Log4j 手册

文档版本：1.2

编者：陈华

联系方式:clinker@163.com

发布日期：2009 年 2 月 17 日

目录

1. 简介

1.1. 概述

程序开发环境中的日志记录是由嵌入在程序中以输出一些对开发人员有用信息的语句所组成。例如，跟踪语句（trace），结构转储和常见的 System.out.println 或 printf 调试语句。log4j 提供分级方法在程序中嵌入日志记录语句。日志信息具有多种输出格式和多个输出级别。

使用一个专门的日志记录包，可以减轻对成千上万的 System.out.println 语句的维护成本，因为日志记录可以通过配置脚本在运行时得以控制。log4j 维护嵌入在程序代码中的日志记录语句。通过规范日志记录的处理过程，一些人认为应该鼓励更多的使用日志记录并且获得更高程度的效率。

1.2. 主要组件

Log4j 有三个主要组件： loggers、appenders 和 layouts。这三个组件协同工作，使开发人员能够根据消息类型和级别来记录消息，并且在程序运行期控制消息的输出格式位置。

- Logger：日志记录器
  Logger 负责处理日志记录的大部分操作。
- Appender：日志信息的输出目的地
  Appender 负责控制日志记录操作的输出。
- Layout：日志格式化器
  Layout 负责格式化 Appender 的输出。

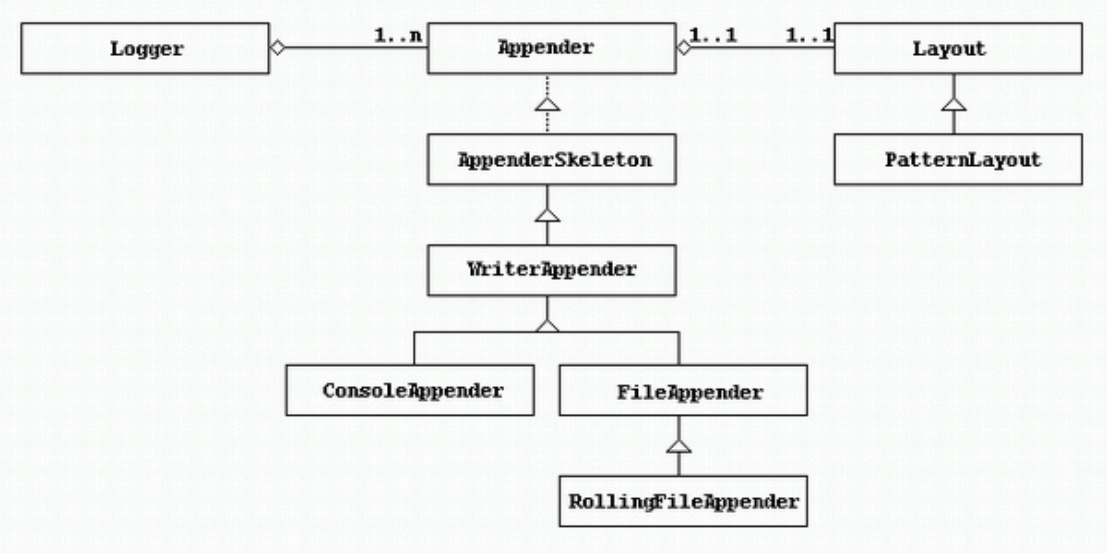

图 1-1 Log4j 的类图

2. Logger

日志记录器(Logger)是日志处理的核心组件。

2.1. 层次结构

Log4j 首要的相对于简单的使用 System.out.println()方法的优点是基于它的在禁止一些特定的信息输出的同时不妨碍其它信息的输出的能力。这个能力源自于日志命名空间，也就是说，所有日志声明的空间，它根据一些开发员选择的公式而分类。

Loggers 被指定为实体，Logger 的名字是大小写敏感的，它们遵循以下的命名规则：

- 命名继承：如果类别的名称（后面加一个点）是其子类别名称的前缀，则它就是另一个类别的祖辈。
- 如果一个类别(Logger)和它的子类别之间没有其它的继承关系，我们就称之为 parent 与 child 的关系。

例如，类别"com.foo"是类别"com.foo.Bar"的 parent。类似的，"java"是"java.util"的 parent，是"java.util.Vector"的父辈。

根(root) 类别位于 logger 继承结构的最上层。它有两种例外：

(1) 它一直存在。
(2) 它不能根据名称而获得。

调用类的静态方法 Logger.getRootLogger 可以得到它。其它所有的 Logger 可以通过静态方法 Logger.getLogger 而得到它们自己的实例。这个方法取希望的 Logger 名作为参数。Logger 的一些基本的方法示例如下：

```
package org.apache.Log4j;
public class Logger {
// Creation & retrieval methods:
public static Logger getRootLogger();
public static Logger getLogger(String name);
// printing methods:
public void debug(Object message);
public void info(Object message);
public void warn(Object message);
public void error(Object message);
// generic printing method:
public void log(Level l, Object message);
}
```

