Threads e Sockets em Java

Threads em Java

Programas e Processos

Um programa é um conceito estático, isto é, um programa é um arquivo em disco que contém um código executável por uma CPU. Quando este programa é executado dizemos que ele é um processo. Portanto um processo é um programa em execução, ou um conceito dinâmico. Note que um programa em execução aloca recursos, como memória, disco, impressora, isto é tudo o que precisa para ser executado. Podemos até considerar a CPU como um recurso alocado por um processo, uma vez que podem haver vários processos em execução ao mesmo tempo e só um deles está com o recurso CPU num determinado instante.
Um mesmo programa pode ser executado várias vezes simultaneamente. Assim, podemos ter um só programa e vários processos (deste programa) em execução simultaneamente. Uma outra forma de dizer é que temos várias linhas de execução deste programa.
O sistema operacional controla a execução dos vários processos:
a) dando uma fatia de tempo para cada um de acordo com algum esquema de prioridade.
b) garantindo o sincronismo entre os processos quando os mesmo precisam trocar informações.

Threads em Java

Em Java é possível lançar várias linhas de execução do mesmo programa. Chamamos a isso de Threads ou MultiThreading. A diferença com os processos e programas acima é que o Java é interpretado. Quem cuida dos vários Threads de um programa é o próprio interpretador Java.
Algumas vantagens em relação aos processos:
a) O chaveamento entre os threads é mais rápido que o chaveamento entre processos
b) A troca de mensagens entre os threads também é mais eficiente.

Claro que essa maior eficiência ocorre porque o interpretador tem o controle maior sobre os threads. No entanto existe a ineficiência do interpretador que é grande.

Vejamos primeiramente um exemplo de execução seqüencial.
No exemplo abaixo o método `f()` da classe `TesteA` é chamado sequencialmente 5 vezes. A cada chamada espera-se 1000 milisegundos:

```
// Sequencial.java
import java.lang.Thread;

public class Sequencial extends Object {

 public static void main(String args[]) throws Exception {
   int i;
   TesteA TA = new TesteA();

   for (i=0; i<5; i++) {
     TA.f();
     Thread.sleep(1000);
   }
```

```
 }
}

class TesteA {
  private int v;
  public void f() {
   v++;
   System.out.println("Valor de v: "+v);
  }
}
```

A saída seria:

```
Valor de v: 1
Valor de v: 2
Valor de v: 3
Valor de v: 4
Valor de v: 5
```

Vejamos agora um exemplo contendo duas linhas de execução. Para diferenciar a chamada normal da chamada via Thread colocamos um parâmetro no método `f(String x)`.
Em primeiro lugar lançamos o método `f` da classe `TesteA`, usando a construção Thread. Esta execução irá concorrer com as chamadas consecutivas e sequenciais normais do método `f`:

```
// SequencialA.java
import java.lang.Thread;

public class SequencialA extends Object {

 public static void main(String args[]) throws Exception {
   int i;
   TesteA TA = new TesteA();

   TesteA TTA = new TesteA();
   Thread thA = new Thread(TTA);
   thA.start();

   for (i=0; i<5; i++) {
     TA.f("normal");
     Thread.sleep(1000);
   }
 }
}

class TesteA implements Runnable {
  private int v;
  public void run() {
     f("run");
  }
  public void f(String x) {
   v++;
   System.out.println(x+" Valor de v: "+v);
```

```
   }
}
```

A saida seria:

```
normal Valor de v: 1
run Valor de v: 1
normal Valor de v: 2
normal Valor de v: 3
normal Valor de v: 4
normal Valor de v: 5
```

Note que a chamada via thread entra no meio da saida das chamadas normais.

A interface Runnable

O exemplo acima mostra a primeira forma de lançar um thread, implementando a interface **`Runnable`**. Toda classe que implementa a interface **`Runnable`** deve especificar um método cuja assinatura é **`public void run()`**, executado no momento em que a linha de execução é inicializada.

