目录

目录
1.简介
2.架构总览
3.系统设计
- 3.1架构分层
  - 3.1.1.什么是分层架构?
  - 3.1.2.分层有什么好处?
  - 3.1.3.如何来做系统分层?
  - 3.1.4.分层架构不足
- 3.2高性能
  - 3.2.1.提高系统核心处理数
  - 3.2.2.减少单次任务响应时间
- 3.3高可用
  - 3.3.1.**可用性度量**
  - 3.3.2.系统设计
  - 3.3.3.**系统运维**
- 3.4高拓展
  - 3.4.1.复杂性
  - 3.4.2.设计思路
    - **1.拆分**
    - **2.存储层**
    - 3.业务层

4.复杂度分析
- 4.1.简介
- 4.2.架构演进
  - 4.2.1.单体到服务化演进
  - 4.2.2.服务化到微服务演进
  - 4.2.3.分布式与微服务
- 4.3.分布式一致性
  - 4.3.1. CAP & Base理论
  - 4.3.2.重试
  - 4.3.3.分布式事务
    - 1.2pc-xa协议
    - 2.3pc协议
    - 3.Tcc协议
  - 4.3.4.事务管理器
    - 1.本地事务管理器
    - 2.外部事务管理器
  - 4.3.5.MQ与DB一致性
  - 4.3.6.DB与缓存一致性
  - 4.3.7.兜底核对
    - 1.自核对
    - 2.搭建核对系统
  - 4.3.8.Saga机制
- 4.4.可观测性理论
  - 4.4.1.定义
  - 4.4.2.价值
  - 4.4.3.三大支柱
- 4.5.基于调用链的服务治理

5.基础设施

5.1.后台任务
1.Hangfire
2.Rabbitmq
3.Quartz
5.2.事件总线
5.2.1.全局配置
5.2.2.服务注入
2.1 路由模式
2.2 主题模式
5.2.3.事件模型
5.2.4.消息订阅
5.2.5.消息推送
5.1 路由模式
5.2 主题模式
5.2.6.业务处理
5.3.数据仓储
5.3.1.全局配置
5.3.2.服务注册
5.3.3.仓储实现
1.实体&聚合根
2.仓储
3.数据传输对象（Dto）
4.工作单元
5.应用服务
5.4.配置中心
1.本地配置
2.分布式配置中心
5.5.服务发现
5.5.1.注册中心Consul
1.全局配置
2.服务注入
5.5.2.基于服务发现的ApI网关Ocelot
1.全局配置
2.服务注册
3.Handle重写
5.5.3.服务网格ServiceMesh
5.5.4.网关限流/熔断/降级/会话保持/黑名单
1.限流
2.熔断
3.降级
4.会话保持
5.黑名单
5.6.链路追踪
5.7.审计日志
1.本地审计
1.启用审计日志服务
2.Es采集审计
1.启用
5.8.依赖注入
5.8.1.Ioc
5.8.2.AutoFac
1.容器声明
2.获取当前作用域
5.8.3.Location Ioc
5.8.4.全局注入

1.启用
5.8.5.反射
1.启用
2.自定义反射
5.9.缓存
1.启用
2.启用布隆过滤器
5.10.限流
1.配置
2.启用
5.11.Map
5.12.Rpc
5.12.1.ProtoFile声明
5.12.2.服务端
5.12.3.客户端
5.12.4.注入
5.12.5.明文
5.13.Orm
5.13.1.Dapper
1.全局配置
2.启用
3.配置分表策略
4.启用自动分表
5.使用
5.13.2.SqlSugar
1.配置
2.注册
3.使用
6.演进规划

