# 一、用二进制文件存储遇到的问题

你能获取到这篇文章吗？<https://blog.csdn.net/qq_40374647/article/details/113694439>

我无法直接访问外部网站或文章，包括您提供的链接。不过，您可以告诉我这篇文章的主要内容或您感兴趣的具体部分，我会尽力帮助您理解和解答相关问题。如果您有特定的问题或需要讨论文章中的某个技术点，请详细说明，我将根据您的需求提供帮助。

**问题分析**

根据您的描述，主要问题是二进制序列化在处理大数据量对象时性能较差，特别是在内存管理和垃圾回收方面。具体表现如下：

* **小数据量**：100万次序列化耗时4.74秒，内存几乎没有GC。
* **大数据量**：将Data长度修改为100K后，100万次序列化耗时39.29秒，且GC频繁。

**问题原因**

1. **内存占用**：
   * 序列化过程中，MemoryStream需要将数据写入流中，然后再通过ToArray方法复制一份，导致内存占用翻倍。
   * 对于100K的数据，每次序列化需要占用约200K的内存（102462 \* 2 = 204924 Byte）。
   * 100万次序列化总共需要申请和释放约190.85GB的内存。
2. **垃圾回收**：
   * 频繁的内存分配和释放导致GC频繁触发，影响性能。

**优化方案**

1. **复用MemoryStream**：
   * 将MemoryStream声明在类中，每次序列化完不要释放，而是重置流的位置。
   * 这样可以减少内存分配和释放的次数，降低GC的频率。

**示例代码**

csharp

using System;

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

public class Student

{

public string Name { get; set; }

public int Age { get; set; }

public string Country { get; set; }

public byte[] Data { get; set; }

}

public class Program

{

private static MemoryStream \_memoryStream = new MemoryStream();

public static void Main(string[] args)

{

*// 初始化学生对象*

var student = new Student

{

Name = "张三",

Age = 20,

Country = "中国",

Data = new byte[100 \* 1024] *// 100K*

};

*// 测量序列化时间*

var timeMeasurer = new TimeMeasurer();

timeMeasurer.Start();

for (int i = 0; i < 1000000; i++)

{

Serialize(student);

}

timeMeasurer.Stop();

Console.WriteLine($"总耗时: {timeMeasurer.ElapsedMilliseconds} 毫秒");

}

public static byte[] Serialize(Student student)

{

\_memoryStream.Position = 0; *// 重置流的位置*

\_memoryStream.SetLength(0); *// 清空流的内容*

var formatter = new BinaryFormatter();

formatter.Serialize(\_memoryStream, student);

return \_memoryStream.ToArray();

}

}

**优化效果**

* **时间**：优化后，用时比之前减少了两倍以上。
* **GC**：虽然GC仍然频繁，但相比之前有所减少。

**进一步优化建议**

1. **使用更高效的序列化库**：
   * 考虑使用如protobuf-net、MessagePack等高效的二进制序列化库，这些库在性能和内存管理上通常优于BinaryFormatter。
2. **异步操作**：
   * 如果可能，使用异步序列化和反序列化方法，减少主线程的阻塞。
3. **分批处理**：
   * 对于大数据量，可以考虑分批处理，避免一次性处理过多数据导致内存压力过大。

通过这些优化措施，可以进一步提升序列化性能，减少内存占用和GC频率。

这个问题，是处理二进制文件必然会遇到的吗？还是说只是特例？

**问题的普遍性**

这个问题在处理二进制文件时并不罕见，尤其是在涉及大量数据的序列化和反序列化时。以下是一些常见的场景和原因：

**常见场景**

1. **大数据量的序列化**：
   * 当序列化对象包含大量数据（如大数组、大字符串等）时，内存占用会显著增加。
   * 每次序列化都会创建新的内存对象，导致频繁的内存分配和释放，从而触发垃圾回收（GC）。
2. **频繁的序列化操作**：
   * 在高并发或高性能要求的应用中，频繁的序列化操作会导致大量的内存分配和释放，影响性能。
3. **不合理的内存管理**：
   * 如果没有合理管理内存，例如每次都创建新的MemoryStream对象而没有复用，会导致不必要的内存开销。

**原因分析**

1. **内存分配和释放**：
   * 每次序列化时，MemoryStream需要分配新的内存来存储数据。
   * 使用ToArray方法时，会再次分配新的内存来复制数据，导致内存占用翻倍。
2. **垃圾回收**：
   * 频繁的内存分配和释放会触发垃圾回收，特别是当对象生命周期较短时，GC会频繁运行，影响性能。

