

## 一口气搞懂「Flink Metrics」监控指标和性能优化， 全靠这 33 张图和 7 千字（建议收藏）

本文作者：在 IT 中穿梭旅行

本文档来自公众号：3 分钟秒懂大数据

微信扫码关注，土哥拉你进 **Flink** 技术  
交流群，获取更多大数据技术



备注：加技术群

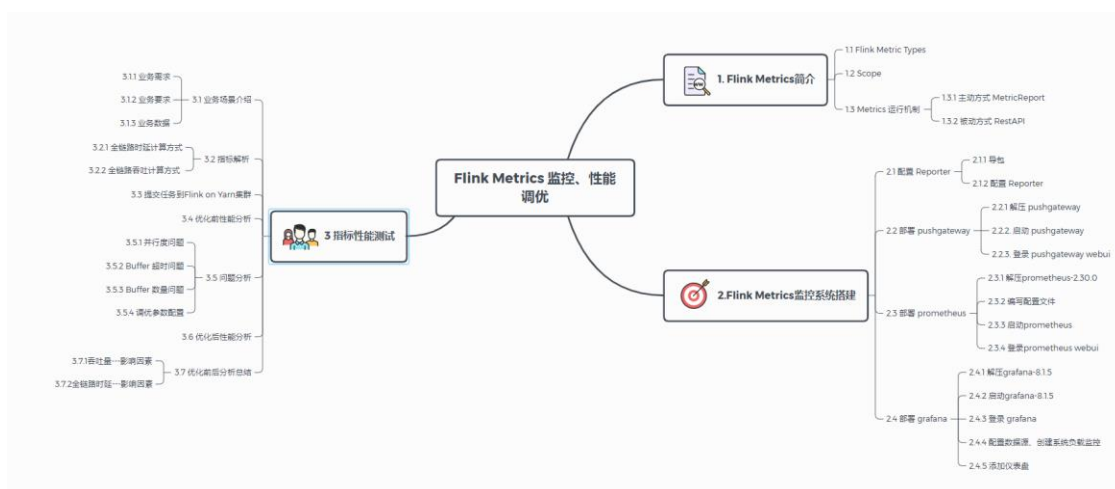


## 前言

大家好，我是土哥。

最近在公司做 Flink 推理任务的性能测试，要对 job 的全链路吞吐、全链路时延、吞吐时延指标进行监控和调优，其中要使用 Flink Metrics 对指标进行监控。

接下来这篇文章，干货满满，我将带领读者全面了解 Flink Metrics 指标监控，并通过实战案例，对全链路吞吐、全链路时延、吞吐时延的指标进行性能优化，彻底掌握 Flink Metrics 性能调优的方法和 Metrics 的使用。大纲目录如下：



## 1 Flink Metrics 简介

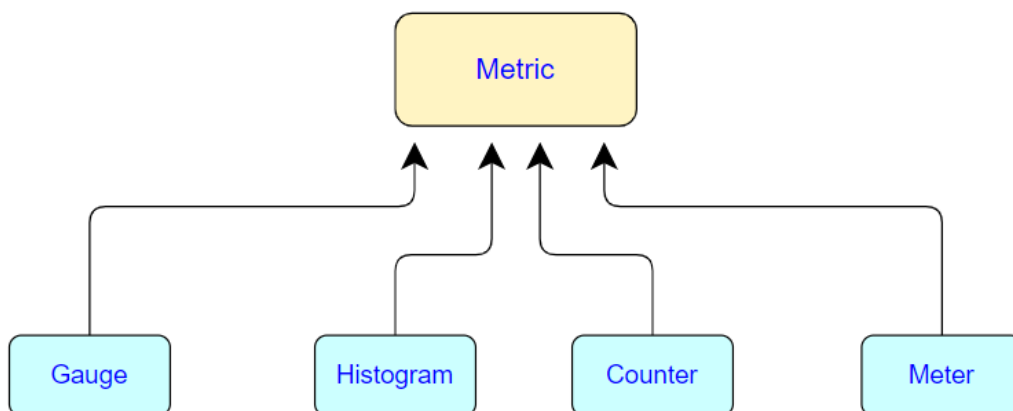
Flink Metrics 是 Flink 集群运行中的各项指标，包含机器系统指标，比如：CPU、内存、线程、JVM、网络、IO、GC 以及任务运行组件（JM、TM、Slot、作业、算子）等相关指标。

Flink Metrics 包含两大作用：

1. 实时采集监控数据。在 Flink 的 UI 界面上，用户可以看到自己提交的任务状态、时延、监控信息等等。
2. 对外提供数据收集接口。用户可以将整个 Flink 集群的监控数据主动上报至第三方监控系统，如：prometheus、grafana 等，下面会介绍。

## 1.1 Flink Metric Types

Flink 一共提供了四种监控指标：分别为 Counter、Gauge、Histogram、Meter。



### 1. Count 计数器

统计一个指标的总量。写过 MapReduce 的开发人员就应该很熟悉 Counter，其实含义都是一样的，就是对一个计数器进行累加，即对于多条数据和多兆数据一直往上加的过程。其中 Flink 算子的接收记录总数 (numRecordsIn) 和发送记录总数 (numRecordsOut) 属于 Counter 类型。

使用方式：可以通过调用 `counter(String name)` 来创建和注册 MetricGroup

```
public class MyMapper extends RichMapFunction<String, String> {
    private transient Counter counter;

    @Override
    public void open(Configuration config) {
        this.counter = getRuntimeContext()
            .getMetricGroup()
            .counter("myCustomCounter", new CustomCounter());
    }

    @Override
    public String map(String value) throws Exception {
        this.counter.inc();
        return value;
    }
}
```

## 2. Gauge 指标瞬时值

Gauge 是最简单的 Metrics，它反映一个指标的瞬时值。比如要看现在 TaskManager 的 JVM heap 内存用了多少，就可以每次实时的暴露一个 Gauge，Gauge 当前的值就是 heap 使用的量。

使用前首先创建一个实现 `org.apache.flink.metrics.Gauge` 接口的类。返回值的类型没有限制。您可以通过在 `MetricGroup` 上调用 `gauge`

```
public class MyMapper extends RichMapFunction<String, String> {
    private transient int valueToExpose = 0;

    @Override
    public void open(Configuration config) {
        getRuntimeContext()
            .getMetricGroup()
            .gauge("MyGauge", new Gauge<Integer>() {
                @Override
                public Integer getValue() {
                    return valueToExpose;
                }
            });
    }

    @Override
    public String map(String value) throws Exception {
        valueToExpose++;
        return value;
    }
}
```

### 3. Meter 平均值

用来记录一个指标在某个时间段内的平均值。Flink 中的指标有 Task 算子中的 `numRecordsInPerSecond`, 记录此 Task 或者算子每秒接收的记录数。

