RocketMQ

1消息的特殊处理

1.1 错乱的消息顺序

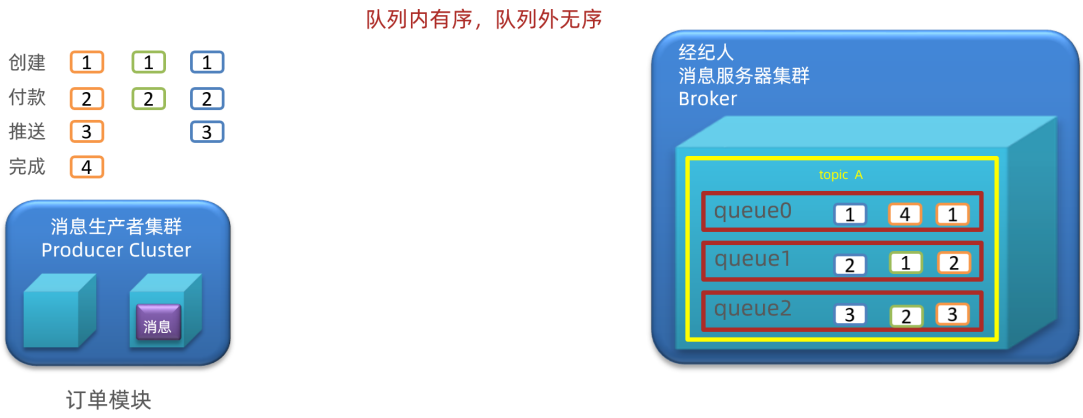
原因

消息有序指的是可以按照消息的发送顺序来消费(FIFO)。RocketMQ可以严格的保证消息有序，可以分为分区有序或者全局有序。

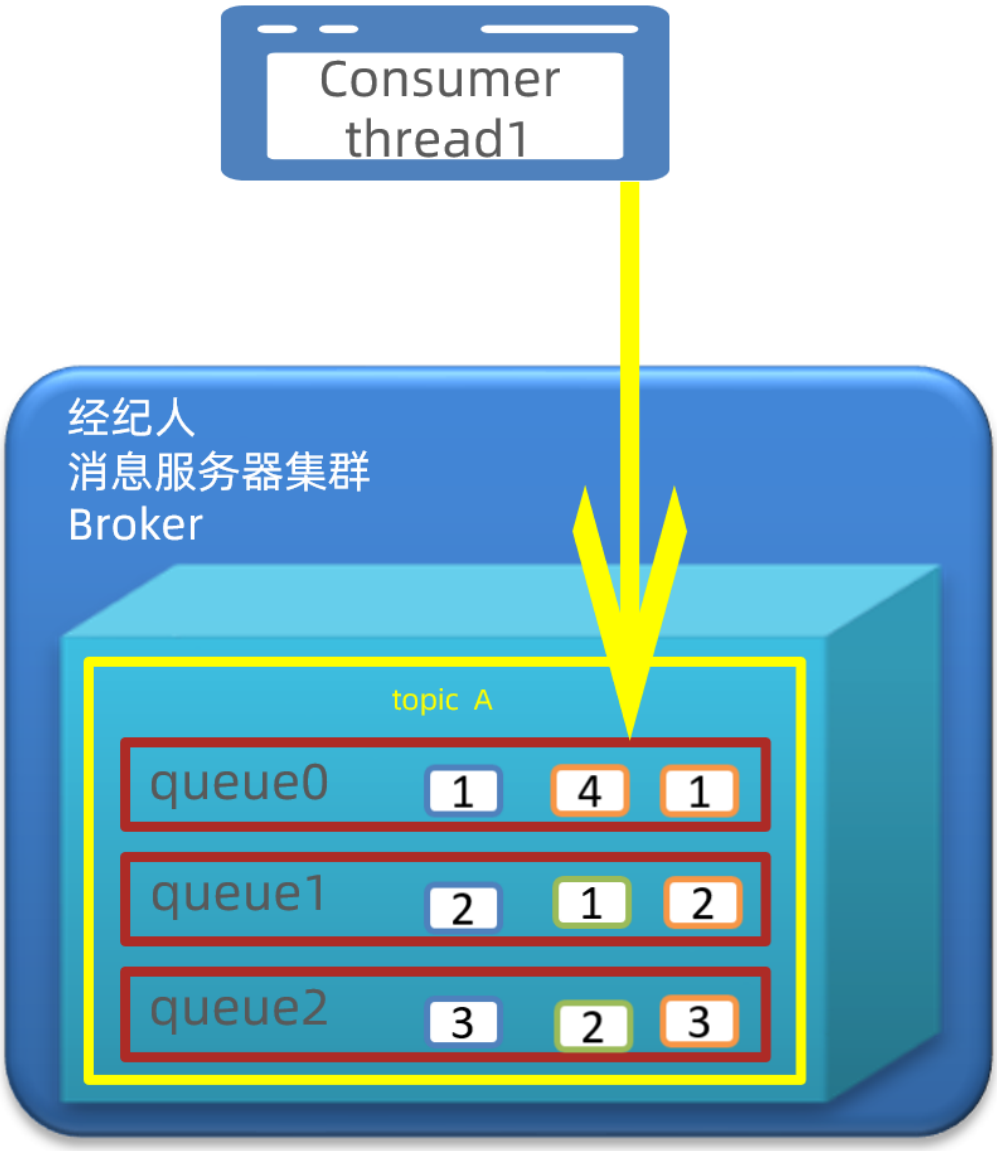
顺序消费的原理解析，在默认的情况下消息发送会采取Round Robin轮询方式把消息发送到不同的queue(分区队列)；而消费消息的时候从多个queue上拉取消息，这种情况发送和消费是不能保证顺序。但是如果控制发送的顺序消息只依次发送到同一个queue中，消费的时候只从这个queue上依次拉取，则就保证了顺序。当发送和消费参与的queue只有一个，则是全局有序；如果多个queue参与，则为分区有序，即相对每个queue，消息都是有序的。

下面用订单进行分区有序的示例。一个订单的顺序流程是：创建、付款、推送、完成。订单号相同的消息会被先后发送到同一个队列中，消费时，同一个OrderId获取到的肯定是同一个队列。

消息错乱原因

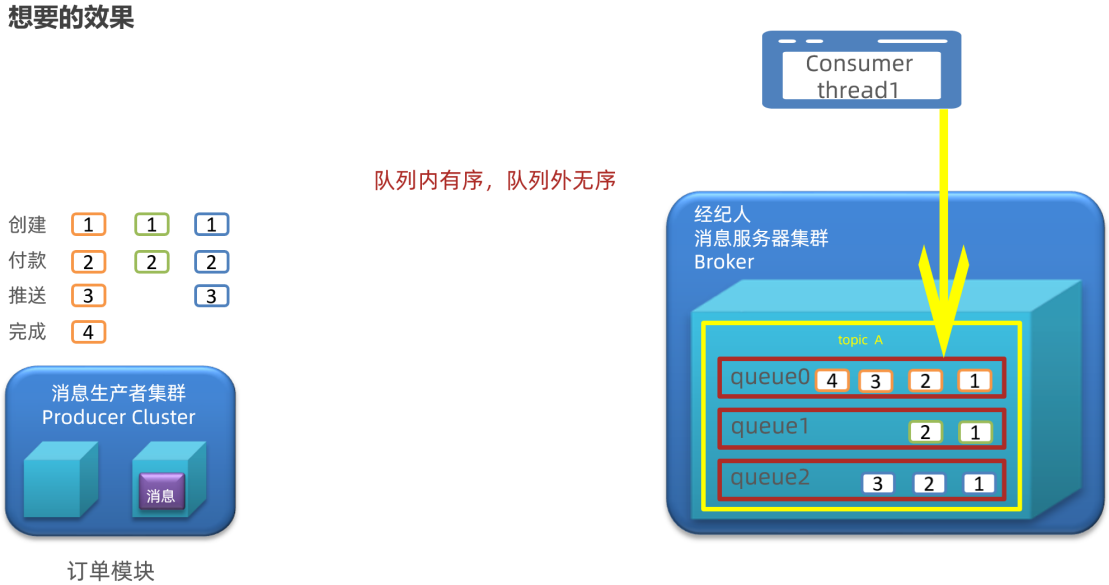


先读到第一个订单的创建和完成消息



想要的效果

想要的效果



订单模块

1.2 顺序消息

1.2.1订单步骤实体类

```
package com.itheima.order.domain;

/**
 * 订单的步骤
 */
public class OrderStep {
    private long orderId;
    private String desc;

    public long getOrderId() {
        return orderId;
    }

    public void setOrderId(long orderId) {
        this.orderId = orderId;
    }

    public String getDesc() {
        return desc;
    }

    public void setDesc(String desc) {
        this.desc = desc;
    }

    @Override
    public String toString() {
        return "OrderStep{" +
                "orderId=" + orderId +
                ", desc='" + desc + '\'' +
                '}';
    }
}
```

