**Redis**

1 Redis 란

2 Redis 설치

 1) AWS 설치

 2) 윈도우10 docker 설치

 3) 윈도우에 설치하기

  - 이건 보류

3 Redis 데이터 타입

4 Spring Data Redis

5 Spring Session Redis

6 Pub/Sub

7 트랜잭션

8 Redis 확장

9 기타

CLI

|  |  |
| --- | --- |
| KEYS \* | 키값 찾기 |
| INCR key | 그키값을 찾아 숫자를 하나 증가시킨다 |
| DEL key | 키값지우기 키까지 |
| ./redis-cli  KEYS "\*"  | xargs ./redis-cli DEL | 키값 지우기 키까지 패턴으로 다지움 |
| 127.0.0.1:6379> INCRBYFLOAT gg 1  "14"  127.0.0.1:6379> INCRBYFLOAT gg 1  "15"  127.0.0.1:6379> INCRBYFLOAT gg 2  "17"  127.0.0.1:6379> INCRBYFLOAT gg -1 | 값 증가 증감 등등. |
| PUBSUB CHANNELS \* | 채널 열어져있는거 확인 |

**1. Redis 란?**

REmote DIctionary System의 약자로 메모리 기반의 Key/Value Store.

**1) 홈페이지﻿**

 - https://redis.io(해석해 보면)

   Redis는 데이터베이스, 캐시 및 메시지 브로커로 사용되는 오픈 소스 (BSD 라이센스) 인 메모리 데이터 구조 저장소입니다.

   string, hashes, lists, sets, sorted sets과 같은 range queries(값 범위에 따른 쿼리),

   bitmaps, hyperloglogs, geospatial indexes와 같은 radius queries(반경, 범위)를 이용하는 데이터 구조들을 지원 합니다.

   Redis는 내장 된 Replication, Lua scripting(가벼운 명령형/절차식 언어로, 확장 언어로 쓰일 수 있는 스크립팅 언어),

   LRU eviction(Least Recently Used. 캐시 항목 추가를 위한 공간을 확보하기 위해 캐시에서 필요 없는 항목을 결정 eviction 알고리즘. 가장 오래된 항목들을 삭제하는 알고리즘),

   트랜잭션 및 다른 수준의 on-disk peristence(지속성),

   Redis Sentinel을 통한 고 가용성 및 Redis Cluster를 통한 자동 파티셔닝을 제공합니다.

   당신은 문자열에 추가하는 것, 해쉬의 값을 증가, 요소를 목록으로 푸시하기, 집합의 교차,합 및 차이 계산,

   또는 정렬 된 집합에서 가장 높은 순위의 구성원을 얻는 것과 같은 아주 작은(원자성)의 기능들을 실행할 수 있습니다.

   탁월한 성능을 달성하기 위해 Redis는 메모리 내 데이터 세트를 사용합니다.

   유스 케이스에 따라 데이터 세트를 매번 디스크에 덤프하거나 각 명령을 로그에 추가하여 유지할 수 있습니다.

   기능이 풍부한 네트워크화 된 메모리 내 캐시 만 있으면 영속성을(Persistence) 선택적으로 비활성화 할 수 있습니다.

   Redis는 또한 master-slave 비동기 복제, 매우 빠른 논 블로킹 첫 번째 동기화, net split에서 부분 재 동기화와의 자동 재 연결을

   위한 간단한(사소한) 설치를 제공 합니다.

* Transactions
* Pub/Sub(메시지 발송, 구독)
* Lua scripting
* Keys with a limited time-to-live
* LRU eviction of keys
* Automatic failover

 - 대부분의 프로그래밍 언어에서 Redis를 사용할 수 있습니다.

   Redis는 ANSI C로 작성되었으며 Linux, \* BSD, OS X와 같은 대부분의 POSIX 시스템에서 외부 종속성없이 작동합니다.

   Linux와 OS X는 Redis가 개발되고 테스트 된 두 가지 운영 체제이며 배포를 위해 Linux를 사용할 것을 권해 드립니다.

   Redis는 SmartOS와 같은 Solaris 기반 시스템에서 작동하지만 지원은 최선의 노력입니다.

   Windows 빌드에 대한 공식적인 지원은 없지만 Microsoft는 Redis의 Win-64 포트를 개발하고 유지 관리합니다.

 - 2006년 살바토르 산필리포가 C로 개발

**2) 다운로드**

 - 안정적인 버전 3.2.9

 - 윈도우 버전은 공식적으로 지원 안하지만 Microsoft Open Tech group이 개발 관리

   https://github.com/MSOpenTech/redis

**2. Redis 설치**

1) AWS 에 설치해 보기

 - aws EC2 인스턴스 생성, Centos 7.2 설치

  >> wget http://download.redis.io/releases/redis-3.2.6.tar.gz

>> tar xzvf redis-3.2.6.tar.gz

>> sudo make install

를 실행하면 실패한다.

yum -y install gcc-c++ 실행 후 make clean 을 하면

jemalloc/jemalloc.h: No such file or directory 실패 오류가 남

>> sudo make distclean

>> sudo make install

2) 윈도우 10 에 docker를 이용해 설치해 보기

 - <https://store.docker.com/editions/community/docker-ce-desktop-windows> 에서 Docker CE for Windows 버전을 다운로드

  docker run -d -p 6379:6379 --name "my-redis" redis:latest

를 하면 redis 가 설치 됨

redis 의 경우 서비스이기 때문에 접속을 위해서는

docker exec -it my-redis /bin/bash

를 실행한다.

3) 윈도우 10 에 centos 설치 후 docker 설치해 보기

docker search centos

여러 이미지 중에 yum update가 되어 있는 아래 이미지를 다운로드

docker pull januswel/centos

docker run -i -t januswel/centos

redis 디렉토리를 만들고 이동

[root@4fba1960774a redis]# pwd

/redis

[root@4fba1960774a redis]# yum install wget

[root@4fba1960774a redis]# yum install epel-release

[root@4fba1960774a redis]# yum -y install gcc-c++

[root@4fba1960774a redis]# wget http://download.redis.io/releases/redis-3.2.6.tar.gz

[root@4fba1960774a redis]# tar xzvf redis-3.2.6.tar.gz

폴더 이동후

[root@4fba1960774a redis-3.2.6]# make install

설치 끝. redis 폴더로 돌아와 node 설치

[root@4fba1960774a redis]# wget https://nodejs.org/dist/v6.10.3/node-v6.10.3.tar.gz

[root@4fba1960774a redis]# tar xvfz ./node-v6.10.3.tar.gz

[root@4fba1960774a node-v6.10.3]#./configure

[root@4fba1960774a node-v6.10.3]#make

[root@4fba1960774a node-v6.10.3]#make install

[root@4fba1960774a node-v6.10.3]# node --version

v6.10.3

[root@4fba1960774a node-v6.10.3]# npm install redis

[root@4fba1960774a study]# pwd

/redis/node-v6.10.3/study

docker attach 로 띄우면 창을 두개 띄워도 하나로 동기화 되어 버린다.

하나는 docker attach 로 하나는 docker exec로 띄운다.

docker pull gt1000/my-redis:1.0

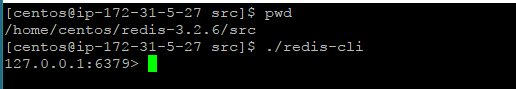
4) 윈도우용 Redis 설치해 보기

 - 생략. 나중에 시간나면...... ㅋ.ㅋ

5) 실행 파일(여러 실행 파일을 제공하지만 아래 두가지에 초점을 맞춤)

 - redis-server : 레디스의 실제 저장소. 클러스터 모드 또는 독립 실행형 모드로 실행  
    

 - redis-cli : 레디스 커맨드를 실행할 수 있는 커맨드라인 인터페이스. 레디스 서버에 접속하는 방법을 보여줌



6) GUI Tool

<https://redisdesktop.com/>

<http://fastoredis.com/>

<https://github.com/luin/medis>

**3. Redis 데이터 타입**

1) 데이터 타입

 가. 문자열

  - 레디스에서 가장 다양한 데이터 타입, 512MB까지

    정수, 부동소수점, 바이너리 데이터(비디오, 이미지등)와 같은 어떠한 종류의 데이터도 저장가능

  - 캐시 메카니즘, 자동 만료되는 캐시, 계수 계산(페이지뷰, 비디오 뷰 등)

예제)

127.0.0.1:6379> SET current\_chapter "Chapter 1"  
 OK  
 127.0.0.1:6379> EXPIRE current\_chapter 60           // 만료 시간을 60초  
 (integer) 1  
 127.0.0.1:6379> GET current\_chapter  
 "Chapter 1"  
 127.0.0.1:6379> TTL current\_chapter                 // 생존 시간을 리턴, -2 만료되거나 존재하지 않으면, -1키가 존재하지만 만료 시간을 저장하지 않음  
 (integer) 50

  - 레디스는 싱글 스레드 기반으로 동작한다.

    counter 키 값이 1이고 A와 B가 동시에 증가를 시키면 A는 2, B는 3을 얻음

  - string 이라고 해서 문자열만 저장하는 것이 아니고 바이너리(binary) 데이터도 저장 할 수 있음.

