



prometheus에 대해서 비기너 관점에서 설명 부탁해.

Prometheus를 처음 접하는 분 입장에서 쉽게 풀어볼게요.

1. Prometheus가 뭐냐?

Prometheus(프로메테우스)는 **모니터링과 알람 시스템**이에요.

쿠버네티스 같은 클라우드 네이티브 환경에서 가장 널리 쓰이는 도구 중 하나죠.

핵심은 “시간에 따라 변하는 수치(메트릭)를 모아서 저장하고, 그걸 분석/시각화/알림에 활용한다”는 거예요.

2. 어떤 데이터를 모으냐?

- **CPU 사용량** (예: 이 서버 CPU가 80% 쓰이고 있네)
- **메모리 사용량**
- **네트워크 트래픽**
- **애플리케이션 요청 수 / 응답 시간**
- **에러 발생 횟수**

즉, 시스템과 애플리케이션의 건강 상태를 숫자로 보는 거죠.

3. 구조 (아주 단순화)

- **Prometheus 서버**: 데이터를 긁어와서 저장하는 뇌 같은 존재
- **Exporter**: 데이터를 제공해 주는 센서 같은 역할 (예: node_exporter는 서버 CPU/메모리 지표 제공)
- **Time-series DB**: 모인 데이터를 시계열로 저장 (시간 흐름에 따른 값)
- **Alertmanager**: 특정 조건(예: CPU 90% 이상 5분 지속)을 만족하면 알람을 보냄
- **Grafana**: Prometheus 데이터를 보기 좋게 차트로 보여주는 화면(대시보드)

4. 특징

- **Pull 방식**: Prometheus가 대상 시스템에 직접 가서 데이터를 긁어옴 (push 아님)
- **PromQL**: Prometheus Query Language라는 자체 쿼리 언어로 데이터 조회
- **오픈소스**: CNCF(Cloud Native Computing Foundation) 프로젝트, 무료

5. 실제 사용 예

- 쿠버네티스 클러스터 모니터링: 파드 CPU/메모리/네트워크 상태 추적
- 웹 서비스 상태 추적: 초당 요청 수, 오류율 모니터링
- 알람 설정: "에러율이 5% 넘으면 슬랙에 메시지 보내라"

정리하면, Prometheus는 **시간에 따라 변하는 데이터를 모아서 지켜보고, 필요하면 알람까지 주는 시스템**이에요.

보통 **Prometheus + Grafana** 조합으로 많이 쓰이고, 쿠버네티스 운영의 기본 도구로 자리 잡았어요.

원하시면 제가 그림으로 구조(서버, 익스포터, 알람, 대시보드 흐름) 그려드릴 수도 있어요. 그림까지 볼까요?



응. 지금 내가 운영환경에서 kubernetes(1.31.1), grafana(8.5.10) 사용중이거든. 그래서 prometheus 대 해서 궁금한게 많아. 기초적인것 부터해서 잘 알고 싶으니깐 , 잘 설명해줘.

좋습니다 😊 지금 환경(Kubernetes 1.31.1 + Grafana 8.5.10)에서 Prometheus를 기초부터 제대로 잡아두면 운영에 큰 도움이 돼요.

천천히, "비기너가 이해할 수 있는 선"부터 "쿠버네티스 운영자 시선"까지 연결해서 풀어드릴게요.

1. Prometheus를 한 줄로 말하면

시계열(time-series) 데이터베이스 + 모니터링 시스템

즉, "시간에 따라 변하는 시스템 지표(CPU, 메모리, 요청 수 등)를 수집하고 저장하고 조회하는" 도구예요.

2. 왜 Prometheus인가?

- **쿠버네티스 친화적**: 파드/노드 같은 객체의 메트릭을 자동으로 발견(service discovery)
- **경량 + 빠름**: 전용 시계열 DB라서 수백만 개 지표를 저장/검색 가능
- **알람 기능 내장**: 임계치 조건 걸어서 Slack, 이메일, PagerDuty 같은 곳으로 알람 보냄
- **오픈소스**: CNCF 족보에 들어간 만큼 안정적으로 커뮤니티 지원

3. 동작 원리 (기본 흐름)

1. 메트릭 노출

각 시스템(쿠버네티스 노드, 애플리케이션)이 **/metrics** HTTP 엔드포인트를 열어줌.

예: `http://<pod-ip>:9100/metrics`

👉 이걸 "Exporter"가 담당 (node_exporter, kube-state-metrics, etc.)

2. Prometheus가 Pull

Prometheus 서버가 주기적으로 **/metrics**를 긁어옴 (기본 15초 간격).

