益 得 网

www.yidei.net

1，做中国最专业最大的免费电子书网站；
2，益得网所有书籍版权属原作者或出版社所有；
3，益得网不保证书籍的质量问题；
4，益得网希望网友支持正版,尊重作者的劳动成果；
5，益得网所有书籍来自网络，如有侵权请来信通知，益得网在 24 小时内删除。

Mail: yi_dei@163.com

Lucene in Action

中文版

ERIK HATCHER
OTIS GOSPODNETIC
zw_ren 译

版本：0.12

版	年月日	内容	作成
0.01	2006.02.01	[序]和[前言]部分	zw_ren
0.1	2006.03.28	[第一章]和[第二章部分]	zw_ren
0.11	2006.10.23	[致谢]、[关于本书]、[第二章 2.5 和 2.6 小节]	zw_ren
0.12	2006.10.24	[第二章 2.7 小节部分]	zw_ren

主要内容

第一部分 Lucene 核心

1. 接触 Lucene
2. 索引
3. 为程序添加搜索
4. 分析
5. 高极搜索技术
6. 扩展搜索

第二部分 Lucene 应用

7. 分析常用文档格式
8. 工具和扩充
9. Lucene 其它版本
10. 案例学习

前言

Lucene 开始是做为个人项目。1997 年末，工作无常，我想拿自己的东西来交易。Java 当时是新的热门编程语言，而且我需要理由来学习它。我已经懂得如何编写搜索软件，并且我想我也许可以靠编写搜索软件来维持生计，所以我写了 Lucene。

几年以后，2000 年，我认识到我并不喜欢卖东西。我对谈判协议和合同没任何兴趣，并且我不想雇用员工成立公司。我只是喜欢编写软件，而不是出售它。所以我把 Lucene 放到了 SourceForge 上，看开源是否可以使我继续做我喜欢做的事。

一些人马上开始用 Lucene。大约一年以后，在 2001 年，Apache 的人提出采用 Lucene。Lucene 邮件列表中的消息每天都稳定的增长。代码的捐助也慢慢开始了，不过大多是针对 Lucene 的边缘补充：我依然是唯一的完全理解其核心的开发人员。但是，Lucene 开始成为真正的合作项目。

现在，2004 年，Lucene 已经有一群深入理解其核心的开发人员。我也不是每天都做它的开发，这个强有力的队伍在做着实质性的添加和改进。

这些年来，Lucene 已经被翻译为其它的语言版本，包括 C++，C#，Perl 和 Python。在最初的 Java 以及其它各版本中，Lucene 的应用比我想象的要广泛得多。它在各种应用中为搜索提供动力，如财富 100 的讨论组、商业的 Bug 跟踪、微软提供的邮件搜索还有数十亿页面级的ＷＥＢ搜索引擎。在业内，我被介绍为“ Lucene 之父 ”，越来越多的人告诉我他们在项目中使用 Lucene。我始终认为我听说了所有 Lucene 应用的很小一部分。

Lucene 比我去销售它时得到更广泛的应用。应用开发人员看起来更喜欢开源。在有问题时，他们可以经常查看一下源代码，而不是去联系技术支持（然后等待回答，并期望他们正确理解了问题）。如果这还不够，邮件列表成员提供的免费支持比大多的商业支持还要好。一个机能性的开源项目如 Lucene 使得应用开发人员更有效率、更具生产性。

Lucene 通过开源，变得比我想象的强大的多。我使它运转，而 Lucene 社区使得它更加繁荣。

所以 Lucene 的下一步怎样？我无法回答。通过这本书，你也会成为 Lucene 社区的一员，而且将来就靠你把 Lucene 带到新的 Place 了，一路顺风。

DOUG CUTTING

Lucene 和 Nutch 的创始人

序

来自 Erik Hatcher

我从 Internet 初期就对搜索和索引感兴趣。我已经用 majordomo、MUSH(邮件用户的 Shell)还有一些 Perl、AWK 和 Shell 脚本来管理邮件列表。我实现了一个 CGI Web 接口，允许用户来搜索这个列表的档案并在底层使用 grep 来搜索其他用户的配置。然后出现了 Yahoo!、AltaVista 和 Excite，我经常访问这些网站。

我第一个孩子 Jakob 出生后，我的电子照片存档迅速增多。我想到开发一个管理这些图片的系统，这样我就可以为每一幅图片添加一些元信息，例如关键字和拍摄日期，当然还要能够以任意尺寸来放置图片。在 90 年代末期，我利用微软的 MIS(Microsoft Index Server)、ASP(Active Server Pages)和进行图片操作的第三方 COM 组件实现了基于文件系统的系统原型。那时，我的职业生活充满了这些技术，我可以利用业余时间对程序进行修补。

我的职业生活转向了 Java 技术，并且我越来越少使用 Windows。为了以与操作系统无关的方式用 Java 技术重新实现我的个人图片档案和搜索引擎，我找到了 Lucene。Lucene 的易用性远远超出我的预料——我用过很多其它开源的库和工具，概念比较简单但却非常难用。

2001 年，Steve Loughran 和我开始撰写《Java Development with Ant》。我们做成一个图片搜索引擎并把它扩展为文档搜索引擎。这个示例程序贯穿 Ant 全书，并能被定制为一个图片搜索引擎。和 Ant 之间的关系不只是因为其简化的编译和打包过程而且来自于一个自定义的 Ant 任务：<index>。使用 Lucene 在构建的过程中生成索引文件。这个 Ant 任务现在生存于 Lucene 的沙箱(Sandbox)中，我们将在本书第 8.4 小节中进行讲解。

这个 Ant 任务实际应用在我的博客系统中，被我称为 BlogScene(http://www.blogscene.org/erik)。在创建一个博客后，我运行 Ant 构建过程。它将索引新的博客，并把它们上传到我的服务器。我的博客服务器由一个 Servlet、一些验证模板和一个 Lucene 索引组成，可以提供大量查询，甚至是联合查询。与其它博客系统相比，BlogScene 从特征和细节上差很多，但是它的全文搜索功能十分强劲。

我现在在 Virginia 大学的 ARP(Applied Research in Patacriticism)团队工作，我把文本分析、索引和搜索的经验用于测试并在讨论量子物理和文学的联系中拓展我的思维。“诗人是世界上不被承认的工程师。”

来自 *Otis Gospodnetic*

我对信息检索和管理的兴趣与热情从 Middlebury 大学的学生时代起就开始了。那时，我发现了很好的信息资料，就是网络。虽然当时网络仍在初期，但是对聚集、分析、索引和搜索的长期需求是很明显的。我开始对创建一个信息仓库着迷，开始写 Web 爬行器，并想方设法来搜索这些收集的信息。我估计搜索大量未知领域是个终极应用。在那样的背景下，我开始了一系列有其同特征(采集和搜索信息)的工程的第一个。

1995 年，同学 Marshall Levin 和我创建了 WebPh，一个用来收集和检索个人联系信息的开源程序。基本上，它是一个 Web 接口(CGI)的简单电话薄程序，同类程序中最早的之一。大学和政府机构是这个程序的主要用户，而且很多仍然在使用它。1997 年，通过我的 WebPh，我开始创建 Populus，

一个当时很流行的站点。尽管技术(类似于 WebPh)还不成熟，Populus 担起了它的重任并可与 WhoWhere、Bigfoot 和 Infospace 等重量级角色相较。

两个关于个人联系信息的项目以后，是该探究一下新领域了。我开始了下一个冒险 Infojump，它负责从在线的简讯、杂志、报纸和期刊中提取高质量的信息。除了我自己的软件(大量的 Perl 模块和脚本)，Infojump 使用了一个叫 Webinator 的网络爬行器和一个全文检索产品 Texis。Infojump 在 1998 年提供的服务就像现在的 FindArticles.com。

尽管 WebPh、Populus 和 Infojump 完整地提供了它们各自的服务，但它们都有技术缺陷。它们每一个中缺少的部分就是一个利用反向索引支持全文搜索的信息检索库。为了避免制造相同的轮子，我开始寻找解开我疑惑的方案。2000 年初，我发现了 Lucene，我正在寻找的缺少的部分，并喜欢上了它。

我很早就加入 Lucene 项目，当时它还在 SourceForge，后来在 2002 年 Lucene 转到 Apache 软件基金全的麾下。我对 Lucene 的热爱源于它是多年来我脑中诸多想法的核心组件。这些想法之一是 Simpy，我最后的宠物项目。Simpy 是个有特色的个人网络服务，允许用户标记、索引、搜索和分享在网上找到的信息。它大量使用了 Lucene，数以千记的索引，并由 Doug Cutting 的另一个项目 Nutch 驱动。我对 Lucene 项目的积极参与赢得了与 Erik Hatcher 合著 Lucene in Action 的邀请。

Lucene in Action 是关于 Lucene 最全面的资料。它在以下的十个章节中包含的信息围绕所有你创建基于 Lucene 的复杂应用时用到的知识。这是一个平滑和轻松的合作过程的结果，就像和 Lucene 社区一样。当人们有类似的兴趣，并且不论要面对怎样的现实都想为全球知识共享做贡献时，Lucene 和 Lucene in Action 例示了我们可以做些什么。

致谢

首先并且是最重要的，我们感谢我们的妻子 Carole(Erik)和 Margaret(Otis)，一直支持这本书的写作。没有她们的支持，这本书就不可能出版。Erik 感谢他的两个儿子，Ethan 和 Jakob，因为他们的忍耐和理解，Erik 写这本书时没有时间陪他们玩耍。

我们真诚感谢 Doug Cutting。没有 Doug 的贡献，就不可能有 Lucene。没有其他 Lucene 的贡献者，Lucene 就会少很多特征、更多的 Bug，Lucene 的成长就会花更长的时间。感谢所有的贡献者，包括 Peter Carlson、Tal Dayan、Scott Ganyo、Eugene Gluzberg、Brian Goetz、Christoph Goller、Mark Harwook、Tim Jones、Daniel Naber、Andrew C. Oliver、Dmitry Serebrennikov、Kelvin Tan 和 Matt Tucher。同时，我们感谢所有贡献在第 10 章的案例的人：Dion Almaer、Michael Cafarella、Bob Carpenter、Karsten Konrad、Terence Parr、Robert Selvaraj、Ralf Steinbach、Holger Stenzhorn 和 Craig Walls。

关于本书

Lucene in Action 为使用最好的 Java 开源搜索引擎的用户提供所有细节、最好的实践、警告、技巧。本书假设读者熟悉基本的 Java 编程。Lucene 本身是个 Java 档案(JAR)文件并能集成到简单的命令行程序和大型企业级应用程序中。

Roadmap

我们在本书第 1 部分覆盖 Lucene 核心编程接口(API)使你在将 Lucene 整合到你的程序中时愿意使用它：

- 第 1 章，接触 Lucene。我们介绍了一些基本的信息搜索术语和 Lucene 的主要竞争对手。我们很快地构建了一个你马上能用或修改以适应需要的简单索引和搜索程序。这个示例程序向你打开了探索 Lucene 其它能力的大门。
- 第 2 章使你熟悉 Lucene 基本的索引操作。我们描述了索引数值和日期的不同字段类型和各种技术。包括调整索引过程、优化索引以及如何处理线程安全。
- 第 3 章向你介绍基本的搜索，包括 Lucene 如何根据查询来排列文档的细节。我们讨论基础的查询类型及它们如何通过用户输入的查询表达式创建。
- 第 4 章深入研究 Lucene 的索引核心，分析过程。分析器创建块及单词、单词流和单词过滤器。我们创建了一些定制的分析器，showcasing synonym injection and metaphone(like soundex) replacement.也分析了非英语语言，典型的分析汉字文本的示例。
- 第 5 章讲述搜索章节剩余的。我们描述了一些高级的搜索特征，包括排序、过滤及使用词向量。高级的查询类型在此出现，包括 SpanQuery 家族。最后，我们讨论了 Lucene 对查询多索引的内建支持，并行的及远程的。
- 第 6 章超越高级搜索，向你展示了如何扩展 Lucene 的搜索能力。你将学到如何定制搜索结果的排序、扩展查询表达式分析、实现 Hit 收集和调整查询性能。

