1.JAVA混乱的日志体系

换乱的java日志体系

case：

SLF4J-JCL

LOG4J-CORE

LOGBACK

SLF4J-SIMPLE

JCL-OVER-SLF4J

LOGBACK-CORE

LOG4J

LOG4J-API

LOG4J-JUL

LOG4J-JCL

LOGBACK-ACCESS

LOGBACK-CLASSIC

SLF4-API

SLF4J-LOGJ12

LOGBACK-CLASSIC

LOG4J-SLF4J-IMPL

1.1 JAVA日志体系概述

问题：

常用的日志框架有哪些?

大家目前正在使用的

名称	jar	描述
log4j	log4j-1.2.17	早期常用的日志组件
logback	logback-core,logback-classic,logback-access	性能由于log4j
log4j2	log4j,log4j-api,log4j-core	log4j升级
Java.util.logging	jdk	jdk实现,tomcat默认实现

开发一个类似spring框架，或者开发一个组件，如何选择

选择任何一种实现，都不太好，不同的日志输出不一样，日志也会打印多份。如何解决，日志做抽象层

1.1.1 Apache Commons Logging（JCL）

1-官网介绍
记录组件
在编写一个库时，记录信息是非常有用的。然而，外面有很多日志实现，一个库不能把某一个特定的日志实现强加给作为库一部分的整体应用。

Logging包是不同日志实现之间的一个超薄桥梁。一个使用commons-logging API的库可以在运行时与任何日志实现一起使用。Common-logging自带对许多流行的日志实现的支持，为其他的日志实现编写适配器是一项相当简单的任务。

应用程序（而不是库）也可以选择使用commonons-logging。虽然日志记录实现的独立性对应用程序来说不像库那样重要，但使用commons-logging确实允许应用程序在不重新编译代码的情况下改变到不同的日志实现。

请注意，commons-logging不会尝试初始化或终止运行时使用的底层日志实现，这是应用程序的责任。然而，许多常用的日志实现都会自动初始化；在这种情况下，应用程序可以避免包含任何特定于所使用的日志实现的代码。
2-实现原理
org.apache.commons.logging.impl.LogFactoryImpl#discoverLogImplementation
3-缺点，只能实现一种，通过静态绑定实现，不易扩展，适配器模式。

1.1.2 SLFJ

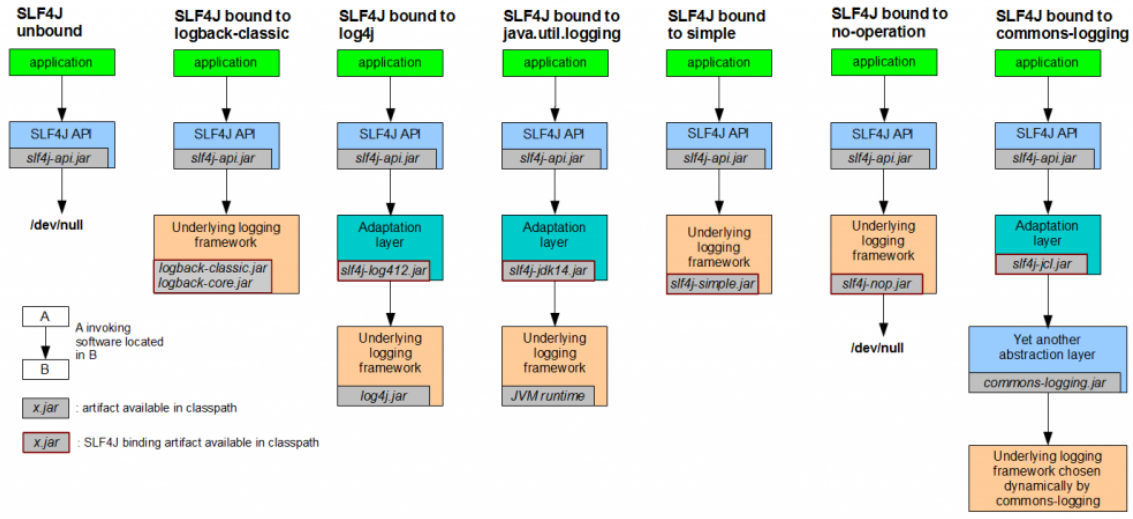
slf4j全 称为Simple Logging Facade for JAVA，java简单日志门面。类似于Apache Common-Logging，是对不同日志框架提供的一个门面封装(是接口而非实现)，可以在部署的时候不修改任何配置即可接入一种日志实现方案。但是，他在编译时静态绑定真正的Log库。使用SLF4J时，如果需要使用某一种日志实现，那么你必须选择正确的SLF4J的jar包的集合（各种桥接包）。
----基于OSGI模块化框架详解
特点：
动态加载、更新、和卸载模块而不用停止服务
实现系统的模块化、版本化，允许多版本bundule（模块）同时服务--注释，很多三方jar中的MANIFEST.MF进行子描述，就是一个例子

```
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Created-By: Apache Maven
Built-By: ceki
Build-Jdk: 1.8.0_121
Bundle-Description: The slf4j API
Bundle-Version: 1.7.30
Implementation-Version: 1.7.30
X-Compile-Source-JDK: 1.5
X-Compile-Target-JDK: 1.5
Implementation-Title: slf4j-api
Bundle-ManifestVersion: 2
Bundle-SymbolicName: slf4j.api
Bundle-Name: slf4j-api
Bundle-Vendor: SLF4J.ORG
Bundle-RequiredExecutionEnvironment: J2SE-1.5
Automatic-Module-Name: org.slf4j
Export-Package: org.slf4j;version=1.7.30, org.slf4j.spi;version=1.7.30
 , org.slf4j.helpers;version=1.7.30, org.slf4j.event;version=1.7.30
Import-Package: org.slf4j.impl;version=1.6.0
Service model允许模块、插件相互依赖但松耦合，分享服务更简单
```

