目录

第1章 Java 日志系统

1 日志发展历程

日志的重要性是随着系统的膨胀而显现的，在一个庞大的系统中查错没有各种日志信息是寸步难行的。所以在系统加入日志是必须的。

最原始的日志方式，就是在程序的适当地方添加 System.out.println()方法，但是带来的问题是，系统稳定后，日志太大，如果要减少日志量，就需要重新修改程序，虽然只是注释掉 System.out.println()方法。但是万一系统再次出错，又要改。

当然，也有一种解决办法可以实现处理这个问题，在 Java 中，可以这么写：

```
public class LogUtil {
    private static boolean logSwitch=true;   //日志开关,默认为开
    public static void log(String smg){
        if(logSwitch){
            System.out.println(smg);
        }
    }
}

public class ManualLogApp {
    public static void main(String[] args){
        new ManualLogApp().test();
    }

    public void test(){
        LogUtil.log("我在手动打日志,呵呵!");
    }
}
```

通过这两个类，就实现一个简单的日志工具。当你不想打印日志的时候，可以在 LogUtil 中改 logSwitch 的值为 flase，那么日志打印功能就关闭了。如果你愿意，你可以将这个开关放到一个配置文件中去修改。还可以让日志打印到文件，但是当你修改实现这个功能的时候，这个日志工具会变得相当的复杂。

为了解决这个问题，程序猿们针对不同的语言平台为做了一系列日志工具包，

可应用于 java、.net、php、c++，这些日志包都是免费的，使用非常方便，可以极大提高编程效率。并且，为了让众多的日志工具有一个相同操作方式，还提供了一些通用的日志工具包。

最早得到广泛使用的是 log4j，许多应用程序的日志部分都交给了 log4j，不过作为组件开发者，他们希望自己的组件不要紧紧依赖某一个工具，毕竟在同一个时候还有很多其他很多日志工具，假如一个应用程序用到了两个组件，恰好两个组件使用不同的日志工具，那么应用程序就会有两份日志输出了。

为了解决这个问题，Apache Commons Logging （之前叫 Jakarta Commons Logging，JCL）粉墨登场，JCL 只提供 log 接口，具体的实现则在运行时动态寻找。这样一来组件开发者只需要针对 JCL 接口开发，而调用组件的应用程序则可以在运行时搭配自己喜好的日志实践工具。

所以即使到现在你仍会看到很多程序应用 JCL＋log4j 这种搭配，不过当程序规模越来越庞大时，JCL 的动态绑定并不是总能成功，具体原因大家可以 Google 一下，这里就不再赘述了。解决方法之一就是在程序部署时静态绑定指定的日志工具，这就是 SLF4J 产生的原因。

Tips：

关于程序绑定的概念：

绑定指的是一个方法的调用与方法所在的类(方法主体)关联起来。对 java 来说，绑定分为静态绑定和动态绑定；或者叫做前期绑定和后期绑定.

静态绑定：

在程序执行前方法已经被绑定(也就是说在编译过程中就已经知道这个方法到底是哪个类中的方法)，此时由编译器或其它连接程序实现。例如：C。针对 java 简单的可以理解为程序编译期的绑定；这里特别说明一点，java 当中的方法只有 final，static，private 和构造方法是前期绑定

动态绑定：

后期绑定：在运行时根据具体对象的类型进行绑定。

跟 JCL 一样，SLF4J 也是只提供 log 接口，具体的实现是在打包应用程序时所放入的绑定器(名字为 slf4j-XXX-version.jar)来决定，XXX 可以是 log4j12, jdk14, jcl, nop 等，他们实现了跟具体日志工具(比如 log4j)的绑定及代理工作。举个例子：如果一个程序希望用 log4j 日志工具，那么程序只需针对 slf4j-api 接口编程，然后在打包时再放入 slf4j-log4j12-version.jar 和 log4j.jar 就可以了。

现在还有一个问题，假如你正在开发应用程序所调用的组件当中已经使用了 JCL 的，还有一些组件可能直接调用了 java.util.logging，这时你需要一个桥接器（名字为 XXX-over-slf4j.jar）把他们的日志输出重定向到 SLF4J，所谓桥接器就是一个假的日志实现工具，比如当你把 jcl-over-slf4j.jar 放到 CLASS_PATH 时，即使某个组件原本是通过 JCL 输出日志的，现在却会被 jcl-over-slf4j “骗到”SLF4J 里，然后 SLF4J 又会根据绑定器把日志交给具体的日志实现工具。

2 常见的几种日志

2.1 Log4j

Apache 的一个开放源代码项目，通过使用 Log4j，我们可以控制日志信息输送的目的地是控制台、文件、GUI 组件、甚至是套接口服务器、NT 的事件记录器、UNIX Syslog 守护进程等；用户也可以控制每一条日志的输出格式；通过定义每一条日志信息的级别，用户能够更加细致地控制日志的生成过程。这些可以通过一个配置文件来灵活地进行配置，而不需要修改程序代码，是一种经典的日志解决方案。内部把日志系统抽象封装成 Logger 、appender 、pattern 等实现。我们可以通过配置文件轻松的实现日志系统的管理和多样化配置。

2.2 LOGBack

Logback 是由 log4j 创始人设计的又一个开源日记组件。logback 当前分成三个模块：logback-core,logback- classic 和 logback-access。

- logback-core 是其它两个模块的基础模块。
- logback-classic 是 log4j 的一个 改良版本。此外 logback-classic 完整实现 SLF4J API 使你可以很方便地更换成其它日记系统如 log4j 或 JDK14 Logging。
- logback-access 访问模块与 Servlet 容器集成提供通过 Http 来访问日志的功能。

LOGBack 作为一个通用可靠、快速灵活的日志框架，将作为 Log4j 的替代和 SLF4J 组成新的日志系统的完整实现。官网上称具有极佳的性能，在关键路径上执行速度是 log4j 的 10 倍，且内存消耗更少。

2.3 SLF4J

简单日记门面(Facade)SLF4J 是为各种 loging APIs 提供一个简单统一的接口，从而使得最终用户能够在部署的时候配置自己希望的 loging APIs 实现。Logging API 实现既可以选择直接实现 SLF4J 接口的 loging APIs 如： NLOG4J、SimpleLogger。也可以通过 SLF4J 提供的 API 实现来开发相应的适配器如 Log4jLoggerAdapter、JDK14LoggerAdapter。

slf4j 全称为 Simple Logging Facade for JAVA，java 简单日志门面。类似于 Apache Common-Logging，是对不同日志框架提供的一个门面封装，可以在部署的时候不修改任何配置即可接入一种日志实现方案。但是，他在编译时静态绑定真正的 Log 库。使用 SLF4J 时，如果你需要使用某一种日志实现，那么你必须选择正确的 SLF4J 的 jar 包的集合（各种桥接包）。

使用 slf4j 的常见代码：

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class A {
```

```
    private static Log logger = LogFactory.getLog(this.getClass());
}
```

slf4j 静态绑定原理：

SLF4J 会在编译时会绑定 import org.slf4j.impl.StaticLoggerBinder; 该类里面实现对具体日志方案的绑定接入。任何一种基于 slf4j 的实现都要有一个这个类。如：org.slf4j.slf4j-log4j12-1.5.6: 提供对 log4j 的一种适配实现。注意：如果有任意两个实现 slf4j 的包同时出现，那么就可能出现问题。

2.4 Common-Logging

目前广泛使用的 Java 日志门面库。通过动态查找的机制，在程序运行时自动找出真正使用的日志库。但由于它使用了 ClassLoader 寻找和载入底层的日志库，导致了象 OSGI[1]这样的框架无法正常工作，由于其不同的插件使用自己的 ClassLoader。尽管 OSGI 的这种机制保证了插件之间的互相独立，但这也使得 Apache Common-Logging 无法工作。

还有一点，Common-logging 也是 apache 提供的一个通用的日志接口。用户可以自由选择第三方的日志组件作为具体实现，像 log4j，或者 jdk 自带的 logging，common-logging 会通过动态查找的机制，在程序运行时自动找出真正使用的日志库。

当然，common-logging 内部有一个 Simple logger 的简单实现，但是功能很弱。所以使用 common-logging，通常都是配合着 log4j 来使用。使用它的好处就是，代码依赖是 common-logging 而非 log4j， 避免了和具体的日志方案直接耦合，在有必要时，可以更改日志实现的第三方库。

使用 common-logging 的常见代码：

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class A {
    private static Log logger = LogFactory.getLog(this.getClass());
}
```

动态查找原理

Log 是一个接口声明。LogFactory 的内部会去装载具体的日志系统，并获得实现该 Log 接口的实现类。LogFactory 内部装载日志系统的流程如下：

- 首先，寻找 org.apache.commons.logging.LogFactory 属性配置。
- 否则，利用 JDK1.3 开始提供的 service 发现机制，会扫描 classpah 下的 META-INF/services/org.apache.commons.logging.LogFactory 文件，若找到则装载里面的配置，使用里面的配置。
- 否则，从 Classpath 里寻找 commons-logging.properties ，找到则根据里

[1] OSGi(Open Service Gateway Initiative，开放服务网关协议)技术是 Java 动态化模块化系统的一系列规范。

面的配置加载。

- 否则，使用默认的配置：如果能找到 Log4j 则默认使用 log4j 实现，如果没有则使用 JDK14Logger 实现，再没有则使用 commons-logging 内部提供的 SimpleLog 实现。

从上述加载流程来看，只要引入了 log4j 并在 classpath 配置了 log4j.xml ，则 commons-logging 就会使 log4j 使用正常，而代码里不需要依赖任何 log4j 的代码。

Tips：

JCL 在获取第一个 Log 时，通过 LogFactory 进行具体实现的加载，因为是通过类加载器进行动态加载的，所以编译时不会出错，而到运行时才发现动态加载没有成功。

3 日志系统的比较

3.1 Log4J 与 LogBack

LOGBack 作为一个通用可靠、快速灵活的日志框架，将作为 Log4j 的替代和 SLF4J 组成新的日志系统的完整实现。LOGBack 声称具有极佳的性能，“某些关键操作，比如判定是否记录一条日志语句的操作，其性能得到了显著的提高。这个操作在 LogBack 中需要 3 纳秒，而在 Log4J 中则需要 30 纳秒。 LogBack 创建记录器（logger）的速度也更快：13 微秒，而在 Log4J 中需要 23 微秒。更重要的是，它获取已存在的记录器只需 94 纳秒，而 Log4J 需要 2234 纳秒，时间减少到了 1/23。跟 JUL 相比的性能提高也是显著的”。

另外，LOGBack 的所有文档是全面免费提供的，不象 Log4J 那样只提供部分免费文档而需要用户去购买付费文档。

3.2 SLF4J 和 JCL

SLF4J 库类似于 Apache Common-Logging。但是，他在编译时静态绑定真正的 Log 库。使用 SLF4J 时，如果你需要使用某一种日志实现，那么你必须选择正确的 SLF4J 的 jar 包的集合。 如此便可以在 OSGI 中使用了。

另外，SLF4J 支持参数化的 log 字符串，避免了之前为了减少字符串拼接的性能损耗而不得不写的

if(logger.isDebugEnable())

现在你可以直接写：

logger.debug(“current user is: {}”, user)

拼装消息被推迟到了它能够确定是不是要显示这条消息的时候，但是获取参数的代价并没有幸免。同时，日志中的参数若超过三个，则需要将参数以数组的形式传入，如：

```
Object[] params = {value1, value2, value3};
logger.debug(“first value: {}, second value: {} and third value: {}.”, params);
```

common-logging 通过动态查找的机制，在程序运行时自动找出真正使用的

日志库。由于它使用了 ClassLoader 寻找和载入底层的日志库， 导致了象 OSGI 这样的框架无法正常工作，因为 OSGI 的不同的插件使用自己的 ClassLoader。OSGI 的这种机制保证了插件互相独立，然而却使 Apache Common-Logging 无法工作。

现在，hibernate、Jetty、spring-OSGi、Wicket 和 MINA 等项目都已经迁移到了 SLF4J，由此可见 SLF4J 的影响力不可忽视。

第2章 Head First Log4j

3 Log4j 的框架

The previous chapter presented a very simple usage case for log4j. This chapter discusses the log4j architecture and the rules governing its components. Log4j has three main components: loggers, appenders and layouts. These three types of components work together to enable developers to log messages according to their level. They control the format of log messages as well as their output destination.

Tips：

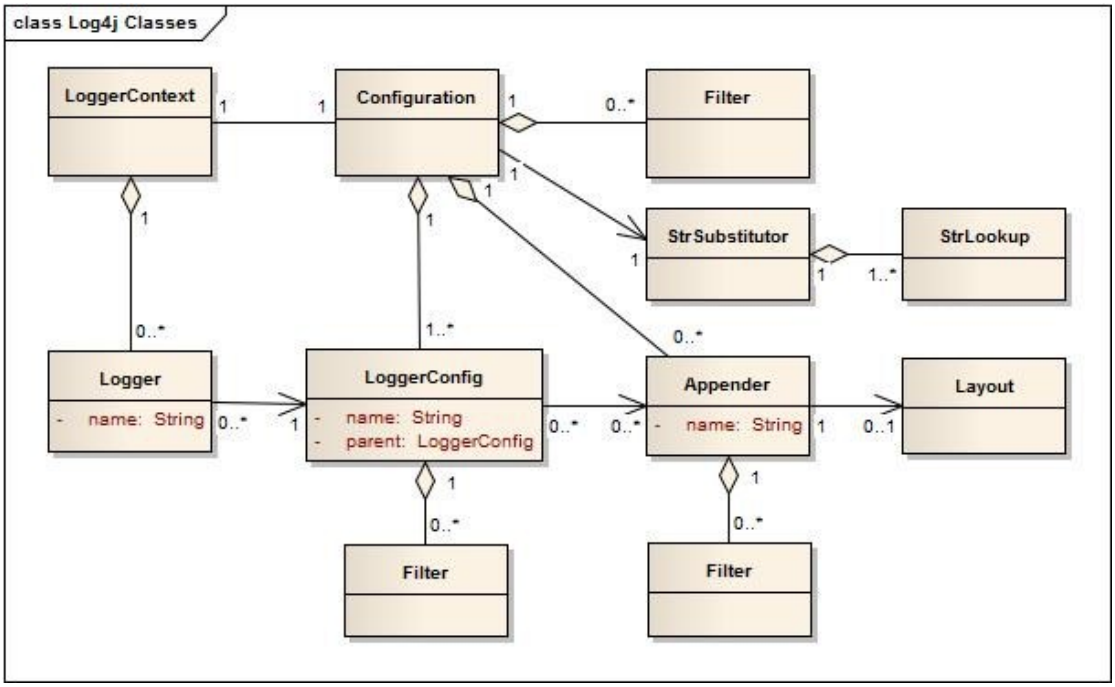
框架的三个组件：记录器、输出源和布局器。

The reader familiar with the java.util.logging API introduced in JDK 1.4, will recognize that log4j's architecture is very similar although log4j offers much more functionality. Log4j requires JDK 1.1 whereas java.util.logging will only run on JDK 1.4. Most of the concepts outlined in this document are reproduced with little variation in java.util.logging albeit with somewhat different names. In case you had any doubts regarding log4j's lineage[1], the present log4j architecture dates back to early 1999, the JDK 1.4 logging API was not even a proposal at the time.

