hnswalg.h

向量数据库——hnswlib 源码剖析 - 知乎 (zhihu.com)
HNSW算法(nsmlib/hnswlib)-CSDN博客

class HierarchicalNSW

【构造】

HierarchicalNSW构造函数一

- 作用：用于创建 HNSW 层次结构索引
- 核心代码：func loadIndex

```
HierarchicalNSW(SpaceInterface<dist_t> *s, const std::string &location,
                bool /**nmslib*/ = false, size_t max_elements = 0,
                bool allow_replace_deleted = false)
    : allow_replace_deleted_(allow_replace_deleted) {
  loadIndex(location, s, max_elements);
}
```

HierarchicalNSW构造函数二

这个构造函数没有使用loadIndex(location, s, max_elements)直接构建索引
而是创建了一个空的HierarchicalNSW索引空间

```
HierarchicalNSW(SpaceInterface<dist_t> *s, size_t max_elements, size_t M = 16,
                size_t ef_construction = 200, size_t random_seed = 100,
                bool allow_replace_deleted = false)
    : link_list_locks_(max_elements),
      label_op_locks_(MAX_LABEL_OPERATION_LOCKS),
      element_levels_(max_elements),
      allow_replace_deleted_(allow_replace_deleted) {

  //初始化成员变量
  max_elements_ = max_elements;
  num_deleted_ = 0;
  data_size_ = s->get_data_size();
  fstdistfunc_ = s->get_dist_func();
  dist_func_param_ = s->get_dist_func_param();
  M_ = M;
```

```
    maxM_ = M_;
    maxM0_ = M_ * 2;
    //潜在的邻居列表，用于优选邻居节点
    ef_construction_ = std::max(ef_construction, M_);
    ef_ = 10;

    //随机数生成器，随机生成层数信息
    level_generator_.seed(random_seed);
    //更新概率生成器
    update_probability_generator_.seed(random_seed + 1);

    //第0层的最大节点跳表大小
    size_links_level0_ = maxM0_ * sizeof(tableint) + sizeof(linklistsizeint);
    //第0层的一个向量的结构（请看下文的图解）
    size_data_per_element_ =
        size_links_level0_ + data_size_ + sizeof(labeltype);
    //数据域
    offsetData_ = size_links_level0_;
    label_offset_ = size_links_level0_ + data_size_;

    //这个字段好像没有什么意义？？？
    offsetLevel0_ = 0;

    //开辟第0层的内存
    data_level0_memory_ =
        (char *)malloc(max_elements_ * size_data_per_element_);

    //当前节点的数量
    cur_element_count = 0;

    //已访问节点缓存池
    visited_list_pool_ = new VisitedListPool(1, max_elements);

    // 用于对第一个节点进行特殊处理的初始化
    enterpoint_node_ = -1;
    maxlevel_ = -1;

    //开辟空的节点邻居跳表
    linkLists_ = (char **)malloc(sizeof(void *) * max_elements_);
    size_links_per_element_ =
        maxM_ * sizeof(tableint) + sizeof(linklistsizeint);

    //分层策略
    mult_ = 1 / log(1.0 * M_);
    revSize_ = 1.0 / mult_;
}
```

【构建】

func loadIndex

- 想调用这个方法，前提是存在一个已经构建好了的二进制索引文件，里面有我们想要的HNSW空间
- 相当于把之前save的文件读取了一遍（不是重构！！！是加载！！！）

#？？？这个方法的每一步需要细看

```
void loadIndex(const std::string &location, SpaceInterface<dist_t> *s,
               size_t max_elements_i = 0) {

  //使用 `std::ifstream` 类以二进制方式打开指定位置 (`location`) 的索引文件。
  std::ifstream input(location, std::ios::binary);

  // 获取文件大小（获取方式如下）
  input.seekg(0, input.end); //文件指针移动到末尾
  std::streampos total_filesize = input.tellg();//返回末尾指针的位置下标
  input.seekg(0, input.beg);  //指针回到文件的开头

  //读取索引文件的基本信息
  readBinaryPOD(input, offsetLevel0_);  //这个字段无实际意义，可以视作对齐字节
  readBinaryPOD(input, max_elements_);
  readBinaryPOD(input, cur_element_count);

  size_t max_elements = max_elements_i;
  if (max_elements < cur_element_count)
    max_elements = max_elements_;
  max_elements_ = max_elements;
  readBinaryPOD(input, size_data_per_element_);
  readBinaryPOD(input, label_offset_);
  readBinaryPOD(input, offsetData_);
  readBinaryPOD(input, maxlevel_);
  readBinaryPOD(input, enterpoint_node_);

  //读取距离度量参数
  readBinaryPOD(input, maxM_);  //每个元素在 HNSW 图中的最大邻居数（除了第一层）
  readBinaryPOD(input, maxM0_);  //第一层元素的最大邻居数
  readBinaryPOD(input, M_);  //HNSW 图中每个节点的实际邻居数（应用中M_==maxM_）
  readBinaryPOD(input, mult_);  //用于计算新增向量落在哪一层，正文会详细讲到
  readBinaryPOD(input, ef_construction_);  //HNSW 图构建过程中使用的控制索引查询/建立的时延权衡？？？

  //读取空间接口信息
  data_size_ = s->get_data_size();
  fstdistfunc_ = s->get_dist_func();
  dist_func_param_ = s->get_dist_func_param();
```

```
  //记录文件读取位置
  auto pos = input.tellg();

  // 可选 - 检查索引文件是否损坏
  input.seekg(cur_element_count * size_data_per_element_, input.cur);
  for (size_t i = 0; i < cur_element_count; i++) {

    unsigned int linkListSize;
    readBinaryPOD(input, linkListSize);
    if (linkListSize != 0) {
      input.seekg(linkListSize, input.cur);
    }
  }
  //关闭文件
  input.clear();
  //指针回到pos位置
  input.seekg(pos, input.beg);


  //分配最下面一层的内存（全部节点数*每个元素的大小），开辟一个char指针，返回指针头
  data_level0_memory_ = (char *)malloc(max_elements * size_data_per_element_);
  //往最下面那一次内存储索引
  input.read(data_level0_memory_, cur_element_count * size_data_per_element_);


  //每个节点对应的邻居跳表所要占的空间
  //其空间大小的计算方式是：最多近邻节点数*4 + 4
  /*
  因为typedef unsigned int tableint; typedef unsigned int linklistsizeint;
  所以sizeof都是4
  */
  size_links_per_element_ =
      maxM_ * sizeof(tableint) + sizeof(linklistsizeint);


  //第0层节点的邻居表的大小
  size_links_level0_ = maxM0_ * sizeof(tableint) + sizeof(linklistsizeint);


  //这些锁有什么用？？？
  //？？？我知道这里是创建了max_elements个互斥锁，然后和节点邻居表更新锁
link_list_locks_交换
  std::vector<std::mutex>(max_elements).swap(link_list_locks_);
  //label_op_locks_是哈希并发锁，都是MAX_LABEL_OPERATION_LOCKS又是什么？？？
  std::vector<std::mutex>(MAX_LABEL_OPERATION_LOCKS).swap(label_op_locks_);

  //图操作经常需要判断哪些节点已经走过，这里提供一个已经申请好空间的池子，减少内存频繁申
请释放的开销
  visited_list_pool_ = new VisitedListPool(1, max_elements);
```

```
  //节点邻居跳表，char**，每个节点对应数据依然是连续数组
  //linkLists_是邻居表存储实体，是一个二维数组，行由max_elements即最大向量个数指定，列由该向量实际落在的层数在构建该向量时具体分配内存
  linkLists_ = (char **)malloc(sizeof(void *) * max_elements);

  //每个节点在哪一层，是vector数组，数组索引是节点内部id。vector，初始化为max_elements个0（默认都在第0层，同时如果在第i层，就一定在第i-1层）
  element_levels_ = std::vector<int>(max_elements);


  //mult_正文详细聊
  revSize_ = 1.0 / mult_;

  //可能是步长？？？但是为啥构建过程需要步长呢，下文完全用不到
  ef_ = 10;


  for (size_t i = 0; i < cur_element_count; i++) {  //遍历当前层的所有待插节点
    label_lookup_[getExternalLabel(i)] = i;  //label_lookup_是label与内部id的映射，unordered_map？？？

    //获取这个节点的邻居节点表的内存大小
    unsigned int linkListSize;
    readBinaryPOD(input, linkListSize);

    if (linkListSize == 0) {
      element_levels_[i] = 0;  //如果这个节点没有邻接点，那么这个节点插入第0层
      linkLists_[i] = nullptr; //这个节点的邻居节点跳表也是空指针
    } else {

      //如果有邻居节点，就开始计算层数
      //size_links_per_element_是每个元素最大邻接点内存容量
      //linkListSize是这个节点的实际的近邻节点数量（单位是byte）【？？？读出来时其实是level0的邻接点节点】
      //密集：如果实际的近邻节点数量>最大邻接点内存容量，就往高层（非0层）插入该节点
      //稀疏：如果实际的近邻节点数量<最大邻接点内存容量，就往第0层插入该节点
      element_levels_[i] = linkListSize / size_links_per_element_;  //？？？

      //开辟这个节点的邻居节点跳表，并存储近邻节点
      linkLists_[i] = (char *)malloc(linkListSize);
      input.read(linkLists_[i], linkListSize);
    }
  }

  //这行代码放在这里是增量构建用的，初次构建用不着
  for (size_t i = 0; i < cur_element_count; i++) {  //遍历当前层的所有待插节点
    if (isMarkedDeleted(i)) {
      num_deleted_ += 1;
```

```
        if (allow_replace_deleted_)
          deleted_elements.insert(i);
      }
    }


    input.close();

    return;
  }
```

mult_ = 1 / log(1.0 * M_);
revSize_ = 1.0 / mult_;