1.简介

文档读者可面向开发，测试，产品，架构师人员

从面向对象，面向服务理论，设计思想，技术实现，市场前沿技术方案选型等角度

进行云原生场景下分布式微服务应用实现及方向演变方案论述，并借`GlasssixBasicProject`项目做了文字描述及演示

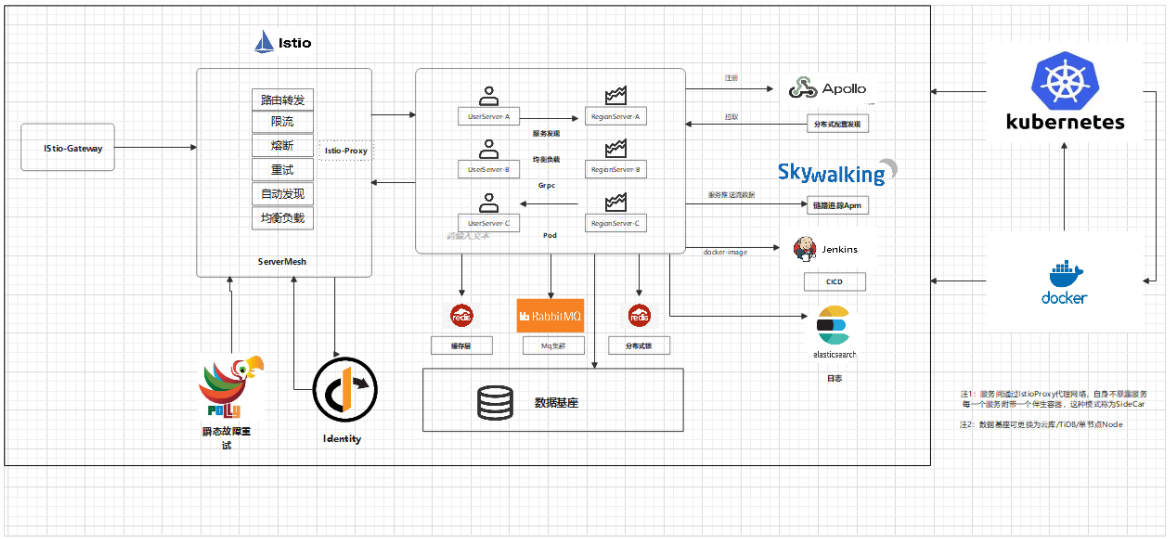
以下包括但不限于：

- 服务历程
- 设计思想
- 框架实现
- 领域划分
- 基础设施
- 场景化三高
- 内部复杂性
- 常驻问题等解决方案

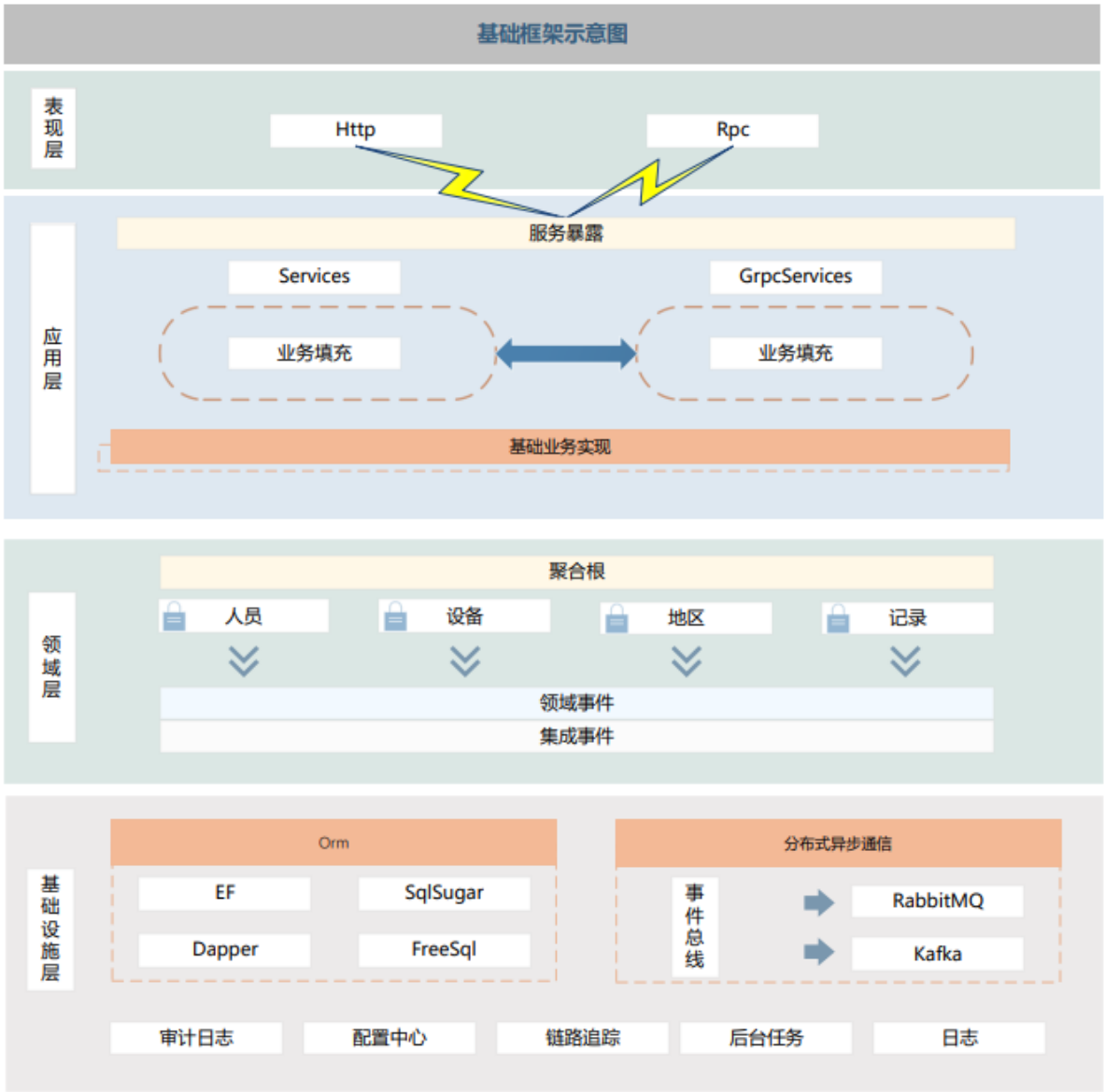
以及设计模式及服务思想方法论

2.架构总览

宏观



微观



