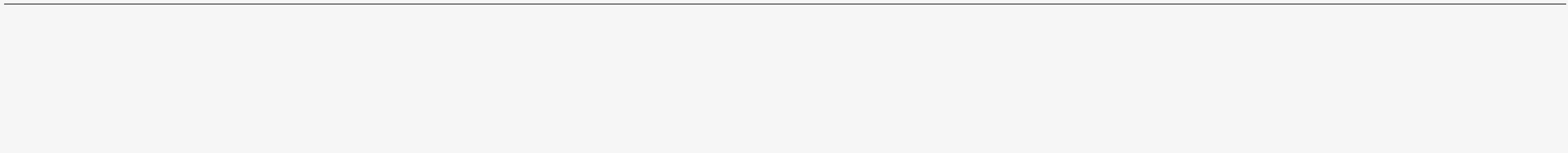


JVM 개발자를 위한

안정적인 서비스 운영을 위한 즉시 적용 가능한 엔지니어링 기법



대상

- JVM 기반 개인, 팀 프로젝트 배포 및 운영 경험이 있는 분
- 주니어 백엔드 엔지니어

목차

- 안정적인 서비스란?
- 서비스 태동기
- 서비스 성장기
- 서비스 성숙기
- 정리

서비스에 영향을 주는 요인은 무엇일까

안정적인 서비스란?

안정적인 서비스란?

안정적인 서비스란?

Stable, Stability

견고한, (외부의 영향에도) 같은 상태를 유지하는 성질

안정적인 서비스란?

Service Level Indicator

서비스의 성능, 신뢰성 등 품질을 객관적으로 측정하기
위한 정량적 지표

안정적인 서비스란?

Service Level Indicator

서비스의 성능, 신뢰성 등 품질을 객관적으로 측정하기
위한 정량적 지표

응답 시간, 요청 실패율, 가동 시간, 처리량 등

안정적인 서비스란?

- 내부 요인
- 외부 요인

안정적인 서비스란?

- 내부 요인 애플리케이션 결함
 - 논리적 오류, 메모리 누수
 - 비효율적인 쿼리, 동기화
- 외부 요인

아키텍처 결함

- SPoF, SSoT
- 서비스 간 강한 결합

인프라 결함

- 부족한 가용 리소스, 서버
- 이중화, DR의 부재
- 잘못된 배포 전략

안정적인 서비스란?

- **내부 요인** 애플리케이션 결함
 - 논리적 오류, 메모리 누수
 - 비효율적인 쿼리, 동기화
- **외부 요인**

- 아키텍처 결함
 - SPoF, SSoT
 - 서비스 간 강한 결합

- 인프라 결함
 - 부족한 가용 리소스, 서버
 - 이중화, DR의 부재
 - 잘못된 배포 전략

**인지하였으나 개선하지 않는 것들
또는 못하는 것들**

안정적인 서비스란?

- 내부 요인
- 외부 요인

안정적인 서비스란?

- 내부 요인 예측 불가능한 트래픽
- 외부 요인 • 트래픽 급증
 • DDoS

- 외부 서비스 의존성
 - 서드파티 장애
 - 응답 지연

- 인프라 결함
 - 클라우드 서비스 장애
 - 네트워크 장애

안정적인 서비스란?

- 내부 요인 예측 불가능한 트래픽
- 외부 요인 • 트래픽 급증
 • DDoS

외부 서비스 의존성

- 서드파티 장애
- 응답 지연

인프라 결함

- 클라우드 서비스 장애
- 네트워크 장애

인지하기 어렵지만 대비해야 하는 것들

엔지니어링 기법?

엔지니어링 기법?

전략 : 장기적 목표를 달성하기 위한 계획, 목표, 방향성

기술 : 전략을 실현하기 위한 구체적인 행동 (설계, 해결법 등)

서비스를 설계하는 단계에서는 무엇을 고려하는가?

서비스 태동기

서비스 태동기

서비스 태동기

페르미의 추정 Fermi Estimate

서비스 태동기

페르미의 추정 Fermi Estimate

어떠한 문제에 대해 기초적인 지식과 논리적 추론만으로
짧은 시간 안에 **대략적인 근사치를 추정**하는 방식

서비스 태동기

페르미의 추정 Fermi Estimate

e.g. 출시될 서비스의 DAU / TPS는 얼마인가?

서비스 태동기

페르미의 추정 Fermi Estimate

e.g. 출시될 서비스의 DAU / TPS는 얼마인가?

- 예상 사용자 수 : 300만
- 세션 유효 기간 : 7일
- 일일 메신저 접속 횟수 : 30회
- 일일 메시지 전송 횟수 : 100회

서비스 태동기

페르미의 추정 Fermi Estimate

e.g. 출시될 서비스의 DAU / TPS는 얼마인가?

- 예상 사용자 수 : 300만
- 세션 유효 기간 : 7일
- 일일 메신저 접속 횟수 : 30회
- 일일 메시지 전송 횟수 : 100회

DAU = 예상 사용자 수 / 세션 유효 기간

TPS = DAU x 인당 일일 트랜잭션 수 / 86,400

서비스 태동기

페르미의 추정 Fermi Estimate

e.g. 출시될 서비스의 DAU / TPS는 얼마인가?

- 예상 사용자 수 : 300만
- 세션 유효 기간 : 7일
- 일일 메신저 접속 횟수 : 30회
- 일일 메시지 전송 횟수 : 100회

DAU = 예상 사용자 수 / 세션 유효 기간

$$3,000,000 / 7 = 428,571\text{명}$$

TPS = DAU x 인당 일일 트랜잭션 수 / 86,400

$$428,571 \times 130 / 86,400 = 647 \text{ TPS}$$

서비스 태동기

페르미의 추정 Fermi Estimate

e.g. 출시될 서비스의 DAU / TPS는 얼마인가?

- 예상 사용자 수 : 300만
- 세션 유효 기간 : 7일
- 일일 메신저 접속 횟수 : 30회
- 일일 메시지 전송 횟수 : 100회

DAU = 예상 사용자 수 / 세션 유효 기간

$$3,000,000 / 7 = 428,571\text{명}$$

TPS = DAU x 인당 일일 트랜잭션 수 / 86,400

$$428,571 \times 130 / 86,4000 = 647 \text{ TPS}$$

충족해야 할 최소한의 목표를 추정

서비스 태동기

페르미의 추정 Fermi Estimate

e.g. 출시될 서비스의 DAU / TPS는 얼마인가?

- 예상 사용자 수 : 300만
- 세션 유효 기간 : 7일
- 일일 메신저 접속 횟수 : 30회
- 일일 메시지 전송 횟수 : 100회

$$\text{DAU} = \text{예상 사용자 수} / \text{세션 유효 기간}$$
$$3,000,000 / 7 = 428,571\text{명}$$

$$\text{TPS} = \text{DAU} \times \text{인당 일일 트랜잭션 수} / 86,400$$
$$428,571 \times 130 / 86,4000 = 647 \text{ TPS}$$

충족해야 할 최소한의 목표를 추정

성능 테스트 진행 후 이러한 수치를 바탕으로 서비스 운영에 필요한 인프라 리소스 할당 및 스케일 아웃 전략을 구성

서비스 태동기

모니터링 & 관측 가능성 Monitoring & Observability

서비스 태동기

모니터링 & 관측 가능성 Monitoring & Observability

Monitoring

정보 수집, 영향, 관리 또는 지시를 목적으로 다양한 활동 또는 정보를 지속적으로 지켜보는 것

Observability

시스템의 출력 변수를 사용하여 특정 상태 변수에 대한 정보를 알아낼 수 있을 때, 그 상태 변수는 관측 가능하다고 하며, 시스템의 모든 상태 변수가 관측 가능할 때 그 시스템이 관측 가능하다고 한다

서비스 태동기

모니터링 & 관측 가능성 Monitoring & Observability

Monitoring

정보 수집, 영향, 관리 또는 지시를 목적으로 다양한 활동 또는 정보를 지속적으로 지켜보는 것

CPU, Memory, Direct Buffer, Disk I/O
Network I/O 등의 Metric, System Log..

Observability

시스템의 출력 변수를 사용하여 특정 상태 변수에 대한 정보를 알아낼 수 있을 때, 그 상태 변수는 관측 가능하다고 하며, 시스템의 모든 상태 변수가 관측 가능할 때 그 시스템이 관측 가능하다고 한다

Trace ID, Span ID, Traceparent Header..

