信息科学系
低年级讨论班

（3）

裘宗燕

北京大学数学学院信息科学系

2010年3月～6月

检索

《代码之美》第**4**章

问题

- 问题：在大量数据中定位某些数据，也就是“查找”或“检索”

  检索是大部分程序中经常做的工作

- 本章通过一个简单具体问题，分析查找过程中对各种因素的权衡，提出一些有趣的技术和问题，其中特别关注“时间开销”

- 问题：作者博客 **http://www.tbray.org/ongoing** 有大量访问。到**2006**年作者撰写本文时，日志记录了 **140,070,104** 条信息（**1.4** 亿条），占用 **28,489,788,532** 字节存储空间（约 **28 G** 字节）

- 一个日志行的形式（**>200**字符）：

  c80-216-32-218.cm-upc.chello.se - - [08/Oct/2006:06:37:48 -0700] "GET /ongoing/When/200x/2006/10/08/Grief-Lessons HTTP/1.1" 200 5945 "http://www.tbray.org/ongoing/" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)

  某人**2006**年**10**月**8**日从瑞典**chello**匿名登录，在 **Windows NT** 上用 **IE6.0** 浏览器，通过 **HTTP/1.1** 协议请求文档 **Grief-Lessons**，该请求从博客首页得到文件链接，被成功处理，返回 **5945** 字节的回复，等等

问题

- 分析一个网站的服务器日志，可以得到很多有用信息。如
  - 哪些 **IP** 来的访问量大
  - 哪些文章最受欢迎
  - 下面先考虑：哪篇文章阅读次数最多
- 人工检索完全不实际。要用计算机检索，就要描述被检索的串
  - 要做的是检索类似下面形式的串的文本行（日志行）

    /ongoing/When/200x/2006/10/08/Grief-Lessons
  - 直接写代码查找太麻烦；直接写要求匹配的串，要写太多
  - 这些做法太费事，而且缺乏灵活性（需求可能变化）
- 最好是用正则表达式描述所需串的匹配模式

  从应用看，正则表达式就是写文本匹配模式的语言。学会正则表达式，能很方便地做许多文本处理问题。好程序员都应该熟悉它

正则表达式

- 由于作者博客的文件组织结构，可以用一个很简单的正则表达式描述需要在日志文件里检索访问博客文件的行：

  "GET /ongoing/When/\d\d\dx/\d\d\d\d/\d\d/\d\d/[^ .]+ "

  对其他检索问题，写出的表达式可能更简单或更复杂

- 解释：（实例：**/ongoing/When/200x/2006/10/08/Grief-Lessons**）

  开始是一串正常字符

  \d 与数字匹配，**x**，**/** 都是普通字符

  [^ .] 与任何非空格非圆点字符匹配，**+** 表示匹配一个或多个

  最后的空格表示要求匹配一个空格

  模式最后的 **[^ .]+** 能匹配 **Grief-Lessons**，不匹配 **IMG0038.jpg**

- 下面用这个正则表达式检索信息
- 可直接在命令行中调用工具程序，如 **Unix** 的 **grep**。下面考虑用 **Ruby**。这是一种目前使用较广的脚本语言，是一种面向对象语言

用正则表达式检索

示例 **4-1** 输出所有得到的有关文章的行

```
1 ARGF.each_line do |line|
2   if line =~ %r{GET /ongoing/When/\d\d\dx/\d\d\d\d/\d\d/\d\d/[^ .]+ }
3     puts line
4   end
5 end
```

- 解释：

  ARGF 是代表任意输入源的特殊变量，输入可以来自文件或管道。如果执行本程序时给了参数，**ARGF** 就把它们看作文件名

  do 描述一个循环，**each_line** 是针对指定输入的一个方法，它逐一返回文件里的行，文件结束返回假。循环里的 **|line|** 要求将输入的行约束于变量 **line**，以便在循环里处理

  =~ 表示模式匹配。循环中的条件语句说：如果该行与给定模式匹配就输出它（其中还有些细节，可以参考 **Ruby** 相关资料）

用正则表达式检索

输出整个行不合适，下面程序输出匹配得到的文件名

```
1 ARGF.each_line do |line|
2   if line =~ %r{GET /ongoing/When/\d\d\dx/(\d\d\d\d/\d\d/\d\d/[^ .]+) }
3     puts $1
4   end
5 end
```

- 修改：只加了一对括号（红色标出）

  在 **Ruby** 和另一些支持正则表达式的语言里，可以用括号标出关注的模式段，在程序里用 **$1** 表示被第一个括号匹配的串，**$2, $3 ...**

- 输出示例：

```
2003/10/10/FooCampMacs
2006/11/13/Rough-Mix
2003/05/22/StudentLookup
2003/11/13/FlyToYokohama
2003/07/31/PerlAngst
2003/05/21/RDFNet
2003/02/23/Democracy
2005/12/30/Spolsky-Recursion
2004/05/08/Torture
2004/04/27/RSSticker
```