예제)

127.0.0.1:6379> MSET key1 "test1" key2 "test2"

OK

127.0.0.1:6379> get key1

"test1"

127.0.0.1:6379> get key2

"test2"

127.0.0.1:6379> mget key1 key2

1) "test1"

2) "test2"

  - 노드를 이용해 문자열로 투표 시스템 개발하기

var redis = require("redis");

var client = redis.createClient();

function setKey() {

client.set("article:12345:headline", "Google wants to Turn Your Clothes Into a Computer");

client.set("article:10001:headline", "For Millennials, the End of the TV Viewing Party");

client.set("article:60056:headline", "Alicia Vikander, Who Portrayed,xxxxx");

}

function upVote(id) {

var key = "article:" + id + ":votes";

client.incr(key);

}

function downVote(id) {

var key = "article:" + id + ":votes";

client.decr(key);

}

function showResults(id) {

var headlineKey = "article:" + id + ":headline";

var voteKey = "article:" + id + ":votes";

client.mget([headlineKey, voteKey], function(err, replies) {

console.log('The article "' + replies[0] + '" has', replies[1], 'votes');

});

}

setKey();

upVote(12345);

upVote(12345);

upVote(12345);

upVote(10001);

upVote(10001);

downVote(10001);

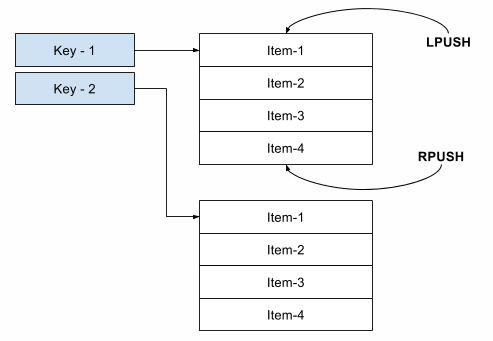
upVote(60056);

showResults(12345);

showResults(10001);

showResults(60056);

client.quit();

 나. 리스트  
   

  - collection, stack, queue와 같이 동작, 많은 이벤트 시스템은 리스트 커멘트가 원자적인 특성을 갖고 있어

    병열 시스템이 큐에서 엘리먼트를 얻어낼때 중복으로 얻지 않도록 보장해 주기때문에 레디스 리스트를 queue로 사용

    이벤트 큐 : Resque, Celery, Logstash를 포함한 많은 툴

    최근 사용자 글 저장하기 : 트위트는 사용자의 최근 트윗을 저장할 때 리스트를 사용

  - 리스트에 블로킹(blocking) 커멘드가 존재하면 linked list라서 처음, 끝에서의 추가, 삭제시 항상 O(1), 일정시간을 가짐

  - 최대 개수는 2(32승) - 1개, 40억개 이상의 엘리먼트를 가질 수 있음

 예제)  
 127.0.0.1:6379> LPUSH books "Clean Code"                  // 리스트의 맨 처음에 입력  
 (integer) 1  
 127.0.0.1:6379> RPUSH books "JavaScript Pattern & Test"   // 리스트의 맨 마지막에 입력  
 (integer) 2  
 127.0.0.1:6379> LPUSH books "한글테스트"  
 (integer) 3  
 127.0.0.1:6379> LRANGE books 0 -1                         // index는 0, 1로 증가... -1은 맨 마지막  
 1) "\xed\x95\x9c\xea\xb8\x80\xed\x85\x8c\xec\x8a\xa4\xed\x8a\xb8"       // 한글이 깨짐... 찾아봐야 함  
 2) "Clean Code"  
 3) "JavaScript Pattern & Test"  
 127.0.0.1:6379> rpush mylist a b

(integer) 2

127.0.0.1:6379> rpop mylist

"c"

127.0.0.1:6379> rpop mylist

"b"

127.0.0.1:6379> rpop mylist

(nil) // 비어있는 리스트에 대해서 pop을 하면 nil을 반환

  - Capped Lists                  ....................................................................... 시간 되면 작성  
  - Blocking operation on lists....................................................................... 시간 되면 작성    
  - 일반적인 큐 시스템 구현

// queue.js

function Queue(queueName, redisClient) {

this.queueName = queueName;

this.redisClient = redisClient;

this.queueKey = 'queues:' + queueName;

// zero means no timeout

this.timeout = 0;

}

Queue.prototype.size = function(callback) {

this.redisClient.llen(this.queueKey, callback);

};

Queue.prototype.push = function(data) {

this.redisClient.lpush(this.queueKey, data);

};

Queue.prototype.pop = function(callback) {

this.redisClient.brpop(this.queueKey, this.timeout, callback);

};

exports.Queue = Queue;

//consumer-worker.js

var redis = require("redis");

var client = redis.createClient();

var queue = require("./queue");

var logsQueue = new queue.Queue("logs", client);

function logMessages() {

logsQueue.pop(function(err, replies) {

var queueName = replies[0];

var message = replies[1];

console.log("[consumer] Got log: " + message);

logsQueue.size(function(err, size) {

console.log(size + " logs left");

});

logMessages();

});

}

logMessages();

// product-worker.js

var redis = require("redis");

var client = redis.createClient();

var queue = require("./queue");

var logsQueue = new queue.Queue("logs", client);

var MAX = 5;

for (var i = 0; i < MAX; i++) {

logsQueue.push("Hello world #" + i);

}

console.log("Created " + MAX + " logs");

client.quit();

 다. 해시

  - 필드를 값으로 매핑, 필드 이름과 값이 문자열. 메모리 최적화(장점)

  - hash-max-ziplist-entries 와 hash-max-ziplist-value 설정을 기반

 예제)

127.0.0.1:6379> HSET movie "title" "test1"

(integer) 1

127.0.0.1:6379> HMSET movie "year" 1927 "watchers" 1000000

OK

127.0.0.1:6379> HGET movie "title"

"test1"

127.0.0.1:6379> HINCRBY movie "watchers"

(error) ERR wrong number of arguments for 'hincrby' command

127.0.0.1:6379> HINCRBY movie "watchers" 3

(integer) 1000003

  - 해시와 노드를 이용한 투표 시스템

var redis = require("redis");

var client = redis.createClient();

function saveLink(id, author, title, link) {

client.hmset("link:" + id, "author", author, "title", title, "link", link,

"score", 0);

}

function upVote(id) {

client.hincrby("link:" + id, "score", 1);

}

function downVote(id) {

client.hincrby("link:" + id, "score", -1);

}

function showDetails(id) {

client.hgetall("link:" + id, function(err, replies) {

console.log("Title:", replies['title']);

console.log("Author:", replies['author']);

console.log("Link:", replies['link']);

console.log("Score:", replies['score']);

console.log("--------------------------");

});

}

saveLink(123, "dayvson", "Maxwell Dayvson's Github page", "https://github.com/dayvson");

upVote(123);

upVote(123);

saveLink(456, "hltbra", "Hugo Tavares's Github page", "https://github.com/hltbra");

upVote(456);

upVote(456);

downVote(456);

showDetails(123);

showDetails(456);

client.quit();

2) 고급 데이터 타입

 가. 셋

  - 순서가 없고 동일한 문자열이 없는 콜렉션

  - 데이터 필터링, 데이터 그룹핑, 엘리먼트십 확인 등에 사용

    redis-cli를 이용한 셋 예제

   SADD, SINTER, SDIFF, SUNION,SRANDMEMBER, SISMEMBER, SREM, SCARD 등

  예제)

127.0.0.1:6379> SADD user:max:favorite\_artists "Arcade File" "Arctic Monkeys" "Bell & Sebastian" "Lenine"

(integer) 4

127.0.0.1:6379> SADD user:hugo:favorite\_artists "Daft Punk" "The Kooks" "Arctic Monkeys"

