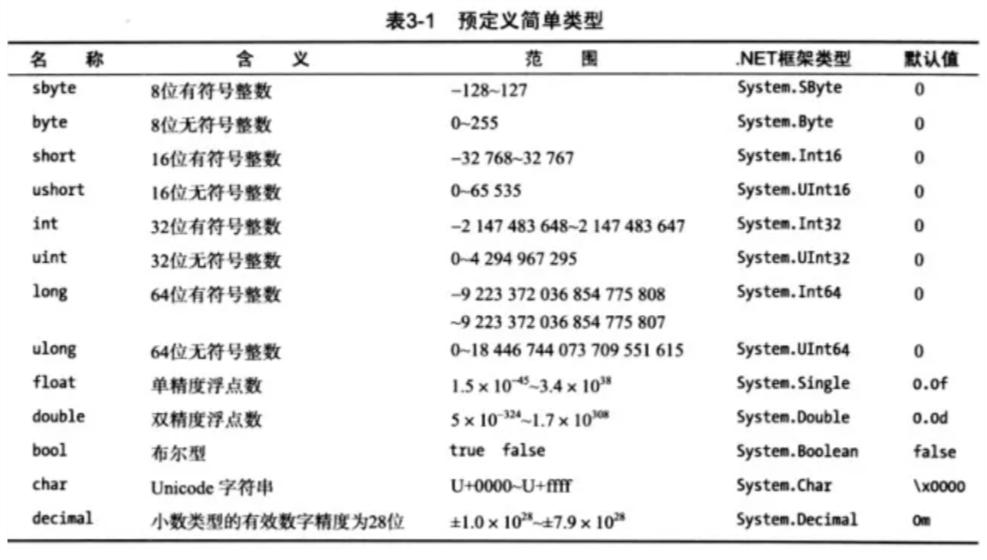
# C#基础

### 1、数据类型：引用类型和值类型





### 2、符号运算符==和Equals

在C#中，==和Equals方法都可以用来比较两个对象是否相等，但它们之间有一些重要的区别：

#### **语义差异**

* ==：对于值类型，==比较的是它们的值是否相等；对于引用类型，==比较的是两个引用是否指向同一个内存地址。
* Equals：是Object类的一个方法，默认情况下，Equals方法和==运算符的行为相同，即比较引用是否相等。然而，许多类（如String、Integer等）重写了Equals方法，使其比较对象的内容是否相等。1234

#### **使用场景**

* ==：适用于比较基本类型或需要检查两个引用是否指向同一对象的情况。
* Equals：适用于比较对象的内容是否相等，特别是在处理可能被子类重写的相等性逻辑时。15

#### **空引用处理**

* ==：可以安全地用于比较null引用，不会抛出异常。
* Equals：如果对象为null，调用Equals方法会抛出NullReferenceException，除非显式检查null或使用Equals的静态版本Equals(object a, object b)。5

#### **性能考虑**

* ==：通常比调用Equals方法更快，因为它是一个运算符，可以直接编译成机器指令。
* Equals：可能需要额外的函数调用开销，尤其是在没有重写的情况下，可能会涉及虚方法调用。5

#### **重写和自定义**

* ==：不能被重写，但可以为自定义类型提供运算符重载。
* Equals：可以被重写以提供自定义的相等性逻辑。在重写Equals方法时，通常也需要重写GetHashCode方法，以确保在散列集合中使用对象时的一致性。

### 3、try{}catch{}finally{}

try{}：正常代码块

catch{}：捕捉异常

finally{}：最后都会执行的代码，放这里。因此，while循环不能放在try{}catch外面。

### 4、C#form.listBox1没有访问级别，无法访问

当你遇到 “C# form.listBox1 没有访问级别，无法访问” 这个问题，通常是因为 listBox1 控件的访问修饰符默认是 private，在另一个类中无法直接访问。以下为你详细介绍解决该问题的几种方法：

#### **方法一：将 listBox1 的访问修饰符修改为 public**

在 Visual Studio 的设计器中，你可以修改控件的访问级别。具体操作步骤如下：

1. 打开窗体的设计视图。
2. 选中 ListBox 控件（listBox1）。
3. 在属性窗口中，找到 Modifiers 属性，将其从 private 改为 public。

#### **方法二：在 Form1 类中添加公共方法来操作 ListBox**

这种方法避免了直接暴露 ListBox 控件，而是通过公共方法来进行操作，提高了代码的封装性。以下是示例代码：

收起

csharp

using System;using System.Threading;using System.Windows.Forms;

namespace WindowsFormsApp1{

public partial class Form1 : Form

{

private DataAdder dataAdder;

public Form1()

{

InitializeComponent();

dataAdder = new DataAdder(this);

}

private void button1\_Click(object sender, EventArgs e)

{

dataAdder.StartAddingData();

}

// 提供一个公共方法用于向 ListBox 添加数据

public void AddItemToListBox(string item)

{

if (listBox1.InvokeRequired)

{

listBox1.Invoke((MethodInvoker)delegate

{

listBox1.Items.Add(item);

listBox1.SelectedIndex = listBox1.Items.Count - 1;

});

}

else

{

listBox1.Items.Add(item);

listBox1.SelectedIndex = listBox1.Items.Count - 1;

}

}

}

public class DataAdder

{

private Form1 form;

public DataAdder(Form1 form)

{

this.form = form;

}

public void StartAddingData()

{

Thread dataAddingThread = new Thread(AddDataToForm);

dataAddingThread.Start();

}

private void AddDataToForm()

{

for (int i = 0; i < 10; i++)

{

Thread.Sleep(1000);

string newData = $"Data {i}";

form.AddItemToListBox(newData);

}

}

}}

#### 代码解释

* **Form1**类****：
  + AddItemToListBox 方法是一个公共方法，用于向 ListBox 中添加数据。在方法内部，使用 InvokeRequired 检查是否需要跨线程调用，如果需要则使用 Invoke 方法在 UI 线程上执行添加操作。
* **DataAdder**类****：
  + 通过调用 form.AddItemToListBox 方法来向 ListBox 中添加数据，而不是直接访问 ListBox 控件。

#### 方法三：使用事件机制

可以定义一个事件，当需要添加数据时触发该事件，在窗体类中处理该事件并更新 ListBox。示例代码如下：

收起

csharp

using System;using System.Threading;using System.Windows.Forms;

namespace WindowsFormsApp1{

public partial class Form1 : Form

{

private DataAdder dataAdder;

public Form1()

{

InitializeComponent();

dataAdder = new DataAdder();

dataAdder.DataAdded += DataAdder\_DataAdded;

}

private void button1\_Click(object sender, EventArgs e)

{

dataAdder.StartAddingData();

}

private void DataAdder\_DataAdded(object sender, string e)

{

if (listBox1.InvokeRequired)

{

listBox1.Invoke((MethodInvoker)delegate

{

listBox1.Items.Add(e);

listBox1.SelectedIndex = listBox1.Items.Count - 1;

});

}

else

{

listBox1.Items.Add(e);

listBox1.SelectedIndex = listBox1.Items.Count - 1;

}

}

}

public class DataAdder

{

public event EventHandler<string> DataAdded;

public void StartAddingData()

{

Thread dataAddingThread = new Thread(AddDataToForm);

dataAddingThread.Start();

}

private void AddDataToForm()

{

for (int i = 0; i < 10; i++)

{

Thread.Sleep(1000);

string newData = $"Data {i}";

DataAdded?.Invoke(this, newData);

}

}

}}

#### 代码解释

* **DataAdder**类****：
  + 定义了一个 DataAdded 事件，当有新数据需要添加时触发该事件。
* **Form1**类****：
  + 在构造函数中订阅 DataAdded 事件，当事件触发时，在 DataAdder\_DataAdded 方法中更新 ListBox 控件。

以上几种方法都可以解决 form.listBox1 无法访问的问题，你可以根据具体需求选择合适的方法。

### 5、继承问题：

#### 1、C#程序中有两个类，A类和B类，B类继承A类，A类和B类有两个同样的方法，B类调用方法的时候，调用的是自己的方法还是继承自A类的方法？

在 C# 中，当 B 类继承 A 类且 A 类和 B 类有两个同样的方法时，B 类调用该方法的情况取决于方法的定义方式，以下是具体介绍：

##### A类中方法没有特殊修饰

* 如果 A 类中的方法就是一个普通方法，B 类中出现了一个签名（方法名、参数列表、返回类型）完全相同的方法，这种情况下会产生编译警告，提示 B 类中的方法隐藏了 A 类中的方法。在 B 类实例调用该方法时，默认会调用 B 类自己的方法。

##### A 类方法用 virtual 修饰，B 类方法用 override 修饰

* 当 A 类中的方法被定义为virtual（虚方法），表示该方法可以在子类中被重写。B 类中使用override关键字来重写 A 类中的虚方法，这种情况下，B 类实例调用该方法时，会调用 B 类中重写后的方法。这是实现多态性的一种常见方式，通过这种方式，子类可以根据自身的需求对父类的方法进行定制化实现。

##### A 类方法没有用 virtual 修饰，B 类方法用 new 修饰

* 如果 A 类中的方法不是虚方法，而 B 类中使用new关键字来定义了一个与 A 类中方法签名相同的方法，这表示 B 类中的方法是一个新的方法，它隐藏了 A 类中的同名方法。当通过 B 类的实例调用该方法时，会调用 B 类中用new修饰的这个新方法。

以下是一个简单的示例代码来演示上述情况：

收起

csharp

class A{

// 普通方法

public void DoSomething()

{

Console.WriteLine("A.DoSomething");

}

// 虚方法

public virtual void DoSomethingVirtual()

{

Console.WriteLine("A.DoSomethingVirtual");

}}

class B : A{

// 隐藏A类中的DoSomething方法

public void DoSomething()

{

Console.WriteLine("B.DoSomething");

}

// 重写A类中的DoSomethingVirtual方法

public override void DoSomethingVirtual()

{

Console.WriteLine("B.DoSomethingVirtual");

}

// 隐藏A类中的DoSomethingVirtual方法，创建新的方法

public new void DoSomethingVirtualNew()

{

Console.WriteLine("B.DoSomethingVirtualNew");

}}

class Program{

static void Main()

{

B b = new B();

b.DoSomething(); // 调用B类自己的DoSomething方法

b.DoSomethingVirtual(); // 调用B类重写后的DoSomethingVirtual方法

b.DoSomethingVirtualNew(); // 调用B类中用new修饰的DoSomethingVirtualNew方法

A a = b as A;

a.DoSomething(); // 调用A类的DoSomething方法

a.DoSomethingVirtual(); // 调用B类重写后的DoSomethingVirtual方法

a.DoSomethingVirtualNew(); // 调用A类的DoSomethingVirtual方法，因为A类没有被隐藏的版本

}}

### 6、事件：

#### 1、事件在C#中是如何实现的？

在 C# 中，事件是一种特殊的委托实例，它允许类或对象在特定的事情发生时通知其他类或对象。下面从多个方面详细介绍事件在 C# 中的实现。

##### 1. 事件的基本实现步骤

###### 1.1 定义委托类型

委托定义了事件处理程序的方法签名，事件处理程序必须与委托的签名相匹配。

收起

csharp

// 定义一个委托类型，用于表示事件处理程序的方法签名public delegate void MyEventHandler(object sender, EventArgs e);

上述代码定义了一个名为 MyEventHandler 的委托类型，它接受两个参数：一个 object 类型的 sender 表示事件的发送者，一个 EventArgs 类型的 e 包含事件的相关信息。

###### 1.2 声明事件

在一个类中声明事件，使用之前定义的委托类型。

收起

csharp

public class EventPublisher{

// 声明一个事件，使用之前定义的委托类型

public event MyEventHandler MyEvent;

// 触发事件的方法

public void RaiseEvent()

{

// 检查事件是否有订阅者

if (MyEvent != null)

{

// 触发事件，传递事件发送者和事件参数

MyEvent(this, EventArgs.Empty);

}

// 也可以使用空条件运算符简化代码

// MyEvent?.Invoke(this, EventArgs.Empty);

}}

在 EventPublisher 类中，声明了一个名为 MyEvent 的事件，它的类型是之前定义的 MyEventHandler 委托。RaiseEvent 方法用于触发事件，在触发事件之前会检查是否有订阅者。

###### 1.3 订阅事件

其他类或对象可以订阅该事件，并提供相应的事件处理程序。

收起

csharp

public class EventSubscriber{

// 事件处理方法，与委托的签名相匹配

public void HandleEvent(object sender, EventArgs e)

{

Console.WriteLine("事件已触发！");

}}

class Program{

static void Main()

{

EventPublisher publisher = new EventPublisher();

EventSubscriber subscriber = new EventSubscriber();

// 订阅事件，将订阅者的事件处理方法添加到事件的调用列表中

publisher.MyEvent += subscriber.HandleEvent;

// 触发事件

publisher.RaiseEvent();

}}

在 Main 方法中，创建了 EventPublisher 和 EventSubscriber 的实例，然后使用 += 运算符将 subscriber 的 HandleEvent 方法订阅到 publisher 的 MyEvent 事件上。最后调用 RaiseEvent 方法触发事件。

##### 2. 使用内置委托类型

C# 提供了一些内置的委托类型，如 EventHandler 和 EventHandler<TEventArgs>，可以直接使用这些委托类型来定义事件，而无需自己定义委托类型。

收起

csharp

public class EventPublisher{

// 使用内置的 EventHandler 委托类型声明事件

public event EventHandler MyEvent;

public void RaiseEvent()

{

MyEvent?.Invoke(this, EventArgs.Empty);

}}

public class EventSubscriber{

public void HandleEvent(object sender, EventArgs e)

{

Console.WriteLine("事件已触发！");

}}

class Program{

static void Main()

{

EventPublisher publisher = new EventPublisher();

EventSubscriber subscriber = new EventSubscriber();

publisher.MyEvent += subscriber.HandleEvent;

publisher.RaiseEvent();

}}

EventHandler 是一个常用的内置委托类型，它的签名与前面自定义的 MyEventHandler 相同。

##### 3. 自定义事件参数

如果需要在事件中传递更多的信息，可以自定义事件参数类，继承自 EventArgs 类。

收起

csharp

// 自定义事件参数类，继承自 EventArgspublic class MyEventArgs : EventArgs{

public string Message { get; set; }

public MyEventArgs(string message)

{

Message = message;

}}

public class EventPublisher{

// 使用 EventHandler<MyEventArgs> 委托类型声明事件

public event EventHandler<MyEventArgs> MyEvent;

public void RaiseEvent()

{

MyEventArgs args = new MyEventArgs("这是自定义事件参数！");

MyEvent?.Invoke(this, args);

}}

public class EventSubscriber{

public void HandleEvent(object sender, MyEventArgs e)

{

Console.WriteLine($"事件已触发，消息：{e.Message}");

}}

class Program{

static void Main()

{

EventPublisher publisher = new EventPublisher();

EventSubscriber subscriber = new EventSubscriber();

publisher.MyEvent += subscriber.HandleEvent;

publisher.RaiseEvent();

}}

在这个例子中，定义了一个自定义的事件参数类 MyEventArgs，它包含一个 Message 属性。EventPublisher 类中的 MyEvent 事件使用 EventHandler<MyEventArgs> 委托类型，在触发事件时传递自定义的事件参数。

##### 4. 事件的底层实现原理

从底层来看，事件实际上是对委托的一种封装。事件提供了一种安全的方式来使用委托，它只允许在声明事件的类内部调用委托，外部类只能进行订阅（使用 +=）和取消订阅（使用 -=）操作，不能直接调用委托。

例如，上述的 EventPublisher 类中的 MyEvent 事件，在编译器处理后，实际上会生成一个私有委托字段和对应的 add 和 remove 访问器方法。add 访问器用于添加事件处理程序，remove 访问器用于移除事件处理程序。可以手动实现这些访问器来更精细地控制事件的订阅和取消订阅过程：

收起

csharp

public class EventPublisher{

private EventHandler<MyEventArgs> myEvent;

public event EventHandler<MyEventArgs> MyEvent

{

add

{

// 自定义添加事件处理程序的逻辑

Console.WriteLine("添加事件处理程序");

myEvent += value;

}

remove

{

// 自定义移除事件处理程序的逻辑

Console.WriteLine("移除事件处理程序");

myEvent -= value;

}

}

public void RaiseEvent()

{

MyEventArgs args = new MyEventArgs("这是自定义事件参数！");

myEvent?.Invoke(this, args);

}}

在这个例子中，手动实现了 MyEvent 事件的 add 和 remove 访问器，在添加和移除事件处理程序时会输出相应的信息。

综上所述，C# 中的事件通过委托机制实现，提供了一种松耦合的方式来实现对象之间的通信。可以根据具体需求自定义委托类型、事件参数和事件的访问器逻辑。

#### 2、C#事件Event只能通过+=方式来订阅吗？

在 C# 中，事件（Event）主要通过+= 运算符来订阅事件处理程序，但并非只能通过这种方式，下面详细介绍：

##### 1. 使用 += 运算符订阅事件

这是最常见、最标准的订阅事件的方式。通过 += 可以将一个方法添加到事件的调用列表中。当事件触发时，这些订阅的方法都会被依次调用。

****示例代码****：

收起

csharp

using System;

// 定义一个包含事件的类public class EventPublisher{

// 定义事件

public event EventHandler MyEvent;

// 触发事件的方法

public void RaiseEvent()

{

MyEvent?.Invoke(this, EventArgs.Empty);

}}

// 定义一个事件订阅者类public class EventSubscriber{

// 事件处理方法

public void HandleEvent(object sender, EventArgs e)

{

Console.WriteLine("事件已触发！");

}}

class Program{

static void Main()

{

EventPublisher publisher = new EventPublisher();

EventSubscriber subscriber = new EventSubscriber();

// 使用 += 订阅事件

publisher.MyEvent += subscriber.HandleEvent;

// 触发事件

publisher.RaiseEvent();

}}

在上述代码中，publisher.MyEvent += subscriber.HandleEvent; 这行代码将 subscriber 对象的 HandleEvent 方法订阅到 publisher 对象的 MyEvent 事件上。

##### 2. 使用委托实例化的方式订阅事件

从技术上来说，你可以在事件声明时使用委托实例来初始化事件，虽然这种方式并不常见。

****示例代码****：

收起

csharp

using System;

public class EventPublisher{

public event EventHandler MyEvent = (sender, e) => Console.WriteLine("默认事件处理程序");

public void RaiseEvent()

{

MyEvent?.Invoke(this, EventArgs.Empty);

}}

class Program{

static void Main()

{

EventPublisher publisher = new EventPublisher();

publisher.RaiseEvent();

}}

在这个例子中，MyEvent 事件在声明时就被初始化为一个匿名方法，这相当于订阅了一个默认的事件处理程序。

##### 3. 动态订阅事件

可以使用反射在运行时动态地订阅事件。这种方式比较复杂，通常用于需要在运行时根据某些条件决定是否订阅事件的场景。

****示例代码****：

收起

csharp

using System;using System.Reflection;

public class EventPublisher{

public event EventHandler MyEvent;

public void RaiseEvent()

{

MyEvent?.Invoke(this, EventArgs.Empty);

}}

public class EventSubscriber{

public void HandleEvent(object sender, EventArgs e)

{

Console.WriteLine("事件已触发！");

}}

class Program{

static void Main()

{

EventPublisher publisher = new EventPublisher();

EventSubscriber subscriber = new EventSubscriber();

// 使用反射获取事件信息

EventInfo eventInfo = typeof(EventPublisher).GetEvent("MyEvent");

MethodInfo handlerMethod = typeof(EventSubscriber).GetMethod("HandleEvent");

// 创建委托实例

Delegate handlerDelegate = Delegate.CreateDelegate(eventInfo.EventHandlerType, subscriber, handlerMethod);

// 使用反射订阅事件

eventInfo.AddEventHandler(publisher, handlerDelegate);

// 触发事件

publisher.RaiseEvent();

}}

在这个例子中，通过反射获取事件信息和事件处理方法信息，然后创建委托实例并使用 AddEventHandler 方法订阅事件。

综上所述，虽然 += 是最常用的事件订阅方式，但 C# 还提供了其他方式来订阅事件，以满足不同的编程需求

# 小项目练习

## 1、使用 EF Core 来完成打开 MySQL 数据库、

创建数据表、

进行对象深度复制并将其存储到数据库的操作、

将数据库配置参数保存到本地配置文件。

### **1. 项目准备**

首先，你需要创建一个 C# 项目（例如控制台应用程序），并通过 NuGet 包管理器安装以下必要的库：

* Microsoft.EntityFrameworkCore：EF Core 的核心库。
* Pomelo.EntityFrameworkCore.MySql：用于与 MySQL 数据库交互的 EF Core 提供程序。
* Newtonsoft.Json：用于实现对象的深度复制。

### **2. 定义实体类**

定义要存储到数据库的实体类，这里以 Person 和 Address 为例，Person 类包含一个 Address 类型的引用，体现了实体之间的关联关系。

收起

csharp

using Newtonsoft.Json;

// 定义地址类public class Address{

public int Id { get; set; }

public string Street { get; set; }

public string City { get; set; }

public string ZipCode { get; set; }}

// 定义人员类，包含地址引用public class Person{

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

public Address Address { get; set; }

// 实现深度复制方法

public Person DeepCopy()

{

string json = JsonConvert.SerializeObject(this);

return JsonConvert.DeserializeObject<Person>(json);

}}

### **3. 创建 DbContext 类**

DbContext 类是 EF Core 与数据库交互的核心，它负责管理数据库连接、实体集合以及数据操作。

收起

csharp

using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext{

public DbSet<Person> Persons { get; set; }

public DbSet<Address> Addresses { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

// 配置数据库连接字符串，需根据实际情况修改

string connectionString = "server=localhost;user=root;password=your\_password;database=your\_database;AllowPublicKeyRetrieval=true;";

optionsBuilder.UseMySql(connectionString, ServerVersion.AutoDetect(connectionString));

}}

#### 参数：MySQL 的AllowPublicKeyRetrieval 是什么？

AllowPublicKeyRetrieval 是 MySQL 连接字符串中的一个参数，主要用于处理 MySQL 客户端与服务器之间的公钥检索问题，下面从其作用、使用场景、安全风险以及配置示例几个方面详细介绍：

#### 作用

在 MySQL 8.0 及更高版本中，默认使用 caching\_sha2\_password 作为认证插件，这种认证方式在某些情况下需要客户端从服务器获取公钥来进行加密通信。AllowPublicKeyRetrieval 参数决定了客户端是否允许从服务器检索公钥。

#### 使用场景

* ****默认不允许情况****：当 AllowPublicKeyRetrieval 设置为 false（默认值）时，如果客户端需要公钥但没有提前配置，就会导致认证失败。例如，使用旧版本的 MySQL 连接器或者没有正确配置公钥的情况下，连接会报错。
* ****允许公钥检索****：将 AllowPublicKeyRetrieval 设置为 true，客户端在认证过程中如果需要公钥，会主动向服务器发起请求获取公钥，从而完成认证过程。这种方式适用于大多数需要使用 caching\_sha2\_password 认证插件的场景，尤其是在没有提前配置公钥的情况下。

#### 安全风险

将 AllowPublicKeyRetrieval 设置为 true 会带来一定的安全风险。因为允许客户端从服务器检索公钥可能会导致中间人攻击（MITM）的风险增加。攻击者可能会拦截客户端与服务器之间的通信，篡改公钥信息，从而窃取用户的敏感信息。因此，在生产环境中，如果不是必要情况，不建议将该参数设置为 true。

#### 配置示例

在不同的编程语言和框架中，配置 AllowPublicKeyRetrieval 参数的方式略有不同，以下是一些常见的示例：

#### **1.C# 使用 EF Core 连接 MySQL**

收起

csharp

using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext{

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

string connectionString = "server=localhost;user=root;password=your\_password;database=your\_database;AllowPublicKeyRetrieval=true;";

optionsBuilder.UseMySql(connectionString, ServerVersion.AutoDetect(connectionString));

}}

#### **2.Java 使用 JDBC 连接 MySQL**

收起

java

import java.sql.Connection;import java.sql.DriverManager;import java.sql.SQLException;

public class MySQLConnectionExample {

public static void main(String[] args) {

String url = "jdbc:mysql://localhost:3306/your\_database?AllowPublicKeyRetrieval=true";

String user = "root";

String password = "your\_password";

try (Connection connection = DriverManager.getConnection(url, user, password)) {

System.out.println("Connected to the database!");

} catch (SQLException e) {

e.printStackTrace();

}

}}

#### 替代方案

为了避免使用 AllowPublicKeyRetrieval=true 带来的安全风险，可以采取以下替代方案：

* ****使用旧的认证插件****：将 MySQL 用户的认证插件改为 mysql\_native\_password，这样就不需要进行公钥检索。可以使用以下 SQL 语句修改用户的认证插件：

收起

sql

ALTER USER 'your\_user'@'localhost' IDENTIFIED WITH mysql\_native\_password BY 'your\_password';

* ****提前配置公钥****：在客户端配置文件中提前配置好服务器的公钥，避免在认证过程中进行公钥检索。不同的客户端和框架配置公钥的方式不同，需要参考相应的文档进行配置。

分享

在 C# 中使用 EF Core 连接 MySQL 时，如何配置其他连接参数？

除了 EF Core，还有哪些常见的.NET 数据库操作框架？

如何在生产环境中安全地配置 MySQL 的 AllowPublicKeyRetrieval 参数？

#### EF的类DbContext

分享

DbContext 是 Entity Framework（EF）和 Entity Framework Core 中一个非常核心的类，它在数据访问层扮演着至关重要的角色，下面从多个方面为你详细介绍。

#### 定义与作用

DbContext 是 Entity Framework 中的一个抽象类，充当数据库与应用程序之间的桥梁。它封装了与数据库交互的逻辑，负责管理实体对象的生命周期、跟踪实体状态的变化，以及将这些变化持久化到数据库中。可以把它看作是数据库在应用程序中的一个会话，通过它可以执行查询、插入、更新和删除等操作。

#### 主要功能

1. ****实体集管理****：DbContext 包含多个 DbSet<TEntity> 属性，每个 DbSet<TEntity> 对应数据库中的一个表，代表了该表中所有实体对象的集合。通过 DbSet<TEntity>，可以对数据库表中的数据进行增删改查操作。
2. ****状态跟踪****：DbContext 会跟踪实体对象的状态，例如新增、修改、删除或未修改。当调用 SaveChanges 方法时，DbContext 会根据实体的状态生成相应的 SQL 语句，将这些更改保存到数据库中。
3. ****数据库连接管理****：DbContext 负责管理与数据库的连接，包括打开、关闭连接，以及处理连接池等。
4. ****事务管理****：可以使用 DbContext 来启动、提交或回滚事务，确保数据操作的一致性和完整性。

#### 使用示例

#### **1. 创建 DbContext 派生类**

首先，需要创建一个继承自 DbContext 的类，该类将作为应用程序与数据库之间的交互接口。以下是一个简单的示例：

收起

csharp

using Microsoft.EntityFrameworkCore;

public class MyDbContext : DbContext{

public DbSet<Product> Products { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

// 配置数据库连接字符串

optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=MyDatabase;Trusted\_Connection=True;");

}

}}

public class Product{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }}

#### **2. 使用 DbContext 进行数据操作**

以下是一些常见的数据操作示例：

##### **查询数据**

收起

csharp

using (var context = new MyDbContext()){

var products = context.Products.ToList();

foreach (var product in products)

{

Console.WriteLine($"Id: {product.Id}, Name: {product.Name}, Price: {product.Price}");

}}

##### **插入数据**

收起

csharp

using (var context = new MyDbContext()){

var newProduct = new Product { Name = "New Product", Price = 9.99m };

context.Products.Add(newProduct);

context.SaveChanges();}

##### **更新数据**

收起

csharp

using (var context = new MyDbContext()){

var productToUpdate = context.Products.Find(1);

if (productToUpdate != null)

{

productToUpdate.Price = 19.99m;

context.SaveChanges();

}}

##### **删除数据**

收起

csharp

using (var context = new MyDbContext()){

var productToDelete = context.Products.Find(1);

if (productToDelete != null)

{

context.Products.Remove(productToDelete);

context.SaveChanges();

