

MCPサーバーの基礎から実践レベルの知識まで

自己紹介

- ▶ azukiazusa
- ▶ <https://azukiazusa.dev>
- ▶ FE(フロントエンド|ファイアーエムブレム)が
好き



アジェンダ

- ▶ MCP サーバーの基礎知識について(10分)
- ▶ MCP サーバー構築デモ(15分)
- ▶ MCP サーバーの実践的な知識について(20分)

MCPとは何か

Model Context Protocol (MCP)

アプリケーションが LLM にコンテキストを渡す方法を標準化するためのプロトコル

- ▶ Anthropicが開発・発表
- ▶ LSP(Language Server Protocol)の発想を参考
- ▶ ツールのインターフェースを統一

なぜMCPが必要なのか?

- ・ LLMには知識カットオフがあり、最新の情報や組織内の情報を取得できない
- ・ Web検索をしたり、社内ドキュメントを参照しその情報をコンテキストに渡す必要がある
- ・ 従来は function calling といった仕組みが使われてきた
 - ・ 天気情報を取得するために天気情報 API を呼び出したり、Slack API を呼び出してメッセージを送信したりする関数を LLM が呼び出す

function calling

- ▶ ツールのインターフェースが標準化されていない
- ▶ LLMごとに異なる実装が必要
- ▶ ツールを配布する手段がない
- ▶ 開発者が独自に実装する必要がある

function callingの例

```
const response = await openai.chat.completions.create({  
  model: "gpt-4",  
  tools: [  
    {  
      type: "function",  
      function: {  
        name: "get_weather",  
        description: "天気を取得",  
        parameters: {  
          type: "object",  
          properties: {  
            location: {  
              type: "string",  
            },  
          },  
        },  
      },  
    },  
  ],  
});
```

```
const response = await anthropic.messages.create({  
  model: "claude-3-5-sonnet",  
  tools: [  
    {  
      name: "get_weather",  
      description: "天気を取得",  
      input_schema: {  
        type: "object",  
        properties: {  
          location: {  
            type: "string",  
          },  
        },  
      },  
    },  
  ],  
});
```

MCP が解決したこと

- ・ ツールのインターフェースを標準化し、パッケージマネージャーでの配布も容易になった
- ・ 各プログラミング言語向けの SDK が提供されているため、効率よく MCP サーバーを開発できた
- ・ 企業が自社のデータを LLM に提供する手段として普及が進んだ
- ・ 現在では Anthropic が提供する Claude だけでなく、OpenAI の GPT や Google の Gemini など、主要な LLM が MCP をサポートし事実上の標準となっている

MCPがあるとできること

- ▶ Googleカレンダーに予定を追加
- ▶ Notionにメモを追加
- ▶ Slackにメッセージを送信
- ▶ Web検索を実行

MCPの仕組み

MCP クライアント・サーバーモデル



MCPのトランSPORT

stdio(標準入出力)

- 標準入力/出力を使用した通信
- ローカル環境で動作

Streamable HTTP

- HTTP を使用した通信
- Web ブラウザ上で動作するので注目を浴びている

SSE(非推奨)

- 互換性維持のために残されている

JSON-RPC 2.0

- ▶ JSON-RPC 2.0 (<https://www.jsonrpc.org/specification>)を使用して通信
- ▶ JSON-RPC とは、リモートプロシージャコール (RPC) を JSON フォーマットで実装するための軽量なプロトコル

```
{  
  "jsonrpc": "2.0",  
  "id": 1,  
  "method": "tools/call",  
  "params": {  
    "name": "get_weather",  
    "arguments": {  
      "location": "Tokyo"  
    }  
  }  
}
```

MCPの3つの機能

リソース

ユーザーや LLM がアクセスできるデータ

プロンプト

再利用可能なプロンプトテンプレート

ツール

LLM が呼び出せる外部ツール

→ この発表ではツールに焦点を当てる

MCP サーバー構築デモ

TypeScript SDK を使用してサイコロツールを実装

TypeScript SDKのインストール

```
npm install @modelcontextprotocol/sdk zod
```

サーバーの基本構造

```
import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";

const server = new McpServer({
  name: "dice-server",
  version: "1.0.0",
});
```

