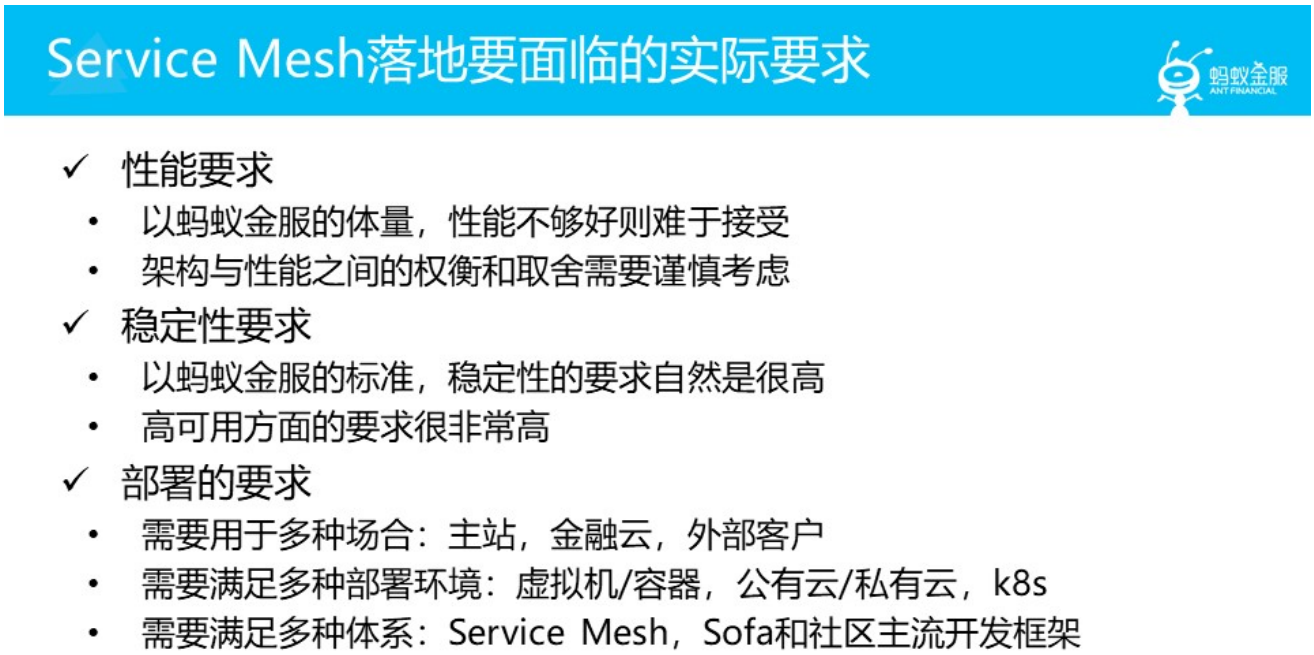
大规模微服务架构下的Service Mesh探索之路

今天给大家带来的内容叫做Service Mesh探索之路，但是在前面加了一个定语：大规模微服务架构下。之所以加上这个词，是因为我们这个体系是在蚂蚁金服这样一个大的架构下进行的，蚂蚁金服的体量大家可以想象，所以这个探索会带有一个非常隆重的色彩：对性能/规模/高可用等方面的思考。

今年6月初，在深圳的GIAC大会，我们同事披露了这个正在开发中的Service Mesh产品，我们现在暂时命名为Sofa Mesh。我们目前的产品都在Sofa品牌下，比如Sofa RPC，Sofa Boot等。今天我们详细介绍Sofa Mesh这个单独产品，上次大会只是简单披露，也就是给大家介绍说我们有这样一个产品，而我今天的内容是把这个产品详细展开。

主要是三个内容：一是Sofa Mesh的技术选型，二是它的架构设计，以及在最后跟大家聊一下，蚂蚁金服在Sofa Mesh上的开源策略。

第一个是技术选型。



先上来一堆要求，刚才我们提到过的，因为是大规模，而蚂蚁金服的体量，大家可以想象到的。实际上在性能，稳定性上，我们的衡量标准，我们考虑的基石，都是以蚂蚁金服这样的一个规模来考虑的。

在这样一个规模下，我们会涉及到一些跟其他公司不太一样的地方，比如说：我们在**性能**的考量上会比较重一些。因为如果性能不高的话，可能没法支撑我们这样一个规模。在考虑性能的时候，就有另外一层考量：架构和性能之间的这个权衡和取舍是要非常谨慎的。性能要求不太高的情况下，架构可能的选择，和需要比较高性能的情况下，可能会有完全不一样的取舍。稳定性就不必说了。

部署方面的要求，首先是我们会用于多种场合：

- 主站是指我们蚂蚁金服内部，比如大家用的最多的支付宝。
- 金融云，可能有一部分和我们有联系的同学会有所了解，这个是我们推出的针对金融行业的云。
- 然后还有我们的外部客户

部署上会要求这三个场合都能使用。

部署环境也会有多种，刚才我们调查到，有部分同学相对比较前沿一些，现在就已经上k8s了。有部分同学还是停留在以前的虚拟机以及物理机这种状态，也有一部分自己上了容器，还有部分同学可能会使用不同的公有云和私有云。这几种不同的环境，我们都是需要满足的。

第三点可能要特殊一些，需要满足各种体系。刚才我们在调查的时候了解到，有部分同学是在做旧有系统改造，那在改造的时候就会遇到一个问题：除了Service Mesh之外，还需要跟原来的体系，比如说Sofa，或者社区主流框架如Dubbo，Spring Cloud，相互之间打通和过渡。怎么在技术转型期间平滑的让业务做变更，是我们在整个技术选型之前提出的实际要求。整个技术选型是在这样一个背景下进行的。

我们做技术选型的时候，有两大方向：

- 一个选择是在开源产品上做

  我们先看右边的路线，起点是找一个开源产品，fork出来，做增强/扩展/定制，和内部集成。因为开源产品还在继续往前走，所以我们会持续做版本更新，也可以从社区拿到最新版本。相当于是从开源社区做**获取**，然后接下来做反馈，让我们的一些产品，我们做的东西反馈回去。

  这条路线比较大的好处是从一开始就可以得到社区的支持，社区往前走的时候也跟着往前走。如果做的比较好，愿意让自己的产品反哺社区，那么社区也可以从中受益。

当然这里面有一个小问题，就是说可能我们自己这个产品路线和开源产品路线可能会有一些分歧，可能我们领先一步，也可能他们领先一步，或者一个事情可能有两个做法。这种情况下，如何让社区的接受我们的改动，会变成这条路线上比较头疼的一个问题。

这是两条路线上的第一条，选择以开源产品为起点。

- 另外一种思路全新打造

  或者，如果手上已经有一套类库或者框架，可以在这个基础上做包装。

  这条路线有一个好处，**可控性**比较强。因为整个体系是全新打造或者在原有体系上演进而来的，整套体系基本上都是自己的开发团队完全可控的。

  这条路线会遇到一个问题，因为长期上看我们也是希望开源的，而开源就意味着不能将自己内部太多的定制化的东西直接做进去，所以在架构上需要考虑可扩展性，可以定制化。因为开源出去的应该是一个标准产品，这样的产品才可以得到社区和客户的认可。客户希望看到一个干净的东西，也需要做扩展，整个体系在设计上会有所不同。

两条路线的终点，从图上看，我们有两个目标：

1. 第一个目标是内部落地

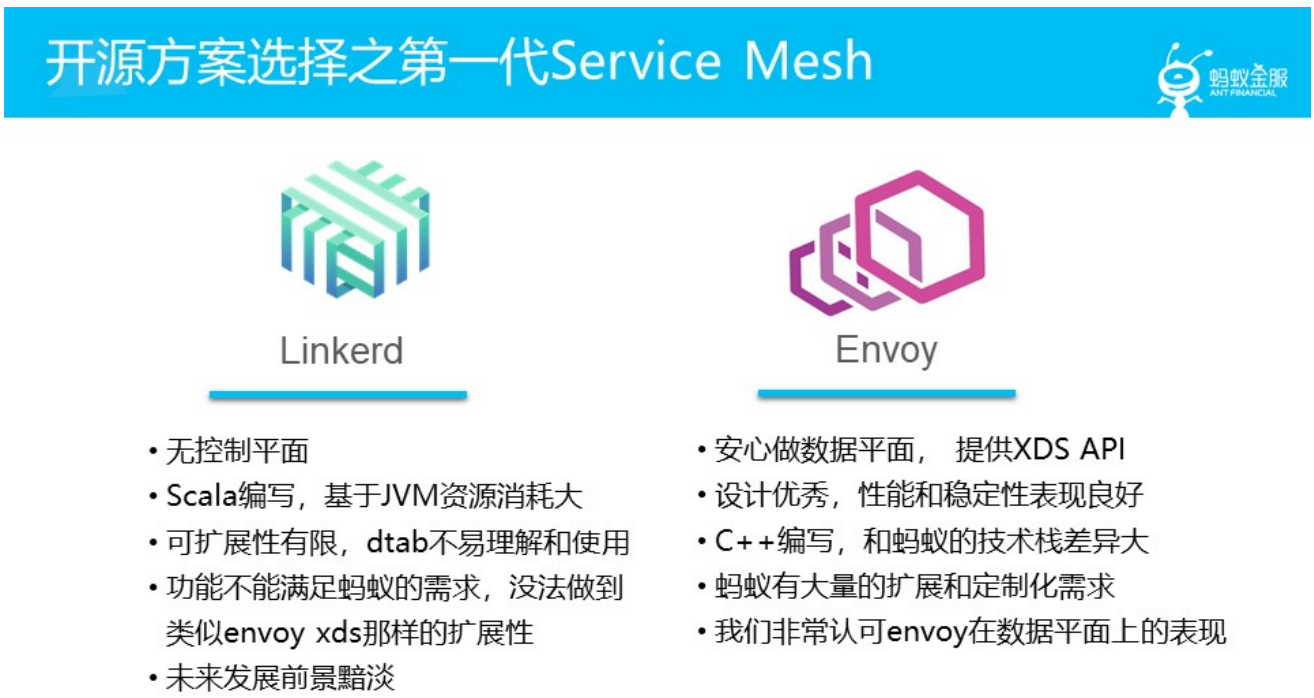
   前面提到的，我们需要在蚂蚁金服主站这样的一个巨大规模的场景下落地，这是蚂蚁金服自身的需求。

