zhuanlan.zhihu.com

分布式队列编程优化篇 - 知乎专栏

49-62 分钟

“分布式队列编程”是一个系列文，之前我们已经发布了《分布式队列编程模型、实战》，主要剖析了分布式队列编程模型的需求来源、定义、结构以及其变化多样性；根据作者在新美大实际工作经验，给出了队列式编程在分布式环境下的一些具体应用。本文将重点阐述工程师运用分布式队列编程构架的时候，在生产者、分布式队列以及消费者这三个环节的注意点以及优化建议。

确定采用分布式队列编程模型之后，主体架构就算完成了，但工程师的工作还远远未结束。天下事必做于细，细节是一个不错的架构向一个优秀的系统进阶的关键因素。优化篇选取了作者以及其同事在运用分布式队列编程模型架构时所碰到的典型问题和解决方案。这里些问题出现的频率较高，如果你经验不够，很可能会“踩坑”。希望通过这些讲解，帮助读者降低分布式队列编程模型的使用门槛。本文将对分布式队列编程模型的三种角色：生产者（Producer），分布式队列（Queue），消费者（Consumer）分别进行优化讨论。

在分布式队列编程中，生产者往往并非真正的生产源头，只是整个数据流中的一个节点，这种生产者的操作是处理 - 转发（Process-Forward）模式。

这种模式给工程师们带来的第一个问题是吞吐量问题。这种模式下运行的生产者，一边接收上游的数据，一边将处理完的数据发送给下游。本质上，它是一个非常经典的数学问题，其抽象模型是一些没有盖子的水箱，每个水箱接收来自上一个水箱的水，进行处理之后，再

将水发送到下一个水箱。工程师需要预测水源的流量、每个环节水箱的处理能力、水龙头的排水速度，最终目的是避免水溢出水箱，或者尽可能地减小溢出事件的概率。实际上流式编程框架以及其开发者花了大量的精力去处理和优化这个问题。下文的缓存优化和批量写入优化都是针对该问题的解决方案。

第二个需要考虑的问题是持久化。由于各种原因，系统总是会宕机。如果信息比较敏感，例如计费信息、火车票订单信息等，工程师们需要考虑系统宕机所带来的损失，找到让损失最小化的解决方案。持久化优化重点解决这一类问题。

缓存优化

处于“处理 - 转发”模式下运行的生产者往往被设计成请求驱动型的服务，即每个请求都会触发一个处理线程，线程处理完后将结果写入分布式队列。如果由于某种原因队列服务不可用，或者性能恶化，随着新请求的到来，生产者的处理线程就会产生堆积。这可能会导致如下两个问题：

- 系统可用性降低。由于每个线程都需要一定的内存开销，线程过多会使系统内存耗尽，甚至可能产生雪崩效应导致最终完全不可用。
- 信息丢失。为了避免系统崩溃，工程师可能会给请求驱动型服务设置一个处理线程池，设置最大处理线程数量。这是一种典型的降级策略，目的是为了系统崩溃。但是，后续的请求会因为没有处理线程而被迫阻塞，最终可能产生信息丢失。例如：对于广告计费采集，如果采集系统因为线程耗尽而不接收客户端的计费行为，这些计费行为就会丢失。

缓解这类问题的思路来自于CAP理论，即通过降低一致性来提高可用性。生产者接收线程在收到请求之后第一时间不去处理，直接将请求缓存在内存中（牺牲一致性），而在后台启动多个处理线程从缓存中读取请求、进行处理并写入分布式队列。与线程所占用的内存开销

相比，大部分的请求所占内存几乎可以忽略。通过在接收请求和处理请求之间增加一层内存缓存，可以大大提高系统的处理吞吐量和可扩展性。这个方案本质上是一个内存生产者消费者模型。

批量写入优化

如果生产者的请求过大，写分布式队列可能成为性能瓶颈，有如下几个因素：

- 队列自身性能不高。
- 分布式队列编程模型往往被应用在跨机房的系统里面，跨机房的网络开销往往容易成为系统瓶颈。
- 消息确认机制往往会大大降低队列的吞吐量以及响应时间。

