cURL 请求到树状 JSON 转换工具开发文档

背景与目标

在日常接口调试与测试场景中，研发与测试同学经常使用 Charles 等抓包工具捕获网络请求，并将其导出为 cURL 命令格式。这些 cURL 命令包含了完整的请求信息（URL、请求方法、请求头、请求体等），是复现与分析问题的宝贵素材。然而，当我们需要对这些请求返回的 JSON 数据进行特定处理，例如，抽取其中具有层级关系的特定字段（如用例、目录、组织架构等）时，手动操作既繁琐又易出错。

本项目旨在开发一个 Go 语言编写的命令行工具，自动化完成“cURL → API 请求 → 结构化 JSON → 树状精简 JSON”的完整处理链路。

- **输入**：一个从 Charles 或其他工具复制的 cURL 命令。
- **输出**：一个仅保留核心节点内容（如 `case_title`）与层级结构的 JSON 文件。

整个处理流程如下：

1. **解析 cURL**：从输入的 cURL 命令字符串中，精确解析出请求方法 (Method)、目标 URL、请求头 (Headers) 和请求体 (Body)。
2. **执行 HTTP 请求**：使用解析出的参数，模拟原始请求，向目标服务器发起真实的 HTTP 调用。
3. **校验响应**：检查 HTTP 响应状态码，确保请求成功 (2xx)，并验证响应内容为有效的 JSON 格式。
4. **抽取树状结构**：递归遍历响应的 JSON 数据，根据预设的规则（或用户指定的字段名），提取出表达层级关系的内容字段和子节点数组。
5. **输出文件**：将抽取的树状精简 JSON 序列化，并保存到本地文件中。

这个工具将帮助用户一键完成从原始 cURL 命令到目标精简 JSON 的转换，极大提升处理特定嵌套结构数据的效率。

约束与边界

为确保工具的健壮性与实用性，我们定义了以下约束与边界条件：

- **cURL 命令覆盖范围**：
  - 支持常见的 cURL 参数，包括：
    - `-X` / `--request`：指定请求方法（GET, POST, PUT, DELETE 等）。
    - `-H` / `--header`：设置请求头，支持多个 -H 参数。
    - `--data` / `--data-raw`：定义 HTTP POST 请求的请求体。
    - `-G` / `--get`：将 --data 中的数据通过 URL 查询参数发送。

        - URL 中直接包含的查询字符串。
    - 工具将**忽略**与请求结构无关的 cURL 开关，如 `--compressed`，`-i`，`-v` 等，因为我们的目标是复现请求本身，而非 cURL 客户端的特定行为。
- **响应格式要求**：
    - HTTP 响应的 `Content-Type` 应为 `application/json` 或其他可被解析为 JSON 的类型。
    - 响应体必须是结构良好 (Well-formed) 的 JSON 文本。非 JSON 响应（如 HTML 页面、纯文本）将导致错误。
- **网络与错误处理策略**：
    - **网络错误**：对于 DNS 解析失败、TCP 连接超时、TLS 握手失败等网络层面的问题，工具将直接报错并退出，同时提供明确的错误信息。
    - **非 2xx 状态码**：
        - 当服务器返回非 2xx（如 4xx 客户端错误、5xx 服务器错误）的状态码时，工具默认会视为请求失败。
        - 它将打印状态码和完整的响应体（如果存在），以便用户排查问题（例如，权限不足 `401/403`，或请求参数错误 `400`）。
        - 程序将不会继续执行 JSON 抽取流程。

输入输出定义

为了提供灵活的使用方式，工具支持多种输入来源和标准的输出格式。

输入方式

用户可以通过以下三种方式提供请求信息：

1. **从标准输入 (stdin) 读取 cURL**：
    - 这是最便捷的方式，用户可以直接将复制的 cURL 命令粘贴到终端中。
    - `cat curl.txt | ./tool`
2. **从文件读取 cURL**：
    - 用户可以将 cURL 命令保存为一个文本文件，通过命令行参数指定该文件。
    - `./tool --curl-file /path/to/curl.txt`
3. **通过命令行参数直接传入**：
    - 对于不方便使用 cURL 的场景，用户也可以直接提供请求的核心要素。
    - `./tool --url "http://example.com/api" --header "Authorization: Bearer XXX" --data '{"key": "value"}'`