(integer) 3

   - 딜 추적 시스템 개발 : 이미 등록된 회원이면... 안 보내고, 미등록 회원이면 메일 보내고 등록

var redis = require("redis");

var client = redis.createClient();

function markDealAsSent(dealId, userId) {

client.sadd(dealId, userId);

}

function sendDealIfNotSent(dealId, userId) {

client.sismember(dealId, userId, function(err, reply) {

if (reply) {

// user exist

console.log("Deal", dealId, "was already sent to user", userId);

} else {

// not exist

console.log("Sending", dealId, "to user", userId);

// code to send the deal to the user would go here...

markDealAsSent(dealId, userId);

}

});

}

function showUsersThatReceivedAllDeals(dealIds) {

client.sinter(dealIds, function(err, reply) {

console.log(reply + " received all of the deals: " + dealIds);

});

}

function showUsersThatReceivedAtLeastOneOfTheDeals(dealIds) {

client.sunion(dealIds, function(err, reply) {

console.log(reply + " received at least one of the deals: " + dealIds);

});

}

markDealAsSent('deal:1', 'user:1');

markDealAsSent('deal:1', 'user:2');

markDealAsSent('deal:2', 'user:1');

markDealAsSent('deal:2', 'user:3');

sendDealIfNotSent('deal:1', 'user:1');

sendDealIfNotSent('deal:1', 'user:2');

sendDealIfNotSent('deal:1', 'user:3');

showUsersThatReceivedAllDeals([ "deal:1", "deal:2" ]);

showUsersThatReceivedAtLeastOneOfTheDeals([ "deal:1", "deal:2" ]);

client.quit();

 나. 정렬된 셋

  - 셋과 비슷하지만, 모든 엘리먼트는 연관 점수를 가지는 점수로 정렬된, 중복 문자열이 없는 콜렉션

    점수가 동일할 경우 사전 편집 순으로 정렬

  redis-cli를 이용한 정렬된 셋 예제)

ZADD 는 정렬된 셋에 하나 이상의 엘리먼트를 추가

ZADD leaders 100 "Alice"

ZADD leaders 100 "Zero"

ZADD leaders 102 "Hugo"

ZADD leaders 101 "Max"

ZREVRANGE : 높은 점수에서 낮은 점수로 Hugo > Max > Zero > Alice

ZREM : 엘리먼트 삭제, ZEM leaders "Hugo"

ZSCORE : 점수를 리턴, ZSCORE leaders "Max" -----> 101

ZRANK : 낮은 순에서 높은순의 등수를 리턴, ZRANK leaders "Max" ------>

 다.비트맵

  - 실제 데이터 타입이 아니며, 내부적으로 문자열이다.

    문자열의 비트 연산자 집합이라고 말할 수 있다.

    setbit로 bit를 설정할 수 있고, getbit로 bit 설정여부를 확인할 수 있다.

    가장 큰 장점은 0과 1의 상태를 가지는 아이템들을 대단히 효율적으로 저장하고 읽을 수 있다는데 있다.

   a. redis-cli를 이용한 비트맵 예제

    - 하루 방문자를 계산해 보자. 사용자가 방문을 하면 해당 사용자 아이디에 매핑하는 bit값을 1로 세팅

      이때 필요한 메모리는 단지 1bit이기 때문에 512M로 40억(512 \* 1024 \* 1024 \* 8) 유저의 방문정보를 기록 가능

      setbit visitor 10 1    // 10번 사용자

      setbit visitor 17 1    // 17번 사용자

      setbit visitor 19 1    // 19번 사용자

      setbit visitor 102 1    // 102번 사용자

      setbit visitor 107 1    // 107번 사용자

      getbit visitor 17       // 17번 고객 방문여부, 1 리턴

      getbit visitor 20       // 20번 고객 방문여부, 0 리턴

      bitcounter visitor      // 방문 유저수 5

 라. 하이퍼로그로그

  - 실제 데이터 타입이 아니며 개념적으로는 알고리즘이다.

    어떤 데이터셋 집합에서 유일한 원소의 개수를 검사하기 우해서 사용하는 알고리즘이다. 약간의 오차를(0.81퍼센트의 표준오차, 100개중에 99.19개 정확) 허용하는 대신 매우 작은

    메모리로 유니크한 원소의 개수를 검사할 수 있다.

    웹사이트 고유 방문자수, 특정 날짜 시간의 웹사이트 검색 고유 키워드, 책의 고유단어 개수 등

    PFADD는 하나 이상의 문자열을 하이퍼 로그로그에 추가함

 127.0.0.1:6379> PFADD nickname "gt1000" "test" "gt1003" "gt1005"        // nickname에 4개의 키워드가 등록

(integer) 1

127.0.0.1:6379> PFADD nickname "gt1000" "gt1003"                        // 이미 존재 0

(integer) 0

127.0.0.1:6379> PFCOUNT nickname

(integer) 4

**4. Spring Data Redis**

<http://projects.spring.io/spring-data/>

1) 왜 스프링 데이터 Redis?

* Spring Framework는 최고의 풀 스택 Java / JEE 애플리케이션 프레임 워크.   
  가벼운 컨테이너와 의존성 삽입, AOP 및 편리한 서비스 추상화 제공.
* Spring Data Redis는 Spring 애플리케이션의 Redis에 대한 쉬운 구성과 액세스를 제공.  
  저장소와의 상호 작용을 위한 하위 수준 및 높은 수준의 추상화를 모두 제공하여 사용자를 인프라 관련 문제로부터 자유롭게 함.

2) 요구사항

* JDK 레벨 6.0 이상
* Spring Framework 5.0.0.M5 이상
* 키 값 저장소의 경우 Redis 2.6.x 이상
* 스프링 데이터 Redis는 현재 최신 3.2 릴리스에 대해 테스트.
* Spring 알기(이해하기)
* NoSQL 및 Key Value 저장소

3) Redis 연결

dependencies {

compile('org.springframework.boot:spring-boot-starter-data-redis')

}

 -  RedisTemplate, RedisConnectionFactory 두개만 설정하면 됨.

 가. RedisConnectionFactory

  - Redis 서버와의 통신을 위한 low-level 추상화를 제공

  - 설정에 따라서 새로운 RedisConnection 또는 이미 존재하는 RedisConnection을 리턴.  
    RedisConnection 은 Redis 서버와의 통신 추상화를 제공하며, exception 발생시 Spring DataAccessException으로 전환.

  - RedisConnection 은 binary value를 인자로 받고 결과를 리턴하는 low-level method를 리턴.

  - Jedis, Jredis(1.7 Deprecated), Lettuce, SRP, RJC 등의 클라이언트 라이브러리가 있음.

    Jedis 를 보편적으로 많이 사용하는거 같음

 나. RedisTemplate

  - Redis 서버에 Redis Command를 수행하기 위한 high-level 추상화를 제공

  - 오브젝트 serialization 과 connection management를 수행

  - Redis 서버에 데이터 CRUD를 위한 Key Type Operations 와 *Key Bound Operations 인터페이스를 제공*

*Key Type Operations*

* *ValueOperations -*

*Redis string (or value) operations*

* *ListOperations - Redis list operations*
* *SetOperations - Redis set operations*
* *ZSetOperations - Redis zset (or sorted set) operations*
* *HashOperations - Redis hash operations*
* *HyperLogLogOperations - Redis HyperLogLog operations like (pfadd, pfcount,…?)*
* *GeoOperations - Redis geospatial operations like GEOADD, GEORADIUS,…?)*

*Key Bound Operations*

* *BoundValueOperations - Redis string (or value) key bound operations*
* *BoundListOperations - Redis list key bound operations*
* *BoundSetOperations - Redis set key bound operations*
* *BoundZSetOperations - Redis zset (or sorted set) key bound operations*
* *BoundHashOperations - Redis hash key bound operations*
* *BoundGeoOperations - Redis key bound geospatial operations*

  - thread-safe 하며, 재 사용이 가능

  - 대부분의 기능에 RedisSerializer 인터페이스를 사용.

    StringRedisSerializer, JdkSerializationRedisSerializer, JacksonJsonRedisSerializer, Jackson2JsonRedisSerializer, GenericJackson2JsonRedisSerializer, OxmSerializer를 사용할 수 있음.

  - Redis에 저장된 키와 값이 java.lang.String이 되도록하는 것이 일반적이므로 StringRedisTemplate 확장 기능을 제공.

