那些年我们用过的日志框架

转载 pengjunlee 2018-05-09 10:25:53 44435 收藏

分类专栏： Java 文章标签： log4j logback

目前常见的Java日志框架和facades如下：

① log4j
② logback
③ SLF4J
④ commons-logging
⑤ j.u.l (即java.util.logging)

其中，①-③为同一个作者(Ceki)所写。④被很多开源项目所用，⑤是Java原生库(以下用j.u.l简写来代替)，但是在Java 1.4中才被引入。

这么多的日志库，我们该如何选择呢，我认为，这并非一道非此即彼的选择题，但是在了解它们的历史渊源和优劣以及相互关系的基础上才能更好地适配自己的项目。

下面我将上述这些框架串起来讲一下，如有疏漏请见谅。

1. Logging frameworks的上古时期(Java 1.3及以前)

在上古时期，Java打日志依赖System.out.println(), System.err.println()或者e.printStackTrace()。Debug日志被写入STDOUT流,错误日志被写入STDERR流。

这种方式目前小脚本中也依然使用广泛。但是在生产环境或大的项目中，Debug日志通常被重定向到/dev/null中: >/dev/null, 错误日志被重定向到本地文件中: >stderr.log。看起来很完美，是吗？实则不然，这样打日志有一个非常大的缺陷：无法可定制化。

具体来讲，没有一个类似开关的东东来切换是否打印Debug日志，当我们定位问题时需要输出Debug日志到文件去查看，而不是到/dev/null里，是吗？日志无法定制化，我们只能硬编码到代码里，不需要时再注释掉相关代码，重新编译。

还有一些缺陷，比如：无法更细粒度地输出日志，换句话说，缺少当前成熟的日志框架常见的LOG LEVEL控制。

而Java本身也没有提供相应的Library，在这样恶劣的境况下，Log4j勇敢地站了出来，拯救劳苦大众。

Log4j可以说是一个里程碑式的框架，它提出的一些基本理念，深深地影响了后来者，直至今天，这些理念也依然在被广泛使用：

Logger

我们来看下维基百科对Logger的定义：

```
A Logger is an object that allows the application to log without regard to where the output is sent/stored.
The application logs a message by passing an object or an object and an exception with an optional severity level
to the logger object under a given a name/identifier.
```

翻译过来，意思是说：

Logger是一个允许应用记录日志的对象，开发者不必需考虑输出位置。应用可将具体需要打印的信息通过一个Object传递。每个Logger互相独立，通过名字或标识符来区分。

Appender

每个appender可独立配置记录日志的设备，可以是文件、数据库、消息系统等。

Level

每个打印日志都可以单独制定日志级别。外部通过配置文件来控制输出级别，不同的输出级别打印不同的日志信息。

2. J.U.L姗姗来迟

后来，Sun公司开始意识到JDK需要一个记录日志的特性。受Log4j的启发，Sun在Java1.4版本中引入了一个新的API, 叫java.util.logging, 但是，j.u.l功能远不如Log4j完善，如果开发者要使用它，就意味着需要自己写Appenders(Sun称它为Handlers)，而且，只有两个Handlers可被使用：Console和File，这就意味着，开发者只能将日志写入Console和文件。

如前面所述，j.u.l在Java 1.4才被引入，在这之前，并没有官方的日志库供开发者使用。于是便有了很多日志相关的"轮子"。我想这应该是当前会有如此多日志框架的一个很重要的原因。

回顾历史，一方面，在Java 1.4之前，第三方日志库已经被广泛使用了，占得了先机。另一方面，j.u.l在被引入时性能和可用性都很差，直到1.5甚至以后才有了显著提升。

3-1. Logging facades出现及进化

由于项目的日志打印必然依赖以上两个框架中至少一个，无论是j.u.l还是log4j,开发者必须去两个都配置。这时候，Apache的commons-logging出现了。本质上来讲，commons-logging并非一个日志打印框架，而是一个API bridge, 它起到一个连接和沟通的作用，开发者可以使用它来兼容logging frameworks(j.u.l和log4j)。有了它，第三方库就可以使用commons-logging来做一个中间层，去灵活选择j.u.l或者log4j，而不必强加依赖。

