Runtime Infrastructure - Feature #3003

improve performance of BaseDataType.deepCopy()

02/24/2016 02:06 PM - Eric Faulhaber

Status:	New	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:		**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:	Performance and Scalability Improvements		
billable:	No	**vendor_id:**	GCD

Description

History

#1 - 02/24/2016 02:14 PM - Eric Faulhaber

The BaseDataType.deepCopy method stands out in JVM profiling/sampling.

It is invoked for every in-scope, undo-able variable at the entry of every block for UNDO support. So, if you have accumulated a large number of variables in the stack, particularly in deeply nested blocks, you will have a lot of these calls. Therefore, the implementations of the delegate methods need to be very fast.

We currently use the standard copy constructors of the BaseDataType subclasses in most or all cases. Some of the calls are likely expensive just by nature of the backing data structure (e.g., decimal, I would guess). Also, these code paths honor 4GL assignment semantics. I wonder if we can do away with some of the overhead of these semantics, maybe by creating variants of the duplicate method which are streamlined for the UNDO case (dropping pending error checks, for instance). There may be other work we can avoid in these methods.

#2 - 02/25/2016 12:19 PM - Guy M

Hi,

We discussed today whether we should investigate making search API calls non-TRANSACTION - which might make a deep copy unnecessary. However, I've just checked, and our search APIs are already non-TRANSACTION.

So I reviewed the behaviour of Progress - and it appears that UNDO doesn't do anything outside of a TRANSACTION anyway.

Check this out:

```
/* test1a.p */

RUN test2.p
```

```
/* test2.p */

DEFINE VARIABLE cTest AS CHARACTER INITIAL "Hello World".

REPEAT ON ERROR UNDO, LEAVE:

    cTest = "Goodbye World".

    RUN test3.p.

END.
```

```
MESSAGE PROGRAM-NAME(1) cTest VIEW-AS ALERT-BOX.
```

```
/* test3.p */
MESSAGE PROGRAM-NAME(1) VIEW-AS ALERT-BOX.
RETURN ERROR.
```

Because there is no transaction, the UNDO does not affect the variable in test2.p and "test2.p Goodbye World" is displayed.

However, when you use:

```
/* test1b.p */
FIND FIRST per EXCLUSIVE-LOCK.
RUN test2.p
```

the repeat block within test2.p is within a transaction, the UNDO affects the variable, and "Hello World" is displayed.

Since P2J is mimicking Progress, does this not mean that you only need to do a deep copy at block entry when a TRANSACTION is present?

#3 - 02/25/2016 12:55 PM - Greg Shah

> Since P2J is mimicking Progress, does this not mean that you only need to do a deep copy at block entry when a TRANSACTION is present?

Correct. And in most cases, we already avoid doing this snapshotting when a transaction is not active. There is 1 caveat where we can improve this, but otherwise this is already fully implemented.

I'll leave the "hair-ball" that is temp-tables and database table UNDO to Eric.

The way we handle UNDOable variables:

1. During conversion, if a variable or parameter is calculated as UNDO, then we make sure that a call is made to the TransactionManager.register(), TransactionManager.registerGlobal() or TransactionManager.registerUndo() methods. This call is made within the scope that the variable or parameter is instantiated. The result of this call is that a reference to the variable or parameter is stored an a list that is then processed when that scope (and any contained/nested scope) is entered, iterated, retried or existed.

2. The snapshotting logic is contained in TransactionManager.createBackupSet() and TransactionManager.updateBackupSet(). The create method is used on first entry to a scope that is in an active transaction AND which is NOT marked as a NO_TRANSACTION block. During the create method, a list of undoable references is created using deepCopy(). The update method is used during other block events (e.g. iteration) and it just walks the existing list and uses deepCopy() to refresh the snapshot contents of the already existing instances. Again, this only occurs when the current scope is in an active transaction AND that current scope is NOT marked as a NO_TRANSACTION block. Both of these usages can be found in TransactionManager.backupWorker().

3. When we exit blocks normally we discard any snapshots.

4. When we have to process UNDO, we walk the list of undoables and copy the state back into the source variable or parameter.

5. Eric is right that we may have some ways to optimize deepCopy() or replace it with something more optimal.

6. There is one place where I see a use of this facility that is not protected by a check for being in a transaction AND for the current scope to NOT be marked as a NO_TRANSACTION block. In TransactionManager.deregister() we call TransactionManager.createBackupSet() without that protection. That is used in the case where an undoable variable is passed as a parameter to a function and in the function the parameter is NO-UNDO. In this case we temporarily deregister that instance from our snapshotting lists and recreate the list for the function and any contained scopes. We do have to do something there, but I suspect that we are doing more than is needed. That is probably something we can improve. I don't know if that is the cause of what you are detecting.

