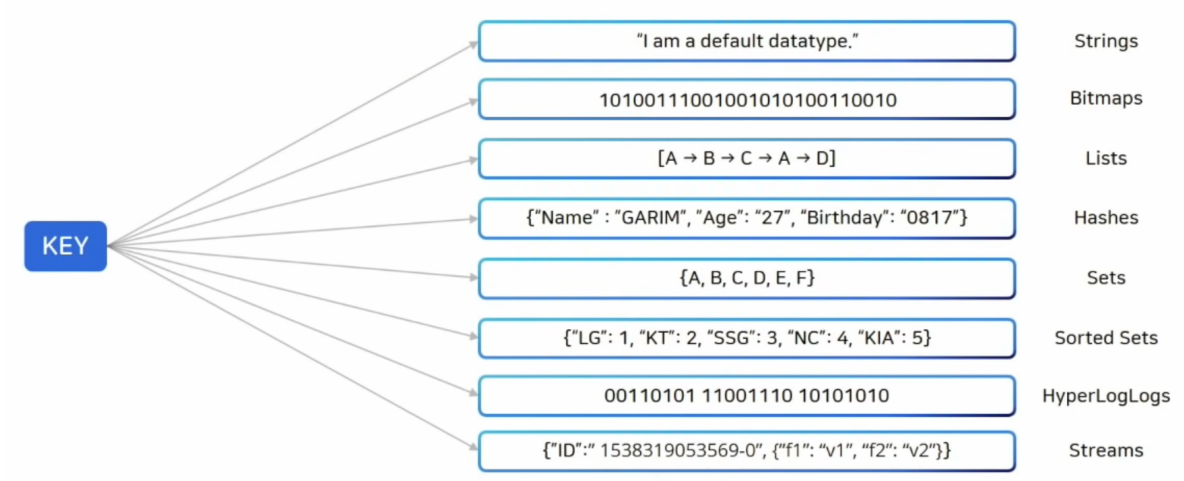
Redis Week1 [data types]

Commands	Version	Syntax	Description
PFADD	2.8.9	key ele [ele ...]	원소(element) 추가
PFCOUNT	2.8.9	key [key ...]	원소 개수 조회
PFMERGE	2.8.9	destkey sourcekey [sourcekey ...]	집합 머지(Merge)

Redis의 장점 중 하나는 Key-Value 스토리지에서 Value는 단순한 Object가 아니라 다양한 자료구조를 갖기 때문이다.

String, Set, Sorted Set, Hash, List 등 다양한 타입을 지원한다.



Strings

- redis 문자열은 텍스트, 직렬화된 개체 및 이진 배열을 포함하여 바이트 시퀀스를 저장한다.
- key / value가 일대일 관계이다. (반대로 Lists, Sets, Sorted Sets, Hashes는 일대다 관계이다.)
- key와 value 모두 최대 길이는 512MB이다.(cluster 모드에서 약 250MB가 넘어가면 리밸런싱을 못하는 이슈는 있음 이럴 경우 직접 문제가 되는 키가 있는 노드에 접속해서 삭제를 해야함)
- 적절한 key 사이즈를 유지하는 것이 좋다.
  - 사용자 보기 관점: "user_1000_email”
  - 메모리 절약 관점: "u1000e"
- key를 구성할땐 ‘_’ 를 사용해서 key를 구성하는 것이 좋다(이건 엔터프라이즈 버전에서 SQL 사용하는 것땜에 그럼)

command

- **SET**: SET, SETNX, SETEX, SETPEX, MSET, MSETNX, APPEND, SETRANGE
- **GET**: GET, MGET, GETRANGE, STRLEN
- **INCR**: INCR, DECR, INCRBY, DECRBY, INCRBYFLOAT

- **Enterprise**: SETS, DELS, APPENDS (subquery)

Commands	Version	Syntax	Description
DECR	1.0.0	key	1씩 감가, 신규이면 -1로 setting.
DECRBY	1.0.0	key decrement	decrement만큼 감소. 신규이면 -decrement로 setting.
DEL	1.0.0	key [key ...]	데이터를 삭제
GET	1.0.0	key	데이터를 조회
GETSET	1.0.0	key value	기존 데이터를 조회하고 새 데이터를 저장
INCR	1.0.0	key	1씩 증가, 신규이면 1로 setting.
INCRBY	1.0.0	key increment	increment만큼 증가. 신규이면 increment로 setting.
MGET	1.0.0	key [key ...]	여러개의 데이터를 한번에 조회
SET	1.0.0	key value [EX seconds][PX milliseconds] [NX|XX]	데이터를 저장, key가 이미 있으면 덮어쓴다.
SETNX	1.0.0	key value	지정한 key가 없을 경우에만 데이터를 저장(잠금기능 구현할때 사용)
MSET	1.0.1	key value [key value ...]	여러개의 데이터를 한번에 저장
MSETNX	1.0.1	key value [key value ...]	지정한 key가 없을 경우에만, 여러개의 데이터를 한번에 저장
APPEND	2.0.0	key value	데이터를 추가, 지정한 key가 없으면 저장
SETEX	2.0.0	key seconds value	지정한 시간(초) 이후에 데이터 자동 삭제
SETRANGE	2.2.0	key offset value	지정한 위치(offset)부터 데이터를 겹쳐쓴다
STRLEN	2.2.0	key	데이터의 바이트수를 리턴
GETRANGE	2.4.0	key start end	데이터의 일부 문자열을 조회
INCRBYFLOAT	2.6.0	key increment	실수연산, increment만큼 증가. 신규이면 increment로 setting.
PSETEX	2.6.0	key milliseconds value	지정한 시간(밀리초) 이후에 데

