Ch05. Label

5.1. Definition

프로메테우스에서 Label은 메트릭에 추가되어 메트릭을 취급하는 추가 정보를 제공하는 방식을 취한다. 메트릭에는 여러 개의 Label을 추가할 수 있고, 이런식으로 메트릭을 조회시 Label에 의한 조합으로 조회조건을 구성할 수 있다.

여기서 Label은 시계열을 포함한 Key-Value 이며, Metric이름, 시계열에 의해 고유하게 식별된다.

- 웹 서비스에서 PATH(URI)로 구별되는 HTTP Request에 대한 아래와 같은 메트릭이 있는 경우 처럼, Metric이름에 PATH(URI)정보를 넣어 구성하는 방법을 생각해 볼 수 있다.

```
http_request_login_total
http_request_logout_total
http_request_adduser_total
http_request_comment_total
http_request_view_total
```

- 위 방법은 프로메테우스 내부에서 또는 PromQL로 해당 Metric 데이터에 대해 일괄적인 취급이 어려울 수 있다. 프로메테우스에서는 Label을 이용하여 다음과 같은 표현으로 PATH(URI)에 대해 Label을 이용할 수 있다. 이런식으로 하나의 `http_request_total` 이라는 Metric을 기반으로 PromQL을 사용하여 해당 Metric전체에 대한 집계된 Request 비율, 특정 PATH의 비율 등을 구할 수 있다.

```
http_request_total{path="/login"}
http_request_total{path="/logout"}
http_request_total{path="/adduser"}
http_request_total{path="/comment"}
http_request_total{path="/view"}
```

5.2. Instrumentation Label & Target Label

Label은 `Instrumentation Label`(계측 레이블)과 `Target Label`(대상 레이블), 이 두가지 구별하여 볼 수 있다. PromQL로 작업하는 경우, 이 두가지에 차이는 크게 없으나, 그 성질은 다르다.

- Instrumentation Label: Application이나 Library 내부에서 알 수 있는 내용을 담고 있다. 예를 들어 Application이 수신한 HTTP Request의 타입, 대상이 되는 DB 정보등, 측정되는 내부사항들에 대한 정보를 담게 된다.
- Target Label: 프로메테우스가 테이더를 수집하는 특정 모니터링 대상을 식별하는 내용을 함께 담고 있다. 예를 들어, Application 자체의 유형, 개발이나 운용되는 환경, Application Instance와 관련된 내용 등, 측정 대상의 정보를 담을 수도 있다. 뒤의 “서비스검색” Chapter에서 더 자세히 다룬다.

5.3. Instrumentation

Label 시작해 보기

앞에 있었던 예제(예제3-3)에 Label을 추가하여 사용해 본다. 정의 부분에 `labelnames=['path']` 처럼 Label을 정의 하였고, 이것은 해당 Metric이 `path` 라고 불리우는 하나의 Label을 갖고 있다는 뜻이다.

- 계측과정에서 해당 Metric을 사용하는 경우, 저장할 내용을 parameter로 하는 `“labels”` method를 반드시 호출하여야 한다. (Java에서는 `labels`, Go에서는 `WithLabelValues` method를 사용)

```
$ more 5-1-example.py
import http.server
from prometheus_client import start_http_server, Counter

REQUESTS = Counter('hello_worlds_total',
        'Hello Worlds requested.',
        labelnames=['path'])

class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        REQUESTS.labels(self.path).inc()
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello World")

if __name__ == "__main__":
    start_http_server(8000)
    server = http.server.HTTPServer(('localhost', 8001), MyHandler)
    server.serve_forever()
$
```

- 위의 App을 실행하고, `http://localhost:8001/foo`, `http://localhost:8001/bar` 등으로 접근해 보고, http://localhost:8000/metrics의 내용을 확인해 보면 다음과 같은 결과를 얻을 수 있다.

```
# HELP hello_worlds_total Hello Worlds requested.
# TYPE hello_worlds_total counter
hello_worlds_total{path="/foo"} 3.0
hello_worlds_total{path="/bar"} 2.0
hello_worlds_total{path="/"} 1.0
```