3. 저장 (시계열 DB)

시점별 데이터가 **metric_name{label="value"}** 형식으로 저장됨.

예:

```
arduino

node_cpu_seconds_total{cpu="0",mode="user"} 12345.6
```

4. 조회 (PromQL)

Prometheus 전용 쿼리 언어(PromQL)로 데이터 조회.

👉 예:

- 현재 CPU 사용률:

```
promql

rate(node_cpu_seconds_total{mode="user"}[5m])
```

- 파드 메모리 사용량 top 5:

```
promql

topk(5, container_memory_usage_bytes)
```

5. 시각화/알람

- Grafana에서 Prometheus를 데이터 소스로 연결해 대시보드 만들기
- Alertmanager랑 연계해서 알람 발송

4. 구성 요소 정리

- **Prometheus 서버** → 데이터 긁고 저장
- **Exporter** → 지표 제공 (노드, 앱, 하드웨어)
- **Alertmanager** → 알람 발송기
- **Grafana** → 시각화 대시보드
- **Service Discovery** → 쿠버네티스 클러스터에서 새로 생긴 Pod/Service 자동 인식

5. 쿠버네티스 환경에서 Prometheus

쿠버네티스에서는 직접 설치보다는 **Helm chart**나 **Prometheus Operator**를 많이 써요.

- **Prometheus Operator** = 쿠버네티스 CRD로 Prometheus를 쉽게 운영할 수 있도록 도와줌
- CRD 예시:
 - **ServiceMonitor**: 특정 서비스의 메트릭 수집 정의
 - **PodMonitor**: 파드 단위로 수집 정의
 - **PrometheusRule**: 알람 규칙 정의

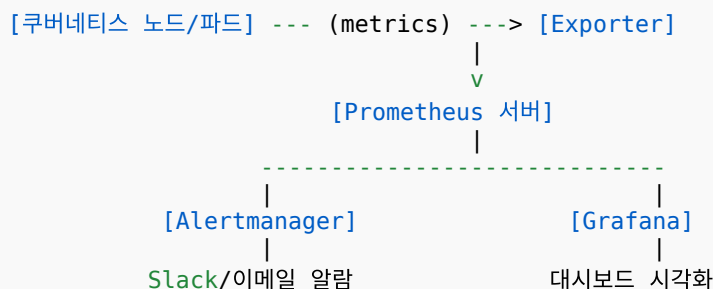
👉 이 덕분에, `kubectl apply -f`만으로 모니터링 대상을 확장할 수 있음.

6. 실제 운영에서 가장 많이 쓰는 Exporter

- **node-exporter**: 노드 CPU, 메모리, 디스크, 네트워크
- **kube-state-metrics**: 쿠버네티스 리소스 상태 (파드 상태, 디플로이먼트 replicas 등)
- **cAdvisor**: 컨테이너 CPU/메모리 사용량
- **애플리케이션 레벨 exporter**: MySQL exporter, Redis exporter, Kafka exporter 등

7. 예시 그림 (흐름 요약)

less



8. 지금 버전대(쿠버네티스 1.31.1, Grafana 8.5.10)에서 주의할 점

- Prometheus Operator 최신 릴리스는 쿠버네티스 1.30~1.31도 지원
- Grafana 8.5.10은 Prometheus 2.x 계열과 호환 OK
- 네임스페이스 구조를 잘 짜서, 운영환경/개발환경 모니터링을 분리하는 게 좋음
- 처음엔 “CPU, 메모리, 디스크” 같은 리소스 기본 모니터링부터 시작하는 게 안정적

👉 정리하면:

Prometheus는 쿠버네티스 운영 환경에서 표준 모니터링 도구예요.

Exporter → **Prometheus** → **Grafana/Alertmanager** 라는 단순한 구조지만,
쿠버네티스와 결합하면 엄청난 확장성을 발휘합니다.

혹시 원하시면 제가 쿠버네티스용 Prometheus Operator 설치 흐름 (Helm chart 예제 포함) 도 차근차근 정리해드릴까요?



pro mq|에 대해서 최대한 많이 알려줘. (다양한 예시와 함께)

좋습니다 👍 Prometheus의 진짜 힘은 **PromQL (Prometheus Query Language)** 에 있습니다.

“PromQL만 잘 다루면 모니터링 반은 끝났다”라는 말이 있을 정도예요.

조금 길어도 되니까, 기초 → 심화 → 실무 예시까지 쭉 정리해드릴게요.

