03

Reuso de threads

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/threads2/capitulo-2.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

Neste capítulo vamos ver como podemos controlar melhor as threads, mas antes disso vamos realmente enviar dados entre o cliente e o servidor.

Recebendo comandos

A ideia do nosso projeto é que o cliente envie comandos para o servidor, e baseado neles é executada uma tarefa. Vamos mostrar como receber um comando. No lado do servidor, ou seja, na classe `DistribuirTarefas`, através daquele `Socket`, é preciso pegar o `InputStream` do cliente:

```
@Override
public void run() {
    try {
        System.out.println("Distribuindo as tarefas para o cliente " + socket);
        Scanner entradaCliente = new Scanner(socket.getInputStream());

        while (entradaCliente.hasNextLine()) {
            String comando = entradaCliente.nextLine();
            System.out.println(comando);
        }
        entradaCliente.close();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Embrulhamos o `InputStream` do cliente no `Scanner` para simplificar a leitura.

No lado do cliente, usamos o `OutputStream` para enviar os dados/comandos:

```
public class ClienteTarefas {

    public static void main(String[] args) throws Exception {
        Socket socket = new Socket("localhost", 12345);
        System.out.println("Conexão Estabelecida");

        PrintStream saida = new PrintStream(socket.getOutputStream());
        saida.println("c1");

        //aguardando enter
        Scanner teclado = new Scanner(System.in);
        teclado.nextLine();

        saida.close();
        teclado.close();
        socket.close();
```

```
        }
    }
```

Ao testar o código, isto é, rodar o servidor primeiro, depois o cliente, devemos receber o comando `c1` no lado do servidor!

Reaproveitando threads

Já estabelecemos uma conexão entre cliente e servidor. A implementação é ainda bem simples, mas à medida que avançarmos no treinamento, vamos introduzir mais complexidade.

O nosso foco neste treinamento na verdade não são os sockets, mas sim as threads! Criamos esse exemplo para ter uma aplicação real e poder aplicar novos recursos sobre threads. E é exatamente isso que vamos fazer agora.

Reparem que criamos para cada novo cliente, uma nova thread dedicada. Se temos 10 clientes, vamos ter 10 threads, proporcionalmente. Como aprendemos, uma thread é mapeada para uma thread nativa do sistema operacional. Isso tem o seu custo, nós sempre devemos ter cuidado e pensar antecipadamente quantas threads a nossa aplicação pode criar para melhorar o uso dos recursos.

Felizmente o Java já vem preparado para reaproveitar as threads através de um pool. Um pool nada mais é do que um gerenciador de objetos. Ele possui um limite de objetos que podemos estabelecer. Além disso, podemos reaproveitar esses objetos! Quem conhece um pool de conexões do mundo de banco de dados, é exatamente isso que queremos utilizar para o mundo de threads.

Para usar um pool de threads, devemos utilizar a classe `Executors`. Ela possui vários métodos estáticos para criar o pool específico. No exemplo, através do método `newFixedThreadPool` criaremos um pool com uma quantidade de threads pré-definidas:

```
ExecutorService poolDeThreads = Executors.newFixedThreadPool(5);
```

Assim nunca teremos mais do que 5 threads na aplicação e elas serão reaproveitadas. A pergunta ainda é: como podemos passar a nossa tarefa (`Runnable`) para o pool? Para tal, existe o método `execute(runnable)` :

```
ExecutorService poolDeThreads = Executors.newFixedThreadPool(5);
poolDeThreads.execute(distribuirTarefas); // distribuirTarefas é o nosso Runnable
```

Quando não sabemos exatamente qual é o número ideal para o nosso pool, pode fazer sentido usar um pool que cresce dinamicamente. Novamente já há uma implementação pronta:

```
ExecutorService poolDeThreads = Executors.newCachedThreadPool();
poolDeThreads.execute(distribuirTarefas);
```

A quantidade de threads cresce à medida que a demanda aumenta. O legal é que o pool também diminui a quantidade quando uma thread fica ociosa mais de 60 segundos.

Por fim, existe um pool com uma única thread:

```
ExecutorService poolDeThreads = Executors.newSingleThreadExecutor();
poolDeThreads.execute(distribuirTarefas);
```

Vamos testar no nosso servidor a diferença entre o pool com cache e o com número fixo de threads:

```
public static void main(String[] args) throws Exception {
    System.out.println("---- Iniciando Servidor ----");
    ServerSocket servidor = new ServerSocket(12345);
    ExecutorService threadPool = Executors.newFixedThreadPool(2);//max 2 Threads

    while (true) {
        Socket socket = servidor.accept();
        System.out.println("Aceitando novo cliente na porta "+ socket.getPort());

        DistribuirTarefas distribuirTarefas = new DistribuirTarefas(socket);
        threadPool.execute(distribuirTarefas);
    }
}
```

Testando o ExecutorService

Através do nosso `ExecutorService` podemos atender no máximo 2 clientes. Se precisarmos de mais uma thread, o *service* bloqueia a execução e espera até que um outro cliente devolva uma thread.

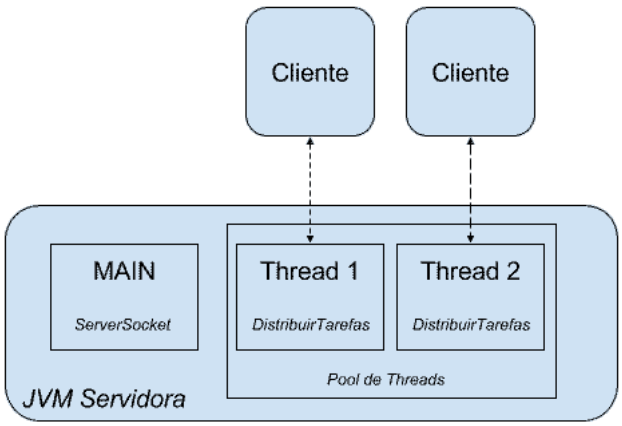
Quando rodarmos o nosso servidor e conectarmos mais de dois clientes, repare que o terceiro cliente fica bloqueado e a saída não aparece no console do servidor.

Como já falamos, quando não sabemos direito qual é o valor máximo ideal de threads, podemos utilizar o *CachedThreadPool*:

```
ExecutorService threadPool = Executors.newCachedThreadPool();
```

Ao tentar a execução novamente com vários clientes, nenhum deles será bloqueado porque o pool cresce dinamicamente.

Por fim, implementamos a seguinte arquitetura:



Ainda há muito a melhorar, mas agora é a hora dos exercícios!

O que aprendemos?

- Enviar dados pelo Socket
  - Para tal usamos no cliente o `OutputStream` que passamos para um `Scanner`
- Usar um pool de threads de implementações diferentes
  - Testamos o `FixedThreadPool` e `CachedThreadPool`
  - Refatoramos a nossa aplicação para executar a tarefa `DistribuirTarefas` pelo pool de threads
  - Para executar um `Runnable` pelo pool usamos o método `execute(..)`