-wiser-

这是关于全文搜索引擎wiser源码的讲解；

搜索引擎概述

搜索引擎是一类系统或软件的统称，作用是从文档的集合中查找（检索）出匹配信息需求（查询）的文档，信息需求是由单词、问题等构成的。

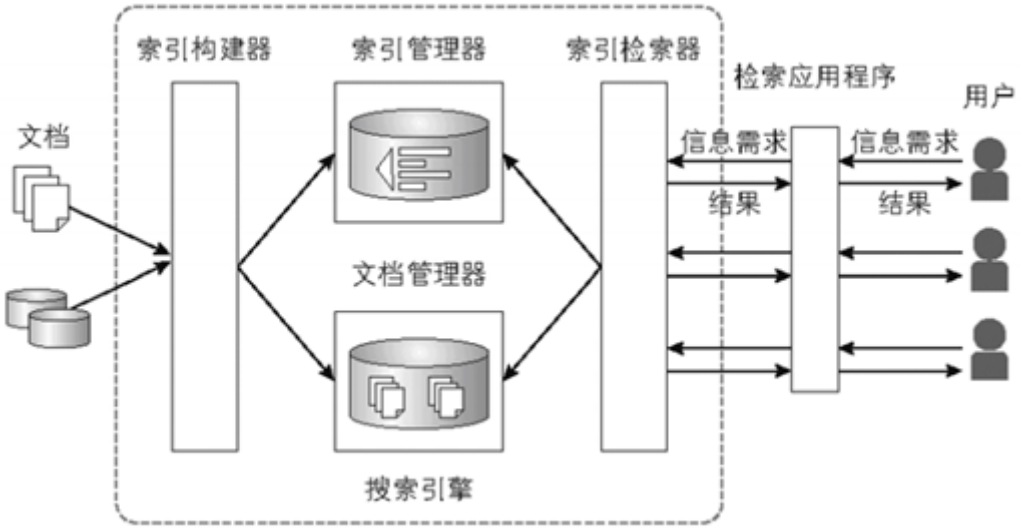
搜索引擎一般由以下 4 个组件构成：

l 索引管理器（Index Manager）

l 索引检索器（Index Searcher）

l 索引构建器（Indexer）

l 文档管理器（Document Manager）



索引管理器组件的作用是管理带有索引结构的数据，索引结构是一种用于进行高速检索的数据结构。对索引的访问也是通过索引管理器进行的。

索引检索器是利用索引进行全文搜索处理的组件。索引检索器根据来自检索应用程序用户的查询，协同索引管理器进行检索处理。在大多数情况下，索引检索器都会根据某种标准对与查询相匹配的检索结果排序，并将排在前面的结果返回给应用程序。

索引构建器是从作为检索对象的文本文档中生成索引的组件。索引构建器会先通过解析将文本文档分解为单词序列，然后再将该单词序列转换为索引结构。在搜索引擎中，将生成索引的环节称为索引构建。

文档管理器是管理文档数据库的组件，文档数据库中储存着作为检索对象的文档。文档管理器会先从文档数据库中取出与查询相匹配的文档，然后再根据需要从该文档中提取出一部分内容作为摘要。

倒排索引的结构

教材或专业书等图书后面的索引中，通常都会写有关键词（单词）和出现了该关键词的页码。由于关键词是按词典顺序排列的，所以查找时无需逐一浏览，只需按照拼音字母的顺序逐渐缩小查找范围，就能轻松地找到关键词。而只要再向这个关键词的旁边看一眼，就能立刻知道该关键词出 现在哪一页了。实际上，倒排索引具有与图书索引完全相同的逻辑结构。

下面就让我们以一本书中的文档为例来具体看看倒排索引吧。这本书由以下两页组成，内容分别如下所示。

第 1 页（P1）： I like search engines.

第 2 页（P2）： I search keywords in Google.

表 1-1 示例书籍中的倒排索引

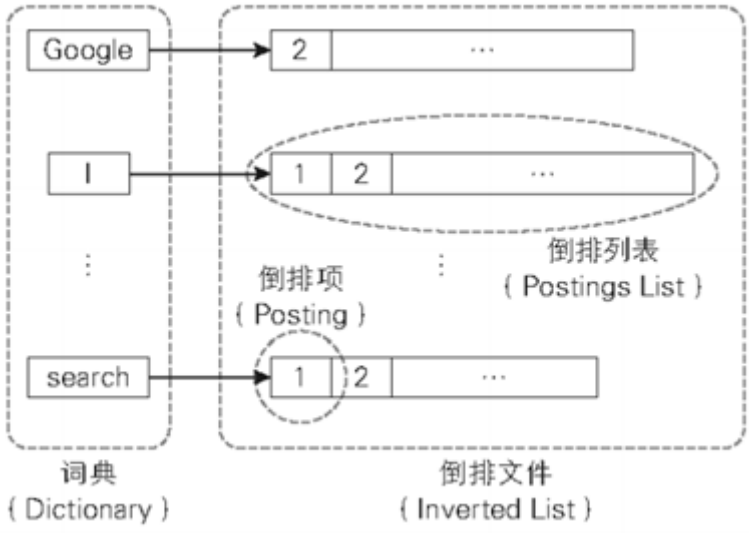
engine	P1
Google	P2
I	P1,P2
in	P2
keyword	P2
like	P1
search	P1, P2

将表示“在哪一页上使用了哪个单词”的表格转换为“哪个单词出现在了哪一页上”的表格，就可以制作出倒排索引了。另外，之所以称这种索引为倒排索引，是由于进行过的交换表格行、列的操作叫作“倒排”。

在生成的倒排索引中，我们建立起了页中的单词和页的对应关系。也就是说，我们是把页当成了构建索引的单位。之所以这样做，是因为在翻阅图书时，人们通常是以“页”作为单位的。

那么，对于其他情况又要如何处理呢？例如，对于 Web 上的 HTML文档，我们可以将 1 个 HTML网页作为构建索引的单位。

由此可见，对于每种作为检索对象的数据，构建索引的单位都是不同的。在全文搜索中，将构建索引的单位统称为“文档”（Document），将文档的标识信息称为“文档编号”。文档编号类 似图书的页码，用于唯一地标识某个文档。因此，也可以这样说，所谓倒排索引就是“把单词和单词所在文档的文档编号对应起来的表格”。 另外，在倒排索引中，将表示单词和文档编号对应关系的信息称为倒排项（Posting），将各个单词的倒排项的集合称为倒排列表（Postings List）。例如，在刚刚的倒排索引中，单词 search的倒排项是 P1 和 P2，倒排列表是 P1、P2 的集合，记作“P1,P2”。



若要从倒排索引中查找出包含了某个单词的文档，只需要先从词典中找到该单词，然后获取与之对应的倒排列表，最后从倒排列表中获取文档编号即可。

那么，我们又该如何查找同时包含了多个单词的文档呢？查找时只需要先从词典中找出各个单词，然后分别获取这些单词的倒排列表并加在一起，由此计算出包含在各个倒排列表中的 文档编号的交集。

除此以外，还有另一种倒排文件，称作“**单词级别的倒排文件**”（。这种倒排文件中不仅带有有关单词出现在了哪个文档中的信息，还带有单词出现在了文档中的什么位置（从开头数是第几个单词）这一信息。

在单词级别的倒排文件中，各个倒排项的表示方法如下所示。

DocID; offset1, offset2...

还是以刚刚使用过的两个文档为例，从头数的话，单词 search是文档 1（P1）中的第 3 个单词，是文档 2（P2）中的第 2 个单词，因此其倒排列表是

search: D1;3, D2;2

我们刚刚讲解的是如何利用文档级别的倒排文件查找同时包含 search和 engine 的文档。但是利用这种方法得到的检索结果，未必都是关于（search engine）的文档。例如，虽然，下面的文档也同样包含了 search和 engine，但却与搜索引擎无关。

I search for a gas station because my car's engine doesn't start.

（因为汽车的引擎发动不起来了，所以我要找加油站。）

因此，要想查找关于搜索引擎的文档，就需要从倒排索引中找出含有短语 search engine 的文档。而要想从倒排索引中查找短语，就需要使用刚刚介绍过的单词级别的倒排文件。

在使用单词级别的倒排文件查找短语时，前几步与使用文档级别的倒排列表相同，即也是先从词典中找出单词 search和 engine，然后分别获取它们的倒排列表，最后算出这两个倒排列表中文档编号的交集。但是到这里还没有结束，查找短语时还需要确认 search和 engine 是否是相邻出现的。

制作中文文档的倒排索引

在中文的句子中，由于各单词是词间不留空白连续书写的，所以就需要使用不同于英文的方法，才能将句子分割成单词或字符的序列。

若要将类似中文的句子，即单词无法通过空白划分出来的句子分割成单词序列，通常有以下两种方法。

l 词素解析分割法

l N-gram（q-gram）分割法

词素解析（Morphological Analysis）分割法是一种将句子分割为“词素”序列的方法。词素是语言中含有意义的最小单位。例如，如果使用词素解析分割法分割“全文搜索引擎”这段文本，那 么可以得到如下结果。

全文 搜索 引擎

由于中文的语法极其复杂，所以一般认为对中文句子正确地进行词素解析是件非常困难的事。近几年，在词素解析上，一般采用的是机器学习的方法。机器学习的过程是先学习由手工 作业正确分割句子后得到的数据，然后推理出应该如何分割未知的句子（以及如何标注词性等）。一般认为现代词素解析的精度已经非常高了，在大多数情况下，都能正确地判断出中文句子应该在哪里分割成词。不过，对于博客等环境中常用的含有大量口语表达的句子，精度还是会大幅下降的。

N-gram分割法是一种将句子分割成由 N 个字符组成的片段序列的方法，每个片段称作一个 N-gram。N 的取值通常为 2 或 3。N = 1 时称作 uni-gram（一元 gram），N = 2 时称作 bigram（二元 gram），N = 3 时称作 tri-gram（三元 gram）。例如，如果使用 bi-gram去分割“全文搜索引擎”这段文本，那么可以得到如下结果。

全文 文搜 搜索 索引 引擎

N-gram分割法作为一种不依赖具体语言的句子分割方法，广泛应用于以亚洲国家的语言为主的各种语言中。

由词素构成的倒排索引的优缺点：

与由 N-gram构成的倒排索引相比，由词素构成的倒排索引由于从文中分割出的词元数更少，所以词典和倒排文件的尺寸也就更小。由此就产生了高速进行构建处理和搜索处理的可能性。

不过这种倒排索引也存在缺点，即会发生所谓的“检索遗漏”问题。检索遗漏指的是，尽管查询实际就包含在文档中，但就是找不到与查询相匹配的内容。这是由查询与通过词素解析从文

中分割出的词素不一致导致的。例如，将“哆哆嗦嗦”分割成词素后还是“哆哆嗦嗦”，可如果检索的是“哆嗦”，就无法检索到包含“哆哆嗦嗦”的文档了。那些尚未收录在词素解析词典中的新词和以口语方式使用的单词也都面临同样的问题。

由 N-gram构成的倒排索引的优缺点：

与由词素构成的倒排索引不同，由 N-gram构成的倒排索引不会产生检索遗漏问题。也就是说，在由 N-gram构成的倒排索引中，基本上只要查询包含在文档中，就一定能找得到。要想能够检索到少于 N 个字符的字符串，通常需要事先制作由 M-gram（M< N ）构成的倒排索引。

但是，从刚刚的“全文搜索引擎”的例子中也能看出来，相比于词素解析，在同一个文档中使用 N-gram 产生的词元通常较多。因此，词典和倒排文件的尺寸自然也就更大，从而导致构建处理和搜索处理的速度下降。而且，由于 N-gram并不考虑单词的界限，所以在由 N-gram构成的倒排索引中，会发生检索“华山”，却也能找到包含“九华山”的文档这样的问题。

在开发搜索引擎的过程中，重要的是根据文档的特性灵活运用这两种分割方法。为了体现出运用上的灵活性，我们需要设计出不依赖句子分割方法的搜索引擎。例如，可以不以从文档的开头数是第几个词元为基础，而是以从文档的开头数是第几个字符为基础来构建倒排项。

实现倒排索引

使用二叉查找树实现词典时，要先将数据对（的列表）按照单词的词典顺序排列，然后存储到存储器中。数据对是由单词和对应着该单词的倒排列表的引用信息构成的。例如，若用内 存上的二叉查找树实现之前例子中的词典，就会得到如图 1-3 所示的树形结构。树中的各个结点是通过地址引用（指针）连接起来的。

