MCP

MCP — 模型上下文协议

概述

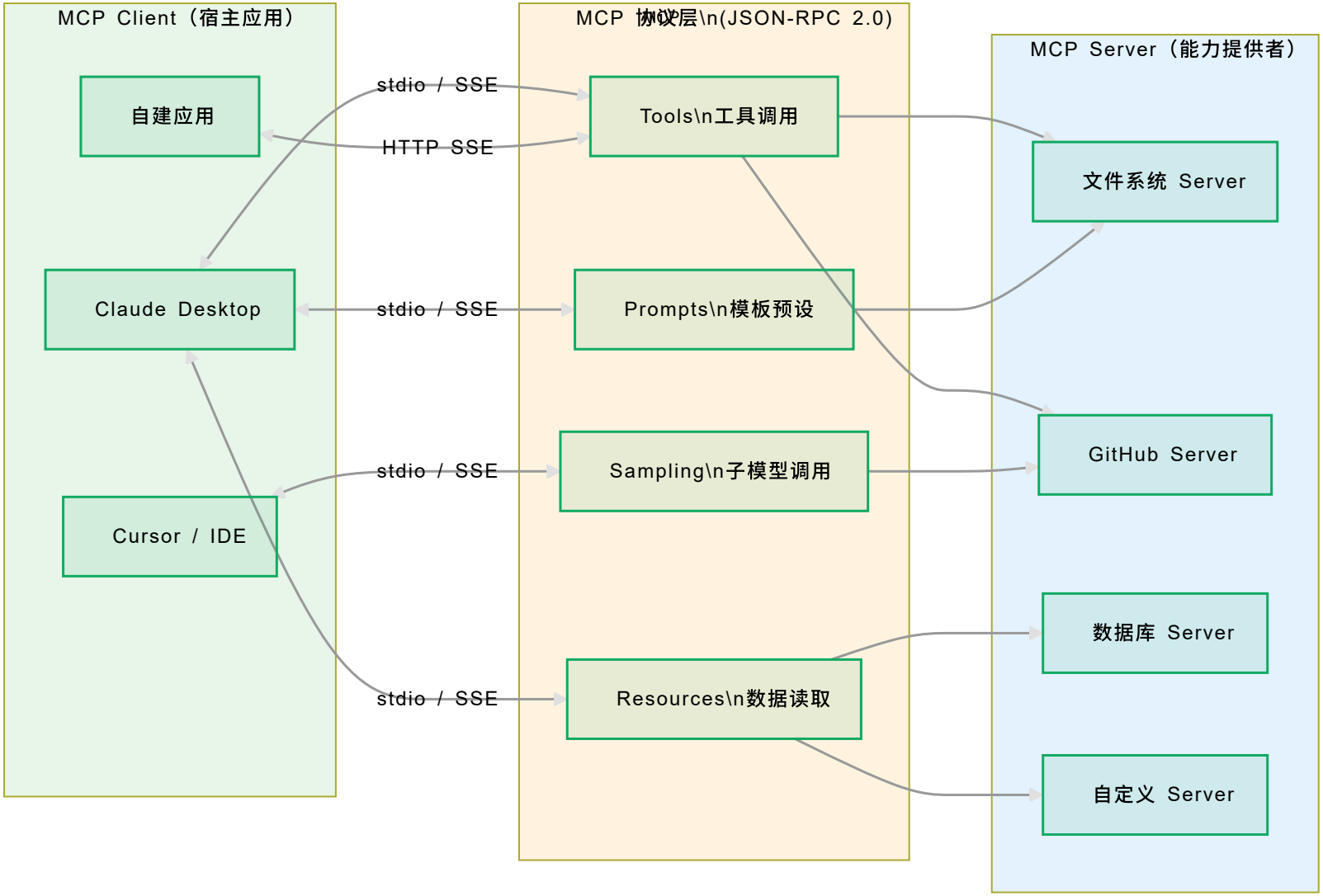
Model Context Protocol (MCP) 是由 Anthropic 开源的开放协议，为 AI 模型与外部数据源、工具之间提供标准化的通信接口。它让大语言模型能够安全、可控地访问文件系统、API、数据库等外部资源，是构建 AI Agent 基础设施的关键组件。

为什么需要 MCP?

