Spring Boot 热部署与日志

Spring Boot 热部署与日志

1.springboot中devtools热部署

2.带你弄清混乱的JAVA日志体系!

3. logback日志的集成

1.springboot中devtools热部署

1.1 引言

> 为了进一步提高开发效率,springboot为我们提供了全局项目热部署,日后在开发过程中修改了部分代码以及相关配置文件后,不需要每次重启使修改生效,在项目中开启了springboot全局热部署之后只需要在修改之后等待几秒即可使修改生效。

1.2 开启热部署

1.2.1 项目中引入依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <optional>true</optional>
</dependency>
```

1.2.2IDEA中配置

当我们修改了类文件后，idea不会自动编译，得修改idea设置。

- （1）File-Settings-Compiler-Build Project automatically
- （2）ctrl + shift + alt + / ,选择Registry,勾上 Compiler autoMake allow when app running

13.2.3 启动项目检测热部署是否生效

 # 1.启动出现如下日志代表生效

```
2019-07-17 21:23:17.566  INFO 4496 --- [  restartedMain] com.baizhi.InitApplication               : Starting InitApplication on chenyannandeMacBook-Pro.local with PID 4496 (/Users/chenyannan/IdeaProjects/ideacode/springboot_day1/target/classes started by chenyannan in /Users/chenyannan/IdeaProjects/ideacode/springboot_day1)
2019-07-17 21:23:17.567  INFO 4496 --- [  restartedMain] com.baizhi.InitApplication               : The following profiles are active: dev
2019-07-17 21:23:17.612  INFO 4496 --- [  restartedMain] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@66d799c5: startup date [Wed Jul 17 21:23:17 CST 2019]; root of context hierarchy
2019-07-17 21:23:18.782  INFO 4496 --- [  restartedMain] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8989 (http)
2019-07-17 21:23:18.796  INFO 4496 --- [  restartedMain] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2019-07-17 21:23:18.797  INFO 4496 --- [  restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.20
```

> 注意:日志出现restartedMain代表已经生效,在使用热部署时如果遇到修改之后不能生效,请重试重启项目在试

2.带你弄清混乱的JAVA日志体系!

引言

还在为弄不清commons-logging.jar、log4j.jar、sl4j-api.jar等日志框架之间复杂的关系而感到烦恼吗?

还在为如何统一系统的日志输出而感到不知所措嘛?

您是否依然存在这样的烦恼。比如，要更改spring的日志输出为log4j 2，却不知该引哪些jar包，只知道去百度一下所谓的博客，照着人家复制，却无法弄懂其中的原理?

不要急，不要方！徐庶老师带你们弄懂其中的原理

正文

日志框架发展史

张三 老程序员 jdk1.3 System.out.println("")追踪 ； 异常 try catch{ System.out.println("") } 输出一些关键变量 System.out.println("")
上线，部署了大量的System.out.println("") 。 异常 ---> 张三 服务确认异常信息， 非常郁闷？ 追踪

信息 ，记录在文本， 去服务器追踪就不会查无对症，
1. logUtil loginfo 替换 . 就可以顺利追踪到了。 不会立马反馈， 而且随着项目运营越来越多用户请求量越来越大， 1天1G， 2G
2. 日志信息 按天迭代， 2020-10-01.log ,... 按物理大小跌打 2020-10-01-20M.log
3. 用户出现异常loginfo(错误信息,等级) ,能不能给我马上发送邮件。
4. 能不能记录日志的时候按等级来区分 追踪1 信息2 调试3 异常 4 ，1天400\500M
5.i/o 异步 ... 自由控制格式 ..
开源 log4j . 受到广大开发者欢迎。 log4j simple log4j nop.....。
apatch基金会收纳

jdk官方 sun ， 开发出来了一个日志框架 jul java.util.logging .

市面上的日志框架非常的混乱， 一个项目 一个模块 log4j， 一个模块 jul ， 一个jboss-logging.
jdk 日志门面 JCL jakarta Commons Logging JCL（不实现日志功能，整合日志的） spring

张三 发现并不好用， 张三离开了apatch， 独自开发日志门面 **slf4j** . （不实现日志功能，整合日志的） 适配器、**桥接器。**

模块1 官方 JCL门面 jul
适配器
模块2 开源 slf4j **桥接器** **log4j**

apatch log4j2 性能 log4j高出好多倍
张三 logback 性能 log4j高出好多倍

日志实现	日志门面
log4j 淘汰	JCL
jul java.util.logging 别的	**SJF4J**
log4j2	
logback	

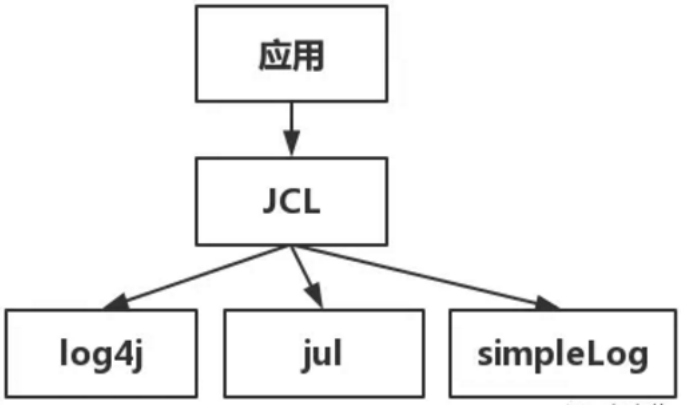
早年，你工作的时候，在日志里使用了log4j框架来输出，于是你代码是这么写的

