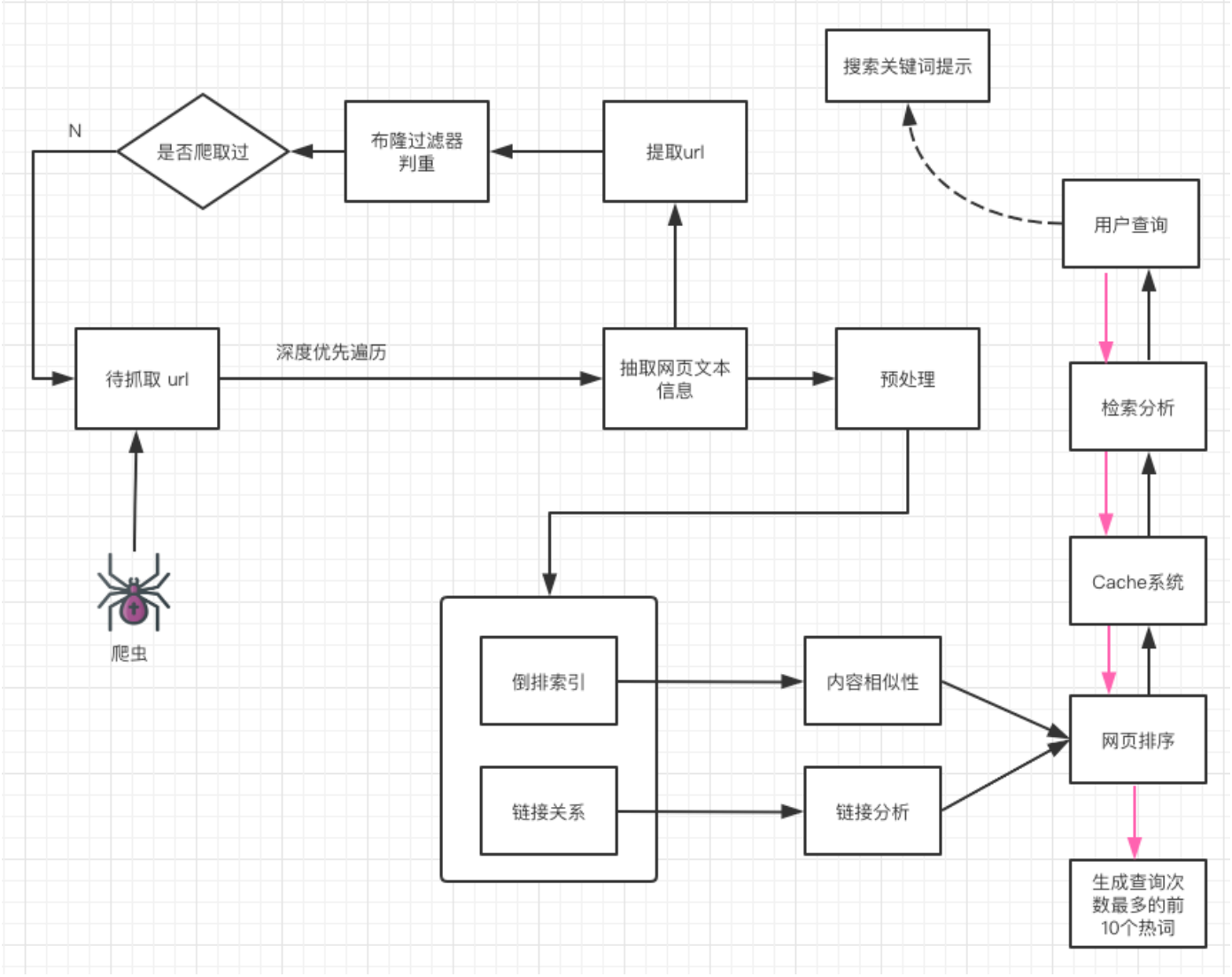
搜索引擎系统架构图

搜索引擎整体架构图如下图所示，大致可以分为搜集，预处理，索引，查询这四步，每一步的技术细节都很多，我们将在下文中详细分析每一步的工作原理。



搜索引擎工作原理详细剖析

一、搜集

即爬虫爬取文件

网络爬虫（Web crawler），是一种按照一定的规则，自动地抓取万维网信息的程序或者脚本，它们被广泛用于互联网搜索引擎或其他类似网站，可以自动采集所有其能够访问到的页面内容，以获取或更新这些网站的内容和检索方式。从功能上来讲，爬虫一般分为数据采集，处理，储存三个部分。传统爬虫从一个或若干初始网页的URL开始，获得初始网页上的URL，在抓取网页的过程中，不断从当前页面上抽取新的URL放入队列,直到满足系统的一定停止条件。

