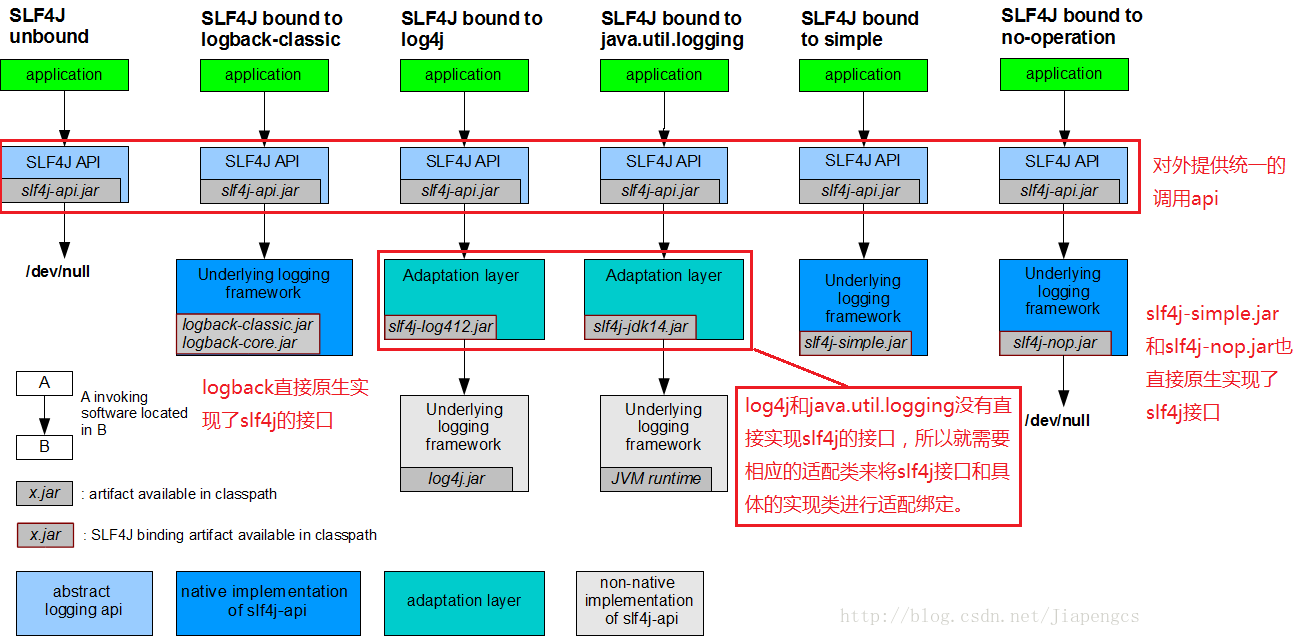
# SLF4J

什么是SLF4J?

意思为简单日志门面，它是把不同的日志系统的实现进行了具体的抽象化，只提供了统一的日志使用接口，使用时只需要按照其提供的接口方法进行调用即可，由于它只是一个接口，并不是一个具体的可以直接单独使用的日志框架，所以最终日志的格式、记录级别、输出方式等都要通过接口绑定的具体的日志系统来实现，这些具体的日志系统就有log4j,logback,java.util.logging等，它们才实现了具体的日志系统的功能。

如何使用SLF4J?

既然SLF4J只是一个接口，那么实际使用时必须要结合具体的日志系统来使用，我们首先来看SLF4J和各个具体的日志系统进行绑定时的框架原理图：



其实slf4j原理很简单，他只提供一个核心slf4j api(就是slf4j-api.jar包)，这个包只有日志的接口，并没有实现，所以如果要使用就得再给它提供一个实现了些接口的日志包，比 如：log4j,common logging,jdk log日志实现包等，但是这些日志实现又不能通过接口直接调用，实现上他们根本就和slf4j-api不一致，因此slf4j又增加了一层来转换各日志实现包的使用

slf4j-api(接口层)

|

各日志实现包的连接层( slf4j-jdk14, slf4j-log4j)

|

各日志实现包

为什么要使用SLF4J?