使用方式：通过 `markEvent()` 方法注册事件的发生。通过 `markEvent(long n)` 方法注册同时发生的多个事件。

```
public class MyMapper extends RichMapFunction<Long, Long> {
    private transient Meter meter;

    @Override
    public void open(Configuration config) {
        this.meter = getRuntimeContext()
            .getMetricGroup()
            .meter("myMeter", new MyMeter());
    }

    @Override
    public Long map(Long value) throws Exception {
        this.meter.markEvent();
        return value;
    }
}
```

#### 4. Histogram 直方图

Histogram 用于统计一些数据的分布，比如说 Quantile、Mean、StdDev、Max、Min 等，其中最重要一个是统计算子的延迟。此项指标会记录数据处理的延迟信息，对任务监控起到很重要的作用。

使用方式：通过调用 `histogram(String name, Histogram histogram)` 来注册一个 MetricGroup

```

public class MyMapper extends RichMapFunction<Long, Long> {
    private transient Histogram histogram;

    @Override
    public void open(Configuration config) {
        this.histogram = getRuntimeContext()
            .getMetricGroup()
            .histogram("myHistogram", new MyHistogram());
    }

    @Override
    public Long map(Long value) throws Exception {
        this.histogram.update(value);
        return value;
    }
}

```

## 1.2 Scope

Flink 的指标体系按树形结构划分，域相当于树上的顶点分支，表示指标大的分类。每个指标都会分配一个标识符，该标识符将基于 3 个组件进行汇报：

1. 注册指标时用户提供的名称；
2. 可选的用户自定义域；
3. 系统提供的域。

例如，如果 A.B 是系统域，C.D 是用户域，E 是名称，那么指标的标识符将是 A.B.C.D.E。你可以通过设置 `conf/flink-conf.yaml` 里面的 `metrics.scope.delimiter` 参数来配置标识符的分隔符(默认“.”)

举例说明：以算子的指标组结构为例，其默认为：

`<host>.taskmanager.<tm_id>.<job_name>.<operator_name>.<subtask_index>`

算子的输入记录数指标为：

`hlinkui.taskmanager.1234.wordcount.flatmap.0.numRecordsIn`

## 1.3 Metrics 运行机制

在生产环境下，为保证对 Flink 集群和作业的运行状态进行监控，Flink 提供两种集成方式：

### 1.3.1 主动方式 MetricReport

Flink Metrics 通过在 `conf/flink-conf.yaml` 中配置一个或者一些 reporters，将指标暴露给一个外部系统。这些 reporters 将在每个 job 和 task manager 启动时被实例化。

### 1.3.2 被动方式 RestAPI

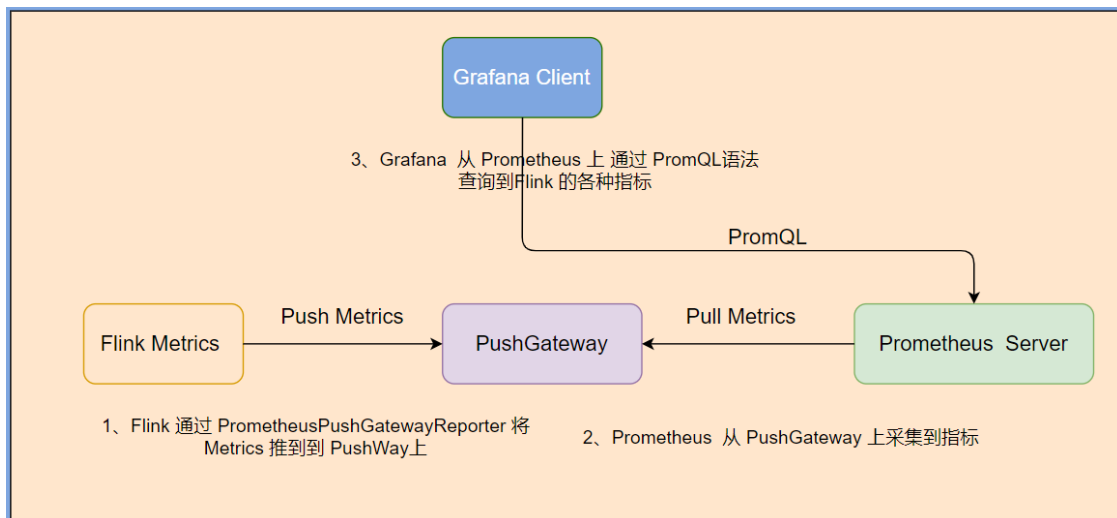
通过提供 Rest 接口，被动接收外部系统调用，可以返回集群、组件、作业、Task、算子的状态。Rest API 实现类是 `WebMonitorEndpoint`

## 2 Flink Metrics 监控系统搭建

Flink 主动方式共提供了 8 种 Report。

我们使用 `PrometheusPushGatewayReporter` 方式通过 `prometheus + pushgateway + grafana` 组件搭建 Flink On Yarn 可视化监控。

当用户使用 Flink 通过 session 模式向 yarn 集群提交一个 job 后，Flink 会通过 `PrometheusPushGatewayReporter` 将 metrics push 到 `pushgateway` 的 9091 端口上，然后使用外部系统 `prometheus` 从 `pushgateway` 进行 pull 操作，将指标采集过来，通过 `Grafana` 可视化工具展示出来。原理图如下：








首先，我们先在 Flink On Yarn 集群中提交一个 Job 任务，让其运行起来，然后执行下面的操作。



## 2.1 配置 Reporter

下面所有工具、jar 包已经全部下载好，需要的朋友在公众号后台回复：**02**，可以全部获取到。

 flink-metrics-prometheus_2.11-1.13.2.jar	2021/9/28 13:47
 grafana-enterprise-8.1.5.linux-amd64.tar.gz	2021/9/28 11:25
 grafana-enterprise-8.1.5-1.x86_64.rpm	2021/9/28 11:38
 prometheus-2.30.0.linux-amd64.tar.gz	2021/9/28 11:18
 pushgateway-1.4.1.linux-amd64.tar.gz	2021/9/28 10:50

### 2.1.1 导包

将 flink-metrics-prometheus\_2.11-1.13.2.jar 包导入 flink-1.13.2/bin 目录下

### 2.1.2 配置 Reporter

选取 PrometheusPushGatewayReporter 方式，通过在官网查询 Flink 1.13.2 Metrics 的配置后，在 flink-conf.yaml 设置，配置如下：

```
metrics.reporter.promgateway.class: org.apache.flink.metrics.prometheus.  
PrometheusPushGatewayReporter
```

```
metrics.reporter.promgateway.host: 192.168.244.129  
metrics.reporter.promgateway.port: 9091  
metrics.reporter.promgateway.jobName: myJob  
metrics.reporter.promgateway.randomJobNameSuffix: true  
metrics.reporter.promgateway.deleteOnShutdown: false  
metrics.reporter.promgateway.groupingKey: k1=v1;k2=v2  
metrics.reporter.promgateway.interval: 60 SECONDS
```

