日志技术（下）

0. 学习目标

1. 日志门面和日志体系
2. SLF4J
3. logback的使用
4. log4j2的使用
5. SpringBoot中的日志使用

1. 日志门面

当我们的系统变的更加复杂的时候，我们的日志就容易发生混乱。随着系统开发的进行，可能会更新不同的日志框架，造成当前系统中存在不同的日志依赖，让我们难以统一的管理和控制。就算我们强制要求所有的模块使用相同的日志框架，系统中也难以避免使用其他类似spring,mybatis等其他的第三方框架，它们依赖于我们规定不同的日志框架，而且他们自身的日志系统就有着不一致性，依然会出来日志体系的混乱。

所以我们需要借鉴JDBC的思想，为日志系统也提供一套门面，那么我们就可以面向这些接口规范来开发，避免了直接依赖具体的日志框架。这样我们的系统在日志中，就存在了日志的门面和日志的实现。

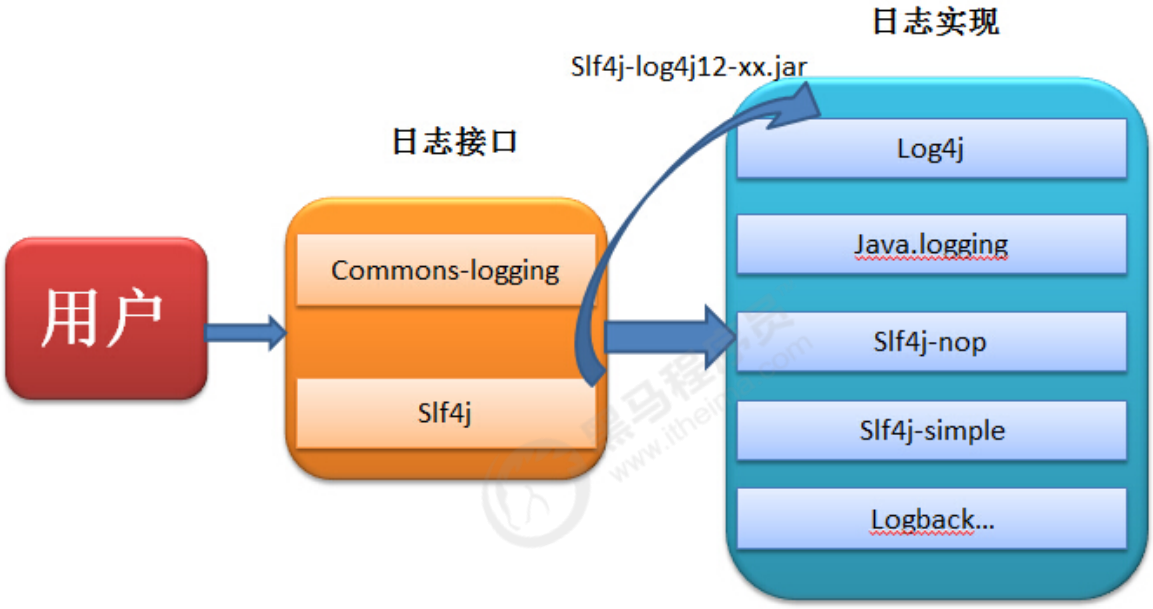
常见的日志门面 ：

JCL、slf4j

常见的日志实现：

JUL、log4j、logback、log4j2

日志门面和日志实现的关系：



日志框架出现的历史顺序：

log4j -->JUL-->JCL--> slf4j --> logback --> log4j2

2. SLF4J的使用

简单日志门面(Simple Logging Facade For Java) SLF4J主要是为了给Java日志访问提供一套标准、规范的API框架，其主要意义在于提供接口，具体的实现可以交由其他日志框架，例如log4j和logback等。当然slf4j自己也提供了功能较为简单的实现，但是一般很少用到。对于一般的Java项目而言，日志框架会选择slf4j-api作为门面，配上具体的实现框架（log4j、logback等），中间使用桥接器完成桥接。

官方网站： https://www.slf4j.org/

SLF4J是目前市面上最流行的日志门面。现在的项目中，基本上都是使用SLF4J作为我们的日志系统。

SLF4J日志门面主要提供两大功能：

1. 日志框架的绑定
2. 日志框架的桥接

2.1 SLF4J入门

1. 添加依赖

```xml
<!--slf4j core 使用slf4j必须添加-->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.27</version>
</dependency>
<!--slf4j 自带的简单日志实现 -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.27</version>
</dependency>
```

2. 编写代码

```java
public class Slf4jTest {

    // 声明日志对象
    public final static Logger LOGGER =
LoggerFactory.getLogger(Slf4jTest.class);

    @Test
    public void testQuick() throws Exception {
        //打印日志信息
        LOGGER.error("error");
        LOGGER.warn("warn");
        LOGGER.info("info");
        LOGGER.debug("debug");
        LOGGER.trace("trace");

        // 使用占位符输出日志信息
        String name = "jack";
        Integer age = 18;
        LOGGER.info("用户：{},{}", name, age);

        // 将系统异常信息写入日志
        try {
            int i = 1 / 0;
        } catch (Exception e) {
            // e.printStackTrace();
            LOGGER.info("出现异常：", e);
        }
    }
}
```

为什么要使用SLF4J作为日志门面？

```
* 1. 使用SLF4J框架，可以在部署时迁移到所需的日志记录框架。

* 2. SLF4J提供了对所有流行的日志框架的绑定，例如log4j，JUL，Simple logging和NOP。因此可以在部署时切换到任何这些流行的框架。

* 3. 无论使用哪种绑定，SLF4J都支持参数化日志记录消息。由于SLF4J将应用程序和日志记录框架分离，因此可以轻松编写独立于日志记录框架的应用程序。而无需担心用于编写应用程序的日志记录框架。

* 4. SLF4J提供了一个简单的Java工具，称为迁移器。使用此工具，可以迁移现有项目，这些项目使用日志框架(如Jakarta Commons Logging(JCL)或log4j或Java.util.logging(JUL))到SLF4J。
```

2.2 绑定日志的实现（Binding）

如前所述，SLF4j支持各种日志框架。SLF4j发行版附带了几个称为“SLF4j绑定”的jar文件，每个绑定对应一个受支持的框架。

使用slf4j的日志绑定流程:

1. 添加slf4j-api的依赖
2. 使用slf4j的API在项目中进行统一的日志记录
3. 绑定具体的日志实现框架
    1. 绑定已经实现了slf4j的日志框架,直接添加对应依赖
    2. 绑定没有实现slf4j的日志框架,先添加日志的适配器,再添加实现类的依赖
4. slf4j有且仅有一个日志实现框架的绑定（如果出现多个默认使用第一个依赖日志实现）

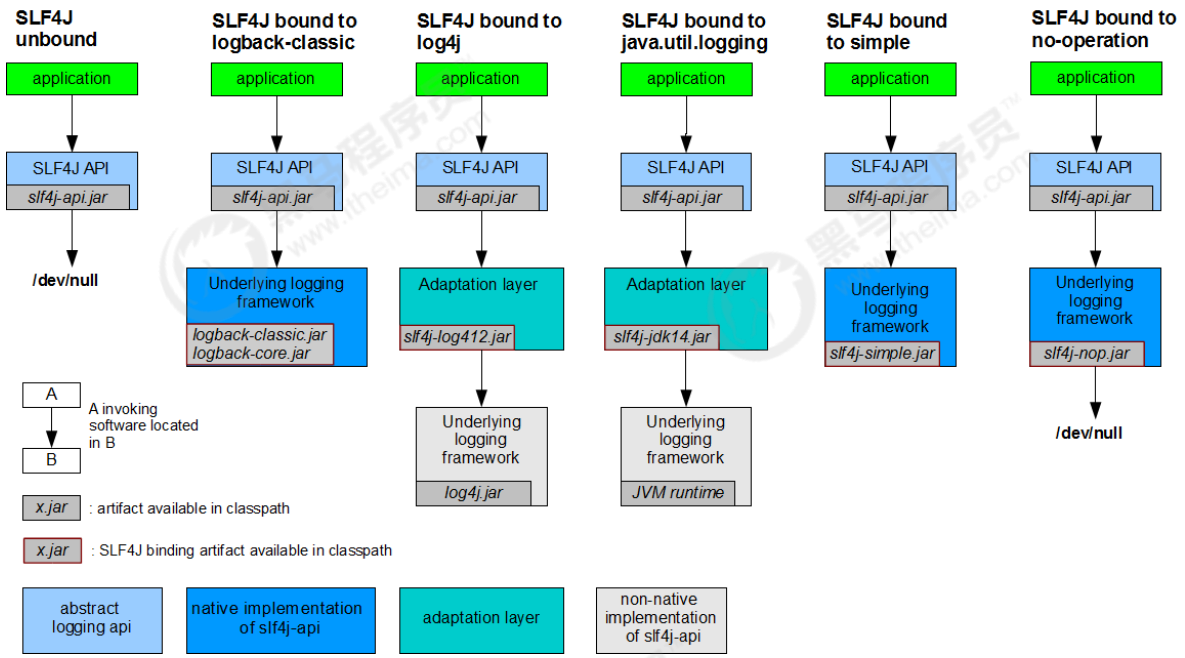
通过maven引入常见的日志实现框架：

```
<!--slf4j core 使用slf4j必须添加-->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.27</version>
</dependency>

<!-- log4j-->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.27</version>
</dependency>
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>

<!-- jul  -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-jdk14</artifactId>
    <version>1.7.27</version>
</dependency>
```

```
<!--jcl -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-jcl</artifactId>
    <version>1.7.27</version>
</dependency>

<!-- nop -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-nop</artifactId>
    <version>1.7.27</version>
</dependency>
```

要切换日志框架，只需替换类路径上的slf4j绑定。例如，要从java.util.logging切换到log4j，只需将slf4j-jdk14-1.7.27.jar替换为slf4j-log4j12-1.7.27.jar即可。

SLF4J不依赖于任何特殊的类装载。实际上，每个SLF4J绑定*在编译时*都是硬连线的，以使用一个且只有一个特定的日志记录框架。例如，slf4j-log4j12-1.7.27.jar绑定在编译时绑定以使用log4j。在您的代码中，除了*slf4j-api-1.7.27.jar之外*，您只需将您选择的**一个且只有一个**绑定放到相应的类路径位置。不要在类路径上放置多个绑定。以下是一般概念的图解说明。

