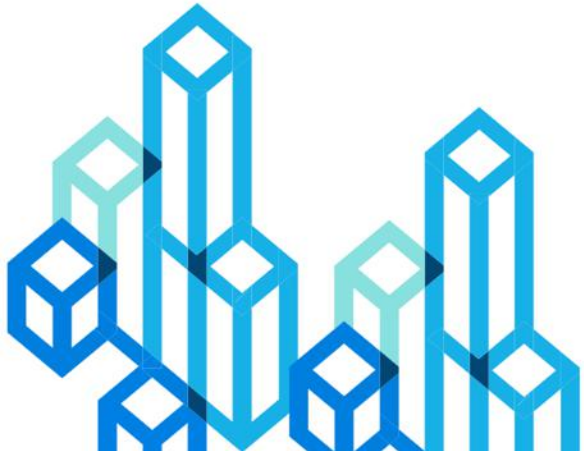
Takin客户端的性能优化实践

陆学慧 @ 数列科技



大家好，我是平威

交个朋友多交流

陆学慧（平威）

- 数列科技联合创始人 & CTO
- 全球首款生产压测开源产品Takin的主要作者
- 曾就职于阿里巴巴中间件团队

01 Takin架构简介

02 日志模块高性能设计和优化

03 性能防腐之路

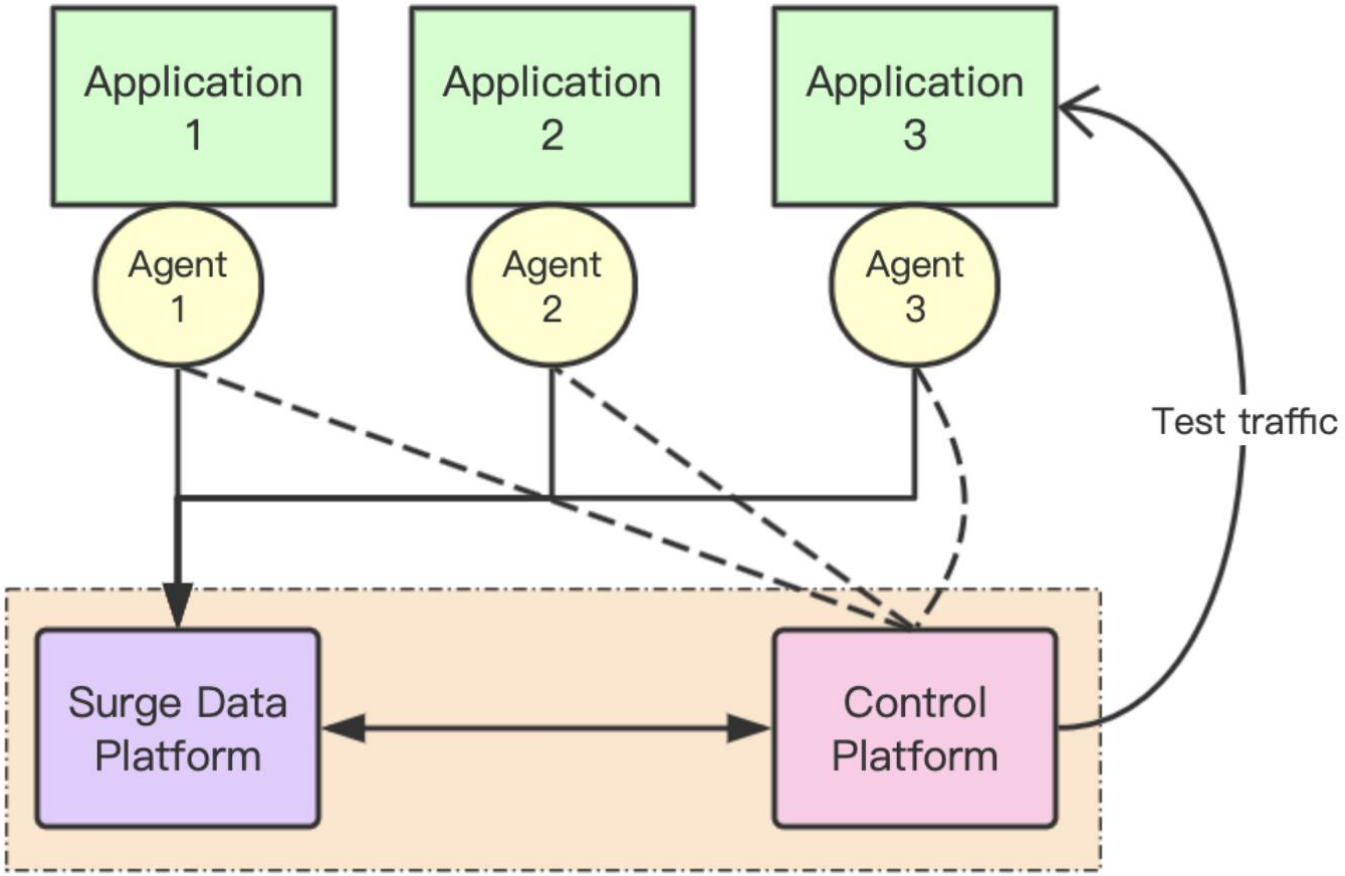
04 Takin-LinkAgent后续规划

Takin架构介绍

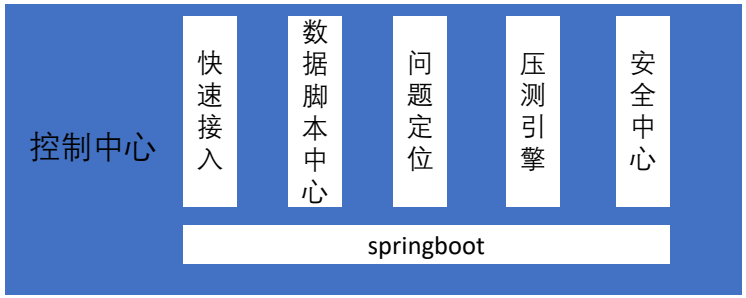
