Kubernetes 환경 JVM OOM Kill 원인 분석 — UseContainerSupport와 cgroup 메모리 인식 문제

로컬 서버에서 수년간 문제없이 돌아가던 Java 빌드가 K8s 컨테이너로 옮기자마자 OOM Kill로 죽기 시작했다. 메모리를 8GB나 줬는데도 죽고, 같은 빌드를 돌려도 메모리 사용량이 매번 다르다. 실제 운영 환경에서 발생한 이 문제를 추적한 기록이다.

문제 상황

기존에는 RHEL 7.6, 물리 메모리 64GB 서버에서 Ant + JDK 8u181로 빌드했다. 이걸 K8s 클러스터의 컨테이너 POD(memory limit 8GB)로 옮겼더니 빌드 도중 프로세스가 죽었다. 빌드 대상은 프래그먼트 30개짜리 호스트 프로젝트 1개, 소스 파일 7,000~8,000개 규모.

POD memory limit을 8GB로 잡아도 빌드 중 OOM Kill이 발생했고, 같은 프로젝트를 같은 설정으로 빌드해도 메모리 점유율이 매번 달랐다. 그리고 JVM의 OutOfMemoryError 로그가 어디에도 없었다. 로그가 안 남았다는 건, JVM이 에러를 던지기 전에 외부에서 프로세스 자체를 죽였다는 뜻이다.

원인: JVM이 컨테이너 메모리 제한을 모른다

JVM은 -Xmx를 따로 지정하지 않으면 "이 시스템의 물리 메모리"를 기준으로 기본 힙 크기를 잡는다. 보통 물리 메모리의 1/4. 로컬 서버에서는 물리 메모리 64GB니까 기본 힙 ~16GB, 전체 메모리에 여유가 충분하니 아무 문제가 없다.

K8s에서는 사정이 다르다. 각 POD에 cgroup으로 메모리 상한을 부여하는데, JDK 8u181은 이 cgroup 제한을 인식하지 못한다. JVM은 호스트 노드 전체 메모리를 보고 힙을 계산한다.

```
로컬 서버 (64GB):
  JVM이 보는 메모리 = 64GB (실제)
  기본 Xmx = ~16GB
  실제 사용 가능 = 64GB → 문제 없음

K8s POD (8GB limit, JDK 8u181):
  JVM이 보는 메모리 = 64GB (호스트 노드 전체)
  기본 Xmx = ~16GB (POD limit의 2배)
  실제 사용 가능 = 8GB → cgroup OOM Kill
```

JVM이 힙을 8GB 이상 확장하려는 순간 cgroup이 메모리 초과를 감지하고 SIGKILL을 보낸다. SIGKILL은 프로세스에 정리할 기회를 주지 않는다. OutOfMemoryError 출력도, shutdown hook 실행도, heap dump도 없다. 프로세스가 그냥 사라진다. 이게 로그가 안 남는 이유였다.

메모리 사용량이 매번 다른 현상도 여기서 나온다. JVM이 64GB 기준으로 힙을 잡으니 GC 타이밍에 따라 힙 확장 시점이 달라지고, 빌드 태스크의 스레드 스케줄링도 비결정적이라 대형 파일이 동시에 처리되는 타이밍이 빌드마다 다르다. POD가 다른 워커 노드에 배치되면 호스트 물리 메모리 자체가 달라져서 JVM의 기본 힙 계산도 바뀐다.

JDK 버전별 컨테이너 인식

JDK에는 이후 버전에서 컨테이너 인식 기능이 추가되었다.

- 8u131 ~ 8u181: -XX:+UseCGroupMemoryLimitForHeap 플래그가 있지만 실험적이고 기본 비활성. CPU는 인식 못 한다.
- 8u191+: -XX:+UseContainerSupport가 도입되어 cgroup v1에서 메모리와 CPU를 인식한다.
- 8u372+: cgroup v2도 지원. (Kubernetes 공식 문서 기준)
- JDK 11+: 컨테이너 인식이 완전히 내장.

한 가지 빠뜨리기 쉬운 점이 있는데, 8u191의 UseContainerSupport는 cgroup v1에서만 동작한다. RHEL 8은 cgroup v1이 기본이라 운영 환경에서는 보통 문제가 안 되지만, Docker Desktop(macOS/Windows)이나 최신 Linux 배포판은 cgroup v2를 쓰므로 로컬에서 테스트할 때 결과가 다를 수 있다.

해결

-Xmx 명시 (모든 환경에서 동작)

JDK 버전이나 cgroup 버전에 관계없이 -Xmx를 명시하면 JVM의 자동 계산을 덮어쓴다.

```
# POD memory limit = 8GB 기준
java \
  -Xmx4g \
  -Xms2g \
  -XX:MaxMetaspaceSize=512m \
  -jar your-build-tool.jar
```

-Xmx는 POD limit의 50~60%가 적당하다. JVM은 힙 외에 Metaspace, 스레드 스택, Native 메모리, GC 오버헤드를 별도로 쓰기 때문이다.

