Collections & Pipelines (Streams) em Java — Guia Avançado (nível sênior)

Autor: ChatGPT — Conteúdo técnico e exemplos

Data: 30/09/2025

Sumário

1. Introdução

Este documento explica com profundidade o Java Collections Framework e o Streams API — o modelo de pipelines funcionais introduzido no Java 8. Destina-se a programadores sêniores que precisam dominar não apenas o 'como' mas o 'porquê' e 'quando' usar certas estruturas e operações para escrever código eficiente, correto e escalável.

2. Visão geral do Java Collections Framework (JCF)

O JCF define interfaces genéricas (Collection, List, Set, Queue, Deque, Map) e várias implementações. Pontos chave que um sênior deve dominar:

- Diferença entre interfaces e implementações (contrato vs. comportamento).

- Caraterísticas importantes: ordenação, unicidade, navegação, eficiência de acesso/inserção.

- Coleções especializadas: EnumSet, EnumMap, WeakHashMap, IdentityHashMap, ConcurrentHashMap, CopyOnWriteArrayList.

3. Principais implementações e quando usar

ArrayList — Lista dinâmica com acesso O(1) por índice; amostragem melhor para leitura e iteração rápida; custo de inserção/remover no meio O(n).

LinkedList — Bom para inserções frequentes no meio ou nas extremidades; implementa Deque; acesso por índice O(n).

HashSet / HashMap — Implementações baseadas em hashing; operações básicas O(1) amortizado; sensível a equals/hashCode.

LinkedHashSet / LinkedHashMap — Mantém ordem de inserção (ou acesso, se configurado), útil para caches LRU.

TreeSet / TreeMap — Implementações ordenadas (red■black tree); operações O(log n); útil quando ordenação natural ou comparator é requerida.

PriorityQueue — Fila baseada em heap (ordem natural ou comparator).

ArrayDeque — Deque eficiente; prefira a ArrayDeque para pilhas/filas ao invés de Stack ou LinkedList em muitos casos.

ConcurrentHashMap — Map concorrente altamente escalável; não bloqueia globalmente para operações comuns.

CopyOnWriteArrayList — Perfeito para leituras muito mais frequentes que escritas; cada mutação cria uma cópia.

WeakHashMap — Chaves mantidas com referências fracas — útil para caches que não devem impedir GC.

4. Métodos essenciais de Collection e Map

Collection (List/Set/Queue):

- add(E e), addAll(Collection c)
- remove(Object o), removeIf(Predicate filter)
- contains(Object o), containsAll(Collection c)
- size(), isEmpty()
- iterator(), forEach(Consumer action)
- toArray(), stream(), parallelStream()
- clear(), retainAll(Collection c), removeAll(Collection c)

Map — métodos avançados que todo sênior deve dominar:

- get(Object key), put(K key, V value), remove(Object key)
- containsKey(Object key), containsValue(Object value), keySet(), values(), entrySet()
- getOrDefault(key, defaultValue) — evita checagens nulas
- putIfAbsent(key, value) — atomically put if absent
- computeIfAbsent(key, mappingFunction) — inicialização lazy segura
- computeIfPresent(key, remappingFunction)
- compute(key, remappingFunction) — para lógica completa atômica
- merge(key, value, remappingFunction) — útil para acumular valores
- replaceAll(BiFunction) — aplicar transformação para todos os valores
- forEach(BiConsumer) — iteração eficiente sobre entries

Exemplo: computeIfAbsent padrão para agrupar:

```
Map<Key, List<Value>> map = new HashMap<>();
for (Item it : items) {
    map.computeIfAbsent(it.getKey(), k -> new ArrayList<>()).add(it.getValue());
}
```

5. Streams e Pipelines — conceitos

Stream é uma sequência de elementos com operações de pipeline: criação → zero ou mais operações intermediárias (lazy) → operação terminal (eager). Pontos chave:

- Streams não armazenam dados; são 'views' sobre a fonte.

- Operações intermediárias são lazily avaliadas; nada executa até a operação terminal.

- Preferir expressões sem efeitos colaterais; evitar mutação durante pipeline.

- Streams podem ser paralelos, mas isso exige cuidado com a fonte e operações.