3.1 Log4j 的整体架构

Log4j uses the classes shown in the diagram below.

[1] lineage 英 ['lɪnɪɪdʒ] n. 血统；家系，[遗] 世系



图：Log4j 的框架结构

Tips：

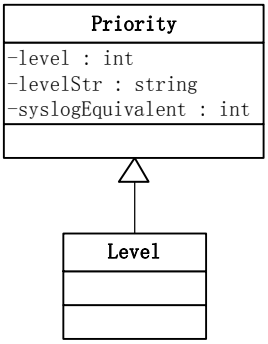
这是从 Log4j2 抄过来的图，应该大致是一样的。

Applications using the Log4j 2 API will request a Logger with a specific name from the LogManager. The LogManager will locate the appropriate LoggerContext and then obtain the Logger from it. If the Logger must be created it will be associated with the LoggerConfig that contains either a) the same name as the Logger, b) the name of a parent package, or c) the root LoggerConfig. LoggerConfig objects are created from Logger declarations in the configuration. The LoggerConfig is associated with the Appenders that actually deliver the LogEvents.

9 深入 Logger 的源码

9.1 日志的级别

下面是日志级别类的继承结构：



图：日志的继承结构

从上面可以看出，日志的级别 Level 中共包含三个字段：

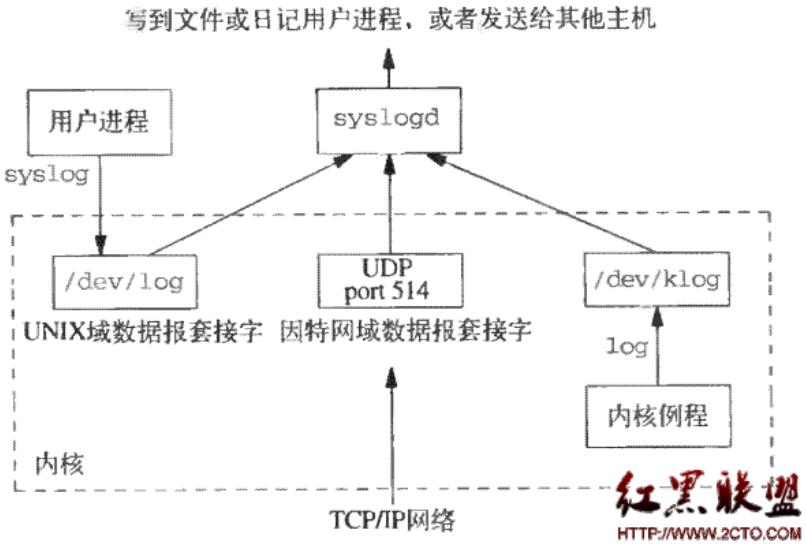
- level：Log4j 的日志级别，Debug 是 10000，Info 是 20000，Warn 是 30000，Error 是 40000，Fatal 是 50000，Trace 是 5000，All 是最小的 Int 值，Off 是最大的 Int 值；
- levelStr：Log4j 的日志级别名字，如 INFO；
- syslogEquivalent：对应 Syslog 日志的级别。

关于 Syslog

系统日志（Syslog）协议是在一个 IP 网络中转发系统日志信息的标准，它是在美国加州大学伯克利软件分布研究中心（BSD）的 TCP/IP 系统实施中开发的，目前已成为工业标准协议，可用它记录设备的日志。Syslog 记录着系统中的任何事件，管理者可以通过查看系统记录随时掌握系统状况。系统日志通过 Syslog 进程记录系统的有关事件，也可以记录应用程序运作事件。通过适当配置，还可以实现运行 Syslog 协议的机器之间的通信。通过分析这些网络行为日志，可追踪和掌握与设备和网络有关的情况。

Unix/Linux 系统中的大部分日志都是通过一种叫做 syslog 的机制产生和维护的。syslog 是一种标准的协议，分为客户端和服务器端，客户端是产生日志消息的一方，而服务器端负责接收客户端发送来的日志消息，并做出保存到特定的日志文件中或者其他方式的处理。

在 Linux 中，常见的 syslog 服务器端程序是 syslogd 守护程序。这个程序可以从三个地方接收日志消息：（1）Unix 域套接字 /dev/log；（2）UDP 端口 514；（3）特殊的设备/dev/klog（读取内核发出的消息）。相应地，产生日志消息的程序就需要通过上述三种方式写入消息，对于大多数程序而言就是向/dev/log 这个套接字发送日志消息。



图：

这里说下 Syslog 中的级别的定义。

Severity 的定义如下：

Numerical Code	Severity
0	Emergency: system is unusable
1	Alert: action must be taken immediately
2	Critical: critical conditions
3	Error: error conditions
4	Warning: warning conditions
5	Notice: normal but significant condition
6	Informational: informational messages
7	Debug: debug-level messages

从上表中可以看出，其级别和 syslogEquivalent 中是对应的，也就是说，在 Level 类中不仅指定了 Log4j 的级别，还定义了和 Syslog 中的级别对应的级别。

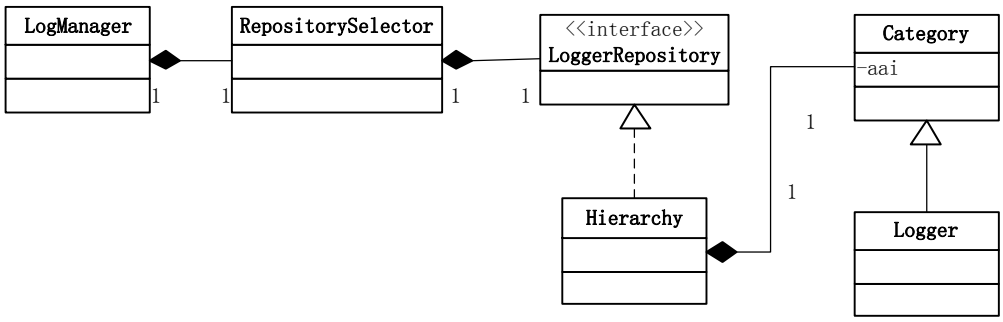
9.2 LogManager

Use the LogManager class to retreive Logger instances or to operate on the current LoggerRepository. When the LogManager class is loaded into memory the default initalzation procedure is inititated. The default intialization procedure is described in the short log4j manual.

Tips:

LogManager 要注意三点：一是，该全局只有一个，在第一次加载到内存的时候进行初始化；二是，加载配置文件是在初始化时完成的；三是，通过该类可以控制 logger；

下图是 LogManager 的结构：



图：继承结构

从上面的继承结构中可以看出 Log4j 的工作原理：

- LogManager 全局唯一并进行初始化；
- LogManager 保存了一个 RepositorySelector 对象，用来选择保存日志的仓库；默认情况下使用 DefaultRepositorySelector 作为选择器；
- 默认的仓库是 Hierarchy，它实现了 LoggerRepository 接口；
- 仓库中保存的对象是日志 Logger；
- LogManager 中的大部分操作都是通过间接获取 LoggerRepository 对象，从而被 LoggerRepository 类代理完成的。

9.2.1 初始化过程

读取初始化配置文件并进行配置。分为两个步骤：

首先从 System.getProperty()中获取系统的配置：log4j.defaultInitOverride、log4j.configuration 和 log4j.configuratorClass 是虚拟机系统初始化时提供的日志配置；configuratorClass 用于指明使用哪个类来加载配置，缺省情况下，xml 文件使用 DOMConfigurator 来加载，属性配置文件 property 通过 PropertyConfigurator 来加载；

另一是获取用户类路径下的 log4j.xml 和 log4j.properties 两个配置文件；

注意，上述文件是按照顺序加载的。如果虚拟机系统已经配置了日志的输出，则下面的两个配置将不起作用。

```
try {
    OptionConverter.selectAndConfigure(url, configuratorClassName,
            LogManager.getLoggerRepository());
} catch (NoClassDefFoundError e) {
    LogLog.warn("Error during default initialization", e);
}
```

上面是 LogManager 中完成初始化的语句，可见，其是通过 OptionConverter 类中提供的工具方法完成的初始化工作。

```
static
public
void selectAndConfigure(URL url, String clazz, LoggerRepository hierarchy) {
```

```
  Configurator configurator = null;
  String filename = url.getFile();

  if(clazz == null && filename != null && filename.endsWith(".xml")) {
    clazz = "org.apache.log4j.xml.DOMConfigurator";
  }

  if(clazz != null) {
    LogLog.debug("Preferred configurator class: " + clazz);
    configurator = (Configurator) instantiateByClassName(clazz,
                                Configurator.class,
                                null);
    if(configurator == null) {
      LogLog.error("Could not instantiate configurator ["+clazz+"].");
      return;
    }
  } else {
    configurator = new PropertyConfigurator();
  }

  configurator.doConfigure(url, hierarchy);
 }
```

该方法最终通过确定配置文件的类型，用配置工具类完成了初始化配置工作。关于 Dom 或 Properties 的相关内容，请参考配置的部分。

9.2.2 初始化过程译文

Log4j 类库没有对它的环境作任何假设。特别是，log4j 没有默认的输出源。然而，在某些定义明确的环境下，日志记录器类的静态的初始化器将尝试自动配置 log4j。java 语言保证在往内存中装载类时，类的静态初始化器仅仅可以被调用一次。不同类装载器可能装载相同类的不同拷贝，记住这是很重要的。Java 虚拟机认为这些相同类的拷贝是完全不相关的。

在依赖运行环境的应用程序的正确入口处，默认的初始化是非常有用的。例如，在 web 服务器(web-server)的控制下，相同的应用程序可以被当作一个独立的应用程序、applet 或者 servlet。

下面定义的是确切的默认注视化算法：

- 设定 log4j.defaultInitO 覆盖系统属性的为任何其它值，“false”将导致 log4j 忽略默认的初始化过程(这个过程)。
- 设定资源字符串变量为 log4j.configuration 的系统属性值。指定默认初始化文件的最好方法是通过 log4j.configuration 的系统属性。万一系统属性 log4j.configuration 没有定义，可以设定字符串变量资源到它默认值

“log4j.properties”。

- 尝试转换资源变量为 URL
- 假如资源变量不能转换为 URL，例如由于一个 MalformedURLException 异常，然后通过调用返回值为 URL 的 org.apache.log4j.helpers.Loader.getResource(resource, Logger.class)方法在 classpath 中查找资源。注意，字符串"log4j.properties"包含(constitutes)一个丑陋的 URL。

  参考 Loader.getResource(java.lang.String)方法，获得查找路径的列表。
- 假如没有 URL，忽略默认的初始化。其它，通过 URL 来配置 log4j。

常常使用 PropertyConfigurator 类解析 URL 来配置 log4j，若 URL 以“.xml”后缀名结束，将使用 DOMConfigurator 来解释。你可以有选择指定自定义的配置器。log4j.configuratorClass 的系统属性值被当作你自定义配置器的完整类名。你所指定的自定义配置器必须实现 Configurator 接口。

9.2.3 解析配置文件

上面提到，在 LogManager 中会去加载日志的配置文件（这里以 Properties 的形式进行讲解），通过解析后，会去调用 PropertyConfigurator 类完成初始化的配置。

首先来看一个方法：

```
void parseCategory(Properties props, Logger logger, String optionKey,
            String loggerName, String value) {
}
props：加载的配置文件
logger：要配置的Logger，如RootLogger
optionKey：配置文件中的有效的键，如log4j.rootlogger
loggerName：Logger的名字，如Root
value：配置文件中，对应键的值
```

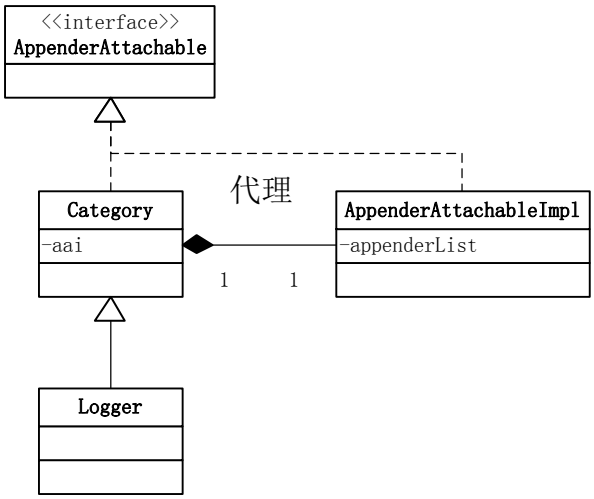
上面这个方法可以看做是用来加载配置的入口。程序在加载初始化的时候会先去找 Root 的配置，解析出来 Root 的 Logger，然后调用上面的方法来加载 Root 日志的 Appender 和 Layout 等。解析顺序如下所示：

- 解析 Root 的相关配置；
- 解析日志工厂；
- 解析其他日志的相关配置

具体过程可以参看代码进行了解。

9.3 Logger

下图是 Logger 的继承结构：



Logger 是 log4j 中最重要的类，通过该类可以获取唯一的日志对象，如果目前还不存在，则创建新的日志对象。

所有的 Logger 对象都由 LogManager 进行管理。Use the LogManager class to retreive Logger instances or to operate on the current LoggerRepository. When the LogManager class is loaded into memory the default initalzation procedure is inititated. The default intialization procedure is described in the short log4j manual.

