17 | 如何设计一锤子买卖的 SDK？

在前三个模块里，我将微服务根据**目的性**划分为三大类：读、写与扣减类，并针对每一大类涉及的各项技术问题讲解了应对方案。其实，**每一类微服务除了本身业务特点涉及的技术问题外，在纯技术维度也有很多共性问题，比如 SDK 如何设计、服务如何部署等。**

本模块将针对上述微服务中的**共性技术问题**进行深入讲解，首先咱们先来讨论微服务对外的门面——对外接口的 SDK 如何设计。

微服务骨架全局观

参考维基百科对于微服务的定义：微服务是指通过技术语言无关的协议（如 HTTP、ProtoBuf等）向外提供业务服务（常以接口的形式）的独立进程。它具有规模小、支持异步消息通信、可独立部署，以及可实现构建和分发自动化的特点。

结合上面的描述和前几模块里讲解过的架构图，便可以得到一个微服务所包含的内容，具体由以下 6 个部分组成：

1. 对外暴露的接口，由它直接对外提供各类业务服务能力；
2. 消费其他微服务发送过来的消息；
3. 可独立部署的微服务的代码；
4. 微服务持久化数据所**依赖的**数据库、缓存等存储；
5. 微服务完成一项业务能力需要依赖的其他微服务，比如提供提单的微服务就需要依赖库存微服务的扣减接口；
6. 微服务对外也会发送消息，来完成微服务间除接口以外的通信。

基于上述介绍的微服务中涉及的几大组件，可以将它们进行归类，梳理出如下图 1 所示的架构：

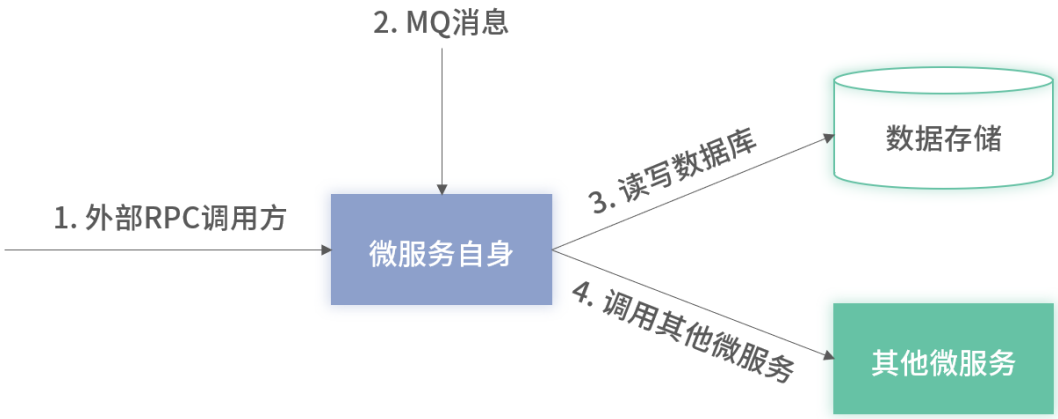

图 1：微服务骨架图

我将上述微服务里提到的 6 个组件分为三大类：

1. 第一类为对外提供的接口和接收的外部消息，称为**上游**；
2. 第二类为**微服务本身**；
3. 第三类为微服务依赖的其他组件，称为**下游**（如存储、其他微服务接口等）。

把这个分类和前几模块里提及的各种架构图对比，你会发现它们都包含上述三大类中的全部或部分内容，比如有的微服务依赖存储，有的依赖其他微服务。

在构建高可用微服务时，可以从上述三大类微服务进行入手。在本讲以及后续的两讲，我将按此思路讲解微服务对上游、自身以及下游（外部依赖）如何进行设计，以便构建一个更加健壮的微服务。

