

# 27. 고급 주제

## 클래스 로더 구조

### 1. 클래스 로더란?

클래스 로더(ClassLoader)는 `.class` 파일을 읽어들이며 메모리에 클래스 객체(Class<?> 객체)로 로딩하는 JVM 구성 요소다.

JVM은 프로그램 실행 중에 필요한 클래스를 동적으로 로딩하며, 이 과정을 클래스 로딩(Class Loading)이라고 한다.

### 2. 클래스 로딩 과정

클래스 로딩은 크게 다음 5단계로 구성된다:

#### 1. 로딩(Loading)

`.class` 파일을 바이너리로 읽어 메모리에 로드

#### 2. 링크(Linking)

- 검증(Verification): 유효한 클래스인지 검사
- 준비(Preparation): static 필드 메모리 할당 및 기본값 설정
- 해결(Resolution): 참조된 클래스들을 로딩

#### 3. 초기화(Initialization)

static 블록과 static 필드에 값 할당

#### 4. 사용(Using)

인스턴스 생성, 메서드 호출 등

#### 5. 언로딩(Unloading)

참조되지 않는 클래스는 GC에 의해 제거 (주로 사용자 정의 로더에서만)

### 3. JVM의 기본 클래스 로더 계층 구조

JVM에는 부트스트랩 → 확장 → 애플리케이션 → 사용자 정의 로더로 이어지는 트리 구조가 존재한다.

```
1 [ Bootstrap ClassLoader ] (C/C++로 구현된 JVM 내부)
2     ↓
3 [ Platform ClassLoader (JDK 9+) / Extension ClassLoader (JDK 8 이하) ]
4     ↓
5 [ Application ClassLoader ]
6     ↓
7 [ Custom ClassLoader (직접 구현) ]
```

### ◆ 3.1 Bootstrap ClassLoader

- 가장 최상위 클래스 로더
  - JVM 내부(C 코드)에 구현되어 있음 (Java로 접근 불가)
  - `JAVA_HOME/lib`에 있는 `rt.jar`, `java.base` 모듈 등 핵심 클래스를 로딩
  - 예: `java.lang.String`, `java.util.*`
- 

### ◆ 3.2 Platform or Extension ClassLoader

- JDK 9+: `PlatformClassLoader`
  - JDK 8 이하: `ExtClassLoader`
  - `JAVA_HOME/lib/ext` 또는 모듈 시스템 기반 클래스 로딩 수행
- 

### ◆ 3.3 Application ClassLoader

- 클래스패스에 포함된 `.jar`, `.class` 파일을 로딩
  - 보통 우리가 작성한 일반 애플리케이션 클래스를 로딩
  - `ClassLoader.getSystemClassLoader()` 로 접근 가능
- 

### ◆ 3.4 사용자 정의 클래스 로더

- `ClassLoader`를 상속해 직접 구현
  - 보안, 클래스 격리, 리플렉션 기반 프레임워크 등에서 활용됨
  - 예: Spring, Tomcat, OSGi 등은 커스텀 클래스 로더 사용
- 

## 4. 클래스 로더의 위임 모델 (Parent Delegation Model)

클래스를 로딩할 때 다음 순서로 시도함:

1 | 자기 자신 → 부모에게 위임 → 부모 로더 → Bootstrap 로더

✅ 부모가 로딩에 실패했을 때만 자신이 직접 로딩 시도  
이 방식 덕분에 핵심 API가 오염되지 않음

---

## 5. ClassLoader 주요 메서드

```
1  ClassLoader loader = ClassLoader.getSystemClassLoader();
2
3  // 클래스 로드
4  Class<?> clazz = loader.loadClass("com.example.MyClass");
5
6  // 클래스 로더 확인
7  System.out.println(clazz.getClassLoader()); // ApplicationClassLoader
```

### 주요 API

| 메서드                                 | 설명                                   |
|-------------------------------------|--------------------------------------|
| <code>loadClass(String name)</code> | 클래스 로딩 시도                            |
| <code>findClass(String name)</code> | 직접 클래스 찾기 (오버라이드용)                   |
| <code>defineClass(...)</code>       | byte[] → Class 객체 생성 (직접 로더 구현 시 사용) |