Os comandos abaixo cria uma nova linha de execução:

```
TesteA TTA = new TesteA();
Thread thA = new Thread(TTA);
thA.start();
```

Outro exemplo com a interface Runnable

No exemplo abaixo lançamos 3 vezes alternadamente o método f via Thread e via normal:

```
// SequencialB.java
import java.lang.Thread;

public class SequencialB extends Object {

 public static void main(String args[]) throws Exception {

   TesteA TA = new TesteA();
   TesteA TB = new TesteA();
   TesteA TC = new TesteA();

   TesteA TTA = new TesteA();
   Thread thA = new Thread(TTA);
   thA.start();
   TA.f("normal");
   Thread.sleep(500);

   TesteA TTB = new TesteA();
   Thread thB = new Thread(TTB);
   thB.start();
```

```
    TB.f("normal");
    Thread.sleep(500);

    TesteA TTC = new TesteA();
    Thread thC = new Thread(TTC);
    thC.start();
    TC.f("normal");
    Thread.sleep(500);
  }
}

class TesteA implements Runnable {
   private int v;
   public void run() {
      f("run");
   }
   public void f(String x) {
    v++;
    System.out.println(x+" Valor de v: "+v);
   }
}
```

A saida seria:

```
normal Valor de v: 1
run Valor de v: 1
normal Valor de v: 1
run Valor de v: 1
normal Valor de v: 1
run Valor de v: 1
```

Porque a variável v tem valor 1 para todas as chamadas?

Estendendo a classe Thread

A segunda forma de utilizar threads em Java é estender a própria classe Thread, presente no pacote java.lang.Thread.
Da mesmo jeito que a anterior deve estar presente o método `public void run()`, que é chamado sempre que uma linha de execução é criada para o objeto.

O exemplo abaixo é o mesmo que o `SequencialA` acima e a saida é a mesma:

```
// SequencialC.java
import java.lang.Thread;

public class SequencialC extends Object {

 public static void main(String args[]) throws Exception {
   int i;
   TesteA TA = new TesteA();
```

```
    TesteA TTA = new TesteA();
    TTA.start();

    for (i=0; i<5; i++) {
      TA.f("normal");
      Thread.sleep(1000);
    }
  }
}

class TesteA extends Thread {
  private int v;
  public void run() {
     f("run");
  }
  public void f(String x) {
   v++;
   System.out.println(x+" Valor de v: "+v);
  }
}
```

Outro exemplo estendendo a classe Thread

O exemplo abaixo é o mesmo que o `SequencialB` acima e a saida é a mesma:

```
// SequencialD.java
import java.lang.Thread;

public class SequencialD extends Object {

 public static void main(String args[]) throws Exception {

    TesteA TA = new TesteA();
    TesteA TB = new TesteA();
    TesteA TC = new TesteA();

    TesteA TTA = new TesteA();
    TTA.start();
    TA.f("normal");
    Thread.sleep(500);

    TesteA TTB = new TesteA();
    TTB.start();
    TB.f("normal");
    Thread.sleep(500);

    TesteA TTC = new TesteA();
    TTC.start();
    TC.f("normal");
    Thread.sleep(500);
  }
}

class TesteA extends Thread {
  private int v = 0;
  public void run() {
```

```
      f("run");
   }
   public void f(String x) {
    v++;
    System.out.println(x+" Valor de v: "+v);
   }
}
```

Mais um exemplo

No exemplo abaixo em vez de esperar algum tempo (Thread.sleep), o método `ff` usa efetivamente a CPU contando até 1, 2 e 3 milhões. São lançados 3 threads. Cada um deles é interrompido no meio das contagens produzindo uma saída do tipo:

```
chamada 2 - contei ate 100.000.000

chamada 1 - contei ate 100.000.000

chamada 3 - contei ate 100.000.000

chamada 3 - contei ate 200.000.000

chamada 2 - contei ate 200.000.000

chamada 1 - contei ate 200.000.000

chamada 2 - contei ate 300.000.000

chamada 3 - contei ate 300.000.000

chamada 1 - contei ate 300.000.000
```

```
// Paralelo.java
import java.lang.Thread;

public class Paralelo extends Object {

 public static void main(String args[]) throws Exception {

   Loops La = new Loops("chamada 1");
   Loops Lb = new Loops("chamada 2");
   Loops Lc = new Loops("chamada 3");

   // Lança em paralelo os contadores
   La.start();
   Lb.start();
   Lc.start();
 }
}

class Loops extends Thread {
  String st;
```

```
  // metodo construtor
  public Loops(String x) {
    st = x;
  }
  // metodo chamado pelo xx.start()
  public void run() {
     ff();
  }
  // metodo que conta
  public void ff() {
   int i;
   // conta ate 100.000.000
   for (i=0;i<100000000;i++){}
   System.out.println("\n"+st+" - contei ate 100.000.000");
   // conta ate 200.000.000
   for (i=0;i<200000000;i++){}
   System.out.println("\n"+st+" - contei ate 200.000.000");
   // conta ate 300.000.000
   for (i=0;i<300000000;i++){}
   System.out.println("\n"+st+" - contei ate 300.000.000");
  }
}
```

