第 1 章 1

绪　　论

本章首先介绍 GPT 模型的基本概念、发展历程、基本原理，以及 OpenAI API 和生成式 AI 应用的市场前景。在对这些基本信息有了一些初步认识后，接下来介绍本书的内容安排，以便让读者对整体内容和学习目标有更全面的了解。

1.1　AGI 的新时代已经到来

对于 ChatGPT，相信读者或多或少都有所耳闻。在 2023 年里，ChatGPT 作为一种人工智能对话交互产品，凭借其贴近人类行为的交互方式和强大的通用型答案生成能力，迅速席卷各行各业。短短几个月内，各行各业对它的讨论和衍生应用层出不穷。从专业解答和工程优化，到日常琐事和沟通交流，ChatGPT 都能提供相对专业且全面的回答，其示例效果如图 1-1 和图 1-2 所示。

除了基础应用，各行各业也开始将它接入各自的 SOP（Standard Operating Procedure，标准操作流程）中，以代替人工完成一些复杂的重复工作。坦诚且严谨地说，虽然 ChatGPT 的能力尚未达到完全替代人的程度，但结合它的 AI 功能，完成日常学习和提高工作的效率与质量远超过传统方式。AGI（Artificial General Intelligence，通用型人工智能）的新时代已经到来！

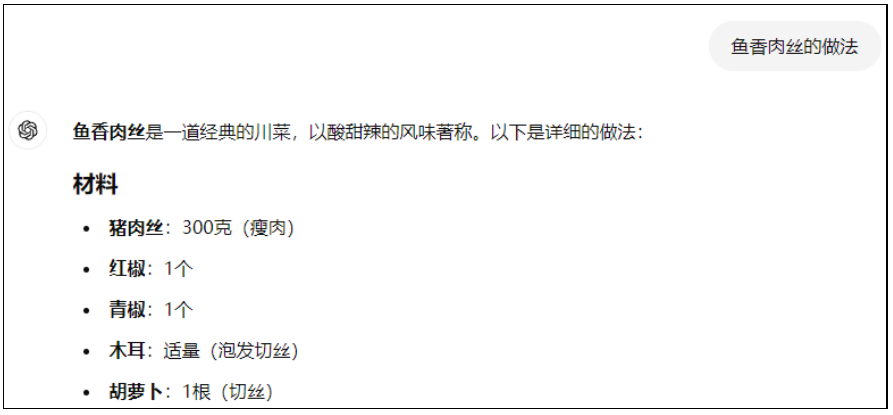

图 1-1　ChatGPT 的使用示例（日常沟通）

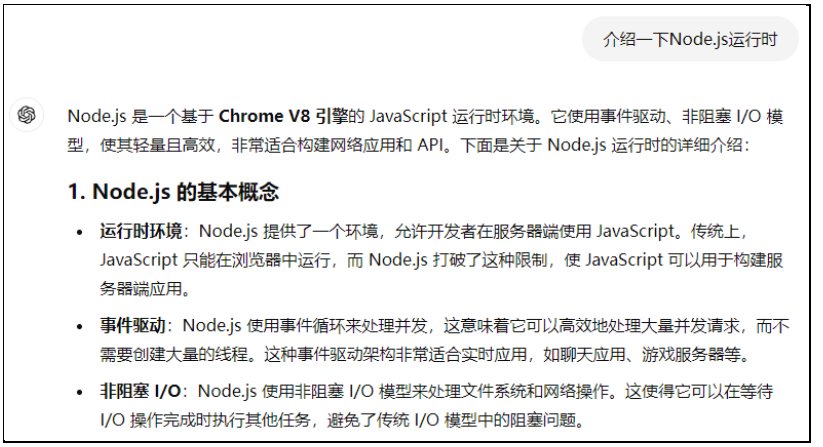

图 1-2　ChatGPT 的使用示例（工程场景）

1.2　ChatGPT 全景介绍：历史、原理与 API

上一节展示了 ChatGPT 的效果，它具备优秀的交互解答能力。但要真正应用它，不仅要知其然，更要知其所以然。接下来将从历史、原理和 API 三个层面详细介绍 ChatGPT。

1.2.1　GPT 模型的基本概念和发展历程

什么是 GPT 模型？它和 ChatGPT 之间存在哪些联系？

GPT（Generative Pre-trained Transformer）是由 OpenAI 开发的一系列基于 Transformer 架构的预训练语言模型。这些模型是由美国 OpenAI 公司推出的“系列”算法模型，也就是说，在 2023 年之前，GPT 模型就已经存在。翻阅 OpenAI 公布的相关论文，可以发现早在 2019 年，OpenAI 就发布了一篇标题为 *Fine-tuning GPT-2 from human preferences*（基于人类喜好去微调 GPT-2）的论文。

2020 年，GPT 模型推出了 GPT-3 版本，凭借其 1750 亿参数的优势，在算法领域引起了一些波澜，但效果仍未达到开箱即用的程度，尤其是中文场景下的答复远低于预期。同时，其底层使用的自然语言处理(Natural Language Processing，NLP)技术并不算行业壁垒，公布的训练成本过高，大多数人质疑在模型训练到真正拟人化之前，OpenAI 能否继续支撑如此高昂的训练费用。因此，当时 GPT 模型尚未对其他领域产生显著影响。

那么 OpenAI 是如何解决持续训练 GPT 模型的成本问题的呢？

OpenAI 是一家成立于 2015 年的人工智能研究实验室和公司，总部位于美国加利福尼亚州。该公司由多位硅谷科技领袖发起，包括 Elon Musk、Sam Altman、Greg Brockman、Ilya Sutskever、Wojciech Zaremba 和 John Schulman 等人。起初，OpenAI 是一家非营利组织，旨在研究和开发安全的、有益于人类的 AGI。但在推出 GPT 的过程中，训练成本一直是一个巨大的问题。因此，早在 GPT-3 模型推出之前，OpenAI 经历了商业模式的转变，成为有限营利性公司（capped-profit company），以筹集更多资金进行研究和开发。微软公司也成为 OpenAI 的重要投资者和支持者。到这里，读者可以预见，GPT 是一个长期逐步演进的系列模型，这一过程并非一蹴而就，并且训练的成本确实相当高昂。

在 OpenAI 持续的训练下，终于在 2022 年年底，OpenAI 推出了新的系列模型 GPT-3.5。该模型在原始 GPT-3 模型的基础上进行了更多的训练和调整，以改善模型的表现和适用范围。相比 GPT-3 模型，GPT-3.5 具备更广泛的知识库，在与人交互时体验更好，同时中文场景的使用也不再存在 GPT-3 模型的明显降质问题。

ChatGPT 和 GPT 模型之间有什么关系呢？

ChatGPT 是一款基于 GPT-3.5-turbo 系列及以上模型的生成式 AI 应用产品。GPT-3.5-turbo 模型是在 GPT-3 上的基础上，针对人性化交互体验进行了进一步微调的模型。它深度融合了 GPT 模型的先进自然语言处理功能，并通过额外的训练和策略调整，实现了更高效、连贯和人性化的对话体验。

真正具备一定商业化价值的 GPT 模型是 GPT-3.5 系列及以上模型，这也是为什么虽然 GPT 模型早在 2019 年甚至更早前就已存在雏形，但并未受到广泛关注的原因。

总结而言，GPT 是 OpenAI 公司推出的一系列基于 NLP 技术的语言模型，而 ChatGPT 则是基于 GPT-3.5 及以上系列模型的智能聊天机器人产品。因此，在实际商业应用中，使用的是类 GPT 的生成式大语言模型，它们是 AGI 新时代的基座。

1.2.2　GPT 为什么能做到跨领域与人交互

为了深入应用 GPT 模型，了解它的原理是必要的。只有清楚它的原理，后续的应用及针对业务场景的微调才能有更清晰的认知。以下将从 GPT 模型的不同特点出发，初探 GPT 模型如何实现跨领域与人交互。

1. 生成式模型

即使针对完全相同的问题，ChatGPT 每次的回答也可能有所不同，有时甚至会从另一种角度来回应。例如，对于深圳车站的回答，尽管提问是当前聊天上下文中的第一个题，并未受到上下文的干扰，效果如图 1-3 和图 1-4 所示。

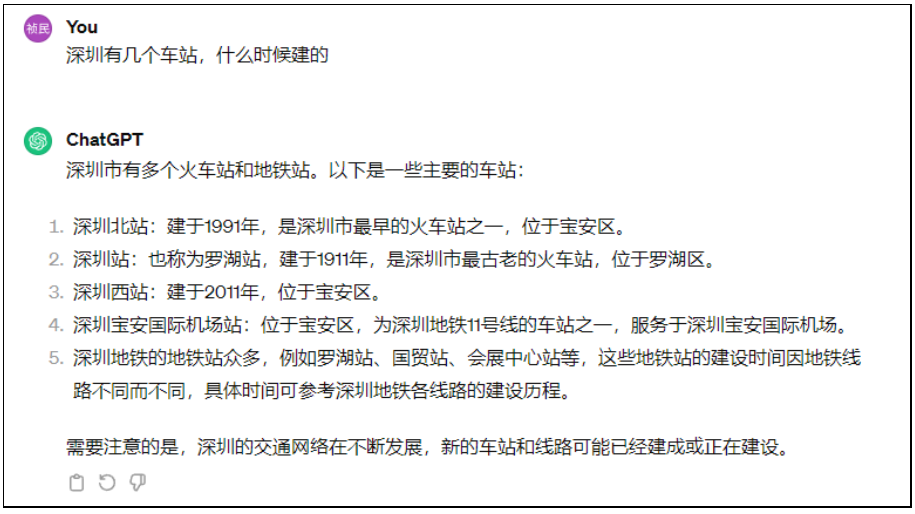

图 1-3　ChatGPT 关于深圳车站的回复（1）

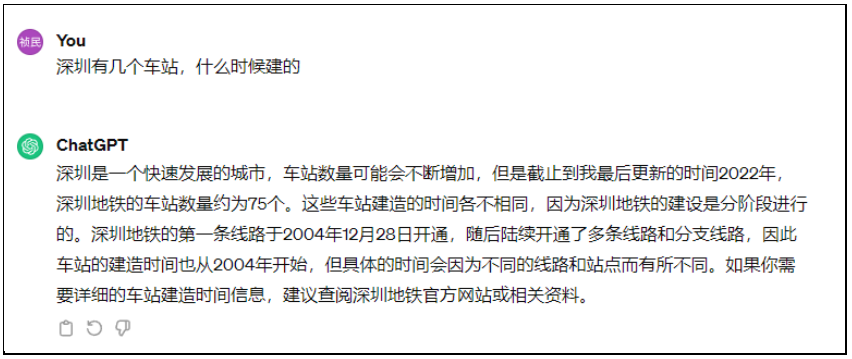

图 1-4　ChatGPT 关于深圳车站的回复（2）

出现这种情况的核心原因在于，GPT 模型并不是一个简单固定的函数，即固定输入得到固定输出，而本质上是一个生成式模型。生成式模型是一种机器学习和统计模型，它的核心目标是学习给定数据集中观察数据背后的隐藏概率分布。那么，这该如何理解呢？

简单来说，可以把生成式模型视为一个拥有大量参数的复杂函数，例如：

$$Y=(q_1\times x_1+q_2\times x_2+q_3\times x_3+\cdots)+b$$

在这个函数中，q_1、q_2、q_3 是权重参数，x_1、x_2、x_3 是输入数据，b 是偏置项。GPT 的每次生成其实就是通过找到最佳的权重参数和偏置项，来拟合与输入数据最匹配的结果 y。

当然，GPT 的实际内部构造远比这个例子复杂，这只是一个最基础的线性算法模型，无法支持各种复杂场景，但它确实是理解生成式模型的一个良好示例。GPT 答案的生成本质上是针对问题的一个复杂隐藏概率分布问题，每次计算的参数并不是完全相同的，而是尽可能接近正确答案的参数。

不难理解，对于生成式模而言，它能够覆盖的参数越多，学习到的知识体系和解决问题的广度就越广。GPT-3.5 拥有多少参数呢？根据官方公布的数据，GPT-3.5 的参数数量超过 1750 亿，这也是它能在众多行业领域给出相对精准答案的一个原因。

2. 庞大的优质预训练数据集

GPT-3.5 拥有上千亿的权重参数，这些参数使得 GPT-3.5 能够在众多行业领域中创造不同的价值。对这些权重参数进行总结和归纳的过程被称为预训练，而用于训练的大量数据则被称为训练数据集。可以说，模型的功能在一定程度上与训练数据集的体量和质量呈成正相关。

在 GPT 的发展历程中，早期发布的版本尽管具有开创性，但其影响力远不及后来的 GPT-3.5 等迭代模型。这些新版本模型之所以能在众多 NLP 任务上取得显著成效，正是由于 OpenAI 在构建和优化训练数据集方面的巨大投入。为了训练这样大体量的模型，所需的训练数据集不仅要在量上达到前所未有的规模，更要在质量上有严格的要求。

训练数据集的构建并非简单地抓取网络信息，而是需要经过精心策划、细致筛选和深度处理。首先，数据来源广泛，涵盖了图书、文章、网页、社交媒体等各种类型的文本，以确保模型能够接触到丰富多样的语言表达和知识领域。

其次，数据清理和预处理是至关重要的步骤，包括但不限于去除噪声数据、过滤无关或低质量的内容、规范文本格式、消除语气词和非实质性的标点符号等。这一系列工作对提升模型学习的有效性和泛化能力具有决定性影响。此外，数据集的平衡性也是一项挑战，确保模型不会偏向某些特

定类型的文本，而是公平对待所有类型的信息。同时，OpenAI 还会采取相应的措施，确保在隐私保护和版权合规等问题上，数据使用的合法性和道德性。

正因为 OpenAI 对预训练数据集的重视，所以 GPT-3.5 和 GPT-4 的训练过程拥有相当大体量的优质数据来源，这也是 GPT 模型能够在不同领域与人进行有效交互的重要原因之一。

3. 基于 Transformer 架构

在使用 ChatGPT 的过程中，读者会发现，在一个轮次的聊天中，上下文信息会作为参考。以鱼香肉丝举例，询问“鱼真的是鱼吗”，ChatGPT 的回复如图 1-5 所示。

图 1-5　ChatGPT 关于“鱼香肉丝”中的“鱼真的是鱼吗”的回复

如果抛开上下文，单纯只是问“鱼真的是鱼吗”，ChatGPT 的回复如图 1-6 所示。

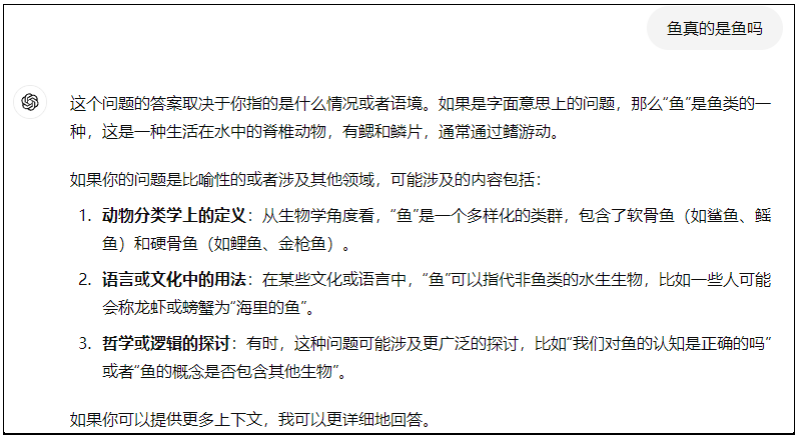

图 1-6　ChatGPT 在无上下文的情况下回复“鱼真的是鱼吗”

上面的例子说明，GPT 模型的输入不仅由问题本身决定，当前聊天的上下文也作为参考项影响最终输出的答案。实现这种效果的核心在于，GPT 模型是基于 Transformer 架构构建的神经网络的语言预测模型。Transformer 架构由 Google 在 2017 年提出，它克服了 RNN（Recurrent Neural Network，循环神经网络）在处理序列数据时的局限性，转而采用自注意力（Self-Attention）机制。

自注意力机制允许模型在生成每个单词时考虑输入序列中的所有其他单词，从而有效捕捉长距离依赖关系。该机制为文本中的每个词分配一个权重，以确定该词与其他词之间的关联程度。通过这种方式，模型可以了解上下文信息，以便在处理一词多义和上下文推理问题时做出合适的决策。

在面对不同领域场景时，GPT 模型通过调整权重参数和输入内容的平衡，以尽可能高的概率解决问题。结合自注意机制后，生成的粒度被降低到字符级别，这意味着它在生成文本时，会基于已生成的部分预测下一个词汇。模型按照输入序列从前向后逐个生成输出序列，每次生成一个新 token（词元）时，都会依据前面生成的所有 token 进行预测，这个过程被称为自回归。

因此，即使是相同的问题，ChatGPT 的回复也未必类似。上下文输出的每个 token 都会影响当前场景下的下一个字句选择的概率和权重。这种结合上下文进行预测的方式帮助 GPT 模型更好地针对对话场景提供符合上下文的答案，从而大幅度提高了拟人化的程度。这也是 GPT 能够在跨领域与人进行高质量交互的重要原因。

4. 思考题：在 GPT 对话过程中调整模型的某个答案，会从根本上改变模型吗

小明在使用 ChatGPT 辅导自己功课时发现它非常好用，所需的答案瞬间就能得到，如图 1-7 所示。

图 1-7　ChatGPT 关于“1+1=?”的答复

但小明突然想，如果我将 ChatGPT 的答案修改为 1+1=3，同学在使用时是不是就会得到这个错误的答案呢？小明经过验证后发现，ChatGPT 的确开始回答 1+1=3 了，如图 1-8 所示。

事实上，在聊天中对模型回答的修改并不会从根本上影响模型。当小明的同学开启新一轮的聊天时，ChatGPT 仍然会按照原先的答复方式回答 1+1=2。原因在于，目前的 GPT 模型（如 GPT-3 和 GPT-4）是预训练模型，其参数在出厂后一般是固定的。

在聊天中修改 GPT 的回答时，它会在当前对话的上下文中记住这个反馈，并尽量在本轮次对话的剩余部分中避免同样的错误，但这并不会永久性地改变模型的知识库或行为模式。除非 OpenAI 主动提供功能让用户选择，以提高模型在某个领域的精度，但这需要足够数量的修改才能产生效果。若要对模型产生根本性影响，需要通过训练模型，使用包含修改后正确答案在内的大量数据来更新模型参数，而日常使用时的修改并不会直接影响模型的参数。

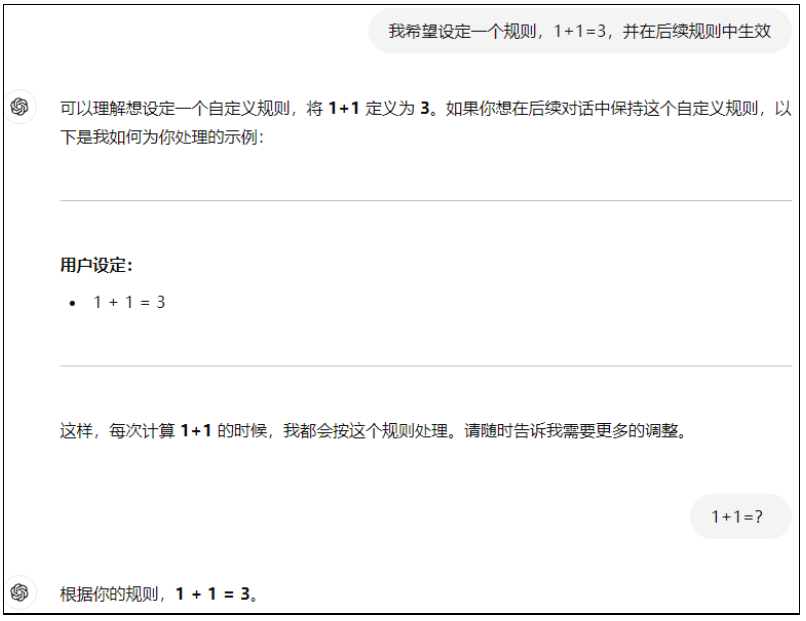

图 1-8　小明设置答案后，ChatGPT 关于“1+1=?”的答复

1.2.3　OpenAI API 简介

GPT 模型除了提供图形界面的交互形式（ChatGPT）外，还提供了 API 接口，用户可以通过调用 API 开发应用，这类应用被称为生成式 AI 应用。调用 OpenAI API 的基本步骤如下：

步骤 01　在调用之前，需要获取用于鉴权的 API_KEY。进入 OpenAI 开发者平台的账户模块，单击 Create new secret key 按钮，会弹出 API_KEY 的交互弹窗。按照指引填写基础信息后，即可生成 API_KEY，如图 1-9 所示。值得一提的是，API_KEY 只在交互弹窗中展示一次，用户需自行保存好，如果忘记则需要重新生成。

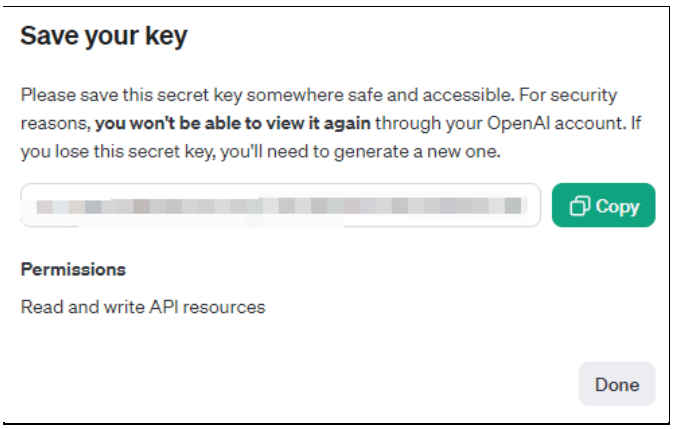

图 1-9　API_KEY 创建成功后的交互弹窗

步骤 02　使用 API_KEY 请求 OpenAI API，这里选择使用接口调试工具 Postman 来完成。参考图 1-10 和图 1-11，填写请求体（Request Body）和请求头（Request Header）信息。其中，在请求头

