InstagramID

1. 개요 (Overview)

이 문서는 InstagramId 클래스를 학습용 프로젝트로 구현하기 위한 요구사항을 정의합니다.

목표는 분산 시스템 환경에서 고유 ID를 생성하는 방법을 이해하고, 비트 연산(`bit manipulation`)을 활용한 효율적인 데이터 구조 설계를 실습하는 것입니다. 이 프로젝트를 통해 개발자는 다음과 같은 지식을 습득할 수 있습니다.

- 분산 ID 생성 전략: Instagram, Twitter(Snowflake) 등에서 사용하는 ID 생성 방식의 원리를 이해합니다.
- 비트 연산: Long 타입의 64비트를 활용하여 [시간, 샤드, 시퀀스] 값을 효율적으로 저장하고 추출합니다.
- 동시성 제어: 여러 스레드가 동시에 ID를 요청할 때 시퀀스 값이 겹치지 않도록 동기화 메커니즘을 사용하는 방법을 사용합니다.
- 클래스 설계 원칙: `Comparable`, `Serializable` 인터페이스 구현, 불변 객체(`Immutable Object`) 설계, 정적 팩토리 메서드(`static factory method`) 패턴을 적용합니다.

2. 기능 요구사항 (Functional Requirements)

2.1. ID 구조

생성되는 고유 ID는 64비트 정수(long) 형태여야 하며, 다음 세 가지 컴포넌트로 구성됩니다.

- 타임스탬프 (Timestamp): 41비트
  - 밀리초(milliseconds) 단위의 시간을 저장합니다.
  - 특정 에포크(epoch) 시점(`igOurEpoch`)을 기준으로 현재 시간을 저장하여, ID의 유효 기간을 늘립니다.
  - 이를 통해 약 41년 동안 고유 ID 생성이 가능해야 합니다.
- 샤드 ID (Shard ID): 13비트
  - 분산 시스템에서 서버 또는 데이터베이스 인스턴스를 식별하는 데 사용되는 논리적 샤드(logical shard) 값을 저장합니다.

    - 이를 통해 최대 2^{13}개 (8,192개)의 샤드를 식별할 수 있어야 합니다.
- 시퀀스 (Sequence): 10비트
    - 동일한 밀리초(millisecond) 및 샤드 내에서 고유성을 보장하기 위한 자동 증가 값 (auto-incrementing value)입니다.
    - 이를 통해 밀리초당 샤드당 최대 2^{10}개 (1,024개)의 ID를 생성할 수 있어야 합니다.

2.2. 핵심 메서드

- `generateId()`: 타임스탬프, 현재 샤드 ID, 다음 시퀀스 값을 조합하여 새로운 `InstagramId` 객체를 생성하고 반환합니다. 이 메서드는 thread-safe 해야 합니다.
- `makeRawId(rawTimestamp, shard, sequence)`: 주어진 3개의 인자를 비트 연산하여 64비트의 `long` 타입 ID를 생성합니다.
- `getTimestamp()`: ID 값에서 타임스탬프를 추출하여 반환합니다.
- `getShardId()`: ID 값에서 샤드 ID를 추출하여 반환합니다.
- `getSequence()`: ID 값에서 시퀀스 값을 추출하여 반환합니다.

2.3. 유틸리티 및 기타 기능

- 바운드(Bound) 생성: 특정 타임스탬프에 해당하는 가장 작은 ID (`lowerBound`)와 가장 큰 ID (`upperBound`)를 생성하는 정적 메서드를 제공해야 합니다. 이는 데이터베이스 쿼리에서 시간 범위를 이용한 검색 성능을 최적화하는 데 유용합니다.
- 직렬화(Serialization): `Serializable` 인터페이스를 구현하여 객체를 직렬화/역직렬화할 수 있어야 합니다.
- 비교(Comparison): `Comparable` 인터페이스를 구현하여 ID 값에 따라 객체를 비교할 수 있어야 합니다.
- 로깅: `Slf4j`를 사용하여 ID 생성 및 디버깅 정보를 로깅해야 합니다.

3. 비기능 요구사항 (Non-Functional Requirements)

- 성능: ID 생성은 매우 낮은 지연 시간(low latency)으로 이루어져야 하며, 초당 수십만 건 이상의 ID를 생성할 수 있어야 합니다.