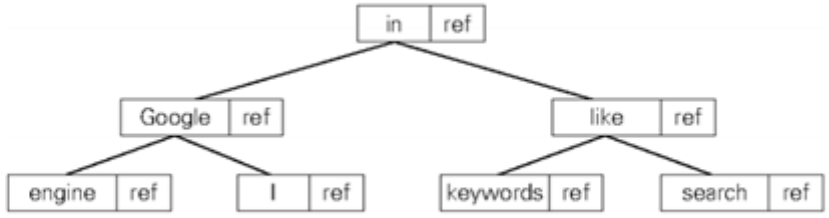
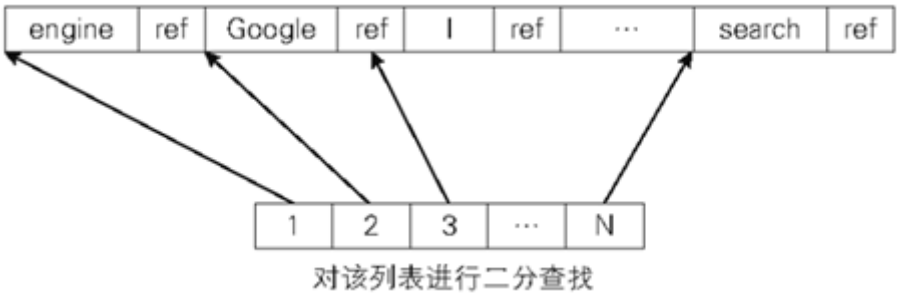
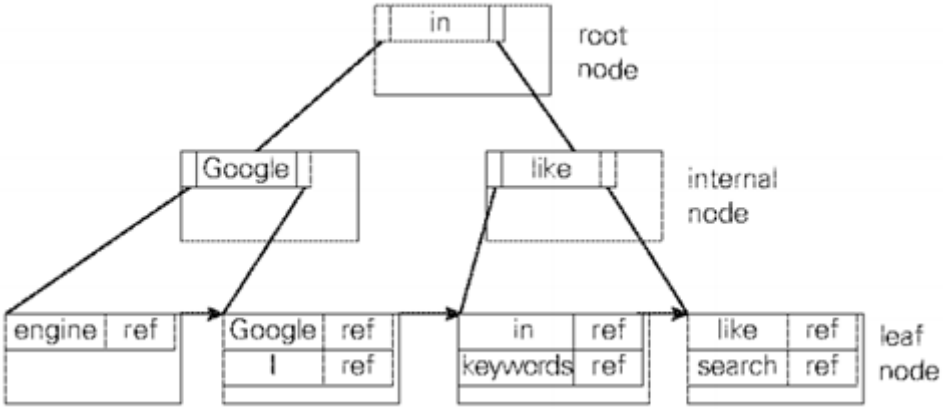


图 1-3　在内存上实现词典（使用二叉查找树）

同样地，在二级存储上实现词典时，也要先将数据对按照单词的词典顺序排列，然后一个接一个地存储到存储器上。但是，如果只是单纯地一个接一个地存储，就无法知道各数据对应 该在哪里结束了，因此在此之上还要维护一个列表，用于存储从开头算起每个数据对的偏移量。



B+ 树是一种平衡的多叉树，属于从 B 树派生出来的树形结构。在 B+ 树中，所有的记录都存储在树中的叶结点（Leaf Node）上，内部结点（Internal Node）上只以关键字的顺序存储关键字。



一般来说，使用**倒排索引的检索处理流程**如下；

首先，根据各自倒排列表长度的升序，对查询中的单词进行排序。之所以这样做是为了在对多个倒排列表两两计算交集的时候，尽可能地减少比较的次数。接下来，从查询中取出最前面的单词，获取与之对应的（最短的）倒排列表。

然后，依次计算该倒排列表与查询中剩余单词的倒排列表的交集，最终生成包含全部单词的倒排列表。

接下来，计算刚生成的倒排列表中的各文档与查询的关联度。此时，若还需要根据日期等属性值而非关联度对检索结果进行排序的话，则还需要从每个文档中获取相应的属性值。

最后，在按照关联度和属性值对检索结果进行排序后，取出检索结果中的前 k 个文档。在取出结果的时候，能将所有结果都提供给用户固然好，然而当结果数量过多时，通常 只提供根据某种标准选出的“优质结果”。

关联度的计算方法

在 Web 搜索引擎中，一般是按照文档与查询的关联度对检索结果进行排序的。计算关联度的方法有余弦相似度（Cosine Similarity）和 OkapiBM25 等。

在计算余弦相似度时，需要把文档和查询映射到以单词（Term）为维度的向量空间上，文档向量和查询向量的夹角（内积）越小，说明文档和查询的关联度越高。

而 OkapiBM25 则是基于“文档是否匹配查询是由概率决定的”这一统计原理，根据单词的出现频率等因素计算出查询与文档相关联的概率，这个概率越大，说明文档和查询的关联度越高。

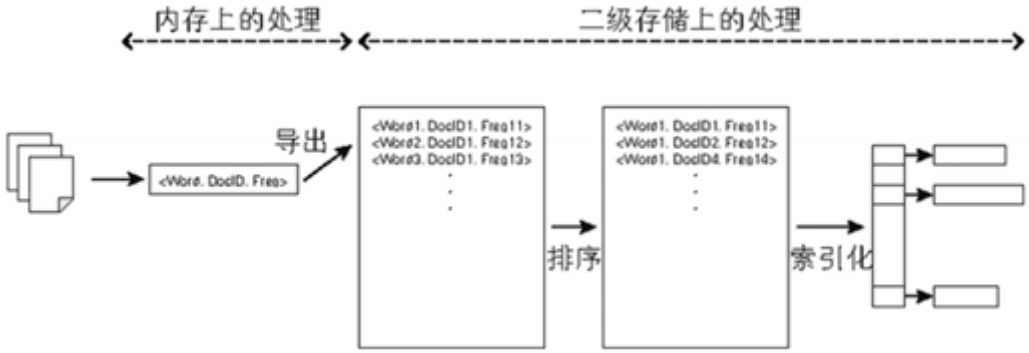
基于排序的索引构建法

首先，对各文档中构成该文档的每个单词都建立一条形如“单词、文档编号、单词在文档中的出现次数（TF）”的记录，然后将该记录写入到二级存储上的文件的末尾（❶）。

接下来，将文件中的各条记录优先按照单词的升序排列，单词字段相同的记录需再按照文档编号的升序排列（❷）。

这时的排序方法通常会选择合并排序，例如像下面这样做：首先以块的整数倍为单位将多条记录加载到内存中，然后对这些记录进行快速排序并将排序后的记录导出到文件中，最后利用多路合并排序将多个导出的文件合并在一起。

最后，从第一行开始逐行地读取排序后的文件，取出每个单词的文档编号的列表，并用这些列表构建出各个单词的倒排列表（❸）。此外，压缩倒排列表的操作也是在❸的步骤中进行。



基于合并的索引构建法

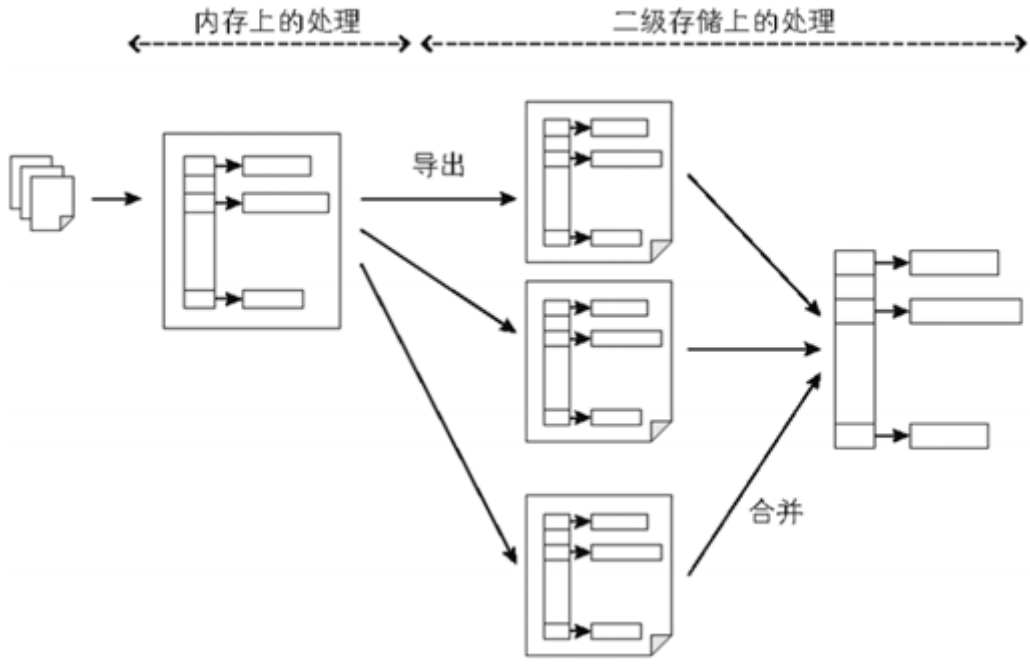
基于合并的索引构建法是一种先在内存上构建出倒排索引的片段，然后将这些片段导出到二级存储，最后将导出的多个倒排索引片段合并在一起，以此来构建最终的倒排索引的方法。

首先，在内存上构建出以单词为键，以倒排列表为值的映射表（Mapping），即由部分数据构成的倒排索引的片段（❶）。

每当遇到文档中的单词不在映射表中时，都要将该单词加入到映射表中（❷）。

当映射表的大小（事先已设定好）达到内存大小的上限时，就将该映射表导出到文件中（❸）。

像这样反复处理，直到处理完所有的文档，最后利用多路合并将导出的多个文件合并在一起，构建出最终的倒排索引（❹）。另外，压缩倒排列表的操作也是在❹的步骤中进行。



构建倒排索引

提取词元

先来简单地复习一下构建倒排索引的步骤吧。

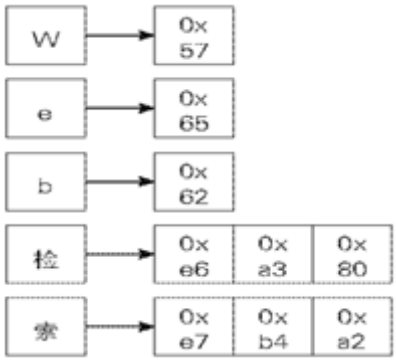
l 从作为检索对象的文档中提取出词元及其出现的位置；

l 对于每个词元，将其所在文档的引用信息（文档编号）和出现在文档中的位置保存起来；

以上就是构建倒排索引的过程。在这个过程中，我们还需要利用 N-gram或词素解析的方法将句子分割成词元的序列。

从句子中分割出词元的处理看似简单，但是在处理中文时，还是要稍加注意才行。之所以这样说是因为Wikipedia 的词条都是用 UTF-8 的字符编码表示的，因此在进行处理时，不得不考虑这种字符编码的特性。

在 UTF-8 中，是用 1 到 4 个字节的长度来表示 1 个字符的。例如，像数字和拉丁字母等在英文中使用的字符都是用 1 个字节表示的，而在中文中使用的字符则多半要用 3 个字节才能表示。因此，在 UTF-8 中，“Web 检索”这个字符串就是由如下所示的含有 9 个字节的字节序列表示的。



在 wiser 中，为了避开由使用 UTF-8 带来的处理上的麻烦，我们在每次获取 N-gram时，都会先将字符串的编码从 UTF-8 转换成 UTF-32。UTF-32 是一种以 4 字节（32 bit）的数值为单位表示 Unicode 字符的编码方式。由于 Unicode 的字符与表示该字符的数值是一一对应的，所以在 UTF-32 中，由 N-gram分割而成的词元所含有的字节数就变成固定的了，这样就简化了程序上的处理过程。

构建倒排索引

由于用作样本数据的 Wikipedia 的文档相对较大，所以只在内存上为所有文档构建倒排索引并不现实。因此，用 wiser 构建倒排索引时，我们还要使用硬盘或 SSD 等存储器 （二级存储器）。

在 wiser 中，我们将采用一种新方法构建倒排索引，该方法源自在第 1 章中讲解过的基于归并的构建方法。大致描述一下这个方法的话，就是对于某个文档集合，先在内存上为其建立一个较小的倒排索引，然后将这个较小的倒排索引和存储器上的倒排索引合并，通过反复进行这两步操作，最终就能一点点地在存储器上构建出较大的倒排索引了。其实如果想要在存储器上创建倒排列表，最直接的方法就是不断地将倒排项（文档编号和位置信息）添加到存储器上的倒排列表的末尾。

