# 常用日志组件（Common-logging、slf4j、log4j、logback）调用关系源码解读

## 简单介绍

common-logging是Apahe提供的通用日志接口，以至于我们在使用spring的时候都要导入common-logging的jar包才可以，common-logging是采用动态查找的机制发现日志实现的，具体我们在下面分析。

slf4j同common-logging类似，SLF4J，即简单日志门面（Simple Logging Facade for Java），也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](http://baike.baidu.com/item/Facade" \t "_blank)，不同于common-logging的是slf4j是在编译期静态绑定Log实现库的，所以说使用slf4j时必须导入某一个日志实现

Log4j是[Apache](http://baike.baidu.com/item/Apache/8512995" \t "_blank)的一个开源项目，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](http://baike.baidu.com/item/%E6%8E%A7%E5%88%B6%E5%8F%B0" \t "_blank)、文件、[GUI](http://baike.baidu.com/item/GUI)组件，甚至是套接口服务器、[NT](http://baike.baidu.com/item/NT/3443842" \t "_blank)的事件记录器、[UNIX](http://baike.baidu.com/item/UNIX" \t "_blank) [Syslog](http://baike.baidu.com/item/Syslog)[守护进程](http://baike.baidu.com/item/%E5%AE%88%E6%8A%A4%E8%BF%9B%E7%A8%8B)等；我们也可以控制每一条

Logback是由[log4j](http://baike.baidu.com/item/log4j" \t "_blank)创始人设计的又一个开源日志组件。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志系统如log4j或JDK14 Logging。logback-access访问模块与[Servlet](http://baike.baidu.com/item/Servlet/477555" \t "_blank)容器集成提供通过Http来访问日志的功能。（摘自百度百科）

## Common-logging动态查找机制

Common-logging出自apache之手，所以apache项目中好多都使用了common-logging作用源代码的日志接口，其中较为广泛使用的就是我们几乎每天都要接触的spring框架，如下：

**package** org.springframework.core.env;

**import** org.apache.commons.logging.Log;

**import** org.apache.commons.logging.LogFactory;

**public** **abstract** **class** AbstractPropertyResolver **implements** ConfigurablePropertyResolver {

**protected** **final** Log logger = LogFactory.*getLog*(getClass());

}

在spring源代码中，如果用到日志打印的地方都有如上一段代码，从logFactory返回一个Log接口实现类，common-logging是采用动态查找机制查找日志实现的，下面看一下LogFactory.getLog方法的源代码：

**public** **static** Log getLog(String name) **throws** LogConfigurationException {

**return** *getFactory*().getInstance(name);

}

**public** **static** **final** String *FACTORY\_PROPERTIES* = "commons-logging.properties";

**public** **static** **final** String *FACTORY\_PROPERTY* = "org.apache.commons.logging.LogFactory";

**protected** **static** **final** String *SERVICE\_ID* = "META-INF/services/org.apache.commons.logging.LogFactory";

**public** **static** **final** String *FACTORY\_DEFAULT* = "org.apache.commons.logging.impl.LogFactoryImpl";

**public** **static** LogFactory getFactory() **throws** Exception {

//获取此线程对应的类加载器

ClassLoader contextClassLoader = *getContextClassLoaderInternal*();

// 如果这个加载器对应的logFactory实现已经存在于factories静态变量中，那么直接返回

LogFactory factory = *getCachedFactory*(contextClassLoader);

**if** (factory != **null**) {

**return** factory;

}

//读取项目根路径下的commons-logging.properties文件

Properties props = *getConfigurationFile*(contextClassLoader, *FACTORY\_PROPERTIES*);

ClassLoader baseClassLoader = contextClassLoader;

/\*\*

\* 第一步：查找系统属性中是否存在org.apache.commons.logging.LogFactory值，若有，则使用该值作为LogFactory的实现类

\*/

String factoryClass = *getSystemProperty*(*FACTORY\_PROPERTY*, **null**);

**if** (factoryClass != **null**) {

factory = *newFactory*(factoryClass, baseClassLoader, contextClassLoader);

}

**if** (factory == **null**) {

/\*\*

\* 第二步：查找META-INF/services/目录下的是否有org.apache.commons.logging.LogFactory文件

\*/

**final** InputStream is = *getResourceAsStream*(contextClassLoader, *SERVICE\_ID*);

**if**( is != **null** ) {

BufferedReader rd = **new** BufferedReader(**new** InputStreamReader(is, "UTF-8"));

String factoryClassName = rd.readLine();

**if** (factoryClassName != **null** && ! "".equals(factoryClassName)) {

factory = *newFactory*(factoryClassName, baseClassLoader, contextClassLoader );

}

}

}

**if** (factory == **null**) {

**if** (props != **null**) {

/\*\*

\* 第三步：在之前加载的common-logging.xml文件中查找是否有key为org.apache.commons.logging.LogFactory的属性，