## 2.2 部署 pushgateway

Pushgateway 是一个独立的服务，Pushgateway 位于应用程序发送指标和 Prometheus 服务器之间。

Pushgateway 接收指标，然后将其作为目标被 Prometheus 服务器拉取。可以将其看作代理服务，或者与 blackbox exporter 的行为相反，它接收度量，而不是探测它们。

### 2.2.1 解压 pushgateway

```
258295 9月 28 11:38 grafana-enterprise-8.1.5-1.x86_64.rpm
132 9月 14 18:28 prometheus-2.30.0
54 5月 28 22:37 pushgateway-1.4.1
$
```

### 2.2.2. 启动 pushgateway

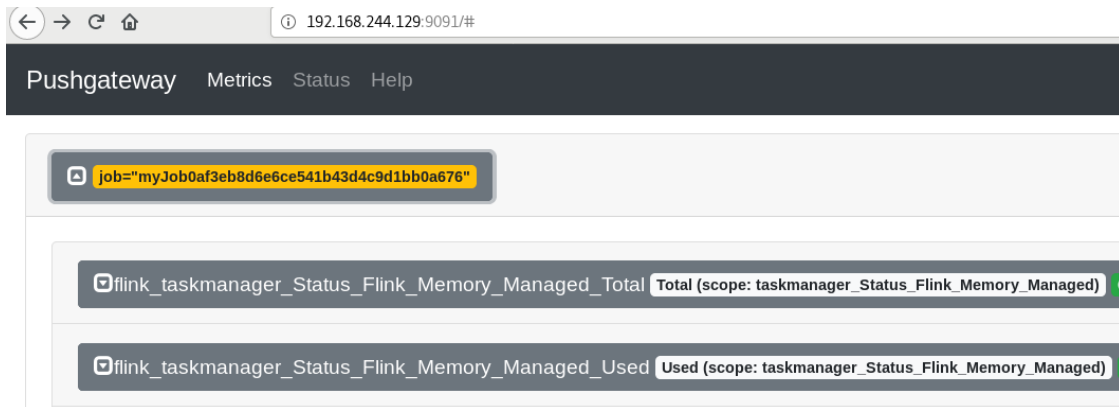
进入到 pushgateway-1.4.1 目录下

```
./pushgateway &
```

查看是否在后台启动成功

```
ps aux|grep pushgateway
```

### 2.2.3. 登录 pushgateway webui



## 2.3 部署 prometheus

Prometheus（普罗米修斯）是一个最初在 SoundCloud 上构建的监控系统。自 2012 年成为社区开源项目，拥有非常活跃的开发人员和用户社区。为强调开源及独立维护，Prometheus 于 2016 年加入云原生云计算基金会（CNCF），成为继 Kubernetes 之后的第二个托管项目。

### 2.3.1 解压 prometheus-2.30.0

```
9月 28 11:38 grafana-enterprise-8.1.5-1.x86_64.rpm
9月 14 18:28 prometheus-2.30.0
5月 28 22:37 pushgateway-1.4.1
```

### 2.3.2 编写配置文件

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['192.168.244.129:9090']
        labels:
          instance: 'prometheus'
  - job_name: 'linux'
    static_configs:
      - targets: ['192.168.244.129:9100']
        labels:
          instance: 'localhost'
  - job_name: 'pushgateway'
    static_configs:
      - targets: ['192.168.244.129:9091']
        labels:
          instance: 'pushgateway'
```

### 2.3.3 启动 prometheus

./prometheus --config.file=prometheus.yml &

启动完后，可以通过 ps 查看一下端口

ps aux|grep prometheus

### 2.3.4 登录 prometheus webui

Targets

All Unhealthy Collapse All

**prometheus (1/1 up)** [show less](#)

Endpoint	State	Labels
<a href="http://192.168.244.129:9090/metrics">http://192.168.244.129:9090/metrics</a>	UP	instance="prometheus" job="prometheus"

**pushgateway (1/1 up)** [show less](#)

Endpoint	State	Labels
<a href="http://192.168.244.129:9091/metrics">http://192.168.244.129:9091/metrics</a>	UP	instance="pushgateway" job="pushgateway"

## 2.4 部署 grafana

Grafana 是一个跨平台的开源的度量分析和可视化工具，可以通过将采集的数据查询然后可视化的展示，并及时通知。它主要有以下六大特点：

- 1、展示方式：快速灵活的客户端图表，面板插件有许多不同方式的可视化指标和日志，官方库中具有丰富的仪表盘插件，比如热图、折线图、图表等多种展示方式；
- 2、数据源：Graphite, InfluxDB, OpenTSDB, Prometheus, Elasticsearch, CloudWatch 和 KairosDB 等；
- 3、通知提醒：以可视方式定义最重要指标的警报规则，Grafana 将不断计算并发送通知，在数据达到阈值时通过 Slack、PagerDuty 等获得通知；
- 4、混合展示：在同一图表中混合使用不同的数据源，可以基于每个查询指定数据源，甚至自定义数据源；
- 5、注释：使用来自不同数据源的丰富事件注释图表，将鼠标悬停在事件上会显示完整的事件元数据和标记；
- 6、过滤器：Ad-hoc 过滤器允许动态创建新的键/值过滤器，这些过滤器会自动应用于使用该数据源的所有查询。

### 2.4.1 解压 grafana-8.1.5

```
liyaozhou      145 9月  28 16:28 grafana-8.1.5
liyaozhou 60258295 9月  28 11:38 grafana-enterprise-8.1.5-1.x86_64.rpm
liyaozhou 60470179 9月  28 11:25 grafana-enterprise-8.1.5.linux-amd64.tar.gz
liyaozhou      144 9月  28 15:03 prometheus-2.30.0
liyaozhou      54 5月  28 22:37 pushgateway-1.4.1
kMetricsSoft]$ rm -rf grafana-enterprise-8.1.5.linux-amd64.tar.gz
kMetricsSoft]$ ll