倒排列表和倒排文件的数据结构如下：

```
/* 倒排列表（以文档编号和位置信息为元素的链表结构）*/
typedef struct _postings_list {
  int document_id;              /* 文档编号 */
  UT_array *positions;          /* 位置信息的数组 */
  int positions_count;          /* 位置信息的条数 */
  struct _postings_list *next; /* 指向下一个倒排列表的指针 */
} postings_list;

/* 倒排索引（以词元编号为键，以倒排列表为值的关联数组） */
typedef struct {
  int token_id;                  /* 词元编号（Token ID）*/
  postings_list *postings_list; /* 指向包含该词元的倒排列表的指针 */
  int docs_count;                /* 出现过该词元的文档数 */
  int positions_count;           /* 该词元在所有文档中的出现次数之和 */
```

```
  UT_hash_handle hh;            /* 用于将该结构体转化为哈希表 */
} inverted_index_hash, inverted_index_value;
```

虽然一边遍历倒排列表的链表，一边计数也能统计出出现过某个词元的docs_count，但是若每次检索时都要统计一遍的话就会影响效率。因此，在构建索引的阶段，我们要先将统计好的文档数存储起来。同理，还要将词元的出现次数（positions_count）也一并存储起来。

函数add_document()

在 wiser 中，我们首先调用了函数 add_document()，该函数的作用是为文档的标题和正文构建倒排索引以及用于存储文档的数据库。在函数 add_document() 内部会进行如下的处理。

l 从文档中取出词元；

l 为每个词元建立倒排列表，并更新小倒排索引；

l 每当小倒排索引增长到一定大小，就将其与存储器上的倒排索引合并到一起；

```
static void add_document(wiser_env *env, const char *title, const char *body)
{
  if (title && body) {
    UTF32Char *body32;
    int body32_len, document_id;
    unsigned int title_size, body_size;

    title_size = strlen(title);
    body_size = strlen(body);

    /* 将文档存储到数据库中并获取该文档对应的文档编号 */
    db_add_document(env, title, title_size, body, body_size);
    document_id = db_get_document_id(env, title, title_size);

    /* 转换文档正文的字符编码 */
    if (!utf8toutf32(body, body_size, &body32, &body32_len)) {
      /* 为文档创建倒排列表 */
      text_to_postings_lists(env, document_id, body32, body32_len,
                             env->token_len, &env->ii_buffer);
      env->ii_buffer_count++;
      free(body32);
    }
    env->indexed_count++;
    print_error("count:%d title: %s", env->indexed_count, title);
  }

  /* 存储在缓冲区中的文档数量达到了指定的阈值时，更新存储器上的倒排索引 */
  if (env->ii_buffer &&
      (env->ii_buffer_count > env->ii_buffer_update_threshold || !title)) {
    inverted_index_hash *p;

    print_time_diff();

    /* 更新所有词元对应的倒排项 */
    for (p = env->ii_buffer; p != NULL; p = p->hh.next) {
      update_postings(env, p);
    }
    free_inverted_index(env->ii_buffer);
```

```
    print_error("index flushed.");
    env->ii_buffer = NULL;
    env->ii_buffer_count = 0;

    print_time_diff();
  }
}
```

首先，我们将标题和正文存储到了用于存储文档的数据库中。

由于 SQLite 会自动为存储到数据库中的记录分配 ID，所以我们就把这个 ID 用作文档编号。

接下来的步骤中，我们通过调用函数 text_to_postings_lists()，并根据文档编号（document_id）和文档内容（body32），更新了存储在变量 env->ii_buffer 中的小倒排索引。

然后我们要判断是否需要合并索引。当 title 为 NULL时，或者当已构建出小倒排索引的文档数量达到了阈值（env->ii_buffer_update_threshold）时，就合并索引。另外，title 为 NULL还标志着所有的文档都已经处理完了。

env->ii_buffer_update_threshold 是一个阈值，决定了将多少个文档存储到小倒排索引中之后，就需要将小倒排索引与存储器上的倒排索引合并了。该阈值设定得越小，内存的使用量也就越小，但是这样会增加对存储器的访问次数。反过来，该阈值设定得越大，内存的使用量也就越大，但是这样能减少对存储器的访问次数。

最后在的步骤中我们通过调用函数 update_postings() 合并了倒排索引，并将合并后的结果写入数据库（存储器）中。

函数 text_to_postings_lists()

该函数的作用是为文档编号和构成文档内容的字符串建立倒排列表的集合（倒排文件）。

```
/**
 * 为构成文档内容的字符串建立倒排列表的集合
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] document_id 文档编号。为0时表示把要查询的关键词作为处理对象
 * @param[in] text 输入的字符串
 * @param[in] text_len 输入的字符串的长度
 * @param[in] n N-gram中N的取值
 * @param[in,out] postings 倒排列表的数组（也可视作是指向小倒排索引的指针）。若传入的指针指向了NULL，
 *                         则表示要新建一个倒排列表的数组（小倒排索引）。若传入的指针指向了之前就已经存在的倒排列表的数组，
 *                         则表示要添加元素
 * @retval 0 成功
 * @retval -1 失败
 */
int text_to_postings_lists(wiser_env *env,
                           const int document_id, const UTF32Char *text,
                           const unsigned int text_len,
                           const int n, inverted_index_hash **postings)
{
  /* FIXME: now same document update is broken. */
  int t_len, position = 0;
  const UTF32Char *t = text, *text_end = text + text_len;

  inverted_index_hash *buffer_postings = NULL;

  for (; (t_len = ngram_next(t, text_end, n, &t)); t++, position++) {//6
```

```
    /* 检索时，忽略掉由t中长度不足N-gram的最后几个字符构成的词元 */
    if (t_len >= n || document_id) {
      int retval, t_8_size;
      char t_8[n * MAX_UTF8_SIZE];

      utf32toutf8(t, t_len, t_8, &t_8_size);//7

      retval = token_to_postings_list(env, document_id, t_8, t_8_size,
                                      position, &buffer_postings);//8
      if (retval) { return retval; }
    }
  }

  if (*postings) {//9
    merge_inverted_index(*postings, buffer_postings);//10
  } else {
    *postings = buffer_postings;
  }

  return 0;
}
```

首先，我们通过调用位于 token.c 中的函数 ngram_next()，从字符串 t 中取出了一个 N-gram，同时还获取了词元的长度 t_len和指向其首地址的指针 t（❻）。

接下来要做的是为由函数 ngram_next() 返回的每个词元创建倒排列表。在创建过程中，我们首先将词元的字符编码由 UTF-32 转换成了 UTF-8（❼），随后通过调用函数

token_to_postings_list()，将该词元添加到倒排列表中（❽）。有关函数 token_to_postings_list() 的讲解我们也先放到后面。

让我们继续往下读，当❻的循环一结束，仅由传入本函数的文档构成的倒排索引也就构建出来了。为了便于理解，我们再来看一个具体的例子。假设有这样一个文档，文档编号是 10，正文的内容是“自制搜索引擎”。将该文档传递给函数 text_to_postings_lists() 后，就可以构建倒排索引。

构建出倒排索引以后，如果已经存在小倒排索引了（❾），就调用函数 merge_inverted_index() 将其与刚刚构建出的倒排索引合并（❿）。反之，如果还没有小倒排索引，就将刚刚构建出的倒排索引作为小倒排索引。有关函数 token_to_postings_list() 的讲解同样也先放到后面。

函数 ngram_next()

在 wiser 中，由文件 token.c 中的函数 ngram_next() 负责将句子分割成词元。

```
/**
 * 将输入的字符串分割为N-gram
 * @param[in] ustr 输入的字符串（UTF-8）
 * @param[in] ustr_end 输入的字符串中最后一个字符的位置
 * @param[in] n N-gram中N的取值。建议将其设为大于1的值
 * @param[out] start 词元的起始位置
 * @return 分割出来的词元的长度
 */
static int ngram_next(const UTF32Char *ustr, const UTF32Char *ustr_end,
           unsigned int n, const UTF32Char **start)
{
  int i;
  const UTF32Char *p;

  /* 读取时跳过文本开头的空格等字符 */
  for (; ustr < ustr_end && wiser_is_ignored_char(*ustr); ustr++) {//11
  }

  /* 不断取出最多包含n个字符的词元，直到遇到不属于索引对象的字符或到达了字符串的尾部 */
  for (i = 0, p = ustr; i < n && p < ustr_end
       && !wiser_is_ignored_char(*p); i++, p++) {//12
  }

  *start = ustr;
  return p - ustr;
}
```

在读取构成词元的字符时，我们首先跳过了文本开头的空格等不属于索引对象的字符（⓫）。这里使用的是空循环，因此可以使用";"作为 for 语句的循环体，但是为了易于理解，我们还是写成了空语句块"{}"的形式。函数 wiser_is_ignored_char() 的作用是判断给定的字符是否不属于检索对象。

接下来，通过⓬中的 for 语句，我们从文本中取出了 n个字符。虽然这也是个空循环，**但是循环条件并不简单，在循环时既要考虑不属于索引对象的字符，还要防止指针 p 超出字符串的末尾。**

函数 token_to_postings_list()

下面再来看一下在函数 text_to_postings_lists() 中被调用的函数 token_to_postings_list()。函数 token_to_postings_list() 的作用是为文档中的一个词元创建倒排列表。

```
/**
 * 为传入的词元创建倒排列表
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] document_id 文档编号
 * @param[in] token 词元（UTF-8）
 * @param[in] token_size 词元的长度（以字节为单位）
 * @param[in] position 词元出现的位置
 * @param[in,out] postings 倒排列表的数组
 * @retval 0 成功
 * @retval -1 失败
 */
int token_to_postings_list(wiser_env *env,
                        const int document_id, const char *token,
                        const unsigned int token_size,
                        const int position,
                        inverted_index_hash **postings)
{
  postings_list *pl;
  inverted_index_value *ii_entry;
  int token_id, token_docs_count;

  token_id = db_get_token_id(
               env, token, token_size, document_id, &token_docs_count);//13
  if (*postings) {//14
    HASH_FIND_INT(*postings, &token_id, ii_entry);//15
  } else {
    ii_entry = NULL;//16
  }
  if (ii_entry) {//17
    pl = ii_entry->postings_list;//18
    pl->positions_count++;//19
  } else {
    ii_entry = create_new_inverted_index(token_id,
                                         document_id ? 1 :
token_docs_count);//20
    if (!ii_entry) { return -1; }
    HASH_ADD_INT(*postings, token_id, ii_entry);//21

    pl = create_new_postings_list(document_id);//22
    if (!pl) { return -1; }
    LL_APPEND(ii_entry->postings_list, pl);//23
  }
  /* 存储位置信息 */
```

```
  utarray_push_back(pl->positions, &position);//24
  ii_entry->positions_count++;//25
  return 0;
}
```

首先，我们通过调用函数 db_get_token_id()，获取了词元对应的编号（⓭）。如果之前已将编号分配给了该词元，那么在此处获取的正是这个编号；反之，如果之前没有分配编号，那么函数 db_get_token_id() 会为该词元分配一个新的编号。

再往下看，如果存在已经构建好的小倒排索引（⓮），那么我们就从中获取关联到该词元编号上的倒排列表（⓯）。在获取时，我们以 token_id 为键调用了HASH_FIND_INT() 的宏，并将从小倒排索引（postings）中获取到的倒排列表存储到变量 ii_entry（在内部实际上是 ii_entry->postings_list）中。

而如果找不到以 token_id 为键的倒排列表，那么就先将变量 ii_entry的值设为 NULL（⓰）。

如果变量 ii_entry的值不为 NULL，也就是说小倒排索引中存在关联到该词元上的倒排列表（⓱），那么我们就先将指针 pl指向该倒排列表（⓲），**然后再将该倒排列表中词元的出现次**数增加 1（⓳）。在计算用于对检索结果进行排名的分数时，会用到词元的出现次数。**

反之，如果变量 ii_entry的值为 NULL，也就是说小倒排索引中不存在关联到该词元上的倒排列表，那么我们就先调用函数 create_new_inverted_index()，生成一个空的小倒排索引（⓴），

然后再调用 HASH_ADD_INT()，将该词元添加到新建的小倒排索引中（㉑）。接下来，通过调用函数 create_new_postings_list()，我们创建出了仅由 1 个文档构成的倒排列表 pl（㉒），随后又将该倒排列表添加到了刚刚生成的小倒排索引中（㉓）。

此时，指针 pl指向关联到词元上的倒排列表。

接着，我们又通过调用函数 utarray_push_back()，**将词元的出现位置添加到了倒排列表中存储着出现位置的数组的末尾**（㉔）。

最后，我们又将当前词元在所有文档中的出现次数之和增加 1。出现次数之和的数据存储在关联到词元的倒排列表中（㉕）。

函数 update_postings()

