目录

§1 工具使用

§1.1 杂项

json 格式的 array 中的 element 是保持顺序的（顺序敏感的），而其它类型不是顺序敏感的。

Eclipse 中删除无用的 import:《Ctrl+ Shift + O》do it manually or Window -> Preferences -> Java -> Editor -> Save Actions -> Organize Imports to have it organized automatically whenever save a class。或者在 Package Explorer 窗口中，使用《Ctrl+ Shift + O》快捷键。

安装 Mate 桌面环境后，会碰到没有声音的情况，此时可以删除 pulseaudio，执行命令后重启机器。

§1.2 Linux Shell

使用 nutch 时，如果目录中文件太多，且包含了很多不需要的类型（如音频，视频），可使用 rsync 命令，保持源文件夹目录结构，同时只提取指定类型。

```
rsync -av --exclude='path1/exclude' --exclude='path2/
exclude' [source] [dest]
rsync -av --include='*.shtml' --include='*.doc' --include='*.pdf' --
include='*/' --exclude='*' /mnt/d data/
```

注意 [source] 与 [source/] 不同，前者表示拷贝 source 目录，后者表示拷贝目录下的内容。如果需要过滤的文件太多，可使用 [--exclude-from=FILE] 参数，其中 FILE 存储需过滤的文件或目录。也可过滤类型，如 [--exclude=*/.svn*]。

```
sudo apt-get autoremove pulseaudio
```

标准输入 stdin，标准输出 stdout，标准错误 stderr，其文件描述符分别为 0,1,2。> 默认为将标准输出（stdout）重定向到其它地方，2>&1 表示将标准错误输出（stderr）重定向到标准输出（stdout）。&>file 表示把标准输出（stdout）和标准错误（stderr）都输出到文件 file 中。特别注意，2>1 表示将 stderr 重定向到文件 1 中，而非 stdout。

let 命令的替代表示形式是: ((算术表达式))。例如，let 'j=i*6+2' 等价于 ((j=i*6+2))。当表达式中有 Shell 的特殊字符时，必须用双引号或单引号将其括起来。例如,let ``val=a|b''。如果不括起来，Shell 会把命令行 let

val=a|b 中的''|'' 看作管道符号，将其左右两边看成不同的命令，因此，将无法正确执行。

linux 中除了常见的读（r）、写（w）、执行（x）权限以外，还有 3 个特殊的权限，分别是 setuid、setgid 和 stick bit。

```
[root@MyLinux ~]# ls -l /usr/bin/passwd /etc/passwd
-rw-r--r-- 1 root root  1549 08-19 13:54 /etc/passwd
-rwsr-xr-x 1 root root 22984 2007-01-07 /usr/bin/passwd
```

/etc/passwd 文件存放的各个用户的账号与密码信息，/usr/bin/passwd 是执行修改和查看此文件的程序，但从权限上看，/etc/passwd 仅有 root 权限的写（w）权，可实际上每个用户都可以通过/usr/bin/passwd 命令去修改这个文件，于是这里就涉及了 linux 里的特殊权限 setuid，正如 -rwsr-xr-x 中的 s，setuid 就是：让普通用户拥有可以执行“只有 root 权限才能执行”的特殊权限，setgid 同理是让普通用户拥有“组用户”才能执行的特殊权限”。

/tmp 目录是所有用户共有的临时文件夹，所有用户都拥有读写权限，这就必然出现一个问题，A 用户在/tmp 里创建了文件 a，此时 B 用户看了不爽，在/tmp 里把它给删了（因为拥有读写权限），那肯定是不行的。实际上不会发生这种情况，因为有特殊权限 stick bit（粘贴位）权限，drwxrwxrwt 中的最后一个 t 的意思是：除非目录的 owner 和 root 用户才有权限删除它，除此之外其它用户不能删除和修改这个目录。也就是说，/tmp 目录中，只有文件的拥有者和 root 才能对其修改和删除，其他用户则不行，避免了上面所说的问题。

如何设置以上特殊权限，如下所示，suid 的二进制串为 100，换算十进制为 4，guid 的二进制串为 010，stick bit 二进制串为 001，换算成 1。在一些文件设置了特殊权限后，字母不是小写的 s 或者 t，而是大写的 S 和 T，那代表此文件的特殊权限没有生效，是因为尚未赋予它对应用户的 x 权限。

```
setuid: chmod u+s xxx              setuid:chmod 4755 xxx
setgid: chmod g+s xxx              setgid:chmod 2755 xxx
stick bit : chmod o+t xxx          stick bit:chmod 1755 xxx
```

§1.3 git 基本命令

git add 命令主要用于把要提交的文件的信息添加到索引库中，当使用 **git commit** 时，git 将依据索引库中的内容来进行文件的提交。**git add <path>** 表示 add to index only files created or modified and not those deleted，因此 **git add <path>** 添加的文件不包括已经删除了的文件，<path> 可以是文件也可以是目录。git 不仅能判断出 <path> 中，修改（不包括已删除）了的文件，还能判断出新建的文件，并把它们的信息添加到索引库中。

git add -u <path> 表示 add to index only files modified or deleted and not those created，即把 <path> 中所有的 tracked 文件

添加到索引库，它不会处理 untracked 的文件。*git add -A <path>* 表示把 <path> 中所有的 tracked 文件和所有的 untracked 的文件信息添加到索引库。当省略 <path> 时，<path> 等价于'./'（当前目录）。

