我们在devicepush这个服务中完成的是我们平台内部逻辑代码的设计，其中主要是完成各种方式的推送。类似于设别推送通道，第三方推送平台的推送。

我们在这个服务中首先是生产者消费数据；中间使用cmq来业务解耦，并消峰填谷。在消费者端在去拉数据进行消费。（我们这次的代码就是构建比较灵活，即使新加入一些推送方式改的代码量也比较小）。

在这个服务中我们的服务平台有qq阅读，起点中文网，红袖添香等小说网站。每个小说网站所要求的推送方式都不。但是传入过来的数据格式是相同的，虽然最终的推送方式是不相同的，但是基本的业务逻辑是一样的。

在service层面，我们使用了一个**基于annotation方式的通用报文处理框架**，传统的方法通过配置文件，关联到相应的处理类，但是新增报文的时候需要修改配置文件显得十分不灵活。

所以我们使用了annotation的方式避免了文件的配置，易于扩展。

大体流程如下：

1. 构建annotation接口，用于表示业务处理类
2. **定义了一个hander 接口，里面有一个hander（）方法**
3. **构建报文处理基类（BaseMiPushHandler），这个类实现hander接口。**

在这个基类中，实现了hander方法，首先检测传递过来的参数是否满足要求。如果满足要求就将消息封装起来，最后调用**生产者接口将消息发送出去**。

1. **构建一个系统启动时发现并注册消息处理类**。HandlerManager

在这个类里面，我们首先定义了一个concurrenthashmap 用于存储要注册的类，

其中key用于存储了我们推送的类型（这个类是枚举形式的），类似与小米通道全量推送，华为通道单推等。Value是用于存储hander类。这个类实现了InitializingBean, ApplicationContextAware，我们调用InitializingBean的afterPropertiesSet（）方法，用它在资源加载完以后，通过applicationContext.getBeansOfType(Handler.class);获取了hander；最后通过遍历hander，在里面调用handler.getClass().getAnnotation(HandlerType.class)找到对应的处理类，注册到map中。

1. 构建报文处理类

这个报文处理类继承了报文处理基类。在这个报文处理类中使用注解，注解里面写有注册时候需要用到的信息。

**本次推送主要分为两种：设别推送和第三方推送。**

**在这两种推送的接口中，我们对传过来的信息封装。**

Handler handler = handlerManager.getHandler(req.getPlatformEnum(), req.getBizTypeEnum());

我们定义了一个类PushTemplate，在这个类中我们实例化了前面所说的**注册消息处理类，通过传入的参数获取具体的报文处理类。（传入的参数有消息推送通道等）。然后在这个方法里面实现了一种类似于钩子一样的操作。在前面判断传入的是否能够获取报文处理类的实例，如果不能则报错。然后调用hander类的hander方法。其实调用这个方法后我们最终就是进入了相关报文处理类的实例里面，在这个里面我们开始调用生产者的接口去生产消息。**

**但是现在有一个问题就是我们具有有全量推送和非全量推送的方式，还分为设备推送和第三方推送，那么我们怎么保证在推送的时候找到正确的生产者呢？**

**做法：各个生产者在都实现了一个IPublisher的接口，并且通过@PostConstruct（ 被@PostConstruct修饰的方法会在服务器加载Servlet的时候运行）完成每个具体生产者的注册。所以我们最开始调用生产者数据的时候，通过传入的参数可以具体加载某个生产者去生产。**

**在生产者内部我们就是开一个线程池去将消息塞入到消息中间件cmq中，对于全量的推送，我们分为一片一片的推送，每次推送15个。如果失败了，我们就重新尝试三次。最终失败就记录下来最后上报到loggerserver。**

**在消费者端通过从cmq中拉取消息，对于不同的推送方式，我们只需去实现官方提供的api即可。（主要是启动一个http服务，首先鉴权，然后封装消息，最后对消息进行推送。）**

**Cmq的高可用保证：**

CMQ 在每条消息返回到用户时，将消息数据备份 3 份写到不同物理机上，当某台物理机出故障时，后台数据复制机制能够对数据快速迁移，保证用户数据3 个备份可用，可靠保证达 99.999999%，业务连续可用性99.95%承诺，引入 Raft 算法实现数据强一致性。

