pc端解析kml文件 v1.3 260125

说明：在v1.2的基础上新增滤波函数确保，每个分段的点数不超过MAX_POINTS_PER_SEGMENT阈值+1。且大数组都使用malloc分配空间，尽可能使用堆空间，而不是使用栈空间，防止栈空间爆满，导致死机

代码

代码：

代码块

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <float.h>
#include <ctype.h>
#include <math.h>
#include <cctype>
//#include "KML_Parser.h"

/* ================== 一、系统配置与常量定义 ================== */

/* 读取缓冲区大小：
 * 4KB 是常见的磁盘块大小，能保证较高的 I/O 效率。
 * 如果内存极度紧张，可降至 512 字节，但会增加 SD 卡/Flash 的读写次数。
 */
#define READ_BUF_SIZE       4096

 /* LineString 内存缓存点数：
  * 为了避免每解析一个点就写一次文件（导致 I/O 瓶颈），
  * 我们在内存中积攒 128 个点（约 128 * 12 = 1.5KB）后再批量写入。
  */
#define LINE_CACHE_SIZE     128

  /* 缓冲区限制：防止恶意的超长标签或数字导致内存溢出 */
#define TAG_BUF_SIZE        64      // 足够容纳 "LineString", "coordinates" 等
#define NUM_BUF_SIZE        32      // 足够容纳高精度浮点数如 "-123.12345678"

#define MAX_CP_COUNT        64   //最大支持CP点个数
/* 判定重合的阈值（度），2e-4f约为 25-32m米 */
```

```c
#define SPLIT_THRESHOLD     2e-4f

#define  MIN_SEGMENT_COUNT 200 //每个分段的最小点数数量

#define MAX_POINTS_PER_SEGMENT 3000 //每个分段的最大点数数量

/* 临时文件名 */
#define FILE_POINT_NAME     "D:\\KT_Project\\OPEN_SOURCE\\new\\X-TRACK-main\\Software\\X-Track\\USER\\App\\Resource\\Track\\point_temp.bin"
#define FILE_LINE_NAME      "D:\\KT_Project\\OPEN_SOURCE\\new\\X-TRACK-main\\Software\\X-Track\\USER\\App\\Resource\\Track\\line_temp.bin"
#define FILE_LINE_FILTERED_NAME     "D:\\KT_Project\\OPEN_SOURCE\\new\\X-TRACK-main\\Software\\X-Track\\USER\\App\\Resource\\Track\\line_temp_filter.bin"
/* ================== 二、数据结构定义 == ================ */
typedef enum {
    KML_OK = 0,
    KML_ERR_FILE_OPEN,    // 文件打开失败
    KML_ERR_IO_READ,      // 读取错误
    KML_ERR_IO_WRITE,     // 写入错误（如磁盘满）
    KML_ERR_SEEK,         // 文件定位错误（fgetpos/fsetpos）
    KML_ERR_BAD_FORMAT,   // KML 格式严重错误
    KML_ERR_INTERNAL,      // 内部逻辑错误（如 malloc 失败）
    KML_ERR_NO_DATA      //无有效数据
} KmlError;


/* * 解析器状态机枚举
 * 分为三类：
 * 1. 语法动作状态：正在读标签名、属性等。
 * 2. 逻辑上下文状态：当前在 Point 还是 LineString 内部。
 * 3. 专用数据解析状态：处理 coordinates 数值流。
 */
typedef enum {
    // --- 动作状态 (Action States) ---
    STATE_IDLE,                 // 空闲，寻找下一个 '<'
    STATE_TAG_NAME,             // 正在读取标签名（如 "Point"）
    STATE_ATTR_SKIP,            // 正在跳过属性（如 id="123"），直到遇到 '>'
    STATE_TAG_END_NAME,         // 正在读取结束标签名（如 "Point" 在 </Point> 中）

    // --- 逻辑上下文 (Context States) ---
    STATE_CTX_ROOT,             // 根层级
    STATE_CTX_PLACEMARK,        // 在 <Placemark> ... </Placemark> 内部
    STATE_CTX_POINT,            // 在 <Point> ... </Point> 内部
    STATE_CTX_LINE,             // 在 <LineString> ... </LineString> 内部

    // --- 数据解析 (Data Parsing) ---
```

```
75      STATE_COORD_PARSE           // 核心状态：正在 <coordinates>...</coordinates> 内部解析数字
76  } ParseState;
77
78  /* * 紧凑的坐标点结构体
79   * 大小：12 字节
80   * 对齐：在 32 位系统上自然对齐
81   */
82  typedef struct {
83      float lon; // 经度
84      float lat; // 纬度
85      float alt; // 海拔
86  } GeoPoint;
87
88  /* * 解析上下文核心结构体
89   * 设计原则：包含所有运行时状态，支持重入（即不使用 static/global 变量）。
90   * 内存占用：约 2KB + 缓冲区。
91   */
92  typedef struct {
93      // 状态机控制
94      ParseState curr_state;        // 当前正在进行的动作（读标签/读属性/空闲）
95      ParseState logic_ctx;         // 当前所处的层级（是 Point 还是 LineString?）
96      ParseState logic_ctx_prev;    // 备份状态：进入 coordinates 前的状态，用于解析完后恢复
97
98      // 临时字符串缓冲区
99      char tag_buf[TAG_BUF_SIZE];  // 存放当前正在解析的标签名
100     uint16_t tag_len;             // 标签名当前长度
101     char num_buf[NUM_BUF_SIZE];  // 存放当前正在解析的一个数字字符串
102     uint16_t num_len;             // 数字字符串长度
103
104     // 单个点的解析状态
105     GeoPoint temp_point;          // 正在拼装的坐标点
106     uint8_t coord_comp_idx;       // 当前解析到第几个分量？0=经度，1=纬度，2=海拔
107     uint8_t has_valid_num;        // 标志位：num_buf 里是否有待处理的数字
108     uint8_t is_gx_coord; // 标记当前是否为 <gx:coord> 格式（空格分隔经纬度）
109
110     // LineString 处理专用字段
111     fpos_t line_len_pos;          // 记录文件位置：写入点数占位符的地方
112     uint32_t line_point_cnt;      // 计数器：当前 LineString 已经解析了多少个点
113
114     // LineString 批量写入缓存
115     GeoPoint line_cache[LINE_CACHE_SIZE]; // 缓存数组
116     uint16_t cache_idx;                   // 缓存当前使用量
117
118     // 文件句柄
119     FILE* fp_point; // 输出：点数据文件
```