2. 第二个目标是技术输出

   因为蚂蚁金服在公司策略上有科技输出的内容，不仅仅我们自己用，我们还需要给出去。

现在我们来看这个问题：目标在这里，然后有左右两条路线，我们该怎么选择？在做的技术选型的时候，这是一个非常大的分歧点，到底是从左边走，还是从右边走？

在公布结果之前，我们先来看一下有什么可选方案。



这是开源方案的选择，第一代的Service Mesh。

左边的Linkerd，这个基本上，目前看，大家都已经有点嫌弃了。因为它没有控制平面，用Scala写的，基于JVM，资源消耗比较大。它的可扩展性比较有限的，相对于Envoy的扩展性。然后它里面有个dtab，有接触到的同学就会有认识：dtab的语法，非常的不人性，很难理解，使用不太方便。另外它的功能是远远不够的，对于蚂蚁金服来说。另外这个产品本身的发展前景已经很暗淡了，所以这个选项就被淘汰了。

Envoy是非常不错的，做了一些令我们意外的事情：安心的去做好数据平面，没有往上面做很多的东西，而是创造性的提出了XDS API。整个设计是非常优秀的，性能和稳定性也表现得非常好，甚至看到业界有一个趋势，有一部分的公司开始把他们的nginx替换了，不再用nginx了，而是用envoy。也就是说，现在它的稳定性和性能达到和nginx一个级别，nginx大家应该都有听说过，envoy已经是这样一个工业成熟度。

我们当时选型时是比较头疼的，因为它是c++写的，c++14。和我们技术栈的差异会比较大，因为蚂蚁的技术栈是以Java为主，长期的话，我们可能部分转到Golang上去。在这种情况下，C++的技术栈，会让我们比较尴尬，也不是说我们找不到会c++的同学，而是说，长期上会和我们的方向不一致，我们要在Java和Golang的技术栈之外再加一个c++，这就比较难受。

然后我们内部会有大量扩展和定制化的需求。因为我们内部有我们自己的产品，我们自己的需求，我们的通讯方案，我们内部的追踪，监控，日志方案，所以工作量非常大。

总结说，我们觉得Envoy很好，但是我们不能简单用。但是它在数据平面上的表现我们是非常认可的，Envoy在这点做得非常好。

开源方案里面的第二代，istio是我们当时的第一选择，重点关注对象。Istio现在最大的问题在于它迟迟不能发布生产可用版本，大家如果对istio有了解的话，会知道istio刚刚发布了0.8版本，第一个长期支持版本，但是这个版本也不是生产可用。不出意外的话，按照目前的进度，istio应该会在7月份发布它的1.0版本，但是从我们目前的感受上看，1.0估计可能还是不能工业级的使用。所以需要等，而我们没法等，但是Istio的理念和方向我们非常认可。大家看一看，我们这个技术选型有多纠结。

右边的Conduit，现在Conduit的最大限制是它只支持k8s。而现在蚂蚁金服还没有普及k8s，我们现在还有很多系统是跑在非k8s上的。第二是它的数据平面是Rust编写的，这个语言更加小众了，在座的同学有没有人了解Rust这门语言？或者听过。（备注：现场大概十几个人举手）大概10%左右的同学听过。好，Rust语言排名大概在50名左右。这个语言本身还是蛮认可的，我还很喜欢这个语言，它的一些特性还是非常有道理，如果掌握好还是可以写出非常好的产品，但是它的入门台阶会比较高一点。这个地方比较讨厌的事情是说，因为这个语言本身比较小众，所

以基本上是没办法从社区借力的。这里可以举个例子，大家可以看一下Conduit的committer的人数，大概是25个左右，还包括像我这种只提交了几行代码的。Conduit从12月份开源到现在已经有半年时间，半年时间只有这么多的committer，其中真正有贡献大概9到10个人，基本上都是他自己的员工。也就说这个产品基本上没办法从社区借力，一个产品，如果大家一起来帮忙，其实很多的细节是可以完善的，但是Conduit就卡在Rust语言上。

然后还是同样有技术栈的问题，因为这个原因，基本上Conduit我们也没法用了。

我们再看一下国内的在Service Mesh领域，其他的一些比较前卫的同学，他们的选择会是什么？

首先是华为，华为自己做了一套Golang版本，名字叫做Mesher。这是由他们之前的一套类库演进而来。它走的路线是，先有类库和框架，然后加proxy，proxy打通了之后再慢慢的开始添加控制平面。这是一条非常非常标准的路线，我这边给一个词叫做**老成持重**，因为这条路是最安全的：每一步都是基于现有的产品，很快就可以到下一个里程碑，然后每个里程碑都可以解决一些实际问题，可以直接得到一些红利，这个方案是比较比较稳妥的。比如说第一步是把proxy做进去，有了这个切入口之后，就在第一时间获取跨语言的红利，还有技术栈下沉的好处。然后控制平面的创新，可以在这个基础上慢慢往前做。

在对接Istio这一条上，现在华为的策略，我们现在从公开途径了解到的是：部分对接istio，也就是有一部分的API兼容Istio。但是细节上还不太清楚，因为它的开源还没出来，目前得到的消息是，会在7月份开源。

第二个是新浪微博的Motan Mesh，他们也是Golang的，但他不太一样，是全新实现。他们用Go语言重新写了一把，主要原因是因为它没有golang类库，Motan是基于Java的。

刚才看到的这两个产品，他们的思路大体上是相同的，差异在哪里？就是启动的时候是用已有的类库还是重新写？这两个选择之间最大的麻烦在于编程语言，华为原来有go的类库，所以继续用golang包装一下就好了。但是新浪的类库用的是Java，而sidecar选择的是go语言，所以只能重新做了。

国内公司的选择之二：开源方案定制

腾讯：Tencent Service Mesh

- 数据平面选择Envoy：成熟产品，符合腾讯语言体系，内部广泛使用
- 控制平面据传“挣扎了一下”，最终还是选择Istio，进行定制和扩展，解耦k8s

我们再看腾讯，最近看到他们有类似的产品出来。我们看看他们的资料：在数据平台上继续选择Envoy，因为它比较成熟。腾讯的话大家比较熟悉，尤其是腾讯有非常深厚的c++背景，所以Envoy对他们来说，技术栈是非常OK的。而且之前内部其他领域Envoy也是在用的，所以底层非常自然的选择了Envoy。然后控制平面上，据传是"挣扎了一下"。这个词是我抄过的，"他们挣扎了一下"，最后还是选了Istio。然后自己做定制和扩展，然后注意到他们也解耦了k8s。这也是其中一个关键的点：要不要绑定k8s?

国内公司的选择之三：另辟蹊径

UCloud：Service Mesh

- 非常有意思的轻量ServiceMesh实践
- 从Istio中剥离Pilot和Envoy
- 去掉Mixer和Auth
- 定制Pilot，实现ETCD Adapter
- 脱离k8s运行

这里还有UCloud的一个很有意思的做法，另辟蹊径啊。他的方案很有意思，是一个轻量级的实践：从Istio里面，将Envoy和Pilot单独剥离出来。就是说不用Istio整体，把Mixer和Auth的模块去掉，只要最重要的Envoy，然后把Pilot剥离出来。然后这个Pilot还是个定制版，把其他的adapter干掉了。Pilot主要是做服务发现，它底层用ETCD，做了一个ETCD的adapter，把其他的adapter从Pilot中去掉。做完这几个事情之后，整个体系就可以脱离k8s了，这是一个比较有意思的实践。

总结：在讲我们技术决策过程之前，我们过了一下目前市场上的主要产品，以及一部分实践者的做法。

我们现在来详细讲一下，Sofa Mesh在技术选型上的考虑。

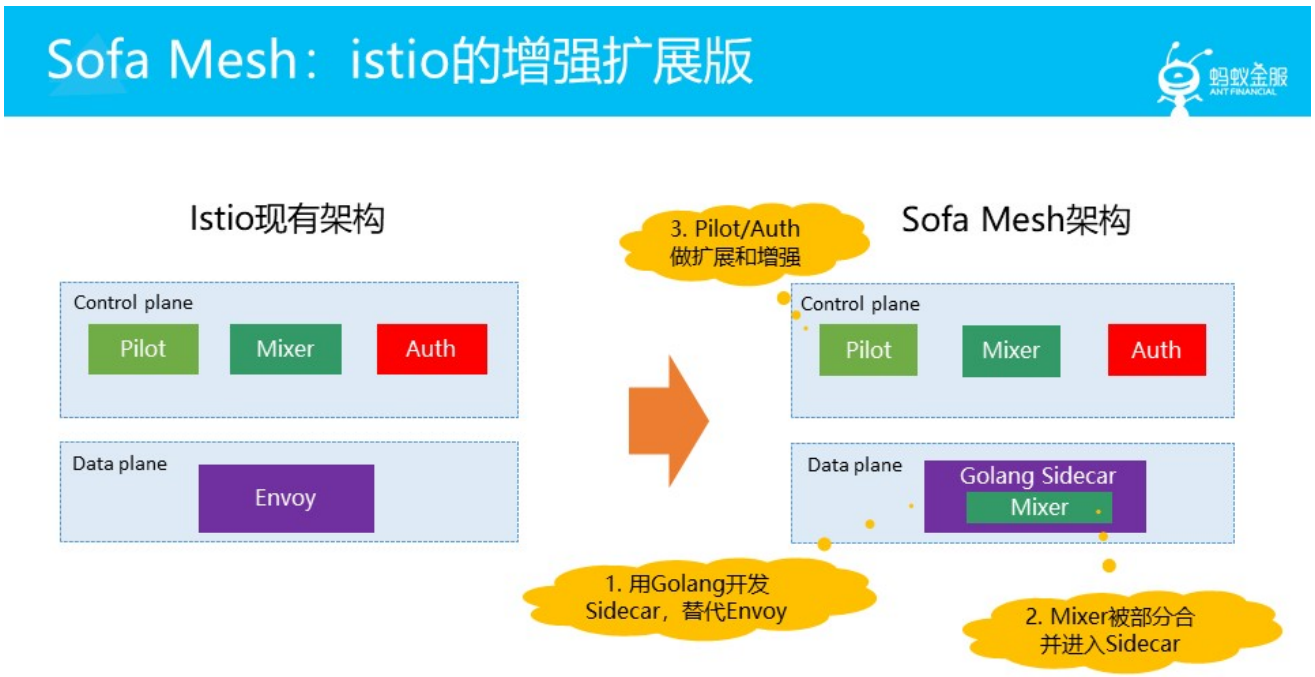
首先第一个，数据平台上Envoy是最符合我们要求的，Envoy确实好。第二个事情是Envoy提出的XDS API设计是非常令人称道的，我们现在对这个的评价是非常高的。它实际上是一套通用的API，由于时间的缘故，我今天就不在现场展开API的细节。只能说XDS API基本上已经成为数据平面和控制平面之间的一个事实标准。