```
import org.apache.log4j.Logger;
\\省略
Logger logger = Logger.getLogger(Test.class);
logger.trace("trace");
\\省略
```

但是，岁月流逝，sun公司对于log4j的出现内心隐隐表示嫉妒。于是在jdk1.4版本后，增加了一个包为java.util.logging，简称为jul，用以对抗log4j。于是，你的领导要你把日志框架改为jul，这时候你只能一行行的将log4j的api改为jul的api，如下所示

```
import java.util.logging.Logger;
\\省略
Logger loggger = Logger.getLogger(Test.class.getName());
logger.finest("finest");
\\省略
```

可以看出，api完全是不同的。那有没有办法，将这些api抽象出接口，这样以后调用的时候，就调用这些接口就好了呢?
这个时候jcl(Jakarta Commons Logging)出现了，说jcl可能大家有点陌生，讲commons-logging-xx.jar组件，大家总有印象吧。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。JCL可以实现的集成方案如下图所示

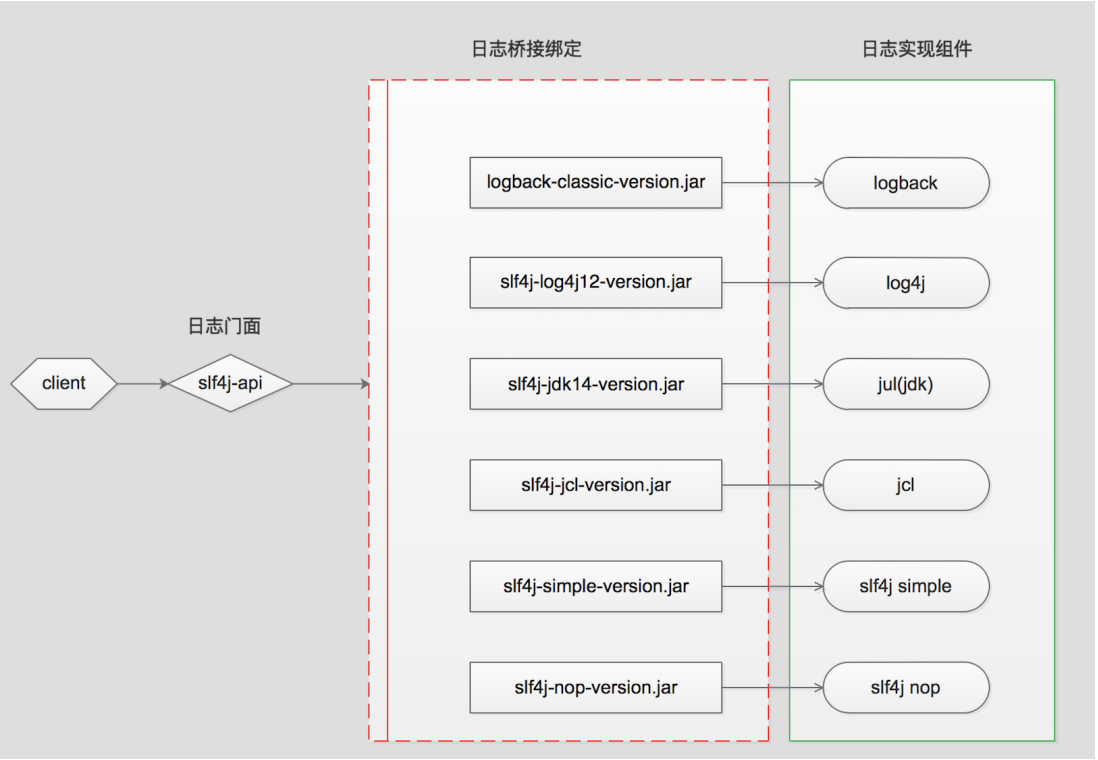


jcl默认的配置：如果能找到Log4j 则默认使用log4j 实现，如果没有则使用jul(jdk自带的) 实现，再没有则使用jcl内部提供的SimpleLog 实现。
于是，你在代码里变成这么写了

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
\\省略
Log log =LogFactory.getLog(Test.class);
log.trace('trace');
\\省略
```