Threads e Sockets em Java

O exemplo abaixo e o seguinte estão no livro:
"Aprendendo Java 2"
Mello, Chiara e Villela
Novatec Editora Ltda. – www.novateceditora.com.br

Vejamos abaixo um exemplo de cliente e servidor de eco, usando sockets TCP.

- servidor – recebe uma linha do cliente e devolve essa mesma linha para o cliente.
- cliente – espera o usuário digitar uma linha, envia essa linha para o servidor, recebe essa linha de volta do servidor e mostra no vídeo.

```
// ServidorDeEco.java
import java.io.*;
import java.net.*;
public class ServidorDeEco {
 public static void main(String args[]) {
  try {
   // criando um socket que fica escutando a porta 2000.
   ServerSocket s = new ServerSocket(2000);
   // loop principal.
   while (true) {
    // Aguarda alguém se conectar. A execução do servidor
    // fica bloqueada na chamada do método accept da classe
    // ServerSocket. Quando alguém se conectar ao servidor, o
    // método desbloqueia e retorna com um objeto da classe
```

```
    // Socket, que é uma porta da comunicação.
    System.out.print("Esperando alguém se conectar...");
    Socket conexao = s.accept();
    System.out.println(" Conectou!");
    // obtendo os objetos de controle do fluxo de comunicação
    BufferedReader entrada = new BufferedReader(new
         InputStreamReader(conexao.getInputStream()));
    PrintStream saida = new
         PrintStream(conexao.getOutputStream());
    // esperando por alguma string do cliente até que ele
    // envie uma linha em branco.
    // Verificar se linha recebida não é nula.
    // Isso ocorre quando conexão é interrompida pelo cliente
    // Se a linha não for null(o objeto existe), podemos usar
    // métodos de comparação de string(caso contrário,estaria
    // tentando chamar um método de um objeto que não existe)
    String linha = entrada.readLine();
    while (linha != null && !(linha.trim().equals(""))) {
     // envia a linha de volta.
     saida.println("Eco: " + linha);
     // espera por uma nova linha.
     linha = entrada.readLine();
    }
    // se o cliente enviou linha em branco, fecha-se conexão.
    conexao.close();
    // e volta-se ao loop, esperando mais alguém se conectar
   }
  }
  catch (IOException e) {
   // caso ocorra alguma excessão de E/S, mostre qual foi
   System.out.println("IOException: " + e);
  }
 }
}
```

Vamos agora ao cliente correspondente.

```
// ClienteDeEco.java
import java.io.*;
import java.net.*;
public class ClienteDeEco  {
 public static void main(String args[]) {
  try {
   // para se conectar ao servidor, cria-se objeto Socket.
   // O primeiro parâmetro é o IP ou endereço da máquina que
```

```
    // se quer conectar e o segundo é a porta da aplicação.
    // Neste caso, usa-se o IP da máquina local (127.0.0.1)
    // e a porta da aplicação ServidorDeEco (2000).
    Socket conexao = new Socket("127.0.0.1", 2000);
    // uma vez estabelecida a comunicação, deve-se obter os
    // objetos que permitem controlar o fluxo de comunicação
    BufferedReader entrada = new BufferedReader(new
        InputStreamReader(conexao.getInputStream()));
    PrintStream saida = new
        PrintStream(conexao.getOutputStream());

    String linha;
    // objetos que permitem a leitura do teclado
    BufferedReader teclado =
       new BufferedReader(new InputStreamReader(System.in));
    // loop principal
    while (true) {
     // lê a linha do teclado
     System.out.print("> ");
     linha = teclado.readLine();
     // envia para o servidor
     saida.println(linha);
     // pega o que o servidor enviou
     linha = entrada.readLine();
     // Verifica se é linha válida, pois se for null a conexão
     // foi interrompida. Se ocorrer isso, termina a execução.
     if (linha == null) {
      System.out.println("Conexão encerrada!");
      break;
     }
     // se a linha não for nula, deve-se imprimi-la no vídeo
     System.out.println(linha);
    }
   }
   catch (IOException e) {
    // caso ocorra alguma excessão de E/S, mostre qual foi.
    System.out.println("IOException: " + e);
   }
  }
}
```

Qual o problema na solução acima?