Loggers 可以被分配的级别。所有级别的集合包括：DEBUG、INFO、WARN、ERROR、FATAL。它们被定义于 org.apache.Log4j.Level 类。虽然我们不鼓励，但是你们可以通过继承 Level 类来定义你们自己的级别。我们随后将介绍一个比较好的方法。

如果一个 Logger 没有被分配一个级别，那么它将从一个被分配了级别的最接近它的 ancestor 哪里继承。正规的说：级别的继承：对于一个给定的 Logger C，它的继承的级别等于从 C 开始上溯到的第一个拥有非空级别的 Logger 的级别。

为了保证所有的 Logger 最终能够继承到一个级别，根 Logger 通常有一个已经定义了的级别。

以下四个表中的数据演示了根据以上规则得到的结果。

例 1：

类别名	分配的级别	继承的级别
root	Proot	Proot
X	none	Proot
X.Y	none	Proot
X.Y.Z	none	Proot

在例子 1 中，只有根 Logger 定义了一个级别，它的级别的值"Proot"被所有其它的 Loggers X、 X.Y 和 X.Y.Z 所继承。

例 2：

类别名	分配的级别	继承的级别
root	Proot	Proot
X	Px	Px
X.Y	Pxy	Pxy
X.Y.Z	Pxyz	Pxyz

在例子 2 中，所有的 Logger 都有一个被分配的级别值，所以它们不需要级别继承。

例 3：

类别名	分配的级别	继承的级别
root	Proot	Proot

X	Px	Px
X.Y	none	Px
X.Y.Z	Pxyz	Pxyz

在例子 3 中，根 Logger，以及 X 和 X.Y.Z 被分别分配了级别 Proot，Px 和 Pxyz。Logger X.Y 从它的父类别 X 继承了级别值 Px。

例 4：

类别名	分配的级别	继承的级别
root	Proot	Proot
X	Px	Px
X.Y	none	Px
X.Y.Z	none	Px

在例子 4 中，根 Logger 和 X 被分别分配了级别"Proot"和"Px"，Logger X.Y 和 X.Y.Z 从被分配了级别的最接近它们的祖先 X 那里得到继承。

2.2. 输出级别

我们需要通过调用 Logger 的输出的实例方法之一来实现日志请求。这些输出的方法是 debug、info、warn、error、fatal 和 log。

通过定义输出方法来区分日志的请求的级别。例如，如果 c 是一个 Logger 的实例，那么声明 c.info("…")就是一个 INFO 级别的日志请求。

如果一个日志的请求的级别高于或等于日志的级别那么它就能被启用。反之，将被禁用。一个没有被安排级别的 Logger 将从它的父辈中得到继承。这个规则总结如下：

基本的选择规则：假如在一个级别为 q 的 Logger 中发生一个级别为 p 的日志请求，如果 p>=q，那么请求将被启用。

这是 Log4j 的核心原则。它假设级别是有序的。对于标准级别，我们定义 DEBUG<INFO <WARN<ERROR<FATAL。

以下是关于这条规则的一个例子。

```
// 取得名为 com.foo 的 logger 实例
Logger logger = Logger.getLogger("com.foo");
```

```
// 现在设置它的级别。
// 一般情况下你不需要编程式的设置 logger 的级别，通常都是通过配置文件来设置的。
cat.setLevel(Level.INFO);
Logger barlogger = Logger.getLogger("com.foo.Bar");

// 这个请求可用，因为 WARN >= INFO
logger.warn("Low fuel level.");

// 这个请求不可用，因为 DEBUG < INFO.
logger.debug("Starting search for nearest gas station.");

// 名为 com.foo.Bar 的 logger 继承了 com.foo 的级别，
// 因此，下面的请求可用，因为 INFO >= INFO
barlogger.info("Located nearest gas station.");

//这个请求不可用，因为 DEBUG < INFO
barlogger.debug("Exiting gas station search");
```

调用 getLogger 方法将返回一个同名的 Logger 对象的实例。 例如，

```
Categoty x = Logger.getLogger("wombat");
Categoty y = Logger.getLogger("wombat");
```

x 和 y 参照的是同一个 Logger 对象。

这样我们就可以先定义一个 Logger，然后在代码的其它地方不需传参就可以重新得到我们已经定义了的 Logger 的实例。

同基本的生物学理论——父先于子相反，Log4j 的 loggers 可以以任何顺序创造和配置。特别是，一个后实例化的"parent"logger 能够找到并且连接它的子 logger。

配置 Log4j 的环境通常在一个应用程序被初始化的时候进行，最好的方法是通过读一个

配置文件。这个方法我们将简短介绍。

Log4j 使得通过软件组件命名 logger 很容易。我们可以通过 Logger 的静态的初始化方法在每一个类里定义一个 logger，令 logger 的名字等于类名的全局名，而实现 logger 的命名。这是一个实效的简单的定义一个 logger 的方法。因为日志输出带有产生日志的类的名 字，这个命名策略使得我们更容易定位到一个日志信息的来源。虽然普通，但却是命名 logger 的常用策略之一。Log4j 没有限制定义 logger 的可 能。开发员可以自由的按照它们的意愿定义 logger 的名称。