输出格式与规范

- **核心要求**：输出的 JSON 文件**仅保留**树状层级关系与节点内容两个关键信息。
  - **节点内容键名**：统一强制为 `case_title`。无论原始数据中的字段是 `title`，`name`，还是 `label`，在最终输出时都会被重命名为 `case_title`。
  - **子节点键名**：统一强制为 `children`。无论原始数据中的子节点数组键是 `nodes`，`items`，`sub_items` 还是其他，最终输出时都会被重命名为 `children`。
- **文件路径与命名**：
  - 用户可以通过 `--out` 参数指定输出文件的完整路径和名称。
  - 如果未指定，建议工具默认在当前工作目录下生成文件，并采用一个有意义的命名规则，例如：`output_{timestamp}.json` 或基于 URL 派生，如 `api_v1_projects_cases_{timestamp}.json`。

一个典型的输出示例如下：

代码块

```
{
  "case_title": "顶层节点A",
  "children": [
    {
      "case_title": "子节点A1",
      "children": []
    },
    {
      "case_title": "子节点A2",
      "children": [
        {
          "case_title": "孙子节点A2a",
          "children": []
        }
      ]
    }
  ]
}

```

流程设计与组件划分

为了实现清晰、可维护的代码结构，我们将工具的核心功能划分为以下几个独立的模块。

模块化说明

1. **cURL 解析器 (Curl Parser)**

- **职责**：接收 cURL 命令字符串，将其解析为一个结构化的 Go 对象，包含 `Method`，`URL`，`Headers`，`Body`。
- **实现**：通过字符串分割和正则表达式匹配，识别 `-X`，`-H`，`--data` 等关键参数及其对应的值。需要处理各种参数的简写形式和组合情况。

2. **HTTP 执行器 (HTTP Executor)**
   - **职责**：接收解析后的请求对象，使用 Go 的 `net/http` 标准库，构造并发送 HTTP 请求。
   - **实现**：创建一个 `http.Request` 对象，设置好方法、URL、所有请求头以及请求体，然后通过 `http.Client` 发送该请求。
3. **响应校验器 (Response Validator)**
   - **职责**：检查 HTTP 执行器返回的 `http.Response`。
   - **实现**：首先检查 `error` 是否为 `nil`，然后判断 `response.StatusCode` 是否在 200-299 的范围内。最后，尝试读取响应体，并初步验证其是否为合法的 JSON 格式。
4. **树抽取器 (Tree Extractor)**
   - **职责**：这是工具的核心。它接收原始的、已解码的 JSON 数据（通常是 `map[string]interface{}` 或 `[]interface{}`），并根据规则递归地抽取出目标树状结构。
   - **实现**：采用深度优先搜索（DFS）或广度优先搜索（BFS）策略。对于每个 JSON 对象，查找用户指定的“节点内容”字段和“子节点数组”字段。如果找到，则构建新的精简对象，并对子节点数组中的每个元素递归调用自身。
5. **序列化与落盘 (Serializer & Writer)**
   - **职责**：将抽取器生成的结果（一个 Go 结构体或 `map`）序列化为格式化的 JSON 字符串，并写入到指定的文件中。
   - **实现**：使用 `encoding/json` 包的 `json.MarshalIndent` 来生成带缩进的美观 JSON。使用 `os.WriteFile` 将结果以 `UTF-8` 编码（无 BOM）写入磁盘。

数据流图（文字说明）

1. **开始**：用户通过 CLI 提供 cURL 字符串或请求参数。
2. cURL 字符串进入 **cURL 解析器**，输出一个结构化的 `Request` 对象。
3. `Request` 对象被传递给 **HTTP 执行器**，后者发出网络请求，并返回一个 `Response` 对象和一个 `error`。
4. `Response` 和 `error` 进入 **响应校验器**。如果校验失败，流程终止并向用户报告错误。如果成功，则将响应体（`[]byte`）传递给下一步。
5. 响应体字节流被解码为通用的 `interface{}` 类型，然后送入 **树抽取器**。
6. **树抽取器**递归处理数据，生成一个符合输出规范的、精简的树状 `interface{}`。

7. 最后，这个精简的树状 `interface{}` 被送入 **序列化与落盘** 模块，该模块将其转换为格式化的 JSON 字符串，并保存到文件中。
8. **结束**：文件成功保存，程序正常退出。

