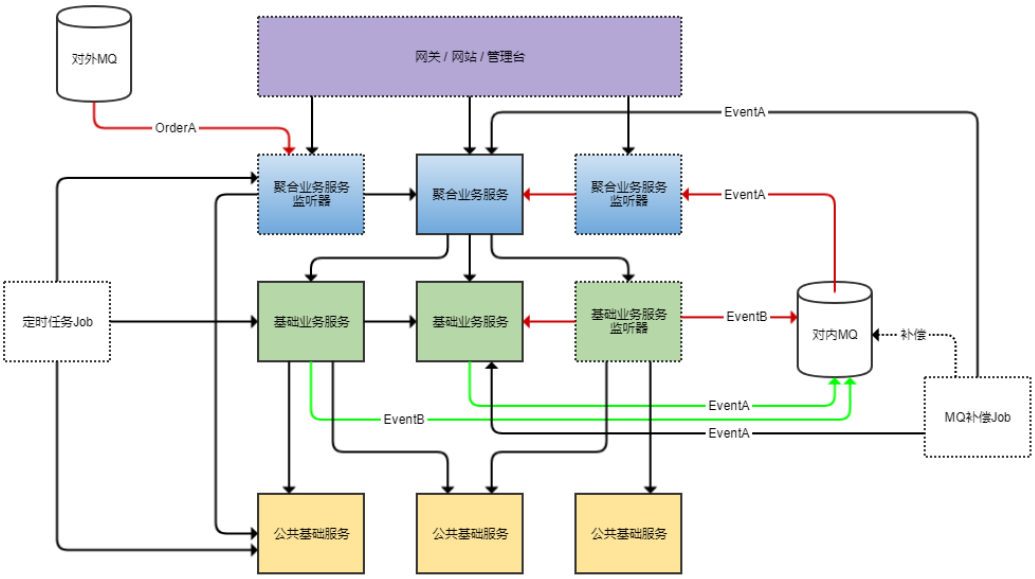
朱晔的互联网架构实践心得 S1E2：屡试不爽的架构三马车

【下载本文 PDF 进行阅读】

这里所说的三架马车是指**微服务**、**消息队列**和**定时任务**。如下图所示，这里是一个三驾马车共同驱动的一个立体的互联网项目的架构。不管项目是大是小，这个架构模板的形态一旦定型了之后就不太会变，区别只是我们有更多的服务有更复杂的调用，更复杂的消息流转，更多的Job，整个架构整体是可扩展的，而且不会变形，这个架构可以在很长的一段时间内无需有大的调整。



图上画了虚线框的都代表这个模块或项目是不包含太多业务逻辑的，纯粹是一层皮（会调用服务但是不会触碰数据库）。黑色线的箭头代表依赖关系，绿色和红色箭头分别是 MQ 的发送和订阅消息流的方向。具体在后文都会进一步详细说明。

微服务

微服务并不是一个很新的概念，在 10 年前的时候我就开始实践这个架构风格，在四个公司的项目中全面实现了微服务，越来越坚信这是非常适合互联网项目的一个架构风格。不是说我们的服务一定要跨物理机器进行远程调用，而是我们通过进行有意的设计让我们的业务在一开始的时候就按照领域进行分割，这能让我们对业务有更充分的理解，能让我们在之后的迭代中轻易在不同的业务模块上进行耕耘，能让我们的项目开发**越来越轻松**，轻松来源于几个方面：

1. 如果我们能进行微服务化，那么我们一定事先经过比较完善的产品需求讨论和领域划分，每一个服务精心设计自己领域内的表结构，这是一个很重要的设计过程，也决定了整个技术架构和产品架构是匹配的，对于 All-In-One 的架构往往会省略这一过程，需求到哪里代码写到哪里。
2. 我们对服务的划分和指责的定位如果是清晰的，对于新的需求，我们就能知道需要在哪里改怎么样的代码，没有复制粘贴的存在少了很多坑。
3. 我们大多数的业务逻辑已经开发完毕，直接重用即可，我们的新业务只是现有逻辑的聚合。在 PRD 评审后，开发得到的结论是只需要组合分别调用 ABC 三个服务的 XYZ 方法，然后在 C 服务中修改一下 Z 方法增加一个分支逻辑，就可以构建起新的逻辑，这种爽快的感觉难以想象。
4. 在性能存在明显瓶颈的时候，我们可以针对性地对某些服务增加更多机器进行扩容，而且因为服务的划分，我们更清楚系统的瓶颈所在，从 10000 行代码定位到一行性能存在问题的代码是比较困难的，但是如果这 10000 行代码已经是由 10 个服务构成的，那么先定位到某个服务存在性能问题然后再针对这个服务进行分析一下子降低了定位问题的复杂度。
5. 如果业务有比较大的变动需要下线，那么我们可以肯定的是底层的公共服务是不会淘汰的，下线对应业务的聚合业务服务停掉流量入口，然后下线相关涉及到的基础服务进行部分接口即可。如果拥有完善的服务治理平台，整个操作甚至无需改动代码。

