Programação Concorrente

Coleções Concorrentes

Prof. Rodrigo Campiolo

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

26 de junho de 2019

Introdução

- Coleções concorrentes proveem estruturas para manipular coleções que suportam acessos concorrentes.
- Antes do Java 5, já haviam coleções sincronizadas para garantir thread-safety, no entanto, a implementação provia baixa concorrência por serializar todos os acessos ao estado das coleções.

```
List<Integer> syncList = Collections.synchronizedList(new ArrayList<>());
Map<Integer, String> syncMap = Collections.synchronizedMap(new HashMap<>());
```

- Neste tópico, abordaremos as coleções concorrentes do pacote **java.util.concurrent**.

BlockingQueue

Uma interface para fila que possui operações que bloqueiam ao aguardar e/ou armazenar por elementos. Possui cinco implementações:

- **ArrayBlockingQueue**: uma fila bloqueante que armazena os elementos em *array*.
- **LinkedBlockingQueue**:uma fila bloqueante que armazena os elementos em lista ligada.
- **SynchronousQueue**: uma fila bloqueante em que cada operação de inserção deve aguardar por uma de remoção e vice-versa.
- **PriorityBlockingQueue**: uma fila bloqueante que ordena os elementos por prioridade.
- **DelayQueue**: uma fila bloqueante em que os elementos só podem ser removidos após um temporizador.

BlockingQueue

- Não aceita elementos nulos.
- Usadas comumente para aplicações com produtores e consumidores.
- As operações são *thread-safe*.
- Não suporta operação para finalização (p. ex. *close*) para dizer que não serão adicionados novos elementos.
- Pode ter sua capacidade de elementos limitada.

BlockingQueue - Métodos

- Documentação: `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html`.
- Resumo das operações básicas:

	Exceção	Retorno	Bloqueio	Time out
Inserir	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remover	remove()	poll()	take()	poll(time, unit)
Inspecionar	element()	peek()	-	-

- `int drainTo(Collection<? super E> c)`
  Remove todos os elementos e adiciona em uma coleção.
- `int drainTo(Collection<? super E> c, int maxElements)`
  Remove até maxElements e adiciona em uma coleção.

BlockingDeque

Uma interface de deque que possui operações que bloqueiam ao aguardar e/ou armazenar elementos.

- **LinkedBlockingDeque**: um deque bloqueante que armazena elementos em lista ligada.

TransferQueue

Uma interface **BlockingQueue** que aguarda por consumidores para receber elementos.

- **LinkedTransferQueue**: implementa **TransferQueue** em lista ligada.

BlockingDeque - Métodos

- Documentação: `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html`.
- Métodos similares a *BlockingQueue*.
- Métodos bloqueantes para o início do deque:
  - `putFirst(e)`: insere elemento no início do deque.
  - `takeFirst()`: devolve e remove elemento do início do deque.
- Métodos bloqueantes para o final do deque:
  - `putLast(e)`: insere elemento no final do deque.
  - `takeLast()`: devolve e remove elemento do final do deque.

TransferQueue - Métodos

- Documentação: `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html`.
- `int getWaitingConsumerCount()`
  Devolve o número estimado de consumidores.
- `boolean hasWaitingConsumer()`
  Devolve true se há ao menos um consumidor aguardando elemento.
- `void transfer(E e)`
  Transfere elemento para consumidor, esperando se necessário.
- `boolean tryTransfer(E e)`
  Transfere elemento imediatamente, se possível.
- `tryTransfer(E e, long timeout, TimeUnit unit)`
  Transfere elemento imediatamente ou até expirar o *timeout*.

Atividades

1. Implemente duas versões do problema do produtor/consumidor com M produtores e N consumidores usando **ArrayBlockingQueue** e **LinkedBlockingQueue**. Compare o desempenho das duas implementações.
2. Implemente o problema do produtor/consumidor para uma estrutura com os seguintes campos: número, simbolo, naipe, que representa uma carta de baralho. A implementação deve possibilitar que após 10 cartas produzidas por dois produtores, outros dois consumidores pegarão somente as maiores cartas. Os produtores somente devem produzir mais cartas após os consumidores removerem 3 cartas cada um. As cartas remanescentes podem continuar na estrutura. Use a ordenação do baralho da menor para maior: A, 2, ..., 10, Q, J, K.