```
120      FILE* fp_line;  // 输出：线数据文件
121
122      KmlError error; // 错误标记
123
124      uint32_t valid_point_cnt; //统计有效解析的点数量
125
126
127
128      //CP点管理
129      GeoPoint cp_list[MAX_CP_COUNT];        // 假设最多100个CP点
130      uint16_t cp_count;            // 已解析的CP点数量
131      uint8_t cp_list_visit[MAX_CP_COUNT];
132
133      // 分段管理
134      fpos_t global_header_pos;   // 文件最开始的位置（记录总段数）
135      fpos_t segment_header_pos;  // 当前分段头部的位置（记录当前段点数）
136
137      uint32_t total_segments;    // 总分段数
138      uint32_t curr_seg_points;   // 当前分段内的点数
139
140  } KmlCtx;
141
142  /* ================== 三、辅助函数实现 ================== */
143
144  /* * 重置临时点数据
145   * 用途：在开始解析一个新的 Point 之前，清空之前残留的数据。
146   */
147  static void reset_temp_point(KmlCtx* ctx) {
148      ctx->temp_point.lon = 0.0f;
149      ctx->temp_point.lat = 0.0f;
150      ctx->temp_point.alt = 0.0f; // 使用 0.0f作为无海拔的哨兵值（Sentinel）
151      ctx->coord_comp_idx = 0;
152      ctx->num_len = 0;
153      ctx->has_valid_num = 0;
154      // 安全起见，清零缓冲区
155      memset(ctx->num_buf, 0, NUM_BUF_SIZE);
156  }
157
158  /* * 刷新 LineString 缓存
159   * 用途：将内存中的 128 个点一次性写入磁盘，减少 fwrite 调用次数。
160   */
161  static void flush_line_cache(KmlCtx* ctx) {
162      if (ctx->cache_idx > 0) {
163          size_t written = fwrite(ctx->line_cache, sizeof(GeoPoint), ctx-
      >cache_idx, ctx->fp_line);
164          if (written != ctx->cache_idx) {
165              ctx->error = KML_ERR_IO_WRITE; // 磁盘满或写入错误
```

```
166             }
167             ctx->cache_idx = 0; // 清空缓存计数
168         }
169     }
170 
171 
172     /* 结束当前分段（回填点数） */
173     static void close_current_segment(KmlCtx* ctx) {
174         flush_line_cache(ctx); // 必须先清空缓存
175         if (ctx->error != KML_OK) return; // 新增：检查缓存刷新错误
176 
177         fpos_t end_pos;
178         if (fgetpos(ctx->fp_line, &end_pos) != 0) { ctx->error = KML_ERR_SEEK;
    return; }
179 
180         // 回跳到当前段的头部，写入点数
181         if (fsetpos(ctx->fp_line, &ctx->segment_header_pos) != 0) { ctx->error =
    KML_ERR_SEEK; return; }
182         if (fwrite(&ctx->curr_seg_points, sizeof(uint32_t), 1, ctx->fp_line) != 1)
    { ctx->error = KML_ERR_IO_WRITE; return; }
183 
184         // 跳回文件末尾
185         if (fsetpos(ctx->fp_line, &end_pos) != 0) { ctx->error = KML_ERR_SEEK;
    return; }
186     }
187 
188 
189     /* 开启新分段（写入占位符） */
190     static void start_new_segment(KmlCtx* ctx) {
191         // 记录新段头部位置
192         if (fgetpos(ctx->fp_line, &ctx->segment_header_pos) != 0) { ctx->error =
    KML_ERR_SEEK; return; }
193 
194         // 写 0 占位
195         uint32_t placeholder = 0;
196         if (fwrite(&placeholder, sizeof(uint32_t), 1, ctx->fp_line) != 1) {
197             ctx->error = KML_ERR_IO_WRITE;
198             return;
199         } // 修复：添加花括号，仅错误时return
200 
201         ctx->curr_seg_points = 0;
202         ctx->total_segments++;
203     }
204 
205     /* 检查当前点是否匹配任意 CP 点 */
206     static int is_match_cp(KmlCtx* ctx, GeoPoint* p) {
207         for (int i = 0; i < ctx->cp_count; i++) {
```

```
208             if (ctx->cp_list_visit[i] == 1 || i == 1)continue;
209             float d_lon = fabsf(p->lon - ctx->cp_list[i].lon);
210             float d_lat = fabsf(p->lat - ctx->cp_list[i].lat);
211             if (d_lon < SPLIT_THRESHOLD && d_lat < SPLIT_THRESHOLD) {
212                 printf("匹配第几个点:%d\n", i + 1);
213                 ctx->cp_list_visit[i] = 1;
214                 return 1; // 匹配!
215             }
216         }
217         return 0;
218     }
219
220
221     /* ================== 四、核心逻辑：坐标数值处理 ================== */
222
223     /* * 提交单个数值 (Commit Number)
224      * 触发时机：遇到逗号(,) 或 空格/换行 时。
225      * 功能：将 num_buf 中的字符串转为 float，填入 temp_point 的对应分量。
226      */
227     static void commit_number(KmlCtx* ctx) {
228         if (!ctx->has_valid_num) return; // 防御：处理连续逗号 "123,,456"
229
230         ctx->num_buf[ctx->num_len] = '\0'; // 确保字符串 NULL 结尾
231         float val = strtof(ctx->num_buf, NULL); // 修复：用strtof替代atof，避免double
    转float
232
233         // 根据索引填入经度、纬度或海拔
234         switch (ctx->coord_comp_idx) {
235         case 0: ctx->temp_point.lon = val; break;
236         case 1: ctx->temp_point.lat = val; break;
237         case 2: ctx->temp_point.alt = val; break;
238         default: break; // 忽略 KML 中可能出现的第4维数据 (如时间戳)
239         }
240
241         ctx->coord_comp_idx++; // 准备接收下一个分量
242         ctx->num_len = 0;      // 重置缓冲区指针
243         ctx->has_valid_num = 0; // 标记缓冲区为空
244     }
245
246     /* * 提交完整的点 (Commit Point)
247      * 触发时机：遇到空格、换行 或 <coordinates> 结束符 '<'。
248      * 功能：将拼装好的 GeoPoint 写入文件或缓存。
249      */
250     static void commit_point(KmlCtx* ctx) {
251         // 1. 确保最后一个正在解析的数字（通常是海拔）被保存
252         commit_number(ctx);
253
```

