InfoQ Maven专栏（五）——自动化Web应用集成测试

March 13th, 2011

自动化集成测试的角色

本专栏的上一篇文章讲述了Maven与持续集成的一些关系及具体实践，我们都知道，自动化测试是持续集成必不可少的一部分，基本上，没有自动化测试 的持续集成，都很难称之为真正的持续集成。我们希望持续集成能够尽早的暴露问题，但这远非配置一个 Hudson/Jenkins服务器那么简单，只有真正用心编写了较为完整的测试用例，并一直维护它们，持续集成才能孜孜不倦地运行测试并第一时间报告问 题。

自动化测试这个话题很大，本文不想争论测试先行还是后行，这里强调的是测试的自动化，并基于具体的技术（Maven、 JUnit、Jetty等）来介绍一种切实可行的自动化Web应用集成测试方案。当然，自动化测试还包括单元测试、验收测试、性能测试等，在不同的场景 下，它们都能为软件开发带来极大的价值。本文仅限于讨论集成测试，主要是因为笔者觉得这是一个非常重要却常常被忽略的实践。

基于Maven的一般流程

集成测试与单元测试最大的区别是它需要尽可能的测试整个功能及相关环境，对于测试Web应用而言，通常有这么几步：

1. 启动Web容器
2. 部署待测试Web应用
3. 以Web客户端的角色运行测试用例
4. 停止Web容器

启动Web容器可以有很多方式，例如你可以通过Web容器提供的API采用编程的方式来启动容器，但在Maven的环境下，配置插件显得更简单。如果你了解Maven的生命周期模型， 就可能会想到，我们可以在pre-integration-test阶段启动容器，部署待测试应用，然后在integration-test阶段运行集成 测试用例，最后在post-integrate-test阶段停止容器。也就是说，对于步骤1，2和4我们只须进行一些简单的配置，不必编写额外的代码。 第3步是以黑盒的形式模拟客户端进行测试，需要注意的是，这里通常要求你理解一些基本的HTTP协议知识，例如服务端在什么情况下应该返回HTTP代码 200，什么时候应该返回401错误，以及所支持的Content-Type是什么等等。

至于测试用例该怎么写，除了需要用到一些用来访问Web以及解析响应详细的基础设施工具类之外，其他内容与单元测试大同小异，基本就是准备测试数据、访问服务、验证返回值等等。

一个简单的例子

谈了不少理论，现在该给个具体的例子了，譬如现在有个简单的Servlet，它接受参数a和b，做加法后返回二者之和，如果参数不完整，则返回HTTP 400错误，表示客户端的请求有问题。

```
1  public class AddServlet
2      extends HttpServlet
3  {
4      @Override
5      protected void doGet( HttpServletRequest req, HttpServletResponse resp )
6          throws ServletException,
7              IOException
8      {
9          String a = req.getParameter( "a" );
10         String b = req.getParameter( "b" );
11
12         if ( a == null || b == null )
13         {
14             resp.setStatus( 400 );
15             return;
16         }
17
18         int result = Integer.parseInt( a ) + Integer.parseInt( b );
19
20         resp.setStatus( 200 );
```

```
        resp.getWriter().print( result );
    }
}
```

为了测试这段代码，我们需要一个Web容器，这里暂且使用Jetty，因为目前来说它与Maven集成的相对最好。Jetty提供了一个Jetty Maven Plugin，借助该插件，我们可以随时启动Jetty并部署Maven默认目录布局的Web项目，实现快速开发和测试。这里我们需要的是在pre-integration-test阶段启动Jetty，在post-integrate-test阶段停止容器，对应的POM配置如下：

```
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>jetty-maven-plugin</artifactId>
        <version>7.3.0.v20110203</version>
        <configuration>
          <stopPort>9966</stopPort>
          <stopKey>stop-jetty-for-it</stopKey>
        </configuration>
        <executions>
          <execution>
            <id>start-jetty</id>
            <phase>pre-integration-test</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <daemon>true</daemon>
            </configuration>
          </execution>
          <execution>
            <id>stop-jetty</id>
            <phase>post-integration-test</phase>
            <goals>
              <goal>stop</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

XML代码中第一处configuration是插件的全局配置，stopPort和 stopKey是该插件用来停止Jetty需要用到的TCP端口及消息关键字。接着是两个executation元素，第一个executation将 jetty-maven-plugin的run目标绑定至Maven的pre-integration-test生命周期阶段，表示启动容器，第二个 executation将stop目标绑定至post-integration-test生命周期阶段，表示停止容器。需要注意的是，启动Jetty时我 们需要配置deamon为true，让Jetty在后台运行以免阻塞mvn命令。此外，jetty-maven-plugin的run目标也会自动部署当 前Web项目。

准备好Web容器环境之后，我们接着看一下测试用例代码：

```
public class AddServletIT
{
    @Test
    public void addWithParametersAndSucceed()
        throws Exception
    {
        HttpClient httpclient = new DefaultHttpClient();
        HttpGet httpGet = new HttpGet( "http://localhost:8080/add?a=1&b=2" );
        HttpResponse response = httpclient.execute( httpGet );

