

----- Ep9 NodeJs 实现邮箱验证服务 -----

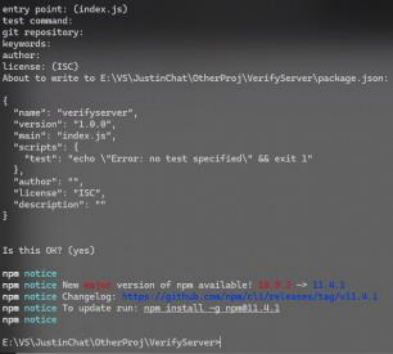
》》》什么是 npm ？

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中运行指令

npm init

一路回车



1.变量声明	JavaScript 使用 var、let 和 const 来声明变量。 <div><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></div>	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： <div><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></div>	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	<div><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></div>	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：	let a = 5, b = 10; console.log(a + b); // 15 console.log(a > b); // false console.log(a === 5); // true						

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）

```
protojs > ...
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：	<div>在 Protobuf 中，oneof 的语法如下：</div> <div>message Example { oneof response { int32 id = 1; string name = 2; bool is_active = 3; } }</div> <div>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。</div>				
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：	<div>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。</div> <div>message Message { oneof content { string text = 1; bytes image = 2; bytes video = 3; } }</div> <div>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。</div>				

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
```

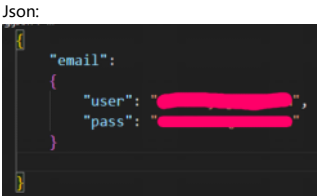
protoDescriptor.message:	假设在 message.proto 文件中定义了一个叫做 message 的服务或者消息类型，这一行代码从 protoDescriptor 中获取该消息类型的定义。
message_proto:	这是一个包含 message 服务或消息类型的对象，允许你在代码中使用它，比如创建消息实例、调用服务等。

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:



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。 <ul style="list-style-type: none">• '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。<u>utf8 可以确保返回的内容是字符串类型，而不是 Buffer 对象。</u>
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。 <ul style="list-style-type: none">• 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。
-------	--

	<div>2. 等待任务完成：你可以通过调用 <code>get()</code> 来等待任务完成，并获取它的结果。<code>get()</code> 会阻塞调用线程，直到异步任务完成。</div> <div>3. 处理异常：如果异步任务在执行过程中抛出异常，<code>get()</code> 会重新抛出该异常，允许你在主线程中处理。</div>						
常见用法：	<div>通常，<code>std::future</code> 与 <code>std::async</code> 配合使用，<code>std::async</code> 用于启动一个异步任务，<code>std::future</code> 用来接收任务的结果。</div> <div>示例代码：</div> <div>#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; } 1. <code>std::async</code>：用来启动一个异步任务，返回一个 <code>std::future</code> 对象。这个对象代表了将来某个时刻的结果。 2. <code>result.get()</code>：<code>get()</code> 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。<code>get()</code> 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 <code>get()</code> 会重新抛出这个异常，因此可以在调用 <code>get()</code> 的地方捕获并处理异常。)</div>						
常用成员函数：	<table><tr><td><code>get()</code>：</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><code>valid()</code>：</td><td>检查 <code>std::future</code> 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><code>wait()</code>：</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>	<code>get()</code> ：	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。	<code>valid()</code> ：	检查 <code>std::future</code> 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）。	<code>wait()</code> ：	等待异步任务完成，但不会返回结果，仅用于同步操作。
<code>get()</code> ：	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。						
<code>valid()</code> ：	检查 <code>std::future</code> 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）。						
<code>wait()</code> ：	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	<code>std::promise</code> 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 <code>std::future</code> 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	<div>1. 创建 <code>promise</code> 和 <code>future</code></div> <div>#include <future> std::promise<int> promise_obj; std::future<int> future_obj = promise_obj.get_future(); 2. 设置值（生产者线程） 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"))); 3. 获取值（消费者线程） int result = future_obj.get(); // 阻塞直到值被设置</div>
典型应用场景	<div>1. 线程间传递异步结果</div> <div>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; 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; } 2. 异常传递 如果生产者线程发生错误，可以通过 <code>set_exception</code> 传递异常： try { // 可能抛出异常的操作</div>

	<pre> } catch (...) { promise_obj.set_exception(std::current_exception()); } }</pre>
关键特性	<p>1. 单通信</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。

