Nachdenkzettel: Collections

1. ArrayList oder LinkedList – wann nehmen Sie was?

Bei statischen Datenstrukturen, die eine feste Große bzw. einen festen Speichergebrauch besitzen, verwendet man Array List. Wenn die Datenstrukturen dynamisch sind und die Größe ohne Aufwand geändert werden kann und sich der Speichergebrauch verändert, verwendet man LinkedList.

* Da ArrayList intern ein Array benutzt, ist der Zugriff auf ein spezielles Element über die Position in der Liste sehr schnell. Eine LinkedList muss aufwändiger durchsucht werden, und dies kostet Zeit.
* Die verkettete Liste ist aber deutlich im Vorteil, wenn Elemente mitten in der Liste gelöscht oder eingefügt werden; hier muss einfach nur die Verkettung der Hilfsobjekte an einer Stelle verändert werden. Bei einer ArrayList bedeutet dies viel Arbeit, es sei denn, das Element kann am Ende gelöscht oder – bei ausreichender Puffergröße – eingefügt werden. Soll ein Element nicht am Ende eingefügt oder gelöscht werden, müssen alle nachfolgenden Listenelemente verschoben werden.
* Bei einer ArrayList kann die Größe des internen Feldes zu klein werden. Dann bleibt der Laufzeitumgebung nichts anderes übrig, als ein neues, größeres Feld-Objekt anzulegen und alle Elemente zu kopieren.

2. Interpretieren Sie die Benchmarkdaten von: <http://java.dzone.com/articles/java-collection-performance>. Fällt etwas auf?



3. Wieso ist CopyOnWriteArrayList scheinbar so langsam?

Weil CopyOnWriteArray vor Operationen wie add,set.. eine neue Kopie des Arrays anfertigt und dann erst die Änderung hinzufügt. Damit schützt man die „Originalversion“ des Arrays. Es wird daher bevorzugt eingesetzt, wenn man nur wenige Änderungen durchführen muss und es viele Leseansichten gibt.

4. Wie erzeugen Sie eine thread-safe Collection (die sicher bei Nebenläufigkeit ist) (WAS?? die Arraylists, Linkedlists, Maps etc. sind NICHT sicher bei multithreading??? Wer macht denn so einen Mist???)

Methode 1: Man benutzt Collections.synchronizedList(): Wenn mehrere Zugriffe auf eine Liste geschehen, sollte vorher sichergestellt werden, dass diese jeweils alle über eine vorher synchronisierte Liste geschehen. Bei jeder Iteration über eine Liste wird also manuell synchronisiert.

Methode 2: Man benutzt CopyOnWriteArrayList. Bei CopyOnWriteArrayList, wird bei allen veränderlichen Operationen (add, set, remove..) eine separate Kopie des schon vorliegenden Arrays unternommen, dadurch ist es thread-safe.

5. Achtung Falle!

List|<Integer> list = new ArrayList<Integer>;

Iterator<Integer> itr = list.iterator();

**while**(itr.hasNext()) {

**int** i = itr.next();

**if** (i > 5) { *// filter all ints bigger than 5*

list.remove();

}

}

Falls es nicht klickt: einfach ausprobieren...

Macht das Verhalten von Java hier Sinn?

Gibt es etwas ähnliches bei Datenbanken? (Stichwort: Cursor. Ist der ähnlich zu Iterator?)

6. Nochmal Achtung Falle: What is the difference between get() and remove() with respect to Garbage Collection?

7. Ihr neuer Laptop hat jetzt 8 cores! Ihr Code für die Verarbeitung der Elemente einer Collection sieht so aus:

Iterator<Integer> itr = list.iterator();

**while**(itr.hasNext()) {

**int** i = itr.next();

*//do something with i….*

}

War der Laptop eine gute Investition?

Für die Mutigen: mal nach map/reduce googeln!