**Raft算法**

**实现 Raft 算法两个最重要的事是：选主和复制日志。**

**Rfat**每个节点有**三种状态：Follower，Candidate，Leader**，状态之间是互相转换的。

**2. 选主 Leader Election**

**2.1 正常情况下选主**

假设现在有n个节点，n个节点一开始的状态都是 Follower。在一个节点倒计时结束 (Timeout) 后，这个节点的状态变成 Candidate 开始选举，它给其他几个节点发送选举请求 (RequestVote)。

其他四个节点都返回成功，这个节点的状态由 Candidate 变成了 Leader，并在每个一小段时间后，就给所有的 Follower 发送一个 Heartbeat 以保持所有节点的状态，Follower 收到 Leader 的 Heartbeat 后重设 Timeout。这是最简单的选主情况，**只要有超过一半的节点投支持票了，Candidate 才会被选举为 Leader。**

**2.2 Leader 出故障情况下的选主**

一开始已经有一个 Leader，所有节点正常运行. Leader 出故障挂掉了，其他四个 Follower 将进行重新选主。选举流程与正常的选举相同。在 Raft 里，第几轮选举是有记录的，重新加入的 Leader 是第一轮选举 (Term 1) 选出来的，而现在的 Leader 则是 Term 2，所有原来的 Leader 会自觉降级为 Follower

**2.3 多个 Candidate 情况下的选主**

如果同时有两个候选人向大家邀票，这时通过类似加时赛来解决，两个候选者在一段timeout比如300ms互相不服气的等待以后，因为双方得到的票数是一样的，一半对一半，那么在300ms以后，再由这两个候选者发出邀票，这时同时的概率大大降低，那么首先发出邀票的的候选者得到了大多数同意，成为领导者Leader，而另外一个候选者后来发出邀票时，那些Follower选民已经投票给第一个候选者，不能再投票给它，它就成为落选者了，最后这个落选者也成为普通Follower一员了。

3 复制日志

3.1 正常情况下

一开始，Leader 和 两个 Follower 都没有任何数据。

客户端发送请求给 Leader，储存数据 “AAA”，Leader 先将数据写在本地日志，这时候数据还是 **Uncommitted** (还没最终确认，红色表示)

Leader 给两个 Follower 发送 AppendEntries 请求，数据在 Follower 上没有冲突，则将数据暂时写在本地日志，Follower 的数据也还是 Uncommitted。

Follower 将数据写到本地后，返回 OK。Leader 收到后成功返回，**只要收到的成功的返回数量超过半数 (包含Leader)**，Leader 将数据 “sally” 的状态改成 Committed。( 这个时候 Leader 就可以返回给客户端了)

Leader 再次给 Follower 发送 AppendEntries（附录） 请求，收到请求后，Follower 将本地日志里 Uncommitted 数据改成 Committed。这样就完成了一整个复制日志的过程，三个节点的数据是一致的，

3.2 网络故障下

如果在这一过程中，发生了网络分区或者网络通信故障，使得Leader不能访问大多数Follwers了，那么Leader只能正常更新它能访问的那些Follower服务器，而大多数的服务器Follower因为没有了Leader，他们重新选举一个候选者作为Leader，然后这个Leader作为代表于外界打交道，如果外界要求其添加新的日志，这个新的Leader就按上述步骤通知大多数Followers，如果这时网络故障修复了，那么原先的Leader就变成Follower，在失联阶段这个老Leader的任何更新都不能算commit，都回滚，接受新的Leader的新的更新。

**Cmq的生产可靠**

客户端生产的消息在set 中超过半数的broker 刷盘成功后会返回确认消息告知生产消息成功。如果在一定时间之内客户端没有收到确认信息需要重试来确保消息发送成功。

