Selectively Preserving Pre-initialized Java Classes

—

Preserving Initialized Static Field Values by Caching Java Heap Object Subgraphs

Jiangli Zhou/5.15.2020

Objective

- Provide a general solution for pre-initializing Java classes and preserving the processed Java heap objects as part of the CDS (Class Data Sharing) archive image for additional runtime savings
  - Design doc (covers pre-linking and pre-resolution in addition to pre-initialization)

Benefits

- Improve performance by avoiding executing Java bytecode in `<clinit>` at runtime
  - Faster server startup, save CPU usages
  - May reduce GC and JIT compiler overhead
- Make it easier to do class pre-initialization for developers

Goals and Non-Goals

- Provide a flexible and general solution to cache pre-initialized Java classes loaded by the builtin class loaders
- Work with static archiving (-Xshare:dump) that is currently available in JDK 11
- Extendable for the dynamic archiving (currently available in OpenJDK 14), but is not the current focus
- Supporting classes loaded by user defined class loaders is not a goal in the current scope
    - Address separately

What is Class Initialization

- Java VM Specification §5.5:
  - "*A class or interface has at most one class or interface initialization method and is initialized by the Java Virtual Machine invoking that method (§5.5). A method is a class or interface initialization method if all of the following are true:*
    - *It has the special name <clinit>.*
    - *It is void (§4.3.3).*
    - *In a class file whose version number is 51.0 or above, the method has its ACC_STATIC flag set and takes no arguments (§4.6).*" §2.9.2
  - "*Initialization of a class or interface consists of executing its class or interface initialization method (§2.9.2).*"

Existing Selected Static Field Pre-initialization in JDK

- Caching Java Heap Subgraphs introduced in OpenJDK 12 supports pre-initializing selected static fields and preserving the initialized values
  - Backported in JDK 11
  - All Java objects reachable from the initialized static field (reference type) are archived at dump time
    - Copying objects to the Java heap archive regions
    - Updating pointers to copied objects
  - At runtime during the initialization of a class with archived static fields, the archived values are retrieved and installed back to the mirror object (`j.l.Class` instance of a loaded class)

Limitations of the Existing Static Field Pre-initialization

- Does not support application class well
- Only support a small set of selected static fields in JDK classes
  - VM code maintains a list of hard-coded class and field names to indicate which static fields for pre-initializing and archiving, e.g.:

```
{"java/lang/Integer$IntegerCache",  "archivedCache"},
{"jdk/internal/module/ArchivedModuleGraph",
"archivedModuleGraph"},
```

- Archived static field values are stored separately from the mirror object
  - Runtime needs to store the values back into the static fields

Limitations of the Existing Static Field Pre-initialization (Continued)

- `Final` must be removed from static field declaration to avoid compiler error
- Field declarator must not include the variable initializer
- Java code needs to call

  `jdk.internal.misc.VM.initializeFromArchive(class)` in class static initializer (`<clinit>`) to retrieve archived field values at runtime

```
static final class ListN<E> extends AbstractImmutableList<E>
        implements Serializable {
    static @Stable List<?> EMPTY_LIST;
    static {
        VM.initializeFromArchive(ListN.class);
        if (EMPTY_LIST == null) {
            EMPTY_LIST = new ListN<>();
        }
    }
```

Overview of the General Class Pre-initialization and Caching Proposal/Design

- Support selectively preserving pre-initialized JDK & application classes and individual static fields
  - Focus on class level support initially, general support for individual fields (partial class pre-initialization) can be addresses when needed
  - Not all classes (and static fields) are suitable for caching pre-initialized values
    - Runtime context dependency (e.g. computed field value is unique to a particular runtime environment, calling `System.currentTimeMillis`)
    - Static field initialization creates Thread, file descriptor, etc
    - Register native library, initialize field offsets for JNI access, etc
  - Opt-in approach with annotation (adopted in this design)
    - Easy to use and maintain; Minimal Java source modification; Minimal runtime overhead