至于这个Log具体的实现类，JCL会在ClassLoader中进行查找。这么做，有三个缺点，缺点一是效率较低，二是容易引发混乱，三是在使用了自定义ClassLoader的程序中，使用JCL会引发内存泄露。
JCL动态查找机制进行日志实例化，执行顺序为：commons-logging.properties---->系统环境变量------->log4j--->jul--->simplelog---->nooplog

于是log4j的作者觉得jcl不好用，自己又写了一个新的接口api，那么就是slf4j。关于slf4j的集成图如下所示



理解slf4j日志门面了吗，它跟jcl机制不一样。 它就相当于这个游戏机， 我本身没有游戏， 只提供一个运行游戏的平台（门面）要运行哪个游戏我不管， 你给我放哪块光盘我就运行哪个游戏。 JCL是自己去找，先找到哪个运行哪个

Slf4j与其他各种日志组件的桥接说明

jar包名	说明
slf4j-log4j12-1.7.13.jar	Log4j1.2版本的桥接器，你需要将Log4j.jar加入Classpath。
log4j-slf4j-impl.jar	Log4j2版本的桥接器,还需要`log4j-api.jar` `log4j-core.jar`
slf4j-jdk14-1.7.13.jar	java.util.logging的桥接器，Jdk原生日志框架。
slf4j-nop-1.7.13.jar	NOP桥接器，默默丢弃一切日志。
slf4j-simple-1.7.13.jar	一个简单实现的桥接器，该实现输出所有事件到System.err. 只有Info以及高于该级别的消息被打印，在小型应用中它也许是有用的。
slf4j-jcl-1.7.13.jar	Jakarta Commons Logging 的桥接器. 这个桥接器将Slf4j所有日志委派给Jcl。
logback-classic-1.0.13.jar(requires logback-core-1.0.13.jar)	Slf4j的原生实现，Logback直接实现了Slf4j的接口，因此使用Slf4j与Logback的结合使用也意味更小的内存与计算开销

如图所示，应用调了sl4j-api，即日志门面接口。日志门面接口本身通常并没有实际的日志输出能力，它底层还是需要去调用具体的日志框架API的，也就是实际上它需要跟具体的日志框架结合使用。由于具体日志框架比较多，而且互相也大都不兼容，日志门面接口要想实现与任意日志框架结合可能需要对应的桥接器，上图红框中的组件即是对应的各种桥接器！

我们在代码中需要写日志，变成下面这么写

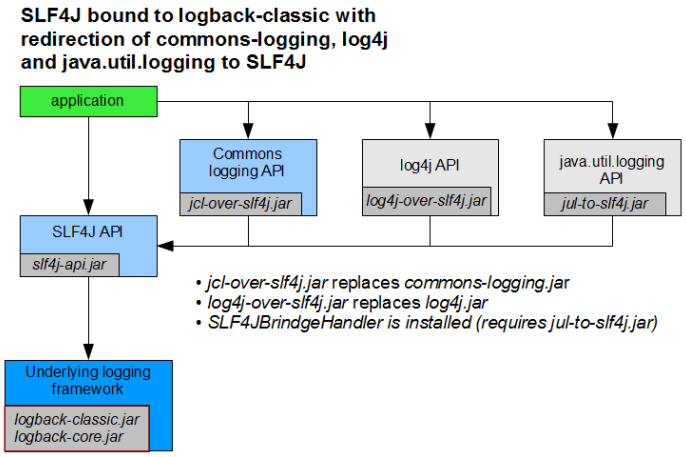
```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
//省略
Logger logger = LoggerFactory.getLogger(Test.class);
// 省略
logger.info("info");
```

