# 一、在SQLite中日期用哪种类型

在 SQLite 中，存储包含毫秒的日期时间数据时，通常有以下几种选择：

1. **TEXT 类型**：使用 TEXT 类型存储日期时间字符串，这是最常见的做法。你可以使用 strftime 函数来格式化日期时间，使其包含毫秒。
2. **REAL 类型**：使用 REAL 类型存储 Julian 日数，这是一种浮点数表示法，可以精确到秒。不过，这种方式不太直观，通常用于需要高性能计算的场景。
3. **INTEGER 类型**：使用 INTEGER 类型存储 Unix 时间戳（自 1970-01-01 00:00:00 UTC 以来的秒数）。这种方式也不直接支持毫秒，但可以通过乘以 1000 来存储毫秒。

**推荐使用 TEXT 类型**

对于大多数应用场景，推荐使用 TEXT 类型来存储日期时间，因为它更直观且易于读取和处理。以下是你的 SQL 语句的示例：

sqlite

-- 添加 LastModifiedTime 列，类型为 TEXT

ALTER TABLE FileHashes ADD COLUMN LastModifiedTime TEXT;

-- 更新 LastModifiedTime 列，设置为当前日期时间，格式化为 YYYY-MM-DD HH:MM:SS.SSS

UPDATE FileHashes SET LastModifiedTime = strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime');

**SQLite 的数据类型**

SQLite 支持以下几种数据类型：

* **NULL**：表示空值。
* **INTEGER**：有符号整数，根据值的大小存储为 1、2、3、4、6 或 8 字节。
* **REAL**：浮点数，存储为 8 字节 IEEE 浮点数。
* **TEXT**：文本字符串，使用数据库的编码（UTF-8、UTF-16BE 或 UTF-16LE）存储。
* **BLOB**：二进制数据，按原样存储。

**总结**

* **TEXT** 类型最适合存储包含毫秒的日期时间。
* 使用 strftime 函数可以方便地格式化日期时间字符串。

-- SQLite

select \* from sqlite\_master;

select \* from FileHashes;

select \* from sqlite\_sequence;

select sqlite\_version();

select datetime('now');

select datetime('now','localtime');

select strftime('%Y-%m-%d %H:%M:%f','now','localtime');

alter table FileHashes add column LastModifiedTime text;

update FileHashes set LastModifiedTime = strftime('%Y-%m-%d %H:%M:%f','now','localtime');

# 二、SQLite中设置默认值和触发器

在 SQLite 中，可以通过触发器或默认值来实现插入数据时自动给 CreatedTime 字段赋值。以下是两种方法的示例：

### 方法一：使用默认值

在创建表时，可以为 CreatedTime 字段指定一个默认值，这样在插入数据时如果没有显式指定 CreatedTime，它将自动被设置为当前日期时间。

sqlite

-- 创建表时指定 CreatedTime 的默认值

CREATE TABLE FileHashes (

id INTEGER PRIMARY KEY,

FilePath TEXT NOT NULL,

HashValue TEXT NOT NULL,

CreatedTime TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')),

LastModifiedTime TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime'))

);

### 方法二：使用触发器

sqlite

-- 创建触发器，在插入记录时自动设置 CreatedTime

CREATE TRIGGER set\_created\_time

BEFORE INSERT ON FileHashes

FOR EACH ROW

BEGIN

SELECT

CASE

WHEN NEW.CreatedTime IS NULL THEN

NEW.CreatedTime = strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')

END;

END;

-- 创建触发器，在更新记录时自动设置 LastModifiedTime

CREATE TRIGGER update\_last\_modified\_time

AFTER UPDATE ON FileHashes

FOR EACH ROW

BEGIN

UPDATE FileHashes SET LastModifiedTime = strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime') WHERE id = OLD.id;

END;

# 三、完整脚本及示例

以下是一个完整的示例，包括创建表、添加 CreatedTime 和 LastModifiedTime 字段、以及创建相应的触发器来自动设置这些字段的值。

**创建表**

首先，创建一个 FileHashes 表，并添加 CreatedTime 和 LastModifiedTime 字段。

sqlite

CREATE TABLE FileHashes (

id INTEGER PRIMARY KEY AUTOINCREMENT,

FilePath TEXT NOT NULL,

HashValue TEXT NOT NULL,

CreatedTime TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')),

LastModifiedTime TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime'))

);

