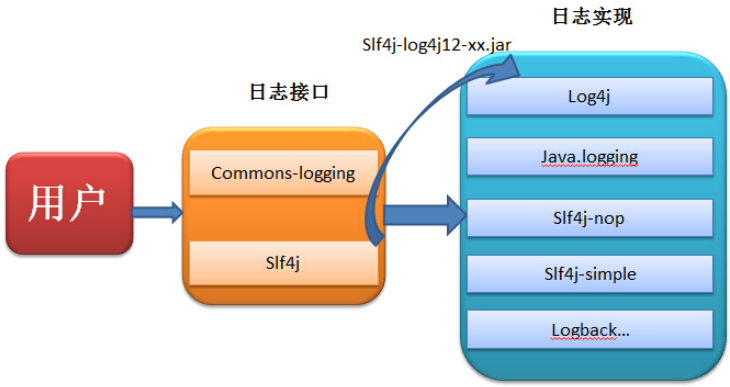
日志那点事儿——slf4j源码剖析

> **前言：**
>
> 说到日志，大多人都没空去研究，顶多知道用logger.info或者warn打打消息。那么commons-logging,slf4j,logback,log4j,logging又是什么关系呢？其中一二，且听我娓娓道来。
>
> 手码不易，转载请注明_xingoo！
>
> 涉及到的内容：日志系统的关系、Slf4j下载、源文件jar包的使用、Slf4j源码分析、JVM类加载机制浅谈

首先八卦一下这个日志家族的成员，下面这张图虽然没有包含全部的内容，但是基本也涵盖了日志系统的基本内容，不管怎么说，先记住下面这张图：



通过上面的图，可以简单的理清关系！

commons-logging和slf4j都是日志的接口，供用户使用，而没有提供实现！

log4j,logback等等才是日志的真正实现。

当我们调用接口时，接口的工厂会自动寻找恰当的实现，返回一个实现的实例给我服务。这些过程都是**透明**化的，用户不需要进行任何操作！

这里有个小故事，当年Apache说服log4j以及其他的日志来按照commons-logging的标准编写，但是由于commons-logging的类加载有点问题，实现起来也不友好，因此log4j的作者就创作了slf4j，也因此而与commons-logging两分天下。至于到底使用哪个，由用户来决定吧。

这样，slf4j出现了，它通过简单的实现就能找到符合自己接口的实现类，如果不是满足自己标准的日志，可以通过一些中间实现比如上面的slf4j-log4j12.jar来进行适配。

