

CRIANDO UM FORNECEDOR ASSÍNCRONO COM MECANISMO DE RETRY

```
20 public class SupplierAssincData {
21
22     // Fornecedor de dados assíncrono com retry
23     static class AsyncDataSupplier implements Supplier<CompletableFuture<String>> {
24         private final int maxRetries;
25         private final long retryDelay;
26         private final Random random = new Random();
27         private static final double FAILURE_PROBABILITY = 0.7;
28
29         public AsyncDataSupplier(int maxRetries, long retryDelay) {
30             this.maxRetries = maxRetries;
31             this.retryDelay = retryDelay;
32         }
33
34         @Override
35         public CompletableFuture<String> get() {
36             return CompletableFuture.supplyAsync(this::fetchDataWithRetry);
37         }
38
39         private String fetchDataWithRetry() {
40             for (int attempt = 0; attempt < maxRetries; attempt++) {
41                 try {
42                     // Simulando uma operação de busca de dados que pode falhar
43                     if (random.nextDouble() < FAILURE_PROBABILITY) { // 70% de chance de falha
44                         throw new RuntimeException("Falha na busca de dados");
45                     }
46                     return "Dados obtidos com sucesso após " + (attempt + 1) + " tentativa(s)";
47                 } catch (Exception e) {
48                     if (attempt == maxRetries - 1) {
49                         throw new RuntimeException("Falha após " + maxRetries + " tentativas", e);
50                     }
51                     try {
52                         TimeUnit.MILLISECONDS.sleep(retryDelay);
53                     } catch (InterruptedException ie) {
54                         Thread.currentThread().interrupt();
55                         throw new RuntimeException("Interrompido durante o retry", ie);
56                     }
57                 }
58             }
59             throw new RuntimeException("Não deveria chegar aqui");
60         }
61     }
62
63     public static void main(String[] args) {
64
65         // Demonstração do AsyncDataSupplier
66         AsyncDataSupplier asyncSupplier = new AsyncDataSupplier(3, 1000);
67         CompletableFuture<String> future = asyncSupplier.get();
68         future.thenAccept(System.out::println)
69             .exceptionally(e -> {
70                 System.err.println("Erro: " + e.getMessage());
71                 return null;
72             });
73
74         // Aguarda a conclusão da operação assíncrona
75         try {
76             Thread.sleep(5000);
77         } catch (InterruptedException e) {
78             Thread.currentThread().interrupt();
79         }
80     }
81 }
```

Neste artigo iremos discutir uma aplicação da interface Supplier usado para criar um processo que simula a busca de dados de forma assíncrona, o que significa que ele roda em paralelo ao fluxo principal do programa, sem bloquear a execução do código.

Que tal falarmos sobre Threads e Operações Assíncronas?

Antes de entrar no código, é importante entender o que são operações assíncronas e Threads:

- **Thread:** Imagine que a CPU do seu computador é uma cozinheira e os processos são as receitas que ela está preparando. Uma thread é como uma das mãos da cozinheira. Se ela tem duas mãos, pode trabalhar em duas coisas ao mesmo tempo (por exemplo, picar legumes com uma mão e mexer uma panela com a outra). Em programação, uma thread é uma unidade de execução que pode rodar um pedaço de código em paralelo a outras threads.
- **Operação Assíncrona:** Em vez de esperar que uma tarefa termine antes de começar outra (como um cozinheiro esperando a água ferver antes de cortar os legumes), você pode iniciar uma tarefa e deixá-la rodando em segundo plano enquanto continua fazendo outras coisas. Quando a tarefa assíncrona terminar, ela “avisa” que terminou (como um timer de cozinha que apita quando a comida está pronta).

Explicando o Código Passo a Passo:

AsyncDataSupplier (Fornecedor de Dados Assíncrono)

Essa classe implementa o `Supplier<CompletableFuture<String>>`, o que significa que ela deve fornecer (supply) um valor do tipo `CompletableFuture<String>`. O `CompletableFuture` é uma ferramenta poderosa para operações assíncronas, porque permite que você execute tarefas em segundo plano e, quando estiverem completas, você pode lidar com o resultado ou com qualquer erro que tenha ocorrido.

Construtor

```
public AsyncDataSupplier(int maxRetries, long retryDelay)
```

Este é o construtor da classe. Ele recebe dois parâmetros:

- `maxRetries`: o número máximo de tentativas para buscar os dados.
- `retryDelay`: o tempo de espera entre cada tentativa, em milissegundos.

