Preventing Recursive Future Method Calls in Salesforce

MAY 18, 2018 / PAVAN

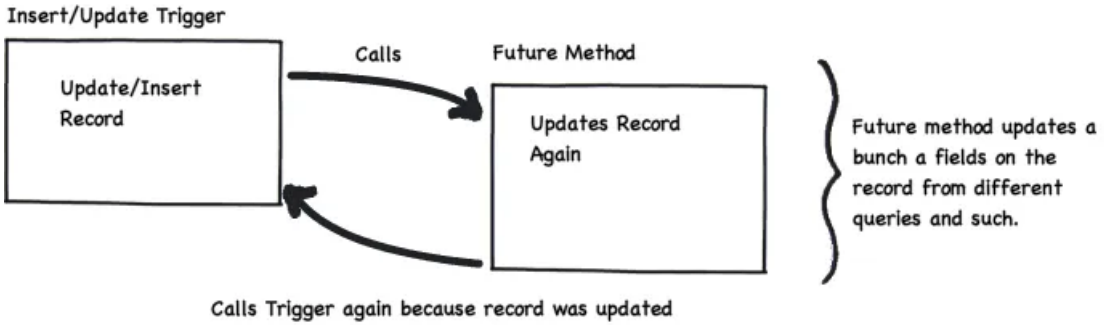
Governor limits are runtime limits enforced by the Force.com platform to ensure that your code doesn't, among other things, hog memory resources, lock up the database with an excessive amount of calls or create infinite code loops. Working within governor limits requires you to sometimes become creative when writing Apex.

One way to work within Force.com platform limits as to use asynchronous Apex methods with the future annotation. Calls to these methods execute asynchronously when the server has available resources and are subject to their own additional limits:

- No more than 10 method calls per Apex invocation
- No more than 200 method calls per Salesforce.com license per 24 hours
- The parameters specified must be primitive dataypes, arrays of primitive datatypes, or collections of primitive datatypes.
- Methods with the future annotation cannot take sObjects or objects as arguments.

- Methods with the future annotation cannot be used in Visualforce controllers in either getMethodName or setMethodName methods, nor in the constructor.
- You cannot call a method annotated with future from a method that also has the future annotation. Nor can you call a trigger from an annotated method that calls another annotated method.

One issue that you can run into when using future methods is writing a trigger with a future method that calls itself recursively. Here is a simple scenario. You have a trigger that inserts/updates a record (or a batch of them) and then makes a future method call that performs more processing on the *same record(s)*. The issue is that this entire process becomes recursive in nature and you receive the error, "System.AsyncException: Future method cannot be called from a future method…" Here is what it looks like:



There are a couple of ways to prevent this recursive behavior.

1. Add a new field to the object so the trigger can inspect the record to see if it is being called by the future method.

The trigger checks the value of IsFutureContext**c in the list of Accounts passed into the trigger. If the IsFutureContext**c value is true then the trigger is being called from the future method and the record shouldn't be processed. If the value of IsFutureContext__c is false, then the trigger is being called the first time and the future method should be called and passed the Set of unique names.

```
trigger ProcessAccount on Account (before insert, before update) {
    Set&lt;String&gt; uniqueNames = new Set&lt;String&gt;();
    for (Account a : Trigger.new) {
        if (a.IsFutureContext__c) {
            a.IsFutureContext__c = false;
        } else {
            uniqueNames.add(a.UniqueName__c);
        }
    }
    if (!uniqueNames.isEmpty())
        AccountProcessor.processAccounts(uniqueNames);
}
```

The AccountProcessor class contains the static method with the future annotation that is called by the trigger. The method processes each Account and sets the value of IsFutureContext__c to false before committing. This prevents the trigger from calling the future method once again.

```
public class AccountProcessor {

    @future
    public static void processAccounts(Set&lt;String&gt; names) {
        // list to store the accounts to update
        List&lt;Account&gt; accountsToUpdate = new List&lt;Account&gt;();
        // iterate through the list of accounts to process
        for (Account a : [Select Id, Name, IsFutureContext__c From Account where UniqueName__c IN :names]) {
            // ... do you account processing
            // set the field to true, since we are about to fire the trigger again
            a.IsFutureContext__c = true;
            // add the account to the list to update
            accountsToUpdate.add(a);
        }
        // update the accounts
        update accountsToUpdate;
    }

}
```

2. Use a static variable to store the state of the trigger processing.

According to the Apex docs, static variables are used to store information that is shared within the confines of a class. All instances of the same class share a single copy of the static variable. All triggers that are spawned by the same request can communicate with each other by referencing static variables in a related class. A recursive trigger can use the value of this class variable to determine when to exit the recursion. I've used this method many times before and was pleasantly surprised to find that this class is also shared when calling a method annotated as future.

The shared ProcessControl class with the static variable that is shared by the trigger and used to determine when to exit the process.

```
public class ProcessorControl {
    public static boolean inFutureContext = false;
}
```

In this case the trigger inspects the current value of the static variable ProcessorControl.inFutureContext to determine whether to process the records. If the value is false, then the trigger is being called the first time and the future method should be called and passed the Set of unique names. If the value is true then the trigger is being called from the future method and the records should not be processed.

```
trigger ProcessAccount on Account (before insert, before update) {
    Set&lt;String&gt; uniqueNames = new Set&lt;String&gt;();
    if (!ProcessorControl.inFutureContext) {
        for (Account a : Trigger.new)
            uniqueNames.add(a.UniqueName__c);

        if (!uniqueNames.isEmpty())
            AccountProcessor.processAccounts(uniqueNames);
    }
}
```

With this methodology, the method with the future annotation processes each Account and sets the value of the shared static variable to false before committing the records. This prevents the trigger from calling the future method once again.

```
public class AccountProcessor {

    @future
    public static void processAccounts(Set&lt;String&gt; names) {
        // list to store the accounts to update
        List&lt;Account&gt; accountsToUpdate = new List&lt;Account&gt;();
        // iterate through the list of accounts to process
        for (Account a : [Select Id, Name From Account where UniqueName__c IN :names]) {
            // ... do your account processing
            // add the account to the list to update
            accountsToUpdate.add(a);
        }
        ProcessorControl.inFutureContext = true;
        // update the accounts
        update accountsToUpdate;
    }

}
```

One of these examples should work in most cases with one caveat. With the increased usage of future method in installed packages, you may run into problems if your trigger is called from *another package's future method*. You'll again run into the "**System.AsyncException: Future method cannot be called from a future method…" error**. What Salesforce needs is an Apex function that determines whether the method is currently executing in a future call.

Please Note: In the case our Account object contains a unique string field thereby making it easy to call the same code from an insert or update. Your org will probably not have this field so you will need to make some change to pass the IDs to the future method based upon whether you are doing an insert or update.