读书笔记1 接受和发送消息

DefaultMQPushConsumer

- 示例代码

```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer( consumerGroup: "group_name_test");
consumer.setNamesrvAddr("name-server-1-ip:9876;name-server-2-ip:9876");
consumer.setMessageModel(MessageModel.CLUSTERING); // MessageModel.BROADCASTING
consumer.subscribe( topic: "topicTest",  subExpression: "*");
consumer.registerMessageListener(new MessageListenerConcurrently() {
    @Override
    public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {
        System.out.println(Thread.currentThread().getName() + " receive new message" + msgs + "%n");
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
});
consumer.start();
```

多个consumer组合，提高并发，配合MessageModel使用

消息模式 集群/广播

clustering模式下，同一个ConsumerGroup里每个consumer 只消费订阅的一部分内容，
一个consumerGroup所有consumer消费内容合并才是 Topic内容，从而达到负载均衡的目的

nameServer：多个ip端口 通过 “；”隔开

Topic用来标识消息类型，需要提前创建
也可以过滤消息，例如：
consumer.subscribe(**"topicTest"**, **“tag1|| tag2 || tag3”**);
表示TopicTest下， 带有tag1、2、3的消息， null /“*”表示所有消息

1. 处理流程

DefaultMQPushConsumer 的处理流程主要在 DefaultMQPushConsumerImpl

```
/**
 * 消费者分组从指定的消费队列(MessageQueue)拉取消息（一个PullRequest 对应一个消费者分组对topic的一个队列的消费。）
 * 在按topic 做rebalance操作的时候会被触发一次。
 *
 * 所以， 这里总结下来，每一个消费者分组对topic的某一个队列进行消费，是通过rebalance操作来触发的，
 * 而rebalance操作又是由消费者的加入，退出，订阅和取消订阅来触发的。
 * 一旦消费了队列， 其实就是拉一批消费一批再拉下一批，循环往复。
 *
 *
 * @param pullRequest
 */
public void pullMessage(final PullRequest pullRequest) {
```

com.alibaba.rocketmq.client.impl.consumer.DefaultMQPushConsumerImpl#pullMessage

PushConsumer中通过PullRequest， 通过长轮询兼容pull 和push的优点

长轮询源码

org.apache.rocketmq.broker.longpolling.PullRequestHoldService#run

```
if (this.brokerController.getBrokerConfig().isLongPollingEnable()) {
    this.waitForRunning( interval: 5 * 1000);
} else {
    this.waitForRunning(this.brokerController.getBrokerConfig().getShortPollingTimeMills());
}

long beginLockTimestamp = this.systemClock.now();
this.checkHoldRequest();
long costTime = this.systemClock.now() - beginLockTimestamp;
if (costTime > 5 * 1000) {
    log.info("[NOTIFYME] check hold request cost {} ms.", costTime);
}
```

```
private void checkHoldRequest() {
    for (String key : this.pullRequestTable.keySet()) {
        String[] kArray = key.split(TOPIC_QUEUEID_SEPARATOR);
        if (2 == kArray.length) {
            String topic = kArray[0];
            int queueId = Integer.parseInt(kArray[1]);
            final long offset = this.brokerController.getMessageStore().getMaxOffsetInQueue(topic, queueId);
            try {
                this.notifyMessageArriving(topic, queueId, offset);
            } catch (Throwable e) {
                log.error( var1: "check hold request failed. topic={}, queueId={}", topic, queueId, e);
            }
        }
    }
}
```

