Logging in a multithreaded environment and with CompletableFuture construct using MDC

Many of you might have used MDC for contextual logging. It works very well for the use cases where the context is added/removed/updated/modified on a single thread as the diagnostic context is managed per-thread basis. But then, the problem arises when you start logging in a multithreaded environment and using the Java constructs CompletionStage and CompletableFuture.

Logging with MDC Context

First of all, let's go through the logs of a program using MDC.

Consider a situation where a lot of client requests are coming to the server and hitting a resource controller. Following is the code

```
Logger logger = LoggerFactory.getLogger(ResourceController.class);

public Response registerUser(final String phoneNo) {
    MDC.put("phoneNo", phoneNo);
    logger.debug("Request received to register user");

    //create user in DB
    int userId = createUserInDB(phoneNo);
    MDC.put("userId", Integer.toString(userId));
    logger.debug("Created user in database");

    //generate an OTP for the user
    String otp = generateOTP(userId);
    logger.debug("Generated OTP for user");

    //send the OTP to user
    String messageId = sendOTP(userId, otp);
    MDC.put("messageId", messageId);
    logger.debug("OTP sent to user");

    return OK;
}
```

MDCLoggingInSingleThread.java hosted with ❤ by GitHub view raw

The code is pretty self-explanatory where we register a user with his/her phone no. At each step, we add relevant information to the MDC context.

Let's look at the JSON logs when this code is executed.

```
{
  "@timestamp":"2019-12-24T11:42:52.951",
  "thread":"pool-7-thread-1",
  "message":"Request received to register user",
  "mdc":{
    "phoneNo":"+919876543211"
  }
}
{
  "@timestamp":"2019-12-24T11:42:53.541",
  "thread":"pool-7-thread-1",
  "message":"Created user in database",
  "mdc":{
    "phoneNo":"+919876543211",
    "userId":"223"
  }
}
{
  "@timestamp":"2019-12-24T11:42:53.545",
  "thread":"pool-7-thread-1",
  "message":"Generated OTP for user",
  "mdc":{
    "phoneNo":"+919876543211",
    "userId":"223"
  }
}
{
  "@timestamp":"2019-12-24T11:42:54.849",
  "thread":"pool-7-thread-1",
  "message":"OTP sent to user",
  "mdc":{
    "phoneNo":"+919876543211",
    "userId":"223",
    "messageId":"2cy798jou7l44oi90l1f3b9"
  }
}
```

MDCContext.json hosted with ❤ by GitHub view raw

The Problem: Logging with MDC Context in a multithreaded environment

Let's now look at the logging in a multithreaded environment. Below is a similar program from the previous section with async construct.

```
Logger logger = LoggerFactory.getLogger(ResourceController.class);

public CompletionStage<Response> registerUser(final String phoneNo) {
    MDC.put("phoneNo", phoneNo);
    logger.debug("Request received to register user");

    return CompletableFuture.supplyAsync({
        //create user in DB
        int userId = createUserInDB(phoneNo);
        MDC.put("userId", Integer.toString(userId));
        logger.debug("Created user in database");

        return userId;
    }, dbExecutor)
    .thenApplyAsync(userId -> {
        //generate an OTP for the user
        String otp = generateOTP(userId);
        logger.debug("Generated OTP for user");

        //send the OTP to user
        String messageId = sendOTP(userId, otp);
        MDC.put("messageId", messageId);
        logger.debug("OTP sent to user");

        return OK;
    }, webServiceExecutor);
}
```

MissingMDCLoggingInAsync.java hosted with ❤ by GitHub view raw

The above code is different in following manners from the previous one

- The response of the resource is wrapped inside CompletionStage
- Creation of user now happens in *dbExecutor* executor service
- Generation of OTP and message sending happens in *webServiceExecutor* executor service

