Replicación en MongoDB

Desde las primeras lecciones, hemos estado usando un servidor independiente, un único servidor mongod. Es una forma fácil de empezar, pero peligrosa de ejecutar en producción. ¿Qué pasa si su servidor falla o deja de estar disponible? Su base de datos no estará disponible al menos por un tiempo. Si hay problemas con el hardware, es posible que tengas que mover tus datos a otra máquina. En el peor de los casos, los problemas de disco o de red podrían dejar datos corruptos o inaccesibles.

¿Que es la replicación?

La replicación es una forma de mantener copias idénticas de sus datos en múltiples servidores y se recomienda para todas las implementaciones de producción. La replicación mantiene su aplicación en ejecución y sus datos seguros, incluso si algo le sucede a uno o más de sus servidores.

Con MongoDB, la replicación se configura creando un conjunto de réplicas. Un conjunto de réplicas es un grupo de servidores con un servidor principal (el servidor que realiza escrituras) y varios servidores secundarios (que guardan copias de los datos del principal). Si el servidor primario fracasa, los secundarios pueden elegir un nuevo primario entre ellos.

Si está utilizando la replicación y un servidor deja de funcionar, aún puede acceder a sus datos desde los otros servidores del conjunto. Si los datos de un servidor están dañados o son inaccesibles, puede hacer una nueva copia de los datos de uno de los otros miembros del conjunto.

Estas notas presentan los conjuntos de réplicas y explica cómo configurar la replicación en su sistema. Si está menos interesado en la mecánica de replicación y simplemente desea crear un conjunto de réplicas para pruebas/desarrollo o producción, utilice la solución en la nube de MongoDB, MongoDB Atlas. Es fácil de usar y ofrece una opción de nivel gratuito para experimentar. Alternativamente, para administrar clústeres de MongoDB en su propia infraestructura, puede utilizar Ops Manager.

MongoDB Ops Manager es una plataforma de gestión para MongoDB que facilita la administración de MongoDB en escala para operaciones complejas. Proporciona herramientas para monitorear, respaldar y configurar despliegues de MongoDB, automatizando tareas rutinarias y mejorando la eficiencia operativa. Permite a los administradores de bases de datos supervisar el estado y rendimiento de sus bases de datos, configurar alertas, realizar backups y restauraciones, y mucho más, todo desde una interfaz centralizada.

Configuración de un conjunto de réplicas, primera parte

Procedemos a configurar un conjunto de réplicas de tres nodos en una sola máquina para comenzar a experimentar con la mecánica del conjunto de réplicas. Este es el tipo de configuración que puede programar solo para configurar y ejecutar una réplica y luego modificarla con comandos administrativos en el shell mongo o simular particiones de red o fallas del servidor para comprender mejor cómo MongoDB maneja la alta disponibilidad y la recuperación ante desastres. En producción, siempre debe utilizar un conjunto de réplicas y asignar un host dedicado a cada miembro para evitar la contención de recursos y proporcionar aislamiento contra fallas del servidor. Para brindar mayor resiliencia, también debe usar el formato de conexión DNS Seedlist para especificar cómo se conectan sus

aplicaciones a su conjunto de réplicas. La ventaja de usar DNS es que los servidores que alojan a los miembros del conjunto de réplicas de MongoDB se pueden cambiar en rotación sin necesidad de reconfigurar los clientes (específicamente, sus cadenas de conexión).

El formato de conexión DNS Seedlist de MongoDB es una convención que permite usar un prefijo mongodb+srv:// en una cadena de conexión. Este formato simplifica la cadena de conexión especificando un nombre de dominio que, a través de registros DNS SRV y TXT, resuelve a una lista de direcciones de servidores MongoDB. Facilita la gestión de clusters de MongoDB, especialmente en entornos distribuidos, permitiendo actualizaciones y cambios en la configuración del cluster sin necesidad de modificar la cadena de conexión en las aplicaciones cliente.

Dada la variedad de opciones de virtualización y nube disponibles, es casi igual de fácil crear un conjunto de réplicas de prueba con cada miembro en un host dedicado. Proporcionamos un script Vagrant para permitirle experimentar con esta opción.1

Para comenzar con nuestro conjunto de réplicas de prueba, primero creemos directorios de datos separados para cada nodo.

Creamos tres directorios en el directorio data que ya tenemos configurado:

```
mkdir rs1 rs2 rs3
```

Arrancamos tres servidores en tres puertos diferentes y usando los directorios de datos creados previamente:

```
mongod --replSet mdbDefGuide --dbpath c:\data\rs1 --port 27017 --smallfiles --oplogSize 200
mongod --replSet mdbDefGuide --dbpath c:\data\rs2 --port 27018 --smallfiles --oplogSize 200
mongod --replSet mdbDefGuide --dbpath c:\data\rs3 --port 27019 --smallfiles --oplogSize 200
```

Las opciones smallfiles y oplogSize:

--smallfiles: Esta opción modifica el almacenamiento preasignado para archivos de datos. Cuando se utiliza --smallfiles, MongoDB reduce el tamaño de los archivos de datos y el espacio reservado para el journal. Esto es útil en entornos con espacio en disco limitado y no se recomienda para la producción en la mayoría de los casos.

--oplogSize: Este parámetro especifica el tamaño en megabytes del oplog, el log de operaciones, para una instancia de MongoDB en un conjunto de réplicas. El oplog es un registro de todas las operaciones que modifican los datos de la base de datos. Un tamaño más grande del oplog permite un período más extenso de operaciones para la replicación, pero también utiliza más espacio en disco. En tu comando, --oplogSize 200 establece el tamaño del oplog en 200 megabytes.

