Programação Concorrente

Executando Tarefas

Prof. Rodrigo Campiolo

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

6 de novembro de 2018

Introdução

- O que são tarefas (tasks)?
- Como uma tarefa representa uma unidade de trabalho/processamento a ser executado, nada mais natural que **executar tarefas em threads**.
- Aplicações concorrentes muitas vezes são estruturadas segundo o processamento de tarefas.
- A execução de uma tarefa pode:
  - devolver ou não resultados.
  - ser síncrona ou assíncrona.
  - ser cancelada, agendada ou periódica.
- Como otimizar o uso de threads para processar tarefas?

Runnable vs Callable

- Ambas são interfaces e possibilitam executar tarefas.
- **Runnable** está no pacote **java.lang** e **Callable** no pacote **java.util.concurrent**.
- **Runnable** não retorna um resultado após executar uma tarefa. Deve-se sobrescrever o método **run()**.
- **Callable** retorna um resultado após executar uma tarefa. Deve-se sobrescrever o método **call()**.

Tarefas com Runnable

```
public class Task implements Runnable {
    @Override
    public void run() {
        System.out.println("Running task ...");
        try {
            // do something
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
    }
}
```

Tarefas com Callable

```
public class Task implements Callable<String> {
    @Override
    public String call() {
        String result = "";
        System.out.println("Running task ...");
        try {
            // do something
            Thread.sleep(2000);
            result = "Fake result :-P";
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
        System.out.println("Task done.");
        return result;
    }
}
```

Executando tarefas Callable

- O resultado de um **Callable** deve ser armazenado em **Future**.
- A interface **Future** representa o resultado de uma computação assíncrona.
- **Future** provê métodos para cancelar, checar e obter o resultado de uma computação.
- O método **get()** aguarda o término da computação e devolve o resultado armazenado em **Future**.
- Para executar uma tarefa usando o **Callable** usa-se o framework **Executor**.

Executando tarefas com Callable - Exemplo

```
class MainCallableExample  {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Task task = new Task();
        Future<String> future = executorService.submit(task);

        try {
            System.out.println("Resposta: " + future.get());
        } catch (InterruptedException | ExecutionException e) {
            System.out.println(e);
        }
    }
}
```

Atividades

1. Implemente um programa que calcule o fatorial de um número em uma thread usando o Runnable.
2. Implemente um programa que calcule o fatorial de um número em uma thread usando o Callable.

* em ambos, o resultado deve ser apresentado pela thread principal.

FutureTask

- **FutureTask** representa uma computação assíncrona cancelável.
- Usos:
  - usado pelo framework **Executor** para representar tarefas assíncronas.
  - usado para representar qualquer computação demorada que pode ser iniciada antes da necessidade dos resultados.
- Possui métodos para:
  - iniciar e cancelar uma computação.
  - verificar se a computação está completa.
  - recuperar o resultado da computação.
- Pode ser usado para encapsular objetos **Runnable** e **Callable**.

Classe **FutureTask** - Exemplo

```
public class Preloader {
  private final FutureTask<ProductInfo> future =
    new FutureTask<ProductInfo>(new Callable<ProductInfo>() {
      public ProductInfo call() throws DataLoadException {
          return loadProductInfo();
      }});

  private final Thread thread = new Thread(future);


  public void start() { thread.start(); }

  public ProductInfo get() throws DataLoadException, InterruptedException {
    try {
      return future.get();
    } catch (ExecutionException e) {
      Throwable cause = e.getCause();
      if (cause instanceof DataLoadException)
          throw (DataLoadException) cause;
      else throw launderThrowable(cause);
    }
  }
}
```

Framework **Executor**

- Desvantagens em criar threads sem controle:
  - sobrecarga do ciclo de vida da thread (criação e liberação).
  - consumo de recursos (p. ex. memória).
  - limite de threads que podem ser criadas.
- Framework **Executor**: execução de tarefas assíncronas que suporta gerenciamento e políticas de execução de tarefas.
- A interface **Executor** é a base do framework e desacopla a *submissão da tarefa e a execução da tarefa*.

Políticas de execução

- Em quais threads as tarefas serão executadas?
- Qual a ordem as tarefas devem ser executadas (FIFO, LILO, prioridade)?
- Quantas tarefas podem ser executadas concorrentemente?
- Quantas tarefas podem ser enfileiradas para execução?
- Quais ações devem ser executadas antes e depois de executar uma tarefa?