**可靠生产带来的一个问题就是消息的重复，在网络异常等情况下很可能CMQ broker已经存储消息成功只是确认包在网络上丢失了，这样客户端重试生产后，在broker上存在两条重复的消息。**

**Cmq对于消息等幂性有自己的解决方案:**

MQ消息发送上半场，即上图中的1-3

1，发送端MQ-client将消息发给服务端MQ-server

2，服务端MQ-server将消息落地

3，服务端MQ-server回ACK给发送端MQ-client

如果3丢失，发送端MQ-client超时后会重发消息，可能导致服务端MQ-server收到重复消息。

**此时重发是MQ-client发起的，消息的处理是MQ-server，为了避免步骤2落地重复的消息，对每条消息，MQ系统内部必须生成一个inner-msg-id，作为去重和幂等的依据，这个内部消息ID的特性是：**

**（1）全局唯一**

**（2）MQ生成，具备业务无关性，对消息发送方和消息接收方屏蔽**

**Cmq 怎么保证消息不被重复消费？**

消费者拉取消息时会指定当前消息的隐藏时间，在隐藏时间内消费者比较显式的对消息进行确认删除，如果超过隐藏时间没有主动删除，此条消息将重新对外可见，可以继续消费。（具体意思就是你取出来一条消息，这条消息对其他人就不可见了）

**存储可靠**

CMQ SET中一个节点为leader 其他节点为follower，leader 负责所有消息的生产消费。当生产消息到达leader 节点后，通过raft 一致性模块将请求顺序写raft log 并同步刷盘，同时将构造好的raft log 按顺序通过网络发送到其他follower节点，follower节点同步刷盘并返回成功。当leader 收到过半数的节点同步成功信息后将此条请求提交到mq 处理状态机，由mq 状态机将请求应用到相应queue。

**五、如何保证消息消费时的幂等性**

其实消息重复消费的主要原因在于回馈机制（RabbitMQ是ack，Kafka是offset)，在某些场景中我们采用的回馈机制不同，原因也不同，例如消费者消费完消息后回复ack, 但是刚消费完还没来得及提交系统就重启了，这时候上来就pull消息的时候由于没有提交ack或者offset，消费的还是上条消息。

那么如何怎么来保证消息消费的幂等性呢？实际上我们只要保证多条相同的数据过来的时候只处理一条或者说多条处理和处理一条造成的结果相同即可，但是具体怎么做要根据业务需求来定，例如入库消息，先查一下消息是否已经入库啊或者说搞个唯一约束啊什么的，还有一些是天生保证幂等性就根本不用去管，例如redis就是天然幂等性。

还有一个问题，消费者消费消息的时候在某些场景下要放过消费不了的消息，遇到消费不了的消息通过日志记录一下或者搞个什么措施以后再来处理，但是一定要放过消息，因为在某些场景下例如spring-rabbitmq的默认回馈策略是出现异常就没有提交ack，导致了一直在重发那条消费异常的消息，而且一直还消费不了，这就尴尬了，后果你会懂的。

**六、如何保证消息的可靠性传输？**

由于笔者只使用和实践过RabbitMQ和Kafka，RocketMQ和ActiveMQ了解的不深，所以分析一下RabbitMQ和Kafka的消息可靠性传输的问题。、

（一）RabbitMQ

（1）生产者弄丢了数据

　　生产者将数据发送到RabbitMQ的时候，可能数据就在半路给搞丢了，因为网络啥的问题，都有可能。此时可以选择用RabbitMQ提供的事务功能，就是生产者发送数据之前开启RabbitMQ事务（channel.txSelect），然后发送消息，如果消息没有成功被RabbitMQ接收到，那么生产者会收到异常报错，此时就可以回滚事务（channel.txRollback），然后重试发送消息；如果收到了消息，那么可以提交事务（channel.txCommit）。但是问题是，RabbitMQ事务机制一搞，基本上吞吐量会下来，因为太耗性能。