如此强大的功能，是如何实现的呢？

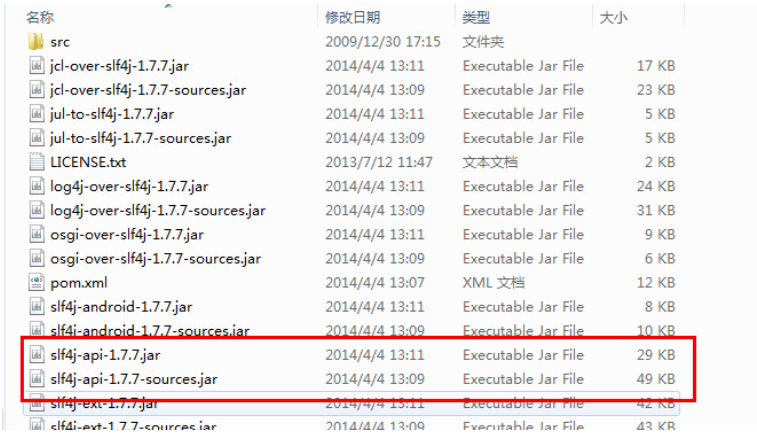
slf4j下载

首先为了查阅源码，这里先教大家如何使用开源的jar包！

例如在官网：http://www.slf4j.org/download.html

这里提供给我们两个版本，linux下的tar.gz压缩包，和windows下的zip压缩包。

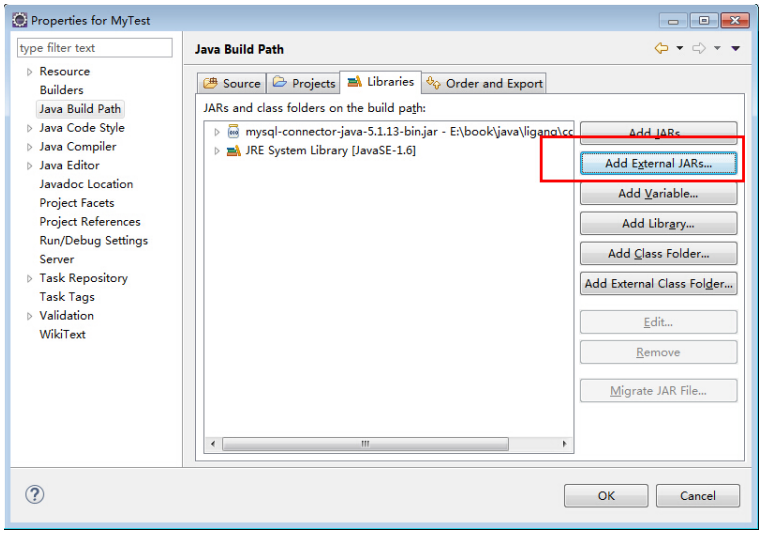
下载zip文件后解压，可以找到提供给我们的使用工具包。一般来说，这种开源的项目会为我们提供两种jar包，就拿slf4j（有人叫他，撒拉风four接，很有意思的名字）slf4j.jar、slf4j-source.jar：



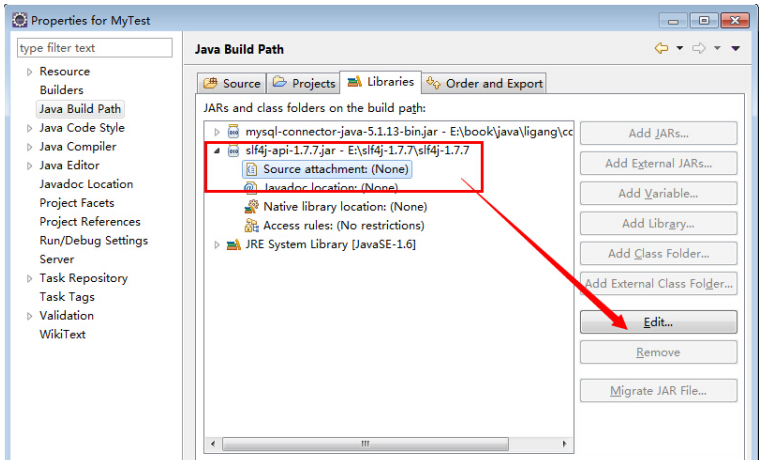
这里slf4j-api-xxx.jar就是它的核心包，而slf4j-api-xxx-source.jar是它的源码包，里面包含了未编译的java文件。

那么如何使用呢？

首先在eclipse中添加外部的jar包，引入api.jar



添加jar包，然后编辑sourceattachment，可以点击edit，也可以双击

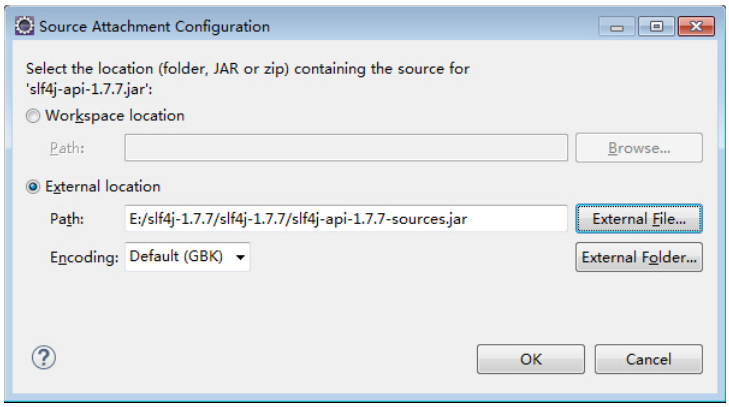


Hadoop(17)
Hbase(2)
Java(135)
JavaScript(71)
Kafka(2)
linux(29)
Logstash(16)
MongoDB(2)
Oozie(12)
Oracle(44)
Python(1)
Redis(1)
Ruby(8)
Scala(8)
Spark(19)
Spring(25)
Sqoop(3)
TensorFlow(1)
程序人生(117)
缓存系统(6)
机器学习(8)
全栈折腾(11)
软件考试(1)
设计模式(23)
数据仓库(9)
数学理论
算法(84)
网络(19)
杂谈(16)

