Java Class Pre-resolution, and Pre-initialization

Status: To Be Reviewed
Authors: jianglizhou@google.com,
Last Updated: 2020-5-15

Objective

During the execution of a Java application, the used Java classes must be loaded, linked (JVMS [1] §5.4), resolved (optionally, JVMS [1] §5.4.3) and initialized (JVMS [1] §5.5). These operations are relatively expensive, and are repeated for the same set of classes across different JVM processes when running the same Java application. Some of the JDK classes and application library classes may be commonly used for different Java applications. The loading/linking/resolution/initialization work is repeated for those classes in every JVM process running those applications.

Class Data Sharing (CDS), supports archiving (caching) loaded class metadata and some of the selected Java heap objects in an archive image (.jsa). When that is being used, it avoids repeated classloading work for Java applications and improves performance. However, the resolution, initialization and some of the linking work are still repeated during application runtime. The total runtime cost for those operations can be as high as class loading and verification. A measurement done by compiling HelloWorld.java using Javac compiler/JDK 11 on a linux-x64 machine indicates the total cost for class initialization alone can be up to 400ms. The cost can be substantially higher for large Java applications.

This design doc proposes extending the CDS archiving mechanism to pre-resolve and pre-initialize classes, and preserve the processed data as part of the archive to achieve additional savings.

Goals

- Improve application performance running on JVM when archive is enabled.
- Provide a flexible and general solution to achieve class pre-initialization and pre-resolution that works for both JDK classes and application classes loaded by the builtin class loaders.
- Support a mechanism for G1 archive heap regions with broader usages for application class' objects. The heap object archiving technique can be applied to cases where an archived static field may be mutable after class initialization.
- Works on top of the static archiving (-Xshare:dump) initially. The design and implementation should be considered and extendable for the dynamic archiving scenario, which simplifies CDS usage.

Non-Goals

- Supporting application classes loaded by user defined class loaders is not a goal in the current scope in this design. Classes loaded by user defined class loaders will be addressed separately with a solution built on top of the current design.
- Dynamic archiving [5] (currently available in OpenJDK 14) is not in the current scope. The pre-resolving and pre-initialization work can be extended to dynamic archiving in later phases.

Background

Below are some high-level descriptions about class linking and initialization in JVM specification.

"Linking is the process of taking a class or interface and combining it into the run-time state of the Java Virtual Machine so that it can be executed. Initialization of a class or interface consists of executing the class or interface initialization method (§2.9)." [1] §5

"• *A class or interface is completely loaded before it is linked.*
• *A class or interface is completely verified and prepared before it is initialized.*" [1] §5.4

There is an existing framework [2] [3] introduced in OpenJDK 12 for Hotspot JVM that supports pre-initializing selected static fields within JDK library classes and preserving the initialized values. The design of the mechanism is described in Caching Java Heap Objects [4].

For a pre-initialized and preserved static field, all Java objects reachable from the static field form a complete and closed graph. The entire graph of objects are archived in the Java heap archive regions. At runtime, during the initialization of a class with archived static fields, the preserved values are retrieved and installed back to the class mirror object by invoking `jdk.internal.misc.VM.initializeFromArchive(class)` method. The mechanism speeds up the runtime execution by bypassing the bytecode execution that initializes those related fields. It has been applied to a set of selected JDK library classes' static fields, whose initialization cost are relatively high, for example the system modules graph (`jdk/internal/module/ArchivedModuleGraph`) and Integer cache (`java/lang/Integer$IntegerCache`).
Following is an example of how `ImmutableCollections$ListN` pre-initializes the `EMPTY_LIST` field:

```
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<>();
            }
        }
...
```

However, the current pre-initialization solution can only be applied to restricted usages:

- The static fields must be from JDK classes.
- The list of the fields are hard-coded in the VM.
- Curtain types of field manipulation are not allowed at runtime after initialization, due to unimplemented GC support.

There are also limitations to the existing approach:

- `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.

The design for general class pre-initialization support is built on top of the existing solution and expands the usage for Java application classes and more JDK classes.

Overview

This proposal utilizes the existing CDS archiving mechanism by doing more pre-processing of Java classes at archive generation time and minimizing the runtime overhead of class initialization and resolution. At dump time, all archived classes are fully linked. Targeted constant pool references are resolved and the resolved states are preserved within the archived data. Annotated classes and static fields are initialized at dump time and the initialized results are cached in the archive. At runtime, the JVM no longer needs to resolve a constant pool reference if the specific reference is already resolved ahead-of-time (at dump time). Class initialization cost can be reduced with more pre-initialized static fields.