1、slf4j是一个日志接口，自己没有具体实现日志系统，只提供了一组标准的调用api,这样将调用和具体的日志实现分离，使用slf4j后有利于根据自己实际的需求更换具体的日志系统，比如:之前使用的具体的日志系统为log4j,想更换为logback时，只需要删除log4j相关的jar,然后加入logback相关的jar和日志配置文件即可，而不需要改动具体的日志输出方法，试想如果没有采用这种方式，当你的系统中日志输出有成千上万条时，你要更换日志系统将是多么庞大的一项工程。如果你开发的是一个面向公众使用的组件或公共服务模块，那么一定要使用slf4的这种形式，这有利于别人在调用你的模块时保持和他系统中使用统一的日志输出。

2、slf4j日志输出时可以使用{}占位符，如，logger.info("testlog: {}", "test")，而如果只使用log4j做日志输出时，只能以logger.info("testlog:"+"test")这种形式，前者要比后者在性能上更好，后者采用+连接字符串时就是new 一个String 字符串，在性能上就不如前者。

如果系统中之前已经使用了log4j做日志输出，想使用slf4j作为统一的日志输出，该怎么办呢？

如果之前系统中是单独使用log4j做为日志输出的，这时再想使用slf4j做为日志输出时，如果系统中日志比较多，此时更改日志输出方法肯定是不太现实的，这个时候就可以使用log4j-over-slf4j.jar将使用log4j日志框架输出的日志路由到slf4j上来统一采用slf4j来输出日志。

结合各日志实现包使用时提供的jar包情况为：

SLF4J和logback结合使用时需要提供jar:slf4j-api.jar,logback-classic.jar,logback-core.jar

SLF4J和log4j结合使用时需要提供jar:slf4j-api.jar,slf4j-log4j12.jar,log4j.jar

SLF4J和JDK中java.util.logging结合使用时需要提供jar:slf4j-api.jar,slf4j-jdk14.jar

SLF4J和simple(SLF4J本身提供的一个接口的简单实现)结合使用时需要提供的jar:slf4j-api.jar,slf4j-simple.jar

当然还有其他的日志实现包，以上是经常会使用到的一些。

**总结如下：**

1、slf4j是java的一个日志门面，实现了日志框架一些通用的api，log4j和logback是具体的日志框架。

2、他们可以单独的使用，也可以绑定slf4j一起使用。

单独使用，分别调用框架自己的方法来输出日志信息。绑定slf4j一起使用。调用slf4j的api来输入日志信息，具体使用与底层日志框架无关（需要底层框架的配置文件）。显然不推荐单独使用日志框架。假设项目中已经使用了log4j，而我们此时加载了一个类库，而这个类库依赖另一个日志框架。这个时候我们就需要维护两个日志框架，这是一个非常麻烦的事情。而使用了slf4j就不同了，由于应用调用的抽象层的api，与底层日志框架是无关的，因此可以任意更换日志框架

# Log4j

## Log4j基本使用方法

Log4j由三个重要的组件构成：日志信息的优先级，日志信息的输出目的地，日志信息的输出格式。日志信息的优先级从高到低有ERROR、WARN、 INFO、DEBUG，分别用来指定这条日志信息的重要程度；日志信息的输出目的地指定了日志将打印到控制台还是文件中；而输出格式则控制了日志信息的显 示内容。

定义配置文件

其实您也可以完全不使用配置文件，而是在代码中配置Log4j环境。但是，使用配置文件将使您的应用程序更加灵活。Log4j支持两种配置文件格式，一种是XML格式的文件，一种是Java特性文件（键=值）。下面我们介绍使用Java特性文件做为配置文件的方法：

1.配置根Logger，其语法为：

log4j.rootLogger = [ level ] , appenderName, appenderName, …

其中，level 是日志记录的优先级，分为OFF、FATAL、ERROR、WARN、INFO、DEBUG、ALL或者您定义的级别。Log4j建议只使用四个级别，优 先级从高到低分别是ERROR、WARN、INFO、DEBUG。通过在这里定义的级别，您可以控制到应用程序中相应级别的日志信息的开关。比如在这里定 义了INFO级别，则应用程序中所有DEBUG级别的日志信息将不被打印出来。 appenderName就是指B日志信息输出到哪个地方。您可以同时指定多个输出目的地。