## 6. 사용자 정의 ClassLoader 예시

```
1  public class MyClassLoader extends ClassLoader {
2      @Override
3      protected Class<?> findClass(String name) throws ClassNotFoundException {
4          byte[] b = loadClassDataFromSomewhere(name);
5          return defineClass(name, b, 0, b.length);
6      }
7
8      private byte[] loadClassDataFromSomewhere(String name) {
9          // 파일 읽기, 네트워크 다운로드 등
10     }
11 }
```

! 사용자 정의 로더는 보안, 격리, AOT 디코딩 등에서 유용함

## 7. 클래스 로더 격리와 충돌

클래스 로더가 다르면 같은 클래스라도 서로 다른 것으로 간주된다.

```
1  ClassLoader loader1 = new MyClassLoader();
2  ClassLoader loader2 = new MyClassLoader();
3
4  Class<?> c1 = loader1.loadClass("com.example.Foo");
5  Class<?> c2 = loader2.loadClass("com.example.Foo");
6
7  System.out.println(c1 == c2); // false
```

## 8. Spring과 클래스 로더

- Spring Framework는 **ClassPath scanning**을 위해 `ClassLoader`를 사용
- Spring Boot에서는 **DevTools**로 인해 **클래스 리로드**를 위한 별도 클래스 로더가 사용됨
- `Thread.currentThread().getContextClassLoader()`는 종종 스프링/서드파티 프레임워크에서 중요한 역할을 함

## 9. 클래스 로더 관련 디버깅 팁

```
1  class<?> clazz = Class.forName("com.example.MyClass");
2  System.out.println(clazz.getClassLoader());
3
4  ClassLoader loader = clazz.getClassLoader();
5  while (loader != null) {
6      System.out.println(loader);
7      loader = loader.getParent();
8  }
```

## 10. 정리: 클래스 로더의 역할

| 역할                     | 설명                              |
|------------------------|---------------------------------|
| <code>.class</code> 로딩 | 파일 시스템, JAR, 네트워크 등             |
| 보안                     | 특정 로더 범위 내 클래스만 접근 허용           |
| 격리                     | 웹 컨테이너 간 격리, 플러그인 시스템           |
| 커스터마이징                 | 동적 바이트코드 삽입 (예: ByteBuddy, ASM) |


## 11. 실전 활용 예시

- **Tomcat**: 웹 앱마다 별도 `ClassLoader` 운영
- **Spring Boot DevTools**: 자동 리로드용 `ClassLoader` 격리
- **OSGi**: 완전한 모듈화 지원 위한 `ClassLoader` 격리 모델

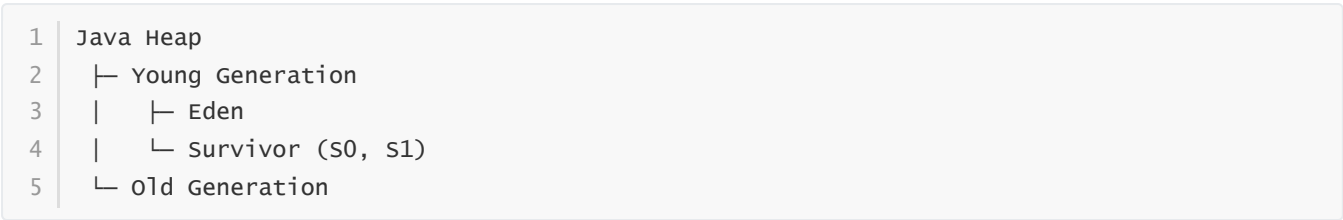
# Garbage Collection 알고리즘

## 1. Garbage Collection(GC)이란?

Garbage Collection은 더 이상 사용되지 않는 객체를 자동으로 탐지하고 제거함으로써 메모리를 회수하는 **JVM의 자동 메모리 관리 시스템**이다.

 목표: 개발자가 `free()` 나 `delete` 같은 수동 메모리 해제를 하지 않아도 **메모리 누수 없이 안전한 프로그래밍**을 가능하게 함

## 2. GC가 관리하는 메모리 영역: Java Heap



- **Young Generation (신생 영역):** 새로 생성된 객체가 배치됨
- **Old Generation (노년 영역):** 오래 살아남은 객체가 이주됨
- **Permanent Generation / Metaspace:** 클래스 메타 정보 (Java 8부터 Metaspace로 이동)

