Guide to the MMTk refactoring

Robin Garner
Daniel Frampton

July 29, 2018

1 Introduction

This is a guide to the recent MMTk refactoring, intended to help people who have existing MMTk plans to refactor these plans to fit the new structure.

2 The new structure

The most obvious change is that an MMTk plan has now been split into 3 or more classes. The main aim of this refactoring is to enable the use of instance methods for methods that used to be static, and hence make better use of inheritance to structure a plan.

Previously, system-global data was kept in static fields of a plan, and thread-local data in instance fields of a plan. Now, the separation has become stronger by separating thread-local data into a separate class hierarchy.

2.1 Collection

The mechanism used for performing garbage collection has become much more general. Each (global and local) plan class now provides a method called `collectionPhase`, which takes an integer parameter `phaseId`. The `StopTheWorld` class defines the phases for a stop-the-world collection, and processes them sequentially at each garbage collection, calling the `collectionPhase` method of the plan for each phase. This method follows a common pattern, and for some phases, the plan will perform actions required by this layer, and then delegate to the super-class, while for the remaining phases it will simply delegate to the superclass. This is considerably more flexible than the `threadLocalPrepare` etc methods provided by the old system.

Collection phases are represented by a `Phase` object that specifies a timer name and whether the phase has local and/or global components, and in which order they are executed.

2.2 Tracing

The core of most garbage collections is the operation of tracing the heap. This has been separated into a separate class hierarchy, allowing collectors such as the generational collectors to create separate nursery and full-heap trace classes rather than have complex methods that behave differently in the different cases.

As a consequence, policy and VM interface classes that participate in tracing now also take `TraceLocal` parameters.

2.3 Interfacing with the VM

The most visible change in the interface is that the `Plan` class has been replaced with three classes: `ActivePlan`, `ActivePlanLocal` and `ActivePlanConstraints`. These classes extend the corresponding selected plan class in much the same way as the old `Plan` class.

Selecting the plan at build time is now done by substituting a value rather than by a long series of boolean options. For example, `config/build/gc/SemiSpace` now has,

```
export RVM_WITH_JMTK_PLAN="org.mmtk.plan.semispace.SS"
```

and the various `ActivePlan` classes construct their superclass names according to the naming scheme outlined below.

Code that needs to access the global or local plan instances can call methods of ActivePlan to locate the correct object.

3 Refactoring an existing plan

This section describes in outline how to fit an existing plan into the new structure. It goes though the process of changing the existing MarkSweep collector to work in the new structure.

1. Create a package for the plan. For example, the marksweep collector now lives in the package `org.mmtk.plan.marksweep`.
2. Create 4 classes,
   - The global class, in this case `MS`, which will extend `StopTheWorld` in the case of a full-heap collector, or `Gen` for a classic generational collector.
   - The local class, eg `MSLocal`, which will extend `StopTheWorldLocal` or `GenLocal` respectively. The name of the class must be constructed by appending 'Local' to the global class name for the build process to work.
   - A constraints class, eg `MSConstraints`, extending the appropriate Constraints superclass, (again, the naming scheme is mandatory) and

- One or more trace classes, eg `MSTraceLocal`, extending `TraceLocal` or any other trace classes that may be necessary for the collector. For example, generational collectors define a `MatureTraceLocal` class, and inherit a `NurseryTraceLocal` class from the `generational` package. The naming scheme for trace classes is not mandatory.

Now start moving fields and methods from the original source file to the appropriate new class.

3.1 The Global Class

1. Move constant definitions (eg `ALLOC_MS`) to the global class, as well as space definitions and the corresponding space descriptors. Generally every static field of the old class will become a static field of the new global class.

2. Create an instance field in the global class, like this

   ```
   public final Trace msTrace;
   ```

   This is the global trace object, where the shared queues used during tracing etc are held. The constructor for the global trace class should create the instance.

3. Create a minimal global collectionPhase method, and include the contents of the existing globalPrepare() and globalRelease() methods.

   ```
   public final void collectionPhase(int phaseId)
       throws InlinePragma {
     if (phaseId == PREPARE) {
       super.collectionPhase(phaseId);
       // contents of the globalPrepare() method
       return;
     }
     if (phaseId == RELEASE) {
       // contents of the globalRelease() method
       super.collectionPhase(phaseId);
       return;
     }
     super.collectionPhase(phaseId);
   }
   ```

   Note that PREPARE should probably delegate to super before doing plan-specific things, and RELEASE afterwards. The available collection phases are defined in `StopTheWorld`.