第 2 部分超越 Lucene 内建的工具并向你展示围绕 Lucene 可以做什么。

- 第 7 章，我们创建了可重用、可扩展的用来分析 Word、HTML、XML、PDF 及其它格式文档的框架。
- 第 8 章包括围绕 Lucene 的扩展和工具。我们描述了一些 Lucene 的索引查看和开发工具以及 Lucene 沙箱中的好东西。高亮搜索项就是这种你想要的沙箱扩展，还有在 Ant 构建过程中创建索引的其它工具。使用 noncore 分析器，并使用类似 WordNet 的索引。
- 第 9 章描述 Lucene 翻译成其它各种语言的版本，如 C++、C#、Perl 和 Python。
- 第 10 章将 Lucene 的技术细节带到大量优秀的案例学习中。这些案例由那些创建了以 Lucene 为核心的有趣的、快速的、可升级的程序的开发者提供。

谁应该阅读本书？

在程序中需要强大搜索能力的开发人员需要阅读这本书。Lucene in Action 也适合于那些对 Lucene 或索引和搜索技术好奇的开发人员，他们可能不会马上就用到它。把 Lucene 添加到你的工具

箱对以后的项目来说是值得的—搜索是个热门的话题并且将来也会是。

这本书主要使用 Java 版的 Lucene(来自 Apache Jakarta)，并且大多数示例使用 Java。最适合熟悉 Java 的读者。Java 经验是很有帮助的，然而 Lucene 已经翻译成很多其它的语言包括 C++、C#、Python 和 Perl。概念、技术甚至 API 本身都和 Java 版 Lucene 差不多。

代码示例

本书的源代码可以从 Manning 的网站 http://www.manning.com/hatcher2 上下载。代码的使用说明包含在代码包的 README 文件。

书中出现的大多数代码是由我们编写并包含在代码包中。某些代码(尤其是案例代码)不在我们的代码包中提供。书中的代码片断归贡献者所有。同时，我们包含了Lucene代码库的部分代码，基于Apache软件许可协议(http://www.apache.org/licenses/LICENSE-2.0)。

代码示例不包括 package 和 import 语句，以节省空间；具体请参照实际代码。

为什么是 JUnit?

我们相信书中的代码示例应该都是高质量的。典型的“ hello world ”例子经常帮助读者测试他们的环境。

我们使用独特的方法来使用书中的代码示例。大部分示例是实际的JUnit测试用例(http://www.junit.org)。JUnit，是Java单元测试框架，可以断言一个特殊情况是否能以可重复的方式出现。通过IDE或Ant进行自动JUnit测试用例可以一步一步地构筑系统。我们在本书用使用JUnit是因为平时都在其它项目中使用，并想让你看看我们如何编码。测试驱动开发(Test Driven Development, TDD)是我们强烈推荐的开发模式。

如果你对 JUnit 不熟，请阅读以下基础。我们也建议你阅读 Dave Thomas 和 Andy Hunt 编著的《Pragmatic Unit Testing in Java with JUnit》，还有 Vincent Massol 和 Ted Husted 编著的《JUnit in Action》。

JUnit 基础

这部分是对 JUnit 快速但当然不完整的介绍。我们将提供理解我们示例代码所需的基础知识。首先，我们的 JUnit 测试用例继承 junit.framework.TestCase 并且很多通过部 LiaTestCase 基类间接继承它。我们的具体测试类附合这个命名习惯：给类名加后缀 Test。例如，我们的 QueryParser 的测试是 QueryParserTest.java。

JUnit 自动执行所有类似 public void testXXX()的方法，此处 XXX 是个任意有意义的名称。JUnit 测试方法必须简洁，保持好的设计。(例如创建可重复的功能模块等等)

断言

JUnit 建立在一组 assert 语句上，使你自由编写简洁的测试代码并使 JUnit 框架处理失败状态及指出细节。最常用的 assert 语句是 assertEquals；一些是为不同的数据类型而重载的 assertEquals 方法。一个示例测试方法如下：

```
public void testExample() {
    SomeObject obj = new SomeObject();
```

```
        assertEqueals(10, obj.someMethod());
    }
```

如果指定的值(在本例中的 10)不等于真实值(本例中是调用 obj 的 someMethod 的返回值)，assert 方法抛出运行时异常。除了 assertEquals，为了方便还有一些其他 assert 方法。我们也使用 assertTrue(expression)、assertFalse(expression)和 assertNull(expression)语句。这些测试分别判断这个表达式是否是 true、false 和 null。

assert 语句有个接受一个附加的 String 参数的重载表示。String 参数都是用来汇报的，在测试失败时向开发人员指出更多信息。我们使用这个 String 消息参数以更好的描述。

通过以这种风格编写我们的测试用例，可以从我们构建大系统的复杂中解放出来，而且可以每次只关注更少的细节。利用合适的测试用例，我们能够增强信心和灵活性。信心来自于我们知道代码的变化如优化算法不会破坏系统的其它部分，因为出现这种情况的话，自动测试组件能让我们在它影响产品之前发现。重构是一种改变代码内部结构的艺术(或者说科学)，所以它能够适应变化的需求而又不影响系统的对外接口。

在上下文中的 JUnit

让我们看一下到目前为止谈论的 JUnit 并把它放到本书的上下文中。JUnit 测试用例继承于 junit.framework.TestCase，且测试方法都类似 public void testXXX()形式。我们的测试用例之一(第 3 章)如下：

```
public class BasicSearchingTest extends LiaTestCase {
    public void testTerm() throws Exception {
        IndexSearcher searcher = new IndexSearcher(directory);
        Term t = new Term("subject", "ant");
        Query query = new TermQuery(t);
        Hits hits = searcher.search(query);
        assertEquals("JDwA", 1, hits.length());
                                                    One hit expected for
                                                    search for "ant"
        t = new Term("subject", "junit");
        hits = searcher.search(new TermQuery(t));
        assertEquals(2, hits.length());
                                                Two hits expected for "junit"
        searcher.close();
    }
}
```

当然，我们将在之后解释这个测试用例中使用的 Lucene API。现在我们只关注 JUnit 的细节。testTerm 方法中的 directory 变量没在此类中定义。JUnit 提供一个在执行每个测试方法之前的初始化钩子；这个钩子是名为 public void setUp()的方法。我们的 LiaTestCase 基类以这种方式实现 setUp：

```
public abstract class LiaTestCase extends TestCase {
    private String indexDir = System.getProperty("index.dir");
    protected Directory directory;
```

```
    protected void setUp() throws Exception {
        directory = FSDirectory.getDirectory(indexDir, false);
    }
}
```

如果 testTerm 中的第一个断言失败，我们会得到一个异常：

```
junit.framework.AssertionFalsedError: JDwA expected:<1> but was:<0>
    at lia.searching.BasicSearchingTest.
→   testTerm(BasicSearchingTest.java:20)
```

这个失败指出我们的测试数据与预期的结果不同。

测试 Lucene

本书中的大部分测试都是测试 Lucene 本身的。实际上，这是否现实呢？难道要测的不是我们自己写的代码而是库本身？有个 Test Driven Development 的姊妹篇是用来学习 API 的：Test Driven Learning。它为新 API 写测试以了解它是如何工作以及你能从中得到什么时非常有帮助。这正是我们在大部分代码示例中所做的，所以测试都是测试 Lucene 它本身。但是不要把这些为学习而做的测试抛开。保留它们以确保你在升级到新版的 API 或因 API 改变而重构时，它们能够保持真值。

模型对象

在一些用例中，我们使用模型对象来测试。模型对象用来作为探测器传入真实的业务逻辑，以判断这个业务逻辑是否正常工作。例如，第 4 章中有个 SynonymEngine 接口（4.6 节）。使用这个接口的真实业务逻辑是个分析器。当我们想测试这个分析器本身时，SynonymEngine 使用什么类型就不重要，我们只想使用一个定义良好并有可预见行为的对象。模型对象可以使得测试用例尽可能简单，这样它们每次只测试系统的一个方面，在测试失败要修正什么错误时没有纠缠的依赖。使用模型对象的一个好处来自于设计的变动，例如关系的分离和设计使用接口代替直接具体实现。

我们的测试数据

为了避免每个小节都要用完全不同的数据，书中大部多都是围绕一组公共的示例数据以提供一致性。这些示例数据由书籍详细资料组成。表 1 显示了这些数据，你可以参考它来理解我们的例子。

表 1 本书中用到的示例数据

Title / Author	Category	Subject
A Modern Art of Education Rudoif Steiner	/education/pedagogy	education philosophy psychology practice Waldorf
Imperial Secrets of Health and Logevity Bob Flaws	/health/alternative/Chinese	diet chinese medicine qi gong health herbs
Tao Te Ching 道德經 Stephen Mitchell	/philosophy/eastern	taoism
Gödel, Escher, Bach: an Eternal Golden Braid	/technology/computers/ai	artificial intelligence number theory mathematics music

Douglas Hofstadter		
Mindstorms Seymour Papert	/technology/computers/programming/eduction	children computers powerful ideas LOGO eduction
Java Development with Ant Erik Hatcher, Steve Loughran	/technology/computers/programming	apache jakarta ant build tool junit java development
JUnit in Action Vincent Massol, Ted Husted	/technology/computers/programming	junit unit testing mock objects
Lucene in Action Otis Gospodnetic, Erik Hatcher	/technology/computers/programming	lucene search
Extreme Programming Explained Kent Beck	/technology/computers/programming/methodology	extreme programming agile test driven development methodology
Tapestry in Action Howard Lewis-Ship	/technology/computers/programming	tapestry web user interface components
The Pragmatic Programmer Dave Thomas, Andy Hunt	/technology/computers/programming	pragmatic agile methodology developer tools

除了表中所显示的数据字段，还有 ISBN、URL 和出版日期。种类和标题字段是我们主观值，但是另一些都是有关这些书的真实客观值。

代码约定和下载

列表和正文中的代码都以等宽字体的形式出现以与普通文本区分。在正文中 Java 方法名称通常都不包含完整声明。

为了与页面宽度相适应，代码格式化为固定宽度，在需要的时候加了续行符。

我们没有包含 import 语句并很少使用类的完整名称，这样节省了可用空间。参考 Lucene 的 Java 文档获取这些信息。所有像样的 IDE 都有自动添加 import 语句的功能；Erik 使用 IDEA JintelliJ 来写程序，这样就不能知道类的完整名称，而 Otis 用 XEmacs 来完成相同的工作。把 Lucene 的 JAR 包添加到你的工程的 classpath 中，就设置好了。利用 classpath，我们假定 Lucene 的 JAR 和其它任何必须的 JAR 包都在 classpath 中可用，不必显式的写明。

我们为这本书写了大量的例子，并且免费提供给你。所有代码的压缩包放在Manning的网站上：http://www.manning.com/hatcher2。运行这些例子的详细步骤在这个压缩包的主目录里的README文件中。

第一部分

Lucene 核心

本书的前半部分覆盖了“盒子外的”(嗯…，JAR 外的)Lucene。你将以整体的观点接触 Lucene 并开发一个完整的索引和搜索程序。每个后续的章节系统地深入研究特定的内容。“索引”数据和文档和后来的对它们的“搜索”是使用 Lucene 的第一步。在从索引过程返回后，“分析”将使你深入理解在使用 Lucene 索引文本时发生了什么。搜索是 Lucene 真正的亮点：本部分以仅使用内建特征的“高级搜索”技术结束，“扩展搜索”显示了 Lucene 针对定制目的的扩展性。

第一章　接触 Lucene

本章包括

- 理解 Lucene
- 使用基本的索引 API
- 使用搜索 API
- 考虑可替换产品

Lucene 流行和成功的一个关键因素是它的简单。谨慎公开的索引和搜索 API 是良好设计的软件的特征。结果是，你在使用它之前，不必深入了解 Lucene 的信息索引和搜索工作是如何完成的。而且，Lucene 简单的 API 需要你学习的仅仅是少数的几个类。

在本章中，我们展示了如何利用 Lucene 和示例程序执行基本的索引和搜索。然后主要介绍这些处理中所需要的所有核心元素。我们也概述了其它类似的 Java/非 Java、免费的和商业的产品。

1.1 信息组织和访问的演化

为了理解这个复杂的世界，人们发明了分类、分级、种别、种类和其它类型的分层组织模式。用来在库集合中组织项目的杜威十进制分类法是个典型的分层组织模式。网络和电子数据库给我们带来大量的信息。一些公司例如 Yahoo!分类组织了在线数据。渐渐地，可利用的数据的数量非常巨大，我们需要更动态的方式来查找信息。虽然我们可以把数据分类，使用成千上网的类别和子类别的数据也不是一个有效的方法。