    StringRedisSerializer를 사용하며, 저장된 키와 값은 사람이 읽을 수 있음.

 @Configuration

public class RedisConfig {

@Bean

public JedisConnectionFactory jedisConnectionFactory() {

JedisConnectionFactory jedisConnectionFactory = new JedisConnectionFactory();

jedisConnectionFactory.setHostName("localhost");

jedisConnectionFactory.setPort(6379);

jedisConnectionFactory.setTimeout(0);

jedisConnectionFactory.setUsePool(true);

return jedisConnectionFactory;

}

@Bean

public RedisTemplate<String, Object> redisTemplate() {

RedisTemplate<String, Object> redisTemplate = new RedisTemplate<String, Object>();

redisTemplate.setKeySerializer(new StringRedisSerializer());

redisTemplate.setValueSerializer(new StringRedisSerializer());

redisTemplate.setConnectionFactory(jedisConnectionFactory());

return redisTemplate;

}

@Bean

public StringRedisTemplate stringRedisTemplate() {

StringRedisTemplate stringRedisTemplate = new StringRedisTemplate();

stringRedisTemplate.setKeySerializer(new StringRedisSerializer());

stringRedisTemplate.setValueSerializer(new StringRedisSerializer());

stringRedisTemplate.setConnectionFactory(jedisConnectionFactory());

return stringRedisTemplate;

}

}

4) 예제

@Slf4j

@RunWith(SpringRunner.class)

@SpringBootTest

public class SpringbootRedisApplicationTests {

--- 전체 소스는 github ---

@Resource(name="redisTemplate")

private ValueOperations<String, String> valueOperations;

@Resource(name="redisTemplate")

private ValueOperations<String, User> userOperations;

/\*\*

\* Redis string (or value) operations

\*/

@Resource(name="redisTemplate")

private ListOperations<String, String> listOperations;

/\*\*

\* Redis string (or value) operations

\*/

@Resource(name="redisTemplate")

private SetOperations<String, String> setOperations;

/\*\*

\* Redis string (or value) operations

\*/

@Resource(name="redisTemplate")

private ZSetOperations zSetOperations;

/\*\*

\* Redis string (or value) operations

\*/

@Resource(name="redisTemplate")

private HashOperations hashOperations;

@Before

public void 깨끗한\_상태로() {

// Set<String> keys = redisTemplate.keys("\*");

// log.info("@@@@@@@@@ 현재 키가 {} 개 들어 있습니다.", keys.size());

// for (String key : keys) {

// redisTemplate.delete(key);

// }

}

@After

public void 원래대로\_되돌림() {

System.out.println();

// try {

// redisTemplate.execute(new RedisCallback() {

// @Override

// public Object doInRedis(RedisConnection connection) throws DataAccessException {

// connection.flushAll();

// return null;

// }

// });

// } catch (Exception e) {

// log.warn("모든 캐시를 삭제하는데 실패했습니다.", e);

// }

}

@Test

public void valueOperations() {

log.info("----------- valueOperations start --------");

String key1 = "key1";

String key2 = "허균";

valueOperations.set(key1, "value1");

valueOperations.set(key2, "홍길동, 허생전");

log.info("{}의 값은 = {}, 사이즈는 = {} 입니다.", key1, valueOperations.size(key1));

log.info("{}의 값은 = {}, 사이즈는 = {} 입니다.", key2, valueOperations.size(key2));

log.info("----------- valueOperations end --------");

}

@Test

public void listOperations() {

log.info("----------- listOperations start --------");

String key1 = "여자랭킹";

listOperations.rightPush(key1, "이나영");

listOperations.rightPush(key1, "소녀시대 윤아");

String key2 = "남자랭킹";

listOperations.leftPush(key2, "이승환");

listOperations.leftPush(key2, "송광호");

log.info(" {} 의 왼쪽 값은 {} ", key1, listOperations.leftPop(key1));

log.info(" {} 의 오른쪽 값은 {} ", key2, listOperations.rightPop(key2));

log.info("----------- listOperations end --------");

}

@Test

public void setOperations() {

log.info("----------- setOperations start --------");

String key1 = "user:gt1000:intro";

setOperations.add(key1, "잘 생긴 훈남");

setOperations.add(key1, "영화감삼");

setOperations.add(key1, "감성적");

setOperations.add(key1, "제일 중요한건.... 소고기 함 쏴 알려 주께...");

Set<String> intro = setOperations.members(key1);

intro.forEach((value) -> log.info(" 나는 = {} ", value));

// for(String value : intro) {

// log.info(" 나는 = {} ", value);

// }

log.info("----------- setOperations end --------");

}

@Test

public void hashOperations() {

log.info("----------- hashOperations start --------");

String key1 = "user:gt1000:detail";

hashOperations.put(key1, "age", 43);

hashOperations.put(key1, "재산", "개털이다");

hashOperations.put(key1, "job", "언제 짤릴지 모르는 프로그래머");

hashOperations.put(key1, "술은 쇠주", 1);

Map<String, Object> entry = hashOperations.entries(key1);

log.info(" 그이 나이 {} 세", entry.get("age"));

log.info(" 모아둔 재산은 {}", entry.get("재산"));

log.info(" 직업은 {}", entry.get("job"));

log.info(" 주량은 소주 {} 병이다.", entry.get("술은 쇠주"));

log.info("----------- hashOperations end --------");

}

@Test

public void userTest() {

String userId1 = "gt1000";

String userId2 = "gt1003";

String userId3 = "gt1005";

User user1 = new User(userId1, "엄청 멋진 놈");

user1.setAge(30);

User user2 = new User(userId2, "당신의 천사가 되고 싶은 사람 == gt1000");

user2.setAge(40);

User user3 = new User(userId3, "천사 보다 더 좋은 사람 == gt1000");

user3.setAge(43);

userOperations.set(userId1, user1);

userOperations.set(userId2, user2);

userOperations.set(userId3, user3);

log.info("key = {}, value = {}", userId1, userOperations.get(userId1));

log.info("key = {}, value = {}", userId2, userOperations.get(userId2));

log.info("key = {}, value = {}", userId3, userOperations.get(userId3));

}

}

# 5. Spring Session Redis

<http://docs.spring.io/spring-session/docs/current/reference/html5/guides/httpsession.html>

단일 서버 환경에서 세션 기능을 사용하기 위해서는 HttpSession 객체를 활용하고,

멀티 서버 환경에서 세션 기능을 사용하기 위해서는 세션 정보를 Redis에 저장할 수 있습니다.

이 경우 세션 정보는 Redis 서버가 재시작 하기 전까지 유지 됩니다.

1) gradle

compile 'org.springframework.session:spring-session:1.3.1.RELEASE'

2) spring 설정

@EnableRedisHttpSession ======================================> 이 애노테이션을 추가

@SpringBootApplication

public class SpringbootRedisApplication extends SpringBootServletInitializer {

3) 소스

@Controller

public class LoginController {

@GetMapping(value = "/session")

@ResponseBody

String loginProcess(HttpSession session, Model model, User user) {

String sessionKey = "session-gt1000";

User sessionUser = (User) session.getAttribute(sessionKey);

if(sessionUser == null) {

user.setId(sessionKey);

user.setName("핸섬보이");

session.setAttribute(sessionKey, user);

log.info("신규 세션 생성 key = {}, value = {}", sessionKey, user);

} else {

log.info("이미 존재하는 세션 key = {}, value = {}", sessionKey, sessionUser);

}

return "success";

}

}

<http://localhost/session>

을 실행해 보면 실제 redis 서버에는 3개의 key 값이 추가 된다.

key = spring:session:sessions:2aae307a-0e02-4166-9d0c-be6c5a862a00

key = spring:session:expirations:1496843580000

key = spring:session:sessions:expires:2aae307a-0e02-4166-9d0c-be6c5a862a00

실행해 보면 아래 값이 나온다. 더 자세한 내용은.... 생략

127.0.0.1:6379> hkeys spring:session:sessions:2aae307a-0e02-4166-9d0c-be6c5a862a00

1) "sessionAttr:session-gt1000"

2) "maxInactiveInterval"

3) "lastAccessedTime"

4) "creationTime"

127.0.0.1:6379> hkeys spring:session:expirations:1496843580000

(error) WRONGTYPE Operation against a key holding the wrong kind of value

127.0.0.1:6379> hkeys spring:session:sessions:expires:2aae307a-0e02-4166-9d0c-be6c5a862a00

(error) WRONGTYPE Operation against a key holding the wrong kind of value

**6. Pub/Sub**

- Pub/Sub 시스템에서는 채널에 구독 신청을 한 모든 subscribe에게 메시지를 전송함. 단순히 전송하기 때문에 보관하지 않음.

