烽燧信息科技

**接口测试指导手册**

内部文档(版本号V1.0)

更新记录

| **日期** | **版本号** | **作者** | **说明** |
| --- | --- | --- | --- |
| 2017-08-10 | v1.0 | Cacho | 创建 |

目录

[1. 接口测试介绍 3](#_Toc490226366)

[2. Web Service介绍 5](#_Toc490226367)

[2.1 Web Service基本概念 5](#_Toc490226368)

[2.2 Webservice两种方式 12](#_Toc490226369)

[3. 接口测试工具SoapUI 24](#_Toc490226370)

[3.1 SoapUI介绍 24](#_Toc490226371)

[3.2 SoapUI创建/导入项目 25](#_Toc490226372)

[3.3 创建测试用例 27](#_Toc490226373)

[3.4 执行测试用例 28](#_Toc490226374)

[3.5 断言 28](#_Toc490226375)

# 接口测试介绍

接口测试是项目测试的一部分，它测试的主要对象是接口，是测试系统组件间接口的一种测试。

接口测试主要用于检测外部系统与所测系统之间以及内部各系统之间的交互点。测试的重点是检查数据交互、传递、和控制管理过程以及系统间的相互依赖关系等。

* **如何设计接口测试用例？**

首先，明确出发点，和所有的测试一样，接口测试出发点是你要证明所测的程序是错误的。以这个出发点为导向，你的设计行为就会尽量朝这个方向，更易发现问题；

其次，选择好测试对象。对于一个系统做接口测试选择好的测试对象是接口测试关键。一个系统有无数的接口，每个接口如果分别测试，那将是很痛苦的一件事情，而且任何一个内部接口的变动，都将导致我们用例的不可用。

* **接口的分类**

可将这些最外层的接口分为两类：一类是数据进入系统的接口；一类是数据流出系统的接口。

数据进入系统的接口实际是我们用例执行调用的接口，可通过变化参数对这些接口进行调用，模拟外部的使用；

数据流出的接口则是我们用例真正该验证的点。数据从哪里流出，流出时的状态如何，此时系统是什么状态都是我们所应该验证的。

* **确认完整的测试对象的功能**

确认外部接口提供给使用这些接口的外部用户什么样的功能，外部用户真正需要什么样的功能。此两个功能一定要准确详细，用例的设计要严格按照测试对象功能设计才是正确的用例。最后当出发点、对象、功能都确定了，就可以真正设计用例了。

* **详细介绍如何设计一个结构好、可读性高、渗透性强的接口测试用例**

接口测试用例设计和测试用例设计一样，用例设计的内容应该包括：测试环境、测试数据、主要测试功能点、执行操作以及预期结果。

1）接口测试环境分为两种：一种是程序内部的环境；一种是程序所调用外部接口的环境。

2）接口测试测试数据分为接口参数数据和用例执行所需系统数据。数据的设计、准备测试用例的数据上需要花费更多的心思。要通过好的测试数据使用例查找问题。

i. 接口参数数据需对每个参数根据测试接口的实际的功能进行分析，在符合业务逻辑的情况下进行逻辑组合排列，不要遗漏了某些边界值和错误点的数据。

ii. 每个用例执行所需系统数据和接口参数数据尽可能的采用不一样的数据 ，使用例更容易发现问题。

3）测试功能点，如果一个接口功能复杂时推荐对接口用例进行结构划分，这样用例具有更好的可读性和维护性。接口划分原则为以接口提供的功能点的不同进行合适粒度的划分。同一功能点的用例又可根据测试环境的不同、数据的不同进行用例的填充。

4）接口测试用例执行操作非常简单，就是所测接口的调用。

5）预期结果验证，这也是接口用例设计的很关键的一步 ，应该细而不冗余。每个用例均需验证，避免一个用例中重复做相同的验证，提高测试用例的效率。

# Web Service介绍

* 1. Web Service基本概念

Web Service也叫XML Web Service， WebService是一种可以接收从Internet或者Intranet上的其它系统中传递过来的请求，轻量级的独立的通讯技术。是通过SOAP在Web上提供的软件服务，使用WSDL文件进行说明，并通过UDDI进行注册。

**XML：**(Extensible Markup Language)扩展型可标记语言。面向短期的临时数据处理、面向万维网络，是Soap的基础。

**Soap：**(Simple Object Access Protocol)简单对象存取协议。是XML Web Service 的通信协议。当用户通过UDDI找到你的WSDL描述文档后，他通过可以SOAP调用你建立的Web服务中的一个或多个操作。SOAP是XML文档形式的调用方法的规范，它可以支持不同的底层接口，像HTTP(S)或者SMTP。

WSDL：(Web Services Description Language) WSDL 文件是一个 XML 文档，用于说明一组 SOAP 消息以及如何交换这些消息。大多数情况下由软件自动生成和使用。

**UDDI** (Universal Description,Discovery,and Integration) 是一个主要针对Web服务供应商和使用者的新项目。在用户能够调用Web服务之前，必须确定这个服务内包含哪些商务方法，找到被调用的接口定义，还要在服务端来编制软件，UDDI是一种根据描述文档来引导系统查找相应服务的机制。UDDI利用SOAP消息机制（标准的XML/HTTP）来发布，编辑，浏览以及查找注册信息。它采用XML格式来封装各种不同类型的数据，并且发送到注册中心或者由注册中心来返回需要的数据。

* **调用原理：**

Web服务有两层含义：

1）是指封装成单个实体并发布到网络上的功能集合体；

2）是指功能集合体被调用后所提供的服务。

简单地讲，Web服务是一个URL资源，客户端可以通过编程方式请求得到它的服务，而不需要知道所请求的服务是怎样实现的，这一点与传统的分布式组件对象模型不同。

Web服务的体系结构是基于Web服务提供者、Web服务请求者、Web服务中介者三个角色和发布、发现、绑定三个动作构建的。简单地说，Web服务提供者就是Web服务的拥有者，它耐心等待为其他服务和用户提供自己已有的功能；

Web服务请求者就是Web服务功能的使用者，它利用SOAP消息向Web服务提供者发送请求以获得服务；

Web服务中介者的作用是把一个Web服务请求者与合适的Web服务提供者联系在一起，它充当管理者的角色，一般是UDDI。这三个角色是根据逻辑关系划分的，在实际应用中，角色之间很可能有交叉：一个Web服务既可以是Web服务提供者，也可以是Web服务请求者，或者二者兼而有之。图2-1显示了Web服务角色之间的关系:其中，“发布”是为了让用户或其他服务知道某个Web服务的存在和相关信息;“查找（发现）”是为了找到合适的Web服务;“绑定”则是在提供者与请求者之间建立某种联系。