在详细讲解之前，我先剧透下构建微服务这三大类部分的高效法则（技术顺口溜）：**防备上游、做好自己、怀疑下游**。

关于上述三个口号表达的意思，以及它们的由来，我将在下面三讲慢慢道来。

为什么说 SDK 是一锤子买卖

微服务对外是以**接口形式**提供服务的，当接口开发完成上线，运行一段时间之后，形成的全局架构如下图 2 所示：

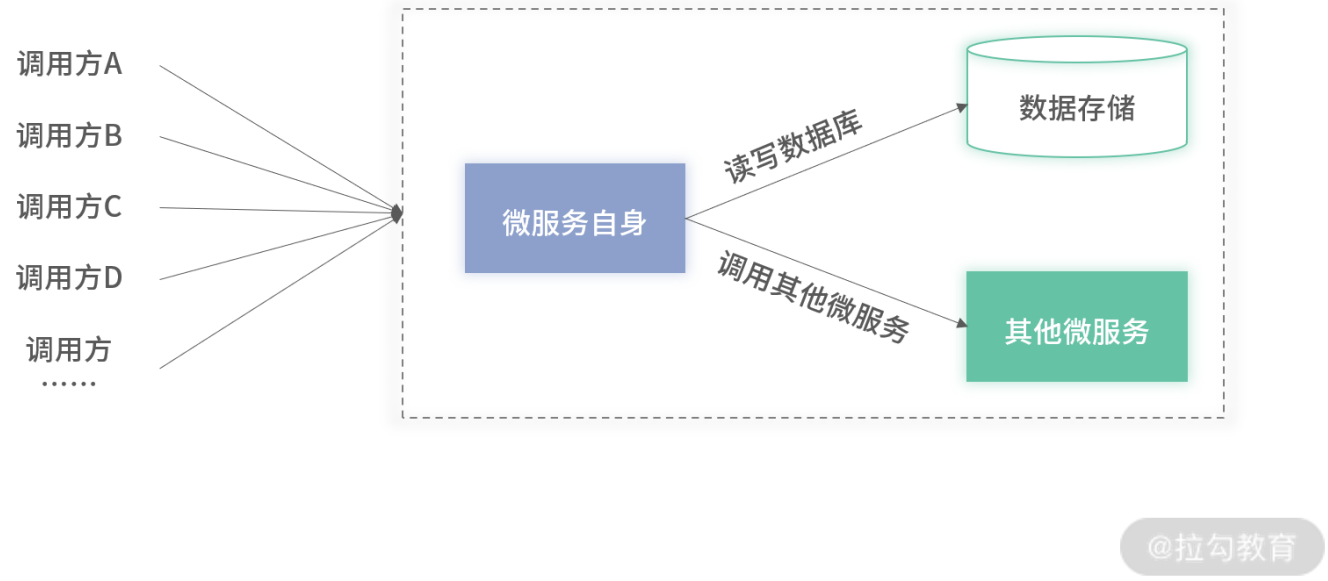

图 2：全局架构图

从上述的架构图里可以看到，接口上线后外部使用方会不断增多。假设上述外部调用方使用的某一个接口里的某一个方法的格式如下：

```
void func(long args1,int args2)
```

此时，因为新的业务需求，你需要**对该接口的上述方法的名称和入参数量进行变更**。修改后的格式如下：

```
void func_new(long args1,int args2,long args3)
```

如果要改成上述格式，你不能直接升级，因为上述两个格式不兼容。如果你先上线，所有的调用方都会报错，因为接口名和方法个数都变了。上述接口你需要提供灰度过程，大致如下：

1. 在微服务里同时提供上述两个接口；
2. **推动**所有的外部调用方切换到修改后的接口；
3. 确认老接口没有调用量后，方可将老接口下线。

从表面来看，只需要三个步骤就可以完成灰度发布过程，但上述第二步操作所需的时间远远超乎你的想象。如果调用方很多，推动所有调用方完成切换的时间短则几周，长则需要半年或者更久，成本非常高。

所以，**定义新的接口时需要考虑未来兼容性，如果接口上线后再想要修改，则需要花费较高的成本**。因此，包含一个微服务所有对外接口的 SDK 是一锤子买卖，设计时需要考虑清楚。

