业务埋点及面板配置

- 监控的实现主要分为三个步骤：

（1）服务端设计监控指标，使用MicroMeter，在代码中埋下监控指标，通过sprinboot的actuator暴露/actuator/prometheus路径

（2）Prometheus定期抓取服务的/actuator/prometheus路径的数据

（3）使用Grafana连到Prometheus，绘制数据图表

1. MicroMeter简介

- 在MicroMeter中，有几个核心概念：
  - MeterRegistry，可以理解为创建和保存Meter的中心
  - Meter，指一组应用于收集应用中的度量数据的接口，在MicroMeter中，主要的Meter为：Timer，Counter，Gauge，DistributionSummary，LongTaskTimer，FunctionCounter，FunctionTimer和TimeGauge
  - Name与Tag，Name指的是某一个指标的名字，通过不同的Tag去区分多种维度进行数据统计
- 如何使用MircoMeter：
  - 引入micrometer jar包 compile 'io.micrometer:micrometer-registry-prometheus:1.3.0'
  - 在@SpringBootConfiguration中声明如下Bean，加入CommonTags，标识此应用的所有埋点都会加入如下标签，如这边加入的就是一个叫做application的tag，标识应用的名字用于区分监控指标，这里推荐一定要打这个tag，用于标识不同的项目

```
@Bean
MeterRegistryCustomizer<MeterRegistry> configure(@Value("${spring.application.name}") String applicationName){
    return registry -> registry.config().commonTags("application", applicationName);
}
```

- MeterRegistry
  - SimpleMeterRegistry：每个Meter的最新数据可以收集到SimpleMeterRegistry实例中，但是这些数据不会发布到其他系统，也就是数据是位于应用的内存中的。
  - CompositeMeterRegistry：多个MeterRegistry聚合，内部维护了一个MeterRegistry的列表。
  - 全局的MeterRegistry：工厂类io.micrometer.core.instrument.Metrics中持有一个静态final的CompositeMeterRegistry实例globalRegistry

SpringBoot中采用了(3)，故本文也只关注了GlobalRegistry的使用

- Meter，这里仅介绍几个频繁的Meter使用
  - Counter
    单值计数器，单调递增

    Counter接口允许使用者使用一个固定值(必须为正数)进行计数。

    使用示例： Metrics.counter("api.count", "uri", "/project/test").increment(); //参数1为meter的name，后面为Tag
  - Timer

    Timer(计时器)适用于记录耗时比较短的事件的执行时间，通过时间分布展示事件的序列和发生频率

    所有的Timer的实现至少记录了发生的事件的数量和这些事件的总耗时，从而生成一个时间序列

    使用示例：Metrics.Timer("api.timer", "uri", "/project/test").record(10, TimeUnit.Seconds)

    或者使用Timer.sample

    ```
    Timer.Sample sample = Timer.start(Metrics.globalRegistry);
    //
    sample.stop(Metrics.Timer("));
    ```

  - Gauge

    Gauge(仪表)是获取当前度量记录值的句柄，也就是它表示一个可以任意变动的单数值度量Meter

    如某个数值的波动与变化，如mapSize或者ListSize等等

    使用示例： Metrics.gauge("list.count", list.size());

- 关于Tag的设计

如果说name标识的是某一类数据，如http.request.count，很清楚的标识了其http请求次数的含义

那么tag就是对这个name的更细粒的区分

如http.request.count("application", "chainAccessPipe", "uri", "/test", "method", "Get")标识了http请求次数对应的几个标签，应用名，uri，http方法

