**[Sqlite3中存储类型和数据类型结合文档解析。](https://www.cnblogs.com/jycboy/p/android.html)**

**sqlite3是个很小的数据库，运行在手机，机顶盒上....那它就不可能像musql，sqlserver那么规范，有很多的数据类型，之前我也以为它定义了很多数据类型，其实不是他就5个存储类,那么多数据类型是根据一整套严谨的规则映射的！！还有什么char，varchar其实都是没有的..下面将结合文档详细讲解，相信看完你会了解更多，其实主要就是翻译文档....**

**sqlite官网：http://www.sqlite.org/**

**Sqlite3数据类型**

大多数的数据库引擎（到现在据我们所知的除了sqlite的每个sql数据库引擎）都使用静态的、刚性的类型，使用静态类型，数据的类型就由它的容器决定，这个容器是这个指被存放的特定列。   
Sqlite使用一个更一般的动态类型系统，sqlite中，值的数据类型跟值本身相关，而不是与它的容器相关。Sqlite的动态类型系统和其他数据库的更为一般的静态类型系统相兼容，但同时，sqlite中的动态类型允许它能做到一些传统刚性类型数据库所不可能做到的事。

//这说的有点抽象，简单的意思就是存储的值和列的类型是分开的，列的类型可以有很多，而存储的类型就5个，这也是为什么sqlite这么小却能支持那么多数据类型的原因。就像映射一样，sqlite有一套自己的数据类型到存储类型的映射。

**1存储类型和数据类型**

**1.0存储类型**

每一个值存储在一个SQLite数据库（或操作的 数据库引擎）都有以下的一个存储类(真正存在数据库中)：

·       NULL值是null。

·       INTEEGER  值是有符号整形，根据值的大小以1,2,3,4,6或8字节存放

·       REAL，值是浮点型值，以8字节IEEE浮点数存放。

·       TEXT，值是文本字符串，使用数据库编码（UTF-8，UTF-16BE或者       UTF-16LE）存放

·       BLOB，只是一个数据块，完全按照输入存放（即没有准换）  
  
从上可以看出存储类比数据类型更一般化。比如INTEGER存储类，包括6中不同长度的不同整形数据类型，这在磁盘上造成了差异。但是只要INTEGER值被从磁盘读出进入到内存进行处理，它们被转换成最一般的数据类型（8-字节有符号整形）。   
  
Sqlite v3数据库中的任何列，除了整形主键列，可以用于存储任何一个存储类型的值。

· sql语句中的中所有值，不管它们是嵌入在sql文本中或者是作为参数绑定到一个预编译的sql语句，它们的存储类型都是未定的。在下面描述的情况中，数据库引擎会在查询执行过程中在数值（numeric）存储类型（INTEGER和REAL）和TEXT之间转换值。

**1.1布尔类型**

**Sqlite没有单独的布尔存储类型，它使用INTEGER作为存储类型，0为false，1为true**

**1.2 Date和Time Datatype**

**Sqlite没有另外为存储日期和时间设定一个存储类集，内置的sqlite日期和时间函数能够将日期和时间以TEXT，REAL或INTEGER形式存放   
l).  TEXT 作为IS08601字符串（"YYYY-MM-DD HH:MM:SS.SSS"）   
2).  REAL 从格林威治时间11月24日，4174 B.C中午以来的天数   
3).  INTEGER 从 1970-01-01 00:00:00 UTC以来的秒数   
程序可以任意选择这几个存储类型去存储日期和时间，并且能够使用内置的日期和时间函数在这些格式间自由转换  
  
2.0 类型近似 (sqlite的数据集合)**

**为了使sqlite和其他数据库间的兼容性最大化，sqlite支持列上“类型近似”的观点，列的类型近似指的是存储在列上数据的推荐类型。这里必须记住一点，这个类型是被推荐，而不是必须的。任何列仍然能存储任意类型的数据。只是一些列，给予选择的话，将会相比于其他的一些类型优选选择一些存储类型，这个列优先选择的存储类型被称为它的“近似”。   
每个sqlite3数据库中的列都被赋予下面类型近似中的一种：   
1).  TEXT   
2).  NUMERIC   
3).  INTEGER   
4).  REAL   
5).  BLOB   （BLOB过去叫做NONE，不过这个词更容易混淆“没有近似”）  
  
