Redis IN ACTION

Josiah L. Carlson

FOREWORD BY Salvatore Sanfilippo

SAMPLE CHAPTER

MANNING

Redis in Action

by Josiah L. Carlson

Chapter 8

brief contents

8

Building a simple social network

This chapter covers

- Users and statuses
- Home timeline
- Followers/following lists
- Posting or deleting a status update
- Streaming API

In this chapter, we'll cover the data structures and concepts necessary to build a system that offers almost all of the back-end-level functionality of Twitter. This chapter isn't intended to allow you to build a site that scales to the extent of Twitter, but the methods that we cover should give you a much better understanding of how social networking sites can be built from simple structures and data.

We'll begin this chapter by talking about user and status objects, which are the basis of almost all of the information in our application. From there, we'll discuss the home timeline and followers/following lists, which are sequences of status messages or users. Continuing on, we'll work through posting status messages, following/unfollowing someone, and deleting posts, which involves manipulating those lists. Finally, we'll build out a fully functioning streaming API with web server to encourage users of the social network to use and play with the data.

In the last chapter, we spent much of our time building an ad-targeting engine that combined user-entered data (the ads and their prices) with click behavior data in order to optimize ad earnings. The ad-targeting engine was query-intensive, in that every request could cause a lot of computation. In this Twitter-like platform, we'll do our best to perform as little work as possible when someone is interested in viewing a page.

To get started, let's build the basic structures that will hold much of the data that our users are interested in.

8.1 Users and statuses

As users interact with Twitter, two types of objects hold the most important information: users and status messages. User objects hold basic identity information, as well as aggregate data about the number of followers, number of status messages posted, and more. The user objects are important because they're the starting point for every other kind of data that's available or interesting. Status messages are also important because they're how individuals express themselves and interact with each other, and are the true content of social networks.

In this section, we'll talk about what data will be stored in the user and status message objects and how we'll store them. We'll also look at a function to create a new user.

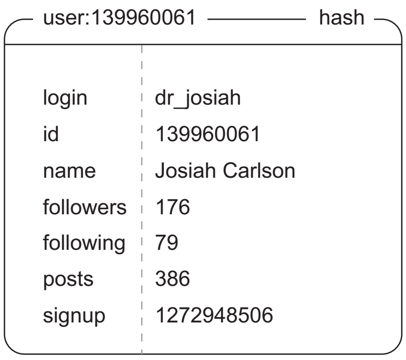
Our first step is to define and create the structure for a user.

8.1.1 User information

In a variety of online services and social networks, user objects can be the basic building blocks from which everything else is derived. Our Twitter work-alike is no different.

We'll store user information inside of Redis as a HASH, similar to how we stored articles in chapter 1. Data that we'll store includes the username of the user, how many followers they have, how many people they're following, how many status messages they've posted, their sign-up date, and any other meta-information we decide to store down the line. A sample HASH that includes this information for a user with the username of dr_josiah (my Twitter username) is shown in figure 8.1.

From this figure, you can see that I have a modest number of followers, along with other information. When a new user signs up, we only need to create an object with the following, followers, and post count set to zero, a new timestamp for the sign-up time, and the relevant username. The function to perform this initial creation is shown next.



user:139960061	hash
login	dr_josiah
id	139960061
name	Josiah Carlson
followers	176
following	79
posts	386
signup	1272948506

Figure 8.1 Example user information stored in a HASH

Listing 8.1 How to create a new user profile `HASH`

```
def create_user(conn, login, name):
    llogin = login.lower()
    lock = acquire_lock_with_timeout(conn, 'user:' + llogin, 1)
    if not lock:
        return None

    if conn.hget('users:', llogin):
        return None

    id = conn.incr('user:id:')
    pipeline = conn.pipeline(True)
    pipeline.hset('users:', llogin, id)
    pipeline.hmset('user:%s'%id, {
        'login': login,
        'id': id,
        'name': name,
        'followers': 0,
        'following': 0,
        'posts': 0,
        'signup': time.time(),
    })
    pipeline.execute()
    release_lock(conn, 'user:' + llogin, lock)
    return id
```

Try to acquire the lock for the lowercased version of the login name. This function is defined in chapter 6.

If we couldn't get the lock, then someone else already has the same login name.

We also store a HASH of lowercased login names to user IDs, so if there's already a login name that maps to an ID, we know and won't give it to a second person.

Each user is given a unique ID, generated by incrementing a counter.

Add the lowercased login name to the HASH that maps from login names to user IDs.

Add the user information to the user's HASH.

Release the lock over the login name.

Return the ID of the user.

In our function, we perform the expected setting of the initial user information in the user's `HASH`, but we also acquire a lock around the user's login name. This lock is necessary: it guarantees that we won't have two requests trying to create a user with the same login at the same time. After locking, we verify that the login name hasn't been taken by another user. If the name hasn't been taken, we generate a new unique ID for the user, add the login name to the mapping of login names to user IDs, and then create the user's `HASH`.

> **SENSITIVE USER INFORMATION** Because the user `HASH` will be fetched countless times for rendering a template, or for returning directly as a response to an API request, we don't store sensitive user information in this `HASH`. For now, we'll assume that hashed passwords, email addresses, and more are stored at other keys, or in a different database entirely.

We've finished creating the user and setting all of the necessary meta-information about them. From here, the next step in building our Twitter work-alike is the status message itself.

8.1.2 Status messages

As we mentioned earlier, whereas user profiles store information about an individual, the ideas that people are trying to express are stored in status messages. As was the case with user information, we'll store status message information inside a `HASH`.

In addition to the message itself, we'll store when the status message was posted, the user ID and login of the user who posted it (so that if we have a status object, we don't need to fetch the user object of the poster to discover their login name), and any additional information that should be stored about the status message. Figure 8.2 shows an example status message.

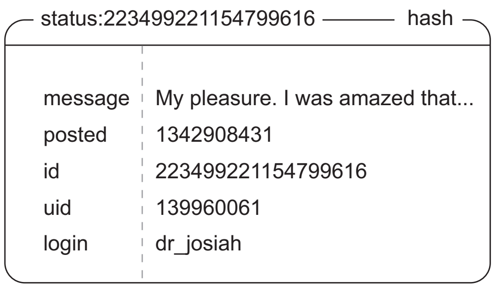


Figure 8.2 Example status message stored in a `HASH`

And that's everything necessary for a basic status message. The code to create such a status message can be seen in the next listing.

