第 3 章　动手写一个 MCP

前面两章探讨了 MCP 的核心概念与技术原理。本章通过实践，实现 MCP Server 的完整开发及上线发布。

MCP 目前支持 5 种主流编程语言的开发：Python、Node.js、Java、Kotlin 和 C#。为了使示例更具代表性且易于理解，本章将使用 Python 语言在 Windows 系统的计算机上进行演示，详细说明开发的完整流程和关键步骤。对于希望使用其他语言进行开发的读者，请参考官方文档 https://mcp-docs.cn/quickstart/server，其中包含完整的配置指南和最佳实践。

3.1　搭建 MCP 开发环境

MCP 开发规定使用 uv 进行虚拟环境创建和依赖管理。uv 是新一代 Python 包管理工具，它的设计目标是替代传统的 pip、venv 和 pip-tools 工具链。得益于其采用 Rust 语言开发，相比传统工具，uv 具有显著的性能优势。

uv 不仅能够更快速地安装和管理 Python 包，还提供了完整的虚拟环境管理功能。它采用并行下载和智能缓存机制，可以提升依赖安装的速度。同时，uv 还提供了更精确的依赖解析和版本控制能力，能够有效避免依赖冲突问题。

3.1.1　安装 uv

安装 uv 可以采用如下两种方式。

（1）使用 pip 安装。如果计算机上已经安装 pip，那么可以直接打开命令行窗口，使用以下命令安装 uv：

```
pip install uv
```

（2）使用 powershell 安装。如果计算机上没有安装 pip，那么可以打开 powershell 窗口，通过输入以下命令安装 uv：

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/
install.ps1 | iex"
```

3.1.2　uv 的基本用法

uv 工具的使用方法与传统 pip 的使用方法极为相似，然而它具备更简洁的语法和更高效的执行性能。在日常开发中，主要会用到以下几个基本命令。

（1）在依赖管理方面，uv 沿用了熟悉的包安装语法。例如，安装单个包可以使用以下命令：

```
uv pip install requests
```

（2）对于虚拟环境的管理，uv 提供了简化的命令：

```
uv venv myenv
```

（3）创建环境后，需要激活它才能使用。uv 提供的激活命令如下：

```
myenv\Scripts\activate
```

（4）当项目中有 requirements.txt 文件时，可以一次性安装所有依赖，命令如下：

```
uv pip install -r requirements.txt
```

（5）uv 也支持直接运行 Python 项目。当项目包含 pyproject.toml 配置文件时，只需要一个命令就能完成依赖安装和脚本的执行，命令如下：

```
uv run python script.py
```

上面的命令实际上整合了传统方式中的两个步骤：先安装依赖，再运行脚本。

MCP 项目之所以推荐使用 uv 进行环境管理，主要基于以下两方面的考虑。

（1）MCP 项目通常依赖多个 Python 模块，uv 通过 pyproject.toml 提供了更现代化的依赖管理方案，能够更好地处理复杂的依赖关系。

（2）uv 优秀的依赖解析机制可以有效避免传统 pip 遇到的依赖冲突问题。最重要的是，uv 显著提升的包管理速度对于 MCP 这类需要频繁管理依赖的项目来说，能够明显改善开发体验。

3.2　搭建一个 MCP Server

本节搭建一个用于查询天气服务的 MCP Server。在开始具体的搭建工作前，需要先新建一个文件夹并将其命名为 example，并在命令行中进入刚刚创建好的名为 example 文件夹目录。

3.2.1　项目初始化

现在开始配置 MCP 项目的开发环境。首先，在命令行中使用 uv 进行项目初始化：

