# MySQL Document Store面面观

* 详解新世代MySQL如何支持NoSQL应用之开发

杜修文 2017 元月

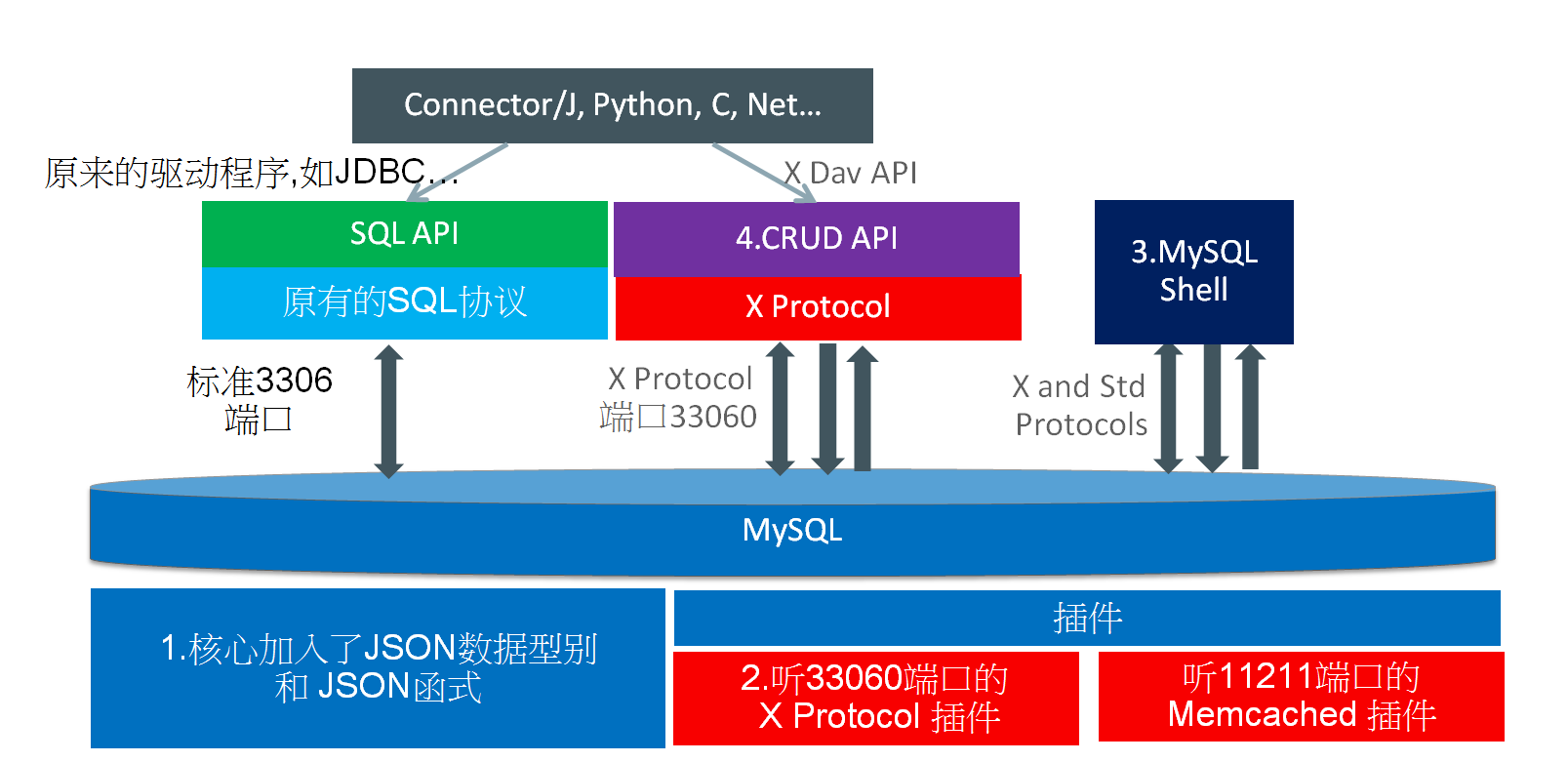
MySQL面世了20多年来在大家的认知中都是一个关系型数据库,但是从2016年起MySQL有了一个革命性的变化-它除了是一个说SQL语言的关系型数据库,它也学会了第二种语言-JSON,现在也是一个NoSQL文件型数据库.

大约在两年前,Oracle的MySQL产品部门体认到了的软件工程正朝向雏型法,快速开发,和开发维运(DevOp)的方向发展.传统的关系型数据库虽以严仅称着,强调数据完整性,和通过正规化带来查询弹性.但相对的以较长时间更动数据架构的做法,要支持现今软件工程的方法论,存在一定程度的适用性问题,同时过度强调正规化也会因常需要以多表的联接(JOIN)才能查一份完整的数据,而存在性能不佳的问题.所以MySQL决定启动一个代码为MySQLng (MySQL Next Generation)项目,这个工程投入了包括数据库核心,前端工具和链接器(Connector)部门大量的开发人力,成就了MySQL在2016年推出的以MySQL当文件储存(MySQL as a Document Store)新功能.

MySQL的架构因此而出现的变化

MySQL在5.7.12以后就是一个完全支持JSON文件的文件型数据库,它从数据库核心到前端工具和支持各种前端应用技术的链接器(Connector)都出现了许多的新功能, 如下图所示,可分成四个部份:

1. 数据库核心加了新的JSON数据型别和JSON函式
2. 新加入一个名为mysqlx的插件,运行时会33060端口,使数据库能通过X Protocol和X Dav API运行应用程序下达的CRUD操作
3. 新的MySQL客户端 – MySQL Shell以JSON文件的CRUD方法直接操作数据
4. 扩增多种链接器 (包括Connector/J, Connector/Python, Connector/Nodejs, Connetor/C,C++, 以及Connector/PHP可通过Javascript) 的功能,包裹(wrap)X Dav API,使用X Protocol能以纯NoSQL方式方式对数据库的JSON文件做操作,使整个应用的开发完全没有SQL的影子.

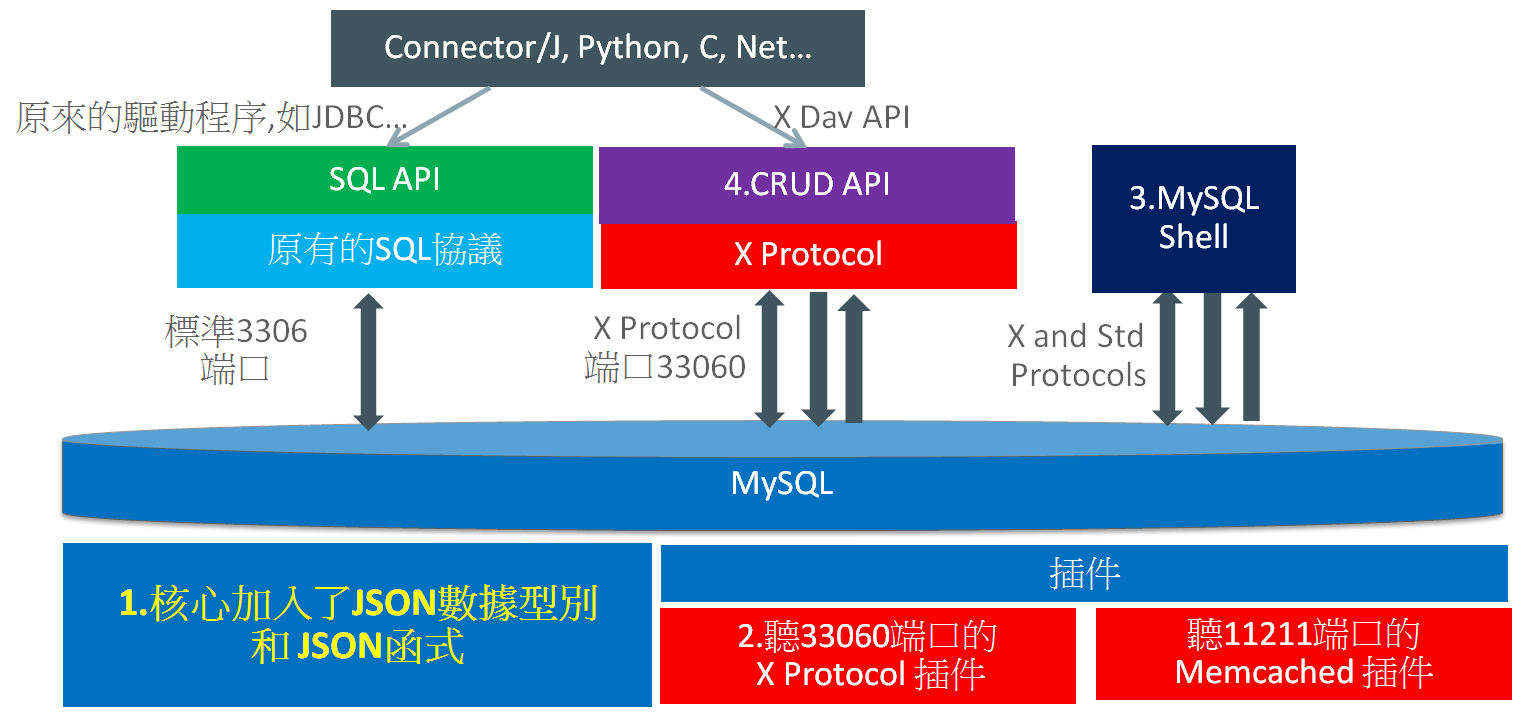


以下对这四个支持以MySQL当做文件存储(MySQL as a Document Store)的各模块模块特性详细说明。

数据库核心加入了新的JSON数据型别和JSON函式

* 1. JSON数据型别

如下图的1.所示,自MySQL 5.7.8起这项功能就含在MySQL中了.您可以用您熟悉的mysql客户端程序登入数据库,建一个有JSON字段的表,再将JSON文件到插入这个字段,具体的范例如下所示.JSON字段是以binary为基础再于字段头标示文件中各对象的位移(offset),如此能让MySQL能快速的找到查询所指定的对象.