서비스 태동기

모니터링 & 관측 가능성 Monitoring & Observability



X (구 Twitter)에서 개발된 오픈소스 플랫폼
트랜잭션 추적에 중점
OpenTracing 기반, Brave (JVM) 지원
2012년 공개



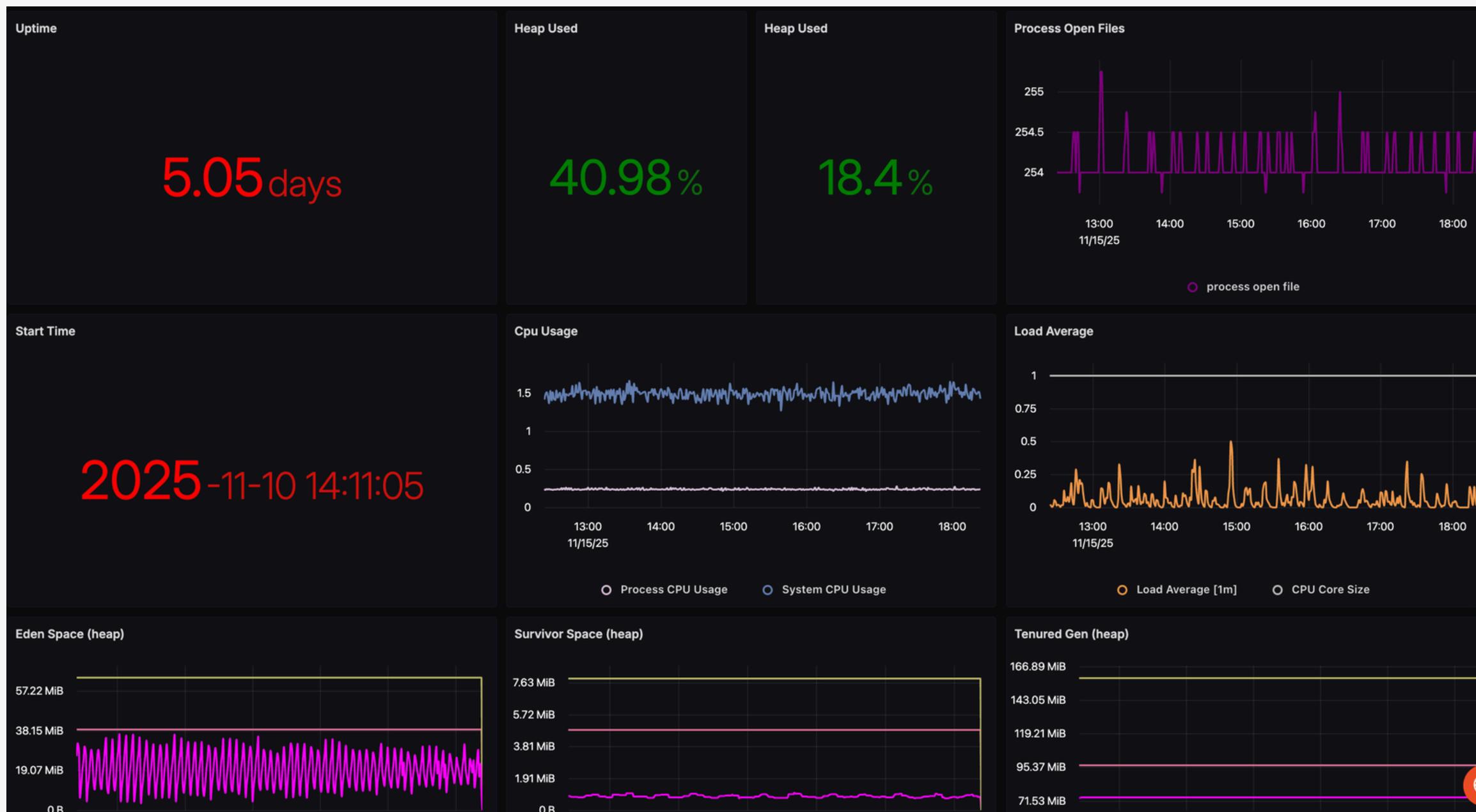
네이버에서 개발한 **분산 추적** 플랫폼
Google Dapper 스타일의 추적 방식 사용
코드 내 성능 측정에는 **BCI** 활용
2015년 공개



CNCF 지원을 받는 프로젝트로 **현재 업계 표준**
OpenTracing과 OpenCensus(Google Census)의 병합

서비스 태동기

모니터링 & 관측 가능성 Monitoring & Observability



서비스가 성장하는 단계에서는 무엇을 고려하는가?

서비스 성장기

서비스 성장기

콜드 스타트 Cold Start

서비스 성장기

콜드 스타트 Cold Start

시스템 또는 그 일부가 생성되거나 재시작된 후
정상적으로 **작동하지 않는** 상황

서비스 성장기

콜드 스타트 Cold Start

시스템 또는 그 일부가 생성되거나 재시작된 후
정상적으로 작동하지 않는 상황

단순 웹 서비스 뿐만 아니라 추천 시스템, 서비스 아키텍처 등
다양한 환경에서 **일관적으로 발생하는 문제**

서비스 성장기

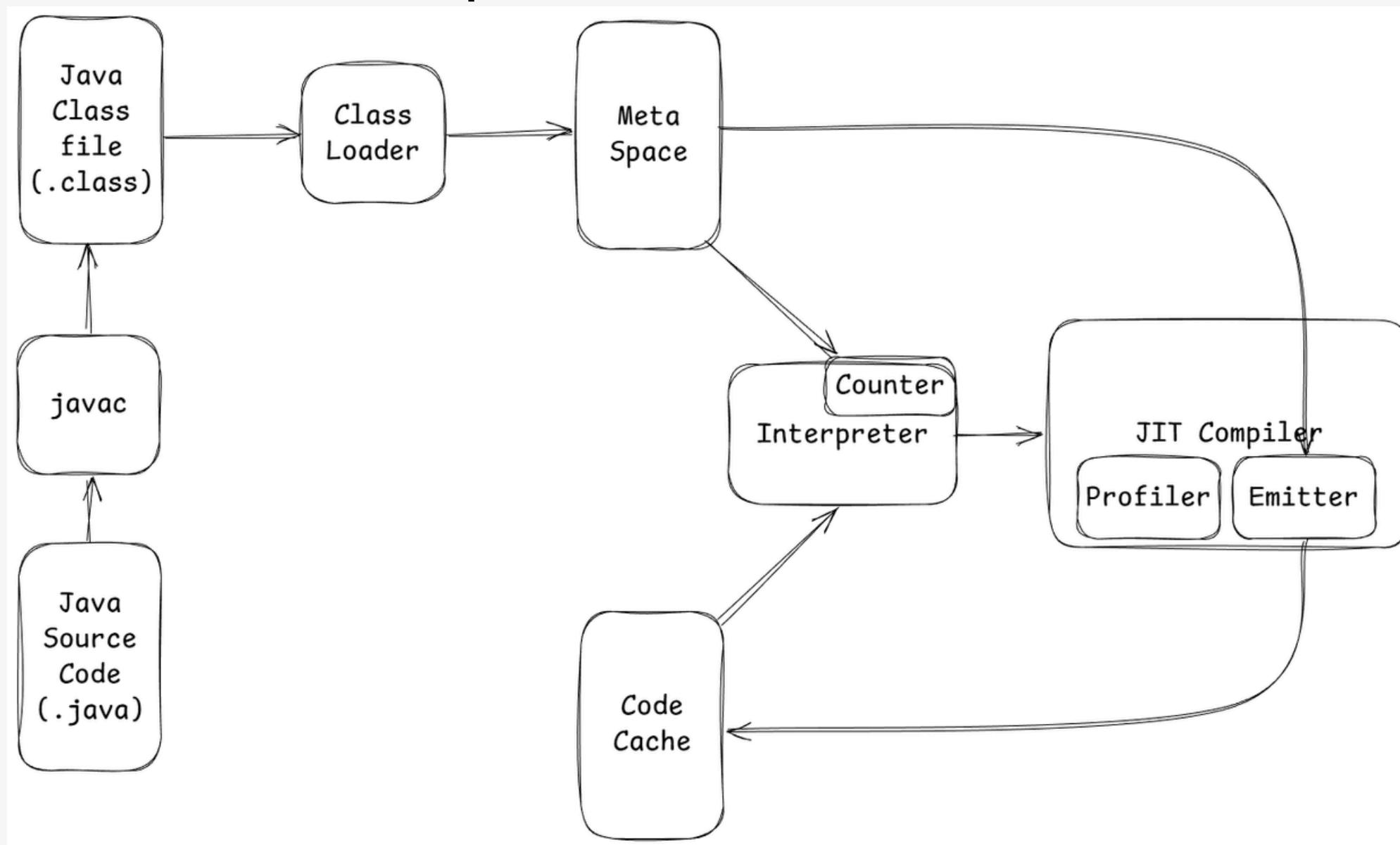
콜드 스타트 Cold Start

- JVM (JIT Compiler & Code Cache)

