-1SoapUI使用说明

# 1 SoapUI 介绍

由于 Web 服务是被程序调用的， 一般不会提供界面让最终用户或测试人员直接使用，在 SoapUI 等工具出现之前，测试人员不得不自己编写程序来测试它， 这就要求测试人员花费很大的精力了解底层的接口，调用关系和详细的协议，导致他们不能把注意力集中到测试中。

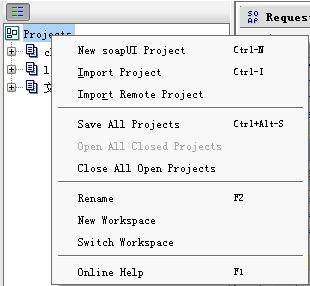
SoapUI 的出现极大的改变了这一局面。作为一个开源的工具，SoapUI 强大的功能、易用的界面，吸引了很多用户。用户可以在 SoapUI 中通过简单的操作完成复杂的测试，不需要了解底层的细节， 极大的减轻了工作量。SoapUI 支持多样的测试， 例如功能测试，性能测试，回归测试等。到目前为止 SoapUI 的下载量已经超过了100万次，成为了Web服务测试标准和领先的 Web 服务测试工具。它不仅仅可以测试基于SOAP的Web 服务，也可以测试REST风格的 Web服务，后者也是本文介绍的重点。

SoapUI基于Java开发，支持多个平台，安装非常简单。读者可以到 SoapUI的 官方网站下载一个安装包 ( 本文使用的是 Window 版本 3.0.1)，直接安装即可。在该安装包中，包括了一个 SoapUI 所需要的 JRE1.6 版本。安装完毕以后，读者需要设置 JAVA\_HOME 变量指向到相应的 JRE 目录，同时修改 PATH 变量，将 JRE1.6 的 bin 目录添加进去。

# 2 SoapUI使用过程

## 2.1创建/导入工程

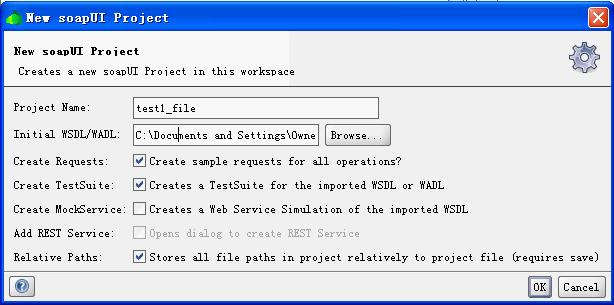
安装并运行SoapUI之后，你就可以创建第一个SoapUI工程了。程序第一次打开时，左侧导航面板上，自动有一个空的Projects工程。右击左侧导航面板中的工作空间节点“Projects”，选择 “New SoapUI Project”。



图表 2-1

页面弹出“New SoapUI Project”TAB页，填入Project Name，Initial WSDL/WADL可填入URL地址或直接导入WSDL文件，导入文件后，

如下图所示：

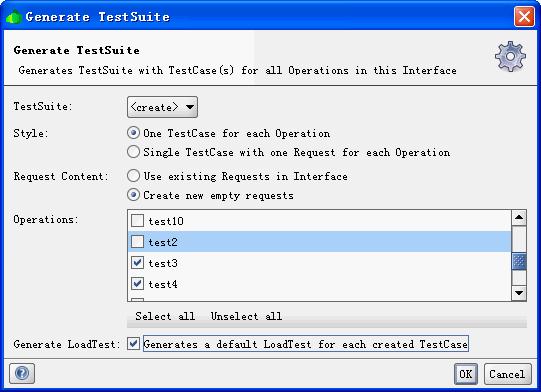


图表 2-2

默认选上：

Create sample requests for all operations?（说明：为每个接口创建一个请求的例子） Creates a TestSuite for the imported WSDL or WADL（说明：为WSDL或WADL创建一个测试包）

