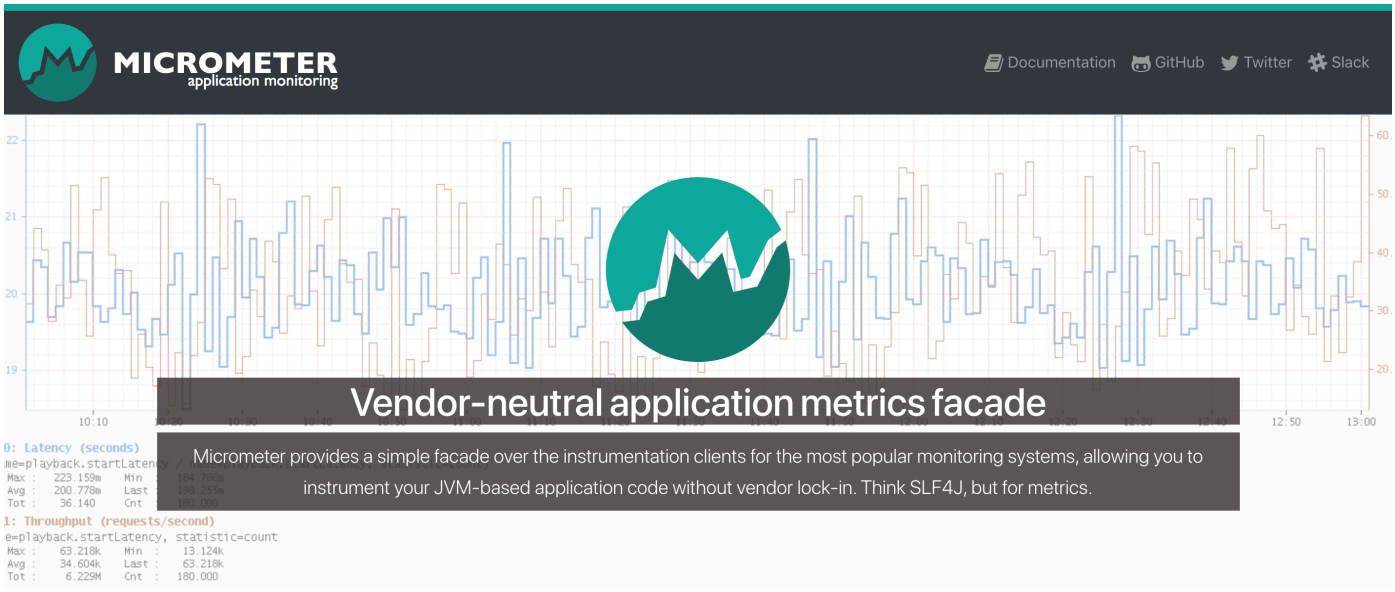
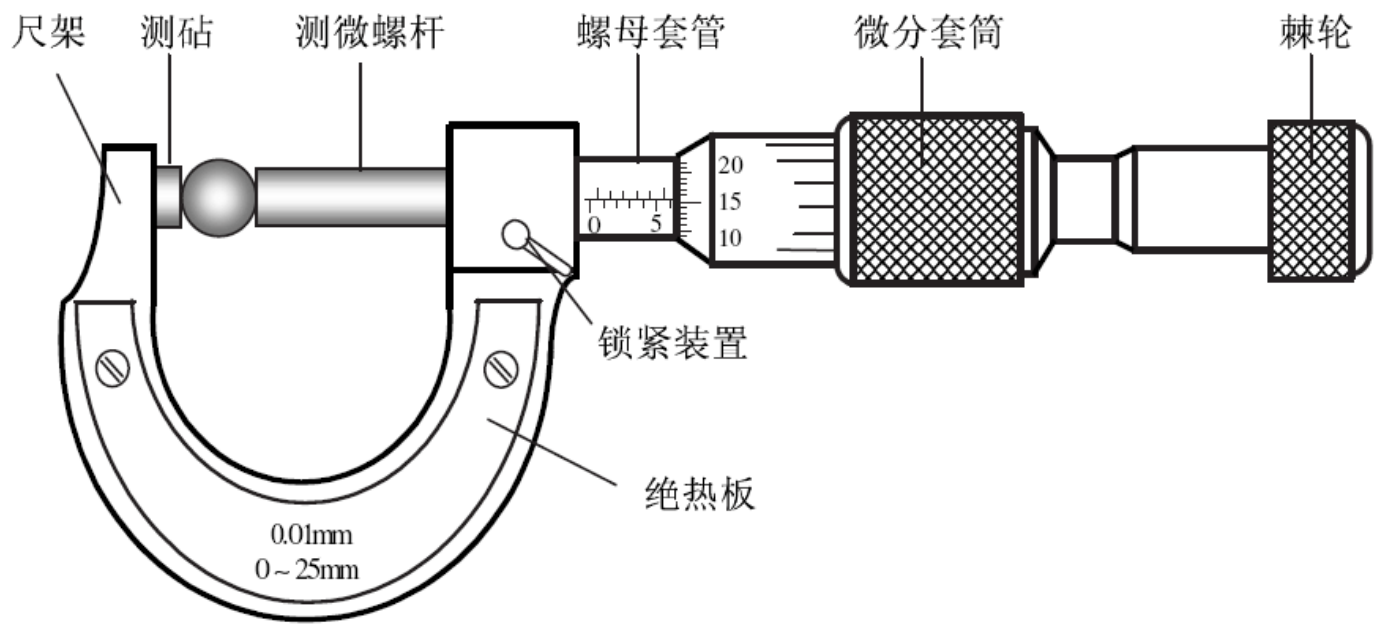
Micrometer为最流行的监控系统提供了一个简单的仪表客户端外观，允许仪表化JVM应用，而无需关心是哪个供应商提供的指标。它的作用和SLF4J类似，只不过它关注的不是Logging（日志），而是application metrics（应用指标）。简而言之，它就是应用监控界的SLF4J。



