JSON 是什么？

JSON 的全称是 JavaScript Object Notation，是一种轻量级的数据交换格式。JSON 与 XML 具有相同的特性，例如易于人编写和阅读，易于机器生成和解析。但是 JSON 比 XML 数据传输的有效性要高出很多。JSON 完全独立与编程语言，使用文本格式保存。

JSON 数据有两种结构：

- Name-Value 对构成的集合，类似于 Java 中的 Map。
- Value 的有序列表，类似于 Java 中的 Array。

一个 JSON 格式的数据示例：

```
{
  "Name": "Apple",
  "Expiry": "2007/10/11 13:54",
  "Price": 3.99,
  "Sizes": [
    "Small",
    "Medium",
    "Large"
  ]
}
```

更多关于 JSON 数据格式的说明参看 JSON 官方网站：http://www.json.org（中文内容参看：http://www.json.org/json-zh.html）

GWT 与 JSON

GWT 中支持的客户端服务器端方法调用和数据传递的标准格式是 RPC。 JSON 并不是 GWT 支持的标准的数据传递格式。那么如何使用 JSON 来作为 GWT 的数据传递格式呢？需要以下几步。

第一，引用 HTTP 和 JSON 支持。

第二，在客户端创建 JSON 数据，提交到服务器

第三，在服务器上重写数据格式解析的代码，使之支持 JSON 格式的数据

第四，在服务器上组织 JSON 格式的数据，返回给客户端。

第五，客户端解析服务器传回的 JSON 数据，正确的显示

引用 HTTP 和 JSON 支持

找到.gwt.xml 文件，在其中的

<inherits name='com.google.gwt.user.User'/>

在之后添加如下的内容：

```
    <inherits name="com.google.gwt.json.JSON"/>
<inherits name="com.google.gwt.http.HTTP"/>
```

其中 com.google.gwt.json.JSON 指的是要使用 JSON，com.google.gwt.http.HTTP 值得是通过 HTTP 调用服务器上的服务方法。

客户端构造 JSON 数据

客户端需要使用 com.google.gwt.json.client 包内的类来组装 JSON 格式的数据，数据格式如下：

数据类型	说明
JSONArray	JSONValue 构成的数组类型
JSONBoolean	JSON boolean 值
JSONException	访问 JSON 结构的数据出错的情况下可以抛出此异常
JSONNull	JSON Null 根式的数据
JSONNumber	JSON Number 类型的数据
JSONObject	JSON Object 类型的数据
JSONParser	将 String 格式的 JSON 数据解析为 JSONValue 类型的数据
JSONString	JSON String 类型的数据
JSONValue	所有 JSON 类型值的超级类型

组合一个简单的 JSON 数据：

```
JSONObject input = new JSONObject();
JSONString value = new JSONString("mazhao");
input.put("name", value);
```

JSON 数据格式为：{name: "mazhao"}

组合一个包含数组类型的复杂 JSON 数据：

```
JSONObject input = new JSONObject();
JSONString value = new JSONString("mazhao");
input.put("name", value);
```

```
JSONArray arrayValue = new JSONArray();
arrayValue.set(0, new JSONString("array item 0"));
arrayValue.set(1, new JSONString("array item 1"));
arrayValue.set(2, new JSONString("array item 2"));
input.put("array", arrayValue);
```

JSON 数据格式为：

```
{name: "mazhao",
  array: {"array item 0", "array item 1", "array item 2"}}
```

注意上述的 JSON 类型的数据，使用的都是 com.google.gwt.json.client 包内的类型。这些类型最终会被编译为 JavaScript 执行。

服务端重写数据解析代码，支持 JSON 格式的数据

在服务器上，需要使用 JSON Java 支持类才能将 JSON 格式的数据转换为各种类型的数据，当然也可以自己写一些解析用的代码。这里我们使用了 www.json.org 上的代码来完成。这组代码与 com.google.gwt.json.client 的代码很相似，只是在 org.json 包内部。