```
uv init
```

接下来创建一个独立的虚拟环境来管理项目依赖：

```
uv venv
```

由于是在 Windows 系统中进行开发，因此使用以下命令进入虚拟环境：

```
.venv\Scripts\activate
```

3.2.2　环境配置

本示例通过 HTTP 请求来查询天气，因此需要安装几个核心依赖包，如图 3-1 所示。其中，依赖包 httpx 用于异步发起 HTTP 请求；依赖包 mcp 是使用 MCP 的必备包。

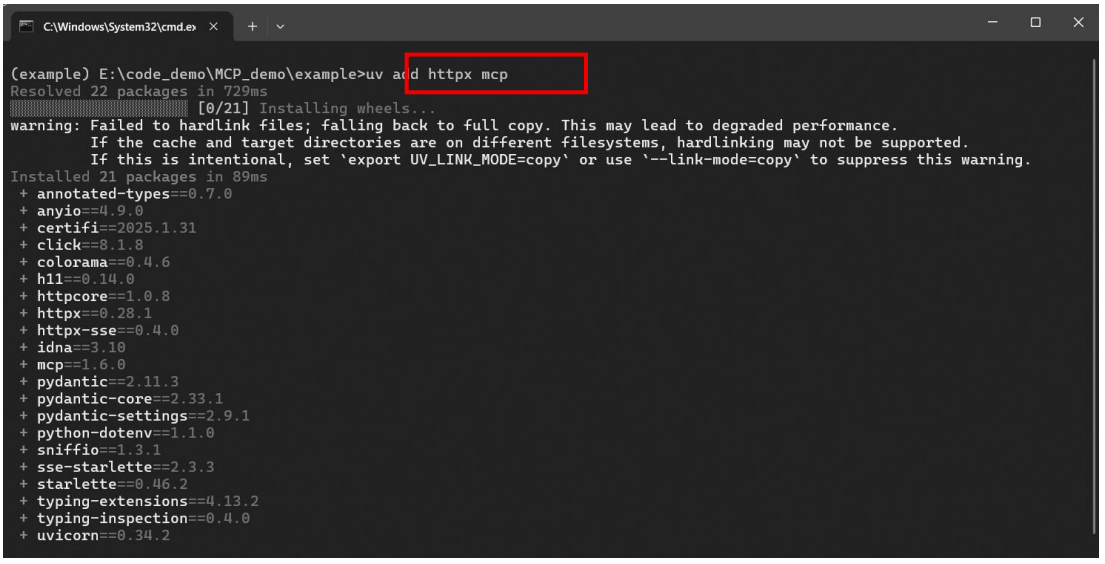

图 3-1　服务环境配置

可以看到，安装依赖包 httpx 的代码如下：

```
uv add httpx mcp
```

3.2.3　构建 MCP Server

本节创建一个 weather.py 文件，实现向 OpenWeather 请求天气的功能。具体步骤如下。

（1）导入依赖包。导入一些必需的依赖包，代码如下：

```
1   import json                                  # 处理 JSON 格式的数据
2   import httpx                                 # 发送异步的 HTTP 请求
3   from typing import Any                       # 导入类型提示工具
4   from mcp.server.fastmcp import FastMCP       # 导入 MCP 的 FastMCP 的类
5   mcp = FastMCP("WeatherServer")     # 创建一个名为 WeatherServer 的实例
```

（2）API 配置。进行 OpenWeather 天气查询网站的 API 的配置及通信，代码如下：

```
# OpenWeather API 配置
6 OPENWEATHER_API_BASE ="https://api.openweathermap.org/data/2.5/
  weather"
7   API_KEY = "YOUR_API_KEY"       # 替换为自己的 OpenWeather API Key
8   USER_AGENT = "weather-app/1.0"
```

（3）获取天气数据。定义一个异步函数，用于向 OpenWeather 网站请求城市的天气信息，并对可能出现的状态异常进行处理，代码如下：

```
# 定义一个查询天气的异步函数
9   async def fetch_weather(city:  str) -> dict[str, Any] | None:
    """
    从 OpenWeather API 获取天气信息。
    : param city:  城市名称（需使用英文，如 Wuhan）
    : return:  天气数据字典；若出错则返回包含 error 信息的字典
    """
# HTTP 请求参数设置
10      params = {
11          "q":  city,
12          "appid":  API_KEY,
13          "units":  "metric",
14          "lang":  "zh_cn"
    }
# HTTP 请求头设置
15      headers = {"User-Agent":  USER_AGENT}
# HTTP 客户端创建
16      async with httpx.AsyncClient() as client:
# 发送 GET 请求查询天气
17          try:
               # 发送 GET 请求查询天气

 18           response = await client.get(OPENWEATHER_API_BASE,
              params=params, headers=headers, timeout=30.0)
               # 检查响应状态码，如果不是 2xx 则抛出异常
19             response.raise_for_status()
               # 将响应的 JSON 解析为字典并返回
20             return response.json()
            # 处理 HTTP 状态错误(如 404 等)
21          except httpx.HTTPStatusError as e:
22              return {"error":  f"HTTP 错误:  {e.response.status_code}"}
            # 处理其他可能会出现的问题
23          except Exception as e:
24              return {"error":  f"请求失败:  {str(e)}"}
```

（4）数据格式化。将网站返回的复杂数据结构转换为用户优化型的文本输出，代码如下：

```
# 定义一个名为 format_weather 的函数，用于处理天气数据
25  def format_weather(data: dict[str, Any] | str) -> str:
    """
    将天气数据格式化为易读文本。
    : param data:  天气数据（可以是字典或 JSON 字符串）
    : return:  格式化后的天气信息字符串
```