### 注意是data字段是JSON资料型别,不是text或varchar资料型别

mysql> CREATE TABLE t1 (data JSON);

Query OK, 0 rows affected (0.08 sec)

### 以字符串形式将数据插入表中,对象名称要在引号内

mysql> INSERT INTO t1(data) VALUES ('{ "series": 1}'), ('{ "series": 7}'), ('{ "series": 3}'), ('{ "series": 4}'), ('{ "series": 10}'), ('{ "series": 2}'), ('{ "series": 6}'), ('{ "series": 5}'), ('{ "series": 8}'), ('{ "series": 11}');

Query OK, 10 rows affected (0.02 sec)

Records: 10 Duplicates: 0 Warnings: 0

### 如果插入的不是合于JSON规范的资料会被MySQL拒绝并报错

mysql> insert into t1(data) values ('abc');

ERROR 3140 (22032): Invalid JSON text: "Invalid value." at position 0 in value for column 't1.data'.

mysql> insert into t1(data) values (123);

ERROR 3140 (22032): Invalid JSON text: "not a JSON text, may need CAST" at position 0 in value for column 't1.data'.

* 1. 数据库出现了新的JSON函式(JSON Function)操作JSON文件的内容,例如JSON\_EXTRACT…

mysql> SELECT \* FROM t1 WHERE json\_extract(data,"$.series") >= 7 AND json\_extract(data,"$.series") <=10;

+----------------+

| data |

+----------------+

| {"series": 7} |

| {"series": 10} |

| {"series": 8} |

+----------------+

3 rows in set (0.00 sec)

目前有23个JSON函式(以后随着更新的版本推出,可能会增加更多的JSON函式以完善处理JSON文件的功能),依这些函式使用的时机可分为检视JSON文件信息、更改(由路径参数指定的)JSON文件之内容、建立JSON文件、取出(由路径参数指定)部份的JSON文件内容、和辅助性函式等五类.

由于JSON函式会以路径(path)指定到JSON文件中一定的位置,取出或参照部份文件的信息,在介绍JSON函式前,先说明路径语法:

* + - 在SQL命令中先要找到存放JSON文件的JSON字段,其语法为[[database].table.]field –找到整个JSON文件
    - 找到字段后再以路径指向存于字段的JSON文件某一部份 – 在JSON文件中的路径的语法如下:
* $ - JSON文件的根
* .identifier – 指定名称,到该名称所对应的值或对象,例如$.user-找出JSON文件中user对象或值
* [array-position] –指定阵列中某一序位的内容(由0开始),例如$.user.address[1]-找到JSON文件的user对象之下的address阵列之第2个序位的地址对象
* \* - 指定某一对象之下的所有成员对象,例如$user.\* 代表找出JSON文件中user对象下所有的成员对象,可能包括名字,生日,及所有的地址…等
* \*\* - 可当成一个wild card,用法是[prefix]\*\*suffix,prefix是选项,也就是不会以\*\*当结尾, 例如$user\*\*.phone指向user对象下的phone对象
* [\*] – 找出阵列中所有的序位的内容

JSON path之范例将在下一段JSON\_EXTRAT函式中再为您做详细的

说明

* + 1. 接下来再介绍五大类JSON函式,其中最常用到的函式是JSON\_EXTRACT或->,以及->>,其他的函式列出的范例稍为琐碎了些,仅用以说明其功能,供读者们参考:
       - 检视JSON文件的内容或属性,包括:
  1. JSON\_VALID(*val*), 检查传入的资料是否是一个合规的JSON文件,目前是以较旧的RFC2627为标准.合规传出1, 不合规传出0,若传入NULL值,则传出的也是NULL值.这个函式能让我们在应用中就先测试所拿到的数据是否是一个JSON文件,如果不是,则应用开发者能在应用中就能先处理.
  2. JSON\_TYPE(*json\_val[,path]*),检视指定的JSON文件或文件中以路径-path指向的数据型别.这个函式的第二个参数-path之后多个路径.
  3. JSON\_KEYS(json\_doc[,path]),传回指定JSON文件的最上一层(或是指定路径)的键值(identifier),以下列范例说明其用途:

# 文件的根之下有两个identifier – “a” 和”b”

mysql> SELECT JSON\_KEYS('{"a": 1, "b": {"c": 30}}');

+---------------------------------------+

| JSON\_KEYS('{"a": 1, "b": {"c": 30}}') |

+---------------------------------------+

| ["a", "b"] |

+---------------------------------------+

# 文件的”根”之下的成员”b”对象之下有一个identifier – “b”

mysql> **SELECT JSON\_KEYS('{"a": 1, "b": {"c": 30}}', '$.b');**

+----------------------------------------------+

| JSON\_KEYS('{"a": 1, "b": {"c": 30}}', '$.b') |

+----------------------------------------------+

| ["c"] |

+----------------------------------------------+

1. JSON\_LENGTH(*json\_doc,path*),传回指定JSON文件最上一层(或是指定路径)的成员数,例如:

#传入JSON\_LENGTH函式的参数为一个有三个元素的数组

mysql> **SELECT JSON\_LENGTH('[1, 2, {"a": 3}]');**

+---------------------------------+

| JSON\_LENGTH('[1, 2, {"a": 3}]') |

+---------------------------------+

| 3 |

+---------------------------------+

#传入JSON\_LENGTH函式的参数有两件,其identifier为a和b

mysql> **SELECT JSON\_LENGTH('{"a": 1, "b": {"c": 30}}');**

+-----------------------------------------+

| JSON\_LENGTH('{"a": 1, "b": {"c": 30}}') |

+-----------------------------------------+

| 2 |

+-----------------------------------------+

#所指定的路径是根之下的b对象之下有一个子对象-c

mysql> **SELECT JSON\_LENGTH('{"a": 1, "b": {"c": 30}}', '$.b');**

+------------------------------------------------+

| JSON\_LENGTH('{"a": 1, "b": {"c": 30}}', '$.b') |

+------------------------------------------------+

| 1 |

+------------------------------------------------+

1. JSON\_DEPTH(*json\_doc*),传回指定JSON文件最大(深)的阶层,例如:

mysql> SELECT JSON\_DEPTH('{}'), JSON\_DEPTH('[]'), JSON\_DEPTH('true');

+------------------+------------------+--------------------+

| JSON\_DEPTH('{}') | JSON\_DEPTH('[]') | JSON\_DEPTH('true') |

+------------------+------------------+--------------------+

| 1 | 1 | 1 |

+------------------+------------------+--------------------+

mysql> SELECT JSON\_DEPTH('[10, 20]'), JSON\_DEPTH('[[], {}]');

+------------------------+------------------------+

| JSON\_DEPTH('[10, 20]') | JSON\_DEPTH('[[], {}]') |

+------------------------+------------------------+

| 2 | 2 |

+------------------------+------------------------+

mysql> SELECT JSON\_DEPTH('[10, {"a": 20}]');

+-------------------------------+

| JSON\_DEPTH('[10, {"a": 20}]') |

+-------------------------------+

| 3 |

+-------------------------------+

1. JSON\_CONTAINS\_PATH(j*son\_doc,one | all, path[,path…]*),检视指定的JSON文件中是否有一个或全部指定的路径,例如:

mysql> SET @j = '{"a": 1, "b": 2, "c": {"d": 4}}';

# 第二个参数’one’代表JSON文件能找到任一个在这个参数之后的路径,就传回true或1,变量j中的JSON文件的”根”之下有a对象

mysql> SELECT JSON\_CONTAINS\_PATH(@j, 'one', '$.a', '$.e');

+---------------------------------------------+

| JSON\_CONTAINS\_PATH(@j, 'one', '$.a', '$.e') |

+---------------------------------------------+

| 1 |

+---------------------------------------------+

#第二个参数’all’代表JSON文件能找到所有在这个参数之后的路径,才会传回true或1,变量j中的JSON文件在”根”之下没有e对象

mysql> SELECT JSON\_CONTAINS\_PATH(@j, 'all', '$.a', '$.e');

+---------------------------------------------+

| JSON\_CONTAINS\_PATH(@j, 'all', '$.a', '$.e') |

+---------------------------------------------+

| 0 |

+---------------------------------------------+

#JSON文件的”根”之下有一对象其identifier为d

mysql> SELECT JSON\_CONTAINS\_PATH(@j, 'one', '$.c.d');

+----------------------------------------+

| JSON\_CONTAINS\_PATH(@j, 'one', '$.c.d') |

+----------------------------------------+

| 1 |

+----------------------------------------+

mysql> SELECT JSON\_CONTAINS\_PATH(@j, 'one', '$.a.d');

+----------------------------------------+

| JSON\_CONTAINS\_PATH(@j, 'one', '$.a.d') |

+----------------------------------------+

| 0 |

+----------------------------------------+

* + 取出JSON文件的内容
  1. JSON\_EXTRACT(*jond\_doc,path*)

例如:

# 定一个名为j的JSON文件的mysql区域变量,其内容如下

set @j='{"user":{"name":{"first":"abc","last":"ssss"},

"salary":12234,"address":[{"street":"def","number":"ghi","phone":"5678890"},

{"street":"jkl","number":"mno","phone":"565656"}]}}';

#整理JSON文件的内容以阶层方式排列,让您比较能了解JSON文件的结构:

#{"user":

{

"name":{"first":"abc","last":"ssss"},

"salary":12234,

"address":

[ {"street":"def","number":"ghi","phone":"5678890"},

{"street":"jkl","number":"mno","phone":"565656"}

]

}

}

# 验证变量j的内容是否是一个合规的 JSON文件

mysql> select json\_valid(@j);

+----------------+

| json\_valid(@j) |

+----------------+

| 1 |

+----------------+

1 row in set (0.00 sec)

# 查出JSON文件”根”以下的内容

mysql> select json\_extract(@j,'$');

+---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+

| json\_extract(@j,'$') |

+---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+

| {"user": {"name": {"last": "ssss", "first": "abc"}, "salary": 12234, "address": [{"phone": "5678890", "number": "ghi", "street": "def"}, {"phone": "565656", "number": "mno", "street": "jkl"}]}} |

