Redis

CM1-2팀 박주원

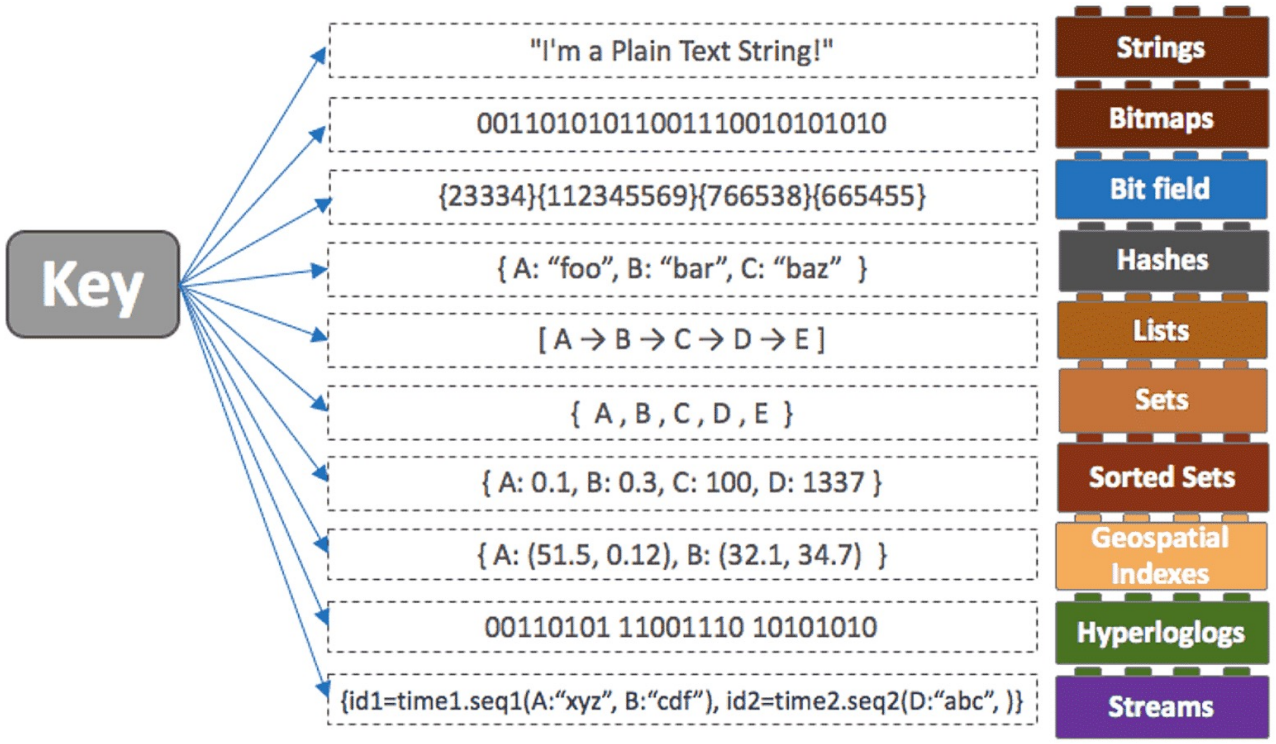
Redis

- **Re**mote **Di**ctionary **S**erver
- Key, Value 구조의 비정형 데이터를 저장하고 관리하기 위한 오픈 소스 기반의 비관계형 데이터베이스 관리 시스템
- 캐시, Persistence Storage, 메시지 브로커로 사용되며 인메모리 데이터 구조를 가진 저장소
- 데이터를 디스크에 쓰는 구조가 아니라 메모리에서 데이터를 처리하기 때문에 속도가 빠름

Redis 특징

- Collection 지원 (Memcached와의 차별점)
  - String – 가장 일반적인 key-value 구조의 형태
  - Sets – String의 집합. 여러개의 값을 하나의 value에 넣을 수 있음. 포스트의 태깅 같은 곳에 사용될 수 있음
  - Sorted Sets – 중복된 데이터를 담지 않는 Set 구조에 정렬 Sort 를 적용한 구조. 랭킹 보드 구현에 사용 가능
  - Lists – Array 형식의 데이터 구조.