使用 *git add ./* 后，如果打算撤销这次添加，则使用命令: *git rm -r --cached ./*。如果忽略某些类型或某些目录时，可在.gitignore 文件中添加内容，最后一行忽略 *mapreduce/ESMapReduce/lib/* 目录下的所有内容。

```
*.class
*.jar
*.ear
mapreduce/ESMapReduce/lib/
```

git 提交环节存在三大部分: *working tree,index file,commit*。working tree 是工作所在的目录，每当在代码中进行了修改，working tree 的状态就改变了。index file 是索引文件，它是连接 working tree 和 commit 的桥梁，每当使用 git add 命令后，index file 的内容就改变了，此时 index file 和 working tree 完成了同步。commit 是代码的一次提交，只有完成提交，代码才真正地进入 git 仓库，使用 git commit 就是将 index file 内容提交到 commit 中。因此，**git diff** 是查看 working tree 与 index file 的差别的，**git diff --cached** 是查看 index file 与 commit 的差别的，**git diff HEAD** 是查看 working tree 和 commit 的差别的（HEAD 代表最近一次 commit）。

发生错误时，如果想回到此前的一个版本，命令为 **git reset --soft**。使用 **git log** 命令，查看日志中有哪些 commit 以及 commit 的 ID，最后使用命令 **git reset commitID** 回退到该版本。

```
git branch 查看本地有哪些分支
git branch -r 查看远程仓库有哪些分支
下载远程仓库（origin/1.2）到本地，分枝名同样为1.2
git checkout -b 1.2 origin/1.2
git checkout master 切换到master分支
git checkout 1.2 切换到1.2分支
```

§1.3.1 不同仓库的操作

(1) git branch -r 查看远程仓库有哪些分支

(2) git clone gitosis@192.168.2.4:xhw/soul-client

(3) 执行 <git init>，修改当前目录下的 <.git/config> 文件，将 xhw 改为 liubo

(4) 执行 <git remote -v> 查看有哪些远程仓库

(5) git remote add smq gitosis@192.168.2.4:/smq/soul-client

(6) git remote add chen gitosis@192.168.2.4:/chengjie/soul-client

(7) 再次执行 <git remote -v> 查看远程仓库有何更新

(8) 执行 <git pull smq master> 拉取更新

§1.4 Maven 入门

(1) 配置 maven 环境，最主要的是设置环境变量: M2_HOME，将其设置为 maven 安装目录，例子目录为: /usr/share/maven;

(2) 修改仓库位置，仓库用于存放平时项目开发依赖的所有 jar 包。例子仓库路径: /opt/maven/repo，为设置仓库路径，必须修改 **$M2_HOME/conf** 目录下的 setting.xml 文件。
<localRepository>/opt/maven/repo</localRepository>
在 shell 中输入并执行 mvn help:system，如果没有错误，在仓库路径下应该多了些文件，这些文件是从 maven 的中央仓库下载到本地仓库的。

(3) 创建 maven 项目，通过 maven 命令行方式创建一个项目，命令为: **mvn archetype:create -DgroupId=com.mvn.test -DartifactId=hello -DpackageName=com.mvn.test -Dversion=1.0**
由于第一次构建项目，所有依赖的 jar 包都要从 maven 的中央仓库下载，所以需要时间等待。做完这一步后，在工程根目录下应该有个 pom.xml 文件，其中 groupId，artifactId 和 version 比较常用。

- project: pom.xml 文件的顶层元素;
- modelVersion: 指明 POM 使用的对象模型的版本，这个值很少改动。
- groupId: 指明创建项目的小组的唯一标识。GroupId 是项目的关键标识，此标识以组织的完全限定名来定义。如 org.apache.maven.plugins 是所有 Maven 插件项目指定的 groupId。
- artifactId: 指明此项目产生的主要产品的基本名称。项目的主要产品通常为一个 jar 包，源代码包通常使用 artifactId 作为最后名称的一部分。典型的产品名称使用这个格式: <artifactId>-<version>.<extension>(比如: myapp-1.0.jar)。
- version: 项目产品的版本号。Maven 帮助你管理版本，可以经常看到 SNAPSHOT 这个版本，表明项目处于开发阶段。

- name: 项目的显示名称，通常用于 maven 产生的文档中。
- url: 指定项目站点，通常用于 maven 产生的文档中。
- description: 描述此项目，通常用于 maven 产生的文档中。

(4) 项目 hello 已经创建完成，但它并不是 eclipse 所需要的项目目录格式，需要把它构建成 eclipse 可以导入的项目。进入到刚创建的项目目录 (~/workspace/hello)，执行: **mvn clean**（告诉 maven 清理输出目录 target），然后执行 **mvn compile**（告诉 maven 编译项目 main 部分的代码，或者两步合成一步 **mvn clean compile**），此次仍会下载 jar 包到仓库中。编译后，项目的目录结构并不可以直接导入到 Eclipse，需执行命令: **mvn eclipse:eclipse**，命令执行完成后就能 import 到 Eclipse。