然而commons-logging对j.u.l和log4j的配置问题兼容得并不好，更糟糕的是，使用commons-logging可能会遇到类加载问题，导致NoClassDefFoundError的错误出现。

最终，log4j的创始人Ceki发起了另一个项目，这便是大名鼎鼎的SLF4j 日志框架，该框架可以看成是log4j的升级版。需要说明的是，log4j 2.0已经被加入Apache基金会，过去几年已经被大幅改善，社区活跃度也非常高，借助开源社区的力量，log4j 2.0目前被加入越来越多得现代化特性，一定程度上，甚至超越了log4j的升级版logback(稍后介绍)，关于log4j 2.0的新特性，请参见这篇文章：THE NEW LOG4J 2.0 。

据slf4j的作者Ceki说，首先，slf4j不仅仅是一个logging framework, 而且一个logging facades, 借助slf4j的log4j adapter, 开发者从slf4j切换到log4j不需要额外改动一行代码，只需要从CLASS_PATH中排除掉slf4j-log4j12.jar。如果想从log4j迁移到logback, 在CLASS_PATH添加slf4j-log4j12.jar, 并将log4j.properties转换为logback.xml即可，这里有一个在线工具可以自动完成转换: logback.xml translator。

slf4j提供了很大的灵活度，开发者可以借助它去灵活选择底层的日志框架。比如，当下更多的开发者比较倾向于使用log4j的升级版logback,因为它具有较log4j更多更好的特性：

1. 配置文件支持xml和Groovy语法(版本号>= 0.9.22)
2. 自动重载有变更的配置文件
3. 自动压缩历史日志
4. 打印异常信息时自动包含package名称级版本号
5. Filters
6. 其它一些很棒的特性

需要说明的是，logback是slf4j接口的一套具体实现，又是同一个作者，因而保证了其和log4j相近的使用方式，也具有slf4j的全部特性。

此外，对于一些大型框架及服务的开发者，需要考虑客户端用户的体验。比如jstorm, 你不能只考虑自己的喜好，或许有人偏好使

用slf4j开发jstorm topology, 而另一些人喜欢用logback。这种情况下，你应该使用slf4j，把最终logging framework的选择权留给用户。

最后，除了slf4j比j.u.l或者log4j更好用，还有一个选择slf4j的现实原因：Java圈的非常多开发者更钟情于slf4j作为他们的logging API, 随大流有时候能少很多不必要的麻烦。

3-2. 日志参数化打印的支持(parameterized logging)

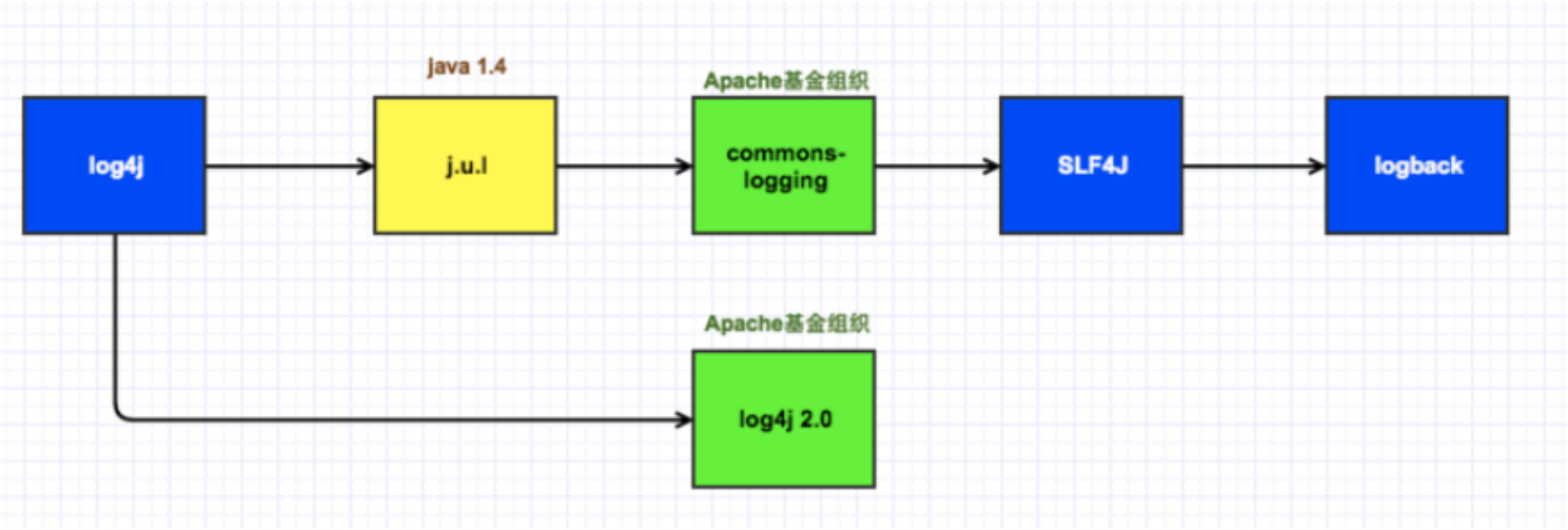
slf4j除了包含该log4j的全部特性外，还提供了parameterized logging特性。这个特性非常有用，它允许开发者在打印日志时借助{}来实现参数化打印：

```
logger.debug("The attribute value is {}", fooIns.getAttribute());
```