然而，以类的所在位置来命名 Logger 好象是目前已知的最好方法。

3. Appenders

3.1. 概念说明

有选择的启用或者禁用日志请求仅仅是 Log4j 的一部分功能。Log4j 允许日志请求被输出到多个输出源。用 Log4j 的话说，一个输出源被称做一个 Appender。Appender 包括 console（控制台）、files（文件）、GUI components（图形的组件）、remote socket servers（socket 服务）、JMS（java 信息服务）、NT Event Loggers（NT 的事件日志）和 remote UNIX Syslog daemons（远程 UNIX 的后台日志服务）。它也可以做到异步记录。

一个 logger 可以设置超过一个的 appender。

用 addAppender 方法添加一个 appender 到一个给定的 logger。**对于一个给定的 logger，它每个生效的日志请求都被转发到 logger 所有的 appender 上和该 logger 的父辈 logger 的 appender 上。**换句话说，appende 自动从它的父辈获得继承。举例来说，如果一个根 logger 拥有一个 console appender，那么所有生效的日志请求至少会被输出到 console 上。如果一个名为 C 的 logger 有一个 file 类型的 appender，那么它就会对它自己以及所有它的子 logger 生效。我们也可以通过设置 appender 的 additivity flag 为 false，来重载 appender 的默认行为，以便继承的属性不在生效。

3.2. Appender 的配置

配置日志信息输出目的地 Appender，其语法为：

```
log4j.appender.appenderName = fully.qualified.name.of.appender.class
log4j.appender.appenderName.option1 = value1
…
log4j.appender.appenderName.option = valueN
```

其中，Log4j 提供的 appender 有以下几种：

org.apache.log4j.ConsoleAppender（控制台），

org.apache.log4j.FileAppender（文件），

org.apache.log4j.DailyRollingFileAppender（每天产生一个日志文件），

org.apache.log4j.RollingFileAppender（文件大小到达指定尺寸的时候产生一个新的文件），

org.apache.log4j.WriterAppender（将日志信息以流格式发送到任意指定的地方）

3.3. Appender 的添加性

调节输出源（appender）添加性的规则如下。

输出源的可添加性（Appender Additivity）：一个名为 C 的 logger 的日志定义的输出将延续到它自身以及它的 ancestor logger 的 appenders。这就是术语"appender additivity"的含义。

然而，logger C 的一个祖先 logger P，它的附加标志被设为 false，那么 C 的输出将被定位到所有 C 的 appender，以及从它开始上溯到 P 的所有 ancestor logger 的 appender。

Loggers 的附加标记（additivity flag）默认为 true。

下表是一个例子。

名称	添加的 Appenders	可添加性标志	输出目标	注释
root	A1	不可用	A1	根 logger 是匿名的，但可以用 Logger.getRootLogger() 方法来访问。根 logger 没有默认的 appender。
x	A-x1, A-x2	true	A1，A-x1, A-x2	x 和 root 的 appenders
x.y	none	true	A1，A-x1, A-x2	x 和 root 的 appenders
x.y.z	A-xyz1	true	A1，A-x1, A-x2, A-xyz1	x.y.z 和 x 和 root 的 appenders
security	A-sec	false	A-sec	由于可添加性为 false，所以没有继承的 appender。

security.access	none	true	A-sec	只有 security 的 appender，因为 security 的可添加性标志为 false

4. Layouts

4.1. 概念说明

经常，用户希望不但自定义输出源，而且定义输出格式。这是通过在 appender 上附加一个 layout 来完成的。layout 负责根据用户的希望来格式化日志请求。而 appender 是负责发送格式化的输出到它的目的地。

PatternLayout，作为 Log4j 标准版中的一部分，让用户以类似 C 语言的 printf 方法的格式来指定日志的输出格式。

例如，转化模式为"%r [%t] %-5p %c - %m%n" 的 PatternLayout 将输出类似如下的信息：

176 [main] INFO org.foo.Bar - Located nearest gas station.

第一个栏位是自从程序开始后消逝的毫秒数。

第二个栏位是做出日志的线程。

第三个栏位是 log 的级别。

第四个栏位是日志请求相关的 logger 的名字。而"-"后的文字是信息的表述。

Log4j 将根据用户定义的公式来修饰日志信息的内容。例如，如果你经常需要记录 Oranges，一个在你当前的项目被用到的对象类型，那么你可以注册一个 OrangeRenderer，它将在一个 orange 需要被记录时被调用。

对象渲染类似类的结构继承。例如，假设 oranges 是 fruits，如果你注册了一个 FruitRenderer，所有的水果包括 oranges 将被 FruitRenderer 所渲染。除非你注册了一个 orange。

