Тонкости Коллекций в Java: о чём многие забывают

Из опыта code-review и ответов на StackOverflow набралось немало моментов, касающихся Java Collections API, которые мне казались очевидными, но другие разработчики о них почему-то не знали или знали, но не чувствовали уверенности их применять. В этой статье я собираю в общую кучу всё, что накопилось.

Содержание:

1. **LinkedList**
2. **List.subList**
3. **PriorityQueue**
4. **EnumSet и EnumMap**
5. **Set.add(E) и Set.remove(E) возвращают булево значение**
6. **Map.put(K, V), Map.remove(K), List.set(idx, E), List.remove(idx) возвращают предыдущий элемент**
7. **Map.keySet() и Map.values()**
8. **Arrays.asList может быть ключом**
9. **Collections.max**
10. **Stack, Vector, Hashtable**

LinkedList

С LinkedList случай особый. Вроде бы лучшего аналога связного списка нет и ходят легенды, что он на самом деле полезен. В действительности ситуаций, когда LinkedList лучше, чем ArrayList, в реальной жизни исключительно мало.

До Java-8 LinkedList ещё мог пригодиться, если вы часто удаляете элементы, идущие не последовательно, по какому-то условию.

В Java-8 для этих целей появился **List.removeIf**, который в ArrayList, конечно, реализован оптимальнее (элементы передвигаются только один раз).

Если вам надо сделать много вставок в разные места (задача сама по себе экзотическая), скорее всего быстрее будет создать новый ArrayList, чем вставлять в существующий LinkedList.

Ну и помните, что LinkedList кушает в несколько раз больше памяти, так как каждый элемент — это отдельный объект в куче со ссылками на следующий и предыдущий.

LinkedList можно использовать только в качестве учебного примера

List.subList

Про это уже [писали](http://habrahabr.ru/post/133357/), но стоит повторить. Наверно, самый недооценённый метод из Collections API. Бывает, что надо каким-то образом обработать часть списка (например, в алгоритмах семейства «разделяй и властвуй» или при распараллеливании задачи). Многие создают метод или класс, который завязывается на три параметра: List, from и to:

void processListPart(List<Item> list, int from, int to) {

for(int idx = from; idx < to; idx++) {

Item item = list.get(idx);

...

}

}

Так незачем делать. Реализации алгоритма должно быть плевать, что она обрабатывает часть списка. Пишите:

void processList(List<Item> list) {

for(Item item : list) {

...

}

}

И вызывайте

processList(list.subList(from, to));

Даже если у вас всё в одном методе, удобнее воспользоваться расширенным циклом for, чем возиться с индексами:

for(Item item : list.subList(from, to)) {...}

Кроме того, subList — полнофункциональный список, он работает и на запись, внося соответствующие изменения в родительский список. Нужно удалить много элементов из середины списка? Ничего нет проще:

list.subList(from, to).clear();

У популярных реализаций вроде ArrayList это выполняется очень быстро.  
  
Надо выяснить, начинается ли список с определённых элементов? И тут subList в руки!

List<String> prefix = Arrays.asList("a", "prefix", "values");

if(myList.size() >= prefix.size() &&

myList.subList(0, prefix.size()).equals(prefix)) {...}

Надо добавить в один список все элементы другого списка за исключением первого? И тут subList придёт на помощь:

list1.addAll(list2.subList(1, list2.size()));

Не забывайте, что можно писать Arrays.asList(array).subList(from, to), поэтому вышесказанное применимо и для непримитивных массивов. Структурно менять вы их не сможете, но передавать кусок массива в метод, принимающий список для чтения — легко.

PriorityQueue

Если subList — самый недооценённый метод, то PriorityQueue — это, на мой взгляд, самый недооценённый класс. Многие сталкиваются с задачей отыскать, скажем, 10 минимальных значений большого несортированного списка. Чаще всего список сортируют и потом берут первые 10 значений. Если исходный список менять нельзя, придётся его ещё скопировать для сортировки. А ведь очередь с приоритетом легко справится с этой задачей:

public static <T extends Comparable<T>> List<T> leastN(Collection<T> input, int n) {

assert n > 0;

PriorityQueue<T> pq = new PriorityQueue<>(Collections.reverseOrder());

for (T t : input) {

if (pq.size() < n) {

pq.add(t);

} else if (pq.peek().compareTo(t) > 0) {

pq.poll();

pq.add(t);

}

}

List<T> list = new ArrayList<>(pq);

Collections.sort(list);

return list;

}

Такой код в зависимости от данных может работать гораздо быстрее, чем сортировка. Например, для n = 10 и случайно заполненного списка из миллиона элементов очередь с приоритетом почти в сто раз обгоняет подход с сортировкой. При этом дополнительной памяти требуется O(n) и входные элементы можно обрабатывать в потоковом режиме (например, выбрать 10 наименьших чисел из входного файла).  
  