Let's look at the JSON logs

```
{
  "@timestamp":"2019-12-24T11:42:52.951",
  "thread":"pool-7-thread-1",
  "message":"Request received to register user",
  "mdc":{
    "phoneNo":"+919876543211"
  }
}
{
  "@timestamp":"2019-12-24T11:42:53.541",
  "thread":"db-executor-5-thread-4",
  "message":"Created user in database",
  "mdc":{
------> Missing    "phoneNo":"+919876543211",
    "userId":"223"
  }
}
{
  "@timestamp":"2019-12-24T11:42:53.545",
  "thread":"web-service-2-thread-3",
  "message":"Generated OTP for user",
  "mdc":{
------> Missing    "phoneNo":"+919876543211",
------> Missing    "userId":"223"
  }
}
{
  "@timestamp":"2019-12-24T11:42:54.849",
  "thread":"web-service-2-thread-3",
  "message":"OTP sent to user",
  "mdc":{
------> Missing    "phoneNo":"+919876543211",
------> Missing    "userId":"223",
    "messageId":"2cy798jou7l44oi90l1f3b9"
  }
}
```

MissingMDCContext.json hosted with ❤ by GitHub view raw

As you can see above, when thread switches happened, context goes missing. This situation is unmanageable when the chain is spread across the whole codebase.

The Solution

Make the complete ecosystem aware of MDC.

We propose the solution by solving the following scenarios:

1. Retaining MDC context while switching threads between executors. Essentially, implementing a MDC aware executor such as ThreadPoolExecutor or ForkJoinPool MDC aware.

2. Retaining MDC context with CompletableFuture construct. We would use learnings from the previous step to achieve it.

Retaining MDC context while switching thread between executors

Following is the proposed flow:

1. Thread X of Executor E1 has MDC context

2. Thread X of Executor E1 then creates a Runnable/Callable task to be submitted to another Executor E2 (E2 could be same as E1)

3. Thread X of Executor E1 *wraps* the task and submits it to Executor E2. Basically, it stores the current MDC context and sets it up just before the task is about to get executed by Executor E2

Let's write the code in 3 parts

1. Wrap up MDC context within a Callable/Runnable task

2. Make *ThreadPoolExecutor* MDC aware

3. Make *ForkJoinPool* MDC aware

Let's look at the methods to wrap MDC context within a Callable/Runnable task

```java
public static <T> Callable<T> wrapWithMdcContext(Callable<T> task) {
    //save the current MDC context
    Map<String, String> contextMap = MDC.getCopyOfContextMap();
    return () -> {
        setMDCContext(contextMap);
        try {
            return task.call();
        } finally {
            // once the task is complete, clear MDC
            MDC.clear();
        }
    };
}

public static Runnable wrapWithMdcContext(Runnable task) {
    //save the current MDC context
    Map<String, String> contextMap = MDC.getCopyOfContextMap();
    return () -> {
        setMDCContext(contextMap);
        try {
            task.run();
        } finally {
            // once the task is complete, clear MDC
            MDC.clear();
        }
    };
}

public static void setMDCContext(Map<String, String> contextMap) {
   MDC.clear();
   if (contextMap != null) {
       MDC.setContextMap(contextMap);
    }
}
```

MDCContextWrapper.java hosted with ❤ by GitHub view raw

In the above method, we wrap the Runnable to be executed where the context is copied and stored first. Further, when `Runnable#run()` is about to get executed, MDC context is set before it and cleared post its execution. That way, *the context is transferred from the previous thread to the new one*.

Next, let's look at MDC aware ThreadPoolExecutor

```java
public class MDCAwareThreadPoolExecutor extends ThreadPoolExecutor {
    //Override constructors which you need

    //Executes the given task sometime in the future.
    @Override
    public void execute(Runnable command) {
        super.execute(wrapWithMdcContext(command));
    }
}
```

MDCAwareThreadPoolExecutor.java hosted with ❤ by GitHub view raw

Building on the logic to previous methods, any task given to the ThreadPoolExecutor will transfer the previous MDC context to the new thread from its pool making it MDC aware.

