

Google 搜索引擎算法的秘密

撰文 / Julian Bucknall 编译 / 丁宁

Google和其他Web搜索引擎是如何让你用一系列的词来找到相关的网页?它们不可能先下载十几亿个网页然后找出你想要的内容吧?它们应该进行了一些预处理,并建立了索引。这篇文章讨论的就是关于这些的事。

关键词: 搜索引擎 文本搜索算法 RLE压缩算法

Google是怎么工作的

各位好。现在你们可能正在议论某些事情,或者正在埋头苦干,而我正在看尊敬的编辑大人的Email。我在这行字下打住了:“是否还记得我曾经提起的建议……”。然后,我就起身去寻找相关的算法了。

我赶紧向Malory Towers的讲师请教有关的算法,在那里珍藏了很多算法的书,这件事搞得我满头大汗。虽然我找到了Google,但是还是没弄明白:它究竟是怎么干的呢?

经过痛苦的摸索和认真的思考后,我有了些模糊的思路:解决问题的关键在于文本搜索。Google和其他Web搜索引擎是如何让你用一系列的词来找到相关的网页?它们不可能先下载十几亿个网页然后找出你想要的内容吧?它们应该进行了一些预处理,并建立了索引。这篇文章讨论的就是关于这些的事。

基本的设计

让我们先看一些假设和定义。首先,假设我们有无数的文档存放在某种数据库里等着我们去搜索。也就是说,一个文本就是一个单独的对象。它可以是HTML页面、新闻组的邮件、Word文档或是其他什么的。而数据库呢?我并不是指真正的数据库,并不是Oracle或者SQL Server什么的,它只是一个用来存放文档的“容器”。它可以是你机器里的一个文件夹,也可以是一个真实的、将文档以二进制流的方式存储的数据库。文本被当作ASCII流处理,换句话说,是一堆用标点符号和空格分隔开的单词。我决定不考虑索引字的问题和PDF文档,谢谢大家的合作。

当我们进行搜索的时候,一些保留字要特殊处理,它们是AND、OR和NOT——这些是逻辑运算符。这样说吧,我们要找同时有“text”和“indexing”的文档时,需要用text AND indexing作为搜索样本,而需要找有“text”或者“indexing”的文档时,就用text OR indexing作为搜索样本。如果我们给出一串不带AND和OR的单词,搜索引擎就默认这些单词是用AND分隔的,因此text indexing被当作text AND indexing处理。顾名思义,NOT是取反运算符。我们将NOT定为最高优先级,AND和OR优先级一样,并从左到右进行求值。那么NOT text AND

indexing就是(NOT text) AND (indexing)。

经过以上定义,我肯定你可以预见到这篇文章将会分成哪几个子标题。首先,我们需要将文档解析为单独的单词。第二,我们需要建立单词的索引便于搜索。老套路了,不过我想,在没有看完整篇文章之前先不要下定论。

从A到Z

现在我们循序渐进,先从简单的开始。第三个运算符看起来是最简单的了,我们从这里开始。

一提起写语法解析器,人们通常都会想起容易下手的文法解析,也就是我们要使用的BNF。表1说明了用来进行文法解析的BNF(;;=表示产生式,|表示可选。第一个定义表示expr可以是一个factor、或者是一个跟在factor以及AND后面的另一个expr、还可以是一个跟在factor以及OR后面的另一个expr。其他的定义以此类推)。

表1: 搜索短语的BNF

```
<expr> ::= <factor> <factor> <expr> <factor> AND <expr> <factor> OR <expr>
<factor> ::= <term> | NOT <factor>
<term> ::= <word> | (expr)
<word> ::= 以空格为结尾的一系列字符
```

写下这个BNF后,将它转化成代码再简单不过了。我们写一个语法解析类,为每个产生式写一个叫ParseXxxx的方法,这个Xxx就是该产生式左边的名称。而代码则实现产生式右边的内容。这就是众所周知的递归下降解析器。

让我们先捡软的下手: ParseFactor。该产生式生成一个term或NOT,其后跟随另一个factor。所以ParseFactor首先要检查当前的单词是否为NOT。如果是,我们需要递归调用ParseFactor;如果不是则调用ParseTerm。其他的方法写起来也是大同小异。

