
        public void setState(byte[] new_state) {
                try {
                        List<String> list=(List<String>)Util.objectFromByteBuf
                        synchronized(state) {
                                state.clear();
                                state.addAll(list);
                        }
                        System.out.println("received state (" + list.size() + '
                        for(String str: list) {
                                System.out.println(str);
                        }
                }
                catch(Exception e) {
                        e.printStackTrace();
                }
```

```
                }
        });
        channel.connect("ChatCluster");
        channel.getState(null, 10000);

        //
        sendMessage();

        //
        channel.close();
    }

    private void sendMessage() throws Exception {
        boolean succeed = false;
        BufferedReader br = null;
        try {
            br = new BufferedReader(new InputStreamReader(System.in));
            while(true) {
                System.out.print(">");
                System.out.flush();
                String line = br.readLine();
                if(line != null && line.equals("exit")) {
                    break;
                } else {
                    Message msg = new Message(null, null, "[" + userName +
                    channel.send(msg);
                }
            }
            succeed = true;
        } finally {
            if(br != null) {
                try {
                    br.close();
                } catch (Exception e) {
                    if(succeed) {
                        throw e;
                    }
```

```
                                }
                        }
                }
        }

        public static void main(String args[]) throws Exception {
                new SimpleChat().start();
        }
}
```

在以上例子中，主线程会阻塞，直到从stdin中读取一行。如果这行是"exit"，那么程序退出，否则向集群中发送一个消息。如果集群中某个实例强行退出，那么集群中的其它实例也会得到通知。Message构造函数的第一个参数如果是null，那么意味着消息将被发送到集群内所有的实例。

1.2 JGroups(2)

发表时间: 2008-06-02 关键字: jgroups

2 API

2．1 Interfaces

2．1．1 Transport

Transport接口只定义了最简单的方法，用于发送和接收消息。其定义如下：

```
public interface Transport {
    void send(Message msg) throws Exception;
    Object receive(long timeout) throws Exception;
}
```

2．1．2 MessageListener

如果说Transport接口是以pull的方式接收消息，那么MessageListener则是以push的方式处理消息。当收到消息时，receive方法会被调用。getState() 和setState()方法用于在实例间传递状态。其定义如下：

```
public interface MessageListener {
    void receive(Message msg);
    byte[] getState();
    void setState(byte[] state);
}
```

2．1．3 ExtendedMessageListener

ExtendedMessageListener继承自MessageListener，它定义了用来在实例间部分传递状态的方法。如果需要传递的状态数据量很大，那么通过配置协议栈，也可以指定使用流的方式传递状态。其定义如下：

```
public interface ExtendedMessageListener extends MessageListener {
        byte[] getState(String state_id);
        void setState(String state_id, byte[] state);

        void getState(OutputStream ostream);
        void setState(InputStream istream);

        void getState(String state_id, OutputStream ostream);
        void setState(String state_id, InputStream istream);
}
```

2．1．4 MembershipListener

当收到view、suspicion message和block event 的时候，相应的方法会被调用。这个接口常用的方法是viewAccepted()，以便在新的实例加入（或者离开）到集群时得到通知。当JGroups推测某个实例可能崩溃时（此时该实例并未离开集群），suspect()方法会被调用，目前没有unsuspect()方法。当JGroups需要通知集群中的实例不要发送消息时，block()方法会被调用。这通常需要配置FLUSH协议，例如为了确保在进行状态传递的时候，没有实例在发送消息。在block()方法返回后，所有发送消息的线程都会被阻塞，知道FLUSH协议解除阻塞。需要注意的是，block()方法内不应该执行耗时的操作，否则整个FLUSH协议都会被阻塞。其定义如下：

```
public interface MembershipListener {
    void viewAccepted(View new_view);
    void suspect(Address suspected_mbr);
    void block();
}
```

2．1．5 ExtendedMembershipListener

ExtendedMembershipListener继承自MembershipListener。当FLUSH协议解除阻塞的时候，unblock()方法会被调用，所有发送消息的线程可以继续发送消息。其定义如下：

```
public interface ExtendedMembershipListener extends MembershipListener {
    void unblock();
}
```

2．1．6 ChannelListener

可以通过调用JChannel接口的addChannelListener(ChannelListener listener)方法来添加ChannelListener。当Channel被连接或者关闭时，相应的方法会北调用。其定义如下：

```
public interface ChannelListener {
    void channelConnected(Channel channel);
    void channelDisconnected(Channel channel);
    void channelClosed(Channel channel);
    void channelShunned();
    void channelReconnected(Address addr);
}
```

2．1．7 Receiver

Receiver继承自MessageListener和MembershipListener。其定义如下：

```
public interface Receiver extends MessageListener, MembershipListener {
}
```

2．1．8 ExtendedReceiver

ExtendedReceiver继承自Receiver、ExtendedMessageListener和ExtendedMembershipListener。其定义如下：

```
public interface ExtendedReceiver extends Receiver, ExtendedMessageListener, ExtendedMembershi
}
```

2．2 Channel

2．2．1 Creating a channel

最常见的创建Channel的方法是通过构造函数，此外也可以通过工厂方法。需要注意的是，集群中所有的实例必须有相同的协议栈。JChannel的构造函数之一如下：

```
public JChannel(String properties) throws ChannelException {
    this(ConfiguratorFactory.getStackConfigurator(properties));
}
```

以上的构造函数中，properties参数是冒号分割的字符串，用来配置协议栈。字符串的最左端的元素定义了最底层的协议。如果properties为null，那么将使用缺省的协议栈，即jgroups-all.jar中的udp.xml。以下是个properties参数的例子：