全球首个开源的生产环境全链路产品
从客户端到服务端一站式流量发起与诊断功能

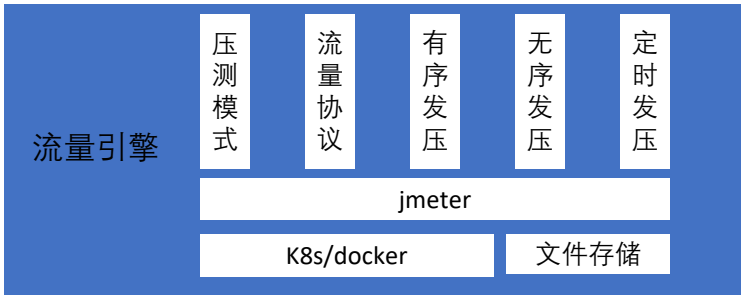
01

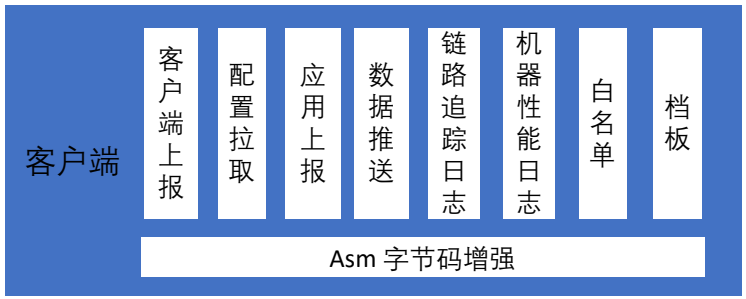
Takin架构简介

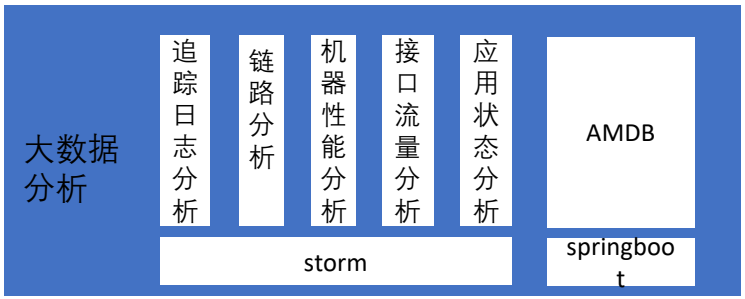

Application 1
Application 2
Application 3
Agent 1
Agent 2
Agent 3
Test traffic
Surge Data Platform
Control Platform

