Understanding the concept behind ThreadLocal

by Cristian Chiovari MVB · **May. 22, 13 · Java Zone · Not set**

Heads up...this article is old!
Technology moves quickly and this article was published 5 years ago. Some or all of its contents may be outdated.

Intro

I was aware of thread local but never had the occasion to really use it until recently. So I started digging a little bit on the subject because I needed an easy way of propagating some user information via the different layers of my web application without changing the signature of each method called.

Small prerequisite info

A thread is an individual process that has its own call stack. In Java, there is one thread per call stack or one call stack per thread. Even if you don't create any new threads in your program, threads are there running without your knowledge.Best example is whenyou just start a simple Java program via main method,then you do not implicitly call new Thread().start(), but the JVM creates a main thread for you in order to run the main method.

The main thread is quite special because it is the thread from which all the other thread will spawn and when this thread is finished, the application ends it's lifecycle.

In a web application server normally there is a pool of of threads ,because a Thread is class quite heavyweight to create.All JEE servers (Weblogic,Glassfish,JBoss etc) have a self tuning thread pool, meaning that the thread pool increase and decrease when is needed so there is not thread created on each request, and existing ones are reused.

Understanding thread local

In order to understand better thread local I will show very simplistic implementation of one custom thread local.

```
package ccs.progest.javacodesamples.threadlocal.ex1;

import java.util.HashMap;
```

```
import java.util.Map;

public class CustomThreadLocal {

    private static Map threadMap = new HashMap();

    public static void add(Object object) {
        threadMap.put(Thread.currentThread(), object);
    }

    public static void remove(Object object) {
        threadMap.remove(Thread.currentThread());
    }

    public static Object get() {
        return threadMap.get(Thread.currentThread());
    }

}
```

So you can call anytime in your application the **add** method on **CustomThreadLocal** and what it will do is to put in a **map** the current thread as **key** and as value the object you want to associate with this thread. This object might be an object that you want to have access to from anywhere within the current executed thread, or it might be an expensive object you want to keep associated with the thread and reuse as many times you want.
You define a class **ThreadContext** where you have all information you want to propagate within the thread.

```
package ccs.progest.javacodesamples.threadlocal.ex1;

public class ThreadContext {

    private String userId;

    private Long transactionId;

    public String getUserId() {
        return userId;
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }

    public Long getTransactionId() {
        return transactionId;
    }

    public void setTransactionId(Long transactionId) {
        this.transactionId = transactionId;
    }

    public String toString() {
        return "userId:" + userId + ",transactionId:" + transactionId;
    }

}
```

Now is the time to use the ThreadContext.

I will start two threads and in each thread I will add a new ThreadContext instance that will hold information I want to propagate for each thread.

```
package ccs.progest.javacodesamples.threadlocal.ex1;

public class ThreadLocalMainSampleEx1 {

    public static void main(String[] args) {
        new Thread(new Runnable() {
            public void run() {
                ThreadContext threadContext = new ThreadContext();
                threadContext.setTransactionId(1l);
                threadContext.setUserId("User 1");
                CustomThreadLocal.add(threadContext);
                //here we call a method where the thread context is not passed as parameter
                PrintThreadContextValues.printThreadContextValues();
            }
        }).start();
        new Thread(new Runnable() {
            public void run() {
                ThreadContext threadContext = new ThreadContext();
                threadContext.setTransactionId(2l);
                threadContext.setUserId("User 2");
                CustomThreadLocal.add(threadContext);
                //here we call a method where the thread context is not passed as parameter
                PrintThreadContextValues.printThreadContextValues();
            }
        }).start();
    }
}
```

Notice:
CustomThreadLocal.add(threadContext) is the line of code where the current thread is associated with the ThreadContext instance
As you will see executing this code the result will be:

```
userId:User 1,transactionId:1
userId:User 2,transactionId:2
```

How this is possible because we did not passed as parameter ThreadContext ,userId or trasactionId to printThreadContextValues ?

```
package ccs.progest.javacodesamples.threadlocal.ex1;

public class PrintThreadContextValues {
    public static void printThreadContextValues(){
        System.out.println(CustomThreadLocal.get());
    }
}
```

Simple enough 🙂
When CustomThreadLocal.get() is called from the internal map of CustomThreadLocal it is retrived the object associated with the current thread.
Now let's see the samples when is used a real ThreadLocal class. (the above CustomThreadLocal class is just to understand the principles behind ThreadLocal class which is very fast and uses memory in an optimal way)

understand the principles behind ThreadLocal class which is very fast and uses memory in an optimal way)

```
package ccs.progest.javacodesamples.threadlocal.ex2;

public class ThreadContext {

    private String userId;
    private Long transactionId;

    private static ThreadLocal threadLocal = new ThreadLocal(){
        @Override
        protected ThreadContext initialValue() {
            return new ThreadContext();
        }

    };
    public static ThreadContext get() {
        return threadLocal.get();
    }
    public String getUserId() {
        return userId;
    }
    public void setUserId(String userId) {
        this.userId = userId;
    }
    public Long getTransactionId() {
        return transactionId;
    }
    public void setTransactionId(Long transactionId) {
        this.transactionId = transactionId;
    }

    public String toString() {
        return "userId:" + userId + ",transactionId:" + transactionId;
    }
}
```

As javadoc describes : ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread

```
package ccs.progest.javacodesamples.threadlocal.ex2;

public class ThreadLocalMainSampleEx2 {

    public static void main(String[] args) {
        new Thread(new Runnable() {
            public void run() {
                ThreadContext threadContext = ThreadContext.get();
                threadContext.setTransactionId(1l);
                threadContext.setUserId("User 1");
                //here we call a method where the thread context is not passed as parameter
                PrintThreadContextValues.printThreadContextValues();
            }
        }).start();
        new Thread(new Runnable() {
            public void run() {
                ThreadContext threadContext = ThreadContext.get();
                threadContext.setTransactionId(2l);
                threadContext.setUserId("User 2");
```

```
                //here we call a method where the thread context is not passed as parameter
                PrintThreadContextValues.printThreadContextValues();
            }
        }).start();
    }
}
```