Micrometer（译：千分尺）



不妨看看SLF4J官网上对于SLF4J的说明：Simple Logging Facade for Java (SLF4J)

现在再看Micrometer的说明：Micrometer provides a simple facade over the instrumentation clients for the most popular monitoring systems.

Metrics（译：指标，度量）

Micrometer提供了与供应商无关的接口，包括 **timers（计时器）**， **gauges（量规）**， **counters（计数器）**， **distribution summaries（分布式摘要）**， **long task timers（长任务定时器）**。它具有维度数据模型，当与维度监视系统结合使用时，可以高效地访问特定的命名度量，并能够跨维度深入研究。

支持的监控系统：AppOptics， Azure Monitor， Netflix Atlas， CloudWatch， Datadog， Dynatrace， Elastic， Ganglia， Graphite， Humio， Influx/Telegraf， JMX， KairosDB， New Relic， Prometheus， SignalFx， Google Stackdriver， StatsD， Wavefront

\1. 安装

Micrometer记录的应用程序指标用于观察、告警和对环境当前/最近的操作状态做出反应。

为了使用Micrometer，首先要添加你所选择的监视系统的依赖。以Prometheus为例：

```
<dependency>
  <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
    <version>${micrometer.version}</version>
</dependency>
```

\2. 概念

2.1. Registry

Meter是收集关于你的应用的一系列指标的接口。Meter是由MeterRegistry创建的。每个支持的监控系统都必须实现MeterRegistry。

Micrometer中包含一个SimpleMeterRegistry，它在内存中维护每个meter的最新值，并且不将数据导出到任何地方。如果你还没有一个首选的监测系统，你可以先用SimpleMeterRegistry：

```
MeterRegistry registry = new SimpleMeterRegistry();
```

注意：如果你用Spring的话，SimpleMeterRegistry是自动注入的

Micrometer还提供一个CompositeMeterRegistry用于将多个registries结合在一起使用，允许同时向多个监视系统发布指标。

```
CompositeMeterRegistry composite = new CompositeMeterRegistry();

Counter compositeCounter = composite.counter("counter");
compositeCounter.increment();

SimpleMeterRegistry simple = new SimpleMeterRegistry();
composite.add(simple);

compositeCounter.increment();
```

