

Коллекции Queue



Юрий
Пеньков




Юрий Пеньков

Java Software Engineer в InnoSTage



План занятия

1. [Структура данных – очередь](#)
2. [Методы Queue](#)
3. [Класс AbstractQueue](#)
4. [PriorityQueue](#)
5. [Deque](#)
6. [Класс Stack](#)



Структура данных — очередь

Структура данных Queue — очередь

Структура, реализующая принцип **First In, First Out** (первый вошел, первый вышел) — это значит, что тот элемент, который был первым добавлен в коллекцию, будет первым из нее извлечен.

<https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html>

Очереди не могут хранить значения `null`.





Queue



Методы Queue

Queue в java — это интерфейс однонаправленной очереди, наследуемый от общего интерфейса **Collection** и предоставляющий для реализации следующие методы:

- **add(e)** — добавляет элемент типа **T** в конец очереди, при успешном добавлении возвращает **true**, при неуспешном — выбрасывает соответствующее исключение.
- **Element()** — возвращает первый элемент из очереди типа **T**, если в очереди нет элементов, выбрасывает исключение **NoSuchElementException**. При этом элемент остается в очереди.
- **remove()** — возвращает первый элемент типа **T** из очереди, при этом удаляет из нее этот элемент, если элементов в очереди нет, выбрасывает исключение **NoSuchElementException**.

Методы Queue

- **boolean offer(T object)** — добавляет элемент типа **T** в конец очереди, при успешном добавлении возвращает **true**, при не успешном — **false**.
- **T peek()** — возвращает первый элемент типа **T** из очереди без последующего удаления элемента из нее, если в очереди нет элементов, метод возвращает **null**.
- **T poll()** — возвращает первый элемент типа **T** из очереди, при этом удаляет из нее элемент, если в очереди нет элементов, метод возвращает **null**.

где, **T** — тип данных, который будет храниться в очереди, может быть любым. Его также можно опустить при объявлении очереди, тогда она будет хранить все объекты, приводя их к типу **Object** и при извлечении любого элемента нужно будет приводить его к ожидаемому типу.

Важно отметить

- Методы `add`, `element` и `remove` требуют обработки исключений.
- Методы `remove` и `poll` – одновременно возвращают и удаляют элементы из очереди.
- Методы `peek` и `element` – возвращают элементы из очереди, оставляя их на том же месте.

Методы, наследуемые от интерфейса Collection

- `addAll`
- `clear`
- `contains`
- `containsAll`
- `equals`
- `hashCode`
- `isEmpty`
- `iterator`
- `removeIf`
- `retainAll`
- `size`
- `toArray`
- `remove`
- `removeAll`
- `splititerator`
- `stream`
- `parallelStream`



Скорость работы

Вставка элемента в конец очереди и извлечение элемента из ее начала
константно **$O(1)$** .

Класс `AbstractQueue`

Класс `AbstractQueue`

Как уже говорилось ранее `Queue` — это интерфейс, описывающий контракт однонаправленной очереди без какой-либо реализации.

Стандартная библиотека Java предоставляет разработчикам реализацию этих методов по умолчанию с помощью класса `AbstractQueue`, который в свою очередь реализует все методы интерфейса `Queue`.

<https://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html>

Примеры реализации `AbstractQueue` в JDK

На основе класса `AbstractQueue` в стандартной библиотеке Java реализованы следующие типы очередей:

1. `ArrayBlockingQueue`
2. `ConcurrentLinkedQueue`
3. `DelayQueue`
4. `LinkedBlockingDeque`
5. `LinkedBlockingQueue`
6. `LinkedTransferQueue`
7. `PriorityBlockingQueue`
8. `PriorityQueue`
9. `SynchronousQueue`

Класс `PriorityQueue`

Класс `PriorityQueue` — единственный класс, наследуемый только от интерфейса `Queue` и реализующей его.

Особенность этой очереди — возможность задавать порядок элементов в ней при вставке на основе сортировки.

