JSON风格指南

版本：0.9

英文版：http://google-styleguide.googlecode.com/svn/trunk/jsoncstyleguide.xml

翻译：Darcy Liu

状态：草稿

简介

该风格指南是对在Google创建JSON APIs而提供的指导性准则和建议。总体来讲，JSON APIs应遵循JSON.org上的规范。这份风格指南澄清和标准化了特定情况，从而使Google的JSON APIs有一种标准的外观和感觉。这些指南适用于基于RPC和基于REST风格的API的JSON请求和响应。

定义

为了更好地实现这份风格指南的目的，下面几项需要说明：

- 属性(property) - JSON对象内的键值对(name/value pair)
- 属性名(property name) - 属性的名称(或键)
- 属性值(property value) - 分配给属性的值

示例：

```
{
  // 一组键值对称作一个 "属性".
  "propertyName": "propertyValue"
}
```

Javascript的数字(*number*)包含所有的浮点数,这是一个宽泛的指定。在这份指南中，数字(*number*)指代Javascript中的数字(*number*)类型，而整型(*integer*)则指代整型。

一般准则

注释

JSON对象中不包含注释。

JSON对象中不应该包含注释。该指南中的某些示例含有注释。但这仅仅是为了说明示例。

```
{
  // 你可能在下面的示例中看到注释,
  // 但不要在你的JSON数据中加入注释.
  "propertyName": "propertyValue"
}
```

双引号

使用双引号

如果（某个）属性需要引号，则必须使用双引号。所有的属性名必须在双引号内。字符类型的属性值必须使用双引号。其它类型值（如布尔或数字）不应该使用双引号。

扁平化数据 VS 结构层次

不能为了方便而将数据任意分组

JSON中的数据元素应以*扁平化*方式呈现。不能为了方便而将数据任意分组。

在某些情况下，比如描述单一结构的一批属性，因为它被用来保持结构层次，因而是有意义的。但是遇到这些情况还是应当慎重考虑，记住只有语义上有意义的时候才使用它。例如，一个地址可以有表示两种方式，但结构化的方式对开发人员来讲可能更有意义：

扁平化地址:

```
{
  "company": "Google",
  "website": "http://www.google.com/",
  "addressLine1": "111 8th Ave",
  "addressLine2": "4th Floor",
  "state": "NY",
  "city": "New York",
  "zip": "10011"
}
```

结构化地址：

```
{
  "company": "Google",
  "website": "http://www.google.com/",
  "address": {
    "line1": "111 8th Ave",
    "line2": "4th Floor",
    "state": "NY",
    "city": "New York",
    "zip": "10011"
  }
}
```

属性名准则

属性名格式

选择有意义的属性名

属性名必须遵循以下准则:

- 属性名应该是具有定义语义的有意义的名称。
- 属性名必须是驼峰式的，ASCII码字符串。
- 首字符必须式字母，下划线(_)或美元符号(*$*)。
- 随后的其他字符可以是字母，数字，下划线(_)或美元符号(*$*)。
- 应该避免使用Javascript中的保留关键字(下文附有Javascript保留字清单)

这些准则反映JavaScript标识符命名的指导方针。使JavaScript的客户端可以使用点符号来访问属性。(例如, `result.thisIsAnInstanceVariable`).

下面是一个对象的一个属性的例子:

```
{
  "thisPropertyIsAnIdentifier": "identifier value"
}
```

JSON Map中的键名

在JSON Map中键名可以使用任意Unicode字符

当JSON对象作为Map(映射)使用时，属性的名称命名规则并不适用。Map（也称作关联数组）是一个具有任意键/值对的数据类型，这些键/值对通过特定的键来访问相应的值。JSON对象和JSON Map在运行时看起来是一样的；这个特性与API设计相关。当JSON对象被当作map使用时，API文件应当做出说明。

Map的键名不一定要遵循属性名称的命名准则。键名可以包含任意的Uincode字符。客户端可使用maps熟悉的方括号来访问这些属性。（例如`result.thumbnails["72"]`）

```
{
  // "address" 属性是一个子对象
  // 包含地址的各部分.
  "address": {
    "addressLine1": "123 Anystreet",
    "city": "Anytown",
    "state": "XX",
    "zip": "00000"
  },
  // "address" 是一个映射
  // 含有响应规格所对应的URL，用来映射thumbnail url的像素规格
  "thumbnails": {
    "72": "http://url.to.72px.thumbnail",
    "144": "http://url.to.144px.thumbnail"
  }
}
```

保留的属性名称

某些属性名称会被保留以便能在多个服务间相容使用

保留属性名称的详细信息，连同完整的列表，可在本指南后面的内容中找到。服务应按照被定义的语义来使用属性名称。

单数属性名 VS 复数属性名

数组类型应该是复数属性名。其它属性名都应该是单数。

数组通常包含多个条目，复数属性名就反映了这点。在下面这个保留名称中可以看到例子。属性名*items*是复数因为它描述的是一组对象。大多数的其它字段是单数。

当然也有例外，尤其是涉及到数字的属性值的时候。例如，在保留属性名中，*totalItems*比*totalItem*更合理。然后，从技术上讲，这并不违反风格指南，因为*totalItems*可以被看作*totalOfItems*,其中*total*是单数（依照风格指南），*OfItems*用来限定总数。字段名也可被改