Método get()

```
@Override
public CompletableFuture<String> get() {
    return CompletableFuture.supplyAsync(this::fetchDataWithRetry);
}
```

Esse é o método principal da interface `Supplier`. Ele retorna um `CompletableFuture<String>`. O `CompletableFuture.supplyAsync()` roda o método `fetchDataWithRetry()` de forma assíncrona, ou seja, em outra thread.

Método fetchDataWithRetry()

```
private String fetchDataWithRetry() {
    for (int attempt = 0; attempt < maxRetries; attempt++) {
        try {
            if (random.nextDouble() < FAILURE_PROBABILITY) {
                throw new RuntimeException("Falha na busca de
dados");
            }
        }
    }
}
```

```

    }
    return "Dados obtidos com sucesso após " + (attempt +
1) + " tentativa(s)";
    } catch (Exception e) {
        if (attempt == maxRetries - 1) {
            throw new RuntimeException("Falha após " +
maxRetries + " tentativas", e);
        }
        try {
            TimeUnit.MILLISECONDS.sleep(retryDelay);
        } catch (InterruptedException ie) {
            Thread.currentThread().interrupt();
            throw new RuntimeException("Interrompido durante o
retry", ie);
        }
    }
    throw new RuntimeException("Não deveria chegar aqui");
}

```

Aqui é onde a mágica acontece:

Loop de Tentativas: O loop for vai tentar buscar os dados até o número máximo de tentativas (`maxRetries`).

1. **Simulação de Falha:** O código simula uma operação que pode falhar 70% das vezes (`random.nextDouble() < FAILURE_PROBABILITY`). Se falhar, ele lança uma exceção (`throw new RuntimeException("Falha na busca de dados")`).
2. **Sucesso:** Se não falhar, ele retorna uma mensagem dizendo que os dados foram obtidos com sucesso.
3. **Tentativas Extras:** Se falhar, ele verifica se ainda pode tentar de novo. Se sim, ele espera um pouco `TimeUnit.MILLISECONDS.sleep(retryDelay)` e tenta novamente.
4. **Falha Final:** Se todas as tentativas falharem, ele lança uma exceção final dizendo que todas as tentativas falharam.

O que acontece em `main()`

```

AsyncDataSupplier asyncSupplier = new AsyncDataSupplier(3, 1000);
CompletableFuture<String> future = asyncSupplier.get();
future.thenAccept(System.out::println)
    .exceptionally(e -> {
        System.err.println("Erro: " + e.getMessage());
        return null;
    });

try {
    Thread.sleep(5000);
}

```

```
} catch (InterruptedException e) {  
    Thread.currentThread().interrupt();  
}
```

Instanciação: Ele cria uma instância de `AsyncDataSupplier` com 3 tentativas e 1 segundo de atraso entre cada tentativa.

Execução Assíncrona: Ele chama `get()` para iniciar a busca de dados assíncrona. O resultado é um `CompletableFuture<String>`.

Manipulação do Resultado: Quando o `CompletableFuture` termina:

- Se tiver sucesso, imprime o resultado.
- Se falhar, imprime uma mensagem de erro.

Espera: Finalmente, ele espera 5 segundos (`Thread.sleep(5000)`) para garantir que a operação assíncrona tenha tempo de terminar antes que o programa feche.

Agora vamos explicar com mais detalhes alguns métodos dentro do contexto do código.

O método `supplyAsync(this::fetchDataWithRetry)`

O método `supplyAsync` pertence à classe `CompletableFuture` e é uma maneira de iniciar uma operação assíncrona, ou seja, rodar uma tarefa em uma thread separada, sem bloquear a thread principal.

No contexto do código:

`CompletableFuture.supplyAsync(this::fetchDataWithRetry)` cria um `CompletableFuture` que vai executar o método `fetchDataWithRetry` de forma assíncrona (em outra thread) e, quando terminar, o resultado será armazenado no `CompletableFuture`.

A expressão `this::fetchDataWithRetry` é uma referência de método, uma forma mais curta e legível de passar um método como argumento para outro método. Nesse caso, estamos dizendo ao `supplyAsync` para rodar o método `fetchDataWithRetry` do objeto atual (`this`).

