Theory: Object interning

9 minutes 0 / 5 problems solved

A large program creates a lot of objects in the memory which may affect the program performance. To reduce the required size of memory, Java can store objects in special pools where the same object can be accessed by different references. This technique works only for several widely-used classes and is known as **object interning**. First, let's try to comprehend this concept with the example of strings.

§1. Reuse of strings

The `String` class is one of the most popular types, and it is a costly one when it comes to memory space. To optimize strings storage in the heap, Java uses a string pool that shares the same objects with the same value.

As you know, it is possible to create a string using `new` operator as well as providing a value in double quotes. First, let's create two strings using the first approach:

```
String greeting1 = new String("hello");
String greeting2 = new String("hello");
```

Since `String` is a reference type, the code above creates two objects `"hello"`, and every variable refers to its own object. Therefore the result of comparing these references is `false`, and, of course, the result of comparing the actual values is `true`.

```
System.out.println(greeting1 == greeting2); // false
System.out.println(greeting1.equals(greeting2)); // true
```

This is quite expected for any reference types, not just strings.

Now we will create two more variables without using the `new` keyword.

```
String greeting3 = "hello";
String greeting4 = "hello";
```

This may surprise you, but actually, both variables refer to the same `"hello"`, which is stored in a special pool for further reuse. This is what reduces the number of objects in memory. The following code demonstrates that both variables have the same reference:

```
System.out.println(greeting3 == greeting4); // true
System.out.println(greeting3.equals(greeting4)); // true
```

Despite the fact that `String` is a reference type, the `==` operator returns `true`, because Java uses the same object from the pool. However, it is still required to compare strings using equals, your program should not depend on whether the object is in the pool or not.

§2. Reuse of wrapper class objects

Like strings, some wrapper classes also use object interning mechanism for reducing the required memory space, but it works a little bit differently.

- The `Boolean` class caches both possible constants.
- The `Character` class caches instances with values from `0` to `127` (`\u0000` to `\u007f`).
- The `Byte`, `Short`, `Integer`, and `Long` cache instances of value `-128` to `127`.
- No cached instances exist for the `Float` and `Double` wrapper classes.

Thus, some wrapper classes store certain ranges of values in special pools to reuse them when creating objects via **boxing** or **autoboxing**.

Current topic:

Object interning

Topic depends on:

✓ Boxing and unboxing

Table of contents:

↑ Object interning

§1. Reuse of strings

§2. Reuse of wrapper class objects

§3. Conclusion

Feedback & Comments

As an example, `System.out.println(i1 == i2)`; prints `true` when `i1` and `i2` have the same integer value between `-128` and `127` and will print `false` if `i1` and `i2` are outside of the `-128` to `127` range even though both are the same.

```
Long i1 = Long.valueOf("127");     // boxing
Long i2 = Long.valueOf("127");     // boxing
System.out.println(i1 == i2);      // true, Java reuses 127

System.out.println(i1.equals(i2)); // true, they have the same value inside
```

It also works with autoboxing:

```
Long i1 = 127L; // autoboxing
Long i2 = 127L; // autoboxing
System.out.println(i1 == i2);      // true
System.out.println(i1.equals(i2)); // true
```

However, it won't work if the objects are created explicitly using `new`.

```
Long i1 = new Long("127");
Long i2 = new Long("127");

System.out.println(i1 == i2);      // false, "new" creates different objects

System.out.println(i1.equals(i2)); // true, they have the same value inside
```

§3. Conclusion

Object interning is an internal feature that allowes you to occupy less memory by sharing an object in pools. It works only for immutable objects. Actually, the object interning is a very specific feature for a concrete JVM (we consider **HotSpot** as the primary reference Java VM implementation). Your programs should not depend on this optimization; do not use `==` to compare reference types even if it works correctly for pooled objects.

Report a typo

74 users liked this theory. **3** didn't like it. **What about you?**

Start practicing

Comments (1) Hints (0) Useful links (0) Show discussion