```
    """
    # 如果传入的是字符串，那么先将其转换为字典
    if isinstance(data, str):
        try:
            data = json.loads(data)
        except Exception as e:
            return f"无法解析天气数据：{e}"
    # 如果数据中包含错误信息，那么直接返回错误提示
    if "error" in data:
        return f"⚠{data['error']}"
    # 提取数据时做容错处理，确保缺少数据也能正常工作
    city = data.get("name", "未知")
    country = data.get("sys", {}).get("country", "未知")
    temp = data.get("main", {}).get("temp", "N/A")
    humidity = data.get("main", {}).get("humidity", "N/A")
    wind_speed = data.get("wind", {}).get("speed", "N/A")
    # weather 可能为空列表，因此用 [0] 前先提供默认字典
    weather_list = data.get("weather", [{}])
    description = weather_list[0].get("description", "未知")
    # 使用 f-string 格式化字符串
    return(
        f"🌍 {city}, {country}\n"
        f"  温度：{temp}°C\n"
        f"💧 湿度：{humidity}%\n"
        f"  风速：{wind_speed} m/s\n"
        f"🌥 天气：{description}\n"
    )
```

（5）使用 MCP 工具函数。使用 MCP 装饰器封装一个工具，实现天气查询及结果格式化，代码如下：

```
# 使用 MCP 的装饰器将其标记为一个工具
@mcp.tool()
# 定义一个异步函数并声明其用途
async def query_weather(city: str) -> str:
    """
    输入指定城市的英文名称，返回今日天气查询结果。
    :param city: 城市名称(需使用英文)
    :return: 格式化后的天气信息
    """
    # 调用查询天气函数
    data = await fetch_weather(city)
    # 调用数据格式化函数
    return format_weather(data)
```

（6）主程序入口。定义程序入口函数，启动 MCP Server 并指定使用 Stdio 作为通信方式，代码如下：

```
# 程序入口
if __name__ == "__main__":
# 以 Stdio 方式运行 MCP Server
    mcp.run(transport='stdio')
```

3.2.4　在 Trae 中配置 MCP Server

3.2.3 节成功构建了一个能够查询天气的 MCP Server。接下来将该服务器配置到 IDE 中以便进行实际应用，我们以 Trae 作为示例进行演示，具体步骤如下。

（1）在 Trae 中配置 MCP Server。在 MCP 中的手动配置部分添加 MCP Server，然后就可以启动 MCP Server 了。这相当于执行命令 uv --directory E:\code_demo\MCP_demo\example run weather.py。注意，需要将示例中的文件夹路径替换为实际路径，具体操作如图 3-2 所示。相关代码如下：