这里也要求我们做到几个方面的原则：

1. 服务的粒度划分需要把控好。我的习惯是先按照领域来分不会错，随着项目的进展慢慢进行更细粒度的拆分。比如对于互联网金融 P2P 业务，一开始可以分为：
    a. 三方合作服务 PartnerInvestService：对接合作的三方理财平台的流量
    b. 普通投资服务 NormalInvestService：最普通形态的资产的主流程
    c. 预约投资产品服务 ReserveInvestService：需要预约投资的资产的主流程
    d. 周期性计划产品服务 AutoInvestService：会定期自动复投的理财产品主流程
    e. 投资人交易服务 TradeService：专门负责处理投资人的交易行为，比如投资
    f. 借款人交易服务 LoanService：专门负责处理借款人的交易行为，比如还款
    g. 用户服务 UserService：处理用户的注册登录等

h. 资产服务 ProjectService：处理资产和标的相关

i. 账户账务服务 AccountService：处理用户的账户各个子账户和账务记录

j. 营销活动服务 ActivityService：处理各种活动、用户的积分体系

k. 会员体系服务 VipService：处理用户的会员成长体系

l. 银行存管服务 BankService：专门用于对接银行存管系统

m. 电子签章服务 DigSignService：专门用于对接三方数字签章系统

n. 消息推送服务 MessageService：专门用于对接三方短信通道和推送 SDK

2. 服务一定是立体的，不是在一个层次上的，如上图，我们的服务有三个层次：

a. 聚合业务服务：高层次的串起来整个流程的具有完整业务形态的业务服务。和基础业务服务不同的是，这里是在完整描述一方面的业务，这个业务往往是由各种基础业务拼装组合起来的。和不同外部合作方的不同合作形式，给用户提供的产品的不同服务形态，都决定了聚合业务服务会有业务流程上的差异化，如果把此类服务下放到基础业务服务中，那么基础业务服务会有各种 if-else 逻辑（根据产品类型、用户类型进行各种 if-else），随着业务的合作不合作，需求变动，基础业务服务会腐化得很厉害，为了避免这个情况，我们把变动的多的聚合业务逻辑放到独立的业务服务中。一般而言，聚合业务服务因为代表了独立的业务流程，它们之间是不会进行相互调用的，但是它们一定会调用大量的各类基础业务服务。在第一点里说的标有蓝色字体的 a~d 这些服务都是此类服务。这个层次的服务的业务逻辑更多是在表达业务流程的复杂性和差异性，不会涉及到具体怎么处理账户信息、账务信息、用户信息，不会涉及到怎么处理具体的投资人和借款人的交易。比如对于预约这类业务形态，它关注的是先要预约资产，然后再由系统进行自动投资，底层完全依赖于投资人交易服务来做整个交易的过程。

b. 基础业务服务：某一个领域业务相关的服务。此类服务之间是允许相互调用的，比如投资人交易服务和借款人交易服务免不了需要和用户服务、资产服务、账户账务服务进行通讯做相关的用户信息查询、标的信息查询、记账等业务操作。之所以投资人交易服务和借款人交易服务定位为基础业务服务是因为，它们处理的是还是某一个具体方面的业务，并不是全流程，在这个抽象层次上，业务不是那么容易变动的，对于复杂的各种业务形态（比如预约交易、自动复投交易、等额本息交易）会在这些服务之上形成聚合业务服务。在第一

