

魔法咪路 Lv1

2019年05月10日 阅读 78

已关注

## QMQ源码分析之Actor

### 前言

QMQ有关actor的一篇[文章](#)阐述了actor的应用场景。即client消费消息的请求会先进入一个RequestQueue，在client消费消息时，往往存在多个主题、多个消费组共享一个RequestQueue消费消息。在这个Queue中，存在不同主题的有不同消费组数量，以及不同消费组有不同consumer数量，那么就会存在抢占资源的情况。举个[文章](#)中的例子，一个主题下有两个消费组A和B，A有100个consumer，B有200个consumer，那么在RequestQueue中来自B的请求可能会多于A，这个时候就存在消费unfair的情况，所以需要隔离不同主题不同消费组以保证fair。除此之外，当consumer消费能力不足，造成broker消息堆积，这个时候就会导致consumer所在消费组总在消费"老消息"，影响全局整体的一个消费能力。因为"老消息"不会存在page cache中，这个时候很可能就会从磁盘load，那么表现是RequestQueue中来自消费"老消息"消费组的请求处理时间过长，影响到其他主题消费组的消费，因此这个时候也需要做策略来避免不同消费组的相互影响。所以QMQ就有了actor机制，以消除各个消费组之间因消费能力不同、consumer数量不同而造成的相互影响各自的消费能力。

## PullMessageWorker

要了解QMQ的actor模式是如何起作用的，就先来看看Broker是如何处理消息拉取请求的。

```
class PullMessageWorker implements ActorSystem.Processor<PullMessageProcessor.PullEntry> {  
    // 消息存储层  
    private final MessageStoreWrapper store;  
    // actor  
    private final ActorSystem actorSystem;  
  
    private final ConcurrentMap<String, ConcurrentMap<String, Object>> subscribers;  
  
    PullMessageWorker(MessageStoreWrapper store, ActorSystem actorSystem) {  
        this.store = store;  
        this.actorSystem = actorSystem;  
    }  
}
```

java





首页 ▾

搜索更新啦



```
void pull(PullMessageProcessor.PullEntry pullEntry) {
    // subject+group作actor调度粒度
    final String actorPath = ConsumerGroupUtils.buildConsumerGroupKey(pullEntry.subject, pullEntry.group);
    // actor调度
    actorSystem.dispatch(actorPath, pullEntry, this);
}

@Override
public boolean process(PullMessageProcessor.PullEntry entry
    , ActorSystem.Actor<PullMessageProcessor.PullEntry> self) {
    QMon.pullQueueTime(entry.subject, entry.group, entry.pullBegin);

    //开始处理请求的时候就过期了，那么就直接不处理了，也不返回任何东西给客户端，客户端等待超时
    //因为出现这种情况一般是server端排队严重，暂时挂起客户端可以避免情况恶化
    // deadline机制，如果QMQ认为这个消费请求来不及处理，那么就直接返回，避免雪崩
    if (entry.expired()) {
        QMon.pullExpiredCountInc(entry.subject, entry.group);
        return true;
    }

    if (entry.isInvalid()) {
        QMon.pullInvalidCountInc(entry.subject, entry.group);
        return true;
    }

    // 存储层find消息
    final PullMessageResult pullMessageResult = store.findMessages(entry.pullRequest);

    if (pullMessageResult == PullMessageResult.FILTER_EMPTY ||
        pullMessageResult.getMessageNum() > 0
        || entry.isPullOnce()
        || entry.isTimeout()) {
        entry.processMessageResult(pullMessageResult);
        return true;
    }

    // 没有拉取到消息，那么挂起该actor
    self.suspend();
    // timer task, 在超时而唤醒actor
    if (entry.setTimerOnDemand()) {
        QMon.suspendRequestCountInc(entry.subject, entry.group);
        // 订阅消息，一有消息来就唤醒该actor
        subscribe(entry.subject, entry.group);
        return false;
    }
}
```





```
        return true;
    }

    // 订阅
    private void subscribe(String subject, String group) {
        ConcurrentMap<String, Object> map = subscribers.get(subject);
        if (map == null) {
            map = new ConcurrentHashMap<>();
            map = ObjectUtils.defaultIfNull(subscribers.putIfAbsent(subject, map), map);
        }
        map.putIfAbsent(group, HOLDER);
    }

    // 有消息来就唤醒订阅的subscriber
    void remindNewMessages(final String subject) {
        final ConcurrentMap<String, Object> map = this.subscribers.get(subject);
        if (map == null) return;

        for (String group : map.keySet()) {
            map.remove(group);
            this.actorSystem.resume(ConsumerGroupUtils.buildConsumerGroupKey(subject, group));
            QMon.resumeActorCountInc(subject, group);
        }
    }

    // ActorSystem内定义的处理接口
    public interface ActorSystem.Processor<T> {
        boolean process(T message, Actor<T> self);
    }
```

