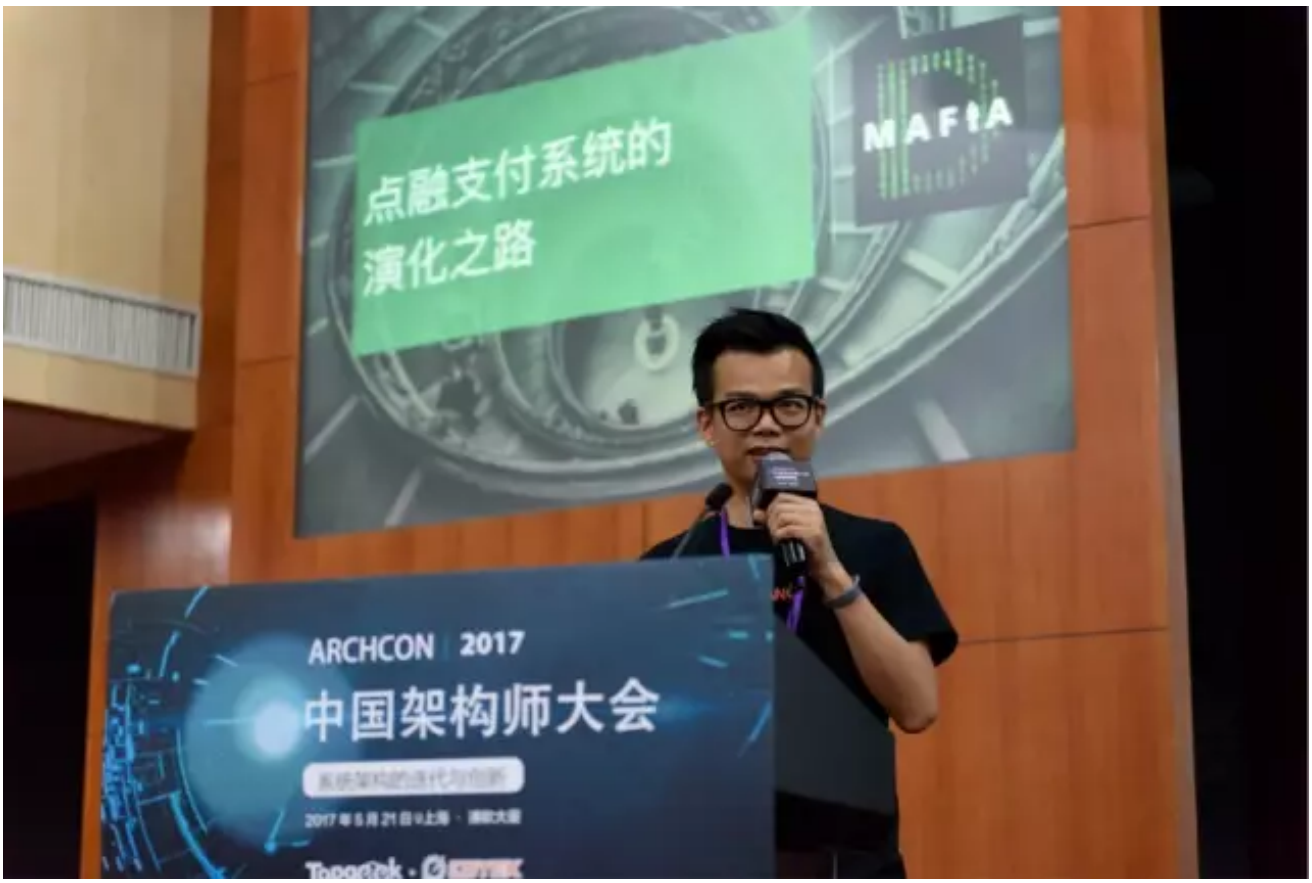
点融支付系统架构的演进丨ArchCon2017中国架构师大会

原创：姚晨 点融黑帮 2017-05-24

5月21日，由上海最大的高端技术社群TopGeek和汇智Tek联合主办的中国架构师大会（ArchCon2017）落幕。
来自点融网、阿里巴巴、携程、美团、新达达等行业的专家就“系统架构的迭代与创新”进行了探讨。