ツールの定義と実装

```
import { z } from "zod";

server.registerTool(
  // ツール名
  "roll_dice",
  {
    title: "Roll Dice", // 人間が読めるツールの名前
    description: "ランダムな数字を生成するサイコロツール", // ツールの説明
    // ツールの入力スキーマ
    inputSchema: {
      sides: z.number().optional().describe("サイコロの面の数(デフォルト: 6)"),
    },
    // ツールの出力スキーマ
    outputSchema: {
      result: z.number(),
    },
  },
  async ({ sides = 6 }) => {
    const result = Math.floor(Math.random() * sides) + 1;

    return {
      content: [{ type: "text", text: JSON.stringify({ result }) }],
      structuredContent: { result },
    };
  },
);
```

- ▶ `description` はツールがツールを呼び出す判断に影響するため重要
- ▶ Zod でスキーマを定義し、LLM に入力と出力の形式を伝える
- ▶ 第 3 引数で呼び出した関数の結果が LLM に返される

サーバーの起動(stdio)

```
import { StdioServerTransport } from "@modelcontextprotocol/sdk/server/stdio.js";  
  
const transport = new StdioServerTransport();  
await server.connect(transport);
```

stdio 通信でサーバーを起動

Streamable HTTP の場合

```
import { StreamableHTTPServerTransport } from "@modelcontextprotocol/sdk/server/streamableHttp.js";
import express from "express";

const app = express();
app.use(express.json());

app.post("/mcp", async (req, res) => {
  const transport = new StreamableHTTPServerTransport({
    sessionIdGenerator: undefined,
    enableJsonResponse: true,
  });

  res.on("close", () => {
    transport.close();
  });

  await server.connect(transport);
  await transport.handleRequest(req, res, req.body);
});

const port = parseInt(process.env.PORT || "3000");
app.listen(port, () => {
  console.log(`MCP Server running on http://localhost:${port}/mcp`);
});
```

MCP Inspectorで動作確認

```
npx @modelcontextprotocol/inspector node build/index.js
```

- ブラウザでツールの動作をテスト
- リクエスト・レスポンスの確認
- デバッグに便利

Claude Desktopでの設定

```
~/Library/Application Support/Claude/clade_desktop_config.json
```

```
{  
  "mcpServers": {  
    "dice": {  
      "command": "node",  
      "args": ["/path/to/build/index.js"]  
    }  
  }  
}
```

Claude Desktopで実行

- ▶ Claude Desktopを再起動
- ▶ チャットで「サイコロを振って」と入力
- ▶ ツールが自動的に呼び出される
- ▶ 結果が返される

MCPサーバーの実践的な知識

- 私が実際に本番レベルで MCP サーバーを開発してきた中で得た知見・失敗談を共有

Web API の設計知識は捨てる

APIのラッパーとして提供すると失敗する

- ▶ REST APIはエンドポイントベースの設計
 - ▶ 1つのリソースに対して GET、POST、PUT、DELETE などの操作ごとにエンドポイントが存在
- ▶ ツールはユーザーが達成したいタスクベースの設計
- ▶ プログラミングでは1つのタスクを達成するために複数の API を組み合わせることが一般的
- ▶ LLM は複数のツールを組み合わせてタスクを達成することが苦手

例: カレンダーAPI

従来のAPI設計

- ▶ `GET /users` - ユーザー取得
 - ▶ `GET /events` - イベント取得
 - ▶ `POST /events` - イベント作成
- この設計に従うと、`get_users`, `get_events`, `create_event` といったツールを作りたくなる

より効果的なツール設計

- ▶ `schedule_meeting` - ミーティングをスケジュール（ツールの実装の中で複数のAPIを呼び出す）
- 1つのタスクを1つのツールで完結

従来のプログラミングの例

```
async function scheduleMeetingWithTanaka() {
  // 1. ユーザーを検索
  const users = await fetch("https://api.example.com/users?name=田中").then(
    (r) => r.json(),
  );
  const tanaka = users.find((u) => u.name.includes("田中"));

  // 2. 田中さんの予定を取得
  const events = await fetch(
    `https://api.example.com/events?userId=${tanaka.id}&date=2025-10-06`,
  ).then((r) => r.json());

  // 3. 空き時間を見つける
  const freeSlots = findFreeSlots(events);

  // 4. ミーティングを作成
  await fetch("https://api.example.com/events", {
    method: "POST",
    body: JSON.stringify({
      title: "定例会議",
      attendees: [tanaka.id],
      startTime: freeSlots[0].start,
      endTime: freeSlots[0].end,
    }),
  });
}
```