서비스 성장기

콜드 스타트 Cold Start

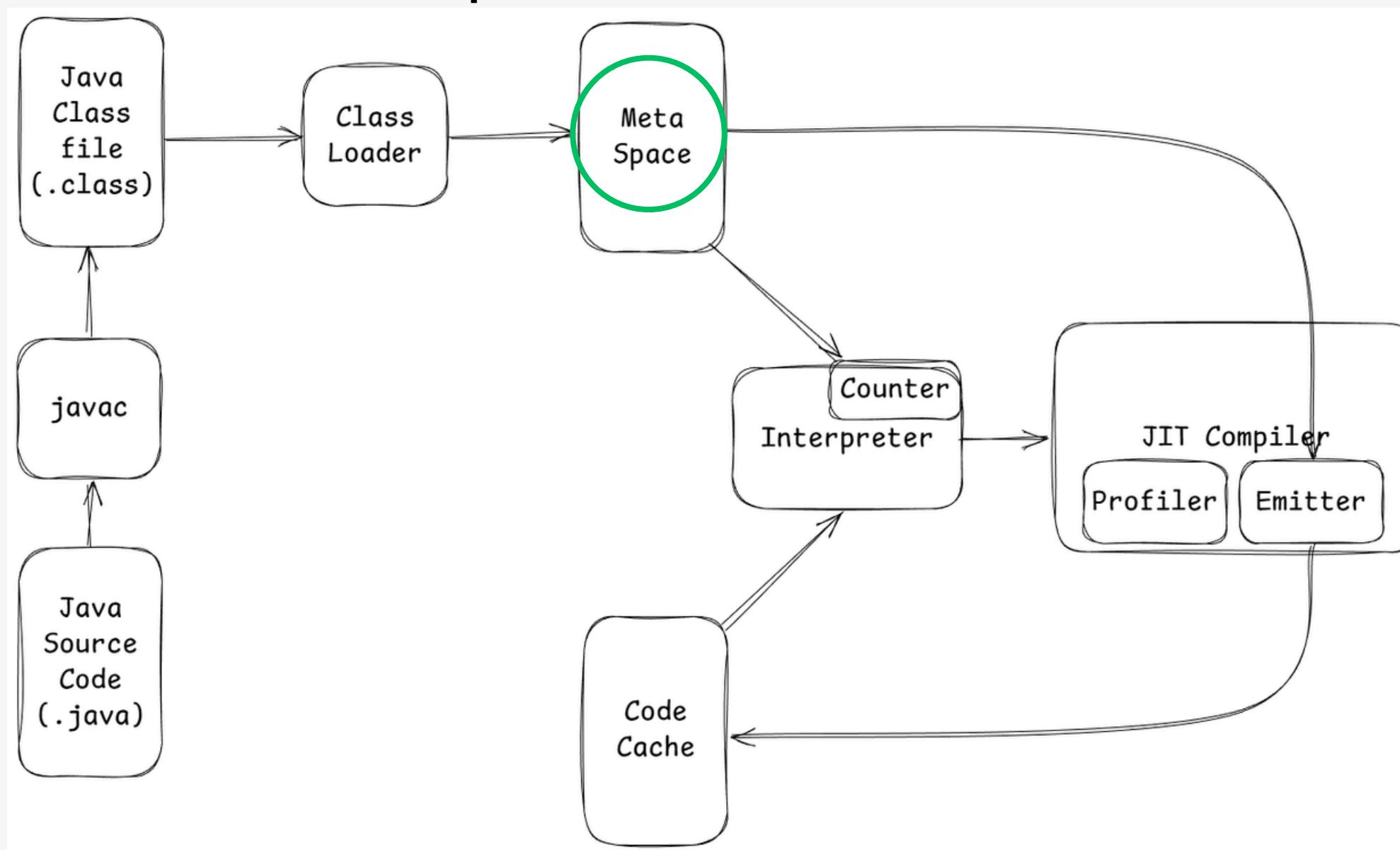
- JVM (JIT Compiler & Code Cache)



서비스 성장기

콜드 스타트 Cold Start

- JVM (JIT Compiler & Code Cache)



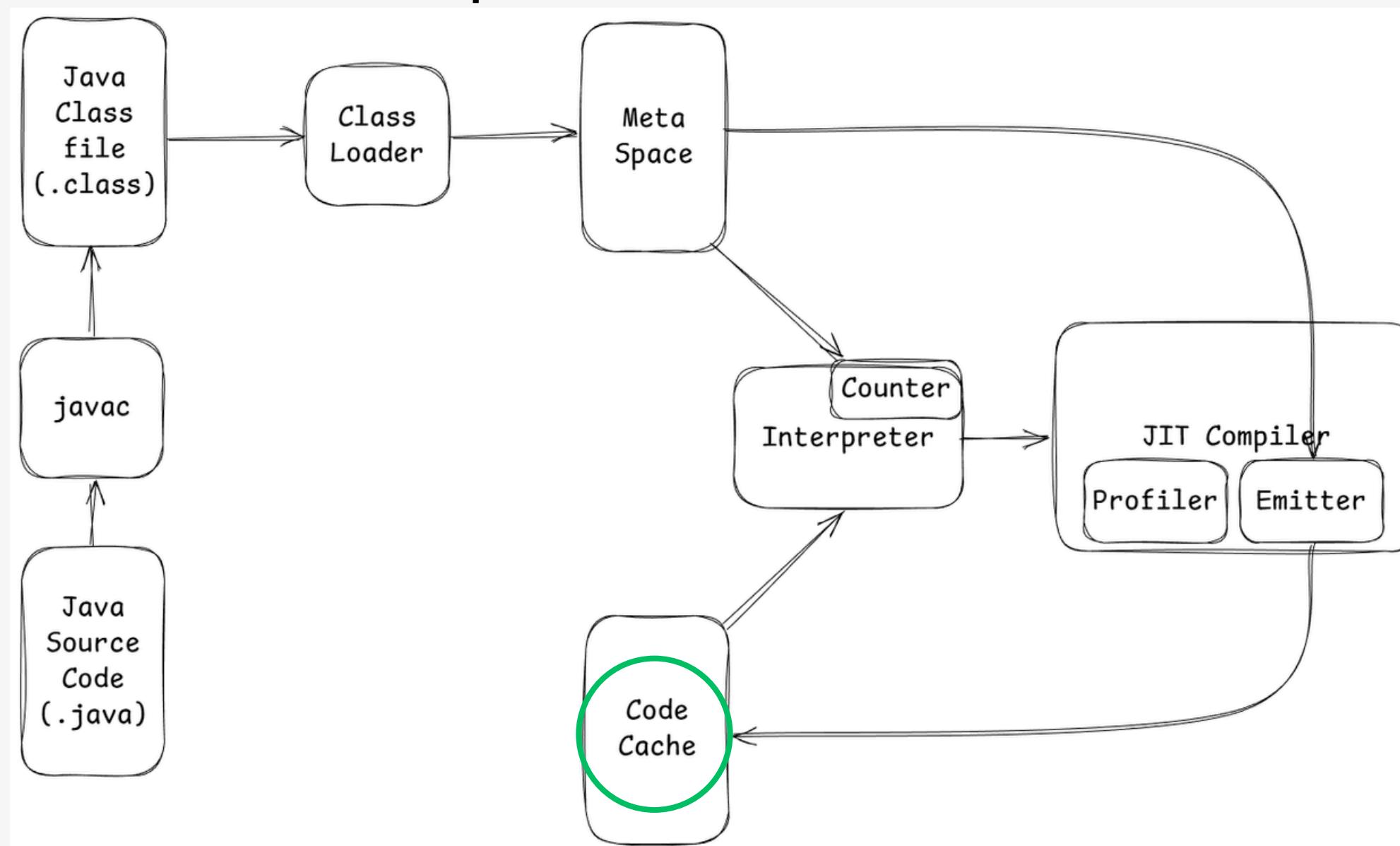
Meta Space

- Native Memory (without JVM)
- ClassLoader가 읽어온 메타데이터를 저장하는 영역
- Reflection으로 인한 성능 저하가 덜한 이유기도 함

서비스 성장기

콜드 스타트 Cold Start

- JVM (JIT Compiler & Code Cache)



Meta Space

- Native Memory (without JVM)
- ClassLoader가 읽어온 메타데이터를 저장하는 영역
- Reflection으로 인한 성능 저하가 덜한 이유이기도 함

Code Cache

- **Profile Guided Optimization**
 - 서비스 런타임 특성에 맞게 최적화
- Bytecode를 Machine language로 변환하는 비용은 생각보다 크므로, 자주 실행되는 핫 코드의 컴파일 결과를 **Code Cache**에 저장한다

서비스 성장기

콜드 스타트 Cold Start

- JVM (JIT Compiler & Code Cache)

`XX:CICompilerCount` 값으로 JIT 컴파일에 사용될 스레드 수를 증가시킬 수 있음

- Bytecode에 대한 컴파일은 Compile Queue를 경유해 실행
- 해당 값 조정으로 컴파일에 사용하는 스레드 수를 증가시킬 수 있음 => 컴파일 처리량 증가
 - CPU 사용량도 증가하며, 인코딩, 압축, 병렬 계산 등의 로직 성능에 영향을 미침

서비스 성장기

콜드 스타트 Cold Start

- JVM (JIT Compiler & Code Cache)

`XX:CICompilerCount` 값으로 JIT 컴파일에 사용될 스레드 수를 증가시킬 수 있음

- Bytecode에 대한 컴파일은 Compile Queue를 경유해 실행
- 해당 값 조정으로 컴파일에 사용하는 스레드 수를 증가시킬 수 있음 => 컴파일 처리량 증가
 - CPU 사용량도 증가하며, 인코딩, 압축, 병렬 계산 등의 로직 성능에 영향을 미침

JIT Compile은 계층적으로 수행 (Java 7에서 도입)

- **Tier 1 ~ 4** (1~3은 C1 Compiler, 4는 C2 Compiler가 수행)
- **단기적인 서버 워크 개선**을 원한다면 => `XX:Tier1CompileThreshold` 임계 값을 조정
- **최대 성능에 빨리 도달**하길 바란다면 => `XX:Tier3~4CompileThreshold` 임계 값을 조정
- C2 Compiler는 코드의 실행 속도 최적화를 최우선하여 컴파일 속도가 느리지만 성능 향상에 큰 도움
 - **프로파일링**을 사용하여 최적화하기 때문 (inlining, Escape Analysis, Loop Unrolling 등 수행)
- Code Cache는 Java 9부터 세 가지 영역으로 분할 (non-method, Profiled, Non-Profiled)

서비스 성장기

콜드 스타트 Cold Start

- Resource Pool (DB Connection Pool, Thread Pool)

서비스 성장기

콜드 스타트 Cold Start

- Resource Pool (DB Connection Pool, Thread Pool)

모든 풀에는 평시 리소스 개수, 최대 리소스 개수를 구분하여 설정함
즉, 항상 모든 리소스가 준비된 상태가 아님

서비스 성장기

콜드 스타트 Cold Start

- Resource Pool (DB Connection Pool, Thread Pool)

Resource Pool 개념을 사용하는 이유?