Pools de threads

- **pool**: termo usado para referir-se a um conjunto de recursos homogêneos mantidos juntos com a finalidade de controle e reuso.
- **thread pool**: gerencia um conjunto de threads de trabalho (*worker threads*).
- Vantagens de usar *pools* de threads:
  - Reuso de threads em vez de criar e liberar threads.
  - Melhora o tempo de resposta durante a solicitação de processamento de tarefas.
  - Possibilita calcular o número de threads concorrentes visando otimizar o uso do processador e memória.
- Java provê diversas implementações de *pools* de thread.

Ciclo de vida Executor

- A implementação da interface *Executor* é usada para criar threads que processam tarefas.
- JVM não finaliza enquanto threads não daemon estiverem em execução.
- Tarefas submetidas via Executor podem estar completadas, executando, esperando por execução na fila.
- A interface **ExecutorService** estende **Executor** para prover métodos para gerenciar o ciclo de vida.

```
public interface ExecutorService extends Executor {
   void shutdown();
   List<Runnable> shutdownNow();
   boolean isShutdown();
   boolean isTerminated();
   boolean awaitTermination(long timeout, TimeUnit unit)
      throws InterruptedException;
   // ... métodos adicionais para submissão de tarefas
}
```

Ciclo de vida Executor

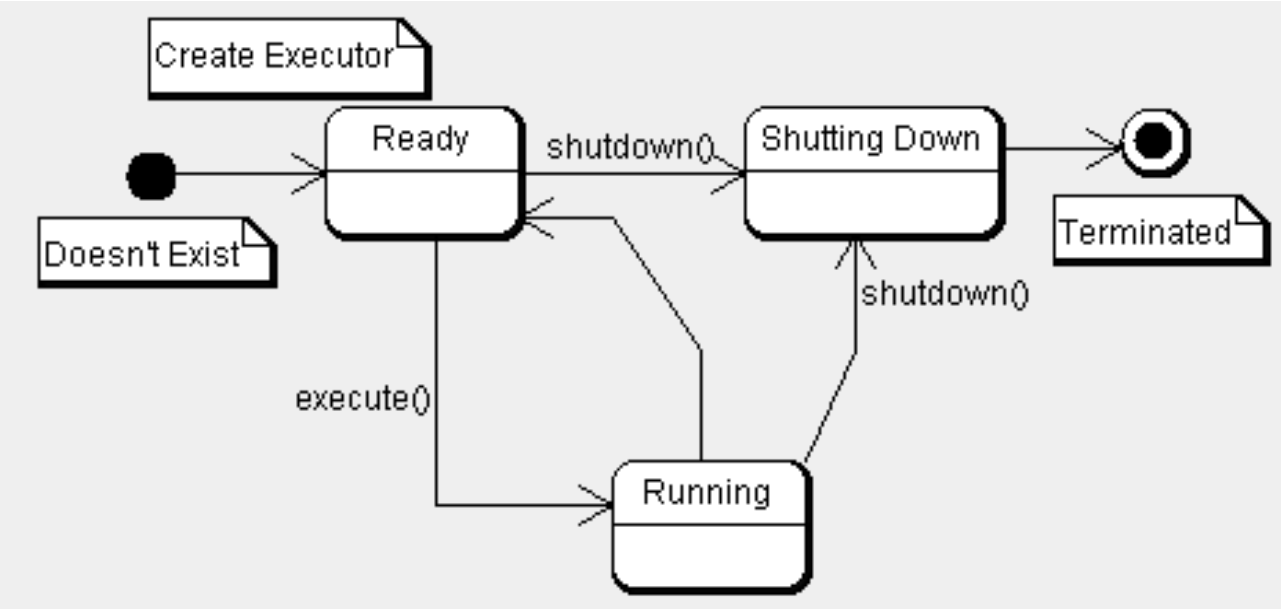

Figura: Ciclo de vida Executor

Criando **ExecutorService**

- A classe **Executors** provê métodos fábrica para **ExecutorService**:
  - **newSingleThreadExecutor**: Cria uma única thread para processar sequencialmente as tarefas em uma ordem especificada (FIFO, LILO, prioridade).
  - **newFixedThreadPool**: Cria um número fixo de threads para processar as tarefas.
  - **newCachedThreadPool**: Cria um pool de threads dinamicamente expansível para atender o aumento na demanda de threads.
  - **newScheduledThreadPool**: Cria um número fixo de threads que suportam adiamento e execução periódica de threads.