```
254         // 2. 完整性检查：如果只有经度没有纬度，视为无效点丢弃
255         if (ctx->coord_comp_idx < 2) {
256             reset_temp_point(ctx);
257             return;
258         }
259 
260         // 3. 根据当前上下文决定去向
261         if (ctx->logic_ctx_prev == STATE_CTX_POINT) {
262             //如果是单个地标点，直接写文件
263             if (fwrite(&ctx->temp_point, sizeof(GeoPoint), 1, ctx->fp_point) != 1)
    {
264                 ctx->error = KML_ERR_IO_WRITE;
265             }
266             else {
267         //         ctx->valid_point_cnt++;  //统计有效点
268             }
269             if (ctx->cp_count < MAX_CP_COUNT) {
270                 ctx->cp_list[ctx->cp_count++] = ctx->temp_point;
271             }
272         }
273         else if (ctx->logic_ctx_prev == STATE_CTX_LINE) {
274             int need_split = is_match_cp(ctx, &ctx->temp_point);
275 
276             // 策略：如果遇到 CP 点，该点作为旧段的结束。
277             // 添加点到缓存（修复：检查数组越界）
278             if (ctx->cache_idx < LINE_CACHE_SIZE) {
279                 ctx->line_cache[ctx->cache_idx++] = ctx->temp_point;
280                 ctx->curr_seg_points++;
281             }
282             else {
283                 ctx->error = KML_ERR_INTERNAL; // 缓存满，标记内部错误
284                 return;
285             }
286 
287             if (ctx->cache_idx >= LINE_CACHE_SIZE) flush_line_cache(ctx);
288 
289             // 如果触发分段，且当前段不为空（避免连续分段产生空段）
290             if (need_split && ctx->curr_seg_points > MIN_SEGMENT_COUNT) {
291                 // 1. 结束当前段
292                 close_current_segment(ctx);
293                 if (ctx->error != KML_OK) return; // 新增：检查错误
294                 // 2. 开启新段
295                 start_new_segment(ctx);
296 
297                 // 3.航线连续，新段的起点应该是 CP 点本身
298                  ctx->line_cache[ctx->cache_idx++] = ctx->temp_point;
299                  ctx->curr_seg_points++;
```

```
300             }
301         }
302 
303         // 4. 重置，为下一个点做准备
304         reset_temp_point(ctx);
305     }
306 
307     /* ================== 五、核心逻辑：XML 标签处理 ================== */
308 
309     /* * 处理开始标签 (Handle Open Tag)
310      * 触发时机：遇到 '>' 且是开始标签时 (如 <LineString>)
311      */
312     static void handle_tag_open(KmlCtx* ctx) {
313         if (ctx->tag_len == 0) return;
314         ctx->tag_buf[ctx->tag_len] = '\0';
315         char* tag = ctx->tag_buf;
316 
317         // 快速过滤：忽略 <?xml...>
318         if (tag[0] == '?' || tag[0] == '!') return;
319 
320         /* 状态切换逻辑 */
321         if (strcmp(tag, "Placemark") == 0) {
322             ctx->logic_ctx = STATE_CTX_PLACEMARK;
323         }
324         else if (ctx->logic_ctx == STATE_CTX_PLACEMARK) {
325             // 只有在 Placemark 内部才关心 Point 或 LineString
326             if (strcmp(tag, "Point") == 0) {
327                 ctx->logic_ctx = STATE_CTX_POINT;
328             }
329             else if (strcmp(tag, "LineString") == 0 || strcmp(tag, "Track") == 0
    || strcmp(tag, "gx:Track") == 0) {
330                 ctx->logic_ctx = STATE_CTX_LINE;
331 
332                 // 如果是文件的第一条线，初始化总头部
333                 if (ctx->total_segments == 0) {
334                     if (fgetpos(ctx->fp_line, &ctx->global_header_pos) != 0) {
335                         ctx->error = KML_ERR_SEEK;
336                         return; // 新增：失败时直接返回
337                     }
338                     uint32_t zero = 0;
339                     if (fwrite(&zero, sizeof(uint32_t), 1, ctx->fp_line) != 1) {
340                         ctx->error = KML_ERR_IO_WRITE;
341                         return;
342                     }
343 
344                     // 立即开始第一段
345                     start_new_segment(ctx);
```

```
346             }
347         }
348     }
349 
350     // 检测坐标数据区开始
351     // 同时兼容 KML 标准 (<coordinates>) 和 Google Earth 扩展 (<gx:coord>)
352     if (strcmp(tag, "coordinates") == 0 || strcmp(tag, "gx:coord") == 0 ||
    strcmp(tag, "coord") == 0) {
353         if (ctx->logic_ctx == STATE_CTX_POINT || ctx->logic_ctx ==
    STATE_CTX_LINE) {
354             ctx->logic_ctx_prev = ctx->logic_ctx; // 备份：你是谁的坐标?
355             ctx->curr_state = STATE_COORD_PARSE;  // 切换：进入数据解析模式
356             reset_temp_point(ctx);
357 
358             if (strcmp(tag, "coord" ) == 0 || strcmp(tag, "gx:coord") == 0) {
359                 ctx->is_gx_coord = 1;
360             }
361             else {
362                 ctx->is_gx_coord = 0;
363             }
364         }
365     }
366 }
367 
368 /* * 处理结束标签 (Handle Close Tag)
369  * 触发时机：遇到 '>' 且是结束标签时 (如 </LineString>)
370  */
371 static void handle_tag_close(KmlCtx* ctx) {
372     ctx->tag_buf[ctx->tag_len] = '\0';
373     char* tag = ctx->tag_buf;
374 
375     /* 状态回退逻辑 */
376     if (strcmp(tag, "Placemark") == 0) {
377         ctx->logic_ctx = STATE_CTX_ROOT;
378     }
379     else if (strcmp(tag, "Point") == 0) {
380         ctx->logic_ctx = STATE_CTX_PLACEMARK;
381     }
382     else if (strcmp(tag, "LineString") == 0 || strcmp(tag, "gx:Track") == 0 ||
    strcmp(tag, "Track") == 0) {
383         ctx->logic_ctx = STATE_CTX_PLACEMARK;
384 
385     }
386 }
387 
388 /* ================== 六、主解析循环入口 ================== */
389 
```

