MCP (Model Context Protocol) 技术规格文档

版本: 1.0
日期: 2025-09-11
作者: Manus AI

1. 简介

Model Context Protocol (MCP) 是一个开放的、标准化的协议，旨在增强大型语言模型（LLM）与外部应用程序、数据源和工具之间的交互。MCP通过提供一个统一的接口，使得LLM能够无缝地集成到各种复杂的应用场景中，从而极大地扩展了AI系统的能力和灵活性。无论是构建AI驱动的集成开发环境（IDE）、增强聊天界面，还是创建自定义的AI工作流，MCP都提供了一种标准化的方式来连接LLM与所需的上下文信息。

本文档详细描述了MCP的技术规格，包括协议的架构设计、核心组件、生命周期管理、传输机制、API规范以及安全考量。本文档旨在为开发者提供一个全面的指南，以便在他们的应用程序中实现和利用MCP。

2. 协议概述

MCP的核心目标是为LLM应用提供一个标准化的上下文交换机制。它借鉴了Language Server Protocol (LSP) 的成功经验，旨在为AI应用生态系统提供一个通用的“USB-C接口”，使得不同的AI模型、数据源和工具能够通过一个统一的协议进行互操作。

2.1. 设计原则

MCP的设计遵循以下几个核心原则：

- **简单易用**: 服务器的构建应该极其简单，使得开发者能够轻松地将他们的数据源和工具集成到MCP生态系统中。
- **高度可组合**: 每个服务器都提供专注的功能，多个服务器可以无缝地组合在一起，形成强大的工作流。
- **安全可靠**: 协议设计充分考虑了安全和隐私问题，通过明确的权限控制和用户授权机制，确保数据和操作的安全性。
- **渐进式功能增强**: 协议的核心功能集非常小，开发者可以根据需要逐步添加和协商更高级的功能，从而保证了协议的灵活性和可扩展性。

2.2. 核心能力

MCP为应用程序提供了以下核心能力：

- **共享上下文信息**: 应用程序可以与语言模型共享丰富的上下文信息，例如文件内容、数据库模式、项目结构等。
- **暴露工具和能力**: 应用程序可以向AI系统暴露各种工具和能力，使得模型能够执行外部操作，例如调用API、查询数据库、执行代码等。
- **构建可组合的集成和工作流**: 通过将多个MCP服务器组合在一起，开发者可以构建出功能强大的、可组合的AI工作流。

3. 架构设计

MCP采用了一种客户端-主机-服务器（Client-Host-Server）的架构。这种架构使得用户能够在不同的应用程序中集成AI能力，同时保持清晰的安全边界和关注点分离。

3.1. 核心组件

MCP的架构由三个核心组件构成：

- **主机 (Host)**: 主机是运行LLM应用程序的进程，它充当着容器和协调者的角色。主机负责创建和管理多个客户端实例，控制客户端的连接权限和生命周期，执行安全策略和用户授权，以及协调AI/LLM的集成和采样。
- **客户端 (Client)**: 客户端由主机创建，每个客户端与一个服务器建立一个隔离的、有状态的连接。客户端负责处理协议的协商和能力交换，双向路由协议消息，管理订阅和通知，以及维护服务器之间的安全边界。
- **服务器 (Server)**: 服务器提供专门的上下文和能力。服务器通过MCP的原语暴露资源、工具和提示。服务器可以独立运行，具有专注的职责，并且可以通过客户端接口请求采样。服务器可以是本地进程，也可以是远程服务。

3.2. 组件交互

主机应用程序创建和管理多个客户端，每个客户端与一个特定的服务器建立1:1的关系。这种设计使得主机能够将来自不同服务器的上下文信息聚合在一起，并将其提供给LLM。同时，主机也能够将LLM的输出分发给相应的服务器，从而实现复杂的交互逻辑。

4. 协议生命周期

MCP为客户端和服务器之间的连接定义了一个严格的生命周期，以确保正确的能力协商和状态管理。生命周期包括三个阶段：初始化、操作和关闭。

4.1. 初始化阶段

初始化阶段是客户端和服务器之间的第一次交互。在此阶段，双方必须就协议版本、能力和实现细节达成一致。客户端通过发送一个 `initialize` 请求来启动初始化过程，服务器则通过一个

