《31 讲带你搞懂 SkyWalking》

徐郡明 资深技术专家

— 拉勾教育出品 —

第24讲：jvm-receiver 插件探秘 不仅有 Trace 还可以有监控

JVMMetricReportServiceHandler

在 SkyWalking OAP 提供了 jvm-receiver-plugin 插件用于接收 Agent 发送的 JVMMetric jvm-receiver-plugin 插件的 SPI 配置文件中

指定的 **ModuleDefine 实现**是 **JVMModule**（名称为 receiver-jvm）

ModuleProvider 实现是 **JVMModuleProvider**（名称为 default）

JVMMetricReportServiceHandler

在 JVMModuleProvider 的 start() 方法中会将 JVMMetricReportServiceHandler 注册到 GRPCServer 中

JVMMetricReportServiceHandler 实现了 JVMMetric.proto 文件中

定义的 JVMMetricReportService gRPC 接口

其 collect() 方法负责处理 JVMMetric 对象

JVMMetricReportServiceHandler

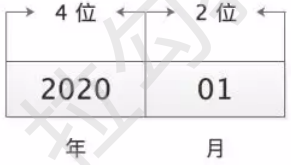
DownSampling.Second

DownSampling.Minute

DownSampling.Hour

DownSampling.Day

DownSampling.Month

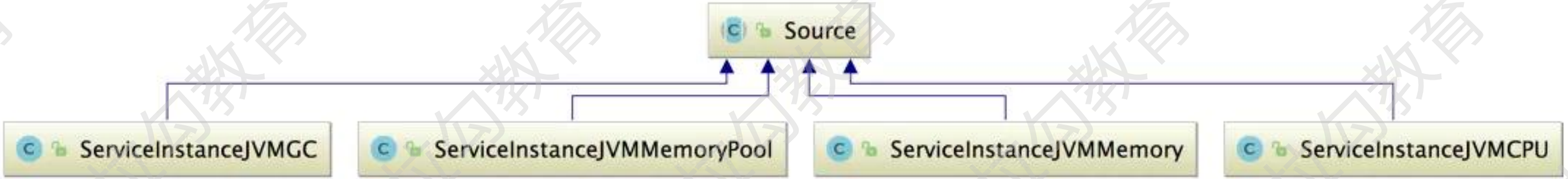


JVMMetricReportServiceHandler

拉勾教育
互联网人实战大学

```
void sendMetric(int serviceInstanceId, long minuteTimeBucket,
    JVMMetric metrics) {
  // 获取 JVMMetric 对应的 ServiceId
  ServiceInstanceInventory serviceInstanceInventory =
      instanceInventoryCache.get(serviceInstanceId);
  int serviceId = serviceInstanceInventory.getServiceId();
  //将 JVMMetric 分类转发
  this.sendToCpuMetricProcess(serviceId, serviceInstanceId,
      minuteTimeBucket, metrics.getCpu());
  this.sendToMemoryMetricProcess(serviceId, serviceInstanceId,
      minuteTimeBucket, metrics.getMemoryList());
  this.sendToMemoryPoolMetricProcess(serviceId, serviceInstanceId,
      minuteTimeBucket, metrics.getMemoryPoolList());
  this.sendToGCMetricProcess(serviceId, serviceInstanceId,
      minuteTimeBucket, metrics.getGcList());
}
```

JVMMetricReportServiceHandler



Dispatcher & DispatcherManager

在 DispatchManager 中维护了一个 Map<Integer, List<SourceDispatcher>> 集合

该集合记录了各个 Source 类型对应的 Dispatcher 实现

其中 Key 是 Source 类型对应的 scope 值，Source 的不同子类对应不同的 scope 值

例如：ServiceInstanceJVMGC 对应的 scope 值为 11，ServiceInstanceJVMCPU 对应的 scope 值为 8

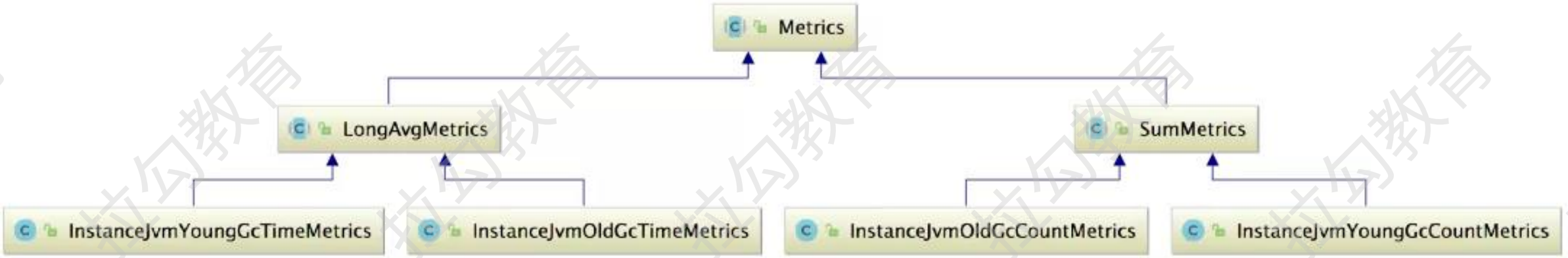
```
public class ServiceInstanceJVMGCDispatcher
    implements SourceDispatcher<ServiceInstanceJVMGC> {...}
```

