

Nachdenkzettel: Collections

1. ArrayList oder LinkedList – wann nehmen Sie was?

Bei kleinen Datenmengen ist es egal, da man kaum einen Unterschied in der Performance spürt. Je größer die `LinkedList`, um so langsamer ist das Auslesen von einzelnen Einträgen. Bei Schreibvorgängen muss hingegen nur an der richtigen Stellen die Referenzen angepasst werden. Daher sind Schreibvorgängen meist schneller als bei einer `ArrayList`. Wenn man die Position des gesuchten Eintrags kennt, ist das Auslesen in einer `ArrayList` immer gleich schnell. Die Größe der `ArrayList` hat darauf keinen Einfluss. Es kommt bei der Entscheidung immer auf den individuellen Anwendungsfall an.

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



Es sticht heraus, dass CopyOnWriteArrayList sehr Ressourcen intensiv sein kann. Es hängt dabei stark vom Anwendungsfall ab. Jedes Collection Implementation hat seine Stärken und Schwächen. Es ist daher zu empfehlen, den eigene Anwendungsfall anzuschauen und anschließend einen Blick in die Benchmarks zu werfen, um bei der Implementation die richtige Entscheidung zu treffen. Mit eigenen Tests sollte die Entscheidung später dann validiert werden.

3. Wieso ist CopyOnWriteArrayList scheinbar so langsam?

Die CopyOnWriteArrayList erstellt bei jeder Modifikation eine frische Kopie der Liste. Bei großen Listen wird dafür viel Rechenpower benötigt. Durch die spezielle Implementation sind die Daten der Liste thread-sicher.

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???)

Die meisten Listen sind nicht thread-sicher, da so weniger Rechenpower benötigt wird und meistens Modifikationen auch schneller sind. Um eine Liste thread-sicher zu machen kann CopyOnWriteArrayList genutzt werden. Hier wird bei jeder Modifikation eine frische Kopie der Liste angefertigt.

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?)

Der Code ist leider fehlerhaft und kann nicht ausgeführt werden.

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

Die remove() Methode entfernt den Eintrag aus der Garbage Collection. Die get() Methode nicht.

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!

Nein es war ein schlechter Kauf, da bei diesem Code nur ein Core verwendet wird. Die Collection wird Stück für Stück durchgegangen. Die restlichen 7 Cores werden nicht eingesetzt und bieten daher keinen Performance Vorteil.