能看出在这里起作用的是这个actorSystem。PullMessageWorker继承了ActorSystem.Processor，所以真正处理拉取请求的是这个接口里的process方法。请求到达pullMessageWorker，worker将该次请求交给actorSystem调度，调度到这次请求时，worker还有个根据拉取结果做反应的策略，即如果暂时没有消息，那么suspend，以一个timer task定时resume；如果在timer task执行之前有消息进来，那么也会即时resume。

## ActorSystem

接下来就看看ActorSystem里边是如何做的 **公平调度**。



[首页](#) ▼[搜索更新啦](#)

```

private final ConcurrentMap<String, Actor> actors;
// 执行actor的executor
private final ThreadPoolExecutor executor;

private final AtomicInteger actorsCount;
private final String name;

public ActorSystem(String name) {
    this(name, Runtime.getRuntime().availableProcessors() * 4, true);
}

public ActorSystem(String name, int threads, boolean fair) {
    this.name = name;
    this.actorsCount = new AtomicInteger();
    // 这里根据fair参数初始化一个优先级队列作为executor的参数, 处理关于前言里说的"老消息"的情况
    BlockingQueue<Runnable> queue = fair ? new PriorityBlockingQueue<>() : new LinkedBlockingQueue<>();
    this.executor = new ThreadPoolExecutor(threads, threads, 60, TimeUnit.MINUTES, queue, new NioExecutorFactory());
    this.actors = Maps.newConcurrentMap();
    QMon.dispatchersGauge(name, actorsCount::doubleValue);
    QMon.actorSystemQueueGauge(name, () -> (double) executor.getQueue().size());
}
}

```

可以看到, 用一个线程池处理actor的调度执行, 这个线程池里的队列是一个优先级队列。优先级队列存储的元素是Actor。关于Actor我们稍后来看, 先来看一下ActorSystem的处理调度流程。

java

```

// PullMessageWorker调用的就是这个方法
public <E> void dispatch(String actorPath, E msg, Processor<E> processor) {
    // 取得actor
    Actor<E> actor = createOrGet(actorPath, processor);
    // 在后文Actor定义里能看到, actor内部维护一个queue, 这里actor仅仅是offer(msg)
    actor.dispatch(msg);
    // 执行调度
    schedule(actor, true);
}

// 无消息时, 则会挂起
public void suspend(String actorPath) {
    Actor actor = actors.get(actorPath);
    if (actor == null) return;

    actor.suspend();
}

