# 28. Metrics 库

Metrics 库实现了一个机制，通过这个机制，producers 可以发布numeric信息，供 consumers 后续查询。  
实际上，生产者通常是其他库或者主进程，而消费者通常是应用程序。

Metrics 本身是一个静态值，并不是有PMD产生的。  
Metric 信息是由推送模型填充的，其中生产者通过调用相关的更新函数来跟新metric库中包含的值。  
消费者通过查询共享内存中的metric数据来获取metric信息。

对于每个mettic，为每个端口ID保留一个单独的值，并且在发布metric时，生产者需要指定哪个端口正在更新。  
此外，还有一个特殊的ID RTE\_METRICS\_GLOBAL， 用于全局统计，不与任何单个设备关联。  
由于metric库是自包含的，因此，对端口号的唯一限制是他们小于 RTE\_MAX\_ETHPORTS，不需要实际端口存在。

## 28.1. 初始化库

在使用库之前，必须通过调用在共享内存中设置mettic存储的 rte\_metrics\_init() 来初始化它。  
这也就是生产者将metric信息发布到哪里以及消费者从哪里查新metric信息。

rte\_metrics\_init(rte\_socket\_id());

这个初始化函数必须在主函数中调用，否则生产者和消费者可能在主程序或次进程中多次调用。？？

## 28.2. 注册metrics

Metrics 必须先注册，这是生产者声明他们将要发布的metric的方式。注册可以单独完成，也可以将一组metric标注为一个组。单独注册使用接口 rte\_metrics\_reg\_name() 实现：

id\_1 = rte\_metrics\_reg\_name("mean\_bits\_in");

id\_2 = rte\_metrics\_reg\_name("mean\_bits\_out");

id\_3 = rte\_metrics\_reg\_name("peak\_bits\_in");

id\_4 = rte\_metrics\_reg\_name("peak\_bits\_out");

一组metric注册使用 rte\_metrics\_reg\_names() 完成：

const char \* const names[] = {

"mean\_bits\_in", "mean\_bits\_out",

"peak\_bits\_in", "peak\_bits\_out",

};

id\_set = rte\_metrics\_reg\_names(&names[0], 4);

如果返回负数，表示注册失败。否则，返回值表示更新metic时使用的 key 值。可以使用 rte\_metrics\_get\_names() 获得将这些key值与metric名称映射起来的映射表。

## 28.3. 更新 metric 值

一旦注册，生产者可以使用 rte\_metrics\_update\_value() 函数更新给定端口的metric。这个函数使用metric注册时返回的key值，也可以使用 rte\_metrics\_get\_names() 查找。

rte\_metrics\_update\_value(port\_id, id\_1, values[0]);rte\_metrics\_update\_value(port\_id, id\_2, values[1]);rte\_metrics\_update\_value(port\_id, id\_3, values[2]);rte\_metrics\_update\_value(port\_id, id\_4, values[3]);

如果metric被注册为一个集合，则可以使用 rte\_metrics\_update\_value() 单独更新他们，或者使用 rte\_metrics\_update\_values() 一起更新：

rte\_metrics\_update\_value(port\_id, id\_set, values[0]);rte\_metrics\_update\_value(port\_id, id\_set + 1, values[1]);rte\_metrics\_update\_value(port\_id, id\_set + 2, values[2]);rte\_metrics\_update\_value(port\_id, id\_set + 3, values[3]);

rte\_metrics\_update\_values(port\_id, id\_set, values, 4);

注意，rte\_metrics\_update\_values() 不能用来更新 multiple sets 的metric，因为不能保证两个集合一个接一个地注册了连续的ID值。

## 28.4. 查询 metrics

消费者可以通过使用返回 struct rte\_metric\_value 数组的接口 rte\_metrics\_get\_values() 来查询metric库。 该数组中的每个条目都包含一个metric值及其关联的key。key值和名称的映射可以使用 rte\_metrics\_get\_names() 函数来获得，该函数返回由key索引的 struct rte\_metric\_name 数组。以下将打印给定端口的所有metric：

void print\_metrics() {

struct rte\_metric\_name \*names;

int len;

len = rte\_metrics\_get\_names(NULL, 0);

if (len < 0) {

printf("Cannot get metrics count\n");

return;

}

if (len == 0) {

printf("No metrics to display (none have been registered)\n");

return;

