Vishnu Bharathi Follow
Loves Learning things.
Sep 16 · 8 min read

Redis Adventures

Redis is not just a key value store, it's more than that. I always had this thought and wanted to learn more about it. At this time, I badly want to get this and this done. But I am not going to do them right away. Instead, I am going to take time to read more about Redis.

I will try to document my journey as I go and share the things that fascinated me along the way.

Getting Started

If you are not familiar with Redis, here is a quote from the its homepage,

> *Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker…….*

I will use docker to run Redis, as I prefer it over downloading and installing. Unfortunately, there is no trace of running Redis with docker on the official site, so I opened an issue on github requesting it. (Ah, the response arrived)

Redis site recommends following this twitter account for updates. So, I just did.

With a single docker command, I have it up and running in my machine.

```
$ docker run --name some-redis -d redis
```

Basics

Before doing anything on my local machine, I am going to head on to http://try.redis.io/ to try out the classic way of learning about Redis.

Some basic commands, that I learnt are

```
# SET KEY VALUE
#==============
SET key "value" # key is case sensitive
SET intKey 10
SETNX intKey 11 # SET NOT EXISTS
```

```
# GET KEY
#========
GET key
GET intKey
```

```
# Int operations
#================
INCR intKey  # Increment and why is it even there?
DECR intKey  # Decrement
```

```
# SET with EXPIRE and TTL (Time to Live)
#=======================================
SET clientToken "super-secret-token"
EXPIRE clientToken 120 # will be stored only for 120
seconds
TTL clientToken
# => 118 means 2 seconds elapsed.
# => -1 means no expiration is set.
# => -2 means expired already.
SET clientToken "refreshed-token"
TTL clientToken # yields -1, as reset.


# List operations
#================
RPUSH game scooby # right push
LPUSH game dexter # left push
LRANGE game 0 -1  # range over all items
LRANGE game 0 0   # range from startIndex to endIndex
RPOP game         # right pop
LPOP game         # left pop
LLEN game         # length of the list
```

```
# Set (Data structure) operations
#================================
# Recommend reading => Wikipedia
SADD powers "flight"
SREM powers "flight"
SMEMBERS powers
SISMEMBER powers flight
SUNION powers weapons
```

```
# Sorted Set
#===========
# Good to follow this. TODO: Dig deep to figure out
# relation and boundaries between sets, sorted sets and
lists
```

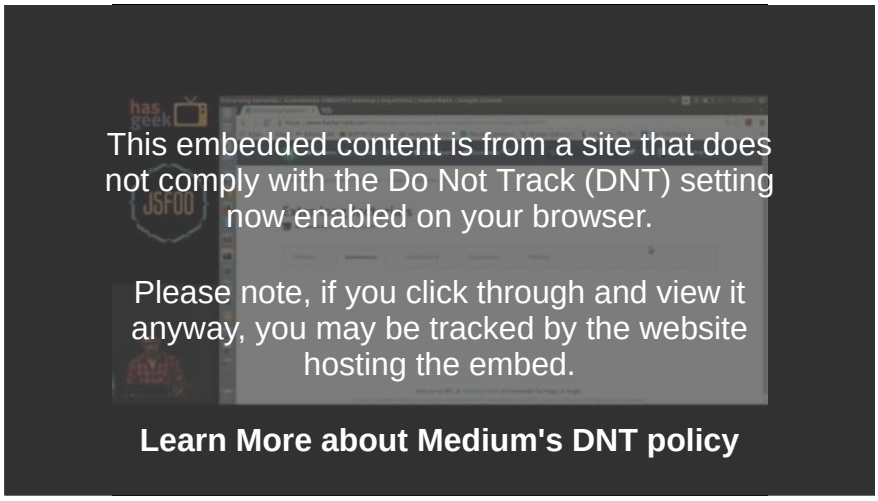
```
# Hashes
#=======
# perfect to hold world objects
HSET user:1001 name "katniss"
HSET user:1001 age 26
HSET user:1001 movie "The Hunger Games"
HGETALL user:1001 # get all props and values
HGET user:1001 # get the value of movie property
HMSET user:1002 name "Neo" age 52 movie "The Matrix" #
multiple set
HDEL user:1002 movie
HINCRBY user:1002 age 1 # atomic increment
```

That's good! I got to know some basic data structures in Redis. Reading https://redis.io/topics/data-types-intro for more information.

It seems like lists in Redis are implemented using Linked Lists. So, it takes constant time for insertion and deletion from both ends.

Bitmaps

Bitmaps are awesome. I first heard about it in a JavaScript conference, where the speaker talks about using bits instead of JSON objects and achieves high memory throughput.