```
{
  "mcpServers": {
    "example-server": {
      "command": "uv",
      "args": [
        "--directory",
        "E:\\code_demo\\MCP_demo\\example",
        "run",
        "weather.py"
      ]
    }
  }
}
```

图 3-2　JSON 配置

（2）测试 MCP Server。现在测试一下 MCP Server 的效果，在对话界面中输入“今

天武汉天气怎么样”，随后可以看到大语言模型调用了 MCP Server，然后成功执行并返回了结果，如图 3-3 所示。

图 3-3　MCP Server 测试效果

至此，查询天气的 MCP Server 就创建完成并在 Trae 上配置成功了。

3.3　MCP Server 的上线发布

MCP Server 测试成功后，便可将其上线发布到 PyPI（Python Package Index）平台，供别人配置使用。

PyPI 是 Python 官方的第三方包软件存储库，作为 Python 生态系统的重要组成部分，它承担着集中存储和分发 Python 包的关键角色。

接下来通过 PyPI 完成 MCP Server 的上线发布。

3.3.1　获取 PyPI 的 API token

登录 PyPI 官方网站，注册账号后在 Account settings 选项卡内创建一个 API token，以备上线发布时使用，如图 3-4 所示。

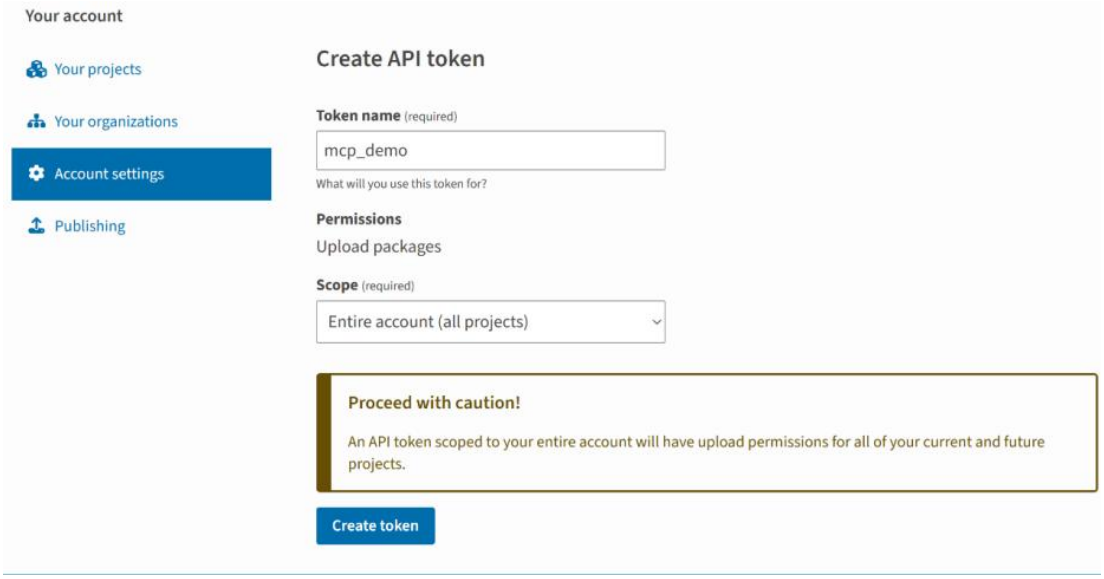

图 3-4　创建 API token

3.3.2　初始化包项目文件夹

首先需要创建一个项目文件夹将其初始化为包，并修改其项目文件。接下来进行逐步演示。

首先创建一个新的文件夹。文件夹命名为想要取的包的名字，注意独特性，不要与 PyPI 上已有的项目重复。

将这个项目文件夹初始化为包，在此目录下的命令行窗口中输入命令 uv init . --package，如图 3-5 所示。

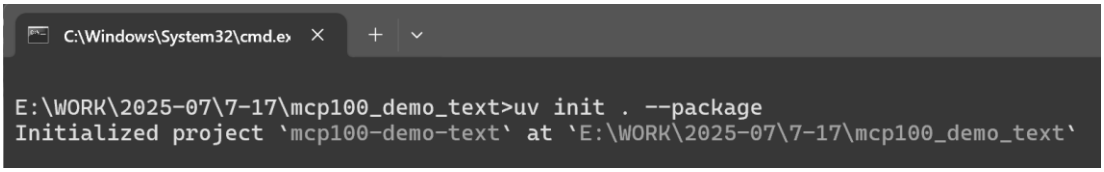

图 3-5　项目初始化

安装 MCP Server 所需的依赖，继续在命令行窗口中输入命令 uv add httpx mcp[cli]，

如图 3-6 所示。

```
E:\WORK\2025-07\7-17\mcp100_demo_text>uv add httpx mcp[cli]
Using CPython 3.13.3
Creating virtual environment at: .venv
Resolved 35 packages in 1.60s
      Built mcp100-demo-text @ file:///E:/WORK/2025-07/7-17/mcp100_demo_text
Prepared 3 packages in 3.24s
▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒ [0/35] Installing wheels...
warning: Failed to hardlink files; falling back to full copy. This may lead to degraded performance.
         If the cache and target directories are on different filesystems, hardlinking may not be supp
orted.
         If this is intentional, set `export UV_LINK_MODE=copy` or use `--link-mode=copy` to suppress
this warning.
Installed 35 packages in 694ms
 + annotated-types==0.7.0
 + anyio==4.9.0
 + attrs==25.3.0
 + certifi==2025.7.14
 + click==8.2.1
 + colorama==0.4.6
 + h11==0.16.0
 + httpcore==1.0.9
 + httpx==0.28.1
 + httpx-sse==0.4.1
 + idna==3.10
 + jsonschema==4.24.0
```

图 3-6　安装项目依赖

进入项目文件夹的 src 子文件，打开名为_init_.py 的 Python 文件。将 main 函数上粘贴在前面开发的 weather.py 代码中，如图 3-7 所示。

将入口函数的 main 函数的内容替换为 mcp.run(transport='stdio')，，并删除 if _name_ 这个判断语句，最终代码如图 3-8 所示。

```python
if __name__ == "__main__":
    # 以标准 I/O 方式运行 MCP 服务器
    mcp.run(transport='stdio')
def main() -> None:
    print("Hello from mcp100-demo-text!")
```

图 3-7　粘贴代码