如果在处理请求和写队列之间添加一层缓存，消息写入程序批量将消息写入队列，可以大大提高系统的吞吐量。原因如下：

- 批量写队列可以大大减少生产者和分布式队列的交互次数和消息传输量。特别是对于高吞吐小载荷的消息实体，批量写可以显著降低网络传输量。
- 对于需要确认机制的消息，确认机制往往会大大降低队列的吞吐量以及响应时间，某些高敏感的消息需要多个消息中间件代理同时确认，这近一步恶化性能。在生产者的应用层将多条消息批量组合成一个消息体，消息中间件就只需要对批量消息进行一次确认，这可能会数量级的提高消息传输性能。

持久化优化

通过添加缓存，消费者服务的吞吐量和可用性都得到了提升。但缓存引入了一个新问题——内存数据丢失。对于敏感数据，工程师需要考虑如下两个潜在问题：

- 如果内存中存在未处理完的请求，而某些原因导致生产者服务宕机，

内存数据就会丢失而可能无法恢复。

- 如果分布式队列长时间不可用，随着请求数量的不断增加，最终系统内存可能会耗尽而崩溃，内存的消息也可能丢失。

所以缓存中的数据需要定期被持久化到磁盘等持久层设备中，典型的持久化触发策略主要有两种：

- 定期触发，即每隔一段时间进行一次持久化。
- 定量触发，即每当缓存中的请求数量达到一定阈值后进行持久化。

是否需要持久化优化，以及持久化策略应该由请求数据的敏感度、请求量、持久化性能等因素共同决定。

分布式队列不等同于各种开源的或者收费的消息中间件，甚至在一些场景下完全不需要使用消息中间件。但是，消息中间件产生的目的就是解决消息传递问题，这为分布式队列编程架构提供了很多的便利。在实际工作中，工程师们应该将成熟的消息中间件作为队列的首要备选方案。

本小节对消息中间件的功能、模型进行阐述，并给出一些消息中间件选型、部署的具体建议。

中间件的功能

明白一个系统的每个具体功能是设计和架构一个系统的基础。典型的消息中间件主要包含如下几个功能：

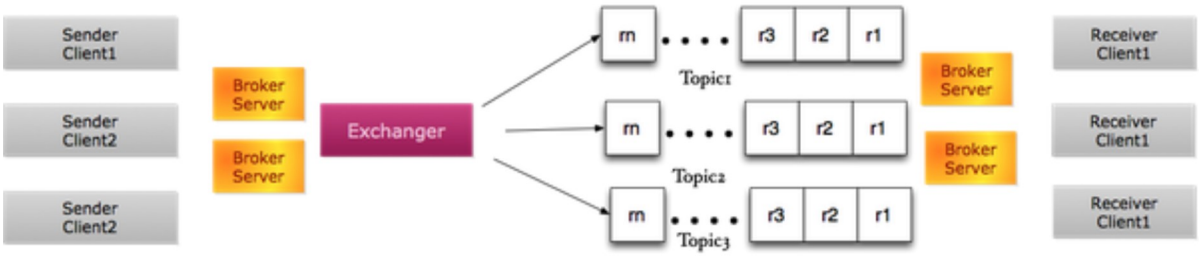
- 消息接收
- 消息分发
- 消息存储
- 消息读取

概念模型

抽象的消息中间件模型包含如下几个角色：

- 发送者和接收者客户端（Sender/Receiver Client），在具体实施过程中，它们一般以库的形式嵌入到应用程序代码中。
- 代理服务器（Broker Server），它们是与客户端代码直接交互的服务端代码。
- 消息交换机（Exchanger），接收到的消息一般需要通过消息交换机（Exchanger）分发到具体的消息队列中。
- 消息队列，一般是一块内存数据结构或持久化数据。

概念模型如下图：



为了提高分发性能，很多消息中间件把消息代理服务器的拓扑图发送到发送者和接收者客户端（Sender/Receiver Client），如此一来，发送源可以直接进行消息分发。