```
String props="UDP(mcast_addr=228.1.2.3;mcast_port=45566;ip_ttl=32):" +
"PING(timeout=3000;num_initial_members=6):" +
"FD(timeout=5000):" +
"VERIFY_SUSPECT(timeout=1500):" +
"pbcast.NAKACK(gc_lag=10;retransmit_timeout=3000):" +
"UNICAST(timeout=300,600,1200):" +
"FRAG:" +
"pbcast.GMS(join_timeout=5000;shun=false;print_local_addr=true)";
```

此外，也可以用File和URL作为构造函数的参数，这种方式允许以本地或者远程的XML文件配置协议栈。XML文件的config节点中的每个子节点定义一个协议，第一个子节点定义了最底层的协议。每个子节点名都对应一个Java类名，缺省的协议名不必是全限定类名，它们位于org.jgroups.stack.protocols包中。如果是自定

义的协议，那么则必须是全限定类名。每个协议可以有零个或多个属性，以name/value对的方式指定。以下是jgroups-all.jar中的udp.xml的内容：

```
<config>
    <UDP
         mcast_addr="${jgroups.udp.mcast_addr:228.10.10.10}"
         mcast_port="${jgroups.udp.mcast_port:45588}"
         tos="8"
         ucast_recv_buf_size="20000000"
         ucast_send_buf_size="640000"
         mcast_recv_buf_size="25000000"
         mcast_send_buf_size="640000"
         loopback="false"
         discard_incompatible_packets="true"
         max_bundle_size="64000"
         max_bundle_timeout="30"
         use_incoming_packet_handler="true"
         ip_ttl="${jgroups.udp.ip_ttl:2}"
         enable_bundling="true"
         enable_diagnostics="true"
         thread_naming_pattern="cl"

         use_concurrent_stack="true"

         thread_pool.enabled="true"
         thread_pool.min_threads="2"
         thread_pool.max_threads="8"
         thread_pool.keep_alive_time="5000"
         thread_pool.queue_enabled="true"
         thread_pool.queue_max_size="1000"
         thread_pool.rejection_policy="Run"

         oob_thread_pool.enabled="true"
         oob_thread_pool.min_threads="1"
         oob_thread_pool.max_threads="8"
         oob_thread_pool.keep_alive_time="5000"
         oob_thread_pool.queue_enabled="false"
```

```
        oob_thread_pool.queue_max_size="100"
        oob_thread_pool.rejection_policy="Run"/>

    <PING timeout="2000"
            num_initial_members="3"/>
    <MERGE2 max_interval="30000"
            min_interval="10000"/>
    <FD_SOCK/>
    <FD timeout="10000" max_tries="5"   shun="true"/>
    <VERIFY_SUSPECT timeout="1500"  />
    <BARRIER />
    <pbcast.NAKACK use_stats_for_retransmission="false"
                   exponential_backoff="150"
                   use_mcast_xmit="true" gc_lag="0"
                   retransmit_timeout="50,300,600,1200"
                   discard_delivered_msgs="true"/>
    <UNICAST timeout="300,600,1200"/>
    <pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
                   max_bytes="1000000"/>
    <VIEW_SYNC avg_send_interval="60000"   />
    <pbcast.GMS print_local_addr="true" join_timeout="3000"
                shun="false"
                view_bundling="true"/>
    <FC max_credits="500000"
                    min_threshold="0.20"/>
    <FRAG2 frag_size="60000"  />
    <!--pbcast.STREAMING_STATE_TRANSFER /-->
    <pbcast.STATE_TRANSFER  />
    <!-- pbcast.FLUSH  /-->
</config>
```

以上XML文件中，UDP协议的mcast_addr属性被指定使用jgroups.udp.mcast_addr系统属性，如果没有配置这个系统属性，那么使用缺省值228.10.10.10。

2．2．2 Setting options

通过setOpt(int option, Object value)方法可以给Channel设置属性，目前支持的属性有：

- Channel.BLOCK 这是一个Boolean型的属性，缺省是false。如果设置成true，那么会接收到block message。
- Channel.LOCAL这是一个Boolean型的属性，缺省设置成true。如果是true，那么当集群中的实例向集群发送消息时，这个实例本身也会收到这个消息。
- Channel.AUTO_RECONNECT这是一个Boolean型的属性，缺省是false。如果设置成true，那么shunned channel 在离开集群后，会自动重新加入到集群中。
- Channel.AUTO_GETSTATE 这是一个Boolean型的属性，缺省是false。如果设置成true，那么shunned channel在自动重新加入到集群后，会自动尝试向集群的coordinator 获得集群的状态（需要AUTO_RECONNECT也设置成true）。

通过Object getOpt(int option)可以或者channel的相关属性值。

2．2．3 Connecting/Disconnecting

通过调用connect(String cluster_name) throws ChannelException方法连接到集群。cluster_name参数指定了集群的名称。集群中的第一个实例被称为coordinator。当集群中的成员发生变化的时候，coordinator会向集群中的其它实例发送view message。

也可以通过调用void connect(String cluster_name, Address target, String state_id,long timeout) throws ChannelException方法连接到集群，并从集群中请求当前的状态。与将这两个操作分开进行相比，在一个方法调用内完成这两个操作，可以允许JGroups对发送的消息进行优化，更重要的是，从调用者角度来看，这两个操作被合并成一个原子操作。cluster_name参数用于指定集群的名称。target参数指定了从集群中的哪个实例获得状态，如果该参数为null，那么会从coordinator获得。如果希望传递部分的状态，那么state_id参数可以用于指定状态id。

当Channel连接到集群后，通过调用String getClusterName()方法可以获得当前连接到的集群名称。通过调用Address getLocalAddress()方法可以获得channel的地址。通过调用View getView()方法可以获得channel的当前view。每当Channel收到view message的时候，channel的当前view就会被更新。

通过调用void disconnect()方法以断开到集群的连接。如果channel已经并没有连接到集群，或者chaneel已经被close，那么调用这个方法没有任何效果。如果channel已经连接到集群，那么这个方法内会向coordinator发送一个离开请求，同时coordinator会向集群中的所有其它实例发送view message，以通知它们该实例的离开。断开连接的channel可以通过调用connect()方法重新连接到集群。

通过void close()方法以释放channel占有的资源。Channel被close之后，调用channel上的任何方法都可能会导致异常。

2．2．4 Sending messages

当channel连接到集群后，可以通过以下方法发送消息。第一个send方法接受一个Message型的参数，如果msg的目标地址不是null，那么消息会发送到指定地址，否则会发送到集群内的所有成员。msg的源地址可以不必手工设置，如果是null，那么会被协议栈的最底层协议（传输协议）设置成channel的本地地址。第二个send方法在内部使用了一个send方法。

