# 问：

帮我设计一下C#封装方法的思路。

现在对SQLite有如下处理：

1、建表：CREATE TABLE IF NOT EXISTS

2、查询数据是否存在：SELECT COUNT(\*) FROM

3、插入数据：INSERT OR IGNORE INTO

4、查询数据明细：SELECT Column1,Column2,Column3 FROM

现在想要封装成通用方法，不知道是把这4种分别分开封装成不同的方法好，

还是组合一下，将某几项封装在一起是一个方法好？

因为在使用时，会连接数据库，然后判断表在不在，不在就新建，在就查数据。

分开的话，每一个方法都要独立的连接数据库一次，不知道是否占用资源 合一起就可以一次连接，使用。

你帮我评估一下，哪种更合适。

# 答：

在设计C#封装方法时，考虑到代码的可维护性、复用性和性能，可以采取以下几种策略来平衡这些需求：

## 1. 单独封装每个操作

**优点**:

* **高内聚低耦合**: 每个方法只负责一个功能，便于理解和维护。
* **复用性强**: 可以在不同场景下单独调用这些方法，灵活性高。

**缺点**:

* **多次连接数据库**: 每次调用方法都会连接数据库，可能会增加资源消耗。

## 2. 组合多个操作

**优点**:

* **减少数据库连接次数**: 通过一次连接完成多个操作，提高性能。
* **简化调用**: 一次调用即可完成多个步骤，使用方便。

**缺点**:

* **耦合度高**: 方法的功能较多，不易于维护和扩展。
* **复用性差**: 如果只需要其中某个操作，仍需调用整个方法。

## 建议方案

结合上述优缺点，建议采用一种混合策略：

1. **基础操作方法**: 封装每个基本操作（建表、查询数据是否存在、插入数据、查询数据明细）为单独的方法。
2. **组合方法**: 在此基础上，再封装一个组合方法，用于常见的业务流程（如连接数据库、判断表是否存在、建表、查询数据等）。

## 示例代码

csharp

using System;

using System.Data.SQLite;

public class SQLiteHelper

{

private string \_connectionString;

public SQLiteHelper(string connectionString)

{

\_connectionString = connectionString;

}

*// 连接数据库*

private SQLiteConnection GetConnection()

{

return new SQLiteConnection(\_connectionString);

}

*// 建表*

public void CreateTableIfNotExists(string tableName, string columns)

{

using (var connection = GetConnection())

{

connection.Open();

string query = $"CREATE TABLE IF NOT EXISTS {tableName} ({columns});";

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

{

command.ExecuteNonQuery();

}

}

}

*// 查询数据是否存在*

public bool DataExists(string tableName)

{

using (var connection = GetConnection())

{

connection.Open();

string query = $"SELECT COUNT(\*) FROM {tableName};";

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

{

int count = Convert.ToInt32(command.ExecuteScalar());

return count > 0;

}

}

}

*// 插入数据*

public void InsertData(string tableName, string values)

{

using (var connection = GetConnection())

{

connection.Open();

string query = $"INSERT OR IGNORE INTO {tableName} VALUES ({values});";

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

{

command.ExecuteNonQuery();

}

}

}

*// 查询数据明细*

public SQLiteDataReader QueryData(string tableName, string columns)

{

using (var connection = GetConnection())

{

connection.Open();

string query = $"SELECT {columns} FROM {tableName};";

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

{

return command.ExecuteReader();

}

}

}

*// 组合方法：连接数据库、判断表是否存在、建表、查询数据*

public SQLiteDataReader ExecuteCommonOperations(string tableName, string columns, string values)

{

using (var connection = GetConnection())

{

connection.Open();

*// 判断表是否存在，不存在则创建*

if (!DataExists(tableName))

{

CreateTableIfNotExists(tableName, columns);

}

*// 插入数据*

InsertData(tableName, values);

*// 查询数据明细*

string query = $"SELECT {columns} FROM {tableName};";

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

{

return command.ExecuteReader();

}

}

}

}

## 使用示例

csharp

class Program

{

static void Main(string[] args)

{

string connectionString = "Data Source=example.db;Version=3;";

SQLiteHelper helper = new SQLiteHelper(connectionString);

string tableName = "Users";

string columns = "Id INTEGER PRIMARY KEY, Name TEXT, Age INTEGER";

string values = "1, 'John Doe', 30";

using (var reader = helper.ExecuteCommonOperations(tableName, columns, values))

