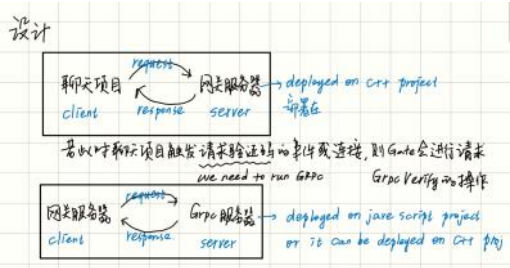


----- Ep9 NodeJs 实现邮箱验证服务 -----



》》》什么是 npm ？

npm 是 Node.js 的包管理工具，主要用于管理 JavaScript 语言的库和工具。它是 Node.js 的默认包管理器，通过它可以轻松地安装、更新、配置和管理项目所需的依赖包。

》》》安装了 nodejs 之后，我们创建一个文件夹并且在其下运行指令

npm init

一路回车

```
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to E:\VS\JustinChat\OtherProj\VerifyServer\package.json:
{
  "name": "verifyserver",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "description": ""
}

Is this OK? (yes)

npm notice
npm notice New major version of npm available! 10.8.2 -> 11.4.1
npm notice Changelog: https://github.com/npm/cli/releases/tag/v11.4.1
npm notice To update run: npm install -g npm@11.4.1
npm notice
E:\VS\JustinChat\OtherProj\VerifyServer\
```

》》》JS 的基本语法：

1. 变量声明	JavaScript 使用 var、let 和 const 来声明变量。						
	<table><tr><td>var:</td><td>传统的声明方式，具有函数作用域。</td></tr><tr><td>let:</td><td>用于声明可变的块级作用域变量。</td></tr><tr><td>const:</td><td>用于声明常量，常量的值不能改变。</td></tr></table>	var:	传统的声明方式，具有函数作用域。	let:	用于声明可变的块级作用域变量。	const:	用于声明常量，常量的值不能改变。
var:	传统的声明方式，具有函数作用域。						
let:	用于声明可变的块级作用域变量。						
const:	用于声明常量，常量的值不能改变。						
示例	let x = 10; // 声明变量x const y = 20; // 声明常量y						

什么是 JS 中的 let ？

在 JavaScript 中，let 是用来声明变量的一种方式。它是 ES6（ECMAScript 2015）引入的，并且相对于 var 有一些重要的改进。主要特点：块级作用域（Block Scope）与 var 不同，let 声明的变量具有块级作用域。这意味着它只在代码块（如函数、条件语句、循环等）内部有效，而 var 声明的变量具有函数作用域，即在整个函数内都可以访问。

```
if (true) {
  let x = 10;
  console.log(x); // 输出 10
}
```

console.log(x); // 报错 ReferenceError: x is not defined

2. 数据类型	JavaScript 有 6 种基本数据类型：												
	<table><tr><td>Number:</td><td>数字类型。</td></tr><tr><td>String:</td><td>字符串类型。</td></tr><tr><td>Boolean:</td><td>布尔类型（true 或 false）。</td></tr><tr><td>Object:</td><td>对象类型。</td></tr><tr><td>Null:</td><td>空值类型，表示“没有值”。</td></tr><tr><td>Undefined:</td><td>未定义类型，表示变量已声明但未赋值。</td></tr></table>	Number:	数字类型。	String:	字符串类型。	Boolean:	布尔类型（true 或 false）。	Object:	对象类型。	Null:	空值类型，表示“没有值”。	Undefined:	未定义类型，表示变量已声明但未赋值。
Number:	数字类型。												
String:	字符串类型。												
Boolean:	布尔类型（true 或 false）。												
Object:	对象类型。												
Null:	空值类型，表示“没有值”。												
Undefined:	未定义类型，表示变量已声明但未赋值。												
示例：	let num = 10; // Number let name = "Alice"; // String let isTrue = true; // Boolean let obj = { name: "Alice", age: 25 }; // Object let nothing = null; // Null let something; // Undefined												

3. 运算符							
	<table><tr><td>算术运算符:</td><td>+, -, *, /, %, ++, --</td></tr><tr><td>比较运算符:</td><td>==, ===, !=, !==, >, <, >=, <=</td></tr><tr><td>逻辑运算符:</td><td>&& (与), (或), ! (非)</td></tr></table>	算术运算符:	+, -, *, /, %, ++, --	比较运算符:	==, ===, !=, !==, >, <, >=, <=	逻辑运算符:	&& (与), (或), ! (非)
算术运算符:	+, -, *, /, %, ++, --						
比较运算符:	==, ===, !=, !==, >, <, >=, <=						
逻辑运算符:	&& (与), (或), ! (非)						
示例:	<pre>let a = 5, b = 10; console.log(a + b); // 15 console.log(a > b); // false console.log(a === 5); // true</pre>						

4. 条件语句	JavaScript 中的条件语句包括 if、else if、else 和 switch。
示例:	<pre>if (x > 10) { console.log("x 大于 10"); } else { console.log("x 小于或等于 10"); }</pre>

5. 循环语句	JavaScript 支持多种循环结构，如 for、while 和 do...while。				
	<table><tr><td>for 循环:</td><td><pre>for (let i = 0; i < 5; i++) { console.log(i); }</pre></td></tr><tr><td>while 循环:</td><td><pre>let i = 0; while (i < 5) { console.log(i); i++; }</pre></td></tr></table>	for 循环:	<pre>for (let i = 0; i < 5; i++) { console.log(i); }</pre>	while 循环:	<pre>let i = 0; while (i < 5) { console.log(i); i++; }</pre>
for 循环:	<pre>for (let i = 0; i < 5; i++) { console.log(i); }</pre>				
while 循环:	<pre>let i = 0; while (i < 5) { console.log(i); i++; }</pre>				

6. 函数定义	在 JavaScript 中，你可以通过函数声明、函数表达式或箭头函数来定义函数。						
	<table><tr><td>函数声明:</td><td><pre>function greet(name) { return "Hello, " + name; } console.log(greet("Alice"));</pre></td></tr><tr><td>函数表达式:</td><td><pre>const add = function(a, b) { return a + b; }; console.log(add(5, 3));</pre></td></tr><tr><td>箭头函数:</td><td><pre>const multiply = (a, b) => a * b; console.log(multiply(4, 2));</pre></td></tr></table>	函数声明:	<pre>function greet(name) { return "Hello, " + name; } console.log(greet("Alice"));</pre>	函数表达式:	<pre>const add = function(a, b) { return a + b; }; console.log(add(5, 3));</pre>	箭头函数:	<pre>const multiply = (a, b) => a * b; console.log(multiply(4, 2));</pre>
函数声明:	<pre>function greet(name) { return "Hello, " + name; } console.log(greet("Alice"));</pre>						
函数表达式:	<pre>const add = function(a, b) { return a + b; }; console.log(add(5, 3));</pre>						
箭头函数:	<pre>const multiply = (a, b) => a * b; console.log(multiply(4, 2));</pre>						

7. 数组	数组是存储多个值的容器，可以包含不同类型的数据。
示例:	<pre>let fruits = ["Apple", "Banana", "Cherry"]; console.log(fruits[0]); // Apple</pre>

8. 对象	对象是由一组键值对组成的数据结构。
示例:	<pre>let person = { name: "Alice", age: 25, greet: function() { console.log("Hello " + this.name); } }; console.log(person.name); // Alice person.greet(); // Hello Alice</pre>

9. 事件处理	JavaScript 常用于网页中的事件处理，例如按钮点击、输入框变化等。
示例:	<pre>document.getElementById("myButton").addEventListener("click", function() { alert("Button clicked!"); });</pre>

》》》代码解析: proto.js （这段代码实现了使用 gRPC 和 protobuf 来加载并定义一个 gRPC 服务的消息协议）

```
1 const path = require('path')
2 const grpc = require('@grpc/grpc-js')
3 const protoLoader = require('@grpc/proto-loader')
4
5 const PROTO_PATH = path.join(__dirname, 'message.proto')
6 const packageDefinition = protoLoader.loadSync(PROTO_PATH, {keepCase:true, longs:String, enums:String, defaults:true, oneofs:true})
7 const protoDescriptor = grpc.loadPackageDefinition(packageDefinition)
8
9 const messageProto = protoDescriptor.message
10
11 module.exports = messageProto
```

1. 引入模块（类似于引入头文件或者外部库）

```
1 const path = require('path')
2 const grpc = require('@grpc/grpc-js')
3 const protoLoader = require('@grpc/proto-loader')
4
```

path:	这是 Node.js 内置的模块，用于处理文件和目录的路径。它提供了路径操作的一些功能，比如拼接路径等。
grpc:	这是使用 gRPC 协议的 Node.js 客户端和服务端的核心库，提供了通信协议的功能。
protoLoader:	这是一个库，用于加载和解析 .proto 文件。 .proto 文件是 Protocol Buffers 的定义文件，定义了消息格式和服务接口。