对象渲染必须实现 ObjectRenderer 接口。

4.2. Layout 的配置

配置日志信息的格式（布局），其语法为：

log4j.appender.appenderName.layout = fully.qualified.name.of.layout.class

log4j.appender.appenderName.layout.option1 = value1

…

log4j.appender.appenderName.layout.option = valueN

其中，Log4j 提供的 layout 有以下几种：

org.apache.log4j.HTMLLayout（以 HTML 表格形式布局），

org.apache.log4j.PatternLayout（可以灵活地指定布局模式），

org.apache.log4j.SimpleLayout（包含日志信息的级别和信息字符串），

org.apache.log4j.TTCCLayout（包含日志产生的时间、线程、类别等等信息）

Log4J 采用类似 C 语言中的 printf 函数的打印格式格式化日志信息，打印参数如下：%m 输出代码中指定的消息

%p 输出优先级，即 DEBUG，INFO，WARN，ERROR，FATAL

%r 输出自应用启动到输出该 log 信息耗费的毫秒数

%c 输出所属的类目，通常就是所在类的全名

%t 输出产生该日志事件的线程名

%n 输出一个回车换行符，Windows 平台为"\r\n"，Unix 平台为"\n"

%d 输出日志时间点的日期或时间，默认格式为 ISO8601，也可以在其后指定格式，比如：%d{yyy MMM dd HH:mm:ss,SSS}，输出类似：2002 年 10 月 18 日 22：10：28，921

%l 输出日志事件的发生位置，包括类目名、发生的线程，以及在代码中的行数。举例：Testlog4.main(TestLog4.java:10)

5. 配置

Log4j 在程序中有充分的可配置性。然而，用配置文件配置 Log4j 具有更大的弹性。目前，它的配置文件支持 xml 和 java properties（key=value）文件两种格式。

让我们以一个例子来演示它是如何做的。假定有一个用了 Log4j 的程序 MyApp。

```
import com.foo.Bar;

// Import Log4j classes.

import org.apache.Log4j.Logger;

import org.apache.Log4j.BasicConfigurator;


public class MyApp {

// Define a static logger variable so that it references the

// Logger instance named "MyApp".

static Logger logger = Logger.getLogger(MyApp.class);


public static void main(String[] args) {

// Set up a simple configuration that logs on the console.

BasicConfigurator.configure();


logger.info("Entering application.");

Bar bar = new Bar();

bar.doIt();

logger.info("Exiting application.");

}

}
```

MyApp 以引入 Log4j 的相关类开始，接着它定义了一个静态 logger 变量，并给予值为"MyApp"类的全路径名称。

MyApp 用了定义在包 com.foo 中的类 Bar。

```
package com.foo;
```

```
import org.apache.Log4j.Logger;
public class Bar {
static Logger logger = Logger.getLogger(Bar.class);
public void doIt() {
    logger.debug("Did it again!");
}
}
```

调用 BasicConfigurator.configure()方法创建了一个相当简单的 Log4j 的设置。它加入一个 ConsoleAppender 到根 logger。输出将被采用了"%-4r [%t] %-5p %c %x - %m%n"模式的 PatternLayout 所格式化。

注意，根 logger 默认被分配了 Level.DEBUG 的级别。

MyApp 的输出为：

0 [main] INFO MyApp - Entering application.

36 [main] DEBUG com.foo.Bar - Did it again!

51 [main] INFO MyApp - Exiting application.

随后的图形描述了在调用 BasicConfigurator.configure()方法后 MyApp 的对象图。

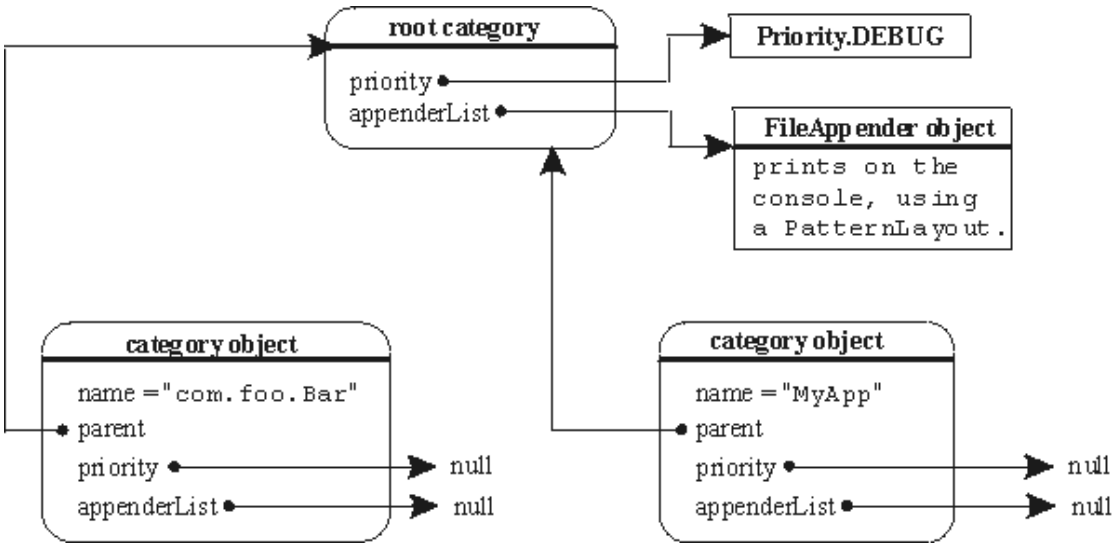

图 5-1 MyApp 对象图

需要提醒的是，Log4j 的子 logger 只连接到已经存在的它们的父代。特别的是，名为 com.foo.bar 的 logger 是直接连接到根 logger，而不是围绕着没用的 com 或 com.foo logger。这显著的提高了程序性能并且减少的内存占用。