+---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+

1 row in set (0.00 sec)

# 找出位于JSON文件”根”-‘$’以下的成员”user”之”name”的内容

mysql> select json\_extract(@j,'$.user.name');

+----------------------------------+

| json\_extract(@j,'$.user.name') |

+----------------------------------+

| {"last": "ssss", "first": "abc"} |

+----------------------------------+

1 row in set (0.00 sec)

# 找出位于JSON文件”根”以下的成员”user”之”address”阵列所有的cell

mysql> select json\_extract(@j,'$.address[\*]');

+-----------------------------------------------------------------------------------------------------------------+

| json\_extract(@j,'$.address[\*]') |

+-----------------------------------------------------------------------------------------------------------------+

| [{"phone": "5678890", "number": "ghi", "street": "def"}, {"phone": "565656", "number": "mno", "street": "jkl"}] |

+-----------------------------------------------------------------------------------------------------------------+

1 row in set (0.00 sec)

# 找出位于JSON文件”根”以下的成员”address”阵列中位于第二位的cell

mysql> select json\_extract(@j,'$.user.address[1]');

+-------------------------------------------------------+

| json\_extract(@j,'$.user.address[1]') |

+-------------------------------------------------------+

| {"phone": "565656", "number": "mno", "street": "jkl"} |

+-------------------------------------------------------+

1 row in set (0.00 sec)

# 找出JSON文件位于”根”之下所有的成员,和路径’$’不同的是,它不会列示文件的根-‘user’,而是user之下的所有成员.

mysql> select json\_extract(@j,'$.\*');

+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+

| json\_extract(@j,'$.\*') |

+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+

| [{"name": {"last": "ssss", "first": "abc"}, "salary": 12234, "address": [{"phone": "5678890", "number": "ghi", "street": "def"}, {"phone": "565656", "number": "mno", "street": "jkl"}]}] |

+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+

1 row in set (0.00 sec)------------------------+

1 row in set (0.00 sec)

# 找出”根”之下”user”成员中任何一个地方的成子员”last”的内容

mysql> select json\_extract(@j,'$.user\*\*.last');

+----------------------------------+

| json\_extract(@j,'$.user\*\*.last') |

+----------------------------------+

| ["ssss"] |

+----------------------------------+

1 row in set (0.00 sec)

# 找出”根”之下”user”成员的”name”成员的所有内容

mysql> select json\_extract(@j,'$.user.name.\*');

+----------------------------------+

| json\_extract(@j,'$.user.name.\*') |

+----------------------------------+

| ["ssss", "abc"] |

+---------------------------------+

1 row in set (0.00 sec)

# 查出JSON文件”根”的成员”user”的”salary”的数据型别,请注意函式是可以圈套的,此处先以JSON\_EXTRACT找到”slary”物件,再传给JSON\_TYPE判断其数据型别

mysql> select json\_type(json\_extract(@j,'$.user.salary'));

+---------------------------------------------+

| json\_type(json\_extract(@j,'$.user.salary')) |

+---------------------------------------------+

| INTEGER |

+---------------------------------------------+

1 row in set (0.00 sec)

* 1. JSON\_SEARCH(json\_doc,one or all, search\_str [,escape\_char[,path]…]),传回指定的JSON文件,或由传入的路径指定的部份JSON文件之下,合搜索字符串相符的对象所在的路径,如果找不到和搜索字符串相符的对象则传回NULL值,范例如下:

# JSON文件由”根”往下找,第一次碰到值为”abc”的位置为在根之下的阵列中第一个位置(cell) - $[0]

mysql> SET @j = '["abc", [{"k": "10"}, "def"], {"x":"abc"}, {"y":"bcd"}]';

mysql> SELECT JSON\_SEARCH(@j, 'one', 'abc');

+-------------------------------+

| JSON\_SEARCH(@j, 'one', 'abc') |

+-------------------------------+

| "$[0]" |

+-------------------------------+

# 找出所有值为”abc”的位置,结果找到两个,除了根之下阵列的第一个位置(cell)之外,在第二个位置(cell)的成员”x”的内容也是”abc”,传回的是一个有两个元素的数组mysql> SELECT JSON\_SEARCH(@j, 'all', 'abc');

+-------------------------------+

| JSON\_SEARCH(@j, 'all', 'abc') |

+-------------------------------+

| ["$[0]", "$[2].x"] |

+-------------------------------+

mysql> SELECT JSON\_SEARCH(@j, 'all', 'ghi');

+-------------------------------+

| JSON\_SEARCH(@j, 'all', 'ghi') |

+-------------------------------+

| NULL |

+-------------------------------+

* 1. -> 相当于JSON\_EXTRACT(),会传出整个对象,例如有一个表mycollection的doc字段内含JSON文件:

mysql> select doc from mycollection limit 4\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

doc: {"\_id": "13a90b0a4af7451c9affb89dae36d5a8", "age": 15, "name": "Sakila"}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

doc: {"\_id": "9d0eec5aa65411e6904b0a0027000003", "policy": {"mainInsured": {"coverageList": [{"item": "ULOB"}]}}}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

doc: {"\_id": "a34efb1c3344487f90c2d0802aa92098", "age": 39, "name": "Mike"}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 4. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

doc: {"\_id": "c8a5d26905e843dfa2fc15083f6fbd0d", "age": 24, "name": "Susanne"}

# mysql> select json\_extract(doc,'$.name') from mycollection limit 4\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

json\_extract(doc,'$.name'): "Sakila"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

json\_extract(doc,'$.name'): NULL

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

json\_extract(doc,'$.name'): "Mike"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 4. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

json\_extract(doc,'$.name'): "Susanne"

4 rows in set (0.00 sec)

# 以”->“也能行到同样的解果:

mysql> select doc->'$.name' from mycollection limit 4\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

doc->'$.name': "Sakila"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

doc->'$.name': NULL

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

doc->'$.name': "Mike"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 4. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

doc->'$.name': "Susanne"

4 rows in set (0.00 sec)

* 1. ->> 等同于JSON\_UNQUOTE(JSON\_EXTRACT()),脱掉引号再传回,方便我们在WHERE条件中找出某些讯息做比较,例如:

mysql> select doc->'$.name' from mycollection where doc->'$.age' > 20 limit 2;

+---------------+

| doc->'$.name' |

+---------------+

| "Sunny" |

| "John" |

+---------------+

2 rows in set (0.00 sec)

mysql> select doc->>'$.name' from mycollection where doc->'$.age' > 20 limit 2;

+----------------+

| doc->>'$.name' |

+----------------+

| Sunny |

| John |

+----------------+

2 rows in set (0.00 sec)

由于JSON\_EXTRACT会传回对象,如果是字符串,会带双引号,必需以JSON\_UNQUOTE()将双引号移除才能用于比较运算

* 1. 建立JSON对象
* JSON\_MERGE(*json\_doc,json\_doc[,json\_doc]…*), 合并多个JSON文件成为一个JSON文件,例如:

mysql> SELECT JSON\_MERGE('[1, 2]', '[true, false]');

+---------------------------------------+

| JSON\_MERGE('[1, 2]', '[true, false]') |

+---------------------------------------+

| [1, 2, true, false] |

+---------------------------------------+

mysql> SELECT JSON\_MERGE('{"name": "x"}', '{"id": 47}');

+-------------------------------------------+

| JSON\_MERGE('{"name": "x"}', '{"id": 47}') |

+-------------------------------------------+

| {"id": 47, "name": "x"} |

+-------------------------------------------+

* JSON\_ARRAY(*val[,val]…*),将传入的值做成一个JSON阵列,例如:

mysql> SELECT JSON\_ARRAY(1, "abc", NULL, TRUE, CURTIME());

+---------------------------------------------+

| JSON\_ARRAY(1, "abc", NULL, TRUE, CURTIME()) |

+---------------------------------------------+

| [1, "abc", null, true, "11:30:24.000000"] |

+---------------------------------------------+

* JSON\_OBJECT(key,val[,key,val]…),将传入的一个以上的key(或identifier)-值对做成一个JSON对象,例如:

mysql> SELECT JSON\_OBJECT('id', 87, 'name', 'carrot');

+-----------------------------------------+

| JSON\_OBJECT('id', 87, 'name', 'carrot') |

+-----------------------------------------+

| {"id": 87, "name": "carrot"} |

+-----------------------------------------+

* 1. 更改JSON文件
  2. JSON\_REMOVE(json\_doc,path[,path]…),自文件中移除由路径所指定的部份文件,可移除多个路径的对象或以\*及\*\*指定多个对象,范例如下:

# 自文件文中移除”根”之下的阵列中第二位置的cell

mysql> SET @j = '["a", ["b", "c"], "d"]';

mysql> SELECT JSON\_REMOVE(@j, '$[1]');

+-------------------------+

| JSON\_REMOVE(@j, '$[1]') |

+-------------------------+

| ["a", "d"] |

+-------------------------+

* 1. JSON\_APPEND(*json\_doc,path,val[,path,val]…*), 在指定的阵列之后再加上所输入的值,传回加上该值后的JSON文件
  2. JSON\_SET(*json\_doc,path,val[,path,val]…*),如果在指定的路径没有值时,会加上输入的值,或指定的路径有值时以输入的值更新之,并传回加上或更新后的JSON文件,如果如果文件中没法找到该路径,则传回NULL值,
  3. JSON\_INSERT(),在指定的路径加上输入的值, 并传回加上或更新值后的JSON文件,如果指定的路径己经有值则不做任何更改
  4. JSON\_REPLACE(),更新指定的路径的值,并传回加上或更新值后的JSON文件,如果找不到指定路径,则不做任何更改

以上三个函式的范例下:

# 文件中位于’$.a’路径的值原来为1,没有’$.c’路径;JSON\_SET(@j, '$.a', 10, '$.c', '[true, false]')指定将路径’$.a’的值改为10,在文件”根”之下加一个路径为’$.c’,值为[true,false]的阵列

mysql> SET @j = '{ "a": 1, "b": [2, 3]}';

mysql> SELECT JSON\_SET(@j, '$.a', 10, '$.c', '[true, false]');

