> 本文由 简悦 SimpRead 转码， 原文地址 kaiwu.lagou.com

用户对于应用的性能总是有着苛刻的要求。在目前的市场上，每一个服务都有着不少的替代选项。如果你的网页打开速度不够快，或者你的 App 在每次刷新时总是长时间显示加载中的图标，那么你多半会失去这个用户。提升性能的前提是了解应用的性能，知道整个系统的瓶颈在哪里。这就需要收集足够多的性能指标数据，并以可视化的形式展现出来。本课时介绍性能指标数据的捕获、收集、分析和展示，用到的工具包括 Micrometer、Prometheus 和 Grafana 等。

捕获数据

第一步是捕获需要收集的性能指标数据，每个应用有自己感兴趣的**性能指标**，性能指标通常可以分成两类，即**业务无关**和**业务相关**。业务无关的性能指标所提供的信息比较底层，比如数据库操作的时间、API 请求的响应时间、API 请求的总数等。业务相关的性能指标的抽象层次较高，比如订单的总数、总的交易金额、某个业务端到端的处理时间等。

Micrometer

为了捕获性能指标数据，我们需要在应用中特定的地方添加代码，不同的编程语言有各自不同的库。Java 平台目前流行的是 Micrometer。Spring Boot 提供了对 Micrometer 的自动配置，只需要添加相关的依赖即可。

Micrometer 为 Java 平台上的性能数据收集提供了一个通用的 API，类似于 SLF4J 在日志记录上的作用。应用程序只需要使用 Micrometer 的通用 API 来收集性能指标数据即可。Micrometer 会负责完成与不同监控系统的适配工作，使得切换监控系统变得很容易，避免了供应商锁定的问题。Micrometer 还支持同时推送数据到多个不同的监控系统。

Micrometer 中有两个最核心的概念，分别是**计量器**（Meter）和**计量器注册表**（Meter Registry）。计量器表示的是需要收集的性能指标数据，而计量器注册表负责创建和维护计量器，每个监控系统有自己独有的计量器注册表实现。

Spring Boot Actuator 提供了对 Micrometer 的自动配置，会自动创建一个组合注册表对象，并把 CLASSPATH 上找到的所有支持的注册表实现都添加进来。只需要在 CLASSPATH 上添加相应的第三方库，Spring Boot 会完成所需的配置。如果需要对该注册表进行配置，添加类型为 MeterRegistryCustomizer 的 bean 即可，如下面的代码所示。在需要使用注册表的地方，可以通过依赖注入的方式来使用 MeterRegistry 对象。

```
@Configuration

public class ApplicationConfig {

@Bean

public MeterRegistryCustomizer<MeterRegistry> meterRegistryCustomizer() {

return registry -> registry.config().commonTags("service", "address");

  }
```

```
}
```

每个计量器都有自己的名称。由于不同的监控系统有自己独有的推荐命名规则，Micrometer 使用英文句点 “.” 分隔计量器名称中的不同部分，比如 a.b.c。Micrometer 会负责完成所需的转换，以满足不同监控系统的需求。

每个计量器在创建时都可以指定一系列标签，标签以名值对的形式出现，监控系统使用标签对数据进行过滤。除了每个计量器独有的标签之外，每个计量器注册表还可以添加通用标签，所有该注册表导出的数据都会带上这些通用标签。上面代码中的 MeterRegistryCustomizer 添加了通用标签 service，对应的值是 address。

计量器用来收集不同类型的性能指标信息。Micrometer 提供了不同类型的计量器实现。

计数器

计数器（Counter） 表示的是单个只允许增加的值。通过 MeterRegistry 的 counter 方法来创建表示计数器的 Counter 对象；还可以使用 Counter.builder 方法来创建 Counter 对象的构建器。Counter 所表示的计数值是 double 类型，其 increment 方法可以指定增加的值，默认情况下增加的值是 1.0。如果已经有一个方法返回计数值，可以直接从该方法中创建类型为 FunctionCounter 的计数器。

下面的代码展示了计数器的创建和使用。

```
@Service

public class CounterService {

private final Counter counter;

public CounterService(

final MeterRegistry meterRegistry) {

this.counter = Counter.builder("simple.counter1")

        .description("A simple counter")

        .tag("type", "counter")

        .register(meterRegistry);

  }

public void count() {

this.counter.increment();
```

```
  }

}
```

