JEP 431：有序集合

所有者	斯图尔特·马克斯
类型	新特性
范围	SE
地位	已关闭/已交付
发布	21
成分	核心库/java.util:集合
讨论	openjdk dot org 的核心 dash 库 dash 开发
审阅者	布赖恩·戈茨
受认可	布赖恩·戈茨
已创建	2022/01/27 22:13
更新	2023/10/23 17:55
问题	8280836

概括

引入新的接口来表示具有明确定义遍历顺序的集合。每个这样的集合都有一个明确定义的第一个元素，第二个元素，依此类推，直到最后一个元素。它还提供了统一的API来访问其第一个和最后一个元素，并以相反的顺序处理其元素。

> "生活只能在回顾中被理解；但它必须向前度过。"
> — 基尔凯郭尔

动机

Java 的collection框架缺乏一种表示具有明确定义遍历顺序的元素序列的集合类型。它还缺乏在这些集合中适用的一致操作集。这些空白一直是问题和抱怨的重要原因。

举例来说，List和Deque都定义了遍历顺序，但它们的共同超类型是Collection，而该类型并未定义遍历顺序。同样，Set也未定义遍历顺序，而其子类型如HashSet也未定义，但SortedSet和LinkedHashSet等子类型却定义了。因此，对遍历顺序的支持分布在类型层次结构中，使得在API中表达某些有用概念变得困难。既然既有Collection也有List都不能描述具有遍历顺序的参数或返回值。Collection过于通用，将这些约束留给了散文规范，可能导致难以调试的错误。而List则过于具体，排除了SortedSet和LinkedHashSet等。

一个相关的问题是，视图集合通常被迫降级到更弱的语义。使用Collections::unmodifiableSet将LinkedHashSet包装为Set，丧失了关于遍历顺序的信息。

由于没有定义它们的接口，与遍历顺序相关的操作要么不一致，要么缺失。虽然许多实现支持获取第一个或最后一个元素，但每个集合都定义了自己的方式，而且有些并不明显或完全缺失：

	第一个元素	最后一个元素
List	`list.get(0)`	`list.get(list.size()-1)`
Deque	`deque.getFirst()`	`deque.getLast()`
SortedSet	`sortedSet.first()`	`sortedSet.last()`
LinkedHashSet	`linkedHashSet.iterator().next()`	`// missing`

其中一些操作显得不必要繁琐，比如获取List的最后一个元素。有些操作甚至需要额外的努力：获取LinkedHashSet的最后一个元素的唯一方式是遍历整个集合。

同样，从头到尾迭代集合的元素是直观而一致的，但反向迭代却不是。所有这些集合都可以使用Iterator、增强for循环、stream()或toArray()进行正向迭代。而反向迭代在每种情况下都不同。NavigableSet为反向迭代提供了descendingSet()视图：

```
for (var e : navSet.descendingSet())
    process(e);
```

Deque通过反向迭代器（reverse Iterator）实现这一点：

```
for (var it = deque.descendingIterator(); it.hasNext();) {
    var e = it.next();
    process(e);
}
```

List使用ListIterator实现反向迭代：

```
for (var it = list.listIterator(list.size()); it.hasPrevious();) {
    var e = it.previous();
    process(e);
}
```

LinkedHashSet最终不提供对反向迭代的支持。在实际中，以反向顺序处理LinkedHashSet的元素的唯一可行方式是将其元素复制到另一个集合中。

类似地，使用流处理集合元素是使用循环处理元素的强大而有效的替代方法，但以相反的顺序获取流可能会很困难。在那些定义遍历顺序的各种集合中，唯一方便支持这一点的是NavigableSet：

```
navSet.descendingSet().stream()
```

其他的要么需要将元素复制到另一个集合中，要么需要从一个定制的Spliterator创建一个反向迭代的流。

这是一个不幸的状况。在集合框架中，具有定义的遍历顺序的集合概念存在于多个地方，但没有一个单一的类型来表示它。因此，对这些集合的一些操作是不一致或缺失的，而以相反的顺序处理元素则从不便利到不可能。我们应该填补这些空白。

描述

我们定义了用于 有序化 collections, 有序化 sets, 和有序化maps的新接口，然后将它们适配到现有的集合类型层次结构中。在这些接口中声明的所有新方法都有默认实现。