讨论

- 正则表达式使工作变得简单清晰，易修改
  - 日志文件的格式不同，检索需求新的或修改的，很容易通过修改这段代码去完成。代码容易修改，有很多重用可能性
  - 设想如何自己写代码完成这一工作，比较一下！
- 正则表达式有漂亮的理论基础，可转换到高效的有穷自动机，实现高效文本匹配。自己写代码需要仔细优化，也未必高效
- 作者特别赞赏 **Ruby** 语言的设计。大家可以关注一下这个语言
- 使用正则表达式是双赢选择
  - 程序员可以减少开发的时间代价
  - 作为开发结果的程序也非常高效

  [当然，这些说法也可以置疑，如程序的高效问题。从正则表达式到自动机的转换，复杂性很高]

基于内容定位的存储系统

- 下面考虑核心问题：文章的受欢迎程度。为此需要
  - 给包含文章名的记录增加一个计数值
  - 根据文章名找出记录，将其加一
  - 最基本的检索问题：用一个关键码检索数据记录。为完成此工作，非常希望有一种能基于内容定位的存储系统
- 计算机硬件中也用到这类机制处理一些底层工作。而许多语言和库为此提供了专门的支持
  - 数据结构里的**Hash**表或字典提供的就是这种功能
  - 不同语言里称谓不同：**map**，**hashmap**，**dictionary**等
- 可以自己写实现，但最好还是用系统或库提供的实现
  - 节约开发时间和成本
  - 语言和库的实现者通常都下了很大功夫（参考第 **18** 章）
  - 如果现成实现不能很好满足实际的特殊要求，再考虑自己开发

统计访问次数

示例 **4-3**，统计文章访问次数

```
1 counts = {}
2 counts.default = 0
3
4 ARGF.each_line do |line|
5   if line =~ %r{GET /ongoing/When/\d\d\dx/(\d\d\d\d/\d\d/\d\d/[^ .]+) }
6     counts[$1] += 1
7   end
8 end
```

- 解释：

  第**1**行定义了一个空**hash**表，第**2**行给它的元素定义默认值**0**

  循环中用文章名检索，将相应值（计数值）加一

  Ruby 中检索不到的项自动设默认值，这里就是设**0**。这种设计使程序员非常方便，可以直接写 **counts[$1] += 1**

检索最受欢迎的10篇文章

示例4-4，最受欢迎的文章

```
1 counts = {}
2 counts.default = 0
4 ARGF.each_line do |line|
5   if line =~ %r{GET /ongoing/When/\d\d\dx/(\d\d\d\d/\d\d/\d\d/[^ .]+) }
6     counts[$1] += 1
7   end
8 end
9
10 keys_by_count = counts.keys.sort { |a, b| counts[b] <=> counts[a] }
11 keys_by_count[0 .. 9].each do |key|
12   puts "#{counts[key]}: #{key}"
13 end
```

- 10行：将 counts 表内容按关键码排序放入另一表 keys_by_count，其中keys方法取出所有关键码（文章名），sort根据后面块里提出的要求对表里的项排序，keys_by_count 前10项是计数值最大的项
- 11-12行取出 keys_by_count 的前 10 项输出

试验

输出示例（对一周约120万条记录，245M数据的试验）：

```
~/dev/bc/ 548> zcat ~/ongoing/logs/2006-12-17.log.gz | \
                time ruby code/report-counts.rb
4765: 2006/12/11/Mac-Crash
3138: 2006/01/31/Data-Protection
1865: 2006/12/10/EMail
1650: 2006/03/30/Teacup
1645: 2006/12/11/Java
1100: 2006/07/28/Open-Data
900: 2006/11/27/Choose-Relax
705: 2003/09/18/NXML
692: 2006/07/03/July-1-Fireworks
673: 2006/12/13/Blog-PR
        13.54 real         7.49 user         0.73 sys
```

- 最下面计时值：实际总时间 **13.54** 秒，其中用户时间 **7.49** 秒
- 看来程序有些慢。怎么办？需要分析

考虑优化

- 作者先用 **Perl** 做了同样试验，发现用时大约是 **Ruby** 的一半。**Perl** 程序没有 **Ruby** 漂亮，但实现做的时间比较长，程序执行速度快
- 程序处理分两个阶段：读入和检索计数，排序。最容易看到的是：这里只要找前**10**项，可能不需要排序，可能减少时间开销
- 用工具做程序执行性能剖析（**profiling**），得知第一阶段用时 **7.36** 秒，第二阶段用时 **0.07** 秒。显然改进第二阶段没有意义
- 改进第一阶段的效率，可能性很小，而且可能需要很大的代价。经过分析得到的结论：程序改进的可能性不大
- 不管怎么说，上面完成了一个有用的程序，其中做了一些与检索有关的事情，但没有写一行与检索有关的代码（检索是 **Ruby** 语言完成的）

