Runtime Infrastructure - Feature #2973

improve ContextLocal performance

01/29/2016 11:36 AM - Eric Faulhaber

Status:	Closed	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Igor Skornyakov	**% Done:**	100%
Category:		**Estimated time:**	0.00 hour
Target version:	Performance and Scalability Improvements		
billable:	No	**vendor_id:**	GCD

Description

Related issues:

Related to User Interface - Feature #3246: reduce the amount of data being se...	**Closed**

History

#1 - 01/29/2016 12:00 PM - Eric Faulhaber

ContextLocal is used very heavily throughout the runtime. Its get method is hit constantly across various subsystems; every time we push and pop scopes, check transaction status, etc, various subsystems (TransactionManager, BlockManager, BufferManager, RecordBuffer, AccumulateManager, etc.) check context local data. The current server-side implementation behind ContextLocal.get relies on multiple hash map lookups of SecurityManager resources, which is relatively expensive.

Greg had the idea of replacing this implementation in the common case by switching to a thread-local storage implementation when code is running on a conversation thread, which should be much faster. The details of how to manage the switches in and out of this mode have yet to be determined, but the idea is that most of a session's lifespan will be spent in conversation mode, so most use cases will benefit to some degree from a faster implementation.

We also need to determine whether there are heavily used modes of execution (e.g., AppServer) which do not always use conversation mode, which would not benefit from this optimization without additional change.

It is not clear how noticeable this change would be. Profiling (sampling) has shown that the context local lookup stands out in some cases, but is less significant in others, especially those with heavy/frequent database I/O.

#2 - 03/24/2016 01:00 AM - Eric Faulhaber

- Status changed from New to WIP

- Assignee set to Igor Skornyakov

Igor, please familiarize yourself with the the current ContextLocal implementation and think about how to best implement the ideas described in note 1. Document your thoughts and questions here. We obviously have to preserve the thread-safety of access to the data, but the performance must be improved over the current implementation.

#3 - 03/24/2016 09:02 AM - Greg Shah

To be clear, we are specifically talking about optimizing the MODE_SERVER mode of ContextLocal, which uses the SecurityManager (and related

classes) for storage. The MODE_STANDALONE is used by our classes when not running in a P2J client or server, and that already uses ThreadLocal (see the Fallback inner class). The MODE_CLIENT uses a static map called contextStorage and already avoids the multi-level security package lookups. So, only the MODE_SERVER cases should be changed.

It is important that you carefully review the following (and the locations from which they are called):

SecurityManager.setInitialSecurityContextWorker()
SecurityManager.dropInitialSecurityContext()
SecurityManager.unassignContext()
SecurityManager.endContext()
SecurityManager.terminateSession()
SecurityContextStack
SecurityContext

In MODE_SERVER, we store context local data in a map (the SecurityContext.tokenMap) that can be accessed from multiple threads. This is ABSOLUTELY CRITICAL, because we do have use cases (even in conversation mode) where more than one thread shares/accesses the same context. This cannot be broken. Understanding the way we set and change/drop security contexts for a given thread is essential.

The problem we are trying to solve is that a call to ContextLocal.get() does so much processing to find the associated value. Since it is called everywhere, any optimization there will make a big difference in performance for the server side.

It is also essential to ensure that code executing on any thread that is not assigned a specific context cannot access any data in other contexts. In other words, the security context must isolate the data so that only threads assigned to that context can access that context's data.

The call path:

1. ContextLocal.get() calls SecurityManager.getToken() which calls SecurityContextStack.getContext() which accesses a ThreadLocal (contextPointer) to find the current thread's SecurityContextStack instance. Looking inside the ThreadLocal, it has its own ThreadLocalMap implementation. I guess this is the first level of map lookups that Eric is referencing.

2. SecurityContextStack.getToken() is called on that instance, which calls SecurityContextStack.getEffectiveContext() which returns the current SecurityContext instance. This chooses between two different possible SecurityContext instances, depending on whether a secondary context was pushed onto this "stack". There really isn't much of a stack since it can only be two levels. This operation is fast, but it means that there is a kind of dynamic assignment of the context. You will have to look carefully at where we push (SecurityManager.pushContextWorker() which is called from 3 different locations, SecurityManager.terminateSession() and SecurityManager.endContext()) and where we pop (SecurityManager.popContextWorker() which is called from 4 different locations, SecurityManager.terminateSession() and SecurityManager.endContext()). If I recall how this works, I think that in practical terms, once a thread has been assigned a context, it won't be changed dynamically until the context is removed from the thread completely. But this really needs to be checked carefully.

3. SecurityContext.getToken() is called on that instance which does the tokenMap.get(key) to lookup the value. This is the second map lookup and is implemented using a HashMap.

I'm not sure my idea of using ThreadLocal helps, since we are already using ThreadLocal. The key question: can we come up with a more efficient way to do this that is secure, reliable and functionally equivalent to the current implementation?

#4 - 03/24/2016 09:05 AM - Igor Skornyakov

I see. Thank you for the detailed explanations.

#5 - 03/24/2016 12:12 PM - Eric Faulhaber

Greg and I have been discussing several approaches to this. The simpler of these, which may buy us back sufficient performance (we'll have to see through testing/profiling), is simply to improve the synchronization of the SecurityContext class.

Many of the methods in this class are synchronized, which tends to be expensive, even in an uncontested scenario. The two resources being protected by this synchronization are the use instance variable and the tokenMap instance variable. Our access is read-heavy, so we should be able to improve upon the current situation with more granular locking, through use of the J2SE concurrency classes.

#6 - 03/24/2016 12:37 PM - Eric Faulhaber

The more complicated approach would be something to consider if the synchronization improvements described in the previous note do not provide enough of a performance improvement.

The idea is to use a ConcurrentHashMap as a static variable of ContextLocal, which would act as a cache of ContextLocal$Wrapper objects. The key used for this map would be composite of a unique identifier for the ContextLocal instance and a unique identifier for the current security context. Both components would have to be very fast to access (i.e., simple integral values), nothing that would require a native call.

If a cache hit fails, we would retrieve the Wrapper instance the way it's currently done, from SecurityManager (including the above synchronization improvements) and add it to the cache.

For this to work, there has to be a secure and fast mechanism to get the current security context's unique identifier from the current thread, so that it can be used in a composite hash key which can be built quickly and whose overridden hashCode and equals methods execute quickly. I'm not sure what this thread-to-context-id mechanism is (this may be where the approach falls short) -- open to ideas.

The other aspect that makes this approach more complicated is context cleanup. At the end of a context, the cache entries associated with that context need to be removed reliably from the cache.

#7 - 03/24/2016 12:58 PM - Constantin Asofiei

Eric Faulhaber wrote:

> For this to work, there has to be a secure and fast mechanism to get the current security context's unique identifier from the current thread, so that it can be used in a composite hash key which can be built quickly and whose overridden hashCode and equals methods execute quickly. I'm not sure what this thread-to-context-id mechanism is (this may be where the approach falls short) -- open to ideas.

For the ContextLocal instance, is safe to use its hashCode() implementation; the same can be used for the SecurityContext. To combine these 32-bit integer values into an unique one, there is some interesting reading here:
http://stackoverflow.com/questions/919612/mapping-two-integers-to-one-in-a-unique-and-deterministic-way and here
https://en.wikipedia.org/wiki/Pairing_function#Cantor_pairing_function

#8 - 03/24/2016 01:24 PM - Eric Faulhaber

Constantin Asofiei wrote:

> Eric Faulhaber wrote:
>
> > For this to work, there has to be a secure and fast mechanism to get the current security context's unique identifier from the current thread, so that it can be used in a composite hash key which can be built quickly and whose overridden hashCode and equals methods execute quickly. I'm not sure what this thread-to-context-id mechanism is (this may be where the approach falls short) -- open to ideas.
>
> For the ContextLocal instance, is safe to use its hashCode() implementation; the same can be used for the SecurityContext.

In practice, most likely, though there is no guarantee of uniqueness with Object.hashCode. From the javadoc:

> As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

Consider a 64-bit pointer represented in 32 bits (highly unlikely for 2 objects to collide, but theoretically possible, and a nightmare to debug if it ever happened), or just some JVM implementation that does it differently.

The issue perplexing me more is how to find the current SecurityContext to get its ID, given the current thread, without doing the same expensive lookup we are trying to get rid of.

#9 - 03/24/2016 01:34 PM - Igor Skornyakov

Constantin Asofiei wrote:

> For the ContextLocal instance, is safe to use its hashCode() implementation; the same can be used for the SecurityContext. To combine these 32-bit integer values into an unique one, there is some interesting reading here: http://stackoverflow.com/questions/919612/mapping-two-integers-to-one-in-a-unique-and-deterministic-way and here https://en.wikipedia.org/wiki/Pairing_function#Cantor_pairing_function

Please note that although NxN has the same cardinality as N and it is possible to find a bijection between them it is not the fact for Java integers as

they are members of the **finite** set.