logback复用了slf4j的API，这意味着使用logback实际上是在使用slf4j的API，不难看出，logback同样支持parameterized logging特性。

4. 各日志框架时间线

以上日志框架，有些是为了解决现有框架的不足，有些是功能的扩展升级，有些是从头到尾重新写的，根据各自出现先后次序，可以将它们放在同一时间线上：



5. SLF4J使用方法

slf4j的使用有两种方式,一种是混合绑定(concrete-bindings), 另一种是桥接遗产(bridging-legacy).

5.1 混合绑定(concrete-bindings)

concrete-bindings模式指在新项目中即开发者直接使用sl4j的api来打印日志, 而底层绑定任意一种日志框架,如logback, log4j, j.u.l等.

混合绑定根据实现原理,基本上有两种形式, 分别为有适配器(adapter)的绑定和无适配器的绑定.

有适配器的混合绑定是指底层没有实现slf4j的接口,而是通过适配器直接调用底层日志框架的Logger, 无适配器的绑定不需要调用其它日志框架的Logger, 其本身就实现了slf4j的全部接口.

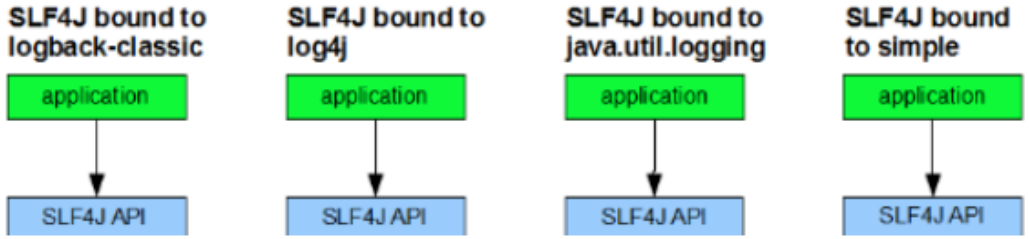
几个混合绑定的包分别是:

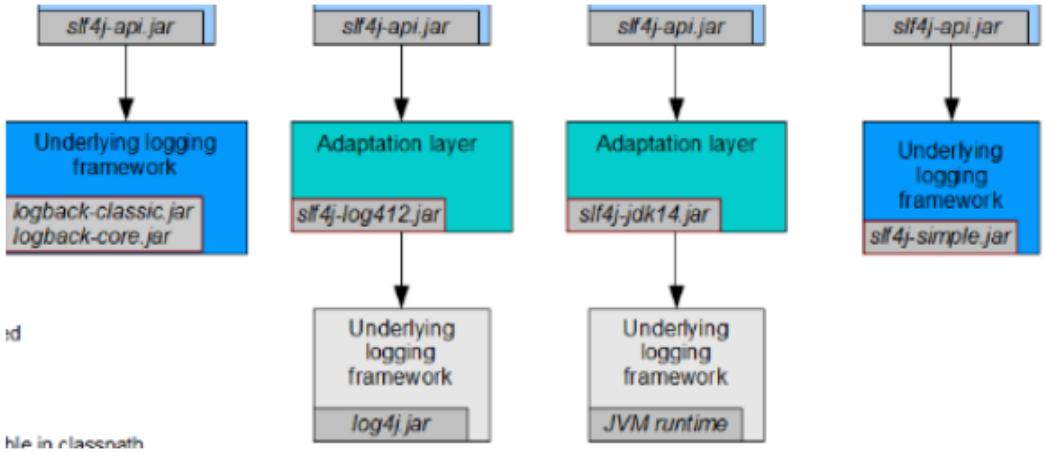
- slf4j-log4j12-1.7.21.jar(适配器, 绑定log4j, Logger由log4j-1.2.17.jar提供)
- slf4j-jdk14-1.7.21.jar(适配器, 绑定l.u.l, Logger由JVM runtime, 即j.u.l库提供)
- logback-classic-1.0.13.jar(无适配器, slf4j的一个native实现)
- slf4j-simple-1.7.21.jar(无适配器,slf4j的简单实现, 仅打印INFO及更高级别的消息, 所有输出全部重定向到System.err, 适合小应用)

