



Pool de Thread

Aula 6



Arquitetura do Pool de Threads

- Uma das possíveis arquiteturas baseadas em Threads, chama-se **Pool de Threads**.
- Um **Pool de Threads** é criado e gerenciado quando o aplicativo é executado.
- Em JAVA, as threads são mapeadas em nível do sistema e operam com os recursos fornecidos pelo S.O.

Pool de Threads





Pool de Threads

- Não precisa se preocupar com o ciclo de vida das threads
- Se concentra em criar a regra de negócio em vez do gerenciamento
- As interfaces Executor, Executors e ExecutorService gerenciam pools de thread



Pool de Threads

- Executors contém vários métodos para a criar um pool de threads pré configuradas
- É uma boa classe para começar, mas não fornece nenhum ajuste refinado
- As outras 2 interfaces oferecem meios para trabalharmos com diferentes implementações de pool de threads em JAVA
- Devemos manter nosso código desacoplado da implementação real do pool de threads e usar estas interfaces em nossa aplicação

Executor e ExecutorService

- ▶ Tem um único método de execução para enviar instâncias executáveis para a execução
- ▶ Exemplo de como adquirir uma instância Executor apoiada por um único pool de threads com uma fila ilimitada para executar tarefas sequencialmente:

```
Executor executor = Executors.newSingleThreadExecutor();  
executor.execute(() -> System.out.println("Hello World"));
```

- ▶ ExecutorService contém um grande número de métodos para controlar o andamento das tarefas e gerenciar o término do serviço.
- ▶ Usando essa interface, podemos enviar as tarefas para execução e também controlar sua execução usando a instância Future retornada

```
ExecutorService executorService = Executors.newFixedThreadPool(10);  
Future<String> future = executorService.submit(() -> "Hello World");  
// some operations  
String result = future.get();
```



Pool de Threads

- Para usar pools de threads, instancie uma implementação da interface `ExecutorService`
 - `ThreadPoolExecutor`
 - `ScheduledThreadPoolExecutor`
- Essas implementações permitem
 - O número básico e máximo do pool (número de threads)
 - O tipo de estrutura de dados para armazenar as tarefas
 - Como tratar tarefas rejeitadas
 - Como criar e terminar threads



ThreadPoolExecutor

- É uma implementação de pool de threads extensível com muitos parâmetros e ganchos para ajuste fino
- Alguns dos principais
 - `corePoolSize`, `maximumPoolSize` e `keepAliveTime`
- O pool consiste de um número fixo de threads principais que são mantidas o tempo todo.
- Consiste também de algumas threads excessivas que podem ser geradas e depois encerradas quando não são mais necessárias.



ThreadPoolExecutor

- Parâmetro `corePoolSize`
 - Número de threads principais que serão instanciadas e mantidas no pool
- Quando uma nova tarefa chega e todas as threads principais estão ocupadas e a fila interna está cheia, o pool poderá crescer até `maximumPoolSize`
- O parâmetro `keepAliveTime` é o intervalo de tempo durante o qual as threads excessivas podem existir no intervalo inativo
- Por padrão, a classe `ThreadPoolExecutor` considera apenas threads não essenciais para remoção
- Para aplicar a mesma política de remoção aos threads principais
 - `allowCoreThreadTimeOut(true)`



Pool de Threads

- Esses parâmetros cobrem uma ampla gama de casos de uso, mas as configurações mais típicas são predefinidas nos métodos estáticos Executors.
- `newFixedThreadPool(int)`
- `newCachedThreadPool`
- `newSingleThreadExecutor`



newFixedThreadPool(int)

- Cria um ThreadPoolExecutor com valores iguais para os parâmetros corePoolSize e maximumPoolSize e um keepAliveTime igual a zero

```
ThreadPoolExecutor executor =  
    (ThreadPoolExecutor) Executors.newFixedThreadPool(2);  
executor.submit(() -> {  
    Thread.sleep(1000);  
    return null;  
});  
executor.submit(() -> {  
    Thread.sleep(1000);  
    return null;  
});  
executor.submit(() -> {  
    Thread.sleep(1000);  
    return null;  
});  
  
assertEquals(2, executor.getPoolSize());  
assertEquals(1, executor.getQueue().size());
```

newCachedThreadPool()