Las opciones --smallfiles y --oplogSize en versiones anteriores de MongoDB tenían funciones específicas para la configuración del almacenamiento y tamaño del oplog. Sin embargo, en versiones recientes de MongoDB, la opción --smallfiles ha sido eliminada. Esta opción estaba relacionada con la reducción del tamaño de los archivos de datos y del espacio reservado para el journal, especialmente útil en el motor de almacenamiento MMAPv1, que también ha sido eliminado en versiones recientes. Por otro lado, la opción --oplogSize sigue siendo relevante para configurar el tamaño del oplog en megabytes en un conjunto de réplicas, lo que es importante para la replicación de datos. Para obtener detalles específicos sobre la configuración en la versión actual de MongoDB, se recomienda consultar la documentación oficial de MongoDB para la versión específica que estés utilizando.

Una vez ejecutados esos tres comandos, tenemos tres procesos de mongod corriendo en nuestra máquina


```
C:\Users\barba>netstat -ano | findstr :27017
  TCP    127.0.0.1:27017        0.0.0.0:0              LISTENING       33948

C:\Users\barba>netstat -ano | findstr :27018
  TCP    127.0.0.1:27018        0.0.0.0:0              LISTENING       14916

C:\Users\barba>netstat -ano | findstr :27019
  TCP    127.0.0.1:27019        0.0.0.0:0              LISTENING       34516
```

Con el trabajo que hemos realizado hasta ahora, cada mongod aún no sabe que los demás existen. Para informarles unos de otros, necesitamos crear una configuración que enumere a cada uno de los miembros y enviar esta configuración a uno de nuestros procesos mongod. Se encargará de propagar la configuración a los demás miembros.

En una cuarta terminal, el símbolo del sistema de Windows o la ventana de PowerShell, inicie un shell mongo que se conecte a una de las instancias de mongod en ejecución. Puede hacer esto escribiendo el siguiente comando. Con este comando, nos conectaremos al mongod que se ejecuta en el puerto 27017:

`mongosh --port 27017`

Luego, en el shell de mongo, cree un documento de configuración y páselo al método rs.initiate() para iniciar un conjunto de réplicas. Esto iniciará un conjunto de réplicas que contiene tres miembros y propagará la configuración al resto de los mongods para que se forme un conjunto de réplicas:

```
test> rsconf = {
...     _id: "mdbDefGuide",
...     members: [
...       {_id: 0, host: "localhost:27017"},
...       {_id: 1, host: "localhost:27018"},
...       {_id: 2, host: "localhost:27019"}
...     ]
...   }
{
  _id: 'mdbDefGuide',
  members: [
    { _id: 0, host: 'localhost:27017' },
    { _id: 1, host: 'localhost:27018' },
    { _id: 2, host: 'localhost:27019' }
  ]
}
test> rs.initiate(rsconf)
{ ok: 1 }
mdbDefGuide [direct: secondary] test>
```

Hay varias partes importantes de un documento de configuración de conjunto de réplicas. El "_id" de la configuración es el nombre del conjunto de réplicas que pasó en la línea de comando (en este ejemplo, "mdbDefGuide"). Asegúrese de que este nombre coincida exactamente.

La siguiente parte del documento es una serie de miembros del conjunto. Cada uno de estos necesita dos campos: un "_id" que es un número entero y único entre los miembros del conjunto de réplicas, y un nombre de host.

Este documento de configuración es la configuración de su conjunto de réplicas. El miembro que se ejecuta en localhost:27017 analizará la configuración y enviará mensajes a los demás miembros, alertándoles de la nueva configuración. Una vez que todos hayan cargado la configuración, elegirán un primario y comenzarán a manejar lecturas y escrituras.

Si está iniciando un conjunto nuevo, puede enviar la configuración a cualquier miembro del conjunto. Si está comenzando con datos de uno de los miembros, debe enviar la configuración al miembro con datos. No puede iniciar un conjunto de réplicas con datos de más de un miembro.

Una vez iniciado, debería tener un conjunto de réplicas completamente funcional. El conjunto de réplicas debe elegir un principal. Puede ver el estado de un conjunto de réplicas utilizando rs.status(). El resultado de rs.status() le informa bastante sobre el conjunto de réplicas, incluidas varias cosas que aún no hemos cubierto, pero no se preocupe, ¡llegaremos allí! Por ahora, eche un vistazo a la matriz de miembros. Tenga en cuenta que nuestras tres instancias de mongod están enumeradas en esta matriz y que una de ellas, en este caso el mongod que se ejecuta en el puerto 27017, ha sido elegida primaria. Los otros dos son secundarios. Si intenta esto usted mismo, seguramente tendrá valores diferentes para "fecha" y los distintos valores de marca de tiempo en este resultado, pero también puede encontrar que se eligió un mongod diferente como primario (eso está totalmente bien).

Hay varias partes importantes de un documento de configuración de conjunto de réplicas. El "_id" de la configuración es el nombre del conjunto de réplicas que pasó en la línea de comando (en este ejemplo, "mdbDefGuide"). Asegúrese de que este nombre coincida exactamente.

La siguiente parte del documento es una serie de miembros del conjunto. Cada uno de estos necesita dos campos: un "_id" que es un número entero y único entre los miembros del conjunto de réplicas, y un nombre de host.

