mcp和function call

Function Call 与 MCP：概念解析与应用指南

01. 如何限制大模型的输出格式

使用提示工程（Prompt Engineering）

最简单直接的方法：在提示词（prompt）中明确要求输出格式（如 JSON、YAML、XML 等），并提供示例。

例如："请以以下 JSON 格式返回：…", 并附上具体字段定义和样例。在提示词中加入示例结构或"输出格式规范"之类的段落，也能有效提高格式契合度。

优点：实现简单，无需额外训练或修改模型本身。

缺点：结构严格的格式（尤其 JSON）仍易出错，难以确保 100% 达标。

使用 Function Call 的方式

这种方式的核心思想是：让大模型不直接输出自然语言，而是通过"调用一个函数"返回结构化参数，这样你就可以用程序严格解析它的返回值，确保输出符合要求。

1. 基本原理

大模型在生成内容时，本质是"预测文本"。如果你只给它自然语言指令，它的输出可能会跑偏（比如漏字段、多字段、错格式）。

Function Call 是在模型 API 层面增加的一种"协议"：

- 你定义一个函数（其实是一个结构化描述，告诉模型这个函数的参数和数据类型）
- 模型会输出一个 JSON 参数对象来"调用"这个函数
- 你拿到这个参数，就能保证它符合你定义的结构

举个简单例子：

```
{
  "functions": [
    {
      "name": "create_task",
      "description": "创建一个任务",
      "parameters": {
        "type": "object",
        "properties": {
```

```
            "title": {"type": "string", "description": "任务标题"},
            "priority": {"type": "integer", "description": "任务优先级，1为最高"},
            "due_date": {"type": "string", "format": "date"}
        },
        "required": ["title", "priority"]
      }
    }
  ]
}
```

这样模型被约束只能"生成"出：

```
{
  "title": "写周报",
  "priority": 1,
  "due_date": "2025-08-15"
}
```

而不会生成你没定义的字段，也不会乱写格式（比如用中文写"优先级一"）。

2. 流程

用 function call 限制输出的完整流程大致是这样：

2.1 定义函数 schema
用 JSON Schema（parameters）描述函数的参数类型、必填字段、值的格式。

2.2 调用模型并指定可用函数
在 API 请求里带上 functions（有的 API 叫 tools）。

2.3 模型决定调用哪个函数

- 如果你要求它必须调用某个函数，可以设置 `function_call={"name": "xxx"}`
- 否则它也可能选择不调用

2.4 返回结果是结构化 JSON
模型会返回一个 `function_call.arguments` 字段，内容就是 JSON 字符串（符合你 schema）。

2.5 解析参数并执行逻辑
你在代码中 `json.loads()` 后就能直接用。

3. 为什么能限制输出格式

因为 function call 的参数必须符合你定义的 schema：

- **类型限制**：string / integer / boolean / array / object
- **必填字段**：required

- **值格式**：format（如 date, email, uri）
- **枚举值**：enum（固定值集合）
- **嵌套结构**：字段可以是对象或数组

这就像给模型套了一个"JSON护栏"，不让它乱跑。即便它生成错了，API 层面会尝试自动修正，或者直接抛错，让你捕获处理。

4. 注意事项

4.1 描述要足够清晰

- 字段含义、取值范围写清楚
- 不要模糊不清，否则模型会生成奇怪的值

4.2 字段名稳定性

- 尽量用英文小写+下划线
- 不要用中文字段名，否则模型容易生成错误

4.3 嵌套结构越复杂，模型越容易出错

- 尽量分层调用多个函数，而不是一次返回超级复杂的 JSON

4.4 有些 API 会自动做 JSON 修正

- 比如 OpenAI、DashScope 在 function call 里会帮你补全缺失字段、修正数据类型
- 但如果修不回来，会返回报错，你要在代码里做兜底

