



# 这就是搜索引擎 核心技术详解

改变全世界人们生活方式的“信息之门”

GO

张俊林 著

# 前　　言

互联网产品形形色色，有产品导向的，有营销导向的，也有技术导向的，但是以技术见长的互联网产品比例相对小些。搜索引擎是目前互联网产品中最具技术含量的产品，如果不是唯一，至少也是其中之一。

经过十几年的发展，搜索引擎已经成为互联网的重要入口之一，Twitter 联合创始人埃文·威廉姆斯提出了“域名已死论”：好记的域名不再重要，因为人们会通过搜索进入网站。搜索引擎排名对于中小网站流量来说至关重要。了解搜索引擎简单界面背后的技术原理其实对很多人都很重要。

## 为什么会有这本书

最初写本搜索引擎技术书籍的想法萌生于两年前，当时的场景是要给团队成员做搜索技术培训，但是我找遍了相关图书，却没有发现非常合适的搜索技术入门书籍。当时市面上的书籍，要么是信息检索理论方面的专著，理论性太强不易懂，而且真正讲搜索引擎技术的章节并不多；要么是 Lucene 代码分析这种过于实务的书籍，像搜索引擎这种充满算法的应用，直接分析开源系统代码并不是非常高效的学习方式。所以当时萌生了写一本既通俗易懂，适合没有相关技术背景的人员阅读，又比较全面，且融入最新技术的搜索引擎书籍，但是真正动手开始写是一年前的事情了。

写书前我给自己定了几个目标。首先内容要全面，即全面覆盖搜索引擎相关技术的主要方面，不仅要包含倒排索引、检索模型和爬虫等常见内容，也要详细讲解链接分析、网页反作弊、用户搜索意图分析、云存储及网页去重，甚至是搜索引擎缓存等内容，这些都是一个完整搜索引擎的有机组成部分，但是详述其原理的书籍并不多，我希望能够尽可能全面些。

第二个目标是通俗易懂。我希望没有任何相关技术背景的人也能够通过阅读这本书有所收获，最好是不懂技术的同学也能大致看懂。这个目标看似简单，其实很不容易达到，我也不敢说这本书已经达到了此目的，但是确实已经尽自己所能去做了。至于具体的措施，则包含以下三个方面。

- 一个是尽可能减少数学公式的出现次数，除非不得已不罗列公式。虽说数学公式具简洁之美，但是大多数人其实对于数学符号是有恐惧和逃避心理的，多年前我也有类似心理，所以但凡可能，尽量不用数学公式。
- 一个是尽可能多举例子，尤其是一些比较难理解的地方，需要例子来增进理解。
- 还有一个是多画图。就我个人的经验来说，尽管算法或者技术是很抽象的，但是如果深入理解其原理，去繁就简，那么一定可以把算法转换成形象的图片。如果不能在头脑中形成算法直观的图形表示，说明并未透彻了解其原理。这是我判断自己是否深入理解算法的一个私有标准。鉴于此，本书中在讲解算法的地方，大量采用了算法原理图，全书包含了超过 300 幅算法原理讲解图，相信这对于读者深入理解算法会有很大的帮助。

第三个目标是强调新现象新技术，比如 Google 的咖啡因系统及 Megastore 等云存储系统、Pregel 云图计算模型、暗网爬取技术、Web 2.0 网页作弊、机器学习排序、情境搜索、社会化搜索等在相关章节都有讲解。

第四个目标是强调原理，不纠缠技术细节。对于新手一个易犯的毛病是喜欢抠细节，只见树木不见森林，搞明白了一个公式却不了解其背后的基本思想和出发点。我接触的技术人员很多，十有七八会有这个特点。这里有个“道术孰优”的问题，何为“道”？何为“术”？举个例子的话，《孙子兵法》是道，而《三十六计》则为术。“道”所述，是宏观的、原理性的、长久不变的基本原理，而“术”则是在遵循基本原理基础上的具体手段和措施，具有易变性。技术也是如此，算法本身的细节是“术”，算法体现的基本思想则是“道”，知“道”而学“术”，两者虽不可偏废，但是若要选择优先级的话，无疑我会选择先“道”后“术”。

以上四点是写书前定下的目标，现在书写完了，也许很多地方不能达到最初的期望，但是尽了力就好。写书的过程很辛苦，起码比我原先想象的要辛苦，因为工作繁忙，所以只能每天早早起床，再加上周末及节假日的时间来完成。也许书中还存在这样那样的缺点，但是我可以无愧地说写这本书是有诚意的。

## 这本书是写给谁的

如果您是下列人员之一，那么本书就是写给您的。

## 1. 对搜索引擎核心算法有兴趣的技术人员

- 搜索引擎的整体框架是怎样的？包含哪些核心技术？
- 网络爬虫的基本架构是什么？常见的爬取策略是什么？什么是暗网爬取？如何构建分布式爬虫？百度的阿拉丁计划是什么？
- 什么是倒排索引？如何对倒排索引进行数据压缩？
- 搜索引擎如何对搜索结果排序？
- 什么是向量空间模型？什么是概率模型？什么是 BM25 模型？什么是机器学习排序？它们之间有何异同？
- PageRank 和 HITS 算法是什么关系？有何异同？SALSA 算法是什么？Hilltop 算法又是什么？各种链接分析算法之间是什么关系？
- 如何识别搜索用户的真实搜索意图？用户搜索目的可以分为几类？什么是点击图？什么是查询会话？相关搜索是如何做到的？
- 为什么要对网页进行去重处理？如何对网页进行去重？哪种算法效果较好？
- 搜索引擎缓存有几级结构？核心策略是什么？
- 什么是情境搜索？什么是社会化搜索？什么是实时搜索？
- 搜索引擎有哪些发展趋势？

如果您对其中三个以上的问题感兴趣，那么这本书就是为您而写的。

## 2. 对云计算与云存储有兴趣的技术人员

- 什么是 CAP 原理？什么是 ACID 原理？它们之间有什么异同？
- Google 的整套云计算框架包含哪些技术？Hadoop 系列和 Google 的云计算框架是什么关系？
- Google 的三驾马车 GFS、BigTable、MapReduce 各自代表什么含义？是什么关系？
- Google 的咖啡因系统的基本原理是什么？
- Google 的 Pregel 计算模型和 MapReduce 计算模型有什么区别？
- Google 的 Megastore 云存储系统和 BigTable 是什么关系？
- 亚马逊公司的 Dynamo 系统是什么？

- 雅虎公司的 PNUTS 系统是什么？
- Facebook 公司的 Haystack 存储系统适合应用在什么场合？

如果您对上述问题感兴趣，相信可以从书中找到答案。

### 3. 从事搜索引擎优化的网络营销人员及中小网站站长

- 搜索引擎的反作弊策略是怎样的？如何进行优化避免被认为是作弊？
- 搜索引擎如何对搜索结果排序？链接分析和内容排序是什么关系？
- 什么是内容农场？什么是链接农场？它们是什么关系？
- 什么是 Web 2.0 作弊？有哪些常见手法？
- 什么是 SpamRank？什么是 TrustRank？什么又是 BadRank？它们是什么关系？
- 咖啡因系统对网页排名有何影响？

最近有一批电子商务网站针对搜索引擎优化，结果被 Google 认为是黑帽 SEO 而导致搜索排名降权，如何避免这种情况？从事相关行业的营销人员和网站站长应该深入了解搜索引擎反作弊的基本策略和方法，甚至是网页排名算法等搜索引擎核心技术。SEO 技术说到底其实很简单，虽然不断发生变化，但是很多原理性的策略总是相似的，万变不离其宗，深入了解搜索引擎相关技术原理将形成您的行业竞争优势。

### 4. 作者自己

我的记性不太好，往往一段时间内了解的技术，时隔几年后就很模糊了，所以这本书也是为我自己写的，以作为技术备查手册。沈利也参与了本书的部分编写工作。

### 致谢

感谢博文视点的付睿编辑，没有她也就没有本书的面世，付编辑在阅稿过程中提出的细致入微的改进点对我帮助甚大。

感谢翻开此书的读者，如果您在阅读本书的过程中发现一些纰漏或者错误，或者是意见建议，希望您能够不吝让我知晓，我会守在 mailjunlin@gmail.com 这个信箱旁敬候您的来信，如果给我微博发信也非常欢迎 <http://www.weibo.com/malefactor>。

特别感谢我的妻子，在近一年的写作过程中，我几乎把能用的所有业余时间都投入在本书

的写作上，她为了不让我分心，承担了所有的家务，不介意没有时间陪她，这本书的诞生且算是送她的一个礼物吧。

于我而言，这本书的写作是一个辛苦而欣喜的过程，有如旅人远行，涉水跋山之际抬头远眺，总能看到曾经忽略的旖旎丽景，若您在阅读本书的过程中也能有此体会，那就是我的荣幸了。

张俊林

2011年6月



# 目 录

<b>第1章 搜索引擎及其技术架构</b> .....	1
1.1 搜索引擎为何重要 .....	1
1.1.1 互联网的发展 .....	1
1.1.2 商业搜索引擎公司的发展.....	3
1.1.3 搜索引擎的重要地位.....	3
1.2 搜索引擎技术发展史 .....	4
1.2.1 史前时代：分类目录的一代.....	4
1.2.2 第一代：文本检索的一代.....	5
1.2.3 第二代：链接分析的一代.....	5
1.2.4 第三代：用户中心的一代.....	5
1.3 搜索引擎的3个目标.....	6
1.4 搜索引擎的3个核心问题.....	7
1.4.1 3个核心问题 .....	7
1.4.2 与技术发展的关系 .....	8
1.5 搜索引擎的技术架构 .....	9
<b>第2章 网络爬虫</b> .....	12
2.1 通用爬虫框架 .....	12
2.2 优秀爬虫的特性 .....	15
2.3 爬虫质量的评价标准 .....	18
2.4 抓取策略.....	19
2.4.1 宽度优先遍历策略（Breath First） .....	20
2.4.2 非完全PageRank策略（Partial PageRank） .....	21
2.4.3 OCIP策略（Online Page Importance Computation） .....	23
2.4.4 大站优先策略（Larger Sites First） .....	23
2.5 网页更新策略 .....	23
2.5.1 历史参考策略 .....	24
2.5.2 用户体验策略 .....	24

2.5.3 聚类抽样策略 .....	24
2.6 暗网抓取（Deep Web Crawling） .....	26
2.6.1 查询组合问题 .....	27
2.6.2 文本框填写问题 .....	29
2.7 分布式爬虫 .....	30
2.7.1 主从式分布爬虫（Master-Slave） .....	31
2.7.2 对等式分布爬虫（Peer to Peer） .....	31
本章提要 .....	34
本章参考文献 .....	34
<b>第3章 搜索引擎索引 .....</b>	<b>36</b>
3.1 索引基础 .....	36
3.1.1 单词—文档矩阵 .....	37
3.1.2 倒排索引基本概念 .....	37
3.1.3 倒排索引简单实例 .....	39
3.2 单词词典 .....	42
3.2.1 哈希加链表 .....	42
3.2.2 树形结构 .....	43
3.3 倒排列表（Posting List） .....	44
3.4 建立索引 .....	45
3.4.1 两遍文档遍历法（2-Pass In-Memory Inversion） .....	45
3.4.2 排序法（Sort-based Inversion） .....	46
3.4.3 归并法（Merge-based Inversion） .....	49
3.5 动态索引 .....	50
3.6 索引更新策略 .....	51
3.6.1 完全重建策略（Complete Re-Build） .....	51
3.6.2 再合并策略（Re-Merge） .....	52
3.6.3 原地更新策略（In-Place） .....	55
3.6.4 混合策略（Hybrid） .....	57
3.7 查询处理 .....	57
3.7.1 一次一文档（Doc at a Time） .....	58
3.7.2 一次一单词（Term at a Time） .....	59
3.7.3 跳跃指针（Skip Pointers） .....	60
3.8 多字段索引 .....	62
3.8.1 多索引方式 .....	62
3.8.2 倒排列表方式 .....	63

3.8.3 扩展列表方式 (Extent List) .....	64
3.9 短语查询.....	64
3.9.1 位置信息索引 (Position Index) .....	65
3.9.2 双词索引 (Nextword Index) .....	66
3.9.3 短语索引 (Phrase Index) .....	67
3.9.4 混合方法.....	67
3.10 分布式索引 (Parallel Indexing) .....	68
3.10.1 按文档划分 (Document Partitioning) .....	69
3.10.2 按单词划分 (Term Partitioning) .....	70
3.10.3 两种方案的比较 .....	72
本章提要.....	73
本章参考文献.....	73
<b>第 4 章 索引压缩.....</b>	<b>76</b>
4.1 词典压缩.....	76
4.2 倒排列表压缩算法 .....	78
4.2.1 评价索引压缩算法的指标.....	79
4.2.2 一元编码与二进制编码.....	79
4.2.3 Elias Gamma 算法与 Elias Delta 算法 .....	81
4.2.4 Golomb 算法与 Rice 算法 .....	81
4.2.5 变长字节算法 (Variable Byte) .....	83
4.2.6 SimpleX 系列算法 .....	84
4.2.7 PForDelta 算法.....	86
4.3 文档编号重排序 (DocID Reordering) .....	89
4.4 静态索引裁剪 (Static Index Pruning) .....	93
4.4.1 以单词为中心的索引裁剪.....	94
4.4.2 以文档为中心的索引裁剪 .....	96
本章提要.....	97
本章参考文献.....	97
<b>第 5 章 检索模型与搜索排序 .....</b>	<b>99</b>
5.1 布尔模型 (Boolean Model) .....	101
5.2 向量空间模型 (Vector Space Model) .....	102
5.2.1 文档表示.....	102
5.2.2 相似性计算.....	104
5.2.3 特征权重计算 .....	106

5.3 概率检索模型 .....	108
5.3.1 概率排序原理 .....	108
5.3.2 二元独立模型（Binary Independent Model） .....	110
5.3.3 BM25 模型 .....	113
5.3.4 BM25F 模型 .....	115
5.4 语言模型方法 .....	116
5.5 机器学习排序（Learning to Rank） .....	119
5.5.1 机器学习排序的基本思路 .....	120
5.5.2 单文档方法（PointWise Approach） .....	121
5.5.3 文档对方法（PairWise Approach） .....	122
5.5.4 文档列表方法（ListWise Approach） .....	123
5.6 检索质量评价标准 .....	125
5.6.1 精确率与召回率 .....	126
5.6.2 P@10 指标 .....	127
5.6.3 MAP 指标（Mean Average Precision） .....	128
本章提要 .....	129
本章参考文献 .....	129
<b>第6章 链接分析 .....</b>	<b>131</b>
6.1 Web 图 .....	131
6.2 两个概念模型及算法之间的关系 .....	133
6.2.1 随机游走模型（Random Surfer Model） .....	133
6.2.2 子集传播模型 .....	135
6.2.3 链接分析算法之间的关系 .....	136
6.3 PageRank 算法 .....	137
6.3.1 从入链数量到 PageRank .....	137
6.3.2 PageRank 计算 .....	138
6.3.3 链接陷阱（Link Sink）与远程跳转（Teleporting） .....	139
6.4 HITS 算法（Hypertext Induced Topic Selection） .....	140
6.4.1 Hub 页面与 Authority 页面 .....	140
6.4.2 相互增强关系 .....	141
6.4.3 HITS 算法 .....	142
6.4.4 HITS 算法存在的问题 .....	144
6.4.5 HITS 算法与 PageRank 算法比较 .....	145
6.5 SALSA 算法 .....	146
6.5.1 确定计算对象集合 .....	146

6.5.2 链接关系传播 .....	148
6.5.3 Authority 权值计算.....	150
6.6 主题敏感 PageRank (Topic Sensitive PageRank) .....	152
6.6.1 主题敏感 PageRank 与 PageRank 的差异.....	152
6.6.2 主题敏感 PageRank 计算流程.....	153
6.6.3 利用主题敏感 PageRank 构造个性化搜索.....	156
6.7 Hilltop 算法 .....	156
6.7.1 Hilltop 算法的一些基本定义.....	157
6.7.2 Hilltop 算法 .....	158
6.8 其他改进算法 .....	162
6.8.1 智能游走模型 (Intelligent Surfer Model) .....	162
6.8.2 偏置游走模型 (Biased Surfer Model) .....	163
6.8.3 PHITS 算法 (Probability Analogy of HITS) .....	163
6.8.4 BFS 算法 (Backward Forward Step) .....	163
本章提要.....	164
本章参考文献.....	164
<b>第 7 章 云存储与云计算 .....</b>	<b>166</b>
7.1 云存储与云计算概述 .....	167
7.1.1 基本假设.....	167
7.1.2 理论基础.....	168
7.1.3 数据模型.....	170
7.1.4 基本问题.....	170
7.1.5 Google 的云存储与云计算架构.....	171
7.2 Google 文件系统 (GFS) .....	173
7.2.1 GFS 设计原则.....	174
7.2.2 GFS 整体架构.....	174
7.2.3 GFS 主控服务器.....	176
7.2.4 系统交互行为 .....	178
7.3 Chubby 锁服务.....	179
7.4 BigTable .....	181
7.4.1 BigTable 的数据模型.....	181
7.4.2 BigTable 整体结构.....	183
7.4.3 BigTable 的管理数据.....	184
7.4.4 主控服务器 (Master Server) .....	186
7.4.5 子表服务器 (Tablet Server) .....	187

7.5 Megastore 系统 .....	191
7.5.1 实体群组切分 .....	192
7.5.2 数据模型 .....	193
7.5.3 数据读/写与备份 .....	195
7.6 Map/Reduce 云计算模型 .....	195
7.6.1 计算模型 .....	196
7.6.2 整体逻辑流程 .....	197
7.6.3 应用示例 .....	198
7.7 咖啡因系统——Percolator .....	199
7.7.1 事务支持 .....	200
7.7.2 观察/通知体系结构 .....	202
7.8 Pregel 图计算模型 .....	203
7.9 Dynomoo 云存储系统 .....	206
7.9.1 数据划分算法（Partitioning Algorithm） .....	207
7.9.2 数据备份（Replication） .....	208
7.9.3 数据读/写 .....	208
7.9.4 数据版本控制 .....	209
7.10 PNUTS 云存储系统 .....	210
7.10.1 PNUTS 整体架构 .....	211
7.10.2 存储单元 .....	211
7.10.3 子表控制器与数据路由器 .....	213
7.10.4 雅虎消息代理 .....	213
7.10.5 数据一致性 .....	214
7.11 HayStack 存储系统 .....	215
7.11.1 HayStack 整体架构 .....	216
7.11.2 目录服务 .....	218
7.11.3 HayStack 缓存 .....	219
7.11.4 HayStack 存储系统 .....	219
本章提要 .....	222
本章参考文献 .....	222
<b>第8章 网页反作弊 .....</b>	<b>224</b>
8.1 内容作弊 .....	224
8.1.1 常见内容作弊手段 .....	225
8.1.2 内容农场（Content Farm） .....	226
8.2 链接作弊 .....	227

8.3	页面隐藏作弊 .....	230
8.4	Web 2.0 作弊方法 .....	231
8.5	反作弊技术的整体思路 .....	232
8.5.1	信任传播模型 .....	233
8.5.2	不信任传播模型 .....	234
8.5.3	异常发现模型 .....	234
8.6	通用链接反作弊方法 .....	236
8.6.1	TrustRank 算法 .....	237
8.6.2	BadRank 算法 .....	238
8.6.3	SpamRank .....	239
8.7	专用链接反作弊技术 .....	240
8.7.1	识别链接农场 .....	240
8.7.2	识别 Google 轰炸 .....	241
8.8	识别内容作弊 .....	241
8.9	反隐藏作弊 .....	241
8.9.1	识别页面隐藏 .....	241
8.9.2	识别网页重定向 .....	242
8.10	搜索引擎反作弊综合框架 .....	242
	本章提要 .....	244
	本章参考文献 .....	244
<b>第 9 章</b>	<b>用户查询意图分析 .....</b>	<b>246</b>
9.1	搜索行为及其意图 .....	246
9.1.1	用户搜索行为 .....	246
9.1.2	用户搜索意图分类 .....	248
9.2	搜索日志挖掘 .....	250
9.2.1	查询会话 (Query Session) .....	250
9.2.2	点击图 (Click Graph) .....	251
9.2.3	查询图 (Query Graph) .....	252
9.3	相关搜索 .....	253
9.3.1	基于查询会话的方法 .....	253
9.3.2	基于点击图的方法 .....	254
9.4	查询纠错 .....	255
9.4.1	编辑距离 (Edit Distance) .....	256
9.4.2	噪声信道模型 (Noise Channel Model) .....	257
	本章提要 .....	257

本章参考文献.....	258
<b>第 10 章 网页去重 .....</b>	<b>259</b>
10.1 通用去重算法框架.....	261
10.2 Shingling 算法.....	262
10.3 I-Match 算法.....	265
10.4 SimHash 算法.....	268
10.4.1 文档指纹计算.....	269
10.4.2 相似文档查找.....	270
10.5 SpotSig 算法.....	272
10.5.1 特征抽取.....	272
10.5.2 相似文档查找.....	273
本章提要.....	274
本章参考文献.....	274
<b>第 11 章 搜索引擎缓存机制 .....</b>	<b>276</b>
11.1 搜索引擎缓存系统架构.....	277
11.2 缓存对象.....	279
11.3 缓存结构.....	281
11.4 缓存淘汰策略（Evict Policy） .....	283
11.4.1 动态策略.....	284
11.4.2 混合策略.....	284
11.5 缓存更新策略（Refresh Policy） .....	285
本章提要.....	286
本章参考文献.....	287
<b>第 12 章 搜索引擎发展趋势 .....</b>	<b>288</b>
12.1 个性化搜索 .....	288
12.2 社会化搜索 .....	290
12.3 实时搜索.....	291
12.4 移动搜索.....	293
12.5 地理位置感知搜索 .....	294
12.6 跨语言搜索 .....	296
12.7 多媒体搜索 .....	298
12.8 情境搜索.....	299

# 第1章 搜索引擎及其技术架构

“天地玄黄 宇宙洪荒  
日月盈昃 辰宿列张  
寒来暑往 秋收冬藏  
闰馀成岁 律吕调阳  
云腾致雨 露结为霜  
金生丽水 玉出昆冈”

《千字经》

搜索引擎已经发展为每个人上网都离不开的重要工具，但是为何搜索引擎有着如此重要的地位？其技术发展历程是怎样的？其基本目标是什么？核心问题又是什么？基本技术架构如何？本章内容即给出上述问题的答案，以使读者对搜索引擎有个宏观的理解。

## 1.1 搜索引擎为何重要

搜索引擎依托于互联网，互联网的蓬勃发展是搜索引擎产品与技术逐步成熟的大背景。离开互联网，搜索引擎将无从谈起。

### 1.1.1 互联网的发展

20世纪90年代初期是互联网后期获得大规模发展的起爆点，之所以如此，是有其技术背景和社会背景的。

1991年，Tim Berners-Lee 将超文本的概念引入互联网，同时推出了 WWW 雏形、配套的 HTTP 传输协议及相应的 Web 服务器技术。1993年，第一个图形浏览器 mosaic 诞生，网页浏览客户端趋于成熟。这些技术与产品为互联网的快速普及和发展做好了技术准备，互联网用户开始从最初的军队和高校等科研机构普及到普通个人用户，为接下来互联网的商业化大规模发

展奠定了技术基础。

1992 年美国副总统戈尔提出“信息高速公路计划”原型提案，1993 年克林顿总统全力推动该计划的实施，以作为振兴美国经济的重要增长点，由此爆发了互联网超常规发展的黄金十年，尽管 2002 年后互联网泡沫破裂，但是该计划催生了一大批基于软件和互联网的高科技公司。

1993 年起，全球互联网进入高速发展期，1994 年全球主机数量首次超过 300 万台，1995 年超过 600 万台，之后以更快的加速度进入长期快速成长通道，如图 1-1 所示。互联网用户也是如此，图 1-2 显示了互联网用户的快速增长趋势。

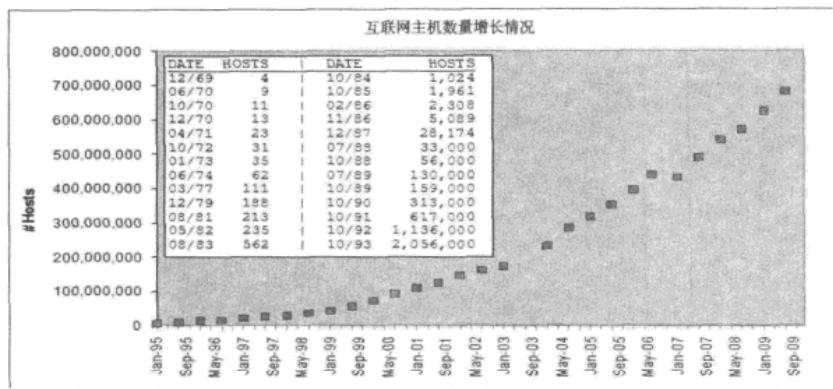


图 1-1 互联网主机数量增长情况 (1995—2009)

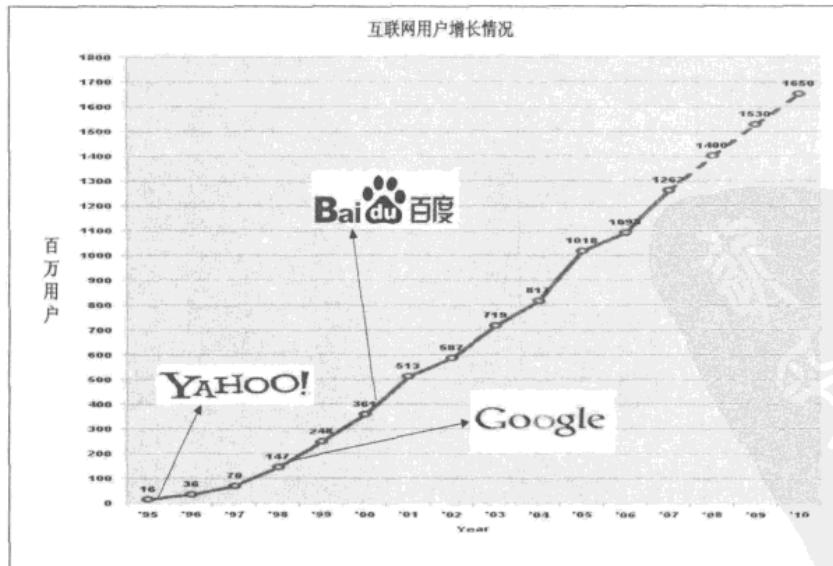


图 1-2 互联网用户增长情况 (1995—2010)



### 1.1.2 商业搜索引擎的发展

搜索引擎的产生和发展，与互联网的蓬勃发展这个大背景是密不可分的。在互联网发展初期，一般互联网用户最常用的应用仅是电子邮箱，而随着 WWW 相关协议和产品的逐步成熟，网站拥有者制作和发布网页信息的成本急剧下降。而 PC 电脑及图形化浏览器的普及，使得普通用户浏览信息成本急剧降低。再加上网络基础设施的大量投入，使得带宽等不断加大。这几个因素交互作用，导致互联网上的信息产生爆炸性增长。在信息量快速增长的情况下，如何能够找到满足用户需求的网页内容就日益成为越来越重要的问题。信息增长速度越快，用户需求越迫切。大的搜索引擎公司就是在这个用户需求背景下，从建立到逐步壮大，乃至发展到今天搜索引擎成为最重要的互联网的应用。

1995 年是搜索引擎商业公司发展的重要起点，其对应的背景是：互联网上的 Web 站点数量首次超过 100 万，此时普通用户已经无法依赖手工浏览的方式来获得自己想要的信息。在这一年产生了很多风云一时的早期搜索引擎公司。Yahoo、InfoSeek、Fast Search、AltaVista、Excite 等曾经非常著名的搜索引擎公司都创建于 1995 年。

Yahoo 依靠人工编辑导航目录，将互联网上重要的站点分门别类整理好，满足了人们查找重要网站的需求，可谓应时而生，从此快速成长为最著名的搜索和门户网站。其他搜索引擎公司则提供基于传统信息检索系统的搜索服务，也都获得了快速成长。

随着互联网的进一步快速发展，信息的爆炸性增长，已有的搜索引擎服务提供商所提供的搜索服务质量并无大的改善，逐渐不能满足用户的需求。Google 于 1998 年成立，以 PageRank 链接分析等新技术大幅度提高了搜索质量，之后高速发展并抢占了绝大多数搜索引擎市场，成长为目前最重要的互联网公司之一。百度则依靠本地化优势，成为中国国内最强势的搜索引擎服务提供商。尽管 2000 年后陆续有大小公司进入搜索市场，但是无论从技术角度还是市场份额角度，搜索市场格局并未发生太大变化。

### 1.1.3 搜索引擎的重要地位

搜索引擎已成为互联网最重要的应用之一，这一点毫无疑问，这也是为何国内各大互联网公司也希望切入搜索市场的主要原因。那么，为何搜索引擎如此重要？

正像前文所述，互联网信息量在过去 15 年获得了爆炸性增长，信息过载的问题就目前来说非常严重，随着互联网个性化的发展趋势逐步展现，普通用户发布信息的成本越来越低，这个问题将会更加严重。这是搜索引擎相关应用越来越重要的一个基础背景。搜索是目前解决信息过载的相对有效方式，在没有更有效的替代解决方式出来之前，搜索引擎作为互联网网站和应用的入口及处于行业制高点的重要地位只会逐步加强。

尽管不论国际还是国内出现了一种新的现象，即成功的新互联网公司屏蔽搜索引擎公司爬虫，比如 Facebook 对 Google 的屏蔽，国内电子商务站点淘宝对百度的屏蔽。但是这种现象仅仅是商业公司之间的竞争策略，也可看做是垂直搜索和通用搜索的竞争，但并非搜索应用与非搜索应用的竞争。即便是 Facebook 和淘宝，面对自己用户产生的海量数据，依然要依靠搜索来为用户提供满意的服务，区别仅仅在于是自己来提供还是第三方公司来提供。所以这种现象并不能作为搜索引擎服务式微的证据。

## 1.2 搜索引擎技术发展史

从搜索引擎所采取的技术来说，可以将搜索引擎技术的发展划分为 4 个时代如图 1-3 所示。

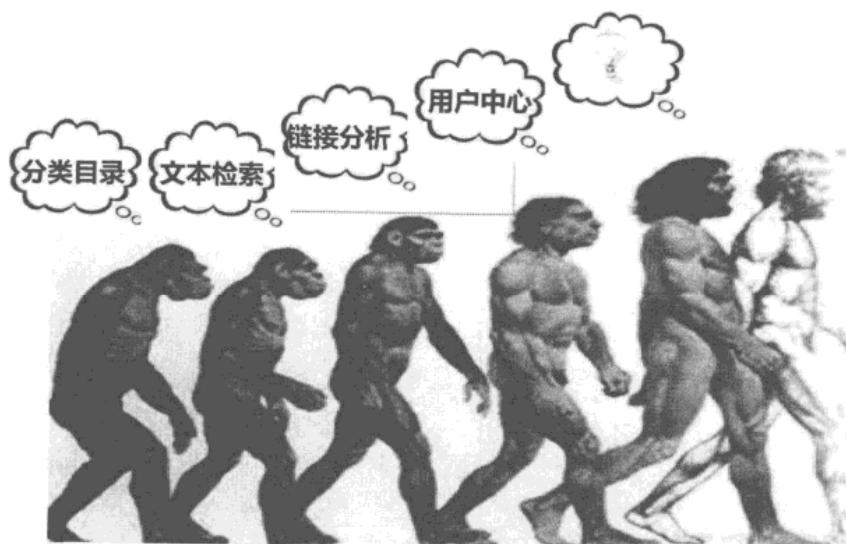


图 1-3 搜索引擎技术发展史

### 1.2.1 史前时代：分类目录的一代

这个时代也可以称为“导航时代”，Yahoo 和国内 hao123 是这个时代的代表。通过人工收集整理，把属于各个类别的高质量网站或者网页分门别类罗列，用户可以根据分级目录来查找高质量的网站。这种方式是纯人工的方式，并未采取什么高深的技术手段。

采取分类目录的方式，一般被收录的网站质量都较高，但是这种方式可扩展性不强，绝大部分网站不能被收录。



### 1.2.2 第一代：文本检索的一代

文本检索的一代采用经典的信息检索模型，比如布尔模型、向量空间模型或者概率模型，来计算用户查询关键词和网页文本内容的相关程度。网页之间有丰富的链接关系，而这一代搜索引擎并未使用这些信息。早期的很多搜索引擎比如 AltaVista、Excite 等大都采取这种模式。

相比分类目录，这种方式可以收录大部分网页，并能够按照网页内容和用户查询的匹配程度进行排序。但是总体而言，搜索结果质量不是很好。

### 1.2.3 第二代：链接分析的一代

这一代的搜索引擎充分利用了网页之间的链接关系，并深入挖掘和利用了网页链接所代表的含义。通常而言，网页链接代表了一种推荐关系，所以通过链接分析可以在海量内容中找出重要的网页。这种重要性本质上是对网页流行程度的一种衡量，因为被推荐次数多的网页其实代表了其具有流行性。搜索引擎通过结合网页流行性和内容相似性来改善搜索质量。

Google 率先提出并使用 PageRank 链接分析技术，并大获成功，这同时引起了学术界和其他商业搜索引擎的关注。后来学术界陆续提出了很多改进的链接分析算法。目前几乎所有的商业搜索引擎都采取了链接分析技术。

采用链接分析能够有效改善搜索结果质量，但是这种搜索引擎并未考虑用户的个性化要求，所以只要输入的查询请求相同，所有用户都会获得相同的搜索结果。另外，很多网站拥有者为了获得更高的搜索排名，针对链接分析算法提出了不少链接作弊方案，这样导致搜索结果质量变差。

### 1.2.4 第三代：用户中心的一代

目前的搜索引擎大都可以归入第三代，即以理解用户需求为核心。不同用户即使输入同一个查询关键词，但其目的也有可能不一样。比如同样输入“苹果”作为查询词，一个追捧 iPhone 的时尚青年和一个果农的目的会有相当大的差距。即使是同一个用户，输入相同的查询词，也会因为所在的时间和场合不同，需求有所变化。而目前搜索引擎大都致力于解决如下问题：如何能够理解用户发出的某个很短小的查询词背后包含的真正需求，所以这一代搜索引擎称之为以用户为中心的一代。

为了能够获取用户的真实需求，目前搜索引擎大都做了很多技术方面的尝试。比如利用用户发送查询词时的时间和地理位置信息，利用用户过去发出的查询词及相应的点击记录等历史信息等技术手段，来试图理解用户此时此地的真正需求。

### 1.3 搜索引擎的 3 个目标

搜索引擎可以说是目前所有互联网应用里技术含量最高的一种，尽管其应用形式非常简单：用户输入查询词，搜索引擎返回搜索结果。但是要为以亿计数的互联网用户提供准确快速的搜索结果，里面包含了很多技术手段。总的来说，搜索引擎技术所希望达到的目标可以归纳为：更全、更快、更准，如图 1-4 所示。

所谓“更全”，是从其索引的网页数量而言的，目前任意一个商业搜索引擎索引网页的覆盖范围都只占了互联网页面的一部分，可以通过提高网络爬虫相关技术来达到此目标。

“更快”这个目标则贯穿于搜索引擎的大多数技术方向，比如索引相关技术、缓存等技术的提出都是直接为了达到此目的。而其他很多技术也间接为此服务，即使是分布式海量云存储平台，也是为了能够处理海量的网页数据，以达到对“更全”和“更快”这两个目标的响应和支持。

搜索引擎目标			
	更全	更快	更准
索引		✓	
索引压缩		✓	
排序			✓
链接分析			✓
反作弊			✓
用户研究			✓
云存储	✓	✓	
爬虫	✓	✓	
网页去重		✓	✓
缓存		✓	

图 1-4 搜索引擎 3 个目标

在这 3 个目标中，如何使得搜索结果“更准”是最为关键的目标。无论是排序技术也好，还是链接分析技术也好，抑或是用户研究等技术，最终都是为了使搜索结果更加准确，以此增强用户体验。对于一个搜索引擎来说，达到“更全”与“更快”可以使它不落后于同类产品，但是如果能够做到“更准”，则能够构建核心竞争能力。



## 1.4 搜索引擎的3个核心问题

如上所述，搜索引擎如何能够搜得更准是其最重要的目标，那么如何才能使得搜索结果更准确？这里面涉及了3个核心问题。

### 1.4.1 3个核心问题

#### 1 用户真正的需求是什么

搜索引擎用户输入的查询请求非常简短，查询的平均长度是2.7个单词。如何从如此短的查询请求里获知隐藏其后的真实用户需求？这是搜索引擎首先需要解决的非常重要的问题。如果不能获取用户真正的搜索意图，搜索的准确性无从谈起，即使后续内容匹配算法再精巧也无济于事。

从另外一个角度看，即使是同一个查询词，不同用户的搜索目的是不同的，如何识别这种差异？如果更进一步，即使是同一个用户发出的同一个查询词，也可能因为用户所处场景不同，其目的存在差异，又如何识别？所有这些都是搜索引擎需要解决的核心问题，即用户在此时此地发出某个查询，他的真实搜索意图到底是什么。

#### 2 哪些信息是和用户需求真正相关的

上述第一个核心问题是从业务需求角度出发的，另外两个核心问题则是从数据角度考虑的。搜索引擎本质上是一个匹配过程，即从海量数据里面找到能够匹配用户需求的内容。所以，在明确用户真实意图这个前提条件做到后，如何找到能够满足用户需求的信息则成为关键因素。

判断内容和用户查询关键词的相关性，一直是信息检索领域的核心研究课题，不断提出的信息检索模型即在试图解决这个问题。相关研究历时近60年，尽管不断有新方法提出，检索效果总体而言也在逐步改进，但是这个领域的基本指导思想还是基于关键词的匹配，包括现在所有搜索引擎的相关性计算部分，其基本计算思路和几十年前相比并无本质差异。

如何能够在这个核心问题上有所突破？这个问题将会越来越重要，而从关键词匹配到让机器真正理解信息所代表的含义是解决这个问题必须迈过的门槛。从目前来看，尽管包括人工智能在内的很多相关研究领域对此有所进展，但是短期内还未能看到解决这一问题的清晰技术思路。

#### 3 哪些信息是用户可以信赖的

搜索本质上是找到能够满足用户需求的信息，尽管相关性是衡量信息是否满足用户需求的

一个重要方面，但并非全部。信息是否值得信赖是另外一个重要的衡量标准。

搜索引擎需要处理的信息对象是互联网上任意用户发布的内容，但是内容发布者所发布内容是否可信并无明确判断标准。这其间存在恶意的信息发布者故意歪曲事实的情况，也有信息发布者无心的错误。在同一个查询的搜索结果内，完全可能存在相互矛盾的搜索答案，此时信息的可信性即成为突出问题。

比如用户想到某一餐馆就餐，在做出消费决定前，在网上搜索曾在此餐馆就餐的用户的过往评论，以此辅助决策。而搜索到的相关内容，完全有可能是餐馆故意发布的一些好评信息，以此误导消费者。但是如果信息发布者是该用户的朋友，那么信息的可信性就会大大增加。

从某种角度看，链接分析之所以能够改善搜索结果，可以认为是对信息的可信度做出的评判。即将网页的重要性作为是否可信赖的一个判断标准，返回重要网页即是返回可信赖网页。

#### 1.4.2 与技术发展的关系

1.3 节对搜索引擎技术发展史做了归纳整理，对照本节提到的搜索引擎 3 个核心问题，可以看到不同历史发展阶段技术的不同侧重点，以及搜索引擎技术螺旋式上升的发展趋势，如图 1-5 所示。

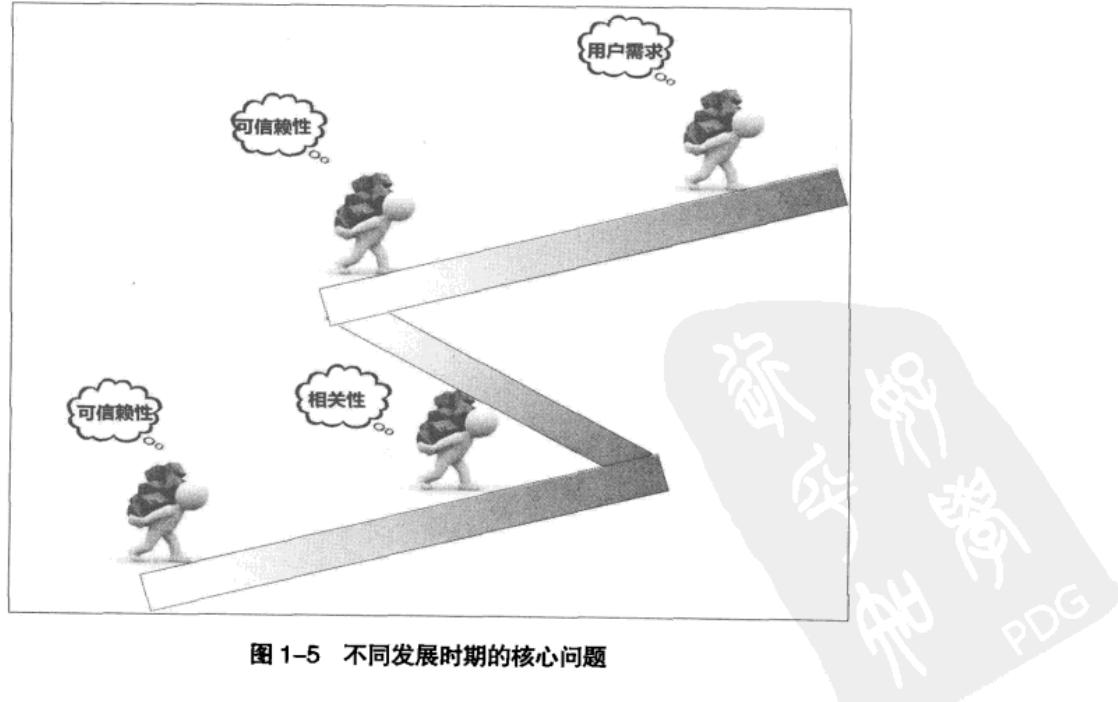


图 1-5 不同发展时期的核心问题

对于分类目录式搜索引擎，其重点关注的是信息的可信度，因为分类目录内收录的网站



经过人工精心筛选，所以具有很强的可信赖性，但是对于用户需求和相关性是不做考虑的，完全靠用户自由浏览来确定。

对于第一代文本检索式搜索引擎，其重点关注的是查询关键词和网页内容的相关性。这种搜索方式假定用户输入的查询关键词就是用户的真实需求，很明显这种假设很难成立。另外，这种搜索方式对于信息的可信度也未做任何识别。

第二代搜索引擎引入链接分析技术，链接关系代表了一种推荐含义，而获得越多推荐的网页其链接分析得分越高，这其实是一种对网页可信度的度量标准。同时，第二代搜索引擎也利用了文本检索模型，来计算查询和网页内容的相关性。所以第二代搜索引擎是综合考虑了信息的相关性和可信性的，但是同样没有对用户需求做关注。

第三代搜索引擎的重点则是用户的真实需求，其他方面则兼顾了第二代搜索引擎的优点，即第三代搜索引擎同时考虑了3个核心问题。

## 1.5 搜索引擎的技术架构

作为互联网应用中最具技术含量的应用之一，优秀的搜索引擎需要复杂的架构和算法，以此来支撑对海量数据的获取、存储，以及对用户查询的快速而准确地响应。本节主要从宏观上介绍搜索引擎的整体架构和各个组成模块的功能。

从架构层面，搜索引擎需要能够对以百亿计的海量网页进行获取、存储、处理的能力，同时要保证搜索结果的质量。如何获取、存储并计算如此海量的数据？如何快速响应用户的查询？如何使得搜索结果能够满足用户的信息需求？这些都是搜索引擎面对的技术挑战。

图1-6是一个通用的搜索引擎架构示意图。搜索引擎由很多技术模块构成，各自负责整体功能的一部分，相互配合形成了完善的整体架构。

搜索引擎的信息源来自于互联网网页，通过网络爬虫将整个互联网的信息获取到本地，因为互联网页面中有相当大比例的内容是完全相同或者近似重复的，“网页去重”模块会对此做出检测，并去除重复内容。

在此之后，搜索引擎会对网页进行解析，抽取出网页主体内容，以及页面中包含的指向其他页面的链接。为了加快响应用户查询的速度，网页内容通过“倒排索引”这种高效查询数据结构来保存，而网页之间的链接关系也会予以保存。之所以要保存链接关系，是因为这种关系在网页相关性排序阶段是可利用的，通过“链接分析”可以判断页面的相对重要性，对于为用户提供准确的搜索结果帮助很大。

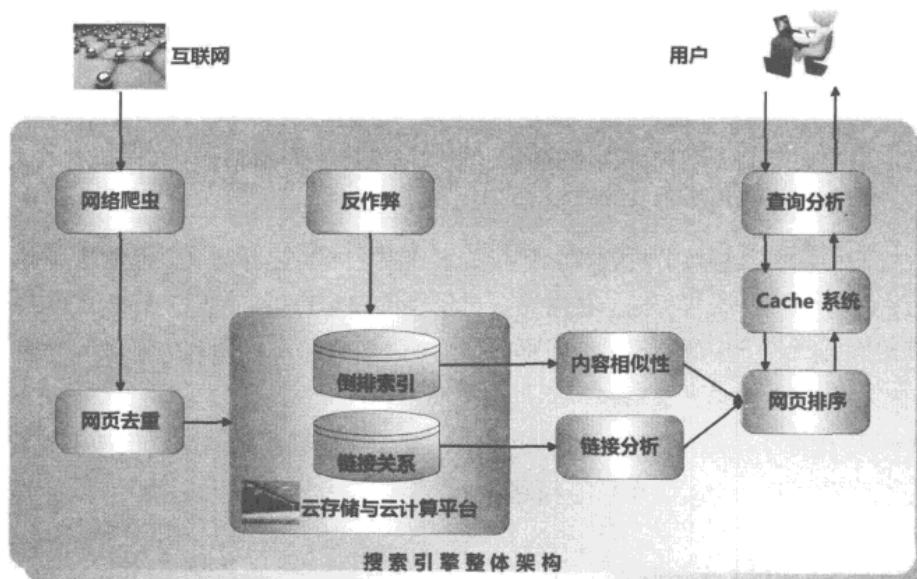


图 1-6 搜索引擎架构

由于网页数量太多，搜索引擎不仅需要保存网页原始信息，还要存储一些中间的处理结果，使用单台或者少量的机器明显是不现实的。Google 等商业搜索引擎为此开发了一整套云存储与云计算平台，使用数以万计的普通 PC 搭建了海量信息的可靠存储与计算架构，以此作为搜索引擎及其相关应用的基础支撑。优秀的云存储与云计算平台已经成为大型商业搜索引擎的核心竞争力。

上面所述是搜索引擎如何获取并存储海量的网页相关信息，这些功能因为不需要实时计算，所以可以被看做是搜索引擎的后台计算系统。搜索引擎的最重要目的是为用户提供准确全面的搜索结果，如何响应用户查询并实时地提供准确结果构成了搜索引擎前台计算系统。

当搜索引擎接收到用户的查询词后，首先需要对查询词进行分析，希望能够结合查询词和用户信息来正确推导用户的真正搜索意图。在此之后，首先在缓存中查找，搜索引擎的缓存系统存储了不同的查询意图对应的搜索结果，如果能够在缓存系统找到满足用户需求的信息，则可以直接将搜索结果返回给用户，这样既省掉了重复计算对资源的消耗，又加快了响应速度；如果保存在缓存的信息无法满足用户需求，搜索引擎需要调用“网页排序”模块功能，根据用户的查询实时计算哪些网页是满足用户信息需求的，并排序输出作为搜索结果。而网页排序最重要的两个参考因素中，一个是内容相似性因素，即哪些网页是和用户查询密切相关的；另外一个是网页重要性因素，即哪些网页是质量较好或者相对重要的，这点往往可以从链接分析的结果获得。结合以上两个考虑因素，就可以对网页进行排序，作为用户查询的搜索结果。



除了上述的子功能模块，搜索引擎的“反作弊”模块成为日益重要的功能。搜索引擎作为互联网用户的上网入口，对于网络流量的引导与分流至关重要，甚至可以说起了决定性的作用。于是，各种“作弊”方式逐渐流行，通过各种手段将网页的搜索排名提高到与其网页质量不相称的位置，这会严重影响用户的搜索体验。所以，如何自动发现作弊网页并对其进行处罚，成为搜索引擎非常重要的组成部分。



## 第2章 网络爬虫

“满面尘灰烟火色，两鬓苍苍十指黑。  
卖炭得钱何所营？身上衣裳口中食。  
可怜身上衣正单，心忧炭贱愿天寒。  
夜来城外一尺雪，晓驾炭车辗冰辙。  
牛困人饥日已高，市南门外泥中歇。  
翩翩两骑来是谁？黄衣使者白衫儿。  
手把文书口称敕，回车叱牛牵向北。  
一车炭，千余斤，宫使驱将惜不得。  
半匹红绡一丈绫，系向牛头充炭直。”

白居易《卖炭翁》

通用搜索引擎的处理对象是互联网网页，目前网页数量以百亿计，所以搜索引擎首先面临的问题就是：如何能够设计出高效的下载系统，以将如此海量的网页数据传送到本地，在本地形成互联网网页的镜像备份。

网络爬虫即起此作用，它是搜索引擎系统中很关键也很基础的构件。本章主要介绍与网络爬虫相关的技术，尽管爬虫技术经过几十年的发展，从整体框架上已相对成熟，但随着互联网的不断发展，也面临着一些有挑战性的新问题。

### 2.1 通用爬虫框架

图 2-1 所示是一个通用的爬虫框架流程。首先从互联网页面中精心选择一部分网页，以这些网页的链接地址作为种子 URL，将这些种子 URL 放入待抓取 URL 队列中，爬虫从待抓取 URL 队列依次读取，并将 URL 通过 DNS 解析，把链接地址转换为网站服务器对应的 IP 地址。然后将其和网页相对路径名称交给网页下载器，网页下载器负责页面内容的下载。对于下载到



本地的网页，一方面将其存储到页面库中，等待建立索引等后续处理；另一方面将下载网页的 URL 放入已抓取 URL 队列中，这个队列记载了爬虫系统已经下载过的网页 URL，以避免网页的重复抓取。对于刚下载的网页，从中抽出其所包含的所有链接信息，并在已抓取 URL 队列中检查，如果发现链接还没有被抓取过，则将这个 URL 放入待抓取 URL 队列末尾，在之后的抓取调度中会下载这个 URL 对应的网页。如此这般，形成循环，直到待抓取 URL 队列为空，这代表着爬虫系统已将能够抓取的网页尽数抓完，此时完成了一轮完整的抓取过程。

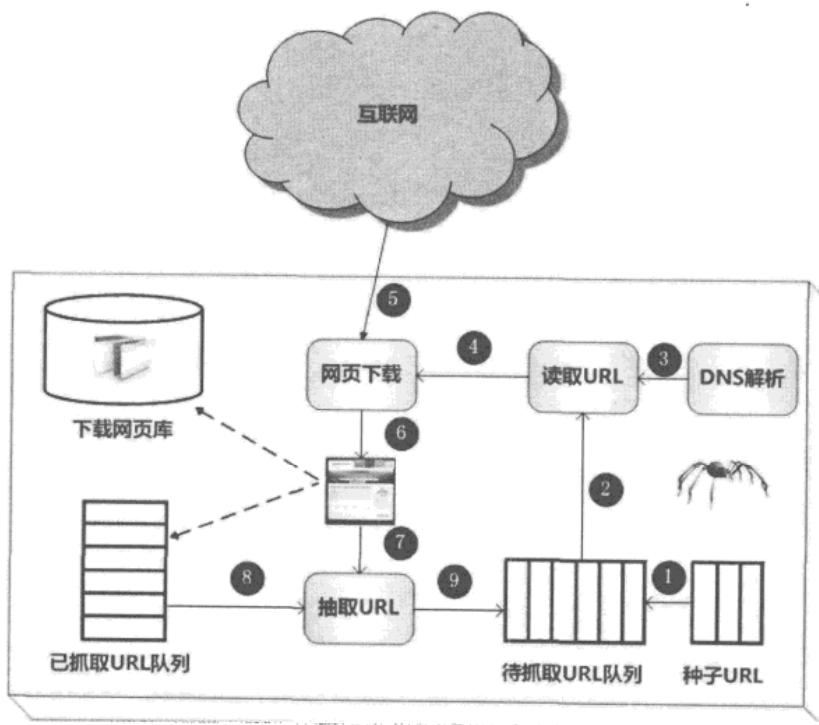


图 2-1 通用爬虫框架

对于爬虫来说，往往还需要进行网页去重及网页反作弊，由于本书有专门章节讲解，所以未在此处列出，详情请参考相关章节。

上述是一个通用爬虫的整体流程，如果从更加宏观的角度考虑，处于动态抓取过程中的爬虫和互联网所有网页之间的关系，可以大致像如图 2-2 所示那样，将互联网页面划分为 5 个部分：

- **已下载网页集合：**爬虫已经从互联网下载到本地进行索引的网页集合。
- **已过期网页集合：**由于网页数量巨大，爬虫完整抓取一轮需要较长时间，在抓取过程中，很多已经下载的网页可能过期。之所以如此，是因为互联网网页处于不断的动态

变化过程中，所以易产生本地网页内容和真实互联网网页不一致的情况。

- **待下载网页集合：**即处于图 2-1 中待抓取 URL 队列中的网页，这些网页即将被爬虫下载。
- **可知网页集合：**这些网页还没有被爬虫下载，也没有出现在待抓取 URL 队列中，不过通过已经抓取的网页或者在待抓取 URL 队列中的网页，总是能够通过链接关系发现它们，稍晚时候会被爬虫抓取并索引。
- **不可知网页集合：**有些网页对于爬虫来说是无法抓取到的，这部分网页构成了不可知网页集合。事实上，这部分网页所占的比例很高。

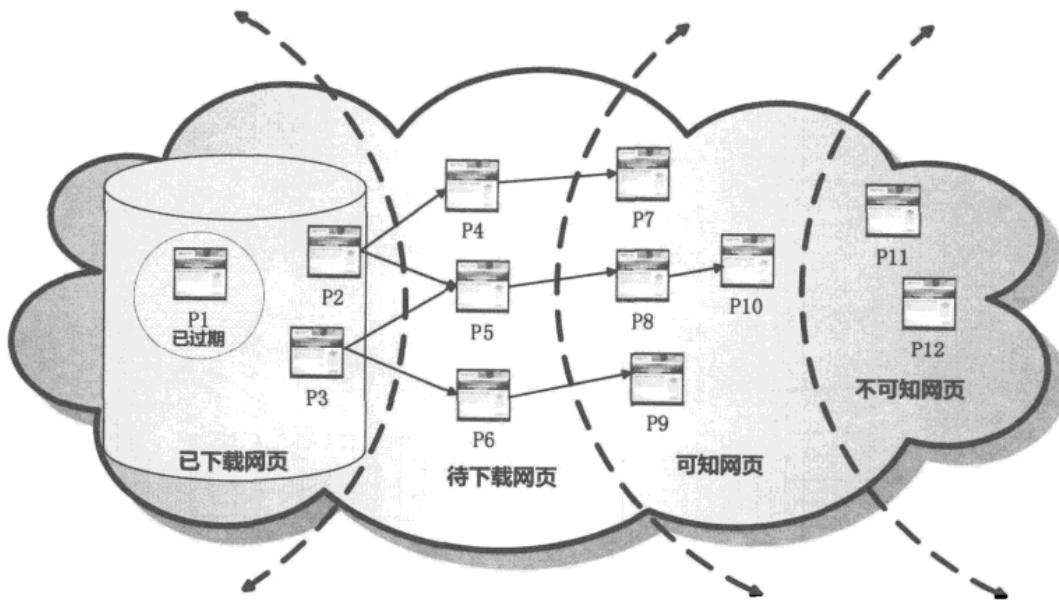


图 2-2 互联网页面划分

从理解爬虫的角度看，对互联网网页给出如上划分有助于深入理解搜索引擎爬虫所面临的主要任务和挑战。

图 2-1 所示是通用的爬虫框架流程，绝大多数爬虫系统遵循此流程，但是并非意味着所有爬虫都如此一致。根据具体应用的不同，爬虫系统在许多方面存在差异，大体而言，可以将爬虫划分为如下 3 种类型。

- **批量型爬虫 (Batch Crawler)：**批量型爬虫有比较明确的抓取范围和目标，当爬虫达到这个设定的目标后，即停止抓取过程。至于具体目标可能各异，也许是设定抓取一定数量的网页即可，也许是设定抓取消耗的时间等，不一而足。



- **增量型爬虫 ( Incremental Crawler )**: 增量型爬虫与批量型爬虫不同，会保持持续不断的抓取，对于抓取到的网页，要定期更新，因为互联网网页处于不断变化中，新增网页、网页被删除或者网页内容更改都很常见，而增量型爬虫需要及时反映这种变化，所以处于持续不断的抓取过程中，不是在抓取新网页，就是在更新已有网页。通用的商业搜索引擎爬虫基本都属此类。
- **垂直型爬虫 ( Focused Crawler )**: 垂直型爬虫关注特定主题内容或者属于特定行业的网页，比如对于健康网站来说，只需要从互联网页面里找到与健康相关的页面内容即可，其他行业的内容不在考虑范围。垂直型爬虫一个最大的特点和难点就是：如何识别网页内容是否属于指定行业或者主题。从节省系统资源的角度来说，不太可能把所有互联网页面下载下来之后再去筛选，这样浪费资源就太过分了，往往需要爬虫在抓取阶段就能够动态识别某个网址是否与主题相关，并尽量不去抓取无关页面，以达到节省资源的目的。垂直搜索网站或者垂直行业网站往往需要此种类型的爬虫。

本章主要讲述增量型爬虫，因为无论从处理的数据量上来讲，还是从所面临的技术难度来讲，这种类型的爬虫相对而言都要复杂一些，当然，其中很多技术对于其他两种类型的爬虫来说也是共通的。

## 2.2 优秀爬虫的特性

对于不同的应用来说，爬虫系统实现方式可能各异，但是实用的爬虫系统都应该具备以下几种特性。

### 高性能

互联网的网页数量庞大如海，所以爬虫的性能至关重要，这里的性能主要是指爬虫下载网页的抓取速度，常见的评价方式是以爬虫每秒能够下载的网页数量作为性能指标，单位时间能够下载的网页数量越多，则爬虫的性能越高。

要提高爬虫的性能，在设计时程序访问磁盘的操作方法及具体实现时数据结构的选择很关键。比如对于待抓取 URL 队列和已抓取 URL 队列，因为 URL 数量非常大，不同实现方式性能表现迥异，所以高效的数据结构对于爬虫性能影响很大。

### 可扩展性

如上所述，爬虫需要抓取的网页数量巨大，即使单个爬虫的性能很高，要将所有网页都下

载到本地，仍然需要相当长的时间周期，为了能够尽可能缩短抓取周期，爬虫系统应该有很好的可扩展性，即很容易通过增加抓取服务器和爬虫数量来达到此目的。

目前实用的大型网络爬虫一定是分布式运行的，即多台服务器专做抓取，每台服务器部署多个爬虫，每个爬虫多线程运行，通过多种方式增加并发性。对于巨型的搜索引擎服务商来说，可能还要在全球范围、不同地域分别部署数据中心，爬虫也被分配到不同的数据中心，这样对于提高爬虫系统的整体性能是很有帮助的。

### 健壮性

爬虫要访问各种类型的网站服务器，可能会遇到很多种非正常情况，比如网页 HTML 编码不规范，被抓取服务器突然死机，甚至是爬虫陷阱等。爬虫对各种异常情况能够正确处理非常重要，否则可能会不定期停止工作，这是无法忍受的。

从另外一个角度来讲，假设爬虫程序在抓取过程中死掉，或者爬虫所在的服务器宕机，健壮的爬虫系统应该能够做到：再次启动爬虫时，能够恢复之前抓取的内容和数据结构，而不是每次都需要把所有工作完全从头做起，这也是爬虫健壮性的一种体现。

### 友好性

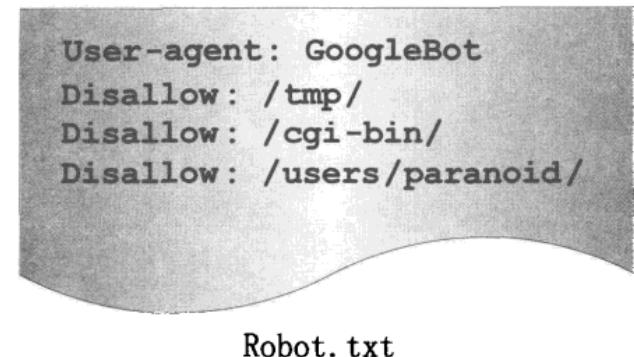
爬虫的友好性包含两方面的含义：一是保护网站的部分私密性，另一是减少被抓取网站的网络负载。

爬虫抓取的对象是各种类型的网站，对于网站拥有者来说，有些内容并不希望被所有人搜索到，所以需要设定协议，来告知爬虫哪些内容是不允许抓取的。目前有两种主流的方法可达此目的：爬虫禁抓协议和网页禁抓标记。

爬虫禁抓协议（Robot Exclusion Protocol）指的是由网站所有者生成一个指定的文件 `robot.txt`，并放在网站服务器的根目录下，这个文件指明了网站中哪些目录下的网页是不允许爬虫抓取的。具有友好性的爬虫在抓取该网站的网页前，首先要读取 `robot.txt` 文件，对于禁止抓取的网页一般不进行下载。

图 2-3 给出了某个网站对应的 `robot.txt` 文件内容，`User-agent` 字段指出针对哪个爬虫，图中示例为 Google 的爬虫，而 `Disallow` 字段则指出不允许抓取的目录。

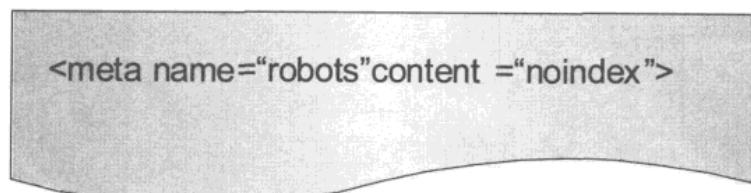
爬虫禁抓协议一般以目录为单位，即整个目录下的网页或内容都不允许被抓取。如果只想让单个网页不被抓取，该如何做呢？网页禁抓标记（Robot META tag）可在此种场合派上用场。



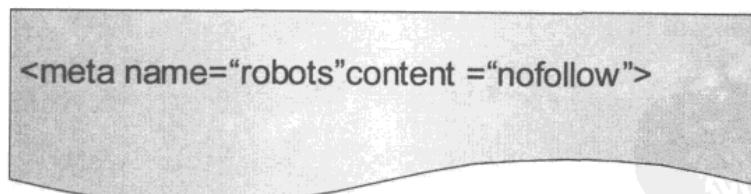
Robot.txt

图 2-3 爬虫禁抓协议

图 2-4 给出了网页禁抓标记的示例，即在网页的 HTML 代码里加入 `meta name="robots"` 标记，`content` 字段指出允许或者不允许爬虫的哪些行为。可以分为两种情形，一种是告知爬虫不要索引该网页内容，以 `noindex` 作为标记；另外一种情形是告知爬虫不要抓取网页所包含的链接，以 `nofollow` 作为标记。通过这种方式，可以达到对网页内容的一种隐私保护。



情形一：禁止索引网页内容



情形二：禁止抓取网页链接

图 2-4 网页禁抓标记

遵循以上协议的爬虫可以被认为是友好的，这是从保护私密性的角度考虑的。另外一种友好性则是，希望爬虫对某网站的访问造成的网络负载较低。爬虫一般会根据网页的链接连续获取某网站的网页，如果爬虫访问网站频率过高，会给网站服务器造成很大的访问压力，有时候甚至会影响网站的正常访问，造成类似 DOS 攻击的效果，所以为了减少网站的网络负载，友好性的爬虫应该在抓取策略部署时考虑每个被抓取网站的负载，在尽可能不影响爬虫性能的情况下，合理地规划抓取策略。

况下，减少对单一站点短期内的高频访问。

### 2.3 爬虫质量的评价标准

上节介绍了优秀爬虫应该具备的几个特性，这是从爬虫开发者的角度考虑的。如果从搜索引擎用户体验的角度考虑，对爬虫的工作效果有不同的评价标准，其中最主要的3个标准是：抓取网页覆盖率、抓取网页时新性及抓取网页重要性。如果这3个方面做得好，则搜索引擎用户体验必佳。

图2-5展示的是一个抽象的互联网网页与搜索引擎爬虫抓取网页的对比图，对于现有的搜索引擎来说，还不存在哪个搜索引擎有能力将互联网上出现的所有网页都下载并建立索引的，所有搜索引擎只能索引互联网的一部分，而所谓的抓取网页覆盖率指的是爬虫抓取的网页数量占互联网所有网页数量的比例，图中所示互联网有5个网页，而爬虫系统抓取了其中的3个。覆盖率越高，等价于搜索引擎的召回率越高，用户体验也越好。

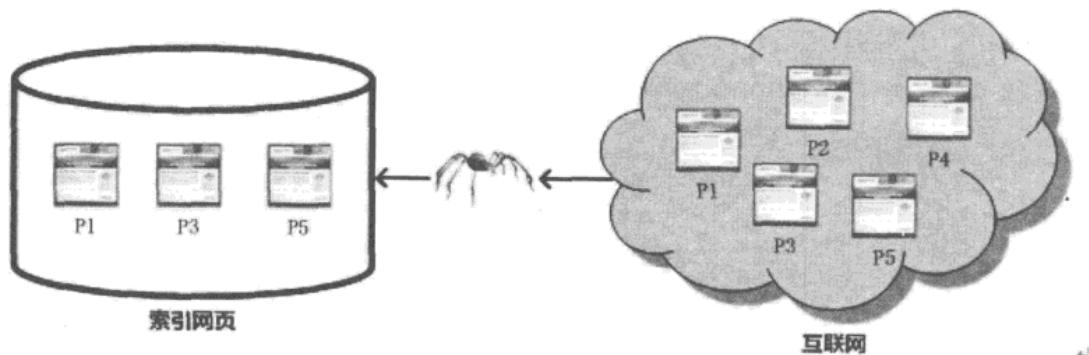


图2-5 索引网页和互联网网页对比

对于爬虫抓到本地的网页来说，很多网页可能已经发生变化，或者被删除，或者内容被更改，因为爬虫完整抓取一轮需要较长的时间周期，所以抓取到的网页中必有一部分是过期的数据，即不能在网页变化后第一时间反映到网页库中，所以网页库中过期的数据越少，则网页的时新性越好，这对用户体验的作用不言而喻。如果时新性不好，用户搜到的是过时数据，甚至可能网页都已不复存在，使用产品时其心情如何可想而知。

互联网尽管网页众多，但是每个网页重要性差异很大，比如来自雅虎新闻的网页和某个作弊网页相比，其重要性差异判若云泥。如果搜索引擎爬虫抓回的网页大都是比较重要的网页，则可说其在抓取网页重要性方面做得很好。这方面做得好，等价于搜索引擎的搜索精度高。

通盘考虑以上3个因素，可以将目前爬虫研发的目标简单描述如下：在资源有限的情况下，



既然搜索引擎只能抓取互联网现存网页的一部分，那么就尽可能选择比较重要的那部分页面来索引；对于已经抓取到的网页，尽可能快地更新其内容，使得索引网页和互联网对应页面内容同步更新；在此基础上，尽可能扩大抓取范围，抓取到更多以前无法发现的网页。3个“尽可能”基本说清楚了爬虫系统为增强用户体验而奋斗的目标。

大型商业搜索引擎为了满足以上3个质量标准，大都开发了多套针对性很强的爬虫系统。以Google为例，如图2-6所示，其至少包含两套不同目的的爬虫系统，一套被称为Fresh Bot，主要考虑网页的时新性，对于内容更新频繁的网页，目前可以达到以秒计的更新周期；而另外一套被称之为Deep Crawl Bot，主要针对其他更新不是那么频繁的网页抓取，以天为更新周期。除此之外，Google投入了很大精力研发针对暗网的抓取系统，本章后续小节会述及此面。

本章后续内容会重点介绍以下4个方面的技术：抓取策略、网页更新策略、暗网抓取和分布式爬虫。之所以如此安排，因为这几项技术是爬虫系统中至关重要的组成部分，基本决定了爬虫系统的质量和性能。

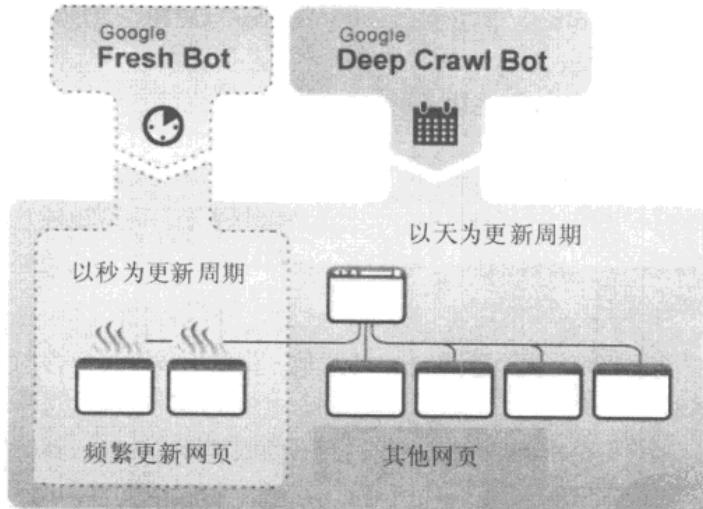


图2-6 Google的两套爬虫系统

其中暗网抓取技术是为了增加网页覆盖率，网页更新策略是为了增加下载网页的时效性，网页重要性评价标准则是抓取策略的核心，而分布式爬虫的分布机制也决定了爬虫系统的性能。正是基于此考虑，后续章节将陆续介绍这4个方面的关键技术。

## 2.4 抓取策略

在爬虫系统中，待抓取URL队列是很关键的部分，需要爬虫抓取的网页URL在其中顺序

排列，形成一个队列结构，调度程序每次从队列头取出某个 URL，发送给网页下载器下载页面内容，每个新下载的页面包含的 URL 会追加到待抓取 URL 队列的末尾，如此形成循环，整个爬虫系统可以说是由这个队列驱动运转的。

待抓取 URL 队列中的页面 URL 顺序是如何确定的？上面所述将新下载页面中包含的链接追加到队列尾部，这固然是一种确定队列 URL 顺序的方法，但并非唯一的手段，事实上，还可以采纳很多其他技术，将队列中待抓取的 URL 进行排序。而爬虫的不同抓取策略，就是利用不同的方法来确定待抓取 URL 队列中 URL 优先顺序的。

爬虫的抓取策略有很多种，但不论方法如何，其基本目标一致：优先选择重要网页进行抓取。在爬虫系统中，所谓网页的重要性，其评判标准可以选择不同方法，但是大部分都是按照网页的流行性来定义的，在本书的“链接分析”一章介绍的 PageRank 就是评价网页重要性的常用标准。

抓取策略方法众多，本节只选择已经被证明效果较好或者是比较有代表性的解决方案，包括以下 4 种：宽度优先遍历策略、非完全 PageRank 策略、OCIP 策略及大站优先策略。

#### 2.4.1 宽度优先遍历策略 ( Breath First )

宽度优先遍历是一种非常简单直观且历史也很悠久的遍历方法，在搜索引擎爬虫一出现就开始采用，新提出的抓取策略往往会将这种方法作为比较基准。但应该注意到的是，这种策略也是一种相当强悍的方法，很多新方法实际效果不见得比宽度优先遍历策略好，所以至今这种方法也是很多实际爬虫系统优先采用的抓取策略。

那么，什么是宽度优先遍历呢？其实上文所说的“将新下载网页包含的链接直接追加到待抓取 URL 队列末尾”，这就是宽度优先遍历的思想。也就是说，这种方法并没有明确提出和使用网页重要性衡量标准，只是机械地将新下载的网页抽取链接，并追加到待抓取 URL 队列中，以此安排 URL 的下载顺序。图 2-7 是这种策略的示意图：假设队列头的网页是 1 号网页，从 1 号网页中抽出 3 个链接指向 2 号、3 号和 4 号网页，于是按照编号顺序依次放入待抓取 URL 队列，图中网页的编号就是这个网页在待抓取 URL 队列中的顺序编号，之后爬虫以此顺序进行下载。

实验表明这种策略效果很好，虽然看似机械，但实际上的网页抓取顺序基本是按照网页的重要性排序的。之所以如此，有研究人员认为：如果某个网页包含很多入链，那么更有可能被宽度优先遍历策略早早抓到，而入链个数从侧面体现了网页的重要性，即实际上宽度优先遍历策略隐含了一些网页优先级假设。

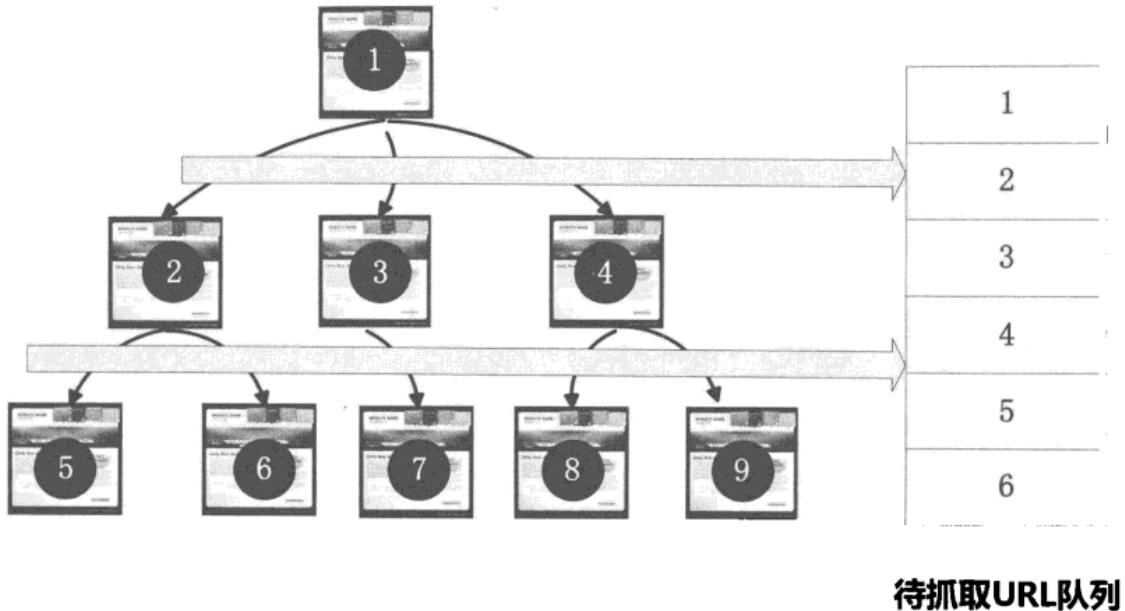


图 2-7 宽度优先遍历策略

#### 2.4.2 非完全 PageRank 策略 (Partial PageRank)

PageRank 是一种著名的链接分析算法，可以用来衡量网页的重要性（技术细节请参考本书“链接分析”一章）。很自然地，可以想到用 PageRank 的思想来对 URL 优先级进行排序。但是这里有个问题，PageRank 是个全局性算法，也就是说当所有网页都下载完成后，其计算结果才是可靠的，而爬虫的目的就是去下载网页，在运行过程中只能看到一部分页面，所以在抓取阶段的网页是无法获得可靠 PageRank 得分的。

如果我们仍然坚持在这个不完整的互联网页面子集内计算 PageRank 呢？这就是非完全 PageRank 策略的基本思路：对于已经下载的网页，加上待抓取 URL 队列中的 URL 一起，形成网页集合，在此集合内进行 PageRank 计算，计算完成后，将待抓取 URL 队列里的网页按照 PageRank 得分由高到低排序，形成的序列就是爬虫接下来应该依次抓取的 URL 列表。这也是为何称之为“非完全 PageRank”的原因。

如果每次新抓取到一个网页，就将所有已经下载的网页重新计算新的非完全 PageRank 值，明显效率太低，在现实中是不可行的。一个折中的办法是：每当新下载的网页攒够  $K$  个，然后将所有下载页面重新计算一遍新的非完全 PageRank。这样的计算效率还勉强能够接受，但是又引来了新的问题：在展开下一轮 PageRank 计算之前，从新下载的网页抽取出包含的链接，很有可能这些链接的重要性非常高，理应优先下载，这种情况该如何解决？非完全 PageRank

赋予这些新抽取出来但是又没有 PageRank 值的网页一个临时 PageRank 值，将这个网页的所有入链传导的 PageRank 值汇总，作为临时 PageRank 值，如果这个值比待抓取 URL 队列中已经计算出来 PageRank 值的网页高，那么优先下载这个 URL。

图 2-8 是非完全 PageRank 策略的一个简略示意图。我们设定每下载 3 个网页即进行新的 PageRank 计算，此时已经有 {P1, P2, P3}3 个网页下载到本地，这 3 个网页包含的链接指向 {P4, P5, P6}，形成了待抓取 URL 队列，如何决定其下载顺序？将这 6 个网页形成新的集合，对这个集合计算 PageRank 值，这样 P4、P5 和 P6 就获得自己对应的 PageRank 值，由大到小排序，即可得出其下载顺序。这里可以假设顺序为：P5、P4、P6，当下载 P5 页面后抽出链接，指向页面 P8，此时赋予 P8 临时 PageRank 值，如果这个值大于 P4 和 P6 的 PageRank，那么优先下载 P8。如此不断循环，即形成了非完全 PageRank 策略的计算思路。

非完全 PageRank 看上去相对复杂，那么是否效果一定优于简单的宽度优先遍历策略呢？不同的实验结果存在争议，有些表明非完全 PageRank 结果略优，有些实验结果结论则恰恰相反。更有研究人员指出：非完全 PageRank 计算得出的重要性与完整的 PageRank 计算结果差异很大，不应作为衡量抓取过程中 URL 重要性计算的依据。

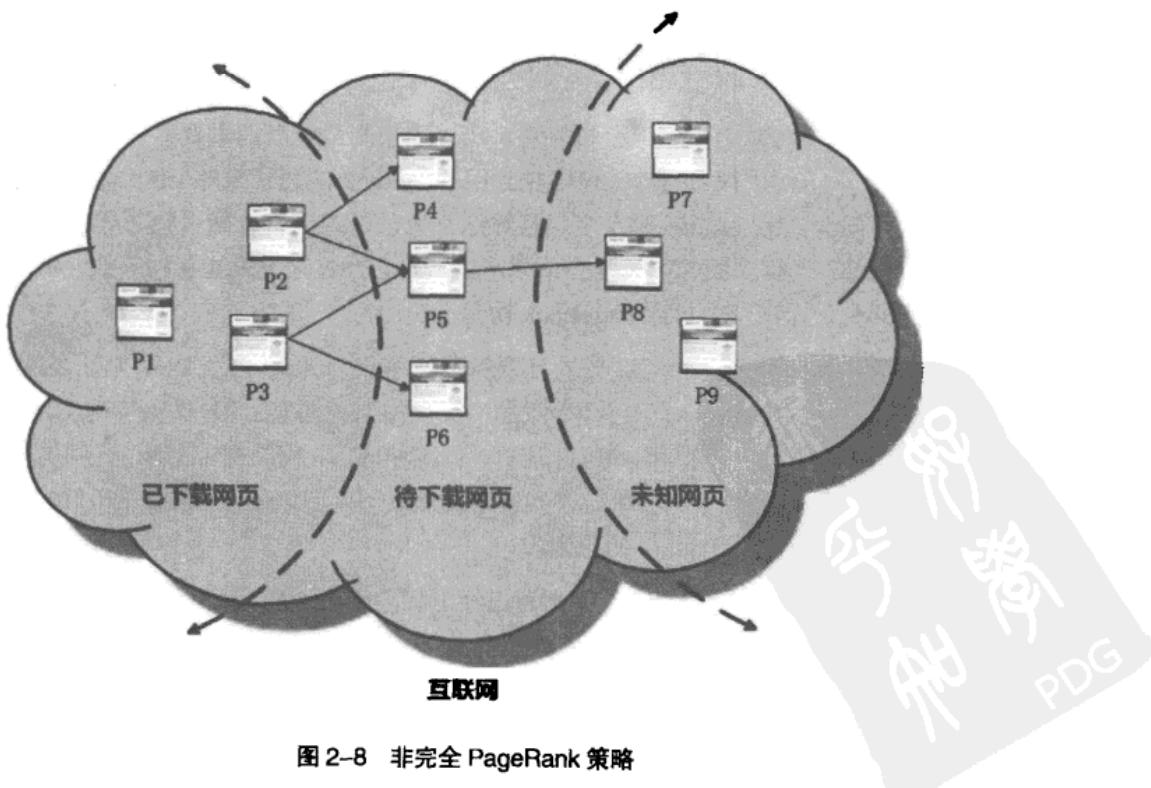


图 2-8 非完全 PageRank 策略



### 2.4.3 OCIP 策略 (Online Page Importance Computation)

OCIP 的字面含义是“在线页面重要性计算”，可以将其看做是一种改进的 PageRank 算法。在算法开始之前，每个互联网页面都给予相同的“现金”(cash)，每当下载了某个页面 P 后，P 将自己拥有的“现金”平均分配给页面中包含的链接页面，把自己的“现金”清空。而对于待抓取 URL 队列中的网页，则根据其手头拥有的现金金额多少排序，优先下载现金最充裕的网页。OCIP 从大的框架上与 PageRank 思路基本一致，区别在于：PageRank 每次需要迭代计算，而 OCIP 策略不需要迭代过程，所以计算速度远远快于 PageRank，适合实时计算使用。同时，PageRank 在计算时，存在向无链接关系网页的远程跳转过程，而 OCIP 没有这一计算因子。实验结果表明，OCIP 是种较好的重要性衡量策略，效果略优于宽度优先遍历策略。

### 2.4.4 大站优先策略 (Larger Sites First)

大站优先策略思路很直接：以网站为单位来衡量网页重要性，对于待抓取 URL 队列中的网页，根据所属网站归类，如果哪个网站等待下载的页面最多，则优先下载这些链接。其本质思想倾向于优先下载大型网站，因为大型网站往往包含更多的页面。鉴于大型网站往往是著名企业的内容，其网页质量一般较高，所以这个思路虽然简单，但是有一定依据。实验表明这个算法效果也要略优于宽度优先遍历策略。

## 2.5 网页更新策略

互联网的动态性是其显著特征，随时都有新出现的页面，页面的内容被更改或者本来存在的页面被删除。对于爬虫来说，并非将网页抓取到本地就算完成任务，也要体现出互联网的这种动态性。本地下载的网页可被看做是互联网页面的“镜像”，爬虫要尽可能保证其一致性。可以假设一种情况：某个网页已被删除或者内容做出重大变动，而搜索引擎对此惘然无知，仍然按其旧有内容排序，将其作为搜索结果提供给用户，其用户体验之糟糕不言而喻。所以，对于已经抓取过的网页，爬虫还要负责保持其内容和互联网页面内容的同步，这取决于爬虫所采用的网页更新策略。

网页更新策略的任务是要决定何时重新抓取之前已经下载过的网页，以尽可能使得本地下载网页和互联网原始页面内容保持一致。常用的网页更新策略有 3 种：历史参考策略、用户体验策略和聚类抽样策略。

### 2.5.1 历史参考策略

历史参考策略是最直观的一种更新策略，它建立于如下假设之上：过去频繁更新的网页，那么将来也会频繁更新。所以，为了预估某个网页何时进行更新，可以通过参考其历史更新情况来做出决定。

这种方法往往利用泊松过程来对网页的变化进行建模，根据每个网页过去的变动情况，利用模型预测将来何时内容会再次发生变化，以此来指导爬虫的抓取过程。但是不同方法侧重不尽相同，比如有的研究将一个网页划分成不同的区域，抓取策略应该忽略掉广告栏或者导航栏这种不重要区域的频繁变化，而集中在主题内容的变化探测和建模上。

### 2.5.2 用户体验策略

一般来说，搜索引擎用户提交查询后，相关的搜索结果可能成千上万，而用户没有耐心查看排在后面的搜索结果，往往只查看前3页搜索内容。用户体验策略就是利用搜索引擎用户的这个特点来设计更新策略的。

这种更新策略以用户体验为核心，即使本地索引的网页内容是过时的，但是如果不影响用户体验，那么晚些更新这些过时网页也未尝不可。所以判断一个网页何时更新为好，取决于这个网页的内容变化所带来搜索质量的变化（往往采用搜索结果排名的变化来衡量），影响越大的网页，则应该越快更新。

用户体验策略保存网页的多个历史版本，并根据过去每次内容变化对搜索质量的影响，得出一个平均值，以此作为判断爬虫重抓该网页时机的参考依据，对于影响越厉害的网页，则越优先调度重新抓取。

### 2.5.3 聚类抽样策略

上面介绍的两种网页更新策略严重依赖网页的历史更新信息，因为这是能够进行后续计算的基础。但是在现实中，为每个网页保存其历史信息，搜索系统会大量增加额外负担。从另外一个角度考虑，如果是首次抓取到的网页，因为没有历史信息，所以也就无法按照这两种思路去预估其更新周期。聚类抽样策略即是为了解决上述缺点而提出的。

聚类抽样策略认为：网页具有一些属性，根据这些属性可以预测其更新周期，具有相似属性的网页，其更新周期也是类似的。于是，可以根据这些属性将网页归类，同一类别内的网页具有相同的更新频率。为了计算某个类别的更新周期，只需对类别内网页进行采样，以这些被采样网页的更新周期作为类别内所有其他网页的更新周期。与之前叙述的两种方法相比较，这种策略一方面无须为每个网页保存历史信息；另一方面，对于新网页，即使没有历史信息，也



可以根据其所属类别来对其进行更新。

聚类抽样策略基本流程如图 2-9 所示，首先根据网页所表现出的特征，将其聚类成不同的类别，每个类别内的网页具有相似的更新周期。从类别中抽取一部分最有代表性的网页（一般抽取最靠近类中心的那些网页），对这些网页计算其更新周期，那么这个更新周期适用于类别内的所有网页，之后即可根据网页所属类别来决定其更新频率。

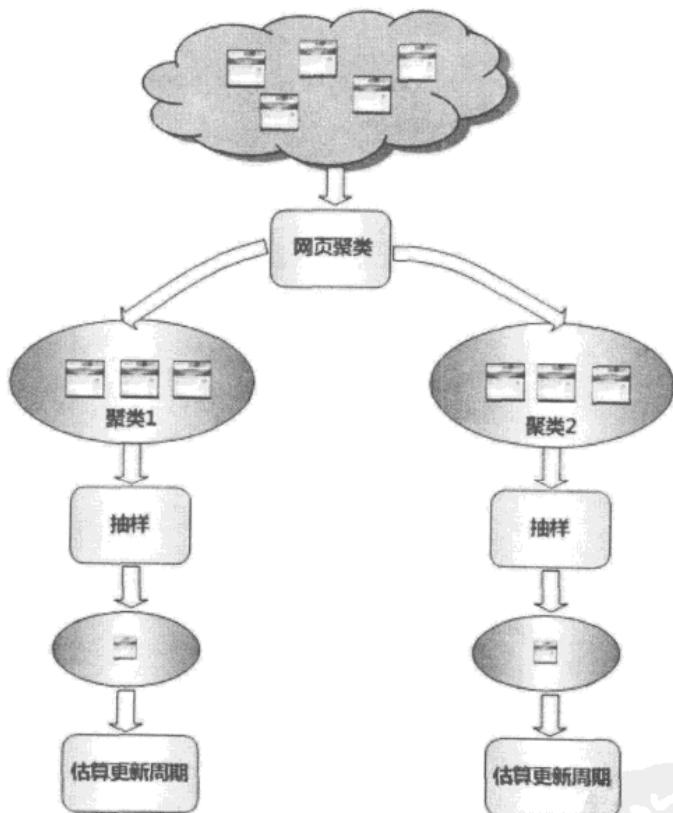


图 2-9 聚类抽样策略

在 Tan 等人的研究中，将能够体现网页更新周期的属性特征划分为两大类：静态特征和动态特征。静态特征包括：页面的内容、图片数量、页面大小、链接深度、PageRank 值等十几种；而动态特征则体现了静态特征随着时间的变化情况，比如图片数量的变化情况、入链出链的变化情况等。根据这两类特征，即可对网页进行聚类。

图 2-9 所示为一个较为通用的流程，不同算法在细节处有差异。比如有些研究直接省略聚类这个步骤，而是以网站作为聚类单位，即假设属于同一个网站的网页具有相同的更新周期，对网站内页面进行抽样，计算其更新频率，之后网站内所有网页以这个更新周期为准。这个假

设虽显粗糙，因为很明显同一网站内网页更新频率差异很大，但是可以省掉聚类这个步骤，在计算效率方面会更可行些。

相关实验表明，聚类抽样策略效果好于前述两种更新策略，但是对以亿计的网页进行聚类，其难度也是非常巨大的。

## 2.6 暗网抓取 (Deep Web Crawling)

物理学研究表明，在目前宇宙所有物质的总体质量中，星系等可见物质只占其中的 20%，不可探测的暗物质则占据了总质量的大约 80%。互联网中的暗网可与宇宙中的暗物质相类比，而其所占网页的比例，更是远大于暗物质占宇宙质量的比例，大约百倍于目前的明网 (Surfacing Web) 网页。

所谓暗网，是指目前搜索引擎爬虫按照常规方式很难抓取到的互联网页面。如前所述，搜索引擎爬虫依赖页面中的链接关系发现新的页面，但是很多网站的内容是以数据库方式存储的，典型的例子是一些垂直领域网站，比如携程旅行网的机票数据，很难有显式链接指向数据库内的记录，往往是服务网站提供组合查询界面，只有用户按照需求输入查询之后，才可能获得相关数据。所以，常规的爬虫无法索引这些数据内容，这是暗网的命名由来。图 2-10 是携程旅行网的机票搜索界面和当当网的图书搜索界面。



图 2-10 携程的机票搜索界面和当当网的图书搜索界面

为了能够对暗网数据进行索引，需要研发与常规爬虫机制不同的系统，这类爬虫被称做暗网爬虫。暗网爬虫的目的是将暗网数据从数据库中挖掘出来，并将其加入搜索引擎的索引，这



样用户在搜索时便可利用这些数据，增加信息覆盖程度。

目前大型搜索引擎服务提供商都将暗网挖掘作为重要研究方向，因为这直接关系到索引量的大小。在此领域的技术差异，将直接体现在搜索结果的全面性上，自然是竞争对手之间的必争之地。Google 目前将其作为重点研发方向，而百度的“阿拉丁计划”目的也在于此。

垂直网站提供的搜索界面，往往需要人工选择或者填写内容，比如机票搜索需要选择出发地、到达地和日期，图书搜索需要指出书名或者作者。而暗网爬虫为了能够挖掘数据库的记录，必须模拟人的行为，填写内容并提交表单。对于暗网爬虫来说，其技术挑战有两点：一是查询组合太多，如果一一组合遍历，那么会给被访问网站造成太大压力，所以如何精心组合查询选项是个难点；第二点在于：有的查询是文本框，比如图书搜索中需要输入书名，爬虫怎样才能够填入合适的内容？这个也颇具挑战性。

### 2.6.1 查询组合问题

暗网爬虫希望能够将某个垂直网站数据库的记录尽可能多地下载到索引库中，但这是有约束和前提条件的，即不能给被访问网站造成过大的访问压力，同时尽可能节省爬虫本身的资源。

垂直搜索网站往往会给用户提供多个查询输入框，不同输入框代表了搜索对象某方面的属性，通过组合这些属性来将搜索范围缩小。对于暗网爬虫来说，一个简单粗暴的方式就是：将各个输入框可能的输入值组合起来形成查询，比如对于机票查询来说，将所有出发城市、所有目的地城市和时间范围的选项一一组合，形成大量的查询，提交给垂直搜索引擎，从其搜索结果里提炼数据库记录。这么做比较野蛮，而且也不是很必要，因为很多组合是无效的，大量的返回结果为空，同时对被访问网站造成了巨大的流量压力。

Google 对此提出了解决方案，称之为富含信息查询模板（Informative Query Templates）技术，为了了解其技术原理，首先需要明白什么是查询模板。我们以图 2-11 所示的“职位搜索”垂直网站来说明。

图 2-11 “职位搜索”垂直网站

在图 2-11 的示例中，为了描述一个职位，完整的查询由 3 个不同的属性构成：职位类别、行业类别和工作地点。如果在向搜索引擎提交查询的时候，部分属性被赋予了值，而其他属性

不赋值，则这几个赋值的属性一起构成了一个查询模板。

图 2-12 是若干个“查询模板”的示例，如果模板包含一个属性，则称之为一维模板，图中模板 1 是一维模板，模板 2 和模板 3 是两个二维模板，模板 4 是三维模板。

模板1= {职位类别}

Http://www.zhiwei.com /query?职位类别= 人力资源& 行业类别= 不限& 工作地点= 不限

模板2= [职位类别, 行业类别]

Http://www.zhiwei.com /query?职位类别= 人力资源& 行业类别= 电子商务& 工作地点= 不限

模板3= [职位类别, 工作地点]

Http://www.zhiwei.com /query?职位类别= 人力资源& 行业类别= 不限& 工作地点= 北京

模板4= [职位类别, 行业类别, 工作地点]

Http://www.zhiwei.com /query?职位类别= 人力资源& 行业类别= 电子商务& 工作地点= 上海

图 2-12 查询模板示例

对于一个固定的垂直搜索来说，其查询模板组合起来有很多，我们的任务是找到富含信息的查询模板，那么什么又是富含信息查询模板呢？Google 的技术方案是如此定义的：对于某个固定的查询模板来说，如果给模板内每个属性都赋值，形成不同的查询组合，提交给垂直搜索引擎，观察所有返回页面的内容，如果相互之间内容差异较大，则这个查询模板就是富含信息查询模板。

以图 2-12 中的模板 2 为例说明，其包含了两个属性：职位类别和行业类别。职位类别有 3 种不同赋值，行业类别有 2 种不同赋值，两者组合有 6 种不同的组合方式，形成这个模板的 6 个查询。将这 6 个查询分别提交给职位搜索引擎，观察返回页面内容变化情况，如果大部分返回内容都相同或相似，则说明这个查询模板不是富含信息查询模板，否则可被认为是富含信息查询模板。

之所以做如此规定，是基于如下考虑：如果返回结果页面内容重复太多，很可能这个查询模板的维度太高，导致很多种组合无搜索结果，抑或构造的查询本身就是错误的，搜索系统返回了错误页面。

假设按照上面方式对所有查询模板一一试探，判断其是否富含信息查询模板，则因为查询模板数量太多，系统效率还是会很低。为了进一步减少提交的查询数目，Google 的技术方案使用了 ISIT 算法。



ISIT 算法的基本思路是：首先从一维模板开始，对一维查询模板逐个考察，看其是否富含信息查询模板，如果是的话，则将这个一维模板扩展到二维，再次依次考察对应的二维模板，如此类推，逐步增加维数，直到再也无法找到富含信息查询模板为止。通过这种方式，就可以找到绝大多数富含信息查询模板，同时也尽可能减少了提交的查询总数，有效达到了目的。Google 的评测结果证明，这种方法和完全组合方式比，能够大幅度提升系统效率。

如果读者对于数据挖掘有所了解，可以看出，Google 提出的算法和数据挖掘里经典的 Apriori 规则挖掘算法有异曲同工之处。

## 2.6.2 文本框填写问题

对于输入中的文本框，需要爬虫自动生成查询，图 2-13 给了一个常用做法的流程图。

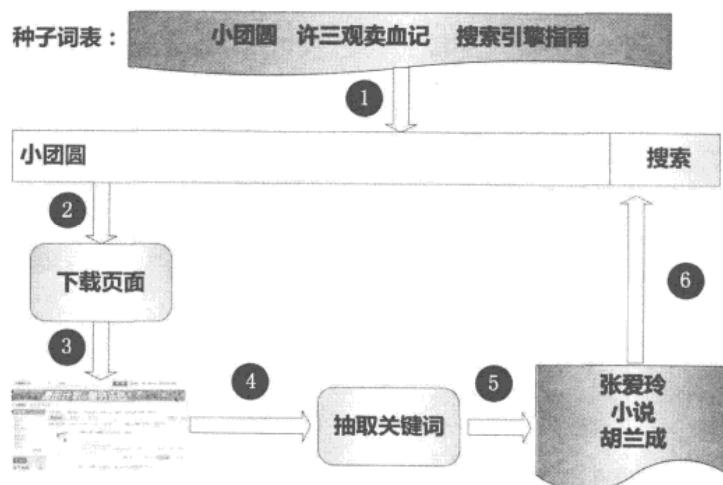


图 2-13 文本框自动填写

在爬虫运转起来之前，因为对目标网站一无所知，所以必须人工提供一些提示。在此例中，通过人工观察网站进行定位，提供一个与网站内容相关的初始种子查询关键词表，对于不同的网站，需要人工提供不同的词表，以此作为爬虫能够继续工作的基础条件。爬虫根据初始种子词表，向垂直搜索引擎提交查询，并下载返回的结果页面。之后从返回结果页面里自动挖掘出相关的关键词，并形成一个新的查询列表，依次将新挖掘出的查询提交给搜索引擎。如此往复，直到无法下载到新的内容为止。通过这种人工启发结合递归迭代的方式，尽可能覆盖数据库里的记录。

## 2.7 分布式爬虫

对于商业搜索引擎来说，分布式爬虫架构是必须采用的技术。面对海量待抓取网页，只有采取分布式架构，才有可能在较短时间内完成一轮抓取工作。

分布式爬虫可以分为若干个分布式层级，不同的应用可能由其中部分层级构成，图 2-14 是一个大型分布式爬虫的 3 个层级：分布式数据中心、分布式抓取服务器及分布式爬虫程序。整个爬虫系统由全球多个分布式数据中心共同构成，每个数据中心负责抓取本地域周边的互联网网页，比如欧洲的数据中心负责抓取英国、法国、德国等欧洲国家的网页，由于爬虫与要抓取的网页地缘较近，在抓取速度上会较远程抓取快很多。

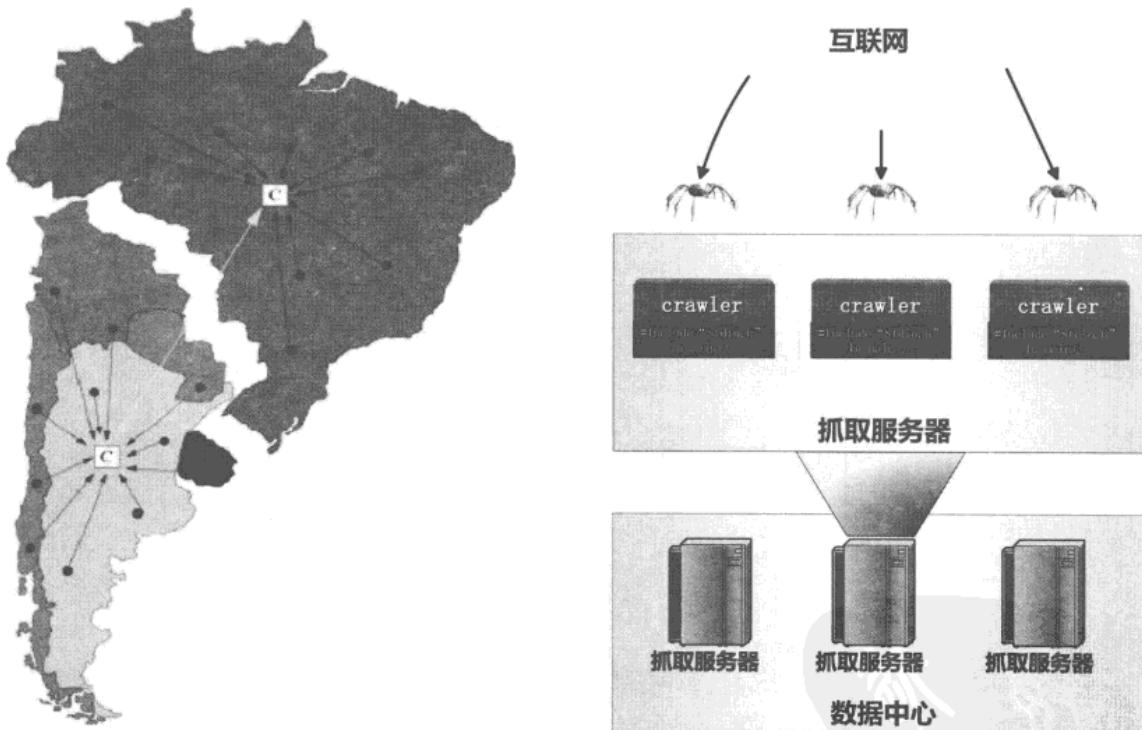


图 2-14 分布式爬虫的几个层级

每个数据中心又由多台高速网络连接的抓取服务器构成，而每台服务器又可以部署多个爬虫程序。通过多层次级的分布式爬虫体系，才可能保证抓取数据的及时性和全面性。

对于同一数据中心的多台抓取服务器，不同机器之间的分工协同方式会有差异，常见的分布式架构有两种：主从式分布爬虫和对等式分布爬虫。

### 2.7.1 主从式分布爬虫 (Master-Slave)

对于主从式分布爬虫，不同的服务器承担不同的角色分工（参考图 2-15），其中有一台专门负责对其他服务器提供 URL 分发服务，其他机器则进行实际的网页下载。URL 服务器维护待抓取 URL 队列，并从中获得待抓取网页的 URL，分配给不同的抓取服务器，另外还要对抓取服务器之间的工作进行负载均衡，使得各个服务器承担的工作量大致相等，不至于出现忙的过忙、闲的过闲的情形。抓取服务器之间没有通信联系，每个抓取服务器只和 URL 服务器进行消息传递。

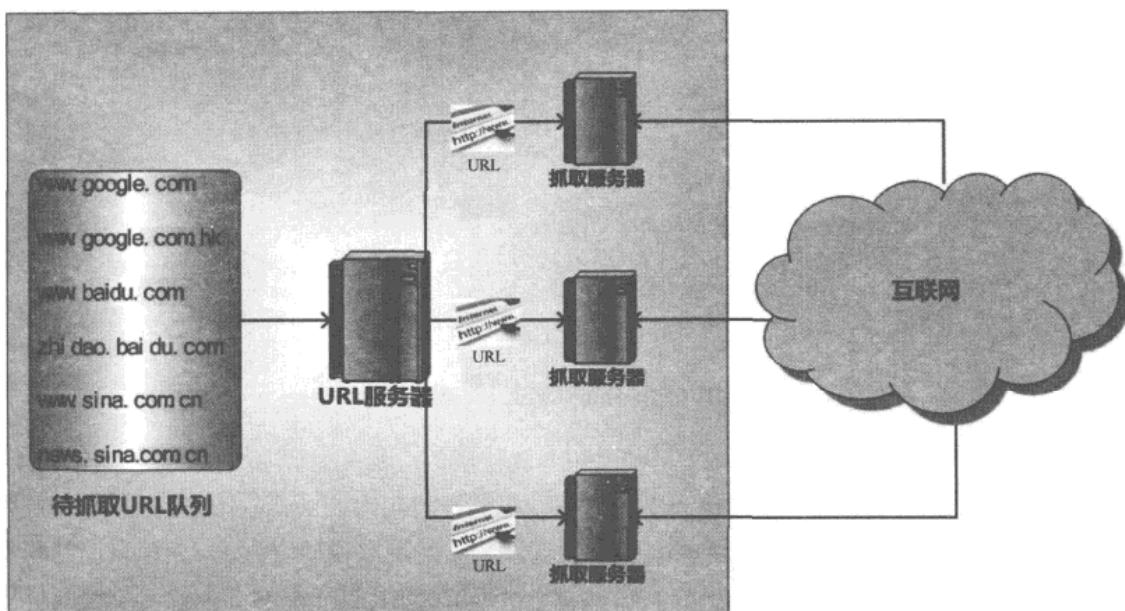


图 2-15 主从式分布爬虫

Google 在早期即采用此种主从式分布爬虫，在这种架构中，因为 URL 服务器承担很多管理任务，同时待抓取 URL 队列数量巨大，所以 URL 服务器容易成为整个系统的瓶颈。

### 2.7.2 对等式分布爬虫 (Peer to Peer)

在对等式分布爬虫体系中，服务器之间不存在分工差异，每台服务器承担相同的功能，各自负担一部分 URL 的抓取工作，如图 2-16 所示即是其中一种对等式分布爬虫，Mercator 爬虫采用此种体系结构（关于 Mercator 爬虫具体技术细节可以参考文献 7）。

由于没有 URL 服务器存在，每台抓取服务器的任务分工就成为问题。在如图 2-16 所示体系结构下，由服务器自己来判断某个 URL 是否应该由自己来抓取，或者将这个 URL 传递给相

应的服务器。至于采取的判断方法，则是对网址的主域名进行哈希计算，之后取模（即  $\text{hash}[\text{域名}] \% m$ ，这里的  $m$  对应服务器个数），如果计算所得的值和抓取服务器编号匹配，则自己下载该网页，否则将该网址转发给对应编号的抓取服务器。

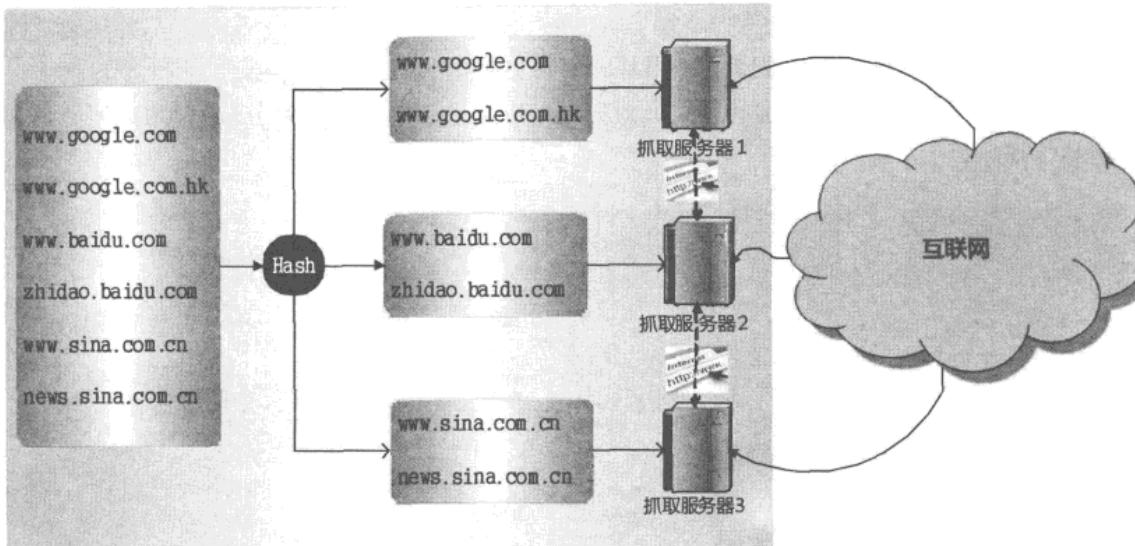


图 2-16 对等分布式爬虫（哈希取模）

以图 2-16 的例子来说，因为有 3 台抓取服务器，所以取模的时候  $m$  设定为 3。图中的 1 号抓取服务器负责抓取哈希取模后值为 1 的网页，当其接收到网址 `www.google.com` 时，首先利用哈希函数计算这个主域名的哈希值，之后对 3 取模，发现取模后值为 1，属于自己的职责范围，于是就自己下载网页；如果接收到网址 `www.baidu.com`，哈希后对 3 取模，发现其值等于 2，不属于自己的职责范畴，则会将这个要下载的 URL 转发给 2 号抓取服务器，由 2 号抓取服务器来进行下载。通过这种方式，每台服务器平均承担大约  $1/3$  的抓取工作量。

由于没有 URL 分发服务器，所以此种方法不存在系统瓶颈问题，另外其哈希函数不是针对整个 URL，而只针对主域名，所以可以保证同一网站的网页都由同一台服务器抓取，这样一方面可以提高下载效率（DNS 域名解析可以缓存），另外一方面也可以主动控制对某个网站的访问速度，避免对某个网站访问压力过大。

图 2-16 这种体系结构也存在一些缺点，假设在抓取过程中某台服务器宕机，或者此时新加入一台抓取服务器，因为取模时  $m$  是以服务器个数确定的，所以此时  $m$  值发生变化，导致大部分 URL 哈希取模后的值跟着变化，这意味着几乎所有任务都需要重新进行分配，无疑会导致资源的极大浪费。

为了解决哈希取模的对等式分布爬虫存在的问题，UbiCrawler 爬虫提出了改进方案，即放



弃哈希取模方式，转而采用一致性哈希方法（Consistent Hash）来确定服务器的任务分工（参考图 2-17）。

一致性哈希将网站的主域名进行哈希，映射为一个范围在 0 到  $2^{32}$  之间的某个数值，大量的网站主域名会被均匀地哈希到这个数值区间。可以如图 2-17 所示那样，将哈希值范围首尾相接，即认为数值 0 和最大值重合，这样可以将其看做有序的环状序列，从数值 0 开始，沿着环的顺时针方向，哈希值逐渐增大，直到环的结尾。而某个抓取服务器则负责这个环状序列的一个片段，即落在某个哈希取值范围内的 URL 都由该服务器负责下载。这样即可确定每台服务器的职责范围。

我们以图 2-17 为例说明其优势，假设 2 号抓取服务器接收到了域名 www.baidu.com，经过哈希值计算后，2 号服务器知道在自己的管辖范围内，于是自己下载这个 URL。在此之后，2 号服务器收到了 www.sina.com.cn 这个域名，经过哈希计算，可知是 3 号服务器负责的范围，于是将这个 URL 转发给 3 号服务器。如果 3 号服务器死机，那么 2 号服务器得不到回应，于是知道 3 号服务器出了状况，此时顺时针按照环的大小顺序查找，将 URL 转发给第一个碰到的服务器，即 1 号服务器，此后 3 号服务器的下载任务都由 1 号服务器接管，直到 3 号服务器重新启动为止。

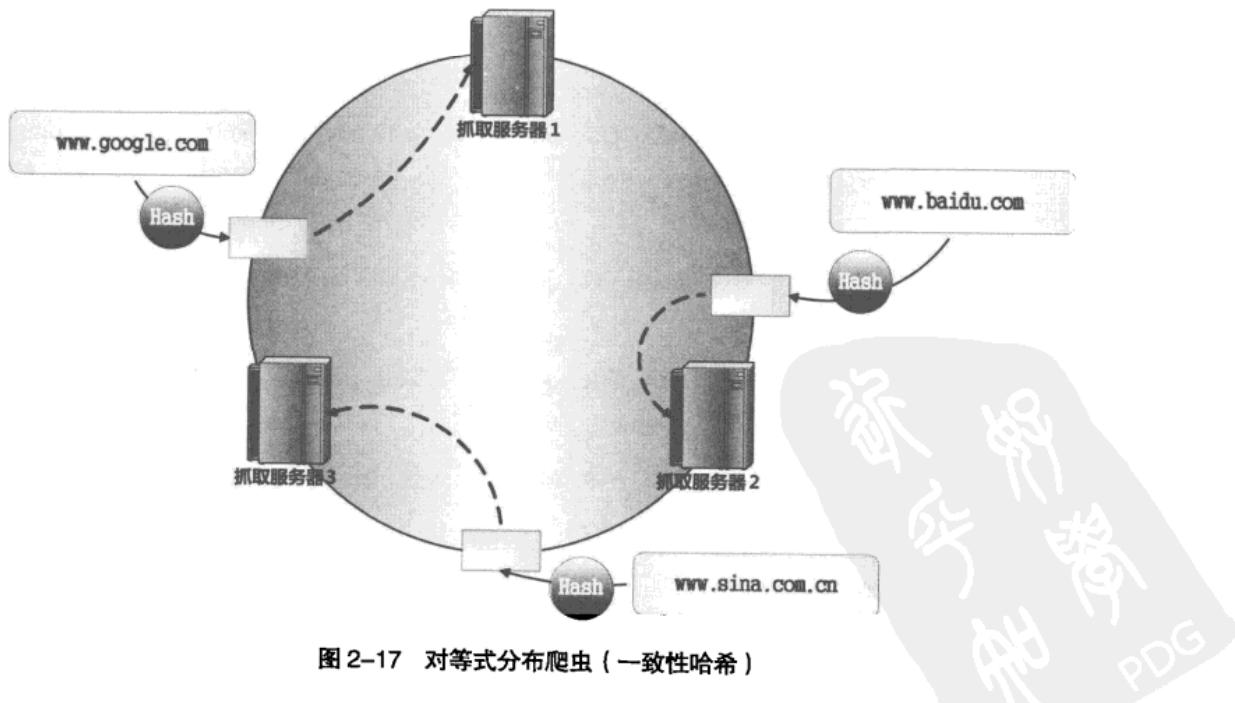


图 2-17 对等式分布爬虫（一致性哈希）

从上面的流程可知，即使某台服务器出了问题，那么本来应该由这台服务器负责的 URL，则由顺时针的下一个服务器接管，并不会对其他服务器的任务造成影响，这样就解决了哈希取

模方式的弊端，将影响范围从全局限制到了局部，如果新加入一台下载服务器也是如此。

## 本章提要

- 从爬虫设计角度讲，优秀的爬虫应该具备高性能，好的可扩展性、健壮性和友好性。
- 从用户体验角度考虑，对爬虫的工作效果评价标准包括：抓取网页覆盖率、抓取网页时新性和抓取网页重要性。
- 抓取策略、网页更新策略、暗网抓取和分布式策略是爬虫系统至关重要的 4 个方面内容，基本决定了爬虫系统的质量和性能。

## 本章参考文献

- [1] Baeza-Yates, R., Castillo, C., Marin, M., and Rodriguez, A. (2005). Crawling a country: better strategies than breadth-first for web page ordering. In Special Interest Tracks and Posters of the 14th International World Wide Web Conference.
- [2] Cho, J. and Garcia-Molina, H. (2000). The evolution of the web and implications for an incremental crawler. In Proceedings of the 26th International Conference on Very Large Data Bases, 200–209.
- [3] Pandey, S. and Olston, C. (2005). User-centric web crawling. In WWW '05: Proceedings of the 14<sup>th</sup> international conference on World Wide Web. ACM, New York, NY, USA, 401–411.
- [4] Boldi, P., Codenotti, B., Santini, M., and Vigna, S.(2002). Ubicrawler: A scalable fully distributed web crawler.
- [5] Cho, J. and Garcia, H. (2003). Effective page refresh policies for Web crawlers. ACM Transactions on Database Systems 28, 4, 390–426.
- [6] Cho, J. and Garcia, H. (2002). Parallel crawlers. In Proc. of the 11<sup>th</sup> International World-Wide Web conference.
- [7] Allan, H. and Marc, N. (1999). Mercator: A scalable, extensible web crawler. World Wide Web, 2(4):219–229.
- [8] Marc, N. and Janet, L. (2001). Wiener. breadth-first crawling yields high-quality pages. In Proceedings of the 10th International World Wide Web Conference, Hong Kong, May 2001. Elsevier Science, 114–118,



- [9] Madhavan J., Ko, D., Kot, L., Ganapathy, V., Rasmussen, A., and Halevy, A.(2008). Google's deep-web crawl. PVLDB, 1(2):1241–1252.
- [10] Tan, Q., Zhuang, Z., Mitra, P., and Giles, C.(2007). Efficiently detecting webpage updates using samples. In ICWE '07: Proceedings of International Conference of Web Engineering. 285–300.