下面是 LogManager 初始化的静态块：

```
static {
  // By default we use a DefaultRepositorySelector which always returns 'h'.
  // RootLogger继承自Logger，是一个final类，是特殊的日志。
  Hierarchy h = new Hierarchy(new RootLogger((Level) Level.DEBUG));
  repositorySelector = new DefaultRepositorySelector(h);

  /** Search for the properties file log4j.properties in the CLASSPATH.  */
  String override =OptionConverter.getSystemProperty(DEFAULT_INIT_OVERRIDE_KEY,
                              null);

  // if there is no default init override, then get the resource
  // specified by the user or the default config file.
  if(override == null || "false".equalsIgnoreCase(override)) {

    String configurationOptionStr = OptionConverter.getSystemProperty(
                              DEFAULT_CONFIGURATION_KEY,
                              null);

    String configuratorClassName = OptionConverter.getSystemProperty(
                                          CONFIGURATOR_CLASS_KEY,
                            null);
```

```
    URL url = null;

    // if the user has not specified the log4j.configuration
    // property, we search first for the file "log4j.xml" and then
    // "log4j.properties"
    if(configurationOptionStr == null) {
        url = Loader.getResource(DEFAULT_XML_CONFIGURATION_FILE);
        if(url == null) {
            url = Loader.getResource(DEFAULT_CONFIGURATION_FILE);
        }
    } else {
        try {
            url = new URL(configurationOptionStr);
        } catch (MalformedURLException ex) {
            // so, resource is not a URL:
            // attempt to get the resource from the class path
            url = Loader.getResource(configurationOptionStr);
        }
    }

    // If we have a non-null url, then delegate the rest of the
    // configuration to the OptionConverter.selectAndConfigure
    // method.
    if(url != null) {
       LogLog.debug("Using URL ["+url+"] for automatic log4j configuration.");
      try {
          OptionConverter.selectAndConfigure(url, configuratorClassName,
                      LogManager.getLoggerRepository());
      } catch (NoClassDefFoundError e) {
          LogLog.warn("Error during default initialization", e);
      }
    } else {
       LogLog.debug("Could not find resource: ["+configurationOptionStr+"].");
    }
  } else {
      LogLog.debug("Default initialization of overridden by " +
          DEFAULT_INIT_OVERRIDE_KEY + "property.");
  }
}
```

LogManager 初始化的时机：在第一次加载 LogManager 的时候，会调用该静态块进行一些初始化动作。

- 使用类 Hierarchy 作为日志的仓库，即保存日志及其父子关系；
- 初始化 RepositorySelector 为 DefaultRepositorySelector。程序中使用这个 RepositorySelector 的目的是为了程序的灵活性，用户可以自定义日志的仓库，并通过该类来指定使用该仓库作为日志的仓库；
- 读取初始化配置文件并进行配置。这里分为两个步骤：
  首先从 System.getProperty()中获取系统的配置：log4j.defaultInitOverride、log4j.configuration 和 log4j.configuratorClass 是虚拟机系统初始化时提供的日志配置；configuratorClass 用于指明使用哪个类来加载配置，缺省情况下，xml 文件使用 DOMConfigurator 来加载，属性配置文件 property 通过 PropertyConfigurator 来加载；
  另一是获取用户类路径下的 log4j.xml 和 log4j.properties 两个配置文件；
  注意，上述文件是按照顺序加载的。如果虚拟机系统已经配置了日志的输出，则下面的两个配置将不起作用。

Tips：
关于 LogManager 参考相关的内容。

下面是 Logger 中具有的属性：

```
public class Category implements AppenderAttachable {
  protected String   name; // 日志的名字，必有
  volatile protected Level level; // 日志的级别，必有
  volatile protected Category parent; // 日志的父亲节点
  protected ResourceBundle resourceBundle; // Xxx
  protected LoggerRepository repository; // 该日志所在的仓库
  AppenderAttachableImpl aai; // 该日志的输出源
  protected boolean additive = true; // 输出源的叠加性，初值为true

  // 方法略. . .
}
```

9.3.1 输出源的绑定

AppenderAttachable 是一个接口，该接口定义的方法都是对 logger 进行绑定输出源的操作。

其 API 如下所示：

```
public interface AppenderAttachable {
  public void addAppender(Appender newAppender); // 添加Appender
  public  Enumeration getAllAppenders(); // 获取Appender
  public  Appender getAppender(String name);
  public  boolean isAttached(Appender appender); // 判断是否添加过Appender
  void removeAllAppenders(); // 删除Appender
  void removeAppender(Appender appender);
```

```
  void removeAppender(String name);
}
```

下面是该接口的默认实现：

```
public class AppenderAttachableImpl implements AppenderAttachable {
   /** Array of appenders. */
  protected Vector  appenderList;

  // 实现的方法略
}
```

从上述方法中可以看到，保存 Appender 的结构是个 Vector，同一个 Logger 上可以绑定多个输出源。

Tips：

每个 Logger 都有一个 AppenderAttachableImpl 的实现，因此，每个 Logger 可以各自维护各自的输出源。

9.3.2 日志级别的继承

日志的添加是在 Logger 中通过静态的 getLogger 方法实现新日志的添加，此时并未给新添加的日志设置级别，这是合理的。通常用户也不需要关心具体到某个日志的级别，而是在层次关系上设置级别。那么是否可以设置具体日志的级别呢？答案是肯定的，通过 Logger 中的 setLevel 方法可以给特定的日志设置其显示级别。

未设置的日志的级别为 Null 值。

所谓日志级别的继承实际上是在进行打印日志时提供了一种判断策略，而不是在创建的时候直接继承其父亲的级别，此时其级别可能还是 Null 值。级别的继承是日志父子关系上的继承。

```
  public  Level getEffectiveLevel() {
     for(Category c = this; c != null; c=c.parent) {
        if(c.level != null)
         return c.level;
     }
     return null; // If reached will cause an NullPointerException.
  }
```

9.3.3 日志的打印过程

第一步

通过级别进行过滤，判断该日志是否需要进行打印输出。这个过滤分为两层，一是仓库级别的过滤；二是日志级别的过滤，通过 getEffectiveLevel()方法可以获取该日志的有效级别，可能是已有的，也可能是继承自双亲的。

```
  public  void info(Object message) {
```

```
    if(repository.isDisabled(Level.INFO_INT))
        return;
    if(Level.INFO.isGreaterOrEqual(this.getEffectiveLevel()))
        forcedLog(FQCN, Level.INFO, message, null);
}
```

第二步

构造日志事件，并通过调用输出源来进行日志的打印输出。日志间存在着父子关系，这个父子关系是由 Hierarchy 来维护的，每个日志都有其父日志（包括 Root 日志），这个父日志和前缀的层级关系是不同的，关于这点请参考下节的内容。

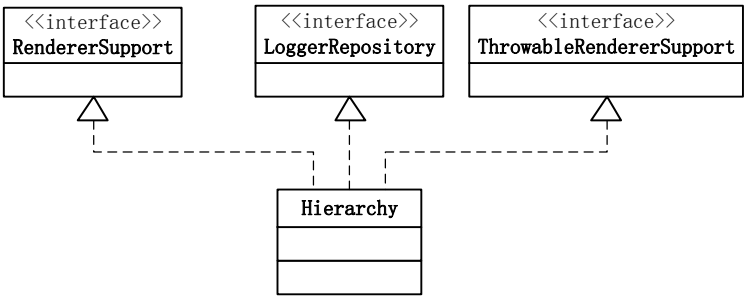
```
public void callAppenders(LoggingEvent event) {
  int writes = 0;

  for(Category c = this; c != null; c=c.parent) {
    // Protected against simultaneous call to addAppender, removeAppender,...
    synchronized(c) {
      if(c.aai != null) {
        writes += c.aai.appendLoopOnAppenders(event);
      }
      if(!c.additive) {
        break;
      }
    }
  }

  if(writes == 0) {
    repository.emitNoAppenderWarning(this);
  }
}
```

Tips：

注意，在配置文件中定义的层级关系实际上会当做一个正常的日志 Logger，而不是一个前缀节点。具体参考配置中的 Logger 的配置。

9.5 Hierarchy

下图是 Hierarchy 的继承结构：



图：Hierarchy 的继承结构

在 Hierarchy 中具有一个唯一的构造函数，其参数是一个 Root 日志。在 log4j 中，所有的日志都是根日志的后代：

```
public
Hierarchy(Logger root) {
  ht = new Hashtable(); // 保存所有的日志，Key是日志的名字，Value是一个Logger
  listeners = new Vector(1); // 仓库的监听者，HierarchyEventListener
  this.root = root;
  // Enable all level levels by default.
  setThreshold(Level.ALL); // 缺省所有日志都打印
  this.root.setHierarchy(this); // 保存根日志的仓库
  rendererMap = new RendererMap();
  defaultFactory = new DefaultCategoryFactory(); // 真正构造日志的工厂
}
```

Tips：

真正的日志构造过程是这样的，通过 Logger 的静态方法来获取日志 → Logger 的仓库（Hierarchy） → 构造 Logger 的工厂→ Logger。

下面我们分析一下日志仓库具有的功能：

```
public interface LoggerRepository {
  // 在设置了仓库的级别后可以通过isDisabled()方法来判断某个级别的日志是否被丢弃了
  boolean isDisabled(int level);
  public void setThreshold(Level level);
  public void setThreshold(String val);
  public Level getThreshold();

  // 从仓库里获取日志等
  public Logger getLogger(String name);
  public Logger getLogger(String name, LoggerFactory factory);
  public Logger getRootLogger();
  public abstract Logger exists(String name);
  public Enumeration getCurrentLoggers();
```

```
  // 仓库的监听器
  public void addHierarchyEventListener(HierarchyEventListener listener);
  public void emitNoAppenderWarning(Category cat);
  public abstract void fireAddAppenderEvent(Category logger, Appender appender);

  // 仓库的相关控制
  public abstract void shutdown();
  public abstract void resetConfiguration();
}
```

下面是其重要实现：

```
public class Hierarchy implements LoggerRepository, RendererSupport,
ThrowableRendererSupport {

  private LoggerFactory defaultFactory;
  private Vector listeners;

  Hashtable ht;
  Logger root;
  RendererMap rendererMap;

  int thresholdInt;
  Level threshold;

  boolean emittedNoAppenderWarning = false;
  boolean emittedNoResourceBundleWarning = false;

  private ThrowableRenderer throwableRenderer = null;

  // 方法略
}
```

9.5.1 LoggerFactory

LoggerFactory 是一个接口，实现该接口的目的是为了创建 Logger 或 Logger 的子类。缺省的实现类是 DefaultCategoryFactory。

9.5.2 父子关系更新

新增加到仓库中的日志需要对当前仓库中已有的日志进行一些更新操作，这个增加日志的动作如下所示：

```
public Logger getLogger(String name, LoggerFactory factory) {
    // System.out.println("getInstance("+name+") called.");
```

```
    CategoryKey key = new CategoryKey(name);
    // Synchronize to prevent write conflicts. Read conflicts (in
    // getChainedLevel method) are possible only if variable
    // assignments are non-atomic.
    Logger logger;

    synchronized (ht) {
        Object o = ht.get(key);
        if (o == null) { // 新日志，则更新父节点
            logger = factory.makeNewLoggerInstance(name);
            logger.setHierarchy(this);
            ht.put(key, logger);
            updateParents(logger);
            return logger;
        } else if (o instanceof Logger) { // 直接找到
            return (Logger) o;
        } else if (o instanceof ProvisionNode) {
            // 是ProvisionNode，更新子节点再更新父节点
            // System.out.println("("+name+") ht.get(this) returned ProvisionNode");
            logger = factory.makeNewLoggerInstance(name);
            logger.setHierarchy(this);
            ht.put(key, logger);
            updateChildren((ProvisionNode) o, logger);
            updateParents(logger);
            return logger;
        } else {
            // It should be impossible to arrive here
            return null; // but let's keep the compiler happy.
        }
    }
}
```

正如我们前面看到的，在 Hierarchy 中有一个 Hashtable 来保存日志对象，其保存的方式是键值对，键为 CategoryKey（可认为是 Logger 的名字），值为日志对象 Logger。

新添加日志时，需要将其放置到 Hashtable 中，并调用 updateParents 方法来更新其父子关系。该方法的源码如下所示：

This method loops through all the *potential* parents of 'cat'. There 3 possible cases:

1) No entry for the potential parent of 'cat' exists We create a ProvisionNode for this potential parent and insert 'cat' in that provision node.

2) There entry is of type Logger for the potential parent. The entry is 'cat's

nearest existing parent. We update cat's parent field with this entry. We also break from the loop because updating our parent's parent is our parent's responsibility.

3) There entry is of type ProvisionNode for this potential parent. We add 'cat' to the list of children for this potential parent.

Tips：

ProvisionNode 继承自 Vector，用来保存子节点，没有其他功能。

```
final private void updateParents(Logger cat) {
    String name = cat.name;
    int length = name.length();
    boolean parentFound = false;

    // System.out.println("UpdateParents called for " + name);

    // if name = "w.x.y.z", loop thourgh "w.x.y", "w.x" and "w", but not
    // "w.x.y.z", 循环了3次，不是四次
    for (int i = name.lastIndexOf('.', length - 1); i >= 0; i = name
            .lastIndexOf('.', i - 1)) {
        String substr = name.substring(0, i);

        // System.out.println("Updating parent : " + substr);
        CategoryKey key = new CategoryKey(substr); // simple constructor
        Object o = ht.get(key);
        // Create a provision node for a future parent.
        // 没有找见，创建一个ProvisionNode，用作未来的父节点
        if (o == null) {
            // System.out.println("No parent "+substr+" found. Creating
ProvisionNode.");
            ProvisionNode pn = new ProvisionNode(cat);
            ht.put(key, pn);
        } else if (o instanceof Category) { // 父节点是一个Logger
            parentFound = true;
            cat.parent = (Category) o;
            // System.out.println("Linking " + cat.name + " -> " +
            // ((Category) o).name);
            break; // no need to update the ancestors of the closest
                   // ancestor
        } else if (o instanceof ProvisionNode) { // 父节点是ProvisionNode
            ((ProvisionNode) o).addElement(cat);
        } else {
            Exception e = new IllegalStateException(
                    "unexpected object type " + o.getClass() + " in ht.");
```

```
                e.printStackTrace();
            }
        }
        // If we could not find any existing parents, then link with root.
        if (!parentFound)
            cat.parent = root; // 不存在父，则其父为根节点
    }
```

更新子节点过程

We update the links for all the children that placed themselves in the provision node 'pn'. The second argument 'cat' is a reference for the newly created Logger, parent of all the children in 'pn' We loop on all the children 'c' in 'pn': If the child 'c' has been already linked to a child of 'cat' then there is no need to update 'c'. Otherwise, we set cat's parent field to c's parent and set c's parent field to cat.

```
final private void updateChildren(ProvisionNode pn, Logger logger) {
    // System.out.println("updateChildren called for " + logger.name);
    final int last = pn.size();

    for (int i = 0; i < last; i++) {
        Logger l = (Logger) pn.elementAt(i);
        // System.out.println("Updating child " +p.name);

        // Unless this child already points to a correct (lower) parent,
        // make cat.parent point to l.parent and l.parent to cat.
        if (!l.parent.name.startsWith(logger.name)) {
            logger.parent = l.parent;
            l.parent = logger;
        }
    }
}
```

此时，更新的前提是，仓库中已经存在该日志，且为 ProvisionNode，新加进来的日志与该已存在的日志同名。判断原来 ProvisionNode 下的子日志的前缀是否为新加入的日志，即判断父子关系是否已经建立正确了，否则需要修改父子关系。

Tips：

节点间的父子关系，即 Logger 中的 parent 属性，在某些时刻并不等同于类名前缀间的匹配关系。如 x.y.z 和 x.y，如果先创建 x.y.z 的日志，则此时在仓库中一共有两个 ProvisionNode，一个 Logger，且该 Logger 的父亲日志是 Root，而不是某个 ProvisionNode，另外需要注意 ProvisionNode 也不能作为某个日志的父亲。

在创建 x.y 时，发现 ProvisionNode 下的 x.y.z 的父日志是 Root，而不是 x.y，因此需要修改。

9.5.3 仓库的监听器

仓库的监听者都是 HierarchyEventListener 接口的子类，目前来看，这个监听器好像没啥用。源码如下所示：

```
public interface HierarchyEventListener {
    public void addAppenderEvent(Category cat, Appender appender);
    public void removeAppenderEvent(Category cat, Appender appender);
}
```

9.6 LoggingEvent

The internal representation of logging events. When an affirmative decision is made to log then a LoggingEvent instance is created. This instance is passed around to the different log4j components.

This class is of concern to those wishing to extend log4j.

