

7. Spring Boot 日志

本节目标

1. 了解日志文件的用途
2. 学习SpringBoot 日志文件的配置

1. 日志概述

为什么要学习日志

日志对我们来说并不陌生,从JavaSE部分,我们就在使用 `System.out.print` 来打印日志了.通过打印日志来发现和定位问题,或者根据日志来分析程序的运行过程.在Spring的学习中,也经常根据控制台的日志来分析和定位问题.

随着项目的复杂度提升,我们对日志的打印也有了更高的需求,而不仅仅是定位排查问题.

比如需要记录一些用户的操作记录(一些审计公司会要求),也可能需要使用日志来记录用户的一些喜好,把日志持久化,后续进行数据分析等.但是 `System.out.print` 不能很好的满足我们的需求,我们就需要使用一些专门日志框架(专业的事情交给专业的人去做).

日志的用途

通过前面的学习,我们知道日志主要是为了发现问题,分析问题,定位问题的,但除此之外,日志还有很多用途

1. 系统监控

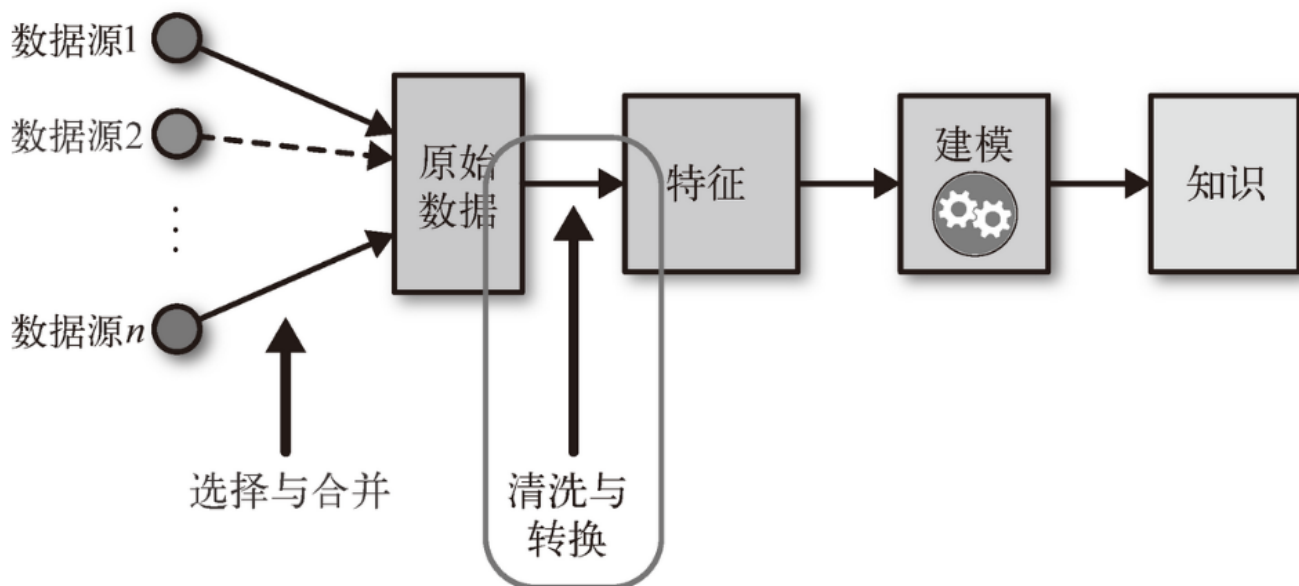
监控现在几乎是一个成熟系统的标配,我们可以通过日志记录这个系统的运行状态,每一个方法的响应时间,响应状态等,对数据进行分析,设置不同的规则,超过阈值时进行报警.比如统计日志中关键字的数量,并在关键字数量达到一定条件时报警,这也是日志的常见需求之一

2. 数据采集

数据采集是一个比较大的范围,采集的数据可以作用在很多方面,比如数据统计,推荐排序等.

- 数据统计: 统计页面的浏览量(PV), 访问量(UV), 点击量等, 根据这些数据进行数据分析, 优化公司运营策略
- 推荐排序: 目前推荐排序应用在各个领域, 我们经常接触的各行各业很多也都涉及推荐排序, 比如购物, 广告, 新闻等领域. 数据采集是推荐排序工作中必须做的一环, 系统通过日志记录用户的浏览历史, 停留时长等, 算法人员通过分析这些数据, 训练模型, 给用户做推荐.

下图中的数据源, 其中一部分就来自于日志记录的数据.



3. 日志审计

随着互联网的发展, 众多企业的关键业务越来越多的运行于网络之上. 网络安全越来越受到大家的关注, 系统安全也成为了项目中的一个重要环节, 安全审计也是系统中非常重要的部分. 国家的政策法规、行业标准等都明确对日志审计提出了要求. 通过系统日志分析, 可以判断一些非法攻击, 非法调用, 以及系统处理过程中的安全隐患.

比如, 大家平时都在做运营系统, 其中运营人员在通过界面处理一些数据的时候, 如果没有清楚的日志操作记录, 一条数据被删除或者修改, 你是无法找到是谁操作的, 但是如果你做了相应的记录, 该数据被谁删除或者修改就会一目了然.

还有一些内部的违规和信息泄漏(比如客户信息被卖掉)现象出现后, 如果未记录留存日志, 为事后调查提供依据, 则事后很难追查(一些公司查看客户的信息都会被记录日志, 如果频繁查询也会报警)

2. 日志使用

Spring Boot 项目在启动的时候默认就有日志输出, 如下图所示:

```
2023-09-21 11:40:28.577 INFO 21404 --- [main] com.example.demo.SpringIocApplication : No active profile set, falling back to 1 default profile: "default"
2023-09-21 11:40:30.027 INFO 21404 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2023-09-21 11:40:30.049 INFO 21404 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2023-09-21 11:40:30.049 INFO 21404 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.79]
2023-09-21 11:40:30.262 INFO 21404 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2023-09-21 11:40:30.262 INFO 21404 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1625 ms
2023-09-21 11:40:30.692 INFO 21404 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2023-09-21 11:40:30.703 INFO 21404 --- [main] com.example.demo.SpringIocApplication : Started SpringIocApplication in 2.779 seconds (JVM running for
```

它打印的日志和 `System.out.print` 有什么不同呢

```
1 @RestController
2 public class LoggerController {
3     @RequestMapping("/logger")
```

```
4     public String logger(){
5         System.out.println("打印日志");
6         return "打印日志";
7     }
8 }
```

观察日志输出

```
2023-09-20 16:45:43.800 INFO 19812 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2023-09-20 16:45:43.800 INFO 19812 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2023-09-20 16:45:43.801 INFO 19812 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 0 ms
打印日志
```

可以看到, 我们通过 `System.out.print` 打印的日志, 比 `SpringBoot` 打印的日志缺少了很多信息.

`SpringBoot` 内置了日志框架 `Slf4j`, 我们可以直接在程序中调用 `Slf4j` 来输出日志

2.1 打印日志

打印日志的步骤:

- 在程序中得到日志对象.
- 使用日志对象输出要打印的内容.