}}

#### 生命周期管理

DbContext 是一个相对轻量级的对象，但它维护着与数据库的连接和实体状态的跟踪信息，因此需要合理管理其生命周期。

一般来说，DbContext 的生命周期应该尽量短，以避免长时间占用数据库连接资源。在 [ASP.NET](https://asp.net/" \t "https://www.doubao.com/chat/_blank) Core 应用程序中，可以使用依赖注入来管理 DbContext 的生命周期，通常采用 Scoped 生命周期，即每个 HTTP 请求创建一个新的 DbContext 实例，并在请求结束时释放该实例。

#### 总结

DbContext 是 Entity Framework 和 Entity Framework Core 中不可或缺的一部分，它为开发人员提供了一个方便、高效的方式来与数据库进行交互。通过合理使用 DbContext，可以简化数据访问层的开发，提高代码的可维护性和可测试性。

### **4. 配置 MySQL 数据库访问权限**

在 MySQL 中，你需要确保用户具有创建表和读写数据的权限。可以使用以下 SQL 语句为用户授予相应的权限：

收起

sql

-- 授予用户在指定数据库上的所有权限：

GRANT ALL PRIVILEGES ON your\_database.\* TO 'your\_username'@'localhost';

-- 刷新权限使更改生效：

FLUSH PRIVILEGES;

将 your\_database 替换为你实际使用的数据库名，your\_username 替换为你使用的 MySQL 用户名。

#### C# 配置 MySQL 数据库访问权限有哪些方式？权限有哪些？如何配置？

在 C# 中本身不能直接配置 MySQL 数据库的访问权限，但可以通过执行 SQL 语句或者调用 MySQL 管理工具相关的 API 来实现权限的配置。以下详细介绍相关信息：

#### 常见的权限类型

MySQL 提供了多种权限，以下是一些常见的权限及其作用：

##### ****数据操作权限****

* + SELECT：允许用户从表中查询数据。
  + INSERT：允许用户向表中插入新的数据记录。
  + UPDATE：允许用户修改表中的现有数据。
  + DELETE：允许用户从表中删除数据记录。

##### ****数据库结构操作权限****

* + CREATE：允许用户创建数据库、表、索引等数据库对象。
  + ALTER：允许用户修改数据库对象的结构，如修改表的列定义。
  + DROP：允许用户删除数据库、表等数据库对象。

##### ****用户和权限管理权限****

* + CREATE USER：允许创建新的 MySQL 用户。
  + GRANT OPTION：允许用户将自己拥有的权限授予其他用户。
  + REVOKE：允许用户撤销其他用户的权限。

#### 配置权限方式及具体操作

##### **1. 使用 MySqlConnector 执行 SQL 语句**

MySqlConnector 是一个用于在 .NET 应用程序中连接和操作 MySQL 数据库的开源库。以下是一个使用 MySqlConnector 执行 SQL 语句来配置 MySQL 数据库访问权限的示例：

收起

csharp

using MySqlConnector;using System;

class Program{

static void Main()

{

// 数据库连接字符串，使用具有足够权限的用户登录

string connectionString = "Server=localhost;Database=mysql;Uid=root;Pwd=your\_password;";

try

{

using (MySqlConnection connection = new MySqlConnection(connectionString))

{

// 打开数据库连接

connection.Open();

// 创建一个新用户

string createUserQuery = "CREATE USER 'newuser'@'%' IDENTIFIED BY 'newpassword';";

using (MySqlCommand createUserCommand = new MySqlCommand(createUserQuery, connection))

{

createUserCommand.ExecuteNonQuery();

}

// 授予新用户对指定数据库的 SELECT 和 INSERT 权限

string grantPermissionsQuery = "GRANT SELECT, INSERT ON your\_database.\* TO 'newuser'@'%';";

using (MySqlCommand grantPermissionsCommand = new MySqlCommand(grantPermissionsQuery, connection))

{

grantPermissionsCommand.ExecuteNonQuery();

}

// 刷新权限使更改生效

string flushPrivilegesQuery = "FLUSH PRIVILEGES;";

using (MySqlCommand flushPrivilegesCommand = new MySqlCommand(flushPrivilegesQuery, connection))

{

flushPrivilegesCommand.ExecuteNonQuery();

}

Console.WriteLine("用户创建和权限授予成功。");

}

}

catch (Exception ex)

{

Console.WriteLine($"发生错误: {ex.Message}");

}

}}

****代码解释****

1. ****连接数据库****：使用 MySqlConnection 类创建一个数据库连接对象，并打开连接。需要使用具有足够权限（如 root 用户）的用户登录，以便执行用户创建和权限授予操作。
2. ****创建新用户****：执行 CREATE USER 语句创建一个新的 MySQL 用户。
3. ****授予权限****：执行 GRANT 语句为新用户授予对指定数据库（your\_database）中所有表的 SELECT 和 INSERT 权限。
4. ****刷新权限****：执行 FLUSH PRIVILEGES 语句使权限更改立即生效。

##### 2. 调用 MySQL 管理工具的 API（较复杂，一般不常用）

一些 MySQL 管理工具提供了 API 供外部程序调用，通过调用这些 API 可以实现权限配置。但这种方式相对复杂，需要了解管理工具的 API 文档和使用方法。例如，某些云服务提供商提供的 MySQL 数据库管理服务，可能会有相应的 SDK 来管理数据库用户和权限。

**注意事项**

* ****权限管理安全****：在配置权限时，应遵循最小权限原则，只授予用户完成其工作所需的最少权限，以降低安全风险。
* ****异常处理****：在执行 SQL 语句时，可能会出现各种异常，如数据库连接失败、SQL 语法错误等，需要进行适当的异常处理。
* ****权限刷新****：每次修改用户权限后，都需要执行 FLUSH PRIVILEGES 语句使更改生效。

##### 1. MySQL.Data

* ****简介****：MySQL.Data 是 MySQL 官方提供的用于 .NET 平台的 [ADO.NET](https://ado.net/" \t "https://www.doubao.com/chat/_blank) 数据提供程序，它提供了一系列类和方法，用于在 C# 程序中连接、操作和管理 MySQL 数据库，也可用于配置数据库访问权限。
* ****使用示例****

收起

csharp

using System;using MySql.Data.MySqlClient;

class Program{

static void Main()

{

// 数据库连接字符串，使用具有足够权限的用户登录

string connectionString = "Server=localhost;Database=mysql;Uid=root;Pwd=your\_password;";

try

{

using (MySqlConnection connection = new MySqlConnection(connectionString))

{

// 打开数据库连接

connection.Open();

// 创建一个新用户

string createUserQuery = "CREATE USER 'newuser'@'%' IDENTIFIED BY 'newpassword';";

using (MySqlCommand createUserCommand = new MySqlCommand(createUserQuery, connection))

{

createUserCommand.ExecuteNonQuery();

}

// 授予新用户对指定数据库的 SELECT 和 INSERT 权限

string grantPermissionsQuery = "GRANT SELECT, INSERT ON your\_database.\* TO 'newuser'@'%';";

using (MySqlCommand grantPermissionsCommand = new MySqlCommand(grantPermissionsQuery, connection))

{

grantPermissionsCommand.ExecuteNonQuery();

}

// 刷新权限使更改生效

string flushPrivilegesQuery = "FLUSH PRIVILEGES;";

using (MySqlCommand flushPrivilegesCommand = new MySqlCommand(flushPrivilegesQuery, connection))

{

flushPrivilegesCommand.ExecuteNonQuery();

}

Console.WriteLine("用户创建和权限授予成功。");

}

}

catch (Exception ex)

{

Console.WriteLine($"发生错误: {ex.Message}");

}

}}

* ****特点****：
  + 官方支持，与 MySQL 数据库的兼容性好。
  + 功能全面，提供了丰富的类和方法来处理各种数据库操作。

##### **2. Dapper**

* ****简介****：Dapper 是一个轻量级的对象映射器，它本身不是专门针对 MySQL 的库，但可以与 MySqlConnector 或 MySQL.Data 结合使用，简化数据库操作代码。通过它可以更方便地执行 SQL 语句来配置 MySQL 数据库访问权限。
* ****使用示例****

收起

csharp

using System;using Dapper;using MySqlConnector;

class Program{

static void Main()

{

// 数据库连接字符串，使用具有足够权限的用户登录

string connectionString = "Server=localhost;Database=mysql;Uid=root;Pwd=your\_password;";

try

{

using (MySqlConnection connection = new MySqlConnection(connectionString))

{

// 打开数据库连接

connection.Open();

// 创建一个新用户

string createUserQuery = "CREATE USER 'newuser'@'%' IDENTIFIED BY 'newpassword';";

connection.Execute(createUserQuery);

// 授予新用户对指定数据库的 SELECT 和 INSERT 权限

string grantPermissionsQuery = "GRANT SELECT, INSERT ON your\_database.\* TO 'newuser'@'%';";

connection.Execute(grantPermissionsQuery);

// 刷新权限使更改生效

string flushPrivilegesQuery = "FLUSH PRIVILEGES;";

connection.Execute(flushPrivilegesQuery);

Console.WriteLine("用户创建和权限授予成功。");

}

}

catch (Exception ex)

{

Console.WriteLine($"发生错误: {ex.Message}");

}

}}

* ****特点****：
  + 性能高，在执行 SQL 查询和命令时速度快。
  + 代码简洁，通过扩展方法简化了数据库操作代码。

##### **3. Entity Framework Core with Pomelo**

* ****简介****：Entity Framework Core 是微软开发的一个跨平台的对象关系映射（ORM）框架，Pomelo.EntityFrameworkCore.MySql 是用于 MySQL 数据库的 EF Core 提供程序，使用它可以通过面向对象的方式来操作数据库，包括配置数据库访问权限。
* ****使用示例****

收起

csharp

using Microsoft.EntityFrameworkCore;

using System;

// 定义 DbContext 类

public class MyDbContext : DbContext{

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

string connectionString = "Server=localhost;Database=mysql;Uid=root;Pwd=your\_password;";

optionsBuilder.UseMySql(connectionString, ServerVersion.AutoDetect(connectionString));

}}

class Program{

static void Main()

{

try

{

using (var context = new MyDbContext())

{

// 创建一个新用户

string createUserQuery = "CREATE USER 'newuser'@'%' IDENTIFIED BY 'newpassword';";

context.Database.ExecuteSqlRaw(createUserQuery);

// 授予新用户对指定数据库的 SELECT 和 INSERT 权限

string grantPermissionsQuery = "GRANT SELECT, INSERT ON your\_database.\* TO 'newuser'@'%';";

context.Database.ExecuteSqlRaw(grantPermissionsQuery);

// 刷新权限使更改生效

string flushPrivilegesQuery = "FLUSH PRIVILEGES;";

context.Database.ExecuteSqlRaw(flushPrivilegesQuery);

Console.WriteLine("用户创建和权限授予成功。");

}

}

catch (Exception ex)

{

Console.WriteLine($"发生错误: {ex.Message}");

}

}}

* ****特点****：
  + 面向对象编程，通过实体类和 DbContext 来操作数据库，提高了开发效率。
  + 支持数据库迁移，方便进行数据库结构的版本控制。

#### C#可以在哪里配置 MySQL 数据库访问权限?

分享

在 C# 里，自身无法直接对 MySQL 数据库的访问权限进行配置，不过可以借助执行 SQL 语句或者调用 MySQL 管理工具的 API 来实现权限配置。下面详细介绍不同的配置途径：

##### 1.通过代码执行 SQL 语句配置

在 C# 代码里执行 MySQL 的 SQL 语句，从而完成数据库访问权限的配置。常用的库有 MySqlConnector 和 MySQL.Data。

##### **使用 MySqlConnector**

收起

csharp

using MySqlConnector;using System;

class Program{

static void Main()

{

string connectionString = "Server=localhost;Database=mysql;Uid=root;Pwd=your\_password;";

try

{

using (MySqlConnection connection = new MySqlConnection(connectionString))

{

connection.Open();

// 创建新用户

string createUserQuery = "CREATE USER 'newuser'@'%' IDENTIFIED BY 'newpassword';";

using (MySqlCommand createUserCommand = new MySqlCommand(createUserQuery, connection))

{

createUserCommand.ExecuteNonQuery();

}

// 授予权限

string grantPermissionsQuery = "GRANT SELECT, INSERT ON your\_database.\* TO 'newuser'@'%';";

using (MySqlCommand grantPermissionsCommand = new MySqlCommand(grantPermissionsQuery, connection))

{

grantPermissionsCommand.ExecuteNonQuery();

}

// 刷新权限

string flushPrivilegesQuery = "FLUSH PRIVILEGES;";

using (MySqlCommand flushPrivilegesCommand = new MySqlCommand(flushPrivilegesQuery, connection))

{

flushPrivilegesCommand.ExecuteNonQuery();

}

Console.WriteLine("权限配置成功");

}

}

catch (Exception ex)

{

Console.WriteLine($"发生错误: {ex.Message}");

}

}}

##### **使用 MySQL.Data**

收起

csharp

using MySql.Data.MySqlClient;using System;

class Program{

static void Main()

{

string connectionString = "Server=localhost;Database=mysql;Uid=root;Pwd=your\_password;";

try

{

using (MySqlConnection connection = new MySqlConnection(connectionString))

{

connection.Open();

// 创建新用户

string createUserQuery = "CREATE USER 'newuser'@'%' IDENTIFIED BY 'newpassword';";

using (MySqlCommand createUserCommand = new MySqlCommand(createUserQuery, connection))

{

createUserCommand.ExecuteNonQuery();

}

// 授予权限

string grantPermissionsQuery = "GRANT SELECT, INSERT ON your\_database.\* TO 'newuser'@'%';";

using (MySqlCommand grantPermissionsCommand = new MySqlCommand(grantPermissionsQuery, connection))

{

grantPermissionsCommand.ExecuteNonQuery();

}

// 刷新权限

string flushPrivilegesQuery = "FLUSH PRIVILEGES;";

using (MySqlCommand flushPrivilegesCommand = new MySqlCommand(flushPrivilegesQuery, connection))

{

flushPrivilegesCommand.ExecuteNonQuery();

}

Console.WriteLine("权限配置成功");

}

}

catch (Exception ex)

{

Console.WriteLine($"发生错误: {ex.Message}");

}

}}

##### **2.借助 MySQL 管理工具**

除了代码方式，还能使用 MySQL 管理工具来配置访问权限，这些工具可以独立于 C# 程序运行。

##### **MySQL Workbench**

这是 MySQL 官方推出的可视化管理工具，通过它能图形化地配置数据库访问权限，具体步骤如下：

1. 开启 MySQL Workbench 并连接到 MySQL 服务器。
2. 选择 “管理” -> “用户和权限”。
3. 在 “用户和权限” 窗口，点击 “添加账户” 创建新用户，或者选择已有用户进行权限修改。
4. 在 “登录信息” 选项卡设置用户名、主机和密码。
5. 在 “架构权限” 选项卡选择要授予权限的数据库和表，勾选相应权限。
6. 在 “管理权限” 选项卡可授予高级管理权限。
7. 配置完成后，点击 “应用” 保存更改。

##### **Navicat for MySQL**

Navicat 是一款功能强大的数据库管理工具，其配置权限的步骤如下：

1. 打开 Navicat 并连接到 MySQL 服务器。
2. 右键点击连接，选择 “用户管理”。
3. 在 “用户管理” 窗口，点击 “新建用户” 创建新用户，或选择已有用户修改权限。
4. 在 “常规” 选项卡设置用户名、主机和密码。
5. 在 “权限” 选项卡选择要授予的权限，可按数据库、表或具体操作进行设置。
6. 配置完成后，点击 “确定” 保存更改。

**注意事项**

* ****安全问题****：在配置权限时，要遵循最小权限原则，仅授予用户完成工作所需的最少权限，以此降低安全风险。
* ****权限刷新****：每次使用 SQL 语句修改用户权限后，都要执行 FLUSH PRIVILEGES 语句使更改立即生效。

### **5. 使用迁移创建数据表———失败(已解决)**

在包管理器控制台中执行以下命令来创建迁移文件并更新数据库，从而创建相应的数据表：

Add-Migration InitialCreate

Update-Database

* Add-Migration InitialCreate：该命令会根据定义的实体类和 DbContext 生成一个迁移文件，记录数据库结构的变化。
* Update-Database：此命令会将迁移文件中的更改应用到数据库，创建相应的数据表。

////办法二：使用Add-Migration InitialCreate创建迁移文件就之后，还可以在代码中使用Migrate应用迁移，而不是在程序包管理控制台执行指令Update-Database。

using (var context = new AppDbContext())

{

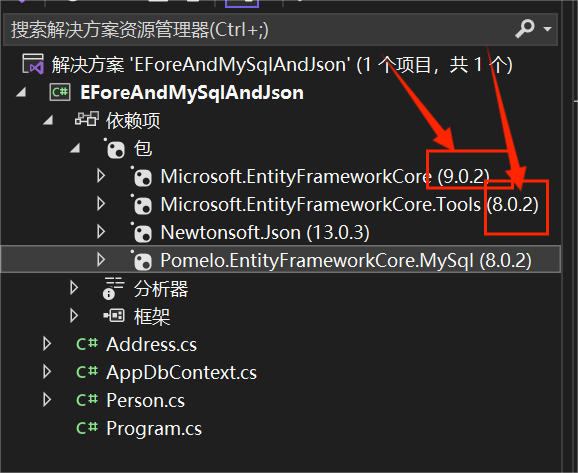
// 1.Migrate方法是将未应用的迁移文件，应用到数据库，生成表格。

// 2.它的功能与Update-Database指令是一样的。

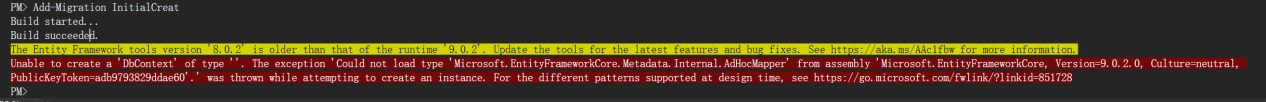
context.Database.Migrate();

}

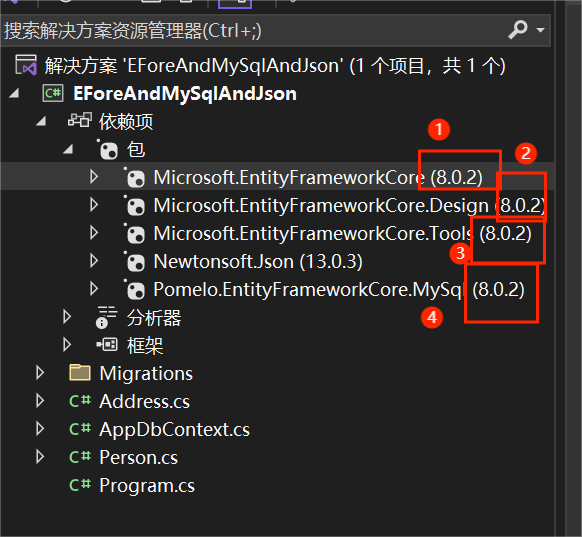
#### 失败原因1：包版本不对



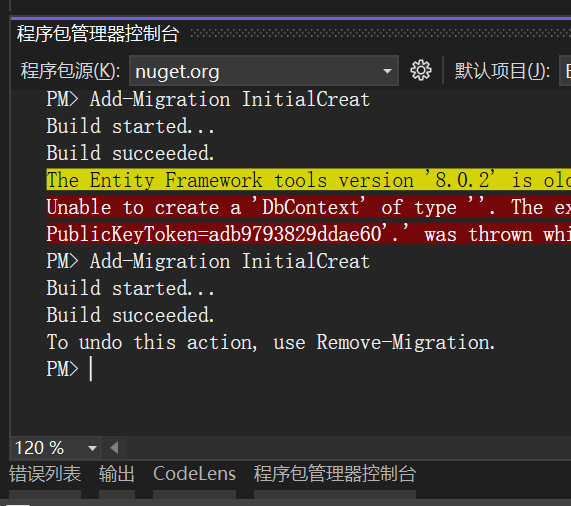
数据迁移报错，错误提示：



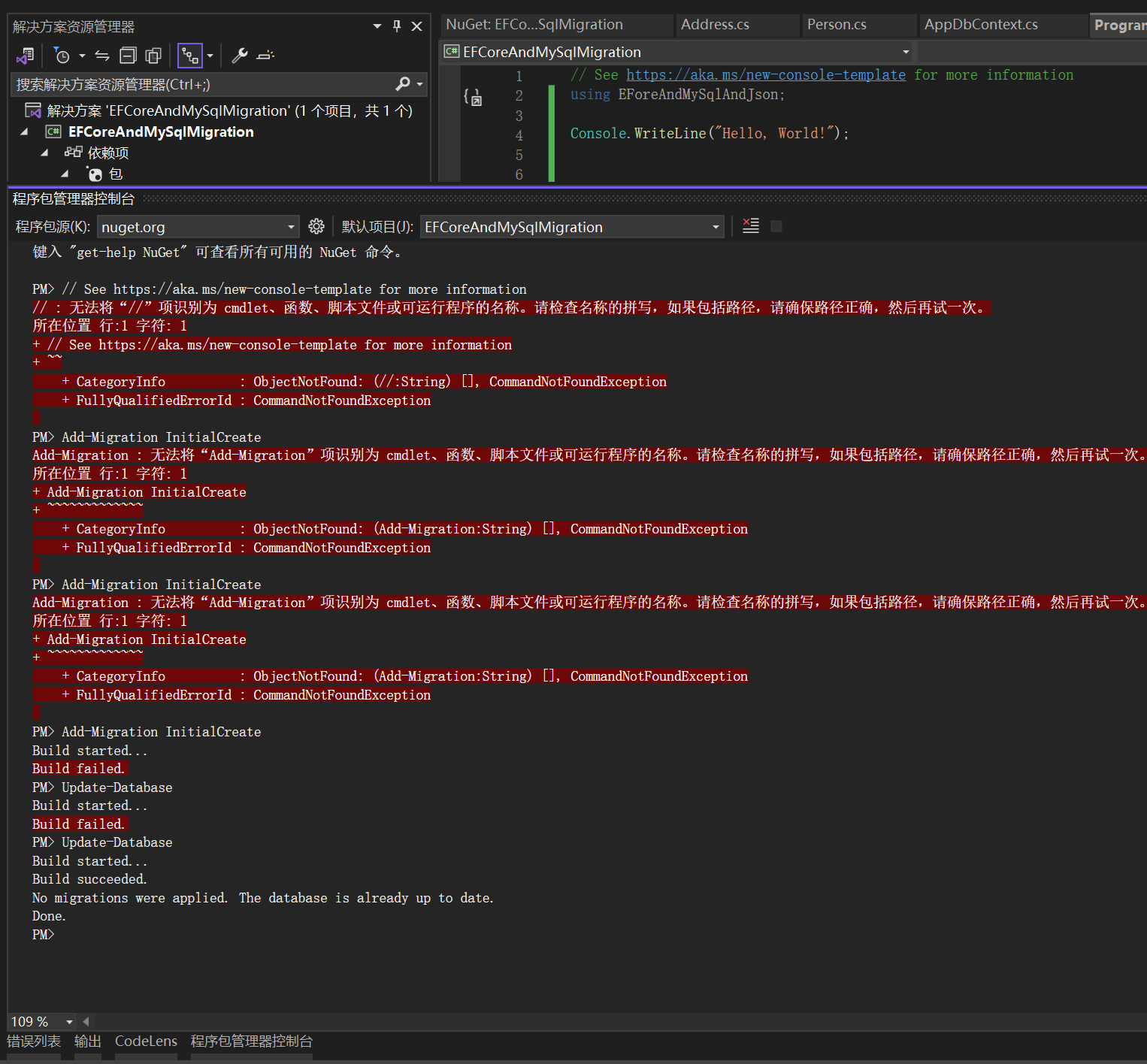
删除错误依赖包，统一4个依赖包版本：



第一遍：

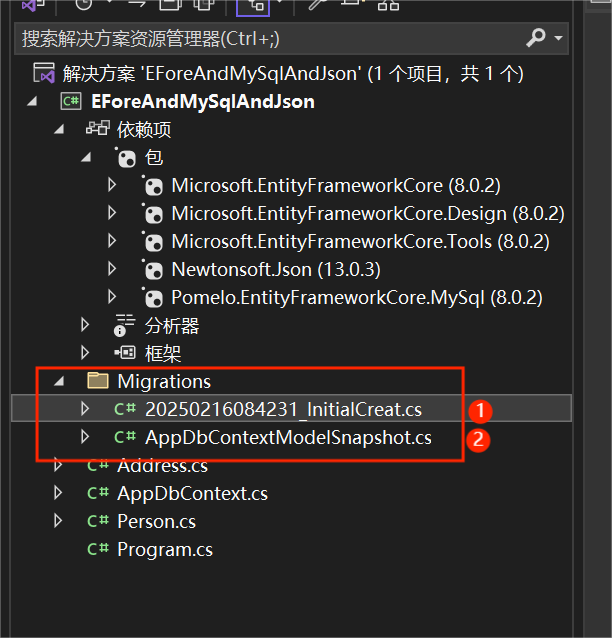


第二遍：



解决方案自动添加一个文件夹Migrations，里面新建了两个文件。

奇怪，第二遍测试，不显示这个文件夹。



#### 失败原因2：

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

#region 加载Json配置文件

/\*

\* 【bug：使用ConfigurationBuilder，会导致数据迁移失败；

\* 原因：可能是拿不到路径，因为没有运行；

\* 解决办法：先使用字符串，再使用ConfigurationBuilder】

\*/

//// 创建 ConfigurationBuilder 实例

//var configuration = new ConfigurationBuilder()

// // 设置配置文件的基础路径，通常是应用程序的基目录

// .SetBasePath(Directory.GetCurrentDirectory())

// // 添加 Appsettings.json 文件作为配置源，设置文件为必需,且支持自动重新加载

// .AddJsonFile("Appsettings.json", optional: false, reloadOnChange: true)

// //// 构建最终的 IConfiguration 对象

// .Build();

// 读取 DatabaseSettings 部分的配置信息

//string connectionString = configuration["ConnectionStrings:DefaultConnection"];

#endregion

//// 配置数据库连接字符串（为了安全，代码中避免明文）

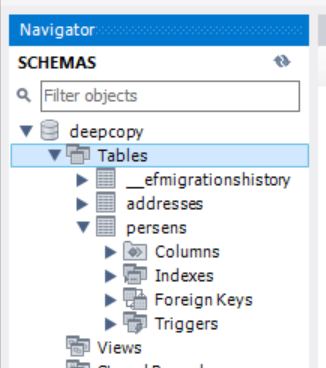
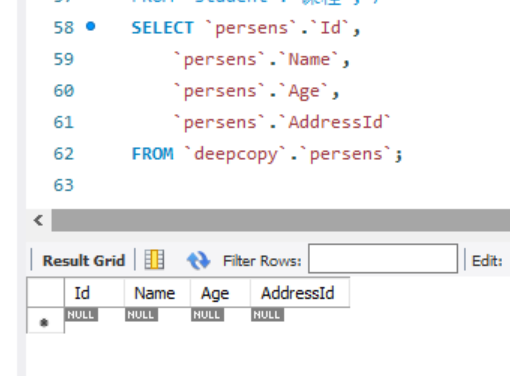
/// string localhost="127.0.0.1";

string connectionString = "server=localhost;port=3306;user=root;password=1;database=deepcopy;AllowPublicKeyRetrieval=true;";

base.OnConfiguring(optionsBuilder);

optionsBuilder.UseMySql(connectionString,ServerVersion.AutoDetect(connectionString));

#### 数据库查看结果：

 ---

### 5-1、在迁移过程中，如果数据库结构发生变化，如何更新迁移？

在使用 Entity Framework Core（EF Core）进行数据库迁移时，当数据库结构发生变化（例如添加、修改或删除实体类的属性，或者更改实体类之间的关系），需要更新迁移以反映这些变化。以下是详细的操作步骤和示例：

#### 1. 修改实体类

假设你已经有了一个简单的 Product 实体类和对应的 DbContext 类，现在需要对 Product 类进行修改，例如添加一个新的属性。

收起

csharp

// 原有的 Product 类

public class Product{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

}

// 修改后的 Product 类，添加了 Description 属性

public class Product{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

public string Description { get; set; }}

#### 2. 创建新的迁移

在包管理器控制台（PMC）或命令行中执行以下命令,创建一个新的迁移，该迁移将记录数据库结构的变化：

##### **使用包管理器控制台（适用于 Visual Studio 项目）**

Add-Migration AddDescriptionToProduct

这里的 AddDescriptionToProduct 是迁移的名称，你可以根据实际的更改内容自定义迁移名称，以便清晰地描述这次迁移所做的更改。

##### **使用命令行（适用于 .NET Core 项目）**

收起

plaintext

dotnet ef migrations add AddDescriptionToProduct

执行该命令后，EF Core 会分析当前实体类的状态与上一次迁移时的状态之间的差异，并生成一个新的迁移文件。这个文件通常位于项目的 Migrations 文件夹中，包含了用于更新数据库结构的 SQL 语句。