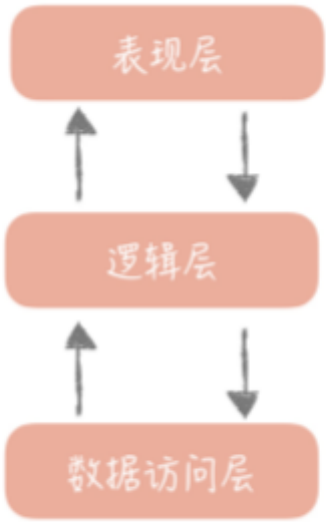
3.系统设计

3.1架构分层

3.1.1.什么是分层架构?

传统单体应用上，系统可被简单划分为`逻辑层->数据层->表现层`

例如：`MVC`



三层架构示意图

简单点说，软件架构分层是一种常见的设计方式，它是将整体系统拆分成 N 个层次，每 个层次有独立的职责，多个层次协同提供完整的功能

3.1.2.分层有什么好处?

1.分层可以简化软件设计，使不同的人专心不同的事

2.分层带来系统强解耦，系统业务变清晰

3.分层提高物理资源利用率，例如单体下需要一台`8u32g`服务器，利用率80%，分层下各服务只需`2u4g`，利用率>90%

4.带来很高的复用性，利于技术人员做横向拓展

3.1.3.如何来做系统分层?