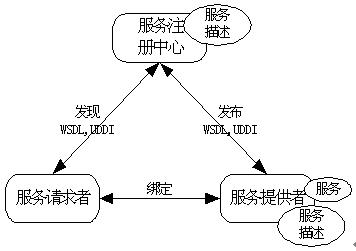


图2-1 Web service的体系结构

* 实现一个完整的Web服务包括以下步骤：

1）Web服务提供者设计实现Web服务，并将调试正确后的Web服务通过Web服务中介者发布，并在UDDI注册中心注册； （发布）

2）Web服务请求者向Web服务中介者请求特定的服务，中介者根据请求查询UDDI注册中心，为请求者寻找满足请求的服务； （发现）

3）Web服务中介者向Web服务请求者返回满足条件的Web服务描述信息，该描述信息用WSDL写成，各种支持Web服务的机器都能阅读；（发现）

4）利用从Web服务中介者返回的描述信息生成相应的SOAP消息，发送给Web服务提供者，以实现Web服务的调用；（绑定）

5） Web服务提供者按SOAP消息执行相应的Web服务，并将服务结果返回给Web服务请求者。（绑定）

* 调用方式：

1） Net下采用GET/POST/SOAP方式动态调用WebService的简易灵活方法(C#)

webservice 的调用有3种方式

i. httpget

ii. httppost

iii. httpsoap

soap 的优点是可以传递结构化的 数据，而前两种不行。

btw, soap 最终也是使用 HTTP 传送 XM

* 安全：

Webservice为作为方便的服务被用广大领域使用的同时，也成为了黑客们的美食。在这里，本文将就目前对Webservice安全所能做的改进做简单介绍。

在Webservice中的安全主要分为以下三个方面。

传输 SSL/HTTPS 对连接加密，而不是传输数据

消息 数据加密(XML Encryption) 数字签名(XML-DSIG)

底层架构 利用应用服务安全机制

传输时的安全是最容易被加入到你的Webservice应用中的，利用现有的SSL 和HTTPS协议，就可以很容易的获得连接过程中的安全。然而这种安全实现方法有两个弱点。一是它只能保证数据传输的安全，而不是数据本身的安全，数据一旦到达某地，那么就可以被任何人所查看。而在Webservice中，一份数据可能到达多个地方，而这份数据却不该被所有的接受者所查看。二是它提供的是要么全有要么全无的保护，你不能选择哪部分数据要被保护，而这种可选择性也是在Webservice中所常要用到的。

第二层的保护是对于消息本身的保护。你可以使用已有的XML安全扩展标准，实现数字签名的功能，从而保证你的消息是来自特定方并没有被修改过。XML文件的加密技术从更大程度上加强了Webservice的安全，它能够定制数据传输到后，能否被接受者所查看，进一步完善了传输后的安全，业界也在不断的制定Webservice的安全标准，比如SAML 和 WS-Security。

最后一层保护就是依靠底层架构的安全，这更多的来自于操作系统和某些中间件的保护。比如在J2EE中，主持Webservice的应用服务器。目前很多的J2EE应用服务器都支持Java Authentication and Authorization Service (JAAS)，这是最近被加入到J2SE 1.4当中的。利用主持Webservice的服务器，实现一些安全机制这是很自然的做法。另一种利用底层架构的安全方法就是，做一个独立的负责安全的服务器，Webservice的使用者和创建者都需要与之取得安全信任。

* **特点：**

Web Service的主要目标是跨平台的可互操作性。为了实现这一目标，Web Service 完全基于XML（可扩展标记语言）、XSD（XML Schema）等独立于平台、独立于软件供应商的标准，是创建可互操作的、分布式应用程序的新平台。因此使用Web Service有许多优点:

1) 跨防火墙的通信

如果应用程序有成千上万的用户，而且分布在世界各地，那么客户端和服务器之间的通信将是一个棘手的问题。因为客户端和服务器之间通常会有防火墙或者代理服务器。传统的做法是，选择用浏览器作为客户端，写下一大堆ASP页面，把应用程序的中间层暴露给最终用户。这样做的结果是开发难度大，程序很难维护。 要是客户端代码不再如此依赖于HTML表单，客户端的编程就简单多了。如果中间层组件换成Web Service的话，就可以从用户界面直接调用中间层组件，从而省掉建立ASP页面的那一步。要调用Web Service，可以直接使用Microsoft SOAP Toolkit或.net这样的SOAP客户端，也可以使用自己开发的SOAP客户端，然后把它和应用程序连接起来。不仅缩短了开发周期，还减少了代码复杂度，并能够增强应用程序的可维护性。同时，应用程序也不再需要在每次调用中间层组件时，都跳转到相应的"结果页"。

2) 应用程序集成

企业级的应用程序开发者都知道，企业里经常都要把用不同语言写成的、在不同平台上运行的各种程序集成起来，而这种集成将花费很大的开发力量。应用程序经常需要从运行的一台主机上的程序中获取数据；或者把数据发送到主机或其它平台应用程序中去。即使在同一个平台上，不同软件厂商生产的各种软件也常常需要集成起来。通过Web Service，应用程序可以用标准的方法把功能和数据"暴露"出来，供其它应用程序使用。

XML Web services 提供了在松耦合环境中使用标准协议（HTTP、XML、SOAP 和 WSDL）交换消息的能力。消息可以是结构化的、带类型的，也可以是松散定义的。

3) B2B的集成

B2B指的是Business to Business，as in businesses doing business with other businesses,商家(泛指企业)对商家的电子商务，即企业与企业之间通过互联网进行产品、服务及信息的交换。通俗的说法是指进行电子商务交易的供需双方都是商家(或企业、公司)，她们使用了Internet的技术或各种商务网络平台，完成商务交易的过程。

Web Service是B2B集成成功的关键。通过Web Service，公司可以只需把关键的商务应用"暴露"给指定的供应商和客户，就可以了，Web Service运行在Internet上，在世界任何地方都可轻易实现，其运行成本就相对较低。Web Service只是B2B集成的一个关键部分，还需要许多其它的部分才能实现集成。 用Web Service来实现B2B集成的最大好处在于可以轻易实现互操作性。只要把商务逻辑"暴露"出来，成为Web Service，就可以让任何指定的合作伙伴调用这些商务逻辑，而不管他们的系统在什么平台上运行，使用什么开发语言。这样就大大减少了花在B2B集成上的时间和成本。