为*itemCount*，这样看起来更象单数.

```
{
  // 单数
  "author": "lisa",
  // 一组同胞, 复数
  "siblings": [ "bart", "maggie"],
  // "totalItem" 看起来并不对
  "totalItems": 10,
  // 但 "itemCount" 要好些
  "itemCount": 10,
}
```

命名冲突

通过选择新的属性名或将API版本化来避免命名冲突

新的属性可在将来被添加进保留列表中。JSON中不存在命名空间。如果存在命名冲突，可通过选择新的属性名或者版本化来解决这个问题。例如，假设我们由下面的JSON对象开始：

```
{
  "apiVersion": "1.0",
  "data": {
    "recipeName": "pizza",
    "ingredients": ["tomatoes", "cheese", "sausage"]
  }
}
```

如果我们希望将来把*ingredients*列为保留字，我们可以通过下面两件事情来达成。

1.选一个不同的名字

```
{
  "apiVersion": "1.0",
  "data": {
    "recipeName": "pizza",
    "ingredientsData": "Some new property",
    "ingredients": ["tomatoes", "cheese", "sausage"]
  }
}
```

2.在主版本上重新命名属性

```
{
  "apiVersion": "2.0",
  "data": {
    "recipeName": "pizza",
    "ingredients": "Some new property",
    "recipeIngredients": ["tomatos", "cheese", "sausage"]
  }
}
```

属性值准则

属性值格式

属性值必须是Unicode 的 booleans（布尔）, 数字(numbers), 字符串(strings), 对象(objects), 数组(arrays), 或 null.

JSON.org上的标准准确的说明了哪些类型的数据可以作为属性值。这包含Unicode的布尔(booleans), 数字(numbers), 字符串(strings), 对象(objects), 数组(arrays), 或 null。JavaScript表达式是不被接受的。APIs应该支持该准则，并为某个特定的属性选择最合适的数据类型（比如，用numbers代表numbers等）。

好的例子:

```
{
  "canPigsFly": null,     // null
  "areWeThereYet": false, // boolean
  "answerToLife": 42,     // number
  "name": "Bart",         // string
  "moreData": {},         // object
  "things": []            // array
}
```

不好的例子:

```
{
  "aVariableName": aVariableName,         // Bad - JavaScript 标识符
  "functionFoo": function() { return 1; } // Bad - JavaScript 函数
}
```

空或Null 属性值

考虑移除空或null值

如果一个属性是可选的或者包含空值或*null*值，考虑从JSON中去掉该属性，除非它的存在有很强的语义原因。

```
{
  "volume": 10,

  // 即使 "balance" 属性值是零，它也应当被保留，
  // 因为 "0" 表示 "均衡"
  // "-1" 表示左倾斜和"+1" 表示右倾斜
  "balance": 0,

  // "currentlyPlaying" 是null的时候可被移除
  // "currentlyPlaying": null
}
```

枚举值

枚举值应当以字符串的形式呈现

随着APIs的发展，枚举值可能被添加，移除或者改变。将枚举值当作字符串可以使下游用户幽雅的处理枚举值的变更。

Java代码：

```
public enum Color {
  WHITE,
  BLACK,
  RED,
  YELLOW,
  BLUE
}
```

JSON对象：

```
{
  "color": "WHITE"
}
```

属性值数据类型

上面提到，属性值必须是布尔(booleans), 数字(numbers), 字符串(strings), 对象(objects), 数组(arrays), 或 null. 然而在处理某些值时，定义一组标准的数据类型是非常有用的。这些数据类型必须始终是字符串，但是为了便于解析，它们也会以特定的方式被格式化。

日期属性值

日期应该使用**RFC3339**建议的格式

日期应该是RFC 3339所建议的字符串格式。

```
{
  "lastUpdate": "2007-11-06T16:34:41.000Z"
}
```

时间间隔属性值

时间间隔应该使用**ISO 8601**建议的格式

时间间隔应该是ISO 8601所建议的字符串格式。

```
{
  // 三年, 6个月, 4天, 12小时,
  // 三十分钟, 5秒
  "duration": "P3Y6M4DT12H30M5S"
}
```

纬度/经度属性值

纬度/经度应该使用**ISO 6709**建议的格式

纬度/经度应该是ISO 6709所建议的字符串格式。 而且, 它应该更偏好使用 e Â±DD.DDDDÂ ±DDD.DDDD 角度格式.

```
{
  // 自由女神像的纬度/经度位置.
  "statueOfLiberty": "+40.6894-074.0447"
}
```