Alternatives

- Class and field list
  - Desire for improving usability
- Interface (`java.io.Serializable` or a new interface?)
  - Can only support classes, but not individual field
  - Not future-proof against implementation changes, not better than the annotation approach
  - Runtime overhead with an extra interface in the class hierarchy
- Static analysis
  - May not identify all cases that are not suitable for preserving pre-initialized values
  - Expensive to develop, no resources available for the required work

Proposed Annotations

- ***`@jdk.internal.vm.annotation.Preserve`***
  - A runtime visible annotation (see JVMS, §4.7.20 The RuntimeVisibleTypeAnnotations Attribute) to tag a JDK class or individual static fields for pre-initialization
- ***`@jdk.internal.vm.annotation.DontPreserve`***
  - A runtime visible annotation to tag a JDK class that should not be pre-initialized
  - Optional, help prevent others from annotating a class using `@Preserve` without realizing potential issues
- ***`@com.google.common.annotations.Preserve`***
  - A runtime visible annotation to tag an application class or individual static fields for pre-initialization

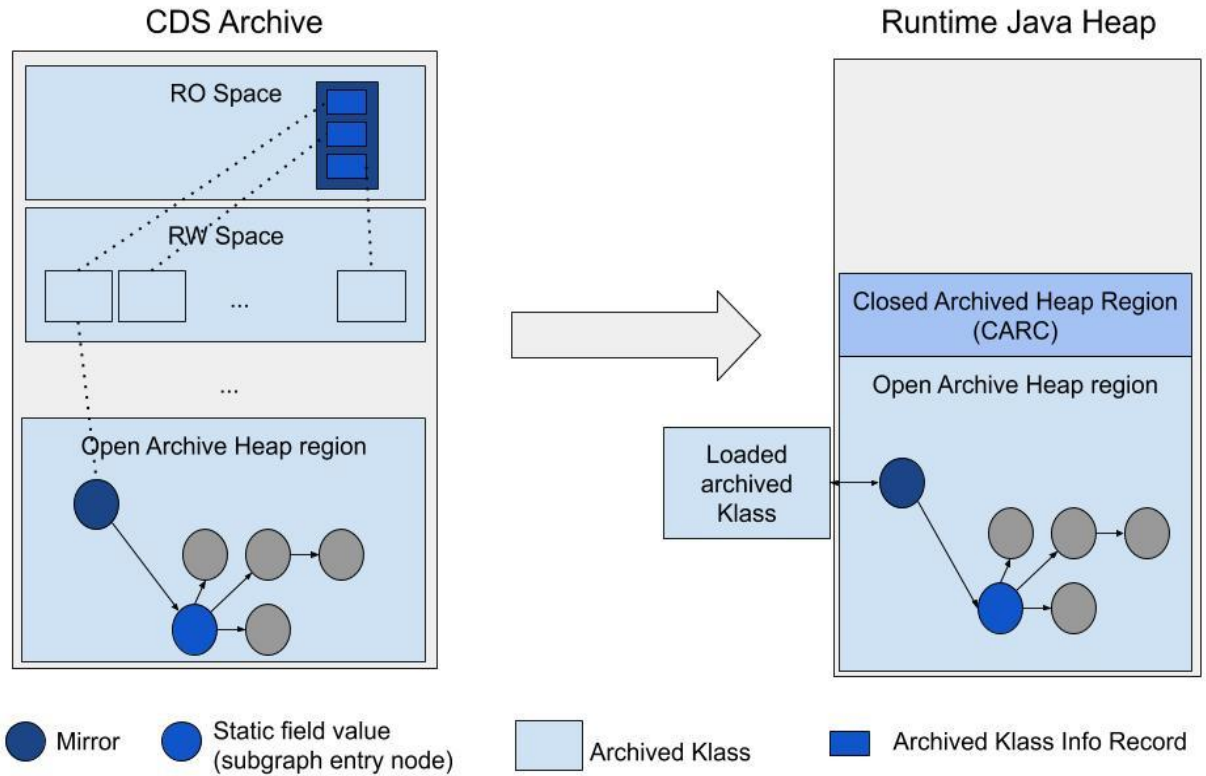
Class Level @Preserve

- The previous ListN example can be simplified with this design:

```
@Preserve
static final class ListN<E> extends AbstractImmutableList<E>
        implements Serializable {
    static final List<?> EMPTY_LIST = new ListN<>();
```

```
static final class ListN<E> extends AbstractImmutableList<E>
            implements Serializable {
        static @Stable List<?> EMPTY_LIST;
        static {
            VM.initializeFromArchive(ListN.class);
            if (EMPTY_LIST == null) {
                EMPTY_LIST = new ListN<>();
            }
        }
```