# 第3章 搜索引擎索引

“吾有三剑，唯子所择；皆不能杀人，且先言其状。一曰含光，视之不可见，运之不知有。其所触也，混然无际，经物而物不觉。二曰承影，将旦昧爽之交，日夕昏明之际，北面而察之，淡淡焉若有物存，莫识其状。其所触也，窃窃然有声，经物而物不疾也。三曰宵练，方昼则见影而不见光，方夜见光而不见形。其触物也，然而过，随过随合，觉疾而不血刃焉。此三宝者，传之十三世矣，而无施于事。匣而藏之，未尝启封。”

《列子·汤问》

索引其实在日常生活中是很常见的，比如书籍的目录就是一种索引结构，目的是为了让人们能够更快地找到相关章节内容。再比如像 hao123 这种类型的导航网站本质上也是互联网页面中的索引结构，目的类似，也是为了让用户能够尽快找到有价值的分类网站。

在计算机科学领域，索引也是非常常用的数据结构。其根本目的是为了在具体应用中加快查找速度。比如在数据库中，在很多高效数据结构中，都会大量采用索引来提升系统效率。

具体到搜索引擎，索引更是其中最重要的核心技术之一，面对海量的网页内容，如何快速找到包含用户查询词的所有网页？倒排索引在其中扮演了关键的角色。本章主要讲解与倒排索引相关的技术。

## 3.1 索引基础

本节通过引入简单实例，介绍与搜索引擎索引有关的一些基本概念，了解这些基本概念对于后续深入了解索引的工作机制非常重要。



### 3.1.1 单词—文档矩阵

单词—文档矩阵是表达两者之间所具有的一种包含关系的概念模型，图 3-1 展示了其含义。图 3-1 的每列代表一个文档，每行代表一个单词，打对钩的位置代表包含关系。

		文档1	文档2	文档3	文档4	文档5	文档6
		词汇1	✓			✓	
词汇2			✓	✓			
词汇3					✓		
词汇4		✓				✓	
词汇5			✓				
词汇6				✓			

图 3-1 单词—文档矩阵

从纵向即文档这个维度来看，每列代表文档包含了哪些单词，比如文档 1 包含了词汇 1 和词汇 4，而不包含其他单词。从横向即单词这个维度来看，每行代表了哪些文档包含了某个单词。比如对于词汇 1 来说，文档 1 和文档 4 中出现过词汇 1，而其他文档不包含词汇 1。矩阵中其他的行列也可做此种解读。

搜索引擎的索引其实就是实现单词—文档矩阵的具体数据结构。可以有不同的方式来实现上述概念模型，比如倒排索引、签名文件、后缀树等方式。但是各项实验数据表明，倒排索引是单词到文档映射关系的最佳实现方式，所以本章主要介绍倒排索引的技术细节。

### 3.1.2 倒排索引基本概念

在本小节，我们会解释倒排索引常用到的一些专用术语，为了表达的便捷性，在本书后续章节会直接使用这些术语。

- **文档 ( Document )**: 一般搜索引擎的处理对象是互联网网页，而文档这个概念要更宽泛些，代表以文本形式存在的存储对象。相比网页来说，涵盖更多形式，比如 Word、PDF、html、XML 等不同格式的文件都可以称为文档，再比如一封邮件、一条短信、一条微博也可以称为文档。在本书后续内容中，很多情况下会使用文档来表示文本信息。
- **文档集合 ( Document Collection )**: 由若干文档构成的集合称为文档集合。比如海量的互联网网页或者说大量的电子邮件，都是文档集合的具体例子。

- **文档编号 ( Document ID )**: 在搜索引擎内部，会为文档集合内每个文档赋予一个唯一的内部编号，以此编号来作为这个文档的唯一标识，这样方便内部处理。每个文档的内部编号即称为文档编号，后文有时会用 DocID 来便捷地代表文档编号。
- **单词编号 ( Word ID )**: 与文档编号类似，搜索引擎内部以唯一的编号来表征某个单词，单词编号可以作为某个单词的唯一表征。
- **倒排索引 ( Inverted Index )**: 倒排索引是实现单词—文档矩阵的一种具体存储形式。通过倒排索引，可以根据单词快速获取包含这个单词的文档列表。倒排索引主要由两个部分组成：单词词典和倒排文件。
- **单词词典 ( Lexicon )**: 搜索引擎通常的索引单位是单词，单词词典是由文档集合中出现过的所有单词构成的字符串集合，单词词典内每条索引项记载单词本身的一些信息及指向倒排列表的指针。
- **倒排列表 ( PostingList )**: 倒排列表记载了出现过某个单词的所有文档的文档列表及单词在该文档中出现的位置信息，每条记录称为一个倒排项 ( Posting )。根据倒排列表，即可获知哪些文档包含某个单词。
- **倒排文件 ( Inverted File )**: 所有单词的倒排列表往往顺序地存储在磁盘的某个文件里，这个文件即被称为倒排文件，倒排文件是存储倒排索引的物理文件。

关于这些概念之间的关系，通过图 3-2 可以比较清晰地看出来。

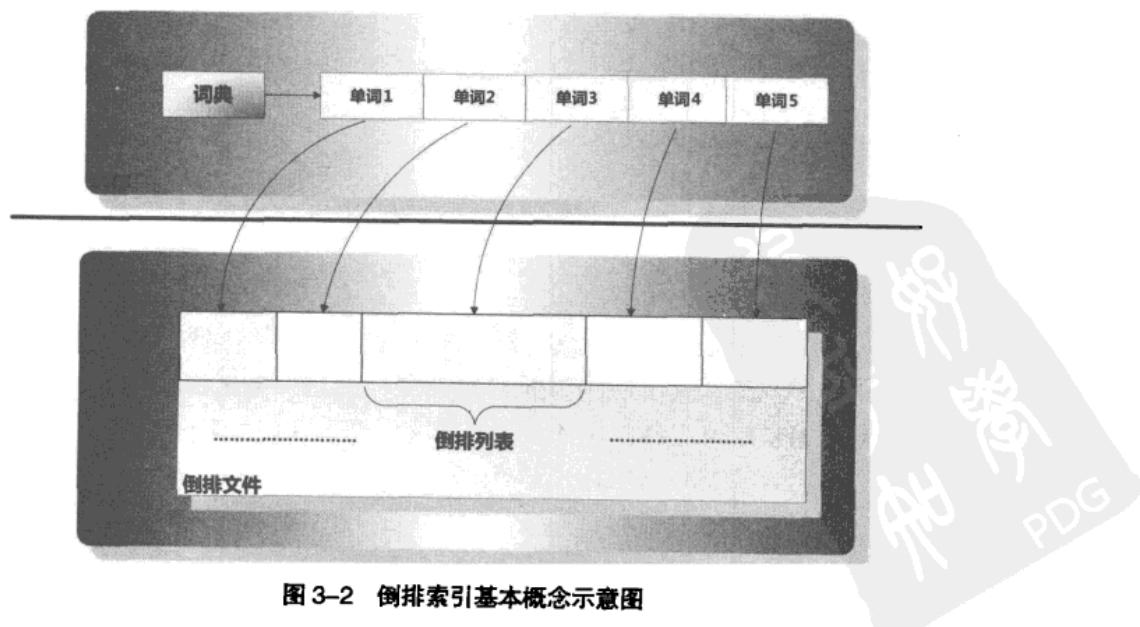


图 3-2 倒排索引基本概念示意图



### 3.1.3 倒排索引简单实例

倒排索引从逻辑结构和基本思路上讲非常简单。下面我们通过具体实例来进行说明，使读者能够对倒排索引有一个宏观而直接的感受。

假设文档集合包含 5 个文档，每个文档内容如图 3-3 所示，在图中最左端一栏是每个文档对应的文档编号，我们的任务就是对这个文档集合建立倒排索引。

文档编号	文档内容
1	谷歌地图之父跳槽Facebook
2	谷歌地图之父加盟Facebook
3	谷歌地图创始人拉斯离开谷歌加盟Facebook
4	谷歌地图之父跳槽Facebook 与 Wave项目取消有关
5	谷歌地图之父拉斯加盟社交网站 Facebook

图 3-3 文档集合

中文和英文等语言不同，单词之间没有明确的分隔符号，所以首先要用分词系统将文档自动切分成单词序列，这样每个文档就转换为由单词序列构成的数据流。为了系统后续处理方便，需要对每个不同的单词赋予唯一的单词编号，同时记录下哪些文档包含这个单词，在如此处理结束后，我们可以得到最简单的倒排索引（参考图 3-4）。在图 3-4 中，“单词 ID”一列记录了每个单词的单词编号，第 2 列是对应的单词，第 3 列即每个单词对应的倒排列表。比如单词“谷歌”，其单词编号为 1，倒排列表为 {1,2,3,4,5}，说明文档集合中每个文档都包含了这个单词。

之所以说图 3-4 所示的倒排索引是最简单的，是因为这个索引系统只记载了哪些文档包含某个单词，而事实上，索引系统还可以记录除此之外的更多信息。图 3-5 是一个相对复杂些的倒排索引，与图 3-4 所示的基本索引系统相比，在单词对应的倒排列表中不仅记录了文档编号，还记载了单词频率信息 (TF)，即这个单词在某个文档中出现的次数，之所以要记录这个信息，是因为词频信息在搜索结果排序时，计算查询和文档相似度是一个很重要的计算因子，所以将其记录在倒排列表中，以方便后续排序时进行分值计算。在图 3-5 所示的例子里，单词“创始人”的单词编号为 7，对应的倒排列表内容有 (3; 1)，其中 3 代表文档编号为 3 的文档包含这个单词，数字 1 代表词频信息，即这个单词在 3 号文档中只出现过 1 次，其他单词对应的倒

排列表所代表的含义与此相同。




单词ID	单词	倒排列表 ( DocID )
1	谷歌	1,2,3,4,5
2	地图	1,2,3,4,5
3	之父	1,2,4,5
4	跳槽	1,4
5	Facebook	1,2,3,4,5
6	加盟	2,3,5
7	创始人	3
8	拉斯	3,5
9	离开	3
10	与	4
11	Wave	4
12	项目	4
13	取消	4
14	有关	4
15	社交	5
16	网站	5

图 3-4 最简单的倒排序引




单词ID	单词	倒排列表 ( DocID;TF )
1	谷歌	(1;1),(2;1),(3;2),(4;1),(5;1)
2	地图	(1;1),(2;1),(3;1),(4;1),(5;1)
3	之父	(1;1),(2;1),(4;1),(5;1)
4	跳槽	(1;1),(4;1)
5	Facebook	(1;1),(2;1),(3;1),(4;1),(5;1)
6	加盟	(2;1),(3;1),(5;1)
7	创始人	(3;1)
8	拉斯	(3;1),(5;1)
9	离开	(3;1)
10	与	(4;1)
11	Wave	(4;1)
12	项目	(4;1)
13	取消	(4;1)
14	有关	(4;1)
15	社交	(5;1)
16	网站	(5;1)

图 3-5 带有单词频率信息的倒排序引

实用的倒排序引还可以记载更多的信息，如图 3-6 所示的索引系统除了记录文档编号和单词频率信息外，额外记载了两类信息，即每个单词对应的文档频率信息（对应图 3-6 的第 3 列）



及单词在某个文档出现位置的信息。



单词ID	单词	文档频率	倒排列表 (DocID;TF;<POS>)
1	谷歌	5	(1;1;<1>),(2;1;<1>),(3;2;<1;6>),(4;1;<1>),(5;1;<1>)
2	地图	5	(1;1;<2>),(2;1;<2>),(3;1;<2>),(4;1;<2>),(5;1;<2>)
3	之父	4	(1;1;<3>),(2;1;<3>),(4;1;<3>),(5;1;<3>)
4	跳槽	2	(1;1;<4>),(4;1;<4>)
5	Facebook	5	(1;1;<5>),(2;1;<5>),(3;1;<8>),(4;1;<5>),(5;1;<8>)
6	加盟	3	(2;1;<4>),(3;1;<7>),(5;1;<5>)
7	创始人	1	(3;1;<3>)
8	拉斯	2	(3;1;<4>),(5;1;<4>)
9	离开	1	(3;1;<5>)
10	与	1	(4;1;<6>)
11	Wave	1	(4;1;<7>)
12	项目	1	(4;1;<8>)
13	取消	1	(4;1;<9>)
14	有关	1	(4;1;<10>)
15	社交	1	(5;1;<6>)
16	网站	1	(5;1;<7>)

图 3-6 带有单词频率、文档频率和出现位置信息的倒排索引

文档频率信息代表了在文档集合中有多少个文档包含某个单词，之所以要记录这个信息，其原因与单词频率信息一样，这个信息在搜索结果排序计算中是一个非常重要的因子。而单词在某个文档中出现位置的信息并非索引系统一定要记录的，在实际的索引系统里可以包含，也可以选择不包含这个信息，之所以如此是因为这个信息对于搜索系统来说并非必需，位置信息只有在支持短语查询的时候才能够派上用场。

以单词“拉斯”为例，其单词编号为 8，文档频率为 2，代表整个文档集合中有两个文档包含这个单词，对应的倒排列表为  $\{(3;1;<4>),(5;1;<4>)\}$ ，其含义为在文档 3 和文档 5 出现过这个单词，单词频率都为 1，单词“拉斯”在两个文档中的出现位置都是 4，即文档中第 4 个单词是“拉斯”。

如图 3-6 所示的倒排索引已经是一个非常完备的索引系统，实际搜索系统的索引结构基本如此，区别无非是采取哪些具体的数据结构来实现上述逻辑结构。

有了这个索引系统，搜索引擎可以很方便地响应用户的查询，比如用户输入查询词“Facebook”，搜索系统查找倒排索引，从中可以读出包含这个单词的文档，这些文档就是提供给用户的搜索结果，而利用单词频率信息、文档频率信息即可对这些候选搜索结果进行排序，计算文档和查询的相似性，按照相似性得分由高到低排序输出，此即为搜索系统的部分内部流

程，具体实现方案本书第5章会做详细描述。

## 3.2 单词词典

单词词典是倒排索引中非常重要的组成部分，它用来维护文档集合中出现过的所有单词的相关信息，同时用来记载某个单词对应的倒排列表在倒排文件中的位置信息。在支持搜索时，根据用户的查询词，去单词词典里查询，就能够获得相应的倒排列表，并以此作为后续排序的基础。

对于一个规模很大的文档集合来说，可能包含几十万甚至上百万的不同单词，能否快速定位某个单词，这直接影响搜索时的响应速度，所以需要高效的数据结构来对单词词典进行构建和查找，常用的数据结构包括哈希加链表结构和树形词典结构。

### 3.2.1 哈希加链表

图3-7是这种词典结构的示意图。这种词典结构主要由两个部分构成，主体部分是哈希表，每个哈希表项保存一个指针，指向冲突链表，在冲突链表里，相同哈希值的单词形成链表结构。之所以会有冲突链表，是因为两个不同单词获得相同的哈希值，如果是这样，在哈希方法里被称做是一次冲突，可以将相同哈希值的单词存储在链表里，以供后续查找。

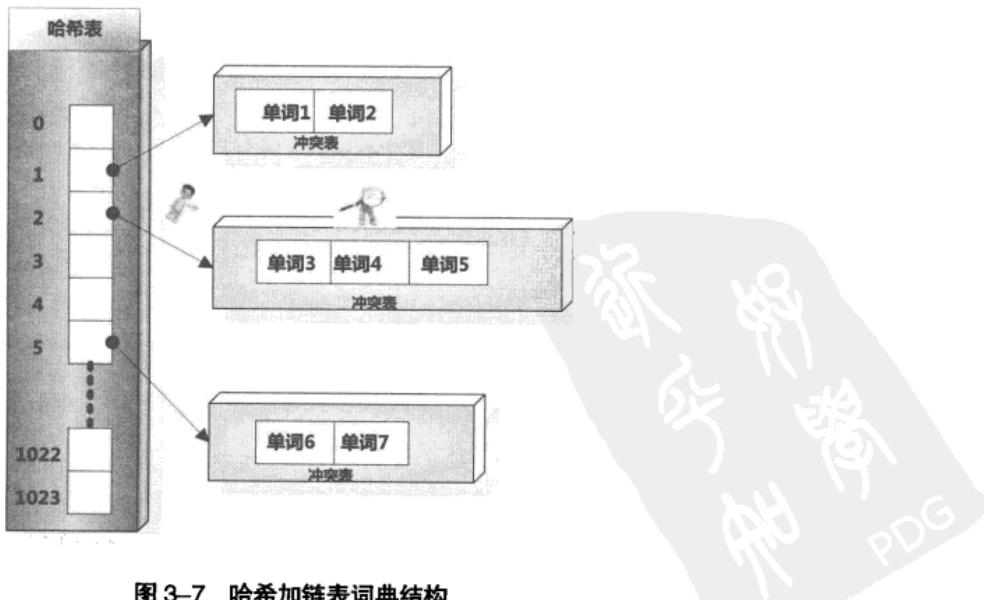


图3-7 哈希加链表词典结构

在建立索引的过程中，词典结构也会相应地被构建出来。比如在解析一个新文档的时候，



对于某个在文档中出现的单词 T，首先利用哈希函数获得其哈希值，之后根据哈希值对应的哈希表项读取其中保存的指针，就找到了对应的冲突链表。如果冲突链表里已经存在这个单词，说明单词在之前解析的文档里已经出现过。如果在冲突链表里没有发现这个单词，说明该单词是首次碰到，则将其加入冲突链表里。通过这种方式，当文档集合内所有文档解析完毕时，相应的词典结构也就建立起来了。

在响应用户查询请求时，其过程与建立词典类似，不同点在于即使词典里没出现过某个单词，也不会添加到词典内。以图 3-7 为例，假设用户输入的查询请求为单词 3，对这个单词进行哈希，定位到哈希表内的 2 号槽，从其保留的指针可以获得冲突链表，依次将单词 3 和冲突链表内的单词比较，发现单词 3 在冲突链表内，于是找到这个单词，之后可以读出这个单词对应的倒排列表来进行后续的工作，如果没有找到这个单词，说明文档集合内没有任何文档包含单词，则搜索结果为空。

### 3.2.2 树形结构

B 树（或者 B+树）是另外一种高效查找结构，图 3-8 是一个 B 树结构示意图。B 树与哈希方式查找不同，需要字典项能够按照大小排序（数字或者字符序），而哈希方式则无须数据满足此项要求。

B 树形成了层级查找结构，中间节点用于指出一定顺序范围的词典项目存储在哪个子树中，起到根据词典项比较大小进行导航的作用，最底层的叶子节点存储单词的地址信息，根据这个地址就可以提取出单词字符串。

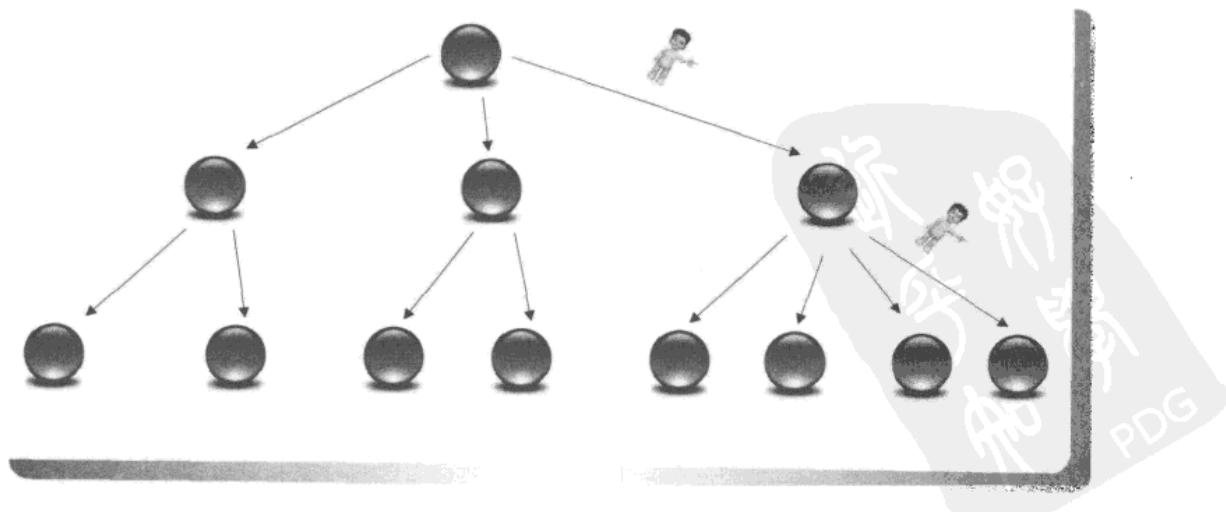


图 3-8 B 树查找结构

### 3.3 倒排列表（Posting List）

在本章第一小节介绍的简单索引例子中，大致可以看到倒排列表起到的作用，倒排列表用来记录有哪些文档包含了某个单词。一般在文档集合里会有很多文档包含某个单词，每个文档会记录文档编号（DocID），单词在这个文档中出现的次数（TF）及单词在文档中哪些位置出现过等信息，这样与一个文档相关的信息被称做倒排索引项（Posting），包含这个单词的一系列倒排索引项形成了列表结构，这就是某个单词对应的倒排列表。图 3-9 是倒排列表的示意图，在文档集合中出现过的所有单词及其对应的倒排列表组成了倒排索引。

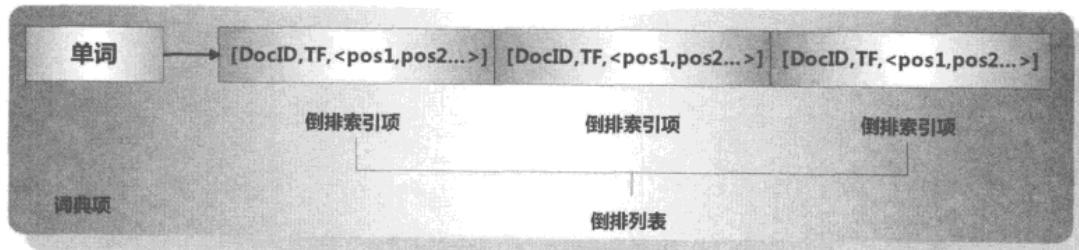


图 3-9 倒排列表示意图

在实际的搜索引擎系统中，并不存储倒排索引项中的实际文档编号，而是代之以文档编号差值（D-Gap）。文档编号差值是倒排列表中相邻的两个倒排索引项文档编号的差值，一般在索引构建过程中，可以保证倒排列表中后面出现的文档编号大于之前出现的文档编号，所以文档编号差值总是大于 0 的整数。如图 3-10 所示的例子中，原始的 3 个文档编号分别是 187、196 和 199，通过编号差值计算，在实际存储的时候就转化成了：187、9、3。

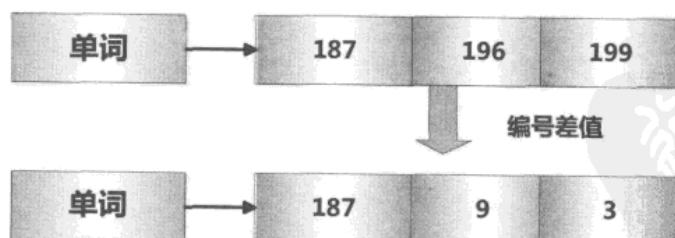


图 3-10 文档编号差值示例

之所以要对文档编号进行差值计算，主要原因是为了更好地对数据进行压缩，原始文档编号一般都是大数值，通过差值计算，就有效地将大数值转换为了小数值，而这有助于增加数据的压缩率。至于为何这样，本书第 4 章在专门讲述索引压缩时会叙述其原因。



## 3.4 建立索引

正如前面章节所述，索引结构如果建立好了，可以提高搜索的速度，那么给定一个文档集合，索引是如何建立起来的呢？建立索引的方式有很多种，本节叙述比较实用的3种建立索引的方法。

### 3.4.1 两遍文档遍历法（2-Pass In-Memory Inversion）

顾名思义，此方法需要对文档集合进行两遍扫描，图3-11是这种方法的示意图。值得注意的一点是：此方法完全是在内存里完成索引的创建过程的，而另外两种方法则是通过内存和磁盘相互配合来完成索引建立任务的。

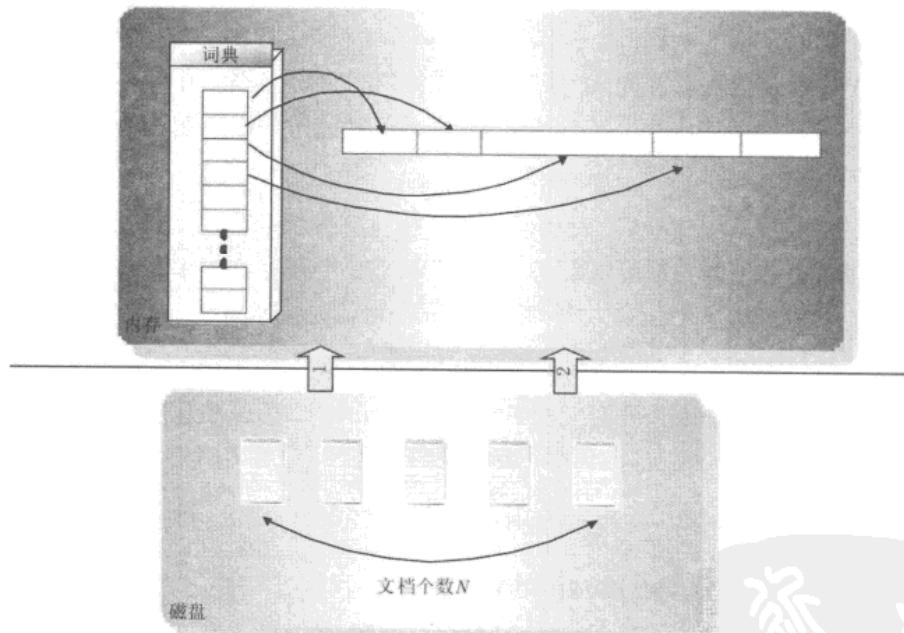


图3-11 两遍文档遍历法

#### 第一遍文档遍历

在第一遍扫描文档集合时，该方法并没有立即开始建立索引，而是收集一些全局的统计信息。比如文档集合包含的文档个数  $N$ ，文档集合内所包含的不同单词个数  $M$ ，每个单词在多少个文档中出现过的信息  $DF$ 。将所有单词对应的  $DF$  值全部相加，就可以知道建立最终索引所需的内存大小是多少，因为一个单词对应的  $DF$  值如果是 10，说明有 10 个文档包含这个单词，那么这个单词对应的倒排列表应该包含 10 项内容，每一项记载某个文档的文档 ID 和单词在该

文档对应的出现次数 TF。

在获得了上述 3 类信息后，就可以知道最终索引的大小，于是在内存中分配足够大的空间，用来存储倒排索引内容。如图 3-11 所示，在内存中可以开辟连续存储区域，因为第一遍扫描已经获得了每个单词的 DF 信息，所以将连续存储区划分成不同大小的片段，词典内某个单词根据自己对应的 DF 信息，可以通过指针，指向属于自己的内存片段的起始位置和终止位置，将来在第二遍扫描时，这个单词对应的倒排列表信息会被填充进这个片段中。

综上所述，第一遍扫描的主要目的是获得一些统计信息，并根据统计信息分配内存等资源，同时建立好单词相对应倒排列表在内存中的位置信息，即主要做些资源准备工作。

## 第二遍文档遍历

在第二遍扫描的时候，开始真正建立每个单词的倒排列表信息，即对某个单词来说，获得包含这个单词的每个文档的文档 ID，以及这个单词在文档中的出现次数 TF，这样就可以不断填充第一遍扫描所分配的内存空间。当第二遍扫描结束的时候，分配的内存空间正好被填充满，而每个单词用指针所指向的内存区域“片段”，其起始位置和终止位置之间的数据就是这个单词对应的倒排列表。

经过两遍扫描完成索引建立后，即可将内存的倒排列表和词典信息写入磁盘，这样就完成了建立索引的过程。从上述流程可以看出，索引的构建完全是在内存中完成的，这就要求内存一定要足够大，否则如果文档集合太大，内存未必能够满足需求。

从另外一个角度看，在建立索引的过程中，从磁盘读取文档并解析文档基本是最消耗时间的一个步骤，而两遍扫描法因为要对文档集合进行两遍遍历，所以从速度上不占优势，在实际中采用这种方法的系统并不常见。而下面介绍的两种方法都是对文档集合进行一遍扫描，所以在速度方面明显占优。

### 3.4.2 排序法 (Sort-based Inversion)

两遍遍历法在建立索引的过程中，对内存的消耗要求较高，不同的文档集合包含文档数量大小不同，其所需内存大小是不确定的。当文档集合非常大时，可能因内存不够，导致无法建立索引。排序法对此做出了改进，该方法在建立索引的过程中，始终在内存中分配固定大小的空间，用来存放词典信息和索引的中间结果，当分配的空间被消耗光的时候，把中间结果写入磁盘，清空内存里中间结果所占空间，以用做下一轮存放索引中间结果的存储区。这种方法由于只需要固定大小的内存，所以可以对任意大小的文档集合建立索引（参考图 3-12）。

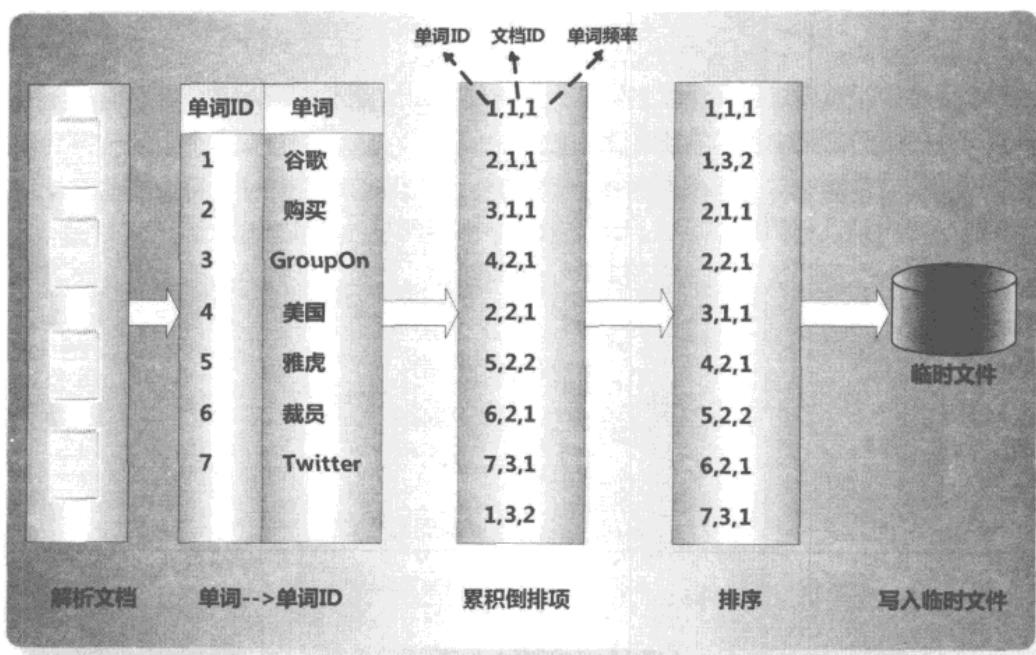


图 3-12 排序法

### 中间结果内存排序

图 3-12 是排序法在内存中建立索引中间结果的示意图。读入文档后，对文档进行编号，赋予唯一的文档 ID，并对文档内容解析。对于文档中出现的单词，通过查词典将单词转换为对应的单词 ID，如果词典中没有这个单词，说明是第一次碰到，则赋予单词以唯一的单词 ID 并插入词典中。在完成了由单词映射为单词 ID 的过程之后，可以对该文档内每个单词建立一个（单词 ID、文档 ID、单词频率）三元组，这个三元组就是单词对应文档的倒排列表项，将这个三元组追加进中间结果存储区末尾。如果文档内的所有单词都经过如此处理，形成三元组序列的形式，则该文档被处理完成，开始依次序处理下一文档，过程与此类似。

随着新的文档不断被处理完成，存储三元组集合的中间结果所占用的内存会越来越大，词典里包含的新单词也越来越多，当分配的内存定额被占满时，该方法对三元组中间结果进行排序。排序的原则是：主键是单词 ID，即首先要按照单词 ID 由小到大排序；次键是文档 ID，即在相同单词 ID 的情况下，按照文档 ID 由小到大排序。通过以上方式，三元组变为有序形式。为了腾出内存空间，将排好序的三元组写入磁盘临时文件中，这样就空出内存来进行后续文档的处理。这里需要注意的是：在建立索引的过程中，词典是一直存储在内存中的，每次清空内存只是将中间结果写入磁盘。随着处理文档的增加，词典占用的内存会逐渐增加，由于分配内存是固定大小，而词典占用内存越来越大，也就是说，越往后，可用来存储三元组的空间是越

越来越少的。

之所以要对中间结果进行排序，主要是为了方便后续的处理。因为每一轮处理都会在磁盘产生一个对应的中间结果文件，当所有文档处理完成后，在磁盘中会有多个中间结果文件，为了产生最终的索引，需要将这些中间结果文件合并。图 3-13 是如何对中间结果进行合并的示意图。

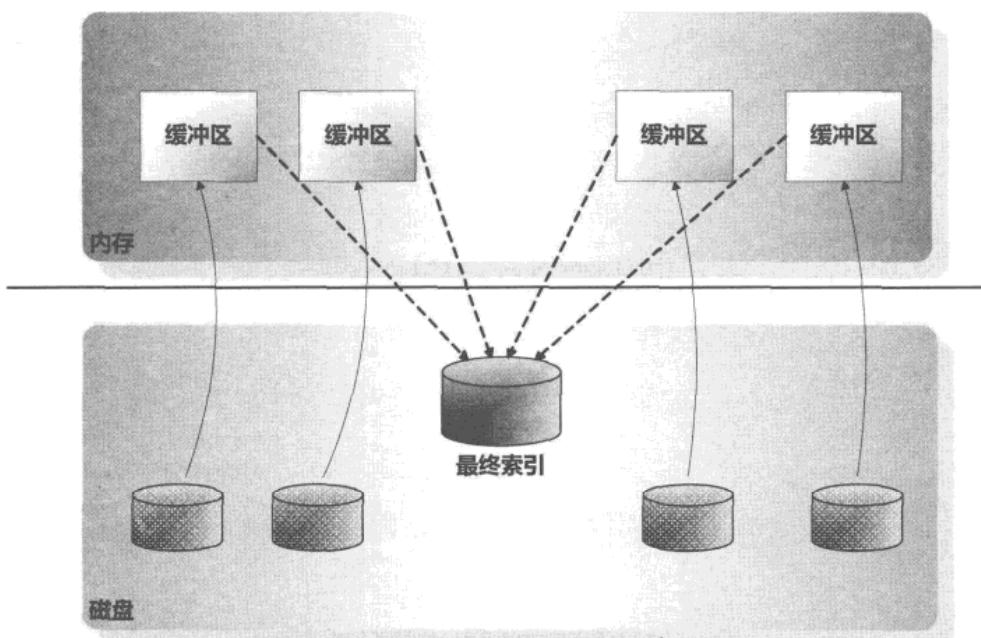


图 3-13 中间结果合并

### 合并中间结果

如图 3-13 所示，在合并中间结果的过程中，系统为每个中间结果文件在内存中开辟一个数据缓冲区，用来存放文件的部分数据。因为在形成中间结果文件前，已经按照单词 ID 和文档 ID 进行了排序，所以进入缓冲区的数据已经是有序的。合并过程中，将不同缓冲区中包含的同一个单词 ID 的三元组进行合并，如果某个单词 ID 的所有三元组全部合并完成，说明这个单词的倒排列表已经构建完成，则将其写入最终索引中，同时将各个缓冲区中对应这个单词 ID 的三元组内容清空，这样缓冲区就可以继续从中间结果文件中读入后续的三元组来进行下一个单词的三元组合并。当所有中间结果文件都依次被读入缓冲区，在合并完成后，就形成了最终的索引文件。

### 3.4.3 归并法 (Merge-based Inversion)

排序法分配固定大小内存来建立索引，所以无论要建立索引的文档集合有多大，都可以通过这种方法完成。但是如上所述，在分配的内存定额被消耗光时，排序法只是将中间结果写入磁盘，而词典信息一直在内存中进行维护，随着处理的文档越来越多，词典里包含的词典项越来越多，所以占用内存越来越大，导致后期中间结果可用内存越来越少。归并法对此做出了改进，即每次将内存中数据写入磁盘时，包括词典在内的所有中间结果信息都被写入磁盘，这样内存所有内容都可以被清空，后续建立索引可以使用全部的定额内存。

图 3-14 是归并法的示意图。其整体流程和排序法大致相同，也是分为两个大的阶段，首先在内存里维护中间结果，当内存占满后，将内存数据写入磁盘临时文件，第二阶段对临时文件进行归并形成最终索引。

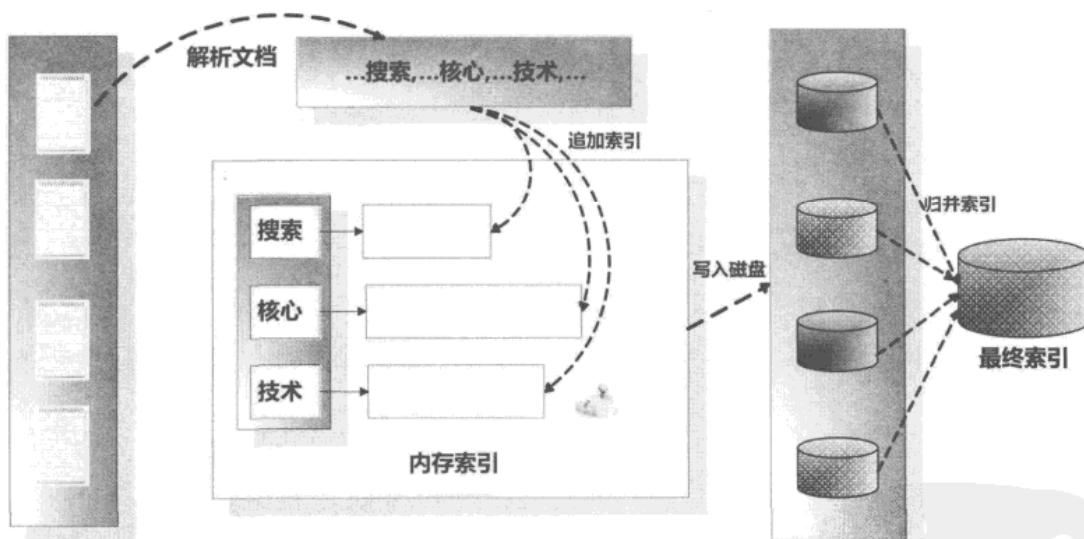


图 3-14 归并法

尽管从整体流程看，和排序法大致相同，但是在具体实现方式上有较大差异。

首先，排序法在内存中存放的是词典信息和三元组数据，在建立索引的过程中，词典和三元组数据并没有直接的联系，词典只是为了将单词映射为单词 ID。而归并法则是在内存中建立一个完整的内存索引结构，相当于对目前处理的文档子集单独在内存中建立起了一整套倒排索引，和最终索引相比，其结构和形式是相同的，区别只是这个索引只是部分文档的索引而非全部文档的索引。

其次，在将中间结果写入磁盘临时文件时，归并法会将整个内存的倒排索引写入临时文件，

对于某个单词的倒排列表在写入磁盘文件时，将词典项放在列表最前端，之后跟随相应的倒排列表，这样依次将单词和对应的倒排列表写入磁盘文件，随后彻底清空所占内存。而排序法如上节所述，只是将三元组数据排序后写入磁盘临时文件，词典作为一个映射表一直存储在内存中。

在最后的临时文件合并为最终索引的过程中，两者也有差异。排序法因为保存的是有序三元组信息，所以在合并时，是对同一单词的三元组依次进行合并；而归并法的临时文件则是每个单词对应的部分倒排列表，所以在合并时针对每个单词的倒排列表进行合并，形成这个单词的最终倒排列表。另外，归并法在最后的合并过程中形成最终的词典信息。

### 3.5 动态索引

如果搜索引擎需要处理的文档集合是静态集合，那么在索引建立好之后，就可以一直用建好的索引响应用户查询请求。但是，在真实环境中，搜索引擎需要处理的文档集合往往是动态集合，即在建好初始的索引后，后续不断有新文档进入系统，同时原先的文档集合内有些文档可能被删除或者内容被更改。典型的例子就是桌面搜索，当新下载一篇文档，那么搜索系统需要及时将其纳入，删除某篇文档也需要在非常短的时间内在搜索结果里体现出来。大多数搜索引擎面临的应用场景是类似的动态环境。

索引系统如何能够做到实时反映这种变化呢？动态索引可以实现这种实时性要求，图 3-15 是动态索引的示意图。在这种动态索引中，有 3 个关键的索引结构：倒排索引、临时索引和已删除文档列表。

倒排索引就是对初始文档集合建立好的索引结构，一般单词词典存储在内存，对应的倒排列表存储在磁盘文件中。临时索引是在内存中实时建立的倒排索引，其结构和前述的倒排索引是一样的，区别在于词典和倒排列表都在内存中存储。当有新文档进入系统时，实时解析文档并将其追加进这个临时索引结构中。已删除文档列表则用来存储已被删除的文档的相应文档 ID，形成一个文档 ID 列表。这里需要注意的是：当一篇文档内容被更改，可以认为是旧文档先被删除，之后向系统内增加一篇新的文档，通过这种间接方式实现对内容更改的支持。

当系统发现有新文档进入时，立即将其加入临时索引中。有文档被删除时，则将其加入删除文档队列。文档被更改时，则将原先文档放入删除队列，解析更改后的文档内容，并将其加入临时索引中。通过这种方式可以满足实时性的要求。

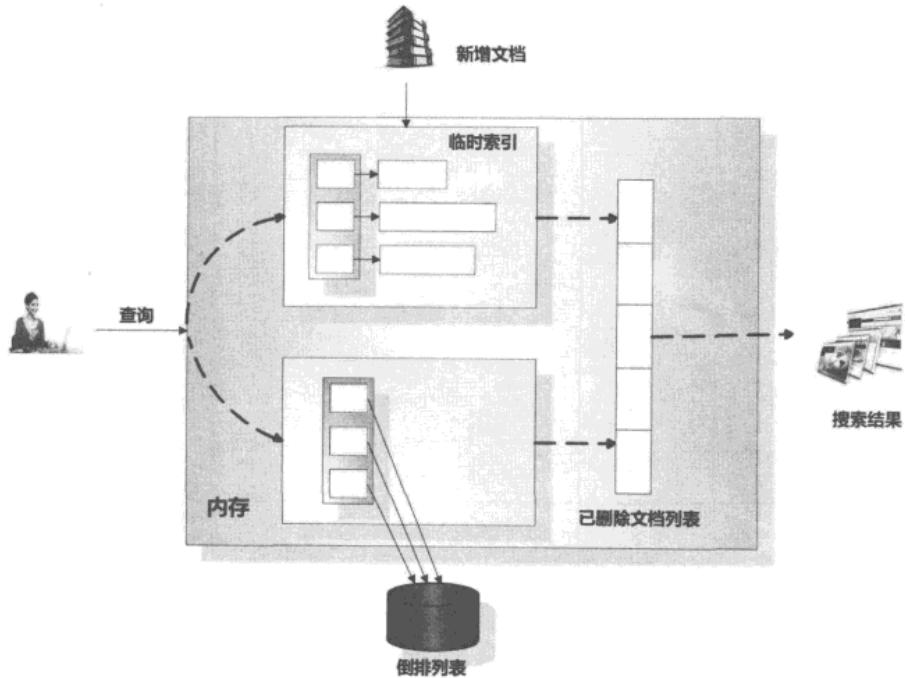


图 3-15 动态索引

如果用户输入查询请求，则搜索引擎同时从倒排索引和临时索引中读取用户查询单词的倒排列表，找到包含用户查询的文档集合，并对两个结果进行合并，之后利用删除文档列表进行过滤，将搜索结果中那些已经被删除的文档从结果中过滤，形成最终的搜索结果，并返回给用户。这样就能够实现动态环境下的准实时搜索功能。

## 3.6 索引更新策略

动态索引通过在内存中维护临时索引，可以实现对动态文档和实时搜索的支持。但是服务器内存总是有限的，随着新加入系统的文档越来越多，临时索引消耗的内存也会随之增加。当最初分配的内存将被使用完时，要考虑将临时索引的内容更新到磁盘索引中，以释放内存空间来容纳后续的新进文档，此时要考虑合理有效的索引更新策略。

常用的索引更新策略有 4 种：完全重建策略、再合并策略、原地更新策略及混合策略。

### 3.6.1 完全重建策略 (Complete Re-Build)

完全重建策略是一个相当直观的方法，当新增文档达到一定数量，将新增文档和原先的老

文档进行合并，然后利用前述章节提到的建立索引的方式，对所有文档重新建立索引。新索引建立完成后，老的索引被遗弃释放，之后对用户查询的响应完全由新的索引负责。图 3-16 是这种策略的说明示意图。

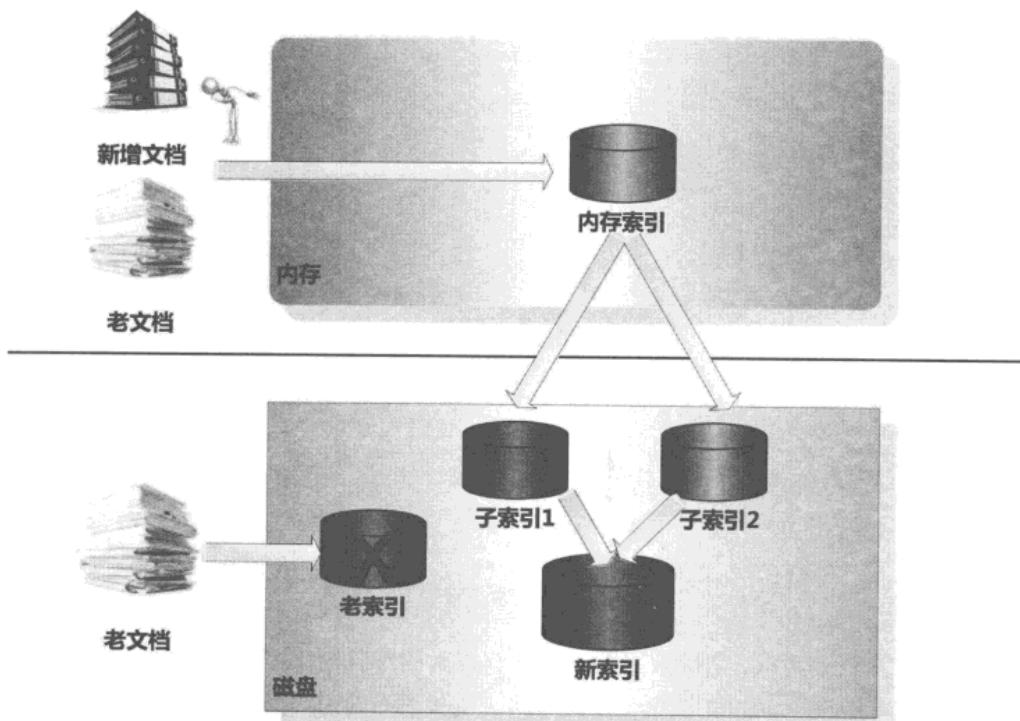


图 3-16 完全重建策略

因为重建索引需要较长时间，在进行索引重建的过程中，内存中仍然需要维护老的索引，来对用户的查询做出响应。只有当新索引完全建立完成后，才能释放旧的索引，将用户查询响应切换到新索引上。

这种重建策略比较适合小文档集合，因为完全重建索引的代价较高，但是目前主流商业搜索引擎一般是采用此方式来维护索引的更新的，这与互联网本身的特性有关。

### 3.6.2 再合并策略 (Re-Merge)

有新增文档进入搜索系统时，搜索系统在内存维护临时倒排索引来记录其信息，当新增文档达到一定数量，或者指定大小的内存被消耗完，则把临时索引和老文档的倒排索引进行合并，以生成新的索引。图 3-17 是这种策略的一种图示。

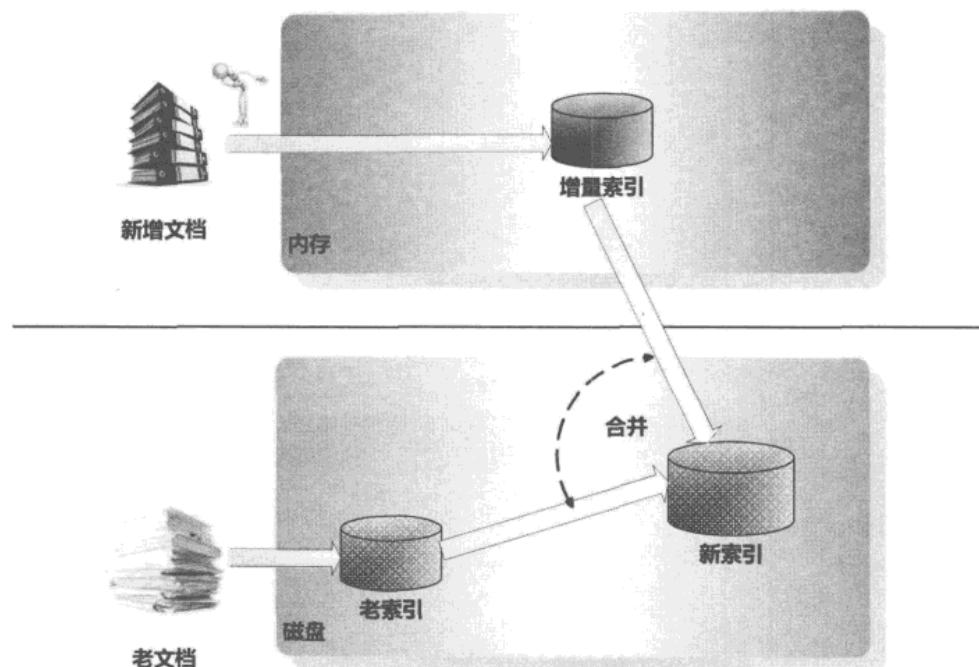


图 3-17 再合并策略

在实际的搜索系统中，再合并策略按照以下步骤进行索引内容的更新。

- 当新增文档进入系统，解析文档，之后更新内存中维护的临时索引，文档中出现的每个单词，在其倒排列表末尾追加倒排列表项，这个临时索引可称为增量索引。
- 一旦增量索引将指定的内存消耗光，此时需要进行一次索引合并，即将增量索引和老的倒排索引内容进行合并，图 3-18 是合并过程示意图。这里需要注意的是：倒排文件里的倒排列表存放顺序已经按照索引单词字典顺序由低到高进行了排序，增量索引在遍历词典的时候也按照字典序由低到高排列，这样对老的倒排文件只需进行一遍扫描，并可顺序读取，减少了文件操作中比较耗时的磁盘寻道时间，可以有效地提高合并效率。

在合并过程中，需要依次遍历增量索引和老索引单词词典中包含的单词及其对应的倒排列表，可以用两个指针分别指向两套索引中目前需要合并的单词（参考图 3-18 中箭头所指），按照如下方式进行倒排列表的合并。

考虑增量索引的单词指针指向的单词，如果这个单词在词典序中小于老索引的单词指针指向的单词，说明这个单词在老索引中未出现过，则直接将这个单词对应的倒排列表写入新索引的倒排文件中，同时增量索引单词指针后移指向下一个单词。

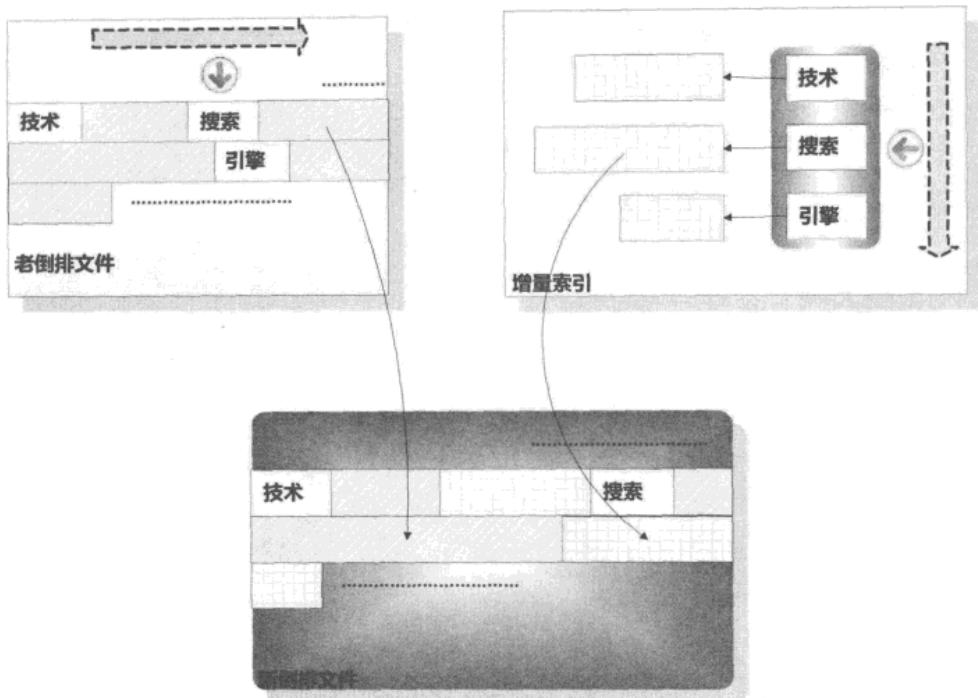


图 3-18 合并增量索引与老的倒排索引内容

如果两个单词指针指向的单词相同，说明这个单词在增量索引和老索引中同时出现，则将老索引中这个单词对应的倒排列表写入新索引的倒排列表，然后把增量索引中这个单词对应的倒排列表追加到其后，这样就完成了这个单词所有倒排列表的合并。图 3-18 中箭头所指单词是搜索，说明此时合并到了该单词，因为在老的索引系统和增量索引中都包含这个单词，所以首先将老索引对应的倒排列表追加到新索引倒排文件末尾，之后将增量索引中搜索这个单词对应的倒排列表追加在其后，这样就完成了这个单词索引项的合并。两个索引的单词指针都移动到下一个单词继续进行合并。

如果某个单词只在老索引中出现过，即发现老索引的单词指针指向的单词，其词典序小于增量索引单词指针指向的单词，则直接将老索引中对应的倒排列表写入新索引倒排文件中。老索引的单词指针后移指向下一个单词，继续进行合并。

当两个索引的所有单词都遍历完成后，新索引建成，此时可以遗弃释放老索引，使用新索引来响应用户查询请求。

同样地，在按照这个策略进行索引合并的过程中，为了能够响应用户查询，在合并索引期间，需要使用老索引响应用户查询请求。

再合并策略是效率非常高的一种索引更新策略，主要原因在于：在对老的倒排索引进行遍



历时，因为已经按照索引单词的词典序由低到高排好顺序，所以可以顺序读取文件内容，减少磁盘寻道时间，这是其高效的根本原因。但是这种方法也有其缺点，因为要生成新的倒排序文件，所以对于老索引中的很多单词来说，尽管其倒排列表并未发生任何变化，但是也需要将其从老索引中读取出来并写入新索引中，这种对磁盘输入输出的消耗是没有太大必要且非常耗时的。

### 3.6.3 原地更新策略 (In-Place)

原地更新策略的基本出发点，可以认为是试图改进再合并策略的缺点。也就是说，在索引更新过程中，如果老索引的倒排列表没有变化，可以不需要读取这些信息，而只对那些倒排列表变化的单词进行处理。甚至希望能更进一步：即使老索引的倒排列表发生变化，是否可以只在其末尾进行追加操作，而不需要读取原先的倒排列表并重写到磁盘另一个位置？如果能够达到这个目标，明显可以大量减少磁盘读/写操作，提升系统执行效率。

为了达到上述目标，原地更新策略在索引合并时，并不生成新的索引文件，而是直接在原先老的索引文件里进行追加操作（参考图 3-19），将增量索引里单词的倒排列表项追加到老索引相应位置的末尾，这样就可达到上述目标，即只更新增量索引里出现的单词相关信息，其他单词相关信息不做变动。

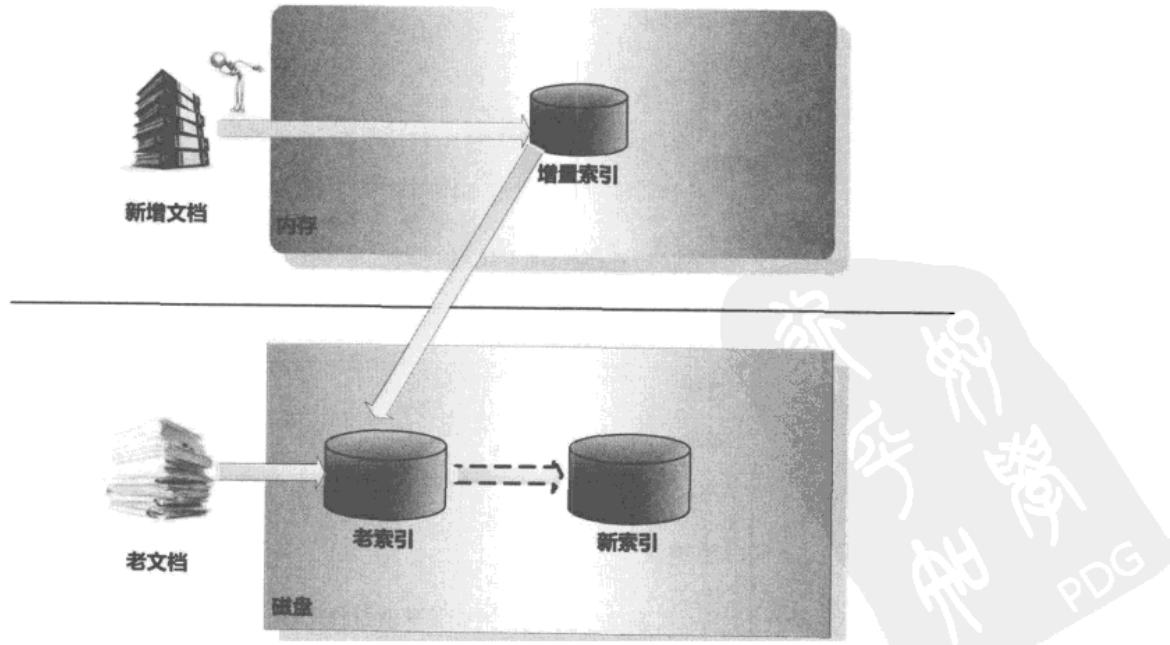


图 3-19 原地更新策略

但是这里存在一个问题：对于倒排文件中的两个相邻单词，为了在查询时加快读取速度，其倒排列表一般是顺序序列存储的，这导致没有空余位置用来追加新信息。为了能够支持追加操作，原地更新策略在初始建立的索引中，会在每个单词的倒排列表末尾预留出一定的磁盘空间，这样，在进行索引合并时，可以将增量索引追加到预留空间中。

图 3-20 是这种合并策略的一个说明。在图中，老索引中每个单词的倒排列表末尾都预留出空余磁盘空间，以作为信息追加时的存储区域。在对新增索引进行合并时，按照词典序，依次遍历新增索引中包含的单词，并对新增倒排列表的大小和老索引中相应预留空间大小进行比较，如果预留空间足够大，则将新增列表追加到老索引即可，如果预留空间不足以容纳新增倒排列表，那么此时需要在磁盘中找到一块完整的连续存储区，这个存储区足以容纳这个单词的倒排列表，之后将老索引中的倒排列表读出并写入新的磁盘位置，并将增量索引对应的倒排列表追加到其后，这样就完成了一次倒排列表的“迁移”工作。

在如图 3-20 所示的例子中，可以看出，单词“技术”和“引擎”在老索引中的预留空间足够大，所以对增量索引只需做追加写入即可，但是对于单词“搜索”来说，其预留空间不足以容纳新增倒排列表，所以这个单词的倒排列表需要迁移到磁盘另外一个连续存储区中。

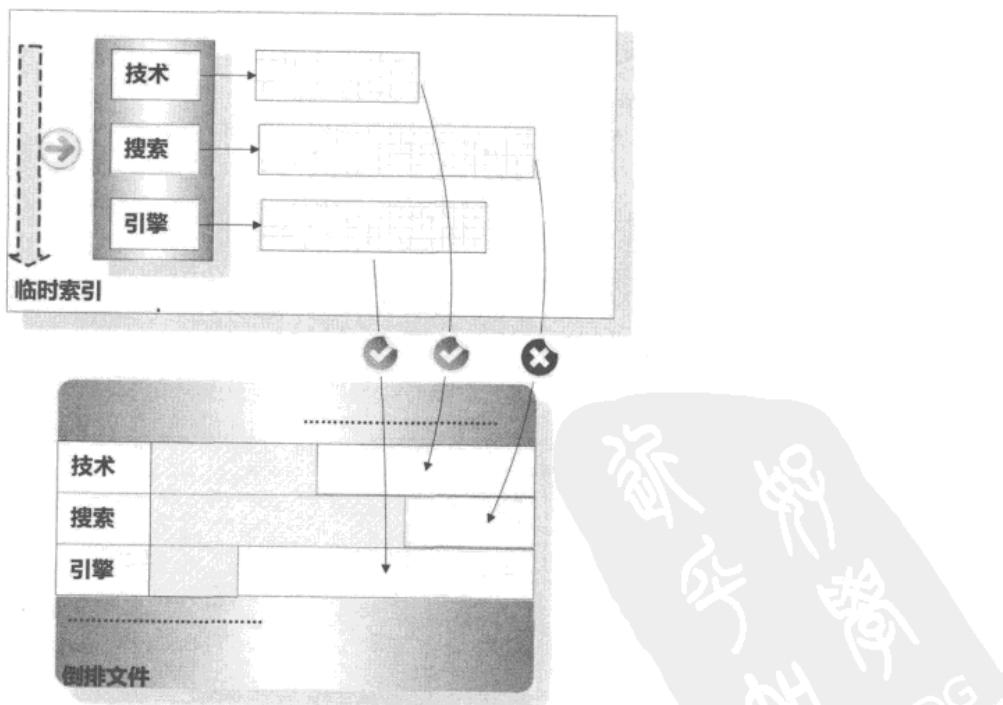


图 3-20 原地更新策略索引合并

原地更新策略的出发点很好，但是实验数据证明其索引更新效率比再合并策略低，主要是



出于以下两个原因。

在这种方法中，对倒排列表进行“迁移”是比较常见的操作，为了能够进行快速迁移，需要找到足够大的磁盘连续存储区，所以这个策略需要对磁盘可用空间进行维护和管理，而这种维护和查找成本非常高，这成为该方法效率的一个瓶颈。

对于倒排文件中的相邻索引单词，其倒排列表顺序一般是按照相邻单词的词典序存储的，但是由于原地更新策略对单词的倒排列表做数据迁移，某些单词及其对应倒排列表会从老索引中移出，这样就破坏了这种单词连续性，导致在进行索引合并时不能进行顺序读取，必须维护一个单词到其倒排文件相应位置的映射表，而这样做，一方面降低了磁盘读取速度，另外一方面需要大量的内存来存储这种映射信息。

#### 3.6.4 混合策略 ( Hybrid )

混合策略的出发点是能够结合不同索引更新策略的长处，将不同的索引更新策略混合，以形成更高效的方法。

混合策略一般会将单词根据其不同性质进行分类，不同类别的单词，对其索引采取不同的索引更新策略。常见的做法是：根据单词的倒排列表长度进行区分，因为有些单词经常在不同文档中出现，所以其对应的倒排列表较长，而有些单词很少见，则其倒排列表就较短。根据这一性质将单词划分为长倒排列表单词和短倒排列表单词。长倒排列表单词采取原地更新策略，而短倒排列表单词则采取再合并策略。

之所以这样做，是由于原地更新策略更适合长倒排列表单词，因为这种策略能够节省磁盘读/写次数，而长倒排列表单词的读/写开销明显要比短倒排列表单词大很多，所以如果采用原地更新策略，效果体现得比较显著。而大量短倒排列表单词读/写开销相对而言不算太大，所以利用再合并策略来处理，则其顺序读/写优势也能被充分利用。

### 3.7 查询处理

为搜索引擎构建索引，其目的是能更快速地提取与用户查询相关的文档信息，假设搜索引擎已经根据前述章节内容建立好了索引，如何利用倒排索引来响应用户的查询？本节主要描述搜索引擎对于用户查询的处理过程。目前有两种常见的查询处理机制，一种被称做一次一文档方式，另外一种被称为一次一单词方式。除了两种基本的查询处理过程，本节还介绍跳跃指针这种查询优化过程。

下面我们用一个具体的例子来分别说明两种基本查询处理方式的运行机制。在这个例子中，假设用户输入的查询为“搜索引擎 技术”，而“搜索引擎”这个单词对应的倒排列表中，

文档 ID 依次为 {1,3,4}，“技术”这个单词对应的倒排列表中，文档 ID 列表为 {1,2,4}。从这两个倒排列表可以看出，文档 1 和文档 4 同时包含了这两个查询词。

### 3.7.1 一次一文档 (Doc at a Time)

搜索引擎接收到用户的查询后，首先将两个单词的倒排列表从磁盘读入内存。所谓的一次一文档，就是以倒排列表中包含的文档为单位，每次将其中某个文档与查询的最终相似性得分计算完毕，然后开始计算另外一个文档的最终得分，直到所有文档的得分都计算完毕为止。

图 3-21 是一次一文档的计算机制示意图，为了便于理解，图中对于两个单词的倒排列表中的公共文档（文档 1 和文档 4）进行了对齐。图中虚线箭头标出了查询处理计算的行进方向。

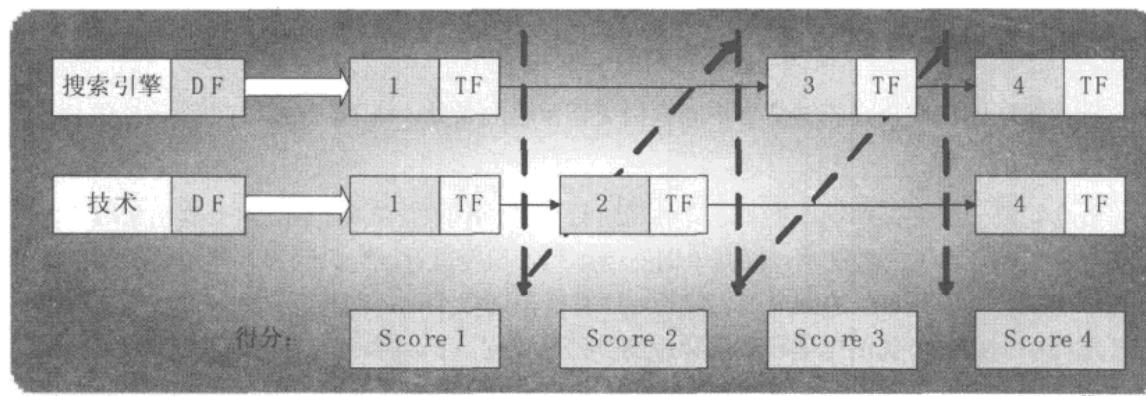


图 3-21 一次一文档

对于文档 1 来说，因为两个单词的倒排列表中都包含这个文档，所以可以根据各自的 TF 和 IDF 等参数计算文档和查询单词的相似性（具体计算方法请参考第 5 章内容，此处为了方便说明，对相似性计算做了简化处理），之后将两个分数相加获得了文档 1 和用户查询的相似性得分。随后搜索系统开始处理文档 2，因为文档 2 只在“技术”这个词汇的倒排列表中，所以根据相应的 TF 和 IDF 计算相似性后，即可得出文档 2 和用户查询的相似性得分。用类似的方法依次处理文档 3 和文档 4。所有文档都计算完毕后，根据文档得分进行大小排序，输出得分最高的  $K$  个文档作为搜索结果输出，即完成了一次用户查询的响应。

因为搜索系统的输出结果往往是限定个数的，比如指定输出 10 个结果，所以在实际实现一次一文档方式时，不必保存所有文档的相关性得分，而只需要在内存中维护一个大小为  $K$  的优先级别队列，用来保存目前计算过程中得分最高的  $K$  个文档即可，这样可以节省内存和计算时间，一般会采用根堆数据结构来实现这个优先级别队列，在计算结束时，按照得分大小输出就可以实现搜索的目标。

### 3.7.2 一次一单词 (Term at a Time)

一次一单词的计算过程与一次一文档不同，一次一文档可以直观理解为在单词—文档矩阵中，以文档为单位，纵向进行分数累计，之后移动到后续文档接着计算，即计算过程是“先纵向再横向”；而一次一单词则是采取“先横向再纵向”的方式，即首先将某个单词对应的倒排列表中的每个文档 ID 都计算一个部分相似性得分，也就是说，在单词—文档矩阵中首先进行横向移动，在计算完毕某个单词倒排列表中包含的所有文档后，接着计算下一个单词倒排列表中包含的文档 ID，即进行纵向计算，如果发现某个文档 ID 已经有了得分，则在原先得分基础上进行累加。当所有单词都处理完毕后，每个文档最终的相似性得分计算结束，之后按照大小排序，输出得分最高的  $K$  个文档作为搜索结果。

图 3-22 是一次一单词的运算机制图示说明，图中虚线箭头指示出了计算的行进方向，为了保存数据，在内存中使用哈希表来保存中间结果及最终计算结果。搜索系统首先对包含“搜索引擎”的所有文档进行部分得分计算，比如对于文档 1，可以根据 TF 和 IDF 等参数计算这个文档对“搜索引擎”这个查询词的相似性得分，之后根据文档 ID 在哈希表中查找，并把相似性得分保存在哈希表中。依次对文档 3 和文档 4 进行类似的计算。当“搜索引擎”这个单词的所有文档都计算完毕后，开始计算“技术”这个单词的相似性得分，对于文档 1 来说，同样地，根据 TF 和 IDF 等参数计算文档 1 和“技术”这个单词的相似性得分，之后查找哈希表，发现文档 1 已经存在得分（“搜索引擎”这个单词和文档 1 的相似性得分），则将哈希表对应的得分和刚刚计算出的得分相加作为最终得分，并更新哈希表中文档 1 对应的得分分值，获得了文档 1 和用户查询最终的相似性得分，之后以类似的方法，依次计算文档 2 和文档 4 的得分。当全部计算完毕时，哈希表中存储了每个文档和用户查询的最终相似性得分，排序后输出得分最高的  $K$  个文档作为搜索结果。这样，就以一次一单词的方式完成了对用户查询的响应。

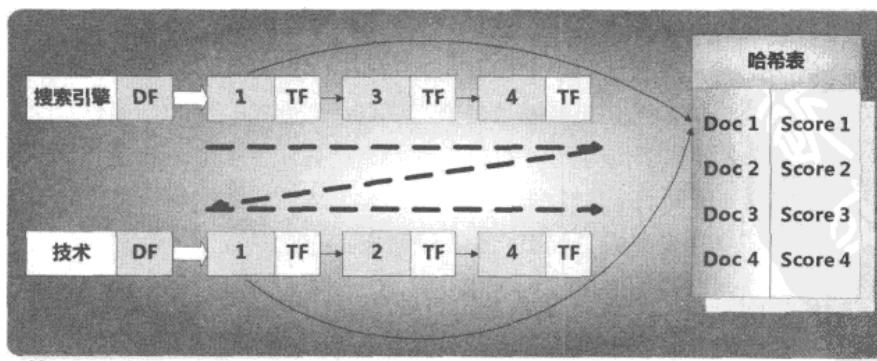


图 3-22 一次一单词

### 3.7.3 跳跃指针 ( Skip Pointers )

如果用户输入的查询包含多个查询词，搜索引擎一般默认是采取与逻辑来判别文档是否满足要求，即要求相关网页必须包含所有的查询词，比如用户输入查询“搜索引擎 技术”，只有同时包含两个词汇的网页才会被认为是相关的，只包含其中任意一个关键词的网页不会作为搜索结果输出。

很明显，对于这种应用场景，一次一文档的查询处理方式是比较适合的。对于多词查询，找到包含所有查询词的文档，等价于求查询词对应的倒排列表的交集。

图 3-23 显示了“搜索引擎”和“技术”两个单词对应的倒排列表，为了简化问题，我们假设倒排列表里只存储文档 ID。从图 3-23 可以看出，包含“搜索引擎”单词的文档，分别为文档 5、文档 7、文档 8 等。假设用户输入查询“搜索引擎 技术”，搜索系统需要从索引结构中找出包含所有用户查询单词的文档列表，在图 3-23 的索引结构示例中，即需要找出“搜索引擎”和“技术”这两个查询词各自对应倒排列表的文档 ID 交集{5,21,23}，这 3 个文档包含所有查询词汇，即是我们需要的文档。

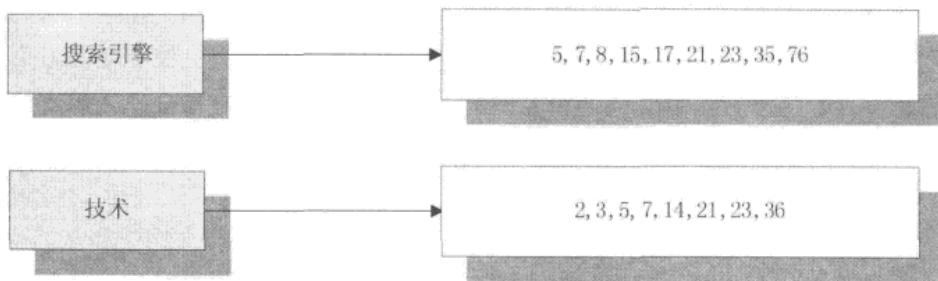


图 3-23 索引示例

如果倒排列表直接存储包含查询词的文档 ID，那么计算交集是非常直观和简单的，其基本操作即是在一个倒排列表里查询某个文档 ID 是否存在，通过归并排序即可获得  $O(m+n)$  时间复杂度的算法。如果文档 ID 是以文档编号差值 (D-Gap) 形式存储的，另外这个差值是以压缩后的方式编码的，那么如何求倒排列表的交集就会复杂化。为了求两个查询词对应倒排列表的交集，首先需要将两个单词对应的倒排列表读入内存，然后对数据解压缩恢复到文档编号差值的形式，之后还需要将其恢复成文档 ID 的有序列表，在此基础上才能进行集合交集运算。而跳跃指针思想的引入，即可加快倒排列表求交集这一计算过程。

跳跃指针的基本思想是将一个倒排列表数据化整为零，切分为若干个固定大小的数据块，一个数据块作为一组，对于每个数据块，增加元信息来记录关于这个块的一些信息，这样即使是面对压缩后的倒排列表，在进行倒排列表合并的时候也能有两个好处：一个好处是无须解压缩所有倒排列表项，只解压缩部分数据即可；另外一个好处是无须比较任意两个文档 ID，通



过这两种方式有效节省了计算资源和存储资源。

图 3-24 即是将“Google”这个查询词对应的倒排列表加入跳跃指针后的数据结构。我们假设对于“Google”这个单词的倒排列表来说，数据块的大小为 3，即每块数据包含 3 个文档 ID 及其词频信息。之后，我们可以在每块数据前面加入管理信息，比如第 1 块的管理信息是<<5,Pos1>>，其中数字 5 代表块中第 1 个文档 ID 的编号，Pos1 即是跳跃指针，指向第 2 块的起始位置。

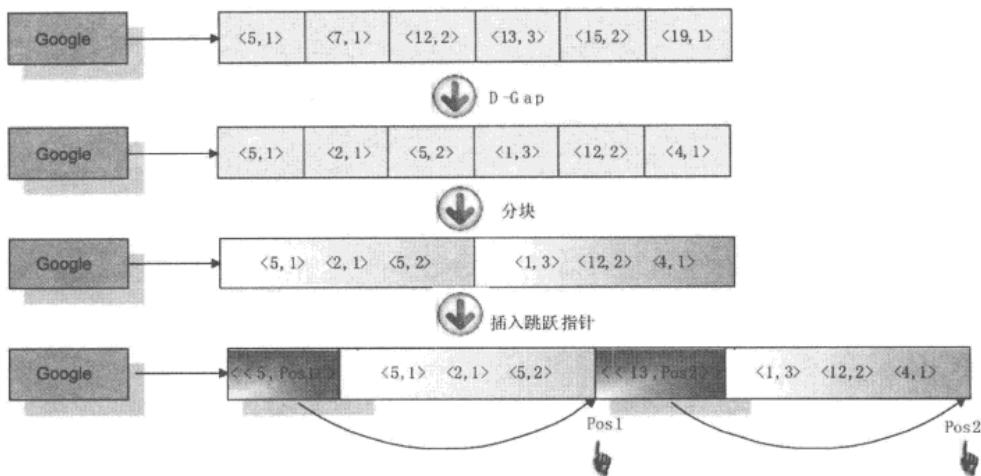


图 3-24 跳跃指针

建好上述索引结构，如何在一个带有跳跃指针的倒排列表里查找某个文档是否存在呢？

假设我们需要在单词“Google”压缩后的倒排列表里查找文档编号为 7 的文档。首先，对倒排列表前两个数值进行数据解压缩，读取第 1 组的跳跃指针数据，发现其值为<5,Pos1>，其中 Pos1 指出了第 2 组的跳跃指针在倒排列表中的起始位置，于是可以解压缩 Pos1 位置处连续两个数值，得到<13,Pos2>。跳跃指针的数值 5 和数值 13，分别表示两组数据中最小编号文档的文档 ID，我们要查找的 7 号文档落在两者之间，说明如果 7 号文档包含在单词“Google”的倒排列表中的话，一定会在第 1 组中，否则说明倒排列表中不包含这个文档。于是可以依次对第 1 组中的数据进行解压缩，并根据最小文档编号逆向恢复其原始的文档编号，当读到<2,1>的时候，可以知道这个文档对应的原始文档 ID 为： $5+2=7$ ，与我们正在查找的文档编号相同，说明 7 号文档在单词“Google”的倒排列表中，于是可以结束这次查找。而求两个查询词的倒排列表交集就是反复在两个倒排列表中查找某个文档是否存在，将同时在两个倒排列表中出现的文档 ID 作为计算结果。

从上面的查找过程可以看出，相对不包含跳跃指针的索引来说，我们只需要对其中一个数据块进行解压缩和文档编号查找即可获得结果，不用将所有索引数据都进行解压缩和比较操

作，很明显这将加快查找速度，并节省内存空间。

上面例子设定分组大小为 3，而在实际应用中，如何设定数据块或者数据组的大小对于效率有影响。数据块越小，则使用跳跃指针向后进行跳跃的可能性越大，但是缺点是增加了指针比较操作的次数；数据块越大，则可以有效减少指针比较次数，但是使用跳跃指针向后跳跃的可能性越小。所以需要根据数据情况对块大小进行合理设置才能取得最优结果。实践表明一个简单有效的启发规则是：假设倒排列表长度为  $L$ （即包含  $L$  个文档 ID），使用  $\sqrt{L}$  作为块大小，则效果较好。

## 3.8 多字段索引

在很多实际的搜索应用领域，搜索引擎所要处理的文档是有一定结构的，即文档包含明确区分的多个字段。比如电子邮件，包含“发件人”、“收件人”、“标题”和“正文”几个重要字段。再比如论文搜索引擎，论文是有规范格式的，包含“标题”、“作者”、“摘要”、“正文”、“参考文献”等几个字段。对于这种多字段类型文档，搜索引擎应该能够支持用户指定某个字段作为搜索范围，比如邮件搜索应用应该允许用户在标题里搜索某个关键词是否出现。

即使是对互联网网页，也可以切割为不同的字段，搜索关键词出现在不同字段里代表的权重是不同的。如果搜索关键词出现在标题，很明显这个网页的相关性会高于只在正文中出现关键词的网页。所以区分不同字段对于搜索引擎的相关性评分也有很大帮助。

为了能够支持以上需求，搜索引擎需要能够对多字段进行索引，而实现多字段索引有 3 种方式：多索引方式、倒排列表方式和扩展列表方式。

### 3.8.1 多索引方式

多索引方式针对每个不同的字段，分别建立一个索引，当用户指定某个字段作为搜索范围时，可以从相应的索引里提取结果。图 3-25 是这种方式的示意图。我们假设要处理的文档分为“标题”、“摘要”和“正文”3 个字段，本节后续的例子也是以这种类型的文档来做说明的。

当用户没有指定特定字段时，搜索引擎会对所有字段都进行查找并合并多个字段的相关性得分，对于多索引方式来说，就需要对多个索引进行读取，所以这种方式的效率会比较低。

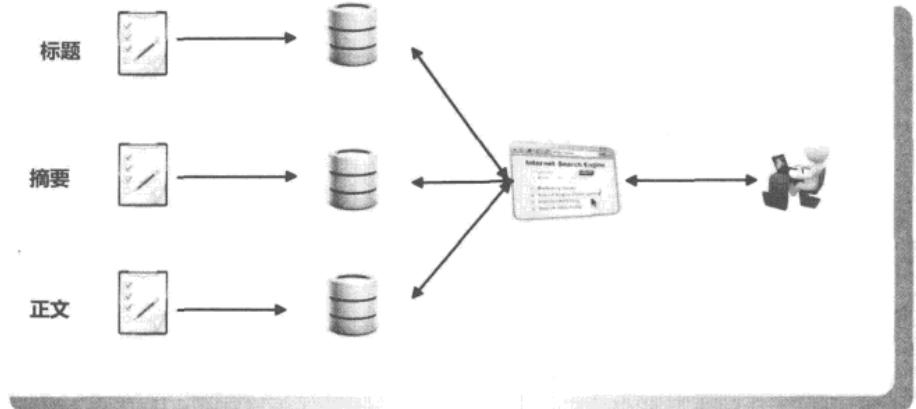


图 3-25 多索引方式

### 3.8.2 倒排列表方式

为了能够支持对指定字段的搜索，也可以将字段信息存储在某个关键词对应的倒排列表内，在倒排列表中每个文档索引项信息的末尾追加字段信息，这样在读出用户查询关键词的倒排列表的同时，就可以根据字段信息，判断这个关键词是否在某个字段出现，以此来进行过滤，并保留指定字段内出现过搜索词的文档作为搜索结果返回。

图 3-26 是一个具体的示例。上面提到，我们要处理的文档包含了 3 个字段，所以可以用 3 个比特位（Bit）来分别表示关键词是否在某个字段出现过，每个比特位对应一个字段。如果关键词在某个字段出现过，则相应的比特位设定为 1，否则设定为 0。我们假设 3 个比特位从左到右依次分别对应“标题”、“摘要”和“正文”这种顺序，如果比特位的值为“101”，则代表某个关键词在“标题”和“正文”中出现过，但是没有在“摘要”中出现。

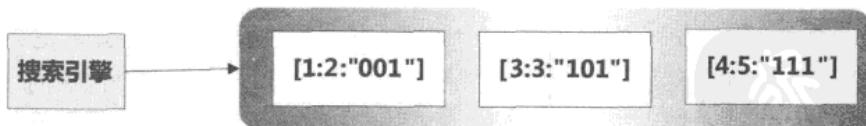


图 3-26 倒排列表方式

如图 3-26 所示的例子显示了“搜索引擎”这个关键词对应的倒排列表，从中可以看出，有 3 个文档包含关键词“搜索引擎”，它们对应的文档编号为 1、3 和 4，紧跟在文档编号之后的是单词频率信息，末尾则是字段信息。我们以文档 1 存储的信息为例，文档 1 的单词频率为 2，相应的字段标记为“001”，说明文档 1 只在“正文”中出现过“搜索引擎”这个关键词。其他的文档所存储的信息含义与此类似。

当用户指定在文档标题中搜索“搜索引擎”这个查询词的时候，根据倒排列表中的字段信息可以看出，文档 3 和文档 4 满足条件，于是输出这两个文档作为搜索结果。

### 3.8.3 扩展列表方式 (Extent List)

扩展列表是实际中应用得比较多的支持多字段索引的方法。这个方法为每个字段建立一个列表，这个列表记载了每个文档这个字段对应的出现位置信息。

图 3-27 是扩展列表的示意图，为了简便，图中只展示了针对“标题”字段所建立的扩展列表，其他字段的扩展列表与此类似。在标题扩展列表中，记载了所有文档的标题位置信息，比如第 1 项存储的信息为 $<1,(1,4)>$ ，代表对于文档 1 来说，其标题的位置为从第 1 个单词到第 4 个单词这个范围，扩展列表内其他项含义类似。

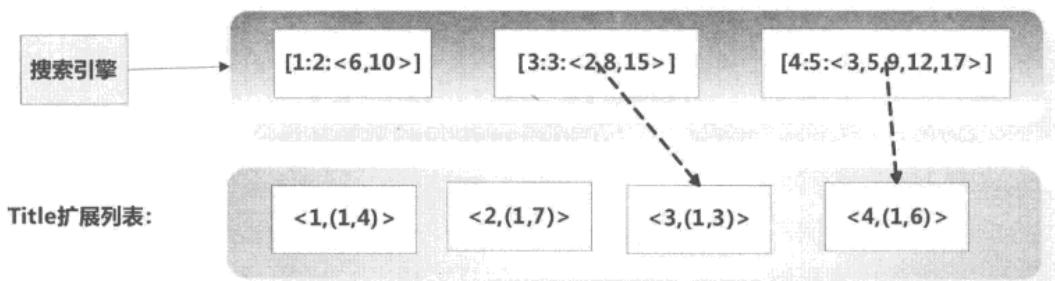


图 3-27 扩展列表方式

假设用户指定在标题字段里搜索“搜索引擎”这个查询，通过倒排列表可以知道文档 1、文档 3 和文档 4 包含这个查询词，接下来需要判断：这些文档是否在标题中出现过查询词？对于文档 1 来说，“搜索引擎”这个查询词的出现位置是第 6 和第 10 这两个位置，而通过对称的标题扩展列表可知，文档 1 的标题范围是位置 1 到位置 4，这说明文档 1 的标题内不包含查询词，即文档 1 不满足要求。对于文档 3 来说，“搜索引擎”在文档中的出现位置是 2、8 和 15，而对应的标题扩展列表中，标题出现范围为位置 1 到位置 3，说明在位置 2 出现的这个查询词是在标题范围内的，即满足要求，可以作为搜索结果输出。文档 4 也类似，是满足搜索条件的文档，于是可以输出文档 3 和文档 4 作为搜索结果。

## 3.9 短语查询

短语是很常见的语言现象，几个经常连在一起被使用的单词就构成了短语，比如“你懂的”。短语强调单词之间的顺序，有时尽管是同样的单词，顺序颠倒后会产生完全不同的含义，比如“懂你的”和“你懂的”含义相差甚远。



搜索引擎如何能够支持短语呢？如果单词的倒排列表只存储文档编号和单词频率信息，其保留的信息是不足以支持短语搜索的，因为单词之间的顺序关系没有保留。搜索引擎支持短语查询，本质问题是如何在索引中维护单词之间的顺序关系或者位置信息。较常见的支持短语查询的技术方法包括：位置信息索引、双词索引及短语索引这3类，为了能够更有效地利用存储和计算资源，也可以将三者结合使用。

### 3.9.1 位置信息索引（Position Index）

由前面的章节我们知道，对于词典中某个单词的倒排列表，往往存储3种信息：文档ID、单词频率和单词位置信息。一般情况下不存储单词位置信息，因为这种信息数量过大，一旦加入位置信息，单词的倒排列表长度会剧烈增长，这样一方面消耗存储空间，另一方面影响磁盘读取效率，对快速响应用户查询不利。但是如果索引记载单词位置信息，则能很方便地支持短语查询。

假设用户输入短语查询“爱情买卖”，图3-28说明了位置索引是如何支持短语查询的，在单词倒排列表中存储了文档ID、单词频率及位置信息，比如 $<5,2,[3,7]>$ 索引项的含义是：5号文档包含“爱情”这个单词，且这个单词在文档中出现2次，其对应的位置为3和7，其他倒排列表项的含义与此相同。

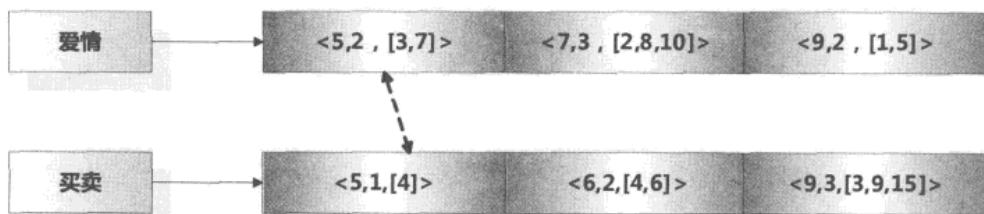


图3-28 位置信息索引

搜索引擎获得了用户查询后，从磁盘分别读入两个单词的倒排列表，通过倒排列表记载的信息可知，文档5和文档9同时包含两个查询词，为了判断在这两个文档中，用户查询是否以短语的形式存在，还要判断位置信息。“爱情”这个单词在5号文档的出现位置是3和7，而“买卖”在5号文档的出现位置是4，可以看出5号文档的位置3和位置4分别对应单词“爱情”和“买卖”，即两者是一个短语形式，而对于9号文档，做同样的分析后可知，这两个查询词并没有连续在文档中出现，所以搜索引擎会将5号文档作为搜索结果返回，这样就通过位置信息完成了对短语查询的支持。对于超过两个查询词的短语，也可用类似的方式得到结果。

尽管位置索引可以支持短语查询，但是对于有些查询，其付出的存储和计算代价很高，读者可以思考一下“我的团长我的团”这种短语查询的存储和计算代价。

### 3.9.2 双词索引 (Nextword Index)

双词索引是另外一种可以对短语查询提供支持的索引结构。短语至少包含两个单词，也可能包含多个单词，统计数据表明，二词短语在短语中所占比例最大，所以如果能够针对二词短语提供快速查询，也能解决短语查询的问题。

对于两个单词构成的短语来说，一般称第1个单词为“首词”，第2个单词为“下词”。图3-29是双词索引的数据结构，在内存中包含两个词典，分别是“首词”词典和“下词”词典，“首词”词典有指针指向“下词”词典某个位置，“下词”词典存储了紧跟在“首词”词典后的常用短语的第2个单词，“下词”词典的指针指向包含这个短语的倒排列表。比如对于“我的”这个短语，其倒排列表包含文档5和文档7，对于“的父亲”这个短语，其倒排列表包含文档5，词典中其他词典项也是类似的含义。

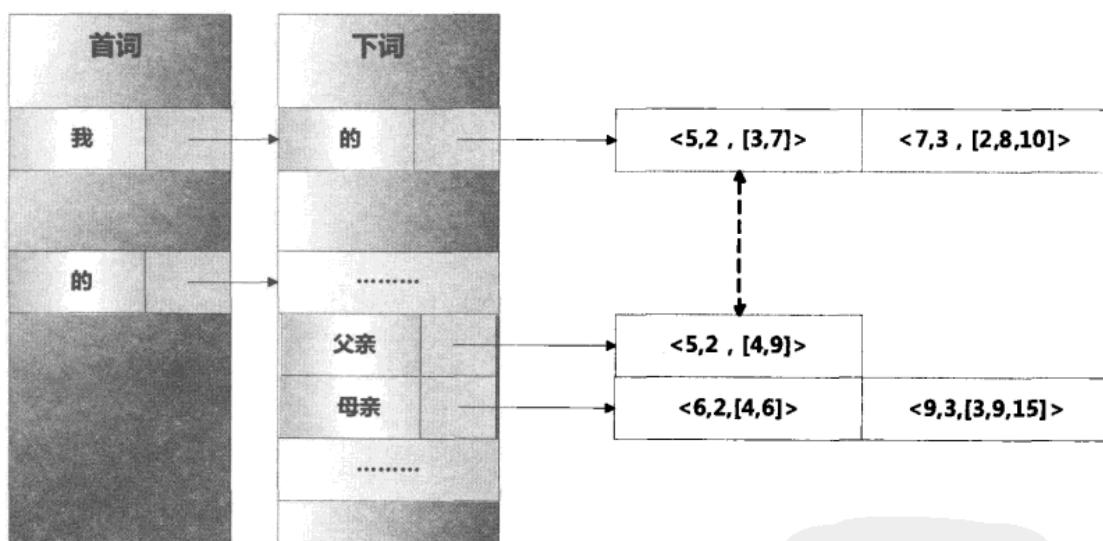


图3-29 双词索引

假设用户输入短语查询“我的父亲”，搜索引擎首先将其拆分成两个短语“我的”和“的父亲”，然后分别查找词典信息，发现包含“我的”这个短语的是文档5和文档7，而包含“的父亲”这个短语的有文档5。查看其对应的出现位置，可以知道文档5是符合条件的搜索结果。这样就完成了对短语查询的支持。

双词索引同样使得索引急剧增大，原先的索引结构中，词典是一维的，而双词索引词典结构是二维的，这样倒排列表个数会发生爆炸性增长，所以一般实现时并非对所有单词都建立双词索引，而是只对计算代价高的短语建立双词索引，比如包含“我”、“的”等停用词的短语，这些短语如果用常规方法处理的话，存储和计算代价太高，采用双词索引可以极大地缓解这种



状况，对于一般的短语，可以使用位置信息等常规手段来达到目的。

### 3.9.3 短语索引 (Phrase Index)

位置索引和双词索引可以有效支持短语查询，但其实还有一种更直观的方法来解决这个问题，那就是直接在索引中加入短语索引。之前介绍的常规索引结构中，词典都是以单词作为查询和存储单位的，如果将其扩展，在词典中直接加入多词短语并维护短语的倒排列表，这样也可以对短语进行支持。

短语索引有自己的缺点：不可能事先将所有短语都建好索引。通用的做法是挖掘出热门短语，为这些短语专门建立索引，对于其他的短语查询，则采用常规方法处理。挖掘短语的数据来源可以是多样化的，比如可以从用户查询日志（Search Log）挖掘或者从文本本身挖掘。

图 3-30 是加入短语索引后的整体索引结构，当搜索引擎接收到用户查询后，首先在短语索引里查找，如果找到，则计算后返回给用户搜索结果，否则仍然利用常规索引进行查询处理。

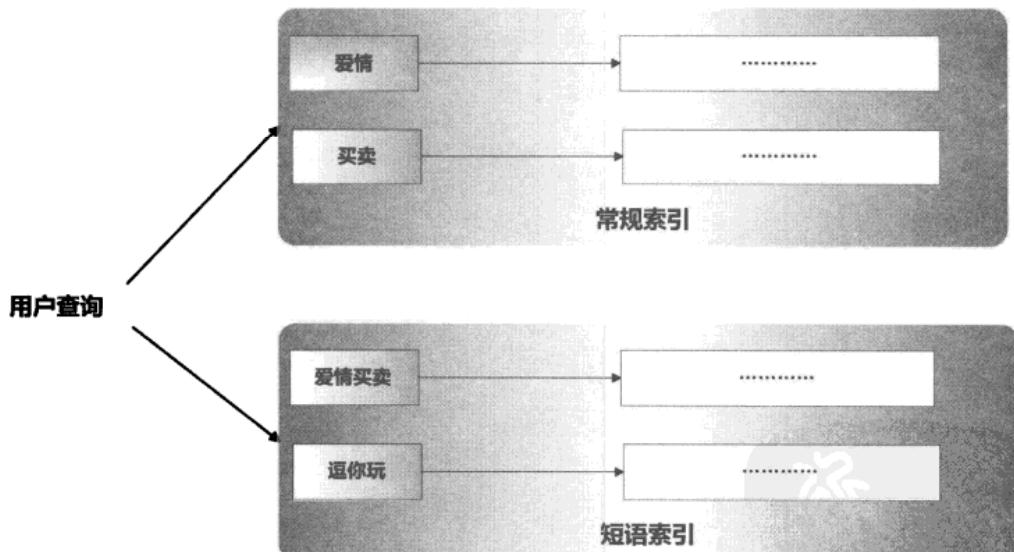


图 3-30 短语索引

### 3.9.4 混合方法

从上述介绍可以看出，不同的短语索引结构有其各自的特点，位置索引适合处理常规的短语查询，即计算代价较小的短语，双词索引适合处理计算代价较高的短语查询，而短语索引则适合处理热门短语查询或者文本中高频率度出现的短语，此三者是以互补关系存在的，如果能够

在构建系统时有机集成三者，就能使系统效率发挥出综合优势。

图 3-31 是同时利用以上 3 种方式的混合索引结构，短语索引用来对热门短语和高频短语进行索引，双词索引对包含停用词等高代价短语进行索引。接收到用户查询后，系统首先在短语索引中查找，如果找到则返回结果，否则到双词索引中查找，如果找到则返回结果，否则从常规索引中对短语进行处理，这样可以充分发挥各自的优势。

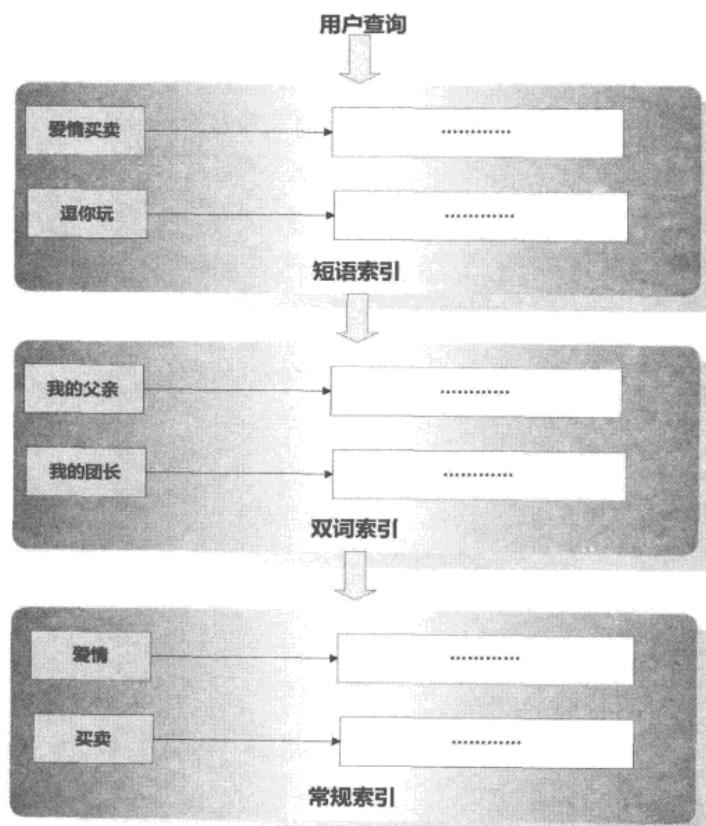


图 3-31 混合索引

### 3.10 分布式索引 (Parallel Indexing)

当搜索引擎需要处理的文档集合数量非常庞大时，靠单机往往难以承担如此重任，此时需要考虑分布式解决方案，即每台机器维护整个索引的一部分，由多台机器协作来完成索引的建立和对查询的响应。至于多台机器如何分工协作，目前常用的分布式索引方案包括两种：按文档对索引划分和按单词对索引划分。

### 3.10.1 按文档划分 ( Document Partitioning )

所谓的按文档对索引划分，就是将整个文档集合切割成若干子集合，而每台机器负责对某个文档子集合建立索引，并响应查询请求。图 3-32 是该方式的一个示意图，假设文档集合包含 5 个文档，将其切割成两个子集合，分别交由两台机器建立索引。

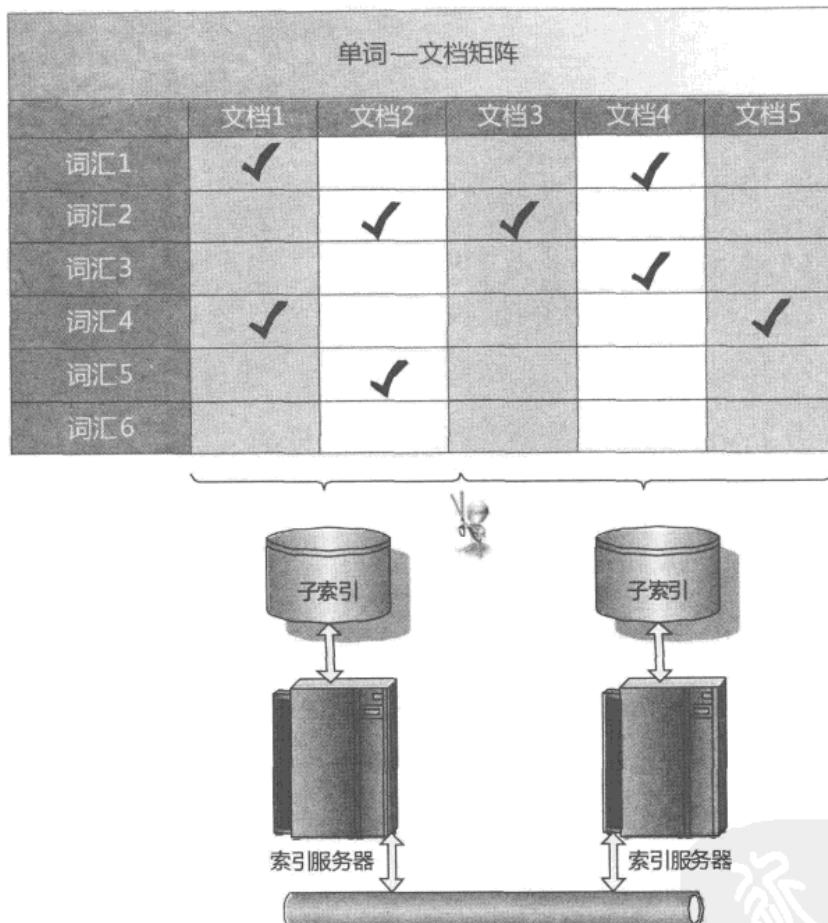


图 3-32 按文档划分

如图 3-33 所示为此分布式索引方案对用户查询的响应过程。查询分发服务器接收到用户查询请求后，将查询广播给所有索引服务器。每个索引服务器负责部分文档子集合的索引维护和查询响应，当索引服务器接收到用户查询后，按照本章前述小节所述计算相关文档，并将得分最高的  $K$  个文档送返查询分发服务器。查询分发服务器综合各个索引服务器的搜索结果后，合并搜索结果，将得分最高的  $m$  个文档作为最终搜索结果返回给用户，这样就完成了对一次用户查询的响应。

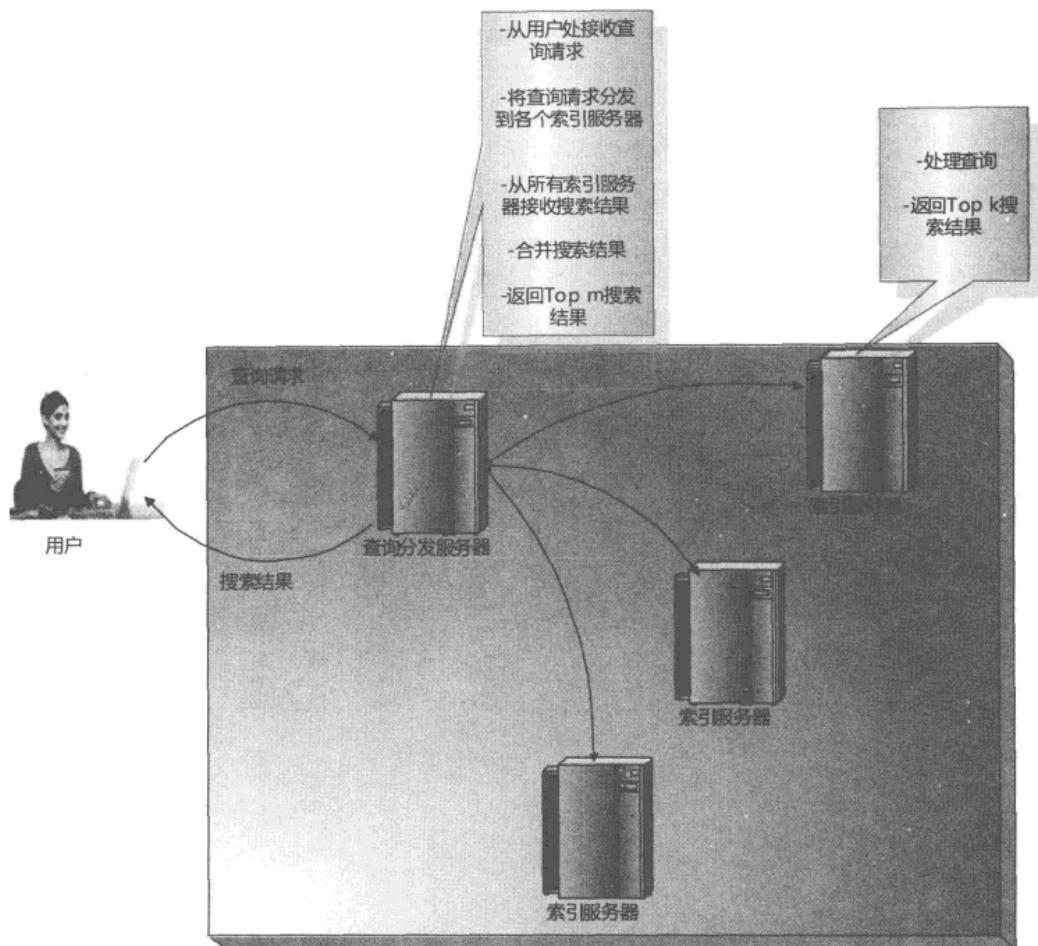


图 3-33 文档划分方式对用户查询的响应

### 3.10.2 按单词划分 (Term Partitioning)

按单词划分索引方式与上述按文档划分索引方式不同，不是对文档集合进行切割，而是对单词词典进行划分，每个索引服务器负责词典中部分单词的倒排列表的建立和维护，图 3-34 是这种分布式索引方案的示意图。假设单词词典包含了 6 个单词，这种方式将每 3 个单词的倒排列表存储在一台索引服务器上，以此协作方式来完成整个索引系统。

按照单词对索引进行划分的分布式方案中，如何响应用户查询？图 3-35 是这种方案响应用户查询请求的示意图，值得注意的是，在图中所示体系结构下，搜索系统的查询处理方式只能是一次一单词的方式（细节参见 3.7 节内容）。

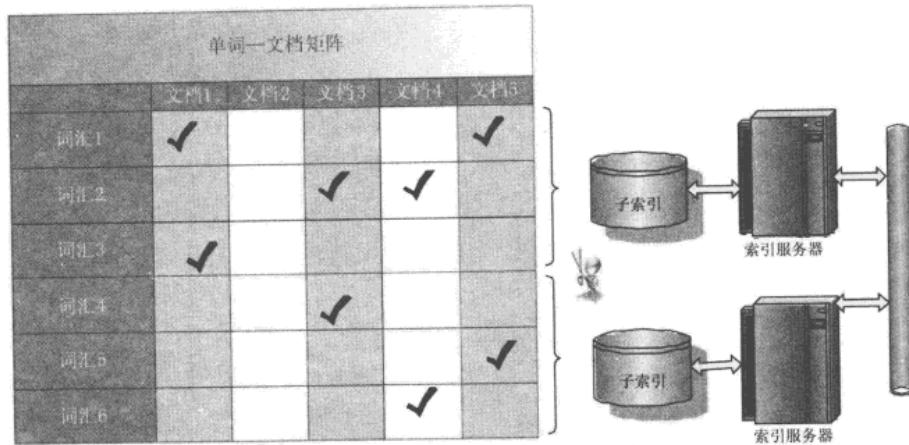


图 3-34 按单词划分

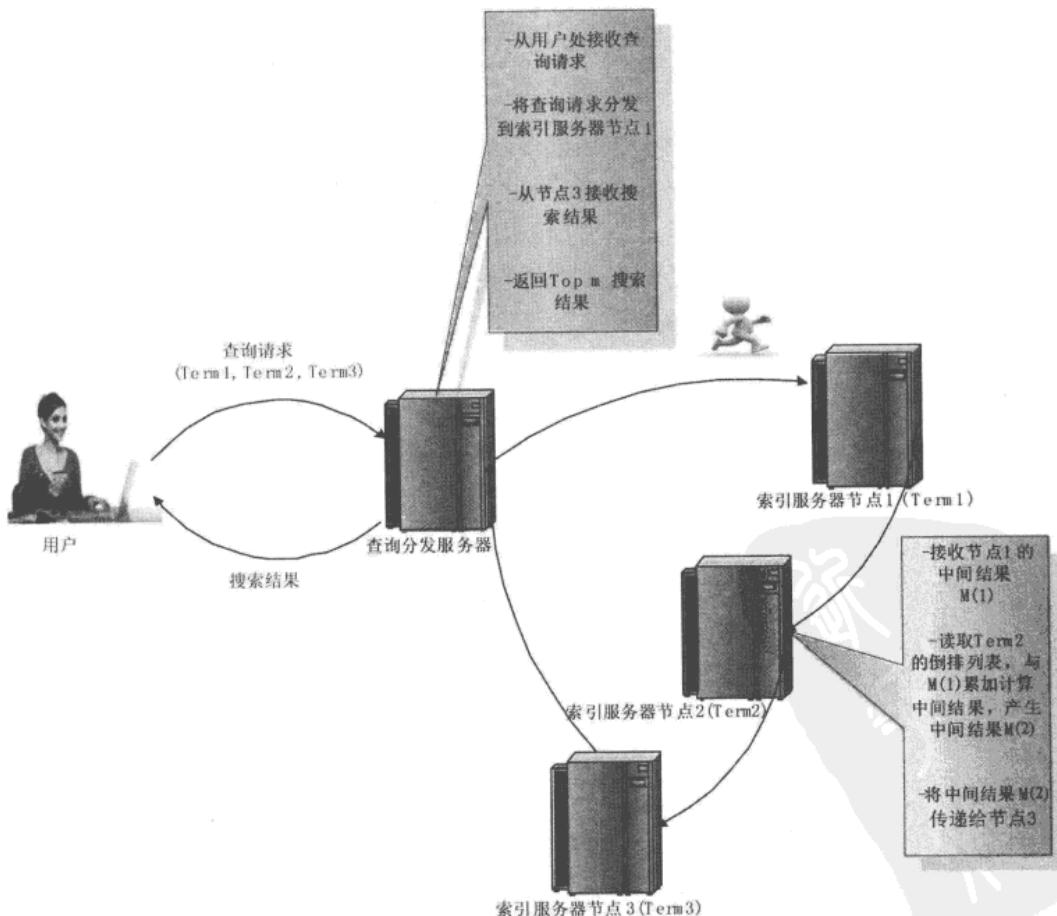


图 3-35 单词倒排列表分布情况查询处理

在图示例子中，假设用户输入的查询包含 3 个单词 Term1、Term2 和 Term3，查询分发服务器接收到用户查询后，将查询转发给包含单词 Term1 倒排列表的索引服务器节点 1，索引服务器节点 1 提取 Term1 的倒排列表，并累计计算搜索结果的中间得分，然后将查询和中间结果传递给包含单词 Term2 倒排列表的索引服务器节点，索引服务器节点 2 也是类似处理，并继续传递到索引服务器节点 3。索引服务器节点 3 处理完成后，将最终结果返回给查询分发服务器，查询分发服务器计算得分最高的  $K$  个文档作为搜索结果输出。很明显，这是典型的一次一单词的查询处理方式。

### 3.10.3 两种方案的比较

以上两种分布式索引技术方案，按文档来对索引进行划分是比较常用的，而按单词进行索引划分只在比较特殊的应用场合才使用。之所以如此，是因为按单词进行索引划分在以下几个方面存在不足。

#### 可扩展性

搜索引擎处理的文档集合往往是在不断变动的，如果是按文档来对索引划分，则很容易支持新增的文档，只要增加索引服务器，将新增的文档由新增索引服务器来负责处理即可，对系统其他部分影响很小，在实现上也非常方便。但是如果是按单词进行索引划分，则对几乎所有的索引服务器都有直接影响，因为新增文档可能包含所有词典单词，即需要对每个单词的倒排列表进行更新，实现起来相对复杂。

#### 负载均衡

按单词进行索引划分在索引服务器的负载均衡方面做得也不是很好。有些单词比较常见，几乎出现在所有文档中，比如一些常用词，这些单词的倒排列表会非常庞大，可能会达到几十兆字节。如果是按文档进行索引划分，这种单词的倒排列表会比较均匀地分布在不同的索引服务器上，而按单词进行索引划分，某个常见单词的倒排列表全部内容都由一台索引服务器维护，如果这个单词同时又是一个流行词汇，会有很多用户搜索，维护这个单词倒排索引的索引服务器会成为负载过大的性能瓶颈。

#### 容错性

假设在分布式索引系统里，某台索引服务器发生故障，对于按文档进行索引划分的方案来说，因为这仅影响到部分文档子集合，即使某台索引服务器不能响应查询，其他索引服务器仍然能够响应，对于用户来说，并不会直接感受到这种故障的影响。但是对于按单词进行索引划分的情况，若索引服务器发生故障，则某些单词的倒排列表无法访问，用户查询这些单词的时



候，会发现没有搜索结果，这直接影响用户体验。

### 对查询处理方式的支持

如上节所述，按单词进行索引划分只能支持一次一单词这种查询处理方式，而按文档进行索引划分则不受此限制，可以同时支持两种不同的查询处理方式。对于某些类型的搜索来说，需要一次一文档这种查询处理方式支持，否则在机制上就无法实现。按文档进行索引划分可以根据具体情况选择不同的查询处理方式，而按单词进行索引划分则没有这种灵活性。

## 本章提要

- 倒排索引是搜索引擎用来快速查找包含某个单词的文档集合的数据结构。
- 倒排索引由单词词典和所有单词对应的倒排列表构成。
- 倒排列表由倒排列表项构成，一般倒排列表项包含文档 ID、单词出现次数和单词在文档出现位置的信息，而文档 ID 则采取文档编号差值方式编码。
- 3 种常用的建立倒排索引的方法是：两遍文档遍历法、排序法、归并法。
- 常用的索引更新策略有 4 种：完全重建策略、再合并策略、原地更新策略及混合策略。
- 目前有两种常见的查询处理机制，一种被称做一次一文档方式，另一种被称为一次一单词方式。
- 实现多字段索引有 3 种方式：多索引方式、倒排列表方式和扩展列表方式。
- 较常见的支持短语查询的技术方法包括：位置信息索引、双词索引及短语索引这 3 类，为了能够更有效地利用存储和计算资源，也可以将三者结合使用。
- 目前常用的分布式索引方案包括两种：按文档对索引划分和按单词对索引划分。

## 本章参考文献

- [1] Heinz, S. and Zobel, J. (2003). Efficient single-pass index construction for text databases, *Jour. of the American Society for Information Science and Technology* .54(8), 713–729.
- [2] Zobel, J., Heinz, S. and Williams, H. E. (2001). In-memory hash tables for accumulating text vocabularies. *Information Processing Letters* 80, 271.
- [3] Brown, E. W., Callan, J. P., and Croft, W. B. (1994). Fast incremental indexing for full-text information retrieval. In *Proceedings of the International Conference on Very Large Databases*, Santiago . Morgan Kaufmann, 192–202.