  - Code에서 `REQUESTS.labels(self.path).inc()`는 현재 HTTP Request에서 찾아낼 수 있는 `path값`을 “REQUEST 메트릭 - Counter 타입 - Label ‘path’”에 그 `path값`을 Counter 타입으로 증분 저장한다.
  - 그래서 테스트에서 몇 가지의 PATH(URI)를 지정하여 조회하였을때, 그 PATH(URI)를 값으로 사용하고, 이름은 “path”인 Label에 저장한 것이다.
- Label의 Naming규칙은 Metric Naming규칙과 같으며(:사용만 차이), Metric Name과 달리 Namespace에 소속되지 않으므로, 원하는 이름을 사용하여도 된다. 그러나 다만, Instrumentation Label의 이름에는 Target Label의 성격으로 볼 수 있는 env, cluster, service, team, zone, region 같은 성향의 것들은 피하는 게 좋다.

Metric

메트릭이라는 단어는 다소 모호하며, 문맥과 경우에 따라 다양한 의미로 해석된다. 그 중 중요한 몇 가지 경우를 확인해 본다. 우선 아래와 같은 Metrics가 조회된 경우를 살펴 보자.

```
# HELP latency_seconds Latency in seconds
# TYPE latency_seconds summary
latency_seconds_sum{path="/bar"} 3.0
latency_seconds_count{path="/bar"} 4.0
```

- Metric = Metric Family
  - `latency_seconds` 는 Family로서, Metric 자체의 이름 혹은 관련된 타입을 나타내며, Client Library에서는 해당 Metric에 대한 정의 그 자체 이다.
- Metric = Metric Child
  - `latency_seconds{path="/bar"}` 는 Child로서, Client Library에서 `labels()` 메소드의 반환에 의해 만들어진 값을 나타낸다. Summary타입의 경우에는, Label이름과 함께 `_sum`, `_count` 시계열 값을 포함한다.
- Metric = Time Series Data

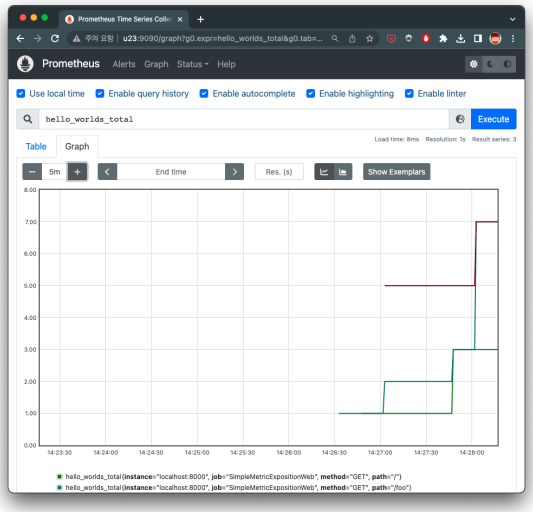
    - `latency_seconds_sum{path="/bar"}`는 Name과 Label로 구분되는 실계열 데이터이다.
- 참고로 Label이 없는 Guage Metric의 경우, Family, Child, Time Series은 모두 동일하다. (Label이 없으므로, 동일한 것으로 취급한다고 해석하는게 나을듯...)

