无论从设计上还是实现上，Logback相对log4j而言有了相对多的改进。不过尽管难以一一细数，这里还是列举部分理由为什么选择logback而不是log4j。牢记logback与log4j在概念上面是很相似的，它们都是有同一群开发者建立。所以如果你已经对log4j很熟悉，你也可以很快上手logback。如果你喜欢使用log4j,你也许会迷上使用logback。

**1. 更快的执行速度**

基于我们先前在log4j上的工作，logback 重写了内部的实现，在某些特定的场景上面，甚至可以比之前的速度快上10倍。在保证logback的组件更加快速的同时，同时所需的内存更加少。

**2.更充足的测试**

Logback 历经了几年，数不清小时数的测试。尽管log4j也是测试过的，但是Logback的测试更加充分，跟log4j不在同一个级别。我们认为，这正是人们选择Logback而不是log4j的最重要的原因。人们都希望即使在恶劣的条件下，你的日记框架依然稳定而可靠。

**3. logback-classic 非常自然的实现了SLF4J**

logback-classic中的登陆类自然的实现了SLF4J。当你使用 logback-classic作为底层实现时，涉及到LF4J日记系统的问题你完全不需要考虑。更进一步来说，由于 logback-classic强烈建议使用SLF4J作为客户端日记系统实现，如果需要切换到log4j或者其他，你只需要替换一个jar包即可，不需要去改变那些通过SLF4J API 实现的代码。这可以大大减少更换日记系统的工作量。

**4. 扩展文档**

Logback附带详细的和不断更新的文档。

**5.使用XML配置文件或者Groovy**

配置logback的传统方法是通过XML文件。在文档中，大部分例子都是是用XML语法。但是，对于logback版本0.9.22，通过Groovy编写的配置文件也得到支持。相比于XML，Groovy风格的配置文件更加直观，连贯和简短的语法。  
现在， 已经有一个工具自动把logback.xml文件迁移至logback.groovy。

**6.自动重新载入配置文件**

Logback-classic可以在配置文件被修改后，自动重新载入。这个扫描过程很快，无资源争用，并且可以动态扩展支持在上百个线程之间每秒上百万个调用。它和应用服务器结合良好，并且在JEE环境通用，因为它不会调用创建一个单独的线程来做扫描。

**7.优雅地从I/O错误中恢复**

FileAppender和它的子类，包括RollingFileAppender，可以优雅的从I/O错误中恢复。所以，如果一个文件服务器临时宕机，你再也不需要重启你的应用，而日志功能就能正常工作。当文件服务器恢复工作，logback相关的appender就会透明地和快速的从上一个错误中恢复。

**8.自动清除旧的日志归档文件**

通过设置TimeBasedRollingPolicy 或者 SizeAndTimeBasedFNATP的 maxHistory 属性，你就可以控制日志归档文件的最大数量。如果你的回滚策略是每月回滚的，并且你希望保存一年的日志，那么只需简单的设置maxHistory属性为12。对于12个月之前的归档日志文件将被自动清除。

**9. 自动压缩归档日志文件**

[RollingFileAppender](http://logback.qos.ch/manual/appenders.html#RollingFileAppender)可以在回滚操作中，自动压缩归档日志文件。压缩通常是异步执行的，所以即使是很大的日志文件，你的应用都不会因此而被阻塞。

**10. 谨慎模式**

在谨慎模式中，在多个JVM中运行的多个FileAppender实例，可以安全的写入统一个日志文件。谨慎模式可以在一定的限制条件下应用于RollingFileAppender。

**11.Lilith**

Lilith是logback的一个记录和访问事件查看器。它相当于log4j的 chainsaw，但是Lilith设计的目的是处理大量的日志记录。

**12. 配置文件中的条件处理**

开发者通常需要在不同的目标环境中变换logback的配置文件，例如开发环境，测试环境和生产环境。这些配置文件大体是一样的，除了某部分会有不同。为了避免重复，logback支持[配置文件中的条件处理](http://logback.qos.ch/manual/configuration.html#conditional)，只需使用<if>,<then>和<else>，那么同一个配置文件就可以在不同的环境中使用了。

**13.过滤**

Logback拥有远比log4j更丰富的过滤能力。例如，让我们假设，有一个相当重要的商业应用部署在生产环境。考虑到大量的交易数据需要处理，记录级别被设置为WARN，那么只有警告和错误信息才会被记录。现在，想象一下，你在开发环境遇到了一个臭虫，但是在测试平台中却很难发现，因为一些环境之间(生产环境/测试环境)的未知差异。

使用log4j，你只能选择在生产系统中降低记录的级别到DEBUG，来尝试发现问题。但是很不幸，这会生成大量的日志记录，让分析变得困难。更重要的是，多余的日志记录会影响到生产环境的性能。  
使用logback，你可以选择保留只所有用户的WARN级别的日志，而除了某个用户，例如Alice，而她就是问题的相关用户。当Alice登录系统，她就会以DEBUG级别被记录，而其他用户仍然是以WARN级别来记录日志。这个功能，可以通过在配置文件的XML中添加4行。请在相关章节中查找MDCFilter