6. Operações intermediárias (mais usadas)

- **filter(Predicate)**: filtra elementos

- **map(Function)**: transforma cada elemento

- **flatMap(Function>)**: achata streams aninhados

- **distinct()**: remove duplicatas (usa equals())

- **sorted()**: ordena (natural ou comparator) — pode ser custoso

- **peek(Consumer)**: para debug; execute efeitos colaterais leves

- **limit(long maxSize), skip(long n)**: controla elementos (útil para streams infinitos)

- **takeWhile / dropWhile (Java 9+)**: operadores baseados em predicado que respeitam ordenação

- **mapToInt / mapToLong / mapToDouble**: conversão para streams primitivos (evitar boxing)

- **boxed()**: converte primitive stream para Stream

Exemplo: flatMap vs map

```
List<List<String>> listas = List.of(List.of("a","b"), List.of("c"));
List<String> plano = listas.stream()
    .flatMap(List::stream)
    .collect(Collectors.toList());
```

7. Operações terminais (mais usadas)

- **forEach(Consumer)**: itera; use forEachOrdered se a ordem for importante em paralelo

- **toArray()**: materializa em array

- **reduce(BinaryOperator) / reduce(identity, accumulator, combiner)**: redução imutável ou mutável

- **collect(Collector)**: redução mutável — use Collectors

- **min / max (Comparator)**: obter extremo

- **count()**: retorna long

- **anyMatch / allMatch / noneMatch**: testes booleanos, potencialmente short-circuit

- **findFirst / findAny**: encontrar um elemento — findAny em paralelo é mais eficiente se ordem não importa

- **summaryStatistics via mapToInt(...).summaryStatistics()**: estatísticas numéricas

8. Collectors e redução mutável

Collectors encapsulam estratégias eficientes de redução mutável. Principais collectors:

- Collectors.toList(), toSet(), toCollection(Supplier)

- Collectors.toMap(keyMapper, valueMapper, mergeFunction, mapSupplier)

- Collectors.joining(delimiter, prefix, suffix) — para Strings

- Collectors.counting(), averagingInt/Long/Double, summingInt/Long/Double

- Collectors.summarizingInt/Long/Double — retorna IntSummaryStatistics etc.

- Collectors.groupingBy(classifier), groupingBy(classifier, downstreamCollector)

- Collectors.partitioningBy(predicate) — map boolean → lista/collector

- Collectors.mapping(mapper, downstream)

- Collectors.collectingAndThen(downstream, finisher) — pós-processamento

Importante: características do Collector (Collector.Characteristics):

- UNORDERED: resultado independente da ordem. - CONCURRENT: pode acumular resultados de forma concorrente. - IDENTITY_FINISH: a função finisher é identidade (ótimo para performance).

Exemplo: multi-nível groupingBy com downstream

```
Map<Category, Map<Status, Long>> stats = items.stream()
    .collect(Collectors.groupingBy(Item::getCategory,
           Collectors.groupingBy(Item::getStatus, Collectors.counting())));
```

9. Streams primitivos e Optional

Use IntStream/LongStream/DoubleStream para evitar boxing/unboxing. Exemplo de uso e obtenção de estatísticas:

```
IntSummaryStatistics st = items.stream()
    .mapToInt(Item::getValue)
    .summaryStatistics();
```

Optional — evite retornar nulls; métodos úteis: isPresent/get, orElse, orElseGet, orElseThrow, map, flatMap, ifPresent.

10. Paralelismo com streams e Spliterator

Paralelismo usa ForkJoinPool.commonPool() e Spliterator.trySplit() para dividir a fonte. Pontos avançados:

- Nem todo source se paraleliza bem: ArrayList e arrays são ótimos; LinkedList não (costly split).

- Spliterator tem characteristics: SIZED, SUBSIZED, ORDERED, SORTED, IMMUTABLE, CONCURRENT.

- Use streams paralelos quando operações são CPU-bound, sem efeitos colaterais e a fonte é splittable eficientemente.

- Para controlar paralelismo pode-se usar ForkJoinPool personalizado: ForkJoinPool.commonPool alterável via system property ou submeter tarefa ao pool.

- Evite usar parallelStream() em APIs que fazem I/O, bloqueios, ou código que usa ThreadLocal / frameworks que dependem da thread.