随笔档案(916)

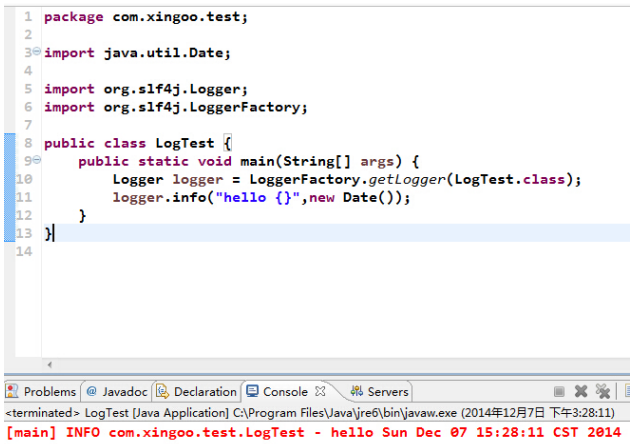
2017年8月 (7)
2017年7月 (7)
2017年6月 (14
2017年5月 (4)
2017年4月 (2)
2017年3月 (8)
2017年2月 (9)
2017年1月 (3)
2016年12月 (9
2016年11月 (9
2016年10月 (1
2016年9月 (18
2016年8月 (15
2016年7月 (10
2016年6月 (13
2016年5月 (15
2016年4月 (14
2016年3月 (27
2016年2月 (12
2016年1月 (20
2015年12月 (1
2015年11月 (3
2015年10月 (2
2015年9月 (11
2015年8月 (13
2015年7月 (12
2015年6月 (10
2015年5月 (9)
2015年4月 (22
2015年3月 (23
2015年2月 (33
2015年1月 (16
2014年12月 (1
2014年11月 (3
2014年10月 (1
2014年9月 (25
2014年8月 (19
2014年7月 (8)
2014年6月 (21
2014年5月 (20

引入source文件，这样，我们就是查看api.jar包中的class文件的源码了！



接下来进入正题，slf4j源码的解读！

首先日志的用法很简单，通过工厂factory获取log对象，然后打印消息就可以了！看一下效果，无图无真相！



main的代码在这里：

```
package com.xingoo.test;

import java.util.Date;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class LogTest {
    public static void main(String[] args) {
        Logger logger = LoggerFactory.getLogger(LogTest.class);
        logger.info("hello {}",new Date());
    }
}
```

这里也可以看到Slf4j的一个很重要的特性，**占位符**！—— {} 可以任意的拼接字符串，自动的填入字符串中！用法用户可以自己去尝试，这里就不再赘述了。

为了便于理解下面的代码，推荐先了解一下**facade外观模式**，因为Slf4j就是利用外观模式，提供对外的接口！

可以参考之前整理的设计模式的帖子：设计模式

首先，直接用LoggerFactory的静态工厂获取一个Logger对象，我们先看下getLogger方法！

```
1  public static Logger getLogger(Class clazz) {
2     return getLogger(clazz.getName());
3   }
```

这里把传入的类，提取出名字，再填写到getLogger静态方法中！这里博友们可能有一个疑问，为什么要获取类的名字，而根据名字来获取对象呢。因为每个类使用的日志处理实现可能不同，iLoggerFactory中也是根据名字来判断一个类的实现方式的。

```
public static Logger getLogger(String name) {
  ILoggerFactory iLoggerFactory = getILoggerFactory();
  return iLoggerFactory.getLogger(name);
 }
```

在getLogger方法中，通过getLoggerFactory获取工厂，然后获取日志对象！看来一切的迷雾都在getILoggerFactory()中！

```
 1   public static ILoggerFactory getILoggerFactory() {
 2     if (INITIALIZATION_STATE == UNINITIALIZED) {
 3       INITIALIZATION_STATE = ONGOING_INITIALIZATION;
 4       performInitialization();
 5     }
 6     switch (INITIALIZATION_STATE) {
 7       case SUCCESSFUL_INITIALIZATION:
 8         return StaticLoggerBinder.getSingleton().getLoggerFactory();
 9       case NOP_FALLBACK_INITIALIZATION:
10         return NOP_FALLBACK_FACTORY;
11       case FAILED_INITIALIZATION:
12         throw new IllegalStateException(UNSUCCESSFUL_INIT_MSG);
13       case ONGOING_INITIALIZATION:
14         // support re-entrant behavior.
15         // See also http://bugzilla.slf4j.org/show_bug.cgi?id=106
16         return TEMP_FACTORY;
17     }
18     throw new IllegalStateException("Unreachable code");
19   }
20 }
```