```
390  KmlError parse_kml_file(const char* input_file) {
391      //使用堆分配空间
392      KmlCtx* ctx = (KmlCtx*)malloc(sizeof(KmlCtx));
393      if (!ctx) return KML_ERR_INTERNAL;
394      uint8_t* read_buf = NULL;
395      size_t n_read;
396      int is_self_closing = 0; // 标记是否为 <Point/> 这种自闭合标签
397      KmlError final_err = KML_OK; // 用于保存最终错误码的局部变量
398      // 1. 初始化上下文
399      memset(ctx, 0, sizeof(KmlCtx));
400      ctx->curr_state = STATE_IDLE;
401      ctx->logic_ctx = STATE_CTX_ROOT;
402
403      // 2. 打开文件
404      FILE* fp_in = fopen(input_file, "rb");
405      if (!fp_in) return KML_ERR_FILE_OPEN;
406
407      ctx->fp_point = fopen(FILE_POINT_NAME, "wb");
408      ctx->fp_line = fopen(FILE_LINE_NAME, "wb");
409
410      // 错误处理：任何文件打开失败都需清理
411      if (!ctx->fp_point || !ctx->fp_line) {
412          ctx->error = KML_ERR_FILE_OPEN; // 记录错误以便统一处理
413          goto cleanup;
414      }
415
416      // 3. 分配读取缓冲区（可根据系统情况改为静态数组）
417      read_buf = (uint8_t*)malloc(READ_BUF_SIZE);
418      if (!read_buf) {
419          ctx->error = KML_ERR_INTERNAL;
420          goto cleanup;
421      }
422
423      // 4. 流式处理大循环
424      while ((n_read = fread(read_buf, 1, READ_BUF_SIZE, fp_in)) > 0) {
425          for (size_t i = 0; i < n_read; i++) {
426              if (ctx->error != KML_OK) goto cleanup; // 检测到 IO 错误立即停止
427
428              char c = read_buf[i];
429
430              // ------------------------------------------------------
431              // 核心状态机设计：分为 "数据流模式" 和 "标签模式"
432              // ------------------------------------------------------
433
434              // 分支 A: 坐标数据解析模式（性能敏感路径）
435              // 一旦进入 <coordinates>，我们就不再解析 XML 结构，而是专注解析
    "120.1,30.2 120.2..."
```

```
436             if (ctx->curr_state == STATE_COORD_PARSE) {
437                 if (c == '<') {
438                     // 遇到 '<' 意味着坐标区结束了 (即使是 </coordinates>)
439 
440                     // 1. 提交残留数据 (防止类似 "100,20<" 的边界情况)
441                     if (ctx->has_valid_num) commit_point(ctx);
442 
443                     // 2. 状态迁移：退出数据模式，准备解析结束标签
444                     ctx->logic_ctx = ctx->logic_ctx_prev;
445                     ctx->curr_state = STATE_TAG_NAME;
446                     ctx->tag_len = 0;
447                     is_self_closing = 0;
448                 }
449                 else if (c == ',' || isspace((int)c)) {
450                     // 分隔符：提交当前数字
451                     commit_number(ctx);
452                     if (isspace((int)c)) {
453                         // 在标准kml格式下空格/换行意味着一个 Point 结束 (前提是已拿到经纬度)
454                         if (!ctx->is_gx_coord) {
455                             if (ctx->coord_comp_idx > 0) commit_point(ctx);
456                         }
457                     }
458                 }
459                 else {
460                     // 收集数字字符
461                     if (ctx->num_len < NUM_BUF_SIZE - 1) {
462                         // 简单的白名单过滤，确保 atof 安全
463                         if ((c >= '0' && c <= '9') || c == '.' || c == '-' || c == '+' || c == 'e' || c == 'E') {
464                             ctx->num_buf[ctx->num_len++] = c;
465                             ctx->has_valid_num = 1;
466                         }
467                     }
468                 }
469                 continue; // 在数据模式下，直接跳过后面的 XML 逻辑
470             }
471 
472             // 分支 B: XML 结构解析模式
473             switch (ctx->curr_state) {
474             case STATE_IDLE: // 等待 '<'
475                 if (c == '<') {
476                     ctx->curr_state = STATE_TAG_NAME;
477                     ctx->tag_len = 0;
478                     is_self_closing = 0;
479                 }
480                 break;
```

```
481
482             case STATE_TAG_NAME: // 正在读标签名
483                 if (c == '>') {
484                     // 标签结束：<Tag>
485
486                     if (is_self_closing) handle_tag_close(ctx); // 处理 <Tag/>
487                     else handle_tag_open(ctx);
488                     // 如果没进入坐标模式，就回到空闲
489                     if (ctx->curr_state != STATE_COORD_PARSE) {
490                         ctx->curr_state = STATE_IDLE;
491                     }
492                 }
493                 else if (isspace((int)c)) {
494                     // 遇到空格：<Tag attr...>
495                     handle_tag_open(ctx); // 此时标签名已完整，先处理
496                     ctx->curr_state = STATE_ATTR_SKIP; // 然后去跳过属性
497                 }
498                 else if (c == '/') {
499                     if (ctx->tag_len == 0) {
500                         ctx->curr_state = STATE_TAG_END_NAME; // 是结束标签 </
501                     }
502                     else {
503                         is_self_closing = 1; // 是自闭合标签 <Tag/
504                     }
505                 }
506                 else {
507                     // 记录标签名，防止缓冲区溢出
508                     if (ctx->tag_len < TAG_BUF_SIZE - 1) {
509                         ctx->tag_buf[ctx->tag_len++] = c;
510                     }
511                 }
512                 break;
513
514             case STATE_ATTR_SKIP: // 属性跳过模式
515                 // 我们不解析属性，只等待 '>' 结束
516                 if (c == '>') {
517                     if (is_self_closing) handle_tag_close(ctx);
518                     if (ctx->curr_state != STATE_COORD_PARSE) {
519                         ctx->curr_state = STATE_IDLE;
520                     }
521                 }
522                 else if (c == '/') {
523                     is_self_closing = 1; // <Tag attr="val"/>
524                 }
525                 break;
526
527             case STATE_TAG_END_NAME: // 正在读结束标签 </Tag...
```