爬虫一开始是不知道该从哪里开始爬起的，所以我们可以给它一组优质种子网页的链接，比如新浪主页，腾讯主页等，这些主页比较知名，在 Alexa 排名上也非常靠前，拿到这些优质种子网页后，就对这些网页通过广度优先遍历不断遍历这些网页，爬取网页内容，提取出其中的链接，不断将其将入到待爬取队列，然后爬虫不断地从 url 的待爬取队列里提取出 url 进行爬取，重复以上过程...

当然了，只用一个爬虫是不够的，可以启动多个爬虫并行爬取，这样速度会快很多。

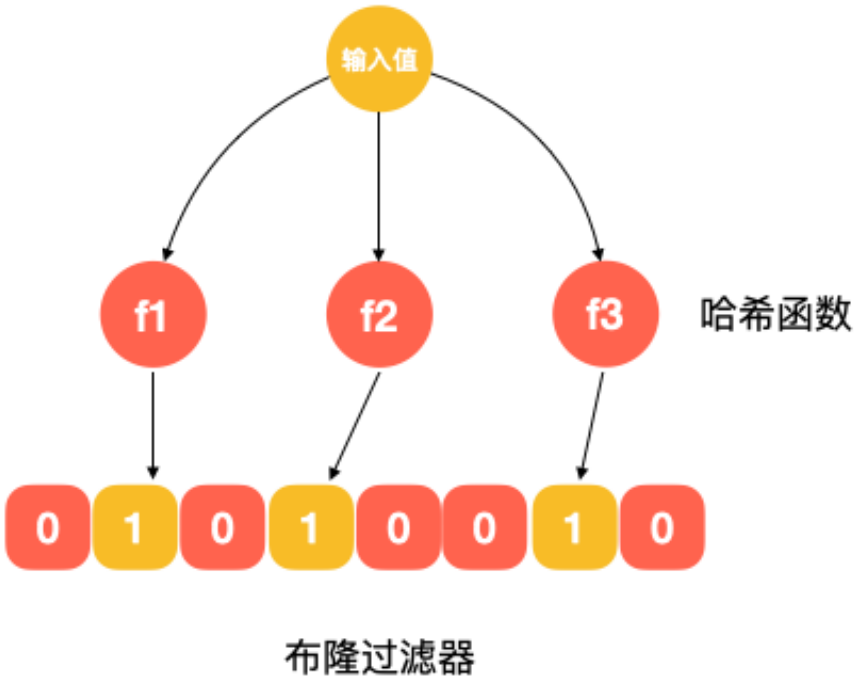
1、待爬取的 url 实现

待爬取 url 我们可以把它放到 Redis 里，保证了高性能，需要注意的是，Redis 要开启持久化功能，这样支持断点续爬，如果 Redis 挂掉了，重启之后由于有持续久功能，可以从上一个待爬的 url 开始重新爬。

2、如何判重

如何避免网页的重复爬取呢，我们需要对 url 进行去重操作，去重怎么实现？可能有人说用散列表，将每个待抓取 url 存在散列表里，每次要加入待爬取 url 时都通过这个散列表来判断一下是否爬取过了，这样做确实没有问题，但我们需要注意到的是这样需要会出巨大的空间代价，有多大，我们简单算一下，假设有 10 亿 url （不要觉得 10 亿很大，像 Google, 百度这样的搜索引擎，它们要爬取的网页量级比 10 亿大得多），放在散列表里，需要多大存储空间呢?