5. 一个完整示例（DashScope）

DashScope 支持 function call，比如：

```
import dashscope
from dashscope import Generation

dashscope.api_key = "xxx"

functions = [
    {
        "name": "create_task",
        "description": "创建一个任务",
        "parameters": {
            "type": "object",
            "properties": {
                "title": {"type": "string"},
                "priority": {"type": "integer"},
                "due_date": {"type": "string", "format": "date"}
```

```
            },
            "required": ["title", "priority"]
        }
    }
]

response = Generation.call(
    model="qwen-plus",
    prompt="帮我生成一个重要的任务，明天截止",
    functions=functions,
    function_call={"name": "create_task"}
)

print(response.output.choices[0].message["function_call"]["arguments"])
```

好处：

- 保证返回的 arguments 是合法 JSON
- 结构固定，字段不会乱
- 方便直接 `json.loads()` 转成 Python dict 用

使用 MCP 的流程

MCP（Model Call Protocol）是一种面向生产级 AI 应用的标准化交互协议。它不仅利用大模型的 function call 能力，还引入了**版本控制、元信息交换、状态管理、跨模型兼容性设计**等机制，使得不同模型、不同服务之间可以统一调度与解析。

1. 基本原理

普通 function call 只是让模型返回 JSON 参数，而 MCP 的目标是建立一种类似“远程过程调用（RPC）”的可靠通信协议：

- 定义清晰的服务接口（Service Interface）
- 支持多轮调用中的上下文延续（session_id / trace_id）
- 内置错误码与重试策略
- 兼容多种 backend（OpenAI / Qwen / Claude / 自研模型）

因此，MCP = Function Call + 协议头 + 标准化 payload + 错误语义 + 可观测性

2. MCP 流程

2.1 定义 MCP Service Schema

不同于简单 functions 数组，MCP 使用带命名空间和版本的服务描述：

```
{
  "service": "task/v1/create_task",
```

```
  "description": "创建一个新的任务条目",
  "parameters": {
    "type": "object",
    "properties": {
      "title": { "type": "string" },
      "priority": { "type": "integer", "enum": [1, 2, 3] },
      "due_date": { "type": "string", "format": "date" }
    },
    "required": ["title"]
  }
}
```

注意：这里用了 /v1/ 版本号，便于未来升级。

2.2 构造 MCP 请求包

请求包含 protocol metadata 和实际调用内容：

```
{
  "protocol": "MCP/1.0",
  "request_id": "req_abc123",
  "session_id": "sess_xyz789",
  "timestamp": "2025-04-05T10:00:00Z",
  "model_hint": "qwen-plus",
  "services": [
    {
      "name": "create_task",
      "service": "task/v1/create_task",
      "parameters": { ... }
    }
  ],
  "prompt": "帮我创建一个高优先级任务：写周报，明天截止"
}
```

2.3 模型网关路由并执行

后端网关识别 protocol: MCP/1.0，将 services 映射为对应平台的 function call 输入（例如转成 DashScope 或 OpenAI 格式），发送给模型。

2.4 返回标准化 MCP 响应

无论底层模型是什么，统一返回 MCP 格式的输出：

```
{
  "protocol": "MCP/1.0",
  "request_id": "req_abc123",
  "status": "success",
  "service_call": {
```

```
    "name": "create_task",
    "service": "task/v1/create_task",
    "arguments": {
      "title": "写周报",
      "priority": 1,
      "due_date": "2025-08-15"
    }
  },
  "timestamp": "2025-04-05T10:00:05Z"
}
```

如果失败，则返回标准 error code：

```
{
  "protocol": "MCP/1.0",
  "request_id": "req_abc123",
  "status": "error",
  "error": {
    "code": "INVALID_ARGUMENT",
    "message": "missing required field: title",
    "details": { ... }
  }
}
```

2.5 客户端解析并处理结果

客户端无需关心模型类型，只需按 MCP 协议解析即可：

```
if response["status"] == "success":
    args = response["service_call"]["arguments"]
    # 执行业务逻辑
else:
    handle_error(response["error"])
```

3. 为什么能更好地限制输出格式

相比原始 function call，MCP 提供了更强的约束力：

协议一致性：所有服务遵循同一套 envelope 结构
版本隔离：v1 和 v2 接口互不干扰
错误标准化：统一 error codes（如 PARSE_FAILED, MISSING_REQUIRED_FIELD）
可观测性增强：内置 request_id、session_id、timestamp，适合日志追踪
相当于从“散装 JSON 调用”进化到“微服务风格的 AI 调用”。