#10 - 03/25/2016 09:40 AM - Greg Shah

Thoughts:

1. ContextLocal instances should always be stored in a static member. This allows for shared access to the stored data value. That means that we have a limited number of these in the project. This will be in the tens or maybe hundreds, but certainly not in the thousands or millions. At construction, each ContextLocal can be given a simple integer ID and we can limit that to a range (5 or 6 bits will be enough for the forseeable future, but we would surely be able to live within a limit of a 16-bit value).

2. SecurityContext instances are not as long-lived, but the simultaneous number of these at any point in time is likewise easily fit within some bounded number. 16-bit is probably reasonable, but 32-bit or 48-bit is extremely safe over the long term. At construction, this ID would be assigned. We would have to maintain that ID space and reclaim IDs when the associated SecurityContext instance is gone. This is needed because on a long-running system we might overflow a 16-bit or 32-bit value but we never need that many simultaneous IDs.

Considering the above, we might easily design a solution that uses a 64-bit value as a bit-field, storing a SecurityContext ID in the most significant 32-bits (or even 48-bits) and the context local ID in the least significant 16-bits. That will be unique and fast and can be used as a key.

When a security context is cleaned up, we must ensure that all associated values for the SecurityContext ID are removed from the static storage. This is easily done with this bitfield approach, since one can iterate over the keys and mask off the ContextLocal portion of the key to see if the key matches the SecurityContext.

The only tricky part is how to map each Thread instance to its backing SecurityContext ID. This must be very fast (and cannot be using ThreadLocal since we are trying to avoid the cost of the thread local internal map lookups.

The number of Thread instances concurrently active on a system are limited by practicality. Generally, there are only hundreds of threads at any given time, but it is possible to have thousands. Tens of thousands starts to become a problem for real systems, but still theoretically possible. Each thread already has platform-specific TID as a unique 64-bit value (see Thread.getId()) and this is already stored in the Java heap as an internal member. So, calling Thread.currentThread().getId() is a pretty fast way to get that 64-bit value. It is probably faster than a hashcode lookup for the Thread instance. Either one of those is unique. The key question for me is whether we can quickly use some value like this to find the SecurityContext ID. If so, then this solution may have merit.

#11 - 04/07/2016 05:40 AM - Igor Skornyakov

Created task branch 2973a from the trunk revision 10995.

#12 - 04/07/2016 07:55 AM - Igor Skornyakov

I've implemented optimization mentioned in the node 5 along with some refactoring.

Committed to the task branch 2973 revision 10996.

Please take a look.
Is any additional optimization is supposed in the scope of this task?

Thank you.

#13 - 04/07/2016 08:52 AM - Igor Skornyakov

Task branch 2973a was rebased from the trunk revision 10996. Committed as revision 10997.

#14 - 04/07/2016 09:19 AM - Eric Faulhaber

Code review 2973a/10997:

Yes, this is what I was looking for in the scope of note 5, thanks.

One question: why did you make tokenMap a ConcurrentHashMap, when you also protected all access to it with the read/write tokenMapLock? Is there any advantage to this change? Is there a performance disadvantage?

#15 - 04/07/2016 09:49 AM - Greg Shah

I would like to see some testing that detects if this has a measurable performance impact (and what that might be).

Also, please note that the CTRL-C tests will be very important for this change.

Overall, I don't have a concern with the code changes.

Constantin: any thoughts?

#16 - 04/07/2016 09:51 AM - Igor Skornyakov

Eric Faulhaber wrote:

> Code review 2973a/10997:
>
> Yes, this is what I was looking for in the scope of note 5, thanks.
>
> One question: why did you make tokenMap a ConcurrentHashMap, when you also protected all access to it with the read/write tokenMapLock? Is there any advantage to this change? Is there a performance disadvantage?

The lock provides an exclusive access to the tokenMap for the group operations only. Please note that single updates are all protected by readLock so they rely on the concurrent features of the ConcurrentHashMap.

#17 - 04/07/2016 10:46 AM - Eric Faulhaber

There is a regression when launching the P2J server:

```
com.goldencode.p2j.cfg.ConfigurationException:  Initialization failure
        at com.goldencode.p2j.main.StandardServer.hookInitialize(StandardServer.java:1675)
        at com.goldencode.p2j.main.StandardServer.bootstrap(StandardServer.java:840)
        at com.goldencode.p2j.main.ServerDriver.start(ServerDriver.java:415)
```

```
        at com.goldencode.p2j.main.CommonDriver.process(CommonDriver.java:396)
        at com.goldencode.p2j.main.ServerDriver.process(ServerDriver.java:147)
        at com.goldencode.p2j.main.ServerDriver.main(ServerDriver.java:756)
Caused by: java.lang.ExceptionInInitializerError
        at com.goldencode.p2j.main.StandardServer$12.initialize(StandardServer.java:1103)
        at com.goldencode.p2j.main.StandardServer.hookInitialize(StandardServer.java:1671)
        ... 5 more
Caused by: java.lang.NullPointerException
        at java.util.concurrent.ConcurrentHashMap.putVal(ConcurrentHashMap.java:1011)
        at java.util.concurrent.ConcurrentHashMap.putIfAbsent(ConcurrentHashMap.java:1535)
        at com.goldencode.p2j.security.SecurityContext.addToken(SecurityContext.java:388)
        at com.goldencode.p2j.security.SecurityContextStack.setEditing(SecurityContextStack.java:215)
        at com.goldencode.p2j.security.SecurityManager.openBatch(SecurityManager.java:5045)
        at com.goldencode.p2j.directory.DirectoryService.openBatch(DirectoryService.java:3255)
        at com.goldencode.p2j.security.SecurityAdmin.removeSubjectFromACLs(SecurityAdmin.java:6190)
        at com.goldencode.p2j.security.SecurityAdmin.deleteUser(SecurityAdmin.java:1595)
        at com.goldencode.p2j.admin.AdminServerImpl.deleteUser(AdminServerImpl.java:1377)
        at com.goldencode.p2j.main.TemporaryAccountPool.deleteTemporaryAccounts(TemporaryAccountPool.java:155)
        at com.goldencode.p2j.main.TemporaryAccountPool.<clinit>(TemporaryAccountPool.java:99)
        ... 7 more
```

#18 - 04/07/2016 10:48 AM - Igor Skornyakov

Eric Faulhaber wrote:

> There is a regression when launching the P2J server:
> [...]

The ConcurrentHashMap doesn't allow null values. Do we need such values?
Thank you.

#19 - 04/07/2016 11:20 AM - Igor Skornyakov

If we really need null values we can wrap it into Optional. However it seems that null is used in a single place: at the SecurityContextStack.setEditing method. May be it is better to fix this place?

#20 - 04/07/2016 11:32 AM - Eric Faulhaber

Igor Skornyakov wrote:

> If we really need null values we can wrap it into Optional. However it seems that null is used in a single place: at the SecurityContextStack.setEditing method. May be it is better to fix this place?

The latter makes sense to me. Greg, do you have an opinion on this?

#21 - 04/07/2016 11:52 AM - Greg Shah

> However it seems that null is used in a single place: at the SecurityContextStack.setEditing method. May be it is better to fix this place?

Yes, it is completely safe to do that.

One note: using the "system-security-editing" string as the key is not secure. This is a latent security hole. Any code can call SecurityManager.getToken("system-security-editing") or SecurityManager.hasToken("system-security-editing") and this will leak information. Future changes could cause the value to be set.

A better approach would be to create a private static final Object EDITING_KEY = new Object(); in SecurityContextStack. Then use that in place of the "system-security-editing" key.

#22 - 04/07/2016 12:00 PM - Igor Skornyakov

Greg Shah wrote:

> > However it seems that null is used in a single place: at the SecurityContextStack.setEditing method. May be it is better to fix this place?
>
> Yes, it is completely safe to do that.
>
> One note: using the "system-security-editing" string as the key is not secure. This is a latent security hole. Any code can call SecurityManager.getToken("system-security-editing") or SecurityManager.hasToken("system-security-editing") and this will leak information. Future changes could cause the value to be set.
>
> A better approach would be to create a private static final Object EDITING_KEY = new Object(); in SecurityContextStack. Then use that in place of the "system-security-editing" key.

Done.
Committed to the task branch 2973 revision 10998.

