Если вы видите что-то необычное, просто сообщите мне.

Redis Caching

In part 1 of this series we used Persistent to store a User type in a Postgresql database. Then in part 2 we used Servant to create a very simple API that exposed this database to the outside world. This week, we're going to look at how we can improve the performance of our API using a Redis cache.

If you've already read this part, you can move onto part 4 and learn how we test this system! You can also download our Production Checklist for a lot more ideas on useful tools to use in your production Haskell applications!

You can follow along the code for this part by looking at our Github repository! You'll specifically want to look at two files. First, the Cache module for the Redis specific code, and then the CacheServer module for an updated version of our Servant server.

CACHING 101

One cannot overstate the importance of caching in both software and hardware. There's a hierarchy of memory types from registers to RAM, to the File system, to a remote database, and so on. Accessing each of these gets progressively slower (by orders of magnitude). But the faster means of storage are more expensive, so we can't always have as much as we'd like.

But memory usage operates on a very important principle. When we use a piece of memory once, we're very likely to use it again in the near-future. So when we pull something out of long-term memory, we can temporarily store it in short-term memory as well. This way when we need it again, we can get it faster. After a certain point, that item will be overwritten by other more urgent items. This is the essence of caching.

REDIS

Redis is an application that allows us to create a key-value store of items. It functions like a database, except it only uses these keys. It lacks the sophistication of joins, foreign table references and indices. So we can't run the kinds of more complex queries that are possible on an SQL database. But we can run simple key lookups, and we can do them faster. In this article, we'll use Redis as a short-term cache for our user objects.

For this article, we've got one main goal for cache integration. Whenever we "fetch" a user using the GET endpoint in our API, we want to store that user in our Redis cache. Then the next time someone requests that user from our API, we'll grab them out of the cache. This will save us the trouble of making a longer call to our Postgres database.

#CONNECTING TO REDIS Haskell's Redis library has a lot of similarities to Persistent and Postgres. First, we'll need some sort of data that tells us where to look for our database. For Postgres, we used a simple ConnectionString with a particular format. Redis uses a full data type called ConnectInfo. (In our code, we alias this type as RedisInfo).

```
data ConnectInfo = ConnectInfo
  { connectHost :: HostName -- String
  , connectPort :: PortId   -- (Can just be a number)
  , connectAuth :: Maybe ByteString
  , connectDatabase :: Integer
  , connectMaxConnection :: Int
  , connectMaxIdleTime :: NominalDiffTime
  }
```

This has many of the same fields we stored in our PG string, like the host IP address, and the port number. The rest of this article assumes you are running a local Redis instance at port 6379. This means we can use defaultConnectInfo. As always, in a real system you'd want to grab this information out of a configuration, so you'd need IO.

```
fetchRedisConnection :: IO ConnectInfo
fetchRedisConnection = return defaultConnectInfo
```

With Postgres, we used withPostgresqlConn to actually connect to the database. With Redis, we do this with the connect function. We'll get a Connection object that we can use to run Redis actions.

```
connect :: ConnectInfo -> IO Connection
```

With this connection, we simply use runRedis, and then combine it with an action. Here's the wrapper runRedisAction we'll write for that:

```
runRedisAction :: ConnectInfo -> Redis a -> IO a
runRedisAction redisInfo action = do
  connection <- connect redisInfo
  runRedis connection action
```

THE REDIS MONAD

Just as we used the SqlPersistT monad with Persist, we'll use the Redis monad to interact with our Redis cache. Our API is simple, so we'll stick to three basic functions. The real types of these functions are a bit more complicated. But this is because of polymorphism related to transactions, and we won't be using those.

```
get :: ByteString -> Redis (Either x (Maybe ByteString))
set :: ByteString -> ByteString -> Redis (Either x ())
setex :: ByteString -> ByteString -> Int -> Redis (Either x ())
```

Redis is a key-value store, so everything we set here will use ByteString values. The get function takes a ByteString of the key and delivers the value as another ByteString. The set function takes both the serialized key and value and stores them in the cache. The setex function does the same thing as set except that it also sets an expiration time for the item we're storing.

Expiration is a very useful feature, since most relational databases don't have this. The nature of a cache is that it's only supposed to store a subset of our information at any given time. If we never expire or delete anything, it might eventually store our whole database. That would defeat the purpose of using a cache! It's memory footprint should remain low compared to our database. So we'll use setex in our API.