```python
def main() -> None:
    mcp.run(transport='stdio')
```

图 3-8　修改后的代码

3.3.3　文件打包上传

在命令行窗口中输入命令 uv build，将项目进行打包，如图 3-9 所示。

```
E:\WORK\2025-07\7-17\mcp100_demo_text>uv build
Building source distribution...
Building wheel from source distribution...
Successfully built dist\mcp100_demo_text-0.1.0.tar.gz
Successfully built dist\mcp100_demo_text-0.1.0-py3-none-any.whl
```

图 3-9　项目打包

继续在命令行窗口中输入命令 uv publish --token <your PyPI Api token>，将项目上传

到 PyPI 上，注意将<your PyPI Api token>替换为在 3.3.1 节中获取的 PyPI 的 token，如图 3-10 所示。

```
E:\WORK\2025-07\7-17\mcp100_demo_text>uv publish --token pypi-AgEIcHlwaS5vcmcCJDljMTM3OTE5LWQ5NjctNDkw
Ni1iNmM1LTMwMTBmODRkMzlkZQACK[illegible]QAABiB8fSCz8BzGLv
oyGj_9CR0DyP8Npd5kCZxJq1fyaX5aew
Publishing 2 files https://upload.pypi.org/legacy/
Uploading mcp100_demo_text-0.1.0-py3-none-any.whl (2.7KiB)
Uploading mcp100_demo_text-0.1.0.tar.gz (17.8KiB)
```

图 3-10　项目上传

登录 PyPI 网站，可以看到刚刚发布的项目，如图 3-11 所示。

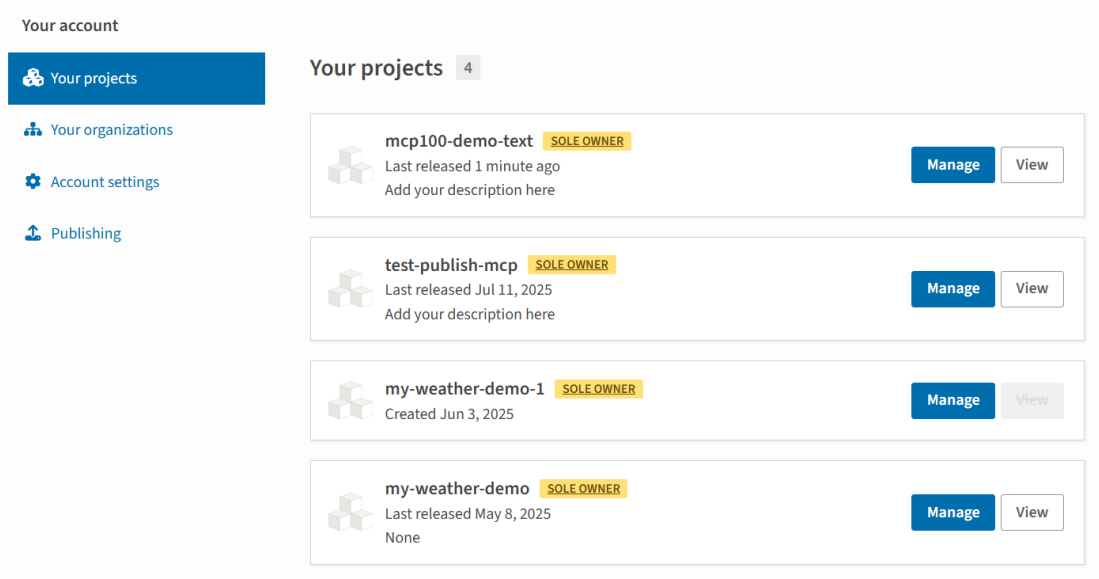

图 3-11　PyPI 项目列表

3.3.4　包测试

继续在 Trae 中测试 MCP，在 JSON 配置文件中输入以下命令，其中，args 中的内容即为包名，配置成功后旁边会出现对勾符号，如图 3-12 所示。

```
{
   "mcpServers": {
     "weather_demo": {
      "command": "uvx",
      "args": [
        "mcp100-demo-text"
      ]
     }
   }
}
```

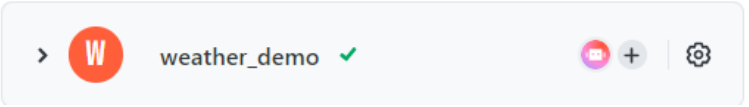

图 3-12　MCP 配置成功

输入“今天哈尔滨天气怎么样”，可以看到调用了刚刚加入的 MCP 并返回了结果，此项目开发完成，如图 3-13 所示。

图 3-13　MCP 测试