- [4] Buttcher, S. and Clarke, C. L. (2005). Indexing time vs. query time trade-offs in dynamic information retrieval systems. In Proceedings of the International Conference on Information and Knowledge Management, Bremen, Germany, ACM Press, 317–318.
- [5] Cutting, D. and Pedersen, J. (1990). Optimisations for dynamic inverted index maintenance. In Proceedings of the ACM-SIGIR International Conference on Research and Development in Information Retrieval. Brussels, Belgium, J.-L. Vidick, Ed. ACM Press, 405–411.
- [6] Lester, N., Zoble, J., and Williams, H. E. (2006). Efficient online index maintenance for text retrieval systems. *Information of Processing and Management*.
- [7] Williams, H. E., Zobel, J., and Bahle, D. (2004). Fast phrase querying with combined indexes. *ACM Trans on Information System*. 22, 4, 573–594.
- [8] Lester, N., Zobel, J., and Williams, H. E. (2004). In-Place versus Re-Build versus Re-Merge: Index Maintenance Strategies for Text Retrieval Systems. In Proceedings of the 27th Conference on Australasian Computer Science, Darlinghurst, Australia.
- [9] Moffat, A., Zobel, J., (Oct. 1996). Self-indexing inverted list for fast text retrieval. *ACM Transactions on Information Systems* 14 (4), 349–379.
- [10] Lim, L., Wang, M., Padmanabhan, S., Vitter, J.S., and Agarwal, R.(2003). Dynamic maintenance of web indexes using landmarks. In: Proceedings of the 12th international conference on World Wide Web. 102.
- [11] Lempel, R.(2009). Term-at-a-time and document-at-a-time evaluation. In Course on Introduction to Search Engine Technology, Tech- nion, Israel, <http://webcourse.cs.technion.ac.il/236620/inter2006-2007/ho/WCFiles/lec5-evaluation.pdf>.
- [12] Bahle, D., Williams, H. E. and Zobel, J. (2002). Efficient phrase querying with an auxiliary index, in Proc. ACM-SIGIR Int. Conf. on Research and Development in Information Retrieval”, Tampere, Finland, pp. 215–221.
- [13] Zobel, J., Mo\_at, (2006). Inverted list for text search engines. *ACM Computer Surveys* 38(2) 6.
- [14] Badue, C., Baeza-yates, R., Ribeiro, B., and Ziviani, N. (2001). Distributed query processing using partitioned inverted files. In Proceedings of String Processing and Information Retrieval Symposium, Laguna de San Rafael, Chile. G. Navarro, Ed. IEEE Computer Society, 10–20.



- [15] Cacheda, F., Plachouras, V., and Ounis, I. (2004). Performance analysis of distributed architectures to index one terabyte of text. In Proceedings of the European Conference on IR Research, Sunderland, UK, S. McDonald and J. Tait, Eds. 395–408. Lecture Notes in Computer Science, Springer, Vol. 2997.



# 第4章 索引压缩

“佛土生五色茎，  
一花一世界，  
一叶一如来。”

《华严经》

对于海量网页数据，为其建立倒排索引往往需要耗费较大的磁盘空间，尤其是一些比较常见的单词，其对应的倒排列表可能大小有几百兆。如果搜索引擎在响应用户查询的时候，用户查询中包含常见词汇，就需要将大量的倒排列表信息从磁盘读入内存，之后进行查询处理给出搜索结果。由于磁盘读/写速度往往是个瓶颈，所以包含常用词的用户查询，其响应速度会受到严重影响。索引压缩则可以利用数据压缩算法，有效地将数据量减少，这样一方面可以减少索引占用的磁盘空间资源，另一方面可以减少磁盘读/写数据量，加快用户查询的响应速度。

倒排索引主要包含两个构成部分：单词词典和单词对应的倒排列表。所以针对索引的压缩算法，也分为针对词典的压缩和针对倒排列表的压缩。而对倒排列表压缩又可以细分为无损压缩和有损压缩两种。所谓无损压缩，就是将原始倒排列表数据量减小，但是信息并不会因为占用空间的减小而有所损失，通过解压缩算法可以完全恢复原始信息。而有损压缩则是通过损失部分不重要的信息，以此来获得更高的数据压缩率。

本章主要介绍针对搜索引擎索引的压缩算法，首先介绍针对词典的相关压缩技术，第4.2节会介绍针对倒排列表的经典压缩算法，之后是对文档ID重排序技术的说明，这是另一种形式的无损压缩思路。最后一节简单介绍静态索引裁剪这一有损压缩算法。

## 4.1 词典压缩

为了快速响应用户查询，词典数据往往会全部加载到内存中，以加快查找速度。如果文档集合数据量很大，包含的不同单词数目会非常多，内存是否能够放得下全部词典信息



就成了问题。为了减小词典信息所占内存，一般可考虑采用词典压缩技术来达到此目的。

上一章介绍了词典的组织方式有两种：哈希加链表和 B 树形词典结构。在链表内部或者 B 树的叶子节点会存储单词相关信息，一般至少会存储以下 3 项：单词本身内容，文档频率信息（DF）及指向倒排列表的指针信息。图 4-1 是词典结构和信息的示意图。

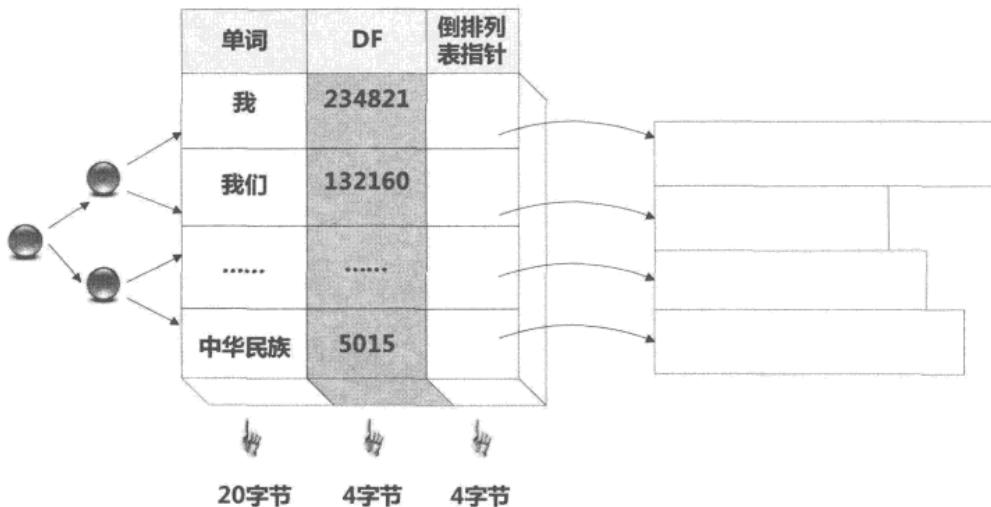


图 4-1 词典信息

对于某个词典项来说，文档频率信息和倒排列表指针信息各自使用 4 个字节表示即可，而单词本身内容可长可短，长的单词比如“中华人民共和国”，短的单词比如“我”，长度差异很大。为了能够容纳最长的单词，需要给单词内容分配足够大的空间，我们假设词典中最长单词是 10 个汉字，即每个单词内容需要分配 20 个字节，即使是像“我”这种单字词，也会占用 20 个字节的位置，浪费严重，很明显这里可以采取一些优化的数据结构来省出存储空间。

图 4-2 是针对单词内容存储结构的一种优化措施，在这个技术方案里，可以将单词连续存储在某个内存区域，原先存储单词内容的部分由指向这个存储区对应单词起始位置的指针代替，单词结尾可以用词典中下一个单词的指针所指向位置来做判断，如此就可以将原先浪费的存储空间利用起来。

在上述优化词典结构的基础上，还可以继续做出改进，图 4-3 是进一步改进方案的示意图。其基本思想是：将连续词典进行分块，图中的例子是将每两个单词作为一个分块，在实际开发时，可动态调整分块大小，以获取最优压缩效果。原先每个词典项需要保留一个指向连续词典区的指针，而分块之后，相邻的两个词典项可以共享同一个指针，这样每两个词典项就节省出了一个 4 字节长的指针信息，因为此时连续词典分块内包含多个单词，

为了能够标出其分隔位置，需要为每个单词增加单词长度信息，这样就可以在提取单词时对块内的不同单词予以正确区分。

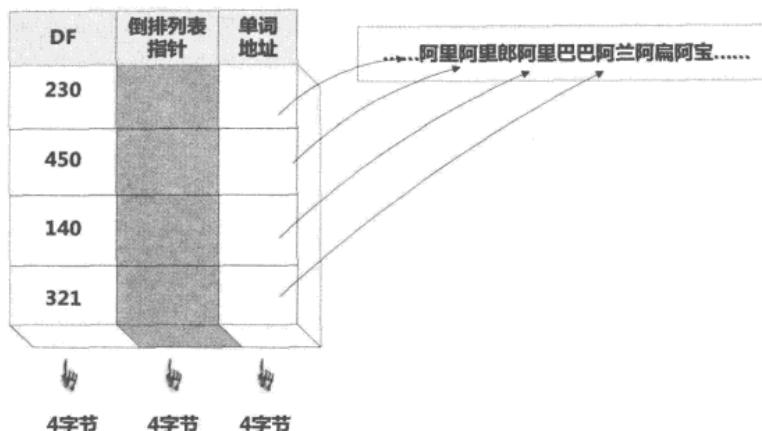


图 4-2 优化的词典结构

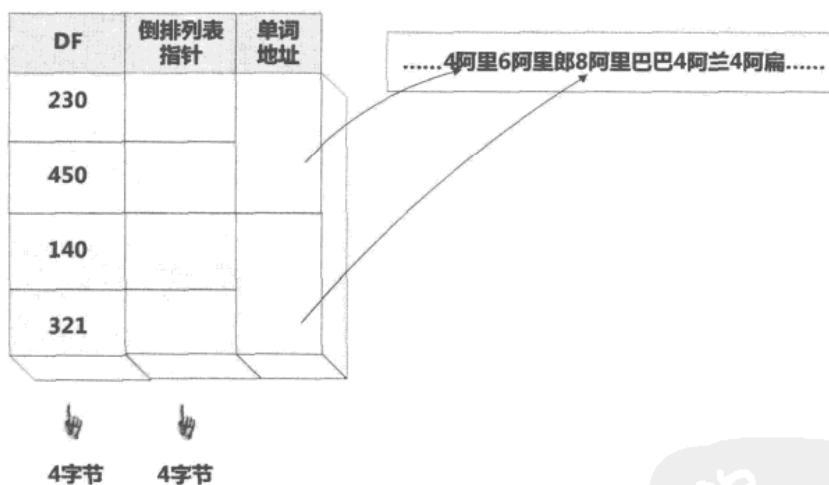


图 4-3 进一步优化的词典结构

实验表明，经过上述优化的词典比不做优化的词典占用内存数量减少了 60%，可见这些优化措施还是非常有效的手段。

## 4.2 倒排列表压缩算法

单词对应的倒排列表一般记载 3 类信息：文档编号、词频信息及单词位置序列信息。



因为文档编号及单词位置序列是依次递增的，所以通常的做法是存储其差值，而非原始数据。经过差值转换，文档编号和单词位置信息往往会被转换成大量的小整数，而词频信息大部分是小整数，因为一个单词在正文中出现的频率通常都不高。压缩算法的处理对象就是这 3 类信息，从以上描述可以看出，倒排列表数据有其特点，即数字分布严重不均衡，小数值占了相当大的比例。

#### 4.2.1 评价索引压缩算法的指标

目前有很多种倒排列表压缩算法可供选择，但是评判算法的优劣需要定量指标。一般来说，评价倒排列表压缩算法会考虑 3 方面的指标：压缩率、压缩速度和解压速度。

所谓压缩率，就是数据压缩前大小和压缩后大小的比例关系，很明显，压缩率越高，就越节省磁盘空间，同时也节省了倒排列表从磁盘读入到内存的 I/O 时间。

压缩速度是指压缩一定量的数据所花费的时间，相对而言，这个指标不如其他两个指标重要，因为压缩往往是在建立索引过程中进行的，而建立索引是一个后台运行过程，不需要即时响应用户查询，即使速度慢些也没有太大关系。另外，建立索引的次数相对而言也不算多，所以从几个方面考虑，压缩速度不是一个重要指标。

解压速度在 3 个指标中是最重要的，其含义是将压缩数据再次恢复为原始数据所花费的时间。因为搜索引擎在响应用户查询时，从磁盘读入的是压缩后的数据，需要实时解压以快速响应用户，所以解压速度直接关系到系统的用户体验，其重要性不言而喻。

#### 4.2.2 一元编码与二进制编码

一元编码（Unary Code）和二进制编码（Binary Code）是所有倒排列表压缩算法的基本构成元素，不论压缩算法内部逻辑思路是怎样的，最终都要以这两种格式来对数据进行表示。要么是以一元编码和二进制编码混合的方式，要么是单独以二进制编码的方式。可以认为这两种编码格式是压缩算法的基础构件。

一元编码是非常简单直观的数据表示方式，对于整数  $X$  来说，使用  $X-1$  个二进制数字 1 和末尾一个数字 0 来表示这个整数。图 4-4 是 1 到 5 这几个数字相对应的一元编码。可以看出，一元编码仅仅适合表示非常小的整数，比如对于数字 23000，如果用一元编码表示明显是很不经济的。

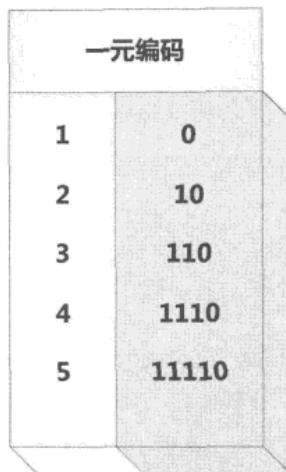


图 4-4 一元编码

二进制表示方式是计算机内部的标准数据存储方式，即由二进制数字 0 和 1 进行组合来表示实际的数据。不同的比特宽度代表了不同的数字表示范围，图 4-5 给出了不同的比特宽度能够表达的数值范围及一些对应的二进制表示示例。比如对于比特宽度为 3 的情形，即用 3 个比特位来表示数字，那么其能够表达的数值范围为 0 到 7 这 8 个数字；对于数字 5 来说，将 3 个比特位设置为二进制 101 的方式就能够获得表达，其他例子与此相同。在计算机内部，一般是以字节为单位的，即比特宽度为 8 的二进制编码就是一个字节。相对于一元编码来说，只要比特宽度足够长，那么二进制编码可以表示几乎任意范围的数字。

比特宽度	数字表示范围	二进制表示例子
1	2 (0-1)	0:<0> 1:<1>
2	4 (0-3)	0:<00> 1:<01> 2:<10> 3:<11>
3	8 (0-7)	1:<001> 3:<011> 5:<101>
4	16 (0-15)	1:<0001> 10:<1010>
5	32 (0-31)	1:<00001> 30:<11110>

图 4-5 二进制编码方式

了解了一元编码和二进制编码这两个压缩基础构件，我们下面介绍一些经典的倒排列表压缩算法。

### 4.2.3 Elias Gamma 算法与 Elias Delta 算法

Elias Gamma 压缩算法利用分解函数将待压缩的数字分解为两个因子，之后分别用一元编码和二进制编码来表达这两个因子。

Elias Gamma 算法采用的分解函数如下。

$$x=2^e+d$$

其中， $x$  为待压缩的数字， $e$  和  $d$  分别为其因子，得到分解因子后，对于因子  $e+1$  采用一元编码来表示，对于因子  $d$  采用比特宽度为  $e$  的二进制编码来表示。

假设待压缩的数字为 9，因为 9 可以分解为 2 的 3 次方加 1，所以  $e=3$  而  $d=1$ ，对  $e+1$  用一元编码后为：1110，而对  $d$  用比特宽度为 3 的二进制编码进行转换后为 001，将两者拼接获得：1110:001，此即为数字 9 对应的 Elias Gamma 编码。

Elias Delta 算法是建立在 Elias Gamma 算法基础上的改进，在概念上可以认为是利用两次 Elias Gamma 算法，将待压缩的数字分解为 3 个因子，之后利用一元编码和二进制编码来进行数值压缩。

对于待压缩数字  $x$ ，Elias Delta 算法根据分解函数，将其分解为因子  $e$  和因子  $d$ ，对因子  $d$  采用比特宽度为  $e$  的二进制编码进行压缩，而对于因子  $e+1$ ，则使用 Elias Gamma 算法再次进行压缩，即分解出另外两个因子。由此过程可以看出，Delta 算法等于使用了两次 Elias Gamma 算法，将待压缩数字分解为 3 个因子，然后分别编码获得最后的压缩数据。

我们仍然以数字 9 来说明 Elias Delta 的编码过程。根据分解函数可知， $e=3$  而  $d=1$ ，对  $d$  利用 3 比特宽度的二进制编码压缩为 001，对  $e+1$  即数字 4 采用 Elias Gamma 算法获得编码 110:00，将两者拼接获得最终压缩编码为 110:00:001。

从以上两个压缩算法的过程可以看出，Elias Delta 算法对于大数值来说压缩效果要优于 Elias Gamma 算法。

### 4.2.4 Golomb 算法与 Rice 算法

Golomb 算法和 Rice 算法在大的思路上也与上述的两个 Elias 算法类似，即根据分解函数将待压缩数值分解为两个因子，分别用一元编码和二进制编码来进行数据压缩表示。不同点在于，它们采用了和 Elias 算法不同的分解函数。

对于待压缩数值  $X$ ，Golomb 和 Rice 算法采用了如下的因子分解方式：

$$\text{因子 } 1 = (X-1)/b$$

$$\text{因子 } 2 = (X-1) \bmod b$$

因子 1 是将  $X$  减 1 后除以参数  $b$  向下取整后获得的整数，将其值加 1 然后采用一元编码压缩；而因子 2 则是对参数  $b$  进行取模运算，获得取值范围在 0 到  $b-1$  之间的  $X$  除以  $b$  后的余数，对因子 2 使用二进制编码进行压缩，因为取模运算决定了其取值范围一定在 0 到  $b-1$  之间，所以二进制编码的比特宽度设定为  $\log(b)$  就能表示这个范围内的数值。

对于 Golomb 算法和 Rice 算法来说， $b$  如何取值是其关键，这两个算法的不同点也仅仅在于  $b$  的不同取值方式。 $b$  的取值依据是什么呢？ $b$  的设定依赖于待编码数值序列的平均值或者中位数，在此基础上，Golomb 和 Rice 算法对  $b$  的设定采取了不同的策略，假设一个待压缩数值序列的平均值为 Avg，则 Golomb 算法设定为：

$$b=0.69 \times \text{Avg};$$

这里的 0.69 是个经验参数，而 Rice 算法则要求  $b$  一定要为 2 的整数次幂，同时  $b$  必须是所有小于平均值 Avg 的 2 的整数次幂的数值中最接近 Avg 的数值。假设 Avg 的数值为 113 的话，则  $b$  会被设定为 64，因为如果是 128，则超过了 113，如果是 32，则此值不是小于 113 且最接近 113 的数值，只有 64 符合这些条件。

图 4-6 给出了利用 Golomb 算法和 Rice 算法对待压缩数值序列进行压缩的示例。首先，获得数值序列的平均值 133，Golomb 算法设定  $b=78$ ，其二进制编码的比特宽度可以取 6 或者 7，即对在 0 到 77 范围内的数值，如果因子分解后因子 2 的值小于 50，则采取 6 比特宽度，如果因子 2 的值大于 50 则采取 7 比特宽度，这样可以有效减少压缩数据的大小；而 Rice 算法设定  $b=64$ ，其二进制编码的比特宽度取 6。当这些参数都可以确定下来后，即可对数值进行压缩编码，我们以数值 144 为例说明，首先对其减去 1，即需要对 143 进行压缩。对于 Golomb 算法来说，其对应的因子 1 取值为 1，数值加 1 后采用一元编码得到 10，因子 2 取值为 65，对其进行二进制编码得到 1000001，拼接两者获得最终编码 10:1000001。对于 Rice 算法来说，143 对应的两个因子分别为 2 和 15，对 2 加 1 后进行一元编码得到 110，对 15 进行二进制编码得到 001111，拼接两者得到最终编码 110:001111。

对于 Rice 算法来说，之所以要设定  $b$  为 2 的整数次幂，是为了在具体实现算法时能够采取掩码操作或者比特位移操作等快速运算方式，所以 Rice 在运算效率方面要高于 Golomb 算法。

而对设定好的参数  $b$  来说，其适用范围可以是局部的也可以是全局的，所谓全局的适用范围，就是说确定好  $b$  后所有单词对应的倒排列表都采用同一个数值；而所谓局部的适用范围，则是不同单词的倒排列表采取不同的参数  $b$ 。一般来说，如果索引非常庞大，则采取局部参数  $b$  效果更好，原因也很明显，因为局部参数  $b$  等于是对待编码序列根据局部分布情况进行自适应调整，这样会使得压缩效果更好。

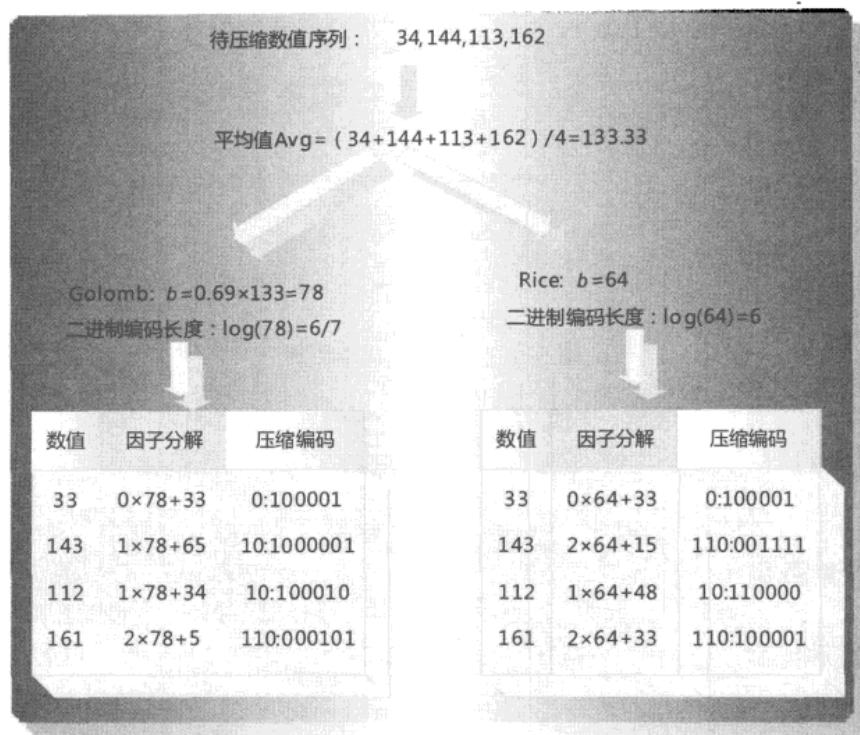


图 4-6 Golomb 算法和 Rice 算法示例

#### 4.2.5 变长字节算法 (Variable Byte)

变长字节算法以字节（即比特宽度为 8）为一个基本存储单位，而之前介绍的压缩算法都是变长比特算法，即以比特位（Bit）作为基本存储单位。之所以称之为“变长字节”，是因为对于不同的待压缩数字来说，在压缩编码后占用的字节数目不一定相同，可长可短，是变化的。

一个字节包含 8 个比特位，因为对于变长字节算法来说，每个数字压缩后的字节数目是变动的，为了确定两个连续数字压缩后的边界，需要利用字节中一个比特位作为边界判断符号，一般如果边界判断比特位设定为 0，则可以认为这个字节是数字压缩编码的最后一个字节，而如果设定为 1，则说明后续的字节仍然属于当前的压缩数据数字。所以每个字节在概念上被划分为两部分，其中一个比特位用来做边界判断；其他 7 个比特位采用二进制编码来存储压缩数据，即每个字节的数值表示范围为 128 个数字。

给定一个数值，如何对其进行变长字节压缩？图 4-7 给出了数值 33549 的变长字节压缩示例，因为一个字节可表示的数值范围为 0 到 127，所以按照 128 作为基数将待压缩数值进行分解，分解后的因子分别为 2、6 和 13，则在相应的字节存储这几个因子的二进制

编码，同时设置边界指示位置为 1、1 和 0，即说明这个数值需要 3 个字节来进行存储。

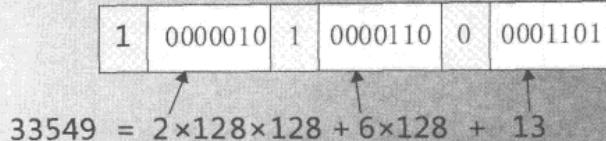


图 4-7 变长字节压缩示例

对于多个连续待压缩数值，则每个数值依照上例进行压缩编码，将其压缩编码连续存储即可。图 4-8 给出了多个连续数值序列对应的变长字节编码序列。

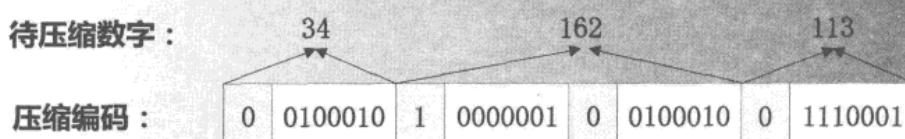


图 4-8 多数值的变长字节编码序列

#### 4.2.6 SimpleX 系列算法

SimpleX 系列算法最常见的是 Simple9，在此基础上有改进算法 Simple16、relate10 及 carryover12 等，本节我们主要介绍 Simple9 算法的计算思路。

Simple9 是一种字对齐算法，最常用的是利用 32 个比特位（即 4 个字节）来作为一个压缩单位，给定固定大小的压缩单位后，每个压缩单位试图存储多个待压缩的数字。图 4-9 给出了 Simple9 算法的字划分布局。

从图 4-9 可以看出，每个 32 位比特字被划分为两个组成部分，其中 4 个比特作为管理数据存储区，剩下 28 个比特作为压缩数据存储区，而压缩数据存储区根据实际使用情况，可以划分为 9 种布局类型（这也是 Simple9 的名字来源）。4 比特类型指示位用来指示后续的数据存储区属于哪种类型，而图中的 B 则指出了后续数据存储区的基本单元的比特宽度。

比如对于 B 为 1 的情况，说明后续数据存储区基本单元比特宽度为 1，数据存储区共有 28 个比特，所以数据区被划分为 28 个 1 比特宽度的基本单元构成。因为一个比特只能存储二进制数值 0 或者 1，即区分两种状况，所以这个 32 位比特字可以存储 28 个范围在 0

到 1 的数字。

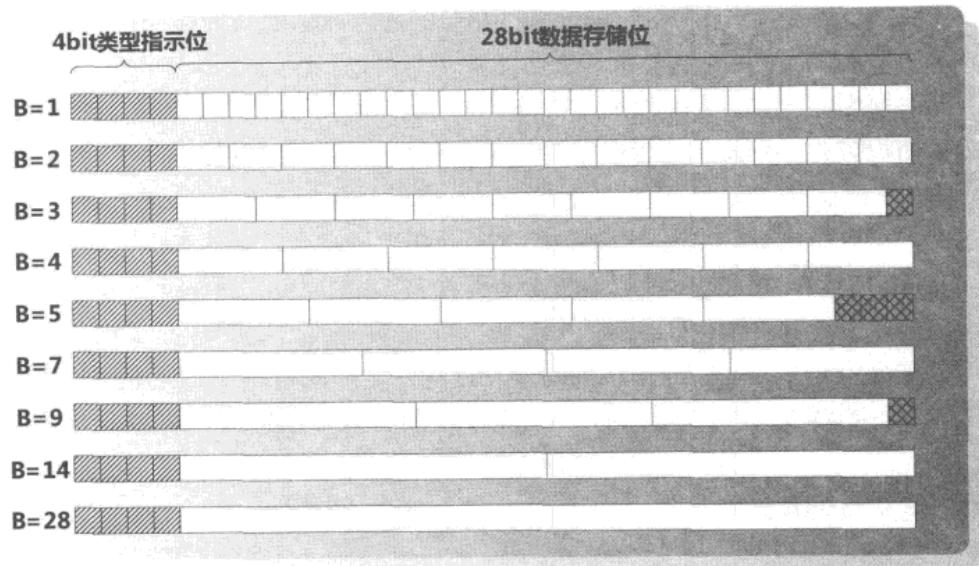


图 4-9 Simple9 的 9 种字划分布局

对于  $B$  为 2 的情况，数据存储区基本构成单元比特宽度为 2，即数据存储区由 14 个 2 比特位基本单元拼接而成，因为 2 比特位可以表达 4 个数字，所以这种类型的布局，在一个 32 位比特字中，可以存储 14 个范围在 0 至 3 的数字。

类似地，还可以根据  $B$  的不同大小设置其他的布局方式，图 4-9 给出了 9 种不同的布局，这里要注意的是：对于有些宽度的基本构成单元，因为不能被 28 整除，所以在数据存储区会有字节被废弃不用。比如对于  $B$  为 3 的情况，则最后一位比特位被废弃（图中有网线的单元指出了在各种情况下废弃的比特位）。

在利用 Simple9 对数据进行压缩时，首先读取待压缩数值队列后续的 28 个数字，如果发现这 28 个数字都是 0 或者 1，那么说明可以利用  $B=1$  这种布局来存储，则将 28 个数字编码为二进制后存入 28 个数据位，同时在类型指示位标出这种类型。如果发现有大于 1 的数值，说明  $B=1$  这个布局无法容纳这组数据，于是由读取后续 28 个数字改为读取后续的 14 个数字，判断是否能够使用  $B=2$  这种布局，即判断后续 14 个待压缩数据是否都在 0 到 3 的数值范围，如果是的话则采取  $B=2$  布局压缩，否则继续考虑  $B=3$  的布局，依次如此试探，即可将待压缩数字都表示为固定长度为 32 位比特的压缩表示方式。

利用 Simple9 进行解码的时候，因为编码是固定长度的，所以可以一次读取 4 个字节（即 32 比特），从 4 比特类型指示位判断后续数据存储区是哪种布局方式，对于每种布局方式可以事先准备好对应的掩码，判断出布局方式后，直接利用掩码可以一次性解压出后

续的多个压缩数字。

Simple16 等后续改进算法在 Simple9 基础上做出了改进，改进思路大致相同。首先，Simple9 的类型指示位由 4 比特位构成，即说明其可以指示 16 种不同情况，而 Simple9 因为只有 9 种类型，所以并未完全利用这 4 比特指示位置，可以将 9 种布局进行扩展，扩展为更多种布局方式，Simple16 就是将后续 28 位数据存储区划分为 16 种组成方式。另外一点，从图 4-9 可以看出，Simple9 在有些布局方式下会有废弃比特位，而这明显是对存储资源的浪费，可以考虑如何将其利用起来。Simple9 之所以会出现废弃位，是因为它对 28 位数据存储区是等宽度划分的，所以必然会有无法被 28 整除而出现的废弃位。Simple16 等改进算法则放宽了这种要求，即数据存储区不一定是等宽的，比如对于 Simple9 的  $B=5$  的状况，可以划分为 4 个 5 比特及后续的两个 4 比特，这样就可以全部利用存储区的位置，同时表达能力得到增强，即在压缩时，可以有更多种方式来进行压缩编码。

#### 4.2.7 PForDelta 算法

PForDelta 压缩算法是目前解压速度最快的一种倒排文件压缩算法，其基本出发点是：尽可能一次性压缩和解压多个数值。尽管 SimpleX 系列算法也利用了这一点，但 PForDelta 算法在这方面做得更好。

一次性压缩多个数值面临实际困难：因为连续的数值序列有大有小，如果每个数值按照序列中最大的数值来决定比特宽度，很明显对于小数值来说会存在空间浪费的情形，而如果不按照最大数值分配比特宽度，那么对于后续连续数值来说，因为比特宽度不够，存在有些大数值无法表示的问题。

PForDelta 希望能在压缩率和压缩解压速度方面找到一个平衡点，使得算法可以一次压缩解压多个数值。PForDelta 的折中方案是：对于待编码的连续  $K$  个数值（一般  $K$  取 128，即一次性压缩解压 128 个数值），找出其中 10% 比例的大数，根据剩下 90% 的数值范围来决定应该采取的比特宽度，而 10% 的大数则当做异常数据单独存储，即采取了“照顾大多数”的原则，对于少数情况做特殊处理。

下面以具体实例来看看 PForDelta 算法的静态结构、压缩过程及解压过程。假设一次性要压缩的数值序列为：24,40,9,13,31,67,19,44,22,10，即  $K$  取 10，另外为了方便说明问题，这里取 30% 的大数作为异常数，而在真实场景  $K$  往往取 128，大数比例一般取 10%。图 4-10 是 PForDelta 算法压缩以上数据序列后形成的静态结构，压缩后的数据可以划分为 3 块：异常数据存储区、常规数据存储区和异常链表头。

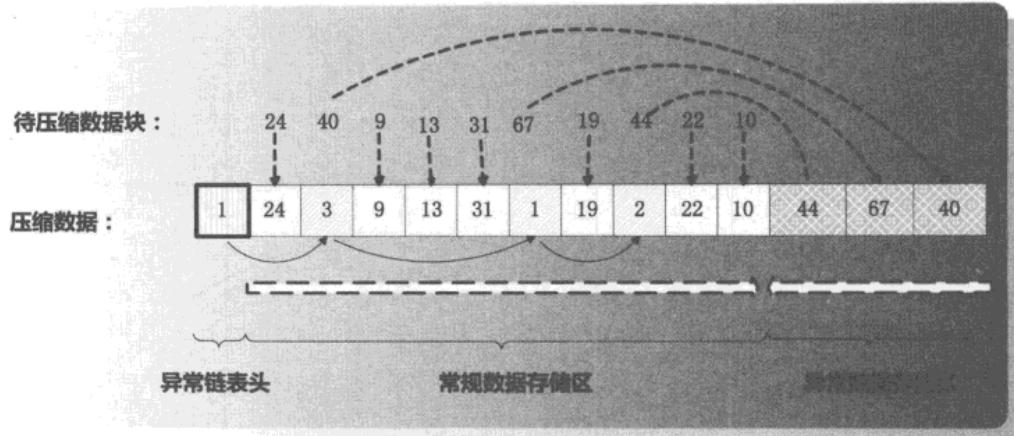


图 4-10 PForDelta 算法压缩数据的静态结构

异常数据存储区存储的是数值序列中 30% 比例的大数，这些异常数值不做压缩编码，每个数值用 4 个字节来存储，并放置在静态结构的尾端，存储顺序与其在原始数值序列的出现顺序相反。

常规数据存储区则存储了 70% 的小数及一个异常数据链表结构，在上述例子中，除了 30% 的 3 个大数外，最大的数值为 31，所以为了能够表示这 70% 的数值，比特宽度需要设定为 5，因为每个数值利用 5 个比特位存储，那么可存储的最大数值为 31，这样就保证这 70% 的数值都可以以等长的比特宽度压缩存储。尽管 30% 的大数已经被存储在尾端，但是为了能够记载这些数在原始序列中的位置信息，PForDelta 在常规数据存储区维护了异常大数的一个顺序链表，比如对于异常数值 40，其常规数据存储区对应位置设定为 3，含义是指跳过后面 3 个数值即是另外一个异常大数的位置，同理，数值 67 的对应位置设定为 1，意即跳过后面 1 位数值即是后续的异常大数位置，通过这样就将异常大数的位置信息保留下来，这样在解压的时候能够快速恢复原始数据。

异常链表头则存放一个指针，指向了异常链表的第一个数值的位置，本例中链表头的值为 1，意即跳过后续一个常规数值即可获得第一个异常大数的位置，通过链表头，就可以将所有异常大数快速串联起来。

以上介绍的是 PForDelta 算法的静态压缩结构，那么给定上述 10 个待压缩数据流，是如何形成最终的压缩结构的？PForDelta 采取了 3 个步骤来形成压缩结构，图 4-11 展示了这 3 个连续的步骤。

第 1 步，根据待压缩的数据流，确定 30% 比例的大数个数及分别是哪些数值，在此例中依次为 40、67 和 44，根据剩下的 70% 数值大小，可以确定压缩采取的比特宽度为 5，此时即可确定静态结构所需存储区的大小。

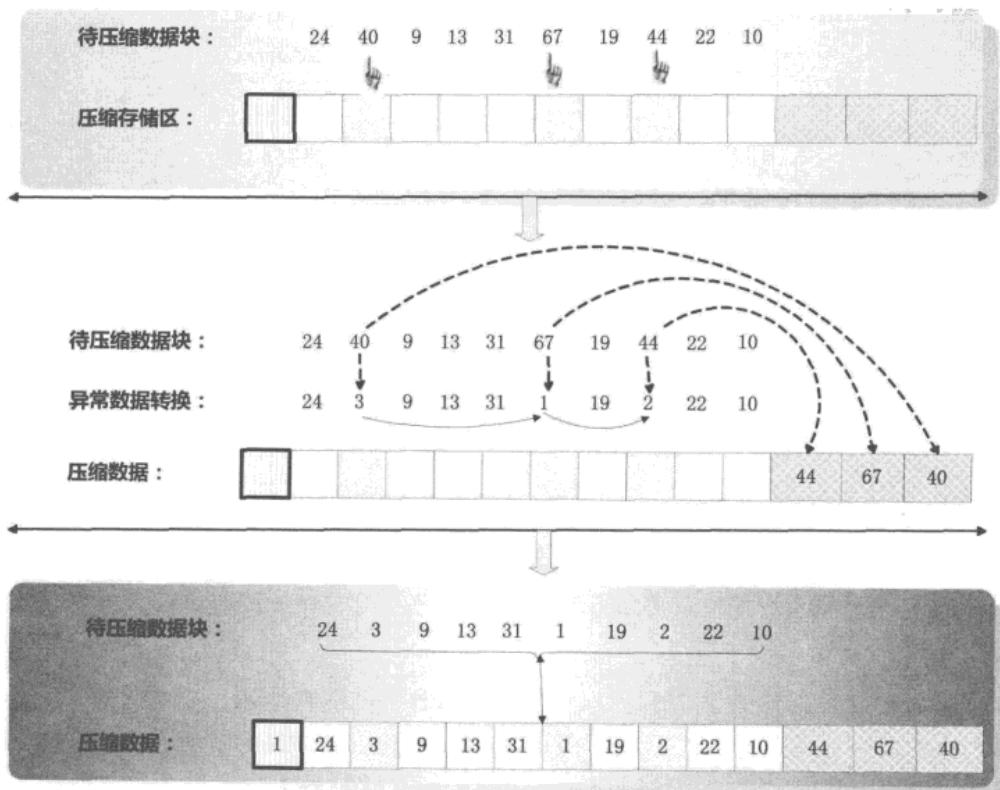


图 4-11 PForDelta 的压缩过程

第 2 步，对原始数据循环遍历，本步骤主要处理异常大数，一方面将遇到的大数逆序放置到静态数据尾端；另一方面将原先的异常大数转换成链表结构，所以 3 个异常大数对应位置被置换成 3、1 和 2，形成了链表结构。

第 3 步，一次性将所有 10 个数值快速压缩，然后存入常规数据存储区中。经过第 2 步后，可以保证 10 个待压缩的数据都在 32 以下，如果是这样就可以采取措施快速压缩。

通过以上 3 个步骤，即可对多个数值成功地进行一次性压缩，至于解压，则可以理解为压缩过程后面两个步骤的逆过程，即通过两遍完成。首先，一次性将常规存储区的 10 个数值进行解压，然后根据异常链表头，依次顺序读出异常大数的位置，结合尾部存储的异常大数恢复原始数值序列。

多项实验表明，PForDelta 的解压速度是现有压缩算法里最快的，而且性能大幅度超过其他算法。主要原因是其能够一次性解压多个数据，尽管恢复异常数据速度比较慢，但是因为其所占比例很小，所以影响不大；另一点，PForDelta 之所以将解压设置为两遍方式，而非一遍做完，是为了在程序中避免做 IF-ELSE 这种分支判断，因为大量分支判断严重影响



响解压执行速度。比如 Simple9 算法解压，每 4 个字节就需要做一次分支判断，因为要判断压缩数据是 9 种布局中的哪一种；而变长字节算法解压则每个字节都需要做分支判断，以决定当前字节是否是某个数字的最后一个字节，相对应地，PForDelta 在其解压流程中是不做任何分支判断的，这也是其速度快的重要原因之一。

### 4.3 文档编号重排序 (DocID Reordering)

对于搜索引擎来说，在建立索引的过程中，要对每个网页赋予唯一的文档编号（文档 ID），在搜索引擎内部，都以文档 ID 来标记某个网页，以方便搜索引擎内部处理。常见的做法是随机赋予某个网页一个编号，比如可以根据爬虫抓取网页的时间先后顺序来依次顺序编号。

索引中某个单词的倒排列表，一般会记录包含这个单词的所有网页的文档 ID 和单词词频信息，索引压缩技术可以有效压缩这些数值信息，以增加搜索引擎效率。对于索引压缩技术来说，如果需要压缩的原始数据的值越小，压缩效果越好，之前介绍的倒排列表中文档 ID 采用 D-Gap（文档差值编号）进行编码就是出于这种考虑。

文档编号重排序技术考虑在文档编号上做文章，不是随机赋予网页一个文档 ID，而是通过对文档 ID 进行更加合理的编号，使得在倒排列表中相邻的两个文档其编号也尽可能相邻，这样可以使相邻文档的 D-Gap 值尽可能小，而对小数值采用压缩算法效率会更高。所以，这个技术的核心目的是：重新编排文档的编号，以使得某个单词倒排列表中相邻的两个文档其文档编号尽可能相近。

为了达到这个目的，我们希望内容越相似的网页，在编排文档编号时，其文档编号越相邻。互联网包含了海量的网页数据，但是这些网页并非毫不相干，很多网页讲述的内容主题相同，这些内容相似的网页所使用的词汇有很大一部分是重叠的，这些相同的单词表达了这批网页所代表的主题。如果能够将这些内容相似的网页依次编号，那么对于表达主题内容的某个单词来说，包含这个单词的大部分网页的文档编号就是相近的，即其倒排列表中的文档编号相邻，以此达到该技术的核心目标。

下面以具体例子来说明文档编号重排序的基本技术思路（参考图 4-12）。

假设搜索引擎爬虫抓取了 5 个互联网页面，并随机赋予网页一个编号，得到以下网页集合。

原始文档编号	网页内容
1	谷歌确认赫利将辞去 YouTube CEO 职位
2	Facebook 或将推出社交签到功能 Deals

(续表)

原始文档编号	网页内容
3	YouTube 创始人兼 CEO 称将逐步退居幕后
4	用 Facebook Places 到歌手广告牌签到
5	YouTube CEO 赫利卸职转做咨询岗位

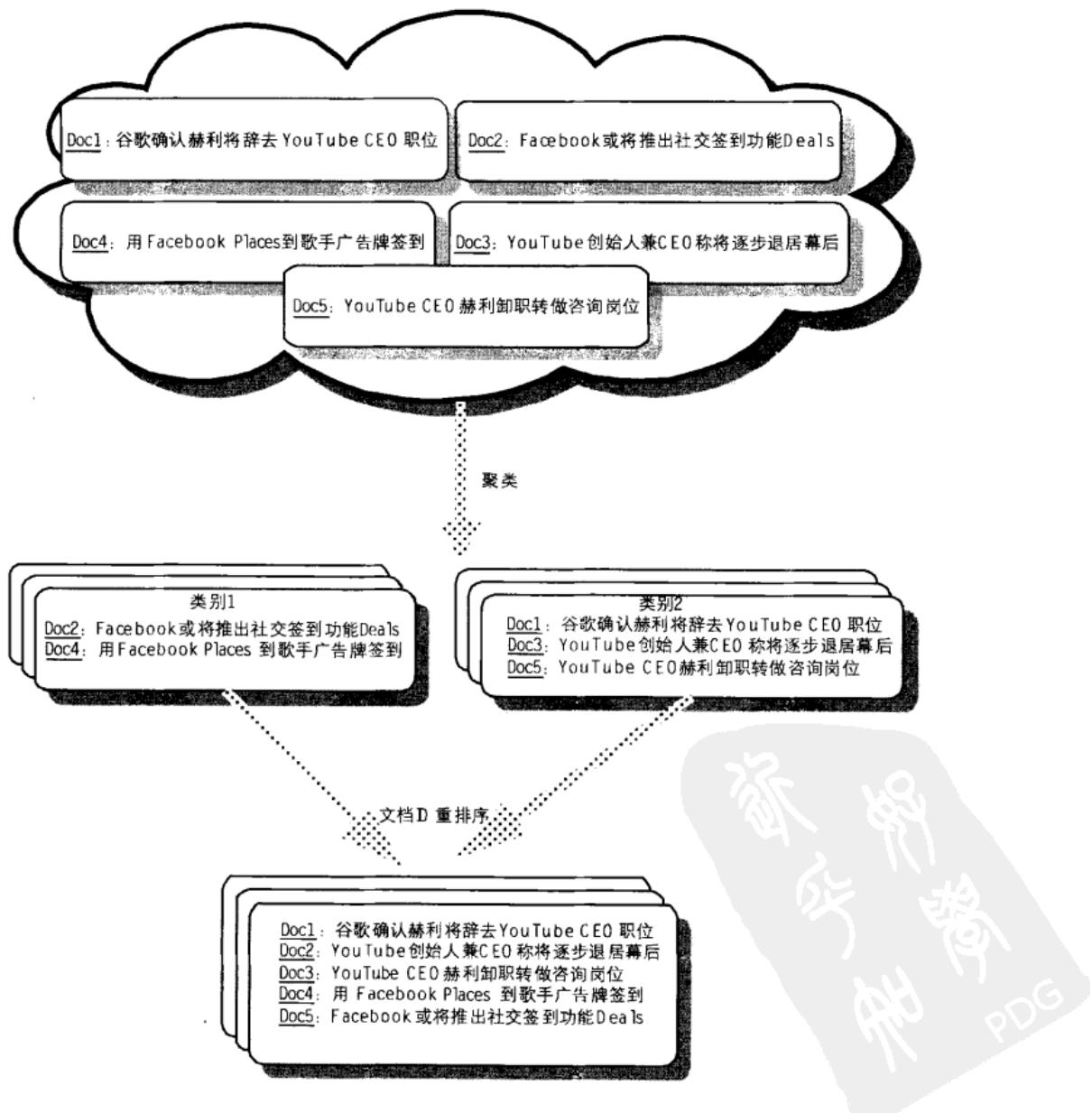


图 4-12 文档编号重排序

从网页内容可以看出，文档 1、文档 3 和文档 5 讲述相似的内容，同样，文档 2 和文档 4 也是如此。可以采用文本聚类的方法，将内容相似的网页聚合在一起，在这个例子里，聚类结果有两个，分别讲述了 YouTube 和 Facebook 公司的新闻事件。

接下来，搜索引擎将重新赋予文档编号，基本原则是内容相似的网页赋予相邻的文档编号，于是网页获得了如下新的文档编号。

新的文档编号	网页内容
1	谷歌确认赫利将辞去 YouTube CEO 职位
5	Facebook 或将推出社交签到功能 Deals
2	YouTube 创始人兼 CEO 称将逐步退居幕后
4	用 Facebook Places 到歌手广告牌签到
3	YouTube CEO 赫利卸职转做咨询岗位

通过以上的文档重新编号，对索引内容有何影响？图 4-13 给出了文档重新编号前后索引差异的图示。这里需要注意的是：为了便于说明问题，倒排列表只包含了文档 ID，另外文档 ID 是以 D-Gap 的方式存储的。

我们以单词“YouTube”为例，看看文档重新编号前后索引内容有何变化。在文档重新编号之前，文档 1、文档 3 和文档 5 中包含了“YouTube”，所以这个单词对应的倒排列表中应该记录的文档 ID 为{1,3,5}，由于采用 D-Gap 方式，对应的数值为相邻文档编号的差值，即{1,2,2}。

经过文档编号重排序，原先的文档 3 重新编号为文档 2，原先的文档 5 重新编号为文档 3，按照 D-Gap 方式，单词“YouTube”对应的倒排列表为{1,1,1}，即原先的文档编号差值 2 都变为了差值 1。索引中的其他单词倒排列表变化也都类似。通过这种方式，就将倒排列表中需要记录的文档编号有效地减小，在此基础上运用索引压缩算法，可以获得更高的压缩率。

以上介绍的方法基本体现了文档编号重排序技术的基本思想和流程，在这个流程里，文本聚类是个很重要的组成部分，但是互联网网页数据量非常大，面对海量数据，文本聚类的速度难以满足实际需求。相关研究人员提出了一些启发规则来改善这种状况，一种非常有效的启发规则是：将页面 URL 相似的网页聚合在一起。这里主要考虑到，同一个网站的很多页面表达的主题内容是近似的，通过这种方式可以非常高效地将网页聚合在一起，之后根据聚类结果对文档 ID 进行重新编号（参考图 4-14 所示），这样就可以高效地完成文档编号重排序任务。

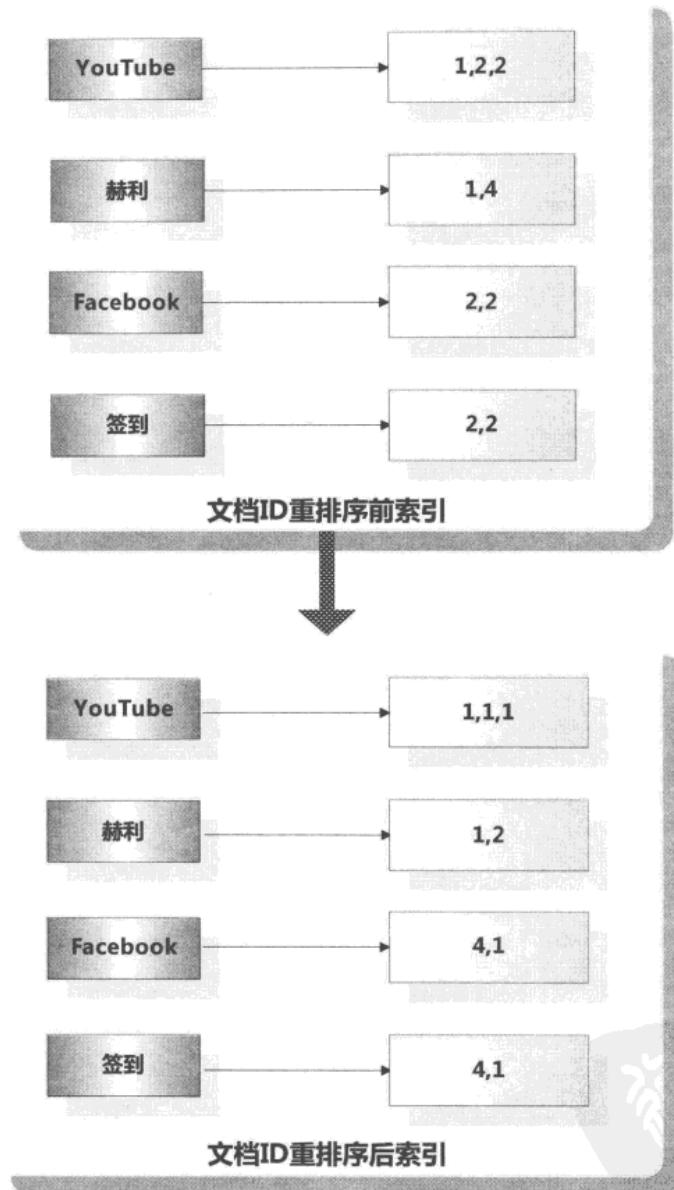


图 4-13 文档 ID 重排序后索引内容的变化

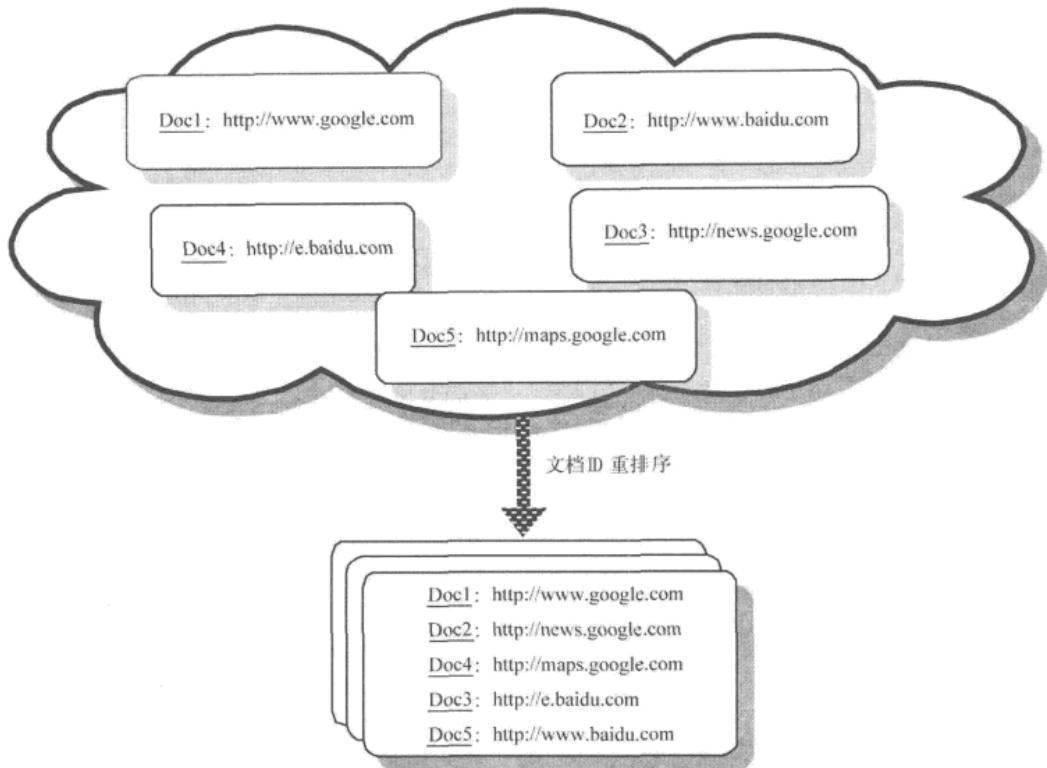


图 4-14 根据网页 URL 进行文档 ID 重排序

#### 4.4 静态索引裁剪 ( Static Index Pruning )

本章前述压缩算法都属于无损压缩，即数据在压缩前后没有任何丢失，本节要讲述的静态索引裁剪则属于有损压缩，通过主动抛弃一部分不重要的信息来达到更好的数据压缩效果。

静态索引裁剪的出发点是基于如下考虑：对于用户查询来说，搜索系统往往只需要返回相关性最高的  $K$  个网页，不需要将所有相关网页都呈现给用户。既然如此，对于词典中的某个单词，其对应的倒排列表中的索引项，对计算网页和用户查询最终相关性得分的贡献是不同的，有的对于计算最终得分很重要，有些则不是那么重要。所以可以将那些不重要的索引项从倒排索引中清除掉，只保留重要的索引项，这样就有效地减少了索引的大小，同时尽可能保证搜索质量，使得用户基本察觉不到索引项是不完整的。

静态索引裁剪在以上出发点的基础上，在具体设计算法时，大体有两种不同的思路，一种被称为以单词为中心的索引裁剪，一种被称为以文档为中心的索引裁剪。

#### 4.4.1 以单词为中心的索引裁剪

对于某个单词来说，其对应的倒排列表记载了出现这个单词的文档编号信息，以单词为中心的索引剪裁，其剪裁对象就是单词对应倒排列表中的文档。

假设我们已经有了一个相似性计算函数  $g(\text{term}, \text{doc})$ ，这个函数可以计算倒排索引中某个单词和其对应倒排列表某个文档的相似性（详情请参考本书第5章），那么就可以利用这个函数的计算结果来作为是否保留某个索引项的评判标准。

图4-15给出了以单词为中心的索引裁剪示意图。对于词典中的单词“搜索引擎”来说，其倒排列表中包含4个文档：文档1、文档3、文档4和文档7。函数 $g$ 用来计算文档和单词“搜索引擎”的相关性，通过函数 $g$ 分别得出对应的得分，其中文档1的得分为0.3，文档2的得分为0.25，文档3的得分为0.4，文档7的得分为0.18。

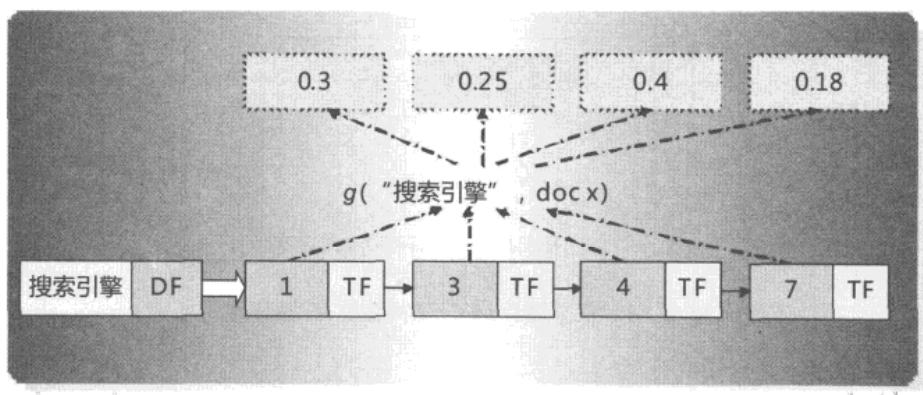


图4-15 以单词为中心的索引裁剪

一种直观的裁剪方法是设定一个相似性得分阈值，只要通过函数 $g$ 计算的得分低于这个阈值，则清除掉这个索引项，如果得分高于这个阈值，则保留索引项。比如我们设定阈值 $b$ 为0.35，则对于“搜索引擎”这个单词来说，其倒排列表中的索引项只有文档4被保留，其他3个索引项被清除，这样就有效地减少了倒排索引的大小。同时，因为保留了相关性得分较高的文档索引项，所以对搜索前列结果没有大的影响，不会损害用户体验。

但是这个直观方法人为设定了一个全局阈值 $b$ ，而这个阈值不考虑词典中单词的具体情况，统一使用同一设定值，这样做会带来问题，假设我们设定阈值 $b$ 为0.45，那么对于“搜索引擎”这个单词来说，其所有索引项都会被清除掉，这并不是我们希望看到的，因为如果有用户以“搜索引擎”作为查询的时候，搜索系统的返回结果为空，而全局性阈值总是难免将某些单词的索引项全部清除。

为了解决以上问题，可以增加一个约束条件：每个词典中的单词，其对应的倒排列表



中都应该至少保留  $K$  个索引项，这个  $K$  可根据具体情况来设定，比如可以设定其为搜索引擎的默认返回结果条数。增加以上约束后，可以采用更加复杂和灵活的策略来对索引进行剪裁，我们以如图 4-16 所示的具体例子来说明这个新的剪裁方式。在这个例子中，我们考虑词典中的两个单词，一个是单词“搜索引擎”，其倒排列表和上文所述例子一致；另一个单词“技术”，其倒排列表包含两个索引项：文档 2 和文档 3。另外，这里设定  $K$  为 2，即搜索引擎默认输出两个结果。需要说明，实际搜索引擎的这个数值一般较大，这里的设定是为了方便说明问题。

新的裁剪策略采取如下方式：每个单词至少保留  $K$  个索引项，如果本身索引项不足  $K$  个，那么就全部保留，对于这些单词的倒排列表不做裁剪。比如图中的单词“技术”，因为只有两项，所以不做裁剪。

对于多于  $K$  个索引项的单词，首先用  $g$  函数对每个文档和单词的相关度进行打分，根据得分对文档由高到低排序，因为我们至少要保留  $K$  项，所以得分排在前  $K$  个的索引项是需要保留的项目，但是只保留  $K$  项还不够，系统还需要留些富余项目，即那些分数比第  $K$  项得分稍低些的项目也可以保留下，不做裁剪。为了达到这个目的，可以对第  $K$  个的得分进行打折，即乘上一个折扣因子，得出一个阈值  $a$ ，剩下的项目，如果得分高于这个阈值，则保留，低于这个阈值，则裁剪掉。

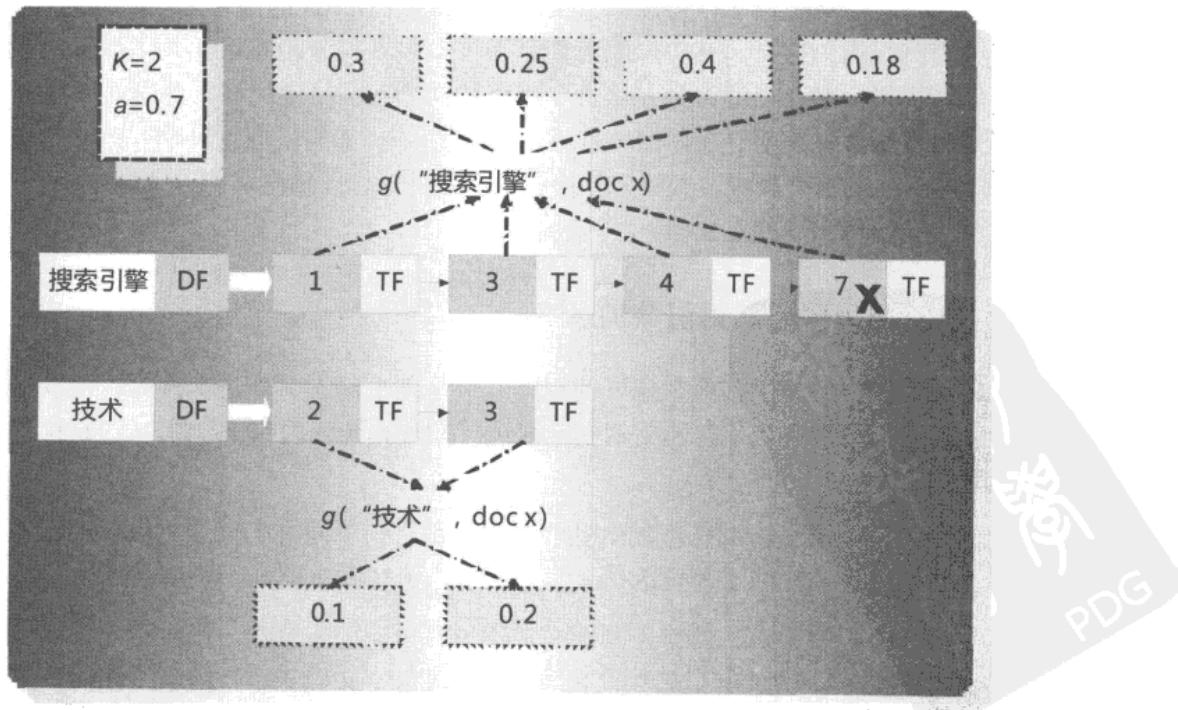


图 4-16 理想的索引裁剪方式

在图 4-16 中，我们来看看如何裁剪单词“搜索引擎”对应的倒排列表项。首先根据函数  $g$  对每个索引项的重要性打分，每个文档对应的分数如图 4-16 所示，其得分由高到低分别为：文档 4、文档 1、文档 3 和文档 7。因为  $K$  被设定为 2，所以可以确定保留的是文档 4 和文档 1，文档 1 的得分为 0.3。进一步，为了能够留下富余项，考虑对 0.3 进行折扣，设定折扣因子  $a$  为 0.7，由此得出阈值 0.21，因为文档 3 得分高于阈值，所以保留，文档 7 的得分低于阈值，所以被裁剪掉。对于词典中每个单词都按照以上规则进行裁剪，则完成了整个索引的静态裁剪过程，得到一个缩水版的索引。

为什么要在这里保留富余索引项呢？我们还是以图 4-16 来说明其原因。如果不做富余项保留，可以看到，对于单词“搜索引擎”来说，只保留文档 4 和文档 1，文档 3 会被裁剪掉。假设用户输入的查询为“搜索引擎 技术”两个单词，从其各自得分情况可以看出，对于这个查询来说，文档 3 是综合得分最高的，应该排在搜索结果头条，但是如果将其索引项裁剪后，其排名将大幅下降，如果是其他例子，甚至可能导致本应被搜索到的结果无法搜索出来。所以保留富余索引项是出于保证复杂查询的搜索质量考虑的。

以上是一个灵活的“理想化”静态索引裁剪过程，之所以说是理想化的，是因为实验结果证明，这个方法裁剪效果不佳，往往裁剪不了多少索引项。后续实验证明，如果首先对所有索引项的原始得分减去得分最低索引项的得分（在图 4-16 中，对于“搜索引擎”这个单词来说，就是所有索引项得分减去文档 7 的得分 0.18），之后再采取如上的裁剪过程，则效果会得到很大的提升。其中原因可能是因为索引项得分相差不大，比较集中在某个区间，所以减去得分最低项的得分，等价于将分数评价标准精度提高，此时对于原先细微的得分差异也可以进行裁剪。在如图 4-16 所示的例子中，如果采取这种调整策略，单词“搜索引擎”对应的倒排列表中，文档 3 会被裁剪掉，而之前理想化裁剪方案则是予以保留的，所以这种方式的压缩率相对大些。

#### 4.4.2 以文档为中心的索引裁剪

以单词为中心的索引裁剪在建好的倒排索引基础上对索引项进行删除，而以文档为中心的索引剪裁则可以认为是在建立索引之前进行的数据预处理措施。图 4-17 是这种裁剪方式的示意图。

我们知道，尽管一篇文档包含很多单词，但是这些单词对于表达文档主题作用是不一样的，有些单词对于文档主题表达是必不可少的，而有些单词则不是那么重要。

以文档为中心的索引裁剪首先判断文档所包含单词的重要性，经过一定的重要性得分计算，保留重要单词，抛弃不重要的单词。首先对每个文档先进行缩水，然后在此基础上建立倒排索引，这同样能够达到减小索引大小的目的。不过与以单词为中心的索引裁剪相

比，这种方式对于有些单词来说，其倒排列表为空，比较极端但是直观的例子就是停用词，比如“的”等单词，因为不能表达意义，这些单词在绝大多数情况下会被裁剪掉，导致索引项为空，而以单词为中心的索引裁剪则能控制索引项个数，以保证单词的索引项不空，所以是比以文档为中心的索引裁剪更常用的方法。

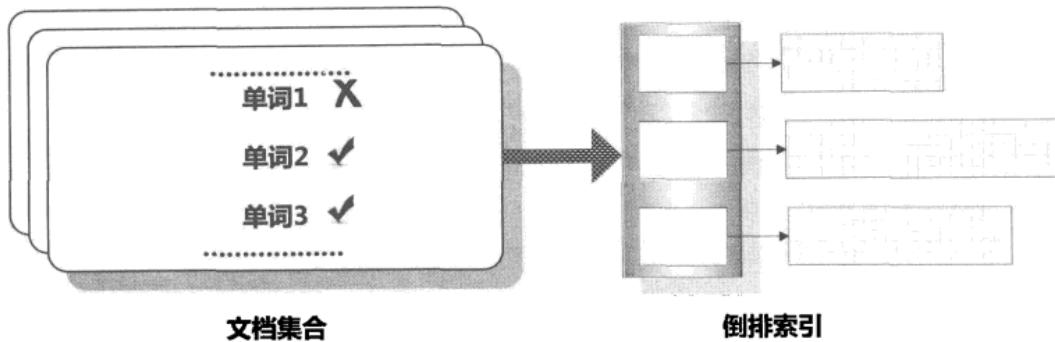


图 4-17 以文档为中心的索引裁剪

## 本章提要

- 搜索引擎索引压缩包括对词典的压缩和对倒排列表的压缩。
- 针对倒排列表的压缩可分为无损压缩和有损压缩两种，无损压缩更常用些，有损压缩只在特殊场合使用。
- 一元编码和二进制编码是所有倒排列表压缩算法的基本构件，不论具体压缩算法如何，最终都要落实到这两种方法上。
- 常用的压缩算法包括：Elias Gamma 算法、Elias Delta 算法、Golomb 算法、Rice 算法、变长字节算法、SimpleX 系列算法和 PForDelta 算法，在实际使用中往往混合采用不同算法来获得更好的压缩效果。
- 文档 ID 重排序通过文档聚类并重排文档 ID 编号来获得较高的索引压缩率。
- 静态索引裁剪是一种有损压缩算法，通过抛弃一部分不重要的索引项来获得较好的压缩效果。

## 本章参考文献

- [1] Anh, V. and Moffat, A.(2004). Index compression using fixed binary codewords. In Proc. of the 15<sup>th</sup> Int. Australasian Database Conference, 61–67.

- [2] Anh, V. and Moffat, A.(2005). Inverted index compression using word-aligned binary codes. *Inf. Retrieval*, 8(1):151–166.
- [3] Blanco, R. and Barreiro, A.(2005). Document identifier reassignment through dimensionality reduction. In Proc. of the 27<sup>th</sup> European Conf. on Information Retrieval, 375–387.
- [4] Heman, S. (2005). Super-scalar database compression between RAM and CPU-cache. MS Thesis, Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands.
- [5] Scholer, F., Williams, H., Yiannis, J., and Zobel, J.(2002). Compression of inverted indexes for fast query evaluation. In Proc. of the 25<sup>th</sup> Annual SIGIR Conf. on Research and Development in Information Retrieval, 222–229.
- [6] Mo, A. and Stuiver, L. (2000). Binary Interpolative Coding for Effective Index Compression. *Information Retrieval*, 3(1):25- 47.



# 第5章 检索模型与搜索排序

“得意贤士不可不举，不得意贤士不可不举，尚欲祖述尧、舜、禹、汤之道，将不可以不尚贤。夫尚贤者，政之本也。”

墨子《尚贤上第八》

搜索结果排序是搜索引擎最核心的构成部分，很大程度上决定了搜索引擎的质量好坏及用户接受与否。尽管搜索引擎在实际结果排序时融合了上百种排序因子，但最重要的两个因素还是用户查询和网页的内容相关性及网页链接情况，关于网页链接分析算法在本书第6章有详述，本章主要介绍的是：给定用户搜索词，如何从内容相关性的角度对网页进行排序。

判断网页内容是否与用户查询相关，这依赖于搜索引擎所采用的检索模型。关于检索模型的研究，从信息检索学科建立之初就一直是研究重点，到目前为止，已经提出了多种各异的模型，本章将介绍其中最重要的几种检索模型：布尔模型、向量空间模型、概率模型、语言模型及最近几年兴起的机器学习排序算法。

尽管检索模型多种多样，但其在搜索引擎中所处的位置和功能是相同的，图5-1给出了一个搜索引擎计算内容相似性的框架。当用户产生了信息需求后，构造查询词，以此作为信息需求的具体体现，搜索引擎在内部会对用户的查询词构造内部的查询表示方法。对于海量的网页或者文档集合，对每个文档，在搜索系统内部也有相应的文档表示方法。搜索引擎的核心是判断哪些文档是和用户需求相关的，并按照相关程度排序输出，所以相关度计算是将用户查询和文档内容进行匹配的过程，而检索模型就是用来计算内容相关度的理论基础及核心部件。

什么样的检索模型是个好模型呢？用户发出查询词Q后，我们可以把要搜索的文档集合按照“是否相关”及“是否包含查询词”两个维度，将其划分为4个象限（参见图5-2），其中，第一象限的文档出现了用户查询词同时被用户判定为相关的；第二象限的文档不包含用户查询词但是被用户判断为相关的；第三象限的文档出现了用户查询词但被用户判定为不相关的；而第四象限的文档则是不包含用户查询词且被用户判断为不相关的。一个好

的检索模型应该做到的是：在排序结果中，提升第一象限和第二象限文档的排名，抑制第三象限和第四象限文档的排名。

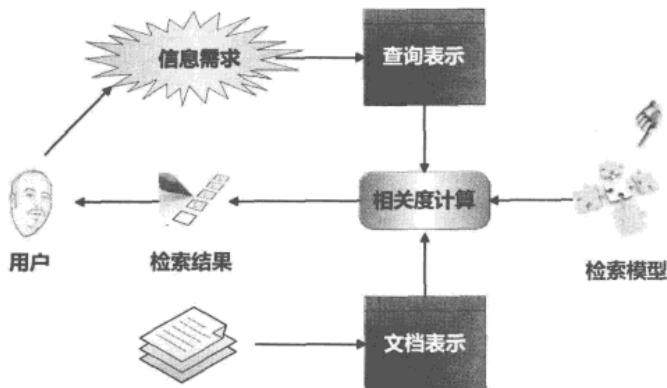


图 5-1 内容相似性计算框架

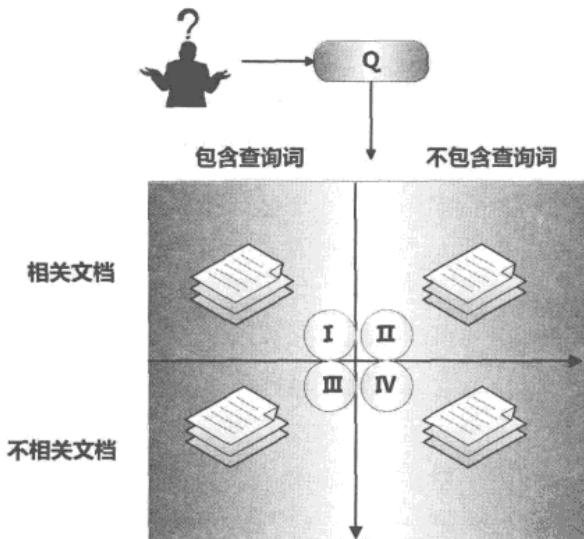


图 5-2 文档划分的 4 个象限

读者看图 5-2 可能会产生疑问：如果文档不包含查询词，会是相关的搜索结果吗？答案很明显是肯定的，比如用户搜索“电脑”，如果一个文档没有出现过这个单词，但是出现了“PC”或者“计算机”，那也是相关的。至于处理此种情况，往往是查询扩展技术或者语义搜索讨论的范围，目前绝大多数检索模型考虑的文档对象主要是第一象限和第三象限的对象，即一定出现搜索词的文档。

这里需要注意的一点是：检索模型理论研究都存在理想化的隐含假设，即假设用户需



求已经通过查询非常清晰明确地被表达出来，所以检索模型的任务不牵扯到对用户需求建模。很明显这与事实相距甚远，即使是相同的查询词，不同用户的需求目的可能差异很大，而检索模型对此无能为力。所以可以将检索模型看做是：在用户需求已经很明确地由查询词表征的情况下，如何找出内容相关的文档。如果查询词不能精确代表用户真实需求，那么无论检索模型再优秀也无济于事，这是为何搜索引擎发展到目前阶段，重点转向了填补用户真实需求和发出的查询词之间的鸿沟的原因，此发展趋势有其必然性。

## 5.1 布尔模型（Boolean Model）

布尔模型是检索模型中最简单的一种，其数学基础是集合论。在布尔模型中，文档与用户查询由其包含的单词集合来表示，两者的相似性则通过布尔代数运算来进行判定。

用户查询一般使用逻辑表达式，即使用“与/或/非”这些逻辑连接词将用户的查询词串联，以此作为用户的信息需求的表达。比如用户希望查找和苹果公司相关的信息，可以用以下逻辑表达式来作为查询：

苹果 AND (乔布斯 OR iPad2)

其代表的含义是：如果一篇文档包含单词“苹果”，同时也包含单词“乔布斯”或者“iPad2”两者其中的任意一个，那么这个文档就是满足用户需求的。所以对于布尔模型来说，满足用户逻辑表达式的文档就算是相关的。

我们以具体例子来说明，图 5-3 是一个文档集合，其中包含 5 个文档，每个文档的内容在图 5-3 中都已标出。为了简单起见，我们只考虑“苹果”、“乔布斯”和“iPad2”这 3 个单词，在搜索系统内部，单词—文档矩阵如图 5-4 所示，其代表的含义是单词出现在哪些文档中，比如对于单词“苹果”来说，文档 D2、D3、D5 包含这个单词，在图中以对钩来表达这一状况，其他单词文档关系与此类似。

D 1: 乔布斯最新近照曝光略显消瘦。
D 2: 分析师称苹果今年仍将占据大多数触摸屏产能。
D 3: 苹果公司首席执行官史蒂夫·乔布斯宣布，iPad2 将于 3 月 11 日在美国上市。
D 4: 乔布斯推动了世界，iPod、iPhone、iPad、iPad2，一款一款接连不断。
D 5: 乔布斯出门买了四袋苹果。

图 5-3 文档集合

		单词—文档矩阵				
		D1	D2	D3	D4	D5
关键词	苹果		✓	✓		✓
	乔布斯	✓		✓	✓	✓
	iPad2			✓	✓	

Q=苹果 AND ( 乔布斯 OR iPad2)

图 5-4 单词—文档矩阵

假设此时搜索系统接收到了用户查询：“苹果 AND ( 乔布斯 OR iPad2 )”，搜索系统会检查单词—文档矩阵，找出满足这个逻辑表达式的文档。对于逻辑表达式“( 乔布斯 OR iPad2 )”来讲，其含义是：文档需要包含单词“乔布斯”或者单词“iPad2”中任意一个。从单词—文档矩阵可以看出，满足这个逻辑条件的文档包括 D1、D3、D4、D5，而包含“苹果”这个单词的文档集合是 D2、D3、D5。按照用户查询的要求，两个集合求交集，得出 D3、D5。这两个文档就是满足整个逻辑表达式的文档，将其输出即为搜索结果。

由上可看出，布尔模型简单直观，但是正是因为其简单性，导致一些明显的缺点。从上面例子可以看出，只要文档满足逻辑表达式，就会被认为是相关的，其他文档被认为是不相关的，即其结果输出是二元的，要么相关要么不相关，至于文档在多大程度上和用户查询相关，能否按照相关程度排序输出搜索结果？这些布尔模型都无能为力，所以无法对搜索结果根据相关性进行排序，其搜索结果过于粗糙。同时，对于普通用户来说，要求其以布尔表达式的方式构建查询，无疑要求过高。

## 5.2 向量空间模型 ( Vector Space Model )

向量空间模型历史悠久，最初由信息检索领域奠基人 Salton 教授提出，经过相关学科几十年的探索，目前已经是非常成熟和基础的检索模型。向量空间模型作为一种文档表示和相似性计算的工具，不仅仅在搜索领域，在自然语言处理、文本挖掘等诸多其他领域也是普遍采用的有效工具。

### 5.2.1 文档表示

作为表示文档的工具，向量空间模型把每个文档看做是由  $t$  维特征组成的一个向量，特征的定义可以采取不同方式，可以是单词、词组、N-gram 片段等多种形式，最常用的还是以单词作为特征。其中每个特征会根据一定依据计算其权重，这  $t$  维带有权重的特征共



同构成了一个文档，以此来表示文档的主题内容。

图 5-5 展示了 4 个文档在 3 维向量空间中如何表示，比如对于文档 2 来说，由 3 个带有权重的特征组成{w21,w22,w23}，w21 代表了第 2 个文档的第 1 维特征的权重，其他的权值代表相似含义。在实际应用系统中，维度是非常高的，达到成千上万维很正常，图中为了简化说明，只列出 3 个维度。在搜索这种应用环境下，用户查询可以被看做是一个特殊的文档，将其也转换成  $t$  维的特征向量。至于特征的权值如何计算，本节后面会专门讲述。

向量空间模型文档表示			
	特征1	特征2	特征3
文档1	w11	w12	w13
文档2	w21	w22	w23
文档3	w31	w32	w33
文档4	w41	w42	w43
查询	q1	q2	q3

图 5-5 向量空间模型中的文档表示

图 5-6 展示的是具体的文档表示实例，对于文档 D4 和 D5 及用户查询，通过特征转换，可以将其转换为带有权值的向量表示。

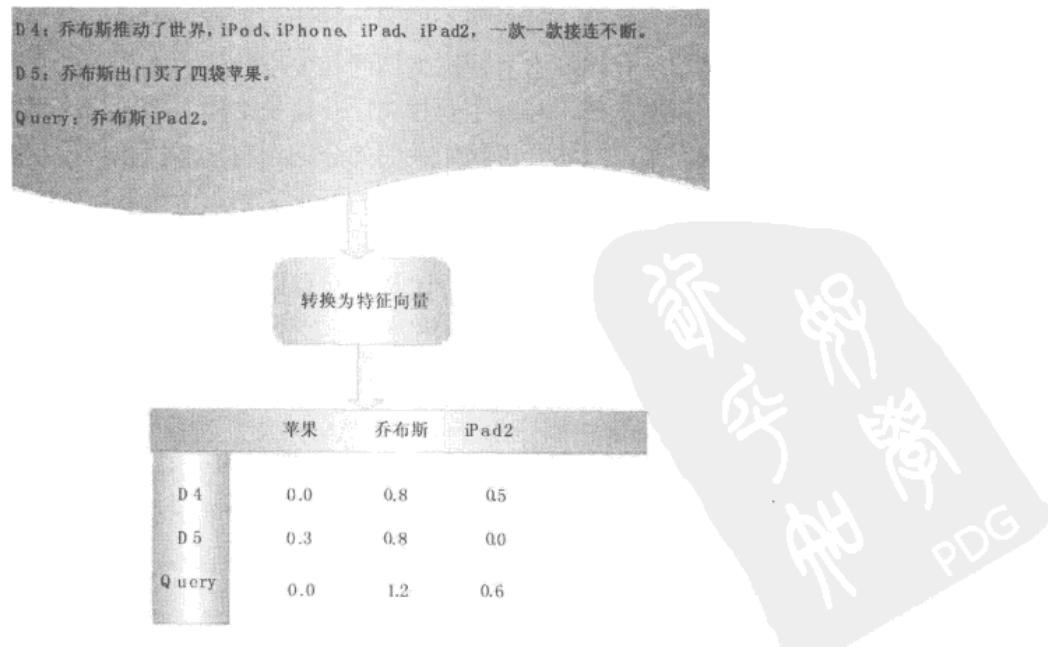


图 5-6 将文档转换为特征向量

### 5.2.2 相似性计算

将文档转换为特征向量后，就可以计算文档之间或者是查询和文档之间的相似性了。对于搜索排序这种任务来说，给定用户输入的查询，理论上应该计算查询和网页内容之间的“相关性”，即文档是否和用户需求相关，之后按照相关程度由高到低排序。向量空间模型将问题做了转换，即以查询和文档之间的内容相似性来作为相关性的替代，按照文档和查询的相似性得分由高到低排序作为搜索结果，但是要注意，两者并非等同的概念。

给定用户查询特征向量和文档特征向量，如何计算其内容相似性？Cosine 相似性是最常用也是非常有效的计算相似性的方式。Cosine 相似性计算定义如下：

$$\text{Cosine}(Q, D_i) = \frac{\sum_{j=1}^t w_{ij} \times q_j}{\sqrt{\sum_{j=1}^t w_{ij}^2 \times \sum_{j=1}^t q_j^2}}$$

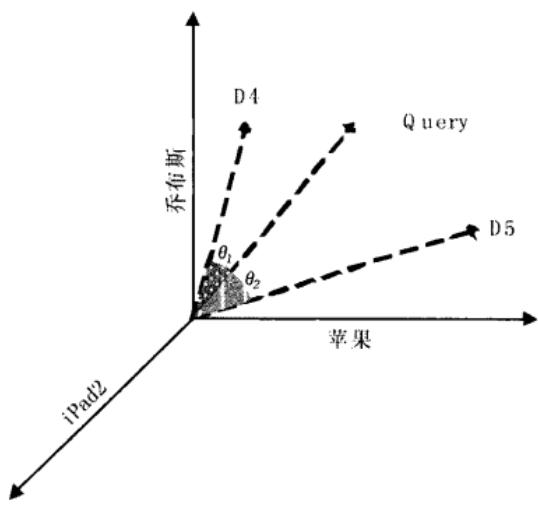
这个公式计算用户查询  $Q$  和  $D_i$  文档的相似性，公式中的分子部分，将文档的每个特征权值和查询的每个特征权值相乘取和，这个过程也叫做求两个向量的点积；公式的分母部分是两个特征向量在欧式空间中长度的乘积，作为对点积计算结果的规范化。之所以要对特征向量的长度做规范化操作，主要是对长文档的一种惩罚机制，否则的话，计算结果往往是长文档得分较高，而这并非因为长文档与查询更相关，而是因为其长度较长，导致特征权值比短文档要大，所以加入规范化操作抑制长文档在排序中的位置。

Cosine 公式中的规范化方法有一个明显的缺陷，即存在对长文档的过分抑制：如果同时有相关的短文档和长文档，它会使短文档的相关度数值大大高于长文档的相关度数值。设想以下情况：一篇短文档，与某主题相关；一篇长文档，除了与这个主题相关之外还讨论了多个其他主题，其中与这个主题相关的部分词频与短文档相仿。在这种情况下，两篇文档中有关这个主题的特征词的权值大致相当，长文档由于包含的索引词很多，文档向量长度比较大，规范化之后有关这个主题的特征词的权重会比短文档小得多。于是，经过相似度计算，短文档得到的相关度数值会比长文档高得多。在 TREC 测试任务中可以很清楚地观察到这种现象：使用 Cosine 规范化方法，对于一个包含 4 个词的查询，返回的第一个结果大多数是 10 个词以下的文档。

为了便于理解 Cosine 相似性，可以将每个文档及查询看做是  $t$  维特征空间中的一个数值点，每个特征形成  $t$  维空间中的一个维度，连接特征空间原点和这个数值点形成一个向量，而 Cosine 相似性就是计算特征空间中两个向量之间的夹角，这个夹角越小，说明两个特征向量内容越相似，夹角越大，说明两个向量内容越不同。考虑一种极端情况：两个完全相同的文档，其在向量空间中的两个向量是重叠的，通过 Cosine 相似性计算得到的相似性结果为 1。



图 5-7 是上述计算的一个形象化示意图，在这个例子里，特征维度为 3，即由“苹果”、“乔布斯”和“iPad2”3 个单词组成特征空间，每个单词代表三维空间中的一个维度。对于文档 D4、D5 和查询“乔布斯 iPad2”，是三维空间中 3 个数值点，和坐标原点相连形成一个向量。查询和文档的相似性就是两个向量之间的夹角大小，从示例中可以看出，查询和文档 D4 之间的夹角要小于与文档 D5 形成的夹角，这意味着查询和文档 D4 更相似些。



D4: 乔布斯推动了世界，iPod、iPhone、iPad、iPad2，一款一款接连不断。

D5: 乔布斯出门买了四袋苹果。

Query: 乔布斯 iPad2。

图 5-7 Cosine 相似性图示

我们以 Cosine 计算公式来计算图 5-6 示例中的文档与查询的相似度，假设文档 D4 的特征向量为 {0.0, 0.8, 0.5}，即对于“乔布斯”的这个特征权值是 0.8，“iPad2”的特征权值是 0.5，由于 D4 中没有出现“苹果”，所以对应的特征权值是 0。相似地，D5 的特征向量为 {0.3, 0.8, 0.0}，用户查询“乔布斯 iPad2”的特征向量为 {0.0, 1.2, 0.6}。有了特征向量，可以参照 Cosine 公式，计算出查询和文档 D4、D5 的相似性如下：

$$\text{Cosine}(\text{Query}, \text{D4}) = \frac{0.8 \times 1.2 + 0.5 \times 0.6}{\sqrt{(0.8^2 + 0.5^2) \times (1.2^2 + 0.6^2)}} = 0.79$$

$$\text{Cosine}(\text{Query}, \text{D5}) = \frac{0.8 \times 0.8}{\sqrt{(0.3^2 + 0.8^2) \times (1.2^2 + 0.6^2)}} = 0.25$$

从上述计算结果可以看出，文档 D4 与查询的 Cosine 相似性要大于文档 D5 与查询的

相似性，在搜索结果排序中，会将 D4 排在 D5 前面。

### 5.2.3 特征权重计算

文档和查询转换为特征向量时，每个特征都会赋予一定的权值，本节叙述如何对特征计算相应的权值。在向量空间模型里，特征权值的计算框架一般被称做 Tf\*IDF 框架，虽然具体计算方式可以有多种，但是大都遵循这一框架，而这一计算框架考虑的主要计算因子有两个：词频 Tf 和逆文档频率 IDF。

#### 词频因子（Tf）

Tf 计算因子代表了词频，即一个单词在文档中出现的次数，一般来说，在某个文档中反复出现的单词，往往能够表征文档的主题信息，即 Tf 值越大，越能代表文档所反映的内容，那么应该给予这个单词更大的权值。这是为何引入词频作为计算权值的重要因子的原因。

具体计算词频因子的时候，基于不同的出发点，可以采纳不同的计算公式。最直接的方式就是直接利用词频数，比如文档中某个单词出现过 5 次，就将这个单词的 Tf 值计为 5。

一种词频因子的变体计算公式是：

$$W_{Tf} = 1 + \log(Tf)$$

即将词频数值 Tf 取 Log 值来作为词频权值，比如单词在文档中出现过 4 次，则其词频因子权值是 3，公式中的数字 1 是为了平滑计算用的，因为如果 Tf 值为 1 的情况下，取 Log 后值为 0，即本来出现了一次的单词，按照这种方法计算会认为这个单词从来没有在文档中出现过，为了避免这种情形，采用加 1 的方式来进行平滑。之所以要对词频取 Log，是基于如下考虑：即使一个单词出现了 10 次，也不应该在计算特征权值时，比出现 1 次的情况权值大 10 倍，所以加入 Log 机制抑制这种过大的差异。

另外一种单词词频因子的变体计算公式是：

$$W_{Tf} = a + (1 - a) \times \frac{Tf}{\text{Max}(Tf)}$$

这种方法被称为增强型规范化 Tf，其中的  $a$  是调节因子，过去经验取值 0.5，新的研究表明取值为 0.4 效果更好。公式中的 Tf 代表这个单词的实际词频数目，而 Max(Tf) 代表了文档中所有单词中出现次数最多的那个单词对应的词频数目。之所以要如此操作，主要出于对长文档的一种抑制，因为如果文档较长，与短文档相比，则长文档中所有单词的 Tf



值会普遍比短文档的值高，但是这并不意味着长文档与查询更相关。用单词实际词频除以文档中最高词频，等于将绝对的数值进行了规范化转换，公式的含义就转换为：同一个文档内单词之间的相对重要性。即使一个文档很长，单词词频普遍很高，但是除以文档最高词频，那么通过这种计算方式得出的数值比短文档来说并不一定就大。这样就剔除了文档长度因素的影响，长文档和短文档的词频因子就成为可比的了。

### 逆文档频率因子 (IDF)

词频因子是与文档密切相关的，说一个单词的 Tf 值，指的是这个单词在某个文档中的出现次数，同一个单词在不同文档中 Tf 值很可能不一样。而逆文档频率因子 IDF 则与此不同，它代表的是文档集合范围的一种全局因子。给定一个文档集合，那么每个单词的 IDF 值就唯一确定，跟具体的文档无关。所以 IDF 考虑的不是文档本身的特征，而是特征单词之间的相对重要性。

所谓逆文档频率因子 IDF，其计算公式如下：

$$\text{IDF}_k = \log \frac{N}{n_k}$$

其中的  $N$  代表文档集合中总共有多少个文档，而  $n_k$  代表特征单词  $k$  在其中多少个文档中出现过，即文档频率。由公式可知，文档频率  $n_k$  越高，则其 IDF 值越小，即越多的文档包含某个单词，那么其 IDF 权值越小。IDF 反映了一个特征词在整个文档集合中的分布情况，特征词出现在其中的文档数目越多，IDF 值越低，这个词区分不同文档的能力越差。在极端情况下，考虑一个在文档集合中所有文档中都出现的特征词 “term”，即  $n_k=N$ ，这说明无论搜索任何主题，“term”这个词都会出现在所有相关和不相关的文档中，因此“term”对任何主题都没有区分相关文档和不相关文档的能力，这时“term”的 IDF 值为 0。例如，在一个有关计算机领域的文档集合中，特征词“电脑”几乎肯定会在所有文档中，这时用它进行搜索没有任何效果。但如果另一个文档集合中包括计算机领域相关的文档和很多有关经济学的文档，那么在这个集合中使用“电脑”搜索计算机相关的文档效果会比较好。也就是说，“电脑”这个特征词在第 1 个文档集合中区分不同文档的能力很差，在第 2 个文档集合中区分能力很强。而 IDF 就是衡量不同单词对文档的区分能力的，其值越高，则其区分不同文档差异的能力越强，反之则区分能力越弱。整体而言，IDF 的计算公式是基于经验和直觉的，有研究者进一步分析认为：IDF 代表了单词带有的信息量的多少，其值越高，说明其信息含量越多，就越有价值。

### Tf\*IDF 框架

Tf\*IDF 框架就是结合了上述的词频因子和逆文档频率因子的计算框架，一般是将两者

相乘作为特征权值，特征权值越大，则越可能是好的指示词，即：

$$\text{Weight}_{\text{word}} = \text{Tf} \times \text{IDF}$$

从上述公式可以看出，对于某个文档 D 来说：

如果 D 中某个单词的词频很高，而且这个单词在文档集合的其他文档中很少出现，那么这个单词的权值会很高。

如果 D 中某个单词的词频很高，但是这个单词在文档集合的其他文档中也经常出现，或者单词词频不高，但是在文档集合的其他文档中很少出现，那么这个单词的权值一般。

如果 D 中某个单词词频很低，同时这个单词在文档集合的其他文档中经常出现，那么这个单词权值很低。

经过几十年的不断探索，向量空间模型已经相当成熟，并被各种领域广泛采用。从数学模型的角度看，向量空间模型简单直观，用查询和文档之间的相似性来模拟搜索中的相关性。与布尔模型相比，也能对文档与查询的相关性进行打分排序。但是总体而言，向量空间模型是个经验型的模型，是靠直觉和经验不断摸索完善的，缺乏一个明确的理论来指导其改进方向。