点击OK按钮后，页面弹出保存工程的提示，以project名称+“- soapui-project.xml”的形式进行命名，因此上述工程在保存时页面给出默认命名为test1\_file-soapui-project.xml，直接点击保存即可。保存成功后，页面继续弹出“Generate TestSuite”TAB页：



图表 2-3

选择：

Single TestCase with one Request for each Operation（说明：为每个接口的请求都创建一个测试用例）

Create new empty requests（说明：创建一个空的请求）

Operations中选择要测试的WS接口方法，如果一个WS有多个方法，

Operations中会列出所有方法，只须选择要测试的方法即可，上图，去掉了test10、test2等接口的测试。

最后勾选上Generates a default LoadTest for each created TestCase（说明：为每个创建好的测试用例生成一个默认的负载测试）

选择完毕后，点击OK按钮， 进入测试用例命名页面，命名完毕后，确定。

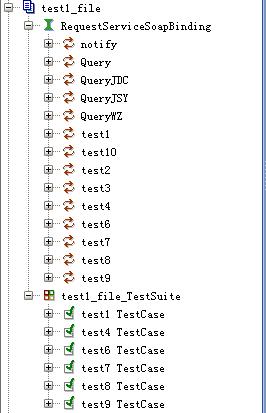


图表 2-4

在测试用例编写完毕后，可使用ctrl+s键，保存当前的工程。 如果要导入其他人的工程，可通过选择“Import Project”，找到test-soapui-project.xml，选中后即可导入工程。

## 2.2 创建测试用例

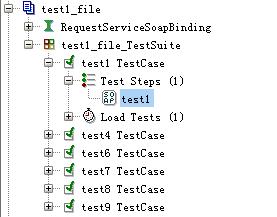
上面操作已经增加了test1的Web服务，接下来可以执行请求了。在上面增加接口的时候，已经根据WSDL的Schema定义为每一个操作创建了默认请求。



图表 2-5

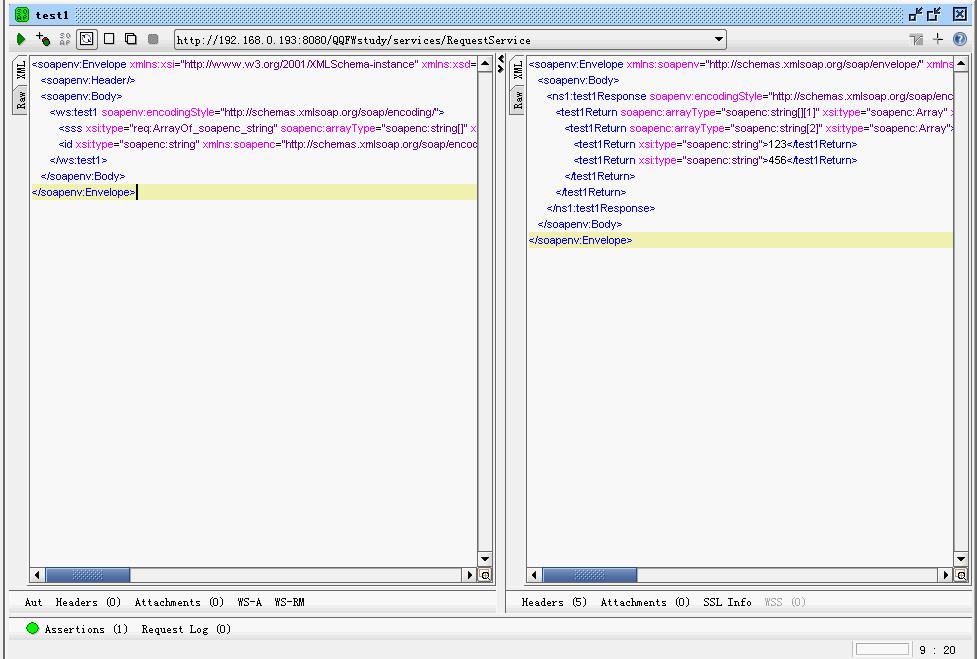
在RequestServiceSoapBinding节点下展开了WS服务中所有的方法，而我们的测试包test1\_file\_TestSuite中根据“创建、导入工程”的第4步，而仅创建了我们要测试的方法的测试用例。