1.2 常用的日志组成方案与应用场景

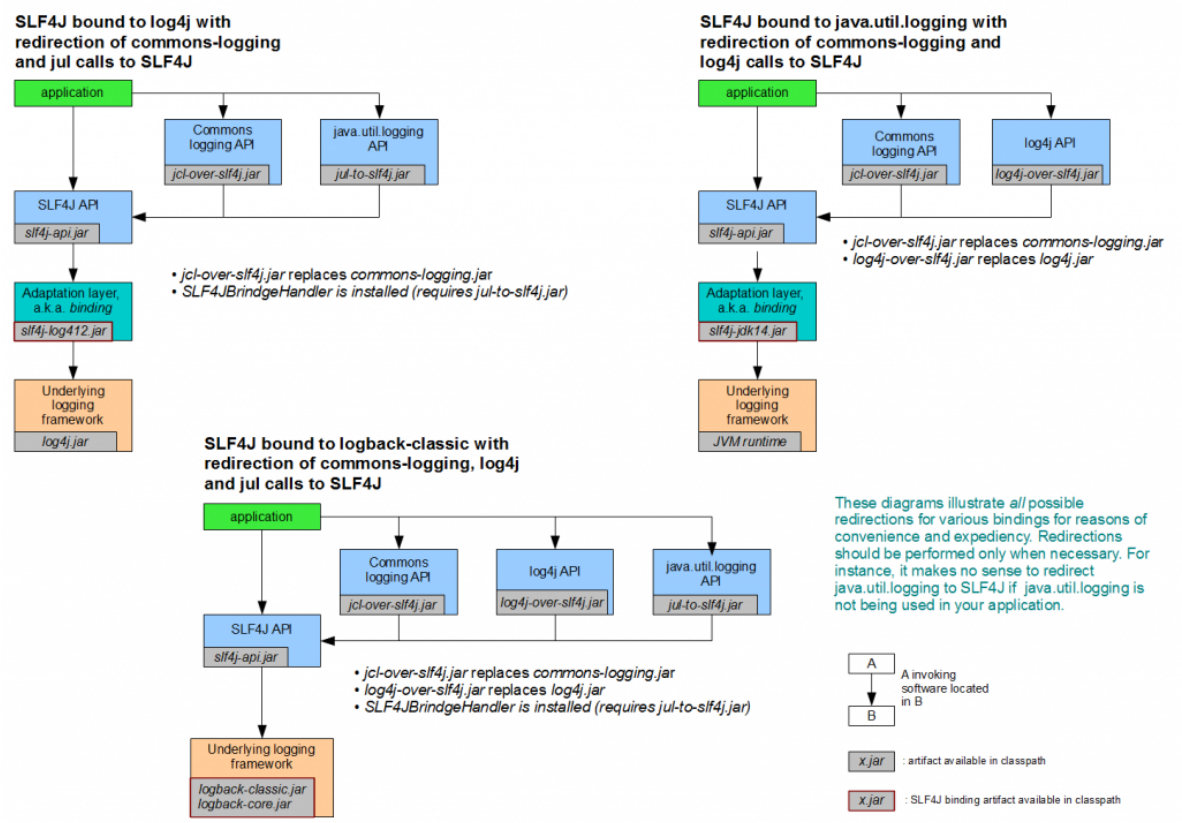
1.2.1 jcl

1.2.2 slfj




```
桥接包
slfj-log4j12.jar(桥接log4j)
slf4j-jdk14.jar(桥接jdk Logging)
slf4j-jcl.jar(桥接jcl)
log4j-slf4j-impl.jar(桥接log4j2)
logback-classic.jar(桥接logback)
```

1.2.3 日志切换，适配器



如果当前系统之中再用jcl打印日志，比如spring4，但这是想加入slfj来打印日志，就会出现两类日志输出，如何既觉

只要classpath当中指定了slfj适配器，包，即可无缝江源日志输出转移到slfj上来
jcl-over-slfj:转移 jcl日志至slf4j

log4j-over-slfj:转移log4j日志至slf4j

jul-over-slfj:转移jul日志至slf4j

名称	描述	相关JAR包
门面	slf4jAPI接口	slf4j-api.jar
桥接	用于slf4j连接对应日志实现	slfj-log4j12.jar,slfj-jdk14.jar,log4j-slf4j-impl,logback-classic,slf4j-jcl.jar
适配器	用于将原日志输出无缝转移到slf4j	cl-over-slf4j.jar,log4j-over-slfj,jul-over-slfj,
具体实现	日志的具体实现	log4j.jar,,logback,log4j2,java.util.logging

1.2.4 循环依赖

如果clappsth中既有桥接器也有适配器，日志会被踢来踢去，陷入死循环

1.3.slfj+log4j2 统一系统应用日志

由于系统组件中，可能采用了不同的日志体系，spring5之前,spring采用的是apache-common-log，spring5之后，采用spring-jcl

1.4 日志规范

1．【强制】应用中不可直接使用日志系统（Log4j、Logback）中的 API，而应依赖使用日志框架（SLF4J、JCL--Jakarta Commons Logging）中的 API，使用门面模式的日志框架，有利于维护和各个类的日志处理方式统一。
说明：日志框架（SLF4J、JCL--Jakarta Commons Logging）的使用方式（推荐使用 SLF4J）
使用 SLF4J：

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
private static final Logger logger = LoggerFactory.getLogger(Test.class);