下面，让我们再来看一下在函数 add_document() 中被调用的函数 update_postings() 吧。

```
/**
 * 将内存上（小倒排索引中）的倒排列表与存储器上的倒排列表合并后存储到数据库中
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] p 含有倒排列表的倒排索引中的索引项
 */
void update_postings(const wiser_env *env, inverted_index_value *p)
{
  int old_postings_len;
  postings_list *old_postings;

  if (!fetch_postings(env, p->token_id, &old_postings,
                      &old_postings_len)) {//26
    buffer *buf;
    if (old_postings_len) {
      p->postings_list = merge_postings(old_postings, p->postings_list);
      p->docs_count += old_postings_len;//27
    }
    if ((buf = alloc_buffer())) {//28
```

```
      encode_postings(env, p->postings_list, p->docs_count, buf);//29
      db_update_postings(env, p->token_id, p->docs_count,
                         BUFFER_PTR(buf), BUFFER_SIZE(buf));//30
      free_buffer(buf);
    }
  } else {
    print_error("cannot fetch old postings list of token(%d) for update.",
                p->token_id);
  }
}
```

首先，我们通过调用函数 fetch_postings()，从存储器中取出了作为合并源的倒排列表（㉖）。

如果存储器中存在作为合并源的倒排列表，那么就调用函数 merge_postings()，将该倒排列表和要合并进来的倒排列表（p->postings_list）合并在一起（㉗）。有关函数 merge_postings() 的讲解先放到后面，我们继续往下看。

接下来，我们申请了一块临时的缓冲区（㉘），利用这块缓冲区和函数 encode_postings()，将内存上的倒排列表转换成了字节序列（㉙）。

最后，我们又通过调用函数 db_update_postings()，将转换后的字节序列存储到了存储器中（㉚）。

函数 merge_inverted_index()

下面我们再来看一下在函数 text_to_postings_lists() 中被调用的函数 merge_inverted_index()。函数 merge_inverted_index() 的作用是合并内存上的两个倒排索引。

```
/**
 * 合并两个倒排索引
 * @param[in] base 合并后其中的元素会增多的倒排索引（合并目标）
 * @param[in] to_be_added 合并后就被释放的倒排索引（合并源）
 *
 */
void merge_inverted_index(inverted_index_hash *base,
                          inverted_index_hash *to_be_added)//31
{
  inverted_index_value *p, *temp;

  HASH_ITER(hh, to_be_added, p, temp) {//32
    inverted_index_value *t;
    HASH_DEL(to_be_added, p);//33
    HASH_FIND_INT(base, &p->token_id, t);//34
    if (t) {//35
      t->postings_list = merge_postings(t->postings_list, p->postings_list);//36
      t->docs_count += p->docs_count;//37
      free(p);
    } else {
      HASH_ADD_INT(base, token_id, p);//38
    }
  }
}
```

该函数会先接收两个内存上的倒排索引作为参数（㉛），然后将合并源（传递给第 2 个参数的倒排索引）中的内容合并到目标（传递给第 1 个参数的倒排索引）中。合并完成后，该函数还会从内存上释放出传递给第 2 个参数的倒排索引所占用的空间。

具体的合并方法是，先将存储在作为合并源的倒排索引中的所有倒排列表逐一取出，存到临时变量里（㉜），然后再将刚刚取出的倒排列表从作为合并源的关联数组中删除（㉝）。

接下来，用刚取出的倒排列表所对应的词元，到合并目标中去查找与该词元对应的倒排列表（㉞）。**如果合并目标中存在相应的倒排列表（**㉟），就调用函数 merge_postings()，将合并源和合并目标中的元素所带有的倒排列表合并在一起（㊱）**，并将出现过该词元的文档数相加（㊲）——有关函数 merge_postings() 的实现我们稍后再讲解——反之，如果合并目标中没有

相应的倒排列表，就将获取的合并源中的倒排列表直接添加到作为合并目标的关联数组中（㊳）。另外，在这里，我们通过使用 uthash提供的宏 HASH_ITER()，实现了将合并源中的元素逐一取出的循环。

函数 merge_postings()

最后，我们再来详细地看一下在函数 merge_inverted_index() 和函数 update_postings() 中都调用了的函数 merge_postings() 吧。

```
/**
 * 获取将两个倒排列表合并后得到的倒排列表
 * @param[in] pa 要合并的倒排列表
 * @param[in] pb 要合并的倒排列表
 *
 * @return 合并后的倒排列表
 *
 * @attention 若base和to_be_added（参见函数merge_inverted_index）中的任意一个被破坏了，
 *            或者二者中含有相同的文档编号，则该函数的行为不可预知
 */
static postings_list * merge_postings(postings_list *pa, postings_list *pb)//39
{
  postings_list *ret = NULL, *p;//40
  /* 用pa和pb分别遍历base和to_be_added（参见函数merge_inverted_index）中的倒排列表中的元素， */
  /* 将二者连接成按文档编号升序排列的链表 */
  while (pa || pb) {//41
    postings_list *e;
    if (!pb || (pa && pa->document_id <= pb->document_id)) {//42
      e = pa;//43
      pa = pa->next;//44
    } else if (!pa || pa->document_id >= pb->document_id) {//45
      e = pb;//46
      pb = pb->next;//47
    } else {
      abort();
    }
    e->next = NULL;//48
    if (!ret) {
      ret = e;
    } else {
      p->next = e;
    }
    p = e;
  }
  return ret;
}
```

该函数会接收两个内存上的倒排列表作为参数（㊴），并返回将其合并后的倒排列表。我们使用变量 ret 来管理合并后的倒排列表（㊵）。

在分别使用变量 pa 和 pb 遍历两个倒排列表中的元素（倒排项）的过程中，我们要不断地从 pa 和 pb 所指向的两个元素中挑选出文档编号较小的一方，然后将其添加到合并后的倒排列表中。

如果 pa 所指向的文档编号小于 pb 所指向的文档编号（㊷），那么就将 pa 所指向的元素添加到合并后的倒排列表中（㊸），并让 pa 指向下一个元素（㊹）。

反之，如果 pb 所指向的文档编号小于 pa 所指向的文档编号（㊺），那么就将 pb 所指向的元素添加到合并后的倒排列表中（㊻），并让 pb 指向下一个元素（㊼）。

另外，在实际的合并过程中，在（㊸）和（㊻）两处，我们并没有将要添加的元素直接添加到倒排列表中，而是先将其保存到了变量 e 中。到了后面㊽的那一段代码，我们会将变量 e 添加到合并后的倒排列表中。如果 pa 和 pb 双方都指向了各自倒排列表中最后一个元素之后的位置，那么合并处理就此结束（㊶）。

至此为止，我们就梳理完了以函数 add_document() 为入口的构建倒排索引的处理流程。下面我们再来简单地回顾一下整个处理流程吧。

l 从文档中取出词元；

l 为每个词元创建倒排列表并将该倒排列表添加到小倒排索引中；

l 每当小倒排索引增长到一定大小，就将其与存储器上的倒排索引合并到一起。

开始检索

检索处理的大致流程

假设搜索引擎接收到了内容为“自制搜索引擎”的查询，那么接下来就会进行如下的检索处理；

① 将查询分割为词元（如果使用的是 bi-gram，那么就会分割出“自制”“制搜”“搜索”“索引”“引擎”5 个词元）。

② 将分割出的各个词元，按照出现过该词元的文档数量进行升序排列（升序排列的理由将在稍后阐明）。

③ 获取各个词元的倒排列表，并从中取出文档编号和该词元在文档中出现位置的列表。

④ 如果所有词元都出现在同一个文档中，并且这些词元的出现位置都是相邻的，那么就将该文档添加到检索结果中。

⑤ 计算已添加到检索结果中的各文档与查询的匹配度（在 wiser 中，我们使用 TF-IDF 值作为匹配度）。

⑥ 将检索结果按照匹配度的降序排列。

⑦ 从经过排序的检索结果中取出排在前面的若干个文档作为检索结果返回。

另外，在第②步中，之所以要将分割出的各个词元按照出现过该词元的文档数量进行升序排列，是因为这样做可以尽早缩小检索结果的范围。而尽早缩小检索结果的范围，就可以减少在步骤④中进行的比较处理的次数。

使用倒排索引进行检索

Search（）函数

```
/**
 * 进行全文检索
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] query 查询
 */
void search(wiser_env *env, const char *query)
{
  int query32_len;
  UTF32Char *query32;

  if (!utf8toutf32(query, strlen(query), &query32, &query32_len)) {
    search_results *results = NULL;

    if (query32_len < env->token_len) {
      print_error("too short query.");
    } else {
      query_token_hash *query_tokens = NULL;
      split_query_to_tokens(
        env, query32, query32_len, env->token_len, &query_tokens);
      search_docs(env, &results, query_tokens);
    }

    print_search_results(env, results);

    free(query32);
  }
}
```

首先，我们将查询字符串的编码由 UTF-8 转换成了 UTF-32。

随后判断了查询字符串的长度是否大于 N-gram中 N 的取值。如果长度大于 N，就调用函数 split_query_to_tokens()，将词元从查询字符串中提取出来。

接下来，以刚刚提取出来的词元作为参数，调用函数 search_docs()，开始进行检索处理。

最后的步骤中，调用了用于打印检索结果的函数 print_search_results()。至此，检索处理的流程就结束了。函数 print_search_results() 会先将检索结果从 results 中逐一取出，然后以检索结果中的文档编号为查询条件，从文档数据库中取出相应的文档标题，最后输出获取到的标题和检索的得分。

split_query_to_tokens() 函数

```
/**
 * 从查询字符串中提取出词元的信息
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] text 查询字符串
 * @param[in] text_len 查询字符串的长度
 * @param[in] n N-gram中N的取值
 * @param[in,out] query_tokens 按词元编号存储位置信息序列的关联数组
 *                             若传入的是指向NULL的指针，则新建一个关联数组
 * @retval 0 成功
 * @retval -1 失败
 */
int
split_query_to_tokens(wiser_env *env,
```

```
                        const UTF32Char *text,
                        const unsigned int text_len,
                        const int n, query_token_hash **query_tokens)
{
  return text_to_postings_lists(env,
                                0, /* 将document_id设为0 */
                                text, text_len, n,
                                (inverted_index_hash **)query_tokens);
}
```

从源代码可以看出，该函数实际上就是又去调用了在第 3 章讲解过的函数 text_to_postings_lists()。之所以这样做，是因为从字符串中生成倒排列表的处理过程，和从查询中提取出词元后再取出各词元位置信息的处理过程有很多共通之处。

另外，在检索时，我们将 0 传递给了函数 text_to_postings_lists() 的第 2 个参数，以表示不需要使用文档编号。也就是说，在函数 text_to_postings_lists() 的内部，是根据文档编号是否为 0 来判别当前是构建模式还是检索模式的。

检索处理流程的具体示例

首先，我们将查询字符串分割成了 3 个词元 —— “搜索 ”“索引”“引擎”。然后，分别扫描关联到这 3 个词元上的倒排列表。如果能够找到一个在所有倒排列表中都出现过的文档编号，那么就将它所指向的文档加入到候选检索结果中。

在这个过程中，我们要将第一个词元“搜索”称为“词元 A”。

具体的检索流程如下。

首先，从词元 A 的倒排列表的开头获取文档编号为 15 的元素；然后，在另外两个词元“索引”和“引擎”的倒排列表中，检查是否也存在文档编号为 15 的元素。

在词元“索引”的倒排列表中，第一个文档编号是 13，第二个文档编号是大于 15 的 17。此时就可以确定，文档编号小于 17 的文档一定不属于候选检索结果。之所以这样说，是因为倒排列表是按照文档编号的升序排列的。既然词元 A“搜索”未曾出现在编号小于 15 的文档中，词元“索引”也未曾出现在编号大于 15 且小于 17 的文档中，那么，在编号小于 17 的文档中就一定不可能同时出现“搜索”和“索引”。

由于编号小于 17 的文档不可能成为检索结果，所以会继续向后读取词元 A 的倒排列表，直到发现某个文档编号不小于 17 的元素为止。继续向后读，就读取到了文档编号为 18 的元素。

至此为止，我们做了如下几件事。

l **首先获取了词元 A的文档编号，然后检查了其他的词元是否也带有相同的文档编号；**

l **如果没有发现带有相同文档编号的词元，那么接下来就继续向后读取词元 A的倒排列表，直到遇到更大的文档编号为止。**

下一次循环时，词元 A 的文档编号是 18。对于除词元 A 以外的其他词元，只需要继续向后读取其各自的倒排列表，就可以知道是否包含着文档编号为 18 的元素了，或者说就可以知道哪个倒排列表中含有编号为 18 的文档了。由于在本例中所有的倒排列表中都含有编号为 18 的文档，所以该文档就成为了候选检索结果。

但编号为 18 的文档不是正式检索结果，而只是候选检索结果。这就需要诸位回忆一下在第 1 章中讲解过的有关短语检索的知识了。例如，假设我们将内容为“不可能”的查询发送给了搜索引擎，虽然有些文档也同时包含了“不可”和“可能”，但是这些文档却未必包含连在一起的“不可能”3 个字。**以“他不可一世的态度可能源于他童年时的经历”这句话为例**，虽然先后包含了“不可”和“可能”这两个词，但是却并没有包含“不可能”这个短语。因此要想查找像是“不可能”这样的短语，就必须确认“不可”和“可能”是不是相邻出现的。