```
void send(Message msg) throws ChannelNotConnectedException, ChannelClosedException
void send(Address dst, Address src, Serializable obj) throws ChannelNotConnectedException, Char
```

以下是个发送消息的例子：

```
Hashtable data;
try {
    Address receiver=channel.getView().getMembers().first();
    channel.send(receiver, null, data);
}
catch(Exception ex) {
    // handle errors
}
```

2．2．5 Receiving messages

Channel 以异步的方式从网络上接收消息，然后把消息存放在一个无界队列中。当调用receive()方法时，会尝试返回队列中取得下一个可用的消息。如果队列中没有消息，那么会被阻塞。如果timeout小于等于零，那么会永远等待下去；否则会等待timeout指定的毫秒数，直到收到消息或者抛出TimeoutException。需要注意的是，JChannel. receive(long timeout)方法已经被标记为deprecated。根据channel options的不同，receive()方法可能返回以下类型的对象：

- Message 普通消息。Message.makeReply()可以同于创建消息的应答，即以当前消息的源地址作为应答消息的目标地址。
- View 当集群的成员发生变化的时候，集群的每个成员都会收到view message。当两个或者多个子集群（subgroups）合并成一个的时候，集群中的成员会收到MergeView message。如果需要在子集群合并时处理子集群状态的合并，那么通常需要在单独的线程里执行耗时的操作。
- SuspectEvent 当集群中的某个成员被怀疑崩溃时，集群中的其它成员会收到SuspectEvent message。通过调用SuspectEvent.getMember()可以得到可疑成员的地址。在收到这个消息后，通常还会收到view message。
- BlockEvent 当收到BlockEvent message后，实例应该停止发送消息，然后应该调用Channel.blockOk()方法（目前JChannel.blockOk()方法是一个空方法）确认已经停止发送消息。当Channel.blockOk()方法调用完毕之后，所有发送消息的线程都会被阻塞直到FLUSH协议解除阻塞。为了接收BlockEvent message，需要设置Channel.BLOCK属性为true。
- UnblockEvent 当收到UnblockEvent message后，实例可以继续发送消息。
- GetStateEvent 当收到GetStateEvent message后，实例应该保存当前的状态，并将当前状态的一份拷贝作为参数调用Channel.returnState()方法，然后JGroups会将这个状态返回给请求状态的实例。为了接收GetStateEvent message，需要在协议栈中配置pbcast.STATE_TRANSFER协议。

- StreamingGetStateEvent当收到StreamingGetStateEvent message后，实例应该通过StreamingGetStateEvent.getArg()返回的输出流返回状态。为了接收StreamingGetStateEvent message，需要在协议栈中配置pbcast.STREAMING_STATE_TRANSFER协议。
- SetStateEvent当收到SetStateEvent message后，实例应该通过SetStateEvent.getArg()返回的字节数组取得状态。
- StreamingSetStateEvent当收到StreamingSetStateEvent message后，实例应该通过StreamingSetStateEvent.getArg()返回的输入流取得状态。为了接收StreamingSetStateEvent message，需要在协议栈中配置pbcast.STREAMING_STATE_TRANSFER协议。

以下是个使用pull方式接收消息的例子：

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

import org.jgroups.BlockEvent;
import org.jgroups.Channel;
import org.jgroups.GetStateEvent;
import org.jgroups.JChannel;
import org.jgroups.Message;
import org.jgroups.SetStateEvent;
import org.jgroups.UnblockEvent;
import org.jgroups.View;
import org.jgroups.util.Util;

public class PollStyleReceiver implements Runnable {
        //
        private JChannel channel;
        private Map<String, String> state = new HashMap<String, String>();
        private String properties = "UDP(mcast_addr=228.1.2.3;mcast_port=45566;ip_ttl=32):" +
        "PING(timeout=3000;num_initial_members=6):" +
        "FD(timeout=5000):" +
        "VERIFY_SUSPECT(timeout=1500):" +
        "pbcast.NAKACK(gc_lag=10;retransmit_timeout=3000):" +
        "UNICAST(timeout=300,600,1200):" +
        "FRAG:" +
```

```
        "pbcast.GMS(join_timeout=5000;shun=false;print_local_addr=true):" +
        "pbcast.STATE_TRANSFER:" +
        "pbcast.FLUSH";


    public void start() throws Exception {
            //
            channel = new JChannel(properties);
            channel.connect("PollStyleReceiver");
            channel.setOpt(Channel.BLOCK, Boolean.TRUE);
            channel.getState(null, 10000);

            new Thread(this).start();

            sendMessage();

            channel.close();
    }

    @SuppressWarnings({ "unchecked", "deprecation" })
    public void run() {
            while(true) {
                    try {
                            Object obj = channel.receive(0);
                            if(obj instanceof Message) {
                                    System.out.println("received a regular message: " + (Me
                                    String s = (String)((Message)obj).getObject();
                                    String key = s.substring(0, s.indexOf("="));
                                    String value = s.substring(s.indexOf("=") + 1);
                                    state.put(key, value);
                            } else if(obj instanceof View) {
                                    System.out.println("received a View message: " + (View)
                            } else if(obj instanceof BlockEvent) {
                                    System.out.println("received a BlockEvent message: " +
                                    channel.blockOk();
                            } else if(obj instanceof UnblockEvent) {
                                    System.out.println("received a UnblockEvent message: "
```

```
                        } else if(obj instanceof GetStateEvent) {
                                System.out.println("received a GetStateEvent message: '
                                channel.returnState(Util.objectToByteBuffer(copyState(s
                        } else if(obj instanceof SetStateEvent) {
                                System.out.println("received a SetStateEvent message: '
                                this.state = (Map<String, String>)Util.objectFromByteBu
                                System.out.println("current state: " + printState(this.
                        } else {
                                System.out.println(obj);
                        }
                } catch(Exception e) {
                        e.printStackTrace();
                        break;
                }
        }
}

private void sendMessage() throws Exception {
        boolean succeed = false;
        BufferedReader br = null;
        try {
                br = new BufferedReader(new InputStreamReader(System.in));
                while(true) {
                        System.out.print("> ");
                        System.out.flush();
                        String line = br.readLine();
                        if(line != null && line.equals("exit")) {
                                break;
                        } else if(line.indexOf("=") > 0 || line.indexOf("=") == line.le
                                Message msg = new Message(null, null, line);
                                channel.send(msg);
                        } else {
                                System.out.println("invalid input: " + line);
                        }
                }
                succeed = true;
        } finally {
```

```
                if(br != null) {
                        try {
                                br.close();
                        } catch (Exception e) {
                                if(succeed) {
                                        throw e;
                                }
                        }
                }
        }
}

private Map<String, String> copyState(Map<String, String> s) {
        Map<String, String> m = new HashMap<String, String>();
        for(String key : s.keySet()) {
                m.put(key, s.get(key));
        }
        return m;
}

private String printState(Map<String, String> s) {
        StringBuffer sb = new StringBuffer();
        sb.append("[");
        for(Iterator<String> iter = s.keySet().iterator(); iter.hasNext(); ) {
                String key = iter.next();
                sb.append(key).append("=");
                sb.append(s.get(key));
                if(iter.hasNext()) {
                        sb.append(", ");
                }
        }
        sb.append("]");
        return sb.toString();
}

public static void main(String args[]) throws Exception {
        new PollStyleReceiver().start();
```

```
        }
}
```