1.2.1发送消息

```
package com.itheima.order;

import com.itheima.order.domain.OrderStep;
import org.apache.rocketmq.client.producer.DefaultMQProducer;
import org.apache.rocketmq.client.producer.MessageQueueSelector;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.common.message.Message;
import org.apache.rocketmq.common.message.MessageQueue;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class Producer {
    public static void main(String[] args) throws Exception {
```

```
        DefaultMQProducer producer = new DefaultMQProducer("group1");
        producer.setNamesrvAddr("localhost:9876");
        producer.start();

        List<OrderStep> orderList = new Producer().buildOrders();

        //设置消息进入到指定的消息队列中
        for (final OrderStep order : orderList) {
            Message msg = new Message("topic1", order.toString().getBytes());
            //发送时要指定对应的消息队列选择器
            SendResult result = producer.send(msg, new MessageQueueSelector() {
                //设置当前消息发送时使用哪一个消息队列
                public MessageQueue select(List<MessageQueue> list, Message
message, Object o) {
                    //根据发送的信息不同，选择不同的消息队列
                    //根据id来选择一个消息队列的对象，并返回->id得到int值
                    long orderId = order.getOrderId();
                    long mqIndex = orderId % list.size();
                    return list.get((int) mqIndex);
                }
            }, null);
            System.out.println(result);
        }

        producer.shutdown();


        //for (int i = 0; i < 10; i++) {
        //    Message msg = new Message("topic1", ("hello
rocketmq"+i).getBytes("UTF-8"));
        //    SendResult result = producer.send(msg);
        //    System.out.println("返回结果：" + result);
        //}
    }

    /**
     * 生成模拟订单数据
     */
    private List<OrderStep> buildOrders() {
        List<OrderStep> orderList = new ArrayList<OrderStep>();

        OrderStep orderDemo = new OrderStep();
        orderDemo.setOrderId(1L);
        orderDemo.setDesc("创建");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(2L);
        orderDemo.setDesc("创建");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(1L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(3L);
```

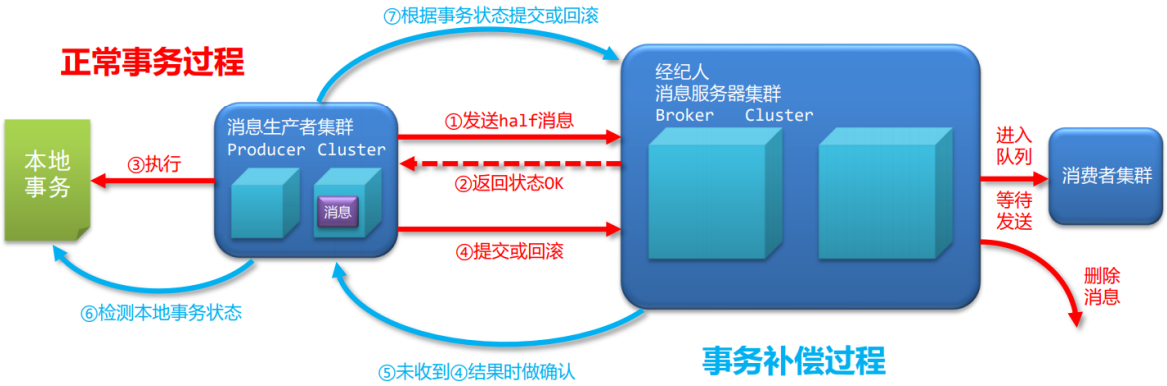
```
        orderDemo.setDesc("创建");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(2L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(3L);
        orderDemo.setDesc("付款");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(2L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(1L);
        orderDemo.setDesc("推送");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(3L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        orderDemo = new OrderStep();
        orderDemo.setOrderId(1L);
        orderDemo.setDesc("完成");
        orderList.add(orderDemo);

        return orderList;
    }
}
```

1.2.2接收消息

```
         //使用单线程的模式从消息队列中取数据，一个线程绑定一个消息队列
        consumer.registerMessageListener(new MessageListenerOrderly() {
            //使用MessageListenerOrderly接口后，对消息队列的处理由一个消息队列多个线程服
务，转化为一个消息队列一个线程服务
            public ConsumeOrderlyStatus consumeMessage(List<MessageExt> list,
ConsumeOrderlyContext consumeOrderlyContext) {
                for (MessageExt msg : list) {
                    System.out.println(Thread.currentThread().getName()+"。消息："
+ new String(msg.getBody())+"。queueId:"+msg.getQueueId());
                }
                return ConsumeOrderlyStatus.SUCCESS;
            }
        });
```

1.3 事务消息

1. 正常事务过程
2. 事务补偿过程



1.4 事务消息状态

1. 提交状态：允许进入队列，此消息与非事务消息无区别
2. 回滚状态：不允许进入队列，此消息等同于未发送过
3. 中间状态：完成了half消息的发送，未对MQ进行二次状态确认
4. 注意：事务消息仅与生产者有关，与消费者无关

1.5 事务消息

提交状态

```
//事务消息使用的生产者是TransactionMQProducer
TransactionMQProducer producer = new TransactionMQProducer("group1");
producer.setNamesrvAddr("localhost:9876");
//添加本地事务对应的监听
producer.setTransactionListener(new TransactionListener() {
//正常事务过程
public LocalTransactionState executeLocalTransaction(Message message, Object o)
{
return LocalTransactionState.COMMIT_MESSAGE;
}
//事务补偿过程
public LocalTransactionState checkLocalTransaction(MessageExt messageExt) {
return null;
}
});
producer.start();
Message msg = new Message("topic8",("事务消息：hello rocketmq ").getBytes("UTF-
8"));
SendResult result = producer.sendMessageInTransaction(msg,null);
System.out.println("返回结果："+result);
producer.shutdown();
```

回滚状态

```
            producer.setTransactionListener(new TransactionListener() {
            //正常事务
            @Override
            public LocalTransactionState executeLocalTransaction(Message msg,
Object arg) {
                return LocalTransactionState.ROLLBACK_MESSAGE;
            }
            //事务补偿
            @Override
            public LocalTransactionState checkLocalTransaction(MessageExt msg) {
                return null;
            }
        });
```

中间状态

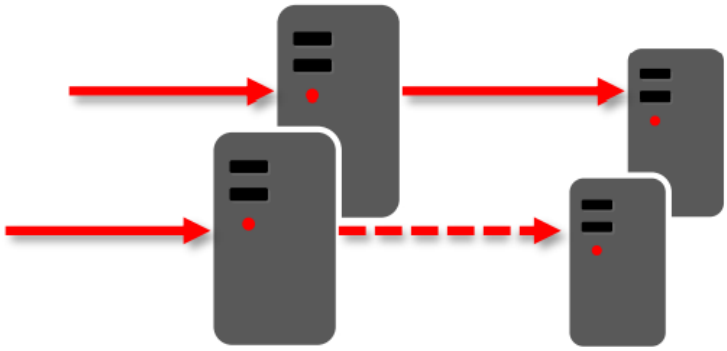
```
public static void main(String[] args) throws Exception {
        TransactionMQProducer producer=new TransactionMQProducer("group1");
        producer.setNamesrvAddr("localhost:9876");
        producer.setTransactionListener(new TransactionListener() {
            //正常事务
            @Override
            public LocalTransactionState executeLocalTransaction(Message msg,
Object arg) {
                return LocalTransactionState.UNKNOW;
            }
            //事务补偿 正常执行UNKNOW才会触发
            @Override
            public LocalTransactionState checkLocalTransaction(MessageExt msg) {
                System.out.println("事务补偿");
                return LocalTransactionState.COMMIT_MESSAGE;
            }
        });
        producer.start();
        Message msg = new Message("topic13", "hello rocketmq".getBytes("UTF-
8"));
        SendResult result = producer.sendMessageInTransaction(msg, null);
        System.out.println("返回结果: " + result);

        //事务补偿生产者一定要一直启动着
        //producer.shutdown();
    }
```