JSON结构和保留属性名

为了使APIs保持一致的借口，JSON对象应当使用以下的结构。该结构适用于JSON的请求和响应。在这个结构中，某些属性名将被保留用作特殊用途。这些属性并不是必需的，也就是说，每个保留的属性可能出现零次或一次。但是如果服务需要这些属性，建议遵循该命名条约。下面是一份JSON结构语义表，以Orderly格式呈现(现在可以被纳入 JSONSchema)。你可以在该指南的最后找到关于JSON结构的例子。

```
object {
  string apiVersion?;
  string context?;
  string id?;
  string method?;
  object {
    string id?
  }* params?;
  object {
    string kind?;
    string fields?;
    string etag?;
    string id?;
    string lang?;
    string updated?; # date formatted RFC 3339
    boolean deleted?;
    integer currentItemCount?;
    integer itemsPerPage?;
    integer startIndex?;
    integer totalItems?;
    integer pageIndex?;
    integer totalPages?;
    string pageLinkTemplate /^https?:/ ?;
    object {}* next?;
    string nextLink?;
    object {}* previous?;
    string previousLink?;
    object {}* self?;
    string selfLink?;
    object {}* edit?;
    string editLink?;
    array [
      object {}*;
    ] items?;
```

```
    ] items?;
  }* data?;
  object {
    integer code?;
    string message?;
    array [
      object {
        string domain?;
        string reason?;
        string message?;
        string location?;
        string locationType?;
        string extendedHelp?;
        string sendReport?;
      }*;
    ] errors?;
  }* error?;
}*;
```

JSON对象有一些顶级属性，然后是*data*对象或*error*对象，这两者不会同时出现。下面是这些属性的解释。

顶级保留属性名称

顶级的JSON对象可能包含下面这些属性

apiVersion

```
属性值类型: 字符串(string)
父节点: -
```

呈现请求中服务API期望的版本，以及在响应中保存的服务API版本。应随时提供*apiVersion*。这与数据的版本无关。将数据版本化应该通过其他的机制来处理，如etag。

示例:

```
{ "apiVersion": "2.1" }
```

context

```
属性值类型: 字符串(string)
父节点: -
```

客户端设置这个值，服务器通过数据作出回应。这在JSON-P和批处理中很有用，用户可以使用*context*将响应与请求关联起来。该属性是顶级属性，因为不管响应是成功还是有错误，*context*总应当被呈现出来。*context*不同于*id*在于*context*由用户提供而*id*由服务分配。

示例:

请求 #1:

```
http://www.google.com/myapi?context=bart
```

请求 #2:

```
http://www.google.com/myapi?context=lisa
```

响应 #1:

```
{
  "context": "bart",
  "data": {
    "items": []
  }
}
```

响应 #2:

```
{
  "context": "lisa",
  "data": {
    "items": []
  }
}
```

公共的JavaScript处理器通过编码同时处理以下两个响应:

```
function handleResponse(response) {
  if (response.result.context == "bart") {
    // 更新页面中的 "Bart" 部分。
  } else if (response.result.context == "lisa") {
    // 更新页面中的 "Lisa" 部分。
  }
}
```

id

```
属性值类型: 字符串(string)
父节点: -
```

服务提供用于识别响应的标识(无论请求是成功还是有错误)。这对于将服务日志和单独收到的响应对应起来很有用。

示例:

```
{ "id": "1" }
```

method

```
属性值类型: 字符串(string)
父节点: -
```

表示对数据即将执行，或已被执行的操作。在JSON请求的情况下，*method*属性可以用来指明对数据进行何种操作。在JSON响应的情况下，*method*属性表明对数据进行了何种操作。

一个JSON-RPC请求的例子，其中*method*属性表示要在*params*上执行的操作:

```
{
  "method": "people.get",
  "params": {
    "userId": "@me",
    "groupId": "@self"
  }
}
```

params

```
属性值类型: 对象(object)
父节点: -
```

这个对象作为输入参数的映射发送给RPC请求。它可以和*method*属性一起用来执行RPC功能。若RPC方法不需要参数，则可以省略该属性。

示例:

```
{
  "method": "people.get",
  "params": {
    "userId": "@me",
    "groupId": "@self"
  }
}
```

data

```
属性值类型: 对象(object)
父节点: -
```

包含响应的所有数据。该属性本身拥有许多保留属性名，下面会有相应的说明。服务可以自由地将自己的数据添加到这个对象。一个JSON响应要么应当包含一个*data*对象，要么应当包含*error*对象，但不能两者都包含。如果*data*和*error*同时出现，则*error*对象优先。

error

```
属性值类型: 对象(object)
父节点: -
```

表明错误发生，提供错误的详细信息。错误的格式支持从服务返回一个或多个错误。一个JSON响应可以有一个*data*对象或者一个*error*对象，但不能两者都包含。如果*data*和*error*都出现，*error*对象优先。

示例:

```
{
  "apiVersion": "2.0",
  "error": {
    "code": 404,
    "message": "File Not Found",
    "errors": [{
      "domain": "Calendar",
      "reason": "ResourceNotFoundException",
      "message": "File Not Found
    }]
  }
}
```

---

data对象的保留属性名

JSON对象的*data*属性可能包含以下属性。

data.kind

```
属性值类型: 字符串(sting)
父节点: data
```

*kind*属性是对某个特定的对象存储何种类型的信息的指南。可以把它放在*data*层次，或*items*的层次，或其它任何有助于区分各类对象的对象中。如果*kind*对象被提供，它应该是对象的第一个属性（详见下面的*属性顺序*部分）。

