Tutorial Desenvolvendo um Chat Utilizando JGroups

O objetivo deste tutorial é mostrar como configurar o JGroups e como escrever um aplicativo simples mostrando os principais métodos da API.

Capítulo 1. Instalação

1.1. Download

A bliblioteca JGroups, será utilizada para este tutorial e pode ser baixada pelo link: https://sourceforge.net/projects/javagroups/files/JGroups. Para este tutorial, é utilizada a versão binária do JGroups 3.0, então baixe um dos arquivos jgroups-3.x.y.jar (por exemplo, jgroups-3.6.15.Final.jar).

Este arquivo JAR contém:

- Core do JGroups, demonstração e classes de teste (selecionadas).
- INSTALL.html: instruções detalhadas de configuração, além de resolução de problemas.
- Exemplos de arquivos de configuração, por exemplo, udp.xml ou tcp.xml.

OBS: Observe que o JGroups 3.0 requer o JDK 6.

1.2. Configuração

Adicione jgroups-3.x.y.jar ao seu *classpath*. Caso esteja utilizando o sistema de criação de log log4j, também deverá incluir log4j.jar (isso não é necessário se estiver utilizando o sistema de criação de log do JDK).

1.3 Testando sua configuração

Para ver se seu sistema pode encontrar as classes JGroups, execute o seguinte comando:

```
java org.jgroups.Version
```

ou

```
java -jar jgroups-3.x.y.jar
```

Você deve ver a seguinte saída (mais ou menos) se a classe for encontrada:

```
$ java org.jgroups.Version
   Versão: 3.6.15.Beta1
```

Capítulo 2. Desenvolvendo uma aplicação simples

O objetivo deste capítulo é escrever um simples aplicativo de bate-papo baseado em texto (SimpleChat), com os seguintes recursos:

- Todas as instâncias do *chat* se encontram e formam um *cluster*.
- Não há necessidade de executar um servidor central de bate-papo ao qual as instâncias precisam se conectar. Portanto, não há um único ponto de falha.
- Uma mensagem de bate-papo é enviada para todas as instâncias do *cluster*.
- Uma instância recebe um retorno de chamada de notificação quando outra instância sai (ou falha) e quando outras instâncias ingressam.
- (Opcional) Mantemos um estado compartilhado comum em todo o cluster, por exemplo, o histórico de bate-papo. Novas instâncias adquirem esse histórico de instâncias existentes.

2.1. Criando um canal e ingressando em um cluster

Para entrar em um cluster, usaremos um *JChannel*. Uma instância do *JChannel* é criada com uma configuração (por exemplo, um arquivo XML) que define as propriedades do canal. Para realmente se conectar ao *cluster*, o método `connect(String clustername)` é utilizado. Todas as instâncias de canal que chamam `connect()` com o mesmo argumento se juntarão ao mesmo *cluster*. Então, será criado um *JChannel* e conectado a um cluster chamado "*ChatCluster*":

```
import org.jgroups.JChannel;

public class SimpleChat {
    JChannel channel;
    String user_name = System.getProperty("user.name", "n/a");

    private void start() throws Exception {
        channel = new JChannel(); //use o config default, udp.xml
        channel.connect("ChatCluster");
    }

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

Primeiro será criado um canal usando o construtor vazio. Isso configura o canal com as propriedades padrão. Alternativamente, seria possível passar um arquivo XML para configurar o canal, por exemplo, novo `JChannel("udp.xml")`.

O método `connect()` associa o *cluster "ChatCluster"*. Observe que não é necessário criar explicitamente um *cluster* de antemão; `connect()` cria o *cluster* se for a primeira instância. Todas as instâncias que se juntam ao mesmo *cluster* estarão no mesmo *cluster*, por exemplo, se tivermos:

- ch1 juntando "cluster-um"
- ch2 juntando "cluster-dois"
- ch3 juntando "cluster-três"
- ch4 juntando "cluster-quatro"
- ch5 juntando "cluster-cinco"

Então, assim tendo 3 *clusters*: "cluster-um" com instâncias ch1 e ch4, "cluster-dois" com ch2 e ch3, e "cluster-três" com apenas ch5.

2.2. O loop de eventos principal e o envio de mensagens de chat

Agora, é executado um loop de eventos, que lê a entrada de `stdin('uma mensagem')` e a envia para todas as instâncias atualmente no *cluster*. Quando "*exit*" ou "*quit*" são inseridos, saímos do *loop* e fechamos o canal.