```
528                 if (c == '>') {
529                     handle_tag_close(ctx);
530                     ctx->curr_state = STATE_IDLE;
531                 }
532                 else if (!isspace((int)c)) {
533                     if (ctx->tag_len < TAG_BUF_SIZE - 1) {
534                         ctx->tag_buf[ctx->tag_len++] = c;
535                     }
536                 }
537                 break;
538 
539             default:
540                 ctx->curr_state = STATE_IDLE;
541                 break;
542             }
543         }
544     }
545 
546     // 检查是否有硬件/系统级的读取错误
547     if (ferror(fp_in)) {
548         ctx->error = KML_ERR_IO_READ;
549         goto cleanup;
550     }
551 
552 
553     // --- 结束处理 ---
554     // 1. 关闭最后一段
555     if (ctx->curr_seg_points > MIN_SEGMENT_COUNT) {
556         close_current_segment(ctx);
557     }
558 
559     // 2. 回填文件总头部的 Total Segments
560     if (ctx->fp_line && ctx->error == KML_OK) {
561         fpos_t end_pos;
562         if (fgetpos(ctx->fp_line, &end_pos) == 0) {
563             if (fsetpos(ctx->fp_line, &ctx->global_header_pos) == 0) {
564                 fwrite(&ctx->total_segments, sizeof(uint32_t), 1, ctx-
    >fp_line);
565                 fsetpos(ctx->fp_line, &end_pos);
566             }
567             else {
568                 ctx->error = KML_ERR_SEEK;
569             }
570         }
571         else {
572             ctx->error = KML_ERR_SEEK;
573         }
```

```
574         }
575
576     cleanup:
577         /* 资源释放与清理 */
578         if (ctx) {
579             final_err = ctx->error;
580         }
581         else {
582             final_err = KML_ERR_INTERNAL;
583         }
584         if (read_buf) free(read_buf);
585         if (fp_in) fclose(fp_in);
586
587         if (ctx) {
588             if (ctx->fp_point) fclose(ctx->fp_point);
589             if (ctx->fp_line) fclose(ctx->fp_line);
590             free(ctx); // 释放结构体内存
591             ctx = NULL; // 防止悬空指针
592         }
593
594         // 如果解析中途失败，删除生成的垃圾文件
595         if (final_err != KML_OK) {
596             remove(FILE_POINT_NAME);
597             remove(FILE_LINE_NAME);
598         }
599
600         return final_err;
601     }
602
603
604     /*
605     * 对分段线段文件进行滤波，确保每个分段的点数不要超过阈值
606     */
607     KmlError filter_kml_lines(const char* src_file_path, const char* dst_file_path)
         {
608         FILE* fp_src = fopen(src_file_path, "rb");
609         FILE* fp_dst = fopen(dst_file_path, "wb");
610         KmlError ret_error = KML_OK;
611
612         if (!fp_src || !fp_dst) {
613             if (fp_src) fclose(fp_src);
614             if (fp_dst) fclose(fp_dst);
615             return KML_ERR_FILE_OPEN;
616         }
617         uint16_t write_idx;
618         uint32_t total_segments = 0;
619         //修改为堆上分配内存
```

```
620         GeoPoint* read_cache = NULL;
621         GeoPoint* write_cache = NULL;
622         // 读取总段数
623         if (fread(&total_segments, sizeof(uint32_t), 1, fp_src) != 1) {
624             ret_error = KML_ERR_IO_READ;
625             goto error_cleanup;
626         }
627         // 写入总段数
628         if (fwrite(&total_segments, sizeof(uint32_t), 1, fp_dst) != 1) {
629             ret_error = KML_ERR_IO_WRITE;
630             goto error_cleanup;
631         }
632 
633         read_cache = (GeoPoint*)malloc(sizeof(GeoPoint) * LINE_CACHE_SIZE);
634         write_cache = (GeoPoint*)malloc(sizeof(GeoPoint) * LINE_CACHE_SIZE);
635         if (!read_cache || !write_cache) {
636             ret_error = KML_ERR_INTERNAL; // 内存不足
637             goto error_cleanup;
638         }
639         /*
640         // 栈上内存分配 (约 3KB)
641         GeoPoint read_cache[LINE_CACHE_SIZE];
642         GeoPoint write_cache[LINE_CACHE_SIZE];
643         */
644         write_idx = 0;
645 
646         // 逐段处理
647         for (uint32_t i = 0; i < total_segments; i++) {
648             uint32_t src_count = 0;
649             if (fread(&src_count, sizeof(uint32_t), 1, fp_src) != 1) {
650                 ret_error = KML_ERR_IO_READ;
651                 goto error_cleanup;
652             }
653 
654             // 记录回填位置
655             fpos_t segment_head_pos;
656             if (fgetpos(fp_dst, &segment_head_pos) != 0) {
657                 ret_error = KML_ERR_SEEK;
658                 goto error_cleanup;
659             }
660 
661             // 写占位符
662             uint32_t placeholder = 0;
663             if (fwrite(&placeholder, sizeof(uint32_t), 1, fp_dst) != 1) {
664                 ret_error = KML_ERR_IO_WRITE;
665                 goto error_cleanup;
666             }
```

```
667
668         uint32_t dst_count = 0;
669         write_idx = 0; // 每个新段开始时，重置写缓存索引
670
671         // === 策略分支 ===
672         if (src_count <= MAX_POINTS_PER_SEGMENT) {
673             /* 场景 A: 直接复制 */
674             uint32_t remain = src_count;
675             while (remain > 0) {
676                 uint16_t chunk = (remain > LINE_CACHE_SIZE) ? LINE_CACHE_SIZE
    : (uint16_t)remain;
677
678                 if (fread(read_cache, sizeof(GeoPoint), chunk, fp_src) !=
    chunk) {
679                     ret_error = KML_ERR_IO_READ;
680                     goto error_cleanup;
681                 }
682
683                 // 统一使用 write_cache 写入
684                 if (write_idx + chunk > LINE_CACHE_SIZE) {
685                     if (fwrite(write_cache, sizeof(GeoPoint), write_idx,
    fp_dst) != write_idx) {
686                         ret_error = KML_ERR_IO_WRITE;
687                         goto error_cleanup;
688                     }
689                     write_idx = 0;
690                 }
691
692                 memcpy(&write_cache[write_idx], read_cache, chunk *
    sizeof(GeoPoint));
693                 write_idx += chunk;
694                 remain -= chunk;
695             }
696             dst_count = src_count;
697         }
698         else {
699             /* 场景 B: 均匀抽稀 */
700             if (MAX_POINTS_PER_SEGMENT <= 1) {
701                 ret_error = KML_ERR_INTERNAL;
702                 goto error_cleanup;
703             }
704
705             float step = (float)(src_count - 1) / (float)
    (MAX_POINTS_PER_SEGMENT - 1);
706             float next_sample_idx = 0.0f;
707             uint32_t processed = 0;
708             uint8_t is_last_point_processed = 0; // 新增：标记尾点是否已处理
```