2. 设置 .proto 文件路径

```
const PROTO_PATH = path.join(__dirname, 'message.proto')
```

__dirname:	这是 Node.js 中的一个全局变量，表示当前模块文件的目录路径。
path.join(__dirname, 'message.proto'):	这里将当前文件目录和 message.proto 拼接起来，得到 message.proto 文件的绝对路径。

3. 加载 .proto 文件

```
const packageDefinition = protoLoader.loadSync(PROTO_PATH, {keepCase:true, longs:String, enums:String, defaults:true, oneofs:true})
```

protoLoader.loadSync: 这个函数可以同步加载指定的 .proto 文件，并返回其内容（返回一个包含 .proto 文件内容的对象）。这里传入的 PROTO_PATH 是 .proto 文件的路径。传入的第二个参数是一个配置对象，含有以下选项：

keepCase: true	保持原有的字段名大小写（默认会将字段名转为小写）。
longs: String	在 .proto 文件中， long 类型会被转换为字符串，以避免大数字导致的精度问题。
enums: String	将枚举类型的值转换为字符串，而不是数字。
defaults: true	为每个字段设置默认值。
oneofs: true	为 oneof 类型的字段提供正确的值。

什么是 oneofs ?

在 Protocol Buffers（简称 Protobuf）中， oneof 是一种特殊的语法，用于在定义一组互斥的字段，即在同一时间只能设置其中一个字段的值。

1. 基本概念:	oneof 允许你在一个消息中定义多个字段，但这些字段的值在同一时刻只能有一个有效值。这样可以节省存储空间，并确保在处理数据时，只有一个字段被使用。				
2. 使用场景:	oneof 常用于表示一个字段可以是多个不同类型中的某一种。例如，某个消息可能有多种类型的响应字段，但同一时刻只能有一个字段有效。比如，可能的字段包括：整数、字符串、布尔值或某个嵌套消息等。				
3. 语法:	<p>在 Protobuf 中， oneof 的语法如下：</p> <pre>message Example { oneof response { int32 id = 1; string name = 2; bool is_active = 3; } }</pre> <p>在上面的示例中， Example 消息定义了一个 oneof 字段 response，它包含了三个互斥的字段： id、 name 和 is_active。在实际使用时， Example 消息对象只能设置其中一个字段，不能同时设置多个。</p>				
4. 重要特性:	<table><tr><td>互斥性和自动清除:</td><td>同一时刻， oneof 中的字段只能有一个被赋值。当你为 oneof 中的某个字段赋值时，其他字段的值会自动清除，变为未设置状态。</td></tr><tr><td>默认值:</td><td>每个 oneof 中的字段都有其默认值。比如，如果没有设置某个字段，它的默认值可能是零、空字符串或 false，具体取决于字段类型。</td></tr></table>	互斥性和自动清除:	同一时刻， oneof 中的字段只能有一个被赋值。当你为 oneof 中的某个字段赋值时，其他字段的值会自动清除，变为未设置状态。	默认值:	每个 oneof 中的字段都有其默认值。比如，如果没有设置某个字段，它的默认值可能是零、空字符串或 false，具体取决于字段类型。
互斥性和自动清除:	同一时刻， oneof 中的字段只能有一个被赋值。当你为 oneof 中的某个字段赋值时，其他字段的值会自动清除，变为未设置状态。				
默认值:	每个 oneof 中的字段都有其默认值。比如，如果没有设置某个字段，它的默认值可能是零、空字符串或 false，具体取决于字段类型。				
5 优点:	<table><tr><td>节省空间:</td><td>oneof 允许多个字段共享同一内存区域，节省了存储空间。</td></tr><tr><td>清晰的设计:</td><td>通过 oneof，可以清晰地表达多个字段之间的互斥关系，避免了无效字段的填充。</td></tr></table>	节省空间:	oneof 允许多个字段共享同一内存区域，节省了存储空间。	清晰的设计:	通过 oneof，可以清晰地表达多个字段之间的互斥关系，避免了无效字段的填充。
节省空间:	oneof 允许多个字段共享同一内存区域，节省了存储空间。				
清晰的设计:	通过 oneof，可以清晰地表达多个字段之间的互斥关系，避免了无效字段的填充。				
6. 示例:	<p>假设你正在设计一个聊天应用，可能需要不同类型的消息（文本、图片、视频等）。你可以使用 oneof 来定义消息的类型，这样每个消息只能有一种类型的数据。</p> <pre>message Message { oneof content</pre>				

```

{
  string text = 1;
  bytes image = 2;
  bytes video = 3;
}

```

在这个例子中，Message 消息的 content 字段可以是文本、图像或视频，但只能有一个有效。

4. 加载并解析 gRPC 服务

```
const protoDescriptor = grpc.loadPackageDefinition(packageDefinition)
```

grpc.loadPackageDefinition: 这个函数接受 protoLoader 加载后的包定义（即 packageDefinition），然后根据这些定义来创建 gRPC 服务的 JavaScript 版本。

protoDescriptor: 这个变量包含了从 .proto 文件中提取的服务和消息类型的描述信息，接下来可以通过它访问相应的 gRPC 服务定义和消息类型。

5. 获取消息类型定义

```
const messageProto = protoDescriptor.message
```

protoDescriptor.message: 假设在 message.proto 文件中定义了一个叫做 message 的服务或者消息类型，这一行代码从 protoDescriptor 中获取该消息类型的定义。

message_proto: 这是一个包含 message 服务或消息类型的对象，允许你在代码中使用它，比如创建消息实例、调用服务等。

6. 导出消息类型（将 message_proto 导出，使得其他文件能够引用这个文件并使用 message_proto 中定义的消息和服务）

```
module.exports = messageProto
```

module.exports: 这是 Node.js 中的一个语法，用于将一个模块的内容导出，供其他文件引用。

》》》配置的设置和读取:

Json:

```

{
  "email": {
    "user": " ",
    "pass": " "
  }
}

```

Config.js:

fs.readFileSync('config.json', 'utf8')	fs.readFileSync() 是 fs 模块中的一个同步方法，用于读取文件的内容。同步方法会在完成任务后返回结果（如果文件是文本文件则返回字符串类型对象），并且会阻塞代码的执行，直到文件读取完毕。 • 'utf8': 这是文件的字符编码，指定读取的文件是以 UTF-8 编码方式来解码的。utf8 可以确保返回的内容是字符串类型，而不是 Buffer 对象。
JSON.parse()	JSON.parse() 是 JavaScript 中内置的一个方法，用于将 JSON 格式的字符串转换成 JavaScript 对象。 • JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式，用于表示结构化的数据。JSON.parse() 会将符合 JSON 格式的字符串解析为 JavaScript 对象，使得我们能够访问其中的成员数据。

》》》465 这个端口的作用?

端口 465 主要用于 SMTP (Simple Mail Transfer Protocol) 加密传输，特别是在通过 SMTPS 协议进行安全的电子邮件发送时。

具体来说，端口 465 用于 SMTP over SSL/TLS（即通过 SSL/TLS 加密的 SMTP）连接。虽然这个端口曾经是标准端口之一，但它在 2001 年被 IETF（互联网工程任务组）弃用了，推荐使用 587 端口进行加密的邮件发送。然而，端口 465 仍然被一些邮件服务器和客户端应用程序支持和使用。

》》》》auth -> Authorization 译为：授权

》》》》什么是 std::future ？？

std::future 是 C++11 标准中引入的一个模板类，用于处理异步操作的结果。它允许你获取一个异步任务（通常由 std::async 或线程创建）执行后的返回值或异常。简而言之，std::future 提供了与异步操作的结果进行交互的机制。