没有新消息 不回急于返回，通过循环不断查看状态， 每次5s， 第三次还没有 返回空。

等待过程中，接收到新消息 **notifyMessageArriving 返回结果**

长轮询的核心是 消息不是接受后立即返回，等待5s 先hold， 有新请求到达，就利用现有连接 立刻返回consumer

主动权还是在Consumer， Broker中有大量积压，也不会主动推给Consumer

Hold住Consumer请求时，需要占用资源， 适合消息队列这种客户端数量可控的场景

2、 DefaultMqPushConsumer 流量控制

pushComsumer的核心还是 Pull， 能根据自身处理速度调整消息速度。

DefaultMQPushConsumerImpl
private ConsumeMessageService **consumeMessageService**;

```
public ConsumeMessageOrderlyService(DefaultMQPushConsumerImpl defaultMQPushConsumerImpl,
    MessageListenerOrderly messageListener) {
    this.defaultMQPushConsumerImpl = defaultMQPushConsumerImpl;
    this.messageListener = messageListener;

    this.defaultMQPushConsumer = this.defaultMQPushConsumerImpl.getDefaultMQPushConsumer();
    this.consumerGroup = this.defaultMQPushConsumer.getConsumerGroup();
    this.consumeRequestQueue = new LinkedBlockingQueue<Runnable>();

    this.consumeExecutor = new ThreadPoolExecutor(
        this.defaultMQPushConsumer.getConsumeThreadMin(),
        this.defaultMQPushConsumer.getConsumeThreadMax(),
        keepAliveTime: 1000 * 60,
        TimeUnit.MILLISECONDS,
        this.consumeRequestQueue,
        new ThreadFactoryImpl( threadNamePrefix: "ConsumeMessageThread_"));

    this.scheduledExecutorService = Executors.newSingleThreadScheduledExecutor(new ThreadFactoryImpl( threadNamePrefix: "Consum
}
```

Pull的消息直接到线程池很难监控、管理
ROcketMq 定义了 快照类： ProcessQueue来解决， PushConsumer运行时，每个Message Queue都有对应的ProcessQueue对象来保存消息处理状态的快照

ProcessQueue对象主要是一个TreeMap 和一个 读写锁
TreeMap以 offset 为key， 消息内容为value； 读写锁控制并发访问
客户端每次pull请求前 都会进行3个判断控制流量
org.apache.rocketmq.client.impl.consumer.DefaultMQPushConsumerImpl#pullMessage

```
long cachedMessageCount = processQueue.getMsgCount().get();
long cachedMessageSizeInMiB = processQueue.getMsgSize().get() / (1024 * 1024);

if (cachedMessageCount > this.defaultMQPushConsumer.getPullThresholdForQueue()) {
    this.executePullRequestLater(pullRequest, PULL_TIME_DELAY_MILLS_WHEN_FLOW_CONTROL);
    if ((queueFlowControlTimes++ % 1000) == 0) {
        log.warn(
            var1: "the cached message count exceeds the threshold {}, so do flow control, minOffset={}, maxOffset={}, cc
            this.defaultMQPushConsumer.getPullThresholdForQueue(), processQueue.getMsgTreeMap().firstKey(), processQueu
    }
    return;
}

if (cachedMessageSizeInMiB > this.defaultMQPushConsumer.getPullThresholdSizeForQueue()) {
    this.executePullRequestLater(pullRequest, PULL_TIME_DELAY_MILLS_WHEN_FLOW_CONTROL);
    if ((queueFlowControlTimes++ % 1000) == 0) {
        log.warn(
            var1: "the cached message size exceeds the threshold {} MiB, so do flow control, minOffset={}, maxOffset={},
            this.defaultMQPushConsumer.getPullThresholdSizeForQueue(), processQueue.getMsgTreeMap().firstKey(), process
    }
    return;
}

if (!this.consumeOrderly) {
    if (processQueue.getMaxSpan() > this.defaultMQPushConsumer.getConsumeConcurrentlyMaxSpan()) {
        this.executePullRequestLater(pullRequest, PULL_TIME_DELAY_MILLS_WHEN_FLOW_CONTROL);
        if ((queueMaxSpanFlowControlTimes++ % 1000) == 0) {
            log.warn(
                var1: "the queue's messages, span too long, so do flow control, minOffset={}, maxOffset={}, maxSpan={},
                processQueue.getMsgTreeMap().firstKey(), processQueue.getMsgTreeMap().lastKey(), processQueue.getMaxSpa
                pullRequest, queueMaxSpanFlowControlTimes);
        }
        return;
    }
} else {
    if (processQueue.isLocked()) {
        if (!pullRequest.isLockedFirst()) {
```