在我们动手前,还有一些事情要考虑一下。搜索短语解析器非常容易实现,但是我们如何保存解析的结果呢?我们需要建立一个可以用于单词索引的数据结构。我们使用逆波兰式(RPN)来存储解析的结果,逆波兰式将操作符放在操作数的后面。比如word1 AND word2这个查询将会被存储为word1 word2 AND; NOT (word1 OR word2)将被存储为word1 word2 OR NOT,诸如此类。你以后将会发现,我将这个方案用于解析串。

代码列表1演示了TaaSearchParser类,用来生成RPN串。

(如果你仔细看代码的话会发现R P N串使用一个链表TaaRPNNodes来存储,而不是一个“串”。)再看factor的方法spParseFactor,你会看到它先检查该单词是否为NOT。如果不是,该方法调用spParseTerm方法并将其结果返回;如果是NOT,该方法继续调用spParseFactor方法,而新的节点NOT被加到了链表中。TaaRPNNode类的Append方法实现“将一个节点添加到列表中”的功能。其他的产生式方法类似。

搜索短语解析时出的错都会在spParseTerm方法里被截获(它是BNF文法的最底层)。这些错误包括:缺少右括号;使用了运算符(AND、OR、NOT)作为搜索短语(比如: this AND and);还有缺少单词(比如: this AND)。

这里有个有趣的方法就是spGetNextWord,它用来在搜索短语取得单个的单词。我们假设搜索短语很简洁,由一些用空格分开的单词组成。但是总免不了有些意外,我们还得注意括号的存在,例如“NOT (this AND that)”将会被解析成“NOT”、“(this”、“AND”和“that)”。所以我们还要把括号当作分隔符处理。

这简直就是魔法

很好,上面的只是小点心,现在我们开始大餐:建立索引。第一步是解析出文本中的所有单词。

在搜索短语解析器中,我们已经可以将短语中的单词一个地读出了。所以要在文档的处理中做同样的工作,可谓是轻车熟路。不过,这次定义的规则要复杂一些。是的,我在研究Google时在各种杂乱的代码里陷入了一片昏暗中。

现在把我的发现进行组织,然后压缩一下。这是学校教导我们的话语:“空格是分隔符”。那就是说,我们在文本中一个字符接一个字符地找,碰到了空格就算是到了单词的结尾。就这样不停的重复,直到文本中所有的单词都被找到。

这么单纯的算法好像无法满足我们的需要。比如说,标点符号是无需分离出来处理作为结果的。算法还有一些问题:我们在遇到连续几个空格的时候,得到的结果是不正确的,但是无法发现。所以我们在查找单词的时候还要扫描多个空格。

学校教导我们的另一句话是:“单词值包含字母和数字”。我们有两种选择:搜索包含文字和数字的单词,或者搜索单词之间的空格和标点符号(也就是非字母和数字的字符)。这很简单,一个普通的程序员能在15分钟内立马搞定。但是句子里如果包含“It's non-beginner”的话,将会把[It's]分成两个单词[It]和[s],[non-beginner]分成[non]和[beginner],但这不是我们需要的。

我们需要一个折衷的方案。我们将分隔符字符当作参数传递给程序(最好是一个字符串),而该程序是一个状态机。对于第二个例子,什么时候进行状态转换由当前的字符决定,或者是分隔符参数来决定。

这个方法很好,但是我们该如何挖掘出我们需要的单词呢?在回答这个问题前,我们先走个弯路。

欢迎回到真实世界

为了更好地说明后面的算法细节,我们先要有个可靠而良好的设计:先取一串文本,将其中的单词提取出来,单词由一些分隔符来分隔。现在我开始深入如何使用这些单词的细节。是的,现在开始有如何实现这个单词解析器的感觉了,我们是不是应该开始下一步了?但是为了对得起我头上的程序员帽子,我得让代码更通用。如果我实现了这样一个解析器,在解析的同时却在想着我该如何使用这些单词,那么这个解析器将会困扰着我。

这样来说吧,我们应该将处理单词的代码从提取程序里合理地分解出来。编程中拆分代码是一件很不错的事:它提高了现有代码的复用性。

让我们继续讨论下去。这样做的结果是编写一个通用的单词解析器实体,我还可以将它用到其他的方案中:实现单词计数程序、在文本中查找单词的程序、或者分析作者的词汇。如果不这样做,我们的单词解析器只能用于今天的内容。

