



쿠버네티스에서 `crictl` 명령어에 대해 설명과 사용법, 유지보수 측면에서 활용법 등에 대해 알려줘

좋은 주제 짚으셨어요.

`crictl`은 **Kubernetes**의 하부 런타임(Container Runtime Interface, CRI)를 직접 다루는 저수준 도구예요. 보통 쿠버네티스의 파드는 `kubelet`이 **containerd**, **CRI-O** 같은 런타임을 통해 실제 컨테이너를 관리하죠. `kubectl`은 클러스터 관점, `crictl`은 노드 관점에서 컨테이너를 들여다보는 툴이라고 보면 됩니다.

◆ 1. 기본 개념

- `kubectl` → API 서버와 통신 (클러스터 레벨)
- `crictl` → `kubelet`이 사용하는 **CRI 소켓**과 직접 통신 (노드 레벨)

즉, 파드가 제대로 안 떴서 `kubectl describe pod`로는 감이 안 올 때, `crictl`로는 진짜 컨테이너 레벨의 상태, 이미지, 로그, 이벤트를 확인할 수 있습니다.

예를 들어,

```
bash

kubectl get pod mypod -n default -o wide
```

이걸로 "Pending"만 보인다면, 노드 안에 들어가서 `crictl ps -a` 하면 컨테이너가 실제로 만들어졌는지, 실패했는지, 이미지 풀 실패인지 바로 알 수 있죠.

◆ 2. 주요 명령어

목적	명령	설명
런타임 연결 확인	<code>crictl info</code>	CRI 런타임과 버전 정보 확인
컨테이너 목록 보기	<code>crictl ps</code> / <code>crictl ps -a</code>	실행 중/모든 컨테이너 조회
파드 상태 보기	<code>crictl pods</code>	파드 샌드박스 상태 조회
로그 확인	<code>crictl logs <container-id></code>	컨테이너 표준 로그 확인
이미지 목록	<code>crictl images</code>	로컬 노드 이미지 캐시 조회
이미지 삭제	<code>crictl rmi <image-id></code>	캐시된 이미지 제거
실행 중지/삭제	<code>crictl stop <container-id></code> / <code>crictl rm <container-id></code>	컨테이너 강제 정리
디버그용 쉘	<code>crictl exec -it <container-id> /bin/sh</code>	<code>kubectl exec</code> 처럼 접근
상태 검사	<code>crictl inspect <container-id></code>	컨테이너 세부 JSON 정보
이벤트 확인	<code>crictl events</code>	런타임 레벨 이벤트 스트림

◆ 3. 설정 방법

기본 설정 파일은 `/etc/crictl.yaml` 입니다.

예시:

```
yaml

runtime-endpoint: unix:///run/containerd/containerd.sock
image-endpoint: unix:///run/containerd/containerd.sock
timeout: 10
debug: false
```

Minikube나 일반 Kubernetes 노드에서는 보통 containerd를 사용하므로 위 경로 그대로 맞습니다.

만약 CRI-O면 `unix:///var/run/crio/crio.sock` 으로 바꿔줘야 해요.

◆ 4. 유지보수/운영 측면 활용 포인트

1. 파드 실패 원인 파악 (Runtime 레벨)

- `kubectl`로는 안 보이는 “이미지 Pull 실패”, “sandbox 생성 실패” 등 런타임 내부 에러를 직접 볼 수 있습니다.
- 특히 “driver name not found”, “image pull timeout”, “namespace cni failure” 류 문제에 유용합니다.

2. 노드 디스크 정리

- `crictl images + crictl rmi`로 오래된 이미지 캐시 제거 가능
- `crictl ps -a + crictl rm`으로 종료된 컨테이너 잔재 제거 가능
→ kubelet이 주기적으로 안 지워줄 때 수동 유지보수로 사용

3. kubelet 디버깅

- kubelet이 “containerd와 통신 오류”가 있을 때,
`crictl info`가 응답하는지 확인하면 런타임쪽 문제인지 kubelet쪽 문제인지 분리 진단이 됩니다.