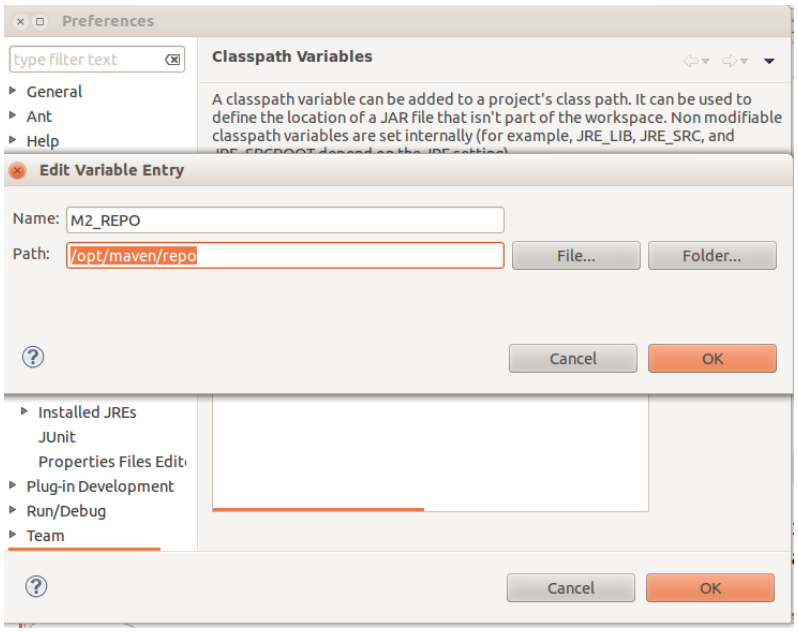

图 1-1 Eclipse 配置 Maven 仓库路径

(5) 打开 eclipse，在其中配置 maven 仓库路径，配置路径为: Window-->Perferences-->java-->Build Path-->Classpath Variables，新建变量（M2_REPO）的类路径，如图 1-1 示。

(6) 包的更新与下载，如果发现 junit 版本比较旧，想换成新版本，修改项目下的的 pom.xml 文件为

```
<dependencies>
  <dependency>
     <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>4.8.1</version>
```

```
      <scope>test</scope>
    </dependency>
  </dependencies>
```

就可改变 junit 的版本号，在以后的 maven 操作中,maven 会自动下载依赖的 jar 包。Maven 中央仓库地址为 http://search.maven.org，假如想下载 struts 的 jar 包，可在 url 内搜索 struts。

(7) 某些 jar 包可能位于远程机器上，因此需要配置 maven 仓库，配置仓库代码如下，其中的 id 属性无特别含义，仅用于标识仓库。当下载完某个 jar 包后，其在本地仓库的相对路径形如 <groupId>/<artifactId>/<version>/*.jar，例如: /opt/maven/repo/org/ansj/tree_split/1.0.1/tree_split-1.0.1.jar。

```
<repositories>
  <repository>
    <id>ansj-maven-repo</id>
    <url>https://raw.github.com/ansjsun/mvn-repo/gh-pages</url>
  </repository>
  <repository>
    <id>remote-maven-repo</id>
    <url>http://search.maven.org</url>
  </repository>
</repositories>
```

(8) 有时候 Maven 访问很慢，可在 settings.xml 里重新设置中央仓库的镜像，中央仓库的默认地址为:http://repo.maven.apache.org/maven2/

```
<mirrors>
 <mirror>
   <id>ibiblio</id>
   <mirrorOf>central</mirrorOf>
   <name>Human Readable Name for this Mirror.</name>
   <url>http://mirrors.ibiblio.org/maven2</url>
 </mirror>
</mirrors>
```

打算执行 maven 工程下某个类时，例如 test 目录下的某个类，执行命令如下: **mvn exec:java -X -Dexec.mainClass="org.ansj.liubo.test.test" -Dexec.classpathScope=test**，其中的 classpathScope=test 告诉 maven 执行 test 类，而非 main 类。执行 test 部分的某些类之前，必须执行 **mvn test-compile**，以编译 test 部分的代码（test 类中必须有 main 函数），如没有编译，则执行会报错。当准备向执行类添加参数时，使用如下命令。

```
mvn exec:java -Dexec.mainClass="org.elasticsearch.application.WebDemo"
  -Dexec.args="/mnt/f/tmp/content.txt /mnt/f/tmp/result3.txt"
  -Dexec.classpathScope=test
## 如下命令执行TestNG框架中的test类(没有main函数)
mvn test -Dtest="com.soul.elasticsearch.test.OfficialDataTest"
```

§1.5 Maven 常用插件

Maven 本质上是一个插件框架，其核心并不执行任何具体的构建任务，所有这些任务都交给插件完成，例如编译源代码由 maven-compiler-plugin 完成。进一步说，每个任务对应一个插件，每个插件会有一个或者多个目标（goal），例如 maven-compiler-plugin 的 compile 目标用来编译 src/main/java/目录的 code，而 testCompile 目标则用来编译 src/test/java/目录的 code。

用户可通过两种方式调用 Maven 插件目标。第一种方式是将插件目标与生命周期阶段（lifecycle phase）绑定，这样用户在命令行只输入了生命周期阶段，例如 Maven 默认将 maven-compiler-plugin 的 compile 目标与 compile 生命周期阶段绑定，因此命令 **mvn compile** 实际上先定位到 compile 这一生命周期阶段，然后再根据绑定关系调用 maven-compiler-plugin 的 compile 目标。第二种方式是直接在命令行指定要执行的插件目标，例如 **mvn archetype:generate** 就表示调用 maven-archetype-plugin 的 generate 目标，这种带冒号的调用方式与生命周期无关。

Maven 有两个插件列表，第一个列表 GroupId 为 org.apache.maven.plugins，这里的插件最为成熟，具体地址为：http://maven.apache.org/plugins/index.html。第二个列表的 GroupId 为 org.codehaus.mojo，这里的插件没有那么成熟，但也十分有用，地址为：http://mojo.codehaus.org/plugins.html。

为使项目结构更为清晰，Maven 区别对待 Java 代码文件和资源文件，maven-compiler-plugin 用来编译 java 代码，maven-resources-plugin 则用来处理 resource 文件。默认的资源文件目录是 src/main/resources。很多用户会添加额外的资源文件目录，这个时候就可以通过配置 maven-resources-plugin 来实现。此外，资源文件过滤也是 Maven 的一大特性，可以在资源文件中使用 **$propertyName** 形式的 Maven 属性，然后配置 maven-resources-plugin 以开启对资源文件的过滤，之后就可以通过命令行或者 Profile，针对不同环境传入不同的属性值，以实现灵活构建。