Exemplo: executar stream paralelo em pool customizado

```
ForkJoinPool pool = new ForkJoinPool(20);
try {
    pool.submit(() -> list.parallelStream().map(...).collect(...)).get();
} finally {
    pool.shutdown();
}
```

11. Map avançado: compute / merge / putIfAbsent

Esses métodos permitem atualizações atômicas e expressivas sem usar sincronização externa.

Exemplos práticos:

```
// Acumular contagens
map.merge(key, 1, Integer::sum);

// Inicializar lista e adicionar (concorrente-safe quando usar ConcurrentHashMap)
map.computeIfAbsent(key, k -> new ArrayList<>()).add(value);

// Atualizar valor com lógica complexa
map.compute(key, (k, v) -> (v == null) ? initialValue : update(v));
```

12. Concorrência e coleções concorrentes

Conhecimentos importantes:

- Collections.synchronizedList(..) fornece sincronização por wrapper — ainda precisa sincronizar ao iterar.

- CopyOnWriteArrayList: ideal para leituras intensivas; gravações custosas.

- ConcurrentHashMap: uso para alta concorrência; métodos atomics (compute, merge) são thread-safe.

- BlockingQueue (ArrayBlockingQueue / LinkedBlockingQueue / PriorityBlockingQueue) para produtor/consumidor.

- ConcurrentSkipListMap / ConcurrentSkipListSet oferecem ordenação concorrente.

13. Boas práticas e armadilhas comuns

- **Evite mutação compartilhada em pipelines**: Mutação compartilhada leva a condições de corrida e resultados não determinísticos em parallel streams.

- **Use entrySet() ao iterar Map quando precisar de key+value**: Evita lookup adicional via get().

- **Prefira coleções imutáveis quando possível**: List.of(), Map.of(), Collectors.toUnmodifiableList() reduzem bugs.

- **Cuidado com equals/hashCode de chaves**: Uma chave mutável quebra a integridade do Map.

- **Evite Stream para lógica com muitos efeitos colaterais**: Streams são melhores para transformações puras.

- **Atenção com toList() em Java 8 vs toList() em Java 16**: Collectors.toList() em Java 8 pode retornar ArrayList mutável; Java 16+ tem List.of estilo imutável em alguns métodos.

14. Patterns e exemplos reais

Agrupamento e agregação (groupingBy + downstream) — padrão de relatório:

```
Map<String, Double> somaPorCategoria = items.stream()
    .collect(Collectors.groupingBy(Item::getCategory, Collectors.summingDouble(Item::getAmount)));
```

Converter Map para outra estrutura:

```
List<Result> results = map.entrySet().stream()
    .map(e -> new Result(e.getKey(), e.getValue()))
    .collect(Collectors.toList());
```

15. Cheatsheet: métodos mais importantes (resumo rápido)

- Collection.stream(), parallelStream(), forEach, removeIf
- Stream: filter, map, flatMap, distinct, sorted, peek, limit, skip
- Terminal: collect, reduce, count, findFirst/findAny, anyMatch/allMatch/noneMatch
- Collectors: toList, toSet, toMap, groupingBy, partitioningBy, joining, counting
- Map: get, put, remove, putIfAbsent, computeIfAbsent, merge, replaceAll, forEach
- Concurrent: ConcurrentHashMap, CopyOnWriteArrayList, BlockingQueue

Apêndice — exemplos de código mais extensos

Exemplo: Collector customizado (contar palavras em paralelo)

```
Collector<String, Map<String,Integer>, Map<String,Integer>> wordCountCollector =
    Collector.of(
        HashMap::new,
        (map, word) -> map.merge(word, 1, Integer::sum),
        (map1, map2) -> { map2.forEach((k,v) -> map1.merge(k, v, Integer::sum)); return map1; },
        Collector.Characteristics.UNORDERED
    );

Map<String,Integer> counts = words.parallelStream().collect(wordCountCollector);
```

Exemplo: uso correto de parallelStream sem estado compartilhado

```
List<Integer> soma = IntStream.range(0, 1_000_000)
    .parallel()
    .map(i -> heavyComputation(i))
    .boxed()
    .collect(Collectors.toList());
```

FIM — dicas finais

1) Teste e meça: sempre profile antes e depois de otimizar. 2) Prefira clareza e imutabilidade quando possível. 3) Conheça o custo de alocação e boxing. 4) Entenda o comportamento de paralelismo da sua fonte (Spliterator).