How CDS Copies Class Metadata into the Archive

This page describes the object copying algorithm to be implemented in JDK-8234693 Consolidate CDS static and dynamic archive dumping code.

(Latest webrev is here. The links below currently point to the webrev, but will be updated to point to the jdk/jdk repo after JDK-8234693 is pushed).

Overview

When you dump a static (**-Xshare:dump**) or dynamic (**-XX:ArchiveClassesAtExit**) CDS archive, the VM copies eligible class metadata into a buffer, and writes this buffer to a JSA file.

The main copying algorithm is implemented in the ArchiveBuilder class. Functionalities specific to static or dynamic dumping are implemented in the StaticArchiveBuilder and DynamicArchiveBuilder subclasses.

The copying algorithm basically starts scanning the class metadata using MetaspaceClosure, starting from the set of roots provided by **StaticArchiveBuilder::iterate_roots()** or **DynamicArchiveBuilder::iterate_roots()**. The copying is done in the following steps:

Step 1. Determine Iteration Order

When dumping the static archive, we want the contents to be deterministic (see JDK-8241071). Basically, if you run "**java -Xshare:dump**" twice, you should get two **classes.jsa** files that are bit-by-bit identical.

Note that **ArchiveBuilder::iterate_roots()** scans some hashtables of symbols and classes. Some of these hashtables uses hashkeys that may be derived from (a) random numbers (e.g., **Symbol::identity_hash()**), or (b) addresses that may vary from run to run (e.g., **SystemDictionaryShared::dumptime_classes_do()**). Therefore, the iteration order is not stable.

To get a stable iteration order, we use **ArchiveBuilder::gather_klasses_and_symbols()**, which iterate the class metadata with **ArchiveBuilder::iterate_roots()** to discover all Symbols and Klasses that are eligible for archiving. It then sorts the Symbols and Klasses using a stable sorting order into the arrays **ArchiveBuilder::symbols()** and **ArchiveBuilder::klasses()**.

See comments in **ArchiveBuilder::gather_klasses_and_symbols()** for more detail.

Step 2. Categorize Read-only and Read-write Objects

This step is implemented by **ArchiveBuilder::gather_source_objs()**, which iterates the metadata with **ArchiveBuilder::iterate_sorted_roots()**. All objects that are eligible for copying are entered (by reference) into **ArchiveBuilder::_rw_src_objs** or **ArchiveBuilder::_ro_src_objs**, depending on whether they are read-write or read-only.

In this step, we also remember the locations of all the pointers in the source objects using **ArchiveBuilder::_ptrmap**. This is used later in embedded pointer relocation (see below).

For simplicity, let's assume we have only read-write objects to copy. The source objects look like this.

```
address       value      field type    field name
   // object "foo" @ 0x100
@0x100:       0x0        intx          Foo::intxValue1
@0x108:       0x208      Bar*          Foo::barPtrA
@0x110:       0x1        intx          Foo::intxValue1
@0x118:       0x200      Bar*          Foo::barPtrB


   // objects ineligible for copying (skipped) ...


   // object "bar1" @ 0x200
@0x200:       0x11       intx          Bar::someField
   // object "bar2" @ 0x208
@0x208:       0x22       intx          Bar::someField
```

Let's assume that ArchiveBuilder::iterate_roots() points to a single object (**foo** at 0x100). When we start iterating with foo, we will discover **bar2** first (when we scan **foo->barPtrA**), and then **bar1** (when we scan **foo->barPtrB**). After scanning **bar2** and **bar1**, we stop because we have found no more new objects to scan.

At this point, **ArchiveBuilder::_rw_src_objs** will contain the following information:

```
_rw_src_objs->at(0)->obj()           == 0x100    foo
_rw_src_objs->at(0)->size_in_bytes() == 32       sizeof(Foo)
_rw_src_objs->at(1)->obj()           == 0x208    bar2
_rw_src_objs->at(1)->size_in_bytes() == 8        sizeof(Bar)
_rw_src_objs->at(2)->obj()           == 0x200    bar1
_rw_src_objs->at(2)->size_in_bytes() == 8        sizeof(Bar)
```

Also, **_rw_src_objs->_ptrmap** essentially remembers that:

```
There is a pointer at 0x108   (&foo->barPtrA, offset == 0x08)
There is a pointer at 0x118   (&foo->barPtrB, offset == 0x18)
```

Step 3. Copy Source Objects into Output Buffer

This is done by **ArchiveBuilder::dump_rw_region()** (and **ArchiveBuilder::dump_ro_region()**). The copying is fairly straightforward: all the objects that should be copied into the RW region are already stored in the array inside _rw_src_objs, along with their sizes. So we just linearly allocate the copies in **ArchiveBuilder::_rw_region**, and copy the contents of the source objects to their copies using **memcpy()**. See **ArchiveBuilder::make_shallow_copy()** for details.