2. 集群搭建

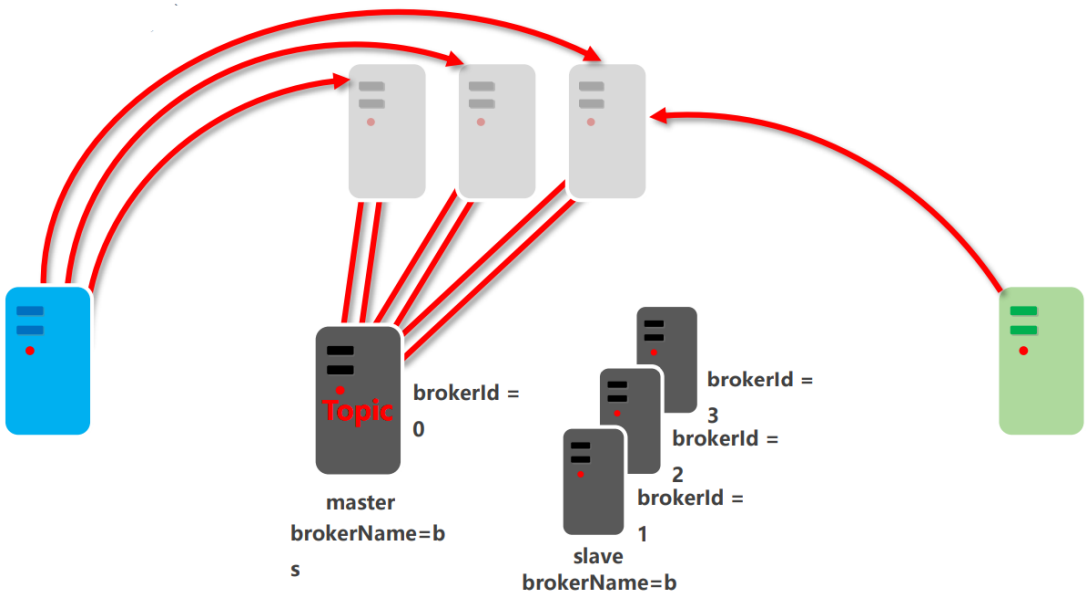
2.1 RocketMQ集群分类

1. 单机
    1. 一个broker提供服务（宕机后服务瘫痪）
2. 集群
    1. 多个broker提供服务（单机宕机后消息无法及时被消费）
    2. 多个master多个slave

1. master到slave消息同步方式为同步（较异步方式性能略低，消息无延迟）
2. master到slave消息同步方式为异步（较同步方式性能略高，数据略有延迟）



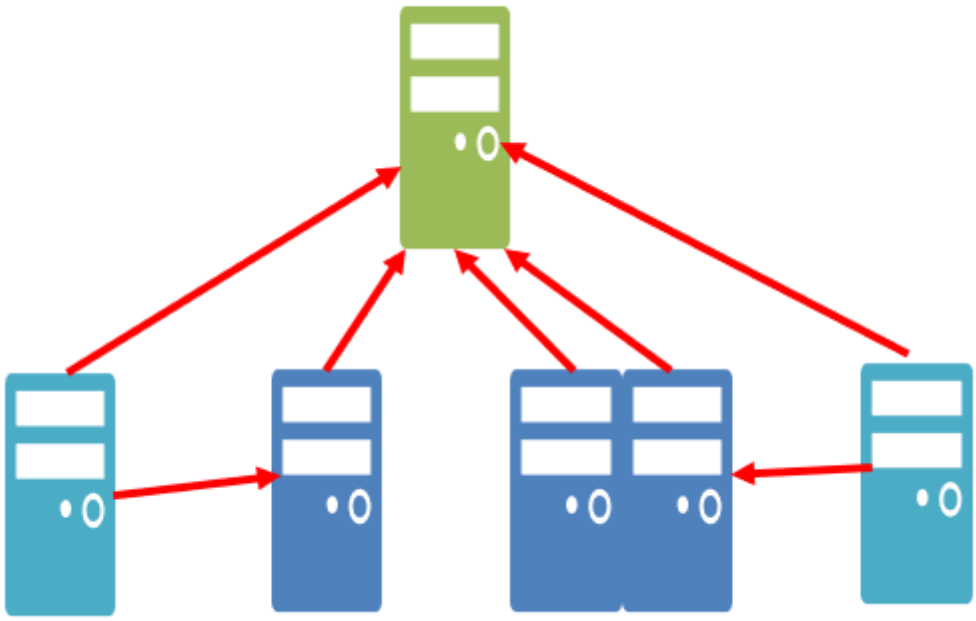
2.2 RocketMQ集群特征



RocketMQ集群工作流程

1. 步骤1：NameServer启动，开启监听，等待broker、producer与consumer连接
2. 步骤2：broker启动，根据配置信息，连接所有的NameServer，并保持长连接
3. 步骤2补充：如果broker中有现存数据， NameServer将保存topic与broker关系
4. 步骤3：producer发信息，连接某个NameServer，并建立长连接
5. 步骤4：producer发消息
    1. 步骤4.1若果topic存在，由NameServer直接分配
    2. 步骤4.2如果topic不存在，由NameServer创建topic与broker关系，并分配
6. 步骤5：producer在broker的topic选择一个消息队列（从列表中选择）
7. 步骤6：producer与broker建立长连接，用于发送消息
8. 步骤7：producer发送消息

comsumer工作流程同producer



双主双从集群搭建:

操作步骤：**注意两台机器同时操作**

1. 配置服务器环境:

```
vim /etc/hosts
```

```
# nameserver
192.168.200.129 rocketmq-nameserver1
192.168.200.130 rocketmq-nameserver2
# broker
192.168.200.129 rocketmq-master1
192.168.200.129 rocketmq-slave2
192.168.200.130 rocketmq-master2
192.168.200.130 rocketmq-slave1
```

2. 配置完毕后重启网卡，应用配置

```
systemctl restart network
```

3. 关闭防火墙或者开发指定端口对外提供服务

```
# 关闭防火墙
systemctl stop firewalld.service
# 查看防火墙的状态
firewall-cmd --state
# 禁止firewall开机启动
systemctl disable firewalld.service
```

4. 配置服务器环境

```
vim /etc/profile
```

```
#set rocketmq
ROCKETMQ_HOME=/rocketmq
PATH=$PATH:$ROCKETMQ_HOME/bin
export ROCKETMQ_HOME PATH
```

5. 配置完毕后重启网卡，应用配置

```
source /etc/profile
```

6. 将rocketmq解压到/rocketmq

7. 创建集群服务器的数据存储目录

```
#master 数据存储目录
mkdir /rocketmq/store
mkdir /rocketmq/store/commitlog
mkdir /rocketmq/store/consumequeue
mkdir /rocketmq/store/index

#slave 数据存储目录
mkdir /rocketmq/store-slave
mkdir /rocketmq/store-slave/commitlog
mkdir /rocketmq/store-slave/consumequeue
mkdir /rocketmq/store-slave/index
```

8. 注意master与slave如果在同一个虚拟机中部署，需要将存储目录区分开

9. **第一台129机器上**

```
cd /rocketmq/conf/2m-2s-sync
```

```
vim broker-a.properties
```

```
#所属集群名字
```

```
brokerClusterName=rocketmq-cluster
#broker名字，注意此处不同的配置文件填写的不一样
brokerName=broker-a
#0 表示 Master，>0 表示 Slave
brokerId=0
#nameServer地址，分号分割
namesrvAddr=rocketmq-nameserver1:9876;rocketmq-nameserver2:9876
#在发送消息时，自动创建服务器不存在的topic，默认创建的队列数
defaultTopicQueueNums=4
#是否允许 Broker 自动创建Topic，建议线下开启，线上关闭
autoCreateTopicEnable=true
#是否允许 Broker 自动创建订阅组，建议线下开启，线上关闭
autoCreateSubscriptionGroup=true
#Broker 对外服务的监听端口
listenPort=10911
#删除文件时间点，默认凌晨 4点
deleteWhen=04
#文件保留时间，默认 48 小时
fileReservedTime=120
#commitLog每个文件的大小默认1G
mapedFileSizeCommitLog=1073741824
#ConsumeQueue每个文件默认存30W条，根据业务情况调整
mapedFileSizeConsumeQueue=300000
#destroyMapedFileIntervalForcibly=120000
#redeleteHangedFileInterval=120000
#检测物理文件磁盘空间
diskMaxUsedSpaceRatio=88
#存储路径
storePathRootDir=/rocketmq/store
#commitLog 存储路径
storePathCommitLog=/rocketmq/store/commitlog
#消费队列存储路径存储路径
storePathConsumeQueue=/rocketmq/store/consumequeue
#消息索引存储路径
storePathIndex=/rocketmq/store/index
#checkpoint 文件存储路径
storeCheckpoint=/rocketmq/store/checkpoint
#abort 文件存储路径
abortFile=/rocketmq/store/abort
#限制的消息大小
maxMessageSize=65536
#flushCommitLogLeastPages=4
#flushConsumeQueueLeastPages=2
#flushCommitLogThoroughInterval=10000
#flushConsumeQueueThoroughInterval=60000
#Broker 的角色
#- ASYNC_MASTER 异步复制Master
#- SYNC_MASTER 同步双写Master
#- SLAVE
brokerRole=SYNC_MASTER
#刷盘方式
#- ASYNC_FLUSH 异步刷盘
#- SYNC_FLUSH 同步刷盘
flushDiskType=SYNC_FLUSH
#checkTransactionMessageEnable=false
#发消息线程池数量
#sendMessageThreadPoolNums=128
#拉消息线程池数量
```

```
#pullMessageThreadPoolNums=128
```

```
vim broker-b-s.properties
```

```
#所属集群名字
brokerClusterName=rocketmq-cluster
#broker名字，注意此处不同的配置文件填写的不一样
brokerName=broker-b
#0 表示 Master，>0 表示 Slave
brokerId=1
#nameServer地址，分号分割
namesrvAddr=rocketmq-nameserver1:9876;rocketmq-nameserver2:9876
#在发送消息时，自动创建服务器不存在的topic，默认创建的队列数
defaultTopicQueueNums=4
#是否允许 Broker 自动创建Topic，建议线下开启，线上关闭
autoCreateTopicEnable=true
#是否允许 Broker 自动创建订阅组，建议线下开启，线上关闭
autoCreateSubscriptionGroup=true
#Broker 对外服务的监听端口
listenPort=11011
#删除文件时间点，默认凌晨 4点
deleteWhen=04
#文件保留时间，默认 48 小时
fileReservedTime=120
#commitLog每个文件的大小默认1G
mapedFileSizeCommitLog=1073741824
#ConsumeQueue每个文件默认存30W条，根据业务情况调整
mapedFileSizeConsumeQueue=300000
#destroyMapedFileIntervalForcibly=120000
#redeleteHangedFileInterval=120000
#检测物理文件磁盘空间
diskMaxUsedSpaceRatio=88
#存储路径
storePathRootDir=/rocketmq/store-slave
#commitLog 存储路径
storePathCommitLog=/rocketmq/store-slave/commitlog
#消费队列存储路径存储路径
storePathConsumeQueue=/rocketmq/store-slave/consumequeue
#消息索引存储路径
storePathIndex=/rocketmq/store-slave/index
#checkpoint 文件存储路径
storeCheckpoint=/rocketmq/store-slave/checkpoint
#abort 文件存储路径
abortFile=/rocketmq/store-slave/abort
#限制的消息大小
maxMessageSize=65536
#flushCommitLogLeastPages=4
#flushConsumeQueueLeastPages=2
#flushCommitLogThoroughInterval=10000
#flushConsumeQueueThoroughInterval=60000
#Broker 的角色
#- ASYNC_MASTER 异步复制Master
#- SYNC_MASTER 同步双写Master
#- SLAVE
brokerRole=SLAVE
#刷盘方式
```

```
#- ASYNC_FLUSH 异步刷盘
#- SYNC_FLUSH 同步刷盘
flushDiskType=ASYNC_FLUSH
#checkTransactionMessageEnable=false
#发消息线程池数量
#sendMessageThreadPoolNums=128
#拉消息线程池数量
#pullMessageThreadPoolNums=128
```

```
rm -rf broker-a-s.properties
rm -rf broker-b.properties
```