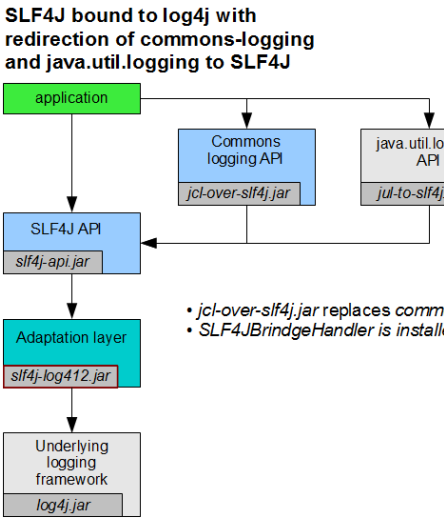
在代码中，并不会出现具体日志框架的api。程序根据classpath中的桥接器类型，和日志框架类型，判断出logger.info应该以什么框架输出！注意了，如果classpath中不小心引了两个桥接器，那会直接报错的！

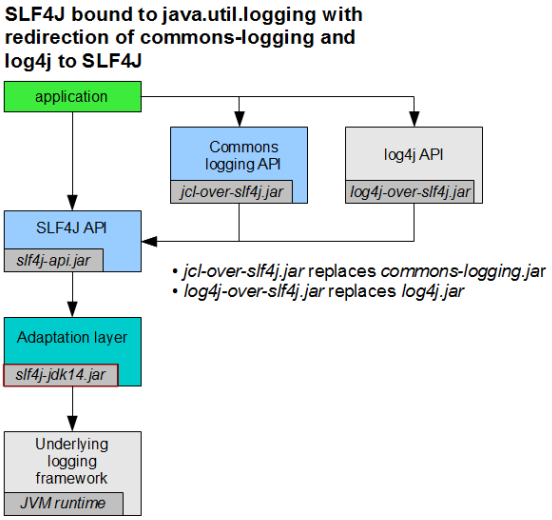
因此，在阿里的开发手册上才有这么一条

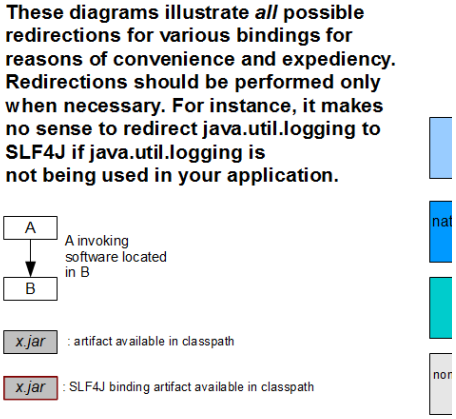
强制：应用中不可直接使用日志系统（log4j、logback）中的 API，而应依赖使用日志框架 SLF4J 中的 API。使用门面模式的日志框架，有利于维护和各个类的日志处理方式的统一。

- **如果要将jcl或jul 转slf4j呢?**









ok，至此，基础知识完毕，下面是实战！

日志实战

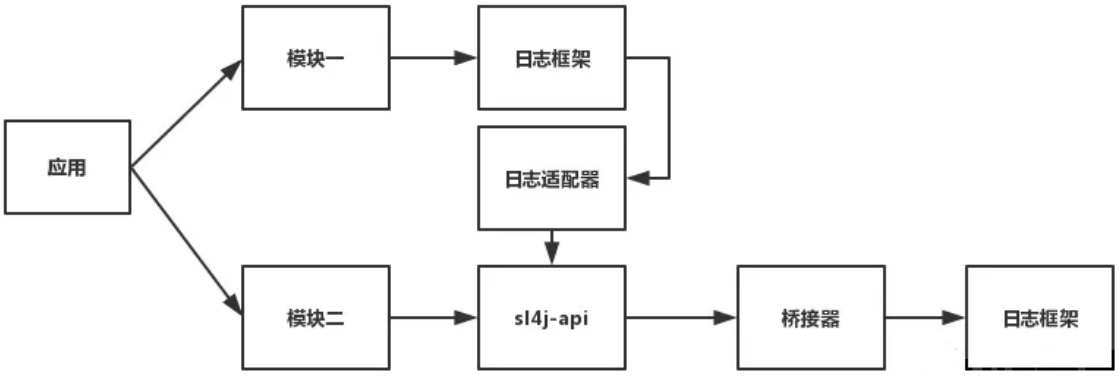
案例一

一个项目，一个模块用log4j，另一个模块用slf4j+log4j2,如何统一输出?