Então, `supplyAsync` pega esse método e o executa em uma thread separada do fluxo principal do programa, permitindo que outras partes do código continuem rodando enquanto a operação de busca de dados ocorre em segundo plano. Quando você usa `CompletableFuture.supplyAsync()`, a thread que executa o código assíncrono (neste caso, o método `fetchDataWithRetry`) é gerenciada automaticamente por um pool de threads fornecido pelo Java.

E quando o `fetchDataWithRetry` é chamado?

```
CompletableFuture<String> future = asyncSupplier.get();
```

Quando você chama `asyncSupplier.get()`, ele executa o código dentro do método `get` da classe `AsyncDataSupplier`. Esse método faz uma única coisa: ele chama `CompletableFuture.supplyAsync(this::fetchDataWithRetry)`.

Execução Assíncrona: Como explicado antes, `supplyAsync` pega o método `fetchDataWithRetry` e o executa em uma thread separada, ou seja, fora do fluxo principal.

Isso significa que assim que você chama `asyncSupplier.get()`, o método `fetchDataWithRetry` é imediatamente "agendado" para ser executado em uma thread paralela.

Portanto, `fetchDataWithRetry` é chamado logo que `get()` é invocado, mas a execução dele ocorre em uma thread separada devido ao `supplyAsync`.

O método `thenAccept()`

O método `thenAccept` é parte da API de `CompletableFuture`. Ele é utilizado para definir o que deve acontecer quando a operação assíncrona (iniciada pelo `supplyAsync`) terminar com sucesso.

No contexto do código:

`thenAccept(System.out::println)` é chamado depois que o `CompletableFuture` gerado pelo `supplyAsync` completa sua execução. Ele pega o resultado do `CompletableFuture` (ou seja, o retorno do método `fetchDataWithRetry`) e passa esse resultado para o método `println` de `System.out`, que, por sua vez, imprime o resultado na tela.

Vamos imaginar que `fetchDataWithRetry` conseguiu obter os dados após 2 tentativas. O método retorna uma string como "Dados obtidos com sucesso após 2 tentativa(s)". Quando essa string é retornada, o `CompletableFuture` "completa" com sucesso, e o `thenAccept` é automaticamente chamado com essa string como argumento. O `System.out::println` então imprime essa string.

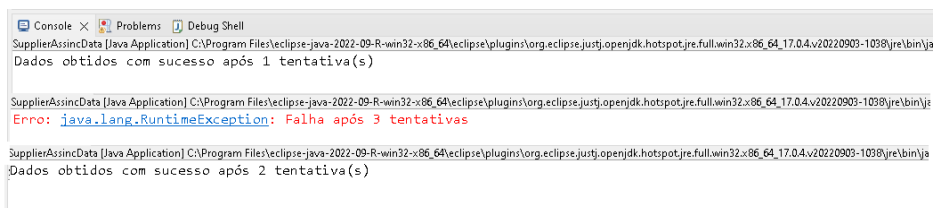
E finalmente:

```
try {
    TimeUnit.MILLISECONDS.sleep(retryDelay);
} catch (InterruptedException ie) {
    Thread.currentThread().interrupt();
    throw new RuntimeException("Interrompido durante o retry", ie);
}
```

A operação simulada (`random.nextDouble()`) é rápida, a chance de uma interrupção ocorrer durante sua execução é baixa e o `CompletableFuture` já fornece mecanismos para cancelamento e timeout, que podem ser suficientes em muitos casos.

Entretanto não é possível implementar `TimeUnit.MILLISECONDS.sleep(retryDelay)` sem lidar com a interrupção, porque o método `sleep` de `TimeUnit` (assim como `Thread.sleep`) sempre lança uma exceção `InterruptedException` se a thread for interrompida durante o `sleep`.

Abaixo estão alguns resultados da execução do código:



```
Console × Problems Debug Shell
SupplierAssincData [Java Application] C:\Program Files\ eclipse-java-2022-09-R-win32-x86_64\ eclipse\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.17.0.4.v20220903-1038\jre\bin\ja
Dados obtidos com sucesso após 1 tentativa(s)

SupplierAssincData [Java Application] C:\Program Files\ eclipse-java-2022-09-R-win32-x86_64\ eclipse\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.17.0.4.v20220903-1038\jre\bin\ja
Erro: java.lang.RuntimeException: Falha após 3 tentativas

SupplierAssincData [Java Application] C:\Program Files\ eclipse-java-2022-09-R-win32-x86_64\ eclipse\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.17.0.4.v20220903-1038\jre\bin\ja
Dados obtidos com sucesso após 2 tentativa(s)
```