2.2. Meters

Micrometer提供一系列原生的Meter，包括Timer , Counter , Gauge , DistributionSummary , LongTaskTimer , FunctionCounter , FunctionTimer , TimeGauge。不同的meter类型导致有不同的时间序列指标值。例如，单个指标值用Gauge表示，计时事件的次数和总时间用Timer表示。

每一项指标都有一个唯一标识的名字和维度。“维度”和“标签”是一个意思，Micrometer中有一个Tag接口，仅仅因为它更简短。一般来说，应该尽可能地使用名称作为轴心。

（PS：指标的名字很好理解，维度怎么理解呢？如果把name想象成横坐标的话，那么dimension就是纵坐标。Tag是一个key/value对，代表指标的一个维度值）

2.3. Naming meters（指标命名）

Micrometer使用了一种命名约定，用.分隔小写单词字符。不同的监控系统有不同的命名约定。每个Micrometer的实现都要负责将Micrometer这种以.分隔的小写字符命名转换成对应监控系统推荐的命名。你可以提供一个自己的NamingConvention来覆盖默认的命名转换：

```
1 registry.config().namingConvention(myCustomNamingConvention);
```

有了命名约定以后，下面这个timer在不同的监控系统中看起来就是这样的：

```
1 registry.timer("http.server.requests");
```

在Prometheus中，它是http_server_requests_duration_seconds

在Atlas中，它对应的是httpServerRequests

在InfluxDB中，对应的是http_server_requests

（PS：每项指标都有一个名字，不同的监控系统的命名规则（风格）都不太一样，因此可能同一个指标在不同的监控系统中有不同的名字。简单地来说，比如内存使用率这个指标可能在Prometheus中用MemoryUsage表示，在InfluxDB中用mem_usage表示，因此每个监控系统都要提供一个命名转换器，当看到mem.usage的时候InfluxDB应该知道说的是内存使用率，对应的指标名称是mem_usage。这就好比，中文“你好”翻译成英文是“hello”，翻译成日文是“こんにちは”）

2.3.1. Tag naming

假设，我们想要统计HTTP请求数和数据库调用次数，那么可以这样写：

```
1 registry.counter("database.calls", "db", "users");        // 数据库调用次数
2 registry.counter("http.requests", "uri", "/api/users");  // HTTP请求数
```

2.3.2. Common tags

Common tags可以被定义在registry级别，并且会被添加到每个监控系统的报告中

预定义的Tags有host , instance , region , stack等

```
registry.config().commonTags("stack", "prod", "region", "us-east-1");
registry.config().commonTags(Arrays.asList(Tag.of("stack", "prod"), Tag.of("region",
"us-east-1"))); // 二者等价
```

2.3.4. Tag values

Tag values must be non-null

2.4. Meter filters

每个registry都可以配置指标过滤器，它有3个方法：

Deny (or accept) meters from being registered

Transform meter IDs

Configure distribution statistics for some meter types.

实现MeterFilter就可以加到registry中

```
registry.config()
    .meterFilter(MeterFilter.ignoreTags("too.much.information"))
    .meterFilter(MeterFilter.denyNameStartsWith("jvm"));
```

过滤器按顺序应用，所有的过滤器形成一个过滤器链（chain）

2.4.1. Deny/accept meters

接受或拒绝指标

```
new MeterFilter() {
    @Override
    public MeterFilterReply accept(Meter.Id id) {
       if(id.getName().contains("test")) {
          return MeterFilterReply.DENY;
       }
       return MeterFilterReply.NEUTRAL;
    }
}
```

MeterFilter还提供了许多方便的静态方法用于接受或拒绝指标

2.4.2. Transforming metrics

一个转换过滤器可能是这样的：

```
new MeterFilter() {
    @Override
    public Meter.Id map(Meter.Id id) {
       if(id.getName().startsWith("test")) {
          return id.withName("extra." + id.getName()).withTag("extra.tag", "value");
       }
       return id;
    }
}
```