Detailed Design

Setting JDK Classes in ‘linked’ State at Restore Time

“*Linking a class or interface involves verifying and preparing that class or interface, its direct superclass, its direct superinterfaces, and its element type (if it is an array type), if necessary.*” [1] §5.4

When linking a class, `InstanceKlass::link_class_impl` first links the super types (all super classes and super interfaces) of the current class. For the current class, it then verifies and rewrites the bytecode, checks loader constraints, links methods, initializes the itable and vtable, and sets the current class to 'linked' state.

At archiving dumping time, the VM loads JDK and application classes using their corresponding class loaders. Currently, application classes for user defined class loaders are handled differently and are not considered within this design. The loaded classes are verified and fully linked before the data is persisted into an archive file. All class states within the archive are reset to ‘allocated’ state.

When loading an archived class at runtime, `SystemDictionary::load_shared_class` makes sure all the super types in the class hierarchy for the current class are loaded first. If not, the archived class is not used and the JVM then dynamically loads the requested class from the JAR file. That ensures the runtime class hierarchy is the same as dump time.

When an archived class is 'loaded' from the archive, the class is restored (see `InstanceKlass::restore_unshareable_info`). At the end of the restoration, all methods are linked and an archived JDK class is effectively in 'linked' state. As a result, JVM can safely set the class state to 'linked' after restoration. That avoids the work for re-iterating the super types in the class hierarchy at a later point. It is also a prerequisite for class pre-initialization per JVM specification:

“*Prior to initialization, a class or interface must be linked, that is, verified, prepared, and optionally resolved.*” [1] §5.5

Pre-resolving Constant Pool Entries

“*A run-time constant pool is a per-class or per-interface run-time representation of the constant_pool table in a class file (§4.4). It contains several kinds of constants, ranging from numeric literals known at compile-time to method and field references that must be resolved at run-time.*” [1] §2.5.5

“*Many Java Virtual Machine instructions - anewarray, checkcast, getfield, getstatic, instanceof, invokedynamic, invokeinterface, invokespecial, invokestatic, invokevirtual, ldc, ldc_w, ldc2_w, multianewarray, new, putfield, and putstatic - rely on symbolic references in the run-time* *constant pool. Execution of any of these instructions requires resolution of the symbolic reference*.

Resolution is the process of dynamically determining one or more concrete values from a symbolic reference in the run-time constant pool. Initially, all symbolic references in the run-time constant pool are unresolved.” [1] §5.4.3

At archive static dumping time (using a classlist), only some of the class loading and module initialization Java code is executed. The number of resolved constant pool references due to bytecode execution is minimum. All CONSTANT_String references are eagerly and explicitly resolved by JVM. All remaining constant pool references in archived class are unresolved. Currently, all constant pool entries except CONSTANT_String references are reset to an unresolved state when archived data is persisted.

Presolving constant pool entries can improve runtime performance, and also provides opportunities for AOT to generate more efficient code.

Pre-resolve CONSTANT_Class references

Well-Known Classes

During Universe::genesis at JVM startup time, the VM loads a set of 'well-known' system classes, such as `j.l.Object` and `j.l.String`. It also creates all one dimensional primitive array klass. Those work are done very early during the JVM initialization, before any bytecode execution. Therefore, it is safe to pre-resolve any `CONSTANT_Class` entries to a well-known class or its supertypes, because it is guaranteed that the referenced class is already loaded when the constant pool entry is accessed.

During archive dump time, the JVM iterates through all constant pool entries for archived class and resolves the `JVM_CONSTANT_UnresolvedClass` entries to the well-known classes and their supertypes.

All resolved class references to the following are preserved in the archived class data:

- a well-known klass (see `SystemDictionary::resolve_well_known_classes`) or one of its supertypes
- current klass
- a super type of the current klass
- one dimensional primitive array klass

All other resolved class references are still removed/cleared at dump time.

TODO: Pre-resolve non-well-known class constants.

Pre-resolving Field Constants

“*A class or interface C may be initialized only as a result of:*

• The execution of any one of the Java Virtual Machine instructions new, getstatic, putstatic, or invokestatic that references C (§new, §getstatic, §putstatic, §invokestatic).

...” [1] §5.5

Class initialization is required by field resolution.

TODO: Supporting pre-resolving field constants will be investigated after productizing the support for pre-initialization JDK classes.