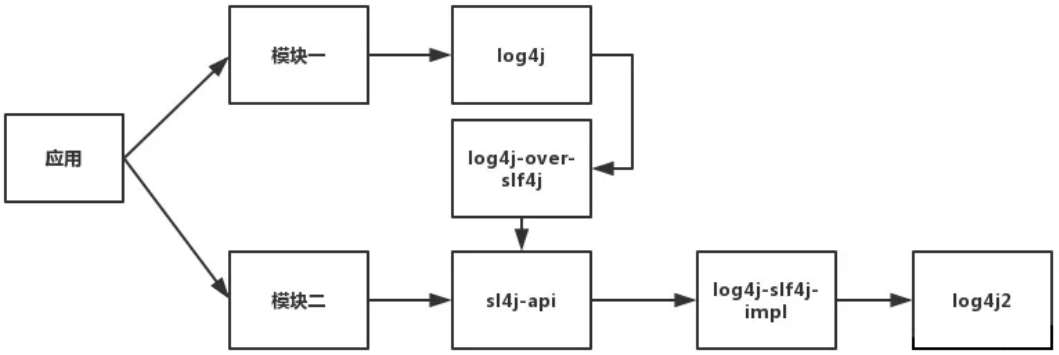
其实在某些中小型公司，这种情况很常见。我曾经见过某公司的项目，因为研发不懂底层的日志原理，日志文件里头既有log4j.properties,又有log4j2.xml，各种API混用，惨不忍睹!

还有人用着jul的API，然后拿着log4j.properties，跑来问我，为什么配置不生效！简直是一言难尽！

OK，回到我们的问题，如何统一输出！OK，这里就要用上slf4j的适配器，slf4j提供了各种各样的适配器，用来将某种日志框架委托给slf4j。其最明显的集成工作方式有如下:



进行选择填空，将我们的案例里的条件填入,根据题意应该选log4j-over-slf4j适配器，于是就变成下面这张图



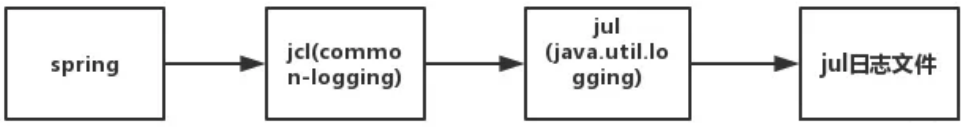
就可以实现日志统一为log4j2来输出!

ps:根据适配器工作原理的不同，被适配的日志框架并不是一定要删除！以上图为例，log4j这个日志框架删不删都可以，你只要能保证log4j的加载顺序在log4j-over-slf4j后即可。因为log4j-over-slf4j这个适配器的工作原理是，内部提供了和log4j一模一样的api接口，因此你在程序中调用log4j的api的时候，你必须想办法让其走适配器的api。如果你删了log4j这个框架，那你程序里肯定是走log4j-over-slf4j这个组件里的api。如果不删log4j，只要保证其在classpth里的顺序比log4j前即可!

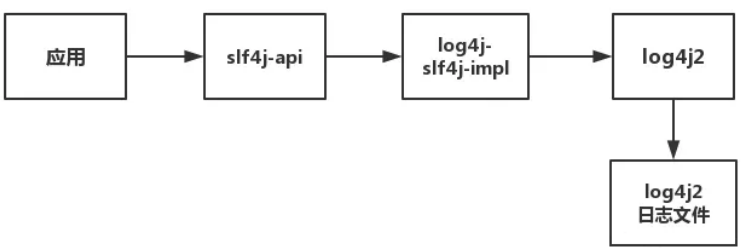
案例二

如何让spring以log4j2的形式输出?

spring默认使用的是jcl输出日志，由于你此时并没有引入Log4j的日志框架，jcl会以jul做为日志框架。此时集成图如下

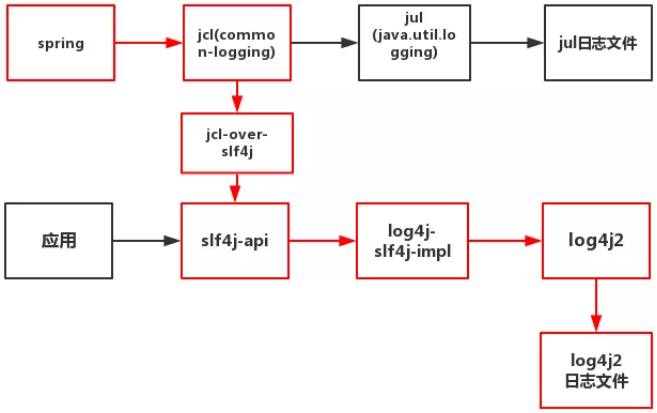


而你的应用中，采用了slf4j+log4j-core，即log4j2进行日志记录，那么此时集成图如下



那我们现在需要让spring以log4j2的形式输出？怎么办?