在 MCP 出现之前，每个 AI 应用集成新数据源都需要写自定义代码——GitHub 一个接口、Slack 一个接口、公司内部 API 又一个接口。MCP 用一套统一协议解决了"重复造轮子"的问题：定义一次 Server，所有兼容 MCP 的 Client 都能使用。

核心架构

MCP 采用 **Client-Server** 架构，通过 JSON-RPC 2.0 进行通信：



三个核心角色：

角色	职责	类比
Client	发起请求，连接 Server，将能力提供给 AI 模型	"插件管理器"

角色	职责	类比
Server	暴露 Tools、Resources、Prompts 等能力	"插件实现"
Protocol	定义 JSON-RPC 通信格式和消息类型	"USB 接口标准"

关键理解

MCP **不是**另一个 LLM 框架或 SDK，它是一个**通信协议**。类似 HTTP 之于 Web 开发——你不需要"学 HTTP 框架"，只需要知道如何发起请求和响应。MCP 同理，Server 和 Client 各自用 JSON-RPC 说话。

三种核心能力

1. Tools — 让 AI 执行操作

Tools 允许 AI 模型调用函数并获取结果，是 MCP 最常用的能力类型。

```go
// Go 实现：注册一个「查询天气」Tool
server.AddTool(mcp.Tool{Name: "get_weather", Description: "查询指定地点的天气"},
    func(ctx context.Context, req mcp.CallToolRequest) (*mcp.CallToolResult, error) {
        city := req.Arguments["city"].(string)
```

```
        weather := queryWeather(city) // 调用实际业务逻辑
        return mcp.SuccessResult(mcp.TextContent(weather)), nil
    })
```

客户端调用流程：

```
// Client 侧发起 Tool 调用
result, err := client.CallTool(ctx, "get_weather", map[string]any{
    "city": "Beijing",
})
```

典型场景： 搜索代码仓库、创建 GitHub Issue、发送 Slack 消息、执行数据库查询。

2. Resources — 让 AI 读取数据

Resources 提供结构化的只读数据访问，AI 可以根据 URI 按需读取：

```
// 注册一个 Resource Template（支持参数化 URI）
server.AddResourceTemplate(
    mcp.ResourceTemplate{URI: "repo://{owner}/{repo}/file/{path}", Name: "Repository File"},
    func(ctx context.Context, req mcp.ReadResourceRequest) (*mcp.ReadResourceResult, error) {
        // 根据 URI 参数读取文件内容
```

```
        content := readFile(req.Arguments["path"].(string))
        return mcp.SuccessResult(mcp.TextContent(content)), nil
    },
)
```