- 새로운 리소스를 생성하는 비용이 너무 크기 때문

서비스 성장기

콜드 스타트 Cold Start

- Resource Pool (DB Connection Pool, Thread Pool)

Resource Pool 개념을 사용하는 이유?

- 새로운 리소스를 생성하는 비용이 너무 크기 때문

DB Connection Pool

1. 네트워크 오버헤드 (TCP Handshake)
2. 사용자 인증
3. 세션 생성 및 초기화
4. 세션을 통한 쿼리 질의

JVM ThreadPool (Virtual Thread X)

1. 메모리 내 스레드 객체 생성
2. start() → JNI (JVM_StartThread)
 - a. osThread.cpp > System Call
3. OS 스레드 생성 및 스케줄러 등록
4. JVM 스레드와 OS 스레드 연결 (Mount)
5. run() 실행

스택 메모리 할당, TCB 생성, 스케줄러 등록

서비스 성장기

콜드 스타트 Cold Start

- Resource Pool (DB Connection Pool, Thread Pool)

Resource Pool 개념을 사용하는 이유?

- 새로운 리소스를 생성하는 비용이 너무 크기 때문

DB Connection Pool

1. 네트워크 오버헤드 (TCP Handshake)
2. 사용자 인증
3. 세션 생성 및 초기화
4. 세션을 통한 쿼리 질의

JVM ThreadPool (Virtual Thread X)

1. 메모리 내 스레드 객체 생성
 2. start() → JNI (JVM_StartThread)
 - a. osThread.cpp > System Call
 3. OS 스레드 생성 및 스케줄러 등록
 4. JVM 스레드와 OS 스레드 연결 (Mount)
 5. run() 실행
- 스택 메모리 할당, TCB 생성, 스케줄러 등록

서비스 성장기

콜드 스타트 Cold Start

- Resource Pool (DB Connection Pool, Thread Pool)

DB Connection Pool (HikariCP)

서비스 성장기

콜드 스타트 Cold Start

- Resource Pool (DB Connection Pool, Thread Pool)

DB Connection Pool (HikariCP)

spring.datasource.hikari.minimum-idle와 hikari.maximum-pool-size를 동일한 값으로 설정

- 요청이 증가함에 따라 커넥션을 생성하지 말고 한 번에 생성

서비스 성장기

콜드 스타트 Cold Start

- Resource Pool (DB Connection Pool, Thread Pool)

Tomcat ThreadPool

서비스 성장기

콜드 스타트 Cold Start

- Resource Pool (DB Connection Pool, Thread Pool)

Tomcat ThreadPool

minSpareThreads 값 조정

- 활성 & 유휴 스레드를 모두 포함하는 최소 스레드 수 지정 값

MaxThreads, maxConnections, acceptCount는 Tomcat이 병목으로 확인된 경우에 조정

- 처음부터 조정을 고려해야 할 값은 아님, 특히 maxConnections, acceptCount는 스케일 아웃이 어려운 제약(On-premise)이 있는 상황에서 먼저 고려해볼 수 있음

서비스 성장기

콜드 스타트 Cold Start

- Resource Pool (DB Connection Pool, Thread Pool)

JVM ThreadPool (Virtual Thread X)

필요한 경우, `setPrestartAllCoreThreads = true` 설정

- 작업 요청이 발생할 때마다 스레드를 1개씩 생성하지 않고, `corePoolSize`만큼 미리 생성
- 또는, `prestartCoreThread()`를 통해 원하는 개수만큼 생성할 수도 있음 (Loop)

서비스 성장기

콜드 스타트 Cold Start

- Resource Pool (DB Connection Pool, Thread Pool)

JVM ThreadPool (Virtual Thread X)

필요한 경우, `setPrestartAllCoreThreads = true` 설정

- 작업 요청이 발생할 때마다 스레드를 1개씩 생성하지 않고, `corePoolSize`만큼 미리 생성
- 또는, `prestartCoreThread()`를 통해 원하는 개수만큼 생성할 수도 있음 (Loop)

maxPoolSize는 지정된 Queue Capacity를 웃도는 요청이 왔을 때 증가시킬 스레드 수

- Queue Capacity = `Integer.MAX_VALUE`
- `corePoolSize`를 `maxPoolSize`과 동일하게 설정하는 것도 방법

서비스 성장기

우아한 종료 Graceful Shutdown

서비스 성장기

우아한 종료 Graceful Shutdown

서비스 종료 요청을 받았을 때, 처리하고 있던 작업을 **모두 완료하고**
자원을 정리한 후 안전하게 종료하는 방식

서비스 성장기

우아한 종료 Graceful Shutdown

서비스 종료 요청을 받았을 때, 처리하고 있던 작업을 모두 완료하고 자원을 정리한 후 안전하게 종료하는 방식

Spring Boot 2.3+ 부터는 Graceful shutdown을 지원
`AbstractApplicationContext.doClose()`

서비스 성장기

우아한 종료 Graceful Shutdown

- Tomcat ThreadPool (Spring Boot 2.3 이전)

```
● ● ●

@Component
public class GracefulShutdown implements ApplicationListener<ContextClosedEvent> {
    private static final long GRACE_PERIOD_SECONDS = 30;

    @Override
    public void onApplicationEvent(ContextClosedEvent event) {
        if (event.getApplicationContext() instanceof ServletWebServerApplicationContext) {
            ServletWebServerApplicationContext context =
                (ServletWebServerApplicationContext) event.getApplicationContext();

            if (context.getWebServer() instanceof TomcatWebServer) {
                TomcatWebServer tomcatWebServer = (TomcatWebServer) context.getWebServer();
                Connector connector = tomcatWebServer.getTomcat().getConnector();
                connector.pause();

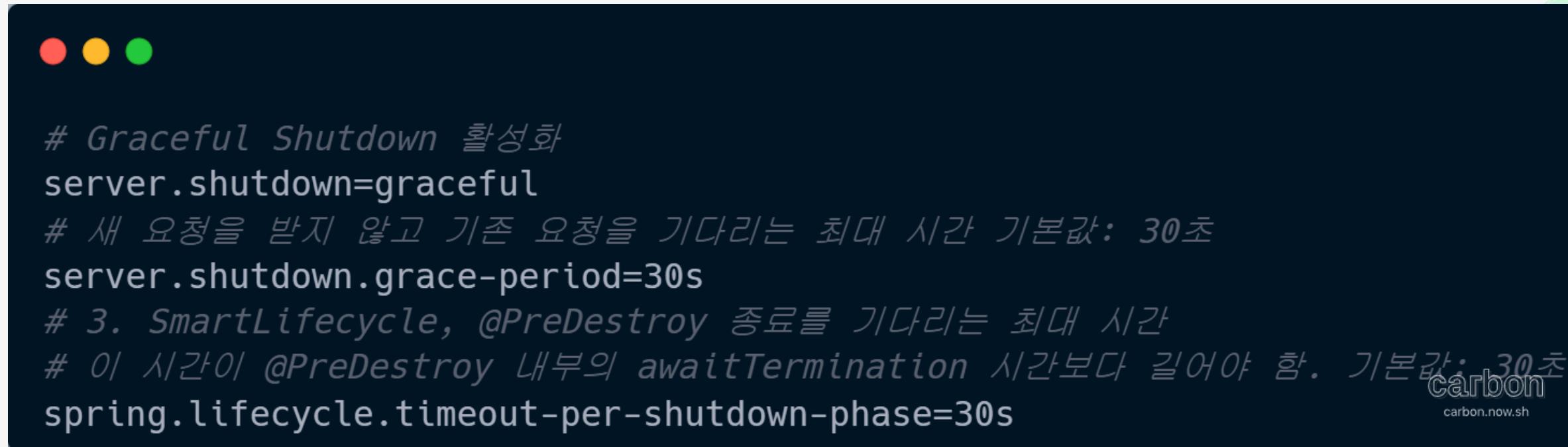
                Executor executor = connector.getProtocolHandler().getExecutor();
                if (executor instanceof ExecutorService) {
                    ExecutorService executorService = (ExecutorService) executor;
                    executorService.shutdown();

                    try {
                        if (!executorService.awaitTermination(GRACE_PERIOD_SECONDS, TimeUnit.SECONDS)) {
                            executorService.shutdownNow();
                        }
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    }
                }
            }
        }
    }
}
```

