削峰：

削峰填谷的应用场景：

举个业务场景的栗子，秒杀业务：

上游发起下单操作

下游完成秒杀业务逻辑（库存检查，库存冻结，余额检查，余额冻结，订单生成，余额扣减，库存扣减，生成流水，余额解冻，库存解冻）

上游下单业务简单，每秒发起了10000个请求，下游秒杀业务复杂，每秒只能处理2000个请求，很有可能上游不限速的下单，导致下游系统被压垮，引发雪崩。

为了避免雪崩，常见的优化方案有两种：

1. 业务上游队列缓冲，限速发送
   1. 问答，
   2. 分层过滤
2. 业务下游队列缓冲，限速执行
   1. 排队 MQ

rabbitmq提供了一种服务质量保障功能，即在非自动确认消息的前提下，如果一定数目的消息未被确认，不进行消费新的消息。

使用 basicqos方法：

在消费端进行使用。 0 1 false

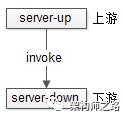
prefetSize：0

prefetCount：这个值一般在设置为非自动ack的情况下生效，一般大小为1

global： true是channel级别， false是消费者级别

注意：我们要使用非自动ack

问：站点与服务，服务与服务上下游之间，一般如何通讯？  
  
答：有两种常见的方式

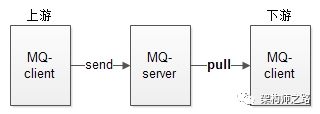


上游将消息发给MQ，MQ将消息推送给下游

问：为什么会有流量冲击？  
  
答：不管采用“直接调用”还是“MQ推送”，都有一个缺点，下游消息接收方无法控制到达自己的流量，如果调用方不限速，很有可能把下游压垮。

问：MQ怎么改能缓冲流量？

答：由MQ-server推模式，升级为MQ-client拉模式。



如果你看过秒杀系统的流量监控图的话，你会发现它是一条直线，就在秒杀开始那一秒是一条很直很直的线，这是因为秒杀请求在时间上高度集中于某一特定的时间点。这样一来，就会导致一个特别高的流量峰值，它对资源的消耗是瞬时的。

但是对秒杀这个场景来说，最终能够抢到商品的人数是固定的，也就是说100人和10000人发起请求的结果都是一样的，并发度越高，无效请求也越多。

但是从业务上来说，秒杀活动是希望更多的人来参与的，也就是开始之前希望有更多的人来刷页面，但是真正开始下单时，秒杀请求并不是越多越好。因此我们可以设计一些规则，让并发的请求更多地延缓，而且我们甚至可以过滤掉一些无效请求。

为什么要削峰

为什么要削峰呢？或者说峰值会带来哪些坏处？

我们知道服务器的处理资源是恒定的，你用或者不用它的处理能力都是一样的，所以出现峰值的话，很容易导致忙到处理不过来，闲的时候却又没有什么要处理。但是由于要保证服务质量，我们的很多处理资源只能按照忙的时候来预估，而这会导致资源的一个浪费。

这就好比因为存在早高峰和晚高峰的问题，所以有了错峰限行的解决方案。

削峰的存在，一是可以让服务端处理变得更加平稳，二是可以节省服务器的资源成本。

针对秒杀这一场景，削峰从本质上来说就是更多地延缓用户请求的发出，以便减少和过滤掉一些无效请求，它遵从“请求数要尽量少”的原则。

今天，我就来介绍一下流量削峰的一些操作思路：**排队、答题、分层过滤**。

这几种方式都是无损（即不会损失用户的发出请求）的实现方案，当然还有些有损的实现方案，包括我们后面要介绍的关于稳定性的一些办法，比如限流和机器负载保护等一些强制措施也能达到削峰保护的目的，当然这都是不得已的一些措施，因此就不归类到这里了。

**排队**

要对流量进行削峰，最容易想到的解决方案就是用消息队列来缓冲瞬时流量，把同步的直接调用转换成异步的间接推送，中间通过一个队列在一端承接瞬时的流量洪峰，在另一端平滑地将消息推送出去。在这里，消息队列就像“水库”一样，拦蓄上游的洪水，削减进入下游河道的洪峰流量，从而达到减免洪水灾害的目的。