영역	할당	비고
JVM Heap (-Xmx)	4,096 MB (50%)	애플리케이션 작업 영역
Metaspace	512 MB (6%)	클래스 로딩
Thread stacks	~64 MB (1%)	스레드 수에 비례
Native / OS	~3,500 MB (43%)	커널 버퍼, cgroup 오버헤드
합계	≤ 8,192 MB	POD limit 이내

-Xmx를 POD limit과 똑같이 잡는 경우가 있는데(예: -Xmx8g), 이러면 힙 외 메모리가 limit을 넘어서 결국 OOM Kill이 다시 발생한다.

Ant 빌드라면 ANT_OPTS로 전달한다.

```
export ANT_OPTS="-Xmx4g -Xms2g -XX:MaxMetaspaceSize=512m"
ant build
```

CI 파이프라인에서도 마찬가지다.

```
# .gitlab-ci.yml
build:
  script:
    - export ANT_OPTS="-Xmx4g -Xms2g -XX:MaxMetaspaceSize=512m"
    - ant build
```

JDK 업그레이드 (권장)

8u191 이상으로 올리면 UseContainerSupport가 활성화되어 JVM이 컨테이너 메모리를 자동 인식한다. 비율 기반 설정이 가능해진다.

```
java \
  -XX:MaxRAMPercentage=50.0 \
  -XX:MaxMetaspaceSize=512m \
  -jar your-build-tool.jar
```

MaxRAMPercentage=50.0은 컨테이너 메모리의 50%를 힙 상한으로 쓰겠다는 뜻이다. POD memory limit을 나중에 바꿔도 빌드 스크립트를 건드릴 필요가 없다.

8u181에서 8u191 전용 옵션을 쓰면 JVM이 기동 안 된다

8u181 환경에서 -XX:+UseContainerSupport나 -XX:MaxRAMPercentage를 넣으면 Unrecognized VM option 에러가 나고 JVM이 시작조차 하지 않는다. 여러 환경에 JDK 버전이 섞여 있으면 -Xmx가 안전하다.

멀티스레드와 메모리 증폭

빌드 도구가 멀티스레드로 동작하면 스레드 수도 메모리에 영향을 준다. 예를 들어 JS minify를 Google Closure Compiler로 처리하는 빌드에서는 파일마다 AST를 메모리에 올린다. 12개 스레드가 동시에 대형 파일을 처리하면 피크 메모리가 확 올라간다.

8u181에서는 Runtime.getRuntime().availableProcessors()가 호스트 노드 전체 코어 수를 반환해서, 빌드 도구가 이 값으로 스레드 수를 정하면 의도보다 많은 스레드가 뜬다. 빌드 도구에 스레드 수 제한 옵션이 있으면 POD 환경에서는 낮춰 두는 게 좋다.

테스트

Docker 컨테이너에서 실제로 재현되는지 확인했다.

환경

테스트 환경인 Docker Desktop은 cgroup v2로 동작한다. 8u191의 UseContainerSupport는 cgroup v1만 지원하므로 JVM 자동 인식 검증에는 cgroup v2를 지원하는 8u372를 썼다. 실 운영 환경(RHEL 8)은 cgroup v1이 기본이라 8u191에서도 같은 동작을 할 것으로 본다.

- Docker Desktop 27.3.1 (macOS, aarch64), cgroup v2
- openjdk:8u181-jdk-slim, eclipse-temurin:8u372-b07-jdk

시뮬레이터

멀티스레드 빌드의 메모리 패턴을 재현하는 MemorySimulator를 만들었다. 빌드 엔진의 스레드 수 결정 로직을 그대로 가져오고, 각 스레드에서 소스 문자열 + AST + 결과 문자열을 동시에 잡아 minify의 메모리 패턴을 흉내냈다.

```
// 빌드 엔진과 동일한 스레드 수 결정 로직
int cpuCount = (Math.max(1, Runtime.runtime.availableProcessors() - 1) * 2);
int threadCount = Math.min(taskCount, cpuCount);
threadCount = Math.min(GLOBAL_MAX_THREAD_COUNT, threadCount);  // 상한 12
```

인자로 스레드 수, 태스크 수, 태스크당 할당량(MB)을 받는다.

```
FROM openjdk:8u181-jdk-slim
WORKDIR /build
COPY src/MemorySimulator.java .
RUN javac MemorySimulator.java
ENTRYPOINT ["java", "-cp", "/build"]
```