4) 软件和数据重用

Web Service在允许重用代码的同时，可以重用代码背后的数据。使用Web Service，再也不必像以前那样，要先从第三方购买、安装软件组件，再从应用程序中调用这些组件；只需要直接调用远端的Web Service就可以了。另一种软件重用的情况是，把好几个应用程序的功能集成起来，通过Web Service "暴露"出来，就可以非常容易地把所有这些功能都集成到你的门户站点中，为用户提供一个统一的、友好的界面。 可以在应用程序中使用第三方的Web Service 提供的功能，也可以把自己的应用程序功能通过Web Service 提供给别人。两种情况下，都可以重用代码和代码背后的数据。

从以上论述可以看出，Web Service 在通过Web进行互操作或远程调用的时候是最有用的。不过，也有一些情况，Web Service根本不能带来任何好处。

* Web Service有以下缺点：

1）单机应用程序

目前，企业和个人还使用着很多桌面应用程序。其中一些只需要与本机上的其它程序通信。在这种情况下，最好就不要用Web Service，只要用本地的API就可以了。COM非常适合于在这种情况下工作，因为它既小又快。运行在同一台服务器上的服务器软件也是这样。当然Web Service 也能用在这些场合，但那样不仅消耗太大，而且不会带来任何好处。

2）局域网的一些应用程序

在许多应用中，所有的程序都是在Windows平台下使用COM，都运行在同一个局域网上。在这些程序里，使用DCOM会比SOAP/HTTP有效得多。与此相类似，如果一个.net程序要连接到局域网上的另一个.net程序，应该使用.net Remoting。其实在.net Remoting中，也可以指定使用SOAP/HTTP来进行Web Service 调用。不过最好还是直接通过TCP进行RPC调用，那样会有效得多。

3）XML Web Service的应用

最初的 XML Web Service 通常是可以方便地并入应用程序的信息来源，如股票价格、天气预报、体育成绩等等。

以XML Web Service 方式提供现有应用程序，可以构建新的、更强大的应用程序，并利用 XML Web Service 作为构造块。

例如，用户可以开发一个采购应用程序，以自动获取来自不同供应商的价格信息，从而使用户可以选择供应商，提交订单，然后跟踪货物的运输，直至收到货物。而供应商的应用程序除了在Web上提供服务外，还可以使用XML Web Service检查客户的信用、收取货款，并与货运公司办理货运手续。

* 1. Webservice两种方式

第一次接触Webservice，对于很多概念不太理解，尤其是看到各个OpenAPI的不同提供方式时，更加疑惑。如google map api采用了AJAX方式，通过javascript提供API，而淘宝TOP则采用直接的HTTP+XML请求方式，最令我疑惑的是教材上讲的WSDL，UDDI从没有在这些API中出现过。现在知道了WebService原来有两种方式，一是SOAP协议方式，在这种方式下需要WSDL,UDDI等，二是REST方式，这种方式根本不需要WSDL,UDDI等。而且REST方式现在看来是更加流行，更有前途的方式。

在SOA的基础技术实现方式中WebService占据了很重要的地位，通常我们提到WebService第一想法就是SOAP消息在各种传输协议上交互。近几年REST的思想伴随着SOA逐渐被大家接受，同时各大网站不断开放API提供给开发者，也激起了REST风格WebService的热潮。

在收到新需求Email之前，我对REST的理解仅仅是通过半懂不懂的看了Fielding的REST博士论文，说实话当时也就是希望了解这么一个新概念，对于其内部的思想只是很肤浅的了解了一下。

ASF的最新需求就是可能需要实现REST风格的WebService集成，因此不得不好好的去看看REST的真正思想含义以及当前各大网站的设计方式。后面所要表述的也是我这个初学者的一些看法和观点，抛砖引玉，希望在我将REST融入到ASF之前能够获得更多的反馈和意见。

* **SOAP**

其实SOAP最早是针对RPC的一种解决方案，简单对象访问协议，很轻量，同时作为应用协议可以基于多种传输协议来传递消息（Http,SMTP等）。但是随着SOAP作为WebService的广泛应用，不断地增加附加的内容，使得现在开发人员觉得SOAP很重，使用门槛很高。在SOAP后续的发展过程中，WS-\*一系列协议的制定，增加了SOAP的成熟度，也给SOAP增加了负担。

* SOAP消息格式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <?xml version="1.0"?>  <soap:Envelope  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">  <soap:Header>    <m:Trans xmlns:m="http://www.w3schools.com/transaction/"    soap:mustUnderstand="1">234    </m:Trans>  </soap:Header>  <soap:Body>    <m:GetPrice xmlns:m="http://www.w3schools.com/prices">      <m:Item>Apples</m:Item>    </m:GetPrice>  </soap:Body>  </soap:Envelope> |

* **REST 服务介绍**

REST（Representational State Transfer）是 Roy Fielding 博士在2000年提出的一种新的软件架构风格，它以资源（resource）为核心，使用 HTTP、 URI、XML 以及HTML等现有流行协议和标准来完成对资源的操作及显示。这些操作包括获取、创建、修改和删除资源，分别对应于 HTTP 协议的 GET、POST、PUT 和 DELETE 方法。REST 架构定义了以下设计准则：

1）网络中的所有事物都被抽象为资源（resource）。

2）每个资源对应一个唯一的资源标识（resource identifier）。

3）通过通用的连接器接口(generic connector interface)对资源进行操作。

4）对资源的各种操作不会改变资源标识。

5）所有的操作都是无状态的（stateless）。

REST 服务（RESTful Service）是一种基于 HTTP 和 REST 准则的轻量级 Web 服务。这类服务可以看作一系列资源（resource）的集合，服务的定义可以视为以下三个切面的组合 :

1. 访问 Web Service 的 URL，如：http://example.com/resources。
2. Web Service 所支持的数据 MIME 类型，如：JSON, XML, YAML 等。
3. Web Service 使用 HTTP 协议支持的操作，如 GET, POST, PUT, DELETE。

相比目前流行的 Web 服务实现方案 SOAP 和 XML-RPC, REST 服务更加简洁，它可以完全通过HTTP协议实现，还可以利用缓存 Cache 来提高响应速度，其性能，效率和易用性等方面均优于 SOAP 协议。