计量仪

计量仪（Gauge） 表示的是单个变化的值。与计数器的不同之处在于，计量仪的值并不总是增加的。与创建 Counter 对象类似，Gauge 对象可以从计量器注册表中创建，也可以使用 Gauge.builder 方法返回的构建器来创建。

下面的代码展示了计量仪的创建。当 Gauge 对象创建之后，每次捕获数据时，会自动调用创建时提供的方法来获取最新值。

```
@Service

public class GaugeService {

public GaugeService(final MeterRegistry meterRegistry) {

    Gauge.builder("simple.gauge1", this, GaugeService::getValue)

        .description("A simple gauge")

        .tag("type", "gauge")

        .register(meterRegistry);

  }

private double getValue() {

return ThreadLocalRandom.current().nextDouble();

  }

}
```

计时器

计时器（Timer）通常用来记录事件的持续时间。计时器会记录两类数据：**事件的数量**和**总的持续时间**。在使用计时器之后，就不再需要单独创建一个计数器，计时器可以从注册表中创建，或者使用 Timer.builder 方法返回的构建器来创建。Timer 提供了不同的方式来记录持续时间，第一种方式是使用 record 方法来记录 Runnable 和 Callable 对象的运行时间；第二种是使用 Timer.Sample 来手动启动和停止计时。

如果一个任务的耗时很长，直接使用 Timer 对象并不是一个好的选择，因为 Timer 对象只有在任务完成之后才会记录时间。更好的选择是使用 LongTaskTimer 对象。LongTaskTimer 对象可以在任务进行中记录已经耗费的时间，它通过注册表的 more().longTaskTimer 方法来创建。

下面代码展示了计时器的创建和使用，其中的 record 方法记录 Runnable 对象的执行时间，start 方法用来启动计时，stop 方法用来停止计时。

```
@Service

public class TimerService {

private final MeterRegistry meterRegistry;

private final Timer timer1;

private Timer.Sample sample;

public TimerService(

final MeterRegistry meterRegistry) {

this.meterRegistry = meterRegistry;

this.timer1 = Timer.builder("simple.timer1")

        .description("A simple timer 1")

        .register(meterRegistry);

  }

public void record() {

this.timer1.record(() -> {

try {
```

```
        Thread.sleep(ThreadLocalRandom.current().nextLong(1000));

      } catch (final InterruptedException e) {

      }

    });

  }

public void start() {

this.sample = Timer.start(this.meterRegistry);

  }

public void stop() {

this.sample.stop(Timer.builder("simple.timer2")

        .description("A simple timer 2")

        .register(this.meterRegistry));

  }

}
```

大多数时候，需要计时的是一个方法的执行时间，此时更简单的做法是使用 @Timed 注解。下面代码中的 @Timed 注解添加在 search 方法上，指定了性能指标的名称和发布的百分比数值。

```
@Timed(value = "happyride.address.search", percentiles = {0.5, 0.75, 0.9})

public List<AddressView> search(final Long areaCode, final String query) {

}
```

为了 @Timed 注解可以生效，需要添加 Spring AOP 和 AspectJ 的 aspectjweaver 依赖，同时使用 @EnableAspectJAutoProxy 注解来启用 AspectJ 的支持，并创建 TimedAspect 对象，如下面的代码所示。

```
@Configuration

@EnableAspectJAutoProxy

public class ApplicationConfig {

@Bean

public TimedAspect timedAspect(final MeterRegistry meterRegistry) {

return new TimedAspect(meterRegistry);

  }

}
```

分布概要

分布概要（Distribution Summary）用来记录事件的分布情况。计时器本质上也是一种分布概要，表示分布概要的 DistributionSummary 对象可以从注册表中创建，也可以使用 DistributionSummary.builder 方法提供的构建器来创建。分布概要根据每个事件所对应的值，把事件分配到对应的桶（Bucket）中。Micrometer 默认的桶值从 1 到最大的 long 值，可以通过 minimumExpectedValue 和 maximumExpectedValue 来控制值的范围。

