Programação Concorrente

java.util.concurrent Package

COMPUTAÇÃO UFCG

Universidade Federal de Campina Grande

java.util.concurrent Package

Disponibiliza classes utilitárias úteis para desenvolvimento em programação concorrente. Este pacote inclui algumas pequenas estruturas extensíveis padronizadas, bem como algumas classes que fornecem funcionalidade útil e são, de outra forma, tediosas ou difíceis de implementar. Aqui estão breves descrições dos principais componentes.

Simplificando o Gerenciamento de Threads

ExecutorService

- Um objeto que executa tarefas enviadas
- Fornece uma maneira de desacoplar o envio de tarefas da mecânica de como cada tarefa será executada
- normalmente usado em vez de criar threads explicitamente

```
RunnableTask implements Runnable {
    public void run() {
        System.out.println("Asynchronous task");
    }
}

new Thread(new RunnableTask()).start()
new Thread(new RunnableTask()).start()
new Thread(new RunnableTask()).start()
```

```
ExecutorService executorService =
Executors.newFixedThreadPool(10);

executorService.execute(new RunnableTask());
executorService.execute(new RunnableTask());
executorService.execute(new RunnableTask());

executorService.shutdown();
```

Criando um ExecutorService

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

Cria um Executor que usa uma única thread operando em uma fila ilimitada. Se esse thread única terminar devido a uma falha durante a execução antes do desligamento, uma nova tomará seu lugar, se necessário, para executar tarefas subsequentes.

Criando um ExecutorService

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

Cria um Executor que cria novas threads conforme necessário, mas reutilizará threads construídas anteriormente quando estiverem disponíveis.

Criando um ExecutorService

```
ExecutorService executorService = Executors.newFixedThreadPool(10);
```

Cria um pool de threads que utiliza um número fixo de threads operando em uma fila ilimitada compartilhada. A qualquer momento, no máximo n threads estarão ativas processando tarefas. Se tarefas adicionais forem enviadas quando todas as threads estiverem ativas, elas esperam na fila até que uma thread esteja disponível.

Delegando uma tarefa para um ExecutorService

- `void execute(Runnable)`
- `Future<?> submit(Runnable)`
- `Future<T> submit(Callable<T>)`

Qual a diferença entre Runnable e Callable?

Delegando uma tarefa para um ExecutorService

- `void execute(Runnable)`
- `Future<?> submit(Runnable)`
- `Future<T> submit(Callable<T>)`

Qual a diferença entre Runnable e Callable?

```
Runnable runnableTask = () -> {
    try {
        TimeUnit.MILLISECONDS.sleep(300)
    ;
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
};
```

```
Callable<String> tarefa = () -> {
    Thread.sleep(300);
    return "Resultado da tarefa";
};
```

Future

Um Future representa o resultado de uma computação assíncrona. O resultado só pode ser recuperado usando o método get quando a computação for concluída, bloqueando se necessário até que esteja pronto

Future

Um Future representa o resultado de uma computação assíncrona. O resultado só pode ser recuperado usando o método get quando a computação for concluída, bloqueando se necessário até que esteja pronto

```
Future<String> resultado = executor.submit(tarefa);

System.out.println("Aguardando resultado...");
System.out.println("Resultado: " + resultado.get());
```

Criando um ScheduledExecutorService

```
ExecutorService executorService = Executors.newSingleThreadScheduledExecutor();
ExecutorService executorService2 = Executors.newScheduledThreadPool(10);
```

O ScheduledExecutorService é um pool de threads que pode agendar tarefas para serem executadas após um determinado atraso ou para serem executadas periodicamente.

Delegando uma tarefa para um ScheduledExecutorService

ScheduledFuture<?> schedule(Runnable, delay, TimeUnit)

- Envia um runnable que é habilitado após o atraso especificado.

ScheduledFuture<T> schedule(Callable<T>, delay, TimeUnit)

- Envia um callable que é habilitado após o atraso especificado.