2.1.1 在程序中得到日志对象

在程序中获取日志对象需要使用日志工厂 `LoggerFactory`, 如下代码所示:

```
1 private static Logger logger = LoggerFactory.getLogger(LoggerController.class);
```

`LoggerFactory.getLogger` 需要传递一个参数, 标识这个日志的名称. 这样可以更清晰的知道是哪个类输出的日志. 当有问题时, 可以更方便直观的定位到问题类

注意: **Logger 对象是属于 `org.slf4j` 包下的**, 不要导入错包.

@RestController

```
public class LoggerController {  
    private static Logger logger = LoggerFactory.getLogger(LoggerController.class);
```

Class to Import

Class	Maven
Logger (org.slf4j)	Maven: org.slf4j:slf4j-api:1.7.36 (slf4j-api-1.7.36.jar)
Logger (java.util.logging)	< 1.8 > (rt.jar)
Logger (jdk.internal.instrumentation)	< 1.8 > (rt.jar)
Logger (jdk.nashorn.internal.runtime.logging)	< 1.8 > (nashorn.jar)
Logger (ch.qos.logback.classic)	Maven: ch.qos.logback:logback-classic:1.2.12 (logback-classic-1.2.12.jar)
Logger (org.apache.logging.log4j)	Maven: org.apache.logging.log4j:log4j-api:2.17.2 (log4j-api-2.17.2.jar)
Logger (com.sun.istack.internal.logging)	< 1.8 > (rt.jar)
Logger (com.sun.javaafx.logging)	< 1.8 > (jfxrt.jar)
Logger (com.sun.media.jfxmedia.logging)	< 1.8 > (jfxrt.jar)
Logger (com.sun.org.slf4j.internal)	< 1.8 > (rt.jar)

@RestController

```
public class LoggerController {  
    private static Logger logger = LoggerFactory.getLogger(LoggerController.class);
```

Class to Import

Class	Maven
LoggerFactory (org.slf4j)	Maven: org.slf4j:slf4j-api:1.7.36 (slf4j-api-1.7.36.jar)
LoggerFactory (com.sun.org.slf4j.internal)	< 1.8 > (rt.jar)

2.1.2 使用日志对象打印日志

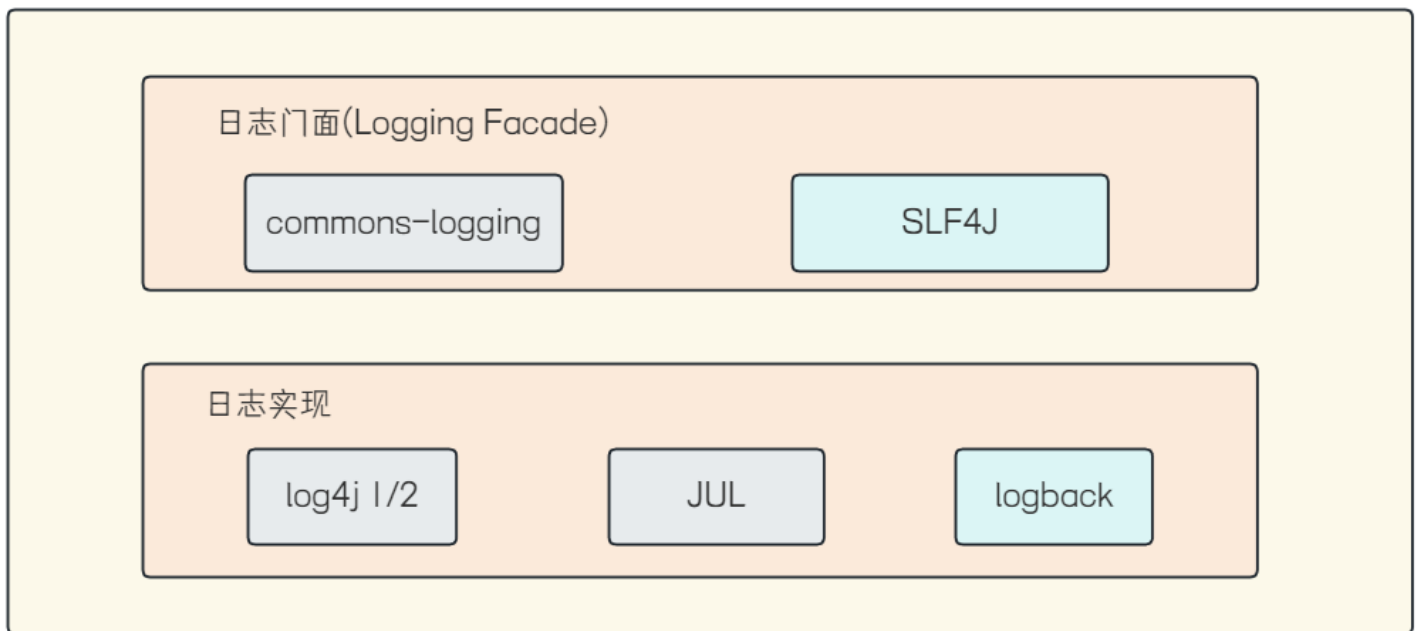
日志对象的打印方法有很多种，我们可以先使用 info() 方法来输出日志，如下代码所示：

```
1 import org.slf4j.Logger;  
2 import org.slf4j.LoggerFactory;  
3 import org.springframework.web.bind.annotation.RequestMapping;  
4 import org.springframework.web.bind.annotation.RestController;  
5  
6 @RestController  
7 public class LoggerController {  
8     private static Logger logger = LoggerFactory.getLogger(LoggerController.clas  
9  
10    @RequestMapping("/logger")  
11    public String logger(){  
12        logger.info("-----要输出日志的内容-----");  
13        return "打印日志";  
14    }  
15  
16 }
```

打印日志效果展示：

```
2023-09-20 17:24:56.422 INFO 22900 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'  
2023-09-20 17:24:56.422 INFO 22900 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'  
2023-09-20 17:24:56.423 INFO 22900 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms  
2023-09-20 17:24:56.447 INFO 22900 --- [nio-8080-exec-1] c.e.demo.controller.LoggerController : -----要输出日志的内容-----
```

2.2 日志框架介绍(了解)



SLF4J不同于其他日志框架, 它不是一个真正的日志实现, 而是一个抽象层, 对日志框架制定的一种规范, 标准, 接口. 所有SLF4J并不能独立使用, 需要和具体的日志框架配合使用.