会判断获取但未处理的消息个数、消息总大小、offset跨度
任何一个超过 就隔一段时间再拉取，从而控制流量

DefaultPullConsumer

Simple中代码作为范例： 除了各种参数外 主要还有3件事

```
    DefaultMQPullConsumer consumer = new DefaultMQPullConsumer("please_rename_unique_group_name_5");
    consumer.setNamesrvAddr("127.0.0.1:9876");
    consumer.start();

    Set<MessageQueue> mqs = consumer.fetchSubscribeMessageQueues("broker-a");
    for (MessageQueue mq : mqs) {                                    获取message queue 并遍历
        System.out.printf("Consume from the queue: %s%n", mq);
        SINGLE_MQ:
        while (true) {
            try {
                PullResult pullResult =
                    consumer.pullBlockIfNotFound(mq, null, getMessageQueueOffset(mq), 32);
                System.out.printf("%s%n", pullResult);
                putMessageQueueOffset(mq, pullResult.getNextBeginOffset());          维护 offset
                switch (pullResult.getPullStatus()) {
                    case FOUND:
                        break;
                    case NO_MATCHED_MSG:
                        break;
                    case NO_NEW_MSG:                                 根据不同消息状态处理
                        break SINGLE_MQ;
                    case OFFSET_ILLEGAL:
                        break;
                    default:
                        break;
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    consumer.shutdown();
}

private static long getMessageQueueOffset(MessageQueue mq) {
    Long offset = OFFSE_TABLE.get(mq);
    if (offset != null)
        return offset;

    return 0;
}

private static void putMessageQueueOffset(MessageQueue mq, long offset) {
    OFFSE_TABLE.put(mq, offset);
}
```

Consumer的启动、 关闭

Consumer分为 push 、 pull两种方式
PullConsumer 主动权较高，根据实际需要暂停、停止、启动消费， 需要注意offset保存，异常处理部分需要offset写入磁盘
记准了每个Message queue的offset 才能保证消费准确性

DefaultPushConsumer的退出， 要调用。shutdown() ,释放资源 、保存offset，需要加到consumer所在应用的退出逻辑

此外，PushConsumer 启动的时候 ，会进行配置检查，然后连接NameServer，获取topic

为什么NameServer 错的情况下 仍然不报错退出?

因为RocketMq集群 认为可以有多个NameServer、Broker，某个机器异常 整体仍然可用，
所以某个连接异常，只是不断重试， 不会退出

生产者： 不同场景 有不同写入策略 同步、异步、延迟、消息事务等

```
// groupName
DefaultMQProducer producer = new DefaultMQProducer( producerGroup: "unique_group_name");
// 一个jvm启动多个producer时，通过instanceName区分，默认default
producer.setInstanceName("instance1");
// 发送失败重试次数
producer.setRetryTimesWhenSendFailed(3);
// nameserver
producer.setNamesrvAddr("name_server1_ip:9876;name_server2_ip:9876");
producer.start();

for (int i = 0; i < 100; i++) {
    try {
        Message msg = new Message( topic: "TopicTest",  tags: "taga", ("Hello rocketMq" + i).getBytes());
        producer.send(msg, new SendCallback() {
            @Override
            public void onSuccess(SendResult sendResult) { }
            @Override
            public void onException(Throwable e) { }
        });
    } catch (RemotingException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
producer.shutdown();
```

同步还是异步，这里是异步

延迟消息：
Broker 收到后延迟一段时间再处理。
创建Message时， setDelayTimeLevel(int level) 1s/5s/10s/30s/1m/2m/..../2h 等

自定义消息发送规则：
因为一个Topic 有多个Message Queue，如果是Producer的默认配置，会轮流想各个 Message Queue发。
Consumer消费时，会经过负载均衡，不特定设置 则被消费情况未知
使用 MessageQueueSelector， 自定一个实现，自定义参数

对事务的支持：

rocketMq 通过两阶段方式实现事务

案例：A银行账户 转1000-》B银行， A扣 和B增加同时成功、同时失败

TransactionMQProducer 处理流程：
1、先发一个B增加1000 消息；
2、成功后，做从A扣1000操作；
3、根据操作2 的结果，确定操作1 时 commit 还是rollback

具体流程：
1、发送方 向 rocketMq 发送待确认消息
2、 rocketmq 将 待确认消息 持久化， 并给发送方 success。===》 1阶段完成
3、 发送方执行本地逻辑
4、发送方根据本地执行结果 向rocketmq 发 二次确认（commit / rollback）
Commit 则将1阶段标记为“可投递”； rollback 则删除一阶段消息
5、异常情况下， 步骤4二次确认未到达 rocketMq， 服务器若干时间后 回查
6、发送方收到回查， 检查本地逻辑，回复 rollback / commit
7、 rocketmq 收到回查，返回步骤4