#23 - 04/07/2016 01:29 PM - Eric Faulhaber

I ran the full search test group. When I disconnected BPM at the end, while the P2J server was still running, I got this:

```
[04/07/2016 17:26:10 GMT] (com.goldencode.p2j.util.Agent:WARNING) Agent encountered an error while executing a
 command for appserver app_server
java.lang.NullPointerException
        at com.goldencode.p2j.util.FileSystemOps$ContextContainer.initialValue(FileSystemOps.java:1119)
        at com.goldencode.p2j.security.ContextLocal.get(ContextLocal.java:417)
        at com.goldencode.p2j.security.ContextLocal.get(ContextLocal.java:374)
        at com.goldencode.p2j.util.FileSystemOps$ContextContainer.obtain(FileSystemOps.java:1097)
        at com.goldencode.p2j.util.FileSystemOps.setSearchPath(FileSystemOps.java:1059)
        at com.goldencode.p2j.util.EnvironmentOps.setSearchPath(EnvironmentOps.java:496)
        at com.goldencode.p2j.util.EnvironmentOps.setSearchPath(EnvironmentOps.java:526)
        at com.goldencode.p2j.util.Agent.prepare(Agent.java:1516)
        at com.goldencode.p2j.util.Agent.access$1000(Agent.java:59)
        at com.goldencode.p2j.util.Agent$ResetContextCommand.execute(Agent.java:1557)
        at com.goldencode.p2j.util.Agent.resetContext(Agent.java:432)
        at com.goldencode.p2j.util.AgentPool$BoundPool.terminateConnection(AgentPool.java:887)
        at com.goldencode.p2j.util.Agent$3.execute(Agent.java:597)
        at com.goldencode.p2j.util.Agent.listen(Agent.java:379)
        at com.goldencode.p2j.util.AgentPool.start(AgentPool.java:432)
        at com.goldencode.p2j.util.AppServerManager.startAppServer(AppServerManager.java:877)
        at com.goldencode.p2j.main.StandardServer.standardEntry(StandardServer.java:274)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:497)
        at com.goldencode.p2j.util.MethodInvoker.invoke(MethodInvoker.java:76)
        at com.goldencode.p2j.net.Dispatcher.processInbound(Dispatcher.java:705)
        at com.goldencode.p2j.net.Conversation.block(Conversation.java:319)
        at com.goldencode.p2j.net.Conversation.run(Conversation.java:163)
        at java.lang.Thread.run(Thread.java:745)
```

I have never seen this with the old implementation, so this appears to be a regression.

#24 - 04/07/2016 01:31 PM - Igor Skornyakov

Eric Faulhaber wrote:

I ran the full search test group. When I disconnected BPM at the end, while the P2J server was still running, I got this:
[...]
I have never seen this with the old implementation, so this appears to be a regression.

I see. Investigating.

#25 - 04/07/2016 01:37 PM - Constantin Asofiei

Review for 2973a rev 10999:

1. the tokenMapLock.readLock() and tokenMapLock.writeLock() can be saved in SecurityContext instance fields, to avoid resolving them each time they are needed.
2. transferAllTokens is still synchronized
3. transferAllTokens - you are locking for writing only on this.tokenMapLock. For the other.tokenMapLock, no locking is done, although you are adding entries to the other.tokenMap. Shouldn't there be write-locking with the other.tokenMapLock, too?
4. cleanupWorker(boolean reset): why does this code need a write lock? tokenMap is only read, not written to, in this block.

```
      tokenMapLock.writeLock().lock();
      try
      {
         clone.putAll(tokenMap);
      }
      finally
      {
         tokenMapLock.writeLock().unlock();
      }
```

5. cleanupWorker(boolean reset, Map tm) - shouldn't this filter condition:

```
reset || !(key instanceof ContextLocal) || ((ContextLocal) key).isResetAllowed()
```

be

```
!reset || !(key instanceof ContextLocal) || ((ContextLocal) key).isResetAllowed()
```

to properly match the negation of the previous condition:

```
         if (payload == null ||
             (reset && key instanceof ContextLocal && !((ContextLocal) key).isResetAllowed()))
         {
            continue;
         }
```

Also, about using both a read-write lock and a ConcurrentMap. The concept of a read lock is to allow multiple threads to read the data in parallel but block only and only when an exclusive/write lock is acquired by another thread, which modifies the data. So, I don't understand why you are using both read/write locks and ConcurrentMap - writing to or reading from the tokenMap can be protected only by the tokenMapLock (as either will not be possible if a write lock is in effect).

I would expect for this to work: use a normal HashMap implementation for the tokenMap, protect all changes on this map with a write-lock and all reading from this map with a read-lock. This removes the synchronization overhead of the ConcurrentHashMap implementation.

#26 - 04/07/2016 01:42 PM - Igor Skornyakov

Eric,
I understand that NPE happens because FileSystemOps.WorkArea.fs is null. This can be of course a result of my changes but indirectly. Continue investigation.

#27 - 04/07/2016 01:52 PM - Eric Faulhaber

In terms of performance, it was on the fastest end of the range of my recent timings for this test group, but not drastically better. If I factor in the fact that one test hit the #2738 bug and took 2 minutes to time out, it was actually about a minute faster (~1.5%) than my best recent time. Not dramatic, but a step in the right direction. However, the timings of these runs can swing by 15 minutes, so it's hard to draw a conclusion from just one data point.

Igor, I would like to try a slight permutation on your implementation. Please use HashMap instead of ConcurrentHashMap for tokenMap and extend the purpose of tokenMapLock, such that it gets a write lock for all operations that update tokenMap (not just for the group operations), and a read lock for the operations that only read from it, such as getToken.

My reasoning is that previous profiling identified getToken as the primary bottleneck. None of the other operations were noticeable. So, I want to bias the access to be as fast as possible for this method. The premise is that the HashMap.get implementation is slightly faster than ConcurrentHashMap.get, which I'm not sure about, but I think it's worth trying. Thanks.

#28 - 04/07/2016 01:54 PM - Eric Faulhaber

Igor Skornyakov wrote:

> I understand that NPE happens because FileSystemOps.WorkArea.fs is null. This can be of course a result of my changes but indirectly. Continue investigation.

Perhaps it's a latent defect that your changes revealed through a change in timing or sequence of operations.

#29 - 04/07/2016 02:02 PM - Constantin Asofiei

Eric Faulhaber wrote:

> I ran the full search test group. When I disconnected BPM at the end, while the P2J server was still running, I got this:
> [...]
> I have never seen this with the old implementation, so this appears to be a regression.

The only way I see for wa.fs to be null in this line wa.fs.setSearchPath(paths, caseSens); is for RemoteObject.obtainInstance obtain instance to return null, on this line:

```
         wa.fs = (FileSystem) RemoteObject.obtainInstance(FileSystem.class,
                                                          true);
```

And RemoteObject.obtainInstance() will return:

1. if SessionManager.get().getSession() is not null, this will never return null, as RemoteObject.obtainNetworkInstance can never return null.
2. otherwise, RemoteObject.obtainLocalInstance can return null on line 1137:

```
        if (!Dispatcher.lockMethods(list, methods, null, false, false))
        {
           return null;
        }
```

or line 1159:

```
            if (impl == null)
            {
               // no registered server matches the requested interface
               return null;
            }
```

So, the question is: is it possible that the Agent's P2J client was terminated while FileSystemOps$ContextContainer.initialValue was being executed (so that @RemoteObject.obtainLocalInstance is used)? Otherwise, if obtainNetworkInstace is used, I don't see how RemoteObject.obtainInstance() can ever return null.

#30 - 04/07/2016 02:02 PM - Igor Skornyakov

Constantin Asofiei wrote:

> Review for 2973a rev 10999:
>
> 1. the tokenMapLock.readLock() and tokenMapLock.writeLock() can be saved in SecurityContext instance fields, to avoid resolving them each time they are needed.

I do not think that such an optimization is important enough to break the idiom.

> 1. transferAllTokens is still synchronized

Fixed.

> 1. transferAllTokens - you are locking for writing only on this.tokenMapLock. For the other.tokenMapLock, no locking is done, although you are adding entries to the other.tokenMap. Shouldn't there be write-locking with the other.tokenMapLock, too?

There where no locks on the other initially. Actually this method is used only once for a fresh-made other which is not published yet. So at least now the synchronization is not needed.

1. cleanupWorker(boolean reset): why does this code need a write lock? tokenMap is only read, not written to, in this block.
   [...]

Actually I would prefer to keep writeLock() for the whole cleanupWorker(boolean reset) method. I've just retained initial logic with two synchronized sections.

1. cleanupWorker(boolean reset, Map tm) - shouldn't this filter condition:
   [...]
   be
   [...]
   to properly match the negation of the previous condition:
   [...]

Fixed.

> Also, about using both a read-write lock and a ConcurrentMap. The concept of a read lock is to allow multiple threads to read the data in parallel but block only and only when an exclusive/write lock is acquired by another thread, which modifies the data. So, I don't understand why you are using both read/write locks and ConcurrentMap - writing to or reading from the tokenMap can be protected only by the tokenMapLock (as either will not be possible if a write lock is in effect).
>
> I would expect for this to work: use a normal HashMap implementation for the tokenMap, protect all changes on this map with a write-lock and all reading from this map with a read-lock. This removes the synchronization overhead of the ConcurrentHashMap implementation.

