MySQL JSON 数据类型的使用

什么是json

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。它基于ECMAScript的一个子集。

JSON采用完全独立于语言的文本格式，但是也使用了类似于C语言家族的习惯（包括C、C++、C#、Java、JavaScript、Perl、Python等）。

这些特性使JSON成为理想的数据交换语言。 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 语法规则

JSON 语法是 JavaScript 对象表示语法的子集。

- 数据在键值对中
- 数据由逗号分隔
- 花括号保存对象
- 方括号保存数组

JSON 名称/值对

JSON 数据的书写格式是：名称/值对。

名称/值对组合中的名称写在前面（在双引号中），值对写在后面(同样在双引号中)，中间用冒号隔开：

```
"firstName":"John"
```

这很容易理解，等价于这条 JavaScript 语句：

```
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）:数组是值（value）的有序集合
- 对象（在花括号中）:对象是一个无序的“名称/值”对”集合
- null

实例

```
{
    "name": "BeJson",
    "url": "http://www.bejson.com",
    "page": 88,
    "isNonProfit": true,
    "address": {
        "street": "科技园路.",
        "city": "江苏苏州",
        "country": "中国"
    },
    "links": [
        {
            "name": "Google",
            "url": "http://www.google.com"
        },
        {
            "name": "Baidu",
            "url": "http://www.baidu.com"
        },
        {
            "name": "SoSo",
            "url": "http://www.SoSo.com"
        }
    ]
}
```

优点

1. 方便于传输，较少冗余的字符。当然直接传二进制是最好的，但面临难解析的问题。亦可以是xml、纯字符串的方式，但json有其独到的好处。google有个自己的协议，叫protobuf，有兴趣可了解一下。
2. 方便转换。有很多的json api提供了json字符串转成对象、对象转换成json串的方法。
3. 易于阅读。json代码的良好结构，可以很直观地了解存的是什么内容。

MySQL 5.7.8 开始添加json

As of MySQL 5.7.8, MySQL supports a native JSON data type that enables efficient access to data inJSON (JavaScript Object Notation) documents. The JSON data type provides these advantages overstoring JSON-format strings in a string column:

从 MySQL 5.7.8 开始，MySQL支持原生态的JSON数据类型。这使得MySQL数据库可以高效地访问JSON 文档类型的数据。JSON数据类型提供以下优势，在列的类型为字符串类型的时，使用JSON类型的字符串优势更大。

- Automatic validation of JSON documents stored in JSON columns. Invalid documents produce an error.
- Optimized storage format. JSON documents stored in JSON columns are converted to an internal format that permits quick read access to document elements. When the server later must read a JSON values stored in this binary format, the value need not be parsed from a text representation. The binary format Creating JSON Values is structured to enable the server to look up subobjects or nested values directly by key or array index without reading all values before or after them in the document.
- JSON列中存储的JSON文档能够自动验证。无效的文件会产生一个报错。
- 优化存储类型。JSON列中存储的JSON文档被转换为一个内部格式,内部格式允许快速的读和访问文档元素。当服务器之后必须读取存储为二进制格式的JSON数据的值时,就不需要从文本表示方式来解析了。创建JSON数据值的二进制格式使服务器查找子对象、直接通过键得到的嵌套值、数组索引时，不需要读取文档中所有值，不管这些值是已经在文档中还是之后加进来的。

MySQL 5.7.8开始支持json字段类型,并提供了不少内置函数,通过计算列,甚至还可以直接索引json中的数据!

为何说json原生支持非常关键呢,不是可以自己在客户端处理json然后保存字符串到mysql不就完了?

来看一看，原生支持到底有什么意义

```
create table t (
 id int not null,
 js json not null,
 PRIMARY KEY (id)
)
```

> 1.可以直接过滤记录

select * from t where js->'$.a'=100

避免了要将所有记录都读取出来，在客户端进行过滤。

> 2.可以直接update，而无须先读取

update t set js=json_set(js,'$.a',js->'$.a'+1) where id=1

单条原子更新

update t,t1 set t.js=json_merage(t.js,t1.js) where t.id=t1.id

跨表更新

> 3.可以在一条SQL中完成多条纪录的修改!

update t set js=json_set(js,'$.a',123) where id in(1,2)

没有原生的支持，这个是很难实现的。

> 4.通过json类型，完美的实现了表结构的动态变化
> 除了一般意义上的增加表字段，还包括嵌套其他对象与数组

update t set js=json_array_append(js,'$.sonAry',123) where id =1

增加一个子节点到sonAry中，无须子表。

> 5.通过计算生成列在json上建立索引

