Java日志系统框架的设计与实现

在Java 领域，存在大量的日志组件，open-open收录了21个日志组件。日志系统作为一种应用程序服务，对于跟踪调试、程序状态记录、崩溃数据恢复都有着重要的作用，我们可以把Java日志系统看作是必不可少的跟踪调试工具。

1.简介

日志系统是一种不可或缺的跟踪调试工具,特别是在任何无人职守的后台程序以及那些没有跟踪调试环境的系统中有着广泛的应用。长期以来，日志系统作为一种应用程序服务,对于跟踪调试、程序状态记录、崩溃数据恢复都有非常现实的意义。这种服务通常以两种方式存在：

1.日志系统作为服务进程存在。Windows中的的事件日志服务就属于这种类型，该类型的日志系统通常通过消息队列机制将所需要记录的日志由日志发送端发送给日志服务。日志发送端和日志保存端通常不在同一进程当中，日志的发送是异步过程。这种日志服务通常用于管理员监控各种系统服务的状态。

2.日志系统作为系统调用存在。Java世界中的日志系统和Unix环境下诸多守护进程所使用的日志系统都属于这种类型。日志系统的代码作为系统调用被编译进日志发送端，日志系统的运行和业务代码的运行在同一进程空间。日志的发送多数属于同步过程。这种日志服务由于能够同步反映处系统运行状态，通常用于调试跟踪和崩溃恢复。

本文建立的日志系统基本属于第二种类型，但又有所不同。该日志系统将利用Java线程技术实现一个既能够反映统一线程空间中程序运行状态的同步日志发送过程,又能够提供快速的日志记录服务，还能够提供灵活的日志格式配置和过滤机制。

1.1系统调试的误区

在控制台环境上调试Java程序时,此时往控制台或者文本文件输出一段文字是查看程序运行状态最简单的做法，但这种方式并不能解决全部的问题。有时候，对于一个我们无法实时查看系统输出的系统或者一个确实需要保留我们输出信息的系统,良好的日志系统显得相当必要。因此，不能随意的输出各种不规范的调试信息，这些随意输出的信息是不可控的,难以清除，可能为后台监控、错误排除和错误恢复带来相当大的阻力。

1.2日志系统框架的基本功能

一个完备的日志系统框架通常应当包括如下基本特性：

所输出的日志拥有自己的分类：这样在调试时便于针对不同系统的不同模块进行查询,从而快速定位到发生日志事件的代码。

日志按照某种标准分成不同级别：分级以后的日志，可以用于同一分类下的日志筛选。

支持多线程：日志系统通常会在多线程环境中使用，特别是在Java系统当中，因此作为一种系统资源，日志系统应当保证是线程安全的。

支持不同的记录媒介：不同的工程项目往往对日志系统的记录媒介要求不同，因此日志系统必须提供必要的开发接口，以保证能够比较容易的更换记录介质。

高性能：日志系统通常要提供高速的日志记录功能以应对大系统下大请求流量下系统的正常运转。

稳定性：日志系统必须是保持高度的稳定性，不能因为日志系统内部错误导致主要业务代码的崩溃。

1.3常用日志系统简介

在Java世界中，以下三种日志框架比较优秀：

1）Log4J

最早的Java日志框架之一，由Apache基金会发起，提供灵活而强大的日志记录机制。但是其复杂的配置过程和内部概念往往令使用者望而却步。

2）JDK1.4LoggingFramework

继Log4J之后，JDK标准委员会将Log4J的基本思想吸收到JDK当中，在JDK1.4中发布了第一个日志框架接口，并提供了一个简单实现。

3）CommonsLoggingFramwork

该框架同样是Apache基金会项目，其出现主要是为了使得Java项目能够在Log4J和JDK1.4lLoggingFramework的使用上随意进行切换，因此该框架提供了统一的调用接口和配置方法。

2.系统设计

由于Log4J得到广泛应用，从使用者的角度考虑，本文所设计的框架，采用了部分Log4J的接口和概念，但内部实现则完全不同。使用Java实现日志框架，关键的技术在于前面提及的日志框架特性的内部实现，特别是：日志的分类和级别、日志分发框架的设计、日志记录器的设计以及在设计中的高性能和高稳定性的考虑。

2.1系统架构

日志系统框架可以分为日志记录模块和日志输出模块两大部分。日志记录模块负责创建和管理日志记录器（Logger），每一个Logger对象负责按照不同的级别（LoggerLevel）接收各种记录了日志信息的日志对象（LogItem），Logger对象首先获取所有需要记录的日志，并且同步地将日志分派给日志输出模块。日志输出模块则负责日志输出器（Appender）的创建和管理，以及日志的输出。系统中允许有多个不同的日志输出器，日志输出器负责将日志记录到存储介质当中。

日志记录器Logger是整个日志系统框架的用户使用接口，程序员可以通过该接口记录日志，为了实现对日志进行分类，系统设计允许存在多个Logger对象，每一个Logger负责一类日志的记录，Logger类同时实现了对其对象本身的管理。LoggerLevel类定义了整个日志系统的级别，在客户端创建和发送日志时，这些级别会被使用到。Logger对象在接收到客户端创建和发送的日志消息时，同时将该日志消息包装成日志系统内部所使用的日志对象LogItem，日志对象除了发送端所发送的消息以外，还会包装诸如发送端类名、发送事件、发送方法名、发送行号等等。这些额外的消息对于系统的跟踪和调试都非常有价值。包装好的LogItem最终被发送给输出器，由这些输出器负责将日志信息写入最终媒介，输出器的类型和个数均不固定，所有的输出器通过AppenderManager进行管理，通常通过配置文件即可方便扩展出多个输出器。