CLI 规范与示例

工具将提供一组清晰的命令行参数，以支持不同的使用场景。

参数约定

- `--from-curl` (string):直接从命令行接收一个 cURL 字符串。为避免 shell 转义问题，建议用单引号包裹。
- `--curl-file` (string):从指定的文本文件路径读取 cURL 命令。
- `--url` (string):直接指定请求的 URL。当不使用 cURL 输入时，此参数为必需。
- `--header` (string slice):直接指定请求头，格式为 `"Key: Value"`。可多次使用以添加多个头。
- `--data` (string):直接指定请求体。
- `--method` (string):指定请求方法，默认为 `GET`。当使用 `--data` 时，通常需要设为 `POST` 或 `PUT`。
- `--out` (string):指定输出 JSON 文件的路径。如果留空，则默认输出到 `output_{timestamp}.json`。
- `--title-key` (string slice):指定节点内容的候选键名，按优先级顺序排列。默认值为 `case_title,title,name`。工具会依次查找这些键，使用第一个找到的值。
- `--children-keys` (string slice):指定子节点数组的候选键名，按优先级顺序排列。默认值为 `children,nodes,sub_cases,items,data`。

用法演示

1. 典型用法：从粘贴板读取 cURL 并抽取

代码块

```
# 直接将 cURL 命令粘贴到终端
./extractor --from-curl 'curl "http://example.com/api/cases/1" -H "Authorization: Bearer mytoken"'

```

> 工具将请求 `http://example.com/api/cases/1`，然后在当前目录生成 `output_xxxxxxxx.json`。

2. 从文件读取 cURL，并自定义输出路径与抽取规则

```
代码块
1  # my_curl.txt 文件内容: curl -X POST 'http://api.service.com/v2/tree' --data
   '{"id": 100}'
2  ./extractor --curl-file my_curl.txt \
3              --out /tmp/result.json \
4              --title-key "nodeName,label" \
5              --children-keys "subNodes"
6
```

> 工具将读取 `my_curl.txt`，发起 POST 请求，并从响应中寻找 `nodeName` 或 `label` 作为标题，`subNodes` 作为子节点数组，最终结果保存到 `/tmp/result.json`。

3. 非 cURL 模式：手动指定参数

```
代码块
1  ./extractor --url "https://another.api/resource" \
2              --header "X-API-Key: somekey" \
3              --header "Content-Type: application/json" \
4              --method POST \
5              --data '{"filter": "active"}'
6
```

错误场景提示

- **cURL 解析失败**：`Error: Failed to parse cURL command. Please check the syntax.`
- **网络请求失败**：`Error: HTTP request failed: Get "http://example.com": dial tcp: lookup example.com: no such host`
- **服务器返回非 2xx 状态码**：`Error: Received non-2xx status code: 401 Unauthorized. Response body: {"error": "token expired"}`
- **响应非 JSON**：`Error: Response body is not valid JSON.`

核心抽取规则

这是本工具的灵魂所在，决定了如何从一个庞大复杂的 JSON 中精确地提取出我们需要的“骨架”。

1. 仅保留节点内容字段

- **默认行为**：默认情况下，抽取器会寻找名为 `case_title` 的字段，并将其值作为节点的内容。
- **候选键机制**：考虑到不同 API 的设计差异，工具允许用户通过 `--title-key` 参数提供一个有序的候选键列表，例如 `title,name,label`。

    - 对于任意一个 JSON 对象，抽取器会按顺序检查 `title`、`name`、`label` 是否存在。
    - 它会使用**第一个**找到的非空字符串值作为节点内容。
- **统一输出键**：无论原始键名是什么，在最终生成的 JSON 中，该字段的键名**永远是** `case_title`。

2. 层级关系识别

- **默认行为**：默认在 JSON 对象中查找名为 `children` 的字段，并期望其值是一个数组，代表子节点列表。
- **候选键机制**：同样，用户可以通过 `--children-keys` 参数提供一个候选列表，如 `children,nodes,sub_cases,items`。
    - 抽取器会按顺序查找这些键，并使用第一个找到的数组作为子节点列表。
- **递归抽取**：一旦找到子节点数组，抽取器会对数组中的每一个元素（如果它是一个 JSON 对象）**递归地执行相同的抽取逻辑**。这个过程会一直持续，直到最深的层级，从而完美地保留任意嵌套结构的层级关系。
- **统一输出键**：无论原始子节点数组的键名是什么，在最终生成的 JSON 中，该数组的键名**永远是** `children`。

