Threads em Java no Spring Boot 3.2

O conceito de **threads** é fundamental para o desenvolvimento de aplicações com alto desempenho, principalmente em sistemas que exigem processamento simultâneo de várias tarefas, como servidores web, processamento de dados em tempo real e sistemas distribuídos.

Este artigo oferece uma análise técnica detalhada sobre o funcionamento de threads em Java, exemplificando a criação e execução de threads, seu monitoramento, e a evolução do uso de threads no ecossistema Java, com destaque para as inovações no **Spring Boot 3.2**.

O conteúdo inclui exemplos práticos que vão desde os conceitos mais simples até os mais avançados, para proporcionar uma compreensão clara e completa.

Threads e sua Relevância

O conceito de **multithreading** surgiu com a necessidade de aproveitar ao máximo os **processadores multi-core** e executar múltiplas tarefas simultaneamente.

Nos primeiros sistemas, as tarefas eram executadas de forma sequencial. Com o avanço da tecnologia e a introdução de **processadores multi-core**, foi possível realizar a execução **paralela**, utilizando múltiplos núcleos do processador para executar várias threads ao mesmo tempo. Isso trouxe um salto significativo na **performance** de sistemas.

No Java, o suporte a threads foi introduzido desde suas primeiras versões com a classe java.lang.Thread, que possibilitou aos desenvolvedores criar programas **multithreaded**.

Com o tempo, novas abstrações e melhorias foram sendo feitas para facilitar o uso de threads de forma mais eficiente e performática.

Thread em Java

Uma **thread** é o menor unidade de execução de um processo.

Cada thread executa seu próprio fluxo de controle, mas compartilha os recursos de memória do processo ao qual pertence, permitindo a **concorrência** entre várias tarefas.

No Java, as threads podem ser gerenciadas de duas formas principais:

- **Thread tradicional**: A criação de threads é feita diretamente por meio da classe Thread.
- **Thread virtual**: Introduzidas no **JDK 21**, permitindo a criação de threads de forma mais eficiente.

A execução de múltiplas threads pode ocorrer em dois cenários:

1. **Execução paralela**: Quando o sistema possui múltiplos núcleos de processamento, as threads podem ser alocadas em diferentes núcleos, executando de forma simultânea.
2. **Execução concorrente**: Em sistemas de núcleo único, a execução é alternada rapidamente entre as threads, de forma que o sistema parece estar executando várias tarefas ao mesmo tempo.

Criação e Execução de Threads em Java

Exemplo 1: Estendendo a Classe Thread – Simples

Vamos começar com um exemplo simples de criação de thread, onde a classe MinhaThread estende Thread e sobrescreve o método run():

```
public class MinhaThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Contagem de " + Thread.currentThread().getName() + ": " + i);
        }
    }

    public static void main(String[] args) {
        MinhaThread t1 = new MinhaThread();
        MinhaThread t2 = new MinhaThread();

        t1.start(); // Inicia a thread t1
        t2.start(); // Inicia a thread t2
    }
}
```

Entrada (Input):

- O código não exige entrada do usuário, sendo executado diretamente no main.

Saída (Output): A saída pode ser variada, já que o **agendador de threads** do sistema operacional decide qual thread será executada a cada momento, mas provavelmente será algo assim:

```
Contagem de Thread-0: 0
Contagem de Thread-1: 0
Contagem de Thread-1: 1
Contagem de Thread-0: 1
Contagem de Thread-0: 2
Contagem de Thread-1: 2
Contagem de Thread-1: 3
Contagem de Thread-0: 3
Contagem de Thread-0: 4
```

Contagem de Thread-1: 4

Exemplo 2: Implementando a Interface Runnable – Médio

Agora, vamos criar um exemplo onde usamos a interface Runnable para criar e gerenciar múltiplas threads. A vantagem dessa abordagem é a flexibilidade, pois a classe não precisa estender Thread e pode implementar outras interfaces.

```
public class TarefaRunnable implements Runnable {
    private String nome;

    public TarefaRunnable(String nome) {
        this.nome = nome;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(nome + " - Contagem: " + i);
        }
    }

    public static void main(String[] args) {
        Runnable tarefa1 = new TarefaRunnable("Thread 1");
        Runnable tarefa2 = new TarefaRunnable("Thread 2");

        Thread t1 = new Thread(tarefa1);
        Thread t2 = new Thread(tarefa2);

        t1.start();
        t2.start();
    }
}
```

Entrada (Input):

- Nenhuma entrada interativa; o código é executado diretamente pelo método main.