我们假设每个网页 url 平均长度 64 字节，则 10 亿个 url 大约需要 60 G 内存，如果用散列表实现的话，由于散列表为了避免过多的冲突，需要较小的装载因子（假设哈希表要装载 10 个元素，实际可能要分配 20 个元素的空间，以避免哈希冲突），同时不管是用链式存储还是用红黑树来处理冲突，都要存储指针，各种这些加起来所需内存可能会超过 100 G，再加上冲突时需要在链表中比较字符串，性能上也是一个损耗，当然 100 G 对大型搜索引擎来说不是什么大问题，但其实还有一种方案可以实现远小于 100 G 的内存：布隆过滤器。

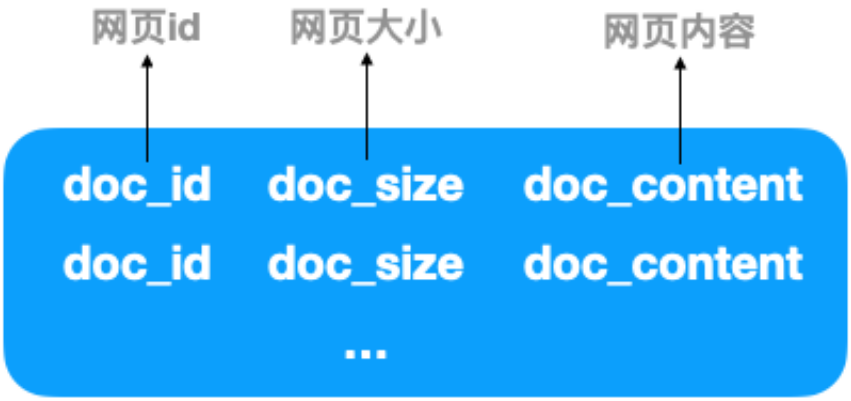


针对 10 亿个 url，我们分配 100 亿个 bit，大约 1.2 G, 相比 100 G 内存，提升了近百倍！可见技术方案的合理选择能很好地达到降本增效的效果。

当然有人可能会提出疑问，布隆过滤器可能会存在误判的情况，即某个值经过布隆过滤器判断不存在，那这个值肯定不存在，但如果经布隆过滤器判断存在，那这个值不一定存在,针对这种情况我们可以通过调整布隆过滤器的哈希函数或其底层的位图大小来尽可能地降低误判的概率，但如果误判还是发生了呢，此时针对这种 url 就不爬好了，毕竟互联网上这么多网页，少爬几个也无妨。

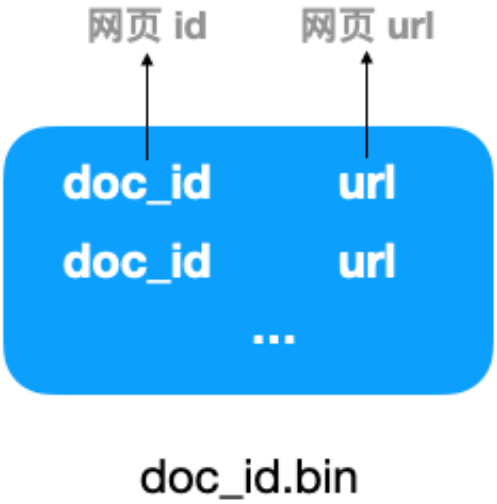
3、网页的存储文件: doc_raw.bin

爬完网页，网页该如何存储呢，有人说一个网页存一个文件不就行了，如果是这样，10 亿个网页就要存 10 亿个文件，一般的文件系统是不支持的，所以一般是把网页内容存储在一个文件（假设为 doc_raw.bin）中，如下



当然一般的文件系统对单个文件的大小也是有限制的，比如 1 G，那在文件超过 1 G 后再新建一个好了。

图中网页 id 是怎么生成的，显然一个 url 对应一个网页 id，所以我们可以增加一个发号器，每爬取完一个网页，发号器给它分配一个 id，将网页 id 与 url 存储在一个文件里，假设命名为 doc_id.bin,如下