Listing 8.2 How to create a status message `HASH`

```
def create_status(conn, uid, message, **data):
    pipeline = conn.pipeline(True)
    pipeline.hget('user:%s'%uid, 'login')
    pipeline.incr('status:id:')
    login, id = pipeline.execute()

    if not login:
        return None

    data.update({
        'message': message,
        'posted': time.time(),
        'id': id,
        'uid': uid,
        'login': login,
    })
    pipeline.hmset('status:%s'%id, data)
    pipeline.hincrby('user:%s'%uid, 'posts')
    pipeline.execute()
    return id
```

Get the user's login name from their user ID.

Create a new ID for the status message.

Verify that we have a proper user account before posting.

Prepare and set the data for the status message.

Record the fact that a status message has been posted.

Return the ID of the newly created status message.

There isn't anything surprising going on in the status creation function. The function fetches the login name of the user, gets a new ID for the status message, and then combines everything together and stores it as a `HASH`.

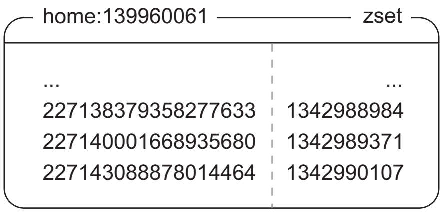
We'll talk about making the status message visible to followers in section 8.4, so sit tight for now, as we now examine the most commonly used view into lists of status messages: a user's *home timeline*.

8.2 Home timeline

When people visit Twitter after logging in, the first view that they see is what's referred to as their *home timeline*. This is a list of status messages that have been posted by the user and all of the people they're following. As the primary entry point to what users see, this data should be as easy to retrieve as possible.

In this section, we'll talk about the data to be stored in the home timeline and how to fetch information to display the home timeline quickly. We'll also talk about other important status message timelines.

As mentioned earlier in this chapter, we want to be able to fetch all of the data required for a given view as quickly as possible. For the home timeline, which will store the list of status messages that have been posted by the people that the current user is following, we'll use a ZSET to store status IDs as ZSET members, with the timestamp of when the message was posted being used as the score. Figure 8.3 shows an example home timeline.



...	...
227138379358277633	1342988984
227140001668935680	1342989371
227143088878014464	1342990107

Figure 8.3 When someone visits their home timeline on a site like Twitter, they see the most recently posted messages that people they follow have written. This information is stored as a ZSET of status ID/timestamp pairs. Timestamp information provides the sort order, and the status ID shows us what information to pull in a second step.

Because the home timeline is just referencing status messages—it doesn't contain the status messages themselves—our function to fetch the most recently posted status messages must also fetch the status message data. The next listing shows the code to fetch a page of messages from the home timeline.

Listing 8.3 A function to fetch a page of recent status messages from a timeline

```
def get_status_messages(conn, uid, timeline='home:', page=1, count=30):
    statuses = conn.zrevrange(
        '%s%s'%(timeline, uid), (page-1)*count, page*count-1)

    pipeline = conn.pipeline(True)
    for id in statuses:
        pipeline.hgetall('status:%s'%id)

    return filter(None, pipeline.execute())
```

We'll take an optional "timeline" argument, as well as page size and status message counts.

Fetch the most recent status IDs in the timeline.

Actually fetch the status messages themselves.

Filter will remove any "missing" status messages that had been previously deleted.

That function will fetch status messages in reverse chronological order from the provided timeline, which defaults to the home timeline.

A second important timeline is the timeline of posts that a user has posted. Where the home timeline includes posts from other people, the user's timeline will include only those posts from the user. These timelines can be seen when visiting a user's profile, and are the primary entry point for finding someone interesting. To fetch a page of statuses from a given user, we can call the same `get_messages()` function, but we'll pass `profile:` as the `timeline` argument in our call.

Now that a user can fetch the home timeline, we should discuss how to manage the list of users that someone is following, and the list of users that are following them.

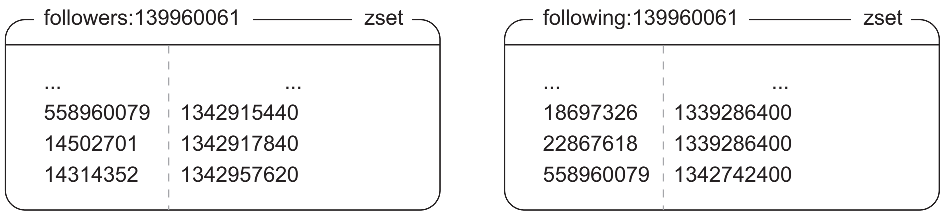


Figure 8.4 To know who's following a user, we store user ID/timestamp pairs in a ZSET. The user IDs are the people who're following that user, and the timestamp is when they started following the user. Similarly, the users that a user is following are stored as a ZSET of user ID/timestamp pairs of the user ID of the person being followed, and the timestamp of when the user followed them.

8.3 Followers/following lists

One of the primary services of a platform like Twitter is for users to share their thoughts, ideas, and dreams with others. Following someone means that you're interested in reading about what they're saying, with the hope that others will want to follow you.

In this section, we'll discuss how to manage the lists of users that each user follows, and the users that follow them. We'll also discuss what happens to a user's home timeline when they start or stop following someone.

When we looked at the home and profile timelines in the last section, we stored status IDs and timestamps in a ZSET. To keep a list of followers and a list of those people that a user is following, we'll also store user IDs and timestamps in ZSETs as well, with members being user IDs, and scores being the timestamp of when the user was followed. Figure 8.4 shows an example of the followers and those that a user is following.

As we start or stop following a user, there are following and followers ZSETs that need to be updated, as well as counts in the two user profile HASHes. After those ZSETs and HASHes have been updated, we then need to copy the newly followed user's status message IDs from their profile timeline into our home timeline. This is to ensure that after we've followed someone, we get to see their status messages immediately. The next listing shows the code for following someone.