示例：

```
// "Kind" indicates an "album" in the Picasa API.
{"data": {"kind": "album"}}
```

data.fields

```
属性值类型: 字符串(string)
父节点: data
```

表示做了部分GET之后响应中出现的字段，或做了部分PATCH之后出现在请求中的字段。该属性仅在做了部分GET请求/批处理时存在，且不能为空。

示例：

```
{
  "data": {
    "kind": "user",
    "fields": "author,id",
    "id": "bart",
    "author": "Bart"
  }
}
```

data.etag

```
属性值类型：字符串(string)
父节点：data
```

响应时提供etag。关于GData APIs中的ETags详情可以在这里找到：http://code.google.com/apis/gdata/docs/2.0/reference.html#ResourceVersioning

示例：

```
{"data": {"etag": "W/"C0QBRXcycSp7ImA9WxRVFUk.""}}
```

data.id

```
属性值类型：字符串(string)
父节点：data
```

一个全局唯一标识符用于引用该对象。*id*属性的具体细节都留给了服务。

示例：

```
{"data": {"id": "12345"}}
```

data.lang

```
属性值类型：字符串(string)(格式由BCP 47指定)
父节点：data (或任何子元素)
```

表示该对象内其他属性的语言。该属性模拟HTML的*lang*属性和XML的*xml:lang*属性。值应该时BCP 47中定义的一种语言值。如果一个单一的JSON对象包含的数据有多种语言，服务负责制定和标明

的lang属性的适当位置。

示例:

```
{"data": {
  "items": [
    { "lang": "en",
      "title": "Hello world!" },
    { "lang": "fr",
      "title": "Bonjour monde!" }
  ]}
}
```

data.updated

```
属性值类型: 字符串(string)(格式由RFC 3339指定)
父节点: data
```

指明条目更新的最后日期/时间(RFC 3339)，由服务规定。

示例:

```
{"data": {"updated": "2007-11-06T16:34:41.000Z"}}
```

data.deleted

```
属性值类型: 布尔(boolean)
父节点: data (或任何子元素)
```

一个标记元素，当出现时，表示包含的条目已被删除。如果提供了删除属性，它的值必须为*true*; 为*false*会导致混乱，应该避免。

示例:

```
{"data": {
  "items": [
    { "title": "A deleted entry",
      "deleted": true
    }
  ]}
}
```

data.items

```
属性值类型: 数组(array)
父节点: data
```

属性名*items*被保留用作表示一组条目(例如,Picasa中的图片，YouTube中的视频)。这种结构的目的是给与当前结果相关的集合提供一个标准位置。例如，知道页面上的*items*是数组，JSON输出便可能插入一个通用的分页系统。如果*items*存在，它应该是*data*对象的最后一个属性。（详见下面的*属性顺序*部分）。

示例:

```
{
  "data": {
    "items": [
      { /* Object #1 */ },
      { /* Object #2 */ },
      ...
    ]
  }
}
```

用于分页的保留属性名

下面的属性位于*data*对象中，用来给一列数据分页。一些语言和概念是从OpenSearch规范中借鉴过来的。

下面的分页数据允许各种风格的分页，包括:

- 上一页/下一页 - 允许用户在列表中前进和后退，一次一页。*nextLink* 和*previousLink*属性 (下面的"链接保留属性名"部分有描述) 用于这种风格的分页。
- 基于索引的分页 - 允许用户直接跳到条目列表的某个条目位置。例如，要从第200个条目开始载入10个新的条目，开发者可以给用户提供一个URL的查询字符串*?startIndex=200*。
- 基于页面的分页 - 允许用户直接跳到条目内的具体页。这跟基于索引的分页很类似,但节省了开发者额外的步骤，不需再为新一页的条目计算条目索引。例如，开发人员可以直接跳到第20页，而不是跳到第200条条目。基于页面分页的网址，可以使用查询字符串*?page=1*或*?page=20*。*pageIndex*和 *totalPages* 属性用作这种风格的分页.

在这份指南的最后可以找到如何使用这些属性来实现分页的例子。

data.currentItemCount

```
属性值类型: 整数(integer)
父节点: data
```

结果集中的条目数目。应该与items.length相等，并作为一个便利属性提供。例如，假设开发者请求一组搜索条目，并且要求每页10条。查询集共有14条。第一个条目页将会有10个条目，因此*itemsPerPage*和*currentItemCount*都应该等于10。下一页的条目还剩下4条；*itemsPerPage*仍然是10,但是*currentItemCount*是4.