What you describe is a standard semantics. Another (more generic one) is to use readLock() for operations that can be safely done concurrently (in our case - simple operations with ConcurrentMap) and writeLock() for operations which cannot be done in concurrently even with the operations of the first kind. In our case these are group operations. See comment for the tokenMapLock field.

Committed to the task branch 2973 revision 11000.

#31 - 04/07/2016 02:07 PM - Constantin Asofiei

Followup for note 29: OTOH, this can be a fallback for the incorrect cleanupWorker(boolean reset, Map tm) filter condition, which ended up resetting the SessionManager.activeSession context-local field.

Also, another bug in cleanupWorker(boolean reset, Map tm) - the doReset filter you are using needs to be used for this block, too:

```
      for (Object key: allKeys)
      {
         Object payload = tm.get(key);

         if (payload != null)
         {
            reset(payload);

            // remove it from the clone map
            tm.remove(key);
         }
      }
```

Here, keys for which isResetAllowed returns false must NOT be reset. So, please double-check the cleanupWorker logic matches the previous one.

#32 - 04/07/2016 02:18 PM - Igor Skornyakov

Eric Faulhaber wrote:

> In terms of performance, it was on the fastest end of the range of my recent timings for this test group, but not drastically better. If I factor in the fact that one test hit the #2738 bug and took 2 minutes to time out, it was actually about a minute faster (~1.5%) than my best recent time. Not dramatic, but a step in the right direction. However, the timings of these runs can swing by 15 minutes, so it's hard to draw a conclusion from just one data point.
>
> Igor, I would like to try a slight permutation on your implementation. Please use HashMap instead of ConcurrentHashMap for tokenMap and extend the purpose of tokenMapLock, such that it gets a write lock for all operations that update tokenMap (not just for the group operations), and a read lock for the operations that only read from it, such as getToken.
>
> My reasoning is that previous profiling identified getToken as the primary bottleneck. None of the other operations were noticeable. So, I want to bias the access to be as fast as possible for this method. The premise is that the HashMap.get implementation is slightly faster than ConcurrentHashMap.get, which I'm not sure about, but I think it's worth trying. Thanks.

Eric,
Based on my experience this will not improve performance (the difference in the HashMap.get() and ConcurrentHahMap().get() performance is negligible especially comparing to the contended lock acquisition overhead which will be greater in the version you suggest).
However if your tests show that Map.get() is a bottleneck the reason can be that this map is effectively an IdentityMap. May be it is better to consider using keys with good hash codes to avoid degeneration of the map to a linked list? Moreover in some situations (when collision are unavoidable and the Map is updated frequently) I found more practical to use a TreeMap to avoid spikes (of course if the average O(log(n)) access time is acceptable at all).

In any case if you believe that it is worth to check I will implement your suggestion.

#33 - 04/07/2016 02:23 PM - Igor Skornyakov

Constantin Asofiei wrote:

> Followup for note 29: OTOH, this can be a fallback for the incorrect cleanupWorker(boolean reset, Map tm) filter condition, which ended up resetting the SessionManager.activeSession context-local field.
>
> Also, another bug in cleanupWorker(boolean reset, Map tm) - the doReset filter you are using needs to be used for this block, too:
> [...]
> Here, keys for which isResetAllowed returns false must NOT be reset. So, please double-check the cleanupWorker logic matches the previous one.

OK, I will double check. Please not that it doesn't make sense to check the predicate twice: in the initial logic the keys which do not satisfy the predicate where just ignored. I've move the check up in a hope to reduce the size of the weightedKeys and which is sorted and appended.

#34 - 04/07/2016 02:26 PM - Constantin Asofiei

Igor Skornyakov wrote:

> OK, I will double check. Please not that it doesn't make sense to check the predicate twice: in the initial logic the keys which do not satisfy the predicate where just ignored. I've move the check up in a hope to reduce the size of the weightedKeys and which is sorted and appended.

The logic in cleanupWorker does this:

- step 1: it sorts or keys by their weight, with non-weighted keys put first (you are doing this ok)
- step 2: it goes through ALL keys, and checks if they can be reset or not, and does this reset on the order determined on step 1. Your code no longer checks if the key can be reset - it resets always!

#35 - 04/07/2016 02:31 PM - Igor Skornyakov

Constantin Asofiei wrote:

> The logic in cleanupWorker does this:
>
> - step 1: it sorts or keys by their weight, with non-weighted keys put first (you are doing this ok)
> - step 2: it goes through ALL keys, and checks if they can be reset or not, and does this reset on the order determined on step 1. Your code no longer checks if the key can be reset - it resets always!

This is not the fact. The keys which cannot be reset just do not come to allKeys.

#36 - 04/07/2016 02:40 PM - Constantin Asofiei

Igor Skornyakov wrote:

> Constantin Asofiei wrote:
>
> > The logic in cleanupWorker does this:
> >
> > - step 1: it sorts or keys by their weight, with non-weighted keys put first (you are doing this ok)
> > - step 2: it goes through ALL keys, and checks if they can be reset or not, and does this reset on the order determined on step 1. Your code no longer checks if the key can be reset - it resets always!
>
> This is not the fact. The keys which cannot be reset just do not come to allKeys.

Ah, this was valid before the reset filter bug, which should have been !reset bug. I think now the issue Eric mentioned in note 23 should be fixed.

#37 - 04/07/2016 02:48 PM - Constantin Asofiei

Greg, I think we should consider how many elements will tokenMap hold, on maximum. ContextLocal is instantiated in 128 places, at this time. For each case, only an entry will be added to this tokenMap, for each security context. So, excluding the cases when tokenMap holds as key something else (as the "system-security-editing" case mentioned in note 21 and the sm.addToken("headless", Boolean.TRUE); case in SecurityAdmin c'tor), tokenMap keys are always ContextLocal instances, which are defined as static variables, and can be at most 128 instances of them, for each security context.

As an alternative, we can:

1. assume all keys in tokenMap are ContextLocal instances (and change them to a ContextLocal where they are not)
2. associate to each ContextLocal instance a numeric ID from 0 to 129 (128 already ContextLocal, 2 pending, 130 total). We should be able to automate this via AspectJ.
3. instead of using a tokenMap, use a Object[] array, where each index will hold the value for that ContextLocal instance.

This way, we can remove the map overhead completely. If the performance is still not improved, then the bottleneck remains the SecurityContextStack.contextPointer, which is a ThreadLocal instance.

#38 - 04/07/2016 02:54 PM - Constantin Asofiei

Constantin Asofiei wrote:

> ... tokenMap keys are always ContextLocal instances, which are defined as static variables...

Well, they are all static variables, except these cases:

```
DynamicTablesHelper.context
ForeignNuller.context
GlobalEventManager.context
IdentityPool.context
InMemoryLockManager.context
LockTableUpdater.sessionID
Persistence.context
SequenceIdentityManager.context
SessionManager.activeSession
SyncCoordinatesAspect.contextData
```

If any one of these really needs to be an instance field, then what I proposed in note 37 can not be used.

#39 - 04/07/2016 03:06 PM - Greg Shah

> This way, we can remove the map overhead completely. If the performance is still not improved, then the bottleneck remains the SecurityContextStack.contextPointer, which is a ThreadLocal instance.

I like this idea.

> Well, they are all static variables, except these cases:

Most of these are Eric's to determine.

> SessionManager.activeSession

SessionManager is a singleton and so this instance var is effectively static (and can be made so).

Hynek: can you report on SyncCoordinatesAspect.contextData (see above discussion)?

#40 - 04/07/2016 03:16 PM - Constantin Asofiei

Greg Shah wrote:

> This way, we can remove the map overhead completely. If the performance is still not improved, then the bottleneck remains the SecurityContextStack.contextPointer, which is a ThreadLocal instance.

I like this idea.

Another headache is how to hide/ensure no one else can instantiate a ContextLocal with an ID already assigned for another ContextLocal sub-class. I think it might be possible to build a (ID-to-ContextLocal concrete class) mapping (the concrete class sometimes is anonynmous, sometimes a static inner class) and never allow instantiation of ContextLocal if the mapping doesn't match. As this will be done at instantiation, the overhead should have no impact.

Otherwise, we will have again a security issue similar to what you mentioned in note 21 (related to "system-security-editing" string as key).

#41 - 04/07/2016 03:23 PM - Igor Skornyakov

Constantin Asofiei wrote:

> Another headache is how to hide/ensure no one else can instantiate a ContextLocal with an ID already assigned for another ContextLocal sub-class. I think it might be possible to build a (ID-to-ContextLocal concrete class) mapping (the concrete class sometimes is anonynmous, sometimes a static inner class) and never allow instantiation of ContextLocal if the mapping doesn't match. As this will be done at instantiation, the overhead should have no impact.
>
> Otherwise, we will have again a security issue similar to what you mentioned in note 21 (related to "system-security-editing" string as key).