**解决方案**

1. **复用MemoryStream**：
   * 如前所述，将MemoryStream声明在类中，每次序列化完重置流的位置，而不是每次都创建新的对象。
   * 这样可以减少内存分配和释放的次数，降低GC的频率。
2. **使用高效的序列化库**：
   * 考虑使用如protobuf-net、MessagePack等高效的二进制序列化库，这些库在性能和内存管理上通常优于BinaryFormatter。
3. **异步操作**：
   * 使用异步序列化和反序列化方法，减少主线程的阻塞，提高应用的响应性和性能。
4. **分批处理**：
   * 对于大数据量，可以考虑分批处理，避免一次性处理过多数据导致内存压力过大。

**特例与普遍性**

* **特例**：在某些特定场景下，如果数据量较小且序列化操作不频繁，问题可能不会那么明显。
* **普遍性**：在处理大数据量或高频序列化操作时，这些问题几乎是不可避免的，因此需要采取相应的优化措施。

**总结**

处理二进制文件时，尤其是涉及大量数据的序列化和反序列化，确实会遇到内存管理和性能问题。这些问题并不是特例，而是普遍存在的。通过合理的内存管理和使用高效的序列化库，可以有效缓解这些问题，提升应用的性能和稳定性。

我现在需要对几千个文件建立hash值记录，以便有新文件时判断该文件是否已存在。那需要记录文件路径、文件名、Hash值。那处理这个判断是用二进制文件好，还是用SQLite好？

# 二、用二进制文件存储数据和用SQLite存储数据的优缺点比较

### 选择方案

对于您的需求，即对几千个文件建立哈希值记录并进行快速查找，可以选择两种方案：使用二进制文件或使用SQLite数据库。下面分别分析这两种方案的优缺点，帮助您做出决策。

### 1. 使用二进制文件

#### 优点

* **简单**：实现相对简单，不需要额外的数据库依赖。
* **轻量**：文件系统本身就是一个轻量级的存储方式，适合小型项目。

#### 缺点

* **性能**：随着文件数量的增加，查找效率会显著下降。每次查找都需要读取整个文件并进行线性搜索。
* **维护性**：二进制文件的维护和调试较为困难，尤其是在文件格式发生变化时。
* **扩展性**：难以扩展，不适合大规模数据的管理和查询。

### 2. 使用SQLite数据库

#### 优点

* **性能**：SQLite是一个嵌入式数据库，支持高效的索引和查询，能够快速查找和插入数据。
* **易维护**：SQL语言提供了丰富的查询和管理功能，易于维护和调试。
* **扩展性**：SQLite支持事务处理和并发操作，适合处理中等规模的数据集。
* **可靠性**：SQLite具有良好的数据完整性和一致性保证。

#### 缺点

* **复杂性**：相对于二进制文件，使用SQLite需要一定的数据库知识和配置。
* **资源消耗**：虽然SQLite是轻量级的，但在某些资源受限的环境中可能会有一定的开销。

### 推荐方案

基于您的需求，**推荐使用SQLite数据库**。以下是几个主要原因：

1. **高效查询**：SQLite支持索引，可以快速查找文件的哈希值，即使文件数量增加也不会显著影响性能。
2. **易维护**：SQL语言提供了丰富的查询和管理功能，便于维护和调试。
3. **扩展性**：SQLite支持事务处理和并发操作，适合处理中等规模的数据集。
4. **可靠性**：SQLite具有良好的数据完整性和一致性保证，确保数据的可靠存储。

### 示例代码

以下是一个使用SQLite数据库记录文件路径、文件名和哈希值的示例代码：

#### 安装SQLite

首先，确保安装了SQLite的NuGet包：

sh

dotnet add package System.Data.SQLite

#### 创建数据库和表

csharp

using System;

using System.Data.SQLite;

class Program

{

static void Main(string[] args)

{

*// 创建数据库连接*

string connectionString = "Data Source=file\_hashes.db;Version=3;";

using (var connection = new SQLiteConnection(connectionString))

{

connection.Open();

*// 创建表*

string createTableQuery = @"

CREATE TABLE IF NOT EXISTS FileHashes (

Id INTEGER PRIMARY KEY AUTOINCREMENT,

FilePath TEXT NOT NULL,

FileName TEXT NOT NULL,

HashValue TEXT NOT NULL

);";

using (var command = new SQLiteCommand(createTableQuery, connection))

