Including the ThreadContext when writing log4j2 logs via a Java Static method - Is it thread safe?

Ask Question

In a web service application I am using a static method to set log4j ThreadContext variables for logging purposes as below,

```
public static void setLogParams(String company_id, String userId) {
    ThreadContext.put("company_id_val", company_id);
    ThreadContext.put("user_id_val", userId);
}
```

Each and every web service requests will be call above method initially and then `loj4j Logger object` will use for do the rest. Above given values are not same every time and may vary request to request.

My question: Is above scenario is thread safe? Will different web service requests share the same `company_id` and `user_id` since both of these parameters are holding same references? Then it will be confusing. Should I use a non-static method instead?

I did gone through the similar question below

Are non-synchronised static methods thread safe if they don't modify static class variables?

But I need to clarify this.

java web-services concurrency thread-safety log4j

edited May 23 '17 at 10:31
Community ♦
1 1

asked Jul 3 '15 at 8:28
Supun
60 12

2 Answers

In short

It's safe.

Explanation

`ThreadContext` is a map that is scoped to every thread. Casually spoken, every thread has its own instance of a map, and the values are not visible to the other threads. See https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html for more details on the topic of thread locals. It does not matter, whether the method, in which you set the `ThreadContext` is static or not.

There is one caveat: Java application servers/web servers use thread pooling and reuse threads. This means once a thread finished it's web request, it is reused to process the next request. The data within the `ThreadContext` remains and is valid for the next request. Retaining data for the next request is usually not desirable. You should clear the `ThreadContext` after you're done with the request:

```
try{
    setLogParams(company_id, userId);
    ... // do your business logic
} finally {
    clearLogParams(); // Something like ThreadContext.clear();
}
```

Update: I did not cover the static/unsynchronized method part but here's the answer to this:

Your method does not store the parameters in your own storage; rather it stores them within the `ThreadContext` . Arguments (no matter whether they are in a static or non-static method) do not overlap between calls and threads. The lifecycle of an argument ends with the end of the method call (as long as you do not store the data, but references are a different topic).

You would run into race conditions if you stored the `customer_id` for example by your own in a static variable:

```
class RequestDataHolder {
   // this is not thread safe since multiple threads access the same data
   public static String customer_id;

   public static void setLogParams(String company_id, String userId){
      RequestDataHolder.customer_id = customer_id;
```

```
    }
}
```

You would have to synchronize all method calls to guarantee one-after-one operations that would turn your whole system effectively into a system that behaves like single-threaded but with multiple threads and massive drawbacks.

Static methods are not related to static variables when it comes to synchronization. There are however scenarios, where you can invoke synchronized non-static methods that behave differently, but that leads us too far from your original question.

edited Jul 3 '15 at 9:25

answered Jul 3 '15 at 8:49

mp911de
6,823 2 13 41

Thank you. Clearing Thread Context should be done to ensure that. (Please note that I alter the question slightly) – Supun Jul 3 '15 at 8:58

Thanks! Its explaining a lot. – Supun Jul 3 '15 at 9:08

Parameters are defined **only for the invocation of the methods**. Parameters are not shared between different invocations, even from the same thread.

Parameters works, at almost all the effects, as local variables. If you do

```
public void myMethod(int i) {
```

or

```
public void myMethod() {
  int i;
```

in both cases `i` is a variable accessible only in the scope of that method invocation. If the method is created again, *a different, independent copy* is created.

If there was not this case, the answer that you link to would make little sense (and, in fact, there would be nearly impossible to have concurrent systems; imagine that you would need to synchronize every static method, like `Integer.parseInt(String)`)

answered Jul 3 '15 at 8:44

SJuan76
21.6k 6 31 71

Thanks, and I edit the question. It will give you a clear idea now. – Supun Jul 3 '15 at 8:54