Programação Concorrente

Programação com Threads em Java

Prof. Rodrigo Campiolo

UTFPR - Universidade Tecnológica Federal do Paraná
DACOM - Departamento de Computação
BCC - Bacharelado Ciência da Computação

20 de agosto de 2018

Programação com Threads em Java

Threads em Java

- Toda thread em Java está associada com uma instância da class **Thread**.
- Há duas estratégias para uso de threads em Java:
  - Diretamente controlar a criação e o gerenciamento, isto é, realizar a instanciação e chamada sempre que for necessário.
  - Delegar as tarefas da aplicação para um **executor**.

Programação com Threads em Java

Threads implementando a interface **Runnable**

```
public class HelloRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        new Thread(new HelloRunnable()).start();
    }
}
```

Programação com Threads em Java

Threads com a subclasse **Thread**

```
public class HelloThread extends Thread {

    @Override
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        new HelloThread().start();
    }
}
```

Programação com Threads em Java

interface Runnable x subclasse Thread

- Runnable possibilita estender outras classes.
- Runnable é mais comum como entrada para métodos.
- Subclasse Thread é mais simples de usar.

Programação com Threads em Java

Atividades

1. Implemente o exemplo anterior usando Lambda Expression.
2. Faça um programa que receba um valor indicando um número de threads a serem instanciadas e teste os dois modos de criar threads em Java.
   * dica: use o **Thread.sleep** para pausar as threads por um intervalo de tempo.
3. Implemente o exemplo de código que gera a condição de disputa e tente gerar um teste para que ocorra um problema de segurança (safety).

Programação com Threads em Java

Solução 1. Threads com Lamba Expression

```
public class HelloRunnableLambda {

    public static void main(String[] args) {
        new Thread( () -> {
            System.out.println("Hello from a thread. LAMBDA, LAMBDA.");
            System.out.println("My ID is " + Thread.currentThread().getId());
        }).start();
    }
}
```

Programação com Threads em Java

Parando temporariamente threads

- **Thread.sleep** faz a thread corrente suspender a execução por um período de tempo.
- Viabiliza a liberação do processador para outras threads.
- Pode ser usado para agendar execuções periódicas.
- O tempo especificado pode ser em milissegundos ou nanosegundos.
- O tempo não é preciso, pois dependem do SO.
- Interrupções também podem finalizar o tempo de parada (implica em gerar **InterruptedException**).

Exemplo de uso do **Thread.sleep**

```
public class ThreadSleepExample implements Runnable
{
    @Override
    public void run() {
        try {
            System.out.println("Example of thread sleep");
            // Thread parada por 5s
            Thread.sleep(5000);
            System.out.println("After 5 seconds, I am here.");

            // Thread parada por 0 ms e 500.000 ns (0,5ms)
            Thread.sleep(0, 500_000);
            System.out.println("After 500.000 nanoseconds, I am here.");
        } catch (InterruptedException ex) {
            System.out.println("My sleep time was interrupted.");;
        }
    }
    //...
}
```

Interrupção de threads

- Usado para indicar que uma thread deveria parar o que está fazendo e realizar outra tarefa.
- Comumente, a tarefa consiste em finalizar a própria execução.
- Para enviar uma interrupção para uma thread X, deve se invocar o método **X.interrupt()**.
- A thread que recebe a interrupção deve suportar sua interrupção via tratamento de exceção (caso 1) ou via a verificação da flag de interrupção (caso 2).
- **Thread.interrupted()** pode ser usado para verificar periodicamente se ocorreu uma interrupção.