Sequenced collections

有序集合是一个具有明确定义遍历顺序的集合（这里使用的“序列化”一词是序列动词的过去分词，意味着“按照特定顺序排列元素”）。序列化集合具有第一个和最后一个元素，并且它们之间的元素具有后继和前驱。序列化集合支持在两端进行常见操作，并支持从第一个到最后一个以及从最后一个到第一个（即，正向和反向）处理元素。

```
interface SequencedCollection<E> extends Collection<E> {
    // new method
    SequencedCollection<E> reversed();
    // methods promoted from Deque
    void addFirst(E);
    void addLast(E);
    E getFirst();
    E getLast();
    E removeFirst();
    E removeLast();
```

新的 `reversed()` 方法提供了原始集合的反向排序视图。对原始集合的任何修改都会在视图中可见。如果允许，对视图的修改将写入原始集合。

反向排序的视图使得所有不同的序列化类型都能够使用所有常见的迭代机制在两个方向上处理元素：增强的for循环，显式的 `iterator()` 循环，`forEach()`，`stream()`，`parallelStream()` 和 `toArray()`。

例如，以前从LinkedHashSet获取反向排序的流可能相当困难；现在只需简单地使用：

```
linkedHashSet.reversed().stream()
```

（`reversed()` 方法本质上是一个重命名的 `NavigableSet::descendingSet`，并升级为 `SequencedCollection`。）
以下方法从 `Deque` 中升级为 `SequencedCollection`。它们支持在两端添加、获取和删除元素：

- `void addFirst(E)`
- `void addLast(E)`
- `E getFirst()`
- `E getLast()`
- `E removeFirst()`
- `E removeLast()`

`add*(E)` 和 `remove*()` 方法是可选的，主要是为了支持不可修改的集合情况。如果集合为空，`get*()` 和 `remove*()` 方法将抛出 `NoSuchElementException`。

在 `SequencedCollection` 中没有定义 `equals()` 和 `hashCode()`，因为其子接口具有冲突的定义。

Sequenced sets

一个有序化set是一个不包含重复元素的 `Set`，它是一个 `SequencedCollection`。

```
interface SequencedSet<E> extends Set<E>, SequencedCollection<E> {
    SequencedSet<E> reversed(); // covariant override
}
```

集合（例如`SortedSet`）通过相对比较来定位元素，因此无法支持显式定位操作，如在`SequencedCollection`超接口中声明的`addFirst(E)`和`addLast(E)`方法。因此，这些方法可能会抛出 `UnsupportedOperationException`。

`SequencedSet` 的 `addFirst(E)` 和 `addLast(E)` 方法对于像 `LinkedHashSet` 这样的集合具有特殊的语义：如果元素已经存在于集合中，则将其移动到适当的位置。这纠正了 `LinkedHashSet` 中的一个长期缺陷，即无法重新定位元素。

Sequenced maps

一个有序化map是一个具有明确定义遍历顺序的 `Map`。

```
interface SequencedMap<K,V> extends Map<K,V> {
    // new methods
    SequencedMap<K,V> reversed();
    SequencedSet<K> sequencedKeySet();
    SequencedCollection<V> sequencedValues();
    SequencedSet<Entry<K,V>> sequencedEntrySet();
    V putFirst(K, V);
    V putLast(K, V);
    // methods promoted from NavigableMap
    Entry<K, V> firstEntry();
    Entry<K, V> lastEntry();
    Entry<K, V> pollFirstEntry();
    Entry<K, V> pollLastEntry();
}
```

