验证 MySQL MGR 双机房双活架构可行性

原创 洪斌 玩转MySQL 2025年05月06日 11:23 四川

在实际生产环境中，很多企业都希望实现**双机房双活**，以提升业务的高可用性和容灾能力。恰好最近某客户问到：**MGR 如何实现双机房双活？**

MySQL Group Replication（MGR）作为官方的高可用方案，支持多主模式、异步复制链路故障转移（Asynchronous Replication Channel Failover）。那么，在双活场景下是否可行？有哪些坑？

为此，我用cursor写了个测试脚本，来快速验证 MGR 架构下双机房双活的可行性。

设计目标

- 验证 MySQL MGR 在双机房部署下，能否实现双活（即两地同时可写、数据自动同步、故障自动切换）。
- 自动化部署、切换、故障模拟、数据一致性校验 减少人工干预，提升测试效率

主要功能

1.一键部署双集群

- 支持单主 / 多主两种模式
- 使用 dbdeployer 快速搭建两套 MGR 集群，模拟双机房环境

2.灵活复制链路测试

- 可配置单向或双向复制
- 自动建立异步复制通道，模拟双活场景下的数据写入

3.自动化故障与恢复

- 支持主节点故障模拟
- 自动等待新主选举
- 节点重启与重新加入集群
- 完整还原生产故障场景

4.数据一致性与复制状态校验

- 自动插入测试数据
- 校验两地数据同步情况
- 实时监控复制链路健康状态

测试结论

通过该脚本的自动化测试，得出如下结论：

- **单主模式**下不建议配置双向复制，推荐使用 **MySQL Shell 部署 ClusterSet 架构**，通过 **MySQL Router 实现两集群双活**
- 单主或多主模式下若配置双向复制，**必须启用 skip_replica_start** ，否则节点重启后在加入集群前会先同步对方集群的数据，导致与本组 GTID 不一致，无法加入集群
- **MySQL Shell 暂不支持配置双向复制**，只能部署 InnoDB ClusterSet 架构

使用方法示例

```
### 部署单主模式集群
python mgr_test.py -s

### 部署多主模式集群
python mgr_test.py -m

### 运行单主单向复制测试
python mgr_test.py -ss

### 运行多主双向复制测试
python mgr_test.py -aa
```