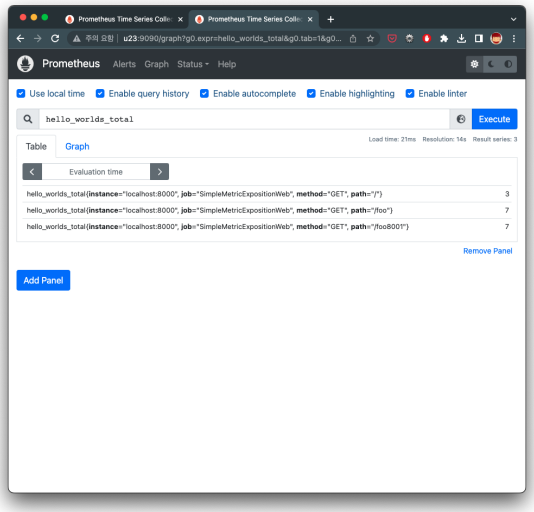
Multiple Label

메트릭을 정의할 때, Label들을 지정하고, Label호출시 같은 순서로 값을 지정할 수 있다. (python list type의 특성을 따라 암묵적으로 그 순서를 Index로서 인정하여 처리)

```
$ more 5-2-mLabel1.py
import http.server
from prometheus_client import start_http_server, Counter

REQUESTS = Counter('hello_worlds_mLabel_total',
        'Hello Worlds with mLabel requestedt.',
        labelnames=['path', 'httpcmd'])

class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        REQUESTS.labels(self.path, self.command).inc()
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello World")

if __name__ == "__main__":
    start_http_server(8000)
    server = http.server.HTTPServer(('localhost', 8001), MyHandler)
    server.serve_forever()
$
```

- 앞에서와 마찬가지로, `http://localhost:8001/foo`, `http://localhost:8001/bar` 등 다영한 URI를 포함하여, curl로 다양하게 접근해 보고, 프로메테우스에서 hello_worlds_total을 조회해 본다.





- (Python과 Go의 경우) Label이름은 Client Library에서 메트릭 정의에 있는 이름과 반드시 일치해야 한다. 실제 예제에서 사용한 Label은 `labelnames=['path', 'httpcmd']` 에서 사용한 `path`, `httpcmd` 이다.
    - 예제 소스에서 이 Label에 데이터를 조작하는 `REQUESTS.labels(self.path, self.command).inc()`를 살펴보면, `self.path`, `self.command` 는 위의 Python App이 포함하고 있는 http.server Library의 설계와 구현에 따른 내용이며, HTTP Request의 Header에 포함되어 있는 `URI`와 `Commad_Method`로 정의되어 있다.

- 직접계측을 수행하는 경우에도 반드시 사전에 해당 Metric의 Label 이름에 대해 알고 있어야 한다. Label을 모르면, 로그기반의 다른 모니터링 도구가 필요할 수도 있다.

Child

Metric의 Label 메소드의 반환 값을 Child(=Label로 구분되어 취급되는 Metric으로 보아도 무방할 듯…)라 부르며, Child는 나중을 위해 별도로 저장할 수도 있다.

Child의 사용에는 몇 가지 기술적 주의 사항이 있다.

- Client Library를 사용하여 Instrumentaion하는 과정에서 매번 Child Label을 조회하는 것은 성능에 좋지 못하다. 대신 Code내부에서 해당 Label을 조회한 다음, 그 데이터를 메모리에 가지고 있으면서, 재활용하는 것이 필요하다.
- Code의 간결성과 일반성을 위한 Code Level에서 해당 Child Label에 대해 Facade(Label명을 인수로 전달하여 활용하는 방식)나 Wrapper(기능 또는 인수를 포함하여, Filter 또는 일반화 Wrap Coding)하지마라. 조회된 결과의 Reference를 사용하라 (결국 Object로 메모리에서 처리하라)
- 아래는 Code 내부에서 Child를 한 번 조회하고, 재사용 하는 간단한 Python 예제이다.

```
from prometheus_client import Counter

FETCHES = Counter('cached_fetches_total',
          'Fetches from cached.',
          labelnames=['cache'])

class MyCache(obejct):
  def __init__(self, name)
    self._fetches = FETCHES.labels(name)
    self._cache = {}

  def fetch(self, item)
    self._fetches.inc()
    return self._cache.get(item)

  def store(self, item, value)
    self._cache[item] = value
```