{

command.ExecuteNonQuery();

}

}

}

}

#### 插入数据

csharp

using System;

using System.Data.SQLite;

using System.Security.Cryptography;

using System.Text;

using System.IO;

class Program

{

static void Main(string[] args)

{

*// 创建数据库连接*

string connectionString = "Data Source=file\_hashes.db;Version=3;";

using (var connection = new SQLiteConnection(connectionString))

{

connection.Open();

*// 插入数据*

string filePath = @"C:\path\to\file.txt";

string fileName = Path.GetFileName(filePath);

string hashValue = ComputeFileHash(filePath);

string insertQuery = @"

INSERT INTO FileHashes (FilePath, FileName, HashValue)

VALUES (@FilePath, @FileName, @HashValue);";

using (var command = new SQLiteCommand(insertQuery, connection))

{

command.Parameters.AddWithValue("@FilePath", filePath);

command.Parameters.AddWithValue("@FileName", fileName);

command.Parameters.AddWithValue("@HashValue", hashValue);

command.ExecuteNonQuery();

}

}

}

static string ComputeFileHash(string filePath)

{

using (var sha256 = SHA256.Create())

{

using (var stream = File.OpenRead(filePath))

{

byte[] hashBytes = sha256.ComputeHash(stream);

return BitConverter.ToString(hashBytes).Replace("-", "").ToLowerInvariant();

}

}

}

}

#### 查询数据

csharp

using System;

using System.Data.SQLite;

class Program

{

static void Main(string[] args)

{

*// 创建数据库连接*

string connectionString = "Data Source=file\_hashes.db;Version=3;";

using (var connection = new SQLiteConnection(connectionString))

{

connection.Open();

*// 查询数据*

string filePath = @"C:\path\to\file.txt";

string query = @"

SELECT FileName, HashValue

FROM FileHashes

WHERE FilePath = @FilePath;";

using (var command = new SQLiteCommand(query, connection))

{

command.Parameters.AddWithValue("@FilePath", filePath);

using (var reader = command.ExecuteReader())

{

while (reader.Read())

{

string fileName = reader.GetString(0);

string hashValue = reader.GetString(1);

Console.WriteLine($"File: {fileName}, Hash: {hashValue}");

}

}

}

}

}

}

### 结论

使用SQLite数据库可以更好地满足您的需求，提供高效、可靠的文件哈希值记录和查询功能。希望上述示例代码能帮助您快速实现这一功能。

建表前是否需要判断表是否已存在？ 表插入新数据前是否需要判断数据是否已存在？

### 建表前判断表是否存在

在创建表之前，判断表是否已存在是一个好的实践，可以避免重复创建表导致的错误。SQLite 提供了 CREATE TABLE IF NOT EXISTS 语句，可以在表不存在时创建表，如果表已存在则不会执行任何操作。

#### 示例代码

csharp

using System;

using System.Data.SQLite;

class Program

{

static void Main(string[] args)

{

*// 创建数据库连接*

string connectionString = "Data Source=file\_hashes.db;Version=3;";

using (var connection = new SQLiteConnection(connectionString))

{

connection.Open();

*// 创建表*

string createTableQuery = @"

CREATE TABLE IF NOT EXISTS FileHashes (

Id INTEGER PRIMARY KEY AUTOINCREMENT,

FilePath TEXT NOT NULL,

FileName TEXT NOT NULL,

HashValue TEXT NOT NULL,

UNIQUE (FilePath) -- 确保 FilePath 唯一

);";

using (var command = new SQLiteCommand(createTableQuery, connection))

{

command.ExecuteNonQuery();

}

}

}

}

### 插入新数据前判断数据是否已存在

在插入新数据前判断数据是否已存在也是一个好的实践，可以避免重复插入相同的数据。有几种方法可以实现这一点：

1. **使用 INSERT OR IGNORE 语句**：如果表中已经存在相同的 FilePath，则忽略插入操作。
2. **使用 SELECT 语句检查数据是否存在，然后再插入**。

#### 方法1：使用 INSERT OR IGNORE

在创建表时，可以添加一个唯一约束（如 UNIQUE (FilePath)），然后使用 INSERT OR IGNORE 语句插入数据。如果 FilePath 已经存在，插入操作将被忽略。

csharp

using System;

using System.Data.SQLite;

using System.Security.Cryptography;

using System.Text;

using System.IO;

class Program

{

static void Main(string[] args)