REST其实并不是什么协议也不是什么标准，而是将Http协议的设计初衷作了诠释，在Http协议被广泛利用的今天，越来越多的是将其作为传输协议，而非原先设计者所考虑的应用协议。SOAP类型的WebService就是最好的例子，SOAP消息完全就是将Http协议作为消息承载，以至于对于Http协议中的各种参数（例如编码，错误码等）都置之不顾。其实，最轻量级的应用协议就是Http协议。Http协议所抽象的get,post,put,delete就好比数据库中最基本的增删改查，而互联网上的各种资源就好比数据库中的记录（可能这么比喻不是很好），对于各种资源的操作最后总是能抽象成为这四种基本操作，在定义了定位资源的规则以后，对于资源的操作通过标准的Http协议就可以实现，开发者也会受益于这种轻量级的协议。

* **自己理解的将REST的思想归结以下有如下几个关键点：**

1. 面向资源的接口设计

所有的接口设计都是针对资源来设计的，也就很类似于我们的面向对象和面向过程的设计区别，只不过现在将网络上的操作实体都作为资源来看待，同时URL的设计也是体现了对于资源的定位设计。后面会提到有一些网站的API设计说是REST设计，其实是RPC-REST的混合体，并非是REST的思想。

1. 抽象操作为基础的CRUD

这点很简单，Http中的get,put,post,delete分别对应了read,update,

create,delete四种操作，如果仅仅是作为对于资源的操作，抽象成为这四种已经足够了，但是对于现在的一些复杂的业务服务接口设计，可能这样的抽象未必能够满足。其实这也在后面的几个网站的API设计中暴露了这样的问题，如果要完全按照REST的思想来设计，那么适用的环境将会有限制，而非放之四海皆准的。

1. Http是应用协议而非传输协议

这点在后面各大网站的API分析中有很明显的体现，其实有些网站已经走到了SOAP的老路上，说是REST的理念设计，其实是作了一套私有的SOAP协议，因此称之为REST风格的自定义SOAP协议。

1. 无状态，自包含

这点其实不仅仅是对于REST来说的，作为接口设计都需要能够做到这点，也是作为可扩展和高效性的最基本的保证，就算是使用SOAP的WebService也是一样。

* **REST VS SOAP**

1. 成熟度：

SOAP虽然发展到现在已经脱离了初衷，但是对于异构环境服务发布和调用，以及厂商的支持都已经达到了较为成熟的情况。不同平台，开发语言之间通过SOAP来交互的web service都能够较好的互通（在部分复杂和特殊的参数和返回对象解析上，协议没有作很细致的规定，导致还是需要作部分修正）

REST国外很多大网站都发布了自己的开发API，很多都提供了SOAP和REST两种Web Service，根据调查部分网站的REST风格的使用情况要高于SOAP。但是由于REST只是一种基于Http协议实现资源操作的思想，因此各个网站的REST实现都自有一套，在后面会讲诉各个大网站的REST API的风格。也正是因为这种各自实现的情况，在性能和可用性上会大大高于SOAP发布的web service，但统一通用方面远远不及SOAP。由于这些大网站的SP往往专注于此网站的API开发，因此通用性要求不高。

ASF上考虑发布REST风格的Web Service，可以参考几大网站的设计（兄弟公司的方案就是参考了类似于flickr的设计模式），但是由于没有类似于SOAP的权威性协议作为规范，REST实现的各种协议仅仅只能算是私有协议，当然需要遵循REST的思想，但是这样细节方面有太多没有约束的地方。REST日后的发展所走向规范也会直接影响到这部分的设计是否能够有很好的生命力。

总的来说SOAP在成熟度上优于REST。

1. 效率和易用性：

SOAP协议对于消息体和消息头都有定义，同时消息头的可扩展性为各种互联网的标准提供了扩展的基础，WS-\*系列就是较为成功的规范。但是也由于SOAP由于各种需求不断扩充其本身协议的内容，导致在SOAP处理方面的性能有所下降。同时在易用性方面以及学习成本上也有所增加。

REST被人们的重视，其实很大一方面也是因为其高效以及简洁易用的特性。这种高效一方面源于其面向资源接口设计以及操作抽象简化了开发者的不良设计，同时也最大限度的利用了Http最初的应用协议设计理念。同时，在我看来REST还有一个很吸引开发者的就是能够很好的融合当前Web2.0的很多前端技术来提高开发效率。例如很多大型网站开放的REST风格的API都会有多种返回形式，除了传统的xml作为数据承载，还有（JSON,RSS,ATOM）等形式，这对很多网站前端开发人员来说就能够很好的mashup各种资源信息。

因此在效率和易用性上来说，REST更胜一筹。

1. 安全性：

这点其实可以放入到成熟度中，不过在当前的互联网应用和平台开发设计过程中，安全已经被提到了很高的高度，特别是作为外部接口给第三方调用，安全性可能会高过业务逻辑本身。

SOAP在安全方面是通过使用XML-Security和XML-Signature两个规范组成了WS-Security来实现安全控制的，当前已经得到了各个厂商的支持，.net ，php，java都已经对其有了很好的支持（虽然在一些细节上还是有不兼容的问题，但是互通基本上是可以的）。

REST没有任何规范对于安全方面作说明，同时现在开放REST风格API的网站主要分成两种，一种是自定义了安全信息封装在消息中（其实这和SOAP没有什么区别），另外一种就是靠硬件SSL来保障,但是这只能够保证点到点的安全，如果是需要多点传输的话SSL就无能为力了。安全这块其实也是一个很大的问题，今年在BEA峰会上看到有演示采用SAML2实现的网站间SSO，其实是直接采用了XML-Security和XML-Signature，效率看起来也不是很高。未来REST规范化和通用化过程中的安全是否也会采用这两种规范，是未知的，但是加入的越多，REST失去它高效性的优势越多。

1. 应用设计与改造：

我们的系统要么就是已经有了那些需要被发布出去的服务，要么就是刚刚设计好的服务，但是开发人员的传统设计思想让REST的形式被接受还需要一点时间。同时在资源型数据服务接口设计上来说按照REST的思想来设计相对来说要容易一些，而对于一些复杂的服务接口来说，可能强要去按照REST的风格来设计会有些牵强。这一点其实可以看看各大网站的接口就可以知道，很多网站还要传入function的名称作为参数，这就明显已经违背了REST本身的设计思路。