+-------------------------------------------------+

| JSON\_SET(@j, '$.a', 10, '$.c', '[true, false]') |

+-------------------------------------------------+

| {"a": 10, "b": [2, 3], "c": "[true, false]"} |

+-------------------------------------------------+

# 文件中位于’$.a’路径己有值,文件”根”之下没有路径’$.c’, JSON\_INSERT(@j, '$.a', 10, '$.c', '[true, false]')不会改变文件路径’$.a’的值,而会在文件的”根”之下插入一个阵列identifier “c”,其值为输入的阵列

mysql> SELECT JSON\_INSERT(@j, '$.a', 10, '$.c', '[true, false]');

+----------------------------------------------------+

| JSON\_INSERT(@j, '$.a', 10, '$.c', '[true, false]') |

+----------------------------------------------------+

| {"a": 1, "b": [2, 3], "c": "[true, false]"} |

+----------------------------------------------------+

#文件中位于’$.a’路径原来的值为1,文件”根”之下没有路径’$.c’, JSON\_REPLACE(@j, '$.a', 10, '$.c', '[true, false]')会以10替换在文件根之下indetifier ’a’的值,不会在文件的”根”之下插入identifier “c”以及该函式所指定的阵列

mysql> SELECT JSON\_REPLACE(@j, '$.a', 10, '$.c', '[true, false]');

+-----------------------------------------------------+

| JSON\_REPLACE(@j, '$.a', 10, '$.c', '[true, false]') |

+-----------------------------------------------------+

| {"a": 10, "b": [2, 3]} |

+-----------------------------------------------------+

* 辅助性的函式

1. JSON\_QUOTE(*json\_val*),如果参数是一个字符串或数值,则在其前后加上双引号 - “使它成为JSON对象,如果字符串中有引号,则传回的结果中会在前后加上一跳脱码\,再加上双引号,例如:

mysql> SELECT JSON\_QUOTE('null'), JSON\_QUOTE('"null"');

+--------------------+----------------------+

| JSON\_QUOTE('null') | JSON\_QUOTE('"null"') |

+--------------------+----------------------+

| "null" | "\"null\"" |

+--------------------+----------------------+

mysql> SELECT JSON\_QUOTE('[1, 2, 3]');

+-------------------------+

| JSON\_QUOTE('[1, 2, 3]') |

+-------------------------+

| "[1, 2, 3]" |

+-------------------------+

1. JSON\_UNQUOTE(*val*),脱掉只传回的结果只取对象的内容值,例如:

mysql> SET @j = '"abc"';

mysql> SELECT @j, JSON\_UNQUOTE(@j);

+-------+------------------+

| @j | JSON\_UNQUOTE(@j) |

+-------+------------------+

| "abc" | abc |

+-------+------------------+

* 1. JOSN函式可以灵活的应用在处理于表中存JSON文件的各种场景,本文以建表时自JSON文件中找出指定的数据,和以指定两个表的JSON文件内的值做JOIN并更新表中的JSON文件为范例说明JSON函式的运用.
     + 在JSON文件建虚拟字段和索引以加强性能

在文件型数据库中一堆JSON文件的集称之为Collection(本文暂译为“集”),在MySQL则以表对应collection,存放一堆文件.然而要在一堆JSON文件中快速找到符合查询条件的那几个文件(条件还要以JSON函式到文件中找出的值来比对)得靠 5.7版所提供的虚拟字段,我们甚至可以用虚拟字段所建的索引,使查询能速找到到符合条的JSON文件,再以JSON函式自文件中取出查询所指定的数据.这也是X Protocol对Collection查询.时数据库在底层的InnoDB引擎中实现的方式.以下范例说明这个做法:

# 建一个表 – t1,内含一个JSON字段,和一个虚拟字段 – id,而 id的值是经由JSON\_EXTRACT函式算出的,”STORED”子句指定虚拟字段的值实际存放在表中,是占用空间的字段,由于我们把主键加在这个虚拟字段,它必需为STORED.

CREATE TABLE t1

(data JSON, id INT AS (JSON\_EXTRACT(data,”$.id”)) STORED,

PRIMARY KEY(id));

#如果加上的是非主键,则可定为不占空间的”VIRTUAL”,在下例中为该”id”虚拟字段加上索引时会自每一行的存放”data”字段的JSON文件中找出各文件的”根”之下的”series”成员的值,将这个值加到索引的B+树的各节点上,也就是表不实际存series的值,但是索引”series\_idx”的节点会存各文件的”series”成员的值

ALTER TABLE t1

ADD COLUMN series INT AS (JSON\_EXTRACT(data, "$.series")),

ADD INDEX series\_idx (series);

# 查询的WHERE条件以函式”JSON\_EXTRACT”找出各行”data”字段的JSON文件中的”series”成员的值来比对是否在3和5之间

SELECT data, series FROM t1 WHERE

JSON\_EXTRACT(data,"$.series") BETWEEN 3 AND 5;

series BETWEEN 3 AND 5;

# 传回的结果如下:

+------------------------------------+----+

| data | series |

+------------------------------------+----+

| {"series": 3, "inverted": 8} | 3 |

| {"series": 4, "inverted": 7} | 4 |

| {"series": 5, "inverted": 6} | 5 |

+------------------------------------+----+

# 由于JSON\_EXTRACT(data,"$.series")的值都存在于索引”series\_idx”,所以MySQL的优化器能利用该索引快速找到符合条件的JSON文件是位于那些行(这点可由EXPLAIN该SQL的产出证明).这种以函式索引支持查询是5.7版才出现的新功能.

EXPLAIN SELECT data FROM t1 WHERE JSON\_EXTRACT(data,"$.series") BETWEEN 3 AND 5

#传回的结果如下:

+----+----------------+--------+---------------+--------+…+------------------------------+

| id | select\_type | table | partitions | type | | Extra |

+----+----------------+--------+---------------+--------+…+------------------------------+

| 1 | SIMPLE | t1 | NULL | range | | Using index condition |

+----+----------------+--------+---------------+--------+…+------------------------------+

上列SQL命令相当于直接在查询的WHERE子句将series字段当成搜寻条件

select `test`.`t1`.`data` AS `data` from `test`.`t1`

where (`test`.`t1`.`series` between 3 and 5)

* 以JSON函式更新文件和做JOIN的条件:

#下列 t1和t2表都有一个名为data的JSON字段,我们想找出t1表中所有JSON文件”根”之下”series”成员的值和在t2表的JSON文件之”根”之下对象”b\_series”阵列中第一个位置(cell)之值相等的行,并将该行的JSON文件的”根”之下名为”inverted”的对象的值改为11-‘t2表的JSON文件之”根”之下”b\_series”阵列中第一个位置(cell)的值’,则更新数据的SQL命令如下:

UPDATE t1, t2

SET t1.data= JSON\_INSERT(t1.data,"$.inverted",

11 – JSON\_EXTRACT(t2.data,"$.b\_series[0]"))

WHERE

JSON\_EXTRACT(t1.data, "$.series")

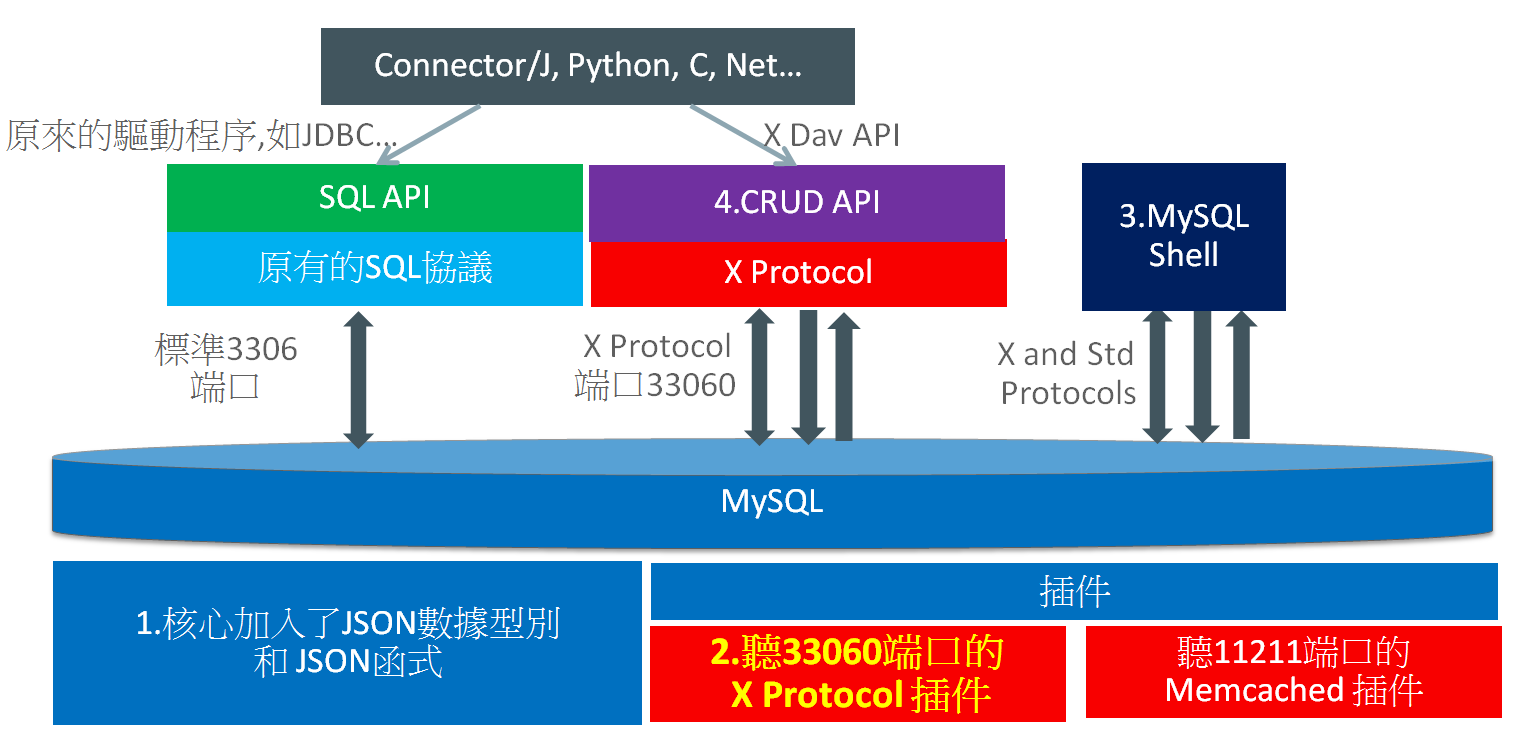
= JSON\_EXTRACT(t2.data,"$.b\_series[0]");

由以上的说明展示了,MySQL加上了JSON资料型别和23个JOSN函式,使我们能通过传统的SQL接口完整的操作JSON文件,并享有通过索引加快查询的性能.同时由于能将JSON文件存于InnoDB表中,原来关系型数据库所有的支持事务、多版本并行控制(MVCC)、以及备份监控等功能均能完整保留.另一方面数据库管理师也不需另外适应一套新的数据库技术.让MySQL的用户可以同时保有关系型数据库的严谨性,又能拥抱NoSQL技术的弹性和JSON文件不需要做JOIN,就能查出满足应用的需求的数据所带来的性能提升.