我们可以通过筛选tag，来获得我们所需关注的某个或某几个标签所对应的数据

比如我们可以通过application标签来筛选chainAccessPipe的数据而不需要其他application不为chainAccessPipe的数据

2. 埋点通用建议

监控的目的在于检测线上环境中服务的运行状况，为了更清晰的了解线上服务的状态，我们对机器的物理状态，服务的性能，业务指标等等都需要进行监控

micrometer-registry-prometheus的jar包已经为我们提供了许多通用性能监控的集成，如：

cpu的状态，数据库连接池的状态，jvm的状态，http请求的次数，时间等等，我们所要做的就是利用其提供的指标，得到我们想要的数值

- 一般来说，对于性能指标

对于CPU，我们关注于cpu的使用情况百分比

对于数据库连接池，我们可能关注于active的线程所占的百分比与idle的线程所占的百分比

对于jvm，关注于gc_pause的频率，每次pause的时间，还有堆内存使用情况等等

对于http请求，我们可能关注于请求的频率，请求的响应时间等数据

这些指标的收集micrometer-registry-prometheus已经为我们做好了，不需要我们再做什么，所需要做的就是绘制图表，这将会在（4）中介绍

- 对于业务指标，我们需要结合meter的属性，

counter，是用来计数的

timer，包含了计数与计时的功能，所有需要记录速率的地方需要使用timer

gauge，可以记录某个数值的变化，如map的size，list的size等等

一些通用的业务指标，比如某些重点service的处理速率（如TPS），kafka的consumer的处理速率等等，可以使用timer记录

还有异常出现的次数，频率等，可以使用counter记录

某些实例数的变化，或者某些需要关注的map或者list的size，可以使用gauge记录

这些都是业务指标所必须要关注的点，方便于运维人员或者是开发人员甚至是测试人员及时发现问题，定位问题

如在chainaccess系统中，我们使用了timer来记录KafkaConsumer的处理时间

```
@KafkaListener(topics = CHAIN_ACCESS_TX_COMMON_TOPIC)
public void consumeTransaction(ConsumerRecord<String, String> record) {
     Timer.Sample sample = Timer.start(Metrics.globalRegistry);

     //business logic

     sample.stop(txCommonTimer);

}
CounterExceptiongaugemapSize
```

- 一般来说，我们会将所有的meter放置一个公共的地方，比如KafkaConsumer中用到的所有meter，我将其放在一起，方便于维护与拓展

```
 public class KafkaConsumerMeters {

     public static Timer txCommonTimer() {
         return Metrics.timer("kafka.consumer.tx.common", Tags.of("topic", CHAIN_ACCESS_TX_COMMON_TOPIC));
     }

     public static Timer getTxReceiptTimer() {
         return Metrics.timer("kafka.consumer.tx.common", Tags.of("topic", CHAIN_ACCESS_GET_TX_RECEIPT_COMMON
_TOPIC));
     }

     public static Counter invalidSecretCounter() {
         return Metrics.counter("kafka.consumer.tx.common.exception.time", Tags.of("exceptionType", "invalidS
ecret"));
     }

     ..............
   }
```

- 业务指标埋点的目的是为了方便于运维人员或者是开发人员甚至是测试人员及时发现问题，定位问题，故而业务埋点，需要对业务有着

深刻的理解，能够梳理好业务，细化出我们所需要的数据，设计出良好的meter的name与tag，数据归纳的时候才能有更好的展示

3. Prometheus概述及建议

- 关于Prometheus的原理及设计这里便不做过多的讲述，可以自行了解

我们只需要了解Prometheus是一个时序数据库，它可以通过配置，定期的去抓取某个URI的数据，所存储的数据的索引为当前时间

value即为抓取到的数据，同时它提供了PromQL的各类查询语句，函数等等，供我们使用来聚合各类数据，运算出我们所关注的数值，形成图表

- 一个简单的Prometheus抓取配置示例：

```
scrape_configs:
# The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
- job_name: 'admin_prometheus'

# metrics_path defaults to '/metrics'
# scheme defaults to 'http'.
metrics_path: actuator/prometheus

static_configs:
- targets: ['localhost:9090']
```

- Prometheus还能很好的接入k8s，kafka，监控这些组件的状态

目前我们公司的Prometheus部署情况为：

多个K8s集群，每个K8s集群中都内置了Prometheus监控自身的集群状态

同时在外部还部署了一个Prometheus抓取所有的K8s的集群状态

这样是从一个宏观的角度在设计这个监控维度，监控的数据标签也不仅仅限于某个应用，而是监控了所有集群的所有应用，我们只需要筛选出我们所需要的Tag即可

- 要想使我们的应用接入Prometheus，只需修改某个K8s集群的某个服务对应的service的yaml文件，在"metadata"下加入如下标签即可

```
"annotations": {
   "prometheus.io/app-metrics-path": "/actuator/prometheus",
   "prometheus.io/scrape": "true"
 }
```

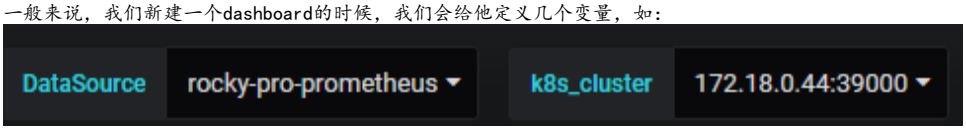
这样子Prometheus就能通过抓取该应用的/actuator/prometheus路径采集到数据指标了，当然应用也需暴露这个uri