- JMS와 유사하게 메세지 처리에 활용할 수 있는데 1:1 형태의 Queue 뿐만 아니라, 1:N 형태의 Publish/Subscribe 도 지원.

- Publish는 channel 에 메시지를 보내고, subscribe 는 하나 이상의 channel 로 부터 메시지를 수신

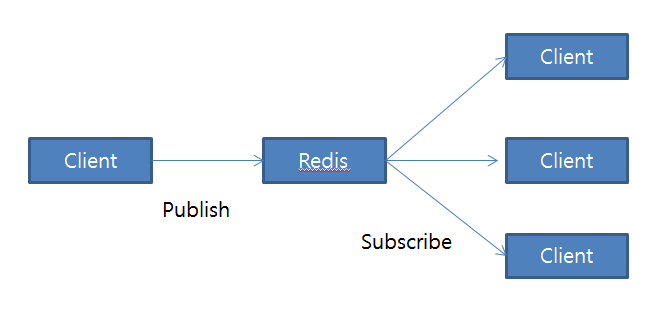
- 메시지를 발송하고 채널에서 구독하는 커맨드를 제공함

- Pub/Sub 구조에서 사용되는 Queue를 일반적으로 Topic 이라고 함.

- 뉴스와 날씨 대시보드, 채팅 어플리케이션, 지하철 지연경고 등의 푸시 알림등에서 사용됨

- 일반적인 Pub/Sub 시스템의 경우 Subscribe 하는 하나의 Topic에서만 Subscribe하는데 반해서, redis에서는 pattern matching을 통해서 다수의 Topic에서 message 를 subscribe할 수 있음

  예를 들어 topic 이름이 job.java job.node 이라는 두개의 Topic이 있을때, "PSUBSCRIBE job.\*"라고 하면 두개의 Topic에서 동시에 message를 subscribe할 수 있음.



예) Subscribe 명령을 이용한 채널 구독

gt1000 채널 구독 신청

[centos@ip-172-31-5-27 ~]$ redis-cli

127.0.0.1:6379> subscribe gt1000

Reading messages... (press Ctrl-C to quit)

1) "subscribe"

2) "gt1000"

3) (integer) 1

1) "message"

2) "gt1000"

3) "hi gt1000"

1) "message"

2) "gt1000"

3) "show me the money!!!!"

gt1003 채널 구독 신청

[centos@ip-172-31-5-27 ~]$ redis-cli

127.0.0.1:6379> subscribe gt1003

Reading messages... (press Ctrl-C to quit)

1) "subscribe"

2) "gt1003"

3) (integer) 1

1) "message"

2) "gt1003"

3) "i love gt1003"

예) Publish 명령을 통한 채널 메시지 발행

[centos@ip-172-31-5-27 ~]$ redis-cli

127.0.0.1:6379> publish gt1000 "hi gt1000"

(integer) 1

127.0.0.1:6379> publish gt1003 "i love gt1003"

(integer) 1

127.0.0.1:6379> publish gt1000 "show me the money!!!!"

(integer) 1

127.0.0.1:6379>

1) Push 메시지 데이터 형식

 - Push 메시지는 3개의 요소들로 구성된 배열값을 전송

* subscribe : 채널을 성공적으로 subscribe 했음
* message : 채널로 부터 전송된 일반 메시지
* unsubscribe : 채널을 성공적으로 unsubscribe 했음

2) Spring Data Pub/Sub

 가. 메시지 발송

  - RedisTemplate의 convertAndSend(channel, message) 를 이용하여 설정된 channel 로 메시지를 전송

인터페이스

public interface MessagePublisher {

void publish(String message);

}

public class RedisMessagePublisher implements MessagePublisher {

@Autowired

private StringRedisTemplate stringRedisTemplate;

@Autowired

private ChannelTopic topic;

public RedisMessagePublisher() {

}

public RedisMessagePublisher(StringRedisTemplate stringRedisTemplate, ChannelTopic topic) {

this.stringRedisTemplate = stringRedisTemplate;

this.topic = topic;

}

public void publish(String message) {

stringRedisTemplate.convertAndSend(topic.getTopic(), message);

}

}

 나. 메시지 수신

  - org.springframework.data.redis.listener.adapter.MessageListenerAdapter 와 org.springframework.data.redis.listener.RedisMessageListenerContainer 를 설정해야 함.

 @Component

public class RedisMessageSubscriber implements MessageListener {

public static List<String> messageList = new ArrayList<String>();

public void onMessage(Message message, byte[] pattern) {

messageList.add(message.toString());

System.out.println("Message received: " + message.toString());

}

}

/\*\*

\* subscriber 역할

\* @return

\*/

@Bean

MessageListenerAdapter messageListener() {

return new MessageListenerAdapter(new RedisMessageSubscriber());

}

/\*\*

\* Redis channel로 부터 메시지를 수신 받아 해당 MessageListenerAdapter에 발송(전파). 등록된 리스너에게 비동기적으로 메시지 발송(전파)

\* @return

\*/

@Bean

RedisMessageListenerContainer redisContainer() {

RedisMessageListenerContainer container = new RedisMessageListenerContainer();

container.setConnectionFactory(jedisConnectionFactory());

container.addMessageListener(messageListener(), topic());

return container;

}

/\*\*

\* 메시지 발송

\* @return

\*/

@Bean

MessagePublisher redisPublisher() {

return new RedisMessagePublisher(stringRedisTemplate(), topic());

}

@Bean

ChannelTopic topic() {

// 채널 이름

return new ChannelTopic("gt1000");

}

@Controller

public class PubSubController {

@Autowired

MessagePublisher redisPublisher;

@GetMapping(value = "/pubsub")

@ResponseBody

String pubsub(Model model) throws Exception {

String message = "Message " + UUID.randomUUID();

redisPublisher.publish(message);

return "success";

}

}

**7. 트랜잭션**

* 레디스의 트랜잭션은 순서대로 원자적으로 실행되는 커맨드의 열.
* MULTI 커맨드는 트랜잭션의 시작을 표시하고, EXEC 커맨드는 트랜잭션의 마지막을 표시.
* MULTI 커맨드와 EXEC 커맨드간의 모든 커맨드는 직렬화되고 원자적으로 실행.
* 레디스는 트랜잭션의 처리 도중에 다른 클라이언트의 요청에 응하지 않음.
* 트랜잭션의 모든 커맨드는 클라이언트의 큐에 쌓이고, EXEC 커맨드가 실행되면 서버로 전달.
* EXEC 대신에 DISCARD 커맨드를 사용하면 트랜잭션은 실행되지 않음.
* 레디스 클라이언트는 저장된 커맨드에서 문법 에러를 발견하면, 레디스 서버로 전달될 트랜잭션이 실행되는 것을 막음.
* 레디스 트랜잭션은 과정을 실행하다가 실패하더라도 롤백하지 않음
* 레디스는 트랜잭션의 커맨드를 순서대로 실행하고, 커맨드 중 일부가 실패하면, 다음 커맨드로 처리.
* 단점은 모든 커맨드가 큐에 쌓이기 때문에 트랜잭션 내부에서 어떠한 결정도 내릴 수 없음

예) 은행 이체

var redis = require("redis");

var client = redis.createClient();

function transfer(from, to, value, callback) {

// 원 계좌의 현재 잔액을 얻음

client.get(from, function(err, balance) {

// transaction start. 큐에 저장됨

var multi = client.multi();

// 키가 존재하지 않을 경우 0으로 초기화 하고, 지정된 값 만큼 뺌

multi.decrby(from, value);

// 지정된 값 만큼 더함

multi.incrby(to, value);

// 금액이 충분한지 확인

if (balance >= value) {

// transaction end

multi.exec(function(err, reply) {

// null 값과 DECRBY 커맨드가 실행된 후 원계좌의 잔액을 전달하고 콜백 함수 실행

callback(null, reply[0]);

});

} else {

// transaction 무시

multi.discard();

callback(new Error("Insufficient funds"), null);

}

});

}

client.mset("gt1000:checkings", 200, "gt1003:checkings", 300, function(err, reply) {

console.log("gt1000 checkings: 200");

console.log("gt1003 checkings: 300");

transfer("gt1000:checkings", "gt1003:checkings", 100, function(err, balance) {

if (err) {

console.log(err);