- Cria outro pool de threads pré configurado
- Este método não recebe um número de threads
- Definimos o corePoolSize como 0 e definimos o maximumPoolSize como Integer.MAX_VALUE.
- keepAliveTime é de 60 segundos

```
ThreadPoolExecutor executor =  
    (ThreadPoolExecutor) Executors.newCachedThreadPool();  
executor.submit(() -> {  
    Thread.sleep(1000);  
    return null;  
});  
executor.submit(() -> {  
    Thread.sleep(1000);  
    return null;  
});  
executor.submit(() -> {  
    Thread.sleep(1000);  
    return null;  
});  
  
assertEquals(3, executor.getPoolSize());  
assertEquals(0, executor.getQueue().size());
```



newSingleThreadExecutor()

- Cria outro pool de threads pré configurado contendo uma única thread
- O executor de thread único é ideal para criar um loop de eventos
- Os parâmetros corePoolSize e maximumPoolSize são iguais a 1 e o keepAliveTime é 0.

```
AtomicInteger counter = new AtomicInteger();

ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
    counter.set(1);
});
executor.submit(() -> {
    counter.compareAndSet(1, 2);
});
```



ScheduledThreadPoolExecutor

- Estende a classe `ThreadPoolExecutor`
- E também implementa a interface `ScheduledExecutor Service` com vários métodos adicionais:
- `schedule`
 - Executa uma tarefa uma vez após um atraso especificado
- `scheduleAtFixedRate`
 - Executa uma tarefa após um atraso inicial especificado e depois executá-la repetidamente com um determinado período
 - O argumento `period` é o tempo medido entre os horários de início das tarefas, portanto, a taxa de execução é fixa



ScheduledThreadPoolExecutor

- `scheduleWithFixedDelay`
 - é semelhante ao `scheduleAtFixedRate`, pois executa repetidamente a tarefa especificada, mas o atraso especificado é medido entre o final da tarefa anterior e o início da próxima
 - A taxa de execução pode variar dependendo do tempo que leva para executar uma determinada tarefa
- Usamos o método `Executors.newScheduledThreadPool()` para criar um `ScheduledThreadPoolExecutor`
 - com um dado `corePoolSize`,
 - `maximumPoolSize` ilimitado
 - `keepAliveTime` zerado



ScheduledThreadPoolExecutor

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(5);
executor.schedule(() -> {
    System.out.println("Hello World");
}, 500, TimeUnit.MILLISECONDS);
```

```
CountDownLatch lock = new CountDownLatch(3);

ScheduledExecutorService executor = Executors.newScheduledThreadPool(5);
ScheduledFuture<?> future = executor.scheduleAtFixedRate(() -> {
    System.out.println("Hello World");
    lock.countDown();
}, 500, 100, TimeUnit.MILLISECONDS);

lock.await(1000, TimeUnit.MILLISECONDS);
future.cancel(true);
```




ForkJoinPool

- Foi introduzido no Java 7 para resolver o problema comum de gerar múltiplas tarefas em algoritmos recursivos
- Ficávamos sem threads rapidamente usando um ThreadPoolExecutor, pois cada tarefa ou subtarefa requer sua própria thread para ser executado
- Em uma estrutura fork/join, qualquer tarefa pode gerar (fork) várias subtarefas e aguardar sua conclusão usando o método join.
- O benefício é que ele não cria uma nova thread para cada tarefa ou subtarefa, em vez disso, implementa o algoritmo de roubo de trabalho



ForkJoinPool - Exemplo


- Percorrer uma árvore de nós e calcular a soma de todos os valores de folha. Sua estrutura de dados:

```
static class TreeNode {  
  
    int value;  
  
    Set<TreeNode> children;  
  
    TreeNode(int value, TreeNode... children) {  
        this.value = value;  
        this.children = Sets.newHashSet(children);  
    }  
}
```



ForkJoinPool - Exemplo

- Agora, se quisermos somar todos os valores em uma árvore em paralelo, precisamos implementar uma interface `RecursiveTask<Integer>`
- Cada tarefa recebe seu próprio nó e adiciona seu valor à soma dos valores de seus filhos.