怎么解析 JSON 术诀呢？针对上述中的复杂的 JSON 数据：

```
{name: "mazhao",
  array: {"array item 0", "array item 1", "array item 2"}}
```

可以使用如下的方式解析：

```
JSONObject jsonObject = new JSONObject(payload);
String name = jsonObject.getString("name");
```

```
    System.out.println("name is:" + name);
JSONArray jsonArray = jsonObject.getJSONArray("array");
for(int i = 0; i < jsonArray.length(); i++) {
   System.out.println("item " + i + " :" + jsonArray.getString(i));
}
```

其中 payload 指的是上述的 JSON 格式的数据。

那么如何写 GWT 的 Service 来得到 Payload 的数据呢？需要两点，第一，需要建立一个 Service 类，第二，覆盖父类的 processCall 方法。

示例代码：

```
    package com.jpleasure.gwt.json.server;

    import com.google.gwt.user.client.rpc.SerializationException;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import com.jpleasure.gwt.json.client.HelloWorldService;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;

    /**
 * Created by IntelliJ IDEA.
 * User: vaio
 * Date: 2007-9-4
 * Time: 22:08:31
 * To change this template use File | Settings | File Templates.
 */
public class HelloWorldServiceImpl extends RemoteServiceServlet implements
 HelloWorldService {
   public String processCall(String payload) throws SerializationException {
      try {
         JSONObject jsonObject = new JSONObject(payload);
         String name = jsonObject.getString("name");

             System.out.println("name is:" + name);
         JSONArray jsonArray = jsonObject.getJSONArray("array");
         for(int i = 0; i < jsonArray.length(); i++) {
            System.out.println("item " + i + " :" + jsonArray.getString(i));
         }
      } catch (JSONException e) {
         e.printStackTrace();  //To change body of catch statement use File |
```

```
 Settings | File Templates.
      }

           return "success";
   }
}
```

在服务器上组织 JSON 格式的数据，返回给客户端

同上

客户端解析服务器传回的 JSON 数据，正确的显示

同上

Struts2 返回 json 需要 jsonplugin-0[1].25 的包

然后我们的配置文件中需要继承 json-default

Java 代码

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC
3.         "-//Apache Software Foundation//DTD Struts Configuration 2.0//
   EN"
4.         "http://struts.apache.org/dtds/struts-2.0.dtd">
5.
6. <struts>
7.
8.     <package name="com.action.testJson" extends="json-default" nam
   espace="/" >
9.         <action name="jsonUser" class="com.action.testJson.JsonAction
   " method="testUser">
10.            <result type="json"/>
11.        </action>
12.        <!-- Add actions here -->
13.    </package>
14.</struts>
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
```

```
        "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>

    <package name="com.action.testJson" extends="json-default"
namespace="/" >
        <action name="jsonUser" class="com.action.testJson.JsonAction"
method="testUser">
            <result type="json"/>
        </action>
        <!-- Add actions here -->
    </package>
</struts>
```

然后我们的 Action 中需要返回的 json 信息需要加上注解
Java 代码

```
1.  //pizza
2.  package com.action.testJson;
3.
4.  import java.util.ArrayList;
5.  import java.util.List;
6.
7.  import com.googlecode.jsonplugin.annotations.JSON;
8.  import com.opensymphony.xwork2.ActionSupport;
9.
10. public class JsonAction extends ActionSupport {
11.
12.     private static final long serialVersionUID = -
        4082165361641669835L;
13.
14.     Users user=new Users();
15.     List userList=new ArrayList();
16.
17.
18.     public String testUser(){
19.         System.out.println("in the json Acton");
20.         userInit();
21.         userList.add(user);
22.         return SUCCESS;
23.     }
24.
25.     public void userInit(){
```