Pre-resolving Method Constants

Class initialization is required by method resolution.

TODO: Supporting pre-resolving method constants will be investigated after productizing the support for pre-initialization JDK classes.

Pre-initializing Class

“*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).” [1] §2.9.2

“*Initialization of a class or interface consists of executing its class or interface initialization method (§2.9.2).*” [1] §5.5

This design proposes a flexible and JVM spec compliant solution:

- Pre-initialize a selected class fully ahead-of-time and preserve all initialized static values within the class mirror object (the `j.l.Class` instance representing the class) if all static fields within the class are suitable (please see the Subgraph Checking Phase section for static fields that are not good candidates) for pre-initialization.

- Partially pre-initialize targeted static field(s) ahead-of-time and record the initialized values separately from the mirror object, if some of the static fields within the class are not suitable for pre-initialization. Currently for JDK classes, that can already be supported by the existing feature with Caching Java Heap Objects [4]. The drawback of the current solution is that the JDK class and field names must be hard-coded in the VM and cannot support application classes. If there is additional benefit, a solution that avoids the hard-coded names can be utilized to support partial class pre-initialization.

The existing Caching Java Heap Objects [4] mechanism provides a framework that is required to implement both above. The proposed solution is built on top of that.

Pre-initialization Marker

This design is currently built on top of the static archive dumping without executing Java applications (*-Xshare:dump*).

A class static initializer with runtime dependencies or accessing native resources is not suitable for pre-initialization:

- Register native library, initialize field IDs for JNI access, etc
- Creates file descriptor(s) or thread(s), etc (such cases can be detected by dump time subgraph checks)
- Has dependendency on current date, time (calling into `System.currentTimeMillis` for example), or other runtime states that may change in different execution environments

The majority of the JDK classes either have static initializers that do not contain the above cases and can be safely pre-initialized, or do not have `<clinit>` methods. Only a small number of JDK classes that have static initializers may contain above cases and thus are not good candidates for pre-initialization. In this design, an opt-in solution is proposed for both JDK classes and application classes, at both class and field levels.

There are different alternatives for specifying a class or field for pre-initializing:

- Using a class-list or field-list as input
- Annotating a class/field at Java source level
- Using interface (how about `java.io.Serializable` or a new interface? Cannot support field level partial class pre-initialization)
- Static analysis

The annotation approach is adopted in this design based on following considerations:

- Developers have better control
- Easier to maintain and use
- Minimal runtime overhead

The following annotation names and packages are not finalized and may change based on feedback from OpenJDK community:

- ***@jdk.internal.vm.annotation.Preserve***: A runtime visible annotation (see [1], §4.7.20 The RuntimeVisibleTypeAnnotations Attribute) that is used to tag a JDK class or individual static fields for pre-initialization.

- ***@jdk.internal.vm.annotation.DontPreserve***: A runtime visible annotation used to tag a JDK class that should not be pre-initialized. This is optional, and any JDK classes without *`@Preserve`* is not pre-initialized (explicitly) and preserved at dump time. However, a JDK class may be annotated with *`@DontPreserve`* to prevent others from annotating it using `@Preserve` without realizing potential issues associated with pre-initializing that particular class.

- ***@<pkg_name_tbd>.Preserve:*** A runtime visible annotation that is used to tag an application class or individual static fields for pre-initialization.

The opt-in solution is a safer (compared to an opt-out solution) and flexible solution that can be expanded to include more JDK classes for pre-initialization when needed. Annotations within source code are used as an ‘indicator’ for JVM to identify candidates for pre-initialization, the VM also performs explicit checks at dump time to ensure well-known unsafe cases are not preserved in the archive. Please see sections below for more details on dump time checks.

Static analysis may be implemented to detect if a static initializer calls JNI native to register native functions, initialize field IDs or calls `System.currentTimeMillis` and etc. However static analysis may still not cover all cases thoroughly. A developer driven solution with annotations to mark classes for pre-initialization at Java source code level is more robust and flexible.

Pre-initialing JDK Classes and Preserving Static Field Values

Class level @Preserve

The earlier `ImmutableCollections$ListN` class example can be simplified with the use of @Preserve to opt-in for pre-initializing and preservation.

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

When a JDK class is annotated with *@Preserve*, a run-time visible annotation with field descriptor that denotes the annotation is included in the class file's attribute table. At dump time, the VM collectes such classes into a hash table for processing. The VM also looks for *@DontPreserve* annotation and ensures the static field values from those classes are not preserved in the archive.