Вообще людям свойственно изучить пару-тройку структур данных и пользоваться ими везде. Не ленитесь, познакомьтесь с разными структурами.

EnumSet и EnumMap

До сих пор встречается код, где значения типа enum используют в качестве ключей в HashSet и HashMap. Хотя это работает, но оно неоправданно расточительно. Существующие специальные классы EnumSet и EnumMap значительно производительнее. Так если в enum не больше 64 разных значений, EnumSet хранит всё в одном поле типа long в битовой маске. EnumMap содержит все значения в обычном массиве той же длины, сколько элементов в enum, а ключи не хранит вовсе. Так как у каждого значения в enum есть порядковый номер ordinal(), можно легко перейти от enum-ключа к элементу массива. Также никогда не нужно менять размер массива.

Set.add(E) и Set.remove(E) возвращают булево значение

Часто вижу подобный код:

if(!set.contains(item)) {

set.add(item);

*// do something*

} else {

*// do something else*

}

Не надо забывать, что операция добавления в Set возвращает true, если добавление успешно (то есть элемента не было) и false, если такой элемент уже был. Незачем усложнять код и два раза пробивать элемент по хэш-таблице или двоичному дереву, ведь можно написать:

if(set.add(item)) {

*// do something*

} else {

*// do something else*

}

Аналогично с удалением. Цепочка

if(set.contains(item)) {

set.remove(item);

...

}

заменяется на

if(set.remove(item)) { ... }.

Map.put(K, V), Map.remove(K), List.set(idx, E), List.remove(idx) возвращают предыдущий элемент

Из той же оперы ситуация. Методы, изменяющие или удаляющие элемент в коллекции возвращают предыдущее значение, и этим надо пользоваться. Не надо писать, например, так:

Item item = myMap.get(key);

myMap.put(key, newItem);

Написать просто Item item = myMap.put(key, newItem);. Хотите поменять местами две записи в Map с ключами key1, key2? Временная переменная не нужна:

myMap.put(key1, myMap.put(key2, myMap.get(key1)));

Map.keySet() и Map.values()

Многие почему-то забывают, что Map.keySet() и Map.values() возвращают отображения исходного Map, которые позволяют удалять элементы (если Map модифицируемый). Надо оставить в Map только записи с определёнными значениями (и любыми ключами)? Пожалуйста:

myMap.values().retainAll(toRetain);

Также работает removeAll, а с Java-8 ещё и removeIf:

*// Сгруппируем сотрудников по названиям подразделений*

Map<String, List<Employee>> perDepartment = employees.stream().collect(groupingBy(Employee::getDepartmentName, HashMap::new, toList()));

*// Оставим только крупные подразделения с числом сотрудников от 10*

perDepartment.values().removeIf(o -> o.size() < 10);

Arrays.asList может быть ключом

Бывает, что вам нужно сформировать Map или Set, используя кортеж значений. Например, у вас есть PoJo-объекты Item, у которых имеются поля name, type, version. У них уже написан equals и hashCode, их можно складывать в HashSet, всё нормально. Но вы хотите выбрать из коллекции уникальные объекты только по полям name и type, игнорируя version. Менять существующие equals и hashCode нельзя. В таких ситуациях люди часто создают отдельный класс только с полями name и type и используют его в качестве ключа. Однако для одноразовой операции проще использовать Arrays.asList():

Map<List<Object>, Item> map = new HashMap<>();

for(Item item : items) {

map.put(Arrays.asList(item.name, item.type), item);

}

Collection<Item> unique = map.values();

Arrays.asList() создаёт список из нужного числа элементов и у него как раз подходящие реализации equals и hashCode: никакой boilerplate не нужен. Так можно создать ключ любой длины, причём корректно обработаются null-значения и примитивы (брагодаря боксингу). Не сработает только, если вы хотите в составе ключа иметь массив.