#### 3. 应用新的迁移到数据库

创建新的迁移文件后，需要将其应用到数据库中，以更新数据库结构。同样可以使用包管理器控制台或命令行来完成：

##### **使用包管理器控制台**

收起

plaintext

Update-Database

##### **使用命令行**

收起

plaintext

dotnet ef database update

执行这些命令后，EF Core 会将新的迁移文件中的 SQL 语句应用到数据库，从而更新数据库结构，使其与当前实体类的定义保持一致。

#### 4. 回滚迁移（可选）

如果在应用迁移后发现问题，或者需要将数据库结构恢复到之前的状态，可以使用回滚操作。可以指定回滚到某个特定的迁移版本：

##### **使用包管理器控制台**

收起

plaintext

Update-Database <PreviousMigrationName>

其中 <PreviousMigrationName> 是你要回滚到的迁移的名称。

##### **使用命令行**

收起

plaintext

dotnet ef database update <PreviousMigrationName>

#### 5. 删除迁移（可选）

如果新创建的迁移还未应用到数据库，并且你想要删除它，可以使用以下命令：

##### **使用包管理器控制台**

收起

plaintext

Remove-Migration

##### 使用命令行

收起

plaintext

dotnet ef migrations remove

##### **注意事项**

* 迁移文件是版本化的，每次创建新的迁移时，EF Core 会记录迁移的顺序，确保数据库结构的更新是按照正确的顺序进行的。
* 在进行迁移操作之前，建议备份数据库，以防意外情况导致数据丢失或损坏。
* 不同的数据库可能对迁移操作有一些细微的差异，需要根据实际使用的数据库进行相应的调整。

### **6. 深度复制对象并保存到数据库**

以下是完整的代码示例，展示了如何创建原始对象、进行深度复制，并将复制后的对象保存到 MySQL 数据库。

收起

csharp

using System;

class Program{

static void Main()

{

// 创建原始对象

Address originalAddress = new Address

{

Street = "123 Elm St",

City = "Springfield",

ZipCode = "12345"

};

Person originalPerson = new Person

{

Name = "Alice",

Age = 30,

Address = originalAddress

};

// 深度复制对象

Person copiedPerson = originalPerson.DeepCopy();

using (var context = new AppDbContext())

{

try

{

// 将复制后的对象添加到 DbContext

context.Persons.Add(copiedPerson);

// 保存更改到数据库

int rowsAffected = context.SaveChanges();

if (rowsAffected > 0)

{

Console.WriteLine("数据插入成功！");

}

else

{

Console.WriteLine("数据插入失败。");

}

}

catch (Exception ex)

{

Console.WriteLine($"发生错误: {ex.Message}");

}

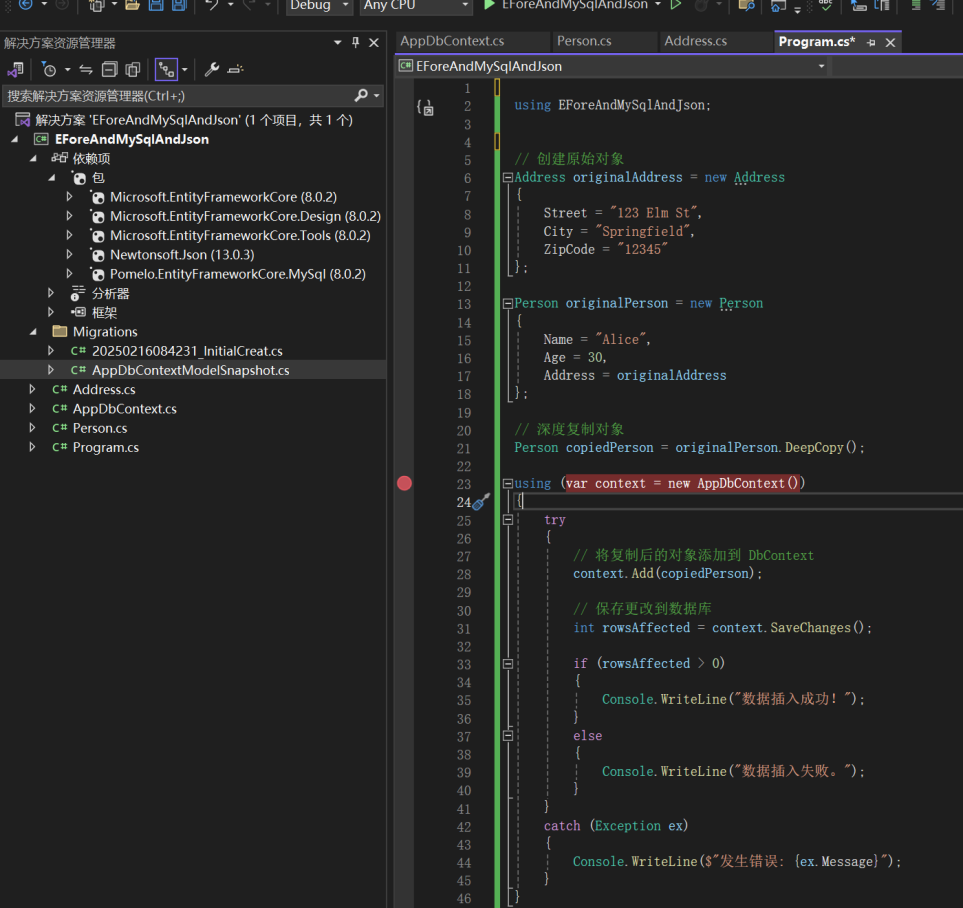
}

}}

**代码解释**

* ****实体类****：Person 和 Address 类定义了数据的结构，Person 类中的 Address 属性建立了与 Address 类的关联。
* **DbContext**配置****：AppDbContext 类继承自 DbContext，通过 OnConfiguring 方法配置 MySQL 数据库的连接字符串。
* ****深度复制****：Person 类中的 DeepCopy 方法使用 Newtonsoft.Json 库将对象序列化为 JSON 字符串，再反序列化为新对象，实现深度复制。
* ****数据插入****：创建 AppDbContext 实例，将复制后的 Person 对象添加到 Persons 集合中，调用 SaveChanges 方法将更改保存到数据库。

#### 6-1.在控制台实例化person和address，将数据写入数据库deepcopy的数据表Persons和Addresses。



### **7.创建本地配置文件**

在项目根目录下创建一个名为 appsettings.json 的文件，并添加以下内容：

{

"ConnectionStrings": {

"DefaultConnection": "server=localhost;user=root;password=your\_password;database=your\_database;AllowPublicKeyRetrieval=true;"

}}

请将 your\_password 替换为你的 MySQL 密码，your\_database 替换为你要使用的数据库名称。

注意事项

* 确保 MySQL 服务器正在运行，并且数据库用户具有创建表和插入数据的权限。
* 及时处理可能出现的异常，例如数据库连接失败、数据插入错误等，以保证程序的健壮性。
* 对于连接字符串中的 AllowPublicKeyRetrieval=true，在某些 MySQL 版本和配置下可能需要添加该参数以解决公钥检索问题。

#### 7-1.C#列举几个在项目根目录下创建一个名为 appsettings.json 的文件的方式

并添加以下内容：

{ "ConnectionStrings": { "DefaultConnection": "server=[localhost](https://localhost/" \t "https://www.doubao.com/chat/_blank);user=root;password=your\_password;database=your\_database;AllowPublicKeyRetrieval=true;" }}

分享

以下为你介绍几种在 C# 项目根目录下创建 appsettings.json 文件并添加指定内容的方式：

##### 方式一：手动创建

这是最为直接的方法，适用于你想要快速完成文件创建和内容添加的场景。

###### 步骤

1. 在 Visual Studio 或其他代码编辑器中打开你的项目。
2. 找到项目根目录（通常包含 .csproj 文件的目录）。
3. 在项目根目录下，右键点击空白处，选择 “添加” -> “新建项”。
4. 在弹出的 “添加新项” 对话框中，选择 “JSON 文件”，将文件名命名为 appsettings.json，然后点击 “添加”。
5. 在新创建的 appsettings.json 文件中，添加以下内容：

收起

json

{

"ConnectionStrings": {

"DefaultConnection": "server=localhost;user=root;password=your\_password;database=your\_database;AllowPublicKeyRetrieval=true;"

}}

将 your\_password 替换为你的 MySQL 数据库密码，your\_database 替换为你要使用的数据库名称。

##### **方式二：使用代码创建**

如果你希望通过代码自动创建并添加内容，可以使用以下示例代码。此方式适合在项目初始化脚本或自动化部署流程中使用。

using System.IO;

class Program{

static void Main()

{

// 定义 appsettings.json 文件的内容

string jsonContent = @"{

""ConnectionStrings"": {

""DefaultConnection"": ""server=localhost;user=root;password=your\_password;database=your\_database;AllowPublicKeyRetrieval=true;""

}

}";

// 获取项目根目录

string projectRoot = Directory.GetCurrentDirectory();

// 拼接 appsettings.json 文件的完整路径

string filePath = Path.Combine(projectRoot, "appsettings.json");

// 将内容写入文件

File.WriteAllText(filePath, jsonContent);

System.Console.WriteLine("appsettings.json 文件已创建并添加内容。");

}}

代码解释

1. ****定义 JSON 内容****：使用字符串变量 jsonContent 存储 appsettings.json 文件的内容，注意需要对双引号进行转义。
2. ****获取项目根目录****：使用 Directory.GetCurrentDirectory() 方法获取当前项目的根目录。
3. ****拼接文件路径****：使用 Path.Combine 方法将项目根目录和文件名拼接成完整的文件路径。
4. ****写入文件****：使用 File.WriteAllText 方法将 JSON 内容写入到 appsettings.json 文件中。

##### **方式三：使用模板引擎（如 Razor）**

如果你需要根据不同的环境动态生成 appsettings.json 文件的内容，可以使用 Razor 模板引擎。这种方式适合在复杂的配置管理场景中使用。

###### **步骤**

1. 安装 Microsoft.AspNetCore.Razor.Runtime NuGet 包。
2. 创建一个 Razor 模板文件，例如 appsettings.template.cshtml，内容如下：

收起

razor

{

"ConnectionStrings": {

"DefaultConnection": "server=localhost;user=root;password=@Model.Password;database=@Model.Database;AllowPublicKeyRetrieval=true;"

}

}

1. 使用以下代码生成 appsettings.json 文件：

收起

csharp

using Microsoft.AspNetCore.Razor.Language;using System.IO;

class Program{

class AppSettingsModel

{

public string Password { get; set; }

public string Database { get; set; }

}

static void Main()

{

// 定义模型数据

var model = new AppSettingsModel

{

Password = "your\_password",

Database = "your\_database"

};

// 读取 Razor 模板文件

string template = File.ReadAllText("appsettings.template.cshtml");

// 配置 Razor 引擎

var engine = RazorProjectEngine.Create(RazorConfiguration.Default);

// 解析模板

var document = engine.Process("appsettings.template.cshtml", template);

// 生成 C# 代码

var csharpDocument = document.GetCSharpDocument();

// 动态编译并执行模板

var result = csharpDocument.GeneratedCode

.Replace("@Model.Password", model.Password)

.Replace("@Model.Database", model.Database);

// 获取项目根目录

string projectRoot = Directory.GetCurrentDirectory();

// 拼接 appsettings.json 文件的完整路径

string filePath = Path.Combine(projectRoot, "appsettings.json");

// 将内容写入文件

File.WriteAllText(filePath, result);

System.Console.WriteLine("appsettings.json 文件已创建并添加内容。");

}}

代码解释

1. ****定义模型类****：创建 AppSettingsModel 类，用于存储数据库密码和数据库名称。
2. ****读取 Razor 模板文件****：使用 File.ReadAllText 方法读取 appsettings.template.cshtml 文件的内容。
3. ****配置 Razor 引擎****：使用 RazorProjectEngine.Create 方法创建 Razor 引擎。
4. ****解析模板并生成 C# 代码****：使用 engine.Process 方法解析模板，然后使用 document.GetCSharpDocument 方法生成 C# 代码。
5. ****动态替换模型数据****：将生成的 C# 代码中的 @Model.Password 和 @Model.Database 替换为实际的密码和数据库名称。
6. ****写入文件****：将替换后的内容写入到 appsettings.json 文件中。

#### 7-2.在 C# 中，将 appsettings.json 配置文件加载到数据库初始化代码中有多种常见的方式

以下为你详细介绍：

##### **方式一：使用 Microsoft.Extensions.Configuration 进行加载**

这是在 .NET Core 及 .NET 5+ 项目中最常用的方式，Microsoft.Extensions.Configuration 提供了强大的配置管理功能。

###### **步骤**

1. ****安装 NuGet 包****：确保项目中安装了 Microsoft.Extensions.Configuration.Json 包，你可以在包管理器控制台中执行以下命令进行安装：

收起

plaintext

Install-Package Microsoft.Extensions.Configuration.Json

1. ****编写代码加载配置文件并初始化数据库连接****：

收起

csharp

using Microsoft.Extensions.Configuration;using System;using System.IO;using MySqlConnector; // 假设使用 MySQL 数据库

class Program{

static void Main()

{

// 创建配置构建器

var configuration = new ConfigurationBuilder()

.SetBasePath(Directory.GetCurrentDirectory())

.AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)

.Build();

// 获取数据库连接字符串

string connectionString = configuration.GetConnectionString("DefaultConnection");

try

{

// 创建数据库连接

using (MySqlConnection connection = new MySqlConnection(connectionString))

{

// 打开数据库连接

connection.Open();

Console.WriteLine("数据库连接已打开。");

// 这里可以执行数据库操作

}

}

catch (Exception ex)

{

Console.WriteLine($"数据库连接出错: {ex.Message}");

}

}}

###### **代码解释**

* ConfigurationBuilder：用于构建配置对象，通过 SetBasePath 方法设置配置文件的基础路径，AddJsonFile 方法添加 appsettings.json 文件。
* GetConnectionString：从配置中获取名为 DefaultConnection 的数据库连接字符串。
* MySqlConnection：使用获取到的连接字符串创建 MySQL 数据库连接，并打开连接。

##### **方式二：使用 JsonSerializer 手动解析（较基础的方式）**

这种方式不依赖于 Microsoft.Extensions.Configuration，直接使用 System.Text.Json 中的 JsonSerializer 手动解析 appsettings.json 文件。

#### **代码示例**

收起

csharp

using System;using System.IO;using System.Text.Json;using MySqlConnector;

class Program{

class AppSettings

{

public ConnectionStringsSection ConnectionStrings { get; set; }

}

class ConnectionStringsSection

{

public string DefaultConnection { get; set; }

}

static void Main()

{

try

{

// 读取 appsettings.json 文件内容

string json = File.ReadAllText("appsettings.json");

// 反序列化 JSON 内容到 AppSettings 对象

AppSettings settings = JsonSerializer.Deserialize<AppSettings>(json);

// 获取数据库连接字符串

string connectionString = settings.ConnectionStrings.DefaultConnection;

// 创建数据库连接

using (MySqlConnection connection = new MySqlConnection(connectionString))

{

// 打开数据库连接

connection.Open();

Console.WriteLine("数据库连接已打开。");

// 这里可以执行数据库操作

}

}

catch (Exception ex)

{

Console.WriteLine($"出错: {ex.Message}");

}

}}

###### **代码解释**

* 定义 AppSettings 和 ConnectionStringsSection 类来表示 appsettings.json 文件的结构。
* 使用 File.ReadAllText 方法读取 appsettings.json 文件的内容。
* 使用 JsonSerializer.Deserialize 方法将 JSON 内容反序列化为 AppSettings 对象。
* 从反序列化后的对象中获取数据库连接字符串，并创建和打开数据库连接。

##### **方式三：使用依赖注入（适用于 [ASP.NET](https://asp.net/" \t "https://www.doubao.com/chat/_blank) Core 项目）**

在 [ASP.NET](https://asp.net/" \t "https://www.doubao.com/chat/_blank) Core 项目中，可以利用依赖注入机制将配置对象注入到需要使用的服务中。

###### **步骤**

1. ****配置依赖注入****：在 Startup.cs（.NET Core 3.1 及以前）或 Program.cs（.NET 5+）中进行配置。

收起

csharp

// Program.cs（.NET 6+ 示例）using Microsoft.AspNetCore.Hosting;using Microsoft.Extensions.Configuration;using Microsoft.Extensions.DependencyInjection;using Microsoft.Extensions.Hosting;using MySqlConnector;

var builder = WebApplication.CreateBuilder(args);

// 加载配置文件var configuration = builder.Configuration;

// 获取数据库连接字符串string connectionString = configuration.GetConnectionString("DefaultConnection");

// 注册数据库连接服务

builder.Services.AddScoped<MySqlConnection>(\_ => new MySqlConnection(connectionString));

var app = builder.Build();

// 示例：从服务容器中获取数据库连接using (var scope = app.Services.CreateScope()){

var connection = scope.ServiceProvider.GetRequiredService<MySqlConnection>();

try

{

connection.Open();

Console.WriteLine("数据库连接已打开。");

}

catch (Exception ex)

{

Console.WriteLine($"数据库连接出错: {ex.Message}");

}}

app.Run();

###### **代码解释**

* 使用 WebApplication.CreateBuilder 创建应用程序构建器，并通过 builder.Configuration 加载配置文件。
* 使用 GetConnectionString 方法获取数据库连接字符串。
* 使用 AddScoped 方法将 MySqlConnection 服务注册到依赖注入容器中。
* 在应用程序启动时，从服务容器中获取 MySqlConnection 实例并打开连接。

## 2、C#如何使用迁移创建SQL server数据表，详细步骤

在 C# 中使用 Entity Framework Core（EF Core）的迁移功能来创建数据表是一种常见且高效的方式。下面以一个简单的控制台应用程序为例，详细介绍使用迁移创建数据表的步骤。

### **1. 创建项目**

首先，打开 Visual Studio，创建一个新的 C# 控制台应用程序项目。

### **2. 安装必要的 NuGet 包**

EF Core 支持多种数据库，这里以 SQL Server 为例，需要安装以下 NuGet 包：

* Microsoft.EntityFrameworkCore.SqlServer：用于与 SQL Server 数据库交互的 EF Core 提供程序。
* Microsoft.EntityFrameworkCore.Tools：提供用于执行迁移命令的工具。

可以通过 NuGet 包管理器控制台执行以下命令进行安装：

收起

plaintext

Install-Package Microsoft.EntityFrameworkCore.SqlServer

Install-Package Microsoft.EntityFrameworkCore.Tools

### **3. 定义实体类**

实体类代表数据库中的表结构，以下是一个简单的 Product 实体类示例：

收起

csharp

public class Product{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }}

### **4. 创建 DbContext 类**

DbContext 类是 EF Core 与数据库交互的核心，它负责管理实体集合和数据库连接。

收起

csharp

using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : DbContext{

public DbSet<Product> Products { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

// 配置数据库连接字符串，根据实际情况修改

optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=YourDatabase;Trusted\_Connection=True;");

}}

### **5. 启用迁移功能**

在包管理器控制台中，确保项目上下文选择正确（即包含 DbContext 类的项目），然后执行以下命令启用迁移功能：

收起

plaintext

Add-Migration InitialCreate

* Add-Migration 是 EF Core 提供的命令，用于创建一个新的迁移。
* InitialCreate 是迁移的名称，你可以根据实际情况自定义。

执行该命令后，EF Core 会根据定义的实体类和 DbContext 生成一个迁移文件，该文件记录了数据库结构的初始状态。迁移文件通常位于项目的 Migrations 文件夹中，包含两个文件：一个是迁移的主文件，另一个是快照文件。

### **6. 应用迁移到数据库**

迁移文件生成后，需要将其应用到数据库中，以创建相应的数据表。在包管理器控制台中执行以下命令：

收起

plaintext

Update-Database

* Update-Database 命令会将最新的迁移文件应用到数据库，创建或更新数据库结构。

### **7. 验证数据表创建**

可以使用 SQL Server Management Studio（SSMS）或其他数据库管理工具连接到指定的数据库，查看是否成功创建了 Products 表。

### **完整代码示例**

收起

csharp

using Microsoft.EntityFrameworkCore;using System;

// 定义实体类public class Product{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }}

// 定义 DbContext 类public class ApplicationDbContext : DbContext{

public DbSet<Product> Products { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

// 配置数据库连接字符串，根据实际情况修改

optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=YourDatabase;Trusted\_Connection=True;");

}}

class Program{

static void Main()

{

using (var context = new ApplicationDbContext())

{

try

{

// 确保数据库存在并应用最新迁移

context.Database.Migrate();

Console.WriteLine("数据库迁移成功！");

}

catch (Exception ex)

{

Console.WriteLine($"迁移过程中出现错误: {ex.Message}");

}

}

}}

### **注意事项**

* 迁移文件是版本化的，每次对实体类或 DbContext 进行更改后，都需要创建新的迁移并应用到数据库。
* 不同的数据库需要使用不同的 EF Core 提供程序，例如使用 MySQL 数据库需要安装 Pomelo.EntityFrameworkCore.MySql 包，并在 OnConfiguring 方法中使用 UseMySql 配置连接字符串。

## 5.C#截取字符串最后几个字符？

分享

在 C# 中，有多种方法可以截取字符串的最后几个字符，下面为你详细介绍这些方法。

### **方法一：使用 Substring 方法**

Substring 是 String 类提供的一个常用方法，它可以从指定位置开始截取字符串的一部分。若要截取字符串的最后几个字符，可结合字符串的长度来计算起始位置。

****示例代码****：

收起

csharp

using System;

class Program{

static void Main()

{

string originalString = "HelloWorld";

int lastCharCount = 5;

if (originalString.Length >= lastCharCount)

{

string lastChars = originalString.Substring(originalString.Length - lastCharCount);

Console.WriteLine(lastChars);

}

else

{

Console.WriteLine("字符串长度小于要截取的字符数。");

}

}}

****代码解释****：

* 首先定义了一个原始字符串 originalString 和要截取的最后字符数量 lastCharCount。
* 通过 originalString.Length - lastCharCount 计算出截取的起始位置，调用 Substring 方法进行截取。
* 为避免出现索引越界异常，添加了判断条件 originalString.Length >= lastCharCount。

### **方法二：使用 Span<T>（适用于 .NET Core 2.1 及以上版本）**

Span<T> 是一种轻量级、高性能的类型，用于表示连续的内存区域。可以将字符串转换为 ReadOnlySpan<char> 类型，然后进行切片操作。

****示例代码****：

收起

csharp

using System;

class Program{

static void Main()

{

string originalString = "HelloWorld";

int lastCharCount = 5;

if (originalString.Length >= lastCharCount)

{

ReadOnlySpan<char> span = originalString.AsSpan();

ReadOnlySpan<char> lastCharsSpan = span.Slice(span.Length - lastCharCount);

string lastChars = lastCharsSpan.ToString();

Console.WriteLine(lastChars);

}

else

{

Console.WriteLine("字符串长度小于要截取的字符数。");

}

}}

****代码解释****：

* 先将字符串 originalString 转换为 ReadOnlySpan<char> 类型的 span。
* 使用 Slice 方法从指定位置开始切片，得到最后几个字符的 ReadOnlySpan<char>。
* 最后将切片结果转换为字符串类型输出。

### **方法三：使用 LINQ 的 TakeLast 方法**

LINQ（Language Integrated Query）提供了许多方便的方法来处理集合，TakeLast 方法可以从集合的末尾开始选取指定数量的元素。

****示例代码****：

收起

csharp

using System;using System.Linq;

class Program{

static void Main()

{

string originalString = "HelloWorld";

int lastCharCount = 5;

if (originalString.Length >= lastCharCount)

{

string lastChars = new string(originalString.TakeLast(lastCharCount).ToArray());

Console.WriteLine(lastChars);

}

else

{

Console.WriteLine("字符串长度小于要截取的字符数。");

}

}}

****代码解释****：

* 调用 TakeLast 方法从字符串末尾选取指定数量的字符。
* 使用 ToArray 方法将选取的字符转换为字符数组。
* 通过 new string 构造函数将字符数组转换为字符串。

以上三种方法都可以实现截取字符串最后几个字符的功能，你可以根据具体需求和使用场景选择合适的方法。

## 6.C#比较字符串数组里面的元素是否完全等于另一个字符串数组的字符串

在 C# 里要比较两个字符串数组的元素是否完全相等，需要从多个维度去考量，包括元素内容、元素顺序以及数组长度等。下面为你详细介绍几种不同的比较方式及对应的代码示例。

### **1. 考虑元素顺序和内容的完全相等**

当要求两个数组不仅元素内容相同，而且元素的顺序也必须一致时，可以使用以下方法。

#### **使用 SequenceEqual 方法（推荐）**

SequenceEqual 是 System.Linq 命名空间下的扩展方法，它会逐个比较两个序列中的元素，只有当两个序列长度相同且对应位置的元素都相等时，才会返回 true。

收起

csharp

using System;using System.Linq;

class Program{

static void Main()

{

string[] array1 = { "apple", "banana", "cherry" };

string[] array2 = { "apple", "banana", "cherry" };

string[] array3 = { "banana", "apple", "cherry" };

bool isEqual12 = array1.SequenceEqual(array2);

bool isEqual13 = array1.SequenceEqual(array3);

Console.WriteLine($"array1 和 array2 是否完全相等: {isEqual12}");

Console.WriteLine($"array1 和 array3 是否完全相等: {isEqual13}");

}}

#### **手动遍历数组比较**

手动编写代码，通过循环逐个比较数组中的元素。

收起

csharp

using System;

class Program{

static bool AreArraysEqual(string[] arr1, string[] arr2)

{

if (arr1.Length != arr2.Length)

{

return false;

}

for (int i = 0; i < arr1.Length; i++)

{

if (arr1[i] != arr2[i])

{

return false;

}

}

return true;

}

static void Main()

{

string[] array1 = { "apple", "banana", "cherry" };

string[] array2 = { "apple", "banana", "cherry" };

string[] array3 = { "banana", "apple", "cherry" };

bool isEqual12 = AreArraysEqual(array1, array2);

bool isEqual13 = AreArraysEqual(array1, array3);

Console.WriteLine($"array1 和 array2 是否完全相等: {isEqual12}");

Console.WriteLine($"array1 和 array3 是否完全相等: {isEqual13}");

}}

### **2. 不考虑元素顺序，仅考虑元素内容相等**

若只关注两个数组中是否包含相同的元素，而不考虑元素的排列顺序，可以采用以下方法。

#### **使用 HashSet<T> 进行比较**

HashSet<T> 是一种无序集合，它可以高效地判断元素是否存在。将两个数组的元素分别添加到 HashSet<T> 中，然后比较两个集合是否相等。

收起

csharp

using System;using System.Collections.Generic;using System.Linq;

class Program{

static bool AreArraysEqualIgnoringOrder(string[] arr1, string[] arr2)

{

if (arr1.Length != arr2.Length)

{

return false;

}

HashSet<string> set1 = new HashSet<string>(arr1);

HashSet<string> set2 = new HashSet<string>(arr2);

return set1.SetEquals(set2);

}

static void Main()

{

string[] array1 = { "apple", "banana", "cherry" };

string[] array2 = { "cherry", "apple", "banana" };

string[] array3 = { "banana", "apple", "date" };

bool isEqual12 = AreArraysEqualIgnoringOrder(array1, array2);

bool isEqual13 = AreArraysEqualIgnoringOrder(array1, array3);

Console.WriteLine($"array1 和 array2 是否内容相等（不考虑顺序）: {isEqual12}");

Console.WriteLine($"array1 和 array3 是否内容相等（不考虑顺序）: {isEqual13}");

}}

#### **使用 GroupBy 方法统计元素数量进行比较**

通过 GroupBy 方法对数组中的元素进行分组统计，然后比较两个数组中每个元素的出现次数是否相同。

收起

csharp

using System;using System.Linq;

class Program{

static bool AreArraysEqualIgnoringOrder(string[] arr1, string[] arr2)

{

if (arr1.Length != arr2.Length)

{

return false;