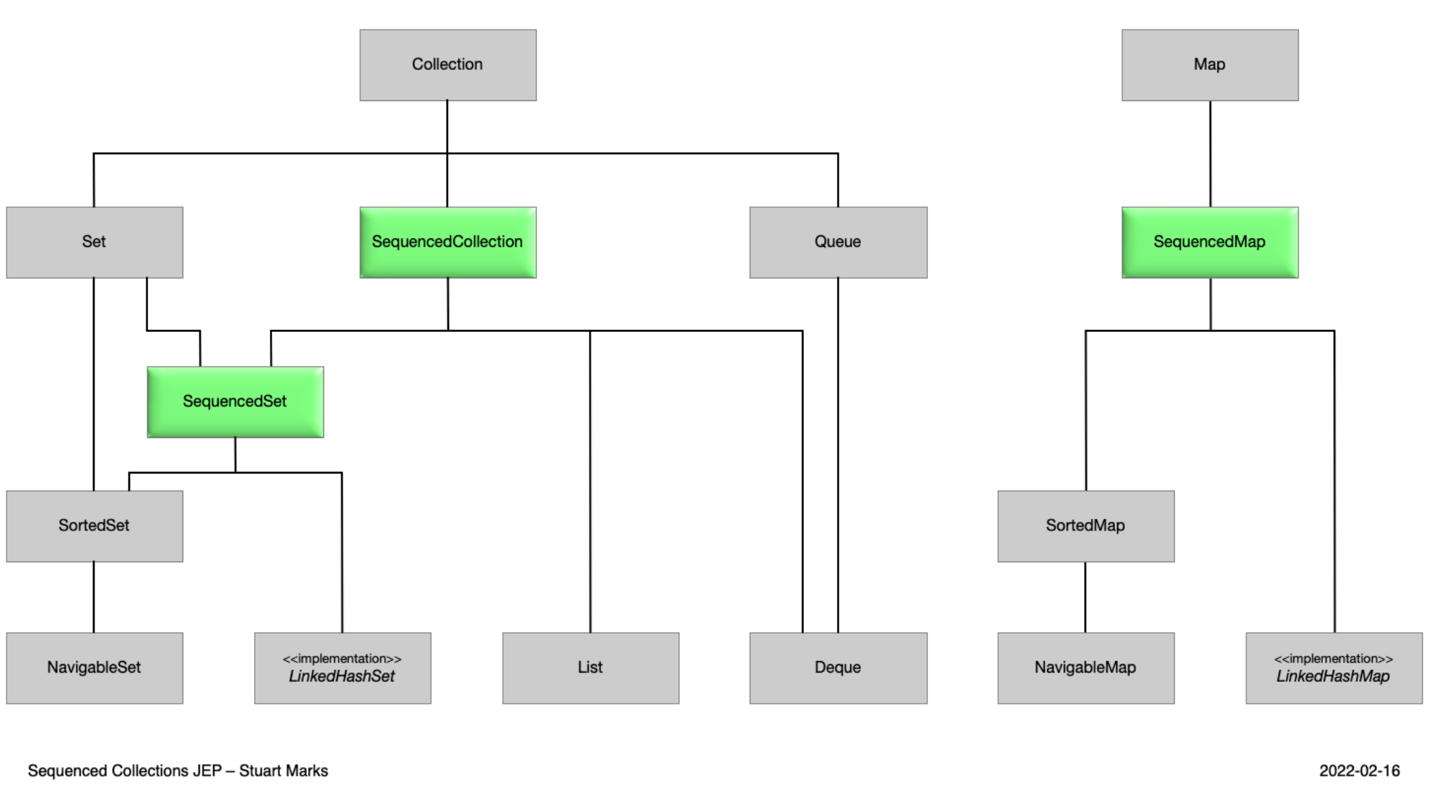
新`put*(K, V)`方法具有特殊情况语义，类似于 的相应`add*(E)`方法`SequencedSet`： 对于诸如 之类的地图`LinkedHashMap`，如果条目已存在于地图中，则它们具有重新定位条目的附加效果。对于诸如此类的地图 `SortedMap`，这些方法会抛出异常`UnsupportedOperationException`。

以下方法`SequencedMap`是从 `NavigableMap`. 它们支持在两端获取和删除条目：

- `Entry<K, V> firstEntry()`
- `Entry<K, V> lastEntry()`
- `Entry<K, V> pollFirstEntry()`
- `Entry<K, V> pollLastEntry()`

改造

上面定义的三个新接口完全适合现有的集合类型层次结构（单击放大）：



详细地说，我们对现有的类和接口进行以下调整：

- `List`现在将`SequencedCollection`作为其直接超接口，
- `Deque`现在将`SequencedCollection`作为其直接超接口，
- `LinkedHashSet`另外实现了`SequencedSet`，
- `SortedSet`现在将`SequencedSet`作为其直接超接口，
- `LinkedHashMap`另外实现了`SequencedMap`，
- `SortedMap`现在将`SequencedMap`作为其直接超接口。

我们对 `reversed()` 方法进行了协变覆盖，放置在适当的位置。例如，`List::reversed` 被覆盖为返回 `List` 类型的值，而不是 `SequencedCollection` 类型的值。

我们还向 `Collections` 实用类添加了新的方法，用于为这三种新类型创建不可修改的包装器：

- `Collections.unmodifiableSequencedCollection(sequencedCollection)`
- `Collections.unmodifiableSequencedSet(sequencedSet)`
- `Collections.unmodifiableSequencedMap(sequencedMap)`

替代方案

类型

一个替代方案是重新定义 `List` 接口作为一个通用的序列化集合类型。确实，`List` 是有序的，但它还支持通过整数索引进行元素访问。然而，许多序列化数据结构并不自然地支持索引，因此可能需要迭代地支持它。这将导致索引访问的性能为 O(n)，而不是期望的 O(1)，从而延续了 `LinkedList` 的错误。