主要功能：	1.获取结果：你可以使用 std::future 来获取一个异步操作的返回值。当异步任务完成时，std::future 会提供该任务的结果。 2.等待任务完成：你可以通过调用 get() 来等待任务完成，并获取它的结果。get() 会阻塞调用线程，直到异步任务完成。 3.处理异常：如果异步任务在执行过程中抛出异常，get() 会重新抛出该异常，允许你在主线程中处理。						
常见用法：	<p>通常，std::future 与 std::async 配合使用，std::async 用于启动一个异步任务，std::future 用来接收任务的结果。</p> <p>示例代码：</p> <pre>#include <iostream> #include <future> #include <thread> // 一个简单的异步函数 int add(int a, int b) { std::this_thread::sleep_for(std::chrono::seconds(2)); // 模拟耗时操作 return a + b; } int main() { // 使用 std::async 启动一个异步任务 std::future<int> result = std::async(std::launch::async, add, 3, 4); // 这里可以做其他事情，也可以等待 result.get() 获取异步任务的结果 std::cout << "异步任务正在执行..." << std::endl; // 获取异步任务的结果，这里会阻塞直到任务完成 int sum = result.get(); // 获取 add(3, 4) 的返回值 std::cout << "计算结果: " << sum << std::endl; return 0; }</pre> <p>1.std::async：用来启动一个异步任务，返回一个 std::future 对象。这个对象代表了将来某个时刻的结果。 2.result.get()：get() 会阻塞调用它的线程，直到异步任务完成并返回结果。在异步任务完成之前，主线程会继续执行其他代码。get() 还会处理异常，如果异步任务抛出异常，它会在主线程中重新抛出该异常。 (异常处理：如果异步任务抛出异常，调用 get() 会重新抛出这个异常，因此可以在调用 get() 的地方捕获并处理异常。)</p>						
常用成员函数：	<table><tr><td>get()：</td><td>等待异步任务完成并获取其结果。如果任务抛出异常，它会重新抛出。</td></tr><tr><td>valid()：</td><td>检查 std::future 是否包含一个有效的异步任务（即检查它是否已经与某个异步操作关联）。</td></tr><tr><td>wait()：</td><td>等待异步任务完成，但不会返回结果，仅用于同步操作。</td></tr></table>	get()：	等待异步任务完成并获取其结果。如果任务抛出异常，它会重新抛出。	valid()：	检查 std::future 是否包含一个有效的异步任务（即检查它是否已经与某个异步操作关联）。	wait()：	等待异步任务完成，但不会返回结果，仅用于同步操作。
get()：	等待异步任务完成并获取其结果。如果任务抛出异常，它会重新抛出。						
valid()：	检查 std::future 是否包含一个有效的异步任务（即检查它是否已经与某个异步操作关联）。						
wait()：	等待异步任务完成，但不会返回结果，仅用于同步操作。						

》》》》什么是 std::promise ??

在 C++ 中，std::promise 是一个与多线程编程相关的工具类，定义在 <future> 头文件中。它通常与 std::future 配合使用，用于在线程之间传递异步操作的结果。

核心作用	std::promise 允许一个线程（生产者线程）设置一个值或异常，另一个线程（消费者线程）可以通过关联的 std::future 对象获取这个值。这种机制实现了线程间的单向数据传递。
基本用法	<p>1.创建 promise 和 future</p> <pre>#include <future> std::promise<int> promise_obj; std::future<int> future_obj = promise_obj.get_future();</pre> <p>2.设置值（生产者线程）</p> <pre>promise_obj.set_value(42); // 传递结果 // 或者传递异常：promise_obj.set_exception(std::make_exception_ptr(std::runtime_error("Error")));</pre> <p>3.获取值（消费者线程）</p> <pre>int result = future_obj.get(); // 阻塞直到值被设置</pre>
典型应用场景	<p>1.线程间传递异步结果</p> <pre>void producer(std::promise<int> p) { // 模拟耗时操作 std::this_thread::sleep_for(std::chrono::seconds(1)); p.set_value(100); } int main() { std::promise<int> p;</pre>

	<pre>std::future<int> f = p.get_future(); std::thread t(producer, std::move(p)); std::cout << "Result: " << f.get() << std::endl; // 阻塞等待结果 t.join(); return 0; }</pre> <p>2. 异常传递</p> <p>如果生产者线程发生错误，可以通过 <code>set_exception</code> 传递异常：</p> <pre>try { // 可能抛出异常的操作 } catch (...) { promise_obj.set_exception(std::current_exception()); }</pre>
关键特性	<p>1. 单次通信</p> <p>每个 <code>std::promise</code> 只能设置一次值（或异常），多次调用 <code>set_value</code> 会抛出 <code>std::future_error</code>。</p> <p>2. 移动语义</p> <p><code>std::promise</code> 不可拷贝，但可以通过 <code>std::move</code> 转移所有权：</p> <pre>std::promise<int> p1; std::promise<int> p2 = std::move(p1); // 合法</pre> <p>3. 生命周期管理</p> <p>如果 <code>std::promise</code> 在未设置值时被销毁，关联的 <code>std::future</code> 会抛出 <code>std::future_error</code>（错误码为 <code>broken_promise</code>）。</p>
与 <code>std::future</code> 的配合	<ul style="list-style-type: none">• <code>std::future</code> 通过 <code>get()</code> 获取值（阻塞直到值就绪）。• 可通过 <code>wait()</code> 或 <code>wait_for()</code> 实现超时等待。• <code>valid()</code> 方法检查 <code>future</code> 是否关联到有效的共享状态。

》》》》 java script 中的 Promise:

Promise 对象有三种状态：

1. Pending（待定）：初始状态，表示 Promise 还没有完成。
2. Fulfilled（已完成）：表示操作成功完成，并且 Promise 被解析。
3. Rejected（已拒绝）：表示操作失败，并且 Promise 被拒绝。

Promise 的工作原理	<p>一个 Promise 对象的作用就是将一个异步操作的结果（成功或失败）封装起来，提供一个统一的接口，使得你可以在异步操作完成后执行相应的操作，而不会阻塞程序的执行。</p> <pre>const promise = new Promise((resolve, reject) => { let success = true; if (success) { resolve("成功了！"); // 操作成功时调用 resolve() } else { reject("出错了！"); // 操作失败时调用 reject() } });</pre>
---------------	--

定义：

Promise 构造函数	<p>Promise 构造函数接受一个 executor 函数作为参数，这个函数有两个参数：resolve 和 reject。</p> <ul style="list-style-type: none">• <code>resolve(value)</code>：表示异步操作成功，value 是成功的返回值。Promise 会从 "待定" 状态变为 "已完成" 状态。• <code>reject(reason)</code>：表示异步操作失败，reason 是失败的原因。Promise 会从 "待定" 状态变为 "已拒绝" 状态。
参数：	Promise 的构造函数接受一个回调函数，resolve 和 reject 是传递给这个回调函数的两个参数。
返回值：	返回的是一个 Promise 对象。Promise 对象最终会进入两种状态之一：resolved（操作成功）或 rejected（操作失败）。

》》JavaScript 中的 Promise 函数与 C++ 中的 `std::future` 和 `std::promise` 是什么类比关系？

JavaScript 中的 Promise 和 C++ 中的 `std::future` 和 `std::promise` 都与异步操作的结果传递和处理相关，它们的基本功能相似，但实现方式和用法有所不同。

类比关系	<ul style="list-style-type: none">• JavaScript 的 Promise 类似于 C++ 中的 <code>std::future</code>。• JavaScript 中的 <code>resolve()</code> 和 <code>reject()</code> 类似于 C++ 中的 <code>std::promise::set_value()</code> 和 <code>std::promise::set_exception()</code>。
------	--

》》》》 email.js 文件中，对发送邮件的函数做一些分析。

```
7 // 创建发送邮件的函数
8 ~ function SendMail(mailOptions)
9 {
10     return new Promise(function(reslove, reject)
11     {
12         transport.sendMail(mailOptions, function(){});
13     });
14 }
15 }
```

函数名:	SendMail
参数:	mailOptions
返回值:	返回一个 Promise 对象 (Promise 函数的返回值)

new Promise(function(resolve, reject) {...})

在 SendMail 函数中, 我们使用 Promise 函数, 这个函数:

函数名:	Promise
参数:	执行器函数 <code>function(resolve, reject) { xxxxxx }</code> (执行器函数本身接收两个参数: resolve 和 reject) <pre>return new Promise(function(reslove, reject) { /* 执行器函数的定义 */ transport.sendMail(mailOptions, function(error, info) { // ... }); });</pre>
返回值:	Promise 对象 (这个 Promise 对象的状态会随着异步操作的完成 (成功或失败) 而改变)

transport.sendMail(...)