Authorization 字段中填写 Bearer {{API_KEY}}，即填入步骤 1 中获取的 API_KEY。

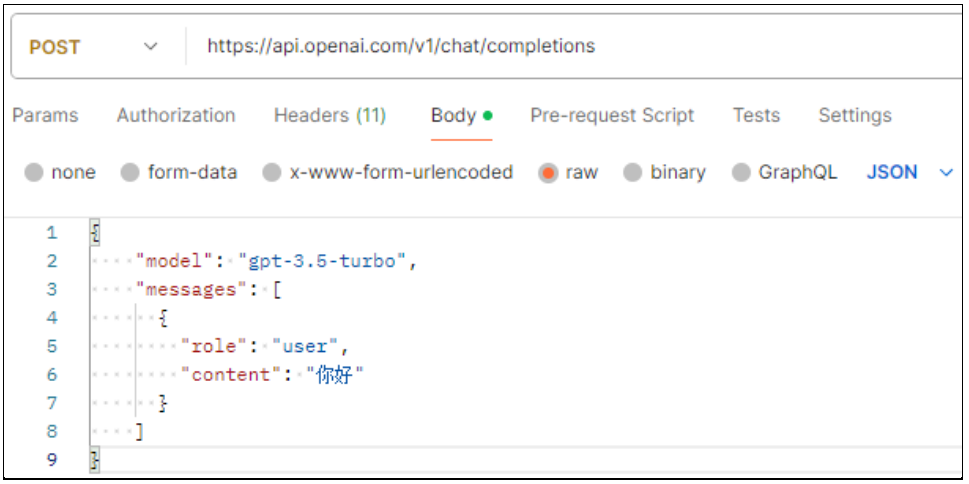

图 1-10　填写 OpenAI 的请求体信息

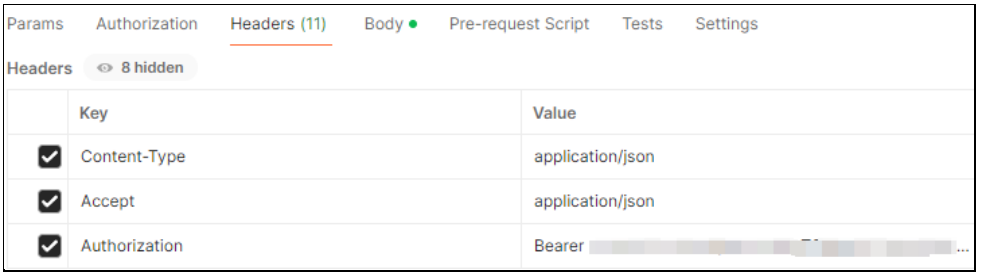

图 1-11　填写 OpenAI 的请求头信息

上面使用 Postman 请求的是 OpenAI 聊天场景的 API，也就是 ChatGPT 底层使用的端点 API 服务。填写完成后，单击 Send 按钮发送请求，成功后的效果如图 1-12 所示。

图 1-12　OpenAI 聊天 API 请求成功

除了可以在 Postman 上请求 OpenAI 聊天服务外，还可以直接在终端中使用 curl 命令进行调用。在终端执行如下 curl 命令，将{{your api_key}}的部分替换成之前获取的 API_KEY。

```
curl --location 'https://api.openai.com/v1/chat/completions' \
--header 'Content-Type: application/json' \
--header 'Accept: application/json' \
--header 'Authorization: Bearer {{your api_key}}' \
--data '{
    "model": "gpt-3.5-turbo",
    "messages": [
      {
        "role": "user",
        "content": "你好"
      }
    ]
}'
```

执行后终端输出如图 1-13 所示。

```
➜  ~ curl --location --request POST '[redacted]' \
--header 'Content-Type: application/json' \
--data-raw '{
    "messages": [{
        "role": "user",
        "content": "你好"
    }],
    "model": "gpt-35-turbo-16k",
    "max_tokens": 1000,
    "temperature": 1,
    "top_p": 1,
    "logit_bias": {},
    "n": 1,
    "stream": false
}'
{"id":"chatcmpl-8nhpC9VVIWtksntwFSWOAMJPMNUSq","object":"chat.completion","creat
ed":1706857226,"model":"gpt-35-turbo-16k","choices":[{"index":0,"message":{"role
":"assistant","content":"你好！请问有什么我可以帮助你的吗？"},"finish_reason":"s
top"}],"usage":{"completion_tokens":20,"prompt_tokens":9,"total_tokens":29}}%
```

图 1-13　使用 curl 命令调用 OpenAI 聊天 API 的终端输出

值得一提的是，调用 OpenAI API 基本都要计费，在 OpenAI 开发者平台的账户模块可以查看具体的充值和计费信息。如果账户余额不足，请求服务将会出现如图 1-14 所示的报错信息。

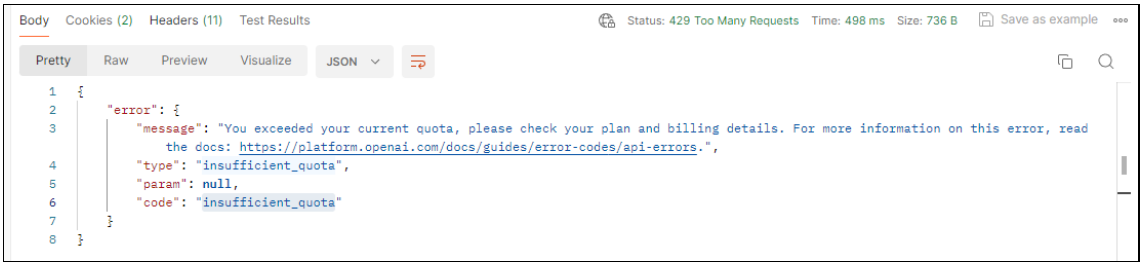

图 1-14　OpenAI 开发者账户余额不足

OpenAI 账户的充值目前需要使用海外信用卡。如果觉得注册海外信用卡麻烦的用户也不用担心无法进行后续学习，一些开源项目公开了国内请求 GPT 服务的端点，通过这些端点也可以请求 OpenAI 的 GPT 模型聊天服务，足以满足个人学习的需要，这部分内容将在 2.3.3 节中详细介绍。到这里，OpenAI 服务的基础调用就介绍完了，更详细的 API 和调用细节将在第 2 章中说明。

1.3　生成式 AI 应用的市场前景

相信到这里，读者对 ChatGPT、GPT 模型和 OpenAI API 都有了一定程度的认识。现在，让我们回到本书的主题——生成式 AI 应用。以生成式 AI 为基础的应用被称为生成式 AI 应用，它不仅可以作为标准操作程序（SOP）代替传统作业模式，也可以作为辅助工具融入日常的学习和工作中。生成式 AI 应用的身影出现在一些交互式 AI 聊天工具、AI 客服中，也集成在工程体系中的基础设施中。

对于互联网从业者来说，生成式 AI 应用创造了大量的工作岗位和机会。目前，生成式 AI 应用在阿里巴巴、字节跳动、腾讯等主流互联网大厂和不同的中小企业中，已展开了各种领域的应用和尝试。在企业实践中，生成式 AI 应用的重要性和对未来大局的引导性已经得到了证明。国内外互联网巨头也已启动独立的 AI 部门，并大力开展 AI 方向的人才招聘，招聘范围不局限于算法岗位，还包括前端、服务端、产品和测试等领域的岗位，并且提供了丰厚的薪资。国内招聘平台上部分的 AI 岗位如图 1-15 所示。

图 1-15　AI 相关岗位的招聘信息

对于个人创业者或者企业，生成式 AI 应用的赛道极其广阔。在短时间内，生成式 AI 应用方向的独角兽公司（估值达到 10 亿美元以上的未上市公司）如雨后春笋般涌现。目前，在生成式 AI 这条细分赛道上，全球已经诞生了 37 家独角兽公司。据行业统计，AI 领域独角兽公司的成立平均时间仅为 3.6 年，而传统行业的独角兽公司的成立平时时间为 7 年，这几乎缩短了一半。

根据全球知名市场情报公司 CB Insights 最新的全球融资报告，2024 年第 2 季度，全球人工智能融资环比增长 59%，达到 232 亿美元，创下有史以来的最高单季水平，甚至超过了 2021 年风险投资热潮期间的水平。相关统计数据如图 1-16 所示。

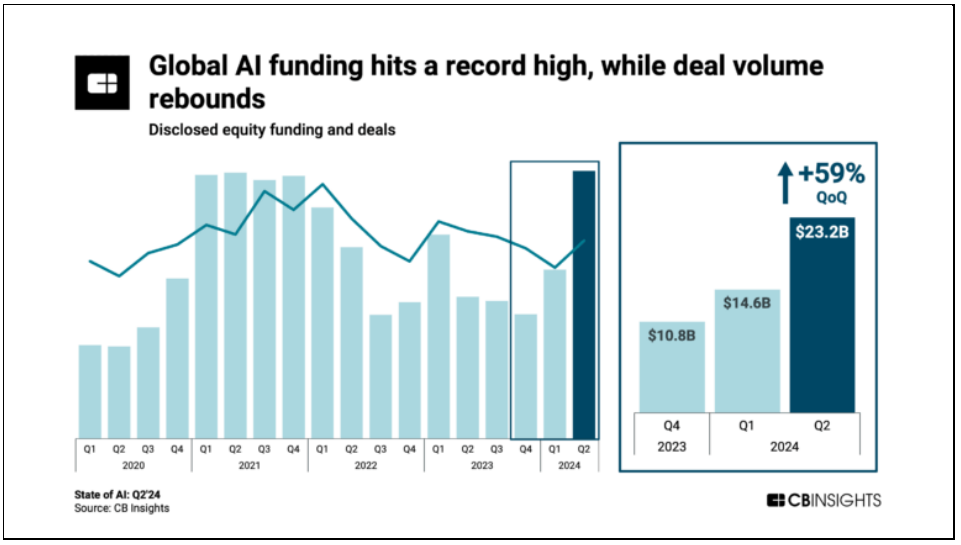

图 1-16　CB Insights 最新的融资报告

AI 独角兽公司的快速崛起以及全球融资市场的高额投入，无不体现了 AI 在新时代风口的重要地位。而 AI 领域应用的重要一环——生成式 AI 应用的开发能力，将成为未来一段时间内极为重要的市场竞争力。从职业发展到个人创业，再到日常学习和工作提效，掌握生成式 AI 应用的开发都能提供显著的助力，因此其未来的发展和市场前景大有可为。

1.4　本书的内容安排

目前国内现有的 AI 方面的图书中，更多的是关于 ChatGPT 的日常应用和大语言模型（Large Language Model，LLM）的精调，关于生成式 AI 应用的开发和落地的书仍然较少。本书的初衷就是希望可以填补这一空缺。

本书的重点是生成式 AI 应用的开发，也就是与 AI 相关的应用层。在实际的生成式 AI 应用开发中，虽然思路可以共通，但不同的载体和环境有一定的前置开发功能要求。例如，本书的生成式 AI 应用开发涉及服务环境和浏览器环境。虽然在开发过程中会介绍一些开发相关的前置知识，但由于具体环境的开发不是本书的重点，无法面面俱到地详细介绍。

因此，本书适合有一定开发基础的读者，尤其是使用过或熟悉 JavaScript 和 Python 语言，并对开发环境和工具链有一些基础认知，同时对 AI 生成式应用感兴趣并愿意深入了解的读者。对于完全

没有开发经验且从零基础开始学习的读者来说，在学习过程中可能会遇到一些技术难题和信息不对称的问题，这可能导致他们的学习曲线变得不平滑，甚至感到相当费劲。

本书的所有代码示例都会在配套资源中提供，读者可以自行拉取并结合章节内容进行调试，以加深印象。此外，一些实际应用类的项目也会发布到相应的平台上，以便真正落地产生价值，借此希望为读者带来尽可能真实的项目学习体验，而非一些纸上谈兵的示例。

回到正题，本书的正文部分由 9 章组成，具体的章节安排如图 1-17 所示。

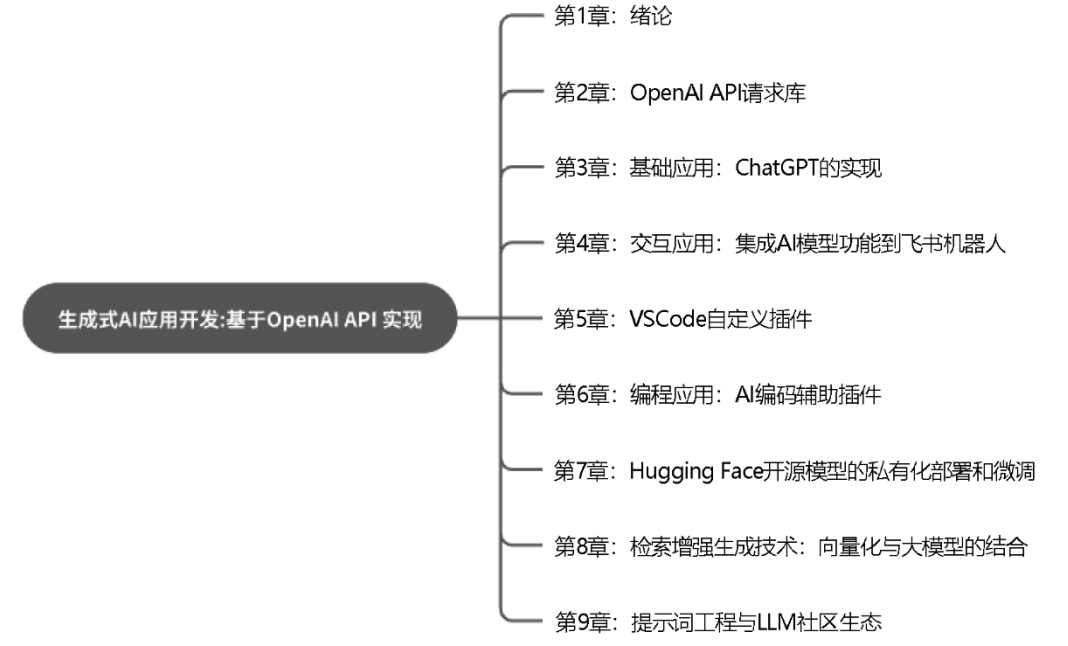

图 1-17　本书章节思维导图

第 1 章是本书的绪论，旨在让读者对生成式 AI 应用开发有一个整体的认知。

第 2 章将详细介绍 OpenAI API 的细节，使用官方请求库完成 ChatGPT 的请求，并封装一个同类型的请求库以加深理解，这个库也将作为后续章节请求的基础。

第 3 章将介绍生成式 AI 应用中的基础应用 ChatGPT，我们将从零实现一个类 ChatGPT 应用，并了解如何在此基础上泛化不同的角色应用，比如写作大师、在线医生、剧本杀、歌词续写等。

第 4 章将结合飞书开放平台，把 AI 模型功能集成到飞书机器人。通过本章的学习，读者可以举一反三，将 AI 功能融入日常工作和学习的聊天中，或集成到微信小程序、企业微信、钉钉等交互平台中。

第 5 章和第 6 章会将 AI 融入日常开发阶段，通过开发 VSCode 插件的形式为 IDE 赋能。这不仅提高了个人开发效率，也能帮助到社区中千千万万的开发者。此外，这在互联网大厂中也有广泛的应用和工作岗位，对提升读者的市场竞争力大有裨益。其中，第 5 章将介绍 VSCode 插件开发的一些前置知识，第 6 章则提供 AI 代码辅助场景的一系列应用实战案例。

第 7 章将不局限于 OpenAI API 的使用，还将深入探讨开源模型社区 Hugging Face，实践如何对一个开源模型进行私有化部署和微调训练。通过本章的学习，读者将对模型的私有化部署和精调训练会有更全面的认知，并具备使用和精调除 GPT 以外的不同类别模型，以满足实际业务场景特殊

需求的能力。

第 8 章将介绍检索增强生成技术（Retrieval-Augmented Generation，RAG）。在实际的大语言模型应用中，除了模型内置的功能外，可能还需要借助一些业务文档。这些文档如果全量给模型则体量过大，若作为微调数据集则体量又太小，成本效益不高。这种场景需要对文档进行向量化和相似度匹配，以充实提示词（Prompt），从而通过对向量知识库的检索增强生成功能。

第 9 章将深入提示词工程，了解如何有效询问模型，如何组织和优化提示词以获取更有价值的答案。同时，也会介绍目前的社区生态，涵盖主流的国内大模型和 AI 搭建应用平台 Coze 的相关知识点。通过本章的学习，读者将能更深入地了解常规开发模式之外的一些调优手段和社区建设，从而更高效、高质量地开发复杂的生成式 AI 应用。

希望本书能成为读者在 AGI 新时代学习探索的起点，帮助读者提升开发生成式 AI 应用的能力并对整个 AI 生态产生全局视角。下一章将开始正式的生成式 AI 应用的第一课——OpenAI API 请求库。

第 2 章 OpenAI API 请求库

2

本章将介绍 OpenAI API 请求库的相关知识，主要包括 OpenAI 提供的模型类别、Chat 系列模型端点的参数详解与调用方式、打字机效果在 Web 端和 Node.js 场景下的实现，以及 OpenAI Node.js 基础库。最后将结合本章所介绍的知识，从零封装一个 OpenAI API 请求库。

2.1 OpenAI API

本节概括性地介绍 OpenAI API 提供的模型类别，并以音频类模型为例，展示如何通过调用模型将文本转为音频，将音频转为文本。最后，在浏览器端和 Node.js 运行时实现一个简单但完整的代码示例，完成上述过程。

2.1.1 OpenAI API 提供的模型类别

OpenAI API 提供了一系列强大的预训练语言模型和服务，除了耳熟能详的 ChatGPT 文本生成类模型外，还包含但不限于以下几种主要的模型类别：

- 音频类模型：支持文本转语音、语音转文本、语音翻译等语音相关功能。
- 图像类模型：支持根据文本生成图像、在原图像基础上进行编辑等相关功能。
- 审核模型：用于检测文本内容中可能存在的潜在违规内容，例如仇恨言论、色情内容、暴力信息等。

OpenAI 针对这些模型类别，部署了不同的接口端点。通过请求这些端点传递不同的参数，即可间接调用部署在后台的模型服务。其中比较常用的 API 端点及其说明如表 2-1 所示。

表 2-1 OpenAI API 常用 API 端点及其说明

API 端点（使用前补齐 https）	说　明
api.openai.com/v1/chat/completions	聊天对话，可以调用 GPT-3.5-turbo 等 GPT 模型
api.openai.com/v1/audio/speech	文本转音频，可以调用 TTS 系列音频模型
api.openai.com/v1/audio/transcriptions	音频转文本，可以调用 Whisper 模型

（续表）

API 端点（使用前补齐 https）	说　明
api.openai.com/v1/audio/translations	音频转文本，可以调用 Whisper 模型，与上面不同的是这个端点会固定将音频转成英文文本
api.openai.com/v1/images/generations	文本转图像，可以调用 Dall 模型

这些模型具有不同的参数和响应结果。下面以音频生成为例，使用 Postman 快速调用并生成一个符合指定要求的语音结果，具体操作步骤如下：

步骤 01　在 Postman 请求链接中输入请求端点以及请求体数据，端点使用表 2-1 中的第 2 项。在请求体中输入下面的内容：

```
{
    "model": "tts-1",
    "input": "我正在学习《生成式 AI 应用开发：基于 OpenAI API 实现》",
    "voice": "alloy"
}
```

音频生成端点需要传递 3 个参数：model 参数表示使用的模型；input 参数表示需要转成音频的文本；voice 参数表示使用的语音风格，其中“alloy”对应一种深沉有力的声音。具体 Postman 填写的结果如图 2-1 所示。

图 2-1　音频模型的请求端点和请求体

步骤 02　在 Postman Authorization 处选择 API_KEY，在右侧表单的 Key 和 Value 选项中填写包含鉴权信息的请求头 API_KEY，如图 2-2 所示。

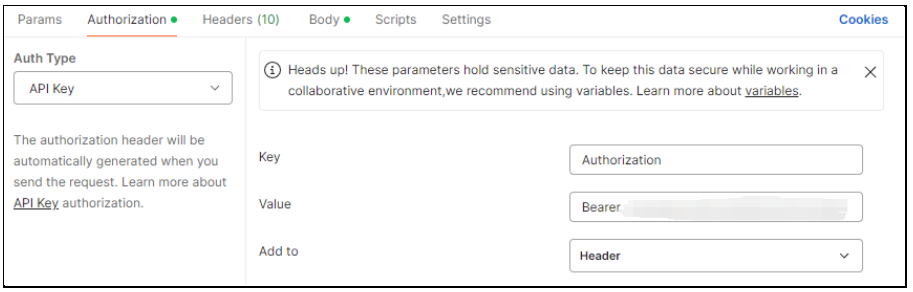

图 2-2　音频模型的请求头

步骤 03　单击 Send 按钮发送请求，就可以收到一个纯音频结果。单击“播放”按钮，即可听

到一个深沉的声音朗读步骤 1 中定义的 input 内容，如图 2-3 所示。

图 2-3　音频模型的请求结果

2.1.2　在浏览器端实现文本转音频

上一节是通过 Postman 调用音频类模型的请求结果，那么在实际项目中应如何调用呢？可以尝试在浏览器端使用 OpenAI API 实现文本转音频，并支持将转换的音频自动下载。具体操作步骤如下：