Interrupção de threads - Caso 1

```
    //... thread myLittleThread
    while(true) {
      // Pause for 10 seconds
      try {
          Thread.sleep(10000); //sleep suports interrupts
          // doing something
      } catch (InterruptedException e) {
          // We've been interrupted
          // Doing something
      }
    }

    System.out.println("Goodbye message: BYE. :-)");
} // end myLittleThread

// someone calls
myLittleThread.interrupt();
```

Interrupção de threads - Caso 2

```
    //... thread myLittleThread
    while(true) {
        // doing something funny, but slow

        if (Thread.interrupted()) {
            throw new InterruptedException();
        }
    }

} // end myLittleThread

// someone calls
myLittleThread.interrupt();
```

Interrupção de threads

- O mecanismo de interrupção usa uma flag interna denominada **interrupt status**.
- **Thread.interrupt()** seta essa flag.
- **Thread.interrupted()** zera a flag e **isInterrupted()** não zera a flag.
- **InterruptedException** zera a flag ao ser invocado.

Aguardando threads para continuar

- Seja **X** uma Thread, o método **X.join()** faz com que o fluxo atual aguarde a thread X finalizar para continuar a sua execução.
- Há três formas de aguardar por finalizações de threads: **join()**, **join(long millis)**, **join(long millis, int nanos)**.
- A segunda e terceira forma fazem com que a thread aguarde um tempo máximo a finalização. Se o tempo expirar, a thread continua independente da finalização das threads que estavam sendo aguardadas.
- O **join()** também deve tratar uma **InterruptedException**.
- O método **isAlive()** também pode ser usado para aguardar uma thread finalizar, pois devolve se ela continua em execução.

Exemplo **join()**

```
public class ThreadJoinExample implements Runnable
{
    @Override
    public void run() {
        try {
            System.out.println("Example of thread sleep");
            // Thread parada por 30s
            Thread.sleep(30000);
            System.out.println("After 30 seconds, I am here.");
        } catch (InterruptedException ex) {
            System.out.println("My sleep time was interrupted.");;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        System.out.println("Start main ...");
        Thread t1 = new Thread(new ThreadJoinExample());
        t1.start();
        t1.join(); //waiting t1 finish to continue
        System.out.println("Continue main ...");
        System.out.println("Stop main.");
    }
}
```

Exemplo **isAlive()**

```
    public static void main(String[] args) throws InterruptedException {
      System.out.println("Start main ...");
      Thread t1 = new Thread(new ThreadIsAliveExample());

      t1.start();

      while (t1.isAlive()) {
          System.out.println("Waiting 3 seconds before asking again...");
          Thread.sleep(3000);
      }

      System.out.println("Continue main ...");
      System.out.println("Stop main.");
}
```

Atividades

1. Faça um programa em Java que inicie três threads e, cada thread, espere um tempo aleatório para terminar.
2. Faça uma thread Java que realize a leitura de um arquivo texto com frases e exiba as frases a cada 10 segundos.
3. Faça um programa Java que envia interrupções para as threads dos exercícios anteriores. As threads devem fazer o tratamento dessas interrupções e realizar uma finalização limpa.
4. Faça uma Thread que monitora um conjunto de threads e exiba quais threads receberam sinais de interrupção.
5. Faça uma thread Java que fica aguardando uma sequência numérica de tamanho arbitrário digitado por usuário para realizar uma soma. Use o join().

Estados de uma thread em Java

- O tipo enumerado **THREAD.STATE** apresenta os seguintes estados:
  - NEW: thread não foi iniciada (start).
  - RUNNABLE: thread executando na JVM.
  - BLOCKED: thread aguarda por uma trava (lock) do monitor.
  - WAITING: thread espera por outra thread para executar uma ação.
  - TIMED_WAITING: idem WAITING, mas espera por um tempo máximo.
  - TERMINATED: thread finalizada.
- O método **Thread.getState()** retorna um desses estados.