서비스 성장기

우아한 종료 Graceful Shutdown

- Tomcat ThreadPool (Spring Boot 2.3 이후)



```
# Graceful Shutdown 활성화
server.shutdown=graceful
# 새 요청을 받지 않고 기존 요청을 기다리는 최대 시간 기본값: 30초
server.shutdown.grace-period=30s
# 3. SmartLifecycle, @PreDestroy 종료를 기다리는 최대 시간
# 0/ 시간이 @PreDestroy 내부의 awaitTermination 시간보다 길어야 함. 기본값: 30초
spring.lifecycle.timeout-per-shutdown-phase=30s
```

The screenshot shows a terminal window with a dark background and three colored dots (red, yellow, green) at the top left. The terminal displays configuration code for a graceful shutdown. The code includes annotations in Korean explaining the configuration: '# Graceful Shutdown 활성화' (Enable Graceful Shutdown), '# 새 요청을 받지 않고 기존 요청을 기다리는 최대 시간 기본값: 30초' (Default maximum time to wait for existing requests before accepting new ones: 30 seconds), and '# 3. SmartLifecycle, @PreDestroy 종료를 기다리는 최대 시간' (Default maximum time to wait for the completion of the @PreDestroy lifecycle phase). The code uses the 'server' and 'spring.lifecycle' properties in the application's configuration file.

서비스 성장기

우아한 종료 Graceful Shutdown

- Custom Bean (with ThreadPoolExecutor)

```
● ● ●

@Component
public class CustomTaskProcessor {
    private final ExecutorService executorService = Executors.newFixedThreadPool(5);
    private static final long GRACE_PERIOD_SECONDS = 30;

    @PreDestroy
    public void gracefulShutdown() {
        executorService.shutdown();

        try {
            if (!executorService.awaitTermination(GRACE_PERIOD_SECONDS, TimeUnit.SECONDS)) {
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            executorService.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
}
```

carbon
carbon.now.sh

서비스 성장기

DB 커넥션 풀 최적화 DB Connection Pool Optimization

서비스 성장기

DB 커넥션 풀 최적화 DB Connection Pool Optimization

DB Connection Pool (HikariCP)

데이터베이스에서 적용할 Connection은 $(\text{core_count} \times 2) + \text{Effective_spindle_count(HDD)}$

- PostgreSQL의 Number of Database Connections 출처 (HikariCP도 여길 참조)
- Core 수 == Connection 수가 이상적인 수치로 보이거나, Disk I/O, Network Buffer, RTT도 함께 고려해야 함, 즉 Core 수 보다 많지만 CPU 경쟁 상태로 인한 오버헤드는 최소화되는 최적 값을 탐색
 - 모니터링 (메트릭 수집 용), 제어 커넥션 수는 제외

서비스 성장기

DB 커넥션 풀 최적화 DB Connection Pool Optimization

DB Connection Pool (HikariCP)

데이터베이스에서 적용할 Connection은 $(\text{core_count} \times 2) + \text{Effective_spindle_count(HDD)}$

- PostgreSQL의 Number of Database Connections 출처 (HikariCP도 여길 참조)
- Core 수 == Connection 수가 이상적인 수치로 보이나, Disk I/O, Network Buffer, RTT도 함께 고려해야 함, 즉 Core 수 보다 많지만 CPU 경쟁 상태로 인한 오버헤드는 최소화되는 최적 값을 탐색
 - 모니터링 (메트릭 수집 용), 제어 커넥션 수는 제외
- 위 공식은 HDD를 사용한 벤치마크 기준이며, SSD를 사용한 경우는 I/O(Spindle I/O)로 인한 DISK 차단 및 탐색 비용이 낮으므로 더 낮은 값으로 조정할 수 있음
 - e.g. 4 core, 1개의 HDD => 9 = $((4 * 2) + 1)$

서비스 성장기

DB 커넥션 풀 최적화 DB Connection Pool Optimization

DB Connection Pool (HikariCP)

데이터베이스에서 적용할 Connection은 $(\text{core_count} \times 2) + \text{Effective_spindle_count(HDD)}$

- PostgreSQL의 Number of Database Connections 출처 (HikariCP도 여길 참조)
- Core 수 == Connection 수가 이상적인 수치로 보이나, Disk I/O, Network Buffer, RTT도 함께 고려해야 함, 즉 Core 수 보다 많지만 CPU 경쟁 상태로 인한 오버헤드는 최소화되는 최적 값을 탐색
 - 모니터링 (메트릭 수집 용), 제어 커넥션 수는 제외
- 위 공식은 HDD를 사용한 벤치마크 기준이며, SSD를 사용한 경우는 I/O(Spindle I/O)로 인한 DISK 차단 및 탐색 비용이 낮으므로 더 낮은 값으로 조정할 수 있음
 - e.g. 4 core, 1개의 HDD => 9 = ((4 * 2) + 1)

서비스에서 교착 상태를 피하기 위한 Pool 구성은 $\text{TXn} \times (\text{CNm} - 1) + 1$ 공식

- 최적의 Pool 구성은 아니지만, 교착 상태를 피하기 위한 최소한의 크기
- 한 스레드 내 동시 커넥션 수(CN)를 **최소한으로 유지하는 것이 바람직함**

서비스 성장기

DB 커넥션 풀 최적화 DB Connection Pool Optimization

DB Connection Pool (HikariCP)

JDBC Driver 구성 시 최적화

- `rewriteBatchedStatements=true`를 통한 Insert 일괄 처리
 - Statement.getGeneratedKeys(), INSERT ... ON DUPLICATE KEY UPDATE에 대한 제약이나 SQL 주입이 허용될 수 있다라는 주의 사항이 있지만, JPA에서는 상관없음
- `cacheResultSetMetadata=true`를 통한 ResultSetMetaData 캐싱
 - ResultSetMetaData는 컬럼 수, 정보(SQL 타입, Null 여부 등), 읽기 전용 여부를 포함

서비스 성장기

DB 커넥션 풀 최적화 DB Connection Pool Optimization

DB Connection Pool (HikariCP)

JDBC Driver 구성 시 최적화

- `rewriteBatchedStatements=true`를 통한 Insert 일괄 처리
 - Statement.getGeneratedKeys(), INSERT ... ON DUPLICATE KEY UPDATE에 대한 제약이나 SQL 주입이 허용될 수 있다라는 주의 사항이 있지만, JPA에서는 상관없음
- `cacheResultSetMetadata=true`를 통한 ResultSetMetaData 캐싱
 - ResultSetMetaData는 컬럼 수, 정보(SQL 타입, Null 여부 등), 읽기 전용 여부를 포함
- `cacheServerConfiguration=true`를 통한 메모리 임시 테이블 생성 최소화
- `elideSetAutoCommits=true`를 통한 autoCommit 설정 최적화 (트랜잭션 시작시 통신 X)
- 그 외에도 언급하지 않은 많은 옵션이 있으며, 항상 서비스에 적합한 옵션인지 테스트가 필요함

서비스 성장기

DB 커넥션 풀 최적화 DB Connection Pool Optimization

DB Connection Pool (HikariCP)

다시 강조하자면, 앞선 공식을 활용하더라도 결국 최적의 값을 찾는 단계가 필요

- 서비스 규모 추정을 바탕으로 성능 테스트를 진행하며, 최적의 성능이 나오는 구성을 찾아야 함
 - Connection을 추가했음에도 변화가 없거나 오히려 처리량이 떨어지는 지점을 확인
- 성능 테스트 시에는 **flexy-pool** 라이브러리를 활용해볼 수 있음
 - IncrementPoolOnTimeoutConnectionAcquisitionStrategy

서비스 성장기

DB 커넥션 풀 최적화 DB Connection Pool Optimization

DB Connection Pool (HikariCP)

다시 강조하자면, 앞선 공식을 활용하더라도 결국 최적의 값을 찾는 단계가 필요

- 서비스 규모 추정을 바탕으로 성능 테스트를 진행하며, 최적의 성능이 나오는 구성을 찾아야 함
 - Connection을 추가했음에도 변화가 없거나 오히려 처리량이 떨어지는 지점을 확인
- 성능 테스트 시에는 flexy-pool 라이브러리를 활용해볼 수 있음
 - IncrementPoolOnTimeoutConnectionAcquisitionStrategy

커넥션 생성 빈도를 낮추기 위해 max-lifetime을 무작정 길게 설정하지 말 것

- DB failover 시 서비스가 다른 데이터 소스를 바라보는 시점이 늦춰져 복구 시간이 더 길어질 수 있음
- 심한 경우, DB failover를 수행할 때 서비스도 재배포해야 함
- API 서버와 DB 간 방화벽이나 프록시 계층이 있는 경우, Stale Connection 문제가 발생할 수 있음