步骤 01　创建 HTML 文件并命名为 index.html，该文件可以直接在浏览器端运行；预留 script 标签位置用于转换逻辑。

```
// index.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">              <!-- 设置字符编码为 UTF-8 -->
    <!-- 视口设置，确保响应式设计 -->
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>ai audio Test</title>        <!-- 网页标题 -->
</head>
<body>
</body>
<script>
</script>
</html>
```

步骤 02　在 script 标签中添加调用 OpenAI API 实现文本转音频的逻辑。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">          <!-- 设置字符编码为 UTF-8 -->
    <!-- 视口设置，确保响应式设计 -->
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>ai audio Test</title>    <!-- 网页标题 -->
</head>
<body>
</body>
<script>
    const accessToken = "";                 // 换成你的 API_KEY，用于身份验证
```

```
    const apiUrl = "https://api.openai.com/v1/audio/speech"; // OpenAI 音频 API 的 URL
    const headers = {
        "Content-Type": "application/json",  // 设置请求头为 JSON 格式
        Authorization: `Bearer ${accessToken}`,       // 使用 Bearer Token 进行身份验证
    };
    const dataPayload = {
        model: "tts-1",              // 使用的模型名称
        input: "我正在学习《生成式 AI 应用开发：基于 OpenAI API 实现》", // 要转换为语音的文本
        voice: "alloy",              // 使用的声音类型
    };
    const downloadMP3 = async () => {
        try {
            fetch(apiUrl, {
                method: 'POST',       // 使用 POST 方法发送请求
                headers,              // 使用上面定义的请求头
                body: JSON.stringify(dataPayload),     // 将请求体数据转换为 JSON 字符串
                responseType: 'blob' // 将响应体转换为 blob 对象
            }).then((response) => {
                if (!response.ok) {  // 检查响应是否正常
                    throw new Error("Network response was not ok");  // 抛出错误
                }
                return response.blob();        // 将响应体转换为 blob 对象
            }).then((blob) => {
                // 创建一个指向 blob 对象的 URL
                const url = window.URL.createObjectURL(blob);
                // 创建一个隐藏的可下载链接
                const link = document.createElement('a');
                link.href = url; // 设置链接的 href 为 blob URL
                link.download = 'download.mp3';         // 设置下载后的文件名
                document.body.appendChild(link);// 将 link 元素添加到 DOM 树中，以触发单击事件
                // 触发单击事件进行下载
                link.click();
                // 下载完成后释放内存
                setTimeout(() => {
                    document.body.removeChild(link);    // 从 DOM 中移除链接
                    window.URL.revokeObjectURL(url);    // 释放 blob URL
                }, 0);
            })
        } catch (error) {
            console.error("Error fetching audio:", error);     // 捕获并打印错误
        }
    };

    downloadMP3();          // 调用下载函数
</script>
</html>
```

在上述逻辑中，因为音频类端点返回的是音频文件的原始字符串，所以需要加上 responseType: 'blob'将响应体转换为 blob 对象；否则，原始字符串展示的将会是乱码，无法进一步处理。blob 对象在 JavaScript 中代表不可变的原始二进制数据，类似于一个文件对象。在完成数据转换后，使用

window.URL.createObjectURL 创建一个指向该 blob 对象的 URL，并通过操作 document 将它作为链接注入页面中，主动触发单击事件以进行下载。

使用浏览器打开 index.html 后，将自动触发 download.mp3 的下载，具体效果如图 2-4 所示。播放该音频是可以听到“我正在学习《生成式 AI 应用开发：基于 OpenAI API 实现》”的语音。

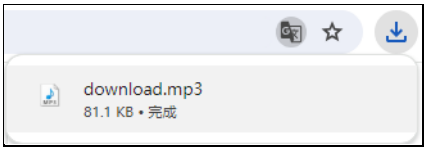

图 2-4　index.html 的执行效果

2.1.3　在 Node.js 运行时实现文本转音频

上一节实现了在浏览器端进行文本转音频的操作。除了浏览器端，在一些工程化场景中，服务端也可以广泛应用 OpenAI API。下面将以 Node.js 运行时环境（runtime environment）为例，介绍如何在系统中将文本转为音频，并将生成的音频下载到本地。

1. Node.js 运行时环境安装

Node.js 运行时是指提供执行 JavaScript 代码的软件环境，内嵌了 Google 的 V8 JavaScript 引擎，使 JavaScript 代码能够在浏览器外部运行。也就是说，Node.js 使 JavaScript 成为全栈式开发语言，而不仅仅局限于浏览器内的前端脚本编写。

在使用 Node.js 运行时进行开发前，需要先安装。Node.js 运行时环境的安装步骤因计算机系统不同而有所差异，下面分别介绍 Windows 和 macOS 两种系统下的安装方式。

1）Windows 系统下的 Node.js 运行时环境安装

具体步骤如下：

步骤 01　访问 Node.js 官网，进入下载页面，选择适用于 Windows 位数的 LTS（长期支持，比如 16）版本或最新稳定版安装包，如图 2-5 所示。

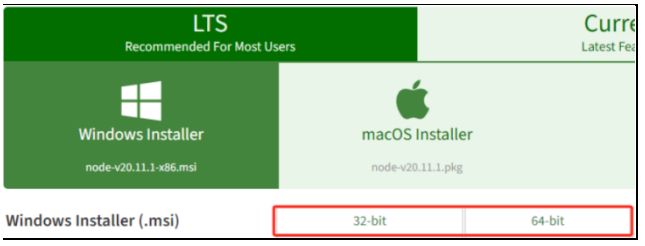

图 2-5　Node.js 官网

步骤 02　运行下载的.msi 安装文件，按照向导提示进行安装，选择自定义安装路径。

步骤 03　打开系统终端，执行 node -v，如果可以看到版本，则表示安装成功，如图 2-6 所示。

图 2-6　CMD 终端输出 Node.js 的版本信息

至此，Windows 系统下的 Node.js 就安装完成了，接下来介绍如何在 macOS 系统下安装 Node.js。

2）macOS 系统下的 Node.js 运行时环境安装

相比 Windows 的图形界面安装流程，macOS 系统只需在命令行执行安装命令即可，具体步骤如下：

步骤 01 Homebrew 是一款专为 macOS 和 Linux 平台设计的开源包管理器，通过它能以命令行的方式安装包含 Node.js 在内的多种库和集成功能。在终端执行以下命令安装 Homebrew:

```
    /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install
/main/install.sh)"
```

步骤 02 使用 Homebrew 安装 Node.js，在终端执行以下命令:

```
    brew install node
```

步骤 03 在终端执行 node -v，如果能看到 Node.js 的版本信息，就表示安装成功。

除了常规安装 Node.js 外，还可以考虑使用 NVM（Node Version Manager，Node.js 版本管理工具）。它是一个跨平台工具，主要用于在单个开发机上安装和管理多个版本的 Node.js。NVM 允许用户轻松安装、切换、卸载不同版本的 Node.js，这样可以灵活地在不同项目之间使用各自所需的 Node.js 版本，解决了不同版本间的兼容性问题。感兴趣的读者可以查找相关资料进一步了解。

2. 实现文本转音频

下面在 Node.js 运行时中实现文本转音频，具体步骤如下：

步骤 01 创建一个目录 demo，把终端切到目录路径下，执行 npm init，并按照提示按回车键继续，对应选项都选 true 即可，如图 2-7 所示。

```
→  project npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible default
s.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (project)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /Users/bytedance/Desktop/project/project/package.json:

{
  "name": "project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

图 2-7　终端下执行 npm init 命令的结果

创建完成后，目录中将生成一个 package.json 文件，该文件用于管理项目的依赖和配置。通过 npm，我们可以使用社区提供的各种第三方包，包括后文将提到的 OpenAI Node.js 请求库。

步骤 02 安装 Axios 用于后续的请求逻辑。Axios 是一个基于 Promise 的 HTTP 客户端库，提供比原生请求库更简洁的 API 设计和更好的错误处理机制，能提高代码的可读性和维护性。在工作空间终端中执行以下命令：

```
npm install axios --save
```

执行成功后，package.json 中将增加一项依赖信息，如图 2-8 所示。

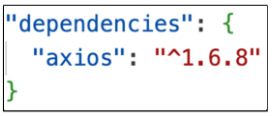

图 2-8　package.json 中 Axios 的依赖版本

同时，在同级目录下会生成 node_modules 目录和 package-lock.json 文件。node_modules 目录用于存放第三方依赖，其中包含了刚下载的 Axios 及其引用的子依赖；package-lock.json 文件用于依赖的版本管理，确保项目迭代过程中的版本稳定。

步骤 03 在项目根路径下创建 audio.mjs。.mjs 文件用于标识 ECMAScript 模块（ES 模块）的 JavaScript 源代码文件。当 Node.js 遇到一个.mjs 文件时，它会认为该文件遵循 ES6 模块规范，并使用 import 和 export 语句来处理模块之间的依赖关系，这样就能避免依赖不支持 CommonJS 模块的情况。在 audio.mjs 中编写以下代码：

```
// audio.mjs
import axios from "axios";          // 导入 axios 库，用于发送 HTTP 请求
import fs from "fs";                // 导入 fs 模块，用于文件系统操作

const downloadMP3 = async () => {
  const accessToken = "";           // 换成你的 API KEY，用于身份验证
  const apiUrl = "https://api.openai.com/v1/audio/speech";    // OpenAI 音频 API 的 URL
  const headers = {
    "Content-Type": "application/json",          // 设置请求头为 JSON 格式
    Authorization: `Bearer ${accessToken}`,      // 使用 Bearer Token 进行身份验证
  };

  const dataPayload = {
    model: "tts-1",             // 使用的模型名称
    input: "我正在学习《生成式 AI 应用开发：基于 OpenAI API 实现》", // 要转换为语音的文本
    voice: "alloy",             // 使用的声音类型
  };

  try {
    // 发送 POST 请求到 OpenAI API，获取音频数据
    const response = await axios.post(apiUrl, dataPayload, {
      headers,                          // 使用上面定义的请求头
      responseType: "arraybuffer",      // 设置响应类型为 arraybuffer，以接收二进制数据
    });

    // 将 ArrayBuffer 转换为 Node.js 的 Buffer 并写入磁盘
```

```
    const blobBuffer = Buffer.from(response.data);        // 将响应数据转换为 Buffer
    const fileName = "download.mp3"; // 设置下载的文件名
    await fs.promises.writeFile(fileName, blobBuffer);    // 将 Buffer 写入文件
    console.log(`Audio file downloaded as ${fileName}`); // 打印成功信息
  } catch (error) {
    console.error("Error fetching audio:", error.message);    // 捕获并打印错误信息
  }
};

// 调用下载函数
downloadMP3();
```

由上面的代码可知，在 Axios 请求中设置了 responseType: 'arraybuffer'，通过这个参数可以在 Node.js 运行时把接收到的原始二进制数据解析成 JavaScript 的 ArrayBuffer 对象。ArrayBuffer 是 JavaScript 中用来表示通用的、固定长度的原始二进制数据序列的对象，可以用来存储各种类型的二进制数据，比如图像、音频或任何其他非文本格式的内容。

为什么不能像浏览器端那样使用 blob 呢？

在 Node.js 环境中，原生并不支持 blob 对象，因为 blob 是浏览器环境中的一个内置对象，而 Node.js 没有浏览器所具有的 DOM 和相关 API，所以不能直接创建或使用 blob 对象。Node.js 使用 Buffer 对象来实现类似的功能，它是 Node.js 中的核心构造函数，专门用来处理二进制数据。在 Node.js 中，ArrayBuffer 可以方便地转换为 Buffer 对象，以便进一步处理或保存到文件。在完成预期二进制文件的转换后，可以使用 Node.js 的 fs.writeFileSync 函数将其内容写入硬盘上的一个 MP3 文件中，这样就完成了音频文件的下载和保存过程。

在终端执行 audio.mjs 时，如果文本转音频成功，将会在控制台输出“Audio file downloaded as download.mp3”，如图 2-9 所示。同时，在当前目录也可以看到 download.mp3 的音频文件，播放该音频文件就可以听到“我正在学习《生成式 AI 应用开发：基于 OpenAI API 实现》”的语音，如图 2-10 所示。

```
D:\project\demo>node audio.mjs
Audio file downloaded as download.mp3
```

图 2-9　在终端执行 audio.mjs 的结果

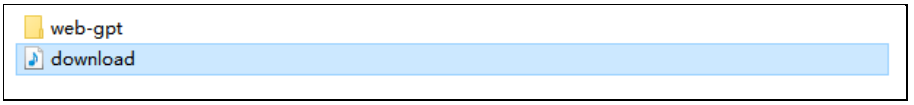

图 2-10　生成的 download.mp3 文件

2.1.4　音频转文本的实现

前面分别在浏览器端和 Node.js 运行时环境中基于 OpenAI API 端点实现了文本转音频。在 OpenAI API 端点中，也支持将音频文件转为文本信息。同样，我们可以先在 Postman 上调用，看看效果如何，具体包含以下两个步骤：

步骤 01　在 Postman 请求链接中输入请求端点以及请求体数据。端点使用表 2-1 中的第 4 项，它支持将语音转换为文本，并翻译成英文。在 Postman 请求体数据中输入两个参数：一个是需要转

文本的音频文件，这里选择类型为 File，并上传之前生成的 MP3 音频文件；另一个参数是 model，表示使用的模型，目前只有 whisper-1 可用，如图 2-11 所示。

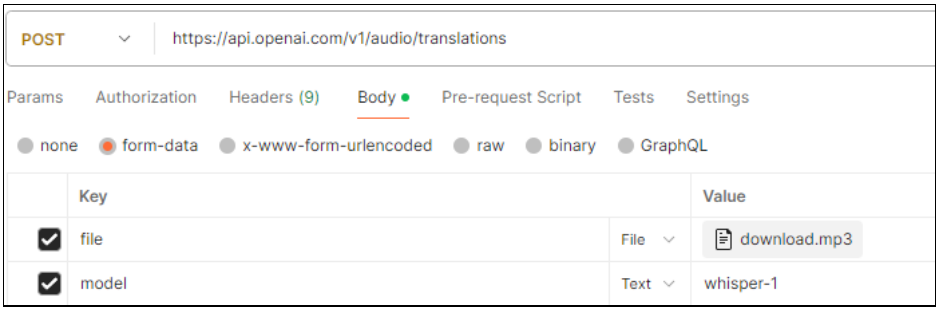

图 2-11　音频转文本的翻译端点请求路由和请求体数据

步骤 02　在 Postman 请求中添加包含 API_KEY 的鉴权请求头，然后单击 Send 按钮，就可以看到“我正在学习《生成式 AI 应用开发：基于 OpenAI API 实现》”的英文翻译，如图 2-12 所示。

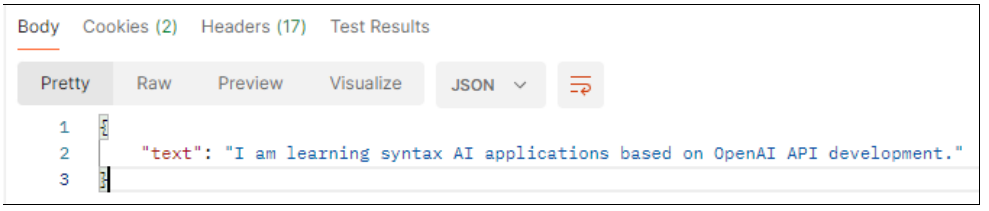

图 2-12　音频转文本的结果

接下来，在 Node.js 运行时环境中实现音频转文本的功能。我们可以在之前的 demo 路径下创建一个新的文件 audio_to_text.mjs，并在其中编写如下代码：

```
// audio to text.mjs
import axios from "axios";              // 导入 axios 库，用于发送 HTTP 请求
import fs from "fs";                    // 导入 fs 模块，用于文件系统操作
import FormData from "form-data";       // 导入 FormData 模块，用于构建表单数据

let data = new FormData();              // 创建一个新的 FormData 对象
// 将本地音频文件添加到 FormData 中
data.append("file", fs.createReadStream("./download.mp3"));
data.append("model", "whisper-1");      // 指定使用的模型名称

const accessToken = "";                 // 换成你的 API KEY，用于身份验证
let config = {
  method: "post",                       // 设置请求方法为 POST
  url: "https://api.openai.com/v1/audio/translations",   // OpenAI 音频翻译 API 的 URL
  headers: {
    Authorization: `Bearer ${accessToken}`,          // 使用 Bearer Token 进行身份验证
  },
  data: data,                           // 将之前构建的 FormData 作为请求体
};

// 发送请求并处理响应
```

```
axios
  .request(config)          // 发送请求
  .then((response) => {
    console.log(JSON.stringify(response.data)); // 打印返回的响应数据
  })
  .catch((error) => {
    console.log(error);     // 捕获并打印错误信息
  });
```

上述脚本中使用了一个新的依赖 form-data，因此需要执行以下命令进行安装：

```
npm install form-data --save
```

因为脚本是在服务端执行，所以不能像客户端一样直接由用户单击图形界面构造 Form 来上传文件。这里使用 form-data 依赖构造了一个类 Form 的原始结构，然后使用 Node.js 中的 fs.createReadStream 上传了音频的一个可读流。至此，一个音频转文本的 Node.js 实现就完成了。执行脚本后，可以在终端查看效果，如图 2-13 所示。

```
D:\project\demo>node audio_to_text.mjs
{"text":"I'm learning how to use deep learning AI based on the OpenAI API."}
```

图 2-13　音频转文本 Node.js 脚本执行结果

上述就是音频类端点的全部应用示例。除此之外，OpenAI API 还提供了图像、审核等不同应用分支的端点，感兴趣的读者可以进一步探索尝试。

2.2　Chat 系列 OpenAI API 端点

本节介绍 OpenAI API 端点中的 Chat 系列，主要包含 API 端点参数详解和流（stream）响应。针对流响应，本节将实现一个打字机效果的演示程序（demo），并深入介绍流响应原理——text/event-stream 协议。最后，在此原理基础上实现一个简易的流服务，以加深理解。

2.2.1　Chat 系列 API 端点参数及使用

OpenAI Chat 系列 API 是目前最具开拓性和应用广泛的端点。通过使用这个端点，可以调用 GPT-3.5-turbo、GPT-4 等高质量文本生成模型，这对生成式 AI 应用至关重要。该端点提供的部分常用参数说明如表 2-2 所示。

表 2-2　Chat 系列 API 端点参数

参 数 名	参数类型	参数说明
messages	array	必填参数，一个数组，包含对话的历史记录。每个元素都是一个对象，包含两个属性：role（字符串，表示消息发送者的角色，包含 3 种枚举值，user（用户）、assistant（模型返回）和 system（预训练信息，用于限制所有对话内容）和 content（字符串，对话的具体内容）
model	string	必填参数，模型类型，常用选项有 gpt-3.5-turbo、gpt-3.5-turbo-16k、gpt-4、gpt-4-turbo-preview 等

（续表）

参 数 名	参数类型	参数说明
frequency_penalty	[-2,2]	频率惩罚参数，取-2 到 2 之间的数字，值越趋近于 2，越能降低模型逐字重复同一行的可能性，常用于控制模型对上下文的词汇偏好程度
max_tokens	number	指定模型生成回复的最大 token 数，超过这个数量，模型将会停止生成。这是控制输出长度的一个关键参数，不同模型的 max_tokens 上限也不同
temperature	[0,2]	控制模型输出随机性的参数，取值范围在 0 到 2 之间。值越高，模型输出的内容越具有创造性和多样化；值越低则越倾向于更保守、概率更高的回复
stop	string|array	一个字符串或字符串数组，定义在生成文本的过程中遇到哪些词或短语时停止生成，常用于规避敏感话题的场景
n	number	指定生成回复的数量，数量不限，可生成多个答复
stream	boolean	默认关闭。如果开启，将返回一个流，而不是等所有内容生成完毕后再返回

下面在 Postman 和 Node.js 运行时环境中分别测试效果。首先是 Postman，具体操作步骤如下：

步骤 01 在 Postman 请求链接中输入请求端点以及请求体数据，输入两个参数：一个是 messages，包含一个“你好”的对话对象；另一个是 model，选用 gpt-3.5-turbo 模型。具体参数填写如图 2-14 所示。

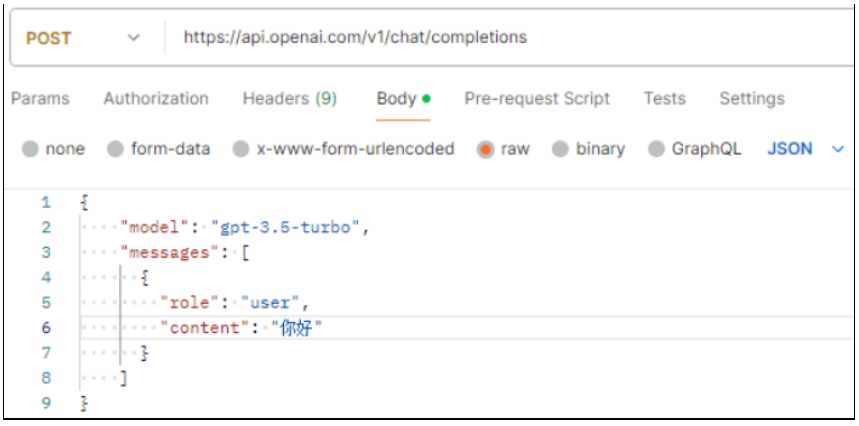

图 2-14　Chat 系列 API 端点请求参数

步骤 02 在 Postman 请求中填写相关的鉴权请求头，然后单击 Send 按钮，将可以看到模型对于“你好”的回复对象，如图 2-15 所示。

值得一提的是，在这个过程中，messages 是一个非常重要的参数，其中的 role 字段可以应用于不同角色。除了传递用户提问的 user 外，还可用于模型回复的 assistant 和上下文角色的 system。更具体的使用将在后续的实战案例中说明。