4. 성능 점검용

- 특정 노드에 컨테이너가 너무 많을 때 `crictl ps | wc -l`로 갯수 체크
- 오래된 로그, 종료 컨테이너 수를 주기적으로 스크립트로 점검 가능

◆ 5. 실무 예시 (유지보수 스크립트)

```
bash

#!/bin/bash
# 오래된 컨테이너 및 이미지 정리

echo "[INFO] List of exited containers:"
crictl ps -a --state Exited

echo "[INFO] Removing exited containers..."
crictl ps -a --state Exited -q | xargs -r crictl rm

echo "[INFO] Removing dangling images..."
crictl images -q | xargs -r crictl rmi
```

→ cron으로 하루 한 번 돌려도 괜찮습니다.

crictl은 kubelet이 멈춘 상태에서도 동작하므로, 노드 복구나 수동정리에 특히 유용합니다.

◆ 6. 한줄 정리

- **kubect**l이 “API 서버”를 본다면,
crictl은 “노드 속의 **containerd** 엔진”을 직접 들여다보는 손전등이에요.
- 클러스터 전체 문제보다 **개별 노드 레벨의 런타임 디버깅, 청소, 확인**에 탁월합니다.

원하신다면 **cric**tl을 활용한 “노드 상태 점검/청소 자동화 스크립트” 도 만들어드릴 수 있습니다.

그걸로 **containerd** 상태를 주기적으로 검증하게끔요.

그 부분도 볼까요?



응. 그래

좋죠. 바로 **운영에 바로 넣을 수 있는 crictl 기반 점검·청소 스크립트** 드릴게요.

컨테이너드(containerd)/CRI-O 모두 대응하고, 드라이런/로그/임계치/나이 필터까지 넣었습니다.

crictl 노드 점검·청소 스크립트

파일명 예: **crictl-node-maint.sh** (루트 권장). 노드에서 실행합니다.

```
bash

#!/usr/bin/env bash
set -euo pipefail

# =====[ 설정 ]=====
# 런타임 소켓 자동 감지 (containerd 우선, 실패 시 CRI-O)
RUNTIME_ENDPOINT_DEFAULTS=(
    "unix:///run/containerd/containerd.sock"
    "unix:///var/run/containerd/containerd.sock"
    "unix:///var/run/crio/crio.sock"
)

# 얼마 이상 디스크가 차면 청소 트리거할지 (GiB)
MIN_FREE_GB_THRESHOLD="${MIN_FREE_GB_THRESHOLD:-20}"

# 나이 기준: N일 지난 종료(Exited) 컨테이너/이미지만 청소
KEEP_DAYS="${KEEP_DAYS:-7}"

# 한 번에 지울 최대 개수 (안전장치)
MAX_REMOVE_CONTAINERS="${MAX_REMOVE_CONTAINERS:-200}"
MAX_REMOVE_IMAGES="${MAX_REMOVE_IMAGES:-200}"

# 드라이런: 1이면 실제 삭제 안 함
DRY_RUN="${DRY_RUN:-1}"

# 로그 파일 (journalctl도 병행 권장)
LOG_FILE="${LOG_FILE:-/var/log/crictl-maint.log}"
```

```
# 필요한 커맨드
REQUIRED_CMDS=(crictl jq awk sed date stat df)

# =====[ 공통 함수 ]=====
log() { echo "[$(date '+%F %T')] $*" | tee -a "$LOG_FILE" ; }
die() { log "ERROR: $*"; exit 1; }

need_cmds() {
    for c in "${REQUIRED_CMDS[@]"; do
        command -v "$c" >/dev/null 2>&1 || die "필요한 명령 누락: $c"
    done
}

detect_runtime_endpoint() {
    if [[ -n "${RUNTIME_ENDPOINT:-}" ]]; then
        export CONTAINER_RUNTIME_ENDPOINT="$RUNTIME_ENDPOINT"
        return
    fi
    for ep in "${RUNTIME_ENDPOINT_DEFAULTS[@]"; do
        if [[ -S "${ep#unix://}" ]]; then
            export CONTAINER_RUNTIME_ENDPOINT="$ep"
            log "runtime-endpoint 자동 감지: $ep"
            return
        fi
    done
    die "CRI runtime endpoint를 찾지 못했습니다. RUNTIME_ENDPOINT 환경변수로 지정하세요."
}

run_crictl() {
    # crictl은 환경변수 CONTAINER_RUNTIME_ENDPOINT 또는 --runtime-endpoint를 봅니다.
    crictl "$@"
}

human_gb() {
    # 바이트→GiB (반올림)
    awk 'BEGIN{gb=ARGV[1]/1024/1024/1024; printf "%.1f", gb}' "$1"
}

age_days() {
    # RFC3339 시간→현재와 차이(일)
    local ts="$1"
    local t_epoch now_epoch
    t_epoch=$(date -d "$ts" +%s 2>/dev/null || date -j -f "%Y-%m-%dT%H:%M:%S%Z" "$ts" +%s
2>/dev/null || echo 0)
    now_epoch=$(date +%s)
    if [[ "$t_epoch" -eq 0 ]]; then echo 9999; return; fi
    awk -v t="$t_epoch" -v n="$now_epoch" 'BEGIN{print int( (n-t)/86400 )}'
}

# =====[ 점검 섹션 ]=====
check_runtime_health() {
    log "=== 런타임 연결 확인 ==="
    run_crictl info | tee -a "$LOG_FILE" >/dev/null || die "crictl info 실패 (runtime 연결 문제)"
}

report_disk_usage() {
    log "=== 디스크 사용량 보고 ==="
    df -h | tee -a "$LOG_FILE" >/dev/null
    local free_gb
    free_gb=$(df --output=avail -B1 / | tail -1)
    free_gb=$(human_gb "$free_gb")
    log "루트 파티션 여유 공간: ${free_gb}GiB (임계치 ${MIN_FREE_GB_THRESHOLD}GiB)"
}

report_top_images() {
    log "=== 큰 이미지 TOP 20 ==="
    # crictl images -v 는 SIZE 컬럼 포함
    run_crictl images -v | awk 'NR==1 || NR>1' | sort -k7 -hr | head -20 | tee -a "$LOG_FILE"
>/dev/null || true
}

