Is it possible to create a ThreadLocal for the carrier thread of a Java virtual thread?

Asked 1 year, 11 months ago Modified 1 year, 9 months ago Viewed 5k times

18

JEP-425: Virtual Threads states that a new virtual thread "should be created for every application task" and makes twice a reference to the possibility of having "millions" of virtual threads running in the JVM.

The same JEP implies that each virtual thread will have access to its own thread-local value:

> Virtual threads support thread-local variables [...] just like platform threads, so they can run existing code that uses thread locals.

Thread locals are many times used for the purpose of caching an object that is not thread-safe and expensive to create. The JEP warns:

> However, because virtual threads can be very numerous, use thread locals after careful consideration.

Numerous indeed! Especially given how virtual threads are not pooled (or at least shouldn't be). As representative of a short-lived task, using thread locals in a virtual thread for the purpose of caching an expensive object seems to be borderline void of meaning. Unless! We can from a virtual thread create and access thread locals bound to its carrier thread 🤔

For clarification, I would like to go from something like this (which would have been perfectly acceptable when using only native threads capped to the size of a pool, but this is clearly no longer a very effective caching mechanism when running millions of virtual threads continuously re-created:

```java
static final ThreadLocal<DateFormat> CACHED =
ThreadLocal.withInitial(DateFormat::getInstance);
```

To this (alas this class is not part of the public API):

```java
static final ThreadLocal<DateFormat> CACHED = new
jdk.internal.misc.CarrierThreadLocal();
// CACHED.set(...)
```

Before we even get there. One must ask, is this a safe practice?

Well, as far as I have understood virtual threads correctly, they are merely logical stages executed on a platform thread (aka. the "carrier thread") with the ability to unmount instead of being blocked waiting. So I am assuming - please correct me if I am wrong - that 1) the virtual thread will never be interleaved by another virtual thread on the same carrier thread or rescheduled on another carrier thread *unless the code would have blocked otherwise* and therefore, if 2) the operation we invoke on the cached object never blocks, then the task/virtual thread will simply run from start to finish on the same carrier and so yes, it would be safe to cache the object on a platform thread-local.

With the risk of answering my own question, JEP-425 indicates this is not possible:

> Thread-local variables of the carrier are unavailable to the virtual thread, and vice-versa.

I could not find a public API to get the carrier thread or allocate thread locals explicitly on a platform thread [from a virtual thread], but that's not to say my humble research is definitive. Maybe there is a way?

Then I read [JEP-429: Scoped Values]() which at first glance seems to be a stab by the Java Gods to get rid of the `ThreadLocal` altogether, or at least provide an alternative for virtual threads. In fact, the JEP uses verbiage such as "migration to scoped values" and says they are "preferred to thread-local variables, especially when using large numbers of virtual threads".

For all the use cases discussed in the JEP, I can only agree. But towards the bottom of this document, we also find this:

> There are a few scenarios that favor thread-local variables. An example is caching objects that are expensive to create and use, such as instances of java.text.DateFormat. Notoriously, a DateFormat object is mutable, so it cannot be shared between threads without synchronization. Giving each thread its own DateFormat object, via a thread-local variable that persists for the lifetime of the thread, is often a practical approach.

In light of what was discussed earlier, using a thread-local may be "practical" but not very ideal. In fact, JEP-429 itself actually started off with a very telling remark: "if each of a million virtual threads has mutable thread-local variables, the memory footprint may be significant".

To summarize:

Have you found a way to allocate thread locals on a carrier thread from the virtual thread?

If not, is it safe to say that for applications using virtual threads, the practice of caching objects in a thread-local is dead and one will have to implement/use a different approach such as a concurrent cache/map/pool/whatever?

java caching thread-local virtual-threads

Share
Improve this question
Follow

edited Jan 8, 2023 at 13:54

asked Jan 8, 2023 at 11:56
Martin Andersson
19.5k ● 10 ● 91 ● 119

2 Answers

Sorted by: Highest score (default)

21

You wrote

> So I am assuming - please correct me if I am wrong - that
>
> 1. the virtual thread will never be interleaved by another virtual thread on the same carrier thread or rescheduled on another carrier thread *unless the code would have blocked otherwise* and therefore, if
> 2. the operation we invoke on the cached object never blocks, then the task/virtual thread will simply run from start to finish on the same carrier and so yes, it would be safe to cache the object on a platform thread-local.

