削峰：

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

**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。

# //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);