在这种情况下，我们其实是想用Envoy的，但是刚才提到我们有个技术栈选择的问题：我们不愿意将c++纳入到我们主流的技术栈。然后我们本身有太多的扩展和定制，逼得我们不得不去改Envoy，我们不能简单的拿过去用，我们需要做很多扩展的。

另外一个事情是，我们这个proxy不仅仅是用于Mesh，我们有可能把它引入到API Gateway里头，以及后面会提到的名为Edge Sidecar的模块。因为这个原因，所以，怎么说呢，想用，但是不合适用。

第二就是在Istio上，控制平面这一块Istio可以说是做的最好的。基本上，到目前为止，在控制平面上，暂时我们还没有看到做的比Istio更好的产品，或者说思路。目前Istio整个设计理念，包括它的产品方向，也是我们非常认可的。

但是Istio的性能是目前最大的问题，而我们有一个重要的前提：大规模应用。要用在蚂蚁金服主站这样一个场景下，性能和稳定性对我们非常非常的重要。第二个问题是它对非k8s的支持不够理想，因为我们还涉及到一个k8s没有完全上线的问题。第三个是和侵入式框架互通的问题，我们内部用的是SOFA，对外推出的时候我们的客户用的可能是Dubbo或者Spring Cloud，Mesh上去之后，两个系统现在走不通，这是大问题。



最终我们的策略是这样的，这是我们Sofa Mesh的技术选型：左边是Istio现有的架构，Envoy/Pilot/Mixer/Auth，右边是我们Sofa Mesh的架构。

- 最重要的第一点：我们用Golang开发的Sidecar替换Envoy，用Golang重写整个数据平面。
- 第二点是我们会合并一部分的Mixer内容进到Sidecar，也就是Mixer的一部分功能会直接做进Sidecar。
- 第三点是我们的Pilot和Auth会做扩展和增强。

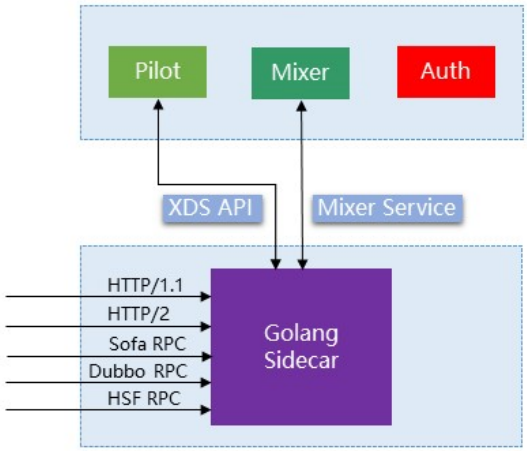
这是我们整个的技术选型方案，实际上是Istio的一个增强和扩展版本，我们会在整个Istio的大框架下去做这个事情，但是会做一些调整。

这是今天的第一部分内容，Sofa Mesh的技术选型。

然后我们来详细介绍一下在这个技术选型上我们怎么去做实现。

Golang版Sidecar

- ✓ 参照Envoy的设计
- ✓ 实现XDS API
- ✓ 兼容Istio
- ✓ 支持HTTP/1.1和HTTP/2
- ✓ 扩展Sofa/Dubbo/HSF

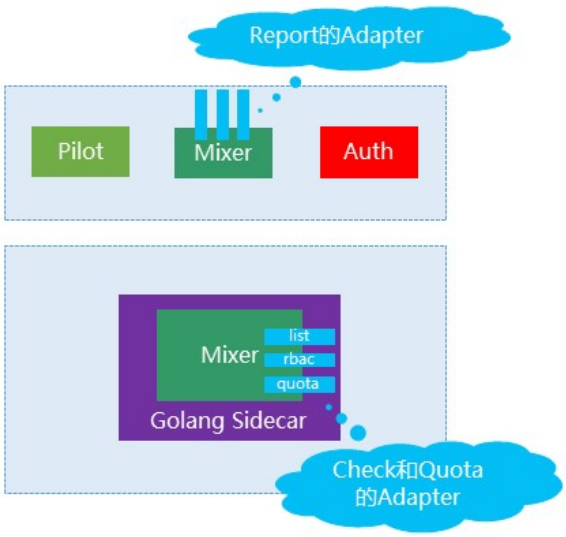


首先是Golang版本的Sidecar，我们会参考Envoy，非常明确的实现XDS API。因为XDS API是目前的事实标准，所以我们选择遵循，然后我们会让它兼容Istio。

在协议支持上，我们会支持标准的HTTP/1.1和HTTP/2，也就是大家常见的REST和gRPC协议。然后我们会增加一些特殊的协议扩展，包括Sofa协议，Dubbo协议，HSF协议。我们现在正在做这几个协议的扩展，然后XDS API我们支持，mixer service我们没有改动，遵循现有实现。

最大的改变：合并部分Mixer功能

- ✓ Mixer三大功能：
  - Check – 同步阻塞
  - Quota – 同步阻塞
  - Report – 异步批量
- ✓ 合并Check和Quota
- ✓ Report暂时保留在Mixer中



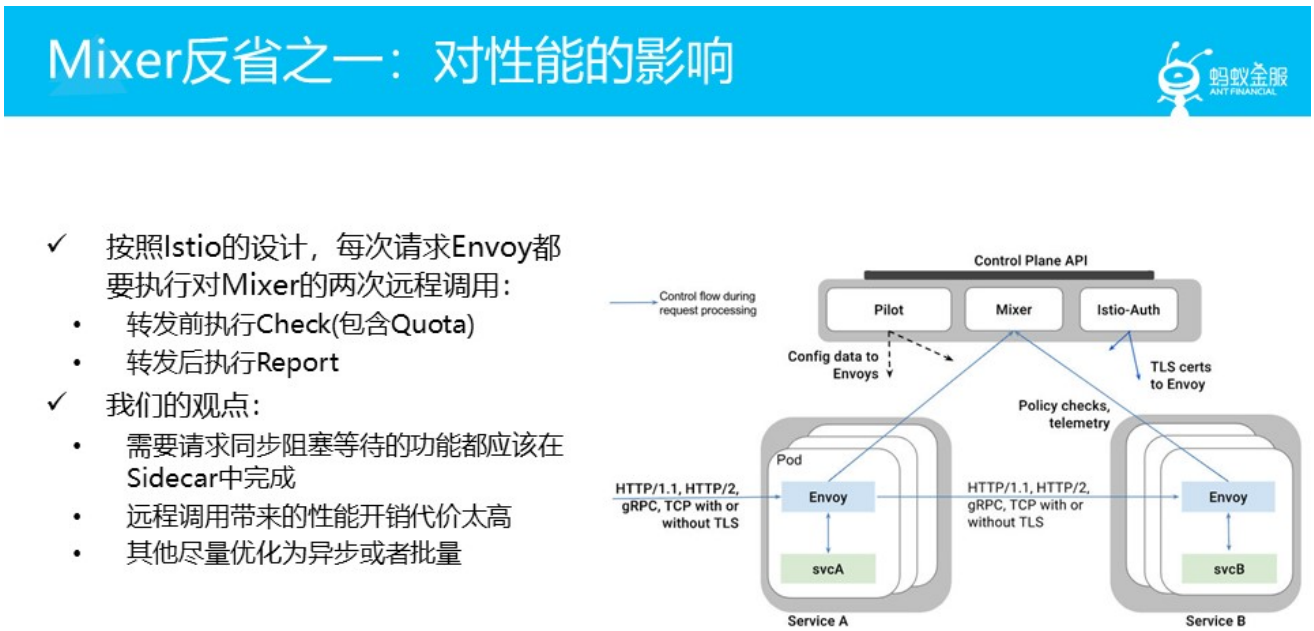
最大的变化在Mixer，其实刚才的Sidecar虽然是全新编写，但是说白了是做Envoy的替换，在架构上没有什么变化。但是第二步的变化就非常大，我们会合并一部分的Mixer功能。

Mixer的三大功能：

1. check。也叫precondition，前置条件检查，比如说黑白名单，权限。
2. quota。比如说访问次数之类。
3. report。比如说日志，度量等。

三大功能里面，注意到，前两个功能是同步阻塞的，就是一定要检查通过，或者是说quota验证OK，才能往下走。如果结果没回来只能等，因为这是业务逻辑，必须要等。而Report是可以通过异步和批量的方式来做的。

在这里，我们现在的决策是：我们会将其中的两个部分(check和quota)合并进来，原有report部分我们会继续保留在mixer里面。



可能大家会问：为什么我们要选择用这个方案，而不是遵循Istio的标准做法？我们之前聊到，我们会尽量去和Istio做兼容，跟随Istio的设计理念和产品方向，但是我们在它的架构上做了一个重大的调整。为什么？

最大的问题就是**对性能的影响**。

给大家解释一下，看右边这个图，Envoy在每次请求进来的时候，要去做两次调用：

1. 第一次在请求转发之前要做一次check，这个check里面包含了quota。Check完成通过，才能把请求转发过去。
2. 请求转发完成之后，再调用report，报告一下响应时间，日志，度量等信息