Takin架构简介


控制中心
快速接入
数据脚本中心
问题定位
压测引擎
安全中心
springboot


流量引擎
压测模式
流量协议
有序发压
无序发压
定时发压
jmeter
K8s/docker
文件存储

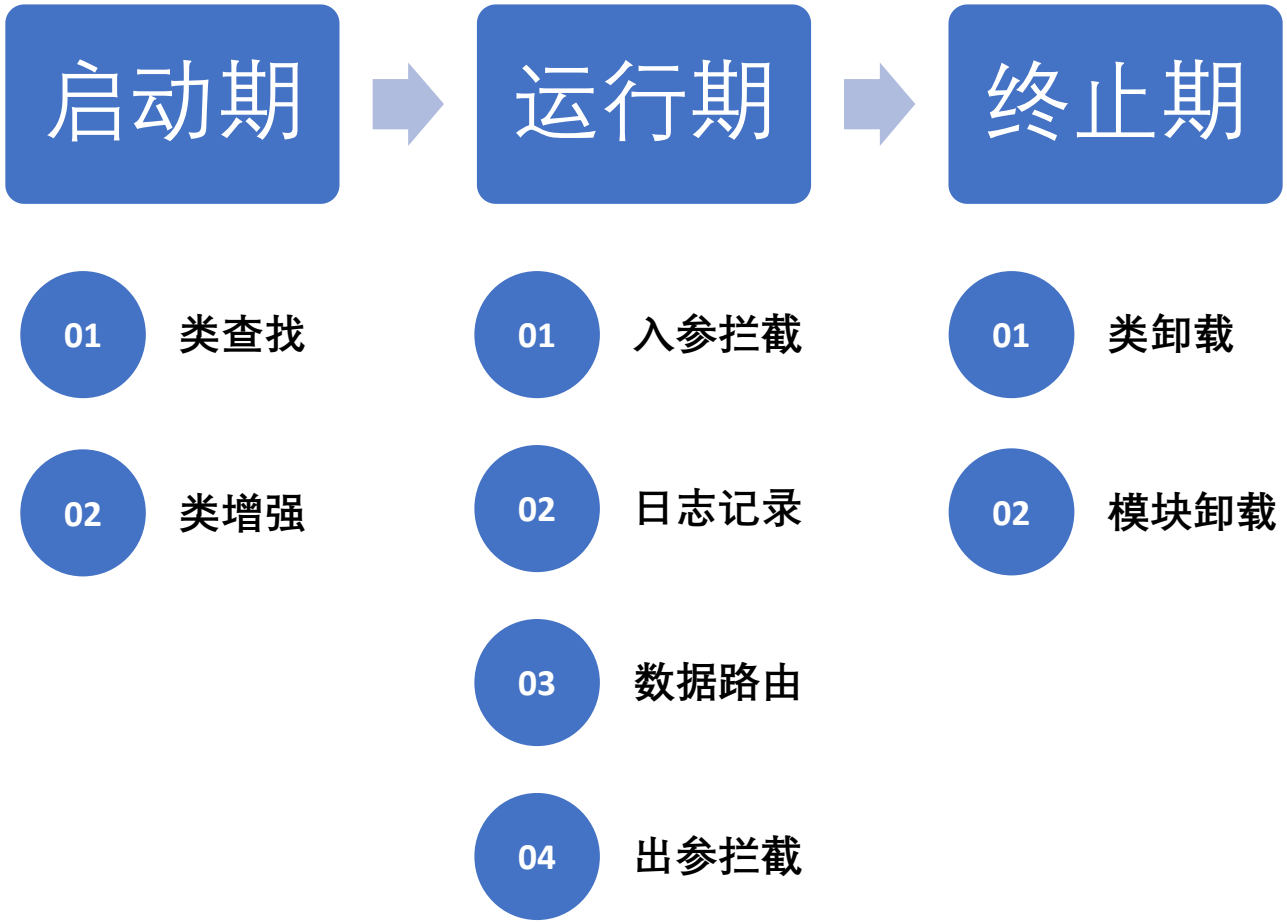

客户端
客户端上报
配置拉取
应用上报
数据推送
链路追踪日志
机器性能日志
白名单
档板
Asm 字节码增强


大数据分析
追踪日志分析
链路分析
机器性能分析
接口流量分析
应用状态分析
AMDB
storm
springboot

日志模块高性能设计和优化

02

LinkAgent的核心流程



日志高性能设计诉求

高性能

- 应用压测过程中会产生大量的日志
- 高性能是日志模块的第一诉求

日志尽量不丢

- 高压力下，处理不过来的情况可以选择丢弃

对应用CPU消耗少

- 日志拼接、打印过程效果很多CPU，需要尽可能少的使用到CPU

对应用内存消耗少

- 运行期产生大量的日志，占用应用不少内存，增加内存和GC的压力，需要有效的减少内存的消耗

日志框架 VS 自研

通用日志

- 应用诊断日志、应用打印异常信息，排查问题用，一般是给人看的
- 考虑通用性，需要有很强的扩展性设计：配置化、日志级别（LogLevel）、日志格式（Layout）、层次结构（Category）、多种输出实现（Appender）
- 通用性诉求大于性能诉求，对于写文件、序列化、格式化等优先考虑扩展性

