Redis Week2

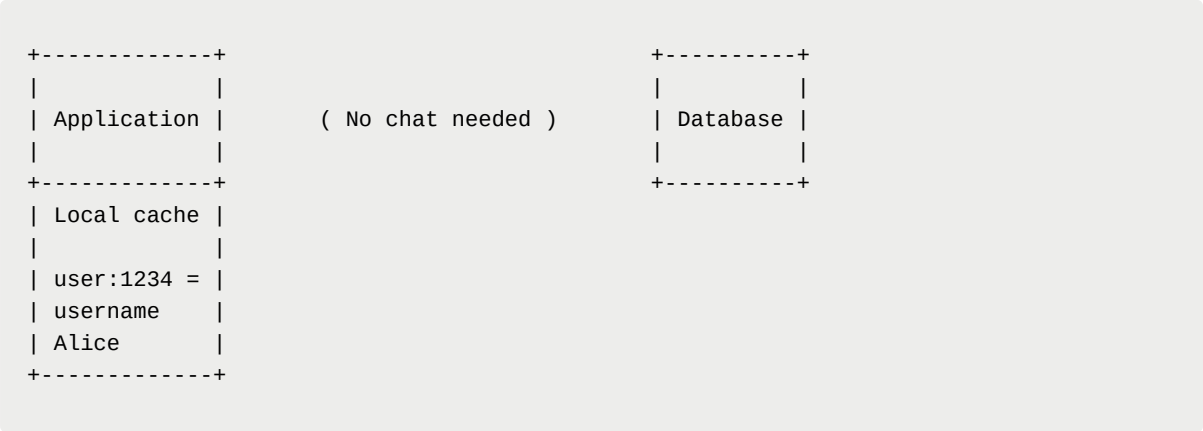
Client-side caching in Redis

Redis 6에서 부터 지원하는 기술로, 불필요한 네트워크 왕복을 줄여서 유저 입장에서 더 빠르게 볼 수 있다.

자주 호출하는 쿼리 응답을 어플리케이션 메모리 내부에 직접 저장을 해서 나중에 데이터베이스에 다시 접속하지 않더라도 응답을 재사용할 수 있다.

기술적으로는 클라이언트가 요청한 키를 추적하고 키 값이 변경되면 클라이언트에 알리는 것이다.

그러므로 클라이언트는 이제 해당 무효화 알람을 받을 때까지 네트워크를 통해 Redis에 쿼리 하지 않아도 응답을 재사용할 수 있다.



Redis Client-side-caching은 Tracking이라고 하며 두가지 모드가 있다.

1. 기본 모드에서 서버는 주어진 클라이언트가 액세스 한 키를 기억하고 동일한 키가 수정되면 무효화 메세지를 보낸다. 이는 서버 측에서 메모리 비용이 들지만 클라이언트가 메모리에 가질 수 있는 키 집합에 대해서만 무효화 메세지를 보낸다.
2. 브로드캐스팅 모드에서는 서버는 클라이언트가 액세스한 키를 기억하려고 시도하지 않으므로 서버측에 메모리 비용이 전혀 들지 않는다. 대신 클라이언트와 같은 키를 구독하고 구독한 prefix와 일치하는 키를 터치할 때마다 알림 메세지를 받는다.

Redis pipelining

개별 명령에 대한 응답을 기다리지 않고 한 번에 여러 명령을 내림으로써 성능을 향상시키는 기술이다. 파이프라이닝은 대부분 redis client에서 지원된다.

redis는 TCP 기반 클라이언트-서버 모델을 따르기 때문에 아래와 같이 동작하고 네트워크 IO에 대한 병목이 존재한다.

- 클라이언트가 서버에 전송한 쿼리는 소켓을 통해 읽혀지고, 보통 blocking 형태로 response 된다.
- 서버는 명령을 처리하고 응답을 다시 클라이언트로 보낸다.