第二台130机器上

```
cd /rocketmq/conf/2m-2s-sync
```

```
vim broker-b.properties
```

```
#所属集群名字
brokerClusterName=rocketmq-cluster
#broker名字，注意此处不同的配置文件填写的不一样
brokerName=broker-b
#0 表示 Master，>0 表示 Slave
brokerId=0
#nameServer地址，分号分割
namesrvAddr=rocketmq-nameserver1:9876;rocketmq-nameserver2:9876
#在发送消息时，自动创建服务器不存在的topic，默认创建的队列数
defaultTopicQueueNums=4
#是否允许 Broker 自动创建Topic，建议线下开启，线上关闭
autoCreateTopicEnable=true
#是否允许 Broker 自动创建订阅组，建议线下开启，线上关闭
autoCreateSubscriptionGroup=true
#Broker 对外服务的监听端口
listenPort=10911
#删除文件时间点，默认凌晨 4点
deleteWhen=04
#文件保留时间，默认 48 小时
fileReservedTime=120
#commitLog每个文件的大小默认1G
mapedFileSizeCommitLog=1073741824
#ConsumeQueue每个文件默认存30W条，根据业务情况调整
mapedFileSizeConsumeQueue=300000
#destroyMapedFileIntervalForcibly=120000
#redeleteHangedFileInterval=120000
#检测物理文件磁盘空间
diskMaxUsedSpaceRatio=88
#存储路径
storePathRootDir=/rocketmq/store
#commitLog 存储路径
storePathCommitLog=/rocketmq/store/commitlog
#消费队列存储路径存储路径
storePathConsumeQueue=/rocketmq/store/consumequeue
#消息索引存储路径
storePathIndex=/rocketmq/store/index
#checkpoint 文件存储路径
```

```
storeCheckpoint=/rocketmq/store/checkpoint
#abort 文件存储路径
abortFile=/rocketmq/store/abort
#限制的消息大小
maxMessageSize=65536
#flushCommitLogLeastPages=4
#flushConsumeQueueLeastPages=2
#flushCommitLogThoroughInterval=10000
#flushConsumeQueueThoroughInterval=60000
#Broker 的角色
#- ASYNC_MASTER 异步复制Master
#- SYNC_MASTER 同步双写Master
#- SLAVE
brokerRole=SYNC_MASTER
#刷盘方式
#- ASYNC_FLUSH 异步刷盘
#- SYNC_FLUSH 同步刷盘
flushDiskType=SYNC_FLUSH
#checkTransactionMessageEnable=false
#发消息线程池数量
#sendMessageThreadPoolNums=128
#拉消息线程池数量
#pullMessageThreadPoolNums=128
```

```
vim broker-a-s.properties
```

```
#所属集群名字
brokerClusterName=rocketmq-cluster
#broker名字，注意此处不同的配置文件填写的不一样
brokerName=broker-a
#0 表示 Master，>0 表示 Slave
brokerId=1
#nameServer地址，分号分割
namesrvAddr=rocketmq-nameserver1:9876;rocketmq-nameserver2:9876
#在发送消息时，自动创建服务器不存在的topic，默认创建的队列数
defaultTopicQueueNums=4
#是否允许 Broker 自动创建Topic，建议线下开启，线上关闭
autoCreateTopicEnable=true
#是否允许 Broker 自动创建订阅组，建议线下开启，线上关闭
autoCreateSubscriptionGroup=true
#Broker 对外服务的监听端口
listenPort=11011
#删除文件时间点，默认凌晨 4点
deleteWhen=04
#文件保留时间，默认 48 小时
fileReservedTime=120
#commitLog每个文件的大小默认1G
mapedFileSizeCommitLog=1073741824
#ConsumeQueue每个文件默认存30W条，根据业务情况调整
mapedFileSizeConsumeQueue=300000
#destroyMapedFileIntervalForcibly=120000
#redeleteHangedFileInterval=120000
#检测物理文件磁盘空间
diskMaxUsedSpaceRatio=88
#存储路径
storePathRootDir=/rocketmq/store-slave
```

```
#commitLog 存储路径
storePathCommitLog=/rocketmq/store-slave/commitlog
#消费队列存储路径存储路径
storePathConsumeQueue=/rocketmq/store-slave/consumequeue
#消息索引存储路径
storePathIndex=/rocketmq/store-slave/index
#checkpoint 文件存储路径
storeCheckpoint=/rocketmq/store-slave/checkpoint
#abort 文件存储路径
abortFile=/rocketmq/store-slave/abort
#限制的消息大小
maxMessageSize=65536
#flushCommitLogLeastPages=4
#flushConsumeQueueLeastPages=2
#flushCommitLogThoroughInterval=10000
#flushConsumeQueueThoroughInterval=60000
#Broker 的角色
#- ASYNC_MASTER 异步复制Master
#- SYNC_MASTER 同步双写Master
#- SLAVE
brokerRole=SLAVE
#刷盘方式
#- ASYNC_FLUSH 异步刷盘
#- SYNC_FLUSH 同步刷盘
flushDiskType=ASYNC_FLUSH
#checkTransactionMessageEnable=false
#发消息线程池数量
#sendMessageThreadPoolNums=128
#拉消息线程池数量
#pullMessageThreadPoolNums=128
```

```
rm -rf broker-a.properties
rm -rf broker-b-s.properties
```

11. 检查启动内存

```
vim /rocketmq/bin/runbroker.sh
```

```
# 开发环境配置 JVM Configuration
JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m"
```

12. 启动服务器（在bin目录下依次启动）

129上

```
nohup sh mqnamesrv &
```

```
nohup sh mqbroker -c ../conf/2m-2s-sync/broker-a.properties &
```

```
nohup sh mqbroker -c ../conf/2m-2s-sync/broker-b-s.properties &
```

130上

```
nohup sh mqnamesrv &
```

```
nohup sh mqbroker -c ../conf/2m-2s-sync/broker-a-s.properties &
```

```
nohup sh mqbroker -c ../conf/2m-2s-sync/broker-b.properties &
```