```

使用 JCL：

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
private static final Log log = LogFactory.getLog(Test.class);
```

2．【强制】所有日志文件至少保存 15 天，因为有些异常具备以“周”为频次发生的特点。对于当天日志，以“应用名.log”来保存，保存在/home/admin/应用名/logs/目录下，过往日志格式为：{logname}.log.{保存日期}，日期格式：yyyy-MM-dd
说明：以 mppserver 应用为例，日志保存在/home/admin/mppserver/logs/mppserver.log，历史日志名称为 mppserver.log.2016-08-01
3．【强制】应用中的扩展日志（如打点、临时监控、访问日志等）命名方式：appName_logType_logName.log。logType:日志类型，如 stats/monitor/access 等；logName:日志描述。这种命名的好处：通过文件名就可知道日志文件属于什么应用，什么类型，什么目的，也有利于归类查找。
说明：推荐对日志进行分类，如将错误日志和业务日志分开存放，便于开发人员查看，也便于通过日志对系统进行及时监控。
正例：mppserver 应用中单独监控时区转换异常，如：mppserver_monitor_timeZoneConvert.log
4．【强制】在日志输出时，字符串变量之间的拼接使用占位符的方式。
说明：因为 String 字符串的拼接会使用 StringBuilder 的 append()方式，有一定的性能损耗。使用占位符仅是替换动作，可以有效提升性能。
正例：logger.debug("Processing trade with id: {} and symbol: {}", id, symbol);
5．【强制】对于 trace/debug/info 级别的日志输出，必须进行日志级别的开关判断。
说明：虽然在 debug(参数)的方法体内第一行代码 isDisabled(Level.DEBUG_INT)为真时（Slf4j 的常见实现 Log4j 和 Logback），就直接 return，但是参数可能会进行字符串拼接运算。此外，如果 debug(getName())
这种参数内有 getName()方法调用，无谓浪费方法调用的开销。
正例：

```
// 如果判断为真，那么可以输出 trace 和 debug 级别的日志
if (logger.isDebugEnabled()) {
 logger.debug("Current ID is: {} and name is: {}", id, getName());
}
```