用消息队列来缓冲瞬时流量的方案，如下图所示：

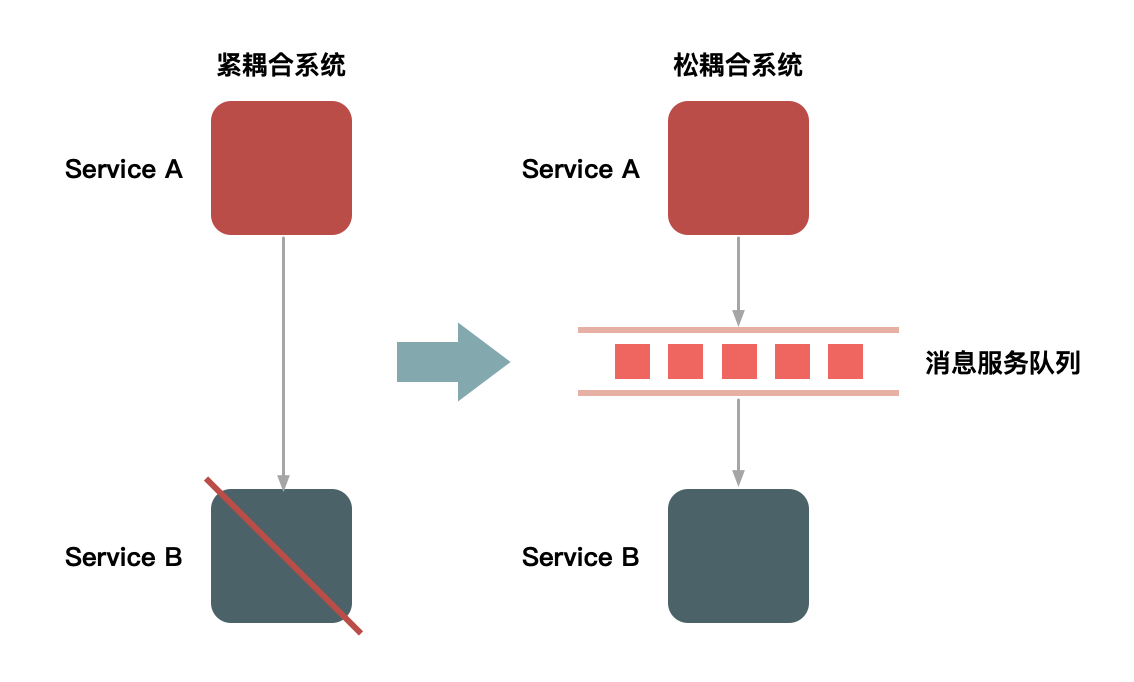


图1 用消息队列来缓冲瞬时流量

但是，如果流量峰值持续一段时间达到了消息队列的处理上限，例如本机的消息积压达到了存储空间的上限，消息队列同样也会被压垮，这样虽然保护了下游的系统，但是和直接把请求丢弃也没多大的区别。就像遇到洪水爆发时，即使是有水库恐怕也无济于事。

除了消息队列，类似的排队方式还有很多，例如：

利用线程池加锁等待也是一种常用的排队方式；

先进先出、先进后出等常用的内存排队算法的实现方式；

把请求序列化到文件中，然后再顺序地读文件（例如基于MySQL binlog的同步机制）来恢复请求等方式。

可以看到，这些方式都有一个共同特征，就是把“一步的操作”变成“两步的操作”，其中增加的一步操作用来起到缓冲的作用。

说到这里你可能会说，这样一来增加了访问请求的路径啊，并不符合我们介绍的“4要1不要”原则。没错，的确看起来不太合理，但是如果不增加一个缓冲步骤，那么在一些场景下系统很可能会直接崩溃，所以最终还是需要你做出妥协和平衡。

**答题**

你是否还记得，最早期的秒杀只是纯粹地刷新页面和点击购买按钮，它是后来才增加了答题功能的。那么，为什么要增加答题功能呢？

这主要是为了增加购买的复杂度，从而达到两个目的。

第一个目的是防止部分买家使用秒杀器在参加秒杀时作弊。2011年秒杀非常火的时候，秒杀器也比较猖獗，因而没有达到全民参与和营销的目的，所以系统增加了答题来限制秒杀器。增加答题后，下单的时间基本控制在2s后，秒杀器的下单比例也大大下降。

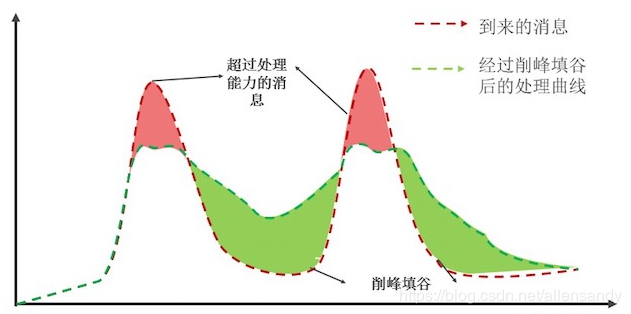
**分层过滤**

前面介绍的排队和答题要么是少发请求，要么对发出来的请求进行缓冲，而针对秒杀场景还有一种方法，就是对请求进行分层过滤，从而过滤掉一些无效的请求。分层过滤其实就是采用“漏斗”式设计来处理请求的，如下图所示。