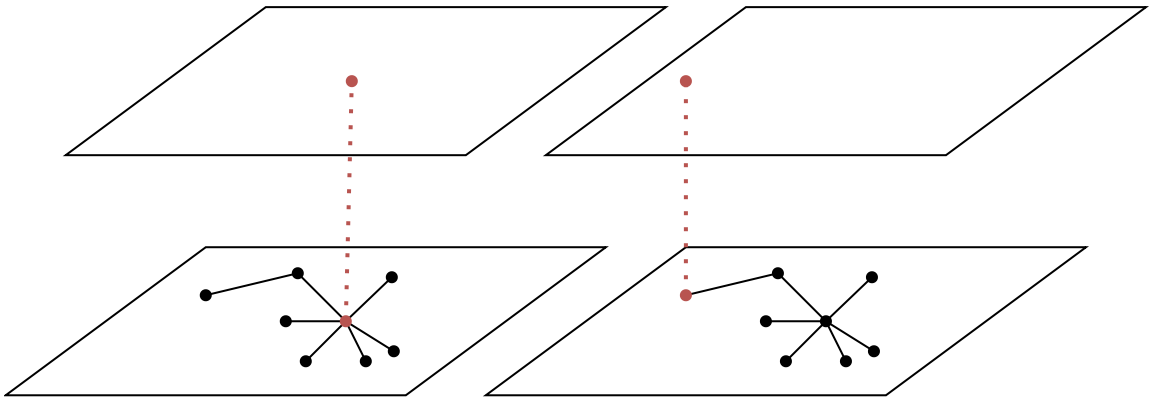
- M_每个层（除了第0层）的节点可以有多少个邻居
- revSize= *log(1.0 * M)*

构建索引时的分层策略

初步估计是在初次构建索引时采用的分层策略

element_levels_[i] = linkListSize / size_links_per_element_;