这个方法稍微复杂一点，总结起来：

第2行~第5行：判断是否进行初始化，如果没有初始化，则修改状态，进入performIntialization初始化！

第6行~第17行：对状态进行测试，如果初始化成功，则通过StaticLoggerBinder获取日志工厂！

那么下面就看一下Slf4j如何进行初始化，又是如何获取日志工厂的！

```
private final static void performInitialization() {
  bind();
  if (INITIALIZATION_STATE == SUCCESSFUL_INITIALIZATION) {
    versionSanityCheck();
  }
 }
```

在初始化中，先bind()，在修改状态，进行版本检查！先看一下版本检查的内容：

```
private final static void versionSanityCheck() {
  try {
    String requested = StaticLoggerBinder.REQUESTED_API_VERSION;

    boolean match = false;
    for (int i = 0; i < API_COMPATIBILITY_LIST.length; i++) {
      if (requested.startsWith(API_COMPATIBILITY_LIST[i])) {
        match = true;
      }
    }
    if (!match) {
      Util.report("The requested version " + requested
              + " by your slf4j binding is not compatible with "
              + Arrays.asList(API_COMPATIBILITY_LIST).toString());
```

```
        Util.report("See " + VERSION_MISMATCH + " for further details.");
    } catch (java.lang.NoSuchFieldError nsfe) {
      // given our large user base and SLF4J's commitment to backward
      // compatibility, we cannot cry here. Only for implementations
      // which willingly declare a REQUESTED_API_VERSION field do we
      // emit compatibility warnings.
    } catch (Throwable e) {
      // we should never reach here
      Util.report("Unexpected problem occured during version sanity check", e);
    }
  }
```

这里获取JDK的版本，并与Slf4j支持的版本进行比较，如果大版本相同则通过，如果不相同，那么进行失败提示！

最关键的要看bind是如何实现的！

```
1  private final static void bind() {
2      try {
3        Set<URL> staticLoggerBinderPathSet = findPossibleStaticLoggerBinderPathSet();
4        reportMultipleBindingAmbiguity(staticLoggerBinderPathSet);
5        // the next line does the binding
6        StaticLoggerBinder.getSingleton();
7        INITIALIZATION_STATE = SUCCESSFUL_INITIALIZATION;
8        reportActualBinding(staticLoggerBinderPathSet);
9        fixSubstitutedLoggers();
10     } catch (NoClassDefFoundError ncde) {
11       String msg = ncde.getMessage();
12       if (messageContainsOrgSlf4jImplStaticLoggerBinder(msg)) {
13         INITIALIZATION_STATE = NOP_FALLBACK_INITIALIZATION;
14         Util.report("Failed to load class \"org.slf4j.impl.StaticLoggerBinder\".");
15         Util.report("Defaulting to no-operation (NOP) logger implementation");
16         Util.report("See " + NO_STATICLOGGERBINDER_URL
17                 + " for further details.");
18       } else {
19         failedBinding(ncde);
20         throw ncde;
21       }
22     } catch (java.lang.NoSuchMethodError nsme) {
23       String msg = nsme.getMessage();
24       if (msg != null && msg.indexOf("org.slf4j.impl.StaticLoggerBinder.getSingleton()") != -1) {
25         INITIALIZATION_STATE = FAILED_INITIALIZATION;
26         Util.report("slf4j-api 1.6.x (or later) is incompatible with this binding.");
27         Util.report("Your binding is version 1.5.5 or earlier.");
28         Util.report("Upgrade your binding to version 1.6.x.");
29       }
30       throw nsme;
31     } catch (Exception e) {
32       failedBinding(e);
33       throw new IllegalStateException("Unexpected initialization failure", e);
34     }
35   }
```