点融科技资深软件工程师姚晨受邀参加本次会议，分享了在构建点融支付系统的心路历程。

我叫姚晨，我来自于点融网。我想作为非架构师的我，在这里给大家分享一下在构建点融支付系统的心路历程，也算是在各位架构师面前班门弄斧。

在点融，支付最开始是作为一个模块存在于点融的大系统之中，它缺乏统一实现、扩展性较差，我们有大概有三到四个支付渠道，每个支付渠道都有自己的端到端的实现。相同的加解密、报文处理和机制有各自的实现。我们需要接入更多的渠道的时候，就在原有的支付模块结构中，只能通过重复代码来实现。这样的话，我们需要一个统一的支付网关一样的存在，统一管理所有支付共通的业务逻辑。

业务逻辑全靠代码固化，这里说的是比如支付渠道的路由选择，当有支付渠道不可用的时候，或者说出于费率或者成功率考虑，类似这样的路由逻辑，其实我们都是通过代码固化的。另外就是说，当我们有新渠道或者新的支付方式、交互方式上线的时候，我们希望做一定的灰度开启，这也是通过代码固化来实现的，也就是说，当我们有调整，比如说我们希望把1%的用户提高到10%甚至提高到50%甚至全部开放，这时候我们需要的是重新修改参数，打包部署上线。这对于一个互联网公司来说，我觉得是无法接受的。

另外，随着点融业务的扩展，我们将原有的点融大系统逐步拆分成多个业务子系统。这时或多或少所有的业务子系统都需要一定的支付能力。这时候如果还是以支付模块存在的话，那就意味着说，每一个子业务系统都需要copy这样一个支付模块。所以我们觉得，那我们就把这个支付模块给变成一个系统。

那为什么支付需要服务化，或者说为什么支付可以服务化？我觉得很简单，其实它就是一个基础服务，因为点融所有的业务系统都需要支付，所以这是一个足够强大的理由说，我们去把这个东西做成一个单独的服务。

支付本身来说它具有清晰的上下文边界，也就是说支付就是支付，充值也好提现也好代扣代付也好，它跟业务紧密相关，但是它跟业务系统却又是松耦合的，是一个理想的服务化对象。

我们需要提供一个演进式的设计，以便适应持续变化的业务需求。这里说的是什么？一开始我们只有两种支付渠道一种交互方式。随着业务的演化，随着支付渠道通路的丰富，以及海外业务的拓展，我们需要支持微信支付，需要支持支付宝支付，甚至未来我们可能会支持区块链支付。类似这样的话，我们希望能够有一个良好的演进式的设计，能够很有弹性的支撑这样的业务扩展，而不是说，当我需要做一个区块链服务的时候，我需要把整个系统推倒重来。

点融的支付服务到现在大概进行了一年左右的时间，从一开始我们在设计这个系统的时候，其实我们就遵循了一定的原则。网上有个参考：12要素应用。我觉得比较重要的几点是这几点。首先我们的支付服务是一个无状态的，无状态的好处在于说，当我们有用户峰值或者有更高要求的访问量或者使用量的时候，我们可以通过简单的加机器来实现动态扩容。

自包含是指我们摒弃了传统的war包部署，也就是Tomcat容器部署，应用本身就是一个jar包，然后通过脚本化的启动，对外界就是一个黑盒子，你只关心的是数据交互。这样的好处是说，运维不再关心说我需要Tomcat还是什么，我只需要说，我知道这个应用自身启动需要哪些东西即可。环境不感知，具体是指我们把所有环境相关的配置统统挪到了一个配置中心，也就是说同一份代码可以无缝跑在开发环境、测试环境、生产环境。这样做的好处是说，我们可以做到完备的CICD过程，也就是当任何一步，比如开发完一个功能，我们打包测试，通过以后我们会自动promote到demo环境，

