**翻译网址：**[**http://docs.spring.io/spring-batch/reference/html/testing.html**](http://docs.spring.io/spring-batch/reference/html/testing.html)

**10.1 创建一个单元测试**

In order for the unit test to run a batch job, the framework must load the job's ApplicationContext. Two annotations are used to trigger this:

* @RunWith(SpringJUnit4ClassRunner.class): Indicates that the class should use Spring's JUnit facilities
* @ContextConfiguration(locations = {...}): Indicates which XML files contain the ApplicationContext.
* @RunWith(SpringJUnit4ClassRunner.class)
* @ContextConfiguration(locations = { "/simple-job-launcher-context.xml",
* "/jobs/skipSampleJob.xml" })
* public class SkipSampleFunctionalTests { ... }

为了让单元测试跑一个批处理的任务，这个框架必须加载这个任务的ApplicationContext,两个注解的触发方式：

* @RunWith(SpringJUnit4ClassRunner.class):表明这个类需要使用Spring的测试工具
* @ContextConfiguration(locations = {...}):表明哪些xml文件包含ApplicationContext
* @RunWith(SpringJUnit4ClassRunner.class)
* @ContextConfiguration(locations = { "/simple-job-launcher-context.xml",
* "/jobs/skipSampleJob.xml" })

public class SkipSampleFunctionalTests { ... }

# 10.2 点对点的批处理任务测试

'End To End' testing can be defined as testing the complete run of a batch job from beginning to end. This allows for a test that sets up a test condition, executes the job, and verifies the end result.

点对点的测试被定义为测试从开始到结束完整的运行一个批处理任务。它允许一个测试设置一个测试条件，执行任务，并且验证最后的结果。

In the example below, the batch job reads from the database and writes to a flat file. The test method begins by setting up the database with test data. It clears the CUSTOMER table and then inserts 10 new records. The test then launches the Job using the launchJob() method. The launchJob() method is provided by the JobLauncherTestUtils class. Also provided by the utils class is launchJob(JobParameters), which allows the test to give particular parameters. The launchJob() method returns the JobExecution object which is useful for asserting particular information about the Job run. In the case below, the test verifies that the Job ended with status "COMPLETED".

在下面的这个例子中，这个批处理任务从数据库读取数据并且写入一个平面文件中。这个测试方法首先要建立数据库与测试数据。它清空CUSTOMER表然后往里面插入10条新的记录。测试然后启动任务使用launchJob()方法，这个launchJob()方法是由JobLauncherTestUtils类提供的。还提供工具类launchJob(JobParameters),允许测试给特定的参数。launchJob()方法返回JobExecution对象有助于任务运行声明特定的信息，在这个实例中，这个测试证实任务的结束与状态"COMPLETED".

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(locations = { "/simple-job-launcher-context.xml",

"/jobs/skipSampleJob.xml" })

public class SkipSampleFunctionalTests {

@Autowired

private JobLauncherTestUtils jobLauncherTestUtils;

private SimpleJdbcTemplate simpleJdbcTemplate;

@Autowired

public void setDataSource(DataSource dataSource) {

this.simpleJdbcTemplate = new SimpleJdbcTemplate(dataSource);

}

@Test

public void testJob() throws Exception {

simpleJdbcTemplate.update("delete from CUSTOMER");

for (int i = 1; i <= 10; i++) {

simpleJdbcTemplate.update("insert into CUSTOMER values (?, 0, ?, 100000)",

i, "customer" + i);

}

JobExecution jobExecution = jobLauncherTestUtils.launchJob().getStatus();

Assert.assertEquals("COMPLETED", jobExecution.getExitStatus());

}

}

# 10.3 测试各个步骤

For complex batch jobs, test cases in the end-to-end testing approach may become unmanageable. It these cases, it may be more useful to have test cases to test individual steps on their own. The AbstractJobTests class contains a method launchStep that takes a step name and runs just that particular Step. This approach allows for more targeted tests by allowing the test to set up data for just that step and to validate its results directly.