下面考虑检索，考虑另一个问题：何人何时访问什么

问题：何人在何时访问什么？

- 另写一些脚本程序处理日志文件，统计出共计有 **12,600,064** 文件获取访问来自 **2,345,571** 个（**230**万）不同主机
- 许多人都可能关心“何人在何时访问什么”（审计员/警察/广告商等）

  下面考虑：给定主机名，找出它访问的所有文章和访问时间

  希望得到一个列表，如果表空，就是没访问
- 完全可以用前面方式写程序。但 **230** 万项的**hash**表可能太大（为保证效率，**hash** 表的负载因子应该比较小）。下面考虑另一方案
- 先简化数据减少数据量（数据预处理，只留下关心的信息）：

  crawl-66-249-72-77.googlebot.com 1166406026 2003/04/08/Riffs
  egspd42470.ask.com 1166406027 2006/05/03/MARS-T-Shirt
  84.7.249.205 1166406040 2003/03/27/Scanner

  第二项是 **Unix/Linux** 标准时间，从 **1970** 年 **1** 月 **1**日开始按秒记
- 下面写一个大程序将数据读入一个大型**hash**表

装填hash表

示例 4-5 读入数据建立大hash表

```
1 class BigHash
2
3   def initialize(file)
4     @hash = {}
5     lines = 0
6     File.open(file).each_line do |line|
7       s = line.split
8       article = s[2].intern
9       if @hash[s[0]]
10        @hash[s[0]] << [ s[1], article ]
11      else
12        @hash[s[0]] = [ s[1], article ]
13      end
14      lines += 1
15      STDERR.puts "Line: #{lines}" if (lines % 100000) == 0
16    end
17  end
18
19  def find(key)
20    @hash[key]
21  end
22
23 end
```

定义类**BigHash**

@hash是实例变量

函数**initialize**逐行读入**file**，一行行处理

s:=line.split把一行切分为字段存入数组**s**

s[0] 是主机**ip**或**url**地址，**s[1]**是时间，**s[2]**是访问的文件，**lines**记录读入行数

每读入**10**万行向屏幕输出一行信息，让用户知道程序正在正常工作

装填程序分析

- 装填**hash**表用了 **55** 分钟 **CPU** 时间，最后的表占用 **1.5G** 内存。可以算出每个主机信息用**680**字节，每个获取操作**126**字节

  试验：检索**2000**次平均用时**0.02**秒，说明**hash**表性能优越

  其中一半查询采用从日志中随机选出的主机名（检索一定成功），另一半用这种主机名的翻转（一定失败）【这样测试好不好？】

- 装载用**55**分钟有些长。一种改进是一次装载后把整个**hash**表卸载到文件，以后直接装入会快一些（注意：现在的问题是在一个大数据集合上做许多各种各样的查询，数据集可能频繁地多次使用）

- 上面程序很简单，在程序开发时间（**waiting-for-the-programmer**）和程序运行时间（ **waiting-for-the-program** ）方面都不错

  但作者觉得不太满意，应该有办法在获得同样查询性能的情况下改进装载时间和内存用量

  考虑用二分检索技术

二分检索

- 作者用**Ruby**开发的二分检索系统不令人满意：减少装载时间**10**分钟但多用**100M**内存，检索也慢了几倍。原因：
  - **Ruby**的**hash**表是语言内建的，采用高效的底层代码。自己做的程序在高层运行，通常会慢一些
  - 在**Ruby**里一切都是对象，对象会有些额外开销，包括空间开销和时间开销。运行慢一些是很可能的（常量因子）
- 下面考虑用**Java**：
  - **Java**里的整数就是整数（**Ruby**里的整数是对象）
  - **Java**数组里附加的设置也更少
- 【用**C**语言实现，可能有更大常量的性能提高，但编程工作量更大】

二分检索

示例 4-6 二分检索

```
1 package binary;
2
3  public class Finder {
4    public static int find(String[] keys, String target) {
5      int high = keys.length;
6      int low = -1;
7      while (high - low > 1) {
8        int probe = (low + high) >>> 1;
9        if (keys[probe].compareTo(target) > 0)
10         high = probe;
11       else
12         low = probe;
13     }
14     if (low == -1 || keys[low].compareTo(target) != 0)
15       return -1;
16     else
17       return low;
18   }
19 }
```