По умолчанию элементы сортируются с использованием «natural ordering». Если нужно изменить порядок элементов, необходимо передать при создании очереди объект `Comparator`, который будет сравнивать объекты при добавлении и выставлять их в порядке сортировки. Данная коллекция так же не поддерживает хранение `null` в качестве элемента.

<https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html>



Пример использования PriorityQueue

Давайте рассмотрим пример отправки электронной почты с поздравлением с Новым Годом своих клиентов.

Если клиентов много, например, 100 тысяч, наш сервис может не справиться с нагрузкой, в таком случае нам поможет очередь `PriorityQueue`.

Вопрос

Почему очередь и как она поможет?

Ответ:

Не нужно руками следить за размером очереди и удалять из нее элементы, метод `poll` сделает это за нас.

Ничего не мешает использовать список (например `ArrayList`), но это будет неудобно в работе, нам придется руками добавлять и удалять элементы из него.

Плюсы очередей в том, что мы можем в один конец добавлять элементы, а из другого читать, тем самым балансируя нагрузку между источником и потребителем (использовать как буфер между ними).

Пример использования PriorityQueue

Положим все письма в эту коллекцию, а наш сервис будет забирать из нее сообщения и отправлять их по мере своей работы (скорости обработки).

```
PriorityQueue<PostMessage> queue = new PriorityQueue<>();
queue.offer(message1);           //добавляем сообщение в очередь
queue.offer(message2);           //добавляем еще одно сообщение в очередь
...
PostMessage message = queue.poll(); //извлекаем message1 из очереди
System.out.println(message.toString()); //выводим сообщение
```



Интерфейс Deque

Интерфейс Deque

Еще один важный интерфейс, наследуемый от интерфейса Queue — интерфейс Deque.

Этот тип расширяет однонаправленную очередь до двунаправленной и позволяет работать в режиме не только **First In, First Out**, но и в **Last In, First Out** (последний вошел, первый вышел).

<https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html>



Deque



Собственные методы интерфейса Deque

- **addFirst(T e), addLast(T e)** — методы добавляют новый элемент в начало или конец соответственно. Если добавить элемент не удалось, выбрасывают исключение `IllegalStateException`.
- **T getFirst(), T getLast()** — методы возвращают элемент очереди с начала или с конца соответственно. Элемент при этом из очереди не удаляется. Если элемента нет (очередь пуста), выбрасывают исключение `NoSuchElementException`.
- **removeFirst(), removeLast()** — методы удаляют элемент с конца или начала соответственно. Если удалить не удалось, выбрасывают исключение `NoSuchElementException`.
- **offerFirst(T e), offerLast(T e)** — методы добавляют новый элемент в начало или в конец соответственно. Если добавить элемент удалось, возвращают `true`, иначе — `false`.

Собственные методы интерфейса Deque

- **T peekFirst(), T peekLast()** — методы возвращают элемент с начала или с конца очереди соответственно. Элемент при этом из очереди не удаляется. Если очередь пуста, возвращают `null`.
- **T pollFirst(), T pollLast()** — методы возвращают элемент с начала или с конца очереди соответственно. Элемент после возвращения удаляется из очереди. Если очередь пуста, возвращают `null`.

где, **T** — тип данных, который будет храниться в очереди, может быть любым. Его также можно опустить при объявлении очереди, тогда она будет хранить все объекты, приводя их к типу `Object`. При извлечении любого элемента нужно будет приводить его к ожидаемому типу.



Примеры реализации Deque в JDK

На основе интерфейса Deque в стандартной библиотеке Java есть следующие коллекции:

1. ArrayDeque
2. ConcurrentLinkedDeque
3. LinkedBlockingDeque
4. LinkedList

Наиболее часто используемые коллекции из этого списка: ArrayDeque, LinkedList

Пример использования `ArrayDeque`

Давайте рассмотрим пример.

У нас есть список сообщений, которые мы хотим обработать, но в какой-то момент при обработке одного из сообщений возникает ошибка и мы хотим вернуть сообщение в наш список (очередь) и обработать позже, в этом случае нам отлично поможет `Deque`, мы сможем вернуть сообщение в начало очереди, и оно по-прежнему будет первым ожидать обработки.