4.3 rocketmq-console集群监控平台搭建

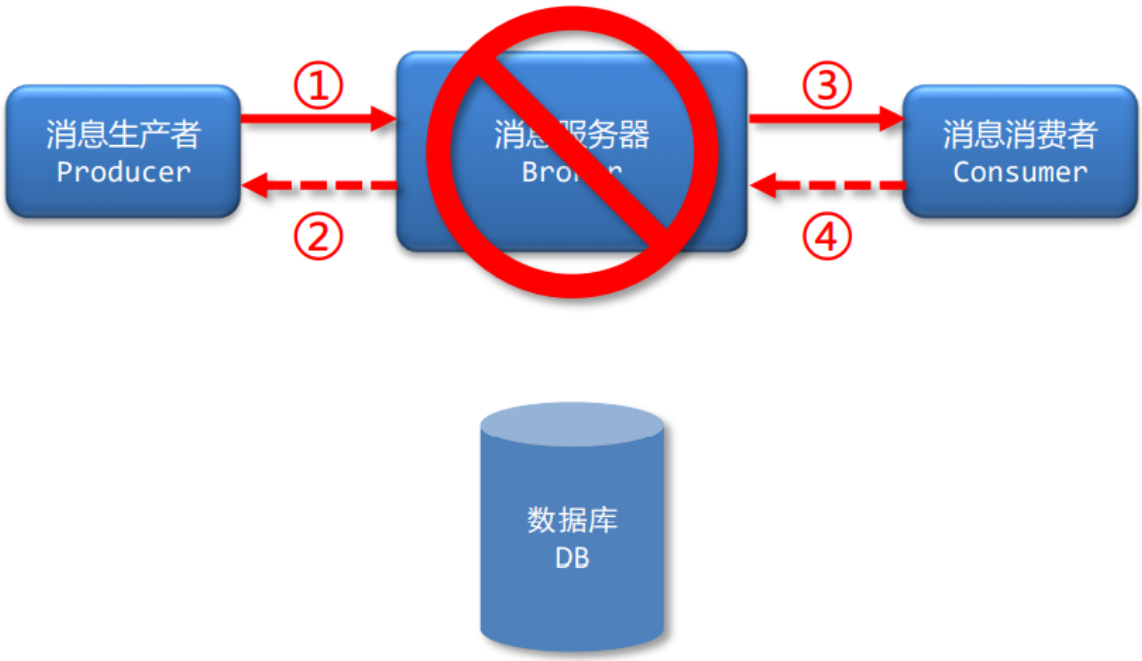
1. incubator-rocketmq-externals是一个基于rocketmq的基础之上扩展开发的开源项目
2. 获取地址：https://github.com/apache/rocketmq-externals
3. rocketmq-console是一款基于java环境开发的（springboot）的管理控制台工具

3. 高级特性（重点）

3.1 消息的存储

1. 消息生成者发送消息到MQ
2. MQ返回ACK给生产者
3. MQ push 消息给对应的消费者
4. 消息消费者返回ACK给MQ

说明：ACK（Acknowledge character）



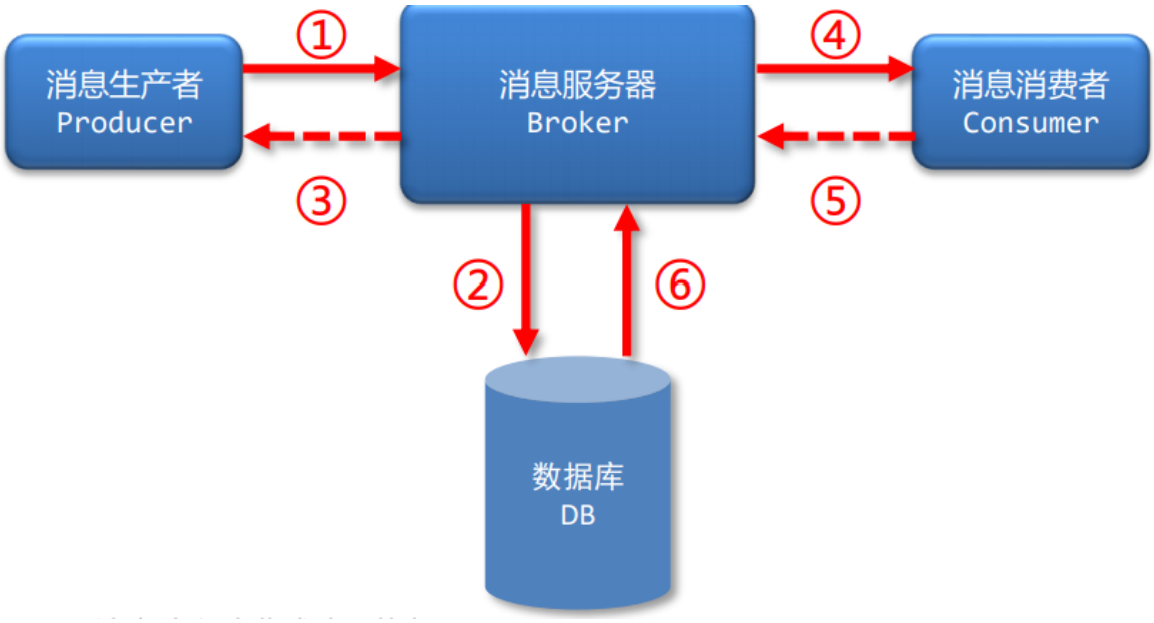
3.2 消息的存储

1. 消息生成者发送消息到MQ
2. MQ收到消息，将消息进行持久化，存储该消息
3. MQ返回ACK给生产者
4. MQ push 消息给对应的消费者
5. 消息消费者返回ACK给MQ

6. MQ删除消息

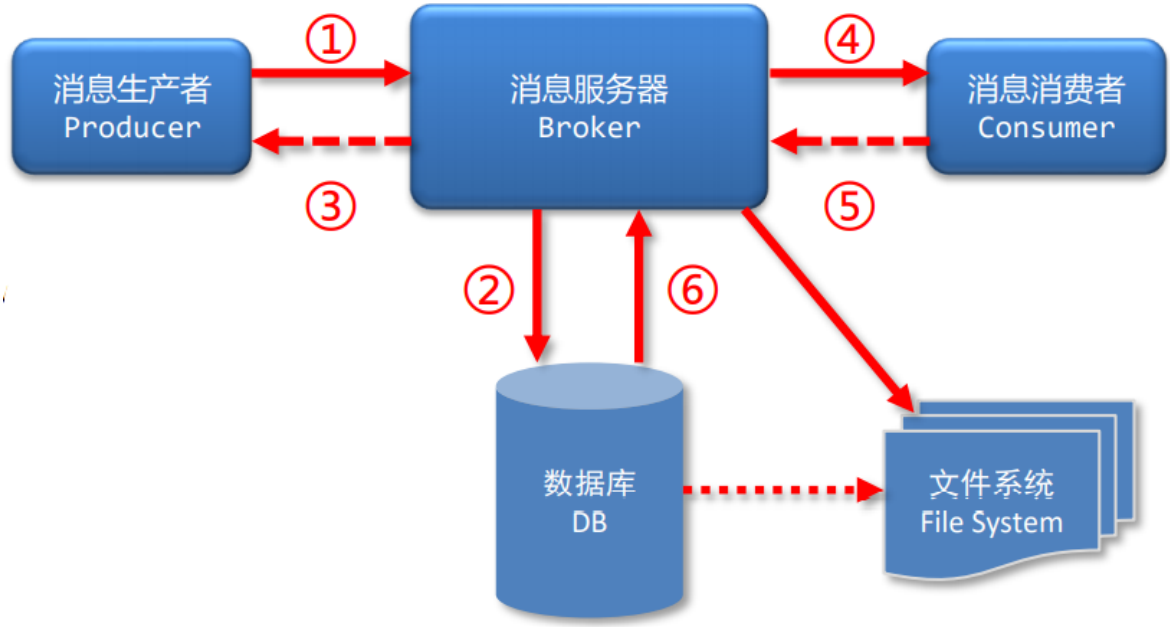
注意：

1. 第⑤步MQ在指定时间内接到消息消费者返回ACK，MQ认定消息消费成功，执行⑥
2. 第⑤步MQ在指定时间内未接到消息消费者返回ACK，MQ认定消息消费失败，重新执行④⑤⑥