**14. SiftingAppender**

SiftingAppender是一个全能的追加器。它可以基于任何给定的实时属性分开（或者筛选）日志。例如，SiftingAppender可以基于用户会话分开日志事件，这样，可以为每一个用户建立一个独立的日志文件。

以上网络整理的内容

下面是我编写的一个demo配置

<configuration scan=*"false"* debug=*"false"*>

<appender name=*"STDOUT"* class=*"ch.qos.logback.core.ConsoleAppender"*>

<encoder>

<pattern>%d{mm:ss} %-5level %logger{36} >>> %msg%n</pattern>

</encoder>

</appender>

<appender name=*"DEBUG\_FILE"* class=*"ch.qos.logback.core.rolling.RollingFileAppender"*>

<filter class=*"ch.qos.logback.classic.filter.LevelFilter"*>

<level>DEBUG</level>

<onMatch>ACCEPT</onMatch>

<onMismatch>DENY</onMismatch>

</filter>

<rollingPolicy class=*"ch.qos.logback.core.rolling.TimeBasedRollingPolicy"*>

<!-- daily rollover -->

<fileNamePattern>logs/debug.%d{yyyy-MM-dd}.log</fileNamePattern>

<!-- keep 30 days' worth of history capped at 3GB total size -->

<maxHistory>30</maxHistory>

<totalSizeCap>3GB</totalSizeCap>

</rollingPolicy>

<encoder>

<pattern>%d [%thread] [%-5level] %file,%line - %msg%n</pattern>

</encoder>

</appender>

<appender name=*"WARN\_FILE"* class=*"ch.qos.logback.core.rolling.RollingFileAppender"*>

<filter class=*"ch.qos.logback.classic.filter.ThresholdFilter"*>

<level>WARN</level>

</filter>

<rollingPolicy class=*"ch.qos.logback.core.rolling.TimeBasedRollingPolicy"*>

<!-- daily rollover -->

<fileNamePattern>logs/warn.%d{yyyy-MM-dd}.log</fileNamePattern>

<!-- keep 30 days' worth of history capped at 3GB total size -->

<maxHistory>30</maxHistory>

<totalSizeCap>3GB</totalSizeCap>

</rollingPolicy>

<encoder>

<pattern>%d [%thread] [%-5level] %file,%line - %msg%n</pattern>

</encoder>

</appender>

<root level=*"DEBUG"*>

<appender-ref ref=*"DEBUG\_FILE"* />

<appender-ref ref=*"WARN\_FILE"* />

<appender-ref ref=*"STDOUT"* />

</root>

</configuration>

**public** **class** TestA {

**static** Logger *log* = LoggerFactory.*getLogger*(TestB.**class**);

**public** **static** **void** main(String[] args) {

*log*.trace("======trace");

*log*.debug("======debug");

*log*.info("======info");

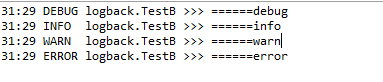
*log*.warn("======warn");

*log*.error("======error");

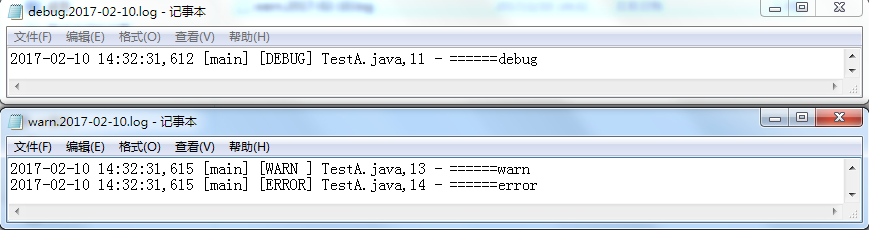
}

}

输出结果:



日志记录情况:



demo每天一个新日志文件,历史数据保留最大30天.日志总占数据量大于3G自动异步删除旧日志

根节点<configuration>包含的属性：

scan:

当此属性设置为true时，配置文件如果发生改变，将会被重新加载，默认值为true。

scanPeriod:

设置监测配置文件是否有修改的时间间隔，如果没有给出时间单位，默认单位是毫秒。当scan为true时，此属性生效。默认的时间间隔为1分钟。

debug:

当此属性设置为true时，将打印出logback内部日志信息，实时查看logback运行状态。默认值为false。

<level>:设置过滤级别

<onMatch>:用于配置符合过滤条件的操作

<onMismatch>:用于配置不符合过滤条件的操作

Logback的过滤器基于三值逻辑（ternary logic），允许把它们组装或成链，从而组成任意的复合过滤策略。过滤器很大程度上受到Linux的iptables启发。这里的所谓三值逻辑是说，过滤器的返回值只能是ACCEPT、DENY和NEUTRAL的其中一个。

如果返回DENY，那么记录事件立即被抛弃，不再经过剩余过滤器；

如果返回NEUTRAL，那么有序列表里的下一个过滤器会接着处理记录事件；

如果返回ACCEPT，那么记录事件被立即处理，不再经过剩余过滤器。

独立的日志标签

**<logger** name="monitor" additivity="false" level="INFO"**>**

这里通过设置**additivity="false"**禁止monitor里的内容向上传递，否则会同时显示在默认的日志中。