```
public static class CountingTask extends RecursiveTask<Integer> {

    private final TreeNode node;

    public CountingTask(TreeNode node) {
        this.node = node;
    }

    @Override
    protected Integer compute() {
        return node.value + node.children.stream()
            .map(childNode -> new CountingTask(childNode).fork())
            .collect(Collectors.summingInt(ForkJoinTask::join));
    }
}
```

- Transmite o conjunto de filhos
- mapeia sobre esse fluxo, criando um novo CountingTask para cada elemento
- executa cada subtarefa bifurcando-a (forking)
- coleta os resultados chamando o método join em cada tarefa bifurcada
- soma os resultados usando o coletor Collectors.summingInt



ForkJoinPool - Exemplo

```
TreeNode tree = new TreeNode(5,  
    new TreeNode(3), new TreeNode(2,  
        new TreeNode(2), new TreeNode(8)));  
  
ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();  
int sum = forkJoinPool.invoke(new CountingTask(tree));
```

- O código para executar o cálculo em uma árvore real




Algoritmo de roubo de trabalho

- Os threads de trabalho podem executar apenas uma tarefa por vez, mas o ForkJoinPool não cria uma thread separada para cada subtarefa
- Cada thread no pool tem sua própria fila dupla que armazena as tarefas.
- Simplificando, threads livres tentam “roubar” o trabalho de deque de threads ocupadas
- Por padrão, uma thread de trabalho obtém tarefas da cabeça de seu próprio deque
- Quando está vazio, a thread pega uma tarefa da cauda do deque de outra thread ocupada ou da fila de entrada global



Algoritmo de roubo de trabalho

- Minimiza a possibilidade de que as threads concorram por tarefas
- Reduz o número de vezes que a thread terá que procurar trabalho, pois funciona primeiro nos maiores pedaços de trabalho disponíveis



ForkJoinPool - Instânciação

- Usar o método estático `commonPool()`
- Obterá uma referência ao pool de threads padrão para cada `ForkFoinTask`

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
```

```
public static ForkJoinPool forkJoinPool = new ForkJoinPool(2);
```