点里说的标有绿色字体的 e~k 这些服务都是此类服务。在这个层次的服务虽然拥有大量的业务逻辑，但是其实已经享受到了很大层度的公共基础服务的重用了，而且和自己业务耦合较弱的额外逻辑往往没有在本服务中堆积，由更多专职的基础业务服务来承担了这部分逻辑。

c. 公共基础服务：负责某一个方面的基础业务（没有什么领域业务逻辑在里面），可以是自治的处理某一个方面的基础业务，也可以和外部通讯实现某一个方面的功能，服务之间是不会相互调用的，但是会被聚合业务服务和基础业务服务调用。在第一点里说的标有橙色字体的 l~n 这些服务都是此类服务。如果以后和外部的合作有变动，因为我们已经定义了对外的服务契约，可以轻易替换这个服务来更换合作的第三方，系统其余的地方几乎不需要修改。所有的三方对接都建议独立出公共基础服务，如果同一个业务对接多个三方渠道，比如推送对接了极光和个推，甚至公共基础服务还可以由一个抽象聚合的推送服务，下面再路由到具体的极光推送和个推推送服务。

希望在这里把这个事情说清楚了，怎么来划分服务怎么划分三个层次的服务是一个很有意思很有必要的事情，在服务划分之后最好有一个明确的文档来描述每一个服务的职责，这样我们在无需阅读 API 的情况下可以大概定位到业务所在的服务，整个复杂的系统就变得很直白了。

3. 每一个服务对接的底层数据表是独立的没有交叉关联的，也就是数据结构是不直接对外的，需要使用其他服务的数据一定通过访问接口进行。好处也就是面向对象设计中封装的好处：

   a. 可以很方便地重构底层的数据结构甚至是数据源，只要接口不变，外部不会感知到。

   b. 性能有问题的情况下需要加缓存、分表、拆库、归档是比较方便的事情，毕竟数据源没有外部依赖。

说白了就是我的数据我做主，我想怎么搞外面管不着，在重构或是做一些高层次技术架构（比如异地多活）的时候，没有底层数据被依赖，这太重要了。当然，坏处或是麻烦的地方就是跨服务的调用使得数据操作无法在一个数据库事务中完成，这并不是什么大问题，一是因为我们这种拆分方式并不会让粒度太细，大部分的业务逻辑是在一个业务服务里完成的，二是后面会提到跨服务的调用不管是通过 MQ 进行的还是直接调用进行的，都会有补偿来实现最终一致性。

4. 考虑到跨机器跨进程调用服务稳定性方面的显著差异。在方法内部进行方法调用，我们需要考虑调用出现异常的情况，但是几乎不需要考虑超时的情况，几乎不需要考虑请求丢失的情况，几乎不需要考虑重复调用的情况，对于远程服务调用，这些点都需要去重点考虑，否则系统整体就是基本可用，测试环境不出问题，但是到了线上问题百出的状态。这就要求对于每一个服务的提供和调用多问几个上面的问题，细细考虑到因为网络问题方法没有执行多次执行或部分执行的情况：

    a. 我们在对外提供服务的时候，不但要告知用户服务提供的业务能力，还要告知用户服务的特性，比如是否是幂等的（对于订单类型的操作服务，相同的订单相同的操作强烈建议是幂等的，这样调用方可以放心进行重试或补偿）；是否需要外部进行补偿（在这里你可能说为什么需要外部进行补偿，服务就不能自己补偿吗，对于内部的子逻辑服务当然可以自己补偿，但是有的时候因为网络原因请求就没有到服务端，服务端一无所知这个调用当然无从去补偿）；是否有频控的限制；是否有权限的限制；降级后的处理方式等等。

    b. 反过来，我们调用其它服务也需要多问几句目标服务的特性，针对性进行设计相应的补偿逻辑、一致性处理逻辑和降级逻辑。我们必须考虑到有些时候并不是服务端的问题，而是请求根本没有到达服务端。

    c. 服务本身往往也会有复杂的逻辑，作为客户端的身份调用大量外部的服务，所以服务端和客户端的角色不是固定不变的，当我们的服务内部有许多客户端来调用服务端的时候，对于每一个子逻辑我们都需要仔细考虑每一个环节。否正会出现的情况就是，这个服务是部分逻辑幂等的或是部分逻辑是具备最终一致性的。