```
CREATE TABLE j1 (
  id int(11) NOT NULL,
  js json NOT NULL,
  s varchar(45) CHARACTER SET utf8mb4 NOT NULL,
  a int(11) GENERATED ALWAYS AS (json_extract(js,'$.a')) STORED,
  PRIMARY KEY (id),
  KEY i_a (a)
)
```

通过a这个生成列`(json_extract(js,'$.a'))`上建立索引,可以利用mysql的索引来快速定位。

`json_extract`还可利用`path`的通配符，发掘更多类型索引。

甚至还可利用`JSON_CONTAINS/JSON_CONTAINS_PATH`来建立索引。这里可以组合出很多。

与MongoDB的差异

1. **mongodb会自动建立表**，也就是可以动态增加表,这点mysql还是必须先定义表
2. **大部分查询插入更新操作**,因为mysql json支持,两者没有什么功能差别了。但是就查询来说,mysql的SQL语法,特别是json_extract的简写模式,可读性比mongodb的要强不少
3. **mongodb实际用牺牲事务性降低系统复杂度来实现高性能**，这方面,要看用户使用场景,很难说优势还是劣势
4. **对于大数据的支持**，mongodb在大数据支持上做的比较好,特别是数据分片,高可用上基本内置实现了；mysql这方面也在不断完善，其差异主要还是在一致性上的选取,弱一致性导致容易分布处理,而强一致性则复杂很多.

我们可以假设,关系数据库设计之初如果放弃强一致性,那么这方面,mongodb等nosql实际也不会有任何优势.

从未来趋势来看,

强一致性方面,传统关系数据库始终是唯一最佳选择,而且通过支持json,把以前Nosql的一些优势也可以纳入到自己体系中.

对于固定的字段,显然传统的表处理方式性能更优,而对于动态字段放置独立的json字段中,也可以很好的适应这类需求.

弱一致性方面,NoSQL更擅长一些,传统关系数据库则可以在应用层面通过多实例数据库的分布方式来实现(来降低一致性提高数据吞吐量).

而在大分布式方面最终一致性方面,传统关系数据库缺乏直接支持,但还是可以通过应用层来实现,而很多NoSQL也在不断完善这方面的功能.

我们期待一个基础关系数据库,每个实例提供强一致性的事务,易用的SQL,并对动态数据作出很好的支持.

并在实例之上有一个分布式的层,可以聚合大量数据库实例,并协助用户实现最终一致性,

这样就比较完美了.

如何使用json数据类型

声明json类型的列

```
create table t(id int,js json,PRIMARY KEY (`id`))
```

插入数据

```
insert into t values(1,'{"a":1,"s":"abc"}')
insert into t values(2,'[1,2,{"a":123}]')
insert into t values(3,'"str"')
insert into t values(4,'123')

# 直接提供字符串即可。还可以用JSON_Array和JSON_Object函数来构造
insert into t values(5,JSON_Object('key1',v1,'key2',v2))
insert into t values(4,JSON_Array(v1,v2,v3))
```

- `JSON_OBJECT([key, val[, key, val] ...])`
- `JSON_ARRAY([val[, val] ...])`
- `JSON_SET(json_doc, path, val[, path, val] ...)`

修改数据

```
update t set js=json_set('{"a":1,"s":"abc"}','$.a',456,'$.b','bbb') where id=1
```

结果`js={"a":456,"s":"abc","b":"bbb"}`

path中`$`就代表整个`doc`,然后可以用`JavaScript`的方式指定对象属性或者数组下标等.执行效果，类似json的语法

```
$.a=456
$.b="bbb"
```

存在就修改,不存在就设置.