But the State of Loom document states:

> You must not make any assumptions about where the scheduling points are any more than you would for today's threads. Even without forced preemption, any JDK or library method you call could introduce blocking, and so a task-switching point.

and further:

> To that end, we plan for the VM to support an operation that tries to forcefully preempt execution at any safepoint. How that capability will be exposed to

> the schedulers is TBD, and will likely not make it to the first Preview.

So

1. The assumption that a virtual thread only releases the carrier thread when it is about to be blocked, only applies to the current preview. Preemptive switching between virtual threads is allowed and even planned for the future.
2. Even if we assume that a virtual thread can only release the carrier thread when performing blocking operations, we can't predict when a blocking operation may occur.
   - One example of operations outside our control, is *class loading*. Loading class data is a blocking operation and class loading is implemented lazily for common JVMs. It's even possible that a method that has been invoked multiple times suddenly executes an uncommon path that uses a class that has not been used before.
   - Another example is *resource loading*. Even an example as simple as your `DateFormat` already involves resources organized in an unspecified way, time zone data or localized month and weekday names, for example.

So, there's no way of having a safely working carrier-local cache and your assumption that using thread locals (or alike) for caching is dead, is indeed right. You may use an object pool instead, but since this implies some sort of synchronization, you might as well consider just using a single `DateFormat` [1] and synchronize on it. This would implement your initial idea of not releasing the carrier thread during the use of the object.

Of course, in this specific example, the better option is to use a `DateTimeFormatter` from the `java.time` API which is thread safe and hence, allows a single instance to be shared by all threads.

[1] or one of multiple, selected in a way that does not involve synchronization

Share
Improve this answer
Follow

edited Mar 17, 2023 at 12:58

answered Mar 17, 2023 at 12:48
Holger
297k ● 42 ● 468 ● 807

Wow, what an excellent answer Holger. I learned something new today, and I am sure your answer will propel many more developers to advance their skills for a long time to come! If I could upvote this answer five times I would. – Martin Andersson Mar 18, 2023 at 13:11

Honestly, I am not sure if I understand your question right. [JEP 429](JEP 429) states the following:

2

- "Every thread-local variable is mutable" and "The thread-local variable serves as a kind of hidden method argument" but it is quite clear that " this can lead to spaghetti-like data flow" and much discipline and some engineering effort are recommended. Some best practices or more restrictions might be helpful ...
- "... if each of a million virtual threads has mutable thread-local variables, the memory footprint may be significant." This raises the question of how an implementation of an algorithm should be designed, taking advantage of the parallelization with virtual threads.
- "The Java Platform should provide a way to maintain immutable and inheritable per-thread data for thousands or millions of virtual threads." This is a comprehensible demand, but it seems that the authors are not sure and some more conceptual work is needed.

Now getting back to your question concerning the "allocation of thread locals on a carrier thread from the virtual thread". Why would you need that? And caching objects in a thread-local makes sense in some scenarios (like caching the principal for the DB access in the example). Since virtual threads are relatively new (Java 19), it might be a bit early to declare thread-locals dead.

Share Improve this answer Follow

answered Jan 8, 2023 at 12:32
Roland J.
86 ● 4

Bullet point 1 I don't understand. My post is not talking about what Scoped Values can be used for? They can be used as a direct replacement for many of the patterns were we used to use a `ThreadLocal`. So that's great! – Martin Andersson Jan 8, 2023 at 13:23

Bullet point nr 2 I also don't understand. Implementation of what algorithm?? TBH, virtual threads and JEP-428: Structured Concurrency significantly lowers the complexity of multithreaded programming in Java. – Martin Andersson Jan 8, 2023 at 13:23

1 The solution I see is either binding the thread-local to the carrier thread (my posted question is asking whether or not this is possible using a JDK-provided API) or simply stop using thread locals for the purpose of caching objects and instead use an alternative, such as an object pool. – Martin Andersson Jan 8, 2023 at 13:34

3 OK, thanks for clarification. I don't know about "binding the thread-local to the carrier thread" and it seems to me that an object pool/cache is the appropriate way. Maybe I am wrong but doesn't the name "thread-local" implies that the local var is isolated from the carrier thread? Otherwise it wouldn't be local, right? – Roland J. Jan 8, 2023 at 13:48

1 I posted some example code to get a better understanding of the problem but it was deleted by the admins. So I am afraid we cannot continue this interesting discourse. – Roland J. Jan 9, 2023 at 6:49