每次traffic都会有两次调用：一次check，一次report。而这是远程调用，因为这两个模块是两个进程，Mixer是单独部署的。

同步阻塞意味着必须要等，**远程调用**意味着有开销而且有延迟。这个事情是发生在**每一次**请求里面，意味着整个的性能一定会受影响。而考虑到我们蚂蚁金服这样一个体量，其实我们是很难承受。所以我们有自己的观点：我们不是太认可这样的一个方式，我们的想法是说我们要把它拆分出来想一想：

- 如果是需要请求做同步阻塞的功能，比如说黑白名单的验证，可能要检查IP地址，可能检查quota。这些逼请求一定要做同步阻塞等待结果的功能，就**不应该放在Mixer中**去完成去远程调用，而应该在Sidecar中完成。

  这是我们的观点，原因就是远程调用带来的系统开销，这个代价实在是太高了！

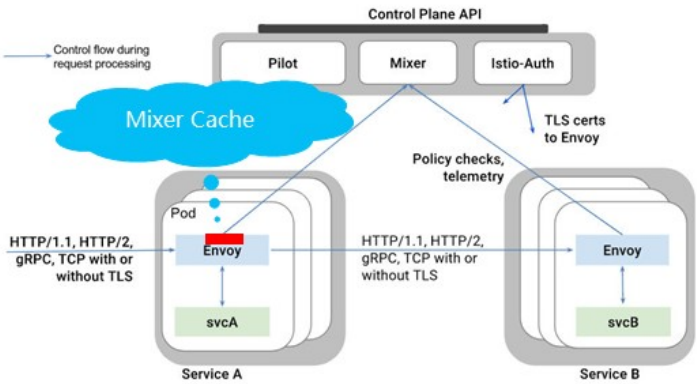
- 然后其他的功能，比如说可以优化为异步的，或者可以以批量方式来提交的，最典型的就是Report。Report其实是可以异步提交，可以把十个请求打包到一个report同时提交，这些都是OK的。

这是我们的基本想法。

Istio的解决方案：添加Mixer Cache

✓ 缓存的工作方式：

- Sidecar 中包含本地缓存，一部分的前置检查可以通过缓存来进行
- 另外，Sidecar 会把待发送的Report数据进行缓冲，这样可能在多次请求之后才调用一次 Mixer
- 前置检查和配额是同步的
- Report数据上送是使用 fire-and-forget 模式异步完成的

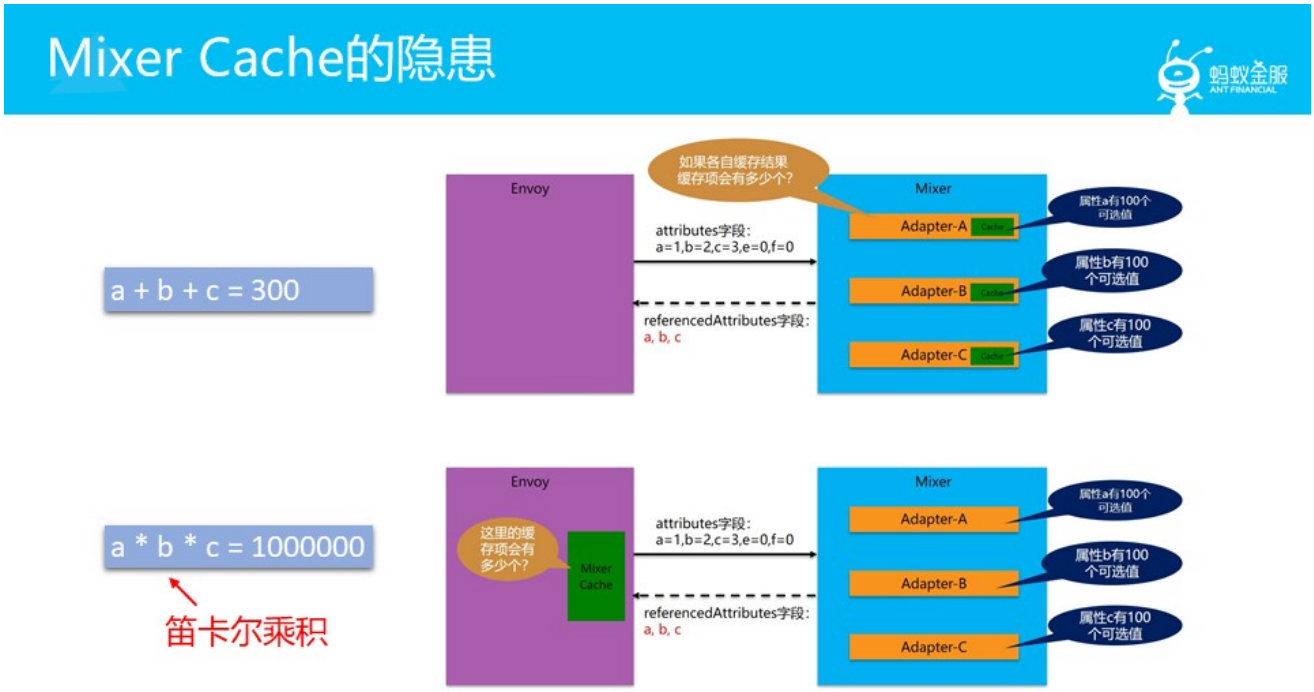


这个问题其实在Istio里面是给了一个解决方案的。最早的时候，Istio 0.1版本中，一出来就发现这个问题。从去年5月份开始到现在，13个月的时间里，他只给了一个解决方案，就是在Mixer上的这个位置加了一个Cache。这个的Cache的想法是：把这些结果缓存在Envoy的内存里面，如果下次的检查参数是相同的，那我们可以根据这样一个缓冲的设计，拿到已经缓存的结果，就可以避免远程调用。这个方式是很理想的，对吧？只要缓存能够命中，那就可以避免这一次远程调用。

然后第二个优化是report，现在的report是通过异步模式完成的，而且是批量。

理论上说，如果这两个事情做到足够理想，Mixer应该就不是瓶颈。对吧？

问题在于：这个Cache真的搞得定吗？



我们给一个简单的例子，我现在假设Mixer有三个adapter。然后它的输入值是不同的属性，属性是istio的概念，理解为若干个输入值。假设，需要三个adapter分别检查A/B/C。如果这三个属性A/B/C，他们只有100个取值范围，每个都是从0到100，我们假设这种最简单的场景。

如果这三个adapter分别做缓存的话，需要多少个缓存项？很容易计算吧？100个a，100个b，100个c，非常容易计算，这种情况下，其实就是a+b+c等于300嘛。理解一下：有三个输入，每个输入只有一百个取值范围，我们要把他们缓存起来。这些缓存大小，就是允许的范围，然后加起来。只要有300个key，就都可以缓存起来。

但是，这个方法中，缓存是做在mixer这边，每个adapter单独缓存。但是，在Istio中，缓存是做在Envoy这端的，因为做在mixer这端是没有用的，还是要远程调用过去。它做缓存的很重要的目标是要在客户端避免远程调用。所以，这种情况下，把缓存放到这里（备注，图中绿色方块）。

大家现在想一想，现在这里只有一个缓存，只有一个key/value。现在还有刚才的这个场景，A/B/C各自的取值范围都是一百。但是现在缓存放在这边的话，实际上的这个key要考虑三个值了，A/B/C的组合。这种情况下，它的最大缓存个数是多少？

（备注：现场回答，a 乘 b 乘 c）

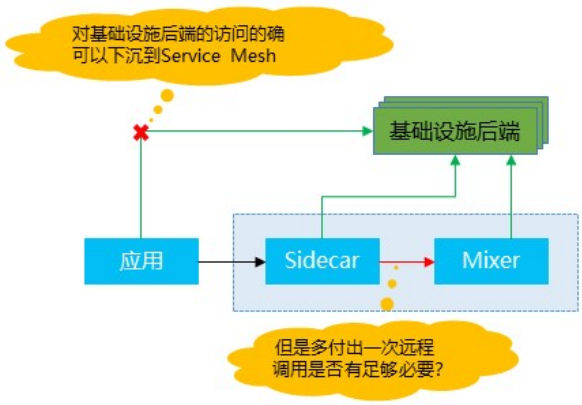
a * b * c？还能 a + b + c吗？做不到了，对不对？现在是 a * b * c，从300变成这么大的数了。为什么？因为缓存是在这个地方做的，根本没有办法像这样分开做，所以这里就变成了一个笛卡尔乘积。

这个笛卡尔乘积有一个很大的麻烦，也就是说，如果adapter检查的某个属性，它的取值范围比较大，比如说要检查客户端的IP地址？你想想，这个IP地址有多少个取值范围？数以几十万几百万计，对吧？这种情况，哪怕在前面再乘以特别小的值，哪怕只是十，二十，如果是加20根本没所谓的，加200，加2000都没所谓的，那乘个200，乘个2000试一下？瞬间就被干掉。IP地址可能只是百万级别，再在前面乘个100，乘个1000，瞬间就疯掉了。这个key值基本上已经是大到不能接受：要么就全放内存，内存爆掉；要不然限制缓存大小，就放1万个，缓存的命中率会非常低，整个缓存相当于失效了。

这个细节，因为时间原因，不在这里详细讲了。

Mixer反省之二：隔离和抽象的层次

- ✓ Mixer的设计目标：
  - 提供统一抽象(Adapter)
  - 隔离基础设施后端和Istio其他部分
  - 容许运维对所有交互进行精细控制合并Check和Quota
- ✓ 我们的反思
  - 认可这样的抽象和隔离，确实有必要从应用中剥离出来
  - 但是要加多一层Mixer，多一次远程调用
  - 抽象和隔离在Sidecar层面完成，也是可以达到效果的
  - 对于Check和Quota，性能损失太大，隔离的效果并不明显