在对执行器函数进行定义的时候, 我们调用了 transport 的成员函数 sendMail()

函数名:	sendMail
参数:	mainOptions 和一个回调函数。 回调函数是这样定义的: <pre>transport.sendMail(mailOptions, function(error, info) { if (error) { console.log(error); reject(error); } else { console.log('邮件已经成功发送', info.response); resolve(info.response); } })</pre>
为什么需要调用 reject 和 resolve 呢? 具体参考以下:	
Promise 构造函数	Promise 构造函数接受一个 executor 函数作为参数, 这个函数有两个参数: resolve 和 reject。 <ul style="list-style-type: none">• resolve(value): 表示异步操作成功, value 是成功的返回值。Promise 会从 "待定" 状态变为 "已完成" 状态。• reject(reason): 表示异步操作失败, reason 是失败的原因。Promise 会从 "待定" 状态变为 "已拒绝" 状态。
参数:	Promise 的构造函数接受一个回调函数, resolve 和 reject 是传递给这个回调函数的两个参数。
返回值:	返回的是一个 Promise 对象。Promise 对象最终会进入两种状态之一: resolved (操作成功) 或 rejected (操作失败)。

》》》》 端口 50051 的用途 / 功能

端口 50051 通常用于 gRPC (Google Remote Procedure Call) 协议的服务。gRPC 是一种高性能、开源的远程过程调用 (RPC) 框架, 它由 Google 开发并使用 Protocol Buffers (protobuf) 作为接口定义语言和消息传输格式。

gRPC 使用端口 50051:

```
void RegisterDialog::on_get_code_clicked()
{
    auto emailStr = ui->email_edit->text();

    QRegularExpression regex(R"((\w+)(\._|_|-)?(\w+)?(\.(\w+))?)")
    bool matchResult = regex.match(emailStr).hasMatch();
    if(matchResult)
    {
        QJsonObject jsonObj;
        jsonObj["email"] = emailStr;
        HttpMgr::GetInstance()->PostHttpRequest(QUrl(gateUrlPrefix + "/get_varifcode"), jsonObj, ReqID::ID_GET_VERIFY_CODE, Module::REGISTER)
    }
    else
    {
        // 这里存储对于客户端 POST 请求的简单处理方法
        ShowTip(tr("邮箱地址不正确"));
        ReqPost("get_varifcode", [] (std::shared_ptr<HttpConnection> connection)
        {
            Json::Value reqRoot;
            Json::Reader reader;

            auto reqData = boost::beast::buffers_to_string(connection->m_Request.body().data());
            JC_CORE_INFO("Receive body is :\n", reqData);
            bool success = reader.parse(reqData, reqRoot);
            if(!success)
            {
                JC_CORE_ERROR("Failed to parse JSON data!\n");
                reqRoot["error"] = ErrorCode::Error Json;
            }
        });
    }
}
```

QT

GateServer

有关系吗?


```

auto reqData = boost::beast::buffers_to_string(connection->m_Request.body().data());
JC_CORE_INFO("Receive body is {}\\n", reqData);
bool success = reader.parse(reqData, reqRoot);
if(!success)
{
    JC_CORE_ERROR("Failed to parse JSON data\\n");
    rspRoot["error"] = ErrorCodes::Error_json;
    boost::beast::ostream(connection->m_Response.body()) << rspRoot.toStyledString();
    return true;
}

std::string email = reqRoot["email"].asString();
message::GetVerifyReq rsp = VerifyGrpcClient::GetInstance()->GetVerifyCode(email);

rspRoot["error"] = rsp.error();
rspRoot["email"] = email;
JC_CORE_INFO("Handling email: {}...\\n", email);

boost::beast::ostream(connection->m_Response.body()) << rspRoot.toStyledString();
return true;
}

```

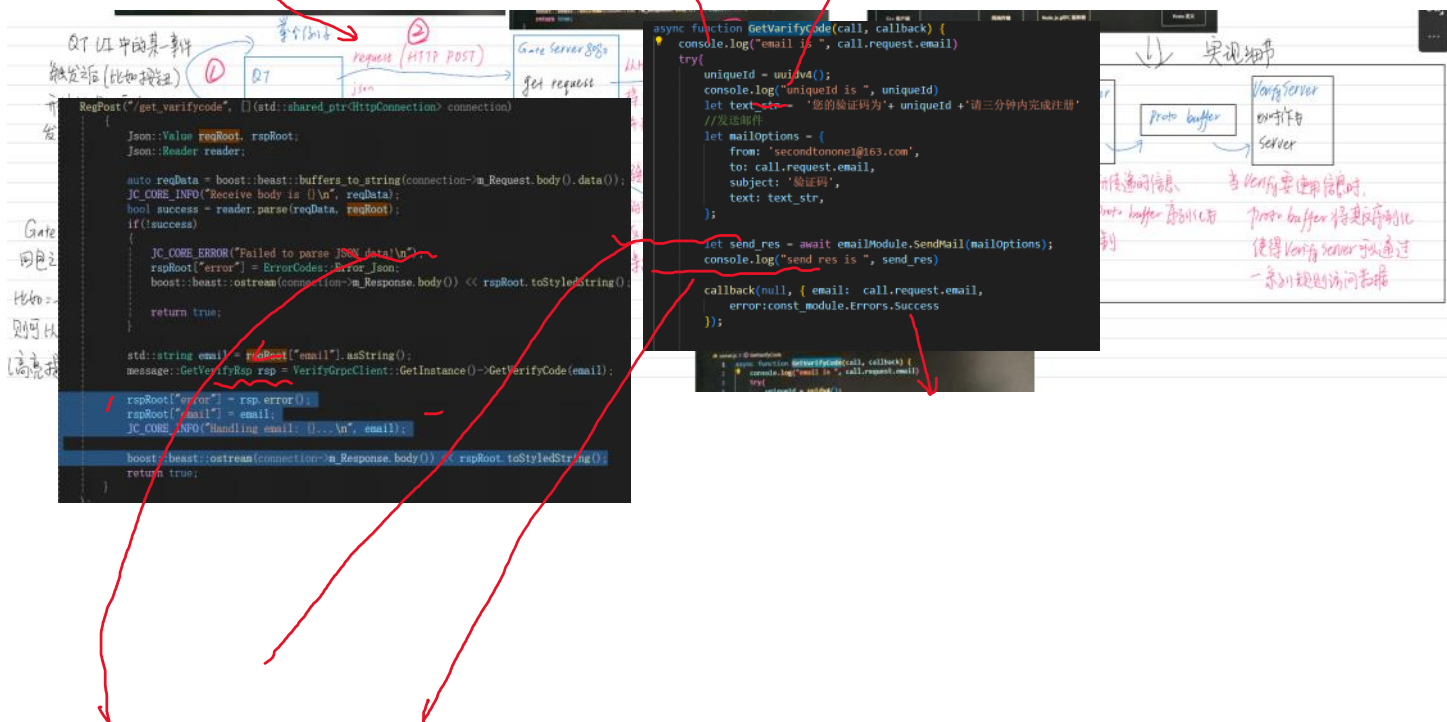
GateServer

```

function main()
{
    var server = new grpc.Server();
    // 添加服务的函数定义(处理逻辑)
    server.addService(messageProto.VerifyService.service, { GetVerifyCode, GetVerifyCode });
    // 将 grpc 服务绑定在 50051 端口, 并指定配置(Insecure 类型连接), 然后在回调中启动监听和日志
    server.bindAsync('0.0.0.0:50051', grpc.ServerCredentials.createInsecure(), () => { server.start(); console.log('Server started on 50051'); });
}

main();

```



上面的图片是我在移动设备上做的笔记，如果实在难看懂，我放了一些文字笔记，可供查阅：

》》客户端和服务端的调用流程：

1. message.proto - 服务契约

```

syntax = "proto3";
package message;

service VerifyService {
    rpc GetVerifyCode (GetVerifyReq) returns (GetVerifyResp) {}
}

message GetVerifyReq {
    string email = 1; // 关键字段
}

message GetVerifyResp {
    int32 error = 1;
    string email = 2;
    string code = 3;
}

```

2. proto.js - Proto 加载器

Javascript	<pre>const path = require('path'); const grpc = require('@grpc/grpc-js'); // 修正, grpc-js const protoLoader = require('@grpc/proto-loader'); const PROTO_PATH = path.join(__dirname, 'message.proto'); // 加载proto文件 const packageDefinition = protoLoader.loadSync(PROTO_PATH, { keepCase: true, longs: String, enums: String, defaults: true, oneofs: true }); // 创建gRPC定义 const protoDescriptor = grpc.loadPackageDefinition(packageDefinition); // 获取message包 const messageProto = protoDescriptor.message; module.exports = messageProto;</pre>
关键作用:	1.动态加载 message.proto 文件 2.生成 JavaScript 可用的服务定义 3.创建 messageProto 对象 (包含服务和方法定义)

3. server.js - gRPC 服务端

Javascript	<pre>// 导入proto定义 const message_proto = require('./proto.js'); async function GetVerifyCode(call, callback) { // 关键点, call.request 来自proto定义 console.log("email is ", call.request.email) // ...邮件发送逻辑... } function main() { var server = new grpc.Server() // 注册服务, 将proto定义与实现函数绑定 server.addService(message_proto.VerifyService.service, // 来自proto.js { GetVerifyCode: GetVerifyCode } // 实现函数) server.bindAsync('0.0.0.0:50051', ...) }</pre>
------------	--