3. 噪声剔除

- 这是规则中最重要的一点：在抽取过程中，除了被识别为“节点内容”的字段和“子节点数组”的字段外，**其余所有字段都将被彻底丢弃**。
- 这确保了最终的输出是极致精简的，只包含 `case_title` 和 `children` 两个键。

4. 空与异常处理

- **空数组**：如果一个节点根据规则没有找到任何子节点数组，或者找到的子节点数组为空，那么在输出的 JSON 中，它的 `children` 字段将是一个空数组 `[]`。
- **缺失字段**：如果一个 JSON 对象既没有找到任何“节点内容”候选键，也没有找到任何“子节点数组”候选键，那么这个对象将被视为空节点，在最终的树中可能会被忽略或表示为一个空对象 `{}`，具体策略可配置。
- **循环引用防护**：在递归遍历过程中，需要记录已访问过的节点（可通过指针或唯一 ID）。如果检测到对已访问节点的重复处理，应立即中断当前分支的递归，以防止无限循环。
- **超深嵌套上限**：为了防止因恶意或错误的输入数据导致栈溢出，应设定一个合理的递归深度上限（例如 100 层）。超过此深度，将停止继续向下抽取，并可能在日志中记录一个警告。

基本实现思路与代码片段 (Go)

以下是一些关键功能的 Go 语言实现思路与示例代码。

将 cURL 切分为 Method, URL, Headers, Body

我们可以使用标准库 `strings` 和 `regexp` 来解析。一个简化的思路是：首先按空格分割命令，然后遍历这些片段，根据前缀（如 `-X`，`-H`）或特征来识别并提取相应的值。

代码块

```
package main

import (
	"fmt"
	"strings"
)

type RequestInfo struct {
	Method  string
	URL     string
	Headers map[string]string
	Body    string
}

// A simplified parser. A robust solution should use a proper library or more complex logic.
func ParseCurl(curlCmd string) (*RequestInfo, error) {
	info := &RequestInfo{
		Method:  "GET", // Default method
		Headers: make(map[string]string),
	}
	parts := strings.Fields(curlCmd)
	if len(parts) < 2 {
		return nil, fmt.Errorf("invalid curl command")
	}

	// Assuming 'curl' is the first part
	for i := 1; i < len(parts); i++ {
		part := parts[i]
		switch part {
		case "-X", "--request":
			if i+1 < len(parts) {
				info.Method = strings.ToUpper(parts[i+1])
				i++
			}
		case "-H", "--header":
			if i+1 < len(parts) {
				header := strings.Trim(parts[i+1], "'")
				kv := strings.SplitN(header, ":", 2)
				if len(kv) == 2 {
					info.Headers[strings.TrimSpace(kv[0])] = strings.TrimSpace(kv[1])
```

```
                                }
                                i++
                        }
                case "--data", "--data-raw":
                        if i+1 < len(parts) {
                                info.Body = strings.Trim(parts[i+1], "'")
                                if info.Method == "GET" {
                                        info.Method = "POST" // Default to POST if data is present
                                }
                                i++
                        }
                default:
                        // Assume the first part that looks like a URL is the URL
                        if info.URL == "" && (strings.HasPrefix(part, "http") || strings.HasPrefix(part, "'http")) {
                                info.URL = strings.Trim(part, "'")
                        }
                }
        }

        if info.URL == "" {
                return nil, fmt.Errorf("URL not found in curl command")
        }

        return info, nil
}

```

使用 net/http 发起请求与读取响应

代码块

```
package main

import (
        "fmt"
        "io"
        "net/http"
        "strings"
        "time"
)

func ExecuteRequest(info *RequestInfo) ([]byte, error) {
```

```
	req, err := http.NewRequest(info.Method, info.URL, strings.NewReader(info.Body))
	if err != nil {
		return nil, fmt.Errorf("failed to create request: %w", err)
	}

	for key, value := range info.Headers {
		req.Header.Set(key, value)
	}

	client := &http.Client{Timeout: 30 * time.Second}
	resp, err := client.Do(req)
	if err != nil {
		return nil, fmt.Errorf("failed to execute request: %w", err)
	}
	defer resp.Body.Close()

	if resp.StatusCode < 200 || resp.StatusCode >= 300 {
		bodyBytes, _ := io.ReadAll(resp.Body)
		return nil, fmt.Errorf("received non-2xx status: %s. Body: %s", resp.Status, string(bodyBytes))
	}

	return io.ReadAll(resp.Body)
}

