从 Prometheus 到 OpenTelemetry: 指标监控的演进与实践

在上一篇：从 Dapper 到 OpenTelemetry：分布式追踪的演进之旅我们讲解了 Trace 的一些核心概念：

* Trace
* Span
* Context
* Baggage 等

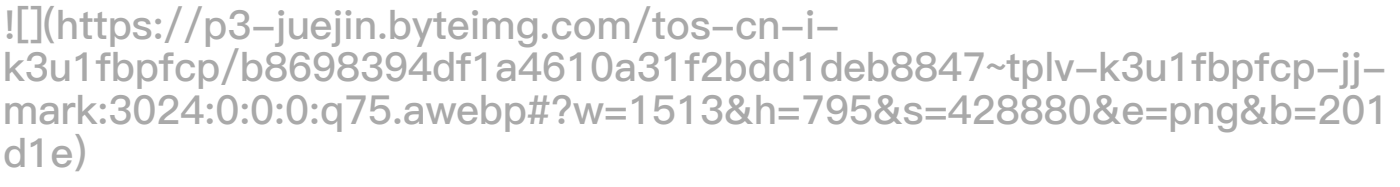
这次我们来讲另一个话题 `Metrics`。

背景

关于 metrics 我最早接触相关概念的就是 prometheus，它是第二个加入 CNCF（云原生）社区的项目（第一个是 kubernetes），可见在云原生领域 Metrics 指标监控从诞生之初就是一个非常重要的组件。

现实也确实如此，如今只要使用到了 kubernetes 相关的项目，对其监控就是必不可少的。

当然也不止是云原生的项目才需要 Metrics 指标监控，我们任何一个业务都是需要的，不然我们的服务运行对开发运维来说都是一个黑盒，无法知道此时系统的运行情况，因此才需要我们的业务系统将一些关键运行指标暴露出来。



业务数据：比如订单的增长率、销售金额等业务数据；同时还有应用自身的资

源占用情况：

* QPS
* Latency
* 内存
* CPU 等信息。

在使用 OpenTelemetry 之前，因为 prometheus 是这部分的绝对标准，所以我们通常都会使用 prometheus 的包来暴露这些指标：

```
<!-- The client -->
<dependency>
  <groupId>io.prometheus</groupId>
  <artifactId>simpleclient</artifactId>
  <version>0.16.0</version>
</dependency>
<!-- Hotspot JVM metrics-->
<dependency>
  <groupId>io.prometheus</groupId>
  <artifactId>simpleclient_hotspot</artifactId>
  <version>0.16.0</version>
</dependency>
```

暴露一个自定义的指标也很简单：

```
import io.prometheus.client.Counter;
class YourClass {
  static final Counter requests = Counter.build()
      .name("requests_total").help("Total requests.").register();

  void processRequest() {
    requests.inc();
    // Your code here.
  }
}
```

> 这是暴露一个单调递增的指标，prometheus 还提供了其他几种指标类型：

* Counter
* Gauge
* Histogram

之后我们只需要在 prometheus 中配置一些抓取规则即可：

```
scrape_configs:
  - job_name: 'springboot'
    scrape_interval: 10s
    static_configs:
      - targets: ['localhost:8080'] # Spring Boot ip+port
```

> 当然如果是运行在 kubernetes 环境，prometheus 也可以基于服务发现配置一些规则，自动抓取我们的 Pod 的数据，由于不是本文的重点就不过多介绍。

基本组件
====

在 OpenTelemetry 中自然也提供了 Metrics 这个组件，同时它也是完全兼容 Prometheus，所以我们理解和使用起来并不复杂。

MeterProvider

不同于 prometheus 客户端中直接提供了 Counter 就可以创建指标了，在 OpenTelemetry 中会提供一个 `MeterProvider` 的接口，使用这个接口可以获取 Meter，再使用 Meter 才可以创建 Counter、Gauge、Histogram 等数据。

下面来看看具体如何使用，这里我以 Pulsar 源码的代码进行演示：

```
public InstrumentProvider(OpenTelemetry otel) {
    if (otel == null) {
        // By default, metrics are disabled, unless the OTel java agent is
configured.
        // This allows to enable metrics without any code change.        otel
= GlobalOpenTelemetry.get();
    }    this.meter = otel.getMeterProvider()
            .meterBuilder("org.apache.pulsar.client")
            .setInstrumentationVersion(PulsarVersion.getVersion())
            .build();
}

LongCounterBuilder builder = meter.counterBuilder(name)
        .setDescription(description)
        .setUnit(unit.toString());
```