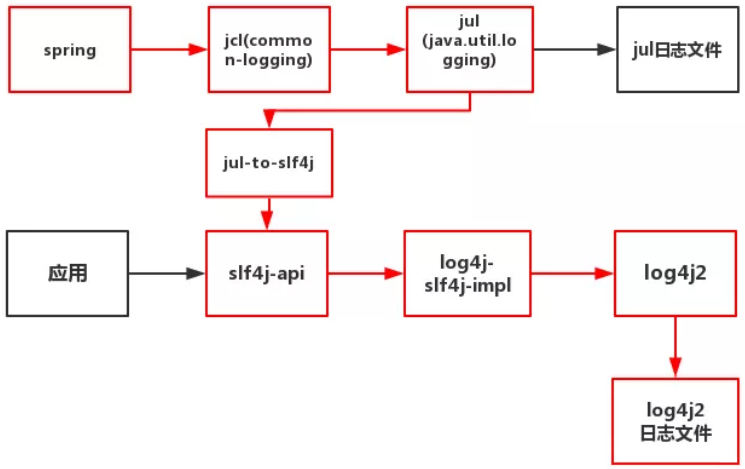
OK,第一种方案，走jcl-over-slf4j适配器，此时集成图就变成下面这样了



在这种方案下，spring框架中遇到日志输出的语句，就会如上图红线流程一样，最终以log4J2的形式输出！

OK，有第二种方案么？

有，走jul-to-slf4j适配器，此时集成图如下



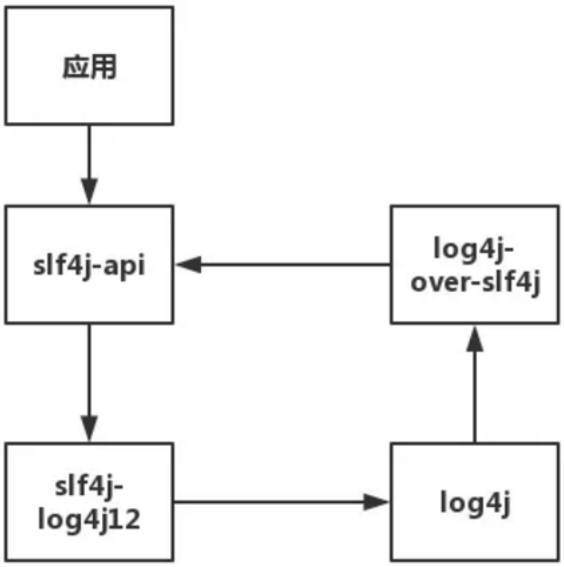
ps:这种情况下，记得在代码中执行

```
SLF4JBridgeHandler.removeHandlersForRootLogger();
SLF4JBridgeHandler.install();
```