7. I doubt there is any optimization in trying to most closely mimic the 4GL implementation of before-image or other disk-based snapshotting that they do. Progress has long advocated setting variables to NO-UNDO because their approach is costly too. There is no getting around some minimum cost.

#4 - 02/26/2016 05:38 AM - Guy M

- File deepCopy_backTrace.png added

Thanks for the detail, Greg.

I've relooked at the call stack - it appears that the deregister() is being called from the init() methods - which are all deregistering NO-UNDO output parameters.

E.g. (from pMapOldToNewRef in acommml0.p)

```
public void init()
         {
            oiEntityTypId.assign(new integer());
```

```
            TransactionManager.deregister(new Undoable[]
            {
               oiEntityTypId,
               oiCommTypId,
               ocKeyValueLst
            });
            oiCommTypId.assign(new integer());
            ocKeyValueLst.assign(new character(""));
         }
```

I'm not sure if this is the scenario that you're talking about in (6), I see that you're passing objects as reference, and so potentially it might have been UNDOable when it came in, and then made NO-UNDO in the sub-procedure.

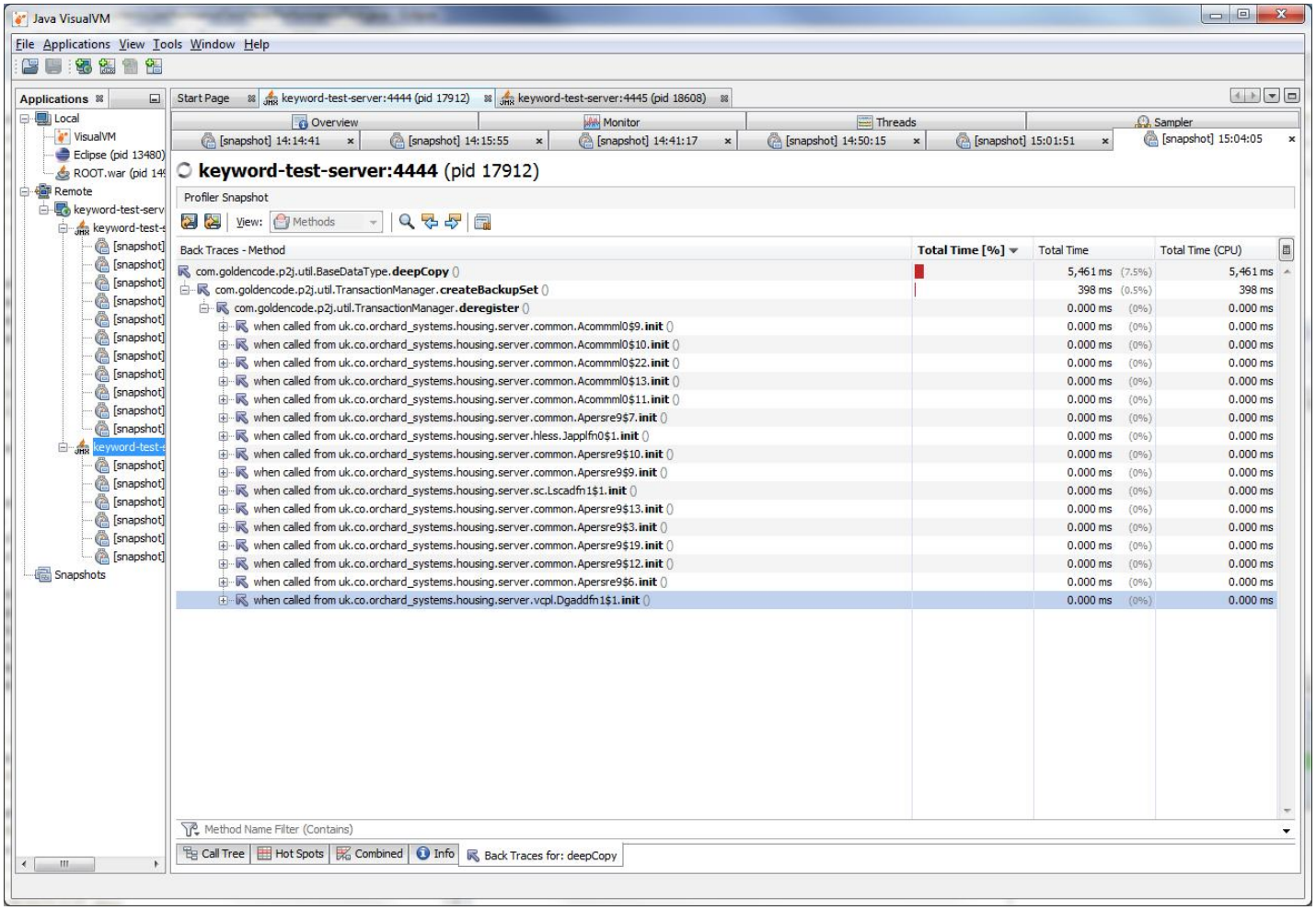
The init() code appears to deregister the OUTPUT parameters in all instances.