			이터 자동 삭제
STRALGO	6.0.0	STRALGO LCS	두 문자열이 얼마나 유사한지 평가
GETEX	6.2.0	key [EX seconds]	데이터 조회와 만료 시간 설정
GETDEL	6.2.0	key	데이터 조회와 삭제
SETS	Ent 7.2.5	key (subquery)	SETS key (subquery)
DELS	Ent 7.2.5	(subquery)	DELS (subquery)
APPENDS	Ent 7.2.5	(subquery)	APPENDS key (subquery)

example

```
# 한개 조회
set <key> <value>
get <key> <value>

# 여러개 조회
mset <key> <value> <key> <value> ...
mget <key> <key> <key> ...



127.0.0.1:6379> set hello "world!"
OK

127.0.0.1:6379> get hello
"world!"

127.0.0.1:6379> get count
"-351"

127.0.0.1:6379> set count 50
OK

127.0.0.1:6379> incr count
(integer) 51

127.0.0.1:6379> get count
"51"

127.0.0.1:6379> incrby count 100
(integer) 151

127.0.0.1:6379> decr count
(integer) 150

127.0.0.1:6379> decrby count 500
(integer) -350
```

```
127.0.0.1:6379> mset a "hello" b "world"
OK

127.0.0.1:6379> mget a b
1) "hello"
2) "world"
```

- 직렬화된 JSON 문자열을 저장하고 지금부터 100초 후에 만료되도록 설정.
-

```
> SET ticket:27 "\"{'username': 'priya', 'ticket_id': 321}\"" EX 100
```

Lists

- 문자열 값의 연결된 목록으로 다음과 같은 용도로 주로 사용된다.
  - 스택과 큐 구현
  - 백그라운드 작업 시스템을 위한 대기열
- 추가 / 삭제 / 조회하는 것은 O(1)의 속도를 가지지만, 중간의 특정 index 값을 조회할 때는 O(N)의 속도를 가지는 단점이 있다.
- 즉, 중간에 추가/삭제가 느리다. 따라서 head-tail에서 추가/삭제 한다. (push / pop 연산)
-

command

- **SET (PUSH)**: LPUSH, RPUSH, LPUSHX, RPUSHX, LSET, LINSERT, RPOPLPUSH
- **GET**: LRANGE, LINDEX, LLEN

- **POP**: LPOP, RPOP, BLPOP, BRPOP
- **REM**: LREM, LTRIM
- **BLOCK**: BLPOP, BRPOP, BRPOPLPUSH
- **Enterprise**: LREVRANGE, LPUSHS, RPUSHS (subquery)

Commands	Version	Syntax	Description
LPUSH	1.0.0	key value [value ...]	왼쪽에서 리스트의 오른쪽에 데이터를 저장
RPOP	1.0.0	key	리스트 오른쪽에서 데이터를 꺼내오고, 리스트에서는 삭제
LPOP	1.0.0	key	리스트 왼쪽에서 데이터를 꺼내오고, 리스트에서는 삭제
RPUSH	1.0.0	key value [value ...]	오른쪽에서 리스트의 왼쪽에 데이터를 저장
LRANGE	1.0.0	key start stop	인덱스로 범위를 지정해서 리스트 조회
LLEN	1.0.0	key	리스트에서 데이터의 총 갯수를 조회
LINDEX	1.0.0	key index	인덱스로 특정 위치의 데이터를 조회
LSET	1.0.0	key index value	인덱스로 특정 위치의 값을 바꿈
LREM	1.0.0	key count value	값을 지정해서 삭제
LTRIM	1.0.0	key start stop	인덱스로 지정한 범위 밖의 값들을 삭제
RPOPLPUSH	1.2.0	key src_key dest_key	RPOP + LPUSH
BLPOP	2.0.0	key [key ...] timeout	리스트에 값이 없을 경우, 지정한 시간만큼 기다려서 값이 들어오면 LPOP 실행
BRPOP	2.0.0	key [key ...] timeout	리스트에 값이 없을 경우, 지정한 시간만큼 기다려서 값이 들어오면 RPOP 실행
BRPOPLPUSH	2.2.0	src_key dest_key timeout	리스트에 값이 없을 경우, 지정한 시간만큼 기다려서 값이 들어오면 RPOPLPUSH 실행
LINSERT	2.2.0	key BEFORE|AFTER pivot value	지정한 값 앞/뒤에 새 값 저장

LPUSHX	2.2.0	key value	기존에 리스트가 있을 경우에만 LPUSH 실행
RPUSHX	2.2.0	key value	기존에 리스트가 있을 경우에만 RPUSH 실행
LPOS	6.0.6	key element	값으로 인덱스를 조회
LMOVE	6.2.0	source destination	리스트간 데이터 이동
BLMOVE	6.2.0	source destination	리스트간 데이터 이동 - 대기
LLS	Ent 6.0.0	key pattern	패턴(pattern)으로 값(value) 조회
LRM	Ent 6.0.0	key pattern	패턴(pattern)으로 값(value) 삭제
LREVRANGE	Ent 6.0.0	key key start stop	인덱스로 범위를 지정해서 역순으로 조회
LPUSHS	Ent 7.2.5	key (subquery)	서브쿼리로 데이터를 저장
RPUSHS	Ent 7.2.5	key (subquery)	서브쿼리로 데이터를 저장