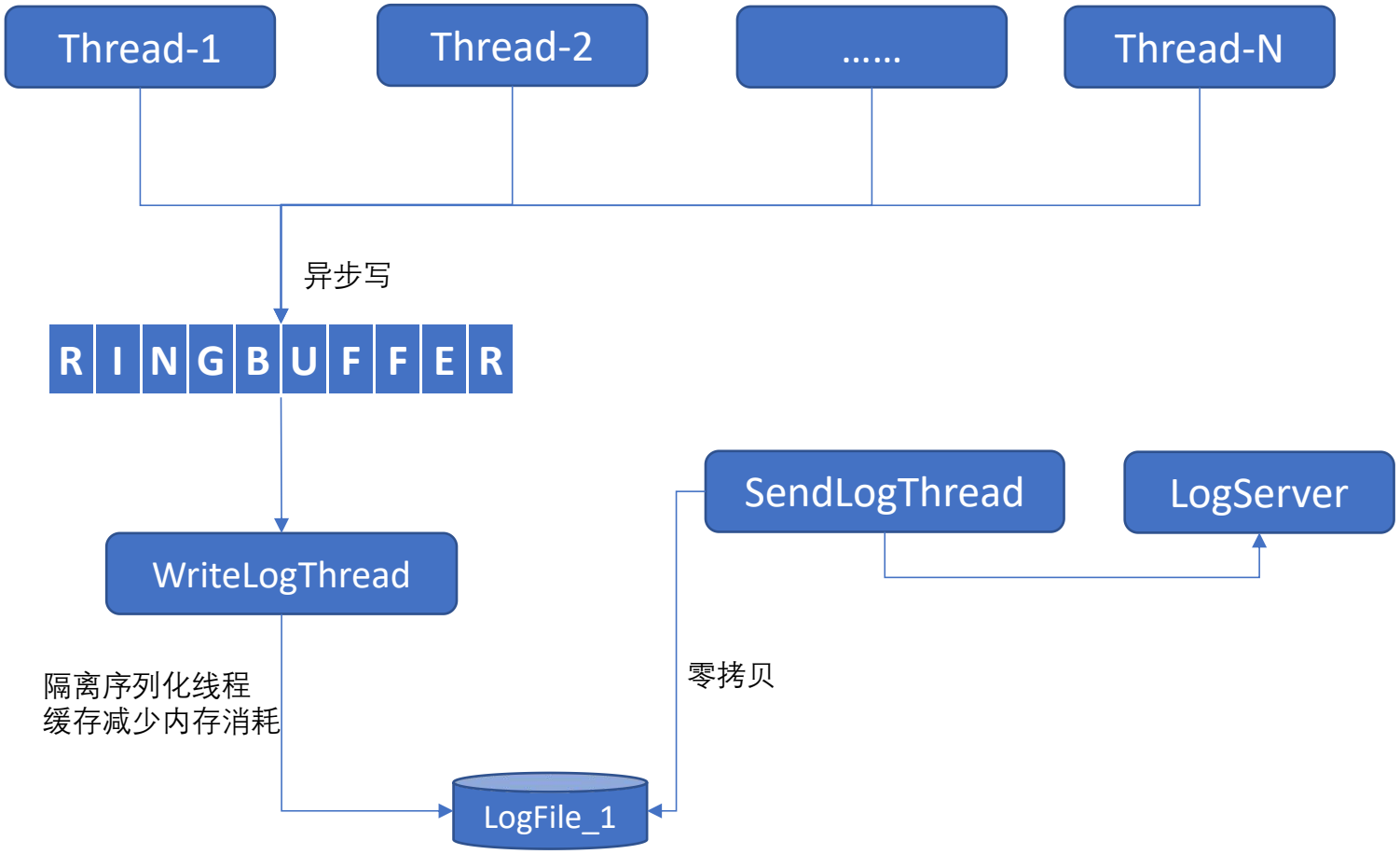
数据日志

- 一般是用来做监控和数据分析的，可以人肉临时分析，也可以给机器分析，要求格式比较固定
- 性能要求极高，因为一般是非主链路，要求对业务影响越小越好
- 对事务性要求不高，极端情况可丢弃
- 不太愿意付出特别多的成本

交易日志

- 一般在日志式文件系统、NoSQL、DB 中使用，如binlog
- 不需要通用，追求极致场景下的高性能和高事务性
- 愿意付出高昂的成本

日志高性能设计


Thread-1
Thread-2
……
Thread-N
异步写
RINGBUFFER
WriteLogThread
SendLogThread
LogServer
隔离序列化线程
缓存减少内存消耗
零拷贝
LogFile_1

日志高性能设计——异步化解耦

	ArrayBlockingQueue	RingBuffer
简要信息	大部分日志框架异步写入的默认实现	性能超高的环形数组
有界性	有界，定长的特性方便做日志的限流	类似ABQ
锁	锁	无锁（很少的CAS，可以批量）
内存友好	内存一次性申请，创建完后不用另行分配内存	类似ABQ
性能（单线程）	189万/秒	247万/秒
性能（64线程）	178万/秒	1844万/秒（约提升10倍左右）