SAVING A USER IN REDIS

So now let's move on to the actions we'll actually use in our API. First, we'll write a function that will actually store a key-value pair of an Int64 key and the User in the database. Here's how we start:

```
cacheUser :: ConnectInfo -> Int64 -> User -> IO ()
cacheUser redisInfo uid user = runRedisAction redisInfo $ setex ??? ??? ???
```

All we need to do now is convert our key and our value to ByteString values. We'll keep it simple and use Data.ByteString.Char8 combined with our Show and Read instances. Then we'll create a Redis action using setex and expire the key after 3600 seconds (one hour).

```
import Data.ByteString.Char8 (pack, unpack)

...

cacheUser :: ConnectInfo -> Int64 -> User -> IO ()
cacheUser redisInfo uid user = runRedisAction redisInfo $ void $
  setex (pack . show $ uid) 3600 (pack . show $ user)
```

(We use void to ignore the result of the Redis call).

FETCHING FROM REDIS

Fetching a user is a similar process. We'll take the connection information and the key we're looking for. The action we'll create uses the bytestring representation and calls get. But we can't ignore the result of this call like we could before! Retrieving anything gives us Either e (Maybe ByteString). A Left response indicates an error, while Right Nothing indicates the key doesn't exist. We'll ignore the errors and treat the result as Maybe User though. If any error comes up, we'll return Nothing. This means we run a simple pattern match:

```
fetchUserRedis :: ConnectInfo -> Int64 -> IO (Maybe User)
fetchUserRedis redisInfo uid = runRedisAction redisInfo $ do
  result <- Redis.get (pack . show $ uid)
```

```
  case result of
    Right (Just userString) -> return $ Just (read . unpack $ userString)
    _ -> return Nothing
```

If we do find something for that key, we'll read it out of its ByteString format and then we'll have our final User object.

UPDATING OUR API

Now that we're all set up with our Redis functions, we have to update the fetchUsersHandler to use this cache. First, we now need to pass the Redis connection information as another parameter. For ease of reading, we'll refer to these using our type synonyms (PGInfo and RedisInfo) from now on:

```
type PGInfo = ConnectionString
type RedisInfo = ConnectInfo

...

fetchUsersHandler :: PGInfo -> RedisInfo -> Int64 -> Handler User
fetchUsersHandler pgInfo redisInfo uid = do
  ...
```

The first thing we'll try is to look up the user by their ID in the Redis cache. If the user exists, we'll immediately return that user.

```
fetchUsersHandler :: PGInfo -> RedisInfo -> Int64 -> Handler User
fetchUsersHandler pgInfo redisInfo uid = do
  maybeCachedUser <- liftIO $ fetchUserRedis redisInfo uid
  case maybeCachedUser of
    Just user -> return user
    Nothing -> do
      ...
```

If the user doesn't exist, we'll then drop into the logic of fetching the user in the database. We'll replicate our logic of throwing an error if we find that user doesn't actually exist. But if we find the user, we need one more step. Before we return it, we should call cacheUser and store it for the future.

```
fetchUsersHandler :: PGInfo -> RedisInfo -> Int64 -> Handler User
fetchUsersHandler pgInfo redisInfo uid = do
  maybeCachedUser <- liftIO $ fetchUserRedis redisInfo uid
  case maybeCachedUser of
    Just user -> return user
    Nothing -> do
      maybeUser <- liftIO $ fetchUserPG pgInfo uid
      case maybeUser of
        Just user -> liftIO (cacheUser redisInfo uid user) >> return user
        Nothing -> Handler $ (throwE $ err401 { errBody = "Could not find user with that ID"
})
```

Since we changed our type signature, we'll have to make a few other updates as well, but these are quite simple:

```
usersServer :: PGInfo -> RedisInfo -> Server UsersAPI
usersServer pgInfo redisInfo =
  (fetchUsersHandler pgInfo redisInfo) :<|>
  (createUserHandler pgInfo)


runServer :: IO ()
runServer = run 8000 (serve usersAPI (usersServer localConnString defaultConnectInfo))
```

And that's it! We have a functioning cache with expiring entries. This means that repeated queries to our fetch endpoint should be faster!

CONCLUSION

Caching is a vitally important way for us to write software that is much faster for our users. Redis is a key-value store we can use as a cache for our most commonly used data. We can use it instead of forcing every single API call to hit our database. In Haskell, the Redis API requires everything to be a ByteString. So we have to deal with some logic surrounding encoding and decoding. But otherwise it operates in a very similar way to Persistent and Postgres. Next, you should move onto part 4, where we'll get into how we test this complicated system!

We're starting to get to the point where we're using a lot of different libraries in our Haskell application! It pays to know how to organize everything, so package management is vital! I tend to use Stack for all my package management. It makes it quite easy to bring all these different libraries together. If you want to learn how to use Stack, check out our free Stack mini-course!

If you're already an expert in package management, perhaps you just want to expand your horizon of different libraries you know! In that case, take a look at our Production Checklist for some ideas!

Revision #1
Created 2022-03-11 06:19:54 UTC by gasick
Updated 2022-03-11 17:11:17 UTC by gasick