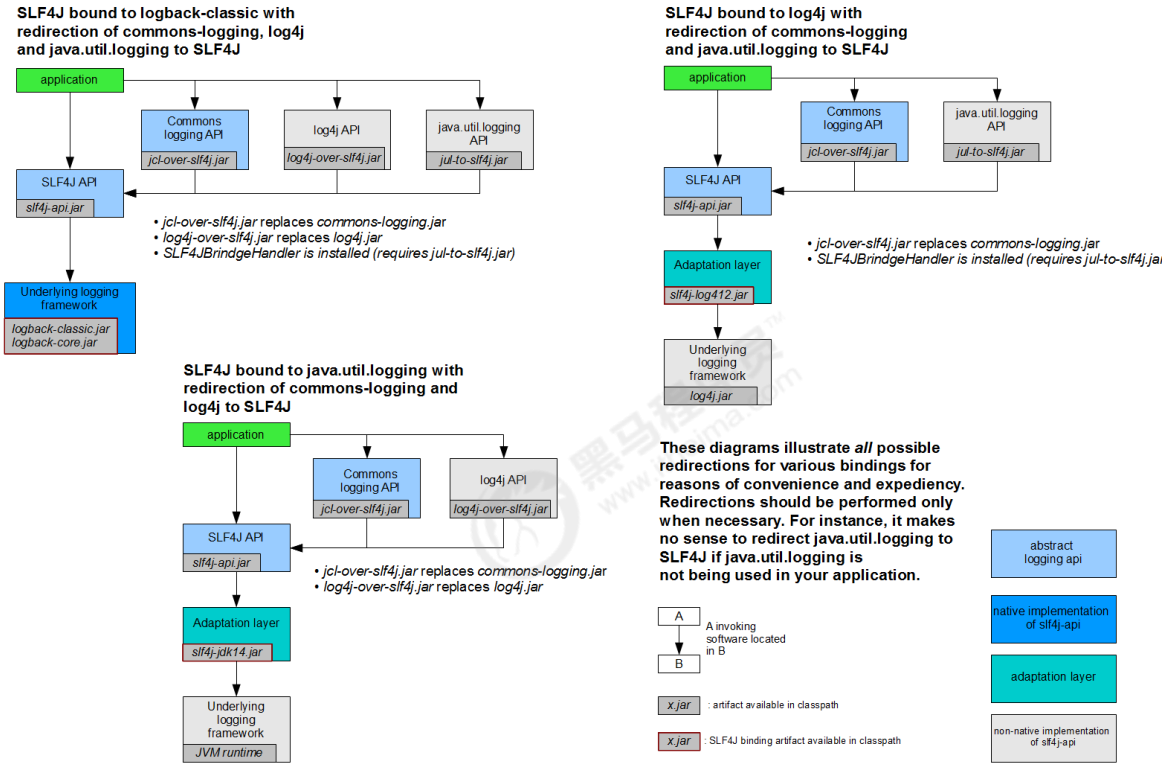


2.3 桥接旧的日志框架（Bridging）

通常，您依赖的某些组件依赖于SLF4J以外的日志记录API。您也可以假设这些组件在不久的将来不会切换到SLF4J。为了解决这种情况，SLF4J附带了几个桥接模块，这些模块将对log4j，JCL和java.util.logging API的调用重定向，就好像它们是对SLF4J API一样。

桥接解决的是项目中日志的遗留问题，当系统中存在之前的日志API，可以通过桥接转换到slf4j的实现

1. 先去除之前老的日志框架的依赖
2. 添加SLF4J提供的桥接组件
3. 为项目添加SLF4J的具体实现



迁移的方式：

如果我们要使用SLF4j的桥接器，替换原有的日志框架，那么我们需要做的第一件事情，就是删除掉原有项目中的日志框架的依赖。然后替换成SLF4j提供的桥接器。

```
<!-- log4j-->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>log4j-over-slf4j</artifactId>
    <version>1.7.27</version>
</dependency>

<!-- jul  -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jul-to-slf4j</artifactId>
    <version>1.7.27</version>
</dependency>

<!--jcl -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jcl-over-slf4j</artifactId>
    <version>1.7.27</version>
</dependency>
```

注意问题：

1. jcl-over-slf4j.jar和 *slf4j-jcl.jar*不能同时部署。前一个jar文件将导致JCL将日志系统的选择委托给SLF4J，后一个jar文件将导致SLF4J将日志系统的选择委托给JCL，从而导致无限循环。
2. log4j-over-slf4j.jar和slf4j-log4j12.jar不能同时出现

3. jul-to-slf4j.jar和slf4j-jdk14.jar不能同时出现
4. 所有的桥接都只对Logger日志记录器对象有效，如果程序中调用了内部的配置类或者是Appender,Filter等对象，将无法产生效果。

2.4 SLF4J原理解析

1. SLF4J通过LoggerFactory加载日志具体的实现对象。
2. LoggerFactory在初始化的过程中，会通过performInitialization()方法绑定具体的日志实现。
3. 在绑定具体实现的时候，通过类加载器，加载org/slf4j/impl/StaticLoggerBinder.class
4. 所以，只要是一个日志实现框架，在org.slf4j.impl包中提供一个自己的StaticLoggerBinder类，在其中提供具体日志实现的LoggerFactory就可以被SLF4J所加载

3. Logback的使用

Logback是由log4j创始人设计的另一个开源日志组件，性能比log4j要好。

官方网站：https://logback.qos.ch/index.html

Logback主要分为三个模块：

- 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，同时它完整实现了slf4j API
- 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

后续的日志代码都是通过SLF4J日志门面搭建日志系统，所以在代码是没有区别，主要是通过修改配置文件和pom.xml依赖

3.1 logback入门

1. 添加依赖

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.25</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.3</version>
</dependency>
```

2. java代码

```
//定义日志对象
public final static Logger LOGGER =
LoggerFactory.getLogger(LogBackTest.class);

@Test
public void testSlf4j(){
    //打印日志信息
    LOGGER.error("error");
    LOGGER.warn("warn");
    LOGGER.info("info");
    LOGGER.debug("debug");
    LOGGER.trace("trace");
}
```

3.2 logback配置

logback会依次读取以下类型配置文件：

- logback.groovy
- logback-test.xml
- logback.xml 如果均不存在会采用默认配置

1. logback组件之间的关系
    1. Logger:日志的记录器，把它关联到应用的对应的context上后，主要用于存放日志对象，也可以定义日志类型、级别。
    2. Appender:用于指定日志输出的目的地，目的地可以是控制台、文件、数据库等等。
    3. Layout:负责把事件转换成字符串，格式化的日志信息的输出。在logback中Layout对象被封装在encoder中。
2. 基本配置信息

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <!--
        日志输出格式：
                        %-5level
            %d{yyyy-MM-dd HH:mm:ss.SSS}日期
                        %c类的完整名称
            %M为method
            %L为行号
            %thread线程名称
                        %m或者%msg为信息
            %n换行
    -->
    <!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度
%msg：日志消息，%n是换行符-->
    <property name="pattern" value="%d{yyyy-MM-dd HH:mm:ss.SSS} %c [%thread]
%-5level  %msg%n"/>


    <!--
        Appender: 设置日志信息的去向,常用的有以下几个
            ch.qos.logback.core.ConsoleAppender (控制台)
```

```
            ch.qos.logback.core.rolling.RollingFileAppender (文件大小到达指定尺寸的时候产生一个新文件)
            ch.qos.logback.core.FileAppender (文件)
    -->
    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <!--输出流对象 默认 System.out 改为 System.err-->
        <target>System.err</target>
        <!--日志格式配置-->
        <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <pattern>${pattern}</pattern>
        </encoder>
    </appender>

    <!--
        用来设置某一个包或者具体的某一个类的日志打印级别、以及指定<appender>。
        <loger>仅有一个name属性，一个可选的level和一个可选的addtivity属性
        name:
            用来指定受此logger约束的某一个包或者具体的某一个类。
        level:
            用来设置打印级别，大小写无关：TRACE, DEBUG, INFO, WARN, ERROR, ALL 和 OFF,
            如果未设置此属性，那么当前logger将会继承上级的级别。
        additivity:
            是否向上级loger传递打印信息。默认是true。
        <logger>可以包含零个或多个<appender-ref>元素，标识这个appender将会添加到这个logger
    -->
    <!--
        也是<logger>元素，但是它是根logger。默认debug
        level:用来设置打印级别，大小写无关：TRACE, DEBUG, INFO, WARN, ERROR, ALL 和 OFF,
        <root>可以包含零个或多个<appender-ref>元素，标识这个appender将会添加到这个logger。
    -->
    <root level="ALL">
        <appender-ref ref="console"/>
    </root>

</configuration>
```