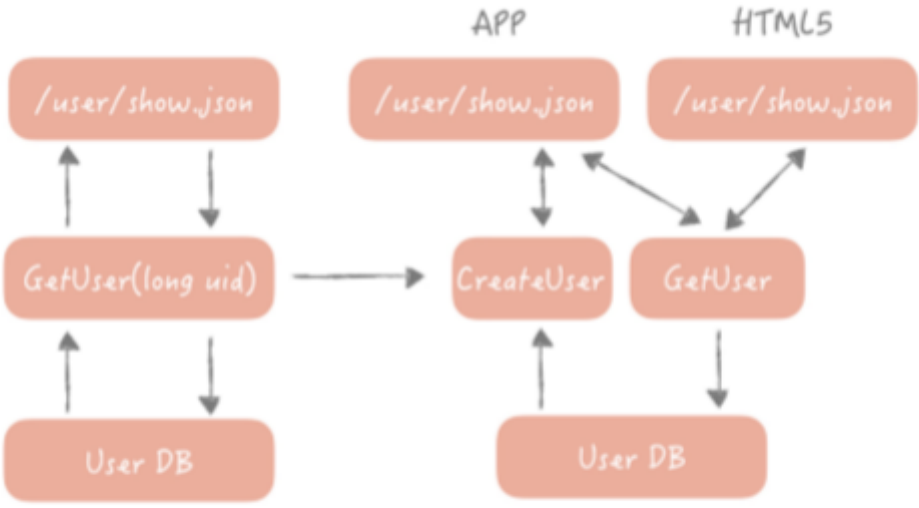
那么我们如何去正确进行分层呢，在我看来：

我们应当理清系统的边界，`mvc`模式下，边界确实更容易辨识，可当我们的系统越来越复杂时，系统的边界就会变的模糊。

那举个例子，用户系统中最基本的返回信息接口`getUser`，`getUser`方法调用`uerDB`交互数据,

这时提出一个新需求，页面获取新用户时如果用户不存在那么将创建一个新用户，同时保留以前的逻辑，这时逻辑层的边界将变的不在清晰，表现层

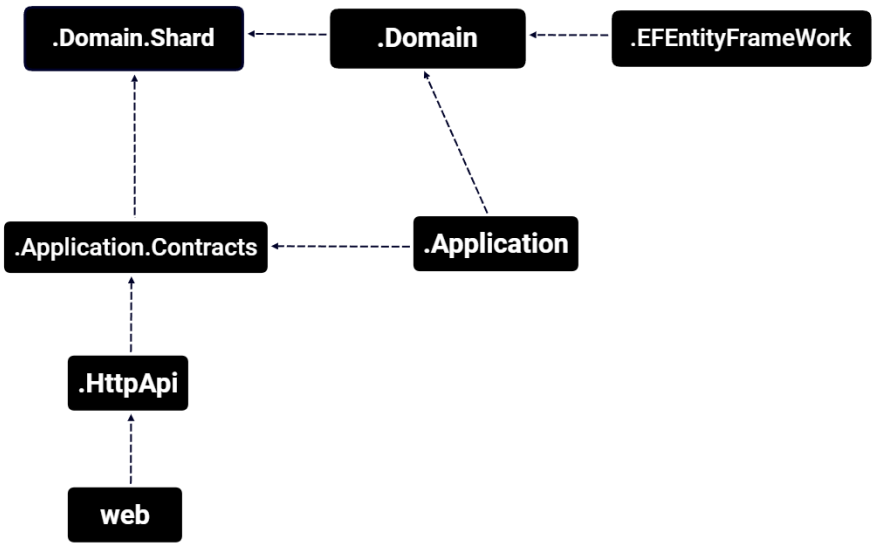
承担了一部分业务逻辑。



获取用户信息接口的进化

那我们如何做，基于领域驱动 DDD 实践的分层应用程序结构

可将应用拆分为：



.Domain.Shared

该项目包含常量、枚举和其他对象，这些实际上是领域层的一部分，但需要由解决方案中的所有层/项目使用

参考项目User服务中`UserInput`类中可能会含有User中的常量字段

.Domain

这是解决方案的领域层。它主要包含**实体**、**聚合根**、**域服务**、**值对象**、**存储库接口**和**其他域对象**。

.**Application.Contracts**

本项目主要包含应用层的应用服务 **接口**和数据传输对象（`DTO`）。它存在分离应用层的接口和实现。这样，接口项目就可以作为合约包共享给客户端

.**Application**

该项目包含项目中定义的接口的应用服务**实现** `.Application.Contracts`

参考项目中`UserService`类

- 取决于`.Application.Contracts`项目是否能够实现接口和使用 `DTO`
- 取决于`.Domain`项目是否能够使用域对象（实体、存储库接口.....等）来执行应用程序逻辑