以上几种绑定可以无缝切换, 不需要改动内部代码. 无论哪种绑定,均依赖slf4j-api.jar.

此外, 适配器绑定需要一种具体的日志框架, 如log4j绑定slf4j-log4j12-1.7.21.jar依赖log4j.jar, j.u.l绑定slf4j-jdk14-1.7.21.jar依赖j.u.l(java runtime提供); 无适配器的直接实现, logback-classic依赖logback-core提供底层功能, slf4j-simple则不依赖其它库.

以上四种绑定的示例图如下:





下面来分析两个典型绑定log4j和logback的用法.

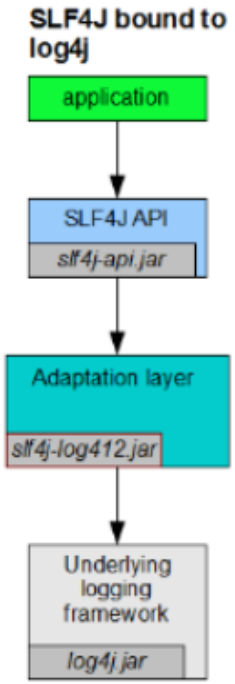
①log4j适配器绑定(slf4j-log4j12)配置:

```
<!--pom.xml-->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.21</version>
</dependency>
```

注意: 添加上述适配器绑定配置后会自动拉下来两个依赖库, 分别是slf4j-api-1.7.21.jar和log4j-1.2.17.jar
基本逻辑: 用户层 <- 中间层 <- 底层基础日志框架层

```
org.slf4j.impl.Log4jLoggerFactory <- StaticLoggerBinder.getSingleton().getLoggerFactory()<-
org.sl4j.impl.StaticLoggerBinder.getSingleton().getLoggerFactory() <- 具体的日志框库的Logger
```

其中org.slf4j.impl.Log4jLoggerFactory在应用层调用, StaticLoggerBinder在中间层实现, 获取具体的日志框库的Logger绑定实例图如下:



应用层(slf4j-api-1.7.21.jar)
用户调用org.slf4j.impl.Log4jLoggerFactory.getLogger获取底层具体的Logger:

```java
// Foo.java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
public class Foo {
    private final static Logger logger = LoggerFactory.getLogger(CommuteNaviInfoParser.class);
    public static void main() {
        logger.info("info:{}..", "hello, sl4j");
    }
}
```

适配层(slf4j-log4j12-1.7.21.jar)由应用层org.slf4j.impl.Log4jLoggerFactory.getLogger内部创建适配层的StaticLoggerBinder:

```
public static Logger getLogger(Class<?> clazz) {
    return StaticLoggerBinder.getSingleton().getLoggerFactory();
}
```

接下来直接由StaticLoggerBinder获取具体的Logger:

```
private StaticLoggerBinder() {
    loggerFactory = new Log4jLoggerFactory();
}
public Log4jLoggerFactory() {
    // force log4j to initialize
    org.apache.log4j.LogManager.getRootLogger();
}
```

注意: 各个StaticLoggerBinder均在适配层实现, 放在org.slf4j.impl中.

