**Python类对象的JSON序列化处理**

2017年07月31日 04:43:01 [杰瑞26](https://me.csdn.net/Jerry_1126) 阅读数：19415

 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/Jerry\_1126/article/details/76409042

不同的编程语言有不同的数据类型; 比如说:  
Python的数据类型有(dict、list、string、int、float、long、bool、None)  
Java的数据类型有(bool、char、byte、short、int、long、float、double)  
C的数据类型有(bit、bool、char、int、short、long、unsigned、double、float)  
Tcl的数据类型(int、bool、float、string)  
Ruby的数据类型(Number、String、Ranges、Symbols、true、false、Array、Hash)  
...  
他们的共同特点是，都有字符串类型！  
所以要实现不同的编程语言之间对象的传递，就必须把对象序列化为标准格式，比如XML，但更好的方法是序列化为JSON，因为JSON表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。

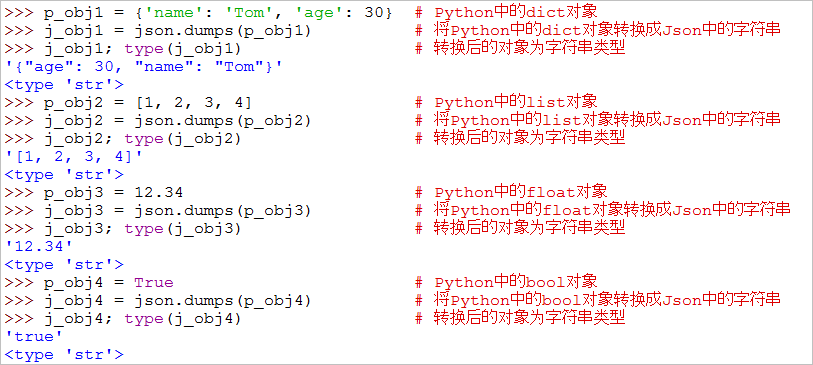
JSON不仅是标准格式，并且比XML更快，而且可以直接在Web页面中读取，非常方便.

**JSON类型            Python类型**  
{}                  dict  
[]                  list  
"string"            str  
1234.56             int或float  
true                True  
false               False  
null                None

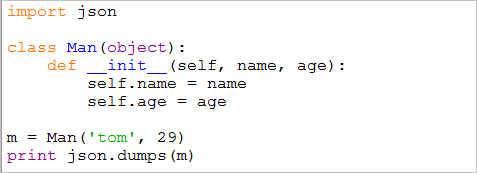
Python内置的json模块提供了非常完善的Python对象到JSON格式的转换。

**json.dumps()**         将Python中的对象转换为JSON中的字符串对象  
**json.loads()**        将JSON中的字符串对象转换为Python中的对象

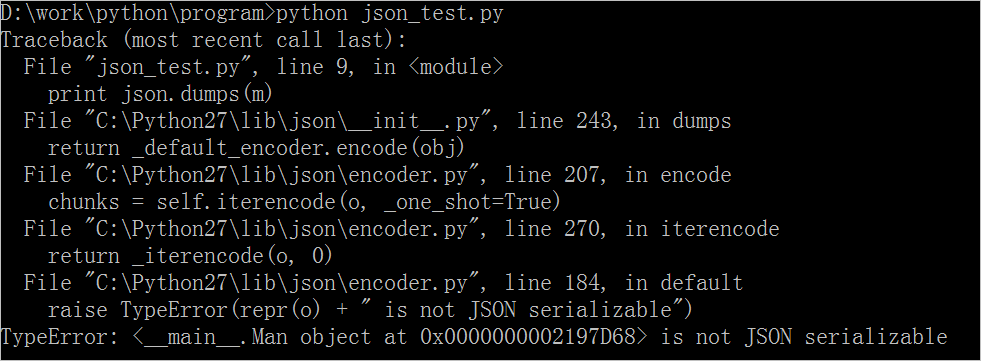
我们先看看如何把Python对象变成一个JSON,转换后的JSON对象，最后都是字符串型。



但是如果是类对象，是不是可以可以直接用json.dumps(obj)来处理呢？比如像下面一个简单类:



运行之后会报下面TypeError: 说该对象非JSON的序列化对象

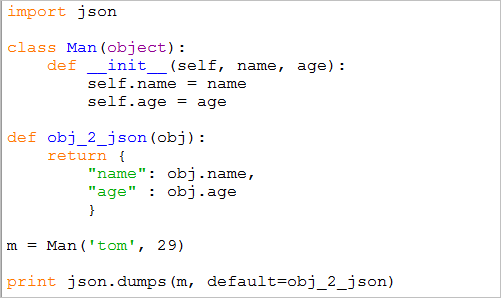


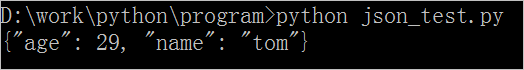
错误的原因是Man对象不是一个可序列化为JSON的对象。如果连class的实例对象都无法序列化为JSON，这肯定不合理！我们仔细看看dumps()方法的参数列表，可以发现，除了第一个必须的obj参数外，dumps()方法还提供了一大堆的可选参数：

