Pipeline CPS method mismatches

Introduction

Jenkins Pipeline uses a library called Groovy CPS to run Pipeline scripts. While Pipeline uses the Groovy parser and compiler, unlike a regular Groovy environment it runs most of the program inside a special interpreter. This uses a continuation-passing style (CPS) transform to turn your code into a version that can save its current state to disk (a file called `program.dat` inside your build directory) and continue running even after Jenkins has restarted. (You can get some more technical background on the plugin page and the library page.)

While the CPS transform is usually transparent to users, there are limitations to what Groovy language constructs can be supported, and in some circumstances it can lead to counterintuitive behavior.

<table>
<thead>
<tr>
<th>JENKINS-31314</th>
<th>Running asynchronous code inside a @NonCPS method should fail cleanly</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESOLVED</td>
<td>makes the runtime try to detect the most common mistake: calling CPS-transformed code from non-CPS-transformed code. The following kinds of things are CPS-transformed:</td>
</tr>
<tr>
<td></td>
<td>• Almost all of the Pipeline script you write (including in libraries).</td>
</tr>
<tr>
<td></td>
<td>• Most Pipeline steps, including all those which take a block.</td>
</tr>
<tr>
<td></td>
<td>The following kinds of things are not CPS-transformed:</td>
</tr>
<tr>
<td></td>
<td>• Compiled Java bytecode, including</td>
</tr>
<tr>
<td></td>
<td>• the Java Platform</td>
</tr>
<tr>
<td></td>
<td>• Jenkins core and plugins</td>
</tr>
<tr>
<td></td>
<td>• the runtime for the Groovy language</td>
</tr>
<tr>
<td></td>
<td>• Constructor bodies in your Pipeline script.</td>
</tr>
<tr>
<td></td>
<td>• Any method in your Pipeline script marked with the @NonCPS annotation.</td>
</tr>
<tr>
<td></td>
<td>• A few Pipeline steps which take no block and act instantaneously, such as <code>echo</code> or <code>properties</code>.</td>
</tr>
</tbody>
</table>

CPS-transformed code may call non-CPS-transformed code or other CPS-transformed code, and non-CPS-transformed code may call other non-CPS-transformed code, but it is impossible for non-CPS-transformed code to call CPS-transformed code. If you try, there will be a mismatch between the method that the CPS interpreter thought it was calling and the method that it actually got a return value from.

Table of Contents

- 1 Introduction
  - 1.1 Table of Contents
- 2 Common problems and solutions
  - 2.1 Use of Pipeline steps from @NonCPS
  - 2.2 Calling non-CPS-transformed methods with CPS-transformed arguments
  - 2.3 Constructors
  - 2.4 Overrides of non-CPS-transformed methods
  - 2.5 Closures inside GString
- 3 False Positives
  - 3.1 Direct invocation of closures stored in object fields or maps

Common problems and solutions

Use of Pipeline steps from @NonCPS

Sometimes users will apply the @NonCPS annotation to a method definition, which bypasses the CPS transform inside that method. This can be done to work around limitations in Groovy language coverage (since the body of the method will execute using the native Groovy semantics), or to get better performance (the interpreter imposes a substantial overhead). However such methods must not call CPS-transformed code such as Pipeline steps. For example, the following will not work:
Calling non-CPS-transformed methods with CPS-transformed arguments

Some Groovy and Java methods take complex types as parameters to support dynamic behavior. A common case is sorting methods that allow callers to specify a method to use for comparing objects (\[\text{JENKINS-44924} \text{ - pipeline groovy script - Sort a list with custom comparator or closure not sorting}\]). Many methods in the Groovy standard library similar to this work correctly after the fix for \[\text{JENKINS-26481} \text{ - Mishandling of binary methods accepting Closure}\], but some methods remain unfixed. For example, the following will not work:

```groovy
def sortByLength(List<String> list) {
    list.toSorted { a, b -> Integer.valueOf(a.length()).compareTo(b.length()) }
}
def sorted = sortByLength(['333', '1', '4444', '22'])
echo(sorted.toString())
```

The closure passed to `Iterable.toSorted` is CPS-transformed, but `Iterable.toSorted` itself is not CPS-transformed internally, so this will not work as intended. The current behavior is that the return value of the method will be the return value of the first call to the closure. In the example, this results in `sorted` being set to `-1`, and the warning in the logs looks like this:

```
expected to call java.util.ArrayList.toSorted but wound up catching org.jenkinsci.plugins.workflow.cps.CpsClosure2.call
```

To fix this case, any argument passed to these methods must not be CPS-transformed. This can be accomplished by encapsulating the problematic method (`Iterable.toSorted` in the example) inside another method, and annotating the outer method with `@NonCPS`, or by creating an explicit class for the argument and annotating the relevant methods on that class with `@NonCPS`.