Collections.min/max

Удивительно, насколько часто можно встретить написанный вручную код, который находит максимальный или минимальный элемент чего-то по какому-нибудь критерию. Казалось бы, такая тривиальная задача должна быть давно решена. На самом деле она и так давно решена: есть методы Collections.min и Collections.max. Раньше было не очень удобно писать компараторы, но в Java-8 всё стало легче.  
  
К примеру, вам нужно найти ключ в Map, соответствующий максимальному значению. Пишите так:

maxKey = Collections.max(map.entrySet(), Map.Entry.comparingByValue()).getKey();

Можно и через Stream API, но Collections.max() несколько быстрее. Если вы не можете использовать Java-8 и компараторы вроде Entry.comparingByValue() вам недоступны, их нетрудно написать.

Stack, Vector, Hashtable, LinkedList

Просто не используйте эти классы. Пользы от них никакой нет. Вместо Stack пользуйтесь ArrayDeque, вместо Vector — ArrayList, вместо Hashtable — HashMap. Если вам нужна потокобезопасность, они вам всё равно не помогут. Возможно, в девятке их всё-таки пометят @Deprecated (смотрите [JEP 277](http://openjdk.java.net/jeps/277)).  
  
.  
  
На сегодня всё. Программируйте с удовольствием!

Комментарии 37

* [[https://habrastorage.org/getpro/habr/avatars/7ad/1ce/310/7ad1ce31064020bdb79dd73c755ad5ff_small.jpg](https://habr.com/ru/users/Fen1kz/)**Fen1kz**](https://habr.com/ru/users/Fen1kz/)17 декабря 2015 в 17:41

**+1**

Отличная статья, спасибо!  
Однако, я бы не стал делать

if(set.add(item)) {

Ибо слышал что влиять на данные в проверках не есть хорошо.

* + [**lany**](https://habr.com/ru/users/lany/)17 декабря 2015 в 18:56

**+4**

Рациональное зерно есть, конечно. Поклонники функционального программирования скажут, что почти любой вызов метода не должен давать побочных эффектов :-) Но та же стандартная библиотека Java (во всяком случае до JDK 7) вполне поощряет if(file.delete()) или if(folder.mkdirs()). Тут побочный эффект явно есть. И даже какой-нибудь if(matcher.find()) меняет состояние объекта — заполняет группы.

* + - [[https://habrastorage.org/getpro/habr/avatars/433/937/30e/43393730ed01e87035567d745cb92dc9_small.png](https://habr.com/ru/users/Googolplex/)**Googolplex**](https://habr.com/ru/users/Googolplex/)18 декабря 2015 в 00:44

**0**

Справедливости ради, file.delete() и file.mkdirs() — это части устаревшего API. В NIO.2 ничего подобного, к счастью, нет.

* + - [**terryP**](https://habr.com/ru/users/terryP/)18 декабря 2015 в 01:10

**+2**

На самом деле, вопрос даже не в том что изменение объекта в проверке спорная практика, а том что разработчики на Java не привыкли к трюкам вида «if(set.add(item))» или «put(...put( ))», поэтому если писать код, который будешь читать только ты — такое допустимо, а вот если ещё куча народа, то я лучше «if(!set.contains(item))… » напишу, чтобы не обяснять всем и каждому на код ревью, что я хотел тут сделать.

* + - * [[https://habrastorage.org/getpro/habr/avatars/358/9dd/65b/3589dd65ba8d1c2f80bd6262be1cd4cb_small.jpg](https://habr.com/ru/users/dougrinch/)**dougrinch**](https://habr.com/ru/users/dougrinch/)18 декабря 2015 в 02:26

**+7**

А чем плох вариант

boolean wasAdded = set.add(item);

if (wasAdded) {

Ничего (почти) лишнего. И объяснять никому ничего не надо.  
  
p.s. И даже изменения в проверке нет.

* + - * [**lany**](https://habr.com/ru/users/lany/)18 декабря 2015 в 08:22

**+3**

Насчёт того, к чему привыкли разработчики, можно много говорить. Определённо, например, Java-разработчики не привыкли к лямбдам и Stream API, потому что они появились не так давно. Но это не повод их не использовать. Если в команде люди чего-то не знают, на мой взгляд лучше им объяснить, чем руководствоваться принципом «не буду использовать крутые штуки, а то ещё коллеги будут плеваться». Я всегда пишу как считаю нужным, не оглядываясь на уровень знаний коллег.  
  