example

```
# 왼쪽에 삽입
lpush <key> <value>

# 오른쪽에 삽입
rpush <key> <value>

# 삭제
lpop <key>
rpop <key>

# LPUSH를 통한 list 생성
127.0.0.1:6379> LPUSH myList "a"
(integer) 1
127.0.0.1:6379> LRANGE myList 0 -1
1) "a"

# LPUSH , RPUSH를 통한 요소 삽입 결과.
127.0.0.1:6379> LPUSH myList "b"
(integer) 2
127.0.0.1:6379> RPUSH myList "c"
(integer) 3
127.0.0.1:6379> LRANGE myList 0 -1
1) "b"
2) "a"
3) "c"
```

```
# LPUSHX , RPUSHX 사용 예
# key가 없는곳에 추가할려고 하는 경우 0을 반환.
127.0.0.1:6379> LPUSHX myList2 "a"
(integer) 0
127.0.0.1:6379> LPUSH myList "d"
(integer) 4


# 기존에 있던 myList요소들은 [ d , b , a , c ] 순으로 되어있음.
# LPOP , RPOP을 통해 맨 좌 우측 요소 한개씩 제거
127.0.0.1:6379> LPOP myList
"d"
127.0.0.1:6379> RPOP myList
"c"
127.0.0.1:6379> LRANGE myList 0 -1
1) "b"
2) "a"

# 현재 List의 요소 길이를 출력.
127.0.0.1:6379> LLEN myList
(integer) 2

# LREM을 통핸 해당 요소 삭제 , count를 0 으로 해서 요소중에 "a"랑 매칭되는 값을 삭제
127.0.0.1:6379> LREM myList 0 "a"
(integer) 1
127.0.0.1:6379> LRANGE myList 0 -1
1) "b"

# 해당 key에 해당되는 index값을 입력받은 값을 수정.
# 현재 b로 남아있던 요소값을 z로 변경
127.0.0.1:6379> LSET myList 0 "z"
OK
127.0.0.1:6379> LRANGE myList 0 -1
1) "z"

# RPOPLPUSH
127.0.0.1:6379> RPOPLPUSH myList hello
"z"
127.0.0.1:6379> LRANGE hello 0 -1
1) "z"

# 한 목록에서 리스트를 원자적으로 팝하고 다른 리스트로 푸시
> LPUSH board:todo:ids 101
(integer) 1
> LPUSH board:todo:ids 273
(integer) 2
> LMOVE board:todo:ids board:in-progress:ids LEFT LEFT
"273"
> LRANGE board:todo:ids 0 -1
1) "101"
> LRANGE board:in-progress:ids 0 -1
1) "273"
```

- 짚 리스트(ZIP LIST): 메모리 절약형 데이터 구조
  - 짚 리스트는 LIST, SORTED SET, HASH에서 공통으로 사용되는 데이터 구조로,
  - 데이터 구조와 주요 오퍼레이션에 대해서 알아봅니다.
  - 그리고 LIST에서 어떻게 사용되는지 살펴봅니다.
- 리스트의 메인 데이터 구조 : 링크드 리스트(Linked List)링크드 리스트의 데이터 구조와 주요 오퍼레이션에 대해서 알아봅니다.
- 퀵 리스트(Quick List)버전 3.2 전에는 엔트리 개수가 512개 이하면 짚 리스트에 저장되고, 513개부터는 링크드 리스트에 저장되었다.레디스 버전 3.2부터 리스트(LIST)의 내부 데이터 타입으로 퀵 리스트 하나로 확정되었다.무엇이 달라졌는지 알아봅시다.

Sets

- Value는 입력된 순서와 상관없이 저장되며, 중복되지 않는다. 즉, value A가 2번 저장되도 결과적으로 하나만 남는다.
- Sets에서는 집합이라는 의미에서 value를 member라 부른다.
- 교집합, 합집합 및 차이와 같은 일반적인 집합 연산을 수행한다.
- 중복된 데이터를 여러번 저장하면 최종 한번만 저장된다.
- 고유한 항목을 추적하는데 사용한다.(게시물에 접속하는 모든 고유한 IP 주소)
- 모든 데이터를 전부 다 가지고 오는 명령이 있으므로 주의해야한다.

command

- **SET**: SADD, SMOVE
- **GET**: SMEMBERS, SCARD, SRANDMEMBER, SISMEMBER, SSCAN
- **POP**: SPOP
- **REM**: SREM
- **집합연산**: SUNION, SINTER, SDIFF, SUNIONSTORE, SINTERSTORE, SDIFFSTORE
- **Enterprise**: SLS, SRM, SLEN, SADDS (subquery)

Commands	Version	Syntax	Description
SADD	1.0.0	key member [member ...]	집합에 member를 추가
SREM	1.0.0	key member [member ...]	집합에서 member를 삭제
SMEMBERS	1.0.0	key	집합의 모든 member를 조회
SCARD	1.0.0	key	집합에 속한 member의 갯수를 조회
SUNION	1.0.0	key [key ...]	합집합을 구함
SINTER	1.0.0	key [key ...]	교집합을 구함
SDIFF	1.0.0	key [key ...]	차집합을 구함
SUNIONSTORE	1.0.0	dest_key src_key [src_key ...]	합집합을 구해서 새로운 집합에 저장
SINTERSTORE	1.0.0	dest_key src_key [src_key ...]	교집합을 구해서 새로운 집합에 저장
SDIFFSTORE	1.0.0	dest_key src_key [src_key ...]	차집합을 구해서 새로운 집합에 저장
SISMEMBER	1.0.0	key member	집합에 member가 존재하는지 확인
SMOVE	1.0.0	src_key dest_key member	소스 집합의 member를 목적 집합으로 이동
SPOP	1.0.0	key [count]	집합에서 무작위로 member를 가져옴
SRANDMEMBER	1.0.0	key [count]	집합에서 무작위로 member를 조회
SSCAN	2.8.0	key cursor [MATCH pattern][COUNT count]	member를 일정 단위 갯수만큼씩 조회
SMISMEMBER	6.2.0	key member [member ...]	집합에 member가 존재하는지 확인 - 여러 개 가능
SLS	Ent 6.0.0	key pattern	패턴(pattern)으로 값(value) 조회
SRM	Ent 6.0.0	key pattern	패턴(pattern)으로 값(value) 삭제
SLEN	Ent 7.0.0	key	키에 속한 멤버 개수를 리턴
SADDS	Ent 7.2.5	key (subquery)	서브쿼리로 member를 추가