**创建触发器**

接下来，创建两个触发器：

1. **插入触发器**：在插入记录时自动设置 CreatedTime。
2. **更新触发器**：在更新记录时自动设置 LastModifiedTime。

sqlite

-- 创建触发器，在插入记录时自动设置 CreatedTime

CREATE TRIGGER set\_created\_time

BEFORE INSERT ON FileHashes

FOR EACH ROW

BEGIN

SELECT

CASE

WHEN NEW.CreatedTime IS NULL THEN

NEW.CreatedTime = strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')

END;

END;

-- 创建触发器，在更新记录时自动设置 LastModifiedTime

CREATE TRIGGER update\_last\_modified\_time

AFTER UPDATE ON FileHashes

FOR EACH ROW

BEGIN

UPDATE FileHashes

SET LastModifiedTime = strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')

WHERE id = OLD.id;

END;

**完整的 SQL 脚本**

将上述内容整合成一个完整的 SQL 脚本：

sqlite

-- 创建 FileHashes 表

CREATE TABLE FileHashes (

id INTEGER PRIMARY KEY AUTOINCREMENT,

FilePath TEXT NOT NULL,

HashValue TEXT NOT NULL,

CreatedTime TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')),

LastModifiedTime TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime'))

);

-- 创建触发器，在插入记录时自动设置 CreatedTime

CREATE TRIGGER set\_created\_time

BEFORE INSERT ON FileHashes

FOR EACH ROW

BEGIN

SELECT

CASE

WHEN NEW.CreatedTime IS NULL THEN

NEW.CreatedTime = strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')

END;

END;

-- 创建触发器，在更新记录时自动设置 LastModifiedTime

CREATE TRIGGER update\_last\_modified\_time

AFTER UPDATE ON FileHashes

FOR EACH ROW

BEGIN

UPDATE FileHashes

SET LastModifiedTime = strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')

WHERE id = OLD.id;

END;

**插入和更新示例**

以下是一些插入和更新记录的示例：

sqlite

-- 插入一条记录

INSERT INTO FileHashes (FilePath, HashValue) VALUES ('/path/to/file1.txt', 'abc123');

-- 更新一条记录

UPDATE FileHashes SET HashValue = 'def456' WHERE id = 1;

**解释**

1. **创建表**：定义了 FileHashes 表，包含 id、FilePath、HashValue、CreatedTime 和 LastModifiedTime 字段。CreatedTime 和 LastModifiedTime 字段的默认值为当前日期时间。
2. **插入触发器**：在插入记录时，如果 CreatedTime 为空，则自动设置为当前日期时间。
3. **更新触发器**：在更新记录时，自动将 LastModifiedTime 设置为当前日期时间。

这样，每次插入新记录时，CreatedTime 将自动被设置为当前日期时间；每次更新记录时，LastModifiedTime 也将自动被更新为当前日期时间。

# 四、在C#中

1. **SQL 脚本**：
   * **表创建**：定义了 FileHashes 表，包含 Id、FileName、FilePath、HashValue、CreatedTime 和 LastModifiedTime 字段。CreatedTime 和 LastModifiedTime 字段的默认值为当前日期时间。
   * **插入触发器**：在插入记录时，如果 CreatedTime 为空，则自动设置为当前日期时间。
   * **更新触发器**：在更新记录时，自动将 LastModifiedTime 设置为当前日期时间。
2. **C# 代码**：
   * **连接字符串**：使用字符串插值来构建连接字符串，提高可读性和安全性。
   * **SQL 脚本**：使用多行字符串（@ 符号）来提高可读性。

**执行 SQL 脚本**

在 C# 中执行这个 SQL 脚本的示例：

csharp

using System;

using System.Data.SQLite;

public class DatabaseInitializer

{

public const string \_sqliteDataName = "FileHashes.db";

public static readonly string \_sqliteConnectionString = $"Data Source={\_sqliteDataName};Version=3;";

public const string \_createTableScript = @"

CREATE TABLE FileHashes (

Id INTEGER PRIMARY KEY AUTOINCREMENT,

FileName TEXT NOT NULL,

FilePath TEXT NOT NULL,

HashValue TEXT NOT NULL,

CreatedTime TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')),

LastModifiedTime TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')),

CONSTRAINT uc\_HashValue UNIQUE (HashValue)

);

CREATE TRIGGER set\_created\_time

BEFORE INSERT ON FileHashes

FOR EACH ROW

BEGIN

WHEN NEW.CreatedTime IS NULL

SET NEW.CreatedTime = strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime');

END;

CREATE TRIGGER update\_last\_modified\_time

AFTER UPDATE ON FileHashes

FOR EACH ROW

BEGIN

UPDATE FileHashes

SET LastModifiedTime = strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')

WHERE id = OLD.id;

END;