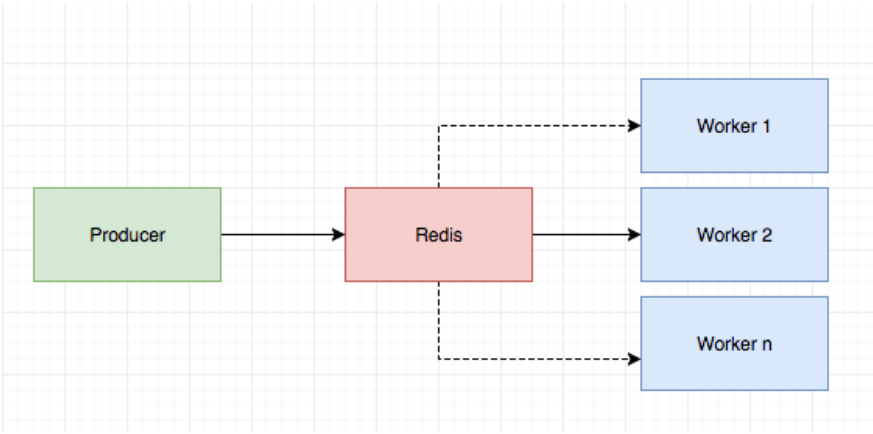
I remembered that optimisation from then, but I didn't know of the name "Bitmap". One fine day, my friend introduced me to the term and discussed how he used it in various use cases.

If you are wondering what bitmaps are, I recommend to take read about it in Wikipedia. In simple terms, it's a collection of bits. `1111000` is a bitmap, that will take lesser memory to represent the attendance of a student in a week than, having separate entries (structs/db entries) for each day. 1's represent that student was present on the day of that week and 0's represent absence.

And you can use Bitmaps in Redis.

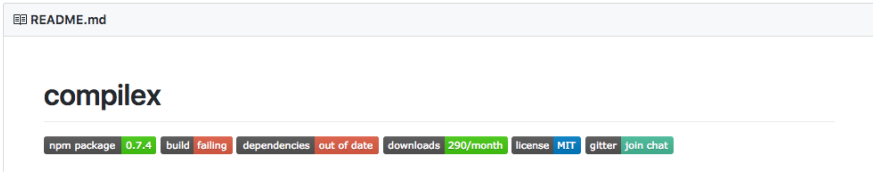
Producer Consumer Pattern

I use this pattern in lot of places. At my work, almost all microservices follow this pattern. Here is a small pictorial representation of this pattern



Producer Consumer Pattern (via draw.io). Dotted line mean listening. Solid line means listening and consuming.

I will demonstrate how this pattern is being used on the project I am working on. It's this project called Badgeit, which is a website that guesses the badges that can be added on a source code repository's README.md file (Usually present to describe what the software does in a code repository).



Example of badges in README.md file of project

We run an API server which has an end point `POST /badges` that accepts a git repository URL as input. This endpoint clones the git repository and reads through the source code files and will send back badge suggestions. Git clone is a costly process and we should not be doing it

for each and every request that arrives. In fact, I run only one server machine which has around 990MB RAM space on AWS. If we clone naively for each and every request we might soon run out of memory. Here are some conditions on how I want cloning to happen.

1. At any given point of time, I want to have finite number of clones to happen on the server. (This finite number is the number of workers)
2. First come first serve basis.
3. Equally distribute the task among the available workers.

So, I created another stateless service called *worker* and decoupled the git repo cloning logic from *api*. *api* service is the producer and worker service is the consumer.

I am already using RabbitMq, which is pretty heavy weight and serves only as a Queue for producer consumer pattern. But I need a piece of software that could act as queue and also as a cache. Hmm. Sounds very familiar. Redis to rescue!

Remove RabbitMq and introduce Redis · Issue #22 · scriptnull/badgeit

RabbitMq is a pure queue. But redis can act as cache + queue. And I suspect redis to be have much lower memory...

github.com

To implement this in Redis, I will make my producer to send LPUSH redis command for adding the payload to a redis list. In my consumer, I will use BRPOP to do a blocking pop operation.

Let me give it a try. Ah! I got it right.

Both the producer and worker can use the following snippet to establish a connection.

```
// Uses github.com/garyburd/redigo/redis Go Library

log.Println("Initializing Redis Message Queue")
conn, err := redis.Dial("tcp", fmt.Sprintf("%s:%s", os.
if err != nil {
        log.Fatalln("Failed to initialize redis message
}
```

Establish Redis connection

Fun Fact: you can use `INFO` redis command to check how many active clients are connected.

```
// Uses github.com/garyburd/redigo/redis Go Library

// push to queue
_, err := conn.Do("LPUSH", "badge:worker", []byte("som

// Blocking pop from queue
payload, err := redis.Strings(redisConn.Do("BRPOP", "b
```

Queue Operations with Redis

MULTI EXEC

If you have used SQL databases, you should be familiar of this fancy term “transactions”. oh yeah! Redis has these two commands for the exact same reason.