而SOAP本身就是面向RPC来设计的，开发人员十分容易接受，所以不存在什么适应的过程。

总的来说，其实还是一个老观念，适合的才是最好的

技术没有好坏，只有是不是合适，一种好的技术和思想被误用了，那么就会得到反效果。REST和SOAP各自都有自己的优点，同时如果在一些场景下如果去改造REST，其实就会走向SOAP（例如安全）。

REST对于资源型服务接口来说很合适，同时特别适合对于效率要求很高，但是对于安全要求不高的场景。而SOAP的成熟性可以给需要提供给多开发语言的，对于安全性要求较高的接口设计带来便利。所以我觉得纯粹说什么设计模式将会占据主导地位没有什么意义，关键还是看应用场景。

同时很重要一点就是不要扭曲了REST,现在很多网站都跟风去开发REST风格的接口，其实都是在学其形，不知其心，最后弄得不伦不类，性能上不去，安全又保证不了，徒有一个看似像模像样的皮囊。

* REST设计的几种形式

参看了几个大型网站的REST风格的API设计，这里做一下大致的说明：

**1）FaceBook:**

请求消息：

在URI设计上采取了类似于REST的风格。例如对于friends的获取，就定义为friends.get，前面部分作为资源定义，后面是具体的操作，其他的API也是类似，资源+操作，因此就算使用http的get方法都可能作了update的操作，其实已经违背了REST的思想，但是说到，其实那么复杂的业务接口设计下，要通过RUCD来抽象所有的接口基本是不实际的。在URI定义好以后，还有详细的参数定义，包括类型以及是否必选。

响应消息：

有多种方式，XML,JSON，XML有XSD作为参考。有点类似于没有Head的SOAP，只不过这里将原来可以定义在WSDL中的XSD抽取出来了。

**2）Flickr:**

请求消息： http://api.flickr.com/services/rest/?method=flickr.test.echo&name=value

这里就可以很明显看出它所定制的REST请求其实和RPC没有什么太大的区别。

消息返回：

正确处理返回

<?xml version="1.0" encoding="utf-8" ?>

<rsp stat="ok">

[xml-payload-here]

</rsp>

错误处理返回

<?xml version="1.0" encoding="utf-8" ?>

<rsp stat="fail">

<err code="[error-code]" msg="[error-message]" />

</rsp>

根据返回可以看出已经违背了REST的思想，还是把Http协议作为传输承载协议，并没有真正意义上使用Http协议作为资源访问和操作协议。

总的来说，只是形式上去模仿REST，自己搞了一套私有协议。

**3）Ebay：**

请求消息：

采用xml作为承载，类似于SOAP，不过去除SOAP消息的封装和包头，同时在请求中附加了认证和警告级别等附加信息。

消息返回：

类似于SOAP消息，不过删除了SOAP的封装和包头，同时在返回结构中增加了消息处理结果以及版本等附加信息。

这个很类似于当前Axis2框架的做法，将SOAP精简，同时根据自身需求丰富了安全，事务等协议内容。

**4）Yahoo Maps：**

请求消息：

采用REST推荐的方式，URI+Parameters。

返回消息：

<?xml version="1.0" encoding="UTF-8"?>

<ResultSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="urn:yahoo:maps"

xsi:schemaLocation="urn:yahoo:maps http://local.yahooapis.com/MapsService/V1/GeocodeResponse.xsd">

<Result precision="address">

<Latitude>37.416384</Latitude>

<Longitude>-122.024853</Longitude>

<Address>701 FIRST AVE</Address>

<City>SUNNYVALE</City>

<State>CA</State>

<Zip>94089-1019</Zip>

<Country>US</Country>

</Result>

</ResultSet>

SOAP的精简xml返回，其他信息，例如出错码等信息由Http协议头来承载。

**5）YouTube：**

请求消息：

可以看到对于资源操作的URI定义也是参数的一部分。

返回消息：

<?xml version="1.0" encoding="utf-8"?>

<ut\_response status="ok">

<user\_profile>

<first\_name>YouTube</first\_name>

<last\_name>User</last\_name>

<about\_me>YouTube rocks!!</about\_me>

<age>30</age>

<video\_upload\_count>7</video\_upload\_count>

</user\_profile>

</ut\_response>

自定义的类SOAP消息。

**6）Amazon：**

请求消息：

https://Amazon FPS web service end point/?AWSAccessKeyId=Your AWSAccessKeyId

&Timestamp=[Current timestamp] &Signature=[Signature calculated from hash of Action and Timestamp]

&SignatureVersion=[Signature calculated from hash of Action and Timestamp]

&Version=[Version of the WSDL specified in YYYY-MM-DD format] &Action=[Name of the API]

&parameter1=[Value of the API parameter1] &parameter2=[Value of the API parameter2]

&...[API parameters and their values]

返回消息：

类似于SOAP的自有协议，消息体中包含了消息状态等附加信息。

**总结：**

看了上面那么多网站的设计，总结一下主要有这么几种设计方式。

请求消息设计：

1． 基本符合REST标准方式：资源URI定义（资源．操作）+参数。这类设计如果滥用get去处理其他类型的操作，那么和2无异。

2． REST风格非REST思想：资源URI定义+参数（包含操作方法名）。其实就是RPC的REST跟风。

3． 类似于SOAP消息，自定义协议，以xml作为承载。（可扩展，例如鉴权，访问控制等），不过那就好比自己定义了一套SOAP和SOAP extends。大型的有实力的网站有的采取此种做法。

响应消息设计：

1. REST标准方式，将Resource State传输返回给客户端，Http消息作为应用协议而非传输协议

2. 以XML作为消息承载体，Http作为消息传输协议，处理状态自包含。

3. 自定义消息格式，类似于SOAP，提供可扩展部分。

作为遵循REST的理念来看我的选择是响应1和请求1的设计。

# 接口测试工具SoapUI

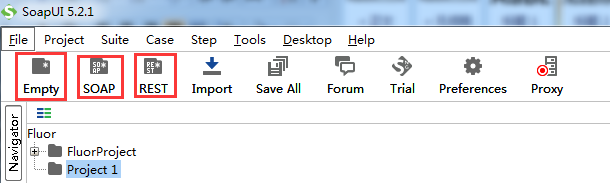
* 1. SoapUI介绍

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。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 5.2.1)，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，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。