```

递归遍历任意 JSON，抽取 `case_title` 与 `children`

这是最核心的部分。我们定义一个递归函数来处理 `interface{}` 类型的数据。

代码块

```
package main

import (
	"encoding/json"
	"fmt"
)

type ExtractionConfig struct {
	TitleKeys    []string
	ChildrenKeys []string
}

// SimplifiedNode represents the output structure.
type SimplifiedNode struct {
```

```
        CaseTitle string            `json:"case_title"`
        Children  []*SimplifiedNode `json:"children"`
}

func ExtractTree(data interface{}, config *ExtractionConfig) *SimplifiedNode {
        obj, ok := data.(map[string]interface{})
        if !ok {
                return nil // Not an object, cannot extract
        }

        node := &SimplifiedNode{
                Children: []*SimplifiedNode{},
        }

        // 1. Find title
        for _, key := range config.TitleKeys {
                if title, ok := obj[key].(string); ok {
                        node.CaseTitle = title
                        break
                }
        }

        // 2. Find children and recurse
        for _, key := range config.ChildrenKeys {
                if children, ok := obj[key].([]interface{}); ok {
                        for _, childData := range children {
                                if childNode := ExtractTree(childData, config); childNode != nil {
                                        node.Children = append(node.Children, childNode)
                                }
                        }
                        break // Found first matching children key
                }
        }

        // Only return node if it has a title or children to avoid empty nodes
        if node.CaseTitle != "" || len(node.Children) > 0 {
                return node
        }

        return nil
}

// Example usage
func ProcessResponse(jsonData []byte, titleKeys, childrenKeys []string) ([]byte, error) {
```

```
    var data interface{}
    if err := json.Unmarshal(jsonData, &data); err != nil {
        return nil, fmt.Errorf("invalid JSON: %w", err)
    }

    config := &ExtractionConfig{
        TitleKeys:    titleKeys,
        ChildrenKeys: childrenKeys,
    }

    // The root could be an object or an array of objects
    if rootArray, ok := data.([]interface{}); ok {
        var results []*SimplifiedNode
        for _, item := range rootArray {
            if node := ExtractTree(item, config); node != nil {
                results = append(results, node)
            }
        }
        return json.MarshalIndent(results, "", "  ")
    }

    rootNode := ExtractTree(data, config)
    if rootNode == nil {
        // Maybe the actual data is nested inside a top-level key like "data"
        if rootMap, ok := data.(map[string]interface{}); ok {
            for _, v := range rootMap { // Try to find the first valid tree
                if node := ExtractTree(v, config); node != nil {
                    return json.MarshalIndent(node, "", "  ")
                }
            }
        }
            return nil, fmt.Errorf("no valid tree structure found")
    }

    return json.MarshalIndent(rootNode, "", "  ")
}

```

将结果写入文件（UTF-8，无 BOM）

`os.WriteFile` 默认就满足我们的需求。

代码块

```
package main

import (
```

```
        "fmt"
        "os"
)

func WriteOutput(filePath string, data []byte) error {
        err := os.WriteFile(filePath, data, 0644)
        if err != nil {
                return fmt.Errorf("failed to write to file %s: %w", filePath,
err)
        }
        fmt.Printf("Successfully wrote simplified JSON to %s\n", filePath)
        return nil
}

```

日志与错误处理

健壮的日志和错误处理机制是保证工具可用性的关键。

- **分级日志**：
  - **INFO**: 用于输出关键的流程性信息，帮助用户了解当前进度。例如：`"Parsing cURL command..."`，`"Executing HTTP request to http://example.com..."`，`"Extracting tree structure..."`，`"Writing output to result.json..."`。
  - **WARN**: 用于报告一些非致命但可能影响结果的异常情况。例如：`"Warning: Recursion depth limit (100) reached. Sub-tree may be incomplete."`，`"Warning: Detected a potential circular reference, skipping node."`。
  - **ERROR**: 用于报告导致程序终止的严重错误。所有 `error` 类型的返回值都应被打印到 `stderr`，并附带清晰的上下文信息，例如：`"Error: Failed to connect to host: dial tcp: i/o timeout"`。
- **重试建议**：
  - 对于某些瞬时的网络错误（如 `connection reset by peer`）或特定的服务器状态码（如 `503 Service Unavailable`），可以在错误信息中向用户提供重试建议。
  - 工具本身**不应**内置自动重试逻辑，因为这可能会在不知情的情况下对服务器造成压力。将重试的决定权交给用户。
- **常见错误诊断**：
  - **DNS/连接错误**：明确提示用户检查网络连接、防火墙设置或目标域名是否正确。
  - **TLS 握手失败**：提示可能是代理问题、证书问题或网络中间人攻击。
  - **401/403 (Unauthorized/Forbidden)**：提示用户检查其请求头中的 `Authorization`、`Cookie` 或 `API Key` 是否正确、有效且未过期。