`initialize` 响应来回应。成功初始化后，客户端必须发送一个 `initialized` 通知，表示已准备好进入操作阶段。

4.2. 操作阶段

在操作阶段，客户端和服务器根据协商的能力交换消息。双方必须遵守协商的协议版本，并且只能使用成功协商的能力。

4.3. 关闭阶段

在关闭阶段，一方（通常是客户端）会优雅地终止协议连接。协议本身没有定义特定的关闭消息，而是依赖于底层的传输机制来信令连接的终止。

5. 传输机制

MCP使用JSON-RPC 2.0来编码消息，并且消息必须是UTF-8编码的。协议目前定义了两种标准的传输机制：

5.1. stdio

在stdio传输中，客户端将MCP服务器作为子进程启动。服务器从其标准输入（stdin）读取JSON-RPC消息，并将其响应发送到其标准输出（stdout）。这种传输机制非常适合本地集成的场景。

5.2. Streamable HTTP

在Streamable HTTP传输中，服务器作为一个独立的进程运行，可以处理多个客户端的连接。这种传输机制使用HTTP POST和GET请求，并可选择性地使用Server-Sent Events (SSE)来流式传输服务器消息。这种机制非常适合远程集成的场景。

6. API 规范

MCP定义了一系列丰富的API，用于实现上下文交换、工具调用、资源管理等功能。

6.1. 服务器功能

服务器可以向客户端提供以下功能：

- **资源 (Resources)**: 服务器可以暴露各种资源，例如文件、数据库表、API端点等。客户端可以列出、读取和订阅这些资源。
- **工具 (Tools)**: 服务器可以暴露各种工具，供语言模型调用。每个工具都有一个唯一的名称和描述其功能的元数据。
- **提示 (Prompts)**: 服务器可以提供模板化的提示和工作流，以引导用户和AI模型的交互。

6.2. 客户端功能

客户端可以向服务器提供以下功能：

- **采样 (Sampling)**: 服务器可以请求客户端进行LLM采样，从而实现代理行为和递归的LLM交互。
- **根 (Roots)**: 服务器可以查询客户端的URI或文件系统边界，以便在其中进行操作。
- **引出 (Elicitation)**: 服务器可以请求客户端从用户那里获取额外的信息。

7. 安全与信任

MCP的设计充分考虑了安全和信任问题。由于协议允许任意的数据访问和代码执行，因此所有实现者都必须仔细处理相关的安全风险。

7.1. 核心原则

- **用户同意和控制**: 用户必须明确同意并理解所有的数据访问和操作。用户必须能够控制共享哪些数据以及执行哪些操作。
- **数据隐私**: 主机必须在向服务器暴露用户数据之前获得明确的用户同意。用户数据应受到适当的访问控制保护。
- **工具安全**: 工具代表着任意代码的执行，必须谨慎对待。主机必须在调用任何工具之前获得明确的用户同意。
- **LLM采样控制**: 用户必须明确批准任何LLM采样请求。用户应该能够控制采样的发生、发送的实际提示以及服务器可以看到的结果。

7.2. 实现指南

虽然MCP协议本身无法在协议层面强制执行这些安全原则，但实现者应该：

- 在其应用程序中构建强大的同意和授权流程。
- 提供清晰的安全影响文档。
- 实现适当的访问控制和数据保护。
- 在其集成中遵循安全最佳实践。
- 在其功能设计中考虑隐私影响。

8. 结论

Model Context Protocol (MCP) 为构建下一代AI应用提供了一个强大而灵活的框架。通过标准化的接口和可组合的架构，MCP使得开发者能够轻松地将语言模型与外部世界连接起来，从而释放出AI的巨大潜力。我们相信，MCP将成为推动AI应用创新和发展的重要基石。

9. 参考文献

- [1] Model Context Protocol Specification. https://modelcontextprotocol.io/specification/2025-06-18
- [2] JSON-RPC 2.0 Specification. https://www.jsonrpc.org/specification
- [3] RFC2119: Key words for use in RFCs to Indicate Requirement Levels. https://www.rfc-editor.org/rfc/rfc2119
- [4] RFC8174: Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. https://www.rfc-editor.org/rfc/rfc8174