Dispatcher & DispatcherManager

拉勾教育
互联网人实战大学

```
public void dispatch(ServiceInstanceJVMGC source) {
    doInstanceJvmYoungGcTime(source);
    doInstanceJvmOldGcTime(source);
    doInstanceJvmYoungGcCount(source);
    doInstanceJvmOldGcCount(source);
}
```

Dispatcher & DispatcherManager



Dispatcher & DispatcherManager


```
@Column private long summation; // 总和
@Column private int count; // 次数
@Column private long value; // 平均值
```

Dispatcher & DispatcherManager


```
@Column private long summation; // 总和
@Column private int count; // 次数
@Column private long value; // 平均值
```

```
this.value = this.summation / this.count;
```

Dispatcher & DispatcherManager

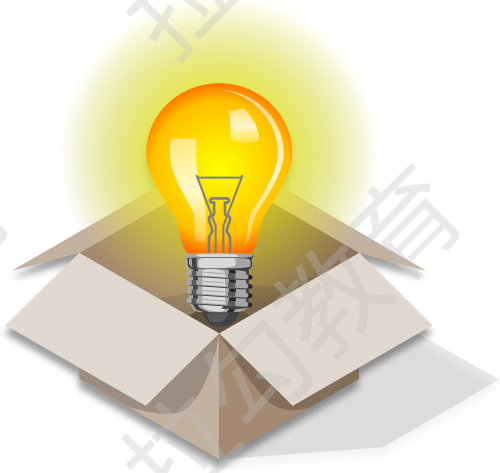

```
this.value = this.summation / this.count;
```

```
@Column(columnName = "entity_id") @IDColumn private String entityId;

@Column(columnName = "service_id")  private int serviceId;
```

Dispatcher & DispatcherManager

SkyWalking OAP 中很多其他类型的监控数据：

- SumMetrics 计算的是时间窗口内的总和
- MaxDoubleMetrics、MaxLongMetrics 计算的是时间窗口内的最大值
- PercentMetrics 计算的是时间窗口内符合条件数据所占的百分比（即 match / total）
- PxxMetrics 计算的是时间窗口内的分位数，例如： P99Metrics、P95Metrics、P70Metrics等
- CPMMetrics 计算的是应用的吞吐量，默认是通过分钟级别的调用次数计算的



Dispatcher & DispatcherManager

对应 ElasticSearch Index 的名称

```
@Stream(name = "instance_jvm_old_gc_time", scopeId = 11,
        builder = InstanceJvmOldGcTimeMetrics.Builder.class, processor = MetricsStreamProcessor.class)
public class InstanceJvmOldGcTimeMetrics extends LongAvgMetrics implements WithMetadata {

    @Setter @Getter @Column(columnName = "entity_id") @IDColumn private java.lang.String entityId;
    @Setter @Getter @Column(columnName = "service_id")  private int serviceId;

    @Override public String id() {
        String splitJointId = String.valueOf(getTimeBucket());
        splitJointId += Const.ID_SPLIT + entityId;
        return splitJointId;
    }
```

