Transacciones en Redis

Tiempo estimado: 15min

Una vez tenemos una base teórica sólida sobre las transacciones, ya podemos pasar a ver más detenidamente cómo las implementa Redis.

Primero, presentamos las características transaccionales implementadas por Redis como, por ejemplo, los programas en serie y las delimitaciones explícitas. A continuación, analizamos cómo se garantizan las propiedades ACID en sistemas Redis. Finalmente, se hace hincapié en cómo procesa el motor los *scripts* Lua.

Al finalizar la lección, el estudiante sabrá:

- Qué características transaccionales implementa Redis.
- Por qué implementa Redis los programas en serie, no así los intercalados.
- Cómo delimitar las transacciones en Redis.
- Cómo se proporcionan las características ACID en Redis.

Introducción

Generalmente, los sistemas de bases de datos NoSQL implementan transacciones ligeras. Redis no es una excepción. Sus principales características son:

- Utiliza programas en serie. No es posible la ejecución de programas intercalados.
- Implementa delimitación explícita.
- No permite configurar nivel de aislamiento para cada transacción.
- Permite configurar registro de transacciones, mediante AOF.
- Todas las transacciones son de L/E. No es posible indicar transacciones de sólo lectura, aunque una transacción de L/E puede ejecutar sin problemas sólo operaciones de lectura.

Programas en serie

Recordemos que un programa (*schedule*) es un plan para ejecutar las operaciones de varias transacciones. Se distingue entre programas en serie e intercalados. Redis sólo soporta los programas en serie (*serial schedules*), aquellos que ejecutan una única transacción cada vez. Hasta que no termina la transacción en curso, no se pasa a la siguiente.

Los programas en serie se implementan mucho más fácilmente que los intercalados. Pero tienen como inconveniente que reducen considerablemente el rendimiento y la concurrencia, pues las transacciones no ceden los recursos hasta que han finalizado. Como Redis es un motor de bases de datos en memoria, los accesos a los datos son muy rápidos. Extraordinariamente rápidos. Más allá del registro de transacciones o las instantáneas, no se accede a datos de disco. Como las consultas a bases de datos clave-valor son muy directas, no hay mucho procesamiento adicional, el rendimiento del sistema es muy bueno, a pesar del bajo nivel de concurrencia permitido.

Delimitación de transacciones

En Redis, las transacciones se deben delimitar explícitamente mediante comandos específicos. Recordemos que en las delimitaciones explícitas, hay que marcar el inicio y el fin de las transacciones.

En el caso de Redis, el inicio se marca con el comando MULTI:

```
MULTI
```

Y el fin de la transacción mediante EXEC o DISCARD:

EXEC
DISCARD

El procesamiento de las transacciones en Redis es diferente al de otros motores de bases de datos. No se marca el inicio y, a continuación, va ejecutando comando a comando. Redis no comienza la ejecución hasta que tiene *todos* los comandos que forman la transacción. Con MULTI, lo que hacemos es notificar a la instancia que vamos a comenzar a proporcionar, uno a uno, la secuencia de comandos de una nueva transacción. Por su parte, mediante EXEC lo que hacemos es notificar que hemos alcanzado el fin de la secuencia y, además, deseamos pase a ejecutarlos como una transacción.

Si por cualquier cosa deseamos descartar los comandos, es decir, cancelar la recopilación de la secuencia de operaciones de la transacción, usaremos DISCARD. Es importante tener claro que este comando descarta o anula la transacción, pero no deshace ningún cambio, básicamente porque no se ha ejecutado nada.

Veamos un ejemplo ilustrativo para asentar mejor lo que acabamos de ver:

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> SET band:1:name "The National"
QUEUED
127.0.0.1:6379> SET band:1:website "americanmary.com"
QUEUED
127.0.0.1:6379> GET band:1:name
QUEUED
127.0.0.1:6379> EXEC
1) OK
2) OK
3) "The National"
127.0.0.1:6379>
```

Cuando se ejecuta MULTI, los comandos se van añadiendo a la transacción. Los cuales se procesarán cuando ejecutemos EXEC. Observe que cada vez que se ejecuta un comando, lo único que estamos haciendo es añadirlo a la transacción, no se ejecuta. Esto se puede observar con el resultado devuelto por los comandos: QUEUED.

Una vez ejecutamos el comando EXEC, el motor cierra la transacción y pasa a ejecutarla. La salida del comando es una lista con los resultados de la ejecución de cada una de sus operaciones. En el ejemplo, los dos primeros comandos son SET y finalizan en OK, indicando que se ejecutaron correctamente. El tercer resultado es la salida del comando GET.

Veamos otro ejemplo. Ahora, una delimitación de comandos descartada, por lo que ninguno de los comandos introducidos se ejecutará:

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> SET band:2:name "Echo and the Bunnymen"
QUEUED
127.0.0.1:6379> SET band:2:website "bunnymen.com"
QUEUED
127.0.0.1:6379> DISCARD
OK
127.0.0.1:6379> GET band:2:name
(nil)
127.0.0.1:6379>
```

Propiedades ACID

Redis no implementa todas las propiedades ACID de las transacciones. Incluso algunas tienen una implementación un poco particular. Vamos a analizarlas cada una detalladamente.

Atomicidad

La atomicidad (*atomicity*) es la propiedad mediante la cual el motor garantiza que ejecutará todas las operaciones de la transacción o ninguna. Si alguna operación falla, la transacción fallará en su conjunto y cualquier cambio realizado se deshará, dejando los datos modificados como estaban antes del comienzo de la transacción.