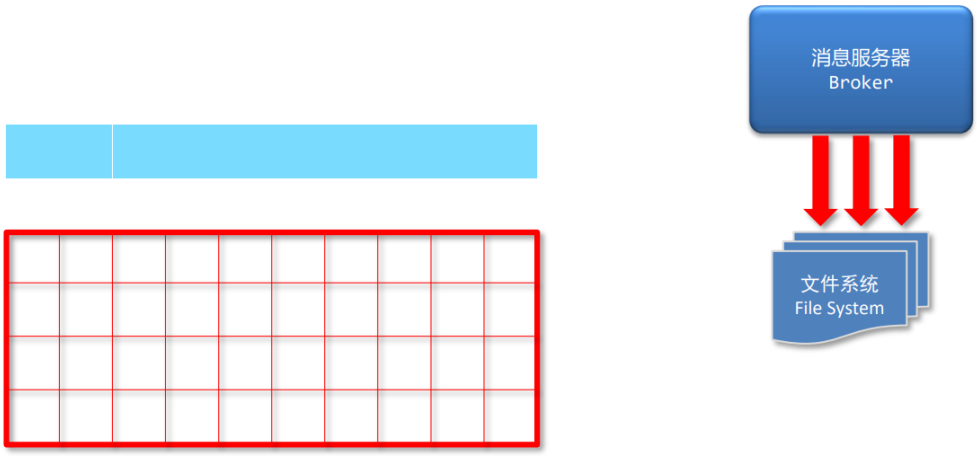
3.3 消息的存储介质

1. 数据库
    1. ActiveMQ
    2. 缺点：数据库瓶颈将成为MQ瓶颈
2. 文件系统
    1. RocketMQ/Kafka/RabbitMQ
    2. 解决方案：采用消息刷盘机制进行数据存储
    3. 缺点：硬盘损坏的问题无法避免

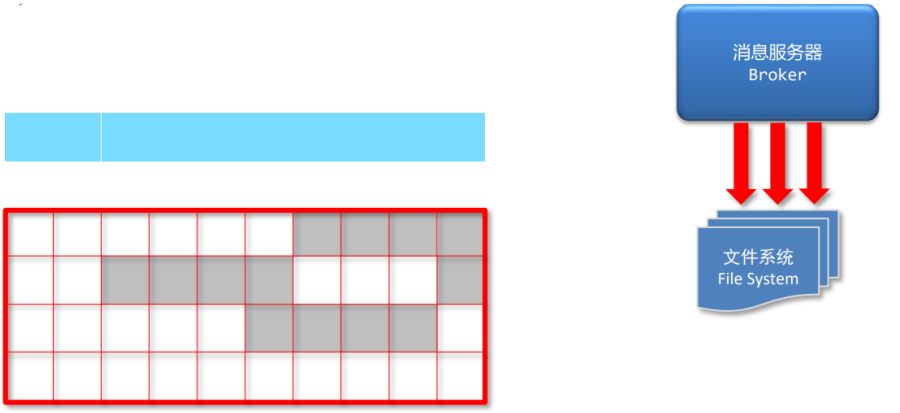


3.4 高效的消息存储与读写方式

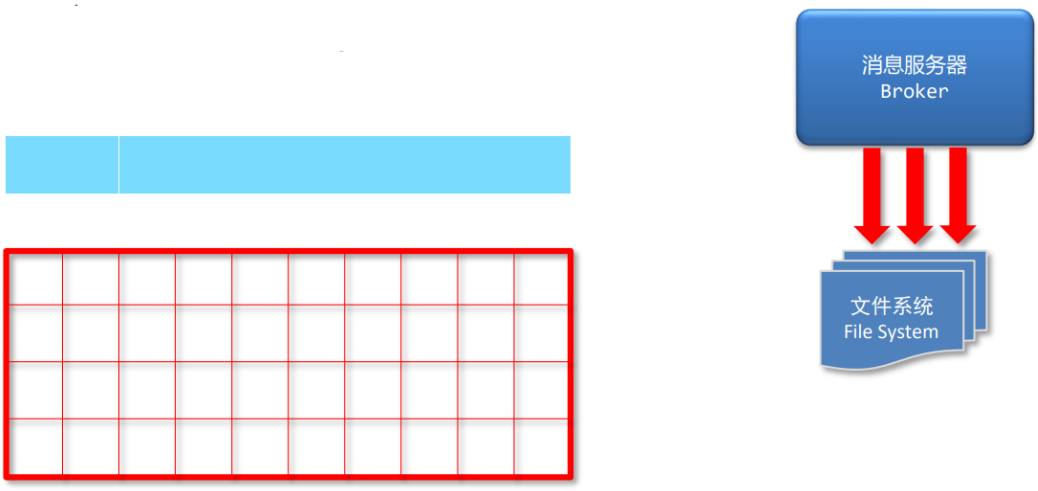
1. SSD（Solid State Disk）

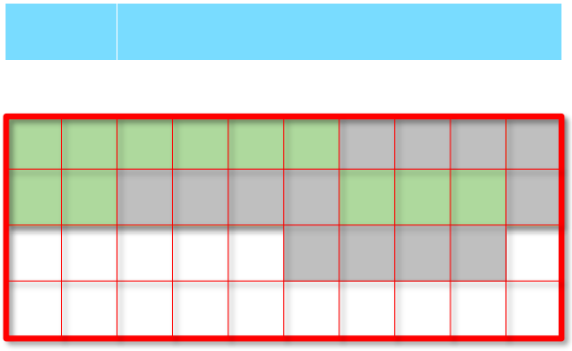


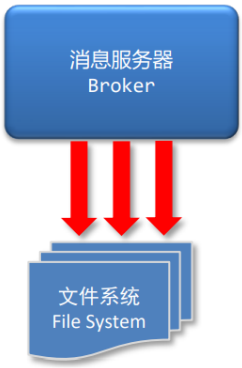
1. 随机写（100KB/s）



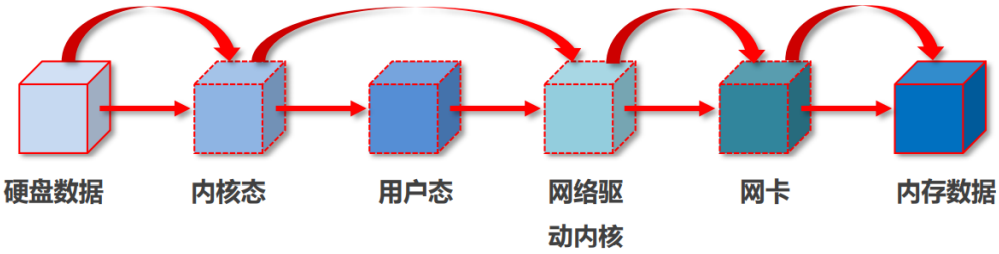
2. 顺序写 （600MB/s）1秒1部电影







2. Linux系统发送数据的方式
3. “零拷贝”技术
    1. 数据传输由传统的4次复制简化成3次复制，减少1次复制过程
    2. Java语言中使用MappedByteBuffer类实现了该技术
    3. 要求：预留存储空间，用于保存数据（1G存储空间起步）



3.5 消息存储结构

1. MQ数据存储区域包含如下内容
    1. 消息数据存储区域
        1. topic
        2. queueId
        3. message
    2. 消费逻辑队列
        1. minOffset
        2. maxOffset
        3. consumerOffset
    3. 索引
        1. key索引
        2. 创建时间索引

3.6 刷盘机制

1. 同步刷盘
    1. 生产者发送消息到MQ，MQ接到消息数据
    2. MQ挂起生产者发送消息的线程
    3. MQ将消息数据写入内存
    4. 内存数据写入硬盘
    5. 磁盘存储后返回SUCCESS
    6. MQ恢复挂起的生产者线程
    7. 发送ACK到生产者

2. 异步刷盘
    1. 生产者发送消息到MQ，MQ接到消息数据
    2.
    3. MQ将消息数据写入内存
    4.
    5.
    6.
    7. 发送ACK到生产者

1. 同步刷盘：安全性高，效率低，速度慢（适用于对数据安全要求较高的业务）
2. 异步刷盘：安全性低，效率高，速度快（适用于对数据处理速度要求较高的业务）

配置方式

```
#刷盘方式
#- ASYNC_FLUSH 异步刷盘
#- SYNC_FLUSH 同步刷盘
flushDiskType=SYNC_FLUSH
```

5.7 高可用性

1. nameserver
    1. 无状态+全服务器注册
2. 消息服务器
    1. 主从架构（2M-2S）
3. 消息生产
    1. 生产者将相同的topic绑定到多个group组，保障master挂掉后，其他master仍可正常进行消息接收
4. 消息消费
    1. RocketMQ自身会根据master的压力确认是否由master承担消息读取的功能，当master繁忙时候，自动切换由slave承担数据读取的工作