》》》 javascript 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 <code>executor</code> 函数作为参数，这个函数有两个参数：<code>resolve</code> 和 <code>reject</code>。</p> <ul style="list-style-type: none">• <code>resolve(value)</code>：表示异步操作成功，<code>value</code> 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，<code>reason</code> 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， <code>resolve</code> 和 <code>reject</code> 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： <code>resolved</code> （操作成功）或 <code>rejected</code> （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 }
```

SendMail(mailOptions) { }

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) 
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: 
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:

- 在许多 gRPC 示例或默认设置中, 端口 50051 被用作默认的服务器端口, 提供远程过程调用服务。
- 例如, 当你运行一个 gRPC 服务时, 服务器通常会监听 50051 端口以接受来自客户端的请求。

》》》》 验证码的发送流程和一些具体细节:

》》》》 在我们的设计中, email 信息是怎样流通的? 特别是 verify server, 他从哪里获取到 email? 和 proto 有关系吗?

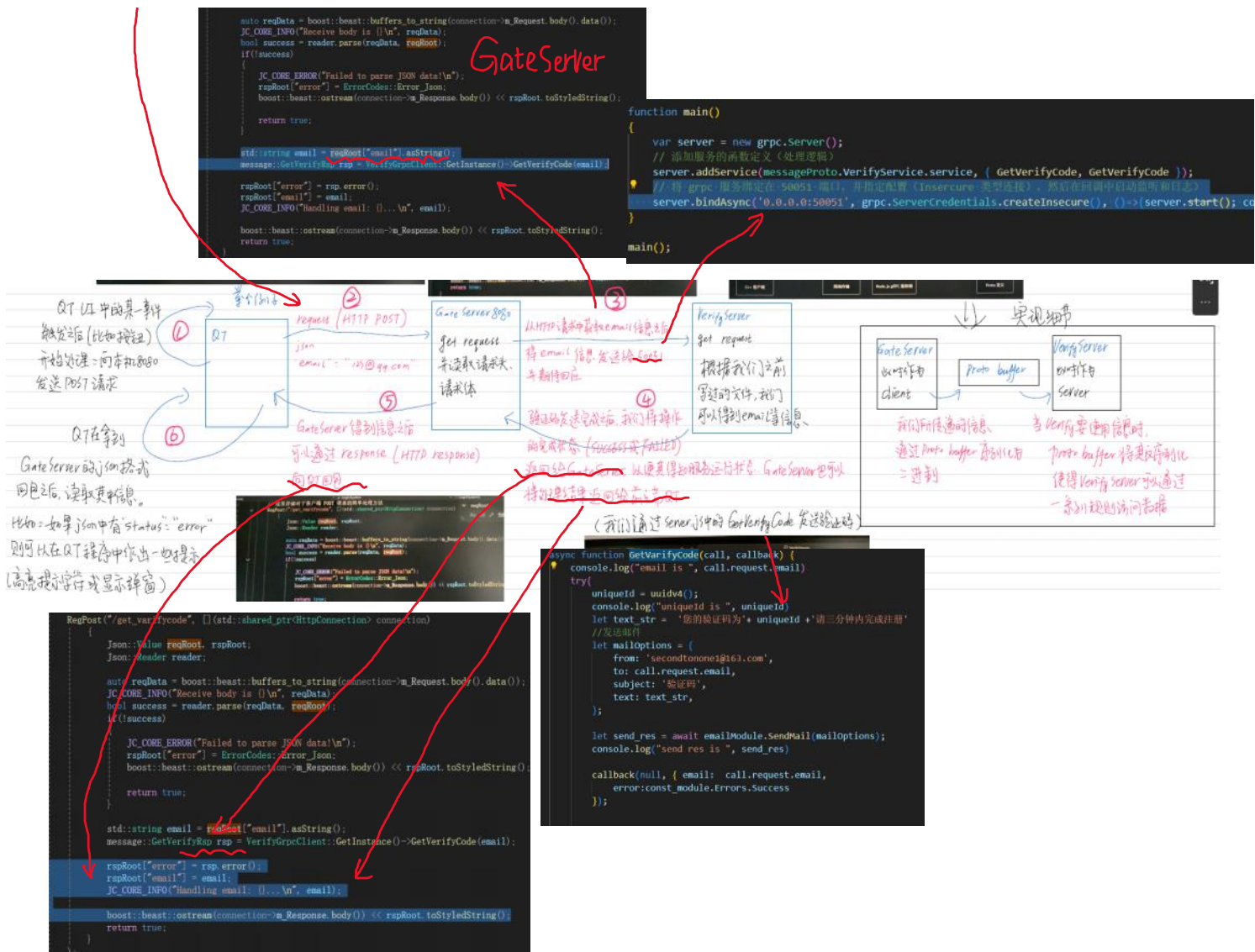


```
void RegisterDialog::on_get_code_clicked()
{
    auto emailStr = ui->email_edit->text();