example

```
sadd <key> <item>

# 존재 여부를 체크, 있으면 1 없으면 0 반환
sismember <key> <item>

# 삭제
srem <key> <value>

# key의 모든 item 조회
smembers <key>


127.0.0.1:6379> sadd myset a # 추가된 member 갯수 반환
(integer) 1

127.0.0.1:6379> sadd myset a
(integer) 0

127.0.0.1:6379> sadd myset b
(integer) 1

127.0.0.1:6379> sadd myset c
(integer) 1

127.0.0.1:6379> srem myset c # 삭제된 member 갯수 반환
(integer) 1

127.0.0.1:6379> smembers myset
1) "b"
2) "a"

127.0.0.1:6379> scard myset
(integer) 2

127.0.0.1:6379> sadd myset c d e f # 여러 member 삽입 가능
(integer) 4

127.0.0.1:6379> smembers myset
1) "d"
2) "c"
3) "a"
4) "f"
5) "b"
6) "e"

127.0.0.1:6379> spop myset 3 # 랜덤 member 삭제
1) "d"
2) "c"
3) "f"

127.0.0.1:6379> smembers myset
1) "a"
2) "b"
3) "e"
```

- 대부분의 명령어는 O(1)이지만, 멤버가 너무 많을 경우 SMEMBERS 명령을 실행할때 O(n)이므로 전체 집합을 단일 명령으로 리턴한다. 그러므로 SSCAN으로 반복 검색해야한다.
- INTSET: SET의 멤버가 정수일때 메모리를 절약하기 위한 데이터 구조

Hashes

- 필드-값 쌍의 컬렉션으로 구성된 자료형
- key 하위에 subkey를 이용해 추가적인 Hash Table을 제공하는 자료구조
- 메모리가 허용하는 한, 제한없이 field들을 넣을 수가 있다.

command

- **SET**: HSET, HMSET, HSETNX
- **GET**: HGET, HMGET, HLEN, HKEYS, HVALS, HGETALL, HSTRLEN, HSCAN, HEXISTS
- **REM**: HDEL
- **INCR**: HINCRBY, HINCRBYFLOAT

Commands	Version	Syntax	Description
HSET	2.0.0	key field value	Field와 value를 저장
HDEL	2.0.0	key field [field ...]	Field로 value를 삭제
HGET	2.0.0	key field	Field로 value를 조회
HLEN	2.0.0	key	Field 갯수 조회
HMSET	2.0.0	key field value [field value	여러개의 field와 value를 저

		...]	장
HMGET	2.0.0	key field [field ...]	여러개의 value를 조회
HKEYS	2.0.0	key	Key에 속한 모든 field name을 조회
HVALS	2.0.0	key	Key에 속한 모든 value를 조회
HGETALL	2.0.0	key	Key에 속한 모든 field와 value을 조회
HINCRBY	2.0.0	key field increment	value를 increment 만큼 증가 또는 감소
HEXISTS	2.0.0	key field	Field가 있는지 확인
HSETNX	2.0.0	key field value	Field가 기존에 없으면 저장
HINCRBYFLOAT	2.6.0	key field increment_float	value를 increment_float 만큼 증가 또는 감소
HSCAN	2.8.0	key cursor [MATCH pattern][COUNT count]	Field, member를 일정 단위 갯수 만큼씩 조회
HSTRLEN	3.2.0	key field	value의 길이(byte)를 조회

example

```
# 한개 값 삽입 및 삭제
hset <key> <subkey> <value>
hget <key> <subkey>

# 여러 값 삽입 및 삭제
hmset <key> <subkey> <value> <subkey> <value> ...
hnget <key> <subkey> <subkey> <subkey> ...

# 모든 subkey와 value 가져오기, Collection에 너무 많은 key가 있으면 장애의 원인이 됨
hgetall <key>

# 모든 value값만 가져오기
hvlas <key>

# field - value : name - jinmin / year - 1995 / month - 3
127.0.0.1:6379> hset hh name jinmin year 1995 month 3
(integer) 3

127.0.0.1:6379> hget hh name
"jinmin"
```

```
127.0.0.1:6379> hget hh year
"1995"

127.0.0.1:6379> hdel hh year
(integer) 1

127.0.0.1:6379> hlen hh
(integer) 2

127.0.0.1:6379> hgetAll hh
1) "name"
2) "jinmin"
3) "month"
4) "3"

127.0.0.1:6379> hkeys hh
1) "name"
2) "month"

127.0.0.1:6379> hvals hh
1) "jinmin"
2) "3"
```

- 기본 사용자 프로필을 해시로 나타낸다.
-

```
> HSET user:123 username martina firstName Martina lastName Elisa country GB
(integer) 4
> HGET user:123 username
"martina"
> HGETALL user:123
1) "username"
2) "martina"
3) "firstName"
4) "Martina"
5) "lastName"
6) "Elisa"
7) "country"
8) "GB"
```

- 777번 장비가 서버에 핑을 보내거나, 요청을 보내거나, 오류를 보낸 횟수에 대한 카운터를 저장한다.
-

```
> HINCRBY device:777:stats pings 1
(integer) 1
> HINCRBY device:777:stats pings 1
(integer) 2
> HINCRBY device:777:stats pings 1
(integer) 3
> HINCRBY device:777:stats errors 1
(integer) 1
> HINCRBY device:777:stats requests 1
(integer) 1
> HGET device:777:stats pings
"3"
> HMGET device:777:stats requests errors
1) "1"
2) "1"
```