```
26.         user.setAge(1);
27.         user.setName("张泽峰");
28.         user.setPassword("nofengPassword");
29.     }
30.
31.     @JSON(name="userString")
32.     public Users getUser() {
33.         return user;
34.     }
35.
36.     @JSON(name="userList")
37.     public List getUserList() {
38.         return userList;
39.     }
40.
41.     public void setUser(Users user) {
42.         this.user = user;
43.     }
44.
45.     public void setUserList(List userList) {
46.         this.userList = userList;
47.     }
48.
49.
50. }
```

JSON Plugin

的说明

Added by Musachy Barroso, last edited by ghostroller on Jul 04, 2008 (view change) SHOW COMMENT

Name	JSON Plugin
Publisher	Musachy Barroso
License	Open Source (ASL2)
Version	0.30
Compatibility	Struts 2.0.6 or later
Homepage	http://code.google.com/p/jsonplugin/
Download	http://code.google.com/p/jsonplugin/downloads/list

Rating?	1 2 3 4 5

Overview

The JSON plugin provides a "json" result type that serializes actions into JSON. The serialization process is recursive, meaning that the whole object graph, starting on the action class (base class not included) will be serialized (root object can be customized using the "root" attribute). If the interceptor is used, the action will be populated from the JSON content in the request, these are the rules of the interceptor:

1. The "content-type" must be "application/json"
2. The JSON content must be well formed, see json.org for grammar.
3. Action must have a public "setter" method for fields that must be populated.
4. Supported types for population are: Primitives (int,long...String), Date, List, Map, Primitive Arrays, Other class (more on this later), and Array of Other class.
5. Any object in JSON, that is to be populated inside a list, or a map, will be of type Map (mapping from properties to values), any whole number will be of type Long, any decimal number will be of type Double, and any array of type List.

Given this JSON string:

```
{
   "doubleValue": 10.10,
   "nestedBean": {
      "name": "Mr Bean"
   },
   "list": ["A", 10, 20.20, {
      "firstName": "El Zorro"
   }],
   "array": [10, 20]
}
```

The action must have a "setDoubleValue" method, taking either a "float" or a "double" argument (the interceptor will convert the value to the right one). There must be a "setNestedBean" whose argument type can be any class, that has a "setName" method taking as argument an "String". There must be a "setList" method that takes a "List" as argument, that list will contain: "A" (String), 10 (Long), 20.20 (Double), Map ("firstName" -> "El Zorro"). The "setArray" method can take as parameter either a "List", or any numeric array.

Installation

This plugin can be installed by copying the plugin jar into your application's /WEB-INF/lib directory. No other files need to be copied or created.

To use maven, add this to your pom:

```
<dependencies>
   ...
   <dependency>
       <groupId>com.googlecode</groupId>
       <artifactId>jsonplugin</artifactId>
       <version>0.26</version>
   </dependency>
   ...
</dependencies>
<repository>
    <id>Maven Plugin Repository</id>
    <url>http://struts2plugin-maven-
repo.googlecode.com/svn/trunk/</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <releases>
      <enabled>true</enabled>
    </releases>
</repository>
```

Customizing Serialization and Deserialization

Use the JSON annotation to customize the serialization/deserialization process. Available JSON annotation fields:

Name	Description	Default Value	Serialization	Deserialization
name	Customize field name	empty	yes	no
serialize	Include in serialization	true	yes	no
deserialize	Include in deserialization	true	no	yes

format	Format used to format/parse a Date field	"yyyy-MM-dd'T'HH:mm:ss"	yes	yes

Excluding properties

A comma-delimited list of regular expressions can be passed to the JSON Result and Interceptor, properties matching any of these regular expressions will be ignored on the serialization process:

```
<!-- Result fragment -->
<result type="json">
  <param name="excludeProperties">
    login.password,
    studentList.*\.sin
  </param>
</result>

<!-- Interceptor fragment -->
<interceptor-ref name="json">
  <param name="enableSMD">true</param>
  <param name="excludeProperties">
    login.password,
    studentList.*\.sin
  </param>
</interceptor-ref>
```

Including properties

A comma-delimited list of regular expressions can be passed to the JSON Result to restrict which properties will be serialized. ONLY properties matching any of these regular expressions will be included in the serialized output.

> **Note**
>
> Exclude property expressions take precedence over include property expressions. That is, if you use include and exclude property expressions on the same result, include property expressions will not be applied if an exclude exclude property expression matches a property first.