}

var grouped1 = arr1.GroupBy(x => x).ToDictionary(g => g.Key, g => g.Count());

var grouped2 = arr2.GroupBy(x => x).ToDictionary(g => g.Key, g => g.Count());

return grouped1.Count == grouped2.Count &&!grouped1.Except(grouped2).Any();

}

static void Main()

{

string[] array1 = { "apple", "banana", "cherry" };

string[] array2 = { "cherry", "apple", "banana" };

string[] array3 = { "banana", "apple", "date" };

bool isEqual12 = AreArraysEqualIgnoringOrder(array1, array2);

bool isEqual13 = AreArraysEqualIgnoringOrder(array1, array3);

Console.WriteLine($"array1 和 array2 是否内容相等（不考虑顺序）: {isEqual12}");

Console.WriteLine($"array1 和 array3 是否内容相等（不考虑顺序）: {isEqual13}");

}}

## C#将json格式数据保存到excel表格

分享

在 C# 中要将 JSON 格式的数据保存到 Excel 表格，可以借助第三方库，这里以 Newtonsoft.Json 用于解析 JSON 数据，ClosedXML 用于创建和操作 Excel 文件为例。以下是实现该功能的详细步骤和示例代码：

### **1. 安装必要的 NuGet 包**

首先，你需要在项目中安装 Newtonsoft.Json 和 ClosedXML 这两个 NuGet 包。可以通过 Visual Studio 的 NuGet 包管理器或者使用 .NET CLI 进行安装：

收起

bash

dotnet add package Newtonsoft.Json

dotnet add package ClosedXML

### **2. 示例代码**

以下是一个完整的示例，展示了如何将 JSON 数据保存到 Excel 表格：

收起

csharp

using Newtonsoft.Json;using Newtonsoft.Json.Linq;using ClosedXML.Excel;using System.IO;

class Program{

static void Main()

{

// 示例 JSON 数据

string jsonData = @"[

{

""Name"": ""John"",

""Age"": 30,

""City"": ""New York""

},

{

""Name"": ""Jane"",

""Age"": 25,

""City"": ""Los Angeles""

}

]";

try

{

// 解析 JSON 数据

var jsonArray = JArray.Parse(jsonData);

// 创建一个新的 Excel 工作簿

using (var workbook = new XLWorkbook())

{

// 添加一个工作表

var worksheet = workbook.Worksheets.Add("Data");

// 获取 JSON 对象的属性名称作为表头

if (jsonArray.Count > 0)

{

var firstObject = jsonArray[0] as JObject;

int columnIndex = 1;

foreach (var property in firstObject.Properties())

{

worksheet.Cell(1, columnIndex).Value = property.Name;

columnIndex++;

}

// 填充数据行

int rowIndex = 2;

foreach (var item in jsonArray)

{

var obj = item as JObject;

columnIndex = 1;

foreach (var property in obj.Properties())

{

worksheet.Cell(rowIndex, columnIndex).Value = property.Value.ToString();

columnIndex++;

}

rowIndex++;

}

}

// 保存 Excel 文件

using (var stream = new FileStream("output.xlsx", FileMode.Create, FileAccess.Write))

{

workbook.SaveAs(stream);

}

Console.WriteLine("数据已成功保存到 output.xlsx 文件中。");

}

}

catch (JsonException ex)

{

Console.WriteLine($"JSON 解析错误: {ex.Message}");

}

catch (IOException ex)

{

Console.WriteLine($"文件操作错误: {ex.Message}");

}

}}

### **3. 代码解释**

* ****JSON 解析****：使用 Newtonsoft.Json 的 JArray.Parse 方法将 JSON 字符串解析为 JArray 对象，方便后续处理。
* ****创建 Excel 工作簿和工作表****：使用 ClosedXML 的 XLWorkbook 类创建一个新的 Excel 工作簿，并添加一个名为 "Data" 的工作表。
* ****设置表头****：从第一个 JSON 对象中获取属性名称，并将其作为 Excel 表格的表头。
* ****填充数据****：遍历 JSON 数组中的每个对象，将对象的属性值填充到 Excel 表格的相应单元格中。
* ****保存 Excel 文件****：使用 FileStream 将生成的 Excel 工作簿保存到本地文件 output.xlsx 中。

### **4. 注意事项**

* 确保 JSON 数据的结构是一致的，即每个对象具有相同的属性名称，否则可能会导致数据填充错误。
* 处理可能出现的异常，如 JSON 解析错误和文件操作错误，以保证程序的健壮性。

通过以上步骤，你就可以将 JSON 格式的数据保存到 Excel 表格中。

分享

## 除了 ClosedXML，还有哪些库可以将 JSON 数据保存到 Excel 表格？

除了 ClosedXML 之外，在 C# 里还有不少其他库可用于把 JSON 数据保存到 Excel 表格，下面为你详细介绍：

### **1. EPPlus**

* ****简介****：EPPlus 是一个开源的 .NET 库，用于创建、读取和修改 Excel 文件（.xlsx 格式），它不需要安装 Microsoft Office 或 Excel 组件，可在服务器端使用。
* ****使用示例****：

收起

csharp

using OfficeOpenXml;using Newtonsoft.Json;using Newtonsoft.Json.Linq;using System.IO;

class Program{

static void Main()

{

string jsonData = @"[

{

""Name"": ""Alice"",

""Age"": 28,

""City"": ""Chicago""

},

{

""Name"": ""Bob"",

""Age"": 32,

""City"": ""Houston""

}

]";

var jsonArray = JArray.Parse(jsonData);

using (var package = new ExcelPackage())

{

var worksheet = package.Workbook.Worksheets.Add("Data");

if (jsonArray.Count > 0)

{

var firstObject = jsonArray[0] as JObject;

int colIndex = 1;

foreach (var property in firstObject.Properties())

{

worksheet.Cells[1, colIndex].Value = property.Name;

colIndex++;

}

int rowIndex = 2;

foreach (var item in jsonArray)

{

var obj = item as JObject;

colIndex = 1;

foreach (var property in obj.Properties())

{

worksheet.Cells[rowIndex, colIndex].Value = property.Value.ToString();

colIndex++;

}

rowIndex++;

}

}

var file = new FileInfo("output\_epplus.xlsx");

package.SaveAs(file);

}

}}

### **2. NPOI**

* ****简介****：NPOI 是一个开源的 .NET 库，能在没有安装 Microsoft Office 的情况下读写 Office 格式文件，支持 .xls（Excel 97 - 2003）和 .xlsx 格式。
* ****使用示例****：

收起

csharp

using NPOI.SS.UserModel;using NPOI.XSSF.UserModel;using Newtonsoft.Json;using Newtonsoft.Json.Linq;using System.IO;

class Program{

static void Main()

{

string jsonData = @"[

{

""Name"": ""Charlie"",

""Age"": 35,

""City"": ""Miami""

},

{

""Name"": ""David"",

""Age"": 22,

""City"": ""Seattle""

}

]";

var jsonArray = JArray.Parse(jsonData);

IWorkbook workbook = new XSSFWorkbook();

ISheet sheet = workbook.CreateSheet("Data");

if (jsonArray.Count > 0)

{

var firstObject = jsonArray[0] as JObject;

IRow headerRow = sheet.CreateRow(0);

int colIndex = 0;

foreach (var property in firstObject.Properties())

{

headerRow.CreateCell(colIndex).SetCellValue(property.Name);

colIndex++;

}

int rowIndex = 1;

foreach (var item in jsonArray)

{

var obj = item as JObject;

IRow dataRow = sheet.CreateRow(rowIndex);

colIndex = 0;

foreach (var property in obj.Properties())

{

dataRow.CreateCell(colIndex).SetCellValue(property.Value.ToString());

colIndex++;

}

rowIndex++;

}

}

using (var file = new FileStream("output\_npoi.xlsx", FileMode.Create, FileAccess.Write))

{

workbook.Write(file);

}

}}

### **3. Microsoft.Office.Interop.Excel**

* ****简介****：这是 Microsoft 提供的 COM 组件，允许 .NET 程序与 Excel 应用程序进行交互。不过使用该组件需要在运行环境中安装 Microsoft Excel。
* ****使用示例****：

收起

csharp

using Microsoft.Office.Interop.Excel;using Newtonsoft.Json;using Newtonsoft.Json.Linq;using System.Runtime.InteropServices;

class Program{

static void Main()

{

string jsonData = @"[

{

""Name"": ""Eve"",

""Age"": 26,

""City"": ""Dallas""

},

{

""Name"": ""Frank"",

""Age"": 38,

""City"": ""Phoenix""

}

]";

var jsonArray = JArray.Parse(jsonData);

Application excelApp = new Application();

Workbook workbook = excelApp.Workbooks.Add();

Worksheet worksheet = workbook.ActiveSheet;

if (jsonArray.Count > 0)

{

var firstObject = jsonArray[0] as JObject;

int colIndex = 1;

foreach (var property in firstObject.Properties())

{

worksheet.Cells[1, colIndex] = property.Name;

colIndex++;

}

int rowIndex = 2;

foreach (var item in jsonArray)

{

var obj = item as JObject;

colIndex = 1;

foreach (var property in obj.Properties())

{

worksheet.Cells[rowIndex, colIndex] = property.Value.ToString();

colIndex++;

}

rowIndex++;

}

}

workbook.SaveAs("output\_interop.xlsx");

workbook.Close();

excelApp.Quit();

Marshal.ReleaseComObject(worksheet);

Marshal.ReleaseComObject(workbook);

Marshal.ReleaseComObject(excelApp);

}}

## C#添加json数据到datagridview表格和listtree控件

在 C# 里要快速将 JSON 数据添加到控件中，需要依据具体使用的控件类型来操作，常见的控件有 DataGridView、ListView 等。下面分别以这两种控件为例，介绍如何把 JSON 数据添加到控件中。

### **准备工作**

首先要安装 Newtonsoft.Json 库来处理 JSON 数据，可以使用 NuGet 包管理器进行安装：

收起

bash

dotnet add package Newtonsoft.Json

### **1. 将 JSON 数据添加到 DataGridView**

DataGridView 是一个用于显示和编辑表格数据的控件，下面的示例展示了如何将 JSON 数据添加到 DataGridView 中：

using System;using System.Data;using System.Windows.Forms;using Newtonsoft.Json;using Newtonsoft.Json.Linq;

namespace JsonToDataGridView{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void Form1\_Load(object sender, EventArgs e)

{

string jsonData = @"[

{

""Name"": ""John"",

""Age"": 30,

""City"": ""New York""

},

{

""Name"": ""Jane"",

""Age"": 25,

""City"": ""Los Angeles""

}

]";

// 解析 JSON 数据

var jsonArray = JArray.Parse(jsonData);

// 创建 DataTable

DataTable dataTable = new DataTable();

if (jsonArray.Count > 0)

{

var firstObject = jsonArray[0] as JObject;

foreach (var property in firstObject.Properties())

{

dataTable.Columns.Add(property.Name);

}

foreach (var item in jsonArray)

{

var obj = item as JObject;

DataRow row = dataTable.NewRow();

foreach (var property in obj.Properties())

{

row[property.Name] = property.Value.ToString();

}

dataTable.Rows.Add(row);

}

}

// 将 DataTable 绑定到 DataGridView

dataGridView1.DataSource = dataTable;

}

}}

****代码解释****：

* 先定义了一个 JSON 字符串 jsonData。
* 利用 JArray.Parse 方法解析 JSON 数据。
* 创建 DataTable 并根据 JSON 对象的属性名添加列。
* 遍历 JSON 数组，将每个对象的属性值添加到 DataTable 的行中。
* 最后把 DataTable 绑定到 DataGridView 上。

### **2. 将 JSON 数据添加到 ListView**

ListView 可用于显示项目列表，以下示例展示了如何将 JSON 数据添加到 ListView 中：

收起

csharp

using System;using System.Windows.Forms;using Newtonsoft.Json;using Newtonsoft.Json.Linq;

namespace JsonToListView{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void Form1\_Load(object sender, EventArgs e)

{

string jsonData = @"[

{

""Name"": ""John"",

""Age"": 30,

""City"": ""New York""

},

{

""Name"": ""Jane"",

""Age"": 25,

""City"": ""Los Angeles""

}

]";

// 解析 JSON 数据

var jsonArray = JArray.Parse(jsonData);

// 添加列

listView1.View = View.Details;

if (jsonArray.Count > 0)

{

var firstObject = jsonArray[0] as JObject;

foreach (var property in firstObject.Properties())

{

listView1.Columns.Add(property.Name);

}

foreach (var item in jsonArray)

{

var obj = item as JObject;

ListViewItem listViewItem = new ListViewItem();

bool isFirstColumn = true;

foreach (var property in obj.Properties())

{

if (isFirstColumn)

{

listViewItem.Text = property.Value.ToString();

isFirstColumn = false;

}

else

{

listViewItem.SubItems.Add(property.Value.ToString());

}

}

listView1.Items.Add(listViewItem);

}

}

}

}}

****代码解释****：

* 同样先定义 JSON 字符串并解析。
* 设置 ListView 的视图模式为 Details。
* 根据 JSON 对象的属性名添加列。
* 遍历 JSON 数组，为每个对象创建 ListViewItem，并将属性值添加到 ListViewItem 及其子项中。
* 最后将 ListViewItem 添加到 ListView 中。

## C#、复杂json加载到dataggidview

string jsonData = @"[

{

""ID"": 1,

""Name"": ""John Doe"",

""Contact"": {

""Email"": ""john.doe@example.com"",

""Phone"": ""123-456-7890""

},

""Skills"": [""C#"", ""JavaScript"", ""SQL""]

},

{

""ID"": 2,

""Name"": ""Jane Smith"",

""Contact"": {

""Email"": ""jane.smith@example.com"",

""Phone"": ""098-765-4321""

},

""Skills"": [""Python"", ""Java"", ""HTML""]

}

]";

// 解析 JSON 数据

var jsonArray = JArray.Parse(jsonData);

// 创建 DataTable

DataTable dataTable = new DataTable();

// 添加列

dataTable.Columns.Add("ID", typeof(int));

dataTable.Columns.Add("Name", typeof(string));

dataTable.Columns.Add("Contact.Email", typeof(string));

dataTable.Columns.Add("Contact.Phone", typeof(string));

dataTable.Columns.Add("Skills", typeof(string));

// 填充数据

foreach (JObject jsonObject in jsonArray)

{

DataRow row = dataTable.NewRow();

row["ID"] = jsonObject["ID"].Value<int>();

row["Name"] = jsonObject["Name"].Value<string>();

row["Contact.Email"] = jsonObject["Contact"]["Email"].Value<string>();

row["Contact.Phone"] = jsonObject["Contact"]["Phone"].Value<string>();

row["Skills"] = string.Join(", ", jsonObject["Skills"].ToObject<string[]>());

dataTable.Rows.Add(row);

}

// 将 DataTable 绑定到 DataGridView

dataGridView1.DataSource = dataTable;

## 怎样把 DataGridView 中的数据导出为 JSON 格式？

分享

在 C# 中，若要将 DataGridView 中的数据导出为 JSON 格式，可以借助 Newtonsoft.Json 库来实现。以下是详细的步骤和示例代码：

### **1. 安装 Newtonsoft.Json 库**

你可以使用 NuGet 包管理器或者 .NET CLI 来安装该库。使用 .NET CLI 的命令如下：

收起

bash

dotnet add package Newtonsoft.Json

### **2. 示例代码**

收起

csharp

using System;using System.Collections.Generic;using System.Data;using System.Windows.Forms;using Newtonsoft.Json;

namespace DataGridViewToJson{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

// 模拟添加数据到 DataGridView

DataTable dataTable = new DataTable();

dataTable.Columns.Add("ID", typeof(int));

dataTable.Columns.Add("Name", typeof(string));

dataTable.Rows.Add(1, "John");

dataTable.Rows.Add(2, "Jane");

dataGridView1.DataSource = dataTable;

}

private void buttonExport\_Click(object sender, EventArgs e)

{

// 调用导出方法

string json = ExportDataGridViewToJson(dataGridView1);

// 显示导出的 JSON 数据（这里简单用 MessageBox 显示，实际应用中可保存到文件等）

MessageBox.Show(json);

}

private string ExportDataGridViewToJson(DataGridView dataGridView)

{

List<Dictionary<string, object>> rows = new List<Dictionary<string, object>>();

foreach (DataGridViewRow row in dataGridView.Rows)

{

if (!row.IsNewRow)

{

Dictionary<string, object> rowData = new Dictionary<string, object>();

foreach (DataGridViewColumn column in dataGridView.Columns)

{

rowData[column.HeaderText] = row.Cells[column.Index].Value;

}

rows.Add(rowData);

}

}

// 使用 Newtonsoft.Json 将数据转换为 JSON 字符串

string json = JsonConvert.SerializeObject(rows, Formatting.Indented);

return json;

}

}}

### **3. 代码解释**

* ****模拟数据添加****：在 Form1 的构造函数里，创建了一个 DataTable 并添加了一些示例数据，然后将其绑定到 DataGridView 上。
* ****导出按钮事件处理****：当点击导出按钮时，会触发 buttonExport\_Click 事件，在该事件中调用 ExportDataGridViewToJson 方法进行数据导出。
* **ExportDataGridViewToJson**方法****：
  + 构建一个 List<Dictionary<string, object>> 用于存储 DataGridView 中的每一行数据，其中每个 Dictionary 代表一行，键为列标题，值为单元格的值。
  + 遍历 DataGridView 的每一行（排除新行），为每行创建一个 Dictionary，并将每列的标题和对应单元格的值添加到 Dictionary 中。
  + 使用 JsonConvert.SerializeObject 方法把 List<Dictionary<string, object>> 序列化为 JSON 字符串，Formatting.Indented 参数用于生成格式化后的 JSON 字符串，方便查看。

### **4. 其他应用场景**

在实际应用中，你可以将生成的 JSON 字符串保存到文件中，示例代码如下：

收起

csharp

private void SaveJsonToFile(string json, string filePath){

System.IO.File.WriteAllText(filePath, json);}

// 在 buttonExport\_Click 事件中调用保存方法private void buttonExport\_Click(object sender, EventArgs e){

string json = ExportDataGridViewToJson(dataGridView1);

string filePath = "data.json";

SaveJsonToFile(json, filePath);

MessageBox.Show("数据已导出为 JSON 文件。");}

通过上述步骤，你就能把 DataGridView 中的数据导出为 JSON 格式，并且可以根据需求进行保存或其他处理。

## 什么是SQL 注入？

使用参数化查询是一种防止 SQL 注入的有效方法。

参数化查询是将用户输入的数据作为参数传递给数据库查询语句，而不是将用户输入的数据直接拼接到 SQL 语句中。这样可以避免恶意用户通过输入特定的 SQL 语句来篡改数据库或获取敏感信息。

SQL 注入攻击是一种常见且危险的网络攻击手段，主要针对使用 SQL 语句与数据库进行交互的应用程序。下面将从原理、攻击方式、危害和防范措施几个方面详细解释。

### **原理**

许多 Web 应用程序在处理用户输入时，会将这些输入直接拼接到 SQL 查询语句中。攻击者利用这一特性，通过构造特殊的输入，改变原 SQL 语句的逻辑结构，从而执行恶意的 SQL 命令，绕过应用程序的正常验证机制，非法获取、修改或删除数据库中的数据。

### **攻击方式**

#### **1. 登录验证绕过**

在常见的用户登录表单中，应用程序可能会使用如下 SQL 语句来验证用户输入的用户名和密码：

sql

SELECT \* FROM users WHERE username = '输入的用户名' AND password = '输入的密码';

攻击者可以在用户名输入框中输入 ' OR '1'='1，密码随意输入，此时拼接后的 SQL 语句变为：

sql

SELECT \* FROM users WHERE username = '' OR '1'='1' AND password = '随意输入的内容';

由于 '1'='1' 始终为真，所以整个条件恒为真，攻击者无需正确的用户名和密码就能绕过登录验证。

#### **2. 数据泄露**

攻击者可能会通过构造恶意输入来获取数据库中的敏感信息。例如，在一个查询商品信息的页面中，原 SQL 语句可能是：

sql

SELECT \* FROM products WHERE product\_id = '输入的商品ID';

攻击者可以输入 1' UNION SELECT username, password FROM users --，拼接后的 SQL 语句变为：

sql

SELECT \* FROM products WHERE product\_id = '1' UNION SELECT username, password FROM users --';

UNION 操作符用于合并两个或多个 SELECT 语句的结果集，-- 是 SQL 中的注释符号，会将后面的内容注释掉。这样，攻击者就可以获取 users 表中的用户名和密码信息。

#### **3. 数据修改与删除**

攻击者还可以构造输入来执行修改或删除数据库数据的操作。例如，在一个更新用户信息的功能中，原 SQL 语句可能是：

sql

UPDATE users SET email = '输入的邮箱' WHERE user\_id = '输入的用户ID';

攻击者可以输入 ' WHERE 1=1; DROP TABLE users; --，拼接后的 SQL 语句变为：

sql

UPDATE users SET email = '' WHERE 1=1; DROP TABLE users; --' WHERE user\_id = '输入的用户ID';

这会导致先将所有用户的邮箱更新为空，然后删除 users 表。

### **危害**

* ****数据泄露****：攻击者可以获取数据库中的敏感信息，如用户的个人信息、财务信息等，从而导致用户隐私泄露和财产损失。
* ****数据篡改****：攻击者可以修改数据库中的数据，破坏数据的完整性和一致性，影响业务的正常运行。
* ****数据库破坏****：攻击者可以删除数据库中的重要数据或整个数据库，导致业务系统瘫痪，造成巨大的经济损失。
* ****服务器被控制****：在某些情况下，攻击者可以通过 SQL 注入攻击获得服务器的控制权，进一步实施其他攻击，如安装后门程序、进行分布式拒绝服务（DDoS）攻击等。

### **防范措施**

* ****使用参数化查询****：参数化查询是防止 SQL 注入攻击最有效的方法。大多数编程语言和数据库驱动都支持参数化查询，它将用户输入作为参数传递给 SQL 语句，而不是直接拼接，从而避免了 SQL 语句被恶意修改。例如，在 C# 中使用 MySqlCommand 进行参数化查询：

csharp

string sql = "SELECT \* FROM users WHERE username = @username AND password = @password";MySqlCommand command = new MySqlCommand(sql, connection);

command.Parameters.AddWithValue("@username", inputUsername);

command.Parameters.AddWithValue("@password", inputPassword);

* ****输入验证****：对用户输入进行严格的验证和过滤，只允许合法的字符和格式。例如，对于数字类型的输入，要确保输入的是有效的数字；对于用户名和密码，要限制其长度和字符范围。
* ****最小权限原则****：数据库用户应只被授予完成其任务所需的最小权限。例如，如果一个应用程序只需要查询数据，那么该用户就不应该被授予修改或删除数据的权限。
* ****更新数据库和应用程序****：及时更新数据库管理系统和应用程序，以修复已知的安全漏洞。数据库厂商会定期发布安全补丁，应用程序开发者也会对程序进行更新和维护。

# 配置文件系列

## 1.使用ConfigurationBuilder类获取appsettings.json中的配置信息？

在 C# 中，可以使用ConfigurationBuilder类来读取appsettings.json文件中的配置信息。以下是详细的步骤和示例代码。

### **步骤 1：安装必要的 NuGet 包**

首先，你需要安装Microsoft.Extensions.Configuration.Json NuGet 包，它提供了从 JSON 文件读取配置信息的功能。可以通过以下几种方式进行安装：

#### **使用 NuGet 包管理器控制台**

在 Visual Studio 中，打开 “工具” -> “NuGet 包管理器” -> “包管理器控制台”，然后运行以下命令：

收起

plaintext

Install-Package Microsoft.Extensions.Configuration.Json

#### **使用 .NET CLI**

在命令行中，导航到项目所在的目录，然后运行以下命令：

收起

sh

dotnet add package Microsoft.Extensions.Configuration.Json

### **步骤 2：创建 appsettings.json 文件**

在项目的根目录下创建一个appsettings.json文件，用于存储配置信息。例如：

收起

json

{

"DatabaseSettings": {

"ConnectionString": "Server=localhost;Database=MyDB;User Id=myUser;Password=myPassword;",

"Timeout": 30

},

"AppSettings": {

"ApiKey": "your-api-key",

"EnableFeature": true

}}

### **步骤 3：使用 ConfigurationBuilder 读取配置信息**

以下是一个控制台应用程序的示例，展示了如何使用ConfigurationBuilder读取appsettings.json中的配置信息：

收起

csharp

using Microsoft.Extensions.Configuration;using System;

class Program{

static void Main()

{

// 创建 ConfigurationBuilder 实例

var configuration = new ConfigurationBuilder()

// 设置配置文件的基础路径，通常是应用程序的基目录

.SetBasePath(AppContext.BaseDirectory)

// 添加 appsettings.json 文件作为配置源，设置文件为必需且支持自动重新加载

.AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)

// 构建最终的 IConfiguration 对象

.Build();

// 读取 DatabaseSettings 部分的配置信息

string connectionString = configuration["DatabaseSettings:ConnectionString"];

int timeout = int.Parse(configuration["DatabaseSettings:Timeout"]);

// 读取 AppSettings 部分的配置信息

string apiKey = configuration["AppSettings:ApiKey"];

bool enableFeature = bool.Parse(configuration["AppSettings:EnableFeature"]);

// 输出配置信息

Console.WriteLine($"Database Connection String: {connectionString}");

Console.WriteLine($"Database Timeout: {timeout}");

Console.WriteLine($"API Key: {apiKey}");

Console.WriteLine($"Enable Feature: {enableFeature}");

}}

### **代码解释**

1. ****创建**ConfigurationBuilder**实例****：使用new ConfigurationBuilder()创建一个新的配置构建器实例。
2. ****设置基础路径****：调用SetBasePath(AppContext.BaseDirectory)方法，将配置文件的基础路径设置为应用程序的基目录。
3. ****添加 JSON 文件****：调用AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)方法，将appsettings.json文件添加为配置源。optional: false表示文件是必需的，reloadOnChange: true表示支持文件内容变化时自动重新加载配置。
4. ****构建配置对象****：调用Build()方法，构建最终的IConfiguration对象。
5. ****读取配置信息****：使用configuration["SectionName:KeyName"]的方式读取配置信息。例如，configuration["DatabaseSettings:ConnectionString"]用于读取DatabaseSettings部分的ConnectionString配置项。

### **注意事项**

* 对于嵌套的配置信息，可以使用冒号（:）分隔不同层级的键名。
* 如果配置项的值是整数、布尔值等非字符串类型，需要进行相应的类型转换。

通过以上步骤，你可以使用ConfigurationBuilder类方便地读取appsettings.json中的配置信息。

## 2.使用 ConfigurationBuilder 类来读取其他类型的配置文件，如 XML 或 INI 文件？

分享

在 C# 中，ConfigurationBuilder 类允许你从多种不同类型的配置文件中读取配置信息，除了常见的 JSON 文件，还能处理 XML 和 INI 文件。下面分别介绍如何使用 ConfigurationBuilder 读取 XML 和 INI 文件的配置信息。

### **读取 XML 配置文件**

#### **1. 安装必要的 NuGet 包**

要读取 XML 配置文件，需要安装 Microsoft.Extensions.Configuration.Xml 包。可以通过以下两种方式进行安装：

* ****使用 NuGet 包管理器控制台****：在 Visual Studio 中，打开 “工具” -> “NuGet 包管理器” -> “包管理器控制台”，运行以下命令：

收起

plaintext

Install-Package Microsoft.Extensions.Configuration.Xml

* ****使用 .NET CLI****：在命令行中，导航到项目所在目录，运行以下命令：

收起

sh

dotnet add package Microsoft.Extensions.Configuration.Xml

#### **2. 创建 XML 配置文件**

在项目根目录下创建一个 appsettings.xml 文件，示例内容如下：

收起

xml

<?xml version="1.0" encoding="utf-8"?><configuration>

<DatabaseSettings>

<ConnectionString>Server=localhost;Database=MyDB;User Id=myUser;Password=myPassword;</ConnectionString>

<Timeout>30</Timeout>

</DatabaseSettings>

<AppSettings>

<ApiKey>your-api-key</ApiKey>

<EnableFeature>true</EnableFeature>

</AppSettings></configuration>

#### **3. 使用 ConfigurationBuilder 读取 XML 配置**

以下是一个控制台应用程序示例：

收起

csharp

using Microsoft.Extensions.Configuration;using System;

class Program{

static void Main()