.**EntityFrameworkCore**

这是 `EF Core` 的集成项目。它定义`DbContext`并实现了`.Domain`项目中定义的存储库接口

- 取决于`.Domain`项目是否能够引用实体和存储库接口

.**HttpApi**

该项目用于定义您的 `API` 控制器

- 取决于`.Application.Contracts`项目是否能够注入应用程序服务接口

3.1.4.分层架构不足

一个成熟的系统中，解决一个问题势必会引出另一个问题，正如此，没有完美的系统，分层后带来的最大问题就是整个代码变的极具复杂性

阅读起来晦涩，编写麻烦

这是显然的，本来请求进来直接过逻辑去取库数据，非要在中间加一堆处理，各种封装验证，增加了开发的成本。

分布式下，分层后网络请求会在系统内部产生自我损耗，也就是多一跳。

那我们具体要不要分层，肯定是的，正如没有完美的系统，但是每个问题都对应着每个解决方案，不可以偏概全，只需找到此中平衡点即可

分层只是软件系统架构的宏观体现，是一种实现方式。微观下系统其中有体现，比如单一职责，开放封闭...等等

合理运用并掌握能帮我们牢牢掌握此种方式

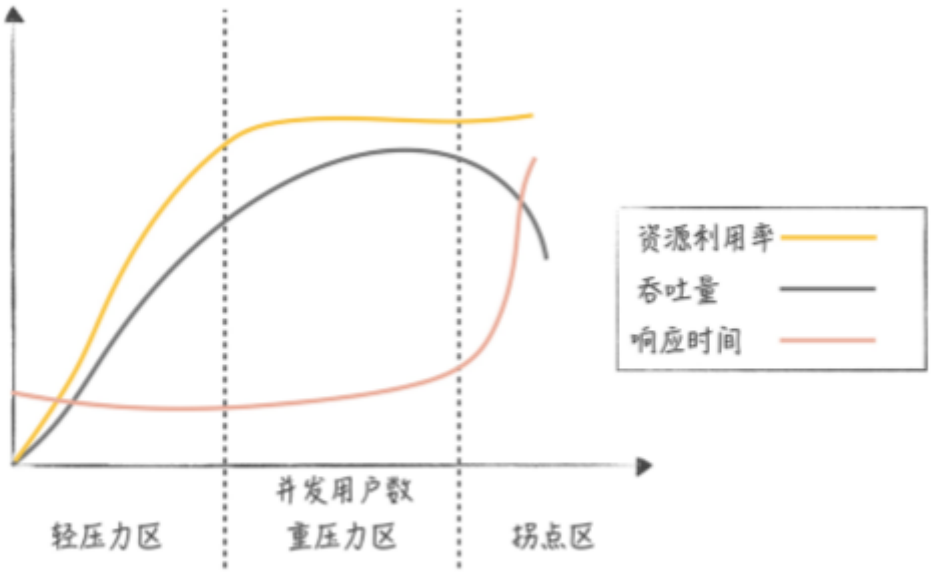
3.2高性能

3.2.1.提高系统核心处理数

提高系统核心处理数就是增加系统的并发能力，比如增加一个U，再增加一个进程，让这两个线程跑在两个不同的核上，从理论上来讲，系统吞吐量可以

提升一倍，但是此时衡量系统吞吐量的方式也就变了,吞吐量=并发进程数/响应时间，此处感兴趣的可以去看看**阿姆达尔定律**，

那么也就是说，并不是一昧增加系统核心数量就可以提高吞吐量，存在一个拐点：



性能测试的拐点模型

从图上可以看出，出于轻压力区时，响应时间平缓，吞吐量递增，资源利用率逐步攀升，此时处于系统健康状态

处于重压力区时，资源利用率到达顶峰开始出现比较平缓的趋势，吞吐量逐步攀升达到顶峰，响应时间缓慢增长，此时系统处于也是处于健康状态下

到达拐点区时，吞吐量不增返降，响应时间却是在逐步攀升

所以，我们在评估系统时通常要进行压力测试从而找出拐点，处于拐点前才能达到性能优化效果，否则适得其反

3.2.2.减少单次任务响应时间

想要减少任务的响应时间，首先要看系统是`Cpu`密集型的还是IO密集型的，针对两种形式，具有两种解决手段

`cpu`密集型

这种类型中，需要处理大量的`Cpu`运算，那么通过编写更高效的方法或算法就可以减轻问题症状，比如系统在计算Hash算法很慢，那么通过编写更高效的算法就可以大大提升系统的性能，发现这种问题通常是要借助一些第三方`Profile`工具，比如Linux的`Perf`等。

`IO`密集型

此种类型比较好排查，IO的问题往往会伴随着很明显的症状，比如内存异常久高不下，写数据异常等等，当然也有可能是依赖系统的问题，mysql问题，缓存问题等，通常也比较容易解决

1.采用工具，一般这些工具会带有丰富工具集来帮助你从各方面做彻底的排查，当然也有一些语言特性工具，可以集成在系统内部

2.采用监控，根据可观测理论，分布式微服务下，应抓住每一个数据，信息，步骤来进行统计分析，如果系统做了监控，那么 可以根据某个步骤响应时间来进行追溯，找到问题点,找到问题后，优化方案也随着问题的不同而不同，如果是数据库访问慢，那就看看是不是锁表了，索引加的合不合适，是不是进行全盘扫描了，有无join操作，如果是网络问题，那就看看参数是否有优化的空间，是否存在丢包问题等等

综上所诉，我们在实际中总会遇到各种各样的问题，总会是有解决方案，总之，**兵来将挡水来土掩。**

3.3高可用

高可用性，是我们常听到的一个词，它指的是系统的无故障运行能力

我们在很多开源组件中看到的Ha模式就是其中一种，`HaProxy`+`Keepalived`来扩充节点达到其高可用性，让系统免于宕机而无法服务,例如`Consul集群`的选举模式，`redis集群`的主从，`mysql集群`的主从模式都属于可用性中一种**分布式一致性算法**,具体基于分布式一致性算法这块感兴趣的大家可以线下去了解