◆ PromQL 완전 기초

PromQL은 시계열(time-series) 데이터 쿼리 언어예요.
모든 지표(metric)는 기본적으로 다음 형태로 저장됩니다:

```
graphql
metric_name{label1="value1", label2="value2", ...} value @timestamp
```

예시:

```
arduino
node_cpu_seconds_total{cpu="0", mode="user"} 12345.67
```

- **metric_name** → node_cpu_seconds_total
- **{label="value"}** → 레이블 (cpu, mode 등)
- **value** → 숫자 값
- **timestamp** → 언제 기록된 값인지

◆ 데이터 타입

PromQL은 4가지 기본 데이터 타입을 지원합니다:

1. **Instant Vector** → 한 시점에서 여러 시계열 값

```
nginx
node_memory_active_bytes
```

👉 지금 이 순간의 메모리 사용량 값들

2. **Range Vector** → 시간 구간 동안의 값 집합

```
css
node_cpu_seconds_total[5m]
```

👉 지난 5분간 CPU 값

3. **Scalar** → 단일 숫자 값

```
1, 42, 3.14
```

4. **String** → 문자열 (거의 안 씀)

◆ 연산자 기초

PromQL은 수학 연산 가능:

- `+`, `-`, `*`, `/`
- `rate(metric[5m]) * 100` → 5분간 증가율 * 100%

비교 연산도 가능:

- `up == 0` → 죽은(target down) 지표 찾기

집계 연산:

- `sum()`, `avg()`, `max()`, `min()`, `count()`, `topk()`, `bottomk()`

◆ 함수들 (핵심)

PromQL에는 함수가 많아요. 자주 쓰는 것들:

1. `rate()`

Counter(누적 증가 지표)의 초당 증가율 계산

```
promql
rate(node_cpu_seconds_total{mode="user"}[5m])
```

2. `irate()`

가장 최근 두 개 샘플로 순간 증가율 계산 (짧은 스파이크 감지용)

3. `increase()`

특정 구간 동안 총 증가량

```
promql
increase(http_requests_total[1h])
```

4. `avg_over_time()` / `min_over_time()` / `max_over_time()`

시간 구간 평균/최소/최대

```
promql
avg_over_time(node_load1[10m])
```

5. `histogram_quantile()`

지연(latency) 같은 히스토그램에서 p95, p99 계산

```
promql
histogram_quantile(0.95, rate(http_request_duration_seconds_bucket[5m]))
```

◆ 레이블 조작

1. `by()` / `without()` → 그룹 집계

```
promql
sum(rate(container_cpu_usage_seconds_total[5m])) by (pod)
```

2. `label_replace()` → 레이블 변환

```
promql
label_replace(up, "namespace", "$1", "job", "(.*)")
```

◆ 실무 예시 모음

1. 노드별 CPU 사용률

```
promql
100 - (avg by (instance) (rate(node_cpu_seconds_total{mode="idle"}[5m])) * 100)
```

2. 파드별 메모리 사용량

```
promql
sum(container_memory_usage_bytes{container!=""}) by (pod, namespace)
```

3. 네임스페이스별 네트워크 트래픽

```
promql
sum(rate(container_network_receive_bytes_total[5m])) by (namespace)
```

4. 파드 재시작 횟수

```
promql
increase(kube_pod_container_status_restarts_total[1h])
```

5. HTTP 요청 수 (RPS)

```
promql
rate(http_requests_total[1m])
```

6. 에러율 (5분 기준)

```
promql

(sum(rate(http_requests_total{status=~"5.."}[5m]))
/
sum(rate(http_requests_total[5m]))) * 100
```

7. 파드 CPU Top 5

```
promql

topk(5, sum(rate(container_cpu_usage_seconds_total[5m])) by (pod))
```

8. 슬로우 요청 (95th percentile)

```
promql

histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket[5m])) by (le))
```

◆ PromQL “패턴” 3가지

- **Counter** → **rate()** / **increase()**
(계속 증가하는 값, 예: 요청 수, 바이트 수)
- **Gauge** → **직접 조회** / **avg_over_time()**
(상시 변하는 값, 예: 메모리 사용량)
- **Histogram** → **histogram_quantile()**
(지연시간, 크기 분포 등 퍼센타일 계산용)

◆ 마무리 요약

- PromQL은 “시간 기반 지표”를 수학처럼 조작할 수 있는 언어
- 핵심은 데이터 타입 (**Instant vs Range Vector**) + 함수(**rate, increase, histogram_quantile** 등)
- 운영환경에서는 CPU/메모리, 네트워크, 에러율, 요청지연이 가장 자주 모니터링됨

👉 원하시면 제가 **PromQL 치트시트 (한 장짜리 정리, PDF 출력용)** 버전도 만들어드릴 수 있어요. 그걸 프린트해서 책상 옆에 붙여두면 진짜 유용합니다. 만들까요?