{

*// 创建数据库连接*

string connectionString = "Data Source=file\_hashes.db;Version=3;";

using (var connection = new SQLiteConnection(connectionString))

{

connection.Open();

*// 创建表*

string createTableQuery = @"

CREATE TABLE IF NOT EXISTS FileHashes (

Id INTEGER PRIMARY KEY AUTOINCREMENT,

FilePath TEXT NOT NULL,

FileName TEXT NOT NULL,

HashValue TEXT NOT NULL,

UNIQUE (FilePath) -- 确保 FilePath 唯一

);";

using (var command = new SQLiteCommand(createTableQuery, connection))

{

command.ExecuteNonQuery();

}

*// 插入数据*

string filePath = @"C:\path\to\file.txt";

string fileName = Path.GetFileName(filePath);

string hashValue = ComputeFileHash(filePath);

string insertQuery = @"

INSERT OR IGNORE INTO FileHashes (FilePath, FileName, HashValue)

VALUES (@FilePath, @FileName, @HashValue);";

using (var command = new SQLiteCommand(insertQuery, connection))

{

command.Parameters.AddWithValue("@FilePath", filePath);

command.Parameters.AddWithValue("@FileName", fileName);

command.Parameters.AddWithValue("@HashValue", hashValue);

command.ExecuteNonQuery();

}

}

}

static string ComputeFileHash(string filePath)

{

using (var sha256 = SHA256.Create())

{

using (var stream = File.OpenRead(filePath))

{

byte[] hashBytes = sha256.ComputeHash(stream);

return BitConverter.ToString(hashBytes).Replace("-", "").ToLowerInvariant();

}

}

}

}

#### 方法2：使用 SELECT 语句检查数据是否存在

在插入数据前，先使用 SELECT 语句检查数据是否存在，如果不存在再进行插入操作。

csharp

using System;

using System.Data.SQLite;

using System.Security.Cryptography;

using System.Text;

using System.IO;

class Program

{

static void Main(string[] args)

{

*// 创建数据库连接*

string connectionString = "Data Source=file\_hashes.db;Version=3;";

using (var connection = new SQLiteConnection(connectionString))

{

connection.Open();

*// 创建表*

string createTableQuery = @"

CREATE TABLE IF NOT EXISTS FileHashes (

Id INTEGER PRIMARY KEY AUTOINCREMENT,

FilePath TEXT NOT NULL,

FileName TEXT NOT NULL,

HashValue TEXT NOT NULL,

UNIQUE (FilePath) -- 确保 FilePath 唯一

);";

using (var command = new SQLiteCommand(createTableQuery, connection))

{

command.ExecuteNonQuery();

}

*// 插入数据*

string filePath = @"C:\path\to\file.txt";

string fileName = Path.GetFileName(filePath);

string hashValue = ComputeFileHash(filePath);

*// 检查数据是否存在*

string checkQuery = "SELECT COUNT(\*) FROM FileHashes WHERE FilePath = @FilePath;";

using (var checkCommand = new SQLiteCommand(checkQuery, connection))

{

checkCommand.Parameters.AddWithValue("@FilePath", filePath);

int count = (int)checkCommand.ExecuteScalar();

if (count == 0)

{

*// 插入数据*

string insertQuery = @"

INSERT INTO FileHashes (FilePath, FileName, HashValue)

VALUES (@FilePath, @FileName, @HashValue);";

using (var insertCommand = new SQLiteCommand(insertQuery, connection))

{

insertCommand.Parameters.AddWithValue("@FilePath", filePath);

insertCommand.Parameters.AddWithValue("@FileName", fileName);

insertCommand.Parameters.AddWithValue("@HashValue", hashValue);

insertCommand.ExecuteNonQuery();

}

}

else

{

Console.WriteLine("文件已存在，跳过插入操作。");

}

}

}

}

static string ComputeFileHash(string filePath)

{

using (var sha256 = SHA256.Create())

{

using (var stream = File.OpenRead(filePath))

{

byte[] hashBytes = sha256.ComputeHash(stream);

return BitConverter.ToString(hashBytes).Replace("-", "").ToLowerInvariant();

}

}

}

}

### 总结

* **建表前判断表是否存在**：使用 CREATE TABLE IF NOT EXISTS 语句。
* **插入新数据前判断数据是否已存在**：可以使用 INSERT OR IGNORE 语句或 SELECT 语句检查数据是否存在，然后再插入。

这两种方法都可以有效地避免重复创建表和插入重复数据，提高数据的一致性和可靠性。