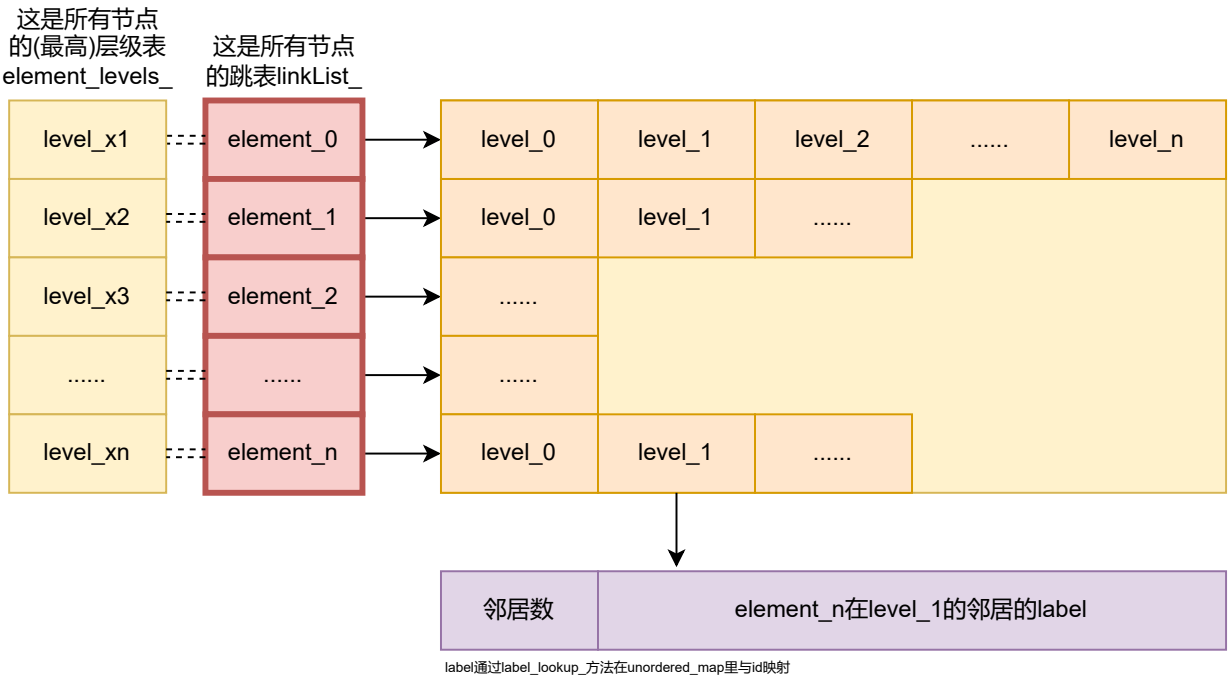
这条策略是将密集的节点往上放，稀疏的节点往下放，示意图如下所示：



而且越密集的点，要准确表示，理论上也应该把它放高

节点邻居表linkLists_

- level*0是全节点的；其它层的邻居节点作减法就行；这个减法是查询近邻点的element_levels*就可以了

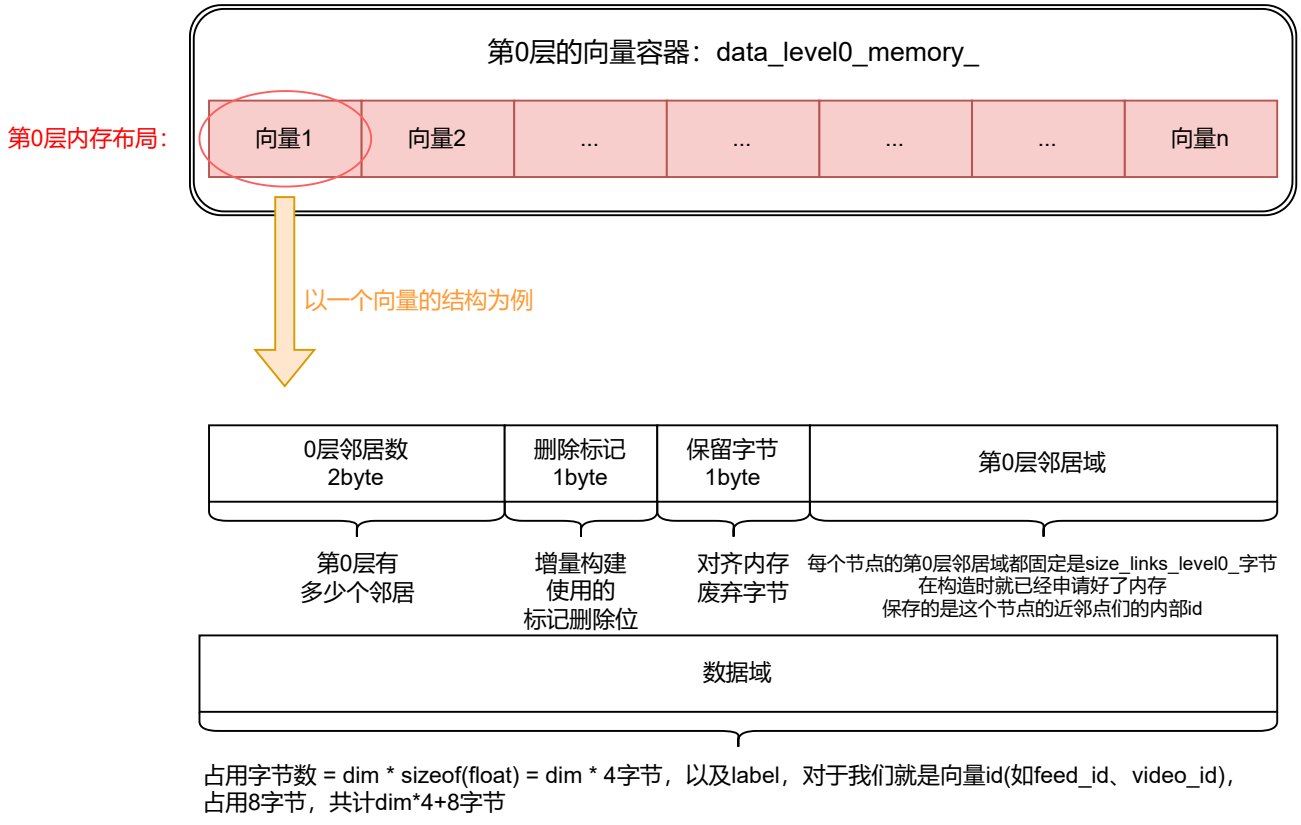


节点邻居跳表linkLists_的各层结构详细解释如下所示：

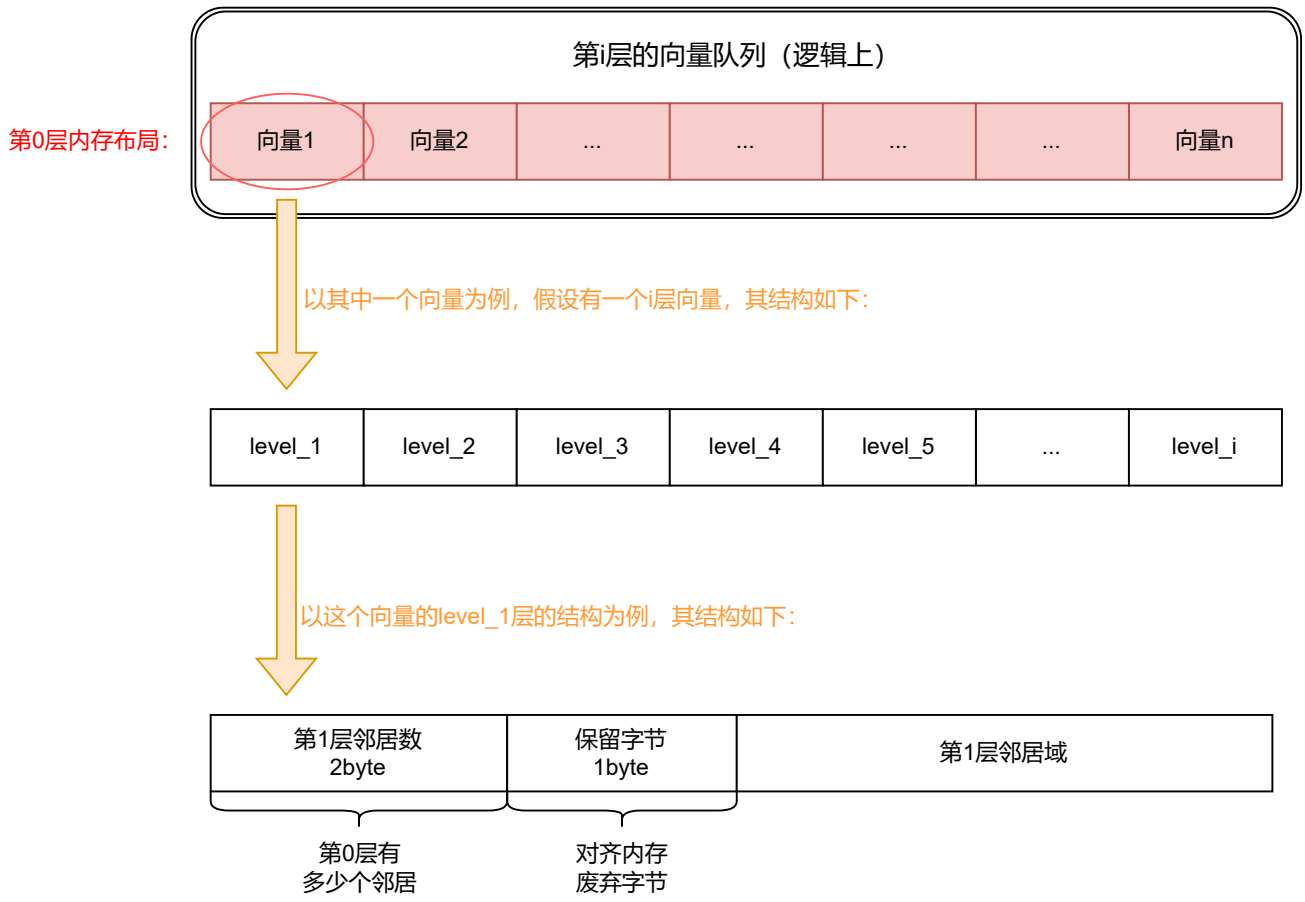
第0层结构

第0层保存了全部的向量数据，包括：

- 原始向量
- label（向量内部id）
- 邻居域（每个向量在第0层的近邻向量的id）



非0层结构



#可能的优化设计？？？

- element在level_i的近邻点一定是level_0的子集，level_i存了是不是低层就可以不用存了呢
- 因为这里构造的时候向量大小是固定好了的，这种优化可以尝试但是性价比可能不大

【增加】

func addPoint - 替换删除点

- 作用：该函数用于将一个新点添加到 HNSW 索引中，并可以选择替换已标记为删除的点
- 参数：
  - `data_point`：要添加的点的数据指针。
  - `label`：要添加的点的外部标签（用于标识点的唯一标识）。
  - `replace_deleted`（可选，默认值为 `false`）：是否允许替换已删除的点。

```
void addPoint(const void *data_point, labeltype label,
              bool replace_deleted = false) {

  // 这里利用了c++的并发编程机制，上了一个增量锁
  // 但凡其它addPoint读取到这里都得等一个PV操作
  std::unique_lock<std::mutex> lock_label(getLabelOpMutex(label));
```

```
  if (!replace_deleted) {
    addPoint(data_point, label, -1);  //这是一个方法重载，直接去添加新点
    return;
  }


// 检查是否存在可替换的已删除点
  tableint internal_id_replaced;
  //获取已删除元素的集合lock_deleted_elements的互斥锁（上锁）
  std::unique_lock<std::mutex> lock_deleted_elements(deleted_elements_lock);
  //如果deleted_elements不空，表示is_vacant_place（存在可替换的位置）
  bool is_vacant_place = !deleted_elements.empty();
  //如果可以替换，获取这个将要被替换的节点的内部id
  if (is_vacant_place) {
    internal_id_replaced = *deleted_elements.begin();
    deleted_elements.erase(internal_id_replaced);  //然后从 deleted_elements集合中
erase这个节点
  }
  lock_deleted_elements.unlock();  //（解开lock_deleted_elements）

  // 如果没有可替换的已删除点，则直接调用另一个重载版本的`addPoint`函数添加新点
  // else add point to vacant place
  if (!is_vacant_place) {
    addPoint(data_point, label, -1);
  } else {
    //如果有可替换的已删除点，我们假设对已删除的元素没有并发操作

    //获取要替换的已删除点的外部标签 (`label_replaced`)
    labeltype label_replaced = getExternalLabel(internal_id_replaced);
    //将新点的外部标签 (`label`) 设置给要替换的已删除点
    setExternalLabel(internal_id_replaced, label);

    //上标签查找表互斥锁，防止其它线程变动标签查找表
    std::unique_lock<std::mutex> lock_table(label_lookup_lock);
    //更新标签查找表
    label_lookup_.erase(label_replaced);
    label_lookup_[label] = internal_id_replaced;
    //解锁
    lock_table.unlock();
    //取消标记为已删除（这里指的是第0层的结构里占1字节的删除标记）
    unmarkDeletedInternal(internal_id_replaced);
    //更新点（直接调用更新方法，详情见更新方法）
    updatePoint(data_point, internal_id_replaced, 1.0);
  }
}
```

func addPoint - 直接加入点

- 作用：该函数用于将一个新点添加到 HNSW 索引中，并指定要添加的层级
- 参数：
  - `data_point`：要添加的点的数据指针。
  - `label`：要添加的点的外部标签（用于标识点的唯一标识）。
  - `level`(可选，默认值为 -1)：指定要插入的层级，如果为 -1，则会随机选择层级。

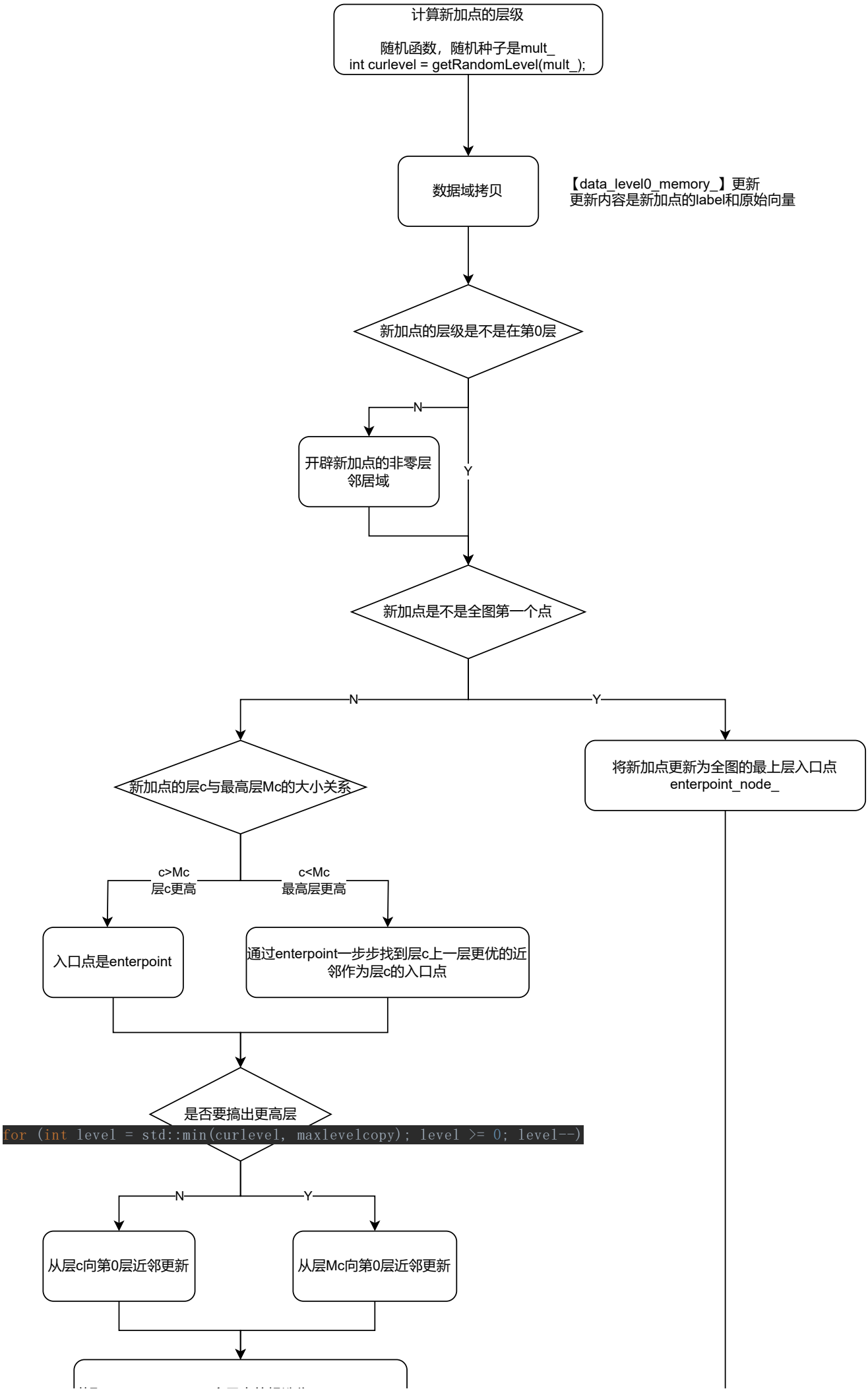
```
tableint addPoint(const void *data_point, labeltype label, int level) {
  tableint cur_c = 0;
  {
    /*
     检查是否存在相同标签的元素
     如果存在，则更新该元素的值，释放锁并返回其内部 ID
    */
    std::unique_lock<std::mutex> lock_table(label_lookup_lock);  //上label查找锁
    auto search = label_lookup_.find(label);  //查找是否存在相同标签的元素，返回的是查询结果的迭代器，如果元素不存在，则返回label_lookup_.end()


    if (search != label_lookup_.end()) {  //如果元素存在
      tableint existingInternalId = search->second;  //获取这个元素的内部id（本质是通过label找到内部id）
      lock_table.unlock(); //解开查找表锁

          //如果说查到了，却是个已删除的节点
      if (isMarkedDeleted(existingInternalId)) {
        unmarkDeletedInternal(existingInternalId);  //取消这个节点的删除标记
      }
      updatePoint(data_point, existingInternalId, 1.0);  //更新这个节点，下面有对更新方法的详细的解释

      return existingInternalId; //返回更新了的元素的内部id并跳出方法
    }

   //如果待加元素不存在，就是真真正正地要add了（而不是update）
   //下标从0开始，cur_element_count下标从1开始
   //例如，cur_element_count=100指的是0~99这100个节点
   //那么新增节点的下标（cur_c）就是99
    cur_c = cur_element_count;
    cur_element_count++;  //节点总数+1
    label_lookup_[label] = cur_c; //新增节点的内部id存入label查找表！！！！！！！！！！！（解释了内部id和label之间的关系）
  }



  //接下来是真正地add，而不是调用update方法
```