对应 Column 的名称

对应 Document Id 是 TimeBucket(时间窗口) + 下划线 + instanceId 三部分构成的

Dispatcher & DispatcherManager

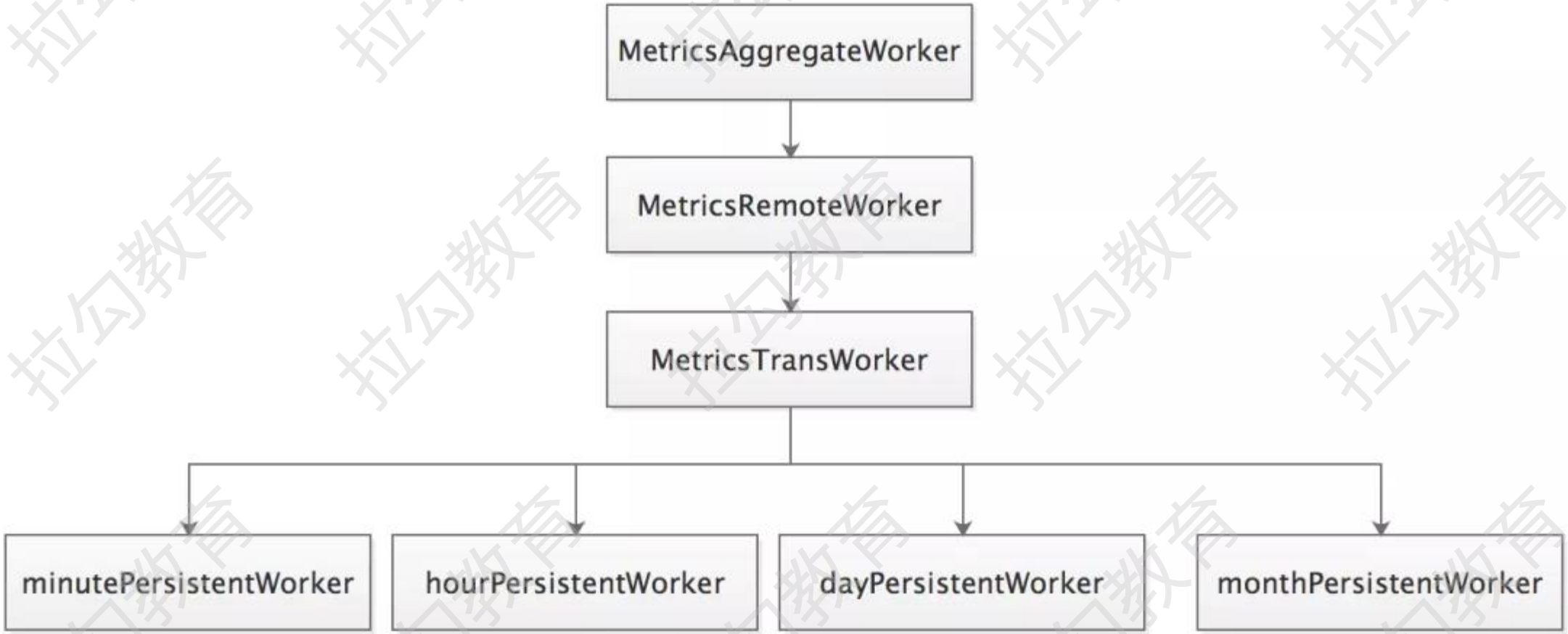
拉勾教育
互联网人实战大学

```
private void doInstanceJvmOldGcTime(ServiceInstanceJVMGC source) {
    // 创建 InstanceJvmOldGcTimeMetrics 对象
    InstanceJvmOldGcTimeMetrics metrics =
        new InstanceJvmOldGcTimeMetrics();
    if (!new EqualMatch().setLeft(source.getPhrase())
            .setRight(GCPhrase.OLD).match()) {
        return;    // 只处理 Old GC
    }
    metrics.setTimeBucket(source.getTimeBucket()); // 分钟级别的时间窗口
    metrics.setEntityId(source.getEntityId()); // serviceInstanceId
    metrics.setServiceId(source.getServiceId()); // serviceId
    metrics.combine(source.getTime(), 1); // 记录 GC 时间，count 为1
    // 交给 MetricsStreamProcessor 继续后续处理
    MetricsStreamProcessor.getInstance().in(metrics);
}
```

MetricsStreamProcessor


```
private Map<Class<? extends Metrics>,
MetricsAggregateWorker> entryWorkers = new HashMap<>();
```

MetricsStreamProcessor



MetricsStreamProcessor

拉勾教育
互联网人实战大学

```
// 创建 minutePersistentWorker
MetricsPersistentWorker minutePersistentWorker =
    minutePersistentWorker(moduleDefineHolder, metricsDAO, model);
// 创建 MetricsTransWorker，后续 worker 指向 minutePersistenceWorker 对象(以及
// hour、day、monthPersistentWorker)
MetricsTransWorker transWorker =
  new MetricsTransWorker(moduleDefineHolder, stream.name(),
          minutePersistentWorker, hourPersistentWorker,
                  dayPersistentWorker, monthPersistentWorker);
// 创建 MetricsRemoteWorker，并将 nextWorker 指向上面的MetricsTransWorker对象
MetricsRemoteWorker remoteWorker = new
 MetricsRemoteWorker(moduleDefineHolder, transWorker, stream.name());
// 创建MetricsAggregateWorker，并将 nextWorker 指向上面的
// MetricsRemoteWorker 对象
MetricsAggregateWorker aggregateWorker =
   new MetricsAggregateWorker(moduleDefineHolder, remoteWorker,
          stream.name());
// 将上述 worker 链与指定 Metrics 类型绑定
entryWorkers.put(metricsClass, aggregateWorker);
```

