HowToUseInjectionFramework

How to use injection framework

This page will be getting more details about Hadoop development and testing using AspectJ based Injection Framework.

Introduction

The idea of code injection is fairly simple: it is an infusion of new or modification of existing behavior into a code of a software application. Fault Injection, which will be discussed later, is a similar mechanism for adding errors and exceptions into an application's logic to achieve a higher coverage and fault tolerance of the system. Different implementations of this idea are available today. Hadoop's inject framework is built on top of Aspect Oriented Paradigm (AOP) implemented on top of AspectJ toolkit.

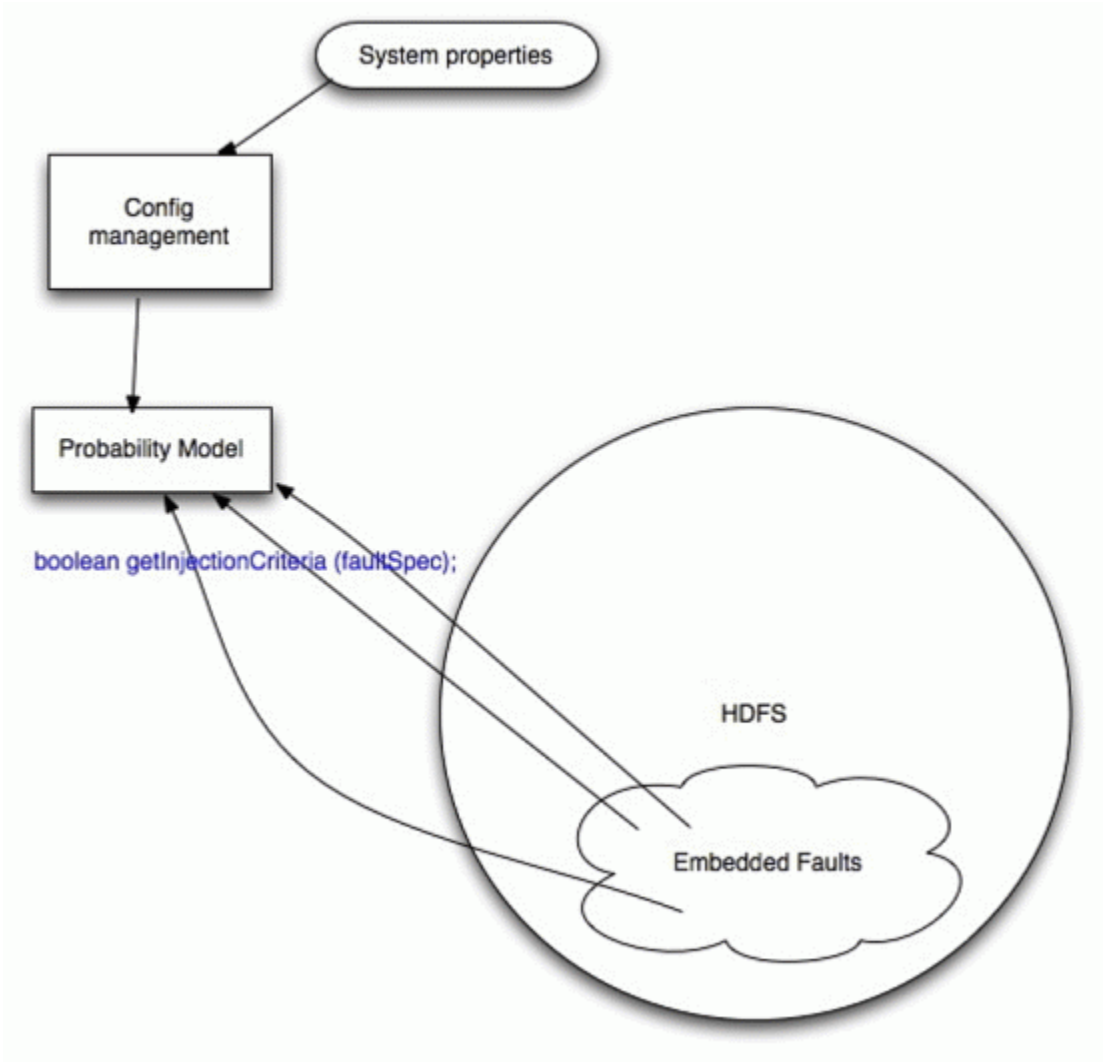
The document below relates to injection technique with regards to code/behavior modifications and to injection of programmatic faults (fault-injection). I'll refer to both injection and fault injection aspects of the framework unless narrower cases are explicitly mentioned.