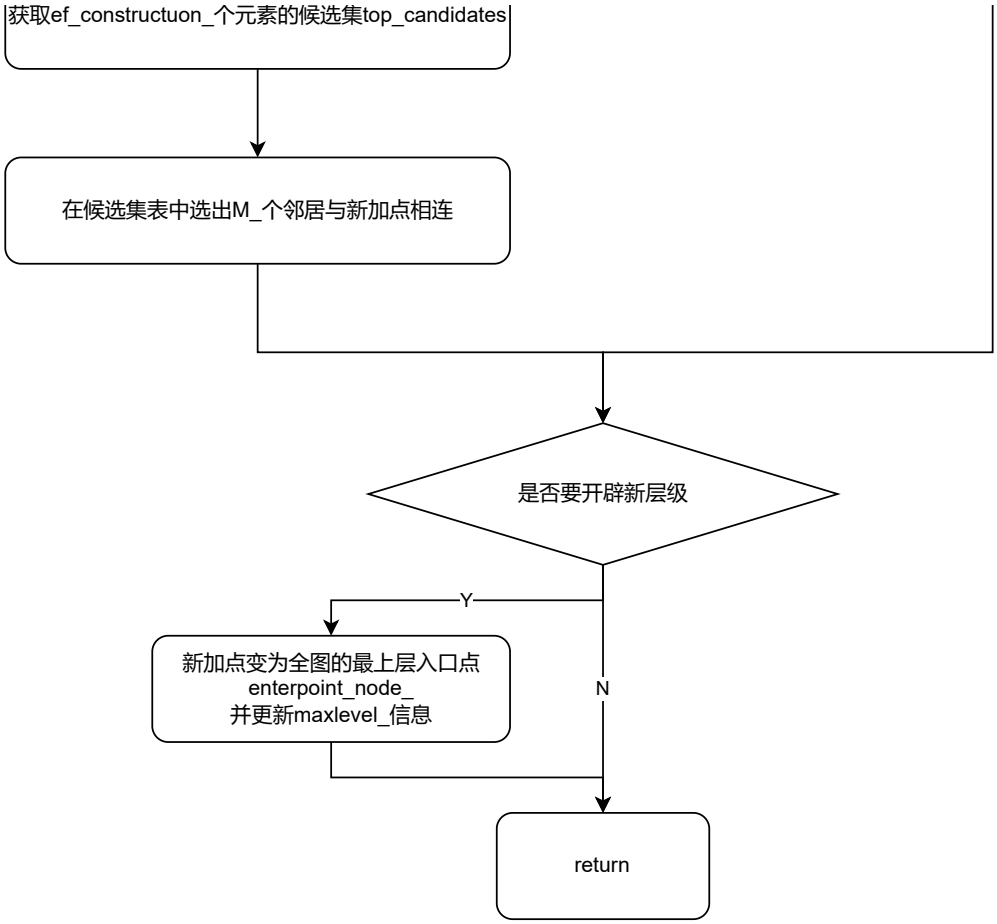
```
  std::unique_lock<std::mutex> lock_el(link_list_locks_[cur_c]);  //增员锁，每个节点都有一个，这里是锁上新增节点的增员锁
  int curlevel = getRandomLevel(mult_);  //利用上面提到的mult_来随机生成层数（构造函数里我们已经赋值了）？？？
  if (level > 0)  //如果我们指定过层级就覆盖，如果没有指定过层级就用随机策略
    curlevel = level;

  element_levels_[cur_c] = curlevel;  //新节点的层级信息赋值

  std::unique_lock<std::mutex> templock(global);  //锁上全局锁，用来保护最大层级maxlevel_的，因为接下来的操作可能要涉及“开新层”，防止多线程重复建层
  int maxlevelcopy = maxlevel_;
  if (curlevel <= maxlevelcopy)
    templock.unlock(); //如果不用开层就解锁全局锁


  tableint currObj = enterpoint_node_;  //enterpoint_node_是从索引文件中直接读取的，是这个HNSW的第一个点（起始点）
  tableint enterpoint_copy = enterpoint_node_;

  //在第0层向量存储内存后面覆盖一个element的内存
  //把原先内存里的残存数据全部覆盖为0（腾空/归零）
  //offsetLevel0_可以从文件中读取，但是此处是0（构造函数里面赋值了）
  memset(data_level0_memory_ + cur_c * size_data_per_element_ + offsetLevel0_,
         0, size_data_per_element_);

  //参考第0层的向量结构，这里是在覆盖label和原始向量到刚才归零的内存区
  //把label赋值给getExternalLabeLp的返回值labeltype *
  //把data_point赋值给getDataByInternalId的返回值char *
  memcpy(getExternalLabeLp(cur_c), &label, sizeof(labeltype));
  memcpy(getDataByInternalId(cur_c), data_point, data_size_);

  //再开一片跳表的内存
  if (curlevel) {  //curlevel非0就运行
    linkLists_[cur_c] =
        (char *)malloc(size_links_per_element_ * curlevel + 1);
    //归零
    memset(linkLists_[cur_c], 0, size_links_per_element_ * curlevel + 1);
  }

  if ((signed)currObj != -1) {  //如果新加节点不是第一个节点（不是起始点）//这里的第一个点是全图的第一个点

    //这个if本质上是用来挑选一个更合适的入口点currObj
    if (curlevel < maxlevelcopy) {  //如果新加层级<最高层级
      dist_t curdist = fstdistfunc_(data_point, getDataByInternalId(currObj),
                                    dist_func_param_);  //计算出新加点和起始点之间的距离
```

```
      //从最高层往curlevel+1遍历；目的是从上往下寻找更近邻的点
      for (int level = maxlevelcopy; level > curlevel; level--) {
        bool changed = true;


        while (changed) {
          changed = false;
          unsigned int *data;
          std::unique_lock<std::mutex> lock(link_list_locks_[currObj]);  //锁上起始点的近邻表锁
          data = get_linklist(currObj, level);  //查询起始点在level层的近邻节点
          int size = getListCount(data);

          //然后遍历这些近邻节点与新加点之间的距离
          tableint *datal = (tableint *)(data + 1);
          for (int i = 0; i < size; i++) {
            tableint cand = datal[i];
            dist_t d = fstdistfunc_(data_point, getDataByInternalId(cand),
                                    dist_func_param_);
            //如果距离更短就标记更新
            if (d < curdist) {
              curdist = d;
              currObj = cand;  //如果起始点的邻接点更近，下一次就搜索近邻点的近邻点
              changed = true;  //退出更新的条件是：最优点周围的所有邻接点距离都大于最优点，导致changed无法置为true而跳出while循环
            }
          }
        }


      }
    }


    bool epDeleted = isMarkedDeleted(enterpoint_copy);
    //再从curlevel, maxlevelcopy中较小的往第0层遍历
    for (int level = std::min(curlevel, maxlevelcopy); level >= 0; level--) {

      //生成某一层的候选集
      //调用searchBaseLayer函数从 HNSW 图的当前层级level开始，从当前元素currObj搜索与新元素 data_point最近的ef_construction_个元素 (候选者)
      //搜索结果存储在优先队列 (top_candidates) 中，元素按与新元素的距离排序 (距离较大的元素排在前面)
      std::priority_queue<std::pair<dist_t, tableint>,
                          std::vector<std::pair<dist_t, tableint>>,
                          CompareByFirst>
          top_candidates = searchBaseLayer(currObj, data_point, level);
      if (epDeleted) {  //如果起始点被标记删除了
        top_candidates.emplace(
```

```
            fstdistfunc_(data_point, getDataByInternalId(enterpoint_copy),
                         dist_func_param_),
            enterpoint_copy);  //因为删除了，会导致起始点不会被纳入候选集的计算，因此
这里还得补算一下被删除的起始点
        if (top_candidates.size() > ef_construction_)  //如果候选集尺寸超了就用pop
删掉最远点
          top_candidates.pop();
      }

      //调用mutuallyConnectNewElement函数将新元素data_point与内部ID-cur_c与候选列表
top_candidates中的元素连接起来
      currObj = mutuallyConnectNewElement(data_point, cur_c, top_candidates,
                                          level, false);
    }
  } else {  //新加点是全图的第一个节点
    //那么当前层就是最大层了
    enterpoint_node_ = 0;  //而新加点就是enterpoint_node_，内部id为0（下标）
    maxlevel_ = curlevel;
  }

  // 如果需要更新最大层级
  if (curlevel > maxlevelcopy) {
    enterpoint_node_ = cur_c;  //最顶层的起始点就变成了新加点
    maxlevel_ = curlevel;
  }
  return cur_c;
}
```