{

var configuration = new ConfigurationBuilder()

.SetBasePath(AppContext.BaseDirectory)

.AddXmlFile("appsettings.xml", optional: false, reloadOnChange: true)

.Build();

string connectionString = configuration["DatabaseSettings:ConnectionString"];

int timeout = int.Parse(configuration["DatabaseSettings:Timeout"]);

string apiKey = configuration["AppSettings:ApiKey"];

bool enableFeature = bool.Parse(configuration["AppSettings:EnableFeature"]);

Console.WriteLine($"Database Connection String: {connectionString}");

Console.WriteLine($"Database Timeout: {timeout}");

Console.WriteLine($"API Key: {apiKey}");

Console.WriteLine($"Enable Feature: {enableFeature}");

}}

****代码解释****：

* SetBasePath(AppContext.BaseDirectory)：设置配置文件的基础路径为应用程序的基目录。
* AddXmlFile("appsettings.xml", optional: false, reloadOnChange: true)：将 appsettings.xml 文件添加为配置源，optional: false 表示文件是必需的，reloadOnChange: true 表示支持文件内容变化时自动重新加载配置。
* Build()：构建最终的 IConfiguration 对象。

### **读取 INI 配置文件**

#### **1. 安装必要的 NuGet 包**

要读取 INI 配置文件，需要安装 Microsoft.Extensions.Configuration.Ini 包。可以通过以下两种方式进行安装：

* ****使用 NuGet 包管理器控制台****：在 Visual Studio 中，打开 “工具” -> “NuGet 包管理器” -> “包管理器控制台”，运行以下命令：

收起

plaintext

Install-Package Microsoft.Extensions.Configuration.Ini

* ****使用 .NET CLI****：在命令行中，导航到项目所在目录，运行以下命令：

收起

sh

dotnet add package Microsoft.Extensions.Configuration.Ini

#### **2. 创建 INI 配置文件**

在项目根目录下创建一个 appsettings.ini 文件，示例内容如下：

收起

ini

[DatabaseSettings]ConnectionString=Server=localhost;Database=MyDB;User Id=myUser;Password=myPassword;Timeout=30

[AppSettings]ApiKey=your-api-keyEnableFeature=true

#### **3. 使用 ConfigurationBuilder 读取 INI 配置**

以下是一个控制台应用程序示例：

收起

csharp

using Microsoft.Extensions.Configuration;using System;

class Program{

static void Main()

{

var configuration = new ConfigurationBuilder()

.SetBasePath(AppContext.BaseDirectory)

.AddIniFile("appsettings.ini", optional: false, reloadOnChange: true)

.Build();

string connectionString = configuration["DatabaseSettings:ConnectionString"];

int timeout = int.Parse(configuration["DatabaseSettings:Timeout"]);

string apiKey = configuration["AppSettings:ApiKey"];

bool enableFeature = bool.Parse(configuration["AppSettings:EnableFeature"]);

Console.WriteLine($"Database Connection String: {connectionString}");

Console.WriteLine($"Database Timeout: {timeout}");

Console.WriteLine($"API Key: {apiKey}");

Console.WriteLine($"Enable Feature: {enableFeature}");

}}

****代码解释****：

* SetBasePath(AppContext.BaseDirectory)：设置配置文件的基础路径为应用程序的基目录。
* AddIniFile("appsettings.ini", optional: false, reloadOnChange: true)：将 appsettings.ini 文件添加为配置源，optional: false 表示文件是必需的，reloadOnChange: true 表示支持文件内容变化时自动重新加载配置。
* Build()：构建最终的 IConfiguration 对象。

通过上述步骤，你可以使用 ConfigurationBuilder 类从 XML 和 INI 配置文件中读取配置信息。

## 3.如何使用 ConfigurationBuilder 类来读取配置文件中的加密信息？

在 C# 中，ConfigurationBuilder 本身并不直接支持读取加密的配置信息，但可以通过自定义配置提供程序或在读取配置后进行解密操作来实现读取加密配置文件的功能。以下分别介绍两种常见的实现方式。

### **方式一：读取配置后进行解密操作**

#### **1. 加密配置文件**

首先，需要对配置文件中的敏感信息进行加密。可以使用对称加密算法，如 AES（高级加密标准）。以下是一个简单的加密示例：

收起

csharp

using System;using System.IO;using System.Security.Cryptography;using System.Text;

public class EncryptionHelper{

public static string Encrypt(string plainText, string key)

{

byte[] encrypted;

using (Aes aesAlg = Aes.Create())

{

aesAlg.Key = Encoding.UTF8.GetBytes(key);

aesAlg.IV = new byte[aesAlg.BlockSize / 8];

ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);

using (MemoryStream msEncrypt = new MemoryStream())

{

using (CryptoStream csEncrypt = new CryptoStream(msEncrypt, encryptor, CryptoStreamMode.Write))

{

using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))

{

swEncrypt.Write(plainText);

}

encrypted = msEncrypt.ToArray();

}

}

}

return Convert.ToBase64String(encrypted);

}

public static string Decrypt(string cipherText, string key)

{

string plaintext = null;

byte[] cipherBytes = Convert.FromBase64String(cipherText);

using (Aes aesAlg = Aes.Create())

{

aesAlg.Key = Encoding.UTF8.GetBytes(key);

aesAlg.IV = new byte[aesAlg.BlockSize / 8];

ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);

using (MemoryStream msDecrypt = new MemoryStream(cipherBytes))

{

using (CryptoStream csDecrypt = new CryptoStream(msDecrypt, decryptor, CryptoStreamMode.Read))

{

using (StreamReader srDecrypt = new StreamReader(csDecrypt))

{

plaintext = srDecrypt.ReadToEnd();

}

}

}

}

return plaintext;

}}

#### **2. 创建加密的配置文件**

假设我们有一个 appsettings.json 文件，将其中的敏感信息进行加密后保存：

收起

json

{

"DatabaseSettings": {

"ConnectionString": "这里可以是加密后的连接字符串",

"Password": "这里可以是加密后的密码"

}}

#### **3. 使用 ConfigurationBuilder 读取配置并解密**

收起

csharp

using Microsoft.Extensions.Configuration;using System;

class Program{

static void Main()

{

var configuration = new ConfigurationBuilder()

.SetBasePath(AppContext.BaseDirectory)

.AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)

.Build();

string encryptedConnectionString = configuration["DatabaseSettings:ConnectionString"];

string encryptionKey = "your-encryption-key"; // 加密密钥

string decryptedConnectionString = EncryptionHelper.Decrypt(encryptedConnectionString, encryptionKey);

Console.WriteLine($"Decrypted Connection String: {decryptedConnectionString}");

}}

### **方式二：自定义配置提供程序**

#### **1. 创建自定义配置提供程序**

自定义一个配置提供程序，在读取配置时进行解密操作：

收起

csharp

using Microsoft.Extensions.Configuration;using System;using System.Collections.Generic;using System.IO;using System.Text;

public class EncryptedJsonConfigurationProvider : ConfigurationProvider{

private readonly string \_path;

private readonly string \_encryptionKey;

public EncryptedJsonConfigurationProvider(string path, string encryptionKey)

{

\_path = path;

\_encryptionKey = encryptionKey;

}

public override void Load()

{

if (File.Exists(\_path))

{

var json = File.ReadAllText(\_path);

var decryptedJson = EncryptionHelper.Decrypt(json, \_encryptionKey);

Data = ParseJson(decryptedJson);

}

}

private IDictionary<string, string> ParseJson(string json)

{

// 这里可以使用 JSON 解析库（如 Newtonsoft.Json）来解析 JSON 字符串

// 并将其转换为键值对的字典

// 为了简化示例，这里省略具体实现

return new Dictionary<string, string>();

}}

public class EncryptedJsonConfigurationSource : IConfigurationSource{

private readonly string \_path;

private readonly string \_encryptionKey;

public EncryptedJsonConfigurationSource(string path, string encryptionKey)

{

\_path = path;

\_encryptionKey = encryptionKey;

}

public IConfigurationProvider Build(IConfigurationBuilder builder)

{

return new EncryptedJsonConfigurationProvider(\_path, \_encryptionKey);

}}

#### **2. 使用自定义配置提供程序**

收起

csharp

using Microsoft.Extensions.Configuration;using System;

class Program{

static void Main()

{

string encryptionKey = "your-encryption-key";

var configuration = new ConfigurationBuilder()

.Add(new EncryptedJsonConfigurationSource("appsettings.encrypted.json", encryptionKey))

.Build();

string connectionString = configuration["DatabaseSettings:ConnectionString"];

Console.WriteLine($"Connection String: {connectionString}");

}}

### **注意事项**

* ****密钥管理****：加密密钥的安全性至关重要，不要将密钥硬编码在代码中，可以使用环境变量、密钥管理服务（如 Azure Key Vault）等方式来安全存储密钥。
* ****加密算法选择****：示例中使用的 AES 算法是一种常用的对称加密算法，在实际应用中可以根据需求选择更合适的加密算法。
* ****错误处理****：在加密和解密过程中，需要进行适当的错误处理，以确保程序的健壮性。

## 3-1.如何对配置文件中的所有信息进行加密？

要对配置文件中的所有信息进行加密，可以按照以下步骤操作，下面以 JSON 配置文件为例，结合 C# 语言和 AES 加密算法详细说明。

### **1. 选择加密算法**

这里选用 AES（高级加密标准）对称加密算法，它具有较高的安全性和效率。

### **2. 安装必要的 NuGet 包**

若使用 .NET 项目，可直接使用 System.Security.Cryptography 命名空间中的类来实现 AES 加密，无需额外安装 NuGet 包。

### **3. 实现加密和解密方法**

收起

csharp

using System;using System.IO;using System.Security.Cryptography;using System.Text;

public class EncryptionHelper{

// 加密方法

public static string Encrypt(string plainText, string key)

{

byte[] encrypted;

using (Aes aesAlg = Aes.Create())

{

// 设置加密密钥

aesAlg.Key = Encoding.UTF8.GetBytes(key);

// 初始化向量（IV），这里简单设置为全 0，实际应用中应随机生成并妥善保存

aesAlg.IV = new byte[aesAlg.BlockSize / 8];

// 创建加密转换对象

ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);

using (MemoryStream msEncrypt = new MemoryStream())

{

using (CryptoStream csEncrypt = new CryptoStream(msEncrypt, encryptor, CryptoStreamMode.Write))

{

using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))

{

// 写入明文

swEncrypt.Write(plainText);

}

encrypted = msEncrypt.ToArray();

}

}

}

// 将加密后的字节数组转换为 Base64 字符串

return Convert.ToBase64String(encrypted);

}

// 解密方法

public static string Decrypt(string cipherText, string key)

{

string plaintext = null;

// 将 Base64 字符串转换为字节数组

byte[] cipherBytes = Convert.FromBase64String(cipherText);

using (Aes aesAlg = Aes.Create())

{

// 设置解密密钥

aesAlg.Key = Encoding.UTF8.GetBytes(key);

// 初始化向量（IV），需与加密时使用的一致

aesAlg.IV = new byte[aesAlg.BlockSize / 8];

// 创建解密转换对象

ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);

using (MemoryStream msDecrypt = new MemoryStream(cipherBytes))

{

using (CryptoStream csDecrypt = new CryptoStream(msDecrypt, decryptor, CryptoStreamMode.Read))

{

using (StreamReader srDecrypt = new StreamReader(csDecrypt))

{

// 读取解密后的明文

plaintext = srDecrypt.ReadToEnd();

}

}

}

}

return plaintext;

}}

### **4. 加密配置文件**

收起

csharp

using System.IO;

class Program{

static void Main()

{

// 配置文件路径

string configFilePath = "appsettings.json";

// 加密密钥，实际应用中应妥善保管，可通过环境变量等方式获取

string encryptionKey = "your-32-byte-encryption-key";

// 读取配置文件内容

string configContent = File.ReadAllText(configFilePath);

// 加密配置文件内容

string encryptedContent = EncryptionHelper.Encrypt(configContent, encryptionKey);

// 保存加密后的内容到新文件

string encryptedFilePath = "appsettings.encrypted.json";

File.WriteAllText(encryptedFilePath, encryptedContent);

Console.WriteLine("Configuration file encrypted successfully.");

}}

### **5. 解密并读取配置文件**

收起

csharp

using Microsoft.Extensions.Configuration;using System;

class Program{

static void Main()

{

// 加密配置文件路径

string encryptedFilePath = "appsettings.encrypted.json";

// 加密密钥，需与加密时使用的一致

string encryptionKey = "your-32-byte-encryption-key";

// 读取加密后的配置文件内容

string encryptedContent = File.ReadAllText(encryptedFilePath);

// 解密配置文件内容

string decryptedContent = EncryptionHelper.Decrypt(encryptedContent, encryptionKey);

// 创建临时配置文件用于读取

string tempConfigFilePath = "appsettings.temp.json";

File.WriteAllText(tempConfigFilePath, decryptedContent);

// 使用 ConfigurationBuilder 读取配置

var configuration = new ConfigurationBuilder()

.SetBasePath(AppContext.BaseDirectory)

.AddJsonFile(tempConfigFilePath, optional: false, reloadOnChange: false)

.Build();

// 读取配置项示例

string connectionString = configuration["DatabaseSettings:ConnectionString"];

Console.WriteLine($"Connection String: {connectionString}");

// 删除临时配置文件

File.Delete(tempConfigFilePath);

}}

### **注意事项**

* ****密钥管理****：加密密钥的安全性至关重要，不要将密钥硬编码在代码中。可以使用环境变量、密钥管理服务（如 Azure Key Vault、AWS Secrets Manager）等方式来安全存储和管理密钥。
* ****初始化向量（IV）****：示例中简单地将 IV 设置为全 0，实际应用中应随机生成 IV 并与加密数据一起保存，以提高加密的安全性。
* ****错误处理****：在加密、解密和文件操作过程中，需要进行适当的错误处理，以确保程序的健壮性。

通过以上步骤，你可以对配置文件中的所有信息进行加密，并在需要时解密读取。

# 数学基础：矩阵

## 1、C#用齐次矩阵平移、旋转、缩放二维空间坐标

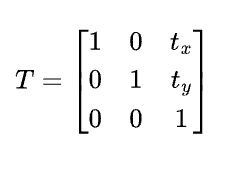
在 C# 中使用齐次矩阵对二维空间坐标进行平移、旋转和缩放操作，下面为你详细介绍实现步骤和示例代码。

### **原理概述**

在二维空间里，我们可以借助 3x3 的齐次矩阵来实现平移、旋转和缩放操作。以下是三种操作对应的齐次矩阵：

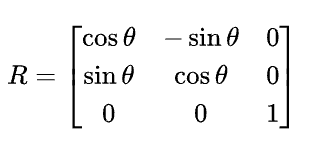
#### **平移矩阵**

假设在 x 方向平移 ，在 y 方向平移 ，平移矩阵  为：



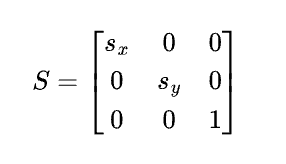
#### **旋转矩阵**

假设绕原点旋转θ角度，旋转矩阵R为：



#### **缩放矩阵**

假设在 x 方向缩放Sx倍，在 y 方向缩放Sy倍，缩放矩阵  为：



### **代码实现**

收起

csharp

using System;

class HomogeneousMatrix2D{

// 定义 3x3 齐次矩阵类

public class Matrix3x3

{

public double[,] elements = new double[3, 3];

public Matrix3x3()

{

// 初始化为单位矩阵

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++)

{

if (i == j)

elements[i, j] = 1;

else

elements[i, j] = 0;

}

}

}

// 矩阵乘法

public static Matrix3x3 Multiply(Matrix3x3 a, Matrix3x3 b)

{

Matrix3x3 result = new Matrix3x3();

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++)

{

result.elements[i, j] = 0;

for (int k = 0; k < 3; k++)

{

result.elements[i, j] += a.elements[i, k] \* b.elements[k, j];

}

}

}

return result;

}

// 矩阵乘以二维点（齐次坐标）

public static double[] Multiply(Matrix3x3 matrix, double[] point)

{

double[] result = new double[3];

for (int i = 0; i < 3; i++)

{

result[i] = 0;

for (int j = 0; j < 3; j++)

{

result[i] += matrix.elements[i, j] \* point[j];

}

}

return result;

}

// 创建平移矩阵

public static Matrix3x3 Translation(double tx, double ty)

{

Matrix3x3 translationMatrix = new Matrix3x3();

translationMatrix.elements[0, 2] = tx;

translationMatrix.elements[1, 2] = ty;

return translationMatrix;

}

// 创建旋转矩阵

public static Matrix3x3 Rotation(double angle)

{

Matrix3x3 rotationMatrix = new Matrix3x3();

double cosAngle = Math.Cos(angle);

double sinAngle = Math.Sin(angle);

rotationMatrix.elements[0, 0] = cosAngle;

rotationMatrix.elements[0, 1] = -sinAngle;

rotationMatrix.elements[1, 0] = sinAngle;

rotationMatrix.elements[1, 1] = cosAngle;

return rotationMatrix;

}

// 创建缩放矩阵

public static Matrix3x3 Scaling(double sx, double sy)

{

Matrix3x3 scalingMatrix = new Matrix3x3();

scalingMatrix.elements[0, 0] = sx;

scalingMatrix.elements[1, 1] = sy;

return scalingMatrix;

}

}

static void Main()

{

// 定义一个二维点，转换为齐次坐标 [x, y, 1]

double[] point = { 1, 2, 1 };

// 平移操作

double tx = 3;

double ty = 4;

Matrix3x3 translationMatrix = Matrix3x3.Translation(tx, ty);

double[] translatedPoint = Matrix3x3.Multiply(translationMatrix, point);

// 旋转操作（绕原点旋转 45 度，转换为弧度）

double angle = Math.PI / 4;

Matrix3x3 rotationMatrix = Matrix3x3.Rotation(angle);

double[] rotatedPoint = Matrix3x3.Multiply(rotationMatrix, point);

// 缩放操作

double sx = 2;

double sy = 3;

Matrix3x3 scalingMatrix = Matrix3x3.Scaling(sx, sy);

double[] scaledPoint = Matrix3x3.Multiply(scalingMatrix, point);

// 组合操作：先平移，再旋转，最后缩放

Matrix3x3 combinedMatrix = Matrix3x3.Multiply(scalingMatrix, Matrix3x3.Multiply(rotationMatrix, translationMatrix));

double[] combinedTransformedPoint = Matrix3x3.Multiply(combinedMatrix, point);

// 输出结果

Console.WriteLine($"平移后的坐标: ({translatedPoint[0]}, {translatedPoint[1]})");

Console.WriteLine($"旋转后的坐标: ({rotatedPoint[0]}, {rotatedPoint[1]})");

Console.WriteLine($"缩放后的坐标: ({scaledPoint[0]}, {scaledPoint[1]})");

Console.WriteLine($"组合变换后的坐标: ({combinedTransformedPoint[0]}, {combinedTransformedPoint[1]})");

}}

### **代码解释**

1. **Matrix3x3**类****：用于表示 3x3 的齐次矩阵，包含矩阵初始化、矩阵乘法、矩阵乘以二维点（齐次坐标）以及创建平移、旋转和缩放矩阵的方法。
2. **Multiply**方法****：实现了矩阵乘法和矩阵乘以二维点（齐次坐标）的操作。
3. **Translation**方法****：创建平移矩阵。
4. **Rotation**方法****：创建旋转矩阵。
5. **Scaling**方法****：创建缩放矩阵。
6. **Main**方法****：定义一个二维点并转换为齐次坐标，分别进行平移、旋转、缩放操作，还进行了组合操作，最后输出变换后的坐标。

### **注意事项**

* 角度需使用弧度制，可通过 Math.PI 进行角度与弧度的转换。
* 组合操作时，矩阵相乘的顺序很重要，因为矩阵乘法不满足交换律。

## 2、C#三维图像的平移、旋转、缩放

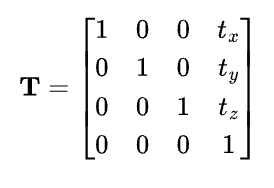
在 C# 中实现三维图像的平移、旋转和缩放操作，通常会借助齐次矩阵来完成。以下为你详细介绍具体实现步骤和示例代码：

### **原理概述**

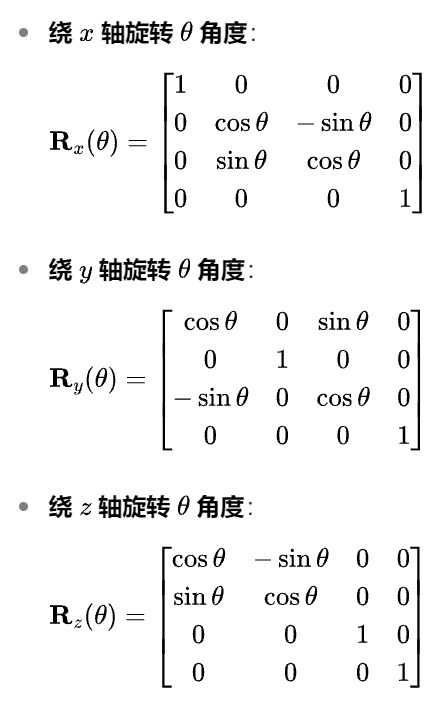
在三维空间里，可使用 4x4 的齐次矩阵来表示平移、旋转和缩放变换。通过将三维点的齐次坐标与相应的变换矩阵相乘，就能实现对该点的变换。以下是三种变换对应的矩阵：

#### **平移矩阵**

假设在  方向平移 ， 方向平移 ， 方向平移 ，平移矩阵  为：

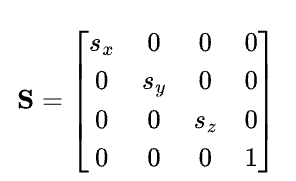


#### **旋转矩阵**



#### **缩放矩阵**

假设在  方向缩放  倍， 方向缩放  倍， 方向缩放  倍，缩放矩阵  为：



### **代码实现**

收起

csharp

using System;

class ThreeDTransformations{

// 定义 4x4 齐次矩阵类

public class Matrix4x4

{

public double[,] Elements = new double[4, 4];

public Matrix4x4()

{

// 初始化为单位矩阵

for (int i = 0; i < 4; i++)

{

for (int j = 0; j < 4; j++)

{

if (i == j)

Elements[i, j] = 1;

else

Elements[i, j] = 0;

}

}

}

// 矩阵乘法

public static Matrix4x4 Multiply(Matrix4x4 a, Matrix4x4 b)

{

Matrix4x4 result = new Matrix4x4();

for (int i = 0; i < 4; i++)

{

for (int j = 0; j < 4; j++)

{

result.Elements[i, j] = 0;

for (int k = 0; k < 4; k++)

{

result.Elements[i, j] += a.Elements[i, k] \* b.Elements[k, j];

}

}

}

return result;

}

// 矩阵乘以三维点（齐次坐标）

public static double[] Multiply(Matrix4x4 matrix, double[] point)

{

double[] result = new double[4];

for (int i = 0; i < 4; i++)

{

result[i] = 0;

for (int j = 0; j < 4; j++)

{

result[i] += matrix.Elements[i, j] \* point[j];

}

}

return result;

}

// 创建平移矩阵

public static Matrix4x4 CreateTranslationMatrix(double tx, double ty, double tz)

{

Matrix4x4 translationMatrix = new Matrix4x4();

translationMatrix.Elements[0, 3] = tx;

translationMatrix.Elements[1, 3] = ty;

translationMatrix.Elements[2, 3] = tz;

return translationMatrix;

}

// 创建绕 x 轴旋转的矩阵

public static Matrix4x4 CreateRotationXMatrix(double angle)

{

Matrix4x4 rotationMatrix = new Matrix4x4();

double cosAngle = Math.Cos(angle);

double sinAngle = Math.Sin(angle);

rotationMatrix.Elements[1, 1] = cosAngle;

rotationMatrix.Elements[1, 2] = -sinAngle;

rotationMatrix.Elements[2, 1] = sinAngle;

rotationMatrix.Elements[2, 2] = cosAngle;

return rotationMatrix;

}

// 创建绕 y 轴旋转的矩阵

public static Matrix4x4 CreateRotationYMatrix(double angle)

{

Matrix4x4 rotationMatrix = new Matrix4x4();

double cosAngle = Math.Cos(angle);

double sinAngle = Math.Sin(angle);

rotationMatrix.Elements[0, 0] = cosAngle;

rotationMatrix.Elements[0, 2] = sinAngle;

rotationMatrix.Elements[2, 0] = -sinAngle;

rotationMatrix.Elements[2, 2] = cosAngle;

return rotationMatrix;

}

// 创建绕 z 轴旋转的矩阵

public static Matrix4x4 CreateRotationZMatrix(double angle)

{

Matrix4x4 rotationMatrix = new Matrix4x4();

double cosAngle = Math.Cos(angle);

double sinAngle = Math.Sin(angle);

rotationMatrix.Elements[0, 0] = cosAngle;

rotationMatrix.Elements[0, 1] = -sinAngle;

rotationMatrix.Elements[1, 0] = sinAngle;

rotationMatrix.Elements[1, 1] = cosAngle;

return rotationMatrix;

}

// 创建缩放矩阵

public static Matrix4x4 CreateScalingMatrix(double sx, double sy, double sz)

{

Matrix4x4 scalingMatrix = new Matrix4x4();

scalingMatrix.Elements[0, 0] = sx;

scalingMatrix.Elements[1, 1] = sy;

scalingMatrix.Elements[2, 2] = sz;

return scalingMatrix;

}

}

static void Main()

{

// 定义一个三维点，转换为齐次坐标 [x, y, z, 1]

double[] point = { 1, 2, 3, 1 };

// 平移操作

double tx = 2;

double ty = 3;

double tz = 4;

Matrix4x4 translationMatrix = Matrix4x4.CreateTranslationMatrix(tx, ty, tz);

double[] translatedPoint = Matrix4x4.Multiply(translationMatrix, point);

// 绕 x 轴旋转操作（旋转 45 度，转换为弧度）

double rotationAngleX = Math.PI / 4;

Matrix4x4 rotationXMatrix = Matrix4x4.CreateRotationXMatrix(rotationAngleX);

double[] rotatedXPoint = Matrix4x4.Multiply(rotationXMatrix, point);

// 缩放操作

double sx = 1.5;

double sy = 2;

double sz = 2.5;

Matrix4x4 scalingMatrix = Matrix4x4.CreateScalingMatrix(sx, sy, sz);

double[] scaledPoint = Matrix4x4.Multiply(scalingMatrix, point);

// 输出结果

Console.WriteLine($"原始点: ({point[0]}, {point[1]}, {point[2]})");

Console.WriteLine($"平移后的点: ({translatedPoint[0]}, {translatedPoint[1]}, {translatedPoint[2]})");

Console.WriteLine($"绕 x 轴旋转后的点: ({rotatedXPoint[0]}, {rotatedXPoint[1]}, {rotatedXPoint[2]})");

Console.WriteLine($"缩放后的点: ({scaledPoint[0]}, {scaledPoint[1]}, {scaledPoint[2]})");

}}

### **代码解释**

1. **Matrix4x4**类****：
   * Elements：用于存储 4x4 矩阵的元素。
   * 构造函数：将矩阵初始化为单位矩阵。
   * Multiply 方法：实现了矩阵乘法和矩阵与三维点（齐次坐标）的乘法。
   * CreateTranslationMatrix 方法：创建平移矩阵。
   * CreateRotationXMatrix、CreateRotationYMatrix、CreateRotationZMatrix 方法：分别创建绕 、、 轴旋转的矩阵。
   * CreateScalingMatrix 方法：创建缩放矩阵。
2. **Main**方法****：
   * 定义一个三维点并转换为齐次坐标。
   * 分别进行平移、绕  轴旋转和缩放操作，创建对应的矩阵并与原始点相乘。
   * 输出原始点以及经过平移、旋转和缩放后的点的坐标。

### **注意事项**

* 旋转角度需使用弧度制，若输入的是角度值，可通过 Math.PI / 180 \* degrees 将其转换为弧度。
* 矩阵乘法不满足交换律，不同的变换顺序可能会得到不同的结果。