2.5. Counters（计数器）

Counter接口允许以固定的数值递增，该数值必须为正数。

```
MeterRegistry registry = new SimpleMeterRegistry();

//  写法一
Counter counter = registry.counter("counter");

//  写法二
Counter counter = Counter
    .builder("counter")
    .baseUnit("beans") // optional
    .description("a description of what this counter does") // optional
    .tags("region", "test") // optional
    .register(registry);
```

2.5.1. Function-tracking counters

跟踪单调递增函数的计数器

```
Cache cache = ...; // suppose we have a Guava cache with stats recording on
registry.more().counter("evictions", tags, cache, c -> c.stats().evictionCount()); //
evictionCount()是一个单调递增函数，用于记录缓存被剔除的次数
```

2.6. Gauges

gauge是获取当前值的句柄。典型的例子是，获取集合、map、或运行中的线程数等。

MeterRegistry接口包含了用于构建gauges的方法，用于观察数字值、函数、集合和map。

```
List<String> list = registry.gauge("listGauge", Collections.emptyList(), new
ArrayList<>(), List::size); //监视非数值对象
List<String> list2 = registry.gaugeCollectionSize("listSize2", Tags.empty(), new
ArrayList<>()); //监视集合大小
Map<String, Integer> map = registry.gaugeMapSize("mapGauge", Tags.empty(), new
HashMap<>());
```

还可以手动加减Gauge

```
AtomicInteger n = registry.gauge("numberGauge", new AtomicInteger(0));
n.set(1);
n.set(2);
```

2.7. Timers（计时器）

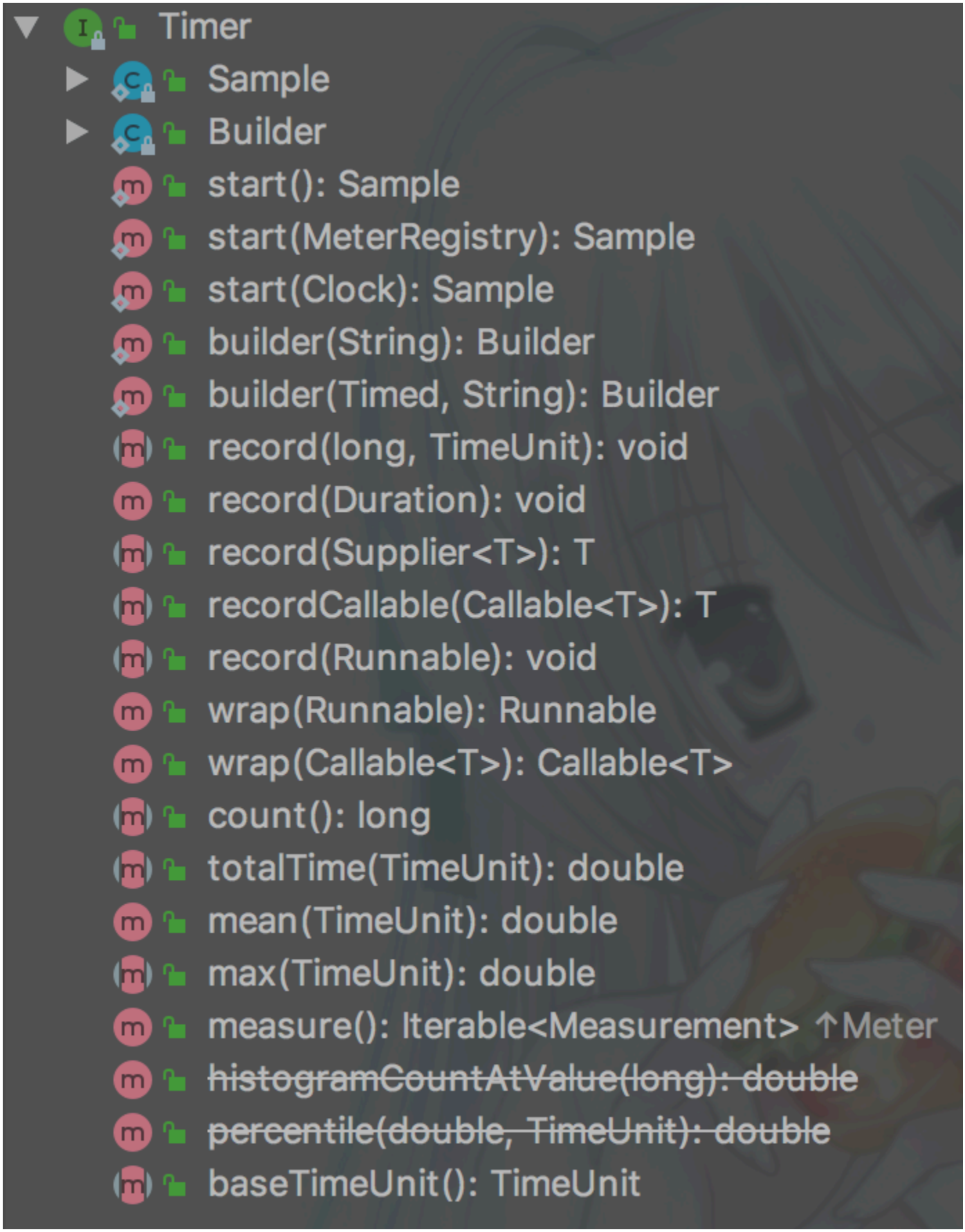
Timer用于测量短时间延迟和此类事件的频率。所有Timer实现至少将总时间和事件次数报告为单独的时间序列。