} else {

console.log("Transferred 100 from gt1000 to gt1003")

console.log("gt1000 balance:", balance);

}

client.quit();

});

});

예)  spring data multi.exec

@Test

public void multiExec() {

redisTemplate.execute(new RedisCallback<Object>() {

@Override

public Object doInRedis(RedisConnection connection) throws DataAccessException {

connection.multi();

for(int i=0; i<10; i++) {

connection.set( ("key" + i).getBytes(), (String.valueOf(i)).getBytes());

}

List<Object> result = connection.exec();

return null;

}

});

}

예) spring data discard

@Test

public void discard() {

redisTemplate.execute(new RedisCallback<Object>() {

@Override

public Object doInRedis(RedisConnection connection) throws DataAccessException {

connection.watch();

connection.multi();

for(int i=100; i<110; i++) {

connection.set( ("key" + i).getBytes(), (String.valueOf(i)).getBytes());

if(i == 105) {

connection.discard();

// 이걸 해 줘야 트랜잭션이 다시 시작됨

connection.multi();

}

}

List<Object> result = connection.exec();

return null;

}

});

}

redis 서버에 106, 107, 108, 109 가 들어감

if(i == 103 || i == 106) {

connection.discard();

// 이걸 해 줘야 트랜잭션이 다시 시작됨

connection.multi();

}

이렇게 하면 107, 108, 109 만 들어감. 앞에 것은 전부 버림

예) spring data redis watch

@Test

public void watch() {

redisTemplate.execute(new RedisCallback<Object>() {

@Override

public Object doInRedis(RedisConnection connection) throws DataAccessException {

connection.watch( ("key:" + 1).getBytes() );

connection.set( ("key:" + 1).getBytes(), (String.valueOf(205)).getBytes());

connection.multi();

for(int i=200; i<210; i++) {

connection.set( ("key:" + i).getBytes(), (String.valueOf(i)).getBytes());

}

List<Object> result = connection.exec();

return null;

}

});

}

redis 서버에 205만 들어감

**8. Redis 확장**

1) 복제

 - 레디스 인스턴스(master)에 쓰기 작업을 진행하는 동안, 하나 이상의 인스턴스(slave)가 레디스 master의 복제본이 되는 것을 복제라고 함

 - 레디스 서버의 인스턴스를 슬레이브로 만드는 3가지 방법

* 설정 파일에 slaveof IP PORT 지시자를 추가하고 해당 설정 파일을 사용해 레디스를 시작
* redis-server 커맨드 라인 옵션에 --savleof IP PORT를 사용
* SLAVEOF IP PORT 커맨드를 사용

master

[centos@ip-172-31-5-27 ~]$ redis-server --port 5555

..... 생략 ......

..... slave1, slave2 추가 후 로그 .....

The server is now ready to accept connections on port 5555

26653:M 10 Jun 05:44:01.844 \* Slave 127.0.0.1:6666 asks for synchronization

26653:M 10 Jun 05:44:01.844 \* Full resync requested by slave 127.0.0.1:6666

26653:M 10 Jun 05:44:01.844 \* Starting BGSAVE for SYNC with target: disk

26653:M 10 Jun 05:44:01.845 \* Background saving started by pid 26683

26683:C 10 Jun 05:44:01.849 \* DB saved on disk

26683:C 10 Jun 05:44:01.849 \* RDB: 6 MB of memory used by copy-on-write

26653:M 10 Jun 05:44:01.930 \* Background saving terminated with success

26653:M 10 Jun 05:44:01.930 \* Synchronization with slave 127.0.0.1:6666 succeeded

26653:M 10 Jun 05:44:49.050 \* Slave 127.0.0.1:7777 asks for synchronization

26653:M 10 Jun 05:44:49.050 \* Full resync requested by slave 127.0.0.1:7777

26653:M 10 Jun 05:44:49.050 \* Starting BGSAVE for SYNC with target: disk

26653:M 10 Jun 05:44:49.050 \* Background saving started by pid 26713

26713:C 10 Jun 05:44:49.054 \* DB saved on disk

26713:C 10 Jun 05:44:49.054 \* RDB: 4 MB of memory used by copy-on-write

26653:M 10 Jun 05:44:49.066 \* Background saving terminated with success

26653:M 10 Jun 05:44:49.066 \* Synchronization with slave 127.0.0.1:7777 succeeded

slave1

[centos@ip-172-31-5-27 ~]$ redis-server --port 6666 --slaveof 127.0.0.1 5555

[centos@ip-172-31-5-27 ~]$ redis-server --port 7777 --slaveof 127.0.0.1 5555

복제 테스트

[centos@ip-172-31-5-27 ~]$ redis-cli -p 5555 set testkey "master slave test"

OK

[centos@ip-172-31-5-27 ~]$ redis-cli -p 6666 get testkey

"master slave test"

[centos@ip-172-31-5-27 ~]$ redis-cli -p 7777 get testkey

"master slave test"

[centos@ip-172-31-5-27 ~]$

 - 복제본은 레디스를 확장하기 위해 광범위하게 쓰이며, 모든 읽기 작업은 복제본에서 처리되고 마스터는 쓰기 명령만 처리

[centos@ip-172-31-5-27 ~]$ redis-cli -p 6666 set testkey2 "test2"

(error) READONLY You can't write against a read only slave.

 - 레디스가 싱글 인스턴스에서 동작하면, 슬레이브에서 마스터로 승격할 수 있는 failover가 자동으로 수행되지 않음. 자동 failover는 Redis Sentinel의 핵심

[centos@ip-172-31-5-27 ~]$ redis-cli -p 5555 DEBUG SEGFAULT

Error: Server closed the connection

[centos@ip-172-31-5-27 ~]$ redis-cli -p 6666 SLAVEOF NO ONE

OK

[centos@ip-172-31-5-27 ~]$ redis-cli -p 7777 SLAVEOF 127.0.0.1 6666

OK

[centos@ip-172-31-5-27 ~]$ redis-cli -p 6666 set testkey2 "test2"

OK

[centos@ip-172-31-5-27 ~]$ redis-cli -p 7777 get testkey2

"test2"

2) 파티셔닝

* 데이터를 나누고, 여러 장비에 걸쳐 데이터를 분산 시키는 행위
* 수평 파티셔닝, 수직 파티셔닝이 있음

수평 파티셔닝(Sharding) - 키를 여러 레디스 인스턴스로 걸쳐 분산 시키는 것을 의미  
수직 파티셔닝 - 여러 레디스 인스턴스로 키의 값을 분산하는 것을 의미

* 더 좋은 성능, 유지 보수성, 가용성이 요구될 때 장비 클러스터에서 수행
* 레디스를 처음 설계 했을때는 분산해서 저장할 의도가 없었으나 하나의 서버로는 모든 데이터를 저장하기 어려울 정도로 커지거나  
  네트웍 대역폭이 모든 트래픽을 처리하기에 충분하지 않게 되었음.

 가. 범위 파티셔닝

* 키의 범위를 기반으로 데이터를 분산.  
  user1 부터 user5000 까지의 id 를 가지는 사용자 그룹이 있다고 하면  
  user1 ~ user1000 은 1번 인스턴스, user1001 ~ user2000 은 2번 인스턴스로 분산
* 키의 첫번째 문자를 기반으로 분리 하는 방법도 있음  
  A ~ G는 1번, H ~ O 는 2번 등
* 단점은 분포가 고르지 않다는 것
* 레디스 인스턴스 개수가 변경되면 범위 분포가 그에 따라 변경되기 때문에 레디스 장비 목록의 변경을 쉽게 수용하지 못한다는 점.  
  장비를 추가 하거나 삭제 할때 상당한 양의 데이터를 무효화 할 수 있음