Sorted sets

- 연관된 점수로 정렬된 고유한 문자열 멤버의 모음, 둘 이상의 문자열이 동일한 점수를 갖는 경우 문자열은 사전순으로 정렬된다.
  - 온라인 게임에서 순위를 표시할때 사용(카트라이더, 배틀그라운드 등수)
  - rate-limiter를 구현할때 사용, 과도한 API 요청을 막기 위해 슬라이딩 윈도우 속도 제한기를 만들 수 있다.
- 일반적으로 set은 정렬이 되어있지 않고 insert한 순서대로 들어간다. 그러나 sorted set은 set의 특성을 그대로 가지면서 추가적으로 저장된 member들의 순서도 관리한다.
- value는 중복이 불가능하고 score는 중복이 가능하다.
- 내부에서는 오름차순으로 내부 정렬
- score 값이 같으면 사전 순으로 정렬되어 저장된다.(이거땜에 게임 세션 끝나고 MVP 선정할때 점수 같아도 연출이 안나오는 버그가 나올 수 있다.)

command

- **SET**: ZADD

- **GET**: ZRANGE, ZRANGEBYSCORE, ZRANGEBYLEX, ZREVRANGE, ZREVRANGEBYSCORE, ZREVRANGEBYLEX, ZRANK, ZREVRANK, ZSCORE, ZCARD, ZCOUNT, ZLEXCOUNT, ZSCAN
- **POP**: ZPOPMIN, ZPOPMAX
- **REM**: ZREM, ZREMRANGEBYRANK, ZREMRANGEBYSCORE, ZREMRANGEBYLEX
- **INCR**: ZINCRBY
- **집합연산**: ZUNIONSTORE, ZINTERSTORE
- **Enterprise**: ZISMEMBER, ZLS, ZRM, SLEN, SADDS (subquery)

Commands	Version	Syntax	Description
ZADD	1.2.0	key score member [score member ...]	집합에 score와 member를 추가
ZCARD	1.2.0	key	집합에 속한 member의 갯수를 조회
ZINCRBY	1.2.0	key increment member	지정한 만큼 score 증가, 감소
ZRANGE	1.2.0	key start stop [withscores]	index로 범위를 지정해서 조회
ZRANGEBYSCORE	1.2.0	key min max [withscores] [limit offset count]	score로 범위를 지정해서 조회
ZREM	1.2.0	key member [member ...]	집합에서 member를 삭제
ZREMRANGEBYSCORE	1.2.0	key min max	score로 범위를 지정해서 member를 삭제
ZREVRANGE	1.2.0	key start stop [withscores]	index로 범위를 지정해서 큰 것부터 조회
ZSCORE	1.2.0	key member	member를 지정해서 score를 조회
ZCOUNT	2.0.0	key min max	score로 범위를 지정해서 갯수 조

			회
ZRANK	2.0.0	key member	member를 지정해서 rank(index)를 조회
ZREVRANK	2.0.0	key member	member를 지정해서 reverse rank(index)를 조회
ZREMRANGEBYRANK	2.0.0	key start stop	index로 범위를 지정해서 member를 삭제
ZUNIONSTORE	2.0.0	dest_key numkeys src_key [src_key ...][WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]	합집합을 구해서 새로운 집합에 저장
ZINTERSTORE	2.0.0	dest_key numkeys src_key [src_key ...][WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]	교집합을 구해서 새로운 집합에 저장
ZREVRANGEBYSCORE	2.2.0	key max min [withscores] [limit offset count]	score로 범위를 지정해서 큰 것부터 조회
ZSCAN	2.8.0	key cursor [MATCH pattern][COUNT count]	score, member를 일정 단위 갯수만큼씩 조회
ZRANGEBYLEX	2.8.9	key min max [limit offset count]	member로 범위를 지정해서 조회
ZLEXCOUNT	2.8.9	key min max	member로 범위를 지정해서 갯수 조회
ZREMRANGEBYLEX	2.8.9	key min max	member로 범위를 지정해서 member를 삭제
ZREVRANGEBYLEX	2.8.9	key max min [limit offset count]	member로 범위를 지정해서 큰 것부터 조회
ZPOPMIN	5.0.0	key	작은 값부터 꺼내온다

ZPOPMAX	5.0.0	key	큰 값부터 꺼내온다
BZPOPMIN	5.0.0	key	데이터가 들어오면 작은 값부터 꺼내온다
BZPOPMAX	5.0.0	key	데이터가 들어오면 큰 값부터 꺼내온다
ZMSCORE	6.2.0	member [member ...]	member의 score를 리턴 - 여러 개 가능
ZRANDMEMBER	6.2.0	key	임의(random)의 멤버를 조회
ZRANGESTORE	6.2.0	dst src start stop	조회해서 다른 키에 저장
ZUNION	6.2.0	numkeys key [key ...]	합집합을 구함
ZINTER	6.2.0	numkeys key [key ...]	교집합을 구함
ZDIFF	6.2.0	numkeys key [key ...]	차집합을 구함
ZDIFFSTORE	6.2.0	destination numkeys key [key ...]	차집합을 구해서 새로운 집합에 저장
ZISMEMBER	Ent 6.0.0	key member	집합에 member가 존재하는지 확인
ZLS	Ent 6.0.0	key pattern	패턴(pattern)으로 값(value) 조회
ZRM	Ent 6.0.0	key pattern	패턴(pattern)으로 값(value) 삭제
ZLEN	Ent 7.0.0	key	키에 속한 멤버 개수를 리턴
ZADDS	Ent 7.2.5	key (subquery)	서브쿼리로 데이터 추가