现在将以测试test1方法为例，来介绍用例的创建过程。按照下图所示，打下测试包下的“test1 TestCase”，在展开的“Test Steps”下选择“test1”，双击打开。



图表 2-6

双击“test1”后，在SoapUI的右侧会出现请求编辑器：



图表 2-7

请求编辑器分为三部分：

1.顶部的工具栏，包含一组请求相关的动作、操作

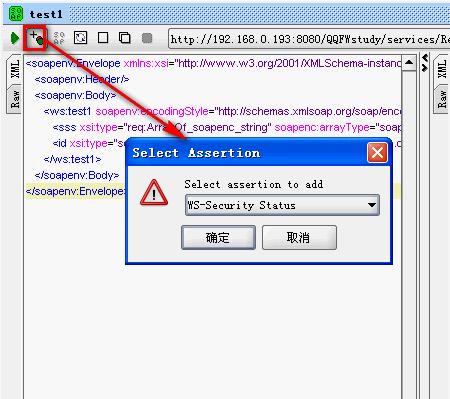
2.左边是请求区域

3.右边是响应区域

SoapUI默认生成的请求中，”?”表示需要被替换的内容。根据需要，可以替换或者删除掉这些值。本接口需要一个名为id的入参，可在请求区域找到如下内容：<id xsi:type="soapenc:string"xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">?</id> “id”即为参数名，找到上面的“？”，替换为abcd任意字符串。

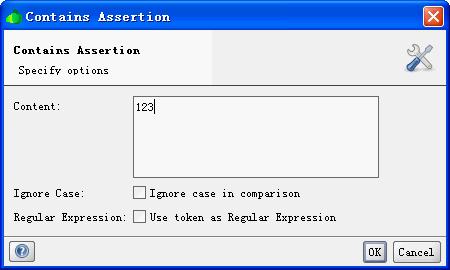
通过按下工具栏最左边的按钮（绿色箭头）来发送本次请求，请求会在后台执行，响应内容会出现在编辑器的右边，test1方法没有任何逻辑，任意的入参均不会影响到输出结果，出参为一个一维数组，第一个值为123，第二个值为456。

根据上述返回的结果报文后，可看到接口已被正确的调用，为在测试中不用人为地进行接口功能是否正确的判断，因此加入断言Assertions，可由程序直接对返回结果进行判断。点击下图左上角的增加断言按钮：



图表 2-8

会弹出“Select Assertion”对话框，通过下拉框选择“Contains”的断言，确定后弹出如下对话框，在Content中填入内容，此处是表示返回的结果报文里应该包含的字段，根据我们test1接口的返回值，填写如下，点击“OK”，插入断言完毕，程序会在运行用例时，自动帮我们校验返回的结果报文是否包含“123”内容。



图表 2-9

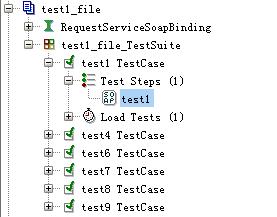
说明：

“Test Steps”中可创建多个测试用例，组成一个测试用例集，在运行该test steps时，会根据用例的顺序从上到下将用例进行一次测试，将上一用例的输出作为下一用例的输入再组织相应的用例，此处待进一步研究。