Конечно, put(...put( )) — довольно экстремальный пример, поэтому его стоит вынести в метод типа swapValues(map, key1, key2), тогда всё станет гораздо понятнее.

* [[https://habrastorage.org/getpro/habr/avatars/3d8/ed1/d4a/3d8ed1d4aabbb2ccb1ebb363b7e2c927_small.jpg](https://habr.com/ru/users/leventov/)**leventov**](https://habr.com/ru/users/leventov/)17 декабря 2015 в 21:08

**+1**

Добавлю:  
  
1. Часто идет код с двойными проверками в Map, например if (!map.containsKey(key)) { map.put(key, value) }. Похоже на случай с Set, но осложняется тем, что в Map есть дополнительная неоднозначность: null вернули, потому что записи не было, или потому что была запись со значением null? Новые методы, добавленные в Java 8 в интерфейс Map, полностью покрывают возможные ситуации (конкретно в примере выше — map.putIfAbsent()).  
  
2. Иногда встречается итерация по Map:  
  
for (K key: map.keySet()) {  
doSmth(key, map.get(key));  
}  
  
Очень неэффективно, не говоря об итерации через entrySet(), в Java 8 лучше использовать map.forEach((k, v) -> ...)

* + [**lany**](https://habr.com/ru/users/lany/)18 декабря 2015 в 04:48

**0**

Спасибо за дополнение. Кстати, неправда ваша :-) Вот описание [putIfAbsent](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html" \l "putIfAbsent-K-V-):

If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.

От null-ключей и null-значений желательно отказаться вообще. Во-первых, их не поддерживают ConcurrentHashMap и ConcurrentSkipListMap. Сколько раз видел страдальцев, которые переделывали существующую HashMap на ConcurrentHashMap и обнаруживали, что всё стало падать из-за null'ов. Во-вторых, новые методы (семейство compute, merge) как раз предполагают, что null-значение — это отсутствие значения.

* + - [[https://habrastorage.org/getpro/habr/avatars/3d8/ed1/d4a/3d8ed1d4aabbb2ccb1ebb363b7e2c927_small.jpg](https://habr.com/ru/users/leventov/)**leventov**](https://habr.com/ru/users/leventov/)18 декабря 2015 в 04:53

**0**

В чем неправда? Нуллы — плохо, да, я их упомянул лишь как фактор, который вносит дополнительное замешательство.

* + - * [**lany**](https://habr.com/ru/users/lany/)18 декабря 2015 в 05:36

**0**

Неправда в том, что putIfAbsent не различает ситуации «значение null» и «значение отсутствует». Может, конечно, я не так понял :-)

* + - * + [**poxvuibr**](https://habr.com/ru/users/poxvuibr/)18 декабря 2015 в 16:58

**0**

If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.

Перевод:

Если указанный ключ ещё не связан с каким либо значением (или значение — null), связывает его с заданным значением и возвращает null, в ином случае возвращает его текущее значение.

[**lany**](https://habr.com/ru/users/lany/)18 декабря 2015 в 18:07

**0**

Да не, английский я-то понял :-) Я мысль [leventov](http://habrahabr.ru/users/leventov/) мог не понять. Плохо сформулировал.

* + - * + [[https://habrastorage.org/getpro/habr/avatars/3d8/ed1/d4a/3d8ed1d4aabbb2ccb1ebb363b7e2c927_small.jpg](https://habr.com/ru/users/leventov/)**leventov**](https://habr.com/ru/users/leventov/)18 декабря 2015 в 23:58

**0**

Если отбросить всякие нуллы, putIfAbent все равно в 2 операции только делается, до Java 8, и если это не ConcurrentMap

* [**poxvuibr**](https://habr.com/ru/users/poxvuibr/)17 декабря 2015 в 22:02

**0**

Статья очень хорошая, спасибо!

* [**mbait**](https://habr.com/ru/users/mbait/)18 декабря 2015 в 02:05

**0**

.