如果事件所对应的值较小，可以通过 scale 来设置一个值来对数值进行放大。与分布概要密切相关的是直方图和百分比（percentile）。大多数时候，我们并不关注具体的数值，而是数值的分布区间，比如在查看 HTTP 服务响应时间的性能指标时，选择几个重要的百分比，如 50%、75% 和 90% 等，关注的是这些百分比数量的请求都在多少时间内完成。

下面的代码展示了分布概要的创建和使用。

```
@Service

public class DistributionSummaryService {

private final DistributionSummary summary;

public DistributionSummaryService(final MeterRegistry meterRegistry) {

this.summary = DistributionSummary.builder("simple.summary")
```

```
        .description("A simple distribution summary")

        .tag("type", "distribution_summary")

        .minimumExpectedValue(1.0)

        .maximumExpectedValue(10.0)

        .publishPercentiles(0.5, 0.75, 0.9)

        .register(meterRegistry);

  }

public void record(final double value) {

this.summary.record(value);

  }

}
```

发布数据

在捕获了性能指标数据之后，下一步是把这些数据发布到后台的处理系统。有很多开源和商用的系统可供选择，Micrometer 都可以提供集成，示例应用使用的是 Prometheus，与其他监控系统的不同在于，Prometheus 采取的是主动抽取数据的方式，也就是拉模式。因此客户端需要暴露 HTTP 服务，并由 Prometheus 定期来访问以获取数据。

Prometheus 数据抓取

对于 Prometheus 来说，Spring Boot Actuator 会自动配置一个 URL 为 /actuator/prometheus 的 HTTP 服务来供 Prometheus 抓取数据。不过该 Actuator 服务默认是关闭的，需要通过 Spring Boot 的配置打开，如下面的代码所示。

```
management:

endpoints:

enabled-by-default: false
```

```
web:

exposure:

include: prometheus

endpoint:

prometheus:

enabled: true
```

接下来需要配置 Prometheus 来抓取应用提供的数据。下面的代码是 Prometheus 的配置文件 prometheus.yml 的内容，其中 scrape_interval 设置抓取数据的时间间隔，scrape_configs 设置需要抓取的目标，这里使用的是静态的服务器地址。Prometheus 支持抓取目标的自动发现，具体请查看官方文档。

```
global:

scrape_interval: 10s

scrape_configs:

- job_name: 'simple'

metrics_path: '/actuator/prometheus'

static_configs:

- targets:

- "localhost:8080"
```

使用推送网关

Prometheus 的一般工作模式是拉模式，也就是由 Prometheus 主动的定期获取数据，对于一些运行时间较短的任务来说，拉模式不太适用。这些任务的运行时间可能短于 Prometheus 的数据抓取间隔，导致数据无法被收集，此时应该使用推送网关（Push Gateway）来主动推送数据，这是一个独立的应用，作为应用和 Prometheus 服务器之间的中介。应用推送数据到推送网关，Prometheus 从推送网关中拉取数据。

Spring Boot Actuator 提供了对推送网关的自动配置，可以定期推送数据。应用的代码只需要通过正常的方式使用 Micrometer 发布性能指标数据即可。下面的代码给出了推送网关的相关配置。

```
management.metrics.export.prometheus.pushgateway:

enabled: true

base-url: http://localhost:9091

job: batch-task

shutdown-operation: push

grouping-key:

instance: ${random.value}
```

下表是推送网关的配置项说明。

配置项	说明
base-url	推送网关的地址
job	任务的名称
shutdown-operation	应用关闭时的行为，push 的含义是在关闭时推送一次数据
grouping-key	分组名称和值

在每次进行推送时，如果性能指标的名称，以及分组名称和值都相同，推送的数据会替换之前的值。Prometheus 在抓取推送网关的数据时，使用的是 /metrics 路径。

节点导出工具

Prometheus 提供了节点导出工具（Node exporter）来收集硬件相关的性能指标数据，包括 CPU、内存、文件系统和网络等。

Grafana

Prometheus 提供了界面来查询性能指标的值，并绘制简单的图形。如果需要更强大的展示方式，可以使用 Grafana，其可以用 Prometheus 作为数据源，并提供了不同类型的图表和表格作为展示方式，同时还提供了图形化界面来对图表进行配置。当以 Prometheus 作为数据源时，则需要了解基本的 Prometheus 查询语法。