- **超时 (Timeout)**：提示用户目标服务器响应缓慢，可以尝试通过一个（可选的）`--timeout` 参数来延长等待时间。

安全与合规

在处理可能包含敏感信息的 cURL 命令时，必须遵守基本的安全规范。

- **避免在日志中打印敏感头**：
  - 在打印调试信息或错误报告时，必须对已知的敏感请求头（如 `Authorization`，`Cookie`，`X-Api-Key` 等）的值进行脱敏处理。
  - 可以实现一个过滤器，在记录日志时将这些头的值替换为 `[REDACTED]`。
- **本地仅保存必要输出**：
  - 工具处理的原始 cURL 和完整的服务器响应不应被持久化存储到本地，除非用户明确要求（例如通过调试标志）。
  - 默认情况下，只有最终的、经过精简的树状 JSON 才会被写入文件。
- **重放风险提示**：
  - 在工具的帮助文档或首次使用的提示中，应明确告知用户：此工具会**实际执行** cURL 命令中包含的 HTTP 请求。
  - 如果 cURL 是一个执行创建、修改或删除操作的非幂等请求（如 `POST /users`），重复运行此工具将导致该操作被重复执行。用户需要自行评估并承担此风险。

性能与扩展

对于未来的迭代，可以考虑以下性能优化和扩展性设计。

- **大 JSON 的流式解析**：
  - 对于非常大的 JSON 响应（例如超过 100MB），一次性将其读入内存并用 `encoding/json.Unmarshal` 解析可能会消耗大量内存。
  - 可以考虑使用 `encoding/json.Decoder` 进行流式解析。通过 `decoder.Token()` 方法逐个读取 JSON 的 token（`{`，`[`，`string`，`number` 等），可以只在内存中构建我们需要的精简树，而无需加载整个原始 JSON DOM。这对于内存受限的环境至关重要。
- **并发与速率限制**：
  - 虽然当前工具是单次执行模型，但未来若扩展为批量处理多个 cURL 的模式，则需要考虑并发。
  - 可以使用 Go 的 `goroutine` 和 `channel` 来并发执行多个 HTTP 请求。
  - 为了避免对目标服务器造成过大压力，应引入速率限制机制，例如使用 `golang.org/x/time/rate` 包来控制每秒的请求数量。
- **增加新的 Children 键或 Title 备选键**：

- 当前的设计已经通过 `--title-key` 和 `--children-keys` 参数提供了良好的扩展性。
- 未来可以考虑将其配置化，例如允许用户通过一个 YAML 或 JSON 配置文件来定义更复杂的抽取规则，以适应更多样化的 API 响应结构。

测试与验收

为保证工具的质量和稳定性，需要建立完善的测试体系。

单元测试建议

- **cURL 解析器**：
  - 测试各种合法的 cURL 命令，包括不同的方法、带/不带请求体、多个 Header、不同格式的 URL 等。
  - 测试边界情况，如不完整的命令、包含特殊字符的参数等。
- **响应抽取器**：
  - 这是测试的重点。构造多种结构的 mock JSON 数据。
  - 测试不同深度的嵌套、不同的 title/children 键名、节点内容为空、子节点数组为空或不存在等情况。
  - 测试包含循环引用的 JSON，确保程序不会崩溃。
  - 测试数组作为根节点的 JSON。
- **边界用例**：
  - 输入为空或格式错误的 cURL。
  - HTTP 响应体为空或非 JSON 格式。
  - 抽取的 title 或 children 字段类型不正确（例如，title 是一个数字，children 是一个对象）。

集成测试示例

- **使用 Mock Server**：
  - 在测试代码中启动一个临时的 `net/http/httptest.Server`。
  - 该服务器根据不同的请求路径返回预设好的 JSON 响应或错误状态码。
  - 集成测试将覆盖从 cURL 解析到文件写入的完整流程，验证各组件协同工作是否正常。
- **使用本地 JSON 文件**：
  - 可以跳过 HTTP 请求阶段，直接从本地文件读取 JSON，测试抽取和写入逻辑。