응, 내가 pdf프린터 만들거니깐, 너는 내용 중심으로 알려줘

좋습니다. 그러면 PDF로 뽑아서 책상 옆에 붙여두고 바로 참고할 수 있는 **PromQL 치트시트** 느낌으로 내용만 정리해드릴게요.

PromQL cheatsheet

1. 기본 구조

```
graphql
```

```
metric_name{label1="value1", label2="value2"} value @timestamp
```

예:

```
arduino
```

```
node_cpu_seconds_total{cpu="0", mode="user"} 12345.67
```

2. 데이터 타입

- **Instant Vector:** 한 시점의 값들 → `node_memory_MemAvailable_bytes`
- **Range Vector:** 일정 기간 값들 → `node_cpu_seconds_total[5m]`
- **Scalar:** 단일 숫자 → `1`, `3.14`
- **String:** 문자열 (거의 안 씀)

3. 주요 연산자

- 산술: `+`, `-`, `*`, `/`
- 비교: `>`, `<`, `==`, `!=`
- 집계: `sum()`, `avg()`, `max()`, `min()`, `count()`, `topk()`, `bottomk()`
- 그룹: `by(label)`, `without(label)`

4. 자주 쓰는 함수

- `rate(v[5m])` → Counter 증가율 (초당)
- `irate(v[5m])` → 순간 증가율
- `increase(v[1h])` → 구간 동안 총 증가량
- `avg_over_time(v[10m])` → 기간 평균
- `max_over_time(v[10m])` → 기간 최대값
- `histogram_quantile(0.95, ...)` → 퍼센타일 (p95, p99)
- `abs()`, `sqrt()`, `round()` → 수학 함수

5. 실무 패턴

- **Counter 지표** (계속 누적): `rate()`, `increase()`
- **Gauge 지표** (현재 상태): 직접 조회, `avg_over_time()`
- **Histogram** (분포형): `histogram_quantile()`

6. 대표 쿼리 예시

시스템 리소스

- CPU 사용률 (%)

```
promql
100 - (avg by (instance) (rate(node_cpu_seconds_total{mode="idle"}[5m])) * 100)
```

- 메모리 사용량 (Pod 단위)

```
promql
sum(container_memory_usage_bytes{container!=""}) by (pod, namespace)
```

- 디스크 사용량

```
promql
node_filesystem_size_bytes - node_filesystem_free_bytes
```

- 네트워크 수신 트래픽 (네임스페이스별)

```
promql
sum(rate(container_network_receive_bytes_total[5m])) by (namespace)
```

쿠버네티스 상태

- 파드 재시작 횟수 (최근 1시간)

```
promql
increase(kube_pod_container_status_restarts_total[1h])
```

- Ready 아닌 파드 찾기

```
promql
kube_pod_status_ready{condition="true"} == 0
```

- Deployment 원하는 Replica 대비 실제 Replica

```
promql
```

```
kube_deployment_status_replicas_available / kube_deployment_spec_replicas
```

애플리케이션 지표

- HTTP 요청 수 (RPS)

```
promql
```

```
rate(http_requests_total[1m])
```

- 에러율 (%)

```
promql
```

```
(sum(rate(http_requests_total{status=~"5.."}[5m]))
/
sum(rate(http_requests_total[5m]))) * 100
```

- 요청 지연 95th percentile

```
promql
```

```
histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket[5m])) by (le))
```

Top/Bottom 분석

- CPU 사용량 Top 5 파드

```
promql
```

```
topk(5, sum(rate(container_cpu_usage_seconds_total[5m])) by (pod))
```

- 메모리 사용량 최저 3개 파드

```
promql
```

```
bottomk(3, sum(container_memory_usage_bytes) by (pod))
```

7. 빠른 팁

- Counter 지표는 꼭 **rate()**나 **increase()** 써야 의미 있음
- Pod/Namespace 단위 집계는 **by(namespace, pod)** 같이 label grouping 활용
- Grafana 패널에서는 **avg()** 대신 **max()**를 쓰는 경우도 많음 (스파이크 감지용)
- Alert 조건은 보통 **5분 이상 지속** 조건을 붙여서 일시적 튀는 값 필터링

👉 이걸 PDF로 뽑으면 바로 참고 가능한 **운영자용 PromQL 포켓 레퍼런스**가 될 겁니다.

원하시면 제가 이걸 **한글+영문 병기 버전**으로 정리해드릴 수도 있어요. (예: “CPU 사용률 (CPU Usage)”) PDF 프린터로 뽑아 쓸 때는 그게 더 보기 편할 수도 있는데, 그렇게 만들어드릴까요?