The dump-time pre-initialization and preservation process for JDK classes involves initializing classes from the collected table, checking static fields and identifying any values that are not suitable for pre-initializing/preservation, excluding those identified classes, copying preservable static field values into archive heap region(s).

Class Initializing Phase

At dump time, some of the loaded classes may be implicitly initialized, as a result of executing Java code to load classes from the *classlist*. The VM needs to explicitly initialize the rest of the classes in the collected table. The operation is performed after loading all classes from the *classlist* and before copying & relocating class metadata to archive spaces.

Explicit class initialization is done via calling `InstanceKlass::initialize()` and follows the procedure described in JVM specification [1] section §5.5.

Subgraph Checking Phase

Java heap object archiving process is done after class metadata copying and relocation step. All directly and indirectly reachable Java heap objects from a reference type static field form a complete subgraph. A static field should not be archived if any of the following types of objects are 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

Before initialized static field values are archived, the VM iterates all classes in the collected table to walk all reachable objects from every reference type static field within a class using depth-first search algorithms and makes sure unsuitable values (please see above) are not archived. During the search, if a non-preservable object is found, the state is populated upwards to all the objects (nodes) that directly or indirectly reference the non-preservable object. As a result, it avoids traversing a subgraph again if the same object that contains a non-preservable subgraph is encountered again during the process.

A class is excluded for preserving if any of the reference type static fields is found not archivable/prevserable.

Static Field Values Preserving Phase

The static field value preserving is built on top of the existing algorithm that supports selective static field archiving [3]. During the process, the VM starts from a root object, which is the value of a reference type static field (referenced from a mirror object) and is also the entry point of a subgraph formed by all directly and indirectly reachable objects from the root. The VM follows references and walks all objects within the subgraph using depth-first search algorithms, and copies objects to the archive heap region. Pointers (references) within the archived objects are updated as part of the process.

This design enhances the current subgraph archiving technique by allowing mirrors (`java.lang.Class` instances that represent loaded classes on the Java side). During the subgraph walking and archiving process, if a mirror object is encountered the VM archives the mirror but stops following references from it. Mirrors can be safely included in an archived subgraph because they are singleton objects representing loaded classes within the system. Dump time resets mirror's instance fields to default values (see default value descriptions for primitive types and reference type in JVM spec [1] §2.3 & §2.4 sections), which guarantees no non-preservable objects are included in the subgraph. Archived mirrors are restored and materialized during runtime class loading time for the corresponding archived classes. The mechanism built in the static field archiving ensures all dependent classes (the class types of the objects included within a subgraph) are initialized before an archived static field value is materialized at runtime. That guarantees mirrors within any of the materialized subgraph are fully initialized and usable when being accessed at runtime. Supporting mirror objects enables more classes for pre-initialization. Such classes include `java.lang.Integer`, `java.lang.Long`, etc.

Non-mirror `java.lang.Class` objects cannot be preserved in the current scope as they contain references to `java.lang.ClassLoader` object, `java.security.ProtectionDomain` object, etc, which contain data that's specific to a runtime execution environment and should not be preserved. Further research is needed to seek a solution that can support archiving ClassLoader objects.

The existing static field subgraph archiving resets all local static fields within a mirror object and stores the archived field values (reference types only) in a separate per-class record. With this design, all archived static field values including primitive types and reference types from a class are preserved within the class' archived mirror object. Runtime does not need to store the values back to the fields. The separate per-class record is still needed to store the dependent classes of the current class.

Archived Class State Flags

Two archived class states are introduced in this design to facilitate further runtime optimizations during archived class restoration and initialization:

- **`is_pre_initialized`**
- **`is_preserved`**

The is_pre_initialized Flag

An archived class is set with the `is_pre_initialized` flag, iff both of the following conditions are true for the class:

- The static fields are primitive types or special reference types. Currently those special reference types are limited to `j.l.Object` and `j.l.String` types, which are initialized before other classes.
- The current class is an interface, or the direct superclass is `j.l.Class` or an archived class with `is_pre_initialized` flag set.

At runtime when an archived class with `is_pre_initialized` flag is loaded, the class can be set to `fully_initialized` state, which avoids the overhead of traversing supertypes in the normal initialization process.