Meter Exporter

Meter Exporter 则是一个 OpenTelemetry 独有的概念，与我们之前讲到的一样：OpenTelemetry 作为厂商无关的平台，允许我们将数据写入到任何兼容的产品里。

所以我们在使用 Metrics 时需要指定一个 exporter：

Exporter 类型	作用	备注	参数
OTLP Exporter	通过 OpenTelemetry Protocol（OTLP） 发送指标数据到 collect。	默认生产环境中推荐使用，需要将数据发送到支持 OTLP 的后端，如 OpenTelemetry Collector。	-Dotel.metrics.exporter=otlp (default)
Console Exporter	将指标数据打印到控制台的导出器。	开发和调试，快速查看指标数据。	-Dotel.metrics.exporter=console
Prometheus Exporter	将指标数据以 Prometheus 抓取的格式暴露给 Prometheus 服务。	与 Prometheus 集成，适用于需要 Prometheus 监控的场景，这个可以无缝和以往使用 prometheus 的场景兼容	-Dotel.metrics.exporter=prometheus

Metric Instruments

与 prometheus 类似，OpenTelemetry 也提供了以下几种指标类型：

* **Counter**：单调递增计数器，比如可以用来记录订单数、总的请求数。
* **UpDownCounter**：与 Counter 类似，只不过它可以递减。
* **Gauge**：用于记录随时在变化的值，比如内存使用量、CPU 使用量等。
* **Histogram**：通常用于记录请求延迟、响应时间等。

同时每个指标还有以下几个字段：

* Name：名称，必填。
* Kind：类型，必填。
* Unit：单位，可选。
* Description：描述，可选。

```
messageInCounter = meter
        .counterBuilder(MESSAGE_IN_COUNTER)
        .setUnit("{message}")
        .setDescription("The total number of messages received for this topic.")
        .buildObserver();
```

还是以 Pulsar 的为例，`messageInCounter` 是一个记录总的消息接收数量的 Counter 类型。

```
subscriptionCounter = meter
        .upDownCounterBuilder(SUBSCRIPTION_COUNTER)
        .setUnit("{subscription}")
        .setDescription("The number of Pulsar subscriptions of the topic served by this broker.")
        .buildObserver();
```

这是记录一个订阅者数量的指标，类型是 UpDownCounter，也就是可以增加

减少的指标。

```
private static final List<Double> latencyHistogramBuckets =
        Lists.newArrayList(.0005, .001, .0025, .005, .01, .025, .05, .1, .25,
.5, 1.0, 2.5, 5.0, 10.0, 30.0, 60.0);


DoubleHistogramBuilder builder =
meter.histogramBuilder("pulsar.client.producer.message.send.duration")

        .setDescription("Publish latency experienced by the application,
includes client batching time")
        .setUnit(Unit.Seconds.toString())
        .setExplicitBucketBoundariesAdvice(latencyHistogramBuckets);
```

这是一个记录 Pulsar producer 发送延迟的指标，类型是 `Histogram`。

```
backlogQuotaAge = meter
        .gaugeBuilder(BACKLOG_QUOTA_AGE)
        .ofLongs()
        .setUnit("s")
        .setDescription("The age of the oldest unacknowledged message
(backlog).")
        .buildObserver();
```

这是一个记录最大 unack 也就是 backlog 时间的指标，类型是 `Gauge`。

案例
==

在之前的文章：[实战：如何编写一个 OpenTelemetry Extensions](http://cxyroad.com/"https://crossoverjie.top/2024/04/15/ob/how-to-write-otel-extensions/")中讲过如何开发一个 OpenTelemetry 的 extension，其实当时我就是开发了一个用于在 Pulsar 客户端中暴露指标的一个插件。

> 不过目前 Pulsar 社区已经集成了该功能。

其中的核心代码与上面讲到的类似：

```
public static void registerObservers() {
    Meter meter = MetricsRegistration.getMeter();

    meter.gaugeBuilder("pulsar_producer_num_msg_send")
            .setDescription("The number of messages published in the last
interval")
            .ofLongs()
            .buildWithCallback(
                    r -> recordProducerMetrics(r,
ProducerStats::getNumMsgsSent));

private static void recordProducerMetrics(ObservableLongMeasurement
observableLongMeasurement, Function<ProducerStats, Long> getter) {

    for (Producer producer :
CollectionHelper.PRODUCER_COLLECTION.list()) {
        ProducerStats stats = producer.getStats();
        String topic = producer.getTopic();
        if
(topic.endsWith(RetryMessageUtil.RETRY_GROUP_TOPIC_SUFFIX)) {
            continue;
        }        observableLongMeasurement.record(getter.apply(stats),
                Attributes.of(PRODUCER_NAME,
producer.getProducerName(), TOPIC, topic));
    }}
```