10. 详细API规范

10.1. 初始化API

10.1.1. initialize 请求

客户端必须发送一个 `initialize` 请求来启动协议会话。请求包含客户端支持的协议版本、能力和实现信息。

请求格式：

JSON

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "capabilities": {
      "roots": {
        "listChanged": true
      },
      "sampling": {},
      "elicitation": {}
    },
    "clientInfo": {
      "name": "ExampleClient",
      "title": "Example Client Display Name",
      "version": "1.0.0"
    }
  }
}
```

响应格式：

JSON

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2025-06-18",
    "capabilities": {
      "logging": {},
      "prompts": {
        "listChanged": true
      },
      "resources": {
        "subscribe": true,
        "listChanged": true
      },
      "tools": {
        "listChanged": true
      }
    },
    "serverInfo": {
      "name": "ExampleServer",
      "title": "Example Server Display Name",
      "version": "1.0.0"
    },
    "instructions": "Optional instructions for the client"
  }
}
```

10.1.2. initialized 通知

成功初始化后，客户端必须发送一个 `initialized` 通知：

JSON

```
{
  "jsonrpc": "2.0",
  "method": "notifications/initialized"
}
```

10.2. 资源管理API

10.2.1. resources/list 请求

用于发现可用资源的请求，支持分页。

请求格式：

JSON

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "resources/list",
  "params": {
    "cursor": "optional-cursor-value"
  }
}
```

响应格式：

JSON

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "resources": [
      {
        "uri": "file:///project/src/main.rs",
        "name": "main.rs",
        "title": "Rust Software Application Main File",
        "description": "Primary application entry point",
        "mimeType": "text/x-rust",
        "annotations": {
          "audience": ["user", "assistant"],
          "priority": 0.8,
          "lastModified": "2025-01-12T15:00:58Z"
        }
      }
    ],
    "nextCursor": "next-page-cursor"
  }
}
```

10.2.2. resources/read 请求

用于检索资源内容的请求。

请求格式：

JSON

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "resources/read",
  "params": {
    "uri": "file:///project/src/main.rs"
  }
}
```

响应格式：

JSON

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "contents": [
      {
        "uri": "file:///project/src/main.rs",
        "name": "main.rs",
        "title": "Rust Software Application Main File",
        "mimeType": "text/x-rust",
        "text": "fn main() {\n    println!(\"Hello world!\");\n}"
      }
    ]
  }
}
```

10.2.3. resources/subscribe 请求

用于订阅资源变更通知的请求。

请求格式：

JSON

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "method": "resources/subscribe",
  "params": {
    "uri": "file:///project/src/main.rs"
  }
}
```

更新通知：

```JSON
{
  "jsonrpc": "2.0",
  "method": "notifications/resources/updated",
  "params": {
    "uri": "file:///project/src/main.rs",
    "title": "Rust Software Application Main File"
  }
}
```

10.3. 工具管理API

10.3.1. tools/list 请求

用于发现可用工具的请求，支持分页。

请求格式：

```JSON
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/list",
  "params": {
    "cursor": "optional-cursor-value"
  }
}
```

响应格式：

```JSON
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "tools": [
      {
        "name": "get_weather",
        "title": "Weather Information Provider",
        "description": "Get current weather information for a location",
        "inputSchema": {
```

```
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "City name or zip code"
            }
          },
          "required": ["location"]
        },
        "outputSchema": {
          "type": "object",
          "properties": {
            "temperature": {
              "type": "number",
              "description": "Temperature in celsius"
            },
            "conditions": {
              "type": "string",
              "description": "Weather conditions description"
            }
          },
          "required": ["temperature", "conditions"]
        }
      }
    ],
    "nextCursor": "next-page-cursor"
  }
}
```

10.3.2. tools/call 请求

用于调用工具的请求。

请求格式：

JSON

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "location": "New York"
    }
```

```
  }
}
```