具有TEXT近似的列可以用NULL，TEXT或者BLOB类型存储数据。如果数值数据被插入到具有TEXT近似的列，在被存储前被转换为文本形式   
一个有NUMERIC近似的列可以使用所有5中存储类来存储数据。当文本数据被存放到NUMERIC近似的列中，这个文本的存储类被转换到INTEGER或REAL（根据优先级顺序），如果这个转换是无损的话。对于TEXT和REAL存储类间的转换，如果数据的前15位的被保留的话sqlite就认为这个转换是无损的、可反转的。如果TEXT到INTEGER或REAL的转换不可避免的会造成损失，那么数据将使用TEXT存储类存储。不会企图去转换NULL或BLOB值。   
  
一个字符串可能看起来像浮点数据，有小数点或指数符号，但是只要这个数据可以使用整形存放，NUMERIC近似就会将它转换到整形。比如，字符串 '3.0e+5'存放到一个具有NUMERIC近似的列中，被存为300000，而不是浮点型值300000.0。   
具有INTEGER近似的列和具有NUMERIC近似的列表现相同。它们之间的差别仅处于转换描述上。   
具有REAL近似的列和具有NUMERIC近似的列一样，除了它将整形数据转换成浮点型形式。   
具有BLOB近似的列不会优先选择一个存储列，也不会强制将数据从一个存储类转换到另外一个类。**

**2.1列近似的决定因素 （映射关系）**  
  
列的近似由这个列的声明类型所决定，根据下面的规则：   
  
1） 如果声明类型包含”INT”字符串，那么这个列被赋予INTEGER近似   
  
 2）如果这个列的声明类型包含”CHAR”，”CLOB”，或者”TEXT”中的任意一个，那么这个列就有了TEXT近似。注意类型VARCHAR包含了”CHAR”字符串，那么也就被赋予了TEXT近似   
  
 3） 如果列的声明类型中包含了字符串”BLOB”或者没有为其声明类型，这个列被赋予BLOB近似   
  
 4）如果列的声明类型包含 “REAL”,”FLOA”,”DOUB”中任何一个，那么这个列就是REAL近似。

5）其他的情况，列被赋予NUMERIC近似   
\*\*上面规则顺序对于决定列的近似很重要。一个列的声明类型为”CHARINT”的话同时会匹配规则1和2，但是第1个规则优先，所以这个列的近似将是INTEGER。

**2.2近似名称例子 （映射函数）**  
  
         （下面这个表显示了多少来自更传统的SQL操作的普通数据类型名称，使用上一节中的5个规则，被转换到近似类型。这个表只显示了sqlite能够接受的数据类名称的一个子集。注意到跟随类型名的圆括号内的数值参如：”VARCHAR(255)”）被sqlite忽略—sqlite不在字符串、BLOBS或者数值的长度上强加任何长度限制（除了一个全局的SQLITE\_MAX\_LENGTH限制）。//就是说加括号也没用，所以不用加 ，之前我还一直加。

|  |  |  |
| --- | --- | --- |
| **Example Typenames From The CREATE TABLE Statement or CAST Expression（输入的类型）** | **Resulting Affinity(近似的结果)** | **Rule Used To Determine Affinity** |
| INT INTEGER TINYINT SMALLINT MEDIUMINT BIGINT UNSIGNED BIG INT INT2 INT8 | INTEGER | 1 |
| CHARACTER(20) VARCHAR(255) VARYING CHARACTER(255) NCHAR(55) NATIVE CHARACTER(70) NVARCHAR(100) TEXT CLOB | TEXT | 2 |
| BLOB *no datatype specified* | BLOB | 3 |
| REAL DOUBLE DOUBLE PRECISION FLOAT | REAL | 4 |
| NUMERIC DECIMAL(10,5) BOOLEAN DATE DATETIME | NUMERIC | 5 |

  注意到声明类型为”FLOATING POINT”将被赋予INTEGER近似，而不是REAL近似，因为”INT”在”POINT”后边。

声明类型为”STRING”的将被赋予NUMERIC，不是TEXT。（不动可以看2.1的规则，这是用了规则5）  
  
（从上面可以看出，sqlite3只是从声明类型字符串中去查找它知道的声明类型，比如”XINT”将被赋予INTEGER近似因为这个字符串里面有”INT”，所以这里并不需要一个单独的正确的声明类型，而是只要声明类型字符串里面包含了sqlite所知道的声明类型即可）

