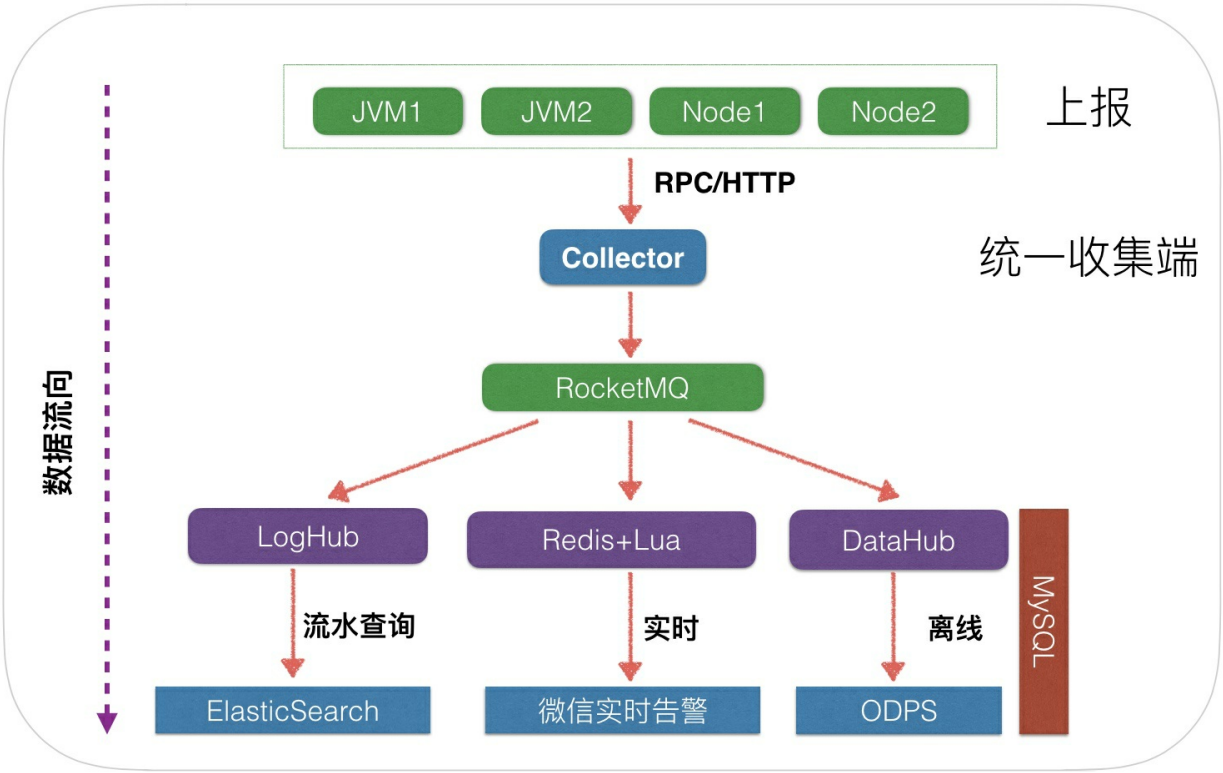
这篇文章我们将介绍一个可落地的最简单的 APM 架构应该如何设计。将从整体的架构概览、agent 上报、数据收集、数据处理四个部分来介绍。

