

SIPE 4기 1차 미션

Spring 두런두런

팀 소개

이윤준



김세정



김수빈



장준환



박영준



김희동



유지예



운영



Index

- 발제 배경
- 스터디 주제
- 미션 목표
- 기대 효과
- 활동 요약

개요

2024 당근 테크 밋업

Tech MeetUP

행사 일정: 10.7 | 코엑스 컨퍼런스룸(남) 3F
참가 신청: 9.2-9.6



SPRING CAMP

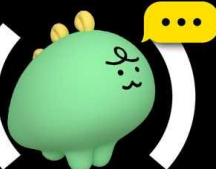

toss slash

W O O W A T E C H

우아한 Te



우아한형제들의 기술조직 이야기를 전함

if () → 

발제 배경

- 영상은 저장만 해두고 못 본 적, 있지 않나요?
- 이제는 '스크랩'이 아닌 '실행'으로!
- 실무 적용 시 마주칠 현실적인 문제까지 탐색

스터디 주제

- 국내외 컨퍼런스에서 발표된 자바/스프링 생태계 기술 주제를 실습
- 단순 시청이 아닌 직접 실습 + 토론 중심

기대효과

- 기술 스택에 대한 실전 감각 향상
- 컨퍼런스 발표를 내 것으로 소화
- 동료들과의 토론을 통한 다양한 시각 확보
- 블로그/포트폴리오/실무 개선 등 정량적 성장 유도

미션 목표

- 자바/스프링 및 백엔드 아키텍처에 대한 인사이트 확보
- 학습에 그치지 않고 실무/개인 프로젝트에 반영
- 주차별 개인 학습 후 학습 내용 토론

UHO
UHO

활동 개요

- Java Performance Update
- Stream Gatherers
- Virtual Thread
- Distributed Lock
- MySQL InnoDB

Java Performance Update



- 성능은 어떻게 측정하는가?
- JVM 의 **just-in-time compilation**
- C2 컴파일러의 최적화
 - JDK-8318446: C2 - "MergeStores"
 - JDK-8340821: FFM API Bulk Operations
 - JDK-8180450: Secondary Super Cache Scaling
 - JDK-8336856: String Concatenation
 - JEP 474: ZGC: Generational Mode by Default
 - JEP 450: Compact Object Headers (Experimental)

Stream Gatherers

- **Gatherer** 는 무엇인가
 - **Collector** 와 비교
 - 무엇이 가능한가
- **Gatherer** 구조 뜯어보기
- **Gatherer Custom**

```
java.util.stream.Gatherer<T, A, R>
```

```
Supplier<A> initializer()
```

Provides the state, if any, to be used during evaluation of the Gatherer.

```
Integrator<A, T, R> integrator()
```

Each input element is applied to the integrator, together with the state, and a Downstream handle for as long as it returns true.

```
BinaryOperator<A> combiner()
```

If it returns anything but the default value, this Gatherer can be parallelized. When parallelized this is used to merge partial results into one.

```
BiConsumer<A, Downstream<R>> finisher()
```

When there are no more input elements, this function is invoked with the state, and a Downstream handle to perform a final action.

```

List<List<String>> result = Stream.of( ...values: "a", "b", "c", "d", "e") Stream<String>
    .gather(Gatherers.windowFixed( windowSize: 2)) Stream<List<...>>
    .toList();
System.out.println(result); // [[a, b], [c, d], [e]]
List<List<String>> result2 = Stream.of( ...values: "a", "b", "c", "d", "e") Stream<String>
    .gather(Gatherers.windowSliding( windowSize: 2)) Stream<List<...>>
    .toList();
System.out.println(result2); // [[a, b], [b, c], [c, d], [d, e]]

List<String> result3 = Stream.of( ...values: "a", "b", "c", "d", "e")
    .gather(Gatherers.scan(() -> "", ( String a, String b) -> a + b))
    .toList();
System.out.println(result3); // [a, ab, abc, abcd, abcde]

List<String> result4 = Stream.of( ...values: "a", "b", "c", "d", "e")
    .gather(Gatherers.fold(() -> "", ( String a, String b) -> a + b))
    .toList();
System.out.println(result4); // [abcde]

List<String> result5 = Stream.of( ...values: "a", "b", "c", "d", "e")
    .gather(Gatherers.mapConcurrent( maxConcurrency: 2, String::toUpperCase))
    .toList();
// maxConcurrency 값만큼 virtual Thread가 생성되어 병렬로 실행됨
System.out.println(result5); // [A, B, C, D, E]

List<IndexedValue<String>> result6 = Stream.of( ...values: "a", "b", "c") Stream<String>
    .gather(new IndexedGatherer<>()) Stream<IndexedValue<...>>
    .toList();
result6.forEach( IndexedValue<String> iv -> System.out.println(iv.index() + " => " + iv.value()));
/*
 * 0 => a
 * 1 => b
 * 2 => c
 */

```

```
import java.util.concurrent.atomic.AtomicInteger;
import java.util.function.Supplier;
import java.util.stream.Gatherer;

public class IndexedGatherer<V> implements Gatherer<V, AtomicInteger, IndexedValue<V>> { 1 usage

    @Override
    public Supplier<AtomicInteger> initializer() { return () -> new AtomicInteger( initialValue: 0); // 초기 상태: 인덱스 0부터 시작 }

    @Override 9 usages
    public Integrator<AtomicInteger, V, IndexedValue<V>> integrator() {
        return ( AtomicInteger state, V element, Downstream<super IndexedValue<...>> downstream) -> {
            int index = state.getAndIncrement(); // 현재 index 가져오고 +1
            downstream.push(new IndexedValue<>(index, element)); // 값 전달
            return true; // 계속 진행
        };
    }
}
```



Java의 미래, Virtual Thread

Virtual Thread 핵심 정리

- Virtual Thread란?