例如，可以考虑用一个图表来显示一个典型的web服务器的请求延迟情况。服务器可以快速响应许多请求，因此定时器每秒将更新很多次。

```
// 方式一
public interface Timer extends Meter {
    ...
    void record(long amount, TimeUnit unit);
    void record(Duration duration);
    double totalTime(TimeUnit unit);
}

// 方式二
Timer timer = Timer
    .builder("my.timer")
    .description("a description of what this timer does") // optional
    .tags("region", "test") // optional
    .register(registry);
```

查看源代码，一目了然，不一一赘述



2.8. Long task timers

长任务计时器用于跟踪所有正在运行的长时间运行任务的总持续时间和此类任务的数量。

Timer记录的是次数，Long Task Timer记录的是任务时长和任务数

```
// 方式一
@Timed(value = "aws.scrape", longTask = true)
@Scheduled(fixedDelay = 360000)
void scrapeResources() {
    // find instances, volumes, auto-scaling groups, etc...
}

// 方式二
LongTaskTimer scrapeTimer = registry.more().longTaskTimer("scrape");
void scrapeResources() {
    scrapeTimer.record(() => {
        // find instances, volumes, auto-scaling groups, etc...
    });
}
```

2.9. Distribution summaries（分布汇总）

distribution summary用于跟踪分布式的事件。它在结构上类似于计时器，但是记录的值不代表时间单位。例如，记录http服务器上的请求的响应大小。

```
DistributionSummary summary = registry.summary("response.size");
```

2.10. Histograms and percentiles（直方图和百分比）

Timers 和 distribution summaries 支持收集数据来观察它们的百分比。查看百分比有两种主要方法：

Percentile histograms（百分比直方图）： Micrometer将值累积到底层直方图，并将一组预先确定的buckets发送到监控系统。监控系统的查询语言负责从这个直方图中计算百分比。目前，只有Prometheus , Atlas , Wavefront支持基于直方图的百分位数近似值，并且通过histogram_quantile , :percentile , hs()依次表示。

Client-side percentiles（客户端百分比）：Micrometer为每个meter ID（一组name和tag）计算百分位数近似值，并将百分位数值发送到监控系统。