4. C++ 客户端 - gRPC 调用方

Cpp	<pre>message::GetVerifyResp VerifyGrpcClient::GetVerifyCode(const std::string& email) { grpc::ClientContext context; message::GetVerifyResp rsp; message::GetVerifyReq req; // 设置请求字段 req.set_email(email); // 设置email值 // 发起gRPC调用 grpc::Status status = m_Stub->GetVerifyCode(&context, req, &rsp); // ...处理响应... }</pre>
-----	---

》》数据流分析: email 如何传递

步骤1:	C++ 客户端设置请求
Cpp	<pre>req.set_email("user@example.com"); // 设置email值</pre>

步骤2:	gRPC 序列化
根据 message.proto 定义:	<pre>message GetVerifyReq { string email = 1; // 字段ID=1, 类型=string }</pre>
过程	<ul style="list-style-type: none">•gRPC 使用 Protocol Buffers 序列化•将 GetVerifyReq 对象转换为二进制格式•序列化规则由 proto 定义:
将数据序列化为 Protocol Buffer 二进制格式:	<pre>0A 10 75 73 65 72 40 65 78 61 6D 70 6C 65 2E 63 6F 6D</pre> <p>0A: 字段ID(1)和类型(string)的组合标识 10: 字符串长度(16字节) 后续数字: " user@example.com " 的ASCII编码</p>

步骤3: 网络传输	二进制数据通过 TCP 发送到 0.0.0.0:50051
传输格式	[gRPC头部] [Protobuf二进制数据]

步骤4:	Node.js 服务端处理
核心机制:	server.addService() 注册的服务处理管道 (gRPC 框架会根据注册的 proto 服务自动处理)
具体流程:	<div><div>服务注册:</div><div><div>Javascript</div><div><pre>server.addService(message_proto.VerifyService.service, // 服务定义 { GetVerifyCode: GetVerifyCode } // 方法实现)</pre></div></div><div><div>message_proto.VerifyService.service 包含:</div><div><ul style="list-style-type: none">• 方法名: GetVerifyCode• 请求类型: GetVerifyReq• 响应类型: GetVerifyRsp</div></div></div> <div><div>自动反序列化:</div><div><ol style="list-style-type: none">1. 接收二进制数据2. 查找注册的 VerifyService 服务3. 找到 GetVerifyCode 方法对应的请求类型 GetVerifyReq4. 按 proto 定义解析二进制数据</div><div><div>Javascript</div><div><pre>// gRPC框架内部伪代码 const requestType = serviceDescriptor.GetVerifyCode.requestType; const deserialized = requestType.deserialize(requestData); call.request = deserialized;</pre></div></div><div><div>字段访问:</div><div><div>Javascript</div><div><pre>// 因为proto定义中有 email 字段 console.log(call.request.email); // "user@example.com"</pre></div></div></div></div>

步骤5:	邮件发送
Javascript	<pre>let mailOptions = { to: call.request.email, // 直接使用反序列化后的值 // ... };</pre>

》》》前面我们了解到，call.request.email 指向了 proto 中的 email，即 C++ 代码（GateServer中）为 proto 指定的 email。那么为什么 call.request.email 可以访问得到 GateServer 传递的 email 信息呢？

Javascript	<pre>// 因为proto定义中有 email 字段 console.log(call.request.email); // "user@example.com"</pre>
为什么这里的 call.request.email 中的数据就是 "user@example.com" ??	

这是因为 Verify Server 端从 proto buffer 中获得了数据，并且根据 proto 中定义的规则自动地构建了对象

Grpc 框架动态的创建对象:	<pre>// 框架内部伪代码 const RequestClass = message_proto.GetVerifyReq; const request = new RequestClass(); // 反序列化二进制数据 request.deserialize(binaryData); // 传递给处理函数 handler(call = { request }, callback);</pre>
(call.request 实际上是 GetVerifyReq 的实例)	<pre>// 编译生成的代码 (message_proto.js) class GetVerifyReq { constructor() { this.email = ""; } set_email(value) { this.email = value; } get_email() { return this.email; } }</pre>

》》》关于 proto 定义在代码中的体现（比如为什么代码这样设计？和 proto 中的定义有什么关系？）

》》proto 定义



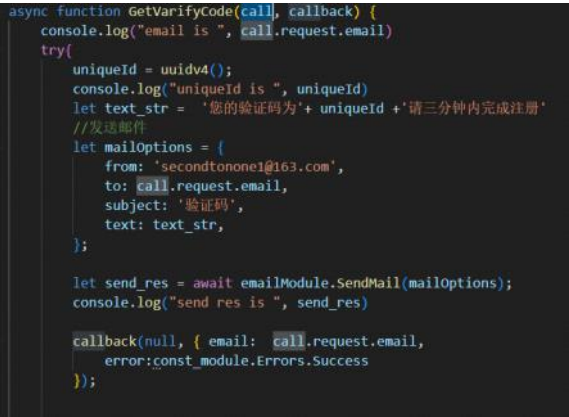
```
syntax = "proto3";
package message;

service VerifyService {
  rpc GetVerifyCode (GetVerifyReq) returns (GetVerifyResp) {}
}

message GetVerifyReq {
  string email = 1;
}

message GetVerifyResp {
  int32 error = 1;
  string email = 2;
  string code = 3;
}
```

Proto 文件中的定义在代码中的映射：

Proto 元素	对应于JavaScript中的代码（图像）	作用
service VerifyService	message_proto.VerifyService.service 	服务描述对象
rpc GetVerifyCode	{ GetVerifyCode: GetVerifyCode } 	方法实现映射
message GetVerifyReq	call.request 	请求对象
message GetVerifyResp	callback(null, {...})	响应对象

```

async function GetVerifyCode(call, callback) {
  console.log("email is ", call.request.email)
  try{
    uniqueId = uuidv4();
    console.log("uniqueId is ", uniqueId)
    let text_str = '您的验证码为'+ uniqueId +'请三分钟内完成注册'
    //发送邮件
    let mailOptions = {
      from: 'secondtonone1@163.com',
      to: call.request.email,
      subject: '验证码',
      text: text_str,
    };

    let send_res = await emailModule.SendMail(mailOptions);
    console.log("send res is ", send_res)

    callback(null, { email: call.request.email,
      error:const_module.Errors.Success
    });
  }
}

```

》》服务注册函数 和 服务实现函数的 设计以及解析

1. 服务实现函数 GetVerifyCode(call, callback)	Javascript				
	<pre> async function GetVerifyCode(call, callback) { // 函数体 } </pre>				
参数设计原理:	<table> <tr> <td>call <u>对象</u>: 封装了 RPC 调用的所有信息</td><td> <ul style="list-style-type: none"> call.request: 反序列化后的请求对象 (对应 GetVerifyReq) call.metadata: 客户端元数据 call.cancel(): 取消调用的方法 </td></tr> <tr> <td>callback <u>函数</u>: 用于发送响应</td><td> 签名: callback(error, response) <ul style="list-style-type: none"> error: 服务端错误 (通常为 null) response: 响应对象 (对应 GetVerifyRsp) </td></tr> </table>	call <u>对象</u> : 封装了 RPC 调用的所有信息	<ul style="list-style-type: none"> call.request: 反序列化后的请求对象 (对应 GetVerifyReq) call.metadata: 客户端元数据 call.cancel(): 取消调用的方法 	callback <u>函数</u> : 用于发送响应	签名: callback(error, response) <ul style="list-style-type: none"> error: 服务端错误 (通常为 null) response: 响应对象 (对应 GetVerifyRsp)
call <u>对象</u> : 封装了 RPC 调用的所有信息	<ul style="list-style-type: none"> call.request: 反序列化后的请求对象 (对应 GetVerifyReq) call.metadata: 客户端元数据 call.cancel(): 取消调用的方法 				
callback <u>函数</u> : 用于发送响应	签名: callback(error, response) <ul style="list-style-type: none"> error: 服务端错误 (通常为 null) response: 响应对象 (对应 GetVerifyRsp) 				