4. 注意事项

4.1 必须建立中央 service registry

维护所有可用 MCP 服务的注册表，确保命名唯一、文档清晰。

4.2 避免过度抽象

不要把所有功能塞进一个 mega-service，建议按领域拆分（task/, meeting/, notification/）。

4.3 网关层要做 schema 兼容转换

比如把 MCP 的 service 映射到 OpenAI 的 function.name，同时保留元信息。

4.4 支持降级模式

当模型不支持 function call 时，MCP 可切换为“schema-guided generation + 后验校验”。

5. 一个完整示例（MCP over DashScope）

```
import dashscope
import json
import uuid
from datetime import datetime

# MCP 网关封装函数
def mcp_call(prompt: str, service_schema: dict):
    request_id = str(uuid.uuid4())
    timestamp = datetime.utcnow().isoformat() + "Z"

    # 构造 MCP 请求包
    mcp_request = {
        "protocol": "MCP/1.0",
        "request_id": request_id,
        "timestamp": timestamp,
        "services": [service_schema],
        "prompt": prompt
    }

    # 转换为 DashScope 兼容格式
    ds_functions = [ {k: v for k, v in service_schema.items() if k != "service"} ]

    try:
        resp = dashscope.Generation.call(
            model="qwen-plus",
            prompt=prompt,
            functions=ds_functions,
            function_call={"name": service_schema["name"]}
        )

        if resp.status_code == 200:
            fc = resp.output.choices[0].message.get("function_call")
            if fc:
```

```
                arguments = json.loads(fc["arguments"])
                return {
                    "protocol": "MCP/1.0",
                    "request_id": request_id,
                    "status": "success",
                    "service_call": {
                        "name": fc["name"],
                        "service": service_schema["service"],
                        "arguments": arguments
                    },
                    "timestamp": datetime.utcnow().isoformat() + "Z"
                }
            else:
                raise ValueError("No function call returned")
        else:
            return make_mcp_error(request_id, "API_ERROR", str(resp))

    except Exception as e:
        return make_mcp_error(request_id, "INTERNAL_ERROR", str(e))

def make_mcp_error(req_id, code, msg):
    return {
        "protocol": "MCP/1.0",
        "request_id": req_id,
        "status": "error",
        "error": {
            "code": code,
            "message": msg
        },
        "timestamp": datetime.utcnow().isoformat() + "Z"
    }

# 使用示例
service = {
    "name": "create_task",
    "service": "task/v1/create_task",
    "description": "创建一个任务",
    "parameters": {
        "type": "object",
        "properties": {
            "title": {"type": "string"},
            "priority": {"type": "integer"},
            "due_date": {"type": "string", "format": "date"}
```

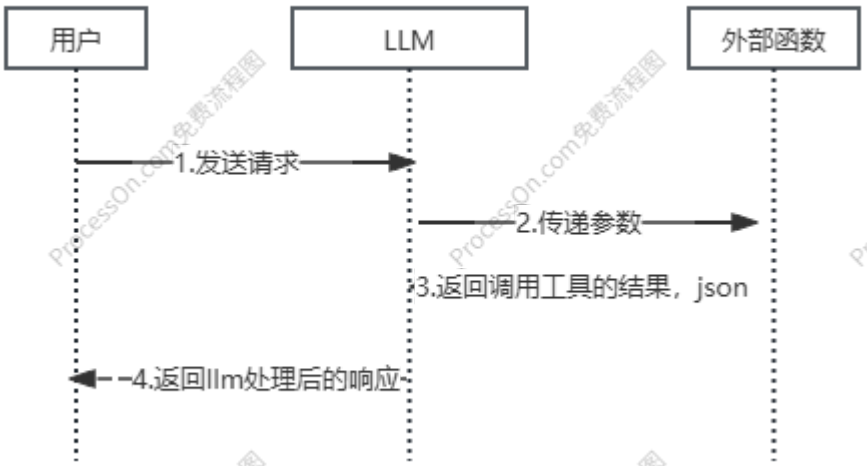
```
        },
        "required": ["title"]
    }
}

result = mcp_call("请创建一个紧急任务：修复登录bug，本周五前完成", service)
print(json.dumps(result, indent=2, ensure_ascii=False))
```