示例：

```
{
  "data": {
    // "itemsPerPage" 不需要与 "currentItemCount" 匹配
    "itemsPerPage": 10,
    "currentItemCount": 4
  }
}
```

data.itemsPerPage

```
属性值类型: 整数(integer)
父节点: data
```

items结果的数目。未必是data.items数组的大小；如果我们查看的是最后一页，data.items的大小可能小于*itemsPerPage*。但是，data.items的大小不应超过*itemsPerPage*。

示例：

```
{
  "data": {
    "itemsPerPage": 10
  }
}
```

data.startIndex

```
属性值类型: 整数(integer)
父节点: data
```

data.items中第一个条目的索引。为了一致，*startIndex*应从1开始。例如，第一组items中第一条的*startIndex*应该是1。如果用户请求下一组数据，*startIndex*可能是10。

示例：

```
{
  "data": {
    "startIndex": 1
  }
}
```

data.totalItemsx

```
属性值类型：整数(integer)
父节点：data
```

当前集合中可用的总条目数。例如，如果用户有100篇博客文章，响应可能只包含10篇，但是*totalItems*应该是100。

示例：

```
{
  "data": {
    "totalItems": 100
  }
}
```

data.pagingLinkTemplate

```
属性值类型：字符串(string)
父节点：data
```

URL模板指出用户可以如何计算随后的分页链接。URL模板中也包含一些保留变量名：表示要载入的条目的*{index}*，和要载入的页面的*{pageIndex}*。

示例：

```
{
  "data": {
    "pagingLinkTemplate": "http://www.google.com/search/hl=en&q=chicago+style+
  }
}
```

data.pageIndex

```
属性值类型: 整数(integer)
父节点: data
```

条目的当前页索引。为了一致，*pageIndex*应从1开始。例如，第一页的*pageIndex*是1。*pageIndex*也可以通过基于条目的分页而计算出来*pageIndex = floor(startIndex / itemsPerPage) + 1*。

示例:

```
{
  "data": {
    "pageIndex": 1
  }
}
```

data.totalPages

```
属性值类型: 整数(integer)
父节点: data
```

当前结果集中的总页数。*totalPages*也可以通过上面基于条目的分页属性计算出来: *totalPages = ceiling(totalItems / itemsPerPage).*。

示例:

```
{
  "data": {
    "totalPages": 50
  }
}
```

用于链接的保留属性名

下面的属性位于*data*对象中，用来表示对其他资源的引用。有两种形式的链接属性：1）对象，它可以包含任何种类的引用（比如JSON-RPC对象），2)URL字符串，表示资源的URIs(后缀总为'Link')。

data.self / data.selfLink

```
属性值类型: 对象(object)/字符串(string)
父节点: data
```

自身链接可以用于取回条目数据。比如，在用户的Picasa相册中，条目中的每个相册对象都会包含一个*selfLink*用于检索这个相册的相关数据。

示例:

```
{
  "data": {
    "self": { },
    "selfLink": "http://www.google.com/feeds/album/1234"
  }
}
```

data.edit / data.editLink

```
属性值类型: 对象(object)/字符串(string)
父节点: data
```

编辑链接表明用户可以发送更新或删除请求。这对于REST风格的APIs很有用。该链接仅在用户能够更新和删除该条目时提供。

示例:

```
{
  "data": {
    "edit": { },
    "editLink": "http://www.google.com/feeds/album/1234/edit"
  }
}
```

data.next / data.nextLink

```
属性值类型: 对象(object)/字符串(string)
父节点: data
```

该下一页链接标明如何取得更多数据。它指明载入下一组数据的位置。它可以同*itemsPerPage*，*startIndex* 和 *totalItems* 属性一起使用用于分页数据。

示例：

```
{
  "data": {
    "next": { },
    "nextLink": "http://www.google.com/feeds/album/1234/next"
  }
}
```

data.previous / data.previousLink

```
属性值类型: 对象(object)/字符串(string)
父节点: data
```

该上一页链接标明如何取得更多数据。它指明载入上一组数据的位置。它可以连同*itemsPerPage*，*startIndex* 和 *totalItems* 属性用于分页数据。

示例：

```
{
  "data": {
    "previous": { },
    "previousLink": "http://www.google.com/feeds/album/1234/next"
  }
}
```

错误对象中的保留属性名

JSON对象的*error*属性应包含以下属性。

error.code

```
属性值类型: 整数(integer)
父节点: error
```

表示该错误的编号。这个属性通常表示HTTP响应码。如果存在多个错误，*code*应为第一个出错的错误码。

示例:

```
{
  "error":{
    "code": 404
  }
}
```

error.message

```
属性值类型: 字符串(string)
父节点: error
```

一个人类可读的信息，提供有关错误的详细信息。如果存在多个错误，*message*应为第一个错误的错误信息。

示例:

```
{
  "error":{
    "message": "File Not Found"
  }
}
```

error.errors

```
属性值类型: 数组(array)
父节点: error
```

包含关于错误的附加信息。如果服务返回多个错误。*errors*数组中的每个元素表示一个不同的错误。

示例:

```
{ "error": { "errors": [] } }
```

error.errors[].domain

```
属性值类型: 字符串(string)
父节点: error.errors
```

服务抛出该错误的唯一识别符。它帮助区分服务的从普通协议错误(如,找不到文件)中区分出具体错误(例如，给日历插入事件的错误)。

示例:

```
{
  "error":{
    "errors": [{"domain": "Calendar"}]
  }
}
```

error.errors[].reason

```
属性值类型: 字符串(string)
父节点: error.errors
```

该错误的唯一识别符。不同于*error.code*属性，它不是HTTP响应码。

示例:

```
{
  "error":{
    "errors": [{"reason": "ResourceNotFoundException"}]
  }
}
```