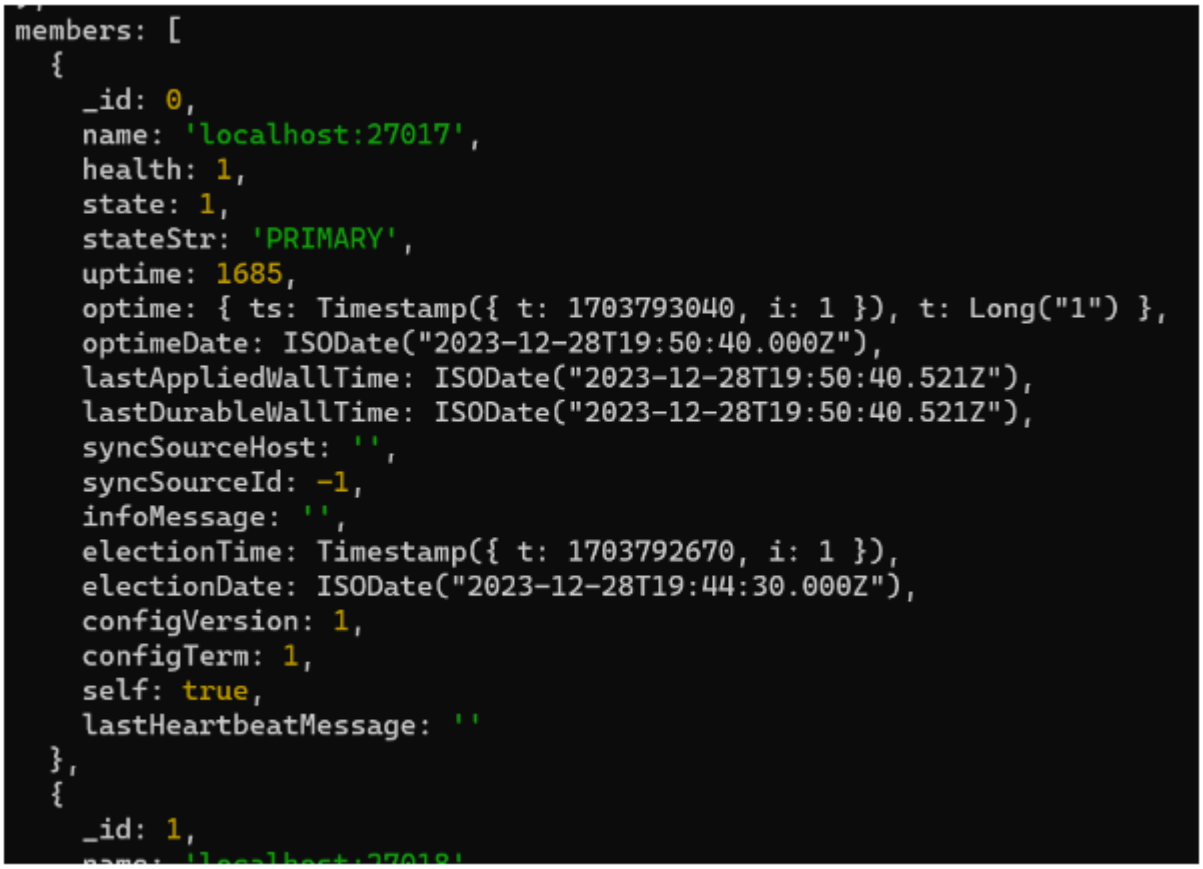
Este documento de configuración es la configuración de su conjunto de réplicas. El miembro que se ejecuta en localhost:27017 analizará la configuración y enviará mensajes a los demás miembros, alertándoles de la nueva configuración. Una vez que todos hayan cargado la configuración, elegirán un primario y comenzarán a manejar lecturas y escrituras.

Si está iniciando un conjunto nuevo, puede enviar la configuración a cualquier miembro del conjunto. Si está comenzando con datos de uno de los miembros, debe enviar la configuración al miembro con datos. No puede iniciar un conjunto de réplicas con datos de más de un miembro.

Una vez iniciado, debería tener un conjunto de réplicas completamente funcional. El conjunto de réplicas debe elegir un principal. Puede ver el estado de un conjunto de réplicas utilizando rs.status(). El resultado de rs.status() le informa bastante sobre el conjunto de réplicas, incluidas varias cosas que aún no hemos cubierto, pero no se preocupe,

¡llegaremos allí! Por ahora, eche un vistazo a la matriz de miembros. Tenga en cuenta que nuestras tres instancias de mongod están enumeradas en esta matriz y que una de ellas, en este caso el mongod que se ejecuta en el puerto 27017, ha sido elegida primaria. Los otros dos son secundarios. Si intenta esto usted mismo, seguramente tendrá valores diferentes para "fecha" y los distintos valores de marca de tiempo en este resultado, pero también puede encontrar que se eligió un mongod diferente como primario (eso está totalmente bien).

Muestro solamente un parte del documento generado por rs.status() para mostrar que el servidor en 27017 ha sido elegido como primario



rs es una variable global que contiene funciones auxiliares de replicación (ejecute rs.help() para ver las funciones auxiliares que expone). Estas funciones casi siempre son simplemente envoltorios de comandos de bases de datos. Por ejemplo, el siguiente comando de base de datos es equivalente a rs.initiate(config):

db.adminCommand({"replSetInitiate" : config})

Con lo anterior, ya hemos configurado nuestro conjunto de réplica. Pasemos a probarlo.

Observando la replicación

Si su conjunto de réplicas eligió mongod en el puerto 27017 como principal, entonces el shell mongo utilizado para iniciar el conjunto de réplicas está actualmente conectado al principal. Debería ver que el prompt del shell cambia a algo como lo siguiente:

mdbDefGuide [direct: primary] test> Esto indica que estamos conectados al primario del conjunto de réplicas que tiene el "_id" "mdbDefGuide". Para simplificar y en aras de la claridad, abreviaremos el indicador del shell mongo a simplemente > en los ejemplos de replicación.