A couple of suggestions (please excuse any ignorance):

1) Could you check whether there is a transaction present before doing the deregister? i.e. if there is no TRANSACTION, is there a need to deregister these parameters anyway? Or can you get strange behaviour - e.g. if a transaction is subsequently started, and the code drops back into the original stack (although I haven't done any test cases of that)?

2) It seems you're deregistering the NO-UNDO output parameters regardless of the UNDOability of the object being passed to the procedure. Would it be worth checking whether the object being passed in, is UNDOable anyway, before deregistering?

3) More controversial - If I understand the code correctly, you're passing the objects by reference (I guess this is a feature of Java?), which for "primitive" Progress types isn't what Progress does (I think even for INPUT-OUTPUT parameters). The input parameters you copy immediately, and then you mutate the copies. Could you not do something similar for the OUTPUT parameters? i.e. create new instances of the output parameters in the body of the internal procedure, and then on exit (return), copy their values to the objects passed in? It would require more objects, but wouldn't require the deepCopy(), would it?



#5 - 03/29/2016 03:37 PM - Greg Shah

> I'm not sure if this is the scenario that you're talking about in (6), I see that you're passing objects as reference, and so potentially it might have been UNDOable when it came in, and then made NO-UNDO in the sub-procedure.

Yes, I was referring to this in item 6.

> 1) Could you check whether there is a transaction present before doing the deregister? i.e. if there is no TRANSACTION, is there a need to deregister these parameters anyway?

No, this one can't be protected like that.

> Or can you get strange behaviour - e.g. if a transaction is subsequently started, and the code drops back into the original stack (although I haven't done any test cases of that)?

Yes, this is exactly the issue.

Each variable is added to the undoables list (in the TransactionManager) in the scope in which it is created. That scope may or may not be in a transaction at that time.

When a reference to such a var is passed as an output parameter, in the 4GL the local var in the called location is implicitly no-undo. Again, we don't know if we are in a transaction here or not. And a transaction can be opened at any time in code that is called within this scope. If we do call code that opens a transaction (or if we are already in a transaction at this point), then leaving the output parameter reference in the undoables list will lead to snapshots being taken of its state when those snapshots should never exist.

I suspect we can find a more efficient way to do this, but we must do something.

Also, I should note that we recently made changes in this area (#2647) and the situation may already be somewhat different. We will certainly look at this to see what we can safely do.

> 2) It seems you're deregistering the NO-UNDO output parameters regardless of the UNDOability of the object being passed to the procedure. Would it be worth checking whether the object being passed in, is UNDOable anyway, before deregistering?

While it is inside the called location, it is NO-UNDO implicitly. We restore it back on exit, so that its previous behavior is back.

> 3) More controversial - If I understand the code correctly, you're passing the objects by reference (I guess this is a feature of Java?), which for "primitive" Progress types isn't what Progress does (I think even for INPUT-OUTPUT parameters). The input parameters you copy immediately, and then you mutate the copies. Could you not do something similar for the OUTPUT parameters? i.e. create new instances of the output parameters in the body of the internal procedure, and then on exit (return), copy their values to the objects passed in? It would require more objects, but wouldn't require the deepCopy(), would it?

Actually, in #2647 we worked in this area already. That update went in as P2J revision 10967. I don't know recall how much of this discussion would have been addressed by that, but there may still be some things to consider.

I've added Constantin and Ovidiu as watchers for this task, in case they have any insights to share.

#6 - 11/16/2016 12:30 PM - Greg Shah

- Target version changed from Milestone 17 to Performance and Scalability Improvements

Files

deepCopy_backTrace.png	241 KB	02/26/2016	Guy M