

Artigo

Invista em você! Saiba como a DevMedia pode ajudar sua carreira.

Processamento assíncrono em Java com Future e FutureTask

Neste artigo vamos apresentar as classes Future e FutureTask, que permitem o processamento de tarefas de forma assíncrona e paralela no Java.



Suporte ao aluno Anotar Marcar como concluído



Artigos > Java > Processamento assíncrono em Java com Future e FutureTask



No começo do Java, para a programação de eventos assíncronos, utilizávamos a classe `Thread` e a interface `Runnable`, que permitiam o desenvolvimento de aplicações paralelas. O problema é que não é possível retornar um valor ao final da execução. Assim, foram adicionadas as classes `FutureTask`, `Future` e `Callable`, que tem mais ou menos a mesma função das anteriores, mas facilitam bastante o desenvolvimento de aplicações paralelas. Veja:

- `Future`: Classe que encapsula uma chamada feita em paralelo, sendo possível cancelar a execução de uma tarefa, descobrir se a execução já terminou com sucesso ou erro, entre outras operações;
- `FutureTask`: É uma implementação da interface Future a ser executada numa chamada em paralelo. Além disso, com ela é possível fazer as mesmas verificações que fazemos com a interface
- `Callable`: Interface para a implementação de uma execução em paralelo. É muito parecida com a interface `Runnable`, mas esta não retorna nenhum valor, enquanto a `Callable` deve retornar um valor ao final da execução;
- `ExecutorService`: Classe para o gerenciamento de execuções em paralelo, já que cria um pool de threads, iniciando e cancelando as execuções. Também é possível cancelar este, evitando assim a criação de novas tarefas.

Para demonstrar na prática como usá-las, esse artigo apresentará alguns programas simples utilizando as principais funcionalidades de cada uma.

Praticando

O primeiro exemplo mostra como implementar uma tarefa em paralelo que apenas gera números aleatórios. Isso foi feito na classe `GerarNumeroAleatorio` com a interface `Callable`. O método que deve ser implementado é o `call()` da interface, que ao final da execução retorna o valor gerado.

No main é criado um pool de threads e uma tarefa utilizando a classe `GerarNumeroAleatorio` na variável `task`. Assim, esse processo é enviado para o pool com o método `submit()` e nessa submissão é retornado um objeto do tipo `Future`. Com ele espera-se a execução terminar no `while`, que fica verificando se a `thread` terminou de executar com o método `isDone()`: isso acontecerá quando for retornado `true`. Então o valor aleatório gerado é retornado. Por fim, o valor retornado é impresso no console e o pool de threads é finalizado com o método `shutdown()`. A **Listagem 1** mostra o código dessa aplicação.

```
1 package future;
2
3 import java.util.Random;
4 import java.util.concurrent.Callable;
5 import java.util.concurrent.ExecutionException;
6 import java.util.concurrent.ExecutorService;
7 import java.util.concurrent.Executors;
```

```

8 import java.util.concurrent.Future;
9
10 /**
11 *
12 * Classe que cria um pool de threads e cria uma thread do tipo GerarNumeroAleatorio
13 * que apenas gera um numero e o retorna para a classe que
14 * criou a thread.
15 *
16 * @author Eduardo Santana
17 *
18 */
19 public class Exemplo1Print {
20
21     private static final ExecutorService threadpool =
22         Executors.newFixedThreadPool(3);
23
24     public static void main(String args[]) throws InterruptedException,
25             ExecutionException {
26
27         GerarNumeroAleatorio task = new GerarNumeroAleatorio();
28         System.out.println("Processando a tarefa ...");
29         Future<Integer> future = threadpool.submit(task);
30         while (!future.isDone()) {
31             System.out.println("A tarefa ainda não foi processada!");
32             Thread.sleep(1); // sleep for 1 millisecond
33             before checking again
34         }
35         System.out.println("Tarefa completa!");
36         long factorial = (long) future.get();
37         System.out.println("O número gerado foi: " + factorial);
38         threadpool.shutdown();
39     }
40
41     // classe que implementa a interface Callable e retorna um numero aleatorio
42     private static class GerarNumeroAleatorio implements Callable<Integer> {
43
44         @Override
45         public Integer call() {
46             Random rand = new Random();
47             Integer number = rand.nextInt(100);
48             return number;
49         }
50     }
51 }
```

