Obsidian Tags-in-One-Place Plugin - 产品需求文档 (PRD)

版本: 1.0
日期: 2026年1月31日
作者: AI Product Manager
状态: 草案

---

1. 执行摘要

1.1 产品概述

Tags-in-One-Place 是一个 Obsidian 插件，能够自动收集 vault 中的所有标签（tags），并将它们写入指定的 Markdown 文件中，支持手动或自动更新。该插件解决了用户需要"一览所有标签"和"标签索引文档化"的核心需求。

1.2 目标用户

- 重度使用标签（tags）进行笔记分类和管理的 Obsidian 用户
- 需要维护标签索引文档（如 `Tags.md` 或 `Index/Tags.md`）的用户
- 希望将标签列表作为"知识库地图"一部分的用户
- 需要定期审查和清理标签系统的用户

1.3 核心价值

- **可见性**：将分散在各个笔记中的标签集中展示在一个文档中
- **可更新性**：一键刷新标签列表，保持索引文档的实时性
- **可配置性**：支持自定义目标文件路径、标签格式、排序规则、过滤条件
- **可扩展性**：为未来功能（如标签统计、层级展示、自动更新）预留架构空间

---

2. 市场与竞品分析

2.1 现有解决方案对比

方案	优点	缺点	适用场景
Tag Page 插件[3][7]	为每个标签生成独立页面，自动更新	生成多个文件，不是"一个文件列出所有标签"	需要标签详情页的场景
QuickAdd 脚本[11][15][16]	轻量、快速部署、无需打包	配置不友好（硬编码），错误处理粗糙，无 UI 界面	临时、简单、手动触发的需求
Dataview 查询	强大的查询能力，实时动态	需要实时渲染，不生成可导出的 Markdown	仅需查看、不需文档化
手动维护	完全控制	劳动密集，容易过时	标签数量少且稳定

2.2 市场缺口

- **核心需求未被满足**：目前没有主流插件专门做"将所有标签写入一个 Markdown 文件并可更新"[6][20]
- **用户痛点**：用户需要在"标签页插件"（生成多个文件）和"手动维护"之间妥协
- **插件优势**：相比 QuickAdd 脚本，插件提供更好的用户体验（设置面板、命令面板、事件监听、错误处理）[28][31][34]

---

3. 产品目标与范围

3.1 MVP (最小可行产品) 功能

核心功能

1. **标签收集**：使用 `getAllTags()` API 遍历 vault，收集所有标签[4][5][17]
2. **文件写入**：将标签列表写入指定的 Markdown 文件（默认 Tags.md）[37][38][44]
3. **手动更新**：提供命令面板命令（如 "Update tag index"）触发更新[31][43][49]
4. **基础配置**：支持配置目标文件路径（设置面板）[28][31][34]

非功能性需求

- **性能**：大型 vault（10,000+ 笔记）下，标签收集和写入应在 3 秒内完成[6][27]
- **健壮性**：处理文件不存在、权限错误、并发写入等边缘情况[37][38]
- **用户体验**：操作反馈（成功/失败通知）、错误提示清晰[31]

3.2 未来功能路线图（Post-MVP）

Phase 2: 增强配置

- **标签格式**：支持列表、表格、分组等多种展示格式
- **排序选项**：字母序、使用频率、最近使用时间
- **过滤规则**：排除特定标签、只包含特定前缀的标签
- **层级展示**：支持嵌套标签（如 #project/work）的层级结构显示

Phase 3: 自动化

- **自动更新**：监听 vault 事件（文件保存、标签修改），自动刷新标签索引[41][42][45]
- **定时更新**：支持设置更新频率（如每 5 分钟）
- **增量更新**：仅在标签发生变化时更新，避免不必要的文件写入

Phase 4: 高级功能

- **标签统计**：显示每个标签的使用次数、关联笔记数
- **标签关系图**：可视化标签共现关系
- **标签建议**：基于现有标签，建议新标签或合并相似标签
- **多文件支持**：支持生成多个标签索引（按文件夹、按类型等）

3.3 明确排除的功能

- ✕ **标签内容编辑**：不支持在索引文件中直接编辑标签（应使用 Tag Wrangler 等专业工具）[33]
- ✕ **标签重命名/删除**：不提供批量标签管理功能（有专门插件如 Tag Wrangler）[29][33]
- ✕ **Dataview 集成**：MVP 不依赖 Dataview，保持轻量

---

4. 技术规格

4.1 核心 API 使用

标签收集

```
import { getAllTags } from 'obsidian';

// 方法 1：遍历所有文件，使用 getAllTags
const files = this.app.vault.getMarkdownFiles();
const tagSet = new Set<string>();

for (const file of files) {
const cache = this.app.metadataCache.getFileCache(file);
if (cache) {
const tags = getAllTags(cache); // 返回 string[] | null
if (tags) {
tags.forEach(tag => tagSet.add(tag));
}
}
}

const allTags = Array.from(tagSet).sort();
```

关键点：

- `getAllTags(cache)` 需要传入 `CachedMetadata`，不是 `MetadataCache`[17][30]
- 必须遍历每个文件的缓存，没有全局 API 直接返回所有标签[6][20]
- 使用 `Set` 去重，避免重复标签[4][5]