② slf4j绑定到logback-classic上

配置:

```
<!--pom.xml-->
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.1.7</version>
</dependency>
```

注意: 添加上述适配器绑定配置后会自动拉下来两个依赖库, 分别是slf4j-api-1.7.21.jar和logback-core-1.0.13.jar

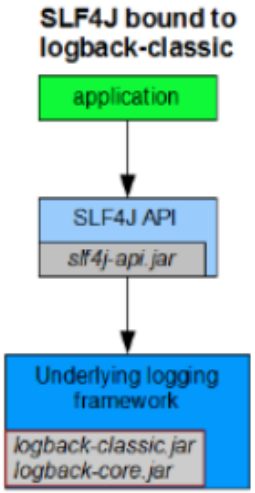
logback-classic没有适配器层, 而是在logback-classic-1.0.13.jar的ch.qos.logback.classic.Logger直接实现了slf4j的org.slf4j.Logger, 并强依赖ch.qos.logback.core中的大量基础类:

```java
import org.slf4j.LoggerFactory;
import org.slf4j.Marker;
import org.slf4j.spi.LocationAwareLogger;

import ch.qos.logback.classic.spi.ILoggingEvent;
import ch.qos.logback.classic.spi.LoggingEvent;
import ch.qos.logback.classic.util.LoggerNameUtil;
import ch.qos.logback.core.Appender;
import ch.qos.logback.core.CoreConstants;
import ch.qos.logback.core.spi.AppenderAttachable;
import ch.qos.logback.core.spi.AppenderAttachableImpl;
import ch.qos.logback.core.spi.FilterReply;

public final class Logger implements org.slf4j.Logger,
    LocationAwareLogger, AppenderAttachable<ILoggingEvent>, Serializable {}
```

绑定示例图:



5.2 桥接遗产(bridging-legacy)

桥接遗产用法主要针对历史遗留项目, 不论是用log4j写的, j.c.l写的,还是j.u.l写的, 都可以在不改动代码的情况下具有另外一种日志框架的能力.

比如,你的项目使用java提供的原生日志库j.u.l写的, 使用slf4j的bridging-legacy模式,便可在不改动一行代码的情况下瞬间具有log4j的全部特性.

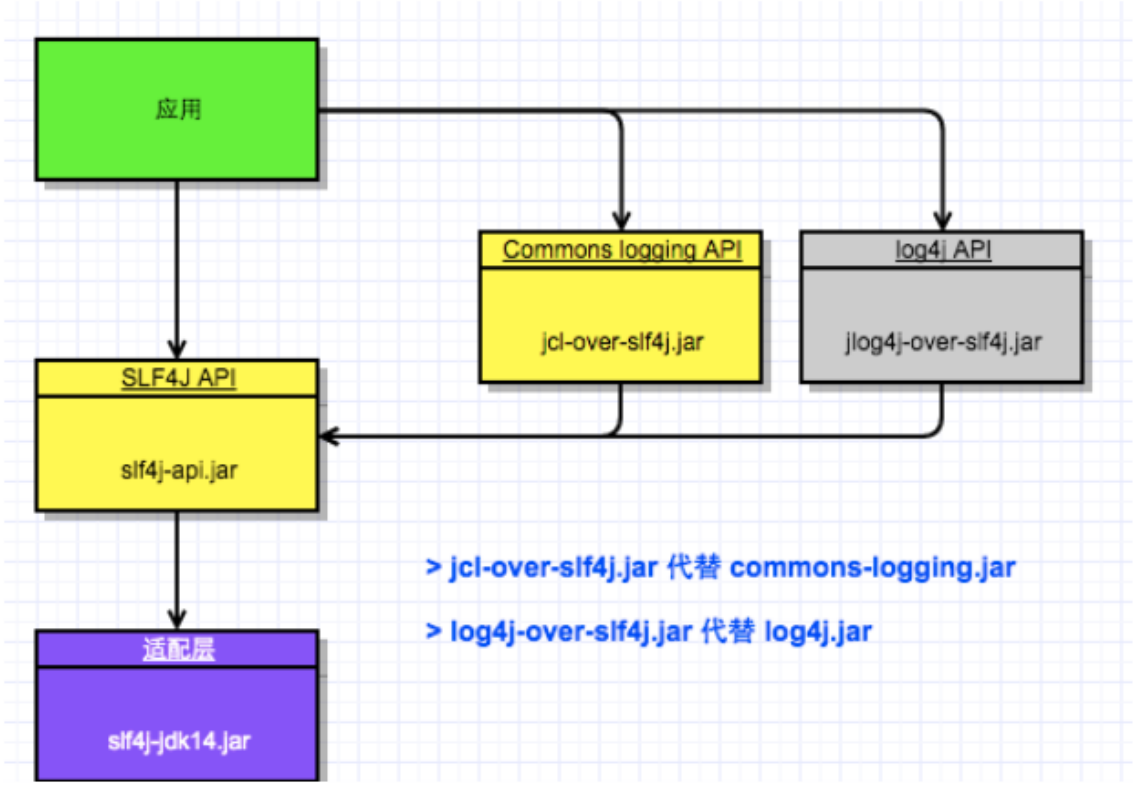
说得更直白一些,就是你的项目代码可能是5年前写的, 当时由于没得选择, 用了一个比较垃圾的日志框架, 有各种缺陷和问题, 如不能按天存储, 不能控制大小, 支持的appender很少, 无法存入数据库等. 你很想对这个已完工并在线上运行的项目进行改造, 显然, 直接改代码, 把旧的日志框架替换掉是不现实的, 因为很有可能引入不可预期的bug.