也就是说，在倒排列表中，我们只需要关注 3 个含有文档编号 18 的方框中的出现位置，并检查“搜索”“索引”“引擎”这 3 个词元是不是顺序（相邻）出现的即可。换句话说，就是以“搜索”的出现位置为起始位置，确认“索引”的出现位置是否为起始位置 + 1，“引擎”的出现位置是否为起始位置 + 2。在本例中，“搜索”的出现位置是 30、“索引”是 31、“引擎”是 32，因此满足相邻出现的条件。记录在编号为 18 的文档中发现了短语后，就可以继续向后处理词元 A 的倒排列表了。

接下来要做的只不过是反复执行上述处理。

继续向后扫描，使词元 A 的文档编号依次变为 30 和 213。当文档编号为 213 时，由于在词元“索引”的倒排列表中，没有比 213 更大的文档编号了，同时也无法再继续向后读取下一个元素了，所以检索处理至此结束。

最后，对找到的检索结果进行排序（由于在本例中只有 1 条检索结果，所以也就没有排序的必要了）。

search_docs()函数

```
/**
 * 检索文档
 * @param[in] env 存储着应用程序运行环境的结构体
```

```
 * @param[in,out] results 检索结果
 * @param[in] tokens 从查询中提取出的词元信息
 */
void search_docs(wiser_env *env, search_results **results,
            query_token_hash *tokens)
{
  int n_tokens;
  doc_search_cursor *cursors;

  if (!tokens) { return; }

  /* 按照文档频率的升序对tokens排序 */
  HASH_SORT(tokens, query_token_value_docs_count_desc_sort);

  /* 初始化 */
  n_tokens = HASH_COUNT(tokens);
  if (n_tokens &&
      (cursors = (doc_search_cursor *)calloc(
                    sizeof(doc_search_cursor), n_tokens))) {
    int i;
    doc_search_cursor *cur;
    query_token_value *token;
    for (i = 0, token = tokens; token; i++, token = token->hh.next) {
      if (!token->token_id) {
        /* 当前的token在构建索引的过程中从未出现过 */
        goto exit;
      }
      if (fetch_postings(env, token->token_id,
                         &cursors[i].documents, NULL)) {
        print_error("decode postings error!: %d\n", token->token_id);
        goto exit;
      }
      if (!cursors[i].documents) {
        /* 虽然当前的token存在，但是由于更新或删除导致其倒排列表为空 */
        goto exit;
      }
      cursors[i].current = cursors[i].documents;
    }
    while (cursors[0].current) {
      int doc_id, next_doc_id = 0;
      /* 将拥有文档最少的词元称作A */
      doc_id = cursors[0].current->document_id;
      /* 对于除词元A以外的词元，不断获取其下一个document_id，直到当前的document_id不小于词
元A的document_id为止 */
      for (cur = cursors + 1, i = 1; i < n_tokens; cur++, i++) {
        while (cur->current && cur->current->document_id < doc_id) {
          cur->current = cur->current->next;
        }
        if (!cur->current) { goto exit; }
        /* 对于除词元A以外的词元，如果其document_id不等于词元A的document_id，*/
        /* 那么就将这个document_id设定为next_doc_id */
        if (cur->current->document_id != doc_id) {
          next_doc_id = cur->current->document_id;
          break;
        }
      }
      if (next_doc_id > 0) {
```

```
        /* 不断获取A的下一个document_id，直到其当前的document_id不小于next_doc_id为止
*/
        while (cursors[0].current
               && cursors[0].current->document_id < next_doc_id) {
          cursors[0].current = cursors[0].current->next;
        }
      } else {
        int phrase_count = -1;
        if (env->enable_phrase_search) {
          phrase_count = search_phrase(tokens, cursors);
        }
        if (phrase_count) {
          double score = calc_tf_idf(tokens, cursors, n_tokens,
                                     env->indexed_count);
          add_search_result(results, doc_id, score);
        }
        cursors[0].current = cursors[0].current->next;
      }
    }
exit:
    for (i = 0; i < n_tokens; i++) {
      if (cursors[i].documents) {
        free_token_positions_list(cursors[i].documents);
      }
    }
    free(cursors);
  }
  free_inverted_index(tokens);

  HASH_SORT(*results, search_results_score_desc_sort);
}
```

首先，要对从查询字符串中取出的词元（token）集合，按照各词元文档频率的升序进行排列。文档频率是指在作为检索对象的所有文档中，出现过某个词元的文档数量。

接下来，我们为每个词元都分配了一块内存空间，用于存储表示当前指向了哪个文档的状态（游标）。稍后遍历每个词元对应的倒排列表时，会用到这些游标。

在接下来的步骤中，我们从词元集合中将词元逐一取出。此时，如果某个词元还没有被分配编号，则说明无法从数据库中获取到该词元的编号，也就是说查询该词元得到的结果为空。一旦出现了这种情况，我们就跳转到 exit 标签以中断检索处理。而如果词元已经被分配了编号，那么接下来我们就通过调用函数 fetch_postings()，从索引（数据库）中获取该词元对应的倒排列表。此时，如果能够正确地获取到倒排列表，我们就设置该词元对应的游标，使其指向倒排列表中的第一个文档。

至此，我们通过直到为止的一系列处理，设定好了各个词元所对应的游标。接下来，和之前讲解具体示例时一样，我们将第一个词元称作词元 A。

在接下来的检索处理中，我们通过不断向后移动各个游标，来查找在所有的倒排列表中共同出现的文档编号。只要还没有扫描到词元 A 所对应的倒排列表的末尾，检索处理就不会结束。的确，我们应该一一检查所有词元的扫描位置，看看有没有扫描到各自倒排列表的末尾，但是在这里，只需要检查对词元 A 的扫描是否结束了即可。

在该循环中，我们首先通过词元 A 的游标获取到了一个文档编号。我们称这个文档编号为“文档编号 A”。

接下来，需要检查除词元 A 以外的所有词元的倒排列表，看看其中 是否也包含文档编号 A。检查时，如果游标指向的文档编号小于文档编号 A，那么就使游标指向下一个文档编号。

在扫描除词元 A 以外的词元的倒排列表的过程中，如果游标到达了该倒排列表的末尾，就要强制中断检索处理。

在的步骤中，如果当前游标所指向的文档编号不等于文档编号 A，那么说明在当前游标所指向的词元的倒排列表中不存在文档编号 A，同时也说明当前游标所指向的文档编号大于文档编号 A。因此，我们就将当前游标所指向的文档编号赋值给变量 next_doc_id。

如果 next_doc_id 大于 0，即文档编号 A 所指向的文档并不属于候选检索结果时，要将词元 A 的游标不断向后移动，直到该游标所指向的文档编号不小于 next_doc_id 为止。通过反复进行上述操作，最终就能达到所有的游标都指向同一个文档编号的状态。

当 next_doc_id 等于 0 时，说明文档编号 A 所指向的文档成为了候选检索结果，因此随后还要通过函数 search_phrase() 来确认作为短语的查询字符串是否存在。如果确实存在短语，那么就调用函数 calc_tf_idf()，计算出用于排序的得分，并调用函数 add_search_result()，将文档编号添加到检索结果的列表中。

接下来继续向后移动词元 A 的游标，并开始重复执行之前的处理过程。

检索一结束，exit 标签后面的语句就会被执行。在执行的过程中，既会回收分配给所有词元的、用来存储检索信息的存储空间，又会回收由查询字符串生成的词元集合的信息。在最后的步骤中，我们对检索结果按照得分的高低进行了降序排列。

search_phrase()函数

下面，我们再来看一下用于短语检索的函数 search_phrase()。不过，在深入阅读源代码之前，需要先稍微了解一下处理短语检索的策略。

假设文档中存在短语，此时构成短语的词元出现的位置如下所示。

| 11、12、13、14

| 86、87、88、89

从各出现位置中减去其在短语内的相对位置后，得到的结果如下所示。

当出现位置依次为 11、12、13、14 时

→ 11、11、11、11

※ 11 - 0 = 11，12 - 1 = 11，13 - 2 = 11，14 - 3 = 11

当出现位置分别为 86、87、88、89 时

→ 86、86、86、86

※ 86 - 0 = 86，87 - 1 = 86，88 - 2 = 86，89 - 3 = 86

由此可以看出，此时所有的出现位置都是相等的。通过这样的减法运算，就可以将检索短语的处理过程转化为“查找所有词元共同出现的位置”这一处理过程了。这与负责查找所有词元同时出现的文档编号的函数 search_docs() 所进行的处理非常相似，所以说，其实在函数 search_phrase() 中进行和函数 search_docs() 同样的处理就可以了。

```
/**
 * 进行短语检索
 * @param[in] query_tokens 从查询中提取出的词元信息
 * @param[in] doc_cursors 用于检索文档的游标的集合
 * @return 检索出的短语数
 */
static int search_phrase(const query_token_hash *query_tokens,
```

```
              doc_search_cursor *doc_cursors)
{
  int n_positions = 0;
  const query_token_value *qt;
  phrase_search_cursor *cursors;

  /* 获取查询中词元的总数 */
  for (qt = query_tokens; qt; qt = qt->hh.next) {
    n_positions += qt->positions_count;
  }

  if ((cursors = (phrase_search_cursor *)malloc(sizeof(
                   phrase_search_cursor) * n_positions))) {
    int i, phrase_count = 0;
    phrase_search_cursor *cur;
    /* 初始化游标 */
    for (i = 0, cur = cursors, qt = query_tokens; qt;
         i++, qt = qt->hh.next) {
      int *pos = NULL;
      while ((pos = (int *)utarray_next(qt->postings_list->positions,
                                        pos))) {
        cur->base = *pos;
        cur->positions = doc_cursors[i].current->positions;
        cur->current = (int *)utarray_front(cur->positions);
        cur++;
      }
    }
    /* 检索短语 */
    while (cursors[0].current) {
      int rel_position, next_rel_position;
      rel_position = next_rel_position = *cursors[0].current -
                                         cursors[0].base;
      /* 对于除词元A以外的词元，不断地向后读取其出现位置，直到其偏移量不小于词元A的偏移量为止
*/
      for (cur = cursors + 1, i = 1; i < n_positions; cur++, i++) {
        for (; cur->current
             && (*cur->current - cur->base) < rel_position;
             cur->current = (int *)utarray_next(cur->positions, cur->current))
{}
        if (!cur->current) { goto exit; }

        /* 对于除词元A以外的词元，若其偏移量不等于A的偏移量，就退出循环 */
        if ((*cur->current - cur->base) != rel_position) {
          next_rel_position = *cur->current - cur->base;
          break;
        }
      }
      if (next_rel_position > rel_position) {
        /* 不断向后读取，直到词元A的偏移量不小于next_rel_position为止 */
        while (cursors[0].current &&
               (*cursors[0].current - cursors[0].base) < next_rel_position) {
          cursors[0].current = (int *)utarray_next(
                                 cursors[0].positions, cursors[0].current);
        }
      } else {
        /* 找到了短语 */
        phrase_count++;
        cursors->current = (int *)utarray_next(
```

```
                                cursors->positions, cursors->current);
        }
      }
exit:
      free(cursors);
      return phrase_count;
    }
    return 0;
}
```

首先，我们统计出了查询中的词元总数，并为查询中的每个词元都分配了一个结构体，用来表示用于短语检索的游标。在初始化这些游标的过程中，会将词元在查询中的出现位置存储到 cur->base 中，将对词元在文档中出现位置的引用存储到 cur->current 中。

我们沿用在讲解函数 search_docs() 时用到的术语，依然称第一个词元为词元 A。不同的是，在该函数中查找的对象是出现位置的列表，而不是文档编号的列表。因此在该函数中使用游标管理的是“在关联着词元和文档编号的出现位置列表中，当前指向的是哪个位置”。第 i个词元的游标存储在 cursors[i].current 中。

正如前文所述，对于所有的词元，都要从头检查其在文档内的出现位置。与通过函数 search_docs() 查找带有相同文档编号的词元类似，在该函数中要查找的是，从各词元的出现位置中减去其在查询中的出现位置后得到的偏移量完全相同的词元。

具体来说，就是用词元 A 的出现位置减去其在查询中的出现位置，得到词元 A 的偏移量，然后，对除词元 A 以外的所有词元逐一进行相同的减法运算。每完成一次减法运算，都要检查所得到的偏移量是否和词元 A 的偏移量相等。如果不相等，则说明短语不存在，因此要重新对变量 next_rel_position赋值，以便跳过没有必要处理的词元的出现位置。

当找不到短语时，由于小于 next_rel_position的偏移量都已经搜索过了，所以我们要向后移动词元 A 的游标，直到其偏移量超过 next_rel_position为止。