3. FileAppender配置

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

    <!-- 自定义属性 可以通过${name}进行引用-->
    <property name="pattern" value="[%-5level] %d{yyyy-MM-dd HH:mm:ss} %c %M %L [%thread] %m %n"/>
    <!--
       日志输出格式:
            %d{pattern}日期
            %m或者%msg为信息
            %M为method
            %L为行号
```

```
            %c类的完整名称
            %thread线程名称
            %n换行
            %-5level
    -->
    <!-- 日志文件存放目录 -->
    <property name="log_dir" value="d:/logs"></property>

    <!--控制台输出appender对象-->
    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <!--输出流对象 默认 System.out 改为 System.err-->
        <target>System.err</target>
        <!--日志格式配置-->
        <encoder
class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <pattern>${pattern}</pattern>
        </encoder>
    </appender>

    <!--日志文件输出appender对象-->
    <appender name="file" class="ch.qos.logback.core.FileAppender">
        <!--日志格式配置-->
        <encoder
class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <pattern>${pattern}</pattern>
        </encoder>
        <!--日志输出路径-->
        <file>${log_dir}/logback.log</file>
    </appender>


    <!-- 生成html格式appender对象 -->
    <appender name="htmlFile" class="ch.qos.logback.core.FileAppender">
        <!--日志格式配置-->
        <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
            <layout class="ch.qos.logback.classic.html.HTMLLayout">
                <pattern>%level%d{yyyy-MM-dd
HH:mm:ss}%c%M%L%thread%m</pattern>
            </layout>
        </encoder>
        <!--日志输出路径-->
        <file>${log_dir}/logback.html</file>
    </appender>

    <!--RootLogger对象-->
    <root level="all">
        <appender-ref ref="console"/>
        <appender-ref ref="file"/>
        <appender-ref ref="htmlFile"/>
    </root>

</configuration>
```

4. RollingFileAppender配置

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

```
    <!-- 自定义属性 可以通过${name}进行引用-->
    <property name="pattern" value="[%-5level] %d{yyyy-MM-dd HH:mm:ss} %c %M
%L [%thread] %m %n"/>
    <!--
       日志输出格式:
            %d{pattern}日期
            %m或者%msg为信息
            %M为method
            %L为行号
            %c类的完整名称
            %thread线程名称
            %n换行
            %-5level
    -->
    <!-- 日志文件存放目录 -->
    <property name="log_dir" value="d:/logs"></property>

    <!--控制台输出appender对象-->
    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <!--输出流对象 默认 System.out 改为 System.err-->
        <target>System.err</target>
        <!--日志格式配置-->
        <encoder
class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <pattern>${pattern}</pattern>
        </encoder>
    </appender>

    <!-- 日志文件拆分和归档的appender对象-->
    <appender name="rollFile"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <!--日志格式配置-->
        <encoder
class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <pattern>${pattern}</pattern>
        </encoder>
        <!--日志输出路径-->
        <file>${log_dir}/roll_logback.log</file>
        <!--指定日志文件拆分和压缩规则-->
        <rollingPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
            <!--通过指定压缩文件名称，来确定分割文件方式-->
            <fileNamePattern>${log_dir}/rolling.%d{yyyy-MM-
dd}.log%i.gz</fileNamePattern>
            <!--文件拆分大小-->
            <maxFileSize>1MB</maxFileSize>
        </rollingPolicy>
    </appender>

    <!--RootLogger对象-->
    <root level="all">
        <appender-ref ref="console"/>
        <appender-ref ref="rollFile"/>
    </root>

</configuration>
```

5. Filter和异步日志配置

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

    <!-- 自定义属性 可以通过${name}进行引用-->
    <property name="pattern" value="[%-5level] %d{yyyy-MM-dd HH:mm:ss} %c %M
%L [%thread] %m %n"/>
    <!--
       日志输出格式:
            %d{pattern}日期
            %m或者%msg为信息
            %M为method
            %L为行号
            %c类的完整名称
            %thread线程名称
            %n换行
            %-5level
    -->
    <!-- 日志文件存放目录 -->
    <property name="log_dir" value="d:/logs/"></property>

    <!--控制台输出appender对象-->
    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <!--输出流对象 默认 System.out 改为 System.err-->
        <target>System.err</target>
        <!--日志格式配置-->
        <encoder
class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <pattern>${pattern}</pattern>
        </encoder>
    </appender>

    <!-- 日志文件拆分和归档的appender对象-->
    <appender name="rollFile"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <!--日志格式配置-->
        <encoder
class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <pattern>${pattern}</pattern>
        </encoder>
        <!--日志输出路径-->
        <file>${log_dir}roll_logback.log</file>
        <!--指定日志文件拆分和压缩规则-->
        <rollingPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
            <!--通过指定压缩文件名称，来确定分割文件方式-->
            <fileNamePattern>${log_dir}rolling.%d{yyyy-MM-
dd}.log%i.gz</fileNamePattern>
            <!--文件拆分大小-->
            <maxFileSize>1MB</maxFileSize>
        </rollingPolicy>
        <!--filter配置-->
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <!--设置拦截日志级别-->
            <level>error</level>
            <onMatch>ACCEPT</onMatch>
```

```
            <onMismatch>DENY</onMismatch>
        </filter>
    </appender>

    <!--异步日志-->
     <appender name="async" class="ch.qos.logback.classic.AsyncAppender">
         <appender-ref ref="rollFile"/>
     </appender>

    <!--RootLogger对象-->
    <root level="all">
        <appender-ref ref="console"/>
        <appender-ref ref="async"/>
    </root>

    <!--自定义logger additivity表示是否从 rootLogger继承配置-->
    <logger name="com.itheima" level="debug" additivity="false">
        <appender-ref ref="console"/>
    </logger>

</configuration>
```