- 하나의 Collection에 너무 많은 아이템 담지 않기
- Expire는 아이탬 개별이 아닌 Collection 전체에 걸림



Redis 특징

- 메인쓰레드는 싱글 쓰레드로 운용됨
  - Atomic 보장, race condition 회피
  - 오래 걸리는 명령을 실행하면 다른 명령에 영향을 줌
  - O(N) 명령어 사용 자제 (ex, KEYS, FLUSHALL, DEL 등)
- 버전 6.x부터는 일부 멀티 쓰레드 도입
  - 클라이언트에서 전송한 명령을 읽고 파싱하는 부분
  - 명령어 처리 결과를 클라이언트에게 전송하는 부분
  - 명령어 실행 자체는 메인쓰레드에서 수행하므로 여전히 싱글 쓰레드
- KEYS 대신 scan 명령 사용 권장 → 하나의 긴 명령을 짧은 여러 명령으로 대체
- Collection의 모든 아이템 가져와야할 때는 일부만 가져오거나 (Sorted set), 큰 컬렉션을 다른 여러개의 컬렉션으로 나누어 저장 (1개당 몇 천개 수준이 적당)

Redis 메모리 관리

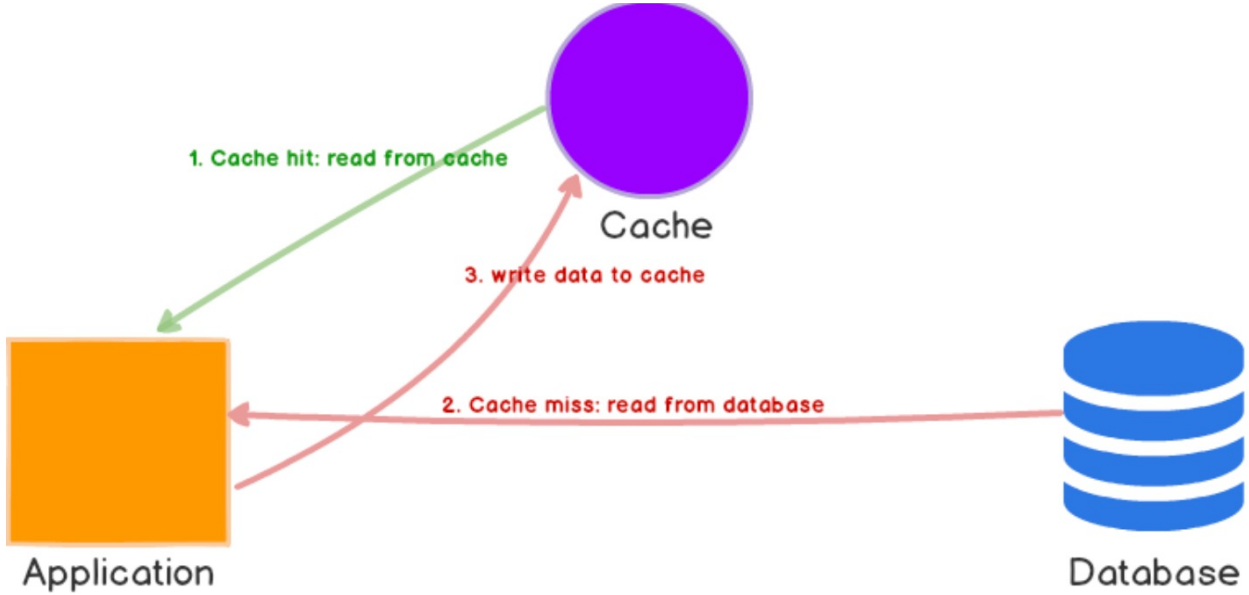
- 다양한 사이즈를 가지는 데이터보다 유사한 크기의 데이터를 가지는 경우가 유리
  - 메모리 단편화 완화
  - 페이지로 관리했을 시 내부 단편화 최소화
- 큰 메모리를 사용하는 instance 보다 적은 메모리를 사용하는 여러 개 instance
  - Redis의 쓰기 방식은 copy on wirte → 필연적으로 fork() 수행
  - (24GB instance) vs (8GB * 3 instance) → 각각 8GB에 해당하는 Instance를 썼을때 → 48GB, 32GB
- Collection 자료구조 최적화
  - Hash -> HashTable 하나 더 사용
  - Sorted Set -> Skiplist, HashTable 둘 다 사용
  - Set -> HashTable 사용
  - 만약 한 Collection에 데이터가 많다면 Hash, Sorted Set, Set에서 ziplist 사용이 메모리 관점에서 유리
  - in-memory 특성 상, 적당한 사이즈의 데이터까지는 특정 알고리즘을 안쓰고 그냥 풀서치(선형 탐색)를 해도 빠르다.
  - 실제로 zip list 를 쓴 것과 안쓴것의 메모리 사용량은 2-30% 정도 차이가 난다고 한다.