## 3. GC 주요 개념

| 개념              | 설명   |
|-----------------|--|
| Minor GC        | Young 영역에서 수행  |
| Major/Full GC   | Old 영역(또는 전체 Heap 포함)을 대상으로 수행                         |
| Stop-The-World  | GC 수행 시 모든 애플리케이션 스레드 일시 정지                            |
| GC Root         | 탐색의 기준이 되는 객체 ( Thread , Static , JNI , System class ) |
| Reachability 분석 | 객체 참조 그래프를 따라 살아 있는 객체 탐색                              |

## 4. Java의 주요 GC 알고리즘

### ◆ 4.1 Serial GC

- 단일 스레드 기반, 모든 GC 작업을 하나의 스레드로 수행
- 멀티코어 활용 못함 → 작은 메모리의 클라이언트 환경에 적합

```
1 | -XX:+UseSerialGC
```

### ◆ 4.2 Parallel GC (또는 Throughput Collector)

- 멀티스레드 기반 GC
- Young GC 와 Old GC 모두 병렬 처리
- Throughput(전체 처리량)을 높이는 데 중점

```
1 | -XX:+UseParallelGC
```

### ◆ 4.3 CMS (Concurrent Mark Sweep) GC ❌ [Deprecated]

- Old 영역에서 Stop-The-World을 최소화하려는 목적
- 마크(Mark) → Sweep 단계가 **동시 실행**
- 단점: Fragmentation 발생 가능, Java 9부터 deprecated

1 | -XX:+UseConcMarkSweepGC

### ◆ 4.4 G1 GC (Garbage First)

- Java 9 이후 기본 GC
- **Heap을 Region 단위로 분할** (Young/Old를 Region 단위로 혼합 관리)
- **Concurrent GC + Stop-The-World GC의 하이브리드**
- 큰 Heap에서도 예측 가능한 GC 시간 제공

1 | -XX:+UseG1GC

| 특징                | 설명                               |
|-------------------|----------------------------------|
| Region 기반         | Heap 전체를 고정된 크기의 Region으로 나눔     |
| Evacuation        | GC 대상 Region의 객체를 다른 Region으로 복사 |
| Pause Time Target | -XX:MaxGCPauseMillis=200 등 설정 가능 |

### ◆ 4.5 ZGC (Java 11+)

- **초저 지연(Low Latency) GC**
- 대부분의 GC 작업이 **스레드와 병렬로 수행**, Stop-The-World는 **몇 밀리초 미만**

1 | -XX:+UseZGC

| 특징                        | 설명          |
|---------------------------|-------------|
| 멀티 스레드 병렬 마킹              | 매우 빠른 마킹    |
| GC 작업 대부분이 <b>non-STW</b> | 매우 짧은 멈춤    |
| 컬러 포인터(Color Pointer)     | 객체 참조 상태 추적 |

## ◆ 4.6 Shenandoah GC (Red Hat, Java 12+)

- ZGC와 비슷한 철학 → **pause time** 단축
- **Region-based, concurrent compacting GC**

```
1 | -XX:+UseShenandoahGC
```

## 5. GC 알고리즘 비교

| GC         | 멀티스레드 | 지연 시간 | Throughput | 메모리 크기 | 비고                |
|------------|-------|-------|------------|--------|-------------------|
| Serial     | ✗     | 높음    | 낮음         | 소형     | 단일 스레드            |
| Parallel   | ✓     | 중간    | 높음         | 중형~대형  | 기본 GC (Java 8)    |
| CMS        | ✓     | 낮음    | 중간         | 중형     | 조각 발생, Deprecated |
| G1         | ✓     | 낮음    | 높음         | 중~대형   | 기본 GC (Java 9~17) |
| ZGC        | ✓     | 매우 낮음 | 높음         | 대형     | STW < 2ms         |
| Shenandoah | ✓     | 매우 낮음 | 중~높음       | 대형     | OpenJDK 기반        |

## 6. GC 튜닝 주요 옵션

```
1 | -XX:+UseG1GC
2 | -XX:MaxGCPauseMillis=200
3 | -XX:+PrintGCDetails
4 | -XX:+PrintGCDateStamps
5 | -Xlog:gc*:file=gc.log:time,uptime,level,tags
```

## 7. GC 로그 예시

```
1 | [GC pause (G1 Evacuation Pause) (young), 0.0056789 secs]
2 |   [Parallel Time: 4.5 ms, GC Workers: 4]
3 |   [Eden: 12M(12M)->0B(10M) Survivors: 2M->2M Heap: 22M(256M)->12M(256M)]
```