2.2.1 门面模式(外观模式)

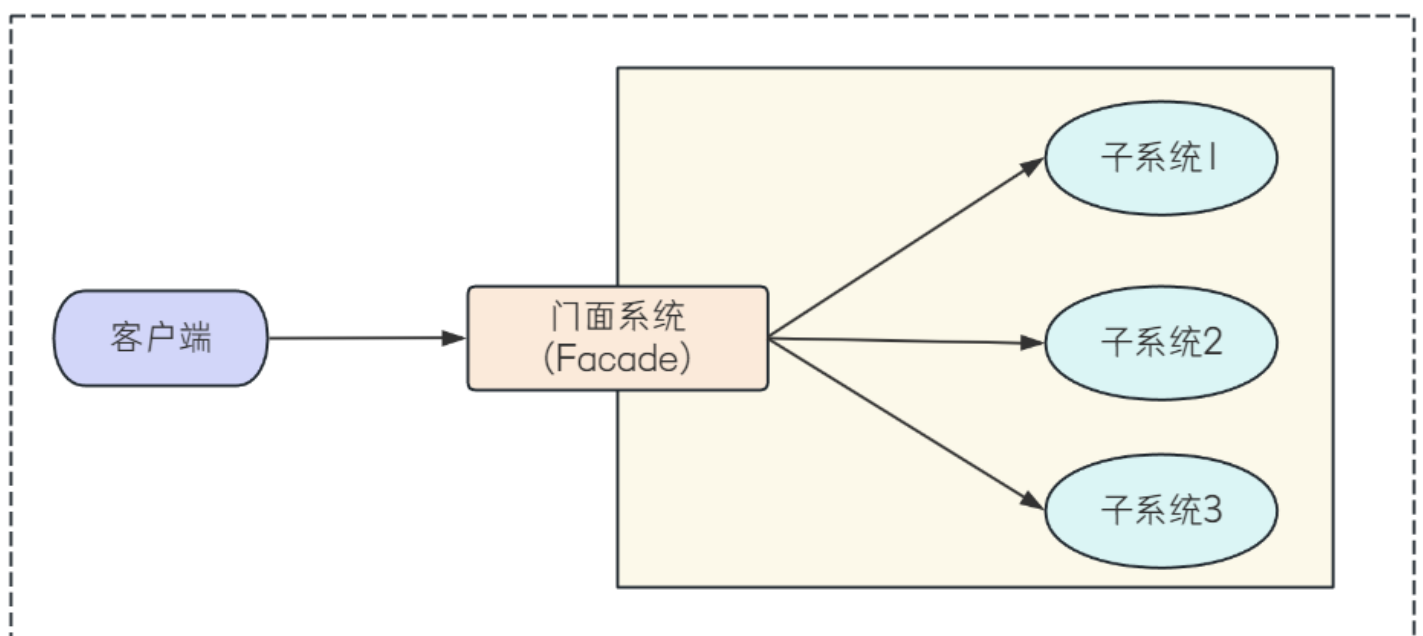
SLF4J是门面模式的典型应用(但不仅仅使用了门面模式).

门面模式定义

门面模式 (Facade Pattern) 又称为**外观模式**, 提供了一个统一的接口, 用来访问子系统中的一群接口. 其主要特征是定义了一个高层接口, 让子系统更容易使用.

原文: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

解释: 要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行. 门面模式提供一个高层次的接口, 使得子系统更易于使用.

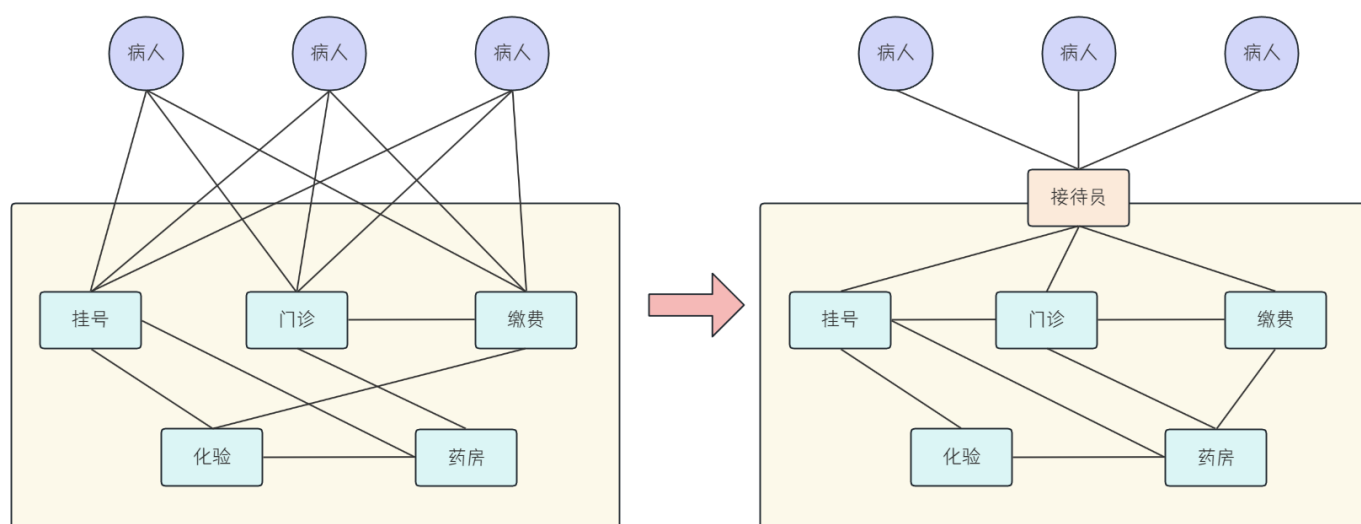


门面模式主要包含2种角色:

外观角色(Facade): 也称门面角色, 系统对外的统一接口.

子系统角色(SubSystem): 可以同时有一个或多个 SubSystem. 每个 SubSystem 都不是一个单独的类, 而是一个类的集合. SubSystem 并不知道 Facade 的存在, 对于 SubSystem 而言, Facade 只是另一个客户端而已(即 Facade 对 SubSystem 透明)

比如去医院看病, 可能要去挂号, 门诊, 化验, 取药, 让患者或患者家属觉得很复杂, 如果有提供接待人员, 只让接待人员来处理, 就很方便.



门面模式的实现

场景: 回家, 我们会开各个屋的灯. 离开家时, 会关闭各个屋的灯

如果家里设置一个总开关, 来控制整个屋的灯就会很方便.

我们使用门面模式的实现

```
1 public class FacadePatternDemo {
2     public static void main(String[] args) {
3         LightFacade lightFacade = new LightFacade();
4         lightFacade.lightOn();
5
6     }
7 }
8
9 /**
10  * 灯的门面
11  */
12 class LightFacade{
13     private Light livingRoomLight = new LivingRoomLight();
14     private Light hallLight = new HallLight();
```

```
15     private Light diningLight = new DiningLight();
16     public void lightOn(){
17         livingRoomLight.on();
18         hallLight.on();
19         diningLight.on();
20     }
21     public void lightOff(){
22         livingRoomLight.off();
23         hallLight.off();
24         diningLight.off();
25     }
26 }
27 interface Light {
28     void on();
29     void off();
30 }
31
32 /**
33  * 客厅灯
34  */
35 class LivingRoomLight implements Light{
36
37     @Override
38     public void on() {
39         System.out.println("打开客厅灯");
40     }
41
42     @Override
43     public void off() {
44         System.out.println("关闭客厅灯");
45     }
46 }
47
48 /**
49  * 走廊灯
50  */
51 class HallLight implements Light{
52
53     @Override
54     public void on() {
55         System.out.println("打开走廊灯");
56     }
57
58     @Override
59     public void off() {
60         System.out.println("关闭走廊灯");
61     }
62 }
```

```

62 }
63
64 /**
65  * 餐厅灯
66  */
67 class DiningLight implements Light{
68
69     @Override
70     public void on() {
71         System.out.println("打开餐厅灯");
72     }
73
74     @Override
75     public void off() {
76         System.out.println("关闭餐厅灯");
77     }
78 }