캐시 서버

- 일반적으로 서비스 운영 초반이거나 규모가 작은 서비스의 경우 WEB-WAS-DB 구조로도 데이터베이스에 무리가 가지 않음
- 하지만 사용자가 늘어나면 데이터베이스가 과부화될 수 있고, 이때 캐시 서버를 도입하는 경우가 많음
- 이 캐시 서버로 사용할 수 있는 것이 바로 Redis
- 같은 요청이 여러 번 들어오는 경우 매번 데이터베이스를 거치는 것이 아니라 캐시 서버에서 첫번째 요청 이후 저장된 결괏값을 바로 내려주기 때문에 DB의 부하를 줄이고 서비스의 속도도 느려지지 않는 장점이 있음

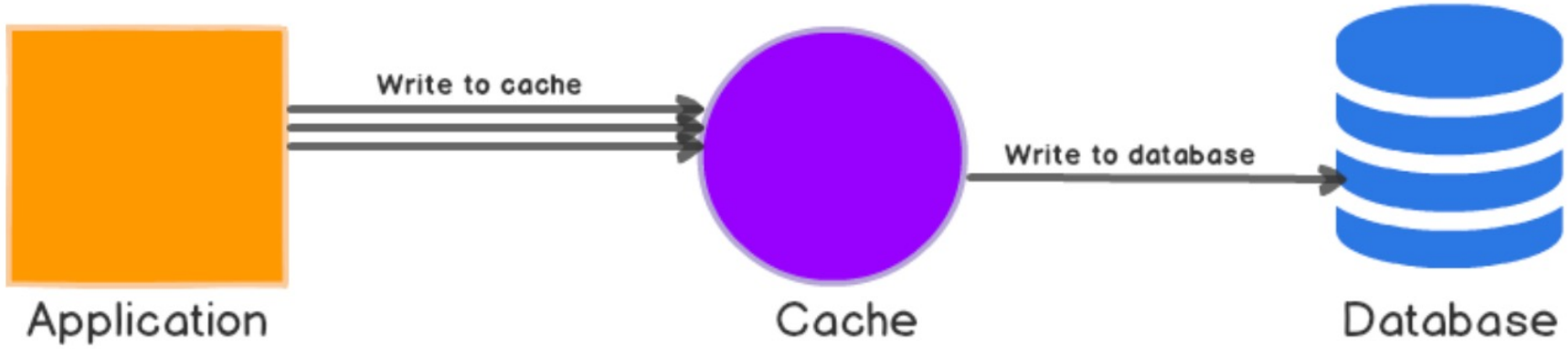
캐시 서버

- Look aside cache (Cache-aside)
  - 클라이언트가 데이터를 요청
  - 웹서버는 데이터가 존재하는지 캐시 서버에 먼저 확인
  - 캐시 서버에 데이터가 있으면 DB에 데이터를 조회하지 않고 캐시 서버에 있는 결과값을 클라이언트에게 바로 반환 (cache hit)
  - 캐시 서버에 데이터가 없으면 DB에 데이터를 조회하여 캐시 서버에 저장하고 결과값을 클라이언트에게 반환 (cache miss)