如何设计稳固的 SDK

因为 SDK 一旦上线后，修改成本会非常高。因此在设计 SDK 时，有一些基本原则建议你遵守，减少上线后的维护成本。

第一个原则：增加接口调用鉴权

当微服务对外提供的接口上线后，理论上所有需要此接口功能的使用方都可以随意调用此接口，微服务的提供方不应该设计鉴权等手段限制调用方的使用。

考虑如下场景后，可能你的想法会稍微改变。

1. 你的接口当前能够支持的最大 QPS 为 1W，而新的调用方会带来每秒 10W 的 QPS。如果这个新的调用方在你还没有完成扩容前，就直接上线，导致的结果可能是你的微服务被瞬间打挂。
2. 接口的入参有一个 Map 字段，文档未有明确标注，但实际此 Map 字段最大支持 100 个 Key 的设置，如果超过 100 个 Key 就会报错。因为没有调用前的申请审批，新接入的调用方的场景里有可能会传入 150 个 Key，导致的结果是直接报错，进而可能产生线上问题。
3. SDK 提供了查询和写入的接口，但查询的接口是基于缓存或 ElasticSearch 实现的，是有毫秒级延迟的。而使用方期望写入后，通过查询接口可以立马查询到数据。如果新的接入方没有前置的审批沟通，直接接入后，会发现接口和预期并不一致，可能会使得此次接入变成无用功，导致成本浪费。

通过增加鉴权，所有的调用方在使用前都需要申请接口调用的权限，在申请的过程中，你可以针对上述提到的问题和调用方一一进行确认，防止出现意外的情况。

第二个原则：接口里的入参需要是对象类型，而不是原子类型

原子类型是指非面向对象里的类，在里面不能再定义字段的类型。比如编程语言里的 int、long、float 等类型。

对象类型是指面向对象里的类，比如如下格式：

```
class ObjectA{
  private long args1;
  private int args2;
}
```

对象类型的好处是当有新的需求时，可以在其中新增字段，而不是修改接口的签名。

在上一小节介绍了 SDK 是一锤子买卖的示例，如果原始接口定义的是如下格式：

```
void func_new(ObjectA object1)
```

当一个新的需求需要在入参增加 args3 字段时，便可以直接在 ObjectA 这个类里添加，而不是修改接口的签名。这样设计的好处是**向后兼容**，只有此次新需求需要使用 args3 字段的调用方才需要升级，而不关心此字段的历史调用方都不需要升级，可以节约推动外部所有客户升级的时间。

第三个原则：接口的出入参不要设计为 Map<String,String> 等集合格式

出入参使用了 Map 格式的设计如下：

```
Map<String,String> func_new(Map<String,String> args);
```

这样设计的好处是**特别灵活**，当接口在日常的升级中需要新增一个字段，如第二个原则里提到的，新增 args3 字段时，整个接口都不需要做任何更改。因为 Map 的 Key 是动态的，可以随意由外部客户传入的。

虽然这样设计有灵活性的优势，但劣势也比较明显。

1. 首先，代码非常难维护。因为 Map 里的 Key 是动态的且是文本的，要识别这些 Key，你需要在代码里使用魔术数或者硬编码进行识别。随着时间的流逝，这种方式会导致代码里随处可见的硬编码，代码阅读起来非常不直观。
2. 其次，Map 的方式是动态，理论上调用方可以往 Map 中插入成百上千的数据。极端情况下，这些数据会把微服务的内存瞬间打挂，对系统的稳定性影响非常大。

第四个原则：入参需要增加条件限制和参数校验

可以分别对读和写接口进行分析。

首先，对于对外暴露的写接口，如果不增加参数校验，可能会导致后续业务无法正常流转。

1. 外部调用方可以传入超过数据库长度限制的参数，有些数据库会直接拦截，并生成数据超长的错误，而有些数据库可能会默认地将数据截断并存储。
2. 对于如手机号、邮箱地址等自带业务格式的数据，如果不做格式拦截，将不符合格式的数据写入数据库之后，后续的业务可能无法流转。比如订单里的收货人的手机号码，如果写错，可能导致订单无法正常配送。

其次，对于对外暴露的读接口，如果不增加参数校验，可能会把数据库打挂。

1. 如果你提供的一个翻页查询功能，常见的查询是使用数据库的"**limit startIndex,size order by xx 字段**"来进行实现的。如果你不进行参数验证，理论上调用方可以传入值为 100000 的 startIndex。实际上，随着 startIndex 的增大，limit 的性能会非常差，极端情况下，如果量太大，数据库很容易挂。
2. 如果你提供了如 like 等模糊匹配功能，如果外部传入一些正则表达式里非常耗费性能的语法，也是有可能把数据库打挂的。

第五个原则：写接口需要保证幂等性

考虑一种场景，如果外部客户调用你的接口超时，它能如何处理?