下面是用直方图构建Timer的一个例子：

```
Timer.builder("my.timer")
    .publishPercentiles(0.5, 0.95) // median and 95th percentile
    .publishPercentileHistogram()
    .sla(Duration.ofMillis(100))
    .minimumExpectedValue(Duration.ofMillis(1))
    .maximumExpectedValue(Duration.ofSeconds(10))
```

\3. Micrometer Prometheus

Prometheus是一个内存中的维度时间序列数据库，具有简单的内置UI、定制查询语言和数学操作。Prometheus的设计是基于pull模型进行操作，根据服务发现定期从应用程序实例中抓取指标。

3.1. 安装

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
    <version>${micrometer.version}</version>
</dependency>
```

3.2. 配置

Prometheus希望通过抓取或轮询单个应用程序实例来获得指标。除了创建Prometheus registry之外，还需要向Prometheus的scraper公开一个HTTP端点。在Spring环境中，一个Prometheus actuator endpoint是在Spring Boot Actuator存在的情况下自动配置的。

下面的示例使用JDK的com.sun.net.httpserver.HttpServer来公布scrape端点：

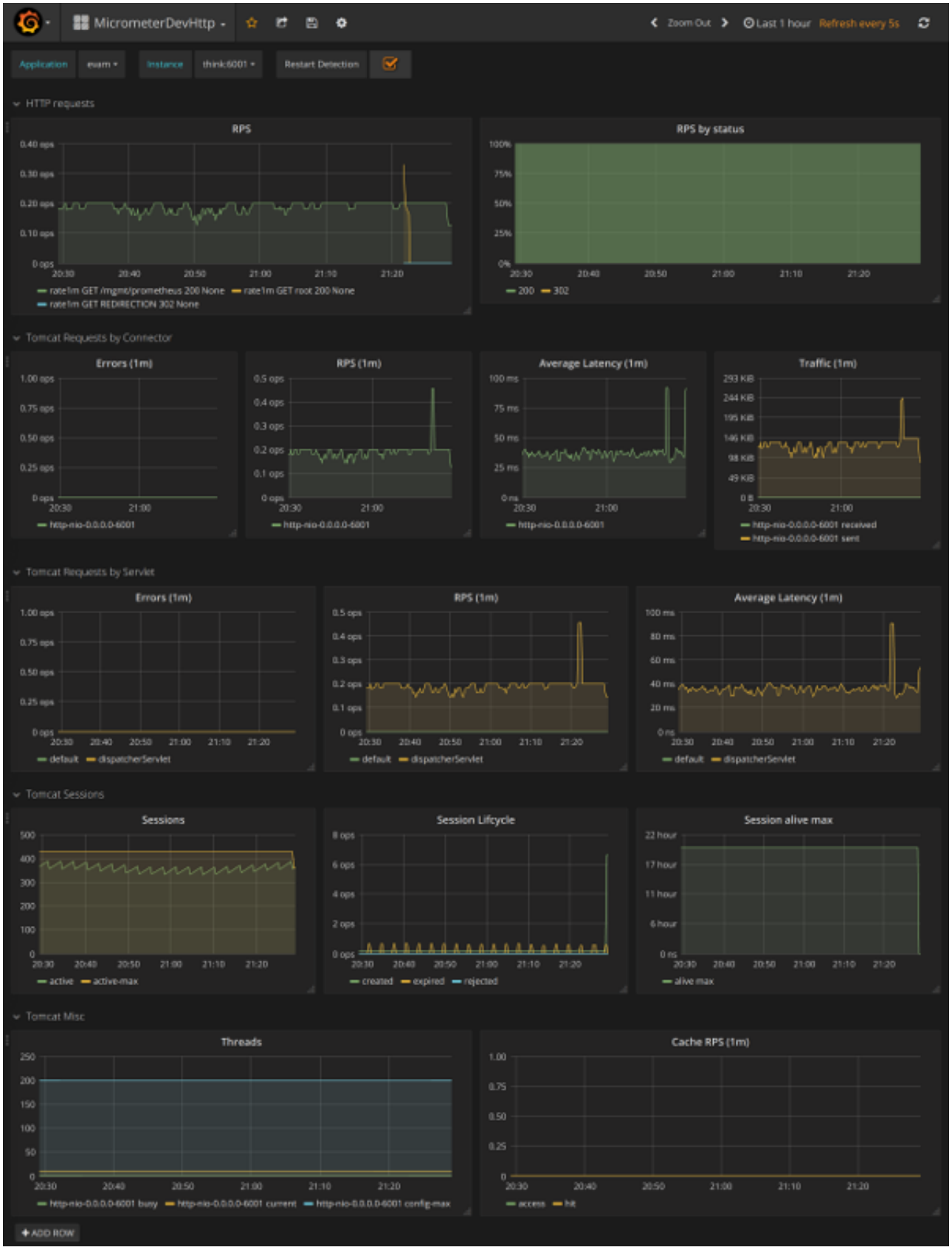
```
PrometheusMeterRegistry prometheusRegistry = new PrometheusMeterRegistry(PrometheusConfig.DEFAULT);