MCPサーバーの例

```
// MCP ツール: 1つのツールでタスクを完結
server.registerTool(
  "schedule_meeting",
  {
    title: "Schedule Meeting",
    description: "指定したメンバーとのミーティングをスケジュール",
    inputSchema: {
      attendeeName: z.string().describe("参加者の名前"),
      title: z.string().describe("ミーティングのタイトル"),
      date: z.string().describe("日付(YYYY-MM-DD)"),
      duration: z.number().describe("所要時間(分)"),
    },
    outputSchema: { eventId: z.string(), startTime: z.string() },
  },
  async ({ attendeeName, title, date, duration }) => {
    // ツール内部で全ての処理を実行
    const user = await searchUser(attendeeName);
    const freeSlot = await findFreeSlot(user.id, date, duration);
    const event = await createEvent(title, [user.id], freeSlot);

    return {
      content: [{ type: "text", text: `ミーティングを作成しました` }],
      structuredContent: { eventId: event.id, startTime: freeSlot.start },
    };
  },
);
```

ツール設計のポイント

- ▶ 提供するツールの数が多くなると、LLMがどのツールを使うべきか迷ってしまう
- ▶ ユーザーが何を達成したいのか、ユースケースを考えてツールを設計する
 - ▶ ユーザーを取得したいのは何のため？
 - ▶ 会議のスケジュールのために空き時間を知りたいの真の目的

コンテキストが大きくなりすぎる問題

LLMにはコンテキスト長の制限がある

- ▶ Claude Code: デフォルトで 25,000 トークン
- ▶ コンテキストが大きいと性能が低下
- ▶ LLM が無関係な情報に注意を奪われる → コンテキスト汚染

従来のプログラミングとの違い

- ・ 現代の富豪的プログラミングでは1,000件のリストをメモリに載せてフィルタリング・ソートしても問題ない
- ・ LLM ではコンテキスト制限があるため同じアプローチは不可
→ API の応答をそのまま返すのは避ける

解決策1: ページネーションの導入

```
server.registerTool(
  "search_users",
  {
    inputSchema: {
      query: z.string().describe("検索クエリ"),
      limit: z.number().optional().default(10).describe("取得件数"),
      offset: z.number().optional().default(0).describe("オフセット"),
    },
  },
  async ({ query, limit = 10, offset = 0 }) => {
    const users = await db.users.search(query).limit(limit).offset(offset);
    const total = await db.users.count(query);

    const result = {
      users,
      total,
      hasMore: offset + limit < total,
    };

    return {
      content: [{ type: "text", text: JSON.stringify(result) }],
      structuredContent: result,
    };
  },
);
```

解決策2: 必要なフィールドだけ取得

```
server.registerTool(
  "get_user",
  {
    inputSchema: {
      userId: z.string().describe("ユーザーID"),
      fields: z
        .array(z.enum(["name", "email", "avatar", "bio"]))
        .optional()
        .default(["name", "email"])
        .describe("取得するフィールド"),
    },
    },
    async ({ userId, fields = ["name", "email"] }) => {
      const user = await db.users.findById(userId).select(fields);

      return {
        content: [{ type: "text", text: JSON.stringify(user) }],
        structuredContent: user,
      };
    },
);
```

解決策3: データを要約・整形

```
server.registerTool(  
  "get_log_summary",  
  {  
    inputSchema: {  
      responseFormat: z  
        .enum(["detailed", "summary"])  
        .optional()  
        .default("summary"),  
    },  
    },  
    async ({ responseFormat = "summary" }) => {  
      const logs = await parseLogs(date);  
      const result = responseFormat === "summary" ? summarizeLogs(logs) : logs;  
      return {  
        content: [{ type: "text", text: JSON.stringify(result) }],  
        structuredContent: result,  
      };  
    },  
);
```