## 8. GC 튜닝 전략

| 상황            | 조치                          |
|---------------|-----------------------------|
| 응답 지연 문제      | G1GC, ZGC 사용 고려             |
| Throughput 중시 | Parallel GC 또는 G1           |
| GC 빈번         | Heap 크기 조정, Minor GC 횟수 줄이기 |

| 상황               | 조치                   |
|------------------|----------------------|
| Full GC 잦음       | 객체 수명 분석 → 메모리 누수 의심 |
| OutOfMemoryError | Heap 크기 확대 or 로직 점검  |

## 9. 실무에서 자주 쓰는 조합

| 목적        | 추천 조합  |
|-----------|--|
| 일반 서버     | <code>G1GC</code> + <code>MaxGCPauseMillis=200</code>  |
| 실시간/지연 민감 | <code>ZGC</code> or <code>ShenandoahGC</code>  |
| JVM 튜닝 필요 | <code>-Xms</code> , <code>-Xmx</code> , GC 로깅, <code>jvisualvm</code> , <code>jstat</code> , <code>GCViewer</code> 등 병행 활용 |

## 10. 관련 툴

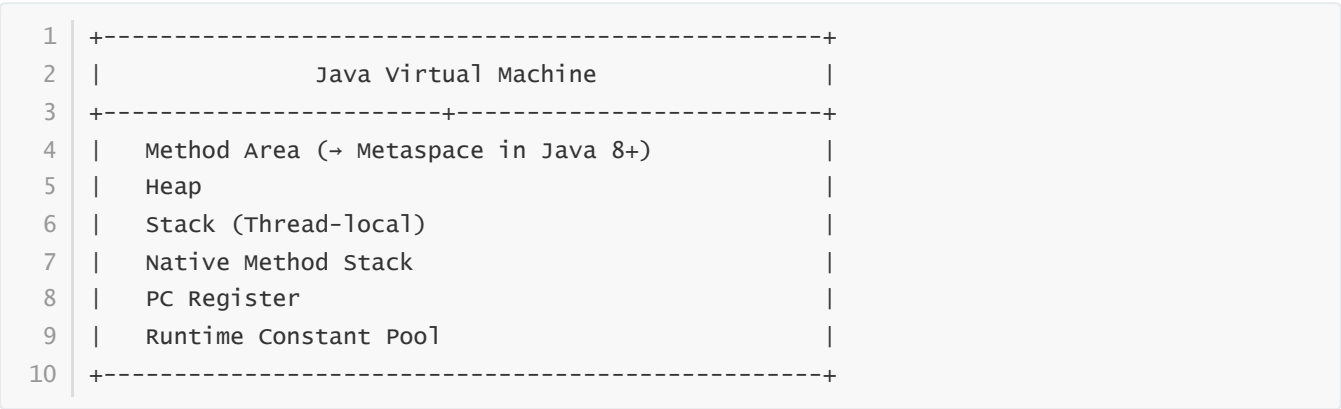
- `jstat -gc <pid>`: GC 통계
- `jvisualvm`: 시각적 GC 모니터링
- `GCViewer`: GC 로그 시각화
- `Flight Recorder`, `Mission Control`: GC + 스레드 + 힙 종합 분석

## 메모리 관리 구조 (Heap, Stack, Metaspace)

JVM Memory Management Structure (Heap, Stack, Metaspace 등)

### 1. JVM 메모리 구조 전체 개요

JVM 메모리는 다음과 같이 여러 영역(Area) 으로 나뉘어 관리된다.





## 2. 힙(Heap) 영역

### 개요

- 객체(instance)와 배열(array)이 저장되는 가장 큰 메모리 공간
- Garbage Collection 대상
- 스레드 간 공유되는 영역

### 내부 구조

- Young Generation (Eden + Survivor)
- Old Generation (Tenured)
- GC가 이 영역을 주로 관리

### 특징

| 항목               | 설명   |
|------------------|--|
| 객체 저장소           | <code>new</code> 키워드로 생성된 객체                             |
| GC 대상            | 참조되지 않는 객체   |
| OutOfMemoryError | <code>java.lang.OutOfMemoryError: Java heap space</code> |