```

门面模式的优点

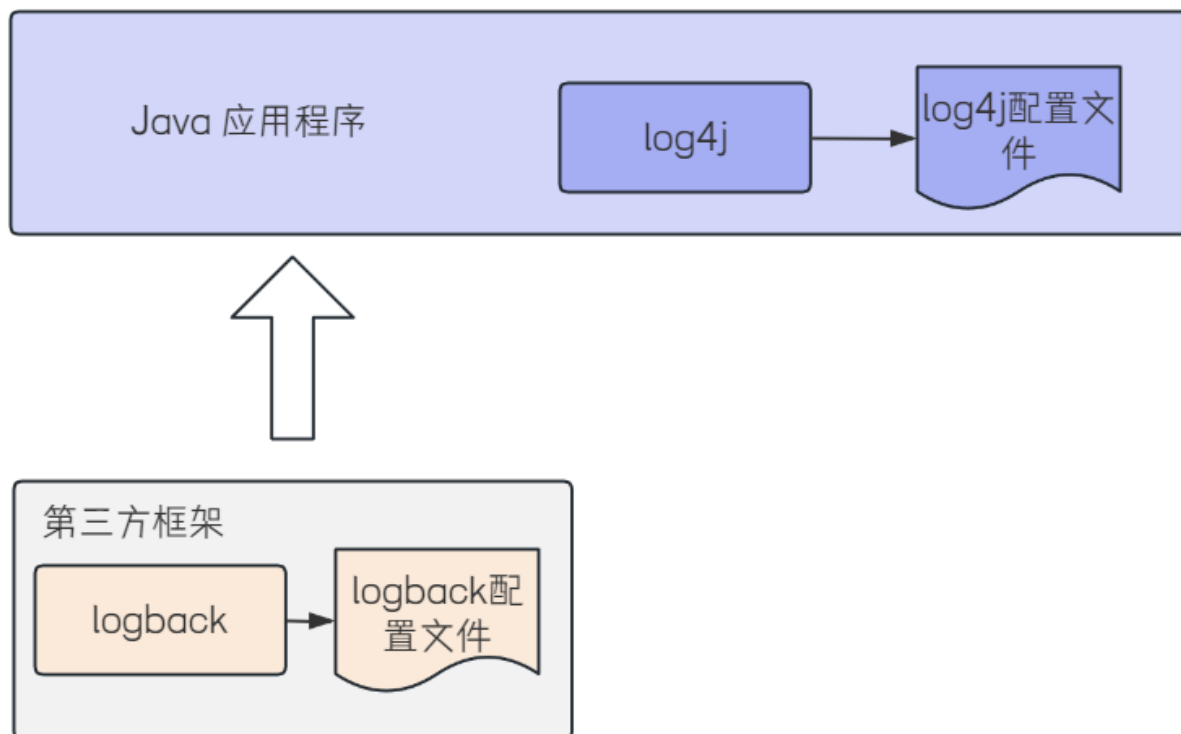
- 减少了系统的相互依赖. 实现了客户端与子系统的耦合关系, 这使得子系统的变化不会影响到调用它的客户端;
- 提高了灵活性, 简化了客户端对子系统的使用难度, 客户端无需关心子系统的具体实现方式, 而只需要和门面对象交互即可.
- 提高了安全性. 可以灵活设定访问权限, 不在门面对象中开通方法, 就无法访问

2.2.2 SLF4J 框架介绍

SLF4J 就是其他日志框架的门面. SLF4J 可以理解为是提供日志服务的统一API接口, 并不涉及到具体的日志逻辑实现.

不引入日志门面

常见的日志框架有log4J, logback等. 如果一个项目已经使用了log4j, 而你依赖的另一个类库, 假如是Apache Active MQ, 它依赖于另外一个日志框架logback, 那么你就需要把logback也加载进去.

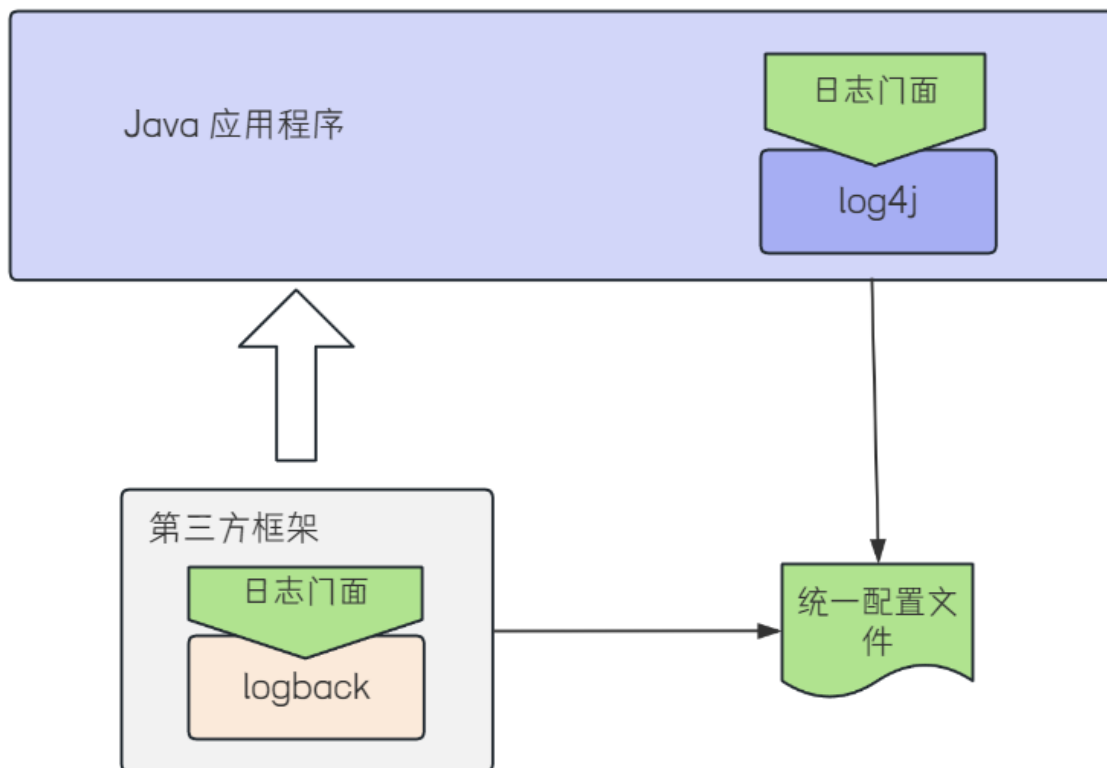


存在问题:

1. 不同日志框架的API接口和配置文件不同, 如果多个日志框架共存, 那么不得不维护多套配置文件(这个配置文件是指用户自定义的配置文件).
2. 如果要更换日志框架, 应用程序将不得不修改代码, 并且修改过程中可能会存在一些代码冲突.
3. 如果引入的第三方框架, 使用了多套, 那就不得不维护多套配置.