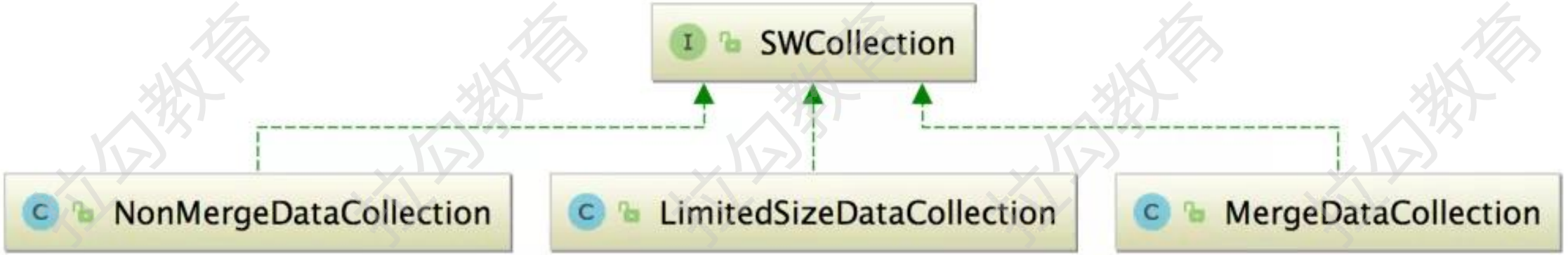
MetricsStreamProcessor


```
this.registerServiceImplementation(DownsamplingConfigService.class,
new DownsamplingConfigService(moduleConfig.getDownsampling()));
```

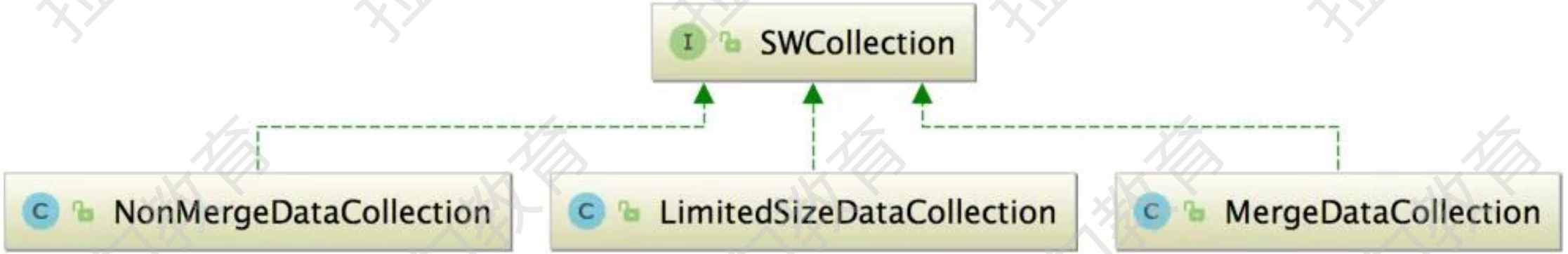
MergeDataCache 缓冲区设计与实现


```
private SWCollection<DATA> pointer; //指向当前正在写入的缓冲队列
private SWCollection<DATA> windowDataA; // A、B两个缓冲队列
private SWCollection<DATA> windowDataB;
```

MergeDataCache 缓冲区设计与实现



MergeDataCache 缓冲区设计与实现



- 当队列的 reading 被设置为 true 时，处于 reading 状态，表示可能有线程在从该队列中读取数据
- 当队列的 writing 被设置为 true 时，处于 writing 状态，表示可能有线程在向该队列中写入数据

MergeDataCache 缓冲区设计与实现


```
private AtomicInteger windowSwitch = new AtomicInteger(0);
```

MergeDataCache 缓冲区设计与实现


```
// 检查 windowSwitch 字段，以及 last 队列是否处于可读状态
public boolean trySwitchPointer() {
   return windowSwitch.incrementAndGet() == 1
         && !getLast().isReading();
   //如果此时 last 队列处于 reading 状态，切换后，last 队列会变成current队列，
   //就会出现两个线程(一个读线程、一个写线程)并发操作该队列的可能，所以需要进行
   // reading 状态的检测
}

//在 trySwitchPointer()方法尝试之后，需要在 finally 代码块中恢复windowSwitch
//字段的值，为下次检查做准备
public void trySwitchPointerFinally() {
   windowSwitch.addAndGet(-1);
}
```