Apenas um cliente por vez pode se conectar ao servidor. Imagine agora que em vez de um servidor de eco, tivéssemos um servidor de “chat” (bate papo). Vários clientes tinham que estar conectados ao mesmo tempo no servidor.

Para que isso possa ocorrer, a solução é usar a Thread. A linha de execução inicial dispara outra linha a cada novo cliente e fica esperando por novas conexões.

Um servidor e cliente de chat

Vamos modificar o servidor/cliente de eco acima, para um servidor/cliente de chat.

O servidor de chat deve aceitar conexão de um cliente, disparar uma thread para atender esse cliente e esperar por conexão de um novo cliente.

A thread que atende um cliente específico deve esperar que este envie uma mensagem e replicar esta mensagem para todos os clientes conectados. Quando esse cliente desconectar a thread deve avisar a todos os clientes conectados que isso ocorreu.

Portanto, é necessário que o servidor guarde em um vetor, todos os clientes conectados num dado instante.

```
// ServidorDeChat.java
import java.io.*;
import java.net.*;
import java.util.*;
public class ServidorDeChat extends Thread {
 public static void main(String args[]) {
  // instancia o vetor de clientes conectados
  clientes = new Vector();
  try {
   // criando um socket que fica escutando a porta 2222.
   ServerSocket s = new ServerSocket(2222);
   // Loop principal.
   while (true) {
    // aguarda algum cliente se conectar. A execução do
    // servidor fica bloqueada na chamada do método accept da
    // classe ServerSocket. Quando algum cliente se conectar
    // ao servidor, o método desbloqueia e retorna com um
    // objeto da classe Socket, que é porta da comunicação.
    System.out.print("Esperando alguem se conectar...");
    Socket conexao = s.accept();
    System.out.println(" Conectou!");
    // cria uma nova thread para tratar essa conexão
    Thread t = new ServidorDeChat(conexao);
    t.start();
    // voltando ao loop, esperando mais alguém se conectar.
   }
  }
  catch (IOException e) {
   // caso ocorra alguma excessão de E/S, mostre qual foi.
   System.out.println("IOException: " + e);
  }
```

```
 }

 // Parte que controla as conexões por meio de  threads.

 // Note que a instanciação está no main.
private static Vector clientes;
// socket deste cliente
private Socket conexao;
// nome deste cliente
private String meuNome;
// construtor que recebe o socket deste cliente
public ServidorDeChat(Socket s) {
 conexao = s;
}

// execução da thread
public void run() {
 try {
  // objetos que permitem controlar fluxo de comunicação
  BufferedReader entrada = new BufferedReader(new
       InputStreamReader(conexao.getInputStream()));
  PrintStream saida = new
       PrintStream(conexao.getOutputStream());
  // primeiramente, espera-se pelo nome do cliente
  meuNome = entrada.readLine();
  // agora, verifica se string recebida é valida, pois
  // sem a conexão foi interrompida, a string é null.
  // Se isso ocorrer, deve-se terminar a execução.
  if (meuNome == null) {return;}
  // Uma vez que se tem um cliente conectado e conhecido,
  // coloca-se fluxo de saída para esse cliente no vetor de
  // clientes conectados.
  clientes.add(saida);
  // clientes é objeto compartilhado por várias threads!
  // De acordo com o manual da API, os métodos são
  // sincronizados. Portanto, não há problemas de acessos
  // simultâneos.

  // Loop principal: esperando por alguma string do cliente.
  // Quando recebe, envia a todos os conectados até que o
  // cliente envie linha em branco.
  // Verificar se linha é null (conexão interrompida)
  // Se não for nula, pode-se compará-la com métodos string
  String linha = entrada.readLine();
  while (linha != null && !(linha.trim().equals(""))) {
   // reenvia a linha para todos os clientes conectados
   sendToAll(saida, " disse: ", linha);
```

```
     // espera por uma nova linha.
     linha = entrada.readLine();
    }
    // Uma vez que o cliente enviou linha em branco, retira-se
    // fluxo de saída do vetor de clientes e fecha-se conexão.
    sendToAll(saida, " saiu ", "do chat!");
    clientes.remove(saida);
    conexao.close();
   }
   catch (IOException e) {
    // Caso ocorra alguma excessão de E/S, mostre qual foi.
    System.out.println("IOException: " + e);
   }
  }

  // enviar uma mensagem para todos, menos para o próprio
  public void sendToAll(PrintStream saida, String acao,
   String linha) throws IOException {
   Enumeration e = clientes.elements();
   while (e.hasMoreElements()) {
    // obtém o fluxo de saída de um dos clientes
    PrintStream chat = (PrintStream) e.nextElement();
    // envia para todos, menos para o próprio usuário
    if (chat != saida) {chat.println(meuNome + acao + linha);}
   }
  }
}
```