那么,如何在不修改代码的前提下, 替换掉旧的日志框架,引入更优秀且成熟的日志框架如如log4j和logback呢? slf4j的bridging-legacy模式便是为了解决这个痛点.

slf4j以slf4j-api为中间层, 将上层旧日志框架的消息转发到底层绑定的新日志框架上.

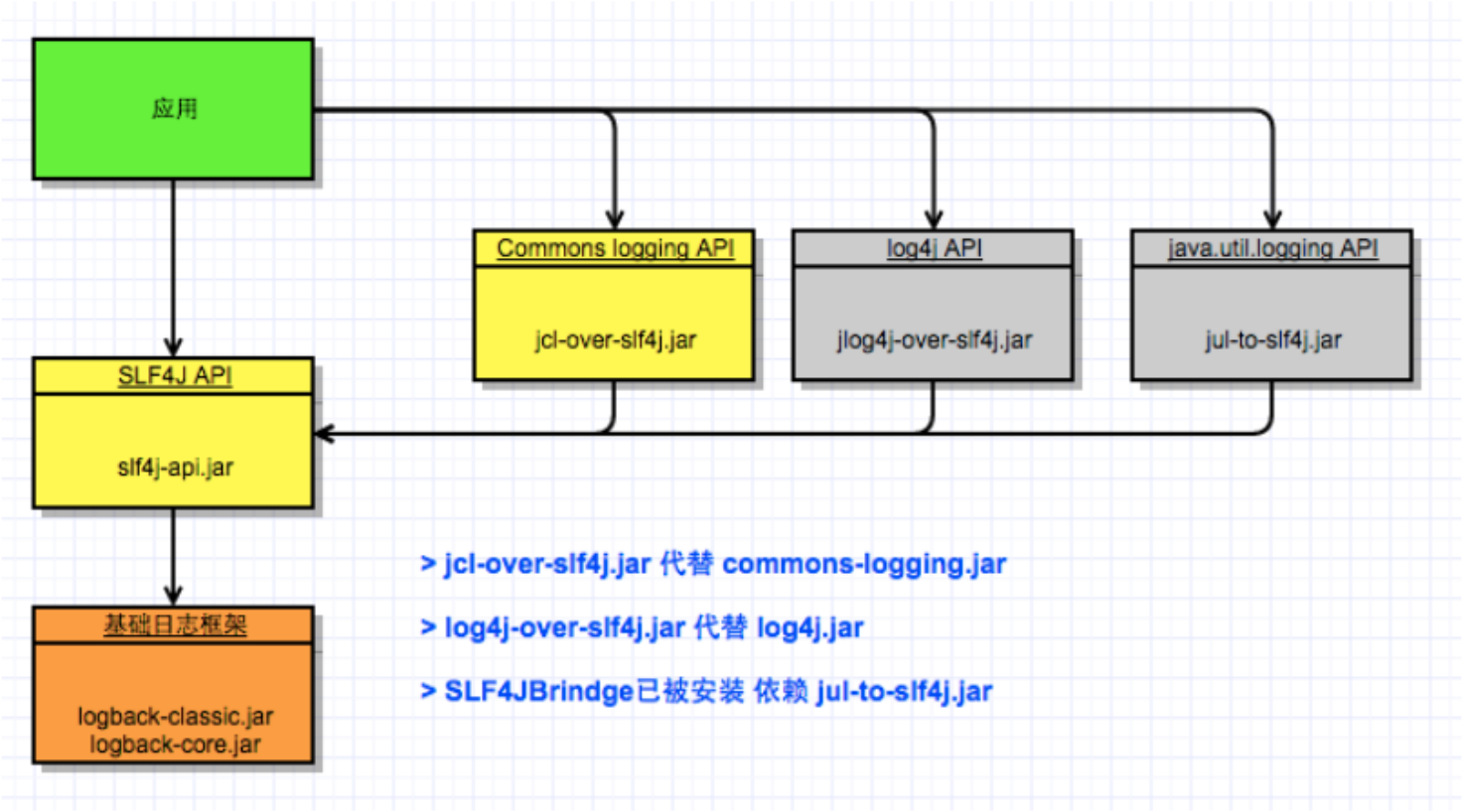
基于不同的底层框架，以SLF4J作为中转层，有如下几种组合用法:

基于j.u.l的facade使用

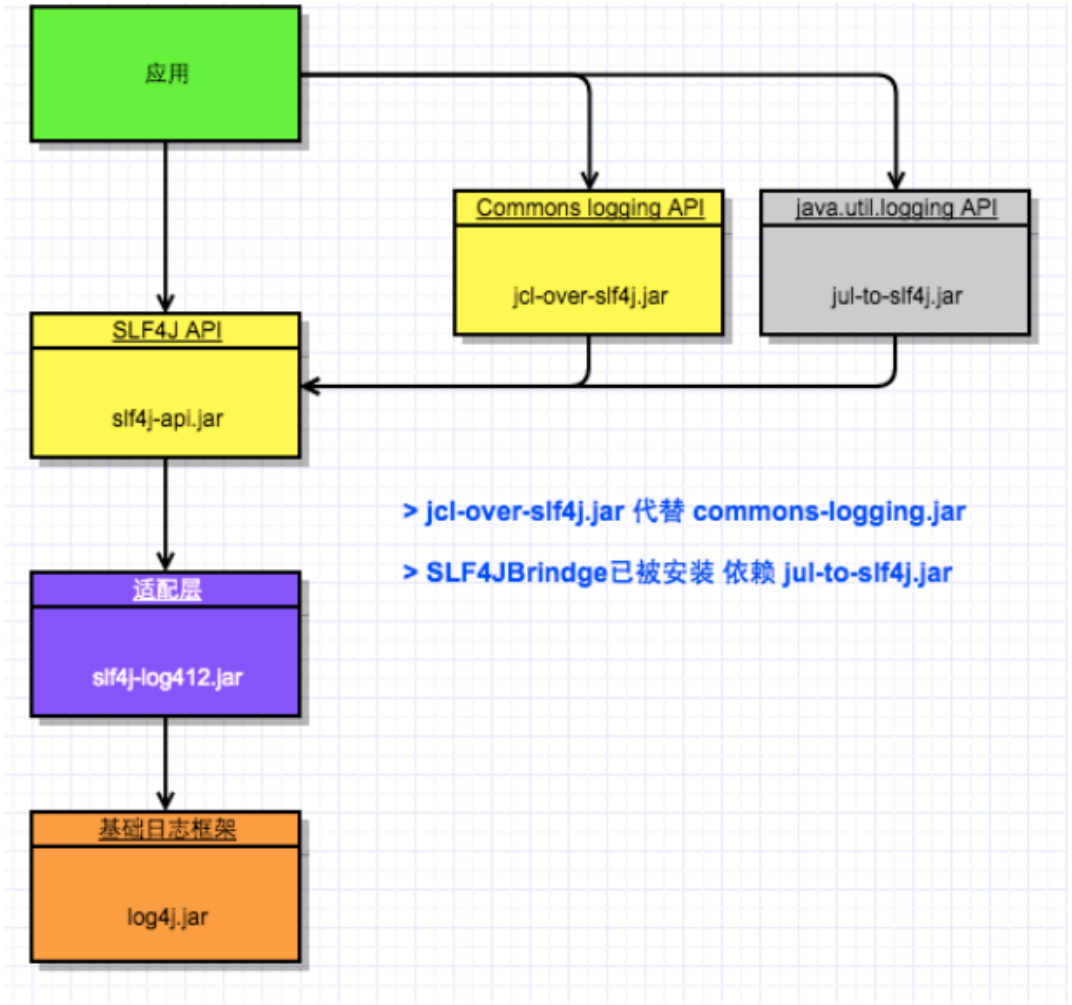


上述facade将slf4j-api.jar绑定到底层基础日志库j.u.l(jvm runtime)上. slf4j-api和底层日志库的Logger通过适配器连接.

基于logback-classic的facade使用



基于log4j的facade使用



举例说明上述facade的使用, 以便于大家理解.
假如我有一个已完成的使用了旧日志框架commons-loggings的项目,现在想把它替换成log4j以获得更多更好的特性.
项目的maven旧配置如下:

```
<dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.2</version>
</dependency>
```

项目代码:

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * Created by xialeizhou on 16/9/20.
 */
public class MainTest {
    private static Log logger = LogFactory.getLog(MainTest.class);

    public static void main(String[] args) throws InterruptedException {
        logger.info("hello,world");
    }
}
```

项目打印的基于commons-logging的日志显示在console上,具体如下:

```
十月 23, 2016 6:52:00 下午 MainTest main
信息: hello,world
```

下面我们对项目改造, 将commongs-logging框架的日志转发到log4j上. 改造很简单, 我们将commongs-logging依赖删除, 替换为相应的facade(此处为jcl-over-slf4j.jar), 并在facade下面挂一个5.1的混合绑定即可.

具体来讲, 将commons-logging.jar替换成jcl-over-slf4j.jar, 并加入适配器slf4j-log412.jar(注意, 加入slf4j-log412.jar后会自动pull下来另外两个jar包), 所以实际最终只需添加facadejcl-over-slf4j.jar和5.1节混合绑定中相同的jar包slf4j-log412.jar即可.

改造后的maven配置:

```
<!--facade-->
<dependency>
```

```
    <groupId>org.slf4j</groupId>
    <artifactId>jcl-over-slf4j</artifactId>
    <version>1.7.21</version>
</dependency>

<!--binding-->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.21</version>