아래의 예시에서는 수행하는 작업들은 각각 1회씩 request와 response를 거친다.

```
127.0.0.1:6379> set a 1
OK
127.0.0.1:6379> incr a
(integer) 2
127.0.0.1:6379> incr a
(integer) 3
127.0.0.1:6379> incr a
(integer) 4
127.0.0.1:6379> incr a
(integer) 5
```

Redis가 로컬에서 동작하고 있다면 클라이언트와 서버 간의 네트워킹 링크가 굉장히 빠르겠지만, 두 호스트 간에 물리적으로 많은 홉이 사용되었다면 지연이 발생될 가능성이 크다.

이 시간을 `RTT(Round Trip Time)`라고 한다. 만약 서버가 **초당 100k(100,000)개**의 요청을 처리할 수 있다고 하더라도, RTT가 250ms라면 **초당 최대 4개의 요청**만을 처리할 수 있게 된다. 로컬에서 동작하는 Redis의 경우엔 RTT가 훨씬 더 짧지만, 많은 read/write를 수행하기 위해서는 여전히 많은 양이다. 이 경우엔 RTT가 0.12ms였는데, 처리량이 초당 100k라면 이에 훨씬 못 미치는 속도이다.

이러한 병목을 해결하기 위해서 piplelining API를 사용한다. 이를 통해 클라이언트는 여려 개의 작업을 쌓아서 한 번에 전송 할 수 있게 된다. 따라서 응답을 전혀 기다리지 않고 여러 명령을 한번에 서버로 보낼 수 있고 응답을 한번에 읽을 수 있다.

또한 실제로 redis server에서 초당 수행 할 수 있는 작업의 수를 크게 향상 시킨다.

Redis keyspace notifications

redis에서 데이터 변경을 받을 수 있는 pub/sub 메커니즘, 이 기능을 사용하면 다음과 같은 사용사례를 구현할 수 있다.

- 좋아요 같은 일부 키가 만료될 때 알림을 받을 수 있음
- 특수한 키에 대한 변경사항을 모니터링 할때

참고할 점은 pub-sub 클라이언트의 연결이 끊어졌다가 다시 연결 되면 연결이 끊어진 시간 동안 전달된 모든 이벤트가 손실된다.

해당 기능을 사용하려뫈 다음과 같은 단계를 거쳐야 한다.

- redis.conf에서 `config set` 명령을 사용해서 활성화 한다.
- 키 패턴을 사용해서 sub 연결을 만든다.

해당 기능은 CPU 리소스를 사용하기 때문에 기본적으로는 비활성화 되어 있다,

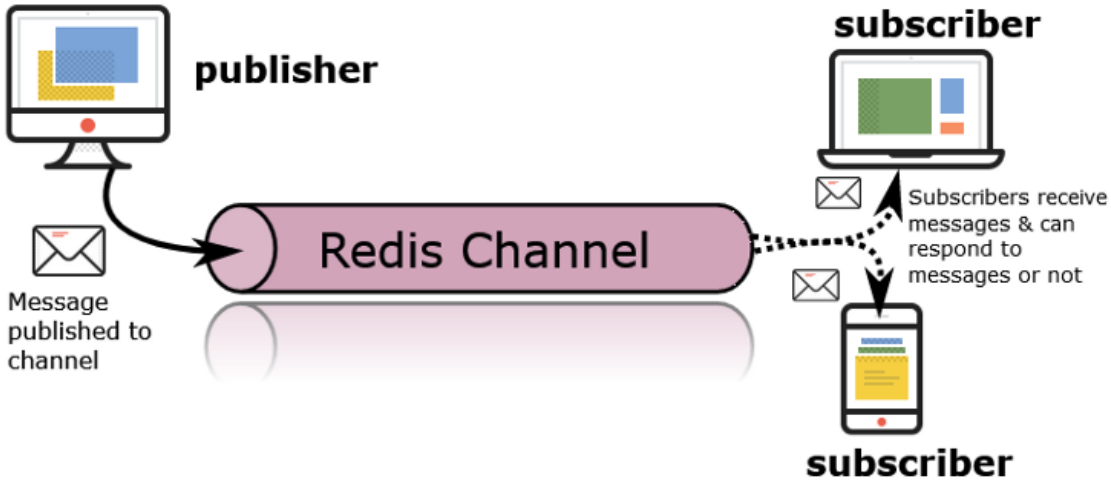
```
$ redis-cli config set notify-keyspace-events KEA
$ redis-cli --csv psubscribe '__key*__:*'
Reading messages... (press Ctrl-C to quit)
"psubscribe","__key*__:*",1
```

유의 사항은 다음과 같다.