对于复杂的批处理任务，测试用例在点对点的测试方法中可能变成难于管理的。这些情况下，凭你们自己的力量用测试用例测试各个步骤更加有用。AbstractJobTests类包含一个方法launchStep需要一个步骤名称和运行特定的步骤。这个方法允许更有针对性的测试通过这个步骤允许测试设置数据并验证其结果。

JobExecution jobExecution = jobLauncherTestUtils.launchStep("loadFileStep");

# 10.4 测试Step-Scoped组件

Often the components that are configured for your steps at runtime use step scope and late binding to inject context from the step or job execution. These are tricky to test as standalone components unless you have a way to set the context as if they were in a step execution. That is the goal of two components in Spring Batch: the StepScopeTestExecutionListener and the StepScopeTestUtils.

时常组件在运行的时候需要配置你的步骤使用步骤并且迟绑定注入上下文从步骤或者是任务执行。这些是机警的测试像单独的组件除非你有一个办法设置上下文就像他们在一个步骤里执行。那是两个组件的目标在spring batch中：StepScopeTestExecutionListener 和 StepScopeTestUtils

The listener is declared at the class level, and its job is to create a step execution context for each test method. For example:

这个监听是公开的在类级别中,它的工作是创建一个步骤为每个测试方法执行上下文。例如：

@ContextConfiguration

@TestExecutionListeners( { DependencyInjectionTestExecutionListener.class,

StepScopeTestExecutionListener.class })

@RunWith(SpringJUnit4ClassRunner.class)

public class StepScopeTestExecutionListenerIntegrationTests {

// This component is defined step-scoped, so it cannot be injected unless

// a step is active...

@Autowired

private ItemReader<String> reader;

public StepExecution getStepExection() {

StepExecution execution = MetaDataInstanceFactory.createStepExecution();

execution.getExecutionContext().putString("input.data", "foo,bar,spam");

return execution;

}

@Test

public void testReader() {

// The reader is initialized and bound to the input data

assertNotNull(reader.read());

}

}

There are two TestExecutionListeners, one from the regular Spring Test framework and handles dependency injection from the configured application context, injecting the reader, and the other is the Spring Batch StepScopeTestExecutionListener. It works by looking for a factory method in the test case for a StepExecution, and using that as the context for the test method, as if that execution was active in a Step at runtime. The factory method is detected by its signature (it just has to return a StepExecution). If a factory method is not provided then a default StepExecution is created.

有两个TestExecutionListeners，一个来自普通的Spring测试框架和处理配置应用上下文的依赖注入，注入reader和其他的Spring batch StepScopeTestExecutionListener.它以一个stepExecution在测试用例中寻找一个工厂方法为动力,并且使用它如同测试方法的上下文。在运行时在一个步骤内如果它运行是活动的。通过工厂方法的签名检测到工厂方法（它仅仅有返回一个StepExecution）。如果一个工厂方法不被提供，会有一个默认的StepExecution被创建

The listener approach is convenient if you want the duration of the step scope to be the execution of the test method. For a more flexible, but more invasive approach you can use the StepScopeTestUtils. For example, to count the number of items available in the reader above:

这个监听方法是方便的，如果你想在测试方法中执行持续的step scope。为了更灵活，但是更侵入性的方法你能使用StepScopeTestUtils。例如，去计算可用产品的数量在reader上面：

int count = StepScopeTestUtils.doInStepScope(stepExecution,

new Callable<Integer>() {

public Integer call() throws Exception {

int count = 0;

while (reader.read() != null) {

count++;

}

return count;

}

});

# 10.5 验证输出文件

When a batch job writes to the database, it is easy to query the database to verify that the output is as expected. However, if the batch job writes to a file, it is equally important that the output be verified. Spring Batch provides a class AssertFile to facilitate the verification of output files. The method assertFileEquals takes two File objects (or two Resource objects) and asserts, line by line, that the two files have the same content. Therefore, it is possible to create a file with the expected output and to compare it to the actual result:

当一个批处理任务写入数据库的时候，查询数据库像预期的一样去验证输出是简单的。然而，如果批处理任务写入一个文件，它同样引用的这个输出需要被验证。Spring Batch 提供了一个类AssertFile使输出文件的验证变得容易。assertFileEquals方法带了两个文件对象（或者是两个资源对象）和断言，一行一行的，两个文件有相同的上下文。因此，它可能创建了一个文件，有预期的输出和对比之后返回的真实结果：

private static final String EXPECTED\_FILE = "src/main/resources/data/input.txt";

private static final String OUTPUT\_FILE = "target/test-outputs/output.txt";

AssertFile.assertFileEquals(new FileSystemResource(EXPECTED\_FILE),

new FileSystemResource(OUTPUT\_FILE));

# 10.6 模拟域对象

Another common issue encountered while writing unit and integration tests for Spring Batch components is how to mock domain objects. A good example is a StepExecutionListener, as illustrated below:

遇到了另一个常见的问题,同时为Spring Batch编写单元测试和集成测试组件是如何模拟域对象。一个很好的例子是StepExecutionListener,如下所示:

public class NoWorkFoundStepExecutionListener extends StepExecutionListenerSupport {

public ExitStatus afterStep(StepExecution stepExecution) {

if (stepExecution.getReadCount() == 0) {

throw new NoWorkFoundException("Step has not processed any items");

}

return stepExecution.getExitStatus();

}

}

The above listener is provided by the framework and checks a StepExecution for an empty read count, thus signifying that no work was done. While this example is fairly simple, it serves to illustrate the types of problems that may be encountered when attempting to unit test classes that implement interfaces requiring Spring Batch domain objects. Consider the above listener's unit test:

上面的监听是框架提供的，并且它检测到stepExecution的read count是为空的，因此它表示没有工作要做。虽然这个例子相当简单，当试图单元测试类的时候它解释了可能遇到的问题类型,实现接口验证Spring Batch 域对象。考虑到上面的侦听器的单元测试:

private NoWorkFoundStepExecutionListener tested = new NoWorkFoundStepExecutionListener();

@Test

public void testAfterStep() {

StepExecution stepExecution = new StepExecution("NoProcessingStep",

new JobExecution(new JobInstance(1L, new JobParameters(),

"NoProcessingJob")));

stepExecution.setReadCount(0);

try {

tested.afterStep(stepExecution);

fail();

} catch (NoWorkFoundException e) {

assertEquals("Step has not processed any items", e.getMessage());

}

}

Because the Spring Batch domain model follows good object orientated principles, the StepExecution requires a JobExecution, which requires a JobInstance and JobParameters in order to create a valid StepExecution. While this is good in a solid domain model, it does make creating stub objects for unit testing verbose. To address this issue, the Spring Batch test module includes a factory for creating domain objects: MetaDataInstanceFactory. Given this factory, the unit test can be updated to be more concise:

因为Spring Batch域模型遵循良好的面向对象原则，StepExecution需要一个JobExecution，JobExecution需要一个JobInstance和JobParameters为了创建一个有效的StepExecution。虽然这是很好的可靠的域模型,为了冗长的单元测试创建存根对象。为了解决这个问题，Spring Batch测试模块引入一个工厂来创建域对象:MetaDataInstanceFactory，考虑到这个工厂，单元测试可以被更新的更简洁：

private NoWorkFoundStepExecutionListener tested = new NoWorkFoundStepExecutionListener();

@Test

public void testAfterStep() {

StepExecution stepExecution = MetaDataInstanceFactory.createStepExecution();

stepExecution.setReadCount(0);

try {

tested.afterStep(stepExecution);

fail();

} catch (NoWorkFoundException e) {

assertEquals("Step has not processed any items", e.getMessage());

}

}

The above method for creating a simple StepExecution is just one convenience method available within the factory. A full method listing can be found in its Javadoc.

上面的方法为了创建一个简单的StepExecution，它仅仅是便利的方法中可用的工厂。一个完整的方法清单中可以找到它在Javadoc里面