如果找到了短语，那么就将用于存储短语出现次数的变量 phrase_count 的值加 1，并使词元 A 的游标指向其下一个出现位置。

由于此时已经知道了短语是否存在，所以也可以就此结束处理，但是考虑到短语的出现次数还会用在后面的评分处理等环节，因此我们还是决定让处理继续进行下去。

通过反复进行上述循环，即可找出全部的短语。最后，在返回了找到的短语个数后，就可以结束处理了。

至此，我们就梳理完了使用倒排索引进行全文搜索处理的流程。如果感到难以理解，不妨先将下面的大致流程记在心中，然后再试着梳理源代码。也可以边梳理边在纸上将处理流程和变量的值写出来。

① 将查询分割成词元。

② 获取每个词元的倒排列表。

③ 从多个倒排列表中查找匹配查询的文档（即在带有相同文档编号的倒排项中，判断位置信息是否是相邻的），并将找到的文档添加到作为检索结果的文档集合中。

④ 计算作为检索结果的文档的得分，并基于该得分对结果排序。

压缩倒排索引

压缩的基础知识

在使用倒排索引进行检索的过程中，总检索时间中的大部分时间往往花费在了从二级存储读取倒排索引上。于是，就经常可以看到在存储倒排索引前，对其进行压缩以减少从二级存储读取的时间，进而使检索处理得以高速运转的对策。

即：**从二级存储中读取（部分）经过压缩的倒排索引的时间 + 还原倒排索引的时间 < 从二级存储中读取（部分）尚未经过压缩的倒排索引的时间；**

倒排索引的压缩方法

我们可以通过使用更少的信息量表示单词的集合来实现词典的压缩。例如，对于按照词典顺序排列的单词列表而言，通过避免重复存储相同的前缀，就可以减少存储词典时所需的必要存储空间。但是，在大多数情况下，由于词典的大小远远小于倒排文件的大小，所以一般认为压缩词典对于加快检索处理的速度并没有太大的贡献。

而倒排文件的压缩，可以通过使用更少的信息量表示其构成要素来实现。构成要素就是指文档编号、单词在文档内的出现次数（TF，TermFrequency，词频）以及由单词在文档内的偏移量构成的整数数组。

倒排文件的压缩方法

在一般的程序中，大多数情况下都会为整数分配 4 或 8 个字节等定长的编码，但是在处理倒排文件时，由于经常要处理大量数值较小的整数，所以为了使用更少的信息量来表示整数，通常都会采用长度可变而非固定的编码方式。另外，由于倒排文件中的整数序列通常都是按照文档编号或文档内偏移量的升序排列的，所以对于这样的整数序列，一旦计算出了前后两个整数的差值，就可以使用更小的数值（更少的信息量）来表示其中的整数了。也就是说正如预期的那样，使用可变长度的编码确实可以带来大幅度的压缩。

unary 编码

unary编码是一种简单直观的编码方法，对于整数 x （x ≥ 0）来说， 只需要用 x 个“1”和 1 个“0”即可表示这个整数。以十进制数的 10 为例，它的 unary编码就是“11111111110”。而当 x = 0 时，其 unary编码为 0。

gamma 编码

在 gamma 编码中，我们首先要将整数 x （x > 0）分解为 的形式，然后用 unary编码表示 e + 1，用比特宽度为 e 的二进制编码表示d 。

所谓二进制编码，就是一种用二进制数字 0 和 1 表示数据的方式。这种方式也是计算机内部标准的数据存储方式。——译者注

还是以整数 10 为例，由于 e = log210 ≈ 3，所以把 d = 10 - 23 = 2 的二进制编码“010”接在 e + 1 = 3 + 1 = 4 的 unary编码“11110”之后，就可以得到 10 的 gamma 编码，即“11110010”。

variable-byte 编码（byte-aligned编码）

variable-byte 编码是一种用由多个字节构成的序列表示整数的编码方式。在对整数编码时，各个字节的构成如下所示。

- **用最右侧的 7 个比特表示数值；**

- **用最左侧的 1 个比特表示是否需要用下一个字节来继续表示该整数的剩余部分（需要时将该比特设为 1）；**

也就是说，这种编码方式可以根据字节数的多少，分别表示如下的整数范围。

- **1 字节：0~27- 1（用 7 个比特来存储数据）**

- **2 字节：0~214- 1（用 14 个比特来存储数据）**

l 3 字节：0~221- 1（用 21 个比特来存储数据）

例如，10 的 variable-byte 编码是“00001010”。

而 1030 的 variable-byte 编码是“10000100:00000110”（为了便于查看，我们在两个字节间插入了一个“:”）。

与以字节为单位进行压缩的 variable-byte 编码相比，由于 delta 编码和 gamma 编码都是在比特级别上进行编码的编码方式，所以可以达到更好的压缩率。但是，在解码时，由于必须进行位操作，所以后者的处理速度可能不如使用 variable-byte 编码时快。因此，为了加快检索处理的速度，也有很多项目会在压缩倒排文件时采用 variable-byte 编码。

由于以上这些编码方式本质上都是通过相同的策略对整数进行编码的，所以统称为**无参数的编码方式（Parameterless Codes)。**

Golomb编码

使用 Golomb 编码对整数 x （x ≥ 0）进行编码时，要使用参数 m （≥ 1）将 x 分解为如下形式。

x = m × q + r;

其中，q 为 x 除以 m 的商，r 为余数。接下来需要对 q 进行 unary编码，对 r 进行如下所示的编码，最后再将这两个编码拼接在一起即可得到 Golomb 编码。在这个过程中，令 b = log2m（向上取整），t = 2b - m。

当0 ≤ r < t 时，用比特宽度为 b - 1 的二进制编码表示 r;

当 t ≤ r < m时，用比特宽度为 b 的二进制编码表示 r + t;

下面以 9 为例，当 m = 5 时，由于 9 = 5×1 + 4（q = 1、r = 4、m = 5)，所以可以得出：

b = log2m = log2 5 ≈ 3;

t = 2b- m = 23- 5 = 3 (< r);

因此，我们要对q（1）进行 unary编码，对 r（4） + t（3）进行比特宽度为 b（3）的二进制编码，最后将二者拼接起来后就得到了 9 的 Golomb 编码“10:111”。

另外，当 m = 1 时，Golomb 编码等同于 unary编码，当 m = 2k(k =1, 2, 3...) 时，Golomb 编码等同于一种叫作 rice 的编码。也就是说，unary编码和 rice 编码都是 Golomb 编码的特殊形式。

由于我们在 wiser 中实现的是 Golomb 编码，所以在后面的章节还会再进一步讲解它。

实现wiser 中的压缩功能

压缩功能源代码的概要

我们会将压缩称为编码，将解压缩称为解码；

在 wiser 中，我们通过 Golomb 编码实现了编码和解码，并通过在函数 update_postings() 中调用的函数 encode_postings() 和 decode_postings() 实现了倒排列表的编码和解码。

encode_postings() 和 decode_postings()

```
/**
 * 对倒排列表进行还原或解码
 * @param[in] env 存储着应用程序运行环境的结构体
```

```
 * @param[in] postings_e 待还原或解码前的倒排列表
 * @param[in] postings_e_size 待还原或解码前的倒排列表中的元素数
 * @param[out] postings 还原或解码后的倒排列表
 * @param[out] postings_len 还原或解码后的倒排列表中的元素数
 * @retval 0 成功
 */
static int
decode_postings(const wiser_env *env,
                const char *postings_e, int postings_e_size,
                postings_list **postings, int *postings_len)
{
  switch (env->compress) {
  case compress_none:
    return decode_postings_none(postings_e, postings_e_size,
                                postings, postings_len);
  case compress_golomb:
    return decode_postings_golomb(postings_e, postings_e_size,
                                  postings, postings_len);
  default:
    abort();
  }
}

/**
 * 对倒排列表进行转换或编码
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] postings 待转换或编码前的倒排列表
 * @param[in] postings_len 待转换或编码前的倒排列表中的元素数
 * @param[out] postings_e 转换或编码后的倒排列表
 * @retval 0 成功
 */
static int encode_postings(const wiser_env *env,
                const postings_list *postings, const int postings_len,
                buffer *postings_e)
{
  switch (env->compress) {
  case compress_none:
    return encode_postings_none(postings, postings_len, postings_e);
  case compress_golomb:
    return encode_postings_golomb(db_get_document_count(env),
                                  postings, postings_len, postings_e);
  default:
    abort();
  }
}
```

这两个函数会根据是否要进行编码的标志（env->compress）去调用相应的编码、解码函数。无需对倒排列表进行编码时，调用的是下面这两个函数。

l encode_postings_none()

l decode_postings_none()

需要通过 Golomb 编码进行编码、解码时，调用的是下面这两个函数。

l encode_postings_golomb()

l decode_postings_golomb()

无需进行压缩时的操作

函数 encode_postings_none() 的作用是将倒排列表转换成字节序列。也就是说，该函数会先从倒排列表的各元素中取出文档编号、位置信息的数量以及位置信息的数组，然后再将这些数据以二进制的形式写入缓冲区。

```
/**
 * 将倒排列表转换成字节序列
 * @param[in] postings 倒排列表
 * @param[in] postings_len 倒排列表中的元素数
 * @param[out] postings_e 转换后的倒排列表
 * @retval 0 成功
 */
static int
encode_postings_none(const postings_list *postings,
                     const int postings_len,
                     buffer *postings_e)
{
  const postings_list *p;
  LL_FOREACH(postings, p) {//1
    int *pos = NULL;
    append_buffer(postings_e, (void *)&p->document_id, sizeof(int));//2
    append_buffer(postings_e, (void *)&p->positions_count, sizeof(int));//3
    while ((pos = (int *)utarray_next(p->positions, pos))) {//4
      append_buffer(postings_e, (void *)pos, sizeof(int));//5
    }
  }
  return 0;
}
```

首先，我们通过调用 utarray的宏 LL_FOREACH()，从倒排列表中逐一取出各个文档编号的出现位置信息（❶）。

然后，将文档编号和存放出现位置信息的数组的大小（出现位置的数量）分别添加到缓冲区中（❷、❸），接着将各个出现位置（❹）也添加到该缓冲区中（❺）。

而在函数 decode_postings_none() 中进行的是函数 encode_postings_none() 的逆操作。即依次取出文档编号、位置信息的数量以及各个位置信息。

Golomb 编码的要点

无需对倒排列表进行编码时，我们只需将被直接转换成了字节序列的倒排列表写入数据库即可。而需要对倒排列表进行编码时，就要先对以下 3 种取自倒排列表的信息进行编码，然后再将转换成字节序列后的数据写入数据库。

l 已存储的文档编号；

l 位置信息的数量；

l 位置信息的数组。

假设有 [13，22，23，40] 这样一个文档编号的序列，我们试着考虑一下应该如何通过 Golomb 编码对其进行编码。首先要计算出这个整数序列中的所有后项与前项的差值。而对于第一个数字，则要计算它和 0 的差值。于是就得到了 [13，9，1，17] 这样一个整数序列。接下来，将所有的数字都减去 1。**因为这样可以利用上文档编号序列中没有重复的编号，并且所有的前后项之差都是大于 1 的数值这个特**

点，使整数序列可以用更小的数值表示。于是，我们又得到了 [12，8，0，16] 这样一个整数序列。接下来，就开始用 Golomb 编码对这个整数序列进行编码。

正如前文所述，进行 Golomb 编码时要使用参数m。我们都知道，对于参数m，**将它的值设为待编码的整数序列的平均值可以得到较高的压缩率。**因此，在本例中我们将 m 的值设为该整数序列的平均值9。

如下所示，Golomb 编码使用了参数 m 以及 q 和 r 两个整数来表示一个整数 n 。整数b和t用于对整数r进行编码，其取值由m决定（b = log2m 向上取整，t = 2 b - m ）。当 m = 9 时，b = 4、t = 7。

n = q ×m + r。

用 Golomb编码对整数序列进行编码

下面就让我们开始用 Golomb 编码对整数序列 [12，8，0，16] 进行编码吧。

l 对第1个整数进行编码

首先，从整数序列中取出第一个整数12。然后用12除以m并舍去小数部分，得出了q的值为 1。 接下来开始编码，首先我们用 unary编码来表示q的值。unary编码是一种通过将1个0附加到n个连续的1之后来表示数值n的编码方式。也就是说，用 unary编码表示1的话，得到的比特序列是10。接下来，计算12除以m的余数，又得到了r的值为3。

