13、Skywalking的埋点数据发送-基于kafka的默认发送机制

rock_fish 关注
2021.06.23 14:02:50 字数 592 阅读 229

TraceSegment **基于Kafka的发送**

基于kafka的数据推送和消费中，有这么一个疑问萦绕：OAPServer的数据聚合是怎么实现的？

- 如果应用A的实例AA的数据都提交在同一个OAPServer实例中，那么在这个实例中就可以进行数据聚合
- 如果应用A的实例AA的数据提交在多个不同一个OAPServer实例中，那么每个实例中聚合之后，还要再把每个实例聚合后的数据再进行一次聚合。

其大致的处理流程是这样的：采集->发送kafka->OAP消费kafka->聚合/存储

这里先看采集和发送的处理，OAP的处理在OAP端流程梳理的章节看

消息发送的分区策略

`KafkaTraceSegmentServiceClient` 中梳理一下2个知识点：
1.看发送的消息是否有key 和分区信息
2.看分区算法

1. 看消息构造

在 `KafkaTraceSegmentServiceClient#consume` 中可以看到构造的 `ProducerRecord` 中 指定了topic和key

```
public void consume(final List<TraceSegment> data) {
        data.forEach(traceSegment -> {
            SegmentObject upstreamSegment = traceSegment.transform();
            ProducerRecord<String, Bytes> record = new ProducerRecord<>(
                topic,//指定了topic
                upstreamSegment.getTraceSegmentId(),//指定了key
                Bytes.wrap(upstreamSegment.toByteArray())
            );
           ...省略部分代码
        });
    }
```

2. 看分区算法

默认情况下未指定Partitioner，则使用Kafka默认的DefaultPartitioner的分区算法
从源码可知，指定了key，则会按照key做hash算出分区，key不变的情况下，所算出的分区是一致的

```
    public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] value
        if (keyBytes == null) {
            return stickyPartitionCache.partition(topic, cluster);
        }
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();
        // hash the keyBytes to choose a partition
        return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;
    }
```


rock_fish 关注
总资产6

21、Skywalking的埋点-Agent动态采样控制
阅读 219

22、skywalking的Trace数据协议
阅读 12

skywalking的日常维护1：com.netflix.zuul.exception.ZuulExcept
阅读 41

推荐阅读

Kafka的设计原理介绍
阅读 354

List 去除重复数据的五种方式
阅读 240

iOS 动态方法解析、消息转发实现原理
阅读 159

RocketMQ保证顺序消费demo
阅读 525

多线程消费顺序-01
阅读 293

根据1，2可知，在使用默认的分区策略的情况下，相同的TraceSegmentId 消息提交到同一个kafka分区中。

4.TraceSegmentId **的生命周期**

TraceSegment的构造中给traceSegmentId进行了赋值。

```
public TraceSegment() {
        this.traceSegmentId = GlobalIdGenerator.generate();
        this.spans = new LinkedList<>();
        this.relatedGlobalTraceId = new NewDistributedTraceId();
        this.createTime = System.currentTimeMillis();
    }
```

一个TraceSegment有唯一的一个TraceSegmentId，接下来就需要梳理清楚是不是一次调用在一个进程内就有一个TraceSegment呢？先给出答案进程创建第一个EntrySpan的时候会先创建一个TraceSegment，其内部承载了一个span集合。

梳理源码的入口是这样的：对于skywalking来说，进程的入口点是创建一个EntrySpan,所以我们从 `ContextManager#createEntrySpan` 源码中跟踪下去就能寻找到答案：

```
    public static AbstractSpan createEntrySpan(String operationName, ContextCarrier carrier) {
        AbstractSpan span;
        AbstractTracerContext context;
        operationName = StringUtil.cut(operationName, OPERATION_NAME_THRESHOLD);
       ...
       context = getOrCreate(operationName, false);//这里有创建Context的机会
       span = context.createEntrySpan(operationName);
        ...
        return span;
    }
```

`context = getOrCreate(operationName, false)` 这个方法中，会有创建TraceContext的逻辑：

```
 context = EXTEND_SERVICE.createTraceContext(operationName, forceSampling);
```

继续跟入代码,可以看到new TracingContext的操作

```
context = new TracingContext(operationName, spanLimitWatcher);
```

TracingContext 中有个成员 `private TraceSegment segment` ,在TracingContext构造方法中可以看到TraceSegment的创建过程

```
TracingContext(String firstOPName, SpanLimitWatcher spanLimitWatcher) {
        this.segment = new TraceSegment();
        ...
}
```

rock_fish
总资产6 共写了9.4W字 获得82个赞 共27个粉丝

关注

写下你的评论... 评论0 赞