* [[https://habrastorage.org/getpro/habr/avatars/91f/9b2/2be/91f9b22be14f4a530718cff16894e846_small.png](https://habr.com/ru/users/naething/)**naething**](https://habr.com/ru/users/naething/)18 декабря 2015 в 07:40

**0**

Пример с PriorityQueue проще заменить на [Ordering.leastOf()](http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/com/google/common/collect/Ordering.html" \l "leastOf%28java.lang.Iterable,%20int%29) из Guava.

* + [**lany**](https://habr.com/ru/users/lany/)18 декабря 2015 в 08:18

**0**

Да, в сторонних библиотеках много всего нужного. Я лично бы заменил на [MoreCollectors.least()](http://amaembo.github.io/streamex/javadoc/one/util/streamex/MoreCollectors.html" \l "least-int-) из моей библиотеки :-) Он и распараллеливается нормально :-)

* [**tebuchet**](https://habr.com/ru/users/tebuchet/)18 декабря 2015 в 10:40

**+1**

Дубли в содержании  
6. Arrays.asList может быть ключом  
9. Arrays.asList может быть ключом

* + [**lany**](https://habr.com/ru/users/lany/)18 декабря 2015 в 11:03

**0**

Ой, и правда. Спасибо!

* [[https://habrastorage.org/getpro/habr/avatars/fc7/2ab/833/fc72ab83365007635ab126459e925156_small.jpg](https://habr.com/ru/users/burjui/)**burjui**](https://habr.com/ru/users/burjui/)18 декабря 2015 в 16:30

**0**

Для аргументов лучше использовать минимальный интерфейс последовательностей — Iterable<T>, если, конечно, не нужен какой-нибудь специфичный для класса коллекции метод:

void processItems(final Iterable<Item> items) {

for (final Item item : items) {

...

}

}

Тогда можно будет передавать в processItems() не только List, но и Set и другие коллекции, а также, если вам не чужда функциональщина, ленивые последовательности вроде Sequence<T> из библиотеки totallylazy ( https://github.com/bodar/totallylazy/ ). Если нужно иметь возможность модифицировать коллекцию — принимайте Collection<T>.

* + [[https://habrastorage.org/getpro/habr/avatars/7eb/763/2e5/7eb7632e5a0d2720be542c25ec40614f_small.jpg](https://habr.com/ru/users/vedenin1980/)**vedenin1980**](https://habr.com/ru/users/vedenin1980/)18 декабря 2015 в 16:57

**+3**

Iterable

ИМХО, использование Iterable хуже чем Collection (если вам реально не нужно Iterable), так как:   
  
1. Вы не сможете получить size, соответственно как только вам потребуется создать массив, ArrayList и т.п. у вас будут проблемы, придется или выделять память c запасом или терять производительность,  
  
2. Вы не сможете работать с большим количеством полезных функций из Collections (guava несколько исправляет эту проблему, но тем не менее),  
  
3. Вы не сможете легко и просто создать stream в Java 8,   
  
В целом, я бы крайне осторожно использовал бы замену Collection на Iterable в больших функциях, чтобы не пришлось рефакторить пол приложения, когда потребуется Collection или не делать костыли и терять производительность.

* + - [[https://habrastorage.org/getpro/habr/avatars/fc7/2ab/833/fc72ab83365007635ab126459e925156_small.jpg](https://habr.com/ru/users/burjui/)**burjui**](https://habr.com/ru/users/burjui/)18 декабря 2015 в 17:58

**0**

Процитирую сам себя:

если, конечно, не нужен какой-нибудь специфичный для класса коллекции метод

size() сюда же. Разумеется, Iterable не подходит для случаев, когда вы нужно знать точное количество элементов. Впрочем, это зависит от задачи.

Вы не сможете работать с большим количеством полезных функций из Collections (guava несколько исправляет эту проблему, но тем не менее)

Вы не сможете легко и просто создать stream в Java 8

Согласен. Впрочем, зато я легко смогу использовать totallylazy, которая функционально (нечаянный каламбур) гораздо богаче Stream из Java 8, частично заменяет Collections и доступна на Android (с Retrolambda идёт на ура).

В целом, я бы крайне осторожно использовал бы замену Collection на Iterable в больших функциях, чтобы не пришлось рефакторить пол приложения, когда потребуется Collection или не делать костыли и терять производительность.