With the above example, if **_rw_region** starts at 0x400, the copies will look like this:

```
address       value      field type    field name
   // copy of object "foo" @ 0x100
@0x400:       0x0        intx          Foo::intxValue1
@0x408:       0x208      Bar*          Foo::barPtrA      <<< still points to source object of "bar2"
@0x410:       0x1        intx          Foo::intxValue1
@0x418:       0x200      Bar*          Foo::barPtrB      <<< still points to source object of "bar1"
   // copy of object "bar2" @ 0x208
@0x420:       0x22       intx          Bar::someField
   // copy of object "bar1" @ 0x200
@0x428:       0x11       intx          Bar::someField
```

Note that the pointers that are embedded in the copies are still pointing to the source objects.

While copying, we also update **ArchiveBuilder::_src_obj_table** to map the source objects to their copies:

```
source foo  @ 0x100   -> copy of foo  @ 0x400
source bar1 @ 0x200   -> copy of bar1 @ 0x428
source bar2 @ 0x208   -> copy of bar2 @ 0x420
```

Step 4. Relocate Embedded Pointers

During this step, we update the pointers embedded in the copies. See **ArchiveBuilder::relocate_embedded_pointers()** for details. Here's an illustration of how it works.

```
+ Iterate over all the source objects: so we visit foo, bar2 and bar1
+ When we visit foo @ 0x100, we find out (using _rw_src_objs->_ptrmap) that it
  has embedded pointers at offsets 0x08 and 0x18.
+ We find out that the copy of foo is at 0x400 using _src_obj_table
+ So we know that there are two embedded points inside the copy of foo:
    @(0x400 + 0x08)  = old pointer value of 0x208
    @(0x400 + 0x18)  = old pointer value of 0x200
+ Using ArchiveBuilder::_src_obj_table, we find out that:
    source object at 0x208 is copied to 0x428
    source object at 0x200 is copied to 0x420
+ So we can update the pointers like this:
    @(0x400 + 0x08)   := 0x420
    @(0x400 + 0x18)   := 0x428
```

When this step is finished, the output buffer looks like this:

```
address       value      field type    field name
   // copy of object "foo" @ 0x100
@0x400:       0x0        intx          Foo::intxValue1
@0x408:       0x420      Bar*          Foo::barPtrA      <<< points to copy of "bar2"
@0x410:       0x1        intx          Foo::intxValue1
@0x418:       0x428      Bar*          Foo::barPtrB      <<< points to copy of "bar1"
   // copy of object "bar2" @ 0x208
@0x420:       0x22       intx          Bar::someField
   // copy of object "bar1" @ 0x200
@0x428:       0x11       intx          Bar::someField
```

When we update the embedded pointers, we also use **ArchivePtrMarker::mark_pointer()** to mark the location of all the embedded pointers. This information is used for relocating the entire archive. E.g., if we want to relocate the output buffer from 0x400 to 0x500, we need to update the embedded pointer of 0x428 to 0x528. See **VM_PopulateDumpSharedSpace::relocate_to_requested_base_address()** for more info.

Performance Considerations

In the previous implementation of static dump, we used MetaspaceClosure for 3 times:

- copy the RW objects
- copy the RO object
- relocate embedded pointers

However, MetaspaceClosure is slow. In the new implementation, we using MetaspaceClosure only twice (in **Steps 1** and **Step 2**). During Step 2, we remember the size of all the source objects, as well as the location of the embedded pointers. This eliminates the use of MetaspaceClosure in the subsequent steps for making copies and relocating embedded pointers. The resulting code is faster, and also easier to understand (no need to think about recursion during copying, etc).

The previous implementation of dynamic dump used MetaspaceClosure even more. As a result, it gets a more pronounced speed up from the new implementation.

Here are the elapsed time of the following test cases (which archive more than 20000 classes) using fastdebug build:

	LotsOfClasses.java	**DynamicLotsOfClasses.java**
Old	42.655 sec	67.014 sec
New	37.027 sec	34.974 sec