Listagem 1. Exemplo utilizando a interface Callable e a classe Future

O resultado da execução do programa é mostrado na **Listagem 2**, onde é possível verificar que a tarefa não é executada assim que é submetida, então o main não avança enquanto a mesma não é executada. Quando isso acontece, o número gerado é impresso na tela.

```

1 Processando a tarefa ...
2 A tarefa ainda não foi processada!
3 A tarefa ainda não foi processada!
4 A tarefa ainda não foi processada!
5 A tarefa ainda não foi processada!
6 A tarefa ainda não foi processada!
7 A tarefa ainda não foi processada!
8 A tarefa ainda não foi processada!
9 A tarefa ainda não foi processada!
10 A tarefa ainda não foi processada!
11 A tarefa ainda não foi processada!
12 A tarefa ainda não foi processada!
13 A tarefa ainda não foi processada!
14 A tarefa ainda não foi processada!
15 A tarefa ainda não foi processada!
16 A tarefa ainda não foi processada!
17 A tarefa ainda não foi processada!
18 A tarefa ainda não foi processada!
19 A tarefa ainda não foi processada!
20 A tarefa ainda não foi processada!
21 A tarefa ainda não foi processada!
22 A tarefa ainda não foi processada!
23 A tarefa ainda não foi processada!
24 A tarefa ainda não foi processada!
25 Tarefa completa!
26 O número gerado foi: 27
```

Listagem 2. Execução da **Listagem 1**

A **Listagem 3** mostra um exemplo um pouco maior, onde é calculado o factorial de um número em uma tarefa que será executada em paralelo. Veja que o método retorna um Long com o valor calculado. O método **main** dessa classe é parecido com o da **Listagem 1**.

A tarefa é criada na variável `task` e é enviada para a execução paralela com o método `submit()`. Ao terminar é retornado o valor calculado que é impresso na tela.

A classe `Fatorial` é uma implementação da interface `Callable` e, por isso, implementa o método `call()`, que tem por objetivo calcular o factorial do número passado como parâmetro. Para isso, é feito um comando `while` que vai multiplicando o valor já calculado por um número e subtraindo 1 deste até que ele chegue em 1.

```
1 package future;
2 import java.util.concurrent.Callable;
3 import java.util.concurrent.ExecutionException;
4 import java.util.concurrent.ExecutorService;
5 import java.util.concurrent.Executors;
6 import java.util.concurrent.Future;
7 import java.util.logging.Level;
8 import java.util.logging.Logger;
9
10
11 public class Exemplo1 {
12
13     private static final ExecutorService threadpool =
14         Executors.newFixedThreadPool(3);
15
16     public static void main(String args[]) throws
17         InterruptedException, ExecutionException {
18
19         Fatorial task = new Fatorial(20);
20         System.out.println("Enviando a tarefa...");
21         Future<Long> future = threadpool.submit(task);
22         System.out.println("Task is submitted");
23         while (!future.isDone()) {
24             System.out.println("Tarefa não terminada ainda...");
25             Thread.sleep(1); // espera para tentar novamente
26         }
27         System.out.println("Tarefa finalizada!");
28         long factorial = (long) future.get();
29         System.out.println("Fatorial de 10 é: " + factorial);
30         threadpool.shutdown();
31     }
32
33
34     private static class Fatorial implements Callable<Long> {
35         private final int number;
36
37         public Fatorial(int number) {
38             this.number = number;
39         }
40
41         @Override
42         public Long call() {
43             long output = 0;
44             try {
45                 output = factorial(number);
46             } catch (InterruptedException ex) {
47                 Logger.getLogger(Exemplo1.class.getName())
48                     .log(Level.SEVERE, null, ex);
49             }
50             return output;
51         }
52
53         private long factorial(int number) throws
54             InterruptedException {
55             if (number < 0) {
56                 throw new IllegalArgumentException
57                     ("Number must be greater than zero");
58             }
59             long result = 1;
60             while (number > 0) {
61                 result = result * number;
62                 number--;
63             }
64             return result;
65         }
66     }
67 }
```

```
65 }  
66 }  
67 }
```

Listagem 3. Programa para calcular o fatorial de um número de forma paralela

Repare que os dois exemplos anteriores tinham apenas uma thread rodando em paralelo.