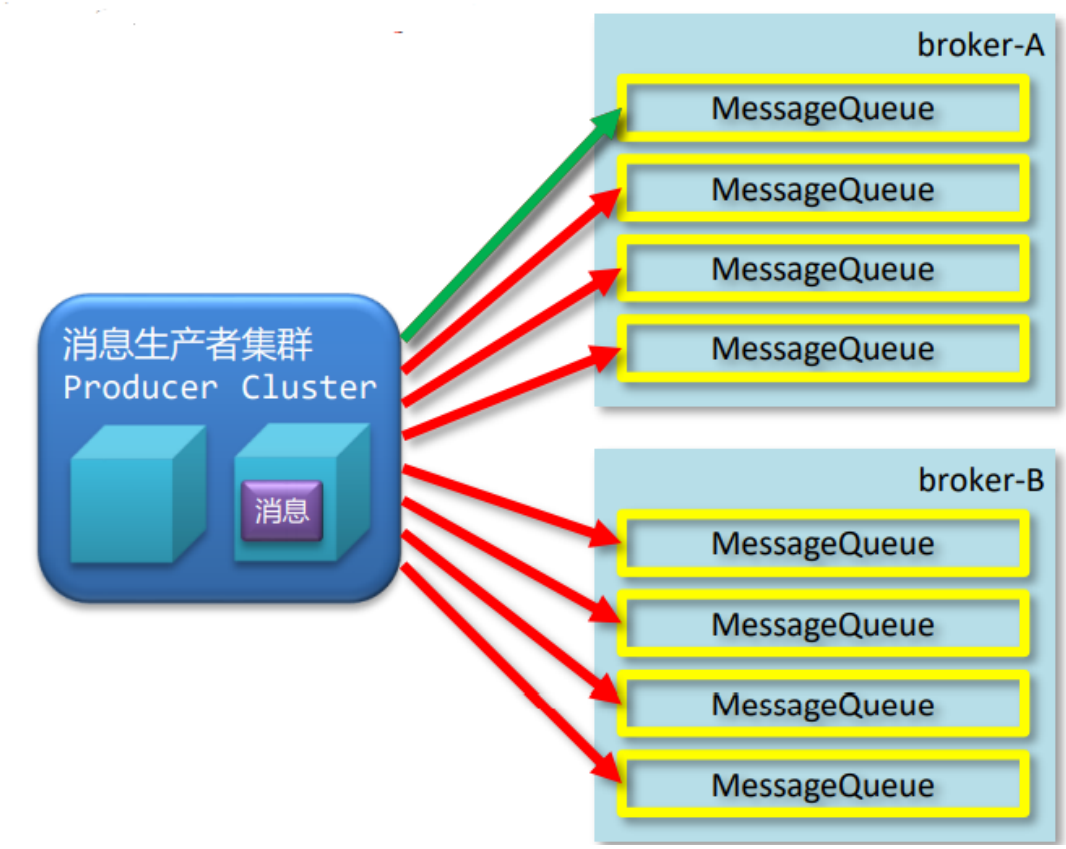
5.8 主从数据复制

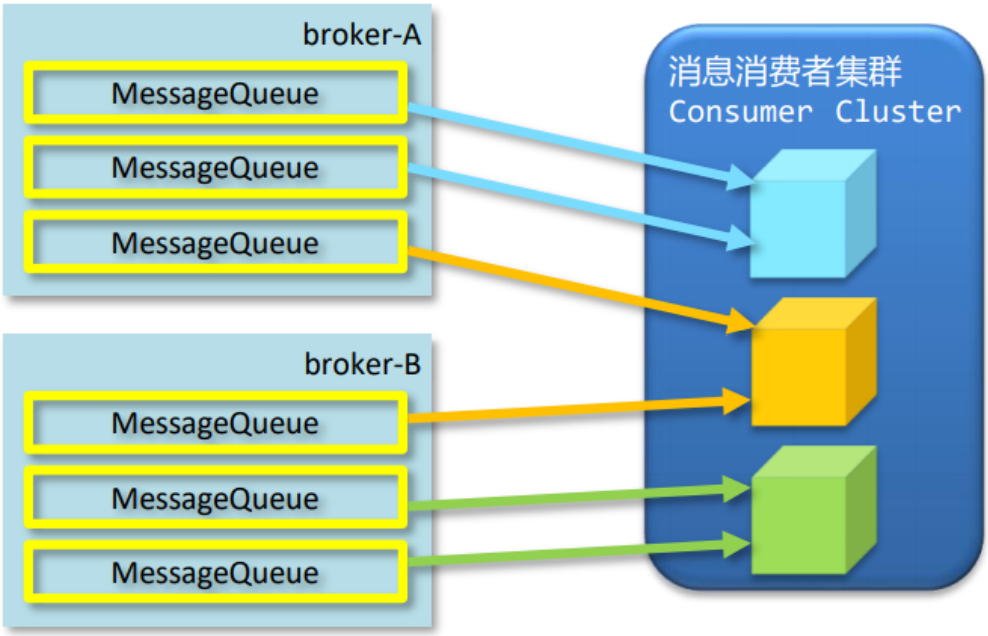
1. 同步复制
    1. master接到消息后，先复制到slave，然后反馈给生产者写操作成功
    2. 优点：数据安全，不丢数据，出现故障容易恢复
    3. 缺点：影响数据吞吐量，整体性能低
2. 异步复制
    1. master接到消息后，立即返回给生产者写操作成功，当消息达到一定量后再异步复制到slave
    2. 优点：数据吞吐量大，操作延迟低，性能高
    3. 缺点：数据不安全，会出现数据丢失的现象，一旦master出现故障，从上次数据同步到故障时间的数据将丢失
3. 配置方式

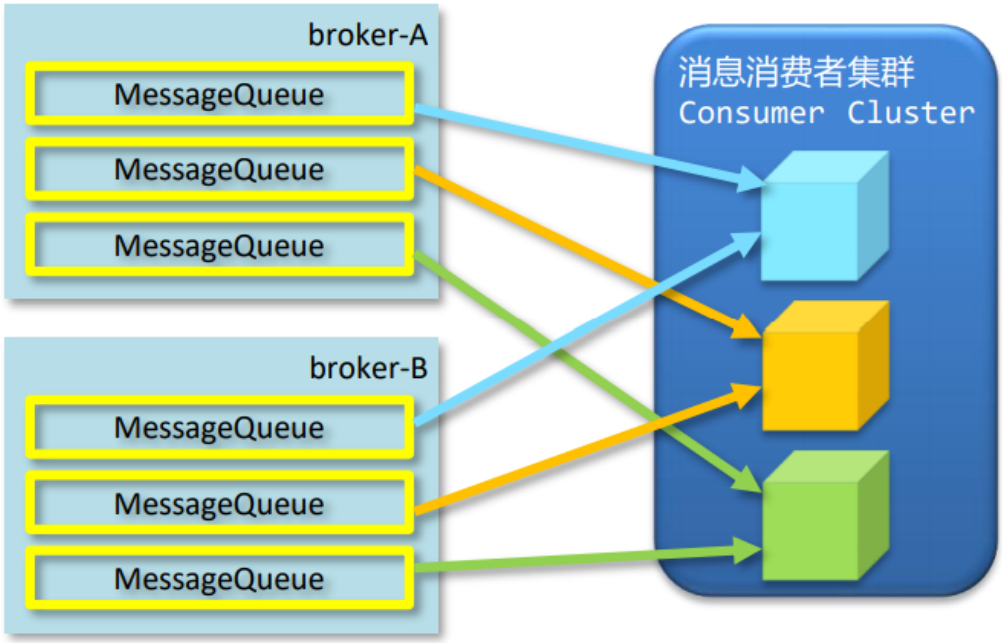
```
#Broker 的角色
#- ASYNC_MASTER 异步复制Master
#- SYNC_MASTER 同步双写Master
#- SLAVE
brokerRole=SYNC_MASTER
```