表示r时要从以下2种模式中选择一种。

\1. 当**r小于t时 →用比特宽度为 b - 1 的二进制比特序列表示 r 的值；**

\2. 当**r不小于t时 →用比特宽度为 b 的二进制比特序列表示 r + t 的值；**

由于此时 r < t （3 < 7），所以要用 3 位二进制数表示 3，再用得到的比特序列“011”来表示 r 的值。至此为止，我们得到的比特序列为“10011”。

l 对第2个整数进行编码；

下面，从整数序列取出第二个整数8。此时q = 0，r = 8。用 unary编码表示q，得到的比特序列为“0”。由于r ≥ t（8 > 7），所以要用4个比特的比特序列“1111”来表示r + t。至此为止，我们得到的比特序列为“10011011:11”。为了便于阅读，我们在每 8 个比特之后都插入了一个“:”，作为字节间的边界（余同)。

l 对第3个整数进行编码；

接下来我们取出了第3个整数0，并按照之前的方法，求出了q和r的值。此时q = 0，r = 0。用 unary编码表示q，得到的比特序列为 0。由于 r < t （0 < 7），所以要用由 3个比特的比特序列“000”来表示 r 的值。至此为止，我们得到的比特序列为“10011011:110000”。

l 对第4个整数进行编码。

最后，我们取出了最后一个整数16。此时，q = 1，r = 7。用 unary编码表示q，得到的序列为 10。由于 r ≥ t （7 = 7），所以要用由4个比特表示的比特序列“1110”来表示r + t。最终我们得到的比特序列为“10011011:11000010:1110”。

至此，对该整数序列的Golomb编码就结束了。可以看出，最终只需要2.5个字节即可表示经过排序的整数序列 [13，22，23，40]。 在存储这 2.5个字节的比特序列时，为了以 1 个字节（= 8 个比特）为最小单位，我们在最后填充了4个值为 0 的比特，填充后的比特序列为“10011011:11000010:11100000”。

将比特序列解码成原先的整数序列

要想解码，必须事先知道经过编码的整数个数以及参数m的取值。在这里，我们已知比特序列中存储了4个整数，并且参数m的值是9。

l 对第1个整数进行解码；

首先，要从比特序列的起始位置开始查找值为 0 的比特。第2个比特的值就是0。如果第n个比特的值为0，那么就将从查找的起始位置到第n - 1个比特之间的数值作为q用 unary编码进行解码。

然后从刚找到的值为 0 的比特开始，再向后读取 b - 1 个（3 个）比特，得到比特序列“011”。将用二进制表示的 011 转换为十进制的整数后，得到的结果是 3。由于求得的整数 3 小于 t （7），所以就直接将求得的数值作为r的值。

因此，r = 3。既然已经知道了3个参数 m 、q 和 r 的取值，那么就可以算出 1×9 + 3 = 12。由此就正确地解出了第一个整数 12。至此，我们就得到了 [12] 这样一个整数序列。

l 对第2个整数进行解码；

接下来继续寻找值为 0 的比特。正好第一个比特就是 0。因此 q = 0。

然后继续向后读取 b - 1 个（3个）比特，得到了比特序列“111”。二进制的“111”即十进制的7。由于求得的数值7不小于t（7），所以还要再读取1个比特。

即，最终读取到的比特序列是“1111”。二进制的“1111”即十进制的 15。

接下来只需再减去 t 即可求得 r 的值，即 r = 15 - t = 15 - 7 = 8。根据参数m、q 和 r，即可解出第二个整数，即 0×9 + 8 = 8。至此，我们得到的数列是 [12，8]。

l 对第3、4个整数进行解码；

由以上步骤易得：解出 0 和 16 两个整数。

l 加上前后项差值后再将各个整数加 1；

由于这个整数序列中的每个整数（除第一个整数以外）其实都是与前 1 个整数的差值，所以我们还要将这个差值加到每个整数上。然后还要再对所有整数都加上 1。至此我们就得到了原先的整数数列 [13，22，23，40]。

Golomb 编码的实现

函数 encode_postings_golomb()

```
/**
 * 对倒排列表进行Golomb编码
 * @param[in] documents_count 文档总数
 * @param[in] postings 待编码的倒排列表
 * @param[in] postings_len 待编码的倒排列表中的元素数
 * @param[in] postings_e 编码后的倒排列表
 * @retval 0 成功
 */
static int
encode_postings_golomb(int documents_count,
                       const postings_list *postings, const int postings_len,
                       buffer *postings_e)
{
  const postings_list *p;

  append_buffer(postings_e, &postings_len, sizeof(int));//6
  if (postings && postings_len) {
    int m, b, t;
    m = documents_count / postings_len;//7
    append_buffer(postings_e, &m, sizeof(int));//8
    calc_golomb_params(m, &b, &t);//9
    {
      int pre_document_id = 0;
```

```
      LL_FOREACH(postings, p) {//10
        int gap = p->document_id - pre_document_id - 1;//11
        golomb_encoding(m, b, t, gap, postings_e);//12
        pre_document_id = p->document_id;
      }
    }
    append_buffer(postings_e, NULL, 0);//13
  }
  LL_FOREACH(postings, p) {//14
    append_buffer(postings_e, &p->positions_count, sizeof(int));
    if (p->positions && p->positions_count) {
      const int *pp;
      int mp, bp, tp, pre_position = -1;

      pp = (const int *)utarray_back(p->positions);//15
      mp = (*pp + 1) / p->positions_count;//16
      calc_golomb_params(mp, &bp, &tp);//17
      append_buffer(postings_e, &mp, sizeof(int));
      pp = NULL;
      while ((pp = (const int *)utarray_next(p->positions, pp))) {
        int gap = *pp - pre_position - 1;
        golomb_encoding(mp, bp, tp, gap, postings_e);
        pre_position = *pp;
      }
      append_buffer(postings_e, NULL, 0);
    }
  }
  return 0;
}
```

在Golomb编码中，由于要使用整数序列中前后项的差值，所以我们要对文档编号和出现位置分别进行编码。

首先，将倒排列表中包含的文档数存储起来（❻）。接下来，计算出用于对文档编号的差值序列进行 Golomb 编码的参数 m 的取值（❼）。在这里，我们使用了文档编号差值的平均值作为参数 m 的取值。文档编号差值的平均值可以使用文档总数除以倒排列表中包含的文档数计算得出。由于在解码时还要用到参数 m，所以在这里我们还要先将参数 m 存储起来（❽）。

接下来，计算出由参数 m 唯一决定的参数 b 和 t 的取值（b = log2m 向上取整，t = 2b - m）（❾）。

然后逐一取出倒排列表中的文档编号（❿）。每取出一个文档编号，就计算其与刚刚存储的文档编号的差值（⓫）。随后用函数golomb_encoding()对计算结果进行编码（⓬），最后以比特为单位将编码结果添加到变量postings_e中。

接下来，再次通过函数append_buffer()，将以比特为单位的信息统一为以字节为最小单位的信息（⓭）。

在标有⓮的循环中，又对由词元先后出现位置的差值构成的整数序列进行了编码。**虽然基本的流程与对文档编号进行编码的流程大致相同，但是由于词元在每个文档中的出现次数都不相同，所以要对每个文档单独计算 Golomb 编码中的参数 m。** 在 ⓯ 的步骤 中，通过调用能返回数组中最后一个元素的函数 utarray_back()，获取了词元在文档中最后一次出现的位置。

接下来，用词元在文档中最后一次出现的位置除以词元在文档中的出现次数，计算出了对出现位置进行编码时需要用到的参数 m（变量mp）（⓰）。在⓱的步骤后，我们又使用了与之前从❾到⓭对文档编号进行编码时相同的流程，对出现位置数组中的整数进行了编码。

函数 golomb_encoding()

```
/**
 * 用Golomb编码对1个数值进行编码
 * @param[in] m Golomb编码中的参数m
 * @param[in] b Golomb编码中的参数b。ceil(log2(m))
 * @param[in] t pow2(b) - m
 * @param[in] n 待编码的数值
 * @param[in] buf 编码后的数据
 */
static inline void golomb_encoding(int m, int b, int t, int n, buffer *buf)
{
  int i;
  /* encode (n / m) with unary code */
  for (i = n / m; i; i--) { append_buffer_bit(buf, 1); }//18
  append_buffer_bit(buf, 0);//19
  /* encode (n % m) */
  if (m > 1) {//20
    int r = n % m;
    if (r < t) {
      for (i = 1 << (b - 2); i; i >>= 1) {
        append_buffer_bit(buf, r & i);
      }
    } else {
      r += t;
      for (i = 1 << (b - 1); i; i >>= 1) {
        append_buffer_bit(buf, r & i);
      }
    }
  }
}
```

首先，通过unary编码对n / m的结果进行编码（⓲），然后通过函数 append_buffer_bit() 将编码后的比特序列写入到缓冲区中（⓳）。 接下来，从⓴的步骤开始，对由 n %m 计算出的数值 r 进行了编码，并将结果也写入到了缓冲区中。

函数 decode_postings_golomb()

```
/**
 * 对经过Golomb编码的倒排列表进行解码
 * @param[in] postings_e 经过Golomb编码的倒排列表
 * @param[in] postings_e_size 经过Golomb编码的倒排列表中的元素数
 * @param[out] postings 解码后的倒排列表
 * @param[out] postings_len 解码后的倒排列表中的元素数
 * @retval 0 成功
 */
static int decode_postings_golomb(const char *postings_e, int postings_e_size,
                          postings_list **postings, int *postings_len)
{
  const char *pend;
  unsigned char bit;

  pend = postings_e + postings_e_size;
  bit = 0x80;//21
  *postings = NULL;
  *postings_len = 0;
```

```
  {
    int i, docs_count;
    postings_list *pl;
    {
      int m, b, t, pre_document_id = 0;

      docs_count = *((int *)postings_e);//22
      postings_e += sizeof(int);
      m = *((int *)postings_e);//23
      postings_e += sizeof(int);
      calc_golomb_params(m, &b, &t);
      for (i = 0; i < docs_count; i++) {//24
        int gap = golomb_decoding(m, b, t, &postings_e, pend, &bit);//25
        if ((pl = malloc(sizeof(postings_list)))) {
          pl->document_id = pre_document_id + gap + 1;//26
          utarray_new(pl->positions, &ut_int_icd);
          LL_APPEND(*postings, pl);
          (*postings_len)++;
          pre_document_id = pl->document_id;
        } else {
          print_error("memory allocation failed.");
        }
      }
    }
    if (bit != 0x80) { postings_e++; bit = 0x80; }//27
    for (i = 0, pl = *postings; i < docs_count; i++, pl = pl->next) {
      int j, mp, bp, tp, position = -1;

      pl->positions_count = *((int *)postings_e);
      postings_e += sizeof(int);
      mp = *((int *)postings_e);
      postings_e += sizeof(int);
      calc_golomb_params(mp, &bp, &tp);
      for (j = 0; j < pl->positions_count; j++) {
        int gap = golomb_decoding(mp, bp, tp, &postings_e, pend, &bit);
        position += gap + 1;
        utarray_push_back(pl->positions, &position);
      }
      if (bit != 0x80) { postings_e++; bit = 0x80; }
    }
  }
  return 0;
}
```

首先，我们要对文档编号的整数序列进行解码。为此，需要先初始化表示当前正指向二进制序列中哪个比特的变量 bit（㉑）。0x80 表示当前正指向第0个比特。

有关该变量的细节，将在稍后讲解函数 golomb_decoding() 时再一同讲解。接下来，在㉒和㉓的步骤中，分别读取了文档数（变量docs_count）和用Golomb 编码进行压缩时所要用到的参数m。

然后对于每个文档（㉔），通过调用函数 golomb_decoding()，对作为文档编号差值的整数（㉕）进行了解码。随后，又根据该差值，还原了原始的文档编号（㉖）。

最后，在对出现位置进行解码时（㉗之后），采用了与之前（从㉑到㉖）对文档编号进行解码时同样的处理过程。

函数golomb_decoding()

```
/**
 * 用Golomb编码对1个数值进行解码
 * @param[in] m Golomb编码中的参数m
 * @param[in] b Golomb编码中的参数b。ceil(log2(m))
 * @param[in] t pow2(b) - m
 * @param[in,out] buf 待解码的数据
 * @param[in] buf_end 待解码数据的结尾
 * @param[in,out] bit 待解码数据的起始比特
 * @return 解码后的数值
 */
static inline int golomb_decoding(int m, int b, int t,
                const char **buf, const char *buf_end, unsigned char *bit)
{
  int n = 0;

  /* decode (n / m) with unary code */
  while (read_bit(buf, buf_end, bit) == 1) {//28
    n += m;//29
  }
  /* decode (n % m) */
  if (m > 1) {//30
    int i, r = 0;
    for (i = 0; i < b - 1; i++) {//31
      int z = read_bit(buf, buf_end, bit);//32
      if (z == -1) { print_error("invalid golomb code"); break; }
      r = (r << 1) | z;//33
    }
    if (r >= t) {//34
      int z = read_bit(buf, buf_end, bit);//35
      if (z == -1) {
        print_error("invalid golomb code");
      } else {
        r = (r << 1) | z;//36
        r -= t;//37
      }
    }
    n += r;//38
  }
  return n;
}
```