Tips：

LoggingEvent 很重要。

LoggingEvent 中有三个构造函数，其中最为重要的一个构造函数为：

```
  /**
     Instantiate a LoggingEvent from the supplied parameters.

     <p>Except {@link #timeStamp} all the other fields of
     <code>LoggingEvent</code> are filled when actually needed.
     <p>
     @param logger The logger generating this event.
     @param level The level of this event.
     @param message  The message of this event.
     @param throwable The throwable of this event.  */
  public LoggingEvent(String fqnOfCategoryClass, Category logger,
                      Priority level, Object message, Throwable throwable) {
    this.fqnOfCategoryClass = fqnOfCategoryClass;
    this.logger = logger;
    this.categoryName = logger.getName();
    this.level = level;
    this.message = message;
    if(throwable != null) {
      this.throwableInfo = new ThrowableInformation(throwable, logger);
    }
    timeStamp = System.currentTimeMillis();
```

```
}
```

其中：
fqnOfCategoryClass 是 Category.class.getName()，是固定的字符串；
logger 是产生该日志事件的日志对象；
throwable 决定是否有异常，可以为空。

Tips：
除该构造函数中提供的参数是必须的外，注意事件戳（创建事件时自动生成），在日志事件类 LoggingEvent 中的其余成员变量都不是必须的。

参考文献

[1] Log4j 使用手册：log4j-manual.pdf

第3章 JDK14 Logging

1 日志的级别

The Level class defines a set of standard logging levels that can be used to control logging output. The logging Level objects are ordered and are specified by ordered integers. Enabling logging at a given level also enables logging at all higher levels.

Clients should normally use the predefined Level constants such as Level.SEVERE.

The levels in descending order are:

- SEVERE (highest value)
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST (lowest value) 更详细，但级别最低

In addition there is a level OFF that can be used to turn off logging, and a level ALL that can be used to enable logging of all messages.

It is possible for third parties to define additional logging levels by subclassing Level. In such cases subclasses should take care to chose unique integer level values and to ensure that they maintain the Object uniqueness property across serialization by defining a suitable readResolve method.

Java 自带的日志分为以下几个级别：

```
// 注意和Log4j中的：Fatal、Error、Warn、Info、Debug对比
public static final Level OFF = new Level("OFF",Integer.MAX_VALUE, defaultBundle);
public static final Level SEVERE = new Level("SEVERE",1000, defaultBundle);
public static final Level WARNING = new Level("WARNING", 900, defaultBundle);
public static final Level INFO = new Level("INFO", 800, defaultBundle);
public static final Level CONFIG = new Level("CONFIG", 700, defaultBundle);

// 注意和Log4j中的：Trace对比，但是更详细，追踪的细节逐渐提高FINE < FINER < FINEST
/**
 * FINE is a message level providing tracing information.
 * <p>
 * All of FINE, FINER, and FINEST are intended for relatively
 * detailed tracing.  The exact meaning of the three levels will
 * vary between subsystems, but in general, FINEST should be used
 * for the most voluminous detailed output, FINER for somewhat
 * less detailed output, and FINE for the  lowest volume (and
```

```
     * most important) messages.
     * <p>
     * In general the FINE level should be used for information
     * that will be broadly interesting to developers who do not have
     * a specialized interest in the specific subsystem.
     * <p>
     * FINE messages might include things like minor (recoverable)
     * failures.  Issues indicating potential performance problems
     * are also worth logging as FINE.
     * This level is initialized to <CODE>500</CODE>.
     */
    public static final Level FINE = new Level("FINE", 500, defaultBundle);
    public static final Level FINER = new Level("FINER", 400, defaultBundle);
    public static final Level FINEST = new Level("FINEST", 300, defaultBundle);

    /**
     * ALL indicates that all messages should be logged.
     * This level is initialized to <CODE>Integer.MIN_VALUE</CODE>.
     */
    public static final Level ALL = new Level("ALL", Integer.MIN_VALUE, defaultBundle);
```

Tips:

7 个级别。

2 LogManager

2.1 成员变量分析

其 API 如下所示:

```
public class LogManager {
    // The global LogManager object
    private static LogManager manager; // 全局唯一，保存日志及相关的控制工作。

    private final static Handler[] emptyHandlers = { };
    private Properties props = new Properties(); // 保存加载的配置
    private PropertyChangeSupport changes
              = new PropertyChangeSupport(LogManager.class);
    private final static Level defaultLevel = Level.INFO;

    // Table of known loggers.  Maps names to Loggers.
    private Hashtable<String,Logger> loggers = new Hashtable<String,Logger>();
    // Tree of known loggers
    private LogNode root = new LogNode(null);
    private Logger rootLogger;
```

```
    // Have we done the primordial reading of the configuration file?
    // (Must be done after a suitable amount of java.lang.System
    // initialization has been done)
    private volatile boolean readPrimordialConfiguration; // 判断是否已经加载了配置
    // Have we initialized global (root) handlers yet?
    // This gets set to false in readConfiguration
    private boolean initializedGlobalHandlers = true;
    // True if JVM death is imminent and the exit hook has been called.
    private boolean deathImminent;
}
```

2.2 初始化过程

与Log4j一样，在java.util.logging中存在一个单一的全局 LogManager 对象，它可用于维护 Logger 和日志服务的一组共享状态。

此 LogManager 对象：

- 管理 Logger 对象的层次结构名称空间。所有指定的 Logger 均存储在此名称空间中。
- 管理一组日志控制属性。这些是供 Handler 及其他日志对象用于自我配置的简单键-值对。

可以使用 LogManager.getLogManager() 获取全局 LogManager 对象。LogManager 对象是在类初始化过程中创建的，过后便不能更改。

在启动时，使用 java.util.logging.manager 系统属性定位 LogManager 类。其源码如下所示：

```
static {
    AccessController.doPrivileged(new PrivilegedAction() {
        public Object run() {
            String cname = null;
            try {
                cname = System.getProperty("java.util.logging.manager");
                if (cname != null) {
                    try {
                        Class clz = ClassLoader.getSystemClassLoader()
                                .loadClass(cname);
                        manager = (LogManager) clz.newInstance();
                    } catch (ClassNotFoundException ex) {
                        Class clz = Thread.currentThread()
                                .getContextClassLoader().loadClass(cname);
                        manager = (LogManager) clz.newInstance();
                    }
```

```
                }
            } catch (Exception ex) {
                System.err.println("Could not load Logmanager \"" + cname
                        + "\"");
                ex.printStackTrace();
            }
            if (manager == null) {
                manager = new LogManager();
            }

            // Create and retain Logger for the root of the namespace.
            manager.rootLogger = manager.new RootLogger();
            manager.addLogger(manager.rootLogger);

            // Adding the global Logger. Doing so in the Logger.<clinit>
            // would deadlock with the LogManager.<clinit>.
            Logger.global.setLogManager(manager);
            manager.addLogger(Logger.global);

            // We don't call readConfiguration() here, as we may be running
            // very early in the JVM startup sequence. Instead
            // readConfiguration
            // will be called lazily in getLogManager().
            return null;
        }
    });
}
```

Tips：

LogManager 是可以指定的。在上面的初始化过程中在加载完 LogManager 后还直接初始化了一个 Root 日志和一个 Global 日志。

虽然 Java 的日志系统也有配置，但与 Log4j 不同，Java 自带的日志系统的配置是在第一次使用 Logger 的时候才加载的；而 Log4j 是在初始化的时候加载和配置的。源码如下所示：

```
public static LogManager getLogManager() {
    if (manager != null) {
        manager.readPrimordialConfiguration(); // 读取原始的配置文件
    }
return manager;
}
```

```
private void readPrimordialConfiguration() {
    if (!readPrimordialConfiguration) { // Boolean值，先判断是否读取过了配置文件
        synchronized (this) {
            if (!readPrimordialConfiguration) {
                // If System.in/out/err are null, it's a good
                // indication that we're still in the
                // bootstrapping phase
                if (System.out == null) {
                    return;
                }
                readPrimordialConfiguration = true;
                try {
                    AccessController.doPrivileged(new PrivilegedExceptionAction() {
                            public Object run() throws Exception {
                                readConfiguration(); // 读取配置
                                return null;
                            }
                        });
                } catch (Exception ex) {
                    // System.err.println("Can't read logging configuration:");
                    // ex.printStackTrace();
                }
            }
        }
    }
}
```

默认情况下，LogManager 从 JRE 目录的属性文件 "lib/logging.properties" 中读取其初始配置。如果编辑该属性文件，则可更改此 JRE 的所有用户的默认日志配置。

另外，LogManager 使用两个可选的允许更好地控制初始配置读取的系统属性：

"java.util.logging.config.class"

"java.util.logging.config.file"

这两个属性可以通过 Preferences API 来设置，既可作为 "java" 命令的命令行属性定义，也可作为传递到 JNI_CreateJavaVM 的系统属性定义。

如果设置了 "java.util.logging.config.class" 属性，则会把属性值当作类名。给定的类将会被加载，并会实例化一个对象，该对象的构造方法负责读取初始配置。（此对象可以使用其他系统属性来控制自己的配置。）此备用配置类可使用 readConfiguration(InputStream) 来定义 LogManager 中的属性。

若未设置"java.util.logging.config.class" ，则使用 "java.util.logging.config.file"

系统属性来指定一个属性文件（以 java.util.Properties 格式）。从此文件读取初始日志配置。

如果这两个属性都没有定义，则如上所述，LogManager 将从 JRE 目录的属性文件 "lib/logging.properties" 中读取其初始配置。

```
    public void readConfiguration() throws IOException, SecurityException {
     checkAccess();

    // if a configuration class is specified, load it and use it.
    // 通过配置类的方式加载配置
    String cname = System.getProperty("java.util.logging.config.class");
    if (cname != null) {
        try {
         // Instantiate the named class.  It is its constructor's
         // responsibility to initialize the logging configuration, by
         // calling readConfiguration(InputStream) with a suitable stream.
         try {
             Class clz = ClassLoader.getSystemClassLoader().loadClass(cname);
             clz.newInstance();
             return;
         } catch (ClassNotFoundException ex) {
             Class clz =
Thread.currentThread().getContextClassLoader().loadClass(cname);
             clz.newInstance(); // 注意这里的用法，是在该配置类的构造函数中完成初始化
             return;
         }
        } catch (Exception ex) {
            System.err.println("Logging configuration class \"" + cname + "\" failed");
            System.err.println("" + ex);
            // keep going and useful config file.
        }
    }

    // 配置文件以流的方式进行加载
    String fname = System.getProperty("java.util.logging.config.file");
    if (fname == null) {
        fname = System.getProperty("java.home");
        if (fname == null) {
         throw new Error("Can't find java.home ??");
        }
        File f = new File(fname, "lib");
        f = new File(f, "logging.properties");
        fname = f.getCanonicalPath();
```

```
    }
    InputStream in = new FileInputStream(fname);
    BufferedInputStream bin = new BufferedInputStream(in);
    try {
        readConfiguration(bin);
    } finally {
        if (in != null) {
         in.close();
        }
    }
   }

   public void readConfiguration(InputStream ins) throws IOException,
SecurityException {
    checkAccess();
    reset();

    // Load the properties
    props.load(ins);
    // Instantiate new configuration objects.
    String names[] = parseClassNames("config");

    for (int i = 0; i < names.length; i++) {
        String word = names[i];
        try {
         Class clz = ClassLoader.getSystemClassLoader().loadClass(word);
         clz.newInstance();
        } catch (Exception ex) {
         System.err.println("Can't load config class \"" + word + "\"");
         System.err.println("" + ex);
         // ex.printStackTrace();
        }
    }

    // Set levels on any pre-existing loggers, based on the new properties.
    setLevelsOnExistingLoggers();

    // Notify any interested parties that our properties have changed.
    changes.firePropertyChange(null, null, null);

    // Note that we need to reinitialize global handles when
    // they are first referenced.
```

```
    synchronized (this) {
        initializedGlobalHandlers = false;
    }
    }
```

Tips：

和 Log4j 类似，配置实际上在启动完成后也可以配置。

2.3 配置的语法格式

Logger 和 Handler 的属性名称是以圆点分隔的 Logger 或 Handler 的名称开头。

全局日志属性可以包括：

属性 "handlers"。该属性为 handler 类定义类名的空白或逗号分隔列表，以便作为处理程序在根 Logger（该 Logger 名为 ""）中加载和注册。每个类名必须用于具有默认构造方法的 Handler 类。注意，刚开始使用这些 Handler 时，它们可能是以延迟方式创建的。

属性 "<logger>.handlers"。该属性为 handler 类定义空白分隔或逗号分隔的列表，以便作为处理程序加载和注册到指定的 logger。每个类名必须用于一个具有默认构造方法的 Handler 类。注意，刚开始使用这些 Handler 时，它们可能是以延迟方式创建的。

属性 "<logger>.useParentHandlers"。该属性定义一个 boolean 值。默认情况下，每个 logger 除了自己处理日志消息外，还可能调用其父级来处理，这往往也会导致根 logger 来处理消息。将此属性设置为 false 时，需要为此 logger 配置 Handler，否则不传递任何消息。

属性 "config"。此属性允许运行任意配置代码。该属性定义类名的空白或逗号分隔的列表。为每个指定类创建新实例。每个类的默认构造方法都可以执行任意代码来更新日志配置，如设置 logger 级别、添加处理程序、添加过滤器，等等。

注意，在 LogManager 配置期间加载的所有类，其搜索顺序是先从系统类路径中搜索，然后才从用户类中搜索。这包括 LogManager 类、任何 config 类和任何 handler 类。

Logger 是按其圆点分隔的名称被组织到命名层次结构中的。因此，"a.b.c" 是 "a.b" 的子级，但 "a.b1" 和 a.b2" 属于同一级。

假定所有以 ".level" 结尾的名称的属性为 Logger 定义日志级别。因此，"foo.level" 就为名称为 "foo" 的 logger 定义了日志级别，进而为指定层次结构中它的所有子级也逐个定义了日志级别。日志级别是按其在属性文件中的定义顺序应用的。因此，树中子节点的级别设置应该迟于其父级设置。属性名 ".level" 可用于设置树的根级。

Tips：

LogManager 对象上的所有方法都是多线程安全的。

2.4 添加日志的过程

同 Log4j 一样，日志在 LogManager 中只会存在一份儿。但是，在 Java 的日志实现中添加日志的过程更加的复杂，用于保存全局日志的结构是 Hashtable。下面是新增日志的实现过程。

第一步

判断日志是否已存在，已经存在则直接返回。

第二步

将新的日志添加到保存日志的 Hashtable 中。

第三步

根据配置文件，设置日志的级别（如果存在的话）。注意，这里仅仅是字符串的匹配过程。

第四步

根据配置文件，设置日志的 Handler，可以存在多个 Handler，并且可以为 Handler 配置级别和叠加性。注意，这里的 Handler 相当于 Log4j 中的 Appender，同样可以设置级别和叠加性。

第五步

更新该日志的前缀节点的存在情况。如 com.log.Slog，则先查看是否存在名为 com 的日志，不存在则加入并初始化；其次，查看 com.log 的存在性，并进行处理。

Tips：

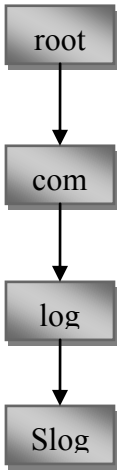
前缀节点的存在性和日志的双亲节点是两个不同的概念。在 Log4j 中同样存在这样的区别，可以对比学习。

第六步

LogNode 被用来维护日志系统中的节点的层次关系和父子关系。所有的节点都可以通过 root 来进行遍历，root 的 LogNode 没有父节点。

```
private LogNode root = new LogNode(null);
```