如果你说，这么多服务，我在实现的时候很难考虑到这些点，我完全不去考虑分布式事务、幂等性、补偿（毫不夸张地说，有的时候我们花了 20%的时间实现了业务逻辑，然后花 80%的时间在实现这些可靠性方面的外围逻辑），行不行？也不是不可以，那么业务在线上跑的时候一定会是千疮百孔的，如果整个业务的处理对可靠性方面的要求不高或是业务不面向用户不会受到投诉的话，这部分业务的是可以暂时不考虑这些点，但是诸如订单业务这种核心的不允许有不一致性的业务还是需要全面考虑这些点的。

5. 考虑到跨机器跨进程调用服务数据传输方面的显著差异。对于本地的方法调用，如果参数和返回值传的是对象，那么对于大部分的语言来说，传的是指针（或指针的拷贝），指针指向的是堆中分配的对象，对象在数据传输上的成本几乎忽略不计，也没有序列化和反序列化的开销。对于跨进程的服务调用，这个成本往往不能忽略不计。如果我们需要返回很多数据，往往接口的定义需要进行特殊的改造：

a. 通过使用分页的形式，一次返回固定的少量数据，客户端按需拉取更多数据。

b. 可以在参数中传类似于 EnumSet 的数据结构，让客户端告知服务端我需要什么层次的数据，比如 GetUserInfo 接口可以提供给客户端 BasicInfo、VIPInfo、InvestData、RechargeData、WithdrawData，客户端可以按需从服务端拿 BasicInfo|VipInfo。

6. 这里还引申出方法粒度的问题，比如我们可以定义 GetUserInfo 通过传入不用的参数来返回不同的数据组合，也可以分别定义 GetUserBasicInfo、GetUserVIPInfo、GetUserInvestData 等等细粒度的接口，接口的粒度定义取决于使用者会怎么来使用数据，更趋向于一次使用单种类型数据还是复合类型的数据等等。

7. 然后我们需要考虑接口升级的问题，接口的改动最好是兼容之前的接口，如果接口需要淘汰下线，需要先确保调用方改造到了新接口，确保调用方流量为 0 观察一段时间后方能从代码下线老接口。一旦服务公开出去，要进行接口定义调整甚至下线往往就没有这么容易了，不是自己说了算了。所以对外 API 的设计需要慎重点。

8. 最后不得不说，在整个公司都搞起了微服务后，跨部门的一些服务调用在商定 API 的时候难免会有一些扯皮的现象发生，到底是我传给你呢还是你自己来拉，这个数据对我没用为什么要在我这里留一下呢？抛开非技术层面的事情不说，这些扯皮也是有一些技术手段来化解的：

   a. 明确服务职责，也就明确了服务应该感知到什么不应该感知到什么。

   b. 跨部门的服务交互的接口定义可以定的很轻，采用只有一个订单号的接口或 MQ 通知+数据回拉的策略（谁数据多谁提供数据接口，不用把数据一次性推给下游）。

   c. 数据提供方可以构建一套通用数据接口，这样可以满足多个部门的需求，无需做定制化的处理。甚至在接口上可以提供落地和不落地两种性质的透传。

你可能看到这里觉得很头晕，为什么微服务需要额外考虑这么多东西，实现的复杂度一下子上升了。我想说的是我们需要换一个角度来考虑这个事情：

1. 我们不需要在一开始的时候对所有逻辑都进行严密的考虑，先覆盖核心流程核心逻辑。因为跨服务成为了服务的提供方和使用方，相当于除了我自己，还有很多其它人会来关系我的服务能力，大家会提出各种问题，这对设计一个可靠的方法是有好处的。

2. 即使在不跨服务调用的时候我们把所有逻辑堆积在一起，也不意味着这些逻辑一定是事务性的，实现严密的，跨服务调用往往是一定程度放大了问题产生的可能性。

3. 我们还有服务框架呢，服务框架往往会在监控跟踪层次和运维系统结合在一起提供很多一体化的功能，这将封闭在内部的方法逻辑打散暴露出来，对于有一个完善的监控平台的微服务系统，在排查问题的时候你往往会感叹这是一个远程服务调用就好了。

4. 最大的红利还是之前说的，当我们以清晰的业务逻辑形成了一个立体化的服务体系之后，任何需求可以解剖为很少量的代码修改和一些组合的服务调用，而且你知道我这么做是不会有任何问题的，因为底层的服务 ABCDEFG 都是经过历史考验的，这种爽快感体验过一次就会大呼过瘾。