Na **Listagem 4** vemos a execução do código apresentado.

```
1 Enviando a tarefa...  
2 Task is submitted  
3 Tarefa não terminada ainda...  
4 Tarefa não terminada ainda...  
5 Tarefa não terminada ainda...  
6 Tarefa não terminada ainda...  
7 Tarefa não terminada ainda...  
8 Tarefa não terminada ainda...  
9 Tarefa não terminada ainda...  
10 Tarefa não terminada ainda...  
11 Tarefa não terminada ainda...  
12 Tarefa finalizada!  
13 Fatorial de 20 é: 2432902008176640000
```

Listagem 4. Execução da **Listagem 3**

Vamos aumentar um pouco a complexidade, onde na **Listagem 5** são iniciadas três tarefas utilizando também a classe `GerarNumeroAleatorio`.

Perceba que as três tarefas foram criadas nas variáveis `tarefa1`, `tarefa2` e `tarefa3` e foram enviadas para serem executadas com o método `submit()`.

Ao final da execução é feita a soma do retorno das três threads e esse valor é exibido no console.

A execução desse código é apresentada na **Listagem 6**.

```
1 package future;  
2 import java.util.Random;  
3 import java.util.concurrent.Callable;  
4 import java.util.concurrent.ExecutionException;  
5 import java.util.concurrent.ExecutorService;  
6 import java.util.concurrent.Executors;  
7 import java.util.concurrent.Future;  
8  
9 public class Exemplo1Soma {  
10  
11     private static final ExecutorService threadpool =  
12         Executors.newFixedThreadPool(3);  
13  
14     public static void main(String args[]) throws  
15         InterruptedException, ExecutionException {  
16  
17         GerarNumeroAleatorio tarefa1 = new GerarNumeroAleatorio();  
18         GerarNumeroAleatorio tarefa2 = new GerarNumeroAleatorio();  
19         GerarNumeroAleatorio tarefa3 = new GerarNumeroAleatorio();  
20  
21         System.out.println("Processando a tarefa ...");  
22         Future<Integer> futureT1 = threadpool.submit(tarefa1);  
23         Future<Integer> futureT2 = threadpool.submit(tarefa2);  
24         Future<Integer> futureT3 = threadpool.submit(tarefa3);  
25  
26  
27         while (!futureT1.isDone() && futureT2.isDone()  
28             && futureT3.isDone()) {  
29             System.out.println("As tarefas ainda não foram  
30                 processadas!");  
31             Thread.sleep(1); // sleep for 1 millisecond  
32                 before checking again  
33         }  
34         System.out.println("Tarefa completa!");  
35         long valor = futureT1.get();  
36         valor = valor + futureT2.get() + futureT3.get();  
37         System.out.println("A soma dos valores gerados são:
```

```

39     " + valor);
40     threadpool.shutdown();
41 }
42
43 private static class GerarNumeroAleatorio implements
44 Callable<Integer> {
45
46     @Override
47     public Integer call() {
48         Random rand = new Random();
49         Integer number = rand.nextInt(100);
50         System.out.println("Valor Gerado: " + number);
51         return number;
52     }
53 }
54 }
55 }
```

Listagem 5. Somando valores gerados por várias Threads

É possível verificar na **Listagem 6** que as três tarefas são executadas em paralelo, gerando cada uma um valor individual. Ao final das três execuções os três valores são somados e essa soma é apresentada na tela.

```

1 Processando a tarefa ...
2 As tarefas ainda não foram processadas!;
3 As tarefas ainda não foram processadas!;
4 As tarefas ainda não foram processadas!;
5 As tarefas ainda não foram processadas!;
6 As tarefas ainda não foram processadas!;
7 As tarefas ainda não foram processadas!;
8 Tarefa completa!
9 Valor Gerado: 21
10 Valor Gerado: 24
11 Valor Gerado: 49
12 A soma dos valores gerados são: 94
```

Listagem 6. Execução da [Listagem 5](#)

Agora vamos implementar um exemplo um pouco maior. Na **Listagem 7** é mostrado um servidor simples que recebe conexões na porta 8000. Ele é iniciado com uma `thread` onde é criada uma nova tarefa com o método `execute`. Esta é implementada com a classe `VerificaRequisicao`, que de 10 em 10 segundos verifica o status de todas as conexões feitas com o servidor, isto é, se ela foi cancela ou se foi terminada com sucesso.