Вопрос

Почему не воспользоваться списком (`ArrayList`) или однонаправленной очередью (`Queue`), в чем этой задаче преимущество двунаправленной очереди (`Deque`)?

Ответ:

Можно воспользоваться любой из трех коллекций, минус списка (`ArrayList`), как и в предыдущей задаче — ручное добавление и удаление элементов, минус однонаправленной очереди (`Queue`) — возможность вернуть элемент только в конец очереди, следовательно, если нам нужно сохранить очередность обработки элементов (сохранить последовательность) следует использовать двунаправленную очередь, чтоб иметь возможность вернуть элемент в самое начало очереди.

Пример использования ArrayDeque

Для добавления элемента в начало очереди, воспользуемся методом интерфейса Deque `addFirst`.

```
Deque<String> deque = new ArrayDeque<>();

deque.add("Message 1");
deque.add("Message 2");
deque.add("Message 3");
deque.add("Message 4");
deque.add("Message 5");

String message = deque.poll(); //poll Message 1

System.out.println(deque);
//Вывод: [Message 2, Message 3, Message 4, Message 5]

//Программа не может обработать наше сообщение message, и мы хотим вернуть его
обратно в начало очереди
deque.addFirst(message);

System.out.println(deque);
//Вывод: [Message 1, Message 2, Message 3, Message 4, Message 5]
```

LinkedList vs ArrayDeque

В примере из предыдущего слайда мы могли бы использовать класс из стандартной библиотеки Java `LinkedList`, если бы не знали следующих отличий между классами `LinkedList` (сделан на основе списка) и `ArrayDeque` (сделан на основе массива):

Производительность вставки и удаления элементов с любого конца у `ArrayDeque` амортизировано (на большом количестве рассматриваемых случаев) выше, чем у `LinkedList`.

Для каждого элемента `LinkedList` выделяется больше памяти (следовательно, вся коллекция занимает больше памяти).

<https://docs.oracle.com/javase/tutorial/collections/implementations/deque.html>



Класс Stack

Класс Stack

Мы с вами рассмотрели две очереди на основе интерфейсов `Queue` и `Deque`, в Java с ранних версий реализован класс `Stack`, реализующей концепцию **Last In, First Out** (последний вошел, первый вышел).

Класс `Stack` — реализован на основе класса `Vector`, в этом его недостаток, так как все операции в классе `Vector` синхронизированные, а значит медленные.

Класс `Stack` реализует следующие методы

- **`boolean empty()`** — возвращает `true` или `false`, проверяя, есть ли элементы в коллекции.
- **`T peek()`** — возвращает последний элемент типа `T` из стека без последующего удаления элемента из него, если в стеке нет элементов, метод возвращает `null`.
- **`T pop()`** — возвращает последний элемент типа `T` из стека при этом удаляет из него элемент, если в стеке нет элементов, метод возвращает `null`.

Класс `Stack` реализует следующие методы

- **`boolean push(T object)`** — добавляет элемент типа `T` в конец стека, при успешном добавлении возвращает `true`, при не успешном добавлении `false`.
- **`int search(Object o)`** — возвращает позицию элемента в стеке, если элемента нет метод возвращает `-1`.

<https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html>

где, `T` — тип данных, который будет храниться в очереди, может быть любым. Его также можно опустить при объявлении очереди, тогда она будет хранить все объекты с приводя их к типу `Object` и при извлечении любого элемента нужно будет приводить его к ожидаемому типу.



Stack





Важный момент

Документация языка Java не рекомендует использование класса `Stack`, вместо него рекомендуется к использованию двунаправленная очередь `Deque`, пример объявления `Deque<Integer> stack = new ArrayDeque<Integer>()`.



Чему мы научились

- Узнали, что такое однонаправленная очередь `Queue`;
- Узнали, что такое двунаправленная очередь `Deque`;
- Какие классы в JDK реализуют очереди;
- Узнали отличия `LinkedList` и `ArrayDeque`.



Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

**Задавайте вопросы и
пишите отзыв о лекции!**

Юрий Пеньков