脚本源码

```python
#!/usr/bin/env python3

from typing import List, Dict, Literal
import subprocess
import time
from dataclasses import dataclass
from datetime import datetime
import argparse
MYSQL_VERSION = "8.0.40"
REPL_USER = "msandbox"
REPL_PASSWORD = "msandbox"
# MGR集群端口配置
MGR1_PORTS = [4316, 4317, 4318]  # 第一个MGR集群的端口
MGR2_PORTS = [5316, 5317, 5318]  # 第二个MGR集群的端口
SANDBOX= "$HOME/workbench/sandboxs"
REPLICATION_CHANNELS = {
```

```
    "mgr1": "async_mgr2_to_mgr1",  # mgr1 集群上的通道
    "mgr2": "async_mgr1_to_mgr2"   # mgr2 集群上的通道
}
@dataclass
class MGRNode:
    port: int
    host: str = '127.0.0.1'
    user: str = 'msandbox'
    password: str = 'msandbox'

    def execute_sql(self, sql: str, vertical: bool = False) -> str:
        cmd = f"mysql -h{self.host} -P{self.port} -u{self.user} -p{se
        if not vertical:
            cmd += " -NB"
        cmd += f" -e \"{sql}\""
        try:
            result = subprocess.run(cmd, shell=True, capture_output=T
            return result.stdout.strip()
        except subprocess.CalledProcessError as e:
            error_msg = e.stderr.strip()
            print("\n📋 最近的错误日志：")
            self.show_error_log()
            raise Exception(e.stderr)

    def show_error_log(self, lines: int = 20):
        """显示最近的错误日志

        Args:
            lines: 显示的日志行数
        """
        try:
            # 获取错误日志文件路径
            log_file = f"{SANDBOX}/mgr{1 if self.port < 5000 else 2}/
            cmd = f"tail -n {lines} {log_file}|grep ERROR"
            result = subprocess.run(cmd, shell=True, capture_output=T
            print(result.stdout.strip())
        except subprocess.CalledProcessError as e:
            print(f"⚠️ 无法读取错误日志：{e.stderr.strip()}")

    def shutdown(self, force: bool = False):
        if force:
            # 只杀死MySQL进程，而不是所有监听该端口的进程
            cmd = f"ps aux | grep mysql.*{self.port} | grep -v grep |
        else:
            cmd = f"mysqladmin -u{self.user} -p{self.password} -h{sel
        subprocess.run(cmd, shell=True, check=True)
```

```
class MGRCluster:
    def __init__(self, name: str, ports: List[int], mysql_version: st
        self.name = name
        self.nodes = [MGRNode(port) for port in ports]
        self.primary_node = self.nodes[0]
        self.mysql_version = mysql_version
        self._primary_node_cache = None
        self._last_primary_check = 0
        self._cache_ttl = 10  # 缓存10秒

    def deploy(self, topology_mode: Literal["single-primary", "multi-
        # 先删除已存在的集群
        try:
            print(f"=== 删除集群 {self.name} ===")
            cmd = f"ps aux | grep {self.name} | grep -v grep | awk '{
            cmd = f"dbdeployer delete {self.name} --concurrent 2>/dev
            subprocess.run(cmd, shell=True, check=False)
        except:
            pass # 忽略删除失败

        # 部署新的MGR集群
        print(f"=== 部署集群 {self.name} ===")
        cmd = f"dbdeployer deploy replication --topology=group -c ski
        if topology_mode == "single-primary":
            cmd += " --single-primary"
        cmd += f" --nodes={len(self.nodes)} --concurrent --port-as-se
        print(f"{cmd}")
        subprocess.run(cmd, shell=True, check=True)
        # 清除缓存
        self._primary_node_cache = None
        self._last_primary_check = 0
    def get_primary_node(self, force_check: bool = False) -> MGRNode:
        """获取主节点

        Args:
            force_check: 是否强制检查，忽略缓存
        """
        current_time = time.time()

        # 如果缓存有效且不强制检查，直接返回缓存的主节点
        if not force_check and self._primary_node_cache and \
           (current_time - self._last_primary_check) < self._cache_tt
            return self._primary_node_cache
        print(f"\n🔍 开始查找集群 {self.name} 的主节点")

        # 先检查组复制状态
```

```
        for node in self.nodes:
            try:
                print(f"  ⚡ 检查节点 {node.port}")
                time.sleep(3)
                group_status = node.execute_sql("""
                    SELECT COUNT(*) FROM performance_schema.replicati
                    WHERE MEMBER_STATE = 'ONLINE'
                """)
                print(f"  📊 在线节点数: {group_status}")

                # 如果有节点在线，从这个节点查询主节点信息
                if int(group_status) > 0:
                    members = node.execute_sql("""
                        SELECT CONCAT(MEMBER_PORT, ' (', MEMBER_STATE
                        FROM performance_schema.replication_group_mem
                    """)
                    print(f"  📋 节点状态: {members.replace('\n', ', ')

                    # 查找主节点
                    for check_node in self.nodes:
                        role = check_node.execute_sql("""
                            SELECT MEMBER_ROLE
                            FROM performance_schema.replication_group_
                            WHERE MEMBER_ID = @@server_uuid
                        """)