error.errors[].message

```
属性值类型: 字符串(string)
父节点: error.errors
```

一个人类可读的信息，提供有关错误的更多细节。如果只有一个错误，该字段应该与*error.message*匹配。

示例:

```
{
  "error":{
    "code": 404
    "message": "File Not Found",
    "errors": [{"message": "File Not Found"}]
  }
}
```

error.errors[].location

```
属性值类型: 字符串(string)
父节点: error.errors
```

错误发生的位置（根据*locationType*字段解释该值）。

示例:

```
{
  "error":{
    "errors": [{"location": ""}]
  }
}
```

error.errors[].locationType

```
属性值类型: 字符串(string)
父节点: error.errors
```

标明如何解释*location*属性。

示例:

```
{
  "error":{
    "errors": [{"locationType": ""}]
  }
}
```

error.errors[].extendedHelp

```
属性值类型: 字符串(string)
父节点: error.errors
```

help text的URI，使错误更易于理解。

示例：（注：原示例这里有笔误，中文版这里做了校正）

```
{
  "error":{
    "errors": [{"extendedHelp": "http://url.to.more.details.example.com/"}]
  }
}
```

error.errors[].sendReport

```
属性值类型: 字符串(string)
父节点: error.errors
```

report form的URI，服务用它来收集错误状态的数据。该URL会预先载入描述请求的参数

示例：

```
{
  "error":{
    "errors": [{"sendReport": "http://report.example.com/"}]
  }
}
```

属性顺序

在JSON对象中属性可有任意顺序。然而，在某些情况下，有序的属性可以帮助分析器快速解释数据，并带来更好的性能。在移动环境下的解析器就是个例子，在这种情况下，性能和内存是至关重要的，不必要的解析也应尽量避免。

Kind属性

Kind属性应为第一属性

假设一个解析器负责将一个原始JSON流解析成一个特定的对象。*kind*属性会引导解析器将适合的对

象实例化。因而它应该是JSON对象的第一个属性。这仅适用于对象有一个kind属性的情况(通常可以在*data*和*items*属性中找到)。

Items属性

***items*应该是*data*对象的最后一个属性**

这使得阅读每一个具体条目前前已读所有的集合属性。在有很多条目的情况下，这样就避免了开发人员只需要从数据的字段时不必要的解析这些条目。

这让阅读所有集合属性先于阅读单个条目。如遇多个条目的情况，当开发者仅需要数据中的字段时，这就可避免解析不必要的条目。

属性顺序示例:

```
// "kind" 属性区分 "album" 和 "photo".
// "Kind" 始终是它父对象的第一个属性.
// "items" 属性是 "data" 对象的最后一个属性.
{
  "data": {
    "kind": "album",
    "title": "My Photo Album",
    "description": "An album in the user's account",
    "items": [
      {
        "kind": "photo",
        "title": "My First Photo"
      }
    ]
  }
}
```

示例

YouTube JSON API

这是YouTube JSON API响应对象的示例。你可以从中学到更多关于YouTube JSON API的内容：http://code.google.com/apis/youtube/2.0/developers_guide_jsonc.html

```
{
  "apiVersion": "2.0",
```

```
"data": {
  "updated": "2010-02-04T19:29:54.001Z",
  "totalItems": 6741,
  "startIndex": 1,
  "itemsPerPage": 1,
  "items": [
    {
      "id": "BGODurRfVv4",
      "uploaded": "2009-11-17T20:10:06.000Z",
      "updated": "2010-02-04T06:25:57.000Z",
      "uploader": "docchat",
      "category": "Animals",
      "title": "From service dog to SURFice dog",
      "description": "Surf dog Ricochets inspirational video ...",
      "tags": [
        "Surf dog",
        "dog surfing",
        "dog",
        "golden retriever",
      ],
      "thumbnail": {
        "default": "http://i.ytimg.com/vi/BGODurRfVv4/default.jpg",
        "hqDefault": "http://i.ytimg.com/vi/BGODurRfVv4/hqdefault.jpg"
      },
      "player": {
        "default": "http://www.youtube.com/watch?v=BGODurRfVv4&feature=youtu
        "mobile": "http://m.youtube.com/details?v=BGODurRfVv4"
      },
      "content": {
        "1": "rtsp://v5.cache6.c.youtube.com/CiILENy73wIaGQn-Vl-0uoNjBBMYDSA
        "5": "http://www.youtube.com/v/BGODurRfVv4?f=videos&app=youtube_gdat
        "6": "rtsp://v7.cache7.c.youtube.com/CiILENy73wIaGQn-Vl-0uoNjBBMYESA
      },
      "duration": 315,
      "rating": 4.96,
      "ratingCount": 2043,
      "viewCount": 1781691,
      "favoriteCount": 3363,
      "commentCount": 1007,
      "commentsAllowed": true
    }
  ]
}
```