## 2.3 创建负载测试

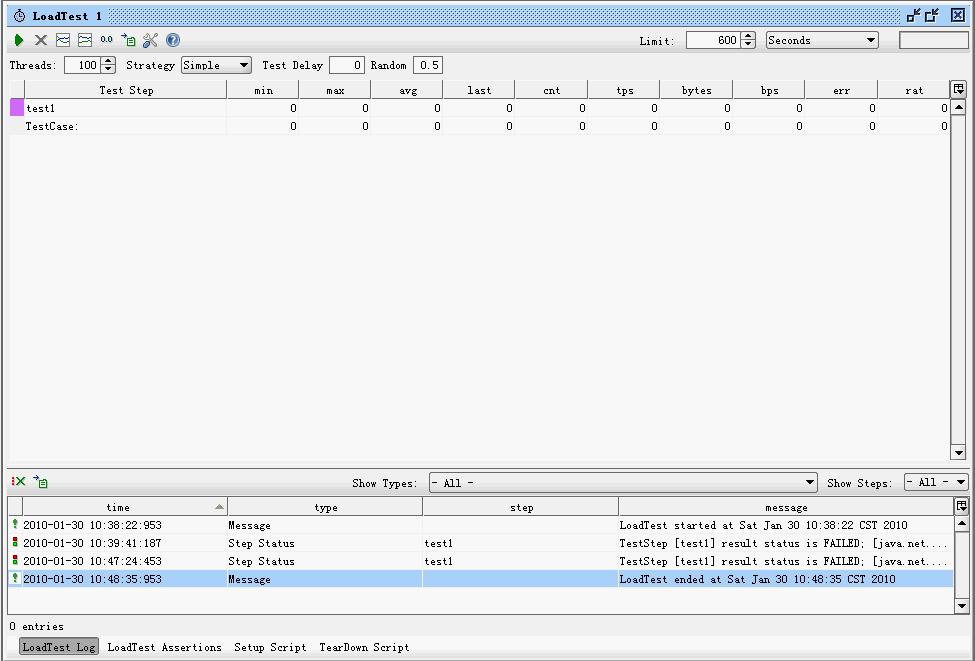
性能测试一般使用loadrunner，或者自己写的调用客户端进行测试。loadrunner是全面的性能测试工具，对一般开发人员来说太重，并且需要license。自己写调用的客户端则测试的统计数据也需要写程序处理，比较麻烦。这里推荐使用SoapUI，SOAPUI可以直接根据WSDL生成SOAP数据包，手工填入参数后可以直接进行性能测试。

在创建完测试用例后，本工程的负载脚本也由在最初创建好工程时，已经默认创建完毕，在此可直接打开使用，如下，可直接点开Load Tests节点，节点下包含名称为“LoadTest1”的负载脚本，双击打开。



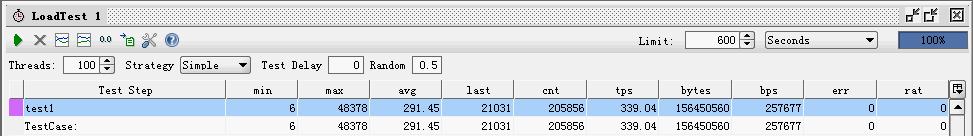
图表 2-10

双击打开后，页面如下显示，设置过程参考如下，场景为100用户并发，持续运行10分钟，没有思考时间。相应的SoapUI可设置Threads=100，Test Delay=0，Limit=600，后面的下拉框选择Seconds，表示600秒。设置完毕后，点击左上方的绿色箭头，程序开始进行负载测试。



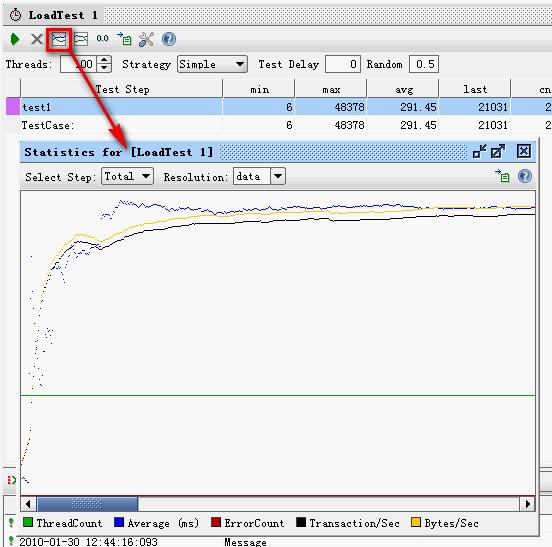
图表 2-11

负载测试过程中，右上方会有进度条显示测试的进度情况，SoapUI提供了2个图表和一个简要列表的形式列出了测试过程中相关数据的监控，如下图，下图为简要列表形式提供的数据：