ScheduledFuture<?> scheduleAtFixedRate(Runnable, delay, period, TimeUnit)

- Envia um runnable que é habilitado primeiro após o atraso inicial fornecido e, posteriormente, com o período fornecido

ScheduledFuture<?> scheduleAtFixedDelay(Runnable, delay, period, TimeUnit)

- Envia um runnable que é habilitado primeiro após o atraso inicial fornecido e, posteriormente, com o atraso fornecido entre o término de uma execução e o início da próxima.

ScheduledExecutorService

```
ScheduledExecutorService executorService =
Executors.newSingleThreadScheduledExecutor();

Future<String> future = executorService.schedule(() -> {
    return "Hello world";
}, 1, TimeUnit.SECONDS);

executorService.scheduleAtFixedRate(() -> {
    // ...
}, 1, 10, TimeUnit.SECONDS);

executorService.scheduleWithFixedDelay(() -> {
    // ...
}, 1, 10, TimeUnit.SECONDS);

executorService.shutdown();
```

Finalizando um ExecutorService

`shutdown()`

- Ele espera até que a execução de todas as tarefas enviadas seja concluída.

`shutdownNow()`

- Ele imediatamente encerra todas as tarefas em execução/pendentes.

Bloqueando até finalizar o ExecutorService

`awaitTermination(timeout, TimeUnit)`

- Ele bloqueia até que todas as tasks completem suas execuções após uma chamada ao shutdown, ou até que ocorra o timeout especificado.

Estruturas de Dados Thread-safe

BlockingQueue

A interface BlockingQueue suporta controle de fluxo (além da fila) introduzindo bloqueio se a estrutura estiver cheia ou vazia. Uma thread tentando enfileirar um elemento em uma fila cheia é bloqueada até que alguma outra thread libere espaço na fila. Da mesma forma, ela bloqueia uma thread tentando excluir de uma fila vazia até que alguma outra thread insira um item. BlockingQueue não aceita um valor nulo.

BlockingQueue

- **Unbounded queue** – pode crescer quase indefinidamente
  - capacidade máxima é Integer.MAX_VALUE)

```
BlockingQueue<String> blockingQueue = new
LinkedBlockingDeque<>();
```

- **Bounded queue** – com capacidade máxima definida

```
BlockingQueue<String> blockingQueue = new
LinkedBlockingDeque<>(10);
```

Blocking Queue API

Adicionando elementos

- **add(e)** – retorna verdadeiro se a inserção foi bem-sucedida, caso contrário, lança uma IllegalStateException
- **put(e)** – insere o elemento especificado em uma fila, aguardando por um slot livre, se necessário
- **offer(e)** – retorna verdadeiro se a inserção foi bem-sucedida, caso contrário, falso
- **offer(e, timeout, TimeUnit)** – tenta inserir o elemento em uma fila e aguarda por um slot disponível dentro de um tempo limite especificado

Blocking Queue API

Removendo elementos

- **take() –** espera por um elemento head de uma fila e o remove. Se a fila estiver vazia, ele bloqueia e espera por um elemento para ficar disponível
- **poll(timeout, TimeUnit)** – recupera e remove o head da fila, esperando até o tempo de espera especificado, se necessário, para que um elemento fique disponível. Retorna null após um timeout

Algumas Implementações para BlockingQueue

- ArrayBlockingQueue
- PriorityBlockingQueue
- DelayQueue
- LinkedBlockingDeque
- LinkedBlockingQueue
- LinkedTransferQueue
- SynchronousQueue

Concurrent Collections

Além de Queues, o pacote também fornece implementações de Collection projetadas para uso em contextos multithread:

- ConcurrentHashMap
- ConcurrentSkipListMap
- ConcurrentSkipListSet
- CopyOnWriteArrayList
- CopyOnWriteArraySet

Concurrent Collections

Quando muitos threads são esperados para acessar uma determinada coleção:

- **ConcurrentHashMap** é preferível a um HashMap sincronizado
- **ConcurrentSkipListMap** é preferível a um TreeMap sincronizado

Concurrent Collections

Um **CopyOnWriteArrayList** é preferível a um ArrayList sincronizado quando o número esperado de leituras supera em muito o número de atualizações para uma lista