程序启动后，程序会将在命令行键入键值对（例如key1=value1）保存到HashMap中。并允许在不同的实例间传递状态。

除了以poll的方式接收消息外，JGroups也支持以push的方式处理消息。通过向JChannel注册Receiver，允许程序以回调的方式处理消息，而不必启动额外的线程来接收消息，同时JGroups在内部也不用使用无界队列来保存消息。一下是个使用push方式处理消息的例子：

```
JChannel ch = new JChannel();
ch.setReceiver(new ExtendedReceiverAdapter() {
    public void receive(Message msg) {
        System.out.println("received message " + msg);
    }
    public void viewAccepted(View new_view) {
        System.out.println("received view " + new_view);
    }
});
ch.connect("bla");
```

2．2．6 State transfer

JGroups支持在集群中维护和传递状态(state)，例如web server的Http Sessions等。集群中的某个实例可以通过JChannel.send()方法发送消息，从而把对状态的修改同步到集群的其它实例中。当一个新的实例加入到集群后，可以调用JChannel.getState()方法向集群中的某个实例（缺省是coordinator）请求获得当前的状态。需要注意的是，JChannel.getState()方法返回的是boolean类型。如果该实例是集群中的第一个实例，那么该方法返回false（也就是说目前没有状态），否则返回true。在接下来JChannel.receive()方法的返回值中会包含SetStateEvent message，或者通过MembershipListener.setState()方法获得状态。JChannel.getState()方法不直接返回状态的原因是，如果JChannel的消息队列中还有未被处理的消息，那么让JChannel.getState()直接返回状态，会破坏消息接收的FIFO顺序保证，传递的状态也会不正确。

假设某个集群中包含A、B 和C三个成员，当D加入到集群的时候，如果D调用了JChannel.getState()，那么会发生以下的调用序列：

1. D 调用 JChannel.getState()方法。假设状态从集群中的A成员取得。
2. A 收到GetStateEvent message或者A注册的Receiver的getState() 方法被调用。A返回了当前状态。
3. D 调用 JChannel.getState()方法返回，返回值是true。
4. D 收到SetStateEvent message或者D注册的Receiver的setState()方法被调用。D取得状态。

2．2．5节的例子中包含了状态传递相关的代码，需要注意的是，在调用JChannel.returnState()方法的时候，为了防止在状态被通过网络发送前，程序的其它地方对状态进行了修改（比如接收到新的消息并更新状态），需要传递当前状态的一份拷贝。

除了通过处理GetStateEvent 和 SetStateEvent消息来传递状态之外，JGroups也支持通过Reciever传递状态，例如在第一章中演示的例子。

JGroups支持传递部分状态和以流的形式传递状态，详细内容可以参考JGroups Manual。

1.3 JGroups(3)

发表时间: 2008-06-03 关键字: jgroups

3 Building Blocks

Building blocks位于org.jgroups.blocks包中，在逻辑上可以视为channels之上的一层，它提供了更复杂的接口。Building blocks并不必依赖于channels，部分building blocks只需要实现了Transport接口的类即可工作。以下简要介绍部分building blocks。

3．1 MessageDispatcher

Channels 通常用于异步地发送和接收消息。然后有些情况下需要同步通信，例如发送者希望向集群发送消息并等待所有成员的应答，或者等待部分成员的应答。MessageDispatcher支持以同步或者异步的方式发送消息，它在构造时需要一个Channel型的参数。

MessageDispatcher提供了Object handle(Message msg)方法，用于以push 方式的接收消息并返回应答（必须可以被序列化），该方法抛出的异常也会被传播到消息发送者。MessageDispatcher在内部使用了PullPushAdapter，PullPushAdapter也是org.jgroups.blocks包中的类，但是已经被标记为deprecated。这种方式被称为MessageDispatcher的server模式。

MessageDispatcher的client模式是指通过调用castMessage或者sendMessage向集群发送消息并同步或者异步的等待应答。castMessage()方法向dests指定的地址发送消息，如果dest为null，那么向集群中所有成员发送消息。castMessage()方法的返回值是RspList，RspList 实现了Map<Address,Rsp> 接口。msg参数中的目的地址会被覆盖。mode参数（由org.jgroups.blocks.GroupRequest类定义）指定了消息是同步还是异步发送，其可选值如下：