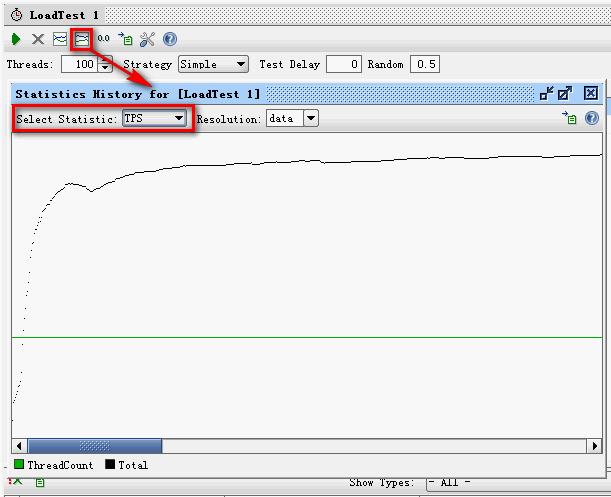
图表 2-12

点击上方红色方框框住的按钮，会弹出下方的监控图表，图中只有曲线，没有任何数据说明，只能看到变化的情况，由于无相应的刻度，而无法直观地看出数据大小：



图表 2-13

SoapUI还提供了另一个图表，此图表与上与图表类似，不过仅能显示线程数与另一统计内容的曲线变化情况，另一统计内容可通过下图红色方框里的“select statistic”进行选择，如下：



图表 2-14

# 3与LoadRunner的比较

使用LoadRunner提供的Webservice协议进行相同接口的测试。

不加校验的脚本（脚本名称：LR\_1 ）如下：

//@oolong 2/2/2010

Action()

{

lr\_start\_transaction("here\_start");

web\_service\_call( "StepName=test1\_101",

"SOAPMethod=RequestJaxRPCService.RequestJaxRPC.test1",

"ResponseParam=response",

"WSDL=C:/Documents and Settings/Owner/桌面

/RequestService.wsdl",

"UseWSDLCopy=1",

"Snapshot=t1264818214.inf",

BEGIN\_ARGUMENTS,

"xml:sss=<sss><string></string></sss>",

"id=aff",

END\_ARGUMENTS,

BEGIN\_RESULT,

END\_RESULT,

LAST);

lr\_end\_transaction("here\_start", LR\_AUTO);

return 0;

}

加了校验的脚本（脚本名称：LR\_2 ）如下，下面的脚本提供了对返回结果的一个校验，类似SoapUI里提供的断言：

Action()

{

char com[] = "123";

lr\_start\_transaction("here\_start");

web\_service\_call( "StepName=test1\_101",

"SOAPMethod=RequestJaxRPCService.RequestJaxRPC.test1",

"ResponseParam=response",

"WSDL=C:/Documents and Settings/Owner/桌面

/RequestService.wsdl",

"UseWSDLCopy=1",

"Snapshot=t1264818214.inf",

BEGIN\_ARGUMENTS,

"xml:sss=<sss><string></string></sss>",

"id=aff",

END\_ARGUMENTS,

BEGIN\_RESULT,

"test1Return[1]=Param\_result",

END\_RESULT,

LAST);

if(strcmp(lr\_eval\_string("{Param\_result}"),com)==0)

{

lr\_end\_transaction("here\_start", LR\_AUTO);

lr\_vuser\_status\_message("成功");

}

else

{

lr\_end\_transaction("here\_start", LR\_FAIL);

lr\_error\_message(lr\_eval\_string("{Param\_result}"));