日志设计优化——隔离序列化线程

```
logger.info("Hello GIAC!");
```

日志格式处理 Layout → 序列化 → 输出处理 Append

- 阻塞业务线程时间长，直接影响业务RT
- 多线程写入文件，锁竞争激烈
- 异常情况下急剧恶化业务RT

- 全内存操作，对业务几乎RT几乎无影响
- 单线程顺序写入，竞争少、写入快、消耗少

日志对象放入队列

读出日志对象序列化 → 批量顺序写入日志

日志设计优化

固定缓存大小减少GC

大量产生的日志，在序列化时会占用大量的内存，提高GC的频率

统一序列化线程后，申请固定大小的8KB缓存重复使用

对于写文件来说，一个4KB大小能完整写完一个文件块

通过控制固定缓存大小，从而控制IOPS

零拷贝发送日志

mmap对于大文件传输有一定优势

对于小文件会产生大量碎片

在多个进程同时操作文件时可能产生引发 coredump 的 signal

最终选择了sendFile方式

自适应的推送频率

固定频率的读取，在资源消耗和实时性上比较难以平衡

推送的日志大小每次等于1MB，立即继续推送

如果不足1MB，等待1秒

连续不满足，持续增加，最多10秒等待

序列化选择UTF-8编码

JVM内码使用Unicode

Unicode转GBK 要查表计算

Unicode 转UTF-8只要算

性能有10%左右的差距

性能防腐之路

03

性能防腐思路

目标严格控制在5%

◆性能基线Benchmark

- RT 20ms - 50ms 1000并发