**2.3 列近似操作例子**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | CREATE TABLE t1(      t  TEXT,     -- text affinity by rule 2      nu NUMERIC,  -- numeric affinity by rule 5      i  INTEGER,  -- integer affinity by rule 1      r  REAL,     -- real affinity by rule 4      no BLOB      -- no affinity by rule 3  );    -- Values stored as TEXT, INTEGER, INTEGER, REAL, TEXT.  INSERT INTO t1 VALUES('500.0', '500.0', '500.0', '500.0', '500.0');  SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;  text|integer|integer|real|text    -- Values stored as TEXT, INTEGER, INTEGER, REAL, REAL.  DELETE FROM t1;  INSERT INTO t1 VALUES(500.0, 500.0, 500.0, 500.0, 500.0);  SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;  text|integer|integer|real|real    -- Values stored as TEXT, INTEGER, INTEGER, REAL, INTEGER.  DELETE FROM t1;  INSERT INTO t1 VALUES(500, 500, 500, 500, 500);  SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;  text|integer|integer|real|integer    -- BLOBs are always stored as BLOBs regardless of column affinity.  DELETE FROM t1;  INSERT INTO t1 VALUES(x'0500', x'0500', x'0500', x'0500', x'0500');  SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;  blob|blob|blob|blob|blob    -- NULLs are also unaffected by affinity  DELETE FROM t1;  INSERT INTO t1 VALUES(NULL,NULL,NULL,NULL,NULL);  SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;  null|null|null|null|null |

　　这是文档中给的例子。。。

**3.0 比较表达式**  
  
        Sqlite v3有一系列有用的比较操作符，包括 "=", "==", "<","<=", ">", ">=", "!=","<>", "IN", "NOT IN", "BETWEEN","IS", 和"IS NOT"   
**3.1 排序**  
  
比较操作的结果基于操作数的存储类型，根据下面的规则：   
  
l  存储类型为NULL的值被认为小于其他任何的值（包括另一个存储类型为NULL的值）   
  
l  一个INTEGER或REAL值小于任何TEXT或BLOB值。当一个INTEGER或REAL值与另外一个INTEGER或REAL值比较的话，就执行数值比较   
  
l  TEXT值小于BLOB值。当两个TEXT值比较的时候，就根据序列的比较来决定结果   
  
l  当两个BLOB值比较的时候，使用memcmp()来决定结果   
  
简单的说：BLOB > TEXT > INTEGER/REAL > NULL  
  
**3.2 比较操作数的近似（Affinity）**  
  
        Sqlite可能在执行一个比较之前会在INTEGER，REAL或TEXT之间转换比较值。是否在比较操作之前发生转换基于操作数的近似（类型）。

注意每一个表的列都有近似列，但表达式不一定有。  
操作数近似（类型）由以下的规则决定：   
l  对一个列的简单引用的表达式与这个列有相同的affinity，注意如果X和Y.Z是列名，那么+X和+Y.Z均被认为是用于决定affinity的表达式   
  
l  一个”CAST(expr as type)”形式的表达式与用声明类型为”type”的列有相同的affinity   
  
l  其他的情况，一个表达式为NONE affinity   
  
3.3 类型转换之前的比较   
         只有在转换是无损、可逆转的时候“应用近似”才意味着将操作数转换到一个特定的存储类。近似在比较之前被应用到比较的操作数，遵循下面的规则（根据先后顺序）:   
l  如果一个操作数有INTEGER，REAL或NUMERIC近似，另一个操作数有TEXT或BLOB近似或没有近似，那么NUMERIC近似被应用到另一个操作数   
  