6. 官方提供的log4j.properties转换成logback.xml

   https://logback.qos.ch/translator/

3.3 logback-access的使用

logback-access模块与Servlet容器（如Tomcat和Jetty）集成，以提供HTTP访问日志功能。我们可以使用logback-access模块来替换tomcat的访问日志。

1. 将logback-access.jar与logback-core.jar复制到$TOMCAT_HOME/lib/目录下
2. 修改$TOMCAT_HOME/conf/server.xml中的Host元素中添加：

```
<Valve className="ch.qos.logback.access.tomcat.LogbackValve" />
```

3. logback默认会在$TOMCAT_HOME/conf下查找文件 logback-access.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <!-- always a good activate OnConsoleStatusListener -->
  <statusListener
class="ch.qos.logback.core.status.OnConsoleStatusListener"/>

  <property name="LOG_DIR" value="${catalina.base}/logs"/>

  <appender name="FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>${LOG_DIR}/access.log</file>
    <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <fileNamePattern>access.%d{yyyy-MM-dd}.log.zip</fileNamePattern>
    </rollingPolicy>
```

```
    <encoder>
        <!-- 访问日志的格式 -->
      <pattern>combined</pattern>
    </encoder>
  </appender>

  <appender-ref ref="FILE"/>
</configuration>
```

4. 官方配置： https://logback.qos.ch/access.html#configuration

4. log4j2的使用

Apache Log4j 2是对Log4j的升级版，参考了logback的一些优秀的设计，并且修复了一些问题，因此带来了一些重大的提升，主要有：

- 异常处理，在logback中，Appender中的异常不会被应用感知到，但是在log4j2中，提供了一些异常处理机制。
- 性能提升， log4j2相较于log4j 和logback都具有很明显的性能提升，后面会有官方测试的数据。
- 自动重载配置，参考了logback的设计，当然会提供自动刷新参数配置，最实用的就是我们在生产上可以动态的修改日志的级别而不需要重启应用。
- 无垃圾机制，log4j2在大部分情况下，都可以使用其设计的一套无垃圾机制，避免频繁的日志收集导致的jvm gc。

官网： https://logging.apache.org/log4j/2.x/

4.1 Log4j2入门

目前市面上最主流的日志门面就是SLF4J，虽然Log4j2也是日志门面，因为它的日志实现功能非常强大，性能优越。所以大家一般还是将Log4j2看作是日志的实现，Slf4j + Log4j2应该是未来的大势所趋。

1. 添加依赖

```
<!-- Log4j2 门面API-->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.11.1</version>
</dependency>
<!-- Log4j2 日志实现 -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.11.1</version>
</dependency>
```

2. JAVA代码

```
public class Log4j2Test {
```

```
    // 定义日志记录器对象
    public static final Logger LOGGER =
LogManager.getLogger(Log4j2Test.class);

    @Test
    public void testQuick() throws Exception {
        LOGGER.fatal("fatal");
        LOGGER.error("error");
        LOGGER.warn("warn");
        LOGGER.info("info");
        LOGGER.debug("debug");
        LOGGER.trace("trace");
    }
}
```

3. 使用slf4j作为日志的门面,使用log4j2作为日志的实现

```
<!-- Log4j2 门面API-->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.11.1</version>
</dependency>
<!-- Log4j2 日志实现 -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.11.1</version>
</dependency>

<!--使用slf4j作为日志的门面,使用log4j2来记录日志 -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.25</version>
</dependency>
<!--为slf4j绑定日志实现  log4j2的适配器 -->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j-impl</artifactId>
    <version>2.10.0</version>
</dependency>
```

4.2 Log4j2配置

log4j2默认加载classpath下的 `log4j2.xml` 文件中的配置。

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn" monitorInterval="5">

    <properties>
        <property name="LOG_HOME">D:/logs</property>
    </properties>
```

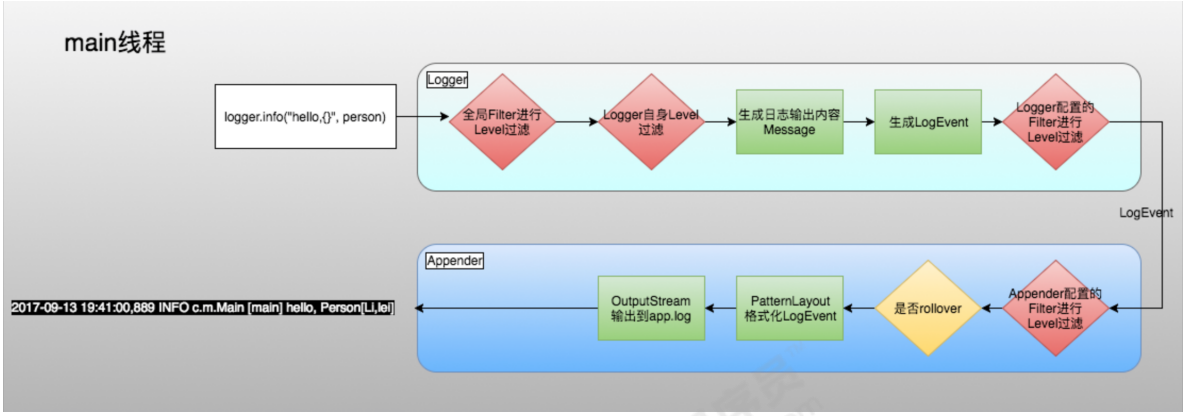
```
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] [%-5level] %c{36}:%L -
-- %m%n" />
        </Console>

        <File name="file" fileName="${LOG_HOME}/myfile.log">
            <PatternLayout pattern="[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%-5level] %l
%c{36} - %m%n" />
        </File>

        <RandomAccessFile name="accessFile" fileName="${LOG_HOME}/myAcclog.log">
            <PatternLayout pattern="[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%-5level] %l
%c{36} - %m%n" />
        </RandomAccessFile>

        <RollingFile name="rollingFile" fileName="${LOG_HOME}/myrollog.log"
                     filePattern="D:/logs/$${date:yyyy-MM-dd}/myrollog-%d{yyyy-
MM-dd-HH-mm}-%i.log">
            <ThresholdFilter level="debug" onMatch="ACCEPT" onMismatch="DENY" />
            <PatternLayout pattern="[%d{yyyy-MM-dd HH:mm:ss.SSS}] [%-5level] %l
%c{36} - %msg%n" />
            <Policies>
                <OnStartupTriggeringPolicy />
                <SizeBasedTriggeringPolicy size="10 MB" />
                <TimeBasedTriggeringPolicy />
            </Policies>
            <DefaultRolloverStrategy max="30" />
        </RollingFile>

    </Appenders>
    <Loggers>
        <Root level="trace">
            <AppenderRef ref="Console" />
        </Root>
    </Loggers>
</Configuration>
```