Listing 8.4 Update the following user's home timeline

```
HOME_TIMELINE_SIZE = 1000
def follow_user(conn, uid, other_uid):
    fkey1 = 'following:%s'%uid
    fkey2 = 'followers:%s'%other_uid

    if conn.zscore(fkey1, other_uid):
        return None

    now = time.time()

    pipeline = conn.pipeline(True)
    pipeline.zadd(fkey1, other_uid, now)
    pipeline.zadd(fkey2, uid, now)
```

Cache the following and followers key names.

If the other_uid is already being followed, return.

Add the uids to the proper following and followers ZSETs.

```
    pipeline.zcard(fkey1)
    pipeline.zcard(fkey2)
    pipeline.zrevrange('profile:%s'%other_uid,
        0, HOME_TIMELINE_SIZE-1, withscores=True)
    following, followers, status_and_score = pipeline.execute()[-3:]

    pipeline.hset('user:%s'%uid, 'following', following)
    pipeline.hset('user:%s'%other_uid, 'followers', followers)
    if status_and_score:
        pipeline.zadd('home:%s'%uid, **dict(status_and_score))
    pipeline.zremrangebyrank('home:%s'%uid, 0, -HOME_TIMELINE_SIZE-1)

    pipeline.execute()
    return True
```

Find the size of the following and followers ZSETs.

Fetch the most recent HOME_TIMELINE_SIZE status messages from the newly followed user's profile timeline.

Update the known size of the following and followers ZSETs in each user's HASH.

Update the home timeline of the following user, keeping only the most recent 1000 status messages.

Return that the user was correctly followed.

CONVERTING A LIST OF TUPLES INTO A DICTIONARY As part of our `follow_user()` function, we fetched a list of status message IDs along with their timestamp scores. Because this is a sequence of pairs, we can pass them directly to the `dict()` type, which will create a dictionary of keys and values, as passed.

This function proceeds in the way we described earlier: we add the appropriate user IDs to the following and followers ZSETs, get the size of the following and followers ZSETs, and fetch the recent status message IDs from the followed user's profile timeline. After we've fetched all of the data, we then update counts inside the user profile HASHes, and update the following user's home timeline.

After following someone and reading their status messages for a while, we may get to a point where we decide we don't want to follow them anymore. To stop following someone, we perform essentially the reverse operations of what we've discussed: removing UIDs from followers and following lists, removing status messages, and again updating the followers/following counts. The code to stop following someone is shown in the following listing.

Listing 8.5 A function to stop following a user

```
def unfollow_user(conn, uid, other_uid):
    fkey1 = 'following:%s'%uid
    fkey2 = 'followers:%s'%other_uid

    if not conn.zscore(fkey1, other_uid):
        return None

    pipeline = conn.pipeline(True)
    pipeline.zrem(fkey1, other_uid)
    pipeline.zrem(fkey2, uid)
    pipeline.zcard(fkey1)
    pipeline.zcard(fkey2)
    pipeline.zrevrange('profile:%s'%other_uid,
        0, HOME_TIMELINE_SIZE-1)
    following, followers, statuses = pipeline.execute()[-3:]
```

Cache the following and followers key names.

If the other_uid isn't being followed, return.

Remove the uids from the proper following and followers ZSETs.

Find the size of the following and followers ZSETs.

Fetch the most recent HOME_TIMELINE_SIZE status messages from the user that we stopped following.

```
    pipeline.hset('user:%s'%uid, 'following', following)
    pipeline.hset('user:%s'%other_uid, 'followers', followers)
    if statuses:
        pipeline.zrem('home:%s'%uid, *statuses)

    pipeline.execute()
    return True
```

Update the known size of the following and followers ZSETs in each user's HASH.

Update the home timeline, removing any status messages from the previously followed user.

Return that the unfollow executed successfully.

In that function, we updated the following and followers lists, updated the followers and following counts, and updated the home timeline to remove status messages that should no longer be there. As of now, that completes all of the steps necessary to start and stop following a user.

Exercise: Refilling timelines

When someone stops following another user, some number of status messages will be removed from the former follower's home timeline. When this happens, we can either say that it's okay that fewer than the desired number of status messages are in the timeline, or we can make an effort to add status messages from the other people that the user is still following. Can you write a function that will add status messages to the user's timeline to keep it full? Hint: You may want to use tasks like we defined in section 6.4 to reduce the time it takes to return from an unfollow call.

Exercise: Lists of users

In addition to the list of users that someone follows, Twitter also supports the ability to create additional named lists of users that include the timeline of posts for just those users. Can you update `follow_user()` and `unfollow_user()` to take an optional "list ID" for storing this new information, create functions to create a custom list, and fetch the custom list? Hint: Think of it like a different type of follower. Bonus points: can you also update your function from the "Refilling timelines" exercise?

Now that we can start or stop following a user while keeping the home timeline updated, it's time to see what happens when someone posts a new status update.

8.4 Posting or deleting a status update

One of the most fundamental operations on a service like Twitter is posting status messages. People post to share their ideas, and people read because they're interested in what's going on with others. Section 8.1.2 showed how to create a status message as a prerequisite for knowing the types of data that we'll be storing, but didn't show how to get that status message into a profile timeline or the home timeline of the user's followers.

In this section, we'll discuss what happens to a status message when it's posted so it can find its way into the home timelines of that user's followers. We'll also talk about how to delete a status message.

You already know how to create the status message itself, but we now need to get the status message ID into the home timeline of all of our followers. How we should perform this operation will depend on the number of followers that the posting user happens to have. If the user has a relatively small number of followers (say, up to 1,000 or so), we can update their home timelines immediately. But for users with larger number of followers (like 1 million, or even the 25 million that some users have on Twitter), attempting to perform those insertions directly will take longer than is reasonable for a user to wait.

To allow for our call to return quickly, we'll do two things. First, we'll add the status ID to the home timelines of the first 1,000 followers as part of the call that posts the status message. Based on statistics from a site like Twitter, that should handle at least 99.9% of all users who post (Twitter-wide analytics suggest that there are roughly 100,000–250,000 users with more than 1,000 followers, which amounts to roughly .1% of the active user base). This means that only the top .1% of users will need another step.