MergeDataCache 缓冲区设计与实现


```
public void switchPointer() {
    if (pointer == windowDataA) { //根据 pointer 当前的指向，进行修改
        pointer = windowDataB;
    } else {
        pointer = windowDataA;
    }
    getLast().reading(); //修改 last 队列的状态
}
```

MergeDataCache 缓冲区设计与实现

MergeDataCache 类的实现，有两个点需要**注意**：

- 它将 A、B 两个队列初始化为 MergeDataCollection 队列
- 它维护了一个 lockedMergeDataCollection 字段

  在开始写入的时候，会先调用 writing() 方法将 lockedMergeDataCollection 字段

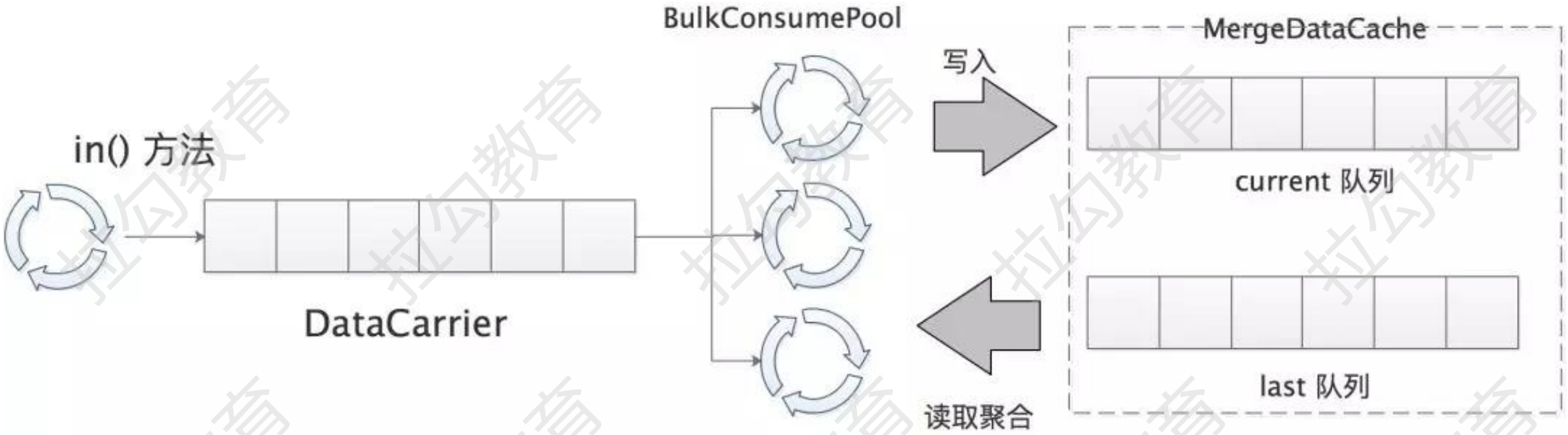
  指向当前的 current 队列，直至写入操作完成

  即使在写入操作过程中发生了 pointer 的切换

  lockedMergeDataCollection 字段的指向也不会发生变化

  在写入操作完成之后，会调用 finishWriting() 方法将 lockedMergeDataCollection 字段设置为 null

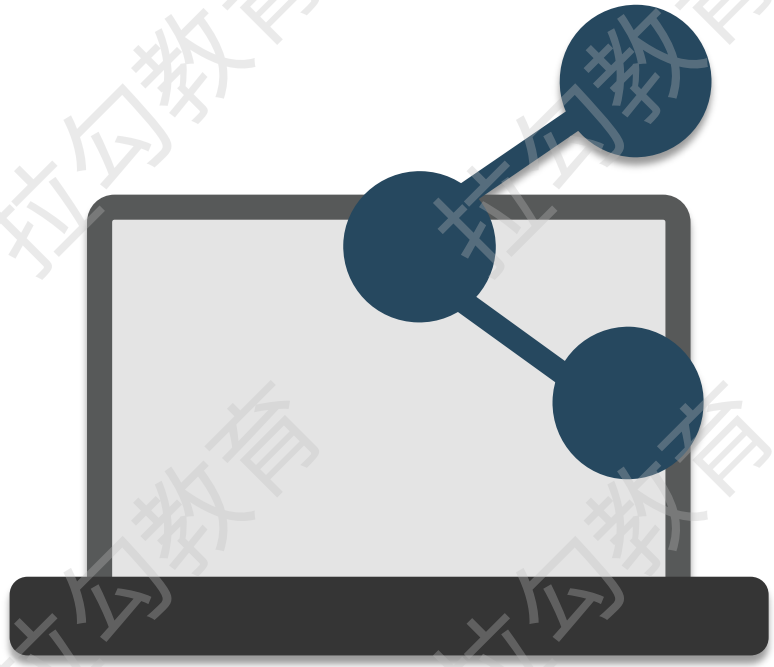
MergeDataCache 缓冲区设计与实现



MergeDataCache 缓冲区设计与实现

下面深入分析两个点：

1. Consumer 线程消费 DataCarrier 并聚合监控数据的相关实现
2. Consumer 线程定期清理 MergeDataCache 缓冲区并发送监控数据的相关实现



MergeDataCache 缓冲区设计与实现


```
private void aggregate(Metrics metrics) {
    //将 lockedMergeDataCollection 指向 current 队列，并设置其 writing标记
    mergeDataCache.writing();
    if (mergeDataCache.containsKey(metrics)) {
        //存在重复的监控数据，则进行合并，
        //不同 Metrics子类的 combine()方法实现有所不同，