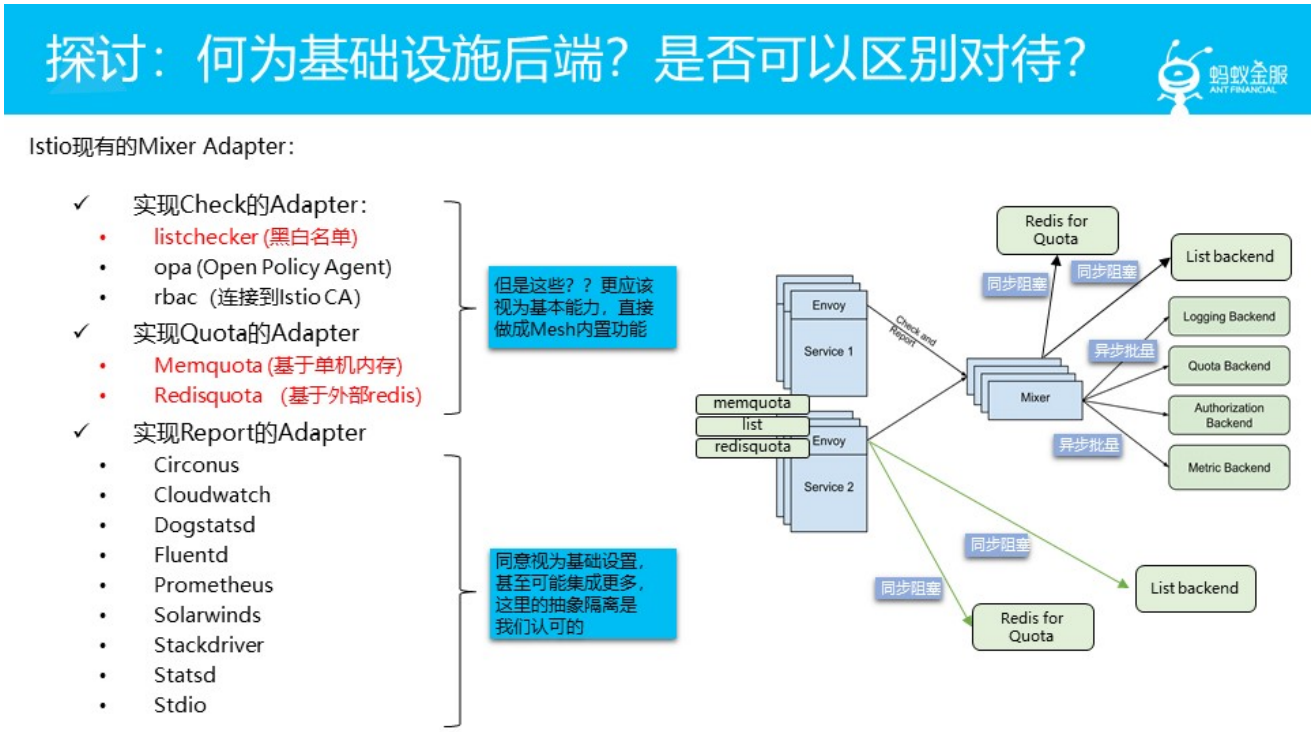


这里讲第二点，我们的反省：隔离怎么做?

Mixer有一个基本的设计目标，就是希望提供一个统一的抽象（就是这个adapter的概念），用它来隔离基础设施后端和Istio的其他部分。但是在这个点上我们的反思是：我们认可这样一个隔离。大家理解基础设施后端的概念吧？举个例子，日志处理如prometheus，各种后端监控系统。这些系统和应用之间，我们认为这种情况下的确应该做隔离，没必要每个应用都去和基础设施后端产生直接的联系。这个观点是我们是赞许的。

但是我们现在的意见是，我们把这条线(备注：连接应用和基础设施后端的标记有红叉的线)从应用里面拿下来之后，我们把它下沉。下沉到Sidecar，够不够？Istio的做法是，它觉得这个地方应该再往前走一步，到Mixer里面。由Mixer去完成和基础设施后端的连接，走这根线（备注：图中连接Mixer和基础设施后端的线）。但是多了这样一个隔离之后的代价，就是在中间的这根红线上，会多一次远程调用。

现在只有两个选择：和基础设施怎么连？这条线（备注：最左边的）大家都认为没必要，这两条线（备注：中间和右边的线）之间选，两条线的差异，就是要付出一次远程调用的代价。



继续反省：什么是基础设施后端?

这里我们做一个列表，整个Istio现有的adapter，大家可以看到，大概是这些。前面这两个部分是实现check和quota的adapter，后面这些adapter是实现report功能。

在这里，我们的反省是：这些功能，比如说黑白名单，比如说基于内存的quota，或者基于外部redis的quota。我们认为这些功能不太应该视为后端基础设施，因为这些功能更应该是说是体系内置的基本能力，应该直接把它们做成Mesh的内置产品，或者说可以做标准化，然后和外部系统集成。这些我认为应该是Mesh的最基础的功能，比如说我们Sofa Mesh可以提供基于Redis的quota方案，直接就把这个功能给出来了。我不认为应该再去跟外界的一个所谓的基础设施后端发生联系。

但是下面这些我们是觉得OK的。这些adapter大家有概念吧，prometheus大家应该都接触过的。剩下的这些在国内可能用的不多，是各种日志和metric相关的功能。把这些视为基础设施后端，我们是非常理解的。包括我们内部，我们蚂蚁也有很多这样的系统，相信各位自家的监控方案也是不一样的。

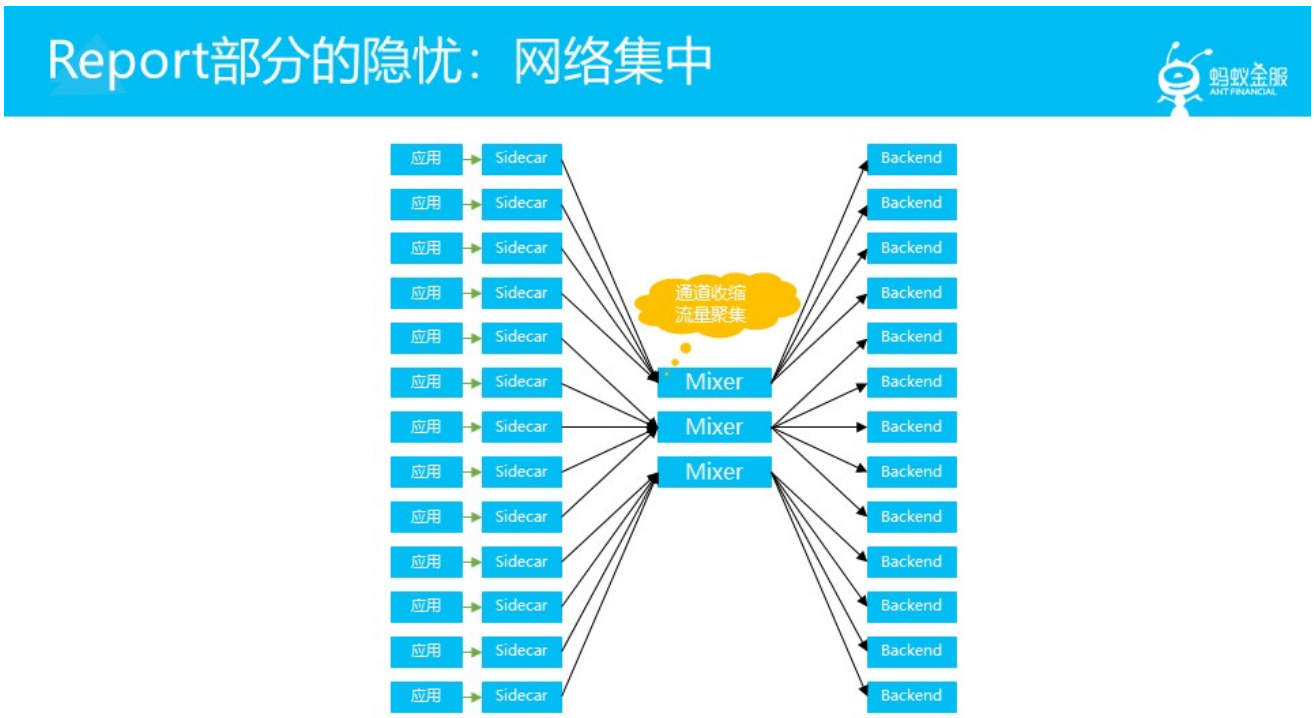
这些视为基础设施，和系统隔离开，我们认为这是非常有必要，可以理解，可以接受。

这是我们在这一点（备注：何为基础设施后端）上和istio的差异。

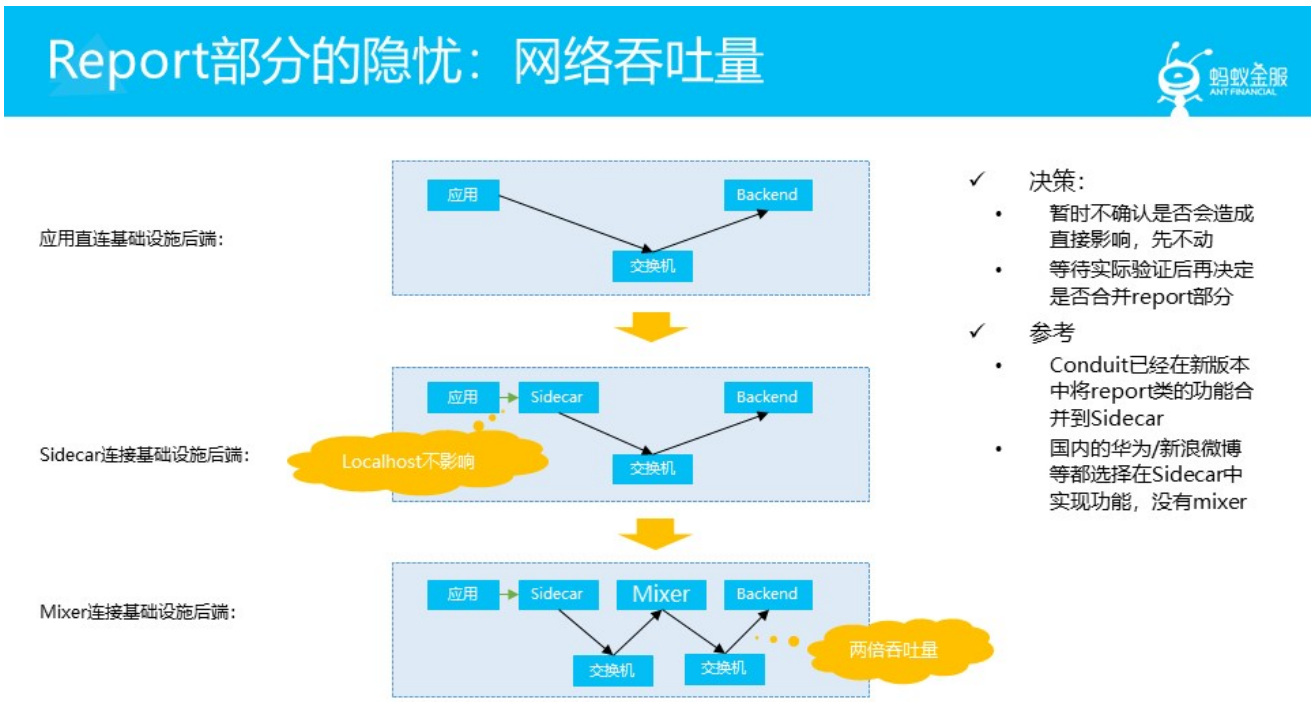
因为时间原因，我们就不再深入去讲，这里我给了一些我博客上的文章。前段时间，我们在做技术选型，在做前面整个架构设计时，在这一点上有些讨论。以及我们最重要的决策：为什么要把Mixer合并进去。细节都在这几篇文章里面，大家如果有兴趣，可以去详细了解。