- GET_FIRST 返回收到的第一个应答。
- GET_ALL 等待所有成员的应答（被怀疑崩溃的成员除外）。
- GET_MAJORITY 等待绝大多数成员（相对与成员的个数）的应答。
- GET_ABS_MAJORITY等待绝大多数成员（一个绝对的数值，只计算一次）的应答。
- GET_N 等待n个应答，如果n大于成员的个数，可能会一直阻塞下去。
- GET_NONE 不等待应答，直接返回，即异步方式。

castMessage()方法的定义如下：

```
public RspList castMessage(Vector dests, Message msg, int mode, long timeout);
```

sendMessage()方法允许向一个成员发送消息，msg参数的目的地址不能为null。如果mode参数是GET_NONE，那么消息的发送变成异步方式；否则mode参数会被忽略（缺省采用GET_FIRST）。sendMessage()方法的定义如下：

```
public Object sendMessage(Message msg, int mode, long timeout) throws TimeoutException;
```

以下是个使用MessageDispatcher的例子：

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

import org.jgroups.Channel;
import org.jgroups.JChannel;
import org.jgroups.Message;
import org.jgroups.blocks.GroupRequest;
import org.jgroups.blocks.MessageDispatcher;
import org.jgroups.blocks.RequestHandler;
import org.jgroups.util.RspList;

public class MessageDispatcherTest {
        //
        private Channel channel;
        private MessageDispatcher dispatcher;
        private boolean propagateException = false;

        public void start() throws Exception {
                //
                channel = new JChannel();
                dispatcher = new MessageDispatcher(channel, null, null, new RequestHandler() {

                        public Object handle(Message msg) {
                                System.out.println("got a message: " + msg);
                                if(propagateException) {
                                        throw new RuntimeException("failed to handle message: '
                                } else {
                                        return new String("success");
                                }
                        }

                });
                channel.connect("MessageDispatcherTest");

                //
```

```
            sendMessage();

            //
            channel.close();
            dispatcher.stop();
    }

    private void sendMessage() throws Exception {
            boolean succeed = false;
            BufferedReader br = null;
            try {
                    br = new BufferedReader(new InputStreamReader(System.in));
                    while(true) {
                            System.out.print("> ");
                            System.out.flush();
                            String line = br.readLine();
                            if(line != null && line.equals("exit")) {
                                    break;
                            } else {
                                    Message msg = new Message(null, null, line);
                                    RspList rl = dispatcher.castMessage(null, msg, GroupRec
                                    System.out.println("Responses:\n" + rl);
                            }
                    }
                    succeed = true;
            } finally {
                    if(br != null) {
                            try {
                                    br.close();
                            } catch (Exception e) {
                                    if(succeed) {
                                            throw e;
                                    }
                            }
                    }
            }
    }
```

```
        public static void main(String[] args) {
                try {
                        new MessageDispatcherTest().start();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

3．2 RpcDispatcher

RpcDispatcher 继承自MessageDispatcher，它允许远程调用集群中其它成员上的方法，并可选地等待应答。跟MessageDispatcher相比，不需要为RpcDispatcher指定RequestHandler。RpcDispatcher的构造函数接受一个Object server_obj参数，它是远程调用的目标对象。RpcDispatcher的callRemoteMethods系列方法用于远程调用目标对象上的方法，该方法可以由MethodCall指定，也可以通过方法名、参数类型指定。跟MessageDispatcher的castMessage()方法和sendMessage()方法类似，callRemoteMethods系列方法也接受一个int mode参数，其含义也相同。以下是个简单的例子：

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

import org.jgroups.Channel;
import org.jgroups.JChannel;
import org.jgroups.blocks.GroupRequest;
import org.jgroups.blocks.RpcDispatcher;
import org.jgroups.util.RspList;

public class RpcDispatcherTest {
        private Channel channel;
        private RpcDispatcher dispatcher;

        public int print(int number) throws Exception {
                return number * 2;
        }

        public void start() throws Exception {
                channel = new JChannel();
```

```
        dispatcher = new RpcDispatcher(channel, null, null, this);
        channel.connect("RpcDispatcherTest");

        //
        sendMessage();

        //
        channel.close();
        dispatcher.stop();
    }

    private void sendMessage() throws Exception {
        boolean succeed = false;
        BufferedReader br = null;
        try {
            br = new BufferedReader(new InputStreamReader(System.in));
            while(true) {
                System.out.print("> please input an int value:");
                System.out.flush();
                String line = br.readLine();
                if(line != null && line.equals("exit")) {
                    break;
                } else {
                    int param = 0;
                    try {
                        param = Integer.parseInt(line);
                    } catch(Exception e) {
                        System.out.println("invalid input: " + line);
                        continue;
                    }
                    RspList rl = dispatcher.callRemoteMethods(null, "print'
                    System.out.println("Responses: \n" + rl);
                }
            }
            succeed = true;
        } finally {
            if(br != null) {
```

```
                                try {
                                        br.close();
                                } catch (Exception e) {
                                        if(succeed) {
                                                throw e;
                                        }
                                }
                        }
                }
        }

        public static void main(String[] args) {
                try {
                        new RpcDispatcherTest().start();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

3．3 ReplicatedHashMap

ReplicatedHashMap 继承自ConcurrentHashMap，并在内部使用了RpcDispatcher。ReplicatedHashMap构造函数的clustername参数指定了集群的名字，集群中所有的实例会包含相同的状态。新加入的实例在开始工作前会从集群中获得当前的状态。对实例的修改（例如通过put，remove方法）会传播到集群的其它实例中，只读的请求（例如get方法）则是本地调用。需要注意的是，ReplicatedHashMap的以下划线开头的方法是用于RpcDispatcher的远程调用的。在ReplicatedHashMap上可以注册 Notification，以便在实例的状态改变时进行回调，所有的回调也是本地的。以下是个简单的例子：

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Iterator;
import java.util.Map;
import java.util.Vector;

import org.jgroups.Address;
import org.jgroups.ChannelFactory;
import org.jgroups.JChannelFactory;
```