        Assert.assertEquals( 200, response.getStatusLine().getStatusCode() );
        Assert.assertEquals( "3", EntityUtils.toString( response.getEntity() ) );
    }

    @Test
    public void addWithoutParameterAndFail()
        throws Exception
    {
        HttpClient httpclient = new DefaultHttpClient();
```

```
        HttpGet httpGet = new HttpGet( "http://localhost:8080/add" );
        HttpResponse response = httpclient.execute( httpGet );

        Assert.assertEquals( 400, response.getStatusLine().getStatusCode() );
    }
}
```

为了能够访问应用，这里用到了HttpClient， 两个测试方法都初始化一个HttpClient，然后创建HttpGet对象用来访问Web地址。第一个测试方法顾名思义用来测试成功的场景，它提供参数 a=1和b=2，执行请求后，验证返回结果成功（HTTP状态码200）并且内容为正确的值3。第二个测试方法则用来测试失败的场景，当不提供参数的时 候，服务器应该返回一个HTTP 400错误。该测试类其实是相当粗糙的，例如有硬编码的服务器URL，这里的目的仅仅是通过尽可能简单的代码来展现一个自动化集成测试的实现过程。

上述代码中，测试类的名称为AddServletIT，而不是一般的**Test，IT表示IntegrationTest，这么命名是为了和单元测试区分开来，这样，鉴于Maven默认的测试命名约定，Maven在test生命周期阶段执行单元测试时，就不会涉及集成测试。现在，我们希望Maven在integration-test阶段执行所有以IT结尾命名的测试类，配置Maven Surefire Plugin如下：

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.7.2</version>
        <executions>
          <execution>
            <id>run-integration-test</id>
            <phase>integration-test</phase>
            <goals>
              <goal>test</goal>
            </goals>
            <configuration>
              <includes>
                <include>**/*IT.java</include>
              </includes>
            </configuration>
          </execution>
        </executions>
      </plugin>
```

通过命名规则和插件配置，我们优雅地分离了单元测试和集成测试，而且我们知道在integration-test阶段，Jetty容器已经启动完成 了。如果你在使用TestNG，那你还可以使用其测试组的特性来分离单元测试和集成测试，Maven Surefire Plugin对其也有着很好的支持。

一切就绪了，运行 **mvn clean install** 以自动运行集成测试，我们可以看到如下的输出片段：

```
[INFO] --- jetty-maven-plugin:7.3.0.v20110203:run (start-jetty) @ webapp-demo ---
[INFO] Configuring Jetty for project: webapp-demo
[INFO] webAppSourceDirectory /home/juven/git_juven/webapp-demo/src/main/webapp does not exist
[INFO] Reload Mechanic: automatic
[INFO] Classes = /home/juven/git_juven/webapp-demo/target/classes
[INFO] Context path = /
...
2011-03-06 14:55:15.676:INFO::Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
[INFO]
[INFO] --- maven-surefire-plugin:2.7.2:test (run-integration-test) @ webapp-demo ---
[INFO] Surefire report directory: /home/juven/git_juven/webapp-demo/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.juvenxu.webapp.demo.AddServletIT
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.344 sec

Results :

```

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- jetty-maven-plugin:7.3.0.v20110203:stop (stop-jetty) @ webapp-demo ---
```

可以看到jetty-maven-plugin:7.3.0.v20110203:run对应了start-jetty，maven- surefire-plugin:2.7.2:test对应了run-integration-test，jetty-maven- plugin:7.3.0.v20110203:stop对应了stop-jetty，与我们的配置和期望完全一致。此外两个测试也都成功了！

小结

相对于单元测试来说，集成测试更难编写，因为需要准备更多的环境，本文只涉及了Web容器最简单的情形，实际的开发情形中，你可能会遇到数据库，第 三方Web服务，更复杂的容器配置和数据格式等等，这都使得编写集成测试变得让人畏惧。然而反过来考虑，无论如何你都需要测试，虽然这个自动化过程的投入 很大，但收益往往更加可观，这不仅仅是手动测试时间的节省，更重要的是，你无法保证手动测试能被高频率的反复执行，也就无法保证问题能被尽早暴露。

对于Web应用来说，编写集成测试有助于你考虑和设计Web应用对外暴露的接口，这种“开发实现”/“测试审察”之间的角色转换往往能造就更清晰的设计，这也是编写测试最大的好处之一。

Maven用户能够得益于Maven的插件系统，不仅能节省大量的编码，还能得到稳定的工具，Jetty Maven Plugin和Maven Surefire Plugin就是最好的例子。本文只涉及了Jetty，如果读者的环境是Tomcat或者JBoss等其他容器，则需要查阅相关的文档以得到具体的实现细 节，你可能对Tomcat Maven Plugin、JBoss Maven Plugin、或者Cargo Maven2 Plugin感兴趣。

本文已经首发于InfoQ中文站，版权所有，原文为《Maven实战（五）——自动化Web应用集成测试》