文件写入

```
import { TFile } from 'obsidian';

// 检查文件是否存在，不存在则创建
const targetPath = this.settings.targetFilePath;
let file = this.app.vault.getAbstractFileByPath(targetPath);

if (!file) {
file = await this.app.vault.create(targetPath, '');
}

if (file instanceof TFile) {
// 生成标签内容
const content = this.generateTagIndexContent(allTags);

// 写入文件
await this.app.vault.modify(file, content);
}
```

关键点:

- 使用 `vault.modify()` 写入文件内容[37][38][44]
- 使用 `vault.create()` 创建不存在的文件[38]
- 必须检查文件类型（`instanceof TFile`）[5]

4.2 设置面板实现

```
import { App, PluginSettingTab, Setting } from 'obsidian';

class TagIndexSettingTab extends PluginSettingTab {
plugin: TagIndexPlugin;

constructor(app: App, plugin: TagIndexPlugin) {
super(app, plugin);
this.plugin = plugin;
}

display(): void {
const { containerEl } = this;
containerEl.empty();

new Setting(containerEl)
  .setName('Target file path')
  .setDesc('Path to the tag index file (e.g., Tags.md or Index/Tags.md)')
  .addText(text => text
    .setPlaceholder('Tags.md')
```

```
      .setValue(this.plugin.settings.targetFilePath)
      .onChange(async (value) => {
        this.plugin.settings.targetFilePath = value;
        await this.plugin.saveSettings();
      })
    );
```

}
}

关键点:

- 使用 PluginSettingTab 创建设置面板[28][31][34]
- 每次修改后调用 saveSettings() 持久化配置[31][34]
- 使用 containerEl.empty() 清空容器，支持动态刷新[28]

4.3 命令注册

import { Plugin, Notice } from 'obsidian';

export default class TagIndexPlugin extends Plugin {
async onload() {
this.addCommand({
id: 'update-tag-index',
name: 'Update tag index',
callback: async () => {
try {
await this.updateTagIndex();
new Notice('Tag index updated successfully!');
} catch (error) {
new Notice(Failed to update tag index: ${error.message});
console.error('Tag index update error:', error);
}
}
});
}
}

关键点:

- 使用 addCommand() 注册命令面板命令[43][49]
- 使用 Notice 提供用户反馈[31]
- 完整的错误处理和日志记录

4.4 事件监听（Post-MVP）

```
// 监听文件修改事件
this.registerEvent(
this.app.vault.on('modify', (file) => {
if (file instanceof TFile) {
// 检查是否为 Markdown 文件
// 标记需要更新标签索引
this.scheduleUpdate();
}
})
);

// 防抖更新（避免频繁写入）
private scheduleUpdate() {
if (this.updateTimer) {
clearTimeout(this.updateTimer);
}
this.updateTimer = setTimeout(() => {
this.updateTagIndex();
}, 3000); // 3 秒后更新
}
```

关键点:

- 使用 registerEvent() 确保事件在插件卸载时自动注销[41][42]
- 使用防抖（debounce）避免频繁触发更新[42][48]
- 支持的事件：vault.on('modify'), vault.on('create'), vault.on('delete'), vault.on('rename')[38][42]

---

5. 用户体验设计

5.1 用户工作流

场景 1: 初次使用

1. 用户安装插件
2. 打开设置面板，配置目标文件路径（默认 Tags.md）
3. 打开命令面板（Ctrl/Cmd+P），输入 "Update tag index"
4. 插件生成标签索引文件，显示成功通知
5. 用户在文件浏览器中找到 Tags.md，查看所有标签

场景 2: 日常更新

1. 用户在笔记中添加了新标签
2. 打开命令面板，运行 "Update tag index"（或配置快捷键）
3. 插件更新索引文件，显示成功通知
4. 用户刷新 Tags.md，看到新标签

场景 3: 自动更新（Post-MVP）

1. 用户在设置中启用"自动更新"
2. 用户正常编辑笔记，添加/删除标签
3. 插件后台自动监听变化，每 3 秒更新一次索引
4. 用户打开 `Tags.md`，始终看到最新标签列表

5.2 默认输出格式（MVP）

Tag Index

Last updated: 2026-01-31 10:45 AM

All Tags (125)

- #archive
- #blog
- #book
- #idea
- #meeting
- #project
- #project/work
- #project/personal
- #todo
- #weekly-review

  ...