交给QA测试。QA测试完了以后我们会Promote交给UAT测试。整个UAT完备以后，我们可以自动化部署到生产。本身这个应用是从一开始的环境打包出来的那个，我们可以通过一个UUID或者Hash的方式来确保整个部署环节，镜像没有被破坏。

第二点就是说，我们在设计整个系统的时候，我们想说我们基于DDD，也就是邻域模型驱动来设计整个系统内部模块。支付服务作为一个领域来说，存在于点融的大系统之中。其实支付里面又有内生的子领域，包括订单，包括用户，包括支付渠道。那我们不希望用传统的一层一层三级架构传递下去，我们希望作为一个模块，比如说订单模块，作为一个商户模块，作为一个支付渠道模块，存在于自身的子模块。这样做的好处是未来系统的扩展。

最后做这个系统的时候，我们用了一些，也不算新的东西，应该是比较不常见的东西，我们用了CQRS和Event Sourcing，CQRS简单来说就是读选分离，这样做的好处在于，通常情况下你会发现你的数据大部分情况下会是写一次读多次，而且读的多样性会远远超过你的预期。那这样，我们可以有针对性的针对写端，做性能优化。我们也可以针对性的对读端做性能优化。比如应用支付系统里面的时候，因为整个系统是基于Event设计的，Event本身没有那么明显的数据格式，也就是我们不需要用传统的关系型数据库来保存Event事件。这里用了MongoDB存储一个一个事件流。最终这些事件流会异步或者同步到一个读DB，也就是MySQL。这个时候我们会根据业务的需要，去做一定量的数据重组，以方便后续的数据展示。

甚至我们都可以不需要View的存在，因为本身数据从重组的过程中就可以通过数据冗余实现View的功能。之所以用Event Sourcing，是来自于CQRS的衍生化的好处。对于支付系统来说，任何涉及到的钱的东西我们都希望能够很好的追踪，可以被审计。Event Sourcing的好处就是说，任何一步订单操作，比如从你下单到执行，到结算到清分，任何一步动作都会作为事件存储。我们可以很清楚的知道，整个订单在系统中是如何流转的，它发生了哪些变化产生了哪些效果。另外Event

Sourcing带来另外一个好处就是说，当我们生产发生问题的时候，其实我们是可以把生产的事件拿到本地或者测试环境，直接就可以找到问题的症结在何处。

在这个过程中我们有些经验教训，第一点，微架构真的很重要。大家都知道架构师是一个很诱人的title，但是你做应用的时候，其实你是这个应用的架构师，某一个很小的应用你也需要一个很细微的调整，以便这个应用能够自适应发展。

大家知道一个公司的系统架构，一定程度上决定了公司的业务发展。但是应用系统的微架构，某种程度上决定了系统的命运。如果它不能适应业务的需求，很快就会被另外一个新的应用给替代。第二点，系统基于事件驱动去开发，尽可能异步化处理。

基于事件驱动的开发的好处在哪？大家用过MQ的人都知道，异步化处理，对于系统的横向扩展是有相当大的好处的。我们在这里实际上做了一个相当于系统内嵌的消息处理机制，也就是说订单事件触发时，会被扔到一个消息中心，而这个消息会被多个感兴趣的监听者所监听。一开始我们可能只有支付，我们希望用户能够充值或者提现。后续我们希望监测一些用户的异常行为，比如说异常提现或者充值异常。这时候我们不需要在原有的支付逻辑上加代码，我们只要监听这个支付事件即可。同样我们也需要在支付成功或者支付失败的时候，希望能够第一时间通知到用户。传统的做法，比如在订单更新状态的时候插入另外一行逻辑处理代码。而基于事件的好处就是说，我们可以类似插件式的处理，比如把插件中心这个东西直接嵌入进来，它只需要监听它干兴趣的事件，做对应的处理即可。后续我们可能会有更多的组件模块，同样这些所有的处理都可以并行处理，而不需要传统的，一步一步的串行处理模式。