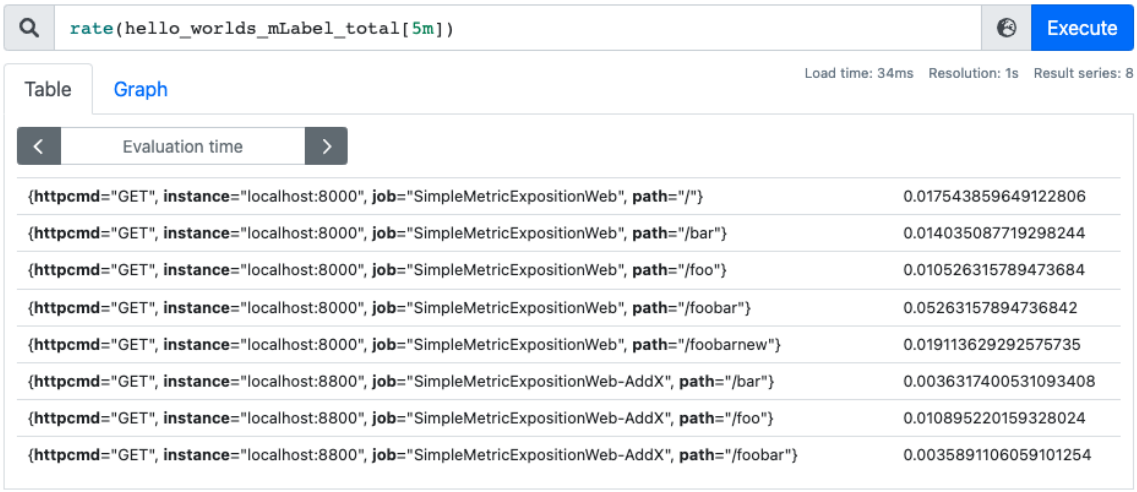
- 또한, Child는 Label이 한 번 호출된 이후 부터 존재할 수 있으므로, 시계열등에서 오해 또는 오류가 없으려면, Code에서 처음에 Child를 호출하여, 초기화 기록이 되도록 하여야 한다. (개발단계에서는 경우에 따라 해당 Child를 제거할 필요가 있을 수 있으며, 보통 Client Library에 제거하는 Method가 포함되어 있다.)

5.4. Aggregation

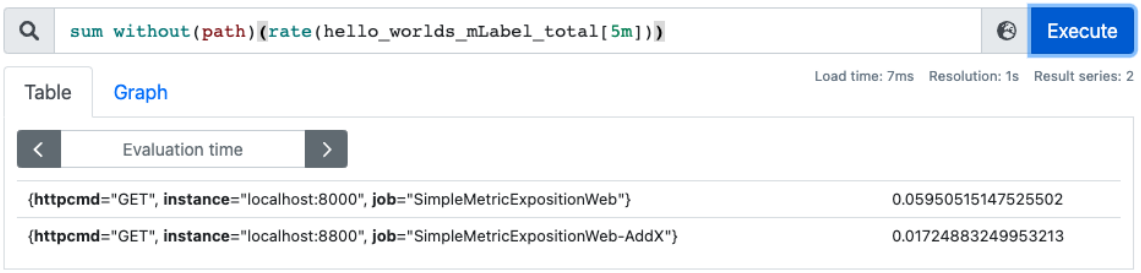
Label의 강점은 집계(Aggregation)에 있다. (상세는 Chap14. Aggregation Operator에서…)

앞에서 사용했던 예제 `5-2-mLabel1.py` 과 `5-2-mLabel2.py` (Listen Port만 다른 같은 내용) 2개의 Daemon을 다시 실행해 보자. 그리고 각각의 Port에 Request를 날려보고, 집계내용을 조회해 보자

- 우선 `rate(hello_worlds_total[5m])` 으로 조회한 출력 결과가 다음과 같다고 가정한다. (출력결과는 프로메테우스 화면의 Table탭 에서 조회한 내용이다. 실제로 실행해 보면, 정수가 아닌 지정된 시간동안의 초당 평균으로 집계된 내용이 출력되는 것에 주의한다. 아마도 예시를 위해 정수로 표현한 듯…)



- 이번에는 `sum without(path)(rate(hello_worlds_mLabel_total[5m]))` 으로 조회한 출력 결과가 다음과 같다고 가정한다. 자세히 보면, `path` 라는 label을 제외하여 무시한 채로 합을 구하고 있다. `path` label을 제외하고 나면, 남은 distinct는 `instance` label에만 있으므로 아래와 같은 결과가 출력 된다.



5.5. Label Pattern

프로메테우스는 시계열 값으로서 측정값은 기본적으로 Float(64bit)만 지원한다. 문자열은 값으로 지원하지 않는데, Label의 값은 문자열이고, 이것을 활용하여, 제한된 조건(몇 가지 가능한 경우만)에서 Label을 활용할 수 있다. (이렇게 하여, 전용의 로그 기반 모니터링 시스템에서 그 흔적을 뒤지거나, 알람을 걸지 않아도 된다.)