```
import org.jgroups.View;
import org.jgroups.blocks.ReplicatedHashMap;

public class ReplicatedHashMapTest implements ReplicatedHashMap.Notification<String, String> {
        //
        private ReplicatedHashMap<String, String> map;

        public void start() throws Exception {
                ChannelFactory factory = new JChannelFactory();
                map = new ReplicatedHashMap<String, String>("ReplicatedHashMapTest", factory, '
                map.addNotifier(this);

                sendMessage();
                map.stop();
        }

        public void entryRemoved(String key) {
                System.out.println("in entryRemoved(" + key + ")");
        }

        public void entrySet(String key, String value) {
                System.out.println("in entrySet(" + key + "," + value + ")");
        }

        public void contentsSet(Map<String, String> m) {
                System.out.println("in contentsSet(" + printMap(m) + ")");
        }

        public void contentsCleared() {
                System.out.println("in contentsCleared()");
        }

        public void viewChange(View view, Vector<Address> newMembers,
                        Vector<Address> oldMembers) {
                System.out.println("in viewChange(" + view + ")");
        }
```

```
    private void sendMessage() throws Exception {
            boolean succeed = false;
            BufferedReader br = null;
            try {
                    br = new BufferedReader(new InputStreamReader(System.in));
                    while (true) {
                            System.out.print("> ");
                            System.out.flush();
                            String line = br.readLine();
                            if (line != null && line.equals("exit")) {
                                    break;
                            } else {
                                    if (line.equals("show")) {
                                            System.out.println(printMap(map));
                                    } else if (line.equals("clear")) {
                                            map.clear();
                                    } else if (line.startsWith("remove ")) {
                                            String key = line.substring(line.indexOf(" ") +
                                            map.remove(key);
                                    } else if (line.startsWith("put ")) {
                                            line = line.replace("put ", "");
                                            int index = line.indexOf("=");
                                            if (index <= 0 || index >= (line.length() - 1))
                                                    System.out.println("invalid input");
                                                    continue;
                                            }
                                            String key = line.substring(0, index).trim();
                                            String value = line.substring(index + 1, line.l
                                                            .trim();
                                            map.put(key, value);
                                    } else {
                                            System.out.println("invalid input: " + line);
                                            continue;
                                    }
                            }
                    }
                    succeed = true;
```

```
                } finally {
                        if (br != null) {
                                try {
                                        br.close();
                                } catch (Exception e) {
                                        if (succeed) {
                                                throw e;
                                        }
                                }
                        }
                }
        }

        private String printMap(Map<String, String> m) {
                StringBuffer sb = new StringBuffer();
                sb.append("[");
                for (Iterator<String> iter = map.keySet().iterator(); iter.hasNext();) {
                        String key = iter.next();
                        String value = map.get(key);
                        sb.append(key).append("=").append(value);
                        if (iter.hasNext()) {
                                sb.append(",");
                        }
                }
                sb.append("]");
                return sb.toString();
        }

        public static void main(String args[]) {
                try {
                        new ReplicatedHashMapTest().start();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

3．4 NotificationBus

NotificationBus 提供了向集群发送通知的能力，通知可以是任何可以被序列化的对象。NotificationBus在内部使用Channel，其start()和stop()方法用于启动和停止。NotificationBus的setConsumer()方法用于注册Consumer接口，其定义如下：

```
public interface Consumer {
    void handleNotification(Serializable n);
    Serializable getCache();
    void memberJoined(Address mbr);
    void memberLeft(Address mbr);
}
```

NotificationBus的getCacheFromCoordinator() 和getCacheFromMember()用于请求集群的状态。前者是从coordinator得到状态，后者从指定地址的成员处得到状态。NotificationBus上注册的Consumer需要实现getCache()方法以返回状态。以下是个简单的例子：

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.Serializable;
import java.util.Iterator;
import java.util.LinkedList;

import org.jgroups.Address;
import org.jgroups.blocks.NotificationBus;

public class NotificationBusTest implements NotificationBus.Consumer {
        //
        private NotificationBus bus;
        private LinkedList<Serializable> cache;

        public void handleNotification(Serializable n) {
                System.out.println("in handleNotification(" + n + ")");
                if (cache != null) {
                        cache.add(n);
                }
        }
```

```
	public Serializable getCache() {
		return cache;
	}

	public void memberJoined(Address mbr) {
		System.out.println("in memberJoined(" + mbr + ")");
	}

	public void memberLeft(Address mbr) {
		System.out.println("in memberLeft(" + mbr + ")");
	}

	@SuppressWarnings("unchecked")
	public void start() throws Exception {
		//
		bus = new NotificationBus("NotificationBusTest", null);
		bus.setConsumer(this);
		bus.start();
		cache = (LinkedList<Serializable>) bus.getCacheFromCoordinator(3000, 1);
		if (cache == null) {
			cache = new LinkedList<Serializable>();
		}
		System.out.println(printCache(cache));

		//
		sendNotification();

		//
		bus.stop();
	}

	private void sendNotification() throws Exception {
		boolean succeed = false;
		BufferedReader br = null;
		try {
			br = new BufferedReader(new InputStreamReader(System.in));
			while (true) {
```

```
                        System.out.print("> ");
                        System.out.flush();
                        String line = br.readLine();
                        if (line != null && line.equals("exit")) {
                                break;
                        } else {
                                bus.sendNotification(line);
                        }
                }
                succeed = true;
        } finally {
                if (br != null) {
                        try {
                                br.close();
                        } catch (Exception e) {
                                if (succeed) {
                                        throw e;
                                }
                        }
                }
        }
}

private String printCache(LinkedList<Serializable> c) {
        StringBuffer sb = new StringBuffer();
        sb.append("[");
        for (Iterator<Serializable> iter = c.iterator(); iter.hasNext();) {
                sb.append(iter.next());
                if (iter.hasNext()) {
                        sb.append(",");
                }
        }
        sb.append("]");
        return sb.toString();
}

public static void main(String[] args) {
```

```
                try {
                        new NotificationBusTest().start();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