```
<!-- Result fragment -->
<result type="json">
  <param name="includeProperties">
    ^entries\[\d+\]\.clientNumber,
```

```
    ^entries\[\d+\]\.scheduleNumber,
    ^entries\[\d+\]\.createUserId
  </param>
</result>
```

Root Object

Use the "root" attribute(OGNL expression) to specify the root object to be serialized.

```
<result type="json">
  <param name="root">
    person.job
  </param>
</result>
```

The "root" attribute(OGNL expression) can also be used on the interceptor to specify the object that must be populated, **make sure this object is not null**.

```
<interceptor-ref name="json">
  <param name="root">bean1.bean2</param>
</interceptor-ref>
```

Wrap with Comments

wrapWithComments can turn safe JSON text into dangerous text. For example,

["*/ alert('XSS'); /*"]

Thanks to Douglas Crockford for the tip!

If the "wrapWithComments" (false by default) attribute is set to true, the generated JSON is wrapped with comments like:

```
/* {
   "doubleVal": 10.10,
   "nestedBean": {
      "name": "Mr Bean"
   },
   "list": ["A", 10, 20.20, {
      "firstName": "El Zorro"
   }],
   "array": [10, 20]
} */
```

This can be used to avoid potential Javascript Hijacks. To strip those comments use:

```
var responseObject = eval("("+data.substring(data.indexOf("\/\*")+2,
data.lastIndexOf("\*\/"))+")");
```

Base Classes

By default properties defined on base classes of the "root" object won't be serialized, to serialize properties in all base classes (up to Object) set "ignoreHierarchy" to false in the JSON result:

```
<result type="json">
  <param name="ignoreHierarchy">false</param>
</result>
```

Enumerations

By default, an Enum is serialized as a name=value pair where value = name().

```
public enum AnEnum {
     ValueA,
     ValueB
  }

  JSON:  "myEnum":"ValueA"
```

Use the "enumAsBean" result parameter to serialize Enum's as a bean with a special property _name with value name(). All properties of the enum are also serialized.

```
public enum AnEnum {
     ValueA("A"),
     ValueB("B");

     private String val;

     public AnEnum(val) {
        this.val = val;
     }
     public getVal() {
        return val;
     }
   }

  JSON:  myEnum: { "_name": "ValueA", "val": "A" }
```

Enable this parameter through struts.xml:

```
<result type="json">
  <param name="enumAsBean">true</param>
</result>
```

Compressing the output.

Set the *enableGZIP* attribute to true to gzip the generated json response. The request **must** include "gzip" in the "Accept-Encoding" header for this to work.

```
<result type="json">
  <param name="enableGZIP">true</param>
</result>
```

Example

Setup Action

This simple action has some fields:

Example:

```
import java.util.HashMap;
import java.util.Map;

import com.opensymphony.xwork2.Action;

public class JSONExample {
    private String field1 = "str";
    private int[] ints = {10, 20};
    private Map map = new HashMap();
    private String customName = "custom";

    //'transient' fields are not serialized
    private transient String field2;

    //fields without getter method are not serialized
    private String field3;

    public String execute() {
        map.put("John", "Galt");
        return Action.SUCCESS;
    }

    public String getField1() {
```

```
        return field1;
    }

    public void setField1(String field1) {
        this.field1 = field1;
    }

    public int[] getInts() {
        return ints;
    }

    public void setInts(int[] ints) {
        this.ints = ints;
    }

    public Map getMap() {
        return map;
    }

    public void setMap(Map map) {
        this.map = map;
    }

    @JSON(name="newName")
    public String getCustomName() {
        return this.customName;
    }
}
```

Write the mapping for the action

1. Add the map inside a package that extends "json-default"
2. Add a result of type "json"

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>

  <package name="example"  extends="json-default">
     <action name="JSONExample" class="example.JSONExample">
```

```
        <result type="json"/>
    </action>
  </package>

</struts>
```

JSON example output

```
{
   "field1" : "str",
   "ints": [10, 20],
   "map": {
       "John":"Galt"
   },
   "newName": "custom"
}
```

JSON RPC

The json plugin can be used to execute action methods from javascript and return the output. This feature was developed with Dojo in mind, so it uses Simple Method Definition to advertise the remote service. Let's work it out with an example(useless as most examples).