最基本的查询方式是使用性能指标的名称，这样可以查询到收集的原始数据，比如 http_request_total 表示所有 HTTP 请求的计数器。可以使用标签进行过滤，比如查询 http_request_total{handler="/search/"} 使用标签 handler 来进行过滤。Prometheus 还支持操作符和函数，典型的操作符如 sum、min、max、avg 和 topk 等，函数包括计算增加速率的 rate()、进行排序的 sort()、计算绝对值的 abs() 等。

Grafana 的界面简单易用，通过 Prometheus 查询得到数据之后，选择不同的图表，再进行配置即可。

Prometheus Operator

性能指标数据的分析需要一个完整的技术栈，包括 Prometheus 服务器、推送网关、节点导出工具和 Grafana 等。在 Kubernetes 上手动安装和配置这些不同的工具是一件非常耗时的任务。更好的做法是使用 Prometheus Operator。

使用 Helm 3 来安装 Prometheus Operator，安装使用的名称空间是 monitoring。

```
helm install prom-o -n monitoring stable/prometheus-operator
```

安装完成之后，所有的相关服务都会启动。Prometheus Operator 会自动收集 Kubernetes 自身的性能指标数据。对于应用的性能指标数据，需要添加服务监控器（Service Monitor）对象，服务监控器是 Prometheus Operator 提供的 Kubernetes 上的自定义资源定义。下面代码给了地址管理服务的服务监控器对象，指定了 Kubernetes 服务的选择器，以及抓取数据的路径。

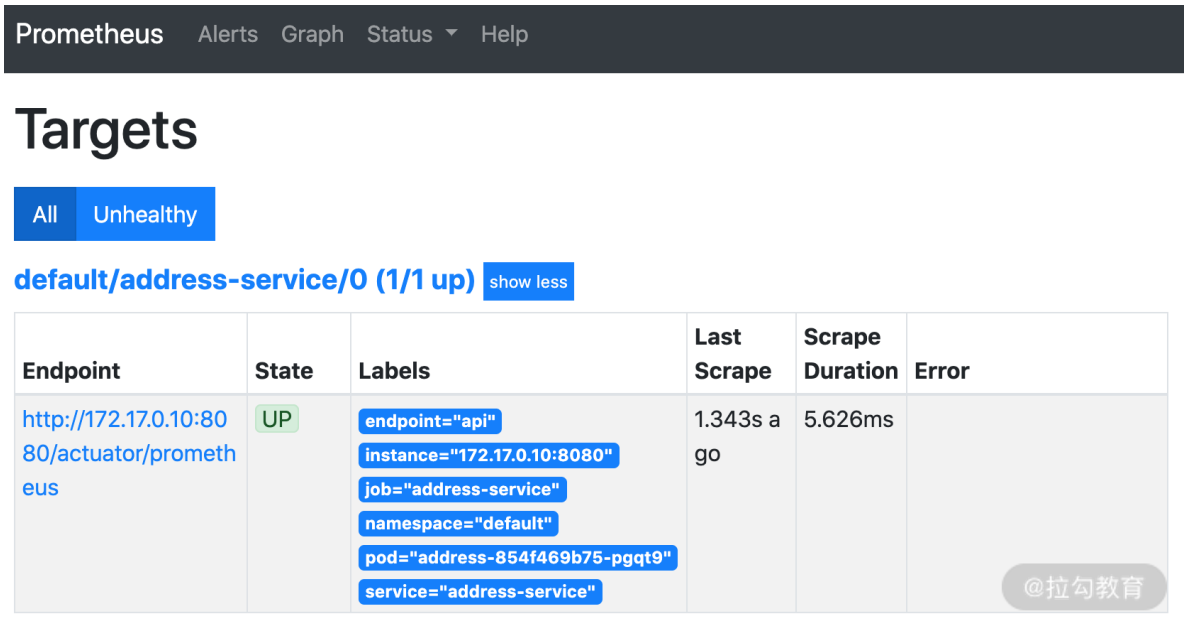
```
apiVersion: monitoring.coreos.com/v1

kind: ServiceMonitor

metadata:

name: address-service

labels:

release: prom-o

spec:

selector:

matchLabels:

app.kubernetes.io/name: address

namespaceSelector:

matchNames:

- default
```

```
endpoints:

- port: api

interval: 10s

path: "/actuator/prometheus"

honorLabels: true
```

