gson的使用

JSON是一种轻量级的数据交换格式。 易于人阅读和编写，同时也易于机器解析和生成(一般用于提升网络传输速率)。

# JSON 语法

## JSON 语法规则

SON 语法是 JavaScript 对象表示法语法的子集。

* 数据在名称/值对中
* 数据由逗号分隔
* 数据中键一定要用""括起来
* 花括号保存对象
* 方括号保存数组

## JSON 名称/值对

SON 数据的书写格式是：名称/值对。名称/值对包括字段名称（在双引号中），后面写一个冒号，然后是值：

|  |
| --- |
| "firstName" : "John" |

## JSON 值

JSON 值可以是：

* 数字（整数或浮点数）
* 字符串（在双引号中）
* 逻辑值（true 或 false）
* 数组（在方括号中）
* 对象（在花括号中）
* null

|  |
| --- |
| 数字：1  字符串："hello"  逻辑值：true  对象：{ "firstName":"John" , "lastName":"Doe" }  数组：{  "employees": [  { "firstName":"John" , "lastName":"Doe" },  { "firstName":"Anna" , "lastName":"Smith" },  { "firstName":"Peter" , "lastName":"Jones" }  ]  }  null值：null |

# Gson的基本用法

Gson是Google推出的用来解析json数据以及将对象转换成json数据的框架。可以很方便地实现json数据与对象的相互转换，还可以自定义需要序列化或反序列化的字段。maven依赖如下：

|  |
| --- |
| <dependency>  <groupId>com.google.code.gson</groupId>  <artifactId>gson</artifactId>  <version>2.7</version>  </dependency> |

Gson提供了fromJson() 和toJson() 两个直接用于解析和生成的方法，前者实现反序列化，后者实现了序列化。同时每个方法都提供了重载方法，我常用的总共有5个。

## 基本数据类型的解析

|  |
| --- |
| Gson gson = new Gson();  int i = gson.fromJson("100", int.class); //100  double d = gson.fromJson("\"99.99\"", double.class); //99.99  boolean b = gson.fromJson("true", boolean.class); // true  String str = gson.fromJson("String", String.class); // String |

## 基本数据类型的生成

|  |
| --- |
| String jsonNumber = gson.toJson(100); // 100  String jsonBoolean = gson.toJson(false); // false  String jsonString = gson.toJson("String"); // "String" |

## POJO类的生成与解析

|  |
| --- |
| #POJO的生成JSON  Gson gson = new Gson();  GsonBean bean = new GsonBean();  bean.setName("xxz");  bean.setAge(1);  bean.setSex(true);  List<String> list = new ArrayList<String>();  list.add("打篮球");  list.add("健身");  bean.setHobbies(list);  String jsonObject = gson.toJson(bean);  System.out.println(jsonObject); //{"name":"xxz","age":1,"sex":true,"hobbies":["打篮球","健身"]} |
| #解析JSON成POJO  Gson gson = new Gson();  String jsonString = "{\"name\":\"xxz\",\"age\":1,\"sex\":true,\"hobbies\":[\"打篮球\",\"健身\"]}";  GsonBean bean = gson.fromJson(jsonString, GsonBean.class);  System.out.println(bean); |

## Gson中使用泛型

|  |
| --- |
| #数组的解析  Gson gson = new Gson();  String jsonArray = "[\"Android\",\"Java\",\"PHP\"]";  String[] strings = gson.fromJson(jsonArray, String[].class); |
| #但对于List将上面的代码中的 String[].class 直接改为 List<String>.class 是行不通的。对于Java来说List<String> 和List<User> 这俩个的字节码文件只一个那就是List.class，这是Java泛型使用时要注意的问题 泛型擦除。  #为了解决的上面的问题，Gson为我们提供了TypeToken来实现对泛型的支持，所以当我们希望使用将以上的数据解析为List<String>时需要这样写。  Gson gson = new Gson();  String jsonArray = "[\"Android\",\"Java\",\"PHP\"]";  List<String> stringList = gson.fromJson(jsonArray, new TypeToken<List<String>>() {}.getType());  for (String s : stringList) {  System.out.println(s);  } |
| public class Result<T> {  private int code;  private String message;  private T data;  set,get...  }  Gson gson = new Gson();  Type gsonBeanType = new TypeToken<Result<GsonBean>>(){}.getType();  String json = "{\"data\":{\"name\":\"xxz\",\"age\":1,\"sex\":true,\"hobbies\":[\"打篮球\",\"健身\"]}}";  Result<GsonBean> gsonBeanResult = gson.fromJson(json,gsonBeanType);  System.out.println(gsonBeanResult.getData()); |

## 属性重命名 @SerializedName 注解的使用

从上面POJO的生成与解析可以看出json的字段和值是的名称和类型是一一对应的，但也有一定容错机制(如第一个例子第3行将字符串的99.99转成double型，但有时候也会出现一些不和谐的情况，如：期望的json格式

|  |
| --- |
| 实际：{"name":"xxz","age":1,"sex":true,"hobbies":["打篮球","健身"]}  想改变成：{"NAME":"xxz","age":1,"sex":true,"hobbies":["打篮球","健身"]}  注意：可以在字段上使用@SerializedName 注解，让字段(反)序列化指定的名字  @SerializedName("NAME")  **private** String name;  为POJO字段提供备选属性名SerializedName注解提供了两个属性，上面用到了其中一个，别外还有一个属性alternate，接收一个String数组。  @SerializedName(value = "NAME", alternate = {"name"})  **private** String name;  注：alternate需要2.4版本，当上面的属性(NAME , name)都中出现任意一个时均可以得到正确的结果。  注：当多种情况同时出时，以最后一个出现的值为准。 |