```
$.c.c=123
```

这个在`javascript`中会出错，因为`.c`为`null`。
但是在`json_set('{}','$.c.c',123)`中，不存在的路径将直接被忽略。

特殊的对于数组，如果目标`doc`不是数组则会被转换成`[doc]`，然后再执行`set`，
如果`set`的下标超过数组长度，只会添加到数组结尾。

```
select json_set('{"a":456}','$[1]',123)

# 结果[{"a":456},123]。目标现被转换成[{"a":456}],然后应用$[1]=123。

select json_set('"abc"','$[999]',123)

# 结果["abc",123]。
```

再举几个例子

```
select json_set('[1,2,3]','$[0]',456,'$[3]','bbb')

# 结果[456,2,3,'bbb']
```

注意:
对于`javascript`中
`var a=[1,2,3]`
`a.a='abc'`
是合法的,但是一旦`a`转成`json`字符串,`a.a`就丢失了。

而在`mysql`中，这种算作路径不存在，因此
`select json_set('[1,2,3]','$.a',456)`
结果还是`[1,2,3]`

然后还有另外两个版本
`JSON_INSERT(json_doc, path, val[, path, val] ...)`
如果不存在对应属性则插入,否则不做任何变动

`JSON_REPLACE(json_doc, path, val[, path, val] ...)`
如果存在则替换,否则不做任何变动

这两个操作倒是没有javascript直接对应的操作
`select json_insert('{"a":1,"s":"abc"}','$.a',456,'$.b','bbb')`
结果`{"a":1,"s":"abc","b":"bbb"}`

`select json_replace('{"a":1,"s":"abc"}','$.a',456,'$.b','bbb')`
结果`{"a":456,"s":"abc"}`

加上删除节点
`JSON_REMOVE(json_doc, path[, path] ...)`
如果存在则删除对应属性,否则不做任何变动
`select json_replace('{"a":1,"s":"abc"}','$.a','$.b')`
结果`{"s":"abc"}`

涉及数组时，三个函数与`json_set`基本一样
`select json_insert('{"a":1}','$[0]',456)`
结果不变，认为`0`元素已经存在了，注意这里结果不是`[{"a":1}]`

`select json_insert('{"a":1}','$[999]',456)`
结果追加到数组结尾`[{"a":1}, 456]`

`select json_replace('{"a":1}','$[0]',456)`
结果`456`！而非`[456]`

`select json_replace('{"a":1}','$[1]',456)`
结果不变。

其实对于`json_insert`和`json_replace`来说一般情况没必要针对数组使用。

`select json_remove('{"a":1}','$[0]')`
结果不变!

`select json_remove('[{"a":1}]','$[0]')`
结果`[]`

总之涉及数组的时候要小心。

`JSON_MERGE(json_doc, json_doc[, json_doc] ...)`
将多个doc合并

`select json_merge('[1,2,3]','[4,5]')`
结果`[1,2,3,4,5]`。

数组简单扩展

`select json_merge('{"a":1}','{"b":2}')`
结果`{"a":1,"b":2}`。两个对象直接融合。

特殊的还是在数组
`select json_merge('123','45')`
结果`[123,45]`。两个常量变成数组

`select json_merge('{"a":1}','[1，2]')`
结果`[{"a":1},1,2]`。目标碰到数组，先转换成`[doc]`

`select json_merge('[1,2]','{"a":1}')`
结果`[1,2,{"a":1}]`。非数组都追加到数组后面。

`JSON_ARRAY_APPEND(json_doc, path, val[, path, val] ...)`
给指定的节点，添加元素，如果节点不是数组，则先转换成`[doc]`

`select json_Array_append('[1,2]','$','456')`
结果`[1,2,456]`

`select json_Array_append('[1,2]','$[0]','456')`
结果`[[1,456],2]`。指定插在`$[0]`这个节点，这个节点非数组，所以等效为
`select json_Array_append('[[1],2]','$[0]','456')`

`JSON_ARRAY_INSERT(json_doc, path, val[, path, val] ...)`
在数组的指定下标处插入元素

`SELECT JSON_ARRAY_INSERT('[1,2,3]','$[1]',4)`
结果`[1,4,2,3]`。在`$`数组的下标1处插入

`SELECT JSON_ARRAY_INSERT('[1,[1,2,3],3]','$[1][1]',4)`
结果`[1,[1,4,2,3],3]`。在`$[1]`数组的下标1处插入

`SELECT JSON_ARRAY_INSERT('[1,2,3]','$[0]',4,'$[1]',5)`
结果`[4,5,1,2,3]`。注意后续插入是在前面插入基础上的，而非`[4,1,5,2,3]`

提取`json`信息的函数
`JSON_KEYS(json_doc[, path])`
返回指定`path`的`key`

`select json_keys('{"a":1,"b":2}')`
结果`["a","b"]`

`select json_keys('{"a":1,"b":[1,2,3]}','$.b')`
结果`null`。数组没有`key`

`JSON_CONTAINS(json_doc, val[, path])`
是否包含子文档