{

while (reader.Read())

{

Console.WriteLine($"Id: {reader["Id"]}, Name: {reader["Name"]}, Age: {reader["Age"]}");

}

}

}

}

## 总结

通过这种方式，你既可以保持每个基础操作的独立性和复用性，又可以通过组合方法来优化常见业务流程的性能。这样既提高了代码的可维护性，又兼顾了性能需求。

# SQLite参数化查询示例：

为了实现动态添加查询条件并传递参数，需要在 SQLiteCommand 中使用参数化查询，以避免 SQL 注入攻击。以下是完善后的

## 代码：

csharp

using System;

using System.Data.SQLite;

public class SQLiteHelper

{

private string \_connectionString;

public SQLiteHelper(string connectionString)

{

\_connectionString = connectionString;

}

// 连接数据库

private SQLiteConnection GetConnection()

{

return new SQLiteConnection(\_connectionString);

}

/// <summary>

/// 查询数据明细

/// </summary>

/// <param name="tableName">表名</param>

/// <param name="columns">列名，如： FilePath,FileName,HashValue </param>

/// <param name="parameters">查询条件数组，每个元素是一个键值对字符串，如： "HashValue=@HashValue" </param>

/// <param name="values">查询条件对应的值数组，与 parameters 对应，如： new object[] { "abc123" } </param>

/// <returns>SQLiteDataReader</returns>

public SQLiteDataReader QueryData(string tableName, string columns, params (string parameter, object value)[] conditions)

{

using (var connection = GetConnection())

{

connection.Open();

string query = $"SELECT {columns} FROM {tableName} WHERE 1=1";

if (conditions != null && conditions.Length > 0)

{

for (int i = 0; i < conditions.Length; i++)

{

query += $" AND {conditions[i].parameter}";

}

}

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

{

if (conditions != null && conditions.Length > 0)

{

for (int i = 0; i < conditions.Length; i++)

{

command.Parameters.AddWithValue($"@p{i}", conditions[i].value);

}

}

return command.ExecuteReader();

}

}

}

}

## 使用示例

### 无条件查询

csharp

class Program

{

static void Main(string[] args)

{

string connectionString = "Data Source=example.db;Version=3;";

SQLiteHelper helper = new SQLiteHelper(connectionString);

string tableName = "Users";

string columns = "Id, Name, Age";

using (var reader = helper.QueryData(tableName, columns))

{

while (reader.Read())

{

Console.WriteLine($"Id: {reader["Id"]}, Name: {reader["Name"]}, Age: {reader["Age"]}");

}

}

}

}

### 带条件查询

csharp

class Program

{

static void Main(string[] args)

{

string connectionString = "Data Source=example.db;Version=3;";

SQLiteHelper helper = new SQLiteHelper(connectionString);

string tableName = "Users";

string columns = "Id, Name, Age";

var conditions = new (string parameter, object value)[]

{

("HashValue=@HashValue", "abc123"),

("Age=@Age", 30)

};

using (var reader = helper.QueryData(tableName, columns, conditions))

{

while (reader.Read())

{

Console.WriteLine($"Id: {reader["Id"]}, Name: {reader["Name"]}, Age: {reader["Age"]}");

}

}

}

}

## 解释

### 1、参数化查询:

* + 使用 params (string parameter, object value)[] conditions 作为参数，允许调用方法时传递任意数量的条件。
  + 如果不传递任何条件，conditions 将默认为 null。
  + 在构建查询字符串时，将 parameters 数组中的每个元素添加到 WHERE 子句中。
  + 使用 command.Parameters.AddWithValue 方法将 values 数组中的值绑定到查询参数中，以防止 SQL 注入攻击。

### 2、动态构建查询:

* + 在 QueryData 方法中，检查 conditions 是否为 null 或空数组。
  + 如果 conditions 不为空，则遍历 conditions 数组，将每个条件添加到查询字符串中，并将对应的值绑定到 SQLiteCommand 的参数中。

### 3、返回 SQLiteDataReader:

* + command.ExecuteReader() 返回一个 SQLiteDataReader 对象，可以在 using 语句中使用它来读取查询结果。

这样，你就可以灵活地添加查询条件，并且代码更加安全和可维护。