然而,MySQL还不仅止于此,还更进一步为我们提供一套全新的NoSQL接口 – X Protocol,使我们能以纯NoSQL的技术开发新的应用.

新增mysqlx插件让数据库支持X Protocol

MySQL自第一个正式发行版(5.7.9)就加上了新的新的JSON数据型别和JSON涵式,往支持NoSQL的路上迈了一大步.但是如果只能通过SQL界面开发NoSQL应用,还是显得有些削足适履,我们要的是能完全脱离SQL的影子,直接以NoSQL的方式和数据库互动.所以MySQLng项目在原有的(默认用3306端口)SQL界面之外(如下图2.的部份)再开发了一个X Plugin,使应用端能通过X Plugin的33060端口调用X Dav API,直接对MySQL数据库的JSON文件做增删改查等操作.所以现在的MySQL不仅是说SQL语言的关系型数据库,也是一个完整支持NoSQL的数据库.事实上,在MySQL 5.6以后,它就能通过Memcached的接口以key-value查询的方式和数据库互动(MySQL memcache的默认端口为11211),当查询方式不涉及复杂的JOIN和子查询时,藉由这个接口绕过SQL层,少了处理SQL的解析和优化的工作负荷,能使性能大辐提升,这也是以NoSQL的方式和数据库互动.现在更支持了JSON文件界面,使我们的NoSQL应用能有更高的可移植性.



* 安装MySQL X Plugin的

MySQL 默认并不具备处理X Protocol的能力,要让MySQL能支持X Protocol必需先装MySQL X 插件,在安装X插件之前MySQL就只能使用(默认听3306)SQL接口,和您熟悉的传统MySQL没什么差异(必需配置插件才能带出X Protocol应该是基于降低复杂度和bug出现的机会的考虑).

MySQL自 5.7.12版以后,您可在软件包的lib/plugin目录中.找到一个名为mysqlx.so(Windows版是mysqlx.dll)的文件,这就X插件的程序库.通过mysql客户端程序(通过默认3306端口),或MSQL Shell均可以安装这个插件:

* 以MySQL客户端程序安装MySQL X插件的命令如下(不一定要用root账户,以任何对mysql.plugin表有插入权的用户都可以):

$ mysql –u<user-name> -hlocalhost –P3306 –p <pasword>

mysql> INSTALL PLUGIN mysqlx SONAME 'mysqlx.so';

* 以MySQL Shell安装MySQL X插件:

在安装好MySQL Shell的环境中:

$ mysqlsh -u <user-name> -h localhost --classic --dba enableXProtocol

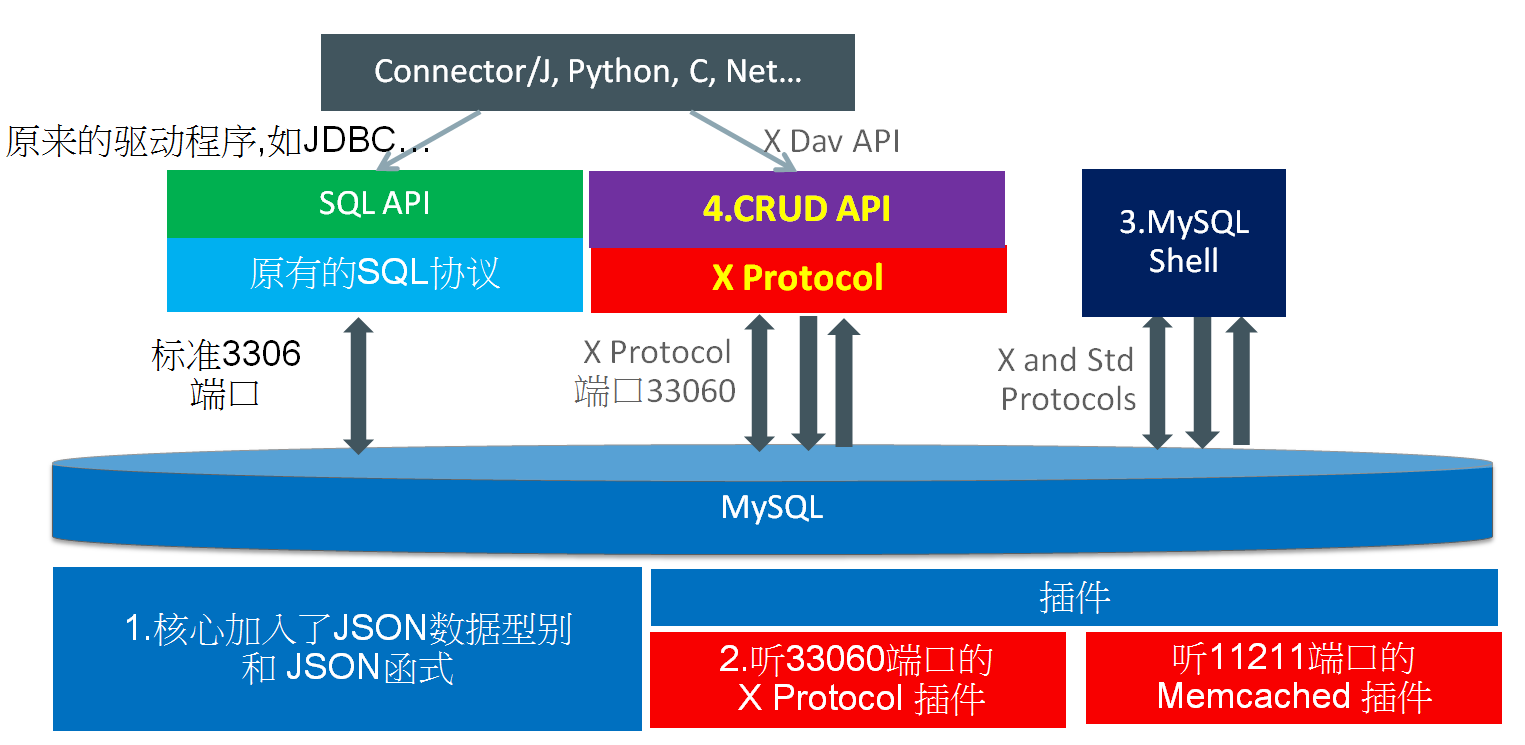
您可以在mysql客户端程序下以下命令,在回清单中找mysqlx以确认MySQL的X插件是否己装好:

mysql> show plugins;

当安装好了X插件,您就可以准备X protocol的客户端,正式将MySQL当成一个全然的文件型数据库.

### MySQL Shell - 操作MySQL文件数据的好帮手

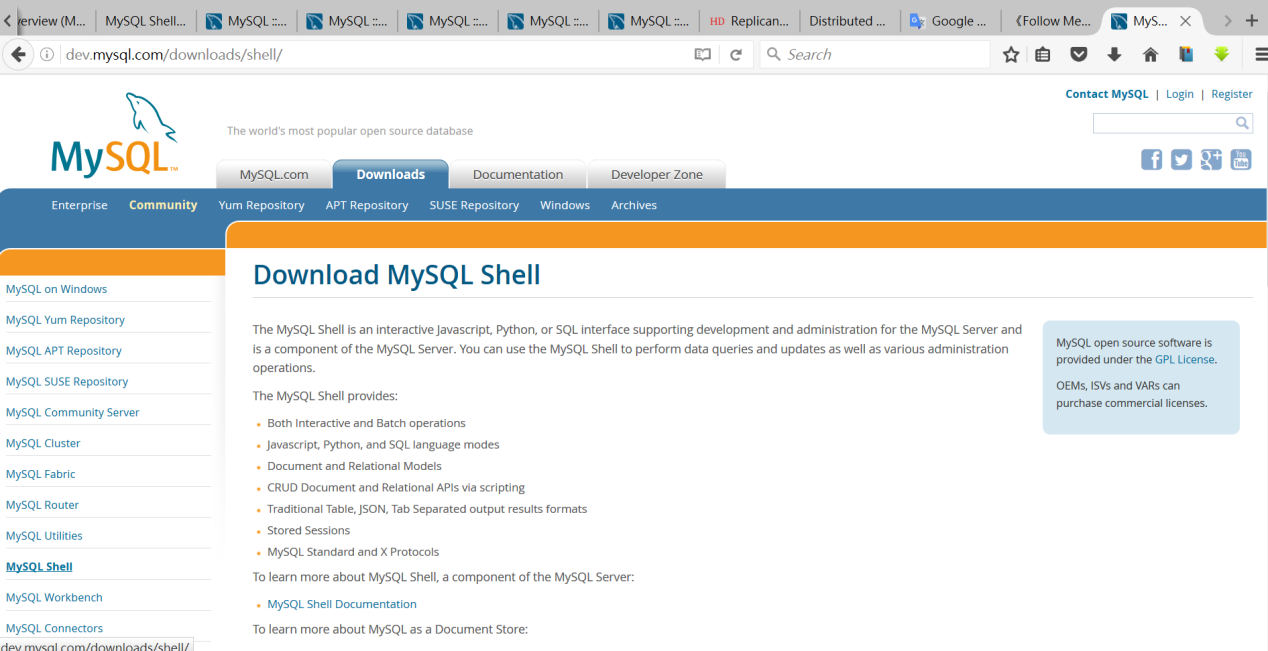
正如同mysql客户端程序为使用SQL接口(默认3306端口)的人提供方便操作的工具一样,MySQL Shell也为各类使用者(DBA,开发者…)提供一个好用的工具,以NoSQL的方式直接操作数据,管理数据库或做临时性的查询.