        //这里的 InstanceJvmOldGcTimeMetrics 的实现就 summation 的累加、
        // count 加一
        mergeDataCache.get(metrics).combine(metrics);
    } else { //该 Metrics 第一次出现，直接写入到
        mergeDataCache.put(metrics);
    }
    //清理 current 队列的 writing 标记，之后清理 lockedMergeDataCollection
    mergeDataCache.finishWriting();
}
```

MergeDataCache 缓冲区设计与实现


```
private void sendToNext() {
    //首先进行队列切换，之后会设置 last 队列的 reading 状态
    mergeDataCache.switchPointer();
    //此时可能其他的 Consumer 线程还在写入last队列，需要等待写入完成
    while (mergeDataCache.getLast().isWriting()) {
        Thread.sleep(10);
    }
    //开始读取 last 队列中的全部 Metrics 数据并发送到下一个 worker 处理
    mergeDataCache.getLast().collection().forEach(data -> {
        nextWorker.in(data);
    });
    //读取完成后，清空 last 队列以及其 reading 状态
    mergeDataCache.finishReadingLast();
}
```

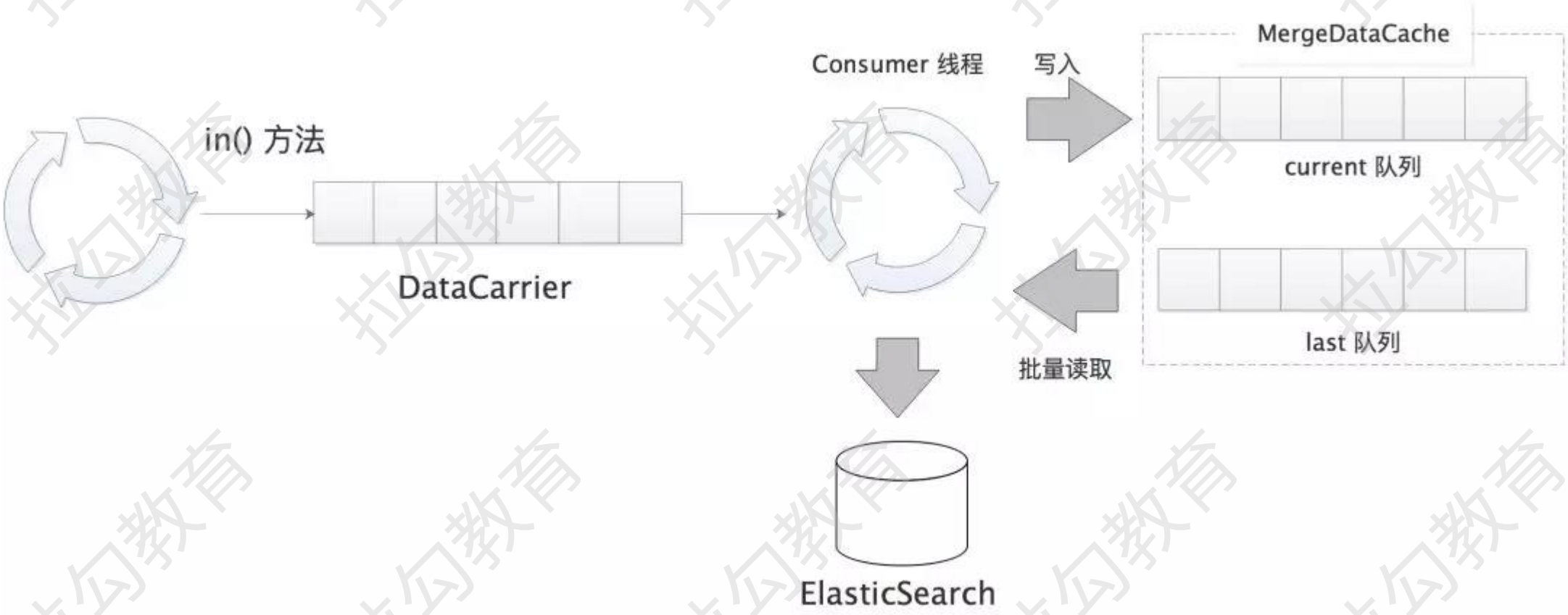
MetricsTransWorker


```
private final MetricsPersistentWorker minutePersistenceWorker;

private final MetricsPersistentWorker hourPersistenceWorker;

private final MetricsPersistentWorker dayPersistenceWorker;

private final MetricsPersistentWorker monthPersistenceWorker;
```