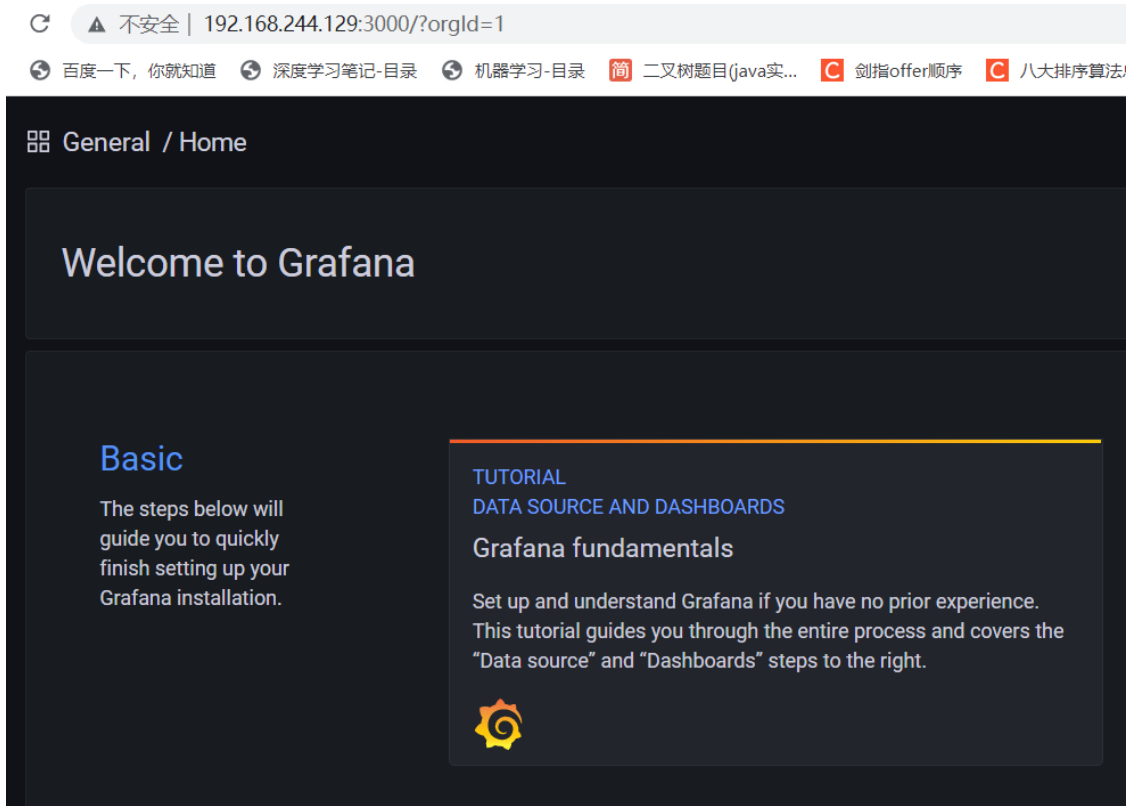
liyaozhou      145 9月  28 16:28 grafana-8.1.5
liyaozhou 60258295 9月  28 11:38 grafana-enterprise-8.1.5-1.x86_64.rpm
liyaozhou      144 9月  28 15:03 prometheus-2.30.0
liyaozhou      54 5月  28 22:37 pushgateway-1.4.1
```

### 2.4.2 启动 grafana-8.1.5