MyApp 类配置 Log4j 是通过调用 BasicConfigurator.configure 方法。其它的类仅仅需要引入 org.apache.Log4j.Logger 类，找到它们希望用的 logger，并且用它就行。

以前的例子通常输出同样的日志信息。幸运的是，修改 MyApp 是容易的，以便日志输出可以在运行时刻被控制。这里是一个小小修改的版本。

```
import com.foo.Bar;
import org.apache.Log4j.Logger;
import org.apache.Log4j.PropertyConfigurator;
public class MyApp {
static Logger logger = Logger.getLogger(MyApp.class.getName());
public static void main(String[] args) {
    // BasicConfigurator replaced with PropertyConfigurator.
    PropertyConfigurator.configure(args[0]);
    logger.info("Entering application.");
    Bar bar = new Bar();
    bar.doIt();
    logger.info("Exiting application.");
}
}
```

修改后的 MyApp 通知程序调用 PropertyConfigurator()方法解析一个配置文件，并且根据这个配置文件来设置日志。

这里是一个配置文件的例子，它将产生同以前 BasicConfigurator 基本例子一样的输出结果。 # Set root logger level to DEBUG and its only appender to A1.

```
Log4j.rootLogger=DEBUG, A1
# A1 is set to be a ConsoleAppender.
Log4j.appender.A1=org.apache.Log4j.ConsoleAppender
# A1 uses PatternLayout.
Log4j.appender.A1.layout=org.apache.Log4j.PatternLayout
Log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
```

假设我们不在对 com.foo 包的任何类的输出感兴趣的话，随后的配置文件向我们展示了实现这个目的的方法之一。

Log4j.rootLogger=DEBUG, A1

Log4j.appender.A1=org.apache.Log4j.ConsoleAppender

Log4j.appender.A1.layout=org.apache.Log4j.PatternLayout

 # Print the date in ISO 8601 format

Log4j.appender.A1.layout.ConversionPattern=**%d** [%t] %-5p %c - %m%n

Print only messages of level WARN or above in the package com.foo.

Log4j.logger.com.foo=WARN

以这个配置文件配置好的 MyApp 将输出如下：

2000-09-07 14:07:41,508 [main] INFO MyApp - Entering application.

2000-09-07 14:07:41,529 [main] INFO MyApp - Exiting application.

当 logger com.foo.bar 没有被分配一个级别，它将从 com.foo 继承，在配置文件中它被设置了 WARN 的级别。在 Bar.doIt 方法中定义的 log 为 DEBUG 级别，低于 WARN，因此 doIt() 方法的日志请求被禁用。

这里是另外一个配置文件，它使用了多个 appenders.

Log4j.rootLogger=debug, **stdout, R**

Log4j.appender.stdout=org.apache.Log4j.ConsoleAppender

Log4j.appender.stdout.layout=org.apache.Log4j.PatternLayout

Pattern to output the caller's file name and line number.

Log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (**%F:%L**) - %m%n

Log4j.appender.**R**=org.apache.Log4j.RollingFileAppender

Log4j.appender.R.File=example.log

Log4j.appender.R.MaxFileSize=**100KB**

```
# Keep one backup file
Log4j.appender.R.MaxBackupIndex=1

Log4j.appender.R.layout=org.apache.Log4j.PatternLayout
Log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%n
```

以这个配置文件调用加强了的 MyApp 类将输出如下信息。

```
INFO [main] (MyApp2.java:12) - Entering application.
DEBUG [main] (Bar.java:8) - Doing it again!
INFO [main] (MyApp2.java:15) - Exiting application.
```

另外,因为根 logger 有被分配第二个 appender,所以输出也将被定向到 example.log 文件。这个文件大小达到 100kb 时将自动备份。备份时老版本的 example.log 文件自动被移到文件 example.log.1 中。

注意我们不需要重新编译代码就可以获得这些不同的日志行为。我们一样可以容易的使日志输出到 UNIX Syslog daemon，重定向所有的 com.foo 到 NT Event logger，或者转发日志到一个远程的 Log4j 服务器,它根据本地 server 的策略来进行日志输出。例如转发日志事件到第二个 Log4j 服务器。