>>> help(json.dumps)  
Help on function dumps in module json:  
  
dumps(obj, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, cls=None, indent=None, separators=None, encoding='utf-8', default=None, sort\_keys=False, \*\*kw)

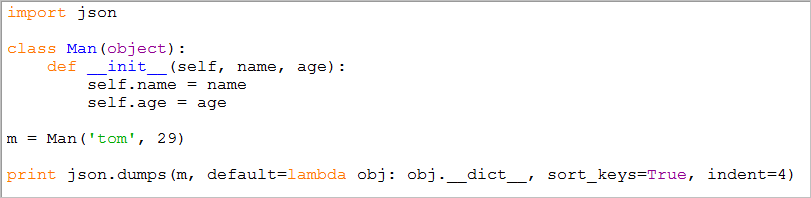
这些可选参数就是让我们来定制JSON序列化。前面的代码之所以无法把Man类实例序列化为JSON，是因为默认情况下，dumps()方法不知道如何将Man实例变为一个JSON的{}对象。

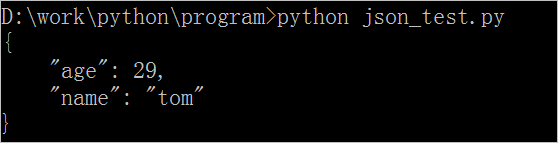
可选参数default就是把任意一个对象变成一个可序列为JSON的对象，我们只需要为Man专门写一个转换函数，再把函数传进去即可：





或者通过一种简单的方式，用lambda方式来转换任意一个类对象为JSON形式:





**lambda obj: obj.\_\_dict\_\_**          会将任意的对象，转换成字典的方式

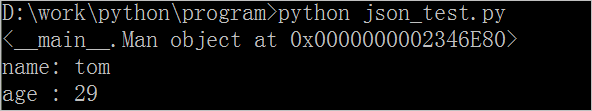
**sort\_keys=True**                    会按照字典中的键来按照ASCII方式来排序

**indent=4**                          会按照键值对以间隔4来直观的显示

同样的道理，如果要将JSON对象反序列化，也可以写个函数来转换:

**son.loads(json\_str, object\_hook=handle)**





从上面的操作，基本上就完成了从JSON字符串方式到类对象的处理.

json.dumps()函数的详细说明

>>> help(json.dumps)  
Help on function dumps in module json:  
  
dumps(obj, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, cls=None, indent=None, separators=None, encoding='utf-8', default=None, sort\_keys=False, \*\*kw)

#  函数作用:       将Python的对象转变成JSON对象

#  skipkeys:       如果为True的话，则只能是字典对象，否则会TypeError错误, 默认False

#  ensure\_ascii:   确定是否为ASCII编码

#  check\_circular: 循环类型检查，如果为True的话

#  allow\_nan:      确定是否为允许的值

#  indent:         会以美观的方式来打印，呈现

#  separators:     对象分隔符，默认为,

#  encoding:       编码方式,默认为utf-8

#  sort\_keys:      如果是字典对象，选择True的话，会按照键的ASCII码来排序

【**最后总结**】

Python语言特定的序列化模块是pickle，但如果要把序列化搞得更通用、更符合Web标准，就可以使用json模块。

json模块的dumps()和loads()函数是定义得非常好的接口的典范。当我们使用时，只需要传入一个必须的参数。但是，当默认的序列化或反序列机制不满足我们的要求时，我们又可以传入更多的参数来定制序列化或反序列化的规则，既做到了接口简单易用，又做到了充分的扩展性和灵活性

# Python实现JSON反序列化类对象

我们的网络协议一般是把数据转换成JSON之后再传输。之前在Java里面，实现序列化和反序列化，不管是jackson,还是fastjson都非常的简单。现在有项目需要用Python来开发，很自然的希望这样的便利也能在Python中体现。

但是在网上看了一些教程，讲反序列化的时候，基本都是转换为dict或者array。这种编程方式我从情感上是无法接受的。难道是这些JSON库都不支持反序列化为类对象？我马上打消了这个念头，Python这样强大的脚本语言，不可能没有完善的JSON库。

于是我就研究了一下原生的json，以及第三方的demjson和simplejson。

### 一、原生json

我仔细研究了原生json的loads方法的定义

def loads(s, encoding=None, cls=None, object\_hook=None, parse\_float=None, parse\_int=None, parse\_constant=None, object\_pairs\_hook=None, \*\*kw)

复制代码

这里面的object\_hook和object\_pairs\_hook参数引起了我的注意，我重点说一下object\_hook。

官方文档的说明如下：

object\_hook is an optional function that will be called with the result of any object literal decoded (a dict). The return value of object\_hook will be used instead of the dict. This feature can be used to implement custom decoders (e.g. JSON-RPC class hinting).