- Para acessá-lo

```
ForkJoinPool forkJoinPool = PoolUtil.forkJoinPool;
```




ForkJoinTask<V>

- Tipo base para tarefas executadas dentro do ForkJoinPool
- Na prática, uma de suas duas subclasses deve ser estendida
 - RecursiveAction -> para tarefas void e
 - RecursiveTask<V> -> para tarefas que retornam um valor
- Ambos possuem um método abstrato, responsável pela lógica da tarefa
 - Compute()



```
public class CustomRecursiveAction extends RecursiveAction {

    private String workload = "";
    private static final int THRESHOLD = 4;

    private static Logger logger =
        Logger.getAnonymousLogger();

    public CustomRecursiveAction(String workload) {
        this.workload = workload;
    }

    @Override
    protected void compute() {
        if (workload.length() > THRESHOLD) {
            ForkJoinTask.invokeAll(createSubtasks());
        } else {
            processing(workload);
        }
    }

    private List<CustomRecursiveAction> createSubtasks() {
        List<CustomRecursiveAction> subtasks = new ArrayList<>();

        String partOne = workload.substring(0, workload.length() / 2);
        String partTwo = workload.substring(workload.length() / 2, workload.length());

        subtasks.add(new CustomRecursiveAction(partOne));
        subtasks.add(new CustomRecursiveAction(partTwo));

        return subtasks;
    }

    private void processing(String work) {
        String result = work.toUpperCase();
        logger.info("This result - (" + result + ") - was processed by "
            + Thread.currentThread().getName());
    }
}
```



```
public class CustomRecursiveTask extends RecursiveTask<Integer> {
    private int[] arr;

    private static final int THRESHOLD = 20;

    public CustomRecursiveTask(int[] arr) {
        this.arr = arr;
    }

    @Override
    protected Integer compute() {
        if (arr.length > THRESHOLD) {
            return ForkJoinTask.invokeAll(createSubtasks())
                .stream()
                .mapToInt(ForkJoinTask::join)
                .sum();
        } else {
            return processing(arr);
        }
    }

    private Collection<CustomRecursiveTask> createSubtasks() {
        List<CustomRecursiveTask> dividedTasks = new ArrayList<>();
        dividedTasks.add(new CustomRecursiveTask(
            Arrays.copyOfRange(arr, 0, arr.length / 2)));
        dividedTasks.add(new CustomRecursiveTask(
            Arrays.copyOfRange(arr, arr.length / 2, arr.length)));
        return dividedTasks;
    }

    private Integer processing(int[] arr) {
        return Arrays.stream(arr)
            .filter(a -> a > 10 && a < 27)
            .map(a -> a * 10)
            .sum();
    }
}
```



Enviando Tarefas para o ForkJoinPool

- Submit()
- Execute()

```
forkJoinPool.execute(customRecursiveTask);  
int result = customRecursiveTask.join();
```

- O método invoke() bifurca (fork) a tarefa e aguarda o resultado

```
int result = forkJoinPool.invoke(customRecursiveTask);
```

- O método invokeAll() é a maneira mais conveniente de enviar uma sequência de ForkJoinTasks ao ForkJoinPool
- Ele recebe tarefas como parâmetros, forks e então retorna uma coleção de objetos Future na ordem em que foram produzidos



Enviando Tarefas para o ForkJoinPool

- Você pode também usar os métodos separados
- Fork()
 - Envia uma tarefa para o pool
 - Mas não aciona a sua execução
- Join()
 - Aciona a sua execução
- Em RecursiveAction o join() retorna null
- Em RecursiveTask<v> ele retorna o resultado da execução da tarefa

```
customRecursiveTaskFirst.fork();  
result = customRecursiveTaskLast.join();
```



Enviando Tarefas para o ForkJoinPool

- No exemplo `RecursiveTask<V>`, usamos o método `invokeAll()`
 - Para enviar uma sequência de subtarefas ao pool
- Poderíamos ter feito a mesma coisa com `fork()` e `join()`
- Embora tínhamos que tratar a ordenação dos resultados
- Para evitar confusão, utilize o `invokeAll()` para enviar mais de uma tarefa para o `ForkJoinPool`



Observações do uso do Fork/Join

- Pode acelerar o processamento de grandes tarefas, desde que:
- Use o menor número possível de pools de threads
 - Na maioria dos casos, a melhor decisão é usar um pool de threads por aplicativo
- Use o pool de threads comuns padrão
 - Se nenhum ajuste específico for necessário
- Use um limite razoável para dividir ForkJoinTask em subtarefas
- Evite qualquer bloqueio em seu ForkJoinTasks



Guava

- Biblioteca popular de utilitários do Google
- Tem muitas classes úteis, incluindo várias implementações úteis de `ExecutorService`
- As classes de implementação não são acessíveis para instânciação direta ou subclasses
- O único ponto de entrada para criar suas instâncias é a classe auxiliar `MoreExecutors`



Add Guava

- Devemos adicionar a dependência ao arquivo Maven para incluir a biblioteca Guava ao projeto
 - Pom.xml

```
<dependency>  
  <groupId>com.google.guava</groupId>  
  <artifactId>guava</artifactId>  
  <version>31.0.1-jre</version>  
</dependency>
```