```
select json_contains('{"a":1,"b":4}','{"a":1}')
结果1

select json_contains('{"a":2,"b":1}','{"a":1}')
结果0

select json_contains('{"a":[1,2,3],"b":1}','[1,2]','$.a')
结果1。数组包含则需要所有元素都存在。

select json_contains('{"a":[1,2,3],"b":1}','1','$.a')
结果1。元素存在数组元素中。
```

`JSON_CONTAINS_PATH(json_doc, one_or_all, path[, path] ...)`
检查路径是否存在

```
select JSON_CONTAINS_PATH('{"a":1,"b":1}', 'one','$.a','$.c')
结果1。只要存在一个

select JSON_CONTAINS_PATH('{"a":1,"b":1}', 'all','$.a','$.c')
结果0。必须全部存在。


select JSON_CONTAINS_PATH('{"a":1,"b":{"c":{"d":1}}}', 'one','$.b.c.d')
结果1。

select JSON_CONTAINS_PATH('{"a":1,"b":{"c":{"d":1}}}', 'one','$.a.c.d')
结果0。
```

`JSON_EXTRACT(json_doc, path[, path] ...)`
获得doc中某个或多个节点的值。

```
select json_extract('{"a":1,"b":2}','$.a')
结果1

select json_extract('{"a":[1,2,3],"b":2}','$.a[1]')
结果2


select json_extract('{"a":{"a":1,"b":2,"c":3},"b":2}','$.a.*')
结果[1,2,3]。a.*通配a所有属性的值返回成数组。


select json_extract('{"a":{"a":1,"b":2,"c":3},"b":4}','$**.b')
结果[2,4]。通配$中所有层次下的属性b的值返回成数组。
```

mysql5.7.9开始增加了一种简写方式：column->path

```
select id,js->'$.id' from t where js->'$.a'=1 order by js->'$.b'
# 等价于
select id,json_extract(js,'$.id')
from t where json_extract(js,'$.a')=1
order by json_extract(js,'$.b')
```

`JSON_SEARCH(json_doc, one_or_all, search_str[, escape_char[, path] ...])`
强大的查询函数，用于在doc中返回符合条件的节点，select则是在表中返回符合要求的纪录。

`select json_search('{"a":"abc","b":{"c":"dad"}}','one','%a%')`

结果`$.a`。和`like`一样可以用`%`和`_`匹配，在所有节点的值中匹配，one只返回一个。

```
select json_search('{"a":"abc","b":{"c":"dad"}}','all','%a%')
# 结果["$.a","$.b.c"]


select json_search('{"a":"abc","b":{"c":"dad"}}','all','%a%',null,'$.b')
# 结果["$.b.c"]。限制查找范围。

select json_search('{"a":"abc","b":{"c":"dad"},"c":{"b":"aaa"}}','all','%a%',null,'$**.b')
# 结果["$.b.c","$.c.b"]。查找范围还可使用通配符！在每个匹配节点和其下查找。
```

注意，只有json_extract和json_search中的path才支持通配，其他json_set,json_insert等都不支持。

`JSON_LENGTH(json_doc[, path])`
返回数组的长度，如果是object则是属性个数，常量则为`1`

```
select json_length('[1,2,3]')
结果3

select json_length('123')
结果1

select json_length('{"a":1,"b":2}')
结果2

可再跟path参数
select json_length('{"a":1,"b":[1,2,3]}','$.b')
结果3
```

`JSON_DEPTH(json_doc)`
返回`doc`深度

```
select json_depth('{}'), json_depth('[]'),json_depth('123')
结果1,1,1

select json_depth('[1,2,3,4,5,6]')
结果2

select json_depth('{"a":{"b":{"c":1}}}')
结果4
```