所以一般来说，如果你要确保说写RabbitMQ的消息别丢，可以开启confirm模式，在生产者那里设置开启confirm模式之后，你每次写的消息都会分配一个唯一的id，然后如果写入了RabbitMQ中，RabbitMQ会给你回传一个ack消息，告诉你说这个消息ok了。如果RabbitMQ没能处理这个消息，会回调你一个nack接口，告诉你这个消息接收失败，你可以重试。而且你可以结合这个机制自己在内存里维护每个消息id的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。

事务机制和cnofirm机制最大的不同在于，事务机制是同步的，你提交一个事务之后会阻塞在那儿，但是confirm机制是异步的，你发送个消息之后就可以发送下一个消息，然后那个消息RabbitMQ接收了之后会异步回调你一个接口通知你这个消息接收到了。

所以一般在生产者这块避免数据丢失，都是用confirm机制的。

（2）RabbitMQ弄丢了数据

就是RabbitMQ自己弄丢了数据，这个你必须开启RabbitMQ的持久化，就是消息写入之后会持久化到磁盘，哪怕是RabbitMQ自己挂了，恢复之后会自动读取之前存储的数据，一般数据不会丢。除非极其罕见的是，RabbitMQ还没持久化，自己就挂了，可能导致少量数据会丢失的，但是这个概率较小。

设置持久化有两个步骤，第一个是创建queue的时候将其设置为持久化的，这样就可以保证RabbitMQ持久化queue的元数据，但是不会持久化queue里的数据；第二个是发送消息的时候将消息的deliveryMode设置为2，就是将消息设置为持久化的，此时RabbitMQ就会将消息持久化到磁盘上去。必须要同时设置这两个持久化才行，RabbitMQ哪怕是挂了，再次重启，也会从磁盘上重启恢复queue，恢复这个queue里的数据。

而且持久化可以跟生产者那边的confirm机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者ack了，所以哪怕是在持久化到磁盘之前，RabbitMQ挂了，数据丢了，生产者收不到ack，你也是可以自己重发的。

哪怕是你给RabbitMQ开启了持久化机制，也有一种可能，就是这个消息写到了RabbitMQ中，但是还没来得及持久化到磁盘上，结果不巧，此时RabbitMQ挂了，就会导致内存里的一点点数据会丢失。

（3）消费端弄丢了数据

RabbitMQ如果丢失了数据，主要是因为你消费的时候，刚消费到，还没处理，结果进程挂了，比如重启了，那么就尴尬了，RabbitMQ认为你都消费了，这数据就丢了。

这个时候得用RabbitMQ提供的ack机制，简单来说，就是你关闭RabbitMQ自动ack，可以通过一个api来调用就行，然后每次你自己代码里确保处理完的时候，再程序里ack一把。这样的话，如果你还没处理完，不就没有ack？那RabbitMQ就认为你还没处理完，这个时候RabbitMQ会把这个消费分配给别的consumer去处理，消息是不会丢的。

（二）Kafka

（1）消费端弄丢了数据

唯一可能导致消费者弄丢数据的情况，就是说，你那个消费到了这个消息，然后消费者那边自动提交了offset，让kafka以为你已经消费好了这个消息，其实你刚准备处理这个消息，你还没处理，你自己就挂了，此时这条消息就丢咯。

大家都知道kafka会自动提交offset，那么只要关闭自动提交offset，在处理完之后自己手动提交offset，就可以保证数据不会丢。但是此时确实还是会重复消费，比如你刚处理完，还没提交offset，结果自己挂了，此时肯定会重复消费一次，自己保证幂等性就好了。

生产环境碰到的一个问题，就是说我们的kafka消费者消费到了数据之后是写到一个内存的queue里先缓冲一下，结果有的时候，你刚把消息写入内存queue，然后消费者会自动提交offset。

然后此时我们重启了系统，就会导致内存queue里还没来得及处理的数据就丢失了

（2）kafka弄丢了数据

这块比较常见的一个场景，就是kafka某个broker宕机，然后重新选举partiton的leader时。大家想想，要是此时其他的follower刚好还有些数据没有同步，结果此时leader挂了，然后选举某个follower成leader之后，他不就少了一些数据？这就丢了一些数据啊。