响应格式：

JSON

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Current weather in New York:\nTemperature:
22°C\nConditions: Partly cloudy"
      }
    ],
    "structuredContent": {
      "temperature": 22,
      "conditions": "Partly cloudy",
      "humidity": 65
    },
    "isError": false
  }
}
```

10.4. 提示管理API

10.4.1. prompts/list 请求

用于发现可用提示模板的请求。

请求格式：

JSON

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "prompts/list"
}
```

响应格式：

JSON

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "prompts": [
      {
        "name": "code_review",
        "title": "Code Review Assistant",
        "description": "Analyze code for potential issues and improvements",
        "arguments": [
          {
            "name": "code",
            "description": "The code to review",
            "required": true
          },
          {
            "name": "language",
            "description": "Programming language",
            "required": false
          }
        ]
      }
    ]
  }
}
```

10.4.2. prompts/get 请求

用于获取特定提示模板的请求。

请求格式：

JSON

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "prompts/get",
  "params": {
    "name": "code_review",
    "arguments": {
      "code": "def hello():\n    print('Hello, World!')",
      "language": "python"
    }
  }
}
```

响应格式：

JSON

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "description": "Code review for Python function",
    "messages": [
      {
        "role": "user",
        "content": {
          "type": "text",
          "text": "Please review this Python code:\n\ndef hello():\n
print('Hello, World!')\n\nProvide feedback on code quality, style, and
potential improvements."
        }
      }
    ]
  }
}
```

11. 错误处理

11.1. 标准错误代码

MCP使用JSON-RPC 2.0的标准错误代码，并定义了一些特定于协议的错误：

错误代码	名称	描述
-32700	Parse error	无效的JSON
-32600	Invalid Request	JSON-RPC请求无效
-32601	Method not found	方法不存在
-32602	Invalid params	参数无效
-32603	Internal error	内部错误
-32000	Server error	服务器错误

11.2. 错误响应格式

JSON

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32602,
    "message": "Unsupported protocol version",
    "data": {
      "supported": ["2025-06-18"],
      "requested": "1.0.0"
    }
  }
}
```

11.3. 超时处理

实现应该为所有发送的请求建立超时机制，以防止连接挂起和资源耗尽。当请求在超时期间内没有收到成功或错误响应时，发送方应该为该请求发出取消通知并停止等待响应。

取消通知格式：

JSON

```
{
  "jsonrpc": "2.0",
  "method": "notifications/cancelled",
  "params": {
    "requestId": 1,
    "reason": "Request timed out"
  }
}
```

12. 进度跟踪

对于长时间运行的操作，MCP支持进度通知机制。

进度通知格式：

JSON

```
{
  "jsonrpc": "2.0",
  "method": "notifications/progress",
  "params": {
    "progressToken": "operation-123",
```

```
    "progress": 0.5,
    "total": 1.0
  }
}
```

13. 日志记录

服务器可以向客户端发送结构化的日志消息。

日志消息格式：

JSON

```
{
  "jsonrpc": "2.0",
  "method": "notifications/message",
  "params": {
    "level": "info",
    "logger": "weather-service",
    "data": "Successfully retrieved weather data for New York"
  }
}
```

支持的日志级别：

- `debug`
- `info`
- `notice`
- `warning`
- `error`
- `critical`
- `alert`
- `emergency`

14. 实现指南

14.1. 服务器实现

实现MCP服务器时，开发者应该：

1. **明确定义服务器的职责范围**：每个服务器应该专注于特定的功能域，避免功能过于复杂。

2. **实现必要的生命周期方法**：确保正确处理初始化、操作和关闭阶段。
3. **声明准确的能力**：只声明服务器实际支持的能力，避免误导客户端。
4. **提供清晰的文档**：为暴露的资源、工具和提示提供详细的描述和使用示例。
5. **处理错误情况**：实现健壮的错误处理机制，提供有意义的错误消息。

14.2. 客户端实现

实现MCP客户端时，开发者应该：

1. **实现能力协商**：正确处理服务器的能力声明，只使用协商成功的功能。
2. **管理多个连接**：支持同时连接到多个服务器，并正确管理每个连接的状态。
3. **实现用户授权**：在执行敏感操作之前，确保获得用户的明确授权。
4. **处理异步操作**：正确处理长时间运行的操作和进度通知。
5. **实现安全控制**：确保用户数据的安全性和隐私性。

