02. Reuso de threads

Date	@29/08/2022
Categoria	Java
Curso	Threads em Java 2: programação concorrente avançada

Tópicos

1. imputStream e outputStream
2. Thread pool
3. Reuso de threads
4. Pool de threads
5. Tipos de pools
6. Criação de um pool
7. Mais clientes do que threads
8. Sobre pools e threads
9. Mão à obra: Alterando o client
10. Mãos à obra: Enviando comandos
11. Mãos à obra: Adicionando um Pool

Reuso de threads

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(..)`

Pool de threads

Um **pool de threads** é um **gerenciador de objetos do tipo** `Thread`. Ele gerencia uma **quantidade de threads** estabelecida, que fica aguardando por tarefas ornecidas pelos clientes. A sua grande vantagem é que além de controlarmos a quantidade de threads disponível para uso dos clientes, também podemos fazer o **reuso de threads** por clientes diferentes, não tendo o gasto de CPU de criar uma nova thread para cada cliente que chega no servidor.

Tipos de pools

Os tipos de pool's de thread são `newFixedThreadPool`, `newCachedThreadPool` e `newSingleThreadExecutor`.

A `newFixedThreadPool` é o pool de threads em que definimos previamente a quantidade de threads que queremos utilizar. Assim, se por exemplo estabelecermos que queremos no máximo 4 threads, este número nunca será extrapolado e elas serão reaproveitadas.

A `newCachedThreadPool` é o pool de threads que cresce dinamicamente de acordo com as solicitações. É ideal quando não sabemos o número exato de quantas threads vamos precisar. O legal deste pool é que ele também diminuí a quantidade de threads disponíveis quando uma thread fica ociosa por mais de 60 segundos.

A `newSingleThreadExecutor` é o pool de threads que só permite uma única thread.

Criação de um pool

O método correto de criar um pool de threads é o : `ExecutorService poolDeThreads = Executors.newFixedThreadPool(5);`

Lembrando que também temos as opções de criar o pool como `newCachedThreadPool()` ou `newSingleThreadExecutor()`. `ExecutorService` é uma interface e a classe `Executors` devolve uma implementação do pool, através dos métodos mencionados em cima. Também podemos dizer que a classe `Executors` é uma fábrica de pools!

Sobre pools e threads

Devemos utilizar o método `execute()` de um pool de threads para pedir a execução de uma tarefa. E já que será executado por uma thread, devemos passar um objeto que implementa `Runnable`, por exemplo:

```
Runnable distribuirTarefas = new DistribuirTarefas(socket);//DistribuirTarefas é um Runnab
le
ExecutorService poolDeThreads = Executors.newFixedThreadPool(5);
poolDeThreads.execute(distribuirTarefas);
```

O `newCachedThreadPool` apesar de ser dinâmico, faz sim o *reaproveitamento de threads*. A sua única diferença é que ele descarta as threads que ficam ociosas por mais de 60 segundos. Diferentemente do `newFixedThreadPool`, o `newCachedThreadPool` pode crescer e diminuir dinamicamente (sob demanda).

Para saber mais: As interfaces do Pool de threads

Nesse capítulo usamos o pool de threads no lado do servidor. Vimos que existem várias implementações, por exemplo `CachedThreadPool`:

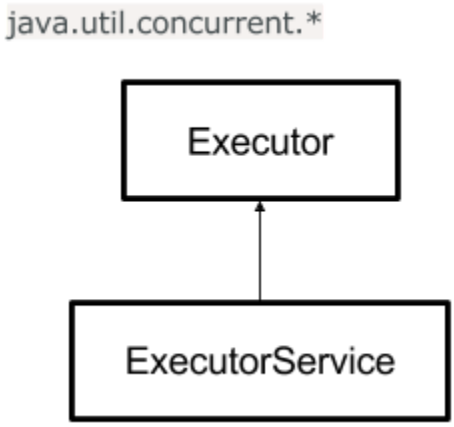
```
ExecutorService pool = Executors.newCachedThreadPool();
```

ou o `SingleThreadExecutor`:

```
ExecutorService pool = Executors.newSingleThreadExecutor();
```

Todas as implementações tem em comum que eles implementam a mesma interface: link do JavaDoc

No entanto, olhando quais são os métodos que a interface define não encontramos o método `execute` que já usamos durante a aula. Isto é porque o `ExecutorService` estende uma outra interface `Executor` que possui apenas um método, justamente aquele `execute`.




```
package java.util.concurrent;

public interface Executor {
    void execute(Runnable command);
}

public interface ExecutorService extends Executor {

//outras definições de métodos
}
```

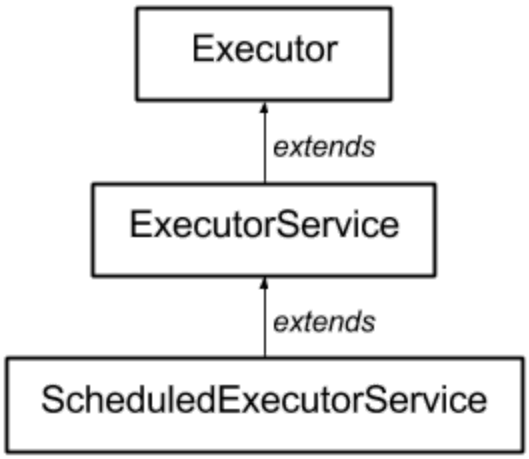
`ExecutorService` é um `Executor`! Faz sentido? Ou seja, poderíamos ter escrito no nosso projeto:

```
Executor pool = Executors.newCachedThreadPool();
```

Por enquanto seria suficiente, mas vamos utilizar ainda métodos específicos da interface `ExecutorService` como o `submit` e `shutdown`. Aguarde!

Vamos explorar mais um pouco a hierarquia das interfaces, já sabemos que a interface `ExecutorService` estende a interface `Executor`.

Existe mais uma interface, a `ScheduledExecutorService` que por sua vez estende o `ExecutorService`.



Também temos uma implementação dela já pronta para usar:

```
Runnable tarefa = ....;
ScheduledExecutorService pool = Executors.newScheduledThreadPool(4);
```

Através desse pool podemos agendar e executar uma tarefa periodicamente, por exemplo:

```
pool.scheduleAtFixedRate(tarefa, 0, 60, TimeUnit.MINUTES);//executamos uma tarefa a cada 6
0 minutos
```