当然,这还有另一个重要的原因:将代码拆分开,你以后要改进算法的时候只要单独某个部分就可以了。如果不拆分,那麻烦就来了。(如果你要的不是解析器呢,你需要处理每个单词吗?情况更复杂。)

那么,将“单词处理器”从“单词解析器”里分离出来,这样我写这个解析器就更容易一些了。还可以先写一个简单的“单词处理器”程序来做测试,比如说,将单词一行一行地写到文本中去。下面,我将写一个程序来处理单词,用来测试单词解析工作是否正常。

让我们开始吧,我将用单词解析程序解析一个包含要处理的文本的流,再用我所说的方法,利用一个包含分隔符的字符串。现在有两种提取出文本中单词的方法:第一种,将它们都塞到一个容器里(字符串列表或文本文件),这样数据量会很大;第二种,每次解析到一个单词就给单词处理器发一个事件,由单词处理器决定怎么处理(是保存、统计、分别进行处理,或是别的什么)。我选择后者,主要原因是:它占用很少的内存,而且如何处理单词取决于它调用的代码。

那么,最后的参数是一个行为程序,它用来进行一些操作。每当一个单词被提取出来,该程序会被调用以对这个单词进行处理。我们的首个行为程序只是将单词输出到文本文件中,但是,接下来我们将要写大量的代码来完成我们的要求。通常,我会把行为程序写成一个简单的过程,但是你不觉得那是八十年代的单纯玩艺么?所以,我将它写成了对象的方法。

哎哟!总算是结束了,列表2演示了这个解析器。你看两眼就会知道,我将小于等于空格的字符都当作了分隔符

(比如说回车、换行、Tab和空格：它们都称作不可见字符)。因此分隔符串只需要包括可见的标点符号即可。

✍ 直入云霄

接下来是什么呢？对了，我们该对文本中取出的单词进行处理。记住：我们的目的是为了重复搜索而建立一个单词索引。理所当然，我们得建一个巨大的结构，然后按键值存放一些单元。每个单元需要包含哪些内容呢？首先是单词，接着就是索引用的键值，接下来还应该包含一个文档列表，这些文档都含有该单词。呵呵，你肯定猜出来了：任何时候，只要你给出单词，我很容易地就能找到哪些文档包含了该单词。这个结构就是所谓的倒向文件。

当然了，有些单词是无需索引的，因为几乎每个文档都会包含它们，就象是a、the、of、you、and、or等等。这些单词被称为停止字，一旦我们得到这样的单词就意味着我们无需处理它。停止字无需另行存一个列表，这是个非常主观的判断。我们的出发点是，我们应该用足够简单的列表在Internet上进行查询。

说起这件事，我不由得脸红了起来。当初我说过，反向文件是巨大的，事实上它大得难以想象。猜想一下，可能有几百万个文档都包含了某个单词，索引中每个单元的大小可能会有几十（甚至几百）兆字节，而这些单元的数量本身也有几千个（几百万也是有可能的）。我可不想这样。

首要的问题就是：如何减少每个单词的文档列表占用的空间？可以让这些列表不包含文档的真实名称。我们可以将文档的名称存放在一个数组里，而每个单元的列表包含的是文档在该数组中的编号（换句话说，就是用给出的编号在数组里找到文档的名称）。

每个索引单元占用的资源还是很多，然而已经方便管理多了。但是，现在的状况还不能让我们进入下一个阶段：使用索引应用于搜索短语。如果搜索短语只包含一个单词，那还可以应付，但是不要忘了，搜索短语中还会包含逻辑运算符AND、OR和NOT。我们如何处理呢？

假设我们现在的搜索短语是text AND indexing，我们查找到这两个单词并得到两个文档列表。这时为了满足AND的条件，需要对这两个列表筛选（时刻提醒自己注意：文档列表要排好序）。设想一下，每个列表都是竖直放置，一个挨着一个。查看两个列表的顶部，如果它们不相等，去掉小的那个，然后接着比较；如果相等，就两个都去掉并将结果加入到存放结果的列表中（记得按编号排序），接着继续比较原来的两个列表。我可以肯定你已经知道了其中的玄机，结果列表中得到的都是原来的两个列表中都包含的文档编号，也就是说，我们完成了AND操作。