2.2日志记录部分的设计

如前文所述，日志记录部分负责接收日志系统客户端发送来的日志消息、日志对象的管理等工作。下面详细描述了日志记录部分的设计要点：

1.日志记录器的管理

系统通过保持多个Logger对象的方式来进行日志记录的分类。每一个Logger对象代表一类日志分类。因此，Logger对象的名称属性是其唯一标识，通过名称属性获取一个Logger对象：

1.LoggerLoggerlogger=Logger.getLogger（“LoggerName”）;

一般的，使用类名来作为日志记录器的名称，这样做的好处在于能够尽量减少日志记录器命名之间的冲突（因为Java类使用包名），同时能够将日志记录分类得尽可能的精细。因此，假定有一UserManager类需要使用日志服务，则更一般的使用方式为：

2.LoggerLoggerlogger=Logger.getLogger（UserManager.class）;

2.日志分级的实现

按照日志目的不同，将日志的级别由低到高分成五个级别：

◆DEBUG-表示输出的日志为一个调试信息；

◆INFO-表示输出的日志是一个系统提示；

◆WARN-表示输出的日志是一个警告信息；

◆ERROR-表示输出的日志是一个系统错误；

◆FATAL-表示输出的日志是一个导致系统崩溃严重错误。

这些日志级别定义在LoggerLevel接口中，被日志记录器Logger在内部使用。而对于日志系统客户端则可使用Logger类接口对直接调用并输出这些级别的日志,Logger的这些接口描述如下：

3.publicvoiddebug（Stringmsg）；//输出调试信息

4.publicvoidinfo（Stringmsg）；//输出系统提示

5.publicvoidwarn（Stringmsg）；//输出警告信息

6.publicvoidfatal（Stringmsg）；//输出系统错误

7.publicvoiderror（Stringmsg）；//输出严重错误

通过对Logger对象上这些接口的调用，直接为日志信息赋予了级别属性，这样为后继的按照不同级别进行输出的工作奠定了基础。

3.日志对象信息的获取

日志对象上包含了一条日志所具备的所有信息。通常这些信息包括：输出日志的时间、Java类、类成员方法、所在行号、日志体、日志级别等等。在JDK1.4中可以通过在方法中抛出并且捕获住一个异常,则在捕捉到的异常对象中已经由JVM自动填充好了系统调用的堆栈,在JDK1.4中则可以使用java.lang.StackTraceElement获取到每一个堆栈项的基本信息，通过对日志客户端输出日志方法调用层数的推算，则可以比较容易的获取到StackTraceElement对象，从而获取到输出日志时的Java类、类成员方法、所在行号等信息。在JDK1.3或者更早的版本中，相应的工作则必须通过将异常的堆栈信息输出到字符串中,并分析该字符串格式得到。

2.3日志输出部分的设计

日志输出部分的设计具有一定的难度，在本文设计的日志系统中，日志的输出、多线程的支持、日志系统的扩展性、日志系统的效率等问题都交由日志输出部分进行管理。

1.日志输出器的继承结构

在日志的输出部分采用了二层结构，即定义了一个抽象的日志输出器（AbstractLoggerAppender），然后从该抽象类继承出实际的日志输出器。

AbstractLoggerAppender定义了一系列的对日志进行过滤的方法，而具体输出到存储媒介的方法则是一个抽象方法，由子类实现。在系统中默认实现了控制台输出器和文件输出器两种，其中控制台输出器的实现颇为简单。

2.文件输出器的内部实现

在日志记录部分的实现中，并没有考虑多线程、高效率等问题，因此文件输出器必须考虑这些问题的处理。在文件输出器内部使用java.lang.Vector定义了一个线程安全的高速缓冲，所有通过日志记录部分分派到文件输出器的日志被直接放置到该高速缓冲当中。同时在文件输出器内部定义一个工作线程，负责定期将高速缓冲中的内容保存到文件，在保存的过程中同时可以进行日志文件的备份等工作。由于采用了高速缓冲的结构，很显然日志客户端的调用已经不再是一个同步调用，从而不再会需要等到文件操作后才返回，提高的系统调用的速度。

2.4设计难点

通过上述设计，一个具有良好扩展能力的高性能日志系统框架就已经具有了一定的雏形。在设计过程中几个难点问题需要进一步反思。

一、是否整个系统应当采用完全异步的结构，通过类似于消息机制的方式来进行由日志客户端发送日志给日志系统。这种方式可以作为日志系统框架另一种运行方式，在后继设计中加以考虑。

二、在文件输出器中可以看到，目前虽然可以扩展多个日志输出器，但是目前提供的抽象类中仅仅提供了对日志的过滤机制，而没有提供的缓存机制，目前的缓存机制被放在文件输出器中实现，因此在未来的进一步设计中，可以将文件输出器中的缓存机制上移到抽象类当中。

2.5设计模式

在设计过程中我们特别注意使用了数个经典的设计模式。如：Logger对象的创建使用了工厂方法模式（FactoryMethod）、由AbstractLoggerAppender和ConsoleAppender以及FileAppender构成了策略模式(Strategy)，除此以外，还大量使用了单例模式(Singleton)。在设计中适当运用设计模式能够加快设计进度、提高设计质量。

3.总结

本文探讨了日志系统的基本特性、实现日志系统的意义、方法和内部结构，并且给出了一种基于Java平台的日志系统的详细设计。同时也指出日志系统会向服务化、异步化的方向发展。作为一种方便的跟踪调试、数据恢复工具，应当提倡在适当的环境下对日志系统的使用。