서비스 성장기

DB 커넥션 풀 최적화 DB Connection Pool Optimization

DB Connection Pool (HikariCP)

다시 강조하자면, 앞선 공식을 활용하더라도 결국 최적의 값을 찾는 단계가 필요

- 서비스 규모 추정을 바탕으로 성능 테스트를 진행하며, 최적의 성능이 나오는 구성을 찾아야 함
 - Connection을 추가했음에도 변화가 없거나 오히려 처리량이 떨어지는 지점을 확인
- 성능 테스트 시에는 flexy-pool 라이브러리를 활용해볼 수 있음
 - IncrementPoolOnTimeoutConnectionAcquisitionStrategy

커넥션 생성 빈도를 낮추기 위해 max-lifetime을 무작정 길게 설정하지 말 것

- DB failover 시 서비스가 다른 데이터 소스를 바라보는 시점이 늦춰져 복구 시간이 더 길어질 수 있음
- 심한 경우, DB failover를 수행할 때 서비스도 재배포해야 함
- API 서버와 DB 간 방화벽이나 프록시 계층이 있는 경우, Stale Connection 문제가 발생할 수 있음

범위에 약간 벗어난 이야기지만, 특정 Connection을 불필요하게 오래 점유하는 상황은 피할 것

- e.g. API 호출 전, 후로 Connection을 물고 있는 비즈니스 로직

서비스 성장기

경쟁 상태 Race Condition

서비스 성장기

경쟁 상태 Race Condition

시스템의 실질적인 동작이 제어할 수 없는 **두 개 이상의 이벤트 순서나 타이밍**에 따라 예기치 않거나 일관되지 않는 결과를 초래하는 상태

서비스 성장기

경쟁 상태 Race Condition

시스템의 실질적인 동작이 제어할 수 없는 두 개 이상의 이벤트 순서나 타이밍에 따라 예기치 않거나 일관되지 않는 결과를 초래하는 상태

하나의 공유 자원에 대해 원자적 연산이 이루어지지 않는 것

서비스 성장기

경쟁 상태 Race Condition

- Idempotency

서비스 성장기

경쟁 상태 Race Condition

- Idempotency

최초 적용 이후에 여러 번 더 적용하더라도 결과를 변경시키지 않는 속성

서비스 성장기

경쟁 상태 Race Condition

- Idempotency

조회(GET)나 삭제(DELETE)처럼 기본적으로 멱등성이 지켜지는 요청이 아닌 경우에도, 만족하게끔 구현 가능

- 요청이나 파일의 해시(Hash) 값, 시간 값 또는 특정 키를 생성하여 중복 여부를 검증하고, 최초의 요청만 반영하도록 처리하는 것

서비스 성장기

경쟁 상태 Race Condition

- Idempotency

조회(GET)나 삭제(DELETE)처럼 기본적으로 멱등성이 지켜지는 요청이 아닌 경우에도, 만족하게끔 구현 가능

- 요청이나 파일의 해시(Hash) 값, 시간 값 또는 특정 키를 생성하여 중복 여부를 검증하고, 최초의 요청만 반영하도록 처리하는 것
- e.g. 생성된 해시 값 / 키를 포함한 데이터를 DB에 저장할 때, Unique constraint를 설정하여 SQLIntegrityConstraintViolationException 예외를 발생시키고, 이를 Catch 후 무시
 - SqS, Kafka 등 Listener 구현체 사용 시, 에러를 캐치하여 Acknowledge를 수동 처리

서비스 성장기

경쟁 상태 Race Condition

- Locking

서비스 성장기

경쟁 상태 Race Condition

- Locking

여러 실행 스레드가 하나의 공유 자원에 동시에 상태를 수정하거나 접근하는 것을 방지하는 동기화 요소

서비스 성장기

경쟁 상태 Race Condition

- Locking

단일 스레드 환경처럼 순차적인 처리가 보장되지 않는 동시성(Concurrency) 환경에서도, 데이터의 정합성과 일관성을 만족하게끔 구현 가능

- 공유 자원에 접근하는 임계 영역(Critical Section)을 설정한 뒤, 하나의 스레드(또는 프로세스)만 접근 권한을 갖도록 제어하고 다른 접근은 대기하도록 처리

서비스 성장기

경쟁 상태 Race Condition

- Locking

단일 스레드 환경처럼 순차적인 처리가 보장되지 않는 동시성(Concurrency) 환경에서도, 데이터의 정합성과 일관성을 만족하게끔 구현 가능

- 공유 자원에 접근하는 임계 영역(Critical Section)을 설정한 뒤, 하나의 스레드(또는 프로세스)만 접근 권한을 갖도록 제어하고 다른 접근은 대기하도록 처리
- e.g. Java에서 synchronized 키워드나 ReentrantLock을 사용하여 특정 메서드나 코드 블록을 감싸고, 해당 영역에 먼저 진입한 스레드가 작업을 마칠 때(락을 해제할 때)까지 다른 스레드의 접근을 차단

서비스 성장기

경쟁 상태 Race Condition

- Locking

단일 스레드 환경처럼 순차적인 처리가 보장되지 않는 동시성(Concurrency) 환경에서도, 데이터의 정합성과 일관성을 만족하게끔 구현 가능

- 공유 자원에 접근하는 임계 영역(Critical Section)을 설정한 뒤, 하나의 스레드(또는 프로세스)만 접근 권한을 갖도록 제어하고 다른 접근은 대기하도록 처리
- e.g. Java에서 synchronized 키워드나 ReentrantLock을 사용하여 특정 메서드나 코드 블록을 감싸고, 해당 영역에 먼저 진입한 스레드가 작업을 마칠 때(락을 해제할 때)까지 다른 스레드의 접근을 차단
- e.g. DB 트랜잭션에서 SELECT FOR UPDATE (Pessimistic Lock) 구문을 사용하여 특정 레코드에 명시적인 락을 걸고, 해당 트랜잭션이 커밋/롤백되기 전까지 다른 트랜잭션이 해당 레코드에 접근하거나 수정하는 것을 방지 (또는 Named Lock 사용)

서비스 성장기

경쟁 상태 Race Condition

- Locking

단일 스레드 환경처럼 순차적인 처리가 보장되지 않는 동시성(Concurrency) 환경에서도, 데이터의 정합성과 일관성을 만족하게끔 구현 가능

- 공유 자원에 접근하는 임계 영역(Critical Section)을 설정한 뒤, 하나의 스레드(또는 프로세스)만 접근 권한을 갖도록 제어하고 다른 접근은 대기하도록 처리
- e.g. Java에서 synchronized 키워드나 ReentrantLock을 사용하여 특정 메서드나 코드 블록을 감싸고, 해당 영역에 먼저 진입한 스레드가 작업을 마칠 때(락을 해제할 때)까지 다른 스레드의 접근을 차단
- e.g. DB 트랜잭션에서 SELECT FOR UPDATE (Pessimistic Lock) 구문을 사용하여 특정 레코드에 명시적인 락을 걸고, 해당 트랜잭션이 커밋/롤백되기 전까지 다른 트랜잭션이 해당 레코드에 접근하거나 수정하는 것을 방지 (또는 Named Lock 사용)
- e.g. 분산 시스템 환경에서 Zookeeper, Redis 등의 외부 공유 저장소를 이용해 Lock을 획득하고 해제하도록 제한하고, 락을 획득하지 못한 서비스는 대기하거나 재시도 처리

서비스 성장기

경쟁 상태 Race Condition

- Locking (Distributed Lock 적용 시 주의점)

락을 소유한 서버의 예측 불가능한 중단이나 락 정보를 관리하는 외부 저장소의 장애로 인해, 락의 소유권이 잘못 만료되거나 중복으로 발급될 수 있음

- 락을 획득한 서버 인스턴스가 긴 GC(Stop-the-world)나 시스템 호출 등으로 일시 중지된 상태에서 락의 만료 시간(TTL)이 만료되는 경우 다른 서버가 만료된 락을 획득해 두 서버가 동시에 임계 영역을 실행하는 문제

서비스 성장기

경쟁 상태 Race Condition

- Locking (Distributed Lock 적용 시 주의점)

락을 소유한 서버의 예측 불가능한 중단이나 락 정보를 관리하는 외부 저장소의 장애로 인해, 락의 소유권이 잘못 만료되거나 중복으로 발급될 수 있음

- 락을 획득한 서버 인스턴스가 긴 GC(Stop-the-world)나 시스템 호출 등으로 일시 중지된 상태에서 락의 만료 시간(TTL)이 만료되는 경우 다른 서버가 만료된 락을 획득해 두 서버가 동시에 임계 영역을 실행하는 문제
- Redis 인스턴스 또는 Sentinel 기반 구조를 저장소로 사용할 때 락을 발급한 후 이 정보가 팔로워 인스턴스로 복제되기 전에 장애가 발생하는 경우, 리더로 승격된 팔로워 인스턴스에는 락 정보가 없는 상태이므로, 다른 서버에게 중복으로 락을 발급하여 정합성이 깨질 수 있음.