O cliente deve aguardar o usuário digitar uma mensagem no teclado e enviar essa mensagem ao servidor.

Mas não é tão simples assim. Há um problema: mensagens podem chegar a qualquer momento do servidor e devem ser mostradas no vídeo. Se você pensou também em thread, acertou. Uma thread é lançada no início e fica esperando qualquer mensagem do servidor para apresentá-la no vídeo. A linha de execução principal do cliente se encarrega de esperar uma mensagem digitada pelo usuário e enviá-la para o servidor.

```
// ClienteDeChat.java
import java.io.*;
import java.net.*;
public class ClienteDeChat extends Thread {
 // Flag que indica quando se deve terminar a execução.
 private static boolean done = false;
 public static void main(String args[]) {
  try {
   // Para se conectar a algum servidor, basta se criar um
```

```
   // objeto da classe Socket. O primeiro parâmetro é o IP ou
   // o endereço da máquina a qual se quer conectar e o
   // segundo parâmetro é a porta da aplicação. Neste caso,
   // utiliza-se o IP da máquina local (127.0.0.1) e a porta
   // da aplicação ServidorDeChat. Nada impede a mudança
   // desses valores, tentando estabelecer uma conexão com
   // outras portas em outras máquinas.
   Socket conexao = new Socket("127.0.0.1", 2222);

   // uma vez estabelecida a comunicação, deve-se obter os
   // objetos que permitem controlar o fluxo de comunicação
   PrintStream saida = new
           PrintStream(conexao.getOutputStream());
   // enviar antes de tudo o nome do usuário
   BufferedReader teclado =
      new BufferedReader(new InputStreamReader(System.in));
   System.out.print("Entre com o seu nome: ");
   String meuNome = teclado.readLine();
   saida.println(meuNome);

   // Uma vez que tudo está pronto, antes de iniciar o loop
   // principal, executar a thread de recepção de mensagens.
   Thread t = new ClienteDeChat(conexao);
   t.start();

   // loop principal: obtendo uma linha digitada no teclado e
   // enviando-a para o servidor.
   String linha;
   while (true) {
    // ler a linha digitada no teclado
    System.out.print("> ");
    linha = teclado.readLine();
    // antes de enviar, verifica se a conexão não foi fechada
    if (done) {break;}
    // envia para o servidor
    saida.println(linha);
   }
  }
  catch (IOException e) {
   // Caso ocorra alguma excessão de E/S, mostre qual foi.
   System.out.println("IOException: " + e);
  }
 }

 // parte que controla a recepção de mensagens deste cliente
 private Socket conexao;
 // construtor que recebe o socket deste cliente
```

```
 public ClienteDeChat(Socket s) {
  conexao = s;
 }
 // execução da thread
 public void run() {
  try {
   BufferedReader entrada = new BufferedReader
     (new InputStreamReader(conexao.getInputStream()));
   String linha;
   while (true) {
    // pega o que o servidor enviou
    linha = entrada.readLine();
    // verifica se é uma linha válida. Pode ser que a conexão
    // foi interrompida. Neste caso, a linha é null. Se isso
    // ocorrer, termina-se a execução saindo com break
    if (linha == null) {
     System.out.println("Conexão encerrada!");
     break;
    }
    // caso a linha não seja nula, deve-se imprimi-la
    System.out.println();
    System.out.println(linha);
    System.out.print("...> ");
   }
  }
  catch (IOException e) {
   // caso ocorra alguma exceção de E/S, mostre qual foi.
   System.out.println("IOException: " + e);
  }
  // sinaliza para o main que a conexão encerrou.
  done = true;
 }
}
```