LLMが誤ったツール呼び出しを行う

ホストメトリック取得ツールで存在しないメトリック名を繰り返し呼び出して失敗し続ける

解決策1: **description** のプロンプトエンジニアリング

- ・ ツールの説明は LLM のコンテキストに含まれるので、プロンプトエンジニアリングの知識が活用できる
- ・ ユーザーがどのような場面でツールを使うべきかを明示する
- ・ サポートしている値の一覧を含める
- ・ ツールの使用例を Few-shot で示す



ツールの `description` の良い例

```
{  
  name: "get_host_metric",  
  description: `ホストのメトリックを取得します。`
```

サポートされているメトリック:

- `cpu_usage`: CPU使用率
- `memory_usage`: メモリ使用率
- `disk_usage`: ディスク使用率

以下のユーザーの質問に答えるためにこのツールを使用してください:

- ホストAはスケールアップが必要かどうか調査してください
- ホストBのパフォーマンスをグラフで表示してください
- ホストCのディスク使用率が高いか確認してください

<example>

- ホストAのCPU使用率を取得: `get_host_metric("hostA", "cpu_usage")`
- ホストBのメモリ使用率を取得: `get_host_metric("hostB", "memory_usage")`

</example>

```
,  
// ...
```

```
}
```

解決策2: エラー応答を詳細にする

悪い例

```
{  
  "code": 404,  
  "message": "Not Found"  
}
```

LLMにとって意味のない応答

エラー応答をドキュメンテーションする

- ▶ なぜツールの呼び出しが失敗したのか、どのように問題を解決できるのかをマークダウン形式で返す
 - ▶ エラーコードやスタックトレースを返すのではなく、具体的かつ実用的な情報を提供
- ▶ エラー応答もプロンプトエンジニアリングの一種
- ▶ 必ずしも JSON 形式で返す必要はない

良いエラー応答の例

```
server.registerTool("get_host_metric", async ({ host, metric }) => {
  try {
    // ...
  } catch (error) {
    const errorMessage = `## Metric Not Found

Your request failed because the specified metric name is invalid
or not available for this host.

## Error Summary

${error.message}

## Possible Causes

- The metric name contains typos or incorrect format
- The metric may not be available on this host
- The metric collection may not be enabled

## Resolving Metric Issues

- Check for typos in the metric name
- Verify the metric is available on this host type`;

    return {
      content: [{ type: "text", text: errorMessage }],
      isError: true,
    };
  }
});
```

IDより人間が読める名前を使う

問題点

LLMは難解な識別子の処理が苦手

- UUID: `usr_1234567890`
- 英数字ID: `abc123xyz`

推奨されるアプローチ

良い例

```
search_user_by_name("Alice");
get_product_by_name("iPhone 15 Pro");
```

悪い例

```
get_user_by_id("usr_1234567890");
get_product_by_id("prod_abc123");
```

人間が読める名前のメリット

- ▶ LLMの理解精度が向上
- ▶ 幻覚(ハルシネーション)の軽減
- ▶ 検索タスクの精度向上

実装のポイント

内部的にIDに変換する処理を実装

まとめ: MCP基礎知識

- MCPはツールのインターフェースを標準化
- クライアント・サーバーモデル
- JSON-RPC 2.0で通信
- リソース・プロンプト・ツールの3つの機能

まとめ: 実装のポイント

- TypeScript SDKで簡単に実装可能
- MCP Inspectorでデバッグ
- Claude Desktopで実際に使用

まとめ: 本番レベルの実装

- タスクベースで設計
- コンテキストサイズに注意
- `description`をプロンプトエンジニアリング
- エラー応答を詳細に
- 人間が読める名前を使用

参考資料

- ▶ Model Context Protocol公式ドキュメント
<https://modelcontextprotocol.io/>
- ▶ TypeScript SDK
<https://github.com/modelcontextprotocol/typescript-sdk>
- ▶ Writing Tools for Agents
<https://www.anthropic.com/engineering/writing-tools-for-agents>
- ▶ The second wave of MCP: Building for LLMs, not developers
<https://vercel.com/blog/the-second-wave-of-mcp-building-for-langs-not-developers>
- ▶ やさしい MCP 入門
<https://www.shuwasystem.co.jp/book/9784798075730.html>

ご清聴ありがとうございました