新加入的 Logger 需要添加到以 root 为根的树中。如，日志 com.log.Slog，此时会存在 4 个 LogNode。



注意，LogNode 维护的是上下级的层次关系，每个节点都是一个 LogNode。注意上面说的，前缀节点见的上下级关系不等同于日志中的父子关系。通过修改，以 root 为根的树中的日志间的父子关系可以很好的维护起来。

下面是 LogNode 的结构。

```
private static class LogNode {
    HashMap<Object,Object> children; // 所有的子LogNode
    Logger logger; // 当前的LogNode是否是一个日志，该属性可以为null值
    LogNode parent; // 该LogNode的父亲

    LogNode(LogNode parent) {
        this.parent = parent;
    }

    // Recursive method to walk the tree below a node and set
    // a new parent logger.
    void walkAndSetParent(Logger parent) {
        if (children == null) {
            return;
        }
        Iterator values = children.values().iterator();
        while (values.hasNext()) {
            LogNode node = (LogNode) values.next();
            if (node.logger == null) {
             node.walkAndSetParent(parent);
            } else {
                doSetParent(node.logger, parent);
             }
    }
    }
}
```

第七步

更新日志间的父子关系。这个过程是这样的，从当前日志的跟节点开始遍历 LogNode 树，直到找到一个层级关系，该层的是一个日志节点。调用 doSetParent() 方法，将该层所在的日志当做当前日志的父亲节点。

注意：Logger 间的父子关系是在 Logger 中维护的。

3 Logger

Logger 对象用来记录特定系统或应用程序组件的日志消息。一般使用圆点分隔的层次名称空间来命名 Logger。Logger 名称可以是任意的字符串，但是它们一般应该基于被记录组件的包名或类名，如 java.net 或 javax.swing。此外，可以创建“匿名”的 Logger，其名称未存储在 Logger 名称空间中。

可通过调用某个 getLogger 工厂方法来获得 Logger 对象。这些方法要么创建一个新 Logger，要么返回一个合适的现有 Logger。

日志消息被转发到已注册的 Handler 对象，该对象可以将消息转发到各种目的地，包括控制台、文件、OS 日志等等。

每个 Logger 都跟踪一个“父”Logger，也就是 Logger 名称空间中与其最近的现有祖先。

每个 Logger 都有一个与其相关的 "Level"。这反映了此 logger 所关心的最低 Level。如果将 Logger 的级别设置为 null，那么它的有效级别继承自父 Logger，这可以通过其父 Logger 一直沿树向上递归得到。

可以根据日志配置文件的属性来配置日志级别，在 LogManager 类的描述中对此有所说明。但是也可以通过调用 Logger.setLevel 方法动态地改变它。如果日志级别改变了，则此变化也会影响它的子 logger，因为任何级别为 null 的子 logger 的有效级别都继承自它的父 Logger。

对于每次日志记录调用，Logger 最初都依照 logger 的有效日志级别对请求级别（例如 SEVERE 或 FINE）进行简单的检查。如果请求级别低于日志级别，则日志记录调用将立即返回。

通过此初始（简单）测试后，Logger 将分配一个 LogRecord 来描述日志记录消息。接着调用 Filter（如果存在）进行更详细的检查，以确定是否应该发布该记录。如果检查通过，则将 LogRecord 发布到其输出 Handler。在默认情况下，logger 也将 LogRecord 沿树递推发布到其父 Handler。

每个 Logger 都有一个与其关联的 ResourceBundle 名称。该指定的包用于本地化日志消息。如果一个 Logger 没有自己的 ResourceBundle 名称，则它将通过其父 Logger 沿树递归继承到 ResourceBundle 名称。

大多数 logger 输出方法都带有 "msg" 参数。此 msg 参数可以是一个原始值，也可以是一个本地化的键。在格式化期间，如果 logger 具有（或继承）一个本地化 ResourceBundle，并且 ResourceBundle 包含 msg 字符串的映射关系，那么用本地化值替换 msg 字符串。否则使用原来的 msg 字符串。通常，格式器使用 java.text.MessageFormat 形式的格式来格式化参数，例如，格式字符串 "{0} {1}" 将两个参数格式化为字符串。

将 ResourceBundle 名称映射到 ResourceBundle 时，Logger 首先试图使用

该线程的 ContextClassLoader。如果 ContextClassLoader 为 null，则 Logger 将尝试 SystemClassLoader。作为初始实现中的临时过渡功能，如果 Logger 无法从 ContextClassLoader 或 SystemClassLoaderis 中找到一个 ResourceBundle，则 Logger 将会向上搜索类堆栈并连续调用 ClassLoader 来试图找到 ResourceBundle（此调用堆栈搜索是为了允许容器过渡到使用 ContextClassLoader，该功能可能在以后版本中取消）。

格式化（包括本地化）是输出 Handler 的责任，它通常会调用格式器。

注意：

日志的格式化输出不必同步发生。它可以延迟，直到 LogRecord 被实际写入到外部接收器。

日志记录方法划分为 5 个主要类别：

- 一系列的 "log" 方法，这种方法带有日志级别、消息字符串，以及可选的一些消息字符串参数。
- 一系列的 "logp" 方法（即 "log precise"），其与 "log" 方法相似，但是带有显式的源类名称和方法名称。
- 一系列的 "logrb" 方法（即 "log with resource bundle"），其与 "logp" 方法相似，但是带有显式的在本地化日志消息中使用的资源包名称。
- 还有跟踪方法条目（"entering" 方法）、方法返回（"exiting" 方法）和抛出异常（"throwing" 方法）的便捷方法。
- 最后，还有一系列在非常简单的情况下（如开发人员只想为给定的日志级别记录一条简单的字符串）使用的便捷方法。这些方法按标准级别名称命名（"severe"、"warning"、"info" 等等），并带有单个参数，即一个消息字符串。

对于不带显式源名和方法名的方法，日志记录框架将尽可能确定日志记录方法中调用了哪个类和方法。但是应认识到，这样自动推断的信息可能只是近似的，甚至可能是完全错误的。这是因为允许虚拟机在 JIT 编译时可以进行广泛的优化，并且可以完全移除栈帧，导致它无法可靠地找到调用的类和方法。

Logger 上执行的所有方法都是多线程安全的。

3.1 获取一个日志

在 Logger 类中有几个类方法 getLogger，可以方便的获取 Logger。其源码如下所示。

```
Find or create a logger for a named subsystem. If a logger has already been created with
the given name it is returned. Otherwise a new logger is created.
If a new logger is created its log level will be configured based on the LogManager
configuration and it will configured to also send logging output to its parent's handlers.
It will be registered in the LogManager global namespace.
Parameters:
name A name for the logger. This should be a dot-separated name and should normally be
based on the package name or class name of the subsystem, such as java.net or javax.swing
public static synchronized Logger getLogger(String name) {
```

```
    LogManager manager = LogManager.getLogManager();
    return manager.demandLogger(name);
}
```

可见，日志的添加过程是通过 LogManager 代理完成的。

在 Logger 中，还支持创建匿名的日志，其源码如下所示。

```
public static synchronized Logger getAnonymousLogger() {
    LogManager manager = LogManager.getLogManager();
    Logger result = new Logger(null, null);
    result.anonymous = true;
    Logger root = manager.getLogger("");
    result.doSetParent(root);
    return result;
}
```

3.2 记录一条日志

日志记录方法划分为 5 个主要类别：

- 一系列的 "log" 方法，这种方法带有日志级别、消息字符串，以及可选的一些消息字符串参数。
- 一系列的 "logp" 方法（即 "log precise"），其与 "log" 方法相似，但是带有显式的源类名称和方法名称。
- 一系列的 "logrb" 方法（即 "log with resource bundle"），其与 "logp" 方法相似，但是带有显式的在本地化日志消息中使用的资源包名称。
- 还有跟踪方法条目（"entering" 方法）、方法返回（"exiting" 方法）和抛出异常（"throwing" 方法）的便捷方法。
- 最后，还有一系列在非常简单的情况下（如开发人员只想为给定的日志级别记录一条简单的字符串）使用的便捷方法。这些方法按标准级别名称命名（"severe"、"warning"、"info" 等等），并带有单个参数，即一个消息字符串。

3.3 Logger 的扩展

程序员可以子类化 Logger 来对日志进行扩展。注意，对于名称空间中的任意点，LogManager 类都可以提供自身的指定 Logger 实现。因此，Logger 的任何子类（它们与新的 LogManager 类一起实现的情况除外）要注意应该从 LogManager 类获得一个 Logger 实例，并应该将诸如 "isLoggable" 和 "log(LogRecord)" 这样的操作委托给该实例。注意，为了截取所有的日志记录输出，子类只需要重写 log(LogRecord) 方法。所有其他日志记录方法作为在此 log(LogRecord) 方法上的调用而实现。

4 LogRecord

LogRecord 对象用于在日志框架和单个日志 Handler 之间传递日志请求。

将 LogRecord 传递到日志框架中后，它在逻辑上已经属于该框架，客户端应用程序不应再使用或更新它。

注意，如果客户端应用程序尚未显式指定源方法名和源类名，则 LogRecord 类将在第一次访问它们时通过解析调用堆栈来自动推导（根据对 getSourceMethodName 或 getSourceClassName 的调用）。因此，如果日志 Handler 要将 LogRecord 传递给另一个线程或者通过 RMI 传输它，并且如果它希望后续获取方法名或类名信息，则其应该调用 getSourceClassName 和 getSourceMethodName 的其中之一来强制将值填入。

作为日志请求，LogRecord 是可以序列化的，但是在序列化的过程中需要注意以下事项：

- LogRecord 类是可序列化的。
- 因为参数数组中的对象可能不可序列化，所以在序列化过程中，应该写入参数数组中所有对象的相应 String（使用 Object.toString）。
- ResourceBundle 不是作为序列化形式的一部分传输的，但是资源包的名称是，而接收对象的 readObject 方法将尝试查找合适的资源包。

5 Handler

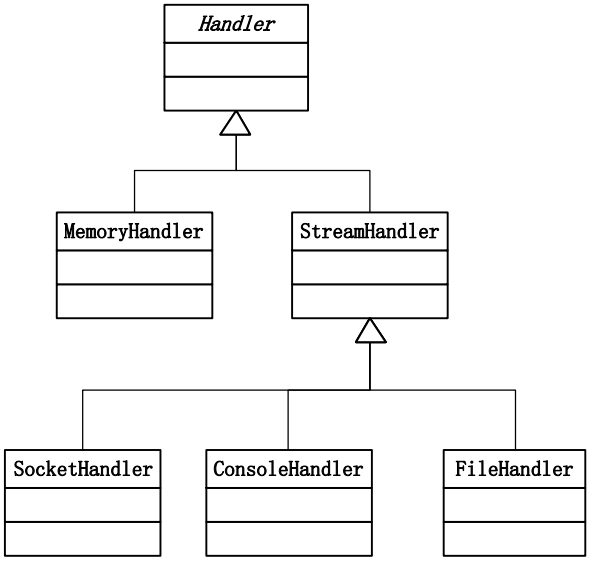
Handler 对象从 Logger 中获取日志信息，并将这些信息导出。例如，它可将这些信息写入控制台或文件中，也可以将这些信息发送到网络日志服务中，或将其转发到操作系统日志中。

可通过执行 setLevel(Level.OFF) 来禁用 Handler，并可通过执行适当级别的 setLevel 来重新启用。下面是 Handler 的 API。

```
public abstract class Handler {
    private static final int offValue = Level.OFF.intValue();
    private LogManager manager = LogManager.getLogManager();
    private Filter filter;
    private Formatter formatter;
    private Level logLevel = Level.ALL;
    private ErrorManager errorManager = new ErrorManager();
    private String encoding;

    // Package private support for security checking.  When sealed
    // is true, we access check updates to the class.
    boolean sealed = true;
    public abstract  void close() // 关闭 Handler，并释放所有相关的资源。
    public abstract  void flush() //刷新所有的缓冲输出。
    public String getEncoding() //返回该 Handler 的字符编码。
    public void setEncoding(String encoding) //设置该 Handler 所用的字符编码。
```

```
    public ErrorManager getErrorManager() //获取该 Handler 的 ErrorManager。
    public Filter getFilter() //获得该 Handler 的当前 Filter。
    public Formatter getFormatter() //返回该 Handler 的 Formatter。
    public Level getLevel() //获得用于指定该 Handler 所记录信息的日志级别。
    public boolean isLoggable(LogRecord record)
    //检查该 Handler 是否实际记录给定的 LogRecord。
    public abstract  void publish(LogRecord record) //发布 LogRecord。
    public protected  void reportError(String msg, Exception ex, int code)
    //用于向该 Handler 的 ErrorManager 报告错误的受保护便利方法。
    public void setErrorManager(ErrorManager em) //为该 Handler 定义一个 ErrorManager。
    public void setFilter(Filter newFilter) //设置 Filter，以控制该 Handler 的输出。
    public void setFormatter(Formatter newFormatter) //设置 Formatter。
    public void setLevel(Level newLevel) //设置日志级别，指定该 Handler 所记录的信息级别。
}
```

Handler 类通常使用 LogManager 属性来设置 Handler 的 Filter、Formatter 和 Level 的默认值。有关每个具体的 Handler 类，请参阅指定的文档。下面是当前 Handler 的实现。



5.1 MemoryHandler

Handler 在内存中的循环缓冲区中对请求进行缓冲处理。

通常，此 Handler 只将传入的 LogRecords 存储到内存缓冲区，并丢弃原来的记录。此缓冲非常经济，并且避免了格式化开销。在一定的触发条件下，MemoryHandler 将其当前的缓冲区内容 push 到目标 Handler 中，此 Handler 通常将内容发布到外界。

有三种主要模型用于触发缓冲区的 push 操作：

- 传入的 LogRecord 类型大于预先定义的 pushLevel 级别。
- 外部类显式地调用 push 方法。

- 如果记录符合所需的某些标准，则子类重写 log 方法，并扫描每个传入的 LogRecord，调用 push。

配置：

默认情况下，使用以下 LogManager 配置属性初始化每个 MemoryHandler。如果没有定义该属性（或者有无效的值），则使用指定的默认值。如果没有定义默认值，则抛出 RuntimeException。

- java.util.logging.MemoryHandler.level 指定 Handler 的级别（默认为 Level.ALL）。
- java.util.logging.MemoryHandler.filter 指定要使用的 Filter 类的名称（默认为无 Filter）。
- java.util.logging.MemoryHandler.size 定义缓冲区的大小（默认为 1000）。
- java.util.logging.MemoryHandler.push 定义 pushLevel（默认为 level.SEVERE）。
- java.util.logging.MemoryHandler.target 指定目标 Handler 类的名称（无默认值）。

Tips：

MemoryHandler 起到的是缓冲的作用，就像继承自 FilterInputStream 的 BufferedInputStream 一样，在一定条件下再将日志输出到指定的 Handler。

另外，由于缓存日志的数组在缓存时是循环进行的，因此，日志存在被覆盖的可能。

5.2 StreamHandler

StreamHandler 是基于流的日志 Handler 的基类。此类主要作为基类,或支持实现其他日志 Handlers 所用的类，主要用于将日志记录 LogRecords 发布到给定 java.io.OutputStream。