由于历史原因，Maven 用于执行测试的插件不是 maven-test-plugin，而是 maven-surefire-plugin。其实大部分时间内，只要测试类遵循通用的命令约定（以 Test 结尾、以 TestCase 结尾、或者以 Test 开头），就几乎不用知晓该插件是否存在。想要跳过测试、排除某些测试类、或者使用一些 Test 特性时，了解 maven-surefire-plugin 的一些配置选项就很有用了。例如 **mvn test -Dtest=FooTest** 这样一条命令的效果是仅运行 FooTest 测试类，这是通过控制 maven-surefire-plugin 的 test 参数实现的。

Maven 默认只允许指定一个主 Java 代码目录和一个测试 Java 代码目录，虽然这是一个应当尽量遵守的约定，但偶尔用户还是希望能够指定多个源码目录，

build-helper-maven-plugin 的 add-source 目标就服务于这个目的，通常它被绑定到默认生命周期的 generate-sources 阶段以添加额外的源码目录。这种做法是不推荐的，因为它破坏了 Maven 的约定，而且可能会遇到其他严格遵守约定的插件工具无法正确识别额外的 source 目录。build-helper-maven-plugin 的另一个非常有用的目标是 attach-artifact，使用该目标你可以以 classifier 的形式选取部分项目文件生成附属构件，并同时 install 到本地仓库，也可以 deploy 到远程仓库。

exec-maven-plugin 很好理解，顾名思义，它能让你运行任何本地的系统程序，在某些特定情况下，运行一个 Maven 外部的程序可能就是最简单的问题解决方案，这就是 exec:exec 的用途，当然，该插件还允许你配置相关的程序运行参数。除了 exec 目标之外，exec-maven-plugin 还提供了一个 java 目标，该目标要求你提供一个 mainClass 参数，然后它能够利用当前项目的依赖作为 classpath，在同一个 JVM 中运行该 mainClass。有时候，为了简单的演示一个命令行 Java 程序，可以在 pom.xml 中配置好 exec-maven-plugin 的相关运行参数，然后直接在命令运行 **mvn exec:java** 以查看运行效果。

进行 Web 开发时，打开浏览器对应用进行手动的测试几乎是无法避免的，这种测试方法通常就是将项目打包成 war 文件，然后部署到 Web 容器中，再启动容器进行验证，这显然十分耗时。为了帮助开发者节省时间，jetty-maven-plugin 应运而生，它完全兼容 Maven 项目的目录结构，能够周期性地检查源文件，一旦发现变更后自动更新到内置的 Jetty Web 容器中。做一些基本配置后（例如 Web 应用的 contextPath 和自动扫描变更的时间间隔），只要执行 **mvn jetty:run**，然后在 IDE 中修改代码，代码经 IDE 自动编译后产生变更，再由 jetty-maven-plugin 侦测到后将更新写入到 Jetty 容器，这时就可以直接测试 Web 页面。需要注意的是，jetty-maven-plugin 并不是宿主于 Apache 或 Codehaus 的官方插件，因此使用的时候需要额外的配置 settings.xml 的 pluginGroups 元素，将 org.mortbay.jetty 这个 pluginGroup 加入。

当项目包含大量模块的时候，集体更新版本就变成一件烦人的事情，versions-maven-plugin 提供了很多目标帮助你管理 Maven 项目的各种版本信息。例如最常用的命令 **mvn versions:set -DnewVersion=1.1-SNAPSHOT** 就能帮助你把所有模块的版本更新到 1.1-SNAPSHOT。该插件还提供了其他一些很有用的目标，display-dependency-updates 能告诉你项目依赖有哪些可用的更新，类似的 display-plugin-updates 能告诉你可用的插件更新，use-latest-versions 能自动帮你将所有依赖升级到最新版本。最后，如果对所做的更改满意，则可以使用 **mvn versions:commit** 提交，不满意的话也可以使用 **mvn versions:revert** 进行撤销操作。

§1.6 Building a Self-Contained Hadoop Job

Non-trivial Hadoop jobs usually have dependencies that go beyond those provided by the Hadoop runtime environment. That means, if your job needs additional libraries you have to make sure they are on Hadoop's classpath as soon as the job is executed. This article shows how you can build a self-contained job JAR that contains all your dependencies.

The Hadoop runtime environment expects additional dependencies inside a lib directory. That's where we need Maven's Assembly plugin: Using the plugin, we create a JAR that contains the project's resources (usually just your job's class files) and a lib directory with all dependencies that are not on Hadoop's classpath already.

First of all, we create an assembly descriptor file (put it in src/main/assembly/hadoop-job.xml). Note that we collect all dependencies on the runtime scope but exclude the project's artifact. Instead, we add the artifact unpacked which is a little surprising. If we didn't do that, Hadoop would look for our dependencies inside the project's artifact JAR and not inside the surrounding assembly JAR's lib directory. For the whole mechanism to work, the class you set in your job driver via Job.setJarByClass() must be from your project, not from your dependencies.

```
<assembly>
  <id>job</id>
  <formats>
    <format>jar</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <dependencySets>
    <dependencySet>
      <unpack>false</unpack>
      <scope>runtime</scope>
      <outputDirectory>lib</outputDirectory>
      <excludes>
        <exclude>${groupId}:${artifactId}</exclude>
      </excludes>
    </dependencySet>

    <dependencySet>
      <unpack>true</unpack>
      <includes>
```

```
        <include>${groupId}:${artifactId}</include>
      </includes>
    </dependencySet>
  </dependencySets>
</assembly>
```