* 1. SoapUI创建/导入项目

安装并运行SoapUI之后，你就可以创建第一个SoapUI项目了，程序打开后可以在面板左上角创建3种类型项目：空项目，SOAP,REST项目

完成之后左边的树形结构中生2部分： 服务定义和测试集。

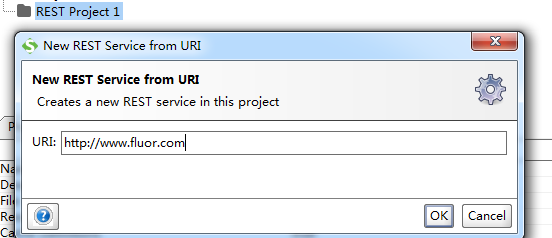


它所提供的测试用例管理与我们项目需要层级映射关系是相当贴近的，对应关系图如下：

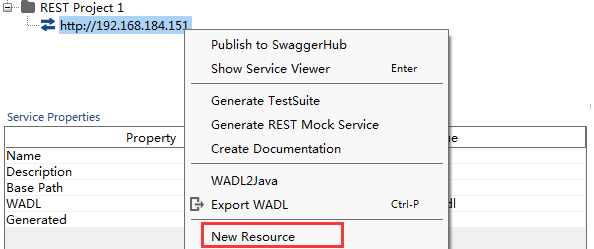


* 创建REST项目

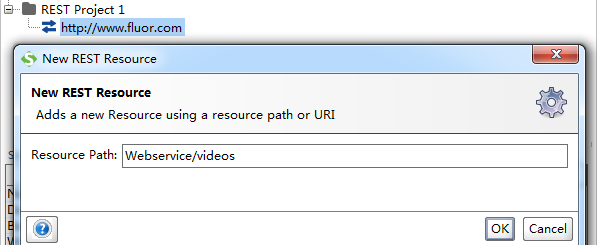
点击REST出现New REST Project窗口，输入URL.