选型标准

要完整的描述消息中间件各个方面非常困难，大部分良好的消息中间件都有完善的文档，这些文档的长度远远超过本文的总长度。但如下几个标准是工程师们在进行消息中间件选型时经常需要考虑和权衡的。

性能

性能主要有两个方面需要考虑：吞吐量（Throughput）和响应时间（Latency）。

不同的消息队列中间件的吞吐量和响应时间相差甚远，在选型时可以

去网上查看一些性能对比报告。
对于同一种中间件，不同的配置方式也会影响性能。主要有如下几方面的配置：

- 是否需要确认机制，即写入队列后，或从队列读取后，是否需要进行确认。确认机制对响应时间的影响往往很大。
- 能否批处理，即消息能否批量读取或者写入。批量操作可以大大减少应用程序与消息中间件的交互次数和消息传递量，大大提高吞吐量。
- 能否进行分区（Partition）。将某一主题消息队列进行分区，同一主题消息可以有多台机器并行处理。这不仅仅能影响消息中间件的吞吐量，还决定着消息中间件是否具备良好的可伸缩性（Scalability）。
- 是否需要进行持久化。将消息进行持久化往往会同时影响吞吐量和响应时间。

可靠性

可靠性主要包含：可用性、持久化、确认机制等。
高可用性的消息中间件应该具备如下特征：

- 消息中间件代理服务器（Broker）具有主从备份。即当一台代理服务宕机之后，备用服务器能接管相关的服务。
- 消息中间件中缓存的消息是否有备份、并持久化。
根据CAP理论，高可用、高一致性以及网络分裂不可兼得。根据作者的观察，大部分的消息中间件在面临网络分裂的情况下下，都很难保证数据的一致性以及可用性。 很多消息中间件都会提供一些可配置策略，让使用者在可用性和一致性之间做权衡。

高可靠的消息中间件应该确保从发送者接收到的消息不会丢失。中间件代理服务器的宕机并不是小概率事件，所以保存在内存中的消息很容易发生丢失。大部分的消息中间件都依赖于消息的持久化去降低消息丢失损失，即将接收到的消息写入磁盘。即使提供持久化，仍有两

个问题需要考虑：

- 磁盘损坏问题。长时间来看，磁盘出问题的概率仍然存在。
- 性能问题。与操作内存相比，磁盘I/O的操作性能要慢几个数量级。频繁持久化不仅会增加响应时间，也会降低吞吐量。

解决这两个问题的一个解决方案就是：多机确认，定期持久化。即消息被缓存在多台机器的内存中，只有每台机器都确认收到消息，才跟发送者确认（很多消息中间件都会提供相应的配置选项，让用户设置最少需要多少台机器接收到消息）。由于多台独立机器同时出故障的概率遵循乘法法则，指数级降低，这会大大提高消息中间件的可靠性。

确认机制本质上是通讯的握手机制（Handshaking）。如果没有该机制，消息在传输过程中丢失将不会被发现。高敏感的消息要求选取具备确认机制的消息中间件。当然如果没有接收到消息中间件确认完成的指令，应用程序需要决定如何处理。典型的做法有两个：

- 多次重试。
- 暂存到本地磁盘或其它持久化媒介。

客户端接口所支持语言

采用现存消息中间件就意味着避免重复造轮子。如果某个消息中间件未能提供对应语言的客户端接口，则意味着极大的成本和兼容性问题。

投递策略（Delivery policies）

投递策略指的是一个消息会被发送几次。主要包含三种策略：最多一次（At most Once ）、最少一次（At least Once）、仅有一次（Exactly Once）。