In the publisher consumer pattern, we just pushed the payload to the queue. What if I want to do rate limiting ? That is neglect the future requests for adding a repository in queue if it is already in the queue or being consumed by a worker.

Is it ok to range over the queue ? Definitely not. A queue is a data structure in which data access should be at one of the ends. Ranging over queues is not going to do us any good, because it is going to get all the elements in the queue and load it in the client’s memory (in my case a api server written in Go).

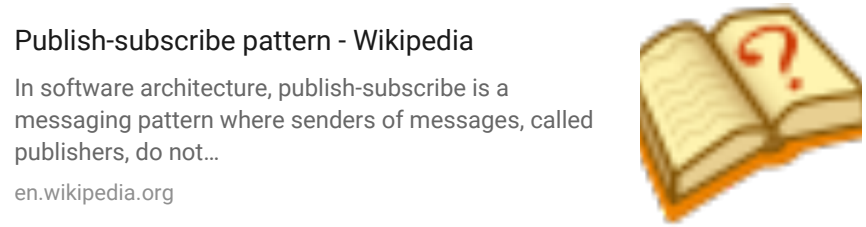
So, I went on to maintain a SET, which is an efficient for doing operations like ISMEMBER. By maintaining a some keys (which denote each payload in the queue) in a set, we can easily determine if a request to queue a repository should be processed.

But the thing here is the set should be in sync with our queue. Before we choose to queue, it MUST be added to the set. If the add to set operation fails, then we MUST not add it to the queue. So, we could just wrap both the insert operations inside a MULTI EXEC block. That way, we don't end up in stale states.

```
remote := "https://github.com/scriptnull/scribble"
jsonPayload := "{}" // assume some dummy data

redisConn.Send("MULTI")
// add to a set
redisConn.Send("SADD", "badgeit:queuedRemotes", remote
// push to queue
redisConn.Send("LPUSH", "badge:worker", []byte(jsonPay
```

Pub/Sub Pattern



Redis has SUBSCRIBE, PUBLISH, UNSUBSCRIBE commands and supports the publish subscribe pattern. I have given it a try only on the command line. But, I can clearly imagine the scope of this feature.

Lets say we have a billing application in an ice cream shop (why ice cream shop? because I love them). After someone purchases an ice cream, the shop collects the phone number and email of the customer. They collect it to send a discount coupon to the customer's mobile as SMS and an Email (Don't ask me why do we have to do something like this. These days, all the people do this stuff). The ice cream shop's server runs two services, one sends SMS to the mobile and other one to send the email. Using a queue here is an anti pattern, because whenever a message arrives in the queue, either one of the service consumes it and it becomes unavailable for the other service. So, it might be wise to use a redis SUBSCRIBE to listen for a particular event and do some action.

This particular approach is very useful, when the number of subscribers increase. Example: let's say that the ice cream shop decides to keep track

of some bounty points for each purchase they do, they might have to just introduce one more service that subscribes to our customer sale event.

Also note that PSUBSCRIBE, PUNSUBSCRIBE are also present, which is a useful variant for pattern matching operations.

```
SUBSCRIBE user.created # listens for user.created event
SUBSCRIBE user.updated

PSUBSCRIBE user.*      # listens for all events starting
with user.
PUNSUBSCRIBE user.*
```

Lua Scripting

I am surprised to know that Redis supports Lua Scripts. Lua is a programming language, which I have tried using previously for developing games. Some of the things in my mind are

"How exactly is Lua scripting useful in Redis? Why did they built it into Redis? Why Lua? why not some other programming language?"

I am still not using it in any application, but when time arrives I will probably understand more about this.

More stuff

I probably just played with swings and circles in Redis, I am sure that there are roller coasters and huge water slides. I hope to ride them in future. But for now, I had fun with what I experienced.

. . .

Thanks for reading. I quote verses from my favourite Tamil literature "Tirukkural" at the end of my blog posts.

> *"துன்புறூஉம் துவ்வாமை இல்லாகும் யார்மாட்டும்*
>
> *இன்புறூஉம் இன்சொ லவர்க்கு."*
>
> *— திருக்குறள்*

Translated meaning (in my words): Poverty will not exist for people who speak pleasant words to everyone.