什么情况下我们会用到同步操作，什么情况下我们会用到异步操作？拿点融的支付业务场景来做简单的举例即可明白，充值跟投资，我们不希望用户被中断，所以我们需要把用户停留在操作界面上。所以整个用户参与度高的情况下，我们会希望说，所有的请求尽量同步化处理，以便用户得到及时反馈。相反，在提现这个场景下，由于支付渠道和银行的限制，其实到账时间往往会超过一小时，甚至一天。这种情况下是一个理想的可以用于异步操作的场景，事实上也是如此。对于点融支付来说，出去充值这一环节，其他的所有事件我们都是尽量异步化处理。

然后我要说一下，其实很多人在一开始写代码的时候会想说，这个类似过度优化。我们一开始想说要尽可能的把代码写得足够灵活，以便适应系统的需要或者业务的需要。但是实际上，其实就好像跳舞，如果你没有一定的节奏，或者没有一定的节拍，其实你很有可能在尬舞，如果你良好的节拍和良好的动作，你可以创造出更优美的舞姿。

同样，我们一开始做支付策略的时候陷入了一个误区，我们会觉得支付策略这些东西是不是可以由产品经理或者风控部门自己定义或自己设计规则？于是我们弄了一个相当于脚本引擎的东西，也就是说我们把一些重工作统统交给了业务人员，我们一开始想象的是说，我们希望减少开发人员的工作，因为我只要执行你的脚本即可，而且业务人员也可以运行脚本，去验证你想要的结果。真正用起来发现一个问题，其实在整个的过程中，整个支付策略的设计过程中，还有实现过程中，我们会发现大部分时间还是开发人员在参与。业务人员不懂脚本怎么写，我们也无法在短时间内提供完备的DSL，以便业务人员轻松书写业务规则。于是我们推倒重来，这样我们就给你一定的节拍和一定的舞步，于是我们设计了新的支付策略，也就是我们给了你一个条条框框，我们希望这个条条框框里，你可以自定义你想要的功能。

这一点挺好玩的，就像刚刚说的支付策略，我们其实是在自己造轮子，而且说起来不好意思，我们应该造了两次。第一次不是太成功，第二次正在验证。但是我觉得，很重要的一点是说，我们不需要反复的自己去造轮子或者任何功能我们都自己去造轮子。正如这个图上画的一样，我觉得可能我们一开始造的那个支付策略的轮子，要么是个三角形要么是个方形。实际上大家都知道，要跑起来的轮子一定得是圆形。所以在现在这种开放社区或者大厂商，各种开源框架比较完善的情况下，我觉得其实开源是一个更好的选择，选择现有开源框架，往往对你来说是更方便更快捷的事情。

可视化数据，监控所有能监控的数据。这一点非常重要，对于你的业务改变，还有对于你的系统的监测，以及系统后续的演化，是非常重要非常关键的。

比如说，我们这里是点融支付业务的核心数据的实时监控，这里我们用到了很多开源框架。从CQRS和Event Sourcing，我们用到了Axon框架，用了CQRS和Event Sourcing之后，我们不需要业务数据埋点，我们不需要重新梳理我们的关键路径，因为每一个事件都是关键路径。我们需要做的只是集中一个点收集这些事件，我们只需要短短几行代码去监听这个消息，然后通过Kafka传递给Logstash，简单处理后交给Elasticsearch，最终用Kibana做一个非常漂亮的图形化展示。所有这些工作大概只需要花费一到两周的时间，这也是为什么我会觉得说，我们不需要重复造轮子，我们要花更多的心思在微架构上调整。

当然，业务数据很重要，性能数据也同样重要。对我们来说，支付很大程度上是跟银行和支付渠道打交道。我们需要监控说，API的调用次数，API的耗时长度。我们也需要说，究竟是我们的问题还是三方支付公司的问题。大家可以看到，图看起来蛮相似的，最上面的是我们服务器的响应的时长，最下面是三方支付公司，我们在请求发往三方支付公司所耗的时长。大家可以发现，我们大部分时间都耗在了三方支付公司，要么是网络要么是支付公司的处理能力。