## 3. 스택(Stack) 영역

### 개요

- 각 스레드마다 독립적으로 생성되는 메모리 공간
- 메서드 호출 시 프레임(frame) 단위로 push
- 메서드 종료 시 pop

### 포함 요소

- 지역 변수, 매개변수
- 리턴 값, 연산 중간값
- 호출 스택 정보

### 특징

| 항목       | 설명   |
|----------|--|
| 메서드 실행 시 | 프레임 생성   |
| 종료 시     | 프레임 제거   |
| 속도       | 매우 빠름 (LIFO 구조)  |
| 예외       | <code>java.lang.StackOverflowError</code> (재귀 무한 호출 등) |

## 4. 메타스페이스(Metaspace)

### 개요

- Java 8부터 **Permanent Generation (PermGen)**을 대체
- 클래스의 **메타데이터(Class 객체, method info 등)** 저장 공간
- JVM이 아닌 **OS 네이티브 메모리**를 사용

### 특징

| 항목    | 설명  |
|-------|---|
| 저장 대상 | 클래스 정의 정보, 메서드 시그니처                                 |
| 크기 관리 | 기본 무제한 ( <code>-XX:MaxMetaspaceSize</code> 로 제한 가능) |
| GC 대상 | 사용되지 않는 클래스 언로드 가능                                  |
| 예외    | <code>java.lang.OutOfMemoryError: Metaspace</code>  |

## 5. PC Register

- 각 스레드마다 생성되는 레지스터
- 현재 실행 중인 **JVM 명령어 주소**를 저장
- JVM이 어떤 바이트코드를 실행하고 있는지 추적

## 6. Native Method Stack

- **JNI (Java Native Interface)** 로 호출된 **C/C++ native 코드** 실행을 위한 공간
- 예외: `java.lang.StackOverflowError` 또는 네이티브 메모리 충돌 발생 가능

## 7. Runtime Constant Pool

- 각 클래스나 인터페이스에 대한 **상수 풀(constant pool)** 저장 영역
- 리터럴 문자열, 상수 값, 메서드/필드 참조 정보 등

예:

```
1 | String a = "hello"; // 문자열 리터럴 → 상수 풀에 저장
```

## 8. 메모리 구조 요약 비교

| 영역           | 용도         | 관리 주체  | 오류                 |
|--------------|------------|--------|--------------------|
| Heap         | 객체 저장      | GC     | OutOfMemoryError   |
| Stack        | 호출/지역 변수   | JVM    | StackOverflowError |
| Metaspace    | 클래스 메타데이터  | OS 메모리 | OutOfMemoryError   |
| PC Register  | 바이트코드 주소   | JVM    | 없음                 |
| Native Stack | JNI 메서드 실행 | JVM/OS | StackOverflowError |

## 9. JVM 메모리 설정 옵션

```
1 -Xms512m           # 초기 힙 크기
2 -Xmx2048m          # 최대 힙 크기
3 -Xss1m             # 스택 크기 설정
4 -XX:MetaspaceSize=128m # 메타스페이스 초기 크기
5 -XX:MaxMetaspaceSize=512m # 메타스페이스 최대 크기
```

## 10. 시각적 정리

```
1 [ JVM Process ]
2  └─ Heap (shared, GC 대상)
3     └─ Eden
4     └─ Survivor (S0/S1)
5     └─ Old
6  └─ Stack (per thread)
7  └─ Metaspace (class metadata, non-heap)
8  └─ PC Register
9  └─ Native Method Stack
```

## 11. 실무 팁

- Heap 이 커도 Stack 은 작을 수 있다 → 대량 재귀 시 스택 오류 가능
- 클래스 로딩이 많으면 Metaspace 증가 → 제한 필요
- Heap Dump 분석 도구: jmap, MAT, VisualVM, YourKit
- 스레드 수가 많다면 -xss 튜닝으로 메모리 효율 최적화 필요

# Unsafe API

## 1. Unsafe란 무엇인가?

- `sun.misc.Unsafe`는 Java에서 일반적으로 허용되지 않는 낮은 수준의 메모리 제어 기능을 제공하는 클래스.
- 예를 들면:
  - 직접 메모리 할당/해제
  - 객체의 필드 오프셋에 직접 접근
  - CAS 연산 (Compare-And-Swap)
  - 클래스 정의 (`defineClass`)
- JVM 내부나 고성능 라이브러리(Lock-Free Queue 등)에서만 사용을 권장