Além disso, é necessário tratar a requisição de cada usuário e para isso foi implementado um processo bem simples, onde o servidor recebe uma mensagem do cliente e a responde com outra mensagem. Para isto utilizamos a classe `TrataRequisicao`, que recebe como parâmetro um Socket aberto com o cliente que fez a requisição. Foi colocado um `Thread.sleep` de cinco segundos na `Thread` que fica analisando as requisições que ainda não terminaram.

```

1 package future;
2
3 import java.io.DataInputStream;
4 import java.io.DataOutputStream;
5 import java.io.IOException;
6 import java.net.ServerSocket;
7 import java.net.Socket;
8 import java.net.SocketTimeoutException;
9 import java.util.ArrayList;
10 import java.util.List;
11 import java.util.concurrent.ExecutorService;
12 import java.util.concurrent.Executors;
13 import java.util.concurrent.Future;
14
15 /**
16 *
17 * Implementacao de um servidor bem simples para demonstrar as varias
18 * formas de implementar programacao paralela com Java.
19 *
20 * @author Eduardo Santana
21 *
22 */
23 public class Servidor implements Runnable {
```

```

24
25     private final ServerSocket serverSocket;
26     private final ExecutorService pool;
27
28     List<Future> requisicoes = new ArrayList<Future>();
29
30     public static void main(String args[]) throws IOException {
31         // cria uma Thread nova executando o servidor
32         System.out.println("Servidor no ar");
33         new Thread(new Servidor(8000, 3)).run();
34     }
35
36     public Servidor(int port, int poolSize) throws IOException {
37         serverSocket = new ServerSocket(port);
38         pool = Executors.newFixedThreadPool(poolSize);
39     }
40
41     public void run() { // run the service
42         try {
43             //cria uma tarefa para a verificação das requisições
44             pool.execute(new VerificaRequisicao(requisicoes));
45             while (true) {
46                 System.out.println("Nova requisição!");
47                 Future req = pool.submit(new TrataRequisicao
48                     (serverSocket.accept()));
49                 // armazena todas as requisições
50                 requisicoes.add(req);
51             }
52         } catch (IOException ex) {
53             pool.shutdown();
54         }
55     }
56 }
57
58 // classe que executa de 10 em 10 segundos e verifica o status das requisições
59 class VerificaRequisicao implements Runnable {
60     private final List<Future> requisicoes;
61
62     VerificaRequisicao(List<Future> requisicoes) {
63         this.requisicoes = requisicoes;
64     }
65
66     public void run() {
67         while (true) {
68             int somaTerminadas = 0;
69             int somaCanceladas = 0;
70             int somaEmExecucao = 0;
71             try {
72                 Thread.sleep(10000);
73                 for (Future f : requisicoes) {
74                     if (f.isDone()) {
75                         somaTerminadas++;
76                     } else if (f.isCancelled()) {
77                         somaCanceladas++;
78                     } else if (!f.isDone()) {
79                         somaEmExecucao++;
80                     }
81                 }
82                 System.out.println("Terminadas: " + somaTerminadas);
83                 System.out.println("Canceladas: " + somaCanceladas);
84                 System.out.println("Execução: " + somaEmExecucao);
85             } catch (InterruptedException e) {
86                 e.printStackTrace();
87             }
88         }
89     }
90 }
91
92 // classe que recebe a requisição do cliente e a responde
93 class TrataRequisicao implements Runnable {
94     private final Socket server;
95
96     TrataRequisicao(Socket server) {
97         this.server = server;
98     }
99
100    public void run() {
101        try {
102            System.out.println("Conectado a: " + server.getRemoteSocketAddress());
103            DataInputStream in = new DataInputStream(server.getInputStream());
104            System.out.println(in.readUTF());
105            DataOutputStream out = new DataOutputStream(server.getOutputStream());
106            Thread.sleep(5000);
107            out.writeUTF("Sua conexão terminou! Tchau!");

```

```

108     server.close();
109 } catch (SocketTimeoutException s) {
110     System.out.println("Socket timed out!");
111 } catch (IOException e) {
112     e.printStackTrace();
113 } catch (InterruptedException e) {
114     // TODO Auto-generated catch block
115     e.printStackTrace();
116 }
117 }
118 }
```