                        if role == "PRIMARY":
                            self.primary_node = check_node
                            # 更新缓存
                            self._primary_node_cache = check_node
                            self._last_primary_check = current_time
                            print(f"  ✅ 找到主节点: {check_node.port}"
                            return check_node
                    break  # 如果找到在线节点但没找到主节点，不需要继续检查其他
            except subprocess.CalledProcessError as e:
                print(f"  ❌ 检查节点 {node.port} 失败")
                continue

        # 清除缓存
        self._primary_node_cache = None
        self._last_primary_check = 0
        raise Exception(f"❗ 集群 {self.name} 中未找到主节点，可能正在进行主

    def init_test_schema(self):
        """初始化测试数据库和表"""
        primary = self.get_primary_node()
        primary.execute_sql("""
```

```
            CREATE DATABASE IF NOT EXISTS mgr_test;
            CREATE TABLE IF NOT EXISTS mgr_test.events (
                id INT AUTO_INCREMENT PRIMARY KEY,
                event_time DATETIME,
                cluster_name VARCHAR(10),
                event_type VARCHAR(20),
                event_data VARCHAR(100)
            );
        """)
    def switch_mode(self, to_single_primary: bool = True):
        """切换MGR模式"""
        primary = self.get_primary_node()
        mode = "单主" if to_single_primary else "多主"
        print(f"\n🔄 切换{self.name}到{mode}模式")

        try:
            mode_sql = "SINGLE_PRIMARY" if to_single_primary else "MU
            primary.execute_sql(f"""
                SELECT group_replication_switch_to_{mode_sql.lower()}
            """)
            # 清除缓存强制重新检查主节点
            self._primary_node_cache = None
            self._last_primary_check = 0
            print(f"✅ {self.name}已切换到{mode}模式")
        except Exception as e:
            print(f"❌ 切换失败：{str(e)}")
            raise
class ReplicationManager:
    def __init__(self, source_cluster: MGRCluster, target_cluster: MG
        self.source = source_cluster
        self.target = target_cluster

    def setup_replication(self, source: MGRNode, target: MGRNode, cha
        channel_clause = f" FOR CHANNEL '{channel}'" if channel else
        auto_failover_clause = ",\n            SOURCE_CONNECTION_AUTO
        target.execute_sql(f"""
            CHANGE REPLICATION SOURCE TO
            SOURCE_HOST='{source.host}',
            SOURCE_PORT={source.port},
            SOURCE_USER='{REPL_USER}',
            SOURCE_PASSWORD='{REPL_PASSWORD}',
            SOURCE_CONNECT_RETRY=3,
            SOURCE_AUTO_POSITION=1{auto_failover_clause}
            FOR CHANNEL '{channel}';
            START REPLICA {channel_clause};
        """)
```

```
201
202     def setup_async_replication(self):
203         source = self.source.get_primary_node()
204         target = self.target.get_primary_node()
205         # 使用全局变量中的通道名称
206         self.setup_replication(source, target, REPLICATION_CHANNELS[s
207
208     def setup_active_active_replication(self):
209         source_primary = self.source.get_primary_node()
210         target_primary = self.target.get_primary_node()
211
212         # 使用全局变量中的通道名称
213         self.setup_replication(source_primary, target_primary,
214                                REPLICATION_CHANNELS[self.target.name],
215                                auto_failover=False)
216         self.setup_replication(target_primary, source_primary,
217                                REPLICATION_CHANNELS[self.source.name],
218                                auto_failover=False)
219 # 部署相关功能
220 class ClusterDeployer:
221     def __init__(self, mysql_version: str = MYSQL_VERSION):
222         self.mysql_version = mysql_version
223
224     def deploy_single_primary_clusters(self, cluster1_ports: List[int
225         """部署两个单主模式MGR集群"""
226         cluster1 = MGRCluster("mgr1", cluster1_ports, self.mysql_vers
227         cluster2 = MGRCluster("mgr2", cluster2_ports, self.mysql_vers
228
229         print("\n🚀 开始部署测试集群")
230         cluster1.deploy("single-primary")
231         cluster2.deploy("single-primary")
232
233         print("\n📝 初始化测试数据库")
234         cluster1.init_test_schema()
235         cluster2.init_test_schema()
236
237         return cluster1, cluster2
238
239     def deploy_multi_primary_clusters(self, cluster1_ports: List[int]
240         """部署两个多主模式MGR集群"""
241         cluster1 = MGRCluster("mgr1", cluster1_ports, self.mysql_vers
242         cluster2 = MGRCluster("mgr2", cluster2_ports, self.mysql_vers
243
244         print("\n🚀 开始部署多主模式测试集群")
245         cluster1.deploy("multi-primary")
246         cluster2.deploy("multi-primary")
```

```
247
248         print("\n📝 初始化测试数据库")
249         cluster1.init_test_schema()
250         cluster2.init_test_schema()
251