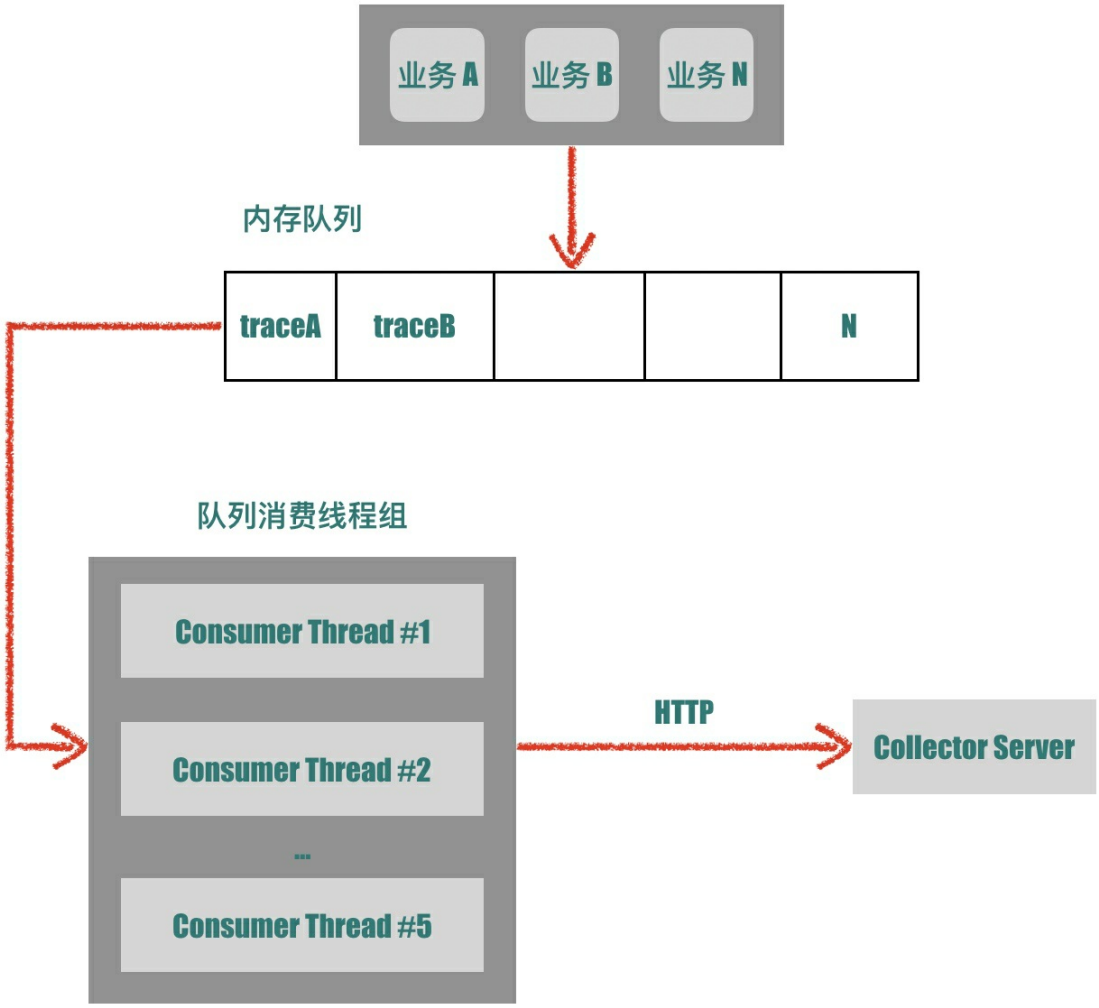
0x01 架构概览

以数据流向的角度看整个后端 APM 的架构如下



0x02 Agent 上报

Agent 上报端采用一个大小为 N 的内存队列缓存产生的调用 trace，N 一般为几千，业务代码写完队列立即 return，如果此时因为队列消费太慢，则允许丢数据，以免造成内存暴涨影响正常业务。设置一个心跳包定时上报当前队列的大小，如果超过 N 的 80%，则进行告警人工干预。



因为 APM 会产生调用次数放大，一次调用可能产生几十次上百次的的链路调用 trace。因此数据一定要合并上报，减少网络的开销。这个场景就是 **合并上报，指定时间还没达到批量的阈值，有多少条报多少条**，针对此场景我写了一个非常简单的工具类，用 BlockingQueue 实现了带超时的批量取

- Add 方法

```
 // 如果队列已满，需要超时等待一段时间，使用此方法
queue.offer(logItem, 10, TimeUnit.MILLISECONDS)
// 如果队列已满，直接需要返回add失败，使用此方法
queue.offer(logItem)
```

- 批量获取方法
  BlockingQueue的批量取方法drainTo()不支持超时特性，但

是注意到poll() 支持，结合这两者的特性我们做了如下的改动（参考部分 Guava 库的源码）

```
public static <E> int batchGet(BlockingQueue<E>
q,Collection<? super E> buffer, int numElements,
long timeout, TimeUnit unit) throws
InterruptedException {
    long deadline = System.nanoTime() +
unit.toNanos(timeout);
    int added = 0;
    while (added < numElements) {
        // drainTo非常高效，我们先尝试批量取，能取多少是
多少，不够的poll来凑
        added += q.drainTo(buffer, numElements -
added);
        if (added < numElements) {
            E e = q.poll(deadline -
System.nanoTime(), TimeUnit.NANOSECONDS);
            if (e == null) {
                break;
            }
            buffer.add(e);
            added++;
        }
    }
    return added;
}
```

完整代码如下

```
private static final int SIZE = 5000;
private static final int BATCH_FETCH_ITEM_COUNT =
50;
```

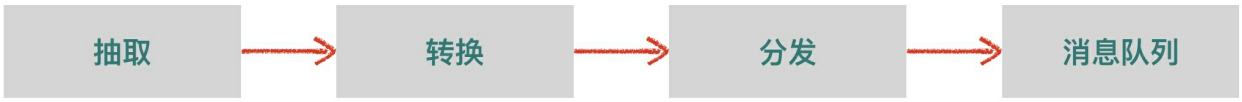
```
private static final int MAX_WAIT_TIMEOUT = 30;
private BlockingQueue<String> queue = new
LinkedBlockingQueue<>(SIZE);

public boolean add(final String logItem) {
    return queue.offer(logItem);
}
public List<String> batchGet() {
    List<String> bulkData = new ArrayList<>();
    batchGet(queue, bulkData,
BATCH_FETCH_ITEM_COUNT, MAX_WAIT_TIMEOUT,
TimeUnit.SECONDS);
    return bulkData;
}
public static <E> int batchGet(BlockingQueue<E>
q,Collection<? super E> buffer, int numElements,
long timeout, TimeUnit unit) throws
InterruptedException {
    long deadline = System.nanoTime() +
unit.toNanos(timeout);
    int added = 0;
    while (added < numElements) {
        added += q.drainTo(buffer, numElements -
added);
        if (added < numElements) {
            E e = q.poll(deadline -
System.nanoTime(), TimeUnit.NANOSECONDS);
            if (e == null) {
                break;
            }
            buffer.add(e);
            added++;
        }
```

```
    }
    return added;
}
```