这个object\_hook根据文档的解释就是一个自定义解码函数，入参数标准反序列化后的dict，我们可以根据自己的规则转换输出为想要的格式。

我又去搜了一下object\_hook，大家对于这个东西的处理方式基本就是用一个静态方法把dict转换成对象。

我们的数据结构是这样的

{"status":1,"info":"发布成功","data":{"id":"52","feed\_id":"70"}}

复制代码

于是我就写了这样的代码：

class Response:

def \_\_init\_\_(self, status, info, data) -> None:

super().\_\_init\_\_()

self.status = status

self.info = info

self.data = data

@staticmethod

def object\_hook(d):

return Response(d['status'], d['info'], d['data'])

...

resp = json.loads(body, object\_hook=Response.object\_hook)

复制代码

一开始呢，确实没有问题，虽然用起来没有java的json库辣么方便，但总归实现了需求。

好景不长，我测试的第一个接口返回的数据中，data是字段一个字符串，反序列化正常。可是后来当接口返回的结构中data字段是一个dict结构的时候，object\_hook的入参居然变成了data字段转换之后的dict({"id":"52","feed\_id":"70"})，而不是完整的数据。

这些懵逼了，上网搜索了一圈没有结论。好吧，我最后又老老实实回到官方文档，read the fucking official document。

不看不知道，一看吓一跳，官方文档用了一种巧妙的方式实现了上面的需求。

>>> class JSONObject:

... def \_\_init\_\_(self, d):

... self.\_\_dict\_\_ = d

...

>>>

>>> data = json.loads(s, object\_hook=JSONObject)

>>> data.name

'ACME'

>>> data.shares

50

>>> data.price

490.1

>>>

复制代码

我服了，把json解析之后的dict直接赋值给对象的属性dict，然后就可以随心所欲的使用属性了，真心方便，动态语言就是好。

以上是官方的json库实现方案，那另外两个知名的第三方库呢？

### 二、demjson

demjson也支持hook。有两种配置的方式：decode函数配置和set\_hook函数配置

#### 1. decode

def decode( txt, encoding=None, \*\*kwargs )

复制代码

decode函数可以指定很多参数，其中就包括hook函数。hook函数的指定是使用键值对的方式，键是hook函数的名称，值是hook函数。

demjson是通过名字来管理hook函数的，所以hookname不是随便指定的，必须是内置的几种hook函数的名称。

* decode\_number
* decode\_float
* decode\_object
* decode\_array
* decode\_string
* encode\_value
* encode\_dict
* encode\_dict\_key
* encode\_sequence
* encode\_bytes
* encode\_default

demjson.decode(body, encode='utf-8',decode\_obbject=Reponse.object\_hook)

复制代码

结果并没有让我很开森，依然是无法处理嵌套结构。 日志中显示如下内容：

2018-01-30 16:01:17,137 poster.py post\_all 73 INFO : {"status":1,"info":"\u53d1\u5e03\u6210\u529f","data":{"id":"54","feed\_id":"72"}}

2018-01-30 16:01:17,138 response.py object\_hook 13 INFO : {'id': '54', 'feed\_id': '72'}

2018-01-30 16:01:17,138 response.py object\_hook 13 INFO : {'status': 1, 'info': '发布成功', 'data': demjson.undefined}

复制代码

很奇怪的是object\_hook 函数被调用了两次，第一次是data字段的内容，第二是全部的内容，但是data字段没有解析出来。 非常奇怪，百思不得其解！！！

#### 2. set\_hook

set\_hook函数跟上面的decode函数不一样，它是JSON类的成员函数，而decode函数是个静态函数。

def set\_hook(self, hookname, **function**)

复制代码

吸取之前的教训，这次我仔细阅读了demjson的文档，还真发现点东西。

Netsted values. When decoding JSON that has nested objects or arrays, the decoding hooks will be called once for every corresponding value, even if nested. Generally the decoding hooks will be called from the inner-most value outward, and then left to right.

这里重点说到嵌套的问题，出现嵌套的时候，每个对应的类型都会调用hook函数一次，而且是从最内层，从左往右。好吧，之前出现的问题全部明白了，原来都是这个规则惹的祸，但是为什么这样设计我暂时还是不明白。

set\_hook的使用方式

j = demjson.JSON()

j.set\_hook( 'decode\_array', my\_sort\_array )

j.decode(body, encode='utf-8')

复制代码

### 三、simplejson

前面说了那么多，simplejson的方式就没什么可说的，跟官方的json库hook方式一致。

## 总结

虽然我的需求是满足了，但是还是有一个大大的问号留在我心中，为什么是这样设计，网上没有找到合适的答案，剩下的需要研究源代码分析了。