5.9 负载均衡

1. Producer负载均衡
    1. 内部实现了不同broker集群中对同一topic对应消息队列的负载均衡
2. Consumer负载均衡
    1. 平均分配
    2. 循环平均分配
3. 广播模式（不参与负载均衡）



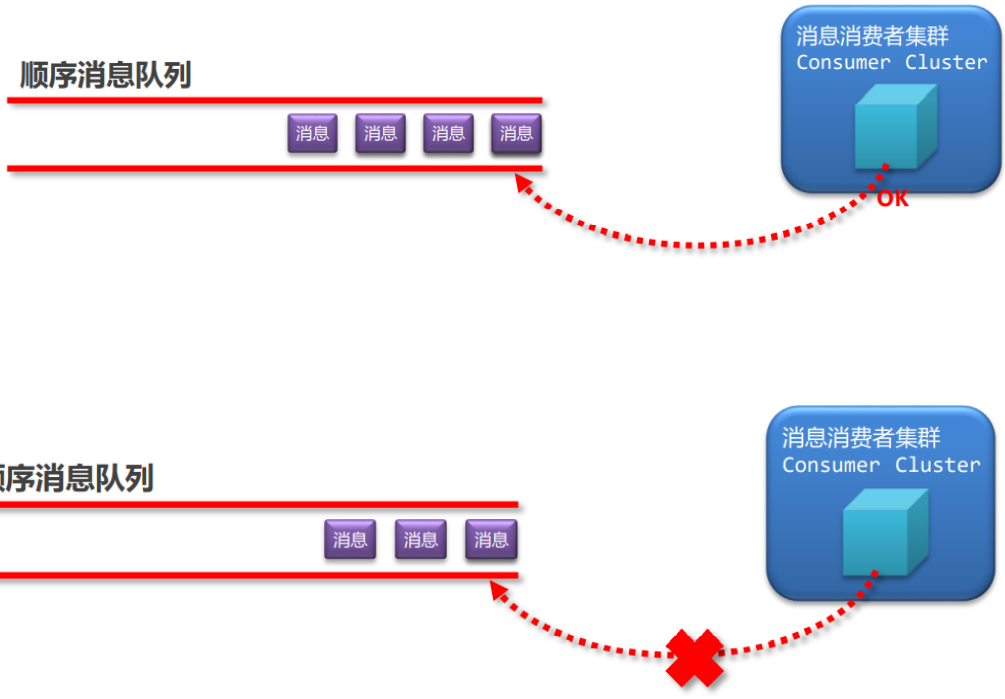




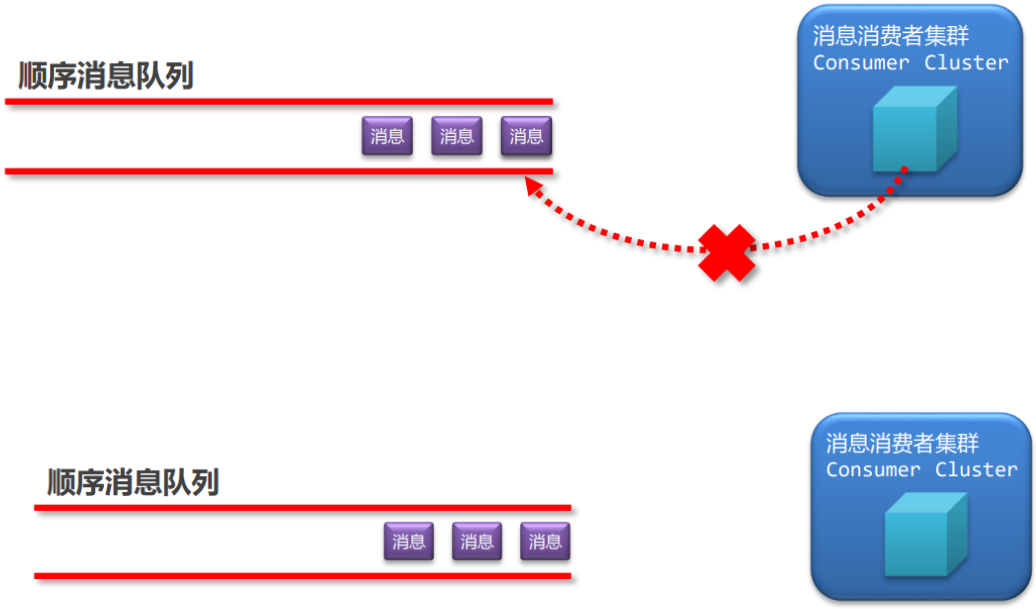
5.10 消息重试

1. 当消息消费后未正常返回消费成功的信息将启动消息重试机制
2. 消息重试机制
    1. 顺序消息
    2. 无序消息

5.10.1 顺序消息重试

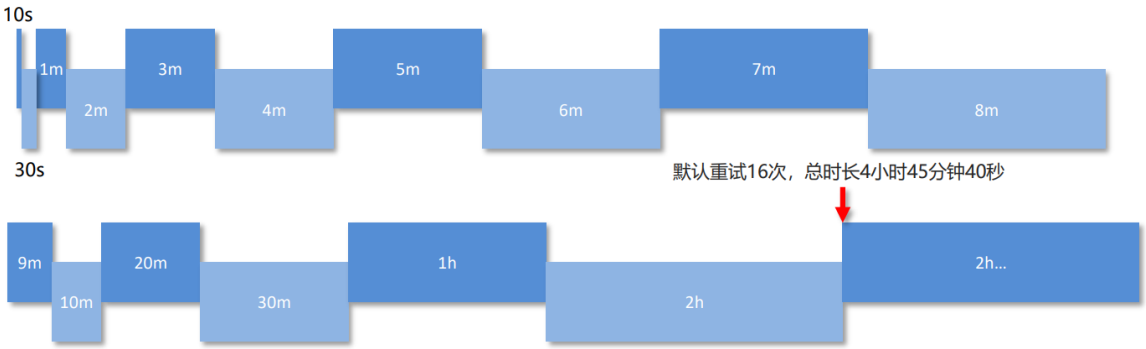


1. 当消费者消费消息失败后，RocketMQ会自动进行消息重试（每次间隔时间为 1 秒）
2. 注意：应用会出现消息消费被阻塞的情况，因此，要对顺序消息的消费情况进行监控，避免阻塞现象的发生



5.10.2 无序消息重试

1. 无序消息包括普通消息、定时消息、延时消息、事务消息
2. 无序消息重试仅适用于负载均衡（集群）模型下的消息消费，不适用于广播模式下的消息消费
3. 为保障无序消息的消费，MQ设定了合理的消息重试间隔时长



5.11 死信队列

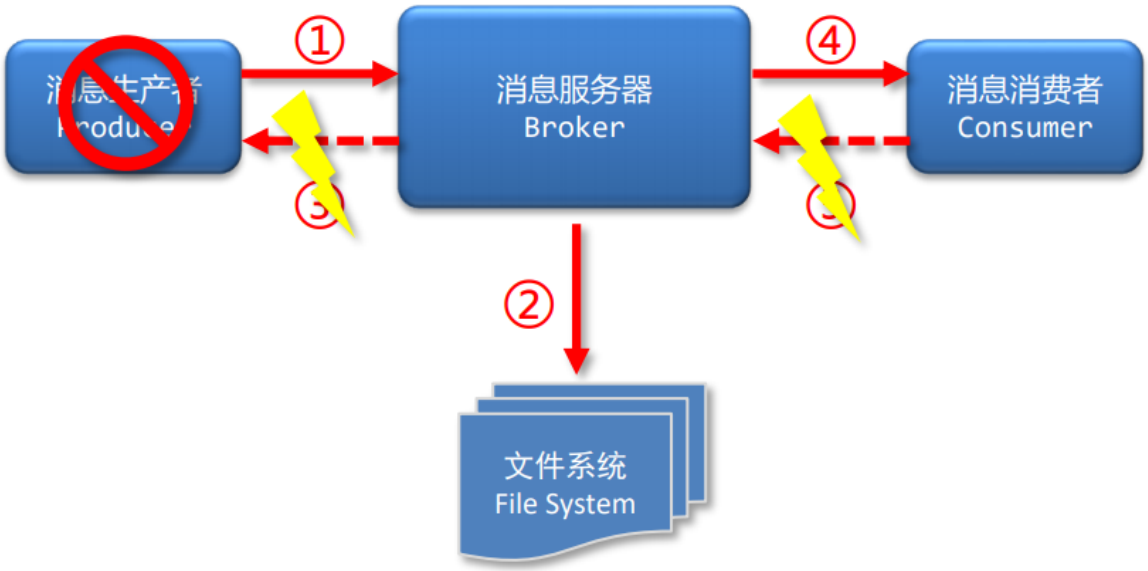
1. 当消息消费重试到达了指定次数（默认16次）后，MQ将无法被正常消费的消息称为死信消息（Dead-Letter Message）
2. 死信消息不会被直接抛弃，而是保存到了一个全新的队列中，该队列称为死信队列（Dead-Letter Queue）\
3. 死信队列特征
    1. 归属某一个组（Gourp Id），而不归属Topic，也不归属消费者
    2. 一个死信队列中可以包含同一个组下的多个Topic中的死信消息
    3. 死信队列不会进行默认初始化，当第一个死信出现后，此队列首次初始化
4. 死信队列中消息特征
    1. 不会被再次重复消费
    2. 死信队列中的消息有效期为3天，达到时限后将被清除

5.12 死信处理

1. 在监控平台中，通过查找死信，获取死信的messageId，然后通过id对死信进行精准消费

5.13 消息重复消费

1. 消息重复消费原因
    1. 生产者发送了重复的消息
        1. 网络闪断
        2. 生产者宕机
    2. 消息服务器投递了重复的消息
        1. 网络闪断
    3. 动态的负载均衡过程
        1. 网络闪断/抖动
        2. broker重启
        3. 订阅方应用重启（消费者）
        4. 客户端扩容
        5. 客户端缩容



5.14 消息幂等

1. 对同一条消息，无论消费多少次，结果保持一致，称为消息幂等性
2. 解决方案
    1. 使用业务id作为消息的key
    2. 在消费消息时，客户端对key做判定，未使用过放行，使用过抛弃
3. 注意：messageId由RocketMQ产生，messageId并不具有唯一性，不能作用幂等判定条件
4. 常见的幂等方法示例

•新增:不幂等 insert into order values (.....)

•查询:幂等

•删除:幂等 delete from 表 where id =1

•修改:不幂等 update account set balance = balance+100 where no=1

•修改:幂等 update account set balance =100 where no=1

计算机考研：

1计算机组成原理：

2网络：3次握手 4次挥手

3操作系统：linux

4数据结构-算法：tree b+ 链表

执行力