这样jul-to-slf4j适配器才能正常工作，详情可以查询该适配器工作原理。

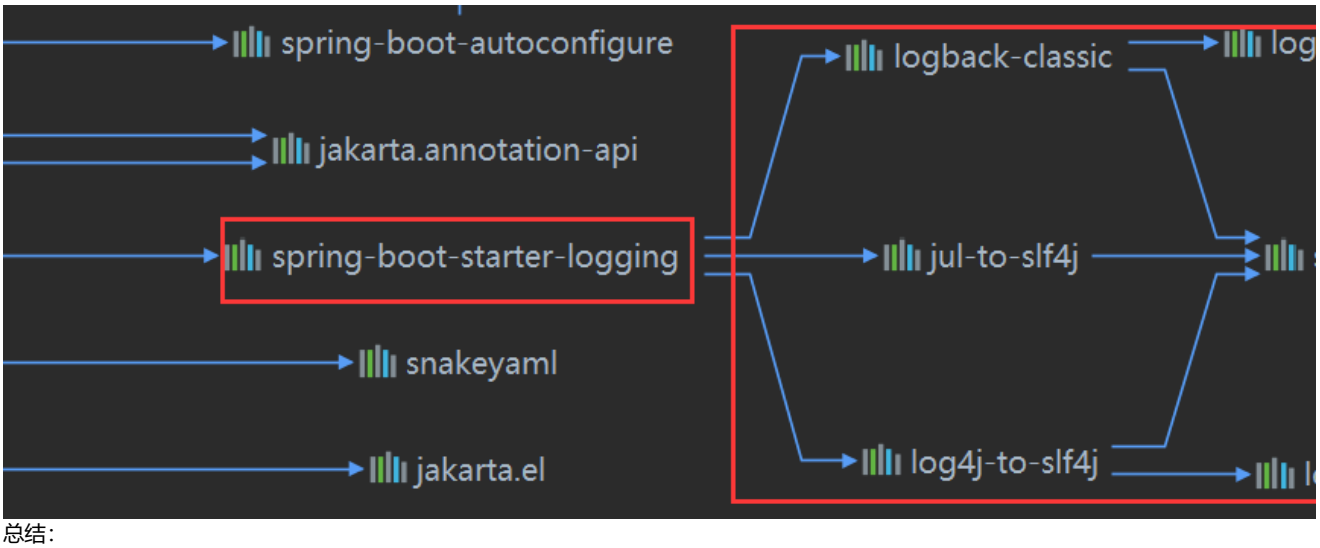
天啦噜！要死循环

假设，我们在应用中调用了sl4j-api，但是呢，你引了四个jar包，slf4j-api-xx.jar,slf4j-log4j12-xx.jar,log4j-xx.jar,log4j-over-slf4j-xx.jar，于是你就会出现如下尴尬的场面



如上图所示，在这种情况下，你调用了slf4j-api，就会陷入死循环中！slf4j-api去调了slf4j-log4j12,slf4j-log4j12又去调用了log4j，log4j去调用了log4j-over-slf4j。最终，log4j-over-slf4j又调了slf4j-api，陷入死循环！

3. logback日志的集成



总结：

1. SpringBoot底层也是使用slf4j+logback的方式进行日志记录
    a. logback桥接：logback-classic
2. SpringBoot也把其他的日志都替换成了slf4j；
    a. log4j 适配： log4j-over-slf4j
    b. jul适配：jul-to-slf4j
    c. 这两个适配器都是为了适配Spring的默认日志：jc

SpringBoot日志使用

- **日志级别**

可以设置TRACE，DEBUG，INFO，WARN，ERROR，FATAL或OFF之一

```
logging:
level:
root: "warn"
org.springframework.web: "debug"
org.hibernate: "error"
```

- **日志格式**

```
2020-12-01 14:01:34.665 TRACE 10072 --- [ main] com.tulingxueyuan.Application : 跟踪
2020-12-01 14:01:34.665 DEBUG 10072 --- [ main] com.tulingxueyuan.Application : 调试
2020-12-01 14:01:34.665 INFO 10072 --- [ main] com.tulingxueyuan.Application : 信息
2020-12-01 14:01:34.665 WARN 10072 --- [ main] com.tulingxueyuan.Application : 警告
2020-12-01 14:01:34.665 ERROR 10072 --- [ main] com.tulingxueyuan.Application : 异常
```

详细介绍

可以使用`logging.pattern.console` 修改默认的控制的日志格式：

```
%clr(%d{${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}}){faint} %clr(${LOG_LEVEL_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}
```

- 2020-12-01 14:01:34.665
  - 日期和时间：毫秒精度，易于排序。
  - **%clr**(%d{${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}})**{faint}**
    - **%clr** 当前内容的颜色 **{faint}**
  - (%d{${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}})
    - 括号中就是要显示的内容
  - %d{**${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}**}
    - **${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}**
      - ${value:value2} springboot的占位符 + null条件的表达式（如果value为null 使用value2)
      - LOG_DATEFORMAT_PATTERN: 系统环境变量中的值，springboot底层会根据对应的配置项将值设置到对应的环境变量中
        - 如：LOG_DATEFORMAT_PATTERN=logging.pattern.dateformat 可以在官网4.7章节中看到对应的关系。或者去源码中看
  - %d{-yyyy-MM-dd HH:mm:ss.SSS}
    - %d logbak的日期显示方式：
    - {-yyyy-MM-dd HH:mm:ss.SSS} 日期的格式
- TRACE
  - 日志级别：ERROR，WARN，INFO，DEBUG，或TRACE。
  - %clr(${LOG_LEVEL_PATTERN:-%5p})
  - %clr 颜色 会根据不同的日志级别输出对应的颜色
  - ${LOG_LEVEL_PATTERN:-%5p}
  - %5 代表当前内容所占字符长度
  - p 输出日志事件的级别。
- 10072
  - 进程ID。
  - %clr(${PID:- }){magenta}
  - %clr {magenta}
  - ${PID:- } springboot的占位符 + null条件的表达式（如果value为null 使用value2)

    - PID 是系统环境变量中的进程ID（由系统分配）
- ---
    - 一个---分离器来区分实际日志消息的开始。
- [main]
    - 线程名称：用方括号括起来（对于控制台输出可能会被截断）。
- com.tulingxueyuan.Application
    - 记录日志的类
- : 跟踪
    - 日志消息。