Class Pre-initialization and Preserving Architecture



Class Initialization Phase At Archive Dump time

- Some of the loaded classes may be implicitly initialized, as a result of executing Java code to load classes from the *classlist*
- Explicitly initialize the rest of the classes annotated with @Preserve
  - After loading all classes from the *classlist* and before copying & relocating class metadata to archive spaces
  - Call `InstanceKlass::initialize()`to do class initialization
    - Follows class initialization procedure described in JVM specification §5.5

Object Subgraph Checking Phase at Archive Dump time

- Check is done after the class metadata copying and relocation step
- A static field should not be archived if any of the following types of objects is found in its subgraph
  - Non-mirror java.lang.Class objects
  - ClassLoader objects
  - java.security.ProtectionDomain objects
  - java.lang.Thread objects
  - Runnable objects
  - java.io.File objects
  - TBD

Static Field Value Preserving Phase at Archive Dump time

- Archiving starts from a reference type static field (a root object)
  - Follow references and walk all objects within the subgraph, and copy objects to the archive heap regions
  - Update pointers to the archived objects
- Support mirror objects within archived subgraphs
  - If a mirror object is encountered, the VM archives the mirror but stops following references from it
  - Allow more classes for pre-initialization and caching
    - E.g. `java.lang.Integer`, `java.lang.Long`, etc

Static Field Value Preserving Phase at CDS Dump time (Continued)

- All archived static field values from a JDK class are preserved within the class' archived mirror object
  - Include all primitive types and reference types
  - Runtime does not need to store the values back to the fields
  - Reachable subgraph objects become live when the mirror becomes live at runtime
  - The separate per-class record is still needed to store the dependent class list of the current class
    - The dependent classes should be initialized before the current class

What Happens at Runtime

- Three cases with different optimization levels, all avoid executing `<clinit>` at runtime
  - An archived class can be set to `fully_initialized` state immediately when loaded and restored at runtime iff following are true (most optimized case)
    - All static fields are primitive types or `j.l.Object` and `j.l.String` types (more types may be allowed)
    - Is an interface, or the direct superclass is `j.l.Class`, or an archived class that can be set to `fully_initialized` at restore time

What Happens at Runtime (Continued)

- Otherwise, If a mirror contains preserved static field values (JDK classes)
  - Class is set to `linked` state when it is loaded and restored
  - When the class initialization is triggered
    - Supertypes are initialized first
    - VM retrieves the archived class info record, all dependent classes in the record are initialized
    - Then class can be set to `fully_initialized` state
- Otherwise, all work described in above case is done, and static field values are retrieved from archived class record and stored back into the mirror object (application classes)
  - The subgraph entry objects for static reference fields are materialized explicitly
    - Objects become live and can be found by GC

Support for Application Classes

- Limiting initial pre-initializing support for application classes (loaded by the system class loader) with only
  - primitive types or String type
  - static final fields
  - static fields with `@stable`
- Static field values are stored separately from the mirrors for application classes
  - Not stored within mirrors

GC Considerations

- Runtime Java Heap regions containing mapped archived objects are pinned
  - Objects are not moved and collected
- Archived objects in open archive heap regions (OARC) are initially dormant (not reachable) until they become reachable/live
- A live archive heap object in OARC stays alive (does not become dormant again) with all existing use cases in JDK

GC Considerations (Continued)

- With broader support, usages may include cases where a live archive object may become unreachable
  - Not common cases but possible
  - If runtime execution adds external references before the object becomes unreachable, the object may contain stale references after the externally referenced objects are collected
    - No issue for tools and applications in prod during normal execution
    - Looks like is not an issue for some of heap dumper or monitoring agents
    - However, should be addressed as a complete solution for preserving pre-initialized classes and static fields