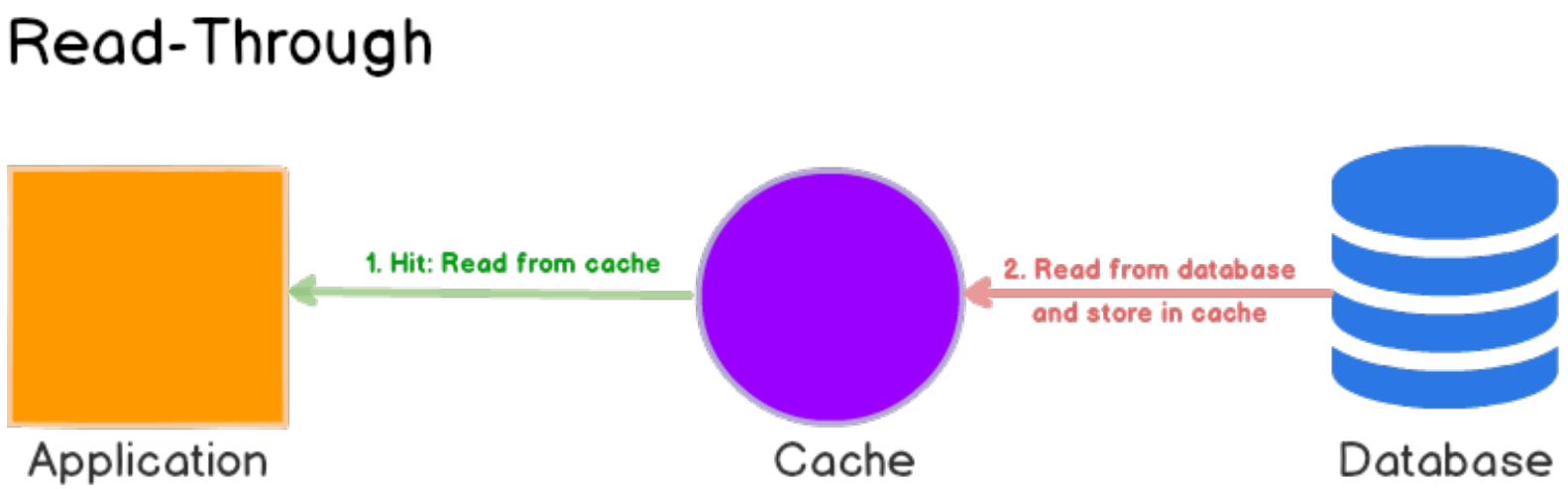
캐시 서버

- Write back
  - 웹 서버는 모든 데이터를 캐시 서버에 저장
  - 캐시 서버에 특정 시간 동안 데이터가 저장됨
  - 캐시 서버에 있는 데이터를 DB에 저장
  - DB에 저장된 캐시 서버의 데이터를 삭제
- Insert query를 한번씩 500번 날리는 것 보다 붙여서 한번에 날리는 것이 더 효율적이라는 원리 (Batching)
- 서버에 장애가 발생하면 데이터 손실 위험 --> 고가용성 설계 필요