- RT 50ms - 200ms 1000并发
- ……
- 持续补充

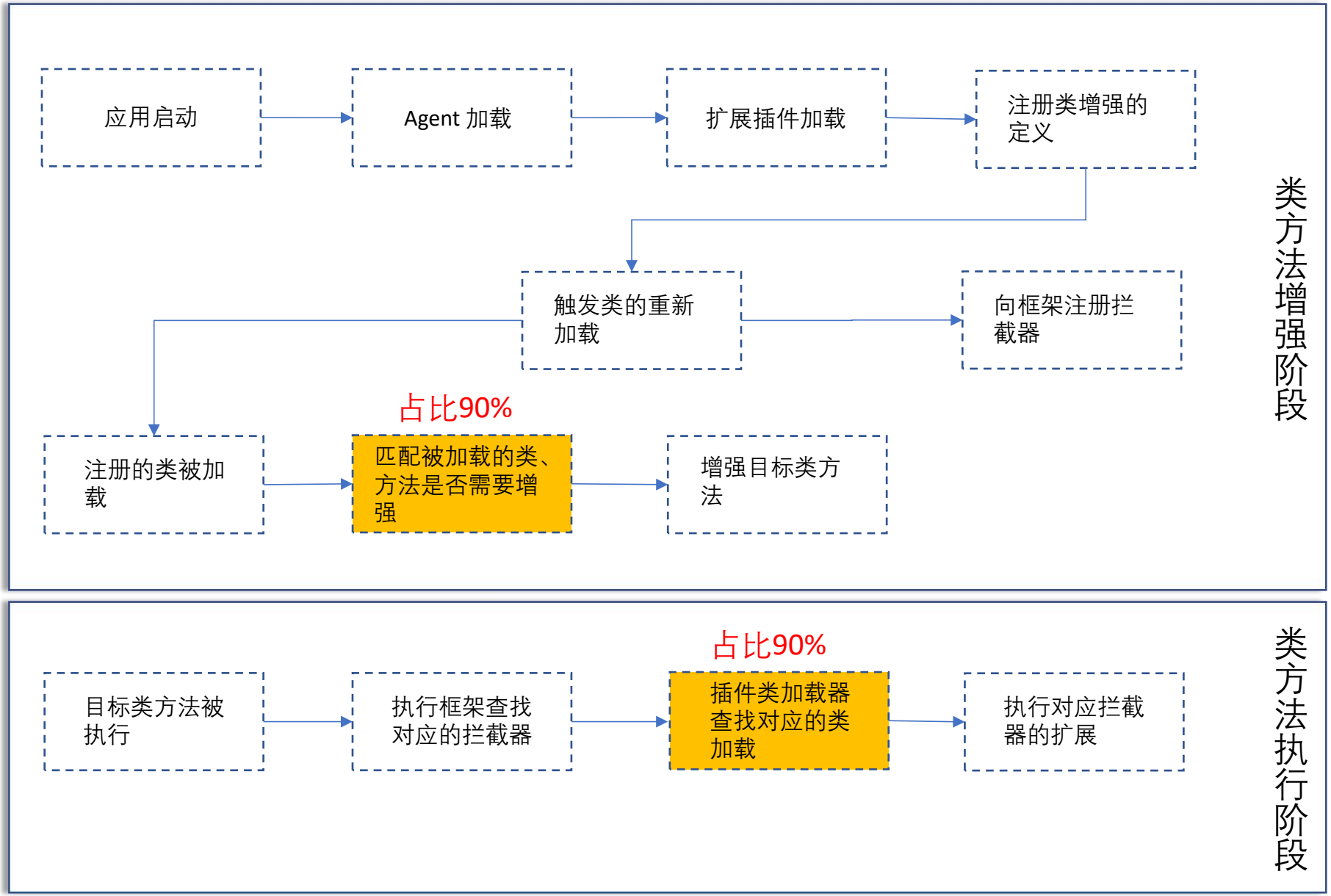
◆ 与流水线集成，自动化验证

- 版本发布
- 定期验证

◆ 更多真实场景的用户反馈

9%	5%	3%
2018	2020	2021

应用启动时间增加200%



优化重点一：增强匹配

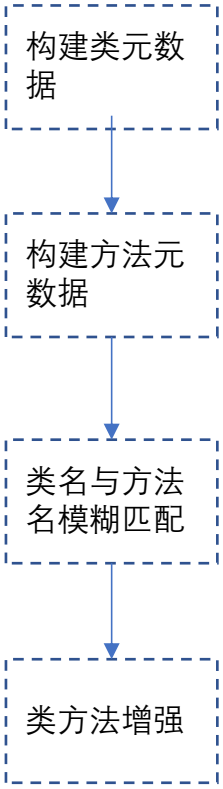
特点

- ✓ 类名与方法名支持模糊匹配
- ✓ 大部分增强全类名匹配
- ✓ 大部分方法全方法名匹配

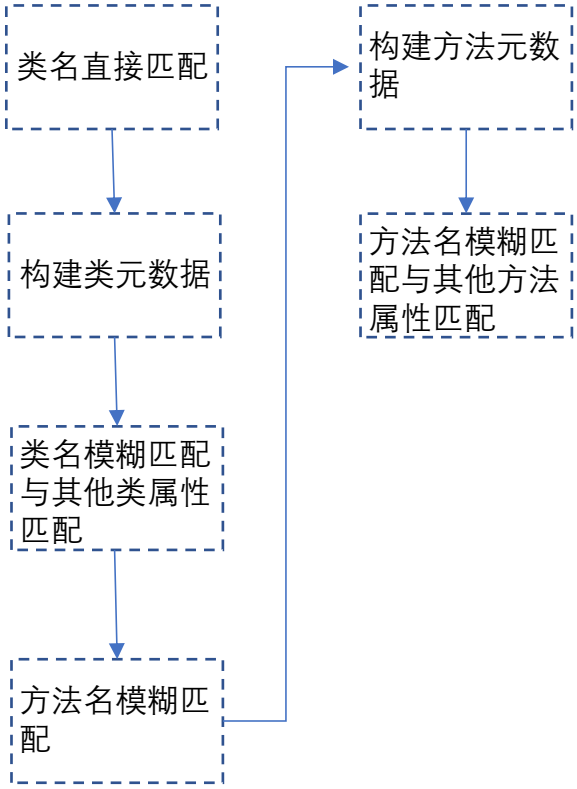
重点优化部分

- ✓ 大多数快速匹配
- ✓ 元数据延迟加载
- ✓ 精确控制模糊匹配

优化前

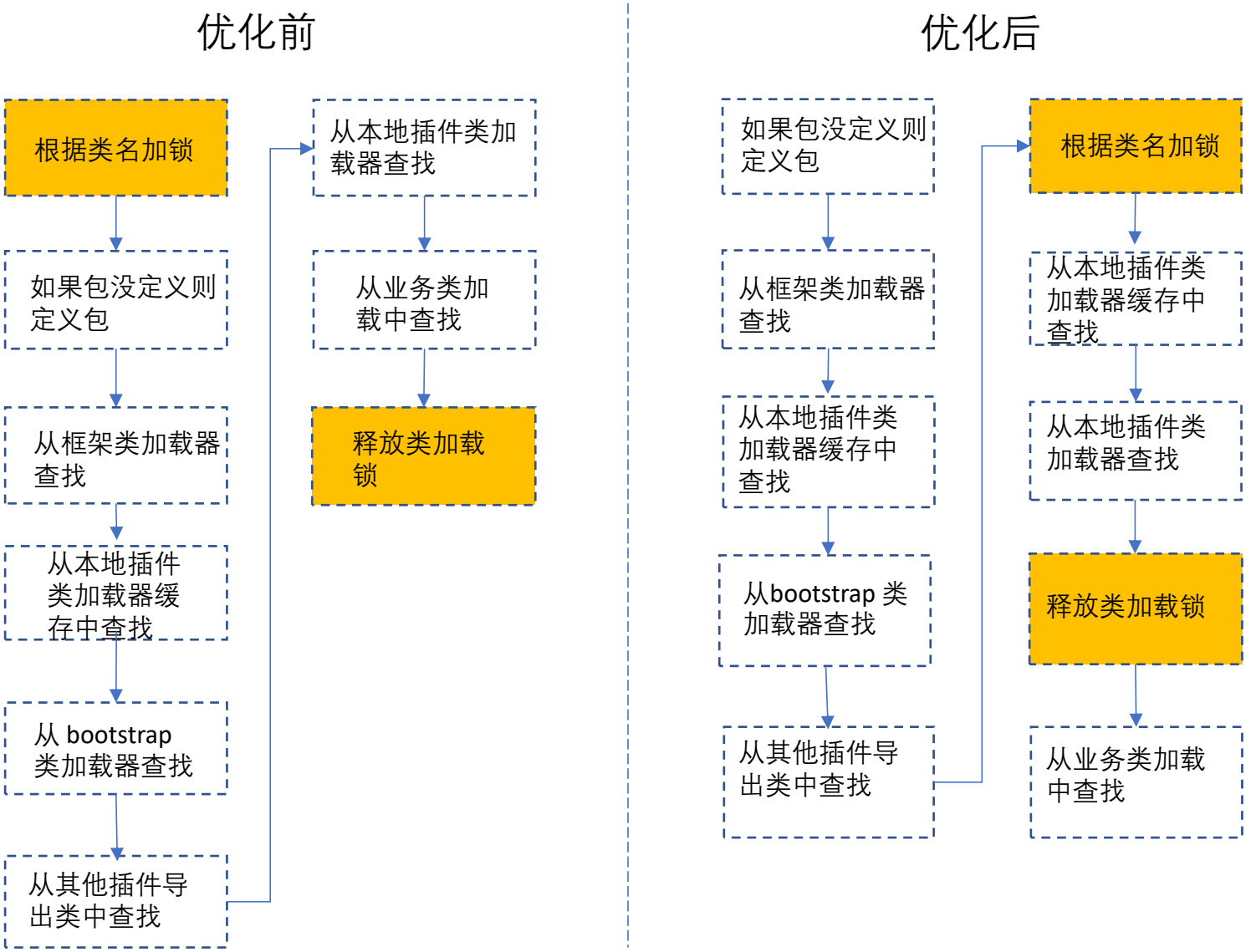


优化后



优化重点二：类加载优化

✓ 优化锁粒度，减少类加载时的锁争抢



时间获取性能优化

```
System.currentTimeMillis();
```

```
jlong os::javaTimeMillis() {
  timeval time;
  int status = gettimeofday(&time, NULL);
  assert(status != -1, "linux error");
  return jlong(time.tv_sec) * 1000  +  jlong(time.tv_usec / 1000);