二、预处理

把其它的 html 标签去掉（标签里的内容保留），因为我们最终要处理的是纯内容（内容里面包含用户要搜索的关键词）

三、分词并创建倒排索引

拿到上述步骤处理过的内容后，我们需要将这些内容进行分词，啥叫分词呢，就是将一段文本切分成一个个的词。比如 「I am a chinese」分词后，就有 「I」，「am」，「a」，「chinese」这四个词,从中也可以看到，英文分词相对比较简单，每个单词基本是用空格隔开的，只要以空格为分隔符切割字符串基本可达到分词效果，但是中文不一样，词与词之类没有空格等字符串分割，比较难以分割。以「我来到北京清华大学」为例，不同的模式产生的分词结果不一样，jieba 分词开源库有如下几种分词模式

```
【全模式】：我/ 来到/ 北京/ 清华/ 清华大学/ 华大/ 大学

【精确模式】：我/ 来到/ 北京/ 清华大学

【新词识别】：他，来到，了，网易，杭研，大厦

【搜索引擎模式】： 小明，硕士，毕业，于，中国，科学，学院，科学院，中国科学院，计算，计算所，后，在，日本，京都，大学，日本京都大学，深造
```

分词一般是根据现成的词库来进行匹配，比如词库中有「中国」这个词，用处理过的网页文本进行匹配即可。当然在分词之前我们要把一些无意义的停止词如「的」，「地」，「得」先给去掉。

经过分词之后我们得到了每个分词与其文本的关系，如下

```
北京   doc1
清华   doc1
大学   doc1   doc2
```

这样我们在搜「大学」的时候找到「大学」对应的行，就能找到所有包含有「大学」的文档 id 了。

此为 **分词+倒排索引**

还有一个问题，根据某个词语获取得了一组网页的 id 之后，在结果展示上，哪些网页应该排在最前面呢，为啥我们在 Google 上搜索一般在第一页的前几条就能找到我们想要的答案。这就涉及到搜索引擎涉及到的另一个重要的算法: PageRank，它是 Google 对网页排名进行排名的一种算法，它以网页之间的超链接个数和质量作为主要因素粗略地分析网页重要性以便对其进行打分。我们一般在搜问题的时候，前面一两个基本上都是 stackoverflow 网页，说明 Google 认为这个网页的权重很高，因为这个网页被全世界几乎所有的程序员使用着，也就是说有无数个网页指向此网站的链接，根据 PageRank 算法，自然此网站权重就高

完成以上步骤，搜索引擎对网页的处理就完了，那么用户输入关键词搜索引擎又是怎么给我们展示出结果的呢。

四、查询

用户输入关键词后，首先肯定是要经过分词器的处理。比如我输入「中国人民」，假设分词器分将其分为「中国」，「人民」两个词，接下来就用这个两词去倒排索引里查相应的文档

```
中国　doc1、doc2
人民　doc3、doc4
```