AI 可以通过 `repo://anthropic/claude-docs/file/README.md` 这样的 URI 读取任意仓库文件。

3. Prompts — 可复用的对话模板

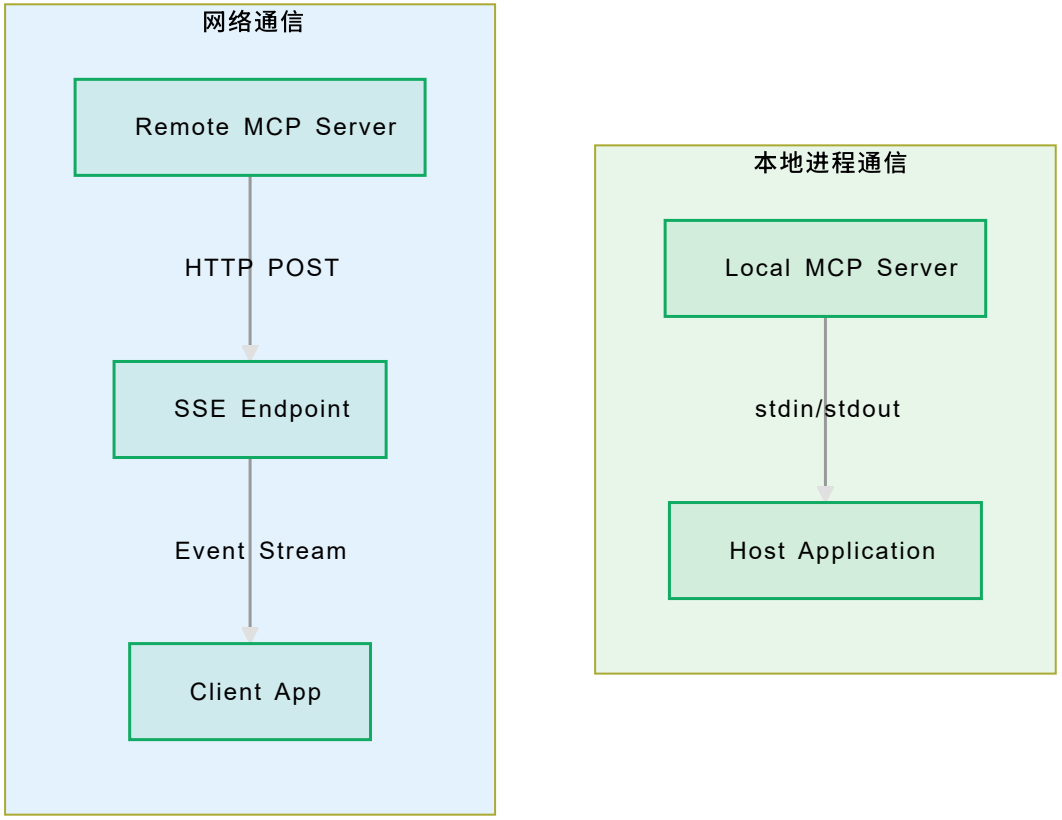
Prompts 允许 Server 预定义结构化提示词模板，Client 加载后自动注入上下文：

```
server.AddPrompt(mcp.Prompt{Name: "code_review", Description: "代码审查模板"},
    func(ctx context.Context, req mcp.GetPromptRequest) (*mcp.GetPromptResult, error) {
        diff := req.Arguments["diff"].(string)
        return &mcp.GetPromptResult{
            Messages: []mcp.PromptMessage{{
                Role: mcp.RoleUser,
                Content: mcp.TextContent(fmt.Sprintf(`请审查以下代码变更：

%s

重点关注：安全性、性能、可读性。`, diff)),
            }},
```

```
    }, nil
})
```


传输层：连接 Client 与 Server

MCP 支持两种传输方式，覆盖从本地到云端的完整场景：



传输方式	适用场景	特点
stdio	本地运行的 Server	零配置，通过标准输入输出通信，适合命令行工具和本地集成
SSE (Server-Sent Events)	远程 Server	通过 HTTP 端口暴露，适合云端服务和多客户端共享

底层原理

无论哪种传输方式，上层使用的都是相同的 **JSON-RPC 2.0** 消息格式。传输层仅负责"把消息送到对方手里"，不关心内容语义。

JSON-RPC 消息格式

MCP 基于 JSON-RPC 2.0，消息结构简洁直观：

初始化握手（Initialize）：

```
// Client → Server
{"jsonrpc":"2.0","id":1,"method":"initialize",
 "params":{"protocolVersion":"2024-11-05","capabilities":{"tools":{},"resources":{}},"clientInfo":
```

```
{"name":"my-client","version":"1.0.0"}}}

// Server → Client 响应
{"jsonrpc":"2.0","id":1,"result":{"protocolVersion":"2024-11-05","capabilities":{"tools":
{},"resources":{}},"serverInfo":{"name":"weather-server","version":"1.0.0"}}}
```

列出可用 Tools：

```
// Client → Server
{"jsonrpc":"2.0","method":"tools/list","id":2}

// Server → Client
{"jsonrpc":"2.0","result":{"tools":[{"name":"get_weather","description":"查询天气","inputSchema":
{"type":"object","properties":{"city":{"type":"string"}}}}}]}}
```

调用 Tool：

```
{"jsonrpc":"2.0","method":"tools/call","id":3,"params":{"name":"get_weather","arguments":
{"city":"Tokyo"}}}
```

实战：从零搭建 MCP Server

以下是一个完整的 Go 实现示例，演示创建一个简单 MCP Server 的最小路径：

Step 1：初始化项目

```
mkdir mcp-weather-server && cd mcp-weather-server
go mod init mcp-weather-server
go get github.com/modelcontextprotocol/go-sdk/mcp
```