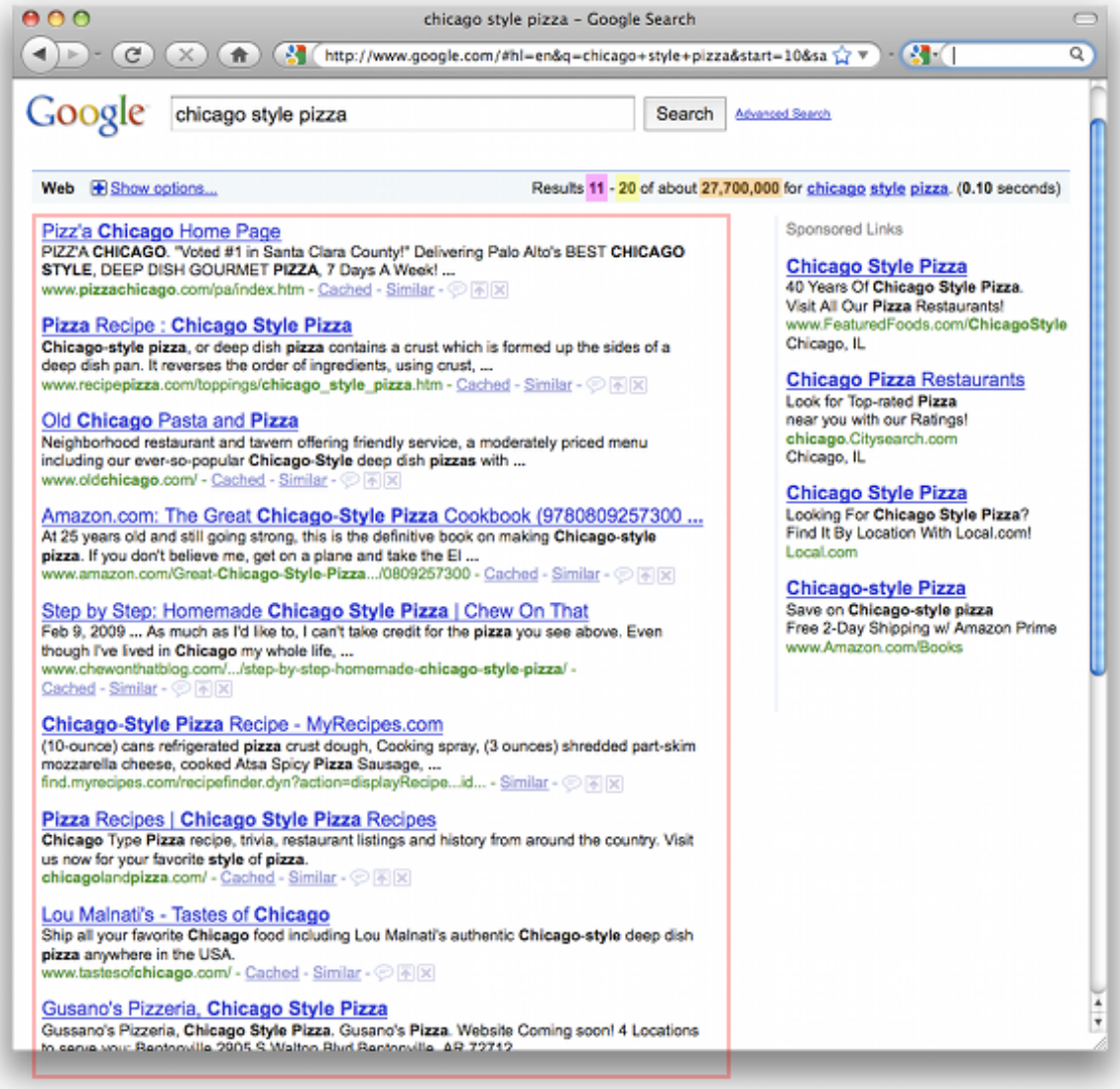
```
  }
}
```

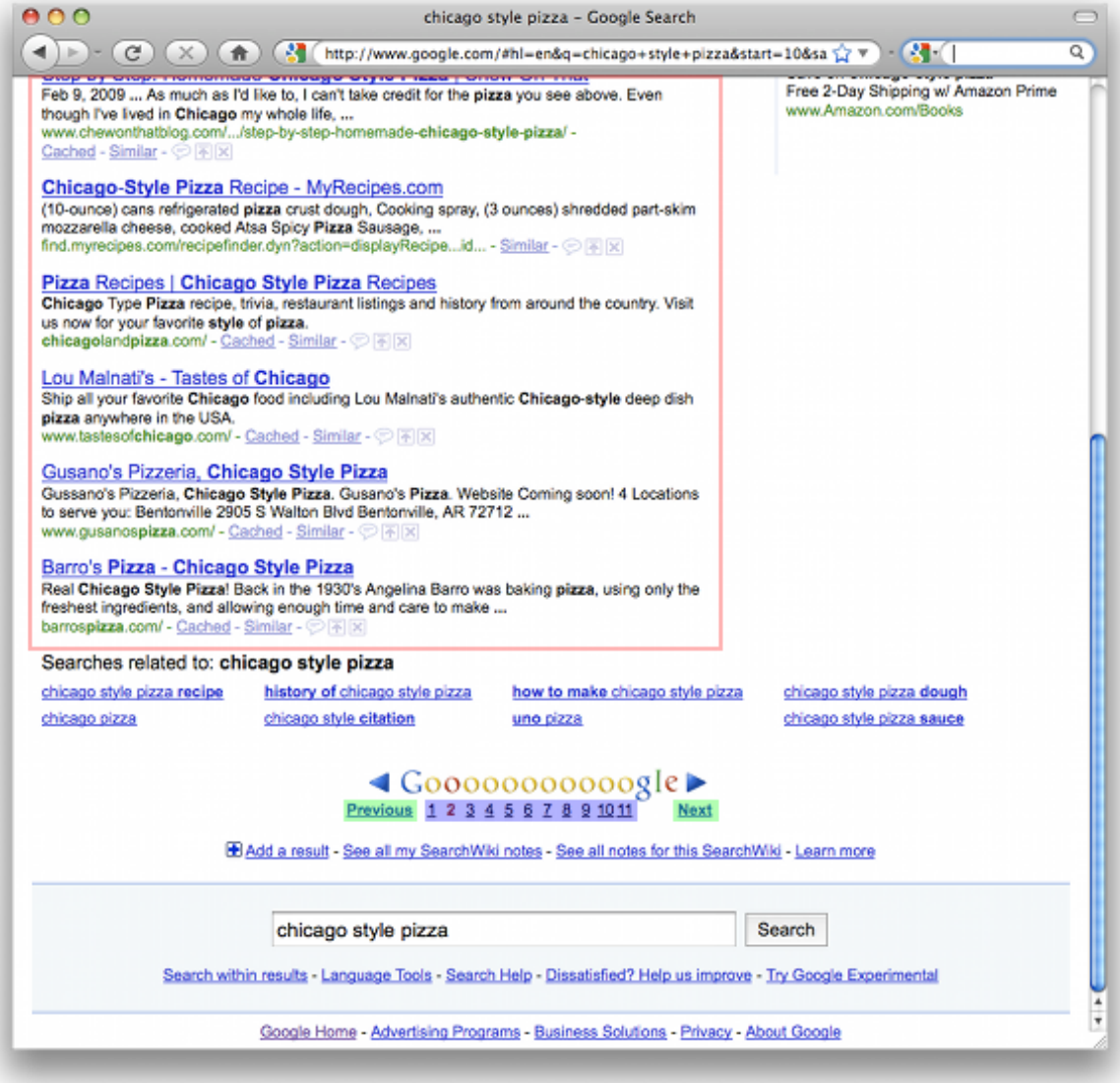
分页示例

如何将Google搜索条目作为JSON对象展现出来，对分页变量也有特别关注。

这个示例仅用作说明。下面的API实际上并不存在。

这是Google搜索结果页面的示例：





这是该页面JSON形式的呈现：

```
{
  "apiVersion": "2.1",
  "id": "1",
  "data": {
    "query": "chicago style pizza",
    "time": "0.1",
    "currentItemCount": 10,
    "itemsPerPage": 10,
    "startIndex": 11,
    "totalItems": 2700000,
    "nextLink": "http://www.google.com/search?hl=en&q=chicago+style+pizza&star
    "previousLink": "http://www.google.com/search?hl=en&q=chicago+style+pizza&
    "pagingLinkTemplate": "http://www.google.com/search/hl=en&q=chicago+style+
    "items": [
      {
        "title": "Pizz'a Chicago Home Page"
        // More fields for the search results
      }
      // More search results
    ]
  }
}
```

这是如何展现屏幕截图中的色块的例子（背景颜色对应下图中的颜色）

- Results 11 - 20 of about 2,700,000 = startIndex
- Results 11 - 20 of about 2,700,000 = startIndex + currentItemCount - 1
- Results 11 - 20 of about 2,700,000 = totalItems
- Search results = items (formatted appropriately)