Constructors

Occasionally, users may attempt to use CPS-transformed code such as Pipeline steps inside of a constructor in a Pipeline script. Unfortunately, the construction of objects via the `new` operator in Groovy is not something that can be CPS-transformed (\[\text{JENKINS-26313} \text{ - Workflow script fails if CPS-transformed methods are called from constructors}\]), and so this will not work. Here is an example that calls a CPS-transformed method in a constructor:

```groovy
class Test {
    def x
    public Test() {
        setX()
    }
    private void setX() {
        this.x = 1;
    }
}
def x = new Test().x
echo "\$\{x\}"
```
The construction of `Test` will fail when the constructor calls `Test.setX` because `setX` is a CPS-transformed method. The warning in the logs from running this script looks like this:

```java
expected to call Test.<init> but wound up catching Test.setX
```

To fix this case, ensure that any methods defined in a Pipeline script that are called from inside of a constructor are annotated with `@NonCPS` and that constructors do not call any Pipeline steps.

**Overrides of non-CPS-transformed methods**

Users may create a class in a Pipeline Script that extends a preexisting class defined outside of the Pipeline script, for example from the Java or Groovy standard libraries. When doing so, the subclass must ensure that any overriding methods are annotated with `@NonCPS` and do not use any CPS-transformed code internally. Otherwise, the overriding methods will fail if called from a non-CPS context. For example, the following will not work:

```java
class Test {
    @Override
    public String toString() {
        return "Test";
    }
}
def builder = new StringBuilder()
builder.append(new Test())
echo(builder.toString())
```

Calling the CPS-transformed override of `toString` from non-CPS-transformed code such as `StringBuilder.append` is not permitted and will not work as expected in most cases. The warning in the logs from running this script looks like this:

```java
expected to call java.lang.StringBuilder.append but wound up catching Test.toString
```

To fix this case, add the `@NonCPS` annotation to the overriding method, and remove any uses of CPS-transformed code such as Pipeline steps from the method.

**Closures inside GString**

In Groovy, it is possible to use a closure in a `GString` so that the closure is evaluated every time the `GString` is used as a `String`. However, in Pipeline scripts, this will not work as expected, because the closure inside of the `GString` will be CPS-transformed. Here is an example:

```java
def x = 1
def s = "x = ${-> x}"
x = 2
echo(s)
```

Using a closure inside of a `GString` as in this example will not work. The warning from the logs when running this script looks like this:

```java
expected to call WorkflowScript.echo but wound up catching org.jenkinsci.plugins.workflow.cps.CpsClosure2.call
```

To fix this case, replace the original `GString` with a closure that returns a `GString` that uses a normal expression rather than a closure, and then call the closure where you would have used the original `GString` as follows:

```java
def x = 1
def s = { x = "$\{x\}" }
x = 2
echo(s())
```

**False Positives**

Unfortunately, some expressions may incorrectly trigger this warning even though they execute correctly. If you run into such a case, please file a new issue (after first checking for duplicates) and set the component to workflow-cps. Here are some of the known false positives and their workarounds.

**Direct invocation of closures stored in object fields or maps**
In Groovy, an expression like `object.fn()` is statically ambiguous and could be one of a few things. For example, it could be invocation a method named `fn` as in the following example:

```groovy
class Test {
    def fn() { 1 }
}
def object = new Test()
object.fn()
```

It could be syntactic sugar for calling `Closure.call` on one of the object's fields (as in `JENKINS-58407` - Fail to call a Closure stored as a class attribute/field):

```groovy
class Test {
    def fn
}
def object = new Test()
object.fn = { -> 1 }
object.fn() // Syntactic sugar for object.fn.call()
```

And it could even be syntax sugar for `Map.get` followed by `Closure.call`:

```groovy
def object = [fn: { -> 1}]
oobject.fn() // Syntactic sugar for object.get('fn').call()
```

The code that detects CPS method mismatches cannot currently distinguish between these cases, and so the second two examples issue a warning even though they are correct (the first example works correctly with no warning). As a workaround, when you are invoking a closure, you can explicitly call `Closure.call` rather than using Groovy syntactic sugar. Here are examples of the workaround applied to the previous two examples:

```groovy
class Test {
    def fn
}
def object = new Test()
object.fn = { -> 1 }
object.fn.call()
```

```groovy
def object = [fn: { -> 1}]
oobject.fn.call()
```