二分检索程序参考了瑞士苏黎世高工的**Gaston Gonnet**的代码（作者博客有讨论）

high和**low**初始设在范围外

程序里没判断是否找到，减少了循环中的判断

循环不变式：**low**或为**-1**或为某个不大于**target**的值，且**high**为第一个大于**target**的值

注意：**(low+high)/2**可能溢出。此时**Java**会报异常

有上述不变式，只需最后去检查**low**的值是否等于**target**。减少循环中操作提高了程序效率

相关实现可在**6**分半装载数据，内存用**1G**，检索也很快

有关二分检索的折中权衡

- 二分查找时间是**O(log N)**，**32**位机器里**N**不大于**32**，**64**位机器里**N**不大于**64**（处理一项的时间），对大多数应用已足够好

  二分查找的代码很短。简短的代码就是漂亮的代码，易理解，易做正确，不易隐藏问题，编译**/**缓存**/JIT**编译时更容易优化

  存储结构简单，建立好排序数组后不需要任何其他索引结构

- 缺点：数据需顺序排列，有些数据难以按顺序存储（少见）。需要在内存做，但现在内存不贵。使用外存的方案都复杂，效率很低

- 对经常更新巨大的数据集，还能用二分检索吗？

  操作系统是以页的方式分散管理计算机内存，自己可以用同样技术
  [这件事比较复杂，请自己想想，试试]

- 建议：

  1. 用系统提供的**hash**表或字典；
  2. 如果性能不满意，采用数组和二分检索；
  3. 如果都不行，再考虑其他解决方案

循环和效率

- 问题：为什么不在循环里检查是否找到目标，而是总让循环做到头（**high==low+1**），一些情况下会做无用功，为什么这样做？

  实际上这才是正确做法，可在数学里严格证明。下面给些直观解释。考虑循环进展的情况：

- **n**元数组（**n**很大），直接找到目标的机会依次为**1/n, 1/(n/2), ...**。只有数组小到一二十个元素时这个机会才变得较大，此时最多再做**3**、**4**次比较。如果目标不在数组里，根本没机会节约时间

  不在循环里检查是否找到元素，每次迭代都省时（循环可能做多次，对**32**位系统最多做**32**次）。在循环里检查目标，能减少几次循环的机会很小，整体看一定是得不偿失

- 正确的二分检索循环应该分两步走：
  - 先通过高效的循环定位
  - 循环结束后检查是否找到元素

大规模检索

- 今天说“检索”或“搜索”更多指网上的搜索。**google/yahoo!/**百度等是这方面的技术先锋（国内外都还有许多提供商）

  这一领域最有影响力的论文是**Cornell**的**Gerald Salton**在**1960**年代发表的论文，之后基础技术上并没有多大进步

- 大规模全文搜索的重要概念是**Posting**，一个**posting**是一个小记录，固定大小，记录的信息是：单词**x**出现在文档**y**的位置**z**。创建索引时读入所有文档并为每个词建一个**posting**；完成后排序把同一个词集中到一起。这时每个词对应一个**posting**列表，列表中每项是一个对，包含一个文档标识符**ID**和一个正文偏移量（词在文档里的位置）

  posting很小，大小固定，数量极多，采用二分检索很自然

  虽然不知道**google**或**yahoo!**确切怎么做，但它们很可能就是在用成千上万的机器在做二分检索

  前些年熟悉搜索技术的人们都嗤笑**goocle**的**20**亿个网页的上限。最近**google**宣布可以处理更多文档，可能就是换了**64**位的文档**ID**

结果评分和Web搜索

- 找到包含给定词的文档不难，下一步是处理AND或OR查询和词组查询。最难的是对结果排序，将最重要结果显示在最前面（ranking）

  计算机科学有个子领域Information Retrieval（IR，信息提取）研究这个问题。至今还没有得到什么很好的结果

  Google等公司已经为网络用户提供了海量数据检索，结果的效果还不错，能在列表前面提供较高质量的检索结果

- 提高检索质量的成就中，一项引人注目的工作是Google的PageRank算法，它主要基于网页的外来链接，认为链接多就是受欢迎

  PageRank算法适应的数据集的特点：大量文档，文档之间存在大量链接。符合这种要求的数据集只有两个：Web页，科学文献

- 大型搜索引擎在处理大量文档和大量用户方面的能力已给人深刻印象。这种搜索基于大规模并行处理：把大问题分解指派到大量小型计算机。这种方式很适合处理posting

Web搜索

- posting相互独立，基于posting的检索很容易划分，可提出很多想法，如基于单词的字母分区处理。要考虑如何组合来自不同检索的结果
- 这里的讨论回避了许多重要问题，包括如何防止网络上的恶意使用者利用网络搜索能力做坏事情
- 结论：

  It is hard to imagine any computer application that does not involve storing data and finding it based on its content. The world's single most popular computer application, web search, is a notable example.

  This chapter has considered some of the issues, notably bypassing the traditional "database" domain and the world of search strategies that involve external storage. Whether operating at the level of a single line of text or billions of web documents, search is central. From the programmer's point of view, it also needs to be said that implementing searches of one kind or another is, among other things, fun.

讨论