";

public static void InitializeDatabase()

{

using (var connection = new SQLiteConnection(\_sqliteConnectionString))

{

connection.Open();

using (var command = new SQLiteCommand(\_createTableScript, connection))

{

command.ExecuteNonQuery();

}

}

}

public static void Main(string[] args)

{

InitializeDatabase();

Console.WriteLine("Database initialized successfully.");

}

}

**运行代码**

1. 确保你已经安装了 System.Data.SQLite 包。你可以通过 NuGet 包管理器安装：

sh

Install-Package System.Data.SQLite

1. 运行 Main 方法，初始化数据库并创建表和触发器。

这样，你的代码和 SQL 脚本应该是正确且高效的。

# command.ExecuteNonQuery();

在 SQLite 中，ExecuteNonQuery 方法用于执行不返回结果集的 SQL 命令，例如 INSERT、UPDATE、DELETE 以及 CREATE TABLE、CREATE TRIGGER 等 DDL（数据定义语言）命令。ExecuteNonQuery 方法返回一个整数值，表示受影响的行数。具体来说，返回值有以下几种情况：

1. **对于 INSERT 语句**：
   * 返回值通常是 1，表示成功插入了一行。
   * 如果插入多行（例如使用 INSERT INTO ... SELECT ...），返回值将是实际插入的行数。
2. **对于 UPDATE 语句**：
   * 返回值是被更新的行数。
   * 如果没有行被更新，返回值为 0。
3. **对于 DELETE 语句**：
   * 返回值是被删除的行数。
   * 如果没有行被删除，返回值为 0。
4. **对于 DDL 语句（如 CREATE TABLE、CREATE TRIGGER、DROP TABLE 等）**：
   * 返回值通常是 0，因为这些语句不直接影响现有的行数。
   * 有些 SQLite 实现可能会返回 -1 或其他特定值，但这不是标准行为。

**总结**

* **INSERT 语句**：返回插入的行数。
* **UPDATE 语句**：返回更新的行数。
* **DELETE 语句**：返回删除的行数。
* **DDL 语句**：通常返回 0，因为这些语句不直接影响现有的行数。

# id INTEGER PRIMARY KEY 和 Id INTEGER PRIMARY KEY AUTOINCREMENT 有什么区别？

在 SQLite 中，INTEGER PRIMARY KEY 和 INTEGER PRIMARY KEY AUTOINCREMENT 都可以用于定义一个自增的主键，但它们之间有一些重要的区别：

### 1. INTEGER PRIMARY KEY

* **自动递增**：当插入一条没有指定主键值的记录时，SQLite 会自动为该列生成一个唯一的整数值。
* **行为**：生成的值是从 1 开始递增的，但如果删除了一些记录，新的记录可能会使用之前删除的记录的 ID。
* **性能**：这种情况下，SQLite 的性能通常更好，因为它不需要维护一个单独的序列来生成主键值。

### 2. INTEGER PRIMARY KEY AUTOINCREMENT

* **自动递增**：与 INTEGER PRIMARY KEY 类似，当插入一条没有指定主键值的记录时，SQLite 会自动为该列生成一个唯一的整数值。
* **行为**：生成的值是从 1 开始递增的，并且即使删除了一些记录，新的记录也不会使用之前删除的记录的 ID。这意味着生成的值是连续的，不会重复。
* **性能**：由于需要维护一个单独的序列来生成主键值，性能可能会稍微差一些。

**具体区别总结：**

* **连续性**：
  + INTEGER PRIMARY KEY：生成的值可能不连续，因为删除记录后，新的记录可能会使用之前删除的记录的 ID。
  + INTEGER PRIMARY KEY AUTOINCREMENT：生成的值总是连续的，不会重复使用之前删除的记录的 ID。
* **性能**：
  + INTEGER PRIMARY KEY：性能更好，因为不需要额外的序列管理。
  + INTEGER PRIMARY KEY AUTOINCREMENT：性能稍差，因为需要维护一个单独的序列。

**示例**

**使用 INTEGER PRIMARY KEY**

sqlite

CREATE TABLE Example1 (

id INTEGER PRIMARY KEY,

name TEXT

);

-- 插入记录

INSERT INTO Example1 (name) VALUES ('Alice');

INSERT INTO Example1 (name) VALUES ('Bob');

-- 删除记录

DELETE FROM Example1 WHERE id = 1;

-- 再次插入记录

INSERT INTO Example1 (name) VALUES ('Charlie');

-- 查询结果

SELECT \* FROM Example1;