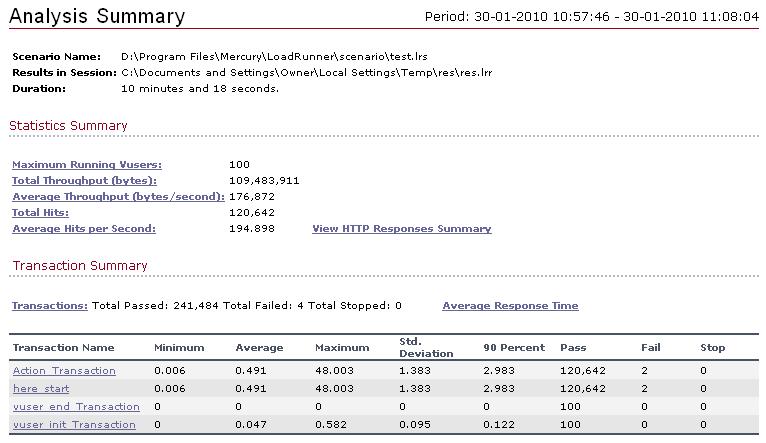
}

return 0;

}

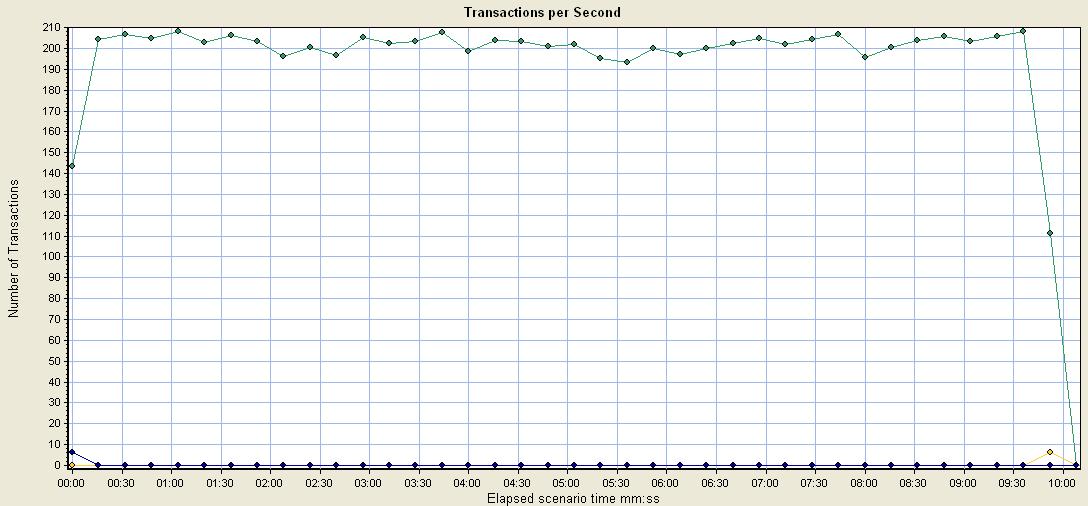
场景与SoapUI的场景一致：100用户并发，持续运行10分钟，没有思考时间。对LR\_2脚本进行性能测试后，发现响应时间比使用SoapUI进行测试的响应时间来的大，因此把校验过程注释掉，使用LR\_1，又进行了一次负载测试。