6. 默认的初始化过程

Log4j 类库不对它的环境做任何假设。特别是没有默认的 Log4j appender。在一些特别的有着良好定义的环境下，logger 的静态 inializer 将尝试自动的配置 Log4j。java 语言的特性保证类的静态 initializer 当且仅当装载类到内存之时只会被调用一次。要记住的重要一点是，不同的类装载器可能装载同一个类的完全不同的拷贝。这些同样类的拷贝被虚拟机认为是完全不相干的。

默认的 initialization 是非常有用的，特别是在一些应用程序所依靠的运行环境被准确的定位的情况下。例如，同一样的应用程序可以被用做一个标准的应用程序，或一个 applet，或一个在 web-server 控制下的 servlet。

准确的默认的 initialization 原理被定义如下：

(1) 设置系统属性 Log4j.defaultInitOverride 为"false"以外的其它值，那么 Log4j 将跳过默认的 initialization 过程。
(2) 设置资源变量字符串给系统属性 Log4j.configuration。定义默认 initialization 文件的最好的方法是通过系统属性 Log4j.configuration。万一系统属性 Log4j.configuration 没有被定义，那么设置字符串变量 resource 给它的默认值 Log4j.properties。
(3) 尝试转换 resource 变量为一个 URL。
(4) 如果变量 resource 的值不能被转换为一个 URL，例如由于 MalformedURLException 违例，那么通过调用 org.apache.Log4j.helpers.Loader.getResource(resource, Logger.class) 方法从 classpath 中搜索 resource，它将返回一个 URL，并通知"Log4j.properties"的值是一个错误的 URL。看 See Loader.getResource(java.lang.String) 查看搜索位置的列表。
(5) 如果没有 URL 被发现，那么放弃默认的 initialization。否则用 URL 配置 Log4j。PropertyConfigurator 将用来解析 URL，配置 Log4j，除非 URL 以".xml"为结尾。在这种情况下的话 DOMConfigurator 将被调用。你可以有机会定义一个自定义的 configurator。系统属性 **Log4j.configuratorClass** 的值取自你的自定义的类名的全路径。自定义的 configurator 必须实现 configurator 接口。

7. 配置范例

7.1. Tomcat

Tomcat 下的初始化默认的 Log4j initialization 典型的应用是在 web-server 环境下。在 tomcat 下，你应该将配置文件 Log4j.properties 放在你的 web 应用程序的 WEB-INF/classes 目录下。 Log4j 将发现属性文件，并且以此初始化。这是使它工作的最容易的方法。

8. Nested Diagnostic Contexts

在现实世界中的系统经常不得不同时处理多个客户端请求。在这样的一个典型的多线程的系统中，不同的线程将处理不同的客户端。Logging 特别能够适应这种复杂的分布式的应用程序的调试和跟踪。一个常见的区分每个客户端所输出的 Logging 的方法是为每个客户端实例化一个新的独立的 Logger。这导致 Logger 的大量产生，管理的成本也超过了 logging 本身。

唯一标识每个 log 请求是一个轻量级的技术。Neil Harrison 在名为“Patterns for Logging Diagnostic Messages”的书中描述了这个方法 in Pattern Languages of Program Design 3, edited by R. Martin, D. Riehle, and F. Buschmann (Addison-Wesley, 1997).

为了唯一标识每个请求，用户把上下文信息推入 NDC(Nested Diagnostic Context)中。

NDC 类示例如下：

```
public class NDC {
// Used when printing the diagnostic
public static String get();

// Remove the top of the context from the NDC.
public static String pop();

// Add diagnostic context for the current thread.
public static void push(String message);

// Remove the diagnostic context for this thread.
public static void remove(); s
}
```

NDC 如同一个堆栈样管理每个线程。注意所有 the org.apache.log4j.NDC 类的方法都是静态的。假设 NDC 输出被开启，每次一个 log 请求被生成时，适当的 log4j 组件为将要输出 log 的线程包含完整的 NDC 堆栈。这是在没有用户的干预的情况下做到的，用户只负责在 NDC 中定位正确的信息，通过在代码中正确位置插入很少的 push 和 pop 方法就行了。相反的，在代码中 per-client 实现方法有着很大变化。

为了演示这点，让我们以一个发送内容到匿名客户端的 servlet 为例。这个 servlet 可以

在开始执行每个其他代码前的初始化时建立 NDC。上下文信息可以是客户主机的名字和其他的请求中固有的信息。

典型的信息包括 cookies。因此，即使 servlet 同时为多个客户同时提供服务，log 被同样的代码初始化，例如属于同一个 logger，依然可以被区别，因为每个客户请求将有不同的 NDC 堆栈。与之相比，Contrast this with the complexity of passing a freshly instantiated logger to all code exercised during the client's request。

不过，一些诡异的程序，例如虚拟主机的 web server 记录日志，不是一般的依靠虚拟主机的上下文，还要依靠软件的组件发出请求。近来 log4j 的发布版本支持多层的树形结构。这个增强允许每个虚拟主机可以处理在树型结构中属于它自己的 logger。