2. 服务注册函数 addService(service, implementations)	javascript <pre>server.addService(message_proto.VerifyService.service, { GetVerifyCode: GetVerifyCode })</pre>																															
参数要求:	<table><tr><th>参数</th><th>类型</th><th>来源</th><th>要求</th></tr><tr><td>1: service</td><td>服务描述对象</td><td>proto 编译生成</td><td>必须与 proto 定义完全匹配</td></tr><tr><td>2: implementations</td><td>方法映射对象</td><td>开发者实现</td><td>方法名必须与 proto 中 rpc 方法名一致</td></tr></table>				参数	类型	来源	要求	1: service	服务描述对象	proto 编译生成	必须与 proto 定义完全匹配	2: implementations	方法映射对象	开发者实现	方法名必须与 proto 中 rpc 方法名一致																
参数	类型	来源	要求																													
1: service	服务描述对象	proto 编译生成	必须与 proto 定义完全匹配																													
2: implementations	方法映射对象	开发者实现	方法名必须与 proto 中 rpc 方法名一致																													
参数使用规范	<table><tr><td>1. 请求对象 call.request</td><td colspan="3">javascript<pre>console.log("email is ", call.request.email)</pre></td></tr><tr><td>要求:</td><td colspan="3"><ul style="list-style-type: none">• 字段必须与 proto 定义一致• 类型必须匹配 (此处 email 为 string)• 访问未定义字段将返回 undefined</td></tr><tr><td>2. 响应回调 callback(null, response)</td><td colspan="3">javascript<pre>callback(null, { email: call.request.email, error: const_module.Errors.Success })</pre></td></tr><tr><td>要求:</td><td colspan="3"><table><tr><th>字段</th><th>proto 定义</th><th>代码要求</th></tr><tr><td>error</td><td>int32 error = 1</td><td>必须是整数</td></tr><tr><td>email</td><td>string email = 2</td><td>必须是字符串</td></tr><tr><td>code</td><td>string code = 3</td><td>可选 (未提供则设为空字符串)</td></tr></table></td></tr></table>				1. 请求对象 call.request	javascript <pre>console.log("email is ", call.request.email)</pre>			要求:	<ul style="list-style-type: none">• 字段必须与 proto 定义一致• 类型必须匹配 (此处 email 为 string)• 访问未定义字段将返回 undefined			2. 响应回调 callback(null, response)	javascript <pre>callback(null, { email: call.request.email, error: const_module.Errors.Success })</pre>			要求:	<table><tr><th>字段</th><th>proto 定义</th><th>代码要求</th></tr><tr><td>error</td><td>int32 error = 1</td><td>必须是整数</td></tr><tr><td>email</td><td>string email = 2</td><td>必须是字符串</td></tr><tr><td>code</td><td>string code = 3</td><td>可选 (未提供则设为空字符串)</td></tr></table>			字段	proto 定义	代码要求	error	int32 error = 1	必须是整数	email	string email = 2	必须是字符串	code	string code = 3	可选 (未提供则设为空字符串)
1. 请求对象 call.request	javascript <pre>console.log("email is ", call.request.email)</pre>																															
要求:	<ul style="list-style-type: none">• 字段必须与 proto 定义一致• 类型必须匹配 (此处 email 为 string)• 访问未定义字段将返回 undefined																															
2. 响应回调 callback(null, response)	javascript <pre>callback(null, { email: call.request.email, error: const_module.Errors.Success })</pre>																															
要求:	<table><tr><th>字段</th><th>proto 定义</th><th>代码要求</th></tr><tr><td>error</td><td>int32 error = 1</td><td>必须是整数</td></tr><tr><td>email</td><td>string email = 2</td><td>必须是字符串</td></tr><tr><td>code</td><td>string code = 3</td><td>可选 (未提供则设为空字符串)</td></tr></table>			字段	proto 定义	代码要求	error	int32 error = 1	必须是整数	email	string email = 2	必须是字符串	code	string code = 3	可选 (未提供则设为空字符串)																	
字段	proto 定义	代码要求																														
error	int32 error = 1	必须是整数																														
email	string email = 2	必须是字符串																														
code	string code = 3	可选 (未提供则设为空字符串)																														

为什么需要遵从这样的设计?

这是 gRPC Node.js 库的标准接口设计, 遵循了 gRPC 的通用模式:	1. 统一处理所有 RPC 调用的入口 2. 分离请求和响应处理 3. 支持异步操作 (如您的 async 函数)
---	---

》》》》 java script 中的重命名导入?

```
const {v4: uuidv4} = require('uuid')
```

{v4: uuidv4} 是一个解构赋值语法, 它的作用是从 require('uuid') 导入的模块中提取出名为 v4 的属性, 并将其赋值给一个新的变量 uuidv4。
(即将 uuid 模块中的 v4 导出重命名为 uuidv4, 这意味着你可以通过 uuidv4 来引用 v4。)

》》》》 sendMail 返回什么值? sendRes 变量的类型是什么? await 有什么作用?

```
try
{
  let uniqueID = uuidv4(); // TODO: let 会导致重复声明?
  console.log('sending verification code to mail...(code is %s)', uniqueID)
  let textStr = '您的验证码为: ' + uniqueID + ', 请在规定时间内完成注册, 超时验证码则会失效。';

  // 发送邮件 (mailOptions 为发送邮件时指定的信息或规范)
  let mailOptions =
  {
    from: 'ffffishyz@163.com',
    to: call.request.email,
    subject: '验证码',
    text: textStr
  }
  let sendRes = await emailModule.SendMail(mailOptions);
}
```

1. SendMail 函数返回值:

SendMail 函数返回一个 Promise, 并且在 transport.sendMail 的回调中, 通过 resolve(info.response) 返回邮件发送成功的响应。

- 当邮件发送成功时, resolve(info.response) 被调用, Promise 会被标记为成功, info.response 会作为 Promise 的返回值传递。
- 如果发生错误, reject(error) 会被调用, Promise 会被标记为失败, 错误信息会被传递。

2. await 的作用:

await 是一个关键字, 它只能在 async 函数中使用, 且作用是等待一个 Promise 对象的解决 (resolve) 或拒绝 (reject), 如果 promise 对象操作进行完成, 则进行下一步代码的操作。

》》》》 callback 是什么? 有什么作用? 填入什么参数?

```
7  async function GetVerifyCode(call, callback)
8  {
9      console.log(call.request.email, " is handling");
10     try
11     {
12         let uniqueID = uuidv4();
13         console.log('sending verification code to mail...(code is %s)', uniqueID)
14         let textStr = '您的验证码为: ' + uniqueID + ', 请在规定时间内完成注册, 超时验证码则会失效。';
15
16         // 发送邮件 (mailOptions 为发送邮件时指定的信息或规范)
17         let mailOptions =
18         {
19             from: 'ffffishyz@163.com',
20             to: call.request.email,
21             subject: '验证码',
22             text: textStr
23         }
24         let sendRes = await emailModule.SendMail(mailOptions);
25         console.log('Send response is ', sendRes);
26
27         // 如果 try 中的代码正常执行, 则会运行 callback 函数
28         // 该函数会将错误信息设置为null, 并且设置传递结果 (此处包括 email 地址和状态码 error)
29         callback(null, { email: call.request.email, error: constModule.Errors.Success });
30     }
}
```

callback 本身不是一个函数签名, 而是一个 grpc 中的一个回调函数, 它是一个函数的引用。

我们可以理解为它是某种形式的函数参数, 通常传递给另一个函数, 并在特定事件或异步操作完成后被调用。

签名:	callback(error, result);
参数:	• error 是一个参数, 通常用来传递错误信息, 如果没有错误则通常传入 null。

•result 是另一个参数，用于传递操作的结果。

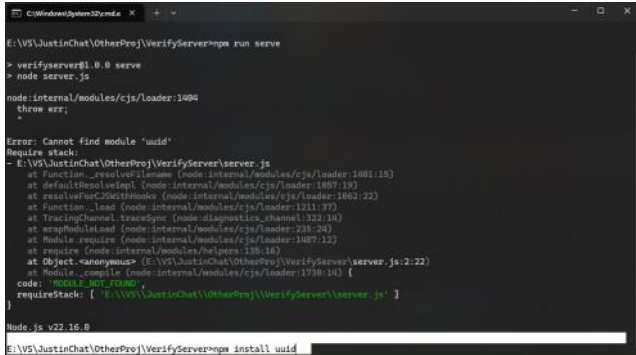
》》》const constModule = require('./const.js')这段代码和const constModule = require('./const')，后者会造成错误？
不会

》》》0.0.0.0: 50051 是本机的 50051 端口吗？

0.0.0.0:50051 并不是指向本机的特定地址，而是表示 绑定到所有可用的网络接口，包括本地地址和外部网络接口。
这意味着你的服务器将能够接收来自本地机器以及局域网（LAN）或外部网络上的客户端的请求。

0.0.0.0:	这是一个通配符地址，表示绑定到所有可用的网络接口，包括本地回环接口（如 127.0.0.1）和任何外部网络接口。
127.0.0.1:	这是本机地址（也称为回环地址），只允许本地计算机上的进程相互通信，外部客户端无法访问。