On similar lines, let's look at the MDC aware ForkJoinPool

```java
public class MDCAwareForkJoinPool extends ForkJoinPool {
    //Override constructors which you need

    @Override
    public <T> ForkJoinTask<T> submit(Callable<T> task) {
        return super.submit(MDCUtility.wrapWithMdcContext(task));
    }

    @Override
    public <T> ForkJoinTask<T> submit(Runnable task, T result) {
        return super.submit(wrapWithMdcContext(task), result);
    }

    @Override
    public ForkJoinTask<?> submit(Runnable task) {
        return super.submit(wrapWithMdcContext(task));
    }

    @Override
    public void execute(Runnable task) {
        super.execute(wrapWithMdcContext(task));
    }
}
```

MDCAwareForkJoinPool.java hosted with ❤ by GitHub view raw

A quick note that I had skipped the methods with `ForkJoinTask` parameters for simplicity.

You can similarly create other MDC aware executors as per your requirement. You get the idea now!

A quick note on use cases of MDC aware executors

1. **Application Code:** Set up these executors in the application code and let it take care of the context all along.

2. **External library customization:** These come handy when a library might expose a method to set your own executor such as `library.setExecutor(mdcAwareExecutor)` which would enable context retention all the way till it reaches back to your application code.

Retaining MDC context with CompletableFuture construct

Remember that for this to work, ***all the components should be MDC aware.*** And for that, we need to address the following scenarios:

1. *When all do we get new instances of CompletableFuture from within this class?* → We need a way to return a MDC aware version of the same rather.

2. *When all do we get new instances of CompletableFuture from outside this class?* → We need a way to return a MDC aware version of the same rather.

3. *Which executor is used when in CompletableFuture class?* → In all circumstances, we need to make sure that all executors are MDC aware.

1. When all do we get new instances of CompletableFuture from within this class?

Scenario A*:* A couple of methods use an overrideable method `CompletableFuture<U> newIncompletableFuture()` to create a new instance of CompletableFuture. Below is a code excerpt from JDK

```java
public CompletableFuture<Void> thenRun(Runnable action) {
    return uniRunStage(null, action);
}

private CompletableFuture<Void> uniRunStage(Executor e, Runnable f) {
    if (f == null) throw new NullPointerException();
    Object r;
    if ((r = result) != null)
        return uniRunNow(r, e, f);
    CompletableFuture<Void> d = newIncompleteFuture();
    unipush(new UniRun<T>(e, d, this, f));
    return d;
}

public <U> CompletableFuture<U> newIncompleteFuture() {
    return new CompletableFuture<U>();
}
```

CompletableFutureThenRun.java hosted with ❤ by GitHub view raw

Solution: We simply extend the CompletableFuture class and override the method `<U> CompletableFuture<U> newIncompletableFuture()`. Now, **this method is surfaced only since JDK 9 and hence you would need this or later versions** for the solution. Below is the code snippet

```java
public class MDCAwareCompletableFuture<T> extends CompletableFuture<T> {

    @Override
    public CompletableFuture newIncompleteFuture() {
        return new MDCAwareCompletableFuture();
    }

}
```

MDCAwareCompletableFuture.java hosted with ❤ by GitHub view raw

So, we simply return the same class version of CompletableFuture that is MDC aware.

Scenario B: A couple of methods internally create a new instance of CompletableFuture itself. Below is a code excerpt from JDK

```java
public static CompletableFuture<Void> runAsync(Runnable runnable) {
    return asyncRunStage(ASYNC_POOL, runnable);
}

static CompletableFuture<Void> asyncRunStage(Executor e, Runnable f) {
    if (f == null) throw new NullPointerException();
    CompletableFuture<Void> d = new CompletableFuture<Void>();
    e.execute(new AsyncRun(d, f));
    return d;
}
```

CompletableFutureRunAsync.java hosted with ❤ by GitHub view raw

You might think why is it different from the previous implementation? The answer is simple! These methods are static and you can not override static methods.