Second, for those users with more than 1,000 followers, we'll start a deferred task using a system similar to what we built back in section 6.4. The next listing shows the code for pushing status updates to followers.

Listing 8.6 Update a user's profile timeline

```
def post_status(conn, uid, message, **data):
    id = create_status(conn, uid, message, **data)
    if not id:
        return None

    posted = conn.hget('status:%s'%id, 'posted')
    if not posted:
        return None

    post = {str(id): float(posted)}
    conn.zadd('profile:%s'%uid, **post)

    syndicate_status(conn, uid, post)
    return id
```

Create a status message using the earlier function.

If the creation failed, return.

Get the time that the message was posted.

If the post wasn't found, return.

Add the status message to the user's profile timeline.

Actually push the status message out to the followers of the user.

Notice that we broke our status updating into two parts. The first part calls the `create_status()` function from listing 8.2 to actually create the status message, and then adds it to the poster's profile timeline. The second part actually adds the status message to the timelines of the user's followers, which can be seen next.

Listing 8.7 Update a user's followers' home timelines

```
POSTS_PER_PASS = 1000
def syndicate_status(conn, uid, post, start=0):
```

Only send to 1000 users per pass.

```
    followers = conn.zrangebyscore('followers:%s'%uid, start, 'inf',
        start=0, num=POSTS_PER_PASS, withscores=True)

    pipeline = conn.pipeline(False)
    for follower, start in followers:
        pipeline.zadd('home:%s'%follower, **post)
        pipeline.zremrangebyrank(
            'home:%s'%follower, 0, -HOME_TIMELINE_SIZE-1)
    pipeline.execute()

    if len(followers) >= POSTS_PER_PASS:
        execute_later(conn, 'default', 'syndicate_status',
            [conn, uid, post, start])
```

Fetch the next group of 1000 followers, starting at the last person to be updated last time.

Iterating through the followers results will update the "start" variable, which we can later pass on to subsequent syndicate_status() calls.

Add the status to the home timelines of all of the fetched followers, and trim the home timelines so they don't get too big.

If at least 1000 followers had received an update, execute the remaining updates in a task.

This second function is what actually handles pushing status messages to the first 1,000 followers' home timelines, and starts a delayed task using the API we defined in section 6.4 for followers past the first 1,000. With those new functions, we've now completed the tools necessary to actually post a status update and send it to all of a user's followers.

> **Exercise: Updating lists**
>
> In the last section, I suggested an exercise to build named lists of users. Can you extend the `syndicate_message()` function to also support updating the list timelines from before?

Let's imagine that we posted a status message that we weren't proud of; what would we need to do to delete it?

It turns out that deleting a status message is pretty easy. Before returning the fetched status messages from a user's home or profile timeline in `get_messages()`, we're already filtering "empty" status messages with the Python `filter()` function. So to delete a status message, we only need to delete the status message `HASH` and update the number of status messages posted for the user. The function that deletes a status message is shown in the following listing.

Listing 8.8 A function to delete a previously posted status message

```
def delete_status(conn, uid, status_id):
    key = 'status:%s'%status_id
    lock = acquire_lock_with_timeout(conn, key, 1)
    if not lock:
        return None

    if conn.hget(key, 'uid') != str(uid):
        return None

    pipeline = conn.pipeline(True)
    pipeline.delete(key)
    pipeline.zrem('profile:%s'%uid, status_id)
```

Acquire a lock around the status object to ensure that no one else is trying to delete it when we are.

If we didn't get the lock, return.

If the user doesn't match the user stored in the status message, return.

Delete the status message.

Remove the status message id from the user's profile timeline.

```
    pipeline.zrem('home:%s'%uid, status_id)
    pipeline.hincrby('user:%s'%uid, 'posts', -1)
    pipeline.execute()

    release_lock(conn, key, lock)
    return True
```

Remove the status message ID from the user's home timeline.

Reduce the number of posted messages in the user information HASH.

While deleting the status message and updating the status count, we also went ahead and removed the message from the user's home timeline and profile timeline. Though this isn't technically necessary, it does allow us to keep both of those timelines a little cleaner without much effort.

Exercise: Cleaning out deleted IDs

As status messages are deleted, "zombie" status message IDs will still be in the home timelines of all followers. Can you clean out these status IDs? Hint: Think about how we sent the messages out in the first place. Bonus points: also handle lists.

Being able to post or delete status messages more or less completes the primary functionality of a Twitter-like social network from a typical user's perspective. But to complete the experience, you may want to consider adding a few other features:

- Private users, along with the ability to request to follow someone
- Favorites (keeping in mind the privacy of a tweet)
- Direct messaging between users
- Replying to messages resulting in conversation flow
- Reposting/retweeting of messages
- The ability to @mention users or #tag ideas
- Keeping a record of who @mentions someone
- Spam and abuse reporting and controls

These additional features would help to round out the functionality of a site like Twitter, but may not be necessary in every situation. Expanding beyond those features that Twitter provides, some social networks have chosen to offer additional functionality that you may want to consider:

- Liking/+1 voting status messages
- Moving status messages around the timeline depending on "importance"
- Direct messaging between a prespecified group of people (like in section 6.5.2)
- Groups where users can post to and/or follow a group timeline (public groups, private groups, or even announcement-style groups)

Now that we've built the last piece of the standard functional API for actually servicing a site like Twitter, let's see what it'd take to build a system for processing streaming API requests.

8.5 Streaming API

As development of our social network continues, at some point we'll want to learn more about what's going on—maybe to discover how many posts are made every hour, the most-talked-about topics, or even who's being mentioned all the time. One way of doing this is to make calls to gather this information. Another way is to record this information inside the functions that perform all of the operations. The third way, which we'll explore in this section, is to build our functions to broadcast simple events, which are received and processed by event listeners to analyze the data.

In this section, I'll describe how to build the back end for a streaming API that functions similar to the streaming API offered by Twitter.

Unlike the other parts of the system that we've already built, the streaming API is a different group of functionalities altogether. The functions that we built to support the typical operations of a site like Twitter in the last several sections were meant to execute and complete quickly. On the other hand, a streaming API request is meant to return data over a longer period of time.