Si su conjunto de réplicas eligió un nodo primario diferente, salga del shell y conéctese al primario especificando el número de puerto correcto en la línea de comando, como lo hicimos cuando iniciamos el shell mongo anteriormente. Por ejemplo, si el puerto principal de su equipo está en el puerto 27018, conéctese usando el siguiente comando:

$ mongosh --puerto 27018

Ahora que está conectado al servidor principal, intente realizar algunas escrituras y vea qué sucede. Primero, inserte 1000 documentos:

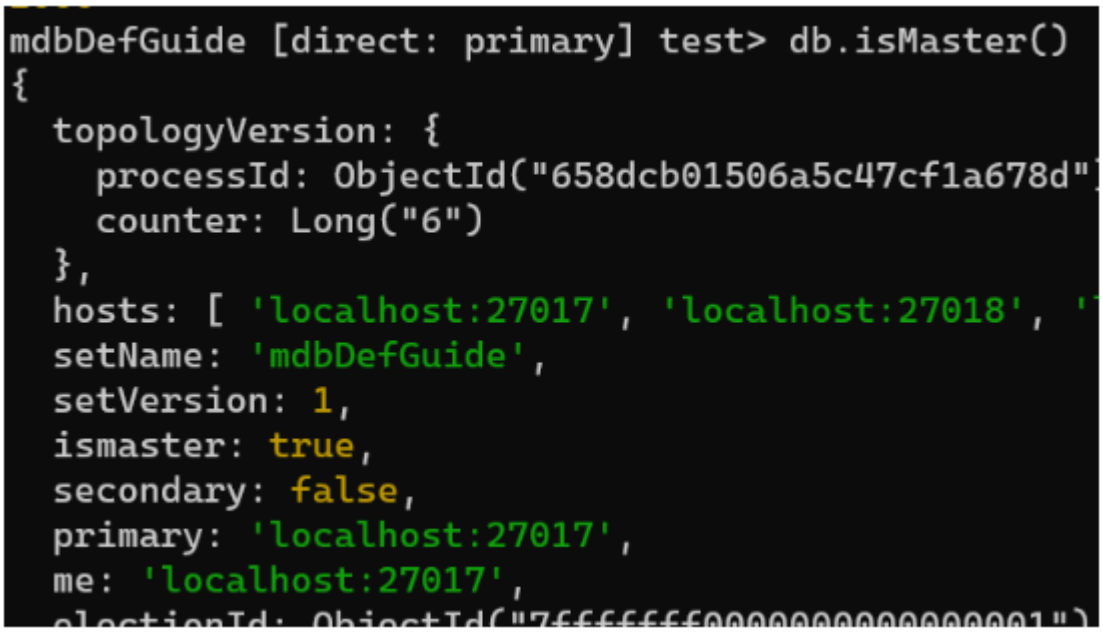
```
mdbDefGuide [direct: primary] test> use test
already on db test
mdbDefGuide [direct: primary] test> for (i=0; i<1000; i++) {db.coll.insert({count: i})}
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or bulkWrite.
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("658dd3f94f685729a68a1281") }
}
mdbDefGuide [direct: primary] test>

mdbDefGuide [direct: primary] test> db.coll.count()
DeprecationWarning: Collection.count() is deprecated. Use countDocuments or estimatedDocumentCount.
1000
mdbDefGuide [direct: primary] test>
```

Nos conectamos a un secundario y vemos si se ejecutó la replicación.

Puede hacer esto saliendo del shell y conectándose usando el número de puerto de uno de los secundarios, pero es fácil adquirir una conexión a uno de los secundarios creando una instancia de un objeto de conexión usando el constructor Mongo dentro del shell que ya está ejecutando.

Primero, use su conexión a la base de datos de prueba en el primario para ejecutar el comando isMaster. Esto le mostrará el estado del conjunto de réplicas, de una forma mucho más concisa que rs.status(). También es un medio conveniente para determinar qué miembro es el principal al escribir código de aplicación o secuencias de comandos:



Si en algún momento se convoca una elección y el mongod al que está conectado se convierte en secundario, puede usar el comando isMaster para determinar qué miembro se ha convertido en primario. El resultado aquí nos dice que localhost:27018 y localhost:27019 son ambos secundarios, por lo que podemos usarlos para nuestros propósitos. Instanciamos una conexión a localhost:27019:

```
secondaryConn = new Mongo("localhost:27019")
connection to localhost:27019
secondaryDB = secondaryConn.getDB("test")
```

test

Ahora, si intentamos realizar una lectura de la colección que se ha replicado en la secundaria, obtendremos un error. Intentemos realizar una búsqueda en esta colección y luego revisar el error y por qué lo obtenemos:

```
secondaryDB.coll.find()
Error: error: {
"operationTime" : Timestamp(1501200089, 1),
"ok" : 0,
"errmsg" : "not master and slaveOk=false",
"code" : 13435,
"codeName" : "NotMasterNoSlaveOk"
```

Los secundarios pueden quedarse atrás del primario (o retrasarse) y no tener las escrituras más actualizadas, por lo que los secundarios rechazarán las solicitudes de lectura de forma predeterminada para evitar que las aplicaciones lean accidentalmente datos obsoletos. Por

lo tanto, si intenta consultar un secundario, recibirá un error que indica que no es el principal. Esto es para proteger su aplicación de conectarse accidentalmente a un secundario y leer datos obsoletos. Para permitir consultas en el secundario, podemos configurar un indicador "Estoy de acuerdo con leer desde secundarios", así:

```
secondaryConn.setSlaveOk()
```

Al hacerlo de esta manera obtenemos un error pues el método setSlaveOK() ya no es válido en las nuevas versiones de MongoDB.