    QRegularExpression regex(R"((\w+)(\.[_]?(\w+)@(\w+)\.(\w+))+)");
    bool matchResult = regex.match(emailStr).hasMatch(); // Don't create temporary QRegularExpression objects. Use a static QRegularExpression.
    if(!matchResult)
    {
        QJsonObject jsonObj;
        jsonObj["email"] = emailStr;
        HttpMgr::GetInstance()->PostHttpRequest(QUrl(gateUrlPrefix + "/get_verifycode"), 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_verifycode", [](std::shared_ptr<HttpConnection> connection)
        {
            Json::Value reqRoot; rspRoot;
            Json::Reader reader;

            auto reqData = boost::boost::buffers::to_string(connection->Request.body().data());
            JC_CORE_INFO("Receive body is [%s]", reqData);
            bool success = reader.parse(reqData, reqRoot);
            if(!success)
            {
                JC_CORE_ERROR("Failed to parse JSON data\n");
                reqRoot["error"] = ErrorCode::Error_Json;
            }
        });
    }
}
```

GateServer



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 - 服务协议

```

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



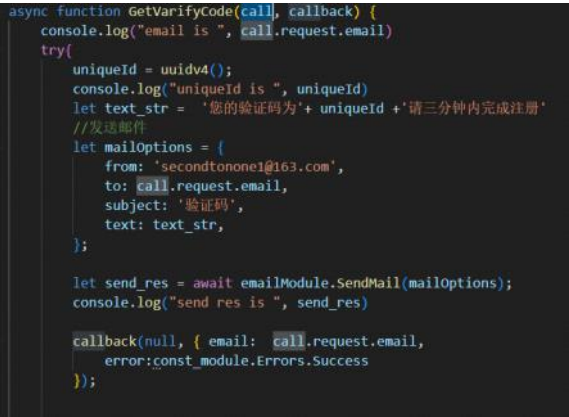
```
syntax = "proto3";
package message;

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

message GetVerifyReq {
  string email = 1;
}

message GetVerifyRsp {
  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 GetVerifyRsp	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><p>签名: callback(error, response)</p><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	<p>签名: callback(error, response)</p> <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	<p>签名: callback(error, response)</p> <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><u>必须与 proto 定义完全匹配</u></td></tr><tr><td>2: implementations</td><td>方法映射对象</td><td>开发者实现</td><td><u>方法名必须与 proto 中 rpc 方法名一致</u></td></tr></table>				参数	类型	来源	要求	1: service	服务描述对象	proto 编译生成	<u>必须与 proto 定义完全匹配</u>	2: implementations	方法映射对象	开发者实现	<u>方法名必须与 proto 中 rpc 方法名一致</u>																
参数	类型	来源	要求																													
1: service	服务描述对象	proto 编译生成	<u>必须与 proto 定义完全匹配</u>																													
2: implementations	方法映射对象	开发者实现	<u>方法名必须与 proto 中 rpc 方法名一致</u>																													
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><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><td colspan="2"></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:	<ol style="list-style-type: none">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: 'fffishyz@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: 'fffishyz@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     }
31 }
```

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')，后者会否造成错误？
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）

```
C:\Windows\System32\cmd.exe X + v
E:\VS\JustinChat\OtherProj\VerifyServer>npm run serve
> verifyserver@1.0.0 serve
> node server.js
node:internal/modules/cjs/loader:1484
  throw err;
  ^
Error: Cannot find module 'uuid'
Require stack:
- E:\VS\JustinChat\OtherProj\VerifyServer\server.js
    at Function._resolveFilename (node:internal/modules/cjs/loader:1481:15)
    at defaultResolveImpl (node:internal/modules/cjs/loader:1657:19)
    at resolveForCJSWithHooks (node:internal/modules/cjs/loader:1662:22)
    at Function.load (node:internal/modules/cjs/loader:1211:37)
    at TracingChannel.traceSync (node:diagnostics_channel:322:14)
    at wrapModuleLoad (node:internal/modules/cjs/loader:235:24)
    at Module.require (node:internal/modules/cjs/loader:1487:12)
    at require (node:internal/modules/helpers:138:16)
    at Object.<anonymous> (E:\VS\JustinChat\OtherProj\VerifyServer\server.js:2:22)
    at Module._compile (node:internal/modules/cjs/loader:1730:14) {
  code: 'MODULE_NOT_FOUND',
  requireStack: [ 'E:\VS\JustinChat\OtherProj\VerifyServer\server.js' ]
}
Node.js v22.16.0
E:\VS\JustinChat\OtherProj\VerifyServer>npm install uuid
```

》》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> <p>// 创建发送邮件的代理</p>

```
npm run serve
Microsoft Windows [版本 10.0.22631.5335]
(c) Microsoft Corporation. 保留所有权利。

E:\VS\JustinChat\OtherProj\VerifyServer>npm run serve
> verifyserver@1.0.0 serve
> node server.js

Grpc server started!
(node:12208) DeprecationWarning: Calling start() is no longer necessary. It can be safely omitted.
(Use 'node --trace-deprecation ...' to show where the warning was created)
```

成功启动

```
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,_____wCXjjs9djlo_pVUFA--.27771S2 1748596286
Send response is 250 Mail OK queued as gzga-smtp-mtada-g1-4,_____wCXjjs9djlo_pVUFA--.27771S2 1748596286
█
```

成功运行。

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。 :)