Threads em Java

Java é amplamente utilizado em sistemas de alta performance, onde o uso eficiente de threads é crucial para maximizar a capacidade de resposta e throughput de aplicações.

Abaixo, abordamos mais exemplos reais e otimizados para desempenho em aplicações multithreaded.

Exemplo Real com ExecutorService e Tarefas Pesadas

Suponha que você esteja desenvolvendo um sistema de processamento de imagens onde várias imagens precisam ser processadas simultaneamente.

Utilizar **ExecutorService** com um pool de threads é uma estratégia altamente performática para distribuir as tarefas entre várias threads sem sobrecarregar o sistema.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class ProcessadorDeImagem implements Runnable {
    private String nomeImagem;

    public ProcessadorDeImagem(String nomeImagem) {
        this.nomeImagem = nomeImagem;
    }

    @Override
    public void run() {
        System.out.println("Processando imagem: " + nomeImagem + " - " +
Thread.currentThread().getName());
        // Simulação de processamento pesado
        try {
            Thread.sleep(3000); // Simula tempo de processamento
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

public class SistemaDeImagens {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(4); // Pool com 4 threads
        String[] imagens = {"img1.jpg", "img2.jpg", "img3.jpg", "img4.jpg", "img5.jpg"};

        for (String imagem : imagens) {
            executor.execute(new ProcessadorDeImagem(imagem));
```

```
        }

        executor.shutdown(); // Finaliza o executor após completar as tarefas
    }
}
```

- **ExecutorService** com FixedThreadPool evita overhead de criação de novas threads repetidamente, reutilizando um número fixo de threads.
- A divisão de tarefas entre as threads permite que a CPU processe várias imagens simultaneamente.

Exemplo Avançado: Paralelismo com ForkJoinPool

Em cenários reais, como em sistemas de cálculos financeiros ou simulações complexas, a divisão de tarefas em subtarefas pode aumentar significativamente a eficiência.

ForkJoinPool é ideal para dividir tarefas recursivamente em subtarefas menores, executando-as em paralelo.

Exemplo: Cálculo recursivo de uma soma em um grande array de números.

```
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

class SomaArrayTask extends RecursiveTask<Long> {
    private final long[] array;
    private final int inicio, fim;
    private static final int LIMIAR = 1000;

    public SomaArrayTask(long[] array, int inicio, int fim) {
        this.array = array;
        this.inicio = inicio;
        this.fim = fim;
    }

    @Override
    protected Long compute() {
        if (fim - inicio <= LIMIAR) {
            long soma = 0;
            for (int i = inicio; i < fim; i++) {
                soma += array[i];
            }
            return soma;
        } else {
            int meio = (inicio + fim) / 2;
            SomaArrayTask tarefa1 = new SomaArrayTask(array, inicio, meio);
            SomaArrayTask tarefa2 = new SomaArrayTask(array, meio, fim);

            tarefa1.fork(); // Divide a tarefa em duas partes
            return tarefa2.compute() + tarefa1.join();
        }
    }
}

public class ForkJoinExemplo {
    public static void main(String[] args) {
```

```
        ForkJoinPool pool = new ForkJoinPool();
        long[] array = new long[1000000];

        for (int i = 0; i < array.length; i++) {
            array[i] = i;
        }

        SomaArrayTask tarefa = new SomaArrayTask(array, 0, array.length);
        long soma = pool.invoke(tarefa);

        System.out.println("Soma total: " + soma);
    }
}
```

- **ForkJoinPool** divide a tarefa de forma recursiva, permitindo que o processamento de grandes conjuntos de dados seja feito paralelamente em várias threads.
- Utiliza um modelo de **work-stealing**, onde threads ociosas “roubam” trabalho de outras threads ocupadas, otimizando a utilização dos recursos da CPU.

Uso de CompletableFuture para Tarefas Assíncronas e Não Bloqueantes

Em aplicações de larga escala, como sistemas de recomendação ou consultas a bancos de dados distribuídos, o uso de tarefas assíncronas permite que o sistema continue respondendo a outras requisições enquanto aguarda os resultados.

```
import java.util.concurrent.CompletableFuture;

public class ExemploCompletableFuture {
    public static void main(String[] args) {
        CompletableFuture<Void> tarefa1 = CompletableFuture.runAsync(() -> {
            System.out.println("Executando tarefa 1 - " + Thread.currentThread().getName());
            // Simulação de trabalho assíncrono
            try {
                Thread.sleep(2000); // Simula um atraso
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        CompletableFuture<Void> tarefa2 = CompletableFuture.runAsync(() -> {
            System.out.println("Executando tarefa 2 - " + Thread.currentThread().getName());
        });

        CompletableFuture<Void> todasTarefas = CompletableFuture.allOf(tarefa1, tarefa2);

        todasTarefas.join(); // Aguarda ambas as tarefas completarem
        System.out.println("Todas as tarefas concluídas.");
    }
}
```

- **CompletableFuture** permite a execução assíncrona de tarefas, liberando a thread principal para continuar trabalhando em outras atividades.
- A API é eficiente para cenários onde há IO pesado ou operações remotas, pois evita o bloqueio da thread principal.

1. **Tamanho do pool de threads**: Configure o tamanho do pool com base nos recursos disponíveis. Utilize fórmulas como número de núcleos CPU * 2 para tarefas CPU-bound ou uma quantidade maior para tarefas IO-bound.
2. **Minimize a sincronização**: Sempre que possível, evite bloqueios excessivos. Sincronize apenas o que é estritamente necessário para evitar perda de paralelismo.
3. **Reduza o overhead de criação de threads**: Use sempre ExecutorService ou ForkJoinPool em vez de criar threads diretamente com new Thread(). Esses frameworks otimizam a reutilização de threads e gerenciam os recursos de forma mais eficiente.
4. **Evite compartilhamento de estados mutáveis**: Se múltiplas threads precisam acessar dados, prefira utilizar estruturas de dados imutáveis ou coleções concorrentes como ConcurrentHashMap.
5. **Monitoramento e profiling**: Utilize ferramentas como JVisualVM, Flight Recorder ou outras ferramentas de profiling para monitorar o desempenho das threads e evitar problemas como **starvation** e **deadlocks**.
6. **Lidar com exceções em threads**: Certifique-se de capturar exceções em tarefas assíncronas para evitar que erros silenciosos prejudiquem o desempenho geral do sistema.

Esses exemplos e práticas otimizam a forma como as threads são usadas, aumentando a eficiência e o desempenho das aplicações multithreaded.

EducaCiência FastCode para a comunidade