```



[首页](#) ▼[搜索更新啦](#)

```

    if (actor == null) return;

    actor.resume();
    // 立即调度, 可以留意一下那个false
    // 当actor是"可调度状态"时, 这个actor是否能调度是取决于actor的queue是否有消息
    schedule(actor, false);
}

private <E> Actor<E> createOrGet(String actorPath, Processor<E> processor) {
    Actor<E> actor = actors.get(actorPath);
    if (actor != null) return actor;

    Actor<E> add = new Actor<>(this.name, actorPath, this, processor, DEFAULT_QUEUE_SIZE);
    Actor<E> old = actors.putIfAbsent(actorPath, add);
    if (old == null) {
        LOG.info("create actorSystem: {}", actorPath);
        actorsCount.incrementAndGet();
        return add;
    }
    return old;
}

// 将actor入队的地方
private <E> boolean schedule(Actor<E> actor, boolean hasMessageHint) {
    // 如果actor不能调度, 则ret false
    if (!actor.canBeScheduled(hasMessageHint)) return false;
    // 设置actor为"可调度状态"
    if (actor.setAsScheduled()) {
        // 提交时间, 和actor执行总耗时共同决定在队列里的优先级
        actor.submitTs = System.currentTimeMillis();
        // 入队, 入的是线程池里的优先级队列
        this.executor.execute(actor);
        return true;
    }
    // actor.setAsScheduled()里, 这里是actor已经是可调度状态, 那么没必要再次入队
    return false;
}

```

actorSystem维护一个线程池, 线程池队列具有优先级, 队列存储元素是actor。actor的粒度是subject+group。Actor是一个Runnable, 且因为是优先级队列的存储元素所以需继承Comparable接口(队列并没有传Comparator参数), 并且actor有四种状态, 初始状态、可调度状态、挂起状态、调度状态(这个状态其实不存在, 但是暂且这么叫以帮助理解)。

接下来看看Actor这个类:





```

private static final int Open = 0;
// 可调度状态
private static final int Scheduled = 2;
// 掩码, 二进制表示:11 与Open和Scheduled作&运算
// shouldScheduleMask&currentStatus != Open 则为不可置为调度状态 (当currentStatus为挂起状态或调
private static final int shouldScheduleMask = 3;
private static final int shouldNotProcessMask = ~2;
// 挂起状态
private static final int suspendUnit = 4;
//每个actor至少执行的时间片
private static final int QUOTA = 5;
// status属性内存偏移量, 用Unsafe操作
private static long statusOffset;

static {
    try {
        statusOffset = Unsafe.instance.objectFieldOffset(Actor.class.getDeclaredField("stat
    } catch (Throwable t) {
        throw new ExceptionInInitializerError(t);
    }
}

final String systemName;
final ActorSystem actorSystem;
// actor内部维护的queue, 后文简单分析下
final BoundedNodeQueue<E> queue;
// ActorSystem内部定义接口, PullMessageWorker实现的就是这个接口, 用于真正业务逻辑处理的地方
final Processor<E> processor;
private final String name;
// 一个actor执行总耗时
private long total;
// actor执行提交时间, 即actor入队时间
private volatile long submitTs;
//通过Unsafe操作
private volatile int status;

Actor(String systemName, String name, ActorSystem actorSystem, Processor<E> processor, fina
    this.systemName = systemName;
    this.name = name;
    this.actorSystem = actorSystem;
    this.processor = processor;
    this.queue = new BoundedNodeQueue<>(queueSize);

    QMon.actorQueueGauge(systemName, name, () -> (double) queue.count());
}