- 기능이 켜져 있는 동안 redis에서 CPU 사용량이 많이 사용되기 때문에 기본적으로는 꺼져있다.
- 키 만료는 실시간이 아니기 때문에 키 만료 이벤트 시에 실시간으로 통보가 되지 않을 수 있다.
- 클러스터 모드로 사용할 때 알림 이벤트는 각 노드에 대해 독립적으로 발생한다. 노드 전체의 모든 키에 대한 알림을 받으려면 모든 노드를 수신해야한다.

Redis Pub/Sub

Pub/Sub은 게시자와 구독자를 정의하는 메시징 패러다임으로, redis에서도 지원을 한다. 다만 일반적인 rabbitMQ 같은 것과는 다르게 redis의 Pub/Sub 시스템은 메시지를 보관(queuing) 하지 않는다. Publish 하는 시점에 이미 실행한 subscribe 명령으로 대기하고 있는 클라이언트들에게만 전달된다.



Transactions

트랜잭션은 나누어지지 않는 최소한의 단위로 만들어 All or Nothing 전략을 취할 수 있도록 하는 단위이다. 즉, 트랜잭션으로 묶게 되면 트랜잭션 내부에서 하나의 로직이 실패하여 오류가 나게 되면 모두 취소 시키면 그렇지 않으면 모두 성공 시키는 것이다.

트랜잭션을 유지하기 위해선 순차성을 가져야 하고 도중에 명령어가 치고 들어오지 못하게 Lock이 필요하다. redis에서는 **MULTI, EXEC, DISCARD 그리고 WATCH 명령어**를 이용하면 된다.

- MULTI
    - Redis의 트랜잭션을 시작하는 커맨드. 트랜잭션을 시작하면 Redis는 이후 커맨드는 바로 실행되지 않고 queue에 쌓인다.
- EXEC

    - 정상적으로 처리되어 queue에 쌓여있는 명령어를 일괄적으로 실행합니다. RDBMS의 Commit과 동일하다.
- DISCARD
    - queue에 쌓여있는 명령어를 실괄적으로 폐기합니다. RDMS의 Rollback과 동일하다.
- WATCH
    - Redis에서 Lock을 담당하는 명령어입니다. 이 명령어는 낙관적 락(Optimistic Lock) 기반이다.
    - **Watch 명령어를 사용하면 이 후 UNWATCH 되기전에는 1번의 EXEC 또는 Transaction 아닌 다른 커맨드만 허용한다**.

먼저 CLI로 MULTI 커맨드를 입력하면 트랜잭션을 사용할 수 있다. 이훙 들어오는 명령어는 바로 실행되는 것이 아니라 큐에 쌓이게 된다. 그리고 마지막에 EXEC 커맨드를 통해 일괄적으로 실행되게 되는 구조이다. 참고로 GET 커맨드 또한 큐로 쌓이게 된다.

```
>> MULTI
"OK"
>> SET SABARADA BLOG
"QUEUED"
>> SET KAROL BLOG
"QUEUED"
>> GET SABARADA # GET은 어떻게 처리되는지 확인
"QUEUED"
>> EXEC   # QUEUED 된 명령어를 실행한 후 결과를 전부 출력
1) "OK"
2) "OK"
3) "BLOG" # GET 또한 바로 처리되지 않고 QUEUED 된 후 EXEC 했을 때 처리
```

트랜잭션 내부의 커맨드가 정상적으로 실행 되었기 때문에 아래와 같이 GET 했을 때 정상 출력 되는 것을 확인할 수 있다.

```
>> GET SABARADA
"BLOG"
>> GET KAROL
"BLOG"
```

다음으로 Rollback이 어떻게 이루어지는지 보도록 하자. redis의 rollback은 RDBMS와 조금 방식이 다르다.