결국 요약하면, 문자열 값을 Metric값으로 활용하는 방법으로 볼 수 있다.

enum ~~ enum Guage

예를 들어 어떤 자원(또는 프로세스 등 컴퓨팅에서의 Resource)의 실행상태(Status)에 대해 알고 싶을 경우가 있다고 가정한다. 해당 자원의 상태는 STARTING, RUNNING, STOPPING, TERMINATED의 4가지 경우라고 가정해 보고, "자원들*상태정보"를 관리하는 관점에서 이것을 어떻게 Metric으로 표현할 것인지 고민해 본다.

- 자원이름은 Label로 하는 Guage Metric을 만들고 상태값(0~3)을 넣는다.
  - 상태값을 기준으로 조회(집계)를 할 수 없다는 단점이 있다. 하나의 resource에 대해, 시계열 기준으로 집계와 유사하게 계산은 가능하지만, 여러 resource들에 대해 실계열 기준으로 status중심의 집계는 어렵다.

```
# HELP guage The current state of resources
# TYPE guage current_resource_state
current_resource_state{resource="svcBlaa"} 1
```

- 자원이름과 상태이름을 Label로 하여, 여러 감시 대상에 대해 Guage Metric을 만들과 상태 Flag를 넣는다.

- 상태 Flag는 Exclusive한 “0” or “1”로 규정하고, 해당 상태 Label에만 Flag 값을 “1”로 넣어 표현한다.

```
# HELP guage The current state of resources
# TYPE guage current_resource_state
current_resource_state{resource="svcBlaa", resource_state="STARTING"} 0
current_resource_state{resource="svcBlaa", resource_state="RUNNING"} 1
current_resource_state{resource="svcBlaa", resource_state="STOPPING"} 0
current_resource_state{resource="svcBlaa", resource_state="TERMINATED"} 0
```

- 각 상태에서 얼마나 많은 resource들이 가동되었는지 확인하기 위해서는 다음과 같은 Query로 resource_state 별로 집계가 가능하다.

```
sum without(resource)(current_resource_state)
```

또 다른 방법으로 (Chap 04의 Bridge Exporter에서 언급되었던) 사용자 정의 수집기(Custom Collector)를 사용할 수도 있다. (자세한 것은 Chapter 12: “익스포터 작성하기” 참고)

- (제약 및 주의 사항) 이러한 enum guage를 사용하는 경우, 다른 Label의 수가 많고, 상태의 수가 많으면, 측정된 샘플과 시계열의 양 때문에 성능에 영향을 미칠 수 있다.

info ~~ info Metric (TBD)

- Info Metric은 쿼리의 사용과 결과에 Annotation방식으로 정보를 표현할때 유용하다. 말그대로 계측으로서의 의미보다는 관리 및 정보제공의 목적으로 사용하는 경우에 활용된다. 아래는 info Metric 샘플

```
# HELP python_info Python Platform Information
# TYPE python_info guage
python_info{ implementation="CPython", major="3", minor="5", patchlevel="2",
             version="3.5.2"} 1.0
```

- PromQL의 곱셈연산자(Join)과 group_left변경자(연산자?)를 이용해, info Metric을 다른 기존의 Metric에 Join할 수 있다. info Matric은 “1” 이므로. Join(곱셈)의 결과에서 연산자의 왼쪽에 있던 다른 메트릭들에 대해 필요한 부분만 추가되는 형태를 보이게 된다.
- 자세한 것은 Chapter 15의 Vector Matching에서 알아 봐라

5.6. Timing to use LABEL

(TSDB에서의 Label이 취급되는 방식) 프로메테우스 및 프로메테우스의 TSDB 구조내에서 Label은 양날의 검과 같은 장단점의 특징(처리성능, 저장용량, 쉬운취급 등에서 상호배타적)이 있으므로 Label을 사용하는 Case에 대한 고민이 중요하다.