When **get** is called , a new ThreadContext instance is associated with the current thread,then the desired values are set the **ThreadContext instance.**
As you see the result is the same as for the first set of samples.

```
userId:User 1,transactionId:1
userId:User 2,transactionId:2
```

(it might be the reverse order ,so don't worry if you see 'User 2' first)

```
package ccs.progest.javacodesamples.threadlocal.ex2;

public class PrintThreadContextValues {
    public static void printThreadContextValues(){
        System.out.println(ThreadContext.get());
    }
}
```

Another very usefull usage of ThreadLocal is the situation when you have a non threadsafe instance of an quite expensive object.Most poular sample I found was with SimpleDateFormat (but soon I'll come with another example when webservices ports will be used)

```
package ccs.progest.javacodesamples.threadlocal.ex4;

import java.text.SimpleDateFormat;
import java.util.Date;

public class ThreadLocalDateFormat {
    // SimpleDateFormat is not thread-safe, so each thread will have one
    private static final ThreadLocal formatter = new ThreadLocal() {
        @Override
        protected SimpleDateFormat initialValue() {
            return new SimpleDateFormat("MM/dd/yyyy");
        }
    };
    public String formatIt(Date date) {
        return formatter.get().format(date);
    }
}
```

Conclusion:

There are many uses for thread locals. Here I describe only two: (I think the most used ones)

- Genuine per-thread context, such as user id or transaction id.
- Per-thread instances for performance.

Like This Article? Read More From DZone

Getting Access Token for Microsoft Graph Using OAuth REST API, Part 1

Zero Code REST With json-server

Prerequisites to Business Agility

Free DZone Refcard
Getting Started With Kotlin

Topics:

Software Design Principles DRY and KISS

by Arvind Singh Baghel MVB · Apr 19, 18 · Java Zone · Tutorial

In this article, I am going to explore software design principles and their benefits, why design principles are useful for us, and how to implement them in our daily programming. We will explore the DRY and KISS software design principles.

The DRY Principle: Don't Repeat Yourself

DRY stand for "Don't Repeat Yourself," a basic principle of software development aimed at reducing repetition of information. The DRY principle is stated as, "Every piece of knowledge or logic must have a single, unambiguous representation within a system."

Violations of DRY

"We enjoy typing" (or, "Wasting everyone's time."): "We enjoy typing," means writing the same code or logic again and again. It will be difficult to manage the code and if the logic changes, then we have to make changes in all the places where we have written the code, thereby wasting everyone's time.

How to Achieve DRY

To avoid violating the DRY principle, divide your system into pieces. Divide your code and logic into smaller reusable units and use that code by calling it where you want. Don't write lengthy methods, but divide logic and try to use the existing piece in your method.

DRY Benefits

Less code is good: It saves time and effort, is easy to maintain, and also reduces the chances of bugs.

One good example of the DRY principle is the helper class in enterprise libraries, in which every piece of code is unique in the libraries and helper classes.

KISS: Keep It Simple, Stupid

The KISS principle is descriptive to keep the code simple and clear, making it easy to understand. After all, programming languages are for humans to understand — computers can only understand 0 and 1 — so keep coding simple and straightforward. Keep your methods small. Each method should never be more than 40-50 lines.

Each method should only solve one small problem, not many use cases. If you have a lot of conditions in the method, break these out into smaller methods. It will not only be easier to read and maintain, but it can help find bugs a lot faster.

Violations of KISS

We have all likely experienced the situation where we get work to do in a project and found some messy code written. That leads us to ask why they have written these unnecessary lines. Just have a look at below two code snippets shown below. Both methods are doing the same thing. Now you have to decide which one to use:

```
1  public String weekday1(int day) {
2      switch (day) {
3          case 1:
4              return "Monday";
```

```
            return "Monday";
        case 2:
            return "Tuesday";
        case 3:
            return "Wednesday";
        case 4:
            return "Thursday";
        case 5:
            return "Friday";
        case 6:
            return "Saturday";
        case 7:
            return "Sunday";
        default:
            throw new InvalidOperationException("day must be in range 1 to 7");
    }
}

public String weekday2(int day) {
    if ((day < 1) || (day > 7)) throw new InvalidOperationException("day must be in range 1

    string[] days = {
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"
    };

    return days[day - 1];
}
```

How to Achieve KISS

To avoid violating the KISS principle, try to write simple code. Think of many solutions for your problem, then choose the best, simplest one and transform that into your code. Whenever you find lengthy code, divide that into multiple methods — right-click and refactor in the editor. Try to write small blocks of code that do a single task.

Benefit of KISS

If we have some functionality written by one developer and it was written with messy code, and if we ask for another developer to make modifications in that code, then first, they have to understand the code. Obviously, if the code is written simply, then there will not be any difficulty in understanding that code, and also will be easy to modify.

Summary

While writing any code or module, keep software design principles in mind and use them wisely, make them your habit so you don't need to keep remembering every time. It will save development time and make your software module robust, which could be easy to maintain and extend.

Like This Article? Read More From DZone

Software Design Principles

The Hollywood Principle

TDD Is More Than Just Test Before Code

Free DZone Refcard
Getting Started With Kotlin

Topics: JAVA, DRY, KISS, DESIGN PRINCIPLES, TUTORIAL

Published at DZone with permission of Arvind Singh Baghel , DZone MVB. See the original article here.
Opinions expressed by DZone contributors are their own.