引入日志门面

引入门面日志框架之后, 应用程序和日志框架(框架的具体实现)之间有了统一的API接口(门面日志框架实现), 此时应用程序只需要维护一套日志文件配置, 且当底层实现框架改变时, 也不需要更改应用程序代码.



SLF4J 就是这个日志门面。

总的来说，SLF4J使你的代码独立于任意一个特定的日志API，这是一个对于开发API的开发者很好的思想。

2.3 日志格式的说明

打印的日志分别代表什么呢？

```
2023-09-21 11:40:28.577 INFO 21404 --- [main] com.example.demo.SpringIocApplication : No active profile set, falling back to 1 default profile: "default"
2023-09-21 11:40:30.027 INFO 21404 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2023-09-21 11:40:30.049 INFO 21404 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2023-09-21 11:40:30.049 INFO 21404 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.79]
2023-09-21 11:40:30.262 INFO 21404 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2023-09-21 11:40:30.262 INFO 21404 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1625 ms
2023-09-21 11:40:30.692 INFO 21404 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2023-09-21 11:40:30.703 INFO 21404 --- [main] com.example.demo.SpringIocApplication : Started SpringIocApplication in 2.779 seconds (JVM running for
```

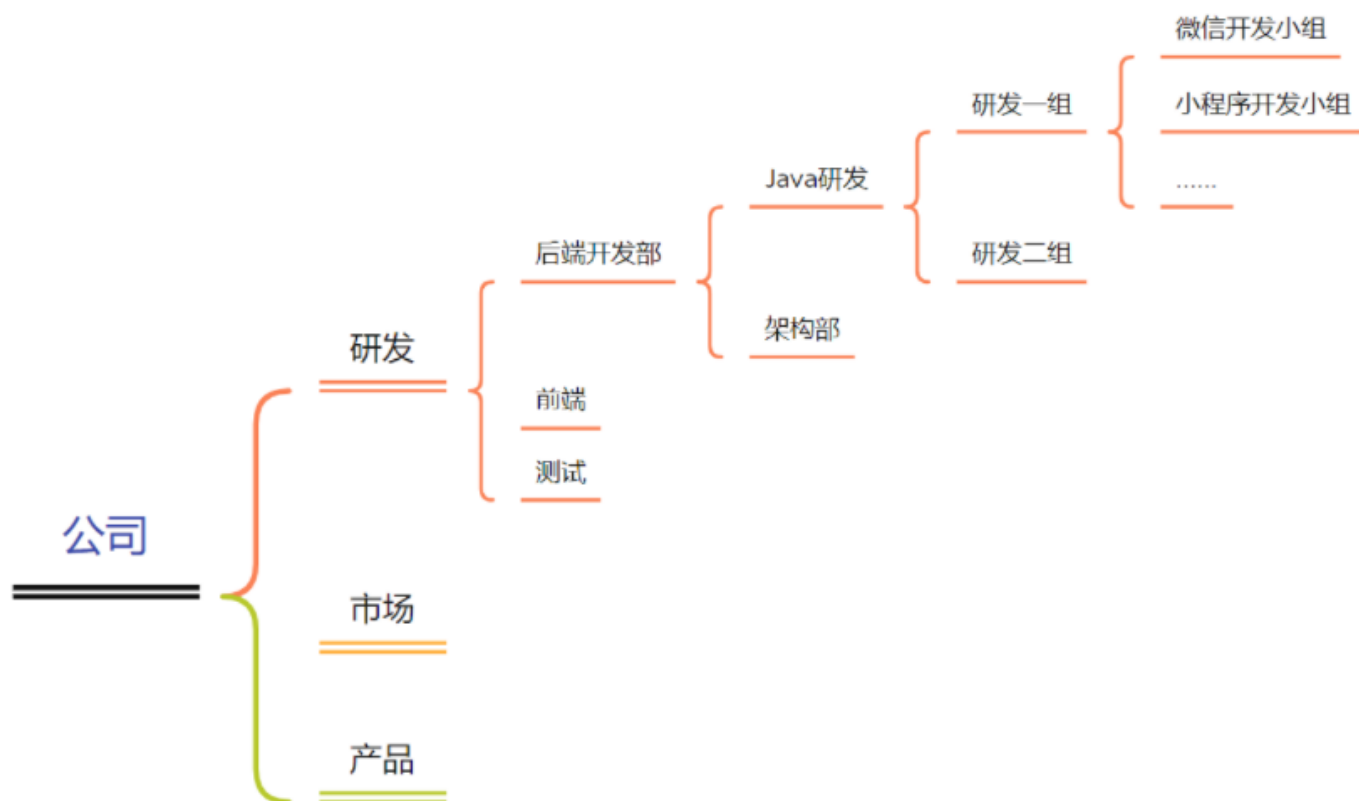
从上图可以看到，日志输出内容元素具体如下：

1. 时间日期：精确到毫秒
2. 日志级别：ERROR, WARN, INFO, DEBUG 或TRACE
3. 进程ID
4. 线程名
5. Logger名(通常使用源代码的类名)
6. 日志内容

2.4 日志级别

日志级别代表着日志信息对应问题的严重性, 为了更快的筛选符合目标的日志信息.

试想一下这样的场景, 假设你是一家 2 万人公司的老板, 如果每个员工的日常工作和琐碎的信息都要反馈给你, 那你一定无暇顾及. 于是就有了组织架构, 而组织架构就会分级, 有很多的级别设置, 如下图所示:



有了组织架构之后, 就可以逐级别汇报消息了, 例如: 组员汇报给组长, 组长汇报给研发一组, 研发一组汇报给 Java 研发, 等等依次进行汇报.

日志级别大概是同样的道理, 有了日志级别之后就可以过滤自己想看到的信息了, 比如只关注error级别的, 就可以根据级别过滤出来error级别的日志信息, 节约开发者的信息筛选时间.

2.4.1 日志级别的分类

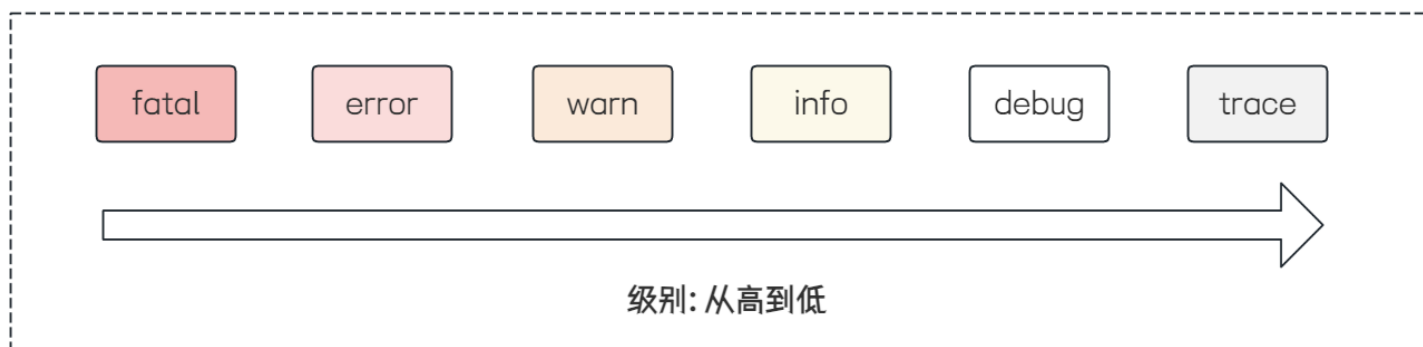
日志的级别从高到低依次为: FATAL、ERROR、WARN、INFO、DEBUG、TRACE