4.3 Log4j2异步日志

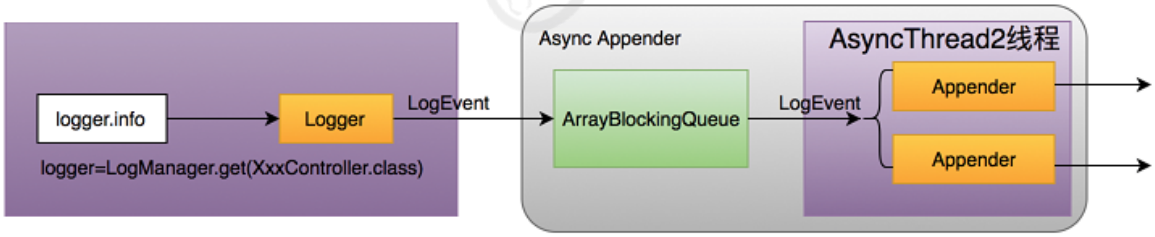
异步日志

log4j2最大的特点就是异步日志，其性能的提升主要也是从异步日志中受益，我们来看看如何使用log4j2的异步日志。

- **同步日志**



- **异步日志**



Log4j2提供了两种实现日志的方式，一个是通过AsyncAppender，一个是通过AsyncLogger，分别对应前面我们说的Appender组件和Logger组件。

注意：配置异步日志需要添加依赖

```
<!--异步日志依赖-->
<dependency>
    <groupId>com.lmax</groupId>
    <artifactId>disruptor</artifactId>
    <version>3.3.4</version>
</dependency>
```

1. AsyncAppender方式

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn">
    <properties>
        <property name="LOG_HOME">D:/logs</property>
    </properties>
    <Appenders>
        <File name="file" fileName="${LOG_HOME}/myfile.log">
            <PatternLayout>
                <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
            </PatternLayout>
        </File>
        <Async name="Async">
            <AppenderRef ref="file"/>
        </Async>
    </Appenders>
    <Loggers>
        <Root level="error">
```