## 2. 왜 위험한가?

| 이유            | 설명                                |
|---------------|-----------------------------------|
| 타입 안전성 무시     | 아무 필드에나 값을 삽입 가능                  |
| GC 우회         | 메모리를 직접 할당 → GC에 의해 관리되지 않음       |
| 보안 무시         | private 필드, final 필드도 조작 가능       |
| JVM 크래시 유발 가능 | 잘못된 오프셋 접근은 SIGSEGV 오류로 JVM 다운 가능 |

그래서 Java 9부터는 `jdk.internal.misc.Unsafe`로 이동되고, `--add-exports` 옵션 없이 사용 불가

## 3. Unsafe 인스턴스 획득 방법

정상적인 방법으로는 사용할 수 없고, 리플렉션을 써야 함:

```
1  import sun.misc.Unsafe;
2
3  import java.lang.reflect.Field;
4
5  public class UnsafeAccess {
6      public static Unsafe getUnsafe() throws Exception {
7          Field f = Unsafe.class.getDeclaredField("theUnsafe");
8          f.setAccessible(true);
9          return (Unsafe) f.get(null);
10     }
11 }
```

## 4. 주요 기능별 API 설명

### ◆ 4.1 메모리 직접 제어

```
1 long address = unsafe.allocateMemory(8); // 8바이트 할당
2 unsafe.putLong(address, 123456789L);    // 해당 주소에 값 저장
3 long value = unsafe.getLong(address);    // 해당 주소에서 값 로드
4 unsafe.freeMemory(address);              // 해제
```

| 메서드                                 | 설명           |
|-------------------------------------|--------------|
| <code>allocateMemory(size)</code>   | OS 메모리 직접 할당 |
| <code>freeMemory(address)</code>    | 직접 해제        |
| <code>putXxx(address, value)</code> | 주소에 값 저장     |
| <code>getXxx(address)</code>        | 주소에서 값 읽기    |

### ◆ 4.2 객체 필드 직접 접근

```
1 Field f = MyClass.class.getDeclaredField("x");
2 long offset = unsafe.objectFieldOffset(f);
3 unsafe.putInt(myObject, offset, 42); // 강제로 x에 42 저장
```

| 메서드                                     | 설명            |
|---|---------------|
| <code>objectFieldOffset(Field f)</code> | 해당 필드의 오프셋 구함 |
| <code>putInt(obj, offset, val)</code>   | 필드에 직접 값 저장   |
| <code>getInt(obj, offset)</code>        | 필드에서 직접 값 읽기  |

### ◆ 4.3 CAS(Compare-And-Swap) 연산

```
1 boolean success = unsafe.compareAndSwapInt(obj, offset, 10, 20);
```

- 멀티스레드 환경에서 락 없이 안전하게 값을 교체
- `java.util.concurrent.atomic` 패키지 구현의 핵심

#### ◆ 4.4 클래스 정의

```
1 | class<?> clazz = unsafe.defineClass(null, classBytes, 0, classBytes.length, null, null);
```

- 바이트코드를 직접 JVM에 로드
- 스프링, 프록시 프레임워크, ByteBuddy 등이 사용

#### ◆ 4.5 객체 생성 (생성자 무시)

```
1 | MyClass obj = (MyClass) unsafe.allocateInstance(MyClass.class);
```

- 생성자를 호출하지 않고 객체 생성
- ORM, 프레임워크 내부에서 사용

### 5. 대표적인 사용 사례

| 분야        | 설명                             |
|-----------|--------------------------------|
| 고성능 라이브러리 | Netty, Cassandra, Kafka 등      |
| 락 프리 알고리즘 | CAS 기반 구조 구현                   |
| 바이트코드 프록시 | ASM, ByteBuddy 등에서 클래스 직접 정의   |
| 메모리 최적화   | JVM 힙 외 직접 메모리 사용 (Off-Heap)   |
| 성능 튜닝     | Unsafe 기반 구조체처럼 동작하는 클래스 작성 가능 |

### 6. Java 9 이후의 변화

| 항목                                  | 설명  |
|-------------------------------------|---|
| 모듈화로 인해 기본 접근 불가                    | Unsafe 는 <code>jdk.unsupported</code> 모듈로 분리됨   |
| 사용하려면 <code>--add-exports</code> 필요 | 예: <code>--add-exports java.base/jdk.internal.misc=ALL-UNNAMED</code>                                 |
| 대체 API로 전환 유도 중                     | <code>VarHandle</code> , <code>ByteBuffer</code> , <code>Foreign Memory API</code> (Project Panama) 등 |