在海量数据中快速搜索信息的需要不仅局限在网络领域——桌面记算机可以存储越来越多的数据。切换并展开目录不是访问已有文档的有效手段。此外，我们不仅要使用计算机的计算能力：他们还能提供多媒体播放和存储。这些对计算机的使用需要有快速找到特定片断的数据，更进一步说，我们需要记富媒体——例如图像、视频和音频等各种格式的文件——能够更简单的定位。

为了理解 Lucene 在搜索中扮演什么角色，让我们从基本开始并学习什么是 Lucene 和它会为你提供什么帮助。

1.2 理解 Lucene

不同的人使用不同的方法解决相同的问题—即信息超负荷问题。一些人使用新的用户接口来工作，一些使用智能代理，还有一些使用发展较为成熟的搜索工具如 Lucene。本章稍后我们展示代码示例之前，我们将提供给你一张高层次的图来说明 Lucene 是什么，它不是什么和它以后会变得怎样。

1.2.1 Lucene 是什么

Lucene是一个高性能、可伸缩的信息搜索(IR)库。它使你可以为你的应用程序添加索引和搜索能力。Lucene是用java实现的成熟的、免费的开源项目，是著名的Apache Jakarta大家庭的一员，并且基于在Apache软件许可 [ASF, License]。同样，Lucene是当前与近几年内非常流行的免费的Java信息搜索(IR)库。

注意 贯穿这本书，我们将使用术语IR(Information Retrieval)来描述像Lucene这样的搜索工具。人们常常将IR库归诸于搜索引擎，但是一定不要将IR库与web搜索引擎混为一谈。

正如你马上就会发现的，Lucene 提供了一组简单却足够强大的核心 API，只需要最小限度地理解全文索引和搜索。你只须学习它的几个类从而把 Lucene 集成到一个应用程序中。因为 Lucene 是一个 Java 库，它并不限定要索引和搜索的内容，这使得它比其它一些搜索程序更具有优势。

刚接触 Lucene 的人经常把它误解为一个现成的程序，类似文件搜索程序或 web 网络爬行器或是一个网站的搜索引擎。那些都不是 Lucene：Lucene 是一个软件库，一个开发工具包(如果你想这样称呼)，而不是一个具有完整特征的搜索应用程序。它本身只关注文本的索引和搜索，并且这些事它完成的非常好。Lucene 使得你的应用程序只针对它的问题域来处理业务规则，而把复杂的索引和搜索实现隐藏在一组简单易用的 API 之后。你可以把 Lucene 认为成一层，应用程序位于它之上，如图 1.5 所示。

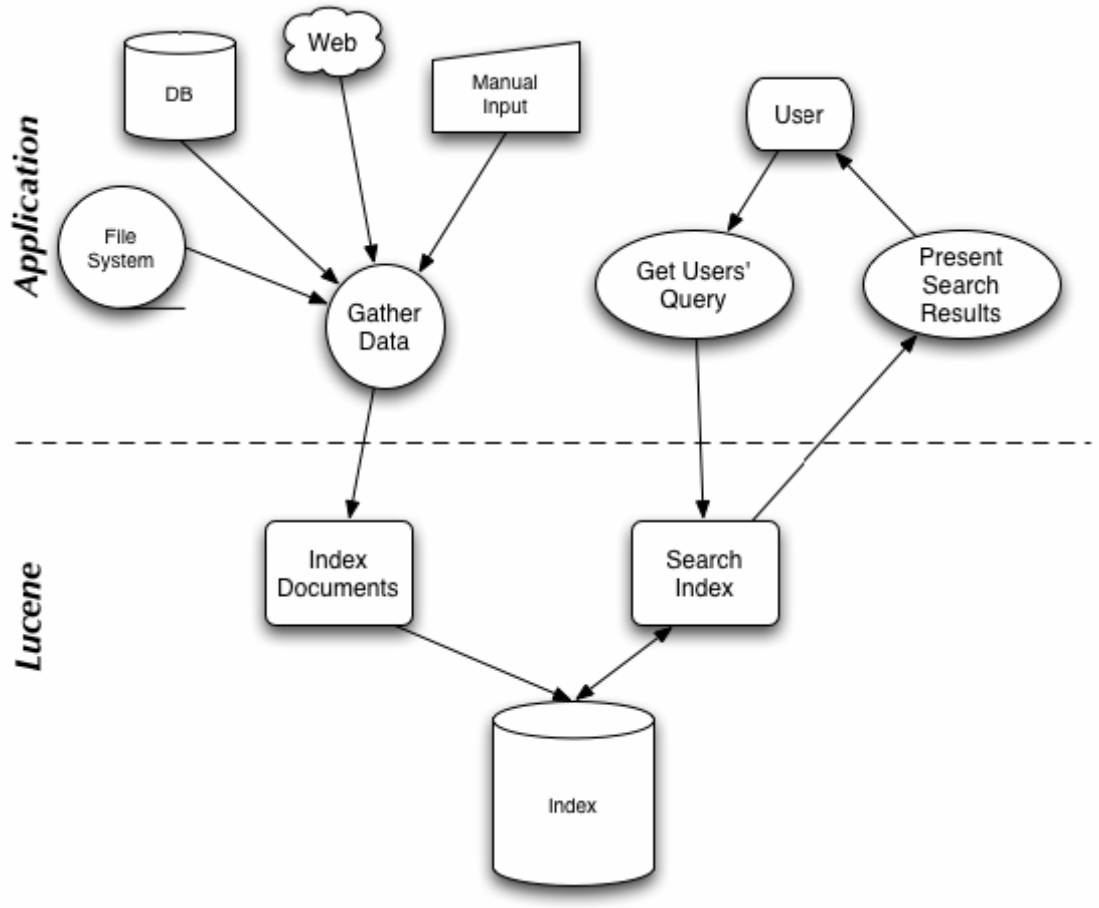

图 1.5 一个集成 Lucene 的典型应用

大量基于Lucene的完整的搜索程序已经构建出来。如果你正在寻找预创建的东西或是一个抓取、文档处理和搜索的框架，请参考Lucene Wiki的“powered by”页(http://wiki.apache.org/jakarta-lucene/PoweredBy)以获得更多选择：Zilverling、SearchBlox、Nutch、LARM和jSearch，还有其它一部分的命名。Nutch和SearchBlox的案例研究包含在第 10 章。

1.2.2 Lucene 能做什么

Lucene 使你可以为你的应用程序添加索引和搜索能力(这些功能将在 1.3 节中描述)。Lucene 可以索引并能使得可以转换成文本格式的任何数据能够被搜索。在图 1.5 可以看出，Lucene 并不关心数据的来源、格式甚至它的语言，只要你能将它转换为文本。这就意味着你可经索引并搜索存放于文件中的数据：在远程服务器上的 web 页面，存于本地文件系统的文档，简单的文本文件，微软 Word 文档，HTML 或 PDF 文件或任何其它能够提取出文本信息的格式。

同样，利用 Lucene 你可以索引存放于数据库中的数据，提供给用户很多数据库没有提供的全文搜索的能力。一旦你集成了 Lucene，你的应用程序的用户就能够像这样来搜索：+George +Rice –eat –pudding，Apple –pie +Tiger，animal:monkey AND food:banana 等等。利用 Lucene，你可以索引和搜索 email 邮件，邮件列表档案，即时聊天记录，你的 Wiki 页面……等等更多。

1.2.3 Lucene 的历史

Lucene 最初是由 Doug Cutting 开发的，在 SourceForge 的网站上提供下载。在 2001 年 9 月做为高质量的开源 Java 产品加入到 Apache 软件基金会的 Jakarta 家族中。随着每个版本的发布，这个项目得到明显的增强，也吸线了更多的用户和开发人员。2004 年 7 月，Lucene1.4 版正式发布，10 月的 1.4.2 版本做了一次 bug 修正。表 1.1 显示了 Lucene 的发布历史。

表 1.1 Lucene 的发布历史

版本	发布日期	里程碑
0.01	2000 年 3 月	第一个开源版本(SourceForge)
1.0	2000 年 10 月	
1.01b	2001 年 7 月	最后的 SourceForge 版本
1.2	2002 年 6 月	第一个 Apache Jakarta 版本
1.3	2003 年 12 月	复合索引格式，查询分析器增加，远程搜索，token 定位，可扩展的 API
1.4	2004 年 7 月	Sorting, span queries, term vectors
1.4.1	2004 年 8 月	排序性能的 bug 修正
1.4.2	2004 年 10 月	IndexSearcher optimization and misc. fixes
1.4.3	2004 年冬	Misc. fixes

注意 Lucene的创建者，Doug Cutting，在信息搜索领域有很强的理论和实践经验。他发表过许多IR主题相关的研究论文并曾在Excite、Apple和Grand Central等公司工作。最近，考虑到web搜索引擎数目的减少和这个领域的潜在垄断，他创建了Nutch，第一个开源的万维网搜索引擎(http://www.nutch.org)，它用来处理抓取、索引和搜索数十亿时常更新的网页。毫不奇怪，Lucene是Nutch的核心，10.1节包括Nutch如何使用Lucene的案例研究。

Doug Cutting 仍然是Lucene的后台主力，但是自从Lucene加入到Apache Jakarta的庇护之后，更多的聪明智慧注入进来。在本书写作时，Lucene的核心工作组有数个积极的开发者，其中两位就是本书的作者。除了官方的项目开发人员，Lucene拥有大量积极的技术用户群，经常贡献补丁，Bug修复和新的特征。

1.2.4 谁在使用Lucene

谁不使用呢？除了在Lucene Wiki的Powered by Lucene页提到的那些组织外，还有大量的知名的跨图组织正在使用Lucene。它为Eclipse IDE、Encyclopedia Britannica CD-ROM/DVD、FedEx、Mayo Clinic、Hewlett-Packard、New Scientist杂志、Epiphany、MIT的OpenCourseware和Dspace、Akamai的EdgeComputing平台等等提供搜索能力。你的名字也将会出现在这个列表中。

1.2.5 Lucene其它版本：Perl，Python，C++，.NET，Ruby

判断一个开源软件是否成功的一种方法是通过它被改编为其它语言版本的数量。使用这个标准，Lucene是非常成功的!尽管开始时Lucene是用Java写的，Lucene已经有很多其它语言的版本了：Perl，Python，C++和.NET，并且一些基础已经用Ruby实现了。这对于那些需要访问用不同的语言写成的应用程序所得到的Lucene索引的开发者来说是个好消息。在第9章你将了解更多关于这方面的东西。

1.3 索引和搜索

所有搜索引擎的核心就是索引的概念：将原始数据处理成一个高效的交差引用的查找结构以便于快速的搜索。让我们对索引和搜索过程做一次快速的高层次的浏览。

1.3.1 什么是索引，为什么它很重要？

想像一下，你需要搜索大量的文件，并且你想找出包含一个指定的词或短语的文件。你如何编写一个程序来做到这个？一个幼稚的方法是针对给定的词或短语顺序扫描每个文件。这个方法有很多缺点，最明显的就是它不适合于大量的文件或者文件非常巨大的情况。这时就出现了索引：为了快速搜索大量的文本，你必须首先索引那个文本然后把它转化为一个可以让你快速搜索的格式，除去缓慢的顺序地扫

描过程。这个转化过程称为索引，它的输出称为一条索引。

你可以把索引理解为一个可以让你快速随机访问存于其内部的词的数据结构。它隐含的概念类似于一本书最后的索引，可以让你快速找到讨论指定主题的页面。在Lucene中，一个索引是一个精心设计的数据结构，在文件系统中存储为一组索引文件。我们在附录B中详细地说明了索引文件的结构，但是目前你只须认为Lucene的索引是一个能快速的词汇查找的工具。

1.3.2 什么是搜索？

搜索是在一个索引中查找单词来找出它们所出现的文档的过程。一个搜索的质量用精确度和召回率来描述。召回率衡量搜索系统搜索到相关文档的能力，精确度衡量系统过滤不相关文档的能力。然而，在考虑搜索时你必须考虑其它一些因素。我们已经提到速度和快速搜索大量文本的能力。支持单个和多个词汇的查询，短语查询，通配符，结果分级和排序也是很重要的，在输入这些查询的时候也是友好的语法。Lucene强大的软件库提供了大量的搜索特征、bells和whistles，所以我们不得不把关于搜索的讨论展开为三章(第3、5、6章)。