Lo hacemos de esta forma:

```
mdbDefGuide [direct: primary] test>
secondaryDB.getMongo().setReadPref('secondary')
```

Y ahora podemos leer del secundario

```
mdbDefGuide [direct: primary] test> secondaryDB.coll.find().limit(5)
[
{ _id: ObjectId("658dd3ef4f685729a68a0e9a"), count: 0 },
{ _id: ObjectId("658dd3ef4f685729a68a0e9b"), count: 1 },
{ _id: ObjectId("658dd3ef4f685729a68a0e9c"), count: 2 },
{ _id: ObjectId("658dd3ef4f685729a68a0e9d"), count: 3 },
{ _id: ObjectId("658dd3ef4f685729a68a0e9e"), count: 4 }
]
```

Tratemos ahora de escribir a un secundario (en nuestro ejemplo el que escucha por el puerto 27019):

```
mdbDefGuide [direct: primary] test> secondaryDB.coll.insert({"count" : 1001})
Uncaught:
MongoBulkWriteError: not primary
Result: BulkWriteResult {
result: {
ok: 1,
writeErrors: [],
writeConcernErrors: [],
insertedIds: [ { index: 0, _id: ObjectId("658de0db4f685729a68a1282") } ],
nInserted: 0,
nUpserted: 0,
nMatched: 0,
nModified: 0,
nRemoved: 0,
upserted: []
}
}
mdbDefGuide [direct: primary] test> secondaryDB.coll.count()
1000
```

Puedes ver que el secundario no acepta la escritura. Un secundario solo realizará escrituras que obtenga a través de la replicación, no de los clientes.

Automatic failover

Si el primario fracasa, uno de los secundarios será automáticamente elegido primario. Para probar esto, detenga el primario:

db.adminCommand({"shutdown" : 1})

Ejecutamos

secondaryDB.isMaster()

mdbDefGuide [direct: primary] test> db.adminCommand({"shutdown" : 1})

MongoNetworkError: connection 1 to 127.0.0.1:27017 closed

test> db

test

test> secondaryDB.isMaster()

{

topologyVersion: {

processId: ObjectId("658dcb4edf88abfa13f29b07"),

counter: Long("5")

},

hosts: ['localhost:27017', 'localhost:27018', 'localhost:27019'],

setName: 'mdbDefGuide',

setVersion: 1,

ismaster: false,

secondary: true,

primary: 'localhost:27018',

me: 'localhost:27019',

lastWrite: {

opTime: { ts: Timestamp({ t: 1703807055, i: 1 }), t: Long("2") },

lastWriteDate: ISODate("2023-12-28T23:44:15.000Z"),

majorityOpTime: { ts: Timestamp({ t: 1703807055, i: 1 }), t: Long("2") },

majorityWriteDate: ISODate("2023-12-28T23:44:15.000Z")

},

maxBsonObjectSize: 16777216,

maxMessageSizeBytes: 48000000,

maxWriteBatchSize: 100000,

localTime: ISODate("2023-12-28T23:44:24.006Z"),

logicalSessionTimeoutMinutes: 30,

connectionId: 69,

minWireVersion: 0,

maxWireVersion: 17,

readOnly: false,

ok: 1,

'$clusterTime': {

clusterTime: Timestamp({ t: 1703807055, i: 1 }),

signature: {

hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "hex"), 0),

keyId: Long("0")

}

},

operationTime: Timestamp({ t: 1703807055, i: 1 }),

isWritablePrimary: false

}

Y vemos que ahora el servidor primario es el del puerto 27018.

En MongoDB, el método db.isMaster() se utiliza para proporcionar información sobre el rol del servidor MongoDB en el contexto de un conjunto de réplicas o sharding. Cuando ejecutas db.isMaster(), el comando devuelve un documento que incluye detalles como si el servidor es el nodo primario en un conjunto de réplicas, la configuración del conjunto de réplicas actual, si está en modo de sharding, y las direcciones de otros nodos en el conjunto de réplicas. Este método es útil para diagnosticar y entender la topología de tu despliegue de MongoDB.

Arrancamos nuevamente el servidor en la 27017.

Localizamos el shell donde lo arrancamos inicialmente:

mongod --replSet mdbDefGuide --dbpath c:\data\rs1 --port 27017

Ese servidor se incorpora como un nodo secundario.

Creemos una conexión el el shell a ese nodo, tal y como hicimos con el nodo 27019 y observamos su comportamiento como nodo secundario, analizando cada comando.

mdbDefGuide [direct: secondary] test> secondary27017 = new Mongo("localhost:27017")

mongodb://localhost:27017/?directConnection=true&serverSelectionTimeoutMS=2000

mdbDefGuide [direct: secondary] test> secondaryDB27017 = secondary27017.getDB("test")

test

```
mdbDefGuide [direct: secondary] test> show collections

coll

mdbDefGuide [direct: secondary] test> secondaryDB27017.coll.count()

MongoServerError: not primary and secondaryOk=false - consider using
db.getMongo().setReadPref() or readPreference in the connection string

mdbDefGuide [direct: secondary] test>
secondaryDB27017.getMongo().setReadPref('secondary')

mdbDefGuide [direct: secondary] test> secondaryDB27017.coll.count()

1000

mdbDefGuide [direct: secondary] test> secondaryDB27017.coll.insert({"count" : 1001})

Uncaught:

MongoBulkWriteError: not primary

Result: BulkWriteResult {

result: {

ok: 1,

writeErrors: [],

writeConcernErrors: [],

insertedIds: [ { index: 0, _id: ObjectId("658e0f014f685729a68a1283") } ],

nInserted: 0,

nUpserted: 0,

nMatched: 0,

nModified: 0,

nRemoved: 0,

upserted: []

}

}
```

Algunos conceptos para recordar hasta aquí:

Los clientes pueden enviar a un servidor primario las mismas operaciones que podrían enviar a un servidor independiente (lecturas, escrituras, comandos, compilaciones de índices, etc.).

Los clientes no pueden escribir en secundarios.

Los clientes, de forma predeterminada, no pueden leer desde los secundarios. Puede habilitar esto configurando explícitamente una configuración "Sé que estoy leyendo desde un secundario" en la conexión.