格式说明:

- 标题 + 更新时间戳
- 总数统计
- 字母顺序排序
- 列表格式（Markdown bullet list）

5.3 错误处理与用户反馈

场景	错误处理	用户反馈
目标文件路径不存在	自动创建文件（包括父文件夹）	Notice: "Created tag index file at {path}"
目标文件被占用/锁定	重试 3 次，失败后报错	Notice: "Failed to update tag index: file is locked"
Vault 为空（无笔记）	生成空标签列表	Notice: "Tag index updated (0 tags found)"
无权限写入文件	报错并记录日志	Notice: "Permission denied: {path}"
标签收集过程中断	捕获异常，部分更新	Notice: "Partial update completed (error: {error})"

---

6. 边缘情况处理

6.1 文件系统边缘情况

边缘情况	处理策略	技术实现
目标文件不存在	自动创建文件（含父文件夹）	`vault.create()` with recursive folder creation
目标文件被其他进程占用	重试 3 次（间隔 500ms），失败后报错	Try-catch with retry logic
目标路径为文件夹	报错："Target path is a folder, not a file"	Check `instanceof TFolder`
目标路径包含非法字符	报错："Invalid file path"	Path validation before write
磁盘空间不足	捕获异常，报错	Catch `ENOSPC` error

6.2 数据边缘情况

边缘情况	处理策略	技术实现
Vault 中无笔记	生成空标签列表（"No tags found"）	Check `files.length === 0`
所有笔记都无标签	生成空标签列表	Check tagSet.size === 0
标签格式错误（如 # 后有空格）	自动过滤掉非法标签	`getAllTags()` 已处理
标签包含特殊字符	保留原样（Obsidian 支持）	No special handling needed
重复标签（大小写不同）	视为不同标签（Obsidian 行为）	No deduplication[33]

6.3 并发与性能边缘情况

边缘情况	处理策略	技术实现
用户快速连续触发更新	防抖：取消前一次未完成的更新	Debounce with `clearTimeout()`
大型 vault（10,000+ 笔记）	显示进度通知："Collecting tags..."	Progress Notice + async processing
更新过程中用户关闭 Obsidian	确保数据完整性（使用 vault.process()）	Atomic write with `vault.process()`[37][44]
多个插件同时写入同一文件	使用 Obsidian 的文件锁机制	vault.modify() handles locking[37]

6.4 配置边缘情况

边缘情况	处理策略	技术实现
用户设置空路径	使用默认路径 `Tags.md`	Fallback to default in settings
用户设置路径指向已有笔记	警告："This will overwrite existing file"	Confirmation modal (Post-MVP)
用户更改路径后旧文件残留	不自动删除，提示用户手动清理	Notice: "Old index file at {old_path}"

7. 技术架构

7.1 插件结构

```
tags-in-one-place/
├── main.ts # 插件主文件
├── settings.ts # 设置定义与设置面板
├── tag-collector.ts # 标签收集逻辑
├── file-writer.ts # 文件写入逻辑
├── formatter.ts # 标签格式化输出
├── manifest.json # 插件元数据
├── versions.json # 版本兼容性
└── styles.css # 可选样式
```

7.2 核心类设计

```
// main.ts
export default class TagsInOnePlacePlugin extends Plugin {
settings: TagIndexSettings;
tagCollector: TagCollector;
fileWriter: FileWriter;

async onload() {
await this.loadSettings();
this.tagCollector = new TagCollector(this.app);
this.fileWriter = new FileWriter(this.app);

    this.addCommand({ /* ... */ });
    this.addSettingTab(new TagsInOnePlaceSettingTab(this.app, this));

}

async updateTagIndex() {
const tags = await this.tagCollector.collectAllTags();
const content = this.formatTagIndex(tags);
await this.fileWriter.writeToFile(this.settings.targetFilePath, content);
}
}

// tag-collector.ts
export class TagCollector {
constructor(private app: App) {}

async collectAllTags(): Promise<string[]> {
const files = this.app.vault.getMarkdownFiles();
const tagSet = new Set<string>();
```

```
  for (const file of files) {
    const cache = this.app.metadataCache.getFileCache(file);
    const tags = getAllTags(cache);
    if (tags) tags.forEach(tag => tagSet.add(tag));
  }

  return Array.from(tagSet).sort();
}
}

// file-writer.ts
export class FileWriter {
constructor(private app: App) {}

async writeToFile(path: string, content: string): Promise<void> {
let file = this.app.vault.getAbstractFileByPath(path);

  if (!file) {
    file = await this.app.vault.create(path, '');
  }

  if (file instanceof TFile) {
    await this.app.vault.modify(file, content);
  } else {
    throw new Error('Target path is not a file');
  }

}
}
```

7.3 依赖与兼容性

- **Obsidian API 版本**：最低 1.0.0（2022 年 10 月发布）
- **Node.js 版本**：16+（Obsidian 打包要求）
- **TypeScript 版本**：4.7+
- **无外部依赖**：仅使用 Obsidian 内置 API

---

8. 附录

8.1 参考资料

- Obsidian API 文档：https://docs.obsidian.md/
- getAllTags API 参考[17]
- Vault 文件操作 API[37][38]
- 插件设置面板实现[28][31][34]
- 事件监听最佳实践[41][42]

8.2 术语表

- **Vault**：Obsidian 中的笔记仓库（文件夹）
- **Tag**：Obsidian 中的标签（以 # 开头）
- **MetadataCache**：Obsidian 的元数据缓存系统
- **TFile**：Obsidian 中表示文件的类型
- **Command Palette**：命令面板（Ctrl/Cmd+P）

8.3 变更记录

版本	日期	变更内容
1.0	2026-01-31	初始版本，定义 MVP 范围和技术规格

文档结束