```

```

report_pods_containers() {
    log "=== 파드 샌드박스 요약 ==="
    run_crictl pods | tee -a "$LOG_FILE" >/dev/null || true

    log "=== 컨테이너 상태 요약 ==="
    run_crictl ps -a | tee -a "$LOG_FILE" >/dev/null || true
}

report_recent_runtime_events() {
    log "=== 최근 런타임 이벤트(일부) ==="
    # 너무 길면 200줄 제한
    run_crictl events --since 2h | tail -200 | tee -a "$LOG_FILE" >/dev/null || true
}

# =====[ 청소 섹션 ]=====
cleanup_exited_containers() {
    log "=== 종료(Exited) 컨테이너 청소 (>${KEEP_DAYS}일) ==="
    local ids=()
    # 상태 필터는 runtime별로 다를 수 있어 broad 하게 ps -a 후 inspect로 createdAt 판단
    while IFS= read -r id; do
        ids+=("$id")
    done <<(run_crictl ps -a -q)

    local cnt=0
    for id in "${ids[@]}; do
        # 상태 점검
        local st created at_days
        st=$(run_crictl inspect "$id" | jq -r '.status.state // .status.stateString // ""')
        created=$(run_crictl inspect "$id" | jq -r '.status.createdAt // ""')
        at_days=$(age_days "$created")

        # Exited/CONTAINER_EXITED 등만, 나이 필터 적용
        if [[ "$st" =~ [Ee]xited|CONTAINER_EXITED|CONTAINER_EXITING ]] && [[ "$at_days" -ge
"$KEEP_DAYS" ]]; then
            log "삭제 후보: $id (state=$st, age=${at_days}d)"
            if [[ "$DRY_RUN" == "1" ]]; then
                :
            else
                run_crictl rm "$id" || log "경고: 컨테이너 삭제 실패 $id"
            fi
            ((cnt++))
            [[ "$cnt" -ge "$MAX_REMOVE_CONTAINERS" ]] && { log "컨테이너 삭제 상한 도달
(${MAX_REMOVE_CONTAINERS})"; break; }
        fi
    done
    log "컨테이너 삭제 처리 수(예정 포함): $cnt"
}

cleanup_dangling_images() {
    log "=== 이미지 청소 (사용 안되고 ${KEEP_DAYS}일+ 후보) ==="
    # 간단 전략: 컨테이너에 사용 중이지 않은 이미지 중 오래된 것 중심 제거
    # 정확한 lastUsed 추적은 런타임마다 달라서, 보수적으로 dangling/none 및 과도 캐시 위주 제거
    local all_imgs=()
    while IFS= read -r img; do
        all_imgs+=("$img")
    done <<(run_crictl images -q)

    local cnt=0
    for img in "${all_imgs[@]}; do
        # 태그/레포명이 <none> 이거나 digest만 있는 dangling 추정 우선 제거
        local detail
        detail=$(run_crictl inspecti "$img" 2>/dev/null || true)
        [[ -z "$detail" ]] && continue

        # 사용중 여부 대략 판별: 해당 이미지 id로 실행 중 컨테이너가 있으면 skip
        if run_crictl ps -a --quiet --image "$img" | grep -q . ; then
            continue
        fi

        # createdAt 나이 추정 (없으면 스킵)