Direct Executor e Direct Executor Service

- Às vezes, queremos executar a tarefa no thread atual ou em um pool de threads, dependendo de algumas condições.
- Preferimos usar uma única interface Executor e apenas alternar a implementação.
- Ainda assim temos que escrever um código clichê
- Felizmente Guava fornece instâncias predefinidas

Direct Executor e Direct Executor Service

```
Executor executor = MoreExecutors.directExecutor();

AtomicBoolean executed = new AtomicBoolean();

executor.execute(() -> {
    try {
        Thread.sleep(500);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    executed.set(true);
});

assertTrue(executed.get());
```

- Devemos preferir esse método ao invés do `MoreExecutors.newDirectExecutorService()`
 - Pois ele cria uma implementação de serviço de executor completo em cada chamada




Terminando o Executor Services

- Desligar a máquina virtual enquanto um pool de threads ainda está executando suas tarefas é um sério problema
- Mesmo com um mecanismo de cancelamento implementado, não há garantia de que as tarefas se comportarão bem e interromperão seu trabalho quando o serviço do executor for encerrado
- Isso pode fazer com que a JVM seja interrompida enquanto que as tarefas ainda continuam fazendo o seu trabalho.



Terminando o Executor Services

- Guava apresenta uma série de serviços executores de saída
- Esses serviços também adicionam um gancho de desligamento com o método `Runtime.getRuntime().addShutdownHook()`
 - Impedindo que JVM seja encerrada, por um período de tempo, antes de encerrar as tarefas interrompidas



Terminando o Executor Services - Exemplo

```
ThreadPoolExecutor executor =  
    (ThreadPoolExecutor) Executors.newFixedThreadPool(5);  
ExecutorService executorService =  
    MoreExecutors.getExitingExecutorService(executor,  
        100, TimeUnit.MILLISECONDS);  
  
executorService.submit(() -> {  
    while (true) {  
    }  
});
```



Listening Decorators

- Permite encapsular o `ExecutorService` e receber instâncias de `ListenableFuture` no envio da tarefa, em vez de instâncias de `Future` simples
- A interface `ListenableFuture` estende `Future` e possui um único método adicional
 - `addListener()`
 - Permite adicionar um listener que é chamado após a conclusão
- É essencial para a maioria dos métodos auxiliares da classe de utilitários `Futures`



Listening Decorators - Exemplo

```
ExecutorService executorService = Executors.newCachedThreadPool();
ListeningExecutorService listeningExecutorService =
    MoreExecutors.listeningDecorator(executorService);

ListenableFuture<String> future1 =
    listeningExecutorService.submit(() -> "Hello");
ListenableFuture<String> future2 =
    listeningExecutorService.submit(() -> "World");

String greeting = Futures.allAsList(future1, future2).get()
    .stream()
    .collect(Collectors.joining(" "));
assertEquals("Hello World", greeting);
```




Pool Threads

- A seguir está o exemplo `WorkerThread1` que será executado por uma thread
- A tarefa faz algo e periodicamente informa o percentual do trabalho realizado
- Na classe `ThreadPoolTest` especificamos o número de tarefas a criar e o tamanho do pool de threads usados para executar as tarefas
- O exemplo usa um número fixo de threads e com menos threads do que tarefas



Pool Thread

- Observe como as tarefas 0 e 1 foram alocadas para as duas threads e rodam até terminar
- Só depois que uma termina é que a próxima tarefa pode ser alocada a thread e iniciar



Escalonamento de Atividades

- `java.util.concurrent`
- **Interface**
 - `ScheduledExecutorService`
- **Subinterfaces:**
 - `Executor, ExecutorService`
- **Implementando Classes:**
 - `ScheduledThreadPoolExecutor`



Escalonamento de Atividades

- Um **ScheduledExecutorService** permite escalonar a **execução de atividades**
- **SingleThreadScheduledExecutor**
- **ScheduledThreadPool**