验收清单

1. **功能覆盖度**：
   - [ ] 支持从 stdin、文件、命令行参数三种输入方式。

- [ ] 正确解析常见的 cURL 参数。
- [ ] 能够处理 GET, POST 等多种请求方法。
- [ ] 能够正确处理网络错误和非 2xx 响应。
- [ ] 能够根据 `--title-key` 和 `--children-keys` 规则正确抽取树状结构。
- [ ] 输出的 JSON 严格遵守 `case_title` 和 `children` 的键名规范。
- [ ] 能够将结果正确写入指定或默认的文件路径。

2. **失败回退策略**：
    - 在任何步骤失败时，程序应以非零状态码退出，并打印有意义的错误信息。
    - 不应产生部分或损坏的输出文件。只在所有步骤成功后，才进行文件写入操作。

使用示例

结合一个具体的请求和响应，展示工具的实际效果。

示例场景

假设我们用 Charles 抓到了一个获取测试用例信息的请求，其 cURL 如下：`curl 'http://my-test-platform.com/v1/api/projects/143901777/test_case/42?minder_id=abcd' -H 'Authorization: Bearer xyz'`

服务器返回的响应（部分）如下：

代码块

```
{
  "errCode": 0,
  "message": "success",
  "data": {
    "value": {
      "case_id": 42,
      "case_title": "Workspace 操作",
      "case_mind": "XXX",
      "minder_info": {
        "title": "新功能",
        "priority": -1,
        "tags": ["t1"],
        "items": [
          {
            "title": "创建文档",
            "tags": [],
            "items": []
          },
          {
```

```
            "title": "分享链接",
            "tags": [],
            "items": []
          }
        ]
      }
    }
  }
}

```

执行命令

我们希望将 `case_title` 和 `minder_info` 中的 `title` 作为节点内容，将 `minder_info` 下的 `items` 作为子节点。因此，我们可以这样执行：

代码块

```
./extractor \
  --from-curl "curl 'http://my-test-platform.com/v1/api/projects/143901777/test_case/42?minder_id=abcd' -H 'Authorization: Bearer xyz'" \
  --title-key "case_title,title" \
  --children-keys "items" \
  --out workspace_case.json

```

示例抽取输出

工具会智能地从 `data.value` 开始发现第一个 `case_title`，然后深入到 `minder_info` 找到 `title` 和 `items`，并递归下去。最终生成的 `workspace_case.json` 文件内容如下：

代码块

```
{
  "case_title": "Workspace 操作",
  "children": [
    {
      "case_title": "新功能",
      "children": [
        {
          "case_title": "创建文档",
          "children": []
        },
        {
          "case_title": "分享链接",
```

```
            "children": []
          }
        ]
      }
    ]
  }

```

解释：

- 根节点的 `case_title` 来自原始 JSON 的 `data.value.case_title`。
- `data.value` 中没有 `items` 字段，但其内部的 `minder_info` 既有 `title` 又有 `items`。工具的递归逻辑会深入探索，并将 `minder_info` 视为一个子节点。
- 最终的输出完全符合我们的要求：一个干净、仅包含层级和标题的树状结构。如果任何节点下没有子节点数组，`children` 会是一个空数组 `[]`。

落地与维护

- **打包为单文件二进制**：
  - 使用 `go build -ldflags="-s -w"` 命令可以编译出一个体积较小的、静态链接的二进制文件。
  - 这使得工具分发和使用极其方便，用户只需下载一个文件即可在不同平台（Linux, macOS, Windows）上运行，无需安装任何依赖。
- **版本约定**：
  - 建议遵循语义化版本（Semantic Versioning）规范（`MAJOR.MINOR.PATCH`）。
  - `MAJOR` 版本更新：当发生不兼容的 CLI 参数变更或输出格式变更时。
  - `MINOR` 版本更新：当增加新功能（如支持新的 cURL 参数）且保持向后兼容时。
  - `PATCH` 版本更新：当修复 bug 且保持向后兼容时。
- **配置项变更规范**：
  - 当需要修改默认的候选键列表（`--title-key`，`--children-keys`）时，应优先通过增加新的默认值来扩展，而不是删除或修改现有值，以保证对老用户的兼容性。
  - 任何涉及默认行为变更的修改，都应在版本的 `CHANGELOG.md` 文件中明确记录。