得到网页 id 后，我们分别去 doc_id.bin，doc_raw.bin 里提取出网页的链接和内容，按权重从大到小排列即可。

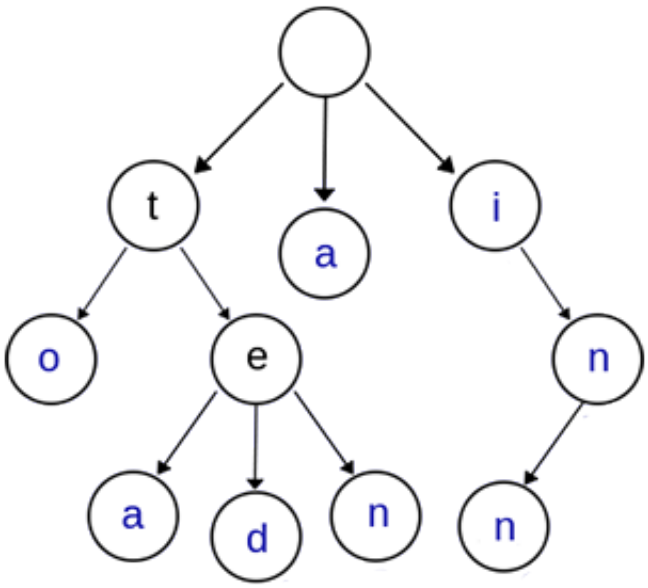
「 TF-IDF 」算法，等下讲。

另外在搜索框输入搜索词的时候，底下会出现一串搜索提示词：

如上图示：输入 chin 这四个字母后，底下会出现一列提示词。

如何实现的，这就不得不提到一种树形结构：Trie 树。

Trie 树又叫字典树、前缀树（Prefix Tree）、单词查找树，是一种多叉树结构，如下图所示：



这颗多叉树表示了关键字集合 ["to"，"tea"，"ted"，"ten"，"a"，"i"，"in", "inn"]。从中可以看出 Trie 树具有以下性质：

```
1．根节点不包含字符，除根节点外的每一个子节点都包含一个字符
2．从根节点到某一个节点，路径上经过的字符连接起来，为该节点对应的字符串
3．每个节点的所有子节点包含的字符互不相同
```

通常在实现的时候，会在节点结构中设置一个标志，用来标记该结点处是否构成一个单词（关键字）。

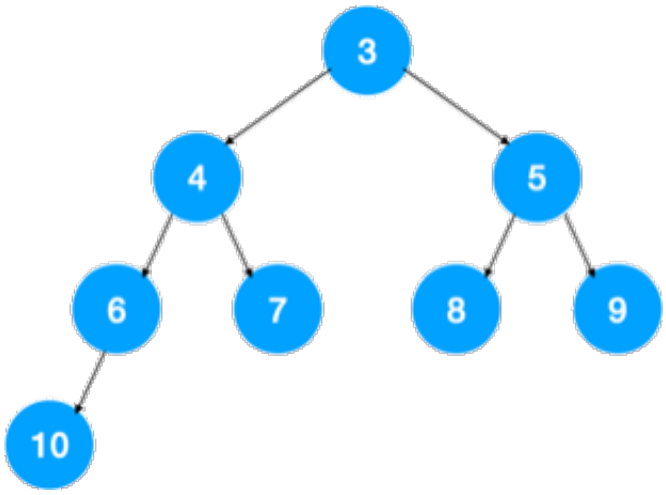
另外我们不难发现一个规律，具有公共前缀的关键字（单词），它们前缀部分在 Trie 树中是相同的，这也是 Trie 树被称为前缀树的原因，有了这个思路，我们不难设计出上文所述搜索时展示一串搜索提示词的思路：

一般搜索引擎会维护一个词库，假设这个词库由所有搜索次数大于某个阈值（如 1000）的字符串组成，我们就可以用这个词库构建一颗 Trie 树，这样当用户输入字母的时候，就可以以这个字母作为前缀去 Trie 树中查找，以上文中提到的 Trie 树为例，则我们输入「te」时，由于以「te」为前缀的单词有 ["tea"，"ted"，"ted"，"ten"]，则在搜索引擎的搜索提示框中就可以展示这几个字符串以供用户选择。

五、寻找热门搜索字符串

Trie 树除了作为前缀树来实现搜索提示词的功能外，还可以用来辅助寻找热门搜索字符串，只要对 Trie 树稍加改造即可。假设我们要寻找最热门的 10 个搜索字符串，则具体实现思路如下：

一般搜索引擎都会有专门的日志来记录用户的搜索词，我们用用户的这些搜索词来构建一颗 Trie 树，但要稍微对 Trie 树进行一下改造，上文提到，Trie 树实现的时候，可以在节点中设置一个标志，用来标记该结点处是否构成一个单词，也可以把这个标志改成以节点为终止字符的搜索字符串个数，每个搜索字符串在 Trie 树遍历，在遍历的最后一个结点上把字符串个数加 1，即可统计出每个字符串被搜索了多少次（根节点到结点经过的路径即为搜索字符串），然后我们再维护一个有 10 个节点的小顶堆（堆顶元素比所有其他元素值都小，如下图示）



如图示：小顶堆中堆顶元素比其他任何元素都小

依次遍历 Trie 树的节点，将节点（字符串+次数）传给小顶堆，根据搜索次数不断调整小顶堆，这样遍历完 Trie 树的节点后，小顶堆里的 10 个节点即是最热门的搜索字符串。