备注链接地址：

- Service Mesh架构反思：数据平面和控制平面的界线该如何划定?
- Mixer Cache: Istio的阿克琉斯之踵
- Istio Mixer Cache工作原理与源码分析(1) - 基本概念
- Istio Mixer Cache工作原理与源码分析(2) - 工作原理
- Istio Mixer Cache工作原理与源码分析(3) - 主流程
- Istio Mixer Cache工作原理与源码分析(4) - 签名



我们还有一部分现在没有合并进来的adapter和mixer，report的这部分。但是这块不是说完全没有问题，我们现在有一个担心，report这块可能会存在一个叫做**网络集中**的问题。比如说，大家会注意到，应用和Sidecar是一对一部署的，有一万个应用，就有一万个Sidecar。基础设施后端也是多机部署的。而现在的方式，流量会先打到Mixer来，Mixer也是高可用的，也是会部署多台。但是这个数量肯定不是一万这个级别，跟这个肯定会有很大的差异。这样流量会先集中，通道会突然间收缩一下。总的流量没变，但是通道的口径要小很多。



对网络吞吐量也会有影响。

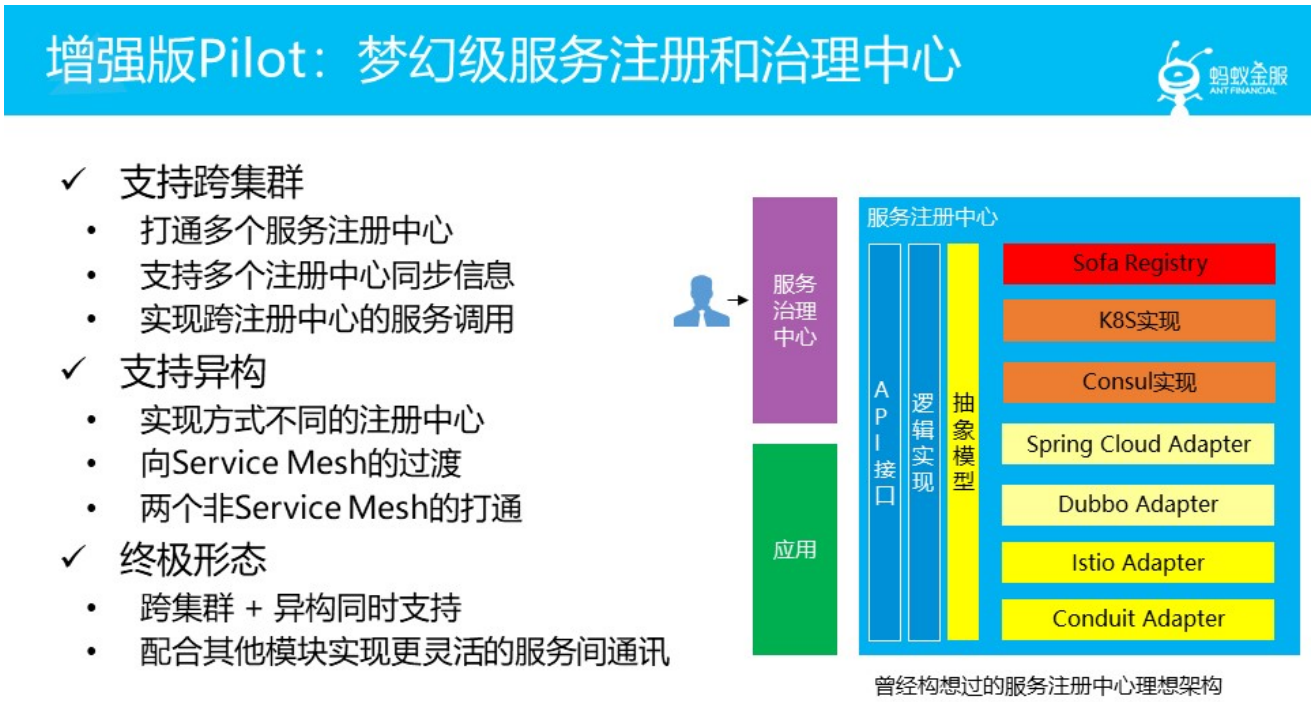
比如最简单的，如果应用直连，走交换机直接就过去了。

如果是Sidecar模式，是在这个位置上（备注：应用和sidecar之间的绿色连线）加一个远程调用，但是应用和Sidecar之间走的是localhost，localhost根本就不走网卡，直接环回地址就走了。对性能不会有什么影响，对网络流量的影响就为零了。所以这两个方案相比，吞吐量不会有变化。

但是，如果在Sidecar和Backend之间再加一个Mixer，这意味着要走两次网络，这样的话会有一个流量翻倍的问题。

所以这个地方可能会带来一些问题，但暂时我们现在还没做决策，我们现在还不是很确定这个问题会不会导致质的影响。所以我们现在暂时还是把它放在这里，就是说我们后面会做验证，如果在我们的网络方案下，这个方式有问题的话，我们可能再合进去。但是如果没问题的话，我们认为分开之后架构确实会更理想一些，所以我们现在暂时先不合并。

给大家一些参考，目前Conduit最新版本已经把report的功能合并进来，然后check的功能，会在后续的计划中合并。我们在国内做一些技术交流，华为新浪微博他们现在通通都是选择在Sidecar里面实现功能，不走mixer。