Inside the POM's dependency section, we set Hadoop to the provided scope，这样的话，Hadoop 相关的 jar 包就不会被打包。

```
<dependencies>
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-core</artifactId>
    <version>2.0.5-alpha</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

Setting the assembly JAR's Main class is optional, but makes the job more user friendly because you don't have to specify the class name explicitly for each run. You can now build your job JAR:

```
mvn clean package
```

Your self-contained job JAR is the file in target ending with -job.jar. Run it using Hadoop's jar sub-command:

```
hadoop jar ./target/releases/soul_seg-0.3-job.jar /hdfs
```

使用上述命令打包时，将会运行 test 目录下的测试类，十分耗时。为此，使用如下命令跳过测试。

```
mvn clean package -Dmaven.test.skip=true
```

§1.7 构造 Java 后台程序

Lately I have been writing a Java program that needs to run in the background (like a daemon). These ideas probably only work in a Unix environment but they have been tested on Linux and Solaris. So you have your program and you want to start it such that it will not be killed when you log out of the shell in which you start it. You could use **nohup**, but **nohup** redirects the standard out and error to files, which is annoying because you are writing all output to a log file anyway.

```
java -jar target/soul_seg-0.3.jar  <&- 1>/dev/null 2>&1 &
```

其中，<&-关闭标准输入，1>/dev/null 2>&1将标准输入重定向到/dev/null，最后一个&挂起该进程。

上面的命令 runs the program in the background, closes standard in, and redirects standard out and error to the /dev/null. By closing standard in to the process the shell will not kill the program when it exits and it is running in the background so it will not interfere with other actions we might want to perform within this shell.

但后台程序发生错误时，上述方法不会提示，如当监听端口被占用时，因此，重定向 stdout 和 stderr 不合适。The solution is to have the Java program detach from standard out and error once its startup is complete. So we will create a method called daemonize() that we will call just before entering the infinite loop that is the main program logic.