配置：

默认情况下，每个 SocketHandler 都是使用以下 StreamHandler 配置属性执行初始化的。如果未定义属性（或者属性具有无效值），则使用指定的默认值。

- java.util.logging.StreamHandler.level 指定 Handler 的默认级别（默认值为 Level.INFO）。
- java.util.logging.StreamHandler.filter 指定要使用的 Filter 类的名称（默认值非 Filter）。
- java.util.logging.StreamHandler.formatter 指定要使用的 Formatter（默认值为 java.util.logging.SimpleFormatter）。
- java.util.logging.StreamHandler.encoding 要使用的字符集编码的名称（默认值为默认平台编码）。

5.3 ConsoleHandler

此 Handler 向 System.err 发布日志记录。默认情况下，使用 SimpleFormatter 生成简短的摘要。

public class ConsoleHandler extends StreamHandler

配置:

默认情况下，每个 ConsoleHandler 都是使用以下 LogManager 配置属性执行初始化的。如果没有定义属性（或者属性具有非法值），则使用指定的默认值。

- java.util.logging.ConsoleHandler.level 为 Handler 指定默认的级别（默认为 Level.INFO）。
- java.util.logging.ConsoleHandler.filter 指定要使用的 Filter 类的名称（默认为无 Filter）。
- java.util.logging.ConsoleHandler.formatter 指定要使用的 Formatter 类的名称（默认为 java.util.logging.SimpleFormatter）。
- java.util.logging.ConsoleHandler.encoding 指定要使用的字符集编码的名称（默认为使用默认平台的编码）。

5.4 SocketHandler

简单的网络日志 Handler。将 LogRecords 发布到网络流连接。默认情况下，XMLFormatter 类用于格式化。

配置:

默认情况下，每个 SocketHandler 是使用以下 LogManager 配置属性执行初始化的。如果未定义属性（或者属性具有无效值），则使用指定的默认值。

- java.util.logging.SocketHandler.level 指定 Handler 的默认级别（默认值为 Level.ALL）。
- java.util.logging.SocketHandler.filter 指定要使用的 Filter 类的名称（默认值非 Filter）。
- java.util.logging.SocketHandler.formatter 指定要使用的 Formatter（默认值为 java.util.logging.XMLFormatter）。
- java.util.logging.SocketHandler.encoding 要使用的字符集编码的名称（默认值为默认平台编码）。
- java.util.logging.SocketHandler.host 指定要连接到的目标主机名（无默认值）。
- java.util.logging.SocketHandler.port 指定要使用的目标 TCP 端口（无默认值）。

输出 IO 流是缓冲的，但是在每次写入 LogRecord 后都将刷新。

5.5 FileHandler

简单的文件日志记录 Handler。FileHandler 可以写入指定的文件，也可以写

入文件轮换集。

public class FileHandler extends StreamHandler

对于文件轮换集而言，到达每个文件的给定大小限制后，就关闭该文件，将其轮换出去，并打开新的文件。通过在基本文件名中添加 "0"、"1"、"2" 等来依次命名旧文件。

默认情况下，IO 库中启用了缓冲，但当缓冲完成时，每个日志记录都要被刷新。默认情况下，XMLFormatter 类用于格式化。

配置：

默认情况下，每个 FileHandler 都是使用以下 LogManager 配置属性执行初始化的。如果没有定义属性（或者属性具有非法值），则使用指定的默认值。

- java.util.logging.FileHandler.level 为 Handler 指定默认的级别（默认为 Level.ALL）。
- java.util.logging.FileHandler.filter 指定要使用的 Filter 类的名称（默认为无 Filter）。
- java.util.logging.FileHandler.formatter 指定要使用的 Formatter 类的名称（默认为 java.util.logging.XMLFormatter）。
- java.util.logging.FileHandler.encoding 指定要使用的字符集编码的名称（默认使用默认的平台编码）。
- java.util.logging.FileHandler.limit 指定要写入到任意文件的近似最大量（以字节为单位）。如果该数为 0，则没有限制（默认为无限制）。
- java.util.logging.FileHandler.count 指定有多少输出文件参与循环（默认为 1）。
- java.util.logging.FileHandler.pattern 为生成的输出文件名称指定一个模式。有关细节请参见以下内容（默认为 "%h/java%u.log"）。
- java.util.logging.FileHandler.append 指定是否应该将 FileHandler 追加到任何现有文件上（默认为 false）。

日志文件的模式

日志文件的模式实际上包含两个部分，一是文件的存储路径；另一是存在文件轮换集时的处理策略。模式由包括以下特殊组件的字符串组成，则运行时要替换这些组件：

"/" 本地路径名分隔符

"%t" 系统临时目录

"%h" "user.home" 系统属性的值

"%g" 区分循环日志的生成号

"%u" 解决冲突的唯一号码

"%%" 转换为单个百分数符号"%"

如果未指定 "%g" 字段，并且文件计数大于 1，那么生成号将被添加到所生成文件名末尾的小数点后面。

例如，文件计数为 2 的 "%t/java%g.log" 模式通常导致在 Solaris 系统中将日志文件写入 /var/tmp/java0.log 和 /var/tmp/java1.log，而在 Windows 95 中，

则将其写入 C:\TEMP\java0.log 和 C:\TEMP\java1.log。两者都是按照 0、1、2 等的序列安排生成号。

通常，将唯一字段 "%u" 设置为 0。但是如果 FileHandler 试图打开文件名并查找当前被另一个进程使用的文件，则增加唯一的字段号并再次重试。重复此操作直到 FileHandler 找到当前没有被使用的文件名。如果有冲突并且没有指定 "%u" 字段，则将该字段添加到文件名末尾的小数点后（它将位于所有自动添加的生成号后面）。

因此，如果三个进程都试图将日志记录到 fred%u.%g.txt，那么它们可能将 fred0.0.txt、fred1.0.txt、fred2.0.txt 作为其循环序列中的首个文件而结束。

注意，使用本地磁盘文件系统时，使用唯一的 id 以避免冲突是系统可靠运行的唯一保证。

6 Filter

Filter 可用于为记录内容提供比记录级别所提供的更细粒度的控制。

每个 Logger 和 Handler 都有一个关联的过滤器。Logger 或 Handler 可以调用 isLoggable 方法来检查是否应该发布给定的 LogRecord。如果 isLoggable 返回 false，则丢弃 LogRecord。

7 Formatter

Formatter 为格式化 LogRecords 提供支持。一般来说，每个日志记录 Handler 都有关联的 Formatter。Formatter 接受 LogRecord，并将它转换为一个字符串。

有些 Formatter（如 XMLFormatter）需要围绕一组格式化记录来包装头部和尾部字符串。可以使用 getHeader 和 getTail 方法来获得这些字符串。

在 JDK 中提供了两个 Formatter 的是实现：

- SimpleFormatter
- XMLFormatter

8 配置

8.1 系统配置的方法

通过配置类进行

通过系统属性"java.util.logging.config.class"可以加载日志初始化的配置类，并通过其构造方法完成日志系统的初始化配置。

```
String cname = System.getProperty("java.util.logging.config.class");
```

通过配置文件

通过系统属性"java.util.logging.config.file"加载配置文件，完成配置。

```
String fname = System.getProperty("java.util.logging.config.file");
```

通过配置文件

上面是通过系统属性指定的配置文件路径。这里是在 java.home 下的配置文件：

```
fname = System.getProperty("java.home");
if (fname == null) {
    throw new Error("Can't find java.home ??");
}
File f = new File(fname, "lib");
f = new File(f, "logging.properties");
fname = f.getCanonicalPath();
```

Tips：

在配置文件中可以进行的配置会在下面进行介绍。

8.2 嵌套配置

在配置文件中，通过"config"属性键，可以指定一个或多个配置类，其形式为配置类的完全类名，通过空格或逗号分隔。在配置的时候，遇到这样的配置会去加载对应的配置类，通过构造函数进行相应的配置工作。

8.3 配置日志的级别

日志级别的配置对具有如下的形式：

com.yang.log.level = info

在进行配置的时候会查看是否有名为 com.yang.log 的 Logger 存在，如果有，则将其级别设置为 info。JDK14 日志中不识别的日志级别会被忽略。

8.4 全局 Handler 配置

handlers = handler1 handler2, handler3

handler1.level = info

handlers 是 Key，Value 是一堆 Handler，通过空白符或逗号分隔。handler1.level 用于配置第一个 Handler 的级别。

Tips：

JDK14 日志的配置对 handlers 的处理是延时的，即在处理 Logger 的时候才会进行配置，在这之前，其配置属性维护在 LogManager 的 Properties 中

8.5 局部 Handler 配置

请查看 Handler 部分的讲解。

参考文献

[1] 官方文档及源码。

第4章 Head First JCL

1 什么是 JCL?

Tips:

JCL 是一个通用的日志接口，可以动态绑定 Log4j 和 JDK 自带的日志，同时自己提供了一个简单的日志实现 SimpleLog。对其他日志系统的支持尚不明确。

commons-logging 是 Apache commons 类库中的一员。Apache commons 类库是一个通用的类库，提供了基础的功能，比如说 commons-fileupload，commons-httpclient，commons-io，commons-codes 等。

commons-logging 能够选择使用 Log4j 还是 JDK Logging，但是他不依赖 Log4j，JDK Logging 的 API。如果项目的 classpath 中包含了 log4j 的类库，就会使用 log4j，否则就使用 JDK Logging。使用 commons-logging 能够灵活的选择使用那些日志方式，而且不需要修改源代码。

Commons Logging (JCL)提供的是一个日志(Log)接口(interface)，同时兼顾轻量级和不依赖于具体的日志实现工具。它提供给中间件/日志工具开发者一个简单的日志操作抽象，允许程序开发人员使用不同的具体日志实现工具。

The Apache Commons Logging (JCL) provides a Log interface that is intended to be both *light-weight* and an *independent* abstraction of other logging toolkits. It provides the middleware/tooling developer with a simple logging abstraction, that allows the user (application developer) to plug in a specific logging implementation.

引用：dwr 主页上面的一句话：

> DWR currently depends on Apache Commons Logging. Because we use commons-logging you are free to choose your logging implementation (Log4j is a common choice).

JCL provides thin-wrapper Log implementations for other logging tools, including Log4J, Avalon LogKit (the Avalon Framework's logging infrastructure), JDK 1.4, and an implementation of JDK 1.4 logging APIs (JSR-47) for pre-1.4 systems. The interface maps closely to Log4J and LogKit.

Familiarity with high-level details of the relevant Logging implementations is presumed. 这句话的意思是说，要想使用 JCL 需要了解实现了 JCL 接口的日志工具的更多的细节。

2 JCL 打印日志的操作

2.1 日志的定义

Obtaining a Log Object. To use the JCL SPI from a Java class, include the following import statements:

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

Note that some components using JCL may either extend Log, or provide a component-specific LogFactory implementation. Review the component documentation for guidelines on how commons-logging should be used in such components.

For each class definition, declare and initialize a log attribute as follows:

```
public class CLASS
{
    private Log log = LogFactory.getLog(CLASS.class);
    ...
    ;
}
```

Tips：

JCL 的初始化是在 LogFactory 中的 getFactory()方法中完成的。

Note that for application code, declaring the log member as "static" is more efficient as one Log object is created per class, and is recommended. However this is not safe to do for a class which may be deployed via a "shared" classloader in a servlet or j2ee container or similar environment. If the class may end up invoked with different thread-context-classloader values set then the member must not be declared static. The use of "static" should therefore be avoided in code within any "library" type project.

Tips：

这里需要关注类加载器的特性。通常将 logger 定义成静态变量有较好的效率，但是在不同的类加载器下可能有一些问题。

2.2 打印日志

Messages are logged to a logger, such as log by invoking a method corresponding to priority. The org.apache.commons.logging.Log interface defines the following methods for use in writing log/trace messages to the log:

```
log.fatal(Object message);
log.fatal(Object message, Throwable t);
log.error(Object message);
log.error(Object message, Throwable t);
log.warn(Object message);
log.warn(Object message, Throwable t);
log.info(Object message);
log.info(Object message, Throwable t);
log.debug(Object message);
log.debug(Object message, Throwable t);
```

log.trace(Object message);
log.trace(Object message, Throwable t);

Semantics for these methods are such that it is expected that the severity, from highest to lowest, of messages is ordered as above.

In addition to the logging methods, the following are provided for code guards:

log.isFatalEnabled();
log.isErrorEnabled();
log.isWarnEnabled();
log.isInfoEnabled();
log.isDebugEnabled();
log.isTraceEnabled();

Tips:
和 Log4j 一样，JCL 提供了 6 个级别的日志。

2.3 Serialization Issues

略。

3 初始化过程分析

As far as possible, JCL tries to be as unobtrusive[1] as possible. In most cases, including the (full) commons-logging.jar in the classpath should result in JCL configuring itself in a reasonable manner. There's a good chance that it'll guess (discover) your preferred logging system and you won't need to do any configuration of JCL at all!

Tips：
在类路径下加入 commons-logging.jar 后，在系统启动时，JCL 会自动检测来选择合适的日志系统。

Note, however, that if you have a particular preference then providing a simple commons-logging.properties file which specifies the concrete logging library to be used is recommended, since (in this case) JCL will log only to that system and will report any configuration problems that prevent that system being used.

When no particular logging library is specified then JCL will silently ignore any logging library that it finds but cannot initialise and continue to look for other alternatives. This is a deliberate design decision; no application should fail to run because a "guessed" logging library cannot be used. To ensure an exception is reported when a particular logging library cannot be used, use one of the available JCL configuration mechanisms to force that library to be selected (ie disable JCL's discovery process).

Tips：
通过 commons-logging.properties 配置文件可以指

[1] unobtrusive 英 [ʌnəb'tru:sɪv] adj. 不唐突的；谦虚的；不引人注目的

定要使用的日志系统。当通过配置文件和自动检测两种方式都失败的时候，会使用 JCL 自带的日志系统。通过 JCL 的日志可以看到检测的整个过程。

3.1 日志工厂初始化

下面是获取日志的过程：

```
private Log log = LogFactory.getLog(CLASS.class);
public static Log getLog(Class clazz) throws LogConfigurationException {
    return getFactory().getInstance(clazz);
}
```

从这个定义看出 getFactory()方法给出了当前使用的日志工厂 LogFactory，而在 LogFactory 中，缓存已经找到的 LogFactory 的对象是 Hashtable，是通过键值对的方式存储的，其 Key 为加载该 LogFactory 的类加载器，Value 是加载进来的 LogFactory。

加载进来的 LogFactory 会得到缓存。

如果根据加载器来获取 LogFactory 不成功的话，需要使用该加载器进行加载 LogFactory。其策略如下所示。

Tips：

在通过类加载器进行加载 LogFactory 之前会先尝试获取 commons-logging.properties，来确定加载时用到的加载器和其他的配置属性。

- The org.apache.commons.logging.LogFactory system property.
  属性键 org.apache.commons.logging.LogFactory 是一个参数的键，其值是一个 LogFactory 的完全类名，指明通过该工厂来生成日志。如果没有该属性，则返回 null 值，进行下面的查找。注意和后面的区别，这里是用系统参数的方式，后面是用配置文件的方式。
- The JDK 1.3 Service Discovery mechanism
  基于 JDK 1.3 的服务发现机制来记载生成日志的工厂。其原理是将路径 "META-INF/services/org.apache.commons.logging.LogFactory" 下的文件以流的形式加载到系统中，并读取构造日志的工厂的类名，然后用类加载器加载该工厂类。
- Use the properties file commons-logging.properties file, if found in the class path of this class. The configuration file is in standard java.util.Properties format and contains the fully qualified name of the implementation class with the key being the system property defined above.
  配置文件中的键值对，org.apache.commons.logging.LogFactory 为键，具体的实现类为值。
- Fall back to a default implementation class (org.apache.commons.logging.impl.LogFactoryImpl).
  若上面都没找到，则用 org.apache.commons.logging.impl.LogFactoryImpl

作为工厂的实现类。该类是 JCL 自带的。

If the properties file method of identifying the LogFactory implementation class is utilized, all of the properties defined in this file will be set as configuration attributes on the corresponding LogFactory instance.