If I understand you correctly there is a trick I use very often:

```
public class Foo
{
private static final AtomicInteger idgen = new AtomicInteger(0);
public final int id = idgen.incrementAndGet();
....
}
```

#42 - 04/07/2016 03:43 PM - Hynek Cihlar

Greg Shah wrote:

> Hynek: can you report on SyncCoordinatesAspect.contextData (see above discussion)?

I believe all our aspects are singleton and so is SyncCoordinatesAspect.

#43 - 04/07/2016 06:02 PM - Igor Skornyakov

Task branch 2973a was rebased from the trunk revision 10998. Pushed up to revision 11002.

#44 - 04/08/2016 02:10 AM - Eric Faulhaber

Igor Skornyakov wrote:

> However if your tests show that Map.get() is a bottleneck the reason can be that this map is effectively an IdentityMap. May be it is better to consider using keys with good hash codes to avoid degeneration of the map to a linked list?

I should clarify: originally, my profiling pointed to SecurityContext.getToken as a moderate bottleneck. I expect it is better now, but I have not yet had time to profile the new implementation.

> Moreover in some situations (when collision are unavoidable and the Map is updated frequently) I found more practical to use a TreeMap to avoid spikes (of course if the average O(log(n)) access time is acceptable at all).

AFAIK, the token map is updated very infrequently, but read very frequently. In some recent dumps of the server's heap, I did notice the token map had a table of 128 elements and some nodes' linked lists were a few levels deep, so the hashing is not perfect and there is a bit of degeneration. However, that I noticed this was incidental; it was not what I was looking for, so I did not explore the map extensively. It is essentially an identity map, since most token keys are the ContextLocal instances themselves. Some time ago, I cached the ContextLocal hash to avoid a native call each time hashCode was invoked, but that did nothing to make the hash value itself well distributed.

Constantin Asofiei wrote:

> Well, they are all static variables, except these cases:
>
> `DynamicTablesHelper.context`

DynamicTablesHelper follows the singleton pattern, so this ContextLocal could be made static.

> `ForeignNuller.context`

Effectively deprecated.

```
GlobalEventManager.context
IdentityPool.context
InMemoryLockManager.context
LockTableUpdater.sessionID
Persistence.context
SequenceIdentityManager.context
```

Each of these were not made static because the objects they are associated with each correspond with a particular database instance, and there can be multiple of these. There probably is a way around this, but unfortunately it would involve yet another map lookup, which would be a bit clumsy and not very OO.

#45 - 04/08/2016 03:45 AM - Constantin Asofiei

Igor, following Eric's response, we can't go with the approach in note 37/38, as the number of ContextLocal instances is not finite.

But, in any case, tokenMap size should be on average in the 100-200 range, and is still small - the map should not degenerate into a linked-list that fast. Also, only the 6 cases Eric mentioned can cause writing to the tokenMap more than once, but this still is done very infrequently.

Considering this, can you think of a way to build a standalone performance test for the ContextLocal, SecurityContext, SecurityContextStack? This way we can test more easily different approaches to optimize the tokenMap.

#46 - 04/08/2016 04:02 AM - Igor Skornyakov

Constantin Asofiei wrote:

> Igor, following Eric's response, we can't go with the approach in note 37/38, as the number of ContextLocal instances is not finite.
>
> But, in any case, tokenMap size should be on average in the 100-200 range, and is still small - the map should not degenerate into a linked-list that fast. Also, only the 6 cases Eric mentioned can cause writing to the tokenMap more than once, but this still is done very infrequently.
>
> Considering this, can you think of a way to build a standalone performance test for the ContextLocal, SecurityContext, SecurityContextStack? This way we can test more easily different approaches to optimize the tokenMap.

This is a good idea. I will implement such test today. To make it more realistic - how may read/write threads typically access the map?
Thank you.

#47 - 04/08/2016 04:17 AM - Igor Skornyakov

BTW if we are talking about a small map with is very infrequently updated but frequently read may be it makes sense to use theory and consider e.g, Cukoo hashing https://en.wikipedia.org/wiki/Cuckoo_hashing? Of course if the map is a real bottleneck.

#48 - 04/08/2016 06:09 AM - Constantin Asofiei

Igor Skornyakov wrote:

> This is a good idea. I will implement such test today. To make it more realistic - how may read/write threads typically access the map?

4GL is inherently single-threaded, but some features are emulated in P2J asynchronously (for example, the CTRL-C processing). If the server-side Conversation thread is executing business logic (and is not blocked waiting for a response/message from i.e. the client), it can't handle a i.e. CTRL-C request; so, the Dispatcher threads will handle any requests incoming from the client-side: these threads will switch the context to that account and execute the request under that context.

So, in P2J the tokenMap is accessed asynchronously only when more than one thread executes under the same context - and this is usually rare. Even appserver Agents are running in Conversation mode.

Anyway, I think we should test how SecurityManager.getToken() behaves: first use a single context, but access it from one or multiple threads (2-3 is enough) - what matters is execute the getToken() many times. Second, build 10-20 threads (each one with their own context) and access getToken() for each context. Each context should have ~100 ContextLocal instances added to it.

After we have some results for this, we can decide if we need to test the SecurityContext.reset() too, or not.

Also, please keep an eye on the SecurityContextStack.contextPointer - this is the first one accessed when SecurityManager.getToken() is called, and resolving the value for the current thread might be costly, especially when is needed many-many times: P2J relies a lot on accessing context-local data, when using the persistence layer and many-many other cases. So it might be a matter of a very high number of hits on this contextPointer, from multiple threads.

#49 - 04/08/2016 07:13 AM - Igor Skornyakov

I've used YourKit (sampling mode) to analyse server behaviour while running the customer test suite (**/*Test.java) (p2j trunk).

From total 51 min 3o seconds run the server spent 65 seconds in the SecurityContext.getToken() method (in fact in the Map.get())/ This is at least noticeable. All other methos from the com.goldencode.p2j.security package took negligible time with one notable exception: SecurityContextStack.getToken() spent 15 second waiting on monitor on the SecurityContext.getToken() method. This means that protecting the later method with readLock makes sense.
Continue investigation.

#50 - 04/08/2016 08:52 AM - Igor Skornyakov

Igor Skornyakov wrote:

I've made the following changes:

1. Replaced ConcurrentHashMap with HashMap as tokenMap guarded bu a RenntrantReadWriteLock with a standard semantics.
2. Increased the initial Map capacity
3. Introduced a synthetic key for the tokenMap.

See 2973 rev 11003

The test run time is now 37 minutes 54 seconds.
However the time spent in the SecurityContext.getToken() method decreased just to 55 seconds and the time spent by SecurityContextStack.getToken() increased to 20 seconds.

#51 - 04/08/2016 08:54 AM - Constantin Asofiei

Igor Skornyakov wrote:

> However the time spent in the SecurityContext.getToken() method decreased just to 55 seconds and the time spent by SecurityContextStack.getToken() increased to 20 seconds.

I'm curious, do you have a total call number for SecurityContext.getToken() and SecurityContextStack.getToken()?

#52 - 04/08/2016 09:08 AM - Eric Faulhaber

Igor Skornyakov wrote:

> The test run time is now 37 minutes 54 seconds.
> However the time spent in the SecurityContext.getToken() method decreased just to 55 seconds and the time spent by SecurityContextStack.getToken() increased to 20 seconds.

I've never seen this level of improvement from run to run. I wonder if the reduction in overall time is due largely to database caching. When I'm measuring performance, I usually bounce the database cluster before each test run. Other file system caching may come into play as well.

#53 - 04/08/2016 10:22 AM - Hynek Cihlar

Btw., have you considered making the locking of reading operations more optimistic? For example StampLock allows the 'sequence locking' mechanism where the reader never blocks but the read operation may have to retry if a write lock is acquired. For this read-heavy workload this could rule out almost all of the heavy synchronizations.

#54 - 04/08/2016 10:31 AM - Igor Skornyakov

Hynek Cihlar wrote:

> Btw., have you considered making the locking of reading operations more optimistic? For example StampLock allows the 'sequence locking' mechanism where the reader never blocks but the read operation may have to retry if a write lock is acquired. For this read-heavy workload this could rule out almost all of the heavy synchronizations.

I was told that the update are rare. In the last test the reads used readLock so there where no contention between readers.

#55 - 04/08/2016 10:49 AM - Hynek Cihlar

Igor Skornyakov wrote:

> Hynek Cihlar wrote:
>
> > Btw., have you considered making the locking of reading operations more optimistic? For example StampLock allows the 'sequence locking' mechanism where the reader never blocks but the read operation may have to retry if a write lock is acquired. For this read-heavy workload this could rule out almost all of the heavy synchronizations.
>
> I was told that the update are rare. In the last test the reads used readLock so there where no contention between readers.