# 通讯：socket

C#socket 套接字 服务器、客户端，添加线程安全、连接超时、连接有效检查、断线重连，其中服务端和客户端发送数据功能，单独写一个方法。

### 1、服务端

using System;

using System.Collections.Generic;

using System.Linq;

using System.Net;

using System.Net.Sockets;

using System.Text;

using System.Threading.Tasks;

namespace TCPIPServerClient01

{

public class MyServer

{

private static readonly object \_locker = new object();

private static List<Socket> \_clients = new List<Socket>();

static Socket serverSocket;

public static void StartTcpServer()

{

// 创建一个 TCP/IP 套接字

/\*Socket\*/

serverSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

// 绑定到本地地址和端口

IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, 8080);

serverSocket.Bind(localEndPoint);

// 开始监听，最大连接数为 10

serverSocket.Listen(10);

Console.WriteLine("Server started, listening on port 8888...");

// 启动一个线程来接受客户端连接

Thread acceptThread = new Thread(AcceptClients);

acceptThread.Start(serverSocket);

// 启动一个线程来检查连接有效性

Thread checkConnectionThread = new Thread(CheckConnections);

checkConnectionThread.Start();

Console.WriteLine("TCP服务器启动");

}

private static void AcceptClients(object obj)

{

//Socket serverSocket = (Socket)obj;

while (true)

{

try

{

// 接受客户端连接

Socket clientSocket = serverSocket.Accept();

Console.WriteLine("Client connected: " + clientSocket.RemoteEndPoint);

lock (\_locker)

{

\_clients.Add(clientSocket);

}

// 为每个客户端启动一个线程来处理消息

Thread clientThread = new Thread(() => HandleClient(clientSocket));

clientThread.Start();

}

catch (Exception ex)

{

Console.WriteLine("Error accepting client: " + ex.Message);

}

}

}

private static void HandleClient(Socket clientSocket)

{

try

{

byte[] buffer = new byte[1024];

int bytesRead;

while ((bytesRead = clientSocket.Receive(buffer)) > 0)

{

string message = System.Text.Encoding.UTF8.GetString(buffer, 0, bytesRead);// Encoding.UTF8.GetString(buffer, 0, bytesRead);

Console.WriteLine("Received from client " + clientSocket.RemoteEndPoint + ": " + message);

// 回显消息给客户端

clientSocket.Send(System.Text.Encoding.UTF8.GetBytes("Server received: " + message));

//SendDataToClient(clientSocket, "Server received: " + message);

}

}

catch (Exception ex)

{

Console.WriteLine("Error handling client: " + ex.Message);

}

//finally

//{

// lock (\_locker)

// {

// \_clients.Remove(clientSocket);

// }

// clientSocket.Close();

//}

}

private static void CheckConnections()

{

while (true)

{

Thread.Sleep(5000); // 每 5 秒检查一次连接

lock (\_locker)

{

for (int i = \_clients.Count - 1; i >= 0; i--)

{

Socket client = \_clients[i];

try

{

// 检查连接是否有效

if (!IsSocketConnected(client))

{

Console.WriteLine("Client disconnected: " + client.RemoteEndPoint);

\_clients.RemoveAt(i);

client.Close();

}

}

catch (Exception ex)

{

Console.WriteLine("Error checking connection: " + ex.Message);

}

}

}

}

}

private static bool IsSocketConnected(Socket s)

{

return !((s.Poll(1000, SelectMode.SelectRead) && (s.Available == 0)) || !s.Connected);

}

private static void SendDataToClient(Socket clientSocket, string message)

{

try

{

byte[] data = Encoding.UTF8.GetBytes(message);

clientSocket.Send(data);

}

catch (Exception ex)

{

Console.WriteLine("Error sending data to client: " + ex.Message);

}

}

}

}

### 2、客户端

#region vertion2.0

private static Socket \_clientSocket;

private static IPEndPoint \_serverEndPoint;

private static ManualResetEvent \_connectDone = new ManualResetEvent(false);

private static readonly object \_locker = new object();

private static bool \_isConnected = false;

public static void StartTcpClient( )

{

\_serverEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8080);

\_clientSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

// 尝试连接服务器

ConnectToServer();

// 启动一个线程来接收服务器消息

Thread receiveThread = new Thread(ReceiveMessages);

receiveThread.Start();

// 启动一个线程来检查连接并进行断线重连

Thread reconnectThread = new Thread(CheckAndReconnect);

reconnectThread.Start();

//// 发送消息给服务器

//while (true)

//{

// string message = Console.ReadLine();

// if (message.ToLower() == "exit")

// {

// break;

// }

// try

// {

// lock (\_locker)

// {

// if (\_isConnected)

// {

// \_clientSocket.Send(Encoding.ASCII.GetBytes(message));

// }

// else

// {

// Console.WriteLine("Not connected to the server.");

// }

// }

// }

// catch (Exception ex)

// {

// Console.WriteLine("Error sending message: " + ex.Message);

// }

//}

//\_clientSocket.Close();

}

private static void ConnectToServer()

{

try

{

\_connectDone.Reset();

\_clientSocket.BeginConnect(\_serverEndPoint, new AsyncCallback(ConnectCallback), \_clientSocket);

// 等待连接完成，超时时间为 5 秒

if (!\_connectDone.WaitOne(5000))

{

Console.WriteLine("Connection timed out.");

\_clientSocket.Close();

\_clientSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

}

}

catch (Exception ex)

{

Console.WriteLine("Error connecting to server: " + ex.Message);

}

}

private static void ConnectCallback(IAsyncResult ar)

{

try

{

Socket clientSocket = (Socket)ar.AsyncState;

clientSocket.EndConnect(ar);

lock (\_locker)

{

\_isConnected = true;

}

Console.WriteLine("Connected to server.");

}

catch (Exception ex)

{

Console.WriteLine("Error connecting to server: " + ex.Message);

lock (\_locker)

{

\_isConnected = false;

}

}

finally

{

\_connectDone.Set();

}

}

private static void ReceiveMessages()

{

try

{

byte[] buffer = new byte[1024];

int bytesRead;

while (true)

{

lock (\_locker)

{

if (!\_isConnected)

{

Thread.Sleep(1000);

continue;

}

}

try

{

bytesRead = \_clientSocket.Receive(buffer);

if (bytesRead > 0)

{

string message = System.Text.Encoding.UTF8.GetString(buffer, 0, bytesRead);

Console.WriteLine("Received from server: " + message);

}

else

{

// 连接断开

lock (\_locker)

{

\_isConnected = false;

}

Console.WriteLine("Connection to server lost.");

}

}

catch (SocketException)

{

// 连接断开

lock (\_locker)

{

\_isConnected = false;

}

Console.WriteLine("Connection to server lost.");

}

}

}

catch (Exception ex)

{

Console.WriteLine("Error receiving message: " + ex.Message);

}

}

private static void CheckAndReconnect()

{

while (true)

{

Thread.Sleep(5000); // 每 5 秒检查一次连接

lock (\_locker)

{

if (!\_isConnected)

{

Console.WriteLine("Trying to reconnect...");

ConnectToServer();

}

}

}

}

public static void SendDataToServer(string message)

{

try

{

byte[] data = System.Text.Encoding.UTF8.GetBytes(message);

\_clientSocket.Send(data);

}

catch (Exception ex)

{

Console.WriteLine("Error sending data to server: " + ex.Message);

}

}

}

#endregion

#### 代码解释

#### **服务器端**

* **SendDataToClient**方法****：该方法接受一个 Socket 对象和一个字符串消息作为参数，将消息转换为字节数组并通过 Socket 发送给客户端。如果发送过程中出现异常，会输出错误信息。
* ****线程安全****：使用 \_locker 对 \_clients 列表的访问进行加锁，确保多线程操作时数据的一致性。
* ****连接超时****：无特别处理，因为服务器端是监听连接，不存在主动连接超时的情况。
* ****连接有效检查****：CheckConnections 方法每隔 5 秒检查一次客户端连接状态，通过 IsSocketConnected 方法判断连接是否断开。
* ****断线重连****：服务器端不需要断线重连，因为它是被动接受连接。

##### C#解释(socket.Poll(1000, SelectMode.SelectRead) && (s.Available == 0)) || !s.Connected

在 C# 中，(socket.Poll(1000, SelectMode.SelectRead) && (socket.Available == 0)) || !socket.Connected 这一表达式用于判断一个 Socket 对象的连接状态是否已断开。下面我们来详细解释这个表达式的各个部分。

###### 整体功能

该表达式的主要目的是检测 Socket 连接是否仍然有效。如果表达式结果为 true，则表示连接已断开；如果结果为 false，则表示连接仍然正常。

###### 表达式各部分解释

**socket.Poll(1000, SelectMode.SelectRead)**

* **Poll**方法****：Socket 类的 Poll 方法用于检查 Socket 的状态，判断它是否有数据可读、可写或是否有错误发生。
* ****参数解释****：
  + **1000**：这是超时时间，单位为毫秒。表示 Poll 方法会在 1000 毫秒（即 1 秒）内尝试检查 Socket 的状态。如果在这个时间内没有满足条件的状态变化，Poll 方法将返回 false。
  + **SelectMode.SelectRead**：SelectMode 是一个枚举类型，SelectMode.SelectRead 表示检查 Socket 是否有数据可读，或者连接是否已关闭、重置或终止。如果 Poll 方法在指定的超时时间内检测到 Socket 有数据可读或者连接已关闭等情况，将返回 true；否则返回 false。

**socket.Available == 0**

* **Available**属性****：Socket 类的 Available 属性返回当前 Socket 接收缓冲区中可用的字节数。
* ****含义****：如果 Available 属性的值为 0，说明接收缓冲区中没有可用的数据。结合 socket.Poll(1000, SelectMode.SelectRead) 为 true 的情况，意味着在检查时没有检测到新的数据可读，并且接收缓冲区为空，这可能是连接已关闭的一个迹象。

**!socket.Connected**

* **Connected**属性****：Socket 类的 Connected 属性是一个布尔值，用于表示 Socket 是否已经连接到远程主机。
* ****含义****：!socket.Connected 表示 Socket 没有连接到远程主机。如果这个条件为 true，则直接说明连接已经断开。

表达式逻辑

整个表达式 (socket.Poll(1000, SelectMode.SelectRead) && (socket.Available == 0)) || !socket.Connected 采用了逻辑或（||）和逻辑与（&&）运算符。具体逻辑如下：

* 如果 socket.Poll(1000, SelectMode.SelectRead) 为 true，表示 Socket 有数据可读或者连接已关闭等情况，并且 socket.Available == 0 也为 true，即接收缓冲区为空，那么说明连接可能已经断开。
* 如果 !socket.Connected 为 true，即 Socket 没有连接到远程主机，那么也说明连接已经断开。

#### **客户端**

* **SendDataToServer**方法****：该方法接受一个字符串消息作为参数，将消息转换为字节数组并通过 \_clientSocket 发送给服务器。如果发送过程中出现异常，会输出错误信息。
* ****线程安全****：使用 \_locker 对 \_isConnected 标志和 \_clientSocket 的访问进行加锁，确保多线程操作时数据的一致性。
* ****连接超时****：在 ConnectToServer 方法中，使用 ManualResetEvent 和 WaitOne 方法设置连接超时时间为 5 秒。
* ****连接有效检查****：ReceiveMessages 方法在接收消息时检查连接状态，若接收字节数为 0 或发生 SocketException，则认为连接断开。
* ****断线重连****：CheckAndReconnect 方法每隔 5 秒检查一次连接状态，若连接断开则尝试重新连接服务器。

##### 异步连接：\_clientSocket.BeginConnect(\_serverEndPoint, new AsyncCallback(ConnectCallback), \_clientSocket)

在 C# 中，\_clientSocket.BeginConnect(\_serverEndPoint, new AsyncCallback(ConnectCallback), \_clientSocket);

这行代码用于发起一个异步连接操作，使得客户端能够以非阻塞的方式尝试连接到服务器。下面为你详细解释这行代码的各个部分。

##### 整体功能

这行代码的主要目的是让客户端的 Socket 对象 \_clientSocket 异步地连接到指定的服务器端点 \_serverEndPoint。异步连接意味着在连接操作进行的同时，程序可以继续执行其他任务，而不需要等待连接完成，从而提高程序的性能和响应能力。

##### **代码详细解释**

###### **\_clientSocket**

\_clientSocket 是一个 Socket 类的实例，代表客户端的套接字。在进行网络通信时，客户端需要通过这个套接字与服务器进行连接和数据交互。Socket 类提供了一系列方法和属性，用于实现网络通信的各种功能，如连接、发送和接收数据等。

###### **BeginConnect 方法**

BeginConnect 是 Socket 类的一个方法，用于发起一个异步连接操作。它的完整签名如下：

收起

csharp

public IAsyncResult BeginConnect(

EndPoint remoteEP,

AsyncCallback callback,

object state);

* ****参数解释****：
  + **remoteEP**：表示要连接的远程端点，在这行代码中是 \_serverEndPoint。\_serverEndPoint 是一个 IPEndPoint 类的实例，它包含了服务器的 IP 地址和端口号信息，用于指定客户端要连接的目标服务器。
  + **callback**：是一个 AsyncCallback 委托类型的参数，用于指定在连接操作完成时要调用的回调方法。在这行代码中，使用 new AsyncCallback(ConnectCallback) 创建了一个 AsyncCallback 委托实例，指向 ConnectCallback 方法。当连接操作完成时，系统会自动调用 ConnectCallback 方法，并将连接操作的结果传递给该方法。
  + **state**：是一个 object 类型的参数，用于传递与异步操作相关的状态信息。在这行代码中，将 \_clientSocket 作为状态信息传递给 BeginConnect 方法，这样在回调方法 ConnectCallback 中就可以获取到客户端的 Socket 对象，从而进行后续的操作。

###### **ConnectCallback 方法**

ConnectCallback 是一个自定义的方法，用于处理连接操作完成后的回调逻辑。它的签名通常如下：

收起

csharp

private void ConnectCallback(IAsyncResult ar){

// 处理连接操作完成后的逻辑}

在这个方法中，IAsyncResult 类型的参数 ar 包含了异步操作的结果信息，通过调用 EndConnect 方法可以获取连接操作的最终状态。例如：

收起

csharp

private void ConnectCallback(IAsyncResult ar){

try

{

Socket clientSocket = (Socket)ar.AsyncState;

clientSocket.EndConnect(ar);

// 连接成功，进行后续操作

Console.WriteLine("Connected to server.");

}

catch (Exception ex)

{

// 连接失败，处理异常

Console.WriteLine("Error connecting to server: " + ex.Message);

}}

###### **示例代码**

以下是一个完整的客户端示例，展示了如何使用 BeginConnect 方法进行异步连接：

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Text;

class Client{

private static Socket \_clientSocket;

private static IPEndPoint \_serverEndPoint;

private static ManualResetEvent \_connectDone = new ManualResetEvent(false);

static void Main()

{

\_serverEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8888);

\_clientSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

// 发起异步连接

\_clientSocket.BeginConnect(\_serverEndPoint, new AsyncCallback(ConnectCallback), \_clientSocket);

// 等待连接完成

\_connectDone.WaitOne();

// 连接完成后，可以进行数据发送和接收操作

try

{

string message = "Hello, server!";

byte[] data = Encoding.ASCII.GetBytes(message);

\_clientSocket.Send(data);

byte[] buffer = new byte[1024];

int bytesRead = \_clientSocket.Receive(buffer);

string response = Encoding.ASCII.GetString(buffer, 0, bytesRead);

Console.WriteLine("Received from server: " + response);

}

catch (Exception ex)

{

Console.WriteLine("Error: " + ex.Message);

}

finally

{

\_clientSocket.Close();

}

}

private static void ConnectCallback(IAsyncResult ar)

{

try

{

Socket clientSocket = (Socket)ar.AsyncState;

clientSocket.EndConnect(ar);

Console.WriteLine("Connected to server.");

\_connectDone.Set();

}

catch (Exception ex)

{

Console.WriteLine("Error connecting to server: " + ex.Message);

\_connectDone.Set();

}

}}

在这个示例中，客户端通过 BeginConnect 方法异步地连接到服务器，在连接完成后，调用 ConnectCallback 方法处理连接结果。同时，使用 ManualResetEvent 来等待连接操作完成，确保在连接成功后再进行数据发送和接收操作。

综上所述，\_clientSocket.BeginConnect(\_serverEndPoint, new AsyncCallback(ConnectCallback), \_clientSocket); 这行代码是实现客户端异步连接服务器的关键步骤，通过这种方式可以提高程序的性能和响应能力。

### 3、C#字符编码：套接字服务端和客户端在同一台电脑，都使用ASCII收发数据，但是发送和接收汉字，结果都是问号？

《注解：已验证，UTF8可以收发汉字》

当 C# 套接字服务端和客户端在同一台电脑，且都使用 ASCII 编码来收发汉字时出现问号，这是因为 ASCII 编码只支持 128 个字符，主要包含英文字母、数字和一些常见符号，并不包含汉字字符集，所以无法正确编码和解码汉字，就会显示为问号。

要解决这个问题，需要使用支持汉字编码的字符集，如 UTF - 8、GB2312、GBK 等，下面为你提供使用 UTF - 8 编码的示例代码。

#### 客户端代码

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Text;

class Client{

static void Main()

{

// 创建一个 TCP/IP 套接字

Socket clientSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

// 服务器的 IP 地址和端口号

IPEndPoint serverEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8888);

try

{

// 连接到服务器

clientSocket.Connect(serverEndPoint);

// 要发送的包含汉字的消息

string message = "你好，服务器！";

// 使用 UTF-8 编码将消息转换为字节数组

byte[] data = Encoding.UTF8.GetBytes(message);

// 发送数据到服务器

clientSocket.Send(data);

// 接收服务器的响应

byte[] buffer = new byte[1024];

int bytesRead = clientSocket.Receive(buffer);

// 使用 UTF-8 编码将接收到的字节数组转换为字符串

string response = Encoding.UTF8.GetString(buffer, 0, bytesRead);

Console.WriteLine("服务器响应: " + response);

// 关闭套接字

clientSocket.Close();

}

catch (Exception ex)

{

Console.WriteLine("发生错误: " + ex.Message);

}

}}

#### 服务端代码

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Text;

class Server{

static void Main()

{

// 创建一个 TCP/IP 套接字

Socket serverSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

// 绑定到本地地址和端口

IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, 8888);

serverSocket.Bind(localEndPoint);

// 开始监听，最大连接数为 10

serverSocket.Listen(10);

Console.WriteLine("服务器已启动，正在监听端口 8888...");

while (true)

{

// 接受客户端连接

Socket clientSocket = serverSocket.Accept();

// 接收客户端发送的数据

byte[] buffer = new byte[1024];

int bytesRead = clientSocket.Receive(buffer);

// 使用 UTF-8 编码将字节数组转换为字符串

string receivedMessage = Encoding.UTF8.GetString(buffer, 0, bytesRead);

Console.WriteLine("接收到客户端消息: " + receivedMessage);

// 给客户端发送响应

string responseMessage = "已收到你的消息: " + receivedMessage;

byte[] responseData = Encoding.UTF8.GetBytes(responseMessage);

clientSocket.Send(responseData);

// 关闭客户端套接字

clientSocket.Close();

}

}}

#### 代码解释

##### 客户端部分

* ****编码发送****：使用 Encoding.UTF8.GetBytes 方法将包含汉字的字符串 message 编码为 UTF - 8 字节数组 data，然后通过 clientSocket.Send(data) 方法将字节数组发送给服务器。
* ****解码接收****：使用 clientSocket.Receive(buffer) 接收服务器的响应字节数组，再使用 Encoding.UTF8.GetString(buffer, 0, bytesRead) 方法将字节数组解码为 UTF - 8 字符串 response。

##### 服务端部分

* ****解码接收****：使用 clientSocket.Receive(buffer) 接收客户端发送的字节数组，然后使用 Encoding.UTF8.GetString(buffer, 0, bytesRead) 方法将字节数组解码为 UTF - 8 字符串 receivedMessage。
* ****编码发送****：将响应消息 responseMessage 使用 Encoding.UTF8.GetBytes(responseMessage) 方法编码为 UTF - 8 字节数组 responseData，再通过 clientSocket.Send(responseData) 发送给客户端。

通过使用 UTF - 8 编码，服务端和客户端就能够正确地发送和接收包含汉字的消息了。

### 4、Socket通讯，如何判断数据接收完成？

在 C# 服务器开发中，判断上一个数据接收完成并避免数据重叠是非常重要的，以下为你介绍几种常见的方法及示例代码。

#### 1. 固定长度协议

##### 原理

规定每个数据包的长度是固定的，服务器按照固定长度来接收数据，当接收到规定长度的数据时，就认为一个数据包接收完成。

##### 示例代码

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Text;

class FixedLengthServer{

private const int BufferSize = 10; // 固定数据包长度

private Socket serverSocket;

public void Start()

{

serverSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, 8888);

serverSocket.Bind(localEndPoint);

serverSocket.Listen(10);

Console.WriteLine("服务器已启动，等待客户端连接...");

Socket clientSocket = serverSocket.Accept();

Console.WriteLine("客户端已连接");

byte[] buffer = new byte[BufferSize];

while (true)

{

int totalBytesRead = 0;

while (totalBytesRead < BufferSize)

{

int bytesRead = clientSocket.Receive(buffer, totalBytesRead, BufferSize - totalBytesRead, SocketFlags.None);

if (bytesRead == 0)

{

// 客户端断开连接

Console.WriteLine("客户端已断开连接");

break;

}

totalBytesRead += bytesRead;

}

if (totalBytesRead == BufferSize)

{

string data = Encoding.UTF8.GetString(buffer);

Console.WriteLine($"接收到完整数据包: {data}");

}

}

}}

class Program{

static void Main()

{

FixedLengthServer server = new FixedLengthServer();

server.Start();

}}

#### 2. 分隔符协议

##### 原理

在每个数据包的末尾添加一个特定的分隔符，服务器在接收数据时，不断查找分隔符，当找到分隔符时，就认为一个数据包接收完成。

#### **示例代码**

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Text;

class DelimiterServer{

private const char Delimiter = '\n'; // 分隔符

private Socket serverSocket;

public void Start()

{

serverSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, 8888);

serverSocket.Bind(localEndPoint);

serverSocket.Listen(10);

Console.WriteLine("服务器已启动，等待客户端连接...");

Socket clientSocket = serverSocket.Accept();

Console.WriteLine("客户端已连接");

byte[] buffer = new byte[1024];

StringBuilder receivedData = new StringBuilder();

while (true)

{

int bytesRead = clientSocket.Receive(buffer);

if (bytesRead == 0)

{

// 客户端断开连接

Console.WriteLine("客户端已断开连接");

break;

}

string data = Encoding.UTF8.GetString(buffer, 0, bytesRead);

receivedData.Append(data);

int delimiterIndex;

while ((delimiterIndex = receivedData.ToString().IndexOf(Delimiter)) != -1)

{

string completePacket = receivedData.ToString(0, delimiterIndex);

Console.WriteLine($"接收到完整数据包: {completePacket}");

receivedData.Remove(0, delimiterIndex + 1);

}

}

}}

class Program{

static void Main()

{

DelimiterServer server = new DelimiterServer();

server.Start();

}}

#### 3. 消息头包含长度信息

##### 原理

在每个数据包的头部添加一个字段，用于表示该数据包的长度。服务器先接收头部信息，解析出数据包的长度，然后根据这个长度继续接收数据，直到接收到完整的数据包。

#### **示例代码**

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Text;

class HeaderLengthServer{

private const int HeaderSize = 4; // 头部长度（用于存储数据包长度信息）

private Socket serverSocket;

public void Start()

{

serverSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, 8888);

serverSocket.Bind(localEndPoint);

serverSocket.Listen(10);

Console.WriteLine("服务器已启动，等待客户端连接...");

Socket clientSocket = serverSocket.Accept();

Console.WriteLine("客户端已连接");

while (true)

{

// 接收头部信息

byte[] headerBuffer = new byte[HeaderSize];

int totalHeaderBytesRead = 0;

while (totalHeaderBytesRead < HeaderSize)

{

int headerBytesRead = clientSocket.Receive(headerBuffer, totalHeaderBytesRead, HeaderSize - totalHeaderBytesRead, SocketFlags.None);

if (headerBytesRead == 0)

{

// 客户端断开连接

Console.WriteLine("客户端已断开连接");

break;

}

totalHeaderBytesRead += headerBytesRead;

}

if (totalHeaderBytesRead == HeaderSize)

{

int packetLength = BitConverter.ToInt32(headerBuffer, 0);

// 接收数据包内容

byte[] packetBuffer = new byte[packetLength];

int totalPacketBytesRead = 0;

while (totalPacketBytesRead < packetLength)

{

int packetBytesRead = clientSocket.Receive(packetBuffer, totalPacketBytesRead, packetLength - totalPacketBytesRead, SocketFlags.None);

if (packetBytesRead == 0)

{

// 客户端断开连接

Console.WriteLine("客户端已断开连接");

break;

}

totalPacketBytesRead += packetBytesRead;

}

if (totalPacketBytesRead == packetLength)

{

string data = Encoding.UTF8.GetString(packetBuffer);

Console.WriteLine($"接收到完整数据包: {data}");

}

}

}

}}

class Program{

static void Main()

{

HeaderLengthServer server = new HeaderLengthServer();

server.Start();

}}

#### 推荐方法：固定长度消息头

在实际开发中，消息头包含长度信息的方法更为灵活和通用，因为它可以处理不同长度的数据包。

而固定长度协议，在处理变长数据时会有局限性，分隔符协议在数据中可能包含分隔符时会出现问题。所以，消息头包含长度信息的方法是一个比较好的选择。

### 5、客户端掉线问题：基于超时时间判断客户端掉线的方法。

在网络编程中，基于超时时间判断客户端掉线是一种常见且有效的方法。下面将详细介绍该方法的原理、实现步骤，并给出 C# 语言的示例代码。

##### 原理

在正常的网络通信中，服务器和客户端之间会有数据的交互。基于超时时间判断客户端掉线的核心思想是：服务器在与客户端建立连接后，会为每个客户端设置一个计时器。当服务器在一定时间（即超时时间）内没有收到客户端发送的数据，或者没有向客户端成功发送数据时，就认为客户端已经掉线。

##### 实现步骤

#### **1. 建立连接**

服务器启动并监听指定端口，等待客户端的连接请求。当有客户端连接时，服务器为该客户端创建一个对应的连接对象，并启动超时计时器。

#### **2. 数据交互**

在服务器和客户端进行数据交互的过程中，每当服务器接收到客户端发送的数据，或者成功向客户端发送数据时，就重置超时计时器。

#### **3. 超时判断**

服务器会定期检查每个客户端的超时计时器。如果某个客户端的计时器超过了预设的超时时间，服务器就认为该客户端已经掉线，并进行相应的处理，如关闭连接、释放资源等。

##### C# 示例代码

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Threading;

class Server{

private const int TimeoutDuration = 10000; // 超时时间，单位：毫秒

private Socket serverSocket;

private Socket clientSocket;

private DateTime lastActivityTime;

public void Start()

{

serverSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, 8888);

serverSocket.Bind(localEndPoint);

serverSocket.Listen(10);

Console.WriteLine("服务器已启动，等待客户端连接...");

clientSocket = serverSocket.Accept();

Console.WriteLine("客户端已连接");

lastActivityTime = DateTime.Now;

// 启动数据接收线程

Thread receiveThread = new Thread(ReceiveData);

receiveThread.Start();

// 启动超时检查线程

Thread timeoutCheckThread = new Thread(CheckTimeout);

timeoutCheckThread.Start();

}

private void ReceiveData()

{

byte[] buffer = new byte[1024];

try

{

while (true)

{

int bytesRead = clientSocket.Receive(buffer);

if (bytesRead == 0)

{

// 客户端正常关闭连接

Console.WriteLine("客户端已正常断开连接");

break;

}

// 重置活动时间

lastActivityTime = DateTime.Now;

string data = System.Text.Encoding.UTF8.GetString(buffer, 0, bytesRead);

Console.WriteLine($"接收到客户端数据: {data}");

}

}

catch (SocketException ex)

{

// 客户端异常断开连接

Console.WriteLine($"客户端异常断开连接，异常信息: {ex.Message}");

}

}

private void CheckTimeout()

{

while (true)

{

Thread.Sleep(1000); // 每秒检查一次

TimeSpan elapsedTime = DateTime.Now - lastActivityTime;

if (elapsedTime.TotalMilliseconds > TimeoutDuration)

{

// 超时，认为客户端掉线

Console.WriteLine("客户端已超时掉线");

clientSocket.Close();

break;

}

}

}}

class Program{

static void Main()

{

Server server = new Server();

server.Start();