Solution: Idea here is to convert CompletableFuture object instance with MDCAwareCompletableFuture version and return it for further chain. The idea here is to use an alternative version of such methods. Below is such an example

```java
//Alternative version of the same code with just creating a new instance
//of MDCAwareCompletableFuture and using it with completeAsync
CompletableFuture<Void> future = new MDCAwareCompletableFuture<String>()
        .completeAsync(() -> {
           //your code
        }, dbExecutorService)
        .thenCompose(storedPasswordHash -> {
            //your code
        })
        .thenAccept(isPasswordSame -> {
            //your code
        });
```

AlternativeToCompletableFutureSupplyAsync.java.java hosted with ❤ by GitHub view raw

Below is a list of methods where you would need to look for an alternative code snippet like above which creates an absolute instance of CompletableFuture.

```
1  static CompletableFuture<Void> allOf(CompletableFuture<?>… cfs)
2  static CompletableFuture<Object> anyOf(CompletableFuture<?>… cfs)
3  static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
4  static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)
5  static CompletableFuture<Void> runAsync(Runnable runnable)
6  static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)
7  static <U> CompletableFuture<U> completedFuture(U value)
8  static <U> CompletionStage<U> completedStage(U value)
9  static <U> CompletableFuture<U> failedFuture(Throwable ex)
10 static <U> CompletionStage<U> failedStage(Throwable ex)
11 CompletionStage<T> minimalCompletionStage()
```

AbsoluteInstanceMethodList.java hosted with ❤ by GitHub view raw

2. When all do we get new instances of CompletableFuture from outside this class?

Scenario: A couple of times you would run into a scenario, where you would get a version of CompletableFuture only. Consider the following scenario:

1. Code is being executed with executors being MDC aware
2. You call an external library such as Retrofit (with Java8 adapter) which returns an instance of CompletableFuture
3. Library completes the execution and the handle is now with the application code

In the above scenario, obviously, you can not control the context within the library unless it provides a way to set an executor where you can set a MDC aware executor.

Solution: Idea here is to store the copy of the current MDC before the execution, convert CompletableFuture object instance with MDCAwareCompletableFuture version and then set the MDC context with the saved one post-execution. Let's look at the code snippet below

```java
public static <T> CompletionStage<T> getMDCHandledCompletionStage(CompletableFuture<T> future, Function<T
    Map<String, String> contextMap = MDC.getCopyOfContextMap();
    return getMDCAwareCompletionStage(future)
            .handle((value, throwable) -> {
                setMDCContext(contextMap);
                if (throwable != null) {
                    return throwableFunction.apply(throwable);
                }
                return value;
            });
}

public static <T> CompletionStage<T> getMDCAwareCompletionStage(CompletableFuture<T> future) {
    return new MDCAwareCompletableFuture<>()
            .completeAsync(() -> null)
            .thenCombineAsync(future, (aVoid, value) -> value);
}
```

GetMDCHandledCompletionStage.java hosted with ❤ by GitHub view raw

A Throwable function has also been added to the parameter to handle the exception. Feel free to tweak and customize it as per your use case.

Typical usage of this might look like

```java
CompletableFuture<MyResult> getDetailsFromExternalService() {
    //Following log contains the complete MDC context
    logger.debug("Calling external service";
    CompletableFuture<MyResult> future = retrofit.callEndpoint();
    return getMDCHandledCompletionStage(future, throwable ->
        logger.error("Error getting details"));
}
```

getMDCHandledCompletionStageUsage.java hosted with ❤ by GitHub view raw

3. Which executor is used when in CompletableFuture class?

There are two types of executors that are used to execute tasks in CompletableFuture: *A default one (ForkJoinPool most of the time) and a supplied one*. Let's see which one is used under what circumstances.

Scenario A: *Default executor* is used when you use "*async" methods without passing an explicit executor* such as `CompletableFuture<U> thenApply**Async**(Function<? super T, ? extends U> fn)`, `CompletableFuture<Void> acceptEither**Async**(CompletionStage<? extends T> other, Consumer<? super T> action)`, `CompletableFuture<Void> run**Async**(Runnable runnable)` and so on...