- 가용성: 시스템의 일부 샤드가 실패하더라도 ID 생성 기능은 중단되지 않아야 합니다. (단, 이 프로젝트에서는 논리적 샤드 개념만 구현)
- 확장성: 시스템에 새로운 샤드를 추가하더라도 ID 생성 방식에 큰 변경이 없어야 합니다.
- 유지보수성: 코드가 명확하고 재사용 가능한 형태로 설계되어야 합니다. 특히 `enum`을 활용한 비트 연산 추상화는 코드의 가독성을 높여야 합니다.

4. 구현 가이드라인 (Implementation Guidelines)

4.1. 클래스 및 인터페이스 설계

- `InstagramId` 클래스는 불변(immutable)으로 설계해야 합니다. 모든 필드는 `final`로 선언하고, `new InstagramId()` 생성자는 `private`으로 선언하여 외부에서 직접 객체를 생성할 수 없게 합니다.
- `Component` `enum`을 사용하여 비트 연산에 필요한 `bits`, `mask`, `shiftBits` 정보를 캡슐화해야 합니다.

4.2. 주요 로직 상세

- 에포크 (Epoch):
  - `igOurEpoch = 1314220021721L` (2011년 8월 24일) 값을 상수로 사용합니다. 이 값은 Instagram이 ID 생성 시스템을 시작한 시점으로 알려져 있습니다.
- 비트 연산:
  - `makeRawId` 메서드에서 각 컴포넌트(`timestamp`, `shard`, `sequence`)를 왼쪽으로 쉬프트(`<<`)하고 비트 OR(`|`) 연산을 사용하여 64비트 ID를 구성합니다.
  - 값 추출 시에는 오른쪽으로 쉬프트(`>>`)하고, 해당 컴포넌트의 마스크(`&`)를 적용하여 불필요한 비트를 제거합니다.
  - `Component` `enum`의 `getShiftBits()` 메서드는 각 컴포넌트가 쉬프트되어야 할 비트 수를 계산하여 반환해야 합니다. `TIMESTAMP(41)`은 샤드(13) + 시퀀스(10) = 23비트만큼 쉬프트 되어야 합니다.
- 동시성:

- 시퀀스 값 (`counter`)은 여러 스레드에서 안전하게 증가시킬 수 있도록 구현합니다. `counter.incrementAndGet()` 메서드는 원자적으로 시퀀스 값을 증가시킵니다.

5. 테스트 요구사항 (Test Requirements)

- 단위 테스트:
  - `generateId()`가 고유한 ID를 생성하는지 확인합니다.
  - `getTimestamp()`, `getShardId()`, `getSequence()` 메서드가 `makeRawId`로 생성된 ID에서 정확한 값을 추출하는지 검증합니다.
  - `lowerBound()`와 `upperBound()` 메서드가 의도한 대로 동작하는지 테스트합니다.
  - 다양한 인코딩(`Base32`, `Base62`, `Base64`, `Hex`)과 디코딩 메서드가 올바르게 상호 변환되는지 검증합니다.
- 성능 테스트:
  - 멀티 스레드 환경에서 `generateId()`를 동시에 호출하여 경쟁 조건(race condition)이 발생하지 않는지, 즉 ID가 중복되지 않는지 확인합니다.

6. 후에 추가될 내용

1. 다양한 인코딩/디코딩: 64비트 ID 값을 `Base32`, `Base62`, `Base64`, `Hex` 등 다양한 문자열 형식으로 변환하고, 반대로 문자열을 다시 `InstagramId` 객체로 변환하는 메서드를 제공해야 합니다.
2. `UniqueId` 추상 클래스를 상속받아 공통 메서드를 구현합니다.
   a. 새로운 아이디 체계 구현
3. `ReentrantLock`은 이 프로젝트의 요구사항에 명시적으로 필요하지 않지만, 더 복잡한 동기화 로직이 필요할 때 사용될 수 있음을 참고합니다.

7. Pull Request

한번에 너무 많은 범위를 PR로 보내면 리뷰하는 입장에서 범위를 특정하기 어렵습니다.

각각의 요구사항을 브랜치로 나누어 리뷰를 요청합니다.

예를 들면 2.1, 2.2, 2.3 을 각각 구현하고 각각의 브랜치로 나누어 PR을 요청 해주세요.

만약 특정 요구사항의 구현이 너무 방대하다면, 브랜치를 나누셔도 좋습니다.