EVERYWHERE.

Writing modular & encapsulated Redis code

Jim Nelson
jnelson@archive.org
Internet Archive

redisconf18

#Redis #RedisConf

presented by:
redislabs
home of redis

Internet Archive

Universal Access to All Knowledge

- Founded 1996, based in San Francisco
- Archive of digital and physical media
- Includes Web, books, music, film, software & more
- Digital holdings: over 35 petabytes & counting
- Over 1 million users per day
- Key collections & services:
  - Wayback Machine
  - Grateful Dead live concert collection
  - The Emularity: in-browser emulation of 8-bit software, IBM PC, Mac, & now handhelds like Speak-and-Spell

Internet Archive ♡ Redis

- Caching & directory lookup backed by 10-node sharded Redis cluster
- Sharding performed client-side via consistent hashing (PHP, Predis)
- Each node supported by two replicated mirrors (fail-over)
- Specialized Redis instances also used throughout IA's services, including
    - Wayback Machine
    - Search (item metadata and full-text)
    - Administrative & usage tracking
    - …and more

Before we get started

- I’m assuming you’ve written Redis client code already
- You don’t need to know the exact details of the Redis protocol…
- …but you need some understanding of how Redis “works”
  - single-threaded
  - different data types (strings, hash, sorted sets, etc.)
  - transactions (MULTI/EXEC)

This presentation uses PHP sample code and Predis (a PHP Redis client) but most concepts presented should work in other languages & clients

Redis rocks...

“Baby, where have you been my whole life?”

- Quick, quick, quick
  - Quick to code
  - Quick to debug
  - Quick response times
  - Quick to model data
- Logical, well-defined command structure
- Atomicity
- Flexibility over rigor

Learning only a handful of straightforward concepts,
I felt I had a solid working understanding of Redis

…but there are issues.

“It’s complicated.” (As in, Facebook-it’s-complicated.)

- Application code mixed with data model code
  - Little-to-no separation between data logic and business logic
- Relational data often sprayed across separate Redis buckets
  - No clean way to normalize or deserialize complex objects if they’re sharded across keys
- Performance considerations trump clean coding practices
  - Modularity ignored in order to minimize round-trips to server

The temptation of Redis is that it looks like a collections framework.

What's missing?

All the things I know and love about designing clean code.

Modularity

Encapsulation

Readability

What is encapsulation?

Encapsulation “says that a program should be divided into modules that hide data structures behind an interface of procedure calls. The purpose of this is to allow you to change the underlying data structure without causing a large ripple effect across the system.”

–Martin Fowler, martinfowler.com

Redis programming today

Sending & receiving data

Redis (and most Redis clients) offer different modes of operation:

- Immediate mode
- Pipeline mode
- Transaction mode
- Atomic pipeline mode

Immediate mode

Send one command, receive one response.

That’s it. No, really.

```
$redis = new Predis\Client();
$name = $redis->get(“user:$id:name”);          // sends GET, recvs name
$email = $redis->get(“user:$id:email”);        // sends GET, recvs email
$login_count = $redis->incr(“user:$id:logins”) // sends INCR, recvs incremented number
```

Simple enough!

Pipeline mode

- Collect (or *schedule*) several commands on the client side
- Send all commands in one burst
- Receive all responses in one burst

```
$redis = new Predis\Client();
$pipeline = $redis->pipeline();    // no network I/O
$pipeline->get(“user:$id:name”);   // no network I/O
$pipeline->get(“user:$id:email”);  // no network I/O
$pipeline->incr(“user:$id:logins”);// no network I/O
$result = $pipeline->execute();    // sends GET/GET/INCR, recvs name, email, login
```

Single round-trip to the server but operations are *not* atomic

Transactions

- MULTI command opens transaction
- Send one or more commands
- EXEC closes transaction & receives all results
- Operations execute atomically

```
$redis = new Predis\Client();
$transaction = $redis->transaction(); // sends MULTI
$transaction->get(“user:$id:name”);   // sends GET, recvs QUEUED
$transaction->get(“user:$id:email”);  // sends GET, recvs QUEUED
$transaction->incr(“user:$id:logins”);// sends INCR, recvs QUEUED
$result = $transaction->execute();    // sends EXEC, recvs name, email, login (post-INCR)
```

Each operation is a separate round-trip to the server, including the MULTI and EXEC

Atomic pipelines

- Schedule several commands on the client side
- All commands are “wrapped” in a MULTI/EXEC
- Send all commands in one burst, receive responses on one burst
- Operations execute atomically

```
$redis = new Predis\Client();
$atomic = $redis->pipeline([‘atomic’ => true]);
$atomic->get(“user:$id:name”);    // no network I/O
$atomic->get(“user:$id:email”);   // no network I/O
$atomic->incr(“user:$id:logins”);// no network I/O
$result = $atomic->execute();     // sends MULTI/GET/GET/INCR/EXEC, recvs name, email, login
```

Atomic pipelines generally are more efficient than raw transactions

Warning:
Ugly code ahead.