1.4 Lucene实战：一个简单的程序

让我们来实战Lucene。首先回忆在1.3.1节描述的索引和搜索文件的问题。此外，假设你要索引和搜索存放于一个目录树中的文件，并不只在一个目录中。为了向你展示Lucene的索引和检索能力，我们将用到两个命令行程序：Indexer和Searcher。首先我们将索引一个包含文本文件的目录树，然后我们搜索创建的索引。

这个示例程序将使你熟悉Lucene的API，简单易用而功能强大。代码清单是完整的，现成的命令行程序。如果文件索引/搜索是你要解决的问题，那你可复制一份代码，用它来适应你的需要。在接下来的章节中，我们将更深入的描述Lucene使用中的每个方面。

在我们可以利用Lucene搜索之前，需要创建一个索引，所以我们开始Indexer程序。

1.4.1 创建一个索引

在本节中，你将看到一个名为Indexer的类和它的四个静态方法。它们共同递归遍历文件系统目录并索引所有具有.txt扩展名的文件。当Indexer执行完毕时，为它的后续Searcher(在1.4.2小节中介绍)留下一个创建好的Lucene索引。

我们不期望你熟悉例子中用到的几个Lucene类和方法，我们马上就会解释它们。在有注释的代码列表之后，我们向你展示了如何使用Indexer。如果你感觉在看到编码之前学习Indexer如何使用很有帮助，直接跳到代码后面的用法讨论部分。

使用Indexer来索引文本文件

列表1.1展示了Indexer命令行程序。它用到两个参数：

- 我们存放Lucene索引的路径
- 包含我们要索引的文本文件的路径

列表 1.1 Indexer：遍历文件系统并且索引.txt文件

```
/**
 * This code was originally written for
 * Erik's Lucene intro java.net article
 */
public class Indexer {
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            throw new Exception("Usage: java " + Indexer.class.getName()
```

```
            + " <index dir> <data dir>");
    }
    File indexDir = new File(args[0]);
    File dataDir = new File(args[1]);

    long start = new Data().getTime();
    int numIndexed = index(indexDir, dataDir);
    long end = new Date().getTime();

    System.out.println("Indexing " + numIndexed + " files took "
        + (end - start) + " milliseconds");
}

// open an index and start file directory traversal
public static int index(File indexDir, File dataDir)
    throws IOException {
    if (!dataDir.exists() || !dataDir.isDirectory()) {
        throw new IOException(dataDir
            + " does not exist or is not a directory");
    }

    IndexWriter writer = new IndexWriter(indexDir,    ① 创建Lucene索引
        new StandardAnalyzer(), true);
    writer.setUseCompoundFile(false);

    indexDirectory(writer, dataDir);

    int numIndexed = writer.docCount();
    writer.optimize();
    writer.close();
    return numIndexed;
}

// recursive method that calls itself when it finds a directory
private static void indexDirectory(IndexWriter writer, File dir)
    throws IOException {

    File[] files = dir.listFiles();
    for (int i = 0; i < files.length; i++) {
        File f = files[i];
        if (f.isDirectory()) {
            indexDirectory(writer, f);      ② 递归
        } else if (f.getName().endsWith(".txt")) {
            indexFile(writer, f);
        }
    }
```

```
    }

    // method to actually index file using Lucene
    private static void indexFile(IndexWriter writer, File f)
        throws IOException {

        if (f.isHidden() || !f.exists() || !f.canRead()) {
            return;
        }
        System.out.println("Indexing " + f.getCanonicalPath());

        Document doc = new Document();
        doc.add(Field.Text("contents", new FileReader(f)));    ③ 索引文件内容
        doc.add(Field.Keyword("filename", f.getCannicalPath())); ④ 索引文件名称
        writer.addDocument(doc);  ⑤ 添加片段到Lucene索引
    }
}
```

有趣的是，代码的大部分是执行目录的遍历(②)。只有IndexWriter的创建和关闭(①)和IndexFile方法中的四行(③，④，⑤)使用了Lucene API—有效的6行代码。

这个示例只关注.txt扩展名的文本文件是为了在说明Lucene的用法和强大功能时保持尽量简单。在第7章，我们将向你展示如何处理非文本文件，并且我们开发了一个现成的小框架来分析和索引几种常见的格式的文档。

运行Indexer

在命令行中，我们针对包含Lucene本身的源文件的本地工作目录运行Indexer。我们使Indexer索引/lucene目录下的文件并将Lucene 索引保存在build/index目录中。

```
% java lia.meetlucene.Indexer build/index /lucene

Indexing /lucene/build/test/TestDoc/test.txt
Indexing /lucene/build/test/TestDoc/test2.txt
Indexing /lucene/BUILD.txt
Indexing /lucene/CHANGES.txt
Indexing /lucene/LICENSE.txt
Indexing /lucene/README.txt
Indexing /lucene/src/jsp/README.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/ru/
→  stemsUnicode.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/ru/
→  test1251.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/ru/
→  testKOI8.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/ru/
→  testUnicode.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/rn/
→  wordsUnicode.txt
```

```
Indexing /lucene/todo.txt
Indexing 13 files took 2205 milliseconds
```

Indexer打印出索引的文件名称，你可以看出它只索引扩展名为.txt的文本文件。

注意 如果你在Windows平台的命令行中运行这个程序，你需要调整命令行的目录和路径分割符。Windows命令行是java build\index c:\lucene。

当索引完成后，Indexer输出它索引的文件数目和所花费的时间。因为报告的时间包含文件目录遍历和索引，你不能把它做为一个正式的性能衡量依据。在我们的示例中，每个索引的文件都很小，但只有了2秒索引这些文件还是不错的。

索引速度是要关注的，我们将在第2章中讨论它。但是通常，搜索更加重要。

1.4.2 在索引中搜索

在Lucene中搜索和索引一样高效和简单。它的功能惊人地强大，在第3章和第5章你将看到。现在，让我们看一下Searcher，一个我们用来搜索Indexer创建的索引的命令行程序。(记住我们的Seacher只是用来示范Lucene的搜索API的用法。你的搜索程序也可以是网页或带有GUI的桌面程序或EJB等形式。)

在上一部分，我们索引了一个目录中的文本文件。在本例中的索引，放在文件系统的一个目录中。我们让Indexer在build/index目录中创建Lucene索引，这个目录和我们调用Indexer的目录相关。在列表1.1中看出，这个索引包含被索引的文件和它们的绝对路径。现在我们要用Lucene来搜索这个索引以找出包含指定文本片段的文件。例如，我们可能想找出包含关键字java或Lucene的所有文件，或者可能想找出包含短语“system requirements”的所有文件。

使用Searcher实现搜索

Searcher程序和Indexer相辅相成并提供命令行搜索的能力。列表1.2展示了Searcher的全部代码。它接受两个命令行参数：

- Indexer创建的索引的路径
- 搜索索引的查询

列表 1.2 Searcher：为参数传来的查询搜索Lucene索引

```
/**
 * This code was originally written for
 * Erik's Lucene intro java.net article
 */
public class Searcher {
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            throw new Exception("Usage: java " + Searcher.class.getName()
                + " <index dir> <auery>");
        }
        File indexDir = new File(args[0]);
        String q = args[1];

        if (!indexDir.exists() || !indexDir.isDirectory()) {
            throw new Exception(indexDir +
                " does not exist or is not a directory.");
        }
```

```
            search(indexDir, q);
        }
        public static void search(File indexDir, String q)
            throws Exception {
            Directory fsDir = FSDirectory.getDirectory(indexDir, false);
            IndexSearcher is = new IndexSearcher(fsDir);    ① 打开索引

            Query query = QueryParser.parse(q, "contents",  ② 分析查询
                new StandardAnalyzer());
            long start = new Date().getTime();
            Hits hits = is.search(query);       ③ 搜索索引
            long end = new Date().getTime();

            System.err.println("Found " + hits.length() +
                " document(s) (in " + (end - start) +
                " milliseconds) that matched query '" +
                   q + "':");

            for (int i = 0; i < hits.length(); i++) {
                Document doc = hits.doc(i);         ④ 得到匹配的文档
                System.out.println(doc.get("filename"));
            }
        }
    }
```

Searcher 类似于 Indexer，只有几行代码与 Lucene 相关。在 search 方法中出现了几种特别的事物，

① 我们使用 Lucene 的 IndexSearcher 和 FSDirectory 类来打开我们的索引以进行搜索。

② 我们使用 QueryParser 来把 human-readable 查询分析成 Lucene 的查询类。

③ 搜索以一个 Hits 对象的形式返回结果集。

④ 注意 Hits 对象包含的仅仅是隐含的文档的引用。换句话说，不是在搜索的时候立即加载，而是采用从索引中惰性加载的方式—仅当调用 hits.doc(int)时。

运行 Searcher

让我们运行 Searcher 并用'lucene'查询在索引中找出几个文档：

```
%java lia.meetlucene.Searcher build/index 'lucene'
Found 6 document(s) (in 66 milliseconds) that matched query 'lucene':
/lucene/README.txt
/lucene/src/jsp/README.txt
/lucene/BUILD.txt
/lucene/todo.txt
/lucene/LICENSE.txt
/lucene/CHANGES.txt
```

输出显示我们用 Indexer 索引的 13 个文档中的 6 个含有 lucene 这个单词，而且这次搜索花费 66 毫秒。因为 Indexer 在索引中存放了文件的绝对路径，Searcher 可以输出它们。在这个例子中，我们决定把文件和路径存为一个字段并没有考虑什么，但是以 Lucene 的观点，可以给要索引的文档附加任意元数据。

当然，你可以使用更多复杂的查询，例如'lucene AND doug'或者'lucene AND NOT slow'或'+lucene +book'等等。第 3、5 和第 6 章所有搜索的不同方面，包括 Lucene 的查询语法。

使用 xargs 工具

Searcher 类对 Lucene 的搜索特征的非常简化的示例。所以，它仅仅把匹配结果输出到标准输出上。然而，Searcher 还有另一个技巧。考虑你需要找出含有指定的关键词或短语的文件，并且你想以某种方式处理这些匹配文件。为了保持简单性，让我们考虑你想使用 UNIX 命令 ls 列出每个匹配的文件，或许看看该文件的大小、许可位或拥有者。既然已经简单地把匹配文档的路径写到标准输出上，又把统计输出写到了标准错误上，你可以利用 UNIX 的 xargs 工具来处理匹配文件，如下：

```
%java lia.meetlucene.Searcher build/index
→ 'lucene AND NOT slow' | xargs ls -l

Found 6 document(s) (in 131 milliseconds) that
→ matched query 'lucene AND NOT slow':
-rw-r--r-- 1 erik staff 4215 10 Sep 21:51 /lucene/BUILD.txt
-rw-r--r-- 1 erik staff 17889 28 Dec 10:53 /lucene/CHANGES.txt
-rw-r--r-- 1 erik staff 2670 4 Nov 2001 /lucene/LICENSE.txt
-rw-r--r-- 1 erik staff 683 4 Nov 2001 /lucene/README.txt
-rw-r--r-- 1 erik staff 370 26 Jan 2002 /lucene/src/jsp/
→ README.txt
-rw-r--r-- 1 erik staff 943 18 Sep 21:27 /lucene/todo.txt
```

在这个例子中，我们选择布尔查询'lucene AND NOT slow'来找出所有含有单词lucene但不含有单词slow的文件。这个查询花费了131毫秒并找出6个匹配文件。我们把Searcher的输入传递给xargs命令，它将会依次使用ls –l命令来列出每个匹配的文件。与之类似，这些匹配文件可以被复制、连接、email或打印到标准输出。

我们的索引和搜索应用程序示例展示了 Lucene 的优点。它的 API 使用简洁。代码的大小(并且这适用于所有使用 Lucene 的程序)与业务目的密切相关——在这个示例中，是 Indexer 中负责寻找文本文件的文件系统爬行器和 Searcher 中打印某一查询的匹配文件名称到标准输出的代码。但是别让这个事实或者这个示例的简单让你觉得自满：在 Lucene 的表面下有相当多的事情发生，而且我们还用到了一些最好的实践经验。为了更好的使用 Lucene，更深入地理解它如何工作及如何在需要的时候扩展它是很重要的。本书作者毫无保留地给出了这些。