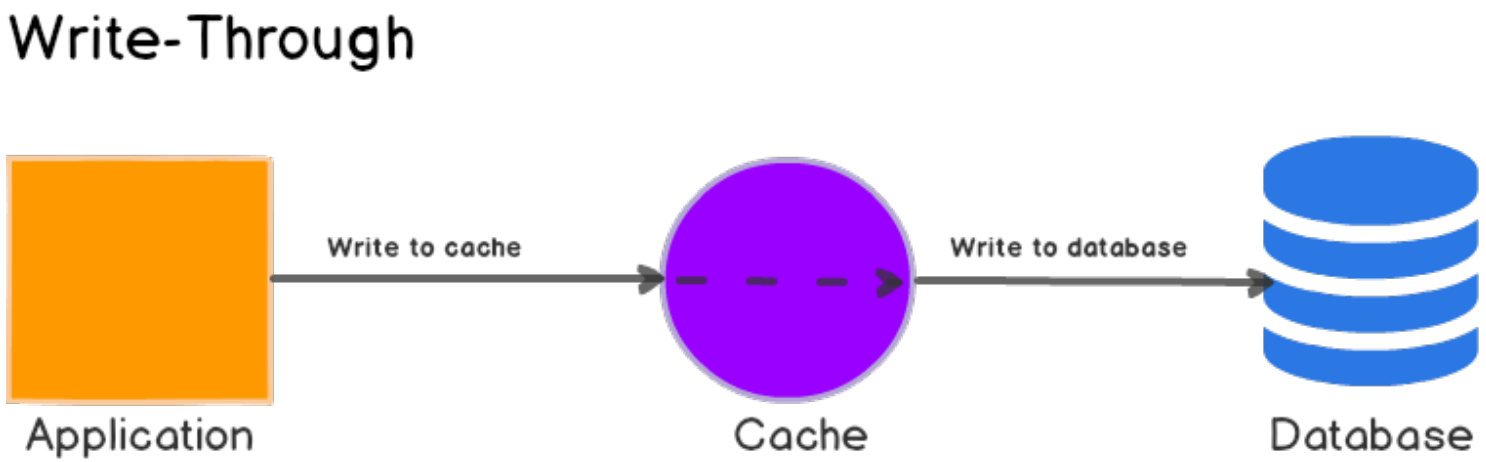
캐시 서버

- Read Through
  - 데이터가 처음 요청될 때 항상 캐시 미스가 발생하고 데이터를 캐시에 로드하는 추가 패널티 발생
  - 개발자는 수동으로 쿼리를 실행하여 캐시를 ‘warming'하거나 ‘pre-heating'하여 이 문제를 처리



캐시 서버

- Write Through
  - Read-Through와 함께 쓰면 데이터 일관성 보장



Redis Persistence

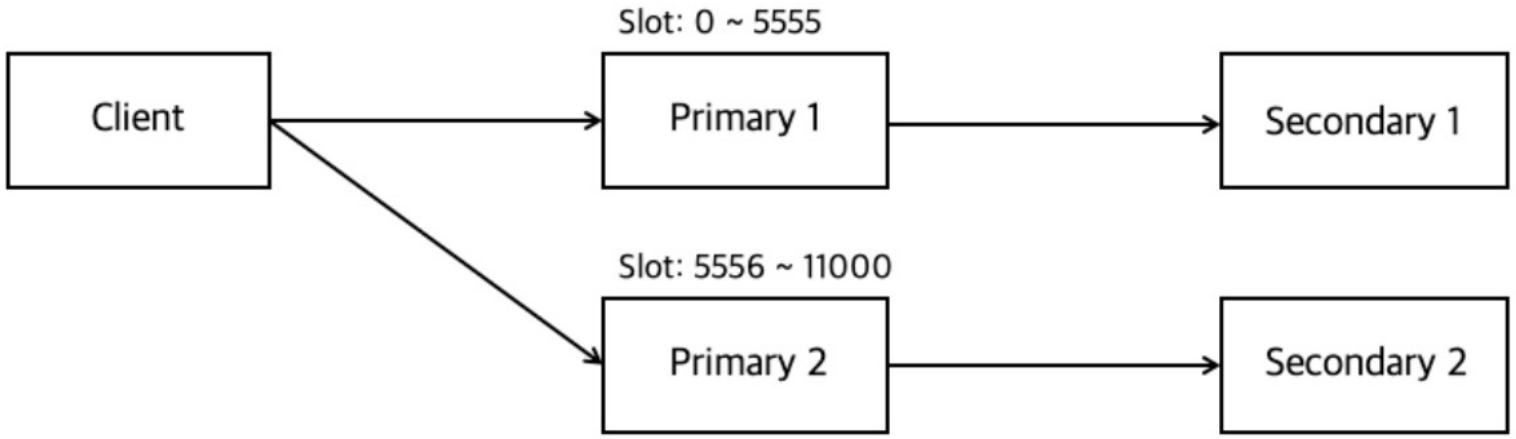
- Redis는 데이터를 메모리에 저장 → Shutdown 시 데이터 휘발
- AOF (Append Only File) = 보관
  - 명령이 실행될 때 마다 파일에 기록 (입력/수정/삭제 시, 조회 x)
  - Aof 파일을 통해 영속성 보장
- RDB (Snapshot) = 백업
  - 특정한 간격으로 메모리에 있는 Redis 데이터 전체를 disk에 바이너리 형태로 기록
  - AOF 보다 size가 작으며, 로딩속도도 빠름