Worse:
It’s written in PHP.

In the name of minimizing round-trips

```
/**
 * A do-everything function so opening a user session is a single
 * round-trip to the Redis server.
 */
function open_user_session($redis, $userid, $emoji_id, $notifications) {
      $pipe = $redis->pipeline();

      $pipe->get(“user:$userid:name”);
      $pipe->get(“user:$userid:member_of”);

      $pipe->hmget(“emoji:$emoji_id”, “tag”, “name”, “image”);

      $today = date(“Y-m-d”);
      $pipe->incr(“logins:$today”);

      $pipe->set(“session:$userid”, time(), “EX”, 600, “NX”);

      $pipe->sadd(“members:logged_in”, $userid);

      $results = $pipe->execute();
```

Do-everything functions break encapsulation

- Details of data layout in Redis is exposed
- Changing data structures means changing lots of code—the “ripple effect”
- Encourages code duplication—tempted to Ctrl+C / Ctrl+V
- Difficult to refactor
- Difficult to maintain

Passing pipeline object to other functions has fundamental problem:
How does the scheduling code receive the results of its read operation(s)?

Transforming & normalizing results

```
/**
 * Do-everything function continued.
 *
 * In Predis (and other clients) transaction and pipeline
 * results are returned in a numeric-indexed array.
 */
function open_user_session($redis, $id, $emoji_id, $notifications) {
      // … pipeline is executed as in previous slide …
      $user = new User($results[0]);
      $user->setMembershipObject(Membership::fromString($results[1]));

      $emoji = new Emoji($results[2][1], $results[2][0]);
      $emoji->setImage(base64_decode($results[2][2]));
      $user->setEmoji($emoji);

      StatsD::gauge(‘login_count’, (int) $results[3]);

      if ($results[4] != null) $notifications->sessionExtended($id);
      if ($results[5] > 0) Syslog::notice(“$id already logged in”);

      return $user;
}
```

Commands are detached from result processing

- Results array is ordered in command-order
- Adding a new command changes index of all results which follow
- Difficult to associate commands with their results
- $results[3][0] is not friendly
- Redis data types don’t necessarily correspond to language data types
- string(“0”) is not FALSE, NULL, nil, undefined, or even an integer in all languages

Even if the code is all contained in a single function,
it’s difficult to match the Redis command with its result(s)

The full function

```
function open_user_session($redis, $id, $emoji_id, $notifications) {
        $pipe = $redis->pipeline();

        $pipe->get(“user:$id:name”);
        $pipe->get(“user:$id:member_of”);

        $pipe->hmget(“emoji:$emoji_id”, “tag”, “name”, “image”);

        $today = date(“Y-m-d”);
        $pipe->incr(“logins:$today”);

        $pipe->set(“session:$id”, “EX”, 600, “NX”);

        $pipe->sadd(“members:logged_in”, $id);

        $results = $pipe->execute();

        $user = new User($results[0]);
        $user->setMembershipObject(Membership::fromString($results[1]));

        $emoji = new Emoji($results[2][1], $results[2][0]);
        $emoji->setImage(base64_decode($results[2][2]));
        $user->setEmoji($emoji);

        StatsD::gauge(‘login_count’, (int) $results[3]);

        if ($results[4] != null) $notifications->sessionExtended($id);
        if ($results[5] > 0) Syslog::notice(“$id already logged in”);

        return $user;
}
```

In a clean world, the do-everything function would be broken up into functions, each bite-sized Redis operation(s).

The hard reality:

Production-scale requirements often require packing operations in pipelines.

<u>The hard reality:</u>

Production-scale requirements often create balls of code mud.

Promises & futures with Redis

Promises & futures

First developed in the 1970s

“Rediscovered” in the 2000s for Web

Sometimes called “deferred programming”

The Big Concept:

Bind a costly scheduled operation
with its resulting value(s) upon completion

Promises & futures

Here’s how I’m using these terms:

A **Promise** object schedules the operation.

