Understand the Hotspot JVM oop

Jul 21, 2019

We summarized oops and compressed oops that often appear in OpenJDK source code.

Target OpenJDK commit is jdk11u changeset `51892:e86c5c20e188`.

What is oop

oop is the "management pointer to the instance in the Hotspot VM". Hotspot VM is the OpenJDK JVM implementation name.

When an instance is created at runtime, memory is allocated from the heap space for the instance. The information to refer to it is oop. This area stores the length and elements of instance fields and arrays.

oop can be called a pointer to that area unless you consider the compression process described below.

What is CompressedOops (compressed oop)

CompressedOops is described in Hotspot JVM's Compressed OOP-Oracle . Here are the key points.

- CompressedOops background
  - An approach to reconciling the following requirements in LP64 systems:
    - I want to maintain the same heap usage as ILP32
      - If the size of oop is changed from 32-> 64 bits, the usage will increase about 1.5 times in total
        - Aside from the specific numbers, we can see that the requirements increase
    - I want to increase the maximum heap size
      - LP32 can only represent addresses up to 4 GB
- UseCompressedOops flag
  - `-XX:+UseCompressedOops`
    - oop compress
  - `-XX:-UseCompressedOops`
    - oop do not compress
  - Default is enabled for 64-bit environment
    - However, is it invalid if the heap area allocated to the JVM is large?
    - Based on the calculation method described below, maybe 32 GB or more will be invalid because it can not be expressed by compressed oop?
- Oop to compress
  - Not all oop usages use compressed values
  - For oops stored in the heap
    - Klass fields for all objects
    - All oop instance fields
    - all elements of the oop array (objArray)
  - Excluded things
    - Such as passing to a function
- Compression algorithm
  - Base and shift (usually 3) as parameters
  - Compute from compressed oop to oop
    - `oop = (compressed oop << shift) + base`
- Zero-based compression oop

  - Base, shift parameters are set to 0 if possible
    - Heap size less than 4 GB
      - base, shift to 0
    - Heap size less than ~ 30 GB
      - base is 0

Oop related classes on Hotspot JVM

Next, we will look at oop and related classes in the OpenJDK source code.

oopDesc

in src / hotspot / share / oops / oop.hpp

- C ++ overlay for any Java object
- In other words, a class that represents a Java object in the Hostspot JVM
  - (Just because of the name makes me worried about that understanding)

At the beginning of the oopDesc is information mark and klass. The mark size is 8 bytes for 64 bit environment, 4 bytes for UseCompressedOops, and 8 bytes otherwise for klass (P.3 of hospot-under-the-hood).

```
class oopDesc {
 ...
 private:
  markOop _mark;
  union _metadata {
    Klass*      _klass;
    narrowKlass _compressed_klass;
  } _metadata;
  ...
```

markOop is an alias for markOopDesc *, which we haven't been able to follow in detail, but it seems to be used for managing, for example, flags related to locks.

klass is a pointer to information about its class, which will come out later for Klass classes.

These members are followed (but not by definition) in instance fields or arrays, if any, by their length or elements.

oop

in src / hotspot / share / oops / oopsHierarchy.hpp

- C ++ accessor for Java reference
- A class that expresses oop in Hotspot JVM as it is

oop is

```
typedef class oopDesc* oop;
```

Or if it is a debug build

```
class oop {
    ...
    oopDesc *o;
    ...
    operator oopDesc*();
    ...
}
```

It has become.

narrowOop

in src / hotspot / share / oops / oopsHierarchy.hpp

- Offset instead of address for an oop within a java object
- A class that represents a compressed oop

narrowOop is simply a typedef to juint. juint is an alias for uint32_t, depending on the environment.

```
typedef juint narrowOop;
```

CompressedOops

in src / hostspot / share / oops / compressedOops.inline.hpp

- Functions for encoding and decoding compressed oops
- Define a function to encode oop to narrowOop and vice versa (decode)

Klass

in `src/hotspot/share/oops/klass.hpp`

- Offers the following two:
  - language level class object (method dictionary etc.)
  - provide vm dispatch behavior for the object
- oopDesc has a pointer to his Klass
- Perhaps Klass also manages static fields in the Java specification

InstanceKlass

in src / hotspot / share / oops / instanceKlass.hpp

- Excerpt from comments
  - VM level representation of a Java class
  - It contains all information needed for an class at execution runtime
- In other words, Java class representation in Hostspot JVM
- For example, if you need constant pool information at runtime, come here
- Inherits Klass