-- 可能的结果：

-- 2 | Bob

-- 1 | Charlie

**使用 INTEGER PRIMARY KEY AUTOINCREMENT**

sqlite

CREATE TABLE Example2 (

id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT

);

-- 插入记录

INSERT INTO Example2 (name) VALUES ('Alice');

INSERT INTO Example2 (name) VALUES ('Bob');

-- 删除记录

DELETE FROM Example2 WHERE id = 1;

-- 再次插入记录

INSERT INTO Example2 (name) VALUES ('Charlie');

-- 查询结果

SELECT \* FROM Example2;

-- 可能的结果：

-- 2 | Bob

-- 3 | Charlie

### 结论

* 如果你需要主键值严格连续且不重复使用已删除的 ID，使用 INTEGER PRIMARY KEY AUTOINCREMENT。
* 如果你对主键值的连续性要求不高，使用 INTEGER PRIMARY KEY 以获得更好的性能。

大多数情况下，INTEGER PRIMARY KEY 已经足够满足需求，除非你有特定的业务需求需要严格的连续性。

# 2024-11-06 SQLite脚本记录

## 2024-11-06 14:33:18

-- SQLite

SELECT \* FROM SQLite\_master;

select COUNT(1) from FileHashes;

select COUNT(1) from FileHashes\_old;

SELECT STRFTIME('%Y-%m-%d',datetime(CreatedTime)) AS CreatedDay,COUNT(1) FROM FileHashes WHERE 1=1 AND CreatedDay = date('now') GROUP BY CreatedDay;

ALTER TABLE FileHashes ALTER COLUMN IsPortrait DROP DEFAULT;

SELECT (CASE WHEN FilePath LIKE '%Computer%' THEN 0 WHEN FilePath LIKE '%Mobile%' THEN 1 ELSE NULL END),\* FROM FileHashes;

ALTER TABLE FileHashes ALTER COLUMN IsPortrait SET DEFAULT NULL;

ALTER TABLE FileHashes RENAME TO FileHashes\_old;

select \* from FileHashes\_old;

INSERT INTO FileHashes (FileName,FilePath,HashValue,IsPortrait,CreatedTime,LastModifiedTime)

SELECT FileName,FilePath,HashValue,

(CASE WHEN FilePath LIKE '%Computer%' THEN 0 WHEN FilePath LIKE '%Mobile%' THEN 1 ELSE NULL END) AS IsPortrait,

CreatedTime,LastModifiedTime

FROM FileHashes\_old;

CREATE TABLE IF NOT EXISTS FileHashes (

    ID INTEGER PRIMARY KEY,

    FileName TEXT NOT NULL,

    FilePath TEXT NOT NULL,

    HashValue TEXT NOT NULL,

    IsPortrait INTEGER DEFAULT NULL,

    CreatedTime TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')),

    LastModifiedTime TEXT DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')),

    CONSTRAINT uc\_HashValue UNIQUE (HashValue)

);

CREATE TRIGGER IF NOT EXISTS set\_created\_time

BEFORE INSERT ON FileHashes

FOR EACH ROW

WHEN NEW.CreatedTime IS NULL

BEGIN

    UPDATE FileHashes

    SET CreatedTime = strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')

    WHERE id = NEW.id;

END;

CREATE TRIGGER IF NOT EXISTS update\_last\_modified\_time

AFTER UPDATE ON FileHashes

FOR EACH ROW

BEGIN

    UPDATE FileHashes

    SET LastModifiedTime = strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime')

    WHERE id = OLD.id;

END;

select \* from FileHashes;

SELECT STRFTIME('%Y-%m-%d',datetime(CreatedTime)) AS CreatedDay,COUNT(1) FROM FileHashes WHERE 1=1 AND CreatedDay = '2024-111-00' GROUP BY CreatedDay;

SELECT \* FROM FileHashes WHERE HashValue = '631B49C74E4540A63C83A72C510F7B9830409A77B179B07A4A9DA8F2C9A085B2';

SELECT COUNT(\*) FROM FileHashes WHERE 1=1 AND HashValue = '631B49C74E4540A63C83A72C510F7B9830409A77B179B07A4A9DA8F2C9A085B2';

SELECT STRFTIME('%Y-%m-%d',datetime(CreatedTime)) AS CreatedDay

      ,COUNT(1) A

FROM FileHashes

WHERE 1=1

AND CreatedDay = date('now')

GROUP BY CreatedDay;

SELECT COUNT(1) Count FROM FileHashes WHERE 1=1 AND STRFTIME('%Y-%m-%d',datetime(CreatedTime)) = date('now') GROUP BY STRFTIME('%Y-%m-%d',datetime(CreatedTime));