try {
    HttpServer server = HttpServer.create(new InetSocketAddress(8080), 0);
    server.createContext("/prometheus", httpExchange -> {
        String response = prometheusRegistry.scrape();
        httpExchange.sendResponseHeaders(200, response.getBytes().length);
        try (OutputStream os = httpExchange.getResponseBody()) {
            os.write(response.getBytes());
        }
    });

    new Thread(server::start).start();
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

3.3. 图表

Grafana Dashboard



\4. Spring Boot 2.0

Spring Boot Actuator提供依赖管理并自动配置Micrometer

Spring Boot 自动配置一个组合的MeterRegistry，并添加一个registry到这个组合MeterRegistry中。

你可以注册任意数量的MeterRegistryCustomizer来进一步配置registry

```
1 @Bean
2 MeterRegistryCustomizer<MeterRegistry> metricsCommonTags() {
3     return registry -> registry.config().commonTags("region", "us-east-1");
4 }
```

你可以在组件中注入MeterRegistry，并注册指标：

```
 1 @Component
 2 public class SampleBean {
 3
 4     private final Counter counter;
 5
 6     public SampleBean(MeterRegistry registry) {
 7         this.counter = registry.counter("received.messages");
 8     }
 9
10     public void handleMessage(String message) {
11         this.counter.increment();
12         // handle message implementation
13     }
14
15 }
```

Spring Boot为Prometheus提供/actuator/prometheus端点

下面是一个简单的例子，scrape_config添加到prometheus.yml中：

```
1 scrape_configs:
2   - job_name: 'spring'
3     metrics_path: '/actuator/prometheus'
4     static_configs:
5       - targets: ['HOST:PORT']
```

\5. JVM、Cache、OkHttpClient

Micrometer提供了几个用于监视JVM、Cache等的binder。例如：

```
 1 new ClassLoaderMetrics().bindTo(registry);
 2 new JvmMemoryMetrics().bindTo(registry);
 3 new JvmGcMetrics().bindTo(registry);
 4 new ProcessorMetrics().bindTo(registry);
 5 new JvmThreadMetrics().bindTo(registry);
 6
 7 //  通过添加OkHttpMetricsEventListener来收集OkHttpClient指标
```

```
OkHttpClient client = new OkHttpClient.Builder()
    .eventListener(OkHttpMetricsEventListener.builder(registry, "okhttp.requests")
        .tags(Tags.of("foo", "bar"))
        .build())
    .build();
//  为了配置URI mapper，可以用uriMapper()
OkHttpClient client = new OkHttpClient.Builder()
    .eventListener(OkHttpMetricsEventListener.builder(registry, "okhttp.requests")
        .uriMapper(req -> req.url().encodedPath())
        .tags(Tags.of("foo", "bar"))
        .build())
    .build();
```

还有很多内置的Binder，看图：

最后，切记**文档是用来查的**，此处只是一个引子，有个大概印象，等需要用的时候再细查文档即可。