**削峰填谷**

请求的到来，往往是没有规律的。

例如，某应用的处理能力是每秒 10 个请求。在某一秒，突然到来了 30 个请求，而接下来两秒，都没有请求到达。在这种情况下，如果直接拒绝 20 个请求，应用在接下来的两秒就会空闲。所以，需要把请求突刺均摊到一段时间内，让系统负载保持在请求处理水位之内，同时尽可能地处理更多请求，从而起到“削峰填谷”的效果。



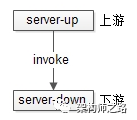
上图中，红色的部分代表超出消息处理能力的部分。观察得出，消息突刺往往都是瞬时的、不规律的，其后一段时间系统往往都会有空闲资源。把红色的那部分消息平摊到后面空闲时去处理，这样既可以保证系统负载处在一个稳定的水位，又可以尽可能地处理更多消息。通过配置流控规则，可以达到消息匀速处理的效果。

**分析问答**

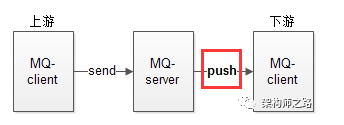
  1、问：站点与服务，服务与服务上下游之间，一般如何通讯？

        答：有两种常见的方式

       一种是“直接调用”，通过RPC框架，上游直接调用下游：



       还有一种，在某些业务场景之下（具体哪些业务场景，见《到底什么时候该使用MQ？》），可以采用“MQ推送”，上游将消息发给MQ，MQ将消息推送给下游。



    这两种都有问题：不管采用“直接调用”还是“MQ推送”，都有一个缺点，下游消息接收方无法控制到达自己的流量，如果调用方不限速，很有可能把下游压垮。

**2 问：以上两种为什么会有流量冲击？**

        答：不管采用“直接调用”还是“MQ推送”，都有一个缺点，下游消息接收方无法控制到达自己的流量，如果调用方不限速，很有可能把下游压垮。

      举个栗子，秒杀业务：

      上游发起高并发的下单操作。

      下游完成秒杀业务逻辑（库存检查，库存冻结，余额检查，余额冻结，订单生成，余额扣减，库存扣减，生成流水，余额解冻，库存解冻）。

       上游下单业务简单，每秒发起了10000个请求，下游秒杀业务复杂，每秒只能处理2000个请求，很有可能上游不限速的下单，导致下游系统被压垮，引发雪崩。

**3、问：MQ怎么改能缓冲流量**？

         答：由MQ-server推模式，升级为MQ-client拉模式。MQ-client根据自己的处理能力，每隔一定时间，或者每次拉取若干条消息，实施流控，达到保护自身的效果。并且这是MQ提供的通用功能，无需上下游修改代码。

**4、问：如果上游发送流量过大，MQ提供拉模式确实可以起到下游自我保护的作用，会不会导致消息在MQ中堆积？**  
         答：下游MQ-client拉取消息，消息接收方能够批量获取消息，需要下游消息接收方进行优化，方能够提升整体吞吐量，例如：批量写。

# RabbitMQ 队列消息的条数限制、队列字节长度限制、队列溢出行为方式

# 策略地址：<https://www.rabbitmq.com/parameters.html#policies>

# //管理策略

# // 得到mq的连接

# Connection conn = RabbitMqUtil.getConnection();

# // 通过连接得到一个通道

# Channel channel = conn.createChannel();

# //管理策略

# // 声明了一个消息队列

# HashMap<String, Object> map = new HashMap<>();

# // 设置队列最大的条数 10条

# map.put("x-max-length", 10);

# // 设置队列溢出方式 保留前10条

# //定义溢出行为-是从头上删除消息还是拒绝新发布

# map.put("x-overflow", "reject-publish");

# RabbitMQ有两种对队列长度的限制方式

# 对队列中消息的条数进行限制 x-max-length

# 对队列中消息的总量进行限制 x-max-length-bytes

# MQ性能问题

一． 要避免流控机制触发

# 服务端默认配置是当内存使用达到40%，磁盘空闲空间小于50M，即启动内存报警，磁盘报警；报警后服务端触发流控（flow control）机制。一般地，当发布端发送消息速度快于订阅端消费消息的速度时，队列中堆积了大量的消息，导致报警，就会触发流控机制。