In the most-read scenarios, StampLock's optimistic lock should outperform ReadWriteLock. See this http://blog.takipi.com/java-8-stampedlocks-vs-readwritelocks-and-synchronized/.

#56 - 04/08/2016 11:57 AM - Igor Skornyakov

Hynek Cihlar wrote:

> Igor Skornyakov wrote:
>
> > Hynek Cihlar wrote:
> >
> > > Btw., have you considered making the locking of reading operations more optimistic? For example StampLock allows the 'sequence locking' mechanism where the reader never blocks but the read operation may have to retry if a write lock is acquired. For this read-heavy workload this could rule out almost all of the heavy synchronizations.
> >
> > I was told that the update are rare. In the last test the reads used readLock so there where no contention between readers.
>
> In the most-read scenarios, StampLock's optimistic lock should outperform ReadWriteLock. See this http://blog.takipi.com/java-8-stampedlocks-vs-readwritelocks-and-synchronized/.

Thank you Hynek. I will take a look.

#57 - 04/08/2016 12:07 PM - Igor Skornyakov

After a close look at the SecurityContext class I got an impression that if the tokenMap is a ConcurrentMap we do not need an additional synchronization at all (well, at least if we drop the double check of cleanup which produces something useful in debug log only).

With the reworked code the only method which the profiler can notice in the SecurityContext is cleanup. All the methods from the com.goldencode.p2j.security package combined (except AssociatedThread.run()) took less than 3 seconds.

The number of test failures with these changes look the same as before.

See 2973a revision 11004.

#58 - 04/08/2016 12:52 PM - Eric Faulhaber

Igor Skornyakov wrote:

> After a close look at the SecurityContext class I got an impression that if the tokenMap is a ConcurrentMap we do not need an additional synchronization at all (well, at least if we drop the double check of cleanup which produces something useful in debug log only).
>
> With the reworked code the only method which the profiler can notice in the SecurityContext is cleanup. All the methods from the com.goldencode.p2j.security package combined (except AssociatedThread.run()) took less than 3 seconds.
>
> The number of test failures with these changes look the same as before.
>
> See 2973a revision 11004.

Nice result.

Unfortunately, that "SecurityContext.cleanupWorker did not complete properly..." message is not merely academic; in the past, it has appeared numerous times in the log and prompted investigations. Also, what about the clearing of the token map and putting all the cloned tokens back in (lines 570-1)? This seems important to a context reset if there were failed tokens, so as not to leak those failed tokens. Can anyone think of a safe way to preserve this functionality (or something similar) without disrupting the performance gains already made?

BTW, the new ContextKey class is missing from the branch.

#59 - 04/08/2016 03:06 PM - Constantin Asofiei

Igor, the code you commented from cleanupWorker was responsible for two things:

1. ensure that, after cleanup, tokenMap will contain only context-local instances which must NOT be cleaned (so they need to survive the context reset, no matter what). On a side note, the NPE in note 23 was cleaning up a context-local instance which should not have been cleaned. Also,

this code:

```
// even if reset was not successful, ensure only the expected tokens will survive the
// reset
tokenMap.clear();
tokenMap.putAll(clone);
```

is critical and responsible for ensuring the tokenMap contains only the ContextLocal vars which need to survive the reset, regardless if the reset had problems or not.

2. beside this, it provides a mechanism to track dependency problems during the context-local reset. If the order of cleaning up context-local vars is not OK, we may end up with a ContextLocal var accessing (and re-initializing) an already cleaned up context-local var (CL-Var1 needs CL-Var2 during reset, but CL-Var2 is cleaned up before CL-Var1 - thus CL-Var2 is re-initialized and re-added to the token map, and "survives" the context-reset). Also, this is another reason why we make a clone of the tokenMap and use that to determine which vars need to survives; after cleaning up, by comparing the content of the clone and the current tokenMap, we can find such problems.

So, I think this should work:

1. this code is needed, as it is critical for tokenMap to contain only the expected ContextLocal vars (which must survive reset), and nothing else:

```
// even if reset was not successful, ensure only the expected tokens will survive the
// reset
tokenMap.clear();
tokenMap.putAll(clone);
```

2. the other part - which compares the current tokenMap with the content of the clone, after cleanup - can be re-written so that it uses another snapshot/copy of the current tokenMap (after cleanupWorker(reset, clone); was executed). This way, by using a non-concurrent copy of the tokenMap, we should avoid the synchronization overhead.

#60 - 04/08/2016 03:17 PM - Igor Skornyakov

Constantin Asofiei wrote:

> So, I think this should work:
>
> 1. this code is needed, as it is critical for tokenMap to contain only the expected ContextLocal vars (which must survive reset), and nothing else:
> [...]

How this approach works with the situation when something was added to the tokenMap after the clone was populated? Actually in my version the cleanupWorker(boolean reset, Map<ContextKey, Object> tm) returns the set of keys which where removed from the clone and tokenMap.keySet().removeAll(removed); removes them from the tokenMap. This is also not perfect (consider situation when the value was replaced), but seems a little bit safer than clear/addAll.

1. the other part - which compares the current tokenMap with the content of the clone, after cleanup - can be re-written so that it uses another snapshot/copy of the current tokenMap (after cleanupWorker(reset, clone); was executed). This way, by using a non-concurrent copy of the tokenMap, we should avoid the synchronization overhead.

Agree.

#61 - 04/08/2016 03:35 PM - Constantin Asofiei

Igor Skornyakov wrote:

> How this approach works with the situation when something was added to the tokenMap after the clone was populated?

When the context is reset, the contract is that no other thread will do work with this context (the appserver Agent is just finishing execution after a request). So the clone snapshot will be valid, in terms of mirroring what the tokenMap holds at the end of an appserver Agent job.

> Actually in my version the cleanupWorker(boolean reset, Map<ContextKey, Object> tm) returns the set of keys which where removed from the clone and tokenMap.keySet().removeAll(removed); removes them from the tokenMap. This is also not perfect (consider situation when the value was replaced), but seems a little bit safer than clear/addAll.

Well, your version doesn't take into account the case when there is a problem in the context-local var ordering by their "weight", and an already-cleaned context-local var (CL-Var2) is re-instantiated and re-added to the tokenMap, when another var (CL-Var1) is being cleaned (as CL-Var2 should have been cleaned after CL-Var1) - so tokenMap may end up with a "surviving" CL-Var2, when it should not have been there. To ensure the context is in the correct state after reset (regardless if we have problems during cleanup or not), is best to let tokenMap have only what we really want to have - the vars which do not need reset.

#62 - 04/08/2016 03:51 PM - Igor Skornyakov

Constantin Asofiei wrote:

> How this approach works with the situation when something was added to the tokenMap after the clone was populated?

When the context is reset, the contract is that no other thread will do work with this context (the appserver Agent is just finishing execution after a request). So the clone snapshot will be valid, in terms of mirroring what the tokenMap holds at the end of an appserver Agent job.

Well, your version doesn't take into account the case when there is a problem in the context-local var ordering by their "weight", and an already-cleaned context-local var (CL-Var2) is re-instantiated and re-added to the tokenMap, when another var (CL-Var1) is being cleaned (as CL-Var2 should have been cleaned after CL-Var1) - so tokenMap may end up with a "surviving" CL-Var2, when it should not have been there. To ensure the context is in the correct state after reset (regardless if we have problems during cleanup or not), is best to let tokenMap have only what we really want to have - the vars which do not need reset.

Sorry Constantin, I do not understand. If we are sure that tokenMap is not changed during the cleanup then two approaches look logically equivalent (remember that original version didn't keep lock for the whole cleanup operation).

1. Make a copy (clone); cleanup clone; replace the content of the tokenMap with the content of clone
2. Make a copy (clone); cleanup clone and return the set of removed keys; remove these keys from the tokenMap.
   The last step can be replaced with tokenMap.keySet().retainAll(clobe.keySet()) or event a loop over tokenMap which will remove those keys which are not in the clone (retainAll) **and** replacing the values of the remaining with ones from the clone.

I do not want to say that I've found a final solution. It is possible that under some special circumstances it requires more elaborate approach. My point was that ConcurrentMap w/o additional synchronisation provides substantial perforamance improvement in a general case.

#63 - 04/08/2016 04:08 PM - Constantin Asofiei

Igor Skornyakov wrote:

> Sorry Constantin, I do not understand. If we are sure that tokenMap is not changed during the cleanup ...

tokenMap is guaranteed to not be changed by other threads. But is not guaranteed that will not be changed if there are problems/bugs in the ordering of the context-local vars, during their cleanup. For example, take ConnectionManager.local - its cleanup method accesses the DatabaseManager.deregisterDatabase API, which in turn uses the DatabaseManager.context variable. So, DatabaseManager.context can't be cleaned before ConnectionManager.local. If it was the other way around, and DatabaseManager.context was cleaned before ConnectionManager.local, then when ConnectionManager.local is cleaned, DatabaseManager.context will be re-added to the tokenMap, when it should not have been there.