- FATAL: 致命信息, 表示需要立即被处理的系统级错误.
- ERROR: 错误信息, 级别较高的错误日志信息, 但仍然不影响系统的继续运行.
- WARN: 警告信息, 不影响使用, 但需要注意的问题
- INFO: 普通信息, 用于记录应用程序正常运行时的一些信息, 例如系统启动完成、请求处理完成等.
- DEBUG: 调试信息, 需要调试时候的关键信息打印.
- TRACE: 追踪信息, 比DEBUG更细粒度的信息事件(除非有特殊用意, 否则请使用DEBUG级别替代)

日志级别通常和测试人员的Bug级别没有关系。

日志级别是开发人员设置的,用来给开发人员看的. 日志级别的正确设置,也与开发人员的工作经验有关. 如果开发人员把error级别的日志设置成了info,就很有可能会影响开发人员对项目运行情况的判断. 出现error级别的日志信息较多时,可能也没有任何问题. 测试的bug级别更多是依据现象和影响范围来判断

日志级别的顺序:



级别越高,收到的消息越少

2.4.2 日志级别的使用

日志级别是开发人员自己设置的. 开发人员根据自己的理解来判断该信息的重要程度

类似公司管理,通常由领导来判断什么样的事情需要汇报,什么样的事情不需要汇报.

针对这些级别, Logger 对象分别提供了对应的方法,来输出日志.

```
1 /**
2  * 打印不同级别的日志
3  * @return
4  */
5 @RequestMapping("/printLog")
6 public String printLog() {
7     logger.trace("===== trace =====");
8     logger.debug("===== debug =====");
9     logger.info("===== info =====");
10    logger.warn("===== warn =====");
11    logger.error("===== error =====");
12
13    return "打印不同级别的日志" ;
14 }
```

SpringBoot 默认的日志框架是Logback, Logback没有 FATAL 级别, 它被映射到 ERROR .

出现fatal日志, 表示服务已经出现了某种程度的不可用, 需要需要系统管理员紧急介入处理. 通常情况下, 一个进程生命周期中应该最多只有一次FATAL记录.

观察打印的日志结果:

```
2023-09-20 18:47:30.790 INFO 36112 --- [nio-8080-exec-3] c.e.demo.controller.LoggerController : ===== info =====
2023-09-20 18:47:30.791 WARN 36112 --- [nio-8080-exec-3] c.e.demo.controller.LoggerController : ===== warn =====
2023-09-20 18:47:30.791 ERROR 36112 --- [nio-8080-exec-3] c.e.demo.controller.LoggerController : ===== error =====
|
```

结果发现, 只打印了info, warn和error级别的日志

这与日志级别的配置有关, 日志的输出级别默认是 info级别, 所以只会打印大于等于此级别的日志, 也就是info, warn和error.

2.5 日志配置

上述是日志的使用, 日志框架支持我们更灵活的输出日志, 包括内容, 格式等.

2.5.1 配置日志级别

日志级别配置只需要在配置文件中设置"logging.level"配置项即可, 如下所示:

后面课程中出现的配置, properties 和 yml 只需要配置其中一个即可.

二者转换方式: Properties文件的点(.) 对应yml文件中的换行

以下两个配置, 根据项目使用其中之一.

Properties配置

```
1 logging.level.root: debug
```

yml配置

```
1 logging:
2   level:
3     root: debug
```

重新运行上述代码, 观察结果:

```
2023-09-20 19:02:14.876 DEBUG 21512 --- [nio-8080-exec-2] c.e.demo.controller.LoggerController : ===== debug =====
2023-09-20 19:02:14.876 INFO 21512 --- [nio-8080-exec-2] c.e.demo.controller.LoggerController : ===== info =====
2023-09-20 19:02:14.876 WARN 21512 --- [nio-8080-exec-2] c.e.demo.controller.LoggerController : ===== warn =====
2023-09-20 19:02:14.876 ERROR 21512 --- [nio-8080-exec-2] c.e.demo.controller.LoggerController : ===== error =====
```

2.5.2 日志持久化

以上的日志都是输出在控制台上的, 然而在线上环境中, 我们需要把日志保存下来, 以便出现问题之后追溯问题. 把日志保存下来就叫持久化.

日志持久化有两种方式

1. 配置日志文件名
2. 配置日志的存储目录

`logging.file.name`

Log file name (for instance, `myapp.log`). Names can be an exact location or relative to the current directory.

`logging.file.path`

Location of the log file. For instance, `/var/log`.

配置日志文件的路径和文件名：

Properties配置

```
1 logging.file.name: logger/springboot.log
```

yaml配置

```
1 # 设置日志文件的文件名
2 logging:
3   file:
4     name: logger/springboot.log
```

后面可以跟绝对路径或者相对路径

运行结果显示, 日志内容保存在了对应的目录下



配置日志文件的保存路径

Properties配置

```
1 logging.file.path: D:/temp
```

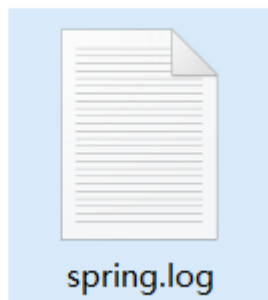
yaml配置

```
1 # 设置日志文件的目录
2 logging:
3   file:
4     path: D:/temp
```

这种方式只能设置日志的路径, 文件名为固定的spring.log

运行程序, 该路径下多出一个日志文件: spring.log

📁 > 此电脑 > Data (D:) > temp



注意:

`logging.file.name` 和 `logging.file.path` 两个都配置的情况下, 只生效其一, 以 `logging.file.name` 为准.

2.5.3 配置日志文件分割

如果我们的日志都放在一个文件中, 随着项目的运行, 日志文件会越来越大, 需要对日志文件进行分割.

当然, 日志框架也帮我们考虑到了这一点, 所以如果不进行配置, 就走自动配置

默认日志文件超过10M就进行分割

配置项	说明	默认值
<code>logging.logback.rollingpolicy.file-name-pattern</code>	日志分割后的文件名格式	<code>\${LOG_FILE}_%d{yyyy-MM-dd}_%i.gz</code>
<code>logging.logback.rollingpolicy.max-file-size</code>	日志文件超过这个大小就自动分割	10MB

配置日志文件分割:

Properties配置

```
1 logging.logback.rollingpolicy.file-name-pattern=${LOG_FILE}.%d{yyyy-MM-dd}.%i
2 logging.logback.rollingpolicy.max-file-size=1KB
```




yaml配置

```
1 logging:
2   logback:
3     rollingpolicy:
4       max-file-size: 1KB
5       file-name-pattern: ${LOG_FILE}.%d{yyyy-MM-dd}.%i
```

- 1. 日志文件超过1KB就分割(设置1KB是为了更好展示. 企业开发通常设置为200M, 500M等, 此处没有明确标准)
- 2. 分割后的日志文件名为: 日志名.日期.索引