Cambiando la configuración del conjunto de réplica

Podemos añadir un servidor en el puerto 27020

Para eso creamos el directorio rs4 en \data y arrancamos un servidor perteneciente al replica set en el puerto 27020

>mongod --replSet mdbDefGuide --dbpath c:\data\rs4 --port 27020

Y ahora añadimos el servidor 27020, usando el método rs.add(....)

mdbDefGuide [direct: secondary] test> rs.add("localhost:27020")

MongoServerError: New config is rejected :: caused by :: replSetReconfig should only be run on a writable PRIMARY. Current state SECONDARY;

mdbDefGuide [direct: secondary] test> db.serverStatus()

{

host: 'DESKTOP-842EKRJ:27019',

version: '6.0.2',

process: 'mongod',

pid: Long("34516"),

Tenemos un error y la razón es que no podemos añadir un nuevo nodo cuando estamos conectados a un servidor secundario desde el shell. El comando db.serverStatus() nos brinda información muy amplia del servidor incluyendo a cual servidor estamos conectados como se ve en parte del resultado de ese comando.

Recordemos que cuando eliminamos el servidor en 27017, el conjunto de réplica eligió al servidor en el puerto 27018 como nodo primario.

Vamos a conectarnos con un shell a ese nodo

mongosh --port 27018

El prompt nos indica que estamos conectados al nodo primario 27018

mdbDefGuide [direct: primary] test>

Y ahora añadimos el nuevo nodo al conjunto de réplica.

rs.add("localhost:27020")

El resultado:

mdbDefGuide [direct: primary] test> rs.add("localhost:27020")

{

ok: 1,

'$clusterTime': {

clusterTime: Timestamp({ t: 1703811093, i: 1 }),

signature: {

hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "hex"), 0),
keyId: Long("0")
}
},
operationTime: Timestamp({ t: 1703811093, i: 1 })
}
mdbDefGuide [direct: primary] test>

Y con db.isMaster() vemos la incorporación del nuevo nodo al conjunto de réplica.

```
mdbDefGuide [direct: primary] test> db.isMaster()
{
topologyVersion: {
processId: ObjectId("658dcb360bdf2488b42225e1"),
counter: Long("9")
},
hosts: [
'localhost:27017',
'localhost:27018',
'localhost:27019',
'localhost:27020'
],
setName: 'mdbDefGuide',
setVersion: 3,
ismaster: true,
secondary: false,
primary: 'localhost:27018',
me: 'localhost:27018',
electionId: ObjectId("7fffffff0000000000000002"),
```

Podemos usar rs.config() para ver toda la configuración del conjunto de réplica. Un ejemplo para ver los hosts que lo componen:

```
mdbDefGuide [direct: primary] test> let config = rs.config()
mdbDefGuide [direct: primary] test> config.members.forEach(member=>print(member.host))
localhost:27017
```

localhost:27018

localhost:27019

localhost:27020

mdbDefGuide [direct: primary] test>

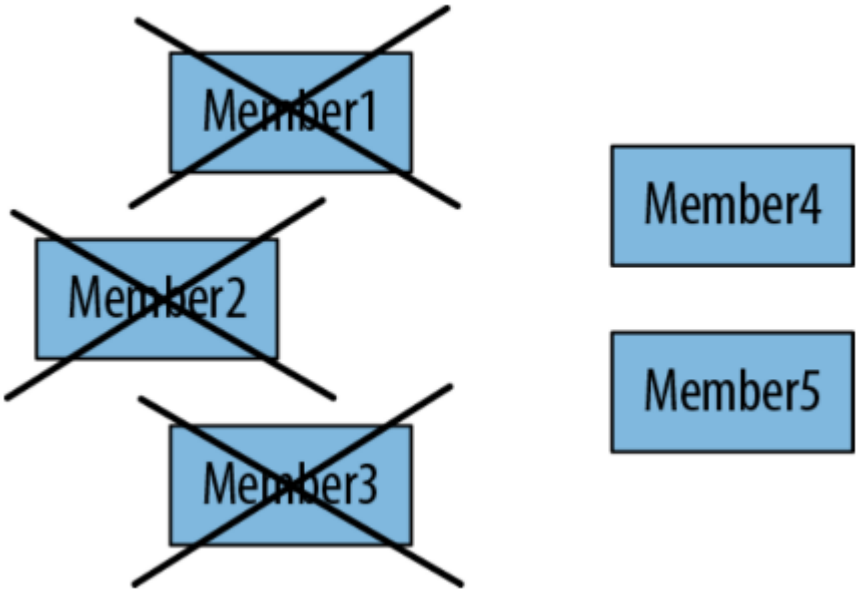
Revisen el objeto config que se obtiene con rs.config() para observar toda la configuración del conjunto de réplica.

Diseño de conjuntos de réplicas

Para planificar tu set, hay ciertos conceptos con los que debes estar familiarizado. Uno de esos conceptos es "Majority": se necesita una mayoría de miembros para elegir un primario, una primario sólo puede seguir siendo primario mientras pueda alcanzar una mayoría, y una escritura es segura cuando se ha replicado a una mayoría. Esta mayoría se define como “más de la mitad de todos los miembros del conjunto”.