In a multi-threaded environment it is possible that two different instances will be returned for the same classloader environment.

Tips：

在 commons-logging.properties 文件中不仅仅可以指定日志的工厂类，还可以有其他的配置项，但这些配置项只针对配置的指定日志工厂类。

3.2 日志工厂的缓存

上面日志工厂的初始化过程中提到，日志工厂是通过类加载器和工厂实现的方式缓存在 Hashtable 中的。实际上，究竟使用什么样的 Hashtable 是可以定制的，但必须是 Hashtable 的子类。

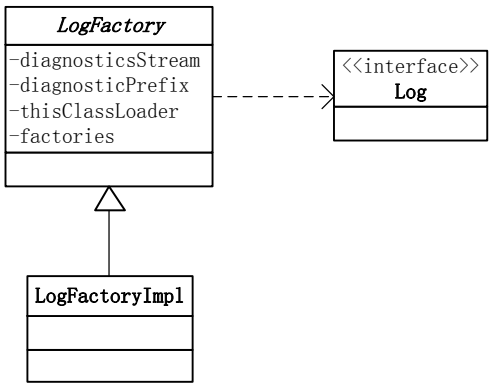
在系统参数中配置"org.apache.commons.logging.LogFactory.HashtableImpl"可以指定自己的 Hashtable 实现类来作为存储日志工厂的存在。注意这里的配置是系统的配置，而不是配置文件中的配置。

缺省情况下，使用"org.apache.commons.logging.impl.WeakHashtable"类作为存储日志工厂的实现。该类是 JCL 实现 Hashtable 的子类。

3.3 LogFactoryImpl

LogFactoryImpl 是 JCL 提供的日志工厂的默认实现，通常使用 JCL 时使用该默认的日志工厂实现即可。但是通过配置是可以使用其他的日志工厂实现的。其继承结构如下所示：

LogFactory 用于创建 Log 实例，Log 是日志的基接口。



图：LogFactoryImpl 的继承结构

3.3.1 初始化容错策略

JCL 提供了三个配置项来确定在日志工厂类加载失败，具体日志实现加载失

败和日志仓库加载失败时的处理策略，其简单介绍如下所示：

表：容错策略

`public static final String ALLOW_FLAWED_CONTEXT_PROPERTY = "org.apache.commons.logging.Log.allowFlawedContext";`
The name (org.apache.commons.logging.Log.allowFlawedContext) of the system property which can be set true/false to determine system behaviour when a bad context-classloader is encountered. When set to false, a LogConfigurationException is thrown if LogFactoryImpl is loaded via a child classloader of the TCCL (this should never happen in sane systems). Default behaviour: true (tolerates bad context classloaders) See also method setAttribute.
缺省为 true

`public static final String ALLOW_FLAWED_DISCOVERY_PROPERTY = "org.apache.commons.logging.Log.allowFlawedDiscovery";`
The name (org.apache.commons.logging.Log.allowFlawedDiscovery) of the system property which can be set true/false to determine system behaviour when a bad logging adapter class is encountered during logging discovery. When set to false, an exception will be thrown and the app will fail to start. When set to true, discovery will continue (though the user might end up with a different logging implementation than they expected). Default behaviour: true (tolerates bad logging adapters) See also method setAttribute.

`public static final String ALLOW_FLAWED_HIERARCHY_PROPERTY = "org.apache.commons.logging.Log.allowFlawedHierarchy";`
The name (org.apache.commons.logging.Log.allowFlawedHierarchy) of the system property which can be set true/false to determine system behaviour when a logging adapter class is encountered which has bound to the wrong Log class implementation. When set to false, an exception will be thrown and the app will fail to start. When set to true, discovery will continue (though the user might end up with a different logging implementation than they expected). Default behaviour: true (tolerates bad Log class hierarchy) See also method setAttribute.

Tips：

这三种策略在置为 false 时，都会抛出异常；而置为 true 时，则由 JCL 来进行容错处理。

3.3.2 日志的发现策略

首先要注意的一点是，在 JCL 中日志工厂和日志并不是简单的生产者和产品之间的关系。JCL 中的工厂的作用有两点，首先是确定日志工厂的实现，这个实现的确定由配置指明，由抽象类 LogFactory 实现日志工厂实现的加载，缺省的实现是 LogFactoryImpl；其次是由实现类确定日志的发现策略，并生成相应的

日志。

日志的发现策略如下：

- 先从配置文件中读取 org.apache.commons.logging.Log 的配置项，如果有，说明用户指明了具体的日志实现，使用之；或者是读取配置文件中旧的配置项 org.apache.commons.logging.log，这个已经弃用了；
- 在配置文件读取后没有找到对应的日志实现，则在系统的属性中读取上述两个的配置项，如果还未找到，则继续查找；
- 此时，进入自动查找流程，依次按照下面的顺序进行查找：
  "org.apache.commons.logging.impl.Log4JLogger"
  "org.apache.commons.logging.impl.Jdk14Logger",
  "org.apache.commons.logging.impl.Jdk13LumberjackLogger",
  "org.apache.commons.logging.impl.SimpleLog"
  上面这些类都实现了 JCL 的 Log 接口，在自动查找的时候会依次使用上面的适配类（即通过该类可以在对应的日志系统中同时创建日志），适配类的查找成功与否是通过异常检测进行的。

3.3.3 日志的发现策略

有必要详细说明一下调用 LogFactory.getLog()时发生的事情。调用该函数会启动一个发现过程，即找出必需的底层日志记录功能的实现，具体的发现过程在下面列出。注意，不管底层的日志工具是怎么找到的，它都必须是一个实现了 Log 接口的类，且必须在 CLASSPATH 之中。Commons Logging API 直接提供对下列底层日志记录工具的支持：Jdk14Logger，Log4JLogger，LogKitLogger，NoOpLogger （直接丢弃所有日志信息），还有一个 SimpleLog。

⑴ Commons 的 Logging 首先在 CLASSPATH 中查找 commons-logging.properties 文件。这个属性文件至少定义 org.apache.commons.logging.Log 属性，它的值应该是上述任意 Log 接口实现的完整限定名称。如果找到 org.apache.commons.logging.Log 属相，则使用该属相对应的日志组件。结束发现过程。

⑵ 如果上面的步骤失败（文件不存在或属相不存在），Commons 的 Logging 接着检查系统属性 org.apache.commons.logging.Log。如果找到 org.apache.commons.logging.Log 系统属性，则使用该系统属性对应的日志组件。结束发现过程。

⑶ 如果找不到 org.apache.commons.logging.Log 系统属性，Logging 接着在 CLASSPATH 中寻找 log4j 的类。如果找到了，Logging 就假定应用要使用的是 log4j。不过这时 log4j 本身的属性仍要通过 log4j.properties 文件正确配置。结束发现过程。

⑷ 如果上述查找均不能找到适当的 Logging API，但应用程序正运行在 JRE 1.4 或更高版本上，则默认使用 JRE 1.4 的日志记录功能。结束发现过程。

⑸ 最后，如果上述操作都失败（JRE 版本也低于 1.4），则应用将使用内建的 SimpleLog。SimpleLog 把所有日志信息直接输出到 System.err。结束发现过程。

获得适当的底层日志工具之后，接下来就可以开始记录日志信息。作为一种标准的 API，Commons Logging API 主要的好处是在底层日志机制的基础上建立

了一个抽象层，通过抽象层把调用转换成与具体实现有关的日志记录命令。 本文提供的示例程序会输出一个提示信息，告诉你当前正在使用哪一种底层的日志工具。

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class CommonLogTest {
    private static Log log = LogFactory.getLog(CommonLogTest.class);

    public static void main(String[] args) {
        log.error("ERROR");
        log.debug("DEBUG");
        log.warn("WARN");
        log.info("INFO");
        log.trace("TRACE");
        System.out.println(log.getClass());
    }
}
```

请试着在不同的环境配置下运行这个程序。例如，

1. 在不指定任何属性的情况下运行这个程序，这时默认将使用 Jdk14Logger；

2. 然后指定系统属性 -Dorg.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog 再运行程序，这时日志记录工具将是 SimpleLog；

3. 把 Log4J 的类放入 CLASSPATH，默认情况下 log4j 会在类路径上寻找 log4j.properties，如果找到就加载，否则查找系统属性 log4j.properties，若还是没有找到，则报告错误。因此只要正确设置了 log4j 的 log4j.properties 配置文件并放在合适位置，就可以得到 Log4JLogger 输出的信息。

如果没有设置 log4j.properties 配置文件，则会输出以下提示：

```
class org.apache.commons.logging.impl.Log4JLogger
log4j:WARN No appenders could be found for logger (CommonLogTest).
log4j:WARN Please initialize the log4j system properly.
```

4 JCL 提供的日志

Tips：
JCL 中的日志需要实现 Log 接口。

4.1 基本记录器

org.apache.commons.logging.Log 接口是 JCL 的基本记录器，其中定义的方法，按严重性由高到低的顺序有：

```
log.fatal(Object message);
```

log.fatal(Object message, Throwable t);
log.error(Object message);
log.error(Object message, Throwable t);
log.warn(Object message);
log.warn(Object message, Throwable t);
log.info(Object message);
log.info(Object message, Throwable t);
log.debug(Object message);
log.debug(Object message, Throwable t);
log.trace(Object message);
log.trace(Object message, Throwable t);
除此以外，还提供下列方法以便代码保护。
log.isFatalEnabled();
log.isErrorEnabled();
log.isWarnEnabled();
log.isInfoEnabled();
log.isDebugEnabled();
log.isTraceEnabled();

4.2 Log 的实现类

org.apache.commons.logging.Log 的具体实现有如下：

org.apache.commons.logging.impl.Jdk14Logger	使用 JDK1.4
org.apache.commons.logging.impl.Log4JLogger	使用 Log4J
org.apache.commons.logging.impl.LogKitLogger	使用 avalon-Logkit
org.apache.commons.logging.impl.SimpleLog	common-logging 自带日志实现类。它实现了 Log 接口，把日志消息都输出到系统错误流 System.err 中。
org.apache.commons.logging.impl.NoOpLog	common-logging 自带日志实现类。它实现了 Log 接口。 其输出日志的方法中不进行任何操作。

Tips：

JCL 中的实际日志类是 JCL 中 Log 接口的实现类，在实现类中定义具体的日志实现，如 Log4j 中的 Logger，Logger 的创建由 Log4j 负责，并建立二者之间一一对应的关系。同时，在工厂实现中需要缓存 JCL 的 Log，以便于查找，通常这个缓存日志的结构是 Hashtable。

4.3 SimpleLog

JCL is distributed with a very simple Log implementation named org.apache.commons.logging.impl.SimpleLog. This is intended to be a minimal implementation and those requiring a fully functional open source logging system are directed to Log4J.

SimpleLog sends all (enabled) log messages, for all defined loggers, to System.err. The following system properties are supported to configure the behavior of this logger:

配置打印的日志级别

org.apache.commons.logging.simplelog.defaultlog - Default logging detail level for all instances of SimpleLog. Must be one of:

trace
debug
info
warn
error
fatal

If not specified, defaults to info.

配置某日志的打印级别

org.apache.commons.logging.simplelog.log.xxxxx - Logging detail level for a SimpleLog instance named "xxxxx". Must be one of:

trace
debug
info
warn
error
fatal

If not specified, the default logging detail level is used.

打印日志的完全类名

org.apache.commons.logging.simplelog.showlogname - Set to true if you want the Log instance name to be included in output messages. Defaults to false.

打印日志的名字

org.apache.commons.logging.simplelog.showShortLogname - Set to true if you want the last component of the name to be included in output messages. Defaults to true.

打印日期及其格式化

org.apache.commons.logging.simplelog.showdatetime - Set to true if you want

the current date and time to be included in output messages. Default is false.

org.apache.commons.logging.simplelog.dateTimeFormat - The date and time format to be used in the output messages. The pattern describing the date and time format is the same that is used in java.text.SimpleDateFormat. If the format is not specified or is invalid, the default format is used. The default format is yyyy/MM/dd HH:mm:ss:SSS zzz.

In addition to looking for system properties with the names specified above, this implementation also checks for a class loader resource named "simplelog.properties", and includes any matching definitions from this resource (if it exists).

Tips：

与 Log4j 类似，SimpleLog 也有一个配置文件，其名为 simplelog.properties，用于上述的配置。

5 JCL Best Practices

Best practices for JCL are presented in two categories: General and Enterprise. The general principles are fairly clear. Enterprise practices are a bit more involved and it is not always as clear as to why they are important.

Enterprise best-practice principles apply to middleware components and tooling that is expected to execute in an "Enterprise" level environment. These issues relate to Logging as Internationalization, and fault detection. Enterprise requires more effort and planning, but are strongly encouraged (if not required) in production level systems. Different corporate enterprises/environments have different requirements, so being flexible always helps.

Tips：

前三个最佳实践在通常情况下都应该坚守，后面是企业实践应该坚持的。

5.1 Code Guards

Code guards are typically used to guard code that only needs to execute in support of logging, that otherwise introduces undesirable runtime overhead in the general case (logging disabled). Examples are multiple parameters, or expressions (e.g. string + " more") for parameters. Use the guard methods of the form log.is<Priority>() to verify that logging should be performed, before incurring the overhead of the logging method call. Yes, the logging methods will perform the same check, but only after resolving parameters.

Tips：

这段的意思是可以在打印日志之前先判断下是否支持该级别的打印。

5.2 日志的级别

Message Priorities/Levels. It is important to ensure that log message are appropriate in content and severity. The following guidelines are suggested:

- fatal - Severe errors that cause premature termination. Expect these to be immediately visible on a status console. See also Internationalization.
- error - Other runtime errors or unexpected conditions. Expect these to be immediately visible on a status console. See also Internationalization.
- warn - Use of deprecated APIs, poor use of API, 'almost' errors, other runtime situations that are undesirable or unexpected, but not necessarily "wrong". Expect these to be immediately visible on a status console. See also Internationalization.
- info - Interesting runtime events (startup/shutdown). Expect these to be immediately visible on a console, so be conservative and keep to a minimum. See also Internationalization.
- debug - detailed information on the flow through the system. Expect these to be written to logs only.
- trace - more detailed information. Expect these to be written to logs only.

5.3 缺省日志级别

Default Message Priority/Level. By default the message priority should be no lower than info. That is, by default debug message should not be seen in the logs.

Tips：
缺省情况下日志最好显示在 info 级别及其以上。

5.4 记录异常

略。

6 JCL 的扩展

JCL is designed to encourage extensions to be created that add functionality. Typically, extensions to JCL fall into two categories:

- new Log implementations that provide new bridges to logging systems
- new LogFactory implementations that provide alternative discovery strategies

参考文献

[1] 官方文档：http://commons.apache.org/proper/commons-logging/guide.html
[2]

第5章 Head First SLF4J

The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks (e.g. java.util.logging, logback, log4j) allowing the end user to plug in the desired logging framework at deployment time.

SLF4J，即简单日志门面（Simple Logging Facade for Java），不是具体的日志解决方案，它只服务于各种各样的日志系统。按照官方的说法，SLF4J 是一个用于日志系统的简单 Facade，允许最终用户在部署其应用时使用其所希望的日志系统。

Slf4j 是对不同日志框架提供的一个门面封装。可以在部署的时候不修改任何配置即可接入一种日志实现方案。和 commons-loging 应该有一样的初衷。个人感觉设从计上更好一些，没有 commons 那么多潜规则。

Tips：

参考和 JCL 的比较。本质上都是为了提供一种日志的接入方案，能够像数据库一样，兼容不同的日志系统。

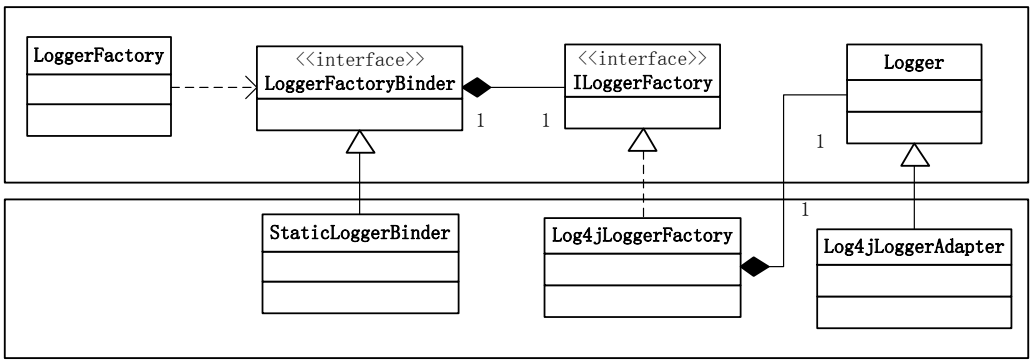
Note that SLF4J-enabling your library implies the addition of only a single mandatory dependency, namely slf4j-api.jar. If no binding is found on the class path, then SLF4J will default to a no-operation implementation.

Tips：

程序中加入 Slf4j，在缺省情况下不会有任何输出。从其他日志迁移到 Slf4j 很简单，还有迁移工具可供使用。

7 适配器实现原理

下图是 Slf4j 的 API 和它的适配器之间的关系。



图：Slf4j 和 Log4j 的适配器

Slf4j 通过 StaticLoggerBinder 唯一性的加载来确认具体的适配器，在这个 StaticLoggerBinder 中来获取对应的日志工厂实现。这里最重要的是，在 Slf4j 中，只有 StaticLoggerBinder 实现 LoggerFactoryBinder 接口，且实现了单例模式，并

且放在固定的目录下，即 org.slf4j.impl 中。因此可以通过类的静态方法直接访问到。

7.1 初始化时的状态变迁

Slf4j 的初始化过程是一个状态变迁的过程，在通过 Slf4j 的接口第一次获取日志的时候 Slf4j 开始启动过程。Slf4j 的具有的状态如下所示：