在实际应用中，只考虑消息中间件的投递策略并不能保证业务的投递策略，因为接收者在确认收到消息和处理完消息并持久化之间存在一

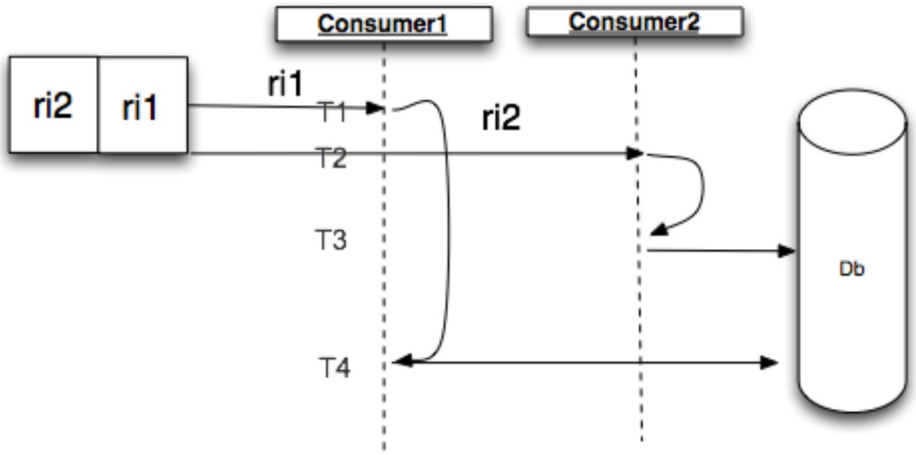
个时间窗口。例如，即使消息中间件保证仅有一次（Exactly Once），如果接收者先确认消息，在持久化之前宕机，则该消息并未被处理。从应用的角度，这就是最多一次（At most Once）。反之，接收者先处理消息并完成持久化，但在确认之前宕机，消息就要被再次发送，这就是最少一次（At least Once）。 如果消息投递策略非常重要，应用程序自身也需要仔细设计。

消费者是分布式队列编程中真正的数据处理方，数据处理方最常见的挑战包括：有序性、串行化（Serializability）、频次控制、完整性和一致性等。

挑战

有序性

在很多场景下，如何保证队列信息的有序处理是一个棘手的问题。如下图，假定分布式队列保证请求严格有序，请求ri2和ri1都是针对同一数据记录的不同状态，ri2的状态比ri1的状态新。T1、T2、T3和T4代表各个操作发生的时间，并且 T1 < T2 < T3 < T4（"<"代表早于）。
采用多消费者架构，这两条记录被两个消费者（Consumer1和Consumer2）处理后更新到数据库里面。Consumer1虽然先读取ri1但是却后写入数据库，这就导致，新的状态被老的状态覆盖，所以多消费者不保证数据的有序性。



串行化

很多场景下，串行化是数据处理的一个基本需求，这是保证数据完整性、可恢复性、事务原子性等的基础。为了在并行计算系统里实现串行化，一系列的相关理论和实践算法被提出。对于分布式队列编程架构，要在在多台消费者实现串行化非常复杂，无异于重复造轮子。

频次控制

有时候，消费者的消费频次需要被控制，可能的原因包括：

- 费用问题。如果每次消费所引起的操作都需要收费，而同一个请求消息在队列中保存多份，不进行频次控制，就会导致无谓的浪费。
- 性能问题。每次消费可能会引起对其他服务的调用，被调用服务希望对调用量有所控制，对同一个请求消息的多次访问就需要有所控制。

完整性和一致性

完整性和一致性是所有多线程和多进程的代码都面临的问题。在多线程或者多进程的系统中考虑完整性和一致性往往会大大地增加代码的复杂度和系统出错的概率。

单例服务优化

几乎所有串行化理论真正解决的问题只有一个：性能。 所以，在性能允许的前提下，对于消费者角色，建议采用单实例部署。通过单实例部署，有序性、串行化、完整性和一致性问题自动获得了解决。另外，单实例部署的消费者拥有全部所需信息，它可以在频次控制上采取很多优化策略。

天下没有免费的午餐。同样，单实例部署并非没有代价，它意味着系统可用性的降低，很多时候，这是无法接受的。解决可用性问题的最直接的思路就是冗余（Redundancy）。最常用的冗余方案是Master-slave架构，不过大部分的Master-slave架构都是Active/active模式，即主从服务器都提供服务。例如，数据库的Master-slave架构就是主从服务器都提供读服务，只有主服务器提供写服务。大部分基于负载均衡设计的Master-slave集群中，主服务器和从服务器同时提供相同的服务。这显然不满足单例服务优化需求。有序性和串行化需要Active/passive架构，即在某一时刻只有主实例提供服务，其他的从服务等待主实例失效。这是典型的领导人选举架构，即只有获得领导权的实例才能充当实际消费者，其他实例都在等待下一次选举。采用领导人选举的Active/passive架构可以大大缓解纯粹的单实例部署所带来的可用性问题。