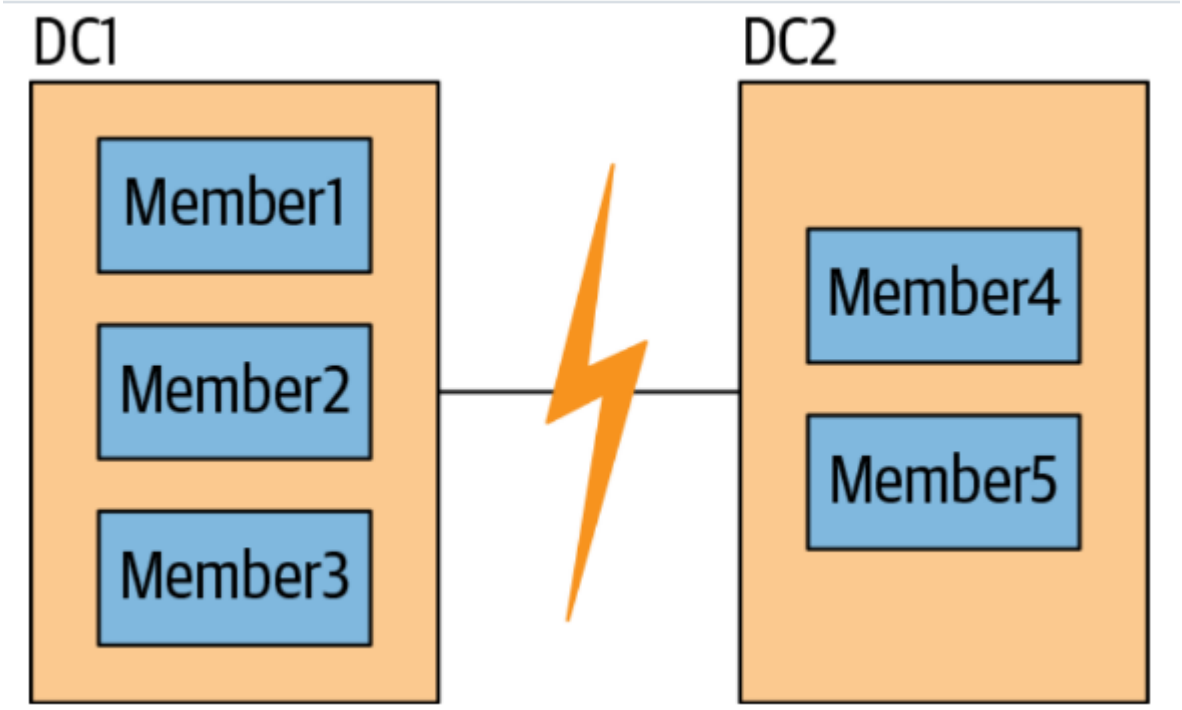
Tenga en cuenta que no importa cuántos miembros estén caídos o no disponibles; La mayoría se basa en la configuración del conjunto.

Por ejemplo, supongamos que tenemos un conjunto de cinco miembros y tres miembros "se caen", como se muestra en la siguiente figura. Todavía quedan dos miembros arriba. Estos dos miembros no pueden alcanzar la mayoría del conjunto (al menos tres miembros), por lo que no pueden elegir un primario. Si uno de ellos fuera primario, dimitiría tan pronto como advirtiera que no podría alcanzar la mayoría. Después de unos segundos, su conjunto constaría de dos miembros secundarios y tres miembros inalcanzables.



Esta manera de operar tiene problemas.

¿por qué los dos miembros restantes no pueden elegir un primario? El problema es que es posible que los otros tres miembros en realidad no se cayeron, sino que fue la red la que se cayó, como se muestra en la siguiente figura En este caso, los tres nodos de la izquierda elegirán una primario, ya que pueden alcanzar la mayoría del conjunto (tres nodos de cinco). En el caso de una partición de red, no queremos que ambos lados de la partición elijan un primario, porque entonces el conjunto tendría dos primarios. Ambos primarios escribirían en la base de datos y los conjuntos de datos divergirían. Exigir una mayoría para elegir o mantener una primario es una buena manera de evitar terminar con más de una primario.



Mongo solamente soporta un nodo primario.

MongoDB no soporta más de un primario en un conjunto de réplicas por razones de consistencia de datos y simplicidad en la resolución de conflictos. Tener un solo nodo primario garantiza que todas las escrituras sean secuenciales y coherentes, lo que es crucial para mantener la integridad de los datos. Si hubiera múltiples primarios, surgirían complicaciones en la gestión de versiones de datos y conflictos de escritura, lo que podría llevar a inconsistencias en la base de datos. Este enfoque de un solo primario ayuda a evitar tales problemas y simplifica la arquitectura del sistema de base de datos.

El método para la elección de un primario

En MongoDB, la elección de un primario en un conjunto de réplicas se realiza mediante un proceso de votación automático. Cuando el nodo primario actual se vuelve inaccesible o deja de funcionar por alguna razón, los nodos secundarios realizan una votación para elegir

un nuevo primario. Cada nodo en el conjunto de réplicas tiene un voto, y el nodo que recibe la mayoría de los votos se convierte en el nuevo primario. Este proceso está diseñado para asegurar una transición rápida y confiable a un nuevo primario, manteniendo la disponibilidad y la consistencia de la base de datos.

Cuando una secundario no puede llegar a un primario, se comunicará con todos los demás miembros y solicitará que sea elegida primario. Estos otros miembros hacen varias comprobaciones: ¿Pueden llegar a un primario que el miembro que busca la elección no puede? ¿El miembro que busca ser elegido está al día con la replicación? ¿Hay algún miembro disponible con mayor prioridad que deba ser elegido en su lugar?

MongoDB utiliza el protocolo de consenso RAFT desarrollado por Diego Ongaro y John Ousterhout en la Universidad de Stanford. Se describe mejor como similar a RAFT y está diseñado para incluir una serie de conceptos de replicación que son específicos de MongoDB, como árbitros, prioridad, miembros sin derecho a voto, preocupación de escritura, etc. La versión 1 del protocolo proporcionó la base para nuevas características como tiempo de conmutación por error más corto y reduce en gran medida el tiempo para detectar situaciones primarias falsas. También evita la doble votación mediante el uso de identificaciones de términos.