l  如果一个操作数有TEXT近似，另一个有没有近似，那么TEXT近似被应用到另一个操作数   
  
l  其他的情况，不应用近似，两个操作数按本来的样子比较   
  
表达式"aBETWEEN b AND c"表示两个单独的二值比较” a >= b AND a <= c”，即使在两个比较中不同的近似被应用到’a’。

**3.4比较例子**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | CREATE TABLE t1(      a TEXT,      -- text affinity      b NUMERIC,   -- numeric affinity      c BLOB,      -- no affinity      d            -- no affinity  );    -- Values will be stored as TEXT, INTEGER, TEXT, and INTEGER respectively  INSERT INTO t1 VALUES('500', '500', '500', 500);  SELECT typeof(a), typeof(b), typeof(c), typeof(d) FROM t1;  text|integer|text|integer    -- Because column "a" has text affinity, numeric values on the  -- right-hand side of the comparisons are converted to text before  -- the comparison occurs.  SELECT a < 40,   a < 60,   a < 600 FROM t1;  0|1|1    -- Text affinity is applied to the right-hand operands but since  -- they are already TEXT this is a no-op; no conversions occur.  SELECT a < '40', a < '60', a < '600' FROM t1;  0|1|1    -- Column "b" has numeric affinity and so numeric affinity is applied  -- to the operands on the right.  Since the operands are already numeric,  -- the application of affinity is a no-op; no conversions occur.  All  -- values are compared numerically.  SELECT b < 40,   b < 60,   b < 600 FROM t1;  0|0|1    -- Numeric affinity is applied to operands on the right, converting them  -- from text to integers.  Then a numeric comparison occurs.  SELECT b < '40', b < '60', b < '600' FROM t1;  0|0|1    -- No affinity conversions occur.  Right-hand side values all have  -- storage class INTEGER which are always less than the TEXT values  -- on the left.  SELECT c < 40,   c < 60,   c < 600 FROM t1;  0|0|0    -- No affinity conversions occur.  Values are compared as TEXT.  SELECT c < '40', c < '60', c < '600' FROM t1;  0|1|1    -- No affinity conversions occur.  Right-hand side values all have  -- storage class INTEGER which compare numerically with the INTEGER  -- values on the left.  SELECT d < 40,   d < 60,   d < 600 FROM t1;  0|0|1    -- No affinity conversions occur.  INTEGER values on the left are  -- always less than TEXT values on the right.  SELECT d < '40', d < '60', d < '600' FROM t1;  1|1|1 |

　　同样是文档给的源码。。。

所有例子中的结果是相同的不变，如果把表达式替换--表达式的形式“a<40”被重写为”40>a”.

4.0 操作符

所有的数学操作符(+, -,\*, /, %, <<, >>, &, |)，在被执行前，都会将两个操作数都转换为数值存储类型（INTEGER和REAL）。即使这个转换是有损和不可逆的，转换仍然会执行。一个数学操作符上的NULL操作数将产生NULL结果。一个数学操作符上的操作数，如果以任何方式看都不像数字，并且又不为空的话，将被转换为0或0.0。

**5排序，分组和复合选择**

当查询结果由ORDER BY子句排序，存储类NULL是第一位的，其次是INTEGER和REAL值穿插在数字顺序中，其次是TEXT值在整理 顺序，最后是BLOB在memcmp()中顺序。在分类之前没有存储类的转换发生。

When groupingvalues with the GROUP BY clause values with different storage classes areconsidered distinct, except for INTEGER and REAL values which are consideredequal if they are numerically equal. No affinities are applied to any values asthe result of a GROUP by clause.

The compoundSELECT operators UNION, INTERSECT and EXCEPT perform implicit comparisonsbetween values. No affinity is applied to comparison operands for the implicitcomparisons associated with UNION, INTERSECT, or EXCEPT - the values arecompared as is.

**6.0排序序列**

当SQLite比较两个String，它使用一个排序序列或排序函数（对同一事物的两种字）来确定哪一个String更好或者两个String一样。SQLite3内置的排序类型： BINARY,NOCASE, and RTRIM.。

* BINARY -比较字符串数据使用memcmp()，忽视文本编码。
* NOCASE - The same as binary, except the 26 upper case characters of ASCII are folded to their lower case equivalents before the comparison is performed. Note that only ASCII characters are case folded. SQLite does not attempt to do full UTF case folding due to the size of the tables required.
* 。
* RTRIM -和BINARY一样，除了尾部的空格字符被忽略。

应用程序可以注册其他功能的使用 整理[sqlite3\_create\_collation()](http://translate.baiducontent.com/transpage?cb=translateCallback&ie=utf8&source=url&query=c3ref%2Fcreate_collation.html&from=en&to=zh&token=&monLang=zh" \t "_blank)接口

**6.1 AssigningCollating Sequences from SQL**

Every column of every table has an associated collating function.If no collating function is explicitly defined, then the collating functiondefaults to BINARY. The COLLATE clause of the [column definition](http://www.sqlite.org/lang_createtable.html#tablecoldef) is used to define alternativecollating functions for a column.

The rules for determining which collating function to use for abinary comparison operator (=, <, >, <=, >=, !=, IS, and IS NOT)are as follows and in the order shown:

1.  If either operand has an explicit collating function assignmentusing the postfix [COLLATEoperator](http://www.sqlite.org/lang_expr.html" \l "collateop" \t "_blank), then the explicit collating function is used forcomparison, with precedence to the collating function of the left operand.

2.  If either operand is a column, then the collating function of thatcolumn is used with precedence to the left operand. For the purposes of theprevious sentence, a column name preceded by one or more unary "+"operators is still considered a column name.