\* 如果有，value值当做logFactory加载

\*/

factoryClass = props.getProperty(*FACTORY\_PROPERTY*);

**if** (factoryClass != **null**) {

factory = *newFactory*(factoryClass, baseClassLoader, contextClassLoader);

}

}

}

/\*\*

\* 第四步：最后如果都没有找到logFactory的实现，那么默认使用LogFactoryImpl

\*/

**if** (factory == **null**) {

factory = *newFactory*(*FACTORY\_DEFAULT*, *thisClassLoader*, contextClassLoader);

}

**return** factory;

}

从以上代码可以得出common-logging分四步去查找日志的实现方案：

1. 查找系统属性中是否存在org.apache.commons.logging.LogFactory值，若有，则使用该值作为LogFactory的实现类
2. 查找META-INF/services/目录下的是否有org.apache.commons.logging.LogFactory文件
3. 在之前加载的common-logging.properties文件中查找是否有key值为org.apache.commons.logging.LogFactory的属性，如果有，value值当做logFactory加载
4. 最后如果都没有找到logFactory的实现，那么默认使用LogFactoryImpl

以上就是common-logging怎么动态查找日志实现的机制。

## Common-logging和log4j日志实现框架整合

上面 我们已经分析过common-logging怎么去查找日志实现方案，无外乎那4中情况，在我们使用log4j的项目中，习惯是创建log4j。properties，并导入log4j.jar，这就OK了，反观log4j的jar包，META-INF/services包下也没有org.apache.commons.logging.LogFactory文件，所以排除了1，2，3步骤，所以只能是第四种，其实common-logging默认支持log4j，只要你导入log4j的jar包即可（对log4j有特殊照顾啊，~毕竟都是一家公司），第四种方案，getFactory方法直接返回LogFactoryImpl类，然后继续看getInstance()方法

**public** Log getInstance(String name) **throws** LogConfigurationException {

Log instance = (Log) instances.get(name);

**if** (instance == **null**) {

instance = newInstance(name);

instances.put(name, instance);

}

**return** instance;

}

//只copy核心代码

**protected** Log newInstance(String name) **throws** LogConfigurationException {

Log instance;

**if** (logConstructor == **null**) {

instance = discoverLogImplementation(name);

}

**return** instance;

}

**private** **static** **final** String[] *classesToDiscover* = {

"org.apache.commons.logging.impl.Log4JLogger",

"org.apache.commons.logging.impl.Jdk14Logger",

"org.apache.commons.logging.impl.Jdk13LumberjackLogger",

"org.apache.commons.logging.impl.SimpleLog"

};

**private** Log discoverLogImplementation(String logCategory){

Log result = **null**;

**if** (*isDiagnosticsEnabled*()) {

**for**(**int** i=0; i<*classesToDiscover*.length && result == **null**; ++i) {

result = createLogFromClass(*classesToDiscover*[i], logCategory, **true**);

}

**if** (result == **null**) {

**throw** **new** LogConfigurationException("No suitable Log implementation");

}

**return** result;

}

到这应该都能看的差不多明白吧，common-logging支持的默认日志实现有log4j和JDK支持的日志实现Jdk14Logger、Jdk13LumberjackLogger，剩下的一个SimpleLog是common-logging本身的日志实现，只支持打印简单日志格式。

多说一句，虽说common-logging默认支持log4j、Jdk14Logger、Jdk13LumberjackLogger，其实只是在common-logging中有相应日志实现的接口，具体日志实现还是在各自的jar包中。

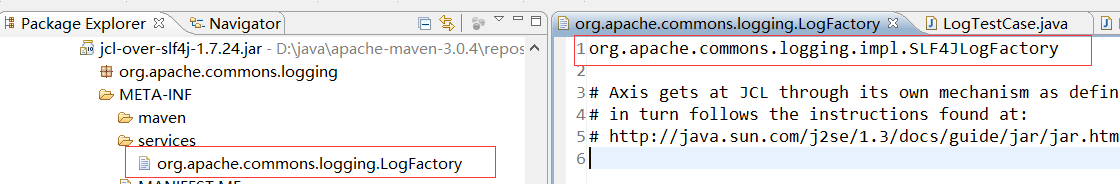
## Common-logging和slf4j桥接整合

我们知道common-logging和slf4j都不是日志的解决方案，它们只是提供了统一的日志接接口，那为什么这两个要整合呢 ？

因为很多框架中都是用common-logging来作为日志实现框架的，但是如果我们的项目中使用了logback作为我们的日志实现方案，那么我们也想让使用了common-logging作为日志实现的框架打印出我们项目所要求的日志格式，但是很不巧的是loabck并没有直接对common-logging做出支持，所以我们要引入slf4j，通过拿slf4j作为桥接，让common-logging和logback建立关系。