项目运行, 多打印一些日志, 日志分割结果:

Data (D:) > Git > JavaEE课件相关资料 > 后端代码 > spring-ioc > logger

名称	修改日期	类型	大小
 springboot.log	2023/9/21 11:42	文本文档	0 KB
 springboot.log.2023-09-21.0	2023/9/21 11:40	0 文件	2 KB
 springboot.log.2023-09-21.1	2023/9/21 11:42	1 文件	2 KB
 springboot.log.2023-09-21.2	2023/9/21 11:42	2 文件	2 KB
 springboot.log.2023-09-21.3	2023/9/21 11:42	3 文件	2 KB

2.5.4 配置日志格式

目前日志打印的格式是默认的

```
2023-09-21 11:42:16.268 INFO 21404 --- [nio-8080-exec-6] c.e.demo.controller.LoggerController : ===== info =====
2023-09-21 11:42:16.270 WARN 21404 --- [nio-8080-exec-6] c.e.demo.controller.LoggerController : ===== warn =====
2023-09-21 11:42:16.270 ERROR 21404 --- [nio-8080-exec-6] c.e.demo.controller.LoggerController : ===== error =====
```

打印日志的格式, 也是支持配置的. 支持控制台和日志文件分别设置

配置项	说明	默认值
logging.pattern.console	控制台日志格式	%clr(%d\${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd'T'HH:mm:ss.SSSXXX}){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}

<code>logging.pattern.file</code>	日志文件的日志格式	<code>%d{\${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd'T'HH:mm:ss.SSSXXX}} \${LOG_LEVEL_PATTERN:-%5p} \${PID:-} --- [%t] %-40.40logger{39} : %m%n\${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}</code>
-----------------------------------	-----------	---

配置项说明:

1. `%clr(表达式){颜色}` 设置输入日志的颜色

支持颜色有以下几种:

- blue
- cyan
- faint
- green
- magenta
- red
- yellow

2. `%d{${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd'T'HH:mm:ss.SSSXXX}}` 日期和时间--精确到毫秒

`%d{}` 日期

`${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd'T'HH:mm:ss.SSSXXX}` 非空表达式, 获取系统属性 `LOG_DATEFORMAT_PATTERN` , 若属性 `LOG_DATEFORMAT_PATTERN` 不存在, 则使用 `-yyyy-MM-dd HH:mm:ss.SSSXXX` 格式, 系统属性可以 `System.getProperty("LOG_DATEFORMAT_PATTERN")` 获取

3. `%5p` 显示日志级别ERROR, MARN, INFO, DEBUG, TRACE.

4. `%t` 线程名. `%c` 类的全限定名. `%M` method. `%L` 为行号. `%thread` 线程名称. `%m` 或者 `%msg` 显示输出消息. `%n` 换行符

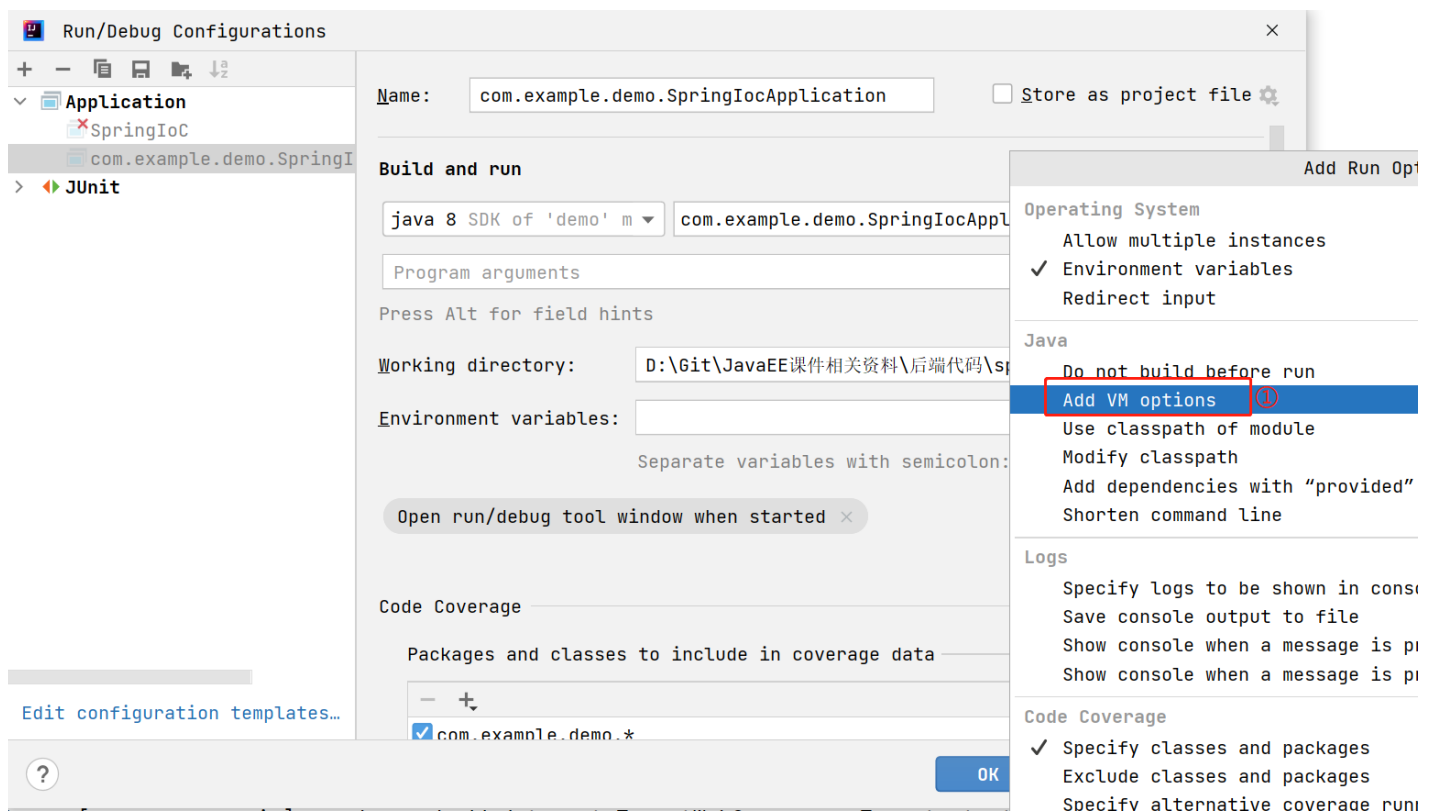
5. `%5` 若字符长度小于5, 则右边用空格填充. `%-5` 若字符长度小于5, 则左边用空格填充. `%.15` 若字符长度超过15, 截去多余字符. `%15.15` 若字符长度小于15, 则右边用空格填充. 若字符长度超过15, 截去多余字符