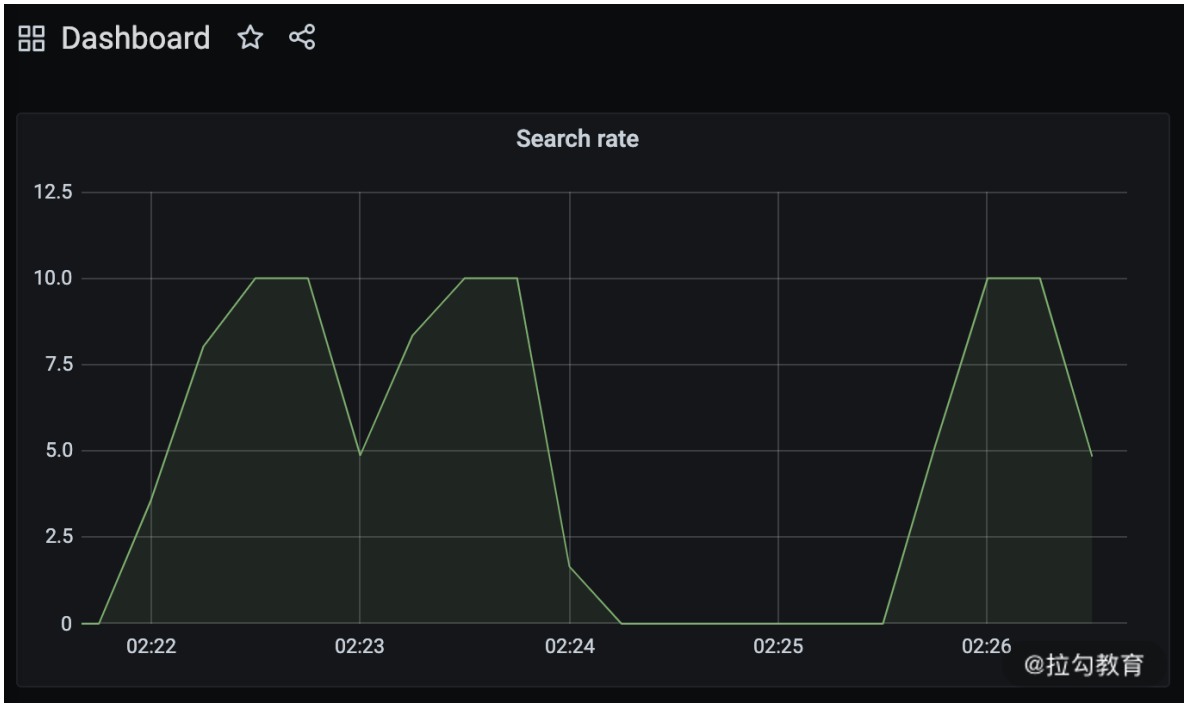
在开发中，可以通过 kubectl 提供的端口转发功能来访问 Prometheus 和 Grafana 服务。通过运行下面的命令，可以在本地机器的 9090 端口访问 Prometheus 的界面。

```
kubectl port-forward -n monitoring svc/prometheus-operated 9090:9090
```

在下图所展示的 Prometheus 的抓取目标界面中，可以看到新添加的地址管理服务。



通过 Grafana 的界面可以创建出各种类型的图表。下图是地址管理服务的搜索操作的请求处理速度。



总结

提升系统性能的首要前提是了解系统的性能，只有收集足够多的性能指标数据，才能找到性能优化的正确方向，并及时处理性能相关的问题。通过本课时的学习，你应该掌握如何使用 Micrometer 来在代码中捕获性能指标数据，以及如何用 Prometheus 来抓取数据。对于 Prometheus 中的数据，可以使用 Grafana 来进行展示，你还应该了解如何在 Kubernetes 上使用 Prometheus Operator，从而构建自己的性能监控的技术栈。