》》》运行示例：（提示我没有安装 uuid、@grpc/grpc-js，于是我使用 npm 重新安装）



》》另外，包含项目内文件的时候一定要特别标注 './'，表示该文件是在当前目录下的文件。否则会报错。

错误示范	<pre>1 const nodemailer = require('nodemailer'); 2 const configModule = require('config');</pre>
正确示范	<pre>const nodemailer = require('nodemailer'); const configModule = require('./config');</pre>

成功启动	
成功运行。	<pre>PS E:\VS\JustinChat\OtherProj\VerifyServer> npm run serve > verifyserver@1.0.0 serve > node server.js Grpc server started! (node:9760) DeprecationWarning: Calling start() is no longer necessary. It can be safely omitted. (Use 'node --trace-deprecation ...' to show where the warning was created) 3480966311@qq.com is handling sending verification code to mail...(code is 9f55b34b-8f25-4123-b460-cd209b80ecd9) 邮件已经成功发送 250 Mail OK queued as gzga-smtp-mtada-g1-4,_____wCkjs9djlo_pVUFA--..2777152 1748596286 Send response is 250 Mail OK queued as gzga-smtp-mtada-g1-4,_____wCkjs9djlo_pVUFA--..2777152 1748596286</pre>

代码上有很多需要注意的地方，不要犯小错误，实测代码可行。 :)

----- Ep 10 iocontext 连接池 -----

》》》这一节的任务：

1.为 incontext 创建一个对象池	(AsioIOServicePool)
2.为 grpc 服务中的 VerifyService::Stub 类型变量创建一个连接池	(RPConPool)

》》》什么是高并发的访问？池的概念是什么？

池的概念：	池是一种通过维护一组资源来管理这些资源的设计模式。在池中，资源通常是有限的，可以是线程、数据库连接、网络连接、内存缓存等。池的目的是通过复用资源来避免每次需要时都重新创建和销毁资源，从而减少开销，提高系统的性能和响应速度。
池的管理通常包括：	1. 资源预先创建：池在初始化时创建一定数量的资源，资源不会立即销毁，直到池不再需要这些资源。 2. 资源复用：当请求资源时，池会分配一个可用的资源，而不是每次都新创建。任务完成后，资源被归还给池，而不是销毁。 3. 资源回收：池会定期回收和清理不再使用的资源，保持池中资源的有效性和可用性
常见的池类型：	1. 线程池（Thread Pool）：用来管理和复用线程，避免频繁创建和销毁线程，适用于需要执行大量并发任务的场景。 2. 连接池（Connection Pool）：用于管理数据库或其他网络连接，通过复用连接避免频繁创建和销毁数据库连接的性能损耗。 3. 对象池（Object Pool）：用于管理复用某些资源对象，类似于数据库连接池或线程池。 4. 内存池（Memory Pool）：用于高效管理内存的分配与释放，避免频繁的内存申请和释放所带来的性能问题。
池解决的问题：	◦ 减少资源开销 ：池通过复用已经创建的资源，避免了每次请求都需要重新创建和销毁资源，减少了系统的开销。 ◦ 提高性能和响应速度 ：通过合理管理资源池中的资源，避免了在高并发情况下系统因频繁创建和销毁资源而导致性能瓶颈。 ◦ 控制并发量 ：池可以通过限制池中最大资源数量来控制系统并发的最大承载能力，从而防止过多的资源占用导致系统崩溃或响应过慢。

什么是高并发的访问：

- **定义**：高并发是指系统需要处理大量的同时到达的请求，通常要求系统能够高效地进行任务分配和资源管理。常见的高并发场景包括大规模的 Web 服务、金融交易平台、即时通讯应用等。
- **特征**：高并发访问通常意味着短时间内大量请求涌入系统，系统需要具备快速处理请求的能力，并且保持响应时间低。高并发不仅仅是请求数量的多，更重要的是并发请求之间的资源共享与调度。

》》》池可以使用 std::vector 来构建，也可以使用 std::queue 来构建，哪一个更合适？

1. std::vector:

std::vector（动态数组）提供高效的随机访问，并支持按需扩展。如果池中的元素数量是可变的，且你需要频繁访问某个元素，vector 可以是一个不错的选择。

优点:	◦ 支持高效的随机访问，适用于频繁访问特定元素的场景。 ◦ 可以动态增长，灵活应对池中元素的变化。 ◦ 在尾部添加和删除元素的时间复杂度为 O(1)。
缺点:	◦ 在中间插入和删除元素时，时间复杂度是 O(n)，这对于频繁插入/删除的池来说可能会影响性能。 ◦ 随机访问时可能需要多次重新分配内存，造成内存碎片化。
适用场景:	◦ 如果池中的元素可以随机访问，且不要求严格的先进先出（FIFO）顺序，vector 可以很好地应对。 ◦ 如果池大小是动态变化的，且需要按位置访问元素，vector 也可以灵活应对。

2. std::queue:

std::queue 是一个基于容器的适配器（队列），它提供先进先出（FIFO）的队列操作。底层通常使用 std::deque 或 std::list 来实现，因此它在前端和尾部的插入和删除是非常高效的。

优点:	◦ 高效的 FIFO 操作，特别适用于需要按照特定顺序处理任务的池（例如线程池）。 ◦ 在队列两端进行操作（入队和出队）的时间复杂度为 O(1)。 ◦ 操作简单，语义清晰，适合处理顺序任务。
缺点:	◦ 不支持随机访问，意味着只能从队头出队和队尾入队，无法灵活访问池中的任意元素。 ◦ 如果需要访问队列中间的元素，必须先出队。
适用场景:	适用于任务处理队列（例如线程池、任务池），其中任务按顺序执行，且不需要随时随机访问池中的元素。

总结：

如果你需要 先进先出（FIFO）顺序，并且不需要随机访问池中的元素，queue 是更合适的选择。	如果是实现一个 线程池，通常使用 queue 作为任务队列，因为线程池中的任务是按照顺序执行的。
--	--

如果你需要 随机访问池中的元素，或者池的大小经常变化，且你不关心先进先出的顺序，vector 可能更适合。

如果是 对象池，比如对象的复用池，使用 vector 会更灵活，因为可以更方便地对池中的对象进行管理和扩展。

》》》关于私有成员的一些理解：

关于 m_Works 的理解	<pre>public: using WorkPtr = std::unique_ptr<boost::asio::executor_work_guard<boost::asio::io_context::executor_type>>; private: std::vector<boost::asio::io_context> m_IOServices; std::vector<WorkPtr> m_Works; std::vector<std::thread> m_Threads; std::size_t m_NextIOService; };</pre>
m_works 存储的 io_context::work 对象是保持线程运行的关键机制：	// 在构造函数中我们这样设计： _works[i] = std::unique_ptr<Work>(new Work(_ioServices[i]));
boost::asio::executor_work_guard<boost::asio::io_context::executor_type> 类型变量的作用：	<div>生存期控制：<ul style="list-style-type: none">• 当 work 对象存在时，io_context 认为有“待处理工作”• 即使没有实际任务，io_context.run() 也不会返回</div> <div>// 伪代码展示原理 void io_context::run() { while (has_work !queue_empty) { // work对象保持has_work=true process_events(); } }</div>
	防止线程退出： <ul style="list-style-type: none">• 没有 work 时: run() 立即返回 → 线程结束• 有 work 时: run() 保持阻塞 → 线程持续运行
	图例： <div></div>

关于m_Threads 的理解	<pre>private: std::vector<boost::asio::io_context> m_IOServices; std::vector<WorkPtr> m_Works; std::vector<std::thread> m_Threads; std::size_t m_NextIOService; };</pre>
线程并行处理机制：（代码示例）	
并行架构：	// 线程创建代码 _threads.emplace_back([this, i]() { _ioServices[i].run(); // 每个线程专属一个io_context });
一对一绑定：	<ul style="list-style-type: none">• 每个线程绑定一个独立的 io_context• 线程T1 ↔ io_context1• 线程T2 ↔ io_context2
无锁并行：	<div></div>
事件处理隔离：	<ul style="list-style-type: none">• 每个 io_context 有自己的事件队列• 线程只处理自己绑定的 io_context 的事件• 天然避免线程竞争（通过这样的设计，我们不需要手动使用互斥锁来保证线程安全）

对于 m_nextIOService 的理解	<pre>private: std::vector<boost::asio::io_context> m_IOServices; std::vector<WorkPtr> m_Works; std::vector<std::thread> m_Threads; std::size_t m_NextIOService; };</pre>
作用：	实现轮询 (round-robin) 负载均衡

	<ul style="list-style-type: none">• 每次调用 GetIOService() 返回下一个可用的 io_context• 确保任务均匀分配到所有 I/O 服务，避免总是在重复处理某一个 io context，从而避免单个 io_context 过载
工作流程:	<pre>boost::asio::io_context& AsioIOServicePool::GetIOService() { if (m_NextIOService == m_IOServices.size()) m_NextIOService = 0; auto& service = m_IOServices[m_NextIOService++]; return service; }</pre>

》》》如何让不同的 io_context 运行在不同的线程上?