14.3. 最佳实践

1. **使用语义化的URI**：为资源使用有意义的URI，便于理解和调试。
2. **提供丰富的元数据**：为资源、工具和提示提供详细的描述和注释。
3. **实现幂等操作**：确保重复调用同一操作不会产生副作用。
4. **使用结构化数据**：尽可能使用结构化的数据格式，便于程序处理。
5. **实现缓存机制**：对于不经常变化的数据，实现适当的缓存机制以提高性能。

15. 扩展机制

MCP设计为可扩展的协议。开发者可以通过以下方式扩展协议：

15.1. 自定义能力

开发者可以在 `experimental` 能力字段中定义自定义能力：

JSON

```
{
  "capabilities": {
    "experimental": {
      "customFeature": {
        "version": "1.0",
        "description": "Custom feature description"
      }
    }
```

```
  }
}
```

15.2. 自定义URI方案

开发者可以定义自定义的URI方案来标识特定类型的资源：

```
Plain Text

custom-scheme://server-id/resource-path?param=value
```

15.3. 自定义传输

除了标准的stdio和HTTP传输外，开发者还可以实现自定义的传输机制，例如WebSocket、gRPC等。

16. 版本控制和兼容性

16.1. 版本格式

MCP使用日期格式的版本号，例如"2025-06-18"。这种格式清楚地表明了协议的发布日期。

16.2. 向后兼容性

MCP致力于保持向后兼容性。新版本的协议应该能够与旧版本的实现进行互操作。当引入不兼容的变更时，协议版本号会相应更新。

16.3. 版本协商

在初始化阶段，客户端和服务器会协商使用的协议版本。如果双方支持的版本不兼容，连接将被拒绝。

17. 性能考虑

17.1. 消息大小

虽然MCP没有严格限制消息的大小，但实现者应该考虑以下因素：

- 大型资源应该支持分块传输
- 工具调用的参数和结果应该保持合理的大小
- 使用压缩来减少网络传输的开销

17.2. 连接管理

- 实现连接池来复用HTTP连接
- 使用keep-alive机制来减少连接建立的开销
- 实现适当的超时和重试机制

17.3. 缓存策略

- 对不经常变化的资源实现客户端缓存
- 使用ETag或Last-Modified头来支持条件请求
- 实现适当的缓存失效机制

18. 测试和调试

18.1. 测试策略

1. **单元测试**：测试各个组件的功能
2. **集成测试**：测试客户端和服务器之间的交互
3. **性能测试**：测试在高负载下的性能表现
4. **安全测试**：测试安全控制和权限管理

18.2. 调试工具

- 使用日志记录来跟踪消息流
- 实现消息转储功能用于调试
- 提供健康检查端点来监控服务状态

19. 部署和运维

19.1. 部署模式

1. **本地部署**：服务器作为本地进程运行
2. **容器化部署**：使用Docker等容器技术部署服务器
3. **云部署**：在云平台上部署服务器
4. **边缘部署**：在边缘设备上部署轻量级服务器

19.2. 监控和告警

- 监控连接数和请求量

- 跟踪错误率和响应时间
- 设置适当的告警阈值
- 实现健康检查和自动恢复机制

20. 总结

Model Context Protocol (MCP) 为构建智能、可组合的AI应用提供了一个强大的基础设施。通过其标准化的接口、灵活的架构和丰富的功能集，MCP使得开发者能够轻松地将语言模型与各种外部系统集成在一起。

本技术规格文档详细描述了MCP的各个方面，从基本概念到具体的实现细节。我们希望这份文档能够帮助开发者更好地理解和使用MCP，从而构建出更加智能和有用的AI应用。

随着AI技术的不断发展，MCP也将持续演进，以满足新的需求和挑战。我们鼓励社区的积极参与，共同推动MCP生态系统的发展和完善。

文档版本: 1.0

最后更新: 2025-09-11

作者: Manus AI

联系方式: 如有问题或建议，请访问 https://modelcontextprotocol.io