令人遗憾的是，除非工程师们自己在消费者实例里面实现Paxos等算法，并在每次消息处理之前都执行领导人选举。否则，理论上讲，没有方法可以保障在同一个时刻只有一个领导者。而对每个消息都执行一次领导人选举，显然性能不可行。实际工作中，最容易出现的问题时机发生在领导人交接过程中，即前任领导人实例变成辅助实例，新部署实例开始承担领导人角色。为了平稳过渡，这两者之间需要有一定的通讯机制，但是，无论是网络分区（Network partition）还是原领导人服务崩溃都会使这种通讯机制变的不可能。

对于完整性和一致性要求很高的系统，我们需要在选举制度和交接制度这两块进行优化。

领导人选举架构

典型的领导人选举算法有Paxos、ZAB（ ZooKeeper Atomic Broadcast protocol）。为了避免重复造轮子，建议采用ZooKeeper的分布式锁来实现领导人选举。典型的ZooKeeper实现算法如下（摘自参考资料[4]）：

Let ELECTION be a path of choice of the application. To volunteer to be a leader:

1.Create znode z with path "ELECTION/guid-n_" with both SEQUENCE and EPHEMERAL flags;
2.Let C be the children of "ELECTION", and i be the sequence number of z;
3.Watch for changes on "ELECTION/guid-n_j", where j is the largest sequence number such that j < i and n_j is a znode in C;

Upon receiving a notification of znode deletion:

1.Let C be the new set of children of ELECTION;
2.If z is the smallest node in C, then execute leader procedure;
3.Otherwise, watch for changes on "ELECTION/guid-n_j", where j is the largest sequence number such that j < i and n_j is a znode in C;

领导人交接架构

领导人选举的整个过程发生在ZooKeeper集群中，各个消费者实例在这场选举中只充当被告知者角色（Learner）。领导人选举算法，只能保证最终只有一个Leader被选举出来，并不保障被告知者对Leader的理解是完全一致的。本质上，上文的架构里，选举的结果是作为令牌（Token）传递给消费者实例，消费者将自身的ID与令牌进行对比，如果相等，则开始执行消费操作。所以当发生领导人换届的情况，不同的Learner获知新Leader的时间并不同。例如，前任

Leader如果因为网络问题与ZooKeeper集群断开，前任Leader只能在超时后才能判断自己是否不再承担Leader角色了，而新的Leader可能在这之前已经产生。另一方面，即使前任Leader和新Leader同时接收到新Leader选举结果，某些业务的完整性要求迫使前任Leader仍然完成当前未完成的工作。以上的讲解非常抽象，生活中却给了一些更加具体的例子。众所周知，美国总统候选人在选举结束后并不直接担任美国总统，从选举到最终承担总统角色需要一个过渡期。对于新当选Leader的候选人而言，过渡期间称之为加冕阶段（Inauguration）。对于即将卸任的Leader，过渡期称为交接阶段（HandOver）。所以一个基于领导人选举的消费者从加冕到卸任经历三个阶段：Inauguration、Execution、HandOver。在加冕阶段，新领导需要进行一些初始化操作。Execution阶段是真正的队列消息处理阶段。在交接阶段，前任领导需要进行一些清理操作。

类似的，为了解决领导人交接问题，所有的消费者从代码实现的角度都需要实现类似ILeaderCareer接口。这个接口包含三个方发inaugurate()，handOver()和execute（ ）。某个部署实例（Learner）在得知自己承担领导人角色后，需要调用inaugurate()方法，进行加冕。主要的消费逻辑通过不停的执行execute（ ）实现，当确认自己不再承担领导人之后，执行handOver()进行交接。

```
public interface ILeaderCareer {
    public void inaugurate();
    public void handOver();
    public boolean execute();
}
```