MULTI 커맨드를 사용 후 DISCARD 명령어를 명시적으로 실행한다 이렇게 하면 큐에 쌓여 있던 명령어가 일괄적으로 없어지게 된다.

```
>> MULTI
"OK"
>> SET SABARADA BLOG
"QUEUED"
>> SET KAROL BLOG
"QUEUED"
>> GET SABARADA
"QUEUED"
>> DISCARD
"OK"
```

만약 도중에 잘못된 명령어를 사용했을 경우에는 큐에 쌓여 있돈 모든 명령어가 DISCARD 되게 된다.

```
>> MULTI
"OK"
>> SET SABARADA 4
"QUEUED"
>> HSET SABARADA 2 3
"QUEUED"
>> DD HKD
(error) unknown command `DD`, with args beginning with: `HKD`,
>> EXEC
(error) Transaction discarded because of previous errors.
```

잘못된 자료구조의 명령어를 사용하는 경우에는 잘못된 명령어가 하나 있더라도 해도 정상적으로 사용한 명령어에 대해서는 잘 적용이 된다.

```
>> MULTI
"OK"
>> HSET SABARADA 2 3
"QUEUED"
>> SET SABARADA 4
"QUEUED"
>> EXEC
ERROR: Something went wrong.
>> GET SABARADA
"4"
```

이런 트랜잭션 방식을 채택한 이유는 대부분 개발 과정에서 일어날 수 있는 에러이고 prod에서는 거의 발생하지 않는 에러이기 때문이다. 또한 rollback을 채택하지 않음으로 빠른 성능을 유지할 수 있다.

Lock

Lock을 구현하는데 사용하는 명령어는 WATCH이다. 내가 해당 값을 변경하고 있는데 다른 사람이 동일하게 key를 건드린다면 잘못된 값이 입력될 수 있기 때문이다. WATCH 명령어를 이용하면 해당 Key는 트랜잭션에서 값 변경을 1번으로 제한할 수 있다. 아래는 SABARADA 라는 Key에 WATCH 명령어를 이용하여 Lock을 걸어둔 상태에서 성공하는 커맨드 Set이다.

```
>> WATCH SABARADA
"OK"
>> MULTI
"OK"
>> SET SABARADA 3
"QUEUED"
>> EXEC
1) "OK"
```

아래는 실패하는 커맨드 셋이다. `WATCH` 를 이용해 KAROL이라는 Key를 Lock을 걸었다. 하지만 2번째에서 해당 Key에 대한 값을 바꿔버렸다. 이후에 트랜잭션을 이용하여 Karol의 값을 변경한다. EXEC를 실행할때 `WATCH` 명령어를 사용한 Key는 CAS(Check And Set)을 내부적으로 돌린다. 하지만 이 경우에는 이미 `KAROL` 이라는 Key의 값이 변경이 되었기 때문에 Check에서 실패가 되며 값 변경은 실패로 돌아간다.

```
>> WATCH KAROL
"OK"
>> SET KAROL 7
"OK"
>> MULTI
"OK"
>> SET KAROL 5
"QUEUED"
>> EXEC
(nil)
```

때문에 아래처럼 값을 변경 후에 명시적으로 WATCH를 풀어주게 되면 정상적으로 값이 변경 될 수 있다.

```
>> WATCH KAROL
"OK"
>> SET KAROL 8
"OK"
>> UNWATCH
"OK"
>> MULTI
"OK"
>> SET KAROL 4
"QUEUED"
>> EXEC
1) "OK"
```

추가적인 예제로 아래처럼 Set으로 값을 바꿔주고 트랜잭션을 2번 하게 되면 2번째에서는 성공하게 된다. 왜냐하면 EXEC가 호출 될 때 UNWATCH가 묵시적으로 호출되기 때문에 2번째 트랜잭션에 대해서는 해당 키에 대해서는 WATCH가 걸려있지 않기 때문이다.

```
>> WATCH KAROL
"OK"
>> SET KAROL 22
"OK"
>> MULTI
"OK"
>> SET KAROL 11
"QUEUED"
>> EXEC
(nil)
>> MULTI
"OK"
>> SET KAORL 11
"QUEUED"
>> EXEC
1) "OK"
```