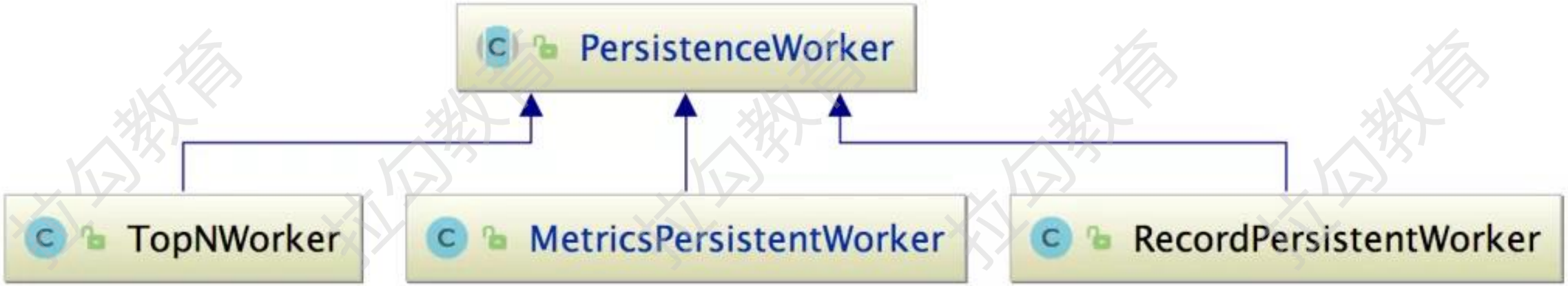
MetricsTransWorker


```
public void in(Metrics metrics) {
    //检测 Hour、Day、Month 对应的 PersistenceWorker 是否为空，若不为空，
    //则将 Metrics 数据拷贝一份并调整时间窗口粒度，交到相应的
    // PersistenceWorker 处理，这里省略了具体逻辑
    //最后，直接转发给 minutePersistenceWorker 进行处理
    if (Objects.nonNull(minutePersistenceWorker)) {
        aggregationMinCounter.inc();
        minutePersistenceWorker.in(metrics);
    }
}
```

MetricsPersistentWorker



MetricsPersistentWorker



MetricsPersistentWorker

拉勾教育
互联网人实战大学

```
void onWork(INPUT input) {
    //检测 current 队列中缓冲的数据量是否打到阈值
    if (getCache().currentCollectionSize() >= batchSize) {
        try {
            //检测是否符合切换缓冲队列的条件，在分析 Windows 抽象类时也说过，
            // trySwitchPointer()会检测 windowSwitch 标记以及 last 队列的 reading状态
            if (getCache().trySwitchPointer()) {
                //切换 current 缓冲队列，同时会设置切换后的 last 队列的 reading标记
                getCache().switchPointer();
                //创建一批请求并批量执行
                List<?> collection = buildBatchCollection();
                batchDAO.batchPersistence(collection);
            }
        } finally { // trySwitchPointerFinally()方法会重制 windowSwitch标记
            getCache().trySwitchPointerFinally();
        }
    }
    cacheData(input); // 写入缓存
}
```

MetricsPersistentWorker

拉勾教育
互联网人实战大学

```
public void cacheData(Metrics input){
    //将 lockedMergeDataCollection 指向 current 队列，并设置其 writing 标记
    mergeDataCache.writing();
    if (mergeDataCache.containsKey(input)){
        //存在重复的监控数据，则进行合并

        Metrics metrics = mergeDataCache.get(input);
        metrics.combine(input);
        //重新计算该监控值，不同 Metrics 实现的计算方式不同，例如，
        // LongAvgMetrics.calculate()方法就是计算平均值
        metrics.calculate();
    } else {
        input.calculate();//第一次计算该监控值
        mergeDataCache.put(input);
    }
    //更新 lockedMergeDataCollection 队列的 writing 状态，然后清空
lockedMergeDataCollection
    mergeDataCache.finishWriting();
}
```