}

metrics = malloc(sizeof(struct rte\_metric\_value) \* len);

names = malloc(sizeof(struct rte\_metric\_name) \* len);

if (metrics == NULL || names == NULL) {

printf("Cannot allocate memory\n");

free(metrics);

free(names);

return;

}

ret = rte\_metrics\_get\_values(port\_id, metrics, len);

if (ret < 0 || ret > len) {

printf("Cannot get metrics values\n");

free(metrics);

free(names);

return;

}

printf("Metrics for port %i:\n", port\_id);

for (i = 0; i < len; i++)

printf(" %s: %"PRIu64"\n",

names[metrics[i].key].name, metrics[i].value);

free(metrics);

free(names);

}

## 28.5. Bit-rate 统计库

Bit-rate 库计算每个活动端口（即网络设备）的指数加权平均值和峰值比特率。  
这些统计信息通过metric库使用以下名称进行发布：

* mean\_bits\_in: 平均入站比特率
* mean\_bits\_out: 平均出站比特率
* ewma\_bits\_in: 平均入站比特率 (EWMA 平滑)
* ewma\_bits\_out: 平均出站比特率 (EWMA 平滑)
* peak\_bits\_in: 峰值入站比特率
* peak\_bits\_out: 峰值出站比特率

一旦初始化，并以适当的频率计时，可以通过查询metric库来获取metric值。

### 28.5.1. 初始化

在使用库之前，必须通过接口 rte\_stats\_bitrate\_create() 来初始化，这个函数返回一个bit-rate计算对象。由于bit-rate库使用metric来报告计算的统计量，因此bit-rate库需要将计算的统计量与metric库一起注册。这通过辅助函数 rte\_stats\_bitrate\_reg() 完成。

struct rte\_stats\_bitrates \*bitrate\_data;

bitrate\_data = rte\_stats\_bitrate\_create();if (bitrate\_data == NULL)

rte\_exit(EXIT\_FAILURE, "Could not allocate bit-rate data.\n");

rte\_stats\_bitrate\_reg(bitrate\_data);

### 28.5.2. 控制采样速率

由于库通过定期采样来工作，而不是使用内部线程，应用程序必须定期调用 rte\_stats\_bitrate\_calc() 。 这个函数被调用的频率应该是计算统计所需要的预期采样频率。 例如，需要按秒统计，那么应该每秒钟调用一次这个函数。

tics\_datum = rte\_rdtsc();

tics\_per\_1sec = rte\_get\_timer\_hz();

while( 1 ) {

/\* ... \*/

tics\_current = rte\_rdtsc();

if (tics\_current - tics\_datum >= tics\_per\_1sec) {

/\* Periodic bitrate calculation \*/

for (idx\_port = 0; idx\_port < cnt\_ports; idx\_port++)

rte\_stats\_bitrate\_calc(bitrate\_data, idx\_port);

tics\_datum = tics\_current;

}

/\* ... \*/

}

## 28.6. 延迟统计库

延迟统计库计算DPDK应用程序的数据包处理延迟，报告数据包处理所需的最小，平均和最大纳秒，以及处理延迟中的抖动。使用以下名称通过metric库报告这些统计信息：

* min\_latency\_ns: 最小处理延迟（纳秒）
* avg\_latency\_ns: 平均处理延迟（纳秒）
* mac\_latency\_ns: 最大处理延迟（纳秒）
* jitter\_ns: 处理等待时间的变化（纳秒）

一旦初始化并以适当的频率采样，可以通过查询metric库来获得这些统计数据。

### 28.6.1. 初始化

使用库之前，需要调用函数 rte\_latencystats\_init() 进行初始化。

lcoreid\_t latencystats\_lcore\_id = -1;

int ret = rte\_latencystats\_init(1, NULL);if (ret)

rte\_exit(EXIT\_FAILURE, "Could not allocate latency data.\n");

### 28.6.2. 触发统计值更新

需要定期调用 rte\_latencystats\_update() 函数，以便更新延迟统计值信息。

if (latencystats\_lcore\_id == rte\_lcore\_id())

rte\_latencystats\_update();

### 28.6.3. 关闭库

完成之后，需要调用 rte\_latencystats\_uninit() 来关闭延迟统计库。

rte\_latencystats\_uninit();