## 5.3 概率检索模型

概率检索模型是目前效果最好的模型之一，在 TREC 等各种检索系统评测会议已经证明了这一点，而且 okapi BM25 这一经典概率模型计算公式已经在商业搜索引擎的网页排序中广泛使用。

### 5.3.1 概率排序原理

概率检索模型是从概率排序原理推导出来的，所以理解这一原理对于理解概率检索模型非常重要。概率排序原理的基本思想是：给定一个用户查询，如果搜索系统能够在搜索结果排序时按照文档和用户需求的相关性由高到低排序，那么这个搜索系统的准确性是最优的。而在文档集合的基础上尽可能准确地对这种相关性进行估计则是其核心。

从概率排序原理的表述来看，这是一种直接对用户需求相关性进行建模的方法，这点与向量空间模型不同，向量空间模型是以查询和文档的内容相似性来作为相关性的代替品。

概率排序原理只是一种指导思想，并未阐述如何才能做到这种理想的情形，在这个框架下，怎样对文档与用户需求的相关性建立模型呢？用户发出一个查询请求，如果我们把



文档集合划分为两个类别：相关文档子集和不相关文档子集，于是就可以将这种相关性衡量转换为一个分类问题。

图 5-8 示意了概率模型作为一个分类问题，对于某个文档 D 来说，如果其属于相关文档子集的概率大于属于不相关文档子集的概率，我们就可以认为这个文档与用户查询是相关的。图中的  $P(R|D)$  代表给定一个文档 D 对应的相关性概率，而  $P(NR|D)$  则代表该文档的不相关概率。即如果  $P(R|D) > P(NR|D)$ ，我们可以认为文档与用户查询是相关的。

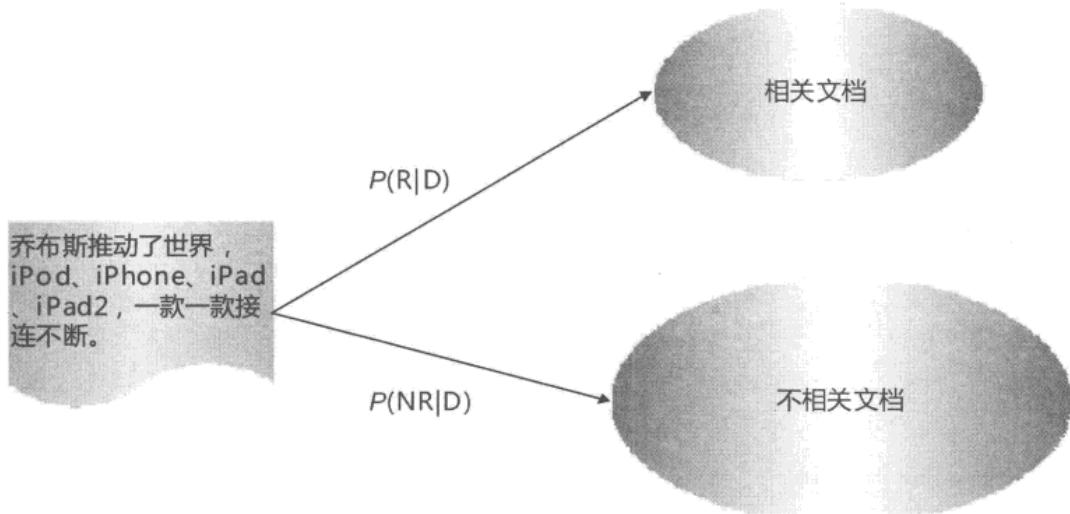


图 5-8 概率模型是一个分类问题

所以现在问题的关键是如何估算  $P(R|D)$  和  $P(NR|D)$  的数值。为了简化问题，首先我们根据贝叶斯规则对两个概率值进行改写，即：

$$P(R|D) = \frac{P(D|R)P(R)}{P(D)}$$

$$P(NR|D) = \frac{P(D|NR)P(NR)}{P(D)}$$

相关性计算的目的是要判断是否  $P(R|D) > P(NR|D)$ ，将改写公式代入成为如下形式：

$$P(R|D) > P(NR|D) \Leftrightarrow \frac{P(D|R)P(R)}{P(D)} > \frac{P(D|NR)P(NR)}{P(D)}$$

因为是同一篇文档，所以右端公式的分母  $P(D)$  是相同的，在做相关性判断的时候可以消掉，并对因子移项，转换成如下形式：

$$\frac{P(D|R)P(R)}{P(D)} > \frac{P(D|NR)P(NR)}{P(D)} \Leftrightarrow \frac{P(D|R)}{P(D|NR)} > \frac{P(NR)}{P(R)}$$

尽管概率模型将相关性判断转换为一个二值分类问题，但搜索应用并不需要进行真正的分类，而是将搜索结果按照相关性得分由高到低排序，所以对于搜索系统来说只需要将文档按照  $\frac{P(D|R)}{P(D|NR)}$  大小降序排列即可，于是问题进一步转换成如何估算因子  $P(D|R)$  和  $P(D|NR)$ ，而二元独立模型提供了计算这些因子的框架。

### 5.3.2 二元独立模型 (Binary Independent Model)

为了能够使得估算上述两个计算因子可行，二元独立模型（BIM 模型）做出了两个假设：

#### 假设 1：二元假设

所谓二元假设，类似于布尔模型中的文档表示方法，一篇文档在由特征进行表示的时候，以特征“出现”和“不出现”两种情况来表示，不考虑词频等其他因素。假设特征集合包含 5 个单词，某个文档 D 根据二元假设，表示为 {1,0,1,0,1}，其含义是这个文档出现了第 1 个、第 3 个和第 5 个单词，但是不包含第 2 个和第 4 个单词。做出二元假设是将复杂问题简单化的一种措施。

#### 假设 2：词汇独立性假设

所谓词汇独立性假设，是指文档里出现的单词之间没有任何关联，任意一个单词在文档的分布概率不依赖于其他单词是否出现。这个假设很明显和事实不符，比如一篇文档出现了单词“乔布斯”，那么出现“苹果”、“iPad”这些单词的概率显然很大，即单词之间是有依赖性的。但是为了简化计算，很多方法都会做出词汇独立性假设。有了词汇独立性假设，我们就可以将对一个文档的概率估计转换为对文档包含单词的概率估计，因为词汇之间没有关联，所以可以将文档概率转换为单词概率的乘积。

在以上两个前提假设下，二元独立模型即可对两个因子  $P(D|R)$  和  $P(D|NR)$  进行估算，在进行形式化描述前，我们举个简单的例子。

上文提到的文档 D 表示为 {1,0,1,0,1}，我们用  $p_i$  来代表第  $i$  个单词在相关文档集合内出现的概率，于是在已知相关文档集合的情况下，观察到文档 D 的概率为：

$$P(D|R) = p_1 \times (1 - p_2) \times p_3 \times (1 - p_4) \times p_5$$



因为在文档中出现了第1个、第3个和第5个单词，所以这些单词在相关文档集合中出现的概率为 $p_i$ ，而第2个和第4个单词没有出现，那么 $1-p_i$ 就代表了单词不在文档出现的概率，根据词汇独立性假设，将对文档D的概率估计转换为每个单词概率的乘积，这样就可以估算因子 $P(D|R)$ 。

对于因子 $P(D|NR)$ ，我们假设用 $s_i$ 代表第*i*个单词在不相关文档集合内出现的概率，于是在已知不相关文档集合的情况下，观察到文档D的概率为：

$$P(D|NR) = s_1 \times (1 - s_2) \times s_3 \times (1 - s_4) \times s_5$$

两个因子已经估算完毕，于是我们可以估算 $\frac{P(D|R)}{P(D|NR)}$ ，即：

$$\frac{P(D|R)}{P(D|NR)} = \frac{p_1 \times (1 - p_2) \times p_3 \times (1 - p_4) \times p_5}{s_1 \times (1 - s_2) \times s_3 \times (1 - s_4) \times s_5}$$

如果可以从文档集合估计 $p_i$ 和 $s_i$ ，那么我们就可以对文档的相关性进行直接计算。这是一个具体的实例，下面我们用形式化方式表示如何计算 $\frac{P(D|R)}{P(D|NR)}$ ：

$$\frac{P(D|R)}{P(D|NR)} = \prod_{i:d_i=1} \frac{p_i}{s_i} \times \prod_{i:d_i=0} \frac{1 - p_i}{1 - s_i}$$

这个公式与上面实例所列公式含义相同，只不过将各个计算因子归为两个部分，其中 $\prod_{i:d_i=1}$ 代表了在文档D中出现过的各个单词的概率乘积， $\prod_{i:d_i=0}$ 则代表了没有在文档D中出现的各个特征单词的概率乘积。进一步对这个公式进行一些数学等价变换，可得：

$$\begin{aligned} \frac{P(D|R)}{P(D|NR)} &= \prod_{i:d_i=1} \frac{p_i}{s_i} \times \left( \prod_{i:d_i=1} \frac{1 - s_i}{1 - p_i} \times \prod_{i:d_i=1} \frac{1 - p_i}{1 - s_i} \right) \times \prod_{i:d_i=0} \frac{1 - p_i}{1 - s_i} \\ &= \left( \prod_{i:d_i=1} \frac{p_i}{s_i} \times \prod_{i:d_i=1} \frac{1 - s_i}{1 - p_i} \right) \times \left( \prod_{i:d_i=1} \frac{1 - p_i}{1 - s_i} \times \prod_{i:d_i=0} \frac{1 - p_i}{1 - s_i} \right) \\ &= \prod_{i:d_i=1} \frac{p_i}{s_i} \frac{(1 - s_i)}{(1 - p_i)} \times \prod_i \frac{1 - p_i}{1 - s_i} \end{aligned}$$

即将计算公式分解为两个组成部分，第1个组成部分 $\prod_{i:d_i=1}$ 代表在文档中出现过的单词所计算得到的单词概率乘积，第2个部分 $\prod_i$ 代表对所有特征词计算所得到的单词概率乘积。因为 $p_i$ 和 $s_i$ 是从相关文档和不相关文档集合中统计出来的全局概率，所以与具体文档无关，这说明对于所有文档来说第2个部分得分都一样，所以在排序中不起作用，于是可

以将这个部分消掉，得到最终的相关性估算公式：

$$\frac{P(D|R)}{P(D|NR)} = \prod_{i:d_i=1} \frac{p_i(1-s_i)}{s_i(1-p_i)}$$

出于计算方便，可以对上述公式取  $\log$  值，得到如下相关性估值公式：

$$\sum_{i:d_i=1} \log \frac{p_i(1-s_i)}{s_i(1-p_i)}$$

到目前为止，我们已经获得了计算相关性的具体方法，剩下的问题就是如何计算单词概率  $p_i$  和  $s_i$ 。给定用户查询，如果我们可以确定哪些文档构成了相关文档集合，哪些文档构成了不相关文档集合，可以利用下表所列数据来估算单词概率：

	相关文档	不相关文档	文档数量
$d_i = 1$	$r_i$	$n_i - r_i$	$n_i$
$d_i = 0$	$R - r_i$	$(N-R) - (n_i - r_i)$	$N - n_i$
文档数量	$R$	$N - R$	$N$

表中第 3 行的  $N$  为文档集合总共包含的文档个数， $R$  为相关文档的个数，于是  $N-R$  就是不相关文档集合的大小。对于某个单词  $d_i$  来说，假设包含这个单词的文档数量共有  $n_i$  个，而其中相关文档有  $r_i$  个，那么不相关文档中包含这个单词的文档数量则为  $n_i - r_i$ 。再考虑表中第 2 列，因为相关文档个数是  $R$ ，而其中出现过单词  $d_i$  的有  $r_i$  个，那么相关文档中没有出现过这个单词的文档个数为  $R - r_i$  个，同理，不相关文档中没有出现过这个单词的文档个数为  $(N-R) - (n_i - r_i)$  个。从表中可以看出，如果我们假设已经知道  $N$ 、 $R$ 、 $n_i$ 、 $r_i$  的话，其他参数是可以靠这 4 个数值推导出来的。

根据表格数据，即可估算  $s_i$  和  $p_i$ 。因为  $p_i$  代表第  $i$  个单词在相关文档集合内出现的概率，在 BIM 模型的二元假设下，可以用包含这个单词的相关文档个数  $r_i$  除以相关文档总数  $R$  来估算，即  $p_i = r_i/R$ 。而  $s_i$  代表第  $i$  个单词在不相关文档集合内出现的概率，所以可以用包含这个单词的不相关文档个数  $n_i - r_i$  除以不相关文档总数  $(N-R)$  来估算，即  $s_i = n_i - r_i / (N-R)$ 。把这两个估值公式代入相关性估值公式即可得出如何计算相关性，但是这里有个问题，相关性估值公式采取了  $\log$  形式，如果  $r_i=0$ ，那么会出现  $\log(0)$  的情形，为了避免这种情况，我们对  $p_i$  和  $s_i$  的估值公式进行平滑，分子部分加上 0.5，分母部分加上 1.0，即：

$$p_i = (r_i + 0.5)/(R + 1.0)$$



$$s_i = (n_i - r_i + 0.5)/(N - R + 1.0)$$

将这两个估值因子代入相关性估算公式，得到如下公式：

$$\sum_{i:q_i=d_i=1} \log \frac{(r_i + 0.5)/(R - r_i + 0.5)}{(n_i - r_i + 0.5)/((N - R) - (n_i - r_i) + 0.5)}$$

其代表的含义是：对于同时出现在用户查询 Q 和文档 D 中的单词，累加每个单词的估值，其和就是文档 D 和查询的相关性度量。这即是二元独立模型推导出的用户查询和文档之间的相关性计算方法，如果我们事先不知道哪些是相关文档，哪些是不相关文档，可以给公式的估算因子赋予固定值，此种情况下该公式等价于在向量空间模型中提到的 IDF 因子。各种实验表明，根据二元独立模型计算相关性实际效果并不好，但是这个模型却是非常成功的概率模型方法 BM25 的基础，这是为何本节花较大篇幅介绍 BIM 模型的原因。

### 5.3.3 BM25 模型

BIM 模型基于二元假设推导而出，即对于单词特征，只考虑是否在文档中出现过，而没有考虑单词的权值。BM25 模型在 BIM 模型基础上，考虑了单词在查询中的权值及单词在文档中的权值，拟合出综合上述考虑因素的公式，并通过实验引入一些经验参数。BM25 模型是目前最成功的内容排序模型。

图 5-9 展示了 BM25 模型的具体计算公式，对于查询 Q 中出现的每个查询词，依次计算单词在文档 D 中的分值，累加后就是文档 D 与查询 Q 的相关性得分。从图中可以看出，计算第  $i$  个查询词的权值时，计算公式可以拆解为 3 个组成部分，第 1 个组成部分就是上节所述的 BIM 模型计算得分，上节也提到，如果赋予一些默认值的话，这个部分的公式等价于 IDF 因子的作用；第 2 个组成部分是查询词在文档 D 中的权值，其中  $f_i$  代表了单词在文档 D 中的词频， $k_1$  和  $K$  是经验参数；第 3 个组成部分是查询词自身的权值，其中  $qf_i$  代表查询词在用户查询中的词频，如果查询较短小的话，这个值往往是 1， $k_2$  是经验参数。BM25 模型就是融合了这 3 个计算因子的相关性计算公式。

在第 2 个计算因子中， $K$  因子代表了对文档长度的考虑，图中所示  $K$  的计算公式中， $dl$  代表文档 D 的长度，而  $avdl$  则是文档集合中所有文档的平均长度， $k_1$  和  $b$  是经验参数。其中参数  $b$  是调节因子，极端情况下，将  $b$  设定为 0，则文档长度因素将不起作用，经验表明一般将  $b$  设定为 0.75 会获得较好的搜索效果。

BM25 公式中包含 3 个自由调节参数，除了调节因子  $b$  外，还有针对词频的调节因子  $k_1$  和  $k_2$ 。 $k_1$  的作用是对查询词在文档中的词频进行调节，如果将  $k_1$  设定为 0，则第 2 部分计算

因子成了整数 1，即不考虑词频的因素，退化成了 BM 模型。如果将  $k_1$  设定较大值，则第 2 部分计算因子基本和词频  $f_i$  保持线性增长，即放大了词频的权值，根据经验，一般将  $k_1$  设定为 1.2。调节因子  $k_2$  和  $k_1$  的作用类似，不同点在于其是针对查询中的词频进行调节，一般将这个值设定在 0 到 1000 较大的范围内，之所以如此，是因为查询往往很短，所以不同查询词的词频都很小，词频之间差异不大，较大的调节参数数值设定范围允许对这种差异进行放大。

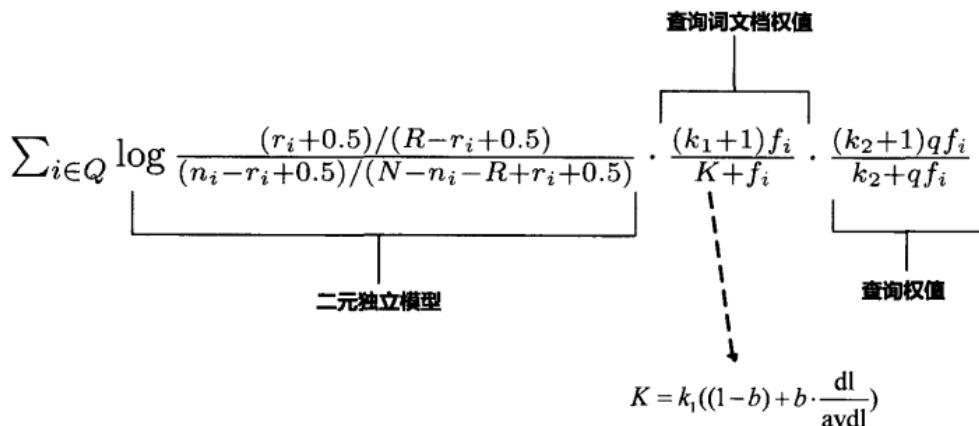


图 5-9 BM25 计算公式

综合来看，BM25 模型计算公式其实融合了 4 个考虑因素：IDF 因子、文档长度因子、文档词频和查询词频，并利用 3 个自由调节因子（ $k_1$ 、 $k_2$  和  $b$ ）对各种因子的权值进行调整组合。

下面我们以用户查询“乔布斯 iPad2”为例，来看看如何实际利用 BM25 公式计算某个文档 D 的相关性。首先假定 BM25 的第 1 个计算因子中，我们不知道哪些是相关文档，所以将相关文档个数  $R$  和包含查询词的相关文档个数  $r$  设定为 0，此时第 1 个计算因子退化成类似于 IDF 的形式：

$$\log \frac{(0+0.5)/(0-0+0.5)}{n_i-0+0.5/((N-0)-(n_i-0)+0.5)} \Leftrightarrow \log \frac{(N-n_i+0.5)}{(n_i+0.5)}$$

因为查询中两个查询词都只出现了一次，所以其对应的都为 1。其他数值假定如下：

文档集合总数  $N=100000$

文档集合中包含单词“乔布斯”的文档个数  $n_{\text{乔布斯}}=1000$

文档集合中包含单词“iPad2”的文档个数  $n_{\text{iPad2}}=100$

文档 D 中出现单词“乔布斯”的词频  $f_{\text{乔布斯}}=8$

文档 D 中出现单词 “iPad2” 的词频  $f_{iPad2}=5$

调节因子  $k_1=1.2$

调节因子  $k_2=200$

调节因子  $b=0.75$

假定文档长度是平均文档长度的 1.5 倍，即  $K=1.2 \times (0.25+0.75 \times 1.5)=1.65$ ，将这些数值代入 BM25 计算公式，可以得出文档 D 和查询的如下相关性得分：

$$\begin{aligned} \text{BM25}(“乔布斯 iPad2”, D) &= \log \frac{100000-1000+0.5}{1000+0.5} \times \frac{(1.2+1) \times 8}{1.65+8} \times \frac{(200+1) \times 1}{200+1} \\ &+ \log \frac{100000-100+0.5}{100+0.5} \times \frac{(1.2+1) \times 5}{1.65+5} \times \frac{(200+1) \times 1}{200+1} \\ &= 8.59 \end{aligned}$$

对于文档集合里的所有文档，都按照上述方法计算，将计算结果按照大小排序，就根据 BM25 公式得出了内容相关性排序。

### 5.3.4 BM25F 模型

Okapi BM25 模型于 1994 年提出，之后被广泛研究并应用在不同系统里，也陆续有在此基础上的改进算法出现。2004 年推出的 BM25F 模型就是一个很典型的 BM25 改进算法。

BM25 模型在计算一个文档和查询的相关性的时候，是把文档当做一个整体来看待的，但是在有些应用场景下，需要将文档内容切割成不同的组成部分，然后对不同成分分别对待。最典型的应用场景就是网页搜索，一个网页由页面标题、Meta 描述信息、页面内容等不同的“域”（Field）构成，在计算内容相关性得分的时候，往往希望标题的权重大些，因为标题是一个网页的中心思想简述。而 BM25 由于没有考虑这点，所以不容易做到不同“域”的内容给予不同权重，BM25F 就非常适合这种将文档划分为若干“域”，不同“域”需要不同权值的应用场景。

BM25F 思路如上，但是不同系统有各异的具体计算方法，本节叙述其中的一种，为了让公式看起来简洁一些，我们用  $W_i^{\text{RSI}}$  代替 BIM 推导出的部分，也即 BM25 公式的第 1 个组成部分：

$$W_i^{\text{RSJ}} = \log \frac{(r_i + 0.5)/(R - r_i + 0.5)}{(n_i - r_i + 0.5)/((N - R) - (n_i - r_i) + 0.5)}$$

BM25F 的计算公式如下：

$$\sum_{i:q_i=d_i=1} W_i^{\text{RSJ}} \times \frac{f'_i}{k_1 + f'_i}$$

BM25F 公式与 BM25 的区别体现在第 2 个计算因子中的  $f'_i$  上，这个因子综合了第  $i$  个单词出现在不同“域”的得分，其定义如下：

$$f'_i = \sum_{k=1}^u w_k \times \frac{f_{ui}}{B_u}$$

$$B_u = ((1 - b_u) + b_u \times \frac{\text{ul}_u}{\text{avul}_u})$$

假设一个文档 D 包含了  $u$  个不同的“域”，而每个“域”的权值设定为  $w_k$ ， $f'_i$  将第  $i$  个单词在各个不同“域”的分值  $f_{ui}/B_u$  加权求和，其中  $f_{ui}$  代表单词在第  $u$  个“域”出现的词频数， $B_u$  则是“域”长度因素。对于  $B_u$  来说， $\text{ul}_u$  是这个“域”的实际长度， $\text{avul}_u$  是文档集合中这个“域”的平均长度，而  $b_u$  则是调节因子，这里要注意的一点是：不同的“域”要设定不同的调节因子，这样搜索效果才好。所以对于 BM25F 来说，除了调节因子  $k_1$  外，如果文档包含  $u$  个不同的“域”，则还需要凭经验设定  $u$  个“域”权值和  $u$  个长度调节因子。

## 5.4 语言模型方法

基于统计语言模型的检索模型于 1998 年首次提出，是借鉴了语音识别领域采用的语言模型技术，将语言模型和信息检索相互融合的结果。

从基本思路上来说，其他的大多数检索模型的思考路径是从查询到文档，即给定用户查询，如何找出相关的文档。语言模型方法的思路正好相反，是由文档到查询这个方向，即为每个文档建立不同的语言模型，判断由文档生成用户查询的可能性有多大，然后按照这种生成概率由高到低排序，作为搜索结果。

给定了一篇文档和对应的用户查询，如何计算文档生成查询的概率呢？图 5-10 展示了利用语言模型来生成查询的过程。首先，可以为每个文档建立一个语言模型，语言模型代表了单词或者单词序列在文档中的分布情况。我们可以想象从文档的语言模型生成查询类似于从一个装满“单词”的壶中随机抽取，一次抽取一个单词，这样，每个单词都有一个



被抽取到的概率。对于查询中的单词来说，每个单词都对应一个抽取概率，将这些单词的抽取概率相乘就是文档生成查询的总体概率。

以图 5-10 中的文档为例，可以看出，文档 D 包含 5 个单词{乔布斯，出门，买了，4 袋，苹果}，每个单词出现次数都是 1 次，所以抽取到这 5 个单词的概率大小相同，都为 0.2，这个单词抽取概率分布就是文档 D 对应的一元语言模型。为文档建立一元语言模型后，即可计算文档 D 生成用户查询{苹果，乔布斯，iPad2}的概率：

$$P(Q|D) = P(\text{苹果}|D) \times P(\text{乔布斯}|D) \times P(\text{iPad2}|D) = 0.2 \times 0.2 \times 0 = 0$$

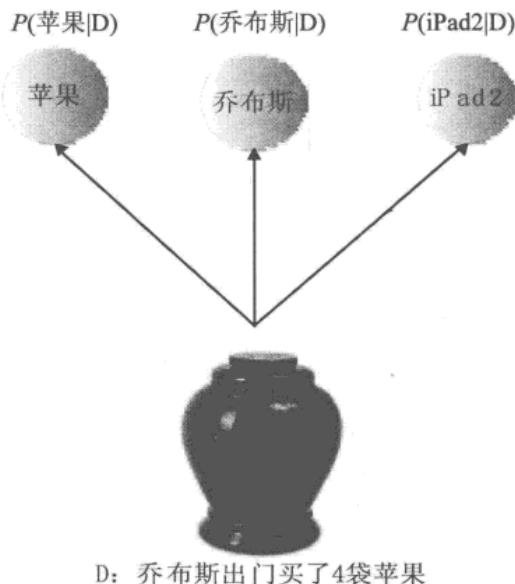


图 5-10 文档生成查询

因为文档中出现过“苹果”和“乔布斯”两个单词，其对应的概率都为 0.2，但是由于文档 D 没有出现过单词“iPad2”，所以这个单词的生成概率为 0，导致查询整体的生成概率也为 0。从这个例子可以看出，由于一个文档文字内容有限，所以很多查询词都没有在文中出现过，这会导致基于语言模型的检索方法失效，这个问题被称做语言模型的数据稀疏问题，是语言模型方法重点需要解决的问题。

一般采用数据平滑的方式解决数据稀疏问题，所谓数据平滑，可以理解为单词分布概率领域的“劫富济贫”，即从文档中出现过的单词的分布概率值中取出一部分，将这些值分配给文档中没有出现过的单词，这样所有单词都有一个非零的概率值，避免计算中出现问题。

语言模型检索方法则是为所有单词引入一个背景概率来做数据平滑。所谓背景概率，

就是给文档集合建立一个整体的语言模型，因为其规模比较大，所以绝大多数查询词都有一个概率值。对于文档集合的一元语言模型，某个单词的背景概率就是这个单词出现的次数除以文档集合的单词总数，这与一个文档计算语言模型是一致的，可以将背景概率的计算理解为：将文档集合内所有文档拼接成一个虚拟的大文档，然后给这个虚拟文档建立语言模型。所以，对于语言模型方法来说，如果文档集合包含  $N$  个文档，则需要建立  $N+1$  个不同的语言模型，其中每个文档建立自己的语言模型，而文档集合语言模型则作为数据平滑的用处。

图 5-11 是加入数据平滑后的文档生成查询概率计算公式，可以看出，每个查询词的生成概率由两部分构成，一部分是之前提到的文档语言模型，另一部分是用做平滑的文档集合语言模型。两者之间可以通过参数调节其权重。

$$P(Q|D)(Q|D) = \prod_{i=1}^n ((1 - \lambda) \frac{f_{q_i, D}}{|D|} + \lambda \frac{c_{q_i}}{|C|})$$

文档语言模型      文档集合语言模型

图 5-11 文档生成查询概率计算

了解了如何计算某个文档生成查询的概率，那么如何对文档按照相关性排序也就很清楚了。图 5-12 是基于语言模型方法的检索系统对搜索结果排序的示意图，用户提交查询  $Q$ ，文档集合内所有文档都计算生成  $Q$  的概率，然后按照生成概率值由大到小排序，这就是搜索结果。

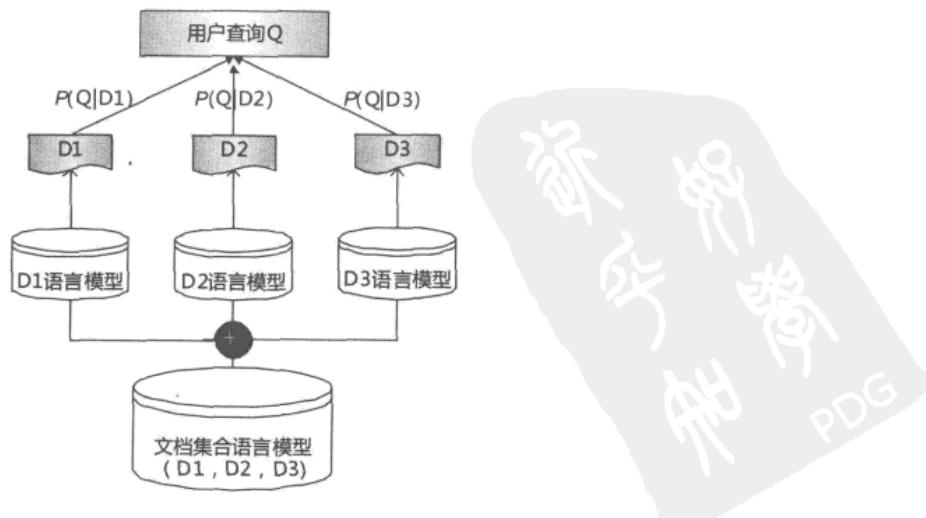


图 5-12 语言模型排序



上述是最基本的语言模型检索方法思路，很多研究在此基础上做出了改进，比如 HMM 隐马尔科夫语言模型、相关模型、翻译模型等诸多改进模型对基础模型做出了改进。目前各种检索评测证明，语言模型检索方法效果略优于精调参数的向量空间模型，与 BM25 等概率模型效果相当。

语言模型检索方法从形式上看与向量空间模型和概率模型有很大差异，其实它们之间有密切联系。通过理论推导，可以得出：语言模型检索方法的排序公式符合概率模型的概率排序原理，所以可以归为概率模型的一种。另外，通过一些公式变形推导，可以将其公式转化为类似于向量空间模型的 Tf\*IDF 的形式，这说明了几种模型在利用计算因子方面的一致性。

## 5.5 机器学习排序 ( Learning to Rank )

利用机器学习技术来对搜索结果进行排序，这是最近几年非常热门的研究领域。信息检索领域已经发展了几十年，为何将机器学习技术和信息检索技术相互结合出现较晚？主要有两方面的原因。

一方面是因为：在前面几节所述的基本检索模型可以看出，用来对查询和文档的相关性进行排序，所考虑的因素并不多，主要是利用词频、逆文档频率和文档长度这几个因子来人工拟合排序公式。因为考虑因素不多，由人工进行公式拟合是完全可行的，此时机器学习并不能派上很大用场，因为机器学习更适合采用很多特征来进行公式拟合，此时若指望人工将几十种考虑因素拟合出排序公式是不太现实的，而机器学习做这种类型的工作则非常合适。随着搜索引擎的发展，对于某个网页进行排序需要考虑的因素越来越多，比如网页的 PageRank 值、查询和文档匹配的单词个数、网页 URL 链接地址长度等都对网页排名产生影响，Google 目前的网页排序公式考虑了 200 多种因子，此时机器学习的作用即可发挥出来，这是原因之一。

另外一个原因是：对于有监督机器学习来说，首先需要大量的训练数据，在此基础上才可能自动学习排序模型，单靠人工标注大量的训练数据不太现实。对于搜索引擎来说，尽管无法靠人工来标注大量训练数据，但是用户点击记录是可以当做机器学习方法训练数据的一个替代品，比如用户发出一个查询，搜索引擎返回搜索结果，用户会点击其中某些网页，可以假设用户点击的网页是和用户查询更加相关的页面。尽管这种假设很多时候并不成立，但是实际经验表明使用这种点击数据来训练机器学习系统确实是可行的。

### 5.5.1 机器学习排序的基本思路

传统的检索模型靠人工拟合排序公式，并通过不断的实验确定最佳的参数组合，以此来形成相关性打分函数。机器学习排序与此思路不同，最合理的排序公式由机器自动学习获得，而人则需要给机器学习提供训练数据。图 5-13 是利用机器学习进行排序的基本原理图。

机器学习排序系统由 4 个步骤组成：人工标注训练数据、文档特征抽取、学习分类函数、在实际搜索系统中采用机器学习模型。

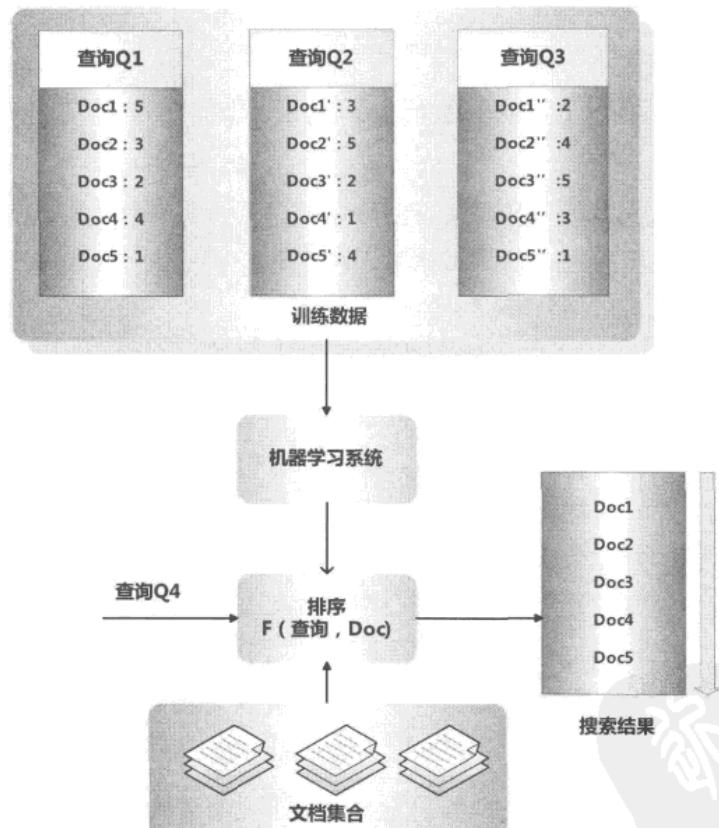


图 5-13 机器学习排序原理

首先，由人工标注训练数据。也就是说，对于某个查询  $Q$ ，人工标出哪些文档是和这个查询相关的，同时标出相关程度，相关程度有时候可以用数值序列来表示，比如从 1 分到 5 分为 5 个档次，1 代表微弱相关，5 代表最相关，其他数值代表相关性在两者之间。对于某个查询，可能相关文档众多，同时用户查询也五花八门，所以全部靠人工标注有时候不太可能。此时，可以利用用户点击记录来模拟这种人工打分机制。



对于机器学习系统来说，输入是用户查询和一系列标注好的文档，机器学习系统需要学习打分函数，然后按照打分函数输出搜索结果。但是在其内部，每个文档是由若干特征构成的，即每个文档进入机器学习系统之前，首先需要将其转换为特征向量。比较常用的特征包括：

- 查询词在文档中的词频信息；
- 查询词的 IDF 信息；
- 文档长度；
- 网页的入链数量；
- 网页的出链数量；
- 网页的 PageRank 值；
- 网页的 URL 长度；
- 查询词的 Proximity 值：即在文档中多大的窗口内可以出现所有查询词。

以上所列只是影响排序的一部分特征，实际上还有很多类似的特征可以作为特征向量中的一维加入。在确定了特征数量后，即可将文档转换为特征向量  $X$ ，前面说过每个文档会人工标出其相关性得分  $Y$ ，这样每个文档会转换为  $\langle X, Y \rangle$  的形式，即特征向量及其对应的相关性得分，这样就形成了一个具体的训练实例。

通过多个训练实例，就可以采用机器学习技术来对系统进行训练，训练的结果往往是一个分类函数或者回归函数，在之后的用户搜索中，就可以用这个分类函数对文档进行打分，形成搜索结果。

从目前的研究方法来说，可以将机器学习排序方法分为以下 3 种：单文档方法、文档对方法和文档列表方法。

### 5.5.2 单文档方法 (PointWise Approach)

单文档方法的处理对象是单独的一篇文档，将文档转换为特征向量后，机器学习系统根据从训练数据中学习到的分类或者回归函数对文档打分，打分结果即是搜索结果。下面我们用一个简单的例子说明这种方法。

图 5-14 是人工标注的训练集合，在这个例子中，我们对于每个文档采用了 3 个特征：查询与文档的 Cosine 相似性分值、查询词的 Proximity 值及页面的 PageRank 数值，而相关性判断是二元的，即要么相关要么不相关，当然，这里的相关性判断完全可以按照相关程度扩展为多元的，本例为了方便说明做了简化。

文档ID	用户查询	Cosine 相似度	Proximity 值	PageRank 值	相关性
2	“微软 产品”	0.24	7	7	相关
3	“苹果 电脑”	0.19	8	2	不相关
4	“苹果 电脑”	0.43	3	3	相关
5	“上市 公司”	0.28	7	1	不相关

图 5-14 训练数据

例子中提供了 5 个训练实例，每个训练实例分别标出了其对应的查询，3 个特征的得分情况及相关性判断。对于机器学习排序系统来说，根据训练数据，需要学习如下的线性打分函数：

$$\text{Score}(Q, D) = a \times CS + b \times PM + c \times PR + d$$

这个公式中，CS 代表 Cosine 相似度变量，PM 代表 Proximity 值变量，PR 代表 PageRank 值变量，而 a、b、c、d 则是变量对应的参数。如果得分大于一设定阈值，则可以认为是相关的，如果小于设定阈值则可以认为不相关。通过训练实例，可以获得最优的 a、b、c、d 参数组合，当这些参数确定后，机器学习系统就算学习完毕，之后即可利用这个打分函数来进行相关性判断。对于某个新的查询 Q 和文档 D，系统首先获得其文档 D 对应的 3 个特征的特征值，之后利用学习到的参数组合计算两者得分，当得分大于设定的阈值，即可判断文档是相关文档，否则判断为不相关文档。

### 5.5.3 文档对方法 (PairWise Approach)

对于搜索任务来说，系统接收到用户查询后，返回相关文档列表，所以问题的关键是确定文档之间的先后顺序关系。单文档方法完全从单个文档的分类得分角度计算，没有考虑文档之间的顺序关系。文档对方法则将重点转向了对文档顺序关系是否合理进行判断。

之所以被称为文档对方法，是因为这种机器学习方法的训练过程和训练目标，是判断任意两个文档组成的文档对<Doc1,Doc2>是否满足顺序关系，即判断是否 Doc1 应该排在 Doc2 的前面。图 5-15 展示了一个训练实例：查询 Q1 对应的搜索结果列表如何转换为文档对的形式，因为从人工标注的相关性得分可以看出，Doc2 得分最高，Doc3 次之，Doc1 得分最低，于是我们可以按照得分大小顺序关系得到 3 个如图 5-15 所示的文档对，将每个文



档对的文档转换为特征向量后，就形成了一个具体的训练实例。

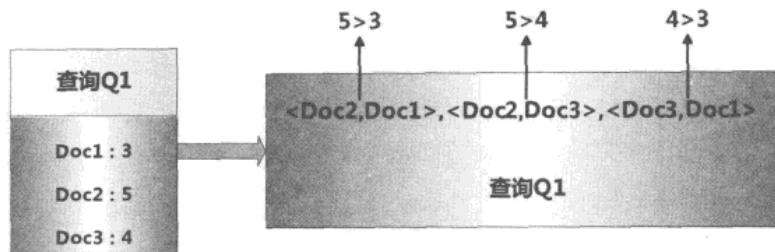


图 5-15 文档对方法的训练实例

根据转换后的训练实例，就可以利用机器学习方法进行分类函数的学习，具体的学习方法有很多，比如 SVM、Boost、神经网络等都可以作为具体的学习方法，但是不论具体方法是什么，其学习目标都是一致的，即输入一个查询和文档对 $\langle Doc1, Doc2 \rangle$ ，机器学习排序能够判断这种顺序关系是否成立，如果成立，那么在搜索结果中 Doc1 应该排在 Doc2 前面，否则 Doc2 应该排在 Doc1 前面。通过这种方式，就完成搜索结果的排序任务。

尽管文档对方法相对单文档方法做出了改进，但是这种方法也存在两个明显的问题。一个问题是：文档对方法只考虑了两个文档对的相对先后顺序，却没有考虑文档出现在搜索列表中的位置。排在搜索结果前列的文档更为重要，如果前列文档出现判断错误，代价明显高于排在后面的文档。针对这个问题的改进思路是引入代价敏感因素，即每个文档对根据其在列表中的顺序具有不同的权重，越是排在前列的权重越大，即在搜索列表前列如果排错顺序的话其付出的代价更高。

另外一个问题：不同的查询，其相关文档数量差异很大，所以转换为文档对之后，有的查询可能有几百个对应的文档对，而有的查询只有十几个对应的文档对，这对机器学习系统的效果评价造成困难。我们设想有两个查询，查询 Q1 对应 500 个文档对，查询 Q2 对应 10 个文档对，假设学习系统对于查询 Q1 的文档对能够判断正确 480 个，对于查询 Q2 的文档对能够判断正确 2 个，如果从总的文档对数量来看，这个学习系统的准确率是  $(480+2)/(500+10)=0.95$ ，即 95% 的准确率，但是从查询的角度，两个查询对应的准确率分别为：96% 和 20%，两者平均为 58%，与纯粹从文档对判断的准确率相差甚远，这对如何继续调优机器学习系统会带来困扰。

#### 5.5.4 文档列表方法 (ListWise Approach)

单文档方法将训练集里每一个文档当做一个训练实例，文档对方法将同一个查询的搜索结果里任意两个文档对作为一个训练实例，文档列表方法与上述两种表示方式不同，是

将每一个查询对应的所有搜索结果列表整体作为一个训练实例，这也是为何称之为文档列表方法的原因。

文档列表方法根据  $K$  个训练实例（一个查询及其对应的所有搜索结果评分作为一个实例）训练得到最优评分函数  $F$ ，对于一个新的用户查询，函数  $F$  对每一个文档打分，之后按照得分顺序由高到低排序，就是对应的搜索结果。

所以关键问题是：拿到训练数据，如何才能训练得到最优的打分函数？本节介绍一种训练方法，它是基于搜索结果排列组合的概率分布情况来训练的，图 5-16 是这种方式训练过程的图解示意。首先解释下什么是搜索结果排列组合的概率分布，我们知道，对于搜索引擎来说，用户输入查询  $Q$ ，搜索引擎返回搜索结果，我们假设搜索结果集合包含 A、B 和 C 3 个文档，搜索引擎要对搜索结果排序，而这 3 个文档的顺序共有 6 种排列组合方式：ABC，ACB，BAC，BCA，CAB 和 CBA，而每种排列组合都是一种可能的搜索结果排序方法。

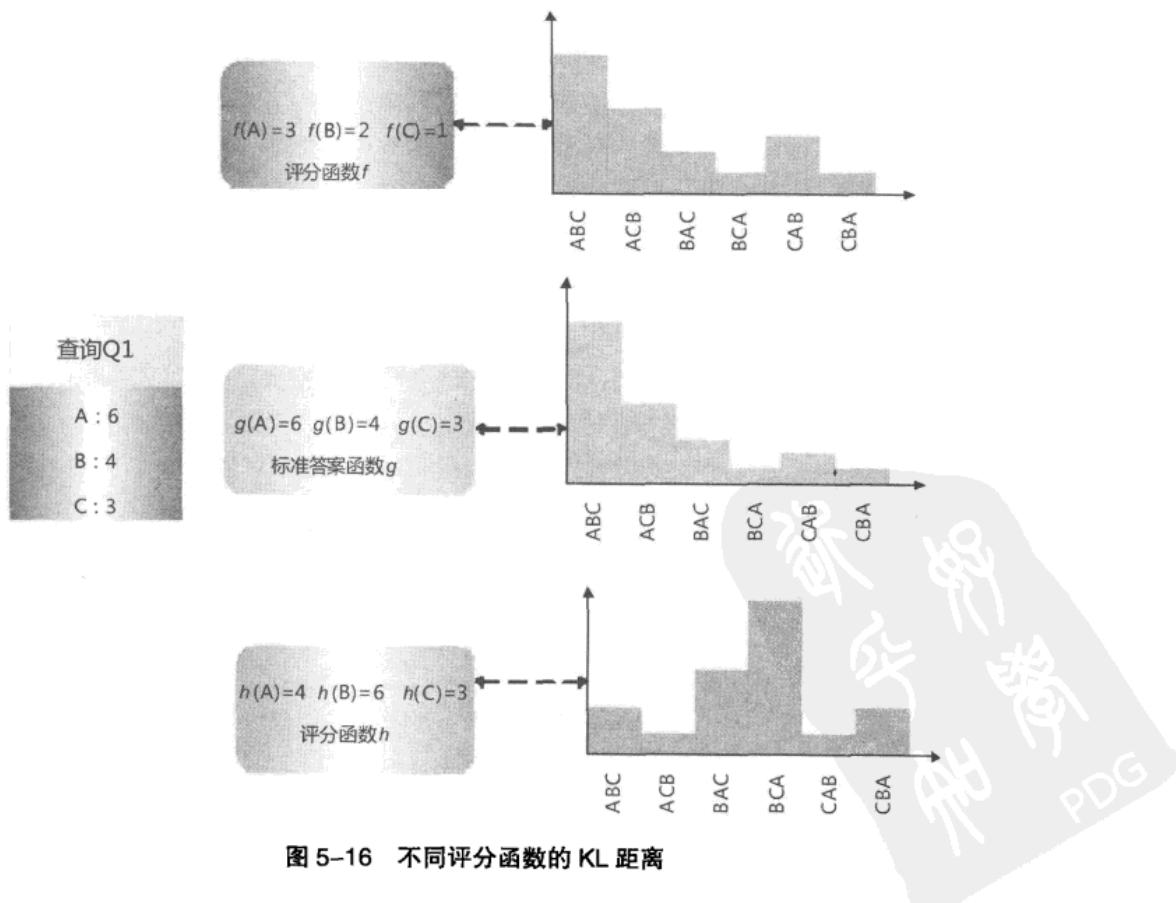


图 5-16 不同评分函数的 KL 距离



对于某个评分函数  $F$  来说，对 3 个搜索结果文档的相关性打分，得到 3 个不同的相关度得分  $F(A)$ 、 $F(B)$  和  $F(C)$ ，根据这 3 个得分就可以计算 6 种排列组合情况各自的概率值。不同的评分函数，其 6 种搜索结果排列组合的概率分布是不一样的。

了解了什么是搜索结果排列组合的概率分布，我们介绍如何根据训练实例找到最优的评分函数。图 5-16 展示了一个具体的训练实例，即查询 Q1 及其对应的 3 个文档的得分情况，这个得分是由人工打上去的，所以可以看做是标准答案。可以设想存在一个最优的评分函数  $g$ ，对查询 Q1 来说，其打分结果是：A 文档得 6 分，B 文档得 4 分，C 文档得 3 分，因为得分是人工打的，所以具体这个函数  $g$  是怎样的我们不清楚，我们的任务就是找到一个函数，使得函数对 Q1 的搜索结果打分顺序和人工打分顺序尽可能相同。既然人工打分（虚拟的函数  $g$ ）已知，那么我们可以计算函数  $g$  对应的搜索结果排列组合概率分布，其具体分布情况如图 5-16 中间的概率分布所示。假设存在两个其他函数  $h$  和  $f$ ，它们的计算方法已知，对应的对 3 个搜索结果的打分在图上可以看到，由打分结果也可以推出每个函数对应的搜索结果排列组合概率分布，那么  $h$  和  $f$  哪个与虚拟的最优评分函数  $g$  更接近呢？一般可以用两个分布概率之间的距离远近来度量这种相似性，KL 距离就是一种衡量概率分布差异大小的计算工具，通过分别计算  $h$  与  $g$  的差异大小及  $f$  与  $g$  的差异大小，可以看出  $f$  比  $h$  更接近于虚拟的最优函数  $g$ ，那么在这两个函数中，我们应该优选  $f$  作为将来搜索可用的评分函数。训练过程就是在可能的函数中寻找最接近虚拟最优函数  $g$  的那个函数作为训练结果，将来作为在搜索时的评分函数。

上述例子只是描述了对于单个训练实例如何通过训练找到最优函数，事实上我们有  $K$  个训练实例，虽然如此，其训练过程与上述说明是类似的，可以认为存在一个虚拟的最优评分函数  $g$ （实际上是人工打分），训练过程就是在所有训练实例基础上，探寻所有可能的候选函数，从中选择那个 KL 距离最接近于函数  $g$  的，以此作为实际使用的评分函数。

经验结果表明，基于文档列表方法的机器学习排序效果要好于前述两种方法。

## 5.6 检索质量评价标准

在搜索系统最终推出之前，一般都要对其性能进行评测。除了时间和空间等运行效率方面的评测外，更重要的是对搜索结果质量进行评测。研发人员可以根据测试结果选择效果较好的搜索技术，或验证搜索系统在真实环境中运行时的实际效果，以辅助系统不断进行设计、研究和改进。因此搜索系统的性能评测对于系统的研制和发展是至关重要的。

如何评价搜索结果质量呢？最广为接受的评价标准是用精确率和召回率这两个指标来评价搜索质量。

### 5.6.1 精确率与召回率

给定一个固定的用户搜索请求，搜索系统将系统认为和用户请求相关的文档返回给用户。对于这次搜索行为，可以根据两个维度来将所有文档构成的集合划分成 4 个互不相交的子集（参考图 5-17）。一个维度是：“该文档是否与用户发出的搜索请求相关”，由此维度，可以将整个文档集合划分为相关与不相关两种类型，图 5-17 中的第 1 列表示相关文档，第 2 列表示不相关文档；第 2 个维度是：“文档是否在本次搜索结果列表里”，由此维度，可以将整个文档集合划分为“在本次搜索结果列表”与“不在本次搜索结果列表”两种类型，图 5-17 中的第 1 行表示本次搜索结果包含的文档列表，第 2 行表示集合中不在本次搜索结果列表中出现的其他文档。

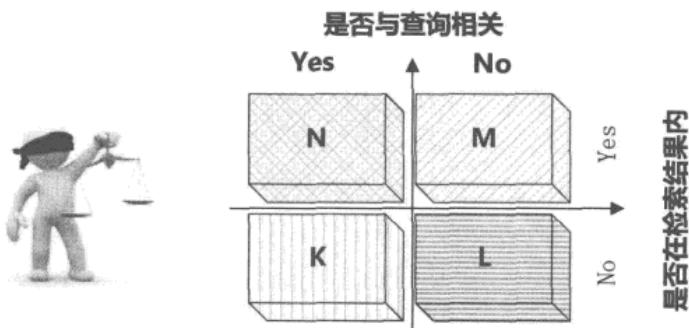


图 5-17 观察文档集合的两个维度

将以上两个划分维度组合，把文档集合切割为 4 个互不相交的子集。如图 5-17 所示坐标中，左上角的子集代表“在本次搜索结果中与搜索请求相关”的文档，假设集合大小为  $N$ ；右上角的子集代表“在本次搜索结果中与搜索请求不相关”的文档，假设集合大小为  $M$ ；左下角的子集代表“在本次搜索结果之外与搜索请求相关”的文档，即那些本来应该由搜索系统返回但因为算法原因没有找到的相关文档，假设集合大小为  $K$ ；右下角的子集代表“在本次搜索结果之外且与搜索请求无关”的文档，假设集合大小为  $L$ 。

在将文档集合划分为 4 个子集的基础上，我们可以对精确率和召回率进行定量描述，如图 5-18 所示是这两个指标的计算方法。

所谓精确率，就是本次搜索结果中相关文档所占的比例，分子为本次搜索结果中的相关文档（即图 5-17 中的左上角子集），分母为本次搜索结果包含的所有文档（即图 5-17 中的第 1 行），两者相除得到精确率。

所谓召回率，即本次搜索结果中包含的相关文档占整个集合中所有相关文档的比例，分子与精确率分子相同，即本次搜索结果中包含的相关文档，分母为整个文档集合所包含



的所有相关文档（即图 5-17 中的第 1 列），两者相除得到召回率。召回率用于评价搜索系统是否把该找出的文档都找出来了。

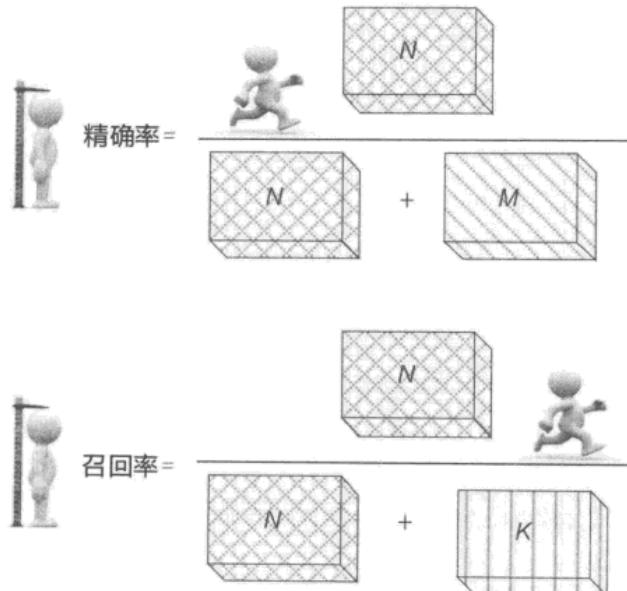


图 5-18 精确率与召回率

精确率和召回率是常见的评估检索系统的指标，但是对于搜索引擎来说，精确率更为重要，因为搜索引擎处理海量数据，一方面在这种环境下，对于某个查询，找到与这个查询相关的所有文档（也即计算召回率公式的分母）难度很大，导致召回率很难准确计算；另外一方面由于数据量比较大，所以能够满足用户需求的文档量也很大，用户很少需要看到所有相关文档，往往是看到一部分即可满足搜索需求，全部召回相关文档对于满足用户需求意义也不是特别重要。而相对应地，精确率在搜索引擎场景下就非常重要了，因为在搜索列表前列的搜索结果如果有太多不相关的内容，直接影响用户体验，所以对于搜索引擎质量评估来说，往往更加关注精确率。

上面介绍的精确率和召回率的计算方法只是通用的计算框架，在具体评估时，需要做更加精细的考虑。常用的评估搜索引擎精度的指标有 P@10 和 MAP。

### 5.6.2 P@10 指标

p@10 指标更关注搜索结果排名最靠前文档的结果质量，它用于评估在搜索结果排名最靠前的头 10 个文档中有多大比例是相关的。图 5-19 是 p@10 计算的一个示例，打对钩的文档代表与用户查询相关，叉号代表无关，在这个例子中，头 10 个文档中包含了 5 个相

关文档，所以其精度为 0.5。

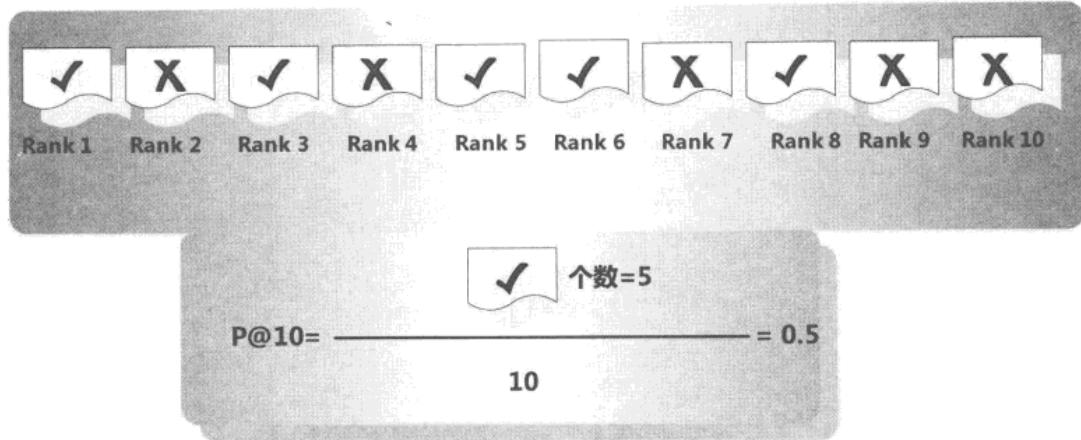


图 5-19 p@10 指标

### 5.6.3 MAP 指标 ( Mean Average Precision )

MAP 指标是针对多次查询的平均准确率衡量标准，是评价检索系统质量的常用指标，如果习惯阅读信息检索相关学术论文的话，会经常在论文中遇到这个评价指标。

要了解 MAP，首先需要了解 AP (Average Precision)。MAP 是衡量多个查询的平均检索质量的，而 AP 是衡量单个查询的检索质量的，图 5-20 是如何计算某次检索的 AP 得分的示意图。例子中假设与用户查询相关的文档有 3 个，经过搜索系统输出后，分别排在搜索结果的第 2 位、第 4 位和第 6 位，如果是一个理想的搜索系统，理论上应该将这 3 个文档排在第 1 位、第 2 位和第 3 位，所以用这 3 个文档的理想排名位置除以实际排名位置，会得到每个文档的得分，3 个文档求平均值得到本次搜索的 AP 值 0.5。AP 值越高，则意味着越接近理想的搜索结果，说明检索系统质量越好。如果例子中的 3 个相关文档分别处于搜索结果的第 1 位、第 2 位和第 3 位，那么 AP 值为 1，这就是理想的搜索结果。AP 指标兼顾了排在前列结果的相关性和系统召回率，这是为何被经常采用的原因。

AP 是针对单次查询的衡量指标，如果存在多组查询，那么每个查询都会有自己的 AP 值，对这些查询的 AP 值求平均值，就得到了 MAP 指标。

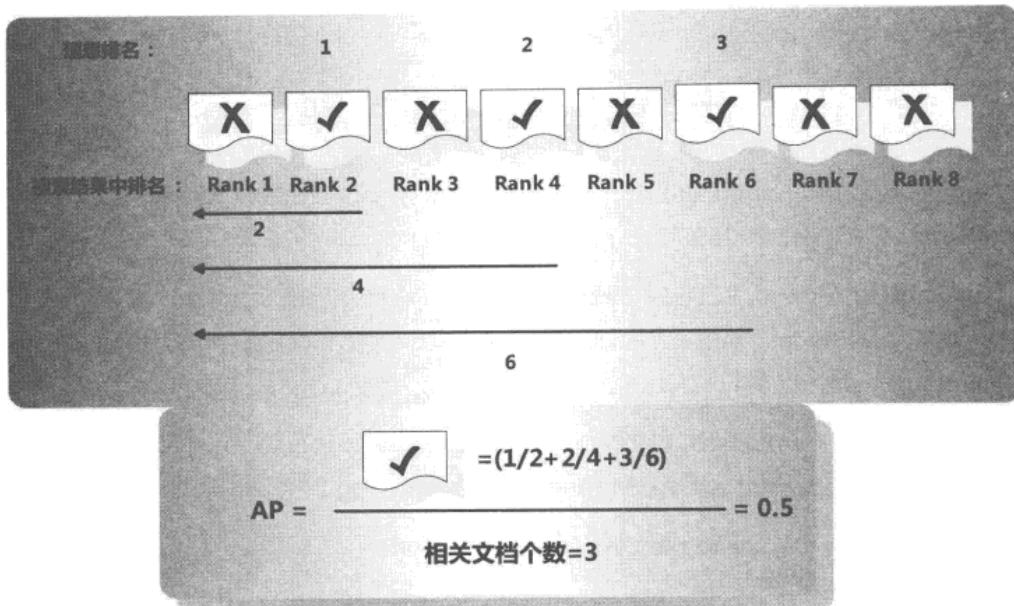


图 5-20 AP 计算过程

## 本章提要

- 检索模型是搜索引擎排序的理论基础，用来计算网页和用户查询的相关性。
- 几种常用的检索模型包括：布尔模型、向量空间模型、概率模型、语言模型及最近几年兴起的机器学习排序算法。
- 目前大部分商业搜索引擎采用概率模型作为相关性排序模型，而 BM25 则是目前效果最好的概率检索模型。
- 精确率和召回率是评价检索系统的常用指标，而对于搜索引擎来说，精确率尤为重要。

## 本章参考文献

- [1] Croft et all., (2009). Search Engines: information retrieval in practice. Pearson Education Asion Limited.
- [2] Salton, G., Fox, E.A., Wu, H. (1983). Extended boolean information retrieval. Communications of the ACM , 26(12), pp.1022-1036.
- [3] Salton, G. (1971). The SMART retrieval system-experiments in automatic document

processing. Englewood Cliffs, Prentice Hall.

- [4] Roberson, S.E. (1977). The probability ranking principle in information retrieval. *Journal of Documentation*, 33(4), pp. 294-304.
- [5] Fuhr, N. (1992). Probabilistic models in information retrieval [J]. *The Computer Journal*, 35(3):243-255.
- [6] Shaw, W.M, Burgin, R. and Howell, P. (1997). Performance standards and evaluation in IR test collections: Vector Space and other retrieval models. *Information Processing and Management*, 33(1), pp.15-36.
- [7] Song, F., and Croft, W.B. (1999). A general language model for information retrieval. In *Proceedings of the 22<sup>th</sup> ACM SIGIR Conference*. pp.279-280.
- [8] Li H.(2009). Learning to rank. ACML 2009 Tutorial.



# 第 6 章 链接分析

“方以类聚，物以群分，吉凶生矣。”

《周易·系辞上》

搜索引擎在查找能够满足用户请求的网页时，主要考虑两方面的因素：一方面是用户发出的查询与网页内容的内容相似性得分，即网页和查询的相关性，第 5 章已经就内容相似性计算做了介绍；另一方面就是通过链接分析方法计算获得的得分，即网页的重要性。搜索引擎融合两者，共同拟合出相似性评分函数，来对搜索结果进行排序。本章主要介绍一些著名的链接分析方法。

## 6.1 Web 图

互联网包含了海量网页，而网页和一般文本的一个重要区别是在页面内容中包含相互引用的链接（Link），如果将一个网页抽象成一个节点，而将网页之间的链接理解为一条有向边，则可以把整个互联网抽象为一个包含页面节点和节点之间联系边的有向图，称之为 Web 图。图 6-1 给出了 Web 图的形象化表示。

Web 图是对互联网的一种宏观抽象，其微观构成元素是一个单独的网页。对于某个单独的网页 A 来说（参见图 6-2），在其内容部分往往会包含指向其他网

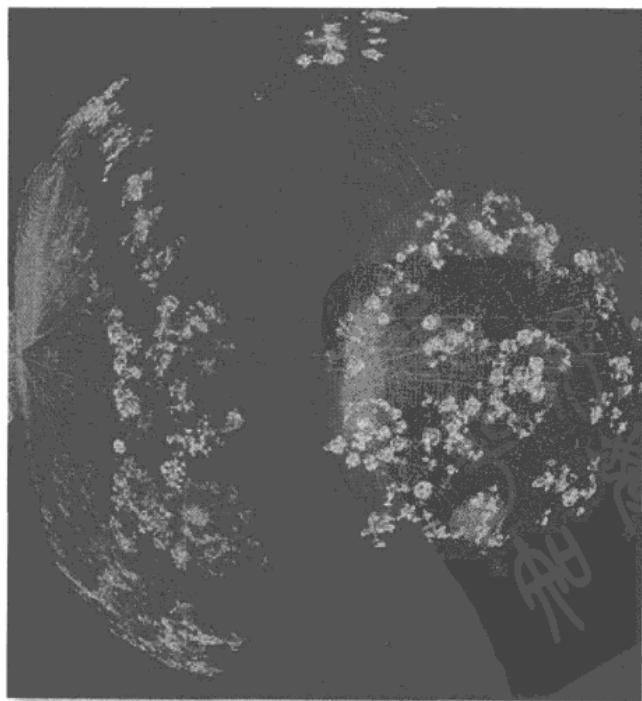


图 6-1 Web 图的形象化表示

页的链接，这些链接一般称为页面 A 的出链 (Out Link)。因为网页 A 是在一个网状结构中，所以不仅网页 A 会指向其他页面，也会有很多其他页面有链接指向网页 A，那么这些指向网页 A 的链接称为网页 A 的入链 (In Link)。在图 6-2 中，网页 A 有两条出链和一条入链。

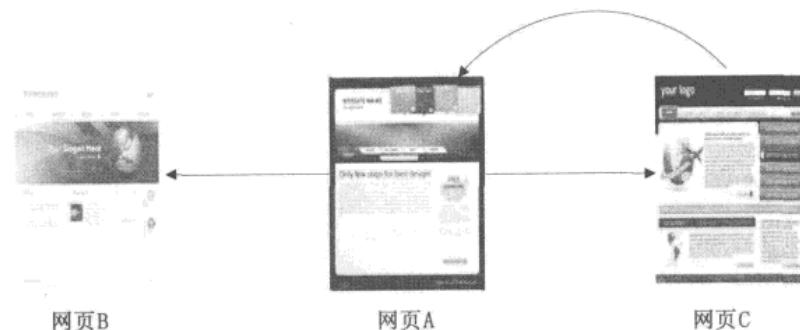


图 6-2 入链与出链

锚文字也是网页中一种常见且非常有用的信息。所谓锚文字，就是页面内某个出链附近的一些描述文字。之所以锚文字会比较重要，是因为锚文字往往是对目标网页的一种概括性描述，所以在很多技术方法里都会利用这个信息来代表目标网页的含义。图 6-3 给出了一个锚文字与链接之间的关系示意图。

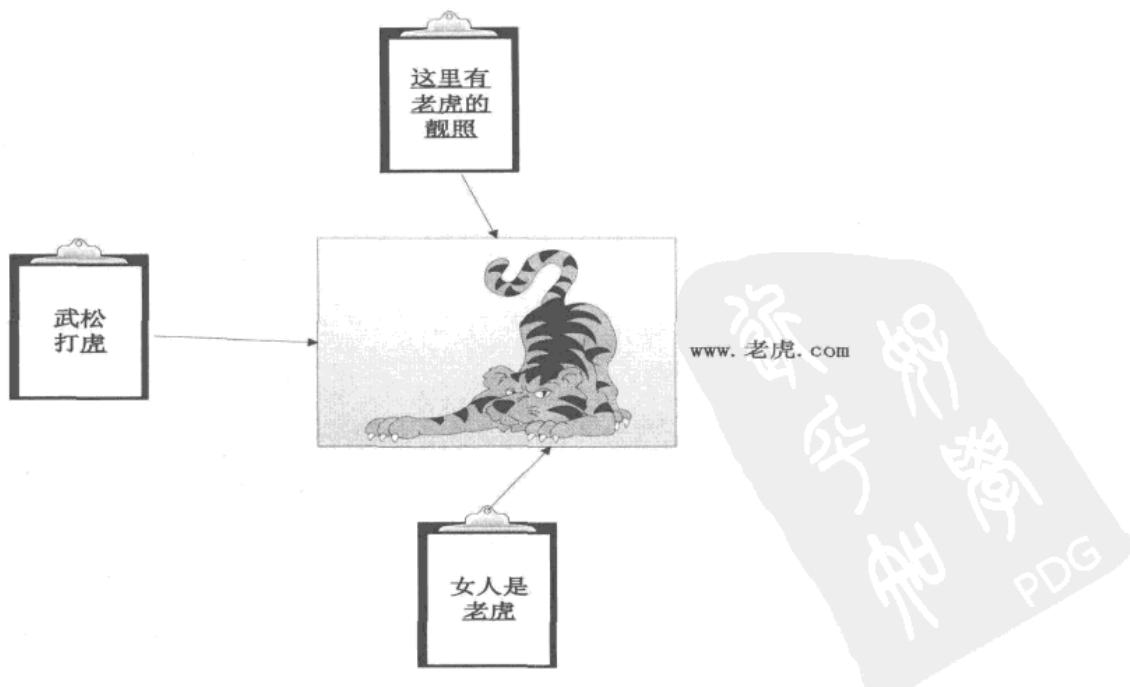


图 6-3 锚文字与链接之间的关系示意图



## 6.2 两个概念模型及算法之间的关系

在介绍具体链接分析算法之前，首先介绍两个概念模型，并对各个链接分析算法之间的关系进行说明，这样有助于读者从宏观角度理解各个算法的基本思路与传承关系。

### 6.2.1 随机游走模型 ( Random Surfer Model )

互联网用户在上网时，往往有类似的网络行为：输入网址，浏览页面，然后顺着页面的链接不断打开新的网页。随机游走模型就是针对浏览网页的用户行为建立的抽象概念模型。之所以要建立这个抽象概念模型，是因为包括 PageRank 算法在内的很多链接分析算法都是建立在随机游走模型基础上的。

图 6-4 给出了随机游走模型的示意图。在最初阶段，用户打开浏览器浏览第 1 个网页，假设我们有一个虚拟时钟用来计时，此时可以设定时间为 1，用户在看完网页后，对网页内某个链接指向的页面感兴趣，于是点击该链接，进入第 2 个页面，此时虚拟时钟再次计时，时钟走向数字 2，如果网页包含了  $k$  个出链，则用户从当前页面跳转到任意一个链接所指向页面的概率是相等的。用户不断重复以上过程，在相互有链接指向的页面之间跳转。如果对于某个页面所包含的所有链接，用户都没有兴趣继续浏览，则可能会在浏览器中输入另外一个网址，直接到达该网页，这个行为称为远程跳转 ( Teleporting )。假设互联网中共有  $m$  个页面，则用户远程跳转到任意一个页面的概率也是相等的，即为  $1/m$ 。随机游走模型就是一个对直接跳转和远程跳转两种用户浏览行为进行抽象的概念模型。

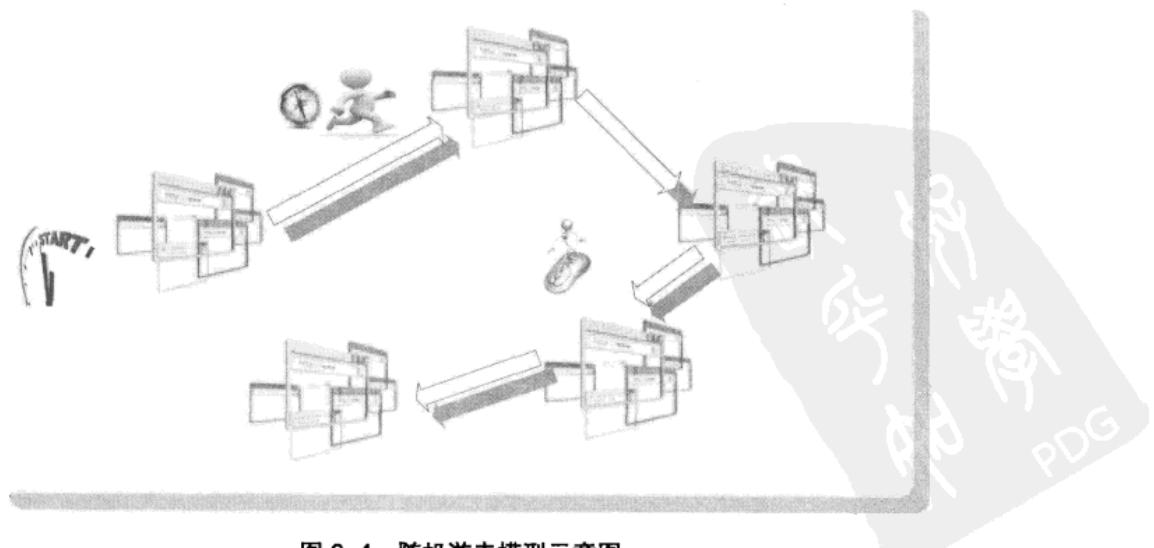


图 6-4 随机游走模型示意图

下面我们给出一个具体的随机游走模型的例子，为简单起见，该例子并未引入远程跳转行为。

在如图 6-5 所示的例子里，假设互联网由 A、B、C 3 个网页构成，其相互链接关系如图中页面节点之间的有向边所示。根据链接关系，即可计算页面节点之间的转移概率，比如对于节点 A 来说，只有唯一一个出链指向节点 B，所以从节点 A 跳转到节点 B 的概率为 1，对于节点 C 来说，其对节点 A 和 B 都有链接指向，所以转向任意一个其他节点的概率为  $1/2$ 。

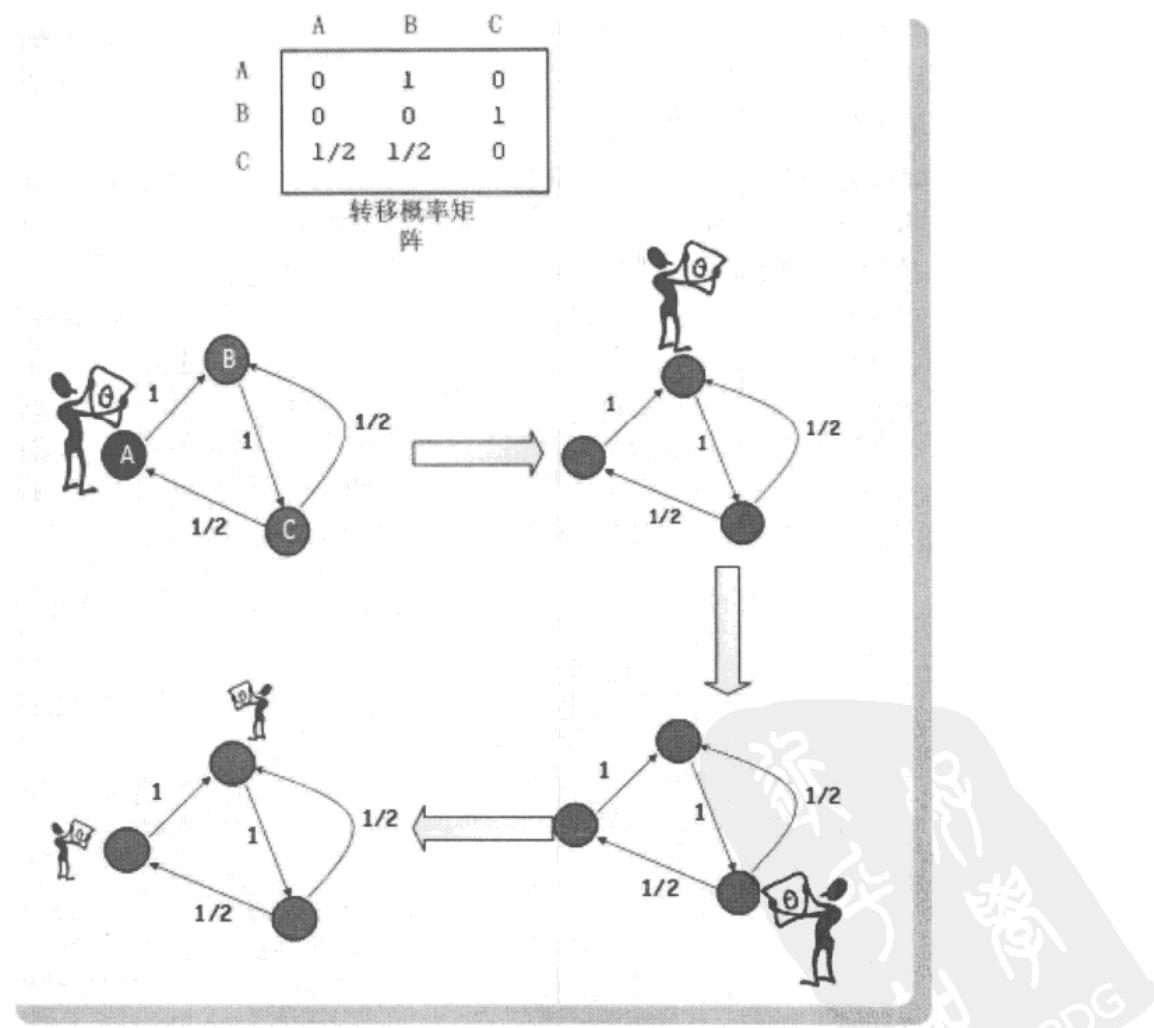


图 6-5 随机游走模型示例

假设在时刻 1，用户浏览页面 A，之后经由链接进入页面 B，然后进入页面 C，此时面



临两种可能选择，跳转进入页面 A 或者页面 B 皆可，两者概率相同，都为 1/2。

假设例子中的互联网包含不止 3 个页面，而是由 10 个页面构成，此时用户既不想跳回页面 A，也不想跳回页面 B，则可以按照 1/10 的概率跳入其他任意一个页面，即进行远程跳转。

### 6.2.2 子集传播模型

子集传播模型是从诸多链接分析算法中抽象出来的概念模型（参见图 6-6）。其基本思想是在做算法设计时，把互联网网页按照一定规则划分，分为两个甚至是多个子集合。其中某个子集合具有特殊性质，很多算法往往从这个具有特殊性质的子集合出发，给予子集合内网页初始权值，之后根据这个特殊子集合内网页和其他网页的链接关系，按照一定方式将权值传递到其他网页。

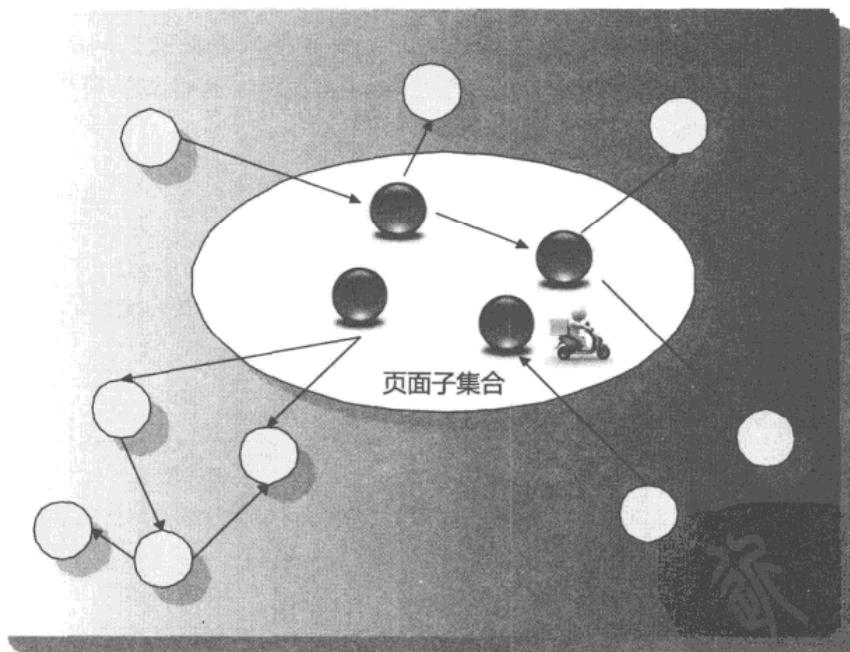


图 6-6 子集传播模型

本章介绍的一些链接分析算法符合子集传播模型，比如 HITS 算法和 Hilltop 算法及其衍生算法，在“网页反作弊”一章（第 8 章）会看到更多符合此模型的链接分析算法。

子集传播模型是个高度抽象的算法框架，很多算法可以认为是属于此框架的具体实例，即整体思路如上面所描述的流程，通常在以下几个方面各自存在不同。

- 如何定义特殊子集合，即在确定子集合内的网页应该有哪些特殊性质，具体算法规则不同。
- 在确定了特殊子集合所具有的性质后，如何对这个特殊子集合内网页给予一定的初始分值？不同算法打分方式各异。
- 从特殊子集合将其分值传播到其他网页时，采取何种传播方式？可传播的距离有多远？不同算法在此阶段也大都有差异。

注意：子集传播模型是本书作者从具体链接分析算法中归纳出的抽象模型，未见有文献明确提出，请读者阅读时谨慎参考。

### 6.2.3 链接分析算法之间的关系

到目前为止，学术界已经提出了很多链接分析算法，图 6-7 列出了其中影响力较大的一些算法及其相互关系，图中不同算法之间的箭头连接代表算法之间的改进关系，比如 SALSA 算法即融合了 PageRank 和 HITS 算法的基本思路。其他算法所代表的关系与此类似，可在图中明显看出算法间的传承关系。

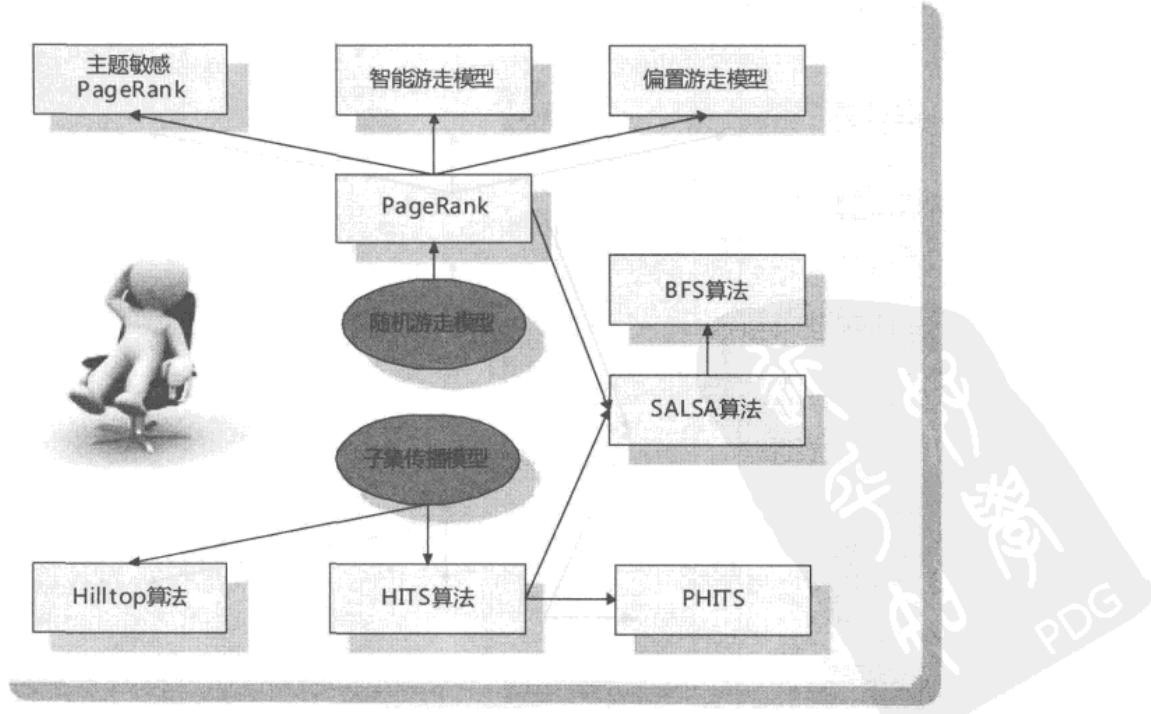


图 6-7 链接分析算法关系图



尽管链接算法很多，但是从其概念模型来说，基本遵循上述小节介绍的随机游走模型和子集传播模型。而从图中可看出，在众多算法中，PageRank 和 HITS 算法可以说是最主要的两个具有代表性的链接分析算法，后续的很多链接分析算法都是在这两个算法基础上衍生出来的改进算法。

在本章后续章节中，将详细介绍 PageRank 算法、HITS 算法、SALSA 算法、主题敏感 PageRank 算法和 Hilltop 算法。这些算法大都已被不同的商业搜索引擎所采用，在实际生活中发挥了很重要的作用。对于图 6-7 所列出的其他链接分析算法，将在本章末尾简述其原理。

### 6.3 PageRank 算法

PageRank 是 Google 创始人于 1997 年构建早期的搜索系统原型时提出的链接分析算法（参见图 6-8），自从 Google 在商业上获得空前的成功后，该算法也成为其他搜索引擎和学术界十分关注的计算模型。目前很多重要的链接分析算法都是在 PageRank 算法基础上衍生出来的。



图 6-8 Google 提出的 PageRank 算法

#### 6.3.1 从入链数量到 PageRank

在 PageRank 提出之前，已经有研究者提出利用网页的入链数量来进行链接分析计算，这种入链方法假设一个网页的入链越多，则该网页越重要。早期的很多搜索引擎也采纳了入链数量作为链接分析方法，对于搜索引擎效果提升也有较明显的效果。

PageRank 除了考虑到入链数量的影响，还参考了网页质量因素，两者相结合获得了更好的网页重要性评价标准。

对于某个互联网网页 A 来说，该网页 PageRank 的计算基于以下两个基本假设：

- **数量假设：**在 Web 图模型中，如果一个页面节点接收到的其他网页指向的入链数量越多，那么这个页面越重要。
- **质量假设：**指向页面 A 的入链质量不同，质量高的页面会通过链接向其他页面传递更多的权重。所以越是质量高的页面指向页面 A，则页面 A 越重要。

通过利用以上两个假设，PageRank 算法刚开始赋予每个网页相同的重要性得分，通过迭代递归计算来更新每个页面节点的 PageRank 得分，直到得分稳定为止。

PageRank 计算得出的结果是网页的重要性评价，这和用户输入的查询是没有任何关系的，即算法是主题无关的。假设有一个搜索引擎，其相似度计算函数不考虑内容相似因素，完全采用 PageRank 来进行排序，那么这个搜索引擎的表现是什么样子的呢？这个搜索引擎对于任意不同的查询请求，返回的结果都是相同的，即返回 PageRank 值最高的页面。

### 6.3.2 PageRank 计算

本节探讨 PageRank 的具体计算过程。PageRank 的计算充分利用了上节提到的两个假设：数量假设和质量假设。网页通过链接关系构建起 Web 图，在初始阶段，每个页面设置相同的 PageRank 值，通过若干轮的计算，会得到每个页面所获得的最终 PageRank 值。随着每一轮的计算进行，网页当前的 PageRank 值会不断得到更新。

在一轮更新页面 PageRank 得分的计算中，每个页面将其当前的 PageRank 值平均分配到本页面包含的出链上，这样每个链接即获得了相应的权值。而每个页面将所有指向本页面的入链所传入的权值求和，即可得到新的 PageRank 得分。当每个页面都获得了更新后的 PageRank 值，就完成了一轮 PageRank 计算。

下面以如图 6-9 所示的例子来说说明 PageRank 的具体计算过程。图中包含 7 个页面，其页面编号分别从 1 到 7，页面之间的链接关系如图所示。这里要注意的一点是：如图 6-9 所示的情况已是经过若干轮计算之后的情形，每个页面已经获得了当前的 PageRank 分值，比如页面 1 当前的 PageRank 分值为 0.304，页面 2 当前的 PageRank 分值为 0.166，其他页面的对应 PageRank 数值也在图中标出。我们接下来看看从当前的状态出发，如何进行下一轮的 PageRank 计算。

在 PageRank 计算中，从页面 A 指向页面 B 的某个链接的权值，代表了从页面 A 传导到页面 B 的 PageRank 分值。而对于页面 A 来说，如果有多个出链，那么页面 A 本身的 PageRank 分值会均等地分配给每个链接。比如图 6-9 中的页面 4，其当前 PageRank 分值为 0.105，包含 3 个出链，分别指向页面 2、页面 3 和页面 5，所以每个出链获得的分值为 0.035。



图 6-9 中其他链接的权值也与页面 4 的出链一样，通过类似计算可以得到。

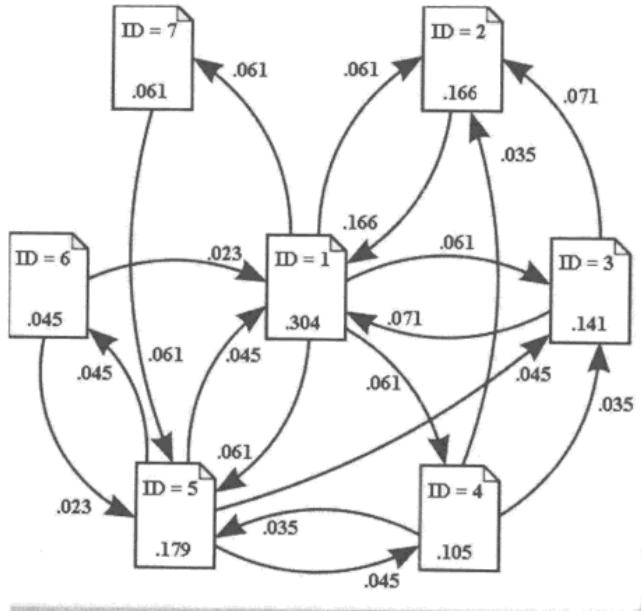


图 6-9 PageRank 计算实例

获得了每个链接传导的权值后，即可进行新一轮的 PageRank 计算。要想得到各个页面节点的得分，只需将网页入链所传入的分值汇总即可。比如图 6-9 中的页面 1，有 4 个入链，分别来自于页面 2、3、5、6。每个链接传导的权值也如图上所标，那么页面 1 的新的 PageRank 值为 4 个入链传入的权值之和，即：

$$\text{PR}(1)=0.166+0.071+0.045+0.023=0.305$$

即得分从上一轮的 0.304 更新到 0.305。

其他页面节点也依次如此计算，以获得新的 PageRank 分值，当所有页面节点的分值得到更新，就完成了一轮 PageRank 计算。如果在新一轮 PageRank 计算之后，发现总体而言，页面节点的 PageRank 值基本稳定，不再发生较大变化，即可结束 PageRank 的计算，以此时得到的得分，作为最后排序时可以利用的 PageRank 得分。

### 6.3.3 链接陷阱 (Link Sink) 与远程跳转 (Teleporting)

互联网页面之间的链接结构实际上很复杂，上一小节介绍了 PageRank 的计算过程，但是对于某些特殊的链接结构，按照上述方法计算 PageRank 会导致问题，一个典型的例子就是“链接陷阱”（参见图 6-10）。

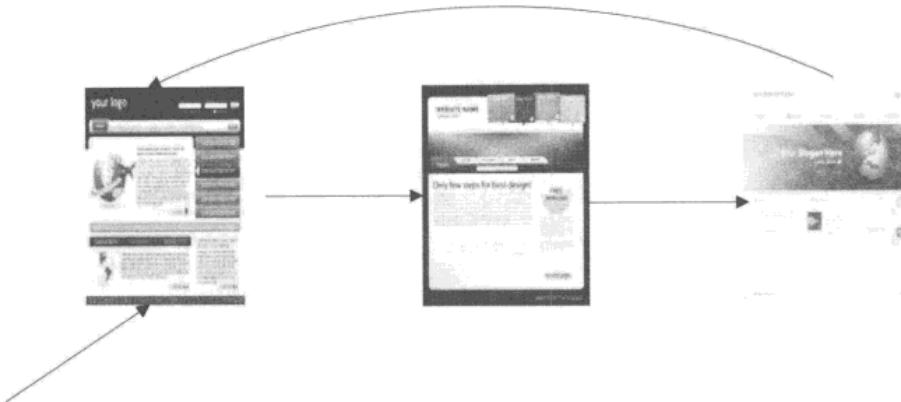


图 6-10 链接陷阱

图 6-10 包含了 3 个网页，相互有链接指向，形成了一个环形结构。这种结构类似于天体中的黑洞，在计算 PageRank 的时候，该结构将导致系统只会吸收传入的分值，而不能将获得的分值传播出去，随着 PageRank 一轮轮地连续运算，链接陷阱内的页面 PageRank 得分越来越高，这与 PageRank 的设计初衷相违背。

远程跳转是解决链接陷阱的通用方式，所谓的远程跳转，即在网页向外传递分值的时候，不限于向出链所指网页传递，也可以以一定的概率向任意其他网页跳转。对于链接陷阱内的网页来说，增加了远程跳转措施后，就像为每个页面增加了指向互联网任意其他页面的虚拟边，权值可以通过这种虚拟边向外传递，以此来避免链接陷阱导致的问题。

## 6.4 HITS 算法 ( Hypertext Induced Topic Selection )

HITS 算法也是链接分析中非常基础且重要的算法，目前已已被 Teoma 搜索引擎 ([www.teoma.com](http://www.teoma.com)) 作为链接分析算法在实际中使用。

### 6.4.1 Hub 页面与 Authority 页面

Hub 页面和 Authority 页面是 HITS 算法最基本的两个定义。所谓 Authority 页面，是指与某个领域或者某个话题相关的高质量网页。比如搜索引擎领域，Google 和百度首页即该领域的高质量网页；比如视频领域，优酷和土豆首页即该领域的高质量网页。所谓的 Hub 页面，指的是包含了很多指向高质量 Authority 页面链接的网页，比如 hao123 首页可以认为是一个典型的高质量 Hub 网页。

图 6-11 给出了一个 Hub 页面实例，这个网页是斯坦福大学计算语言学研究组维护的页



面，这个网页收集了与统计自然语言处理相关的高质量资源，包括一些著名的开源软件包及语料库等，并通过链接的方式指向这些资源页面。这个页面可以认为是“自然语言处理”这个领域的 Hub 页面，相应地，被这个页面指向的资源页面，大部分是高质量的 Authority 页面。

<u><a href="#">Statistical natural language processing and corpus-based computation linguistics: An annotated list of resources</a></u>	
<u><a href="#">Contents</a></u>	
<ul style="list-style-type: none"> <li><u><a href="#">Tools: Machine Translation, POS Taggers, NP chunking, Sequence models, Parsers, Semantic Parsers/SRL, NER, Coreference, Language models, Concordances, Summarization, Other</a></u></li> <li><u><a href="#">Corpora: Large collections, Particular languages, Treebanks, Discourse, WSD, Literature, Acquisition</a></u></li> <li><u><a href="#">SGML/XML</a></u></li> <li><u><a href="#">Dictionaries</a></u></li> <li><u><a href="#">Lexical/morphological resources</a></u></li> <li><u><a href="#">Courses, Syllabi, and other Educational Resources</a></u></li> <li><u><a href="#">Mailing lists</a></u></li> <li><u><a href="#">Other stuff on the Web General, IR, IE/Wrappers, People, Societies</a></u></li> </ul>	
<u><a href="#">Tools</a></u>	
<u><a href="#">Machine Translation systems</a></u>	

图 6-11 自然语言处理领域的 Hub 页面

HITS 算法的目的是通过一定的技术手段，在海量网页中找到与用户查询主题相关的高质量 Authority 页面和 Hub 页面，尤其是 Authority 页面，因为这些页面代表了能够满足用户查询的高质量内容，搜索引擎以此作为搜索结果返回给用户。

#### 6.4.2 相互增强关系

很多算法都是建立在一些假设之上的，HITS 算法也不例外。HITS 算法隐含并利用了两个基本假设：

- **基本假设 1：**一个好的 Authority 页面会被很多好的 Hub 页面指向。
- **基本假设 2：**一个好的 Hub 页面会指向很多好的 Authority 页面。

到目前为止，无论是从 Hub 页面或者 Authority 页面的定义也好，还是从两个基本假设也好，都能看到一个模糊的描述，即“高质量”或者“好的”，那么什么是“好的” Hub 页面？什么是“好的” Authority 页面？两个基本假设给出了所谓“好”的定义。

基本假设 1 说明了什么是“好的” Authority 页面，即被很多好的 Hub 页面指向的页面是好的 Authority 页面，这里两个修饰语非常重要：“很多”和“好的”，所谓“很多”，即被越多的 Hub 页面指向越好，所谓“好的”，意味着指向该页面的 Hub 页面质量越高，则页面越好。这综合了指向本页面的所有 Hub 节点的数量和质量因素。

基本假设 2 则给出了什么是“好的” Hub 页面的说明，即指向很多好的 Authority 页面

的网页是好的 Hub 页面。同样地，“很多”和“好的”两个修饰语很重要，所谓“很多”，即指向的 Authority 页面数量越多越好；所谓“好的”，即指向的 Authority 页面质量越高，则该页面越是好的 Hub 页面。这也综合考虑了该页面有链接指向的所有页面的数量和质量因素。

从以上两个基本假设可以推导出 Hub 页面和 Authority 页面之间的相互增强关系（参考图 6-12），即某个网页的 Hub 质量越高，则其链接指向的页面的 Authority 质量越好；反过来也是如此，一个网页的 Authority 质量越高，则那些有链接指向本网页的页面 Hub 质量越高。通过这种相互增强关系不断迭代计算，即可找出哪些页面是高质量的 Hub 页面，哪些页面是高质量的 Authority 页面。

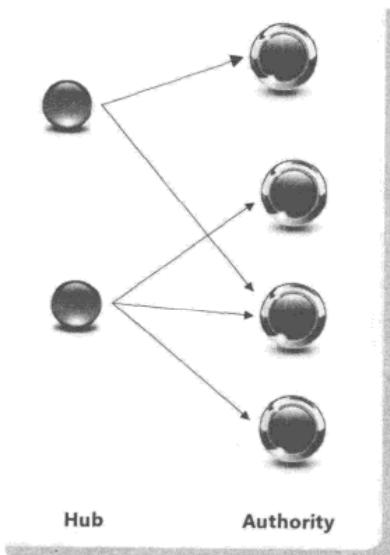


图 6-12 相互增强关系

#### 6.4.3 HITS 算法

HITS 算法与 PageRank 算法一个显著的差异是：HITS 算法与用户输入的查询请求密切相关，而 PageRank 算法是与查询无关的全局算法。HITS 后续计算步骤都是在接收到用户查询后展开的，即是与查询相关的链接分析算法。

HITS 算法接收到了用户查询之后，将查询提交给某个现有的搜索引擎（或者是自己构造的检索系统），并在返回的搜索结果中，提取排名靠前的网页，得到一组与用户查询高度相关的初始网页集合，这个集合被称做根集（Root Set）。



在根集的基础上，HITS 算法对网页集合进行扩充（参考图 6-13），扩充原则是：凡是与根集内网页有直接链接指向关系的网页都被扩充进来，无论是有链接指向根集内页面也好，或者是根集页面有链接指向的页面也好，都被扩充进入扩展网页集合。HITS 算法在这个扩展网页集合内寻找好的 Hub 页面与好的 Authority 页面。

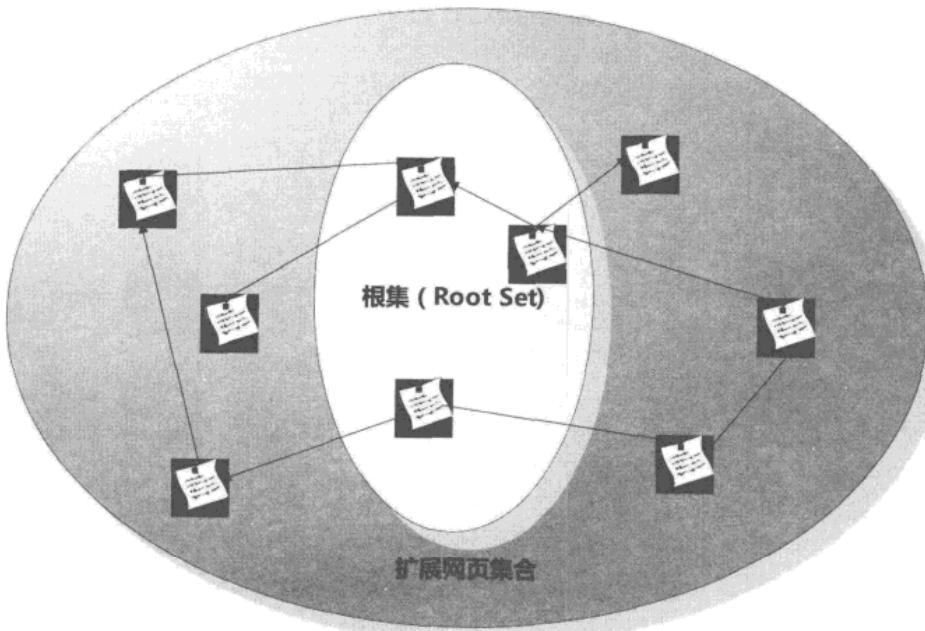


图 6-13 根集与扩展网页集合

对于扩展网页集合来说，我们并不知道哪些页面是好的 Hub 页面或者好的 Authority 页面，每个网页都有潜在的可能，所以对于每个页面都设立两个权值，分别来记载这个页面是好的 Hub 页面或者 Authority 页面的可能性。在初始情况下，在没有更多可利用信息前，每个页面的这两个权值都是相同的，可以都设置为 1。

之后，即可利用上面提到的两个基本假设，以及相互增强关系等原则进行多轮迭代计算，每轮迭代计算更新每个页面的两个权值，直到权值稳定不再发生明显的变化为止。

图 6-14 给出了迭代计算过程中，某个页面的 Hub 权值和 Authority 权值的更新方式。假设以  $A(i)$  代表网页  $i$  的 Authority 权值，以  $H(i)$  代表网页  $i$  的 Hub 权值。在如图 6-14 所示的例子中，扩展网页集合有 3 个网页有链接指向页面 1，同时页面 1 有 3 个链接指向其他页面。那么，网页 1 在此轮迭代中的 Authority 权值即为所有指向网页 1 页面的 Hub 权值之和；类似地，网页 1 的 Hub 分值即为所指向的页面的 Authority 权值之和。

扩展网页集合内其他页面也以类似的方式对两个权值进行更新，当每个页面的权值都

获得了更新，则完成了一轮迭代计算，此时 HITS 算法会评估上一轮迭代计算中的权值和本轮迭代之后权值的差异，如果发现总体来说权值没有明显变化，说明系统已进入稳定状态，则可以结束计算。将页面根据 Authority 权值得分由高到低排序，取权值最高的若干页面作为响应用户查询的搜索结果输出。如果比较发现两轮计算总体权值差异较大，则继续进入下一轮迭代计算，直到整个系统权值稳定为止。

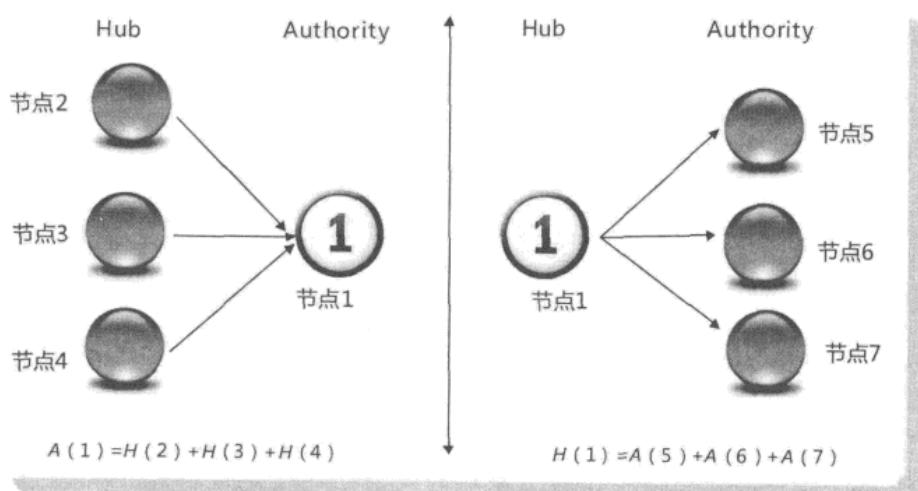


图 6-14 Hub 与 Authority 权值计算

#### 6.4.4 HITS 算法存在的问题

HITS 算法整体而言是个效果很好的算法，目前不仅在搜索引擎领域应用，而且被自然语言处理及社交分析等很多其他计算机领域借鉴使用，并取得了很好的应用效果。尽管如此，最初版本的 HITS 算法仍然存在一些问题，而后续很多基于 HITS 算法的链接分析方法，也是立足于改进 HITS 算法存在的这些问题而提出的。

归纳起来，HITS 算法主要在以下几个方面存在不足。

##### 计算效率较低

因为 HITS 算法是与查询相关的算法，所以必须在接收到用户查询后实时进行计算，而 HITS 算法本身需要进行很多轮迭代计算才能获得最终结果，这导致其计算效率较低，这是实际应用时必须慎重考虑的问题。

##### 主题漂移问题

如果在扩展网页集合里包含部分与查询主题无关的页面，而且这些页面之间有较多的



相互链接指向，那么使用 HITS 算法很可能会给予这些无关网页很高的排名，导致搜索结果发生主题漂移，这种现象被称为紧密链接社区现象（Tightly-Knit Community Effect）。

#### 易被作弊者操纵结果

HITS 算法从机制上很容易被作弊者操纵，比如作弊者可以建立一个网页，页面内容增加很多指向高质量网页或者著名网站的网址，这就是一个很好的 Hub 页面，之后作弊者再将这个网页链接指向作弊网页，于是可以提升作弊网页的 Authority 得分。

#### 结构不稳定

所谓结构不稳定，就是说在原有的扩展网页集合内，如果添加删除个别网页或者改变少数链接关系，则 HITS 算法的排名结果就会有非常大的改变。

### 6.4.5 HITS 算法与 PageRank 算法比较

HITS 算法和 PageRank 算法可以说是搜索引擎链接分析的两个最基础且最重要的算法。从以上对两个算法的介绍可以看出，两者无论是在基本概念模型，还是计算思路及技术实现细节都有很大的不同，下面对两者之间的差异进行逐一说明。

- HITS 算法是与用户输入的查询请求密切相关的，而 PageRank 与查询请求无关。所以，HITS 算法可以单独作为相似性计算评价标准，而 PageRank 必须结合内容相似性计算才可以用来对网页相关性进行评价。
- HITS 算法因为与用户查询密切相关，所以必须在接收到用户查询后进行实时计算，计算效率较低；而 PageRank 则可以在爬虫抓取完成后离线计算，在线直接使用计算结果，计算效率较高。
- HITS 算法的计算对象数量较少，只需计算扩展集合内网页之间的链接关系；而 PageRank 是全局性算法，对所有互联网页面节点进行处理。
- 从两者的计算效率和处理对象集合大小来比较，PageRank 更适合部署在服务器端，而 HITS 算法更适合部署在客户端。
- HITS 算法存在主题泛化问题，所以更适合处理具体的用户查询；而 PageRank 算法在处理宽泛的用户查询时更有优势。
- HITS 算法在计算时，对于每个页面需要计算两个分值，而 PageRank 算法只需计算一个分值即可；在搜索引擎领域，更重视 HITS 算法计算出的 Authority 权值，但是在很多应用 HITS 算法的其他领域，Hub 分值也有很重要的作用。

- 从链接反作弊的角度来说，PageRank 从机制上优于 HITS 算法，而 HITS 算法更容易遭受链接作弊的影响。
- HITS 算法结构不稳定，当对扩展网页集合内链接关系做出很小改变，则对最终排名有很大影响；而 PageRank 算法相对 HITS 而言表现稳定，其根本原因在于 PageRank 计算时的远程跳转。

## 6.5 SALSA 算法

上一小节介绍了 PageRank 算法和 HITS 算法之间的异同之处，SALSA 算法的初衷希望能够结合两者的主要特点，既可以利用 HITS 算法与查询相关的特点，也可以采纳 PageRank 的随机游走模型，这是 SALSA 算法提出的背景。由此可见，SALSA 算法融合了 PageRank 和 HITS 算法的基本思想，从实际效果来说，很多实验数据表明，SALSA 的搜索效果也都优于前两个算法，是目前效果最好的链接分析算法之一。

从整体计算流程来说，可以将 SALSA 划分为两个大的阶段：首先是确定计算对象集合的阶段，这一阶段与 HITS 算法基本相同；第 2 个阶段是链接关系传播过程，在这一阶段则采纳了随机游走模型。

### 6.5.1 确定计算对象集合

PageRank 的计算对象是互联网所有网页，SALSA 算法与此不同，在本阶段，其与 HITS 算法思路大致相同，也是先得到扩展网页集合，之后将网页关系转换为方向二分图形式。

#### 扩展网页集合

SALSA 算法在接收到用户查询请求后，利用现有搜索引擎或者检索系统，获得一批与用户查询在内容上高度相关的网页，以此作为根集。并在此基础上，将与根集内网页有直接链接关系的网页纳入，形成扩展网页集合（参考图 6-15）。之后会在扩展网页集合内根据一定的链接分析方法获得最终搜索结果排名。

#### 转换为无向二分图

在获得了扩展网页集合之后，SALSA 根据集合内的网页链接关系，将网页集合转换为一个二分图。即将网页划分到两个子集合中，一个子集合是 Hub 集合，另一个子集合是 Authority 集合。划分网页节点属于哪个集合，则根据如下规则。