直接加入点的算法逻辑


计算新加点的层级
随机函数，随机种子是mult_
int curlevel = getRandomLevel(mult_);
数据域拷贝
【data_level0_memory_】更新
更新内容是新加点的label和原始向量
新加点的层级是不是在第0层
N
Y
开辟新加点的非零层
邻居域
新加点是不是全图第一个点
N
Y
将新加点更新为全图的最上层入口点
enterpoint_node_
新加点的层c与最高层Mc的大小关系
c>Mc
层c更高
c<Mc
最高层更高
入口点是enterpoint
通过enterpoint一步步找到层c上一层更优的近邻作为层c的入口点
是否要搞出更高层
for (int level = std::min(curlevel, maxlevelcopy); level >= 0; level--)
N
Y
从层c向第0层近邻更新
从层Mc向第0层近邻更新



func updatePoint

- 作用：这段代码的功能是更新 HNSW 图中一个元素的数据及其邻居链接
- 参数：
  - dataPoint: 用于更新的元素的新数据 (const void *)
  - internalId: 要更新的元素的内部 ID (tableint)
  - updateNeighborProbability: 更新邻居连接的概率 (float)

```
void updatePoint(const void *dataPoint, tableint internalId,
                 float updateNeighborProbability) {

  // 使用memcpy函数将新数据复制到要更新的元素的内存空间中
  memcpy(getDataByInternalId(internalId), dataPoint, data_size_);

  int maxLevelCopy = maxlevel_;
  tableint entryPointCopy = enterpoint_node_;

  //如果全图就这一个点，那么直接返回
  //（因为更新向量并不困难，困难的是更新近邻跳表，只不过全图只有一个点就不会有这个麻烦了）
  if (entryPointCopy == internalId && cur_element_count == 1)
    return;

  //获取要更新的元素层级
  int elemLevel = element_levels_[internalId];
  std::uniform_real_distribution<float> distribution(0.0, 1.0);  //随机浮点数生成
```

器？？？

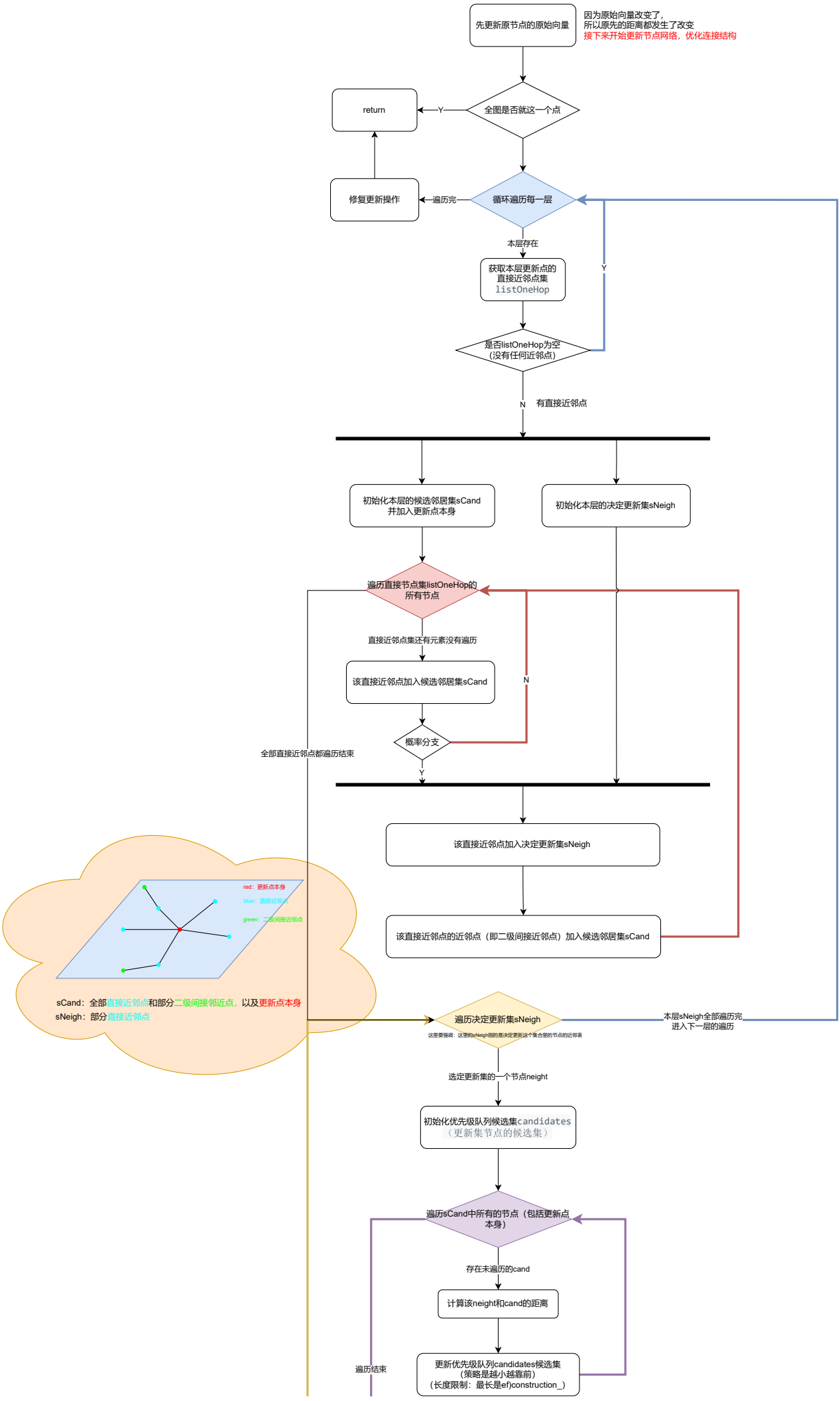
```
  //从第0层往elemLevel遍历，更新邻居节点
  for (int layer = 0; layer <= elemLevel; layer++) {
    //初始化两个无序哈希集合
    std::unordered_set<tableint> sCand;  //候选邻居集：用于存储当前层级可作为邻居的元素集合（无序哈希集合）
    std::unordered_set<tableint> sNeigh;  //决定更新集：用于存储需要更新连接的邻居集合（无序哈希集合）

    //通过内部id和层级直接获取该层的直接近邻点
    std::vector<tableint> listOneHop =
        getConnectionsWithLock(internalId, layer);
    if (listOneHop.size() == 0)  //如果这一层没有直接近邻点，就前往高一层
      continue;

    //该层的直接近邻点插入候选邻居集
    sCand.insert(internalId);
    for (auto &&elOneHop : listOneHop) {
      sCand.insert(elOneHop);

      //随机选择这个点是否需要更新
      //（为啥不是全部直接近邻点全部更新呀？？？这样不会让有的邻居消息过期吗？？？难道是因为向量更新之后可能会变远而做的“启发式更新”吗）
      if (distribution(update_probability_generator_) >
          updateNeighborProbability)
        continue;
      //插入决定更新集
      sNeigh.insert(elOneHop);
      //获取二级间接邻居集并插入候选集
      std::vector<tableint> listTwoHop =
          getConnectionsWithLock(elOneHop, layer);
      for (auto &&elTwoHop : listTwoHop) {
        sCand.insert(elTwoHop);
      }
    }

    //遍历决定更新集
    for (auto &&neigh : sNeigh) {

      //初始化优先级队列：候选集（距离越远，排得越靠前）
      std::priority_queue<std::pair<dist_t, tableint>,
                          std::vector<std::pair<dist_t, tableint>>,
                          CompareByFirst>
          candidates;

      //确定可保留的近邻节点的数量：min{候选集长度上限，sCand的长度}
      size_t size =
          sCand.find(neigh) == sCand.end()  //如果neigh不在sCand中，size-1+1
```

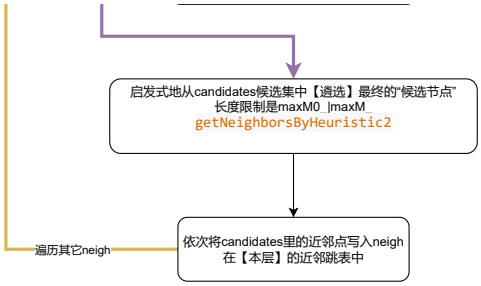
```
                ? sCand.size()
                : sCand.size() - 1; // 如果neigh在sCand中，size-1，减的是更新节点自己本身（前面把自己加上了）
      size_t elementsToKeep = std::min(ef_construction_, size);

      //
      for (auto &&cand : sCand) {

        //排除邻居节点本身
        if (cand == neigh)
          continue;

        //计算邻居和候选节点的距离
        dist_t distance =
            fstdistfunc_(getDataByInternalId(neigh),
                         getDataByInternalId(cand), dist_func_param_);
        //更新候选集
        if (candidates.size() < elementsToKeep) {
          candidates.emplace(distance, cand);
        } else {
          if (distance < candidates.top().first) {
            candidates.pop();
            candidates.emplace(distance, cand);
          }
        }
      }

      // 启发式地从候选集中选择连接
      getNeighborsByHeuristic2(candidates, layer == 0 ? maxM0_ : maxM_);

      {
        std::unique_lock<std::mutex> lock(link_list_locks_[neigh]);  //获取邻居的互斥锁
        linklistsizeint *ll_cur;  //获取邻居在当前层级的邻居列表指针 (`ll_cur`)
        ll_cur = get_linklist_at_level(neigh, layer);
        size_t candSize = candidates.size();
        setListCount(ll_cur, candSize);  //设置邻居列表的元素个数 (`setListCount(ll_cur, candSize)`)
        tableint *data = (tableint *)(ll_cur + 1);

        //将优先队列 (`candidates`) 中的元素依次取出，并写入邻居的邻居列表内存 (`data`) 中
        for (size_t idx = 0; idx < candSize; idx++) {
          data[idx] = candidates.top().second;
          candidates.pop();
        }
      }
    }
  }
```

```
    //修复更新操作可能导致的链接断裂或其他问题（暂且没看完是怎么修复的？？？）
    repairConnectionsForUpdate(dataPoint, entryPointCopy, internalId, elemLevel,
                               maxLevelCopy);
}
```