252         return cluster1, cluster2
253 # 测试相关功能
254 class ClusterTester:
255     def __init__(self, cluster1: MGRCluster = None, cluster2: MGRClus
256         self.cluster1 = cluster1
257         self.cluster2 = cluster2
258
259     def _check_clusters(self):
260         """检查集群是否已部署"""
261         try:
262             # 检查 dbdeployer 中的沙箱列表
263             cmd = "dbdeployer sandboxes --header | grep -E 'mgr[12]'"
264             result = subprocess.run(cmd, shell=True, capture_output=T
265             if result.stdout.strip():
266                 # 找到已部署的集群，初始化集群对象
267                 self.cluster1 = MGRCluster("mgr1", MGR1_PORTS)
268                 self.cluster2 = MGRCluster("mgr2", MGR2_PORTS)
269                 return
270         except subprocess.CalledProcessError:
271             pass
272
273         raise Exception("❌ 未找到已部署的集群，请先使用 's' 或 'm' 命令部署
274
275     @staticmethod
276     def insert_test_data(node: MGRNode, cluster_name: str, event_type
277         """插入测试数据
278
279         Args:
280             node: 目标节点
281             cluster_name: 集群名称
282             event_type: 事件类型（如：初始数据、故障后数据等）
283             data: 事件数据
284         """
285         node.execute_sql(f"""
286             INSERT INTO mgr_test.events (event_time, cluster_name, ev
287             VALUES (NOW(), '{cluster_name}', '{event_type}', '{data}'
288         """)
289
290     @staticmethod
291     def verify_data_sync(node: MGRNode):
292         """验证数据同步状态"""
```

```
        result = node.execute_sql("""
            SELECT CONCAT(
                '时间: ', event_time,
                ', 集群: ', cluster_name,
                ', 类型: ', event_type,
                ', 数据: ', event_data
            ) FROM mgr_test.events ORDER BY event_time
        """)
        print(f"\n📊 在节点 {node.port} 的数据：\n{result}")

    @staticmethod
    def verify_replication_status(node: MGRNode, channel: str = ""):
        """验证复制状态"""
        print(f"\n📡 验证复制通道 '{channel}' 状态")
        cmd = f"mysql -h{node.host} -P{node.port} -u{node.user} -p{no
        try:
            result = subprocess.run(cmd, shell=True, capture_output=T
            print(result.stdout)
        except subprocess.CalledProcessError as e:
            print(f"❌ 检查复制状态失败: {e.stderr if e.stderr else '通道
            raise

    def setup_replication(self, mode: Literal["async", "active-active
        """设置复制关系"""
        self._check_clusters()
        repl_mgr = ReplicationManager(self.cluster1, self.cluster2)

        mode_name = "单向" if mode == "async" else "双向"
        print(f"\n🔄 设置{mode_name}复制")
        if mode == "async":
            repl_mgr.setup_async_replication()
            self.verify_replication_status(self.cluster2.get_primary_
                                           REPLICATION_CHANNELS[self.cl
        else:
            repl_mgr.setup_active_active_replication()
            for cluster in [self.cluster1, self.cluster2]:
                self.verify_replication_status(cluster.get_primary_no
                                               REPLICATION_CHANNELS[clu
    def write_test_data(self, cluster: MGRCluster, event_type: str =
        """写入测试数据"""
        primary = cluster.get_primary_node()
        print(f"\n📝 在{cluster.name}的主节点{primary.port}写入测试数据")
        self.insert_test_data(primary, cluster.name, event_type,
                              f"来自节点{primary.port}")
        time.sleep(1)  # 间隔写入以区分时间顺序
    def simulate_node_failure(self, node: MGRNode, cluster: MGRCluste
```

```
        """模拟节点故障

        Args:
            node: 要故障的节点
            cluster: 节点所属集群

        Returns:
            MGRNode: 新的主节点（如果发生切换）
        """
        print(f"\n💥 模拟故障：关闭节点 {node.port}")
        node.shutdown(force=True)

        # 如果故障节点是主节点，等待新主选举
        if node.port == cluster.primary_node.port:
            return self._wait_for_new_primary(cluster)
        return cluster.get_primary_node()
    def recover_node(self, node: MGRNode, cluster: MGRCluster):
        """恢复故障节点

        Args:
            node: 要恢复的节点
            cluster: 节点所属集群
        """
        print(f"\n🔄 重启节点 {node.port} 并重新加入集群")
        node_index = cluster.nodes.index(node) + 1
        subprocess.run(f"{SANDBOX}/{cluster.name}/node{node_index}/st
                       shell=True, check=True)
        time.sleep(2)  # 等待启动完成
        print(f"\n🔄 节点: {node.port} START GROUP_REPLICATION 重新加入
        node.execute_sql("START GROUP_REPLICATION;")
        print(f"\n🔄 节点: {node.port} START REPLICA 开启异步复制")
        node.execute_sql("START REPLICA;")
        time.sleep(10)  # 等待节点加入
    def verify_cluster_data(self, clusters: List[MGRCluster]):
        """验证集群数据同步状态"""
        print("\n🔍 验证数据同步")
        time.sleep(5)  # 等待数据同步

        for cluster in clusters:
            primary = cluster.get_primary_node()
            # 验证数据同步