En Redis, al igual que en otros motores de bases de datos NoSQL, la atomicidad se implementa a nivel

de operación. Individualmente. Conceptualmente, la atomicidad requiere que se haga a nivel de la unidad lógica que forman las operaciones en su conjunto. Así pues, ¿qué pasa si una operación de la transacción falla? Veámoslo mediante un ejemplo:

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> SET band:2:name "Echo and the Bunnymen"
QUEUED
127.0.0.1:6379> SET band:2:clicks cero
QUEUED
127.0.0.1:6379> INCR band:2:clicks
QUEUED
127.0.0.1:6379> SET band:2:website "bunnymen.com"
QUEUED
127.0.0.1:6379> EXEC
1) OK
2) OK
3) (error) ERR value is not an integer or out of range
4) OK
127.0.0.1:6379> GET band:2:name
"Echo and the Bunnymen"
127.0.0.1:6379> GET band:2:website
"bunnymen.com"
127.0.0.1:6379> GET band:2:clicks
"cero"
127.0.0.1:6379>
```

Como se puede observar, la atomicidad no es a nivel de transacción en su conjunto, sino a nivel de operación. Reiteramos que esto no es específico de Redis. Muchos motores de bases de datos NoSQL utilizan un nivel de atomicidad relajado.

Cuando una operación de la transacción falla, se propaga un error para esa operación, pero se continúa con el resto de la secuencia de operaciones. La transacción no se detiene. Por otra parte, no se deshace ningún cambio realizado hasta ese momento, ya que se sigue con su ejecución. En nuestro caso, añadimos la clave band:2:clicks con el valor textual cero, para que el intento de incremento de su valor falle. Pero eso es todo. Falla, pero continúa con la ejecución del resto de operaciones. De ahí que la clave band:2:website, creada por una operación posterior a la fallida, se haya realizado sin problemas.

Consistencia

La consistencia (*consistency*) o conservación de consistencia (*consistency preservation*) hace referencia a que la base de datos debe quedar en un estado consistente tras la ejecución de las transacciones. Recordemos que, según el fabricante, la consistencia la puede implementar parcial o totalmente el motor de bases de datos. Principalmente mediante el uso de restricciones de integridad. Toda aquella restricción que no implementa el motor, recae en el desarrollador.

En Redis, la consistencia recae íntegramente en los desarrolladores. Es otro aspecto muy común en los sistemas de bases de datos NoSQL. De esta manera van más rápidos, pero cargan a los desarrolladores con tareas adicionales.

Aislamiento

El aislamiento (*isolation*) está relacionado principalmente con dos cosas: la ejecución concurrente de transacciones y el acceso a los datos modificados por otras transacciones concurrentes.

Tal como vimos anteriormente, Redis utiliza programas en serie, por lo que como máximo sólo habrá una transacción en ejecución. Hasta que la transacción en curso no finalice, el motor no pasará a la siguiente. Así pues, debe quedar bastante claro que el acceso a los datos es siempre el esperado, no hay efectos colaterales. Todo lo leído procede de una transacción anterior ya finalizada o de la transacción actual. Como sólo una transacción puede estar modificando datos, no habrá ningún efecto colateral.

Durabilidad

Finalmente, tenemos la durabilidad (*durability*). Esta propiedad está relacionada con los cambios

confirmados. Cuando una transacción ha finalizado, el motor debe garantizar que sus cambios no se perderán, pase lo que pase, aun si se produce una caída inesperada del sistema. Recordemos que esto se consigue mediante el uso del registro de transacciones.

Concretamente, en Redis, la durabilidad se consigue si los DBAs configuran el registro mediante AOF. Este aspecto queda fuera del alcance de este curso y se trata en el curso Administración de Redis. Aunque hay que decir que de manera predeterminada, el registro de transacciones se encuentra activado. Cuando un proyecto requiere que las transacciones cumplan la propiedad de durabilidad, los desarrolladores deben confirmar con los DBAs que se ha activado el registro de transacciones. Y ellos deben saber que es necesario, para evitar que lo desactiven.

Scripts Lua

Teniendo en cuenta que un *script* Lua es una secuencia de operaciones entre las cuales puede haber operaciones de acceso a la base de datos, es importante tener bien claro que los *scripts* se ejecutan como transacciones. Sólo un *script* puede estar ejecutándose cada vez. Los comandos EVAL y EVALSHA actúan como delimitadores. Marcan el inicio y el fin de la transacción, la cual estará formada por las operaciones Lua.

¿Qué ocurre si el *script* falla? Lo visto hasta ahora. Las operaciones contra la base de datos serán atómicas a nivel de operación, no así del *script* en su conjunto. Sólo hay una excepción. Si se produce un fallo, el *script* acabará y no seguirá ejecutándose.

Otra pregunta que puede surgir es: ¿puede haber en ejecución una transacción Lua y otra delimitada por MULTI y EXEC? La respuesta es un rotundo *no*. Los scripts de Lua se ejecutan como transacciones y en Redis sólo puede haber una transacción en ejecución. Da igual si es un *script* o una transacción delimitada por MULTI y EXEC. Es más, si el *script* tarda mucho, Redis puede propagar un error en cualquier otro intento de transacción, similar a *Redis is busy running a script. You can only call SCRIPT KILL or SHUTDOWN NOSAVE*. Cuando un EXEC acaba propagando este error, hay que descartar la transacción mediante DISCARD para poder introducirla de nuevo o bien introducir una distinta mediante la misma conexión.