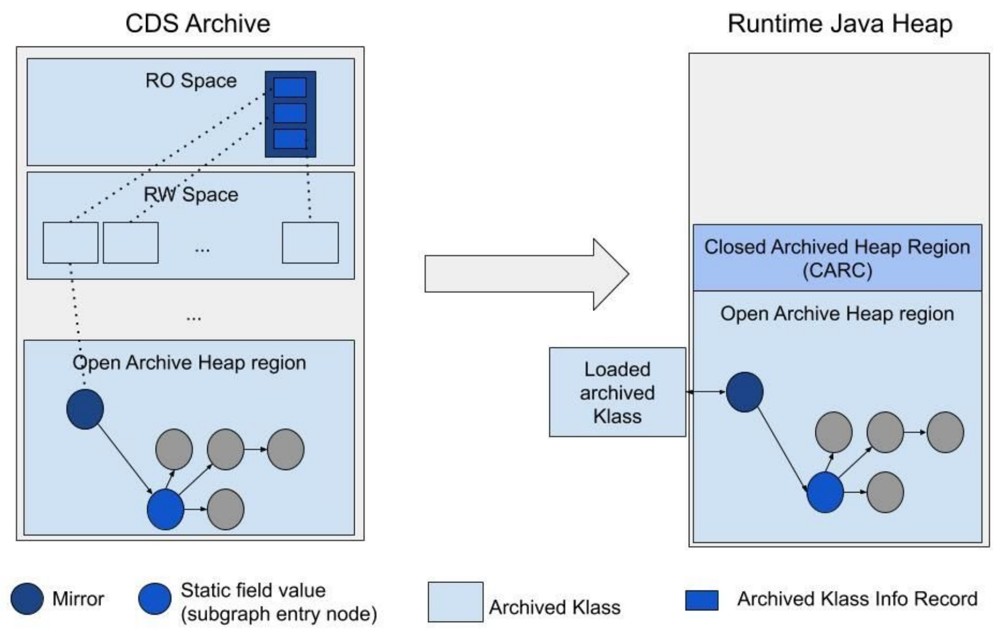
The is_preserved Flag

The `is_preserved` flag is used to indicate that an archive class contains preserved static field values.

Runtime Process for Archived Classes with Preserved Static Fields

At runtime when an archived class is loaded and restored, if its `is_pre_initialized` flag is set, the class is set to `fully_initialized` state when restore is done. No additional initialization work is needed. That is the most optimized case.

If the class does not have the `is_pre_initialized` flag set, it is set to `linked` state when restore is completed. When the class initialization is triggered during Java code execution (see JVM specification [1] §5.5, the VM initializes all its supertypes first. Then retrieves the archived class record. All dependent classes listed in the record are initialized. If the `is_preserved` flag is set, no additional work is needed to populate the field values. Otherwise, the values for all static fields are retrieved from the record and stored back into the mirror object. The entry objects (for a subgraph) of the reference type static fields are materialized. The `<clinit>` method is not executed and the class is set to `fully_initialized` state.



Field level @Preserve

Field level annotation for pre-initializing and preservation is not supported in the design. It is unclear how much additional benefits will be delivered by supporting partial pre-initialization if some of the static fields in an archived class are not suitable for pre-initialization. On the other side, the existing selective static field pre-initialization [3, 4] can still be used to archive individual static fields in JDK classes when needed. Other solutions such as restruct a Java class can be taken so class level pre-initialization and preservation is achievable.

Pre-initializing Application Classes and Preserving Static Field Values

The core algorithms described in above sections for JDK class pre-initialization and preservation can be applied to application classes with additional enhancements to support the system class loader at both dump time and runtime.

This design proposes to initially support pre-initializing application classes (loaded by the system class loader) that only contain:

- `static` fields with primitive types or String type
- `static final` fields
- static fields with @stable (usable for application classes?)

Initially, the static field values are stored separately from the mirrors for application classes. That is needed as archived application classes are not set to linked at restore time. Additional support for application classes will be added incrementally.

Enhance G1 Support for Open Archive Heap Region

As described in [Caching Java Heap Objects](#) [4], all archived Java objects are initially dormant objects when open archive heap regions are mapped to the runtime Java heap. A dormant object becomes a live object (materialized) when the associated archived class is loaded or initialized at runtime. For example, a mirror object is materialized when the VM restores the mirror during the loading process of an archived class at runtime. All objects directly or indirectly referenced from that mirror object also become live (with this proposal). After that point, G1 can find those objects and mark them live during GC operations.

A live object in the open archive heap region should not become dormant again, which was a restriction and a design decision for simplified GC requirements. The purpose of that was to ensure all outgoing pointers from the open archive region(s) were updated by GC correctly. With the only support for mirrors, resolved constant pool arrays, and a small set of selected JDK classes' static field subgraphs in the open archive heap region, the limitation was acceptable.