서비스 성장기

경쟁 상태 Race Condition

- Locking (Distributed Lock 적용 시 주의점)

락을 소유한 서버의 예측 불가능한 중단이나 락 정보를 관리하는 외부 저장소의 장애로 인해, 락의 소유권이 잘못 만료되거나 중복으로 발급될 수 있음

- 락을 획득한 서버 인스턴스가 긴 GC(Stop-the-world)나 시스템 호출 등으로 일시 중지된 상태에서 락의 만료 시간(TTL)이 만료되는 경우 다른 서버가 만료된 락을 획득해 두 서버가 동시에 임계 영역을 실행하는 문제
- Redis 인스턴스 또는 Sentinel 기반 구조를 저장소로 사용할 때 락을 발급한 후 이 정보가 팔로워 인스턴스로 복제되기 전에 장애가 발생하는 경우, 리더로 승격된 팔로워 인스턴스에는 락 정보가 없는 상태이므로, 다른 서버에게 중복으로 락을 발급하여 정합성이 깨질 수 있음.
 - Redis 팀에서는 Redis Cluster 기반의 Distributed Lock 알고리즘인 Redlock을 제시하였지만 (다수의 Redis Leader에 key 분배) 앞서 언급한 애플리케이션 중단 문제나 네트워크 지연 문제를 해소하지 못함
 - 궁극적으로 Zookeeper 같은 합의 알고리즘 기반 시스템을 사용하는 것이 맞다고 봄

서비스 트래픽 성장과 장애는 어떻게 대비해야 하는가?

서비스 성숙기

서비스 성숙기

캐시 최적화 Cache Optimization

- Local Cache

서비스 성숙기

캐시 최적화 Cache Optimization

- Local Cache

프로세스 내 메모리에 데이터를 저장하기에, 네트워크 경유를 하지 않아
빠른 접근 속도로 데이터 조회 가능

서비스 성숙기

캐시 최적화 Cache Optimization

- Local Cache

데이터 일관성 문제를 발생시킴에도 사용하는 이유는?

서비스 성숙기

캐시 최적화 Cache Optimization

- Local Cache

데이터 일관성 문제를 발생시킴에도 사용하는 이유는?

클라이언트 / 서버에서 접근하는 데이터에 대해, 더 빠른 자연 시간으로 제공하는 목적

- 데이터 서빙, 피쳐 플래그, 설정 정보 제공 등

서비스 성숙기

캐시 최적화 Cache Optimization

- Local Cache

데이터 일관성 문제를 발생시킴에도 사용하는 이유는?

클라이언트 / 서버에서 접근하는 데이터에 대해, 더 빠른 자연 시간으로 제공하는 목적

- 데이터 서빙, 피쳐 플래그, 설정 정보 제공 등

Global Cache의 장애, 응답 지연으로 인한 여파를 최소화하는 목적

- 장애가 발생한 경우에 발생할 수 있는 DB에 대한 Cache Stampede 예방 & 장애 복구 시간 확보
- 응답 지연이 발생한 경우에 받는 영향 최소화

서비스 성숙기

캐시 최적화 Cache Optimization

- Local Cache

데이터 일관성 문제를 발생시킴에도 사용하는 이유는?

클라이언트 / 서버에서 접근하는 데이터에 대해, 더 빠른 자연 시간으로 제공하는 목적

- 데이터 서빙, 피쳐 플래그, 설정 정보 제공 등

Global Cache의 장애, 응답 지연으로 인한 여파를 최소화하는 목적

- 장애가 발생한 경우에 발생할 수 있는 DB에 대한 Cache Stampede 예방 & 장애 복구 시간 확보
- 응답 지연이 발생한 경우에 받는 영향 최소화

Global Cache의 부하를 감소시키는 목적

- 트래픽 규모가 커질수록 Global Cache가 모든 트래픽을 감당할 수는 없음
- 네트워크 대역폭 이슈 (뒤에서 다룰 Burst Traffic 등)을 해소하는 목적
- 장기적으로 다층적 캐시 레이어 구성이 필요해짐

서비스 성숙기

캐시 최적화 Cache Optimization

- Local Cache

최근에는 Caffeine, Guava Cache를 자주 사용

동시성 처리, 만료 정책 등에 따라 사용 라이브러리를 선택 & Global Cache와의 동기화 방식 구성 필요

- Asynchronous Pulling이나 Redis Pub / Sub 또는 함께 적용하는 사례가 일반적이라고 생각
 - Kafka 등으로 Local Cache를 동기화하는 것은 조직 내 공용 Kafka Cluster가 있는 경우 고려해야한다고 봄
- Redis 6을 사용 시, jedis나 Lettuce Client를 통해 Client-side Caching를 적용하는 방향도 있음

서비스 성숙기

캐시 최적화 Cache Optimization

- Local Cache (Jedis)



```
HostAndPort node = HostAndPort.from("localhost:6379");
JedisClientConfig clientConfig = DefaultJedisClientConfig.builder()
    .resp3() // RESP3 protocol required
    .build();

CacheConfig cacheConfig = getCacheConfig();
Cache cache = CacheFactory.getCache(cacheConfig);

try (UnifiedJedis client = new UnifiedJedis(node, clientConfig, cache)) {
    client.set("foo", "bar");
    client.get("foo");
    client.get("foo"); // Cache hit
}
```

carbon
carbon.now.sh

서비스 성숙기

캐시 최적화 Cache Optimization

- Local Cache (lettuce)

```
// CacheAccessor 인터페이스를 직접 구현하여야 함
CacheAccessor<String, String> caffeineAccessor = new CaffeineCacheAccessor<>(10000, 10);
StatefulRedisConnection<String, String> connection = redisClient.connect();

CacheFrontend<String, String> frontend = ClientSideCaching.enable(
    caffeineAccessor,
    connection,
    TrackingArgs.Builder.bcast().build()
);

String value = frontend.get("mykey");
String value = frontend.get("mykey"); // Cache Hit
```

서비스 성숙기

캐시 최적화 Cache Optimization

- Compression

서비스 성숙기

캐시 최적화 Cache Optimization

- Compression

Cache에는 자주 조회되는 데이터만 저장하는 것이 이상적이다.

- **파레토 법칙** (전체 결과의 약 80%가 전체 원인의 약 20%에서 발생한다)

서비스 성숙기

캐시 최적화 Cache Optimization

- Compression

Cache에는 자주 조회되는 데이터만 저장하는 것이 이상적이다.

- 파레토 법칙 (전체 결과의 약 80%가 전체 원인의 약 20%에서 발생한다)

하지만 상품 검색이나 속성, 의미론적 유사도, 개인화 추천 등의 기능을 통해 데이터를 제공하는 경우에는 나머지 80%의 원인 또한 자주 조회되는 경향을 보임 (Long Tail Problem)

- 이 경우 대부분의 데이터를 캐시에 올려놓고 최대한 DB에 접근하지 않도록 구성하기도 함
- 메모리에 적재된 데이터가 비대해지면서, TLB에 모든 페이지를 캐싱할 수 없어 Miss 확률이 늘어나더라도, 디스크 접근에 따른 자연 시간은 최소화하는 목적
 - TLB Hit과 Miss의 성능 차이는 100배지만, Disk I/O 보다도 100배 이상 빠름

서비스 성숙기

캐시 최적화 Cache Optimization

- **Compression**

Cache에는 자주 조회되는 데이터만 저장하는 것이 이상적이다.

- **파레토 법칙** (전체 결과의 약 80%가 전체 원인의 약 20%에서 발생한다)

하지만 상품 검색이나 속성, 의미론적 유사도, 개인화 추천 등의 기능을 통해 데이터를 제공하는 경우에는 나머지 80%의 원인 또한 자주 조회되는 경향을 보임 (Long Tail Problem)

- 이 경우 대부분의 데이터를 캐시에 올려놓고 최대한 DB에 접근하지 않도록 구성하기도 함
- 메모리에 적재된 데이터가 비대해지면서, TLB에 모든 페이지를 캐싱할 수 없어 Miss 확률이 늘어나더라도, 디스크 접근에 따른 자연 시간은 최소화하는 목적
 - TLB Hit과 Miss의 성능 차이는 100배지만, Disk I/O 보다도 100배 이상 빠름
- **CDN** (Content Delivery Network)도 유사한 문제를 가지고 있음 (Video, Image, HTML, CSS..)
 - **Compression** 외에 **Edge Caches**, **Active Cache**(Asynchronously pulling 등) 등 활용

서비스 성숙기

캐시 최적화 Cache Optimization

- Compression

대용량 & 대규모 트래픽 시스템을 운영하는 경우, Network Bandwidth를 모두 점유하는 Burst Traffic 문제를 겪을 수 있음

- 특정 이벤트(선착순 티켓팅, 시즌 세일 등)로 인해 대량의 트래픽이 몰리는 경우
- 다수의 동영상이나 대용량 파일 업로드로 인한 I/O가 발생하는 경우