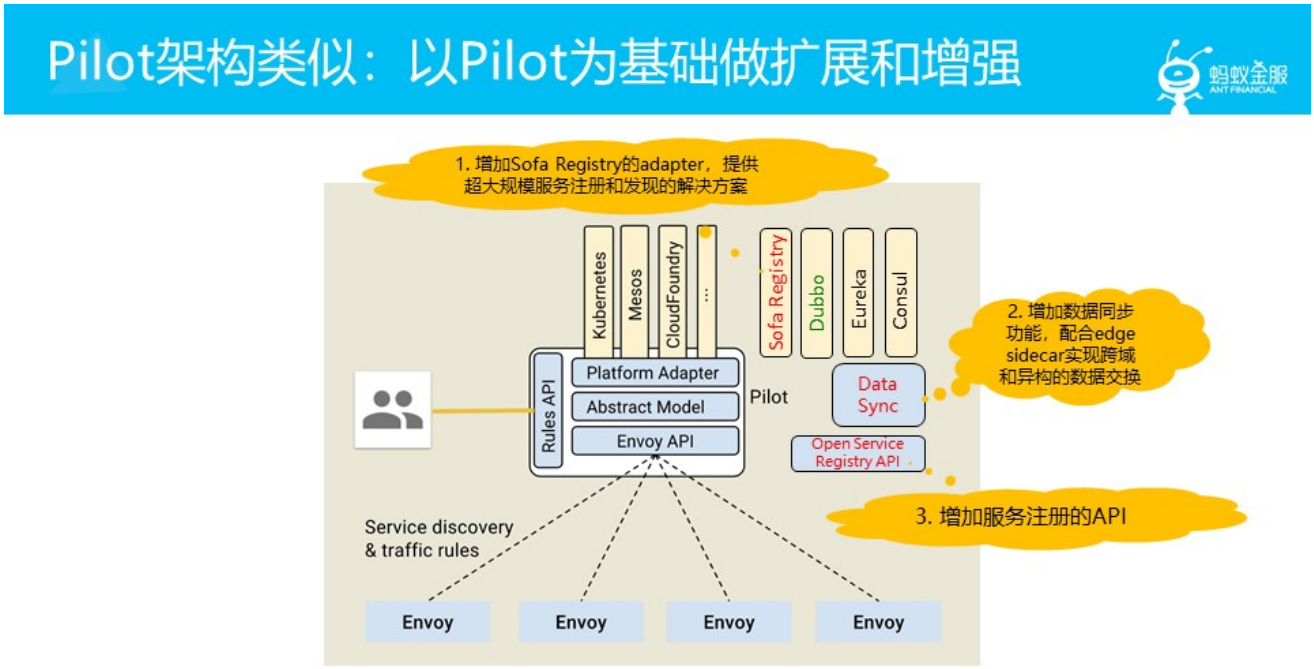


曾经构想过的服务注册中心理想架构

这是我们称之为梦幻级别的服务注册和治理中心，我们对他的要求是比较多的：

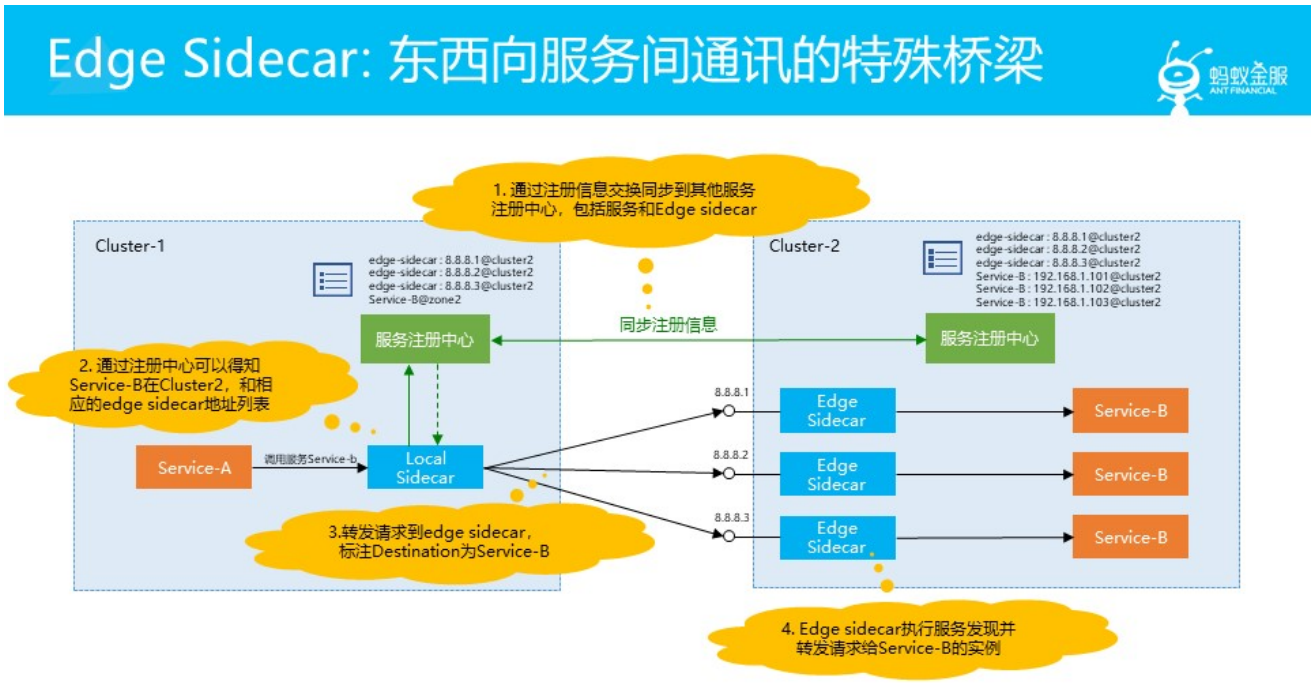
- 我们需要他支持跨集群，比如说我们现在有多个注册中心，多个注册中心之间可以相互同步信息，然后可以做跨注册中心的调用
- 还有支持异构，注册中心可能是不一样的东西。能理解吧，有些是Service Mesh的注册中心，比如Istio的，有些是Spring Cloud的注册中心，比如Consul。
- 然后终极形态，我们希望在两种场景都可以支持。

右边的这个图，是我们构想中的比较理想化的注册中心的架构，我们会有各种adapter实现，会有一个抽象的模型，把他们抽象起来，然后有一些接口。后来，在我们实现的时候发现，Istio的路线跟我们有点像，Istio本身也是做了跨平台的Adapter，也做了一层抽象，然后它也提出了一些API。所以我们最终的决策是：往Pilot靠。



我们以Istio的Pilot模块为基础去做扩展和增强：

- 增加Sofa Registry的Adapter，Sofa Registry是我们内部的服务注册中心，提供超大规模的服务注册和服务发现的解决方案。所谓超大规模，大家能理解吧？服务数以万计。
- 再加一个数据同步的模块，来实现多个服务注册中心之间的数据交换。
- 然后第三点就是希望加一个Open Service Registry API，增加服务注册，因为现在Istio的方案只有服务发现，它的服务注册是走k8s的，用的是k8s的自动服务注册。如果想脱离k8s环境，就要提供服务注册的方案。在服务发现和服务模型已经标准化的情况下，我们希望服务注册的API也能标准化。



这里还有一个比较特殊的产品，因为时间限制，给大家简单了解一下。

我们计划的Edge Sidecar这个产品，它是东西向服务间通讯的一个特殊桥梁。所谓东西向，大家能理解吧？东西向指服务间通讯，也就是A服务调用B服务。对应的还有南北向，南北向通常是指从外部网络进来调用服务，如走API Gateway调用服务。在东西向通讯中，我们有时会需要一个比较特殊的途径，比如说在这个图中，我们有两个集群，两个集群各有各自的服务注册中心。我们通过增强Pilot的方式打通两个注册中心，可以知道对方有什么服务。

当A服务发出一个请求去调用B服务的时候，由于两个集群是隔离的，网络无法相通，肯定直接调用不到的。这时local sidecar会发现，服务B不在本集群，而在右边这个集群里，Local Sidecar就会将请求转发给Edge Sidecar，然后由Edge Sidecar接力完成后续的工作。

7月底北京，第二次Service Mesher线下Meetup

这个模块的功能会比较特殊一点，因为时间限制，在今天的过程当中，Pilot和Edge Sidecar就不再详细展开。

下个月在北京的meetup上，我们这边负责这一块工作的专家，俊雄同学，会给大家详细展开。

现在讲第三部分，大家更关心的一部分，开源策略。

Sofa Mesh的开源策略，可能会和大家之前接触到的一些开源产品，有质的差异，非常的不一样。

蚂蚁金服，开源开放

- ✓ 从 4 月份开始逐步开源金融级分布式架构中的各个组件：
  - SOFA Boot
  - SOFA RPC
  - SOFA Tracer
  - SOFA Lookout
- ✓ 科技开放，走出去看更大的生态
  - 蚂蚁有丰富的业务场景，技术体系也经历了很长时间的发展，沉淀了很多自研产品
  - 蚂蚁本身业务上的开放策略，要求技术也要开放，而且要在更丰富的场景下去磨炼
  - 在此期间，我们趟坑无数，走了N多弯路，演进了N个版本，我们期望能过通过开源和开放，让社区跑的更快，节省更多时间
  - 我们认为金融领域下的分布式架构设计有独特的原则，作为实践者，我们期望能在标准化上跟社区一起沉淀和共建，期望做些贡献，有些建树

备注：这块就不整理了，直接看图中文字。

这是整个大的愿景。

Sofa Mesh的开源态度

Sofa Mesh的开源态度，其实我写左边这些的时候是有很大压力的。用官方话语说，不针对任何人和任何项目，我们不影射任何人。

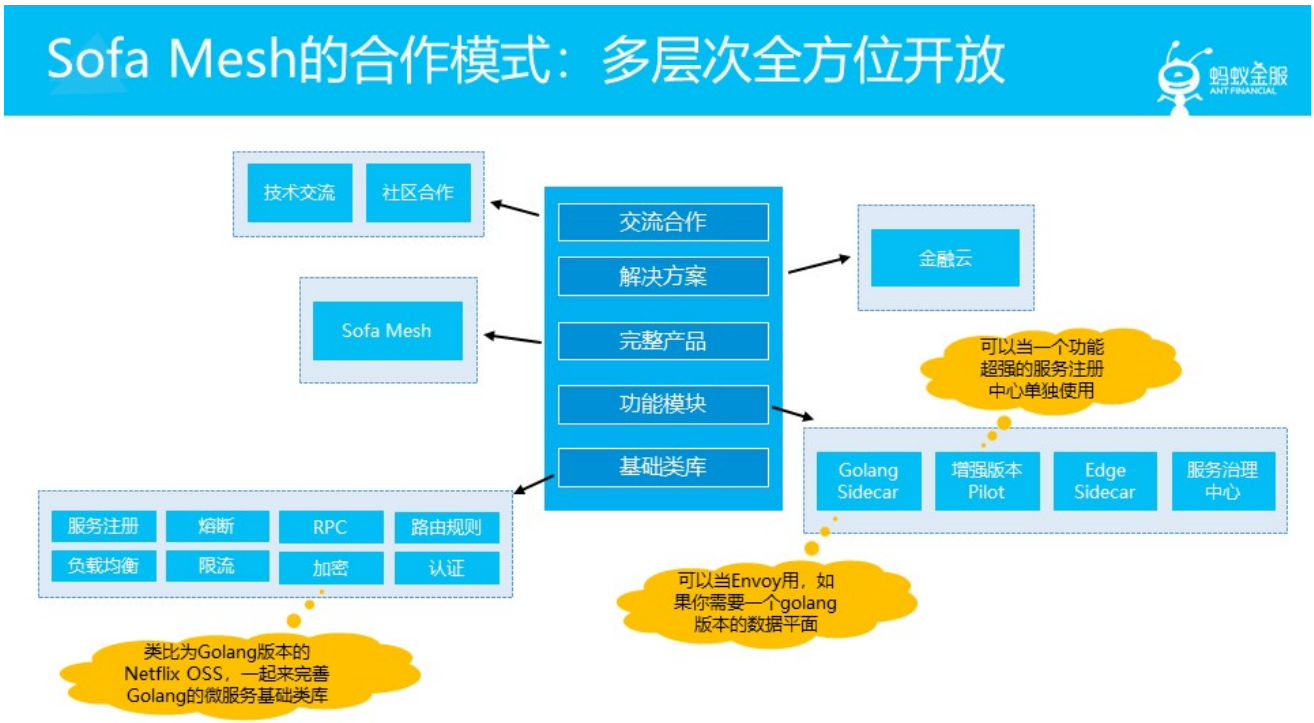
但是，大家如果经常用各种开源产品的话，会发现一些问题。比如说，开源的时机。大家接触的开源产品，尤其是国内的，不管是多大的公司，通常都是产品完成之后，甚至是使用好多年。好处是相对稳，缺点是什么？（备注：现场回答，老）对，技术可能已经很老了，十年前的！还有可能是它都已经放弃了，开源出来时自己不再使用。或者说是一个很新的产品，真的很新，他自己不用，说就是做出来给你用的。（备注：现场哄笑）自己不用的产品给你用，你的第一反应是什么？小白鼠是吗？你愿意做小白鼠吗？你敢把公司的这个产品放上面吗？

Sofa Mesh这次比较特殊，非常非常特殊。我们这个产品，会在非常早的时间点上开源给大家。我甚至可以跟大家说，其实在这个点上，我们更重要的是摆明态度：我们要开源，我们要把这个产品开源给大家，甚至早到我们自己都不认为这是一个完整的产品。为什么？