```





```
}

// actor执行的地方
@Override
public void run() {
    long start = System.currentTimeMillis();
    String old = Thread.currentThread().getName();
    try {
        Thread.currentThread().setName(systemName + "-" + name);
        if (shouldProcessMessage()) {
            processMessages();
        }
    } finally {
        long duration = System.currentTimeMillis() - start;
        // 每次actor执行的耗时累加到total
        total += duration;
        QMon.actorProcessTime(name, duration);

        Thread.currentThread().setName(old);
        // 设置为"空闲状态", 即初始状态 (currentStatus & ~Scheduled)
        setAsIdle();
        // 进行下一次调度
        this.actorSystem.schedule(this, false);
    }
}

void processMessages() {
    long deadline = System.currentTimeMillis() + QUOTA;
    while (true) {
        E message = queue.peek();
        if (message == null) return;
        // 处理业务逻辑
        boolean process = processor.process(message, this);
        // 失败, 该message不会出队, 等待下一次调度
        // 如pullMessageWorker中没有消息时将actor挂起
        if (!process) return;

        // 出队
        queue.pollNode();
        // 每个actor只有QUOTA个时间片的执行时间
        if (System.currentTimeMillis() >= deadline) return;
    }
}

final boolean shouldProcessMessage() {
    // 能够真正执行业务逻辑的判断
```





```
}

// 能否调度
private boolean canBeSchedule(boolean hasMessageHint) {
    int s = currentStatus();
    if (s == Open || s == Scheduled) return hasMessageHint || !queue.isEmpty();
    return false;
}

public final boolean resume() {
    while (true) {
        int s = currentStatus();
        int next = s < suspendUnit ? s : s - suspendUnit;
        if (updateStatus(s, next)) return next < suspendUnit;
    }
}

public final void suspend() {
    while (true) {
        int s = currentStatus();
        if (updateStatus(s, s + suspendUnit)) return;
    }
}

final boolean setAsScheduled() {
    while (true) {
        int s = currentStatus();
        // currentStatus为非Open状态, 则ret false
        if ((s & shouldScheduleMask) != Open) return false;
        // 更新actor状态为调度状态
        if (updateStatus(s, s | Scheduled)) return true;
    }
}

final void setAsIdle() {
    while (true) {
        int s = currentStatus();
        // 更新actor状态位不可调度状态, (这里可以理解为更新为初始状态Open)
        if (updateStatus(s, s & ~Scheduled)) return;
    }
}

final int currentStatus() {
    // 根据status在内存中的偏移量取得status
    return Unsafe.instance.getIntVolatile(this, statusOffset);
}
```





```
        return Unsafe.instance.compareAndSwapInt(this, statusOffset, oldStatus, newStatus);
    }

    // 决定actor在优先级队列里的优先级的地方
    // 先看总耗时，以达到动态限速，保证执行"慢"的请求（已经堆积的消息拉取请求）在后执行
    // 其次看提交时间，先提交的actor先执行
    @Override
    public int compareTo(Actor o) {
        int result = Long.compare(total, o.total);
        return result == 0 ? Long.compare(submitTs, o.submitTs) : result;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Actor<?> actor = (Actor<?>) o;
        return Objects.equals(systemName, actor.systemName) &&
            Objects.equals(name, actor.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(systemName, name);
    }
}
```

Actor实现了Comparable，在优先级队列里优先级是Actor里的total和submitTs共同决定的。total是actor执行总耗时，submitTs是调度时间。那么对于处理较慢的actor自然就会在队列里相对"尾部"位置，这时就做到了根据actor的执行耗时的一个动态限速。Actor利用Unsafe机制来控制各个状态的轮转原子性更新的，且每个actor执行时间可以简单理解为5个时间片。

其实工作进行到这里就可以结束了，但是抱着研究的态度，不妨接着往下看。

Actor内部维护一个Queue，这个Queue是自定义的，是一个Lock-free bounded non-blocking multiple-producer single-consumer queue。JDK里的QUEUE多数都是用锁控制，不用锁，猜测也应该用Unsafe 原子操作实现。那么来看看吧：

```
private static class BoundedNodeQueue<T> {
```

```
    // 头结点、尾节点在内存中的偏移量
```

```
    private final static long enqOffset, deqOffset;
```