```
            <AppenderRef ref="Async"/>
        </Root>
    </Loggers>
</Configuration
```

2. AsyncLogger方式

   AsyncLogger才是log4j2 的重头戏，也是官方推荐的异步方式。它可以使得调用Logger.log返回的更快。你可以有两种选择：全局异步和混合异步。

   - **全局异步**就是，所有的日志都异步的记录，在配置文件上不用做任何改动，只需要添加一个`log4j2.component.properties`配置；

     ```
     Log4jContextSelector=org.apache.logging.log4j.core.async.AsyncLoggerCon
     textSelector
     ```

   - **混合异步**就是，你可以在应用中同时使用同步日志和异步日志，这使得日志的配置方式更加灵活。

     ```
     <?xml version="1.0" encoding="UTF-8"?>
     <Configuration status="WARN">
       <properties>
         <property name="LOG_HOME">D:/logs</property>
       </properties>
       <Appenders>
         <File name="file" fileName="${LOG_HOME}/myfile.log">
           <PatternLayout>
             <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
           </PatternLayout>
         </File>
         <Async name="Async">
           <AppenderRef ref="file"/>
         </Async>
       </Appenders>
       <Loggers>
         <AsyncLogger name="com.itheima" level="trace"
     includeLocation="false" additivity="false">
           <AppenderRef ref="file"/>
         </AsyncLogger>

         <Root level="info" includeLocation="true">
           <AppenderRef ref="file"/>
         </Root>
       </Loggers>
     </Configuration>
     ```

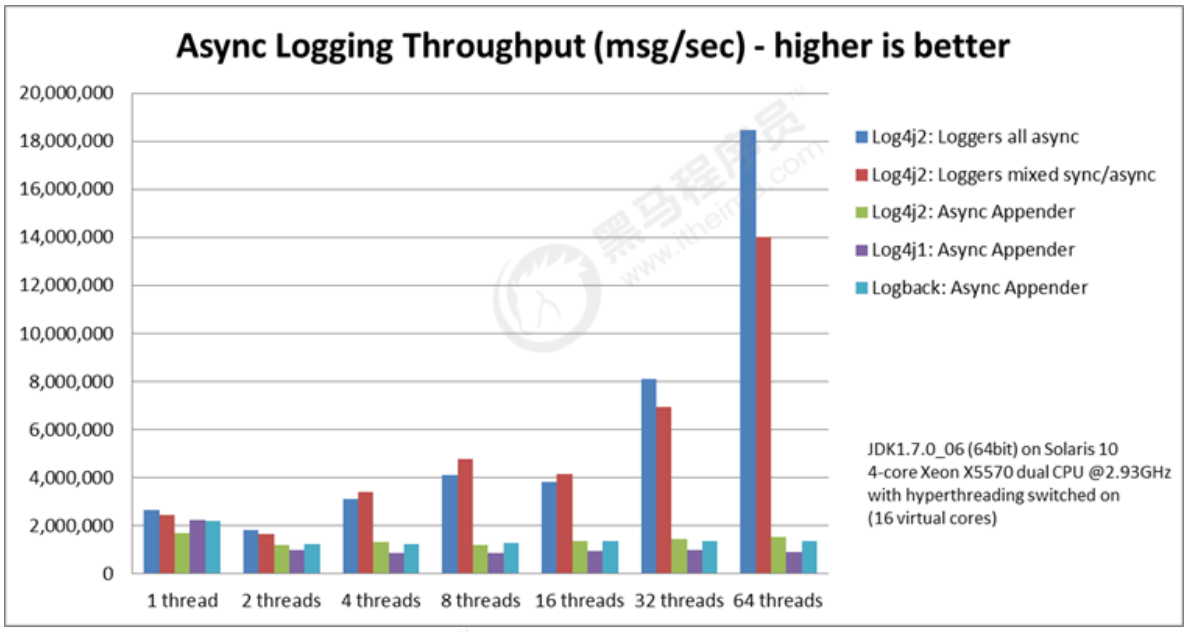
     如上配置： com.itheima 日志是异步的，root日志是同步的。

使用异步日志需要**注意的问题**：

1. 如果使用异步日志，AsyncAppender、AsyncLogger和全局日志，不要同时出现。性能会和AsyncAppender一致，降至最低。
2. 设置includeLocation=false ，打印位置信息会急剧降低异步日志的性能，比同步日志还要慢。

4.4 Log4j2的性能

Log4j2最牛的地方在于异步输出日志时的性能表现，Log4j2在多线程的环境下吞吐量与Log4j和Logback的比较如下图。下图比较中Log4j2有三种模式：1）全局使用异步模式；2）部分Logger采用异步模式；3）异步Appender。可以看出在前两种模式下，Log4j2的性能较之Log4j和Logback有很大的优势。



无垃圾记录

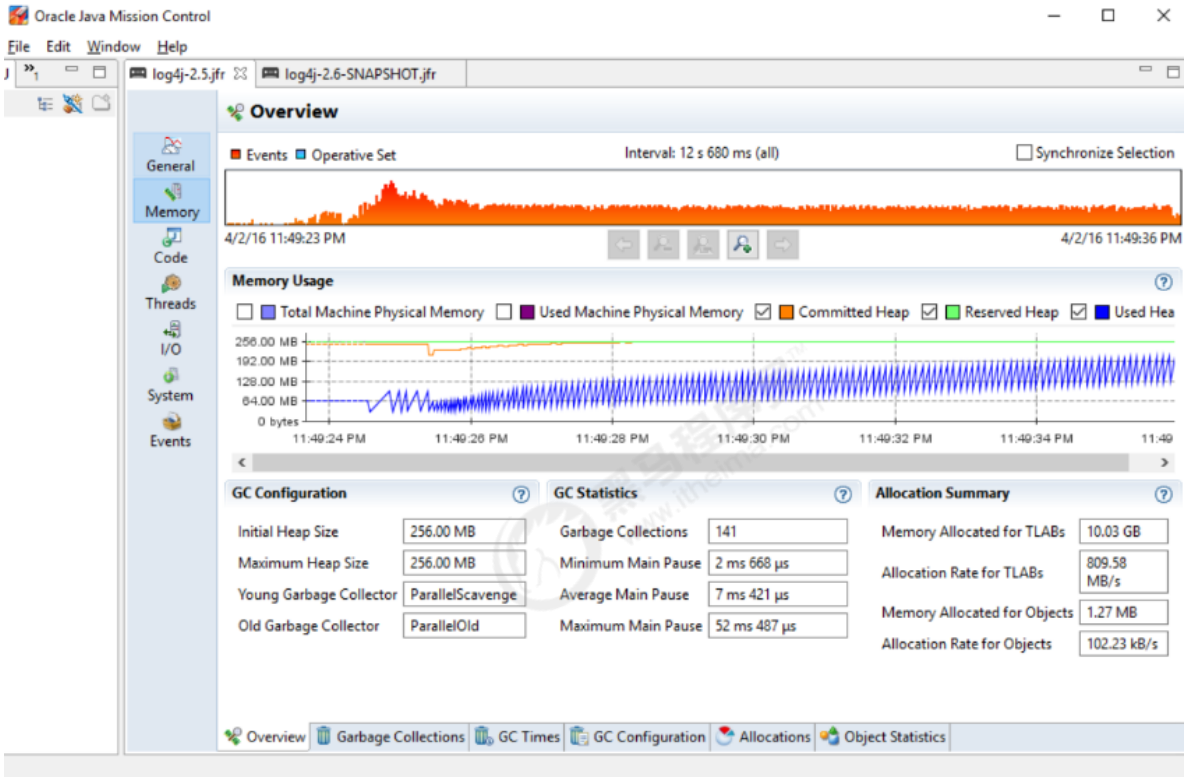
垃圾收集暂停是延迟峰值的常见原因，并且对于许多系统而言，花费大量精力来控制这些暂停。

许多日志库（包括以前版本的Log4j）在稳态日志记录期间分配临时对象，如日志事件对象，字符串，字符数组，字节数组等。这会对垃圾收集器造成压力并增加GC暂停发生的频率。

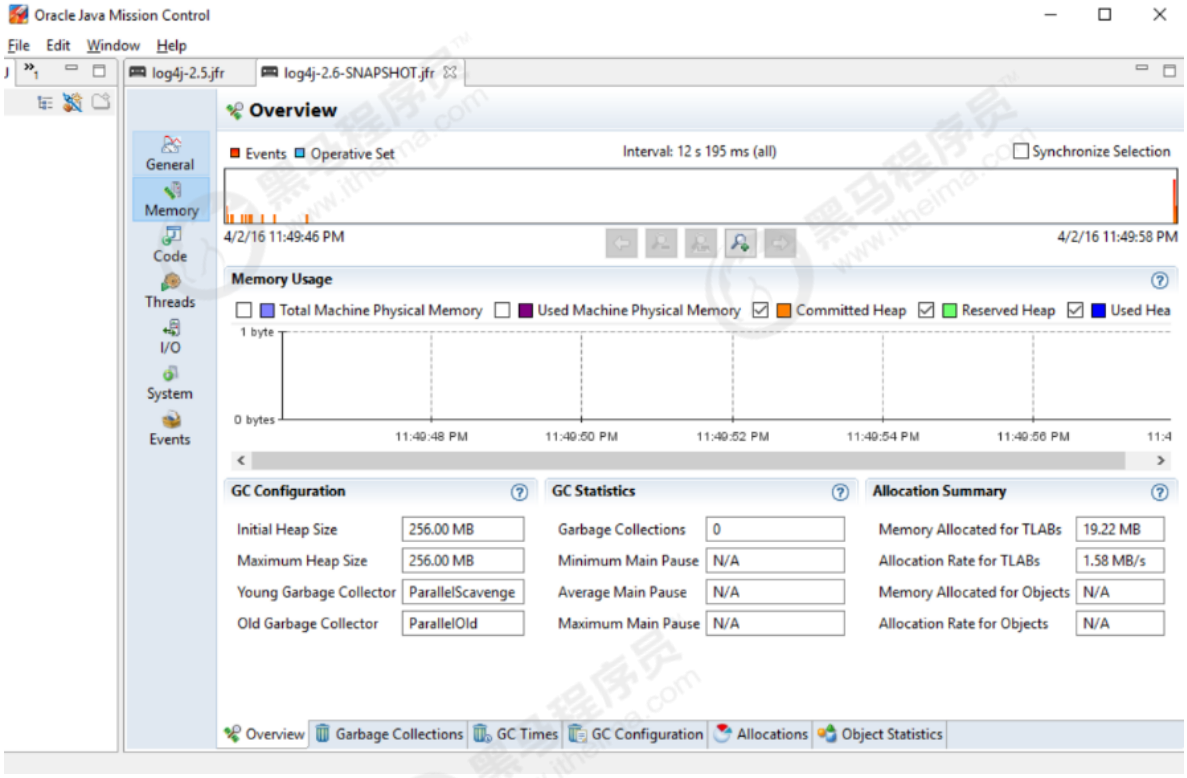
从版本2.6开始，默认情况下Log4j以“无垃圾”模式运行，其中重用对象和缓冲区，并且尽可能不分配临时对象。还有一个“低垃圾”模式，它不是完全无垃圾，但不使用ThreadLocal字段。

Log4j 2.6中的无垃圾日志记录部分通过重用ThreadLocal字段中的对象来实现，部分通过在将文本转换为字节时重用缓冲区来实现。

使用Log4j 2.5：内存分配速率809 MB /秒，141个无效集合。



Log4j 2.6没有分配临时对象：0（零）垃圾回收。



有两个单独的系统属性可用于手动控制Log4j用于避免创建临时对象的机制：

- `log4j2.enableThreadlocals` - 如果“true”（非Web应用程序的默认值）对象存储在ThreadLocal字段中并重新使用，否则将为每个日志事件创建新对象。
- `log4j2.enableDirectEncoders` - 如果将“true”（默认）日志事件转换为文本，则将此文本转换为字节而不创建临时对象。注意： 由于共享缓冲区上的同步，在此模式下多线程应用程序的*同步*日志记录性能可能更差。如果您的应用程序是多线程的并且日志记录性能很重要，请考虑使用异步记录器。

5. SpringBoot中的日志使用

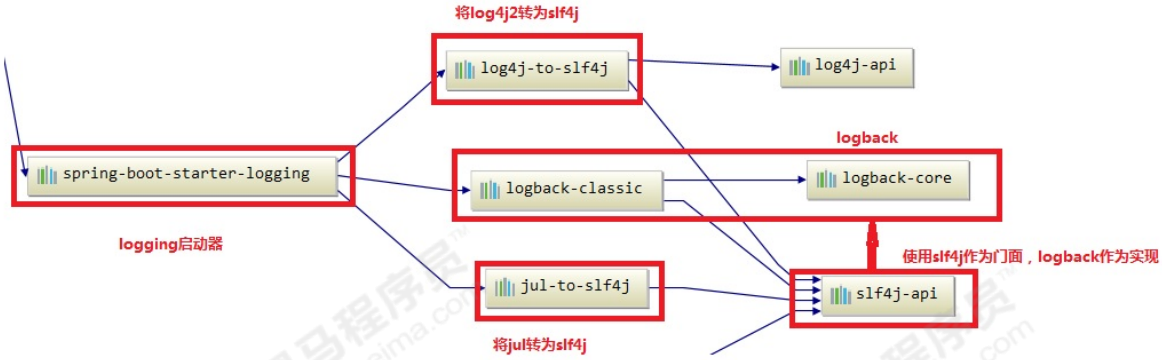
springboot框架在企业中的使用越来越普遍，springboot日志也是开发中常用的日志系统。springboot默认就是使用SLF4J作为日志门面，logback作为日志实现来记录日志。

5.1 SpringBoot中的日志设计

springboot中的日志