1.4 JGroups(4)

发表时间: 2008-06-04 关键字: jgroups

4 Protocol Stack

4．1 Transport protocols

Transport protocols是指协议栈中最底层的协议，它们负责发送和接收消息。JGgroups提供了以下几种transport protocols。

4．1．1 UDP

JGroups中的UDP协议使用IP multicast向集群发送消息，使用UDP datagram向单个的成员发送unicast消息。启动后会打开两个socket，分别是multicast socket和unicast socket。Channel的地址是unicast socket的地址和端口号。UDP通常用于集群中的成员分布于LAN内的情况。

如果使用UDP和PING做为协议栈的底层协议，那么JGroups会使用IP multicast发现集群中的成员，以及向集群发送发送消息。然而，如果IP multicast在子网间被禁用，那么可以设置UDP的ip_mcast属性为false，以便指定UDP使用多个unicast messages向集群发送消息，而不是使用multicast message。此外，还需要设置PING的gossip_系列属性，以便指定PING使用GossipRouter来发现集群中的其它成员。需要注意的是，对GossipRouter的依赖可能会导致single point of failure，而且系统的可伸缩性也比较差。

在启动任何成员之前，首先要启动GossipRouter（否则成员需要处理MergeView消息用于合并subgroup的状态），例如：

```
java org.jgroups.stack.GossipRouter -port 5555 -bindaddress localhost
```

UDP和PING的配置如下：

```
<UDP ip_mcast="false" />
<PING gossip_host="localhost" gossip_port="5555" gossip_refresh="15000" timeout="2000" num_init
```

4．1．2 TCP

当集群中的成员分布于WAN时（路由器会丢弃IP multicast报文），TCP可能是唯一可用的传输协议。当使用TCP作为传输协议是，可用的发现协议有：

- PING with GossipRouter: 跟4.1.1中介绍的一样，p_mcast属性必须设置成false，GossipRouter 也必须先于集群中的成员启动。
- TCPPING: 从特定已知的成员处得到集群中其它成员的信息。
- TCPGOSSIP: 除了允许多个GossipRouters 之外，TCPGOSSIP 跟PING相同。

以下是个使用TCP和TCPPING的例子：

```
<TCP start_port="7800" /> +
<TCPPING initial_hosts="HostA[7800],HostB[7800]" port_range="5" timeout="3000" num_initial_memb
```

使用TCPPING的优点是不需要额外GossipRouters，而是从集群的成员中选择那些已知的成员，例如以上例子中的HostA[7800]和 HostB[7800]，并从这些成员处得到其它成员的信息。TCP协议的start_port="7800"属性指定了选择7800作为端口号，如果该端口号被占用，那么尝试下一个（7801）端口号，直到找到可用的端口号。TCPPING协议会尝试连接HostA和HostB，连接的端口号的范围是从 7800到7800 + port_range -1（在以上例子中是7804）。

以下是个使用TCP和TCPGOSSIP的例子：

```
<TCP />
<TCPGOSSIP initial_hosts="localhost[5555],localhost[5556]" gossip_refresh_rate="10000" num_init
```

以上例子中，initial_hosts 属性用于指定GossipRouter的地址和端口号。GossipRouter需要先于集群中的成员启动。

4．2 Reliable Message

4．2．1 pbcast.NAKACK

NAKACK协议保证了向集群的所有成员发送的消息的传输可靠性，以及消息的FIFO顺序。消息传输的可靠性是指发送的消息不会丢失。此外发送者将发送的消息编号，如果接收者没有收到特定编号的消息，那么发送者会收到重新发送的请求。FIFO顺序是指接收者会以消息发送的顺序接收消息。以下是部分 NAKACK协议的属性：

- retransmit_timeout 逗号分割的一系列毫秒数。例如100,200,400,800,1600。在第一次发送重传输请求前等待100ms，第二次发送重传输请求前等待 200ms，依此类推直到等待1600ms。从这以后，每次发送重传输请求前等待100ms。
- use_mcast_xmit 当某个成员接收到P成员发送的对于消息M的重传输请求，该成员会向P重新发送消息M。考虑到集群中的其它成员也可能没有收到消息M，如果 use_mcast_xmit设置为true，那么该成员会向整个集群重新发送消息M。如果使用UDP作为传输协议，那么JGroups使用IP Multicast；如果使用TCP作为传输协议，那么会发送n-1个unicast消息（n是集群中消息的个数）。
- use_mcast_xmit_req 跟use_mcast_xmit属性类似，不同之处在于对重传输的请求消息进行组播发送。
- xmit_from_random_member 如果xmit_from_random_member设置为true，那么JGroups会从集群的成员的中随机挑选一个成员，并向这个成员发送重传输请求。这样做优点是对于进行了负载均衡，缺点是随机挑选的那个成员可能也没有收到消息，在这种情况下还需要继续发送重传输请求。需要注意的是，如果这个属性设置为true，那么discard_delivered_msgs属性必须设置为false。
- discard_delivered_msgs 如果discard_delivered_msgs设置为true，那么集群中的成员不会缓存其它成员发送的消息（因此不需要STABLE协议来对这些消息进行垃圾收集）。这意味着重传输请求只能发送给消息的最初发送者。

- max_xmit_buf_size 通常收到的消息会缓存在retransmission table中，这个属性指定了retransmission table的上限。如果retransmission table达到上限，那么旧的项目会被丢弃。需要主要的是，设置这个属性可能导致消息丢失。

4．2．2 UNICAST

UNICAST协议保证了单独的发送者和接收者之间传递的消息的传输可靠性，以及消息的FIFO顺序。在可靠的传输协议（例如TCP）之上， UNICAST协议并不是必须的。然而，UNICAST可以防止相同发送者上的并发的消息传递。除非希望如此，否则应该在协议栈中包含UNICAST协议。

以下是部分UNICAST协议的属性：

- retransmit_timeout 逗号分割的一系列毫秒数。例如100,200,400,800,1600。在第一次发送重传输请求前等待100ms，第二次发送重传输请求前等待 200ms，依此类推直到等待1600ms。从这以后，每次发送重传输请求前等待100ms。