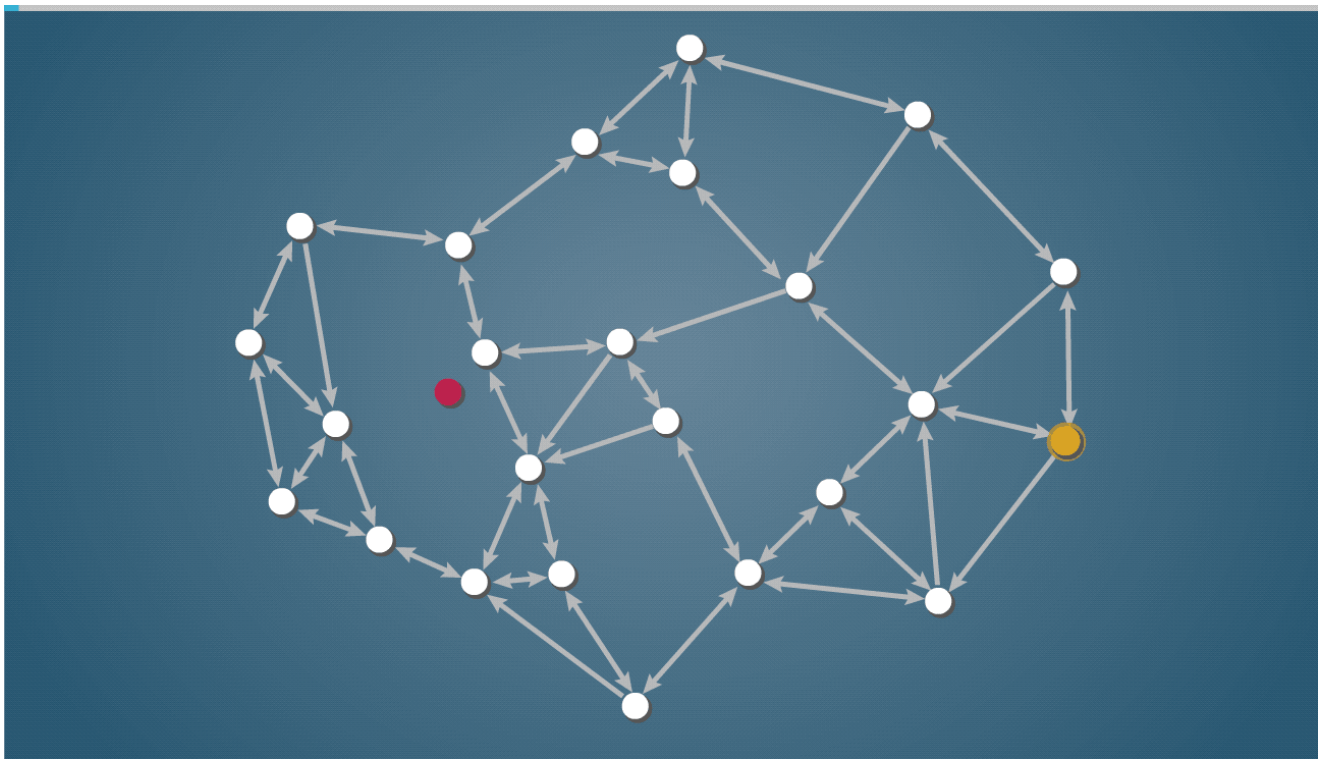
【更新】

undatePoint的算法逻辑


先更新原节点的原始向量
因为原始向量改变了，
所以原先的距离都发生了改变
接下来开始更新节点网络，优化连接结构
全图是否就这一个点
Y
return
循环遍历每一层
遍历完
修复更新操作
本层存在
获取本层更新点的
直接近邻点集
listOneHop
是否listOneHop为空
(没有任何近邻点)
Y
N
有直接近邻点
初始化本层的候选邻居集sCand
并加入更新点本身
初始化本层的决定更新集sNeigh
遍历直接节点集listOneHop的
所有节点
直接近邻点集还有元素没有遍历
该直接近邻点加入候选邻居集sCand
N
概率分支
全部直接近邻点都遍历结束
Y
该直接近邻点加入决定更新集sNeigh
该直接近邻点的近邻点（即二级间接近邻点）加入候选邻居集sCand
red：更新点本身
blue：直接近邻点
green：二级间接近邻点
sCand：全部直接近邻点和部分二级间接邻近点，以及更新点本身
sNeigh：部分直接近邻点
遍历决定更新集sNeigh
这里要强调：这里的sNeigh指的是决定更新这个集合里的节点的近邻表
本层sNeigh全部遍历完，
进入下一层的遍历
选定更新集的一个节点neight
初始化优先级队列候选集candidates
（更新集节点的候选集）
遍历sCand中所有的节点（包括更新点
本身）
存在未遍历的cand
计算该neight和cand的距离
更新优先级队列candidates候选集
(策略是越小越靠前)
(长度限制：最长是ef)construction_)
遍历结束



【查询】



func searchBaseLayer - 指定层进行查询

- 参数
  - `ep_id` : 入口点 (entry point) 的内部 ID (tableint)
  - `data_point` : 查询点
  - `layer` : 层级 (int)

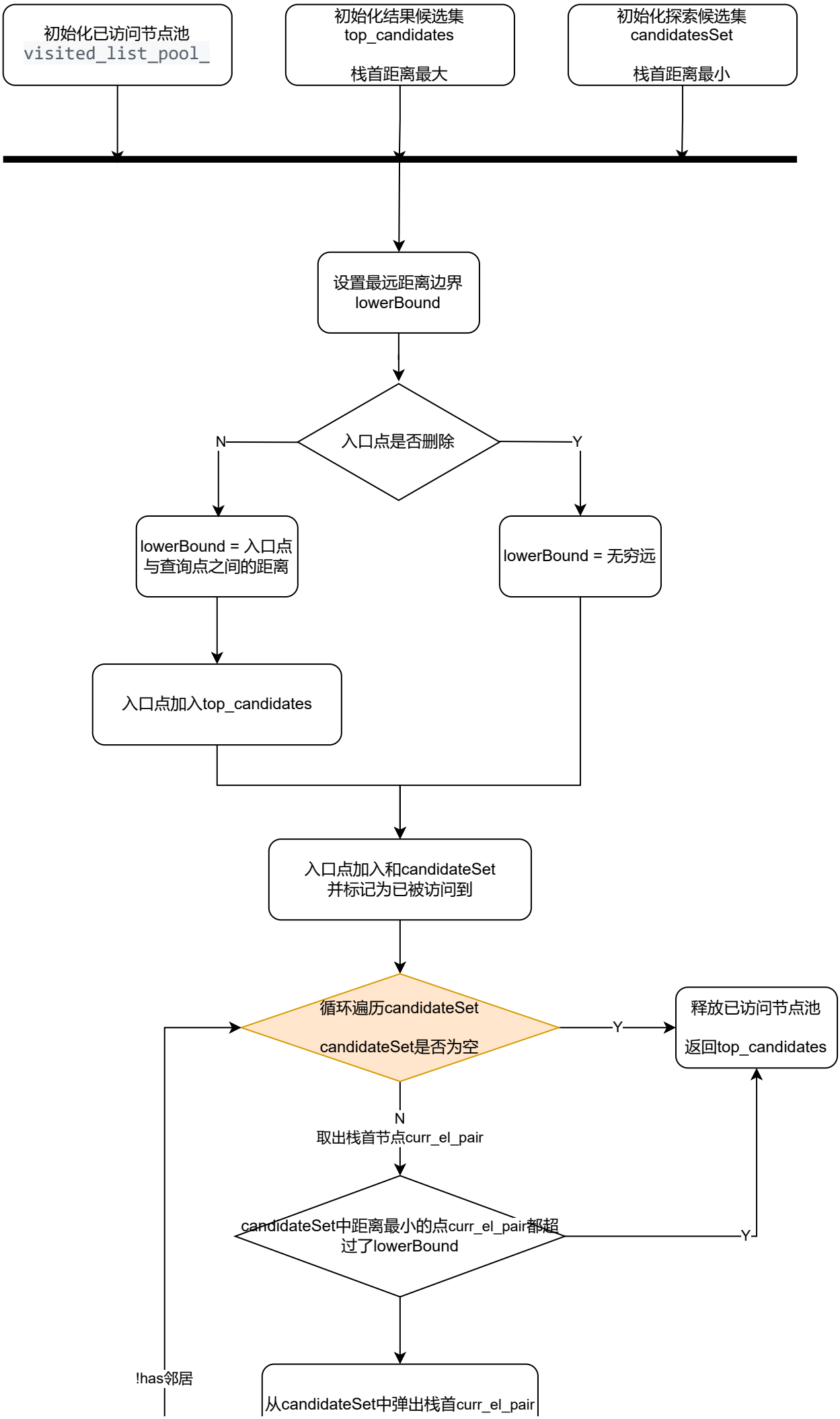
```
std::priority_queue<std::pair<dist_t, tableint>,
                    std::vector<std::pair<dist_t, tableint>>, CompareByFirst>
  searchBaseLayer(tableint ep_id, const void *data_point, int layer) {

    //初始化（刷新）已访问节点池
    VisitedList *vl = visited_list_pool_->getFreeVisitedList();
    vl_type *visited_array = vl->mass;  //已访问数组
    vl_type visited_array_tag = vl->curV;  //已访问数组标记值

    std::priority_queue<std::pair<dist_t, tableint>,
                        std::vector<std::pair<dist_t, tableint>>,
                        CompareByFirst>
```