- 如果一个网页包含出链，这些出链指向扩展网页集合内其他节点，则这个网页可



被归入 Hub 集合。

- 如果一个网页包含扩展网页集合内其他节点指向的入链，则可被归入 Authority 集合。

由以上规则可以看出，如果某个网页同时包含入链和出链，则可以同时归入两个集合。同时，Hub 集合内网页的出链组成了二分图内的边，根据以上法则，将扩展网页集合转换为二分图。

图 6-15 和图 6-16 给出了一个示例，说明了这个转换过程。假设扩展网页集合如图 6-15 所示，由 6 个网页构成，其链接关系如图所示，同时为了便于说明，每个网页给予一个唯一编号。图 6-16 则是将图 6-15 中的网页集合转换为二分图的结果。以网页 6 为例，因为其有出链指向网页节点 3 和网页节点 5，所以可以放入 Hub 集合，也因为编号为 1、3、10 的网页节点有链接指向网页节点 6，所以也可以放入 Authority 集合中。网页节点 6 的两个出链保留，作为二分图的边，但是这里需要注意的是，在转换为二分图后，原先的有向边不再保留方向，转换为无向边，而 HITS 算法仍然保留为有向边，这点与 SALSA 略有不同。

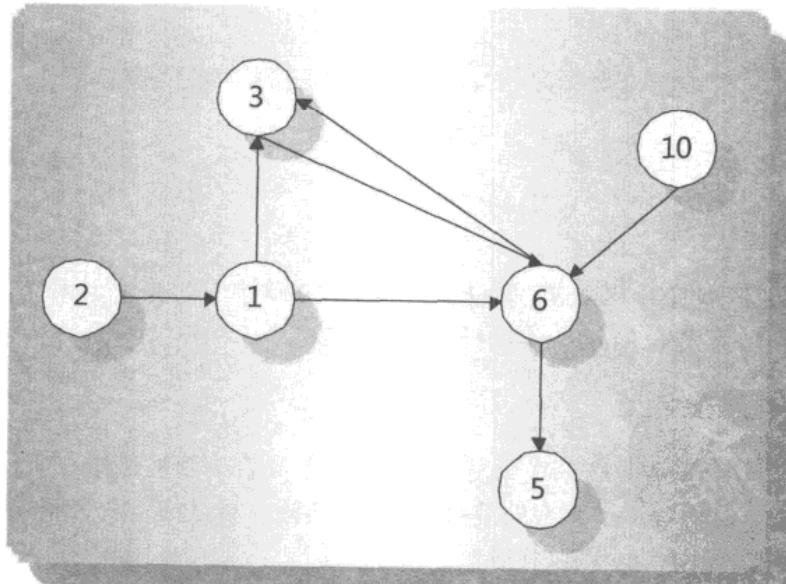


图 6-15 扩展网页集合示例

到这一步骤为止，除了 SALSA 算法将扩展网页集合转换为无向二分图，而 HITS 仍然是有向二分图外，其他步骤和流程，SALSA 算法与 HITS 算法完全相同，因此，SALSA 算法保证是与用户查询相关的链接分析算法。

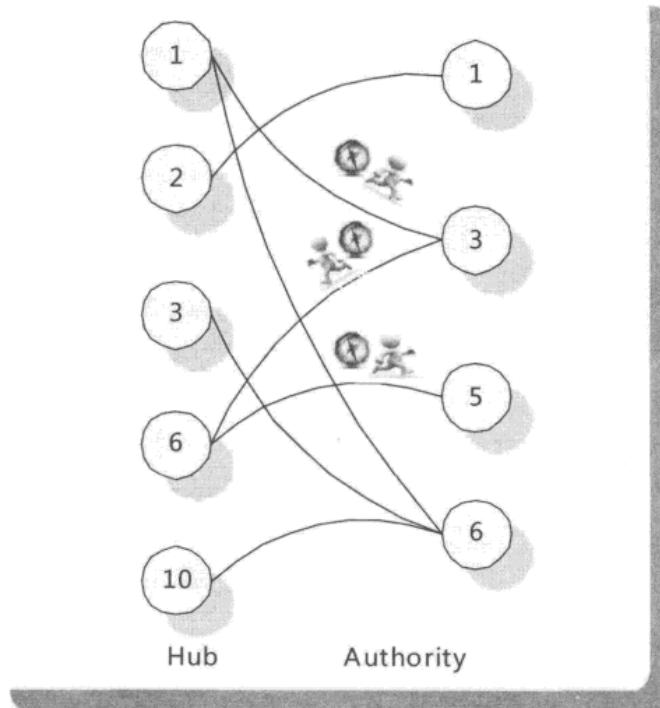


图 6-16 二分图

### 6.5.2 链接关系传播

在链接关系传播阶段，SALSA 算法放弃了 HITS 算法的 Hub 节点和 Authority 节点相互增强的假设，转而采纳 PageRank 的随机游走模型。

#### 链接关系传播概念模型

如图 6-16 所示，假设存在某个浏览器，从某个子集合中随机选择一个节点出发（为方便说明，图中所示为从 Hub 子集合的节点 1 出发，实际计算往往是从 Authority 子集合出发的），如果节点包含多条边，则以相等概率随机选择一条边，从 Hub 子集合跳到 Authority 集合内节点，图中所示为由节点 1 转移到节点 3 之后从 Authority 子集合再次跳回 Hub 子集合，即由节点 3 跳到节点 6。如此不断在两个子集之间转移，形成了 SALSA 自身的链接关系传播模式。

尽管看上去与 PageRank 的链接传播模式不同，但其实两者是一样的，关键点在于：其从某个节点跳到另外一个节点的时候，如果包含多个可供选择的链接，则以等概率随机选择一条路径，即在权值传播过程中，权值是被所有链接平均分配的。而 HITS 算法不同，

HITS 算法属于权值广播模式，即将节点本身的权值完全传播给有链接指向的节点，并不根据链接多少进行分配。

SALSA 的上述权值传播模型与 HITS 模型关注重点不同，HITS 模型关注的是 Hub 和 Authority 之间的节点相互增强关系，而 SALSA 实际上关注的是 Hub-Hub 及 Authority-Authority 之间的节点关系，而另一个子集合节点只是充当中转桥梁的作用。所以，上述权值传播模型可以转化为两个相似的子模型，即 Hub 节点关系图和 Authority 节点关系图。

### Authority 节点关系图

图 6-17 是由 6-16 的二分图转化成的 Authority 节点关系图，Hub 节点关系图与此类似，两者转化过程是相似的，我们以 Authority 节点关系图为例来看如何从二分图转化为节点关系图。

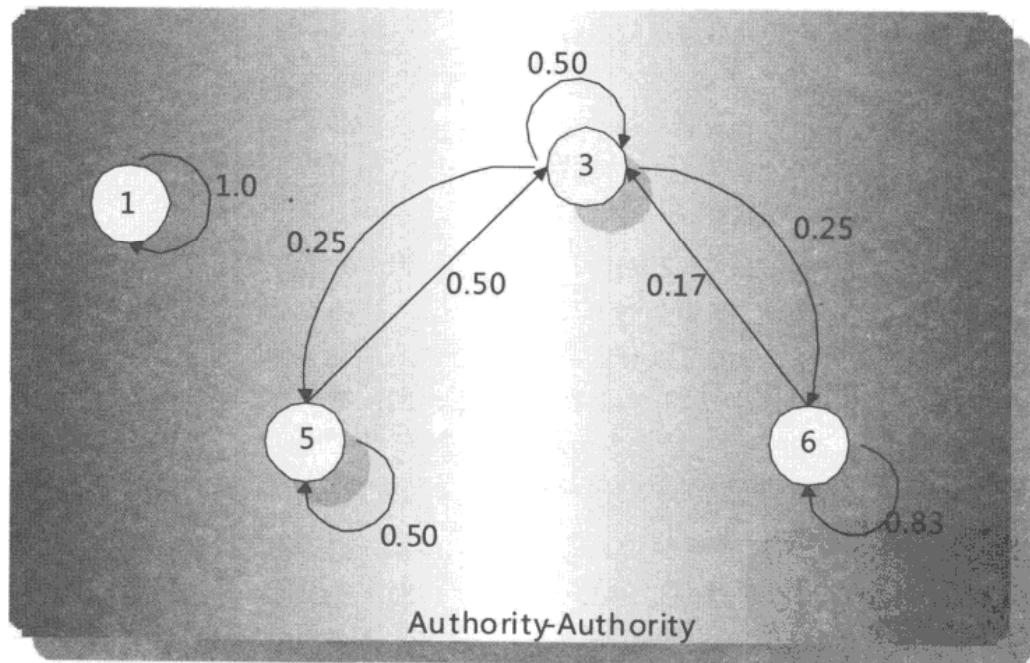


图 6-17 Authority 节点关系图

这里需要注意的是：Authority 集合内从某个节点  $i$  转移到另一个节点  $j$  的概率，与从节点  $j$  转移到节点  $i$  的概率是不同的，即非对称的，所以转换后的 Authority 节点关系图是一个有向图，以此来表示其转移概率之间的差异。

对于图 6-17 这个 Authority 节点关系图来说，图中包含的节点就是二分图中属于 Authority 子集合的节点，关键在于节点之间的边如何建立及节点之间转移概率如何计算。

### 节点关系图中边的建立

之所以在 Authority 节点图中，节点 3 有边指向节点 5，是因为在二分图中，由节点 3 通过 Hub 子集合的节点 6 中转，可以通达节点 5，所以两者之间有边建立。

这里需要注意的是：在二分图中，对于 Authority 集合内的某个节点来说，一定可以通过 Hub 子集合的节点中转后再次返回本身，所以一定包含一条指向自身的有向边。节点 1 因为只有中转节点 2 使得其返回 Authority 子集合中的自身节点，所以只有指向自身的一条边，和其他节点没有边联系，所以例子中的 Authority 节点关系图由两个连通子图构成，一个只有节点 1，另外一个连通子图由剩余几个节点构成。

### 节点之间的转移概率

至于为何 Authority 节点关系图中，节点 3 到节点 5 的转移概率为 0.25，是因为前面介绍过，SALSA 的权值传播模型遵循随机游走模型。在图 6-16 的二分图中，从节点 3 转移到节点 5 的过程中，节点 3 有两条边可做选择来跳转到 Hub 子集合，所以每条边的选择概率为  $1/2$ ，可以选择其中一条边到达节点 6，同样，从节点 6 跳回到 Authority 子集合时，节点 6 也有两条边可选，选中每条边的概率为  $1/2$ 。所以从节点 3 出发，经由节点 6 跳转到节点 5 的概率为两条边权值的乘积，即为  $1/4$ 。

对于指向自身的有向边，其权重计算过程是类似的，我们仍然以节点 3 为例，指向自身的有向边代表从 Authority 子集合中的节点 3 出发，经由 Hub 子集合的节点再次返回节点 3 的概率。从图 6-16 的二分图可以看出，完成这个过程有两条路径可走，一条是从节点 3 到节点 1 返回；另外一条是从节点 3 经由节点 6 后返回；每一条路径的概率与上面所述计算方法一样，因为两条路径各自的概率为 0.25，所以节点 3 返回自身的概率为两条路径概率之和，即为 0.5。图中其他边的转移概率计算方式也是类此。

建立好 Authority 节点关系图后，即可在图上利用随机游走模型来计算每个节点的 Authority 权值。在实际计算过程中，SALSA 将搜索结果排序问题进一步转换为求 Authority 节点矩阵的主秩问题，矩阵的主秩即为每个节点的相应 Authority 得分，按照 Authority 得分由高到低排列，即可得到最终的搜索排序结果。

### 6.5.3 Authority 权值计算

经过数学推导，可以得出 SALSA 与求矩阵主秩等价的 Authority 权值计算公式。图 6-18 中的示意图表明了 SALSA 算法中某个网页节点的 Authority 权值是如何计算的。如图右上角公式所示，决定某个网页  $i$  的 Authority 权值涉及 4 个因子。

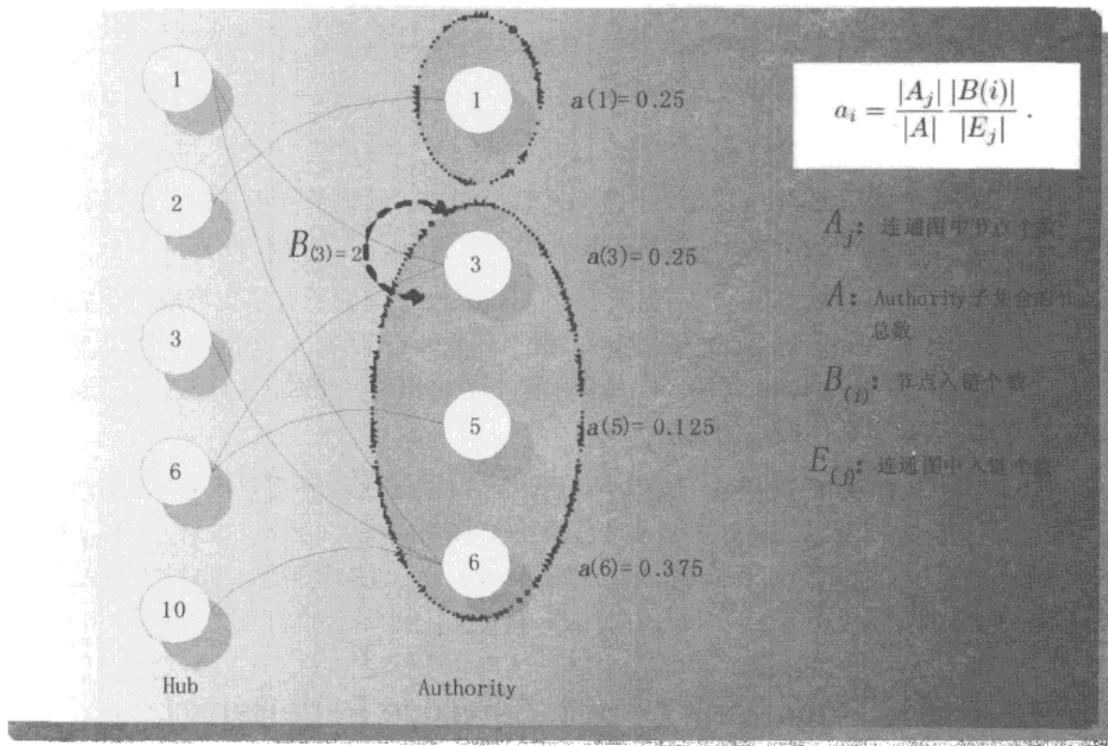


图 6-18 SALSA 节点权值计算公式

- Authority 子集合中包含的节点总数 $|A|$ 。其实这个因子对于 Authority 集合中任意节点来说都是相同的，所以对于最终的根据节点 Authority 权值进行的排序没有影响，只是起到保证权值得分在 0 到 1 之间，能够以概率形式表示权值的作用。
- 网页  $i$  所在连通图中包含的节点个数 $|A_j|$ 。网页所在的连通图包含的节点个数越多，则网页的 Authority 权值越大。
- 网页  $i$  所在连通图中包含的入链总数 $|E_j|$ 。网页所在的连通图包含的入链总数越少，则网页的 Authority 权值越大。
- 网页  $i$  的入链个数 $|B_i|$ 。节点入链越多，则 Authority 权值越大，这个因子是唯一一个和节点本身属性相关的。由此可见，SALSA 权值计算和节点入链个数成正比。

之前图 6-17 的“Authority 节点关系图”由两个连通子图组成，一个由唯一的节点 1 构成，另外一个由节点 3、5、6 3 个节点构成，两个连通子图在图 6-18 中也被分别圈出。

我们以节点 3 为例，看其对应的 4 个计算因素取值。

- Authority 子集共包括 4 个节点。

- 节点 3 所在连通图包含 3 个节点。
- 节点 3 所在连通图共有 6 个入链。
- 节点 3 的入链个数为 2。

所以，节点 3 的 Authority 权值为： $(3/4) \times (2/6)=0.25$ 。其他节点权值的计算过程与此类似。SALSA 根据节点的 Authority 权值由高到低排序输出，即为搜索结果。

由上述权值计算公式可以推出：如果整个 Authority 子集合所有节点形成一个完整的连通图，那么在计算 Authority 权值的过程中，对于任意两个节点，4 个因子中除了节点入链个数外，其他 3 个因子总是相同的，即只有入链个数起作用，此时，SALSA 算法退化为根据节点入链个数决定排序顺序的算法。

从 SALSA 计算 Authority 得分过程中可看出，SALSA 算法不需要像 HITS 算法一样进行不断的迭代计算，所以从计算效率角度看要快于 HITS 算法。另外，SALSA 算法解决了 HITS 算法的计算结果主题漂移问题，所以搜索质量也优于 HITS 算法。SALSA 算法是目前效果最好的链接算法之一。

## 6.6 主题敏感 PageRank ( Topic Sensitive PageRank )

主题敏感 PageRank 是 PageRank 算法的改进版本，该算法已被 Google 使用在个性化搜索服务中。

### 6.6.1 主题敏感 PageRank 与 PageRank 的差异

PageRank 算法基本遵循前面章节提到的随机游走模型，即用户在浏览某个网页时，如果希望跳转到其他页面，则随机选择本网页包含的某个链接，进入另一个页面。主题敏感 PageRank 则对该概念模型做出改进，引入了更符合现实的假设。一般来说用户会对某些领域感兴趣，同时当浏览某个页面时，这个页面也是与某个主题相关的（比如体育报道或者娱乐新闻），所以当用户看完当前页面，希望跳转时，更倾向于点击和当前页面主题类似的链接，即主题敏感 PageRank 是将用户兴趣、页面主题及链接所指向网页与当前网页主题的相似程度综合考虑而建立的模型。很明显，这更符合真实用户的浏览过程。

PageRank 是全局性的网页重要性衡量标准，每个网页会根据链接情况，被赋予一个唯一的 PageRank 分值。主题敏感 PageRank 在此点有所不同，该算法引入了 16 种主题类型，对于某个网页来说，对应某个主题类型都有相应的 PageRank 分值，即每个网页会被赋予 16 个主题相关 PageRank 分值。



在接收到用户查询后，两个算法在处理方式上也有较大差异。PageRank 算法与查询无关，只能作为相似度计算的一个计算因子体现作用，无法独立使用。而主题敏感 PageRank 是查询相关的，可单独作为相似度计算公式使用。而且，在接收到用户查询后，主题敏感 PageRank 还需要利用分类器，计算该查询隶属于事先定义好的 16 个主题的隶属度，并在相似度计算时的排序公式中利用此信息。

### 6.6.2 主题敏感 PageRank 计算流程

主题敏感 PageRank 计算主要由两个步骤构成，第 1 步是离线的分类主题 PageRank 数值计算；第 2 步是在线利用算好的主题 PageRank 分值，来评估网页和用户查询的相似度，以按照相似度排序提供给用户搜索结果。下面以具体示例来了解主题敏感 PageRank 的计算流程。

#### 分类主题 PageRank 计算

主题敏感 PageRank 参考 ODP 网站 ([www.dmoz.org](http://www.dmoz.org))，定义了 16 个大的主题类别，包括体育、商业、科技等。ODP (Open Directory Project) 是人工整理的多层级网页分类导航站点（参见图 6-19），在顶级的 16 个大分类下还有更细致的小粒度分类结构，在最底层目录下，人工收集了符合该目录主题的精选高质量网页地址，以供互联网用户导航寻址。主题敏感 PageRank 采用了 ODP 最高级别的 16 个分类类别作为事先定义的主题类型。

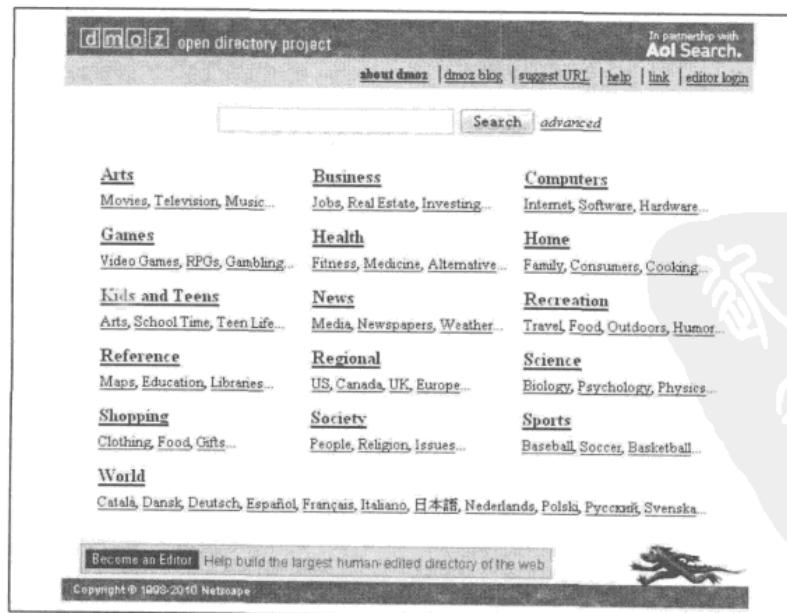


图 6-19 ODP 首页

主题敏感 PageRank 对 16 个类别的主题，依次计算该类别的 PageRank 分值，图 6-20 显示了其计算流程和基本思路，为了简化说明，示意图只表现出了 3 个分类类别。在计算某个类别的 PageRank 分值时，将所有网页划分为两个集合，一个集合是 ODP 对应分类主题下所包括的所有网页，即人工精选的高质量网页，可以称之为集合  $S$ ，剩下的网页放入另一个集合内，可称之为集合  $T$ 。在计算 PageRank 时，由于集合  $S$  内的网页能够很好地表征分类主题，所以赋予较大的跳转概率值。通过这种设定，集合  $S$  内的网页根据链接关系向集合  $T$  中网页传递权值，因为直接有链接指向的往往主题类似，这样就将与该分类主题内容相似的网页赋予较高的 PageRank 值，而无关的网页则赋予较低权重的 PageRank 分值，以此方式达到对网页所包含主题的判断。

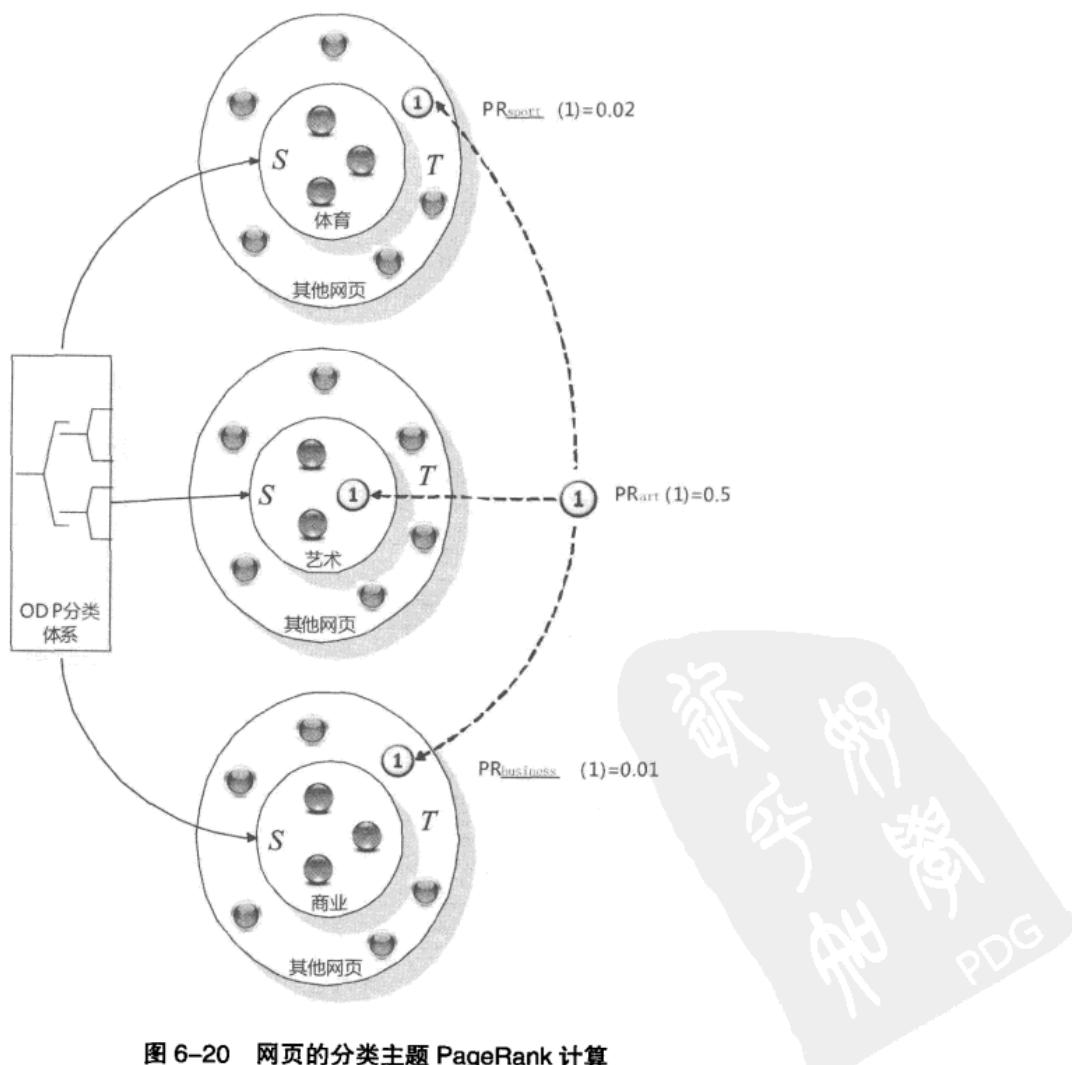


图 6-20 网页的分类主题 PageRank 计算



回到图 6-20，假设有个编号为 1 的网页，其被列到 ODP 目录中的艺术类别中，在对艺术类别进行 PageRank 计算时，1 号网页在集合  $S$  内，计算结束后，该网页获得的 PageRank 分值为 0.5。当计算体育和商业类别的主题 PageRank 分值时，1 号网页在集合  $T$  中，获得了相应的集合  $S$  中网页传递的权值，分别为 0.02 和 0.01。在所有类别计算结束后，1 号网页获得了 3 个不同主题对应的 PageRank 分值，组成一个主题 PageRank 向量。通过类似的方式，互联网内任意网页也可以获得相应的主题相关 PageRank 向量。通过以上过程可以看出，主题相关的 PageRank 分值向量其实代表了某个网页所讲述内容所属类别的概率。

**注意：**在上述计算主题 PageRank 过程中，从集合  $S$  和集合  $T$  的划分及其权值传播方式中可以看出，该步骤计算过程也符合子集传播模型。但是由于本算法主框架及其出发点都是为了改进 PageRank，所以将其归入随机游走模型的衍生算法类别中。

### 在线相似度计算

图 6-21 给出了主题敏感 PageRank 在线计算用户查询与网页相似度的示意图。假设用户输入了查询请求“乔丹”，搜索系统首先利用用户查询分类器对查询进行分类，计算用户查询隶属于定义好的各个类别的概率分别是多少，在我们给出的例子里，“乔丹”隶属于体育类别的概率为 0.6，娱乐类别的概率为 0.1，商业类别的概率为 0.3。

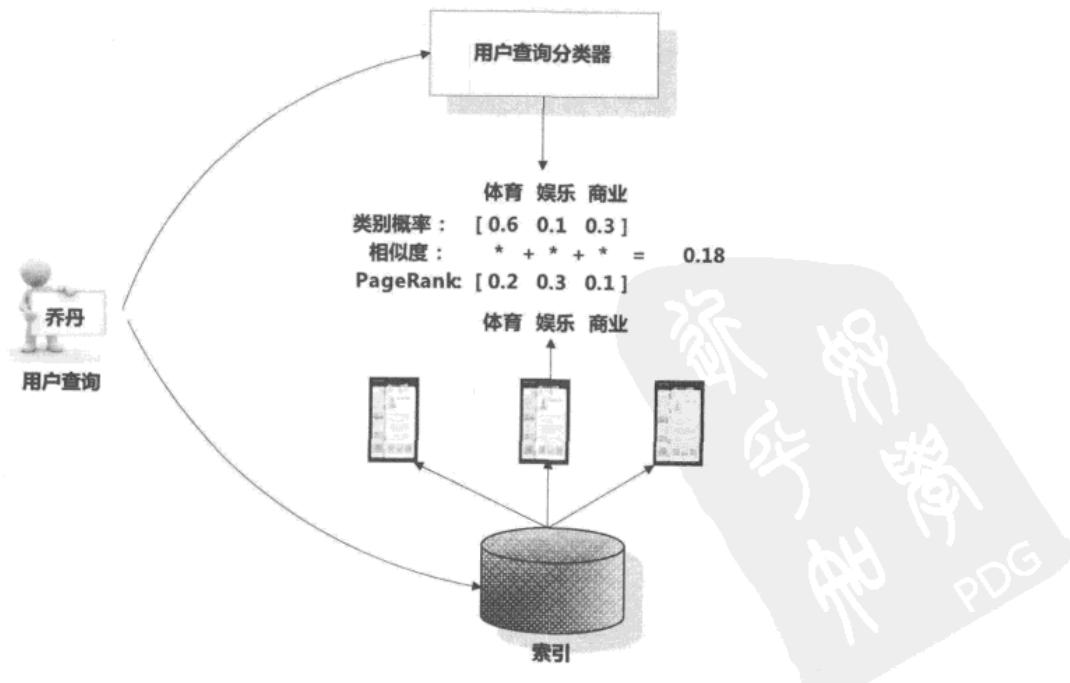


图 6-21 在线相似度计算

在进行上述用户查询分类计算的同时，搜索系统读取索引，找出包含了用户查询“乔丹”的所有网页，并获得上一步骤离线计算好的各个分类主题的 PageRank 值，在图 6-21 的例子里，假设某个网页 A 的各个主题 PageRank 值分别为体育 0.2、娱乐 0.3 及商业 0.1。得到用户查询的类别向量和某个网页的主题 PageRank 向量后，即可计算这个网页和查询的相似度。通过计算两个向量的乘积就可以得出两者之间的相关性。在如图 6-21 所示的例子里，网页 A 和用户查询“乔丹”的相似度为：

$$\text{Sim}(\text{“乔丹”}, A) = 0.6 \times 0.2 + 0.1 \times 0.3 + 0.3 \times 0.1 = 0.18$$

对包含“乔丹”这个关键词的网页，都根据以上方法计算，得出其与用户查询的相似度后，就可以按照相似度由高到低排序输出，作为本次搜索的搜索结果返回给用户。

### 6.6.3 利用主题敏感 PageRank 构造个性化搜索

以上内容介绍的是主题敏感 PageRank 的基本思想和计算流程，从其内在机制来说，这个算法非常适合作为个性化搜索的技术方案。

在如图 6-21 所示的例子里，计算相似度使用的只有用户当前输入的查询词“乔丹”，如果能够对此进行扩展，即不仅使用当前查询词，也考虑利用用户过去的搜索记录等个性化信息。比如用户之前搜索过“耐克”，则可以推断用户输入“乔丹”是想购买运动服饰，而如果之前搜索过“姚明”，则很可能用户希望获得体育方面的信息。通过这种方式，可以将用户的个性化信息和当前查询相融合来构造搜索系统，以此达到个性化搜索的目的，更精准地提供搜索服务。

## 6.7 Hilltop 算法

Hilltop 算法是 Toronto 大学研发的链接分析算法，在 2003 年被 Google 公司收购，而 Google 在之后的排序算法大改版中引入了 Hilltop 算法。

Hilltop 算法融合了 HITS 和 PageRank 两个算法的基本思想。一方面，Hilltop 是与用户查询请求相关的链接分析算法，吸收了 HITS 算法根据用户查询获得高质量相关网页子集的思想，符合子集传播模型，是该模型的一个具体实例；同时，在权值传播过程中，Hilltop 算法也采纳了 PageRank 的基本指导思想，即通过页面入链的数量和质量来确定搜索结果的排序权重。



### 6.7.1 Hilltop 算法的一些基本定义

非从属组织页面（Non-affiliated Pages）是 Hilltop 算法的一个很重要的定义。要了解什么是非从属组织页面，先要搞明白什么是从属组织网站，所谓的从属组织网站，即不同的网站属于同一机构或者其拥有者有密切关联。具体而言，满足如下任意一条判断规则的网站会被认为是从属网站。

- **条件 1：**主机 IP 地址的前 3 个子网段相同，比如：IP 地址分别为 159.226.138.127 和 159.226.138.234 的两个网站会被认为是从属网站。
- **条件 2：**如果网站域名中的主域名相同，比如 www.ibm.com 和 www.ibm.com.cn 会被认为是从属组织网站。

非从属组织页面的含义是：如果两个页面不属于从属网站，则为非从属组织页面。图 6-22 是相关示意图，从图中可以看出，页面 2 和页面 3 同属于 IBM 的网页，所以是从属组织页面，而页面 1 和页面 5、页面 3 和页面 6 都是非从属组织页面。由此也可看出，非从属组织页面代表的是页面的一种关系，单个页面是无所谓从属或者非从属组织页面的。

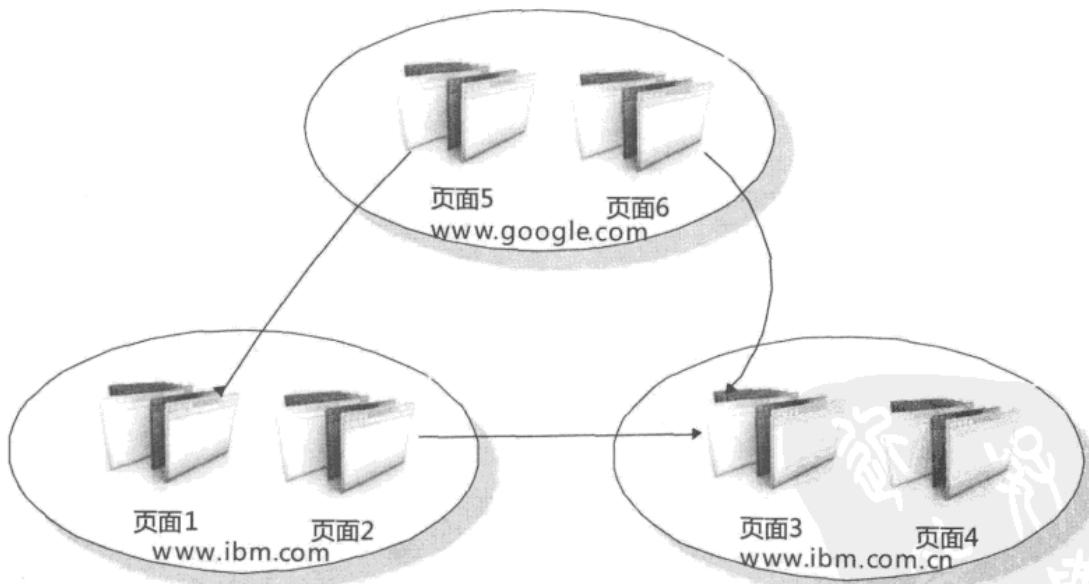


图 6-22 从属组织页面与非从属组织页面

专家页面（Export Sources）是 Hilltop 算法的另外一个重要定义。所谓专家页面，即与某个主题相关的高质量页面，同时需要满足以下要求：这些页面的链接所指向的页面相互之间都是非从属组织页面，且这些被指向的页面大多数是与专家页面主题相近的。

Hilltop 算法将互联网页面划分为两类子集合，最重要的子集合是由专家页面构成的互联网页面子集，不在这个子集合里的剩下的互联网页面作为另外一个集合，这个集合称做目标页面集合（Target Web Servers）。

### 6.7.2 Hilltop 算法

图 6-23 是 Hilltop 算法的整体流程示意。首先从海量的互联网网页中通过一定规则筛选出专家页面子集合，并单独为这个页面集合建立索引。Hilltop 在接收到用户发出的某个查询请求时，首先根据用户查询的主题，从专家页面子集合中找出部分相关性最强的专家页面，并对每个专家页面计算相关性得分，然后根据目标页面和这些专家页面的链接关系来对目标页面进行排序。基本思路遵循 PageRank 算法的链接数量假设和质量原则，将专家页面的得分通过链接关系传递给目标页面，并以此分数作为目标页面与用户查询相关性的排序得分。最后系统整合相关专家页面和得分较高的目标页面作为搜索结果返回给用户。

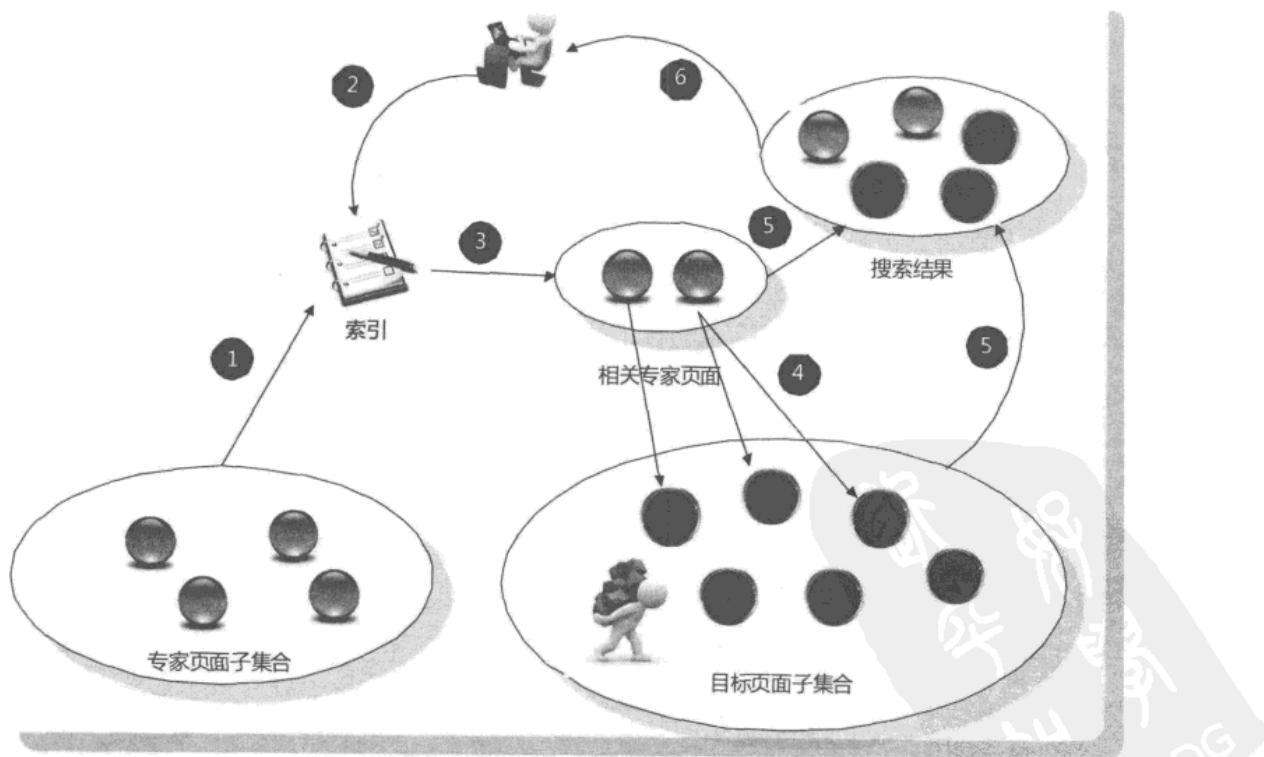


图 6-23 Hilltop 算法流程

若在上述过程中，Hilltop 算法无法得到一个足够大的专家页面集合，则返回搜索结果



为空。由此可以看出，Hilltop 算法更注重搜索结果的精度和准确性，不太考虑搜索结果是否足够多或者对大多数用户查询是否都有相应的搜索结果，所以很多用户发出的查询的搜索结果为空。这意味着 Hilltop 算法可以与某个排序算法相结合，以提高排序准确性，但并不适合作为一个独立的网页排序算法来使用。

从上述整体流程描述可看出，Hilltop 算法主要包含两个步骤：专家页面搜索及目标页面排序。

### 步骤一：专家页面搜索

Hilltop 算法从 1 亿 4 千万网页中，通过计算筛选出 250 万规模的互联网页面作为专家页面集合。专家页面的选择标准相对宽松，同时满足以下两个条件的页面即可进入专家页面集合。

- **条件 1：**页面至少包含  $k$  个出链，这里的数量  $k$  可人为指定。
- **条件 2：** $k$  个出链指向的所有页面相互之间的关系都符合非从属组织页面的要求。

当然，在此基础上，可以设定更严格的筛选条件，比如要求这些专家页面所包含链接指向的页面中，大部分涉及的主题和专家页面的主题必须是一致或近似的。

根据以上条件筛选出专家页面后，即可对专家页面单独建索引，在此过程中，索引系统只对页面中的关键片段（Key Phrase）进行索引。所谓关键片段，在 Hilltop 算法里包含了网页的 3 类信息：网页标题、H1 标签内文字和 URL 锚文字。

网页的关键片段可以支配（Qualify）某个区域内包含的所有链接，支配关系代表了一种管辖范围，不同的关键片段支配链接的区域范围不同，具体而言，页面标题可以支配页面内所有出现的链接，H1 标签可以支配包围在<H1>和</H1>内的所有链接，而 URL 锚文字只能支配本身唯一的链接。图 6-24 给出了关键片段对链接支配关系的示意图，在以“奥巴马访问中国”为标题的网页页面中，标题支配了所有这个页面出现的链接，而 H1 标签的管辖范围仅限于标签范围内出现的两个链接，对于锚文字“中国领导人”来说，其唯一能够支配的就是本身的这个链接。之所以定义这种支配关系，对于第 2 阶段将专家页面的分值传递到目标页面时候会起作用。

系统接收到用户查询 Q，假设用户查询包含了多个单词，Hilltop 如何对专家页面进行打分呢？对专家页面进行打分主要参考以下 3 类信息。

- 关键片段包含了多少查询词，包含的查询词越多，则分值越高，如果不包含任何查询词，则该关键片段不计分。
- 关键片段本身的类型信息，网页标题权值最高，H1 标签次之，再次是链接锚文字。

- 用户查询和关键片段的失配率，即关键片段中不属于查询词的单词个数占关键片段总单词个数，这个值越小越好，越大则得分衰减越多。

Hilltop 综合考虑以上 3 类因素，拟合出打分函数来对专家页面是否与用户查询相关进行打分，选出相关性分值足够高的专家页面，以进行下一步骤操作，即对目标页面进行相关性计算。

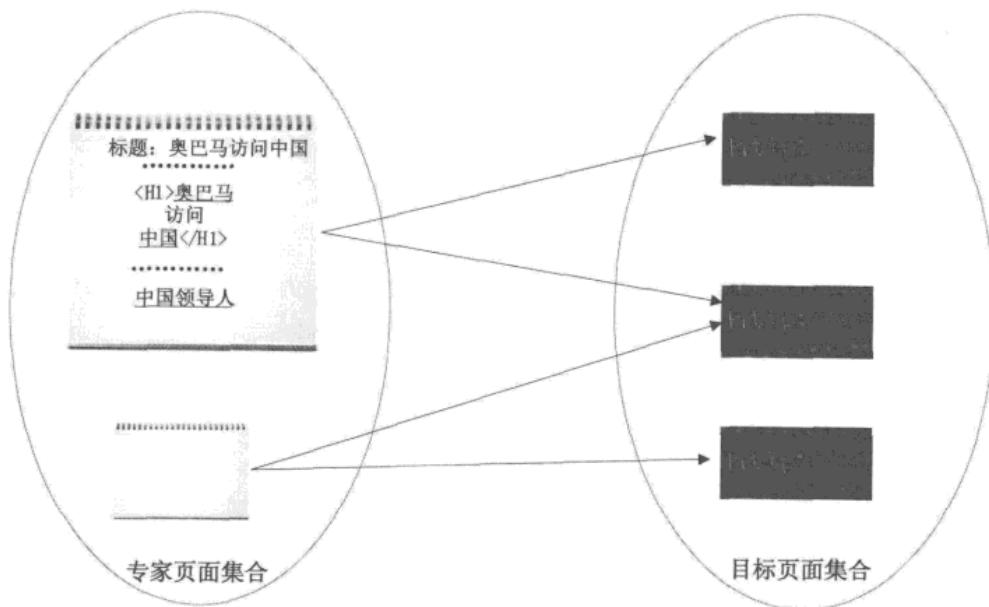


图 6-24 关键片段链接支配关系

## 步骤二：目标页面排序

Hilltop 算法包含一个基本假设，即认为一个目标页面如果是满足用户查询的高质量搜索结果，其充分必要条件是该目标页面有高质量专家页面链接指向。然而，这个假设并不总是成立，比如有的专家页面的链接所指向的目标页面可能与用户查询并非密切相关。所以，Hilltop 算法在这个阶段需要对专家页面的出链仔细进行甄别，以保证选出那些和查询密切相关的目标页面。

Hilltop 在本阶段是基于专家页面和目标页面之间的链接关系来进行的，在此基础上，将专家页面的得分传递给有链接关系的目标页面。传递分值之前，首先需要对链接关系进行整理，能够获得专家页面分值的目标页面需要满足以下两点要求：

- **条件 1：**至少需要两个专家页面有链接指向目标页面，而且这两个专家页面不能是从属组织页面，即不能来自同一网站或相关网站。如果是从属组织页面，则只



能保留一个链接，抛弃权值低的那个链接。

- **条件 2：**专家页面和所指向的目标页面也需要符合一定要求，即这两个页面也不能是从属组织页面。

在步骤一，给定用户查询，Hilltop 算法已经获得相关的专家页面及其与查询的相关度得分，在此基础上，如何对目标页面的相关性打分？上面列出的条件 1 指出，能够获得传递分值的目标页面一定有多个专家页面链接指向，所以目标页面所获得的总传播分值是每个有链接指向的专家页面所传递分值之和。而计算其中某个专家页面传递给目标页面权值的时候是这么计算的：

- a. 找到专家页面中那些能够支配目标页面的关键片段集合  $S$ 。
- b. 统计  $S$  中包含用户查询词的关键片段个数  $T$ ， $T$  越大传递的权值越大。
- c. 专家页面传递给目标页面的分值为： $E \times T$ ， $E$  为专家页面本身在第一阶段计算得到的相关得分， $T$  为 b 步骤计算的分值。

我们以图 6-25 的具体例子来说明。假设专家页面集合内存在一个网页 P，其标题为“奥巴马访问中国”，网页内容由一段<H1>标签文字和另一个单独的链接锚文字组成。该页面包含 3 个出链，其中两个指向目标页面集合中的网页 [www.china.org](http://www.china.org)，另外一个指向网页 [www.obama.org](http://www.obama.org)。出链对应的锚文字分别为“奥巴马”、“中国”和“中国领导人”。

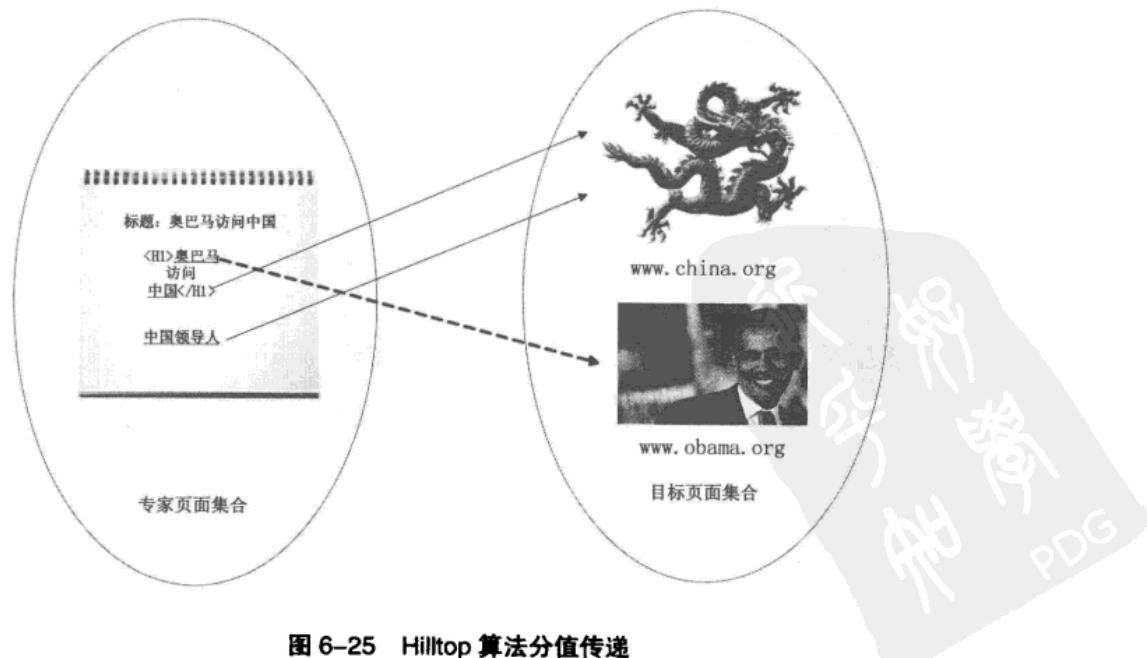


图 6-25 Hilltop 算法分值传递

从图示的链接关系可以看出，网页 P 中能够支配 [www.china.org](http://www.china.org) 这个目标页面的关键片段集合包括：{中国领导人，中国，**<H1>奥巴马访问中国</H1>**，标题：奥巴马访问中国}，而能够支配 [www.obamba.org](http://www.obamba.org) 目标页面的关键片段集合包括：{奥巴马，**<H1>奥巴马访问中国</H1>**，标题：奥巴马访问中国}。

接下来我们分析专家页面 P 在接收到查询时，是怎样将分值传递给与其有链接关系的目标页面的。假设系统接收到的查询请求为“奥巴马”，在接收到查询后，系统首先根据上述章节所述，找出专家页面并给予分值，而网页 P 作为专家页面中的一个页面，并获得了相应的分值 S，我们重点关注分值传播步骤。

对于查询“奥巴马”来说，网页 P 中包含这个查询词的关键片段集合为：{奥巴马，**<H1>奥巴马访问中国</H1>**，标题：奥巴马访问中国}，如上所述，这 3 个关键片段都能够支配 [www.obamba.org](http://www.obamba.org) 页面，所以网页 P 传递给 [www.obamba.org](http://www.obamba.org) 的分值为  $S \times 3$ 。而对于目标页面 [www.china.org](http://www.china.org) 来说，这 3 个关键片段中只有 {**<H1>奥巴马访问中国</H1>**，标题：奥巴马访问中国} 这两个能够支配目标页面，所以网页 P 传递给 [www.china.org](http://www.china.org) 的分值为  $S \times 2$ 。

对于包含多个查询词的用户请求，则每个查询词单独如上计算，将多个查询词的传递分值累加即可。

Hilltop 算法存在与 HITS 算法类似的计算效率问题，因为根据查询主题从专家页面集合中选取主题相关的页面子集也是在线运行的，这与前面提到的 HITS 算法一样会影响查询响应时间。随着专家页面集合的增大，算法的可扩展性存在不足之处。

## 6.8 其他改进算法

上述章节详述了 5 个非常重要的链接分析算法，在此基础上，学术界提出了很多改进方法，本节简述其中几个相对重要方法的基本思路和本质思想。

### 6.8.1 智能游走模型 (Intelligent Surfer Model)

智能游走模型是对 PageRank 算法的改进，算法提出者认为 PageRank 所遵循的随机游走模型不符合真实情况，浏览者在浏览一个页面并选择下一步点击对象时，并非随机的，而是会对链接进行甄别，如果某个链接的网页内容能够表达浏览者向搜索引擎发出的查询词主题，则用户选择点击这个链接的概率会更大。智能游走模型即是对这种情况建立模型，会判断网页包含的链接所指向的网页内容和用户查询的相关性，以此来改善链接分析效果。

智能游走模型考虑到了网页内容和用户输入查询词的相关性，所以是查询相关的链接分析算法。但是需要在线计算某个查询和网页内容的相关性，所以计算速度较慢是其重要

缺陷。

### 6.8.2 偏置游走模型 ( Biased Surfer Model )

偏置游走模型也是针对 PageRank 的随机游走模型的改进，其基本思路和智能游走模型很接近，不同点在于：智能游走模型考虑的是网页内容和用户查询的相关性，而偏置游走模型考虑的是链接指向的网页内容和当前浏览网页内容之间的相似性，即认为如果链接指向网页内容和当前浏览网页内容越相似，则用户越可能点击这个链接。

偏置游走模型由于没有考虑查询，而只需要计算网页内容之间的相似性，可以离线进行计算，并在线使用，所以计算速度要优于智能游走模型。

### 6.8.3 PHITS 算法 ( Probability Analogy of HITS )

PHITS 算法是对 HITS 算法的直接改进。HITS 算法在通过链接进行权值传递时，会将 Hub 或者 Authority 节点的权值全部传递给所有相连链接，而 PHITS 算法认为不同链接其传递权值的能力应该是不同的，这是其基本思想。基于此，PHITS 需要计算两个页面 S 和 T 之间链接的连接强度。

PHITS 算法在页面 S 和页面 T 之间引入主题隐变量，而两个页面之间链接的强度取决于两个页面和这个隐变量的主题耦合关系，所以可以将这个算法看做 PLSI 算法的链接分析版本。虽然看上去复杂，但是 PHITS 算法的本质思想和偏置游走模型是一样的，即考虑的其实是页面 S 和页面 T 之间的内容相似性，内容越相似其链接强度越强，传递权值的能力越强。PHITS 算法和直接计算页面内容相似性的区别，仅仅是加入一个隐藏主题层，通过网页和隐藏层的关系来计算相似性。

### 6.8.4 BFS 算法 ( Backward Forward Step )

BFS 算法作为 SALSA 算法的改进提出，而其本质既是对 SALSA 算法的扩展，也是对 HITS 算法的限制。SALSA 算法在计算网页的 Authority 分值时，仅仅考虑了通过 Hub 节点能够直接建立联系的网页之间的关系，而 HITS 算法考虑的是链接图的整个结构，任意两个网页 S 和 T，如果网页 S 能够通过中间节点链接到网页 T，则网页 S 就对网页 T 有影响，而影响的大小取决于从 S 到 T 的不同路径走法的个数，与 S 到 T 的距离远近没有关系。

BFS 则是对两者的一个折中考虑，解除了 SALSA 算法只允许直接相邻网页才能有影响的限制，只要网页 S 可以通达 T，即可对网页 T 施加影响，但是又对 HITS 的网页影响方式做了限制，如果网页 S 距离网页 T 距离越远，那么网页 S 的影响就随着距离增大而呈

指数衰减，即距离越远，影响越小。通过这种折中，来实现对 SALSA 算法的扩充和对 HITS 算法的限制。

## 本章提要

- 链接分析在搜索引擎搜索结果排序中起到非常重要的作用。
- 绝大部分链接分析算法建立在随机游走模型和子集传播模型基础之上。
- PageRank 和 HITS 算法是最重要且基础的两种链接分析算法，很多链接分析算法是对这两种方法的改进。
- SALSA 算法是目前效果最好的链接分析算法之一，其融合了 HITS 算法与查询相关的特点，以及 PageRank 算法的随机游走模型。
- 主题敏感 PageRank 是对 PageRank 算法的改进，可以应用于个性化搜索中。
- Hilltop 算法除了可以用来改善搜索系统的精确性，还可以用来当做网页反作弊的技术手段。

## 本章参考文献

- [1] Page, L., Brin, S., Motwani, R. and Winograd, T. (1998). The PageRank citation ranking: bringing order to the Web. Stanford Digital Library Technologies.
- [2] Borodin, A., Roberts, G.O., Rosenthal, J.S., and Tsaparas, P. (2001). Finding authorities and hubs from link structures on the World Wide Web, WWW10 Proceedings, Hongkong.
- [3] Lempel, R. and Moran S.(2000). The stochastic approach for link-structure analysis (SALSA) and the TKC effect. In the 9<sup>th</sup> International World Wide Web Conference.
- [4] Kleinberg, J. (1998). Authoritative sources in a hyperlinked environment. In Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms. 668–677.
- [5] Bharat, K. and Mihaila, G. A. (2001). When experts agree: Using non-affiliated experts to rank popular topics. In Proceedings of the 10th International World Wide Web Conference.
- [6] Desikan, P., Nishith, P., Srivastava, J. and Kumar, V. (2005). “Incremental PageRank Computation on evolving graphs”, Poster Paper at Fourteenth International World Wide Web Conference.

Wide Web Conference on May 10-14, 2005, in Chiba, Japan.

- [7] Taher, H. Haveliwala.(2002). Topic-Sensitive PageRank, in Proceedings of the Eleventh International World Wide WEB Conference.
- [8] Xing, W. and Ghorbani, A. (2004). "Weighted PageRank algorithm". Proc. of the 2rd Annual Conference on Communication Networks & Services Research, Fredericton, Canada, pp. 305-314.
- [9] Richardson, M. and Domingos, P. (2002). The intelligent surfer: Probabilistic combination of link and content information in PageRank. In Advances in Neural Information Processing Systems (NIPS) 14.



## 第 7 章 云存储与云计算

“秋水时至，百川灌河；泾流之大，两涘渚崖之间不辨牛马。于是焉河伯欣然自喜，以天下之美为尽在己。顺流而东行，至于北海，东面而视，不见水端。于是焉河伯始旋其面目，望洋向若而叹曰：野语有之曰，‘闻道百，以为莫已若’者，我之谓也。且夫我尝闻少仲尼之闻而轻伯夷之义者，始吾弗信；今我睹子之难穷也，吾非至于子之门则殆矣，吾长见笑于大方之家。”

庄子·《秋水》

对于 Google 这种大型搜索引擎服务提供商来说，需要处理的数据已超百亿，而且绝大部分数据是网页这种无结构或者半结构化的数据。如何构建存储平台和计算平台，使得存储和管理这些海量数据简单化，就成为了重要问题。云存储与云计算平台就是为了解决这个问题而提出的技术方案。目前来说，能够提供一套高效的云存储与云计算平台，已成为搜索引擎公司的核心竞争力。

云存储与云计算是目前非常流行的技术概念，一些国外的大型互联网公司在此走在前列，最典型的是 Google 公司和亚马逊公司，这两个公司是首先应用并大力倡导云计算的龙头企业。尽管 2011 年 4 月亚马逊公司的云计算服务出现了大规模的故障，导致很多构建在其上的互联网公司停止服务，这为云计算领域的发展蒙上了一层阴影，但是云计算作为一种大的技术和商业趋势，一定会继续迅猛前行。

本章以 Google 提供的一整套云存储与云计算技术框架为主，来介绍相关的核心技术，不仅对 GFS/BigTable/MapReduce 三驾马车给出了详细说明，还对 Google 刚部署不久的咖啡因系统及更进一步的 MegaStore 存储系统等新披露的技术进行了讲解。除此之外，还对另外一些有特色的云存储平台进行了介绍，比如亚马逊公司的 Dynamo，雅虎公司的 PNUTS 及 Facebook 公司的 HayStack 存储平台，这些技术各有特点，是非常有代表性的云存储与云计算平台，基本能够概括云计算领域的技术现状。但是有一点需要读者注意：本章所说



的云计算是狭义的，不包括虚拟化等云计算关键技术，总体而言，本章以云存储作为讲解重点。

## 7.1 云存储与云计算概述

本节从云存储与云计算的基本假设、理论基础、数据模型及基本问题等方面开始，使得读者对这个计算领域有个宏观的整体性认识，同时本节对 Google 的整套技术框架做出概述性的说明。

### 7.1.1 基本假设

云存储与云计算首先由大型互联网公司提倡并推行，是由于互联网数据的爆炸性增长导致的。互联网服务和传统软件行业提供的服务有很大差异，也有其特点，所以云存储与云计算平台的设计和构建都遵循以下一些基本假设与要求。

#### 1. 由大量廉价 PC 构成

传统的海量数据存储与管理，往往会选择商用数据库，但是现代云计算和云存储平台与此思路不同，往往使用大量廉价 PC 作为基本的存储节点。一方面这样可以节省企业数据存储和计算的成本；另外一方面，数据库的扩展性不能满足拥有海量数据的互联网企业要求，这是云存储与云计算平台兴起的一个基本背景。

#### 2. 机器节点出现故障是常态

由于是大量普通 PC 构成的分布式系统，众所周知，PC 的故障率很高，尤其是这种众多 PC 协同提供服务的模式，随时都有可能某台机器出现故障，所以云存储与云计算技术方案在设计之初，就应该将这种经常性的机器故障作为一个设计考虑因素，在这个约束前提下提供可靠服务，必须要考虑到数据的可用性与安全性。

#### 3. 水平增量扩展

互联网服务随时都要响应用户的请求，随着数据量的不断增大，需要靠增加机器来承载更多数据，即使这样，也不可能中断服务来进行，所以必须在用户无察觉的情况下实现增量扩展。而对于云存储与云计算平台，应该做到简单增加机器就可以自动实现增量扩展，这往往是通过数据的水平分割实现的。

#### 4. 弱数据一致性

云存储与云计算平台有其优势：高可用性、易扩展性、容错性能好。但这种优势并非

是无代价的，其付出的代价就是数据的弱数据一致性。数据库提供了数据的强数据一致性视角，并在此基础上提供了事务支持，而云计算平台则放松这种强数据一致性，这主要是因为很多互联网服务并不要求这种强数据一致性。

## 5. 读多写少型服务

互联网服务有个特点，即读多写少，也就是说大量的请求是读取数据，少量的服务是对数据做出更改。所以在设计云存储与云计算平台时，应该考虑到这个特点，做出有针对性的优化，保证系统效率。

### 7.1.2 理论基础

CAP、BASE、ACID 及最终一致性等这些基础原理对于深入理解云存储与云计算技术有重要指导作用。

CAP 是对 Consistency/Availability/Partition Tolerance 的一种简称，其分别代表：一致性、可用性和分区容忍性（参考图 7-1）。研究者已经证明，对于一个数据系统来说，CAP 3 个要素不可兼得，同一个系统至多只能实现其中的两个，而必须放宽第 3 个要素来保证其他两个要素被满足。对于分布式系统来说，分区容忍性是天然具备的要求，所以在设计具体技术方案时，必须在一致性和可用性方面做出取舍，要么选择强数据一致性减弱服务可用性，要么选择高可用性容忍弱数据一致性。一般认为，传统的关系数据库在三要素中选择 CA 两个因素，即强数据一致性、高可用性但是可扩展性差。而云存储系统往往关注 AP 因素，即高可扩展性、高可用性，但是以弱数据一致性作为代价。

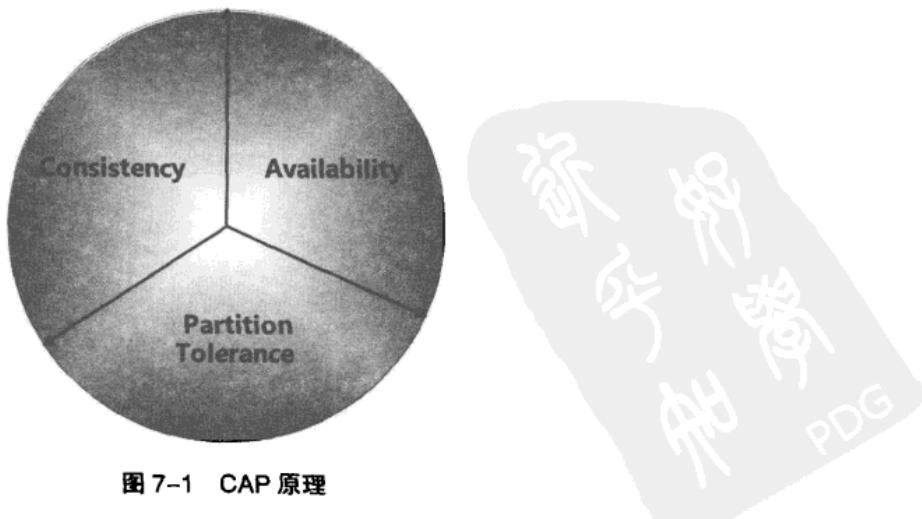


图 7-1 CAP 原理

BASE 原则是和 ACID 原则相反的基本原理，我们先介绍什么是 ACID 原则，ACID 是



关系数据库系统采纳的原则，也是一种简称，其代表含义如下。

**原子性 (Atomicity)**: 是指一个事务要么全部执行，要么完全不执行，也就是不允许一个事务只执行了一半就停止。以银行转账为例，这是一个典型的事务，它的操作可以分成几个步骤：首先从 A 账户取出要转账的金额，A 账户扣除相应的金额之后，将其转入 B 账户的户头，B 账户增加相同的金额。这个过程必须完整地执行，否则整个过程将被取消，回退到事务未执行前的状态。不允许出现从 A 账户已经扣除金额，而没有打入 B 账户这种情形。

- **一致性 (Consistency)**: 事务在开始和结束时，应该始终满足一致性约束条件。比如系统要求  $A+B=100$ ，那么事务如果改变了  $A$  的数值，则  $B$  的数值也要相应修改来满足这种一致性要求。
- **事务独立 (Isolation)**: 如果有多个事务同时执行，彼此之间不需要知晓对方的存在，而且执行时互不影响，不允许出现两个事务交错，间隔执行部分任务的情形。
- **持久性 (Durability)**: 事务的持久性是指事务运行成功以后，对系统状态的更新是永久的，不会无缘由回滚撤销。

数据库系统采纳 ACID 原则，获得高可靠性和强数据一致性。而大多数云存储系统则采纳 BASE 原则，这种原则与 ACID 原则差异很大，具体而言，BASE 原则是指：

- **基本可用 (Basically Available)**: 在绝大多数时间内系统处于可用状态，允许偶尔的失败。
- **软状态或者柔性状态 (Soft State)**: 数据状态不要求在任意时刻都完全保持同步。
- **最终一致性 (Eventual Consistency)**: 与强数据一致性相比，最终一致性是一种弱数据一致性，尽管软状态不要求任意时刻数据保持一致同步，但是最终一致性要求在给定时间窗口内数据达到一致状态。

BASE 原则与 ACID 原则不同，是通过牺牲强数据一致性来获得高可用性的。尽管大多数云存储系统采纳了 BASE 原则，但是有一点值得注意：云存储系统的发展过程正在向逐步提供局部 ACID 特性发展，即全局而言符合 BASE 原则，但是局部支持 ACID 原则，这样就可以吸取两者各自的好处，在两者之间建立平衡，从 Google 的 MegaStore 可以看出这种发展趋势。

大多数云存储系统采纳了最终一致性，在分布式存储架构中，每份数据都要保留多个备份，但是由于客户端程序是并发对数据读/写，可能同时有多个客户端将数据更新到不同的备份中，这可能导致数据的不一致状态，如何维护数据保持一致是个核心问题。所谓强数据一致性，就是要求后续的读取操作看到的都是最新更新的数据；而弱数据一致性则放

宽条件，允许读取到较旧版本的数据，最终一致性则是给出一个时间窗口，在一段时间后，系统能够保证所有备份数据的更新是一致的。

### 7.1.3 数据模型

所谓数据模型，就是云存储架构在应用开发者眼中是何种形式，比较常见的数据模型有两种：**Key/Value** 模式和模式自由（**Schema Free**）列表模式。图 7-2 展示了两者区别，**Key/Value** 模式是一种比较简单的数据模型，每个记录由两个域构成，一个是主键 Key，作为记录的唯一标识，另一个字段则存储记录的数据值；模式自由列表模式情况则复杂一些，同样地，每个记录由一个唯一的主键标识，不同点在于数据值，类似于关系数据库，数据值由若干个列属性构成，但是与数据库不同的是：数据库一旦确定列包含哪些属性，就固定不变，而云存储系统则不受此约束，可以随时增加或者删除某个列属性，同时每一行只存储部分列属性，不必完全存储。

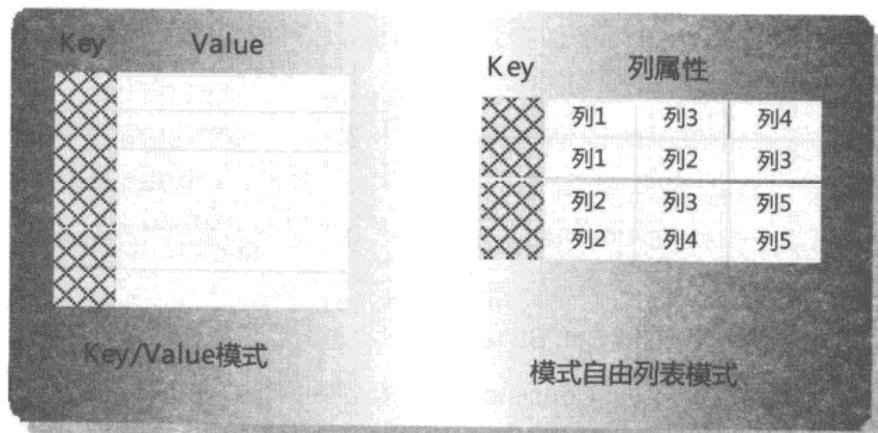


图 7-2 云存储数据模型

在云存储系统内部实现存储结构的时候，最终可以归结为两种实现方式：哈希加链表或者 B+树的方式。哈希加链表查询速度较快，但是一次只能查询一条记录，无法支持批量顺序查找记录（Scan 方式），而 B+树则支持这种查找方式，但是其管理方式相对复杂，所以两种方式各擅胜场，适合不同的应用场合。

### 7.1.4 基本问题

对于一个由大量廉价 PC 构成的分布式存储系统来说，不管具体方案采取何种技术路线，都面临一些共同的问题，这些问题包括。



### 1. 数据如何在机器之间分布

由于数据量巨大，单个机器很难承担存储全部数据的责任，所以必须将数据进行切割，将大数据切割成小份，并将其分配到其他机器上，数据的分布策略是首先要考虑的问题。

### 2. 多备份数据如何保证一致性

在由大量廉价 PC 构成的分布式存储系统中，随时都可能有 PC 出现故障，放置在故障机器上的数据会因此不可用，出于数据可用性方面的考虑，现代云存储系统必须将数据做多备份，并将备份数据放置在不同机器上。而云计算环境下，很多客户端程序会并发对数据进行读/写，这样数据的多个备份之间如何保证其状态是一致的就成了云存储系统的核心问题。

### 3. 如何响应客户端的读/写请求

几千台机器构成了云存储与云计算平台，对于客户端的数据读/写请求，如何将请求在不同功能的机器之间转发，并做出针对性响应，同时读/写延迟尽可能短，数据尽可能保证正确，这些都是云存储与云计算平台需要提供的基本功能和保证。

### 4. 加入（或者坏掉）一台机器如何处理

既然是由大量 PC 构成的系统，随着数据量不断增加，需要的机器也随之增长，如果新加入存储机，系统应该将之纳入管理范围，并自动为其分配任务；同样地，如果某台机器因故障不可用，如何识别这种状态及如何对其负责存储的数据进行迁移与备份，这也是云存储平台的重要问题。

### 5. 如何在机器之间进行负载均衡

大量 PC 共同承担数据存储与读/写响应，很容易出现分工不均的情况，导致有些机器非常繁忙，而有些机器则很空闲，这样的话，繁忙机器很容易成为系统瓶颈，而闲置机器资源又没有充分利用起来，所以如何在机器之间进行负载均衡，使得不会出现瓶颈节点的同时又能充分利用机器资源，这同样是需要考虑的重要问题。

#### 7.1.5 Google 的云存储与云计算架构

Google 对于推动云存储与云计算贡献了很大力量，而且到目前为止，也是在相关技术方面积累最深厚的公司，图 7-3 展示了 Google 的一整套云存储与云计算技术体系。

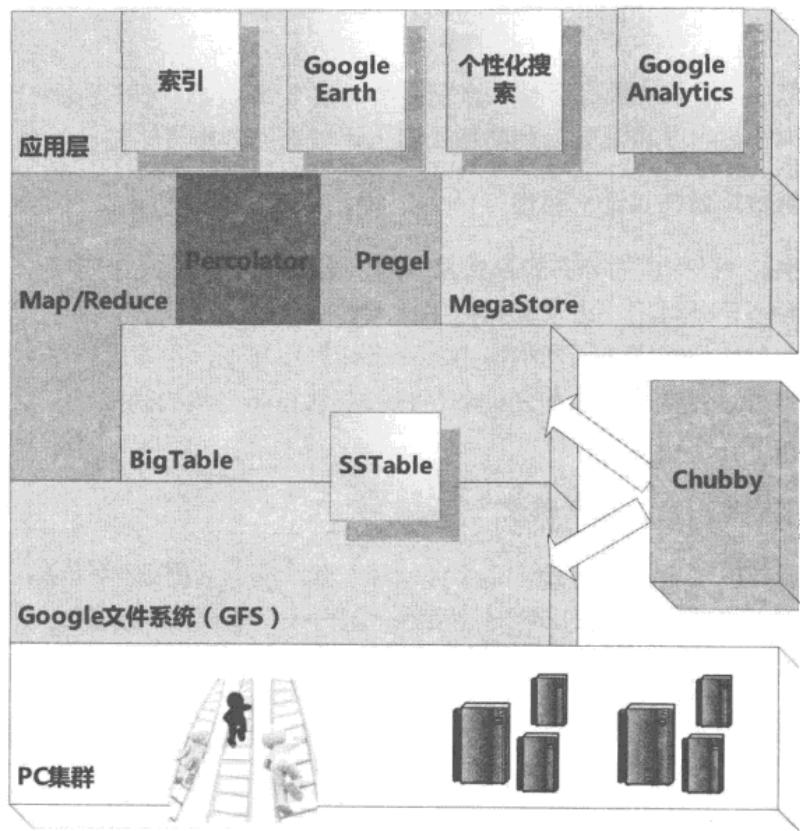


图 7-3 Google 的云存储与云计算架构

从大的角度划分，可以将这些技术划分为两类：一类是云存储技术，另一类是云计算技术。云存储技术包括：GFS 文件系统、Chubby 锁服务、BigTable 及 MegaStore 等一系列不断进化的存储系统；云计算体系则包含 MapReduce、Percolator 及 Pregel 等互补的计算模式。

GFS 是一个分布式文件系统，对海量数据提供了底层的存取支持，至于文件内容本身则不做格式要求；BigTable 提供了数据的结构化和半结构化视图，其数据模型与具体应用更贴近，同时 BigTable 提供了记录行内针对列属性的原子操作，即实现了基于行的事务，但不提供行间或者表间的事务支持；MegaStore 则在 BigTable 基础上在事务支持方面又向数据库方向前进了一步，支持对数据的分组，组内数据支持 ACID 原则，同时强调局部数据的强数据一致性和高可用性，可在此基础上直接提供面向用户的实时交互服务。Chubby 则在 GFS 和 BigTable 中起到了很重要的作用，包括主服务器的选举、元数据的存储、粗粒度的锁服务等。

MapReduce 和 Percolator 这两个模型则是在以上的云存储架构下的计算模型，两者都



是后台计算模型，即其实时性不够，比较适合做后台运算。MapReduce 适合做大规模数据的全局统计，对于数据的增量更新无法支持。而 Percolator 则是对 MapReduce 计算模型的补充，可以对已有数据做部分更新，并在 BigTable 的行事务基础上提供行间事务和表间事务，同时利用观察通知的方式来组织运算系统。而 Pregel 计算模型则是专门针对大型图结构研发的云计算技术。另外，Google 正在进行第二代 GFS 及代号为 Spanner 的第二代 BigTable 的研发。

受 Google 云存储与云计算架构的启发，开源界组织起功能类似的 Hadoop 项目，其项目分支和 Google 相应技术的对比见图 7-4。其中 HDFS 类似于 GFS 文件系统的功能，HBase 则是仿照 BigTable 设计的，Zookeeper 起到类似于 Chubby 的功能，同时 Hadoop 也支持 MapReduce 计算模型。

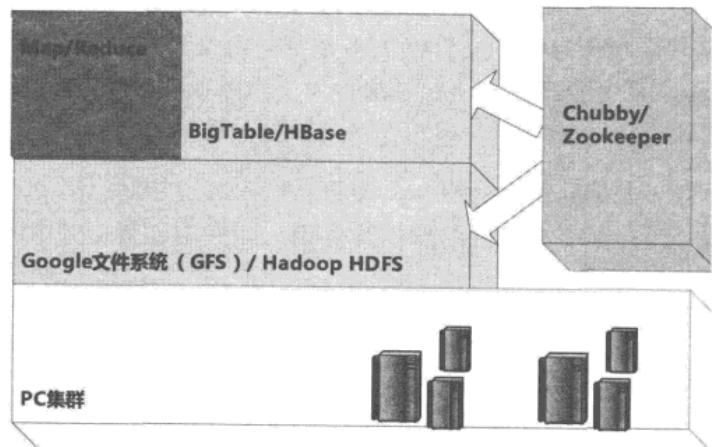


图 7-4 Hadoop 开源架构与 Google 系统对应关系图

Hadoop 得到雅虎公司的资助，近些年获得了很快的发展，很多大的互联网公司都采用它来做大规模数据的后台计算，比如 Facebook 和百度公司就使用 Hadoop，并对这一系统做出了有针对性的改进。

## 7.2 Google 文件系统 (GFS)

Google 文件系统 (Google File System, 简称为 GFS) 是 Google 公司为了能够存储以百亿计的海量网页信息而专门开发的文件系统。在 Google 的整个云存储和云计算技术框架中，GFS 是其他相关技术的基石，因为 GFS 提供了海量非结构化信息的存储平台，并提供了数据的冗余备份、成千台服务器的自动负载均衡及失效服务器检测等各种完备的分布式存储功能。只有在 GFS 提供的基础功能之上，才能开发更加符合应用需求的存储系统和计

算框架。

### 7.2.1 GFS 设计原则

GFS 是针对 Google 公司自身业务需求而开发的，所以考虑到搜索引擎这个应用环境，GFS 在设计之初即定下几个基本的设计原则。

首先，GFS 采用大量商业 PC 来构建存储集群，众所周知，PC 因为是面向普通用户设计的，所以其稳定性并没有很高的保障，尤其是大量 PC 构成的集群系统，每天都有机器死机或者发生硬盘故障，这是一个常态，也是 GFS 在设计时就将其考虑在内的。因此，数据冗余备份、自动检测机器是否还在有效提供服务、故障机器的自动恢复等都在 GFS 的设计目标里。

其次，GFS 文件系统所存储的文件绝大多数都是大文件，文件大小大部分在 100MB 到几个 GB 之间，所以系统的设计应该针对这种大文件的读/写操作做出优化。尽管 GFS 也支持小文件读/写，但是不作为重点，也不会进行有针对性的操作优化。

第三，系统中存在大量的追加写操作，即将新增内容追加到已有文件的末尾，已经写入的内容一般不做更改，很少有文件的随机写行为，即指定已有文件中间某个位置，在这个位置之后写入数据。

第四，对于数据读取操作来说，绝大多数读文件操作都是顺序读，少量的操作是随机读，即按照数据在文件中的顺序，一次顺序读入较大量的数据，而不是不断定位到文件指定位置，读取少量数据。

从下面介绍内容可以看出，GFS 的很多技术思路都是为了满足以上的几个设计目标而提出的。

### 7.2.2 GFS 整体架构

GFS 文件系统主要由 3 个组成部分构成：唯一的主控服务器（Master）、众多的 Chunk 服务器和 GFS 客户端。主控服务器主要做管理工作，Chunk 服务器负责实际的数据存储并响应 GFS 客户端的读/写请求。尽管 GFS 由上千台机器构成，但是在应用开发者眼中，GFS 类似于本地的统一文件系统，分布式存储系统的细节对应用开发者来说是不可见的。

在了解 GFS 整体架构及其组成部分交互流程前，我们首先了解下 GFS 中的文件系统及其文件。在应用开发者看来，GFS 文件系统类似于 Linux 文件系统或者是 Windows 操作系统提供的文件系统，即由目录和存放在某个目录下的文件构成的树形结构。在 GFS 系统中，这个树形结构被称做 GFS 命名空间，同时，GFS 为应用开发者提供了文件的创建、删



除、读取和写入等常见的操作接口（API）。

上节提到，GFS 中存储的都是大文件，文件大小达到几个 GB 是很常见的。虽然每个文件大小各异，但是 GFS 在实际存储的时候，首先会将不同大小的文件切割成固定大小的数据块，每一块被称做为一个 Chunk，通常将 Chunk 的大小设定为 64MB，这样，每个文件就是由若干个固定大小的 Chunk 构成的。图 7-5 是这种情况的示意图。

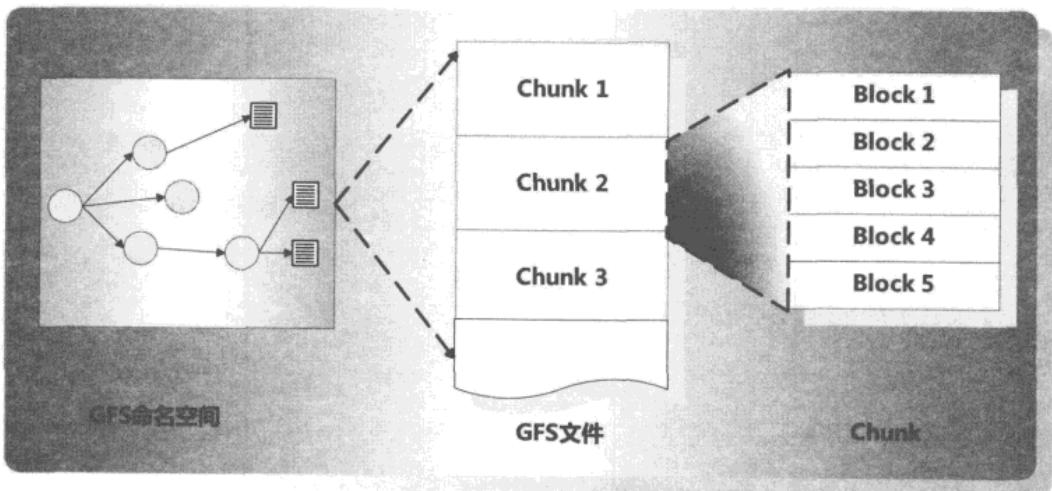


图 7-5 GFS 的文件存储示意图

从图 7-5 中可以看出，每个 GFS 文件被切割成固定大小的 Chunk，GFS 即以 Chunk 为基本存储单位，同一个文件的不同 Chunk 可能存储在不同的 Chunk 服务器上，每个 Chunk 服务器可以存储很多来自于不同文件的 Chunk 数据。另外，在 Chunk 服务器内部，会对 Chunk 进一步切割，将其切割为更小的数据块，每一块被称做为一个 Block，这是文件读取的基本单位，即一次读取至少一个 Block。图 7-5 也标明了这种对 GFS 文件细粒度的切割。总结起来就是：GFS 命名空间由众多的目录和 GFS 文件构成，一个 GFS 文件由众多固定大小的 Chunk 构成，而每个 Chunk 又由更小粒度的 Block 构成，Chunk 是 GFS 中基本的存储单元，而 Block 是基本的读取单元。

图 7-6 显示了 GFS 系统的整体架构，在这个架构中，GFS 主控服务器主要做管理工作，不仅要维护 GFS 命名空间，还要维护 Chunk 的命名空间，之所以如此，是因为在 GFS 系统内部，为了能够识别不同的 Chunk，每个 Chunk 都会赋予一个独一无二的编号，所有 Chunk 的编号构成了 Chunk 命名空间，GFS 主控服务器还记录了每个 Chunk 存储在哪台 Chunk 服务器上等信息。另外，因为 GFS 文件被切割成了 Chunk，GFS 系统内部就需要维护文件名称到其对应的多个 Chunk 之间的映射关系。Chunk 服务器负责对 Chunk 的实际存储，同时响应 GFS 客户端对自己负责的 Chunk 的读/写请求。

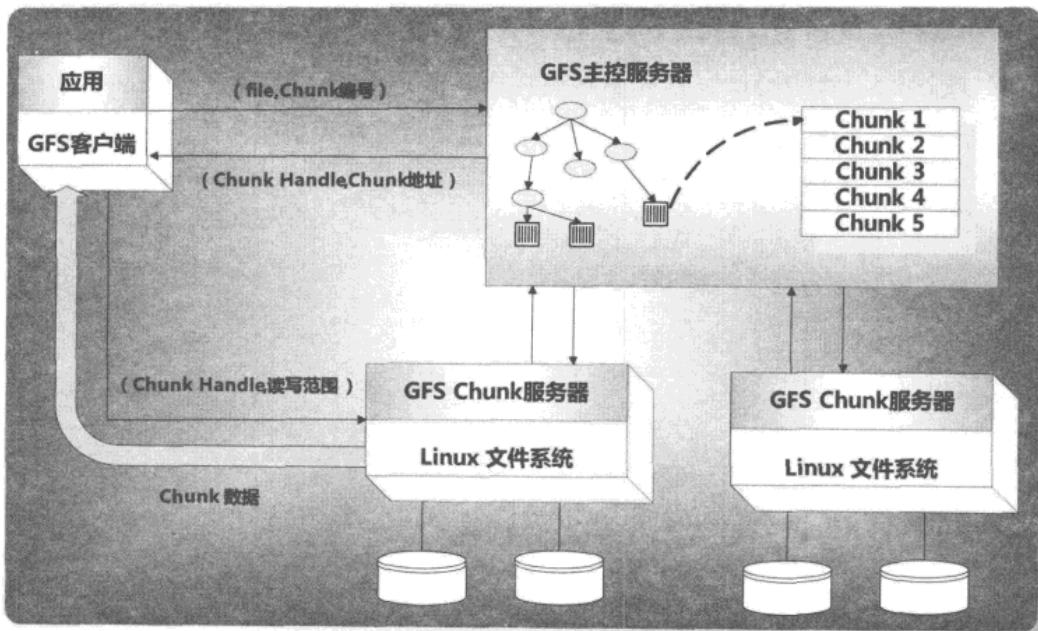


图 7-6 GFS 整体架构

在如图 7-6 所示的 GFS 架构下，我们来看看 GFS 客户端是如何读取数据的。对于 GFS 客户端来说，应用开发者提交的读数据请求是：读取文件 file，从某个位置  $P$  开始读，读出大小为  $L$  的数据。GFS 系统在接收到这种请求后，会在内部做转换，因为 Chunk 大小是固定的，所以从位置  $P$  和大小  $L$  可以推算出要读的数据位于文件 file 中的第几个 Chunk 中，即请求被转换为<文件名 file, Chunk 序号>的形式。随后，GFS 系统将这个请求发送给 GFS 主控服务器，因为 GFS 主控服务器保存了一些管理信息，通过 GFS 主控服务器可以知道要读的数据在哪台 Chunk 服务器上，同时可以将 Chunk 序号转换为系统内唯一的 Chunk 编号，并将这两个信息传回到 GFS 客户端。

GFS 客户端知道了应该去哪台 Chunk 服务器读取数据后，会和 Chunk 服务器建立联系，并发送要读取的 Chunk 编号及读取范围，Chunk 服务器在接收到请求后，将请求数据发送给 GFS 客户端，如此就完成了一次数据读取工作。

### 7.2.3 GFS 主控服务器

Google 的云存储平台有一个显著的特点，就是大量采用主从结构，即单一的主控服务器和众多的存储服务器，主控服务器主要从事系统元数据的存储管理及整个分布式系统的管理，比如负载均衡，数据在存储服务器之间迁移，检测新加入的机器及失效机器等工作。不仅是 GFS，在后续的 Chubby 和 BigTable 的介绍中也可以看出这一明显特点。采取主从



结构的好处是：因为整个系统存在一个全局的主控节点，所以管理起来相对简单。相对应的缺点是：因为主控节点是唯一的，很多服务请求都需要经过主控服务器，所以很容易成为整个系统的瓶颈。另外，正是因为只有唯一的主控节点，所以可能会存在单点失效问题，即如果主控服务器瘫痪，那么整个系统不可用。

GFS 系统是非常典型的具有 Google 风格的设计，本节我们介绍 GFS 主控服务器所管理的系统元数据及对应的管理功能。图 7-7 展示了 GFS 主控服务器所管理的系统数据，维持整个系统正常运转需要 3 类元数据。

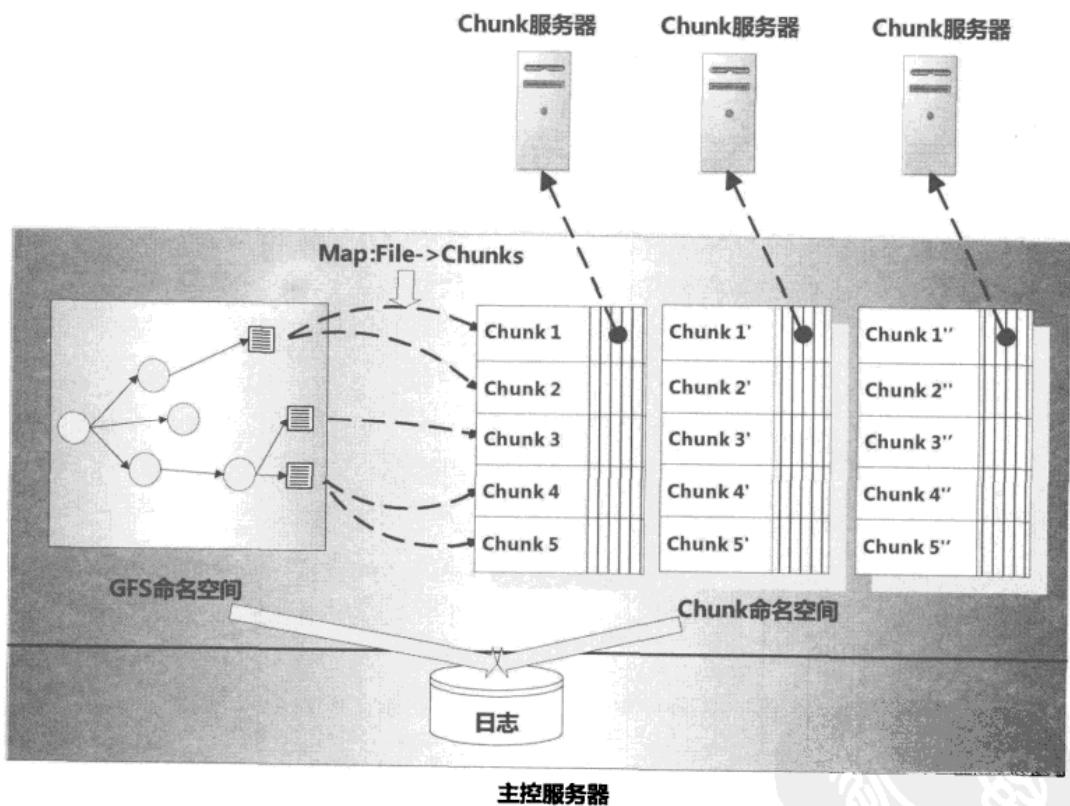


图 7-7 GFS 主控服务器所管理的系统数据

1. **GFS 命名空间和 Chunk 命名空间：**主要用来对目录文件及 Chunk 的增删改等信息进行记录。
2. **从文件到其所属 Chunk 之间的映射关系：**因为一个文件会被切割成众多 Chunk，所以系统需要维护这种映射关系。
3. **每个 Chunk 在哪台 Chunk 服务器存储的信息：**在 GFS 系统中，每个文件会被切

割成若干 Chunk，同时每个 Chunk 会被复制多个备份，并存储在不同服务器上，之所以如此，原因是 GFS 由众多的普通 PC 构成，而 PC 死机是经常事件，如果 Chunk 服务器不可用，那么存储在其上的 Chunk 信息会丢失，为了避免这种情况发生，将每个 Chunk 复制多份，分别存储在不同的机器上，这样即使发生机器故障，也可以在其他机器上找到对应的 Chunk 备份信息。云存储平台必须提供这种数据冗余来保证数据的安全。在如图 7-7 所示的例子中，每个 Chunk 保留 3 个备份，比如对于 Chunk1 来说，Chunk1' 和 Chunk1'' 是其两个复制文件，这 3 个 Chunk 分别存储在不同的 Chunk 服务器中。

有了以上 3 类管理信息，GFS 就可以根据文件名找到对应的 Chunk，同时知道每个 Chunk 存储在哪台 Chunk 服务器上，GFS 客户端程序通过 GFS 主控服务器就可以知道应该到哪里去读/写相应的数据，整个系统就能正常运转起来。

由于管理数据非常重要，所以其安全性必须得到保障，如果管理数据丢失，那么这个 GFS 系统也就不可用。GFS 将前两类管理信息（命名空间及文件到 Chunk 映射表）记录在系统日志文件内，并且将这个系统日志分别存储在多台机器上，这样就避免了信息丢失的问题。对于第 3 类管理数据（Chunk 存储在哪台服务器的信息），主控服务器在启动时询问每个 Chunk 服务器，之后靠定期询问来保持最新的信息。

除了存储管理系统元信息外，主控服务器主要承担一些系统管理工作，比如创建新 Chunk 及其备份数据，不同 Chunk 服务器之间的负载均衡，如果某个 Chunk 不可用，则负责重新生成这个 Chunk 对应的备份数据，以及垃圾回收等工作。

在对数据进行备份和迁移的时候，GFS 重点考虑两个因素：一个是 Chunk 数据的可用性，即如果发现 Chunk 数据不可用，要及时重新备份，以免某个 Chunk 的所有备份都不可用导致数据丢失；另外一个要尽可能减少网络传输压力，因为在不同机器间传递数据，因数据量巨大，所以尽可能减少网络传输压力对于系统整体性能表现很重要。

#### 7.2.4 系统交互行为

本节我们以 GFS 系统如何完成写操作来介绍系统中各个组成部分的交互行为。上文介绍过，出于系统可用性考虑，GFS 系统为每份 Chunk 保留了另外两个备份 Chunk，而 GFS 客户端发出写操作请求后，GFS 系统必须将这个写操作应用到 Chunk 的所有备份，这样才能维护数据的一致性。为了方便管理，GFS 对于多个相互备份的 Chunk，从中选出一个作为主备份，其他的被称做次级备份，由主备份决定次级备份的数据写入顺序（参考图 7-8）。

图 7-8 是 GFS 执行写操作的整体流程。GFS 客户端首先和主控服务器通信，获知哪些 Chunk 服务器存储了要写入的 Chunk，包括主备份和两个次级备份的地址数据。之后，GFS



客户端将要写入的数据推送给 3 个备份 Chunk，备份 Chunk 首先将这些待写入的数据放在缓存中，然后通知 GFS 客户端是否接收成功，如果所有的备份都接收数据成功，GFS 客户端通知主备份可以执行写入操作。主备份自己将缓存的数据写入 Chunk 中，通知次级备份按照指定顺序写入数据，次级备份写完后答复主备份写入成功，主备份会通知 GFS 客户端这次写操作成功完成。

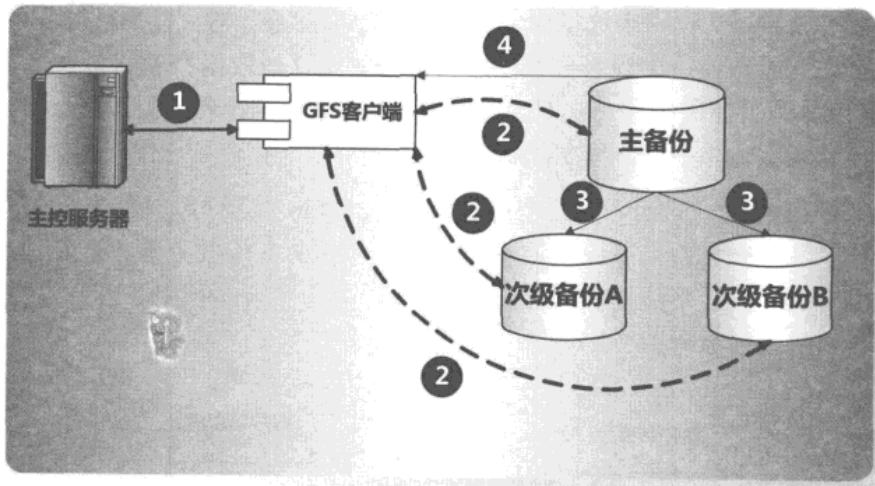


图 7-8 GFS 写操作

上述的写入过程都假设每个步骤成功完成，实际上，每个步骤都有可能出现问题，GFS 系统针对这些可能问题都有相应的处理措施，以保证数据的完整性和可用性。

### 7.3 Chubby 锁服务

Chubby 是 Google 公司研发的针对分布式系统资源管理的粗粒度锁服务，一个 Chubby 实例大约可以负责 1 万台 4 核 CPU 机器之间对资源的协同管理。这种锁服务主要的功能是让众多客户端程序进行相互之间的同步，并对系统环境或者资源达成一致认知。

在客户端程序看来，Chubby 好像是一个类似于文件系统的目录和文件管理系统，并在此基础上提供针对目录和文件的锁服务，Chubby 的文件主要存储一些管理信息或者基础数据，Chubby 要求对文件内容一次性全部读完或者写入，这是为了尽可能抑制客户端程序写入大量数据到文件中，因为 Chubby 的目的不是数据存储，而是对资源的同步管理，所以不推荐在文件中保存大量数据。同时，Chubby 还提供了文件内容更改后的通知机制，客户端可以订阅某个文件，当文件内容发生变化或者一些系统环境发生变化，Chubby 会主动通知这些订阅该文件的客户端，以使得这种信息变化快速传播。

Chubby 的理论基础是 Paxos 一致性协议，Paxos 是在完全分布环境下，不同客户端能够通过交互通信并投票，对于某个决定达成一致的算法。Chubby 以此为基础，但是也做了改造，Paxos 是完全分布的，没有中心管理节点，所以要通过多轮通信和投票来达成最终一致，所以效率会比较低，Chubby 出于系统效率考虑，增加了一些中心管理策略，在达到同一目标的情况下改善了系统效率。

在 GFS 中，Chubby 被用来选举哪台服务器作为主控服务器，在 BigTable 中，则有很多功能：选举主控服务器，主控服务器用此服务来发现其他数据存储服务器，客户端程序根据 Chubby 来找到主控服务器及在 Chubby 文件中存储部分管理数据等，读者在阅读后续章节时可以体会到 Chubby 的具体功能和作用。

Chubby 服务由多个 Chubby 单元构成，每个 Chubby 单元一般包含 5 台服务器，通过选举的方式推举其中一台作为主控服务器，其他 4 台作为备份服务器，之所以这样，主要是防止单台服务器死机后不能提供服务，多台机器可以在某台主控服务器不能提供服务时，由另外一台机器接管（参考图 7-9）。

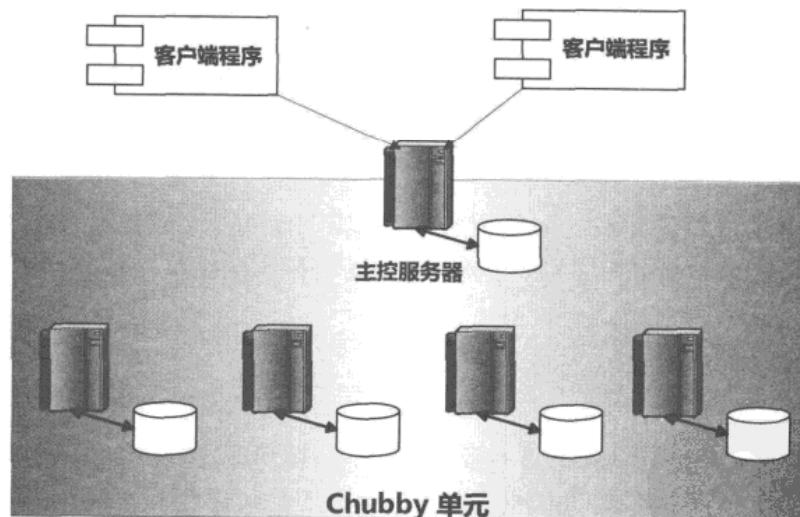


图 7-9 Chubby 体系结构

主控服务器由所有机器选举推出，但是是有任期的，在一段时间内由选举出的服务器充当主控服务器，当任期期满后，会再次投票选举出新的主控服务器。

所有客户端对 Chubby 的读/写请求都由主控服务器来负责，主控服务器遇到写请求后，会更改维护的管理数据，而其他所有备份服务器只是同步管理数据到本地，保持数据和主控服务器一致，当备份机器接收到请求时，会将请求转发给主控服务器，通过这种集中式管理，容易维护管理数据的一致性。



Chubby 向外提供了类似于简单文件系统的接口，树形结构的层级目录和文件构成了 Chubby 管理的数据（参考图 7-10）。比如某个目录节点的名称为：LS/foo/wombat/punch，其中 LS 是整个系统中最高层级的目录，代表锁服务（Lock Service），foo 是某个 Chubby 单元的名称，而 wombat/punch 则是这个 Chubby 单元管理的树形目录结构。

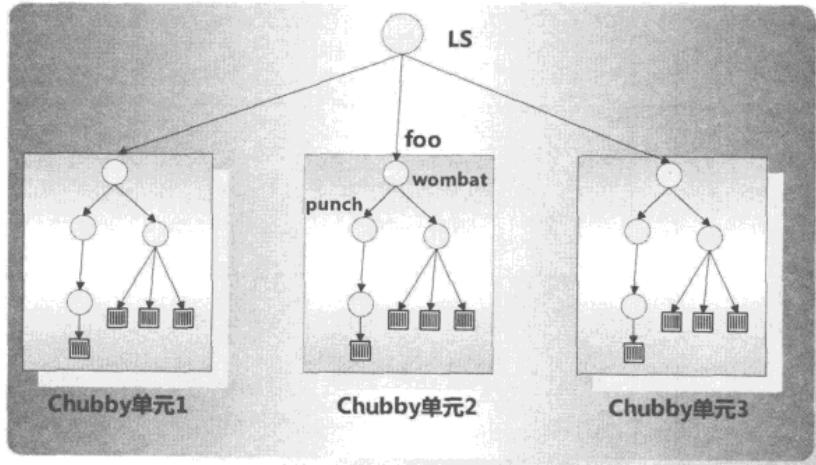


图 7-10 Chubby 的树形目录结构

Chubby 在这个树形目录结构中提供了很多管理功能，比如针对某个目录或者文件的加锁服务，针对目录或者文件的访问权限控制，文件内容存取及事件通知机制。通过这种结构，可以有效实现分布式系统中的同步协同和资源管理功能。

## 7.4 BigTable

BigTable 是一种针对海量结构化或者半结构化数据的存储模型，在 Google 的云存储体系中处于核心地位，起到了承上启下的作用。GFS 是一个分布式海量文件管理系统，对于数据格式没有任何假定，而 BigTable 以 GFS 为基础，建立了数据的结构化解释，对于很多实际应用来说，数据都是有一定格式的，在应用开发者看来，BigTable 建立的数据模型与应用更贴近。MegaStore 存储模型和 Percolator 计算模型都是建立在 BigTable 之上的存储和计算模型。从此可看出，BigTable 在其中的地位之重要。

### 7.4.1 BigTable 的数据模型

所谓数据模型，就是说在应用开发者眼里 BigTable 是怎样的一种结构。BigTable 本质上是一个三维的映射表，其最基础的存储单元是由（行主键、列主键、时间）三维主键（Key）

所定位的。

图 7-11 展示了一个被称为 WebTable 的具体表格，里面存储了互联网的网页。表中每一行存储了某个网页的相关信息，比如网页内容、网页 MetaData 元信息、指向这个网页的链接锚文字等，每行以网页的逆转 URL 作为这一行的主键。BigTable 要求每行的主键一定是字符串的形式，而且在存储表格数据时，按照行主键的字母大小顺序排序存储。

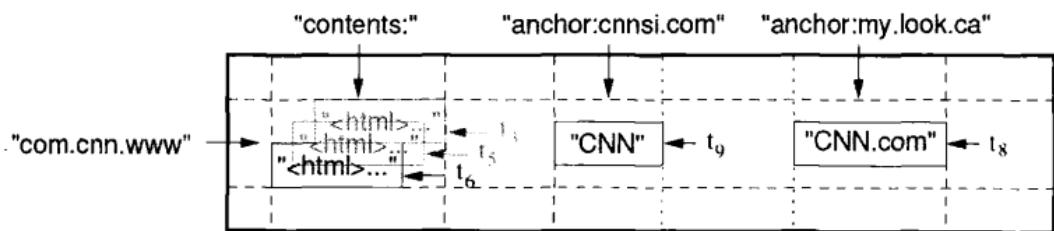


图 7-11 BigTable 数据模型：WebTable

BigTable 中的列主键包含两级，其中第 1 级被称为列家族（Column Families），第 2 级被称为列描述符（Qualifier），两者共同组成一个列的主键，即：

$$\text{列主键} = \text{列家族: 列描述符}$$

以 WebTable 表为例，链接锚文字组成一个列家族 anchor，而每个网页的 URL 地址作为列描述符。anchor: cnnsi.com 这个列主键代表的含义是：这一列存储的是 cnnsi.com 这个网页指向其他页面的链接锚文字。

BigTable 内可以保留同一信息随着时间变化的不同版本，这个不同版本由时间维度来进行表达。比如以行主键 com.cnn.www 和列主键 anchor: cnnsi.com 定位的信息，代表了网页 www.cnnsi.com 指向 www.cnn.com 的链接锚文字，随着时间的变化，这个链接锚文字也可能会不断更改，所以可以存储多个更改版本，比如例子中显示的 T9 这个时间的锚文字内容为 CNN，此外还可以保留其他时间的锚文字信息。

BigTable 的数据模型粗看很像关系型数据库的关系模型，但是两者有重大差别，关系型数据库的列在设计表格之初就已经指定，而 BigTable 是可以随时对表格的列进行增删的，而且每行只存储列内容不为空的数据，这被称做模式自由型（Schema Free）数据库。

BigTable 是个分布式的海量存储系统，成百上千台机器为应用的相关表格提供存取服务。在实际存储表格信息时，会将表格按照行主键进行切割，将一段相邻的行主键组成的若干行数据作为一个存储单元，这被称为一个子表（Tablet），表格由子表构成，而每个子表的数据交由子表服务器来进行管理。图 7-12 是这种情形的形象说明。

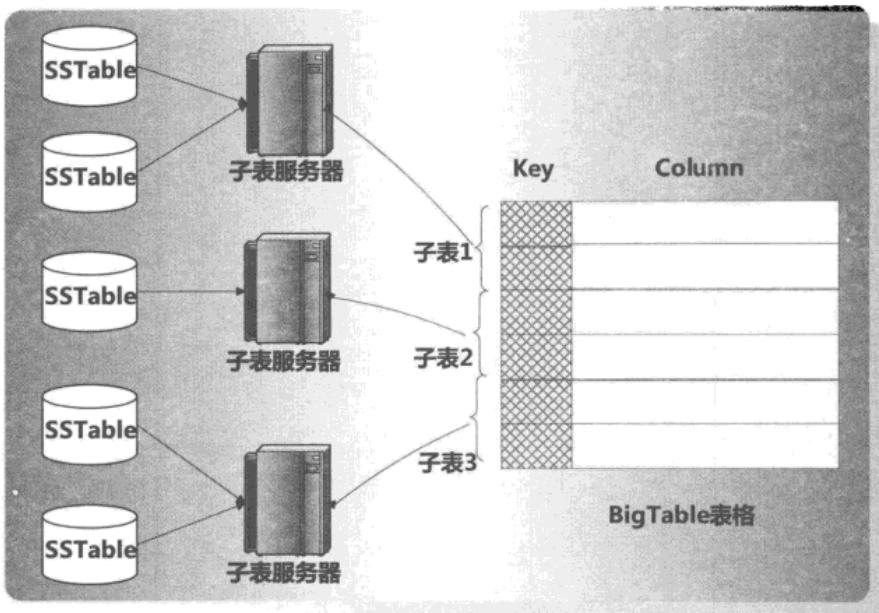


图 7-12 BigTable 表格的子表

BigTable 为应用开发者提供了 API，利用这些 API 可以创建或者删除表格，设定表格的列，插入删除数据，以行为单位来查询相关数据等操作。另外，BigTable 提供针对行数据的事务操作，而不同记录之间不提供事务保证。