example

```
# ZADD : key에 score-member를 추가
127.0.0.1:6379> zadd fruit 2 apple
(integer) 1

127.0.0.1:6379> zadd fruit 10 banana
(integer) 1

# 복수개의 score-member를 추가할 수 있음
127.0.0.1:6379> zadd fruit 8 melon 4 orange 6 watermelon
(integer) 3

# 이미 추가 된 member를 add 시 score가 업데이트
127.0.0.1:6379> zadd fruit 15 apple
(integer) 0

# ZSCORE : member에 해당하는 score 값 리턴
127.0.0.1:6379> zscore fruit apple
"15"

# ZRANK : member에 해당하는 rank(순위) 값 리턴
127.0.0.1:6379> zrank fruit melon
(integer) 2

# ZRANGE : key에 해당하는 start - stop 내가 출력하고 싶은 요소를 추출
127.0.0.1:6379> zrange fruit 0 -1
1) "orange"
2) "watermelon"
3) "melon"
4) "banana"
5) "apple"
```

- 대부분의 정렬된 집합 연산은 O(log(n))이다.

Sorted Set 데이터 구조

- 스킵 리스트(SKIP LIST): Sorted Set의 메인 데이터 구조인 스킵 리스트를 알면,
  - 이제 우리는 눈을 감고도 ZADD가 어떻게 동작하는지,
  - ZRANGE는 수백만 건의 데이터에서 어떻게 그렇게 빨리 조회할 수 있는지 알 수 있다.

- 짚 리스트(ZIP LIST)
  - 짚 리스트의 탄생 배경, 데이터 구조와 기본 동작,
  - Sorted Set에서 사용될 때 성능과 메모리를 얼마나 절약하는지를 알게 된다.

Streams

- 로그를 저장하기 가장 좋은 자료구조
- append-only이고 중간에 데이터가 바뀌지 않는다.
- 읽어올때 id 값 기반으로 시간 범위로 검색한다.

command

Commands	Version	Syntax	Description
XADD	5.0.0	key ID field string [field string ...]	
XLEN	5.0.0	key	
XRANGE	5.0.0	key start end [COUNT count]	
XREVRANGE	5.0.0	key end start [COUNT count]	
XREAD	5.0.0	[COUNT count] [BLOCK milliseconds] STREAMS key [key ...] ID [ID ...]	
XDEL	5.0.0	key ID [ID ...]	
XTRIM	5.0.0	key MAXLEN [~] count	
XGROUP	5.0.0	[CREATE key group id-or-$] [DESTROY key group] [DELCONSUMER key group consumer] [SETID key group id-or-$]	
XREADGROUP	5.0.0	GROUP group consumer [COUNT count] [BLOCK milliseconds] STREAMS key [key ...] ID [ID ...]	
XACK	5.0.0	key group ID [ID ...]	
XPENDING	5.0.0	key group [start end count] [consumer]	
XCLAIM	5.0.0	key group consumer min-idle-time ID [ID ...] [IDLE ms] [TIME ms-unix-	

		time] [RETRYCOUNT count] [FORCE] [JUSTID]	
XAUTOCLAIM	6.2.0	key group consumer min-idle-time start [COUNT count] [JUSTID]	
XINFO	5.0.0	[CONSUMERS key group] [GROUPS key] [STREAM key] [HELP]	

example

- 스트림에 온도 판독값 추가
-

```
> XADD temperatures:us-ny:10007 * temp_f 87.2 pressure 29.69 humidity 46
"1658354918398-0"
> XADD temperatures:us-ny:10007 * temp_f 83.1 pressure 29.21 humidity 46.5
"1658354934941-0"
> XADD temperatures:us-ny:10007 * temp_f 81.9 pressure 28.37 humidity 43.7
"1658354957524-0"
```

XADD 명령문은 다음과 같다.

```
XADD key ID field value [field2 value2 ...]
```

여기서 ID를 *로 준것을 볼 수 있는데 ID를 *로 입력하면 결과로 ID를 리턴하게 된다. ID는 두 개의 숫자로 구성되는데, `<millisecondsTime>-<sequenceNumber>`

앞은 millisecond 단위의 timestamp 숫자이고, 뒤는 같은 밀리 초 내에 또 데이터가 들어오면 중복을 방지하기 위해서 숫자가 1, 2, 3 와 같이 하나씩 증가한다. . 이 sequenceNumber는 8바이트 정수이므로 실제적으로 동일한 밀리 초 내에 생성 될 수 있는 엔트리 수에는 제한이 없다. D를 *로 입력하는 것은 레디스가 있는 서버 시간으로 ID를 생성하는 것이다.

ID를 직접 입력 할 수 도 있다.

```
XADD sensor-1234 1538319053569-1 temperature 98.8
1538319053569-1
```

ID는 이전에 입력된 것보다 항상 커야한다. 작거나 같으면 에러가 발생한다.

- 데이터 길이(항목수) 조회하기
-

```
XLEN sensor-1234
(integer) 2
```

정확히는 ID의 개수를 조회하는 것이다.

- **데이터 조회하기**
-

```
XRANGE sensor-1234 - +
1) 1) 1538319053569-0
    2) 1) "temperature"
        2) "98.7"
2) 1) 1538319053569-1
    2) 1) "temperature"
        2) "98.8"
```

XRANGE 명령문(syntax)은 다음과 같다.

`XRANGE key start end [COUNT count]`

key=sensor-1234, start=-, end=+ 이다. 또한 Count를 사용해 조회할 데이터 개수를 지정할 수 있고 Start와 end에 특정 ID를 지정할 수 있다.

`XRANGE key 1538319053569 1538319053569`

최근 데이터부터 조회하고 싶으면 **XREVRANGE** 명령을 사용한다.