通常来讲，一个并发比较高的系统，系统出现故障所受影响远比性能问题所带来的影响更严重，一个日活百万的系统，分钟内就能损伤上千用户使用体验,那如何解决，我们从多角度去围绕:

3.3.1.可用性度量

可用性是一抽象的概念，与之对应的有两个相关概念：`MTBF`,`MTTR`

`MTBF`（Mean Time Between Failure）是平均故障间隔的意思，代表两次故障的间隔时间，也就是系统正常运转的平均时间, 这个时间越长，系统稳定性越高

`MTTR`（Mean Time To Repair）表示故障的平均恢复时间，也可以理解为平均故障时 间。这个值越小，故障对于用户的影响越小

用一个公式来计算

Availability = `MTBF` **/ (**`MTBF` **+** `MTTR` **)**

系统可用性	年故障时间	日故障时间
90%（一个九）	36.5天	2.4小时
99%（两个九）	3.65天	14.4分
99.9%（三个九）	8小时	1.44分
99.99%（四个九）	52分钟	8.6秒
99.999%（五个九）	5分钟	0.86秒
99.9999%（六个九）	32秒	86毫秒

不同可用性标准下，允许的故障时间

图中可以发现

两个9之前，通常是比较容易达到的，只要不是故意搞破坏或者程序更新后的瞬态问题，基本上可以通过人肉运维来达到，简单暴力易达到

三个9之后，时间由天锐减为8小时，4个9更甚，这个时候要考虑的已经不单单是服务化可用，要建立一系列整套运维体系，发生故障是否能瞬时恢复

是否可以不需要人工干预，工具方面也要建立一系列排查工具及监控，以便出现问题时能快速定位排查

达到5个9乃至六个9后，已经不是人工干预能解决的掉的，这个时候考研的是系统的瞬时恢复，自愈，以及容灾备份，

让机器来处理故障，才能让可用性提高一个层次

3.3.2.系统设计

`Design for failure`是我们设计时要考虑的第一原则，译为：**设计时为故障做好准备**

高并发下，集群单节点故障会成为常态，未雨绸缪才能决胜千里，要把发生故障时情况作为一个考虑点，预先的发现故障并自动化处理，除了在物理层面

所作的处理，程序内部也要有预先定义

如上图`C#`类库`polly`就是为解决瞬态故障所诞生的，那么我们在进行一些不可预知估量的任务时，比如第三方组件的连接，`rpc`的超时等等,就可以将一系列策略纳入其中，例如**重试**，**熔断**，**降级**，**限流**等等

那我们再讲讲这些策略的定义：

`failover`故障转移更是其核心考虑之一，其表现形式为

1.在两个完全对等的node上发生

2.存在主备选举

这个可以很好理解，也就是常见的**Ha模式**及**选举模式**

目前强大的生态环境，`failover`不在由程序去定义，通过容器编排，`k8s`内部就能做到故障转移及自愈,对于我们来说无疑是友好的

只是我们在使用及享受成果的同时，尽量做到了解并理解

3.3.3.系统运维

经历上面一系列手段，我们的系统已经变得比较茁壮，可还是不能途然上线，我们还要经过运维角度的手段来提前预知到发生的错误，我们可以从两个角度来考虑：**故障演练**，**灰度发布**

故障演练指的是对系统进行一些破坏性的手段，观察在出现局部故障时，整体的系统表现是怎样的，从而发现系统中存在的，潜在的可用性问题，也就是我们常说的压力测试。

一个复杂的高并发系统依赖了太多的组件，比方说磁盘，数据库，网卡等，这些组件随时随地都可能会发生故障，而一旦它们发生故障，很可能会如蝴蝶效应一般造成整体服务不可用，因此故障演练尤为重要。

那什么是灰度发布？

灰度发布指的是系统的变更不是一次性地推到线上的，而是按照一定比例逐步推进的。一般 情况下，灰度发布是以机器维度进行的。

比方说，我们先在 10% 的机器上进行变更，同时 观察 Dashboard 上的系统性能指标以及错误日志。如果运行了一段时间之后系统指标比较平稳，

并且没有出现大量的错误日志，那么再推动全量变更。

总结

可以看到，从开发及运维角度看到的是不相同的：

开发注重的取舍及冗余，冗余指的是有备用节点，集群来顶替出故障的服务，比如文中提到的故障转移，重试，限流等等策略；取舍指的是丢卒保车，保障核心服务的健康。

运维角度来看则更偏保守，注重的是如何避免故障的发生，比如更关注变更管理以及如何做故障的演练。

3.4高拓展

从架构设计上来说，高可扩展性是一个设计的指标，它表示可以通过增加机器的方式来线性 提高系统的处理能力，从而承担更高的流量和并发。

那为什么我们在架构设计之初，不预先考虑好使用多少台机器，支持现有的并发呢？ 答案是峰值的流量不可控

一般来说，基于成本考虑，在业务平稳期，我们会预留 30% ~ 50% 的冗余以应对活动或者突发情况可能带来的峰值流量，但是当有一个突发事件发生时，流量可能瞬间提升到 2 ~ 3 倍甚至更高。