            self.verify_data_sync(primary)

            # 验证复制状态 - 使用固定的通道名称
            self.verify_replication_status(primary, REPLICATION_CHANN
    def test_async_failover(self):
```

```
        """测试单向复制故障转移"""
        # 设置单向复制
        self.setup_replication("async")

        # 写入初始数据
        self.write_test_data(self.cluster1, "初始数据")
        time.sleep(2)  # 等待复制

        # 模拟主节点故障
        primary1 = self.cluster1.get_primary_node()
        new_primary = self.simulate_node_failure(primary1, self.clust

        # 在新主写入数据
        self.write_test_data(self.cluster1, "故障转移后")
        time.sleep(2)

        # 恢复故障节点
        self.recover_node(primary1, self.cluster1)

        # 验证数据同步
        self.verify_cluster_data([self.cluster1, self.cluster2])
    def test_active_active_replication(self):
        """测试双向复制"""
        # 设置双向复制
        self.setup_replication("active-active")

        # 在两个集群写入数据
        for cluster in [self.cluster1, self.cluster2]:
            self.write_test_data(cluster, "初始数据")

        # 模拟节点故障和恢复
        primary1 = self.cluster1.get_primary_node()
        new_primary = self.simulate_node_failure(primary1, self.clust

        # 继续写入数据
        self.write_test_data(self.cluster1, "故障之后")
        self.write_test_data(self.cluster2, "故障期间")

        # 恢复故障节点
        self.recover_node(primary1, self.cluster1)

        # 验证数据同步
        self.verify_cluster_data([self.cluster1, self.cluster2])
    def _wait_for_new_primary(self, cluster: MGRCluster, max_wait: in
        """等待新主节点选举完成"""
        print("\n⏳ 等待新主节点选举")
```

```
        new_primary = None
        old_primary_port = cluster.primary_node.port

        for i in range(max_wait // interval):
            try:
                print(f"\n🔄 第 {i + 1} 次尝试查找 {cluster.name} 新主节
                # 强制检查主节点，忽略缓存
                new_primary = cluster.get_primary_node(force_check=Tr
                if new_primary and new_primary.port != old_primary_po
                    print(f"✅ 新主节点选举成功: {new_primary.port}")
                    cluster.primary_node = new_primary  # 更新集群的主节
                    return new_primary
            except Exception as e:
                # print(f"⏳ 等待主节点选举: \n{str(e)}")
                time.sleep(interval)
                continue

        raise Exception(f"❌ 在 {max_wait} 秒内未完成主节点选举")
def deploy_clusters(topology: Literal["single-primary", "multi-primar
    """部署MGR集群"""
    deployer = ClusterDeployer()
    if topology == "single-primary":
        return deployer.deploy_single_primary_clusters(MGR1_PORTS, MGI
    else:
        return deployer.deploy_multi_primary_clusters(MGR1_PORTS, MGR
def run_tests(test_type: Literal["async", "active-active"] = "active-
    """运行指定类型的测试"""
    tester = ClusterTester(*clusters) if clusters else ClusterTester(
    if test_type == "async":
        tester.test_async_failover()
    else:
        tester.test_active_active_replication()
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="MGR集群部署和测试工具")
    parser.add_argument('-s', '--deploy-single', action='store_true',
    parser.add_argument('-m', '--deploy-multi', action='store_true',
    parser.add_argument('-ss', '--single-async', action='store_true',
    parser.add_argument('-sa', '--single-active', action='store_true'
    parser.add_argument('-aa', '--multi-active', action='store_true',

    args = parser.parse_args()

    # 如果没有提供任何参数，显示帮助信息
    if not any(vars(args).values()):
        parser.print_help()
        exit(0)
```

```
    # 映射参数到场景
    if args.deploy_single:
        topology, test_type = "single-primary", None
    elif args.deploy_multi:
        topology, test_type = "multi-primary", None
    elif args.single_async:
        topology, test_type = "single-primary", "async"
    elif args.single_active:
        topology, test_type = "single-primary", "active-active"
    elif args.multi_active:
        topology, test_type = "multi-primary", "active-active"

    clusters = None
    tester = None

    # 如果是测试场景，先检查已部署的集群
    if test_type:
        try:
            tester = ClusterTester()
            tester._check_clusters()
            clusters = (tester.cluster1, tester.cluster2)

            # 根据测试场景切换集群模式
            to_single = topology == "single-primary"
            current_mode = "单主" if to_single else "多主"
            print(f"\n=== 切换到{current_mode}模式 ===")
            for cluster in clusters:
                cluster.switch_mode(to_single)
        except Exception as e:
            print(f"\n⚠️ 未找到已部署的集群或切换模式失败，重新部署集群")
            clusters = None

    # 如果是部署命令或者没有找到已部署的集群，部署新集群
    if not clusters:
        print(f"\n=== 部署{topology}模式集群 ===")
        clusters = deploy_clusters(topology)

    # 如果有测试类型，运行测试
    if test_type:
        print(f"\n=== 运行测试场景: {test_type} ===")
        run_tests(test_type, clusters)
```