 나. 해시 파티셔닝

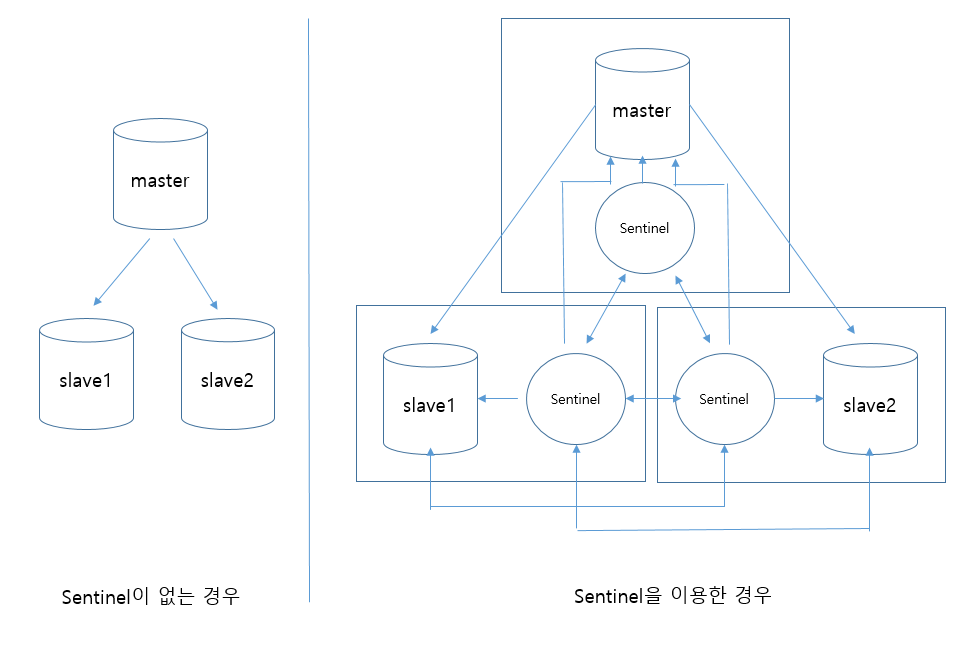
* 범위 파티셔닝보다 더 정교하고 범위 파티셔닝의 단점인 고르지 않은 데이터 분포를 갖지 않음.
* 레디스 커맨드를 보내기 위해 레디스 키에 해시 함수를 적용하 ㄴ후, 해당 해시 값을 사용 가능한 레디스 인스턴스의 개수로 나누고  
  나눈 값의 나머지를 인스턴스의 인덱스로 사용함으로써 레디스 인스턴스를 찾는 것으로 구성

 다. 미리 샤딩하기

* 해시 파티셔닝에서 시간이 경과함에 따라 인스턴스를 추가하고 삭제하는 문제를 처리 하는 방법 중 하나는 데이터를 미리 샤딩하는 것
* 데이터를 많은 인스턴스로 미리 파티셔닝해 장비 목록의 개수를 절대로 변하지 않게 한다는 의미.

3) 레디스 센티널

* master 인스턴스에서 문제가 발생하면, master의 slave 인스턴스 중 하나가 master로 승격돼야 하고, 나머지 slave 인스턴스  
  새로운 master에 대한 slave로 재구성 할 수 있도록 설정을 변경해야 함.  
  레디스 센티널이 있기전에는 수동으로 failover 작업을 했음
* master가 모든 데이터를 갖고 있고, slave는 master에 대한 복사본을 가지고 있음.  
  레디스 센터널은 노드에 데이터를 분산하지 않음. 분산 데이터 저장소가 아님.
* 일반적인 아키텍처는 하나의 레디스 서버마다 하나의 레디스 센티널을 설치 하는 것.  
  레디스 서버와 분리된 프로세스며, 서버와는 다른 포트를 사용



 가. 기본 설정

  - redis 설치 디렉토리 / sentinel.conf

  - redis sentinel 기본 설정은 이름 식별을 갖고 ip와 포트로 식별할 수 있는 master를 항상 모니터링 함.

# 포트

port 26379

# redis master 정보. mymaster는 alias, quorum = 2(failover시 master가 죽었음을 다수결로 결정, 2대의 sentinel이 master에 연결이 안될때 failover)

sentinel monitor mymaster 127.0.0.1 5555 2

# 30초 동안 master 연결이 실패하면 failover. Default is 30 seconds.

sentinel down-after-milliseconds mymaster 30000

# Default is 3 minutes.

sentinel failover-timeout mymaster 180000

# failover가 발생했을때, slave들이 새 master로 부터 데이터를 싱크 받는 개수. 1개씩 순차적으로 master와 sync 하겠다는 의미

sentinel parallel-syncs mymaster 1

sentinel auth-pass mymaster 12345678

# 대몬으로 돌리기 위해서 추가

daemonize yes

pidfile /var/run/sentinel\_26379.pid   
logfile /var/log/sentinel\_5555.log

  - redis 서버가 n대 있다고 하면 이 파일을 각각의 서버에 복사해서 port 만 수정

    redis 의 master,slave 들의 각 conf 파일의 masterauth 와 requirepass 값 수정.  
    sentinel 에 의해서 failover 시에 master 가 slave 가 되고, slave 가 master 가 되는 작업들이 뒤죽박죽 진행이 될텐데, 이때 모든 redis 들의 비밀번호가 같아야 되지 함.  
    sentinel 은 master / slave 관계만 설정해 줄 뿐, 비밀번호를 알아내고, 설정해 주지는 않기 때문. 모든 redis 의 비밀번호를 동일하게 설정

한 서버에서 실행 할때는 sentinel.conf 파일을 다음과 같이 복사해서 사용할 수 있음

$ ./redis-sentinel 26379.conf &  
$ ./redis-sentinel 26380.conf &  
$ ./redis-sentinel 26381.conf &

ps -ef | grep redis를 콘솔에 입력해보면 아래처럼 여섯개의 인스턴스가 잡힙니다.

./redis-server \*:5555

./redis-server \*:6666

./redis-server \*:7777

./redis-sentinel \*:26379 [sentinel]  
./redis-sentinel \*:26380 [sentinel]  
./redis-sentinel \*:26381 [sentinel]

 나. 네트워크 파티션(split-brain)

  - 두 시스템 그룹 간의 모든 네트워크 연결이 동시에 실패하면 네트워크 파티션이 발생합니다.   
    이러한 상태가 발생하면 파티션 양쪽의 시스템이 각각 반대쪽에서 응용 프로그램을 재시작하여 중복 서비스 또는 "스플릿 브레인(split-brain)"을 유발합니다.   
    스플릿 브레인(split-brain)은 클러스터로 구성된 독립적인 두 시스템이 특정 리소스(일반적으로 파일 시스템 또는 볼륨)에 대한 배타적 액세스 권한이 있다고 가정할 경우에 발생합니다.   
    네트워크 파티션으로 인해 발생하는 가장 심각한 문제는 공유 디스크의 데이터에 영향을 준다는 점입니다.

  - 3개의 redis 인스턴스가 있다고 가정할때 네트워크 파티션이 발생해 현재 master가 slave로 부터 분리되고, slave 끼리만 통신할 수 있다면  
    slave 중 하나가 master 로 승격됨.   
    이 상황에서 redis client 는 격리된 master에 쓰기 작업을 계속 한다면 네트워크가 잘 동작해서 모든 redis 간 통신이 잘 된다면 대부분 sentinel은

    이전 master가 새로운 master 의 slave가 되는것에 동의 함.  
    이 경우 프로세스 간 데이터 동기화가 이루어지지 않기 때문에 redis client가 전달한 모든 쓰기 작업 data가 유실됨

  - redis sentinel은 높은 가용성과 관련된 자동 failover 문제를 해결 하지만, 다중 redis 인스턴스 간의 데이터 분산 문제는 해결하지 못함.

    redis cluster 는 다른 접근 방법으로 이 문제를 해결함.

4) 레디스 클러스터

 - 레디스 클러스터는 여러 레디스 인스턴스에 데이터를 자동으로 샤딩할 수 있도록 설계됐고, 네트워크 파티션이 발생하더라도 어느 정도의 가용성을 제공함

 - 레디스가 사용하는 포트는 2개

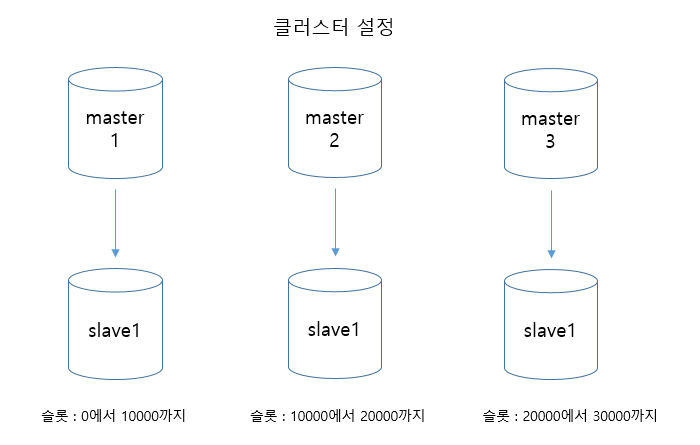
   첫번째 포트(숫자가 낮은 포트)는 클라이언트 통신을 위해 사용됨

   두번째 포트(숫자가 높은 포트)는 문제 탐지, 페일 오버, 재샤딩 등과 같은 메시지 교환을 위해 사용됨

 - 레디스 클러스터 버스는 레디스 노드간의 메시지를 교환하기 위해 바이너리 프로토콜을 사용함.  
   숫자가 낮은 포트는 설정에서 명세하고, 레디스는 숫자가 낮은 포트에 10, 000 을 더해 높은 포트를 할당

 - 마스터 마다 최소 하나의 복제본을 두는 것을 권하며, failover가 클러스트에서 발생하면, slave가 master로 승격 할때까지 문제가 발생한 master로 할당된  
   슬롯의 키는 사용할 수 없음.