有几个事情，这几点大家认可吧？业界最新的技术，Mesh是最新技术大家都已经达成共识了吧？业界最好的架构，当然这个我们还在努力中，尽量做好。然后我们会给大家一个承诺，大家不用担心做小白鼠，你能拿到的产品，我们已经趟过一遍了。

开源动机，这个地方我们也不说大话，就是我们希望能吸引整个社区，谋求这样一个合作，走开源共建的方式。这是为什么我们会选择在现在这个时间点上开源出来。

整个产品的维护，什么样的产品会让你有信心，不用担心中间断掉？最重要的一点是我们自己在用。想想，如果支付宝在用，你担心这个项目死掉吗？对不对？如果这个产品本身是蚂蚁金服这样级别的公司，在它的线上将会使用的产品，而且是同样一个核心的版本。相信在这种情况下，大家就放心了吧？



Sofa Mesh的合作模式，我称之为"多层次全方位开放"。

中间这幅图，最底下的是**基础类库**，实现各种功能。我们希望有这样一套基础类库，类比Netflix的OSS套件。因为Golang的类库做的不是很好，没有Java沉淀的那么好。目标是希望在这个产品做完之后，能给整个社区沉淀出一套Golang的微服务基础类库。最重要的一点，是希望最好能大家合力，在这个点上做出一套成熟稳定性能足够好的产品。这是在类库层面。

在类库之上，功能模块层面，比如说Golang版本的Sidecar，我们希望它能替换Envoy的功能。在原来使用Envoy的情况下可以使用这个Sidecar来替代。体现在什么层次？就是说，如果想用Envoy，也很喜欢它，但是可能又受限于C++语言栈，更希望是Golang语言栈的时候，可以选择我们这一套。或者如果我们抱有同样的想法，比如想把Mixer合进来，可以在Sidecar这个层面上来重用我们的产品，跟我们做合作。或者我们刚才提到的这个产品，增强版本的Pilot，大家有印象吧？我们会实现一个非常强大的，跨各种集群，各种异构的服务注册机制。然后是Edge

Sidecar，在两个不同的区域之间，比如两个不同的机房，IP地址不通的情况下，帮你打通服务间调用。这些功能模块，会以单独的产品和项目出现，你可以在某一个产品上跟我们合作。

第三点就是完整的产品，如果你需要一个完整的Service Mesh的产品，把这些所有的功能都包括进来，没问题，Sofa Mesh可以拿来用。

有些同学可能会需要更完整的解决方案，我们的金融云会提供Sofa Mesh的支持，这是我们的目标。你可以将你的系统，架构在金融云之上。

今天的几位讲师来自不同的公司，我们非常欢迎业界参与。如果大家有意在Service Mesh领域做一些事情，大家可以相互之间做技术的沟通，技术的交流，在社区合作上做一些事情。

有些同学说，我只是用一下，好像没法做什么贡献。其实，"用"是一个很重要的合作，你能够用，你就会遇到问题，有你的诉求，遇到什么样的bug，有什么样需求没有满足。这些对我们来说，是非常重要的输入。在这一点上，欢迎和我们保持合作。

Sofa Mesh的开源宣言，写的比较狗血。但是在这一点上，我觉得这一次Sofa Mesh在开源上还是做的比较有诚意。

首先我们认可这个大方向，我们看好Service Mesh的前景。体现在什么上呢？我们现在规划，未来整个蚂蚁金服内部的大部分应用都会逐渐的往Service Mesh上落。这个内部已经达成一致了，会往这个方向走。

第二是说，"勇敢探索"，"耐心填坑"，有在1.0版本之前用过大型开源产品的同学，对这两个词都应该有深刻体验，对吧？包括前两年用0.*版本和1.1/1.2版本的k8s的同学。任何一个新的技术，一个大的方案出来，前期的时候，这些事情是一定会遇到的。但是我们觉得还是要去趟这个事情。

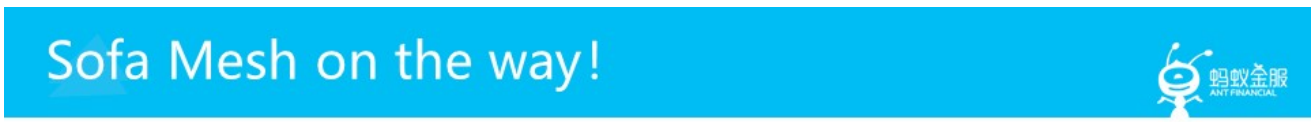
我们要继续推进这样一个技术进步，包括Service Mesh技术社区的推广。大家如果有注意的话说，Service Mesh技术社区已经重新启动了，我们在跟很多的公司，包括甚至我们一些竞争对手合作。从技术进步的角度说，我们欢迎大家在一个公平的基础上做技术交流。

然后我们是愿意做分享的，整个产品，我们接下来所有能开源的东西都会开源出来。除了一些内部定制化的东西，内部没有开源的产品的集成。基本上，你们能看到的东西，也就是我们内部用的东西。

我们寻求和大家的合作，包括刚才讲过的各个层面的合作，哪怕是简单的使用，发现问题给我们提交一些bug，也是非常好的合作契机。

这里我喊一个口号，这个口号有点大，"集结中国力量，共建开源精品"。这里面有个词，比较大一点，我也斟酌了一下，中国这两个字敢不敢用。最后我觉得还是用吧，至少到目前为止，Service Mesh这个技术领域，在全世界目前都还没有成熟的场景落地的情况下，我们目前在这方面的探索，已经是走在最前面的了。

在这一点上，我们是希望能联合国内在这个领域做探索的同学，我们一起来做这个事情。我们开源的一个重要目的，是说不管大家在商业上有什么样的竞争，至少在技术领域上，包括刚才说的可以在类库层面，产品层面，或者社区合作方面，开展合作。我们希望能够尽可能的联合国内的合作伙伴，包括竞争对手一起来营造整个技术氛围，把整个Service Mesh技术体系的基本水准提升上来。



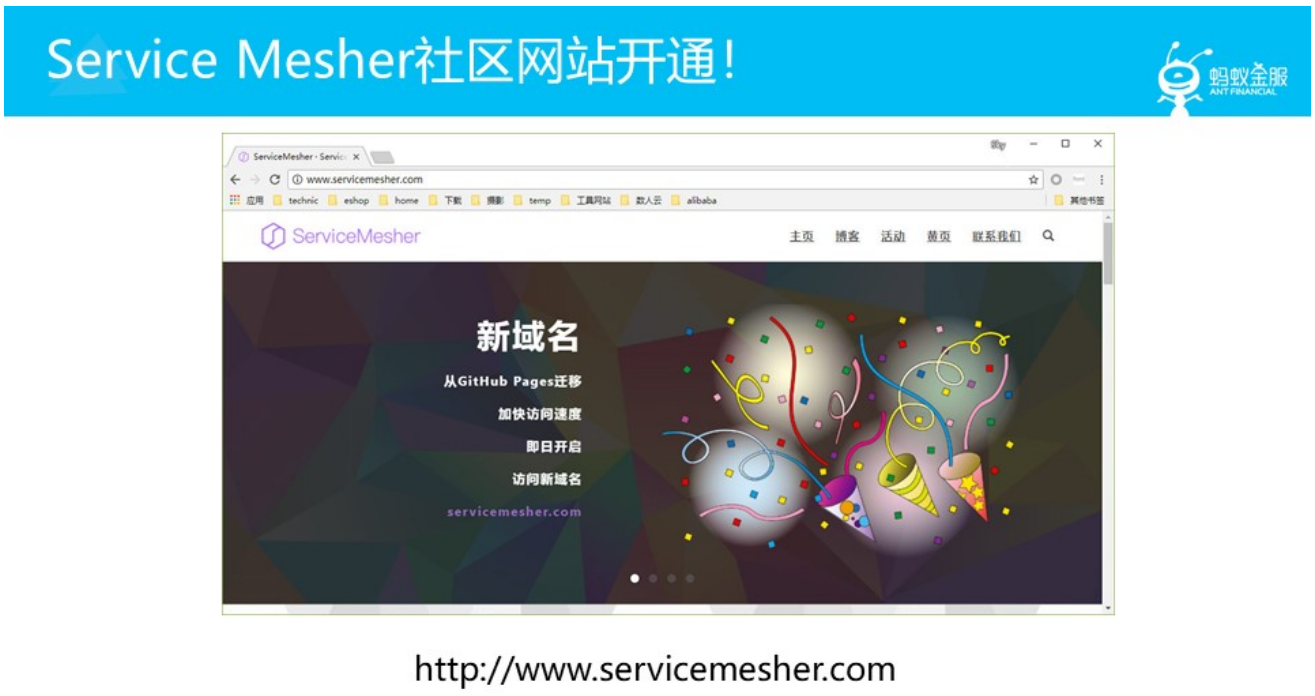
开源准备中，七月，github见!

这一点应该是大家比较关注的，什么时候开源? 我们只能告诉大家说，on the way，正在路上。

本来这一页的写法应该是贴个地址给大家的，但是因为进度的原因还没有实现，有可能会在一到两个星期之后，在7月份的时候开源给大家。

需要澄清的一点，大家的期望值不要太高，因为我们开源出来的第一个版本，主要是释放姿态，把我们的开源共建的姿态释放出来。我们的第一个版本，肯定不是一个完善的版本。（备注：现场有同学问，有在用吗？）内部有用一部分，Sidecar内部已经在用了，但是第二部分的内容，比如说XDS API的集成，我们现在正在做。我们不希望等把产品做完善了，比如说两年之后非常成熟的情况下再来开源。我们希望尽可能早的开源。

（备注：现场提问，7月份的版本，不一定是生产环境可用？）对，是的。有一部分功能是生产可用的，有一部分功能不是，因为我们是迭代上去的。



http://www.servicemesher.com

这是我们刚刚开通的Service Mesh技术社区的官方网站。

微信公众号

微信交流群

加入方式：请添加净超为好友，注明 servicemesher

这个是我们的微信公众号和微信交流群，欢迎关注，欢迎交流。

谢谢大家！