```

```

local created at_days
created=$(echo "$detail" | jq -r '.image.createdAt // ""')
at_days=$(age_days "$created")

if [[ "$at_days" -ge "$KEEP_DAYS" ]]; then
    log "이미지 삭제 후보: $img (age=${at_days}d)"
    if [[ "$DRY_RUN" == "1" ]]; then
        :
    else
        run_crictl rmi "$img" || log "경고: 이미지 삭제 실패 $img"
    fi
    ((cnt++))
    [[ "$cnt" -ge "$MAX_REMOVE_IMAGES" ]] && { log "이미지 삭제 상한 도달(${MAX_REMOVE_IMAGES})";
break; }
fi
done
log "이미지 삭제 처리 수(예정 포함): $cnt"
}

maybe_trigger_cleanup_by_disk() {
    # 여유 공간이 임계치보다 작으면 강제 청소 트리거
    local free_bytes free_gb
    free_bytes=$(df --output=avail -B1 / | tail -1)
    free_gb=$((human_gb "$free_bytes"))
    if awk -v f="$free_gb" -v th="$MIN_FREE_GB_THRESHOLD" 'BEGIN{exit !(f < th)}'; then
        log "여유공간 ${free_gb}GiB < 임계치 ${MIN_FREE_GB_THRESHOLD}GiB → 청소 수행"
        cleanup_exited_containers
        cleanup_dangling_images
    else
        log "여유공간 충분(${free_gb}GiB) → 청소 스킵 (수동 실행 가능)"
    fi
}

# =====[ 실행 진입점 ]=====
usage() {
    cat <<EOF
사용법: sudo env [옵션들] ./${(basename "$0")} [report|clean|autopilot]

서브커맨드:
    report      런타임/디스크/이미지/컨테이너/이벤트 요약 리포트만 출력
    clean       종료 컨테이너/미사용 이미지 청소 수행 (KEEP_DAYS 기준)
    autopilot   디스크 임계치 판단하여 필요 시 청소 (추천)

환경변수(옵션):
    RUNTIME_ENDPOINT          CRI 소켓 경로 (기본: 자동 탐지)
    MIN_FREE_GB_THRESHOLD=<GiB> autopilot 임계치 (기본: ${MIN_FREE_GB_THRESHOLD})
    KEEP_DAYS=<N>              N일 이상 지난 것만 청소 (기본: ${KEEP_DAYS})
    MAX_REMOVE_CONTAINERS=<N>  컨테이너 삭제 상한 (기본: ${MAX_REMOVE_CONTAINERS})
    MAX_REMOVE_IMAGES=<N>      이미지 삭제 상한 (기본: ${MAX_REMOVE_IMAGES})
    DRY_RUN=0|1                1이면 삭제 미수행 (기본: ${DRY_RUN})
    LOG_FILE=/경로/파일        로그 파일 경로 (기본: ${LOG_FILE})

예시:
    DRY_RUN=1 ./${(basename "$0")} report
    DRY_RUN=0 KEEP_DAYS=14 ./${(basename "$0")} clean
    MIN_FREE_GB_THRESHOLD=30 DRY_RUN=0 ./${(basename "$0")} autopilot
EOF
}

main() {
    local cmd="${1:-autopilot}"

    need_cmds
    detect_runtime_endpoint
    check_runtime_health

    case "$cmd" in
        report)
            report_disk_usage
    esac
}