Most modern social networks offer the ability to gather information from their system via some sort of API. One advantage that Twitter has shown over the last several years is that by offering real-time events to third parties, those third parties can develop unique and interesting analyses of the data that Twitter itself may not have had the time or interest to develop.

The first step in building a streaming API is understanding what kind of data we'll be processing and producing.

8.5.1 Data to be streamed

As people perform a variety of actions within our social network, those actions are seen at the various functions that defined our API. In particular, we spent most of our time building out the ability to follow/unfollow users, and post/delete messages. If we'd built other pieces of our social network, we'd also find a variety of other *events* that occur as the result of user behavior. A streaming API is meant to produce a sequence of these events over time as a way of keeping clients or other services updated about a subset of what's going on across the entire network.

In the process of building a streaming API, a variety of decisions must be made, which can be generally reduced to three major questions:

- Which events should be exposed?
- What access restrictions (if any) should exist?
- What kinds of filtering options should be provided?

For now, I won't answer the second question about access restrictions. That's a question that we need to answer when we're building our social network based on expectations of privacy and system resources. But I'll answer the other two questions.

Because we focused on posting/deleting messages and following/unfollowing users, we should offer at least some of those events. To keep things simple for now, we'll only produce message posting and deletion events. But based on the structures that we create and pass around, adding functionality to support follow/unfollow events or events for other actions that we've added should be easy.

The types of filtering options that we'll provide will overlap significantly with the API features and functionality that Twitter provides on the public side of things. In particular, we'll offer the ability to filter over messages with an equivalent of follow (users), track (keywords), and location filters, in addition to a randomly selected subset of messages, similar to Twitter's *firehose* and *sample* streams.

Now that we know what data we'll have access to, let's start looking at how we'll serve the data.

8.5.2 Serving the data

In preceding sections and chapters, when we showed functions that made calls to Redis, we built on the assumption that we had an existing web server that would be calling these functions at just the right time. In the case of a streaming API, the details of streaming data to a client can be more complicated than just plugging these functions into an existing web service stack. In particular, most web servers operate under the assumption that we'll be returning the entire response to a request at once, but this is definitely not the case with a streaming API.

Responses from a streaming API are received status message by status message as they're produced and matched. Though modern technologies like WebSockets and SPDY can offer incremental data production, or even server-side push messages, the protocols involved are still in the process of being finalized, and client-side support in many programming languages is incomplete. But there is a method of producing incremental content with an HTTP server—sending data using the *chunked* transfer encoding.

In this section, we'll build a simple web server that supports streaming to clients that can handle chunked HTTP responses. This is to support our later sections which will actually implement filtering options for streamed message data.

To build this streaming HTTP web server, we have to delve deeper into the Python programming language. In the past, we've attempted to keep everything to standard functions, and in chapter 6, we even started using generators (that was the code that included `yield`). But here, we'll have to use Python classes. This is primarily because we don't want to have to build an entire web server from scratch, and Python already includes servers that we can mix together to handle all of the difficult parts of web serving. If you've used classes in other languages, you'll be comfortable with Python, because classes in Python are similar. They're meant to encapsulate data, with methods to manipulate the data. In our case, most of the functionality that we want to use is already available in existing libraries; we just need to plug them together.

A STREAMING HTTP SERVER

Within Python we have a series of socket server libraries that can be mixed together to offer varying types of functionality. To start, we'll create a server that uses threads in order to process each incoming request separately. When the server receives a request, the server will create a thread to execute a request handler. This request handler is where we'll perform some initial basic routing for GET and POST HTTP requests. Both the threaded server and the request handler are shown in the next listing.

Listing 8.9 Server and request handler for our streaming HTTP server

```
class StreamingAPIServer(
    SocketServer.ThreadingMixIn,
    BaseHTTPServer.HTTPServer):

    daemon_threads = True

class StreamingAPIRequestHandler(
    BaseHTTPServer.BaseHTTPRequestHandler):

    def do_GET(self):
        parse_identifier(self)
        if self.path != '/statuses/sample.json':
            return self.send_error(404)

        process_filters(self)

    def do_POST(self):
        parse_identifier(self)
        if self.path != '/statuses/filter.json':
            return self.send_error(404)

        process_filters(self)
```

Create a new class called "StreamingAPIServer".

This new class should have the ability to create new threads with each request, and should be an HTTPServer.

Tell the internals of the threading server to shut down all client request threads if the main server thread dies.

Create a new class called "StreamingAPI-RequestHandler".

This new class should be able to handle HTTP requests.

Create a method that is called do_GET(), which will be executed on GET requests performed against this server.

Call a helper function that handles the fetching of an identifier for the client.

If the request is not a "sample" or "firehose" streaming GET request, return a "404 not found" error.

Otherwise, call a helper function that actually handles the filtering.

Create a method that is called do_POST(), which will be executed on POST requests performed against this server.

Call a helper function that handles the fetching of an identifier for the client.

If the request is not a user, keyword, or location filter, return a '404 not found' error.

Otherwise, call a helper function that actually handles the filtering.

What we didn't write is the code that actually starts up the server, but we'll get to that in a moment. For now, you can see that we defined a server that created threads on each request. Those threads execute methods on a request handler object, which eventually lead to either `do_GET()` or `do_POST()`, which handle the two major types of streaming API requests: filtered and sampled.

To actually run this server, we'll use a bit of Python magic. This magic allows us to later import a module to use these predefined classes, or it allows us to run the module directly in order to start up a streaming API server. The code that lets us both import the module and run it as a daemon can be seen in the next listing.

Before you put these two blocks of code into a file and run them, remember that we're still missing two functions that are called as part of the streaming API server, `parse_identifier()` and `process_filters()`, which we'll cover next.

Listing 8.10 The code to actually start and run the streaming HTTP server

```
if __name__ == '__main__':
    server = StreamingAPIServer(
        ('localhost', 8080), StreamingAPIRequestHandler)
    print 'Starting server, use <Ctrl-C> to stop'
    server.serve_forever()
```

Run the block of code below if this module is being run from the command line.