The Promise returns a placeholder object for each operation.

The object “receives” the result when the operation completes.

That object is a **Future**.

Promises & futures with Redis

Going back to a key concept:

Redis pipelines and transactions “schedule” work to be performed later.

```
$pipeline = $redis->pipeline();    // no network I/O
$pipeline->get(“user:$id:name”);   // no network I/O
$pipeline->get(“user:$id:email”);  // no network I/O
$pipeline->incr(“user:$id:logins”);// no network I/O
$result = $pipeline->execute();    // sends GET/GET/INCR, recvs name, email, login
```

This matches perfectly with the promises/futures approach

Promises & futures with Redis

Your Redis client might already have support!

A little searching turned up libraries for NodeJS & Scala

I’m sure there’s more—this is not an exotic concept.

Deferred:

Promises & futures for PHP/Predis

Deferred: Promises & futures for PHP/Predis

Programming with Deferred *almost* looks like coding with Predis:

```
$redis = new Predis\Client();
$promises = new Deferred\PromisesPipeline($redis);

$get_name = $promises->get(“user:$id:name”);             // returns Deferred\Future
$hgetall_emoji = $promises->hgetall(“emoji:$emoji_id”); // returns Deferred\Future

$promises->execute(); // network I/O happens here

$name = $get_name->value();       // string
$emoji = $hgetall_emoji->value(); // HGETALL array
```

Scheduling commands with Deferred

Redis commands return placeholder Future objects:

```
$get_name = $promises->get(“user:$id:name”);             // Deferred\Future
$hgetall_emoji = $promises->hgetall(“emoji:$emoji_id”); // Deferred\Future
```

Futures hold no value until the Promises object is executed.
In promises/futures parlance, the Futures are *unfulfilled*.

Fulfilling operations with Deferred

Executing a Promises object *fulfills* the Future objects:

```
$promises->execute();                 // network I/O happens here (“fulfilled”)

$name = $get_name->value();           // string
$emoji = $hgetall_emoji->value(); // HGETALL array
```

Once fulfilled, the resulting value may be retrieved with the `value()` method.

Transforming results

Often we want to normalize or deserialize Redis results. Futures allows for *transformations*:

```
// Deferred\Future
$username = $promises->get(“user:$id:name”);

// reverse the user’s name upon fulfillment
$username->transform(“strrev”); // strrev() is a PHP function

// network I/O -- strrev() is executed here
$promises->execute();

// user’s name backwards
echo $username->value();
```

Transformations: A more interesting example

Use transformations to fetch *objects* instead of arrays of strings:

```
// $emoji is the Deferred\Future
$emoji = $promises->hmget(“emoji:$id”, “name”, “tag”, “image”);

// $hmget is an array of values
$emoji->transform(function ($hmget) {
      $e = new Emoji($hmget[0], $hmget[1]);
      $e->load_image(base64_decode($hmget[2]));

      return $e; // returns an initialized Emoji object
});

$promises->execute();   // network I/O -- transformation happens here

return $emoji->value(); // returns the initialized Emoji object
```

Stacking transformations

Multiple transformations may be registered. They are processed in order:

```
function emoji_image_binary($promises, $id) {
      $image_future = $promises->hget(“emoji:$id”, “image”);

      $image_future->transform(function ($hget) {
            return base64_decode($hget); // this transformation is first
      });

      return $image_future;
}

$image_md5_future = emoji_image_binary($promises, ‘happy-face’);
$image_md5_future->transform(function ($bin) {
      return md5sum($bin); // this transformation is second
});
```

Make HMGET look like HGETALL

In Predis, HGETALL returns a keyed array (or a map):

```
$redis->hgetall(“profile:jim”) // [ “name” => “Jim”, “emoji” => “:elephant:” ]
```

while HMGET returns an indexed array:

```
$redis->hmget(“profile:jim”, “name”, “emoji”) // [ “Jim”, “:elephant:” ]
```

If you have a function which uses HMGET in one code path and HGETALL in another, transformations can smooth out the difference.