실행

```
docker build -f Dockerfile.8u181 -t oom-test:8u181 .
docker build -f Dockerfile.8u372 -t oom-test:8u372 .

# TEST 1: 8u181 + Xmx 미설정 + 256MB → OOM 재현
docker run --rm --memory=256m --memory-swap=256m \
  oom-test:8u181 MemorySimulator 12 100 15

# TEST 2: 8u181 + -Xmx128m → OOM 방지
docker run --rm --memory=256m --memory-swap=256m \
```

```
  oom-test:8u181 -Xmx128m -Xms64m -XX:MaxMetaspaceSize=32m \
  MemorySimulator 2 30 3

# TEST 3a/3b: 스레드 2개 vs 12개
docker run --rm --memory=512m --memory-swap=512m \
  oom-test:8u181 -Xmx256m -Xms128m -XX:MaxMetaspaceSize=64m \
  MemorySimulator 2 50 5    # 3a

docker run --rm --memory=512m --memory-swap=512m \
  oom-test:8u181 -Xmx256m -Xms128m -XX:MaxMetaspaceSize=64m \
  MemorySimulator 12 50 5   # 3b

# TEST 4: 8u372 + Xmx 미설정 → 자동 인식
docker run --rm --memory=512m --memory-swap=512m \
  oom-test:8u372 MemorySimulator 2 50 5

# TEST 5: 8u372 + MaxRAMPercentage=50
docker run --rm --memory=512m --memory-swap=512m \
  oom-test:8u372 -XX:MaxRAMPercentage=50.0 -XX:MaxMetaspaceSize=64m \
  MemorySimulator 2 50 5
```

결과

TEST 1 — 8u181, Xmx 미설정, 컨테이너 256MB

```
java.version  : 1.8.0_181
max heap      : 1742 MB        ← 컨테이너는 256MB인데 호스트 기준으로 잡혔다
processors    : 8
cgroup v2 lim : 268435456

[start] launching 12 worker threads

exit code 137 (OOM Killed by cgroup)
```

max heap 1742MB. 컨테이너의 7배. cgroup이 SIGKILL을 보냈고 JVM 로그는 한 줄도 없다.

TEST 2 — 8u181, -Xmx128m, 동일 컨테이너

```
java.version  : 1.8.0_181
max heap      : 114 MB

[progress] 30/30 completed, heap=58MB

exit code 0 (정상 완료)
```

같은 8u181, 같은 256MB 컨테이너. -Xmx128m 하나 추가한 것만 다르다.

TEST 3 — 스레드 수 비교

	스레드	피크 heap	소요 시간
3a	2	112MB	1,325ms
3b	12	135MB	281ms

-Xmx가 같아도 스레드 수에 따라 피크 메모리가 달라진다. 시뮬레이터에서는 20% 차이인데, 실제 빌드에서 대형 파일의 AST가 동시에 올라가면 차이가 더 벌어진다.

TEST 4 — 8u372, Xmx 미설정, 512MB

```
java.version  : 1.8.0_372
max heap      : 123 MB         ← 512MB / 4, 컨테이너 기준 산정
cgroup v2 lim : 536870912

[progress] 50/50 completed, heap=35MB

exit code 0 (정상 완료)
```

Xmx 미설정인데 max heap이 123MB. TEST 1에서 같은 조건으로 1742MB였던 것과 비교하면 JDK 버전 하나 차이다.

TEST 5 — 8u372, MaxRAMPercentage=50, 512MB

```
java.version  : 1.8.0_372
max heap      : 247 MB         ← 512 * 50%

exit code 0 (정상 완료)
```

비율 기반 설정이 컨테이너 메모리 기준으로 동작한다.

요약

테스트	JDK	컨테이너	-Xmx	max heap	결과
1	8u181	256MB	미설정	1,742MB	exit 137 (OOM Kill)
2	8u181	256MB	128m	114MB	exit 0
3a	8u181	512MB	256m, 2스레드	228MB	exit 0 (피크 112MB)
3b	8u181	512MB	256m, 12스레드	228MB	exit 0 (피크 135MB)
4	8u372	512MB	미설정	123MB	exit 0
5	8u372	512MB	RAMPct=50	247MB	exit 0

마무리

Java 프로세스가 K8s에서 로그 없이 죽으면, 메모리 누수나 JVM 버그를 의심하기 전에 먼저 볼 것이 있다.

- JDK 버전이 컨테이너 메모리를 인식하는가 (8u191+, cgroup v2라면 8u372+)
- -Xmx가 POD memory limit의 50~60%로 명시되어 있는가
- 멀티스레드 빌드라면 스레드 수가 적절한가

-Xmx를 명시하면 JDK 버전이나 cgroup 버전에 관계없이 동작한다.

참고

- https://github.com/GoogleContainerTools/distroless/issues/324 — JDK 8u181 vs 8u191 컨테이너 인식 차이
- https://www.javaspring.net/blog/do-java-flags-xms-and-xmx-overwrite-flag-xx-usecgroupmemorylimitforheap/ — -Xmx vs cgroup 플래그 우선순위
- https://kubernetes.io/blog/2022/08/31/cgroupv2-ga-1-25/ — K8s cgroup v2 GA, JDK 버전별 지원
- https://access.redhat.com/articles/3735611 — RHEL 8 cgroup v1 기본값