```
./bin/grafana-server web &
```

### 2.4.3 登录 grafana

登录用户名和密码都是 admin

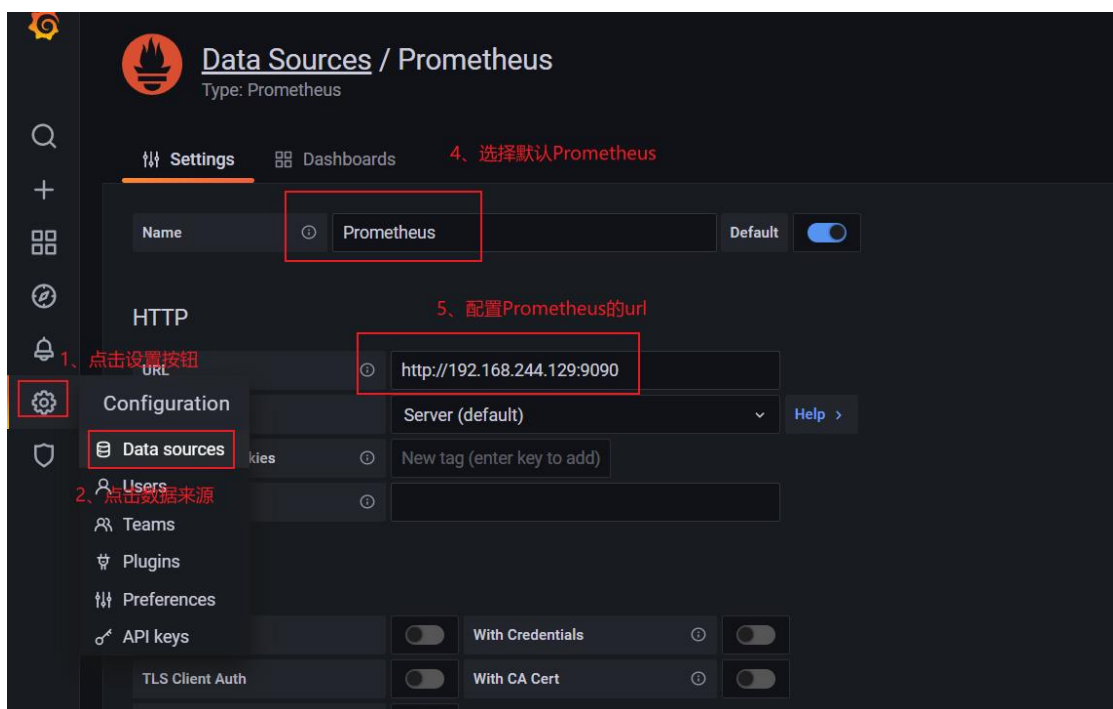


**grafana 配置中文教程:**

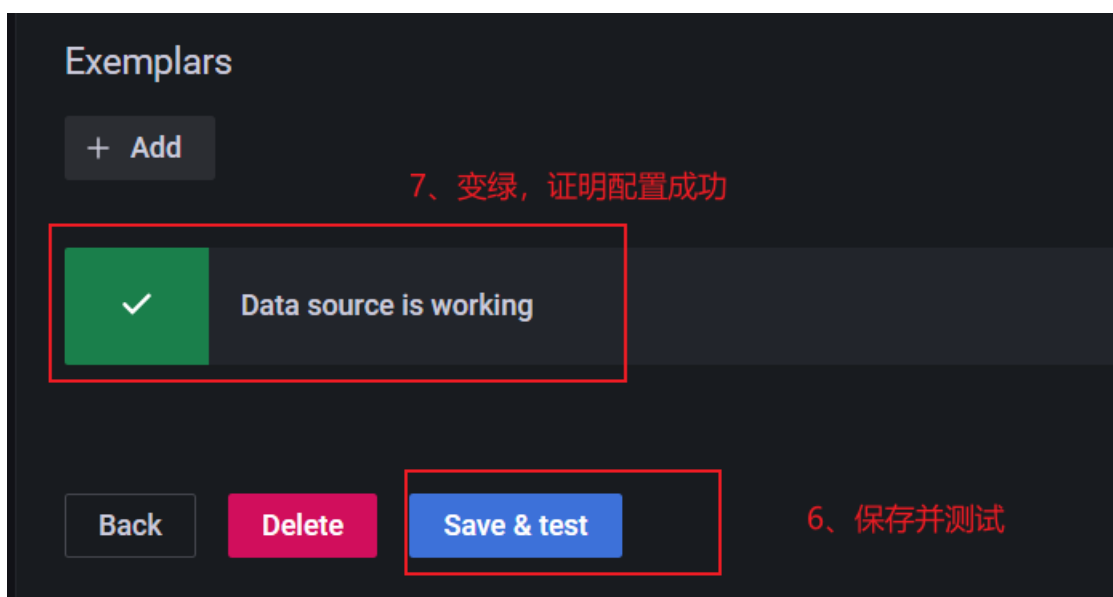
<https://grafana.com/docs/grafana/latest/datasources/prometheus/>

#### 2.4.4 配置数据源、创建系统负载监控

要访问 Prometheus 设置，请将鼠标悬停在配置（齿轮）图标上，然后单击数据源，然后单击 Prometheus 数据源，根据下图进行操作。

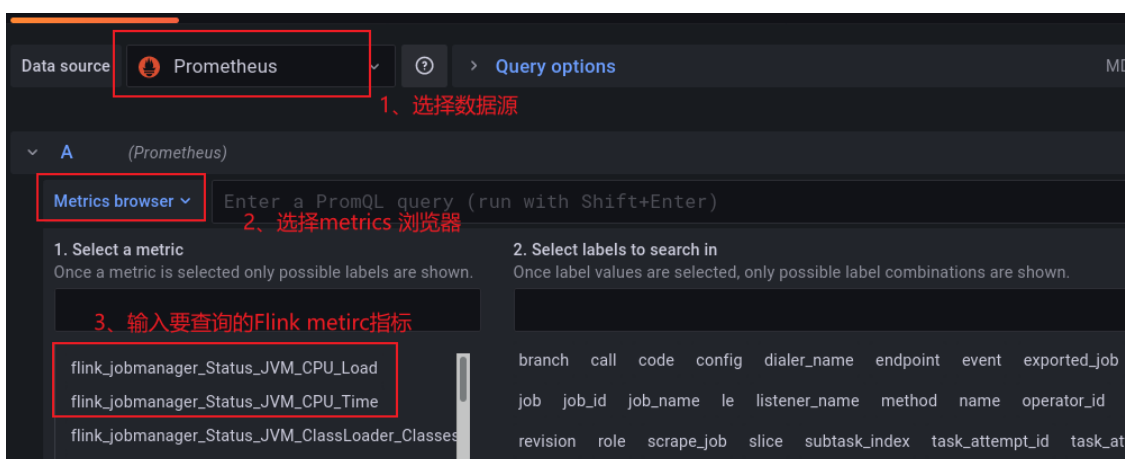
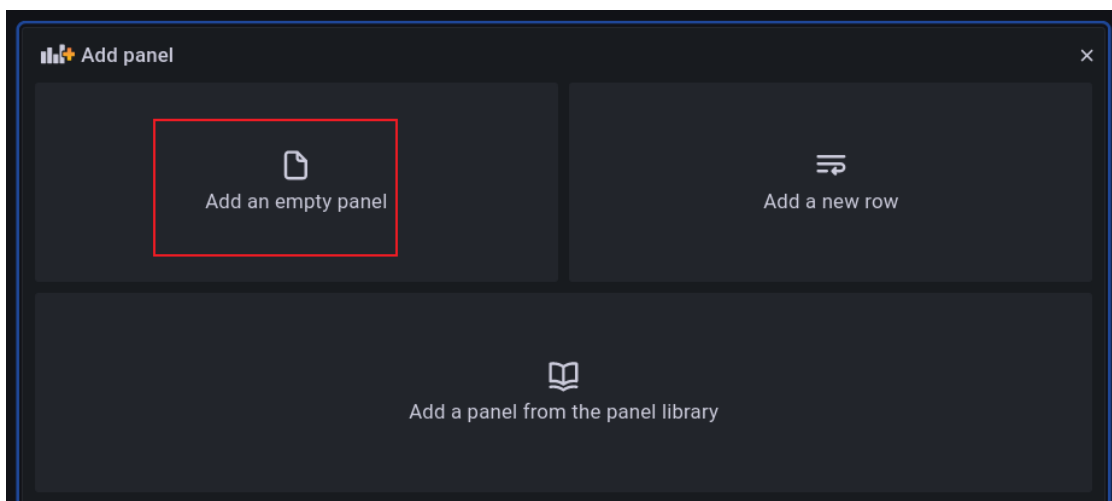


操作完成后，点击进行验证。



## 2.4.5 添加仪表盘

点击最左侧 + 号，选择 DashBoard,选择新建一个 pannel



至此，Flink 的 metrics 的指标展示在 Grafana 中了

flink 指标对应的指标名比较长，可以在 Legend 中配置显示内容，在`{{key}}`将 key 换成对应需要展示的字段即可，如：`{{job_name}},{{operator_name}}`

### 3 指标性能测试

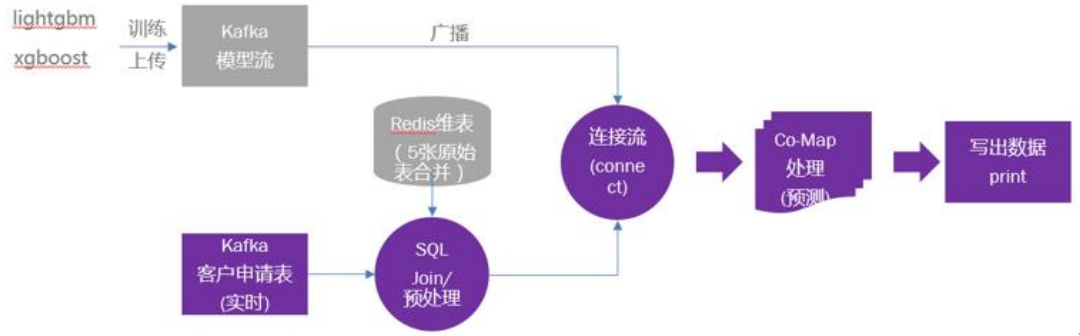
上述监控系统搭建好了之后，我们可以进行性能指标监控了。现在以一个实战案例进行介绍：

#### 3.1 业务场景介绍

金融风控场景

##### 3.1.1 业务需求:

Flink Source 从 data kafka topic 中读取推理数据，通过 sql 预处理成模型推理要求的数据格式，在进行 keyBy 分组后流入下游 connect 算子，与模型 connect 后进入 Co-FlatMap 算子再进行推理，原理图如下：



### 3.1.2 业务要求：

根据模型的复杂程度，要求推理时延到达 20ms 以内，全链路耗时 50ms 以内，吞吐量达到每秒 1.2w 条以上。

### 3.1.3 业务数据：

推理数据：3000w，推理字段 495 个，机器学习 Xgboost 模型字段：495，

## 3.2 指标解析

由于性能测试要求全链路耗时 50ms 以内，应该使用 Flink Metrics 的 Latency Marker 进行计算

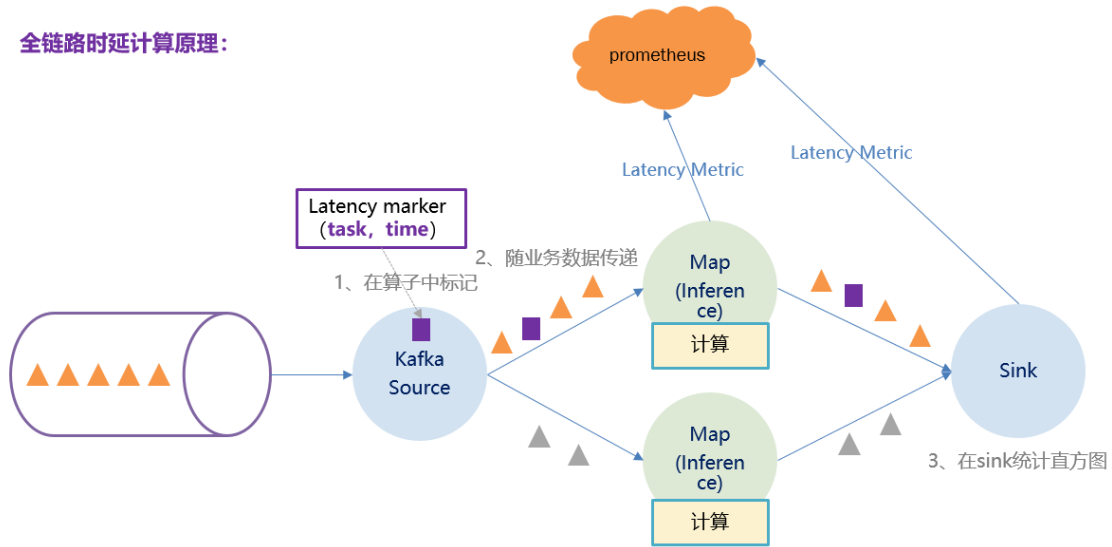
### 3.2.1 全链路时延计算方式：

全链路时延指的是一条推理数据进入 source 算子到数据预处理算子直到最后一个算子输出结果的耗时，即处理一条数据需要多长时间，包含算子内处理逻辑时间，算子间数据传递时间，缓冲区内等待时间。

全链路时延要使用 latency metric 计算。latency metric 是由 source 算子根据当前本地时间生成的一个 marker，并不参与各个算子的逻辑计算，仅仅跟着数据往下游算子流动，每到达一个算子则算出当前本地时间戳并与 source 生成的时间戳相减，得到 source 算子到当前算子的耗时，当到达 sink 算子或者说最后一个算子时，算出当前本地时间戳与 source 算子生成的时间戳相减，即得到全链路时延。原理图如下：



#### 全链路时延计算原理：



由于使用到 Latency marker,所有需要在 flink-conf.yaml 配置参数

`latency.metrics.interval`

系统配置截图如下：