#### 7.4.2 BigTable 整体结构

上一小节介绍了 BigTable 的数据模型，在应用开发者看来，BigTable 就是由很多类似于 WebTable 这样的三维表格共同组成的一个系统，应用开发者只需要考虑具体应用包含哪些表格，每个表格对应包含哪些列，然后就可以在相应的表格内以行为单位增删内容。至于具体表格是如何在内部存储的则交由 BigTable 来进行管理。

图 7-13 是 BigTable 的整体结构示意图，其中主要包含：主控服务器（Master Server）、子表服务器（Tablet Server）和客户端程序（Client）。每个表格将若干连续的行数据划分为一个子表（Tablet），这样表格的数据就会被分解为一些子表。子表服务器主要负责子表的数据存储和管理，同时需要响应客户端程序的读/写请求，其负责管理的子表以 GFS 文件的形式存在，BigTable 内部将这种文件称之为 SSTable，一个子表就是由子表服务器磁盘中存储的若干个 SSTable 文件组成的；主控服务器负责整个系统的管理工作，包括子表的分配、子表服务器的负载均衡、子表服务器失效检测等。客户端程序则是具体应用的接口程序，直接和子表服务器进行通信交互，来读/写某个子表对应的数据。

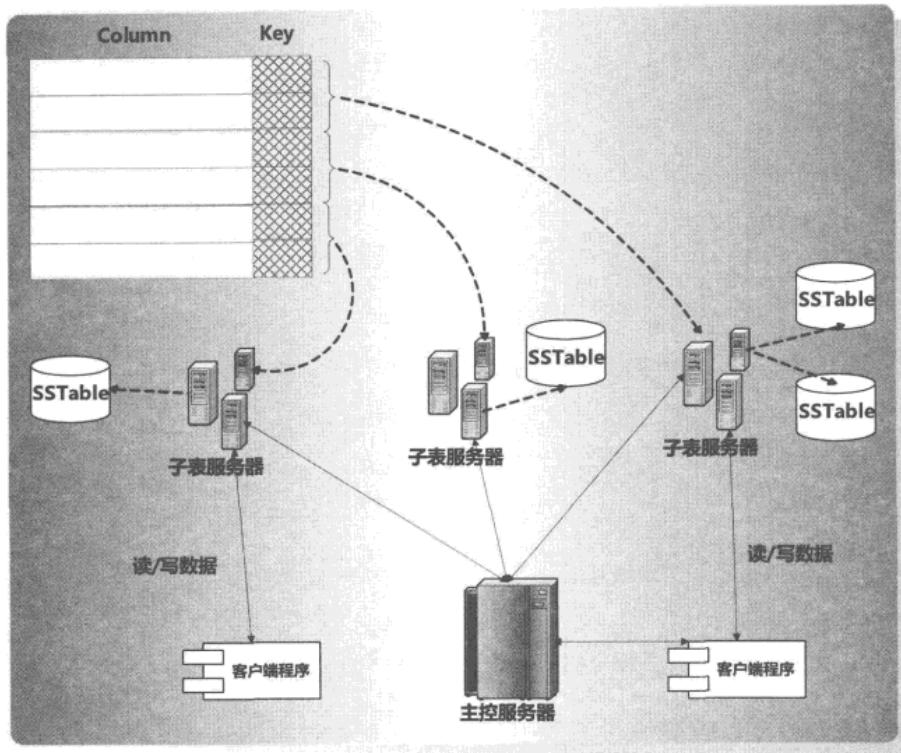


图 7-13 BigTable 的整体结构示意图

### 7.4.3 BigTable 的管理数据

对于具体应用来说，可以根据需要在 BigTable 系统中创建自己的表格，每个表格都会被分割成若干子表，并存储在不同的子表服务器中。那么，BigTable 如何知道每个子表存储在哪台子表服务器中呢？要知道这一点，需要维护一些特殊的管理数据。

BigTable 利用 Chubby 系统和一个被称为元数据表（MetaData Table）的特殊表格来共同维护系统管理数据（参见图 7-14）。元数据表是 BigTable 中一个起着特殊作用的表，这个表格的每一行记载了整个 BigTable 中某个具体子表存储在哪台子表服务器上等管理信息，但是它一样也会被切割成若干子表并存储在不同的子表服务器中。这个表的第一个子表被称为 Root 子表，用来记录元数据表自身除 Root 子表外其他子表的位置信息，因为元数据表的子表也是分布在不同机器上的，通过 Root 子表的记录就可以找到元数据表中其他子表存储在哪台机器上，即通过 Root 子表可以找到完整的元数据表。

元数据表中其他子表的每一行，则记录了 BigTable 中应用程序生成的表格（用户表）某个子表的管理数据。其中，每一行以用户表表名和在这个子表内存储的最后一个行主键

共同构成元数据表内此条记录的行主键，在记录行的数据里则存储了这个子表对应的子表服务器等其他管理信息。而 Chubby 中某个特殊文件则指出了 Root 子表所在的子表服务器地址。这样，Chubby 文件、Root 子表及元数据表中的其他子表构成了三级查询结构，通过这个层级结构就可以定位具体应用的某个子表放置在哪台子表服务器上。

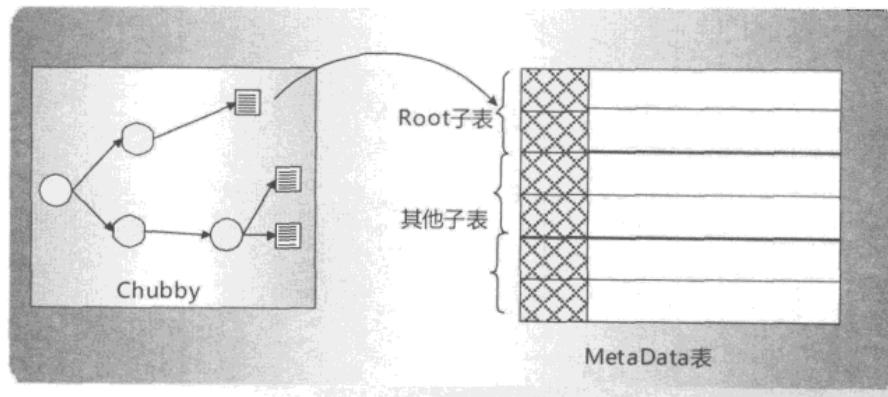


图 7-14 利用 Chubby 和元数据表来共同维护系统管理数据

图 7-15 是这种三级查询结构的示意图，假设某个客户端程序需要查找用户表的某行记录，首先读取 Chubby 系统中的特殊文件，从这个文件可以得知 Root 子表的所在位置，然后根据 Root 子表获知元数据表其他子表所在位置，其他子表每一行的行主键是由用户表表名和对应子表最后一行的行主键共同构成的，所以通过和要查询的用户表及其待查记录的行主键比较，就可以知道是哪台子表服务器存储着这条记录，之后客户端程序将这些信息缓存在本地，并直接和子表服务器通信来读取对应的数据。

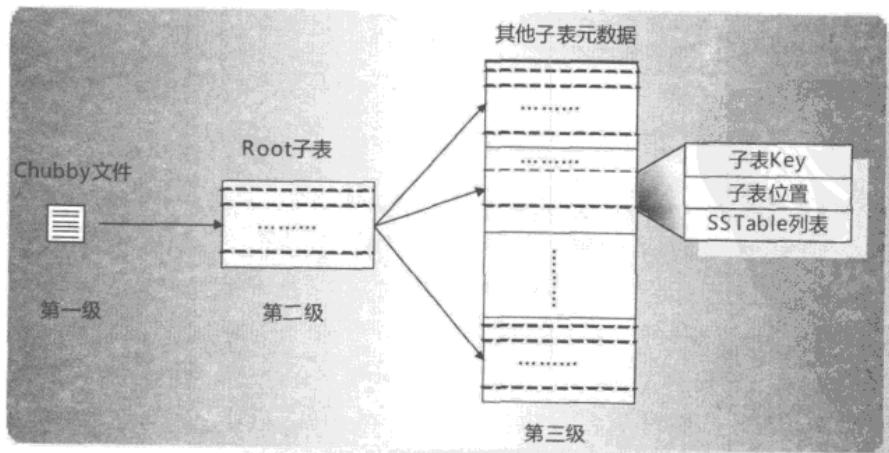


图 7-15 三级查询结构

#### 7.4.4 主控服务器 (Master Server)

主控服务器在 BigTable 中专门负责管理工作，比如自动发现是否有新的子表服务器加入，是否有子表服务器因为各种故障原因不能提供服务，是否有些子表服务器负载过高等情况，并在各种情况下负责子表服务器之间的负载均衡，保证每个子表服务器的负载都是合理的。

当主控服务器刚被启动时，需要获知子表的分配情况，图 7-16 是主控服务器启动时的运行流程。

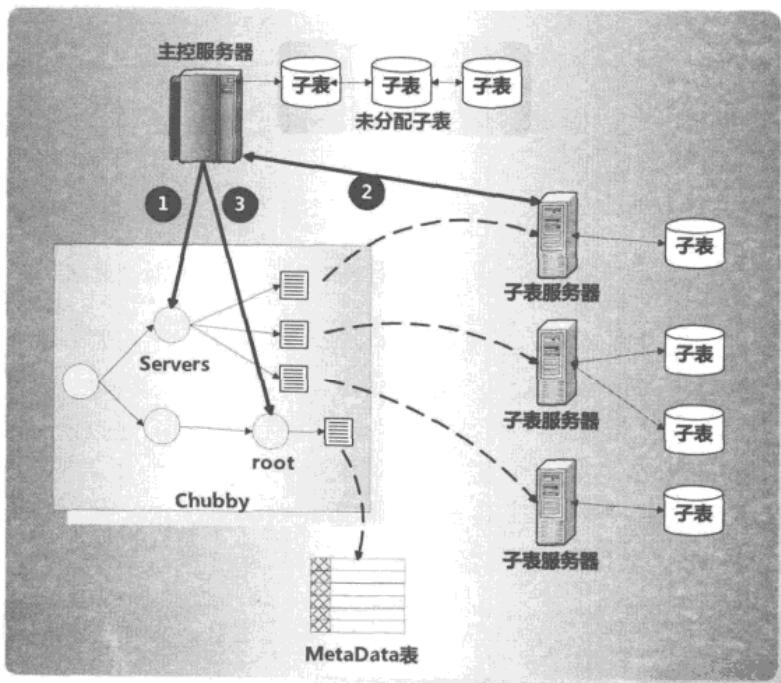


图 7-16 主控服务器启动时的运行流程

Chubby 在 BigTable 的正常运转过程中起了很大的作用，不但在其中存储了最基础的管理数据，还提供了粗粒度的加锁服务。在 Chubby 的树形结构中，有一个特殊的 Servers 目录，每个子表服务器在该目录下生成对应的文件，记载了这个子表服务器的 IP 地址等管理信息。当主控服务器启动时，首先在 Chubby 中获得一个 Master 锁，这样可以阻止其他主控服务器再次启动，避免整个系统中出现多个管理节点。之后，主控服务器读取 Servers 目录，从该目录下的文件可以获得每个子表服务器的地址信息。获得地址信息后，在之后的管理过程中，主控服务器就可以直接和子表服务器进行通信。在启动时，主控服务器和子表服务器通信，获知每个子表服务器存储了哪些子表并记录在内存管理数据中。之后，

主控服务器从 Chubby 的 root 节点可以读取 MetaData 元数据，这里记载了系统中所有子表的信息；通过 MetaData 和子表服务器反馈的信息，两者对比，可能会发现有一部分子表在 MetaData 中，但是没有子表服务器负责存储，说明这些子表是未获得分配的内容，所以将这些子表信息加入一个未分配子表集合中，之后会在适当的时机，将这些未分配子表分配给负载较轻的子表服务器。

当有新的子表服务器加入 BigTable 系统中，这个子表服务器会在 Chubby 的 Servers 目录下生成对应的文件，主控服务器通过周期性地扫描 Servers 目录下的文件可以很快获知有新的子表服务器加入，之后可以将高负载的其他子表服务器的部分数据，或者是未分配子表中的数据交由新加入的服务器来负责管理。

主控服务器会周期性地询问子表服务器的状态，当无法和子表服务器取得联系后，会将 Chubby 的 Servers 目录下对应的文件删除，并将这个子表服务器负责管理的子表放入未分配子表中，之后会将这些子表分配到有空闲空间的子表服务器中。

#### 7.4.5 子表服务器 (Tablet Server)

子表服务器是 BigTable 系统中用来存储和管理子表数据的，从具体功能来讲，子表服务器支持以下功能：

- 存储管理子表数据，包括子表存储、子表恢复、子表分裂、子表合并等。
- 响应客户端程序对子表的写请求。
- 响应客户端程序对子表的读请求。

##### 1. 更新子表数据

对子表内容的更新包括插入或删除行数据，或者插入删除某行的某个列数据等操作。

图 7-17 是子表服务器响应客户端程序更新操作的流程图，当子表服务器接收到数据更新请求时，首先将更新命令记入 CommitLog 文件中，之后将更新数据写入内存中的 MemTable 结构中，当 MemTable 里容纳的数据超过设定大小时，将内容输出到 GFS 文件系统中，形成一个新的 SSTable 文件。一个具体的子表数据就是由若干个陆续从 MemTable 产生的 SSTable 文件构成的。

在 BigTable 系统中，所有对子表的更新操作都是在内存中完成的，MemTable 即是内存中开辟的缓冲区，用来容纳子表的数据更新内容。对于一个分布式存储系统来说，系统故障经常发生。假设在 MemTable 还没有将内存更新内容输出到 SSTable 的时候，子表服务器宕机，那么 MemTable 的数据会丢失。CommitLog 的引入就是为了防止这种情况发生，因为在写入 MemTable 之前的所有操作都在 CommitLog 中记录，即使子表服务器宕机，也

可以根据 CommitLog 的命令再次恢复 MemTable 的内容。

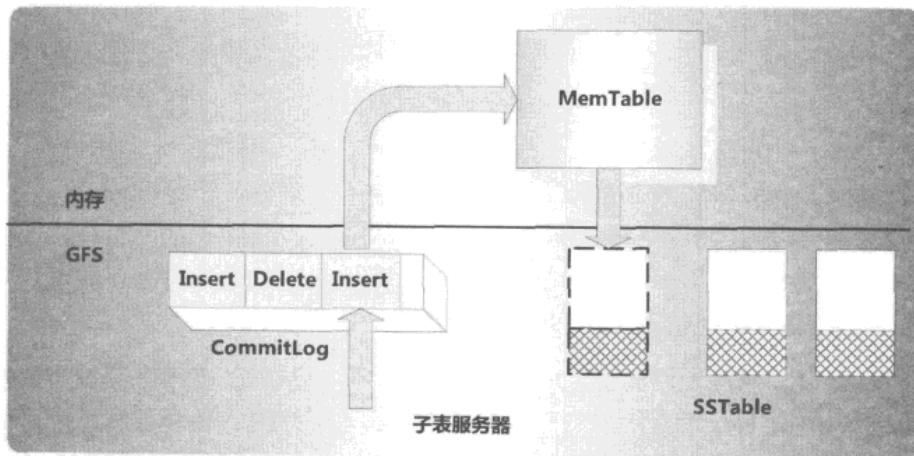


图 7-17 更新操作

SSTable 是 BigTable 内部用来存储数据的文件，其具有特定的格式。图 7-18 体现了 SSTable 的格式信息。每个 SSTable 划分为两块：数据存储区和索引区。数据存储区用来存储具体的数据，本身又被划分成小的数据块，每次读取的单位就是一个数据块。索引区记载了每个数据块存储的行主键范围及其在 SSTable 中的位置信息。当 BigTable 打开一个 SSTable 文件的时候，系统将索引区加载入内存，当要读取一个数据块时，首先在内存中的数据块索引中利用二分查找，快速定位某条行记录在 SSTable 中的位置信息，之后就可以根据位置信息一次性读取某个数据块。

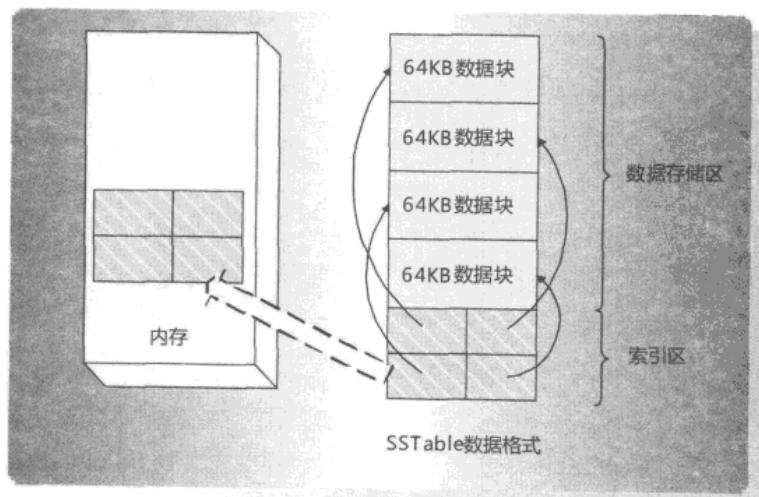


图 7-18 SSTable 结构



## 2. 读取子表数据

由上面叙述可知，一个子表是由内存中的 MemTable 和 GFS 中存储的若干 SSTable 文件构成的。在 MemTable 和 SSTable 中存储的数据都是按照行主键的字母顺序排序的，所以很容易将这些文件看做一个按照行主键排好序的整体序列结构，而读取操作就是首先查找数据的存储位置，如果找到则读出数据。由于 SSTable 在 GFS 文件系统中，为了加快查找速度，BigTable 除了块索引外，还引入了布隆过滤器（Bloom Filter）算法，这种算法只占用少量内存，就可以快速判断某个 SSTable 文件是否包含要读取数据的主键，这样对于很多读操作，避免了在磁盘中查找，加快读取速度（参考图 7-19）。

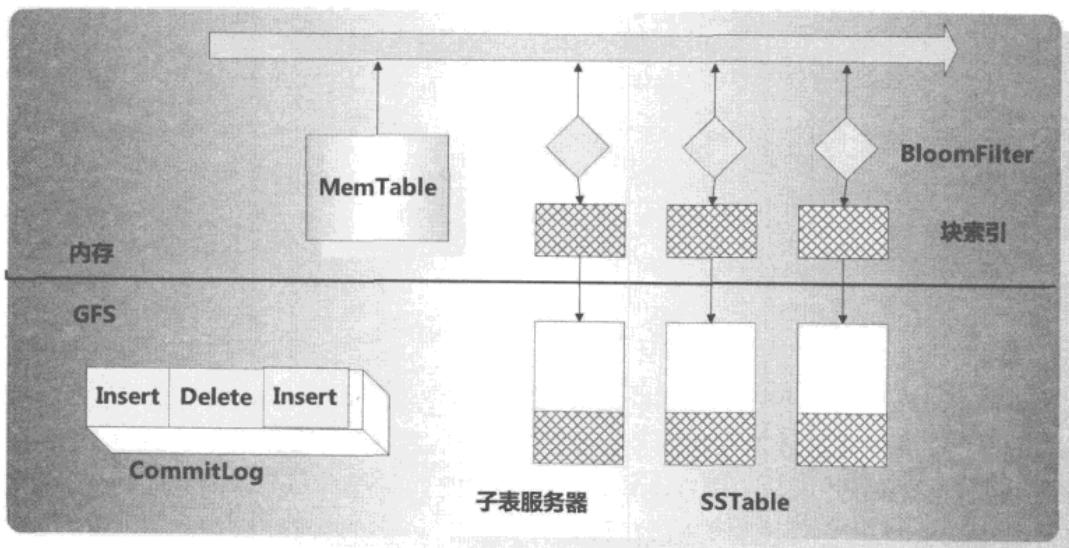


图 7-19 读操作

## 3. SSTable 合并

如果 SSTable 数量过多，会影响系统读取效率，所以子表服务器会周期性地对子表的 SSTable 和 MemTable 进行合并。根据合并规模的差异，存在 3 种不同类型的合并策略：微合并（Minor Compaction）、部分合并（Merging Compaction）及主合并（Major Compaction）（参考图 7-20）。

当 MemTable 写入数据过多，会将内存中的数据写入磁盘中一个新的 SSTable 中，这个过程被称为微合并。这种合并有两种功能：首先，可以减少内存消耗量；其次，由于 MemTable 内数据量不会无限制增长，即使这个子表服务器宕机后重启，系统根据 CommitLog 恢复 MemTable 的速度也会较快。

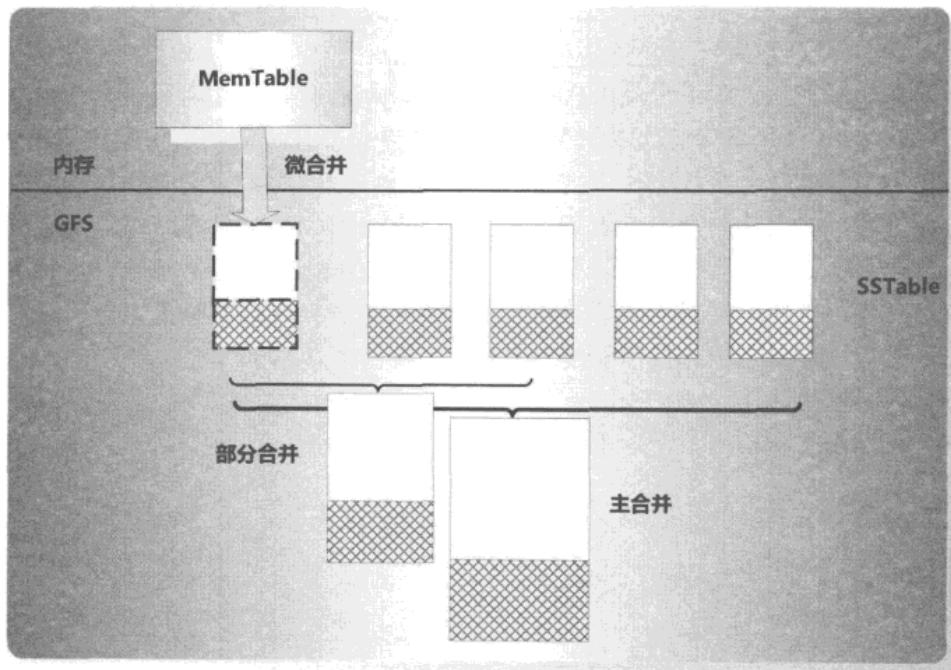


图 7-20 SSTable 合并

把 MemTable 的内容和部分 SSTable 合并的过程叫做部分合并，通过部分合并，可以减少 SSTable 的数量，增加读操作的效率。而将 MemTable 和所有 SSTable 进行合并的过程被称为为主合并，这种合并周期的执行，使得所有子表数据集中到一个 SSTable 中。同时，在合并过程中，会将已经标记为“删除”的记录抛弃，有效回收存储资源。

#### 4. 子表恢复 (Tablet Recovery)

当子表服务器死机后，BigTable 提供了完善的子表恢复机制（参考图 7-21）。当死机的子表服务器重新启动后，会从元数据子表（MetaData）中读取管理信息，包括子表服务器负责管理的子表对应哪些 SSTable 文件，以及 CommitLog 对应的恢复点（Redo Point）。根据 CommitLog 恢复点，子表服务器可以找到 CommitLog 的对应位置，恢复从这个位置之后的所有更新行为到 MemTable 中，这样就完成了 MemTable 的重建工作。从元数据子表中读取到对应的 SSTable 文件后，子表服务器将对应的 SSTable 的块索引读入内存，这样就能够完全恢复到死机前的状态。从这个过程可以看出 CommitLog 的具体功能。

除了上述功能之外，子表服务器还负责子表分裂的管理，当某个子表存储的数据量过大，会将其分裂为两个均等大小的子表，并将相应的管理信息传递到元数据子表中记录，之后的数据更新及读取分别在相应的子表中进行。

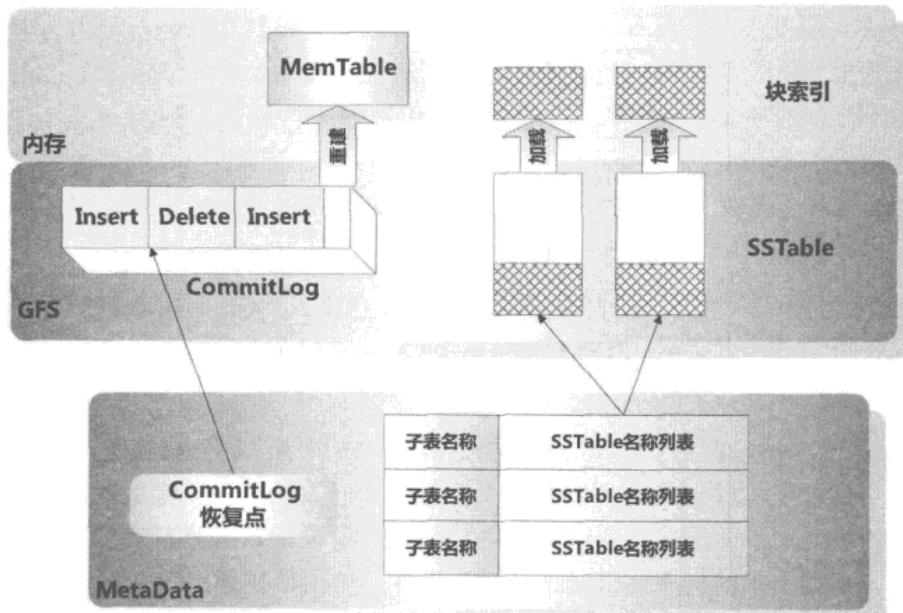


图 7-21 BigTable 提供的子表恢复机制

## 7.5 Megastore 系统

Google 的绝大多数应用是建立在 GFS 文件系统和 BigTable 存储系统之上的，从前述章节关于这两个存储系统的运行机制可以看出，这套系统比较适合做大量数据的后台计算，对于实时交互的应用场景来说，并非这套系统的强势应用场合。

时移世变，目前大多数互联网应用中相当重要的一部分需要与用户进行实时交互，如何针对这些应用的特点构造海量存储系统？这是非常具有挑战性的问题。Megastore 即是 Google 针对这类应用自行研发的海量存储系统。

我们首先看下这类应用本身对存储系统有哪些特殊的要求。首先，由于数据量太大，需要系统具有高可扩展性（Scalability），这对海量存储系统来说是一项非常基础的要求。其次，对互联网应用来说，推出时间早晚其最终的结局可能差异很大，所以存储系统应该支持应用的快速开发和部署。再次，因为是实时与用户交互，所以数据读/写要求满足高速度低延迟的要求。另外，存储系统应该能够保证数据的一致性要求，否则用户写入数据后看到的仍然是过时的老数据，其体验可想而知。同时，系统要具有高可用性（Availability），即使服务提供方内部出现大规模故障，也应该保证用户仍然可用服务。

上面这些对存储系统的要求有些是有内在矛盾的，在所有方面做到最优不太可能，问题就成为了：如何提供一种折中方案，能够在几者之间取得平衡。

目前解决大规模数据存储有两种不同的解决方案，一种是传统的数据库方式，这种方法提供保证数据一致性的接口，而且开发者使用起来非常简单，开发成本低，但是这种方法可扩展性不高，面对超大规模数据无能为力。另外一个方式是 NoSQL 的方法，BigTable 就是一种典型的 NoSQL 技术方案，这种方案可扩展性强、可用性高，能够处理超大规模数据存储，但是往往无法保证数据的强一致性，比如 BigTable 只能对行数据提供事务支持，对跨行跨表操作的数据一致性无法保证。

Megastore 考虑到需求的矛盾性及目前数据库方案和 NoSQL 方案各自的优缺点，希望能够找到一条折中的技术路线，既能够提供 NoSQL 方案的高扩展性，又能够吸取数据库方案的强数据一致性保证。

Megastore 的基本思路是：将大规模数据进行细粒度切割，切分成若干实体群组（Entity Group），在实体群组内提供满足 ACID 语义的强数据一致性服务，但是在实体群组之间提供相对弱些的数据一致性保证。利用改造的 Paxos 协议来将数据分布到多个数据中心，这样同时满足了数据请求的高速度低延迟及高可用性，可用性是通过将数据分布到不同数据中心获得的，而数据请求的高速度低延迟则是靠优化后的 Paxos 协议来保证的。

### 7.5.1 实体群组切分

图 7-22 是实体群组切分及其在各个数据中心分布的示意图，Megastore 将数据切割成很多细粒度的实体群组，每个实体群组会同时分布到不同的数据中心，Megastore 利用 Paxos 协议保证实体群组内数据具有 ACID 语义的强一致性，不同实体群组则提供了较弱的数据一致性。在同一个数据中心内，Megastore 使用 BigTable 来作为数据存储系统。

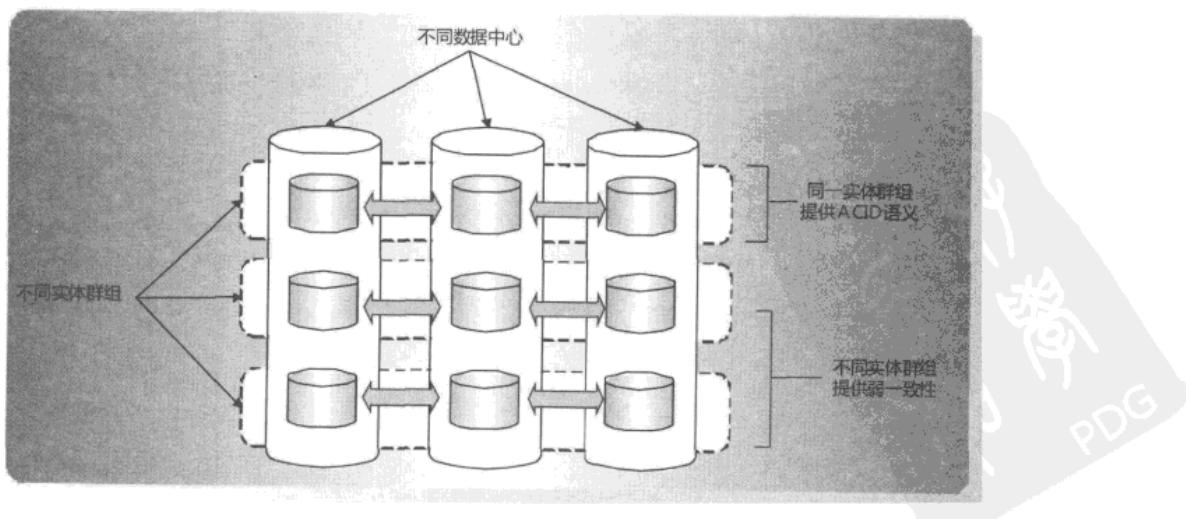


图 7-22 实体群组切分及其在各个数据中心分布的示意图

实体群组之间采用消息队列的方式完成跨群组的事务操作,图 7-23 是这种机制的说明,实体群组 1 发出一个消息,将事务追加到消息队列中,实体群组 2 接收消息队列中自己对应的消息事务并对数据做出更改。Megastore 对跨群组的事务采用了两阶段提交的方式(Two Phase Commit),这种方法相对耗时,但是由于大部分数据更新操作发生在实体群组内部,所以从系统总体效率来说问题不大。另外,Megastore 还提供了实体群组内的局部索引和全局范围的全局索引。

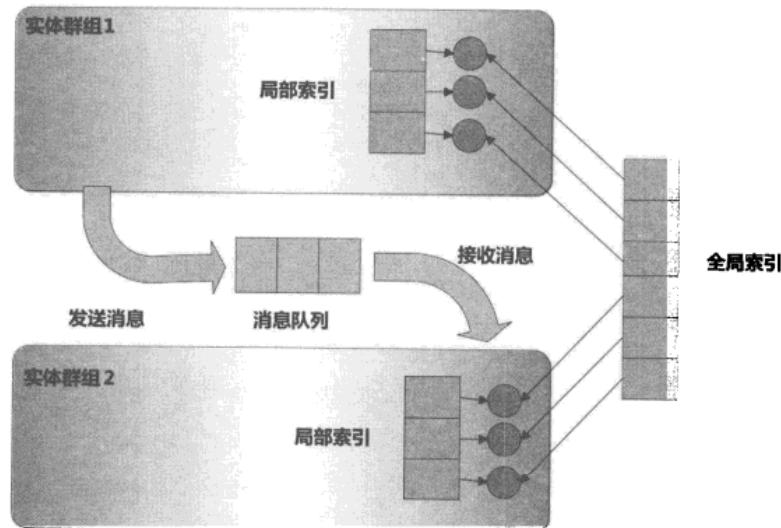


图 7-23 实体群组间通信

大部分应用都可以找到很自然的实体群组切分方法,以电子邮件应用为例,每个不同的用户账号就是一个个自然的实体群组,每个账号的邮件操作应该具有事务支持及强数据一致性,比如用户将一封邮件标记为重要,则应该立即可以看到标记结果,但是用户 A 发送给用户 B 的邮件(不同实体群组)即使在时间上有些短暂延迟,问题也不大。

### 7.5.2 数据模型

Megastore 的数据模型介于关系数据库和 NoSQL 存储系统的数据模型之间,数据模型由一个模式(Schema)定义,Schema 下面可以定义不同的表(Table),每个表可以包含不同的属性(Property),对于某个表来说,其中部分属性是表的主键(Key)。Megastore 中有两种表:实体群组主表(Root Table)和子表(Child Table),子表归属主表管辖,并且要求子表的每条记录需要有外键指向主表。

我们以一个照片分享应用作为实例来说明 Megastore 的数据模型,图 7-24 是其示意图。这个应用包含两个表格,用户表作为主表,每条记录包含用户 ID 和用户名两个属性,其中

用户 ID 是这个主表的主键。照片表是用户表的子表，包含了很多与照片有关的信息作为表的属性，其中用户 ID 和照片 ID 共同构成了表的主键，Megastore 的表属性支持可重复属性，比如例子中照片的 Tag 属性，代表用户给照片打上的文本标签，因为用户可以给同一个照片打上多个标签，所以这个属性是可重复的。照片表中的用户 ID 属性是指向主表的外键，即其属性代表的含义是相同的。主表中的一个实体及其所有子表中有外键指向这个实体的所有信息组成了一个实体群组，在这个应用中，每个用户的信息和所有归属这个用户的照片相关信息组成了一个实体群组。

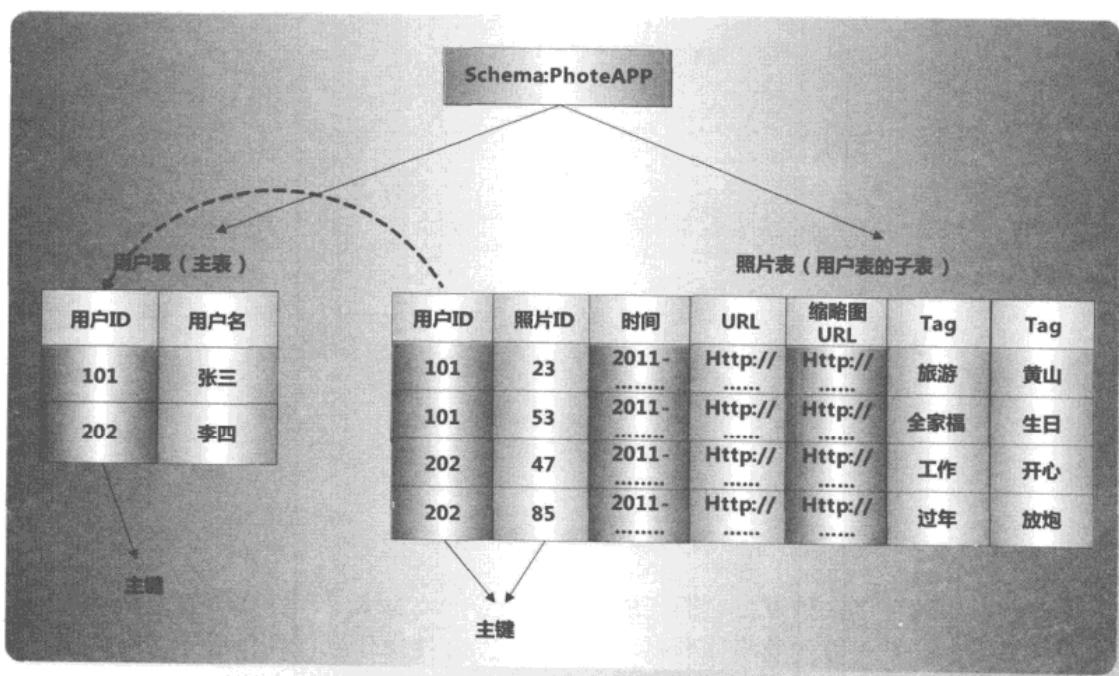


图 7-24 Megastore 数据模型示例

根据图 7-24 中两个表的示例可以看出，例子应用中包含两个实体群组，用户“张三”有两张照片，“李四”也拥有两张照片，用户“张三”和用户“李四”及其所有照片信息各自是一个实体群组。

Megastore 使用 BigTable 来存储数据，BigTable 的列属性由 Megastore 的表名和属性名共同构成，同一个实体群组尽管属于不同的表，但是在 BigTable 中是顺序存储的，这样有利于按照实体群组快速存取数据。图 7-25 展示了上面所举的例子在 BigTable 中是如何存储的。比如用户“张三”和其两张照片的数据顺序存储在 BigTable 中，用户“李四”数据的存储也是如此。

行主键	用户表.用户名	照片表.时间	照片表.URL	照片表.缩略图URL	照片表.Tag
101	张三				
101, 23		2011-.....	Http://.....	Http://.....	旅游,黄山
101, 53		2011-.....	Http://.....	Http://.....	全家福,生日
202	李四				

图 7-25 BigTable 中存储的数据格式

### 7.5.3 数据读/写与备份

Megastore 以实体群组为单位，将每份数据备份在多个数据中心，其利用优化的 Paxos 协议来保持备份数据之间的数据一致性。原始的 Paxos 协议可以保证分布式环境下的数据强一致性，但是效率太低，影响数据可用性，Megastore 通过加入中心控制策略，有效地增加了 Paxos 的执行效率，保证了数据的可用性。

通过优化的 Paxos 协议，Megastore 可以保证不论用户发出的读/写操作从哪个备份数据发起，都可以维持数据的强一致性。对于写操作来说，需要在数据中心之间进行通信来保证数据一致性，而对于读操作来说，因为写操作已经保证了数据的一致性，所以可以在任意一个数据中心保留的备份数据上进行读取。

通过优化的 Paxos 协议，Megastore 实现了数据的可用性及低延迟等存储要求的较好折中。

## 7.6 Map/Reduce 云计算模型

Map/Reduce 是 Google 公司针对海量信息处理提出的非常著名的云计算模型，目前包括 Hadoop 等众多的开源系统都采纳了这一方法，已经成为了主流云计算模型。

Map/Reduce 不仅是一种计算模型，同时也是系统提供的一个计算框架，也就是说，这个计算框架负责将应用程序的计算任务自动分配到众多机器上，并对机器失效等情况自动进行检测跟踪，应用开发者只需要关注应用任务本身要完成的工作，至于底层的分布式管理工作完全交由这个计算框架来完成，这样大大加快了应用的开发进度。

### 7.6.1 计算模型

图 7-26 展示了 Map/Reduce 计算模型的基本思路。输入数据是由 Key/Value 数值对组成的一组记录，通过 Map 操作，将其转换为新的 Key1/Value1 数值对，输入数据的每条记录可能生成多条中间数据记录。Reduce 操作则将中间数据里相同 Key 的 Value 值进行累加等处理（如图 7-26 所示中间数据记录中的网格线和斜纹线代表不同的 Key 值），生成 Key1/Value2 这种汇总的记录形式。对于应用开发者来说，只需要写好 Map 和 Reduce 两个操作的代码，其他工作都由 Map/Reduce 框架完成。

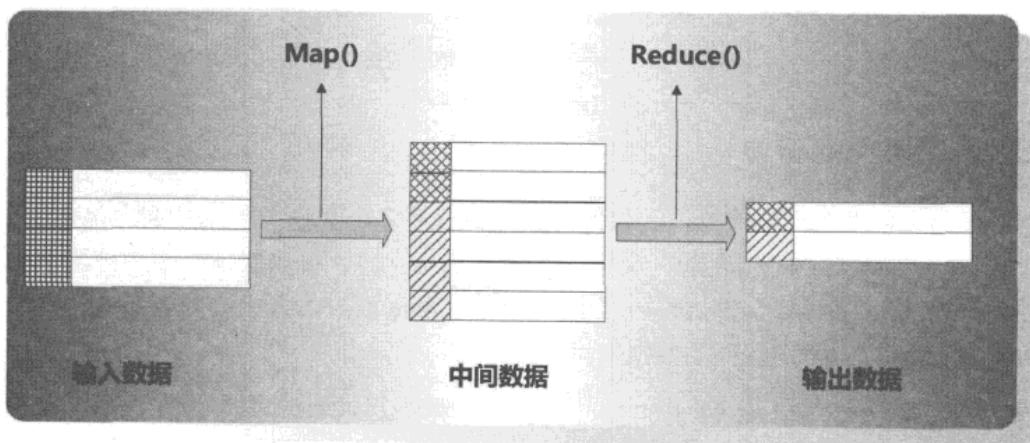


图 7-26 Map/Reduce 计算模型的基本思路

我们以一个简单的例子来说明，假设应用开发的任务是：给定一个网页集合，统计网页中的每个单词的出现次数。图 7-27 是这个简单示例对应的 Map/Reduce 任务示意图。假设网页集合由 4 个网页构成，其对应的 DocID 分别是 D1、D2、D3 和 D4，这是每个输入数据记录的 Key，输入数据记录的 Value 就是这个网页包含的单词，简单起见，我们假设总共有两个单词 w1 和 w2。Map 操作对每个输入数据记录的 Value 值进行转化，比如对于网页 D1 来说，其包含两个单词 w1 和 w2，所以将其转化为两个中间数据记录： $\langle w1, 1 \rangle$  和  $\langle w2, 1 \rangle$ ，其代表含义是 w1 出现过 1 次，w2 出现过 1 次。其他文档经过 Map 操作后也形成类似的中间结果，在中间结果里，记录的 Key 就是单词，而对应的 Value 则是其出现次数，在这里都是数值 1。Reduce 操作针对中间数据进行合并，将相同 Key 的 Value 值累加，得到最终的输出结果： $\langle w1, 2 \rangle$  和  $\langle w2, 4 \rangle$ ，即单词 w1 在这个网页集合里出现过 2 次，单词 w2 在网页集合里出现过 4 次。这样就完成了应用任务，统计出了每个单词的总共出现次数。

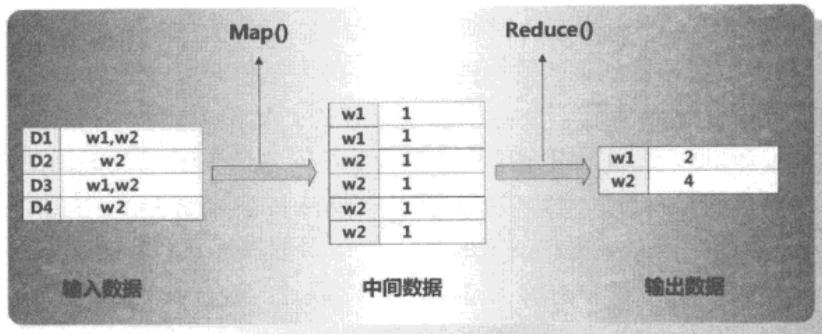


图 7-27 简单示例对应的 Map/Reduce 任务示意图

### 7.6.2 整体逻辑流程

Map/Reduce 是针对海量数据的分布式云计算模型，上述小节只是叙述其基本原理，如果面临海量数据，需要多台机器分工协作完成整个计算任务，图 7-28 展示了 Map/Reduce 整个框架的逻辑流程。

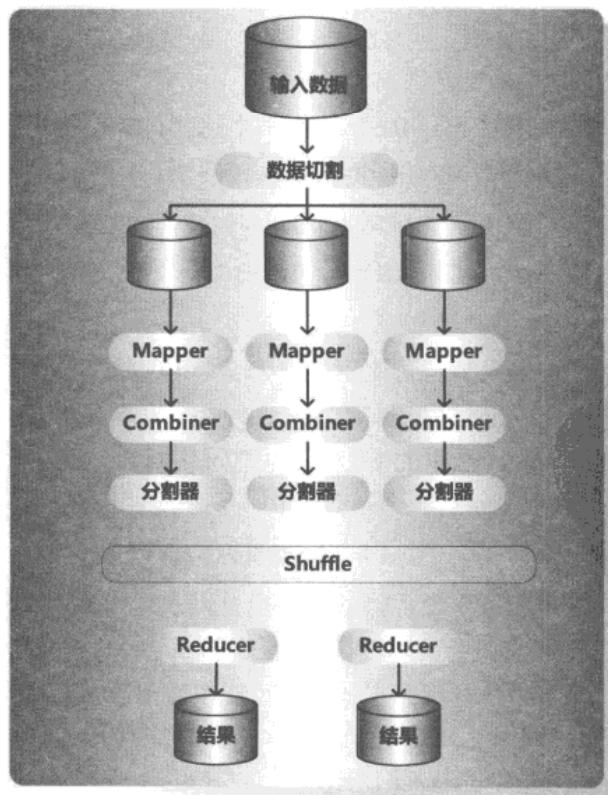


图 7-28 Map/Reduce 整个框架的逻辑流程

Map/Reduce 框架首先将应用任务的数据进行切割，把巨大的原始输入数据切割成固定大小的片段，之后将数据分发到不同的机器上，同时对每个数据片段启动一个 Mapper 任务，来对这个数据片段进行 Map 操作，数据经过 Map 操作后转换为新的 Key/Value 形式，这种中间数据存储在本地机器中。Combiner 任务的目的是对本地的中间数据相同 Key 的 Value 值先行汇总，主要目的是减少后续将中间数据通过网络传输给 Reducer 的数据传输量，减少网络负载；分割器将中间结果根据其 Key 值进行哈希取模运算（即  $\text{Hash}(\text{Key}) \bmod R$ ），将中间结果据此分成  $R$  份，这里的  $R$  是 Reducer 的数目，这么做的目的是让每个 Reducer 只负责汇总其中的一部分数据。多个 Reducer 位于其他机器上，通过网络传输将自己应该负责的各个 Mapper 产生的中间结果取到本地，并按照中间结果记录的 Key 进行排序，之后就可以对这个数据进行 Reduce 操作，将相同 Key 的记录 Value 值进行汇总计算，得到最终计算结果。

Map/Reduce 计算模型本质上是通过分而治之的思想，多机协作来对数据集合进行全局性的统计。在实际应用中，往往是多个 Map/Reduce 子任务先后串联，前面的 Map/Reduce 输出结果作为后续 Map/Reduce 任务的输入数据，类似流水线一样共同完成复杂计算任务。

### 7.6.3 应用示例

上文举了个简单例子来说明 Map/Reduce 的基本思想，本节用网页去重和建立倒排索引这两个稍微复杂些的例子来说明 Map/Reduce 的具体应用。

这里的网页去重采用简单策略，目标是将网页集合内所有内容相同的网页找出来，采取对网页内容取哈希值的方法，比如 MD5，如果两个网页的 MD5 值相同，则可以认为网页内容完全相同。

在 Map/Reduce 框架下，输入数据是网页本身，可以用网页的 URL 作为输入数据的 Key，网页内容是输入数据的 Value；Map 操作则对每个网页的内容利用 MD5 计算哈希值，以这个哈希值作为中间数据的 Key，网页的 URL 作为中间数据的 Value；Reduce 操作则将相同 Key 的中间数据对应的 URL 建立成一个链表结构，这个链表代表了具有相同网页内容哈希值的都有哪些网页。这样就完成了识别内容相同网页的任务。

对于建立倒排索引这个任务来说，输入数据也是网页，以网页的 DocID 作为输入数据的 Key，网页中出现的单词集合是输入数据的 Value；Map 操作将输入数据转化为  $\langle\text{word}, \text{DocID}\rangle$  的形式，即某个单词作为 Key，DocID 作为中间数据的 Value，其含义是单词 word 在 DocID 这个网页出现过；Reduce 操作将中间数据中相同 Key 的记录融合，得到某个单词对应的网页 ID 列表： $\langle\text{word}, \text{List}(\text{DocID})\rangle$ 。这就是单词 word 对应的倒排列表。通过这种方式就可以建立简单的倒排索引，在 Reduce 阶段也可以做些复杂操作，获得形式更为



复杂的倒排列表。

在实际的搜索引擎中，在建立索引之前往往首先对网页去重，上面两个例子可以依次串联起来，即先用 Map/Reduce 方式对网页去重，之后根据去重后的网页来建立倒排索引，将两个 Map/Reduce 任务串联起来形成更加复杂的任务。

## 7.7 咖啡因系统——Percolator

2010 年 6 月，Google 公司宣布咖啡因系统上线，这是一种新的索引更新方式，几乎可以实时对抓取到的内容进行更新并体现在搜索结果内。据说该系统上线后影响了 15% 互联网网页的搜索排名，其实从其功能来讲，咖啡因系统并没有对搜索排序因素做出改变，不会对搜索排名本身有直接影响，只不过随着更新周期的缩短，使得新更新的网页内容更快体现在搜索结果上，或许是最近更新网页新增加的内容导致网页用户查询更相关，所以造成了更新网页的排名上升，影响到内容没有更新的网页，给人的外部观感是直接影响了搜索排名。

咖啡因系统是外部代号，其对应的内部项目名称叫做 Percolator，这本质上是构建在 BigTable 上的一种与 Map/Reduce 计算方式互补的云计算模式，主要用来对搜索引擎的索引系统进行快速增量更新。在部署咖啡因系统之前，Google 搜索的索引更新是利用 Map/Reduce 机制周期性全量更新的，也就是说，每隔一段日期，将新抓取的网页和原来抓取到的网页作为一个整体，利用 Map/Reduce 重新建立一遍索引，很明显，这种方式更新周期比较长。在部署咖啡因系统之后，索引系统可以做到增量更新，对于新抓取到的网页，可以立即更新到索引系统里，新的索引更新周期比原先的方式快了大约 100 倍左右。

咖啡因系统作为一种增量更新模式，并不是 Map/Reduce 的替代品，两者各有所长，起到互补作用。如果是全局性的统计工作，还是比较适合用 Map/Reduce 来做，而对于局部性的更新则比较适合使用 Percolator 系统来处理。另外，Percolator 在 BigTable 的行事务支持的基础上实现了跨行跨表的事务支持，所以提供了对数据处理的强一致性服务，如果应用只有较弱的一致性要求，那么直接使用 BigTable 已经足够，如果有强事务要求的话，则使用 Percolator 比较合适。再次，Percolator 是对海量数据处理的计算模型，如果数据量没有达到一定量级，其实直接采用数据库系统即可满足需求。所以，咖啡因系统可以理解为针对海量数据处理的、提供强一致性支持的局部更新计算模型。这是其与所有其他系统的不同之处。

从设计特点来说，为了能够支持对海量数据的增量更新，Percolator 主要提供了以下两种功能：首先是能够对数据进行随机存取并提供对数据处理的 ACID 事务支持，另外提供

了类似于观察/通知方式的整体计算结构。

### 7.7.1 事务支持

Percolator 提供了支持 ACID 快照隔离语义的跨行跨表事务，如果一个操作涉及更改不同表的不同数据，那么这些更改要么同时生效，要么同时失效，这样保持了数据之间的一致性，这是 Percolator 能够提供增量更新的一个基础要求，在后面我们会举个具体的例子来说明 Percolator 是如何保证这种一致性的。

所谓快照隔离，是指多个用户同时读/写相同的数据时，相互之间关系很复杂，很容易发生阻塞甚至是死锁，快照隔离维护了数据的不同版本，不同的操作针对不同的数据版本进行，以此来增加并发程度并保证数据的修改一致性。Percolator 是在 BigTable 基础上实现的，前面小节我们介绍过，BigTable 在其基本存储单元（Cell）里支持多版本数据的存储，这天然适合进行快照隔离。通过快照隔离语义，Percolator 可以解决写冲突：如果同时有两个应用写同一数据，那么系统可以保证只有一个应用会成功写入。

我们以一个简单例子来说明 Percolator 是如何提供事务支持的，假设任务如下：我们有两个银行账户，“张三”的银行户头有 10 元钱，“李四”的银行账户有 2 元钱，现在需要从“张三”的账户划拨 7 元钱到“李四”的账户中，即划拨后“张三”账户的户头余额为 3 元，“李四”账户的户头余额为 9 元。这个任务往往由两个操作步骤组成：首先从“张三”账户扣除 7 元，然后给“李四”账户增加 7 元。很明显这个任务需要事务支持，否则在操作进行过程中，如果系统出错，很可能钱从“张三”账户扣除成功，但是并未追加到“李四”的账户，导致资金总额出错。而事务支持可以保证：要么两个操作都成功，要么两个操作都失败，不论如何，能够保证两者资金总额是 12 元，即数据是一致的。

针对这个任务，我们看看 Percolator 是如何操作的，BigTable 提供了对数据行的事务支持，Percolator 充分利用这一点，为表中每列数据增加管理数据，其中 Column:Lock 和 Column:Write 用来进行事务支持，另外的管理数据是为了支持订阅/通知体系结构的。要完成上述任务，需要依次执行以下几个步骤。

图 7-29 是任务执行前的某个 BigTable 表格的初始状态，表中每个存储单元（Cell）可以存储不同时间戳的多个数据版本，“5:10 元”代表时间戳为 5 的时候，用户“张三”对应的账户金额为 10 元。从表中可以看出对于存储账户金额的数据 bal:data 列，Percolator 对应增加了两列 bal:lock 和 bal:write，其中 bal:lock 列是用来存储锁的管理数据列，而 bal:write 列则指出了哪个时间戳的数据是当前可用数据，比如“6: data@5”即指明了 bal:data 列的时间戳为 5 的数据是可用数据。

图 7-30 展示了任务的第一步操作，Percolator 从系统获取新的时间戳 7，并在 bal:lock



写入锁标记 Primary，说明这个锁是主锁，同时将“张三”的新的金额 3 元写入数据列 bal:data，因为 BigTable 支持基于行的事务，所以这些操作可以保证其数据一致性。

Key	bal:data	bal:lock	bal:write
张三	6 : 5: 10元	6 : 5:	6 : data@5: 5:
李四	6 : 5: 2元	6 : 5:	6 : data@5: 5:

图 7-29 任务执行前的某个 BigTable 表格初始状态

Key	bal:data	bal:lock	bal:write
张三	7: 3元 6 : 5: 10元	7 : Primary 6 : 5:	7: 6 : data@5: 5:
李四	6 : 5: 2元	6 : 5:	6 : data@5: 5:

图 7-30 任务第 1 步：扣除“张三”的金额

图 7-31 展示了任务的第 2 步操作，Percolator 在 bal:data 列写入“李四”的新的账户金额，同时在 bal:lock 列写入二级锁“Primary@张三.bal”，二级锁指出了主锁所在位置，即“张三”这一行的 bal.lock 列中。之所以在这里写入如此内容的二级锁，主要是防止任务执行失败时，系统能够找到主锁的位置并清除掉未完成任务的主锁。

Key	bal:data	bal:lock	bal:write
张三	7: 3元 6 : 5: 10元	7 : Primary 6 : 5:	7: 6 : data@5: 5:
李四	7: 9元 6 : 5: 2元	7 : Primary@张三.bal 6 : 5:	7: 6 : data@5: 5:

图 7-31 任务第 2 步：增加“李四”的金额

当某一行加锁的时候，其他事务对该行的读/写操作都被暂时阻塞，直到锁被释放才可以继续。图 7-32 指出了当账户金额都得到更改后，这个任务作为一个事务，处于可提交状态。此时系统获取新的时间戳，并清理掉主锁，同时在 bal:write 列指出新的可用数据为 bal:data 中时间戳为 7 的数据，此时所有新的读取操作会根据 bal:write 列找到账户“张三”的最新数据 3 元。

同样地，账户“李四”的二级锁被释放，同时在 bal:write 列指出新的可用数据所在位置：bal:data 的时间戳为 7 的数据（参考图 7-33）。

Key	bal:data	bal:lock	bal:write
张三	8: 7: 3元	8: 7:	8: data@7
	6: 5: 10元	6: 5:	6: data@5
李四	7: 9元	/ : Primary@张三.bal	7:
	6: 5: 2元	6: 5:	6: data@5

图 7-32 释放主锁，指出可用数据

Key	bal:data	bal:lock	bal:write
张三	8: 7: 3元	8: 7:	8: data@7
	6: 5: 10元	6: 5:	6: data@5
李四	8: 7: 9元	8: 7:	8: data@7
	6: 5: 2元	6: 5:	6: data@5

图 7-33 释放二级锁，指出可用数据

通过以上几个步骤，Percolator 往 BigTable 数据行加入管理数据，就实现了跨行跨表的事务支持。上面所举例子非常简单，在实际系统中，往往是对索引系统的数据进行事务支持，这样就可以实现增量更新的功能。

### 7.7.2 观察/通知体系结构

Percolater 采用了观察/通知的机制来将应用程序串接起来形成一个整体，图 7-34 是其运行流程示意图。在 BigTable 的每个子表服务器上，Percolater 都部署了一个 Percolater 控制器（Percolater Worker），不同的应用在控制器登记两类信息：哪个应用程序观察子表的哪些列，在这里每个应用程序被称做一个观察者。在图示中，有两个观察者，观察者 1 关注子表的第 1 列和第 2 列，观察者 2 关注子表的第 3 列。Percolater 控制器不断扫描子表的列内容，如果发现被观察的某列数据做出更改，则通知观察这列数据的观察者，观察者执行相应的程序逻辑操作，并将更新的内容写入子表中，新写入的数据可能会触发其他观察



者启动执行。Percolator 就是通过这种观察/通知的机制将完成一项任务所需的所有步骤串接起来，有点类似于多米诺骨牌，一一依次触发。在图 7-34 中，当 Percolator 控制器发现第 2 列第 2 行被写入数据 X 后，通知观察者 1，观察者 1 执行对应的程序，执行结束后在子表的第 3 行第 3 列写入数据 Y，这触发了观察者 2 的执行。

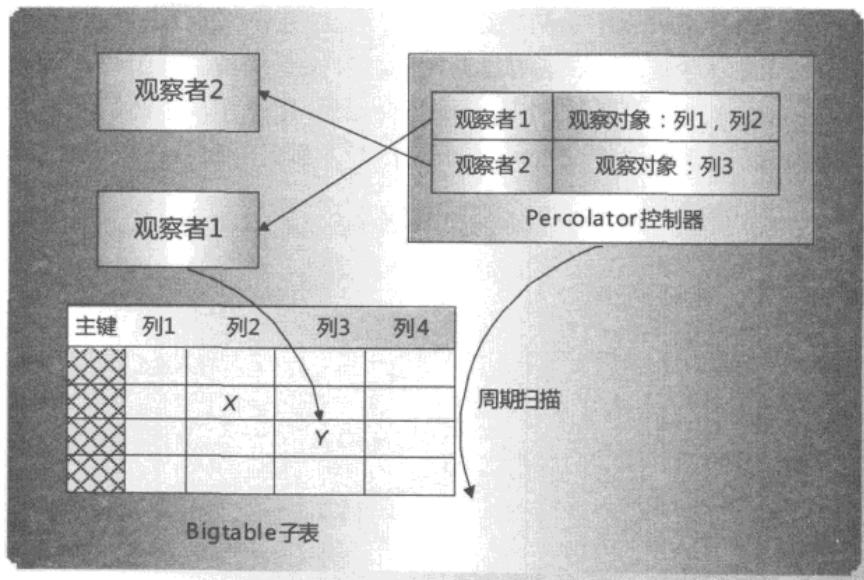


图 7-34 观察/通知体系结构运行流程示意图

Percolator 主要用来对索引内容进行增量更新，当有一批新增网页被抓取后，系统通过 Map/Reduce 方式将新增网页加载进入 BigTable 对应表中，Percolater 的观察/通知机制开始启动，这触发了网页内容处理观察者，将网页内容抽取出来或者是处理网页中包含的链接，网页内容处理观察者将其操作结果写入 BigTable 对应表格，则会触发网页去重观察者，如此这般，一个任务触发另外一个任务，通过大约 10 个子任务，就可以完成新增网页的索引更新。

## 7.8 Pregel 图计算模型

随着 Web 2.0 的兴起，大规模图计算越来越重要，比如社交关系图的计算等，即使是搜索引擎服务本身，前述章节讲述的链接分析也是一个典型的图计算实例。对于非常巨大的计算规模，如何在分布式环境下来对图进行计算成为具有挑战性的问题，尽管通过一系列串接的 Map/Reduce 任务也可以完成图计算的功能，但是因为图计算本身的特点，使用 Map/Reduce 进行图计算需要很高的机器之间的通信开销，而且计算模型也非常复杂。为了

减轻这个问题，Google 提出了 Pregel 计算模型，专门用来解决大规模分布式图计算的问题。

PageRank 需要迭代计算，其实很多图算法都有这个特点，所以 Pregel 将此作为重要特点引入计算模型中，一次迭代计算被称做一个超级步（SuperSteps），系统从一个超级步迈向下一个超级步，直到达到算法的终止条件（参考图 7-35）。Pregel 以图节点为计算中心，在超级步  $S$  中，每个图节点可以汇总从超级步  $S-1$  中其他节点传递过来的消息，改变自身的状态，并向其他节点发送消息，这些消息会在超级步  $S+1$  中被其他节点接收并做出处理。用户只需要自定义一个针对图节点的计算函数  $F(\text{vertex})$ ，用来实现上述的图节点计算功能，至于其他的任务比如任务分配、任务管理、系统容错等都交由 Pregel 系统来实现。

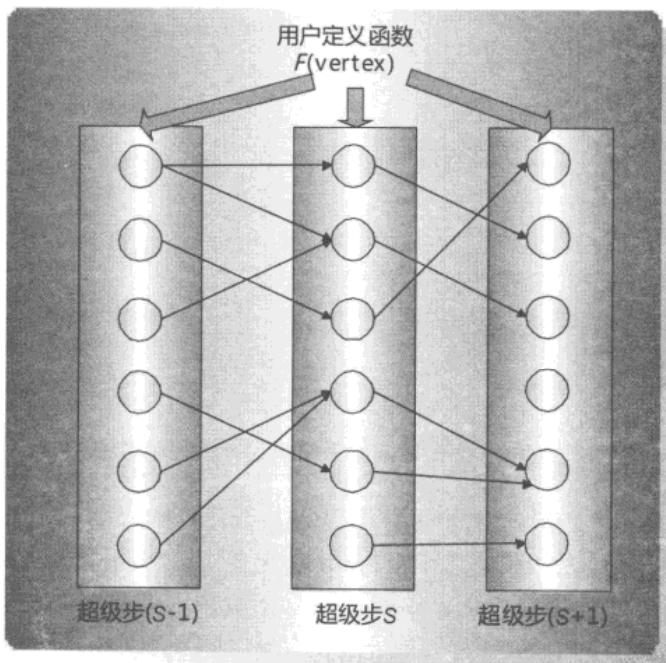


图 7-35 Pregel 的计算模型

每个节点有两种状态：活跃与不活跃。刚开始计算的时候每个节点都处于活跃状态，随着计算的进行，某些节点完成计算任务转为不活跃状态，如果处于不活跃状态的节点接收到新的消息，则再次转为活跃，如果图中所有节点都处于不活跃状态，则计算任务完成，Pregel 输出计算结果。

Pregel 是在多台机器协同下完成图计算任务的，与很多 Google 云存储与云计算模型一样，Pregel 也采用了主从结构来实现整体功能，图 7-36 是其物理结构图。其中一台服务器充当主控服务器，负责整个图结构的任务切分，将其切割成子图，并把任务分配给众多的工作服务器，工作服务器负责计算被分配给自己的子图，主控服务器还命令工作服务器进



行每一个超级步的计算，并收集计算结果。

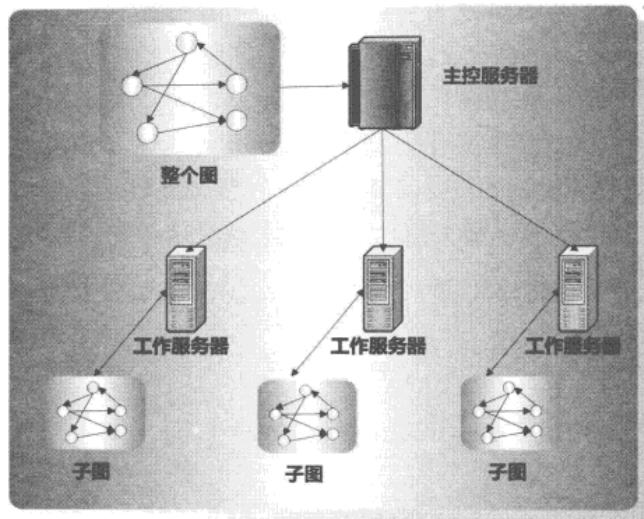


图 7-36 Pregel 的分布协作关系

我们以一个具体的计算任务来作为 Pregel 图计算模型的实例，这个任务要求将图中节点的最大值传播给图中所有其他节点，图 7-37 是其示意图。图中的实线箭头表明了图的链接关系，而图中节点内的数值代表了节点的当前数值，图中虚线代表了不同超级步之间的消息传递关系，同时，带有斜纹标记的图节点是不活跃节点。

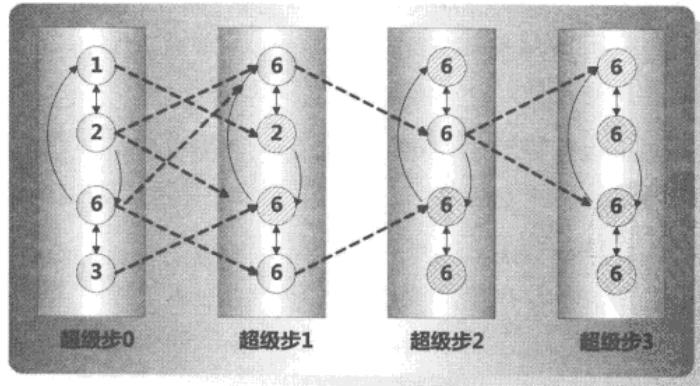


图 7-37 Pregel 传播最大值

从图 7-37 中可以看出，数值 6 是图中最大值，在超级步 0 中，所有节点都是活跃的，系统执行用户函数  $F(\text{vertex})$ ：节点将自身的数值通过链接关系传播出去，接收到消息的节点选择其中的最大值，并和自身数值比较，如果比自身数值大，则更新为新的数值，如果不比自身的数值大，则转为不活跃状态。

在超级步 0 中，每个节点都将自身的数值通过链接传播出去，系统进入超级步 1，执行  $F(\text{vertex})$  函数，第 1 行和第 4 行的节点因为接收到了比自身数值大的数值，所以更新为新的数值 6。第 2 行和第 3 行的节点没有接收到比自身数值大的数，所以转为不活跃状态。在执行完函数后，处于活跃状态的节点再次发出消息，系统进入超级步 2，第 2 行节点本来处于不活跃状态，因为接收到新消息，所以更新数值到 6，重新处于活跃状态，而其他节点都进入了不活跃状态。Pregel 进入超级步 3，所有节点处于不活跃状态，所以计算任务结束，这样就完成了整个任务，最大数值通过 4 个超级步传递给图中所有其他节点。

从上述描述可以看出，Pregel 是一个消息驱动的、以图节点为中心的并且是适合迭代计算的分布式图计算模型。

## 7.9 Dynamo 云存储系统

Dynamo 是亚马逊公司开发的云存储系统，主要用来存储电子商务网站购物车相关数据。Dynamo 在设计哲学上与 Google 的主从结构大异其趣，在整个系统内不存在作为管理控制功能的主控节点，完全采取对等网络（Peer to Peer）的方法，每个节点功能相同，都同时承担数据存储和部分管理功能。Google 范式的单主控节点和 Dynamo 范式的完全对等网络可以认为是海量存储系统的两个极端，有些其他云存储模式介于两者之间。

Dynamo 系统与很多其他云存储系统一样，采用大量商用 PC 构建存储网络，以数据的最终一致性作为代价，来换取系统的高可用性和高容错性。Dynamo 的数据模型相对简单，是很直观的 Key/Value 模式，即每条数据都是由主键 Key 和数据 Value 构成的，不像 BigTable 等系统一样在其数据区有列属性。Dynamo 设计原则如下。

1. **易扩展性。**如果数据量增加，可以通过简单增加机器解决问题，同时在增加机器时不影响现有服务的正常运转，这一点对于云存储系统来说应该说是个标准需求。
2. **对等性与去中心控制节点。**系统内每个节点功能完全对等，都同时承担数据存储和部分管理功能，不存在单独承担管理功能的中心主控节点。这一点 Dynamo 可谓独出心裁，一般的云存储系统都还是要一些单独承载管理功能的主控节点。
3. **机器异构性。**所谓机器异构性，指的是在一个复杂系统内，使用大量机器，但是每台机器性能大不相同，有相对老些的机器，其存储和计算资源较少，也有新型机器，拥有大量存储空间和多核 CPU 等资源，系统应该让资源少的机器承担较少任务，而让资源富余的机器承载较多任务。一般的云存储系统通过负载均衡来达到这个目的，而 Dynamo 则将这个作为设计原则并体现在其设计思路中。

Dynamo 系统作为完全采取 P2P 架构的云存储系统，在技术上展现了很多独特之处，

虽然这种设计方案是否合理在业界存在大量争议，而且目前来看，这种方案能否经得起时间考验颇值得怀疑，但是系统里使用的一些技术思路还是非常值得介绍的。

### 7.9.1 数据划分算法 ( Partitioning Algorithm )

为了达到易扩展性，系统要能够根据数据情况和机器情况，动态地在机器之间分配数据。对于存在主控节点的分布式系统来说，一般由主控节点来决定将海量的数据进行划分，并分配到数据存储节点中。但是对于 Dynamo 这种节点功能完全对等的系统来说，如何决定将哪些数据放置在哪台机器呢？Dynamo 使用了一致性哈希技术来进行数据的划分和分配。

图 7-38 是一致性哈希算法的示意图。Dynamo 将所有主键的哈希数值空间组成一个首尾相接的环状序列，对于每台机器，随机赋予其一个哈希值，这样不同的机器就成了环状序列中的不同节点（图 7-38 中环上的 4 个大圆即代表不同的机器），而这台机器则负责存储落在一段哈希空间内的数据。如果一个数据的哈希值落入某个区段，则顺时针进行查找，找到第 1 台机器，这台机器就负责这个数据的存储，之后的相关存取操作及冗余备份等操作也由其负责。通过这种方式，就自动实现了数据在不同机器之间的动态分配。

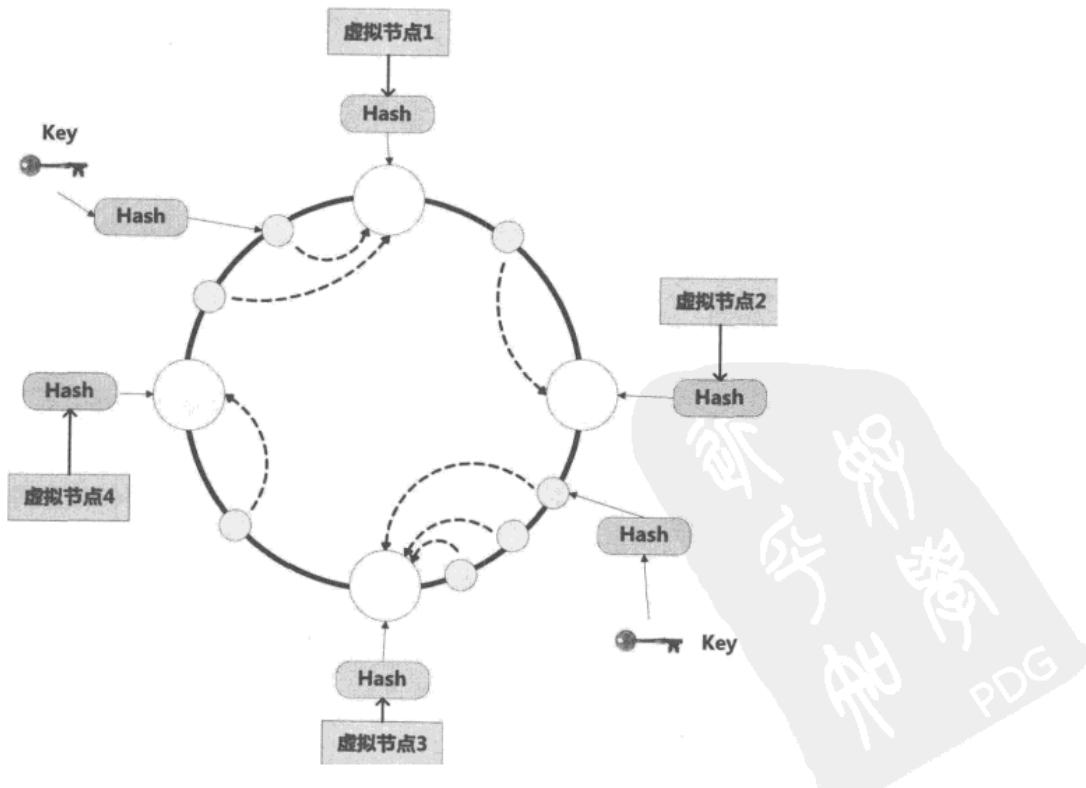


图 7-38 一致性哈希算法

一致性哈希算法有个明显的优点：如果新加入一台机器或者某台机器失效无法提供服务，其影响的范围也只限于其本身负责的那段哈希数值空间及和其顺时针顺序相邻的节点，对于网络内其他节点没有影响，即其影响是局部的，可扩展性非常好。

为了达到机器异构性，Dynamo 还引入了虚拟节点的思想，即将一台物理机器的计算和存储资源分割成若干虚拟机，每个虚拟机作为插入环状序列的服务节点。这样的话，对于高配置的机器，可以划分成数量较多的虚拟机，而对于比较老的机器，则划分较少数量的虚拟机，如此每台物理机器的任务负载量和其资源能力相匹配，同时充分利用了一些配置较低的机器资源。

### 7.9.2 数据备份 (Replication)

大量 PC 构成的云存储系统，数据冗余备份是必须考虑的，因为机器宕机是个常态，为了保证数据总是可用，必须对存储数据进行多个备份。对于有主控节点的分布式存储系统，这个职责往往由主控节点负责，但是对于完全对等网络，则由数据存储节点来接管这一功能。

上节讲过，每个虚拟节点负责存储一部分哈希空间的数据，同时这个节点也负责将这些数据备份到环中顺时针方向后续的  $N-1$  个节点中，这样每份数据就在系统中存在  $N$  份数据，达到冗余存储的目的。

### 7.9.3 数据读/写

每个虚拟节点负责其管辖范围内的数据读/写操作，因为系统中存在多个备份数据，所以对于写操作，要保证数据内容的一致性，而对于读操作，也要保证能够读取到最新的更新数据。

为了达到上述目的，Dynamo 采取了类似于投票机制的数据一致协议，来达成一种可根据具体需求动态配置的系统，来保证系统总是可以读取到最新更新的数据。假设系统对于每份数据共存储  $N$  份，数值  $R$  代表一次成功的读数据操作要求至少有  $R$  份数据成功读取，而数值  $W$  代表一次成功的更新操作要求至少有  $W$  份数据写入成功，如果满足：

$$R+W>N$$

则满足数据一致协议。

我们以图 7-39 的具体例子来说明这种数据一致协议，图 7-39 中， $N$  为 3，即系统中每个数据保留 3 个备份。 $R$  设置为 2，含义是在读取这个数据的时候，至少要有 2 个备份数据读取成功，此次读取才能被认为是有效读取。而  $W$  设置为 2，含义是在写入数据的时候会

同时向 3 个备份写入，至少要有两份数据写入成功，才能认为这是一次有效写入操作。因为  $R+W>N$ ，所以这种配置是符合数据一致协议的。从图 7-39 中可以看出，如果  $R+W>N$ ，则读取操作和写入操作成功的数据一定会有交集（图 7-39 中是数据备份 B），这样就可以保证一定能够读取到最新的更新数据，数据的一致性得到了保证。

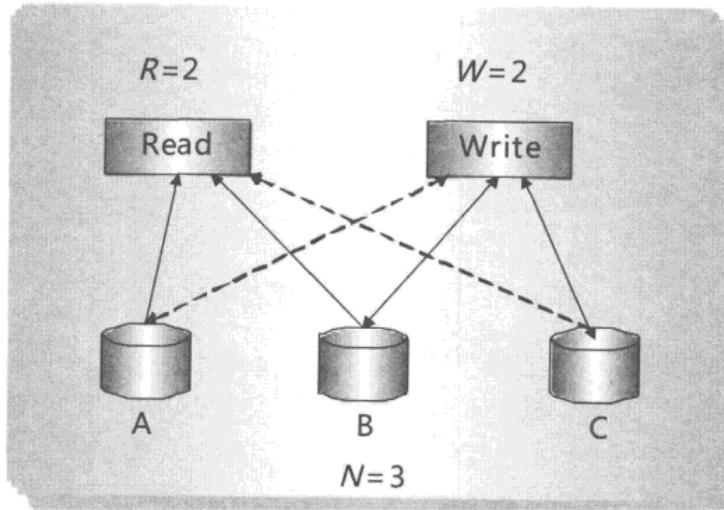


图 7-39 数据一致协议

在满足数据一致协议的前提下， $R$  或者  $W$  设置得越大，则系统延迟越大，因为这取决于最慢的那份备份数据的响应时间。根据系统的需求，可以配置不同的参数组合，比如图 7-39 中可以配置为  $W=1$ ,  $R=3$ ，含义是只要成功写入 1 个备份就算成功，而读取则需要 3 个备份都成功才算有效读取，这种配置明显适合要求写入速度较快，而对读取速度要求不高的应用场合。

数据一致协议可以允许应用系统根据实际需要来配置相应的  $R$ 、 $W$ 、 $N$  参数，增加了云存储系统的灵活性。

#### 7.9.4 数据版本控制

分布式存储系统对于某个数据存在多个备份，数据写入时要尽量保证备份数据同时获得更新，Dynamo 对此采取了数据的最终一致，也就是说，在一定的时间窗口内，对数据的更新会传播到所有备份中，但是在时间窗口内，如果客户端有读取数据的操作，尽管上一小节保证一定可以读取到最新的数据，但还是可能会同时读取到旧版本的数据，Dynamo 保存了数据的新旧等多个版本信息，并采用向量时钟（Vector Clock）技术来进行版本控制，如果发现冲突版本的存在，则交由应用自己进行处理。

一个向量时钟是由一系列（机器节点编号：计数器）构成的，我们以图 7-40 中的例子来说明如何利用向量时钟来进行版本控制，图中所有操作都是针对某个特定的记录 X 的。

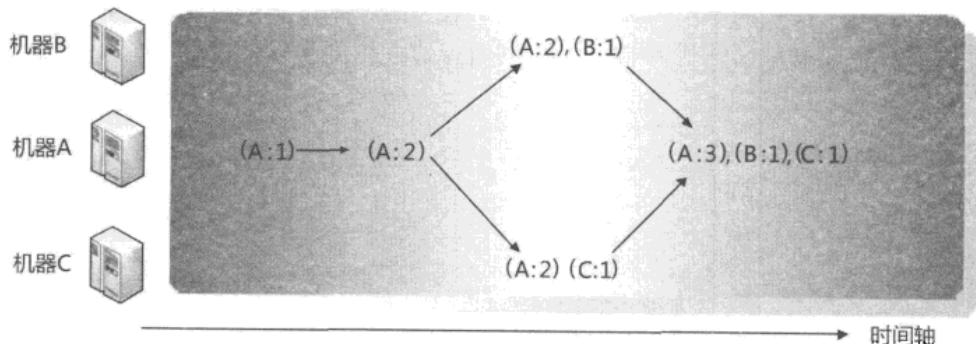


图 7-40 利用向量时钟进行版本控制

假设客户端 W 对记录 X 有个更新请求，而这个请求由机器 A 来负责执行，则形成了初始的向量时钟(A:1)，之后客户端 W 又有对记录 X 的更新请求，这个请求仍然是由机器 A 来执行的，于是向量时钟变为(A:2)，(A:1)和(A:2)存在因果关系，(A:2)是最新版本的数据，而(A:1)可以被废弃。如果在机器 B 上存在一个记录 X 的副本，同一客户端 W 继续要求更新记录 X，此次的请求由机器 B 来负责执行，则机器 B 将更新写入副本中，向量时钟更新为(A:2)，(B:1)。如果与此同时，另外一个客户端 T 请求更新记录 X，而这次的请求由机器 C 负责，则在机器 C 上形成了向量时钟(A:2)，(C:1)，假如此时有客户端读取记录 X，会发现在机器 B 和机器 C 上存在两个有冲突的版本，Dynamo 的设计思路是由客户端自己解决这种冲突。如果客户端成功解决冲突，向量时钟转换为(A:2)，(B:1)，(C:1)，之后再次有对记录 X 的更新请求，并且是机器 A 负责执行，则向量时钟进化为(A:3)，(B:1)，(C:1)。

Dynamo 就是采用这种向量时钟控制版本并发现版本冲突，而由客户端解决版本冲突来保持数据的一致性的。

Dynamo 在实现云存储系统时采取或引入一系列的新技术，除了以上介绍的内容外，还有其他技术，比如利用 Gossip 协议来发现新加入的机器及发现失效机器等，在此不再赘述，有兴趣读者可以参考原始文献。

## 7.10 PNUTS 云存储系统

PNUTS 是 Yahoo 公司构建的提供在线数据服务的云存储系统，与其他的海量云存储系统相似，PNUTS 采取了弱一致性模型，以这种宽松的一致性模型为代价，换取系统更好的可扩展性、高可用性及强容错性。

之所以讲解 PNUTS 系统，是因为这个云存储平台有自己的特点，具有代表性，具体而言，PNUTS 在以下几方面有其特色。

1. 这个云存储平台支持在线实时请求的响应，前面介绍的 Google 的 BigTable 等云存储架构从其设计原理来说，更适合对实时性要求不高的后台计算。
2. PNUTS 支持多数据中心的分布式存储和数据备份与同步。
3. 很多云存储系统对于数据更新，采取先写入系统 Log 文件，事后回放（Replay）的方式来保证数据操作的容错性。PNUTS 则采取了消息代理的机制来保证这一点，虽然从本质上说也类似于 Log 回放机制，但是其表现形式并不相同。
4. 对于数据的一致性，PNUTS 采取了以记录为单位的时间轴一致方法。

PNUTS 的数据模型类似于 BigTable 的数据模型，以行为一个数据单位，即一条记录，每条记录有不同的属性作为列，同时是模式自由的列属性方式，但是区别是 PNUTS 数据记录是二维表，不保留数据的不同时间版本信息。

### 7.10.1 PNUTS 整体架构

PNUTS 支持数据的多数据中心部署，每个数据中心被称做一个区域（Region），每个区域所部署的系统都是完全相同的，每条记录在每个区域都有相应的备份（参考图 7-41）。每个区域内主要包含 3 个基本单元：子表控制器、数据路由器和存储单元，其中存储单元负责实际数据的存储，其他两个部分起到数据管理的作用，消息代理则横跨多个区域，主要负责数据在不同区域的更新与同步。

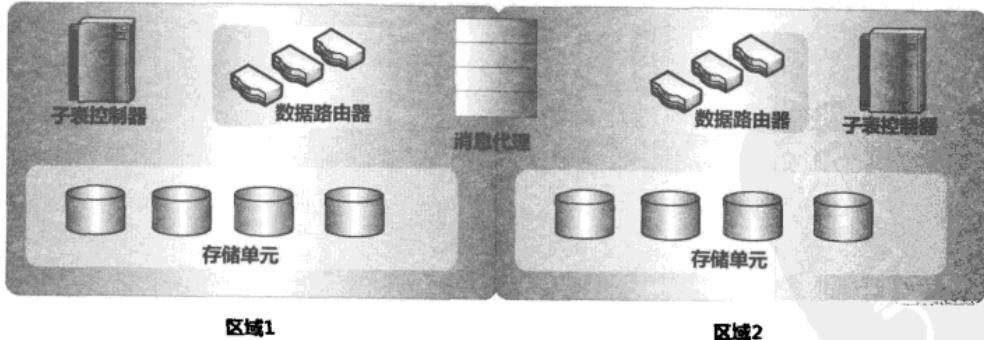


图 7-41 PNUTS 整体架构图

### 7.10.2 存储单元

存储单元负责实际数据的存储，对于每个二维数据表格，若干条记录组成一个子表

(Tablet)，每个存储单元负责存储几百个不同的子表，在具体某个区域内，只保留一份子表。数据的冗余存储是通过不同区域备份来实现的，即每条记录在每个区域都有一个备份。

PNUTS 整体架构对于子表划分，有两种不同的方式：有序划分和哈希划分。所谓有序划分，就是按照记录主键排序，然后将连续的一段记录划分成一个子表，每个子表内的记录主键仍然是有序的。对于这种类型的子表，PNUTS 采用 MySQL 数据库的方式存储。图 7-42 是这种划分方式的示意图，比如子表 2 存储了主键范围在 banana 到 grape 内的记录，其他子表含义类似。

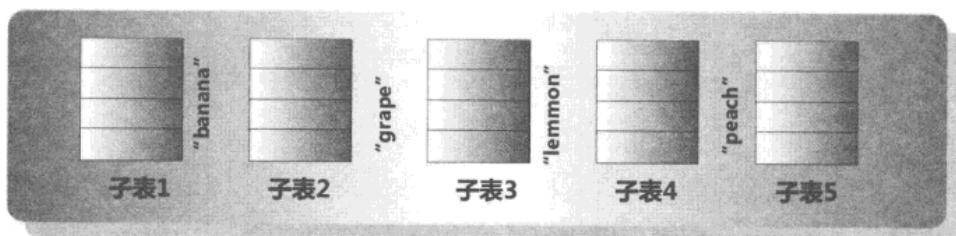


图 7-42 有序划分子表的示意图

哈希划分基本思路与 Dynamo 系统的数据划分方式类似，即对所有记录的主键进行哈希计算，将所有哈希值看做一个闭环，将这个闭环切割，形成不同的子表（参见图 7-43）。

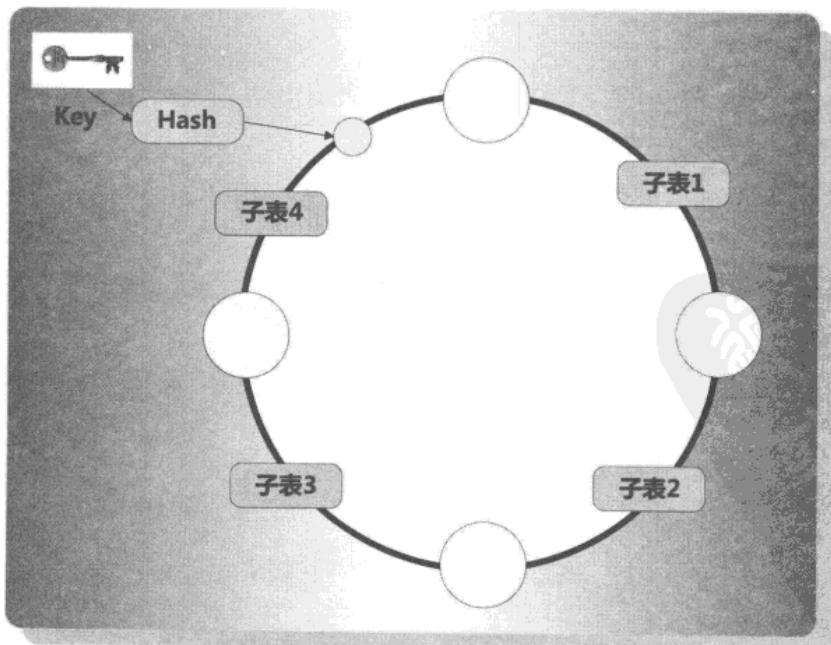


图 7-43 哈希划分子表



### 7.10.3 子表控制器与数据路由器

数据路由器负责查找某条记录所在存储单元的位置，当客户端程序要对某个记录进行读/写时，会询问数据路由器应该和哪个存储单元通信，数据路由器在内存保留记录主键所在存储单元的映射表，通过查找映射表，告知客户端存储单元地址。之后客户端程序和存储单元联系进行数据读取操作（参考图 7-44）。

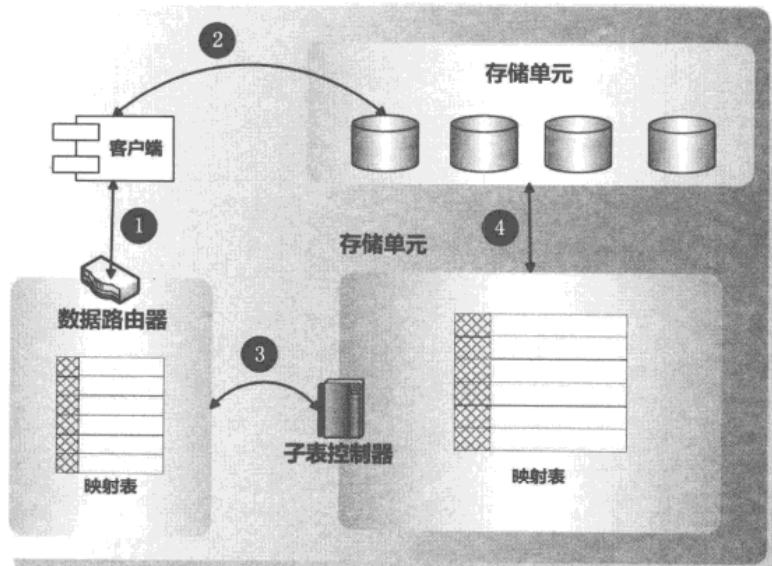


图 7-44 子表控制器与数据路由器

数据路由器的映射表来自于子表控制器，子表控制器负责存储单元的管理，比如负载均衡或者对子表进行分裂等操作，之后会修改对应的映射表，数据路由器周期性地从子表控制器获得更新后的映射表。如果数据路由器的数据没有及时与子表控制器的映射表保持一致，某个客户端此时有读取请求，数据路由器会返回一个过期的存储单元地址，之后客户端和存储单元联系，存储单元会报错，此时数据路由器知道自己的数据过期，会从子表控制器处更新新的映射表。

### 7.10.4 雅虎消息代理

雅虎消息代理负责数据的更新与同步，来保持记录数据的一致性。雅虎消息代理采取发布/订阅的消息队列方式，并且横跨不同数据中心，以保持不同数据中心的数据一致（参考图 7-45）。

前文讲过，每条记录在同一个区域内没有副本，而在其他数据中心各自有一份副本存

在，图 7-45 中假设有 3 个数据中心，所以每条记录在不同数据中心共有 3 个备份，其中某个记录作为主记录，其他两个作为备份记录，所有对记录的更新操作都由主记录来完成，主记录在更新数据后，即向雅虎消息代理发布一条数据更新信息，发布成功即可认为数据更新完成，之后由雅虎消息代理负责将同样的更新操作体现到其他两个备份记录上，雅虎消息队列可以保证这种更新一定可以正确完成，这样就实现了数据的一致性。如果某个客户端对备份记录发出更新数据的请求，备份记录会将这个更新操作引导到主记录，由主记录来完成这种更新操作。

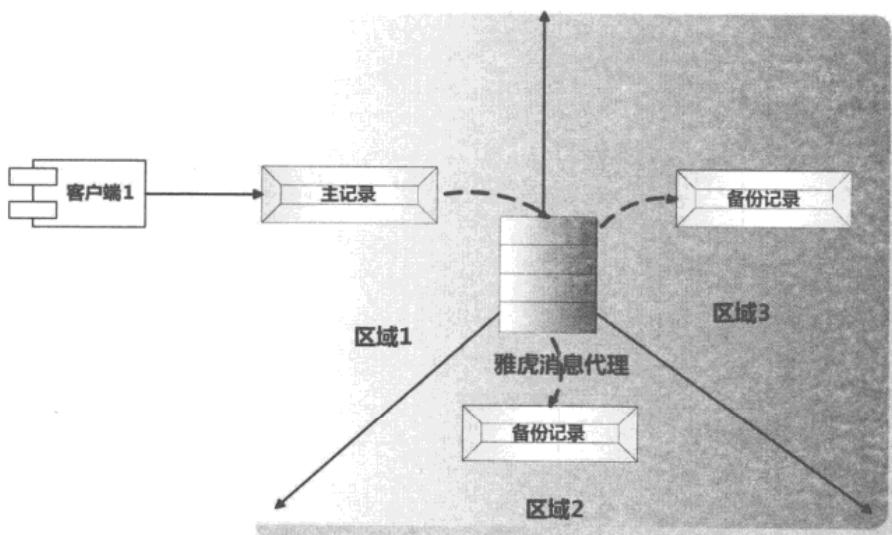


图 7-45 数据更新与同步

从上述过程可以看出，只要主记录完成更新操作并发布消息成功，即可认为更新操作成功完成，之后的数据一致性由雅虎消息代理来保证，这是为何 PNUTS 可以实时响应用户请求的关键。

### 7.10.5 数据一致性

PNUTS 采取了记录级别的时间轴一致性，前文讲述过，所有更新操作都由主记录来完成，雅虎消息代理负责按照相同的更新顺序来更新所有其他备份记录。

图 7-46 展示了这种时间轴一致性的原理，随着系统时间向前推进，记录不断被更新，系统会记载当前记录的版本信息，随着更新的不断进行，记录的版本号持续增加，在某个时间点，记录只保留当前版本的数据，但是由于备份记录和主记录的更新存在时间差，可能整个 PNUTS 系统中存在多个版本的记录，不同版本的数据由版本号可以区分。

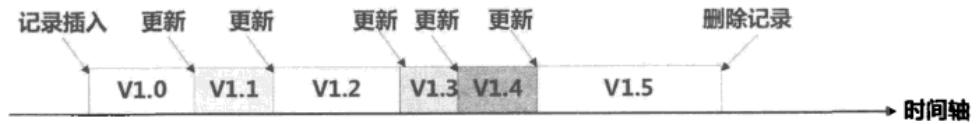


图 7-46 时间轴一致性原理

PNUTS 提供了多种读取 API 来满足不同应用的不同需求，可以指定读取最新版本的记录，但这往往需要较长的响应时间，也可以指定任意版本读取，这会加快系统响应时间，但是只有对数据一致性不敏感的应用才能这么做，对这些应用来说，即使数据有些老旧也不会有太大影响。此外，应用也可以指定读取记录版本号，只要记录本身的版本号大于指定的版本号就可以满足要求（参考图 7-47）。PNUTS 提供的这种不同 API 给整个系统提供了灵活性。

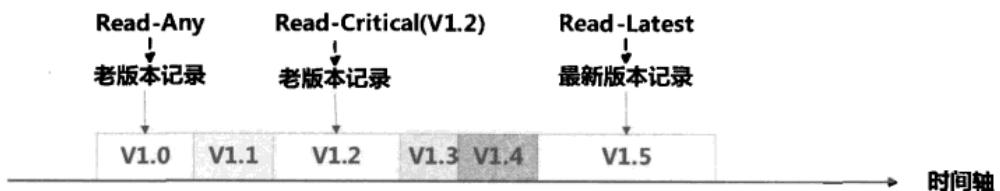


图 7-47 PNUTS 支持灵活读取操作

## 7.11 HayStack 存储系统

HayStack 是 Facebook 公司设计开发的一种对象存储系统，这里的对象主要指用户上传的图片数据。作为一个社交平台，Facebook 用户需要设置头像或者与朋友们分享图片，所以 Facebook 面临海量图片的存储和读取更新等任务，目前 Facebook 存储了超过 2 600 亿的图片数据，而这些数据的存取与管理都是靠 HayStack 系统完成的。

大型商业互联网公司对于类似于 HayStack 的这种对象存储系统有很大的需求，这里的对象往往指满足一定性质的媒体类型，类似于图片数据的存储有其自身特点，典型的特征是：一次写入、多次读取、从不更改、很少删除。很多其他类型的数据也有此种特点，比如邮件附件、视频文件及音频文件等，一般将这种数据称为 BLOB（Binary Large Object）数据，对应的存储可以称为 BLOB 存储系统。因为其特点是读多改少，所以在设计这种存储系统的时候，保证读取效率是需要重点考虑的要素。目前国内的淘宝和腾讯等大型互联网公司也独立开发了类似的存储系统，其实现思路应该与 HayStack 系统差异不大。

为了减少系统读取压力，对于海量的静态数据请求，一般会考虑使用 CDN 来缓存热门请求，对于这样大量的请求由 CDN 系统就可以满足。HayStack 存储系统的初衷是作为 CDN

系统的补充，即热门请求由 CDN 系统负责，长尾的图片数据请求由 HayStack 系统负责。

由于图片数据请求具有读多改少的特点，所以如何优化图片的读取速度是 HayStack 系统的设计核心。一般读取一张图片需要有两次磁盘读操作，首先从磁盘中获得图片的元数据，根据元数据从磁盘中读出图片内容。为了增加读取速度，HayStack 系统的核心目标是减少读取磁盘的次数，将所有图片内容放入内存显然是不太可能的，所以可以考虑将图片的元数据放入内存中，因为相比图片本身内容的数据量来说，图片的元数据小很多，将所有图片的元数据放入内存理论上是可行的，这样就可以将两次磁盘操作减少为一次磁盘操作。

但是实际上，尽管每个图片的元数据量不大，由于图片数量太多，导致内存仍然放不下这么大的数据量。HayStack 在设计时，考虑从两个方面来减少元数据的总体数量：一方面由多个图片数据拼接成一个数据文件，这样就可以减少用于管理数据的数量；另一方面，一个图片的元数据包含多个属性信息，HayStack 考虑将文件系统中的元数据属性减少，只保留必需的属性。通过这两种方式即可在内存保留所有图片的元数据，原先的两次磁盘读取就改为：元数据从内存读取，图片数据从磁盘读取。通过这种方式，有效减少了磁盘读取操作，提高了系统性能。

### 7.11.1 HayStack 整体架构

在了解 HayStack 架构之前，需要熟悉相关的一些基本概念，首先需要了解什么是物理卷（Physical Volume）和逻辑卷（Logical Volume），图 7-48 是这两个基本概念的示意图。HayStack 存储系统由很多 PC 构成，每个机器的磁盘存储若干物理卷，前面讲过，为了减少文件元数据的数量，需要将多个图片的数据存储在同一个文件中，这里的物理卷就是存储多个图片数据对应的某个文件，一般一个物理卷文件大小为 100GB，可以存储上百万个图片数据。不同机器上的若干物理卷共同构成一个逻辑卷，在 HayStack 的存储操作过程中，是以逻辑卷为单位的，对于一个待存储的图片，会同时将这个图片数据追加到某个逻辑卷对应的多个物理卷文件末尾。之所以要这么做，主要是从数据冗余的角度考虑的，即使某台机器宕机，或者因为其他原因不可用，还可以从其他机器的物理卷中读出图片信息，这种数据的冗余是海量存储系统必须考虑的。

在了解了基本概念后，我们来看下 HayStack 系统的整体架构（参考图 7-49）。HayStack 由 3 个部分构成：HayStack 目录服务、HayStack 缓存系统和 HayStack 存储系统。当 Facebook 用户访问某个页面时，目录服务会为其中的每个图片构造一个 URL，通常 URL 由几个部分构成，典型的 URL 如下：

`http://<CDN>/<Cache>/<机器ID>/<逻辑卷ID,图片ID>`

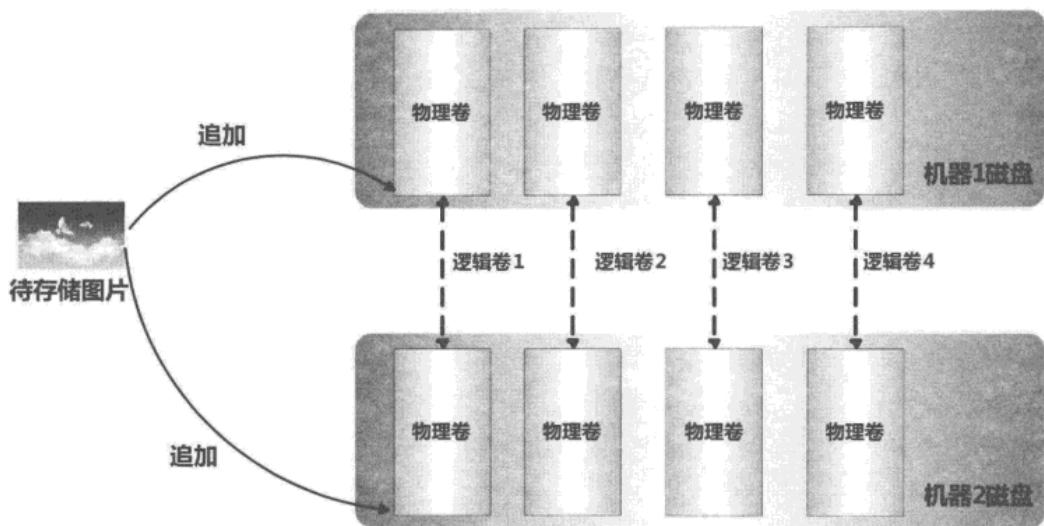


图 7-48 物理卷与逻辑卷概念的示意图

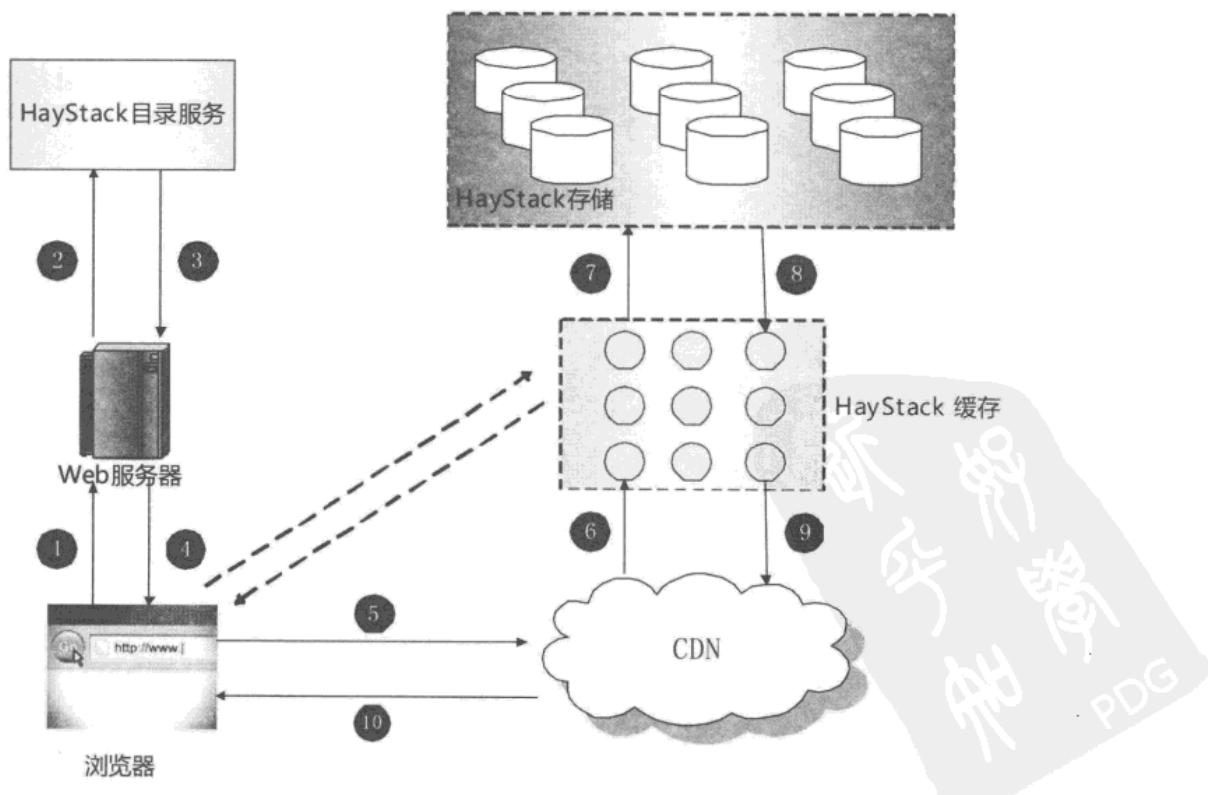


图 7-49 HayStack 系统的整体架构

<CDN>指出了应该去哪个 CDN 读取图片，CDN 在接收到这个请求后，在内部根据逻辑卷 ID 和图片 ID 查找图片，如果找到则将图片返回给用户，如果没有找到，则把这个 URL 的<CDN>部分去掉，将改写后的 URL 提交给 HayStack 缓存系统。缓存系统与 CDN 功能类似，首先在内部查找图片信息，如果没有找到就会到 HayStack 存储系统内读取，并将读出的图片放入缓存中，之后将图片数据返回给用户。这里需要注意的是：目录服务可以在构造 URL 的时候绕过 CDN，直接从缓存系统查找，这样做的目的是减轻 CDN 的压力，其内部查找过程是一样的。

上述是 HayStack 系统读取图片的流程，如果用户上传一张图片，其流程可参考图 7-50。当用户请求上传图片时，Web 服务器从目录服务中得到一个允许写入操作的逻辑卷，同时 Web 服务器赋予这个图片唯一的编号，之后即可将其写入这个逻辑卷对应的多个物理卷中。

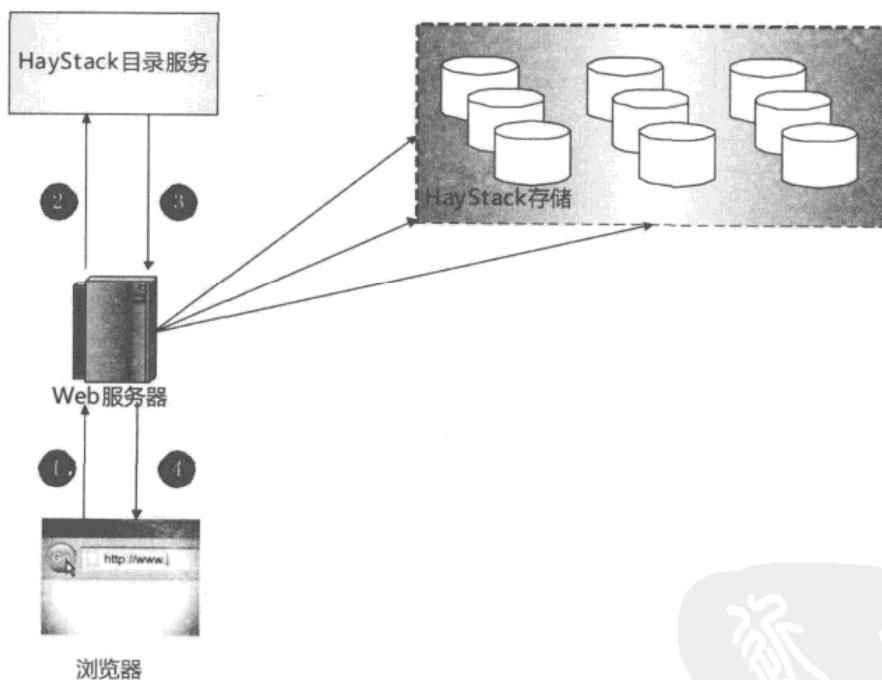


图 7-50 用户上传图片流程

### 7.11.2 目录服务

HayStack 的目录服务是采用数据库实现的，它提供多种功能。首先，目录服务保存了从逻辑卷到物理卷的映射关系表，这样在用户上传图片和读取图片时可以找到正确的文件。其次，目录服务提供了 HayStack 存储系统的负载均衡功能，保证图片写入和读取在不同机器之间负载是相当的，不至于出现机器之间忙闲不均的状况。再次，目录服务还决定是将



用户请求直接提交给缓存系统还是提交给 CDN，以此来对这两者接收到的请求量进行均衡。此外，通过目录服务还可以知道哪些逻辑卷是只读的，哪些逻辑卷可以写入。在有些情况下，某些逻辑卷会被标记为只读的，比如其物理卷已经基本被写满或者存储系统需要进行调试的时候。

### 7.11.3 HayStack 缓存

HayStack 缓存从功能上讲与 CDN 是一致的，缓存接收到的访问请求可能来自 CDN，也可能直接来自用户浏览器请求。在其内部实现，HayStack 采用哈希表的方式存储图片 ID 和其对应的数据，如果在缓存内没有找到图片，则从 HayStack 存储系统中读取图片并加入缓存中，之后将图片内容传给 CDN 或者直接传递给用户。

### 7.11.4 HayStack 存储系统

HayStack 存储系统是整个系统的核心组成部分。对于某台存储机来说，在磁盘存储了若干物理卷文件及其对应的索引文件，在内存为每个物理卷建立一张映射表，存放了图片 ID 到元数据的映射信息。每个图片的元数据包括删除标记位、在物理卷中的文件起始地址及图片大小，根据文件起始位置和图片大小就可以将图片信息读取出来（参考图 7-51 与图 7-52）。

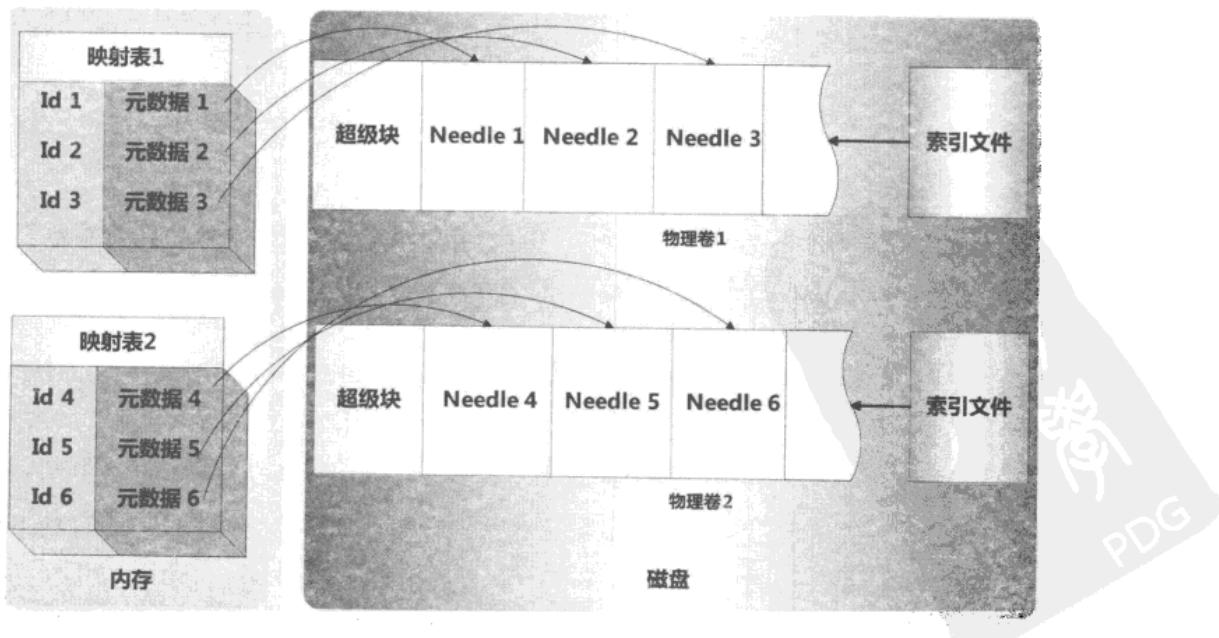


图 7-51 HayStack 存储系统



图 7-52 映射表项

对于每个物理卷文件，由一个超级块和图片数据组成（参考图 7-51），每个图片的信息被称为一个 Needle，图 7-53 是一个 Needle 具体包含的图片属性信息，其中比较重要的属性信息包括图片唯一标记 Key 和辅助 Key、删除标记位、图片大小及图片数据，除此之外还包含一些管理属性及数据校验属性。

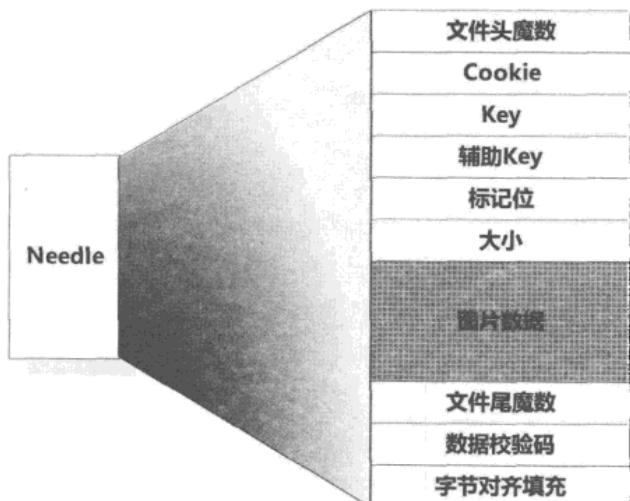


图 7-53 Needle 结构细节

图 7-51 中每个物理卷文件配有一个专门的索引文件，这个索引文件的目的是为了在机器重新启动时，能够快速恢复物理卷在内存中的映射表。其结构与物理卷的结构非常类似，也是由一个超级块和图片的 Needle 信息构成，不同点在于：索引文件里的 Needle 只包含少量重要信息，不包含图片本身的数据（参考图 7-54）。这样在恢复内存映射表的时候，相比顺序扫描非常大的物理卷来逐步恢复，从索引文件恢复的速度会快很多。

了解了 HayStack 的基本构成，下面我们来看看其是如何对文件信息进行存取的。

对于读取图片的请求来说，HayStack 缓存系统会向存储系统提供图片的逻辑卷 ID 编号及图片 ID（由 Key 和辅助 Key 构成），当存储系统接收到请求后，会在内存中的物理卷映射表中查找图片 ID，如果找到，则根据映射表保存的信息获取其在对应物理卷中的文件起始位置和文件大小，如此就可以读到这个图片的内容。

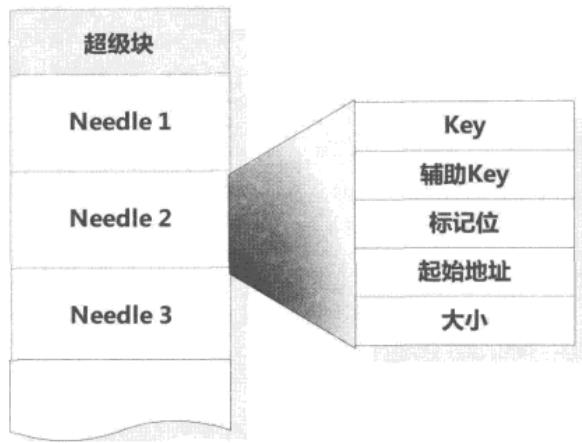


图 7-54 HayStack 索引文件结构

对于上传图片请求，HayStack 存储系统根据 Web 服务器传过来的图片逻辑卷 ID 编号及图片 ID 和图片数据，将这个图片信息追加到对应的物理卷文件末尾，同时在内存的映射表中增加相应的映射信息。

如果用户更改了图片的内容后再次上传，HayStack 存储系统不允许覆盖原先图片信息这种操作，因为这种操作严重影响系统效率，将这个修改的图片当做一个新的图片追加到物理卷的文件末尾，不过这个图片的 ID 是不变的。此时有两种可能，一种情况是更改后的图片的逻辑卷 ID 和原始图片的逻辑卷 ID 不同，这样更新图片会写入不同的物理卷中，此时目录服务修改图片 ID 对应的逻辑卷映射信息，此后对这个图片的请求就直接转换到更新后的图片，原始图片不会再次被访问；另外一种情况是更改后图片的逻辑卷与原始图片的逻辑卷相同，此时 HayStack 存储系统将新图片追加到对应的物理卷末尾，也就是说，同一个物理卷会包含图片的新旧两个版本的数据，但是由于物理卷是顺序追加的，所以更改后的图片在物理卷中的文件起始位置一定大于原始图片的起始位置，HayStack 在接收用户请求时会做判断，读取文件起始位置较大的那张图片信息，这样就保证读取到最新的图片内容。

如果用户删除某张图片，HayStack 系统的操作也很直观，只要在内存映射表中和物理卷上将相应的删除标记位置做出标记即可。系统会在适当的时机回收这些被删除的图片数据空间。

与前面章节讲述的云存储系统比较而言，HayStack 相对简单直观。Facebook 在实际使用中也证明了这种 BLOB 存储系统的高效。

## 本章提要

- CAP 和 BASE 原则是理解目前云存储与云计算系统设计思路的理论基础，从系统发展角度来说，云计算系统正在逐步地融合 ACID 原则的优点。
- 云存储系统的数据模型主要包含简单的 Key/Value 模型和相对复杂些的模式自由式数据模型。
- Google 公司公布的云存储技术包括：GFS 文件系统、Chubby 锁服务、BigTable 及 Megastore 等一系列不断进化的存储系统。主从式设计是 Google 公司在很多系统中体现出来的设计哲学。
- Google 公司公布的云计算体系包含 Map/Reduce、Percolator 及 Pregel 等互补的计算模式。
- Dynamo 系统是亚马逊公司公开的云存储系统，采取了对等式设计思路，并且在其系统实现引入了很多新技术。Google 公司的主从式设计思路和 Dynamo 的对等式设计思路影响了很多实际的云存储系统。
- PNUTS 是雅虎公司设计的云存储系统，跨数据中心存储、消息代理和时间轴一致性等方面是其设计特点。
- HayStack 是 Facebook 公司用来存储海量图片数据的云存储系统，这种 BLOB 式存储也是一种独具特色的云存储方案。  
;

## 本章参考文献

- [1] Ghemawat, S., Gobioff, H., and Leung, S.T.(2003). The Google file system. vol. 37, pp. 29–43.
- [2] Burrows, M. (2006). The chubby lock service for loosely-coupled distributed systems. In OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation, Berkeley, CA, USA. USENIX Association. 335-350.
- [3] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., and Chandra, T. (2008). BigTable: A distributed storage system for structured data. ACM Trans. Comput. Syst., 26(2):1-26.
- [4] Dean, J., and Ghemawat, S. (2004). MapReduce: simplified data processing on large clusters. In 6th OSDI , pp. 137-150.
- [5] Grzegorz, M. et al. (2009). Pregel: a system for large-scale graph processing, PODC '09



Proceedings of the 28<sup>th</sup> ACM symposium on Principles of distributed computing.