从LR可以得到的结果图表较多，以下列出几个示意图：



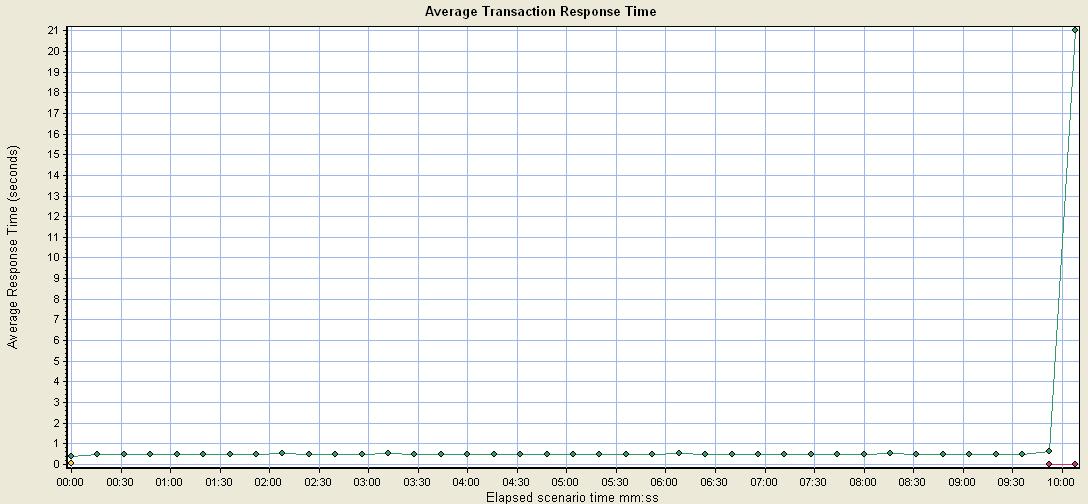
图表 3-1

TPS图如下：



图表 3-2

平均事务响应时间如下：



图表 3-3

可以看到由LR得到的结果，图表丰富，数据完整，提供了更好、更直观的说明作用。

性能测试结果数据比较

|  |  |  |  |
| --- | --- | --- | --- |
| **脚本名称** | **平均响应时间** | **总事务数** | **TPS** |
| SoapUI脚本 | 291.45MS | 205856 | 339.04 |
| LR\_1 | 0.491S | 120646 | 194.898 |
| LR\_2 | 0.606S | 96636 | 159.464 |

由上表及上面的分析得出以下结论：

SoapUI是专门针对ws接口的测试工具，在实现对相同接口测试时，SoapUI表现出来的性能更优越。

SoapUI在发送请求时，是直接以组装好的soap报文进行发送，而LR是使用web\_service\_call方法，从方法传入相应的参数，再由LR组装为soap报文后，再发往接口进行调用，因此LR在组装报文时，会有相应时间的耗费。LR脚本中创建的事务，就包含了这段组装报文的时间，因此响应时间会比SoapUI的响应时间更大。LR与SoapUI的差别应该还有更多，在此我尚未研究的更深入。

对于LR，在测试中若增加对返回结果的校验，也会耗费一定的时间，从上面的数据可以看出，时间差大约0.12s左右，这也与校验中使用的方法有关系，如果方法高效的话，这个时间差也将更少。

全 SoapUI提供的结果数据的分析不如LR那么详细与全面，但对于接口级的测试已足够，且速度更优。

目前WS接口有多种语言可以实现，除了JAVA、C++，当前还有遇到WCF，生成的WSDL文件无法直接读到接口的入参与出参，此种接口生成的WSDL，LoadRunner读取时直接失败，暂找不到解决方法。而使用SoapUI，本人已测试过，可支持java、c++，且wcf这种形式的接口也可支持。

## [[SoapUI] JsonPath 语法 与 XPath 对比](http://www.cnblogs.com/MasterMonkInTemple/p/4772895.html)

|  |  |  |
| --- | --- | --- |
| **XPath** | **JSONPath** | **Description** |
| / | $ | the root object/element 根对象 |
| . | @ | the current object/element 当前对象 |
| / | . or [] | child operator 子对象 |
| .. | n/a | parent operator 父对象 |
| // | .. | recursive descent. JSONPath borrows this syntax from E4X. |
| \* | \* | wildcard. All objects/elements regardless their names. |
| @ | n/a | attribute access. JSON structures don't have attributes. |
| [] | [] | subscript operator. XPath uses it to iterate over element collections and for [predicates](http://www.w3.org/TR/xpath#predicates). In Javascript and JSON it is the native array operator. |
| | | [,] | Union operator in XPath results in a combination of node sets. JSONPath allows alternate names or array indices as a set. |
| n/a | [start:end:step] | array slice operator borrowed from ES4. |
| [] | ?() | applies a filter (script) expression. |
| n/a | () | script expression, using the underlying script engine. |
| () | n/a | grouping in Xpath |

**JSONPath expressions can use the dot–notation**

$.store.book[0].title

or the bracket–notation

$['store']['book'][0]['title']