## 7. 위험성 예시

```
1 // final 필드 우회
2 class Secret {
3     private final int x = 42;
4 }
5 Field f = Secret.class.getDeclaredField("x");
6 long offset = unsafe.objectFieldOffset(f);
7 unsafe.putInt(secretInstance, offset, 999); // final 무력화
```

👉 보안상 치명적이고, 구조적 안정성이 완전히 무너짐

## 8. 정리: 장단점

| 항목  | 장점                  | 단점          |
|-----|---------------------|-------------|
| 성능  | GC 우회, zero-copy 가능 | GC와 분리되어 위험 |
| 제어력 | Java가 허용하지 않는 작업 가능 | 타입 안전성 무시   |
| 실용성 | 고성능 라이브러리에 유용       | JVM 크래시 가능성 |

## 9. 대체 기술

| 목적       | 대체  |
|----------|---|
| 객체 필드 접근 | <code>Reflection</code> , <code>VarHandle</code>                                |
| CAS 연산   | <code>Atomic*</code> 클래스, <code>VarHandle.compareAndSet()</code>                |
| Off-Heap | <code>ByteBuffer</code> , <code>sun.nio.ch.DirectBuffer</code> , Panama FFM API |
| Class 정의 | <code>ClassLoader.defineClass()</code> , <code>ByteBuddy</code>                 |

## 10. 실무 조언

- 절대 일반 애플리케이션에서는 직접 사용하지 말 것
- 반드시 성능 또는 기술적 이유가 명확한 경우에만 신중히 사용
- Java 17 이상에서는 가능하면 **VarHandle**, **Panama API**, **Project Loom** 기반 구조로 전환 권장

## JVM 튜닝 및 프로파일링 (`jvisualvm`, `jconsole`, `Flight Recorder`)

주요 도구: `jvisualvm`, `jconsole`, `Java Flight Recorder`

# 1. JVM 튜닝이란?

- JVM 옵션을 조정하여 메모리, GC, 스레드, JIT 성능을 최적화하는 작업
- 목적:
  - GC 지연 시간 단축
  - OOM 방지
  - CPU 과부하 방지
  - 서비스 응답 속도 향상

# 2. JVM 기본 성능 관련 옵션

| 옵션                                      | 설명               |
|---|------------------|
| <code>-Xms512m</code>                   | 초기 Heap 크기       |
| <code>-Xmx2g</code>                     | 최대 Heap 크기       |
| <code>-Xss512k</code>                   | 스택 크기 설정         |
| <code>-XX:+UseG1GC</code>               | GC 알고리즘 설정       |
| <code>-XX:MaxGCPauseMillis=200</code>   | G1 GC 목표 지연 시간   |
| <code>-XX:+PrintGCDetails</code>        | GC 상세 로그 출력      |
| <code>-Xlog:gc*:file=gc.log:time</code> | Java 9+ GC 로그 설정 |

# 3. 프로파일링 개요

프로파일링(Profiling)은 애플리케이션의 실행 시 성능을 정밀 측정하여 병목이나 리소스 과다 사용 지점을 파악하는 과정이다.

주요 분석 대상:

- Heap 메모리 사용량
- GC 빈도 및 소요 시간
- CPU 사용률
- 스레드 수, 상태
- 메서드 호출 횟수/시간
- OutOfMemoryError, Deadlock 발생 여부



## 4. 도구 ①: jVisualVM

### 개요

- JDK에 기본 포함 (jdk/bin/jvisualvm)
- GUI 기반 통합 프로파일러 + 모니터링 도구
- 원격 JVM 연결 가능 (JMX)

### 기능 요약

| 항목            | 설명                                      |
|---------------|---|
| CPU 프로파일링     | 메서드 단위 실행 시간 분석                         |
| Memory 프로파일링  | 클래스별 인스턴스 수, 메모리 소비량 추적                 |
| GC 모니터링       | 수집 빈도, GC 시간 추적                         |
| Heap Dump 보기  | 실시간 or 파일 기반 분석 가능                      |
| Thread 상태 시각화 | 스레드 수, 상태, Deadlock 표시                  |
| Plugin        | Visual GC, MBeans 브라우저, Sampler 등 추가 가능 |

### 사용법

```
1 | jvisualvm
```

- 좌측 트리 → JVM 선택 → [Monitor], [Threads], [Sampler] 탭 선택
- 메모리 스냅샷 저장: `Heap Dump`