如何存储、 调整Offset

rocketMq中，一种类型消息 再一个Topic ， 为了并行，一般一个Topic 有多个Messagequeue，
Offset 是某个Topic 下的一条消息 在 某个 Message Queue中的位置，通过offset定位消息，指示consumer往后处理

offset 分 本地文件、 Broker代存两种

对于**DefaultmqPushConsumer**，默认是Clustering模式，同一个Consumer group 多个消费者，每个人消费一部分，各自消息不同
Broker 存储和控制offset， 使用 org.apache.rocketmq.client.consumer.store.RemoteBrokerOffsetStore

```
private long fetchConsumeOffsetFromBroker(MessageQueue mq) throws RemotingException, MQBrokerException,
```

DefaultmqPushConsumer 的BroadCasting模式下，每个consumer都收到Topic全部消息，互不干扰，
个自使用 LocalFileOffsetStore 存储 offset到本地

自定义存储offset 参考 LocalFileOffsetStore

```
public void load() throws MQClientException {
    OffsetSerializeWrapper offsetSerializeWrapper = this.readLocalOffset();
    if (offsetSerializeWrapper != null && offsetSerializeWrapper.getOffsetTable() != null) {
        offsetTable.putAll(offsetSerializeWrapper.getOffsetTable());

        for (MessageQueue mq : offsetSerializeWrapper.getOffsetTable().keySet()) {
            AtomicLong offset = offsetSerializeWrapper.getOffsetTable().get(mq);
            log.info( var1: "load consumer's offset, {} {} {}",
                this.groupName,
                mq,
                offset.get());
        }
    }
}
```

```
public void updateOffset(MessageQueue mq, long offset, boolean increaseOnly) {
    if (mq != null) {
        AtomicLong offsetOld = this.offsetTable.get(mq);
        if (null == offsetOld) {
            offsetOld = this.offsetTable.putIfAbsent(mq, new AtomicLong(offset));
        }

        if (null != offsetOld) {
            if (increaseOnly) {
                MixAll.compareAndIncreaseOnly(offsetOld, offset);
            } else {
                offsetOld.set(offset);
            }
        }
    }
}
```

```java
@Override
public long readOffset(final MessageQueue mq, final ReadOffsetType type) {
    if (mq != null) {
        switch (type) {
            case MEMORY_FIRST_THEN_STORE:
            case READ_FROM_MEMORY: {
                AtomicLong offset = this.offsetTable.get(mq);
                if (offset != null) {
                    return offset.get();
                } else if (ReadOffsetType.READ_FROM_MEMORY == type) {
                    return -1;
                }
            }
            case READ_FROM_STORE: {
                OffsetSerializeWrapper offsetSerializeWrapper;
                try {
                    offsetSerializeWrapper = this.readLocalOffset();
                } catch (MQClientException e) {
                    return -1;
                }
                if (offsetSerializeWrapper != null && offsetSerializeWrapper.getOffsetTable() != null) {
                    AtomicLong offset = offsetSerializeWrapper.getOffsetTable().get(mq);
                    if (offset != null) {
                        this.updateOffset(mq, offset.get(), increaseOnly: false);
                        return offset.get();
                    }
                }
            }
            default:
                break;
        }
    }

    return -1;
}
```

```java
@Override
public void persistAll(Set<MessageQueue> mqs) {
    if (null == mqs || mqs.isEmpty())
        return;

    OffsetSerializeWrapper offsetSerializeWrapper = new OffsetSerializeWrapper();
    for (Map.Entry<MessageQueue, AtomicLong> entry : this.offsetTable.entrySet()) {
        if (mqs.contains(entry.getKey())) {
            AtomicLong offset = entry.getValue();
            offsetSerializeWrapper.getOffsetTable().put(entry.getKey(), offset);
        }
    }

    String jsonString = offsetSerializeWrapper.toJson( prettyFormat: true);
    if (jsonString != null) {
        try {
            MixAll.string2File(jsonString, this.storePath);
        } catch (IOException e) {
            log.error("persistAll consumer offset Exception, " + this.storePath, e);
        }
    }
}
```