}
```

高并发

调用gettimeofday()需要从用户态切换到内核态

gettimeofday()的表现受系统的计时器（时钟源）影响，在HPET计时器下性能尤其差

系统只有一个全局时钟源，高并发或频繁访问会造成严重的争用

时间获取性能优化

```
private SystemClock() {
    this.now = new AtomicLong(System.currentTimeMillis());
    scheduler = Executors.newSingleThreadScheduledExecutor(runnable -> {
        Thread thread = new Thread(runnable, "SystemClockScheduled");
        thread.setDaemon(true);
        return thread;
    });
    scheduler.scheduleAtFixedRate(() -> now.set(System.currentTimeMillis()), 1, 1, TimeUnit.MILLISECONDS);
}

public static long now() {
    return getInstance().now.get();
}
```

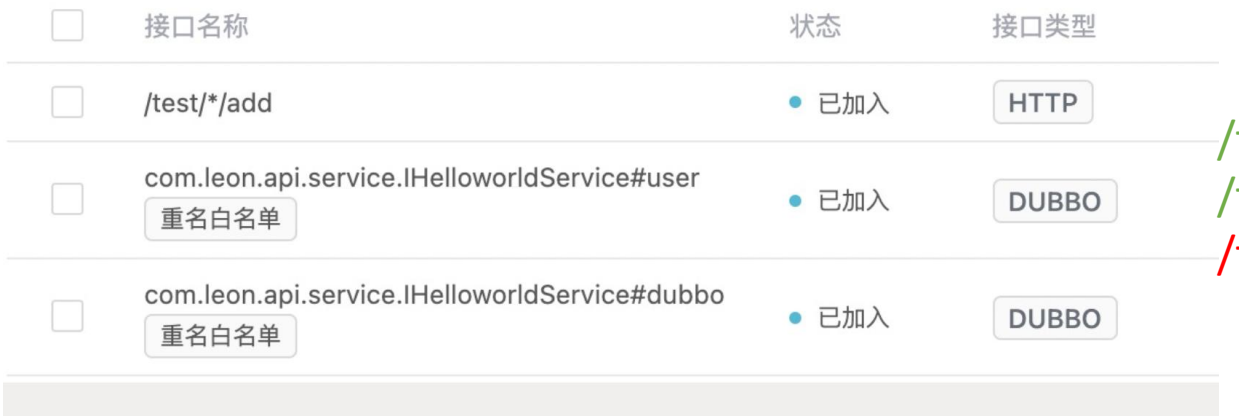
核心思路：利用一个异步线程独立维护进程内的毫秒级时钟，避免过多的系统调用和资源的竞争