Los miembros del conjunto de réplicas se envían latidos (pings) entre sí cada dos segundos. Si un miembro no regresa un latido dentro de 10 segundos, los otros miembros marcan al miembro moroso como inaccesible. El algoritmo electoral hará un “mejor esfuerzo” para que el secundario con la mayor prioridad disponible convoque una elección. La prioridad de los miembros afecta tanto el momento como el resultado de las elecciones; Los nodos secundarios con mayor prioridad convocan elecciones relativamente antes que los noos secundarios con menor prioridad, y también tienen más probabilidades de ganar. Sin embargo, una instancia de menor prioridad puede ser elegida como primario por períodos breves, incluso si hay disponible una instancia secundaria de mayor prioridad. Los miembros del conjunto de réplicas continúan convocando elecciones hasta que el miembro de mayor prioridad disponible se convierta en primario.

Para ser elegido primario, un miembro debe estar al día con la replicación, hasta donde lo saben los miembros a los que puede llegar. Todas las operaciones replicadas están estrictamente ordenadas por un identificador ascendente, por lo que el candidato debe tener operaciones posteriores o iguales a las de cualquier miembro al que pueda alcanzar.

¿Qué es la prioridad?

La prioridad es una indicación de con qué fuerza este miembro "quiere" convertirse en primario. Su valor puede oscilar entre 0 y 100 y el valor predeterminado es 1. Establecer "prioridad" en 0 tiene un significado especial: los miembros con una prioridad de 0 nunca pueden convertirse en primarios. Estos se llaman miembros pasivos.

El miembro de mayor prioridad siempre será elegido primario (siempre que pueda alcanzar la mayoría del conjunto y tenga los datos más actualizados). Por ejemplo, supongamos que agrega un miembro con una prioridad de 1,5 al conjunto, así:

> rs.add({"host": "servidor-4:27017", "prioridad": 1.5})

Suponiendo que los otros miembros del conjunto tengan prioridad 1, una vez que el servidor 4 alcance al resto del conjunto, el primario actual automáticamente renunciará y el servidor 4 se elegirá a sí mismo. Si, por alguna razón, el servidor 4 no pudiera ponerse al día, el principal actual seguiría siendo el principal. Establecer prioridades nunca hará que tu conjunto se quede sin primarias. Tampoco hará que un miembro que está atrasado se convierta en primario (hasta que lo haya alcanzado).

El valor absoluto de "prioridad" sólo importa en relación con si es mayor o menor que las otras prioridades del conjunto: los miembros con prioridades de 100, 1 y 1 se comportarán de la misma manera que los miembros de otro conjunto con prioridades 2, 1 y 1.

Miembros "ocultos"

Los "hidden members" en un conjunto de réplicas de MongoDB son nodos que forman parte del conjunto pero están configurados para no ser elegibles como primarios y no son visibles para las operaciones de lectura del cliente. Estos nodos reciben actualizaciones del oplog como cualquier otro miembro secundario, pero su función principal es a menudo para respaldos o tareas administrativas que no requieren visibilidad por parte de las aplicaciones cliente. Al configurar un nodo como "oculto", aseguramos que este continúe replicando datos sin afectar el tráfico de lectura o la elección de nodos primarios.

Ejemplo

Supongamos el siguiente replica set

> rs.isMaster()

{

...

"hosts" : [

"server-1:27107",

"server-2:27017",

"server-3:27017"

],

...

}

Ocultemos a server-3

> var config = rs.config()

```
> config.members[2].hidden = true
0
> config.members[2].priority = 0
0
> rs.reconfig(config)
```

Ejecutemos nuevamente rs.isMaster()

```
> rs.isMaster()
{
...
"hosts" : [
"server-1:27107",
"server-2:27017"
],
...
}
```

rs.status() y rs.config() seguirán mostrando el miembro; sólo desaparece de isMaster. Cuando los clientes se conectan a un conjunto de réplicas, llaman a isMaster para determinar los miembros del conjunto. Por lo tanto, los miembros ocultos nunca se utilizarán para solicitudes de lectura.

Nodos árbitros

Los nodos árbitros en un conjunto de réplicas de MongoDB son miembros especiales que no almacenan datos ni replican el oplog. Su principal función es participar en las votaciones para la elección del nodo primario. Son útiles en situaciones donde se necesita un voto decisivo para mantener la mayoría en un conjunto de réplicas, especialmente en configuraciones con un número par de nodos. Un árbitro ayuda a prevenir el problema de "división de cerebros" (split brain) en el conjunto de réplicas y asegura la disponibilidad continua del servicio al permitir que se elija un primario incluso cuando algunos nodos están inaccesibles.

Como los árbitros no tienen ninguna de las responsabilidades tradicionales de un servidor mongod, puede ejecutar un árbitro como un proceso liviano en un servidor más débil que el que generalmente usaría para MongoDB. A menudo es una buena idea, si es posible, ejecutar un árbitro en un dominio de falla separado de los demás miembros, para que tenga una "perspectiva externa" del conjunto.

El uso de un árbitro tiene sus desventajas:

No contribuye a la tolerancia a fallos: Aunque ayuda en la toma de decisiones para la elección del nodo primario, el árbitro no almacena datos ni proporciona redundancia adicional para la recuperación de datos.

Complejidad en la configuración: Agregar un árbitro implica una configuración adicional y mantenimiento de un componente más en el sistema.

No mejora el rendimiento de lectura/escritura: Dado que el árbitro no maneja datos, no contribuye a mejorar el rendimiento de las operaciones de lectura o escritura.

Riesgo en escenarios de partición de red: En ciertas situaciones de partición de red, la presencia de un árbitro puede llevar a escenarios donde los nodos que tienen los datos no pueden elegir un primario debido a la falta de mayoría.

Estas desventajas deben ser consideradas cuidadosamente al diseñar la arquitectura de un conjunto de réplicas en MongoDB.