Create an instance of the streaming API server listening on localhost port 8080, and use the StreamingAPIRequestHandler to process requests.

Print an informational line.

Run the server until someone kills it.

IDENTIFYING THE CLIENT

The first of these two functions is a way of fetching identifying information about the client. This basic method extracts an identifier from the request query arguments. For a production scenario, we'd want to perform some amount of client validation of the identifier. Our simple method to parse an identifier from the request can be seen in the next listing.

Listing 8.11 An example function to parse and store the client identifier

```
def parse_identifier(handler):
    handler.identifier = None
    handler.query = {}
    if '?' in handler.path:
        handler.path, _, query = handler.path.partition('?')
        handler.query = urlparse.parse_qs(query)
        identifier = handler.query.get('identifier') or [None]
        handler.identifier = identifier[0]
```

Set the identifier and query arguments to be placeholder values.

If there were query arguments as part of the request, process them.

Extract the query portion from the path and update the path.

Parse the query.

Fetch the list of query arguments with the name "identifier."

Use the first identifier passed.

That function shouldn't do anything surprising; we set some initial values for the query arguments (if we want to use them later) and the identifier, parse the query arguments, and then store the identifier from the query if it was available.

HANDLING HTTP STREAMING

There's one final piece to the HTTP server portion of our request—actually sending the filtered responses. To prepare to send these filtered messages one by one, we first need to verify the requests are valid. Assuming that everything is okay, we must then send to the client the notification that we'll be entering an HTTP mode called *chunked transfer encoding*, which will allow us to send messages one at a time as they come in. The function that performs this validation and the actual transfer of streamed messages to the client is shown next.

Listing 8.12 A function that will verify the request and stream data to the client

```
FILTERS = ('track', 'filter', 'location')
def process_filters(handler):
    id = handler.identifier
    if not id:
        return handler.send_error(401, "identifier missing")
```

Keep a listing of filters that need arguments.

Return an error if an identifier was not provided by the client.

Fetch the method; should be one of "sample" or "filter".

If this is a filtering method, we need to fetch the arguments.

Parse the POST request to discover the type and arguments to the filter.

```
    method = handler.path.rsplit('/')[-1].split('.')[0]
    name = None
    args = None
    if method == 'filter':
        data = cgi.FieldStorage(
            fp=handler.rfile,
            headers=handler.headers,
            environ={'REQUEST_METHOD':'POST',
                     'CONTENT_TYPE':handler.headers['Content-Type'],
        })
```

Fetch any of the filters provided by the client request.

```
        for name in data:
            if name in FILTERS:
                args = data.getfirst(name).lower().split(',')
                break
```

If there were no filters specified, return an error.

For sample requests, pass the query arguments as the "args".

Finally, return a response to the client, informing them that they will be receiving a streaming response.

```
        if not args:
            return handler.send_error(401, "no filter provided")
    else:
        args = handler.query

    handler.send_response(200)
    handler.send_header('Transfer-Encoding', 'chunked')
    handler.end_headers()
```

Use a Python list as a holder for a pass-by-reference variable, which will allow us to tell the content filter to stop receiving messages.

Iterate over the results of the filter.

Send the pre-encoded response to the client using the chunked encoding.

If sending to the client caused an error, then we need to tell the subscriber to unsubscribe and shut down.

Send the "end of chunks" message to the client if we haven't already disconnected.

```
    quit = [False]
    for item in filter_content(id, method, name, args, quit):
        try:
            handler.wfile.write('%X\r\n%s\r\n'%(len(item), item))
        except socket.error:
            quit[0] = True
    if not quit[0]:
        handler.wfile.write('0\r\n\r\n')
```

A few details in this function are tricky, but the basic idea is that we make sure that we have an identifier for the client and fetch the filtering arguments for the specific calls. If everything is okay, we then announce to the client that we'll be streaming responses and pass the actual filtering off to a generator, which will produce the sequence of messages that match the filter criteria.

And that's it for the streaming HTTP server. In the next section, we'll build the methods that will filter messages that pass through the system.

8.5.3 *Filtering streamed messages*

So far we've built a server to serve the streamed messages; now it's time to filter through the messages for streaming. We filter the messages so that a client making a request only sees the messages they're interested in. Though our social network may not have a lot of traffic, sites like Twitter, Facebook, or even Google+ will see tens to hundreds of thousands of events every second. And for both third parties and ourselves, the cost of bandwidth to send all of that information can be quite high, so only sending messages that match up is important.

In this section, we'll write functions and classes that will filter posted messages to be streamed to clients. These filters will plug into the streaming web server we wrote in section 8.5.2. As I mentioned at the beginning of section 8.5, we'll support random sampling of all messages and access to the full firehose, as well as filtering for specific users, words, and the location of messages.

As mentioned way back in chapter 3, we'll use Redis `PUBLISH` and `SUBSCRIBE` to implement at least part of the streaming functionality. More specifically, when users post messages, we'll `PUBLISH` the posted message information to a channel in Redis. Our filters will `SUBSCRIBE` to that same channel, receive the message, and yield messages that match the filters back to the web server for sending to the client.

UPDATING STATUS MESSAGE POSTING AND DELETION

Before we get ahead of ourselves, let's first update our message posting function from section 8.1.2 and message deletion function from section 8.4 to start producing messages to filter. We'll start with posting in the next listing, which shows that we've added a line to our function that sends messages out to be filtered.

Listing 8.13 Updated `create_status()` from listing 8.2 to support streaming filters

```
def create_status(conn, uid, message, **data):
    pipeline = conn.pipeline(True)
    pipeline.hget('user:%s'%uid, 'login')
    pipeline.incr('status:id:')
    login, id = pipeline.execute()

    if not login:
        return None

    data.update({
        'message': message,
        'posted': time.time(),
        'id': id,
        'uid': uid,
        'login': login,
    })
    pipeline.hmset('status:%s'%id, data)
    pipeline.hincrby('user:%s'%uid, 'posts')
    pipeline.publish('streaming:status:', json.dumps(data))    <- The added line to send a message to streaming filters
    pipeline.execute()
    return id
```

All it took was one more line to add streaming support on the posting side. But what about deletion? The update to status message deletion is shown in the following listing.