需要注意的点是，所抓取的指标中也将会加入许多K8S集群的Tag（如instance，k8s_cluster等等）

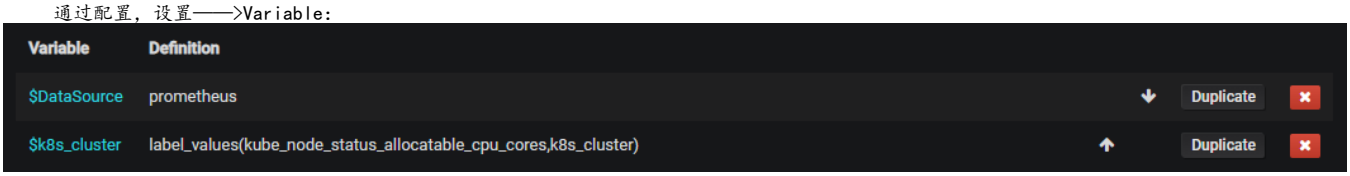
之后进行筛选时我们需要过滤出我们所关注的应用信息

4. Grafana图表绘制

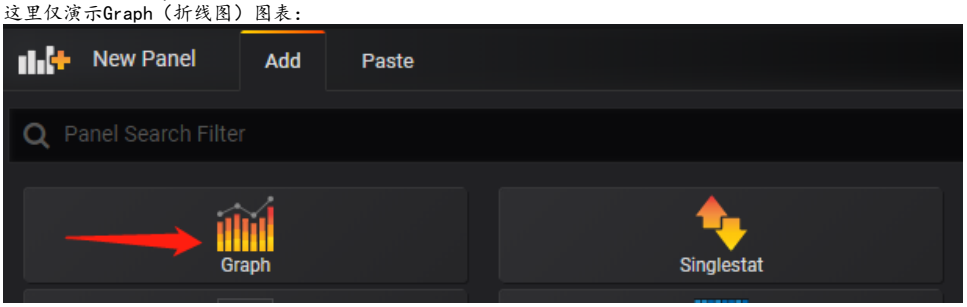
- Grafana的图表绘制实际上也是通过Prometheus的PromQL来聚合数据的
- 一般来说，我们新建一个dashboard的时候，我们会给他定义几个变量，如：



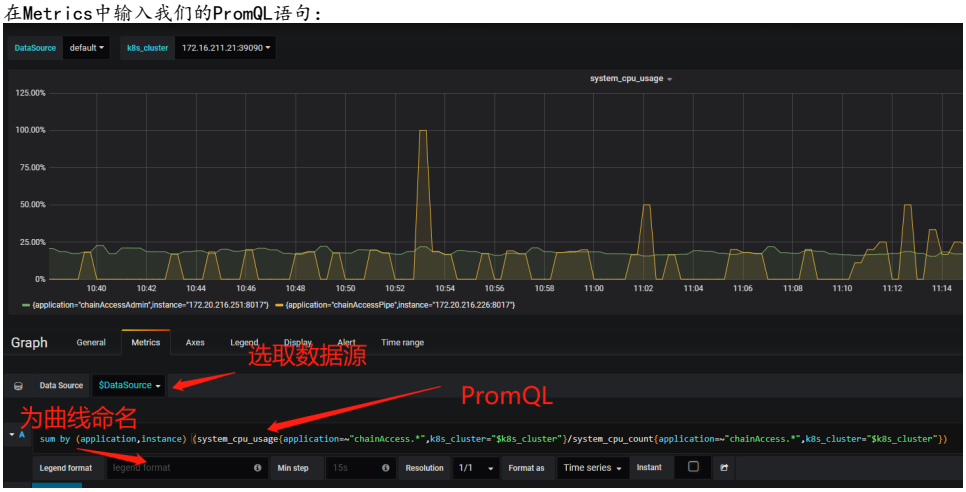
通过配置，设置——>Variable：



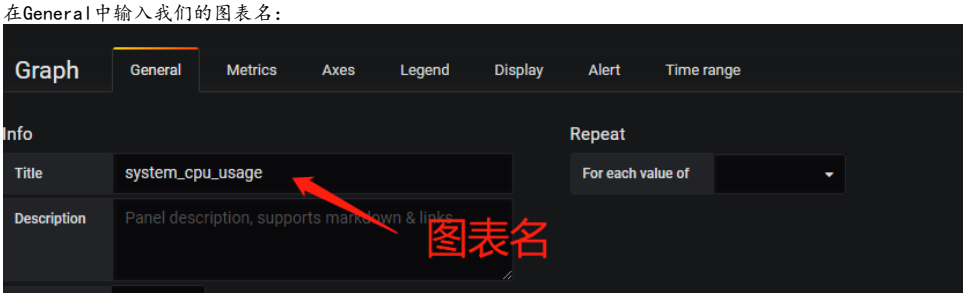
- 绘制图表：
  - 点击右上角的add panel按钮： 
  - 这里仅演示Graph（折线图）图表：



- 在Metrics中输入我们的PromQL语句：



- 在General中输入我们的图表名：



- 在Axes中配置横纵坐标的各类属性（如：单位等等）：

- 可以选择所要看的时间：
- 最终绘制的图表只是上面我们所收集的指标的一个展示，我们关注于速率，关注于百分比，关注于不同的服务之间的对比

这里推荐同种类型的数据应该放在一张图表中，如system_cpu_usage标识的是系统的cpu的使用量，通过不同的tag（如application，instance等等）来区分不同的应用

所以归根到底，还是需要在埋点时设计好指标的name与tag，在绘图的时候才能够更容易的满足我们的需求

- 图表示例：


ChainAccess
Last 1 hour
DataSource
default
k8s_cluster
172.16.211.21:39090
system_cpu_usage
jvm_gc_pause_per_minute
jvm_gc_pause_process_time
jvm_heap_usage_percentage
chain_connector_web3jMap_size
3
http_server_requests_per_minute
http_server_requests_response_time