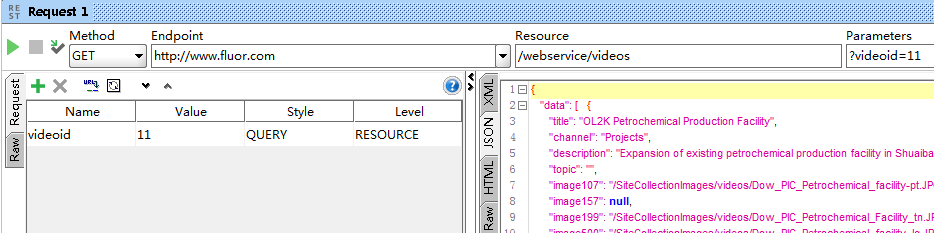


创建新的资源，相当于一个接口



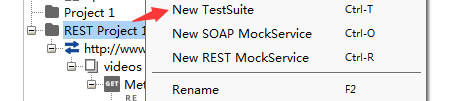
在弹出的窗口填上接口的路径



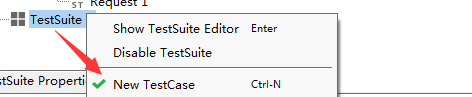
对接口的参数进行设置

* 1. 创建测试用例

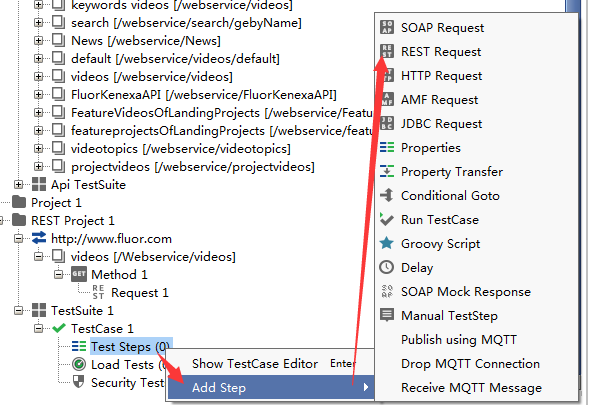
先创建测试用例集



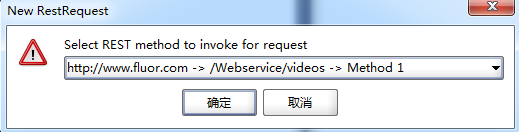
创建测试用例



创建测试步骤

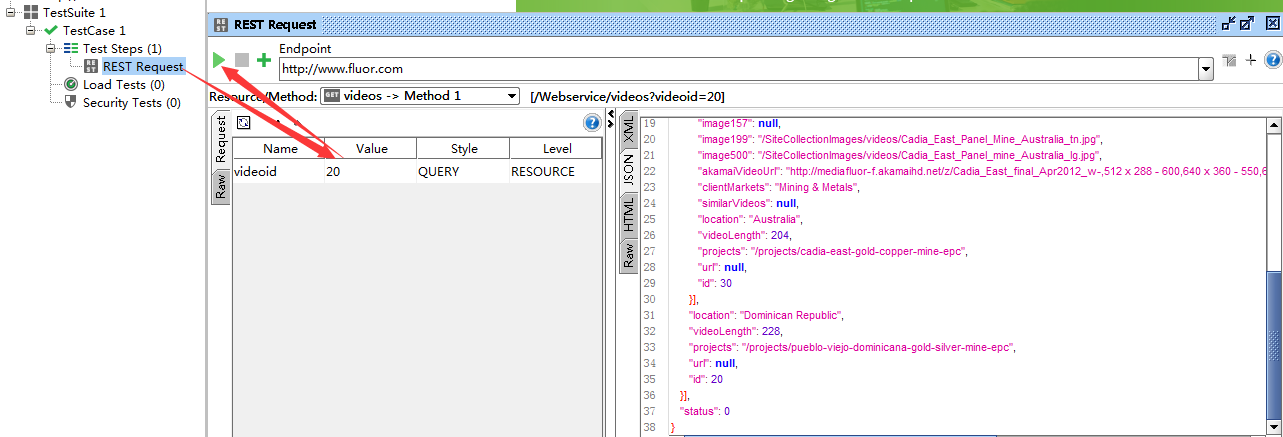


然后选择具体的接口

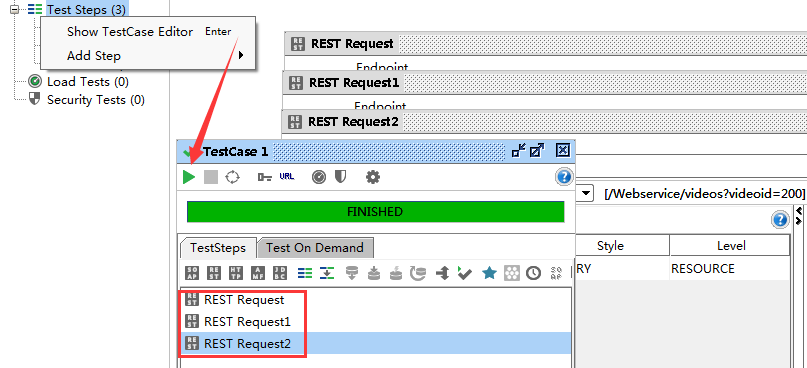


* 1. 执行测试用例

选择需要测试的测试步骤，双击后输入参数，执行测试



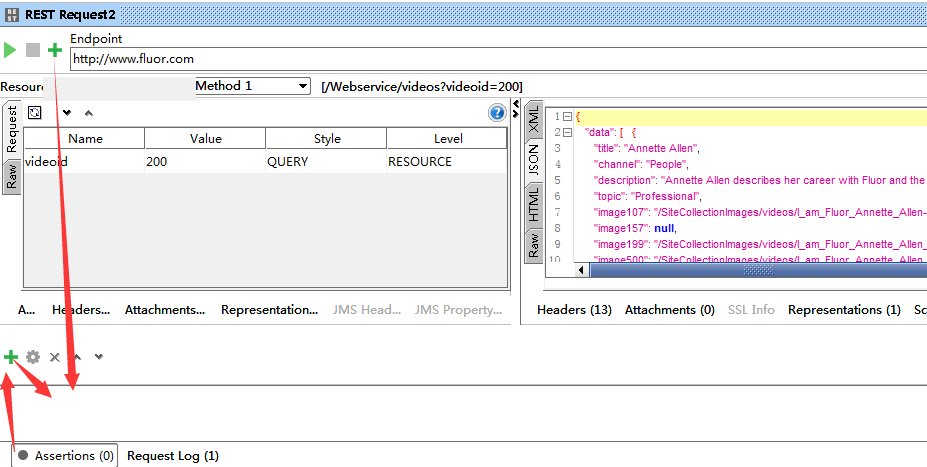
如果有多个测试步骤，可以右键Test Steps，显示TestCase Editer,然后选择需要被执行的测试步骤。



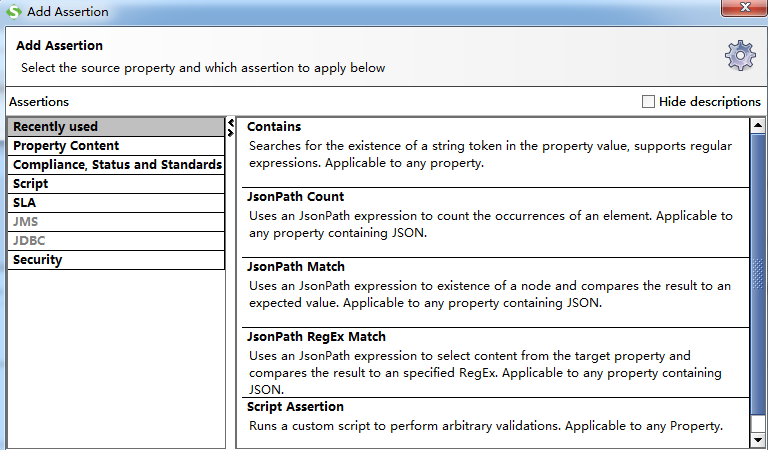
* 1. 断言

为了让工具自动检测测试结果是否符合预期，我们可以使用SoapUI的断言工具。

可以有多种方式调出断言类型：



以下是多种断言类型选择：

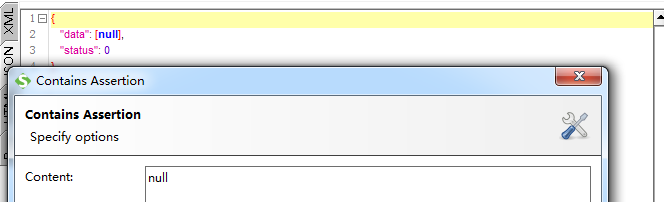


**Contains:** 接口返回数据包含某个文字

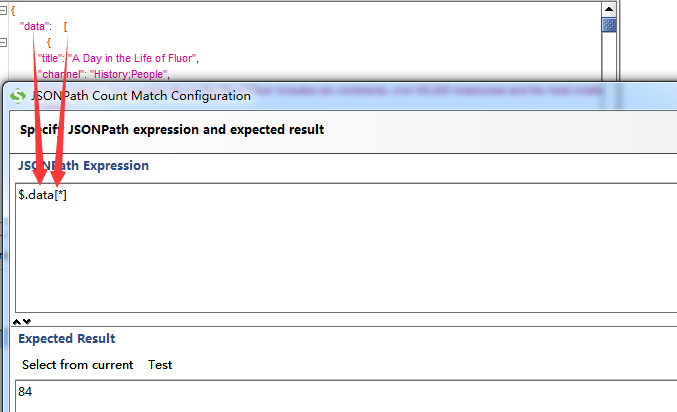
**JsonPath Count:** 接口返回数据的数量。

**JsonPath Match:** 接口返回数据是否正确

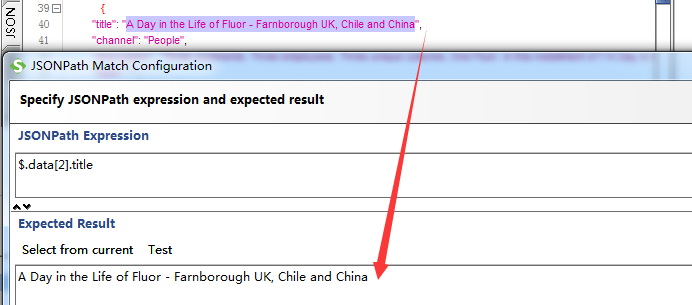
1. 接口返回的数据为JOSN数据时：
2. 当返回的数据只有一条时，可以通过Contains断言函数