用C++写的全文搜索引擎
项目背景
现在我们可以在我们可以很容易的在网上搜索到海量的信息，有些网站往往拥有很多优秀的内容，但是缺乏一个灵活而且高效的搜索引擎，导致这个网站的价值没有很好的体现，基于时间轴的或者tag的推荐从另一种角度上来说其实是忽视了用户搜索的主观能动性。

通用的搜索引擎并不能解决这个问题：首先，不能有针对性的垂直搜索，对内容的抓取和索引简单粗暴，无法得到结构化数据，看不到隐藏的内容属性；其次，通用搜索引擎对内容的排序无法进行定制，实时性不够，无法成为网站社区的有机的一部分。

我们如果能够解决这两个问题，那么相信网站内容的价值也会更好的得到体现，同时用户也会越来越适应通过站内搜索引擎更好的获取自己需要资源。

这个项目就是为了实现这样理想的一个尝试。

功能综述

```
1  利用redis存储倒排索引和网页库(均在内存)，实现高速搜索
2  利用redis建立缓存，若在缓存直接命中，查询相应速度提高接近100倍
3  支持中文分词（cppjieba）
4  支持持久化存储(redis)
5  采用log4cpp作为日志系统
```

项目架构
离线索引模块

```
1  数据结构：{string : {ID : weight},{ID : weight}...},weight（权重）是通过TD-IDF算法并归一后的权重
2
3  存储结构：用redis 的set存储：结构为 key:string set :string(string 需要解析 是 ID 和 weight组成的字符串)
```

查询模块

```
网页排序：BM25算法 + vsm 之前采用空间向量模型和BM25算法结合（两个算法本来就有共通的地方）增加一些定制性

线程模型：task threadpool 任务队列
```

缓存模块

```
内存实现：利用unordered_map 作为缓存数据结构，每个线程维护一个缓存，线程池维护一个缓存，利用timerfd+poll实现缓存动态更新和存盘

redis实现：利用redis的string数据结构来存储缓存，缓存内容为封装好的json数组
```

网络库模块

```
网络模型：reactor模式，epoll + threadpool的实现i/o线程和计算线程分离
```

后期优化方向

```
加入推荐算法

利用Mysql存储网页库文件和索引文件

网络库采用muduo网络库的模型 实现 one loop per thread,实现更高的并发量

redis缓存相应参数的调整
```

1、TF-IDF算法介绍

TF-IDF（term frequency–inverse document frequency，词频-逆向文件频率）是一种用于信息检索（information retrieval）与文本挖掘（text mining）的常用加权技术。

```
    TF-IDF是一种统计方法，用以评估一字词对于一个文件集或一个语料库中的其中一份文件的重要程度。字词的重要性随着它在文件中出现的次数成正比增加，但同时会随着它在语料库中出现的频率成反比下降。

    TF-IDF的主要思想是：如果某个单词在一篇文章中出现的频率TF高，并且在其他文章中很少出现，则认为此词或者短语具有很好的类别区分能力，适合用来分类。
```

（1）TF是词频(Term Frequency)

词频（TF）表示词条（关键字）在文本中出现的频率。

```
这个数字通常会被归一化(一般是词频除以文章总词数)，以防止它偏向长的文件。
即：TF= 在某一类中词条w出现次数/该类中所有词条数目

其中 ni,j 是该词在文件 dj 中出现的次数，分母则是文件 dj 中所有词汇出现的次数总和；
```

（2） IDF是逆向文件频率(Inverse Document Frequency)

逆向文件频率 (IDF)：某一特定词语的IDF，可以由总文件数目除以包含该词语的文件的数目，再将得到的商取对数得到。

如果包含词条t的文档越少, IDF越大，则说明词条具有很好的类别区分能力。

```
即：IDF = log(语料库文档总数/包含词条w的文档数+1)分母+1是为了避免分母为0

其中，|D| 是语料库中的文件总数。 |{j:ti∈dj}| 表示包含词语 ti 的文件数目（即 ni,j≠0 的文件数目）。如果该词语不在语料库中，就会导致分母为零，因此一般情况下使用 1+|{j:ti∈dj}|
```