The replace the content of the tokenMap with the content of clone approach protects against any other cases where cleanup is not done properly, and vars are re-added to the tokenMap. The SecurityContext.cleanupWorker did not complete properly messages tell us there are problems, but if the context is not in the appropriate state after reset, further usage of that Agent context may result in other errors. This helps us also to not chase non-problems, because of the incorrect state of the Agent's context.

#64 - 04/08/2016 04:11 PM - Constantin Asofiei

PS: if you take a look at ContextLocal$Wrapper.cleanup, you will see that after the var is cleaned, it is removed from the tokenMap:

```
                    security.removeToken(ContextLocal.this);
```

So, as vars are cleaned, they are removed; we want to ensure, if they are ~~not~~ re-added incorrectly, they will not survive after cleanupWorker (as clone will not contain them).

#65 - 04/08/2016 06:11 PM - Igor Skornyakov

Constantin Asofiei wrote:

> tokenMap is guaranteed to not be changed by other threads. But is not guaranteed that will not be changed if there are problems/bugs in the ordering of the context-local vars, during their cleanup.

Sorry, how it can happen? Every action is performed in some thread. The two statements above seem to contradict each other. Please clarify. Thank you.

#66 - 04/09/2016 03:28 AM - Igor Skornyakov

I've performed majic regression testing of 2973a revision 11004.

1. all CTRL-C tests passed
2. from the main part the following 3 tests failed: tc_dc_slot_027, tc_dc_slot_029 and tc_job_002

#67 - 04/09/2016 06:15 AM - Constantin Asofiei

Igor Skornyakov wrote:

> Constantin Asofiei wrote:
>
> > tokenMap is guaranteed to not be changed by other threads. But is not guaranteed that will not be changed if there are problems/bugs in the ordering of the context-local vars, during their cleanup.
>
> Sorry, how it can happen? Every action is performed in some thread. The two statements above seem to contradict each other. Please clarify.

What I'm talking about is the reset of an appserver Agent's context: at the time the context is reset, no other thread should do work with that Agent. So the reset is assumed to be on only one thread.

Also, to put into other words what I described in note 63:

1. in best case scenario, all variables which need reset are cleaned up properly (and removed from the tokenMap), and the tokenMap at the end of cleanupWorker remains with only the variables which do not need reset.
2. in worst case scenario, there are problems during the cleanup of context-local vars, and at the end of the cleanupWorker, tokenMap contains variables which should not have been there.

We are coding cleanupWorker for the worst case scenario: we ensure the tokenMap contains only variables which do not need reset, aka the content of the clone map.

#68 - 04/10/2016 06:22 PM - Igor Skornyakov

I've restores the post-cleanup check.

Committed to the task branch 2973a revision 11006.

Automated regression test passed.

#69 - 04/11/2016 04:08 AM - Igor Skornyakov

Task branch 2973a was rebased from the trunk revision 11000. Pushed up to revision 11008.

#70 - 04/11/2016 08:29 AM - Eric Faulhaber

Constantin, would you please review the latest version?

#71 - 04/11/2016 09:39 AM - Constantin Asofiei

Review for 2973a rev 11008:

1. ContextKey
   - we are using AtomicInteger as a generator. My worries are about very long-running instances of a P2J client, where this might wrap. Although the risk of a key collision is small for normal clients (as they are expected to be short-lived), the appserver Agents are expected to be long-lived; even if they are creating at most 10 new keys per cycle (if the context is reset and rebuilt, for the context-local vars which are defined as instance fields), the cases where ContextLocal vars are defined as static will remain with small IDs (i.e. 0-119) and if it wraps, we will get collisions at some point. I think is OK to check into tokenMap if the key is already used, as the context-local var creation is done infrequently (and the overhead should be minimal).
   - for security reasons, the hashCode and equals methods must be made final. This way, no one can "steal" an existing ID and check into the security context.
2. ContextLocal - remove the commented code
3. SecurityContext
   - import java.util.concurrent.locks.*; is no longer needed
   - remove the commented code
4. SecurityContext.cleanupWorker - if cleanupWorker(boolean, Map) is returning the keys which have been removed (and are also removed them from from tokenMap), then removing them from the clone is no longer necessary (and adds unneeded overhead).
5. SecurityContextStack - you have two history entries, please merge them.

rev 11009 contains some coding standards/formatting fixes in SecurityContext and ContextKey.

#72 - 04/11/2016 09:54 AM - Constantin Asofiei

Another issue in SecurityContext.transferAllTokens - removing an element from the map while the map is processed (via stream()) is possible for ConcurrentHashMap, but is not possible for HashMap. So this code:

```
          Object val = tokenMap.remove(key);
```

will throw a ConcurrentModificationException if a HashMap (or maybe other map implemention) is used. Please add a comment there, to warn future readers that this works only for ConcurrentHashMap and if the type of tokenMap is changed, this code might need changes, too.

#73 - 04/11/2016 10:54 AM - Igor Skornyakov

Constantin Asofiei wrote:

> Review for 2973a rev 11008:
>
> 1. ContextKey
> - we are using AtomicInteger as a generator. My worries are about very long-running instances of a P2J client, where this might wrap. Although the risk of a key collision is small for normal clients (as they are expected to be short-lived), the appserver Agents are expected to be long-lived; even if they are creating at most 10 new keys per cycle (if the context is reset and rebuilt, for the context-local vars which are defined as instance fields), the cases where ContextLocal vars are defined as static will remain with small IDs (i.e. 0-119) and if it wraps, we will get collisions at some point. I think is OK to check into tokenMap if the key is already used, as the context-local var creation is done infrequently (and the overhead should be minimal).

I understand the concern. However it is easy to calculate that if a new instance of the ContextKey will be generated every second it will take 270 years to use all possible values. If this still looks like a risk it is easier to use long id and AtomicLong generator. This should not add any noticeable overhead.

The rest of the issue mentioned he and in the note 72 has been fixed. Thank you Constantin.

Committed to the task branch 2973a revision 11010.

#74 - 04/11/2016 11:24 AM - Constantin Asofiei

Igor Skornyakov wrote:

> I understand the concern. However it is easy to calculate that if a new instance of the ContextKey will be generated every second it will take 270 years to use all possible values. If this still looks like a risk it is easier to use long id and AtomicLong generator. This should not add any noticeable overhead.

Regardless if we switch to long, the hash key will still be an integer value (this is what hashCode can return...). I think in your calculation you doubled the amount of values available in a 32-bit space - there are only ~4.3 billion, which reduces your estimate to ~130 years.

Anyway, can you test the overhead of adding the "key exists" check, with the customer test suite? Because I still think is best to eliminate completely a problem, instead of relying on "chance", if the additional overhead is small.

BTW, you still have some commented code in ContextLocal (equals and hashCode methods).

#75 - 04/11/2016 11:38 AM - Igor Skornyakov

Constantin Asofiei wrote:

> Regardless if we switch to long, the hash key will still be an integer value (this is what hashCode can return...). I think in your calculation you doubled the amount of values available in a 32-bit space - there are only ~4.3 billion, which reduces your estimate to ~130 years.

Yes, it was my mistake, but I think that 138 years is still a long time which as far beyond any reasonable estimation of a non-stop run duration. Please note also that to cause an issue the id should not only be re-used but the previous value should be still in use. I do not understand the argument regarding the hashCode value - the additional (CPU) overhead of replacing id type to long is just a cost of comparing int and long which is negligible.

> Anyway, can you test the overhead of adding the "key exists" check, with the customer test suite? Because I still think is best to eliminate completely a problem, instead of relying on "chance", if the additional overhead is small.

Sorry, I do not understand how can we check that the key already exists? And I cannot agree that I suggest to rely on a chance - it is just a realistic estimation.

> BTW, you still have some commented code in ContextLocal (equals and hashCode methods).

Sorry. Fixed in revision 11011.

#76 - 04/11/2016 01:29 PM - Constantin Asofiei

Igor Skornyakov wrote:

> I do not understand the argument regarding the hashCode value - the additional (CPU) overhead of replacing id type to long is just a cost of comparing int and long which is negligible.

Ahhh... I was thinking that Object.hashCode() must return int, but I forgot the map implementations do not rely only on hashCode, they also check equals... so using long will be OK and will make the probability for collisions nonexistent.

#77 - 04/11/2016 01:38 PM - Igor Skornyakov

The ContextKey.id is long in revision 11012.

#78 - 04/11/2016 03:07 PM - Eric Faulhaber

Please address the remaining code review issues.

Constantin, I don't think the changes made from this review warrant re-testing, do you?

#79 - 04/11/2016 03:40 PM - Constantin Asofiei

Eric Faulhaber wrote:

> Please address the remaining code review issues.

All issues are fixed now.