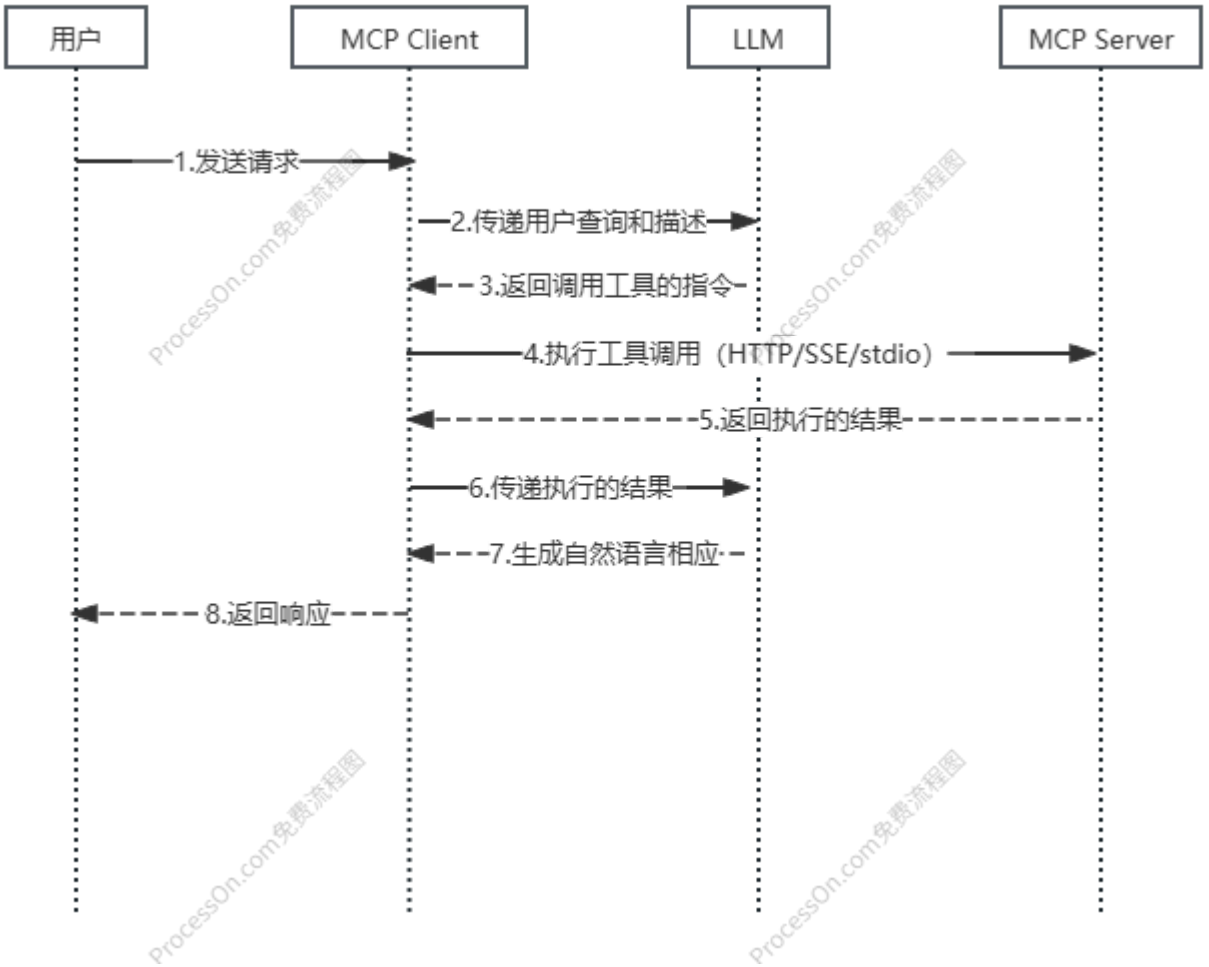
好处总结：

- 上层应用只依赖 MCP 协议，与底层模型解耦
- 支持多模型后端统一接入
- 易于监控、审计、调试
- 可作为企业级 AI 中台的核心通信协议

02. MCP 和 Function Call 的区别

- MCP 是通用协议层的标准化约定（更偏抽象的通用），Function Call 是某大模型厂商特定的实现方式和特性（更偏具体实现）。
- 函数调用 (Function Calling)：2023年，OpenAI等机构引入了函数调用机制，这是一个巨大的进步。开发者可以预先定义结构化的函数（如 `getWeather(location)`），模型在需要时，会输出一个标准化的JSON对象来"请求"调用这个函数，而不是模糊的自然语言指令。





概念对比表

维度	MCP协议（行业通用）	Function Call（厂商私有）
标准化程度	基于行业通用标准（如HTTP、gRPC）	厂商自定义实现（如Lightning、Private API）
跨平台兼容性	支持多模型混用（如混合调用不同厂商模型）	仅限单一厂商模型生态
开发成本	初始配置复杂，长期维护成本低	快速上线，但生态锁定风险高

维度	MCP协议（行业通用）	Function Call（厂商私有）
扩展性	容易集成新工具/模型	扩展依赖厂商支持
安全性	标准协议更成熟，安全机制完善	可能存在私有协议安全漏洞
性能表现	优化空间大，可定制通信层	性能高度依赖厂商实现
适用场景	多云/混合部署、长期项目	快速验证POC、单一厂商深度绑定场景

从刚刚代码的层面再解释一下区别

本节通过五个关键流程阶段，逐一对比 **普通 Function Call** 和 **MCP（Model Call Protocol）** 在设计目标、结构规范和工程实践上的本质差异。

1. 基本原理

维度	Function Call	MCP
核心目的	让模型返回结构化参数，避免自由文本输出	构建标准化 AI 调用协议，实现跨模型、跨服务的统一通信
抽象层级	单次 API 功能调用	类似 RPC 的远程过程调用体系
是否依赖平台	是（如 OpenAI / DashScope 特定语法）	否（通过网关抽象底层差异）
典型场景	简单工具调用（如创建任务、查天气）	企业级 AI 中台、多 Agent 协作系统

> ✅ **关键区别**：Function Call 是“功能”，MCP 是“协议”。

---

2. 流程对比

2.1 定义函数 schema

```
// Function Call - 简单函数描述
{