```
<dependency>
    <artifactId>spring-boot-starter-logging</artifactId>
    <groupId>org.springframework.boot</groupId>
</dependency>
```

依赖关系图：



总结：

1. springboot 底层默认使用logback作为日志实现。
2. 使用了SLF4J作为日志门面
3. 将JUL也转换成slf4j
4. 也可以使用log4j2作为日志门面，但是最终也是通过slf4j调用logback

5.2 SpringBoot日志使用

1. 在springboot中测试打印日志

```
@SpringBootTest
class SpringbootLogApplicationTests {

    //记录器
   public static final Logger LOGGER =
LoggerFactory.getLogger(SpringbootLogApplicationTests.class);

    @Test
    public void contextLoads() {
        // 打印日志信息
        LOGGER.error("error");
        LOGGER.warn("warn");
        LOGGER.info("info"); // 默认日志级别
        LOGGER.debug("debug");
        LOGGER.trace("trace");
```

```
    }

}
```

2. 修改默认日志配置

```
logging.level.com.itheima=trace
#  在控制台输出的日志的格式  同logback
logging.pattern.console=%d{yyyy-MM-dd} [%thread] [%-5level] %logger{50} -
 %msg%n

# 指定文件中日志输出的格式
logging.file=D:/logs/springboot.log
logging.pattern.file=%d{yyyy-MM-dd} [%thread] %-5level %logger{50} - %msg%n
```

3. 指定配置

给类路径下放上每个日志框架自己的配置文件；SpringBoot就不使用默认配置的了

日志框架	配置文件
Logback	logback-spring.xml , logback.xml
Log4j2	log4j2-spring.xml ， log4j2.xml
JUL	logging.properties

> logback.xml：直接就被日志框架识别了

4. 使用SpringBoot解析日志配置

logback-spring.xml：由SpringBoot解析日志配置

```
<encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <springProfile name="dev">
                <pattern>${pattern}</pattern>
            </springProfile>
            <springProfile name="pro">
                <pattern>%d{yyyyMMdd:HH:mm:ss.SSS} [%thread] %-5level
%msg%n</pattern>
            </springProfile>
</encoder>
```

application.properties

```
spring.profiles.active=dev
```

5. 将日志切换为log4j2

```
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <!--排除logback-->
        <exclusion>
            <artifactId>spring-boot-starter-logging</artifactId>
            <groupId>org.springframework.boot</groupId>
        </exclusion>
    </exclusions>
</dependency>
<!-- 添加log4j2 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```