但是，如果服务粒度划分的不合理，层次划分的不合理，底层数据源有交叉，没考虑到网络调用失败，没考虑到数据量，接口定义不合理，版本升级过于鲁莽，整个系统会出各种各样的扩展问题性能问题和 Bug，这是很头痛的，这也就需要我们有一个完善的服务框架来帮助我们定位各种不合理，在之后说到中间件的文章中会再具体着重介绍服务治理这块。

消息队列

消息队列 MQ 的使用有下面几个好处，或者说我们往往处于这些目的来考虑引入 MQ：

1. 异步处理：类似于订单这样的流程一般可以定义出一个核心流程，这个流程用于处理核心订单的状态机，需要尽快同步落库完成，然后围绕订单会衍生出一系列和用户相关的库存相关的后续的业务处理，这些处理完全不需要卡在用户点击提交订单的那刹那进行处理。下单只是一个确认合法受理订单的过程，后续的很多事情都可以慢慢在几十个模块中进行流转，这个流转过程哪怕是消耗 5 分钟，用户也无需感受到。

2. 流量洪峰：互联网项目的一个特点是有的时候会做一些 toC 的促销，免不了有一些流量洪峰，如果我们引入了消息队列在模块之间作为缓冲，那么 backend 的服务可以以自己既有的舒服的频次来被动消耗数据，不会被强压的流量击垮。当然，做好监控是必不可少的，下面再细说一下监控。

3. 模块解耦：随着项目复杂度的上升，我们会有各种来源于项目内部和外部的事件（用户注册登陆、投资、提现事件等），这些重要事件可能不断有各种各样的模块（营销模块、活动模块）需要关心，核心业务系统去调用这些外部体系的模块，让整个系统在内部纠缠在一起显然是不合适的，这个时候通过 MQ 进行解耦，让各种各样的事件

在系统中进行松耦合流转，模块之间各司其职也相互没有感知，这是比较适合的做法。

4. 消息群发：有一些消息是会有多个接收者的，接收者的数量还是动态的（类似指责链的性质也是可能的），在这个时候如果上下游进行一对多的耦合就会更麻烦，对于这种情况就更适用使用 MQ 进行解耦了。上游只管发消息说现在发生了什么事情，下游不管有多少人关心这个消息，上游都是没有感知的。

这些需求互联网项目中基本都存在，所以消息队列的使用是非常重要的一个架构手段。在使用上有几个注意点：

1. 我更倾向于独立一个专门的 listener 项目（而不是合并在 server 中）来专门做消息的监听，然后这个模块其实没有过多的逻辑，只是在收到了具体的消息之后调用对应的 service 中的 API 进行消息处理。listener 是可以启动多份做一个负载均衡的（取决于具体使用的 MQ 产品），但是因为这里几乎没有什么压力，不是 100%必须。注意，不是所有的 service 都是需要有一个配到的 listener 项目的，大多数公共基础服务因为本身很独立不需要感知到外部的其它业务事件，所以往往是没有 listener 的，基础业务服务也有一些是类似的原因不需要有 listener。

2. 对于重要的 MQ 消息，应当配以相应的补偿线作为备份，在 MQ 集群一切正常作为补漏，在 MQ 集群瘫痪的时候作为后背。我在日千万订单的项目中使用过 RabbitMQ，虽然 QPS 在几百上千，远远低于 RabbitMQ 压测下来能抗住的数万 QPS，但是整体上有那么十万分之一的丢消息概率（我也用过阿里的 RocketMQ，但是因为单量较小目前没有观察到有类似的问题），这些丢掉的消息马上会由补偿线进行处理了。在极端的情况下，RabbitMQ 发生了整个集群宕机，A 服务发出的消息无法抵达 B 服务了，这个时候补偿 Job 开始工作，定期从 A 服务批量拉取消息提供给 B 服务，虽然消息处理是一批一批的，但是至少确保了消息可以正常处理。做好这套后备是非常重要的，因为我们无法确保中间件的可用性在 100%。