这样，我们一开始所有的请求处理其实都是同步的，这个时候我们会发现问题，比如说代扣、提现、代付，不需要用户参与的情况下，我们会发现我们的处理能力会急剧的下降，当达到某一个用户的峰值的时候，我们会发现大量的排队大量的超时。这时候我们需要做一个内部的排队处理，我们通过上述的数据分析可以清楚的得到说，原来是三方支付公司或者是网络问题，导致了我们系统性能的下降。

这里我们用到的开源框架Dropwizard会收集所有的系统，最终展示的效果同样也是一个开源框架，我们用了Grafana做一个数据展示。

说完了系统，本身Java程序少不了的必须要监控：JVM，大家可以看到曲线还是比较夸张的，因为我们的用户方式或者说我们的支付方式会比较有规律，所以我们会发现有一个很有规律的变化。上面是JVM的使用情况，下面有GC，还有thread。很多开源框架有一个问题，就是说它在用thread比较挥霍无度，我们通过这个来观察是不是有异常。这个也是通过Dropwizard能够简单实现的。

自动化一切可以自动化的。这里有一些风险，一会儿会提到。简单来说，你需要自动化部署，自动化部署的前提是你需要自动化测试。自动化测试的前提是什么？你需要有一个稳定的测试环境。但是三方支付公司的测试环境通常有各种各样的问题，甚至有些三方支付公司根本不提供测试环境。这时候怎么办？我们自己写。我们在跟三方支付公司测试联调完毕以后，系统上线以后，我们会写对应的mocker，也就是我们把所有三方支付公司的请求处理都mock一遍。不过现阶段可能比较简陋，因为我们更多的是迫于业务的要求，我们更多是正向的测试，没有负向或者没有edge case，这样难免会有些问题。另外mocker会导致说，你不确定三方API是否有变更，这会带来一定的风险。我们更多依赖于支付公司不会轻易发生这样的变化。

未来我们要怎么做？我们用了一些开源框架，对于我们后续的变化会有极大的好处，比如说实时数据分析。当我们可以便捷的进行数据分析，这时候我们可以做适当的智能路由。比如说常见的场景是某一个银行把支付通路给关了，但是它并没有提前告知到支付渠道，支付渠道也不可能提前告知到我们。这个时候，我们传统来说会采用轮巡的方式，比如每五分钟或者每一分钟监测一下，这个时间段这个支付渠道大概有多少笔失败的订单、失败的原因是什么。这时候会有一个问题是说，你一定会有一个时间段的误差，也就是它未必是真实的表象。这时候如果我们有实时数据，如果我们能够做实时分析，那这个会形成一个良好的生态链路，也就是说当一个渠道真正有问题的时候，我们可以拿到第一手资料，进行第一手分析，然后做一个动态的智能切换。

之前也说了，我们设计整个支付服务的时候就考虑到，支付里面每一个子领域都是可以独立拆分的。那就意味着说，未来可能支付服务本身就会变成另外一个微服务化的系统。这样的话，我们会想说，我们到底要固守Java世界还是要采取语言不可知论？最近突然比较火热的Kotlin。拿支付策略来举例，我们现在还是用Java实现，但是或多或少都有些别扭，因为Java本身并不是一个function program language。它本身也不支持友善的DSL。反过来我们知道Ruby在DSL独树一帜，而Go更适合在ATI网关这种应用场景下发挥它的作用，为什么要固守Java世界？对我们来说我们会尝试更多的不同的语言，就好像我们现在的支付管理后台，我们并不是用传统的JSP来做的，我们是用Ruby来写的。未来我们要做的API网关可能会采取Go。上面说的实时数据分析，Java也不擅长这个东西，我们可能会用Scala可能用Ruby甚至可能用更新的Kotlin。因为这些静态化的函数式语言，它的应用场景更适合。

另外是说，最近比较火热的Serverless和Lambda架构，它的好处在哪？其实我们不需要一个stand by的server，比如说我们在考虑计费，我们在做策略试算的时候，其实我们不需要一个24小时stand by的server在那里等着某一个时间段突然爆发的请求，我们只需要说，在需要处理的时候快速的去响应快速的去处理。像Amazon提供的Lambda架构是一个理想的状态，就像刚刚说的计费，计费不需要那么实时，我甚至可以跑批处理。