```
latency.metrics.interval: 60
metrics.reporter.promgateway.class: org.apache.flink.metrics.prometheus.PrometheusPushGatewayReporter
metrics.reporter.promgateway.host: 192.168.244.129
metrics.reporter.promgateway.port: 9091
metrics.reporter.promgateway.jobName: myJob
metrics.reporter.promgateway.randomJobNameSuffix: true
metrics.reporter.promgateway.deleteOnShutdown: true
```

#### 3.2.2 全链路吞吐计算方式：

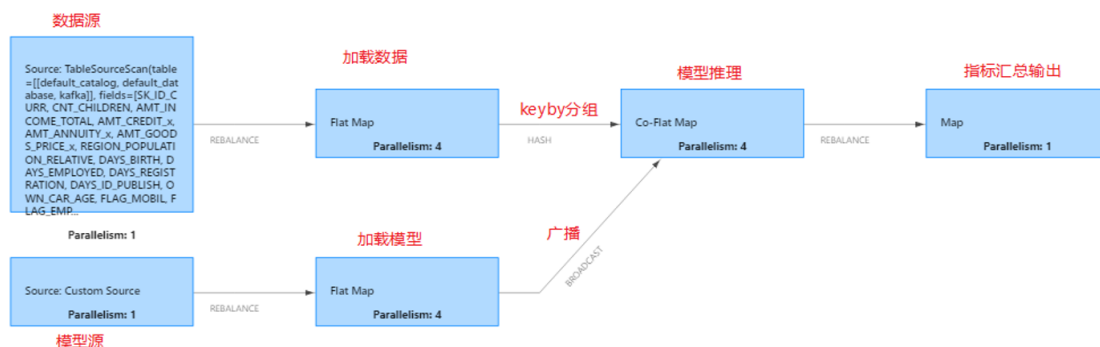
全链路吞吐 = 单位时间处理数据数量 / 单位时间

### 3.3 提交任务到 Flink on Yarn 集群

#### 3.3.1 直接提交 Job

```
# -m jobmanager 的地址
# -yjm 1024 指定 jobmanager 的内存信息
# -ytm 1024 指定 taskmanager 的内存信息
bin/flink run \
-t yarn-per-job -yjm 4096 -ytm 8800 -s 96 \
--detached -c com.threeknowbigdata.datastream.XgboostModelPrediction \
examples/batch/WordCount.jar \
```

提交完成后，我们通过 Flink WEBUI 可以看到 job 运行的任务结果如下：



因为推理模型只是一个 `model`，存在状态中，所以全链路吞吐考虑的是每秒有多少条推理数据进入 `source` 算子到倒数第二个算子（最后一个算子只是指标汇总）流出，这个条数就是全链路吞吐。

可以看到在处理 2000W 条数据时，代码直接统计输出的数值和 `flink webUI` 的统计数值基本一致，所以统计数值是可信的

```

模型加载耗时: 1654
模型加载耗时: 1685
模型加载耗时: 1689
模型加载耗时: 1690
开始推理时间: 1622181424046    第一条时延: 86
500W耗时: 581257    500W条推理耗时: 1728966
1000W耗时: 1125478    1000W条推理耗时: 3393489
2000W耗时: 2172175    2000W条推理耗时: 6869078
    