# 触发流控机制后，RabbitMQ服务端接收发布来的消息会变慢，使得进入队列的消息减少；与此同时RabbitMQ服务端的消息推送也会受到极大的影响，测试发现，服务端推送消息的频率会大幅下降，等待下一次推送的时间，有时等1分钟，有时5分钟，甚至30分钟。

# 一旦触发流控，将导致RabbitMQ服务端性能恶化，推送消息也会变得非常缓慢；因此要做好数据设计，使得发送速率和接收速率保持平衡，而不至于引起服务器堆积大量消息，进而引发流控。通过增加服务器集群节点，增加消费者，来避免流控发生，治标不治本，而且成本高。

二． 从底层取数据一定要非常及时

# 订阅端每隔500MS调用一次amqp\_consume\_message接口函数从socket上获取数据，正常情况下，服务器每次会推送几百条消息，而且推送的频率会比较高；导致订阅端的本机socket缓冲区会很快存满，导致很多消息无法进行缓存，而被丢掉

# 

# 三． 消息大小不要超过4MB

# 客户端与RabbitMQ服务端的最大帧是128K，但消息大小却可支持数MB，这是可能是因为底层做了拆包组包的，目前我还未查看底层代码。

# 用线程来模拟50个发布者和50个订阅者；消息包大小由1K到10MB，当包大小达到4.5MB时，服务器的性能出现明显的异常，传输率尤其是每秒订阅消息的数量，出现波动，不稳定；同时有一部分订阅者的TCP连接出现断开的现象。可能是客户端底层或者RabbitMQ服务端在进行拆包，组包的时候，出现了明显的压力，而导致异常的发生。

# 超过4MB的消息，最好先进行分包。

# 第一次修改本节

# 由于用线程模拟大量发布者，且是服务器单节点，受客户端主机网卡的限制，发布线程没有速度控制，导致有大量数据发送，服务器带宽下行速率也满负荷，上行带宽却明显低于下行速率，导致服务器内存有大量消息堆积，进而触发RabbitMQ服务器paging操作，才出现了上述不稳定和订阅者断开现象。对发布端做适当流量控制，断开连接现象不再出现，但每秒消息数仍然不稳定。

# //管理策略

# // 声明了一个消息队列

# HashMap<String, Object> map = new HashMap<>();

# //标志队列中的消息存活时间，也就是说队列中的消息超过了指定时间会被删除(数字类型，标志时间，以豪秒为单位)

# map.put("x-message-ttl", 1 \* 24 \* 60 \* 60 \* 10000);

# //队列自身的空闲存活时间，当前的queue在指定的时间内，没有consumer、basic.get也就是未被访问，就会被删除。(数字类型，标志时间，以豪秒为单位)

# map.put("x-expires",60 \* 60 \* 10000);

# //最大长度和最大占用空间，设置了最大长度的队列，在超过了最大长度后进行插入会删除之前插入的消息为本次的留出空间（默认操作是如此，我们可以设置overflow来改变，例如用在并发缓冲时）,相应的最大占用大小也是这个道理，当超过了这个大小的时候，会删除之前插入的消息为本次的留出空间。

# map.put("x-max-length-bytes",100 \* 1024 \* 1024);

# // 设置队列最大的条数 10条

# map.put("x-max-length", 10);

# //Queue上附加优先级属性

# map.put("x-max-priority", 5);

# // 设置队列溢出方式 保留前10条

# //定义溢出行为-是从头上删除消息还是拒绝新发布

# ////队列超出最大长度的处理方案 ，队列溢出的默认处理方案：drop-head (default) 或者拒绝消息 reject-publish 我们做并发限流的时候需要设置为超出队列拒绝

# map.put("x-overflow", "reject-publish");

# 

# /\*

# 消息因为超时或超过限制在队列里消失，这样我们就丢失了一些消息，也许里面就有一些是我们做需要获知的。而rabbitmq的死信功能则为我们带来了解决方案。设置了dead letter exchange与dead letter routingkey（要么都设定，要么都不设定）那些因为超时或超出限制而被删除的消息会被推动到我们设置的exchange中，再根据routingkey推到queue中.

# \*/

# map.put("x-dead-letter-exchange","publicsubscrible");

# map.put("x-dead-letter-routing-key","publicsubscrible\_queue");

# 

# channel.queueDeclare(queueName, true, false, false, map);