Step 2：编写 Server 代码

```
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/modelcontextprotocol/go-sdk/mcp"
)
```

```go
func main() {
    // 创建 Stdio 传输（本地模式）
    transport := mcp.NewStdioTransport()

    // 创建 Server 实例
    server := mcp.NewServer(&mcp.ServerInfo{Name: "weather-server", Version: "1.0.0"})

    // 注册 Tool：查询天气
    server.AddTool(mcp.Tool{
        Name:        "get_weather",
        Description: "Query weather for a given city",
        InputSchema: map[string]any{
            "type":       "object",
            "properties": map[string]any{"city": map[string]string{"type": "string"}},
            "required":   []string{"city"},
        },
    }, handleGetWeather)

    // 启动服务
    if err := server.Run(context.Background(), transport); err != nil {
        log.Fatal(err)
    }
}
```

```go
// 处理天气查询的核心逻辑
func handleGetWeather(ctx context.Context, req *mcp.CallToolRequest) (*mcp.CallToolResult, error) {
    city := req.Arguments["city"].(string)

    // 此处接入真实天气 API
    response := fetchFromOpenWeather(city)

    return &mcp.CallToolResult{
        Content: []mcp.Content{mcp.TextContent(response)},
    }, nil
}

func fetchFromOpenWeather(city string) string {
    // TODO: 调用 OpenWeatherMap API
    return fmt.Sprintf("City %s: Sunny, 23°C", city)
}
```

Step 3：配置 Claude Desktop 使用

在 Claude Desktop 的配置文件中添加 Server：

```
{
    "mcpServers": {
        "weather": {
            "command": "go",
            "args": ["run", "/path/to/mcp-weather-server"],
            "transport": "stdio"
        }
    }
}
```

Step 4：验证效果

启动 Claude Desktop 后，在对话中直接告诉 AI "帮我查一下北京的天气"，AI 会自动识别调用 `get_weather` Tool。

代码说明

以上代码展示了 MCP Server 的三个核心步骤：**创建 Server → 注册 Tool → 启动传输**。实际项目中只需替换 `handleGetWeather` 中的业务逻辑即可。

常见开源 MCP Server 参考

Server	用途	技术栈
Filesystem	读写本地文件系统	TypeScript
GitHub	Issue/PR/Code Search	TypeScript
PostgreSQL	数据库查询	Python
GitLab	GitLab 项目管理	TypeScript
Slack	消息收发	Python
Memory	向量存储的知识库	TypeScript