我们以西安一码通来讲，大家每天时时刻刻都会去请求二维码获取最新状态，并且是持续性高热型行为，在西安市民的共同努力下，一码通崩掉。 那我们要如何应对突发的流量呢？这个时候，架构的改造已经来不及了，最快的方式就是堆机器。不过 我们需要保证，扩容了三倍的机器之后，相应的我们的系统也能支撑三倍的流量。那有的人可能会问：这不是显而易见的吗？很简单啊。真的是这样吗？我们来看看做这件事儿难在哪儿

3.4.1.复杂性

在可用性中我们讲到，可以在单机系统中通过增加处理核心的方式，来增加系统的并行处理能 力，但这个方式并不总生效。因为当并行的任务数较多时，系统会因为争抢资源而达到性能 上的拐点，系统处理能力不升反降。

而对于由多台机器组成的集群系统来说也是如此。集群系统中不同的系统分层上可能存在 一些“瓶颈点”，这些瓶颈点制约着系统的横线扩展能力。这句话比较抽象

举个例子：

比方说，你系统的流量是每秒 1000 次请求，对数据库的请求量也是每秒 1000 次。如果流量增加 10 倍，虽然系统可以通过扩容正常服务，数据库却成了瓶颈。

再比方说，单机网络带宽是 `50Mbps`，那么如果扩容到 30 台机器，前端负载均衡的带宽就超过了千兆带宽的限制，也会成为瓶颈点。

其实无状态的服务和组件更易于扩展，而像 `MySQL` 这种存储服务是有状态的，就比较难以扩展。因为向存储集群中增加或者减少机器时，会涉及大量数据的迁移，而一般传统的关系型数据库都不支持。这就是为什么提升系统扩展性会很复杂的主要原因。

除此之外，从例子中你可以看到，我们需要站在整体架构的角度，而不仅仅是业务服务器的角度来考虑系统的扩展性 。所以说，数据库、缓存、依赖的第三方、负载均衡、带宽等等都是系统扩展时需要考虑的因素。我们要知道系统并发到了某一个量级之后，哪一个因素会成为我们的瓶颈点，再从而针对性地进行扩展

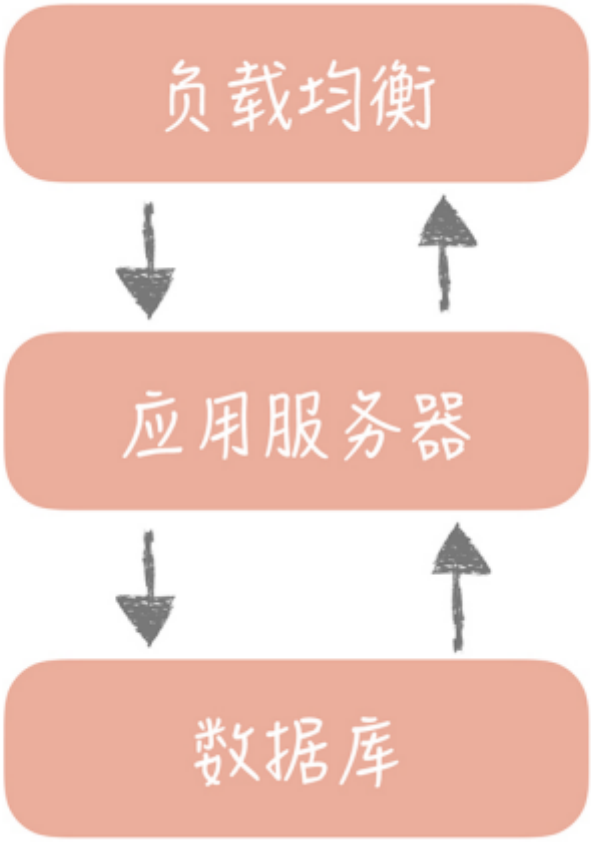
3.4.2.设计思路

1.拆分

拆分是提升系统扩展性最重要的一个思路，它会把庞杂的系统拆分成独立的，有单一职责的 模块。相对于大系统来说，考虑一个一个小模块的扩展性当然会简单一些，将复杂的问题简单化，这就是我们的思路但对于不同类型的模块，我们在拆分上遵循的原则是不一样的。