MetricsPersistentWorker


```
while (getCache().getLast().isWriting()) {
    Thread.sleep(10);  //循环检测 last 队列的 writing 状态
}
```

MetricsPersistentWorker

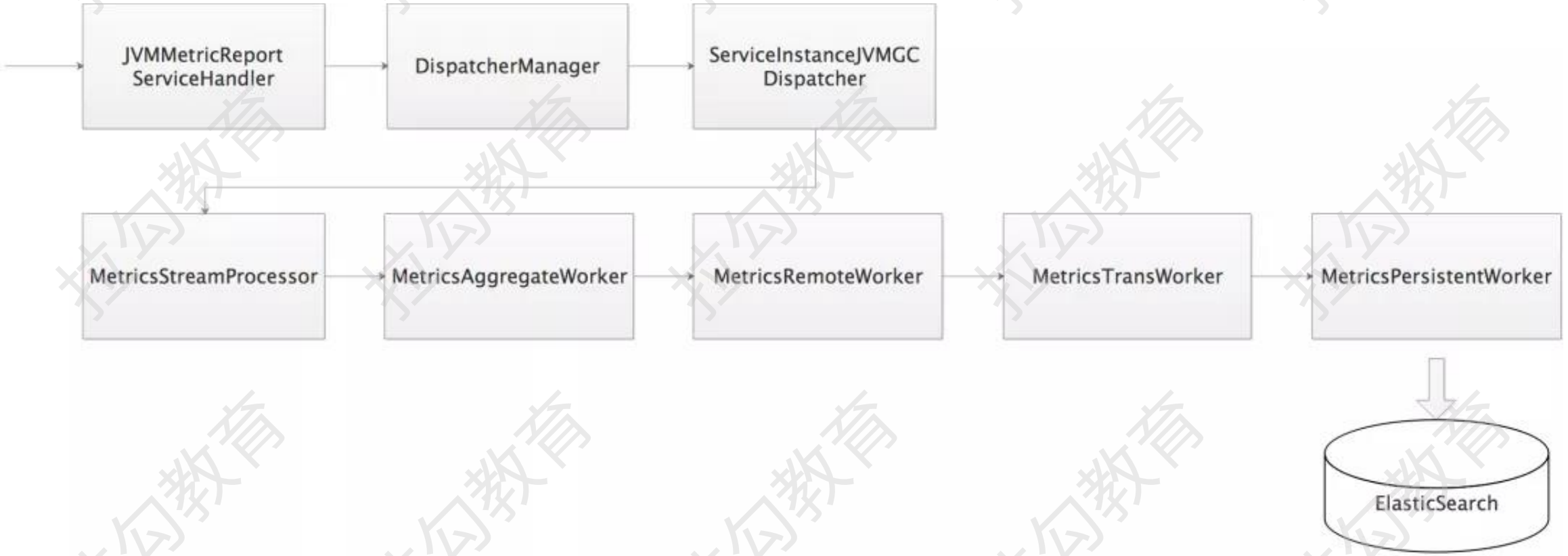

```
//根据id从底层存储中查询 Metrics
Metrics dbData = metricsDAO.get(model, data);
if (nonNull(dbData)) {//已存在相应的 Document
    data.combine(dbData); //已存在则进行合并
    data.calculate(); //重新计算 value 值
    //产生相应的 UpdateRequest 请求，并添加到 batchCollection 集合中
    batchCollection.add(metricsDAO.prepareBatchUpdate(model, data));
} else {
    //产生相应的 IndexRequest 请求，并添加到 batchCollection 集合中
    batchCollection.add(metricsDAO.prepareBatchInsert(model, data));
}
```

MetricsPersistentWorker

拉勾教育
互联网人实战大学

```
public void batchPersistence(List<?> batchCollection) {
    if (bulkProcessor == null) {
        //创建 BulkProcessor，创建方式与前面"ElasticSearch基础入门"小节中展示的示例相同，不再重复
        this.bulkProcessor = getClient().createBulkProcessor(bulkActions, bulkSize, flushInterval,
concurrentRequests);
    }
    batchCollection.forEach(builder -> { //遍历batchCollection，将 Request添加到BulkProcessor中
        if (builder instanceof IndexRequest) {
            this.bulkProcessor.add((IndexRequest)builder);
        }
        if (builder instanceof UpdateRequest) {
            this.bulkProcessor.add((UpdateRequest)builder);
        }
    });
    this.bulkProcessor.flush();//将上面添加的请求发送到 ElasticSearch 集群执行
}
```

总结



Next：第25讲《trace-receiver 插件拆解，Trace 蕴含的宝贵信息（上）》

拉勾教育

—互联网人实战大学—

关注拉勾「教育公众号」
获取更多课程信息

L / A / G / O / U