The data structures pointed to by InstanceKlass and Klass are apparently common, and it is common to internally cast from Klass and use InstanceKlass internally.

```
  # InstanceKlass::cast
  static InstanceKlass* cast(Klass* k) {
    return const_cast<InstanceKlass*>(cast(const_cast<const Klass*>(k)));
  }
```

Take a look at oop with a debugger

Last time, we created an environment that can debug and build OpenJDK, so let's check the value of oop from gdb. Although oop is used in various places, I think that what is easy to see in the debugger is the jarray or jobject type argument of the function defined by JNI. JNI is an interface for calling native code from Java code. For example, if you define a function that passes an array like the following here, the function that receives jarray will correspond to the C code.

```
/**
 * Check oop for an array
 */
class Sample2 {
    static {
        System.loadLibrary("sample");
    }

    public static native void callArrayLength(int[] args);

    public static void main(String[] args) {
        int arr[] = {1, 2, 3};
        callArrayLength(arr);
    }
}
```

```
#include <jni.h>

JNIEXPORT
void JNICALL Java_Sample2_callArrayLength(
  JNIEnv *env, jobject obj, jarray arr
) {
    (*env)->GetArrayLength(env, arr);
}
```

jarray is an alias for jobject, and jobject is just a pointer to an oop, so you can check the oop of the passed array by looking at this value.

(I don't really follow the source code that the jobject is a pointer to an oop. The JavaCallArguments class seems to treat jobject as an oop *.)

The sample code in `jarray arr`the oop, has defined the gdb commands, such as to confirm the oopDesc, the output is as follows.

```
$ make debug_sample2
...
gdb -x 'sample2.gdb' --args ~/workspace/openjdk11u/build/linux-x86_64-normal-server-fastdebug/jdk/bin/java  -D
...

(gdb) r
(gdb) c
...

Thread 2 "java" hit Breakpoint 1, Java_Sample2_callArrayLength (env=0x7ffff001ab90,
    obj=0x7ffff59ef880, arr=0x7ffff59ef890) at sample2.c:4
4           (*env)->GetArrayLength(env, arr);

# (1) print arr value
$1 = 0x7ffff59ef890

# (2) print memory where arr references
0x7ffff59ef890: 0x30    0x69    0x6f    0x19    0x07    0x00    0x00    0x00
0x7ffff59ef898: 0x98    0xf8    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ef8a0: 0x91    0x53    0xa2    0xcd    0xff    0x7f    0x00    0x00
0x7ffff59ef8a8: 0xf8    0xf8    0x9e    0xf5    0xff    0x7f    0x00    0x00

# (3) print memory where oop references
0x7196f6930:    0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7196f6938:    0x40    0x0c    0x00    0x00    0x03    0x00    0x00    0x00
0x7196f6940:    0x01    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x7196f6948:    0x03    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

You can see that the value of jarray corresponding to the array passed in (1) is 0x7ffff59ef890. Since this is a pointer to oop, we went to look ahead (2), and we can see from this that the value of oop `0x0000007196f6930`is (3) went to see the address pointed to by oop. The content is

- 0x7196f6930-0x7196f6937
  - mark (= 0x01)
- 0x7196f6938-0x7196f693b
  - klass (= 0x0c40)

- 0x7196f693c-0x7196f693f
    - length (= 3)
- 0x7196f6940-0x7196f694b
    - Array elements (1, 2, 3)

It becomes.

From this klass value, it is a pointer to follow the Klass corresponding to its class, but it is compressed like oop. The calculation method itself is the same as oop, but the base and shift parameter values are different, and the values are managed by a class called Universe.

```
(gdb) p Universe::_narrow_klass._base
$2 = (address) 0x800000000 ""
(gdb) p Universe::_narrow_klass._shift
$3 = 0
(gdb) p Universe::_narrow_oop._base
$4 = (address) 0x0
(gdb) p Universe::_narrow_oop._shift
$5 = 3
```

You can access Klass from this information.

```
(gdb) p ((Klass *) (((long) (0x00000c40)) + Universe::_narrow_klass._base))->_name->as_C_string()
$7 = 0x7ffff0019390 "[I"
```

But took advantage of JNI in order to confirm this time oop, to define a function for the JNI `jni.h`, `jni.cpp`is likely to be helpful in understanding the data structure of HotspotVM to the other.