（3）TF-IDF实际上是：TF * IDF

某一特定文件内的高词语频率，以及该词语在整个文件集合中的低文件频率，可以产生出高权重的TF-IDF。因此，TF-IDF倾向于过滤掉常见的词语，保留重要的词语。

可以看到**权重系数w= TF*IDF 与一个词在文档中的出现次数成正比，与该词在整个网页库中的出现次数成反比。**

（4） 归一化(一般是词频除以文章总词数),

```
一篇文档包含多个词语 w1,w2,...,wn，还需要对这些词语的权重系数进行归一化处理，以防止它偏向长的文件。
其计算公式如下：
w' = w /sqrt(w1^2 + w2^2 +...+ wn^2)
w'才是需要保存下来的，即倒排索引的数据结构中 InvertIndexTable 的 double 类型所代表的值。
```

什么是 TF-IDF 算法?

简单来说，向量空间模型就是希望把查询关键字和文档都表达成向量，然后利用向量之间的运算来进一步表达向量间的关系。比如，一个比较常用的运算就是计算查询关键字所对应的向量和文档所对应的向量之间的 “相关度”。

TF （Term Frequency)—— “单词频率”
意思就是说，我们计算一个查询关键字中某一个单词在目标文档中出现的次数。举例说来，如果我们要查询 “Car Insurance”，那么对于每一个文档，我们都计算“Car” 这个单词在其中出现了多少次，“Insurance”这个单词在其中出现了多少次。这个就是 TF 的计算方法。

TF 背后的隐含的假设是，查询关键字中的单词应该相对于其他单词更加重要，而文档的重要程度，也就是相关度，与单词在文档中出现的次数成正比。比如，“Car” 这个单词在文档 A 里出现了 5 次，而在文档 B 里出现了 20 次，那么 TF 计算就认为文档 B 可能更相关。

然而，信息检索工作者很快就发现，仅有 TF 不能比较完整地描述文档的相关度。因为语言的因素，有一些单词可能会比较自然地在很多文档中反复出现，比如英语中的 “The”、“An”、“But” 等等。这些词大多起到了链接语句的作用，是保持语言连贯不可或缺的部分。然而，如果我们要搜索 “How to Build A Car” 这个关键词，其中的 “How”、“To” 以及 “A” 都极可能在绝大多数的文档中出现，这个时候 TF 就无法帮助我们区分文档的相关度了。

IDF（Inverse Document Frequency）—— “逆文档频率”
就在这样的情况下应运而生。这里面的思路其实很简单，那就是我们需要去 “惩罚”那些出现在太多文档中的单词。

也就是说，真正携带 “相关” 信息的单词仅仅出现在相对比较少，有时候可能是极少数的文档里。这个信息，很容易用 “文档频率” 来计算，也就是，有多少文档涵盖了这个单词。很明显，如果有太多文档都涵盖了某个单词，这个单词也就越不重要，或者说是这个单词就越没有信息量。因此，我们需要对 TF 的值进行修正，而 IDF 的想法是用 DF 的倒数来进行修正。倒数的应用正好表达了这样的思想，DF 值越大越不重要。

TF-IDF 算法主要适用于英文，中文首先要分词，分词后要解决多词一义，以及一词多义问题，这两个问题通过简单的tf-idf方法不能很好的解决。于是就有了后来的词嵌入方法，用向量来表征一个词。

TF-IDF计算方式（表格版）

假设有语料库一共只要2篇文档：d_1和d_2，其中
$d_1 = (A, B, C, D, A)$一共有5个单词组成；$d_2 = (B, E, A, B)$，一共有4个单词组成。

1.TF

TF即词频(Term Frequency)，每篇文档中关键词的频率（该文档单词/该文档单词总数），对于文档d_1和文档d_2有：

	d_1	d_2
A	$\frac{2}{5}$	$\frac{1}{4}$
B	$\frac{1}{5}$	$\frac{2}{4}$
C	$\frac{1}{5}$	$\frac{0}{4}$
D	$\frac{1}{5}$	$\frac{0}{4}$
E	$\frac{0}{5}$	$\frac{1}{4}$