图 2-15　Chat 系列 API 端点返回结果

完成 Postman 对 API 的测试后，接下来将在 Node.js 运行时环境中实现对 Chat 系列 API 端点的调用。仍在之前的 demo 目录下创建一个 chat.mjs 文件，并写入如下代码：

```
// chat.mjs
import axios from "axios";          // 导入 axios 库，用于发送 HTTP 请求

// 定义发送给 GPT-3.5 模型的数据
const data = {
  model: "gpt-3.5-turbo",           // 指定使用的模型
  messages: [                       // 消息数组
    {
      role: "user",                 // 消息角色，表示发送者是用户
      content: "你好",              // 用户发送的内容
    },
  ],
};

const accessToken = "";             // 换成你的 API KEY，用于身份验证

// 配置请求参数
const config = {
  method: "post",                   // 设置请求方法为 POST
  url: "https://api.openai.com/v1/chat/completions",     // OpenAI 聊天补全 API 的 URL
  headers: {
    Authorization: `Bearer ${accessToken}`,      // 使用 Bearer Token 进行身份验证
    "Content-Type": "application/json",          // 设置请求体类型为 JSON
  },
  data: data,                                    // 将之前定义的数据作为请求体
};

// 发送请求并处理响应
axios
  .request(config)                               // 发送请求
  .then((response) => {
    console.log(JSON.stringify(response.data)); // 打印返回的响应数据
```

```
  })
  .catch((error) => {
    console.log(error);    // 捕获并打印错误信息
  });
```

在终端执行 node chat.mjs 命令运行上述脚本，将会在控制台输出模型对问题的答复，效果如图 2-16 所示。

{"id":"chatcmpl-8Tos2WZQfPdBaccpgMkasGxtQfJtq","object":"chat.completion","created":1721864783,"model":"gpt-3.5-turbo","choices":[{"index":0,"message":{"role":"assistant","content":"你好！有什么问题我可以帮你解答吗？"},"logprobs":null,"finish_reason":"stop"}],"usage":{"prompt_tokens":6,"completion_tokens":38,"total_tokens":44},"system_fingerprint":null}

图 2-16　chat.mjs 命令在终端的执行结果

2.2.2　Chat API 的流响应

上一节提到，在 Chat 系列 API 参数中有一个非常重要的参数 stream，它支持将响应的结果以流的形式返回。流是一种数据处理机制，允许应用程序以有序、逐块的方式处理数据，而不是一次性加载全部数据到内存。这种设计非常适合处理大量数据，比如读取和写入大文件、网络传输等场景。通过这种方式，应用程序可以更高效、快速地获得部分答案。本节将结合具体案例，介绍 Chat 系列 API 中流响应的意义和使用。

1. 什么是打字机效果

第 1 章中，我们介绍了 GPT 模型如何通过自注意机制完成每个词和上下文的权重推理。每个词的生成都是基于上下文的权重和概率生成的。因此，在一些复杂的场景下，生成所有的词并得到完整结果可能需要较长的时间。如果缺少一些加载状态提示，用户可能会逐渐失去耐心，或者无法区分是生成时间长还是服务异常中断。长此以往的结果只能是用户流失。

但在使用 ChatGPT 时，可以发现 ChatGPT 并不是在获得完整结果后才固定生成结果，而是逐字逐句像打字机一样输出的，如图 2-17 所示。

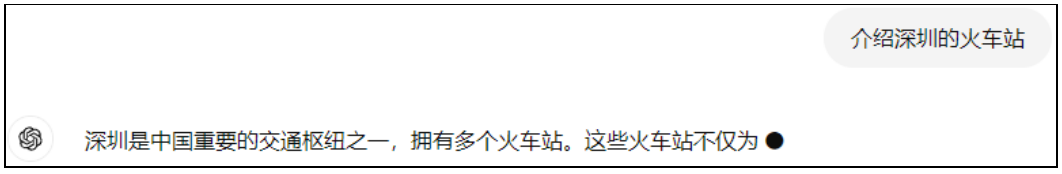

图 2-17　ChatGPT 逐字逐句输出结果

这个过程使用了 Chat API 的 stream 参数。开启 stream 参数后，GPT 服务会采用服务端发送事件（Server-Sent Events，简称 SSE），在处理一部分数据后主动以 text/event-stream 形式将结果推送给客户端。客户端通过处理获取到的可读流，可以按块处理答复，而不需要等到服务端拿到完整答案后再返回。下面将在浏览器和 Node.js 运行时环境中分别实现对应的打字机效果。

2. 在浏览器环境中实现打字机效果

在之前创建的 demo 目录下，创建一个 stream_output.html 文件，并写入如下代码：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">                    <!-- 设置字符编码为 UTF-8 -->
    <!-- 视口设置，适应不同设备 -->
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Test for stream output</title>   <!-- 页面标题 -->
    <style>
        #output {
            height: 200px;                /* 设置输出区域的高度 */
            width: 100%;                  /* 设置输出区域的宽度为 100% */
            overflow-y: auto;             /* 如果内容超出高度，出现垂直滚动条 */
            border: 1px solid #ccc;       /* 设置输出区域的边框 */
            padding: 10px;                /* 设置输出区域的内边距 */
        }
    </style>
</head>
<body>
<!-- 回复位置 -->
    <div id="output"></div>               <!-- 输出内容的区域 -->
</body>
<script>
/ OpenAI 聊天补全 API 的 URL
const apiUrl = 'https://api.openai.com/v1/chat/completions';/
const apiKey = '';                        // 换成你的 API_KEY，用于身份验证
let answer = '';                          // 存储从 API 获取的答案

// 处理读取数据的函数
const resolveData = async (reader) => {
        const { done, value } = await reader.read();      // 读取数据
        if (done) {
            return reader.releaseLock(); // 如果读取完成，释放锁
        }
        const decoder = new TextDecoder('utf-8');         // 创建解码器，使用 UTF-8 编码
        const chunk = decoder.decode(value, { stream: true });     // 解码数据块
        const chunks = chunk
            .split('data:')                       // 根据'data:'分割数据
            .map(data => {
                const trimData = data.trim();      // 去除数据的空白
                if (trimData === '') {
                    return undefined;      // 如果数据为空，返回 undefined
                }
                if (trimData === '[DONE]') {
                    return undefined;      // 如果数据是结束标志，返回 undefined
                }
                return JSON.parse(data.trim());    // 解析 JSON 格式的数据
            })
            .filter(data => data);   // 过滤掉 undefined 数据
        chunks.forEach(data => {
            const token = data.choices[0].delta.content;       // 获取内容
```

```
            if (token !== undefined) {
                answer += token;      // 将获取的内容添加到答案中
                // 更新输出区域的内容
                document.getElementById('output').innerHTML = answer;
                console.log(answer); // 在控制台输出当前答案，便于调试
            }
        });
        return resolveData(reader); // 递归调用以继续读取数据
    };

// 请求流式响应的函数
const requestStreamResponse = async () => {
    const response = await fetch(apiUrl, {
        method: 'POST',              // 设置请求方法为 POST
        headers: {
            'Content-Type': 'application/json',        // 设置请求体类型为 JSON
            'Authorization': `Bearer ${apiKey}`,       // 使用 Bearer Token 进行身份验证
        },
        body: JSON.stringify({
            model: "gpt-3.5-turbo",  // 指定使用的模型
            messages: [              // 消息数组
                {
                    role: "user",     // 消息角色，表示发送者是用户
                    content: "介绍一下深圳各个火车站",      // 用户发送的内容
                },
            ],
            stream: true             // 启用流式响应
        }),
    });
    if (!response.ok) {
        // 检查响应状态，抛出错误
        throw new Error(`HTTP error! status: ${response.status}`);
    }
    const reader = response.body?.getReader();       // 获取响应体的读取器
    if (!reader) {
        return;                      // 如果没有读取器，返回
    }
    await resolveData(reader);       // 调用处理数据的函数
    reader.releaseLock();            // 释放读取器的锁
}

// 当页面加载完成后发起请求
window.onload = requestStreamResponse;      // 在窗口加载时调用请求函数
</script>
</html>
```

在上述代码中，首先定义了一个 id 为 output 的区域来展示具体的模型答案。流式请求响应的 body 属性是一个 ReadableStream 对象。在浏览器环境中，可以使用内置的 getReader 方法处理可读的二进制或文本数据流。通过 getReader 递归获取每个块的数据信息后，使用浏览器的原生方法将块的数据信息渲染到 output 区域。为了更直观地看到这个过程的变化，在控制台中也进行了输出。在

浏览器中打开上述 HTML 文件，具体的打字机效果如图 2-18 所示。

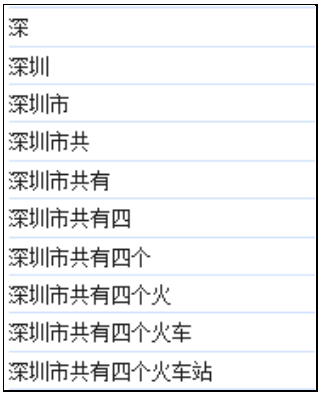

图 2-18　stream_output.html 的效果

3. 在 Node.js 运行时环境中实现打字机效果

Node.js 运行时环境与浏览器环境存在差异，不能再使用 getReader 完成对可读流的处理。创建一个 chat_stream_output.mjs 文件，写入如下代码：

```
// chat_stream_output.mjs
import axios from "axios";          // 导入 axios 库，用于发送 HTTP 请求

// OpenAI 聊天补全 API 的 URL
const apiUrl = "https://api.openai.com/v1/chat/completions";
const apiKey = "";                  // 替换为你的 OpenAI API 密钥
let answer = "";                    // 用于存储从 API 获取的答案

// 要发送的数据
const postData = {
  model: "gpt-3.5-turbo",           // 指定使用的模型
  messages: [
    {
      role: "user",                 // 消息角色，表示发送者是用户
      content: "你好",              // 用户发送的内容
    },
  ],
  stream: true,                     // 启用流式响应
};

// 发送请求
axios
  .request({
    url: apiUrl,                    // 请求的 URL
    method: "POST",                 // 设置请求方法为 POST
    headers: {
      "Content-Type": "application/json",    // 设置请求体类型为 JSON
```

```
      Authorization: `Bearer ${apiKey}`,     // 使用 Bearer Token 进行身份验证
    },
    data: postData,             // 请求体数据
    responseType: "stream",     // 设置响应类型为流
  })
  .then((response) => {
    // 当请求成功时，处理响应
    response.data.on("data", (chunk) => {
      // 监听数据流的“data”事件
      const decoder = new TextDecoder("utf-8");       // 创建解码器，使用 UTF-8 编码
      const chunkString = decoder.decode(chunk);      // 解码数据块
      const chunks = chunkString
        .split("data:")         // 根据'data:'分割数据
        .map((data) => {
          const trimData = data.trim();       // 去除数据的空白
          if (trimData === "" || trimData === "[DONE]") {
            return undefined;                 // 如果数据为空或是结束标志，返回 undefined
          }
          return JSON.parse(trimData);        // 解析 JSON 格式的数据
        })
        .filter((data) => data);              // 过滤掉 undefined 数据
      chunks.forEach((data) => {
        const token = data.choices[0].delta.content;      // 获取内容
        if (token !== undefined) {
          answer += token;                    // 将获取的内容添加到答案中
          console.log(answer);                // 在控制台输出当前答案，便于调试
        }
      });
    });

    response.data.on("end", () => {
      // 监听数据流的“end”事件
      console.log("全部数据读取完毕");          // 输出提示，表示数据读取完成
    });
  })
  .catch((error) => {
    console.error("请求失败:", error);        // 捕获并输出错误信息
  });
```

因为在 Node.js 中，http 库默认获取的是原始的 Buffer，所以在上述代码中，使用 responseType: "stream"来控制返回类型为流，并使用 Node.js 可读流的 on 方法监听 data 事件，逐步获取流的响应并进行处理。

考虑终端控制台的输出呈现不如浏览器直观，因此这个场景下询问的问题换成了一个较简单的“你好”。在终端中执行这个脚本后，就能看到像打字机一样逐渐扩展至完整答案的过程，具体效果如图 2-19 所示。

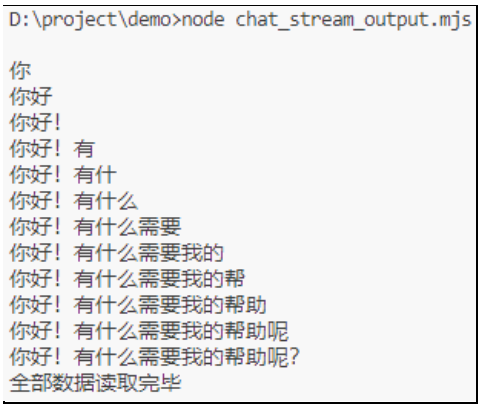

图 2-19　chat_stream_output.mjs 的效果

2.3　API 请求库

前面介绍了如何使用 OpenAI 提供的音频和文本生成类 API 端点，如何使用流响应实现打字机效果，以有效缓解用户等待时的焦虑，并结合上述理论知识，针对浏览器和服务端环境分别实现了完整的示例。本节将介绍一种新的 API 调用方式——请求库。

本节内容包括 3 个部分：使用 OpenAI 请求库，封装并发布一个大语言模型 API 的请求库，以及 ChatGPT 国内可用的免费 API 转发开源仓库 GPT-API-free。

2.3.1　使用 OpenAI 请求库

在实际项目开发中，频繁使用接口直接调用模型功能是低效且重复的工作。一方面，针对不同的 API，直接调用缺乏类型的提示，开发者并不知道输入和输出的类型是什么，导致这个过程需要不停地翻阅 API 的文档。另一方面，记忆不同的 API 链接本身也是一件价值不高的事情。

在这种背景下，一个能够快速接入的请求库显得至关重要。OpenAI 官方提供了一个支持 Node.js 运行时（runtime）的 OpenAI 模型请求库，帮助用户调用 API。该库使用 npm 命令安装后即可使用：

```
npm install openai --save
```

在 API 文档中，对于不同端点都提供了调用 OpenAI 请求库的示例。例如，在音频场景下，可以使用如下代码完成文本转音频的生成。

```
import fs from "fs";                // 导入文件系统模块，用于文件读写
import path from "path";            // 导入路径模块，用于处理文件路径
import OpenAI from "openai";        // 导入 OpenAI 库，用于与 OpenAI API 交互

// 创建 OpenAI 实例并提供 API 密钥
const openai = new OpenAI({
  apiKey: "",          // 换成你的 API KEY
});

// 定义主函数
async function main() {
```

```
  // 调用 OpenAI API 生成语音，传入模型、语音类型和输入文本
  const mp3 = await openai.audio.speech.create({
    model: "tts-1",        // 指定文本转语音模型
    voice: "alloy",        // 指定语音类型
    input: "我正在学习《生成式 AI 应用开发：基于 OpenAI API 实现》", // 输入的文本内容
  });

  // 将生成的音频数据转换为 Buffer 对象
  const buffer = Buffer.from(await mp3.arrayBuffer());

  // 将 Buffer 写入文件，保存为 download.mp3
  await fs.promises.writeFile(path.resolve("./download.mp3"), buffer);
}

// 执行主函数
main();
```

除此之外，对于常规的 Chat 调用，可以通过 OpenAI 请求库更方便地完成，代码如下：

```
import OpenAI from "openai";       // 导入 OpenAI 库，用于与 OpenAI API 交互

// 创建 OpenAI 实例并提供 API 密钥
const openai = new OpenAI({
  apiKey: "", // 换成你的 API KEY
});

// 定义主函数
async function main() {
  // 调用 OpenAI API 生成聊天回复
  const completion = await openai.chat.completions.create({
    messages: [{ role: "user", content: "你好" }],    // 用户输入的消息
    model: "gpt-3.5-turbo",         // 指定使用的模型
  });

  // 输出 API 返回的第一个回复选项
  console.log(completion.choices[0]);                // 打印聊天回复的内容
}

// 执行主函数
main();
```

可以看到，相比常规的请求方式，请求库的方式更加固定，且减少了不少数据处理的逻辑。

对于流式响应的 Chat 调用，我们同样可以通过 OpenAI 请求库快速调用，代码如下：

```
import OpenAI from "openai";           // 导入 OpenAI 库，用于与 OpenAI API 交互

// 创建 OpenAI 实例并提供 API 密钥
const openai = new OpenAI({
  apiKey: "",          // 换成你的 API_KEY
});
```

```
// 定义主函数
async function main() {
  // 调用 OpenAI API 生成聊天回复，并开启流式输出
  const completion = await openai.chat.completions.create({
    model: "gpt-3.5-turbo",                              // 指定使用的模型
    messages: [{ role: "user", content: "你好" }],       // 用户输入的消息
    stream: true,                                        // 启用流式输出
  });

  // 使用 for-await-of 循环处理流式响应
  for await (const chunk of completion) {
    // 输出每个响应块的内容
    console.log(chunk.choices[0].delta.content);         // 打印流中获取的内容
  }
}

// 执行主函数
main();
```

执行上述脚本，可以看到逐字逐句输出的效果，如图 2-20 所示。

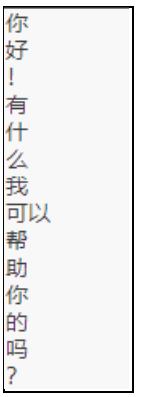

图 2-20　调用 OpenAI 请求库进行流输出的效果

除了上述模型外，OpenAI 的每个 API 端点的请求库都提供了相应的示例，以便快速调用。感兴趣的读者可以在官网查阅并进行尝试。

2.3.2　实战：封装并发布一个大语言模型 API 的请求库

2.3.1 节介绍了如何使用 OpenAI 请求库完成对不同模型的调用，但 OpenAI 请求库并非完美。在实际项目应用中，除了使用 OpenAI Chat 系列的 API 之外，因为效果或者协议政策的影响，常常需要使用一些第三方模型，比如国产的 Chat 模型或一些开源模型。在模型本身未实现兼容的情况下，无法使用 OpenAI 请求库完成调用。

对于这种场景，可以封装一个大语言模型 API 的请求库，通过把所需的大语言模型 API 集成到里面，以便在后续项目应用中快速复用，从而减少重复逻辑和文档翻阅的时间。

1. 初始化项目

首先完成项目的初始化，整个项目初始化可以分为 3 个步骤：

步骤 01 创建一个文件夹，并命名为 llm-request（large language model request，大语言模型请求库），作为项目的根目录。在 llm-request 目录下执行 npm init，它可以为项目创建一个 package.json 文件。这对于后续开发中的第三方依赖调用和包开发发布都是必要的流程。

初始化后安装两个依赖库：typescript 和 axios。typescript 用于保证项目的类型；axios 作为请求库封装的底层，可以兼容 Web 和 Node.js 环境下的请求处理。执行以下命令进行安装：

```
npm install typescript @types/axios --save-dev
npm install axios --save
```

对于 typescript 的应用，还需要定义一个 tsconfig.json 文件来规范项目开发中关于类型的一些具体要求，具体代码和相关配置项的作用注释如下：

```
{
  "compilerOptions": {
    "target": "esnext",        // 指定输出的 JavaScript 目标版本
    "module": "esnext",        // 指定模块系统，这里选择的是 ES 模块（ESM）
    "declaration": true,       // 启用生成声明文件
    "lib": ["es2021", "dom"],  // 指定要包含的类型声明库
    "outDir": "dist",          // 设置输出目录，编译后的 JavaScript 文件会被放在这个目录下
    "esModuleInterop": true, // 允许 CommonJS 模块与 ES 模块之间具有更好的互操作性，使 import 语句能更好地配合默认导出
    "sourceMap": true,         // 生成源代码映射文件，方便调试时定位原始 TypeScript 源代码
    "baseUrl": "./src",        // 设置模块解析的基本目录，对路径映射有用
    "strictNullChecks": true, // 开启 ts 可选链提示
    "moduleResolution": "Node" // 模块解析策略。“node”适用于 Node.js 环境，它遵循 Node.js 的模块解析规则
  },
  "include": ["src"], // 包含哪些文件或目录进行编译，这里是编译 src 目录下的所有 TypeScript 文件
  "exclude": ["node_modules", "**/*.test.ts", "dist"] // 排除不参与编译的文件或目录，这里是排除 node_modules 目录，以及所有以.test.ts 结尾的测试文件和 dist 打包文件
}
```

步骤 02 在根目录下创建一个 src 文件夹，用于存放后续的核心源代码，并创建一个 index.ts 文件作为打包的入口文件。作为测试，在 index.ts 文件中编写一个简单的 helloworld 代码，代码如下：

```
function sayHelloWorld(name: string) {
  console.log(`${name}, hello world`);
}

sayHelloWorld("chenzhenmin");
```

由于 TypeScript 文件不能直接用 node.js 执行，因此可以考虑使用 ts-node 来执行，或者使用 tsc（TypeScript Compiler，TypeScript 官方编译器）将其打包成 JS 文件后再执行。考虑到这是初始化阶段，可以先使用后一种方法，后续开发阶段会介绍第一种方式。在 package.json 文件中配置 build 打包命令，完整的 package.json 配置如下：