4. Copy the `poll` method across. This will probably require no changes.

5. Add any necessary accounting methods. At a minimum, getPagesUsed should add in the contribution of any spaces defined in this level of the plan hierarchy, eg

```
public int getPagesUsed() {
  return (msSpace.reservedPages() + super.getPagesUsed());
}
```

Note that `getPagesReserved` (in `Plan`) is now expressed in terms of `getPagesUsed` and `getCopyReserve`. These methods are all instance methods.

This is probably all that is required for the global plan class.

3.2 The Local Class

1. Define a convenience method in the Local class called global(), like this

   ```
   private static final MS global() throws InlinePragma {
     return (MS)ActivePlan.global();
   }
   ```

2. Move allocator definitions to the new local class.

3. Define a `TraceLocal` object like this

   ```
   private MSTraceLocal trace;
   ```

   initialised in the constructor like

   ```
     trace = new MSTraceLocal(global().msTrace);
   ```

   and define a method

   ```
   public final TraceLocal getCurrentTrace() {
     return trace;
   }
   ```

   to return the current trace. A plan with more than one trace method will need this method to select the correct one—for example a generational plan should return either the nursery or full-heap trace object as required.

4. Copy the allocation methods `alloc`, `postAlloc` (and for copying collectors `allocCopy` and `postCopy`) to the local class. The body of the methods should be changed so that the spaces defined at this level of the plan hierarchy are handled, and then delegate to super, eg

   ```
   public Address alloc(int bytes, int align, int offset, int allocator)
     throws InlinePragma {
     if (allocator == MS.ALLOC_DEFAULT) {
       return ms.alloc(bytes, align, offset, false);
     }
     return super.alloc(bytes, align, offset, allocator);
   }
   ```

   `allocCopy` and `postCopy` now also take an allocator as a parameter. Non-copying plans no longer need to provide `allocCopy` and `postCopy`.

5. Create a collectionPhase method - this takes 2 more parameters than the global equivalent.

```
public final void collectionPhase(int phaseId, boolean participating,
                                  boolean primary)
  throws InlinePragma {
  if (phaseId == MS.PREPARE) {
    super.collectionPhase(phaseId, participating, primary);
    // Contents of threadLocalPrepare()
    trace.prepare();
    return;
  }
  if (phaseId == MS.START_CLOSURE) {
    trace.startTrace();
    return;
  }

  if (phaseId == MS.COMPLETE_CLOSURE) {
    trace.completeTrace();
    return;
  }

  if (phaseId == MS.RELEASE) {
    // Contents of threadLocalRelease()
    trace.release();
    super.collectionPhase(phaseId, participating, primary);
    return;
  }
super.collectionPhase(phaseId, participating, primary);
}
```

   The `primary` flag is used to distinguish one processor in case certain work needs to be done once only, but in a thread-local context. Use this flag instead of adding additional rendezvous points and checking the arrival index.

6. Override `getSpaceFromAllocator` and `getAllocatorFromSpace`. These should look something like

```
public Space getSpaceFromAllocator(Allocator a) {
  if (a == ms) return MS.msSpace;
  return super.getSpaceFromAllocator(a);
}
```

   ie they handle spaces defined at this level, and then delegate.

3.3 The Trace Class

1. The constructor should look like this

```
public MSTraceLocal(Trace trace) {
  super(trace);
}
```

2. Copy the `isLive` method to the trace class, make it an instance method, and simplify it to reflect the standard pattern of handling local stuff and delegating to super.

3. Copy the `traceObject` method across and make it an instance method. Note that the `traceObject` method in a policy now takes a TraceLocal as its first parameter.

4. A copying collector requires a precopyObject method, and should override the willNotMove method.

3.4 The Constraints Class

The `Constraints` class simply renames the existing `Constants` class, and brings the method naming into line with standards.

This class exists so that the host VM can discover certain static properties of the current MMTk plan without introducing additional class initialization dependencies.

3.5 Generational collectors

A `Global` class for a generational collector must provide a `activeMatureSpace()` method, and a `Local` class must provide a `getFullHeapTrace()` method, but is structurally no different to a full-heap collector.