- **文件输出**

默认情况下，Spring Boot仅记录到控制台，不写日志文件。如果除了控制台输出外还想写日志文件，则需要设置一个logging.file.name或logging.file.path属性（例如，在中application.properties）。

下表显示了如何logging.*一起使用这些属性：

logging.file.name	logging.file.path	实例	描述
（没有）	*（没有）*		仅控制台记录。
指定文件名	*（没有）*	my.log	写入指定的日志文件。
（没有）	具体目录	/var/log	写入spring.log指定的目录。

- logging.file.name
    - 可以设置文件的名称， 如果没有设置路径会默认在项目的相对路径下
    - 还可以指定路径+文件名：`name: D:/xushu.log`
- logging.file.path
    - 不可以指定文件名称， 必须要指定一个物理文件夹路径，会默认使用spring.log

- **日志迭代（轮转）**

如果您使用的是Logback，则可以使用`application.properties`或`application.yaml`文件微调日志轮播设置。对于所有其他日志记录系统，您需要直接自己配置轮转设置（例如，如果使用Log4J2，则可以添加`log4j.xml`文件）。

名称	描述
`logging.logback.rollingpolicy.file-name-pattern`	归档的文件名
`logging.logback.rollingpolicy.clean-history-on-start`	如果应在应用程序启动时进行日志归档清理。
`logging.logback.rollingpolicy.max-file-size`	归档前日志文件的最大大小。
`logging.logback.rollingpolicy.total-size-cap`	删除日志档案之前可以使用的最大大小。
`logging.logback.rollingpolicy.max-history`	保留日志存档的天数（默认为7）

- `logging.logback.rollingpolicy.file-name-pattern`
    - ${LOG_FILE}.%d{yyyy-MM-dd}.%i.gz
        - ${LOG_FILE} 对应 logging.file.name
        - %d{yyyy-MM-dd} 日期 年-月-日
        - %i 索引， 当文件超出指定大小后进行的文件索引递增

- **自定义日志配置文件**

可以通过在类路径中包含适日志配置文件来激活各种日志记录系统或使用`logging.config`

Logging System	Customization
Logback	logback-spring.xml, logback-spring.groovy, logback.xml,

	or logback.groovy
Log4j2	log4j2-spring.xml or log4j2.xml
JDK (Java Util Logging)	logging.properties

注意：

- 如果使用自定义日志配置文件 会使用springboot中全局配置文件的logging相关配置失效
- 结合SpringBoot提供Profile来控制日志的生效
    - 注意： 一定要将日志配置文件的文件名改成logback-spring.xml，因为 logback.xml 会在Springboot容器加载前先被logback给加载到， 那么由于logback无法解析springProfile 将会报错：
        - `ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for [springProperty], current ElementPath is [[configuration][springProperty]]`
        `ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for [springProfile], current ElementPath is [[configuration][springProfile]]`

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
 <encoder>
 <!-- 格式 -->
 <springProfile name="dev">
 <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{100} ======= %msg%n</pattern>
 </springProfile>
 <springProfile name="!dev">
 <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{100} ++++++ %msg%n</pattern>
 </springProfile>

 </encoder>
</appender>
```

- **切换日志框架**

- 将 logback切换成log4j2
    1. 将logback的场景启动器排除（slf4j只能运行有1个桥接器）
    2. 添加log4j2的场景启动器
    3. 添加log4j2的配置文件

```
<dependencies>
 <dependency>
 <!--starter-web里面自动添加starter-logging 也就是logback的依赖-->
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
 <exclusions>
 <!--排除starter-logging 也就是logback的依赖-->
 <exclusion>
 <artifactId>spring-boot-starter-logging</artifactId>
  <groupId>org.springframework.boot</groupId>
 </exclusion>
 </exclusions>
 </dependency>

 <!--Log4j2的场景启动器-->
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-log4j2</artifactId>
```

```
 </dependency>
</dependencies>
```

- 将 logback切换成log4j

1. 要将logback的桥接器排除

```
<dependency>
 <!--starter-web里面自动添加starter-logging 也就是logback的依赖-->
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
 <exclusions>
 <exclusion>
 <artifactId>logback-classic</artifactId>
 <groupId>ch.qos.logback</groupId>
 </exclusion>
 </exclusions>
</dependency>
```

2. 添加log4j的桥接器

```
<dependency>
 <groupId>org.slf4j</groupId>
 <artifactId>slf4j-log4j12</artifactId>
</dependency>
```

3. 添加log4j的配置文件

log4j.properties

```
#trace<debug<info<warn<error<fatal
log4j.rootLogger=trace, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n
```