对于OR，我们的工作差不多，不同的是在比较中不相同的编号也会加入到结果列表中去。至于NOT，它就有点复杂了，不

过你肯定能想到主意。（我提示一下重点，本质在于两个列表的其中一个可以为空。这将作为一个简单的习题留给读者。）

事情渐渐地开始麻烦起来：我们必须维持列表升序排列。我们必须为每个逻辑运算符编写进行合并的操作，还必须注意是否会遇到列表不够大的问题。如果我们这个问题不加关注，那么就会因为需要不停增加、列表大小而导致效率低下。

噢，如果我们可以使用一些漂亮的逻辑算法来替代合并杂乱的整数列表该有多好。幸运的是，我们可以这样做。1999年12月，我曾经写过一个bitset类，用于存放布尔型值的数组（比如on或者off、true或者false），并提供了AND、OR和NOT操作的方法。如果我们将索引中每个单词加上一个bitset来替换整数编号列表呢？每个位对应某个文档，如果第n个位被置为1，那么第n号文档就包含该单词，反之不包含。现在搜索操作实现起来就简单多了：只要用bitset的逻辑算法就可以了。

✍ 这些都很重要

虽然这个问题过于细节化，但是还是有必要提一下。bitset的每个位对应某个文档，由于很多文档没有包含该字节，相应的没有对应文档的位将占大多数而被清除。在bitset类中，每8个位被聚成一个字节，所以大部分的字节都为0。这说明在通常情况下，bitset有着相当高的压缩率，很适合用于游长码压缩编码（RLE）。

RLE是一种简单的压缩方案，原理是将一串相等的字节用一个代表长度值的字节并跟随一个原来的字节（文字字节）来替换。有趣的是：当游长为1的时候，你该怎么编码？一个为1的长度字节并跟随原来的字节？那么我们压缩以后的大小比没有压缩前的还要大。我们也不能只输出文字字节，我们希望能有更好的情况：对RLE中文字字节的位置不加限制。遗憾的是，没有一种标准的方法可以解决这个难题。

我决定用一个单独的位做状态位，它标识下一个字节是文字字节，还是长度字节并紧随着文字字节。事实上，情况很杂乱：简单输出单个的位将会引起瓶颈。我又不想输出布尔字节，那太大了。最后，我为了避免向输出的缓冲区写入单个的位，我将状态位每八个一次聚集成一个字节在数据前面输出，该字节包含了八个状态位。我们作了最坏的打算（每个游长都是1），RLE压缩算法得出的结果是增加了八分之一的大小。

然后，我修改了bitset类，可以存储和读取RLE压缩数据流。为了节省解压缩数据的开销，我改动了bitset类，让它将解压缩推迟到需要数据的时候才进行。列表3演示了bitset类的压缩和解压缩。

✍ 揭开神秘的面纱

现在让我们复习一下。我们的单词索引包含了一些列表元，这些单元按照单词的关键字排列，每个单元包含一个单

词和一个bitset。bitset里的每个位如果被置为1，那么它对应某个文档，该文档包含了这个单词。有一个文档名称数组，数组的下标对应了bitset的位编号。

我们在单词索引上已经走得太远了，现在应该考虑一下我们的数据结构了，这取决于我们要记录的单词数量。如果这个数目很大，有几万或几十万，我们可以使用B-tree或是数据库引擎；如果数目比较小（不超过1万），那么哈希表是个不错的选择。为了使本文简洁一些，我决定使用哈希表。现在，我已经实现了它，下面要做的仅仅是在流中存储或读取数据。我特地实现了一个Iterate方法来获得一个迭代子类的实例，通过它来遍历哈希表，使用它可以完全地将单词索引（单词以及bitset都可以）存储到流中，反之从流中读取也是非常简单的。列表4演示了我实现的单词索引类（异常简单）。该类提供了简单的哈希表，外包在读取和写入到流/文件的操作上。

在这里，我们可以做些真实的工作。无论什么单词索引，首先总是要建立索引。我创建了一个文件，用于保存大量文本文件的名称。这很适合我的文档列表，我们可以方便地将这个列表读取到某个字符串列表，还可以取得每个文本文件在列表中的编号。

