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?



Es fällt auf, dass die verschiedenen Collection-Arten unterschiedlich performant sind bei den verschiedenen Anwendungsbeispielen. Je nachdem was mit einer Collection gemacht wird sollte man darauf achten, dass die entsprechende Collection performance technisch dafür geeignet ist, da zum Beispiel CopyOnWriteArrayList wenn man viele Elemente hinzufügt sehr langsam ist.

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?

Jain, da man einen Iterator für eine Liste mit list.iterator(); bekommt, und somit der Iterator ein Attribut der Liste ist könnte Java theoretisch selbst erkennen, dass einer benutzt wird und diesen direkt bei der remove Funktion mit dem aktuellen Wert benutzen, anstatt dass man die remove Funktion direkt vom Iterator aus aufrufen muss, denn man löscht ja nichts aus dem Iterator.

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

Ja, dieser funktioniert als Pointer zu einer bestimmten Zeile in einer Abfrage, dieser kann nach Bedarf auch immer weiter zur jeweils nächsten Zeile bewegt werden. Dies kann im Programmcode jedoch automatisch passieren, ohne dass man diesen immer wieder (in Java mit Iterator.next()) inkrementieren und übergeben muss.

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

Mit list.get(index) bekommt man nur den aktuellen Wert zurückgegeben, während list.remove(index) den Wert zurück gibt und ihn anschließend löscht. Da er nur beim list.remove(index) gelöscht wird, kann auch nur dort der Garbage Collector den Wert tatsächlich löschen.

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?

Nein, denn die Schleife wird nur von höchstens einem Kern gleichzeitig genutzt. Sollten alle Cores genutzt werden, müsste man den Programmcode umschreiben.

Für die Mutigen: mal nach map/reduce googeln!