```
709             while (processed < src_count) {
710                 uint32_t remain = src_count - processed;
711                 uint16_t chunk = (remain > LINE_CACHE_SIZE) ? LINE_CACHE_SIZE
    : (uint16_t)remain;
712 
713                 if (fread(read_cache, sizeof(GeoPoint), chunk, fp_src) !=
    chunk) {
714                     ret_error = KML_ERR_IO_READ;
715                     goto error_cleanup;
716                 }
717 
718                 for (uint16_t k = 0; k < chunk; k++) {
719                     uint32_t global_idx = processed + k;
720                     int is_last_point = (global_idx == src_count - 1);
721 
722                     // 判定逻辑：保留当前点的条件
723                     if (is_last_point || (float)global_idx >= next_sample_idx)
    {
724                         // 核心修改：允许最多 MAX+1 个点
725                         if (dst_count <= MAX_POINTS_PER_SEGMENT) {
726                             write_cache[write_idx++] = read_cache[k];
727                             dst_count++;
728                             is_last_point_processed = is_last_point ? 1 :
    is_last_point_processed;
729                             // 非尾点时推进采样索引
730                             if (!is_last_point) {
731                                 next_sample_idx += step;
732                             }
733                         }
734 
735                         // 缓存满，刷盘
736                         if (write_idx >= LINE_CACHE_SIZE) {
737                             if (fwrite(write_cache, sizeof(GeoPoint),
    write_idx, fp_dst) != write_idx) {
738                                 ret_error = KML_ERR_IO_WRITE;
739                                 goto error_cleanup;
740                             }
741                             write_idx = 0;
742                         }
743                     }
744                 }
745                 processed += chunk;
746             }
747         }
748 
749         // 统一刷入剩余缓存 (无论场景A还是B)
750         if (write_idx > 0) {
```

```
751             if (fwrite(write_cache, sizeof(GeoPoint), write_idx, fp_dst) !=
    write_idx) {
752                 ret_error = KML_ERR_IO_WRITE;
753                 goto error_cleanup;
754             }
755             write_idx = 0;
756         }
757 
758         // 回填点数
759         fpos_t end_pos;
760         if (fgetpos(fp_dst, &end_pos) != 0) { ret_error = KML_ERR_SEEK; goto
    error_cleanup; }
761         if (fsetpos(fp_dst, &segment_head_pos) != 0) { ret_error =
    KML_ERR_SEEK; goto error_cleanup; }
762         if (fwrite(&dst_count, sizeof(uint32_t), 1, fp_dst) != 1) { ret_error
    = KML_ERR_IO_WRITE; goto error_cleanup; }
763         if (fsetpos(fp_dst, &end_pos) != 0) { ret_error = KML_ERR_SEEK; goto
    error_cleanup; }
764     }
765 
766     free(read_cache);
767     free(write_cache);
768     read_cache = NULL;
769     write_cache = NULL;
770     fclose(fp_src);
771     fclose(fp_dst);
772     fp_src = NULL; // 防止后续误用
773     fp_dst = NULL;
774 
775     // 文件替换
776     if (remove(src_file_path) != 0) {
777         ret_error = KML_ERR_IO_WRITE; // 无法删除旧文件
778         goto error_cleanup_no_src_dst;
779     }
780     if (rename(dst_file_path, src_file_path) != 0) {
781         ret_error = KML_ERR_IO_WRITE;
782         goto error_cleanup_no_src_dst;
783     }
784 
785     return KML_OK;
786 
787 error_cleanup:
788     if (read_cache) free(read_cache);
789     if (write_cache) free(write_cache);
790     if (fp_src) fclose(fp_src);
791     if (fp_dst) fclose(fp_dst);
792 
```

```
793 error_cleanup_no_src_dst:
794     remove(dst_file_path); // 失败时清理产生的临时文件
795     return ret_error;
796 }
797
798
799
800
801 /**
802  * @brief 验证Point文件（流式读取，不占用RAM）
803  */
804 void verify_point_bin_stream(const char* file_path) {
805     printf("\n=== Checking Point File: %s ===\n", file_path);
806     FILE* fp = fopen(file_path, "rb");
807     if (!fp) { perror("Open failed"); return; }
808
809     // 1. 获取文件大小和总点数
810     fseek(fp, 0, SEEK_END);
811     long file_size = ftell(fp);
812     size_t total_points = file_size / sizeof(GeoPoint);
813     printf("Total Points: %zu (Size: %ld bytes)\n", total_points, file_size);
814
815     if (total_points == 0) { fclose(fp); return; }
816
817     GeoPoint pt;
818     rewind(fp);
819
820     // 2. 打印
821    // printf("--- Head (First 10) ---\n");
822     for (size_t i = 0; i < total_points; i++) {
823         fread(&pt, sizeof(GeoPoint), 1, fp);
824         printf("[%zu] %.6f, %.6f, %.2f\n", i+1, pt.lon, pt.lat, pt.alt);
825     }
826     fclose(fp);
827 }
828
829 /**
830  * @brief 打印单个点辅助函数
831  */
832 static void print_point(const char* label, size_t index, GeoPoint p) {
833     printf("%s[%zu] Lon: %.6f, Lat: %.6f, Alt: %.2f\n", label, index, p.lon,
    p.lat, p.alt);
834 }
835
836 /**
837  * @brief 验证 Line 文件 (新格式: Header + Array)
838  * 格式: [TotalCount (4B)] + [GeoPoint 1] + [GeoPoint 2] ...
```

```
 */
void verify_line_bin_fixed(const char* file_path) {
    printf("\n=== Checking Line File (Header Mode): %s ===\n", file_path);

    FILE* fp = fopen(file_path, "rb");
    if (!fp) {
        perror("Error opening file");
        return;
    }

    // 1. 读取文件头的总点数
    uint32_t total_points = 0;
    if (fread(&total_points, sizeof(uint32_t), 1, fp) != 1) {
        printf("Error: File too short, cannot read header.\n");
        fclose(fp);
        return;
    }

    printf("Header says Total Points: %u\n", total_points);

    // (可选) 校验文件实际大小是否匹配
    // 理论大小 = 4字节头 + 点数 * 12字节
    long expected_size = sizeof(uint32_t) + (long)total_points *
sizeof(GeoPoint);
    fseek(fp, 0, SEEK_END);
    long actual_size = ftell(fp);
    if (actual_size != expected_size) {
        printf("Warning: File size mismatch! Expected %ld bytes, got %ld
bytes.\n", expected_size, actual_size);
        // 注意：如果不匹配，我们仍然尝试读取，但在生产环境中这通常意味着文件损坏
    }