我们必须建立一个哈希表用来包含停止字，这样就可以使用一个文本文件来包含适当的停止字列表，并使用自己设计的单词解析器来解析它。对每个遇到的单词，我们都把它加入到停止字哈希表中去。

现在我们有了一个文档列表，就可以建立这个单词索引了。我先建立一个包含数目为10000的哈希表。（哈希表会自己增长，但是最好不要频繁地让它重新分配和自增长，所以先建大一点可以提高效率。）

下一步是解析每个文档，并分离所有的单词。对于每个单词，我们先看它是不是在哈希表里。如果是，那么忽略它；如果不是，我们再检查一下它是不是已经在单词索引里；如果不是，我们建立一个新的bitset，将对应文档的位置1，将这个bitset随着单词加到单词索引中去，今后我们就通过这个bitset对象来设置对应文档的位。

为我们的辛勤而喝彩吧

完成了！我们创建了自己的单词索引，现在是使用它的时候了。是否记得文章一开始我们写的搜索短语解析器？对了，现在就是它上场的时候了。

该解析器产生一个RPN表达式。为了测试它，我们需要使用一个标准的、基于堆栈的算法。首先，要建立一个堆栈来放置bitset对象。我们遍历一下RPN表达式，在索引中查找表达式中的每个单词（如果没有找到这个单词，就使用最新创建的bitset来作为结果），接着将结果压入堆栈。对于AND和OR运算符，我们弹出栈顶部的两个bitset，对这两个bitset

使用运算符相应的操作，将得到的结果压入栈中（第二个bitset会被释放，因为它再也没有用处了）。而NOT运算符呢，我们只要弹出顶部的一个bitset，再对它使用NOT运算，将它压回栈中。这样直到RPN表达式的结尾，最终只剩下一个bitset，就在栈的顶端，而它就是查询的结果。

哈，这是怎样的一个结果啊！每个被置为1的位表示对应的文档和这次搜索有关；如果一个位都没有被置为1，就说明没有任何文档符合这次搜索。经过上面的预备，如何实现已经呼之欲出了，你觉得呢？

列表5演示了实现搜索的代码。

新鲜的思路

看完了上面的单词搜索的基本要素，我们来谈谈它的一些局限性。感兴趣的读者可能已经准备作出一番改进了。

首先是搜索解析器没有过滤掉我们指定的停止字。这看上去有点愚蠢，我们搜索停止字（它们经常出现在任何文档中）而结果是没有文档包含它们。

第二件事是单词解析器只能简单地在文本中挑出单词。因此，index、indexes、indexing等等都被当作不同的单词而被单独的处理。有些实现使用了一种单词填补算法，做法是将每个单词都简化到原型。这样，我们搜索index，将会得到与indexes、indexing、indexed等有关的东西。换句话说，搜索是单一的而结果是多重的，如果单词是个动词的话会得到不同的时态语法。原理是：它使用了一个特殊的单词列表，每行都包含了一个根单词，后面是它不同的变体。所以，这个列表要在进行单词索引前就被载入并进行解析。

接下来，bitset需要改进。为了寻找匹配的结果，我们一般要将bitset中的每个位都读一遍，看看它们是否被置为1。有一种十分快速的方法，适用于成千上万文档的查找，我们将它加到了bitset类中：首先扫描那些不为0的字节。然后对这些不为0的字节进行一位一位的查找。这个方法要比我在列表5种提供的方法快上很多。

最后的改进将会引起强烈的震动。前面指出，单词索引只是说明了“哪些文档包含了某个单词”，如果保存文档编号的列表将会更好。这是一个巨大的改变，它意味着：我们不再用bitset。每个单词和文档将会组成一个编号的列表（比方说，行和列）。这在用户需要知道在文档中单词出现的次数时会更好一些，而且我们可以在搜索文本的时候就得到它。不过你要清楚，这将会使得NOT操作变得难以实现（如何选择由你决定）。

到这里，我们渐渐接近文章的尾声了。有趣的是，我在文章中大量复用以前的代码和算法。这只是在说明：如果你有出色的算法和数据结构的基础，你可以结合具体情况更好地使用它，而且还可以做得更好。

本文相关源代码可以在CSDN网站《程序员》专区下载。 