如果承担领导人角色的消费者，在执行execute（ ）阶段得知自己将要下台，根据消息处理的原子性，该领导人可以决定是否提前终止操作。如果整个消息处理是一个原子性事务，直接终止该操作可以快速实现领导人换届。否则，前任领导必须完成当前消息处理后，才进入交接阶段。这意味着新的领导人，在inaugurate()阶段需要进行一定

时间的等待。

排重优化

频次控制是一个经典问题。对于分布式队列编程架构，相同请求重复出现在队列的情况并不少见。如果相同请求在队列中重复太多，排重优化就显得很必要。分布式缓存更新是一个典型例子，所有请求都被发送到队列中用于缓存更新。如果请求符合典型的高斯分布，在一段时间内会出现大量重复的请求，而同时多线程更新同一请求缓存显然没有太大的意义。
排重优化是一个算法，其本质是基于状态机的编程，整个讲解通过模型、构思和实施三个步骤完成。

模型

进行排重优化的前提是大量重复的请求。在模型这一小节，我们首先阐述重复度模型、以及不同重复度所导致的消费模型，最后基于这两个模型去讲解排重状态机。

重复度模型

首先我们给出最小重复长度的概念。同一请求最小重复长度：同一请求在队列中的重复出现的最小间距。例如，请求ri第一次出现在位置3，第二次出现在10，最小重复长度等于7。
是否需要进行排重优化取决于队列中请求的重复度。由于不同请求之间并不存在重复的问题，不失一般性，这里的模型只考了单个请求的重复度，重复度分为三个类：无重复、稀疏重复、高重复。
无重复：在整个请求过程，没有任何一个请求出现一次以上。
稀疏重复：主要的请求最小重复长度大于消费队列长度。
高重复：大量请求最小重复长度小于消费队列长度。
对于不同的重复度，会有不同的消费模型。

无重复消费模型

在整个队列处理过程中，所有的请求都不相同，如下图：

稀疏重复消费模型

当同一请求最小重复长度大于消费者队列长度，如下图。假定有3个消费者，Consumer1将会处理r1，Consumer2将会处理r2，Consumer3将会处理r3，如果每个请求处理的时间严格相等，Consumer1在处理完r1之后，接着处理r4，Consumer2将会处理r2之后会处理r1。虽然r1被再次处理，但是任何时刻，只有这一个消费者在处理r1，不会出现多个消费者同时处理同一请求的场景。

高重复消费模型

如下图，仍然假定有3个消费者，队列中前面4个请求都是r1，它会同时被3个消费者线程处理：

显然，对于无重复和稀疏重复的分布式队列，排重优化并不会带来额外的好处。排重优化所针对的对象是高重复消费模型，特别是对于并行处理消费者比较多的情况，重复处理同一请求，资源消耗极大。

排重状态机

排重优化的主要对象是高重复的队列，多个消费者线程或进程同时处理同一个幂等请求只会浪费计算资源并延迟其他待请求处理。所以，排重状态机的一个目标是处理唯一性，即：同一时刻，同一个请求只有一个消费者处理。如果消费者获取一条请求消息，但发现其他消费者正在处理该消息，则当前消费者应该处于等待状态。如果对同一请求，有一个消费者在处理，一个消费者在等待，而同一请求再次被消费者读取，再次等待则没有意义。所以，状态机的第二个目标是等待唯一性，即：同一时刻，同一个请求最多只有一个消费者处于等待状

态。总上述，状态机的目标是：处理唯一性和等待唯一性。我们把正在处理的请求称为头部请求，正在等待的请求称为尾部请求。
由于状态机的处理单元是请求，所以需要针对每一个请求建立一个排重状态机。基于以上要求，我们设计的排重状态机包含4个状态Init，Process，Block，Decline。各个状态之间转化过程如下图：

