(http://baeldung.com)

An Introduction to ThreadLocal in Java

Last modified: August 27, 2017

by baeldung (http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/) +

1. Overview

In this article, we will be looking at the *ThreadLocal (https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html)* construct from the *java.lang* package. This gives us the ability to store data individually for the current thread – and simply wrap it within a special type of object.

2. *ThreadLocal* API

The *TheadLocal* construct allows us to store data that will be **accessible only** by **a specific thread**.

Let's say that we want to have an *Integer* value that will be bundled with the specific thread:

```
1 ThreadLocal<Integer> threadLocalValue = new ThreadLocal<>();
```

Next, when we want to use this value from a thread we only need to call a *get()* or *set()* method. Simply put, we can think that *ThreadLocal* stores data inside of a map – with the thread as the key.

Due to that fact, when we call a *get()* method on the *threadLocalValue* we will get an *Integer* value for the requesting thread:

```
1 threadLocalValue.set(1);
2 Integer result = threadLocalValue.get();
```

We can construct an instance of the *ThreadLocal* by using the *withInitial()* static method and passing a supplier to it:

```
1 ThreadLocal<Integer> threadLocal = ThreadLocal.withInitial(() -> 1);
```

To remove value from the *ThreadLocal* we can call a *remove()* method:

```
1 threadLocal.remove();
```

To see how to use the *ThreadLocal* properly, firstly, we will look at an example that does not use a *ThreadLocal*, then we will rewrite our example to leverage that construct.

3. Storing User Data in a Map

Let's consider a program that needs to store the user specific *Context* data per given user id:

```
public class Context {
    private String userName;

    public Context(String userName) {
        this.userName = userName;
    }
}
```

We want to have one thread per user id. We'll create a *SharedMapWithUserContext* class that implements a *Runnable* interface. The implementation in the *run()* method calls some database through the *UserRepository* class that returns a *Context* object for a given *userId*.

Next, we store that context in the *ConcurentHashMap* keyed by *userId*:

```
public class SharedMapWithUserContext implements Runnable {

    public static Map<Integer, Context> userContextPerUserId
      = new ConcurrentHashMap<>();
    private Integer userId;
    private UserRepository userRepository = new UserRepository();

    @Override
    public void run() {
        String userName = userRepository.getUserNameForUserId(userId);
        userContextPerUserId.put(userId, new Context(userName));
    }

    // standard constructor
}
```

We can easily test our code by creating and starting two threads for two different *userIds* and asserting that we have two entries in the *userContextPerUserId* map:

```
SharedMapWithUserContext firstUser = new SharedMapWithUserContext(1);
SharedMapWithUserContext secondUser = new SharedMapWithUserContext(2);
new Thread(firstUser).start();
new Thread(secondUser).start();

assertEquals(SharedMapWithUserContext.userContextPerUserId.size(), 2);
```

4. Storing User Data in *ThreadLocal*

We can rewrite our example to store the user *Context* instance using a *ThreadLocal*. Each thread will have its own *ThreadLocal* instance.

When using *ThreadLocal* we need to be very careful because every *ThreadLocal* instance is associated with a particular thread. In our example, we have a dedicated thread for each particular *userId* and this thread is created by us so we have full control over it.

The *run()* method will fetch the user context and store it into the *ThreadLocal* variable using the *set()* method:

```
public class ThreadLocalWithUserContext implements Runnable {

    private static ThreadLocal<Context> userContext
      = new ThreadLocal<>();
    private Integer userId;
    private UserRepository userRepository = new UserRepository();

    @Override
    public void run() {
        String userName = userRepository.getUserNameForUserId(userId);
        userContext.set(new Context(userName));
        System.out.println("thread context for given userId: "
          + userId + " is: " + userContext.get());
    }

    // standard constructor
}
```

We can test it by starting two threads that will execute the action for a given *userId*:

```
T
                                                    );
T
                                                    );
n
n
```

After ru ndard output that *ThreadLocal* was set per given t

```
t : Context{userNameSecret='18a78f8e-24d2-4a
t : Context{userNameSecret='e19f6a0a-253e-42
```

We can wn Context.

5. D(4? *l* with *ExecutorService*

If we want to use an *ExecutorService* and submit a *Runnable* to it, using *ThreadLocal* will yield non-deterministic results – because we do not have a guarantee that every *Runnable* action for a given *userId* will be handled by the same thread every time it is executed.

Because of that, our *ThreadLocal* will be shared among different *userIds.* That's why we should not use a *TheadLocal* together with *ExecutorService.* It should only be used when we have full control over which thread will pick which runnable action to execute.

6. Conclusion

In this quick article, we were looking at the *ThreadLocal* construct. We implemented the logic that uses *ConcurrentHashMap* that was shared between threads to store the context associated with a particular *userId.* Next, we rewrote our example to leverage *ThreadLocal* to store data that is associated with a particular *userId* and with a particular thread.

The implementation of all these examples and code snippets can be found in the GitHub project (https://github.com/eugenp/tutorials/tree/master/core-java-concurrency) – this is a Maven project, so it should be easy to import and run as it is.