```
static final int UNINITIALIZED = 0; // 未初始化
static final int ONGOING_INITIALIZATION = 1; // 正在进行初始化
static final int FAILED_INITIALIZATION = 2; // 初始化失败
static final int SUCCESSFUL_INITIALIZATION = 3; // 初始化成功
static final int NOP_FALLBACK_INITIALIZATION = 4; // 没有找到需要的实现，用NOP替代

static int INITIALIZATION_STATE = UNINITIALIZED; // 初始状态为未初始化
```

Slf4j 只会初始化一次，并获取生成日志的工厂。下面是 Slf4j 中 LoggerFactory 类下获取日志工厂的方法。

```
public static ILoggerFactory getILoggerFactory() {
  if (INITIALIZATION_STATE == UNINITIALIZED) {
    INITIALIZATION_STATE = ONGOING_INITIALIZATION;
    performInitialization();
  }
  switch (INITIALIZATION_STATE) {
  case SUCCESSFUL_INITIALIZATION:
    return StaticLoggerBinder.getSingleton().getLoggerFactory();
  case NOP_FALLBACK_INITIALIZATION:
    return NOP_FALLBACK_FACTORY;
  case FAILED_INITIALIZATION:
    throw new IllegalStateException(UNSUCCESSFUL_INIT_MSG);
  case ONGOING_INITIALIZATION:
    // support re-entrant behavior.
    // See also http://bugzilla.slf4j.org/show_bug.cgi?id=106
    return TEMP_FACTORY;
  }
  throw new IllegalStateException("Unreachable code");
}
```

从中可以看出，只有在第一次初始化的时候才会执行初始化动作，并将初始化状态修改为“正在初始化”。再次通过 getLogger()方法进来获取工厂的时候会走到下面 Switch 里，分情况而定：

- 初始化成功，返回单例的日志工厂实现 StaticLoggerBinder。（这里使用

了单例模式和代理模式)；

- 失败情况，抛出异常；
- 正在初始化，使用临时的工厂，并暂存新建的日志；在日志初始化成功后将临时工厂里的日志迁移到真实的实现，并清空临时工厂里的数据；
- 找不到 Binding 的情况，即在类路径下没有找到日志的实现时，使用 NOPLoggerFactory 工厂进行日志的打印，实际上是什么都不输出。

7.2 适配器绑定

上面提到，Slf4j 在 LoggerFactory 中只会初始化一次，其实现方式是通过类加载器来加载对应的适配程序。所有的适配程序都实现了 LoggerFactoryBinder 接口，该接口的作用是帮助 LoggerFactory 类来获取对应 ILoggerFactory 接口的实例实现。

下面是 LoggerFactoryBinder 接口的 API：

```
public interface LoggerFactoryBinder {
    public ILoggerFactory getLoggerFactory();
    public String getLoggerFactoryClassStr();
}
```

所有的适配程序都实现了 LoggerFactoryBinder 接口，且实现的类名及路径都是固定不变的，都位于 org/slf4j/impl/StaticLoggerBinder.class 下，因此在初始化日志工厂的时候,就通过类加载器去找这个实现类,这个实现类可能存在多个。但最终使用哪一个实现是随机的，是由类加载器的加载顺序决定的。当存在多个实现时，类加载器实际上加载了多次，并将最先加载的类给冲掉，因此通过方法 StaticLoggerBinder.getSingleton();只能得到一个 LoggerFactoryBinder 的实现。

Tips：

LoggerFactoryBinder 接口是在 Slf4j-api.jar 的包中，而 StaticLoggerBinder 是在适配器的各自实现中。

比如 Slf4j 至 Log4j 的适配器包，slf4j-log4j12-1.7.21.jar，下面的一个绑定端口实现如下所示：

```
public class StaticLoggerBinder implements LoggerFactoryBinder {
    // 单例模式
    private static final StaticLoggerBinder SINGLETON = new StaticLoggerBinder();
    public static final StaticLoggerBinder getSingleton() {
        return SINGLETON;
    }

    public static String REQUESTED_API_VERSION = "1.6.99"; // !final
    private static final String loggerFactoryClassStr =
Log4jLoggerFactory.class.getName();
```

```
    /**
     * The ILoggerFactory instance returned by the {@link #getLoggerFactory}
     * method should always be the same object
     */
    private final ILoggerFactory loggerFactory;

    private StaticLoggerBinder() {
        loggerFactory = new Log4jLoggerFactory();
        try {
            @SuppressWarnings("unused")
            Level level = Level.TRACE;
        } catch (NoSuchFieldError nsfe) {
            Util.report("This version of SLF4J requires log4j version 1.2.12 or later.
See also http://www.slf4j.org/codes.html#log4j_version");
        }
    }

    public ILoggerFactory getLoggerFactory() {
        return loggerFactory;
    }

    public String getLoggerFactoryClassStr() {
        return loggerFactoryClassStr;
    }
}
```

这个类中提供了一个无参数的构造函数，提供给 JVM 进行加载时的初始化，以绑定真正的日志工厂。通过程序可见，工厂的真正实现是 Log4jLoggerFactory，这个类是适配器中实现了 Slf4j 中日志工厂接口 ILoggerFactory 的实现类。通过该类，可以将 Slf4j 中创建日志的操作代理给这个类，使得该类调用 Log4j 中创建日志方法的类进行日志的创建。

7.3 日志工厂实现

下面是日志工厂的接口。

```
public interface ILoggerFactory {
    public Logger getLogger(String name);
}
```

下面是适配器中实现了日志工厂接口的实现，其源码如下所示。

```
public class Log4jLoggerFactory implements ILoggerFactory {
    // key: name (String), value: a Log4jLoggerAdapter;
```

```
    ConcurrentMap<String, Logger> loggerMap;
    public Log4jLoggerFactory() {
        loggerMap = new ConcurrentHashMap<String, Logger>();
        // force log4j to initialize
        org.apache.log4j.LogManager.getRootLogger();
    }

    public Logger getLogger(String name) {
        Logger slf4jLogger = loggerMap.get(name);
        if (slf4jLogger != null) {
            return slf4jLogger;
        } else {
            org.apache.log4j.Logger log4jLogger;
            if (name.equalsIgnoreCase(Logger.ROOT_LOGGER_NAME))
                log4jLogger = LogManager.getRootLogger();
            else
                log4jLogger = LogManager.getLogger(name);

            Logger newInstance = new Log4jLoggerAdapter(log4jLogger);
            Logger oldInstance = loggerMap.putIfAbsent(name, newInstance);
            return oldInstance == null ? newInstance : oldInstance;
        }
    }
}
```

管中窥豹,在日志工厂中有一个保存日志的 ConcurrentMap,保存的是 Map,其 Key 是日志的名字，Value 是一个日志的适配 Log4jLoggerAdapter。日志的创建过程是:

- 先在这个缓存的 Map 中查找，找到直接返回;
- 否则,通过 Log4j 创建真实的日志,并构建 Slf4j 的日志(包装器模式),缓存在工厂的实现中。

创建一个新的日志，其创建过程进行了两次，一是由真正的工厂类实现日志的创建；二是封装一个 Slf4j 的日志。

7.4 Slf4j 的日志类

下面是 Logger 的接口 API:

```
public interface Logger {
  final public String ROOT_LOGGER_NAME = "ROOT";
  public String getName(); // 获取日志的名字

  public boolean isTraceEnabled(); // 是否支持追踪
```

```
  public void trace(String msg); // 下面是5个打印日志的方法，支持占位符和自适应参数
  public void trace(String format, Object arg);
  public void trace(String format, Object arg1, Object arg2);
  public void trace(String format, Object... arguments);
  public void trace(String msg, Throwable t);

  public boolean isTraceEnabled(Marker marker); // 是否支持Marker追踪
  public void trace(Marker marker, String msg);
  public void trace(Marker marker, String format, Object arg);
  public void trace(Marker marker, String format, Object arg1, Object arg2);
  public void trace(Marker marker, String format, Object... argArray);
  public void trace(Marker marker, String msg, Throwable t);
  // 略 …
}
```

Log4jLoggerAdapter 是适配 Log4j 的一个实现类，该类实现了 Slf4j 的 Logger 接口，并且保存了实际实现日志记录的 Log4j 中的 Logger 实现，可以将所有让 Slf4j 工作的日志经过代理让 Log4j 实现。

```
public final class Log4jLoggerAdapter extends MarkerIgnoringBase implements
LocationAwareLogger, Serializable {
    private static final long serialVersionUID = 6182834493563598289L;
    final transient org.apache.log4j.Logger logger;

    /**
     * Following the pattern discussed in pages 162 through 168 of "The complete
     * log4j manual".
     */
    final static String FQCN = Log4jLoggerAdapter.class.getName();

    // Does the log4j version in use recognize the TRACE level?
    // The trace level was introduced in log4j 1.2.12.
    final boolean traceCapable;

    // WARN: Log4jLoggerAdapter constructor should have only package access so
    // that
    // only Log4jLoggerFactory be able to create one.
    Log4jLoggerAdapter(org.apache.log4j.Logger logger) {
        this.logger = logger;
        this.name = logger.getName();
        traceCapable = isTraceCapable();
    }
    // 略 … …
```

```
}
```

8 桥接器的实现原理

8.1 什么是桥接器

桥接模式解决的问题是将日志进行统一的输出，即将其他系统中的日志，通过 Slf4j 的桥接器转到一个统一的日志输出系统。起到的是日志系统之间的相互连接。

下面以 JCL 到 Slf4j 的桥接实现为例，简单看下是怎么实现的。

8.2 实现 Log 接口

桥接器的实现实际上是对 JCL 的扩展，需要实现 JCL 的 Log 接口。如下所示，是 Slf4j 的接口实现。

```
public class SLF4JLog implements Log, Serializable {

    private static final long serialVersionUID = 6807286170111672O9L;

    // used to store this logger's name to recreate it after serialization
    protected String name;

    // in both Log4jLogger and Jdk14Logger classes in the original JCL, the
    // logger instance is transient
    private transient Logger logger;

    SLF4JLog(Logger logger) {
        this.logger = logger;
        this.name = logger.getName();
    }

    // 略…
}
```

8.3 实现日志工厂

首先看下 Slf4j 对 JCL 日志工厂的实现。

```
@SuppressWarnings("rawtypes")
public class SLF4JLogFactory extends LogFactory {
    /**
     * The {@link org.apache.commons.logging.Log}instances that have already been
     * created, keyed by logger name.
     */
```

```
    ConcurrentMap<String, Log> loggerMap;
    public SLF4JLogFactory() {
        loggerMap = new ConcurrentHashMap<String, Log>();
    }
    public static final String LOG_PROPERTY = "org.apache.commons.logging.Log";
    protected Hashtable attributes = new Hashtable();
    // 略 … …

    // 通过日志工厂获取日志的实例
    public Log getInstance(String name) throws LogConfigurationException {
        Log instance = loggerMap.get(name);
        if (instance != null) {
            return instance;
        } else {
            Log newInstance;
            Logger slf4jLogger = LoggerFactory.getLogger(name);
            if (slf4jLogger instanceof LocationAwareLogger) {
                newInstance = new SLF4JLocationAwareLog((LocationAwareLogger)
slf4jLogger);
            } else {
                newInstance = new SLF4JLog(slf4jLogger);
            }
            Log oldInstance = loggerMap.putIfAbsent(name, newInstance);
            return oldInstance == null ? newInstance : oldInstance;
        }
    }
}
```

参考 JCL 的扩展部分，通过配置文件可以指定日志的工厂实现和日志类的具体实现，进而通过 JCL 的动态绑定机制加载对应的工厂实现来创建日志。同时通过 loggerMap 缓存真正的日志实现，即建立起 Slf4j 日志和 JCL 日志之间的一一对应的关系。

参考文献

[1] 官网：https://www.slf4j.org/

[2] 官网手册：https://www.slf4j.org/docs.html

[3]