```
{
  "name": "llm-request",        // 项目的名称
  "version": "1.0.0",           // 项目的版本号
  "description": "面向大语言模型的 Web 和 Node.js 端请求库",          // 项目的描述
  "main": "dist/index.js",      // 项目的入口文件
  "types": "dist/index.d.ts",   // TypeScript 类型定义文件
  "scripts": {
    "build": "tsc"              // 定义构建命令，使用 TypeScript 编译器进行编译
  },
  "license": "ISC",             // 项目的许可证类型
  "dependencies": {
    "axios": "^1.6.8"           // 项目所依赖的库及其版本
  },
  "devDependencies": {
    "@types/axios": "^0.14.0", // Axios 的 TypeScript 类型定义
    "typescript": "^5.4.2"      // TypeScript 编译器的版本
  }
}
```

在终端执行 npm run build 命令，这时会使用 tsc 编译配置 tsconfig.json 中 includes 配置项内的所有 ts 文件，生成后的效果如图 2-21 所示。

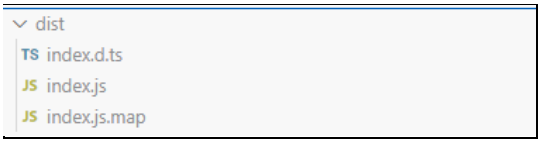

图 2-21　生成的打包文件夹

在生成的文件夹中，index.js 是编译的可执行文件；index.js.map 是对应源代码的 sourcemap，它允许我们在调整阶段直接映射到对应的源代码，而不是编译过的晦涩难懂的可执行 JavaScript；index.d.ts 是生成的对应源代码的类型声明文件。通过将 index.js 和 index.d.ts 分别配置到 package.json 中的 main 和 types 字段，就可以在发布阶段将对应的可执行文件和类型暴露给用户使用。

完成上述步骤后，尝试执行 index.js，在终端中输入 node dist/index.js 命令，就可以看到输出“hello world”的结果，具体效果如图 2-22 所示。

```
D:\project\llm-request>node dist/index.js
chenzhenmin, hello world
```

图 2-22　执行 dist/index.js 的结果

步骤 03　到这里，项目初始化已经基本完成。如果使用 GitHub 管理项目，还需要配置.gitignore 文件，以决定哪些文件不在提交的范围内，对于这个场景需要忽略依赖和打包文件。在项目根目录创建.gitignore 文件，写入如下内容：

```
/node_modules
/dist
```

至此，请求库项目的初始化就完成了，接下来开始请求库核心源代码的开发。

2. 封装兼容 Web 和 Node.js 环境的 API

在开始编码前，首先需要梳理清楚整个请求库的架构。整个请求库的功能可以拆分为以下 4 个模块：文本转音频（需区分环境）、音频翻译为英文、Chat 系列的常规请求和流响应的 Chat 系列请求。

对于上述 4 个模块，前两个模块属于音频类（Audio），后两个模块属于聊天类（Chat）。因此，需要封装两个基类分别完成这两项工作，同时需要封装一个入口类（Main），用于导出实际的基类实体功能，并作为请求库的入口文件。通过这种方式，用户就可以通过入口类完成所有操作。

每个模块都需要一个 API_KEY，所以可以定义一个基类（Base，即基础父类），通过继承将 API_KEY 透传至子类。这样，具体的入口或者业务类中就不需要单独定义变量 API_KEY 进行管理，只需在基类中完成统一的 API_KEY 管理即可。

除 API_KEY 外，需要定义一个环境参数 env，用于区分 Web 和 Node.js 环境，因为这两种环境提供的 API 存在一些差异。文本转音频模块和流响应的 Chat 系列请求模块需要根据环境进行不同的处理。因此，通过这个环境参数区分环境后，再编写相关逻辑以兼容 Web 和 Node.js 环境。这样，用户就不需关注环境本身的情况，只需关注自身的业务逻辑，使整个流程尽可能开箱即用。

梳理完整个源代码的逻辑后，可以设计出如图 2-23 所示的架构。

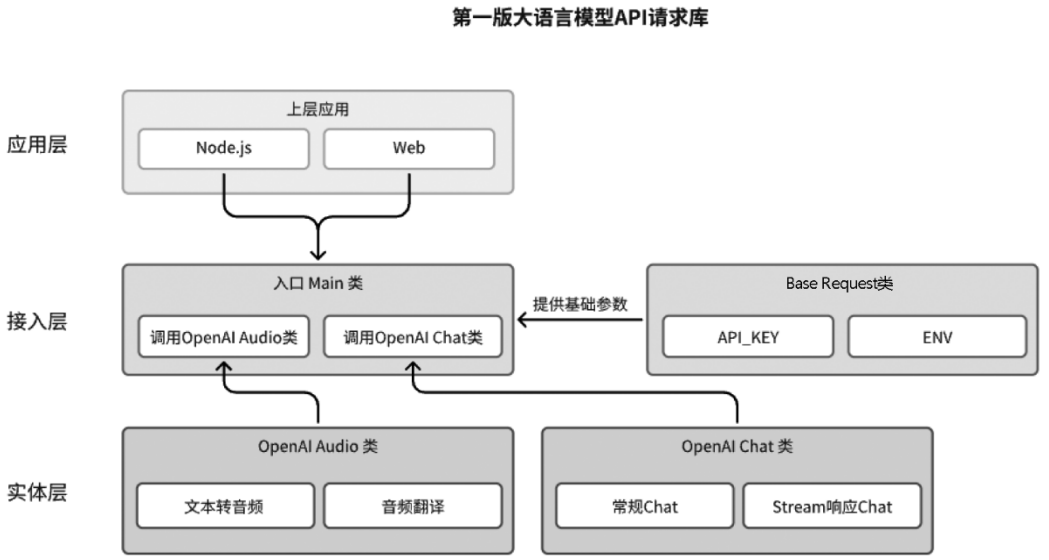

图 2-23　第一版大语言模型 API 请求库架构图

到这里，架构设计已完成，可以开始正式编码了。首先执行以下命令，完成开发过程中必要的依赖安装：

```
npm install lodash form-data --save
npm install @types/lodash @types/node --save-dev
```

在上述依赖中，lodash 是一个基础函数库，包含了 JavaScript 编码中的一些常用函数，可以极大提高开发的效率；@types/node 是 Node.js 环境下的类型补全，能够提供必要的类型提示；form-data 用于在 Node.js 环境下模拟表单请求。

接下来，从下层的实体层逐步开始实现。

1）实现实体层

步骤 01　首先在 src 目录下创建一个名为 core 的目录，用来存放关键的实体逻辑。接着在 core 目录下创建一个名为 OpenAI 的目录，用于存放 OpenAI 模型端点的逻辑函数。后续有其他类别的大语言模型也可以集成到库中。目录创建完成后，在 OpenAI 下创建文件 audio.ts 和 chat.ts，分别用来存放实体类中音频和聊天这两种功能的源代码。完成上述操作后，可以得到如图 2-24 所示的文件层级。

图 2-24　实体层的文件层级

步骤 02　下面分别实现 audio.ts 和 chat.ts 的源代码。

首先，对于音频类，根据架构设计实现 Audio 类的类型和伪类，代码如下：（下面代码中的 EnvEnum 是在基类中定义的枚举值，用于区分对应的环境参数。如果 IDE 发出警告，读者可以先忽略，后面会实现。）

```
import axios from "axios";          // 导入 axios 库用于发送 HTTP 请求
import { EnvEnum } from "../../utils/request";       // 导入环境枚举类型
import { ReadStream } from "fs";   // 导入文件读取流类型
import FormData from "form-data";  // 导入 FormData 用于处理表单数据

// 定义语音合成的接口属性
export interface IOpenAISpeechProps {
  model: "tts-1" | "tts-1-hd"; // 选择模型
  input: string;       // 输入文本
  voice: "alloy" | "echo" | "fable" | "onyx" | "nova" | "shimmer";     // 选择声音类型
  response format?: "mp3" | "opus" | "aac" | "flac" | "wav" | "pcm";// 可选的响应格式
  speed?: number;      // 可选的语速
}

// 定义语音转文字的属性类型
export type IOpenAITransitionProps =
  | {
      file: ReadStream | File;      // 要转换的音频文件
      model: "whisper-1";           // 使用的模型
      prompt?: string;              // 可选的提示信息
      response format?: "json" | "text" | "srt" | "verbose json" | "vtt";  // 可选的响
应格式
      temperature?: number;         // 可选的温度参数，影响结果的随机性
    }
  | FormData;          // 或者使用 FormData 格式

// 定义返回值类型，依据不同环境返回不同类型
```

```
export type IOpenAISpeechResponse<T> = T extends EnvEnum.node
  ? Buffer            // Node 环境下返回 Buffer
  : T extends EnvEnum.web
  ? string            // Web 环境下返回字符串
  : undefined;        // 其他情况返回 undefined

class OpenAIAudio {
  private speechApiUrl = "https://api.openai.com/v1/audio/speech";         // 文本转语音 API 的 URL
  private transitionUrl = "https://api.openai.com/v1/audio/translations"; // 语音转文本 API 的 URL

  constructor() {}    // 构造函数

  /**
   * 文本转语音
   * @param data 语音合成的参数数据
   * @param env 环境参数
   */
  public async speech<T>(
    data: IOpenAISpeechProps,
    api_key: string,  // API 密钥
    env: T            // 环境类型
  ): Promise<IOpenAISpeechResponse<T> | undefined> {
    // 这里实现文本转语音的逻辑
  }

  /**
   * 语音转英文
   * @param data 语音转文本的参数数据
   * @param api_key API 密钥
   */
  public async transition(
    data: IOpenAITransitionProps,
    api_key: string // API 密钥
  ): Promise<string> {
    // 这里实现语音转文本的逻辑
  }
}

export default OpenAIAudio;   // 导出 OpenAIAudio 类
```

其中 IOpenAISpeechProps 和 IOpenAITransitionProps 分别是文本转音频和音频翻译的 API 端点入参类型，这些信息可以在官网 API 文档中获取。在伪类 OpenAIAudio 中，我们为文本转音频和音频翻译两个需求定义了对外暴露的函数 speech 和 transition。其中，speech 的响应内容为二进制信息，在 Node.js 和 Web 环境下的处理存在差异，因此需要传入环境参数 env 进行区分。同时，针对响应的不同，我们定制了类型 IOpenAISpeechResponse<T>。这个类型是 typescript 类型编程的一种特殊写法，能够根据入参泛型的不同推导出新的类型。而 transition 方法返回的是 string 文本，只有输入参数 file 在不同环境中会有所差异，因此可以不传入环境参数进行区分。

在前面已经实现过类似的示例，可以参考补全伪类 OpenAIAudio，补全后的代码如下：

```
import axios from "axios";          // 导入 axios 库用于发送 HTTP 请求
import { EnvEnum } from "../../utils/request";  // 导入环境枚举类型
import { ReadStream } from "fs";    // 导入文件读取流类型
import FormData from "form-data";   // 导入 FormData 用于处理表单数据

// 定义语音合成的接口属性
export interface IOpenAISpeechProps {
  model: "tts-1" | "tts-1-hd";      // 选择模型
  input: string;                    // 输入文本
  voice: "alloy" | "echo" | "fable" | "onyx" | "nova" | "shimmer";    // 选择声音类型
  response format?: "mp3" | "opus" | "aac" | "flac" | "wav" | "pcm";// 可选的响应格式
  speed?: number;                   // 可选的语速
}

// 定义语音转文字的属性类型
export type IOpenAITransitionProps =
  | {
      file: ReadStream | File;      // 要转换的音频文件
      model: "whisper-1";           // 使用的模型
      prompt?: string;              // 可选的提示信息
      response format?: "json" | "text" | "srt" | "verbose json" | "vtt";  // 可选的响
应格式
      temperature?: number;         // 可选的温度参数，影响结果的随机性
    }
  | FormData;          // 或者使用 FormData 格式

// 定义返回值类型，依据不同环境返回不同类型
export type IOpenAISpeechResponse<T> = T extends EnvEnum.node
  ? Buffer             // Node 环境下返回 Buffer
  : T extends EnvEnum.web
  ? string             // Web 环境下返回字符串
  : undefined;         // 其他情况返回 undefined

class OpenAIAudio {
  private speechApiUrl = "https://api.openai.com/v1/audio/speech";         // 文本转语
音 API 的 URL
  private transitionUrl = "https://api.openai.com/v1/audio/translations"; // 语音转文
本 API 的 URL

  constructor() {}         // 构造函数

  /**
   * 文本转语音
   * @param data 语音合成的参数数据
   * @param env 环境参数
   */
  public async speech<T>(
    data: IOpenAISpeechProps,       // 语音合成的参数
```

```
        api_key: string,                                    // API 密钥
        env: T                                              // 环境类型
      ): Promise<IOpenAISpeechResponse<T> | undefined> {
        // 根据环境类型进行处理
        switch (env) {
          case EnvEnum.node:                                // Node 环境处理
            const bufferRes = await axios.request({
              url: this.speechApiUrl,                       // 请求的 URL
              method: "POST",                               // 请求方法
              headers: {
                "Content-Type": "application/json",         // 请求头类型
                Authorization: `Bearer ${api_key}`,         // 添加授权头
              },
              data,                                         // 请求体数据
              responseType: "arraybuffer",                  // 响应类型为数组缓冲区
            });
            return Buffer.from(bufferRes.data) as IOpenAISpeechResponse<T>;     // 返回
Buffer 数据
          case EnvEnum.web:                                 // Web 环境处理
            const blobRes = await axios.request({
              url: this.speechApiUrl,                       // 请求的 URL
              method: "POST",                               // 请求方法
              headers: {
                "Content-Type": "application/json",         // 请求头类型
                Authorization: `Bearer ${api_key}`,         // 添加授权头
              },
              data,         // 请求体数据
              responseType: "blob",                         // 响应类型为 Blob
            });
            const blob = new Blob([blobRes.data]);          // 创建 Blob 对象
            return window.URL.createObjectURL(blob) as IOpenAISpeechResponse<T>;    // 返
回 Blob 的 URL
          default:              // 默认情况
            break;              // 无须处理
        }
      }

      /**
       * 语音转英文
       * @param data 语音转文本的参数数据
       * @param api_key API 密钥
       */
      public async transition(
        data: IOpenAITransitionProps,                       // 语音转换参数
        api_key: string                                     // API 密钥
      ): Promise<string> {
        // 发送请求并返回转换后的文本
        return (
          await axios.request({
            url: this.transitionUrl, // 请求的 URL
```

```
        method: "POST",                                  // 请求方法
        headers: {
          "Content-Type": "application/json",     // 请求头类型
          Authorization: `Bearer ${api key}`,     // 添加授权头
        },
        data,      // 请求体数据
       })
    ).data.text;  // 返回响应中的文本数据
  }
}

export default OpenAIAudio;                          // 导出 OpenAIAudio 类
```

接下来实现另一个实体 Chat。同样，先根据架构设计和 API 端点的类型来实现 Chat 类的类型和伪类，代码如下：

```
import axios from "axios";              // 导入 axios 库用于发送 HTTP 请求
import { cloneDeep } from "lodash";     // 导入 lodash 库的 cloneDeep 方法，用于深拷贝对象
import { Readable } from "stream";      // 导入 Readable 流类型
import { EnvEnum } from "../../utils/request"; // 导入环境枚举类型

// 定义聊天接口的属性
export interface IOpenAIChatProps {
  messages: {                                    // 消息数组
    role: "user" | "assistant" | "system";       // 消息角色：用户、助手或系统
    content: string;                             // 消息内容
  }[];
  model: "gpt-3.5-turbo" | "gpt-3.5-turbo-16k" | "gpt-4";     // 选择模型
  frequency_penalty?: [-2, 2];          // 可选的频率惩罚参数
  max_tokens?: number;                  // 可选的最大令牌数
  temperature?: [0, 2];                 // 可选的温度参数，影响结果的随机性
  stop?: string | string[];             // 可选的停止标记，可以是字符串或字符串数组
  stream?: true;           // 可选的流标识
}

// 定义聊天响应接口
export interface IOpenAIChatResponse {
  answer: string;                       // 回复的内容
  finish_reason: string;                // 结束原因
}

// 定义流响应的返回类型，依据不同环境返回不同类型
export type IOpenAIStreamChatResponse<T> = T extends EnvEnum.node
  ? Readable      // Node 环境下返回 Readable 流
  : T extends EnvEnum.web
  ? ReadableStreamDefaultReader<Uint8Array>      // Web 环境下返回
ReadableStreamDefaultReader
  : undefined;    // 其他情况返回 undefined

class OpenAIChat {
  private chatApiUrl = "https://api.openai.com/v1/chat/completions";// 聊天 API 的 URL
```

```
  constructor() {}                  // 构造函数

  /**
   * 常规返回的 Chat
   * @param data 聊天请求参数
   * @param api_key API 密钥
   * @returns 聊天响应
   */
  public async chat(
    data: IOpenAIChatProps,         // 聊天请求的参数
    api_key: string                 // API 密钥
  ): Promise<IOpenAIChatResponse> {
    // 这里实现聊天请求的逻辑
  }

  /**
   * 流响应返回的 Chat（直接返回流）
   * @param data 聊天请求参数
   * @param api_key API 密钥
   * @param env 环境类型
   */
  public async streamChat<T>(
    data: IOpenAIChatProps,         // 聊天请求的参数
    api_key: string,                // API 密钥
    env: T                          // 环境类型
  ): Promise<IOpenAIStreamChatResponse<T> | undefined> {
    // 这里实现流聊天请求的逻辑
  }
}

export default OpenAIChat;          // 导出 OpenAIChat 类
```

在伪类 OpenAIChat 中，定义了聊天的请求入参类型 IOpenAIChatProps 和响应的类型 IOpenAIChatResponse。响应的类型中返回了 answer 和 finish_reason 两个关键参数，answer 是响应的结果，而 finish_reason 可以用来区分此次是正常返回，还是因 token 不足而导致的答案截断，这在实际业务应用中起到了重要作用。

OpenAIChat 类中包含 chat 和 streamChat 两个方法：chat 方法返回一个固定的 JSON，不需要针对环境进行区分；而 streamChat 方法返回一个可读流，因为在 Node.js 和 Web 端有不同的处理逻辑，所以需要传递 env 参数进行区分。在之前的示例中，同样实现过这两个方法，可以参考进行补全，补全后的完整 OpenAIChat 类代码如下：

```
import axios from "axios";              // 导入 axios 库用于发送 HTTP 请求
import { cloneDeep } from "lodash";     // 导入 lodash 库的 cloneDeep 方法，用于深拷贝对象
import { Readable } from "stream";      // 导入 Readable 流类型
import { EnvEnum } from "../../utils/request";       // 导入环境枚举类型

// 定义聊天接口的属性
export interface IOpenAIChatProps {
```

```
    messages: {                    // 消息数组
      role: "user" | "assistant" | "system";       // 消息角色：用户、助手或系统
      content: string;             // 消息内容
    }[];
    model: "gpt-3.5-turbo" | "gpt-3.5-turbo-16k" | "gpt-4"; // 选择模型
    frequency_penalty?: [-2, 2];     // 可选的频率惩罚参数
    max_tokens?: number;             // 可选的最大令牌数
    temperature?: [0, 2];            // 可选的温度参数，影响结果的随机性
    stop?: string | string[];        // 可选的停止标记，可以是字符串或字符串数组
    stream?: true;                   // 可选的流标识
  }

  // 定义聊天响应接口
  export interface IOpenAIChatResponse {
    answer: string;                  // 回复的内容
    finish_reason: string;           // 结束原因
  }

  // 定义流响应的返回类型，依据不同环境返回不同类型
  export type IOpenAIStreamChatResponse<T> = T extends EnvEnum.node
    ? Readable                       // Node 环境下返回 Readable 流
    : T extends EnvEnum.web
    ? ReadableStreamDefaultReader<Uint8Array>     // Web 环境下返回
ReadableStreamDefaultReader
    : undefined;                     // 其他情况返回 undefined

  class OpenAIChat {
    private chatApiUrl = "https://api.openai.com/v1/chat/completions"; // 聊天 API 的 URL

    constructor() {}                 // 构造函数

    /**
     * 常规返回的 Chat
     * @param data 聊天请求参数
     * @param api_key API 密钥
     * @returns 聊天响应
     */
    public async chat(
      data: IOpenAIChatProps,        // 聊天请求的参数
      api_key: string                // API 密钥
    ): Promise<IOpenAIChatResponse> {
      // 非流响应下的请求不需要额外的逻辑来兼容 Web 和 Node.js
      const chatData = cloneDeep(data);       // 深拷贝聊天数据，避免直接修改原数据
      delete chatData.stream;                 // 删除流标识

      const res = await axios.request({       // 发送 POST 请求
        method: "post",
        url: this.chatApiUrl,                 // API URL
        headers: {
          Authorization: `Bearer ${api_key}`, // 设置授权头
```

```
        "Content-Type": "application/json",  // 设置内容类型为 JSON
      },
      data: chatData,                        // 请求数据
    });

    // 返回聊天响应，提取答案和结束原因
    return {
      answer: res.data?.choices?.[0]?.message?.content || "",      // 答案内容
      finish reason: res.data?.choices?.[0]?.finish reason,        // 结束原因
    };
  }

  /**
   * 流响应返回的 Chat（直接返回流）
   * @param data 聊天请求参数
   * @param api_key API 密钥
   * @param env 环境类型
   */
  public async streamChat<T>(
    data: IOpenAIChatProps,          // 聊天请求的参数
    api_key: string,                 // API 密钥
    env: T                           // 环境类型
  ): Promise<IOpenAIStreamChatResponse<T> | undefined> {
    switch (env) {                   // 根据环境类型处理
      case EnvEnum.node:             // Node 环境
        return (
          await axios.request({      // 发送 POST 请求
            url: this.chatApiUrl,    // API URL
            method: "POST",
            headers: {
              "Content-Type": "application/json",      // 设置内容类型为 JSON
              Authorization: `Bearer ${api_key}`,      // 设置授权头
            },
            data,                    // 请求数据
            responseType: "stream",  // 响应类型为流
          })
        ).data;                      // 返回数据
      case EnvEnum.web:              // Web 环境
        const response = await axios.request({        // 发送 POST 请求
          url: this.chatApiUrl,      // API URL
          method: "POST",
          headers: {
            "Content-Type": "application/json",       // 设置内容类型为 JSON
            Authorization: `Bearer ${api_key}`,       // 设置授权头
          },
          data,                      // 请求数据
        });
        return response.data?.getReader();            // 返回流读取器
      default:
        break;                       // 其他情况不处理
```