java





```
        enqOffset = Unsafe.instance.objectFieldOffset(BoundedNodeQueue.class.getDeclaredField("enqOffset"));
        deqOffset = Unsafe.instance.objectFieldOffset(BoundedNodeQueue.class.getDeclaredField("deqOffset"));
    } catch (Throwable t) {
        throw new ExceptionInInitializerError(t);
    }
}

private final int capacity;
// 尾节点，通过enqOffset操作
private volatile Node<T> _enqDoNotCallMeDirectly;
// 头节点，通过deqOffset操作
private volatile Node<T> _deqDoNotCallMeDirectly;

protected BoundedNodeQueue(final int capacity) {
    if (capacity < 0) throw new IllegalArgumentException("AbstractBoundedNodeQueue.capacity must be non-negative");
    this.capacity = capacity;
    final Node<T> n = new Node<T>();
    setDeq(n);
    setEnq(n);
}

// 获取尾节点
private Node<T> getEnq() {
    // getObjectVolatile这种方式保证拿到的都是最新数据
    return (Node<T>) Unsafe.instance.getObjectVolatile(this, enqOffset);
}

// 设置尾节点，仅在初始化时用
private void setEnq(Node<T> n) {
    Unsafe.instance.putObjectVolatile(this, enqOffset, n);
}

private boolean casEnq(Node<T> old, Node<T> nju) {
    // cas, 循环设置，直到成功
    return Unsafe.instance.compareAndSwapObject(this, enqOffset, old, nju);
}

// 获取头节点
private Node<T> getDeq() {
    return (Node<T>) Unsafe.instance.getObjectVolatile(this, deqOffset);
}

// 仅在初始化时用
private void setDeq(Node<T> n) {
    Unsafe.instance.putObjectVolatile(this, deqOffset, n);
}
```



```

        return Unsafe.instance.compareAndSwapObject(this, deqOffset, old, nju);
    }

    // 与其叫count, 不如唤作index, 但是是否应该考虑溢出的情况?
    public final int count() {
        final Node<T> lastNode = getEnq();
        final int lastNodeCount = lastNode.count;
        return lastNodeCount - getDeq().count;
    }

    /**
     * @return the maximum capacity of this queue
     */
    public final int capacity() {
        return capacity;
    }

    public final boolean add(final T value) {
        for (Node<T> n = null; ; ) {
            final Node<T> lastNode = getEnq();
            final int lastNodeCount = lastNode.count;
            if (lastNodeCount - getDeq().count < capacity) {
                // Trade a branch for avoiding to create a new node if full,
                // and to avoid creating multiple nodes on write conflict á la Be Kind to Your C
                if (n == null) {
                    n = new Node<T>();
                    n.value = value;
                }

                n.count = lastNodeCount + 1; // Piggyback on the HB-edge between getEnq() and c

                // Try to putPullLogs the node to the end, if we fail we continue loopin'
                // 相当于
                // enq -> next = new Node(value); enq = neq -> next;
                if (casEnq(lastNode, n)) {
                    // 注意一下这个Node.setNext方法
                    lastNode.setNext(n);
                    return true;
                }
            } else return false; // Over capacity-couldn't add the node
        }
    }

    public final boolean isEmpty() {
        // enq == deq 即为empty
        return getEnq() == getDeq();
    }

```





```

    * Removes the first element of this queue if any
    *
    * @return the value of the first element of the queue, null if empty
    */
    public final T poll() {
        final Node<T> n = pollNode();
        return (n != null) ? n.value : null;
    }

    public final T peek() {
        Node<T> n = peekNode();
        return (n != null) ? n.value : null;
    }

    protected final Node<T> peekNode() {
        for ( ; ; ) {
            final Node<T> deq = getDeq();
            final Node<T> next = deq.next();
            if (next != null || getEnq() == deq)
                return next;
        }
    }

    /**
     * Removes the first element of this queue if any
     *
     * @return the `Node` of the first element of the queue, null if empty
     */
    public final Node<T> pollNode() {
        for ( ; ; ) {
            final Node<T> deq = getDeq();
            final Node<T> next = deq.next();
            if (next != null) {
                if (casDeq(deq, next)) {
                    deq.value = next.value;
                    deq.setNext(null);
                    next.value = null;
                    return deq;
                } // else we retry (concurrent consumers)
                // 比较套路的cas操作，就不多说了
            } else if (getEnq() == deq) return null; // If we got a null and head meets tail, we
        }
    }

    public static class Node<T> {