 - redis-cli를 통해 레디스에 연결할 때 클러스터 모드로 실행하려면 -c 매개변수가 필요.



**9. 기타**

1) 파이프라인

 - 다중 커맨드를 서버에 한꺼번에 보내는 방법을 말하며, 개별 응답을 기다리지 않고 클라이언트에 의해 응답을 한번에 읽을 수 있음

   레디스 클라이언트가 커맨드를 보내고 레디스 서버로부터 응답을 받는데 걸리는 시간을 Round Trip Time 이라 함

   열개의 커맨드를 파이프라인으로 이용하면 하나의 RTT가 됨

   예)

   var redis = require("redis");

   var client = redis.createClient();

   client.set("key1", "value1");

   client.set("key2", "value2");

   client.set("key3", "value3");

 2) 기타 커맨드

* info 커맨드는 레디스 버전과 운영체제, 연결된 클라이언트, 메모리 사용량, 자정소, 복제본, 카스페이스에 대한 정보를 포함한 모든 레디스 서버 통계를 리턴

        ex) info momory

* DBSIZE는 레디스 서버에 존재하는 키 개수를 리턴
* DEBUG SEGFAULT는 올바르지 않은 메모리 접근을 수행해 레디스 서버 종료. 애플리케이션 개발 중에 버그를 시뮬레이션 할때 유용
* MONITOR는 서버가 처리하는 모든 커맨드를 실시간으로 보여줌  
  레디스 서버가 얼마나 바쁜지 확인할 때 유용. 디버깅시 매우 유용하지만 비용이 소모됨.  
  레디스 공식 문서 페이지에는 비과학적인 벤치마크 테스트에서 MONITOR 커맨드가 레디스의 성능을 50% 이상까지 떨어뜨릴수 있다고 언급
* CLIENT LIST는 클라이언트에 대한 관련 정보와 통계뿐 아니라 서버에 연결된 모든 클라이언트 목록을 리턴
* client setname 는 클라언트의 이름을 변경. 디버깅 목적시 유용
* CLIENT KILL 클라이언트 연결 종료. IP, 포트, ID, 타입으로 가능  
  ex) CLIENT KILL ADDR 127.0.0.1:12345
* FLUSHALL 레디스의 모든 키를 삭제. 삭제된 키는 복구 불가
* RANDOMKEY 존재하는 키 이름 중 무작위로 선택한 하나의 키 이름을 리턴
* EXPIRE 특정키의 타임아웃을 초 단위로 설정. 레디스는 명세한 초가 지난 후에 키를 삭제. 음수의 타임아웃은 키를 바로 삭제
* EXPIREAT 유닉스 타임스탬프를 기반으로 특정 키의 타임아웃을 설정함. 설정한 타임스탬프가 과거라면 즉시 키를 삭제
* TTL 타임아웃 값이 있는 키의 남아있는 생존 시간을 초 단위로 리턴
* PTTL TTL 커맨드와 동일하지만 리턴값은 초가 아니라 밀리초
* PERSIST 특정키에 주어진 현존 타임아웃을 제거. 새로운 타임아웃을 설정하지 않으면 키는 만료되지 않음  
  키의 타임아웃을 삭제하면 1을 리턴, 키의 타임아웃이 존재하지 않으면 0을 리턴
* SETEX  특정키에 값을 저장할때 만료 시간도 함께 원자적으로 설정함. SET 커맨드와 EXPIRE 커맨드의 조합
* DEL 하나 이상의 키를 레디스에서 삭제하고 삭제된 키의 개수를 리턴함
* EXISTS 특정 키가 존재하면 1을 리턴하고, 특정키가 존재하지 않으면 0을 리턴
* PING 'PONG" 문자열을 리턴, 서버/클리이언트 연결 테스트.
* MIGRATE 특정 키를 대상 레디스 서버로 옮긴다. 원자적이어서 키를 옮기는 동안 원본 레디스 서버와 키를 저장할 레디스 서버가  
  둘다 블록, 키를 저장할 서버에 키가 이미 존해하면 실패함
* SELECT 다중 데이터베이스 개념. 기본 16개, 0에서 15까지의 숫자로 식별  
  하나의 서버를 사용하는 것보다 여러개의 redis-server 프로세스를 사용하는 것이 좋음
* AUTH  레디스에 연결할 수 있는 클라이언트의 허가하는데 사용.
* SCRIPT KILL 스크립트가 더 이상 쓰기 작업을 수행하지 않으면 루아 스크립트 실행을 종료.
* SHUTDOWN 모든 클라이언트를 종료하고 최대한 데이터를 저장하려고 한 후 레디스 서버를 종료
* OBJECT ENCODING 커맨드는 주어진 키에서 사용 중인 인코딩 값을 리턴

3) 데이터 타입의 최적화

* 문자열  
  int : 64b 비트 부호있는 정수로 문자열을 표현할때 int가 사용됨  
  set str1 1234  
  embstr : 40바이트 이하 작은 문자열  
  set str2 "an embstr is small"  
  object encoding str2  
  raw : 40 바이트보다 큰 문자열을 표기  
  set str3 " a raw encoded string is anything greater than 39 bytes"  
  object encodeing str3  
  "raw"
* 리스트  
  ziplist : 리스트 크기의 엘리먼트가 list-max-ziplist-entries 설정보다 작고, 리스트의 개별 엘리먼크의 바이트가  
  list-max-ziplist-value 설정보다 작다면 ziplist가 사용  
  linkedlist : 리스크 크기의 엘리먼트가 list-max-ziplist-entries 설정보다 크거나  
  리스트의 개별 엘리먼트의 바이트가 list-max-ziplist-value 설정보다 크면 연결리스트가 사용됨
* 셋  
  intset : 모든 엘리먼트가 정수고, 셋의 개수가 set-max-interset-entries 설정보다 작으면 인트셋이 사용  
  hashtable : 엘리먼트중 하나라도 정수가 아니거나, 셋의 개수가 set-max-interset-entries 설정보다 크면 사용
* 해시  
  집리스트 : 해시의 필드 개수가 hash-max-ziplist-entiries 설정보다 작고 해시의 필드 이름과 값이  
  hash-max-ziplist-value 설정보다 작으면 집리스트가 사용  
  해시테이블 : 해시의 필드 개수가 hash-max-ziplist-entries 설정보다 크거나 해시의 필드값 중 하나라도  
  hash-max-ziplist-value 설정보다 크면 해시테이블이 사용됨
* 정렬된 셋  
  집리스트 : 정렬된 셋의 개수가 set-max-ziplist-entries 설정보다 작고, 정렬된 셋의 엘리먼트 값이 모드 zset-max-ziplist-value  
  설정보다 작으면 집리스트가 사용  
  스킵리스트와 해시테이블 : 정렬된 셋의 개수가 set-max-ziplist-entries 설정보다 크거나 정렬된 셋의 엘리먼트 값 중  
  하나라도 zset-max-ziplist-value 설정보다 크면 스킵리스트와 해시테이블이 사용

4) 영속성

* 레디스는 지속성을 보장하기 위해 데이터를 DISK에 저장할 수 있음.
* 서버가 내려가더라도 DISK에 저장된 데이터를 읽어서 메모리에 로딩함.
* 데이터를 DISK에 저장하는 방식은 크게 두 가지 방식이 있음.

        \* Snapshotting (RDB) 방식  
          순간적으로 메모리에 있는 내용을 DISK에 전체를 옮겨 담는 방식          
        \* AOF(Append On File) 방식  
          Redis의 모든 write/update 연산 자체를 모두 log 파일에 기록하는 형태