```

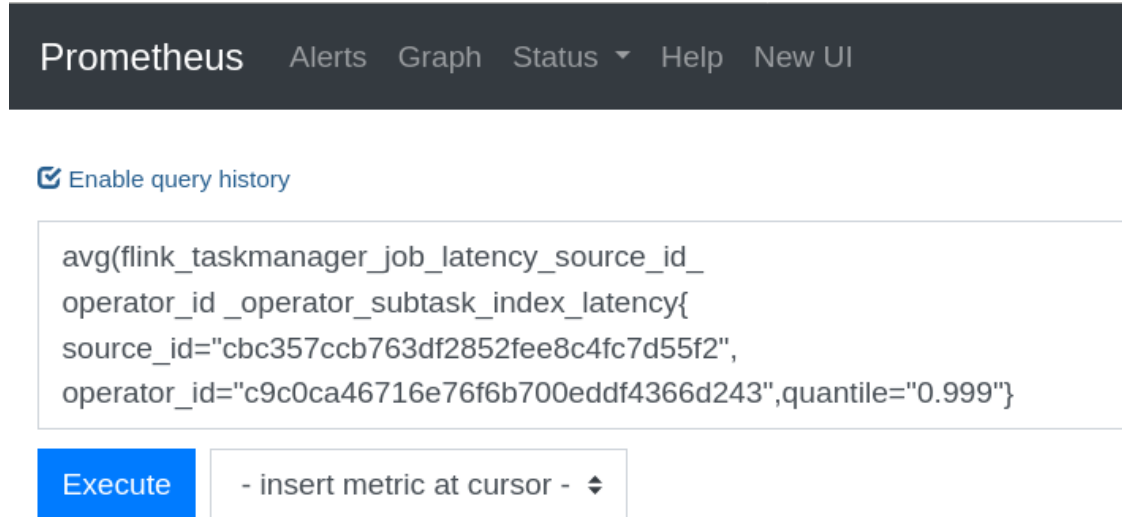
Flink WEBUI 跑的结果数据

Bytes Received	Records Received	Bytes Sent	Records Sent	Parallelism	Start Time	Duration	Tasks
0 B	0	5.82 GB	20,018,331	8	2021-05-28 13:56:59	36m 20s	8
0 B	0	1.00 MB	1	1	2021-05-28 13:56:59	36m 20s	1
1.14 MB	1	752 B	1	8	2021-05-28 13:56:59	36m 20s	8
5.82 GB	20,008,794	334 MB	20,008,783	8	2021-05-28 13:56:59	36m 20s	8
335 MB	20,008,488	0 B	20,008,488	1	2021-05-28 13:56:59	36m 20s	1

打开 Prometheus 在对话框输入全链路时延计算公式

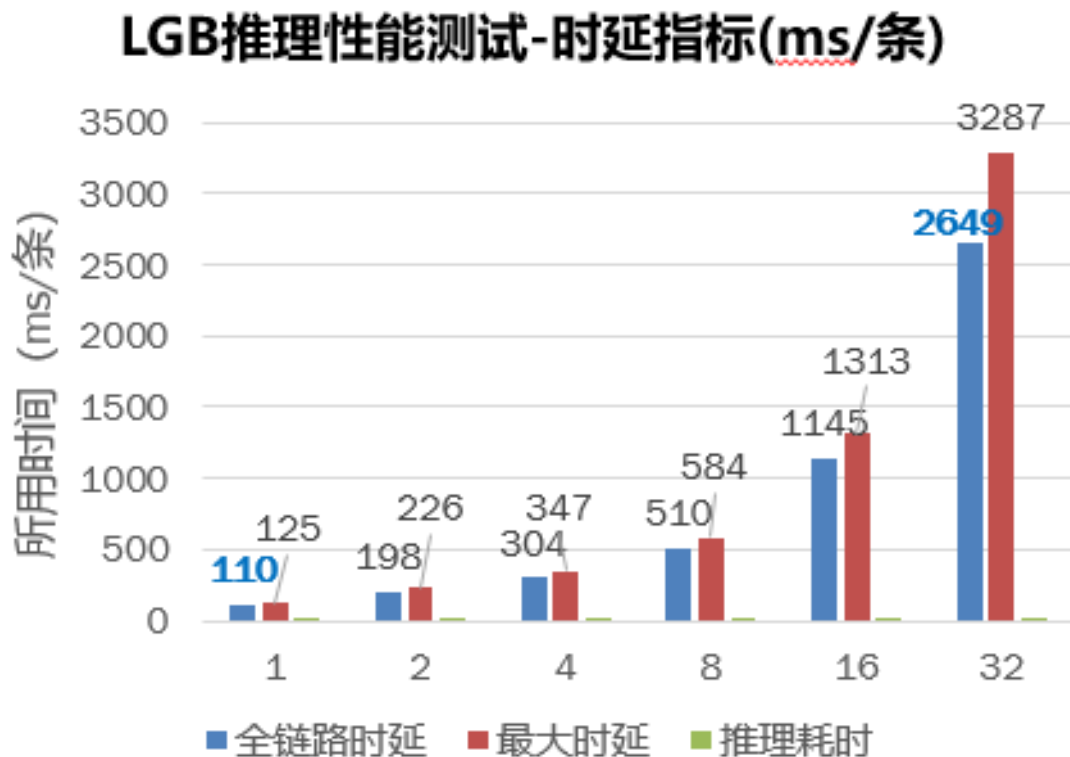
计算公式：

```
avg(flink_taskmanager_job_latency_source_id_
operator_id_operator_subtask_index_latency{
source_id="cbc357ccb763df2852fee8c4fc7d55f2",
operator_id="c9c0ca46716e76f6b700eddf4366d243",quantile="0.999"})
```



### 3.4 优化前性能分析

在将任务提交到集群后，经过全链路时延计算公式、吞吐时延计算公式，最后得到优化前的结果 时延指标统计图如下：



吞吐指标统计图如下：

通过本次测试完后，从图中可以发现：

**时延指标：**加并行度，吞吐量也跟随高，但是全链路时延大幅增长（1 并行至 32 并行，时延从 110ms 增加至 3287ms）

这远远没有达到要求的结果。

### 3.5 问题分析

通过 Prometheus 分析后，结果如下

Detail	SubTasks	TaskManagers	Watermarks	Accumulators	BackPressure	Metrics
			Measurement: 24s ago		Back Pressure Status: <span>HIGH</span>	
SubTask			Ratio	Status		
0			0.98	<span>HIGH</span>		
1			0.98	<span>HIGH</span>		
2			0.99	<span>HIGH</span>		
3			0.97	<span>HIGH</span>		
4			0.96	<span>HIGH</span>		

### 3.5.1 并行度问题：

1. 反压现象：在 Flink WEB-UI 上，可以看到应用存在着非常严重的反压，这说明链路中存在较为耗时的算子，阻塞了整个链路；
2. 数据处理慢于拉取数据：数据源消费数据的速度，大于下游数据处理速度；
3. 增加计算并行度：所以在接下来的测试中会调大推理算子并行度，相当于提高下游数据处理能力；

### 3.5.2 Buffer 超时问题：

1. Flink 虽是纯流式框架，但默认开启了缓存机制（上游累积部分数据再发送到下游）；
2. 缓存机制可以提高应用的吞吐量，但是也增大了时延；
3. 推理场景：为获取最好的时延指标，第二轮测试超时时间置 0，记录吞吐量；

### 3.5.3 Buffer 数量问题：

同上，Flink 中的 Buffer 数量是可以配置的；

1. Buffer 数量越多，能缓存的数据也就越多；
2. 推理场景：为获取最好的时延指标，第二轮测试：减小 Flink 的 Buffer 数量来优化时延指标；

### 3.5.4 调优参数配置

1. SOURCE 与 COFLATMAP 的并行度按照 1:12 配置；
2. Buffer 超时时间配置为 0ms (默认 100ms)；

//在代码中设置