如果独立开一个计费模块的话，你会发现你要一个7×24小时stand by的东西，但是它只是偶尔要处理一些东西。其实我只需要在处理的时候去获取我需要的计算机资源即可。

点融的支付服务运行到现在，我们在考虑做一些技术输出的方向，也就是说我们会跟银行合作，会跟其他公司合作，想要推我们的支付服务。这另外会面临一个比较大的问题，就是如何信任。如果解决不了这个问题，我们面临的问题是我们变成了一个传统的软件开发公司，我们技术外包，把系统开发好，部署交给银行或者交给其他公司去使用。这个想来跟大家知道的现有的科技趋势是背道而驰的。我们会想说有没有更好的办法去解决？我们其实更重要的是说，我们需要保证的，最核心的一点是说，我们希望保证交易的双方是可信的，交易双方的数据是不被篡改的。这个时候我们也会有区块链，相信大家知道，应该是一个理想的选择。但是区块链有一个问题是，交易双方的数据会被整个链路所有人共享。也就是说我跟某个特定银行的交易会被整个链路其他银行或者其他公司共享。虽然数据加密可以解决这个问题，但是我本身就不希望数据流向其他链路。

最近出来的Corda，提供的是分布式总帐的方案。它和区块链的区别在哪？交易数据仅存在于交易双方。当然这个还是比较新的，因为它本身的语言事件也是用Kotlin，还是比较巧的，有待考证。但是本身来，对于支付服务的演化，未来很重要的一点应该是区块链，我们希望服务可以被大家使用，我们也希望我们的服务是可信的，而且数据不被篡改。通过技术，我们应该是可以实现这样的目标。

下面是一些参考资料，第一个是12要素app，告诉你怎样做好12要素的应用，以便适应未来的需要。第二个是Martin Fowler的，第三个也是Martin马大神的。

今天就介绍到这里，谢谢。

现场互动

姚晨回答现场观众提问

提问：我本身也是做支付，我想问一下，你们微化之后，很多服务，相互的调用是怎么样的？管理怎么处理的？还有异步性，很核心的订单如果异步了，可能是为了提高性能，但是加入我们出现了宕机，这样的一些容灾措施应该是如何处理？

姚晨：首先第一个问题，当你微服务化的时候，最重要的是注册中心，还有一个API网关的概念。也就是说，你希望对外提供的是一个服务，但是内部可能是不同的，RPC也好，用哪个实现，这个更多取决于公司内部，用什么不太重要。但是一旦有了很好的配置中心和服务，相当大的程度上能够解决第一个问题。第二个问题是消息发生了遗失怎么办，CQRS的好处是遗失了你可以重复播放，异步不是所有的东西都异步，只是用户发起了之后你可以异步执行。具体订单的状态的流转，其实你可以做同步化，但是每一步同步化，产生的事件没有被及时处理，可以再发送。好处在这里。本身来说，CQRS有一定的学习成本，所以不见得所有的系统都可以跑，尤其是已有系统，改造起来会比较麻烦。

提问：我看到你有引用马大神一些东西，关于这个有两个问题，我看到你们已经应用了他的一些理论，那第一个是说，比如你用到的Event Sourcing，在发生一些变化的时候是变成一个一个Event，这样可以在一个完全开始的状态去把一件事情重复出来。我看到一个引发的问题是说，怎么样结构化的存储这些Event，就是这件事情变化的时候，你是用Event存储起来的，怎么样结构化的存储到一些结构化数据库里面，以你怎么样，比如说怎么样去查询，拿到像结构化数据库里面很容易拿到的一些结果？第二个有用到Serverless和Lambda的架构，需要的时候可以申请一些资源做一些事情，那怎么保证速度和响应时间？

姚晨：先说第二个问题，其实还没有开始用，只是预计未来想要这样发展。但是如果考虑到时间来说，其实你起一个Lambda的function program的话要快得多，你可以保证它在毫秒级。运行过Java层知道，那是秒级才会起的。这个还需要后续的验证。当然我觉得这是一个未来的方向。