生产环境也遇到过，我们也是，之前kafka的leader机器宕机了，将follower切换为leader之后，就会发现说这个数据就丢了。

所以此时一般是要求起码设置如下4个参数：

给这个topic设置replication.factor参数：这个值必须大于1，要求每个partition必须有至少2个副本。

在kafka服务端设置min.insync.replicas参数：这个值必须大于1，这个是要求一个leader至少感知到有至少一个follower还跟自己保持联系，没掉队，这样才能确保leader挂了还有一个follower吧。

在producer端设置acks=all：这个是要求每条数据，必须是写入所有replica之后，才能认为是写成功了。

在producer端设置retries=MAX（很大很大很大的一个值，无限次重试的意思）：这个是要求一旦写入失败，就无限重试，卡在这里了。

（3）生产者会不会弄丢数据

如果按照上述的思路设置了ack=all，一定不会丢，要求是，你的leader接收到消息，所有的follower都同步到了消息之后，才认为本次写成功了。如果没满足这个条件，生产者会自动不断的重试，重试无限次。

六、如何保证消息的顺序性

因为在某些情况下我们扔进MQ中的消息是要严格保证顺序的，尤其涉及到订单什么的业务需求，消费的时候也是要严格保证顺序，不然会出大问题的。

先看看顺序会错乱的俩场景

rabbitmq：一个queue，多个consumer，这不明显乱了

kafka：一个topic，一个partition，一个consumer，内部多线程，这不也明显乱了

　　如何来保证消息的顺序性呢？

rabbitmq：拆分多个queue，每个queue一个consumer，就是多一些queue而已，确实是麻烦点；或者就一个queue但是对应一个consumer，然后这个consumer内部用内存队列做排队，然后分发给底层不同的worker来处理。

kafka：一个topic，一个partition，一个consumer，内部单线程消费，写N个内存queue，然后N个线程分别消费一个内存queue即可。

七、如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时怎么解决？

（一）、大量消息在mq里积压了几个小时了还没解决

几千万条数据在MQ里积压了七八个小时，从下午4点多，积压到了晚上很晚，10点多，11点多

这个是我们真实遇到过的一个场景，确实是线上故障了，这个时候要不然就是修复consumer的问题，让他恢复消费速度，然后傻傻的等待几个小时消费完毕。这个肯定不能在面试的时候说吧。

一个消费者一秒是1000条，一秒3个消费者是3000条，一分钟是18万条，1000多万条，所以如果你积压了几百万到上千万的数据，即使消费者恢复了，也需要大概1小时的时间才能恢复过来。

一般这个时候，只能操作临时紧急扩容了，具体操作步骤和思路如下：

先修复consumer的问题，确保其恢复消费速度，然后将现有cnosumer都停掉。

新建一个topic，partition是原来的10倍，临时建立好原先10倍或者20倍的queue数量。

然后写一个临时的分发数据的consumer程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的10倍数量的queue。

接着临时征用10倍的机器来部署consumer，每一批consumer消费一个临时queue的数据。

这种做法相当于是临时将queue资源和consumer资源扩大10倍，以正常的10倍速度来消费数据。

等快速消费完积压数据之后，得恢复原先部署架构，重新用原先的consumer机器来消费消息。

（二）、消息队列过期失效问题

假设你用的是rabbitmq，rabbitmq是可以设置过期时间的，就是TTL，如果消息在queue中积压超过一定的时间就会被rabbitmq给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在mq里，而是大量的数据会直接搞丢。

这个情况下，就不是说要增加consumer消费积压的消息，因为实际上没啥积压，而是丢了大量的消息。我们可以采取一个方案，就是批量重导，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上12点以后，用户都睡觉了。

这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入mq里面去，把白天丢的数据给他补回来。也只能是这样了。

假设1万个订单积压在mq里面，没有处理，其中1000个订单都丢了，你只能手动写程序把那1000个订单给查出来，手动发到mq里去再补一次。

(三)、消息队列满了怎么搞？

如果走的方式是消息积压在mq里，那么如果你很长时间都没处理掉，此时导致mq都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，消费一个丢弃一个，都不要了，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。