## 使用GsonBuilder导出null值、格式化输出、日期时间

|  |
| --- |
| Gson gson = new GsonBuilder()  //序列化null  .serializeNulls()  // 设置日期时间格式，另有2个重载方法  // 在序列化和反序化时均生效  .setDateFormat("yyyy-MM-dd")  // 禁此序列化内部类  .disableInnerClassSerialization()  //生成不可执行的Json（多了 )]}' 这4个字符）  .generateNonExecutableJson()  //禁止转义html标签  .disableHtmlEscaping()  //格式化输出  .setPrettyPrinting()  .create(); |

# Gson序列化和反序列化

## Gson的流式反序列化

Gson提供了fromJson()和toJson() 两个直接用于解析和生成的方法，前者实现反序列化，后者实现了序列化。同时每个方法都提供了重载方法，我常用的总共有5个。

|  |
| --- |
| Gson.toJson(Object);  Gson.fromJson(Reader,Class);  Gson.fromJson(String,Class);  Gson.fromJson(Reader,Type);  Gson.fromJson(String,Type); |
| #手动的方式就是使用stream包下的JsonReader类来手动实现反序列化。  @Test  public void test9() throws Exception {  String json = "{\"name\":\"怪盗kidou\",\"age\":\"24\"}";  GsonBean bean = new GsonBean();  JsonReader reader = new JsonReader(new StringReader(json));  reader.beginObject(); // throws IOException  while (reader.hasNext()) {  String s = reader.nextName();  switch (s) {  case "name":  bean.setName(reader.nextString());  break;  case "age":  bean.setAge(reader.nextInt()); // 自动转换  break;  case "birthDay":  bean.setBirthDay(DateUtils.parseDate(reader.nextString(), new String[] { "yyyy-MM-dd" })); // 自动转换  break;  case "sex":  bean.setSex(reader.nextBoolean()); // 自动转换  break;  }  }  reader.endObject(); // throws IOException  System.out.println(bean.getName() + "-" + bean.getAge());  } |
| 其实自动方式最终都是通过JsonReader来实现的，如果第一个参数是String类型，那么Gson会创建一个StringReader转换成流操作。 |

## Gson的流式序列化

|  |
| --- |
| 手动方式  JsonWriter writer = new JsonWriter(new OutputStreamWriter(System.out));  writer.beginObject() // throws IOException  .name("name").value("怪盗kidou")  .name("age").value(24)  .name("email").nullValue() //演示null  .endObject(); // throws IOException  writer.flush(); // throws IOException  //{"name":"怪盗kidou","age":24,"email":null} |

# Gson字段过滤

## 基于@Expose注解

@Expose提供了两个属性，且都有默认值，开发者可以根据需要设置不同的值。@Expose 注解从名字上就可以看出是暴露的意思，所以该注解是用于对处暴露字段的。可是我们以前用Gson的时候也没有@Expose 注解还是不正确的序列化为JSON了么?是的，所以该注解在使用new Gson() 时是不会发生作用。毕竟最常用的API要最简单，所以该注解必须和GsonBuilder配合使用。使用方法： 简单说来就是需要导出的字段上加上@Expose 注解，不导出的字段不加。注意是不导出的不加。

@Expose //

@Expose(deserialize = true,serialize = true) //序列化和反序列化都都生效

@Expose(deserialize = true,serialize = false) //反序列化时生效

@Expose(deserialize = false,serialize = true) //序列化时生效

@Expose(deserialize = false,serialize = false) // 和不写一样

注：根据上面的图片可以得出，所有值为false的属性都是可以不写的。

|  |
| --- |
| 在使用Gson时也不能只是简单的new Gson()了。  Gson gson = new GsonBuilder()  .excludeFieldsWithoutExposeAnnotation()  .create();  gson.toJson(jsonString); |

## 基于版本

Gson在对基于版本的字段导出提供了两个注解 @Since 和 @Until,和GsonBuilder.setVersion(Double)配合使用。@Since 和 @Until都接收一个Double值。Since和Until注解

使用方法：当前版本(GsonBuilder中设置的版本) 大于等于Since的值时该字段导出，小于Until的值时该该字段导出。

|  |
| --- |
| class SinceUntilSample {  @Since(4)  public String since;  @Until(5)  public String until;  }  public void sineUtilTest(double version){  SinceUntilSample sinceUntilSample = new SinceUntilSample();  sinceUntilSample.since = "since";  sinceUntilSample.until = "until";  Gson gson = new GsonBuilder().setVersion(version).create();  System.out.println(gson.toJson(sinceUntilSample));  }  //当version <4时，结果：{"until":"until"}  //当version >=4 && version <5时，结果：{"since":"since","until":"until"}  //当version >=5时，结果：{"since":"since"}  注：当一个字段被同时注解时，需两者同时满足条件。 |

## 基于访问修饰符

什么是修饰符? public、static 、final、private、protected 这些就是，所以这种方式也是比较特殊的。使用方式：

|  |
| --- |
| class ModifierSample {  final String finalField = "final";  static String staticField = "static";  public String publicField = "public";  protected String protectedField = "protected";  String defaultField = "default";  private String privateField = "private";  } |

使用GsonBuilder.excludeFieldsWithModifiers构建gson,支持int形的可变参数，值由java.lang.reflect.Modifier提供，下面的程序排除了privateField 、 finalField 和staticField 三个字段。

|  |
| --- |
| ModifierSample modifierSample = new ModifierSample();  Gson gson = new GsonBuilder()  .excludeFieldsWithModifiers(Modifier.FINAL, Modifier.STATIC, Modifier.PRIVATE)  .create();  System.out.println(gson.toJson(modifierSample));  // 结果：{"publicField":"public","protectedField":"protected","defaultField":"default"} |

注意：最简单的方式不序列化可以使用transient。

如：private transient Date birthDay;