关于第一点，我们用到了CQRS不单单是Event Sourcing，因为CQRS的好处是你可以很方便的去用Event Sourcing。如果是纯粹的Event Sourcing，不知道怎么去处理非结构化数据，让它结构化。CQRS的好处是说，你可以把非结构化的数据在读端结构化，也就是说我们会有两个DB，对于点融支付有两块DB，一个是MongoDB存非结构化数据，然后有一个MySQL，用来存最终状态的存放，这时候可以实现结构化处理。

其实任何，当然这里有一点很重要是说，任何事件的触发一定是基于Event产生的那个最终的内存状态。也就是说，其实你存储到非结构化数据里面，实际上是一个一个事件，当这个事件不再被使用的时候，本身这个订单已经被丢失了。当你下一次来操作来查询，查询并不会这样，操作的时候，后台的框架是把所有的Event拿出来，逐个播放，最终可以得到一个真正有效的状态。所以其实，老实讲还是得益于框架的使用，所以我们可以很好的做一个读跟写的数据的分离。

提问：所以你们的Mongo，处理写，MySQL在读，所以这两个中间有一个转化?

姚晨：对，这个转化也得益于，我们只要去监听对Event的状态。

提问：我有两个小问题，我本身是做风控的，想问一下关于风控的问题。第一个问题是说，你们的风控是，比如说像策略平台，是自己实现的还是说引入了外部的技术和系统开发的？第二个问题是说，你们目前风控的拦截，大概有多少？有没有出现因为风控的原因导致的资损这种问题?

姚晨：其实我觉得我们说的风控不是同一个维度的风控，我说的更多是支付本身业务的风控，而不是更外层那种。

提问：就是进入支付之后的风控?

姚晨：对，本身支付业务的风控我们是自己开发的，因为风控老实来讲，还是比较专有的领域。你还是需要自定义的DSL，要么是别人帮你实现的DSL，要么是你自己实现的DSL，这样你的业务人员才能去改那样的规则书写那样的规则，而我们一开始选择的纯脚本化真的不是很好的选择。可能回答不了您的问题。

提问：我有两个小问题，一个是在支付服务里面有没有遇到什么分布式事务的问题？如果有是怎么解决？还有一个是我看到你这边图表，包括业务数据和性能数据，我理解是这两个权限管理都很弱的，你们在权限管理方面是怎么做的?

姚晨：第一个问题，我个人是强烈反对分布式事务，所以在我们系统里面是没有分布式事务存在。我的理解，理论上是可以，但是本身这个框架也支持一个Saga，一个大的事务，但是并不像事务那么明显。之所以用分布式的初衷一定是想要rollback，在出现问题的时候。其实如果有良好的补偿机制，是可以做到程序化的回滚。

提问：就是所有的服务都是带有补偿机制?

姚晨：对。第二个问题是，实际上没错，这两个权限都弱爆了。在Kibana那一块我们用到了开源的一个框架，做了一些基本的权限控制。理论上来说还是可以满足一定的需求。另外我们也咨询过开发公司Elastic，他们提供的解决方案会相对来说成本比较高，所以我们会用SearchGuard我建议你可以尝试一下。

提问：我在你们讲座的过程中，听说点融网作为第三方的支付公司，接的渠道有第三方公司，第三方公司在后面也是接银行、银联、银企，你们为什么不直接接银行？你们接入第三方支付公司，耗时会有提升，提供支付服务，为什么别人找你们做第三方支付，你们又接第三方支付公司？这样别人就可以直接找别人，为什么要找你们？你们作为第三方支付公司，接第三方支付公司，成本也会大很多。