更多说明, 参考: <https://logback.qos.ch/manual/layouts.html#conversionWord>

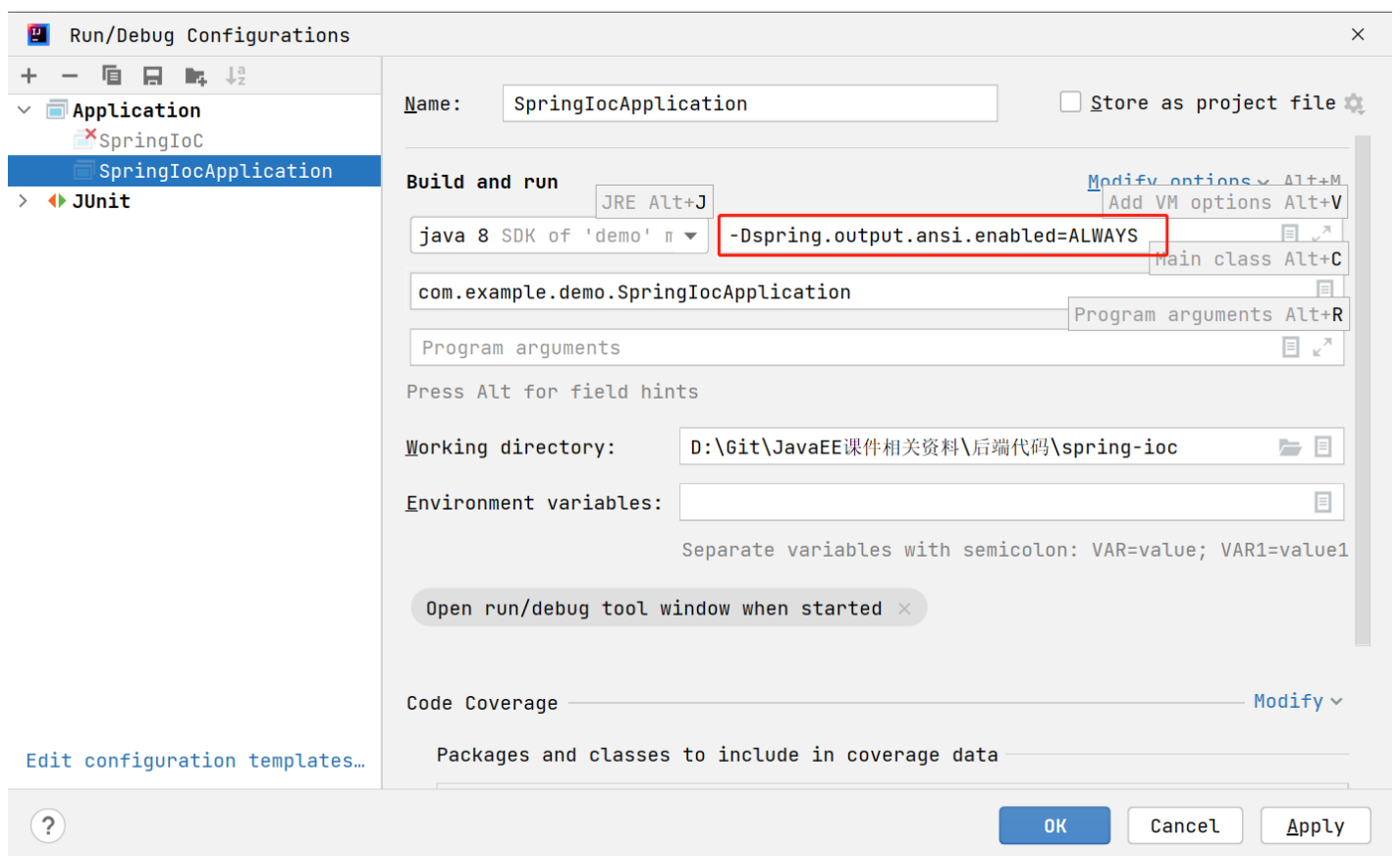
设置了颜色, 却没有生效?

需要配置, 让idea支持控制台颜色显示

1. 打开启动配置, 添加VM options



2. 添加VM options `-Dspring.output.ansi.enabled=ALWAYS`



3. 重新启动程序, 就发现控制台支持颜色了

```

2023-11-10 18:41:21.709 DEBUG 52020 --- [main] c.e.demo.controller.LoggerController : ===== debug =====
2023-11-10 18:41:21.709 INFO 52020 --- [main] c.e.demo.controller.LoggerController : ===== info =====
2023-11-10 18:41:21.709 WARN 52020 --- [main] c.e.demo.controller.LoggerController : ===== warn =====
2023-11-10 18:41:21.709 ERROR 52020 --- [main] c.e.demo.controller.LoggerController : ===== error =====

```

Properties配置

```
1 logging.pattern.console='%d{yyyy-MM-dd HH:mm:ss.SSS} %c %M %L [%thread] %m%n'
```

yml配置

```
1 logging:
2   pattern:
3     console: '%d{yyyy-MM-dd HH:mm:ss.SSS} %c %M %L [%thread] %m%n'
4     file: '%d{yyyy-MM-dd HH:mm:ss.SSS} %c %M %L [%thread] %m%n'
```

项目运行, 观察日志变化:

```
2023-09-21 15:30:22.352 org.springframework.web.servlet.DispatcherServlet initServletBean 525 [http-nio-8080-exec-2] Initializing Servlet 'dispatcherServlet'
2023-09-21 15:30:22.352 org.springframework.web.servlet.DispatcherServlet initServletBean 547 [http-nio-8080-exec-2] Completed initialization in 0 ms
2023-09-21 15:30:22.401 com.example.demo.controller.LoggerController printLog 26 [http-nio-8080-exec-2] ===== info =====
2023-09-21 15:30:22.401 com.example.demo.controller.LoggerController printLog 27 [http-nio-8080-exec-2] ===== warn =====
2023-09-21 15:30:22.401 com.example.demo.controller.LoggerController printLog 28 [http-nio-8080-exec-2] ===== error =====
```

通常情况下, 咱们就使用默认的日志格式打印即可.

3. 更简单的日志输出

每次都使用 `LoggerFactory.getLogger(xxx.class)` 很繁琐, 且每个类都添加一遍, lombok给我们提供了一种更简单的方式.

1. 添加 lombok 框架支持
2. 使用 `@Slf4j` 注解输出日志。

3.1 添加 lombok 依赖

```
1 <dependency>
2   <groupId>org.projectlombok</groupId>
3   <artifactId>lombok</artifactId>
4   <optional>true</optional>
5 </dependency>
```

3.2 输出日志

```

1 import lombok.extern.slf4j.Slf4j;
2 import org.springframework.web.bind.annotation.RestController;
3
4 @Slf4j
5 @RestController
6 public class LogController {
7     public void log(){
8         log.info("-----要输出日志的内容-----");
9     }
10 }

```

lombok提供的 `@Slf4j` 会帮我们提供一个日志对象 `log`, 我们直接使用就可以.



4. 总结

1. 日志是程序中的重要组成部分, 使用日志可以快速的发现和定位问题, Spring Boot 内置了日志框架, 默认情况下使用的是 info 日志级别将日志输出到控制台的, 我们可以通过 lombok 提供的 `@Slf4j` 注解和 `log` 对象快速的打印自定义日志.
2. 日志包含 6 个级别, 日志级别越高, 收到的日志信息也就越少, 我们可以通过配置日志的保存名称或保存目录来将日志持久化.