- [6] Lamport, L., Malkhi, D., and Zhou, L. (2009). Vertical paxos and primary-backup replication. Technical Report MSR-TR-2009-63, Microsoft Research.
- [7] Furman, J., Karlsson, J. S., Leon, J.M., Lloyd, A., Newman, S., and Zeyliger, P. (2008) Megastore: A scalable data system for user facing applications. In ACM SIGMOD/PODS Conference.
- [8] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., and Yerneni R.(2008). Pnutes: Yahoo!'s hosted data serving platform. Proc. VLDB Endow, 1(2):1277-1288.
- [9] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., and Vogels, W.(2007). Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty ACM SIGOPS symposium on Operating systems principles*, pages 205-220, New York, NY, USA, ACM.
- [10] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P.(2010). Finding a Needle in HayStack: Facebook's photo storage. In OSDI (Vancouver, BC).



# 第8章 网页反作弊

“唯之与阿，相去几何？美之与恶，相去若何？人之所畏，不可不畏。荒兮，其未央哉！众人熙熙，如享太牢，如春登台。我独泊兮，其未兆；沌沌兮，如婴儿之未孩；累累兮，若无所归。众人皆有馀，而我独若遗。我愚人之心也哉！俗人昭昭，我独昏昏。俗人察察，我独闷闷。众人皆有以，而我独顽且鄙。我独异于人，而贵食母。”

●老子·《道德经》

网页反作弊是目前所有商业搜索引擎需要解决的重要难点，出于商业利益驱使，很多网站站长会针对搜索引擎排名进行分析，并采取一些手段来提高网站排名，这种行为本身无可厚非，很多优化行为是符合搜索引擎排序规则的，但是也存在一些恶意的优化行为，通过特殊手段将网页的搜索排名提高到与其网页质量不相称的位置，这样会严重影响搜索引擎用户的搜索体验。而搜索引擎为了保证排名的公正性，也需要对作弊行为进行识别和处罚。所谓“道高一尺，魔高一丈”，只要这种经济利益存在，作弊与反作弊会一直作为搜索引擎领域的斗争而存在下去。

本章主要讲解目前常见的一些互联网网页作弊方法及搜索引擎公司对应的反制措施。从大的分类来说，比较常见的作弊方法包括：内容作弊、链接作弊、隐藏作弊及最近几年兴起的Web 2.0作弊方法。学术界和搜索引擎公司也有针对性地提出了各种反作弊算法，本章将介绍比较典型的各类反作弊算法思路，并抽象出了几种反作弊算法的框架。

## 8.1 内容作弊

内容作弊的目的是通过精心更改或者调控网页内容，使得网页在搜索引擎排名中获得与其网页不相称的高排名。搜索引擎排名一般包含了内容相似性和链接重要性计算，内容

作弊主要针对的是搜索引擎排序算法中的内容相似性计算部分，通过故意加大目标词词频，或者在网页重要位置引入与网页内容无关的单词来影响搜索结果排名。

### 8.1.1 常见内容作弊手段

比较常见的内容作弊方式包括如下几种。

#### 1. 关键词重复

对于作弊者关心的目标关键词，大量重复设置在页面内容中。因为词频是搜索引擎相似度计算中必然会考虑的因子，关键词重复本质上是通过提高目标关键词的词频来影响搜索引擎内容相似性排名的。

#### 2. 无关查询词作弊

为了能够尽可能多地吸引搜索流量，作弊者在页面内容中增加很多和页面主题无关的关键词，这本质上也是一种词频作弊，即将原先为 0 的单词词频增加到非 0 词频，以此吸引更多搜索引擎流量。

比如有的作弊者在网页的末端以不可见的方式加入一堆单词词表，也有作弊者在正文内容插入某些热门查询词，甚至有些页面内容是靠机器完全随机生成或者利用其他网页的页面内容片段随机拼凑而成的。

#### 3. 图片 alt 标签文本作弊

alt 标签原本是作为图片描述信息来使用的，一般不会在 HTML 页面显示，除非用户将鼠标放在图片上，但是搜索引擎会利用这个信息，所以有些作弊者将 alt 标签的内容以作弊词汇来填充，达到吸引更多搜索流量的目的。

#### 4. 网页标题作弊

网页标题作为描述网页内容的综述性信息，对于判断一个网页所讲述的主题是非常重要的启发因素，所以搜索引擎在计算相似性得分时，往往会增加标题词汇的得分权重。作弊者利用这一点，将与网页主题无关的目标词重复放置在标题位置来获得好的排名。

#### 5. 网页重要标签作弊

网页不像普通格式的文本，是带有 HTML 标签的，而有些 HTML 标签代表了强调内容重要性的含义，比如加粗标记`<b>`、`</b>`，段落标题`<h>`、`</h>`，字体大小标记等。搜索引擎一般会利用这些信息进行排序，因为这些标记因素能够更好地体现网页的内容所表现

的主题信息。作弊者通过在这些重要位置插入作弊关键词也能影响搜索引擎排名结果。

## 6. 网页元信息作弊

网页元信息，比如网页内容描述区（Meta Description）和网页内容关键词区（Meta Keyword）是供制作网页的人对网页主题信息进行简短描述的，同以上情况类似，作弊者往往也会通过在其中插入作弊关键词来影响网页排名。

通过以上几种常见作弊手段的描述，我们可以看出，作弊者的作弊意图主要有以下几类。

1. 增加目标作弊词词频来影响排名。
2. 增加主题无关内容或者热门查询吸引流量。
3. 关键位置插入目标作弊词影响排名。

### 8.1.2 内容农场（Content Farm）

Google 在 2011 年 2 月高调宣布针对低质量网页内容调整排序算法，据报道此算法影响了大约 11.8% 的网页排名，而这项调整措施是专门针对以 Demand Media 网站为代表的內容农场作弊手法的。

图 8-1 是内容农场运作模式的示意图，内容农场运营者廉价雇佣大量自由职业者，支持他们付费写作，但是写作内容普遍质量低下，很多文章是通过复制稍加修改来完成的，但是他们会研究搜索引擎的热门搜索词等，并有机地将这些词汇添加到写作内容中。这样，普通搜索引擎用户在搜索时，会被吸引进入内容农场网站，通过大量低质量内容吸引流量，内容农场可以赚取广告费用。

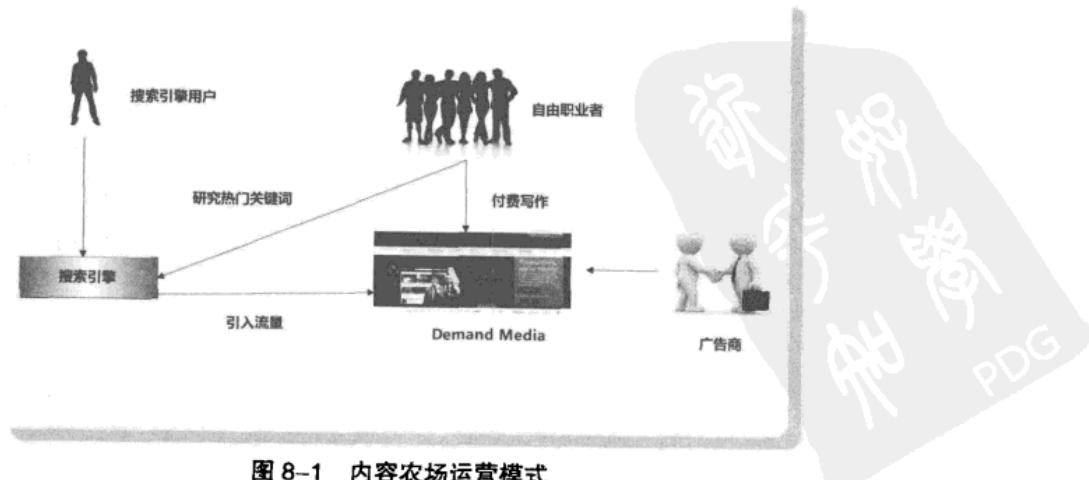


图 8-1 内容农场运营模式



与传统的内容作弊方式相比，内容农场不采用机器拼接内容等机械方式，而是雇佣人员写作，但是由于写作者素质等原因决定了其发布内容质量低下，这种作弊方式搜索引擎往往难以给出是否作弊的明确界定，但是又严重影响搜索结果质量，所以是一种很难处理的作弊手法。

## 8.2 链接作弊

所谓链接作弊，是网站拥有者考虑到搜索引擎排名中利用了链接分析技术，所以通过操纵页面之间的链接关系，或者操纵页面之间的链接锚文字，以此来增加链接排序因子的得分，并影响搜索结果排名的作弊方法。常见的链接作弊方法众多，此节简述几种比较流行的作弊方法。

### 1. 链接农场 (Link Farm)

为了提高网页的搜索引擎链接排名，链接农场构建了大量互相紧密链接的网页集合，期望能够利用搜索引擎链接算法的机制，通过大量相互的链接来提高网页排名。链接农场内的页面链接密度极高，任意两个页面都可能存在互相指向的链接。图 8-2 展示了一个精心构建的链接农场。

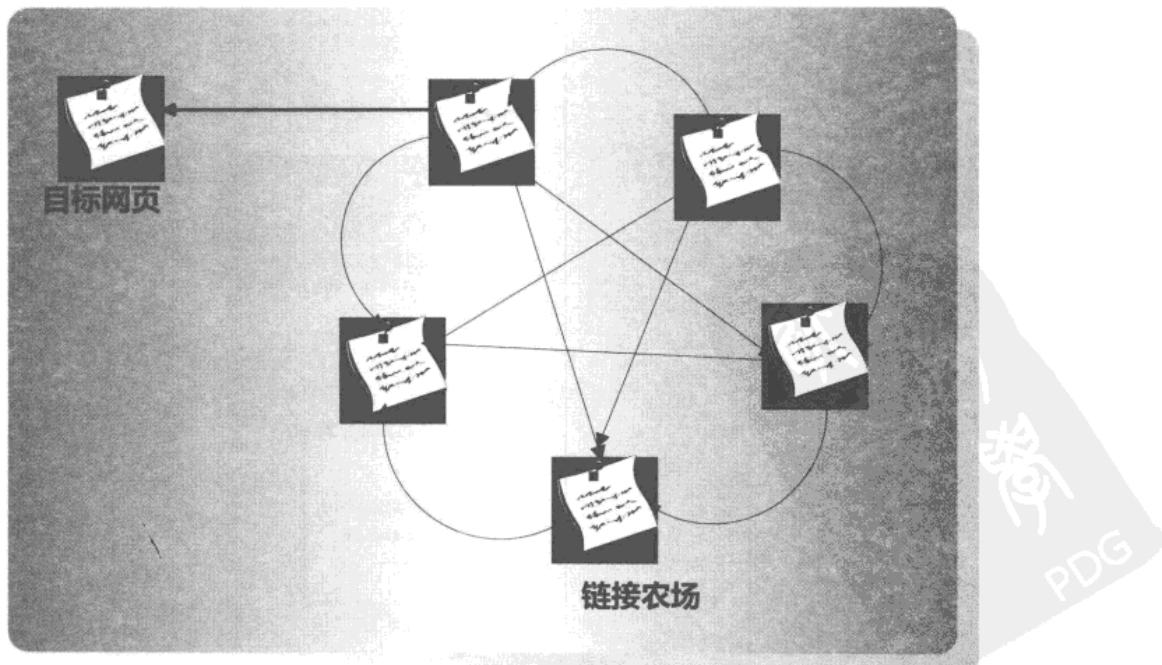


图 8-2 链接农场

## 2. Google 轰炸 (Google Bombing)

锚文字是指向某个网页的链接描述文字，这些描述信息往往体现了被指向网页的内容主题，所以搜索引擎往往会在排序算法中利用这一点。

作弊者通过精心设置锚文字内容来诱导搜索引擎给予目标网页较高排名，一般作弊者设置的锚文字和目标网页内容没有什么关系。

几年前曾经有个著名例子，采用 Google 轰炸来操控搜索结果排名。当时如果用 Google 搜索“miserable failure”，会发现排在第 2 位的搜索结果是美国时任总统小布什的白宫页面，这就是通过构建很多其他网页，在页面中包含链接指向目标页面，其链接锚文字包含“miserable failure”关键词（参考图 8-3 和图 8-4）所达到的效果。通过这种方式就导致了人们看到的搜索结果。

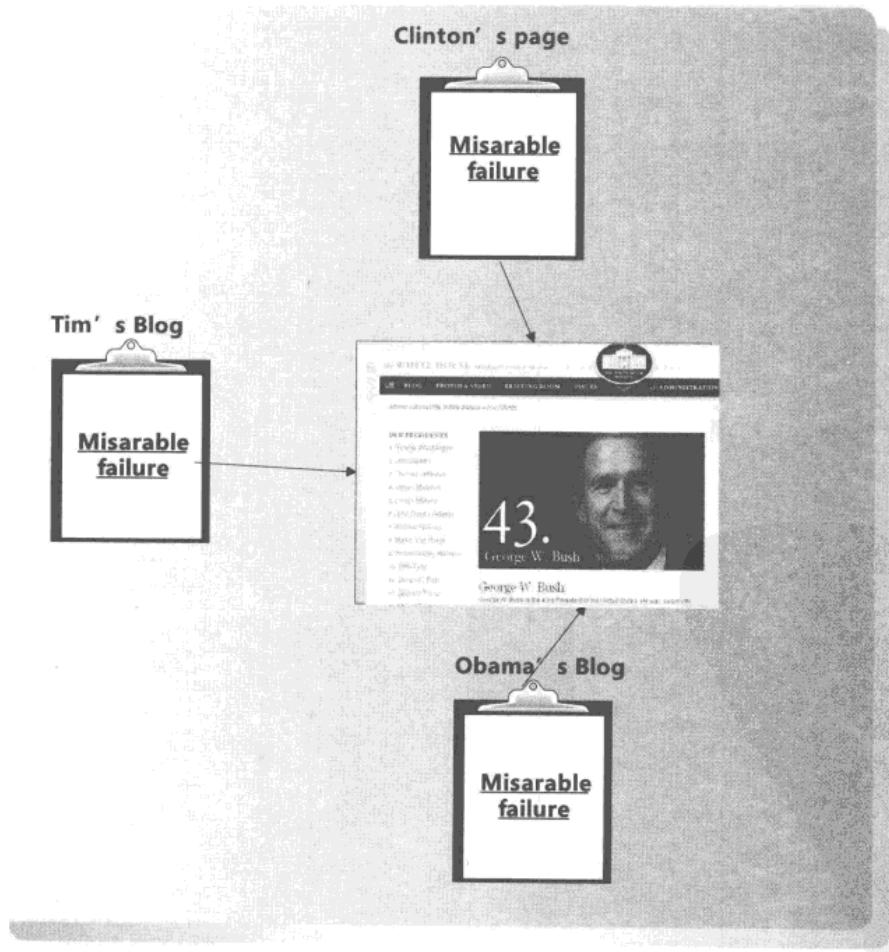


图 8-3 Google 轰炸的原理

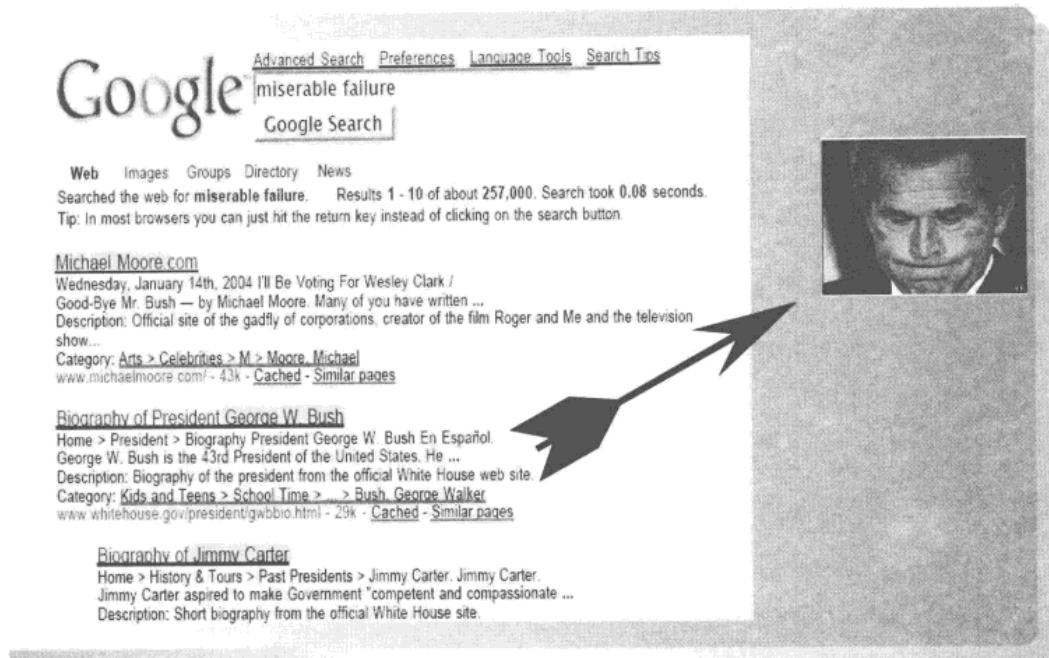


图 8-4 Google 轰炸后的效果

### 3. 交换友情链接

作弊者通过和其他网站交换链接，相互指向对方的网页页面，以此来增加网页排名。很多作弊者过分地使用此手段，但是并不意味着使用这个手段的都是作弊网站，交换友情链接的做法也是正常网站的常规措施。

### 4. 购买链接

有些作弊者会通过购买链接的方法，即花钱让一些排名较高的网站的链接指向自己的网页，以此来提高网站排名。

### 5. 购买过期域名

有些作弊者会购买刚刚过期的域名，因为有些过期域名本身的 PageRank 排名是很高的，通过购买域名可以获得高价值的外链。

### 6. “门页”作弊 (Doorway Pages)

“门页”本身不包含正文内容，而是由大量链接构成的，而这些链接往往会指向同一网站内的页面，作弊者通过制造大量的“门页”来提升网站排名。

## 8.3 页面隐藏作弊

页面隐藏作弊通过一些手段瞒骗搜索引擎爬虫，使得搜索引擎抓取的页面内容和用户点击查看到的页面内容不同，以这种方式来影响搜索引擎的搜索结果。常见的页面隐藏作弊方式有如下几种。

### 1. IP 地址隐形作弊 (IP Cloaking)

网页拥有者在服务器端记载搜索引擎爬虫的 IP 地址列表，如果发现是搜索引擎在请求页面，则会推送给爬虫一个伪造的网页内容，而如果是其他 IP 地址，则会推送另外的网页内容，这个页面往往是有商业目的的营销页面。

### 2. HTTP 请求隐形作弊 (User Agent Cloaking)

客户端和服务器在获取网页页面的时候遵循 HTTP 协议，协议中有一项叫做用户代理项 (User Agent)。搜索引擎爬虫往往会在这一项有明显的特征（比如 Google 爬虫此项可能是：Googlebot/2.1），服务器如果判断是搜索引擎爬虫则会推送与用户看到的不同的页面内容。

图 8-5 是一个 HTTP 请求隐藏作弊的例子，作弊网站服务器推送给搜索引擎爬虫的页面是讲述减肥食品的内容，而推送给页面访问者的则是减肥产品销售推广页面。这样当用户在搜索减肥知识的时候就会直接访问减肥产品页面，从而达到作弊者的商业目的。

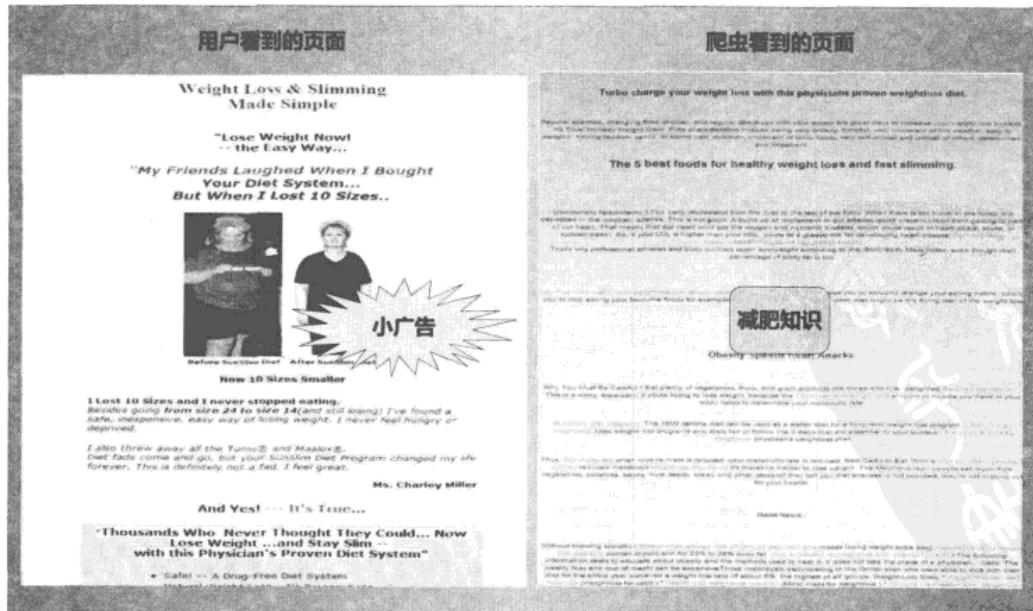


图 8-5 HTTP 请求隐藏作弊

### 3. 网页重定向

作弊者使搜索引擎索引某个页面内容，但是如果是用户访问则将页面重定向到一个新的页面。

### 4. 页面内容隐藏

通过一些特殊的 HTML 标签设置，将一部分内容显示为用户不可见，但是对于搜索引擎来说是可见的。比如设置网页字体前景色和背景色相同，或者在 CSS 中加入不可见层来隐藏页面内容。将隐藏的内容设置成一些与网页主题无关的热门搜索词，以此增加被用户访问到的概率。

## 8.4 Web 2.0 作弊方法

随着 Web 2.0 大潮的兴起，以用户为中心的信息产生和消费模式日益流行，这些产品和应用往往以用户产生内容、内容分享与协作、社会化网络等作为基本特征，比如博客、图片分享网站、Wiki、SNS 网站及微博等。针对大量的 Web 2.0 网站，也出现了相应的 Web 2.0 作弊方法。

### 1. 博客作弊

常见的博客作弊有 3 种：作弊博客（Splog）、博客评论作弊和 TrackBack 作弊。作弊博客是作弊者申请博客空间，而写作的博客内容用于诱导搜索引擎或者博客读者转到作弊者希望提升排名或者营销的网页。因为市场上存在大量的博客空间提供商，可以免费开博客，所以这种作弊成本很低，这也是此作弊方式比较流行的一个原因。

博客博主发布内容，往往允许观看博客的用户发表评论，有些作弊者利用这一点，在博客评论内容里推广产品或者增加指向推广网页的链接地址，这是非常常见的博客评论作弊方式，这种方式在论坛和微博等场合中也非常常见。

TrackBack 机制是博客作者之间相互引用通知的机制。比如博客 A 发表了一篇博文，之后博客 B 看到这篇文章后，发表了一篇主题类似的博文，并在文中使用链接引用博客 A 的博文，如果两者的博客系统都支持 TrackBak 协议，则博客系统会自动在博客 A 的文章后增加指向博客 B 新博文的链接。博客发布系统支持 TrackBack 协议的初衷是引导博客群体形成讨论氛围，不过很多作弊者利用这一点，使用自动 TrackBack 群发软件，向大量博文发出 TrackBack 链接，这样就增加了作弊页面被访问的机会。

## 2. 点评作弊

目前很多网站允许商品使用者对所消费的商品或者服务做出评价，典型的例子是“大众点评网”。这为作弊者打开了另外一个作弊通道：作弊者要在商品评论里面加入与所评论商品无关的广告内容来推广网站，要么提供虚假点评，比如对较差的产品给予较高评价或者打击竞争对手产品等。

## 3. 标签（Tag）作弊

在一些资源共享网站（比如视频共享、图片共享等），往往允许用户为资源打上一些标签，来对所分享内容进行主题说明，其他用户可以使用标签搜索等方式找到这些资源。作弊者往往会在标签里插入推广内容，这样可以吸引流量。

## 4. SNS 作弊

随着 Facebook 等 SNS 平台的日益流行，在 SNS 平台上作弊也逐渐发展起来。一种典型的 SNS 作弊手段是用户个人描述信息（Profile）作弊。作弊者建立一个虚假的个人描述，在其描述部分利用色情等信息吸引他人，并诱导其他用户点击其推广链接或者是向一些用户群组群发广告信息。

## 5. 微博作弊

微博是个人信息发布平台，以信息发布的及时性吸引了大量的用户。同样地，作弊者也会利用这个平台来作弊。一种典型的作弊方式是：作弊者大量关注他人微博，很多人出于礼貌也会将其作为关注者（互粉行为），在吸引到一定量的关注者后，作弊者会发布广告信息，这些广告信息就会出现在其关注者的阅读列表中，以此达到营销的目的。

另外一种典型的作弊方式是利用微博搜索平台，作弊者密切关注热门关键词，之后大量发布包含热门关键词的微博，里面包含一些推广信息，因为很多微博平台默认是按照时间发布先后来对搜索结果进行排序的，所以往往会在热门搜索词的搜索结果前列发现包含大量的作弊微博。

## 8.5 反作弊技术的整体思路

如上所述，目前搜索引擎作弊手段五花八门、层出不穷，作为应对方的搜索引擎，也相应调整技术思路，不断有针对性地提出反作弊的技术方案，所以如果整理反作弊技术方案，会发现技术方法很多，理清思路不易。

尽管如此，如果对大多数反作弊技术深入分析，会发现在整体技术思路上还是有规律



可循的。从基本的思路角度看，可以将反作弊手段大致划分为以下3种：信任传播模型、不信任传播模型和异常发现模型。其中前两种技术模型可以进一步抽象归纳为“链接分析”一章提到的子集传播模型，为了简化说明，此处不再赘述，而是直接将这两个子模型列出。将具体算法和这几个模型建立起关系，有助于对反作弊算法的宏观思路和相互联系建立起清晰的概念。

### 8.5.1 信任传播模型

图8-6展示了信任传播模型的示意图。所谓信任传播模型，基本思路如下：在海量的网页数据中，通过一定的技术手段或者人工半人工手段，从中筛选出部分完全值得信任的页面，也就是肯定不会作弊的页面（可以理解为白名单），算法以这些白名单内的页面作为出发点，赋予白名单内的页面节点较高的信任度分值，其他页面是否作弊，要根据其和白名单内节点的链接关系来确定。白名单内节点通过链接关系将信任度分值向外扩散传播，如果某个节点最后得到的信任度分值高于一定阈值，则认为没有问题，而低于这一阈值的网页则会被认为是作弊网页。

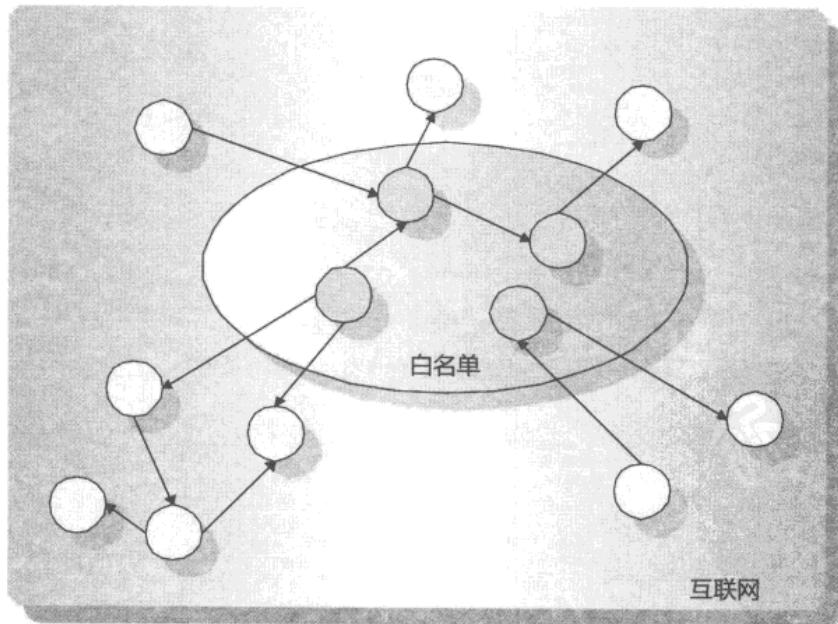


图8-6 信任传播模型示意图

很多算法在整体流程和算法框架上遵循如上描述，其区别点往往体现在以下两方面。

- 如何获得最初的信任页面子集合，不同的方法手段可能有差异。

- b. 信任度是如何传播的，不同的方法可能有细微差异。

### 8.5.2 不信任传播模型

图 8-7 展示了不信任传播模型的整体框架示意图。从大的技术框架上来讲，其和信任传播模型是相似的，最大的区别在于：初始的页面子集合不是值得信任的页面节点，而是确认存在作弊行为的页面集合，即不值得信任的页面集合（可以理解为黑名单）。赋予黑名单内页面节点不信任分值，通过链接关系将这种不信任关系传播出去，如果最后页面节点的不信任分值大于设定的阈值，则会被认为是作弊网页。

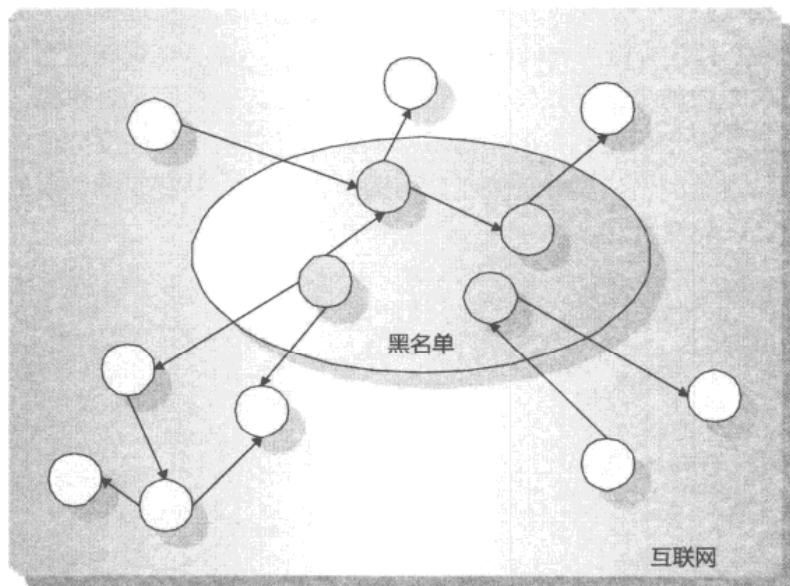


图 8-7 不信任传播模型的整体框架示意图

同样，很多算法可以归入这一模型框架，只是在具体实施细节方面有差异，整体思路基本一致。

### 8.5.3 异常发现模型

异常发现模型也是一个高度抽象化的算法框架模型，其基本假设认为：作弊网页必然存在有异于正常网页的特征，这种特征有可能是内容方面的，也有可能是链接关系方面的。而制定具体算法的流程往往是先找到一些作弊的网页集合，分析出其异常特征有哪些，然后利用这些异常特征来识别作弊网页。

具体来说，这个框架模型又可细分为两种子模型，这两种子模型在如何判断异常方面



有不同的考虑角度。一种考虑角度比较直观，即直接从作弊网页包含的独特特征来构建算法（参见图 8-8）；另外一种角度则认为不正常的网页即为作弊网页，也就是说，是通过统计等手段分析正常的网页应该具备哪些特征，如果网页不具备这些正常网页的特征，则被认为是作弊网页（参见图 8-9）。图 8-8 和图 8-9 体现了这两种不同的思路。

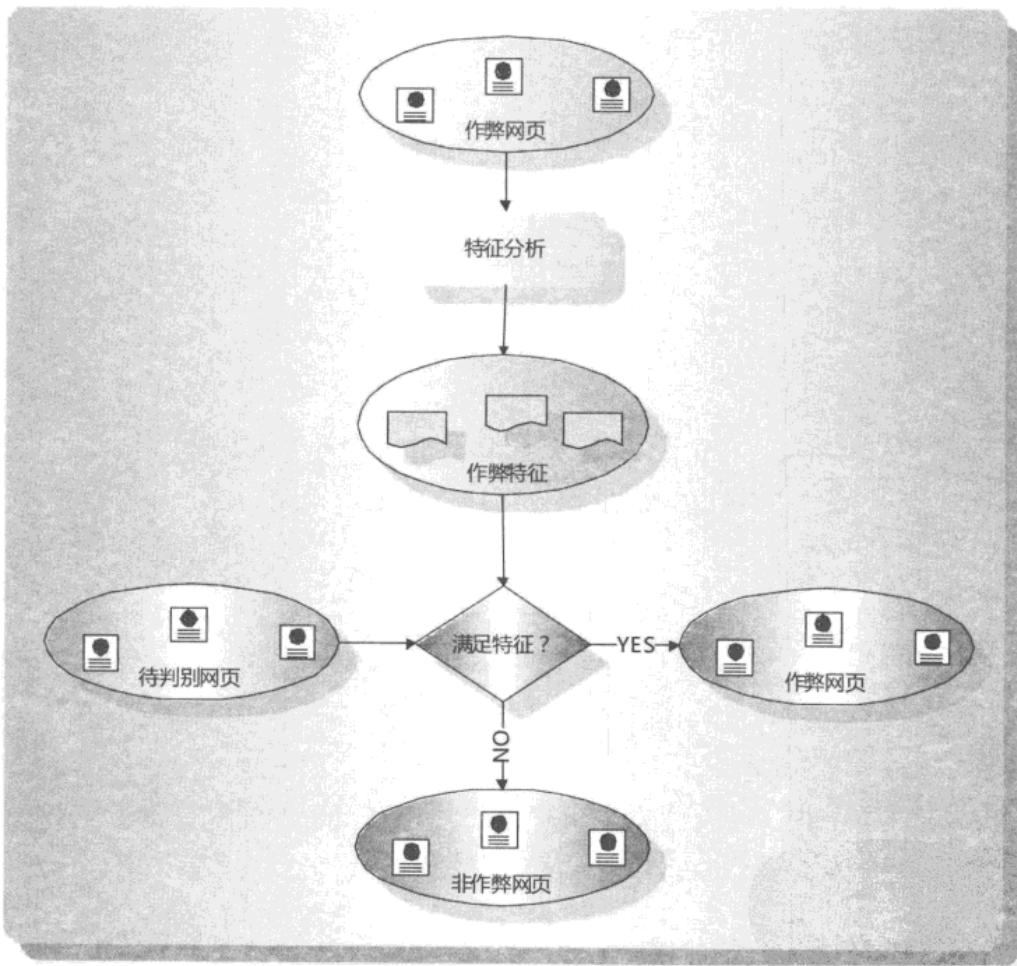


图 8-8 异常发现模型一

尽管反作弊算法五花八门，但是不论采取哪种具体算法，其实都包含了一些基本假设，经常被反作弊算法使用的基本假设有：

- 尽管作弊网页喜欢将链接指向高质量网页，但是很少有高质量网页将链接指向作弊网站。
- 作弊网页之间倾向于互相指向。

很多算法的基本思路都是从这些基本假设出发来构造的。

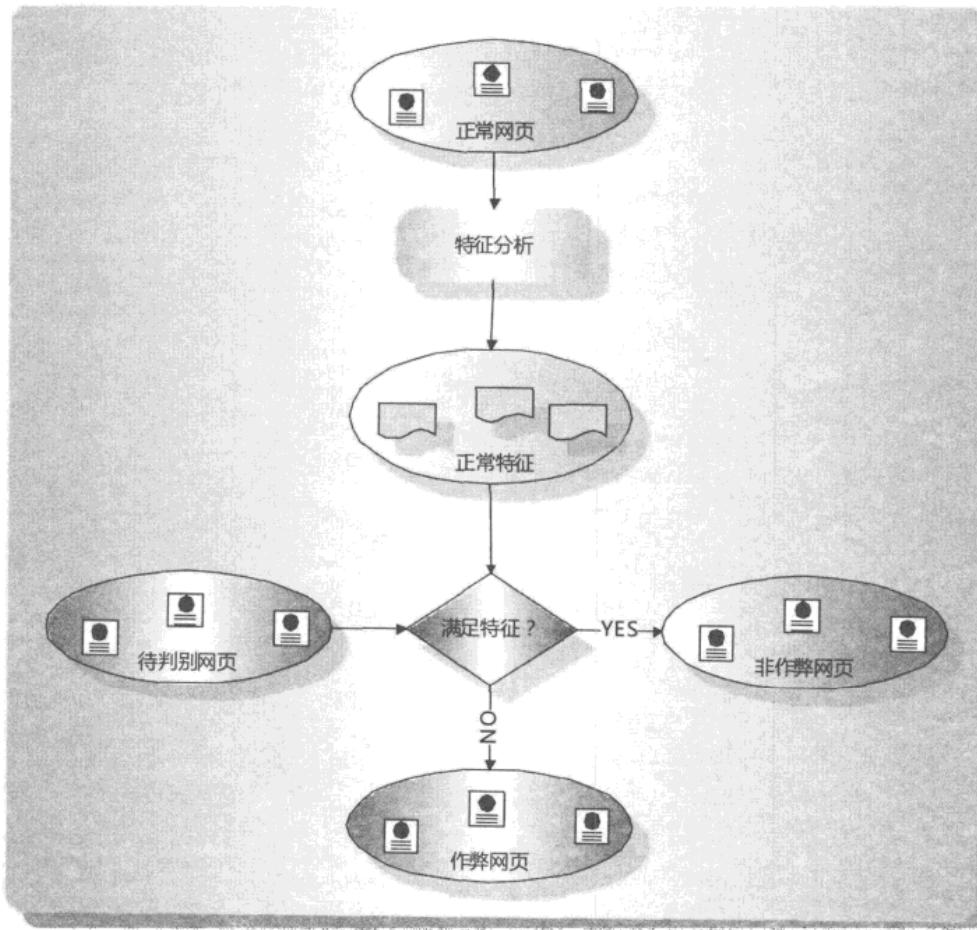


图 8-9 异常发现模型二

## 8.6 通用链接反作弊方法

所谓通用链接反作弊方法，指的是这种反作弊方法不需要针对某种具体的作弊方式来做特征分析，并根据分析结果去构建有针对性的算法。不论采取哪种链接作弊方法，通用反作弊方法都是对其有效的。

上节内容介绍了通用反作弊方法的整体思路，其中信任传播模型、不信任传播模型及图 8-9 所示的异常发现模型有一个共同特点，即不需要拿到作弊网页后进行单独的特征分析，然后根据分析结果构建具体算法。所以如果某个具体反作弊方法属于上述 3 种模型之



一的话，都可以认为是一种通用反作弊方法。

我们分别针对信任传播模型、不信任传播模型及如图 8-9 所示的异常发现模型介绍 3 个代表性算法，它们分别是 TrustRank 算法、BadRank 算法和 SpamRank 算法。

### 8.6.1 TrustRank 算法

TrustRank 算法属于信任传播模型，基本遵循信任传播模型的流程，即算法流程由如下两个步骤组成。

#### 步骤一：确定值得信任的网页集合

TrustRank 算法需要靠人工审核来判断某个网页是否应该被放入信任网页集合，考虑到人工审核工作量过大，所以提出了两种初选信任网页集合的策略，在初选集合基础上再由人工审核。

- **初选策略 1：**高 PageRank 分值网页，即认为高 PageRank 得分的网页是可信赖的，所以可以对网页计算 PageRank 值后，提取少量高分值网页作为初选页面集合。
- **初选策略 2：**逆 PageRank (Inverse PageRank)，在 PageRank 计算过程中，是根据网页入链传入的权值计算的，逆 PageRank 与此相反，根据网页的出链传出的权值计算，即先将网页之间的链接指向关系反转，然后计算 PageRank，选取得分较高的一部分页面子集作为初选页面。

#### 步骤二：将信任分值从白名单网页按照一定方式传播到其他网页

在这个步骤，TrustRank 算法的信任传播方式基于以下两个假设。

- **假设 1：**距离可信网页越近越值得信任，这里的距离指的是通过多少步链接中转可以通达。
- **假设 2：**一个高质量网页包含的出链越少，那么被指向的网页是高质量网页的可能性越大。反过来，如果出链越多，则被指向网页是高质量网页的可能性越小。

基于以上两个假设，在信任传播阶段，TrustRank 算法引入了信任衰减因子 (Trust Dumpling) 和信任分值均分 (Trust Splitting) 策略。

所谓信任衰减，即距离可信网页越远的网页，通过传播得到的信任分值越少。图 8-10 说明了这个策略。在图 8-10 中，节点 2 从节点 1 传递得到信任分值  $b$ ，节点 3 因为距离节点 1 更远，所以从节点 2 获得的分值为  $b$  的平方，因为链接中传递的信任分值都在 0 到 1 之间，所以等于信任分值随着距离的增大获得了衰减。

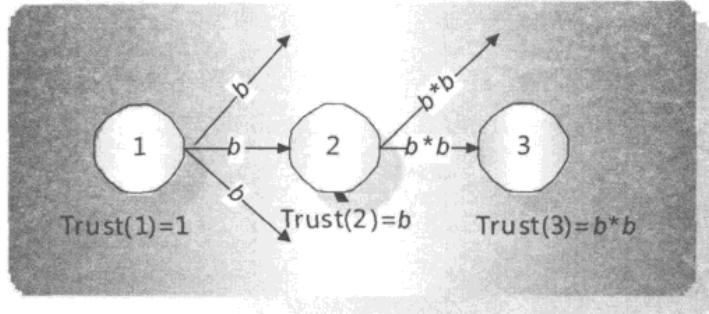


图 8-10 信任衰减

所谓信任分值均分策略，即将网页获得的信任分值按照出链个数平均分配，如果一个网页有  $K$  个出链，则每个出链分配得到  $1/K$  的信任分值，并将这个分值传递给出链指向的页面，图 8-11 说明了这个策略。在图 8-11 中，节点 1 和节点 2 的信任度分值都是 1，节点 1 有两个出链，所以每个出链分配得到  $1/2$  的信任分值，节点 2 有 3 个出链，所以每个出链获得  $1/3$  的信任分值，节点 3 被节点 1 和节点 2 同时指向，所以通过传递获得了  $5/12$  的信任分值，节点 3 再将自己获得的信任分值依次传递出去。

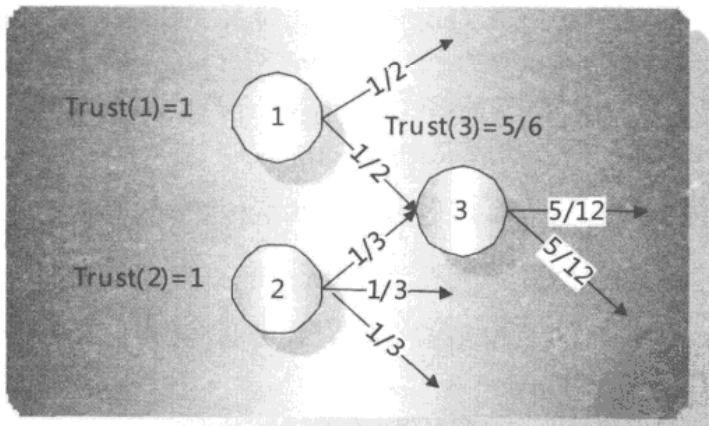


图 8-11 信任分值均分

通过结合以上两个传播策略可以在页面节点图之间传播信任分值，在最后的计算结果中，低于一定信任度的页面会被认为是作弊网页。

### 8.6.2 BadRank 算法

BadRank 据传是 Google 采用的反链接作弊算法。它是一种典型的不信任传播模型，即首先构建作弊网页集合，之后利用链接关系来将这种不信任分值传递到其他网页。



**BadRank** 包含的基本假设是：如果一个网页将其链接指向作弊页面，则这个网页也很可能是作弊网页；而如果一个网页被作弊网页指向，则不能说明这个网页是有问题的，因为作弊网页也经常将其链接指向一些知名网站。所以 **BadRank** 的基本思路是：找到那些有链接关系指向已知作弊网页的页面，这些页面很可能也是作弊网页。

**BadRank** 首先收集一批已经确认的作弊网页形成作弊网页集合（黑名单），黑名单内网页赋予较高的不被信任分值，而不被信任分值是通过网页的链接关系来进行传播的，其计算过程与 **PageRank** 基本相同，与 **PageRank** 不同的是，**PageRank** 是根据网页的出链来进行权值传播，而 **BadRank** 是按照入链来进行权值传播，所以可以将 **BadRank** 理解为首先将网页之间的链接指向关系反转，假设页面 A 有链接指向页面 B，则将链接反转为由 B 指向 A，之后可以按照 **PageRank** 的计算方法进行计算。

**BadRank** 的计算思路是非常有代表性的，后续提出的 **ParentRank** 及 **Anti-Trust PageRank** 在整体思路上与 **BadRank** 是一致的，基本上都是不信任传播模型的具体实现实例。

**BadRank** 的计算思路很明显和 **TrustRank** 是互补的，一个方法是从黑名单出发传播不信任关系，另外一个是白名单出发传播信任关系，所以两者有天然的互补性。

### 8.6.3 SpamRank

**SpamRank** 是一种典型的符合异常发现模型的反作弊方法，也就是说，首先定义正常的网页或者链接关系应该满足哪些特性，如果某些网页不满足这些特性，则可以被认为是异常的，而这些异常网页很可能是作弊网页。

**SpamRank** 是构建在 **PageRank** 计算之上的反作弊算法，**PageRank** 是网页重要性评价指标，通过全局的链接关系可以迭代计算网页的最终 **PageRank** 值。对于某个网页 A 来说，在最终获得其 **PageRank** 值后，可以分析获得哪些网页对于页面 A 的最终 **PageRank** 计算有比较重要的影响，这些网页被称为网页 A 的支持者（Supporter）。

**SpamRank** 的基本假设是：对于正常页面来说，其支持者页面的 **PageRank** 值应该满足 **Power-Law** 统计分布，即 **PageRank** 值有大有小。而作弊网页则不然，其支持者页面的 **PageRank** 值分布不满足 **Power-Law** 分布，具有如下 3 个特点。

1. 支持者页面数量非常巨大。
2. 支持者页面的 **PageRank** 值得分都较低。
3. 支持者页面的 **PageRank** 值都落在一个较小的浮动范围内。

**SpamRank** 就是利用了作弊网页的支持者 **PageRank** 分值的不正常分布规律来自动发现可能的作弊网页的。首先计算网页的支持页面有哪些，之后判断其支持页面的 **PageRank**

分布是否违反了 Power-Law 分布，对于那些明显违反规律的页面作为可疑作弊页面。最后，类似于 BadRank 的思路，通过这些可疑页面的链接关系，发现更多可能有问题的网页，作为可能的作弊网页。

在 SpamRank 算法提出后，又陆续有类似思路的算法提出，比如截断 PageRank 算法（Truncated PageRank）等在基本思想上和 SpamRank 算法思路基本相近。这些都是比较典型的符合异常发现模型的反作弊算法。

## 8.7 专用链接反作弊技术

上一节所述的通用链接反作弊技术与具体作弊方法无关，具有通用性，只要作弊手段采用了链接分析，一般都会有一定的识别作用。但是通用性的代价是针对某些具体的链接作弊方法的，其识别效果因为没有针对性，所以可能不会太好，专用的链接反作弊技术则是非常有针对性的设计算法，往往效果较好。本节简述针对链接农场和 Google 轰炸的专用反作弊技术。

### 8.7.1 识别链接农场

链接农场是作弊者精心构建起来的页面链接关系，和正常的链接必然有不同之处。很多研究通过比较正常网页之间链接关系的统计规律，同时研究链接农场网页之间的链接关系分布规律，通过比较两者之间的差异来识别链接农场。

识别算法比较常用的统计特征包括如下几条。

1. 网页出链的统计分布规律，正常网页的出链满足 Power-law 分布，作弊网页的出链违反该分布。
2. 网页入链的统计分布规律，正常网页的入链也满足 Power-law 分布，作弊网页则违反该分布。
3. URL 名称统计特征，作弊网页的网址倾向于较长，包含更多的点画线和数字等。
4. 很多作弊网页的 URL 地址尽管不同，但是常常会对应同一个 IP 地址。
5. 网页特征会随着时间变化，比如入链的增长率、出链的增长率等，正常网页和作弊网页在这些变化模式上是不同的。

除了对比统计特征外，还可以利用链接农场的结构特征。链接农场的一个结构特征是农场内的网页之间链接关系非常紧密，这也是可以直接用来进行作弊识别的特征。使用一些紧密链接子图自动发现算法，可以识别出这些紧密链接的页面子图，研究表明这种紧密



链接子图中很大比例确实是由作弊网页构成的。

### 8.7.2 识别 Google 轰炸

Google 轰炸利用了指向目标网页的锚文字来操纵搜索结果排名，而锚文字很可能和被指向的页面没有任何语义关系，所以一个直观的判断方式即为判断锚文字是否和被指向页面有语义关系，如果有语义关系存在，则被判断为正常链接，否则可被判断为作弊链接。

但是事实上由于锚文字都比较短小，如果在字面上和被指向页面内容没有直接关系也是很正常的，所以自动判断 Google 轰炸作弊具有较大难度。

## 8.8 识别内容作弊

上述章节是针对链接作弊方法的一些可能反制方法，本节叙述针对内容作弊的一些反制方法。针对内容作弊，往往可以采用一些启发规则或者内容统计分析的方式进行识别。

比如对于重复出现关键词这种作弊方式，可以判断文本内一定大小的窗口中是否连续出现同一关键词，如果是的话则消除掉重复出现的内容。

比如对于标题关键词作弊，可以判断标题词汇在文本正文出现的比例和权重，如果达到一定条件则可判断为标题关键词作弊。

也可以采用一些统计手段来进行内容作弊识别，比如统计正常网页中句子长度的规律、停用词的分布规律或者词性分布规律等，通过比较页面内容统计属性是否异常来识别内容作弊的情况。

## 8.9 反隐藏作弊

常见的隐藏作弊方式包括页面隐藏和网页重定向，下面介绍一些技术思路来识别隐藏作弊网页。

### 8.9.1 识别页面隐藏

页面隐藏的本质特征是向搜索引擎爬虫和用户推送不同内容的页面。所以一个直观的识别这种作弊方式的方法就是对网页做两次抓取，第1次是正常的搜索引擎爬虫抓取，第2次抓取则以模拟人工访问网页的方式抓取。如果两次抓取到的内容有较大差异，则会认为是作弊页面。很明显，这种方法虽然有效，但是对所有页面做多次抓取的成本显然非常高。

考虑到以上方法的效率问题，研究人员希望将识别范围缩小。因为作弊者大都具有商业动机，所以他们认为包含一些热门查询，以及具有商业价值查询词的页面更可能会采取隐藏作弊。可以从查询日志中挖掘最热门的查询，同时挖掘出能够引发搜索结果中出现“赞助商链接”的商业性词汇。经过分别使用搜索引擎爬虫和模拟人工访问，多次抓取排在搜索引擎结果前列的网页，并比较两次下载页面的单词重叠度。研究人员发现包含商业性词汇的页面中，如果网站采取了页面隐藏，则有 98% 的内容是作弊页面，而在包含热门查询词的网页中，这个比例是 73%。

### 8.9.2 识别网页重定向

网页重定向是很容易识别的，目前大部分搜索引擎对于采取了重定向的网页都会有相应的降权惩罚。但是，采取了重定向的网页未必一定就是作弊网站，如何更精确地识别此类作弊方式是个值得探讨的问题。

Strider 系统给出了根据网页重定向来识别到底哪些是作弊网页的解决方案。这个系统首先收集一批作弊页面，然后根据这批作弊网页进行扩展，如果有在论坛中和这些作弊 URL 经常一起出现的网页链接，会逐步将其扩充进可疑页面集合。之后，依次访问这些可疑 URL，并记录下访问时是否做了重定向及重定向到哪个页面，如果某个页面被很多可疑 URL 重定向指向，则认为这个重定向地址是作弊网页，反过来，那些重定向到这个作弊网页的可疑 URL 也被认为是作弊网页，其他可疑 URL 则可以被认为是正常网页。

## 8.10 搜索引擎反作弊综合框架

只要操纵搜索引擎搜索结果能够带来收益，那么作弊动机就会始终存在，尤其是在网络营销起着越来越重要宣传作用的时代尤其如此。作弊与反作弊是相互抑制同时也是相互促进的一个互动过程，“道高一尺，魔高一丈”的故事不断重演。

本章前述内容主要是以技术手段来进行反作弊，而事实上纯粹技术手段目前是无法彻底解决作弊问题的，必须将人工手段和技术手段相互结合，才能取得较好的反作弊效果。技术手段可以分为相对通用的手段和比较特殊的手段，相对通用的手段对于可能新出现的作弊手法有一定的预防能力，但是因为其通用性，所以针对性不强，对特殊的作弊方法效果未必好。而专用的反作弊方法往往是事后诸葛亮，即只有作弊行为已经发生并且比较严重，才可能归纳作弊特征，采取事后过滤的方法。人工手段则与技术手段有很强的互补性，可以在新的作弊方式一出现就被人发现，可以看做一种处于作弊进行时的预防措施。所以从时间维度考虑对作弊方法的抑制来说，通用反作弊方法重在预防，人工手段重在发现，



而专用反作弊方法重在事后处理，其有内在的联系和互补关系存在。

一个有效的搜索引擎反作弊系统一定是一个综合系统，有机融合了人工因素、通用技术手段和专用技术手段。图 8-12 给出了一个综合反作弊系统的框架，用户可以在浏览搜索结果甚至是上网浏览时随时举报作弊网页，比如 Google 推出了浏览器插件来方便用户举报，搜索引擎公司内部会有专门的团队来审核与主动发现可疑页面，经过审核确认的网页可以放入黑名单或者白名单中。

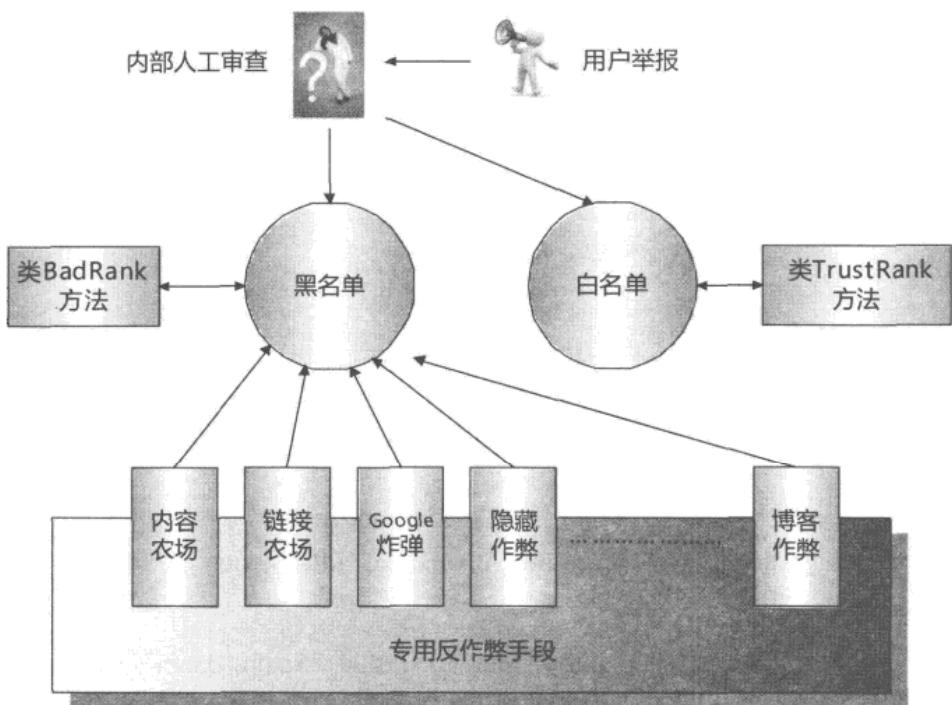


图 8-12 综合反作弊框架

通用的反作弊方法大体有两类，一种类似于 BadRank 的思路，即从黑名单出发根据链接关系探寻哪些是有问题的网页；另外一种类似于 TrustRank 的思路，即从白名单出发根据链接关系排除掉那些没有问题的网页。两者显然有互补关系，通过两者搭配可以形成有效的通用反作弊屏障。这种通用方法的好处是具有预防性，哪怕是新出现的作弊方式，只要作弊网页需要通过链接关系进行操纵，那么通用方法就能在一定程度上起到作用。但是正是因为通用方法的通用性，所以其反作弊思路没有针对性，对于一些特殊的作弊手段无法有效发现。此时，针对特殊作弊手段的方法形成了第 3 道屏障，即搜索引擎公司针对具体作弊方法采取专用技术手段来进行识别，因为有针对性所以效果较好，但是缺点在于一类反作弊方法只能识别专门的作弊手段，对于新出现的作弊方法往往无能为力，而且在时

间上往往滞后于作弊现象。

综上所述，这几种反作弊方法是有互补关系存在的，有效融合三者才能够获得较好的反作弊效果。

## 本章提要

- 作弊与反作弊相生相克，只要作弊存在经济利益，两者斗争一定会持续。
- 常见的作弊方法包括：内容作弊、链接作弊、隐藏作弊和 Web 2.0 作弊。
- 通用反作弊手段大致划分为以下 3 种类型：信任传播模型、不信任传播模型和异常发现模型。
- 纯粹用技术手段目前无法彻底解决作弊问题，必须将人工手段和技术手段相互结合，才能取得较好的反作弊效果。

## 本章参考文献

- [1] Gyongyi, Z. and Garcia-Molina, H. (2005). Web spam taxonomy. In First International Workshop on Adversarial Information Retrieval on the Web.
- [2] Wu, B. and Davison, B. (2005). Cloaking and redirection: a preliminary study. In First International Workshop on Adversarial Information Retrieval on the Web (AIRWeb '05).
- [3] Fetterly, D., Manasse, M. and Najork, M. (2004). Spam, damn spam, and statistics: Using statistical analysis to locate spam web pages. In S. Amer-Yahia and L. Gravano, editors, WebDB, 1–6.
- [4] Ntoulas, A., Najork, M., Manasse, M., and Fetterly, D. (2006). Detecting spam web pages through content analysis. In Proceedings of the 15th International Conference on World Wide Web (Edinburgh, Scotland). WWW '06. ACM Press, New York, NY, 83–92.
- [5] Gyöngyi, Z., Garcia-Molina, H., and Pedersen, J. (2004). Combating web spam with trustrank. In Proceedings of the Thirtieth international Conference on Very Large Data Bases - Volume 30. 576–587.
- [6] Krishnan, V. and Raj, R. (2006). Web Spam Detection with Anti-Trust-Rank. In the 2nd International Workshop on Adversarial Information Retrieval on the Web (AIRWeb) .



- [7] Becchetti, L., Castillo1, C., Donato1, D., Leonardi, S., and Baeza-Yates, R.(2006). Using Rank Propagation and Probabilistic Counting for Link Based Spam Detection. In Proc. of WebKDD'06.
- [8] Baoning ,W. and Brian, D. (2005). Identifying link spam farm pages. In *WWW*, 2005.
- [9] Gyongyi, Z., Garcia-Molina, H., Berkhin, P., and Pedersen, J.(2006). Link spam detection based on mass estimation. In *VLDB*.
- [10] Bencz, A. and Uher, M. (2005). Spamrank: fully automatic link spam detection. In Proceedings of the First International Workshop on Adversarial Information Retrieval on the Web, Chiba, Japan.
- [11] Bencz, A. and Uher, M. (2006). Detecting nepotistic links by language model disagreement. In *WWW*, 939–940.
- [12] Paul, H., Georgia, K., and Hector, G.(2007) “Fighting Spam on Social Web Sites: A Survey of Approaches and Future Challenges,” *IEEE Internet Computing*, vol. 11, pp. 36-45.



# 第9章 用户查询意图分析

“得天下有道：得其民，斯得天下矣；得其民有道：得其心，斯得民矣；得其心有道：所欲与之聚之，所恶勿施，尔也。”

孟子·《论民本》

搜索引擎与用户交互的界面非常简洁，用户输入查询词，搜索引擎返回搜索结果，过程看似简单，背后其实复杂。用户输入的每个查询词都隐含了其深层次的查询意图，而这些查询意图往往需要深入挖掘才能获得。

用户查询意图分析目前是搜索引擎研究的焦点领域，只有准确了解搜索用户到底想要什么，提供满意服务才会成为可能。本章主要介绍与用户查询意图分析相关的技术领域。首先介绍用户的搜索行为及用户查询意图的分类，接下来介绍对搜索日志的不同处理方法，搜索日志是目前搜索引擎广泛采用的深入挖掘用户意图的有效数据源，很多技术手段都是以此作为基础的。在本章后续两节将介绍相关搜索及查询纠错这两个非常常见的搜索引擎功能。

## 9.1 搜索行为及其意图

搜索行为目前已经成为了每个上网人的基本需求，但是用户的搜索行为是怎样一个过程？隐藏在用户查询背后的搜索意图是什么？这都是需要仔细研究的领域，只有这样才能提供更好的用户体验。

### 9.1.1 用户搜索行为

用户之所以会产生搜索行为，往往是在解决任务时遇到自己不熟悉的概念或者问题，由此产生了对特定信息的需求，之后用户会在头脑中逐步形成描述需求的查询词，将查询提交给搜索引擎，然后对搜索结果进行浏览，如果发现搜索结果不能完全解决用户的信息需求，则会根据搜索结果的启发，改写查询，以便更精确地描述自己的信息需求，之后重



新构造新的查询请求，提交给搜索引擎，如此形成用户和搜索引擎交互的闭合回路，直到搜索结果已经解决了自己的需求或者尝试几次无果而终。图 9-1 是描述这种用户搜索行为的示意图。

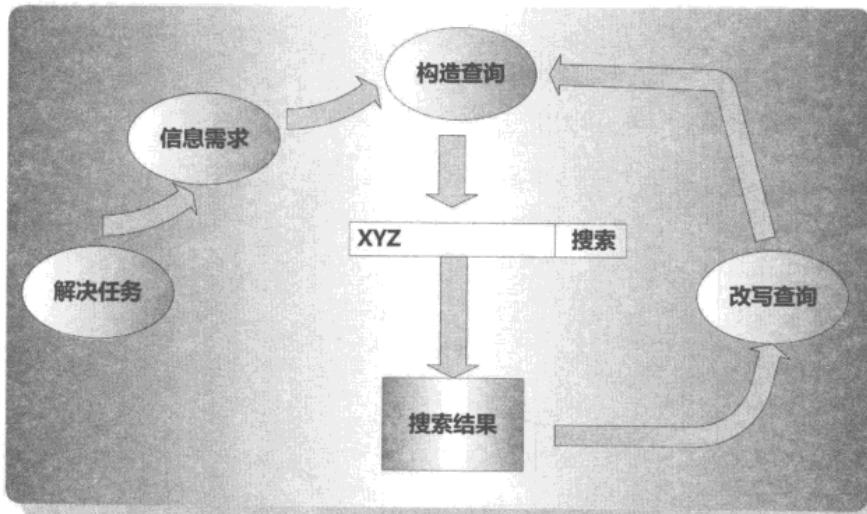


图 9-1 用户搜索行为

从上述过程可以看出，从用户产生信息需求到最终形成用户查询，中间有很大的不确定性，用户未必能够一开始就找到合适的查询词，即使是找到了，也可能存在查询词不能完全描述信息需求的情形，即在形成查询的过程中存在信息丢失的问题。所以后续循环中的查询改写就是用户逐步澄清搜索需求的一个过程。

常见的查询改写有 3 种方式：抽象化改写、具体化改写及同义重构改写。抽象化改写将原来的查询进行语义抽象，比如原先的查询是“东北虎照片”，经过抽象化改写成为“老虎照片”，因为“老虎”在语义概念层次上要更加宽泛，包含了“东北虎”这个概念，所以是一种抽象化的过程，之所以要做这种类型的改写，往往是因为原先查询找到的东西太少，通过概念泛化增加搜索的召回率，以此方式找到更多的内容。

具体化改写正好与此相反，从宽泛的语义概念下行收窄，比如将查询“室外活动场所”改写为“踏青场所”，改写后的查询更加具体，这么做可以更加精确地定位查找内容。

同义重构改写则保持改写前后的查询含义不变，比如将查询“旧汽车”改写为“二手车”，两者代表的含义是相同的，用户如此改写往往是对原先查询的搜索结果不满意，所以换了一种同义说法来搜索。

### 9.1.2 用户搜索意图分类

用户发出的每个搜索请求背后都隐含着潜在的搜索意图，如果搜索引擎能够根据查询词汇自动找出背后的用户搜索意图，然后针对不同的搜索意图，提供不同的检索方法，将更符合用户意图的搜索结果排在前列，无疑会增加搜索引擎用户的搜索体验。目前搜索引擎已经部分实现了这种搜索模式，比如用户搜索“北京 天气”的时候会主动将当天的气温等情况列在搜索结果最前面。图 9-2 是这种方法的一个结构示意图。

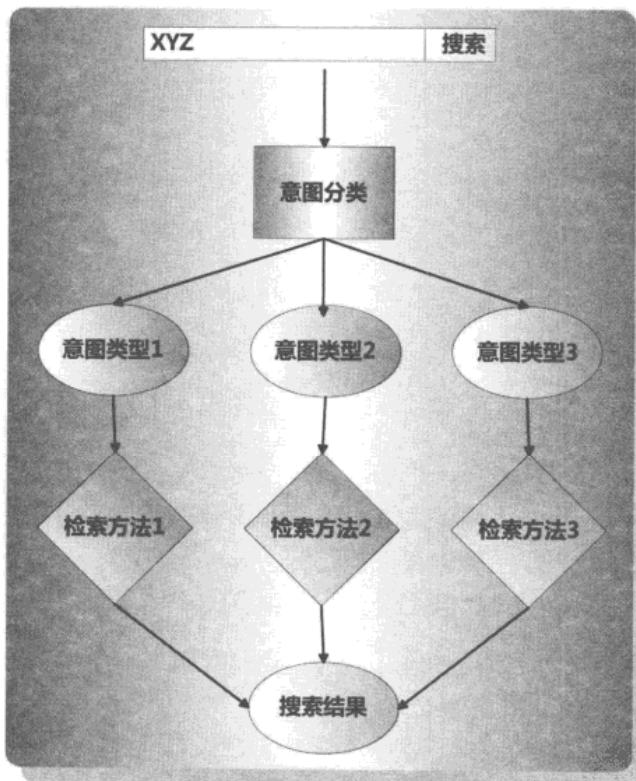


图 9-2 搜索意图分类

应该将用户意图分为哪些类型，目前并没有明确标准可言，不过很多工作都受到了 Broader 等人的意图分类工作的影响，他们通过人工分析查询，将搜索意图分为 3 个大的类别。

1. **导航型搜索 ( Navigational )**: 这种搜索请求的目的是查找具体的某个网站地址，比如著名的公司的网址或者机构的主页等，其特点是想要去某个网页。
2. **信息型搜索 ( Informational )**: 这种搜索请求的目的是为了获取某种信息，比如“如



何做宫保鸡丁”、“美国现任总统是谁”等，其特点是用户想要学到一些新知识。

3. **事务型搜索 ( Transactional )**: 这种搜索请求的目的是为了完成一个目标明确的任务，比如下载 MP3、下载软件或者在淘宝购物等，其特点是想要在网上做一些事情。

Broader 的搜索意图三分法非常有影响力，但是失之于粗糙。雅虎的研究人员在此基础上做了细化，将用户搜索意图划分为如下类别。

1. **导航类**: 用户知道要去哪里，但是为了免于输入 URL 或者不知道具体的 URL，所以用搜索引擎来查找。
2. **信息类**: 又可以细分为如下几种子类型。
  - **直接型**: 用户想知道关于一个话题某个方面明确的信息，比如“2010 年诺贝尔物理奖获得者是谁”或者“为何星星会发光”等。
  - **间接型**: 用户想了解关于某个话题的任意方面的信息，比如粉丝搜索“李宇春”等。
  - **建议型**: 用户希望能够搜索到一些建议、意见或者某方面的指导，比如“如何才能戒烟”等。
  - **定位型**: 用户希望了解在现实生活中哪里可以找到某些产品或服务，比如“买电话卡”等。
  - **列表型**: 用户希望找到一批能够满足需求的信息，比如“北京中关村附近的饭店”等。
3. **资源类**: 这种类型的搜索目的是希望能够从网络上获取某种资源，又可以细分为如下几种子类型。
  - **下载型**: 希望从网络某个地方下载想要的产品或者服务，比如“下载 Win7”等。
  - **娱乐型**: 用户出于消遣的目的希望获得一些有关信息，比如“下载电影”等。
  - **交互型**: 用户希望使用某个互联网软件或者服务提供的结果，比如“北京天气”，用户希望找到一个网站，在这个网站上可以查询北京的天气情况。
  - **获取型**: 用户希望获取一种资源，这种资源的使用场合不限于电脑，比如“折扣券”，用户希望搜到某个产品的折扣券，打印之后在现实生活中使用。

上述是对用户意图的人工整理分类，至于具体技术实现，则可以采取一些通用的分类算法比如 SVM、决策树等完成。

## 9.2 搜索日志挖掘

搜索日志（Query Log）是搜索引擎对用户行为的记录，通过记载用户行为，可以构建更好的算法以使得搜索结果更准确及更具有个性化色彩，搜索日志一般会记载用户发出的查询，发出查询的时间，点击过哪些搜索结果等数据。图 9-3 是一个典型的搜索日志片段。

查询	用户ID	时间	搜索排名	点击网址
尼康相机报价	98532	2011-03-05 00:01:58	2	product.pcpop.com/dc/00264_1.html
2011年运程	93775	2011-03-05 00:01:58	1	2011.sjzyxh.com
韩国劲爆舞曲	85007	2011-03-05 00:01:58	5	www.kugou.com/plist/playbill/2.htm
天涯社区	16197	2011-03-05 00:01:58	8	www.tianya.cn
姚明年薪	85530	2011-03-05 00:01:58	2	zhidao.baidu.com/question/43224630

图 9-3 一个典型的搜索日志片段

从图中可以看出，搜索日志每一条记录记载了查询、发出查询的用户 ID、发出查询的时间、点击网页的网址及这条网址在搜索结果中的排名情况。其中用户 ID 往往是根据用户使用浏览器访问时的 Cookie 信息获得的。

搜索日志包含了很多可用的信息，从中可以挖掘有价值的数据来帮助搜索引擎改善搜索质量，在使用搜索日志前，一般会对搜索日志进行整理，将原始形式的查询日志转换为意义更清晰的中间数据，常用的中间数据包括：查询会话、点击图及查询图。搜索引擎在这些中间数据基础上，可以充分利用用户查询信息来改善应用。

### 9.2.1 查询会话 (Query Session)

通过挖掘搜索日志，可以将同一用户在较短时间段内发出的连续多个查询找出，这样一段日志被称做一个查询会话。比如某个用户想要购买数码相机，在几分钟内连续向搜索引擎发出：“数码相机报价”、“尼康相机图片”、“尼康相机评价”等一系列查询，这形成了一个查询会话。

通过这种方式，可以把原始搜索日志转换为很多查询会话，查询会话内的搜索请求因为是同一用户在较短时间段内发出的，所以这些查询往往有一定的语义相关性。这种语义相关性可以被用来开发相关搜索推荐等具体的搜索应用。



## 9.2.2 点击图 (Click Graph)

除了查询会话外，另外一种常用的中间数据是点击图。从搜索日志记录可以看到，用户发出某个查询后，搜索引擎返回搜索结果，而用户会有选择地点击其中某些链接。这种用户点击行为是很有意义的，一般可以假设：用户之所以会点击这个网址，是因为用户在看了网页标题和搜索引擎摘要后，认为这个网页是和查询比较相关的，所以才会点击。也就是说，可以认为搜索结果里被点击过的网页与用户查询更相关。虽然这种假设并不总是成立，但是这类数据在实践中是非常有用的。

将查询和这个查询对应的点击网址联系起来，可以构建点击图，这是一种二分图，一端的图节点是所有用户发出的查询条件，另一端的图节点是互联网网页的网址，如果发出查询的用户点击过某个网址，则在查询节点和网址节点之间建立有向边，同时这个有向边可以设定权重，一般用点击次数来作为边的权重。

图 9-4 是用户点击图的一个示例，二分图左侧是用户查询，右侧是网页网址，比如对于用户查询“打折机票”来说，一般会点击“携程旅行网”（www.ctrip.com）和“去哪儿”（flight.qunar.com）的网页，所以在两类节点之间有边联系，同时点击次数可以作为边的权值。

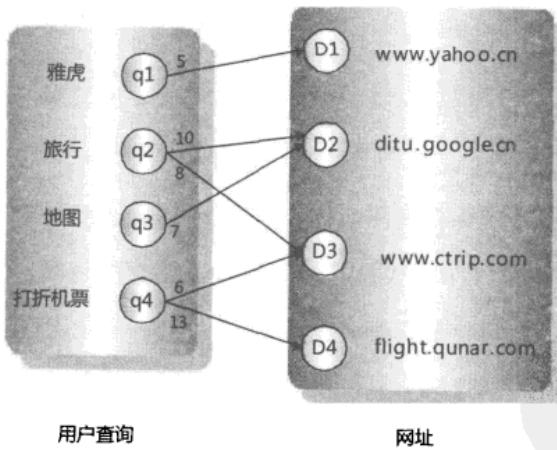


图 9-4 用户点击图的一个示例

点击图是非常有价值的信息，比如从图 9-4 可以看出，查询“旅行”和“打折机票”共同的点击网址是携程网，这从某种角度说明这两个查询是语义相关的，所以从点击图中可以挖掘大量语义相关信息。

### 9.2.3 查询图 (Query Graph)

用户发出的查询之间是有语义关联的，查询图就是试图构建查询之间相互关系的一种数据表示。图 9-5 给出了一个示例，图中的节点是用户发出的查询，而边上的权重则体现了查询之间的语义相关程度。

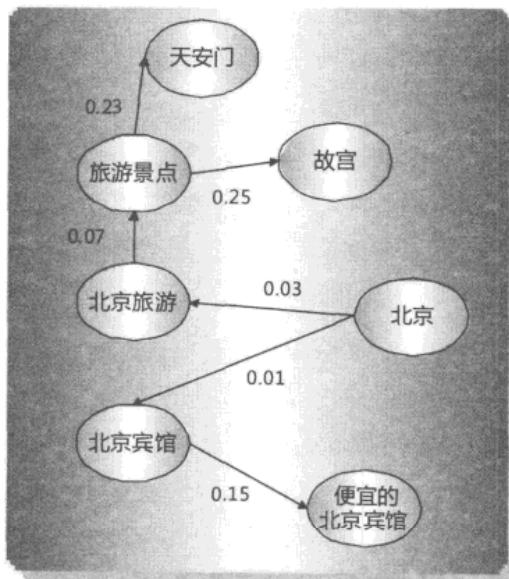


图 9-5 查询图示例

有若干种办法可以构建这种查询图，Baeza-Yates 提出了 5 种构建查询图的方法。

1. 可以用两个查询中重合的单词数目来衡量，重合程度越高，两个查询的相似性越大。
2. 如果两个查询属于同一个查询会话，则可以在两者之间建立联系。
3. 在点击图中如果两个查询有共同的点击网址，则可认为两者有语义关联。
4. 在点击图中，如果两个网址有链接关系，则其对应的查询可以建立语义关系。假设在图 9-4 中雅虎主页 [www.yahoo.cn](http://www.yahoo.cn) 有指向谷歌地图的链接，则查询“雅虎”和“旅行”可以在查询图中建立边。
5. 在点击图中，如果两个网址的页面内容主题相近，那么其对应的查询可以建立语义关系。比如在图 9-4 中，因为“携程旅行网”和“去哪儿”网站在页面内容上比较类似，所以与它们有关联的查询词语义相关，即可以在“旅行”和“打折机票”这两个查询之间建立语义联系。

利用查询图，可以发现查询之间的相似性，在此基础上，可以构建具体应用来使用这



种信息。

### 9.3 相关搜索

相关搜索也常被称做查询推荐，也就是说用户输入某个查询后，搜索引擎向用户推荐与用户输入查询语义相关的其他查询。用户对于自己的信息需求，不一定能够准确地想到合适的搜索词来表达搜索意图，相关搜索可以给用户提示，如果用户觉得搜索引擎推荐的查询更适合自己的搜索意图，那么会改善用户体验。

相关搜索目前已经是搜索引擎提供的标准配置功能，图 9-6 是输入“姚明”作为查询请求时，Google 的相关搜索提供的结果。

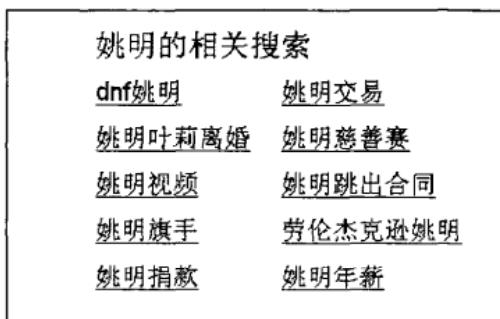


图 9-6 Google 对查询“姚明”的相关搜索

给定用户查询，搜索引擎如何计算相关查询呢？目前主流的做法有两类：基于查询会话的方法和基于点击图的方法。

#### 9.3.1 基于查询会话的方法

对于原始搜索日志，搜索引擎会对其做初步整理，而查询会话是其中一种整理方式。每个查询会话包含了某个固定用户在较短时间内连续发出的查询流，一般而言，在同一查询会话内的查询相互之间存在语义联系，而基于查询会话来进行相关搜索推荐的方法就是利用了这一点。

图 9-7 是这种方法的简明示意图。搜索引擎将原始搜索日志转换为大量的查询会话，之后采用关联规则挖掘等各种数据挖掘算法来对查询会话进行统计处理，挖掘结果往往是一批查询对 $\langle Q_x, Q_y \rangle$ ，这代表  $Q_x$  和  $Q_y$  在查询会话里是经常一起出现的，所以当用户输入其中某一个查询的时候，可以推荐给用户另外一个查询作为相关搜索结果。

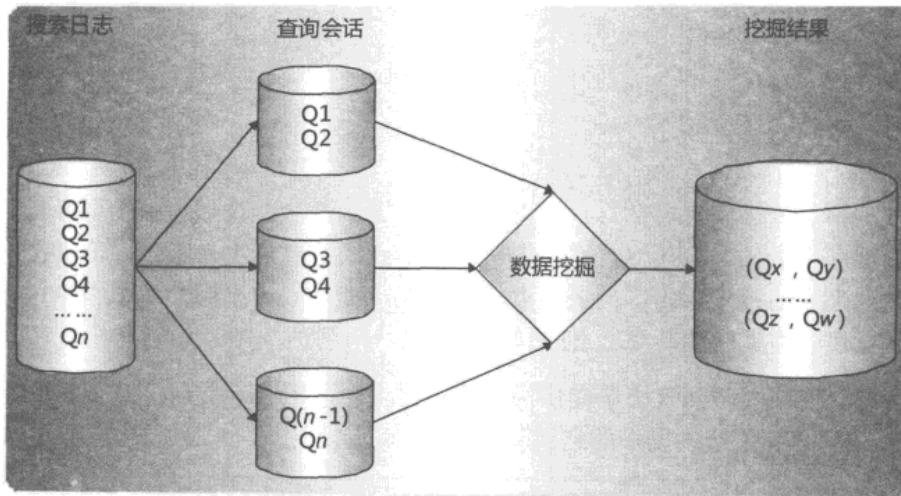


图 9-7 基于查询会话挖掘相关查询

这种方法思路简单，但是存在两个缺点：一个是查询会话的准确切割有一定难度；另外，因为查询会话是以同一用户的输入为基准的，所以在挖掘结果里不能体现不同用户的查询之间的关联。

### 9.3.2 基于点击图的方法

前文讲过，点击图是对原始搜索日志整理后的另外一种中间结果，很多算法使用点击图来进行查询推荐。尽管方法很多，但是其基本指导思想是相同的：如果两个查询各自对应的点击网址中，有很大比例是相同的，那么说明这两个查询在语义上紧密相关，可以作为相互推荐的相关查询。

图 9-8 是一个较为通用的利用点击图来探寻语义相关查询的方法示意图。总的来说，这种方法由两个步骤构成。首先，根据点击图可以将查询表示为其对应点击网址的权重向量，比如图中的  $q_2$  这个查询，因为用户发出这个查询后点击过  $D_2$  和  $D_3$  两个网址，且其点击次数分别是 10 和 8，所以可以构造向量  $[0, 10, 8, 0]$ ，向量的每一维度代表一个网址，以点击次数作为对应的权重。在做出以上处理后，就可以进行第 2 步，计算任意两个查询之间的相似度，在此基础上采用不同的聚类算法可以将查询聚合成大小不同的类别，被聚合到同一类别内的查询可以相互作为相关搜索的结果推荐给用户，比如在图 9-8 的例子中，用户输入查询  $q_2$ ，可以将  $q_3$  和  $q_4$  作为相关搜索的结果推荐给用户。

上述是一个通用的基本算法框架，不同的研究者提出了各种扩展算法，比如在设定查询的权重向量时，不仅考虑点击关系，还可以将点击网址的网页内容相似性考虑进来等。



同时，在聚类算法方面也可以采用不同的方法来实现具体系统。

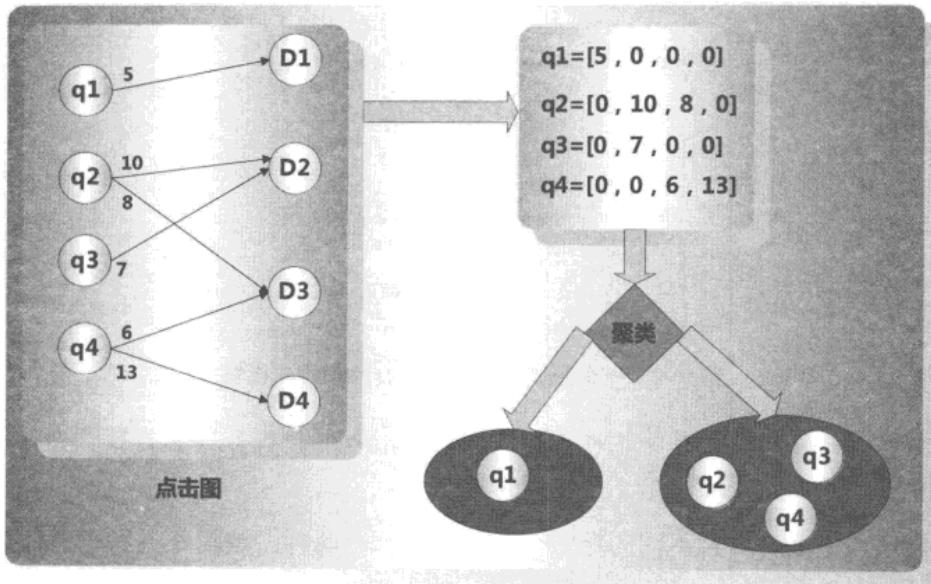


图 9-8 基于点击图的相关查询推荐

基于点击图的方法是非常常见的做法，但是其也有自身的缺点：由于点击图数据量非常庞大，一般聚类算法如何能够快速处理如此大量的数据是有挑战性的。

## 9.4 查询纠错

用户通过键盘敲入查询的时候，一种很常见的错误是输错字符，比如把“周鸿祎”错输成“周鸿一”。统计发现，搜索引擎查询中，大约有 10%~15% 的查询是错误输入的，所以如何能够自动纠正用户的错误查询，这对于搜索结果质量提升有很大帮助。

英文的输入错误主要有两大类型：一种是键盘敲击错误，比如把 i 错敲击为 u，或者是多敲击了一个字符等；另外一种是音节错误，两个单词发音相同但是输入是错误的，比如将 two 错输为 too 等。对于中文来说，往往需要经过中文输入法作为中介，所以更容易因选择了输入法的错误选项，导致错输入同音词，即更容易出现音节类型错误。

从大的流程上讲，查询纠错分为两个步骤。首先在众多的查询中，有正确的有错误的，如何识别哪些查询是错误的？这个需要错误识别机制来达到此目的。再者如果发现某个查询错了，如何找到正确的输入？这个需要错误纠正模块来实现此功能。图 9-9 展示了一个查询纠错的整体流程和构成模块。

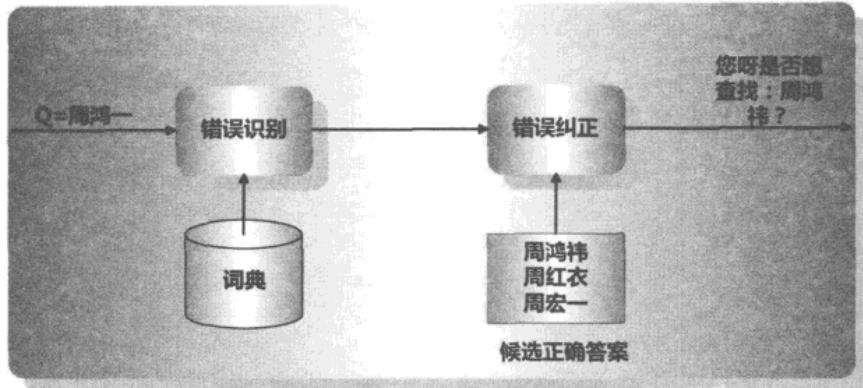


图 9-9 一个查询纠错的整体流程和构成模块

大多数错误识别机制是基于词典的，即将用户输入的查询分词后查找词典，如果在词典里没有找到，那么这很可能是一个错误输入。

至于错误纠正则有各种不同的算法，常见的方法有两种：编辑距离和噪声信道模型。

#### 9.4.1 编辑距离 (Edit Distance)

利用编辑距离来纠正拼写错误历史悠久，大约在 20 世纪 60 年代就开始采用这种方法。这是一种用来衡量两个字符串的拼写差异有多大的算法，对于某个字符串 S 来说，可以通过对其进行几种操作，来逐步将其转换为字符串 T，这些操作包括删除字符、添加字符、更改字符及交换字符顺序 4 种操作。转换过程中所需操作步骤越多，则其编辑距离越大，也意味着这两个字符串差异越大。图 9-10 给出了将错误单词 Mcirossat 转换为 Microsoft 的过程，从这个例子可以看出，两个单词的编辑距离为 4，也就是说从原始字符串转换为目标字符串使用了 4 步操作。如果需要计算的数据量比较大，编辑距离的效率会成问题。

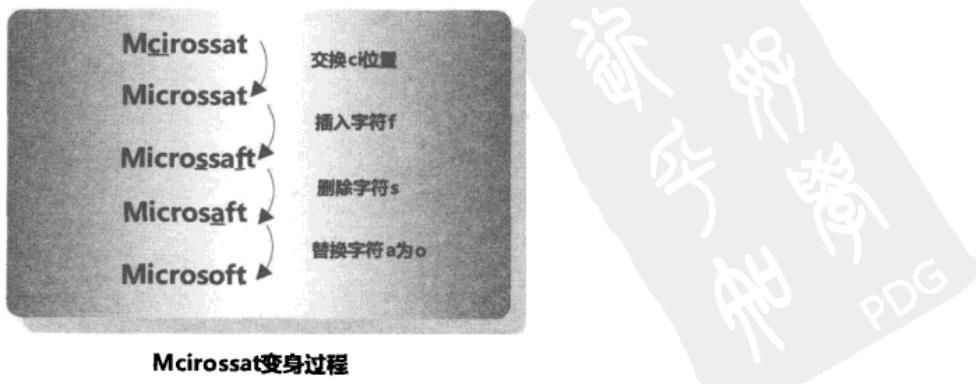


图 9-10 编辑距离示例



### 9.4.2 噪声信道模型 (Noise Channel Model)

噪声信道模型可以理解为正确的查询通过一个噪声信道传输，在传输过程中受到外界干扰，导致在信息接收端收到的查询发生错误，图 9-11 是噪声信道模型的原理示意。

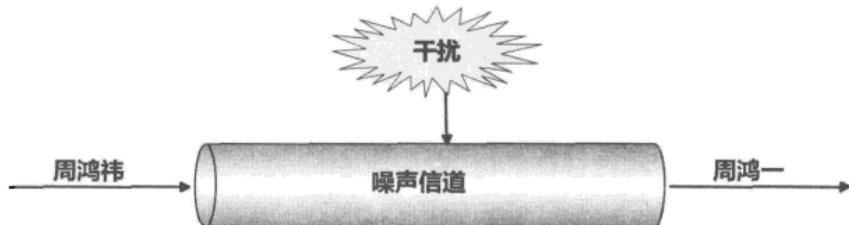


图 9-11 噪声信道模型的原理示意

给定错误查询  $V$  及其对应的可能正确查询  $W$ ，噪声信道模型将查询纠错任务转换成了估计概率  $P(W|V)$ ，其含义是：接收到错误的查询  $V$ ，这个查询对应的正确查询是  $W$  的概率有多大，对于多个候选正确答案，选择概率值最大的作为  $V$  对应的正确查询。将  $P(W|V)$  根据贝叶斯公式转写为：

$$P(W|V) = P(V|W) \times P(W) / P(V)$$

因为对于多个候选答案  $W_1, W_2, \dots, W_n$  来说，这个公式的分母  $P(V)$  都是相同的，所以可以忽略不计，于是问题转换成求：

$$\text{Argmax}(P(V|W) \times P(W))$$

就是说哪个候选答案  $W_i$  使得上面公式得分最大，那么这个  $W_i$  就是错误查询  $V$  对应的正确查询。

上述公式又可以分为两个因子： $P(V|W)$  与  $P(W)$ 。 $P(V|W)$  被称为错误模型，即正确查询  $W$  被错写成  $V$  的概率，具体计算方法有很多种，比如可以用上节提到的编辑距离计算，也可以收集一些被错误拼写的查询例子，用这些例子的统计数据来计算错误模型。 $P(W)$  被称为语言模型，可以通过计算查询  $W$  在所有查询中出现的概率值来估计。如果可以估计以上两个因子，就可以计算哪个  $W_i$  是错误查询  $V$  的正确输入。

## 本章提要

- 准确分析用户的搜索意图是目前搜索引擎研发的潮流与重点方向。
- 用户的搜索意图可以粗分为导航型、信息型和事务型。
- 搜索日志是深入挖掘用户潜在搜索意图最常用的数据来源，而点击图、查询会话

和查询图是由搜索日志整理出的基础数据。

- 相关搜索和查询纠错是非常常见的帮助用户澄清搜索意图的具体应用。

## 本章参考文献

- [1] Fonseca, B. M., Golgher, P. B., de Moura, E. S., and Ziviani, N.(2003). Using association rules to discover search engines related queries. In Proc. of the 1st Latin American Web Congress (LA-WEB'03).
- [2] Jansen, B. J., Spink, A., and Narayan, B.(2007). Query modifications patterns during web searching. in Proc. of 4<sup>th</sup> int. conf. on Information Technology (ITNG'07).
- [3] Boldi, P., Bonchi, F., Castillo, C., Donato, D., Gionis, A., and Vigna, S.(2008). The query-flow graph: model and applications. in Proc.of the ACM 17<sup>th</sup> Conf. on Information and Knowledge Management (CIKM'08).
- [4] Baeza-Yates, R., Hurtado, C., and Mendoza, M. (2004). Query recommendation using query logs in search engines. In International Workshop on Clustering Information over the Web (ClustWeb, in conjunction with EDBT), Crete, Greece.
- [5] Huang, C., et al. (2003) Relevant term suggestion in interactive web search based on contextual information in query session logs. Journal of the American Society for Information Science and Technology, 54(7):638-649.
- [6] Hosseini, M. and bolhassani, H. (2008). Clustering search engines log for query recommendation. CSICC, CCIS 6, pp. 380-387, Springer-Verlag Berlin Heidelberg 2008.
- [7] Damerau, F. (1964). A technique for computer detection and correction of spelling errors. Communications of the ACM. 7(3):171-176.
- [8] Brill, E. and Moore, R. (2000). An improved error model for noisy channel spelling correction. Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics. 286 - 293.