**The following XPath expression**

/store/book[1]/title

JsonPath would look like

x.store.book[0].title

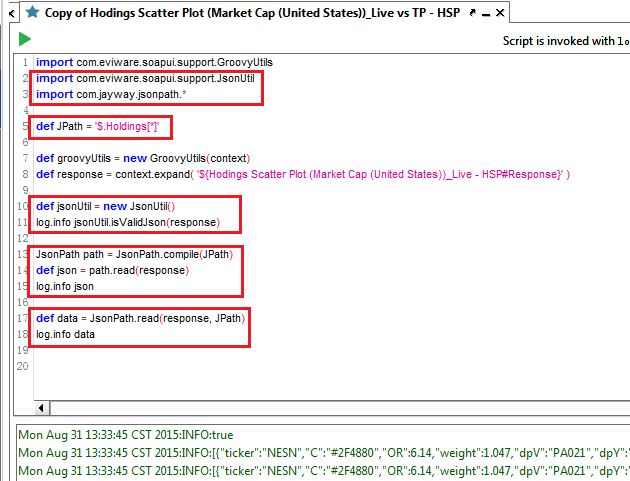
or

x['store']['book'][0]['title']

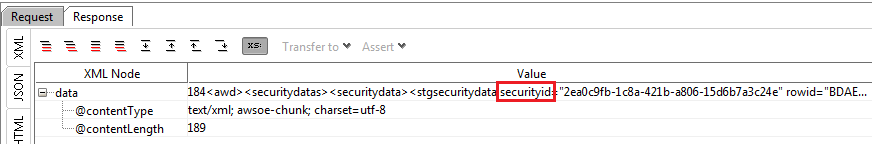
#### JSONPath examples

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | { "store": {      "book": [        { "category": "reference",          "author": "Nigel Rees",          "title": "Sayings of the Century",          "price": 8.95        },        { "category": "fiction",          "author": "Evelyn Waugh",          "title": "Sword of Honour",          "price": 12.99        },        { "category": "fiction",          "author": "Herman Melville",          "title": "Moby Dick",          "isbn": "0-553-21311-3",          "price": 8.99        },        { "category": "fiction",          "author": "J. R. R. Tolkien",          "title": "The Lord of the Rings",          "isbn": "0-395-19395-8",          "price": 22.99        }      ],      "bicycle": {        "color": "red",        "price": 19.95      }    }  } |

|  |  |  |
| --- | --- | --- |
| **XPath** | **JSONPath** | **Result** |
| /store/book/author | $.store.book[\*].author | the authors of all books in the store |
| //author | $..author | all authors |
| /store/\* | $.store.\* | all things in store, which are some books and a red bicycle. |
| /store//price | $.store..price | the price of everything in the store. |
| //book[3] | $..book[2] | the third book |
| //book[last()] | $..book[(@.length-1)] $..book[-1:] | the last book in order. |
| //book[position()<3] | $..book[0,1] $..book[:2] | the first two books |
| //book[isbn] | $..book[?(@.isbn)] | filter all books with isbn number |
| //book[price<10] | $..book[?(@.price<10)] | filter all books cheapier than 10 |
| //\* | $..\* | all Elements in XML document. All members of JSON structure. |



## [[SoapUI] 通过正则表达式从xml格式的response中提取ID](http://www.cnblogs.com/MasterMonkInTemple/p/6668916.html)



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | import com.eviware.soapui.support.GroovyUtils  import java.util.regex.\*     //Get response  def groovyUtils = new GroovyUtils( context )  def holder = groovyUtils.getXmlHolder( "addSecurity#ResponseAsXml" )    //Parse securityId  def data = holder.getNodeValue("//data[1]/text()")  def securityId    Pattern p = ~/securityid=\"\w{8}-\w{4}-\w{4}-\w{4}-\w{12}/  Matcher m = p.matcher(data)    while(m.find()){      securityId = m.group()      securityId = securityId.split("\"")[1]      log.info "the new securityId = "+securityId      testRunner.testCase.setPropertyValue( "securityID", securityId )  } |

http://www.cnblogs.com/MasterMonkInTemple/