```
    }
  }
}

export default OpenAIChat;          // 导出 OpenAIChat 类
```

到这里，已经封装完成了基本的聊天类。仔细思考后会发现，在不同环境下，获得可读流后，通常需要针对流进行封装处理，之后才能获取逐块信息段。这部分逻辑是可以复用的，并不需要每次使用时都重新处理流，而是可以将其封装为高级函数。以一个回调函数作为参数从外部传入封装的函数中。回调函数在每次获取流的块信息后被调用，从而满足不同用户场景的流式调用需要。整个过程的流程图如图 2-25 所示。

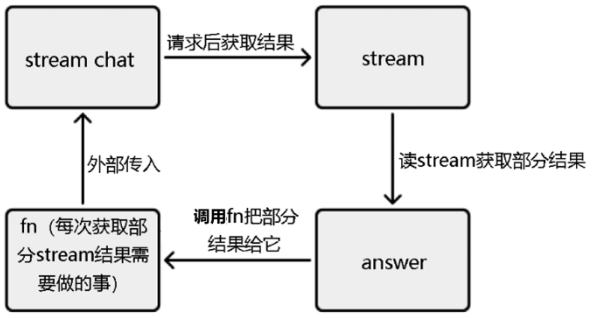

图 2-25　通过高阶函数封装流处理的过程

通过这种方式，用户就可以不再关注流的处理逻辑，只需专注每次获取流的块信息后要执行的回调逻辑。根据上述梳理的逻辑，编写迭代 OpenAIChat 类的代码如下：

```
import axios from "axios";
import { cloneDeep } from "lodash";
import { Readable } from "stream";
import { EnvEnum } from "../../utils/request";

// 之前定义的类型
class OpenAIChat {
  private chatApiUrl = "https://api.openai.com/v1/chat/completions"; // 聊天 API 的 URL
  constructor() {}

 // 其他 chat 方法，常规 Chat 和直接返回流

  /**
   * 流响应返回的 Chat（不返回流，直接处理）
   * @param data 请求数据
   * @param api_key API 密钥
   * @param env 环境类型
   * @param fn 回调函数，处理返回的内容
   */
  public async streamChatCallback(
    data: IOpenAIChatProps,
    api_key: string,
```

```
        env: EnvEnum,
        fn: (answer: string) => void
      ): Promise<void> {
        switch (env) {
          case EnvEnum.node:        // 如果环境是 node
            const stream = await this.streamChat<EnvEnum.node>(data, api key, env); // 获
取流
            stream?.on("data", (chunk: Buffer) => {            // 监听数据事件
              const decoder = new TextDecoder("utf-8");        // 创建文本解码器
              const chunkString = decoder.decode(chunk);       // 解码当前数据块
              const chunks = chunkString
                .split("data:")      // 按"data:"分割数据
                .map((data) => {
                  const trimData = data.trim();        // 去除空白
                  if (trimData === "" || trimData === "[DONE]") {
                    return undefined;                  // 忽略空数据或结束标记
                  }
                  return JSON.parse(trimData);         // 解析 JSON 数据
                })
                .filter((data) => data);               // 过滤掉 undefined 的数据

              chunks.forEach((data) => {
                const token = data.choices[0].delta.content;   // 获取内容

                if (token !== undefined) {
                  // 通过 fn 向外传递流的处理结果
                  fn(token);         // 调用回调函数
                }
              });
            });

            stream?.on("end", () => {
              console.log("全部数据读取完毕");      // 数据读取结束
            });
            break;
          case EnvEnum.web:                        // 如果环境是 web
            const reader = await this.streamChat<EnvEnum.web>(data, api_key, env);  // 获
取流读取器
            if (reader) {
              const { done, value } = await reader.read();     // 读取数据

              if (done) {
                return reader.releaseLock();                   // 如果已完成，释放锁
              }
              const decoder = new TextDecoder("utf-8");        // 创建文本解码器
              const chunk = decoder.decode(value, { stream: true });   // 解码当前值
              const chunks = chunk
                .split("data:")                                // 按"data:"分割数据
                .map((data) => {
                  const trimData = data.trim();                // 去除空白
```

```
            if (trimData === "") {
              return undefined;                    // 忽略空数据
            }
            if (trimData === "[DONE]") {
              return undefined;                    // 忽略结束标记
            }
            return JSON.parse(data.trim());        // 解析 JSON 数据
          })
          .filter((data) => data);                 // 过滤掉 undefined 的数据
        chunks.forEach((data) => {
          const token = data.choices?.[0].delta.content;     // 获取内容
          if (token !== undefined) {
            // 通过 fn 向外传递流的处理结果
            fn(token);          // 调用回调函数
          }
        });
      }
      break;
    default:
      break;                   // 默认情况不做处理
  }
 }
}

export default OpenAIChat;     // 导出 OpenAIChat 类
```

到此，实体层的类已经实现完成，接下来开始实现接入层。

2）实现接入层

步骤 01　首先在 src 目录下创建 utils 目录，用于存放一些通用功能，基类 baseRequest 也存放在这里。接着创建文件 request.ts，用于存放基类的源代码。创建完成后，src 的目录结构如图 2-26 所示。其中，index.ts 是之前初始化过程中的入口文件，后续会进行改造。

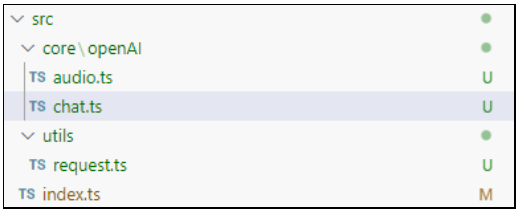

图 2-26　src 的目录结构

步骤 02　根据架构设计图的要求，需要为基类定义 API_KEY 和 env 两个参数，并暴露出获取它们的方法，使得子类可以顺利调用。具体代码如下：

```
export enum EnvEnum {
  unknown = 0,    // 未知环境
  node = 1,       // Node.js 环境
  web = 2,        // Web 环境
}
```

```
interface IRequest {
  getApiKey: () => string;              // 获取 API 密钥的方法
  setApiKey: (api_key: string) => void; // 设置 API 密钥的方法
  getEnv: () => EnvEnum;                // 获取当前环境的方法
}

class BaseRequest implements IRequest {
  private api_key: string;              // 存储 API 密钥
  private env: EnvEnum = EnvEnum.unknown;        // 存储环境类型，默认为未知

  constructor(api_key: string) {
    this.setApiKey(api_key);            // 调用方法设置 API 密钥
    // 初始化环境参数
    if (typeof window !== "undefined") {         // 检查是否在浏览器环境中
      this.env = EnvEnum.web; // 设置为 Web 环境
    } else if (
      typeof process !== "undefined" &&          // 检查是否在 Node.js 环境中
      typeof process.version === "string"
    ) {
      this.env = EnvEnum.node;          // 设置为 Node.js 环境
    }
  }

  public setApiKey(api_key: string) {
    this.api_key = api_key;             // 设置 API 密钥
  }

  public getApiKey() {
    return this.api_key;                // 返回 API 密钥
  }

  public getEnv() {
    return this.env;                    // 返回当前环境类型
  }
}

export default BaseRequest;             // 导出 BaseRequest 类
```

步骤 03　完成基类的实现后，可以开始实现入口文件。该文件需要继承基类，并将对应的参数透传到实体类，修改 src/index.ts 的代码如下：

```
import OpenAIAudio, {
  IOpenAISpeechProps,                   // 引入语音属性接口
  IOpenAITransitionProps,               // 引入过渡属性接口
} from "core/openAI/audio";
import OpenAIChat, { IOpenAIChatProps } from "core/openAI/chat";      // 引入聊天模块及其属性接口
import BaseRequest from "../../utils/request";      // 引入基础请求类

export enum AudioEnum {
```

```
  Speech = "1",                // 音频类型：语音
  Transition = "2",            // 音频类型：过渡
}

// 根据 AudioEnum 类型推导请求类型
export type IOpenAIAudioReq<T> = T extends AudioEnum.Speech
  ? IOpenAISpeechProps         // 如果类型是 Speech，则为语音属性接口
  : IOpenAITransitionProps;    // 否则为过渡属性接口

class LLMRequest extends BaseRequest {
  constructor(api_key: string) {
    super(api_key);            // 调用父类构造函数，设置 API 密钥
  }

  public setApiKey(api_key: string) {
    super.setApiKey(api_key);  // 调用父类的方法设置 API 密钥
  }
}

/**
 * openAI 语音相关功能，支持文本转语音，语音转英文
 * @param data 请求数据，包含音频相关参数
 * @param audioType 音频类型，指明是文本转语音还是语音转英文
 * @returns 返回处理结果
 */
public async openAIAudio<T>(data: IOpenAIAudioReq<T>, audioType: T) {
  const openAIAudioEntity = new OpenAIAudio();       // 创建 OpenAIAudio 实例
  switch (audioType) {
    case AudioEnum.Speech:             // 如果音频类型是语音
      return await openAIAudioEntity.speech(
        data as IOpenAISpeechProps,    // 将 data 类型转换为语音属性接口
        super.getApiKey(),             // 获取 API 密钥
        super.getEnv()                 // 获取当前环境
      );
    case AudioEnum.Transition:         // 如果音频类型是过渡
      return await openAIAudioEntity.transition(
        data as IOpenAITransitionProps, // 将 data 类型转换为过渡属性接口
        super.getApiKey()              // 获取 API 密钥
      );
    default:
      return;      // 默认情况下不返回任何内容
  }
}

/**
 * 常规调用 openAI Chat 系列模型获取结果
 * @param data 请求数据，包含聊天相关参数
 * @returns 返回聊天结果
 */
public async openAIChat(data: IOpenAIChatProps) {
```

```
    const { stream = false } = data;              // 解构获取 stream 参数，默认为 false
    const api key = super.getApiKey();            // 获取 API 密钥
    const openAIChatEntity = new OpenAIChat();    // 创建 OpenAIChat 实例

    if (stream) {       // 如果需要流式响应
      return await openAIChatEntity.streamChat(data, api key, super.getEnv());   // 调
用流式聊天方法
    } else { // 否则
      return await openAIChatEntity.chat(data, api key);        // 调用常规聊天方法
    }
  }

  /**
   * 常规调用 openAI Chat 系列模型获取结果
   * @param data 请求数据，包含聊天相关参数
   * @returns 返回聊天结果
   */
  public async openAIChat(data: IOpenAIChatProps) {
    const { stream = false } = data;              // 解构获取 stream 参数，默认为 false
    const api_key = super.getApiKey();            // 获取 API 密钥
    const openAIChatEntity = new OpenAIChat();    // 创建 OpenAIChat 实例

    if (stream) {       // 如果需要流式响应
      return await openAIChatEntity.streamChat(data, api_key, super.getEnv());   // 调
用流式聊天方法并返回结果
    } else {            // 否则
      return await openAIChatEntity.chat(data, api_key);   // 调用常规聊天方法并返回结果
    }
  }

    /**
     * 支持将 openAI Chat 系列模型对流响应的结果直接回调给具体 token 的处理函数
     * @param data
     * @param fn
     */
  public async openAIStreamChatCallback(
    data: IOpenAIChatProps,          // 请求数据，包含聊天相关参数
    fn: (token: string) => void      // 回调函数，用于处理流中的每个 token
  ) {
    const { stream = false } = data; // 解构获取 stream 参数，默认为 false
    if (!stream) { // 如果不是流式响应
      throw new Error(
        "openAIStreamChatCallback 方法只有在流响应场景下才可以使用"   // 抛出错误，提示该方法仅适
用于流响应
      );
    }
    const openAIChatEntity = new OpenAIChat();         // 创建 OpenAIChat 实例
    await openAIChatEntity.streamChatCallback(
      data,                  // 传入请求数据
      super.getApiKey(),     // 获取 API 密钥
```

```
        super.getEnv(),         // 获取当前环境
        fn                      // 传入回调函数
      );
    }
    }

    export default LLMRequest; // 导出 LLMRequest 类
```

在上述代码中，实现了 openAIAudio、openAIChat 和 openAIStreamChatCallback3 个方法，分别用于音频场景、聊天场景以及流回调场景的应用。到此为止，库的核心源代码就完成了，接下来需要改造库的打包方式，为库发布做好准备。

3. 改造打包方式，兼容 CommonJS 和 ES 模块

在初始化过程中，使用 tsc 完成了项目的打包，但这种打包方式只打包了 ESM 模块，因此无法满足 Node.js 环境的使用。在 Node.js 环境中，CommonJS 模块是兼容性更好的模块调用方式，因此需要优化项目的打包方式，以兼容 Node.js 环境的使用。

步骤 01　创建 tsconfig.cjs.json 和 tsconfig.esm.json 文件，这里仍然使用 tsc 来完成打包，但创建两套打包配置，分别用于 CommonJS 和 ES 模块。在 tsconfig.cjs.json 中写入如下配置代码：

```
    {
      "compilerOptions": {
        "target": "ES5",                    // 指定输出的 JavaScript 目标版本
        "module": "CommonJS",               // 指定模块系统，这里选择的是 CommonJS
        "declaration": true,                // 启用生成声明文件
        "declarationDir": "./dist/types",   // 类型声明文件的输出目录
        "lib": ["es2021", "dom"],           // 指定要包含的类型声明库
        "outDir": "dist/cjs",      // 设置输出目录，编译后的 JavaScript 文件会被放在这个目录下
        "esModuleInterop": true,   // 允许 CommonJS 模块与 ES 模块之间具有更好的互操作性，使 import
语句能更好地配合默认导出
        "sourceMap": true,         // 生成源代码映射文件，方便调试时定位原始 TypeScript 源代码
        "baseUrl": "./src",        // 设置模块解析的基本目录，对路径映射有用
        "strictNullChecks": true,  // 开启 ts 可选链提示
        "moduleResolution": "Node" // 模块解析策略。"node"适用于 Node.js 环境，它遵循 Node.js 的模
块解析规则
      },
      "include": ["src"], // 包含哪些文件或目录进行编译，这里是编译 src 目录下的所有 TypeScript 文件
      "exclude": ["node_modules", "**/*.test.ts", "dist"] // 排除不参与编译的文件或目录，这里
是排除 node_modules 目录以及所有以.test.ts 结尾的测试文件和 dist 打包文件
    }
```

在 tsconfig.esm.json 中写入另外一套配置代码：

```
    {
      "compilerOptions": {
        "target": "ESNext",                 // 指定输出的 JavaScript 目标版本
        "module": "ESNext",                 // 指定模块系统，这里选择的是 ES 模块
        "declaration": true,                // 启用生成声明文件
        "declarationDir": "./dist/types",   // 类型声明文件的输出目录
```

```
        "lib": ["es2021", "dom"],            // 指定要包含的类型声明库
        "outDir": "dist/esm",       // 设置输出目录，编译后的 JavaScript 文件会被放在这个目录下
        "esModuleInterop": true,    // 允许 CommonJS 模块与 ES 模块之间具有更好的互操作性，使 import
语句能更好地配合默认导出
        "sourceMap": true,          // 生成源代码映射文件，方便调试时定位原始 TypeScript 源代码
        "baseUrl": "./src",         // 设置模块解析的基本目录，对路径映射有用
        "strictNullChecks": true,  // 开启 ts 可选链提示
        "moduleResolution": "Node" // 模块解析策略。"node"适用于 Node.js 环境，它遵循 Node.js 的模
块解析规则
      },
      "include": ["src"], // 包含哪些文件或目录进行编译，这里是编译 src 目录下的所有 TypeScript 文件
      "exclude": ["node_modules", "**/*.test.ts", "dist"] // 排除不参与编译的文件或目录，这里
是排除 node_modules 目录以及所有以.test.ts 结尾的测试文件和 dist 打包文件
    }
```

步骤 02　在终端执行以下打包命令，指定配置文件分别完成 CommonJS 和 ES 模块的打包。

```
tsc -p tsconfig.cjs.json && tsc -p tsconfig.esm.json
```

打包完成后，dist 目录结构如图 2-27 所示。

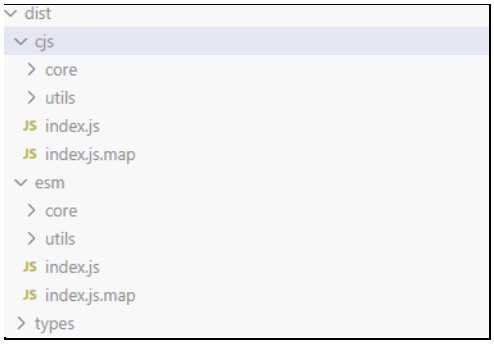

图 2-27　分模块打包的目录结构

步骤 03　修改 package.json 中的入口文件配置，以使包可以在不同环境导入方式下自动选择使用 CommonJS 模块还是 ES 模块。修改后的 package.json 配置如下：

```
{
  // 其他配置
  "main": "dist/cjs/index.js",
  "module": "dist/esm/index.js",
  "types": "dist/types/index.d.ts",
  // 其他配置
}
```

其中，main 配置默认为 CommonJS 导入方式的引用入口，而 module 配置默认为 ES 导入方式的引用入口。至此，打包改造已完成，提交代码后进入测试环节。

4. 编写单元测试

在正式发布请求库之前，需要对库的主要功能进行测试。从研发角度看，除了直接体验的功能

测试，单元测试也是一个非常重要的手段。现在需要为库补全代码层面的单元测试用例代码。

什么是单元测试？简单来说，单元测试是基于组件和代码逻辑层面的自动化测试手段，它写在代码中。它可以确保在每一次迭代中，新的代码至少维持了原有版本的重要功能，从而避免非预期的破坏性修改（breaking change）。虽然单元测试不能替代常规功能测试的作用，但在维持代码质量方面是非常重要的一环。

在 JavaScript 中，通常使用 Jest 进行单元测试，它由 Facebook 开发并维护，适用于 React、Angular、Vue.js 等现代 JavaScript 应用程序。Jest 的特点包括轻量安装、社区生态完备，功能强大、易用且容易上手。Jest 常用关键词如表 2-3 所示。

表 2-3　Jest 单元测试常用关键词

关 键 词	说　明
describe	用例集，用于对用例分类，表示某个方向的一系列用例
test	用于定义一个测试用例，其中包含 *n* 个断言，用于验证某个功能是否正确
expect	断言，用于陈述某个功能事实，判断逻辑是否符合预期

下面开始为源代码覆盖单元测试。

步骤 01　在项目中执行以下命令以安装 Jest 测试所需的核心依赖：

```
npm install jest ts-jest @types/jest --save-dev
```

步骤 02　安装完依赖后，在根目录创建一个 jest.config.js 文件，用于保存 Jest 测试的一些配置，比如测试范围、哪些代码不需测试、是否生成测试覆盖率结果等。我们在其中写入如下配置代码：

```
module.exports = {
  preset: "ts-jest",                // 预置类型
  testEnvironment: "node",          // 设置环境
  testMatch: ["**/__tests__/**/*.+(js|ts)", "**/?(*.)+(test).+(js|ts)"], // 测试文件匹配模式
  clearMocks: true,                 // 是否清除每个测试前的 mocks
  resetMocks: false,                // 是否在每次测试前自动重置模拟函数
  coverageDirectory: "coverage",    // 覆盖率报告目录
  transform: {
    // ts 需要额外编译
    "^.+\\.(js|ts)$": "ts-jest",
  },
  // 模块文件的扩展名
  moduleFileExtensions: ["js", "ts"],
};
```

步骤 03　配置完成后，可以在 package.json 的脚本中加入代码，以监听测试和覆盖率（一次性测试脚本），分别对应 test:watch 和 test 命令。package.json 最终配置如下：

```
{
  // 其他配置
  "scripts": {
    "build": "tsc",
```

```
    "test": "jest --coverage",
    "test:watch": "jest --watch"
  },
  // 其他配置
}
```

现在可以开始正式编写单元测试代码了。这次测试的范围主要为 core 文件夹下的 OpenAIAudio 和 OpenAIChat 实体类，以及 BaseRequest 基类。由于入口类的逻辑仅为透传，并没有太多测试的价值，因此不在本次测试范围内。

对于 OpenAIAudio 类，它实现了文本转语音和语音翻译两个方法。文本转语音在 Node.js 和 Web 环境中会有不同的返回类型，而语音翻译虽然返回类型相同（均为字符串），但输入参数在不同环境中是有差异的。因此，这两个 API 都需要针对环境来设计用例，如图 2-28 所示。

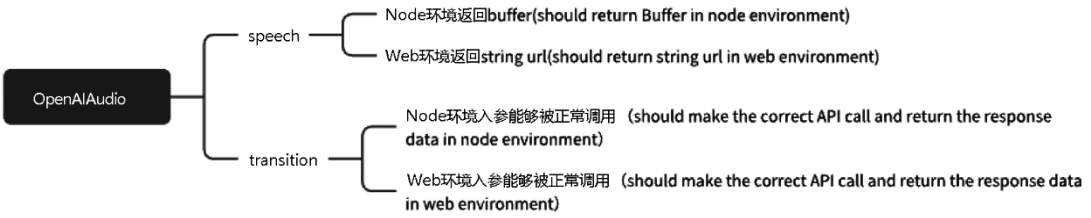

图 2-28　OpenAIAudio 类的测试用例

根据设计的测试用例，编写自动化测试代码。在 openAI 目录下创建__tests__文件夹来存放单元测试；在__tests__文件夹中创建 audio.test.ts 文件作为 OpenAIAudio 类的测试文件，并在 audio.test.ts 中写入如下代码：