Заменять и не надо. Просто пишите новые алгоритмы, исходя из принципа минимализма: нужна последовательность — принимай Iterable, понадобится менять последовательность — меняешь на Collection (не надо переписывать половину кода для этого).

* + - * [[https://habrastorage.org/getpro/habr/avatars/fc7/2ab/833/fc72ab83365007635ab126459e925156_small.jpg](https://habr.com/ru/users/burjui/)**burjui**](https://habr.com/ru/users/burjui/)18 декабря 2015 в 18:08

**0**

Во избежание недопонимания: я только предлагаю использовать Iterable для аргументов, когда не нужна возможность менять последовательность — но и тогда лучше принимать базовые типы вроде Collection и List. Я не призываю возвращать Iterable из функций, тем более что это не нужно, т.к. любая коллекция реализует этот интерфейс, и не вызванному методу решать, как возвращаемая коллекция будет использоваться дальше (если, конечно, речь не идёт о каком-либо API).

* + - * [[https://habrastorage.org/getpro/habr/avatars/7eb/763/2e5/7eb7632e5a0d2720be542c25ec40614f_small.jpg](https://habr.com/ru/users/vedenin1980/)**vedenin1980**](https://habr.com/ru/users/vedenin1980/)18 декабря 2015 в 18:19

**0**

удаленно, уже не актуально…

* + - * [**lany**](https://habr.com/ru/users/lany/)18 декабря 2015 в 18:38

**0**

Кстати, а что там такого актуального есть в totallylazy, чего в Stream API нету? Я заглянул что-то, думал идей натырить для моей либы, но сходу ничего интересного не нашёл.

* + - * + [[https://habrastorage.org/getpro/habr/avatars/fc7/2ab/833/fc72ab83365007635ab126459e925156_small.jpg](https://habr.com/ru/users/burjui/)**burjui**](https://habr.com/ru/users/burjui/)18 декабря 2015 в 19:03

**0**

К сожалению, документации по этой библиотеке кот наплакал, так что приходится копать исходники: https://github.com/bodar/totallylazy/blob/master/src/com/googlecode/totallylazy/Sequence.java  
Просто сравните, какие методы есть у Sequence, и какие — у Stream. Мне, например, недавно пригодились groupBy и zip. К тому же, totallylazy — библиотека для ФП, так что кроме Sequence, там есть Either и Option, которые тоже умеют map/flatMap, а также всякие монады с функторами, предикаты и их модификаторы, да куча всего. Почти все методы доступны для статического импорта. Пример из приложения для Android:

final int color = option(json.optString("some\_color", null))

.filter(not(String::empty))

.flatMap(Color::parseColor)

.getOrElse(Color.WHITE);

view.setBackgroundColor(color);

[**lany**](https://habr.com/ru/users/lany/)19 декабря 2015 в 16:37

**0**

А, ну такое-то есть в [Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html): и filter, и flatMap, и orElse.

* + - НЛО прилетело и опубликовало эту надпись здесь
  + [[https://habrastorage.org/getpro/habr/avatars/aa1/95f/3c6/aa195f3c6fca0296646d138410db7f8a_small.png](https://habr.com/ru/users/Flammar/)**Flammar**](https://habr.com/ru/users/Flammar/)18 декабря 2015 в 23:18

**0**

К сожалению, конструкторы коллекций, по соглашению, тоже принимают в качестве аргумента не Iterable<T>, а Collection<T>.

* [[https://habrastorage.org/getpro/habr/avatars/aa1/95f/3c6/aa195f3c6fca0296646d138410db7f8a_small.png](https://habr.com/ru/users/Flammar/)**Flammar**](https://habr.com/ru/users/Flammar/)19 декабря 2015 в 00:41

**+1**

Мне кажется, программисты боятся использовать в качестве ключей возвращаемые из методов «хитрые» коллекции потому, что помнят, что у «нормальных» классов метод equals сначала проверяет одинаковость классов объектов. А с «производными» списками неодинаковость классов практически гарантирована.

* [[https://habrastorage.org/getpro/habr/avatars/d5b/546/210/d5b546210c1e5d466571335f2099cc95_small.jpg](https://habr.com/ru/users/sergey-b/)**sergey-b**](https://habr.com/ru/users/sergey-b/)23 декабря 2015 в 20:40

**–3**

ArrayList, хотя и тратит меньше памяти, чем LinkedList, но требует, чтобы память была выделена одним целым куском, а это более дорогой ресурс.  
  