Assumptions

For the sake of production code clearness and to avoid undesirable effects an instrumented code is kept separate from production code. A set of extra build target is used to produced injection related artifacts. They are easily distinguishable by '-fi' suffix as in 'Framework Injection' (not to confuse with 'if' keyword of many programming languages).

The current implementation of the FI framework assumes that the faults it will be emulating are of non-deterministic nature. That is, the moment of a fault's happening isn't known in advance and is a coin-flip based.

Architecture of the Injection Framework



Configuration Management

Currently only configuration for injected faults is available. Configuration management allows you to set expectations for faults to happen. The settings can be applied either statically (in advance) or in runtime. The desired level of faults in the framework can be configured two ways:

- editing `src/aop/fi-site.xml` configuration file. This file is similar to other Hadoop's config files
- setting system properties of JVM through VM startup parameters or in `build.properties` file

Probability Model

This is essentially a coin flipper to regulate faults occurrence. The methods of this class are getting a random number between `0.0` and `1.0` and then checking if a new number has happened to be in the range of `0.0` and a configured level for the fault in question. If that condition is true then the fault will occur.

Thus, to guarantee the happening of a fault one needs to set an appropriate level to `1.0`. To completely prevent a fault from happening its probability level has to be set to `0.0`. The default probability level is set to `0` unless the level is changed explicitly through the configuration file or in the runtime. The name of the default level's configuration parameter is `fi.*`

Injection mechanism: AOP and AspectJ

The foundation of Hadoop's FI includes a cross-cutting concept implemented by AspectJ. The following basic terms are important to remember:

- A cross-cutting concept (aspect) is behavior, and often data, that is used across the scope of a piece of software
- In AOP, the **aspects** provide a mechanism by which a cross-cutting concern can be specified in a modular way
- **Advice** is the code that is executed when an aspect is invoked
- **Join point** (or pointcut) is a specific point within the application that may or not invoke some advice

Predefined Join Points

The following readily available join points are provided by AspectJ:

- when a method is called
- during a method's execution
- when a constructor is invoked
- during a constructor's execution
- during aspect advice execution
- before an object is initialized
- during object initialization
- during static initializer execution
- when a class's field is referenced
- when a class's field is assigned
- when a handler is executed

Aspect Example

This is fault injection example:

```
package org.apache.hadoop.hdfs.server.datanode;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.fi.ProbabilityModel;
import org.apache.hadoop.hdfs.server.datanode.DataNode;
import org.apache.hadoop.util.DiskChecker.*;

import java.io.IOException;
import java.io.OutputStream;
import java.io.DataOutputStream;

/**
* This aspect takes care about faults injected into datanode.BlockReceiver
* class
*/
public aspect BlockReceiverAspects {
  public static final Log LOG = LogFactory.getLog(BlockReceiverAspects.class);

  public static final String BLOCK_RECEIVER_FAULT="hdfs.datanode.BlockReceiver";
    pointcut callReceivePacket() : call (* OutputStream.write(..))
      withincode (* BlockReceiver.receivePacket(..))
    // to further limit the application of this aspect a very narrow 'target' can be used as follows
    // target(DataOutputStream)
      !within(BlockReceiverAspects +);

  before () throws IOException : callReceivePacket () {
    if (ProbabilityModel.injectCriteria(BLOCK_RECEIVER_FAULT)) {
      LOG.info("Before the injection point");
      Thread.dumpStack();
      throw new DiskOutOfSpaceException ("FI: injected fault point at " +
      thisJoinPoint.getStaticPart( ).getSourceLocation());
    }
  }
}
```

The aspect has two main parts:

- The join point `pointcut callReceivepacket()` which servers as an identification mark of a specific point (in control and/or data flow) in the life of an application.
- A call to the advice - `before () throws IOException : callReceivepacket()` - will be injected (see *Putting It All Together* below) before that specific spot of the application's code.

The pointcut identifies an invocation of class' `java.io.OutputStream write()` method with any number of parameters and any return type. This invoke should take place within the body of method `receivepacket()` from class `BlockReceiver`. The method can have any parameters and any return type. Possible invocations of `write()` method happening anywhere within the aspect `BlockReceiverAspects` or its heirs will be ignored.

Note 1: This short example doesn't illustrate the fact that you can have more than a single injection point per class. In such a case the names of the faults have to be different if a developer wants to trigger them separately.

Note 2: After the injection step (see *Putting It All Together* below) you can verify that the faults were properly injected by searching for `ajc` keywords in a disassembled class file.