Diagrama de estados de threads

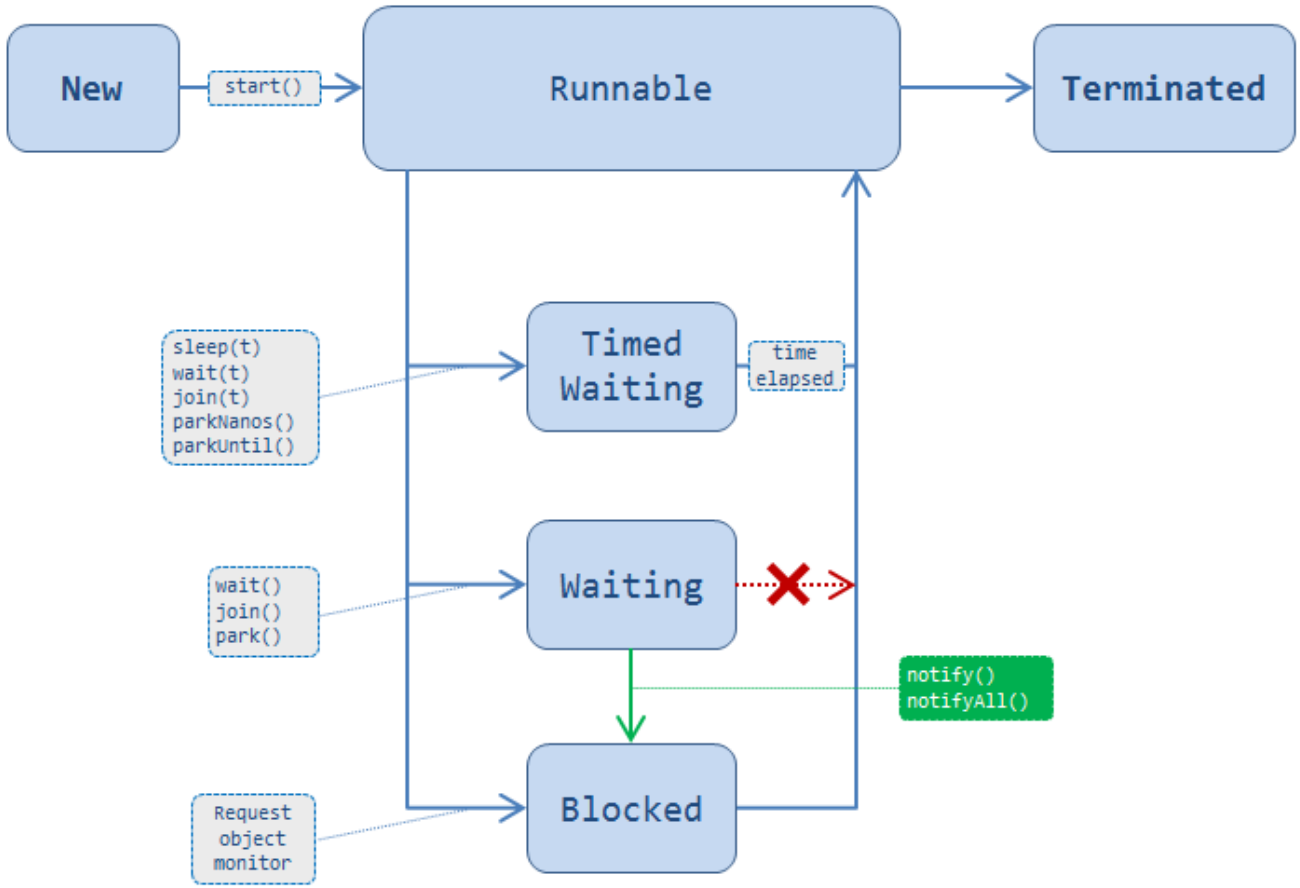

Figura: Diagrama de estados de threads em Java. (Fonte: Stackoverflow)

Prioridade de threads em Java

- Todas as threads possuem uma prioridade.
- Threads de maior prioridade são executadas antes das de menor.
- A thread tem prioridade inicial igual a thread que a criou.
- As prioridades de threads são alteradas por **setPriority(int newPriority)**.
- A prioridade atual pode ser consultada por **getPriority()**.
- Os limites e padrão de prioridade são obtidos por **Thread.MAX_PRIORITY, Thread.MIN_PRIORITY, Thread.NORM_PRIORITY**.