서비스 성숙기

캐시 최적화 Cache Optimization

- Compression

대용량 & 대규모 트래픽 시스템을 운영하는 경우, Network Bandwidth를 모두 점유하는 Burst Traffic 문제를 겪을 수 있음

- 특정 이벤트(선착순 티켓팅, 시즌 세일 등)로 인해 대량의 트래픽이 몰리는 경우
- 다수의 동영상이나 대용량 파일 업로드로 인한 I/O가 발생하는 경우

이 경우 DB, Redis 등의 솔루션 내에서 색인 등을 통해 제공할 데이터를 빠르게 조회 및 필터링하고 있다
라도 네트워크 계층으로 인해 응답을 받는 시간이 지연될 수 있음을 의미

- AWS ElastiCache의 경우 NetworkBandwidthAllowanceExceeded, NetworkBytesOut 등 모니터링 필요
- DB, Redis 복제본이 많을수록 대역폭 이슈를 신경써야 함 (Replication 또한 NetworkBytes)

서비스 성숙기

캐시 최적화 Cache Optimization

- Compression

앞선 상황을 대비하거나 개선하기 위해, 캐시 데이터에 대한 비손실 압축을 적용

- gzip, Snappy(Google), LZ4(DuckDB, Cassandra), ztsd(Facebook) 등
 - gzip은 높은 압축률에 초점을 두어 콜드 데이터를 압축하는데 자주 활용
 - Snappy는 상대적으로 CPU를 덜 사용하면서 빠른 속도, 250MB/s 압축, 압축률 낮음
 - LZ4는 CPU를 더 사용하게 설정하여 빠른 속도, 500MB/s 이상 압축 / Gb/s 압축 해제
 - ztsd는 빠른 속도와 압축률의 균형을 유지, 500MB/s 이상 압축 ~ 1500MB/s 압축 해제
 - CPU 사용량은 lz4(default), Snappy보다 높음

서비스 성숙기

캐시 최적화 Cache Optimization

- **Compression (snappy)**

```
public class SnappyJsonRedisSerializer<T> implements RedisSerializer<T> {
    private final ObjectMapper objectMapper;
    private final Class<T> type;
    // 생성자

    @Override
    public byte[] serialize(T payload) throws SerializationException {
        if (payload == null) {
            return new byte[0];
        }
        try {
            byte[] jsonBytes = objectMapper.writeValueAsBytes(payload);
            return Snappy.compress(jsonBytes);
        } catch (IOException e) {
            throw new SerializationException(e.getLocalizedMessage(), e);
        }
    }

    @Override
    public T deserialize(byte[] bytes) throws SerializationException {
        if (bytes == null || bytes.length == 0) {
            return null;
        }
        try {
            byte[] jsonBytes = Snappy.uncompress(bytes);
            return objectMapper.readValue(jsonBytes, type);
        } catch (IOException e) {
            throw new SerializationException(e.getLocalizedMessage(), e);
        }
    }
}
```

carbon
carbon.now.sh

서비스 성장기

재시도 Retry

서비스 성장기

재시도 Retry

일시적인 오류로 작업이 실패하였다고 판단했을 때 적절한
시간 여유를 가지고 작업을 요청하는 행위

서비스 성장기

재시도 Retry

- backoff & attempts

backoff(ms) & attempts(count)의 설정 기준

- Latency Sensitivity
- 광고 서빙, 전시, 피드, 채팅 서비스 등 사용자에게 밀접하고, 빠른 피드백을 제공해주는 것이 사용자 경험에 영향을 많이 미치는 서비스일 수록 backoff를 짧게 설정해야함
 - e.g. 광고 서비스의 p95는 200ms 이내. backoff <= 50ms, attempts <= 2

서비스 성장기

재시도 Retry

- backoff & attempts

backoff(ms) & attempts(count)의 설정 기준

- Latency Sensitivity
- 광고 서빙, 전시, 피드, 채팅 서비스 등 사용자에게 밀접하고, 빠른 피드백을 제공해주는 것이 사용자 경험에 영향을 많이 미치는 서비스일 수록 backoff를 짧게 설정해야 함
 - e.g. 광고 서비스의 p95는 200ms 이내. backoff \leq 50ms, attempts \leq 2
- 이메일 발송, 결제 서비스 등 비동기 처리가 가능한, 사용자에게 피드백 제공을 빨리 하지 않더라도 사용자 경험에 영향을 적게 미치는 서비스는 backoff를 길게 설정할 수 있음
 - 무조건 처리되어야 하는 신뢰성이 중요한 서비스 등
 - 이메일 발송, 결제 서비스 등은 Optimistic UI를 적절히 활용하기도 함
 - e.g. 결제 서비스의 p95는 2s 이내. backoff \leq 300ms, attempts \leq 5

서비스 성장기

재시도 Retry

- Retry Storm

서비스 성장기

재시도 Retry

- Retry Storm

일시적인 장애가 발생했을 때, 수많은 클라이언트, 서버가 거의 동시에 재시도를 수행하여 트래픽이 폭증하고 장애를 더욱 심화시키는 현상

서비스 성장기

재시도 Retry

- Retry Storm

Latency Sensitivity가 높은 서비스는 어떻게 대응할까?

- Circuit Breaker : 빠른 backoff & attempts 처리 이후 지정된 임계치를 바탕으로 장애를 감지해 일정 기간 동안 재시도 요청을 보내지 않음으로써 장애 격리 (우리 서비스를 보호하는 방향)
 - Request Based, Time Based / Closed → Open → Half Open → Closed
 - 다시 시도해달라는 통지 or 기본 값 처리

서비스 성장기

재시도 Retry

- Retry Storm

Latency Sensitivity가 높은 서비스는 어떻게 대응할까?

- Circuit Breaker : 빠른 backoff & attempts 처리 이후 지정된 임계치를 바탕으로 장애를 감지해 일정 기간 동안 재시도 요청을 보내지 않음으로써 장애 격리 (우리 서비스를 보호하는 방향)
 - Request Based, Time Based / Closed → Open → Half Open → Closed
 - 다시 시도해달라는 통지 or 기본 값 처리
- Bulkhead : 정해진 요청 수(Semaphore, Retry Budget)나 격리된 전용 리소스 풀을 이용해 대기 중인 모든 요청을 한 번에 재시도 처리하지 않도록 방지하고 & 실패 처리하여 클라이언트에게 결과 통지 (상대 서비스를 보호하는 방향)

서비스 성장기

재시도 Retry

- Retry Storm

Latency Sensitivity가 낮은 서비스는 어떻게 대응할까?

- Exponential Backoff & Jitter

- 현재 backoff 양에 비례하는 비율로 증가하는 Exponential Backoff와 일정 범위 안의 난수 값인 Jitter를 합산하여 긴 시간 동안 트래픽을 분산하여 장애 회복이 될 수 있도록 지원
- e.g. Exponential Backoff = 100ms, 2의 제곱 // $-150\text{ms} \leq \text{jitter} \leq 150\text{ms}$
 - A : 115ms \rightarrow 190ms \rightarrow 440ms \rightarrow 750ms \rightarrow 1720ms
 - B : 90ms \rightarrow 230ms \rightarrow 380ms \rightarrow 850ms \rightarrow 1500ms

서비스 성장기

재시도 Retry

- Retry Storm

Latency Sensitivity가 낮은 서비스는 어떻게 대응할까?

- Exponential Backoff & Jitter
 - 현재 backoff 양에 비례하는 비율로 증가하는 Exponential Backoff와 일정 범위 안의 난수 값인 Jitter를 합산하여 긴 시간 동안 트래픽을 분산하여 장애 회복이 될 수 있도록 지원
 - e.g. Exponential Backoff = 100ms, 2의 제곱 // $-150\text{ms} \leq \text{jitter} \leq 150\text{ms}$
 - A : 115ms \rightarrow 190ms \rightarrow 440ms \rightarrow 750ms \rightarrow 1720ms
 - B : 90ms \rightarrow 230ms \rightarrow 380ms \rightarrow 850ms \rightarrow 1500ms
- Dead Letter Queue
 - 비동기 아키텍처를 채택하였고 결과적 일관성 보장을 해야하는 경우에 backoff & attempts를 모두 수행했음에도 실패한 요청들을 별도의 Queue에 일정 기간 보관 후 Scheduler 또는 Backoffice 등을 통해 직접 처리

정리

정리

서비스 태동기

- 페르미의 추정 Fermi Estimate
- 모니터링 & 관측 가능성 Monitoring & Observability

정리

서비스 태동기

- 페르미의 추정 Fermi Estimate
- 모니터링 & 관측 가능성 Monitoring & Observability

서비스 성장기

- 콜드 스타트 Cold Start
- 우아한 종료 Graceful Shutdown
- DB 커넥션 풀 최적화 DB Connection Pool Optimization
- 경쟁 상태 Race Condition

정리

서비스 태동기

- 페르미의 추정 Fermi Estimate
- 모니터링 & 관측 가능성 Monitoring & Observability

서비스 성장기

- 콜드 스타트 Cold Start
- 우아한 종료 Graceful Shutdown
- DB 커넥션풀 최적화 DB Connection Pool Optimization
- 경쟁 상태 Race Condition

서비스 성숙기

- 캐시 최적화 Cache Optimization
- 재시도 Retry

E.O.D