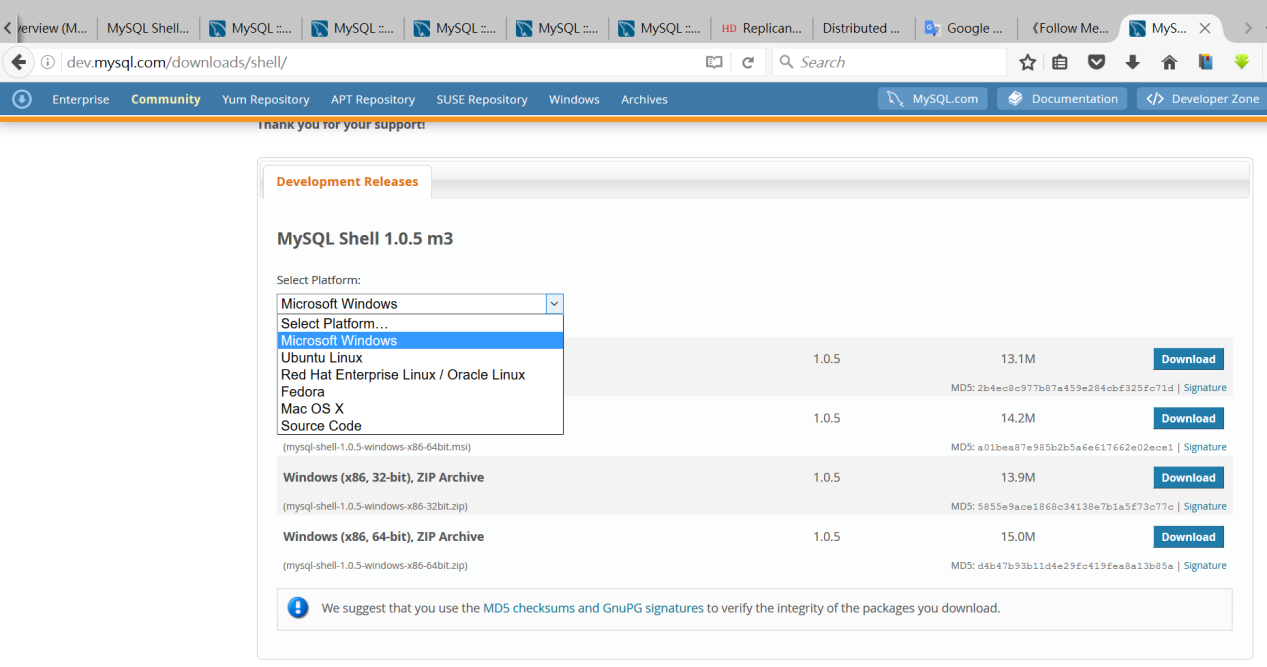
* 取得及安装MySQL Shell

MySQL Shell不在MySQL软件包内,需要单独自dev.mysql.com/downloads下载,其软件包

如下图所示,在MySQL的下载网页的左边的导航列中可选到MySQL Shell的选项.有一点先请读者们注意的是,目前(2016年底),MySQL Shell还是处于第三个beta(DMR)版,在它正式发行之前或许以后会有一些变动或者会加上更多功能.



在下载网页(https://dev.mysql.com/downloads/shell/)可依MySQL Shell的运行环境选择合适的版本



如果您选择ZIP Archive或 tar 包,解开来,不需安装就可以使用mysqlsh程序,但需要设定环境的路径变量,将MySQL Shell目录下的bin子目录的路径加到PATH变量中让我们在任何目录都可以激活mysqlsh.依操作系统的不同,您也可以选择rpm,deb或msi安装包,这些安装包运行很可以解开其软件,并且加上路径变量或将文件放到适当的位置(例如/user/bin),以方便运行.

* 运行MySQL Shell

在操作系统选单中选择MSQL Shell或终端机命令模式下运行msqlsh进入MySQL Shell.

$ mysqlsh

Welcome to MySQL Shell 1.0.4 Development Preview

Copyright (c) 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its

affiliates. Other names may be trademarks of their respective

owners.

Type '\help', '\h' or '\?' for help.

Currently in Javascript mode. Use \sql to switch to SQL mode and execute queries.

mysql-js>

* + 和mysql客户端程序一样,mysqlsh命令也可以加上参数.例如指定以什么帐户连入那个数据库、对话的模式(X, node,或classic)、语法(Javacript,Python,或SQL)、或内含批量作业命令的文件.如果要了解有那些选项,可以用--help或-h选项列示mysqlsh所选项和它们的说明:

$ mysqlsh –u<user-name> -h<host> -p

上列登入数据库的命令看起来是不是很熟悉?或是以下列命令指定URI同样能登入数据库:

$ mysqlsh –uri <user-name>:<password>@<host>:<port>/<db-name>

上列命令同样也是以指定的用户,登入到指定服务器的数据库.进入mysqlsh后可输入\q或\quit退出回到操作系统.

* + MySQL Shell有三种对话(Session)模式:
    - X session,这是默认的互动方式,这个模式无法执行SQL命令,只能运行Javascript或Python的命令,在MySQL的X Session以后可以发展成同时支持多个数据库联机,和数据库分片
    - Node session, 能执行SQL, Javascript或Python的命令,但是Node Session只能含一个数据库联机,以后也不会扩充成同时连多个数据库.
    - Classic session,不用x protocol,在这个模式下能用 SQL命令,但开发接口有限,不支持NoSQL 的CRUD操作,也不能对collection操作.
  + MySQL shell 可以有三种命令模供您选择,
    - Javascript : 默认的命令模式,也可以在mysqlsh命令加上--js选项进入Javascript模式,提示符为mysql-js> ,表示能接受Javascript的命令.
    - Python : 命令加上--py选项进入Python模式,提示符为mysql-py> ,能使mysqlsh接受Python命令

$ mysqlsh –u<user> -h<host> -p --py

* + - SQL:要能在MySQL Shell下SQL命令互动方式必需在MySQL Shell指定classic 或node session,再于命令行加上--sql选项, 这个模式的接口和传统的mysql客户端的操作方式机乎完全相同,一样是SQL命令,进入此模式的命令为:

$ mysqlsh –u<user-name> –p –h<host> --classic --sql;

* + - 如同在mysql客户端程序下键入help,查mysql客户端程序的命令,进入MySQL Shell后可以\?或\help查mysqlsh的操作命令,所有的mysqlsh命令都以\开头,例如\c是建立数据库联机.
  + 在MySQL Shell操作JSON文件

MySQL Shell为JSON文件的操作提供所需要的命令集,这些命令都是基于X Dav API的接口而开发出来的有关,MySQL Shell的X Dav API的详情请参考http://dev.mysql.com/doc/dev/mysqlsh-api-Javascript/

以Javascript命令为例,以下说明以mysqlsh 在JSON文件生命周期中的各种操作命令:

* + - 建立Schema和更换Schema:

MySQL Shell有几个默认的对象,例如session, db…,您可在进入mysqlsh以后键入session查 看目前联机对话的内容,下例显示目前的互动方式为X Session,以root账户登入位于127.0.0.1,使用33060端口:

mysql-js> session

<XSession:root@127.0.0.1:33060>

在session对象上建立schema的函式,以下范例建一个名为”mydemo的Schema:

mysql-js> session.createSchema('mydemo');

<Schema:mydemo>

上列命令相当于在mysql命令程序下达CREATE DATABASE mydemo.

下例可查看我们之前建立的mydemo schema是否存在?目前数据库有那些schema?

schema:

mysql-js> session.getSchemas();

[

<Schema:information\_schema>,

<Schema:acmug>,

<Schema:mydemo>,

<Schema:mysql>,

<Schema:mysqlnews>,

<Schema:performance\_schema>,

<Schema:sakila>,

<Schema:sys>,

<Schema:world>,

<Schema:world\_x>

]

更换使用中的schema,则以mysqlsh命令\use(或者\u)就行了,例如:

mysql-js> \u mydemo

Schema `mydemo` accessible through db.

当然,进入mysqlsh时所指定的参数中也可以指定建立联机后要用那一个database(schema),例如:

­­

$ mysqlsh --uri root@127.0.0.1:33060/mydemo

Creating an X Session to root@127.0.0.1:33060/mydemo

Enter password:\*\*\*\*\*\*\*\*

Default schema `mydemo` accessible through db.

Welcome to MySQL Shell 1.0.4 Development Preview

Copyright (c) 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its

affiliates. Other names may be trademarks of their respective

owners.

Type '\help', '\h' or '\?' for help.

Currently in Javascript mode. Use \sql to switch to SQL mode and execute queries.

mysql-js>

上例中登入的参数中,URI的最后一段/mydemo指定用mydemo为联机的默认schema,进入mysqlsh后响应的内容中有一段” default schema `mydemo` accessible through db”说明了这一点.进入了mysqlsh后还可以键入db查证一下mysqlsh的默认对象db是否放在mydemo上:

mysql-js> db

<Schema:mydemo>

* + - 在当前schema下操作Collection和Document:

在说明mysqlsh对collection的操作前,先简单的说明Document数据库的基本观念:

一个collection是一堆JSON文件的集,相当于关系型数据库的表.JSON Document代表(或描述)一个事情或是一笔记录,相当于关系型数据库在表中的一个row.