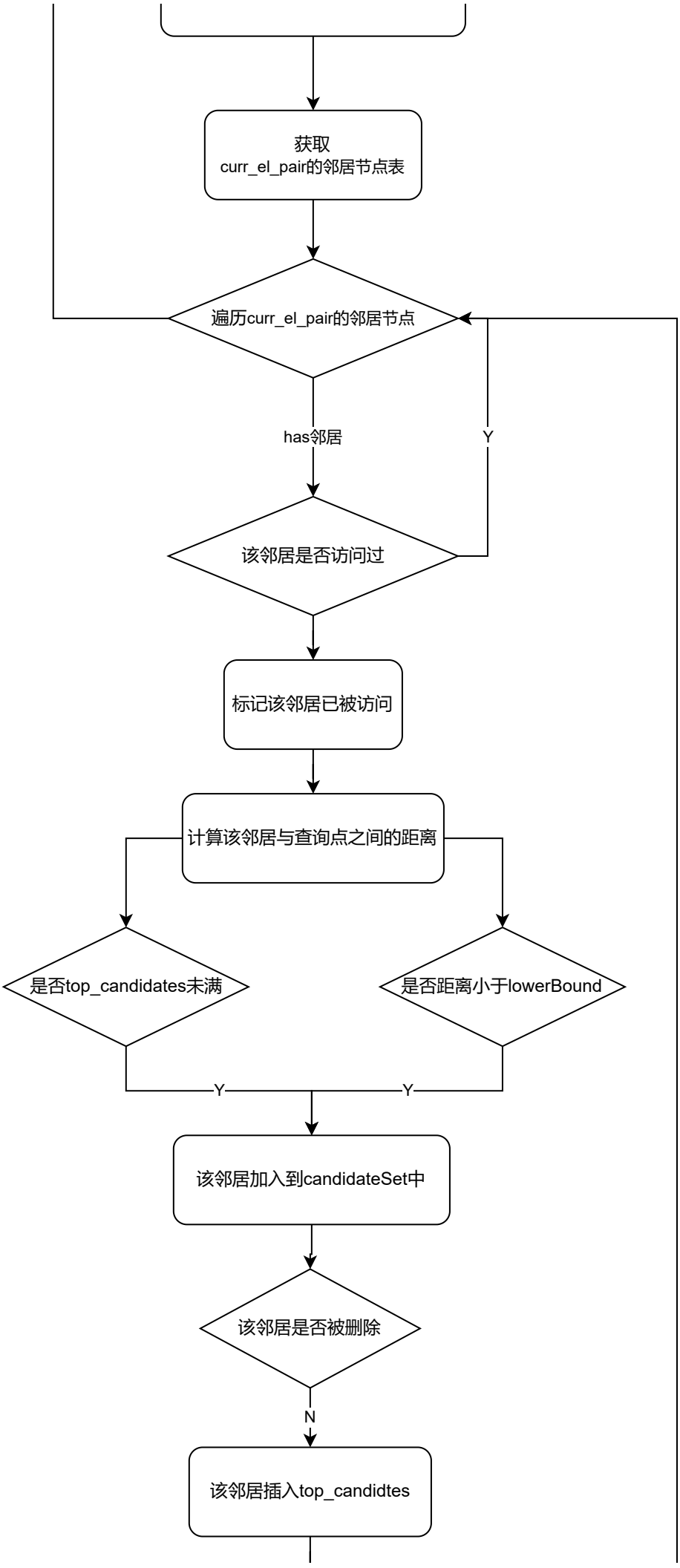
```
        top_candidates;  //候选集，要返回的结果就是它（返回ef_construction_ 个最近邻居）
    std::priority_queue<std::pair<dist_t, tableint>,
                        std::vector<std::pair<dist_t, tableint>>,
                        CompareByFirst>
        candidateSet;  //可以叫它候选集的候选集，存储所有待探索的候选邻居

    dist_t lowerBound;  //设置最远距离边界

    //首先考虑查询点和入口之间的距离
    if (!isMarkedDeleted(ep_id)) {
      //如果入口未被删除，计算查询点和入口点的距离dist，
      //并将入口加入top_candidates和candidateSet队列
      dist_t dist = fstdistfunc_(data_point, getDataByInternalId(ep_id),
                                 dist_func_param_);
      top_candidates.emplace(dist, ep_id);
      lowerBound = dist;
      candidateSet.emplace(-dist, ep_id);
    } else {
      //如果入口标记被删除了，那么最远距离边界就是无限远（无限大）
      lowerBound = std::numeric_limits<dist_t>::max();
      candidateSet.emplace(-lowerBound, ep_id);
    }
    visited_array[ep_id] = visited_array_tag;  //标记入口已经被访问过了

    while (!candidateSet.empty()) {  //循环遍历直到candidateSet为空
      std::pair<dist_t, tableint> curr_el_pair = candidateSet.top();//取出candidateSet里距离最近的节点（因为std::priority_queue是越大越靠前，存入时取负了，top反而是距离最小的，这样能节约比对成本）
      if ((-curr_el_pair.first) > lowerBound &&
          top_candidates.size() == ef_construction_) {
        break;  //最短距离的节点都比lowerBound远，剩下的就可以不用比了，除非top_candidates每满就还能“滥竽充数”
      }


      candidateSet.pop();  //移除该候选节点

      tableint curNodeNum = curr_el_pair.second;  //获取该候选节点的内部id

      std::unique_lock<std::mutex> lock(link_list_locks_[curNodeNum]);  //锁住该候选节点的邻居跳表

          //获取该候选节点的邻居跳表，get_linklist
      int *data;
      if (layer == 0) {
        data = (int *)get_linklist0(curNodeNum);
      } else {
        data = (int *)get_linklist(curNodeNum, layer);
```

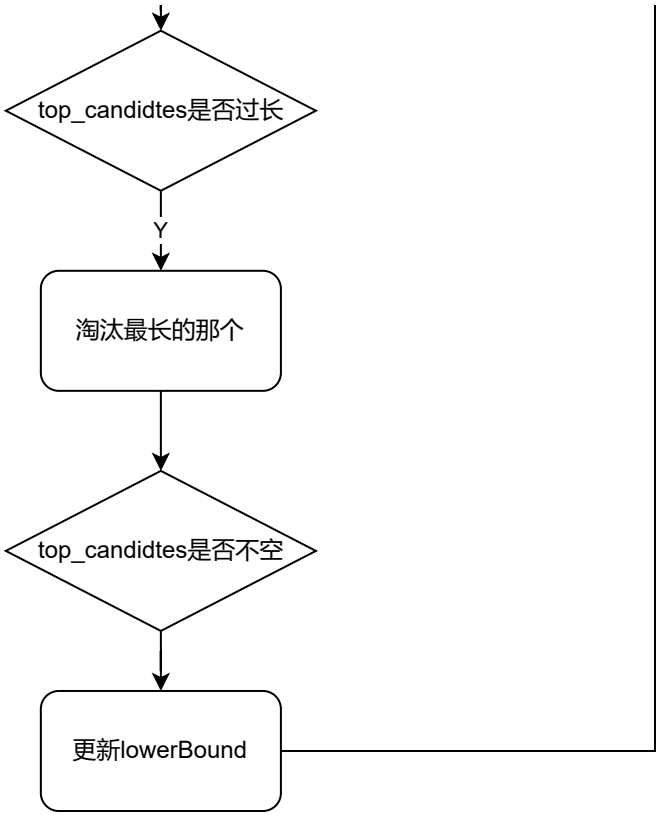
```
        }
        size_t size = getListCount((linklistsizeint *)data);
        tableint *datal = (tableint *)(data + 1);


#ifdef USE_SSE
        _mm_prefetch((char *)(visited_array + *(data + 1)), _MM_HINT_T0);
        _mm_prefetch((char *)(visited_array + *(data + 1) + 64), _MM_HINT_T0);
        _mm_prefetch(getDataByInternalId(*datal), _MM_HINT_T0);
        _mm_prefetch(getDataByInternalId(*(datal + 1)), _MM_HINT_T0);
#endif

        //遍历该候选节点的邻居
        for (size_t j = 0; j < size; j++) {
          tableint candidate_id = *(datal + j);
                  //如果这个邻居访问过了就跳过
                  if (visited_array[candidate_id] == visited_array_tag)
            continue;

          //否则就标记这个邻居已经访问过了，然后访问它
          visited_array[candidate_id] = visited_array_tag;
          char *currObj1 = (getDataByInternalId(candidate_id));

          //计算这个邻居与查询点之间的距离
          dist_t dist1 = fstdistfunc_(data_point, currObj1, dist_func_param_);
          if (top_candidates.size() < ef_construction_ || lowerBound > dist1) {
            candidateSet.emplace(-dist1, candidate_id); //如果距离小于lowerBound或者
top_candidates不满，就把这个邻居加入candidateSet中

            if (!isMarkedDeleted(candidate_id))  //如果这个邻居没有被删除，就无序插入
top_candidates（自动排序的）
              top_candidates.emplace(dist1, candidate_id);

            if (top_candidates.size() > ef_construction_)  //如果长度超过了预设就必须
删除一个（一次只会超出一个/末尾淘汰制）
              top_candidates.pop();

            if (!top_candidates.empty())
              lowerBound = top_candidates.top().first;  //缩小lowerBound(缩到
top_candidates里距离最长的那个)
          }

        }
      }


      visited_list_pool_->releaseVisitedList(vl);  //释放已访问节点池

      return top_candidates;  //返回top_candidates
```

```
    }
```

searchBaseLayer的算法逻辑


初始化已访问节点池
visited_list_pool_
初始化结果候选集
top_candidates
栈首距离最大
初始化探索候选集
candidatesSet
栈首距离最小
设置最远距离边界
lowerBound
入口点是否删除
N
Y
lowerBound = 入口点
与查询点之间的距离
lowerBound = 无穷远
入口点加入top_candidates
入口点加入和candidateSet
并标记为已被访问到
循环遍历candidateSet
candidateSet是否为空
Y
释放已访问节点池
返回top_candidates
N
取出栈首节点curr_el_pair
candidateSet中距离最小的点curr_el_pair都超
过了lowerBound
Y
!has邻居
从candidateSet中弹出栈首curr_el_pair