```
private void start() throws Exception {
    channel = new JChannel();
    channel.connect("ChatCluster");
    eventLoop();
    channel.close();
}

private void eventLoop() {
     BufferedReader in = new BufferedReader(new InputStreamReader(
System.in));
    while(true) {
        try {
            System.out.print("> ");
            System.out.flush();
            String line = in.readLine().toLowerCase();
            if(line.startsWith("quit") || line.startsWith("exit"))
                break;

            line="[" + user_name + "] " + line;
            Message msg = new Message(null, null, line);
            channel.send(msg);
        }
        catch(Exception e) {
        }
    }
}
```

É adicionada a chamada a `eventLoop()` e o fechamento do canal ao método `start()` e fornecida uma implementação de *eventLoop*.

O *loop* de eventos bloqueia até que uma nova linha esteja pronta (a partir da entrada padrão) e, em seguida, envia uma mensagem ao *cluster*. Isso é feito criando uma nova mensagem e chamando `Channel.send()` com ela como argumento.

- O primeiro argumento do construtor *Message* é o endereço de destino. Um endereço de destino nulo envia a mensagem para todos no *cluster* (um endereço não nulo de uma instância envia a mensagem para apenas uma instância).
- O segundo argumento é o nosso próprio endereço. Isso é nulo também, já que a pilha irá inserir o endereço correto de qualquer maneira.
- O terceiro argumento é a linha que se lê de *stdin*, isso usa a serialização Java para criar um `buffer byte[ ]` e definir a carga útil da mensagem. Note que também seria possível serializar o objeto (que é na verdade o caminho recomendado!) E usar o construtor de *Message* que pega um `buffer byte [ ]` como terceiro argumento.

O aplicativo agora está totalmente funcional, exceto que ainda não recebemos mensagens ou visualizamos notificações. Isso é feito na próxima seção abaixo.

2.3. Recebendo mensagens e visualizando notificações de mudança

Agora iremos registrar como um receptor para receber mensagens e visualizar as alterações. Para esse fim, seria possivel implementar o *org.jgroups.Receiver*, no entanto, vamos estender o *ReceiverAdapter*, que tem implementações padrão, e apenas substitui os callbacks (`receive()` e `viewChange()`) nos quais estamos interessados. Agora precisamos estender o *ReceiverAdapter*:

```
public class SimpleChat extends ReceiverAdapter {
```

Defina o receptor em `start()`:

```
private void start() throws Exception {
    channel=new JChannel();
    channel.setReceiver(this);
    channel.connect("ChatCluster");
    eventLoop();
    channel.close();
}
```

E implemente *receive()* e *viewAccepted()*:

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

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

O callback *viewAccepted()* é chamado sempre que uma nova instância se associa ao *cluster*, ou uma instância existente sai (falhas incluídas). Seu método *toString()* imprime o ID de exibição (um ID crescente) e uma lista das instâncias atuais no *cluster*
Em *receive()*, recebemos uma mensagem como argumento. Nós simplesmente obtemos seu buffer como um objeto (novamente usando a serialização Java) e o imprimimos para *stdout*. Também imprimimos o endereço do remetente (*Message.getSrc()*).
Note que nós também podemos pegar o *buffer byte [ ] (payload)* chamando *Message.getBuffer()* e depois desserializando-o, *String line = new String (msg.getBuffer())*.

2.4. Testando o aplicativo SimpleChat

Agora que o aplicativo de demonstração do bate-papo está totalmente funcional, vamos testá-lo. Inicie uma instância do SimpleChat:

```
[linux]/home/bela$ java SimpleChat

-------------------------------------------------------------------
GMS: address=linux-48776, cluster=ChatCluster, physical address=192.168.1.5:42442
-------------------------------------------------------------------
** view: [linux-48776|0] [linux-48776]
>
```

O nome desta instância é linux-48776 e o endereço físico é 192.168.1.5:42442 (endereço IP: porta). Um nome é gerado pelos JGroups (usando o nome do host e um short aleatório) se o usuário não configurá-lo. O nome permanece com uma instância para seu tempo de vida e é mapeado para um UUID subjacente. O UUID então mapeia para um endereço físico. Nós começamos a primeira instância, vamos começar a segunda instância:

```
[linux]/home/bela$ java SimpleChat

-------------------------------------------------------------------
GMS: address=linux-37238, cluster=ChatCluster, physical address=192.168.1.5:40710
-------------------------------------------------------------------
** view: [linux-48776|1] [linux-48776, linux-37238]
>
```

A lista de clusters agora é [linux-48776, linux-37238], mostrando a primeira e segunda instância que se juntou ao *cluster*. Note que a primeira instância (linux-48776) também recebeu a mesma *view*, então ambas as instâncias têm exatamente a mesma *view* com a mesma ordenação de suas instâncias na lista. As instâncias são listadas em ordem de associação ao *cluster*, com a instância mais antiga como primeiro elemento.
Enviar mensagens agora é tão simples quanto digitar uma mensagem após o *prompt* e pressionar *enter*. A mensagem será enviada para o cluster e, portanto, será recebida pelas duas instâncias, incluindo o remetente.
Quando "*exit*" ou "*quit*" é inserido, a instância deixará o *cluster*. Isto significa que uma nova visão será instalada imediatamente.
Para simular uma falha, simplesmente mate uma instância (por exemplo, via CTRL-C ou do gerenciador de processo). A outra instância sobrevivente receberá uma nova visualização, com apenas uma instância (própria) e excluindo a instância com falha.

2.5. Créditos extras: mantendo o estado do cluster compartilhado

A transferência de estado no JGroups é feita implementando-se dois *callbacks* (`getState()` e `setState()`) e chamando o método `JChannel.getState()`. Observe que, para poder usar a transferência de estado em um aplicativo, a pilha de protocolos precisa ter um protocolo de transferência de estado (a pilha padrão usada pelo aplicativo de demonstração é).
O método `start()` agora é modificado para incluir a chamada para `JChannel.getState()`:

```
private void start() throws Exception {
    channel=new JChannel();
    channel.setReceiver(this);
    channel.connect("ChatCluster");
    channel.getState(null, 10000);
    eventLoop();
    channel.close();
}
```

O primeiro argumento do método `getState()` é a instância de destino, e `null` significa obter o estado da primeira instância (o coordenador). O segundo argumento é o tempo limite; aqui estamos dispostos a esperar 10 segundos para transferir o estado. Se o estado não puder ser transferido dentro desse período, uma exceção será lançada. 0 significa esperar para sempre.
ReceiverAdapter define um retorno de chamada `getState()` que é chamado em uma instância existente (geralmente o coordenador) para buscar o estado do *cluster*. Em nosso aplicativo de demonstração, definimos o estado como a conversa de bate-papo. Esta é uma lista simples, à qual acrescentamos todas as mensagens que recebemos. (Observe que esse provavelmente não é o melhor exemplo de estado, pois esse estado sempre cresce. Como solução alternativa, podemos ter uma lista limitada, o que não é feito aqui).
A lista é definida como uma variável de instância:

```
final List<String> state=new LinkedList<String>();
```

Claro, agora precisamos modificar o *receive()* para adicionar cada mensagem recebida ao nosso estado:

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

    synchronized(state) {
        state.add(line);
    }
}
```

A implementação de retorno de chamada *getState()* é

```
public void getState(OutputStream output) throws Exception {
    synchronized(state) {
        Util.objectToStream(state, new DataOutputStream(output));
    }
}
```

O método *getState()* é chamado no provedor de estado , ou seja. uma instância existente, para retornar o estado do *cluster* compartilhado. É passado um fluxo de saída para o qual o estado tem que ser escrito. Observe que o JGroups fecha esse fluxo automaticamente depois que o estado foi gravado, mesmo no caso de uma exceção, portanto, o fluxo não precisa ser fechado.
Como o acesso ao estado pode ser simultâneo, nós o sincronizamos. Em seguida, chamamos *Util.objectToStream()*, que é um método do utilitário JGroups, que grava um objeto em um fluxo de saída.
O método *setState()* é chamado no estado solicitante, a instância que chamou *JChannel.getState()*. Sua tarefa é ler o estado do fluxo de entrada e configurá-lo de acordo:

```
public void setState(InputStream input) throws Exception {
    List<String> list;
                        list=(List<String>)Util.objectFromStream(new
DataInputStream(input));

    synchronized(state) {
        state.clear();
        state.addAll(list);
    }

          System.out.println(list.size()  +  "  messages  in  chat
history):");
```

```
    for(String str: list) {
        System.out.println(str);
    }
}
```

Novamente, chamamos um método utilitário JGroups (`Util.objectFromStream()`) para criar um objeto a partir de um fluxo de entrada.
Em seguida, sincronizamos o estado e definimos seu conteúdo a partir do estado recebido.
Também imprimimos o número de mensagens no histórico de bate-papo recebido para o *stdout*. Observe que isso não é viável com um histórico de bate-papo grande, mas - novamente - poderíamos ter uma lista de histórico de bate-papo limitada.

2.7. Conclusão

Neste tutorial, mostramos como criar um canal, ingressar e sair de um cluster, enviar e receber mensagens, ser notificado sobre mudanças de exibição e implementar a transferência de estado. Essa é a principal funcionalidade fornecida pelos JGroups por meio das APIs *JChannel* e *Receiver*.
O JGroups tem mais duas áreas que não foram cobertas: blocos de construção e a pilha de protocolos.
Blocos de construção são classes que residem na parte superior de um *JChannel* que fornecem um nível de abstração mais alto, por exemplo, correlacionadores de solicitação-resposta, chamadas de método em todo o cluster, *hashmaps* replicados e assim por diante.
A pilha de protocolos permite a personalização completa dos JGroups: os protocolos podem ser configurados, removidos, substituídos, aprimorados ou novos protocolos podem ser gravados e adicionados à pilha

O código para o SimpleChat pode ser encontrado no link:
http://www.jgroups.org/tutorial/code/SimpleChat.java

Web site do JGroups: http://www.jgroups.org

Tutorial em inglês: http://www.jgroups.org/tutorial/html/ch02.html