* + - * + 建立Collection:
      * 通过默认的db对象之createCollection函式,在目前所在的schema之下可以建立collection,db对象代表当下的session所位在的schema,下例为建一个名为collection1的Collection

mysql-js> db.createCollection('collection1');

<Collection:collection1>

* + - * 也可以在参数中加上schema名称,这个功能使用户可以在任何schema下为mydemo建collection:

mysql-js> db.createCollection('mydemo.mycollection');

<Collection:mydemo.mycollection>

只要有权限在指定的schema下新建表的账户,均可在位于任何数据库时运行createCollection以参数<schema-nam>.<collection-name> 的方式为其他的schema加上collection.

* + - * + 加上Document:
      * 有了collection以后,可调用该colleciton 对象的add函式将JSON文件加入collection中:

mysql-js> db.collection1.add({"name": {"lastname": "Tu","firstname": "Ivan"},"height": 180,"weight": 68,"talents": ["swimming","dancing","programming"]});

Query OK, 1 item affected (0.02 sec)

* + - * 再加一个JSON文件:

mysql-js> db.collection1.add({"name": {"lastname": "Wang","firstname": "James"},"height": 170,"weight": 80,"date of birth":"1990-12-30","talents": ["make money","dancing"]});

Query OK, 1 item affected (0.00 sec)

* + - * 查现在己加了多少文件到collection1:

mysql-js> db.collection1.find();

[

{

"\_id": "64b87d23a162d211645db86b2339c6c8",

"date of birth": "1990-12-30",

"height": 170,

"name": {

"firstname": "James",

"lastname": "Wang"

},

"talents": [

"make money",

"dancing"

],

"weight": 80

},

{

"\_id": "b85d1476a062d211645db86b2339c6c8",

"height": 180,

"name": {

"firstname": "Ivan",

"lastname": "Tu"

},

"talents": [

"swimming",

"dancing",

"programming"

],

"weight": 68

}

]

1. documents in set (0.00 sec)
   * + 如果仔细看上列的输出您可以发掘几个有趣的现象:
       - 虽然前面加Document时,并未在文件中加上”\_id”对象,所有的文件都自动加上一个名为”\_id”的对象,存放自动生成的uuid值.
       - 文件的结构能自由定义不象关系数据库需事先定一个固定的表结构,上例中第二个文件比第一个文件多了一个”data of birth”对象,”talents”阵列的数目也比第一个少了一个,这代表JSON文件不需事先定义数据结构,对象名称(identifier)就能表达JSON文件内各对象的含义.
     + 另一边,由mysql客户端程序查看看数据库在引擎层面发生了什么事?

mysql> show create table collection1\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Table: collection1

Create Table: CREATE TABLE `collection1` (

`doc` json DEFAULT NULL,

`\_id` varchar(32) GENERATED ALWAYS AS (json\_unquote(json\_extract(`doc`,'$.\_id'))) STORED NOT NULL,

PRIMARY KEY (`\_id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4

1 row in set (0.00 sec)

上例显示每一个collection都对应一个InnoDB的表,表有两个字段 – doc字段的资料型别是json,另一个\_id字段是由json\_extract函式生成的STORED(自函式算出值实际存放于表中,以后查询时不需再算一次)字段,它的值是在doc字段中找名为\_id的物件之值,并且将主键加在这个\_id字段上。

* 我们也能为collection的weight加上(非唯一)次要索引:

mysql-js> db.collection1.createIndex("weightIdx").field("weight","INTEGER",false).execute();

Query OK (0.06 sec)

* 另一方面,加上索引后在InnoDB表结构如下所示:

mysql> show create table collection1\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Table: collection1

Create Table: CREATE TABLE `collection1` (

`doc` json DEFAULT NULL,

`\_id` varchar(32) GENERATED ALWAYS AS (json\_unquote(json\_extract(`doc`,'$.\_id'))) STORED NOT NULL,

`$ix\_i\_2D8FA3E83FC26B3A572C94D67A6F56CAC0CA1EAB` int(11) GENERATED ALWAYS AS (json\_extract(`doc`,'$.weight')) VIRTUAL,

PRIMARY KEY (`\_id`),

KEY `weightIdx` (`$ix\_i\_2D8FA3E83FC26B3A572C94D67A6F56CAC0CA1EAB`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4

1 row in set (0.00 sec)

由此证明X Dav API会因为这个collection1.createIndex函式在数据库的collection1表上多加了一个名为$ix\_i\_<uuid>的虚拟生成字段,其值是由json\_extract文件中的weight对象取出的值所构成的,并且在其上加了索引-weightIdx

* 您也可以对collection进行条件查询,甚至动能挷定值于条件中,下例为自Collection找出weight > 70的文件

mysql-js> db.collection1.find("weight > :kg").bind("kg",70);

[

{

"\_id": "64b87d23a162d211645db86b2339c6c8",

"date of birth": "1990-12-30",

"height": 170,

"name": {

"firstname": "James",

"lastname": "Wang"

},

"talents": [

"make money",

"dancing"

],

"weight": 80

}

]

1 document in set (0.00 sec)

由于查询条件里有weight对象,执行计划可能会用到前一段所建的weightIdx的索引.

* 也可以指定要由查询的结果中只传回指定的对象,下例显示不需要看整份JSON文件,只要传回name和height对象时可以fields函式指定要传回的对象

mysql-js> db.collection1.find("weight > 70").fields(["name","height"]);

[

{

"height": 170,

"name": {

"firstname": "James",

"lastname": "Wang"

}

}

]

1 document in set (0.00 sec)

* 最后不需要这个collection时,可以删除这个collection,drop collection得用session对象的dropCollection函式,它的必要参数有两个 – schema名称和collection名称,删除后该schema就查不到该collection了

mysql-js> session.dropCollection("mydemo","collection1");

Query OK (0.02 sec)

mysql-js> db.getCollections()

[

]

* 一如SQL可编角本用在mysql客户端程序运行批量作业,我们也可以编一套Javascript或Python脚本在MySQL Shell 运行批量作业,范例如下:

建立一个名为Transaction-sample.js 的角本, 这个角本会运行1000次循环,每次循环插入5个JSON文件,每个循环都是一个事务,也就是一个事务提交5个JSON文件

//汇入mysqlx模块,本地变量-mysqlx指向mysqlx对象,由mysqlx的getSession函式建立数据库联机

var mysqlx = require('mysqlx').mysqlx;

// 建立数据库联机

var session = mysqlx.getSession( {

host: 'localhost', port: 33060,

dbUser: 'root', dbPassword: '<password>' } );

// 将schema移到my\_test

var db = session.getSchema('my\_test');

// 在my\_test schema之下建立一个新的 collection – mycollection,

//如果该collection在此之前己存在则先drop它

session.dropCollection('my\_test','mycollection');

db.createCollection('mycollection');

//取一个collection对象 – myColl,各JSON文件均以其add函式加入collection该中

var myColl = db.getCollection('mycollection');

var iter;

// 循环运行1000次,每次先以session.startTransaction()定事务的起始点

// 循环插入5个文件,如果运行中间有错误或被强制中断,

//之前提交的事务会存在数据库中,做到中间的事务会回滚

//所以collection中存在的文件数目会是5的倍数(实现事务的数据完整性),

//由此证明X Dav API是支持事务和数据完整性的

// 最后以session.commit()结束并提交事务

try {

for (iter = 0; iter < 1000; iter++) {

session.startTransaction();

myColl.add({name: 'Sunny', age: 23, height: 1.3, weight: 73.3}).execute();

myColl.add({name: 'Albert', age: 24, height: 1.4, weight: 74.3}).execute();

myColl.add({name: 'Bred', age: 25, height: 1.5, weight: 75.3}).execute();

myColl.add({name: 'Connie', age: 26, height: 1.6, weight: 76.3}).execute();

myColl.add({name: 'David', age: 27, height: 1.7, weight: 77.3}).execute();

// Commit the transaction if everything went well

session.commit();

print('Data inserted successfully.\n');

}

}

catch (err) {

// 如果有错误,则回滚未完成的事务

session.rollback();

// Printing the error message

print('Data could not be inserted: ' + err.message);

}

将上列脚本送到MySQL Shell运行,结果如下:

$ mysqlsh -f transaction-sample.js

Data inserted successfully.

Data inserted successfully.

…

Data inserted successfully.

//最后会出现1000次Data inserted successfully.

登入查看运行的结果:

mysql-js> db.mycollection.find()