First write the action:

```
package smd;

import com.googlecode.jsonplugin.annotations.SMDMethod;
import com.opensymphony.xwork2.Action;

public class SMDAction {
    public String smd() {
        return Action.SUCCESS;
    }

    @SMDMethod
    public Bean doSomething(Bean bean, int quantity) {
        bean.setPrice(quantity * 10);
        return bean;
    }
}
```

Methods that will be called remotely **must** be annotated with the *SMDMethod* annotation, for security reasons. The method will take a bean object, modify its price and return it. The action can be annotated with the *SMD* annotation to customize the generated SMD (more on that

soon), and parameters can be annotated with *SMDMethodParameter*. As you can see, we have a "dummy", *smd* method. This method will be used to generate the Simple Method Definition (a definition of all the services provided by this class), using the "json" result.

The bean class:

```
package smd;

public class Bean {
    private String type;
    private int price;

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

}
```

The mapping:

```
<package name="RPC" namespace="/nodecorate" extends="json-default">
    <action name="SMDAction" class="smd.SMDAction" method="smd">
        <interceptor-ref name="json">
            <param name="enableSMD">true</param>
        </interceptor-ref>
        <result type="json">
             <param name="enableSMD">true</param>
        </result>
    </action>
</package>
```

Nothing special here, except that **both** the interceptor and the result must be applied to the action, and "enableSMD" must be enabled for both.

Now the javascript code:

```
<s:url id="smdUrl" namespace="/nodecorate" action="SMDAction" />
<script type="text/javascript">
    //load dojo RPC
    dojo.require("dojo.rpc.*");

    //create service object(proxy) using SMD (generated by the json
result)
    var service = new dojo.rpc.JsonService("${smdUrl}");

    //function called when remote method returns
    var callback = function(bean) {
        alert("Price for " + bean.type + " is " + bean.price);
    };

    //parameter
    var bean = {type: "Mocca"};

    //execute remote method
    var defered = service.doSomething(bean, 5);

    //attach callback to defered object
    defered.addCallback(callback);
</script>
```

Dojo's JsonService will make a request to the action to load the SMD, which will return a JSON object with the definition of the available remote methods, using that information Dojo creates a "proxy" for those methods. Because of the asynchronous nature of the request, when the method is executed, a deferred object is returned, to which a callback function can be attached. The callback function will receive as a parameter the object returned from your action. That's it.

Proxied objects

(V0.20) As annotations are not inherited in Java, some user might experience problems while trying to serializ objects that are proxied. eg. when you have attached AOP interceptors to your action.

In this situation, the plugin will not detect the annotations on methods in your action.

To overcome this, set the "ignoreInterfaces" result parameter to false (true by default) to request that the plugin inspects all interfaces and superclasses of the action for annotations on the action's methods.

NOTE: This parameter should only be set to false if your action could be a proxy as there is a performance cost caused by recursion through the interfaces.

```
<action name="contact" class="package.ContactAction" method="smd">
   <interceptor-ref name="json">
      <param name="enableSMD">true</param>
      <param name="ignoreSMDMethodInterfaces">false</param>
   </interceptor-ref>
   <result type="json">
      <param name="enableSMD">true</param>
      <param name="ignoreInterfaces">false</param>
   </result>
   <interceptor-ref name="default"/>
</action>
```