Listagem 7. Servidor com várias Threads

Apenas para verificar o funcionamento do servidor foi implementado um cliente bem simples, conforme mostra a

Listagem 8. A conexão é feita no endereço `localhost` da porta `8000`. Depois de conectado, o cliente envia uma mensagem para o servidor e fica esperando a resposta.

```

1 package future;
2
3 import java.io.DataInputStream;
4 import java.io.DataOutputStream;
5 import java.io.IOException;
6 import java.io.InputStream;
7 import java.io.OutputStream;
8 import java.net.Socket;
9
10 /**
11 *
12 * Cliente do servidor desenvolvido para testar as classes de execução em paralelo
13 *
14 * @author Eduardo Santana
15 *
16 */
17 public class Cliente {
18
19     public static void main(String[] args) {
20         String serverName = "localhost";
21         int port = 8000;
22         try {
23             System.out.println("Iniciando a conexão!");
24             //faz a conexão
25             Socket client = new Socket(serverName, port);
26             System.out.println("Conectado a: " + client.getRemoteSocketAddress());
27             OutputStream outToServer = client.getOutputStream();
28
29             // envia a mensagem para o servidor
30             DataOutputStream out = new DataOutputStream(outToServer);
31             out.writeUTF("Olá Servidor! " + client.getLocalSocketAddress());
32
33             // recebe a resposta do servidor
34             InputStream inFromServer = client.getInputStream();
35             DataInputStream in = new DataInputStream(inFromServer);
36
37             //imprime a resposta
38             System.out.println("Resposta: " + in.readUTF());
39             client.close();
40         } catch (IOException e) {
41             e.printStackTrace();
42         }
43     }
44 }
45 }
```

Listagem 8. Cliente que conecta ao servidor

O resultado da execução do servidor é mostrado na **Listagem 9**, onde é possível verificar as várias requisições feitas pelo cliente e a execução da Thread que examina os status da conexão. Esse status é sempre atualizado com as condições atuais do servidor. Vemos que na primeira execução dessa tarefa não existia ainda nenhuma requisição, enquanto na segunda existiam quatro execuções ainda pendentes e uma terminada; já na terceira existiam cinco terminadas e três pendentes, e finalmente, na última execução todas as oito requisições já haviam sido respondidas.

```

1 | Servidor no ar
2 | Nova requisição!
3 | Verifica o status do servidor!
4 | Terminadas: 0
5 | Canceladas: 0
6 | Execução: 0
7 | Nova requisição!
8 | Conectado a: /127.0.0.1:54841
9 | Olá Servidor! /127.0.0.1:54841
10 | Nova requisição!
11 | Conectado a: /127.0.0.1:54842
12 | Olá Servidor! /127.0.0.1:54842
13 | Nova requisição!
14 | Conectado a: /127.0.0.1:54845
15 | Olá Servidor! /127.0.0.1:54845
16 | Nova requisição!
17 | Nova requisição!
18 | Verifica o status do servidor!
19 | Terminadas: 1
20 | Canceladas: 0
21 | Execução: 4
22 | Conectado a: /127.0.0.1:54846
23 | Olá Servidor! /127.0.0.1:54846
24 | Nova requisição!
25 | Conectado a: /127.0.0.1:54848
26 | Olá Servidor! /127.0.0.1:54848
27 | Nova requisição!
28 | Conectado a: /127.0.0.1:54850
29 | Olá Servidor! /127.0.0.1:54850
30 | Nova requisição!
31 | Conectado a: /127.0.0.1:54861
32 | Olá Servidor! /127.0.0.1:54861
33 | Verifica o status do servidor!
34 | Terminadas: 5
35 | Canceladas: 0
36 | Execução: 3
37 | Conectado a: /127.0.0.1:54889
38 | Olá Servidor! /127.0.0.1:54889
39 | Verifica o status do servidor!
40 | Terminadas: 8
41 | Canceladas: 0
42 | Execução: 0

```

Listagem 9. Execução da Listagem 8

Apenas como curiosidade, a resposta do servidor para o cliente é apenas uma mensagem informando que a conexão está concluída, como mostra o código a seguir:

```

1 | Iniciando a conexão!
2 | Conectado a: localhost/127.0.0.1:8000
3 | Resposta: Sua conexão terminou! Tchau!