> Constantin, I don't think the changes made from this review warrant re-testing, do you?

No, the changes don't require testing. I'm surprised that there were no other "false-negative" failures (as the context-local lookup was improved), maybe test dependency/ordering is stable now.

#80 - 04/11/2016 04:02 PM - Eric Faulhaber

Please merge to trunk. Thank you.

#81 - 04/11/2016 04:07 PM - Igor Skornyakov

Task branch 2973a was merged to the trunk revision 11001 and archived.

#82 - 04/11/2016 05:59 PM - Greg Shah

> but I think that 138 years is still a long time which as far beyond any reasonable estimation of a non-stop run duration

I agree that this is a very unlikely scenario. However, it is possible to see some rare case where keys are generated much more than 1 per second. What if someone mis-coded the construction of a context local such that we were generating millions of instances? In such a case, on a long running server it is possible to see this "should never happen" case.

The problem with this very rare case is that it will be a very nasty thing to try to debug. By its nature, it will only happen in some extreme runtime case and the effect will be very subtle because we will appear to have no direct recreate but every once in a long while we will get a report of some ClassCastException or some other thing that should not be possible.

Considering that the check for this is low cost and should not be in the most common use case (the get()), I would like you to please fix this such that it can never occur. I know it probably will never occur, but if it ever did it would be a very costly problem to diagnose. Solving it permanently is the better plan.

Please create a 2973b branch and put that fix in.

Sorry, I've been in meetings all day otherwise I would have told you this before you merged to trunk.

#83 - 04/11/2016 06:16 PM - Igor Skornyakov

Greg Shah wrote:

> Considering that the check for this is low cost and should not be in the most common use case (the get()), I would like you to please fix this such that it can never occur. I know it probably will never occur, but if it ever did it would be a very costly problem to diagnose. Solving it permanently is the better plan.
>
> Please create a 2973b branch and put that fix in.

Greg,
In the code which was merged the ContextKey.id is long. This makes the collision completely impossible at the cost of a negligible additional overhead (comparing long instead of int for equality)
Of course I can add a check to a ContextKey constructor to ensure the the id is not zero. May be it makes sense to add it to the 3035a branch as the change is very small (one line) and non critical?
Thank you.

#84 - 04/11/2016 06:20 PM - Greg Shah

I'm referring to the wrapping case where an id gets reused because the long overflows and some long-running existing key already has that value. This is protected by adding a "key exists" check as mentioned by Constantin in note 74.

#85 - 04/11/2016 06:20 PM - Greg Shah

> May be it makes sense to add it to the 3035a branch as the change is very small (one line) and non critical?

Yes, that is fine.

#86 - 04/11/2016 06:27 PM - Igor Skornyakov

Greg Shah wrote:

> I'm referring to the wrapping case where an id gets reused because the long overflows and some long-running existing key already has that value. This is protected by adding a "key exists" check as mentioned by Constantin in note 74.

Sorry Greg. I do not understand what you and Constantin mean exactly. The SecurityContext.addToken method already rejects duplicated keys and returns false in such a situation. Do you want to treat such a situation as a severe error and e.g.throw an exception?

#87 - 04/11/2016 06:32 PM - Igor Skornyakov

Igor Skornyakov wrote:

> Sorry Greg. I do not understand what you and Constantin mean exactly. The SecurityContext.addToken method already rejects duplicated keys and returns false in such a situation. Do you want to treat such a situation as a severe error and e.g.throw an exception?

Please note also that even if a new instance of the ContextKey will be generated every **nanosecond** it will take centuries to get a wrap up of the long counter.

#88 - 04/11/2016 07:15 PM - Greg Shah

The problem is that one can create a ContextKey instance which fails the addToken() and then later can call getToken() and a different (unexpected) value will be returned.

I know it is not something that should happen. But why not make it bulletproof so that something unexpected never occurs? Instead of an exception being thrown, just get a new ID until a check for existence shows that the new ID is unique. Since this only happens on the add and not the get, it is not expected to cost enough to be an issue.

#89 - 04/12/2016 03:41 AM - Igor Skornyakov

Greg Shah wrote:

> The problem is that one can create a ContextKey instance which fails the addToken() and then later can call getToken() and a different (unexpected) value will be returned.
>
> I know it is not something that should happen. But why not make it bulletproof so that something unexpected never occurs? Instead of an exception being thrown, just get a new ID until a check for existence shows that the new ID is unique. Since this only happens on the add and not the get, it is not expected to cost enough to be an issue.

Greg,
Of course this can be done very easily. However I do not think that it is a good idea. First of all the ContextKey is immutable as it is desirable for the Map key. Imagine also that some key (perfectly valid and unique) will be registered twice because of the program bug, not the counter wrap. If course the change of the id in this case will not help as it will change the key which is already registered as well and we'll be in an infinite loop.
If we want to discover the fact of the counter's wrap early it is sufficient just to check the the next value is not zero and it can be made in a ContextKey constructor.

#90 - 04/12/2016 03:53 AM - Igor Skornyakov

Igor Skornyakov wrote:

> Of course this can be done very easily. However I do not think that it is a good idea. First of all the ContextKey is immutable as it is desirable for the Map key. Imagine also that some key (perfectly valid and unique) will be registered twice because of the program bug, not the counter wrap. If course the change of the id in this case will not help as it will change the key which is already registered as well and we'll be in an infinite loop.

Sorry, of course we'll do not have an infinite loop - just a corrupted map (not much better).

#91 - 04/12/2016 05:33 AM - Greg Shah

The idea is simply to never set a ContextKey.id that is already in use. No one wants the id to be mutable.

#92 - 04/12/2016 05:47 AM - Igor Skornyakov

Greg Shah wrote:

> The idea is simply to never set a ContextKey.id that is already in use. No one wants the id to be mutable.

Sorry, how it is possible to check that the code is not in use? For a particular instance of the SecurityContext is it already done (but the result seems to be ignored). It is possible of course to implement some complicated logic based e.g. on a Bloom filter but why do you think that a simple non-zero test I've suggested is not sufficient?
Thank you,

#93 - 04/12/2016 06:56 AM - Greg Shah

We aren't checking that code is not in use. At construction of the ContextKey we can simply ask the SecurityContext for the next **available** id. In other words, an id that is known to NOT be in the tokenMap.

Your approach detects if wrapping has occurred and would have to abort further processing. My approach solves the problem and allows the system to keep running indefinitely. Since the ContextKey construction should not be a performance sensitive path, this check if exists approach should not impact performance noticeably.

#94 - 04/12/2016 07:00 AM - Igor Skornyakov

Greg Shah wrote:

> We aren't checking that code is not in use. At construction of the ContextKey we can simply ask the SecurityContext for the next **available** id. In other words, an id that is known to NOT be in the tokenMap.
>
> Your approach detects if wrapping has occurred and would have to abort further processing. My approach solves the problem and allows the system to keep running indefinitely. Since the ContextKey construction should not be a performance sensitive path, this check if exists approach should not impact performance noticeably.

I see. I will think about this. One question: how many instances of the ContextKey are expected to be in use at the same time? Just up to an order of magnitude.
Thank you.

#95 - 04/12/2016 07:02 AM - Greg Shah

> how many instances of the ContextKey are expected to be in use at the same time? Just up to an order of magnitude.

At this time, in the hundreds. I would not expect even 500 in our current implementation, but it could easily approach 150 or more.

#96 - 04/12/2016 07:03 AM - Igor Skornyakov

Greg Shah wrote:

> how many instances of the ContextKey are expected to be in use at the same time? Just up to an order of magnitude.

At this time, in the hundreds. I would not expect even 500 in our current implementation, but it could easily approach 150 or more.

Thank you Greg.

#97 - 04/12/2016 09:39 AM - Igor Skornyakov

Added ContextKey re-use protection.
Committed to the task branch **3035a** revision 11005.

#98 - 04/12/2016 10:44 AM - Greg Shah

Code Review Task Branch 3035a Revision 11005

1. keysInUse is missing javadoc.

2. ContextKey.toString() opening curly brace is on the wrong line.

#99 - 04/12/2016 11:06 AM - Igor Skornyakov

Greg Shah wrote:

> Code Review Task Branch 3035a Revision 11005
>
> 1. keysInUse is missing javadoc.
>
> 2. ContextKey.toString() opening curly brace is on the wrong line.

Fixed. Committed to the task branch **3035a** revision 11006.

#100 - 04/12/2016 12:55 PM - Greg Shah

- Status changed from WIP to Closed

- % Done changed from 0 to 100

Code Review Task Branch 3035a Revision 11006

The changes are fine.

I'm closing this task. If there are problems, we will see them during testing for 3035a.

#101 - 11/16/2016 12:31 PM - Greg Shah

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

#102 - 01/03/2018 02:11 PM - Greg Shah

- Related to Feature #3246: reduce the amount of data being sent to the client-side when an UI attribute is being changed added