2.配置日志信息输出目的地Appender，其语法为：

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

log4j.appender.appenderName.option1 = value1

…

log4j.appender.appenderName.option = valueN

其中，Log4j提供的appender有以下几种：

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

org.apache.log4j.FileAppender（文件），

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

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

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

3.配置日志信息的格式（布局），其语法为：

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

log4j.appender.appenderName.layout.option1 = value1

…

log4j.appender.appenderName.layout.option = valueN

其中，Log4j提供的layout有以e几种：

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

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

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

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

Log4J采用类似C语言中的printf函数的打印格式格式化日志信息，打印参数如下：

%m 输出代码中指定的消息

%p 输出优先级，即DEBUG，INFO，WARN，ERROR，FATAL

%r 输出自应用启动到输出该log信息耗费的毫秒数

%c 输出所属的类目，通常就是所在类的全名

%t 输出产生该日志事件的线程名

%n 输出一个回车换行符，Windows平台为“rn”，Unix平台为“n”

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

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

## 使用步骤

第一步 导包

1. 配合slf4j-api使用：因没有直接继承slf4j标准接口需slf4j-log4j12.jar(中间层),slf4j-api.jar ,log4j.jar(核心)

maven导包 以下配置就有上面三个包

<!-- log4j导包和slf4j接口整合 -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.25</version>

</dependency>

2、单独使用：log4j-1.2.17.jar (核心)、commons-logging-1.2.jar 这就是为什么要导入此包，spring整合了日志功能

第二步 复制log4j的配置文件：log4j.properties，复制到src下面

log4j.properties内容

### 设置###

log4j.rootLogger = debug,stdout,D,E 可以设置info和debug

### 输出信息到控制抬 ###

log4j.appender.stdout = org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target = System.out

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

log4j.appender.stdout.layout.ConversionPattern = [%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} method:%l%n%m%n

### 输出DEBUG 级别以上的日志到=E://logs/error.log ###

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

log4j.appender.D.File = E://logs/log.log //log文件的路径设置

log4j.appender.D.Append = true

log4j.appender.D.Threshold = DEBUG

log4j.appender.D.layout = org.apache.log4j.PatternLayout （可以灵活地指定布局模式）

log4j.appender.D.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss} [ %t:%r ] - [ %p ] %m%n

### 输出ERROR 级别以上的日志到=E://logs/error.log ###

log4j.appender.E = org.apache.log4j.DailyRollingFileAppender

log4j.appender.E.File =E://logs/error.log

log4j.appender.E.Append = true

log4j.appender.E.Threshold = ERROR

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

log4j.appender.E.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss} [ %t:%r ] - [ %p ] %m%n

第三步 代码的实现

@Controller

public class UsersController {

private static Logger logger = Logger.getLogger(UsersController.class);

@RequestMapping("/test")

@ResponseBody

public Map<String, Object> test() {

List<String> list=new ArrayList<String>();

Map<String, Object> map=new HashMap<String, Object>();

list.add("色");

list.add("色1");

map.put("total", list.size()+"条");

map.put("详细内容如下：", list);

logger.error("出错了把");

logger.debug("不好用啊");

return map;

}

}

### 注意问题

注意：网上看到也可以配置监听类Log4jConfigListener此类在spring-web的jar包util中(已经过时)

发现web.xml中Log4jConfigListener类报红,由于这个类是在spring-web-util下,在spring4.3.5中 这个类已经过时的了, 后来我将版本改成了spring5.0.0后,这个类就被删除了,

# logback

**logback的介绍**

logback同样是由log4j的作者设计完成的，拥有更好的特性，用来取代log4j的一个日志框架,是slf4j的原生实现(即直接实现了slf4j的接口，而log4j并没有直接实现，所以就需要一个适配器slf4j-log4j12.jar),logback一共有以下几个模块：

logback-core：其它两个模块的基础模块

logback-classic：它是log4j的一个改良版本，同时它完整实现slf4j API使你可很方便更换成其它日志系统如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