6．【强制】避免重复打印日志，浪费磁盘空间，务必在 log4j.xml 中设置 additivity=false。
正例：<logger name="com.taobao.dubbo.config" additivity="false">

```
    case:
     <logger name="com.jd" level="DEBUG">
        <AppenderRef ref="console"/>
     </logger>
```

7. 【强制】生产环境禁止直接使用 System.out 或 System.err 输出日志或使用 e.printStackTrace()打印异常堆栈。
说明：标准日志输出与标准错误输出文件每次 Jboss 重启时才滚动，如果大量输出送往这两个文件，容易造成文件大小超过操作系统大小限制。
8. 【强制】异常信息应该包括两类信息：案发现场信息和异常堆栈信息。如果不处理，那么通过关键字 throws 往上抛出。
正例：logger.error(各类参数或者对象 toString() + "_" + e.getMessage(), e);

case:不允许记录日志后又抛出异常，因为这样会多次记录日志，只允许记录一次日志

9. 【强制】日志打印时禁止直接用 JSON 工具将对象转换成 String。
说明：如果对象里某些 get 方法被重写，存在抛出异常的情况，则可能会因为打印日志而影响正常业务流程的执行。
正例：打印日志时仅打印出业务相关属性值或者调用其对象的 toString()方法。
10.【推荐】谨慎地记录日志。生产环境禁止输出 debug 日志；有选择地输出 info 日志；如果使用 warn 来记录刚上线时的业务行为信息，一定要注意日志输出量的问题，避免把服务器磁盘撑爆，并记得及时删除这些观察日志。
说明：大量地输出无效日志，不利于系统性能提升，也不利于快速定位错误点。记录日志时请思考：这些日志真的有人看吗？看到这条日志你能做什么？能不能给问题排查带来好处？
11.【推荐】可以使用 warn 日志级别来记录用户输入参数错误的情况，避免用户投诉时，无所适从。如非必要，请不要在此场景打出 error 级别，避免频繁报警。
说明：注意日志输出的级别，error 级别只记录系统逻辑出错、异常或者重要的错误信息。
12.【推荐】尽量用英文来描述日志错误信息，如果日志中的错误信息用英文描述不清楚的话使用中文描述即可，否则容易产生歧义。
说明：国际化团队或海外部署的服务器由于字符集问题，使用全英文来注释和描述日志错误信息。