Redis Failover

- Redis Cluster Failover
- Coordinator Failover
- VIP / DNS 기반 Failover

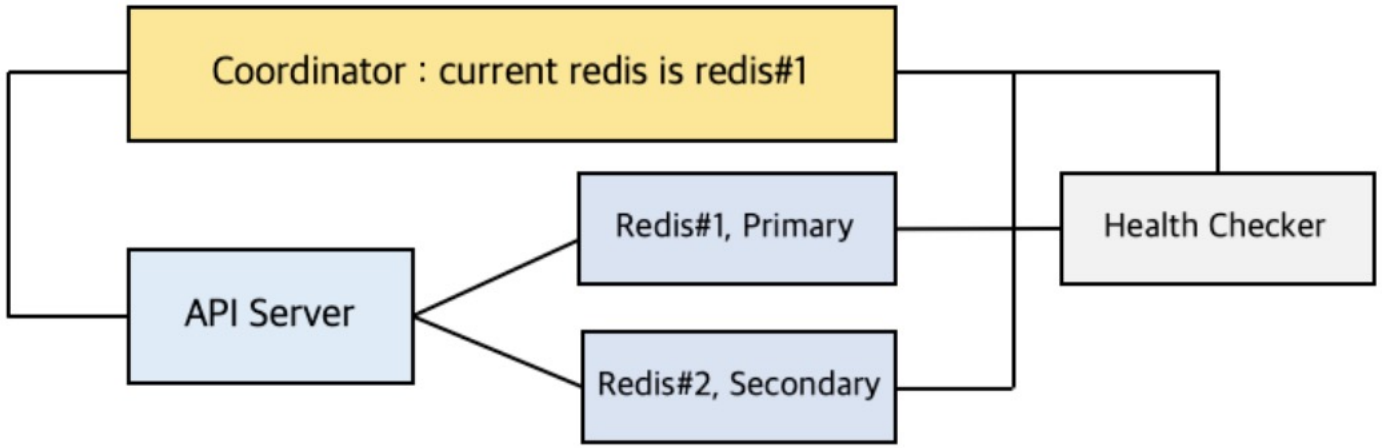
Redis Cluster

- Primary가 죽으면 Secondary가 primary로 승격되는 구조
- Primary 1 데이터 변경 시 secondary 1 만 같이 변경
- 장점
  - 자체적인 primary / secondary Failover
  - Slot 단위의 데이터 관리: 특정 키는 특정 노드에 저장과 같은 명시적 관리 가능
- 단점
  - 메모리 사용량이 더 많다.
  - Migration 정책 결정 필요
  - 라이브러리 구현이 필요