同样，单独使用它时，需要引入以上jar,然后进行配置文件的配置，最后就是在相关类中进行使用，使用时加入以下语句:

private final static Logger logger = LoggerFactory.getLogger(Test.class);

logger.info("打印日志");

logback取代log4j的理由：

1、更快的实现：Logback的内核重写了，在一些关键执行路径上性能提升10倍以上。而且logback不仅性能提升了，初始化内存加载也更小了。

2、非常充分的测试：Logback经过了几年，数不清小时的测试。Logback的测试完全不同级别的。

3、Logback-classic非常自然实现了SLF4j：Logback-classic实现SLF4j。在使用SLF4j中，你都感觉不到logback-classic。而且因为logback-classic非常自然地实现了slf4j ， 所以切换到log4j或者其他非常容易，只需要提供成另一个jar包就OK，根本不需要去动那些通过SLF4JAPI实现的代码。

4、非常充分的文档 官方网站有两百多页的文档。

5、自动重新加载配置文件，当配置文件修改了，Logback-classic能自动重新加载配置文件。扫描过程快且安全，它并不需要另外创建一个扫描线程。这个技术充分保证了应用程序能跑得很欢在JEE环境里面。

6、LIlith是log事件的观察者，和log4j的chainsaw类似。而lilith还能处理大数量的log数据 。

7、谨慎的模式和非常友好的恢复，在谨慎模式下，多个FileAppender实例跑在多个JVM下，能 够安全地写道同一个日志文件。RollingFileAppender会有些限制。Logback的FileAppender和它的子类包括 RollingFileAppender能够非常友好地从I/O异常中恢复。

8、配置文件可以处理不同的情况，开发人员经常需要判断不同的Logback配置文件在不同的环境下（开发，测试，生产）。而这些配置文件仅仅只有一些很小的不同，可以通过,和来实现，这样一个配置文件就可以适应多个环境。

9、Filters（过滤器）有些时候，需要诊断一个问题，需要打出日志。在log4j，只有降低日志级别，不过这样会打出大量的日志，会影响应用性能。在Logback，你可以继续 保持那个日志级别而除掉某种特殊情况，如alice这个用户登录，她的日志将打在DEBUG级别而其他用户可以继续打在WARN级别。要实现这个功能只需加4行XML配置。可以参考MDCFIlter 。

10、SiftingAppender（一个非常多功能的Appender）：它可以用来分割日志文件根据任何一个给定的运行参数。如，SiftingAppender能够区别日志事件跟进用户的Session，然后每个用户会有一个日志文件。

11、自动压缩已经打出来的log：RollingFileAppender在产生新文件的时候，会自动压缩已经打出来的日志文件。压缩是个异步过程，所以甚至对于大的日志文件，在压缩过程中应用不会受任何影响。

12、堆栈树带有包版本：Logback在打出堆栈树日志时，会带上包的数据。

13、自动去除旧的日志文件：通过设置TimeBasedRollingPolicy或者SizeAndTimeBasedFNATP的maxHistory属性，你可以控制已经产生日志文件的最大数量。如果设置maxHistory 12，那那些log文件超过12个月的都会被自动移除。

## 使用步骤

第一步 导包

三个包：slf4j-api-1.7.25.jar,logback-classic-1.2.3.jar,logback-core-1.2.3.jar

maven 导入以下就有以上三个包：已经整合

<!-- logback 日志文件 实现slf4j接口并整合-->

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.3</version>

</dependency>

第二步 logback.xml配置

直接放到src下也就是classes根目录，自动加载

logback的默认配置加载：

1尝试在 classpath下查找文件logback-test.xml；

2如果文件不存在，则查找文件logback.xml；

3如果两个文件都不存在，logback用BasicConfigurator自动对自己进行配置，这会导致记录输出到控制台。

<?xml version="1.0" encoding="UTF-8"?>

<!-- scan:配置文件发生改变会重新加载 (默认就是true)-->

<!-- scanPeriod:60秒检查此配置文件有没有发生变化-->

<!-- debug:设置为true，打印内部logback的debug信息，它本身就很稳定，很明显不需要-->

<configuration scan="true" scanPeriod="60 seconds" debug="false">

第一步：定义参数变量：此步骤可以省略：用于下面几步的值得方便设置

<!-- 定义参数常量：用来配置下面的值：用此${}加载 这个配置可有可无 -->

<!--${catalina.base}的获取tomcat根目录 -->

<property name="log.filePath" value="${catalina.base}/logs/bill"/>

<!-- 展示内容输出格式:显示年月日+那个线程+5个字符宽度显示那个级别level+输出的日志信息 -->

<property name="log.pattern"