1.5 理解核心索引类

在 Indexer 类中可见，你需要以下类来执行这个简单的索引过程：

- IndexWriter
- Directory
- Analyzer
- Document
- Field

接下来是对这些类的一个简短的浏览，针对它们在 Lucene 的角色，给出你粗略的概念。我们将在整本书中使用这些类。

1.5.1 IndexWriter

IndexWriter 是在索引过程中的中心组件。这个类创建一个新的索引并且添加文档到一个已有的索引中。你可以把 IndexWriter 想象成让你可以对索引进行写操作的对象，但是不能让你读取或搜索。不管它的名字，IndexWriter 不是唯一的用来修改索引的类，2.2 小节描述了如何使用 Lucene API 来修改索引。

1.5.2 Directory

Directory 类代表一个 Lucene 索引的位置。它是一个抽象类，允许它的子类(其中的两个包含在 Lucene 中)在合适时存储索引。在我们的 Indexer 示例中，我们使用一个实际文件系统目录的路径传递给 IndexWriter 的构造函数来获得 Directory 的一个实例。IndexWriter 然后使用 Directory 的一个具体实现 FSDirectory，并在文件系统的一个目录中创建索引。

在你的应用程序中，你可能较喜欢将 Lucene 索引存储在磁盘上。这时可以使用 FSDirectory，一个包含文件系统真实文件列表的 Driectory 子类，如同我们在 Indexer 中一样。

另一个 Directory 的具体子类是 RAMDirectory。尽管它提供了与 FSDirectory 相同的接口，RAMDirectory 将它的所有数据加载到内存中。所以这个实现对较小索引很有用处，可以全部加载到内存中并在程序关闭时销毁。因为所有数据加载到快速存取的内存中而不是在慢速的硬盘上，RAMDirectory 适合于你需要快速访问索引的情况，不管是索引或搜索。做为实例，Lucene 的开发者在所有他们的单元测试中做了扩展使用：当测试运行时，快速的内存驻留索引被创建搜索，当测试结束时，索引自动销毁，不会在磁盘上留下任何残余。当然，在将文件缓存到内存的操作系统中使用时 RAMDirectory 和 FSDirectory 之间的性能差别较小。你将在本书的代码片断中看到 Directory 的两个实现的使用。

1.5.3 Analyzer

在文本索引之前，它先通过 Analyzer。Analyzer 在 IndexWriter 的构造函数中指定，司职对文本内容提取关键词并除去其它的内容。如果要索引的内容不是普通的文本，首先要转化成文本，如果 2.1 所示。第 7 章展示了如何从常见的富媒体文档格式中提取文本。Analyzer 是个抽象类，但是 Lucene 中有几个它的实现。有的处理的时候跳过终止词(不能用来把某个文件与其它文件区分开的常用的词)；有的处理时把关键字转化为小写字母，所以这个搜索不是大小写敏感等等。Analyzer 是 Lucene 的一个重要的部分并且不只是在输入过滤中使用。对一个将 Lucene 集成到应用程序中的开发者来说，对 Analyzer 的选择在程序设计中是重要元素。你将在第 4 章学到更多有关的知识。

1.5.4 Document

一个 Document 代表字段的集合。你可以把它想象为以后可获取的虚拟文档—一块数据，如一个网页、一个邮件消息或一个文本文件。一个文档的字段代表这个文档或与这个文档相关的元数据。文档数据的最初来源(如一条数据库记录、一个 Word 文档、一本书的某一章等等)与 Lucene 无关。元数据如作者、标题、主题、修改日期等等，分别做为文档的字段索引和存储。

注意 当我们在本书中提到一个文档，我们指一个Microsoft Word、RTF、PDF或其它文档类型；我们不是谈论Lucene的Document类。注意大小写和字体的区别。

Lucene 只用来处理文本。Lucene 的核心只能用来处理 java.lang.String 和 java.io.Reader。尽管很多文档类型都能被索引并使之可搜索，处理它们并不像处理可以简单地转化为 java 的 String 或 Reader 类型的纯文本内容那样直接。你将在第 7 章学到处理非文本文档。

在我们的 Indexer 中，我们处理文本文件，所以对我们找出的每个文本文件，创建一个 Document 类的实例，用 Field(字段)组装它，并把这个 Document 添加到索引中，完成对这个文件的索引。

1.5.5 Field

在索引中的每个 Document 含有一个或多个字段，具体化为 Field 类。每个字段相应于数据的一个片段，将在搜索时查询或从索引中重新获取。

Lucene 提供四个不同的字段类型，你可以从中做出选择：

- Keyword—不被分析，但是被索引并逐字存储到索引中。这个类型适合于原始值需要保持原样的字段，如 URL、文件系统路径、日期、个人名称、社会安全号码、电话号码等等。例如，我们在 Indexer(列表 1.1)中把文件系统路径作为 Keyword 字段。

- UnIndexed—不被分析也不被索引，但是它的值存储到索引中。这个类型适合于你需要和搜索结果一起显示的字段(如 URL 或数据库主键)，但是你从不直接搜索它的值。因为这种类型字段的原始值存储在索引中，这种类型不适合于存放比较巨大的值，如果索引大小是个问题的话。
- UnStored—和 UnIndexed 相反。这个字段类型被分析并索引但是不存储在索引中。它适合于索引大量的文本而不需要以原始形式重新获得它。例如网页的主体或任休其它类型的文本文档。
- Text—被分析并索引。这就意味着这种类型的字段可以被搜索，但是要小心字段大小。如果要索引的数据是一个 String，它也被存储；但如果数据(如我们的 Indexer 例子)是来自一个 Reader，它就不会被存储。这通常是混乱的来源，所以在使用 Field.Text 时要注意这个区别。

所有字段由名称和值组成。你要使用哪种字段类型取决于你要如何使用这个字段和它的值。严格来说，Lucene 只有一个字段类型：以各自特征来区分的字段。有些是被分析的，有些不是；有些是被索引，然面有些被逐字地存储等等。

表 1.2 提供了不同字段特征的总结，显示了字段如何创建以及基本使用示例。

表 1.2 不同字段类型的特征和使用方法

Fied method/type	Analyzed	Indexed	Stored	Example usage
Field.Keyword(String,String) Field.Keyword(String,Date)		✔	✔	Telephone and Social Security numbers, URLs, personal names, Dates
Field.UnIndexed(String, String)			✔	Document type (PDF, HTML, and so on), if not used as search criteria
Field.UnStored(String,String)	✔	✔		Document titles and content
Field.Text(String,String)	✔	✔	✔	Document titles and content
Field.Text(String,Reader)	✔	✔		Document titles and content

注意所有字段类型都能用代表字段名称和它的值的两个 String 来构建。另外，一个 Keyword 字段可以接受一个 String 和一个 Date 对象，Text 字段接受一个 String 和一个 Reader 对象。在所有情况下，这些值在被索引之前都先被转化成 Reader，这些附加方法的存在可以提供比较友好的 API。

注意 注意Field.Text(String, String)和Field.Text(String, Reader)之间的区别。String变量存储字段数据，而Reader变量不存储。为索引一个String而又不想存储它，可以用Field.UnStored(String, String)。

最后，UnStored 和 Text 字段能够用来创建词向量(高级的话题，在 5.7 节中描述)。为了让 Lucene 针对指定的 UnStored 或 Text 字段创建词向量，你可以使用 Field.UnStored(String, String, true)，Field.Text(String, String, true)或 Field.Text(String, Reader, true)。

在使用 Lucene 来索引时你会经常用到这几个类。为了实现基本的搜索功能，你还需要熟悉同样简单的几个 Lucene 搜索类。

1.6 理解核心搜索类

Lucene 提供的基本搜索接口和索引的一样直接。只需要几个类来执行基本的搜索操作：

- IndexSearcher
- Term
- Query
- TermQuery

- Hits

接下来的部分对这些类提供一个简要的介绍。我们将在深入更高级主题之前，在接下来的章节中展开这些解释。

1.6.1 IndexSearcher

IndexSearcher 用来搜索而 IndexWriter 用来索引：暴露几个搜索方法的索引的主要链接。你可以把 IndexSearcher 想象为以只读方式打开索引的一个类。它提供几个搜索方法，其中一些在抽象基类 Searcher 中实现；最简单的接受单个 Query 对象做为参数并返回一个 Hits 对象。这个方法的典型应用类似这样：

```
IndexSearcher is = new IndexSearcher(
                        FSDirectory.getDirectory("/tmp/index", false));
Query q = new TermQuery(new Term("contents", "lucene"));
Hits hits = is.search(q);
```

我们将在第 3 章中描述 IndexSearcher 的细节，在第 5、6 章有更多信息。

1.6.2 Term

Term 是搜索的基本单元。与 Field 对象类似，它由一对字符串元素组成：字段的名称和字段的值。注意 Term 对象也和索引过程有关。但是它们是由 Lucene 内部生成，所以在索引时你一般不必考虑它们。在搜索时，你可能创建 Term 对象并 TermQuery 同时使用。

```
Query q = new TermQuery(new Term("contents", "lucene"));
Hits hits = is.search(q);
```

这段代码使 Lucene 找出在 contents 字段中含有单词 lucene 的所有文档。因为 TermQuery 对象继承自它的抽象父类 Query，你可以在等式的左边用 Query 类型。

1.6.3 Query

Lucene 中包含一些 Query 的具体子类。到目前为止，在本章中我们仅提到过最基本的 Lucene Query：TermQuery。其它 Query 类型有 BooleanQuery，PhraseQuery，PrefixQuery，PhrasePrefixQuery，RangeQuery，FilteredQuery 和 SpanQuery。所有这些都在第 3 章描述。Query 是最基本的抽象父类。它包含一些通用方法，其中最有趣的是 setBoost(float)，在第 3.5.9 小节中描述。

1.6.4 TermQuery

TermQuery 是 Lucene 支持的最基本的查询类型，并且它也是最原始的查询类型之一。它用来匹配含有指定值的字段的文档，这在前几段只已经看到。

1.6.5 Hits

Hits 类是一个搜索结果(匹配给定查询的文档)文档队列指针的简单容器。基于性能考虑，Hits 的实例并不从索引中加载所有匹配查询的所有文档，而是每次一小部分。第 3 章描述了其中的细节。

1.7 其它类似的搜索产品

在你选择 Lucene 做为你的 IR 库之前，你可能想看看相同领域中的其它方案。我们对你可能相考虑的其它方案做了研究，这个小节对我们的发现做了总结。我们将这些产品分成两大类：

- 信息搜索(IR, Information Retrieval)库
- 索引和搜索程序

第一组比较小；它由一些比 Lucene 小的全文索引和搜索库组成。你可以把这个组的产品嵌入到你的程序中，如前面的图 1.5 所示。

第二组，比较大的组由一些现成的索引的搜索软件组成。这个软件一般设计为针对某种特定的数据，如网页，不如第一组的软件灵活。然而，其中一些产品也提供了它们的底层 API，所以有时你也可以把它们当做 IR 库。

1.7.1 IR 库

在我们对本章的研究中，我们发现两个 IR 库—Egothor 和 Xapian—提供了差不多的特征集合并且基本上都是辅助开发者的。我们也发现了 MG4J，它并不是一个 IR 库而是一套创建 IR 库的有用工具；我们认为使用 IR 的开发者应该了解它。这里是我们对这三种产品的评论。

Egothor

一个全文索引和搜索的 Java 库，Egothor 的核心算法与 Lucene 类似。它已经存在了很多年并拥有少量积极的开发者和用户团体。领头人是捷克工程师 Leo Galambos，一个在 IR 领域有深厚理论背景的博士研究生。他时常参与 Lucene 用户和开发者邮件列表的讨论。

Egothor 提供一个扩展的 Boolean 模块，使得它起到纯 Boolean 模块和 Vector 模块的作用。你可以通过一个查询时参数来选择使用哪个模块。这个软件有大量不同的查询类型，支持类似的查询语法，并允许多线程查询，如果你工作在多 CPU 计算机或搜索远程索引时是相当简单的。

Egothor 多以现成的程序如网络爬行器… …

1.7.2 索引和搜索程序

另一组可用的软件，包括免费的和商业的，包装成打包好的产品。这些软件通常不暴露大量的 API 且不让你基于它构建定制的软件。其中大部分提供了一种机制使你控制有限的参数集合，但却不能以预期的方法来使用这个软件。(当然，也有些特殊情况。)