`XREVRANGE key + -`

- **데이터 읽어오기**
-

```
XREAD count 1 STREAMS sensor-1234 1538322045065-0
1) 1) "sensor-1234"
    2) 1) 1) 1538322045065-1
        2) 1) "temperature"
            2) "98.8"
```

XREAD 명령문은 다음과 같다.

`XREAD [COUNT count] [BLOCK milliseconds] STREAMS key [key ...] ID [ID ...]`

Count는 읽어올 데이터 개수를 지정한다. Count를 지정하지 않으면 모든 데이터를 읽어온다. ID는 지정한 ID 다음의 데이터를 읽어온다. 처음 데이터를 읽으려면 ID로 0을 지정한다. Block은 새 데이터가 들어오기 기다렸다 들어오면 읽어온다. 이 때는 ID로 특별히 $를 사용한다.

`XREAD block 5000 STREAMS sensor-1234 $`

- **데이터 삭제하기**
-

```
XDEL sensor-1234 1538322045065-0
(integer) 1
```

명령문은 다음과 같다.

`XDEL key ID`

- sensor-1234에 데이터가 100가 있다면 오래된 순으로 90개를 지우고 최신 데이터 10개를 남깁니다.

```
XTRIM sensor-1234 maxlen 10
```

XTRIM의 명령문은 다음과 같다.

`XTRIM key MAXLEN [~] count`

특별한 옵션인 ~ 는 약(about)이다. Sensor-1234에 100만개의 데이터가 있다면 999,990개를 지우는데 시간이 걸릴 것이다. 그러면 데이터를 지우는 동안 데이터를 추가(XADD)하거나 처리(XREAD) 될 수 없다. 대량 데이터를 신속히 처리해야하는 스트림에서는 이런 처리 지연이 발생하지 않도록해야 한다. 그래서 짧은 시간에 처리할 수 있을 정도의 데이터를 삭제하는 기능이다.

Stream Consumer groups

- 스트림에서는 소비자(consumer)를 지정해서 데이터를 읽을 수 있고, 그 소비자가 데이터를 제대로 처리했는지 확인하는 방법을 제공하며, 만약 제대로 처리하지 못했다면 다른 소비자에게 할당해서 처리하도록 하는 방법을 제공한다.

- **소비자그룹 만들기: XGROUP**

`XGROUP CREATE sensor-1234 ConsumerGroup $`

테스트 데이터 넣기

```
XADD sensor-1234 * temperature 100
XADD sensor-1234 * temperature 101
XADD sensor-1234 * temperature 102
XADD sensor-1234 * temperature 103
XADD sensor-1234 * temperature 104
XADD sensor-1234 * temperature 105
```

- **소비자그룹을 이용한 데이터 읽기: XREADGROUP**
-

```
XREADGROUP GROUP ConsumerGroup Consumer-A count 1 STREAMS sensor-1234 >
1) 1) "sensor-1234"
    2) 1) 1) 1538568726521-0
        2) 1) "temperature"
            2) "100"
XREADGROUP GROUP ConsumerGroup Consumer-A COUNT 1 STREAMS sensor-1234 >
1) 1) "sensor-1234"
    2) 1) 1) 1538568728545-0
        2) 1) "temperature"
            2) "101"
```

- **처리 여부 확인하기: XPENDING**
-

```
XPENDING sensor-1234 ConsumerGroup
1) (integer) 2
2) 1538568726521-0
3) 1538568728545-0
4) 1) 1) "Consumer-A"
    2) "2"
```

- **처리 확정하기: XACK**

```
XACK sensor-1234 ConsumerGroup 1538568726521-0
(integer) 1
```

처리 여부 다시 확인하기

```
XPENDING sensor-1234 ConsumerGroup
1) (integer) 1
2) 1538568728545-0
3) 1538568728545-0
4) 1) 1) "Consumer-A"
    2) "1"
```

- **팬딩 시간 알아보기: XINFO CONSUMERS**
-

```
XINFO CONSUMERS sensor-1234 ConsumerGroup
1) 1) name
    2) "Consumer-A"
    3) pending
    4) (integer) 1
    5) idle
    6) (integer) 31963
```

- **팬딩된 데이터를 다른 소비자에게 할당하고 처리하기: XCLAIM**
-

```
XCLAIM sensor-1234 ConsumerGroup Consumer-B 30000 1538568728545-0
1) 1) 1538568728545-0
    2) 1) "temperature"
        2) "101"
```

이 데이터 한 건에 대해서만 다른 소비자가 처리하도록 한 것이다.

XCLAIM으로 처리한 후에 XACK를 보내야 한다.

명령문은 다음과 같다.

`XCLAIM key group consumer min-idle-time ID`

min-idle-time은 밀리초를 지정한다.

이제 계속 데이터를 읽어서 처리한다.

```
XREADGROUP GROUP ConsumerGroup Consumer-B COUNT 1 STREAMS sensor-1234 >
1) 1) "sensor-1234"
    2) 1) 1) 1538568731409-0
        2) 1) "temperature"
            2) "102"
```

- **스트림 하나에 여러 소비자 할당하기**

하나의 스트림에 처리해야 할 데이터가 많고, 처리 시간이 오래 걸린다면, 아래에서 처럼 소비자(consumer)를 여러 개 지정할 수 있다.