- Previous/Next = previousLink / nextLink
- Numbered links in “Goooooooooooogle” = Derived from “pageLinkTemplate”. The developer is responsible for calculating the values for {index} and substituting those values into the “pageLinkTemplate”. The pageLinkTemplate’s {index} variable is calculated as follows:
  - Index #1 = 0 * itemsPerPage = 0
  - Index #2 = 2 * itemsPerPage = 10
  - Index #3 = 3 * itemsPerPage = 20
  - Index #N = N * itemsPerPage

附录

附录A:JavaScript中的保留字

下列JavaScript保留字应该避免在属性名中使用

下面的但在在JavaScript语言中被保留，不能作为在点访问符中使用。这份名单代表此时的最佳关键字的知识;列表可能会改变或根据您的特定的执行环境更改。

下面是JavaScript语言中的保留字，且不能在点访问符中使用。这份清单集合了当前最新的关键字，该清单可能会根据具体的执行环境而有所变更或改变。

来自ECMAScript 语言规范第五版

```
abstract
boolean break byte
case catch char class const continue
debugger default delete do double
else enum export extends
false final finally float for function
goto
if implements import in instanceof int interface
let long
native new null
package private protected public
return
short static super switch synchronized
this throw throws transient true try typeof
var volatile void
while with
yield
```


－EOF-