    // 如果没有点，直接退出
    if (total_points == 0) {
        fclose(fp);
        return;
    }

    // 2. 打印前 10 个点
    // 文件指针现在正好位于 Header 之后 (偏移量 4)，直接读取即可
    fseek(fp, sizeof(uint32_t), SEEK_SET); // 确保指针在数据区开头

    printf("--- Head (First 10) ---\n");
    size_t count_to_read = (total_points < 10) ? total_points : 10;
    GeoPoint pt;

    for (size_t i = 0; i < count_to_read; i++) {
```

```
884             fread(&pt, sizeof(GeoPoint), 1, fp);
885             print_point("H", i, pt);
886         }
887 
888         // 3. 打印后 10 个点
889         if (total_points > 10) {
890             printf("--- Tail (Last 10) ---\n");
891 
892             // 计算偏移量：Header大小 + (总点数 - 10) * 点大小
893             long offset = sizeof(uint32_t) + (long)((total_points - 10) *
    sizeof(GeoPoint));
894 
895             // 直接跳转到倒数第10个点的位置
896             fseek(fp, offset, SEEK_SET);
897 
898             for (size_t i = 0; i < 10; i++) {
899                 // 读取并打印
900                 if (fread(&pt, sizeof(GeoPoint), 1, fp) == 1) {
901                     // 计算该点的全局索引
902                     size_t global_idx = total_points - 10 + i;
903                     print_point("T", global_idx, pt);
904                 }
905             }
906         }
907 
908         fclose(fp);
909     }
910 
911     void verify_line_bin_segmented(const char* file_path) {
912         printf("\n=== Checking Segmented Line File: %s ===\n", file_path);
913         FILE* fp = fopen(file_path, "rb");
914         if (!fp) {
915             printf("Error: Cannot open file %s\n", file_path);
916             return;
917         }
918 
919         // 1. 读取总段数
920         uint32_t total_segs = 0;
921         if (fread(&total_segs, sizeof(uint32_t), 1, fp) != 1) {
922             printf("Error: Failed to read total segments\n");
923             fclose(fp);
924             return;
925         }
926         printf("Total Segments: %u\n", total_segs);
927 
928         // 2. 遍历每一段
929         for (uint32_t s = 0; s < total_segs; s++) {
```

```
930             uint32_t pts_in_seg = 0;
931             if (fread(&pts_in_seg, sizeof(uint32_t), 1, fp) != 1) {
932                 printf("Error: Failed to read segment %u point count\n", s + 1);
933                 fclose(fp);
934                 return;
935             }
936             printf("  Segment %u: %u points\n", s + 1, pts_in_seg);
937 
938             // 跳过点数据，或者打印首尾点
939             if (pts_in_seg > 0) {
940                 GeoPoint p;
941                 if (fread(&p, sizeof(GeoPoint), 1, fp) != 1) {
942                     printf("Error: Failed to read start point of segment %u\n", s
    + 1);
943                     fclose(fp);
944                     return;
945                 }
946                 printf("    Start: %.6f, %.6f\n", p.lon, p.lat);
947 
948                 // 跳过中间的点
949                 if (pts_in_seg > 1) {
950                     if (fseek(fp, (pts_in_seg - 2) * sizeof(GeoPoint), SEEK_CUR)
    != 0) {
951                         printf("Error: Failed to seek in segment %u\n", s + 1);
952                         fclose(fp);
953                         return;
954                     }
955                     if (fread(&p, sizeof(GeoPoint), 1, fp) != 1) {
956                         printf("Error: Failed to read end point of segment %u\n",
    s + 1);
957                         fclose(fp);
958                         return;
959                     }
960                     printf("    End  : %.6f, %.6f\n", p.lon, p.lat);
961                 }
962             }
963         }
964         fclose(fp);
965     }
966 
```

使用说明

基础数据单元

`GeoPoint` 结构体

代码中所有的坐标点都以这个结构体形式存储。

- **大小**：12 字节 (3 * 4 bytes)
- **内存布局**：

```
代码块
|   0-3 bytes   |   4-7 bytes   |   8-11 bytes  |
|---------------|---------------|---------------|
|  float (Lon)  |  float (Lat)  |  float (Alt)  |
```

点数据文件 (`point_temp.bin`)

该文件用于存储独立的 `<Point>` 标签数据（如地标）。

- **结构类型**：**纯数组结构 (Flat Array)**
- **特点**：没有任何文件头或元数据，文件中全是紧密排列的 `GeoPoint` 数据。
- **文件布局图**：

```
代码块
[ Start of File ]
+---------------------------+
| GeoPoint 1 (12 Bytes)     |
+---------------------------+
| GeoPoint 2 (12 Bytes)     |
+---------------------------+
| ...                       |
+---------------------------+
| GeoPoint N (12 Bytes)     |
+---------------------------+
[ End of File ]
```

线/轨迹数据文件 (`line_temp.bin`)

该文件用于存储 `<LineString>` 或 `<Track>` 数据。由于轨迹可能由多条不连续的线段组成（分段），该文件采用了 **分段存储结构**。

- **结构类型**：**层级头结构 (Hierarchical Header)**
- **逻辑流程**：先读总段数 -> 循环 -> 读当前段点数 -> 读具体坐标。

文件布局详细说明：

偏移量 (Offset)	数据类型	描述
0x00	uint32_t	**总分段数 (Total Segments)**，假设值为 N
0x04	uint32_t	**第 1 段的点数 (Count 1)**
0x08	GeoPoint	第 1 段 - 第 1 个点
...	...	...
...	GeoPoint	第 1 段 - 最后一个点
Next	uint32_t	**第 2 段的点数 (Count 2)**
Next+4	GeoPoint	第 2 段 - 第 1 个点
...	...	...
Next	uint32_t	**第 N 段的点数 (Count N)**
...	GeoPoint	第 N 段 - 坐标数据

可视化结构图：

代码块

```
[ Global Header ]
+-----------------------------------+
| Total Segments (4 Bytes)          |  <-- 例如: 2 (表示有两条独立的轨迹)
+-----------------------------------+