}}

##### 代码解释

1. **Start**方法****：启动服务器并监听指定端口，等待客户端连接。当有客户端连接时，记录当前时间作为最后活动时间，并启动数据接收线程和超时检查线程。
2. **ReceiveData**方法****：负责接收客户端发送的数据。每当接收到数据时，重置最后活动时间。如果接收数据长度为 0 或发生 SocketException 异常，认为客户端断开连接。
3. **CheckTimeout**方法****：每秒检查一次客户端的超时情况。计算当前时间与最后活动时间的差值，如果超过预设的超时时间，就认为客户端掉线，并关闭连接。

通过以上步骤和代码，你可以在 C# 中实现基于超时时间判断客户端掉线的功能。

### 6、掉线问题：如何在C#中实现心跳包机制？

在 C# 中实现心跳包机制，通常需要在服务器端和客户端分别进行相应的处理。下面为你详细介绍实现步骤并给出示例代码。

#### 心跳包机制原理

心跳包机制是一种用于检测网络连接是否正常的方法。服务器和客户端会定期向对方发送一个特定的数据包（即心跳包），如果在一定时间内没有收到对方的响应，就认为连接可能已经断开。

#### 实现步骤及示例代码

#### **服务器端代码**

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Threading;

class Server{

private const int HeartbeatInterval = 5000; // 心跳包发送间隔（毫秒）

private const int HeartbeatTimeout = 10000; // 心跳包超时时间（毫秒）

private Socket serverSocket;

private Socket clientSocket;

public void Start()

{

serverSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, 8888);

serverSocket.Bind(localEndPoint);

serverSocket.Listen(10);

Console.WriteLine("服务器已启动，等待客户端连接...");

clientSocket = serverSocket.Accept();

Console.WriteLine("客户端已连接");

// 启动心跳包发送线程

Thread heartbeatThread = new Thread(SendHeartbeat);

heartbeatThread.Start();

// 处理客户端数据接收

HandleClientData();

}

private void SendHeartbeat()

{

while (true)

{

try

{

// 发送心跳包

byte[] heartbeatData = new byte[] { 0x01 };

clientSocket.Send(heartbeatData);

// 等待客户端响应

byte[] buffer = new byte[1];

clientSocket.ReceiveTimeout = HeartbeatTimeout;

int bytesRead = clientSocket.Receive(buffer);

if (bytesRead == 0)

{

// 客户端关闭连接

Console.WriteLine("客户端已掉线");

break;

}

}

catch (SocketException ex)

{

if (ex.SocketErrorCode == SocketError.TimedOut)

{

// 心跳包超时，认为客户端掉线

Console.WriteLine("客户端已掉线");

break;

}

}

Thread.Sleep(HeartbeatInterval);

}

}

private void HandleClientData()

{

byte[] buffer = new byte[1024];

try

{

while (true)

{

int bytesRead = clientSocket.Receive(buffer);

if (bytesRead == 0)

{

// 客户端正常关闭连接

Console.WriteLine("客户端已掉线");

break;

}

}

}

catch (SocketException ex)

{

// 客户端异常掉线

Console.WriteLine($"客户端已掉线，异常信息：{ex.Message}");

}

}}

class Program{

static void Main()

{

Server server = new Server();

server.Start();

}}

#### **客户端代码**

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Threading;

class Client{

private const int HeartbeatInterval = 5000; // 心跳包发送间隔（毫秒）

private Socket clientSocket;

public void Connect()

{

clientSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

IPEndPoint remoteEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8888);

try

{

clientSocket.Connect(remoteEndPoint);

Console.WriteLine("已连接到服务器");

// 启动心跳包发送线程

Thread heartbeatThread = new Thread(SendHeartbeat);

heartbeatThread.Start();

// 处理服务器数据接收

HandleServerData();

}

catch (SocketException ex)

{

Console.WriteLine($"连接服务器失败，异常信息：{ex.Message}");

}

}

private void SendHeartbeat()

{

while (true)

{

try

{

// 发送心跳包

byte[] heartbeatData = new byte[] { 0x01 };

clientSocket.Send(heartbeatData);

// 等待服务器响应

byte[] buffer = new byte[1];

int bytesRead = clientSocket.Receive(buffer);

if (bytesRead == 0)

{

// 服务器关闭连接

Console.WriteLine("服务器已断开连接");

break;

}

}

catch (SocketException ex)

{

Console.WriteLine($"与服务器连接中断，异常信息：{ex.Message}");

break;

}

Thread.Sleep(HeartbeatInterval);

}

}

private void HandleServerData()

{

byte[] buffer = new byte[1024];

try

{

while (true)

{

int bytesRead = clientSocket.Receive(buffer);

if (bytesRead == 0)

{

// 服务器正常关闭连接

Console.WriteLine("服务器已断开连接");

break;

}

}

}

catch (SocketException ex)

{

// 服务器异常断开连接

Console.WriteLine($"与服务器连接中断，异常信息：{ex.Message}");

}

}}

class Program{

static void Main()

{

Client client = new Client();

client.Connect();

}}

#### 代码解释

****服务器端****：

* 1. Start 方法：启动服务器并等待客户端连接。连接成功后，启动心跳包发送线程和数据接收线程。
  2. SendHeartbeat 方法：定期向客户端发送心跳包，并等待客户端的响应。如果在指定时间内没有收到响应，认为客户端掉线。
  3. HandleClientData 方法：处理客户端发送的数据。如果接收数据长度为 0 或发生 SocketException 异常，认为客户端掉线。

****客户端****：

* 1. Connect 方法：连接到服务器。连接成功后，启动心跳包发送线程和数据接收线程。
  2. SendHeartbeat 方法：定期向服务器发送心跳包，并等待服务器的响应。如果发生 SocketException 异常，认为与服务器的连接中断。
  3. HandleServerData 方法：处理服务器发送的数据。如果接收数据长度为 0 或发生 SocketException 异常，认为服务器已断开连接。

通过上述代码，你可以在 C# 中实现一个简单的心跳包机制，用于检测服务器和客户端之间的连接状态。

### 7、C#客户端掉线问题：客户端掉线，重连服务器后，显示已经连接成功，服务端发送消息给客户端，突然报错：[System.Net](https://system.net/" \t "https://www.doubao.com/chat/_blank).Sockets.SocketException (10054): 远程主机强迫关闭了一个现有的连接。为什么？有哪些解决方案？

#### 问题原因分析

当 C# 客户端掉线后重连服务器，显示连接成功，但服务器发送消息时却抛出 System.Net.Sockets.SocketException (10054) 异常，提示 “远程主机强迫关闭了一个现有的连接”，通常有以下几种可能原因：

##### 1. 服务器端未正确处理新连接

服务器在客户端重连时，可能没有正确更新客户端的连接信息，仍然使用旧的 Socket 对象进行通信。当尝试向已关闭的 Socket 发送数据时，就会引发该异常。

【注解：我碰到的问题就是这个问题。

解决办法：管理好上线的客户端，如果客户端上线，要将它添加到客户端列表；为每一个客户端启动一个线程处理消息问题，在这里如果客户端掉线，将客户端从列表中删除，同时要关闭掉线的客户端。

】

##### 2. 客户端重连后状态不一致

客户端重连成功后，可能内部状态没有正确重置，导致在接收服务器消息时出现异常，从而主动关闭了连接。

##### 3. 网络不稳定

网络波动可能导致连接在表面上看起来已经恢复，但实际上连接已经损坏，服务器发送消息时就会失败。

##### 4. 资源未正确释放

在客户端掉线和重连过程中，可能存在资源未正确释放的情况，例如缓冲区未清空、线程未正确停止等，这可能会影响后续的通信。

如何在客户端和服务器端正确释放资源以避免连接问题？

在客户端和服务器端正确释放资源对于避免连接问题至关重要。下面分别介绍客户端和服务器端在不同场景下如何正确释放资源，并给出相应的 C# 代码示例。

#### 客户端资源释放

##### 1. 正常关闭连接

当客户端主动关闭连接时，需要确保 Socket 对象被正确关闭，同时停止相关的线程（如果有）。

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Threading;

class Client{

private Socket clientSocket;

private Thread receiveThread;

private bool isRunning = true;

public void Connect()

{

try

{

clientSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

IPEndPoint remoteEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8888);

clientSocket.Connect(remoteEndPoint);

Console.WriteLine("已连接到服务器");

// 启动数据接收线程

receiveThread = new Thread(ReceiveData);

receiveThread.Start();

}

catch (SocketException ex)

{

Console.WriteLine($"连接服务器失败: {ex.Message}");

}

}

private void ReceiveData()

{

byte[] buffer = new byte[1024];

try

{

while (isRunning)

{

int bytesRead = clientSocket.Receive(buffer);

if (bytesRead == 0)

{

// 服务器断开连接

Console.WriteLine("服务器已断开连接");

break;

}

string data = System.Text.Encoding.UTF8.GetString(buffer, 0, bytesRead);

Console.WriteLine($"接收到服务器消息: {data}");

}

}

catch (SocketException ex)

{

Console.WriteLine($"接收消息时出现异常: {ex.Message}");

}

finally

{

CloseConnection();

}

}

public void CloseConnection()

{

isRunning = false;

if (receiveThread != null && receiveThread.IsAlive)

{

receiveThread.Join(); // 等待线程结束

}

if (clientSocket != null && clientSocket.Connected)

{

clientSocket.Shutdown(SocketShutdown.Both); // 关闭读写操作

clientSocket.Close(); // 关闭 Socket

}

Console.WriteLine("客户端连接已关闭");

}}

class Program{

static void Main()

{

Client client = new Client();

client.Connect();

// 模拟一段时间后关闭连接

Thread.Sleep(5000);

client.CloseConnection();

}}

##### 2. 异常处理

在发生异常时，也需要确保资源被正确释放。在 ReceiveData 方法中，使用 try-catch-finally 块，无论是否发生异常，都会在 finally 块中调用 CloseConnection 方法来关闭连接和停止线程。

#### 服务器端资源释放

##### 1. 处理单个客户端连接

当服务器处理单个客户端连接时，需要在客户端断开连接或出现异常时，关闭对应的 Socket 对象。

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Threading;

class Server{

private Socket serverSocket;

private Socket clientSocket;

private Thread receiveThread;

private bool isRunning = true;

public void Start()

{

try

{

serverSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, 8888);

serverSocket.Bind(localEndPoint);

serverSocket.Listen(10);

Console.WriteLine("服务器已启动，等待客户端连接...");

clientSocket = serverSocket.Accept();

Console.WriteLine("客户端已连接");

// 启动数据接收线程

receiveThread = new Thread(ReceiveData);

receiveThread.Start();

}

catch (SocketException ex)

{

Console.WriteLine($"启动服务器失败: {ex.Message}");

}

}

private void ReceiveData()

{

byte[] buffer = new byte[1024];

try

{

while (isRunning)

{

int bytesRead = clientSocket.Receive(buffer);

if (bytesRead == 0)

{

// 客户端断开连接

Console.WriteLine("客户端已断开连接");

break;

}

string data = System.Text.Encoding.UTF8.GetString(buffer, 0, bytesRead);

Console.WriteLine($"接收到客户端消息: {data}");

}

}

catch (SocketException ex)

{

Console.WriteLine($"接收消息时出现异常: {ex.Message}");

}

finally

{

CloseConnection();

}

}

public void CloseConnection()

{

isRunning = false;

if (receiveThread != null && receiveThread.IsAlive)

{

receiveThread.Join(); // 等待线程结束

}

if (clientSocket != null && clientSocket.Connected)

{

clientSocket.Shutdown(SocketShutdown.Both); // 关闭读写操作

clientSocket.Close(); // 关闭 Socket

}

Console.WriteLine("客户端连接已关闭");

}}

class Program{

static void Main()

{

Server server = new Server();

server.Start();

}}

##### 2. 处理多个客户端连接

如果服务器需要处理多个客户端连接，需要为每个客户端连接创建独立的 Socket 对象和线程，并在客户端断开连接时，正确关闭对应的资源。

收起

csharp

using System;using System.Collections.Generic;using System.Net;using System.Net.Sockets;using System.Threading;

class Server{

private Socket serverSocket;

private List<Socket> clientSockets = new List<Socket>();

private List<Thread> clientThreads = new List<Thread>();

private bool isRunning = true;

public void Start()

{

try

{

serverSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, 8888);

serverSocket.Bind(localEndPoint);

serverSocket.Listen(10);

Console.WriteLine("服务器已启动，等待客户端连接...");

while (isRunning)

{

Socket clientSocket = serverSocket.Accept();

Console.WriteLine("客户端已连接");

clientSockets.Add(clientSocket);

Thread clientThread = new Thread(() => HandleClient(clientSocket));

clientThreads.Add(clientThread);

clientThread.Start();

}

}

catch (SocketException ex)

{

Console.WriteLine($"启动服务器失败: {ex.Message}");

}

}

private void HandleClient(Socket clientSocket)

{

byte[] buffer = new byte[1024];

try

{

while (isRunning)

{

int bytesRead = clientSocket.Receive(buffer);

if (bytesRead == 0)

{

// 客户端断开连接

Console.WriteLine("客户端已断开连接");

break;

}

string data = System.Text.Encoding.UTF8.GetString(buffer, 0, bytesRead);

Console.WriteLine($"接收到客户端消息: {data}");

}

}

catch (SocketException ex)

{

Console.WriteLine($"接收消息时出现异常: {ex.Message}");

}

finally

{

CloseClientConnection(clientSocket);

}

}

private void CloseClientConnection(Socket clientSocket)

{

int index = clientSockets.IndexOf(clientSocket);

if (index != -1)

{

clientSockets.RemoveAt(index);

Thread clientThread = clientThreads[index];

clientThreads.RemoveAt(index);

if (clientThread != null && clientThread.IsAlive)

{

clientThread.Join(); // 等待线程结束

}

if (clientSocket != null && clientSocket.Connected)

{

clientSocket.Shutdown(SocketShutdown.Both); // 关闭读写操作

clientSocket.Close(); // 关闭 Socket

}

Console.WriteLine("客户端连接已关闭");

}

}

public void Stop()

{

isRunning = false;

foreach (Socket clientSocket in clientSockets)

{

if (clientSocket != null && clientSocket.Connected)

{

clientSocket.Shutdown(SocketShutdown.Both);

clientSocket.Close();

}

}

clientSockets.Clear();

foreach (Thread clientThread in clientThreads)

{

if (clientThread != null && clientThread.IsAlive)

{

clientThread.Join();

}

}

clientThreads.Clear();

if (serverSocket != null && serverSocket.Connected)

{

serverSocket.Shutdown(SocketShutdown.Both);

serverSocket.Close();

}

Console.WriteLine("服务器已停止");

}}

class Program{

static void Main()

{

Server server = new Server();

server.Start();

// 模拟一段时间后停止服务器

Thread.Sleep(10000);

server.Stop();

}}

##### 总结

* ****客户端****：在正常关闭连接或发生异常时，使用 Shutdown 方法关闭读写操作，再使用 Close 方法关闭 Socket 对象，并停止相关线程。
* ****服务器端****：对于单个客户端连接，处理方式与客户端类似；对于多个客户端连接，需要为每个客户端连接创建独立的资源，并在客户端断开连接时正确释放对应的资源。在服务器停止时，需要关闭所有客户端连接和服务器 Socket 对象，并停止所有相关线程。

### 8、如何优化资源释放的性能？

在网络编程中，优化资源释放的性能可以从多个方面入手，包括合理的线程管理、减少资源占用、避免不必要的锁竞争等。以下是针对客户端和服务器端资源释放性能优化的详细建议及示例代码。

#### 1. 线程管理优化

##### 减少线程创建和销毁开销

频繁地创建和销毁线程会带来较大的性能开销。可以使用线程池来管理线程，避免不必要的线程创建和销毁。

****示例代码（服务器端）****

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Threading;

class Server{

private Socket serverSocket;

private bool isRunning = true;

public void Start()

{

try

{

serverSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, 8888);

serverSocket.Bind(localEndPoint);

serverSocket.Listen(10);

Console.WriteLine("服务器已启动，等待客户端连接...");

while (isRunning)

{

Socket clientSocket = serverSocket.Accept();

Console.WriteLine("客户端已连接");

// 使用线程池处理客户端连接

ThreadPool.QueueUserWorkItem(HandleClient, clientSocket);

}

}

catch (SocketException ex)

{

Console.WriteLine($"启动服务器失败: {ex.Message}");

}

}

private void HandleClient(object state)

{

Socket clientSocket = (Socket)state;

byte[] buffer = new byte[1024];

try

{

while (isRunning)

{

int bytesRead = clientSocket.Receive(buffer);

if (bytesRead == 0)

{

// 客户端断开连接

Console.WriteLine("客户端已断开连接");

break;

}

string data = System.Text.Encoding.UTF8.GetString(buffer, 0, bytesRead);

Console.WriteLine($"接收到客户端消息: {data}");

}

}

catch (SocketException ex)

{

Console.WriteLine($"接收消息时出现异常: {ex.Message}");

}

finally

{

CloseClientConnection(clientSocket);

}

}

private void CloseClientConnection(Socket clientSocket)

{

if (clientSocket != null && clientSocket.Connected)

{

clientSocket.Shutdown(SocketShutdown.Both);

clientSocket.Close();

}

Console.WriteLine("客户端连接已关闭");

}

public void Stop()

{

isRunning = false;

if (serverSocket != null && serverSocket.Connected)

{

serverSocket.Shutdown(SocketShutdown.Both);

serverSocket.Close();

}

Console.WriteLine("服务器已停止");

}}

class Program{

static void Main()

{

Server server = new Server();

server.Start();

// 模拟一段时间后停止服务器

Thread.Sleep(10000);

server.Stop();

}}

##### 线程提前退出机制

在资源释放时，尽量让线程提前退出，避免不必要的等待。可以使用标志位来控制线程的执行。

#### 2. 资源占用优化

##### 及时释放缓冲区

在接收和发送数据时，使用的缓冲区应该及时释放，避免占用过多的内存。

****示例代码（客户端）****

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Threading;

class Client{

private Socket clientSocket;

private bool isRunning = true;

public void Connect()

{

try

{

clientSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

IPEndPoint remoteEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8888);

clientSocket.Connect(remoteEndPoint);

Console.WriteLine("已连接到服务器");

// 启动数据接收线程

Thread receiveThread = new Thread(ReceiveData);

receiveThread.Start();

}

catch (SocketException ex)

{

Console.WriteLine($"连接服务器失败: {ex.Message}");

}

}

private void ReceiveData()

{

while (isRunning)

{

byte[] buffer = new byte[1024];

try

{

int bytesRead = clientSocket.Receive(buffer);

if (bytesRead == 0)

{

// 服务器断开连接

Console.WriteLine("服务器已断开连接");

break;

}

string data = System.Text.Encoding.UTF8.GetString(buffer, 0, bytesRead);

Console.WriteLine($"接收到服务器消息: {data}");

}

catch (SocketException ex)

{

Console.WriteLine($"接收消息时出现异常: {ex.Message}");

break;

}

finally

{

// 及时释放缓冲区

buffer = null;

GC.Collect();

}

}

CloseConnection();

}

public void CloseConnection()

{

isRunning = false;

if (clientSocket != null && clientSocket.Connected)

{

clientSocket.Shutdown(SocketShutdown.Both);

clientSocket.Close();

}

Console.WriteLine("客户端连接已关闭");

}}

class Program{

static void Main()

{

Client client = new Client();

client.Connect();

// 模拟一段时间后关闭连接

Thread.Sleep(5000);

client.CloseConnection();

}}

##### 减少对象创建

尽量减少不必要的对象创建，例如在循环中避免重复创建相同类型的对象。

#### 3. 避免锁竞争

##### 减少锁的使用

在资源释放过程中，尽量减少锁的使用，避免锁竞争带来的性能开销。如果需要使用锁，尽量缩小锁的范围。

##### 使用无锁数据结构

可以使用无锁数据结构来替代传统的有锁数据结构，提高并发性能。

#### 4. 异步操作优化

##### 使用异步 I/O

在网络编程中，使用异步 I/O 可以避免线程阻塞，提高资源利用率。

****示例代码（服务器端异步接收数据）****

收起

csharp

using System;using System.Net;using System.Net.Sockets;

class Server{

private Socket serverSocket;

private bool isRunning = true;

public void Start()

{

try

{

serverSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, 8888);

serverSocket.Bind(localEndPoint);

serverSocket.Listen(10);

Console.WriteLine("服务器已启动，等待客户端连接...");

StartAccept();

}

catch (SocketException ex)

{

Console.WriteLine($"启动服务器失败: {ex.Message}");

}

}

private void StartAccept()

{

serverSocket.BeginAccept(AcceptCallback, null);

}

private void AcceptCallback(IAsyncResult ar)

{

if (!isRunning) return;

Socket clientSocket = serverSocket.EndAccept(ar);

Console.WriteLine("客户端已连接");

StartReceive(clientSocket);

StartAccept();

}

private void StartReceive(Socket clientSocket)

{

byte[] buffer = new byte[1024];

SocketAsyncEventArgs args = new SocketAsyncEventArgs();

args.AcceptSocket = clientSocket;

args.SetBuffer(buffer, 0, buffer.Length);

args.Completed += ReceiveCallback;

clientSocket.ReceiveAsync(args);

}

private void ReceiveCallback(object sender, SocketAsyncEventArgs e)

{

if (e.BytesTransferred == 0)

{

// 客户端断开连接

Console.WriteLine("客户端已断开连接");

CloseClientConnection(e.AcceptSocket);

return;

}

string data = System.Text.Encoding.UTF8.GetString(e.Buffer, 0, e.BytesTransferred);

Console.WriteLine($"接收到客户端消息: {data}");

StartReceive(e.AcceptSocket);

}

private void CloseClientConnection(Socket clientSocket)

{

if (clientSocket != null && clientSocket.Connected)

{

clientSocket.Shutdown(SocketShutdown.Both);

clientSocket.Close();

}

Console.WriteLine("客户端连接已关闭");

}

public void Stop()

{

isRunning = false;

if (serverSocket != null && serverSocket.Connected)

{

serverSocket.Shutdown(SocketShutdown.Both);

serverSocket.Close();

}

Console.WriteLine("服务器已停止");

}}

class Program{

static void Main()

{

Server server = new Server();

server.Start();

// 模拟一段时间后停止服务器

Console.ReadLine();

server.Stop();

}}

通过以上优化措施，可以显著提高资源释放的性能，减少系统开销，提升程序的整体性能。

### Socket、TcpClient 和 TcpListener

* Socket：提供了底层的网络编程接口，灵活性高，适用于需要对网络连接进行精细控制的场景。
* TcpClient：简化了 TCP 客户端的开发，适合快速开发简单的 TCP 客户端程序。
* TcpListener：简化了 TCP 服务器的开发，适合快速开发简单的 TCP 服务器程序。

#### 1、简介

#### Socket

Socket 是一个底层的网络编程类，它提供了对网络通信的基本操作，支持多种网络协议，包括 TCP 和 UDP。通过 Socket 类，可以直接控制网络连接的建立、数据的发送和接收等操作，具有较高的灵活性，但使用起来相对复杂。

#### TcpClient

TcpClient 是一个高级的网络编程类，它封装了 Socket 的一些操作，简化了 TCP 客户端的开发。使用 TcpClient 可以更方便地建立与服务器的连接，并进行数据的发送和接收。

#### TcpListener

TcpListener 是一个高级的网络编程类，用于创建 TCP 服务器。它封装了 Socket 的监听和接受客户端连接的操作，简化了 TCP 服务器的开发。

#### 2、TcpClient 和 TcpListener 实现 TCP 通信的具体步骤是什么？

在 C# 中，使用 TcpClient 和 TcpListener 可以方便地实现 TCP 通信。TcpListener 用于创建 TCP 服务器，监听客户端的连接请求；TcpClient 用于创建 TCP 客户端，连接到服务器并进行数据交互。以下是实现 TCP 通信的具体步骤及示例代码。

##### **服务器端（使用 TcpListener）**

###### 步骤

1. ****创建**TcpListener**实例****：指定服务器要监听的 IP 地址和端口号。
2. ****启动监听****：调用 Start 方法开始监听客户端的连接请求。
3. ****接受客户端连接****：使用 AcceptTcpClient 或 AcceptTcpClientAsync 方法接受客户端的连接，返回一个 TcpClient 对象。
4. ****获取网络流****：通过 TcpClient 的 GetStream 方法获取用于数据传输的 NetworkStream。
5. ****数据读写****：使用 NetworkStream 进行数据的读取和写入操作。
6. ****关闭连接****：通信结束后，关闭 TcpClient 和 TcpListener。

###### 示例代码

收起

csharp

using System;using System.IO;using System.Net;using System.Net.Sockets;using System.Text;

class TcpServer{

static void Main()

{

try

{

// 步骤 1：创建 TcpListener 实例，监听本地所有 IP 地址的 8888 端口

TcpListener listener = new TcpListener(IPAddress.Any, 8888);

// 步骤 2：启动监听

listener.Start();

Console.WriteLine("服务器已启动，等待客户端连接...");

while (true)

{

// 步骤 3：接受客户端连接

TcpClient client = listener.AcceptTcpClient();

Console.WriteLine("客户端已连接");

// 步骤 4：获取网络流

NetworkStream stream = client.GetStream();

// 步骤 5：数据读写

// 读取客户端发送的数据

byte[] buffer = new byte[1024];

int bytesRead = stream.Read(buffer, 0, buffer.Length);

string message = Encoding.UTF8.GetString(buffer, 0, bytesRead);

Console.WriteLine($"收到客户端消息: {message}");

// 向客户端发送响应消息

string response = "消息已收到";

byte[] responseData = Encoding.UTF8.GetBytes(response);

stream.Write(responseData, 0, responseData.Length);

// 步骤 6：关闭连接

stream.Close();

client.Close();

}

}

catch (Exception ex)

{

Console.WriteLine($"发生错误: {ex.Message}");

}

}}

##### **客户端（使用 TcpClient）**

###### 步骤

1. ****创建**TcpClient**实例****：用于连接到服务器。
2. ****连接到服务器****：调用 Connect 或 ConnectAsync 方法连接到指定的服务器 IP 地址和端口号。
3. ****获取网络流****：通过 TcpClient 的 GetStream 方法获取用于数据传输的 NetworkStream。
4. ****数据读写****：使用 NetworkStream 进行数据的读取和写入操作。
5. ****关闭连接****：通信结束后，关闭 TcpClient。

###### **示例代码**

收起

csharp

using System;using System.IO;using System.Net.Sockets;using System.Text;

class TcpClientExample{

static void Main()

{

try

{

// 步骤 1：创建 TcpClient 实例

TcpClient client = new TcpClient();

// 步骤 2：连接到服务器

client.Connect("127.0.0.1", 8888);

Console.WriteLine("已连接到服务器");

// 步骤 3：获取网络流

NetworkStream stream = client.GetStream();

// 步骤 4：数据读写

// 向服务器发送消息

string message = "Hello, Server!";

byte[] messageData = Encoding.UTF8.GetBytes(message);

stream.Write(messageData, 0, messageData.Length);

// 读取服务器的响应消息

byte[] buffer = new byte[1024];

int bytesRead = stream.Read(buffer, 0, buffer.Length);

string response = Encoding.UTF8.GetString(buffer, 0, bytesRead);

Console.WriteLine($"收到服务器响应: {response}");

// 步骤 5：关闭连接

stream.Close();

client.Close();

}

catch (Exception ex)

{

Console.WriteLine($"发生错误: {ex.Message}");

}

}}

###### 注意事项

* 上述示例代码是同步方式实现的 TCP 通信，在实际应用中，为了避免阻塞主线程，建议使用异步方法（如 AcceptTcpClientAsync、ConnectAsync 等）。
* 要确保服务器和客户端使用相同的 IP 地址和端口号进行通信。
* 在处理数据读写时，需要考虑数据的编码和解码，示例中使用的是 UTF-8 编码。

### 10、客户端和服务器端如何确保通信的可靠性？

在使用 TcpClient 和 TcpListener 实现的 C# TCP 通信中，为确保客户端和服务器端通信的可靠性，可以从连接管理、数据传输、错误处理和心跳机制等多个方面进行考虑和处理，以下是详细介绍及示例代码。

#### 1. 连接管理

* ****服务器端监听****：服务器应持续监听客户端的连接请求，避免因意外中断而无法接受新连接。
* ****客户端重连****：客户端在连接失败或断开时，应尝试重新连接服务器。