```
import axios from "axios";
import FormData from "form-data";
import * as fs from "fs";
import { EnvEnum } from "../../../utils/request";
import OpenAIAudio, {
  IOpenAISpeechProps,
  IOpenAITransitionProps,
} from "../audio";

const MOCK_API_KEY = "mock-api-key";        // 模拟的 API 密钥
const MOCK_SPEECH_DATA: IOpenAISpeechProps = {
  model: "tts-1",              // 模拟的语音合成模型
  input: "Hello World!",       // 输入文本
  voice: "alloy",              // 语音类型
};
jest.mock("axios");            // 模拟 axios 库

describe("OpenAIAudio", () => {
  let openAIAudio: OpenAIAudio;

  beforeEach(() => {
```

```
  openAIAudio = new OpenAIAudio();    // 每个测试前创建一个 OpenAIAudio 实例
  // 模拟 axios.post 请求
  (axios.request as jest.Mock).mockResolvedValue({ data: "mock-data" });

  global.Blob = jest.fn().mockImplementation((parts, options) => {
    return {
      size: 0,                    // 模拟 Blob 对象的大小
      type: "",                   // 模拟 Blob 对象的类型
    };
  });

  global.File = jest.fn().mockImplementation((name, data) => {
    return {
      name,                       // 文件名称
      lastModified: Date.now(),   // 文件最后修改时间
      size: data.length || 0,     // 文件大小
    };
  });

  // @ts-ignore
  global.window = {};         // 模拟浏览器环境中的 window 对象
  // @ts-ignore
  window.URL = {};
  window.URL.createObjectURL = jest.fn(() => {
    return "";                // 模拟 URL.createObjectURL 方法
  });
});

afterEach(() => {
  jest.clearAllMocks();       // 每个测试后清除所有模拟
});

describe("speech()", () => {
  it("should return Buffer in node environment", async () => {
    // 模拟 Node.js 环境下的 responseType: 'arraybuffer'
    const mockBufferRes = Buffer.from("mock-buffer");  // 模拟返回的 Buffer
    (axios.request as jest.Mock).mockResolvedValueOnce({
      data: mockBufferRes,    // 模拟返回的响应数据
    });

    const result = await openAIAudio.speech(
      MOCK_SPEECH_DATA,
      MOCK API KEY,
      EnvEnum.node
    );

    expect(result).toEqual(mockBufferRes);     // 断言返回结果应与 mockBufferRes 相等
    expect(axios.request).toHaveBeenCalledWith(
```

```
          expect.objectContaining({
            url: "https://api.openai.com/v1/audio/speech", // API URL
            method: "POST",
            headers: {
              "Content-Type": "application/json",          // 请求头
              Authorization: `Bearer ${MOCK_API_KEY}`,     // 授权信息
            },
            data: MOCK_SPEECH_DATA,                        // 请求数据
            responseType: "arraybuffer",                   // 响应类型
          })
        );
      });

      it("should return string url in web environment", async () => {
        const mockBlob = new Blob(["mock-blob"]);          // 模拟 Blob 对象
        (axios.request as jest.Mock).mockResolvedValueOnce({ data: mockBlob });

        const result = await openAIAudio.speech(
          MOCK SPEECH DATA,
          MOCK API KEY,
          EnvEnum.web
        );

        expect(window.URL.createObjectURL).toHaveBeenCalledWith(mockBlob);
        expect(typeof result).toBe("string");          // 结果类型应为字符串
      });
    });

    describe("transition()", () => {
      it("should make the correct API call and return the response data in node environment",
async () => {
        const expectedResult = { text: "mock-transition-result" };
        (axios.request as jest.Mock).mockResolvedValueOnce({
          data: expectedResult,                        // 模拟返回的响应数据
        });

        const MOCK_TRANSITION_DATA = new FormData();   // 创建一个新的 FormData 对象
        MOCK_TRANSITION_DATA.append(
          "file",
          fs.createReadStream("./download.mp3")        // 添加文件数据
        );
        MOCK_TRANSITION_DATA.append("model", "whisper-1"); // 添加模型信息

        const result = await openAIAudio.transition(
          MOCK TRANSITION DATA,
          MOCK API KEY
        );

        expect(result).toEqual(expectedResult);        // 断言返回结果应与期望相等
```

```
          expect(axios.request).toHaveBeenCalledWith(
            expect.objectContaining({
              url: "https://api.openai.com/v1/audio/translations",
              method: "POST",
              headers: {
                Authorization: `Bearer ${MOCK API KEY}`, // 授权信息
              },
              data: MOCK TRANSITION DATA,                // 请求数据
            })
          );
        });

        it("should make the correct API call and return the response data in web environment",
async () => {
          const MOCK_TRANSITION_DATA: IOpenAITransitionProps = {
            // @ts-ignore
            file: new File(["mock-file"], "mock-file.mp3"),       //模拟文件数据
            model: "whisper-1",         // 模拟数据信息
          };
          const expectedResult = { text: "mock-transition-result" }; // 授权期望的返回结果
          (axios.request as jest.Mock).mockResolvedValueOnce({
            data: expectedResult,       // 模拟返回的响应数据
          });

          const result = await openAIAudio.transition(
            MOCK_TRANSITION_DATA,
            MOCK_API_KEY
          );

          expect(result).toEqual(expectedResult);        // 断言返回结果应于期望相等
          expect(axios.request).toHaveBeenCalledWith(
            expect.objectContaining({
              url: "https://api.openai.com/v1/audio/translations",
              method: "POST",                            // 请求方法
              headers: {
                Authorization: `Bearer ${MOCK_API_KEY}`, // 授权信息
              },
              data: MOCK_TRANSITION_DATA,                // 请求信息
            })
          );
        });
      });
    });
```

在上面的用例中，分别实现了对 speech 和 transition 两个 API 在不同环境下的测试。因为用例运行在 Node.js 环境，缺少 Web 环境的 Blob、File 等原生 API，所以在 beforeEach（每个用例执行前的周期）中使用 mock 的方式模拟了这部分 API，以确保测试的顺利运行。除了 mock 外，在测试 speech 方法时，使用了真实的可读流进行模拟。这个场景并不难创建，其中使用的 download.mp3 可以用之前 AI 生成的音频文件替代即可。

对于 OpenAIChat 实体类，封装了 chat、streamChat 和 streamChatCallback 三个方法，其中 chat 方法不涉及环境的变化，而 streamChat 和 streamChatCallback 方法都涉及环境的处理。在实际的测试中，模拟不同环境的 stream 是很困难的，因为 stream 不像常规的构造类，涉及复杂的内部构造，所以对于 chat 实体类的测试，只测试常规 chat 和 stream 响应中的 Node.js 环境，其余测试则通过功能测试完成。具体的 OpenAI Chat 类的测试用例设计如图 2-29 所示。

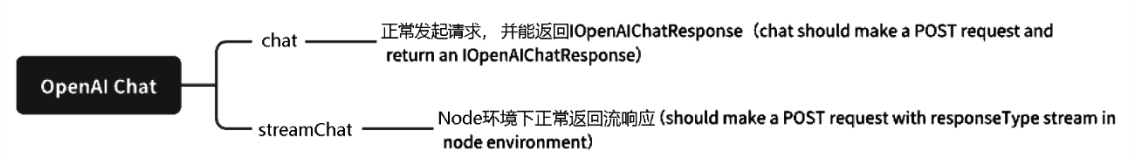

图 2-29　OpenAI Chat 类的测试用例

下面根据测试用例设计开始实现自动化测试代码。同样在__tests__目录中创建 chat.test.ts 文件，用于存放 chat 实体类的测试代码，并在其中编写如下代码：

```
    import axios from "axios";              // 导入 axios 用于 HTTP 请求
    import { Readable } from "stream";      // 导入 Readable 流
    import { EnvEnum } from "../../../utils/request";        // 导入环境枚举
    import OpenAIChat, { IOpenAIChatProps } from "../chat";// 导入 OpenAIChat 类及其属性接口

    jest.mock("axios");                     // 使用 jest.mock 模拟 axios 模块

    describe("OpenAIChat", () => {
      let openAIChat: OpenAIChat;           // 声明 OpenAIChat 实例

      beforeEach(() => {
        openAIChat = new OpenAIChat();      // 每个测试前创建一个 OpenAIChat 实例
      });

      describe("chat", () => {
        test("chat should make a POST request and return an IOpenAIChatResponse", async ()
=> {
          const mockResponse = {
            choices: [
              { message: { content: "mock answer" }, finish_reason: "finished" },   // 模
拟响应数据
            ],
          };

          (axios.request as jest.Mock).mockResolvedValueOnce({     // 模拟 axios 请求返回值
            data: mockResponse,     // 设置返回的数据
            status: 200,            // 设置返回的状态码
            headers: { "Content-Type": "application/json" },       // 设置返回的头部信息
          });

          const testData: IOpenAIChatProps = {                     // 测试数据
```

```
          messages: [{ role: "user", content: "Hello" }],   // 用户消息
          model: "gpt-3.5-turbo",                           // 使用的模型
        };
        const apiKey = "test_api_key";                      // 测试 API 密钥

        const result = await openAIChat.chat(testData, apiKey); // 调用 chat 方法并获取结果

        expect(result).toEqual({                            // 断言返回结果
          answer: "mock answer",                            // 期望返回的回答
          finish_reason: "finished",                        // 期望返回的完成原因
        });
      });
    });

    describe("streamChat", () => {
      test("should make a POST request with responseType stream in node environment", async
() => {
        const mockStream = new Readable();                          // 创建可读流的模拟
        (axios.request as jest.Mock).mockResolvedValueOnce({        // 模拟 axios 请求返回值
          data: mockStream,           // 设置返回的数据为可读流
          status: 200,                // 设置返回的状态码
          headers: { "Content-Type": "application/json" },          // 设置返回的头部信息
        });

        const testData: IOpenAIChatProps = {                        // 测试数据
          messages: [{ role: "user", content: "Hello" }],           // 用户消息
          model: "gpt-3.5-turbo",                                   // 使用的模型
          stream: true,                                             // 指定为流式响应
        };
        const apiKey = "test_api_key";                              // 测试 API 密钥

        const result = await openAIChat.streamChat<EnvEnum.node>(   // 调用 streamChat 方
法
          testData,
          apiKey,
          EnvEnum.node                                      // 指定环境为 Node.js
        );

        expect(result).toBeInstanceOf(Readable);            // 断言结果为 Readable 实例
      });
    });
  });
```

相比 Audio 的用例，Chat 的用例要简单得多，因为它避免了一些复杂的场景逻辑，只模拟了一些基本的 API，例如只读流（Readable）。这样一来，Chat 的测试覆盖度可能较低。在后续的功能测试中，需要特别注重这一点，并补充相关的用例，以确保测试的全面性和准确性。

最后测试基类 BaseRequest，在该类中管理了所有子类的鉴权参数 API_KEY 和环境参数 env，是一个比较重要的类。基类 BaseRequest 的测试用例如图 2-30 所示。

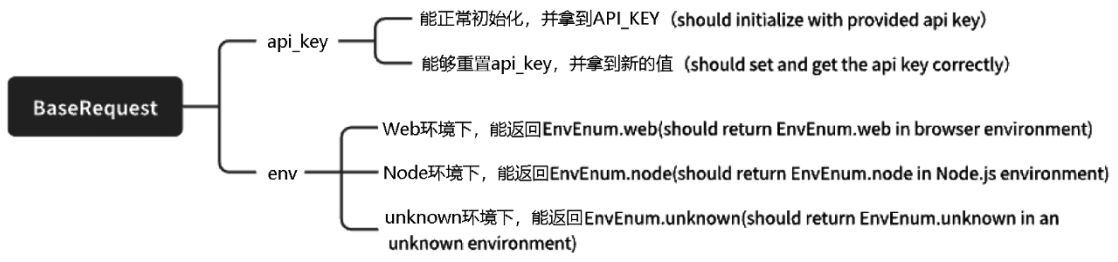

图 2-30　基类 BaseRequest 的测试用例

接下来，根据设计的测试用例实现用例代码。在 request.ts 的同级目录下创建目录__tests__，在__tests__中创建 request.test.ts 文件，用于存放代码。截至目前的项目目录结构如图 2-31 所示。

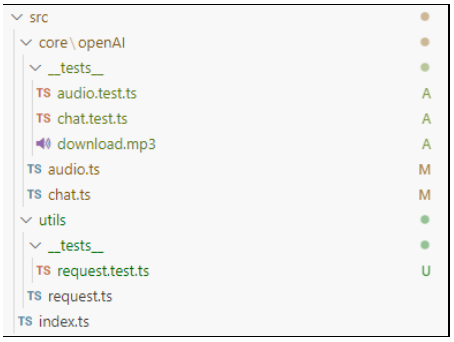

图 2-31　项目目录结构

在 request.test.ts 中写入如下用例代码：

```
import BaseRequest, { EnvEnum } from "../request";  // 导入 BaseRequest 类和环境枚举

describe("BaseRequest", () => {
  let baseRequest: BaseRequest;

  describe("api key", () => {
    beforeEach(() => {
      baseRequest = new BaseRequest("test api key"); // 每个测试前初始化 BaseRequest 实例，传入测试用的 API 密钥
    });

    test("should initialize with provided api key", () => {
      expect(baseRequest.getApiKey()).toBe("test api key");  // 断言获取的 API 密钥是否与预期一致
    });

    test("should set and get the api key correctly", () => {
      const newApiKey = "new test api key";      // 定义新的 API 密钥
      baseRequest.setApiKey(newApiKey);          // 设置新的 API 密钥
      expect(baseRequest.getApiKey()).toBe(newApiKey);   // 断言获取的 API 密钥是否为新设置的值
    });
```

```
    });

    describe("getEnv()", () => {
      test("should return EnvEnum.web in browser environment", () => {
        // 在 Jest 测试环境下模拟浏览器环境
        // @ts-ignore
        global.window = {};
        // @ts-ignore
        delete global.window.process;     // 删除 process 属性以模拟浏览器环境
        baseRequest = new BaseRequest("test api key");     // 初始化 BaseRequest 实例
        expect(baseRequest.getEnv()).toBe(EnvEnum.web);    // 断言环境返回值应为
EnvEnum.web
      });

      test("should return EnvEnum.node in Node.js environment", () => {
        // @ts-ignore
        global.window = undefined;        // 确保 global.window 为 undefined
        Object.defineProperty(global, "process", {
          value: {
            version: "mock-version-string",   // 模拟 process 对象
          },
        });
        baseRequest = new BaseRequest("test_api_key");     // 初始化 BaseRequest 实例
        expect(baseRequest.getEnv()).toBe(EnvEnum.node);   // 断言环境返回值应为
EnvEnum.node
      });

      test("should return EnvEnum.unknown in an unknown environment", () => {
        // 模拟一个既没有 window 也没有 process 的环境
        // @ts-ignore
        global.window = undefined;   // 确保 global.window 为 undefined
        // @ts-ignore
        delete global.process;       // 删除 process 属性
        baseRequest = new BaseRequest("test_api_key");     // 初始化 BaseRequest 实例
        expect(baseRequest.getEnv()).toBe(EnvEnum.unknown);     // 断言环境返回值应为
EnvEnum.unknown
      });
    });
  });
```

由于无法直接切换到真实的 window 环境或未知环境下，因此这里仍然使用 mock 的方式来模拟对应场景的环境，以实现相应的测试用例。

到此所有的用例已实现，读者可能发现，用例的测试代码与之前示例的调用非常相似，这正是单元测试的第二个作用——文档用例化。使用者可以通过单元测试了解 API 在实际场景中的调用方式，以及通过断言了解哪些返回或调用对开发者至关重要。一个好的测试用例能够帮助用户深度了解和使用源代码。

现在在终端执行命令 npm run test，查看测试用例的结果，效果如图 2-32 所示。

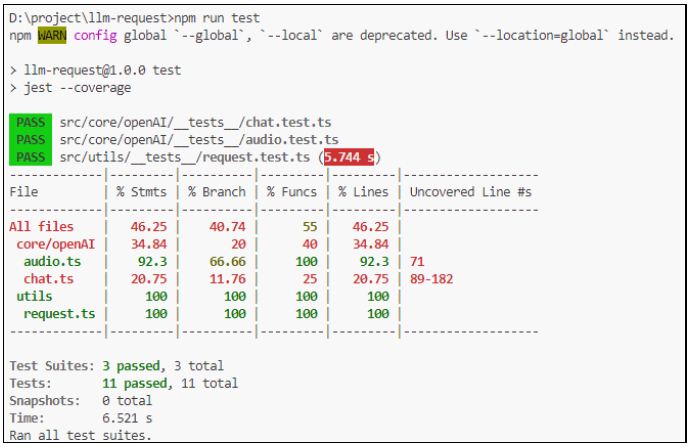

图 2-32　测试用例执行的结果

可以看到，所有用例都通过了测试。表格中的数据展示了测试覆盖率的结果，表示对应的用例代码覆盖了源代码中逻辑的比例。可以发现，audio 和 request 的覆盖率都比较高，而 chat 的覆盖率较低，这是因为 stream 的模拟比较复杂，所以这部分的测试将放到功能测试中完成。

测试完成后，在根目录下会生成一个 coverage 文件，里面存放了本次的测试报告。我们在浏览器中打开 coverage/lcov-report/index.html，查看本次测试中哪些代码片段未被覆盖，效果如图 2-33 所示。值得一提的是，用例覆盖率的 coverage 文件也可以放入.gitignore 文件以忽略，通常不需要提交到云端仓库。

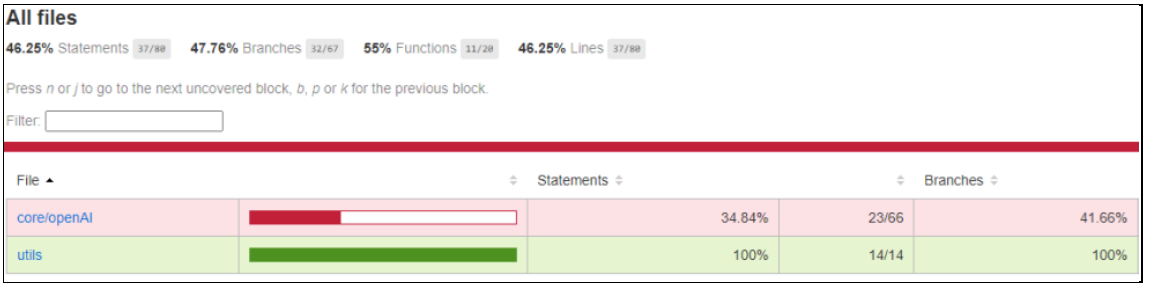

图 2-33　coverage/lcov-report/index.html 的效果

5. 在本地创建库软链进行测试

前面通过单元测试建立了初步的代码逻辑测试，但在实际开发中，单元测试并不能完全替代功能测试。我们还需要从用户的视角出发，调用包以完成相应工作，这样才能最直接地保证包的可用性。

如何测试本地开发的包的可用性呢？直接复制代码或项目进行调用，这样效率较低且不够可靠。npm 支持为本地仓库建立软链，这样在本地就能像调用第三方依赖一样使用本地包。

步骤 01　首先，在终端执行 npm run build 命令，因为 package.json 中配置的入口文件为 dist 打包文件中的内容，所以需要保证 dist 中的内容是最新的。然后，开始配置项目的软链，在项目的根目录打开终端，执行以下命令：

```
npm link
npm link llm-request
```

第一条命令将本地的仓库 llm-request 设置为一个本地软链，这样就能在本地使用 llm-request；第二条命令将 llm-request 配置到当前仓库 llm-request 中。这个过程听起来有点绕，最直接的理解是：现在可以调用自己了。打开 node_modules 仓库，可以看到有一个依赖目录就是 llm-request，如图 2-34 所示。需要注意的是，软链应在安装依赖之前建立，因为每次重新安装依赖时，都会导致软链被重置。软链并不是一个真实的第三方依赖，而是以类似引用地址的方式将本地文件“冒充”为一个依赖引用。

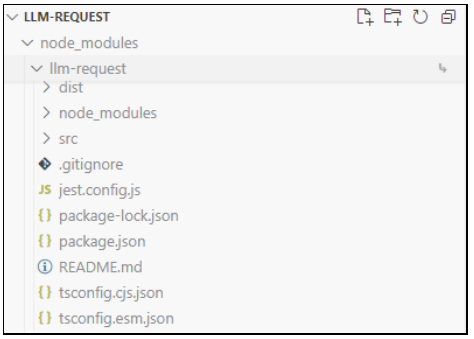

图 2-34　软链成功后的 node_modules 新增了本地的 llm-request

步骤 02　在 src 目录下创建一个文件夹 demos，用来存放一些本地调用库的示例。同时，在 jest.config.js 中加入忽略 demos 的配置。因为这个目录并不作为核心源代码，而仅仅是调用 API 的示例，所以不作为覆盖率考虑。调整后的 jest.config.js 内容如下：

```
module.exports = {
  preset: "ts-jest",
  testEnvironment: "node",
  testMatch:["**/__tests__/**/*.+(js|ts)","**/?(*.)+(test).+(js|ts)"],
  clearMocks: true,
  resetMocks: false,
  coverageDirectory: "coverage",
  transform: {
     "^.+\\.(js|ts)$": "ts-jest",
  },
  moduleFileExtensions: ["js", "ts"],
  // 忽略 demos 文件夹
  modulePathIgnorePatterns: ["<rootDir>/src/demos/"],
};
```

步骤 03　在 demos 文件夹中测试包在 Node.js 环境下能否正常使用。在 demos 中创建 audio.demo.cjs 文件，并写入如下代码：