缺点：不精确 日志记录的数据可以丢，但是不能错，会给排查问题带来巨大的麻烦

减少使用频率

性能没有那么差，在Windows下该方法性能表现优秀
在Linux系的操作系统中用**TSC、JVM_CLOCK**这些更高rating的时钟源性能也没有问题
如果是使用**HPET、ACPI_PM**这类高精度时钟源的时候，高并发下该方法的调用开销明显变高

白名单的匹配优化



/test/a/add YES
/test/GIAC/add YES
/test/add NO

全局白名单

公司所有自动扫描和人工配置的名单
全局共用一份，每个节点配置相同
大型公司有10万+

私域白名单

引入应用私域白名单，节点只拉取跟自己相关的白名单配置
增加确认机制，引入了一些工作量，提升了安全性和性能
单节点数量下降至200左右

正则匹配

自定义匹配

split+indexOf
单点性能提升10倍以上

异常的优化

```java
public synchronized Throwable fillInStackTrace() {
    if (stackTrace != null ||
        backtrace != null /* Out of protocol state */ ) {
        fillInStackTrace(0);
        stackTrace = UNASSIGNED_STACK;
    }
    return this;
}
```

```java
public PressureMeasureError(Throwable e) {
    super(e);
}

public PressureMeasureError(Throwable e, boolean isClusterTest) {
    super(e);
    this.isClusterTest = isClusterTest;
}

@Override
public Throwable fillInStackTrace() {
    return this;
}
```

getHostName的优化

追踪消息发送的具体地址，获取服务器地址和端口号：

```
String port = "";
if (msg.getStoreHost() != null && msg.getStoreHost() instanceof InetSocketAddress) {
    InetSocketAddress address = (InetSocketAddress) msg.getStoreHost();
    storeHost = address.getHostName();
    port = String.valueOf(address.getPort());
} else {
    storeHost = StringUtils.substring(msg.getStoreHost().toString(), 1);
}
```

```
String getHostName(boolean check) {
    if (holder().getHostName() == null) {
        holder().hostName = InetAddress.getHostFromNameService(this, check);
    }
    return holder().getHostName();
}
```

```
if (msg.getStoreHost() != null && msg.getStoreHost() instanceof InetSocketAddress) {
    InetSocketAddress address = (InetSocketAddress) msg.getStoreHost();
    storeHost = address.getAddress() == null ? null : address.getAddress().getHostAddress();
    port = String.valueOf(address.getPort());
} else {
    storeHost = StringUtils.substring(msg.getStoreHost().toString(), 1);
}
```

DNS服务请求量剧增
消息发送失败

获取IP地址

其他优化

01 字符串分割

String.split使用正则实现，大量调用时可以使用StringUtils.split替代，性能平均提升一倍以上

02 字符串查找

String.ind
正则强不
合并的尽
缓存的尽

03 拒绝正则表达式

不要使用正则表达式

04 最小化锁粒度

05 线程池统一管理

理会避免大量无用的线程切换

06 多版本编译替代反射调用

兼容多版本中间件API的时候，需要用反射的方式判断API是否存在，不存在会抛出异常，大量异常会验证影响性能。通过多版本编译的方式，改成直接调用

Devils in the details！

魔鬼在细节

Takin-LinkAgent后续规划

标准化协议、开放、多语言、SerivceMesh

04

Takin-LinkAgent后续规划

标准化协议 定义一套开放的采集协议，避免过度采集

数据共享 与行业其他产品一起共享数据，避免多次采集

ServiceMesh支持 serviceMesh、跨语言的支持

谢谢大家

Takin的Github地址

Takin社区微信群

麦思博(msup)有限公司是一家面向技术型企业的培训咨询机构，携手2000余位中外客座导师，服务于技术团队的能力提升、软件工程效能和产品创新迭代，超过3000余家企业续约学习，是科技领域占有率第1的客座导师品牌，msup以整合全球领先经验实践为己任，为中国产业快速发展提供智库。

高可用架构公众号主要关注互联网架构及高可用、可扩展及高性能领域的知识传播。订阅用户覆盖主流互联网及软件领域系统架构技术从业人员。

高可用架构系列社群是一个社区组织，其精神是“分享+交流”，提倡社区的人人参与，同时从社区获得高质量的内容。