```

```

    report_top_images
    report_pods_containers
    report_recent_runtime_events
    ;;
clean)
    report_disk_usage
    cleanup_exited_containers
    cleanup_dangling_images
    ;;
autopilot)
    report_disk_usage
    maybe_trigger_cleanup_by_disk
    report_top_images
    ;;
*)
    usage; exit 2;;
esac

log "작업 완료 (cmd=$cmd, dry_run=$DRY_RUN, keep_days=$KEEP_DAYS)"
}

main "$@"

```

실행 예

```

bash

# 리포트만 (삭제 없음)
sudo DRY_RUN=1 ./crictrl-node-maint.sh report

# 실제 청소 (14일 이상 지난 종료 컨테이너/이미지)
sudo DRY_RUN=0 KEEP_DAYS=14 ./crictrl-node-maint.sh clean

# 자동 모드: 여유공간 < 20GiB일 때만 청소
sudo DRY_RUN=0 MIN_FREE_GB_THRESHOLD=20 ./crictrl-node-maint.sh autopilot

```

운영 편의: systemd 타이머로 주기 실행

로그는 `/var/log/crictrl-maint.log` 에 누적. 타이머는 매일 밤 02:30에 “autopilot” 수행 예시.

`/etc/systemd/system/crictrl-maint.service`

```

ini

[Unit]
Description=crictrl node maintenance (autopilot)
After=network-online.target

[Service]
Type=oneshot
Environment=DRY_RUN=0
Environment=MIN_FREE_GB_THRESHOLD=20
Environment=KEEP_DAYS=7
ExecStart=/usr/local/sbin/crictrl-node-maint.sh autopilot
Nice=10
IOSchedulingClass=best-effort

```

`/etc/systemd/system/crictrl-maint.timer`

```
ini
```

```
[Unit]
Description=Run crictl maintenance nightly
```

```
[Timer]
OnCalendar=*-*-* 02:30:00
Persistent=true
RandomizedDelaySec=300
```

```
[Install]
WantedBy=timers.target
```

```
bash
```

```
sudo cp crictl-node-maint.sh /usr/local/sbin/ && sudo chmod +x /usr/local/sbin/crictl-node-maint.sh
sudo systemctl daemon-reload
sudo systemctl enable --now crictl-maint.timer
sudo systemctl status crictl-maint.timer
```

현장 디버깅에 바로 쓰는 crictl 찐막 레시피

- 런타임/네트워킹 초기 확인

```
bash
```

```
crictl info
crictl pods
crictl ps -a
crictl events --since 1h | tail -200
```

- 특정 파드 샌드박스가 안 뜰 때

```
bash
```

```
# 파드 샌드박스 식별
crictl pods | grep <pod-name>
# 샌드박스/컨테이너 JSON 상세
crictl inspectp <pod-sandbox-id> | jq .
crictl inspect <container-id> | jq .
```

- 이미지 풀 이슈 추적

```
bash
```

```
crictl images -v | sort -k7 -hr | head
journalctl -u containerd --since "2 hours ago" | grep -i -E "pull|auth|timeout|dns"
```

- 컨테이너 로그가 kubectl로 안 보일 때

```
bash
```

```
crictl ps -a | grep <pod-name>
crictl logs <container-id> | tail -100
```


- 응급 조치 (정체 구간 풀기)

```
bash

# 종료 컨테이너 대량 적체 제거
crictrl ps -a -q | xargs -r crictrl rm
# 태그 없는 dangling 이미지 제거
crictrl images -q | xargs -r crictrl rmi
```

Minikube/WSL2에서의 팁

- Minikube는 기본적으로 **containerd** 사용이 많습니다. 소켓은 보통 `unix:///run/containerd/containerd.sock` 로 충분해요.
- 호스트(WSL)에서가 아니라 **노드 내부**(minikube ssh)에서 실행해야 런타임에 접근됩니다.
- 스크립트는 “노드 OS” 기준으로 설치/배치하세요.

원하시면 위 스크립트에 **네임스페이스/파드명 패턴 화이트리스트**(예: `kube-system` 제외)나, **삭제 전 로그 스냅샷 저장**(컨테이너 로그를 `/var/log/crictrl-snapshots/<pod>_<ts>.log` 로 보관)도 바로 추가해드릴게요. 필요한 옵션 말해주시면 커스터마이징해서 드릴게요.