Make HMGET look like HGETALL

```
function getProfile($promises, $username, array $fields = null) {
    if (!$fields) {
        $future = $promises->hgetall(“profile:$username”);
    } else {
        $future = $promises->hmget(“profile:$username”, $fields);
        $future->transform(function ($hmget) use ($fields) {
            return array_combine($fields, $hmget);
        });
    }

    return $future;
}
```

Binding

Binding allows for transferring the final result to a PHP variable:

```
$username = null;

$future = $promises->get(“user:$id:name”);
$future->bind($username);

// more conveniently (identical to prior 2 lines)
$promises->get(“user:$id:name”)->bind($username);

$promises->execute(); // $username is now set to a string

return $username;
```

Transformations + binding

```
$profile = null;

// get raw user profile data with HGETALL
$future = $promises->hgetall(“user:$id:profile”);
$future->transform(function ($hgetall) {
      // initialize UserProfile with HGETALL results
      return new UserProfile($hgetall[“name”], $hgetall[“location”], $hgetall[“email”]);
});
// bind UserProfile object to $profile variable
$future->bind($profile);
// fulfill
$promises->execute();
```

Transformations are always done first, then binding

Reducing

Multiple Futures may be *reduced* to a single Future:

```
$registered = $promises->sismember(“users:registered”, $userid);
$validated =  $promises->sismember(“users:validated”,  $userid);
$confirmed =  $promises->sismember(“users:confirmed”,  $userid);

// reduce all three SISMEMBERs to a single Future
$is_ready = $promises->reduce($registered, $validated, $confirmed); // returns a \Deferred\Future
$is_ready->transform(function (array $results) {
      return $results[0] && $results[1] && $results[2];
});

$promises->execute();

return $is_ready->value(); // returns boolean
```

Putting it all together

The full function

```
function open_user_session($redis, $id, $emoji_id, $notifications) {
        $pipe = $redis->pipeline();

        $pipe->get(“user:$id:name”);
        $pipe->get(“user:$id:member_of”);

        $pipe->hmget(“emoji:$emoji_id”, “tag”, “name”, “image”);

        $today = date(“Y-m-d”);
        $pipe->incr(“logins:$today”);

        $pipe->set(“session:$id”, “EX”, 600, “NX”);

        $pipe->sadd(“members:logged_in”, $id);

        $results = $pipe->execute();

        $user = new User($results[0]);
        $user->set_membership_object(Membership::from_string($results[1]));

        $emoji = new Emoji($results[2][1], $results[2][0]);
        $emoji->set_image(base64_decode($results[2][2]));
        $user->set_emoji($emoji);

        StatsD::gauge(‘login_count’, (int) $results[3]);

        if ($results[4] != null) $notifications->session_extended($id);
        if ($results[5] > 0) Syslog::notice(“$id already logged in”);

        return $user;
}
```

class User

```
// Returns \Deferred\Future for User instance
static function getFuture($promises, $id)
{
     $name = $promises->get(“user:$id:name”)
     $member_of = $promises->get(“user:$id:member_of”);

     $user_future = $promises->reduce($name, $member_of)->transform($results) {
          $user = new User($results[0]);
          $user->set_membership_object(Membership::from_string($results[1]);

          return $user;
     });

     return $user_future;
}
```

class Emoji

```
// Returns Deferred\Future for Emoji instance
static function getFuture($promises, $emoji_id)
{
    $emoji_future = $promises->hmget(“emoji:$emoji_id”,
                                     “name”, “tag”, “image”);
    $emoji_future->transform(function ($hmget) {
        $emoji = new Emoji($hmget[0], $hmget[1]);
        $emoji->set_image(base64_decode($hmget[2]);

        return $emoji;
    });

    return $emoji_future;
}
```

class Session

```
// Starts the session on the Redis cache and notifies if extended
function startSession($promises, $userid, $notifications)
{
    $set_future = $promises->set(“session:$userid”, “EX”, 600, “NX”);

    $set_future->onFulfilled(function ($set) use ($userid, $notifications) {
        if ($set !== null)
            $notifications->notify_session_extended($userid);
    });

    return $set_future;
}
```

The full function (revised)

```
function open_user_session($redis, $userid, $emoji_id, $notifications) {

     $promises = new \Deferred\PromisesPipeline($redis);

     $user = null;
     User::getFuture($promises, $userid)->bind($user);

     Emoji::getFuture($promises, $emoji_id)->onFulfilled(function ($emoji) use (&$user) {
          $user->set_emoji($emoji);
     });

     Session::create()->startSession($promises, $userid, $notifications);
     LoginTracker::increment($promises);
     Members::setLoggedIn($promises, $userid);

     $promises->execute();

     return $user;
```

“Show me the code.”

https://github.com/internetarchive/deferred

Questions?

EVERYWHERE.

Thank You

redisconf18

#Redis #RedisConf

presented by:

redislabs
home of redis