Exemplo de prioridade de threads

```
class ThreadExample extends Thread {
    @Override
    public void run() {
        System.out.println("Testando prioridades");
        System.out.println("Prioridade máxima: " + Thread.MAX_PRIORITY);
        System.out.println("Prioridade mínima: " + Thread.MIN_PRIORITY);
        System.out.println("Prioridade padrão: " + Thread.NORM_PRIORITY);
        System.out.println("Prioridade: " + this.getPriority());
        this.setPriority(7);  //altera a prioridade para 7
        System.out.println("Nova prioridade: " + this.getPriority());
    }
}
```

User Thread e *Daemon Thread*

- Por padrão, as threads em Java são *user threads*.
- A JVM somente finaliza se todas as threads forem *daemon threads*.
- O método `setDaemon(boolean on)` deve ser usado para marcar uma thread como *daemon thread*.
- O método deve ser executado antes do início da thread.

Entregando o processador

- O método **yield()** solicita ao escalonador para liberar a execução da thread solicitante.
- Em outras palavras, a thread está entregando o processador para as threads que estão aptas a usá-lo.
- O escalonador pode ou não aceitar a solicitação.
- Pode ser útil para depurações, testes e implementação de estruturas de controle de concorrência.

Outros métodos

- **getId()**: devolve o identificar da thread (long). É único mas pode ser reusado.
- **getName()**: devolve o nome associado à thread.
- **setName(String name)**: altera o nome associado à thread.
- Depreciados[1]: *Thread.stop*, *Thread.suspend* e *Thread.resume*.

[1] `https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html`

Grupos de threads

- ThreadGroup é uma classe para manipular um conjunto de threads.
- Possibilita aplicar operações no grupo.
- Criando um grupo:

```
ThreadGroup group = new ThreadGroup("MyThreadsGroup")
```

- Adicionando threads ao grupo:

```
Thread(ThreadGroup group, String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
```

- Também é possível construir árvores com grupos de threads:

```
ThreadGroup base = new ThreadGroup("Base");
ThreadGroup group1 = new ThreadGroup(base, "Group1");
ThreadGroup group2 = new ThreadGroup(base, "Group2");
```

Grupos de threads - Métodos ThreadGroup

- **activeCount()**: devolve número estimado de threads ativas em um grupo e seus subgrupos.
- **activeGroupCount()**: devolve número estimado de grupos ativos no grupo e seus subgrupos.
- **destroy()**: destrói o grupo e todos seus subgrupos.
- **enumerate(Thread[] list)**: copia para um vetor as threads ativas do grupo e dos subgrupos.
- **getMaxPriority()**: devolve a prioridade máxima do grupo.
- **interrupt()**: interrompe todas as threads em um grupo.
- **isDaemon()**: testa se o grupo é um *daemon thread group*.
- **setMaxPriority(int priority)**: altera a prioridade máxima do grupo.

Dicas

- Obter o número de processadores:

```
Runtime.getRuntime(). availableProcessors ()
```

- Finalizar a JVM ativa:

```
System.exit(int status)
```

Atividades

1. Faça um programa em Java que consulte periodicamente o estado de um conjunto de threads.
2. Faça um programa em Java para testar um conjunto de operações sobre um grupo de threads. Use o ThreadGroup.
3. Faça um programa em Java com threads que exiba os números primos entre 0 e 100000.
4. Faça um programa em Java que realize uma busca paralela em um vetor de inteiros. Informe para o método: valor procurado, vetor de inteiros e o número de threads.
5. Faça um programa multithreaded em Java que ordene um vetor usando o Merge Sort recursivo. Faça com que a thread principal dispare duas threads para classificar cada metade do vetor.

Referências

- Oracle. **The Java Tutorials - Concurrency**. Disponível em: `https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html`
- CodeJava. **Understanding Thread Group in Java**. Disponível em: `http://www.codejava.net/java-core/concurrency/understanding-thread-group-in-java`