[

{

"\_id": "ffe5ac80c362d2114072b86b2339c6c8",

"age": 24,

"height": 1.4,

"name": "Albert",

"weight": 74.3

},

{

"\_id": "fffaab7ec362d2114072b86b2339c6c8",

"age": 27,

"height": 1.7,

"name": "David",

"weight": 77.3

}

…

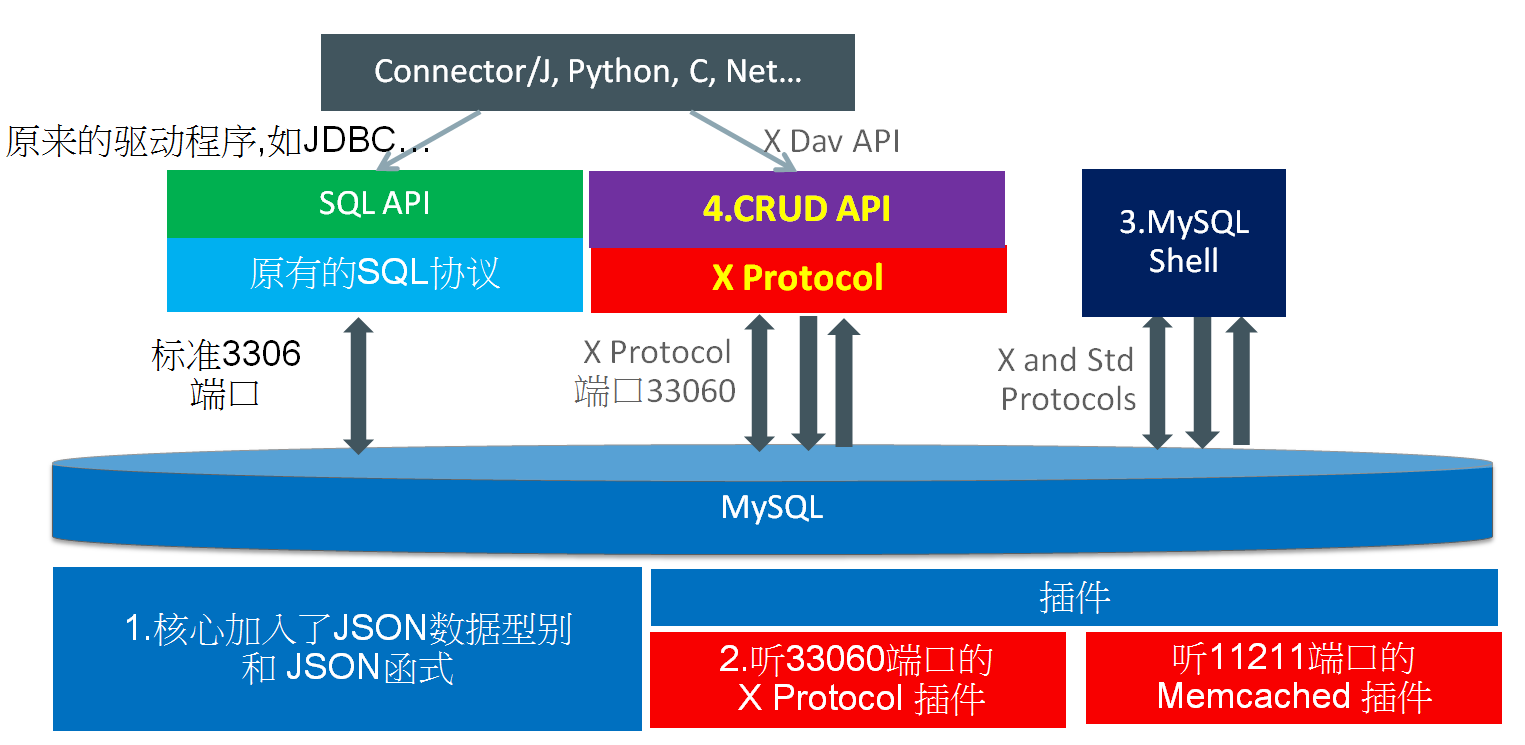
]

5000 documents in set (0.00 sec)

上列例中建数据库联机不是用Connection对象,而是Session对象,在Connection之上引进Session的概念这也透露着X Dav API将朝向支持一个Session可能包含多个Connection,每个Connection分别连到不同的数据库,而能支持数据库分片断(data shard)的方向发展

Connector实现X Dav API支持多种应用程序以JSON和MySQL互动

文行到此,整个MySQL as a Document Store的技术拼图只剩最后一片- 使应用层能通过X Protocol,运行CRUD的函式,直接操作位于数据库内的JSON文件.如下图4.所示,MySQL在各种新版本的应用程序链接器(包括Connector/J, Connectior/NET, Connectior/C++等)加上了新的支持X protocol的模块,以,让应用程序能直接调用这些模块中实现X Dav API的接口的函式,通过X protocol直接使用和更新数据库的JSON文件.这些链接器还在持续完善中,目前支持X Protocol的链接器有Connector/J,Connector/Python,Connector/NET,Connector/C++ 和Connector/Nodejs.



本文的范例以一个Java应用程序 - MySQLStore.java(这个Java程序会插入三个JSON文件到mycollection中,并找出name为”Sakila”,age小于20的文件，并列示出来)为您演示如何调用Connector/J的X Dav API接口.于有兴趣测试这个功能的读者可以按以下步骤配置Java运行环境，和编写Java程序来体验一下Document based应用..

* 准备JSON文件数据库的Java开发环境

要开发及运行MySQLStore Java程序除了要有JDK,还要在合适的Java驱动器,Connector/J 6.0以后的版本才能支持JSON文件数据库,能要到MySQL的软件下载网页(网址是dev.mysql.com/downloads/Connector/J/),点选”Development Releases”页签,才能找到Connector/J 6.0.x版的.jar包.下载这里的压缩文件后做这三个动作后,基本上就完成了开发环境的准备了.

* + 解压缩该文件,
  + 找到mysql-connector-java-6.0.<x>-bin.jar
  + 并将这个Java .jar文件的位置加到操作系统环境变量 – CLASSPATH,或在Java IDE中添加一个链接库给这个.jar文件
* MySQL 文件数据应用的Java程序

package mysqldocstore;

import com.mysql.cj.api.x.\*;

import com.mysql.cj.x.MysqlxSessionFactory;

import com.mysql.cj.x.json.\*;

//以上import的三个package/class都是MySQL Connetor/J 6.0.x 所新增的

public class MySQLdocStore {

public static void main(String[] args) {

String url = "mysqlx://localhost:33060/my\_test?user=itu&password=<password>";

//通过mysqlx建立数据库对话(Session)对象,数据库主机位于localhost

//数据库听33060端口,用户名为itu,schema为my\_test(可不在URI指定Schema)

//<password>要替换成数据库的root密码

XSession mySession = new MysqlxSessionFactory().getSession(url);

//开启一个事务

mySession.startTransaction();

//将位置移到your\_test schema

Schema myDb = mySession.getSchema("your\_test");

//找一个名为mycollection的 Collection,并以myColl变量指向该物件

Collection myColl = myDb.getCollection("mycollection");

//插入三个JSON文件

myColl.add("{\"name\":\"Sakila\", \"age\":15}").execute();

myColl.add("{\"name\":\"Susanne\", \"age\":24}").execute();

myColl.add("{\"name\":\"Mike\", \"age\":39}").execute();

//提交事务

mySession.commit();

//在mycolleciton中找文件,条件是name是’Sakila’字符串,且age小于20

DocResult docs = myColl.find(name like :name AND age < :age").bind("name", "Sakila").bind("age", 20).execute();

//取出查出的文件以doc变量指向该对象,如果找不到符合条件的文件,则返回NULL

DbDoc doc = docs.fetchOne();

//列示查到的JSON文件

System.out.println("Age below 20 is "+doc);

}

}

* 运行结果,由于我将MySQLdocStore.class和msql-connector-java-6.0.<x>-bin.jar包在jar包中以方便布署,所以运行时要指定MySQLdocStore.jar.运行后,在数据库会插入三个JSON文件,最后列示一个JSON文件,其age对象的值小于20.

$ java –jar MySQLdocStore.jar

Age below 20 is {

"\_id" : "bb8c3bf0242a44a0a49767e3dcee7cc9",

"age" : 15,

"name" : "Sakila"

}

* 由MySQL Shell查看mycollection Collection,可看到在mycollection中有三个JSON文件,都是在MySQLdocStore.class程序插入的.

mysql-js> db.mycollection.find();

[

{

"\_id": "2f5a69049e0744f4b1f3abd9bbbbd2dd",

"age": 39,

"name": "Mike"

},

{

"\_id": "bb8c3bf0242a44a0a49767e3dcee7cc9",

"age": 15,

"name": "Sakila"

},

{

"\_id": "d9d3e1c86a3643be93e65ac1981c659f",

"age": 24,

"name": "Susanne"

}

]

1. documents in set (0.00 sec)

* 由SQL界面查mycollection表,也可在mycollection表中看到这些行含有JSON文件.

mysql> select \* from mycollection\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

doc: {"\_id": "2f5a69049e0744f4b1f3abd9bbbbd2dd", "age": 39, "name": "Mike"}

\_id: 2f5a69049e0744f4b1f3abd9bbbbd2dd

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

doc: {"\_id": "bb8c3bf0242a44a0a49767e3dcee7cc9", "age": 15, "name": "Sakila"}

\_id: bb8c3bf0242a44a0a49767e3dcee7cc9

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

doc: {"\_id": "d9d3e1c86a3643be93e65ac1981c659f", "age": 24, "name": "Susanne"}

\_id: d9d3e1c86a3643be93e65ac1981c659f

3 rows in set (0.00 sec)

结论

我认为MySQL演化成JSON文件数据库是MySQL近年来最重要的创新之一,这套技术促成MySQL将NoSQL和关系数据库的优势融于一炉,让广大的MySQL用户能在拥抱新的NoSQL技术的同时,不需学另外一套新的数据库,磨合陌生的新技术.用户可在他们己经熟悉的MySQL上以Schema less数据结构支持Dev/Ops的方法开发,进而加速应用面市,满足应用业务上占得市场先机同时,又能享有关系数据库己经很成熟的数据完整性、支持事务、MVCC(多版本并行控制)和崩溃回复的优点.另一方面DBA们也能使用和原来MySQL相同的工具为数据库备份、调优、监控和安控管制.MySQL这项与时俱进的发展使得MySQL所有的从业人员都能轻松因应IT大环境的改变.让MySQL应用开发者、DBA和应用业务拥有者三方都能同蒙其利.

对开发者而言,除了MySQL数据库由核心到MySQL Shell工具和各种应用的Connector上推出支持JSON的功外,我们还需要在应用开发环境上提供更丰富的Framework,以进一步使MySQL的开发者能快速而方便的开发应用.由于许多Connector的技术(例如Java, Node.js等)己经有很多现成的Framework支持JSON,让我们在这方面的发展有了扎实的基础.例如使用Node.js的开发者或许知道(或用过)MEAN Framework (MondoDB, Express, Angularjs, Node.jd),它能使开发者以更有效率、更容易的开发互联网应用.经过Connector/Node.js开发人员的努力, 现在MySQL也支持MEAN Framework,只是这是组合的含义是MySQL, Ecpress, Angular.js, Node.js了(相关讯息和实作范例可参考http://insidemysql.com/develop-by-example-document-store-working-with-express-js-angularjs-and-node-js/),相信随着MySQL as a document store的功能持续的完善,将有更多的Framework和IDE(整合开发环境)会支持这个技术堆栈,或许MySQL会在LAMP风潮之后为IT界来另一波高潮.