```
senv.setBufferTimeout(0);
```

1. Buffer 数量的配置如下：

修改 flink-conf.yaml

```
memory.buffers-per-channel: 2
memory.float-buffers-per-gate: 2
memory.max-buffers-per-channel: 2
```

配置截图如下：

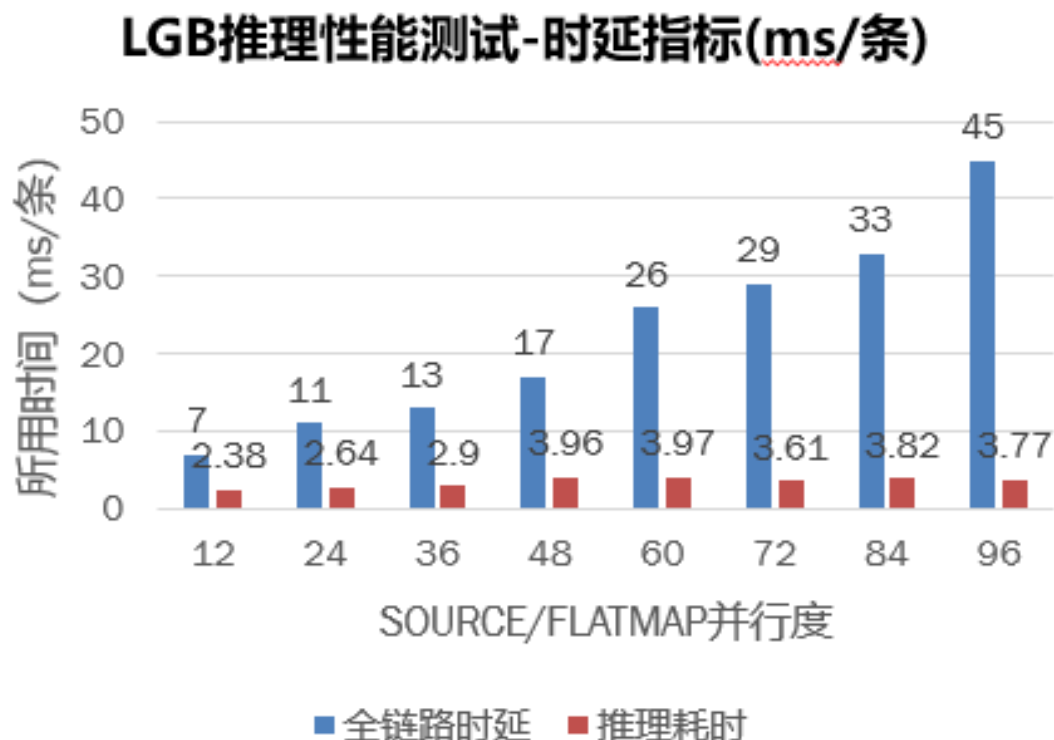
```
latency.metrics.interval: 60
metrics.reporter.promgateway.class: org.apache.flink.metrics.prometheus.PrometheusPushGatewayReporter
metrics.reporter.promgateway.host: 192.168.244.129
metrics.reporter.promgateway.port: 9091
metrics.reporter.promgateway.jobName: myJob
metrics.reporter.promgateway.randomJobNameSuffix: true
metrics.reporter.promgateway.deleteOnShutdown: true

taskmanager.network.memory.buffers-per-channel: 2
taskmanager.network.memory.float-buffers-per-gate: 2
taskmanager.network.memory.max-buffers-per-channel: 2
```

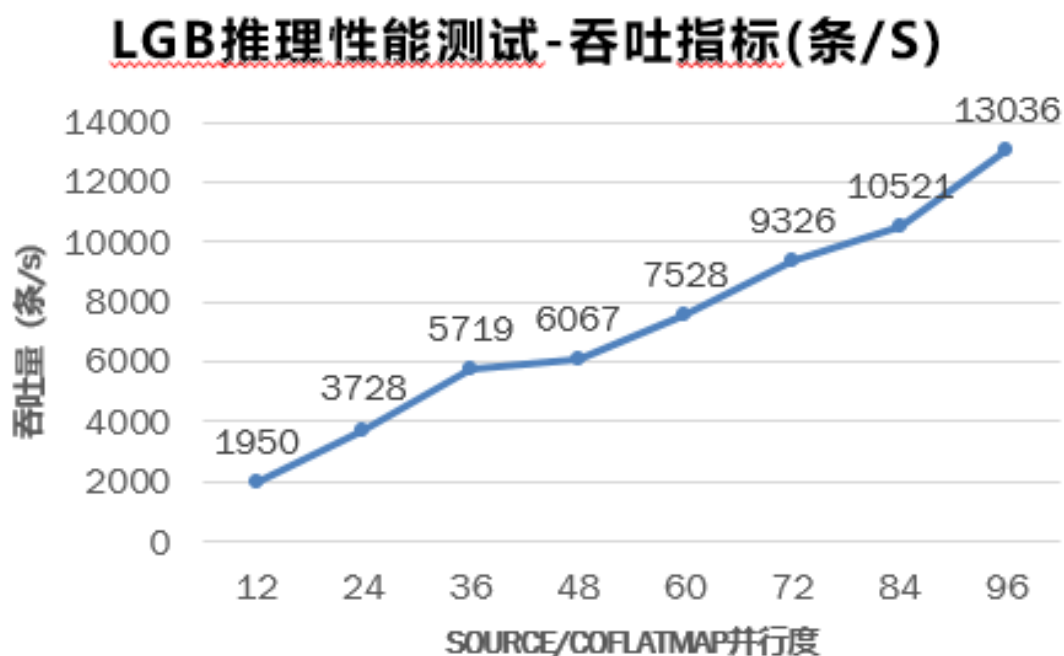
### 3.6 优化后性能分析

经过修改配置后，将任务再次提交到集群后，经过全链路时延计算公式、吞吐时延计算公式，最后得到优化后的结果

时延指标统计图如下：



吞吐指标统计图如下：



优化后 LGB 推理测试总结：

时延指标：并行度提升，时延也会增加，但幅度很小（可接受）。实际上，在测试过程中存在一定反压，若调大 SOURCE 与 COFLATMAP 的并行度比例，全链路时延可进一步降低；吞吐量指标：随着并行度的增加，吞吐量也随着提高，当并行度提高至 96 时，吞吐量可以达到 1.3W，此时的时延维持在 50ms 左右（比较稳定）；

### 3.7 优化前后 LGB 分析总结

如下图所示：

#### 3.7.1 吞吐量---影响因素：

内存：对吞吐和时延没什么影响，并行度与吞吐成正相关

1. 增大 kafka 分区,吞吐增加
2. 增大 source、维表 source 并行度
3. 增大 flatmap 推理并行度

#### 3.7.2 全链路时延---影响因素：

1. **Buffer 超时越短、个数越少、时延越低。**
2. 整个链路是否有算子**堵塞**（车道排队模型）
3. 调大推理算子并行度，时延降低，吞吐升高(即增加了推理的处理能力)