[ Segment 1 Block ]
+-----------------------------------+
| Segment 1 Point Count (4 Bytes)   |  <-- 例如: 1000
+-----------------------------------+
| GeoPoint 1-1 (12 Bytes)           |
| GeoPoint 1-2 (12 Bytes)           |
| ...                               |
| GeoPoint 1-1000 (12 Bytes)        |
+-----------------------------------+

[ Segment 2 Block ]
+-----------------------------------+
| Segment 2 Point Count (4 Bytes)   |  <-- 例如: 500
+-----------------------------------+
| GeoPoint 2-1 (12 Bytes)           |
```

```
| ...                                |
| GeoPoint 2-500 (12 Bytes)          |
+------------------------------------+

[ End of File ]
```

调用说明

代码块

```
const char* kml_file_path = "D:\\KT_Project\\OPEN_SOURCE\\new\\X-TRACK-main\\Software\\X-Track\\USER\\App\\Resource\\Track\\guangzhou100.kml";  // 你的KML文件路径

    printf("Start parsing...\n");
    KmlError err = parse_kml_file(kml_file_path);//提取标记点和航线轨迹到，并按标记点切分到
                                                 //FILE_LINE_NAME文件
    if (err == KML_OK) {
        printf("Success. Data extracted to .bin files.\n");
    }
    else {
        printf("Failed with error code: %d\n", err);
    }
    //对FILE_LINE_NAME文件分段文件进行滤波处理确保每段的点数不超过MAX_POINTS_PER_SEGMENT+1个点
    filter_kml_lines(FILE_LINE_NAME, FILE_LINE_FILTERED_NAME);
    verify_line_bin_segmented(FILE_LINE_NAME);//验证可视化打印输出
    verify_point_bin_stream(FILE_POINT_NAME);//验证可视化打印输出
```

调试说明，**先坚持拿来主义，不去关注内部的代码实现**，按照调用说明先调用**parse_kml_file(kml_file_path)**

再调用**filter_kml_lines(FILE_LINE_NAME, FILE_LINE_FILTERED_NAME)**即可。

第一种做法：按照轨迹数据文件的文件布局，先读出总的分段数目，然后可以利用一个动态大数组使用malloc分配，一次性读出每段的所有坐标点数，渲染到屏幕上。

第二种做法：也可以尝试每次只读出1000个点然后渲染到屏幕上，然后继续读取1000点渲染到屏幕上，这样可以尽可能减少内存的占用，但相对应渲染时间会增加。

如果要匹配当前位置对应在屏幕的哪一个位置，可以使用文件流的方式，比如每次就只读出128个点，然后遍历该点于其中哪一段的哪一个索引匹配（要确定两个索引，i：第几段，j：该段的第几个索引）

简单代码示意

```c
// 定义匹配的容差（根据你的GPS精度调整）
#define MATCH_EPSILON_DEG  4e-5f
#define MATCH_EPSILON_ALT  3.0f  // 海拔容差（米）

typedef struct {
    int32_t seg_index;   // 第几段（从0开始，-1表示未找到）
    int32_t point_index; // 该段中的第几个点（从0开始，-1表示未找到）
} SearchResult;

/**
 * @brief 在线路文件中查找特定坐标的索引
 * @param file_path line_temp.bin 的路径
 * @param target 要查找的目标点
 * @return SearchResult 包含段索引和点索引
 */
SearchResult find_point_index(const char* file_path, GeoPoint target) {
    SearchResult result = { -1, -1 };

    FILE* fp = fopen(file_path, "rb");
    if (!fp) return result;

    // 1. 读取总段数
    uint32_t total_segments = 0;
    if (fread(&total_segments, sizeof(uint32_t), 1, fp) != 1) {
        fclose(fp);
        return result;
    }

    // 2. 申请堆内存缓存（与之前保持一致，约 1.5KB）
    // 这样做既快又不会爆栈
    GeoPoint* cache = (GeoPoint*)malloc(sizeof(GeoPoint) * LINE_CACHE_SIZE);
    if (!cache) {
        fclose(fp);
        return result; // 内存不足
    }

    // 3. 遍历每一段
    for (uint32_t s_idx = 0; s_idx < total_segments; s_idx++) {
        uint32_t points_in_seg = 0;

        // 读取当前段的头部（点数）
        if (fread(&points_in_seg, sizeof(uint32_t), 1, fp) != 1) break;

        uint32_t processed_in_seg = 0;

        // 4. 分块读取当前段的点数据
        while (processed_in_seg < points_in_seg) {
```

```
            // 计算本次要读多少个点
            uint32_t remain = points_in_seg - processed_in_seg;
            uint16_t chunk = (remain > LINE_CACHE_SIZE) ? LINE_CACHE_SIZE :
(uint16_t)remain;

            // 批量读取 IO (时间优化的关键)
            size_t read_cnt = fread(cache, sizeof(GeoPoint), chunk, fp);
            if (read_cnt != chunk) break; // 文件损坏或读取错误

            // 在内存中快速比对
            for (uint16_t k = 0; k < chunk; k++) {
                GeoPoint* p = &cache[k];

                // 核心比较逻辑：检查经纬度和海拔的差值是否在容差范围内
                if (fabsf(p->lon - target.lon) <= MATCH_EPSILON_DEG &&
                    fabsf(p->lat - target.lat) <= MATCH_EPSILON_DEG &&
                    fabsf(p->alt - target.alt) <= MATCH_EPSILON_ALT) {

                    // === 找到了! ===
                    result.seg_index = s_idx;
                    result.point_index = processed_in_seg + k;

                    // 清理资源并返回
                    goto cleanup_success;
                }
            }

            processed_in_seg += chunk;
        }
    }

cleanup_success:
    free(cache);
    fclose(fp);
    return result;
}

void test_search() {
    // 假设你从 GPS 获得了一个点
    GeoPoint my_gps_data = { 113.273455f, 23.180013f, 22.8f };

    SearchResult res = find_point_index(FILE_LINE_NAME, my_gps_data);

    if (res.seg_index != -1) {
        printf("找到目标! \n");
        printf("位于第 %d 段 (Segment Index)\n", res.seg_index);
        printf("是该段的第 %d 个点 (Point Index)\n", res.point_index);
```

```
94        }
95        else {
96            printf("未找到该点，请检查容差设置。\n");
97        }
98    }
```

但还有更好的优化，记录上一次匹配到的`段索引（Segment Index）`和`点索引（Point Index）`。

优先比对上一次匹配的前后感兴趣的窗口，比如前后128个点，如果找到则基本不耗时，如果没找到则再采取从头遍历