Here's code injection example

```
package org.apache.hadoop.security;

import java.io.ByteArrayInputStream;
import java.io.DataInputStream;
import java.io.IOException;

import org.apache.hadoop.io.WritableUtils;

privileged aspect AccessTokenHandlerAspects {
  /** check if a token is expired. for unit test only.
   *  return true when token is expired, false otherwise */
  static boolean AccessTokenHandler.isTokenExpired(AccessToken token) throws IOException {
    ByteArrayInputStream buf = new ByteArrayInputStream(token.getTokenID()
        .getBytes());
    DataInputStream in = new DataInputStream(buf);
    long expiryDate = WritableUtils.readVLong(in);
    return isExpired(expiryDate);
  }

  /** set token lifetime. for unit test only */
  synchronized void AccessTokenHandler.setTokenLifetime(long tokenLifetime) {
    this.tokenLifetime = tokenLifetime;
  }
}
```

The great thing about this is the fact that injected methods needed for a testing will exist in an instrumented build only and will never pollute the production code.

Fault Naming Convention and Namespaces

For the sake of a unified naming convention the following two types of names are recommended for a new aspects development:

- Activity specific notation (when we don't care about a particular location of a fault's happening). In this case the name of the fault is rather abstract `fi.hdfs.DiskError`
- Location specific notation. Here, the fault's name is mnemonic as in `fi.hdfs.datanode.BlockReceiver[optional location details]`

Development Tools

- The Eclipse AspectJ Development Toolkit may help you when developing aspects
- IntelliJ IDEA provides AspectJ weaver and Spring-AOP plugins

Putting It All Together

Aspects (faults) have to injected or woven into the code before they can be used. Follow these instructions:

- To weave aspects in place use:

```
% ant injectfaults
```

- If you misidentified the join point of your aspect you will see a warning (similar to the one shown here) when 'injectfaults' target is completed:

```
[iajc] warning at
src/test/aop/org/apache/hadoop/hdfs/server/datanode/ \
          BlockReceiverAspects.aj:44::0
advice defined in org.apache.hadoop.hdfs.server.datanode.BlockReceiverAspects
has not been applied [Xlint:adviceDidNotMatch]
```

It isn't an error from AspectJ point of view, however Hadoop's build will fail to preserve the integrity of the source code.

- To prepare dev.jar file with all your faults weaved in place use:

```
% ant jar-fault-inject
```

- To create test jars use:

```
% ant jar-test-fault-inject
```

- To run HDFS tests with faults injected use:

```
% ant run-test-hdfs-fault-inject
```

How to Use the Fault Injection Framework

Faults can be triggered as follows:

- During runtime:

```
% ant run-test-hdfs -Dfi.hdfs.datanode.BlockReceiver=0.12
```

To set a certain level, for example 25%, of all injected faults use:

```
% ant run-test-hdfs-fault-inject -Dfi.*=0.25
```

- From a program:

```
package org.apache.hadoop.fs;

import org.junit.Test;
import org.junit.Before;
import org.junit.After;

public class DemoFiTest {
  public static final String BLOCK_RECEIVER_FAULT="hdfs.datanode.BlockReceiver";
  @Override
  @Before
  public void setUp(){
    //Setting up the test's environment as required
  }

  @Test
  public void testFI() {
    // It triggers the fault, assuming that there's one called 'hdfs.datanode.BlockReceiver'
    System.setProperty("fi." + BLOCK_RECEIVER_FAULT, "0.12");
    //
    // The main logic of your tests goes here
    //
    // Now set the level back to 0 (zero) to prevent this fault from happening again
    System.setProperty("fi." + BLOCK_RECEIVER_FAULT, "0.0");
    // or delete its trigger completely
    System.getProperties().remove("fi." + BLOCK_RECEIVER_FAULT);
  }

  @Override
  @After
  public void tearDown() {
    //Cleaning up test test environment
  }
}
```

As you can see above these two methods do the same thing. They are setting the probability level of `hdfs.datanode.BlockReceiver` at 12%. The difference, however, is that the program provides more flexibility and allows you to turn a fault off when a test no longer needs it.

It should be clear that random faults aren't the only possible usage scenario of fault injection. Faults might be initialized by other means as well, i.e. setting certain static variables; instantiation of some objects, etc.

Couple of hints

Eclipse provides very convenient environment for AspectJ development. However, as everything with Eclipse, it has some issues. Here's three simple steps how to make your project AspectJ aware:

- Using project property convert it to AspectJ project
- Re-insert 'Ant builder' as explained in EclipseEnvironment
- Close and re-open open the project

After that you suppose to have type completion, syntax highlighting, and cross-references working for both .java and .aj files.

Additional Information and Contacts

These two sources of information are particularly interesting and worth reading:

- http://www.eclipse.org/aspectj/doc/next/devguide/
- AspectJ Cookbook (ISBN-13: 978-0-596-00654-9)

If you have additional comments or questions for the author check HDFS-435.

Recent presentation (2/11/2011) about code and fault injection could be found here. Here are the slides