3.  Otherwise, the BINARY collating function is used for comparison.

An operand of a comparison is considered to have an explicitcollating function assignment (rule 1 above) if any subexpression of theoperand uses the postfix [COLLATEoperator](http://www.sqlite.org/lang_expr.html" \l "collateop" \t "_blank). Thus, if a [COLLATEoperator](http://www.sqlite.org/lang_expr.html" \l "collateop" \t "_blank) is usedanywhere in a comparision expression, the collating function defined by thatoperator is used for string comparison regardless of what table columns mightbe a part of that expression. If two or more[COLLATEoperator](http://www.sqlite.org/lang_expr.html#collateop) subexpressionsappear anywhere in a comparison, the left most explicit collating function isused regardless of how deeply the COLLATE operators are nested in theexpression and regardless of how the expression is parenthesized.

The expression "x BETWEEN y and z" is logicallyequivalent to two comparisons "x >= y AND x <= z" and workswith respect to collating functions as if it were two separate comparisons. Theexpression "x IN (SELECT y ...)" is handled in the same way as theexpression "x = y" for the purposes of determining the collatingsequence. The collating sequence used for expressions of the form "x IN(y, z, ...)" is the collating sequence of x.

Terms of the ORDER BY clause that is part of a [SELECT](http://www.sqlite.org/lang_select.html) statement may be assigned a collatingsequence using the [COLLATEoperator](http://www.sqlite.org/lang_expr.html" \l "collateop" \t "_blank), in which case the specified collating function is used forsorting. Otherwise, if the expression sorted by an ORDER BY clause is a column,then the collating sequence of the column is used to determine sort order. Ifthe expression is not a column and has no COLLATE clause, then the BINARYcollating sequence is used.

**6.2排序序列的例子**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | CREATE TABLE t1(      x INTEGER PRIMARY KEY,      a,                 /\* collating sequence BINARY \*/      b COLLATE BINARY,  /\* collating sequence BINARY \*/      c COLLATE RTRIM,   /\* collating sequence RTRIM  \*/      d COLLATE NOCASE   /\* collating sequence NOCASE \*/  );                     /\* x   a     b     c       d \*/  INSERT INTO t1 VALUES(1,'abc','abc', 'abc  ','abc');  INSERT INTO t1 VALUES(2,'abc','abc', 'abc',  'ABC');  INSERT INTO t1 VALUES(3,'abc','abc', 'abc ', 'Abc');  INSERT INTO t1 VALUES(4,'abc','abc ','ABC',  'abc');    /\* Text comparison a=b is performed using the BINARY collating sequence. \*/  SELECT x FROM t1 WHERE a = b ORDER BY x;  --result 1 2 3    /\* Text comparison a=b is performed using the RTRIM collating sequence. \*/  SELECT x FROM t1 WHERE a = b COLLATE RTRIM ORDER BY x;  --result 1 2 3 4    /\* Text comparison d=a is performed using the NOCASE collating sequence. \*/  SELECT x FROM t1 WHERE d = a ORDER BY x;  --result 1 2 3 4    /\* Text comparison a=d is performed using the BINARY collating sequence. \*/  SELECT x FROM t1 WHERE a = d ORDER BY x;  --result 1 4    /\* Text comparison 'abc'=c is performed using the RTRIM collating sequence. \*/  SELECT x FROM t1 WHERE 'abc' = c ORDER BY x;  --result 1 2 3    /\* Text comparison c='abc' is performed using the RTRIM collating sequence. \*/  SELECT x FROM t1 WHERE c = 'abc' ORDER BY x;  --result 1 2 3    /\* Grouping is performed using the NOCASE collating sequence (Values  \*\* 'abc', 'ABC', and 'Abc' are placed in the same group). \*/  SELECT count(\*) FROM t1 GROUP BY d ORDER BY 1;  --result 4    /\* Grouping is performed using the BINARY collating sequence.  'abc' and  \*\* 'ABC' and 'Abc' form different groups \*/  SELECT count(\*) FROM t1 GROUP BY (d || '') ORDER BY 1;  --result 1 1 2    /\* Sorting or column c is performed using the RTRIM collating sequence. \*/  SELECT x FROM t1 ORDER BY c, x;  --result 4 1 2 3    /\* Sorting of (c||'') is performed using the BINARY collating sequence. \*/  SELECT x FROM t1 ORDER BY (c||''), x;  --result 4 2 3 1    /\* Sorting of column c is performed using the NOCASE collating sequence. \*/  SELECT x FROM t1 ORDER BY c COLLATE NOCASE, x;  --result 2 4 3 1 |

总算能完啦，虽然不完善，但是够我用啦，以后遇到再慢慢补充。。。

转发请注明出处：<http://www.cnblogs.com/jycboy/p/android.html>

我的CSDN博客：http://blog.csdn.net/jycboy