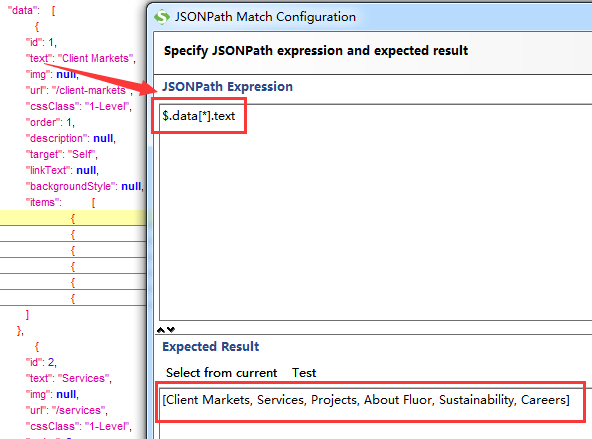
1. 当返回的数据有多条时，可以通过 $.data[\*] 查看总的返回数据数量是否正确。



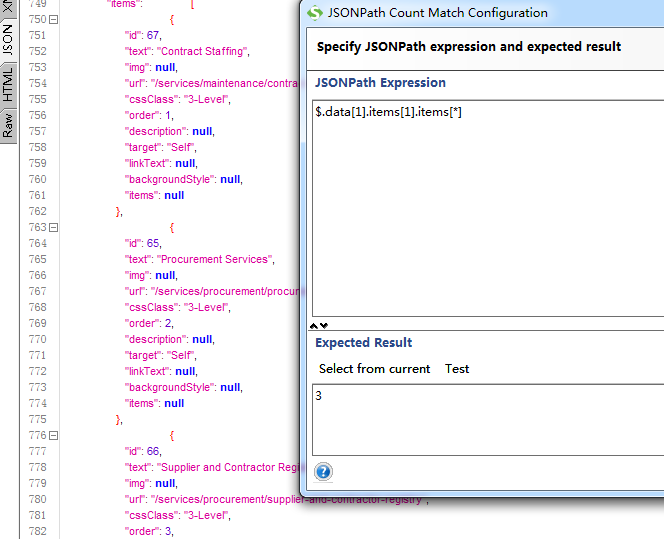
1. 可以抽查第N条数据的某个参数是否正确。$.data[N-1].title



1. 可以查看所有数据的某个参数是否正确。$.data[\*].text

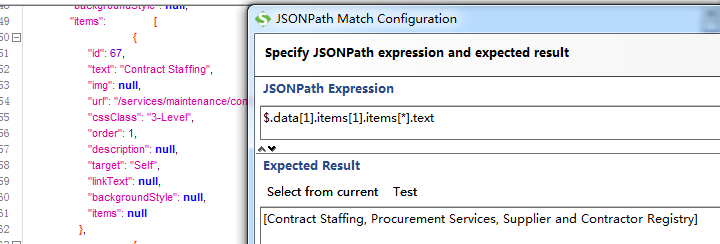


1. 当接口返回的数据包含多层时，比如data [items [items[] ] ]
2. 断言第一层的返回数据数量是否正确 $.data[\*] 。
3. 断言第二层的返回数据数量是否正确 $.data[\*].items[\*]
4. 断言第三层的返回数据数量是否正确 $.data[\*].items[\*].items[\*]
5. 断言第1层第1个数据下级的返回数据数量是否正确 $.data[0].items[\*]
6. 断言第2层第2个数据下级的返回数据数量是否正确 $.data[1].items[1].items[\*]



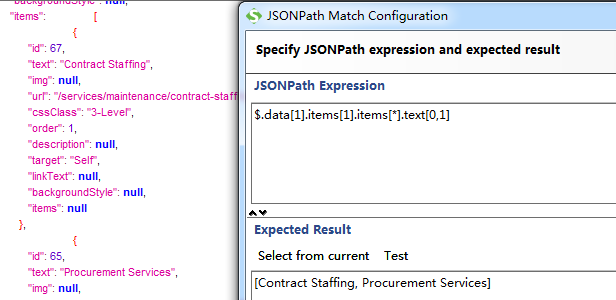
1. 断言第二层第二个数据下级某个参数的返回数据是否正确：

$.data[1].items[1].items[\*].text **严格区分大小写**



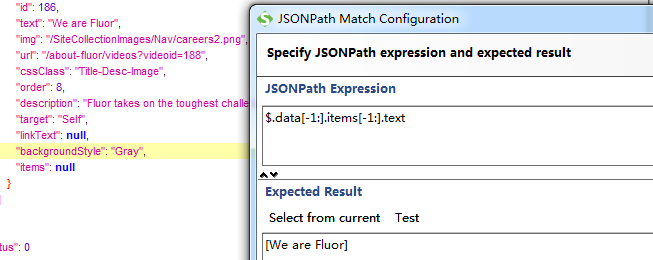
1. 断言第二层第二个数据下级第1~2个数据某个参数的返回数据是否正确。

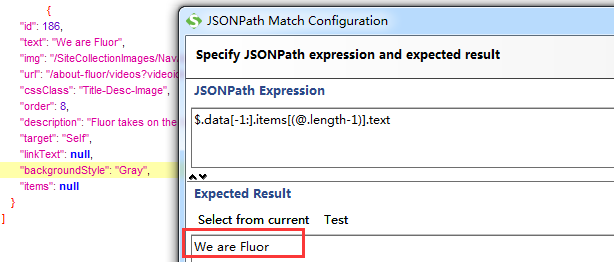
$.data[1].items[1].items[\*].text[0,1] 或者$.data[1].items[1].items[\*].text[:2]



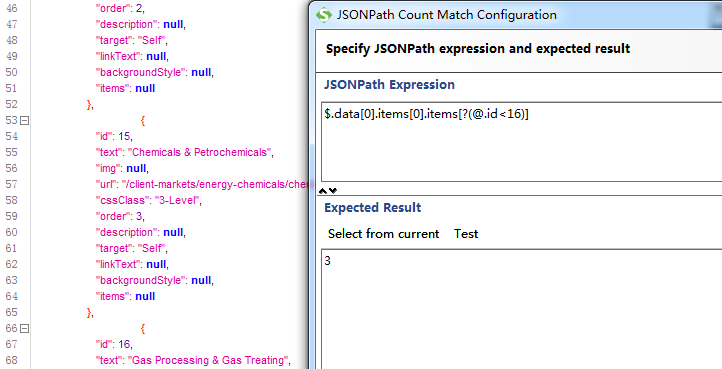
1. 断言最后一个数据的某个参数是否正确，$.data[-1:].items[-1:].text

$.data[-1:].items[(@.length-1)].text





1. 断言某一级的返回数据参数小于某个值的数据或数量。



**具体的规则如下：**

|  |  |  |
| --- | --- | --- |
| **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. |