正如前文所述，变量 bit 的作用是以二进制数的形式表示要从变量 buf的哪个位置开始读取。例如，当 bit 的值为 0x80 时，由于将其转换成二进制后左数第 1 个位置上的比特是 1，所以就 表示要从 buf的第 1 个比特开始读取。以此类推，当 bit 的值为 0x40 时，由于将其转换成二进制后左数第 2 个位置上的比特是 1，所以表示要从 buf的第 2 个比特开始读取。之所以要用 二进制数表示变量 bit，是因为只需要对 bit 和 buf进行逻辑与运算，就可以从 buf与 bit 中值为 1 的比特相对应的位置上读取出 1 个比特值。

下面列出了 bit 的各种取值分别能够读取 buf中哪个位置上的比特值。

bit：0x80 二进制表示：10000000 指向的比特：buf 中的第1 个比特；

bit：0x40 二进制表示：01000000 指向的比特：buf 中的第2 个比特；

bit：0x20 二进制表示：00100000 指向的比特：buf 中的第3 个比特；

bit：0x10 二进制表示：00010000 指向的比特：buf 中的第4 个比特；

bit：0x08 二进制表示：00001000 指向的比特：buf 中的第5 个比特；

bit：0x04 二进制表示：00000100 指向的比特：buf 中的第6 个比特；

bit：0x02 二进制表示：00000010 指向的比特：buf 中的第7 个比特；

bit：0x01 二进制表示：00000001 指向的比特：buf 中的第8 个比特；

首先，我们通过unary编码对二进制序列进行了解码（㉘、㉙），二进制序列指的是在函数golomb_encoding()中进行⓲和⓳两处编码后得到的序列。正如前文所述，函数 read_bit()（㉘）会利用 buf和 bit，从 buf中读取出 1 个比特。根据 Golomb 编码的规则，只要当前 bit 所指向的 buf中的比特值为1，就要累加到n，最后我们就通过这种方法，将“解码后的数 值 ×m ”的结果赋值给了变量 n（㉙）。

㉚之后的内容，是对二进制序列进行的解码处理，这里的二进制序列指的是在函数golomb_encoding()中进行⓴之后的编码所形成的序列。接下来在从㉛到㉝的步骤中，我们将二进制序列 中前 b - 1 个比特的数据解码后赋值给了变量r 。具体做法是反复执行 b - 1 次㉛以后的操作：先从二进制序列中读取 1 个比特（㉜），然后将r左移1个比特，用刚刚读取出的 1 个比特值作为左移后空出的最低比特上的值（㉝）。

当 r 不小于 t 时（㉞），我们还要继续向后读取 1 个比特（㉟），然后只要再重复执行一遍㉝的操作（㊱），并减去 t 的值（㊲），即可解出 r 的值。而当 r 小于 t 时，由于此时 r 中存 放的就是解码后的值，所以在累加到变量 n 之前就不需要再进行什么处理了。

最后只需要再将 n 和 r 加到一起，即可解出原来的数值了（㊳）。

根据以上的这些，我们已经可以实现一个功能完备的搜索引擎了。

wiser的优化及参数的调整

wiser 是一个简单的搜索引擎，希望可以理解搜索引擎的核心部分。因 此，要想使之成为一个实用的搜索引擎，还需要大量的优化工作。

提高检索处理的效率

优化检索处理

在 wiser 现有的检索处理过程中，我们采用了与构建倒排索引时同样的分割方法，即每次向后错开 1 个字符，将查询分割成了 bi-gram的词元序列，并检查了分割出来的词元在文档中是 否是按顺序排列的。在这个过程中有些地方是可以优化的。

下面我们就举例说明。假设我们用 bi-gram的词元为文档“自制搜索引擎”构建出了如下的倒排索引。倒排索引中所有的文档编号均为：

- 自制:0;0
- 制搜:0;1
- 搜索:0;2
- 索引:0;3
- 引擎:0;4

当通过查询字符串“自制搜索引擎”检索上述倒排索引时，只要将其分割成“自制”“搜索”“引擎”3 个词元，并观察到在文档中后一个词元的出现位置（偏移量）总是与前一个词元的出现位置 相距 2 个字符，就可以断定短语“自制搜索引擎”出现在了文档中。也就是说，对于由每次向后错开 1 个字符而形成的 bi-gram的词元构建的倒排索引而言，只需要将查询分割为若干个无 重复部分的词元序列即可。

通过这样的分割，就可以减少需要参与检索的词元的数量。这也就意味着这样的分割有助于减少关联到词元上的倒排列表的获取次数，从而降低对多个倒排列表中的出现位置信息进行 相邻判定时的比较处理的次数。在获取倒排列表时通常都会伴随着大量的 I/O 操作，而进行比较处理时又通常会消耗大量的 CPU 资源，因此只要减少了词元的数量，就能大幅度地提升 检索处理的效率。

将查询分割为无重复部分的词元序列

下面，我们来看一下如何将查询分割为无重复部分的词元序列。在这里，我们需要对函数 text_to_postings_lists() 进行改造，该函数会被从字符串中提取词元的函数 split_query_to_tokens() 所调用。

使用 N-gram进行分割时，改造后的常规做法是从查询字符串的起始位置开始不断地取出一组组无重复部分的 n 个字符。但是，当查询字符串的长度不能被 n 整除时，就会在最后留下一 个长度小于 n 的词元。遇到这种特殊情况时，我们要将末尾的 n 个字符作为 1 个词元。

以“查询字符串”为例，我们先分别用 bi-gram和 tri-gram对其进行分割，分割结果如下所示。

- 使用 bi-gram的分割结果：“查询”“字符”“符串”
- 使用 tri-gram的分割结果：“查询字”“字符串”

像这样，当查询字符串的长度不能被 n 整除时，我们可以通过如下的策略，生成含有部分重复字符的词元。另外，我们将变量 position用作游标，指向查询中正在处理的字符。

- 当 position可以被 n 整除并且候选词元的长度不小于 n 时 →将该候选词元作为正式词元
- 当 position可以被 n 整除并且候选词元的长度小于 n 时 →将该候选词元的前一个候选词元作为词元

由于函数 ngram_next() 是通过每次向后错开 1 个字符来获取词元的，所以可以保证位于最后一个长度小于 n 的词元之前的候选词元其长度一定是 n 。因此，我们就需要定义 3 个新的变量 last_t_len、last_t 以及 last_position，用于存储前一个候选词元的信息。

```
/**
 * 为构成文档内容的字符串建立倒排列表的集合
 * @param[in] env 存储着应用程序运行环境的结构体
 * @param[in] document_id 文档编号。为0时表示把要查询的关键词作为处理对象
 * @param[in] text 输入的字符串
 * @param[in] text_len 输入的字符串的长度
 * @param[in] n N-gram中N的取值
 * @param[in,out] postings 倒排列表的数组（也可视作是指向小倒排索引的指针）。若传入的指针指向了NULL，
 *                         则表示要新建一个倒排列表的数组（小倒排索引）。若传入的指针指向了之前就已经存在的倒排列表的数组，
 *                         则表示要添加元素
 * @retval 0 成功
 * @retval -1 失败
 */
int text_to_postings_lists(wiser_env *env,
                           const int document_id, const UTF32Char *text,
                           const unsigned int text_len,
                           const int n, inverted_index_hash **postings)
{
  /* FIXME: now same document update is broken. */
  int t_len, position = 0;
  const UTF32Char *t = text, *text_end = text + text_len;
  int last_t_len = 0, last_position = 0;
  const UTF32Char *last_t = NULL;

  inverted_index_hash *buffer_postings = NULL;
```

```
for (; (t_len = ngram_next(t, text_end, n, &t)); t++, position++) {
  int filtered_t_len = 0, filtered_position;
  const UTF32Char *filtered_t = NULL;

  /* 在检索时，基本上是当position可以被n整除时才取出词元 */
  if (document_id || ((position % n == 0) && t_len >= n)) {
      filtered_t_len = t_len;
      filtered_t = t;
      filtered_position = position;
  /* 但是，要保证最后一个词元含有n个字符 */
  } else if (t_len < n) {
      if (last_t_len && last_t) {
        filtered_t_len = last_t_len;
        filtered_t = last_t;
        filtered_position = last_position;
      } else {
          break;
      }
  }

  if (filtered_t_len && filtered_t) {
    int retval, t_8_size;
    char t_8[n * MAX_UTF8_SIZE];

    utf32toutf8(filtered_t, filtered_t_len, t_8, &t_8_size);

    retval = token_to_postings_list(env, document_id, t_8, t_8_size,
                                    filtered_position, &buffer_postings);

    if (retval) { return retval; }

      last_t_len = 0;
      last_t = NULL;
    } else {
      last_t_len = t_len;
      last_t = t;
      last_position = position;
    }
}
```

分别用优化前和优化后的 wiser 检索同一个查询后可以发现，检索结果的数量并未发生变化，而检索处理的速度则有所提升。但是由于索引中收录的文档非常多，而且查询的长度又没有达到足够的长度，所以也许并不能切实感到检索处理的速度提升了。

禁用短语检索

在 wiser 中，我们会将一个个二元组存储到倒排索引的倒排列表里，二元组中包括含有某个词元的文档编号和该词元出现的位置。这样的倒排列表称为单词级别的倒排列表。而且我们还 讲解过，通过单词级别的倒排列表可以准确地找出包含在查询中的短语。

我们检索一个 3 字符的字符串。例如“第一个”。接下来，禁用短语检索后再检索一次“第一个”。

可以看出，检索结果在数量上有较大的差距。禁用短语检索前能找到 154 条结果，而禁用短语检索后竟能找到 349 条结果。究其原因可以发现，无论“第一”还是“一个”都是会在大量文档 中出现的词元。在诸如“世界上第一部（架、本……）……”“……一个时代（系统、周期……）”等句子中，就经常会遇到虽然出现了“第一”和“一个”，但是却没有出现“第一个”的情况。

改变检索结果的输出顺序

作为检索结果排序核心的指标

检索系统有时会产生大量的检索结果。此时即使是将检索结果原封不动地返回，用户也无法查阅完所有内容。因此更好的做法是根据某种指标进行评分，然后只将得分较高的文档作为 检索结果呈现给用户。

下面我们就来介绍几个用于对检索结果排序的指标（属性）。

TF-IDF

TF 是 TermFrequency（词频）的缩写，用于描述特定词元在某个特定文档中的出现次数。IDF 是 Inverse Document Frequency（逆文档频率）的缩写，指在所有的文档中，出现过特定词元的文档数的倒数。TF-IDF 值就是上述 TF 和 IDF 的乘积。

假设某个词元在某个文档中的出现次数为 T ，总共有 A 个文档，并且某个词元至少出现过 1 次的文档数为 D ，那么 TF-IDF 值的计算公式如下所示：

$$\mathrm{tf} = T$$
$$\mathrm{idf} = \log \frac{A}{D}$$
$$\mathrm{tf} - \mathrm{idf} = \mathrm{tf} \times \mathrm{idf}$$

文档的最后更新日期

对于某些作为检索对象的文档集合，搜素引擎会根据文档的最后更新日期，而不是查询与文档的相关度，对检索结果进行排序。

例如在 Twitter 的推文检索功能中，就是按照发布日期的降序来呈现检索到的推文的。之所以这样做，是因为用户想浏览的是“最近大家都在热议什么话题”。面对“用户想了解的是特定的 新闻以及大家对这条新闻的评论”这种需求，将发布日期最新的新闻放到最上面再自然不过了。而且，由于 Twitter 的用户界面在设计之初就是按照时间顺序列出推文的，所以检索结果也 沿用这种风格显示的话，浏览起来会更加方便。

PageRank

在对检索结果排序时，Google 会计算一种名为 PageRank 的独有指标，并将其结果作为决定 Web 检索结果呈现顺序的要素之一。

PageRank 的计算，是基于“从受欢迎的网页中精挑细选出的链接所指向的网页应该也很受欢迎吧”这种假设。具体来说，是基于以下 3 个推测。

- 被很多网站的链接指向的网页从某种程度上来说是优质的、受欢迎的
- 被受欢迎网页上的链接指向的网页从某种程度上来说也是优质的
- 网页上的链接的数量与搜索引擎对链接目标网页的推荐程度呈反比

由于 PageRank 的计算只与网页间的链接结构有关，所以计算时会完全忽略网页的内容。在这一点上，PageRank 与 TF-IDF 等计算时只参考文档内容的指标完全不同。

现有的 wiser 在输出前会根据 TF-IDF 值的大小对检索结果进行降序排列，尽管如此，能够在输出前使用除此之外的排序方式也是一个不错的选择。为此，作为练习，下面就让我们改造 一下 wiser，使其能够在输出前按照文档的大小对检索结果进行降序排列。