只是这里使用了 `buildWithCallback` 回调函数，OpenTelemetry 会每隔 30s 调用一次这个函数，通常适用于 Gauge 类型的数据。

```
java -javaagent:opentelemetry-javaagent.jar \
     -Dotel.javaagent.extensions=ext.jar  \
     -Dotel.metrics.exporter=prometheus \
     -Dotel.exporter.prometheus.port=18180 \
```

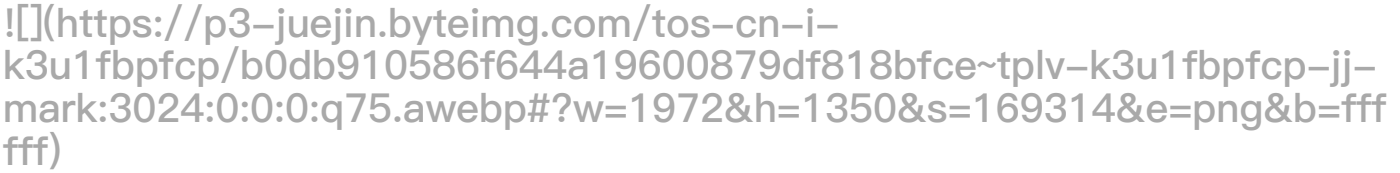
```
    -jar myapp.jar
```

配合上 Prometheus 的两个启动参数就可以在本地 18180 中获取到指标数据：

```
curl http://127.0.0.1:18180/metrics
```

当然也可以直接发往 OpenTelemetry-Collector 中，再由它发往 prometheus，只是这样需要额外在 collector 中配置一下：

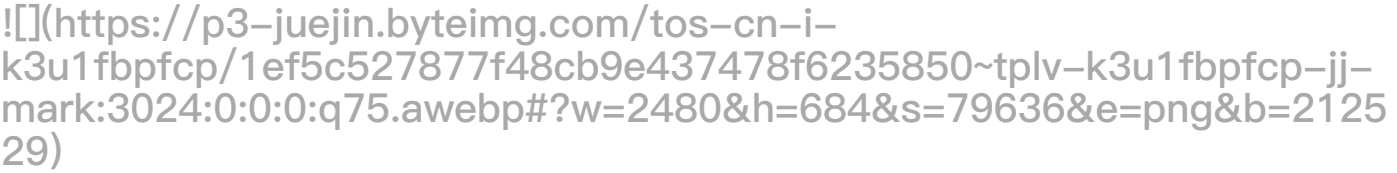
```
exporters:
  debug: {}
  otlphttp:
    metrics_endpoint: http://promethus:8480/insert/0/opentelemetry/api/v1/push
service:
  pipelines:
    metrics:
      exporters:
      - otlphttp
      processors:
      - k8sattributes
      - batch
      receivers:
      - otlp
```



这样我们就可以在 Grafana 中通过 prometheus 查询到数据了。

有一点需要注意，如果我们自定义的指标最好是参考官方的[语义和命名规范

](http://cxyroad.com/
”https://opentelemetry.io/docs/specs/semconv/general/metrics/”)来定义这些指标名称。



比如 OpenTelemetry 的规范中名称是用 **.** 来进行分隔的。

> 切换为 OpenTelemetry 之后自然就不需要依赖 prometheus 的包，取而代之的是 OTel 的包：

```
compileOnly ’io.opentelemetry:opentelemetry-sdk-extension-autoconfigure-spi:1.34.1’
compileOnly ’io.opentelemetry.instrumentation:opentelemetry-instrumentation-api:1.32.0’
```

总结

相对来说 Metrics 的使用比 Trace 简单的多，同时 Metrics 其实也可以和 Trace 进行关联，也就是 [Exemplars](http://cxyroad.com/
”https://opentelemetry.io/docs/specs/otel/metrics/data-model/#exemplars”)，限于篇幅就不在本文展开了，感兴趣的可以自行查阅。

参考链接：

* [github.com/apache/puls...](http://cxyroad.com/
”https://github.com/apache/pulsar/blob/master/pulsar-client/src/main/java/org/apache/pulsar/client/impl/metrics/InstrumentProvider.java”)
* [opentelemetry.io/docs/specs/...](http://cxyroad.com/
”https://opentelemetry.io/docs/specs/semconv/general/metrics/”)
* [opentelemetry.io/docs/specs/...](http://cxyroad.com/

”https://opentelemetry.io/docs/specs/otel/metrics/data-model/#exemplars”)
原文链接: https://juejin.cn/post/7368035468610666511