Executando tarefas via **ExecutorService**

- **void** execute(Runnable command)

  Executa um comando em algum tempo no futuro.

- <T> Future<T> submit(Callable<T> task)

  Submete uma tarefa Callable para execução e devolve um objeto Future.

- Future<?> submit(Runnable task)

  Submete uma tarefa Runnable para execução e devolve um objeto Future.

- <T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks)

  Executa uma coleção de tarefas, devolvendo uma lista de Future após todas estarem completas.

- <T> T invokeAny(Collection<? **extends** Callable<T>> tasks)

  Executa uma coleção de tarefas, devolvendo o resultado da primeira que finalizar com sucesso.

Exemplo 1: **newSingleThreadExecutor** e **execute**

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

executorService.shutdown();
```

Exemplo 2: **submit** e **Runnable**

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

Future future = executorService.submit(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

future.get();  //returns null if the task has finished correctly.
```

Exemplo 3: **submit** e **Callable**

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

Future future = executorService.submit(new Callable(){
    public Object call() throws Exception {
        System.out.println("Asynchronous Callable");
        return "Callable Result";
    }
});

System.out.println("future.get() = " + future.get());
```

Exemplo 4: **invokeAny**

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 1";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 2";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 3";
    }
});

String result = executorService.invokeAny(callables);
System.out.println("result = " + result);
executorService.shutdown();
```

Exemplo 5: **invokeAll**

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 1";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 2";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 3";
    }
});

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){
    System.out.println("future.get = " + future.get());
}

executorService.shutdown();
```

Exemplo 6: **newFixedThreadPool**

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

public class SimpleThreadPool {

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 10; i++) {
            Runnable worker = new WorkerThread("" + i);
            executor.execute(worker);
          }
        executor.shutdown();
        while (!executor.isTerminated()) {
        }
        System.out.println("Finished all threads");
    }
}
```

```
public class WorkerThread implements Runnable {

    private String command;

    public WorkerThread(String s) {
        this.command = s;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() +
            " Start. Command = " + command);
        processCommand();
        System.out.println(Thread.currentThread().getName() + " End.");
    }

    private void processCommand() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Exemplo 7: **newCachedThreadPool**

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

public class SimpleCachedThreadPool {

    public static void main(String[] args) throws InterruptedException {
        ExecutorService executor = Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++) {
            Runnable worker = new WorkerThread("" + i);
            executor.execute(worker);
            Thread.sleep(750);
          }
        executor.shutdown();
        while (!executor.isTerminated()) {
        }
        System.out.println("Finished all threads");
    }
}
```

Exemplo 8: **newScheduledThreadPool**

```
ScheduledExecutorService scheduledExecutorService =
        Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =
    scheduledExecutorService.schedule(new Callable() {
        public Object call() throws Exception {
            System.out.println("Executed!");
            return "Called!";
        }
    },
    5,
    TimeUnit.SECONDS);

System.out.println("result = " + scheduledFuture.get());

scheduledExecutorService.shutdown();
```

Atividades - Framework **Executor**

1. Faça um programa que localize o maior valor em um vetor. Divida o programa em tarefas que localizam o maior valor em um segmento do vetor. O programa deve possibilitar especificar o número de tarefas e o número de threads para resolver o problema.
2. Faça um programa que calcule a soma dos elementos de uma matriz MxN. Divida o programa em tarefas que somam as linhas. O programa deve possibilitar especificar o número de threads para resolver o problema.
3. Faça um programa concorrente para multiplicar duas matrizes.
4. Faça um programa que periodicamente monitore mudanças em um conjunto de arquivos especificados. Se ocorreram mudanças, o programa deve registrá-las em um arquivo de log.
5. Faça um programa que possibilite agendar uma tarefa para ser executada em um horário específico.
6. Faça um programa que execute três algoritmos de ordenação para um conjunto de valores e exiba o resultado apenas do algoritmo que finalizar primeiro (use *invokeAny*).

Referências

- GOETZ, Brian. **Java concurrency in practice**. Upper Saddle River, NJ.: Addison-Wesley, 2006.
- Jenkov. **Thread Safety and Shared Resources**. Disponível em `http://tutorials.jenkov.com/java-concurrency/thread-safety.html`
- Oracle. **The Java Tutorials - Concurrency**. Disponível em: `https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html`