`Deque` 作为一种通用序列类型似乎是有希望的，因为它已经支持了正确的操作集。然而，它混杂了其他操作，包括一系列返回空值的操作（offer、peek 和 poll），堆栈操作（push 和 pop），以及从队列继承的操作。这些操作对于队列来说是合理的，但对于其他集合来说可能不太合适。如果将 `Deque` 重新定义为通用序列类型，那么 `List` 也将成为一个队列，并支持堆栈操作，导致 API 混乱和令人困惑。

命名

我们选择的术语 "sequence" 暗示了按顺序排列的元素。它在各个平台上通常用于表示具有与上述描述类似的语义的集合。

术语 "ordered" 不够具体。我们需要在两个方向上进行迭代，并在两端进行操作。一个有序的集合，比如队列，是一个显著的异常情况：它是有序的，但也明显是不对称的。

在此提案的早期版本中使用的术语 "reversible" 并没有直观地唤起具有两个端点的概念。也许更大的问题是 Map 变体将被命名为 `ReversibleMap`，这会误导地暗示它支持通过键和值进行查找（有时被称为 `BiMap` 或 `BidiMap`）。

Add、Put和`UnsupportedOperationException`

如上所述，诸如 `SortedSet::addFirst` 和 `SortedMap::putLast` 的显式定位 API 会抛出 `UnsupportedOperationException`，因为它们的元素顺序是由相对比较决定的。有一些集合不实现所有 `SequencedCollection` 操作的不对称性可能看起来令人不悦。然而，这是有价值的，因为它将 `SortedSet` 和 `SortedMap` 纳入序列化集合家族，使它们可以更广泛地使用。这种不对称性也与集合框架中之前的设计决策一致。例如，`Map::keySet` 方法返回一个 `Set`，即使实现返回的 `Set` 不支持添加。

或者，可以通过按结构线重新排列接口来将添加操作保持分离。这将导致具有非常薄语义（例如 `AddableCollection`）的新接口类型，在实践中是没有用的，并且会使类型层次结构变得混乱。

历史

该提案是我们2021 年可逆集合提案的渐进演进。与该提案相比，主要的变化包括重命名、添加 `SequencedMap` 接口以及添加不可修改包装方法。

可逆集合提案又基于 Tagir Valeev 的2020年`OrderedMap/OrderedSet`提案。该提案的一些基本概念仍然存在，尽管在细节上有许多差异。

多年来，我们收到了许多请求和提案，主题是将 List 与 Set 或 Map 结合在一起。这些请求的常见主题包括包含唯一元素的 List，或者保持排序的 Set 或 Map。这些请求包括 4152834, 4245809, 4264420, 4268146, 6447049, 和 8037382.

其中一些请求在Java 1.4中引入LinkedHashSet和LinkedHashMap部分解决。虽然这些类确实满足了一些用例，但它们的引入在集合框架提供的抽象和操作中留下了空白，如上所述。

测试

我们将向 JDK 的回归测试套件添加一套全面的测试。

风险和假设

引入继承层次结构较高的新方法存在与明显方法名称（如 `reversed()` 和 `getFirst()`）冲突的风险。

特别关注的是对 `List` 和 `Deque` 上 `reversed()` 方法的协变覆盖。这对于已经同时实现了 `List` 和 `Deque` 的现有集合来说，在源代码和二进制上是不兼容的。在JDK中有两个这样的集合的例子：`LinkedList` 和一个内部类 `sun.awt.util.IdentityLinkedList`。对于 `LinkedList` 类，通过在 `LinkedList` 本身上引入一个新的 `reversed()` 协变覆盖来处理。而内部的 `IdentityLinkedList` 类被移除，因为它不再需要。

提案的早期版本引入了 `SequencedMap` 接口的 `keySet()`、`values()` 和 `entrySet()` 方法的协变覆盖。经过一些分析后，确定这种方法存在太大的不兼容风险；基本上，它使任何现有的子类都无效。选择了另一种方法，即在 `SequencedMap` 中引入新的方法 `sequencedKeySet()`、`sequencedValues()` 和 `sequencedEntrySet()`，而不是调整现有方法以进行协变覆盖。回顾起来，可能是出于相同的原因，在Java 6中引入了 `navigableKeySet()` 方法而不是修改现有的 `keySet()` 方法以进行协变覆盖。

有关不兼容风险的详细分析，请参见附加到CSR（ JDK-8266572）的报告。