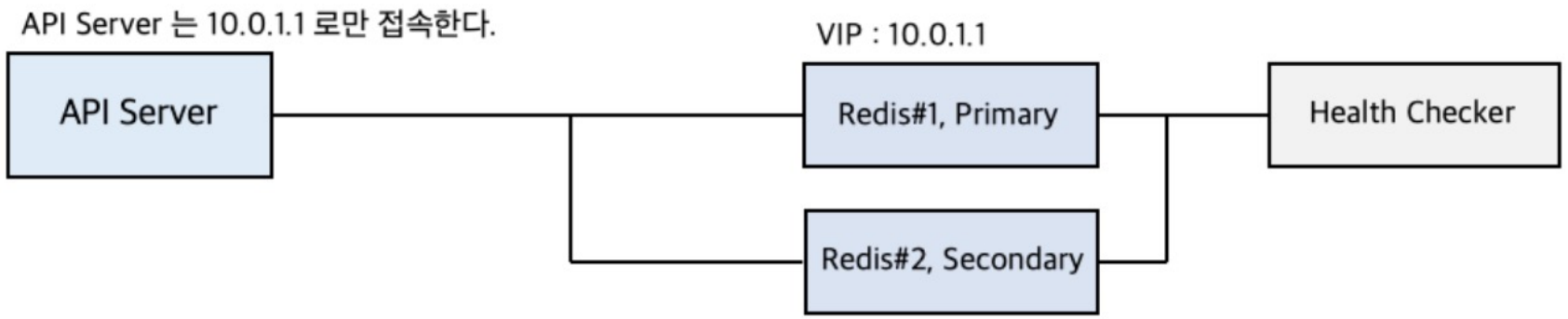
Coordinator 기반 Failover

- zookeeper , etcd, consul 등의 코디네이터를 사용해서 정보를 저장하고 관리



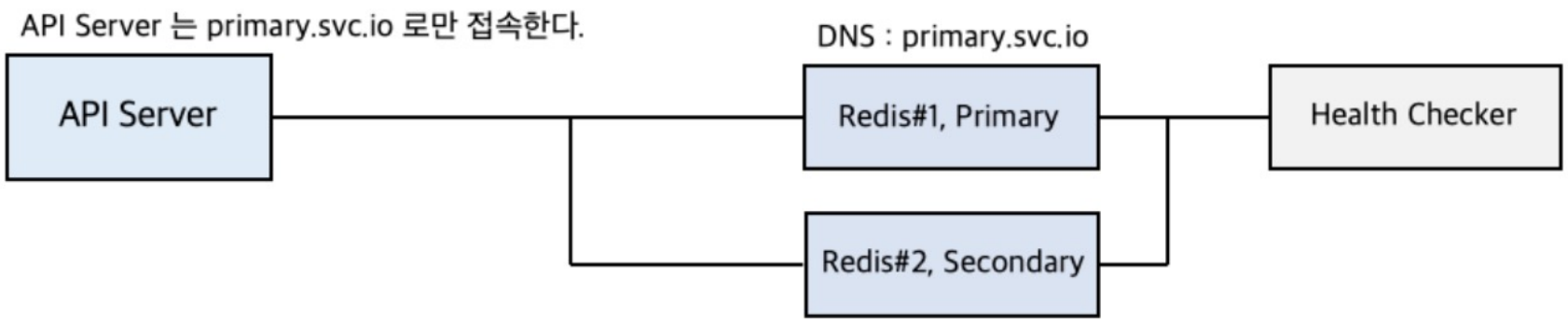
VIP / DNS 기반 Failover

- VIP 기반 Failover 로직은 Redis 서버마다 Vertual IP 할당
- API Server 는 특정 VIP 를 가지고 있는 Redis에만 접속하는 방법



VIP / DNS 기반 Failover

- DNS 기반 Failover 전략도 VIP 방식과 같은 로직을 따르지만 **도메인을 할당한다는 차이**만 존재 → 아마존은 이 전략 사용
- VIP / DNS Failover 방식은 결국 코드를 바꾸는게 아니기 때문에 다른 서비스에서도 사용 가능



Reference

- https://codeahoy.com/2017/08/11/caching-strategies-and-how-to-choose-the-right-one/
- https://www.youtube.com/watch?v=mPB2CZiAkKM