Если есть необходимость вставки в середину, то надо использовать LinkedList, и то, что эта задача названа экзотической, ничего не меняет.

* + [[https://habrastorage.org/getpro/habr/avatars/be3/21c/e59/be321ce5998c10ffff4b20f0c5dd9862_small.jpg](https://habr.com/ru/users/grossws/)**grossws**](https://habr.com/ru/users/grossws/)23 декабря 2015 в 21:54

**+4**

Каждый LinkedList.Node требует выделение 24 байт (при CompressedOops или 32-битной JVM), реальное выделение будет, скорее всего 32 байта или больше (в зависимости от конкретной JVM и её настроек), плюс 24 байта на сам LinkedList.  
  
Стандартный размер ArrayList — 10 элементов — 20 на ArrayList и 16 + 10\*4 на массив из 10 элементов при тех же условиях (на примере hotspot 64bit при Xmx менее 32G).  
  
Т. е. LinkedList на 10 элементов займёт 352 байта, а ArrayList — 96 байт при выравнивании на 32 байта. Не считая дикого насилия над кэшем и отсутствия нормального prefetch при итерации по LL.

Если есть необходимость вставки в середину, то надо использовать LinkedList, и то, что эта задача названа экзотической, ничего не меняет.

Очень сильно зависит от условий. Часто копирование линейного куска памяти много дешевле, чем выделение нового LL.Node и модификация ещё двух соседних.

* + - [[https://habrastorage.org/getpro/habr/avatars/fc7/2ab/833/fc72ab83365007635ab126459e925156_small.jpg](https://habr.com/ru/users/burjui/)**burjui**](https://habr.com/ru/users/burjui/)24 декабря 2015 в 02:17

**0**

Не считая дикого насилия над кэшем и отсутствия нормального prefetch при итерации по LL.

Это, пожалуй, самый важный момент. К тому же, если блок памяти целиком помещается в cache line, это полностью устраняет задержки на чтение RAM, поэтому экономить память и держать данные близко друг к другу очень важно для обеспечения максимальной производительности. Хотя JVM/Dalvik вряд ли дадут выжать из процессора все соки.

* + - [[https://habrastorage.org/getpro/habr/avatars/d5b/546/210/d5b546210c1e5d466571335f2099cc95_small.jpg](https://habr.com/ru/users/sergey-b/)**sergey-b**](https://habr.com/ru/users/sergey-b/)24 декабря 2015 в 02:41

**0**

Да, зависит. А также зависит от соотношения количества операций вставки со всеми остальными операциями. Но вывод о том, что ArrayList или ArrayDeque всегда лучше LinkedList, все равно остается неверным. Именно с этим выводом, сделанным в статье, я не согласен. Нетрудно подобрать такие параметры коллекции и варианты использования, при которых LinkedList будет работать лучше.  
  
Кстати, хорошая идея — реализовать LinkedList на массиве. Надо будет при случае попробовать.

* + - * [**lany**](https://habr.com/ru/users/lany/)24 декабря 2015 в 04:46

**0**

Да, их подобрать и создать искусственный пример можно. Вопрос лишь в том, кому и когда в жизни пригодится этот искусственный пример. Напомню, что LinkedList больше, чем вдвое проигрывает по памяти ArrayList. Так что даже сделав полную копию с нужными вставками в нужные места, вы потратите меньше памяти.

* + - * + [[https://habrastorage.org/getpro/habr/avatars/be3/21c/e59/be321ce5998c10ffff4b20f0c5dd9862_small.jpg](https://habr.com/ru/users/grossws/)**grossws**](https://habr.com/ru/users/grossws/)24 декабря 2015 в 18:56

**0**

Есть ещё забавный подход в scala.collection.immutable.Vector, строится на массивах по 32 элемента, эффективно поддерживает добавление элементов слева и справа, менее эффективно — вставку. Если элементов мало — хранит их в массиве, когда становится много — в trie таких массивов.

[**lany**](https://habr.com/ru/users/lany/)25 декабря 2015 в 06:43

**+1**

В Java внутри есть подобная штука — [SpinedBuffer](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/util/stream/SpinedBuffer.java" \l "SpinedBuffer). Только она для внутреннего использования в Stream API, наружу не выставлена.