如果想让common-logging动态查找到slf4j需要引入一个桥接包，jcl-over-slf4j.jar

我们现在已经知道common-logging通过4中动态查找方式确定日志实现，我们看一下jcl-over-slf4j.jar的包结构：



是不是一目了然，采用了common-logging查找方案中的第二种，在META-INF/services/目录下配置了org.apache.commons.logging.LogFactory文件，

还记着在common-logging中getFactory方法查找到日志工厂SLF4JLogFactory，然后继续看getInstance()方法

**public** Log getInstance(String name) **throws** LogConfigurationException {

Log instance = **null**;

// protect against concurrent access of loggerMap

**synchronized** (loggerMap) {

instance = (Log) loggerMap.get(name);

**if** (instance == **null**) {

Logger logger = LoggerFactory.*getLogger*(name);

loggerMap.put(name, instance);

}

}

**return** (instance);

}

从以上源码中我们可以看到getInstance方法调用了lslf4j包中的LoggerFactory.getLogger(name)方法来确定具体的日志实现，这不正式slf4j和logback的使用方式吗 ？

## Slf4j和logback、log4j日志实现框架整合

在简介中我们可以了解到slf4j是通过静态绑定的方式确定日志实现方案的，在我们的项目中，一般会这样使用

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**public** **class** HomeController **extends** AbstractController{

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

**public** **static** Logger getLogger(Class<?> clazz) {

Logger logger = *getLogger*(clazz.getName());

**return** logger;

}

**public** **static** Logger getLogger(String name) {

ILoggerFactory iLoggerFactory = getILoggerFactory();

**return** iLoggerFactory.getLogger(name);

}

**public** **static** ILoggerFactory getILoggerFactory() {

**if** (INITIALIZATION\_STATE == UNINITIALIZED) { //是否是未初始化状态

INITIALIZATION\_STATE = ONGOING\_INITIALIZATION; //修改状态为-初始化中

performInitialization(); //检查是否有日志现实方案

}

**switch** (INITIALIZATION\_STATE) {

**case** SUCCESSFUL\_INITIALIZATION: //成功

**return** StaticLoggerBinder.getSingleton().getLoggerFactory();

**case** NOP\_FALLBACK\_INITIALIZATION:

**return** NOP\_FALLBACK\_FACTORY;

**case** FAILED\_INITIALIZATION:

**throw** **new** IllegalStateException(UNSUCCESSFUL\_INIT\_MSG);

**case** ONGOING\_INITIALIZATION:

**return** TEMP\_FACTORY;

}

**throw** **new** IllegalStateException("Unreachable code");

}

**private** **final** **static** **void** performInitialization() {

bind();

**if** (INITIALIZATION\_STATE == SUCCESSFUL\_INITIALIZATION) {

versionSanityCheck();检查版本

}

}

**private** **final** **static** **void** bind() {

Set staticLoggerBinderPathSet = findPossibleStaticLoggerBinderPathSet();

reportMultipleBindingAmbiguity(staticLoggerBinderPathSet);

// the next line does the binding

StaticLoggerBinder.getSingleton();

INITIALIZATION\_STATE = SUCCESSFUL\_INITIALIZATION;

reportActualBinding(staticLoggerBinderPathSet);

emitSubstituteLoggerWarning();

}

**private** **static** String *STATIC\_LOGGER\_BINDER\_PATH* = "org/slf4j/impl/StaticLoggerBinder.class";

**private** **static** Set findPossibleStaticLoggerBinderPathSet() {

Set staticLoggerBinderPathSet = **new** LinkedHashSet();

**try** {

ClassLoader loggerFactoryClassLoader = LoggerFactory.**class**

.getClassLoader();

Enumeration paths;

**if** (loggerFactoryClassLoader == **null**) {

paths = ClassLoader.*getSystemResources*(*STATIC\_LOGGER\_BINDER\_PATH*);

} **else** {

paths = loggerFactoryClassLoader

.getResources(*STATIC\_LOGGER\_BINDER\_PATH*);

}

**while** (paths.hasMoreElements()) {

URL path = (URL) paths.nextElement();

staticLoggerBinderPathSet.add(path);

}

} **catch** (IOException ioe) {

Util.*report*("Error getting resources from path", ioe);

}

**return** staticLoggerBinderPathSet;

}

上面这一步是非常关键的一步，其中 loggerFactoryClassLoader.getResources(STATIC\_LOGGER\_BINDER\_PATH); 意思是查找项目根路径下 org/slf4j/impl/StaticLoggerBinder.class 此文件，可以看一下slf4j.jar包中是没有此文件的，反观logback-classic.jar有此文件，所以说slf4j是通过静态绑定的方式确定日志实现类的，你只有把logback.jar导入项目中即可。

可是log4j中没有StaticLoggerBinder.class文件，那是因为slf4j和log4j整合，中间也需要一个桥接包，slf4j-log4j.jar ，导入项目中即可。

通过以上总结，总算知道日志框架之间是怎么搞的了