With the latest design that enhances the subgraph archiving for more general class preinitialization, runtime may encounter cases where a live archived object in the open archive heap region may become dormant again. Those are not common but possible cases, which may be included unintentionally. Restricting the preinitialization support for `final` or `@stable` static field can help avoid the situation. Addressing this issue will enable a broader range of support for pre-initialization for application classes.

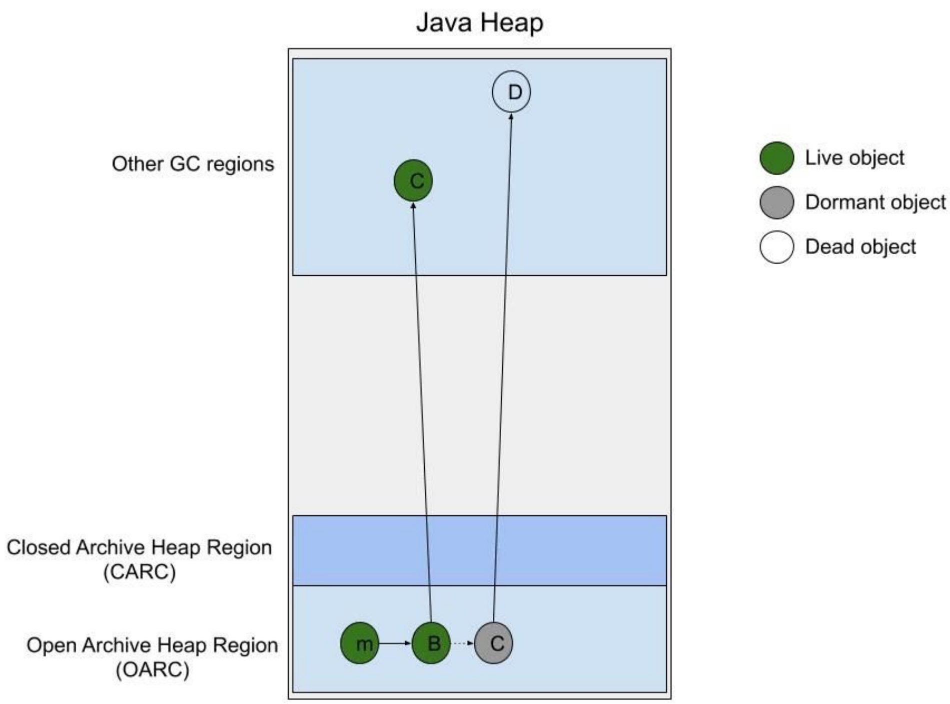
Following is a simple test to demonstrate the particular case.

```
@Preserve
public class A {
  static B b = new B();

  void foo () {
    b.addD();
    b.update();
  }

  public static void main(String args[]) {
    (new A()).foo();
  }
}
```

```
class B {
  C c;

  public B() {
    c = new C();
  }

  public void addD() {
    c.addD();
  }

  public void update() {
    c = new C();
  }
}

class C {
  D d;

  public void addD() {
    d = new D();
  }
}

class D {}
```

If class A is pre-initialized, the preserved subgraph from A's mirror is `mirror->B->C`. All objects in the subgraph become live when materialized at runtime. During Java code execution, `C->D` reference is added. D is not an archived heap object. If the reference of `B->C` is updated to point to a new C instance, the original archived C instance is no longer reachable and becomes dormant again, and D is not reachable either from that point and can be garbage collected. G1 currently does not update the reference from the archived C to D when D is GC'ed, so the dormant C would contain a stale reference. That is not an issue for normal executions (including production jobs, tools execution, etc), but may cause confusions and issues when an agent is walking the heap regions.

One possible solution of handling it: After marking live objects in the open archive heap regions (ORAC), do another iteration of the OARC objects and clear out-going references from objects that are not 'live'. That approach does not require tracking the OARC object state transition (dormant->live->dormant), and is guaranteed to clear all stale references.



Performance Results

TODO: Measure aggregated performance results.

References

[1] The Java® Virtual Machine Specification
[2] JDK-8202035: Archive the set of ModuleDescriptor and ModuleReference objects for observable system modules with unnamed initial module
[3] RFR(L): 8202035: Archive the set of ModuleDescriptor and ModuleReference objects for system modules
[4] Caching Java Heap Objects
[5] JEP 350: Dynamic Archiving