3. 补偿的实现是不带任何业务逻辑的，我们再梳理一下补偿这个事情。如果 A 服务是消息的提供者，B-listener 是消息监听器，听到消息后会调用 B-server 中具体的方法 handleXXMessage(XXMessage message)来执行业务逻辑，在 MQ 停止工作的时候，有一个 Job（可配置补偿时间以及每次拉取的量）来定期调用 A 服务提供的专有方法 getXXMessages(LocalDateTime from, LocalDateTime to, int batchSize)来拉取消息，然后还是（可以并发）调用 B-server 的那个 handleXXMessage 来处理消息。这个补偿的 Job 可以重用的可配置的，无需每次为每一个消息都手写一套，唯一需要多做的事情是 A 服务需要提供一个拉取消息的接口。那你可能会说，我 A 服务这里还需要维

护一套基于数据库的消息队列吗，这个不是自己搞一套基于被动拉的消息队列了吗？其实这里的消息往往只是一个转化工作，A 一定在数据库中有落地过去一段时间发生过变动的数据，只要把这些数据转化为 Message 对象提供出去即可。B-server 的 handleXXMessage 由于是幂等的，所以无所谓消息是否重复处理，这里只是在应急情况下进行无脑的过去一段时间的数据的依次处理。

4. 所有消息的处理端最好对相同的消息处理实现幂等，即使有一些 MQ 产品支持消息处理且只处理一次，靠自己做好幂等能让事情变得更简单。

5. 有一些场景下有延迟消息或延迟消息队列的需求，诸如 RabbitMQ、RocketMQ 都有不同的实现方式。

6. MQ 消息一般而言有两种，一种是（最好）只能被一个消费者进行消费并且只消费一次的，另一种是所有订阅者都可以来处理，不限制人数。不用的 MQ 中间件对于这两种形式都有不同的实现，有的时候使用消息类型来做，有的使用不同的交换机来做，有的是使用 group 的划分来做（不同的 group 可以重复消息相同的消息）。一般来说都是支持这两种实现的。在使用具体产品的时候务必研究相关的文档，做好实验确保这两种消息是以正确的方式在处理，以免发生妖怪问题。

7. 需要做好消息监控，最最重要的是监控消息是否有堆积，有的话需要及时增强下游处理能力（加机器，加线程），当然做的更好点可以以热点拓扑图绘制所有消息的流向流速一眼就可以看到目前哪些消息有压力。你可能会想既然消息都在 MQ 体系中不会丢失，消息有堆积处理慢一点其实也没什么问题。是的，消息可以有适当的堆积，但是不能大量堆积，如果 MQ 系统出现存储问题，大量堆积的消息有丢失也是比较麻烦的，而且有一些业务系统对于消息的处理是看时间的，过晚到达的消息是会认为业务违例进行忽略的。

8. 图上画了两个 MQ 集群，一套对内一套对外。原因是对内的 MQ 集群我们在权限上控制可以相对弱点，对外的集群必须明确每一个 Topic，而且 Topic 需要由固定的人来维护不能在集群上随意增删 Topic 造成混乱。对内对外的消息实现硬隔离对于性能也有好处，建议在生产环境把对内对外的 MQ 集群进行隔离划分。

定时任务

定时任务的需求有那么几类：

1. 如之前所说，跨服务调用，MQ 通知难免会有不可达的问题，我们需要有一定的机制进行补偿。
2. 有一些业务是基于任务表进行驱动的，有关任务表的设计下面会详细说明。
3. 有一些业务是定时定期来进行处理的，根本不需要实时进行处理（比如通知用户红包即将过期，和银行进行日终对账，给用户出账单等）。和 2 的区别在于，这里的任务的执行时间和频次是五花八门的，2 的话一般而言是固定频次的。

详细说明一下任务驱动是怎么一回事。其实在数据库中做一些任务表，以这些表驱动作为整个数据处理的核心体系，这套被动的运作方式是最最可靠的，比 MQ 驱动或服务驱动两种形态可靠多，天生必然是可负载均衡的+幂等处理+补偿到底的，任务表可以设计下面的字段：

- 自增 ID
- 任务类型：表明具体的任务类型，当然也可以不同的任务类型直接做多个任务表。
- 外部订单号：和外部业务逻辑的唯一单号关联起来。
- 执行状态：未处理（等待处理），处理中（防止被其它 Job 抢占），成功（最终成功了），失败（暂时失败，会继续进行重试），人工介入（永远不会再变了，一定需要人工处理，需要报警通知）
- 重试次数：处理过太多次还是失败的可以归类为死信，由专门的死信队列任务单独再进行若干次的重试不行的话就报警人工干预
- 处理历史：每一次的处理结果，Json 的 List 保存在这里供参考
- 上次处理时间：最近一次执行时间
- 上次处理结果：最近一次执行结果
- 创建时间：数据库维护
- 最后修改时间：数据库维护