9. 优化

一个经常引用的依靠于 logging 的参数是可以计算的花费。这是一个合理的概念，一个适度的应用程序可能产生成千上万个日志请求。许多努力花在测量和调试 logging 的优化上。Log4j 要求快速和弹性：速度最重要，弹性是其次。

用户应该注意随后的优化建议。

9.1. 日志为禁用时，日志的优化

当日志被彻底的关闭，一个日志请求的花费等于一个方法的调用加上整数的比较时间。

在 233mhz 的 Pentium II 机器上这个花费通常在 5-50 纳秒之间。

然而，方法调用包括参数构建的隐藏花费。

例如，对于 logger cat，

```
logger.debug("Entry number: " + i + " is " + String.valueOf(entry));
```

引起了构建信息参数的花费，例如，转化整数 i 和 entry 到一个 string，并且连接中间字符串，不管信息是否被输出。这个参数的构建花费可能是很高，它主要决定于被调用的参数的大小。

避免参数构建的花费应如下，

```
if(logger.isDebugEnabled() {
    logger.debug("Entry number: " + i + " is " + String.valueOf(entry[i]));
}
```

如果 logger 的 debug 被关闭这将不会招致参数构建的花费。另一方面，如果 logger 是 debug 的话，它将产生两次判断 logger 是否能用的花费。一次是在 debugenabled，一次是 debug。这是无关紧要的，因为判断日志是否可用只占日志实际花费时间的约 1%。

在 Log4j 里，日志请求在 Logger 类的实例里。Logger 是一个类，而不是一个接口。这大量的减少了在方法调用上的弹性化的花费。

当然用户采用预处理或编译时间技术去编译出所有的日志声明。这将导致完美的执行成效。然而因为二进制应用程序不包括任何的日志声明的结果，日志不可能对那个二进制程序

开启。以我的观点，以这种较大的代价来换取较小的性能优化是不值得的。

9.2. 当日志状态为启用时，日志的优化

这是本质上的优化 logger 的层次。当日志状态为开，Log4j 依然需要比较请求的级别与 logger 的级别。然而，logger 可能没有被安排一个级别；它们将从它们的 father 继承。这样，在继承之前，logger 可能需要搜索它的祖先。

这里有一个认真的努力使层次的搜索尽可能的快。例如，子 logger 仅仅连接到它的存在的 father logger。

在先前展示的 BasicConfigurator 例子中，名为 com.foo.bar 的 logger 是连接到根 logger，因此绕过了不存在的 logger com 和 com.foo。这将显著的改善执行的速度，特别是解析 logger 的层结构时。

典型的层次结构的解析的花费是 logger 彻底关闭时的三倍。

9.3. 日志信息的输出时，日志的优化

这是主要花费在日志输出的格式化和发送它到它的输出源上。这里我们再一次的付出努力以使格式化执行的尽可能快。同 appender 一样。实际上典型的花费大约是 100-300 毫秒。

详情看 org.apache.log4.performance.Logging。

虽然 Log4j 有许多特点，但是它的第一个设计目标还是速度。一些 Log4j 的组件已经被重写过很多次以改善性能。不过，投稿者经常提出了新的优化。你应该满意的知道，以 SimpleLayout 的配置执行测试已经展示了 Log4j 的输出同 System.out.println 一样快。

10.总结

Log4j 是一个用 java 写成的流行的日志包。一个它与众不同的特点是在 logger 中的继承的概念。用 logger 的继承可以以任意的间隔控制日志的状态输出。这个减少了体积和最小化日志的代价。

易管理性是 Log4j API 的优点之一。只要日志定义被加入到代码中，它们就可以用配置文件来控制。它们可以有选择的被禁用，并且按照用户选择好的格式发送到不同的多个输出源上。 Log4j 的包经过良好的设计成，因此不需花费沉重的执行代价就可以将它们保留在产品中。

11.附录

11.1. 参考文档

- Log4j short manual
- Log4j 学习笔记 by heavyz

说明：本文主体为 log4j short manual，翻译者不可考。其余内容取自互联网，作者不可考。

11.2. 比较全面的配置文件

下面的 Log4J 配置文件实现了输出到控制台、文件、回滚文件、发送日志邮件、输出到数据库日志表、自定义标签等全套功能。

```
log4j.rootLogger=DEBUG,CONSOLE,A1,im

log4j.addivity.org.apache=true


# 应用于控制台

log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender

log4j.appender.Threshold=DEBUG

log4j.appender.CONSOLE.Target=System.out

log4j.appender.CONSOLE.Encoding=GBK

log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout

log4j.appender.CONSOLE.layout.ConversionPattern=[framework] %d - %c -%-4r [%t] %-5p %c %x - %m%n

#log4j.appender.CONSOLE.layout.ConversionPattern=[start]%d{DATE}[DATE]%n%p[PRIORITY]%n%x[NDC]%n%t[THREAD] n%c[CATEGORY]%n%m[MESSAGE]%n%n


#应用于文件

log4j.appender.FILE=org.apache.log4j.FileAppender
```

log4j.appender.FILE.File=file.log

log4j.appender.FILE.Append=false

log4j.appender.FILE.Encoding=GBK

log4j.appender.FILE.layout=org.apache.log4j.PatternLayout

log4j.appender.FILE.layout.ConversionPattern=[framework] %d - %c -%-4r [%t] %-5p %c %x - %m%n