第2行~第10行：初始化！首先获取实现日志的加载路径，查看路径是否合法，再初始化StaticLoggerBinder的对象，寻找合适的实现方式使用。

第10行~第22行：如果找不到指定的类，就会报错！

第22行~第31行：如果找不到指定的方法，就会报错！

第31行~第34行：报错！

通关查看代码，可以理解，这个方法的主要功能就是寻找实现类，如果找不到或者指定的方法不存在，都会报错提示！

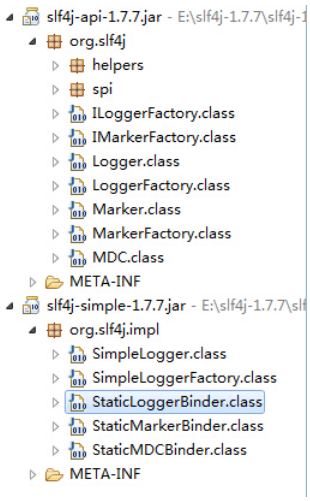
那么如何查找实现类呢？这就要看findPossibleStaticLoggerBinderPathSet方法了！

```
1  private static String STATIC_LOGGER_BINDER_PATH = "org/slf4j/impl/StaticLoggerBinder.class";
2
3    private static Set<URL> findPossibleStaticLoggerBinderPathSet() {
4      // use Set instead of list in order to deal with  bug #138
5      // LinkedHashSet appropriate here because it preserves insertion order during iteration
6      Set<URL> staticLoggerBinderPathSet = new LinkedHashSet<URL>();
7      try {
8        ClassLoader loggerFactoryClassLoader = LoggerFactory.class
9                .getClassLoader();
```

```
10      Enumeration<URL> paths;
11      if (loggerFactoryClassLoader == null) {
12        paths = ClassLoader.getSystemResources(STATIC_LOGGER_BINDER_PATH);
13      } else {
14        paths = loggerFactoryClassLoader
15                .getResources(STATIC_LOGGER_BINDER_PATH);
16      }
17      while (paths.hasMoreElements()) {
18        URL path = (URL) paths.nextElement();
19        staticLoggerBinderPathSet.add(path);
20      }
21    } catch (IOException ioe) {
22      Util.report("Error getting resources from path", ioe);
23    }
24    return staticLoggerBinderPathSet;
25  }
```

这里就是slf4j的源码精华之处！

第1行：它定义了一个字符串，这个字符串是实现类的class地址。然后通过类加载加载指定的文件！



第6行：创建一个Set,因为有可能有多个实现。

第8行~第9行：获取LoggerFactory的类加载器！

第11行~第13行：如果获取不到类加载器则说明是系统加载器，那么在系统路径下获取该资源文件

第13行~第15行：获取到了类加载器，则用该类加载器加载指定的资源文件。

第17行~第20行：解析类加载器的地址。

第24行：返回加载器地址的集合。

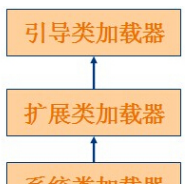
这里不了解类加载器的原理的可能会不大明白！

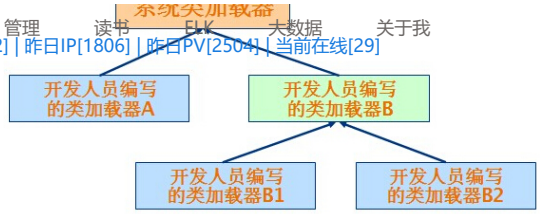
在JVM中，最后的文件都是Class文件，也就是字节码文件，因此需要把该文件加载到JVM中才能运行。而加载的过程，只会执行静态代码块。

类加载器分为三种：BootStrapClassLoader加载器，ExtensionClassLoader标准扩展加载器，SystemClassLoader系统类加载器。

每一种加载器加载指定的class文件会得到不同的类，因此为了能够使用，这里必须要保证LoggerFactory的类加载器与StaticLoggerBinder的类加载是相同的。

为了避免不同的加载器加载后会出现不一致的问题，JVM采用了一种**父类委托机制**的实现方式，也就是说，用户加载器会首先委托父类系统加载器，系统加载器再寻找父类——标准扩展加载器来加载类，而标准扩展加载器又会委托它的父类引导类加载器来加载，引导类加载器是属于最高级别的类加载器，它是没有父类加载器的。这里可以通过一个简单的图来表示他们的关系：