#### **服务器端示例**

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Threading;

class TcpServer{

private TcpListener listener;

private bool isRunning = true;

public TcpServer(int port)

{

listener = new TcpListener(IPAddress.Any, port);

}

public void Start()

{

listener.Start();

Console.WriteLine("服务器已启动，等待客户端连接...");

while (isRunning)

{

try

{

TcpClient client = listener.AcceptTcpClient();

Console.WriteLine("客户端已连接");

// 处理客户端连接的逻辑

Thread clientThread = new Thread(() => HandleClient(client));

clientThread.Start();

}

catch (Exception ex)

{

Console.WriteLine($"接受客户端连接出错: {ex.Message}");

}

}

}

private void HandleClient(TcpClient client)

{

try

{

// 处理客户端通信的逻辑

}

catch (Exception ex)

{

Console.WriteLine($"处理客户端通信出错: {ex.Message}");

}

finally

{

client.Close();

}

}

public void Stop()

{

isRunning = false;

listener.Stop();

}}

#### **客户端示例**

收起

csharp

using System;using System.Net.Sockets;using System.Threading;

class TcpClientExample{

private TcpClient client;

private bool isRunning = true;

private string serverIp;

private int serverPort;

public TcpClientExample(string ip, int port)

{

serverIp = ip;

serverPort = port;

client = new TcpClient();

}

public void Connect()

{

while (isRunning)

{

try

{

client.Connect(serverIp, serverPort);

Console.WriteLine("已连接到服务器");

// 处理与服务器通信的逻辑

break;

}

catch (Exception ex)

{

Console.WriteLine($"连接服务器失败: {ex.Message}");

Thread.Sleep(5000); // 等待 5 秒后重试

}

}

}

public void Disconnect()

{

isRunning = false;

client.Close();

}}

#### 2. 数据传输

* ****数据完整性****：在发送数据时，可以添加数据长度、校验和等信息，接收方根据这些信息验证数据的完整性。
* ****数据分段****：对于大数据量的传输，将数据分段发送，接收方按顺序拼接数据。

#### **示例代码**

收起

csharp

// 发送数据时添加数据长度信息public static void SendData(NetworkStream stream, byte[] data){

byte[] lengthBytes = BitConverter.GetBytes(data.Length);

stream.Write(lengthBytes, 0, lengthBytes.Length);

stream.Write(data, 0, data.Length);}

// 接收数据时根据长度信息读取完整数据public static byte[] ReceiveData(NetworkStream stream){

byte[] lengthBytes = new byte[4];

stream.Read(lengthBytes, 0, lengthBytes.Length);

int dataLength = BitConverter.ToInt32(lengthBytes, 0);

byte[] data = new byte[dataLength];

int totalBytesRead = 0;

while (totalBytesRead < dataLength)

{

int bytesRead = stream.Read(data, totalBytesRead, dataLength - totalBytesRead);

totalBytesRead += bytesRead;

}

return data;}

#### 3. 错误处理

* ****异常捕获****：在数据发送和接收过程中，捕获可能出现的异常，并进行相应的处理，如重新连接、记录日志等。
* ****超时处理****：设置数据发送和接收的超时时间，避免长时间等待。

#### **示例代码**

收起

csharp

try{

// 发送数据

byte[] data = Encoding.UTF8.GetBytes("Hello, Server!");

stream.WriteTimeout = 5000; // 设置发送超时时间为 5 秒

stream.Write(data, 0, data.Length);

// 接收数据

byte[] buffer = new byte[1024];

stream.ReadTimeout = 5000; // 设置接收超时时间为 5 秒

int bytesRead = stream.Read(buffer, 0, buffer.Length);

string response = Encoding.UTF8.GetString(buffer, 0, bytesRead);

Console.WriteLine($"收到服务器响应: {response}");}catch (Exception ex){

Console.WriteLine($"数据传输出错: {ex.Message}");

// 进行错误处理，如重新连接}

#### 4. 心跳机制

* ****发送心跳包****：客户端和服务器定期互相发送心跳包，以检测连接是否正常。
* ****超时判断****：如果在一定时间内没有收到心跳包，则认为连接断开。

#### **服务器端心跳检测示例**

收起

csharp

private void HandleClient(TcpClient client){

NetworkStream stream = client.GetStream();

DateTime lastHeartbeatTime = DateTime.Now;

while (true)

{

try

{

if (stream.DataAvailable)

{

byte[] buffer = new byte[1024];

int bytesRead = stream.Read(buffer, 0, buffer.Length);

string message = Encoding.UTF8.GetString(buffer, 0, bytesRead);

if (message == "Heartbeat")

{

lastHeartbeatTime = DateTime.Now;

}

else

{

// 处理其他业务数据

}

}

if ((DateTime.Now - lastHeartbeatTime).TotalSeconds > 10)

{

Console.WriteLine("客户端连接超时，断开连接");

break;

}

}

catch (Exception ex)

{

Console.WriteLine($"处理客户端通信出错: {ex.Message}");

break;

}

}

client.Close();}

#### **客户端心跳发送示例**

收起

csharp

private void SendHeartbeat(){

while (isRunning)

{

try

{

byte[] heartbeatData = Encoding.UTF8.GetBytes("Heartbeat");

stream.Write(heartbeatData, 0, heartbeatData.Length);

Thread.Sleep(5000); // 每 5 秒发送一次心跳包

}

catch (Exception ex)

{

Console.WriteLine($"发送心跳包出错: {ex.Message}");

// 进行错误处理，如重新连接

}

}}

通过以上几个方面的处理，可以有效提高客户端和服务器端通信的可靠性。

### 11、如何处理网络连接中断的情况？

在使用 TcpClient 和 TcpListener 进行 C# TCP 通信时，网络连接中断是常见的问题，需要妥善处理以确保程序的稳定性和可靠性。下面从服务器端和客户端两个角度，介绍处理网络连接中断情况的方法和示例代码。

#### 服务器端处理网络连接中断

#### **1. 捕获异常**

在接受客户端连接和处理客户端通信的过程中，捕获可能出现的异常，如 SocketException，当捕获到异常时，认为连接中断并进行相应处理。

#### **2. 定期检查连接状态**

可以使用心跳机制定期检查客户端的连接状态，如果一段时间内没有收到客户端的心跳包，则认为连接中断。

#### **示例代码**

收起

csharp

using System;using System.Net;using System.Net.Sockets;using System.Text;using System.Threading;

class TcpServer{

private TcpListener listener;

private bool isRunning = true;

public TcpServer(int port)

{

listener = new TcpListener(IPAddress.Any, port);

}

public void Start()

{

listener.Start();

Console.WriteLine("服务器已启动，等待客户端连接...");

while (isRunning)

{

try

{

TcpClient client = listener.AcceptTcpClient();

Console.WriteLine("客户端已连接");

// 为每个客户端创建一个线程处理通信

Thread clientThread = new Thread(() => HandleClient(client));

clientThread.Start();

}

catch (SocketException ex)

{

Console.WriteLine($"接受客户端连接时出现网络错误: {ex.Message}");

}

}

}

private void HandleClient(TcpClient client)

{

NetworkStream stream = client.GetStream();

DateTime lastHeartbeatTime = DateTime.Now;

try

{

while (true)

{

if (stream.DataAvailable)

{

byte[] buffer = new byte[1024];

int bytesRead = stream.Read(buffer, 0, buffer.Length);

string message = Encoding.UTF8.GetString(buffer, 0, bytesRead);

if (message == "Heartbeat")

{

// 收到心跳包，更新时间

lastHeartbeatTime = DateTime.Now;

}

else

{

// 处理其他业务数据

Console.WriteLine($"收到客户端消息: {message}");

}

}

// 检查是否长时间未收到心跳包

if ((DateTime.Now - lastHeartbeatTime).TotalSeconds > 10)

{

Console.WriteLine("客户端连接超时，断开连接");

break;

}

}

}

catch (SocketException ex)

{

Console.WriteLine($"处理客户端通信时出现网络错误: {ex.Message}");

}

finally

{

// 关闭客户端连接

stream.Close();

client.Close();

}

}

public void Stop()

{

isRunning = false;

listener.Stop();

}}

#### 客户端处理网络连接中断

#### **1. 捕获异常**

在连接服务器、发送和接收数据的过程中，捕获 SocketException 异常，当捕获到异常时，认为连接中断并尝试重新连接。

#### **2. 定期重连机制**

使用定时器或循环，在连接中断后定期尝试重新连接服务器。

#### **示例代码**

收起

csharp

using System;using System.Net.Sockets;using System.Text;using System.Threading;

class TcpClientExample{

private TcpClient client;

private bool isRunning = true;

private string serverIp;

private int serverPort;

private NetworkStream stream;

public TcpClientExample(string ip, int port)

{

serverIp = ip;

serverPort = port;

client = new TcpClient();

}

public void Connect()

{

while (isRunning)

{

try

{

client.Connect(serverIp, serverPort);

Console.WriteLine("已连接到服务器");

stream = client.GetStream();

// 启动心跳发送线程

Thread heartbeatThread = new Thread(SendHeartbeat);

heartbeatThread.Start();

// 处理数据接收

HandleDataReceiving();

}

catch (SocketException ex)

{

Console.WriteLine($"连接服务器失败: {ex.Message}");

// 等待一段时间后重试

Thread.Sleep(5000);

}

}

}

private void SendHeartbeat()

{

while (isRunning)

{

try

{

byte[] heartbeatData = Encoding.UTF8.GetBytes("Heartbeat");

stream.Write(heartbeatData, 0, heartbeatData.Length);

Thread.Sleep(5000); // 每 5 秒发送一次心跳包

}

catch (SocketException ex)

{

Console.WriteLine($"发送心跳包时出现网络错误: {ex.Message}");

// 尝试重新连接

Reconnect();

}

}

}

private void HandleDataReceiving()

{

try

{

while (isRunning)

{

if (stream.DataAvailable)

{

byte[] buffer = new byte[1024];

int bytesRead = stream.Read(buffer, 0, buffer.Length);

string message = Encoding.UTF8.GetString(buffer, 0, bytesRead);

Console.WriteLine($"收到服务器消息: {message}");

}

}

}

catch (SocketException ex)

{

Console.WriteLine($"接收数据时出现网络错误: {ex.Message}");

// 尝试重新连接

Reconnect();

}

}

private void Reconnect()

{

try

{

client.Close();

client = new TcpClient();

client.Connect(serverIp, serverPort);

Console.WriteLine("重新连接到服务器");

stream = client.GetStream();

}

catch (SocketException ex)

{

Console.WriteLine($"重新连接服务器失败: {ex.Message}");

// 等待一段时间后再次尝试

Thread.Sleep(5000);

Reconnect();

}

}

public void Disconnect()

{

isRunning = false;

if (stream != null)

{

stream.Close();

}

client.Close();

}}

#### 总结

* 服务器端通过捕获异常和心跳机制检测客户端连接状态，当连接中断时关闭客户端连接。
* 客户端通过捕获异常和定期重连机制，在连接中断时尝试重新连接服务器，确保通信的连续性。

# MySQL

### 目标设定

1. **具体性（Specific）**：在 60 天内系统掌握 MySQL 数据库从基础操作到高级应用的全部知识，能够独立完成复杂数据库的搭建、管理、优化，熟练编写高效的 SQL 查询语句，解决各类数据库相关问题。
2. **可衡量性（Measurable）**：每周通过在线测试平台进行知识点测验，正确率保持在 85% 以上；完成 8 - 10 个涵盖不同业务场景的实际数据库项目，包括但不限于企业资源管理系统、电商平台数据处理、在线教育课程管理等，以项目成果作为知识掌握的直观验证。
3. **可达成性（Attainable）**：基于初级软件工程师的编程基础，每天合理安排 2 - 3 小时专注学习时间，借助优质学习资源，通过理论学习、实践操作与复盘总结的循环学习模式，逐步实现知识的积累与能力提升。
4. **相关性（Relevant）**：MySQL 作为主流数据库管理系统，广泛应用于软件开发、数据分析等领域，掌握它能极大提升软件工程师在数据处理、存储与管理方面的核心竞争力，与职业发展紧密相连。
5. **时限性（Time - bound）**：整个学习过程严格限定在 60 天内，分为三个阶段，每个阶段明确时间节点与详细学习任务，确保学习进度有条不紊推进。

### 学习阶段与任务安排

#### 第一阶段（1 - 20 天）：基础入门

1. **第 1 - 3 天**
   1. **知识输入**：依据《60 天引爆你的学习力》中多维度学习方法，上午通过阅读《MySQL 必知必会》前 2 章内容，了解 MySQL 数据库基本概念，如数据库、表、字段、索引等，同时观看菜鸟教程 MySQL 专题中对应概念讲解视频，从文字与图像多渠道加深理解，做好笔记。
   2. **实操巩固**：下午参照 MySQL 官方文档，在本地计算机安装 MySQL 数据库，按照文档步骤详细配置，完成后进行多次连接测试，确保安装成功，过程中记录遇到的问题及解决方法。根据心理学记忆规律，睡前回顾当天学习内容，强化记忆。
   3. **营养支持**：遵循营养学建议，保持三餐营养均衡，早餐增加富含蛋白质的食物如鸡蛋、牛奶，为大脑提供充足能量，提升学习专注度。
2. **第 4 - 6 天**
   1. **知识输入**：上午阅读《MySQL 必知必会》第 3 - 4 章，学习数据类型（整数、浮点数、字符串、日期时间等），结合慕课网 MySQL 相关课程中数据类型讲解部分，理解不同数据类型使用场景与存储方式。
   2. **实操巩固**：下午利用 MySQL 命令行工具，每天进行 10 次以上创建、修改和删除数据库及表的操作练习，依据生理学劳逸结合原则，每练习 45 分钟休息 10 分钟，避免大脑疲劳。完成操作后，自行总结操作步骤与注意事项。
3. **第 7 - 9 天**
   1. **知识输入**：上午阅读《MySQL 必知必会》第 5 - 7 章，学习插入、更新和删除数据的 SQL 语句，同时参考牛客网 MySQL 练习题中相关知识点题目解析，加深对语法理解。
   2. **实操巩固**：下午在自己创建的表中进行数据操作练习，每天编写 15 条以上不同类型数据操作语句，完成后对比正确答案，分析错误原因。
4. **第 10 - 12 天**
   1. **知识输入**：上午学习 SELECT 语句基础用法，能查询指定列、过滤数据等，阅读《MySQL 必知必会》相关章节，并参考在线教程中多种查询示例。
   2. **实操巩固**：下午通过编写大量查询语句练习，尝试不同查询条件组合，结合《60 天引爆你的学习力》中总结归纳方法，将常用查询场景分类整理。
5. **第 13 - 15 天**
   1. **知识输入**：上午学习数据排序与聚合函数，如 ORDER BY、GROUP BY、SUM、AVG 等，阅读《MySQL 必知必会》第 8 - 11 章，观看慕课网对应课程讲解视频。
   2. **实操巩固**：下午完成书中练习题，通过实际案例操作，掌握这些函数在数据处理中的应用，每完成一个案例进行复盘，总结操作技巧。
6. **第 16 - 18 天**
   1. **知识输入**：复习本周所学 MySQL 基础操作知识，查漏补缺，结合《60 天引爆你的学习力》中知识关联方法，梳理知识点之间联系，形成知识框架。
   2. **实操巩固**：进行一次模拟测试，使用牛客网 MySQL 练习题组，按照考试要求在规定时间内完成，检验知识掌握程度，分析错题原因，针对性复习。
7. **第 19 - 20 天**
   1. **知识输入**：回顾本周所有学习内容与操作练习，强化记忆薄弱知识点。
   2. **实操巩固**：根据本周学习成果，总结一份 MySQL 基础操作手册，方便后续查阅。

#### 第二阶段（21 - 40 天）：深入学习

1. **第 21 - 23 天**
   1. **知识输入**：上午学习多表查询，包括内连接、外连接、交叉连接等，阅读《MySQL 必知必会》第 12 - 13 章，参考实际业务案例分析不同连接方式应用场景，观看慕课网多表查询专题课程。
   2. **实操巩固**：下午每天编写 5 个以上多表查询案例，案例涉及不同业务逻辑，完成后分析查询结果，理解数据之间关联关系。
2. **第 24 - 26 天**
   1. **知识输入**：上午学习子查询，了解在查询中嵌套子查询方法，阅读相关章节，参考 LeetCode 数据库题目中涉及子查询题目解析，学习不同子查询应用场景与技巧。
   2. **实操巩固**：下午通过练习题掌握子查询使用，每完成一道题，思考是否有其他解法，拓展解题思路。
3. **第 27 - 29 天**
   1. **知识输入**：上午学习索引创建与使用，阅读《高性能 MySQL》中关于索引章节，理解索引对查询性能影响原理。
   2. **实操巩固**：下午在自己数据库表中创建不同类型索引，使用 EXPLAIN 语句分析创建索引前后查询性能变化，记录数据并对比分析。
4. **第 30 - 32 天**
   1. **知识输入**：上午掌握事务处理，了解事务 ACID 特性，阅读相关资料与案例分析。
   2. **实操巩固**：下午通过实际操作练习事务开始、提交与回滚，模拟数据一致性与完整性受破坏场景，学习如何通过事务处理保障数据安全。
5. **第 33 - 35 天**
   1. **知识输入**：上午学习视图创建与使用，理解视图在数据安全与简化查询方面作用，阅读相关资料与教程。
   2. **实操巩固**：下午创建 5 个以上不同类型视图，使用视图进行数据查询，对比直接查询与通过视图查询优缺点。
6. **第 36 - 38 天**
   1. **知识输入**：上午了解存储过程与函数，学习创建和调用方法，参考在线教程与实际代码示例。
   2. **实操巩固**：下午编写存储过程与函数，实现特定业务逻辑功能，调用并测试功能是否正常。
7. **第 39 - 40 天**
   1. **知识输入**：复习本周所学 MySQL 深入知识，梳理知识点之间逻辑关系。
   2. **实操巩固**：完成一个小型数据库项目实践，如小型图书管理系统数据库，将本周所学知识应用到项目中，从数据库设计到 SQL 语句编写，完整实现项目功能，完成后进行项目总结与复盘。

#### 第三阶段（41 - 60 天）：综合提升

1. **第 41 - 43 天**
   1. **知识输入**：上午学习数据库优化，包括查询优化、表结构优化、服务器参数优化等，阅读《高性能 MySQL》相关章节，学习使用 EXPLAIN 语句分析查询执行计划原理。
   2. **实操巩固**：下午使用 EXPLAIN 语句对之前项目中查询语句进行分析，根据分析结果进行查询优化，对比优化前后查询效率，记录优化方法与效果。
2. **第 44 - 46 天**
   1. **知识输入**：上午学习数据库备份与恢复，阅读 MySQL 官方文档中关于数据库备份与恢复部分，了解不同备份方式与恢复策略。
   2. **实操巩固**：下午使用 MySQL 自带工具对自己项目数据库进行备份与恢复操作练习，模拟数据丢失场景，进行数据恢复测试，确保数据安全备份与有效恢复。
3. **第 47 - 49 天**
   1. **知识输入**：上午学习 MySQL 安全管理，包括用户管理、权限分配等，阅读相关资料与最佳实践案例。
   2. **实操巩固**：下午对自己数据库进行用户管理与权限分配设置，按照最小权限原则，保障数据库安全，设置完成后进行权限验证测试。
4. **第 50 - 52 天**
   1. **知识输入**：上午深入学习数据库设计范式，如第一范式、第二范式、第三范式等，阅读相关理论资料与案例分析。
   2. **实操巩固**：下午回顾之前项目数据库设计，检查是否符合范式要求，对不符合地方进行优化，优化完成后分析优化前后数据库结构优缺点。
5. **第 53 - 55 天**
   1. **知识输入**：复习 MySQL 优化、备份恢复、安全管理及数据库设计范式等知识，形成综合知识体系。
   2. **实操巩固**：完成一个综合性较强数据库项目，如小型电商系统数据库，涉及复杂数据关系与业务逻辑，从需求分析、数据库设计到 SQL 语句编写、性能优化、安全设置，完整实现项目功能，完成后进行项目总结与经验分享。
6. **第 56 - 58 天**
   1. **知识输入**：对整个 60 天学习内容进行全面回顾，整理笔记与项目经验，强化重点知识点记忆。
   2. **实操巩固**：进行一次全面知识测试，模拟实际工作场景，使用 LeetCode 数据库题目或企业实际面试题目，检验对 MySQL 知识掌握程度，针对薄弱环节进行强化复习。
7. **第 59 - 60 天**
   1. **知识输入**：再次梳理整个学习过程，总结学习方法与技巧，形成适合自己的 MySQL 学习经验。
   2. **实操巩固**：整理所有学习资料与项目成果，建立个人 MySQL 学习知识库，方便后续查阅与回顾。

#### 学习资源

1. **书籍**：《MySQL 必知必会》《高性能 MySQL》。
2. **在线教程**：菜鸟教程 MySQL 专题、慕课网 MySQL 相关课程。
3. **在线测试平台**：牛客网 MySQL 练习题、LeetCode 数据库题目。

# 硬件问题

### 1、相机、扫码枪通过TCP通讯连接电脑时显示为以太网，若以太网图标消失且无法连接，可能是由以下原因导致。

以下是分步解决方案：

￼

**可能原因**

1. 硬件问题

◦ USB接口松动、扫码枪供电不足或线缆损坏

◦ 扫码枪本身故障

2. 驱动异常

◦ 驱动未正确安装、被误删或更新失败

◦ 系统误识别为其他设备

3. 网络适配器设置错误

◦ 适配器被禁用、IP地址冲突或配置错误

◦ DHCP服务未启用导致无法获取IP

4. 系统或服务问题

◦ 网络相关服务（如DHCP Client）未运行

◦ 系统更新/软件冲突导致兼容性问题

5. 电源管理限制

◦ USB端口或适配器的省电模式自动关闭设备

￼

**解决方案**

第一步：检查硬件连接

1. 重新插拔扫码枪的USB接口，尝试更换其他USB端口。

2. 检查线缆是否有损坏，或更换另一根数据线。

3. 将扫码枪连接到其他电脑，确认是否为硬件故障。

第二步：检查驱动状态

1. 打开设备管理器

◦ Win + X → 选择“设备管理器”。

2. 查看网络适配器

◦ 展开“网络适配器”，检查是否有黄色感叹号或未知设备。

3. 更新/重装驱动

◦ 右键点击扫码枪对应的适配器 → 选择“更新驱动程序”或“卸载设备”，重启后系统会自动重装驱动。

◦ 手动安装：从扫码枪官网下载专用驱动。

第三步：启用网络适配器

1. 打开网络连接设置

◦ Win + R → 输入 ncpa.cpl → 回车。

2. 检查适配器状态

◦ 右键“以太网”图标 → 确保已启用（若显示“禁用”，点击“启用”）。

3. 重置网络配置

◦ 以管理员身份运行命令提示符，依次输入：

bash

复制

ipconfig /release

ipconfig /renew

netsh winsock reset

netsh int

ip reset

◦ 重启电脑。

第四步：配置IP地址

1. 手动设置静态IP

◦ 在网络连接中右键以太网 → 属性 → 双击“Internet协议版本4 (TCP/IPv4)” → 手动输入IP（需与扫码枪的IP段匹配，如扫码枪IP为192.168.1.100，可设置电脑IP为192.168.1.101，子网掩码255.255.255.0）。

第五步：检查系统服务

1. 启动关键服务

◦ Win + R → 输入 services.msc → 找到以下服务并确保状态为“正在运行”：

■ DHCP Client

■ Network Connections

■ Network Setup Service

◦ 右键选择“重新启动”服务。

第六步：关闭省电模式

1. 调整电源管理

◦ 设备管理器 → 右键扫码枪对应的网络适配器 → 属性 → 电源管理 → 取消勾选“允许计算机关闭此设备以节约电源”。

第七步：排除系统/软件冲突

1. 卸载最近更新

◦ 设置 → 更新与安全 → 查看更新历史 → 卸载最近的质量更新/功能更新。

2. 关闭防火墙/杀毒软件

◦ 临时禁用第三方安全软件，测试是否恢复正常。

第八步：系统还原或联系支持

1. 若问题突然出现且无法定位原因，尝试系统还原到之前正常的时间点。

2. 联系扫码枪厂商技术支持，确认是否存在固件更新或专用修复工具。

￼

总结

从硬件排查到驱动修复，再到网络配置调整，逐步缩小问题范围。多数情况下，重新启用适配器、更新驱动或重置网络配置即可解决。如仍无法解决，需考虑硬件故障或系统兼容性问题。

# 如何在GitHub上创建代码仓库？

Add files via upload：通过上传添加文件

Commit changes：提交更改

Commit directly to the main branch：直接向主分支提交

Create a new branch for this commit and start a pull request. Learn more about pull requests.

为此提交创建一个新分支并启动拉取请求。了解有关拉取请求的更多信息。

Processing your files：正在处理你的文件

Preview预览

pull request拉取请求

Name：CTD-adventure

密PSW码：“e”abcABC

以下为你详细介绍在 GitHub 上创建代码仓库的步骤：

### **前提条件**

在创建代码仓库之前，你需要拥有一个 GitHub 账号。若还未注册，可访问 [GitHub 官网](https://github.com/" \t "https://www.doubao.com/chat/_blank)，点击右上角的 “Sign up”，按照提示完成注册流程。

### **创建代码仓库的步骤**

#### **1. 登录 GitHub 账号**

打开 [GitHub 官网](https://github.com/" \t "https://www.doubao.com/chat/_blank)，在页面右上角点击 “Sign in”，输入你的用户名和密码，完成登录。

#### **2. 进入创建仓库页面**

登录成功后，有两种方式可以进入创建仓库的页面：

* ****方式一****：点击页面右上角的 “+” 号，在弹出的菜单中选择 “New repository”。
* ****方式二****：在 GitHub 主页，点击左侧导航栏中的 “Repositories” 选项，然后在该页面的右上角点击 “New” 按钮。

#### **3. 填写仓库相关信息**

进入创建仓库页面后，你需要填写以下必要信息：

* ****Repository name****：为你的仓库命名，这个名称应具有一定的描述性，方便你和其他开发者识别。例如，如果你要创建一个用于存放个人博客代码的仓库，你可以命名为 “personal - blog - code”。
* ****Description（可选）****：简要描述这个仓库的用途，让其他访问者快速了解仓库内容。比如，对于上述个人博客代码仓库，你可以描述为 “This repository contains the source code for my personal blog.”
* ****Public or Private****：选择仓库的可见性。
  + ****Public****：公开仓库，任何人都可以查看仓库中的代码，也可以进行 fork（复刻）和贡献代码。适合开源项目。
  + ****Private****：私有仓库，只有你和你授权的人才能访问和查看代码。适合存放一些不希望公开的项目代码。
* ****Initialize this repository with****：
  + ****Add a README file****：勾选此项后，GitHub 会自动为你的仓库创建一个 README 文件。README 文件是仓库的重要说明文档，你可以在其中介绍项目的背景、功能、使用方法等信息。
  + ****Add .gitignore****：.gitignore 文件用于指定哪些文件或目录不需要被 Git 跟踪，避免将一些不必要的文件（如编译生成的文件、日志文件等）提交到仓库中。你可以从下拉菜单中选择适合你项目的 .gitignore 模板，例如 Python 项目可以选择 Python 模板。
  + ****Choose a license****：为你的项目选择一个开源许可证，明确他人使用、修改和分发你代码的权限和限制。常见的开源许可证有 MIT、Apache 2.0、GPL 等。

#### **4. 创建仓库**

完成上述信息填写后，点击页面下方的 “Create repository” 按钮，GitHub 会立即为你创建一个新的代码仓库。

#### **5. 后续操作**

仓库创建成功后，你可以进行以下操作：

* ****上传代码****：如果你本地已经有代码项目，可以通过 Git 命令将代码推送到新创建的仓库中。示例命令如下：

bash

# 初始化本地仓库git init# 添加文件到暂存区git add .# 提交文件到本地仓库git commit -m "Initial commit"# 添加远程仓库地址git remote add origin https://github.com/yourusername/your-repository-name.git# 推送代码到远程仓库git push -u origin main

* ****克隆仓库****：如果你想在本地获取这个仓库的副本，可以使用 git clone 命令。例如：

bash

git clone https://github.com/yourusername/your-repository-name.git

通过以上步骤，你就可以在 GitHub 上成功创建一个代码仓库，并开始管理你的代码项目。