```
static public void daemonize() {
   System.out.close();//先关闭标准输出
   System.err.close();//先关闭标准错误输出
   final String pidFile = "/tmp/a/pidFile";
   File fPidFile = new File(pidFile);
   FileOutputStream oStream = new FileOutputStream(fPidFile);
   oStream.write(Long.toString(JvmInfo.jvmInfo().pid()).getBytes());
   oStream.close();
   fPidFile.deleteOnExit();
}
static public void main(String[] args) {
   try {
       // do sanity checks and startup actions
       daemonize();
   }
   catch (Throwable e)  {
       System.err.println("Startup failed.");
       e.printStackTrace();
   }
   // do infinite loop
}
```

Now that the program is completely detached from the shell，the only way to stop it is by killing the process. However, to do that you need to know the pid. 获取进程 ID 依赖于 Java 系统库，在 daemonize() 函数中，将进程 ID 保存在文件中（一般在/tmp 目录下）。fPidFile.deleteOnExit() 表示，当进程崩溃或关闭时，自动删除文件。

§2 ElasticSearch Query

§2.1 ES Score Mechanism

Main Formula as:

$$score(q,d) = coord(q,d) \times queryNorm(q) \times \sum_{t \in q} (tf(t \in d) \times idf(t)^2 \times t.getBoost() \times norm(t,d))$$

• t:term，term 是包含域信息的 term，即 title:hello 和 content:hello 是不同的 term。

• coord(q,d): 一次搜索可能包含多个 term，而一篇文档中也可能包含多个 term，此项表示当一篇文档中包含的搜索词越多，则此文档分数越高。只会在 BooleanQuery 的 or 查询中该值可能小于 1，其余查询该值都为 1。

• queryNorm(q): 查询语句 q 的标准化值，此值不影响排序，而使得不同的 query 之间的分数可以比较。其公式为: $queryNorm(q) = \frac{1}{\sqrt{\sum_{t \in q}[idf(t) \times t.getBoost()]^2)}}$

• $tf(t \in d)$: 关于 Term t 在文档 d 中出现的词频的一个值，Lucene 默认实现: $\sqrt{freq}$。

• idf(t):Term t 在多少篇文档中出现过，显示 t 的稀有程度或普遍程度，Lucene 默认实现: $1 + \ln \frac{numDocs}{1+docFreq}$

• $norm(t,d) = d.getBoost() \times lengthNorm(field) \times f.getBoost()$

  - d.getBoost():Document boost，此值越大，说明此文档越重要。
  - f.getBoost():Field boost，此域越大，说明此域越重要。
  - lengthNorm(field): 域中包含的 Term 总数越多，也即文档越长，此值越小，文档越短，此值越大，公式: $\frac{1}{\sqrt{numTerms}}$。

lengthNorm 的值必须除以 Term 总数，在这里叫做归一化处理。Index 中不同文档的长度不一样，很显然，对于任意一个 term，长文档中的 tf 要大的多，因而分数高，这样对小文档很不公平。举一个极端例子，在一篇 1000 万个词的巨著中，"lucene" 这个词出现了 11 次，而在一篇 12 个词的短小文档中，"lucene" 这个词出现了 10 次，如果不考虑长度，巨著应该分数更高，但这篇小文档才是真正关注"lucene" 的。

§2.2 ES Boost Mechanism

The default scoring is the DefaultSimilarity algorithm in core Lucene, you can customize scoring by configuring your own Similarity, or using something like a custom_score query.

§2.2.1 Why Boost?

The first question I had when I started working with scoring was: why do I need to boost at all? Isn't Lucene's scoring algorithm tried and true? It seemed to work pretty well on the handful of test documents that I put in the index. But as I added more and more documents, the results got worse, and I realized why boosting is necessary. Lucene's scoring algorithm does great in the general case, but it doesn't know anything about your specific subject domain or application. Boosting allows you to compensate for different document types, add domain-specific logic, and incorporate additional signals.

Before I can give specific examples, I need to explain a little bit about the search application I've been working on. The application powers site search for IGN, a site about "gaming, entertainment, and everything guys enjoy." We currently index four main types of content from our backend APIs: articles, videos, wiki pages, and "objects" (games, movies, shows, etc.) By default, search results of all types are returned in a single aggregate listing.

Compensating for Different Document Types —Lucene's scoring algorithm works very well if your documents are homogeneous. But if you have different document types, you may need to make some manual adjustments. For example, we index both articles and videos. Articles have a lot of textual content —the entire body of the article —but videos only have a short description field. By default Lucene will prefer a match in a shorter field, so when videos match they tend to score higher than articles.

Since elasticsearch supports searching across multiple indexes, it may have been possible to compensate for different document types by creating separate indexes for each type and performing

searches with a multi-index query. I haven't tested it, but I think the scores from each query would be normalized by the coordination factor, so a top-scoring video would be given about the same weight as a top-scoring article. However, this approach would also calculate term frequencies for each content type individually, and I'm not sure how that would affect the results. Giving articles a small boost was a much simpler solution, especially since we already wanted to control how important each content type was for separate reasons.

Adding Domain-specific Logic —Sometimes you have domain-specific logic that is difficult for Lucene to discern on its own. For example, our review articles are probably one of the most important types of content on our site. Since our users are often looking for our reviews, we gave review articles a small boost so they would score higher than other articles.

Another example is stub wiki pages. Videos and objects are expected to have relatively short text descriptions. Articles are often longer, although sometimes we'll have short articles that announce a bit of news or promote some other content, so a short article is okay. However, a short wiki page is often a sign of a stub, so it should score lower than other results. This is opposite of what Lucene would have done on its own —Lucene would have preferred a match in the shorter wiki page and scored it higher.

Incorporating Additional Signals —For the most part, the importance of a particular piece of content on our site fades with time. For example, a review for a game that was just released may be important this week, but less so next month and even less so a year from now. Out of the box, Lucene does not consider the recency/freshness of content in its scoring algorithm. But if recency factors heavily into scoring in your domain, you may want to incorporate it using a boost. (More details on how to implement a recency boost can be found later in this post.)

We boost our game, movie, and TV show objects if we have written/created more articles and videos about them. A more generic example of this might be boosting products that have been purchased more,

or boosting articles that have more views or comments. Which attributes suggest importance is very domain-specific, so you have to handle it yourself with a boost.

§2.2.2 Boosting at Index Time vs Query Time

Boosts can be applied at index time, when the document is indexed, or at query time when a search is performed. If a particular document will always be more important than others, you may want to consider applying the boost when the document is indexed. Pre-boosted documents are faster to search because there is less work to do when a search is performed. However, even if you know that a document will always be more important, you may not know how much more important. If the boost is too strong, the important document will always appear at the top of results (as long as it matched at all); if the boost is too weak, the important document will not get any real advantage over other documents.

Applying a boost at index time requires that you re-index the document to change the boost. Unless you are manually adding documents to your index and deciding the boost on a case-by-case basis, you likely have some kind of script or program that is building your index and the boosts are determined by some logic or a set of rules. A change in the logic or rules will likely affect many documents, so you will effectively need to rebuild your index for the changes to take effect. If your index is small, this may be appropriate. Our index takes several hours to rebuild, so we avoid applying boosts at index time whenever possible. Applying boosts at query time let us add new boosts, change boost criteria, and change boost strength on-the-fly. This flexibility is well worth the additional runtime cost.

Although you can combine boosts applied at index time and at query time, some boosts must be applied at query time because there isn't enough information at index time to calculate the boost. For example, if you are doing a boost based on document freshness (how close the document's timestamp is to the current time), the current time (when the search is performed) is not known at index time. In this particular example, you could use the time the

document is indexed as the current time if the index is frequently rebuilt and you really want to avoid query time boosts.

§2.2.3 Implementing Boosts

Almost every elasticsearch query type has a boost parameter that allows you to influence the weight of that query versus other queries, but we don't use this parameter because we only have one main query. The main query is a query_string query, which parses the user's query, finds matches, and scores them using Lucene's default scoring algorithm. Then we apply a number of boosts depending on whether certain criteria is met.

In early prototypes, I accomplished the boosting by wrapping the main query in a custom_score query. As the name implies, the custom_score query allows you to calculate the score of each document using custom logic by passing in a script in the script parameter. By default, scripts are interpreted as MVEL, although other languages are supported. You can access the score assigned by the wrapped query via the special _score variable, so I started off doing something like this:

```
{
  "query": {
    "custom_score": {
      "query": { ...the main query... },
      "script": "_score * (doc['class'].value == 'review' ? 1.2 : 1)"
    }
  }
}
```

This worked, but it wasn't very scalable. As I added more boosts, I would end up with an expression with dozens of terms. Also, each document field would have to be stored at index time so that the script can retrieve and evaluate it, which bloats the index and is relatively slow.

Fortunately, there is a much better tool for the job —the custom_filters_score query.This can considerably simplify and increase performance for parameterized based scoring since filters are easily cached for faster performance, and boosting/script is considerably simpler. Converted to a custom_filters_score query, the above example looks like this:

```
{
  "query": {
    "custom_filters_score": {
      "query": { ...the main query... },
      "filters": [
        {
          "filter": {
            "term": {
              "class": "review"
            }
          },
          "boost": 1.2
        }
      ]
    }
  }
}
```

If you want to add additional boosts, just add another filter specifying the criteria and assigning it a boost. You can use any filter, including filters that wrap other filters. If you have multiple filters, you may want to specify how multiple matching filters will be combined by passing the score_mode parameter. By default, the first matching filter' s boost is used, but if you have multiple filters that may match you can set score_mode to something like multiply which would apply all the boosts.

The following query boosts reviews by 20%, boosts articles by 20% (so review articles would be boosted 44%), and penalizes wiki pages that are less than 600 characters long by 80%.