```

Com a classe `FutureTask` e as interfaces `Callable` e `Future` a programação de tarefas assíncronas em Java ficou muito mais fácil, evitando algumas “gambiarras” que antes eram necessárias para a implementação desse tipo de aplicação.

Espero que esse artigo seja útil! Até a próxima!

Links:

- [Documentação da classe FutureTask](#)
- [Documentação da interface Future](#)
- [Documentação da interface Callable](#)

Tecnologias:

Java



[Anotar](#)



[Marcar como concluído](#)

Confira outros conteúdos:



Novidades do Java



Teste unitário com JUnit



Jakarta EE, o futuro do Java EE

PARA QUEM QUER SER PROGRAMADOR DE VERDADE. VAGAS LIMITADAS

Em caso de dúvidas chame no whatsapp

Plano Anual

12x R\$ 59,90

12 MESES = R\$ 718,80

Formação FullStack completa	✓
+10mil exercícios gamificados	✓
+50 projetos reais	✓
Supporte online	✓

Matricule-se

MELHOR OPÇÃO

Plano Recorrente

R\$ 89,90 PRIMEIROS 3 MESES

R\$ 49,90 A PARTIR DO 4º MÊS

12 MESES = R\$ 718,80

Formação FullStack completa	✓
+10mil exercícios gamificados	✓
+50 projetos reais	✓
Supporte online	✓
Pra quem tem pouco limite no cartão	✓
Fidelidade de 12 meses	✓

Matricule-se

Nossos casos de sucesso

Leonardo Carlos

Eu sabia pouquíssimas coisas de programação antes de começar a estudar com vocês, fui me especializando em várias áreas e ferramentas que tinham na plataforma, e com essa bagagem **consegui um estágio logo no início do meu primeiro período na faculdade.**

Lucas Rodrigues

Estudo aqui na Dev desde o meio do ano passado! Nesse período a Dev me ajudou a crescer muito aqui no trampo. **Fui o primeiro desenvolvedor contratado pela minha empresa. Hoje eu lidero um time de desenvolvimento.** Minha meta é continuar estudando e praticando para ser um Full-Stack Dev!

Heráclito Júnior

Economizei 3 meses para assinar a plataforma e sendo sincero valeu muito a pena, pois a **plataforma é bem intuitiva e muuuuito didática a metodologia de ensino.** Sinto que estou EVOLUINDO a cada dia. Muito obrigado!

Julio Cahlen

Nossa! Plataforma maravilhosa. To amando o curso de desenvolvimento front-end, tinha coisas que eu ainda não tinha visto. **A didática é do jeito que qualquer pessoa consegue aprender.** Sério, to apaixonado, adorando demais.

Joelberth Sena

Adquiri o curso de vocês e logo percebi que são os melhores do Brasil. É um passo a passo incrível. **Só não aprende quem não quer. Foi o melhor investimento da minha vida!**

Feline Nunes

Foi um dos melhores investimentos que já fiz na vida e tenho aprendido bastante com a plataforma. Vocês estão fazendo parte da minha jornada nesse mundo da programação, [irei assinar meu contrato como programador](#) [graças a plataforma.](#)



Wanderson Oliveira



[Comprei a assinatura tem uma semana, aprendi mais do que 4 meses estudando outros cursos.](#) Exercícios práticos que não tem como não aprender, estão de parabéns!



José Lucas



Obrigado DevMedia, nunca presenciei [uma plataforma de ensino tão presente na vida acadêmica de seus alunos.](#) parabéns!



Eduardo Dorneles



Aprendi React na plataforma da DevMedia há cerca de 1 ano e meio... [Hoje estou há 1 ano empregado](#) trabalhando 100% com React!



Adauto Junior



Já fiz alguns cursos na área e [nenhum é tão bom quanto o de vocês.](#) Estou aprendendo muito, muito obrigado por existirem. Estão de parabéns... Espero um dia conseguir um emprego na área.

[Ver todos os casos de sucesso](#)



Por Eduardo

Em 2015



Menu

Artigos

Quem Somos

Fale conosco

Plano para Instituição de ensino

Assinatura para empresas

[Assine agora](#)

Hospedagem web por Porta 80 Web Hosting.