# 第 10 章 网页去重

“天之道，其犹张弓欤？高者抑之，下者举之；有馀者损之，不足者补之。天之道，损有馀而补不足。人之道，则不然，损不足以奉有馀。孰能有馀以奉天下，唯有道者。”

老子·《道德经》

统计结果表明，近似重复网页（Near Duplicate Web Page）的数量占网页总数的比例高达全部页面的 29%，而完全相同的页面大约占全部页面的 22%，即互联网页面中有相当大比例的内容是完全相同或者大体相近的。图 10-1 给出了一个示例，例子中的新闻主体内容是几乎完全相同的，但是两个页面的网页布局有较大差异，此种情况在互联网中非常常见。

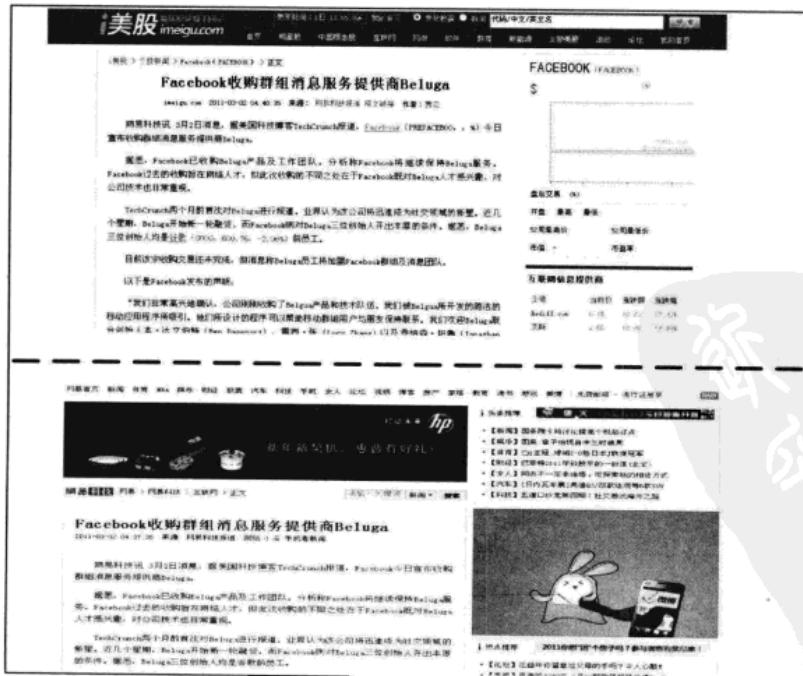


图 10-1 近似重复网页示例

近似重复网页有多种类型，这些重复网页有的是没有一点儿改动的副本，有的在内容上稍做修改，比如同一文章的不同版本，一个新一点，一个老一点，有的则仅仅是网页的格式不同（如 HTML、Postscript）。内容重复可以归结为以下 4 种类型。

- **类型一：**如果两篇文档内容和布局格式上毫无差别，则这种重复可以叫做完全重复页面。
- **类型二：**如果两篇文档内容相同，但是布局格式不同，则叫做内容重复页面。
- **类型三：**如果两篇文档有部分重要的内容相同，并且布局格式相同，则称为布局重复页面。
- **类型四：**如果两篇文档有部分重要的内容相同，但是布局格式不同，则称为部分重复页面。

所谓近似重复网页发现，就是通过技术手段快速全面发现这些重复信息的手段，如何快速准确地发现这些内容上相似的网页已经成为提高搜索引擎服务质量的关键技术之一。

发现完全相同或者近似重复网页对于搜索引擎有很多好处。

1. 首先，如果我们能够找出这些重复网页并从数据库中去掉，就能够节省一部分存储空间，进而可以利用这部分空间存放更多的有效网页内容，同时也提高了搜索引擎的搜索质量和用户体验。
2. 其次，如果我们能够通过对以往收集信息的分析，预先发现重复网页，在今后的网页收集过程中就可以避开这些网页，从而提高网页的收集速度。有研究表明重复网页随着时间不发生太大变化，所以这种从重复页面集合中选择部分页面进行索引是有效的。
3. 另外，如果某个网页的镜像度较高，往往是其内容比较受欢迎的一种间接体现，也就预示着该网页相对重要，在收集网页时应赋予它较高的优先级，而当搜索引擎系统在响应用户的检索请求并对输出结果排序时，应该赋予它较高的权值。
4. 从另外一个角度看，如果用户点击了一个死链接，那么可以将用户引导到一个内容相同页面，这样可以有效地增加用户的检索体验。因而近似重复网页的及时发现有利于改善搜索引擎系统的服务质量。

实际工作的搜索引擎往往是在爬虫阶段进行近似重复检测的，图 10-2 给出了近似重复检测任务在搜索引擎中所处流程的说明。当爬虫新抓取到网页时，需要和已经建立到索引内的网页进行重复判断，如果判断是近似重复网页，则直接将其抛弃，如果发现是全新的内容，则将其加入网页索引中。

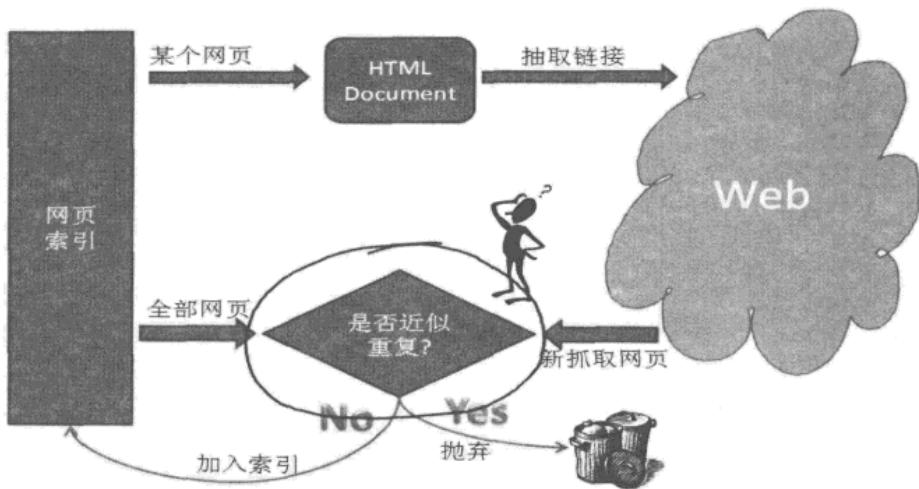


图 10-2 近似重复检测任务在搜索引擎中所处流程说明

## 10.1 通用去重算法框架

对于网页去重任务，具体可以采取的技术手段五花八门，各有创新与特色，但是如果仔细研究，大部分算法的整体流程和框架有诸多相似之处，本节参考一些实践效果较好的去重算法，并归纳整理，总结了相对通用的算法框架。尽管很多算法看似迥异，其框架实则雷同，这与去重任务本身的要求有密切关系，即需要算法能够对海量数据进行快速处理。

图 10-3 给出了这个通用算法框架的流程图，对于给定的文档，首先通过一定的特征抽取手段，从文档中抽取出一系列能够表征文档主体内容的特征集合。这一步骤往往有其内在要求，即尽可能保留文档重要信息，抛弃无关紧要的信息。之所以要抛弃掉部分信息，主要是从计算速度的角度考虑的，一般来说，抛弃的信息越多，计算速度会越快，但是如果抛弃得过多，在此步骤可能会丢失重要信息，所以不同的算法在此步骤需要做出权衡，在速度和准确性方面要通盘考虑，尽可能兼顾两者。

在将文档转换为特征集合后，很多算法就可以直接进入查找相似文档的阶段，但是对于搜索引擎来说，所要处理的网页数量以亿计，算法的计算速度至关重要，否则算法可能看上去很美，但是无实用效果。为了能够进一步加快计算速度，很多高效实用的算法会在特征集合的基础上，对信息进一步压缩，采用信息指纹相关算法，将特征集合压缩为新的数据集合，其包含的元素数量远远小于特征集合数量，有时候甚至只有一个文档指纹。在此处与在特征抽取阶段一样，有可能会有信息丢失，所以也需权衡压缩率和准确性的问题。

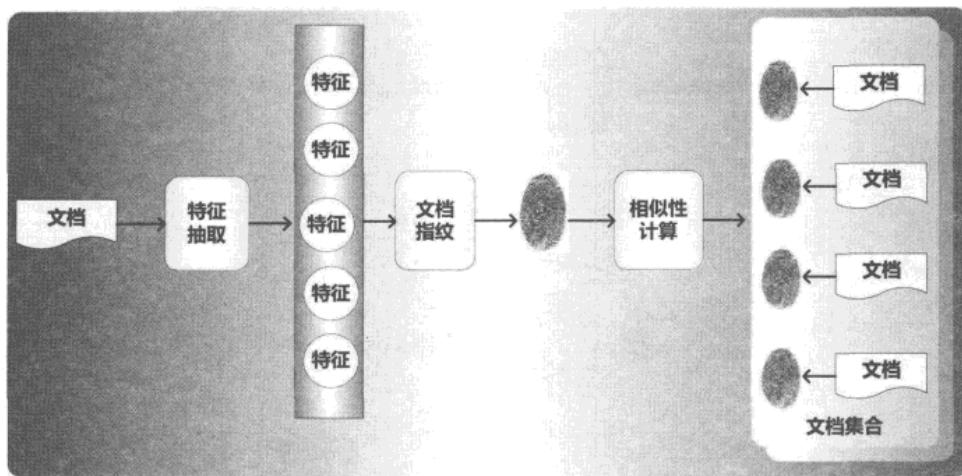


图 10-3 通用的网页去重算法流程框架

当把文档压缩为文档指纹后，即可开始通过相似性计算来判断哪些网页是近似重复页面。对于去重来说，最常用的文本相似性计算是 Jaccard 相似度，大部分去重算法都是以此作为评估两个文档是否近似的标准。另外，由于数据量太大，在计算相似性的时候，如果一一进行比较显然效率很低，在此处不同算法往往采用各种策略来加快相似性匹配过程，比较常见的做法是对文档集合进行分组，对于某个文档，找到比较相似的分组，和分组内的网页进行一一比较，这样可以大大减少比较次数，有效提升系统效率。

上述的通用去重算法框架看上去比较抽象，读者在阅读完后续介绍的具体算法后再次返回本节对照，可以获得更深刻的理解，对此框架的深入理解有助于开发者研发新的高效去重算法，应该意识到：之所以框架如此，是有其深刻原因的。

## 10.2 Shingling 算法

Shingling 算法可以被视为由两个大的步骤组成：第 1 步从文档中抽取能够代表文档内容的特征，第 2 步则根据两个文档对应特征集合的重叠程度来判断是否近似重复。

之所以被称为 Shingling 算法，是因为该方法以 Shingles 作为文档的特征。所谓 Shingles，即将文档中出现的连续单词序列作为一个整体，为了方便后续处理，对这个单词片段进行哈希计算，形成一个数值，每个单词片段对应的哈希值称为一个 Shingle，而文档的特征集合就是由多个 Shingle 构成的。

图 10-4 是 Shingling 算法如何将一篇文本文档转换为特征集合的示意图，可以假想有一个固定大小的移动窗口从文档第 1 个单词（单字）开始依次移动，每次向后移动一个单



词（单字），直到文本末尾。图 10-4 中是以 3 个汉字作为移动窗口的大小，所以第 1 个长度为 3 的汉字串是“新浪发”，对这个汉字串进行哈希计算（Shingling 算法在此处采用 Rabin FingerPrint 算法），即得到一个 shingle，然后窗口向后移动一个汉字，形成第 2 个汉字串“浪发布”，同样对汉字串进行哈希计算，得到第 2 个 shingle，依此类推，窗口不断后移，直到末尾的汉字串“户破亿”为止，这样所有的 shingle 组成的集合就是文档对应的特征集合。

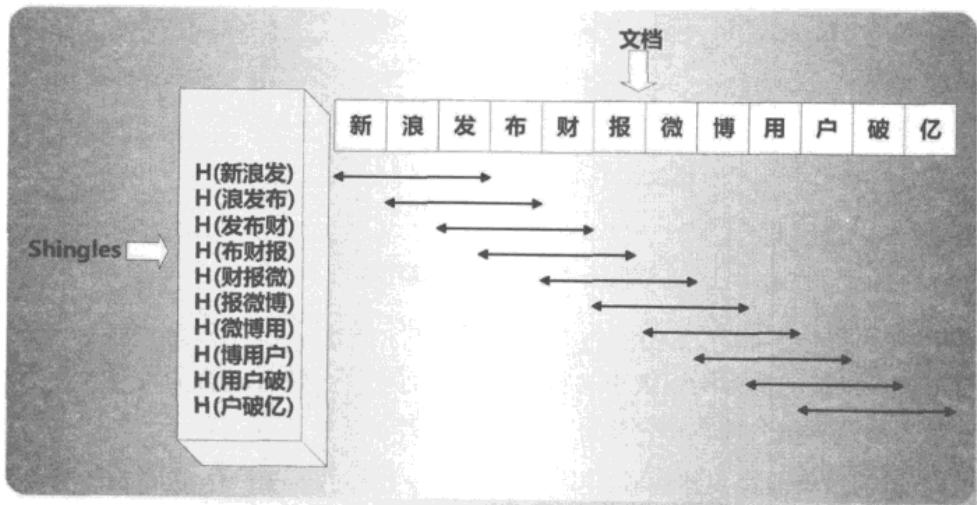
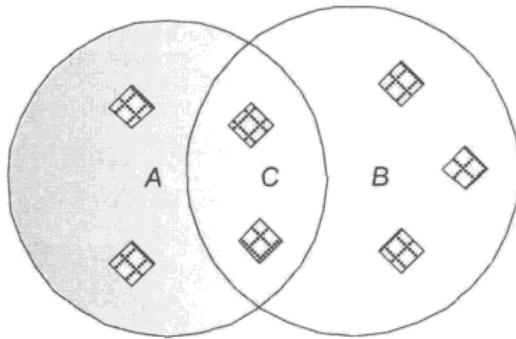


图 10-4 用 Shingling 算法将一篇文本文档转换为特征集合的示意图

如果每个文档都通过如上方式转换为特征集合，如何计算两个文档是否是近似重复网页？Shingling 算法考察两个特征集合的重叠程度，重叠程度越高，则越可能是近似重复网页。具体而言，采用了 Jaccard 相似性来计算这个重叠程度。

图 10-5 是 Jaccard 相似性计算的示意图，这是一种计算集合相似性的经典方法，对于两个集合  $A$  和  $B$  来说，两者的重叠部分由  $C$  来表示。图中集合  $A$  包含 4 个元素，集合  $B$  包含 5 个元素，而两者相同的元素有 2 个，即集合  $C$  的大小为 2。Jaccard 计算两个集合相同的元素占总元素个数的比例，因为图 10-5 中集合  $A$  和  $B$  共有 7 个不同元素，相同元素个数为 2，所以集合  $A$  和集合  $B$  的相似性即为  $2/7$ 。

Shingling 算法通过以上两个步骤即可计算哪些网页是近似重复网页，但是这种方法在实际运行时，计算效率并不高，如果网页数量大，运行时间会过长，并不实用。原因在于把一个文档转换为以 shingles 表示的特征集合形式后，这个文档对应的特征集合仍然太大。同时对于不同长度的文档来说，转换后的特征集合大小各异。而这两点对于高效计算来说都是不利因素。为了加快计算速度，能否将文档的特征集合变为固定长度，同时使得这个长度远远小于原始的特征集合？



$$\text{Jaccard}(A, B) = \frac{|C|}{|A + B|}$$

图 10-5 文档相似性计算

Fetterly 等人提出了针对原始 Shingling 算法改进的算法，其基本思想即如上所述，对于不同的网页，将其转换为固定大小的特征集合，而且这个特征集合的大小要远小于原始 Shingling 转换后特征集合的大小，以此手段来大大提升运算效率。

图 10-6 即为这个改进思路的示意图。前面若干计算过程与原始 Shingling 算法是一致的，即首先将一个文档转换为由 shingles 构成的特征集合。为了能够将文档特征映射为固定大小，引入  $m$  个不同的哈希函数，形成哈希函数簇。对于某个特定的哈希函数  $F$ ，对每个 shingle 都计算出一个对应的哈希数值，取其中最小的那个哈希数值作为代表。这样  $m$  个哈希函数就获得了  $m$  个哈希数值，如此就把文档的特征集合转换为固定大小  $m$ ，同时这个数值也比很多由 shingles 构成的特征集合小很多。通过如上方式，即可把文档对应的特征集合映射为一个固定大小，而且长度比原始方法小很多的数值向量，以加快后续相似度计算的速度。

图 10-6 中为了方便说明问题，哈希函数簇只包含了两个哈希函数，而实际使用的时候往往使用 84 个不同的哈希函数，即将一个文档映射成为由 84 个数值构成的数值向量。为了进一步加快计算速度，可以将 84 个数值进一步压缩：以 14 个连续数值作为一块，将 84 个数值分为 6 块，利用另外一个哈希函数对每一块的 14 个数值进行哈希计算，进一步将文档特征转换为 6 个哈希数值，如果任意两个文档有两个以上的哈希数值是相同的，即可认为是近似重复文档，这个技巧被称为 SuperShingle。

至于计算文档集合的 Jaccard 相似性，一般会采用 Union-Find 算法。Union-Find 算法是经典的计算等价类的高效算法，参考文献众多，此处即不赘述其细节，重点仍然放在去重算法本身。

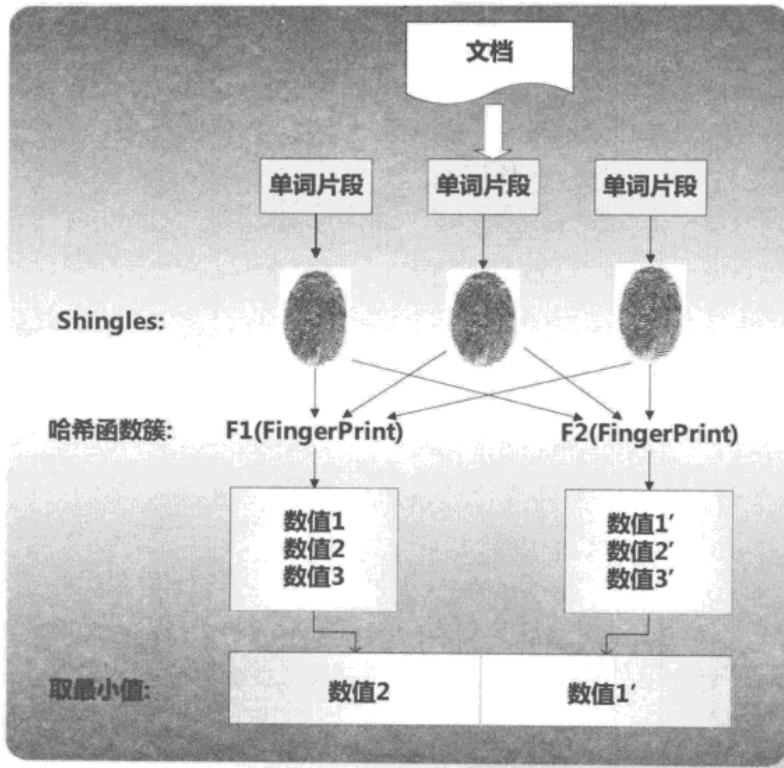


图 10-6 改进的 Shingling 算法思路示意图

经过如上诸般优化措施，改进的 Shingling 算法计算效率已非常高。实验表明，计算一亿五千万个网页，该方法可以在 3 小时内计算完毕，而原始的 Shingling 算法即使是处理三千万网页，也需 10 天才可完成，应该说速度的提升是非常显著的。

### 10.3 I-Match 算法

最初的 I-Match 算法是由 Abdur 等人于 2002 年提出的，其基本流程也遵循本章第一节所述的通用去重算法框架。

图 10-7 是 I-Match 算法流程的示意图。对于该算法来说，非常重要的一个步骤是事先计算出一个全局的特征词典，具体到 I-Match 算法来说，则是根据大规模语料进行统计，对语料中出现的所有单词，按照单词的 IDF 值由高到低进行排序，之后去除掉一定比例 IDF 得分过高及得分过低的单词，保留得分处于中间段的单词作为特征词典，实验表明以这些单词作为特征，其去重效果较好。

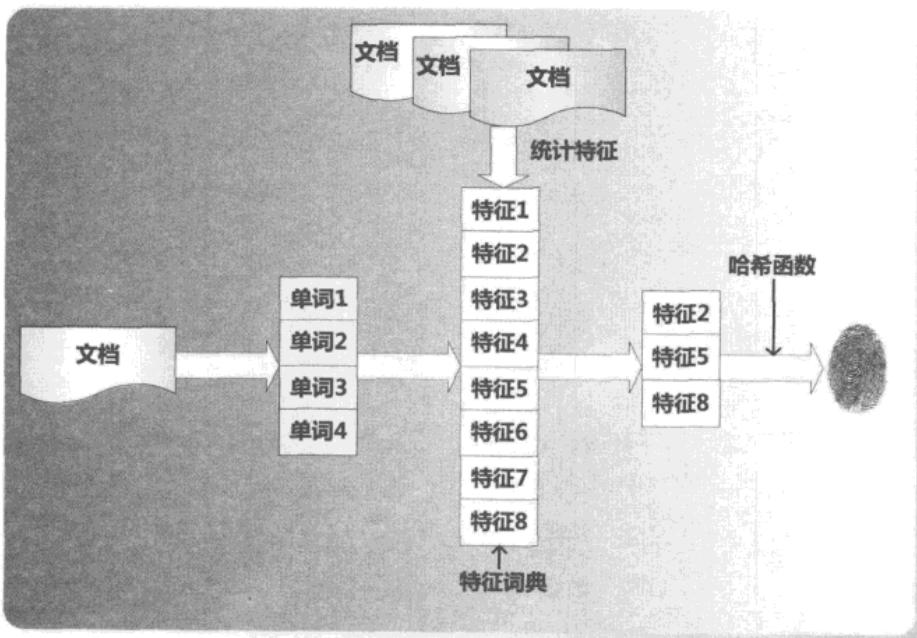


图 10-7 I-Match 算法流程的示意图

获得全局的特征词典后，对于需要去重的网页，扫描一遍即可获得在该页面中出现过的所有单词，对于这些单词，用特征词典进行过滤：保留在特征词典中出现过的单词，以此作为表达网页内容的特征；没有在特征词典中出现过的单词则直接抛弃。通过这种方式，抽取出文档对应的特征，之后利用哈希函数（I-Match 算法采取 SHA1 作为哈希函数）对文档的所有特征词汇整体进行哈希计算，得到一个唯一的数值，以此哈希数值作为该网页的信息指纹。

对网页集合里所有网页都计算出相应的信息指纹后，如何判断两个网页是否是近似重复网页？I-Match 算法于此很直观，可以直接比较两个网页对应的信息指纹，如果两者相同，则被认为是近似重复网页。

回顾上节所讲 Shingling 算法的特征抽取过程，从上述对应的 I-Match 算法的特征抽取过程可以看出，I-Match 算法抽取出的文档特征是一个个独立的单词，单词之间的顺序没有被考虑进来，所以 I-Match 算法对于文档之间单词顺序的变化并不敏感，如果两个文档所包含的单词相同，但是单词顺序进行了变换，I-Match 算法一定会将其算做重复内容。

I-Match 算法的优点在于其效率很高，因为每个文档被映射为单一的哈希值，以单一数值作为文档的表征，必然在计算速度上优于多值表征，因为可以避免复杂的集合运算。

但是 I-Match 算法也包含不少问题，首先，对于短文本来说，很容易出现误判，也就

是说两个文档本来不是近似重复网页，但是 I-Match 算法容易将两者判断为重复内容。之所以会如此，原因就在上文提到的特征词典，假设两个短文本内容并不相似，但是经过特征词典过滤后，只能保留很少几个单词作为文档的特征，而如果这几个单词是相同的，那么自然会将这两个文档误判为近似重复网页。其根本原因在于特征词典覆盖不足，导致文档很多信息被过多过滤，对于短文本这个问题尤其严重。

另外一个更加突出的问题是，I-Match 算法的稳定性不好。所谓稳定性不好，指的是对于某个文档 A 做了一些较小的内容变动，形成新文档 B，本来应该将两者看做近似重复文档，但是 I-Match 算法很可能无法将其计算为我们希望的结果，即 I-Match 算法对于增删单词这种变化比较敏感，这是由于 I-Match 算法所采用的特征词典机制和 SHA1 哈希算法共同导致的。

我们可以考虑如下的极端情形：对于某个文档 A，我们向其中加入一个新的单词 w（即 w 没有在 A 中出现过），形成文档 B。通过 I-Match 算法的特征词典对两个文档进行特征过滤，因为两者的差别只有这个新加入的单词 w，所以如果单词 w 不在特征词典中，那么文档 A 和文档 B 的对应特征集合相同，所以哈希后的信息指纹也一定相同，I-Match 会认为两个文档是近似重复文档，这是我们想要的结果；但是，如果单词 w 出现在特征词典中，那么文档 B 的特征集合会比文档 A 多一个特征，即单词 w，而 SHA1 哈希算法对于这种差异很敏感，会将两个文档映射成两个不同的信息指纹，即出现了稳定性不佳的问题。很明显，这个问题是由特征词典和 SHA1 哈希算法共同决定的，可以看做是 I-Match 算法为了计算效率所付出的代价。

为了解决原始 I-Match 算法存在的稳定性不佳问题，Kolcz 等人提出了改进算法（参考图 10-8）。其基本出发点也很直观：原始 I-Match 算法对于文档内容改变过于敏感，原因在于其严重依赖于特征词典的选择，为了减少这种依赖性，可以考虑同时采用多个特征词典，而每个特征词典大体相近，同时又必须有微小的差异。

对于某个需要判别是否重复的文档 A，对应每个特征词典，生成多个信息指纹。如果向文档 A 增加新的单词 w 形成文档 B，因为存在多个大致相同但有微小差异的特征词典，所以有很大可能某个特征词典不包含这个单词，所以通过这个特征词典算出的文档 B 的信息指纹和文档 A 是相同的。在判断 A 和 B 两个文档是否重复时，同时考虑多个信息指纹的情况，只要两个文档对应的众多信息指纹中有任意一个是相同的，则可以判定两者是重复文档。这样就解决了 I-Match 算法对增删单词过于敏感的问题。

那么如何形成多个“大致相同又有微小差异”的特征词典呢？Kolcz 是如此解决这个问题的：类似于原始 I-Match 算法，形成一个特征词典，为了和其他词典区分，可以称为主特征词典；然后根据主特征词典衍生出其他若干个辅助特征词典，为了能够达到词典内容

大致相同，又能有微小差异，可以考虑从主特征词典中随机抽取很小比例的词典项，之后将其从主特征词典中抛弃，剩下的词典项构成一个辅助特征词典，如此重复若干次就可以形成若干个辅助特征词典，这些辅助特征词典和主特征词典一起作为算法采用的多个特征词典。通过如此做法，即可保证每个词典大致内容相同，其间又有微小差异，能够达到所期望的效果。

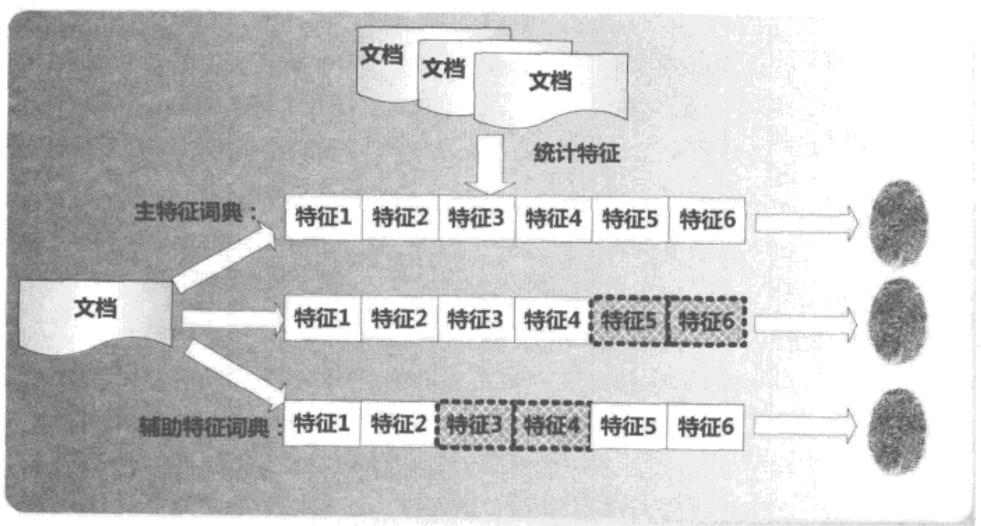


图 10-8 对 I-Match 算法的改进

图 10-8 中演示了这个过程，图中包含两个从主特征词典衍生的辅助特征词典，其中一个抛弃了主特征词典的特征 5 和特征 6，另外一个则抛弃了特征 3 和特征 4，如此就形成了 3 个特征词典。对于某个文档，根据 3 个特征词典分别形成 3 个信息指纹，如果两个文档有任何一个信息指纹相同，则可以判定为重复文档。

原始的 I-Match 算法将文档映射成唯一的信息指纹，虽然增加了计算效率，但是明显存在信息表达不足的问题，改进的 I-Match 算法本质上是将一个文档映射成多个信息指纹，可以认为是将文档里更多的信息进行了编码，这种做法与前述章节的 SuperShingle 的做法类似，SuperShingle 也是将文档压缩成多个信息指纹，区别在于：SuperShingle 是将信息由多到少进行进一步压缩，而改进的 I-Match 算法是从唯一的信息指纹将信息由少到多进行扩展。虽是殊途，毕竟同归。

## 10.4 SimHash 算法

经过实践证明，SimHash 算法可能是目前最优秀的去重算法之一，Google 内部应该采



用以 SimHash 算法为基础的改进去重方法来对网页进行预处理，而且已对此算法申请了专利保护。

严格来说，SimHash 算法可以看做是局部敏感哈希框架（Locality Sensitive Hashing Schema）的一个实现特例。经过理论分析，本章前述章节讲到的“改进的 Shingling 算法”引入多个哈希函数，究其本质，也是局部敏感哈希框架的一个具体实现方式而已。

局部敏感哈希框架之所以在海量文本处理方面大行其道，源于其有趣的特性：两个文档内容越相似，则其对应的两个哈希值也越接近，所以可以将文本内容相似性问题转换为哈希值的相近性问题。而利用哈希值，很明显比文本计算速度快得多，同时用哈希值表示文档，也大大节省了存储空间。这与一般哈希函数的使用目的截然相反，一般哈希函数为了减少冲突，尽可能均匀地将哈希值分布到不同数值空间。

SimHash 算法也可以划分为两个步骤：文档指纹计算和相似文档查找。文档指纹计算的目的是将一篇文本文档转换为固定大小的二进制数值，以此作为文档的信息指纹，相似性查找阶段则根据信息指纹来找出哪些文档是近似重复的。

#### 10.4.1 文档指纹计算

图 10-9 是 SimHash 算法第 1 阶段的具体流程图，通过这个步骤将文档转换为二进制表示的文档指纹。其内容转换过程又可分为如下几个步骤。

首先，从文档内容中抽取一批能表征文档的特征，至于具体实现，则可以采取不同的抽取方法，经过此步骤，获得文档的特征及其权值  $w$ 。

之后，利用一个哈希函数将每个特征映射成固定长度的二进制表示，如图 10-9 所示为长度等于 6 比特的二进制向量，这样每个特征就转换为 6 比特二进制向量及其权值。

接下来，利用权值改写特征的二进制向量，将权重融入向量中，形成一个实数向量。假设某个特征的权值是  $w$ ，则对二进制向量做如下改写：如果二进制的某个比特位是数值 1，则实数向量中对应位置改写为数值  $w$ ；如果比特位数值为 0，则实数向量中对应位置改写为数值  $-w$ ，即权值的负数。通过以上规则，就将二进制向量改为体现了特征权重的实数向量。

当每个特征都进行了上述改写后，对所有特征的实数向量累加获得一个代表文档整体的实数向量。累加规则也很简单，就是将对应位置的数值累加即可。

最后一步，再次将实数向量转换为二进制向量，转换规则如下：如果对应位置的数值大于 0，则设置为二进制数字 1；如果小于等于 0，则设置为二进制数字 0。在如图 10-9 所示的实例中，6 个数值再次转换为长度为 6 比特的二进制数值 110001。如此，就得到了文

档的信息指纹，即最终的二进制数值串。

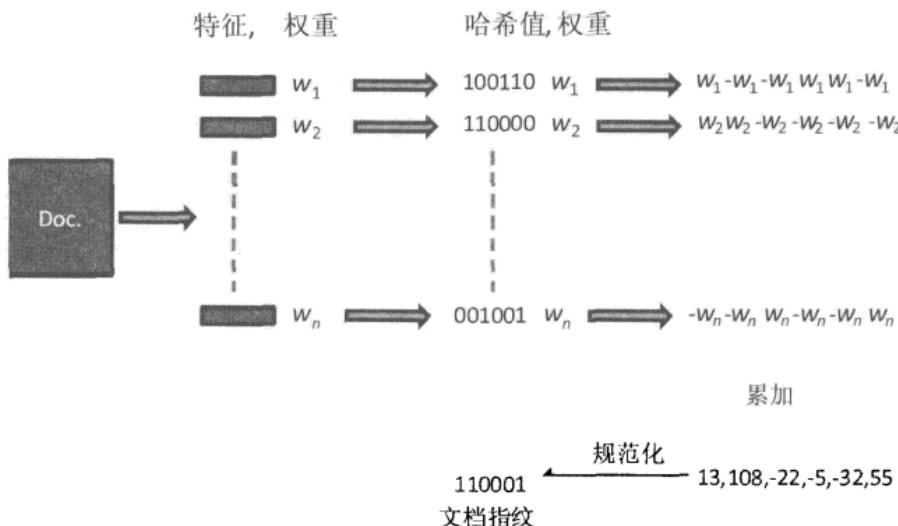


图 10-9 SimHash 算法第 1 阶段的具体流程图

#### 10.4.2 相似文档查找

对每个文档都按照上述规则进行映射，将文档转换为固定大小的二进制数值，在实际计算中，往往会将长度设定为 64，即每个文档转换为 64 比特的二进制数值。

对于两个文档 A 和 B，其内容相似性可以通过比较二进制数值的差异来体现，内容越相似，则二进制数值对应位置的相同的 0 或者 1 越多，两个二进制数值不同的二进制位数被称为“海明距离”。比如假设文档 A 的二进制表示为 1000001，而文档 B 的二进制表示为 1100001，则只有第 2 个位置的二进制数字不同，所以其海明距离为 1。不同的二进制数字个数越多，即海明距离越大，则文档越不相似，一般对于 64 位二进制数来说，判断两个文档是否近似重复的标准是：海明距离是否小于等于 3，如果两个文档的二进制数值小于等于 3 位不同，则判定为近似重复文档。

海量的网页经过上述步骤，转换为海量的二进制数值，此时如果新抓取到一个网页，如何找出近似重复的内容？

一个很容易想到的方式是一一匹配（图 10-10），将新网页 Q 转换为 64 比特的二进制数值，之后和索引网页一一比较，如果两者的海明距离小于等于 3，则可以认为是近似重复网页。这种方法虽然直观，但是计算量过大，所以在以亿计的网页中，实际是不太可行的。

为了加快比较速度，SimHash 采取了变通方法，其本质思想是将索引网页根据文档指

纹进行分组，新网页只在部分分组内进行匹配，以减少新文档和索引网页的比较次数。

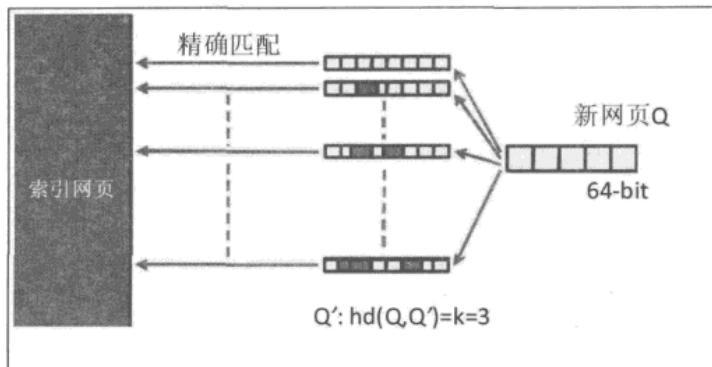


图 10-10 指纹一一比较法

图 10-11 展示了这种思想的具体实现方法，首先对于 64 位长度的二进制数值进行分块，每 16 位作为一块，这样每个二进制数值被划分为 4 块，可分别以 A、B、C、D 块来命名。对于海量的索引网页，依据分块进行聚类，比如对于 A 块来说，根据其 A 块内 16 位二进制聚类，如果 16 位二进制都相同，则这些网页被看做是一个聚类，即一组，这样根据 A 块就可以将所有索引网页分成若干组数据。对于 B、C 和 D 来说也是如此，即相同的 16 位二进制网页作为一个分组。如此，就将所有索引网页聚合成很多组小的数据集合，每一组必有连续 16 位二进制数字是相同的。

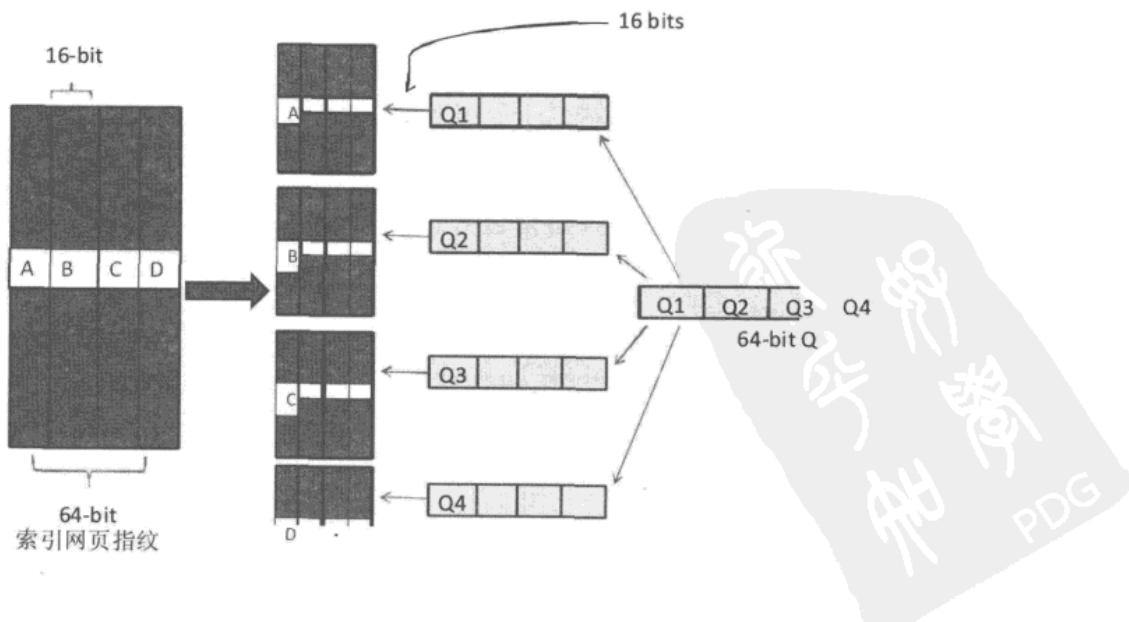


图 10-11 分组分块匹配法

对于新抓取的网页，同样将 64 比特二进制数据分为 4 块：Q1、Q2、Q3、Q4。在索引网页的分组中，找到对应 A 块 16 位和 Q1 完全相同那个分组，之后与分组内的网页一一比较来查找哪些网页是近似重复的。对于 Q2、Q3 和 Q4 也做同样处理。这样就可以用较少的代价，找到全部索引网页中和新抓取网页近似重复的内容。

## 10.5 SpotSig 算法

SpotSig 是个很有趣的算法，最初算法设计者是为了解决新闻内容的近似重复判断而提出这个方法的。

这个算法基于如下观察：很多新闻的主体内容大致相同，但是页面布局往往差异很大，比如有很多导航链接或者广告链接及其文字。那么，新闻主体和页面布局区域在使用文字上有何种明显差异呢？一个很明显的差异是：停用词在两者中的分布是不一样的，新闻正文中停用词一般是比较均匀而频繁出现的，但是在其他区域中却很少出现停用词。SpotSig 提出者觉得这个差异可以被用来识别近似重复新闻。

SpotSig 算法整体框架也符合本章第 1 小节所述，但是在具体实现方法上有独特之处。

### 10.5.1 特征抽取

观察到上述的新闻正文和其他区域中停用词分布的差异，SpotSig 算法考虑用停用词相关的词语片段作为文档的特征。图 10-12 给出了一个特征提取的实例。

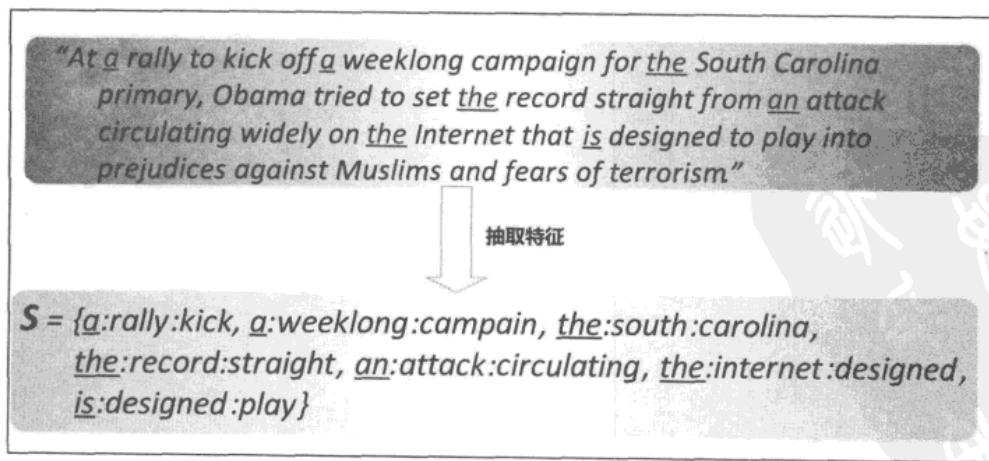


图 10-12 特征提取的实例

首先确定要使用的部分停用词，并以这些停用词在网页中出现的位置作为锚点，在图



10-12 的例子中，选取单词 a、an、the、is，以这 4 个停用词作为锚点。从锚点向后顺序扫描，取固定个数的单词作为特征构成部分，比如例子中第 1 次出现的 a，后面紧跟着单词序列“rally to kick”，假设锚点后跟单词个数设定为 2，即取单词 rally 和 kick 作为这个锚点的特征组成部分，将停用词 to 过滤掉，这样 a:rally:kick 就组成了第 1 个特征。按照同样的方式可以获得文档所有的特征，组成特征集合，以此来表征这个文档。

文档特征是根据停用词来选取的，如果两个网页内容相同，只是布局结构不同，因为页面布局文字很少包含停用词，所以从页面布局内几乎不会抽出特征。也就是说，页面布局对于判断后续内容近似文档基本没有负面影响，这也是 SpotSig 算法一个显著的特色。

另外一个与前述去重算法的不同之处在于：SpotSig 算法直接以文本串作为特征，而正如前述章节所讲，其他算法大都会将文本特征转换为数字特征，而 SpoSig 并未如此做。

### 10.5.2 相似文档查找

获得文档的特征集合后，与 Shingling 算法类似，SpotSig 采用 Jaccard 来计算文档之间的相似性。考虑到计算量过大，SpotSig 采取了两个技术手段来加速查找过程：文档分组和倒排索引。

如果对文档分组，给定一个待判网页，则无须和全部索引网页一一比较，只需要找到合适的分组，与分组内网页比较即可，这样确实可以极大地提升系统速度。上节所述 SimHash 也采取了类似分组的思路来加快计算速度。

但是如何对文档分组？其依据是什么呢？SpotSig 算法对网页集合进行分组是基于如下观察：因为采用的是 Jaccard 公式来计算两个文档的相似性，而且我们只需要找到文档特征重叠度足够高的相似文档即可，那么对于相似性得分可能很低的文档，则尽可能不去匹配。仔细考察 Jaccard 公式，可以得出结论，如果两个文档 A 和 B，其长度相差太大，利用 Jaccard 公式计算出的两者的相似性一定很低，即两者不可能是近似重复文档。所以对于某个待判文档 A 来说，与其相似的文档，长度一定与文档 A 的长度相差不远。既然如此，可以在文档转换为特征集合后，根据特征集合的大小对文档集合进行分组，而在进行相似性计算时，只在合适的分组内一一比较，以此加快查找速度。

为了进一步加快匹配速度，SpotSig 算法对于每个分组内的网页，分别建立一套倒排索引，如图 10-13 所示是分组 k 建立的倒排索引片段。对于每个特征，建立倒排列表，其中 d 代表文档编号，后面的数字代表这个特征在文档 d 出现的次数，即 TF，同时，倒排列表项根据 TF 值由高到低排序。这样在计算 Jaccard 相似度的时候，可以通过动态裁剪的方式只匹配分组内的部分网页即可，不需要对分组内任意网页都计算 Jaccard 相似度，通过此种手段进一步加快了计算速度。

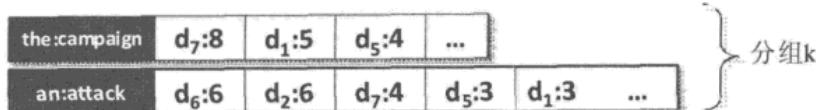


图 10-13 分组 k 建立的倒排索引片段

## 本章提要

- 网页去重时机一般在爬虫新抓取到网页后，对网页建立索引前。
- 一个典型的去重算法由特征抽取、文档指纹生成和相似性计算 3 个关键环节构成。
- 能够快速处理海量数据是搜索引擎对去重算法的内在要求，去重算法设计必须兼顾准确性和运行效率，在两者之间取得平衡。
- 4 种典型的去重算法：Shingling 算法、I-Match 算法、SimHash 算法、SpotSig 算法。看似迥异，很多基本思路相近。

## 本章参考文献

- [1] Broder, A.(1995). Some applications of Rabin's fingerprinting method. In Renato Capocelli, Alfredo De Santis, and Ugo Vaccaro, editors, Sequences II: Methods in Communications, Security, and Computer Science.
- [2] Broder, A., Glassman, S., Manasse, M., and Zweig, G(1997). Syntactic clustering of the Web. In 6th International World Wide Web Conference (Apr. 1997), 393-404.
- [3] Charikar, M.S. (2002). Similarity estimation techniques from rounding algorithms. In 34<sup>th</sup> Annual ACM Symposium on Theory of Computing .
- [4] Fetterly, D., Manasse, M., and Najork, M. (2003). On the evolution of clusters of near-duplicate Web pages. In 1st Latin American Web Congress.
- [5] Chowdhury, A., Frieder, O., Grossman, O., and McCabe, M.C. (2002). Collection statistics for fast duplicate document detection. ACM Transactions on Information Systems, 20(2):171-191.
- [6] Henzinger, M. (2006). Finding near-duplicate Web pages: A large scale evaluation of algorithms. In Proceedings of the 29<sup>th</sup> Annual International ACM SIGIR Conference on Research and Development in information retrieval. Seattle, Washington. 421-428.
- [7] Kołcz, A., Chowdhury, A., and Alspector J. (2004). Improved robustness of



signature-based near-replica detection via lexicon randomization. Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, Seattle, WA, USA. 605–610.

- [8] Theobald, M., Siddharth, J., and Paepcke, A.(2008). Spotsigs: robust and efficient near duplicate detection in large web collections. In SIGIR '08, 563–570.
- [9] Manku, G.S., Jain, A., and Das A. (2007). Detecting near-duplicates for web crawling. In WWW '07.



# 第 11 章 搜索引擎缓存机制

“一切有为法，皆如梦幻泡影，  
如露亦如电，应作如是观”

● 《金刚经》

缓存（Cache）是目前所有搜索引擎都会采用的技术。所谓缓存，就是在高速内存硬件设备内开辟一块数据存储区，用来容纳常见的用户查询及搜索结果（或者索引数据及搜索的中间结果），同时采取一定的管理策略来维护存储区内的数据。当搜索引擎接收到用户查询请求时，首先在缓存系统里查找，如果能够找到则直接返回搜索结果，否则采取正常的搜索流程来返回搜索结果。

为何搜索引擎要引入缓存机制？一则使用缓存系统能够加快用户查询响应的速度；另外还可以有效地减少搜索引擎后台计算量，节省计算资源。

对于一个正常的搜索流程，比如用户输入查询请求“搜索引擎 技术”，搜索引擎需要分别将存储在磁盘上的两个单词的倒排索引读入内存，之后进行解压缩，然后求两个单词对应倒排列表的交集，找到所有包含两个单词的文档集合，根据排序算法来对每个文档的相关性进行打分，按照相关度输出相关度最高的搜索结果。

以上这个流程涉及了磁盘读/写、内存运算等一系列操作，相对比较耗费时间和计算资源。如果将本次搜索结果存储在缓存中，下次遇到相同的查询请求，则可以直接将搜索结果返回，不需要经过上述的复杂流程进行计算。缓存一般用最快的内存设备进行存储，所以响应速度非常快，同时也省略了相当多的磁盘读取和计算步骤，有效地节省了计算资源。

以上搜索加速行为能够成立，其实隐含了一个假设，即：相同的用户查询会反复出现。只有这个假设成立，才能够利用以上措施来加快搜索速度，但是问题是这个假设成立吗？

这涉及用户查询分布本身具有的特点。我们先看下用户搜索请求行为有哪些特点。目前有很多研究集中在分析用户搜索行为，通过对搜索日志的分析，可以得出如下结论。

1. 至少 63.5% 的搜索引擎用户只看搜索结果第 1 页的内容（默认是排名头 10 位的搜索结果）；大约 11.7% 的搜索引擎用户会翻看搜索结果第 2 页内容；至少 79% 的搜



索引擎用户只查看搜索结果前3页的内容。

2. 用户发出的查询请求分布符合逆 Power-Law 规则，即少数查询占了查询总数的相当比例，而大多数查询出现次数非常少。在十亿规模的搜索日志记录中，63.7% 的用户查询只出现过一次，而热门查询占搜索请求总数的比例非常高，最热门的 25 个用户搜索请求占了用户查询请求总数的 1.2%~1.5%；同时，用户查询有很大比例的重复性，大约有 30%~40% 的用户查询是重复查询。
3. 用户查询请求具备时间局部性，即大多数重复的用户查询会在较短的间隔时间被再次重复访问。

通过上面的调查结论，可以看出在一定的时间间隔内，发送到搜索引擎的用户查询有相当比例的重复性，而缓存机制之所以能够运用在搜索引擎里来加快系统响应速度，与这一点是密不可分的。

## 11.1 搜索引擎缓存系统架构

图 11-1 是一个完善的搜索引擎缓存系统架构示意图，当搜索引擎接收到用户查询的时候，会首先在缓存系统查找，看缓存内是否包含用户查询的搜索结果，如果发现缓存已经存储了相同查询的搜索结果，则从缓存内读出结果展现给用户；如果缓存内没有找到相同的用户查询，则将用户查询按照常规处理方式交由搜索引擎返回结果，并将这条用户查询的搜索结果及中间数据根据一定策略调入缓存中，这样下次遇到同样的查询可以直接在缓存中读取，以加快用户响应速度并减少搜索引擎系统的计算负载。

缓存系统包含两个部分，即缓存存储区及缓存管理策略。缓存存储区是高速内存中的一种数据结构，可以存放某个查询对应的搜索结果，也可以存放搜索中间结果，比如一个查询单词的倒排列表。

缓存管理策略又包含两个子系统，即缓存淘汰策略和缓存更新策略。

之所以需要缓存淘汰策略，是因为不论给缓存分配多大空间，当系统运行到一定程度，很可能缓存已经满了，当有新的需要缓存的内容要进入缓存时，需要根据一定的策略，从缓存中剔除一部分优先级别较低的缓存内容，以腾出空间供后续内容放入缓存存储区，如何选择替换项目是缓存淘汰策略需要考虑的问题。

另外，使用缓存系统是有一定风险存在的，即缓存内容和索引内容不一致问题。如果搜索引擎索引的文档集合是静态文档，这个问题是不存在的，因为既然文档集合没有发生任何变化，只要搜索引擎的排序算法不更改，那么针对固定的用户查询，其对应的搜索结

果是固定不变的，所以缓存里面的内容永不过期。

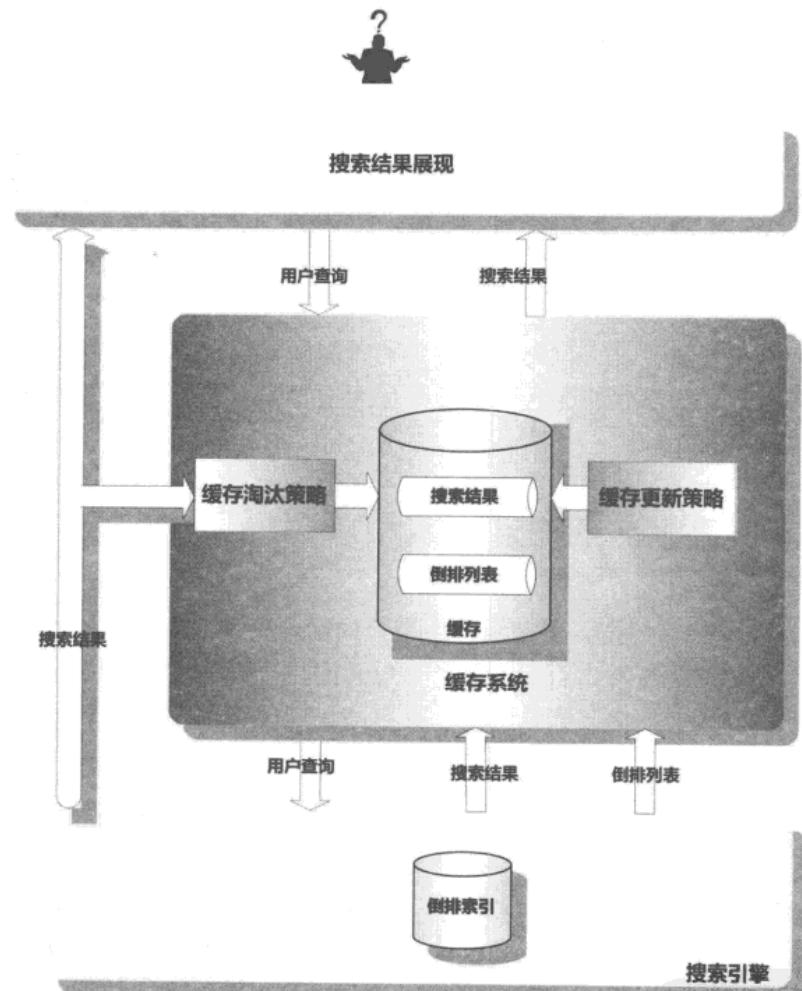


图 11-1 一个完善的搜索引擎缓存系统架构示意图

但是在一般应用场景中，搜索引擎要处理的文档集合是动态变化的，可能会面临新加入的文档，也可能会删除旧的文档或者旧的文档内容发生了变化。当索引已经反映了这种变化，而缓存数据没有随着索引做出相应的变化，那么就会发生缓存内容和索引内容不一致的问题。缓存更新策略就是用来维持两者一致性的。

对搜索引擎缓存系统来说，一个优秀的缓存系统，希望能够在以下几个方面表现出色。

### 1. 最大化缓存命中率

所谓缓存的命中率，就是说一段时间内所有用户发出的查询中，有多大比例的查询对



应的搜索结果是从缓存中获得的。这个比例越高，说明缓存管理策略越成功，就有效地节省了搜索引擎的计算成本。具体而言，不同的缓存淘汰策略就是采用不同算法来获得尽可能高的命中率。

## 2. 缓存内容与索引内容保持一致性

好的缓存管理策略应该避免出现缓存内容和索引内容不一致的状况，因为这种不一致会影响用户搜索体验，所以缓存系统需要有优秀的缓存更新策略来达到这个目的。

## 11.2 缓存对象

对于搜索引擎缓存，在存储区内存放的数据对象并不是唯一的，可以是搜索结果，也可以是某个查询词汇对应的倒排列表，或者是一些搜索的中间结果。

最常见的缓存对象类型是用户查询请求所对应的搜索结果信息，比如网页的标题、URL、包含用户查询词的动态摘要等。图 11-2 给出了将搜索结果作为缓存内容的示例，缓存里保存了“Google”，“百度 贴吧”等用户查询，以及其对应的搜索结果。如果此时有另外一个用户输入“Google”作为查询，则搜索引擎首先在缓存里面查找，发现已经存在这个用户查询项，则直接提取原先的搜索结果作为输出返回给用户。

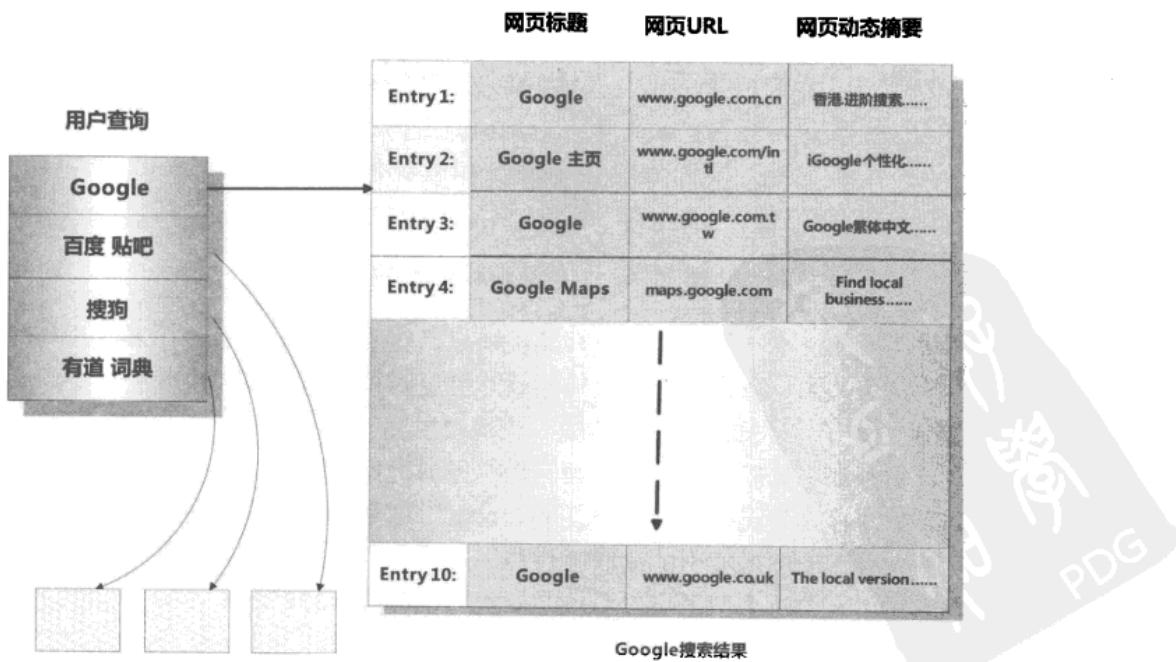


图 11-2 以搜索结果作为缓存内容

另外一种比较常见的存储对象类型是查询词汇对应的倒排列表（Posting List）。图 11-3 是以单词倒排列表作为缓存内容的一个示例图。从图中可以看出，以搜索结果作为缓存内容的情况下，用户查询即使包含多个单词，也是作为一个整体存储在缓存槽里的；而以单词倒排列表作为缓存内容的方式，其存储粒度相对会小些，是以用户查询的分词结果存储在缓存槽里的。比如“百度 贴吧”这个用户查询，在搜索结果作为缓存内容情形下占用一项缓存槽，而在缓存倒排列表方式下会占用两个缓存槽，“百度”和“贴吧”各自占用一个存储位置。

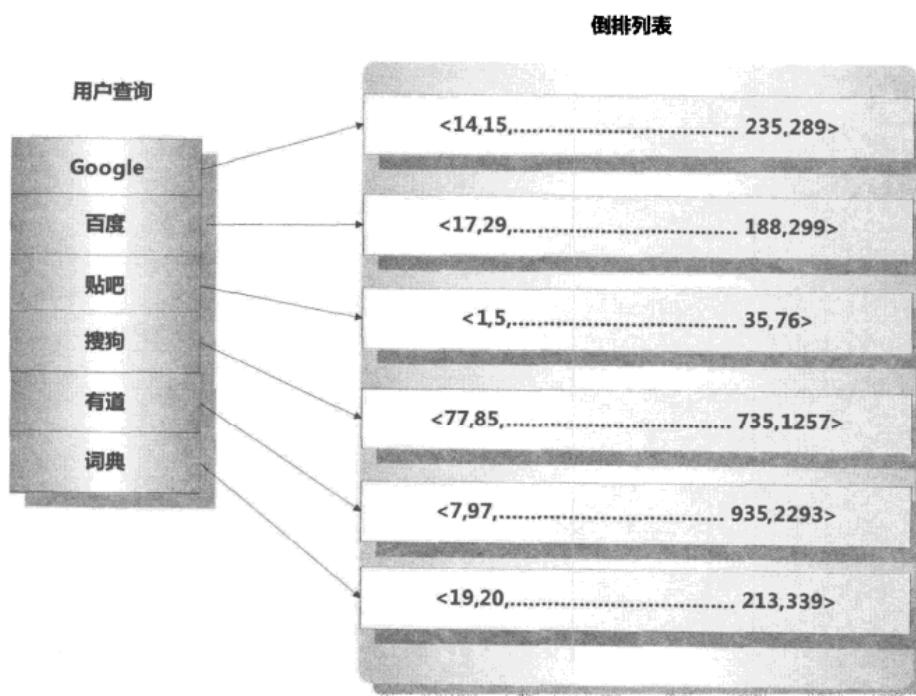


图 11-3 以单词倒排列表作为缓存内容

这两种不同的缓存存储内容各自有其优缺点，对于搜索结果型缓存来说，其用户查询响应速度非常快，因为只需要进行查找运算即可返回结果，但是其粒度比较粗，比如在如图 11-2 所示的例子中，如果此时用户输入查询“百度 百科”，则搜索引擎会发现缓存里面并不存在这个查询，只能按照正常搜索流程，去调用索引数据并进行网页排序等运算。但是倒排列表型缓存因为粒度较小，会发现“百度”这个查询词汇已经在缓存中了，此时只需要从存储在硬盘的倒排索引中读取“百科”这个词汇的倒排列表数据，然后进行排序运算即可返回结果。由这个例子可以看出，倒排列表型缓存粒度小，所以命中率高，但是因为保存的只是倒排列表这种中间数据，所以仍然需要进行后续的计算才能返回最终结果，在用户响应效率方面慢于搜索结果型缓存。而搜索结果型缓存粒度大，如果在缓存内命中



用户查询，则很快给出最终结果，但是命中率要低于倒排列表型缓存。

另外，搜索结果型缓存因为每个搜索结果的大小是可以预估的（一般取前列的  $K$  个搜索结果），所以管理起来比较简单，而倒排列表型缓存需要缓存某个单词的倒排列表，而不同单词的倒排列表大小差异很大，如果遇到一个非常大的倒排列表，可能会对目前的缓存空间造成较大影响，甚至被迫移出经常使用的用户查询缓存项，所以如何管理倒排列表型缓存存储区相对而言比较复杂。

以上两种缓存对象是比较常见的缓存类型，还有一种不太经常使用的方式，即保留两个经常搭配出现单词的倒排列表的交集，以这种中间结果形式作为缓存内容。因为用户查询有很大比例是由 2 个或者 3 个单词组成的，对于多词构成的用户查询，搜索引擎在从硬盘读出每个词汇的倒排列表后，需要进行文档队列的交集运算。而如果能够事先将这些交集运算的计算结果缓存起来，则可以避免后续的交集运算，提高搜索系统返回结果的速度。但是这种词汇组合的数据量非常大，都放置到内存中往往很困难，所以一般这种中间结果会存储在磁盘上。这种类型的缓存不能单独使用，但是可以作为多级缓存中的一个缓存级别存在，对其他类型的缓存起到补充作用。

### 11.3 缓存结构

搜索引擎缓存的结构设计可以有多种选择，最常见的是单级缓存，也可以设计为二级甚至是三级缓存结构。

单级缓存是一种最常见也最简单直接的缓存结构，缓存系统中只包含一个单一缓存，配以缓存管理策略构成了整个缓存系统。图 11-4 左方和右方分别是搜索结果型和倒排列表型单级缓存示意图。

尽管单级缓存只包含一级缓存，但是对于不同缓存对象类型来说，其内部处理流程有一定差异。搜索结果型缓存首先在缓存中查找是否包含用户查询，如果存在则直接将搜索结果返回，否则对用户查询进行处理，由搜索系统返回搜索结果并加入缓存中，之后将搜索结果返回给用户。对于倒排列表型缓存，其处理步骤正好相反，查询处理阶段首先将用户查询分词，之后在缓存中查找这些单词对应的倒排列表，如果所有单词的倒排列表都在缓存中，则由查询处理模块根据单词倒排列表对搜索结果进行排序，并将搜索结果返回给用户。如果发现某些单词的倒排列表不在缓存中，会首先从磁盘读入单词对应的倒排列表，将其放入缓存，之后进行查询处理步骤。

二级缓存结构由两级缓存串联构成，第 1 级缓存是搜索结果型缓存，第 2 级缓存是倒排列表型缓存，图 11-5 是二级缓存示意图。当系统接收到用户查询时，首先在一级缓存查

找，如果找到相同查询请求，则返回搜索结果；如果在一级缓存没有找到完全相同的查询，则转向二级缓存查找构成查询的各个单词的倒排列表，如果某些单词的倒排列表没有在二级缓存中找到，则从磁盘读取对应的倒排列表，进入二级缓存；之后，对所有单词的倒排列表进行求交集运算并根据排序算法排序输出最相关的搜索结果，将相应的用户查询和搜索结果放入一级缓存进行存储，并返回最终结果给用户。采用两级缓存结构的出发点在于能够融合搜索结果型缓存的用户快速响应速度和倒排列表型缓存的命中率高这两个优点。

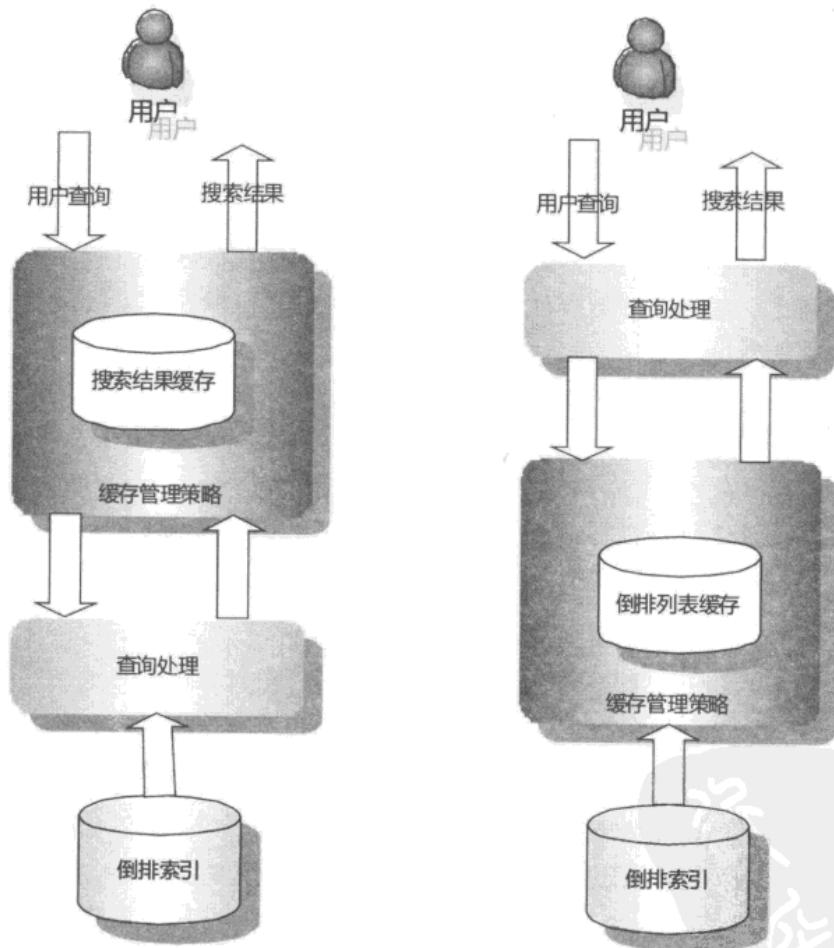


图 11-4 单级缓存

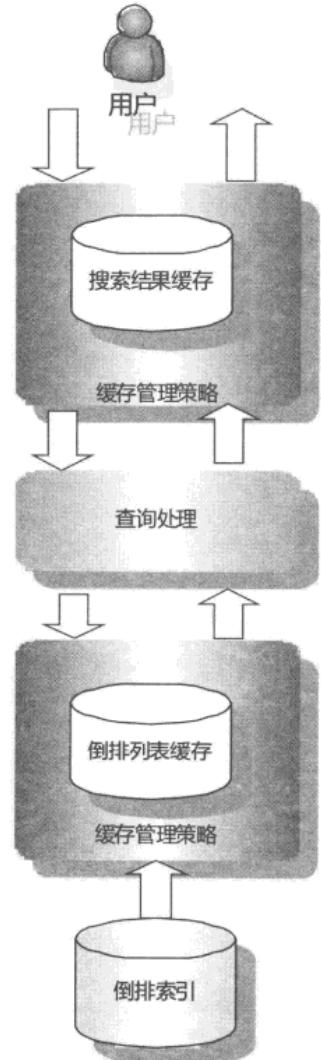


图 11-5 二级缓存示意图

#### 11.4 缓存淘汰策略 (Evict Policy)

缓存淘汰策略是任何缓存必须配备的管理策略。因为缓存的大小总是有限的，当缓存已满的时候，如果有新的缓存项需要加入，那么必须从已有的缓存项中剔除相对最不重要的项目，而不同的缓存淘汰策略就是根据不同的算法来衡量项目的重要性，并剔除掉最不重要项目占用的内存空间。缓存淘汰策略方法众多，从宏观角度，可以将其分为动态策略和静态动态混合策略。

### 11.4.1 动态策略

动态策略的缓存数据完全来自于在线用户查询请求，这种缓存策略的基本思路是：对缓存项保留一个权重值，这个权重值根据查询命中情况动态调整，当缓存已满的情况出现时，优先淘汰权重值最低的那个缓存项，通过这种方式来腾出空间。比较常见的动态策略包括：LRU 策略、LandLord 策略及 SLRU 等改进策略。

#### LRU 策略：最近最少使用策略 (Least Recently Used)

LRU 淘汰策略是计算机领域使用非常广泛的缓存替换算法，在操作系统内存管理和 Web 页面缓存等领域也发挥着重要作用。LRU 策略的基本思想是：当缓存已满时，将在设定的时间范围内使用次数最少的项目剔除出缓存，也就是将在设定时间段范围内最少访问的用户查询剔除掉。

在实际系统中，往往为每个缓存项设置一个计数器，将命中查询的计数器清零，与此同时，其他查询计数器加 1。如果缓存已满，则将计数器数值最大的项目剔除出缓存。

#### LandLord 策略

LandLord 策略是一种加权缓存策略 (Weighted Cache)。其基本计算流程如下：当一个缓存项插入缓存的时候，会根据缓存项能够获得收益和缓存项所占内存大小的比率设定一个过期值 (Deadline)，可以将这个比率理解为系统缓存这个项目的性价比。如果缓存已满，需要剔除项目的时候，选择过期值最小的项目进行淘汰，即淘汰性价比最低的项目。同时，其他未被淘汰的项目对应的过期值都减去被淘汰项目的过期值，如果一个查询请求在缓存中命中时，会相应地将其过期值根据一定策略调大。

#### SLRU 策略：大小自适应 LRU (Size-adjusted LRU)

SLRU 策略是对 LRU 方法的改进。缓存被分为两个部分：非保护区域和保护区域。每个区域的缓存项都按照最近使用频度由高到低排序，频率高端叫做 MRU，低端叫做 LRU。如果某个查询没有在缓存中找到，那么将这个查询放入非保护区域的 MRU 端；如果某个查询在缓存命中，则把这个查询记录放到保护区的 MRU 端；如果保护区已满，则把记录从保护区放入非保护区的 MRU，这样保护区的记录最少要被访问两次。淘汰机制是将非保护区的 LRU 端缓存项淘汰。

### 11.4.2 混合策略

动态策略的缓存数据完全来自于在线的用户查询请求，混合策略与此不同，其缓存数据一方面来自于在线用户查询，一方面来自于搜索日志等历史数据。目前效果较好的混合



策略包括 SDC 策略和 AC 策略。图 11-6 是这种策略的示意图。

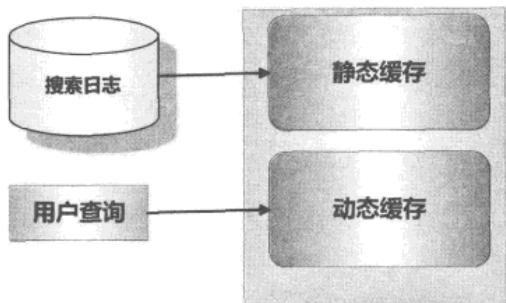


图 11-6 静态动态混合策略示意图

#### SDC 策略：静态动态混合缓存策略 (Static and Dynamic Caching)

SDC 策略是一种混合缓存策略，SDC 将缓存切割为两个部分，一个静态缓存与一个动态缓存。所谓静态缓存，即缓存内容是事先根据搜索日志统计出的最高频的那部分查询请求，在一定时间范围里是相对不变的；而动态缓存则可以配合使用 LRU 等其他缓存管理策略，根据用户查询请求不断更换内容。通过同时使用静态缓存和动态缓存，可以有效增加缓存请求命中率。SDC 是目前效果最好的缓存策略之一。

#### AC 策略：准入策略 (Admission Control)

准入策略是类似于 SDC 策略的一种方法。该方法也将缓存分为两个部分，分别存储高频出现的历史用户查询和动态出现的用户查询及其对应的搜索结果。与 SDC 不同之处在于：SDC 的静态缓存所存储的高频用户查询是完全从过去的搜索日志统计得来的静态内容，而 AC 策略则综合了搜索日志的统计数据、查询长度等多个判断因素，以此来预测某个查询是否会在未来被多次访问，如果判断是，则放入高频用户查询缓存。

## 11.5 缓存更新策略 (Refresh Policy)

如果搜索引擎的索引内容不发生变化，缓存的内容就总是和索引系统保持一致。但是搜索引擎索引经常更新，如果索引内容发生变化，而缓存内容不随着索引变动，会导致缓存内容和索引内容的不一致，这种不一致对于用户的搜索体验会造成负面影响。缓存更新策略就是通过一定的技术手段尽可能保持缓存内容和索引内容的一致性。

目前很多搜索引擎使用简单的更新策略，即在搜索引擎比较繁忙的时候不考虑缓存更新问题，而等到搜索引擎请求很少的时候，比如午夜等时间段，将缓存内的内容批量进行更新，使缓存内容保持和索引内容的一致。这种简单策略适合索引更新不是非常频繁的应

用场景，对于索引更新频繁的场景，需要相对复杂些的缓存更新策略。

根据缓存内容和索引内容联系的密切程度，目前的缓存更新策略可以分为两种：缓存—索引密切耦合策略和缓存—索引非耦合策略。

缓存—索引密切耦合策略在索引和缓存之间增加一种直接的变化通知机制，一旦索引内容发生变化则通知缓存系统，缓存系统根据一定的方法判断哪些缓存的内容发生了改变，然后将改变的缓存内容进行更新，或者设定缓存项为过期，这样就可以紧密跟踪并反映索引变化内容。这种密切耦合策略在实际实现时是非常复杂的，因为频繁的索引更新导致频繁的缓存更新，对系统效率及缓存命中率都会有直接影响。图 11-7 是一个缓存—索引密切耦合策略的示意图。当有新的索引文档进入搜索引擎时，系统会对文档内容进行分析，抽取出文档中  $TF \cdot IDF$  得分较高的索引词汇，并将这些词汇及其得分传递给失效通知模块，因为如果缓存中的查询包含这些索引词汇的话，很可能该文档将会使得缓存内容失效，失效通知模块会评估哪些缓存项需要进行内容更新，如果某项缓存项需要更新，则提取最新的缓存内容更新旧缓存项。

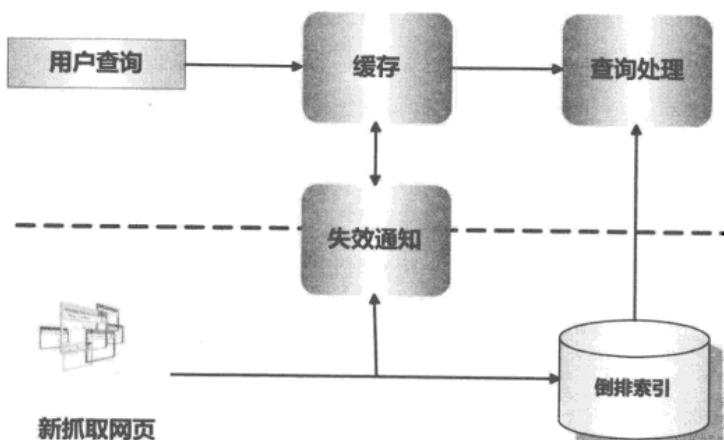


图 11-7 缓存—索引密切耦合策略示意图

缓存—索引非耦合策略则使用相对简单的策略，当索引变化时并不随时通知缓存系统进行内容更新，而是给每个缓存项设定一个过期值（Time To Live），随着时间流逝，缓存项会逐步过期。通过这种方式可以将缓存项和索引的不一致尽可能减小。雅虎的搜索引擎采用了缓存—索引非耦合策略来维护缓存内容的更新。

## 本章提要

- 使用搜索引擎缓存技术可以加快用户响应速度并节省计算资源。



- 缓存系统的目标是最大化缓存命中率和保持缓存内容与索引内容的一致性。
- 缓存存储对象主要包括网页搜索结果及查询词对应的倒排列表。
- 缓存系统可以有多层级结构。
- 缓存淘汰策略方法众多，从宏观角度，可以将其分为动态策略和静态动态混合策略。

## 本章参考文献

- [1] Xie, Y. and Ohallaron, D. (2002). Locality in search engine queries and its implications for caching. The 21st Annual Joint Conference of the IEEE Computer and Communications Societies.
- [2] Markatos, E.P. (2001) On caching search engine query results. Computer Communications 24. 137-143.
- [3] Megiddo, N. and Modha, D.S. (2004) Outperforming LRU with an adaptive replacement cache algorithm. IEEE Computer 37. 58-65.
- [4] Saraiva, P.C., de Moura, E.S., Ziviani, N., Meira, W., Fonseca, R., and Riberio-Neto, B. (2001). Rank preserving two-level caching for scalable search engines. In: Proceedings of the 24th ACM SIGIR Conference. 51-58.
- [5] Sivasubramanian, S., Pierre, G., van Steen, M., and Alonso, G.(2007) Analysis of caching and replication strategies for Web applications. IEEE Internet Computing 11. 60-66.
- [6] Long, X. and Suel, T. (2005). Three-level caching for efficient query processing in large Web search engines. InWWW'05: Proceedings of the 14th International Conference on World Wide Web. ACM Press, New York, NY, 257-266.
- [7] Baeza-Yates, R., Gionis A., Junqueira, P.F., Murdock,V., Plachouras, V., and Silvestri F. (2008). Design trade-offs for search engine caching. ACM Trans. Web, 2(4):1-28.
- [8] Baeza-Yates, R., Junqueira, F., Plachouras, V., and Hans, F. (2007) . Admission policies for caches of search engine results. In SPIRE.

# 第 12 章 搜索引擎发展趋势

“故善战者，求之于势，不责于人。故能择人而任势。任势者，其战人也，如转木石。木石之性，安则静，危则动，方则止，圆则行。故善战人之势，如转圆石于千仞之山者，势也。”

《孙子兵法·兵势第五》

前述章节讲解了搜索引擎相关的核心技术，搜索引擎的快速发展也仅是最近 15 年发生的，这与互联网的发展趋势密切相关。最近几年，互联网在经过了 Web 2.0 的市场培育阶段后，迎来了以互联网用户的个性化和社交化为中心的趋势。同时，移动设备的逐渐流行及与上述两大趋势的融合，促生了很多新型应用，为了迎接和顺应这种趋势，对于搜索引擎来说，也产生了新的挑战。

本章简略叙述搜索引擎为了应对当前互联网发展潮流，所体现出的一些新兴领域或者发展重点。这些技术领域很多并不成熟，正在快速发展之中，所以本章并不详述其技术细节，对于每个趋势，大致讲解其特点及基本技术思路。对于行业发展趋势，不同人观点迥异，本章所述这些观点属作者一家之言，仅供读者参考。

## 12.1 个性化搜索

互联网的发展已经进入了新的阶段，即以用户为中心的阶段。如何通过用户的网上行为建立用户的个人模型，并基于此，提供精准的个性化服务成为各种研究的重点。搜索引擎也不例外，个性化搜索即是为了解决这个问题而提出的技术领域。

对于搜索引擎用户来说，由于其个人兴趣不同，即使是同一个查询词，也可能其搜索意图迥异。比如用户发出查询“Ajax”，如果这个用户是计算机工程人员，那么很有可能希望查找的是技术资料；而如果用户是个球迷，那么很可能希望查找的是阿贾克斯球队的信息。所以即使是相同的查询词，如何为不同的用户提供个性化的搜索结果，成为衡量搜索



引擎搜索质量非常重要的标准。

从技术角度看，个性化搜索任务主要面临两个问题：如何建立用户的个人兴趣模型？在搜索引擎里如何使用这种个人兴趣模型？

个性化搜索的核心是根据用户的网络行为，建立一套准确的个人兴趣模型。图 12-1 是一种比较理想化的用户建模方式，即全面收集与用户相关的信息源，包括用户搜索历史及点击记录，用户浏览过的网页、用户 E-mail 信息、用户所收藏的信息及用户发布的信息比如博客、微博等内容。在此基础上建立用户兴趣模型，用户兴趣模型的表达方式也有很多种选择，比较常见的是从信息源抽取出的关键词及其权重，也可以将关键词映射到语义层面的本体结构，或者是浏览文档形成的层级分类结构。不论采取哪种方式，都通过以上手段建立了能够代表用户长期和短期兴趣的用户模型。

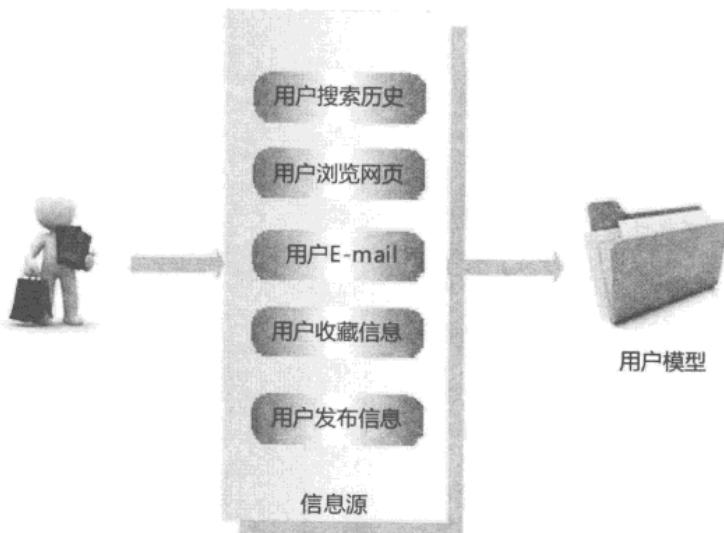


图 12-1 一种比较理想化的用户建模方式

在建好用户模型后，对于搜索引擎来说，如何使用这个用户模型也是需要考虑的问题，从目前的研究来看，一般有两种不同的使用方式：一种比较常见的方式是对初次搜索结果重排序，即利用用户当前的查询词获取排在前列的搜索结果，之后根据用户模型对这些前列搜索结果重新排序，排序原则是与用户个人兴趣越匹配的内容排名越靠前；另外一种方式是对用户当前的查询词进行直接扩展，即从用户兴趣模型里找出与当前查询词密切相关的词汇，之后将用户查询改写为扩展的查询，使用搜索引擎查找结果，这样对于不同的用户，即使是输入相同的查询词，也会获得不同的搜索结果。

为不同用户提供个性化的搜索结果，这必然是搜索引擎总的发展趋势，但是现有技术

方法也存在一些问题。首先是用户的隐私问题，为了获取精准的用户兴趣模型，就需要多方面收集用户信息，而这很可能暴露用户隐私。另外一点，用户的兴趣是不断变化的，而如果太依赖历史信息，可能无法反映用户兴趣的转移和变化。

## 12.2 社会化搜索

随着 Facebook 的逐日流行，社交网络平台和相关应用占据了互联网的主流（参见图 12-2）。社交网络平台强调用户之间的联系和交互，这对传统的搜索技术提出了新的挑战。

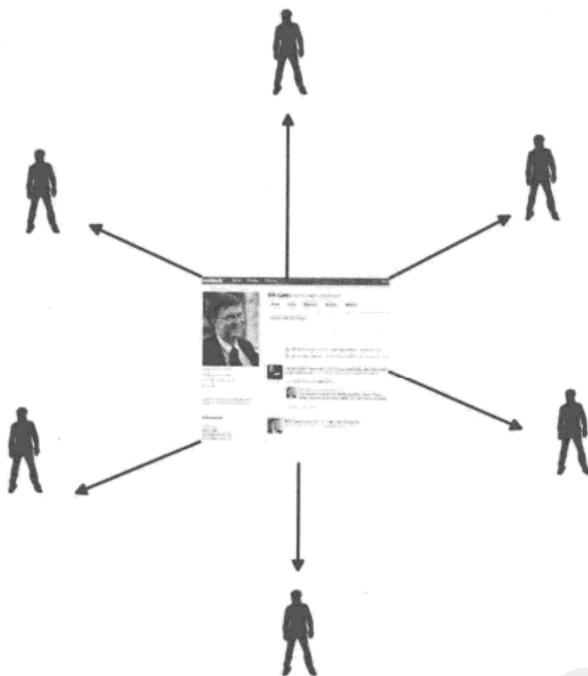


图 12-2 社交网络平台

传统搜索技术强调搜索结果和用户需求的相关性，社会化搜索除了相关性外，还额外增加了一个维度，即搜索结果的可信赖性。对于某个查询，相关的搜索结果可能成千上万，但是如果是处于用户社交网络内其他用户发布的信息、点评的评论或者验证过的信息则更可依赖，这是与用户的心理密切相关的。社会化搜索即结合了这种相关性和可信赖性，为用户提供更准确、更值得信任的搜索结果。

社会化搜索的本质是信息过滤与推荐，即对用户的疑问，社会化搜索系统推荐合适的人来回答用户疑问，或者通过社交关系过滤掉不可信赖内容，推荐可信赖内容，图 12-3 是社会化搜索整体流程示意图。根据其他用户和提问者的关系，社会化搜索系统重点关注 4 类关系：

直接有社交联系的成员，比如熟人或者朋友；有相同兴趣的成员；用户所加入网络社区的成员；领域专家。直接具有社交联系的成员对搜索者具有直接影响力，比如购物搜索时对产品品牌的评价；有相同兴趣的成员则形成了有效的信息过滤者或者推荐者；搜索者所在社区的成员从本质上讲也是有相同兴趣的成员，对于搜索者的问题能够形成有效的信息过滤和推荐；而领域专家则对问题的解答具有权威性。通过结合或者分别使用以上4类社区成员的信息，社会化搜索系统可以给搜索者提供合适的回答者或者值得信赖的信息。

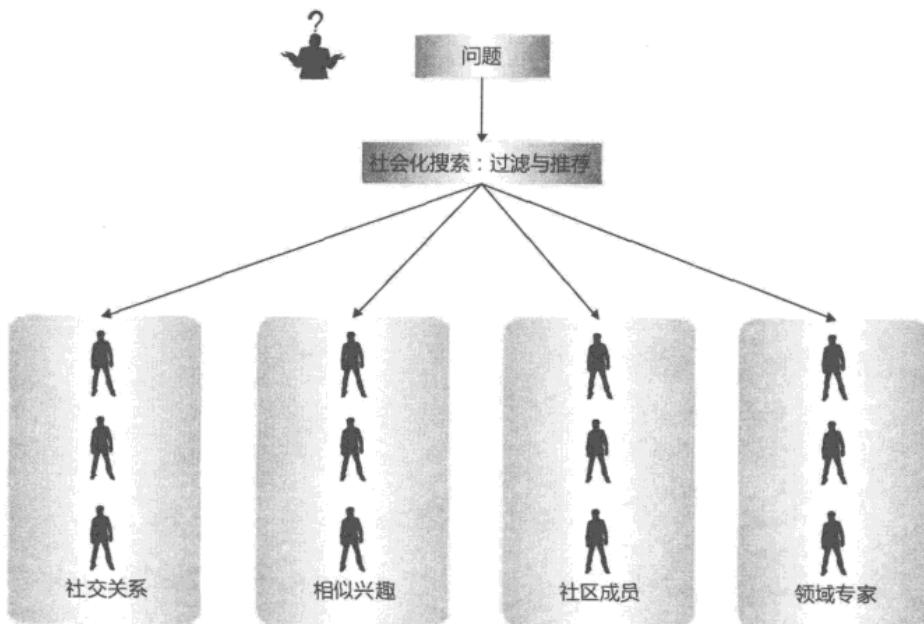


图 12-3 社会化搜索整体流程示意图

社会化搜索从其性质上讲，和个性化搜索是密不可分的，因为用户的社会关系其实也是个性化信息的一个组成部分。社会化搜索将搜索维度从信息维扩展到关系维，丰富了信息源，对于获得更准确的搜索结果帮助很大，具有广阔的发展空间。这个领域方兴未艾，无论是商业公司还是研究机构，对于这个领域还处于探索阶段。

### 12.3 实时搜索

随着Twitter、微博等提供微信息发布的个人媒体平台逐步兴起，对搜索引擎的实时性要求日益提高。微博平台和传统的信息平台比如网页、博客、新闻等相比，有其特殊性。其中一个突出的特点是时效性强，越来越多突发事件的首次发布出现在微博平台上，比如某地地震或者社会事件，这是有其必然性的。

实时搜索与传统的网页搜索有很大差异。实时搜索的核心强调“快”，即用户发布的信息能够第一时间被搜索引擎发现、索引并搜索到。传统搜索引擎在实现机制上很难达到这一点，所以实时搜索在搜索引擎的爬虫、索引系统和搜索结果排序方面都有自己独有的特点（参考图 12-4）。

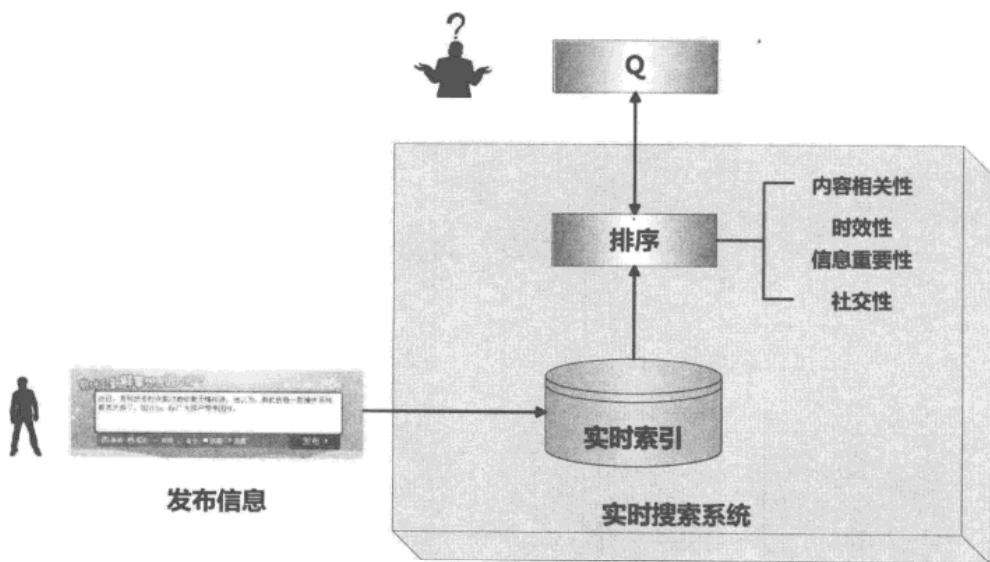


图 12-4 实时搜索系统

对于 Twitter 或者新浪微博这种信息平台来说，信息的快速获取本身不存在问题。而对于搜索引擎服务公司来说，如何能够快速全面地获取微博数据就成了严重的挑战，目前的解决方案大致有两种。一种是与微博平台合作，由信息平台将最新的信息实时推送给搜索引擎。另外一种是由爬虫实时抓取，这里面有若干技术挑战：首先，微博平台作为个人信息发布平台，往往需要用户登录使用，这对于爬虫来说就形成了第 1 道屏障；另外，即使爬虫能够登录微博系统，如何全面获取信息，难度也很大，微博用户以亿计，而且微博内容短小，同时不像网页可以通过链接传递逐步发现更多内容，如何保证信息全面性成为第 2 道屏障，尽管可以通过用户关注关系获得部分微博数据，其全面性是难以保障的；如果爬虫无法保证信息的全面性，那么实时性也是无法满足的，因为很可能最新发布的消息爬虫根本抓取不到。从这几个方面讲，微博平台对于搜索引擎来说是有其天然封闭性的。Google 目前通过与 Twitter 合作的方式获取微博数据。

在索引构建方面，实时搜索要求第一时间对新发布的内容进行索引，即用户发布则信息可搜。这对传统的索引机制提出了挑战，要求索引系统支持在建立索引的过程中，也能够提供搜索服务。



在搜索结果排序方面，实时搜索也有其特点，除了内容相关性要求外，时间因素是搜索排序中首要的考虑因素，很多实时搜索系统默认按照时间顺序排序，即最新发布的信息排在前列。

对于一个完善的实时搜索排序算法来说，一般重点考虑以下 4 方面的因子。

- **内容相关性：**信息是否与用户查询主题相关。
- **时效性：**发布时间越近的信息，其搜索排名应该越靠前。
- **信息重要性：**越重要或者越流行的信息排名应该越靠前，作为重要性或者流行性的判断指标有很多，比如信息发布来源是否可靠（微博中信息发布者的重要性）、被转发次数多少、被评论次数多少等都可以作为判断指标。
- **社交性：**一般来说，微博平台同时也是一个社交平台，微博用户的关注人和被关注人体现了这种社交性。排序时，应该将搜索结果的社交性体现出来，即用户关注人发布的信息排名应该靠前。

综上所述，实时搜索在信息获取、信息索引和排序机制方面都有其特性，以此和传统搜索引擎做出区分，而随着突发事件越来越多在这种平台发布，其重要性不言而喻。

## 12.4 移动搜索

随着智能手机的快速发展，基于手机等移动设备的搜索日益流行。移动设备对搜索应用的需求与 PC 等不同，主要源于其与 PC 相比资源的有限性，比如手机的屏幕较小，可显示区域不多；手机存储和计算资源能力较弱；手机设备打开网页的速度较慢；手机输入较为烦琐等。所以针对移动设备的搜索需要根据手机等移动设备自身的特点，开发适合其资源和设备的搜索应用。

使用手机搜索与 PC 搜索一样，由 3 个步骤组成：用户输入查询，搜索引擎提供搜索结果信息，用户点击打开感兴趣的页面。考虑到移动设备的资源有限，在这 3 个步骤都要考虑如何增加用户的体验，如图 12-5 所示。

对于用户输入查询这一环节来说，因为输入难度比桌面设备大，不够便捷，包括移动设备中的中文输入法由于资源所限、智能性不足，所以对于移动搜索来说，如何让用户输入更便捷是最主要的问题。在移动搜索环境下，用户查询词的自动补全和相关搜索提示等搜索功能更重要，通过这种辅助手段可以有效减少用户的输入次数，增加用户体验。另外，也可以通过非文本输入，比如语音识别输入的方式来减少用户输入的难度。

由于移动设备屏幕较小，如何在这么小的展示面积里让用户更快、更直接地找到答案

至关重要。所以，移动设备搜索对搜索的精度及搜索结果的展示方式要求更高。一般会考虑尽可能将搜索的答案展示在搜索结果的摘要区域，或者提供网页内容更细致准确的摘要信息，这样让用户从搜索结果展示页面就可以直接获取答案，免除用户下载页面和在页面内容里查找的过程。

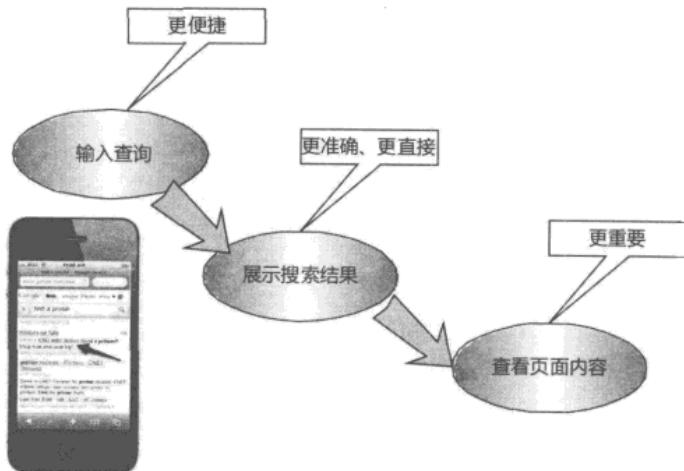


图 12-5 移动搜索 3 步骤

如果用户在搜索结果的摘要区域无法获得答案，就必须点击网页，查看页面内容，以搜索引擎目前的搜索精度来看，这步多多少少是难免的。既然如此，目标网页往往内容很多，包含导航栏、广告栏、大量图片等既消耗带宽资源，又与用户搜索内容无关的信息，所以在这一步骤，移动搜索的问题就转化成了如何提供给用户页面中最重要、最相关的部分。搜索系统可以为用户提供加工过的目标页面，通过分析目标网页的结构，自动提取重要内容，只将这部分内容展示给用户，有效增加用户体验。

智能手机的快速普及只是潮流的开头，以后一定会更加快速地发展，所以移动搜索的市场占有率会逐步上升，而其最核心的要求是如何在资源有限的情况下增加用户体验。

## 12.5 地理位置感知搜索

随着智能手机价格的下降和大众化，智能手机服务也日益流行，基于地理位置感知的搜索是其中的应用之一。目前很多智能手机不仅能够获取用户所在位置的 GPS 信息，而且可以通过陀螺仪等手机内部设备感知用户的朝向，基于这种信息，可以为用户提供准确的地理感知信息及相关搜索服务。

我们可以设想一种应用场景（参考图 12-6），一个旅游爱好者在英国旅游，当他走到“大



“本钟”前面时，地理位置感知搜索可以自动推送给用户关于“大本钟”的相关知识，比如建造历史、设计者、有关逸闻趣事等丰富的相关信息，这就是一种典型的地理位置感知搜索应用，类似的应用在不同的具体领域还有很多。



图 12-6 旅游地理位置感知搜索

图 12-7 是一个地理位置感知搜索系统的架构图，系统主要由 3 个部分构成，首先需要构建地理信息数据，用来存储某地的地理信息，比如街道名称、建筑物名称及相对位置等信息，这些数据是此类应用的基础数据。地理位置感知模块通过用户手机，可以获取用户当前所处位置及朝向等位置信息，通过和地理信息相互比较和推理，获知用户此时面对的是哪个建筑物，之后根据应用的不同，从应用相关领域知识中提取与这个建筑物相关的知识，并显示给用户。

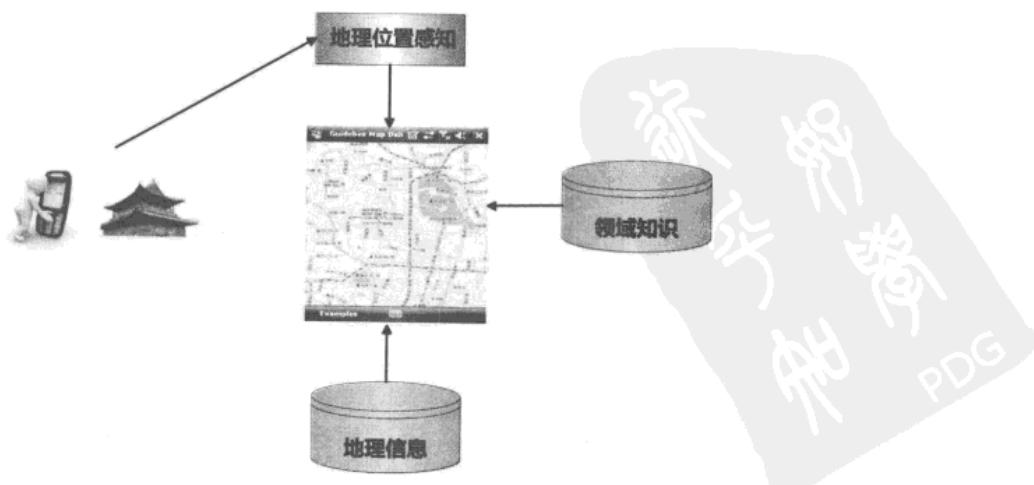


图 12-7 地理位置感知搜索系统的构架图

地理位置感知搜索可以看做是增强现实技术在搜索领域的具体应用，随着智能手机功能的丰富化，此类应用必将大行其道。

## 12.6 跨语言搜索

随着全球化进程席卷全球，互联网所提供的信息资源不再集中于英语等少数几种语言上。另一方面，英语非母语的互联网用户比重也在快速增长，对于大多数不精通外语的用户而言，熟练地使用外语查询所需要的相关的其他语种信息较为困难，而使用母语查询条件搜索出相关的外语信息，再借助于辅助翻译工具浏览信息则相对较为容易。因此自动将用户的母语查询条件翻译为相应的其他语种查询条件，再使用搜索引擎查找出所需的信息，是方便用户获取网上资源的有效途径。跨语言搜索（CLIR: Cross-Language Information Retrieval）研究的正是这方面的内容，它是信息检索研究为了克服语言障碍而发展出来的一个分支。随着互联网的蓬勃发展，研究开发优秀的跨语言信息检索系统显得日益迫切。

Google 目前已经提供多种语言之间的跨语言搜索，图 12-8 是一个具体实例，当用中文搜索“麻省理工”的时候，排在第 1 位的是麻省理工学院的英文主页。同时 Google 也在大力发展机器翻译技术，当用户搜索到外文网页后，可以使用机器翻译技术将网页翻译为用户的母语，尽管目前翻译质量不算非常理想，但是用户可以获知网页的主要内容，这样就有效地增加了搜索范围。

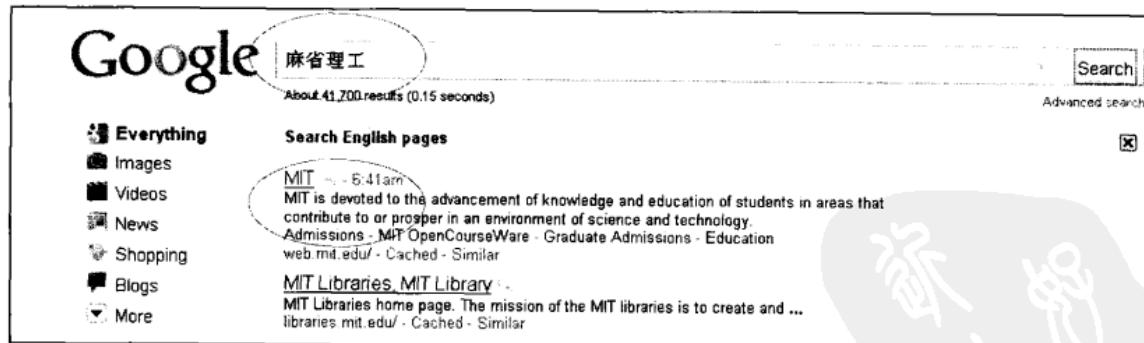


图 12-8 Google 的跨语言搜索功能

一般来说，解决用户查询与查询文档集之间的语言障碍有 3 种不同的技术路线，既可以将查询翻译成与查询文档集相同的语言，也可以将查询文档集翻译成与查询相同的语言，还可以同时将两者映射到与具体语言无关的语义空间。由于查询文档集一般都很大，采取翻译查询文档集到用户查询语言的技术路线代价太高，所以一般的研究集中在其他两种技术路线上，其中最常见的还是将用户查询翻译成文档集的语言，比如用户输入中文的查询



“麻省理工”，跨语言搜索系统将这个查询翻译成 MIT，然后去英文的网页里搜索，获得麻省理工学院的首页。

如何将中文的用户查询翻译为英文的查询？目前主流的方法有3种：机器翻译方法、双语词典查询方法及双语语料挖掘方法（参考图12-9）。

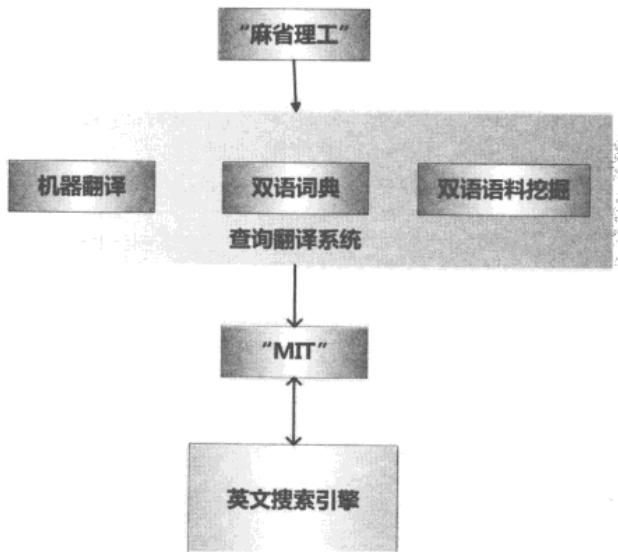


图12-9 将中文的用户查询翻译为英文的查询

机器翻译方法非常直观，即利用现成的机器翻译系统直接将中文的用户查询翻译为英文的查询。但是这个方法有个缺点，一般用户查询较短，不像文章里的语句，没有查询词的上下文信息，所以翻译效果一般不能保证。

双语词典查询方法思路也很简单，对于中文查询，直接查找中英双语词典，将可能的英文翻译找出，问题的关键是一个中文查询词往往有多个对应的翻译项，系统需要判断哪个翻译项才是正确的。另外，双语词典规模往往不够大，很多用户查询可能无法在双语词典里找到，尤其是人名、地名这些比较灵活使用的命名实体，所以这种方法需要解决以上的两个问题。

双语语料挖掘方法与上面两种方法思路不同，是通过准备好的双语语料，比如报道相同事件的中文和英文新闻，然后在这种双语语料基础上计算某个查询词翻译为另外一种语言查询词的概率，选择翻译概率最高的作为查询词翻译的结果。这种方法的缺点主要是获取大规模的双语语料有较大难度。

对于一个全球性的搜索引擎来说，具备跨语言搜索功能是必然的发展趋势，而其基本技术路线一般会采用查询翻译加上网页的机器翻译这两种技术手段。

## 12.7 多媒体搜索

目前主流搜索引擎还是基于文字的，即使是常用的图片搜索和视频搜索，用户输入查询也基本都是文本方式的，在搜索引擎内部也是通过图片标题和页面的上下文等文本进行匹配的（参考图 12-10）。这种基于文本来搜索图片、音频、视频等多媒体信息的方法有其天然缺陷，即用户查询和目标媒体的巨大差异性。多媒体搜索技术则弥补查询和搜索目标之间的差异鸿沟，使得用户可以用图片等多媒体内容作为查询输入，以获得更高的搜索准确性。



图 12-10 基于文字的图片搜索

多媒体形式除了文字外，主要包括图片、音频和视频。多媒体搜索比纯文本搜索从技术上要复杂，一般多媒体搜索包含 4 个主要步骤：多媒体特征提取、多媒体数据流分割、多媒体数据分类和多媒体数据索引搜索（参考图 12-11）。

多媒体特征抽取从原始的图片、音频或者视频中抽取出能够代表其内容的特征，文本形式一般是以关键词作为特征的，而多媒体特征与此不同，比如图片和视频中的视觉特征（色彩、纹理等），音频文件中的音调、音高等信息。这样将原始的多媒体信息转换为内部特征表示，供后续步骤使用。

对于视频和音频流来说，还需要进行数据流分割工作，这两种媒体具有时间维度信息，通过分割，将连续发生的相同或者相近的内容归为一个场景，在场景发生突然或者较大变化处将媒体分割成不同的场景。

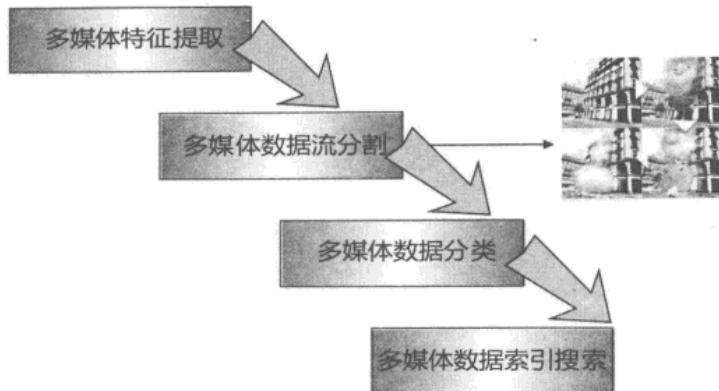


图 12-11 多媒体搜索处理步骤

对于不同的场景，可以对场景分类，比如新闻播报的场景或者爆炸场景等，通过这一步骤，等于给每个场景增加了语义标签。

在做完以上各个步骤的处理后，多媒体搜索系统就可以根据其特征对媒体片段进行索引，之后用户可以输入图片、音频或者视频片段，通过相似度计算，找出和用户查询相近的多媒体内容。

## 12.8 情境搜索

严格来说，情境搜索并非一项技术，而是一种融合了多项技术的产品形态。之前介绍的个性化搜索、社会化搜索、地点感知搜索等各项技术都是支持情境搜索这一产品形态的基础。目前包括 Google 公司等商业搜索公司都在大力提倡这个概念。

所谓情境搜索，就是能够感知人及人所处的环境，针对“此时此地此人”来建立模型，试图理解用户查询的目的，根本目标还是要理解人的信息需求。比如某个用户在苹果专卖店附近发出“苹果”这个搜索请求，基于地点感知及用户的个性化模型，搜索引擎更有可能认为这个查询是针对苹果公司的产品，而非对水果的需求。

图 12-12 给出了情境搜索示意图，搜索系统根据用户过去的查询及点击记录，或者是 IM 等信息，可以对用户的兴趣、身份等建立一份体现个性化信息的用户模型。同时，环境感知模型可以获取用户发出查询的时间、地点、周边环境等情景信息。在众多与用户“此时此地此人”的相关信息的帮助下，对用户发出的查询做出更加合理的解释，更清楚地了解用户的信息需求。总体而言，情境搜索需要的各项技术还不够成熟，这类产品还有相当大的发展空间和潜力。

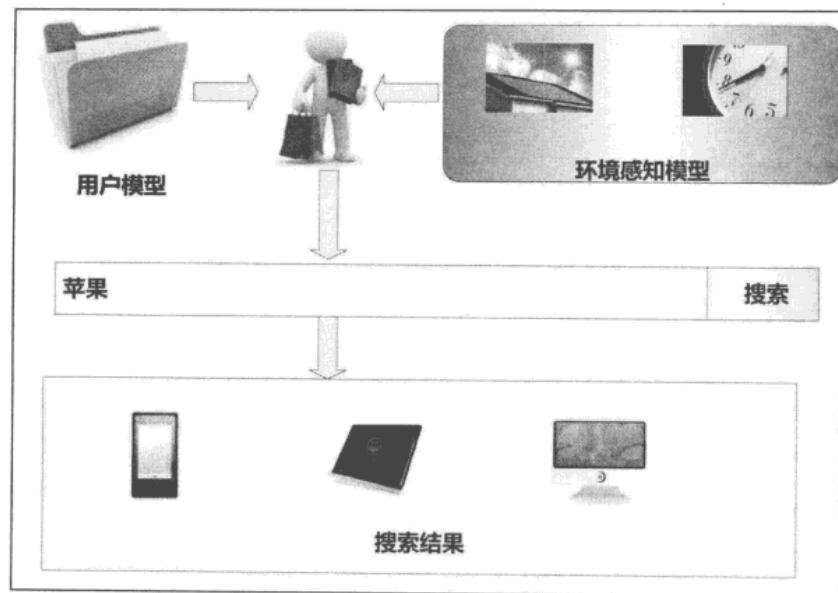


图 12-12 情境搜索示意图

# 这就是搜索引擎 核心技术详解

- 本书的最大特点是内容新颖全面而又通俗易懂。为了增进读者的理解，全书大量引入形象的图片来讲解算法原理，相信读者会发现搜索引擎的核心技术理解起来比原先想象的要简单得多。
- 本书对于实际搜索引擎所涉及的各种核心技术都有全面细致的介绍，除了作为搜索系统核心的网络爬虫、索引系统、排序系统、链接分析及用户分析外，还包括网页反作弊、缓存管理、网页去重技术等实际搜索引擎必须关注的技术，同时用相当大的篇幅讲解了云计算与云存储的核心技术原理。
- 另外，本书也密切关注搜索引擎发展的前沿技术：Google的咖啡因系统及Megastore等云计算新技术、百度的暗网抓取技术阿拉丁计划、内容农场作弊、机器学习排序等。诸多新技术在相关章节都有详细讲解，同时对于社会化搜索、实时搜索及情境搜索等搜索引擎的未来发展方向做了技术展望。
- 本书适合三类人：对搜索引擎核心算法有兴趣的技术人员、对云计算与云存储有兴趣的技术人员、从事搜索引擎优化的网络营销人员及中小网站站长。



责任编辑：付 睿  
封面设计：李 玲

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