  "name": "create_task",
  "description": "创建一个任务",
  "parameters": {
    "type": "object",
    "properties": {
      "title": { "type": "string" },
      "priority": { "type": "integer" }
    },
    "required": ["title"]
```

```
  }
}
```

```
// MCP - 带命名空间与版本的服务接口
{
  "service": "task/v1/create_task",          // ✅ 新增：全局唯一标识 + 版本控制
  "name": "create_task",                     // 用于兼容底层模型
  "description": "创建一个新的任务条目",
  "protocol": "MCP/1.0",                     // ✅ 新增：协议版本声明
  "parameters": { ... }                      // 同上，但更强调可文档化
}
```

2.2发起请求

```
# Function Call - 直接调用特定平台 API
response = Generation.call(
    model="qwen-plus",
    prompt="帮我生成一个任务",
    functions=[func_schema],
    function_call={"name": "create_task"}
)
```

```
# MCP - 封装后的通用请求包（与模型无关）
mcp_request = {
    "protocol": "MCP/1.0",
    "request_id": "req_abc123",
    "session_id": "sess_xyz789",           # ✅ 新增：会话追踪
    "timestamp": "2025-04-05T10:00:00Z",   # ✅ 新增：时间戳
    "model_hint": "qwen-plus",             # ✅ 可选提示模型类型
    "services": [mcp_service_schema],
    "prompt": "帮我生成一个任务"
}

# 由 MCP 网关转发至具体模型（如 Qwen / GPT）
response = mcp_gateway.send(mcp_request)
```

