Related commands

- DISCARD
- EXEC
- MULTI
- UNWATCH
- WATCH

Transactions

MULTI, EXEC, DISCARD and WATCH are the foundation of transactions in Redis. They allow the execution of a group of commands in a single step, with two important guarantees:

- All the commands in a transaction are serialized and executed sequentially. It can never happen that a request issued by another client is served **in the middle** of the execution of a Redis transaction. This guarantees that the commands are executed as a single isolated operation.
- Either all of the commands or none are processed, so a Redis transaction is also atomic. The EXEC command triggers the execution of all the commands in the transaction, so if a client loses the connection to the server in the context of a transaction before calling the EXEC command none of the operations are performed, instead if the EXEC command is called, all the operations are performed. When using the append-only file Redis makes sure to use a single write(2) syscall to write the transaction on disk. However if the Redis server crashes or is killed by the system administrator in some hard way it is possible that only a partial number of operations are registered. Redis will detect this condition at restart, and will exit with an error. Using the `redis-check-aof` tool it is possible to fix the append only file that will remove the partial transaction so that the server can start again.

Starting with version 2.2, Redis allows for an extra guarantee to the above two, in the form of optimistic locking in a way very similar to a check-and-set (CAS) operation. This is documented later on this page.

Usage

A Redis transaction is entered using the MULTI command. The command always replies with OK. At this point the user can issue multiple commands. Instead of executing these commands, Redis will queue them. All the commands are executed once EXEC is called.

Calling DISCARD instead will flush the transaction queue and will exit the transaction.

The following example increments keys `foo` and `bar` atomically.

```
> MULTI
OK
> INCR foo
QUEUED
> INCR bar
QUEUED
> EXEC
1) (integer) 1
2) (integer) 1
```

As it is possible to see from the session above, EXEC returns an array of replies, where every element is the reply of a single command in the transaction, in the same order the commands were issued.

When a Redis connection is in the context of a MULTI request, all commands will reply with the string `QUEUED` (sent as a Status Reply from the point of view of the Redis protocol). A queued command is simply scheduled for execution when EXEC is called.

Errors inside a transaction

During a transaction it is possible to encounter two kind of command errors:

- A command may fail to be queued, so there may be an error before EXEC is called. For instance the command may be syntactically wrong (wrong number of arguments, wrong command name, ...), or there may be some critical condition like an out of memory condition (if the server is configured to have a memory limit using the `maxmemory` directive).
- A command may fail *after* EXEC is called, for instance since we performed an operation against a key with the wrong value (like calling a list operation against a string value).

Clients used to sense the first kind of errors, happening before the EXEC call, by checking the return value of the queued command: if the command replies with QUEUED it was queued correctly, otherwise Redis returns an error. If there is an error while queueing a command, most clients will abort the transaction discarding it.

However starting with Redis 2.6.5, the server will remember that there was an error during the accumulation of commands, and will refuse to execute the transaction returning also an error during EXEC, and discarding the transaction automatically.

Before Redis 2.6.5 the behavior was to execute the transaction with just the subset of commands queued successfully in case the client called EXEC regardless of previous errors. The new behavior makes it much more simple to mix transactions with pipelining, so that the whole transaction can be sent at once, reading all the replies later at once.

Errors happening *after* EXEC instead are not handled in a special way: all the other commands will be executed even if some command fails during the transaction.

This is more clear on the protocol level. In the following example one command will fail when executed even if the syntax is right:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
MULTI
+OK
SET a abc
+QUEUED
LPOP a
+QUEUED
EXEC
*2
+OK
-ERR Operation against a key holding the wrong kind of value
```

EXEC returned two-element Bulk string reply where one is an OK code and the other an -ERR reply. It's up to the client library to find a sensible way to provide the error to the user.

It's important to note that **even when a command fails, all the other commands in the queue are processed** – Redis will *not* stop the processing of commands.

Another example, again using the wire protocol with `telnet`, shows how syntax errors are reported ASAP instead:

```
MULTI
+OK
INCR a b c
-ERR wrong number of arguments for 'incr' command
```

This time due to the syntax error the bad INCR command is not queued at all.

Why Redis does not support roll backs?

If you have a relational databases background, the fact that Redis commands can fail during a transaction, but still Redis will execute the rest of the transaction instead of rolling back, may look odd to you.

However there are good opinions for this behavior:

- Redis commands can fail only if called with a wrong syntax (and the problem is not detectable during the command queueing), or against keys holding the wrong data type: this means that in practical terms a failing command is the result of a programming errors, and a kind of error that is very likely to be detected during development, and not in production.
- Redis is internally simplified and faster because it does not need the ability to roll back.

An argument against Redis point of view is that bugs happen, however it should be noted that in general the roll back does not save you from programming errors. For instance if a query increments a key by 2 instead of 1, or increments the wrong key, there is no way for a rollback mechanism to help. Given that no one can save the programmer from his or her errors, and that the kind of errors required for a Redis command to fail are unlikely to enter in production, we selected the simpler and faster approach of not supporting roll backs on errors.

Discarding the command queue

DISCARD can be used in order to abort a transaction. In this case, no commands are executed and the state of the connection is restored to normal.

```
> SET foo 1
OK
> MULTI
OK
> INCR foo
QUEUED
> DISCARD
OK
> GET foo
"1"
```

Optimistic locking using check-and-set

WATCH is used to provide a check-and-set (CAS) behavior to Redis transactions.

`WATCH`ed keys are monitored in order to detect changes against them. If at least one watched key is modified before the EXEC command, the whole transaction aborts, and EXEC returns a Null reply to notify that the transaction failed.

For example, imagine we have the need to atomically increment the value of a key by 1 (let's suppose Redis doesn't have INCR).

The first try may be the following:

```
val = GET mykey
val = val + 1
SET mykey $val
```

This will work reliably only if we have a single client performing the operation in a given time. If multiple clients try to increment the key at about the same time there will be a race condition. For instance, client A and B will read the old value, for instance, 10. The value will be incremented to 11 by both the clients, and finally SET as the value of the key. So the final value will be 11 instead of 12.

Thanks to WATCH we are able to model the problem very well:

```
WATCH mykey
val = GET mykey
val = val + 1
MULTI
SET mykey $val
EXEC
```

Using the above code, if there are race conditions and another client modifies the result of `val` in the time between our call to WATCH and our call to EXEC, the transaction will fail.

We just have to repeat the operation hoping this time we'll not get a new race. This form of locking is called *optimistic locking* and is a very powerful form of locking. In many use cases, multiple clients will be accessing different keys, so collisions are unlikely – usually there's no need to repeat the operation.

WATCH explained

So what is WATCH really about? It is a command that will make the EXEC conditional: we are asking Redis to perform the transaction only if none of the `WATCH`ed keys were modified. (But they might be changed by the same client inside the transaction without aborting it. More on this.) Otherwise the transaction is not entered at all. (Note that if you WATCH a volatile key and Redis expires the key after you `WATCH`ed it, EXEC will still work. More on this.)

WATCH can be called multiple times. Simply all the WATCH calls will have the effects to watch for changes starting from the call, up to the moment EXEC is called. You can also send any number of keys to a single WATCH call.

When EXEC is called, all keys are `UNWATCH`ed, regardless of whether the transaction was aborted or not. Also when a client connection is closed, everything gets `UNWATCH`ed.

It is also possible to use the UNWATCH command (without arguments) in order to flush all the watched keys. Sometimes this is useful as we optimistically lock a few keys, since possibly we need to perform a transaction to alter those keys, but after reading the current content of the keys we don't want to proceed. When this happens we just call UNWATCH so that the connection can already be used freely for new transactions.

Using WATCH to implement ZPOP

A good example to illustrate how WATCH can be used to create new atomic operations otherwise not supported by Redis is to implement ZPOP (ZPOPMIN, ZPOPMAX and their blocking variants have only been added in version 5.0), that is a command that pops the element with the lower score from a sorted set in an atomic way. This is the simplest implementation:

```
WATCH zset
element = ZRANGE zset 0 0
MULTI
ZREM zset element
EXEC
```

If EXEC fails (i.e. returns a Null reply) we just repeat the operation.

Redis scripting and transactions

A Redis script is transactional by definition, so everything you can do with a Redis transaction, you can also do with a script, and usually the script will be both simpler and faster.

This duplication is due to the fact that scripting was introduced in Redis 2.6 while transactions already existed long before. However we are unlikely to remove the support for transactions in the short-term because it seems semantically opportune that even without resorting to Redis scripting it is still possible to avoid race conditions, especially since the implementation complexity of Redis transactions is minimal.

However it is not impossible that in a non immediate future we'll see that the whole user base is just using scripts. If this happens we may deprecate and finally remove transactions.