```





```
        try {
            nextOffset = Unsafe.instance.objectFieldOffset(Node.class.getDeclaredField("_ne
        } catch (Throwable t) {
            throw new ExceptionInInitializerError(t);
        }
    }

    protected T value;
    protected int count;
    // 也是利用偏移量操作
    private volatile Node<T> _nextDoNotCallMeDirectly;

    public final Node<T> next() {
        return (Node<T>) Unsafe.instance.getObjectVolatile(this, nextOffset);
    }

    protected final void setNext(final Node<T> newNext) {
        // 这里有点讲究，下面分析下
        Unsafe.instance.putOrderedObject(this, nextOffset, newNext);
    }
}
}
```

如上代码，是通过属性在内存的偏移量，结合cas原子操作来进行更新赋值等操作，以此来实现lock-free，这是比较常规的套路。值得一说的是Node里的setNext方法，这个方法的调用是在cas节点后，对"上一位置"的next节点进行赋值。而这个方法使用的是Unsafe.instance.putOrderedObject，要说这个putOrderedObject，就不得不说MESI，缓存一致性协议。如volatile，当进行写操作时，它是依靠storeload barrier来实现其他线程对此的可见性。而putOrderedObject也是依靠内存屏障，只不过是storestore barrier。storestore是比storeload快速的一种内存屏障。在硬件层面，内存屏障分两种：Load-Barrier和Store-Barrier。Load-Barrier是让高速缓存中的数据失效，强制重新从主内存加载数据；Store-Barrier是让写入高速缓存的数据更新写入主内存，对其他线程可见。而java层面的四种内存屏障无非是硬件层面的两种内存屏障的组合而已。那么可见，storestore barrier自然比storeload barrier快速。那么有一个问题，我们可不可以在这里也用cas操作呢？答案是可以，但没必要。你可以想想这里为什么没必要。

关注下面的标签，发现更多相似文章

消息队列



[首页](#) ▾[搜索更新啦](#)

## 安装掘金浏览器插件

打开新标签页发现好内容，掘金、GitHub、Dribbble、ProductHunt 等站点内容轻松获取。快来安装掘金浏览器插件获取高质量内容吧！

## 评论

您需要[绑定手机号](#)后才可在掘金社区内发布内容。

架构随笔 [Lv1](#) Qunar.inc

加个好友吧，我也是去哪儿的，最近也想研究QMQ源码

1月前



回复

魔法咪路 [Lv1](#) (作者) 开发工程师 @ Qu...

回复 架构随笔 [Lv1](#): 好

1月前

架构随笔 [Lv1](#) Qunar.inc

回复 魔法咪路 [Lv1](#) (作者): 加个好友吧，微信m631521383

1月前

## 相关推荐

专栏 · Java3y · 2月前 · Java / 消息队列

### 什么是消息队列？

👍 353

💬 26

专栏 · java闸瓦 · 1月前 · Java / 分布式

### 分布式服务（RPC）+分布式消息队列（MQ）面试题精选

👍 149

💬 3

专栏 · 小姐姐味道 · 1月前 · Kafka / 消息队列

### 开源一个kafka增强：okmq-1.0.0

👍 15

💬 2

专栏 · 猿天地 · 1月前 · 消息队列 / 架构


### 拍拍贷消息系统原理与应用






专栏 · 一条路上的咸鱼 · 1月前 · 消息队列

关于MQ的几件小事（一）消息队列的用途、优缺点、技术选型

 6



专栏 · 猿小源 · 19天前 · 消息队列


RabbitMQ + Quartz +Swagger 使用记录

 1



专栏 · 神一样的编程 · 9月前 · 后端 / 架构

消息队列mq总结

 192

 5

专栏 · thekingisalwayslucky · 1月前 · 消息队列


消息队列（四）阿里RocketMQ


 1



专栏 · QLQ · 1月前 · 消息队列

Java面试之消息队列

 3



专栏 · thekingisalwayslucky · 1月前 · 消息队列

消息队列（三）常见消息队列介绍

 1