获得当前天气和当前时间的 Function Call 具体实现

```
import dashscope
from dashscope import Generation
from datetime import datetime
import random
import json
import requests

dashscope.api_key = "sk-xxxx"
```

```
# 定义工具列表，模型在选择使用哪个工具时会参考工具的name和description
tools = [
    # 工具1 获取当前时刻的时间
    {
        "type": "function",
        "function": {
            "name": "get_current_time",
            "description": "当你想知道现在的时间时非常有用。",
            "parameters": {}  # 因为获取当前时间无需输入参数，因此parameters为空字典
        }
    },

    # 工具2 获取指定城市的天气
    {
        "type": "function",
        "function": {
            "name": "get_current_weather",
            "description": "当你想查询指定城市的天气时非常有用。",
            "parameters": {  # 查询天气时需要提供位置，因此参数设置为location
                "type": "object",
                "properties": {
                    "location": {
                        "type": "string",
                        "description": "城市或县区，比如北京市、杭州市、余杭区等。"
                    }
                }
            },
            "required": [
                "location"
            ]
        }
    }
]

# 基于外部网站的天气查询工具。返回结果示例：{"location": "\u5317\u4eac\u5e02",
"weather": "clear sky", "temperature": 17.94}
def get_current_weather(location):
    api_key = "xxxx"  # 替换为你自己的OpenWeatherMap API密钥，用我的也无所谓啦，反正免
费。
    url = f"http://api.openweathermap.org/data/2.5/weather?q={location}&appid=
{api_key}&units=metric"
    response = requests.get(url)
```

```
    if response.status_code == 200:
        data = response.json()
        weather = data["weather"][0]["description"]
        temp = data["main"]["temp"]
        return json.dumps({"location": location, "weather": weather, "temperature":
temp})
    else:
        return json.dumps({"location": location, "error": "Unable to fetch weather
data"})

# 查询当前时间的工具。返回结果示例："当前时间：2024-04-15 17:15:18。"
def get_current_time():
    # 获取当前日期和时间
    current_datetime = datetime.now()
    # 格式化当前日期和时间
    formatted_time = current_datetime.strftime('%Y-%m-%d %H:%M:%S')
    # 返回格式化后的当前时间
    return f"当前时间：{formatted_time}。"

# 封装模型响应函数
def get_response(messages):
    response = Generation.call(
        model='qwen-plus',
        messages=messages,
        tools=tools,
        seed=random.randint(1, 10000),  # 设置随机数种子seed，如果没有设置，则随机数种
子默认为1234
        result_format='message'  # 将输出设置为message形式
    )
    return response

def call_with_messages():
    print('\n')
    messages = [
        {
            "content": input('请输入：'),  # 提问示例："现在几点了？" "一个小时后几点"
"北京天气如何？"
            "role": "user"
        }
    ]

    # 模型的第一轮调用
    first_response = get_response(messages)
```

```
    assistant_output = first_response.output.choices[0].message
    print(f"\n大模型第一轮输出信息：{first_response}\n")
    messages.append(assistant_output)

    if 'tool_calls' not in assistant_output:  # 如果模型判断无需调用工具，则将
assistant的回复直接打印出来，无需进行模型的第二轮调用
        print(f"最终答案：{assistant_output.content}")
        return

    # 如果模型选择的工具是get_current_weather
    elif assistant_output.tool_calls[0]['function']['name'] == 'get_current_weather':
        tool_info = {"name": "get_current_weather", "role":"tool"}
        location = json.loads(assistant_output.tool_calls[0]['function']['arguments'])
['location']
        tool_info['content'] = get_current_weather(location)

    # 如果模型选择的工具是get_current_time
    elif assistant_output.tool_calls[0]['function']['name'] == 'get_current_time':
        tool_info = {"name": "get_current_time", "role":"tool"}
        tool_info['content'] = get_current_time()

    print(f"工具输出信息：{tool_info['content']}\n")
    messages.append(tool_info)

    # 模型的第二轮调用，对工具的输出进行总结
    second_response = get_response(messages)
    print(f"大模型第二轮输出信息：{second_response}\n")
    print(f"最终答案：{second_response.output.choices[0].message['content']}")

if __name__ == '__main__':
    call_with_messages()
```