这样，我们不能把这种软件直接和 Lucene 相比。然而，其中一些产品可能对你的需求来说是足够的，并能让你的工作运转起来，尽量 Lucene 或其它的 IR 库以长期角度来说是个不错的选择。这里是此类产品中比较流行的几种：

- SWISH， SWISH-E 和 SWISH++ -- http://homepage.mac.com/pauljlucas/software/swish/, http://swish-e.org/
- Glimpse 和 Webglimpse—http://webglimpse.net/
- Namazu—http://www.namazu.org/
- ht://Dig—http://www.htdig.org/
- Harvest和Harvest-NG—http://www.sourceforge.net/projects/harvest/, http://webharvest.sourceforge.net/ng/
- Microsoft Index Server—http://www.microsoft.com/NTServer/techresources/webserv/IndxServ.asp
- Verity—http://www.verity.com/

1.7.3 在线资源

上一小节只是对相关产品的总的看法。很多资源可以帮你找到的其它的 IR 库和产品：

- DMOZ—在DMOZ Open Directory Project(ODP)中，你将发现http://dmoz.org/Computers/Software/Information_Retrieval/和它的子版块有大量的信息。
- Google—尽管Google Directory是基于Open Directory的数据，这两个目录确实不一样。所以你也应该访问http://directory.google.com/Top/Computers/Software/Information_Retrival/。
- Searchtools—有个搜索工具的专门的网站http://www.searchtools.com/。这个网站并不时常更新，但是它已经很多年了并且相当广泛。软件根据操作系统、编程语言、许可证等等进行分类。如果你仅仅对用Java写的搜索软件感兴趣，访问http://www.searchtools.com/toos/tools-java.html。

我们提供了一些 Lucene 的替代品的正面评论，但是我们确信你的工作会让你觉得 Lucene 才是最好

的选择。

1.8 总结

在本章中，你获得了 Lucene 的一些基本知识。现在你知道了 Lucene 是一个 IR 库而不是现成的产品，当然也不是 Lucene 的初识者常常认为的 web 爬行器。你也了解了 Lucene 是如何产生的以及在 Lucene 背后的关键人物和组织。

根据 Manning's in Action 的思想，我们先向你展示了两个独立的程序，Indexer 和 Searcher，它们可以对存储于文件系统的文本文件进行索引和搜索。然后我们主要描述了在那两个程序中用到每个 Lucene 类。最后贡献出我们对类似于 Lucene 的一些产品的研究。

搜索无处不在，在你阅读本书时也可能发生，你对你程序中不可或缺的搜索感兴趣。基于你的需求，集成 Lucene 可能是微不足道的，或者它可能会包含在架构层的设计中。

我们像本章一样组织了后面两章的内容。首先我们需要做的是索引一些文档；在第 2 章详细讨论这个过程。

第二章　索引

本章包括

- 执行基本索引操作
- 在索引时添加 Document 和 Field
- 索引日期、数值和用来排序搜索结果的字段
- 使用影响 Lucene 索引性能和资料消耗的参数
- 优化索引

你可能想搜索存储在硬盘上的文件或者搜索你的邮件、网页甚至数据库中的数据。Lucene 能够帮助你。然页，在你能够搜索之前，你应该对它进行索引，这也是本章你所要学习的内容。

在第 1 章中，你看到了一个简单的索引示例。本章将深入并都你有关索引更新、调整索引过程的参数和更多高级索引技术以帮助你更加了解 Lucene。此处你也会发现 Lucene 索引的结构、当以多线程或多进程访问 Lucene 索引时要注意的重要问题和 Lucene 提供的防止并发修改索引的锁机制。

2.1 理解索引过程

正如你在第 1 章中所见，为索引一个文档只需调用 Lucene API 的几个方法。所以，表面看来，用 Lucene 进行索引是个简单的操作。然而在这些简单 API 的背后隐藏了一些有趣且相当复杂的操作集合。我们可以将这个集合分为三个主要的功能，如图 2.1 所示，在随后的几个小节中描述。

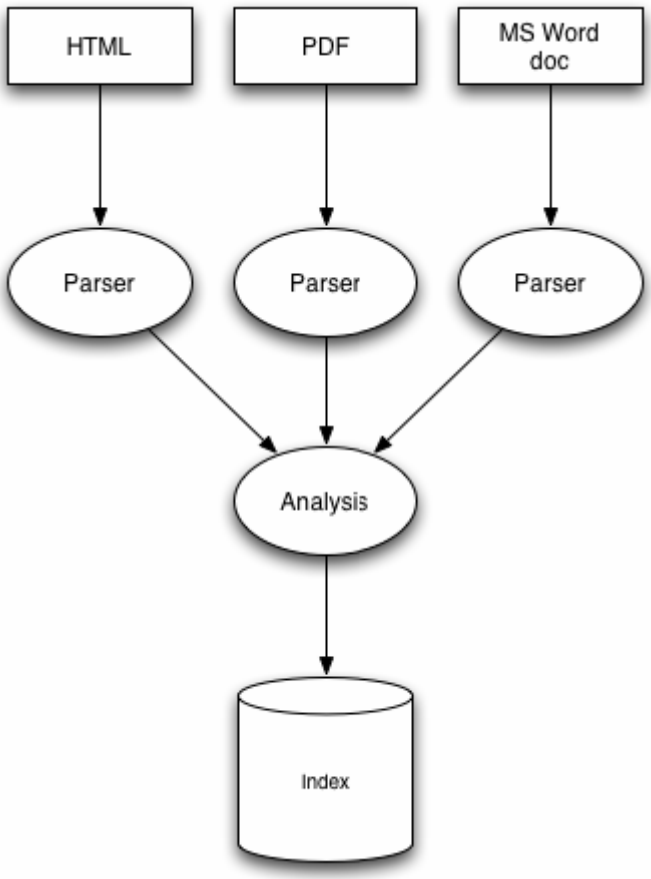

图 2.1 使用 Lucene 索引分为三个主要步骤：
将数据转化为文本，分析，将它保存至索引

2.1.1 转化为文本

为用 Lucene 索引数据，你必须首先将它转化为纯文本单词流，Lucene 能消化的格式。在第 1 章中，我们限制示例索引和搜索.txt 文件，这允许我们分析它的内容并使它来生成 Field 的实例。然而，事情并不总是那么简单。

假设你要索引一些 PDF 格式的手册。为了准备这些手册以进行索引，你必须首先提取出 PDF 文档中的文本信息并使用这些提取的数据来创建 Lucene Document 及其 Field。你回顾 21 页的表 1.2，你会发现 Field 方法总是接受 String 值，有时是 Date 和 Reader 值。没有哪个方法接受 PDF 的类型，既使这种类型存在。在索引 Microsoft Word 文档或其它任何非纯文本格式的文档时你都会遇到这个问题。甚至你在处理使用纯文本的符号的 XML 或 HTML 文档时，你仍然需要足够的智能来准备这些数据来进行索引，避免索引类似 XML 元素或 HTML 标签的东西，而要索引这些文档中的真实数据。

文本提取的细节在第 7 章，我们构建了一个小但是完整的框架以索引图 2.1 所示的所有文档格式及其它几种格式。实际上，你会发现图 2.1 和图 7.3 很类似。

2.1.2 分析

一旦你准备好了要索引的数据并创建了由 Field 组成的 Lucene Document，你就可以调用 IndexWriter 的 addDocument(Document)方法，把你的数据送入 Lucene 索引。当你做这些时，Lucene 首先分析这些数据以使它更适合索引。它将文本数据分割成块或单词，并执行一些可选择的操作。例如，单词可以在索引之前转化为小写，以保证搜索是大小写无关的。典型的，它也有可能排除转入中所有经常出现但无意义的词，例如英语终止词(a, an, the, in, on 等等)。类似的，通常分析输入单词以获取它的本质。

这个非常重要的步骤称为分析。Lucene 的输入能够以很多有趣且有用的方法进行分析，所以我们将在第 4 章详细分析这个过程。目前，把这个步骤想像为一个过滤器。

2.1.3 写索引

在输入被分析完后，就可以添加到索引中了。Lucene 将输入存储在一个反向索引的数据结构中。这个数据结构在允许快速关键字查询的同时有效地利用了磁盘空间。这个结构反向是因为它使用从输入中提取的单词做为查询键值而不是用处理的文档做为中枢入口。换句话说，代替尝试回答这个问题“ 这个文档中含有哪些单词？ ”，这个结构为提供快速回答 “ 哪篇文档含有单词 X？ ” 做了优化。

如果你想一下你常用的 Web 搜索引擎和你典型的查询格式，你会发现你想得到的精确的查询。当今所有的 Web 搜索引擎的核心都是反向索引。使得各搜索引擎不同的是一组严格保密的附加参数来改进这个索引结构。例如 Goolge 知名的级别(PageRank, PR)因素。Lucene 也有它自己的一套技术，你可以在附录 B 中学到其中一些。

2.2 基本索引操作

在第 1 章中，你看到了如何向索引中添加文档。但是我们将在此总结这个过程，同时描述删除和更新操作，以给你一个方便的参数点。

2.2.1 向索引中索加文档

为了总结你已知的，让我们来看一下在本章中作为单元测试基类的代码片断。代码列表 2.1 创建一个复合的索引称做 index-dir(索引目录)，存储于系统临时目录：UNIX 的/tmp，或使用 Windows 的 C:\TEMP。(复合索引在附录 B 中描述)我们使用 SimpleAnalyzer 来分析输入文本，然后我们索引两个简单的 Document，每个都包含四种类型的 Field：Keyword、UnIndexed、UnStored 和 Text。

列表 2.1 在基本测试类的每个测试之前准备一个新的索引

```
public abstract class BaseIndexingTestCase extends TestCase {
    protected String[] keywords = {“1”, “2”};
    protected String[] unindexed = {“Netherlands”, “Italy”};
    protected String[] unstored = {“Amsterdam has lots of bridges”, “Venice has lots of canals”};
    protected String[] text = {“Amsterdam”, “Venice”};
    protected Directory dir;

    protected void setUp() throws IOException {
        String indexDir =
            System.getProperty(“java.io.tmpdir”, “tmp”)   +
            System.getProperty(“file.separator”) + “index-dir”;
        dir = FSDirectory.getDirectory(indexDir, true);
        addDocuments(dir);
    }

    protected void addDocuments(Directory dir)
        throws IOException {
        IndexWriter writer = new IndexWriter(dir, getAnalyzer(), true);
        writer.setUseCompoundFile(isCompound());
```

```
            for (int i = 0; i < keywords.length; i++) {
                Document doc = new Document();
                doc.add(Field.Keyword("id", keywords[i]));
                doc.add(Field.UnIndexed("country", unindexed[i]));
                doc.add(Field.UnStored("contents", unstored[i]));
                doc.add(Field.Text("city", text[i]));
                writer.addDocument(doc);
            }
            writer.optimize();
            writer.close();
        }

        protected Analyzer getAnalyzer() {
            return new SimplyAnalyzer();
        }
        protected boolean isCompound() {
            return true;
        }
    }
```

因为 BaseIndexingTestCase 类要被本章的其它单元测试类继承，我们将指出几个重要的细节。BaseIndexingTestCase 每次 setUp()方法调用时创建相同的索引。因为 setUp()在测试执行之前被调用，每个测试都是针对新创建的索引运行。尽管基类使用 SimpleAnalyzer，子类可以覆盖 getAnalyzer()方法以返回不同的 Analyzer 类型。

不同的 Document

Lucene 的一个重要特征是它允许有不同 Field 的 Document 在同一索引中共存。这就意味着你可以用一个索引来保存代表不同实体的 Document。例如，你可以存放代表零售产品，有名称和价格字段的 Document 和代表 people，有名称、年龄和性别字段的 Document。

附加字段

假设你有个生成给定单词的同意词数组的程序，并且你想用 Lucene 来索引基本词和所有它的同意词。实现的一个方法是遍历所有的同意词并把它们添加到一个 String 中，然后你可以用它来创建 Lucene 字段。索引所有同意词和基本词另一个方法可能是更好的方法是把不同的值添加到相同的字段，如下：

```
String baseWord = "fast";
String synonyms[] = String {"quick", "rapid", "speedy"};
Document doc = new Document();
doc.add(Field.Text("word", baseWord));
for (int i = 0; i < synonyms.length; i++) {
    doc.add(Field.Text("word", synonyms[i]));
}
```