而用户在运行期，也是获取不到引导类加载器的，因此当一个类获取它的类加载器，得到的对象时null，就说明它是由引导类加载器加载的。引导类加载器是负责加载系统目录下的文件，因此源码中使用getSystemresource来获取资源文件。

这个地方虽然有点绕，但是理解起来还应该算简单吧！

如果没有理解加载器的机制，那么推荐看一下《深入理解JVM》，或者推荐的帖子：类加载机制

总结Slf4j工作原理

上面的内容说多不多，说少也不少！

你需要了解：JVM类加载机制、设计模式——外观模式，Eclipse jar包使用，然后就是慢慢的阅读源码。

简单的说下它的原理，就是通过工厂类，提供一个用户的接口！用户可以通过这个外观接口，直接使用API实现日志的记录。而后面的具体实现由Slf4j来寻找加载.寻找的过程，就是通过类加载加载那个叫org/slf4j/impl/StaticLoggerBinder.class的文件，只要实现了这个文件的日志实现系统，都可以作为一种实现方式。如果找到很多种方式，那么就寻找一种默认的方式。

这就是日志接口的工作方式，简单高效，关键是完全解耦！不需要日志实现部分提供任何的修改配置，只需要符合接口的标准就可以加载进来。

分类: Java


xingoo
关注 - 64
粉丝 - 2584
13 0
+加关注

« 上一篇：【Hibernate那点事儿】—— Hibernate应该了解的知识
» 下一篇：【Hibernate那点事儿】—— Hibernate知识总结

posted @ 2014-12-07 16:21 xingoo 阅读(19537) 评论(11) 编辑 收藏

评论列表

#1楼 2014-12-07 22:27 沧海一滴

加载配置文件Log4j.property在源码中哪个class中实现的

支持(0) 反对(0)

#2楼[楼主] 2014-12-08 09:10 xingoo

@ 沧海一滴
LoggerFactory里面，上面介绍的代码都是这里面的。StaticLoggerBinder是实现类比如log4j、Slf4j-simple里面的

支持(0) 反对(0)

#3楼 2014-12-09 08:16 沧海一滴

源码中没找到。在官网faq中找到了：
log4j uses Thread.getContextClassLoader().getResource() to locate the default configuration files and does not directly check the file system.
http://logging.apache.org/log4j/1.2/faq.html#noconfig

支持(0) 反对(0)

#4楼[楼主] 2014-12-09 08:51 xingoo

@ 沧海一滴
哦 你说的是property文件啊，我理解错了。我以为是StaticLoggerBinder呢。这个类负责进行Log4j的实例化，至于加载配置文件，那就应该看Log4j的源码了吧

支持(0) 反对(0)

#5楼 2015-01-21 14:23 沧海一滴

Logger logger = LoggerFactory.getLogger(LogTest.class);
是否线程安全问题？

现在有个功能卡了，
（1）工具类Utils只有一个静态字段private static Logger logger= LoggerFactory.getLogger(LogTest.class);
（2）工具类Utils中一个静态方法中的日志没有打印、
现在怀疑是logger初始化时由于线程安全问题而卡住了。

支持(0) 反对(0)

#6楼[楼主] 2015-02-01 16:58 xingoo

@ 沧海一滴
private static不会出现线程安全问题把？你的问题解决了吗？写个简单的样例，看看能不能重现。

支持(0) 反对(0)

#7楼 2015-02-03 22:37 沧海一滴

@ xingoo
是一个功能中的一部分，因为原因没定位，样例还没想好怎么写。

支持(0) 反对(0)

#8楼 2015-02-03 22:37 沧海一滴

@ xingoo
有没有好的方法来定位原因

支持(0) 反对(0)

#9楼[楼主] 2015-02-03 22:55 xingoo

@ 沧海一滴
不好意思，多线程不太会，最近弄完spring就学学看。暂时帮不了你了

支持(0) 反对(0)

#10楼 2017-03-14 16:11 dreamcatcher-cx

好文得顶

支持(0) 反对(0)

#11楼 2017-08-16 13:54 秋已寒

写的不错。

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。