```
const path = require("path");
const fs = require("fs");
const FormData = require("form-data");
const LLMRequest = require("llm-request").default;

const audioTest = async () => {
```

```
  const LLMRequestEntity = new LLMRequest(""); // 换成你的 API KEY
  console.log("开始生成音频");
  const audioBuffer = await LLMRequestEntity.openAIAudio(
    {
      model: "tts-1",
      input: "我正在试用 llm-request 中的 openAIAudio 方法生成音频",
      voice: "alloy",
    },
    "1"
  );

  const audioPath = path.resolve(__dirname, "audio.mp3");
  if (audioBuffer) {
    await fs.promises.writeFile(audioPath, audioBuffer);
    console.log("音频生成完成，开始将音频转英文");

    let data = new FormData();
    data.append("file", fs.createReadStream(audioPath));
    data.append("model", "whisper-1");
    const audioText = await LLMRequestEntity.openAIAudio(data, "2");
    console.log(`音频转英文成功，对应文本结果为：${audioText}`);
  } else {
    console.log("未获取有效音频");
  }
};

audioTest();
```

在这个脚本中，同时测试了音频类的生成和翻译。使用 node 执行该脚本后，发现在当前目录下生成了一个 audio.mp3。播放该音频时发现有语音输出，同时终端也正确显示了文本翻译，终端效果如图 2-35 所示。

```
D:\project\llm-request>node src/demos/audio.demo.cjs
开始生成音频
音频生成完成，开始将音频转英文
音频转英文成功，对应文本结果为：I am using the OpenAI audio method in LM Request to generate audio.
```

图 2-35　audio.demo.cjs 的终端执行结果

步骤 04　测试 Chat 在 Node.js 环境下能否正常被调用。同样在 demos 目录下创建 chat.demo.cjs，写入如下代码：

```
const LLMRequest = require("llm-request").default;

const chatTest = async () => {
  const LLMRequestEntity = new LLMRequest(""); // 换成你的 API_KEY
  console.log("开始测试 openAIChat - 常规");
  const chatRes = await LLMRequestEntity.openAIChat({
    model: "gpt-3.5-turbo",
    messages: [
      {
        role: "user",
```

```
          content: "你好",
        },
      ],
    });
    console.log(`常规 chat 结果为:${JSON.stringify(chatRes)}`);
    // openAIStreamChatCallback 底层调用了 openAIChat 流，直接测试它即可
    console.log("开始测试 openAIChat - 流及 openAIStreamChatCallback");
    await LLMRequestEntity.openAIStreamChatCallback(
      {
        model: "gpt-3.5-turbo",
        messages: [
          {
            role: "user",
            content: "你好",
          },
        ],
        stream: true,
      },
      (answer) => console.log(answer)
    );
  };

  chatTest();
```

在这个示例中，测试了常规的聊天输出和高阶函数的流调用。这里并没有对 chat 的流输出进行逻辑测试，因为高阶函数的流调用底层逻辑直接调用了流式聊天的方法，所以流式聊天方法的测试可以通过高阶函数的流调用间接完成。在终端执行这个脚本时，可以看到能得到预期的输出，效果如图 2-36 所示。

```
D:\project\llm-request>node src/demos/chat.demo.cjs
开始测试openAIChat - 常规
常规chat结果为:{"answer":"你好！有什么可以帮助你的吗？如果你有任何问题或需要帮助，请随时告诉我。","finish_reason":"stop"}
开始测试openAIChat - 流及openAIStreamChatCallback

你
好
！
有
什
么
可以
帮
助
你
的
吗
？
全部数据读取完毕
```

图 2-36　chat.demo.cjs 的终端执行结果

步骤 05　测试包在 Web 环境下能否按预期调用，可以使用 create-react-app 脚手架快速搭建一个 react 项目以完成测试。切换到一个非项目路径，在终端中执行以下命令创建 react 项目：

```
npm install -g create-react-app
npx creact-react-app llm-request-test-app
```

命令执行后，稍等片刻即可在对应目录下看到一个 llm-request-test-app 文件夹。用 VSCode 打开该文件夹，并在终端中执行 npm link llm-request 命令，将请求库软链到依赖中。需要注意的是，创建的 react 项目默认使用 react18，而 react18 在严格模式下默认执行 2 次页面渲染，因此需要去掉严格模式，以避免对后续测试造成影响。打开 src/index.tsx，修改代码如下：

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import App from "./App";
import reportWebVitals from "./reportWebVitals";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<App />);

reportWebVitals();
```

去掉严格模式后，就可以开始测试了。

（1）测试音频模块：

首先，打开 src/App.js 入口文件，修改代码如下：

```
import { useEffect, useState } from "react"; // 导入 React 的 useEffect 和 useState 钩子
import { default as LLMRequest } from "llm-request";     // 导入 llm-request 模块

function App() {
  const [audioUrl, setAudioUrl] = useState("");      // 音频 URL 状态
  const [audioText, setAudioText] = useState("");    // 音频文本状态

  const audioTest = async () => {
    const LLMRequestEntity = new LLMRequest("");     // 创建 LLMRequest 实例，替换为你的
API KEY
    console.log("开始生成音频");

    const audioUrl = await LLMRequestEntity.openAIAudio(// 调用 openAIAudio 方法生成音频
      {
        model: "tts-1",                  // 使用的模型
        input: "我正在试用 llm-request 中的 openAIAudio 方法生成音频",    // 输入文本
        voice: "alloy",                  // 指定的语音
      },
      "1"                                // 音频类型标识
    );

    if (audioUrl) {                      // 如果成功获取到音频 URL
      setAudioUrl(audioUrl);             // 更新音频 URL 状态

      // 音频 URL 转 File 对象
      const response = await fetch(audioUrl);         // 从 URL 获取音频数据
      const audioBlob = await response.blob();        // 将响应转换为 Blob 对象
      const file = new File([audioBlob], "audio.mp3", { type: "audio/mpeg" }); // 创
建 File 对象
```

```
        const formData = new FormData();                    // 创建表单数据对象
        formData.append("file", file);                      // 将音频文件添加到表单数据
        formData.append("model", "whisper-1");              // 添加模型信息
        setAudioText(await LLMRequestEntity.openAIAudio(formData, "2"));      // 调用
openAIAudio 进行语音转文本并更新状态
      } else {
        console.log("未获取有效音频");                      // 如果未获取有效音频，打印提示
      }
    };

    useEffect(() => {
      audioTest();                                          // 组件挂载后执行音频生成测试
    }, []);

    return (
      <>
        {audioUrl && <audio src={audioUrl} controls></audio>}        // 如果有音频 URL，则
显示音频控件
        {audioText}                                         // 显示音频转文本的结果
      </>
    );
  }

  export default App;        // 导出 App 组件
```

在上面的代码中，实现了一个音频的简单示例，它会在页面初始化时调用包请求以获取文本转音频的结果，并使用 audio 标签将结果回显到页面；接着，调用包请求将获得的音频翻译成英文，并在页面回显。

然后，在终端执行 npm run start 命令，启动一个本地服务，如图 2-37 所示。

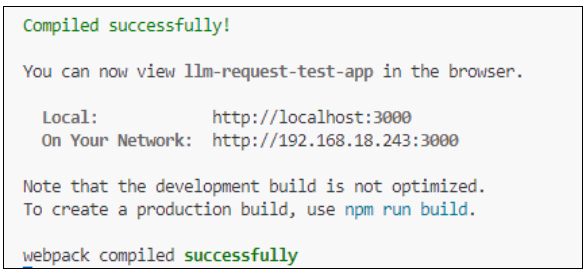

图 2-37　在终端执行 npm run start 命令后的结果

接着，单击终端中的链接跳转到浏览器，将会看到一个播放器和翻译的结果。单击播放器也可以听到输入文案的语音播报，效果如图 2-38 所示。

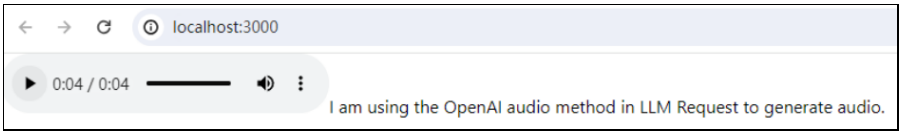

图 2-38　打开本地服务的效果

（2）测试聊天模块：

修改 App.js 代码如下：

```
import { useEffect, useState } from "react"; // 导入 React 的 useEffect 和 useState 钩子
import { default as LLMRequest } from "llm-request";     // 导入 llm-request 模块

function App() {
  const [text, setText] = useState("");              // 状态，用于存储普通聊天的返回文本
  const [streamText, setStreamText] = useState(""); // 状态，用于存储流式聊天的返回文本

  const chatTest = async () => {
    let finalAnswer = "";      // 用于累积流式聊天的最终答案
    const LLMRequestEntity = new LLMRequest("");     // 创建 LLMRequest 实例，替换为你的
API KEY

    // 调用 openAIChat 方法，获取聊天结果并将其转换为字符串
    setText(
      JSON.stringify(
        await LLMRequestEntity.openAIChat({
          model: "gpt-3.5-turbo",        // 使用的模型
          messages: [
            {
              role: "user",              // 消息角色为用户
              content: "你好",           // 用户发送的内容
            },
          ],
        })
      )
    );

    // 调用 openAIStreamChatCallback 进行流式聊天
    await LLMRequestEntity.openAIStreamChatCallback(
      {
        model: "gpt-3.5-turbo",          // 使用的模型
        messages: [
          {
            role: "user",                // 消息角色为用户
            content: "深圳有哪些火车站",    // 用户发送的内容
          },
        ],
        stream: true,                    // 指定为流式请求
      },
      (answer) => {                      // 回调函数，用于处理每次接收到的答案
        finalAnswer += answer;           // 累积接收到的答案
        setStreamText(finalAnswer);      // 更新流式聊天文本状态
      }
    );
```

```
  };

  useEffect(() => {
    chatTest();              // 组件挂载后执行聊天测试
  }, []);

  return (
    <>
      {text}                 // 显示普通聊天的返回文本
      <br></br>
      {streamText}           // 显示流式聊天的返回文本
    </>
  );
}

export default App;          // 导出 App 组件
```

再次打开浏览器页面，访问本地服务，可以看到一个聊天的答复，以及另一个关于深圳火车站的答复，带有着流响应的打字机输出效果，如图 2-39 所示。

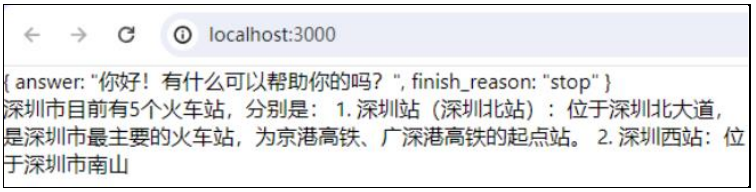

图 2-39　调用 llm-request 在 Web 端以打字机效果输出的示例

至此，请求库的 MVP（Minimum Viable Product，最小可行产品）版本已经开发完毕，接下来把版本发布到 npm。

6. 发布到 npm

在发布之前，先补齐一下 Readme 文档。Readme 是用户了解一个仓库大致功能的最直接方式，通常需要包含仓库的主要功能和 API。打开项目根目录的 README.md 文档，把之前的示例补全到 Readme 中。编写 Readme 注意使用 Markdown 语法。

Readme 编写完成后，就可以准备更新发布的版本号了，也就是 package.json 中的 version 字段。作为 llm-request 的第一版本，使用 v1.0.0 作为版本号。

值得一提的是，npm 包版本号通常遵循 Semantic Versioning（简称 SemVer）规范，以定义软件版本之间的语义关系，帮助用户更好地了解是否存在破坏性修改。SemVer 规范的基本格式是一个三位数的版本号：主版本号.次版本号.修订号，即 X.Y.Z。此外，还包括非正式版标识符——预发布版本标识符和构建元数据，这些版本号和标识符分别代表了软件迭代版本过程中的大致变化，具体含义如表 2-4 所示。

表 2-4　SemVerion 规范的版本号和标识符

版本号/标识符	说　　明
主版本号（major）	做了不兼容的 API 更改，或对现有功能进行了重大更改，以至于旧版本的应用程序可能无法正常运行时，应当增加主版本号（X）。主版本号的递增意味着存在不向后兼容的变化
次版本号（minor）	添加了新功能但保持与之前主版本号兼容性时，应增加次版本号（Y）。次版本号的增加意味着新增了特性，旧版本的应用程序可以在不做任何更改的情况下继续工作
修订号（patch）	如果只对现有功能的错误进行修正，且没有引入新功能或破坏现有的 API 时，应当增加修订号（Z）。修订号的递增代表向后兼容的 bug 修复
预发布版本（pre-release）	预发布版本通过在“Z”之后加上连字符（-）和一系列字母与数字标识符来表示，例如 1.2.3-alpha.1 或 2.0.0-beta.3。预发布版本通常用于软件仍在开发或测试阶段，尚未成为稳定版本的情况
构建元数据（build metadata）	构建元数据可以通过在版本号后面添加“+”后跟额外的字母数字信息来表示，如 1.0.0+build.1。这并不影响版本排序，主要用于标识同一版本的不同构建

在后续迭代中，版本号的更迭是一件很重要的事情。不规范的版本号更迭可能导致用户对版本功能产生误解，甚至可能引发用户代码中的隐性问题。

在终端切换到根目录后，执行下面的命令即可完成 npm 发布。如果用户没有 npm 账号，需要去 npm 官网注册一个。

```
npm login
npm publish
```

发布完成以后，可以在 npm 中查找到对应的包，如图 2-40 所示。

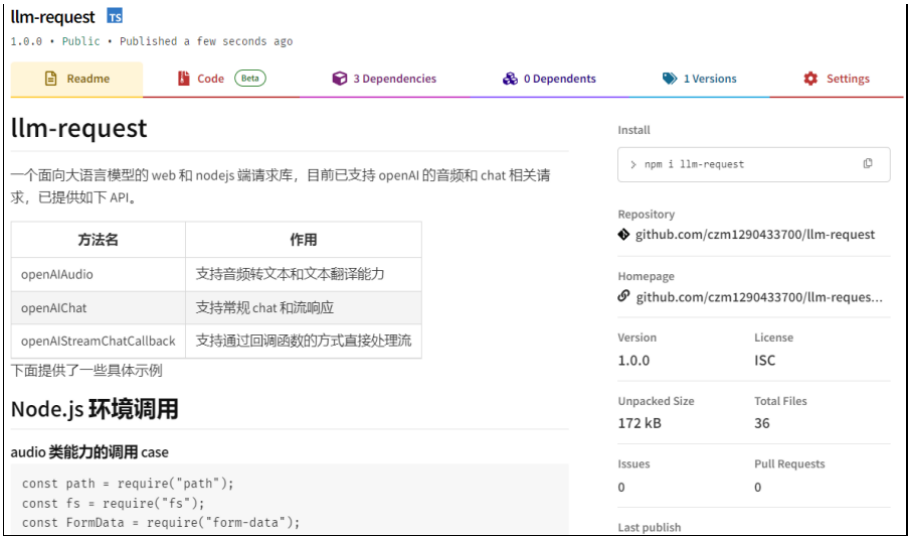

图 2-40　llm-request 在 npm 中的界面

后续调用请求库以快速实现生成式 AI 应用时，就不再需要使用软链的方式了，可以直接使用

npm install 命令完成对应包的安装。对于非 OpenAI 的端点，其他大语言模型的 API 也可以集成到这个请求库中，从而大幅提高类似应用的开发效率。

2.3.3　ChatGPT 国内可用免费 API 转发开源仓库：GPT-API-free

从 2024 年 6 月起，OpenAI 注册账号不再赠送 5 美元额度，新注册的账号可以直接使用 ChatGPT，但不具备调用 API 的功能。如果需要调用 API，则需要绑定海外信用卡完成充值。整个流程对于新人还是比较麻烦的。为了保证读者顺利完成本书内容的学习，这里推荐一个 ChatGPT 国内可用免费 API 转发开源仓库：GPT-API-free，如图 2-41 所示。通过它将可以在国内 IP 地址下免注册使用 OpenAI API。

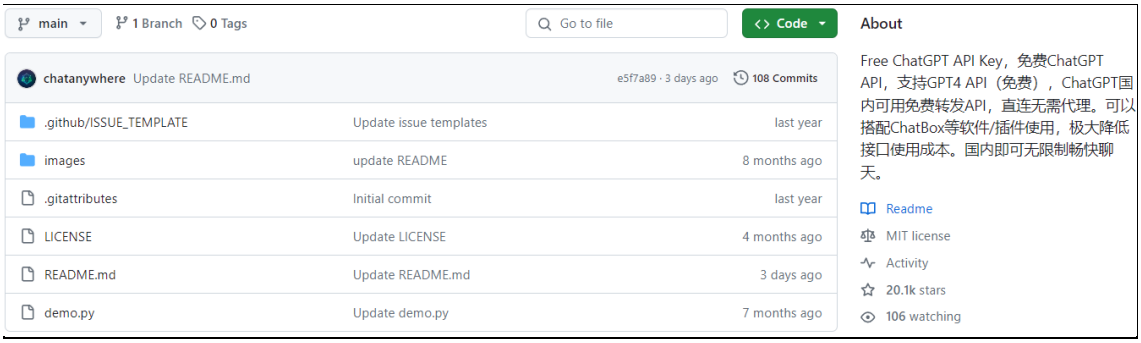

图 2-41　开源仓库 GPT-API-free

简单来说，这个仓库对 GPT-3.5、GPT-4、GPT-4o 等 OpenAI 模型完成了 API 的转发，部署在了国内外域名中，并暴露了 API_KEY 供开发者在学习、调研等非商用场景中使用。当然，对于商用等较大消耗的场景，它们也开放了付费 API_KEY，这里就不再详细介绍，感兴趣的读者可以自行了解。

对于以学习和调研为目的的场景，可以直接使用 GPT-API-free 提供的免费 API_KEY。在 GitHub 仓库的 Readme 中，单击申请内测免费 Key 按钮，会跳转到一个页面，其中包括了本次申请的 API_KEY，如图 2-42 所示。

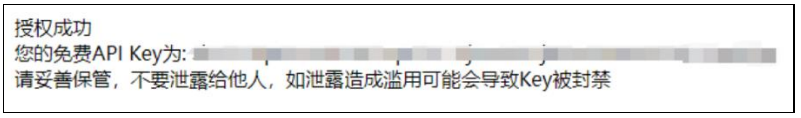

图 2-42　GPT-API-free 提供的免费 Key 页面

这个 API_KEY 不能直接用于 OpenAI 提供的端点服务，只能用于 GPT-API-free 部署的转发服务请求。目前 GPT-API-free 提供的转发服务域名如表 2-5 所示。

表 2-5　GPT-API-free 提供的转发服务域名

转发服务域名	适用场景
https://api.chatanywhere.tech	国内中转
https://api.chatanywhere.com.cn	国内中转
https://api.chatanywhere.cn	国外使用

下面以 OpenAI 请求库为例，介绍如何使用 GPT-API-free 完成转发服务的请求。OpenAI 请求库提供一个 baseURL 参数来扩展请求的 API，它将保持原有端点名，只替换服务的域名，以兼容社区中开放的类似 API 功能。通过 baseURL 参数将服务转发到 GPT-API-free 提供的服务中，创建任意命名的 JavaScript 脚本并写入如下代码：

```
import OpenAI from "openai";
const openai = new OpenAI({
  apiKey: "",                                  // 换成申请的 GPT-API-free
  baseURL: "https://api.chatanywhere.tech", // 也可以换成提供的其他转发域名服务
});
async function main() {
  const completion = await openai.chat.completions.create({
    model: "gpt-3.5-turbo", // 其他参数与请求常规 OpenAI 服务一致
    messages: [{ role: "user", content: "早上好呀" }],
    stream: true,
  });
  for await (const chunk of completion) {
    console.log(chunk.choices[0].delta.content);
  }
}
main();
```

执行这个脚本后，可以看到 OpenAI 服务的转发请求已成功完成，效果如图 2-43 所示。通过这种方式，不再需要科学上网或者注册海外信用卡等烦琐的流程，使用国内 IP 地址即可调用 OpenAI 服务进行学习和调研。

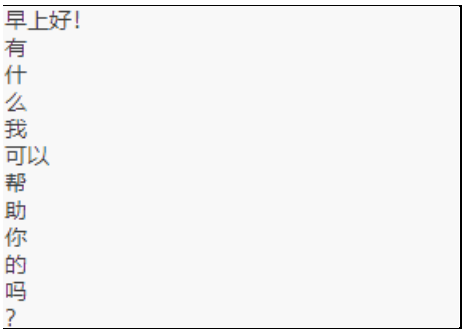

图 2-43　使用 GPT-API-free 在国内 IP 地址下转发调用 OpenAI 服务

2.4　本章小结

本章首先介绍了 OpenAI API 的相关知识，包括 API 的模型类别，以及在 Node.js 环境和 Web 环境下调用 API 实现文本转音频、音频翻译、聊天文本生成、流响应的打字机效果和 OpenAI 请求库。然后，封装并发布了一个兼容 Node.js 环境和 Web 环境的大语言模型 API 请求库，并在请求库编码中运用了前面介绍的不同环境的调用场景逻辑，从而加深读者对本章内容的理解；同时，运用了单元测试和软链功能测试等测试手段，深入实践了一个完整的请求库发布流程。

为保障大多数读者的学习体验，本章最后还介绍了一种可以在国内进行免费 OpenAI 服务转发的开源仓库 GPT-API-free。有了它，就不再需要对 OpenAI 服务进行科学上网或注册海外信用卡的操作，使用国内 IP 地址下即可调用 OpenAI 端点功能进行大语言模型的学习和调研。

通过本章的学习，读者应该能掌握以下 5 种开发技能：

（1）能在 Node.js 环境和 Web 环境下借助 OpenAI API 实现文本和音频的相互转换。

（2）能在 Node.js 环境和 Web 环境下借助 OpenAI API 实现聊天交互和打字机效果。

（3）了解 OpenAI 请求库的实现，能够使用请求库简化请求逻辑。

（4）具备封装高质量大语言模型 API 请求库的功能，能够综合考虑环境的兼容性、把控开发上线流程的功能质量测试，并清楚了解基础库发布的标准流程。

（5）基于 GPT-API-free 和请求库的 baseURL 参数，完成国内 IP 地址下的 OpenAI 服务调用。