代码示例:	<pre>// 遍历 ioservice.size() 创建多个线程，同时使用回调函数，为每个线程内部都启动 ioservice for(std::size_t i = 0; i < m_IOServices.size(); i++) { m_Threads.emplace_back([this, i](){ m_IOServices[i].run(); }); }</pre>
-------	---

问题:

emplace_back 插入的函数会在什么时候运行呢?

为什么这样可以做到为每一个 thread 分配一个 context?

为什么对 io_context 类型变量使用 .run() 函数?

1. lambda 表达式什么时候运行?

什么时候执行:	emplace_back 插入 lambda 后，emplace_back 插入的函数 (lambda 表达式) 会在 std::vector::emplace_back 调用时立即执行。
在那个线程中执行:	同时，lambda 表达式会在被插入的线程中执行，而不是在调用 emplace_back 的线程中执行。
作用:	因为将 lambda 表达式作为参数传递给了 std::thread，所以每次调用 emplace_back 时，都会为该线程创建一个新的执行环境。

2. 为什么可以为每个线程分配一个 io_context?

作用:	通过 for 循环，每次 emplace_back 调用时，都会创建一个新的线程来运行相应的 io_context。（比如运行 .run() 函数）
独立性:	每个 lambda 表达式都绑定了不同的 io_service[i]，因此每个线程在执行时会处理不同的 io_context。换句话说，这种操作确保了每个线程对应一个独立的 io_context。每个 io_context 是独立的，因此它们各自处理自己的事件队列，而不会相互干扰，实现多线程并行处理。

3. ioservices[i].run() 的作用:

ioservices[i].run()	在每个线程中调用 run() 方法，使得该线程去处理它所绑定的 io_context 中的异步事件。
---------------------	--

》》》Stop() 函数的工作原理

```
//因为仅仅执行work.reset并不能让iocontext从run的状态中退出
//当iocontext已经绑定了读或写的监听事件后，还需要手动stop该服务。
for (auto& work : _works) {
    //把服务先停止
    work->get_io_context().stop();
    work.reset();
}

for (auto& t : _threads) {
    t.join();
}
```

为什么需要手动 stop()? 需要解释三点:

第一:	join() 确保了主线程会等待所有工作线程执行完毕后再退出。join() 是一个阻塞操作，它会使主线程等待子线程完成。
首先我们需要知道 std::thread 的成员函数 join() 的功能:	这样，所有的异步操作都得以完整地终结，避免在操作未完全结束时就退出程序。

第二:	• 销毁 work 只是允许 run() 退出
reset() 的局限性:	• 但已注册的异步操作 (如 socket 监听) 会阻止退出

	仅仅执行 work.reset() 不能让 io_context 完全停止，因为 io_context 可能仍然处于运行状态，特别是当 io_context 上有未处理的异步事件（如读写操作）时。work->reset() 用于取消与 io_context 关联的 work 对象，并解除对 io_context.run() 的阻塞。
第三： stop() 的必要性：	<ul style="list-style-type: none">• 强制取消所有未完成操作• 中断 run() 的阻塞状态• 确保线程能及时退出 <p>stop() 方法可以通知 io_context 停止进一步处理新的异步事件。否则，io_context.run() 可能会继续阻塞并处理剩余的事件。</p> <p>如果没有显式调用 stop()，即使 work.reset() 之后，io_context 可能依然会尝试继续处理异步事件（如读取、写入等），这会导致线程一直阻塞在 run() 上（主线程一直在等待子线程中的 .run() 函数运行完成），让主线程无法退出。</p>

》》那么 stop() 函数和 reset() 函数的顺序可以调换吗？

- 如果先调用 work.reset()，work_guard 就会解除对 io_context 的阻塞，这可能导致 io_context 在调用 stop() 时已经没有未完成的任务，或者在 stop() 后立即退出，不会等待其他异步操作完成。
- 如果先调用 work->get_io_context().stop()，stop() 会正确地停止 io_context，然后 work.reset() 会确保阻塞解除，使得 io_context.run() 能顺利退出。

》》》RAII ？ ？

RAII (Resource Acquisition Is Initialization, 资源获取即初始化) 是 C++ 的核心编程范式，通过对象的生命周期自动管理资源（如内存、文件句柄、锁等）。其核心思想是：

资源获取 = 对象构造	在构造函数中获取资源（分配内存、打开文件、加锁等）
资源释放 = 对象析构	在析构函数中释放资源（自动调用，确保不泄漏）
概述：	RAII 就是： <ul style="list-style-type: none">1. 将资源绑定对象生命周期2. 利用构造函数获取资源3. 利用析构函数释放资源
示例：	<div><p>经典 RAII 案例</p><p>1. 智能指针</p><pre>cpp std::unique_ptr<Object> ptr(new Object()); // 自动释放内存</pre><p>2. 文件操作</p><pre>cpp std::ofstream file("log.txt"); // 析构时自动关闭</pre><p>3. 锁管理</p><pre>cpp std::lock_guard<std::mutex> lock(mutex); // 析构时自动解锁</pre></div>

因此，先前的代码也可以这样更改：（在boost 1.88.0 中）

```
void AsioIOServicePool::Stop()
{
    // 停止所有 io_context
    for (auto& ioContext : m_IOServices)
    {
        ioContext.stop();
    }

    // 重置所有的工作对象
    for (auto& work : m_Works)
    {
        work->reset();
    }

    // 等待线程结束，可以考虑设置超时避免阻塞过久
    for (auto& thread : m_Threads)
    {
        if (thread.joinable())
        {
            thread.join(); // 确保线程完成
        }
    }
}
```

```
void AsioIOServicePool::Stop()
{
    for(auto& ioContext : m_IOServices)
    {
        ioContext.stop(); // 停止所有 IO 服务
    }

    m_Works.clear(); // 销毁 work_guard

    for(auto& thread : m_Threads)
    {
        if (thread.joinable())
            thread.join(); // 等待线程结束
    }
}
```

为什么你的 Stop() 函数需要 RAII 式修改？

原代码问题：

```
cpp
// 非RAII风格：需手动管理work.reset()
for (auto& work : m_Works) {
    work->reset(); // 依赖手动调用
}
```

修正后 (RAII风格)：

```
cpp
m_Works.clear(); // 自动调用所有元素的析构函数
```

• RAII 的优势：即使 Stop() 函数中间抛出异常，已销毁的 work_guard 也能确保部分资源释放。

```
thread.join(); // 确保线程完成
```

为什么你的 Stop() 函数需要 RAII 式修改？

原代码问题：

```
cpp
// 非RAII风格，需手动管理work.reset()
for (auto& work : m_Works) {
    work->reset(); // 依赖手动调用
}
```

修正后（RAII风格）：

```
cpp
m_Works.clear(); // 自动调用所有元素的析构函数
```

• **RAII 的优势**：即使 **Stop()** 函数中间抛出异常，已销毁的 **work_guard** 也能确保部分资源释放。

注意：

但是使用 clear()可能会导致 io_context 提前退出，除非你很确定没有剩余的异步操作需要处理。然而使用 reset()更安全，这可以确保 io_context 在所有任务完成之前不会退出，因此推荐使用。

```
>>>> 对于函数的理解
>>>> static ConfigMgr& Inst() {
>>>>     static ConfigMgr cfg_mgr;
>>>>     return cfg_mgr;
>>>> }
```

1. 可见性：

- cfg_mgr 是一个 静态局部变量，其可见性仅限于 Inst() 函数内部。换句话说，cfg_mgr 只能通过调用 Inst() 函数来访问，不能在函数外部直接访问。
- Inst() 函数的返回值是 cfg_mgr 的引用，因此外部可以通过 Inst() 获取到该对象的引用，但直接操作 cfg_mgr 变量本身是不可见的。

2. 生命周期：

- cfg_mgr 是一个 静态局部变量，它的生命周期与程序的运行周期同步。它会在第一次调用 Inst() 函数时被初始化，并且直到程序结束时才会销毁。
- 首次调用 Inst() 时，cfg_mgr 会被初始化，并且这个实例会在整个程序的生命周期内一直存在。

3. 线程同步：

- 在 C++11 及以后版本，静态局部变量的初始化是线程安全的。这意味着当多个线程并发调用 Inst() 时，只有一个线程会初始化 cfg_mgr，其他线程会等待直到 cfg_mgr 初始化完成。
- 一旦 cfg_mgr 被初始化，所有线程都能共享这个同一个实例，且它在整个程序运行期间都不会重新初始化。