Saída (Output): A saída será semelhante a este exemplo:

```
Thread 1 - Contagem: 0
Thread 2 - Contagem: 0
Thread 1 - Contagem: 1
Thread 2 - Contagem: 1
Thread 1 - Contagem: 2
Thread 2 - Contagem: 2
Thread 1 - Contagem: 3
Thread 2 - Contagem: 3
Thread 1 - Contagem: 4
Thread 2 - Contagem: 4
```

Note que a ordem das saídas pode variar, pois as threads estão sendo executadas de maneira concorrente.

Exemplo 3: Thread com Sincronização – Avançado

Para evitar condições de corrida em cenários mais complexos, podemos usar a **sincronização** para garantir que apenas uma thread por vez acesse uma seção crítica do código.

Veja o exemplo abaixo, onde duas threads tentam acessar o mesmo recurso de forma segura:

```
public class ContaBancaria {
    private int saldo = 1000;

    public synchronized void sacar(int valor) {
        if (saldo >= valor) {
            saldo -= valor;
            System.out.println("Saque de " + valor + " realizado. Saldo atual: " + saldo);
        } else {
            System.out.println("Saldo insuficiente para o saque de " + valor);
        }
    }

    public static void main(String[] args) {
        ContaBancaria conta = new ContaBancaria();

        Runnable saque1 = () -> conta.sacar(600);
        Runnable saque2 = () -> conta.sacar(500);

        Thread t1 = new Thread(saque1);
        Thread t2 = new Thread(saque2);

        t1.start();
        t2.start();
    }
}
```

Entrada (Input):

- Nenhuma entrada interativa; o código executa operações de saque em threads.

Saída (Output): A saída será controlada pela sincronização, o que garante que uma thread acesse o método de saque por vez:

```
Saque de 600 realizado. Saldo atual: 400
Saldo insuficiente para o saque de 500
```

Monitoramento de Threads com JConsole

O **JConsole** é uma ferramenta útil para monitoramento de threads em tempo real.

Através dela, podemos visualizar o estado de cada thread em execução, como **executando**, **bloqueada** ou **esperando**, entre outros.

Essa ferramenta é valiosa para entender o comportamento das threads e identificar possíveis problemas de desempenho.

1. **Iniciar o JConsole**: Digite jconsole no terminal ou prompt de comando.
2. **Conectar ao Processo Java**: Após iniciar o JConsole, ele exibirá uma lista de processos Java em execução. Basta selecionar o processo que você deseja monitorar.
3. **Monitoramento das Threads**: Na aba **Threads**, você poderá visualizar o estado de todas as threads, o número de threads ativas e as estatísticas de desempenho.

Threads Virtuais no Spring Boot 3.2

Com o Spring Boot 3.2 e **JDK 21**, o uso de **threads virtuais** torna-se uma maneira mais eficiente de gerenciar a concorrência, especialmente em servidores que lidam com um grande número de requisições simultâneas.

As threads virtuais são mais leves que as threads tradicionais, permitindo a criação e gerenciamento de milhares de threads com menor custo de recursos.

Exemplo de Configuração de Threads Virtuais

No Spring Boot 3.2, as threads virtuais podem ser habilitadas da seguinte forma:

properties
spring.threads.virtual.enabled=true

Isso permite que o Spring utilize threads virtuais para processar requisições HTTP de forma mais escalável, sem comprometer o desempenho do sistema.

Este artigo abordou os conceitos fundamentais de **multithreading** em Java, desde a criação de threads simples até a execução de tarefas mais complexas com sincronização.

Apresentamos exemplos práticos e como monitorar a execução de threads, além de discutir as inovações trazidas pelas **threads virtuais** no **JDK 21** e o impacto dessas melhorias no ecossistema Spring Boot 3.2.

Aprofundar-se no uso de threads em Java é essencial para o desenvolvimento de sistemas modernos, especialmente em ambientes de alta concorrência e desempenho.

A compreensão dessas técnicas e sua aplicação prática pode otimizar significativamente a performance de suas aplicações.

EducaCiência FastCode para a comunidade.