获取
curr_el_pair的邻居节点表
遍历curr_el_pair的邻居节点
has邻居
Y
该邻居是否访问过
标记该邻居已被访问
计算该邻居与查询点之间的距离
是否top_candidates未满
是否距离小于lowerBound
Y
Y
该邻居加入到candidateSet中
该邻居是否被删除
N
该邻居插入top_candidtes



func searchBaseLayerST <是否考虑被删除的元素，是否考虑过滤元素> - 指定策略进行查询

- 泛型：
  - 是否考虑被删除的元素：若为true则考虑被删除的元素
  - 是否考虑过滤元素：若为true，将以参数*isIdAllowed为指标过滤元素（这个id是否允许）
- 参数：
  - `ep_id`：入口点 (entry point) 的内部 ID (tableint)
  - `data_point`：查询点
  - `ef_`： 返回长度
  - `*isIdAllowed`：指标过滤元素（这个id是否允许）
- 这个方法无法指定层进行查询（只能在最底层查询）

```
template <bool has_deletions, bool collect_metrics = false>
  std::priority_queue<std::pair<dist_t, tableint>,
                      std::vector<std::pair<dist_t, tableint>>, CompareByFirst>
  searchBaseLayerST(tableint ep_id, const void *data_point, size_t ef,
                    BaseFilterFunctor *isIdAllowed = nullptr) const {
    //获取已访问池
    VisitedList *vl = visited_list_pool_->getFreeVisitedList();
    vl_type *visited_array = vl->mass;  //访问队列
    vl_type visited_array_tag = vl->curV;  //访问标志

    std::priority_queue<std::pair<dist_t, tableint>,
```

```
                    std::vector<std::pair<dist_t, tableint>>,
                    CompareByFirst>
    top_candidates;  //候选集
std::priority_queue<std::pair<dist_t, tableint>,
                    std::vector<std::pair<dist_t, tableint>>,
                    CompareByFirst>
    candidate_set;  //候选集的候选集（还有哪些元素待查）

dist_t lowerBound;  //最远距离边界

//入口点是否被删，如果被删是否允许参与查询
//是否有过滤指标，入口点是否被允许
if ((!has_deletions || !isMarkedDeleted(ep_id)) &&
    ((!isIdAllowed) || (*isIdAllowed)(getExternalLabel(ep_id)))) {

  //如果满足，步骤类似searchBaseLayer
  dist_t dist = fstdistfunc_(data_point, getDataByInternalId(ep_id),
                             dist_func_param_);
  lowerBound = dist;
  top_candidates.emplace(dist, ep_id);
  candidate_set.emplace(-dist, ep_id);
} else {
  ////如果不满足，步骤类似searchBaseLayer
  lowerBound = std::numeric_limits<dist_t>::max();
  candidate_set.emplace(-lowerBound, ep_id);
}

visited_array[ep_id] = visited_array_tag;  //标记访问过

//以下操作和searchBaseLayer几乎一样，只是多加了一个对节点的策略过滤
while (!candidate_set.empty()) {
  std::pair<dist_t, tableint> current_node_pair = candidate_set.top();

  if ((-current_node_pair.first) > lowerBound &&
      (top_candidates.size() == ef || (!isIdAllowed && !has_deletions))) {
    break;
  }
  candidate_set.pop();

  tableint current_node_id = current_node_pair.second;
  int *data = (int *)get_linklist0(current_node_id);
  size_t size = getListCount((linklistsizeint *)data);


  for (size_t j = 1; j <= size; j++) {
    int candidate_id = *(data + j);

    if (!(visited_array[candidate_id] == visited_array_tag)) {
      visited_array[candidate_id] = visited_array_tag;
```

```
          char *currObj1 = (getDataByInternalId(candidate_id));
          dist_t dist = fstdistfunc_(data_point, currObj1, dist_func_param_);

          if (top_candidates.size() < ef || lowerBound > dist) {
            candidate_set.emplace(-dist, candidate_id);


            if ((!has_deletions || !isMarkedDeleted(candidate_id)) &&
                ((!isIdAllowed) ||
                 (*isIdAllowed)(getExternalLabel(candidate_id))))
              top_candidates.emplace(dist, candidate_id);

            if (top_candidates.size() > ef)
              top_candidates.pop();

            if (!top_candidates.empty())
              lowerBound = top_candidates.top().first;
          }
        }
      }
    }

    visited_list_pool_->releaseVisitedList(vl);
    return top_candidates;
  }
```

func searchKnn - knn全图搜索近邻点

- 参数:
  - query_data：查询点的数据 (const void *)
  - k：需要返回的最近邻个数 (size_t)
  - isIdAllowed：可选参数，用于过滤结果 (BaseFilterFunctor *)，searchBaseLayerST方法用得着

```
std::priority_queue<std::pair<dist_t, labeltype>>
searchKnn(const void *query_data, size_t k,
          BaseFilterFunctor *isIdAllowed = nullptr) const {

  //创建一个优先级队列：结果集
  std::priority_queue<std::pair<dist_t, labeltype>> result;

  //如果没有点就返回空集
  if (cur_element_count == 0)
    return result;
```

```
  tableint currObj = enterpoint_node_;  //从入口点开始作为currObj

  //计算入口点与查询点之间的距离，作为curdist
  dist_t curdist = fstdistfunc_(
      query_data, getDataByInternalId(enterpoint_node_), dist_func_param_);


  //自顶而下地查询
  for (int level = maxlevel_; level > 0; level--) {
    bool changed = true;

    while (changed) {
      changed = false;
      unsigned int *data;

      //找到这一层的currObj的邻居
      data = (unsigned int *)get_linklist(currObj, level);
      int size = getListCount(data);
      metric_hops++;
      metric_distance_computations += size;

      tableint *datal = (tableint *)(data + 1);

      //遍历currObj的邻居，找到更近的点
      for (int i = 0; i < size; i++) {
        tableint cand = datal[i];
        dist_t d = fstdistfunc_(query_data, getDataByInternalId(cand),
                                dist_func_param_);

        if (d < curdist) {
          curdist = d;
          currObj = cand;  //如果邻居更近，就让邻居成为currObj
          changed = true;  //表示当前层是否有可能找到更近的邻居，如果是false表明
currObj的邻居都比currObj相较于查询点更远
        }
      }
    }

  }

  //最终得到了level1层的近邻入口，接下来开始搜索最底层level0

  std::priority_queue<std::pair<dist_t, tableint>,
                      std::vector<std::pair<dist_t, tableint>>,
                      CompareByFirst>
      top_candidates;

  //调用searchBaseLayerST方法在最底层进行查询，获得top_candidates
  if (num_deleted_) {
```

```
    top_candidates = searchBaseLayerST<true, true>(
        currObj, query_data, std::max(ef_, k), isIdAllowed);
  } else {
    top_candidates = searchBaseLayerST<false, true>(
        currObj, query_data, std::max(ef_, k), isIdAllowed);
  }

  //遴选top_candidates
  while (top_candidates.size() > k) {
    top_candidates.pop();
  }

  //规整result集合的格式：<距离，label>
  while (top_candidates.size() > 0) {
    std::pair<dist_t, tableint> rez = top_candidates.top();
    result.push(std::pair<dist_t, labeltype>(rez.first,
                                             getExternalLabel(rez.second)));
    top_candidates.pop();
  }
  return result;  //返回result
}
```

【多线程与并发锁】

hash 并发锁

label*op_locks*锁，这个锁会对 label_id & (65536-1) 进行hash 拿vector 里的mutex 锁，即限制同时65536 个索引可以并发构建

全局锁

global全局锁，这个锁仅在新增向量流程中，且所在层大于当前最大层时。因为涉及更新maxlevel*及相关向量在最新maxlevel*层的邻居配置，所以该锁会阻塞其他新增向量流程。

增量锁

label_lookup_lock 锁。当新增向量 or 查询向量时，用于锁<label,id>映射，临界区较小（其实就是一个set，然后加锁读写）

节点邻居表更新锁

link*list_locks*锁。每个向量都有一个，所以size=max_elements，用于更新自己的邻居表。由于hnsw 的精髓就在于通过向量的新增/更新，不断修正已有邻居向量的邻居以使得更加合理，所以这个锁不仅作用于新增索引的本身，而且作用于涉及的邻居节点（进行反更新）

删除element锁

- deleted_elements_lock 锁（涉及到删除element 才会使用）。

- 对`std::unordered_set<tableint> deleted_elements; // contains internal ids of deleted elements`删除过的`unordered_set`临界区维护
- HNSW的删除并非是物理内存意义上的删除，在第0层结构中提到了一个**删除标记**，本质上是打开了删除标记而已（flag=1）
- 之所以这样做是避免了邻居节点的更新导致的大范围内存的移动

> ⚠ **尚未完成的任务**
>
> - get_linklist方法
> - visited*list_pool*
> - repairConnectionsForUpdate
> - isMarkedDeleted
> - 启发式