```
{
  "query": {
    "custom_filters_score": {
      "query": { ...the main query... },
      "filters": [
        {
          "filter": {
            "term": {
              "class": "review"
            }
          },
          "boost": 1.2
        },
        {
          "filter": {
            "term": {
              "type": "article"
            }
          },
```

```
          "boost": 1.2
        },
        {
          "filter": {
            "and": [
              {
                "term": {
                  "type": "page"
                }
              },
              {
                "range": {
                  "descriptionLength": {
                    "to": 600
                  }
                }
              }
            ]
          },
          "boost": 0.2
        }
      ],
      "score_mode": "multiply"
    }
  }
}
```

Sometimes you want to adjust the strength of a boost based on a field in the document. For example, if you want to boost recent documents, an article published today should be boosted more than an article published yesterday, and an article published yesterday should be boosted more than an article published last week, etc. Even though filters are cached and run relatively quickly, it would be impractical to have a filter for articles published today, another filter for articles published yesterday, another filter for articles published last week, etc. Fortunately, the custom_filters_score query can accept a script parameter instead of a boost for these situations.

Unfortunately elasticsearch doesn't have a recip function, but you can easily implement the same underlying function, y = a / (m * x + b), and pass it to elasticsearch as a script. In the example below, I'm using the values of m, a, and b suggested: m=3.16E-11, a=0.08, and b=0.05. Since some documents in our index have dates in the future, I added abs() to take the absolute value of the difference between the time the query is run and the document's timestamp. I'm also adding 1.0 to the boost value to make it a

freshness boost instead of a staleness penalty.

```
{
  "query": {
    "custom_filters_score": {
      "query": { ...the main query... },
      "params": {
       "now": current time when query is run, expressed as milliseconds
      },
      "filters": [
        {
          "filter": {
            "exists": {
              "field": "date"
            }
          },
          "script":
       "(0.08/((3.16*pow(10,-11))*abs(now-doc['date'].date.getMillis())
+0.05))+ 1.0"
        }
      ]
    }
  }
}
```

With these values, documents dated right now are boosted up to 160% (boost value is 2.6). This falls off to a 100% boost after 10 days, 60% after a month, 15% after 6 months, 8% after a year, and less than 4% after 2 years.

Elasticsearch scripts are cached for faster execution. When using scripts with elasticsearch, pass in values that change from query to query as a parameter via the params parameter rather than doing string interpolation in the script itself. This way, the script stays constant and cacheable, but your parameter still changes with every query.

§2.3 ElasticSearch Query

§2.3.1 Scroll Query

A search request can be scrolled by specifying the scroll parameter. The scroll parameter is a time value parameter (for example: scroll=5m), indicating for how long the nodes that participate in the search will maintain relevant resources in order to continue and support it. This is very similar in its idea to opening a cursor against a database.

A scroll_id is returned from the first search request (and from continuous) scroll requests. The scroll_id should be used when scrolling (along with the scroll parameter, to stop the scroll from expiring). The scroll id can also be passed as part of the search request body. The scroll_id changes for each scroll request and only the most recent one should be used.

```
curl -XGET 'http://localhost:9200/twitter/tweet/_search?scroll=5m' -
d '{
    "query": {
        "query_string" : {
            "query" : "some query string here"
        }
    }
}'
curl -XGET 'http://localhost:9200/_search/scroll?
scroll=5m&scroll_id=c2Nhbjs2OzM0NDg1ODpzRlBLc0FXNlNyNm5JWUc1'
```

Scrolling is not intended for real time user requests, it is intended for cases like scrolling over large portions of data that exists within elasticsearch to reindex it for example. The scan search type allows to efficiently scroll a large result set. It's used first by executing a search request with scrolling and a query:

```
curl -XGET 'localhost:9200/sogou_spellcheck/table/_search?
search_type=scan&scroll=10m&size=50' -d '{
    "query" : {
        "match_all" : {}
    }
}'
```

The scroll parameter control the keep alive time of the scrolling request and initiates the scrolling process. The timeout applies per round trip (i.e. between the previous scan scroll request, to the next).

The response will include no hits, with two important results, the **total_hits** will include the total hits that match the query, and the **scroll_id** that allows to start the scroll process. 在后续阶段，必须使用 **_search/scroll** 作为 endPoint，然后使用第一步获得的 **scroll_id** 作为查询参数。