答案是：**只能进行重试或者反查**，不然别无他法。

因为超时后，调用方并不知道此次写入是否成功，有可能成功，也有可能不成功。通过反查调用方可以确定此次调用是否成功；通过重试，调用方期望你告诉它，上次写入已经成功，无须重试。

上述的反查和重试，技术上称为**幂等性**。写接口的幂等可以在入参增加一个当次调用的全局唯一标识来实现，同时该唯一标识需要写入数据库中，并在数据库里将该字段设置为唯一索引即可。架构如下图 3 所示：

图 3：写接口幂等性架构

通过上述的架构，当超时后，调用方对于当次调用再次重试时，如果前一次超时的请求已经写入成功，那么数据库的唯一索引会对重试请求进行拦截，并提示唯一索引冲突，无法写入。此外，如果调用方选择反查而不是重试，它也可以使用唯一标识进行反查，如果上一次超时的请求已写入成功，反查也能够查询到数据。

关于如何生成全局唯一标识，可以参考“08 | 如何使用分库分表支持海量数据的写入”里介绍的几种方法。

第六个原则：接口返回的结果需要统一，可以直接抛出异常或者使用结果包装类（如 RPCResult）

对外的 SDK 会包含一组接口，这些接口对外返回的格式需要保持统一，要么全是正常业务对象+异常的格式，要么全是通过 RPCResult 包装业务对象的格式。这两个格式没有绝对的优劣之分，但统一的格式有利于调用方统一处理，两种格式混合的方式会增加调用方的处理成本。

显式抛出异常的格式如下：

```
Object func_new(Object args1) throws RPCException
```

其中 RPCException 中需要包含如下字段：

```
Class RPCException{
  private boolean success;//是否成功
  private int code; //如果错误，详细的错误码
  private String msg;
}
```

使用 RPCResult 包装类的格式如下：

```
RPCResult<Object> func_new(Object args1)
```

其中 RPCResult 中需要包含的字段和上述 RPCException 需要包含的格式基本一样，此处不再赘述。

可以看出，这两种方式中包含的错误信息基本一致。**唯一的区别是：异常的方式除了会包含上述信息外，也会包含一些报错的堆栈信息**，如下格式：

```
"thread name" prio=0 tid=0x0 nid=0x0 runnable
at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
at java.net.SocketInputStream.read(SocketInputStream.java:171)
```

因为 RPCException 里已经包含当次请求是否错误，以及导致错误的详细原因，即其中的错误码（code 字段），此外异常的堆栈信息是为了方便微服务的提供方进行问题排查，调用方无须关心，因此，在实际开发中，你可以显式地把 RPC 中抛出异常的堆栈信息屏蔽掉。现在主流的编程语言均已提供上述功能。

最后，不管是 RPCException 还是 RPCResult 里都包含的错误码，即 code 字段，这样做是为了方便调用方能够快速知道导致出错的具体原因，进而根据不同的原因做相对应的处理。比如在有些情况下：

1. 调用方传入的参数不合法，如电话号码传入了字符，导致检验不通过；
2. 微服务提供方依赖的存储故障，如缓存、数据库等宕机等，进而导致当次调用产生错误。

当出现上述两种错误时，对应的处理方式是不一样的。

1. 如果是传入的参数格式错误了，你需要提示客户修改格式重新输入，而不需要联系此微服务的提供方进行处理。
2. 如果是上述第二种错误，你需要立马通知微服务提供方，让对方尽快修复故障，因为下游出现错误，你能做的便是尽快通知。