这些参考实现在 MCP 官方仓库的 `servers` 目录下，可以直接编译运行或二次开发。

安全考虑

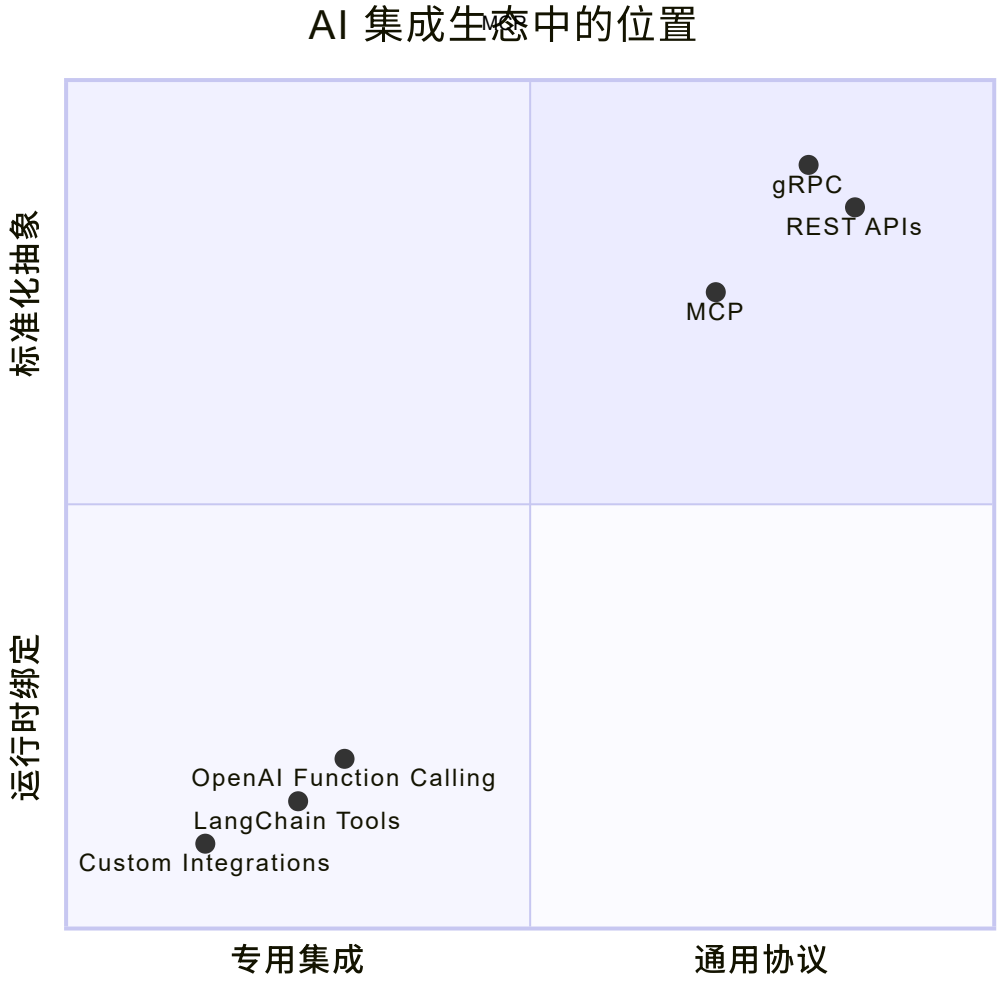
MCP 的设计将权限控制交给了 Host Application（宿主应用），这带来了灵活性的同时也需要注意安全风险：

⚠ 安全最佳实践

- **沙箱隔离**：对 Server 的文件/网络访问权限进行限制，避免恶意 Server 读写敏感数据

- **权限最小化**：只授予 Server 完成工作所需的最小权限集
- **审计日志**：记录所有 Tool 调用和 Resource 访问，便于追踪异常行为
- **用户确认**：对破坏性操作（删除、修改）应在 AI 执行前请求用户确认
- **依赖安全**：检查 Server 的代码依赖，防止供应链攻击

MCP 与相关技术的关系



核心区别：

对比维度	Function Calling	MCP
作用域	LLM API 内部的能力调用机制	跨应用的独立通信协议
可复用性	每次调用需重新定义 function deffinition	一次部署，所有兼容 Client 共享

对比维度	Function Calling	MCP
生命周期	单次请求内有效	持久运行的独立进程
生态位	"LLM 怎么调用函数"	"应用之间怎么交换能力和数据"

两者可以共存

MCP Server 内部仍然可以使用 Function Calling 调用的方式与 LLM 交互。MCP 解决的是"应用层面的互操作"，Function Calling 解决的是"模型层面的函数感知"——二者在不同层级工作。

本章小结

概念	定位	一句话理解
Tools	操作型能力	让 AI "动手做事"
Resources	数据型能力	让 AI "读取数据"
Prompts	模板型能力	让 AI "复用话术"
stdio 传输	本地通信	进程间 stdin/stdout

概念	定位	一句话理解
SSE 传输	网络通信	HTTP 流式推送
JSON-RPC 2.0	消息协议	统一的请求-响应格式

✔ **学习成果**

完成本章后，你应该能够：

- 理解 MCP 的 Client-Server 架构和工作原理
- 区分 Tools、Resources、Prompts 三种核心能力的不同场景
- 具备从零搭建一个 MCP Server 的能力
- 了解 MCP 与传统 Function Calling 的关系与差异

扩展阅读

- MCP 官方规范：[https://modelcontextprotocol.io/specification]
- Go SDK 文档：[https://github.com/modelcontextprotocol/go-sdk]
- TypeScript SDK 文档：[https://github.com/modelcontextprotocol/sdk]

- 官方参考 Server：[https://github.com/modelcontextprotocol/servers]

关联笔记