其中，Lucene 添加所有的单词并把它们索引在同一个字段 word 中，允许你在搜索时使用其中任何一个。

2.2.2 在索引中清除 Document

尽管大多程序关心的是添加 Document 到 Lucene 索引中，一些也需要清除它们。例如，报纸出版社可能只想在可搜索的索引中保留最近一个周的有价值的新闻。另外的程序可能想清除所有包含特定单词的 Document。

Document 的删除是由 IndexReader 来完成的。这个类并不立即从索引中删除 Document。它只做个删除的标志，等待 IndexReader 的 close()方法调用时真正的 Document 删除。理解了这些之后，让我们

看一下列表 2.2：它继承 BaseIndexingTestCase 类，这意味着在每次测试方法运行之前，基类重建两个文档的索引，在 2.2.1 小节中描述。

列表 2.2 根据内部文档号删除 Document

```
public class DocumentDeleteTest extends BaseIndexingTestCase {
    public void testDeleteBeforeIndexMerge() throws IOException {
        IndexReader reader = IndexReader.open(dir);
        assertEquals(2, reader.maxDoc());       ① 下一个 Document 号是 2
        assertEquals(2, reader.numDocs());      ② 索引中有两个 Document
        reader.delete(1);     ③ 删除号码为 1 的 Document

        assertTrue(reader.isDeleted(1));        ④ 删除 Document
        assertTrue(reader.hasDeletions());      ⑤ 包含删除的索引
        assertEquals(2, reader.maxDoc());       ⑥ 1 个索引的 Document，下一个 Document 号是 2
        reader.close();
        reader = IndexReader.open(dir);

        assertEquals(2, reader.maxDoc());       ⑦ 在 IndexReader 重新打开后，
        assertEquals(1, reader.numDocs());         下一个 Document 号是 2
        reader.close();
    }

    public void testDeleteAfterIndexMerge() throws IOException {
        IndexReader reader = IndexReader.open(dir);
        assertEquals(2, reader.maxDoc());
        assertEquals(2, reader.numDocs());
        reader.delete(1);
        reader.close();

        IndexWriter writer = new IndexWriter(dir, getAnalyzer(), false);
        writer.optimize();
        writer.close();

        reader = IndexReader.open(dir);
        assertFalse(reader.isDeleted(1));
        assertFalse(reader.hasDeletions());     ⑧ Optimizing
        assertEquals(1, reader.maxDoc());          renumbers
        assertEquals(1, reader.numDocs());         Documents

        reader.close();
    }
}
```

①②③ 列表 2.2 的代示展示了如何指定 Document 的内部编号来删除 Document。它也展示了 IndexReader 经常混淆的两个方法的不同：maxDoc()和 numDocs()。前者返回下一个可用的内部 Document 号，后者返回索引中的 Document 的数目。因为我们的索引只含有两个 Document，numDocs()返回 2；又因为 Document 号从 0 开始，maxDoc()也返回 2。

注意 每个Lucene的Document有个唯一的内部编号。这些编码不是永久分配的，因为Lucene索引分配时在内部重新分配Document的编号。因此，你不能假定一个给定的Document总是拥有同一个Document编号。

④⑤ 在testDeleteBeforeIndexMerge()方法中的测试也示范了IndexReader的hasDeletions()方法以检查一个索引是否包含有删除标志的Document和isDeleted(int)方法以检查指定编号的Document的状态。

⑥⑦ 可见，numDocs()能够立即感知到Document的删除，而maxDoc()不能。

⑧ 此外，在testDeleteAfterIndexMerge()方法中，我们关闭IndexReader并强制Lucene优化索引以合并索引的各片断。然后我们用IndexReader打开索引，maxDoc()方法返回1而不是2，因为在删除和合并后，Lucene对剩余的Document重新编号。索引中只有一个Document，所以下一下可能Document编号是1。

除了我们通过指定Document编号来删除单个Document之外，你可以用IndexReader的delete(Term)方法删除多个Document。使用这个删除方法，允许你删除所有包含指定Term的Document。例如，为了删除city字段中包含单词Amsterdam的Document，你可以这样用IndexReader：

```
IndexReader reader = IndexReader.open(dir);
reader.delete(new Term("city", "Amsterdam"));
reader.close();
```

你在使用这个方法时要特别小心，因为在所有索引的 Document 中指定一个 term 将会擦除整个索引。这个方法的使用类似于基于 Document 编号的删除方法；将在 2.2.4 小节中描述。

你可能奇怪为什么 Lucene 在 IndexReader 中执行 Document 删除而不是 IndexWriter 中。这个问题在 Lucene 社区中每几个月就问一次，大概因为有缺点或者可能是容易让人误解的类名。Lucene 的用户经常认为 IndexWriter 是唯一可以修改索引的类，且 IndexReader 以只读的形式访问索引。实际上，IndexWriter 只接触索引片断列表和合并片断时的一小部分索引文件。另一方面，IndexReader 知道如何解析所有索引文件。当一个 Document 删除时，IndexReader 在标记它被删除之前首先需要定位包含指定 Document 的片断。目前还没有计划改变这两个 Lucene 类的名称或行为。

2.2.3 恢复 Document

因为 Document 的删除延迟到 IndexReader 实例关闭时才执行，Lucene 允许程序改变想法并恢复已做删除标记的 Document。对 IndexReader 的 undeleteAll()方法的调用通过清除索引目录中的.del 文件来恢复所有删除的 Document。所以在关闭 IndexReader 实例关闭之后 Document 就保留在索引中了。只能使用与删除 Document 时同一个 IndexReader 实例，才能调用 undeleteAll()来恢复 Document。

2.2.4 更新索引中的 Document

“如何才能更新索引中的文档？”是一个在 Lucene 用户邮件列表中经常问的问题。Lucene 并没有提供更新方法；Document 必须首先从索引中删除然后再重新添加它，如列表 2.3 所示。

列表 2.3 通过删除再添加的方法更新索引的 Document

```
public class DocumentUpdateTest extends BaseIndexingTestCase {
    public void testUpdate() throws IOException {
        assertEquals(1, getHitCount("city", "Amsterdam"));
        IndexReader reader = IndexReader.open(dir);
        reader.delete(new Term("city", "Amsterdam"));
        reader.close();

        assertEquals(0, getHitCount("city", "Amsterdam"));
        IndexWriter writer = new IndexWriter(dir, getAnalyzer(), false);
```

```
        Document doc = new Document();
        doc.add(Field.Keyword("id", "1"));
        doc.add(Field.UnIndexed("country", "Netherlands"));
        doc.add(Field.UnStored("contents",
            "Amsterdam has lots of bridges"));
        doc.add(Field.Text("city", "Haag"));
        writer.addDocument(doc);
        writer.optimize();
        writer.close();
        assertEquals(1, getHitCount("city", "Haag"));
    }
    protected Analyzer getAnalyzer() {
        return new WhitespaceAnalyzer();
    }
    private int getHitCount(String fieldName, String searchString) throws IOException {
        IndexSearcher searcher = new IndexSeracher(dir);
        Term t = new Term(fieldName, searchString);
        Query query = new TermQuery(t);
        Hits hits = searcher.search(query);
        int hitCount = hits.length();
        searcher.close();
        return hitCount;
    }
}
```

我们首先删除了 city 字段含有 Amsterdam 的所有 Document；然后添加一个字段与删除的 Document 相同的新 Document，除了把 city 字段设了一个新值。新的 Document 的 city 字段是 Haag 而不是 Amsterdam。我们正确地更新了索引中的一个 Document。

通过定量删除来更新

我们的例子删除和添加单个 Document。如果你需要删除和添加多个 Document，最好是进行批操作。按以下步骤：

1. 打开 IndexReader。
2. 删除所有你要删除的 Document。
3. 关闭 IndexReader。
4. 打开 IndexWriter。
5. 添加你要添加的所有 Document。
6. 关闭 IndexWriter。

要记住：批量 Document 删除和索引总是比交错执行删除和添加操作快。

懂得了更新和删除操作，让我们讨论如何提升索引的性能并尽可能好地利用硬件资源。

技巧　当删除和添加 Document 时，批量进行。这样总是要比交叉删除和添加操作快。

2.3 Document 和 Field 增量

并不是所有的 Document 和 Field 是平等创建的——或者至少你能确定选择性的 Document 或 Field 增量的情况。假设你要写一个索引和搜索公司 Email 的程序。可能需求是给公司员工的 Email 比其它 Email 消息更多的重要性。你会如何做呢？

Document 增量是个使得这种需求能够简单实现的一个特征。默认情况下，所有的 Document 都没

有增量——或者更恰当地说，它们都有相同的增量因数 1.0。通过改变某个 Document 的增量因数，你可能让 Lucene 认为它比索引中的其他 Document 更重要或不重要。执行这些的 API 只需一个方法，setBoost(float)，可以这样用：

```
public static final String COMPANY_DOMAIN = “example.com”;
public static final String BAD_DOMAIN = “yucky-domain.com”;

Document doc = new Document();
String senderEmail = getSenderEmail();
String senderName = getSenderName();
String subject = getSubject();
String body = getBody();
doc.add(Field.Keywork(“senderEmail”, senderEmail));
doc.add(Field.Text(“senderName”, senderName));
doc.add(Field.Text(“subject”, subject));
doc.add(Field.UnStored(“body”, body));
if (getSenderDomain().endsWithIgnoreCase(COMPANY_DOMAIN)) {
    doc.setBoost(1.5);          ① 员工增量因数：1.5
} else if (getSenderDomain().endsWithIgnoreCase(BAD_DOMAIN)) {
    doc.setBoost(0.1);          ② Bad 域增量因数：0.1
}
writer.addDocument(doc);
```

在本例中，我们检查邮件域名来决定发件人是不是公司的员工。

① 当我们索引由公司员工发送的消息时，我们把他们的增量因数设为 1.5，这默认的因数 1.0 大。

② 当我们碰到来自虚构的不可信域的发件人发送的消息时，我们通过把它们的增量因数隐为 0.1 把它们归类为不重要的。

就象你可以增量 Document 一样，你也可以增量个别的字段。当你增量 Document 时，Lucene 内部使用相同的增量因数增量它的每个字段。假设 Email 索引程序的另一个需求是考虑标题字段比发件人的名称字段更重要。换句话说，搜索时对标题字段的匹配比同样对 senderName 字段的匹配更有价值。为了完成这个计划，我们使用 Field 类的 setBoost(float)方法：

```
Field senderNameField = Field.Text(“senderName”, senderName);
Field subjectField = Field.Text(“subject”, subject);
subjectField.setBoost(1.2);
```

在本例中，我们随便选了一个增量因数 1.2，就像我们随便为 Document 选了一个增量因数 1.5 和 0.1 一样。你要用的增量因数值取决于你要达到什么目的；你可能需要做一些实验和调整来达到预期的目标。

值得注意的是字段可以有和它们相关联的固定增量，是由于 Lucene 的算分方式。增量总得来说是个高级特征，没有它很多程序也能工作得很好。

Document 和 Field 增量在搜索时起作用，你将在 3.5.9 小节中学到。Lucene 的搜索结果根据每个 Document 与查询的接近程度来分级，每个匹配的 Document 分被赋于一个分值。Lucene 的评分规则受许多因素影响，增量因数是其中之一。

2.4 索引日期

邮件含有发送和接收日期，文件有很多相关的日期，HTTP 呼应有一个包含请求页面最后修改日期 Last-Modified 头。像很多其他 Lucene 用户一样，你可能需要索引日期。Lucene 带有一个 Field.Keyword(String, Date)方法，还有 DateField 类，这使得索引日期很简单。例如，为了索引当前日期，可以这样：

```
Document doc = new Document();
```

doc.add(Field.Keyword(“indexDate”, new Date()));

在内部，Lucene 使用 DateField 类把给定的日期转化成适于索引的字符串。这样处理日期比较简单，但是你在使用这个方法时必须小心：使用 DateField 把日期转化成可以索引的 String，包括日期的所有部分，甚至毫秒。你将在 6.5 节中看到，这可能会导致某些查询的性能问题。实际上，你很少需要精确到毫秒的日期，至少对查询来说是这样。通常，你可以把日期近似到小时或着甚至天。