value="%d{yyyy-MM-dd HH:mm:ss.SSS}[%thread]%-5level %logger{50}-%msg%n"/>

第二步：设置控制台的输出

<!-- 控制台输出设置 -->

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

<encoder> <!-- 引用上面的输出格式 -->

<pattern>${log.pattern}</pattern>

</encoder>

</appender>

第三步：设置debug的文件输出，和压缩log文件.gz

<!-- debug级别日志文件保存设置 -->

<appender name="debug" class="ch.qos.logback.core.rolling.RollingFileAppender">

<!-- 文件路径 引用上面路径-->

<file>${log.filePath}/debug.log</file>

<!--rollingPolicy 设置日志文件进行压缩成.gz文件 -->

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

<!-- 文件名称 -->

<fileNamePattern>${log.filePath}/debug/debug.%d{yyyy-MM-dd}.log.gz</fileNamePattern>

<!-- 文件最大保存历史数量，因为每天都会产生gz,到了31天产生log就会把之前第一个给覆盖掉 -->

<maxHistory>30</maxHistory>

</rollingPolicy>

<!--encoder 此标签即将我们的这些日志信息转换为字符串，又输出到文件中-->

<encoder>

<pattern>${log.pattern}</pattern>

</encoder>

<!-- filter过滤掉 不是debug的那些日志，只记录debug日志-->

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

<level>DEBUG</level>

<!-- ACCEPT：是DEBUG 就接受 -->

<onMatch>ACCEPT</onMatch>

<!-- DENY 否定过滤掉 -->

<onMismatch>DENY</onMismatch>

</filter>

</appender>

第四步和第五步：设置info、error的文件输出和第三步设置一模一样(可有可无)这里省略

…….

第六步：关键：关联第三、四、五步，设置自己工程哪里需要此log信息打印

<!-- 5个级别trace<debug<info<warn<error 实际项目只需要：debug、info、error-->

<!-- 最后配置：1)需要为自己的那些类进行日志绑定 2)绑定上面配置的4个appender-->

<!--loger 参数 name：指定自己的包，levle从哪个级别开始 参数additivity="true"继承root标签-->

<logger name="com.wwl.controller" level="DEBUG" additivity="true">

<!-- 参数 ref：上面配置的三个级别的name属性值 -->

<appender-ref ref="debug"/>

<appender-ref ref="info"/>

<appender-ref ref="erro"/>

</logger>

<!--上面参数additivity="true" 所以都会输出到控制台 -->

<root level="INFO">

<!-- 参数 ref：上面配置控制台的name属性值 -->

<appender-ref ref="console"/>

</root>

</configuration>

第三步 代码中的实现：上面配置了com.wwl.controller包下需要logback

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

@Controller

@RequestMapping("/statrt")

public class StartrController {

Logger logger=LoggerFactory.getLogger(StartrController.class); //先创建类用slf4j工厂了创建logger

@Resource

private LoginService ls;

@RequestMapping(value="/login",method=RequestMethod.POST)

public String login(String username,String password,Model model) {

logger.info("=====start===="); // info的输出

long startTime = System.currentTimeMillis();

System.out.println(username+":"+password);

List<Users> list = ls.queryCodeNameAndpwd();

model.addAttribute("list", list);

long endTime = System.currentTimeMillis();

logger.error("测试用的"); // error的输出

logger.debug("代码运行时间:[{}ms]",endTime-startTime); // debug的输出

logger.info("=====end====="); // error的输出

return "index";

}

}

以上3步即可

### 注意问题

1、${catalina.base}：此变量是找到tomcat中的根目录

2、工程开启logback会打印很多系统信息，作用：maven启动内部tomcat就可以找到位置

3、不要和其它日志一起用：如log4j的jar包一定要去掉

4、支持写法："代码运行时间:[{}ms]",endTime-startTime