Listing 8.14 Updated `delete_status()` from listing 8.8 to support streaming filters

```
def delete_status(conn, uid, status_id):
    key = 'status:%s'%status_id
    lock = acquire_lock_with_timeout(conn, key, 1)
    if not lock:
        return None

    if conn.hget(key, 'uid') != str(uid):
        return None

    pipeline = conn.pipeline(True)
    status = conn.hgetall(key)
    status['deleted'] = True
    pipeline.publish('streaming:status:', json.dumps(status))
    pipeline.delete(key)
    pipeline.zrem('profile:%s'%uid, status_id)
    pipeline.zrem('home:%s'%uid, status_id)
    pipeline.hincrby('user:%s'%uid, 'posts', -1)
    pipeline.execute()

    release_lock(conn, key, lock)
    return True
```

Fetch the status message so that streaming filters can perform the same filters to determine whether the deletion should be passed to the client.

Mark the status message as deleted.

Publish the deleted status message to the stream.

At first glance, you're probably wondering why we'd want to send the entire status message that's to be deleted to the channel for filtering. Conceptually, we should only need to send message-deleted information to clients that received the status message when it was posted. If we perform the same filtering on deleted messages as we do on newly posted messages, then we can always send message-deleted notifications to those clients that would've received the original message. This ensures that we don't need to keep a record of the status IDs for messages sent to all clients, which simplifies our server and reduces memory use.

RECEIVING STREAMED MESSAGES FOR FILTERING

Now that we're sending information about status messages being posted and deleted to a channel in Redis, we only need to subscribe to that channel to start receiving messages to filter. As was the case in chapter 3, we'll need to construct a special `pubsub` object in order to subscribe to a channel. When we've subscribed to the channel, we'll perform our filtering, and produce one of two different messages depending on whether the message was posted or deleted. The code for handling these operations is next.

Listing 8.15 A function to receive and process streamed messages

```
@redis_connection('social-network')
def filter_content(conn, id, method, name, args, quit):
    match = create_filters(id, method, name, args)

    pubsub = conn.pubsub()
    pubsub.subscribe(['streaming:status:'])

    for item in pubsub.listen():
        message = item['data']
        decoded = json.loads(message)

        if match(decoded):
```

Use our automatic connection decorator from chapter 5.

Create the filter that will determine whether a message should be sent to the client.

Prepare the subscription.

Receive messages from the subscription.

Get the status message information from the subscription structure.

Check if the status message matched the filter.

```
            if decoded.get('deleted'):
                yield json.dumps({
                    'id': decoded['id'], 'deleted': True})
            else:
                yield message

        if quit[0]:
            break

    pubsub.reset()
```

For deleted messages, send a special "deleted" placeholder for the message.

For matched status messages that are not deleted, send the message itself.

If the web server no longer has a connection to the client, stop filtering messages.

Reset the Redis connection to ensure that the Redis server clears its outgoing buffers if this wasn't fast enough.

As I said before, this function needs to subscribe to a channel in Redis in order to receive posted/deleted notifications for status messages. But it also needs to handle cases where the streaming client has disconnected, and it needs to properly clean up the connection if Redis has been trying to send it too much data.

As we covered in chapter 3, there's a Redis server setting to determine the maximum outgoing buffer for subscriptions to support. To ensure that our Redis server stays up even under heavy load, we'll probably want to set `client-output-buffer-limit pubsub` to lower than the default 32 megabytes per connection. Where to set the limit will depend on how many clients we expect to support and how much other data is in Redis.

FILTERING MESSAGES

At this point we've built every other layer; it now remains to actually write filtering. I know, there was a lot of build-up, but you may be surprised to find out that actually filtering messages isn't difficult for any of our cases. To create filters, we'll first define our `create_filters()` function in listing 8.16, which will delegate off to one of a variety of filtering classes, depending on the filter that we want to build. We'll assume that clients are sending reasonable arguments, but if you're considering using any of this in a production scenario, you'll want to add validation and verification.

Listing 8.16 A factory function to dispatch to the actual filter creation

```
def create_filters(id, method, name, args):
    if method == 'sample':
        return SampleFilter(id, args)
    elif name == 'track':
        return TrackFilter(args)
    elif name == 'follow':
        return FollowFilter(args)
    elif name == 'location':
        return LocationFilter(args)
    raise Exception("Unknown filter")
```

For the "sample" method, we don't need to worry about names, just the arguments.

For the "filter" method, we actually worry about which of the filters we want to apply, so return the specific filters for them.

If no filter matches, then raise an exception.

Nothing surprising there: we're distinguishing the different kinds of filters. The first filter we'll create will be the `sample` filter, which will actually implement the functionality of the Twitter-style `firehose`, `gardenhose`, and `spritzer` access levels, and anything in between. The implementation of the sampling filter is shown next.

Listing 8.17 The function to handle `firehose`, `gardenhose`, and `spritzer`

```
def SampleFilter(id, args):
    percent = int(args.get('percent', ['10'])[0], 10)
    ids = range(100)
    shuffler = random.Random(id)
    shuffler.shuffle(ids)
    keep = set(ids[:max(percent, 1)])

    def check(status):
        return (status['id'] % 100) in keep
    return check
```

We're defining a filter class called "SampleFilter", which is created by passing "id" and "args" parameters.

The "args" parameter is actually a dictionary based on the parameters passed as part of the GET request.

We use the "id" parameter to randomly choose a subset of IDs, the count of which is determined by the "percent" argument passed.

We'll use a Python set to allow us to quickly determine whether a status message matches our criteria.

If we create a specially named method called "__call__" on an instance, it will be called if the instance is used like a function.

To filter status messages, we fetch the status ID, find its value modulo 100, and return whether it's in the status IDs that we want to accept.

As you can see, we started using classes again, primarily because we need to encapsulate data and behavior together. This first class that defines sampling does one interesting thing—it uses a random number generator seeded with the user-provided identifier to choose the IDs of status messages that it should accept. This allows the sampling filters to receive a `deleted` notification for a message, even if the client had disconnected (as long as the client reconnected before the delete notification came through). We use Python sets here to quickly determine whether the ID modulo 100 is in the group that we want to accept, as Python sets offer O(1) lookup time, compared to O(n) for a Python list.

