(/)

Filtering a Java Collection by a List

Last updated: December 7, 2019

Written by: baeldung (https://www.baeldung.com/author/baeldung)

Java (https://www.baeldung.com/category/java) +

Java Collections (https://www.baeldung.com/category/java/java-collections)

Java List (https://www.baeldung.com/tag/java-list)

1. Overview

Filtering a *Collection* by a *List* is a common business logic scenario. There are plenty of ways to achieve this. However, some may lead to under-performing solutions if not done properly.

In this tutorial, **we'll compare some filtering implementations and discuss their advantages and drawbacks**.

2. Using a *For-Each* Loop

We'll begin with the most classic syntax, a for-each loop.

For this and all other examples in this article, we'll use the following class:

```
public class Employee {

    private Integer employeeNumber;
    private String name;
    private Integer departmentId;
    //Standard constructor, getters and setters.
}
```

We'll also use the following methods for all examples, for simplicity's sake:

```
private List<Employee> buildEmployeeList() {
    return Arrays.asList(
      new Employee(1, "Mike", 1),
      new Employee(2, "John", 1),
      new Employee(3, "Mary", 1),
      new Employee(4, "Joe", 2),
      new Employee(5, "Nicole", 2),
      new Employee(6, "Alice", 2),
      new Employee(7, "Bob", 3),
      new Employee(8, "Scarlett", 3));
}

private List<String> employeeNameFilter() {
    return Arrays.asList("Alice", "Mike", "Bob");
}
```

For our example, we'll filter the first list of *Employees* based on the second list with *Employee* names to find only the *Employees* with those specific names.

Now, let's see the traditional approach – **looping through both lists looking for matches:**

```
@Test
public void
givenEmployeeList_andNameFilterList_thenObtainFilteredEmployeeList_
usingForEachLoop() {
    List<Employee> filteredList = new ArrayList<>();
    List<Employee> originalList = buildEmployeeList();
    List<String> nameFilter = employeeNameFilter();

    for (Employee employee : originalList) {
        for (String name : nameFilter) {
            if (employee.getName().equals(name)) {
                filteredList.add(employee);
                // break;
            }
        }
    }

    assertThat(filteredList.size(), is(nameFilter.size()));
}
```

This is a simple syntax, but it's quite verbose, and actually quite inefficient. Simply put, it **iterates through the Cartesian product of the two sets** in order to get our answer.

Even adding a *break* to exit early will still iterate on the same order as a Cartesian product in the average case.

If we call the size of the employee list *n*, then *nameFilter* will be on the order just as big, giving us **an $O(n^2)$ classification.**

3. Using Streams and *List#contains*

We'll now refactor the previous method by using lambdas to simplify syntax and improve readability. Let's also use the *List#contains* method as the *lambda filter (/java-stream-filter-lambda)*:

```
@Test
public void
givenEmployeeList_andNameFilterList_thenObtainFilteredEmployeeList_
usingLambda() {
    List<Employee> filteredList;
    List<Employee> originalList = buildEmployeeList();
    List<String> nameFilter = employeeNameFilter();

    filteredList = originalList.stream()
      .filter(employee -> nameFilter.contains(employee.getName()))
      .collect(Collectors.toList());

    assertThat(filteredList.size(), is(nameFilter.size()));
}
```

By using the *Stream API (/java-8-streams-introduction)*, readability has been greatly improved, but our code remains as inefficient as our previous method because it's **still iterating through the Cartesian product internally**. Thus, we have the same $O(n^2)$ classification.

4. Using Streams with *HashSet*

To improve performance, we must use the *HashSet#contains* method. **This method differs from *List#contains* because it performs a *hash code* lookup, giving us a constant-time number of operations:**

```
@Test
public void
givenEmployeeList_andNameFilterList_thenObtainFilteredEmployeeList_
usingLambdaAndHashSet() {
    List<Employee> filteredList;
    List<Employee> originalList = buildEmployeeList();
    Set<String> nameFilterSet =
employeeNameFilter().stream().collect(Collectors.toSet());

    filteredList = originalList.stream()
      .filter(employee ->
nameFilterSet.contains(employee.getName()))
      .collect(Collectors.toList());

    assertThat(filteredList.size(), is(nameFilterSet.size()));
}
```

By using *HashSet (/java-hashset)*, our code efficiency has vastly improved while not affecting readability. **Since *HashSet#contains* runs in constant time, we've improved our classification to *O(n)*.**

5. Conclusion

In this quick tutorial, we learned how to filter a *Collection* by a *List* of values and the drawbacks of using what may seem like the most straightforward method.

We must always consider efficiency because our code might end up running in huge data sets, and performance issues could have catastrophic consequences in such environments.

All code presented in this article is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-collections-list-3).

2 COMMENTS Oldest

View Comments

Comments are closed on this article!

COURSES

ALL COURSES (/ALL-COURSES)
ALL BULK COURSES (/ALL-BULK-COURSES)
ALL BULK TEAM COURSES (/ALL-BULK-TEAM-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
JOBS (/TAG/ACTIVE-JOB/)