在 Struts 2 中使用 JSon ajax 支持

来源： 作者： 发布时间：2007-12-19

JSON 插件提供了一种名为 json 的 ResultType，一旦为某个 Action 指定了一个类型为 json 的 Result，则该 Result 无需映射到任何视图资源。因为 JSON 插件会负责将 Action 里的状态信息序列化成 JSON 格式的数据，并将该数据返回给客户端页面的 JavaScript。

简单地说，JSON 插件允许我们在 JavaScript 中异步调用 Action，而且 Action 不再需要使用视图资源来显示该 Action 里的状态信息，而是由 JSON 插件负责将 Action 里的状态信息返回给调用页面——通过这种方式，就能够完成 Ajax 交互。

Struts2 提供了一种可插拔方式来管理插件，安装 Struts2 的 JSON 插件和安装普通插件并没有太大的区别，相同只需要将 Struts2 插件的 JAR 文档复制到 Web 应用的 WEB-INF/lib 路径下即可。

安装 JSON 插件按如下步骤进行：

（1）登陆 http://code.google.com/p/jsonplugin/downloads/list 站点，下载 Struts2 的 JSON 插件的最新版本，当前最新版本是 0.7，我们能够下载该版本的 JSON 插件。

（2）将下载到的 jsonplugin-0.7.jar 文档复制到 Web 应用的 WEB-INF 路径

下，即可完成 JSON 插件的安装。

实现 Actio 逻辑

假设 wo,en 输入页面中包含了三个表单域，这三个表单域对于三个请求参数，因此应该使用 Action 来封装这三个请求参数。三个表单域的 name 分别为 field1 、field2 和 field3 。

处理该请求的 Action 类代码如下：

```
public class JSONExample
{
  //封装请求参数的三个属性
  private String field1;
  private transient String field2;
  private String field3;
  //封装处理结果的属性
  private int[] ints = {10, 20};
  private Map map = new HashMap();
  private String customName = "custom";
  //三个请求参数对应的 setter 和 getter 方法
  public String getField1()
  {
    return field1;
  }
  public void setField1(String field1)
  {
    this.field1 = field1;
  }
  //此处省略了 field1 和 field2 两个字段的 setter 和 getter 方法
  ...
  //封装处理结果的属性的 setter 和 getter 方法
  public int[] getInts()
  {
    return ints;
  }
  public void setInts(int[] ints)
  {
    this.ints = ints;
  }
  public Map getMap()
  {
    return map;
  }
```

```
    public void setMap(Map map)
    {
        this.map = map;
    }
    //使用注释语法来改变该属性序列化后的属性名
    @JSON(name="newName")
    public String getCustomName()
    {
        return this.customName;
    }
    public String execute()
    {
        map.put("name", "yeeku");
        return Action.SUCCESS;
    }
}
```

在上面代码中，使用了 JSON 注释，注释时指定了 name 域，name 域指定 Action 属性被序列化成 JSON 对象的属性名。除此之外，JSON 注释还支持如下几个域：

serialize：配置是否序列化该属性

deserialize：配置是否反序列化该属性。

format：配置用于格式化输出、解析日期表单域的格式。例如"yyyy-MM-dd'T'HH:mm:ss"。

配置该 Action 和配置普通 Action 存在小小的区别，应该为该 Action 配置类型为 json 的 Result。而这个 Result 无需配置任何视图资源。

配置该 Action 的 struts.xml 文档代码如下：

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <constant name="struts.i18n.encoding" value="UTF-8"/>
    <package name="example" extends="json-default">
        <action name="JSONExample" class="lee.JSONExample">
            <result type="json"/>
        </action>
    </package>
</struts>
```

在上面配置文档中有两个值得注意的地方：

第一个地方是配置 struts.i18n.encoding 常量时，不再是使用 GBK 编码，而是 UTF-8 编码，这是因为 Ajax 的 POST 请求都是以 UTF-8 的方式进行编码的。

第二个地方是配置包时，自己的包继承了 json-default 包，而不再继承默认的 default 包，这是因为只有在该包下才有 json 类型的 Result。