</dependency>
```

现在, 我们的旧项目在没有改一行代码的情况下具有了log4j的全部特性, 下面进行测试.
在resources/下新建一个log4j.properties文件, 对commongs-logging库的日志输出进行定制化:

```
# Root logger option
log4j.rootLogger=INFO, stdout, fout

# Redirect log messages to console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.Threshold = INFO
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

# add a FileAppender to the logger fout
log4j.appender.fout=org.apache.log4j.FileAppender
# create a log file
log4j.appender.fout.File=royce-testing.log
log4j.appender.fout.layout=org.apache.log4j.PatternLayout
# use a more detailed message pattern
log4j.appender.fout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n
```

重新编译运行, console输出变为:

```
2016-10-23 19:26:15 INFO MainTest:11 - hello,world
```

同时在当前目录生成了一个日志文件:

```
% cat royce-testing.log
INFO    2016-10-23 19:26:15,341    0    MainTest    [main]    hello,world
```

可见, 基于facade的日志框架桥接已经生效, 我们再不改动代码的前提下,让commons-logging日志框架具有了log4j12的全部特性.

6. 参考文献

1. why-not-use-java-util-logging
2. https://logback.qos.ch/reasonsToSwitch.html
3. log4j-vs-logback
4. whats-up-with-logging-in-java
5. https://blog.frankel.ch/thoughts-on-java-logging-and-slf4j/
6. https://en.wikipedia.org/wiki/Java_logging_framework
7. https://www.slf4j.org/faq.html

初识Spring Boot框架

weixin_30908941的博客 1806

前面的铺垫文章已经连着写了六篇了，主要是介绍了Spring和SpringMVC框架，小伙伴们在学习的过程中大概也发现了这两个框架需要我们手动配置的地...

优质评论可以帮助作者获得更高权重

评论