姚晨：其实这个有各种方面的原因，因为其实不是我们不想跟银行玩，是银行不想跟我们玩。因为金融行业实际上是有监管，有准入门槛。我们没有牌照或者我们现在互金行业在进行监管。实际上合规还在进行中，其实会有各种各样的资质约束。所以我们只能去跟支付公司打交道。但是本身来说，我觉得支付公司也好，银行也好，其实对于支付系统来说都是一个通道，也就是说我们提供给支付公司或者其他公司来用，我并不是把我的支付能力给到你，因为我没有支付牌照，我不能帮你做二清。那意味着说，我其实是我是帮你提供技术解决方案。支付或多或少都会面临同样的问题，你要做渠道选择，你要做路由策略，你要做监控，你要做审，你要做风控。这些东西都是公有的业务，我们提供的还是一个综合的解决方案，而不是告诉你说我有这个支付通路，你来用我的支付通路。因为这个理论上来说是不合规。

提问：我有两个问题，一个是关于CQRS，你们写入是用MongoDB，后面的消息队列机制，你们有用到什么框架？还有一个是关于区块链的，能不能先回答我第一个问题？这两个问题不太一样。

姚晨：其实是一个内嵌的，我们通过框架内嵌了一个消息队列，当然它支持分布式的消息队列处理。

提问：事件监听有用到吗？

姚晨：事件框架本身就是靠框架支撑的。

提问：会为每一个页面用一种方式去做吗？

姚晨：这不会，但是我们会根据业务需求做一定的优化。其实你的订单，比如说某一个支付订单，其实你可能会需要很多数据，我们传统做法是会有一个view的存在，把数据整合起来。对于我们来说，不需要View的存在，甚至Report DB也不需要，我们可以通过Event的方式做到很多以前需要其他东西辅助才能完成的东西。

提问：主要是用在MySQL？

姚晨：对。

提问：还有一个问题是关于区块链，你们三月份初你们老大在美国发布了跟富士康的那个，区块链框架你们本身是用的Corda？

姚晨： 区块链老实来讲更多是底层服务，支付是基础服务，是构建于底层服务之上的，我们有专门的组负责整个区块链的开发和应用。对于我来说，我的诉求很简单，就需要一个分布式的帐簿就好了。

姚晨在中国架构师大会现场

提问：Corda刚出来，你们团队用了18个月开发。

姚晨：应该还没有开始，就像我说的，我们要微服务化，要解决的最核心的就是交易双方的信任问题。区块链和Corda应该是可以解决类似的问题。目前还在调研阶段。

提问：不是有产品已经上线了？

姚晨：那个更多是信息，那是供应链金融，其实我们想知道的是说，下游的厂商的资质，更多是信息在区块链的保障。我们这边未来要做的更多是支付。

提问：但是那套系统不是也有用到区块链吗？

姚晨：对，那个项目并不是我参与的，所以可能没法很好的回答你。我只能说，我们需要类似的东西去解决这些问题。

提问：我想问一下，因为您这次分享的主题提到了DDD这一块，以前在其他的会上我见到这个很少，这次非常感谢。有一个问题，你提到CQRS架构和Event Sourcing，我觉得还是比较困难的，它会导致我们整个开发思维发生变化，由原来的连续的逻辑思维方式变成一个离散的方式。我不知道像你们这种，在你们公司应用下来，有没有受到阻力？比如人员的要求提高什么的。

姚晨：我觉得还是蛮挑战的，就我自身来说，在整个应用过程中都发现，逻辑思维会有一个很大的转变。当然，我觉得当你接受或者说感受到它的好处的时候，其实也未尝不可。因为你想，其实都是惯性思维。这对于开发人员，会要求更高。因为几乎是完全摒弃了传统的做法。

提问：没错，这有一定的阻力。我们在公司里面想一个人推动这个是非常困难的。以什么方式可以推动这个事情?

姚晨：我在做这个的时候只有我一个人，我最开始负责把它大致的框架弄好，把规则都制定好，后来者紧跟随学习就可以。最开始难点在于整个思维方式有蛮大的变化，但是也是对我来说是学习和适应的过程。

本文作者：姚晨（点融黑帮），混迹于 IT 行业多年的码农，热爱编程，从流水线上的作业系统到衍生品市场的风控系统再到大宗商品市场的交易系统皆有所贡献；善专研喜折腾，典型的不折腾会死星人。设计和构建点融支付系统，致力于推进最佳实践在点融系统中的应用。