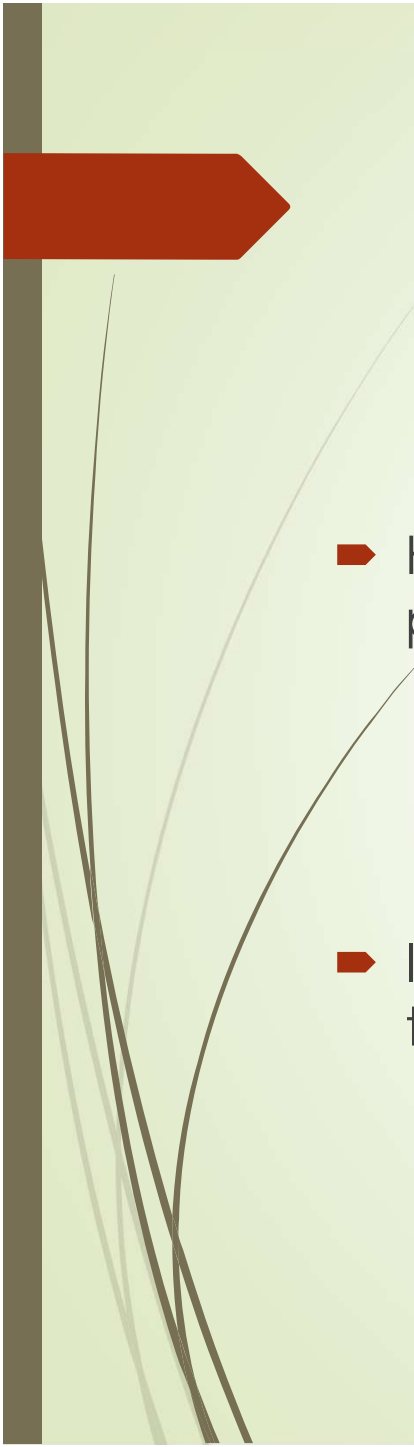
Escalonamento de Tarefas

- CountdownThread

- Exemplo de escalonamento dado um tempo utilizando uma única thread no pool de threads


- CountdownThread2

- Exemplo de escalonamento com periodicidade



ScheduledThreadPoolExecut or

- Há duas formas de executar threads com periodicidade
 - Timer
 - ScheduledThreadPoolExecutor
- Idem ao anterior para receber o executor do pool de threads
 - `ScheduledThreadPoolExecutor sch =
(ScheduledThreadPoolExecutor)
Executors.newScheduledThreadPool(5);`



ScheduledThreadPoolExecut or

- `schedule()`
- `scheduleAtFixedRate()`
- `scheduleWithFixedDelay()`



Classe Timer e TimeTask

- Funcionamento parecido com a anterior
- Pertence ao pacote `java.util.Timer`
- Escalonam instâncias da classe `TimeTask`



Exemplo

- O programa ilustra o uso de um Time Thread para executar uma tarefa
- Implementamos uma subclasse de TimerTask e especificamos o que fazer no método run()
- Escalonamos a execução da tarefa com o método schedule()
- Terminamos a thread por meio do cancel()
- É possível também por meio do System.exit(0)



Exemplo

- Pegar a data correspondente a 23 horas de hoje

```
Calendar ca = Calendar.getInstance();
```

```
ca.set(Calendar.HOUR_OF_DAY, 23);
```

```
ca.set(Calendar.MINUTE, 1);
```

```
ca.set(Calendar.SECOND, 0);
```

```
date time = ca.getTime();
```

```
Timer = new Timer
```

```
//Executa essa thread nessa data específica
```

```
Timer.schedelu(new RemindTask(), time);
```



Execução Repetida de uma tarefa

- Utiliza os mesmo métodos que aprendemos anteriormente
 - `schedule(TimerTask task, long delay, long period);`
 - `schedule(TimerTask task, Date time, long period);`
 - `scheduleAtFixedRate(TimerTask task, long delay, long period);`
 - `scheduleAtFixedRate(TimerTask task, Date firstTime, long period);`