예를 들어, 1초에 3개의 데이터(Data-1,2,3)가 들어오고 애플리케이션에서 하나의 데이터를 처리하는데 1초가 걸린다면 3개의 소비자(Consumer-A,B,C)를 지정해서 각각 다른 애플리케이션에서 처리하도록 한다.

```
Data-1 -> Consumer-A
Data-2 -> Consumer-B
Data-3 -> Consumer-C
Data-4 -> Consumer-A
Data-5 -> Consumer-B
Data-6 -> Consumer-C
```

geospatial

- 두 지점/도시의 경도(세로선/longitude)와 위도(가로선/latitude)를 입력해서 두 지점의 거리를 구한다.
- Geo는 지구(Earth)가 완전한 구(球/sphere)라고 가정합니다. 따라서 최대의 경우 0.5% 정도 오차가 발생할 수 있다.
- Geo는 Sorted Set Data Structure를 사용한다. 따라서 몇 가지 명령은 Sorted Set의 명령을 그대로 사용할 수 있다.
    - 범위 조회: ZRANGE key 0 -1, 삭제: ZREM key member, 개수 조회: ZCARD key

command

- 경도/위도 입력: GEOADD
- 경도/위도 조회: GEOPOS
- 거리 조회: GEODIST
- 주변 지점 조회: GEORADIUSBYMEMBER, GEORADIUS
- 해시값 조회: GEOHASH
- 범위 조회: ZRANGE
- 삭제 조회: ZREM
- 개수 조회: ZCARD

Commands	Version	Syntax	Description
GEOADD	3.2.0	key longitude latitude member [longitude latitude member ...]	
GEOPOS	3.2.0	key member [member ...]	
GEODIST	3.2.0	key member1 member2 [unit]	
GEORADIUSBYMEMBER	3.2.0	key member radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count] [ASC|DESC] [STORE key] [STOREDIST key]	
GEORADIUS	3.2.0	key longitude latitude radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count] [ASC|DESC] [STORE key] [STOREDIST key]	
GEOHASH	3.2.0	key member [member ...]	

example

현재 위치에서 가장 가까운 모든 전기 자동차 충전소를 찾을 수 있는 모바일 앱을 구축한다고 가정할때,

여러 위치를 추가한다.

```
> GEOADD locations:ca -122.27652 37.805186 station:1
(integer) 1
> GEOADD locations:ca -122.2674626 37.8062344 station:2
(integer) 1
> GEOADD locations:ca -122.2469854 37.8104049 station:3
(integer) 1
```

지정된 위치에서 반경 5Km 이내의 모든 위치를 찾고 각 위치까지의 거리를 반환한다.

```
> GEOSEARCH locations:ca FROMLONLAT -122.2612767 37.7936847 BYRADIUS 5 km WITHDIST
1) 1) "station:1"
   2) "1.8523"
2) 1) "station:2"
   2) "1.4979"
3) 1) "station:3"
   2) "2.2441"
```

HyperLogLog

- 굉장히 많은 양의 데이터를 dump 할 때 사용한다.
- 중복되지 않는 대용량 데이터를 count 할 때 주로 사용한다(오차 범위 0.81%)
    - 오차는 1.04/SQRT(m)이다. m은 레지스터(register) 개수로 Redis에서는 16384개를 사용하므로 오차는 0.81%이다.
    - 웹 사이트 방문 ip 개수 카운팅, 하루 종일 크롤링 한 url 개수 몇개 인지, 검색 엔진에서 검색 한 단어 몇개 인지

- set과 비슷하지만 저장되는 용량은 매우 작다 (저장 되는 모든 값이 12kb 고정)
- 하지만 저장된 데이터는 다시 확인할 수 없다. (데이터 보호에 적절)

command

Commands	Version	Syntax	Description
PFADD	2.8.9	key ele [ele ...]	원소(element) 추가
PFCOUNT	2.8.9	key [key ...]	원소 개수 조회
PFMERGE	2.8.9	destkey sourcekey [sourcekey ...]	집합 머지(Merge)

```
> PFADD crawled:20171124 "http://www.google.com/"
(integer) 1

> PFADD crawled:20171124 "http://www.redis.com/"
(integer) 1

> PFADD crawled:20171124 "http://www.redis.io/"
(integer) 1

> PFADD crawled:20171125 "http://www.redisearch.io/"
(integer) 1

> PFADD crawled:20171125 "http://www.redis.io/"
(integer) 1

> PFCOUNT crawled:20171124
(integer) 3

> PFMERGE crawled:20171124-25 crawled:20171124 crawled:20171125
OK

> PFCOUNT crawled:20171124-25
(integer) 4
```

Bitmaps

- 문자열을 비트 벡터처럼 처리할 수 있는 문자열 데이터 유형의 확장, 하나 이상의 문자열에 대해 비트 연산을 수행할 수도 있다.
- bit 단위 연산이 가능하다.
- String이 512MB 저장 할 수 있듯이 2^32 bit까지 사용 가능하다.
- 저장할 때, 저장 공간 절약에 큰 장점이 있다.

command

Commands	Version	Syntax	Description
GETBIT	2.2.0	key offset	bit 값 조회
SETBIT	2.2.0	key offset value	bit 값 조정
BITCOUNT	2.6.0	key [start end]	1인 bit 수를 센다
BITOP	2.6.0	key operation destkey key [key ...]	bit 연산(AND, OR, XOR, NOT) 실행
BITPOS	2.8.7	key bit [start [end]]	지정한 bit의 위치를 구한다
BITFIELD	3.2.0	key [GET type offset] [SET type offset value]	Perform arbitrary bitfield integer operations on strings

example

```
# setbit <key> <offset> <value>
# key: 해당 비트맵을 칭할 값
# offset: 0 보다 큰 정수의 값
# value: 0 또는 1의 비트 값

> setbit 20220410 4885 1

> getbit 20220410 4885
```

```
> bitcount 20220410 # 범위 내의 1로 설정된 bit의 개수를 반환
```

Bitfields

- 임의 비트 길이의 정수 값을 설정해서 증분하거나 가져올 수 있다. 예를들어 부호 없는 1비트 정수에서 부호 있는 63비트 정수에 까지 무엇이든 연산 할 수 있다.
- 바이너리 인코딩 된 문자열로 저장된다.