- Label이 유리한지, 유용하지 않은지 판단하는 힌트는 PromQL에서 메트릭을 사용할때 마다, 해당 Label을 같이 지정해야 하는 경우, 해당 Label은 Metric이름으로 바꾸는 것이 맞다.
- 또 다른 경우로 몇몇 Label로 구분되는 메트릭의 전체 합과 같은 메트릭을 추가로 만드는 경우도 피해야 한다. PromQL은 집계(합)을 계산하는 기능을 가지고 있다.
- Metric Name을 판단하거나 취급하기 위해 Expression을 통한 작업이 필요한 경우라면, 차라리 Label을 사용(남용)하는 것이 나을 수 있다. 특히나, Graph나 Alert에서는 이런 방식의 Matric Name에 대한 Expression을 사용해서는 안된다. 예로서 하드웨어 센서에서 여러 측정값들을 계측,수집하는 경우, 그냥 하나의 Metric에 Label로 넣고, 나중에 필요시 해석하도록 한다. 계측,수집에서 불분명하고, 많은 수의 Label이 되는 것들을 해석해서 처리하지 말라

(TSDB에서 시계열 Data의 적정 규모는?) 프로메테우스는 모니터링 대상 기준 하나의 Metric을 시계열로 저장하는 방식이므로, 프로메테우스가 저장하고 있는 TSDB의 크기는 모니터링 대상 수와 모니터링 Instance당 계측하는 Metric수에 비례한다. 보통 하나의 프로메테우스는 천만개의 시계열 데이터를 처리 할 수 있다고 보면, 다음과 같은 특성을 보이는 것을 이해하자 (Prometheus 1.X기준)

- 천개의 Application Instance를 감시하는 프로메테우스에서 모든 Application Instance에 Metric을 1개 추가 한다면, 프로메테우스에는 천개의 Metirc이 추가되는 것이며, 이는 프로메테우스 처리량의 0.01% 정도로 수용할 만 하다.
- 천개의 모든 Application Instance에 100개의 Metric을 추가해도 이는 프로메테우스 처리량의 1% 정도로 수용할 만 하다.
- 그러나, 하나의 Metric이지만, 12개의 시계열을 갖는 Histogram(10Bucket+sum+count)에 10개의 Label을 같는 경우를 보면, 여기에는 하나의 Metric이지만, 120개의 시계열 데이터가 저장되어야 한다. 하나의 Metric이 1.2%의 저장 및 처리량을 점유하게 된다. (이런 경우가 부담된다면, Summary로의 변경도 고려해야 한다.)

(계측 대상인 시계열 데이터의 적정 Cardinality는?) 이제 Label 또는 Histogram과 관련된 Cardinality에 대해 고려해 본다. 지금은 Cardinality가 10 정도인 시계열 데이터를 사용하는 것은 문제가 안 될 수 있지만, 해당 Cardinality가 계속 증가하는 경우라면, 이야기가 다르다. Metric의 Cardinality가 확장 가능하거나, 추가될 수 있는 Application Instance의 계측과 감시는 위험하다. Cardinality가 10에서 200, 300으로 늘어난다면, TSDB기반의 프로메테우스에서는 성능, 용량의 부담이다.

- 보통의 경우, 임의의 Metric에 대해 Cardinality는 10이하가 권장 된다. 100 정도의 Cardinality를 갖는 적은 수의 Metric도 괜찮다고 보아 줄 수 있다. 그러나, 점점 증가 된다면, 차라리 Log기반의 시스템에 의존하는 것이 더 나을 수 있다.
- 프로메테우스의 설계와 사용에서 Cardinality가 TSDB와 프로메테우스에 미치는 영향에 대해 심각히 고민하여야 한다. 그리고, 비상용 Valve로서 sample_limit의 사용도 고려하라 (Ch.20에서 부하감소 관련 내용 참고)
- 일반적으로 프로메테우스 상위 10개 Matric이 자원의 절반 정도를 사용하는데, 대부분 Label에 의한 Cardinality 때문이다. Label이름을 Metric이름으로 옮기려고 하지마라. Cardinality는 변화가 없으면서 앞서 이야기 한 것처럼, Metric의 취급만 어려워 진다.

+==