Continuing on, we'll now build the `track` filter, which will allow users to track words or phrases in status messages. Similar to our sample filter in listing 8.17, we'll use a class to encapsulate the data and filtering functionality together. The filter class definition is shown in the following listing.

Listing 8.18 A filter that matches groups of words that are posted in status messages

```
def TrackFilter(list_of_strings):
    groups = []
    for group in list_of_strings:
        group = set(group.lower().split())
        if group:
            groups.append(group)

    def check(status):
        message_words = set(status['message'].lower().split())
        for group in groups:
            if len(group & message_words) == len(group):
                return True
        return False
    return check
```

The filter has been provided with a list of word groups, and the filter matches if a message has all of the words in any of the groups.

We'll only keep groups that have at least 1 word.

We'll split words in the message on whitespace.

Then we'll iterate over all of the groups.

If all of the words in any of the groups match, we'll accept the message with this filter.

About the only interesting thing about the tracking filter is to make sure that if someone wants to match a group of words, the filter matches *all* of the words in the message and not just *some* of them. We again use Python sets, which, like Redis SETs, offer the ability to calculate intersections.

Moving on to the follow filter, we're trying to match status messages that were posted by one of a group of users, or where one of the users is mentioned in the message. The class that implements user matching is shown here.

Listing 8.19 Messages posted by or mentioning any one of a list of users

```
def FollowFilter(names):
    names = set()
    for name in names:
        names.add('@' + name.lower().lstrip('@'))

    def check(status):
        message_words = set(status['message'].lower().split())
        message_words.add('@' + status['login'].lower())

        return message_words & names
    return check
```

We'll match login names against posters and messages.

Store all names consistently as '@username'.

Construct a set of words from the message and the poster's name.

Consider the message a match if any of the usernames provided match any of the whitespace-separated words in the message.

As before, we continue to use Python sets as a fast way to check whether a name is in the set of names that we're looking for, or whether any of the names to match are also contained in a status message.

We finally get to the location filter. This filter is different from the others in that we didn't explicitly talk about adding location information to our status messages. But because of the way we wrote our create_status() and post_status() functions to take additional optional keyword arguments, we can add additional information without altering our status creation and posting functions. The location filter for this optional data is shown next.

Listing 8.20 Messages within boxes defined by ranges of latitudes and longitudes

```
def LocationFilter(list_of_boxes):
    boxes = []
    for start in xrange(0, len(list_of_boxes)-3, 4):
        boxes.append(map(float, list_of_boxes[start:start+4]))

    def check(self, status):
        location = status.get('location')
        if not location:
            return False

        lat, lon = map(float, location.split(','))
        for box in self.boxes:
```

We'll create a set of boxes that define the regions that should return messages.

Try to fetch "location" data from a status message.

If the message has no location information, then it can't be inside the boxes.

Otherwise, extract the latitude and longitude of the location.

To match one of the boxes, we need to iterate over all boxes.

```
                if (box[1] <= lat <= box[3] and
                    box[0] <= lon <= box[2]):
                    return True
            return False
        return check
```

If the message status location is within the required latitude and longitude range, then the status message matches the filter.

About the only thing that may surprise you about this particular filter is how we're preparing the boxes for filtering. We expect that requests will provide location boxes as comma-separated sequences of numbers, where each chunk of four numbers defines latitude and longitude ranges (minimum longitude, minimum latitude, maximum longitude, maximum latitude—the same order as Twitter's API).

With all of our filters built, a working web server, and the back-end API for everything else, it's now up to you to get traffic!

8.6 Summary

In this chapter, we've built the majority of functionality that makes a site like Twitter work. Though these structures won't scale to the extent that Twitter does, the methods used can be used to build a small social network easily. With a front end for users to interact with, you can start your own social network with your friends!

If there's one thing that you should take away from this chapter, it's that even immensely popular websites have functionality that can be built with the tools available inside of Redis.

In the upcoming chapters 9 through 11, we'll look into methods to help reduce memory use, methods to help scaling Redis read and write loads, and scripting Redis to simplify (and sometimes help scale) applications. These things will help to scale Redis applications, like our social network, beyond expected single-machine limits. Our first step down this path is chapter 9, where I'll show you how to reduce Redis's memory use.

DATABASES/NOSQL

Redis IN ACTION

Josiah L. Carlson

When you need near-real-time access to a fast-moving data stream, key-value stores like Redis are the way to go. Redis expands on the key-value pattern by accepting a wide variety of data types, including hashes, strings, lists, and other structures. It provides lightning-fast operations on in-memory datasets, and also makes it easy to persist to disk on the fly. Plus, it's free and open source.

Redis in Action introduces Redis and the key-value model. You'll quickly dive into real use cases including simple caching, distributed ad targeting, and more. You'll learn how to scale Redis from small jobs to massive datasets and discover how to integrate with traditional RDBMS or other NoSQL stores. Experienced developers will appreciate the in-depth chapters on clustering and internal scripting.

What's Inside

- Redis from the ground up
- Preprocessing real-time data
- Managing in-memory datasets
- Pub/sub and configuration
- Persisting to disk

Written for developers familiar with database concepts. No prior exposure to NoSQL database concepts nor to Redis itself is required. Appropriate for systems administrators comfortable with programming.

Dr. Josiah L. Carlson is a seasoned database professional and an active contributor to the Redis community.

Free eBook
SEE INSERT

"A great addition to the Redis ecosystem."

—From the Foreword by Salvatore Sanfilippo, Creator of Redis

"The examples, taken from real-world use cases, are one of the major strengths of the book."

—Filippo Pacini, SG Consulting

"Fom beginner to expert with real and comprehensive examples."

—Felipe Gutierrez
VMware/Spring Source

"Excellent in-depth analysis ... insightful real-world examples."

—Bobby Abraham, Integri LLC

"Pure gold!"

—Leo Cassarani
Unboxed Consulting

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/RedisinAction

ISBN 13: 978-1-617290-85-5
ISBN 10: 1-617290-85-8
54499
9 781617 290855

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]