0x03 数据收集服务

数据收集端的职责是对上报上来的数据进行清洗、转换、流转给下一级消息队列进行派发，相当于大数据 ETL（Extract-Transform-Load）流程。数据收集服务要求非常高的性能，如果收集服务慢，上报端队列处理不及时，就会丢数据了。因此这部分的逻辑要尽可能的简单和可靠。



我们现在这部分的接口响应时间的 99 百分位都是在 1ms 以内处理完

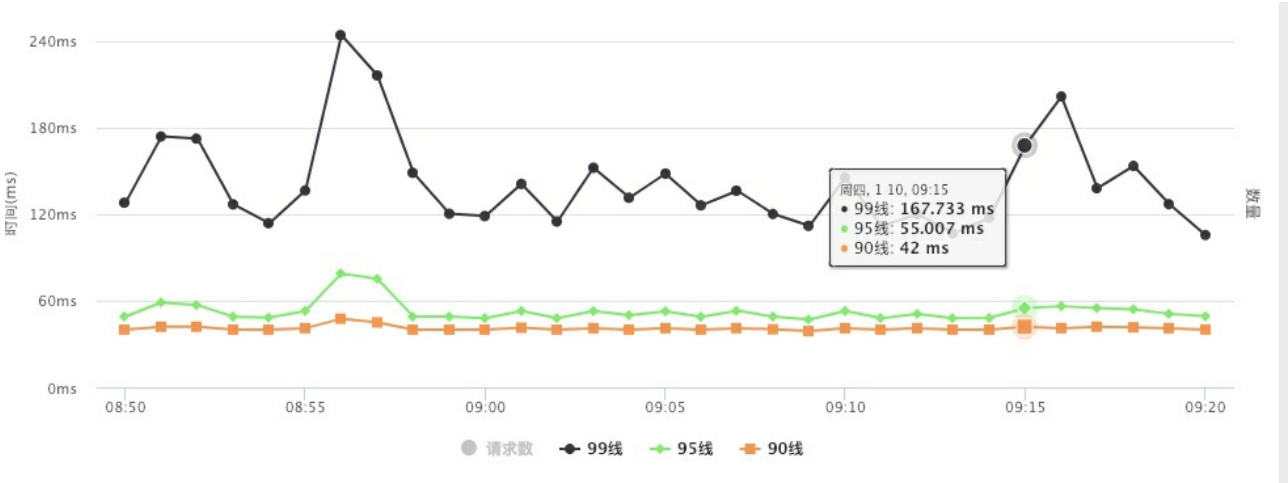
0x04 数据处理端

数据处理可以分为三大部分：流水查询、实时告警、离线报表分析

- 流水查询
  因为我们经常根据一些用户 id 或者特定的字符串来检索整条链路调用，我们在技术选型上选择了 ElasticSearch 来做存储和模糊查询

ElasticSearch 在海量数据存储和文本检索方面的巨大优势和，结合 ELK 工具套件，可以非常轻松的实现数据可视化、运维、监控、告警。比如我们查询特定调用的昨天一整天响应时间百分位，可以非常方便的统计出来

也可以方便统计某个项目整体的响应时间百分位，可以用来衡量服务 SLA 水平



- 实时告警
  实时数据处理可以用时序数据库，也可以用 Redis + Lua 的方式，有告警的情况下可以达到分钟级别的微信、邮件通知，也可以在业务上做告警收敛，避免告警风暴
- 离线处理
  离线处理主要产生一些实时性要求没那么高、ELK 无法生成的

复杂报表，技术栈上有很多可供选择的，我们这里选择的是最传统的阿里云 ODPS。

上面介绍了 APM 数据在后端的完整的流转过程，每个阶段都可以结合业务自己的特点做个性化的技术选型。

0x05 小结

这篇文章我们讲解了整个后端 APM 的数据流向和基本的架构设计。主要有几个要点：

- 第一，数据上报端，采用异步批量合并的方式。
- 第二，数据收集服务对数据做 ETL，且性能要尽可能的响应快速。
- 第三，数据处理端分为了流水查询、实时告警、离线报表分析三个模块，其中实时告警可以使用时序数据库或者 Redis 等方式来实现，告警需要做收敛。

0x06 思考

留一道思考题：在我们线上的 APM 系统中，用到了 Disruptor 高性能无锁队列，你觉得它可以用在 APM 的哪些地方，有哪些优势?

欢迎你在留言区留言，和我一起讨论。