1. 状态机创建时处于Init状态。
2. 对Init状态进行Enqueue操作，即接收一个请求，开始处理（称为头部请求），状态机进入Process状态。
3. 状态机处于Process状态，表明当前有消费者正在处理头部请求。此时，如果进行Dequeue操作，即头部请求处理完成，返回Init状态。如果进行Enqueue操作，即另一个消费者准备处理同一个请求，状态机进入Block状态（该请求称为尾部请求）。
4. 状态机处于Block状态，表明头部请求正在处理，尾部请求处于阻塞状态。此时，进行Dequeue操作，即头部请求处理完成，返回Process状态，并且尾部请求变成头部请求，原尾部请求消费者结束阻塞状态，开始处理。进行Enqueue操作，表明一个新的消费者准备处理同一个请求，状态机进入Decline状态。
5. 状态机进入Decline状态，根据等待唯一性目标，处理最新请求的消费者将被抛弃该消息，状态机自动转换回Block状态。

构思

状态机描述的是针对单个请求操作所引起状态变化，排重优化需要解决队列中所有请求的排重问题，需要对所有请求的状态机进行管理。这里只考虑单虚拟机内部对所有请求状态机的管理，对于跨虚拟机的管理可以采用类似的方法。对于多状态机管理主要包含三个方面：一致性问题、完整性问题和请求缓存驱逐问题。

一致性问题

一致性在这里要求同一请求的不同消费者只会操作一个状态机。由于每个请求都产生一个状态机，系统将会包含大量的状态机。为了兼顾性能和一致性，我们采用ConcurrentHashMap保存所有的状态机。用ConcurrentHashMap而不是对整个状态机队列进行加锁，可以提高并行处理能力，使得系统可以同时操作不同状态机。为了避免处理同一请求的多消费者线程同时对ConcurrentHashMap进行插入所导致状态机不一致问题，我们利用了ConcurrentHashMap的putIfAbsent（）方法。代码方案如下，key2Status用于存储所有的状态机。消费者在处理请求之前，从状态机队列中读取排重状态机TrafficAutomate。如果没有找到，则创建一个新的状态机，并通过putIfAbsent（）方法插入到状态机队列中。

```
private ConcurrentHashMap<T, TrafficAutomate> key2Status =
new ConcurrentHashMap();
TrafficAutomate trafficAutomate = key2Status.get(key);
if(trafficAutomate == null)
{
    trafficAutomate = new TrafficAutomate();
    TrafficAutomate oldAutomate =
key2Status.putIfAbsent(key, trafficAutomate);
    if(oldAutomate != null)
    {
        trafficAutomate = oldAutomate;
    }
}
```

完整性问题

完整性要求保障状态机Init，Process，Block，Decline四种状态正确、状态之间的转换也正确。由于状态机的操作非常轻量级，兼顾完整性和降低代码复杂度，我们对状态机的所有方法进行加锁。

请求缓存驱逐问题（Cache Eviction）

如果不同请求的数量太多，内存永久保存所有请求的状态机的内存开销太大。所以，某些状态机需要在恰当的时候被驱逐出内存。这里有两个思路：

- 当状态机返回Init状态时，清除出队列。
- 启动一个后台线程，定时扫描状态机队列，采用LRU等标准缓存清除机制。

标识问题

每个请求对应于一个状态机，不同的状态机采用不同的请求进行识别。
对于同一状态机的不同消费者，在单虚拟机方案中，我们采用线程id进行标识。

实施

排重优化的主要功能都是通过排重状态机（TrafficAutomate）和状态机队列（QueueCoordinator）来实施的。排重状态机描述的是针对单个请求的排重问题，状态机队列解决所有请求状态机的排重问题。

状态机实施（TrafficAutomate）

根据状态机模型，其主要操作为enQueue和deQueue，其状态由头部请求和尾部请求的状态共同决定，所以需要定义两个变量为head和tail，用于表示头部请求和尾部请求。为了确保多线程操作下状态机的完整性（Integraty），所有的操作都将加上锁。