- JDK 21 정식 도입 (Project Loom)
- 가볍고 빠른 경량 스레드
- JVM이 직접 스케줄링 → OS 스레드 생성 X

- 특징

- `Thread.startVirtualThread()`로 간단 생성
- Thread Pool 없이 수십만 개 생성 가능
- 기존 코드와 100% 호환 (Runnable, Executor 등)



Java의 미래, Virtual Thread

적용 시 주의사항

반드시 체크해야 할 4가지

1. **ThreadLocal** 주의

→ 메모리 누수 가능성 (**Virtual Thread**는 재사용 X)

2. **synchronized** 지양

→ **Carrier Thread Pinning** 발생

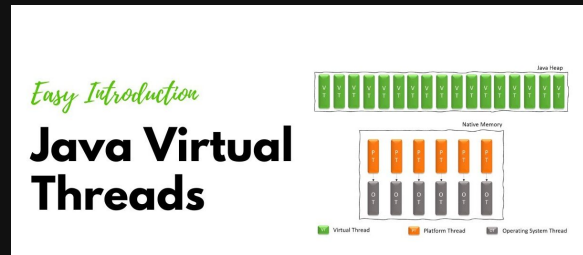
→ **ReentrantLock** 권장

3. 폴링 금지

→ **Virtual Thread**는 생성/삭제가 빠름, **Thread Pool** 쓰지 말 것

4. 리소스 제한 필요

→ **DB** 커넥션 등 유한 자원은 **Semaphore**로 제어



카카오페이 분산 락 구현

우아한 분산락 노하우

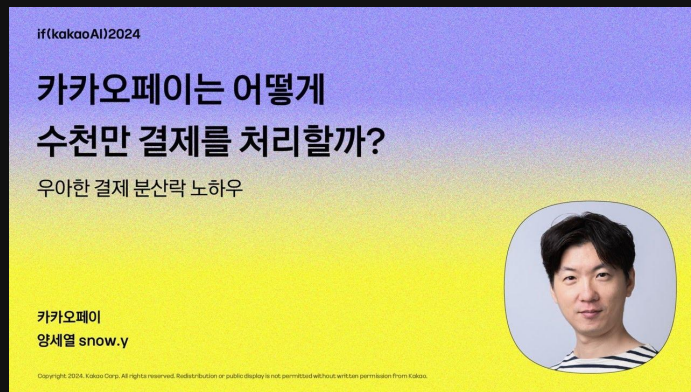
- AOP 기반 분산 락 구현 시 비효율성 발생

→ 임계영역 분리에 따른 성능 저하, 유지보수 어려움

- 이를 함수형(데코레이션)으로 구현하여 성능 향상

- 분산 락 활용 시 Redisson 활용

→ 락 관련 인터페이스, 락 취득 방식 이점



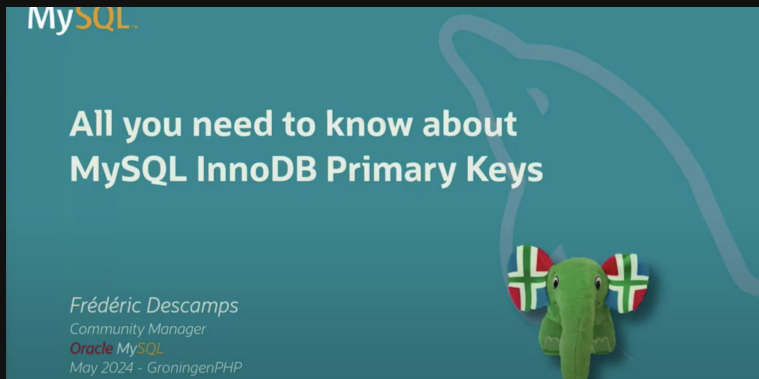
```
@Service ㄹ Heedong *
class MyService{
    private val lockManager: LockManager,
} {
    @DistributedLock new *
    fun doSomething1(lockKey: String) {
        ~~~~~
    }

    fun doSomething2(userId: Long) { ㄹ Heedong *
        lockManager.userLock(userId) {
            ~~~~~
        }
    }
}
```

MySQL InnoDB에서의 PK

MySQL Primary Key에 대한 이모저모

- MySQL InnoDB에서 가장 좋은 PK의 조건
 - 작은 사이즈 + 순차적 증가
 - 순차적으로 증가하지 않는 PK는 많은 **rebalancing**을 유발
 - **Secondary Index**의 **right most column**으로 PK가 쓰이기에 PK의 크기는 작을 수록 좋다.
- GIPK Mode
 - MySQL 8.0.30부터 지원되는 PK 없는 테이블에 PK를 자동 생성해주는 기능
- UUID를 PK로 사용할 때 주의할 점



마지막 한 마디

우리가 남긴 흔적

“좋은 발표를 듣는 것에서 멈추지 않고,
직접 써보고 실험해보며
우리만의 인사이트로 만들었습니다.”

“Virtual Thread를 따라가던 발걸음은
결국 내 코드와 내 시스템으로 이어졌습니다.”

Q & A

The background features abstract, swirling patterns in various shades of green and teal. On the left, there is a large, light teal ring-like shape. On the right, there are concentric, swirling patterns in a vibrant green color. The overall effect is a dynamic and organic visual texture.

Thank you