1.5 性能测试

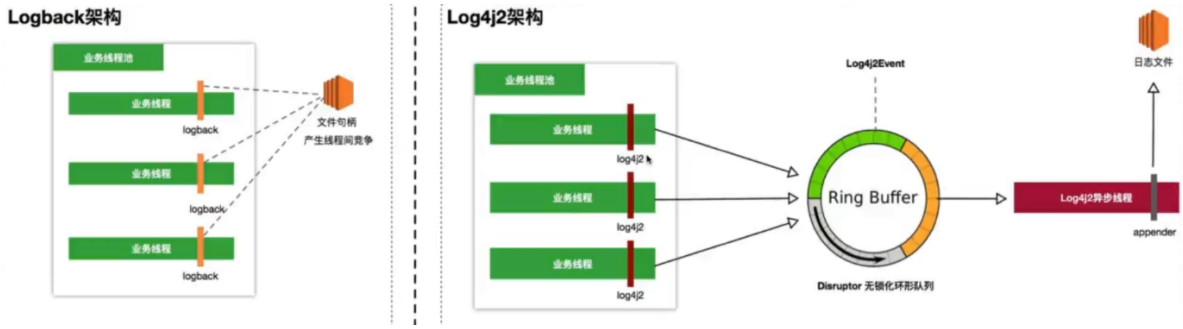
1.5.1 单线程

名称	未开启缓存，立即刷出	开启缓存，不立即刷出	异步-appender-未开启缓存，立即输出	异步appender-开启缓存，不立即输出
log4j	87.342秒	10.757秒	91.752	10.058秒
logback	81.617	5.547	100.245	10.69
log4j2	5.272/5.614/5.196	5.502/5.53/5.453	5.423/5.378/4.953	5.063/4.74/5.246(AsyncRoot)==>5.348/5.712/4.818（Async）

1.5.2 多线程

名称	未开启缓存，立即刷出	开启缓存，不立即刷出	异步appender-未开启缓存，立即输出	异步appender-开启缓存，不立即输出
log4j	102.823	13.324	87.966	10.651
logback	100.853	8.238	111.272	55.813
log4j2	8.164/6.94/7.073	8.02/6.597/8.009	8.11/7.485	8.178/8.226

机构图对比:



log4j2-AsyncLogger

Log4j2中的AsyncLogger的内部使用了Disruptor框架。
参考：https://www.cnblogs.com/yeyang/p/7944906.html

Disruptor简介
Disruptor是英国外汇交易公司LMAX开发的一个高性能队列，基于Disruptor开发的系统单线程能支撑每秒600万订单。

目前，包括Apache Strom、Log4j2在内的很多知名项目都应用了Disruptor来获取高性能。

Disruptor框架内部核心数据结构为RingBuffer，其为无锁环形队列。

单线程每秒能够处理600万订单，Disruptor为什么这么快？

a.lock-free-使用了CAS来实现线程安全
ArrayBlockingQueue使用锁实现并发控制，当get或put时，当前访问线程将上锁，当多生产者、多消费者的大量并发情形下，由于锁竞争、线程切换等，会有性能损失。

Disruptor通过CAS实现多生产者、多消费者对RingBuffer的并发访问。CAS相当于乐观锁，其性能优于Lock的性能。

b.使用缓存行填充解决伪共享问题
计算机体系结构中，内存的访问速度远远低于CPU的运行速度，在内存和CPU之间，加入Cache，CPU首先访问Cache中的数据，CaChe未命中，才访问内存中的数据。