enQueue操作

当一个消费者执行enQueue操作时：如果此时尾部请求不为空，根据等待唯一性要求，返回DECLINE，当前消费者应该抛弃该请求；如果头部请求为空，返回ACCPET，当前消费者应该立刻处理该消息；否则，返回BLOCK，该消费者应该等待，并不停的查看状态机的状态，一直到头部请求处理完成。enQueue代码如下：

```
synchronized ActionEnum enQueue(long id)
{
    if(tail != INIT_QUEUE_ID)
    {
        return DECLINE;
    }

    if(head == INIT_QUEUE_ID)
    {
        head = id;
        return ACCEPT;
    }
    else
    {
        tail = id;
        return BLOCK;
    }
}
```

deQueue操作

对于deQueue操作，首先将尾部请求赋值给头部请求，并将尾部请求置为无效。deQueue代码如下：

```
synchronized boolean deQueue(long id)
{
        head = tail;
```

```
        tail = INIT_QUEUE_ID;
        return true;
}
```

状态机队列实施(QueueCoordinator)

接口定义
状态机队列集中管理所有请求的排重状态机，所以其操作和单个状态机一样，即enQueue和deQueuqe接口。这两个接口的实现需要识别特定请求的状态机，所以它们的入参应该是请求。为了兼容不同类型的请求消息，我们采用了Java泛型编程。接口定义如下：

```
public interface QueueCoordinator<T> {

    public boolean enQueue(T key);

    public void deQueue(T key);

}
```

enQueue操作
enQueue操作过程如下：
首先，根据传入的请求key值，获取状态机， 如果不存在则创建一个新的状态机，并保存在ConcurrentHashMap中。
接下来，获取线程id作为该消费者的唯一标识，并对对应状态机进行enQueue操作。
如果状态机返回值为ACCEPT或者DECLINE，返回业务层处理代码，ACCEPT意味着业务层需要处理该消息，DECLINE表示业务层可以抛弃当前消息。如果状态机返回值为Block，则该线程保持等待状态。
异常处理。在某些情况下，头部请求线程可能由于异常，未能对状态

机进行deQueue操作（作为组件提供方，不能假定所有的规范被使用方实施）。为了避免处于阻塞状态的消费者无期限地等待，建议对状态机设置安全超时时限。超过了一定时间后，状态机强制清空头部请求，返回到业务层，业务层开始处理该请求。
代码如下：

```
public boolean enQueue(T key) {
    _loggingStastic();

    TrafficAutomate trafficAutomate = key2Status.get(key);
    if(trafficAutomate == null)
    {
        trafficAutomate = new TrafficAutomate();
        TrafficAutomate oldAutomate =
key2Status.putIfAbsent(key, trafficAutomate);
        if(oldAutomate != null)
        {
            trafficAutomate = oldAutomate;
        }
    }
    long threadId = Thread.currentThread().getId();

    ActionEnum action = trafficAutomate.enQueue(threadId);

    if(action == DECLINE)
    {
        return false;
    }
    else if (action == ACCEPT)
    {
```

```
            return true;
        }
        //Blocking status means some other thread are working on
this key, so just wait till timeout
        long start = System.currentTimeMillis();
        long span = 0;
        do {
            _nonExceptionSleep(NAP_TIME_IN_MILL);

            if(trafficAutomate.isHead(threadId))
            {
                return true;
            }

            span = System.currentTimeMillis() - start;
        }while(span <= timeout);

        //remove head so that it won't block the queue for too
long
        trafficAutomate.evictHeadByForce(threadId);

        return true;
    }
```

deQueue操作

deQueue操作首先从ConcurrentHashMap获取改请求所对应的状态机，接着获取该线程的线程id，对状态机进行deQueue操作。

enQueue代码如下：

```
public void deQueue(T key) {
    TrafficAutomate trafficAutomate = key2Status.get(key);
```

```
    if(trafficAutomate == null)
    {
        logger.error("key {} doesn't exist ", key);
        return;
    }

    long threadId = Thread.currentThread().getId();

    trafficAutomate.deQueue(threadId);
}
```

源代码

完整源代码可以在QueueCoordinator获取。

[1] Rabbit MQ, Highly Available Queues.
[2] IBM Knowledge Center, Introduction to message queuing.
[3] Wikipedia, Serializability.
[4] Hadoop, ZooKeeper Recipes and Solutions.
[5] Apache Kafka.
[6] Lamport L, Paxos Made Simple.