注意：由语料库得到的字典长度为5，所以最终文档向量化长度为5。

2.IDF

IDF即逆文档频率(Inverse Document Frequency)，文档总数/关键词t出现的文档数目，即$IDF(t) = ln((1 + |D|)/|D_t|)$（还有log等形式，自然对数被证明是最有效的一个公式），计算语料库中每个关键词的IDF值如下：

A	$ln(\frac{1+2}{2})$
B	$ln(\frac{1+2}{2})$
C	$ln(\frac{1+2}{1})$
D	$ln(\frac{1+2}{1})$
E	$ln(\frac{1+2}{1})$

3.结合IF-IDF，文档的向量化表示

举例d_1:

$$d_1 = (x_1, x_2, x_3, x_4, x_5) = (\frac{2}{5} \times ln(\frac{1+2}{2}), \frac{1}{5} \times ln(\frac{1+2}{2}), \frac{1}{5} \times ln(\frac{1+2}{1}), \frac{1}{5} \times ln(\frac{1+2}{1}), \frac{0}{5} \times ln(\frac{1+2}{1}))$$

TF-IDF的变种：

1. 变种1：对数函数变换TF　　解决TF线性增长问题
2. 变种2：对TF进行标准化　　解决长短文档问题
3. 变种3：对数函数变换IDF　　解决IDF线性增长问题
4. 变种4：查询词及文档向量标准化　　解决长短文档问题

变种1：通过对数函数避免 TF 线性增长

很多人注意到 TF 的值在原始的定义中没有任何上限。虽然我们一般认为一个文档包含查询关键词多次相对来说表达了某种相关度，但这样的关系很难说是线性的。拿我们刚才举过的关于 “Car Insurance” 的例子来说，文档 A 可能包含 “Car” 这个词 100 次，而文档 B 可能包含 200 次，是不是说文档 B 的相关度就是文档 A 的 2 倍呢？其实，很多人意识到，超过了某个阈值之后，这个 TF 也就没那么有区分度了。

用 Log，也就是对数函数，对 TF 进行变换，就是一个不让 TF 线性增长的技巧。具体来说，人们常常用 1+Log(TF) 这个值来代替原来的 TF 取值。在这样新的计算下，假设 “Car” 出现一次，新的值是 1，出现 100 次，新的值是 5.6，而出现 200 次，新的值是 6.3。很明显，这样的计算保持了一个平衡，既有区分度，但也不至于完全线性增长。

变种2：标准化解决长文档、短文档问题

经典的计算并没有考虑 “长文档” 和“短文档”的区别。一个文档 A 有 3,000 个单词，一个文档 B 有 250 个单词，很明显，即便 “Car” 在这两个文档中都同样出现过 20 次，也不能说这两个文档都同等相关。对 TF 进行 “标准化”（Normalization），特别是根据文档的最大 TF 值进行的标准化，成了另外一个比较常用的技巧。

对序列 $x_1, x_2, \ldots, x_n$ 进行变换：

$$y_i = \frac{x_i - \bar{x}}{s}，\text{这里}\ \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i，\quad s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

则新序列 $y_1, y_2, \ldots, y_n$ 的均值为 0，而方差为 1，且无量纲。

变种3：对数函数处理 IDF

第三个常用的技巧，也是利用了对数函数进行变换的，是对 IDF 进行处理。相对于直接使用 IDF 来作为 “惩罚因素”，我们可以使用 N+1 然后除以 DF 作为一个新的 DF 的倒数，并且再在这个基础上通过一个对数变化。这里的 N 是所有文档的总数。这样做的好处就是，第一，使用了文档总数来做标准化，很类似上面提到的标准化的思路；第二，利用对数来达到非线性增长的目的。

变种4：查询词及文档向量标准化

还有一个重要的 TF-IDF 变种，则是对查询关键字向量，以及文档向量进行标准化，使得这些向量能够不受向量里有效元素多少的影响，也就是不同的文档可能有不同的长度。在线性代数里，可以把向量都标准化为一个单位向量的长度。这个时候再进行点积运算，就相当于在原来的向量上进行余弦相似度的运算。所以，另外一个角度利用这个规则就是直接在多数时候进行余弦相似度运算，以代替点积运算。