因为所有 Field 值最后都会转化成文本，你可以像 String 一样索引日期。例如，如果你把日期近似到天，以 YYYYMMDD 格式的 String 索引日期，可以使用 Field.Keyword(String, String)方法。运用这种方法的原因是你能够索引在 Unix Epoch(1970 年 1 月 1 日)之前的日期，DateField 不能处理。尽管一些解决这个限制的修补在近几年来被很多人提出，但是没有一个很完美。所以他们只能在 Lucene 补丁列表中找到，而没有包含在 Lucene 中。根据 Lucene 用户提到这个限制的频率，不能索引 1970 年之前的日期通常不是个问题。

> 注意　如果你仅需要对日期进行搜索，而不是时间，用 Field.Keyword(“date”, “YYYYMMDD”)。如果你要取得完整的时间，用 Field.Keyword(“timestamp”, <java.util.Date>)索引另一个 Field。

如果你想把日期或时间格式化为其它形式，一定注意 String 是以字典顺序排序的；这样做允许期间(date-range)查询。以 YYYYMMDD 格式索引日期的一个好处是能够仅用年来查询，或用年和月，或者精确地年和月和日。仅用年查询，使用 PrefixQuery。我们将在 3.4.3 小节中深入讨论 PrefixQuery。

2.5 索引数值

有两种常见的情况，数值索引非常重要。第一种，数值包含在要索引的文本中，并且你想确定这些数值被索引了，这样你就能在后来的搜索中使用它们。例如，你的文档可能包含“ 珠穆朗玛峰高 8848 米。”你想像搜索珠穆朗玛峰时找到含有这个句子的文档一样搜索数字 8848。

第二种情况，有只含有数值的 Field，并且你想索引和搜索它们。此外，你可能想利用这些 Field 来执行范围内搜索(range queries)。例如，如果你要索引邮件消息，有个字段保存邮件大小，你可能想找出所有指定大小的消息；或者，你可能想使用范围搜索来找出所有大小在一个特定范围内的消息。你可能也要根据大小来排序结果。

Lucene 能够通过在内部把它们处理成字符串来索引数值。如果你需要索引自由格式文本中的数值，你要做的第一件事就是选择不过滤数字的 Analyzer。在第 4.3 节中将讨论到的，WhiteSpaceAnalyzer 和 StandardAnalyzer 是两个可能的选择。如果你传递给他们一个类似“ 珠穆朗玛峰高 8848 米 ”的句子，他们把 8848 提取为一个单词，并把它传到索引过程，允许稍后搜索 8848。另外，SimpleAnalyzer 和 StopAnalyzer 将把数字从单词流中分离出来，这意味着搜索 8848 不会有任何匹配的文档。

Field 本身的值是数字时不应该被分析，这样它们应该索引为 Field.Keyword。然而，除了仅仅将它们的原始值添加到索引中之外，还需要对它处理一下，这样可以使范围查询有效。在执行范围查询时，Lucene 使用 Field 的辞典值来排序。假设三个数值 Field 的值分别为 7、71 和 20。尽管它们的自然顺序为 7、20、71，它们的辞典顺序为 20、7、71。解决这种不一致情况的一个简单而通用的方案是使用 0 前置数值 Field，例如：007、020、071。注意这时自然顺序和辞典顺序就一致了。更多关于搜索数值型 Field 的细节，请看 6.3.3 小节。

> 注意　当你索引数值型 Field 时，如果你想使用范围搜索，在它们左边补 0。

2.6 索引用来排序的 Field

当返回搜索结果时，Lucene 默认以他们的分数进行排序。有时，你需要用其它标准来对结果排序。例如，如果你正在搜索邮件，你可能想根据发送或接受时间又或者邮件大小来排序。如果你想利用 Field 值来排序，你必须以被索引但不分词的 Field 形式(如 Field.Keywork)添加。用来排序的 Field 必须能转换成整型、浮点型或字符串型：

Field.Keyword(“size”, “4096”);

Field.Keyword(“price”, “10.99”);

Field.Keyword(“author”, “Arthur C. Clark”);

尽管我们在把数值按字符串进行索引，但你可以在排序时指定正确的 Field 类型，将在 5.1.7 小节中描述。

注意　　用来排序的 Field 必须被索引，但不能被分词。

2.7 控制索引过程

索引小型或中型的文档集合时用 Lucene 默认的配置就能工作得很好。然而，如果你的程序要处理巨型索引，你可能想对 Lucene 的索引过程进行某些控制以获得更好的索引性能。例如，你要索引数十亿的文档并想加速这个过程让它在数分钟而不是数小时内完成。你的计算机可能有多余的内存，但你需要知道如何让 Lucene 利用它。Lucene 有很多参数，使你可以控制在索引时的性能和使用的资源。

2.7.1 调整索引性能

在典型的索引程序中，瓶颈是把索引文件写入磁盘。如果你分析一个索引程序，你会发现大多数时间花费在处理索引文件的代码片断上。因此，你需要让 Lucene 在索引新 Document 和修改已存在的索引文件时足够智能。

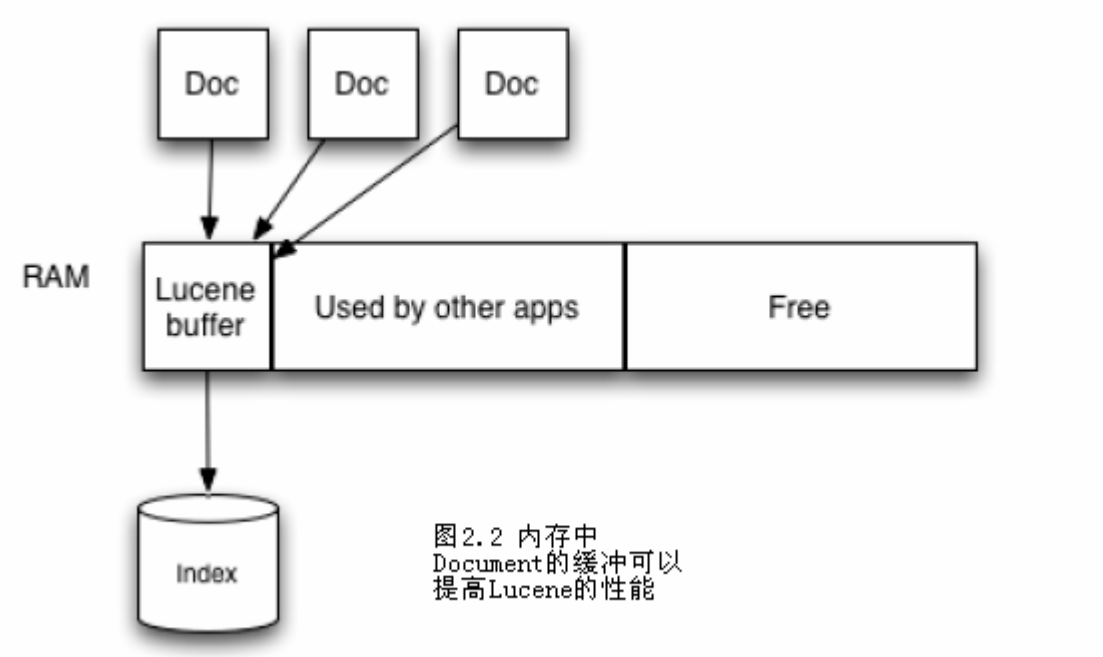

图2.2 内存中Document的缓冲可以提高Lucene的性能

如果 2.2 所示，当新的 Document 添加到 Lucene 索引中时，它们最初放在内存中，而不是立即写入磁盘。这个缓冲是基于性能的考虑；幸运的是 IndexWriter 类公开了几个实例变量，以允许你调整缓冲的大小和写磁盘的频率。这些变量的总结见表 2.1。

表 2.1 索引性能调节的参数

IndexWriter 变量	系统属性	默认值	描述
mergeFactor	org.apache.lucene.mergeFactor	10	控制合并块的频率和大小
maxMergeDocs	org.apache.lucene.maxMergeDocs	Integer.MAX_VALUE	限制每块的 document 数目
minMergeDocs	org.apache.lucene.minMergeDocs	10	控制索引时用的内存数量

IndexWriter 的 mergeFactor 属性使你可以控制在写入磁盘之前可以在内存中存储多少 Document，同时也是合并多个索引块的频率。（索引块在附录 B 中说明。）默认值设为 10，Lucene 写入磁盘之前在单个块中存储 10 个 Document。mergeFactor 的值为 10 同时也表示一旦磁盘上的块数到了 10，Lucene 将把这些块合并为一个块。

例如，如果你把 mergeFactor 设为 10，添加到索引中的每 10 个 Document 就会创建一个新的块。当第 10 个块中有 10 个 Document 时，这 10 个块将合并为 1 个含有 100 个 Document 的块。当有 10 个含有 100 个 Document 的块时，它们将合并到 1 个含有 1000 个 Document 的块中，依此类推。因此，任何时刻，一个索引中都不有超过 9 个块，而且每个合并块的大小都是 10 的幂。

针对这一规则的特殊情况需要用 maxMergeDocs 来处理，IndexWriter 的另一个实例变量：当合并分块时，Lucene 保证没有生成包含多于 maxMergeDocs 个 Document 的块。例如，假设你把 maxMergeDocs 设为 1000。当你添加第 10000 个 Document 时，Lucene 创建第 10 个大小为 1000 的块，而不是合并为一个 10000 个 Document 的块，以保证每 1000 个 Document 就创建一个新块。

既然你已经看到 mergeFactor 和 maxMergeDocs 如何工作，就能得出：使用大一点的 mergeFactor 导致 Lucene 使用更多的内存减少写磁盘的次数，一般会加速索引过程。把 mergeFactor 设置的小会使用更少的内存并导致更频繁地更新索引，这虽然会使索引是最新但却减慢了索引过程。同样的，大一点的 maxMergeDocs 对于批量索引来说更合适，小一点的 maxMergeDocs 对交互性强的索引好一些。大一点的 mergeFactor 意味着较少的合并，它会导致更多的索引文件。尽管这并不影响索引性能，但它可能会使搜索变慢，因为 Lucene 需要打开、读取和处理更多的索引文件。

minMergeDocs 是另一个影响性能的 IndexWriter 实例变量。它的值控制在合并为块之前必须要缓冲多少 Document。minMergeDocs 参数使你可以更多利用内存来加速索引。和 mergeFactor 不同，这个参数不影响磁盘上索引块的大小。

示例：IndexTuningDemo

为了更好地体验不同的 mergeFactor、maxMergeDocs 和 minMergeDocs 的值如何影响索引速度，考虑列表 2.4 所示的 IndexTuningDemo 类。这个类使用四个命令行参数：要添加到索引中的 Document 总数、mergeFactor 的值、maxMerge 的值和 minMergeDocs 的值。这四个参数必须都要指定，必须为整型并且必须以这个顺序指定。为了保证代码尽量简洁，没有对使用方法的检验。

列表 2.4 使用mergeFactor、maxMergeDocs和minMergeDocs的示例

```
Public class IndexTuningDemo {
    Public static void main (String[] args) throws Exception {
        Int docsInIndex = Integer.parseInt(args[0]);

        // create an index called 'index-dir' in a temp directory
        Directory dir = FSDirectory.getDirectory(
            System.getProperty("java.io.tmpdir", "tmp") +
            System.getProperty("file.separator") + "index-dir", true);
        Analyzer analyzer = new SimpleAnalyzer();
        IndexWriter writer = new IndexWriter(dir, analyzer, true);

        // set variables that affect speed of indexing
        writer.mergeFactor = Integer.parseInt(args[1]);          | 调整影响
        writer.maxMergeDocs = Integer.parseInt(args[2]);         | 索引性能
        writer.minMergeDocs = Integer.parseInt(args[3]);         | 的设置
        writer.infoStream = System.out;   ② 使 IndexWriter 将信息
                                            输出到 System.out 上
        System.out.println("Merge factor:   " + writer.mergeFactor);
        System.out.println("Max merge docs:   " + writer.maxMergeDocs);
```

```
            System.out.println("Min merge docs:    " + writer.minMergeDocs);

            long start = System.currentTimeMillis();
            for (int i = 0; i < docsInIndex; i++) {
                Document doc = new Document();
                doc.add(Field.Text("fieldname", "Bibamus"));
                writer.addDocument(doc);
            }
            writer.close();
            long stop = System.currentTimeMillis();
            System.out.println("Time: " + (stop – start) + " ms");
        }
}
```