Solution: Idea here is to make the default executor MDC aware by overriding the method `Executor defaultExecutor()`. As we already know how to create an MDC Aware ForkJoinPool, let's just use that one. Below is the code snippet

```
1  public static final ExecutorService MDC_AWARE_ASYNC_POOL = new MDCAwareForkJoinPool();
2
3  @Override
4  public Executor defaultExecutor() {
5      return MDC_AWARE_ASYNC_POOL;
6  }
```

Scenario B: *Supplied executor* is used when you use any *method with an explicit executor* such as `CompletableFuture<U> thenApplyAsync(Function<? super T, ? extends U> fn, **Executor executor**)`, `CompletableFuture<U> thenRunAsync(Runnable action, **Executor executor**)`, `CompletableFuture<Void> acceptEitherAsync(CompletionStage<? extends T> other, Consumer<? super T> action, **Executor executor**)` and so on...

Solution: Just use MDC aware executors. We had already seen MDCAwareThreadPoolExecutor and MDCAwareForkJoinPool from an earlier section.

Scenario C: *Current executor* is used when you use any "*sync" method* such as `CompletableFuture<Void> thenAccept(Consumer<? super T> action)`, `CompletableFuture<Void> thenRun(Runnable runnable)` and so on..., the current thread will continue the execution which could be one from the default or the supplied one.

Solution: Just like the previous scenario, simply use MDC aware executors before this chain.

Putting it all together

To put it all together, simply follow the theme to keep everything MDC aware. To reiterate again below are the points to remember:

- Use the brand new `MDCAwareCompletableFuture` which we developed by handling the above scenarios hosted on Github.

- A couple of methods in the class `CompletableFuture` instantiates the self version such as `new CompletableFuture...`. For such methods (a list shared previously in Scenario 1B), use an alternative method to get an instance of `MDCAwareCompletableFuture`. An example of using an alternative could be rather than using `CompletableFuture.supplyAsync(...)`, you can choose `new MDCAwareCompletableFuture<>().completeAsync(...)`

- Convert the instance of `CompletableFuture` to `MDCAwareCompletableFuture` by using the method `getMDCAwareCompletionStage` when you get stuck with one

- While supplying an executor as a parameter, make sure that it is MDC Aware such as `MDCAwareThreadPoolExecutor` or `MDCAwareForkJoinPool`

With these, if we were to go back to our original example at the start, the changes needed would be

1. Change `CompletableFuture.supplyAsync(...)` to `new MDCAwareCompletableFuture<>().completeAsync(...)`

2. Make webServiceExecutor MDC Aware

This is how it would look like

```java
Logger logger = LoggerFactory.getLogger(ResourceController.class);

public CompletionStage<Response> registerUser(final String phoneNo) {
    MDC.put("phoneNo", phoneNo);
    logger.debug("Request received to register user");

    //Previous implementation
    //return new CompletableFuture.supplyAsync({
    //New implementation
    return new MDCCompletableFuture<>().completeAsync({
        //create user in DB
        int userId = createUserInDB(phoneNo);
        MDC.put("userId", Integer.toString(userId));
        logger.debug("Created user in database");

        return userId;
    }, dbExecutor)
    .thenApplyAsync(userId -> {
        //generate an OTP for the user
        String otp = generateOTP(userId);
        logger.debug("Generated OTP for user");

        //send the OTP to user
        String messageId = sendOTP(userId, otp);
        MDC.put("messageId", messageId);
        logger.debug("OTP sent to user");

        return OK;
        //also, make the executor MDC aware
    }, webServiceExecutor);
}
```

MDCLoggingInAsync.java hosted with ❤ by GitHub view raw

Summary

We started first by looking at how MDC context is retained and works perfectly fine in a single thread. We then saw that when tasks were running in multiple stages executing in different threads, MDC context wasn't retained.

Further, we saw the solution to retain MDC context while switching threads between executors. Building upon this and analyzing how CompletableFuture works, we discussed solutions based on the scenarios and came up with an extension of it `MDCAwareCompletableFuture` along with guidelines to use it.