```
curl -XGET 'localhost:9200/_search/scroll?scroll=10m' -d 'cNh***TmZ=='
```

上例中，注意 localhost:9200/_search/scroll 与 localhost:9200/sogou_spellcheck/table 的区别。Scroll requests will include a number of hits equal to the size multiplied by the number of primary shards.The "breaking" condition out of a scroll is when no hits has been returned. The total_hits will be maintained between scroll requests。但是，scan search type does not support sorting (either on score or a field) or faceting。

§2.3.2 查询类型

There are different execution paths that can be done when executing a distributed search. The distributed search operation needs to be scattered to all the relevant shards and then all the results are gathered back. When doing scatter/gather type execution, there are several ways to do that, specifically with search engines.

One of the questions when executing a distributed search is how much results to retrieve from each shard. For example, if we have 10 shards, the 1st shard might hold the most relevant results from 0 till 10, with other shards results ranking below it. For this reason, when executing a request, we will need to get results from 0 till 10 from all shards, sort them, and then return the results if we want to insure correct results.

Another question, which relates to search engine, is the fact that each shard stands on its own. When a query is executed on a specific shard, it does not take into account term frequencies and other search engine information from other shards. If we want to support accurate ranking, we would need to first execute the query against all shards and gather the relevant term frequencies, and then, based on it, execute the query.

Also, because of the need to sort the results, getting back a large document set, or even scrolling it, while maintaing the correct sorting behavior can be a very expensive operation. **For large result set scrolling without sorting, the scan search type is available.**

Elasticsearch is very flexible and allows to control the type of search to execute on a per search request basis. The type can

be configured by setting the search_type parameter in the query string（如上例中的 Scroll Query）。

§2.3.3 普通查询

假如索引中存在词组: [quick] [brown] [fox] [jump] [over] [lazy] [dog]。搜索“qu”时，Prefix Query 查询所有前缀为“qu”的 Term，因此如果索引中存在“quack”，“quote”和“quarter”等 Term，那么查询将匹配到这些 Term。如果索引中存在很多类似 Term 时，查询效率很差（注意，Prefix Query 不是精确匹配，效率肯定不高）。Prefix Query 也不能匹配中间字符，如“ball”不能匹配“baseball”，虽然可以采用通配符（Wildcard query）进行修正，但这种方式的查询性能更差。

如果输入英文时想匹配到目标串 [quick] 的任意前缀，应该抽取 [quick] 的前缀序列，其 Term 序列为: [q]/[qu]/[qui]/[quic]/[quick]，这种 Analyzer 称为 Edge NGram。还有一种 Analyzer 称为 N-Grams（字符串 P 的 N-Gram 是 P 中长度为 N 的所有子串），以“brown”这个单词为例，设置（minGram=1 和 maxGram=2），N-Grams 输出 [b]/[r]/[o]/[w]/[n]/[br]/[ro]/[ow]/[wn]，而 Edge NGram 输出 [b]/[br]（maxGram 等于 5 时，Edge NGram 能输出 [b]/[br]/[bro]/[brow]/[brown]）。针对 document 的不同域，应该设置不同的 Analyzer。在处理大量网页时，一般只需对标题和关键词建立 NGrams 索引，而网页内容采用普通的 Analyzer 即可。下面定义了名为 autocomplete 的 **Analyzer**，然后通过 **multi_field** 关键字添加了额外的 autocomplete 分析器。搜索时可使用 name 来访问 name 域的默认版本，或者使用 name.autocomplete 访问另一个版本。

```
{
  "settings":{
    "analysis":{
      "analyzer":{
        "autocomplete":{
          "type":"custom",
          "tokenizer":"standard",
          "filter":[ "standard", "lowercase", "stop", "kstem", "ngram" ]
        }
      }
    }
  }
}
{
  "articles":{
    "properties":{
      "name":{
        "type":"multi_field",
        "fields":{
```

```
          "name":{
            "type":"string"
          },
          "autocomplete":{
            "analyzer":"autocomplete",
            "type":"string"
          }
        }
      }
    }
  }
}
```

§2.3.4 布尔查询

A query that matches documents matching boolean combinations of other queries. The bool query maps to Lucene BooleanQuery. It is built using one or more boolean clauses, each clause with a typed occurrence. 例如:

```
{
    "query": {
        "bool": {
            "must": [
                {
                  "simple_query_string": {
                      "analyzer": "soul_query",
                      "default_operator": "and",
                      "fields": [ "content^1.0", "title^2.0" ],
                      "query": "中国"
                   }
                },
                { "term": { "tag": "行政服务" } }
            ]
        }
    }
}
```

布尔查询的主要作用是可将多个查询组合起来，注意下面查询中的 minimum_should_match，该变量必须设置 should 语句，否则不起作用，如果该变量没有提及，则默认为 0。

```
curl -XGET 192.168.50.75:9200/official_mini/table/_search?pretty -d '{
    "query": {
        "bool": {
            "must": [ {  "term": { "tag": "锡城资讯" } } ],
            "should": [
                {
                    "simple_query_string": {
                        "analyzer": "soul_query",
```

```
                    "default_operator": "and",
                    "fields": [ "content^1.0", "contenttitle^2.0" ],
                    "query": "太湖春涛"
                }
            },
            {
                "term": {
                    "contenttitle.untouched^4.0": "太湖春涛"
                }
            }
        ],
        "minimum_should_match" : 1
    }
  }
}'
```

§2.3.5 Span Multi Term Query

Matches spans which are near one another. One can specify slop, the maximum number of intervening unmatched positions, as well as whether matches are required to be in-order. The span near query maps to Lucene SpanNearQuery. Here is an example:

SpanNearQuery 主要用作精确查询，比如某个 term 之后，是另一个 term，term 之间的距离可以自己设定，从而实现精确匹配。例如搜索包含了“共青团中央下发实施意见”字符串的文章。不妨设“共青团中央下发实施意见”分词为：“共青团中央”，“下发”，“实施意见”。当设置 slop 为 0，inOrder 为 true 时，代码如下：

```
SpanNearQueryBuilder span=QueryBuilders.spanNearQuery();
span.clause(QueryBuilders.spanTermQuery("content","共青团中央") );
span.clause(QueryBuilders.spanTermQuery("content","实施意见") );
span.inOrder(true).slop(1);
client.prepareSearch("test").setQuery(span).execute().actionGet();
```