## 5. 도구 ②: jConsole

### 개요

- JMX(Java Management Extensions) 기반 모니터링 도구
- GUI 간단하고 가볍지만 실시간 분석에 특화

### 주요 기능

| 항목       | 설명                            |
|----------|-------------------------------|
| 메모리 모니터링 | Eden, Survivor, Old 별 사용량 시각화 |
| CPU 사용률  | JVM CPU 점유율 확인                |
| GC 모니터링  | Minor, Full GC 발생 추적          |
| 스레드 수    | Runnable, Blocked 등 상태 분류     |

| 항목          | 설명                    |
|-------------|-----------------------|
| MBeans 브라우저 | JMX MBean 속성 접근/조작 가능 |

1 | jconsole

로컬 JVM 또는 `-Dcom.sun.management.jmxremote`로 원격 연결 가능

## 6. 도구 ③: Java Flight Recorder (JFR)

### 개요

- 고급 성능 프로파일러, 낮은 오버헤드로 장시간 추적 가능
- Java 11 이상 기본 내장 (`jdk.jfr`)
- 기록 파일은 `.jfr` 확장자

### 특징

| 항목        | 설명                              |
|-----------|---------------------------------|
| 오버헤드 낮음   | 실시간 운영 환경에서 사용 가능               |
| 이벤트 기반 기록 | GC, 스레드, Lock, IO, 메서드 호출, 할당 등 |
| 비주얼 분석 도구 | Java Mission Control (JMC) 필요   |

### 사용법

```
1 # 명령어 기반 수집
2 java -XX:StartFlightRecording=filename=app.jfr,duration=60s -jar MyApp.jar
```

### Java Mission Control (JMC)

- `.jfr` 파일을 분석할 수 있는 GUI 도구
- 주요 분석 항목:
  - GC 이벤트
  - Lock 경합
  - 스레드 활동
  - 클래스 로딩/언로딩
  - 메서드 Hotspot 분석

## 7. 실무 시나리오 예시

### 시나리오 1: GC 과다

- GC 로그 분석 ( `PrintGCDetails` )
- G1 GC + `MaxGCPauseMillis` 조정
- Heap 크기 조정 ( `Xmx` , `Xms` )

### 시나리오 2: OOM 발생

- `Heap Dump` 생성 후 `jvisualvm` 으로 분석
- 객체 수가 계속 증가하는 클래스 추적
- `WeakReference` , 캐시 메커니즘 확인

### 시나리오 3: 스레드 증가/Deadlock

- `jstack` 으로 스레드 상태 덤프
- `jvisualvm` , `jconsole` 에서 Thread 탭 분석
- `synchronized` 블록 경합 여부 확인

## 8. 기타 유용한 툴

| 도구  | 설명                          |
|---|-----------------------------|
| <code>jmap -dump:live,format=b,file=heap.bin &lt;pid&gt;</code> | Heap Dump 저장                |
| <code>jstack &lt;pid&gt;</code>                                 | 스레드 스택 덤프                   |
| <code>jstat -gc &lt;pid&gt;</code>                              | GC 사용률 실시간 분석               |
| <code>async-profiler</code>                                     | 네이티브 + 자바 통합 CPU/메모리 분석기    |
| <code>VisualVM Plugins</code>                                   | Visual GC, Profiler 등 확장 가능 |

## 9. JVM 성능 튜닝 전략 요약

| 대상       | 조정 요소  |
|----------|--|
| GC 시간    | GC 종류 선택 (G1/ZGC), Heap 크기                         |
| OOM      | 객체 수명 관리, 캐시/리소스 누수 점검                             |
| 응답 지연    | <code>-XX:MaxGCPauseMillis</code> , Thread/Lock 분석 |
| CPU 과다   | JFR/Profiler로 Hot Method 추적                        |
| GC 로그 분석 | GCViewer, GCEasy, jClarity 사용 가능                   |

## 10. JVM 프로파일링 + 튜닝 통합 흐름

```
1  [ JVM 실행 ]
2      ↓
3  [ GC 로그, JFR, Heap Dump ]
4      ↓
5  [ 도구: jvisualvm, jconsole, JMC ]
6      ↓
7  [ 병목 원인 분석 ]
8      ↓
9  [ JVM 옵션 수정 + 코드 개선 ]
```