Refactoring Java code for Multithreading Enviroment

# Heading

In my service life frequently I encounter problem that a code is giving weird result in staging or production but when it runs from developer machine it shows perfect result.

So, the clue is weird result, try to run it many times and watch result pattern if it is varying then definitely it is due to threading. You can take Thread Dump for analysis but multiple tests give you a rough idea about Thread Problem.

Staging or production are distributed environment where your resources are shared among multiple requests whereas develop machine there is only one thread involved. So it is run fine in developer machine.

It is very important to design your code for Multithreaded environment.

In this context I can recall that once I have faced a problem , a module was designed for stand-alone application. so developer implement the code without thinking of Multithreading as they know they will package the code into jar then invoke from a command line which spawn a JVM and serve only one request.

So far so good but later a new change request comes that it should be hosted in Web not as a standalone app. Web means multiple request so A New Problem was raised from nowhere. So there is no way but has to refactor the module to support Multithreading.

There is so many classes in a module some of them are pivot classes. Make it multithreaded capable is not an easy task. So we have to think a procedure by which we can do it **in less time and minimally affecting the classes.**

I want to share that experience with you people.

In that course we think and try to understand where and How Multithread environment differ from a single thread environment.

Outcomes are.

1. In Multithread environment problem occurs for shareable resource which are not synchronized.
2. If I filter the statement 1 , more specifically I can say sharable resource can create a problem If at least one thread wants to update/write it’s state . then only we can encounter dirty data.
3. If Multiple thread want to read from a resource, then we don’t need to alter them as It will not harm the state.
4. What makes a sharable resource?
5. If in a class, member variables are defined and has setter method then these classes are potential candidate for causing problem.
6. If a class has methods and all variables are method level, then they inherently thread safe. We don’t bother them.
7. If a class has Static variables they are very dangerous as static variable bound to class and one copy shared by all Objects so, If a thread change its value it reflects for all objects.
8. In a project I see every utility classes design as Singleton so we have to mark them candidate of Multithreading refactoring.
9. In a project it is often happens a step is dependent on previous stem or some derived value generated by some business logic. Developer are maintaining them as in Cache or in a Context and they are the potential candidate for Multithreading.
10. Try to find any immutable class then they are thread safe so don’t need to refactor them.

Based on the Outcome we target our module and Identifies the classes need to be refactor. Believe me

This exercise will help you to find exact classes you need to refactor and you can discover they are the High-level classes like manager or any factory classes Utilty classes etc.

**Next the important question How to refactor them?**

**To Refactor them put your thinking Cap and try to pick the tools which will cause minimal change.**

**Tool 1:**

**Check you can refactor all member variable to method variable.**

**First check in that class how many methods use this member variable, then try to find out by search your work space from where these methods are call. If number are small pass that variable as method parameter and remove the member variable.**

**Suppose in EmployeeManager class Employee is member variable and in checkValidEmploye is method where you use Employee Object**

**Code snippet**

Public class EmployeeManager

{

Employee emp;

Public void checkValidEmploye()

{

//some code using emp variable

}

}

Refactor the code

Code

Public class EmployeeManager

{

//Employee emp;// remove it

Public void checkValidEmploye(Employee emp)

{

//some code using method emp variable

}

}

Call employeeManager. checkValidEmploye(new Employee ());

**Tool2:**

**If there is so many member variables and you can’t make it local variables**

**Put the class in ThreadLocal**

**Code snippet**

EmployeeManager mgr;

**Public void validate()**

**{**

mgr**.** . checkValidEmploye();

**}**

**Refactor Code snippet**

**private** **static** ThreadLocal< EmployeeManager > *THREAD\_LOCAL\_CACHE* = **new** ThreadLocal< EmployeeManager >();;

**Public void validate()**

**{**

**If(***THREAD\_LOCAL\_CACHE.get() !=null)*

*{*

*THREAD\_LOCAL\_CACHE.get()***.** . checkValidEmploye();

*}*

*Else*

*{*

*THREAD\_LOCAL\_CACHE.set(new* EmployeeManager() *)*

*THREAD\_LOCAL\_CACHE.get()***.** . checkValidEmploye();

*}*

**}**

**By Thread Local we can ensure An EmployeeManager object exist for a thread. So EmployeeManager would be thread specific so no other thread can access one thread’s Employeemanager. So mitigate the risk of dirty data**

**Tool3> Identify the classes which are singleton basically context,cache,utility classes are single ton so make it thread safe. We can make it to Per thread Per singleton**

**Refactor single ton to normal class by a trick we can do that**

**Single to class look like**

**public class Utility**

**{**

**private static Utility instance= new Utility();**

**private Utility()**

**{**

**}**

**Public static Utility getInstance(){**

**return instance;**

**}**

**Refactor it like following to avoid change in calling classes**

**public class Utility**

**{**

**private** **static** ThreadLocal< **Utility** > *THREAD\_LOCAL\_CACHE* = **new** ThreadLocal< **Utility** >();;

**private Utility()**

**{**

**}**

**Public static Utility getInstance(){**

**If(***THREAD\_LOCAL\_CACHE.get()==null***)**

**{**

*THREAD\_LOCAL\_CACHE.set(new Utility());*

**}**

**return** *THREAD\_LOCAL\_CACHE.get();*

**}**

**This will solve the multithreading problem as each request it will gives a new Utility Object if it’s not have as There is one object per thread we can put some variable to it and use it in future .**

**Please choose the option carefully. This is your experience and design decision which tool you are going to use to alter your code minimally.**