除了这些字段之外，还可能会加一些业务自己的字段，比如订单状态，用户 ID 等等信息作为冗余。任务表可以进行归档减少数据量，任务表扮演了消息队列的性质，我们需要有监控可以对数据积压，出入队不平衡处理不过来，死信数据发生等等情况进行报警。如果我们的流程处理是任务 ABCD 顺序来处理的话，每一个任务因为有自己的检查间隔，这套体系可能会浪费一点时间，没有通过 MQ 实时串联这么高效，但是我们要考虑到的是，任务的处理往往是批量数据获取+并行执行的，和 MQ 基于单条数据的处理是不一样的，总体上来说吞吐上不会有

太多的差异，差的只是单条数据的执行时间，考虑到任务表驱动执行的被动稳定性，对于有的业务来说，这不失为一种选择。

这里再说明一下 Job 的几个设计原则：

1. Job 可以由各种调度框架来驱动，比如 ElasticJob、Quartz 等等，需要独立项目处理，不能和服务混在一起，部署启动多份往往会有问题。当然，自己实现一个任务调度框架也不是很麻烦的事情，在执行的时候来决定 Job 在哪台机器来跑，让整个集群的资源使用更合理。说白了就是两种形态，一种是 Job 部署在那里由框架来触发，还有就是只是代码在那里，由框架来起进程。
2. Job 项目只是一层皮，最多有一些配置的整合，不应该有实际的业务逻辑，不会触碰数据库，大部分情况就是在调用具体服务的 API 接口。Job 项目就负责配置和频次的控制。
3. 补偿类的 Job 注意补偿次数，避免整个任务被死信数据卡住的问题。

三马车都说完了，那么，最后我们来梳理一下这么一套架构下整个项目的模块划分：

- *Site:*
    - *front*
    - *console*
    - *app-gateway*
- *Façade Service:*
    - *partnerinvestservice-api*
    - *partnerinvestservice-server*
    - *partnerinvestservice-listener*
    - *normalinvestservice-api*
    - *normalinvestservice-server*
    - *normalinvestservice-listener*
    - *reserveinvestservice-api*
    - *reserveinvestservice-server*
    - *reserveinvestservice-listener*
    - *autoinvestservice-api*
    - *autoinvestservice-server*

    - *autoinvestservice-listener*
- *Business Service:*
    - *tradeservice-api*
    - *tradeservice-server*
    - *tradeservice-listener*
    - *loanservice-api*
    - *loanservice-server*
    - *loanservice-listener*
    - *userservice-api*
    - *userservice-server*
    - *projectservice-api*
    - *projectservice-server*
    - *accountservice-api*
    - *accountservice-server*
    - *accountservice-listener*
    - *activityservice-api*
    - *activityservice-server*
    - *activityservice-listener*
    - *vipservice-api*
    - *vipservice-server*
    - *vipservice-listener*
- *Foundation Service:*
    - *bankservice-api*
    - *bankservice-server*
    - *digsignservice-api*
    - *digsignservice-server*
    - *messageservice-api*
    - *messageservice-server*
- *Job:*
    - *scheduler-job*
    - *task-job*
    - *compensation-job*

这每一个模块都可以打包成独立的包，所有的项目不一定都要在一个项目空间内，可以拆分为 20 个项目，服务的 api+server+listener 放在一个项目内，这样其实有利于 CICD 缺点就是修改代码的时候需要打开 N 个项目。

之前开篇的时候说过，使用这套简单的架构既能够有很强的扩展余地，复杂程度上或者说工作量上不会比 All-In-One 的架构多多少，看到这里你可能觉得并不同意这个观点。其实这个还是要看团队的积累的，如果团队大家熟悉这套架构体系，玩转微服务多年的话，那么其实很多问题会在编码的过程中直接考虑进去，很多时候设计也可以认为是一个熟能生巧的活，做了多了自然知道什么东西应该放在哪里，怎么去分怎么去合，所以并不会有太多的额外时间成本。这三驾马车构成的这么一套简单实用的架构方案我认为可以适用于大多数的互联网项目，只是有些互联网项目会更偏重其中的某一方面弱化另一方面，希望本文对你有用。