Use this layout for LogFactor 5 analysis

应用于文件回滚

log4j.appender.ROLLING_FILE=org.apache.log4j.RollingFileAppender

log4j.appender.ROLLING_FILE.Threshold=ERROR

log4j.appender.ROLLING_FILE.File=rolling.log

log4j.appender.ROLLING_FILE.Append=true

log4j.appender.CONSOLE_FILE.Encoding=GBK

log4j.appender.ROLLING_FILE.MaxFileSize=10KB

log4j.appender.ROLLING_FILE.MaxBackupIndex=1

log4j.appender.ROLLING_FILE.layout=org.apache.log4j.PatternLayout

log4j.appender.ROLLING_FILE.layout.ConversionPattern=[framework] %d - %c -%-4r [%t] %-5p %c %x - %m%n

#应用于 socket

log4j.appender.SOCKET=org.apache.log4j.RollingFileAppender

log4j.appender.SOCKET.RemoteHost=localhost

log4j.appender.SOCKET.Port=5001

log4j.appender.SOCKET.LocationInfo=true

Set up for Log Facter 5

log4j.appender.SOCKET.layout=org.apache.log4j.PatternLayout

log4j.appender.SOCET.layout.ConversionPattern=[start]%d{DATE}[DATE]%n%p[PRIORI

TY]%n%x[NDC]%n%t[THREAD]%n%c[CATEGORY]%n%m[MESSAGE]%n%n

```
# Log Factor 5 Appender
log4j.appender.LF5_APPENDER=org.apache.log4j.lf5.LF5Appender
log4j.appender.LF5_APPENDER.MaxNumberOfRecords=2000

# 发送日志给邮件
log4j.appender.MAIL=org.apache.log4j.net.SMTPAppender
log4j.appender.MAIL.Threshold=FATAL
log4j.appender.MAIL.BufferSize=10
log4j.appender.MAIL.From=alert@ log4jmanul.org
log4j.appender.MAIL.SMTPHost=smtp.log4jmanul.org
log4j.appender.MAIL.SMTPUsername=username
log4j.appender.MAIL.SMTPPassword=password
log4j.appender.MAIL.Subject=Log4J Message
log4j.appender.MAIL.To=msg@ log4jmanul.org
log4j.appender.MAIL.Bcc=admin@ log4jmanul.org
log4j.appender.MAIL.layout=org.apache.log4j.PatternLayout
log4j.appender.MAIL.layout.ConversionPattern=[framework] %d - %c -%-4r
[%t] %-5p %c %x - %m%n

# 用于数据库
log4j.appender.DATABASE=org.apache.log4j.jdbc.JDBCAppender
log4j.appender.DATABASE.URL=jdbc:mysql://localhost:3306/test
log4j.appender.DATABASE.driver=com.mysql.jdbc.Driver
log4j.appender.DATABASE.user=root
log4j.appender.DATABASE.password=
```

log4j.appender.DATABASE.sql=INSERT INTO LOG4J (Message) VALUES ('[framework] %d - %c -%-4r [%t] %-5p %c %x - %m%n')

log4j.appender.DATABASE.layout=org.apache.log4j.PatternLayout

log4j.appender.DATABASE.layout.ConversionPattern=[framework] %d - %c -%-4r [%t] %-5p %c %x - %m%n

每天新建日志

log4j.appender.A1=org.apache.log4j.DailyRollingFileAppender

log4j.appender.A1.File=log

log4j.appender.A1.Encoding=GBK

log4j.appender.A1.DatePattern='.'yyyy-MM-dd

log4j.appender.A1.layout=org.apache.log4j.PatternLayout

log4j.appender.A1.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L : %m%n

#自定义 Appender

log4j.appender.im = net.cybercorlin.util.logger.appender.IMAppender

log4j.appender.im.host = mail.cybercorlin.net

log4j.appender.im.username = username

log4j.appender.im.password = password

log4j.appender.im.recipient = abc@log4jmanul.org

log4j.appender.im.layout=org.apache.log4j.PatternLayout

log4j.appender.im.layout.ConversionPattern =[framework] %d - %c -%-4r [%t] %-5p %c %x - %m%n

11.3. 日志乱码的解决

缺省配置下，Log4j 的输出内容使用系统默认编码。在中文 Windows 下，一般不会出现

乱码问题。而在 Linux/Unix，可能会由于没有设置匹配的系统环境而导致 log4j 在输出中文时产生乱码，解决方法为：

增加 Encoding 选项。

例如：

log4j.appender.CONSOLE.Encoding=UTF-8

appende 为文件时，处理方法相同。

详细内容参见：org.apache.log4j.WriterAppender 中对 encoding 的处理。