伪共享：Cache是以缓存行（cache line）为单位存储的，当多个线程修改互相独立的变量时，如果这些变量共享同一个缓存行，就会无意中影响彼此的性能。

关于伪共享的深度分析，可参考《伪共享，并发编程的性能杀手》这篇文章。

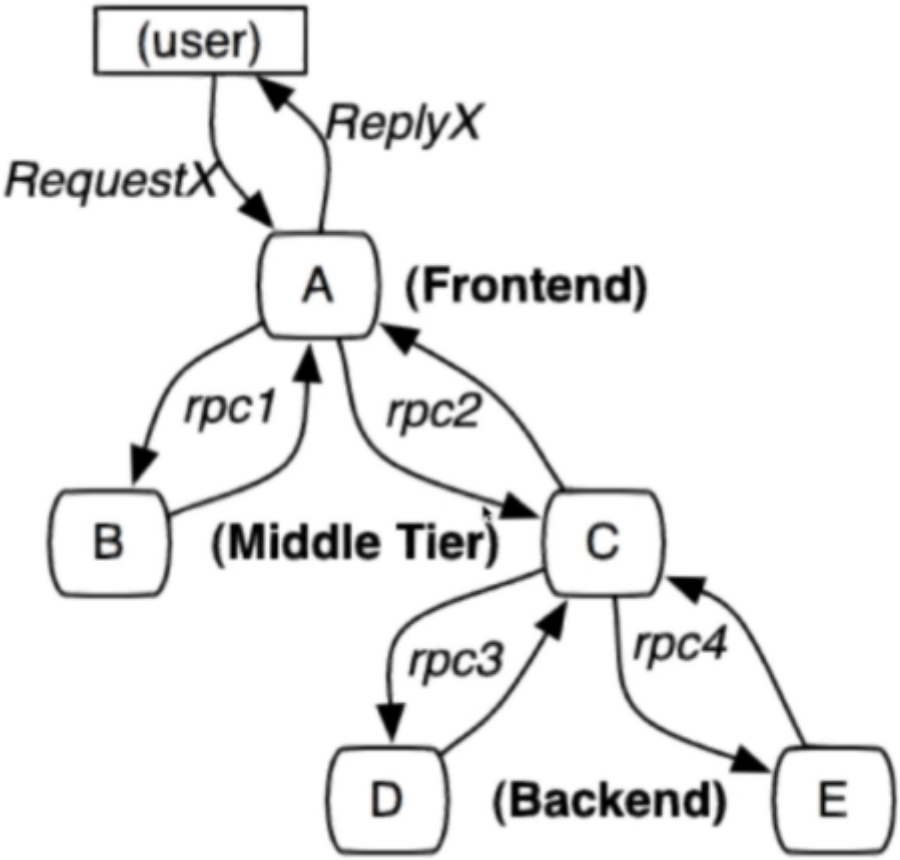
日志输出方式

sync	同步打印日志，日志输出与业务逻辑在同一线程内，当日志输出完毕，才能进行后续业务逻辑操作
Async Appender	异步打印日志，内部采用ArrayBlockingQueue，对每个AsyncAppender创建一个线程用于处理日志输出。
Async Logger	异步打印日志，采用了高性能并发框架Disruptor，创建一个线程用于处理日志输出。

1.6 日志的全链路追踪 简介

1.6.1 全链路追踪的解决方案

全链路追踪的背景



上图是一个典型的微服务调用链路，面对的场景问题如下：

1-如果D服务是一个关键服务，返回结果错误，无论是日志，还是监控平台，并不能很快捷的定位问题出现在了那里，因为不能串联整个调用链路的流程

2-当对某一个服务架构升级或者改造的时候，不好评估影响范围，不明确服务之间的依赖关系，给技术决策带来了困难

3-性能瓶颈，整个调用链路那个环节耗时比较久

4-当一次请求结束后，不好确定执行顺序，都给业务逻辑上的理解带来了困难

需要解决问题：

1-串用调用链，快速定位问题

2-厘清服务依赖关系

3-进行各个服务接口的性能分析

4-跟踪业务流的处理顺序

已有方案

1-Google Dapper

2-Twitter Zipkin

3-Spring Cloud Sleuth

3-1 与springboot及spring组件无缝集成

3-2支持Zipkin输出（mysql，es）

3-3 支持MQ和HTTP方式传输

1.6.2 Spring Cloud Sleuth 介绍

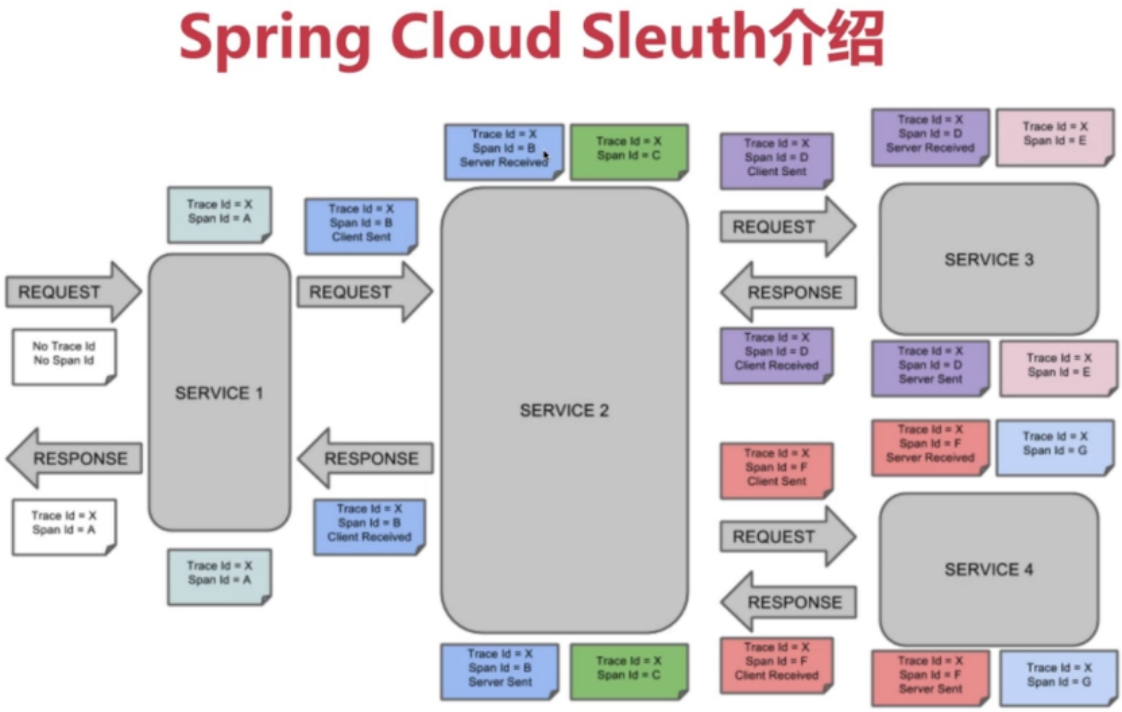
基本概念:

Trace（链路）

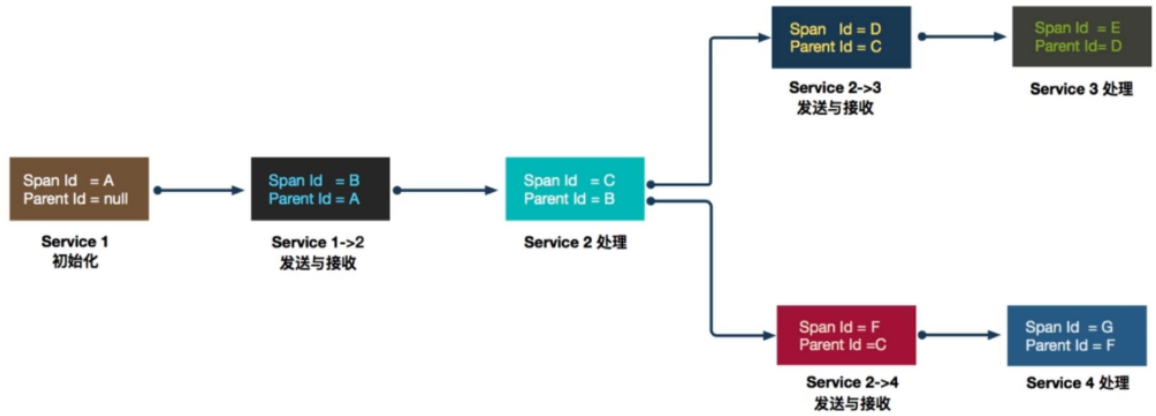
Span（跨度）

Annotation（标注）：CS（发送请求），SR（接受请求），SS（相应发送），CR（相应被客户端接收）

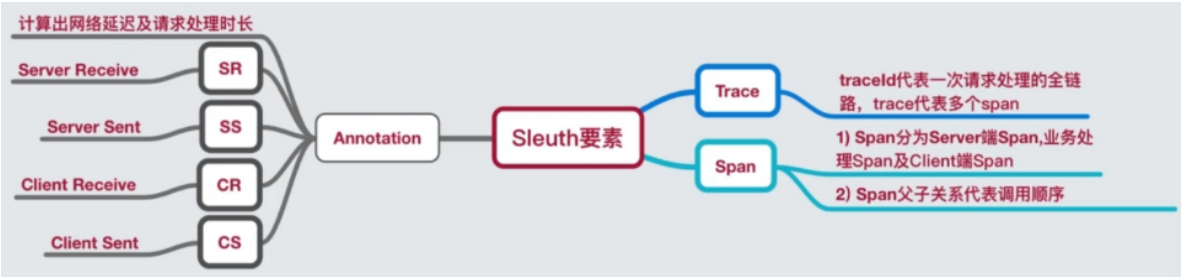
架构图



流程图



基本概念



1.6.3 分布式日志检索解决方案-ELK

E：Elasticsearch ---存储

L：LogStash ---收集

K：Kibana ---展示

流程