在实际实践中，使用 RPCException 还是 RPCResult 其实都可以，只要保持统一即可。不过不管格式如何，上述两个对象都需要包含上述字段。

第七个原则：返回的数据量必须要分页

如果存在以下格式的接口定义，它表示此接口的功能是返回一批数据：

```
List<String> func_new(Object args1);
```

如果接口的入参里没有显式地设置当次查询数据的具体数量，假设当次查询条件命中的数据量非常多，那么一次返回的数据量就会非常多，可能达到上千 KB 或者上百 MB 的数据。

上述这个批量获取数据的接口，如果不分页，会存在以下两个问题：

1. 首先，获取这么大量数据的查询条件，在查询的时候，可能会把数据库或缓存打挂；
2. 其次，数据量越多，网络传输的时间也越长，直接的体现就是接口的性能非常差。

因此，建议所有对外批量接口都增加分页，而不是一次吐出所有数据。这样既可以提升稳定性、又可以提升性能。

第八个原则：所有的接口需要根据接口能力前置设置限流

最后，即使经过上述的几个步骤后，仍有可能一个通过鉴权审批后的调用方，它的系统在某一个时间点出现故障，或者因为一些热门活动导致流量出现飙升，假如这个突发流量超过你的微服务的最大承载量，即使遵循了上述的第一个原则：调用前的鉴权，也无法限制通过鉴权后的调用方带来的突发流量。

对于可能产生的异常流量，可以使用在“16 | 秒杀场景：热点扣减如何保证命中的存储分片不挂？”里提到的前置限流策略来预防。

消息的消费

消息消费指的是你的微服务接受其他微服务发送的消息的场景，在实践中梳理时，此方式的高可用较容易忽略。其实，此种方式和上一小节里的接口方式非常类似，只是**消息是异步的形式**。它和微服务间的同步调用架构如下图 4 所示：

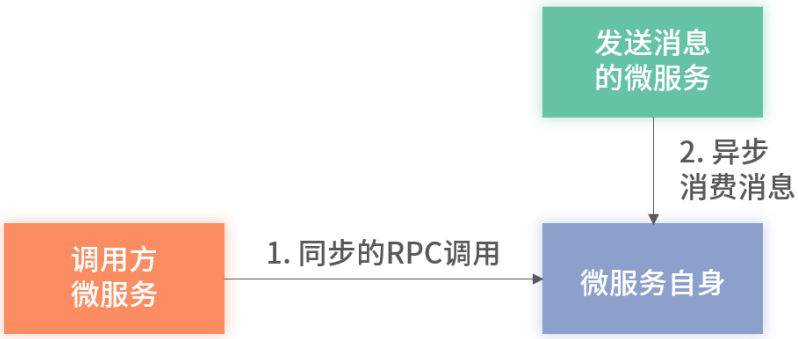

图 4：消息异步消费和接口间同步架构的对比

如果消息消费和接口调用相类似，那么上述接口里的一些原则在消息里依然可以复用，可以参考以下内容。

1. 消息消费需要有前置限流。当消息发送方发送量暴增时，限流可以保证消息消费服务的稳定。
2. 对于消息消费需要保证幂等，不然当消息出现重试后，会出现业务上的脏数据。
3. 消息的数据在消费处理时需要进行前置参数检验。如果未做前置参数校验，同样也有可能写入一些不合法的脏数据。

总结

在本讲里，梳理了如何设计对外 SDK 里的接口的原则，以及如何设计和它架构上相类似的消息消费的原则。在实际工作中，你可以通过这些原则，构建一个更加高可用和兼具兼容性的系统。

你应该还记得，在本讲的开头我直接给出了对外接口的设计准则：**防备上游**。通过本讲介绍的 SDK 的几个落地的细节手段便可以看出原因，它们都是对上游调用方进行鉴权、限流、入参前置校验与拦截，这些都属于防备外部调用的具体手段。

因此，**防备上游**是对外接口设计的基本准则。

最后，留一道思考题。你们团队在微服务对外接口里还有那些准则？可以在留言区和大家一起分享。