ConcurrentLinkedQueue

- Implementação de fila concorrente baseada em algoritmos não bloqueantes.
- Baseado em `http://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf`.
- *Iterators* são fracamente consistentes, ou seja, refletem o estado da fila em um ponto ou após a criação do *iterator*.
- A atomicidade das operações que atuam em um conjunto de elementos não é garantida.
- A consistência de memória é garantida pela relação *happen-before*.

ConcurrentLinkedQueue - Métodos

- `boolean add(E e)` e `boolean offer(E e)`
  Insere um elemento no final da fila.
- `E poll()`
  Devolve e remove o elemento do início da fila.
- `E peek()`
  Devolve o elemento no início da fila.
- `boolean isEmpty()`
  Verifica se a fila está vazia.

ConcurrentMap

- `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html`.
- A interface para estruturas de mapa que provê *thread safety* e garantias de atomicidade.
- A class ConcurrentHashMap implementa essa interface.
- Documentação: `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html`

ConcurrentMap - Métodos

- `default V getOrDefault(Object key, V defaultValue)`
  Devolve o valor associado a chave ou um valor padrão.
- `V putIfAbsent(K key, V value)`
  Adiciona um elemento ao mapa se a chave não está presente.
- `boolean remove(Object key, Object value)`
  Remove um elemento que correspondente a chave e valor.
- `V replace(K key, V value)`
  Substitui um elemento se a chave está presente.

ConcurrentMap - Outros métodos

- `default V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)`
  Executa uma função sobre o mapeamento com chave *key*.
  Exemplo: map.compute(key, (k, v) −> (v == **null**) ? msg : v.concat(msg))
- `default void forEach(BiConsumer<? super K,? super V> action)`
  Executa uma ação específica sobre cada elemento.
  Exemplo: map.forEach((k, v) −> System.out.println(k + ","+ v))
- `default V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)`.
  Executa uma função sobre o mapeamento com chave key, caso contrário, armazena (key, value).
  Exemplo: map.merge(key, msg, (oldValue, newValue) −> oldValue.concat(newValue))
  //adiciona (key, msg) ou oldValue+newValue (newValue = msg)

ConcurrentMap - Outros métodos

- `<U> U search(long parallelismThreshold, BiFunction<? super K,? super V,? extends U> searchFunction)`.
  Realiza uma busca especificada via função e devolve a primeira ocorrência encontrada. Se nenhuma, devolve null.
  Exemplo: `map.search(1, (k, v) -> { return v.size() > 10 ? return k : null ;);`
- `<U> U reduce(long parallelismThreshold, BiFunction<? super K,? super V,? extends U> transformer, BiFunction<? super U,? super U,? extends U> reducer)`
  Devolve o resultado de acumular a transformação dos pares (key, value) usando um redutor para combinar o valor. Se nenhum, devolve null.
  Exemplo: `map.reduce(4, (k, v) -> v.size(), (total, elem) -> total + elem);`

CopyOnWriteArrayList

- `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArrayList.html`.
- Uma estrutura de *ArrayList* em que as operações que resultam em alterações (add, set, outros) são implementados por meio de uma cópia interna do array.
- **Iterator** usa uma instância do momento de sua criação.

Atividades

1. Faça um programa usando *Threads* e *ConcurrentMap* para calcular a frequência de cada letra em um texto.

Referências

- GOETZ, Brian. **Java concurrency in practice**. Upper Saddle River, NJ.: Addison-Wesley, 2006.
- Oracle. **The Java Tutorials - Concurrency**. Disponível em: `https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html`
- Jenkov. **Non-blocking algorithms**. Disponível em `http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html`