4．3 Failure Detection

Failure detection 的目的是检测集群内的成员是否崩溃。当某个成员被怀疑崩溃时，那么会向集群中的每个成员发送SUSPECT 消息，以进行通知。需要注意的时是，Failure detection 并不负责从集群中清除崩溃的成员（实际上是由GMS协议负责），它只是负责发现可能已经崩溃的成员，并通知集群中的其它成员。

4．3．1 FD

FD协议基于心跳消息。如果在timeout指定的毫秒内没有接收到某个成员的应答，并且在尝试了max_tries指定的次数后，那么这个成员会被标记为可疑，并将被GMS协议从集群中清除。

每个成员向其右侧的邻居（当前view的成员列表中，该成员的下一个成员。列表中最后的成员的右侧邻居是列表的第一个成员）发送带有 FdHeader.HEARTBEAT头的消息。当邻居收到这个消息后，它会应答带有FdHeader.HEARTBEAT_ACK头的消息。每当收到应答时，FD协议的last_ack属性会被更新成当前的时间，num_tries也会设置为0。如果当前时间和last_ack之差大于timeout指定的毫秒数，那么FD协议会最多尝试max_tries 指定的次数，如果仍然没有收到应答，那么这个邻居会被标记为可疑。

4．3．2 FD_SOCK

FD_SOCK协议基于一个有TCP sockets组成的环，即集群中的每个成员都通过TCP socket连接到右侧的邻居（当前view的成员列表中，该成员的下一个成员。列表中最后的成员的右侧邻居是列表的第一个成员）。当某个成员检测到它的邻居非正常地关闭了TCP socket之后，那么它会把这个邻居标记为可疑。

4．4 Miscellaneous

4．4．1 STABLE

为了响应可能的重传输请求，集群中的成员需要保存一定数量的消息直到它确定这些消息已经被集群中所有的成员成功地接收。对于某个消息M来说， message stability 意味着M已经被集群中所有的成员接收。

STABLE协议周期性地（或者收到消息的字节数达到的配置的上限）向集群中的所有成员发送stable messages，这些消息中包含了特定成员收到的最大序号。当集群中的每个成员都收到了其它所有成员的stable

messages后，可以计算出目前每个成员已经收到的消息的最小序号，接下来这个序号被发送到集群中每个成员，最后每个成员会从自己的 retransmission tables中删除小于这个最小序号的最小消息。需要注意的是，如果没有在协议栈中配置STABLE，那么可能会导致内存耗尽。以下是个配置STABLE 协议的例子：

```
<pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000" max_bytes="1000000"/>
```

以上例子中stability_delay属性指定，在发送消息前等待1～1000毫秒，以避免所有的成员同时发送消息。desired_avg_gossip属性指定发送stable messages的周期，单位是毫秒，如果是0，那么禁用周期检查。max_bytes指定了在发送stable message消息前，接收到的消息的最大字节数。

4．4．2 pbcast.FLUSH

4．2 Reliable Message中介绍了保证消息可靠传输的协议，但是在某些情况下，这种保证是不够的，考虑以下情况：
集群中某个成员A向集群发送消息M1，此时A的当前View是V1={A，B，C}，也就是说A认为M1将发送到A（如果Channel.LOCAL选项是true）、B和C。正在此时，D也加入到集群中，那么D可能会，也可能不会收到M1。通过在协议栈中配置FLUSH协议可以保证：

- 发送到V1的消息只会被传递到V1。所以以上例子中D不会收到M1。
- 在安装V2前，集群中所有的成员都收到相同的消息。例如一个集群V1={A,B,C}中，C发送了5个消息，A收到了C发送的这5个消息，但是B 只收到了其中前3个。如果此时C崩溃，那么FLUSH协议会保证，在安装V2={A,B} 前，B会收到所有C发送过的消息。在这种情况下，A会向B发送后两个消息。

通常，在以下两种情况下需要使用FLUSH协议：

- State transfer 当某个成员请求状态传递时，它通知其它成员停止发送消息并等待响应。接下来coordinator会将状态发送给这个成员。当该成员接收到状态后，它通知其它成员可以继续发送消息。
- View changes 在安装新的view时，所有发送到V1的消息都会被传递到V1。

FLUSH协议通常在STATE_TRANSFER、STATE_TRANSFER 或者 GMS 协议之上。此外需要注意的时，FLUSH协议必须是协议栈的最上层协议。除了JGroups自动处理FLUSH之外，JGroups也允许开发人员显式调用 Channel.startFlush()方法发起flush。在Channel.startFlush()方法返回后，在调用 Channel.stopFlush()方法之前，可以保证集群中的所有成员不能发送消息，而且Channel.startFlush()方法调用前发送的消息都会被所有成员接收。在调用了Channel.stopFlush()方法之后，集群中的所有成员可以继续发送消息。

如果将Channel.BLOCK属性设置为true（缺省是false），那么可以在flush阶段得到通知。如果采用poll方式，那么在某个成员调用Channel.startFlush()方法后，其它成员会收到EVENT.BLOCK消息，这些成员应该发送EVENT.BLOCK_OK消息进行响应。如果采用push方式，那么channel上注册的MembershipListener的block()方法会被调用。

4．4．3 MERGE2

假设由于某种原因（例如switch故障），某个集群{A,B,C,D,E}，分裂为两个子集群{A,B,C} 和{D,E}，A、B和C

可以互相ping通，D和E可以互相ping通，但是A、B和C却ping不通D和E。在这种情况下，由于两个子集群独立工作，会导致这两个子集群的状态并不相同。当故障解除后，MERGE2协议会通知集群中的成员，这两个子集群将合并成一个。

至于如何处理状态的合并，需要应用程序自己决定，这是因为JGroups并不了解集群的状态。需要注意的是，用于合并的状态的代码应该在单独的线程中执行。一种简单的处理方式是对于原来是主子集群中的成员不做任何处理，对于其它的成员则丢弃当前状态，从合并后集群的coordinator处重新获得状态。