比如

如果部署方式遵照最简单的三层部署架构，负载均衡负责请求的分发，应用服务器负责业务逻辑的处理，数据库负责数据的存储落地。这时所有模块的业务代码都混合在一起了，数据也都存储在一个库

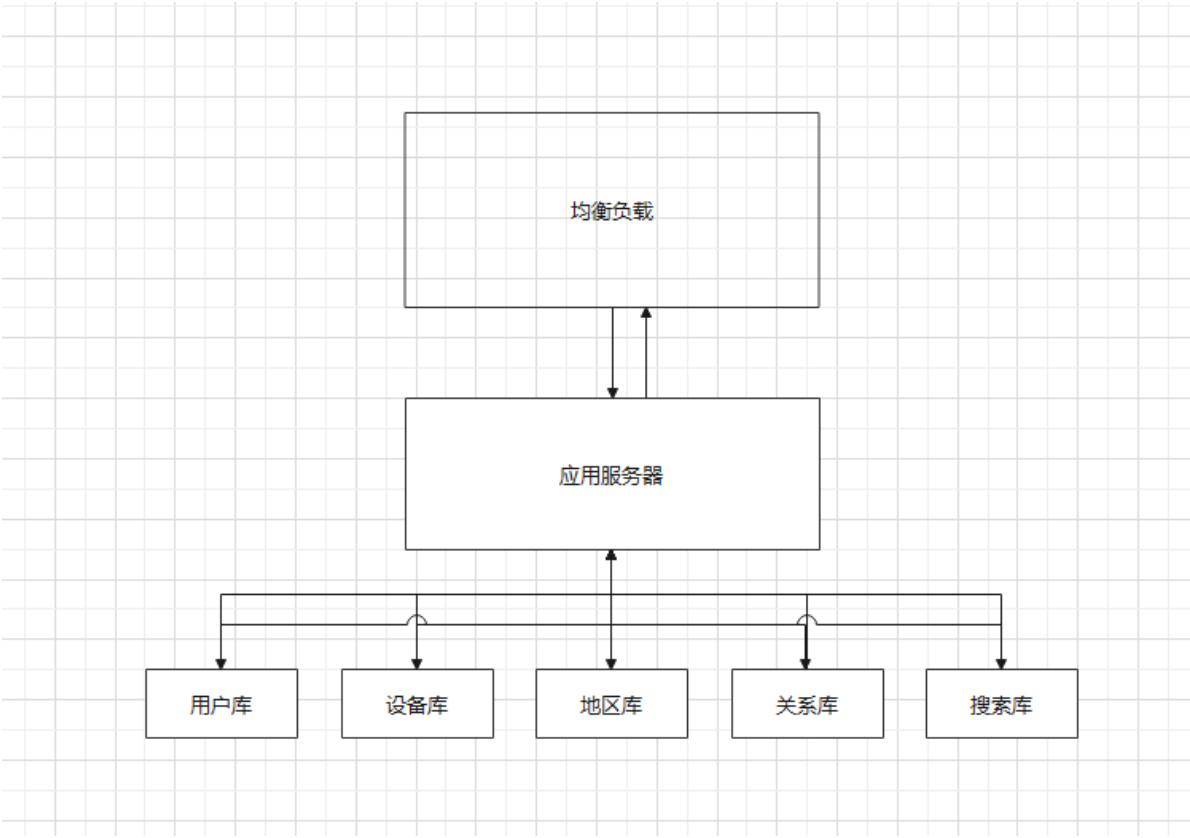


那么此时架构如图所示，会发现尽管应用服务器横向拓展后，而终会因数据库某个点到达瓶颈后而封顶

2.存储层

无论是存储的数据量，还是并发访问量，不同的业务模块之间的量级相差很大，比如说系统中，关系的数据量是远远大于人员数据量的，但是人员数据的访问量却远比关系数据要大。所以假如存储目前的瓶颈点是容量，那么我们只需要针对关系模块的数据做拆分就好 了，而不需要拆分用户模块的数据。所以存储拆分首先考虑的维度是**业务维度**。

拆分之后这个系统就有了人员库，关系库，设备库，记录库，地区库，这样做还能隔离故障，某一个库“挂了”不会影响到其它的数据库



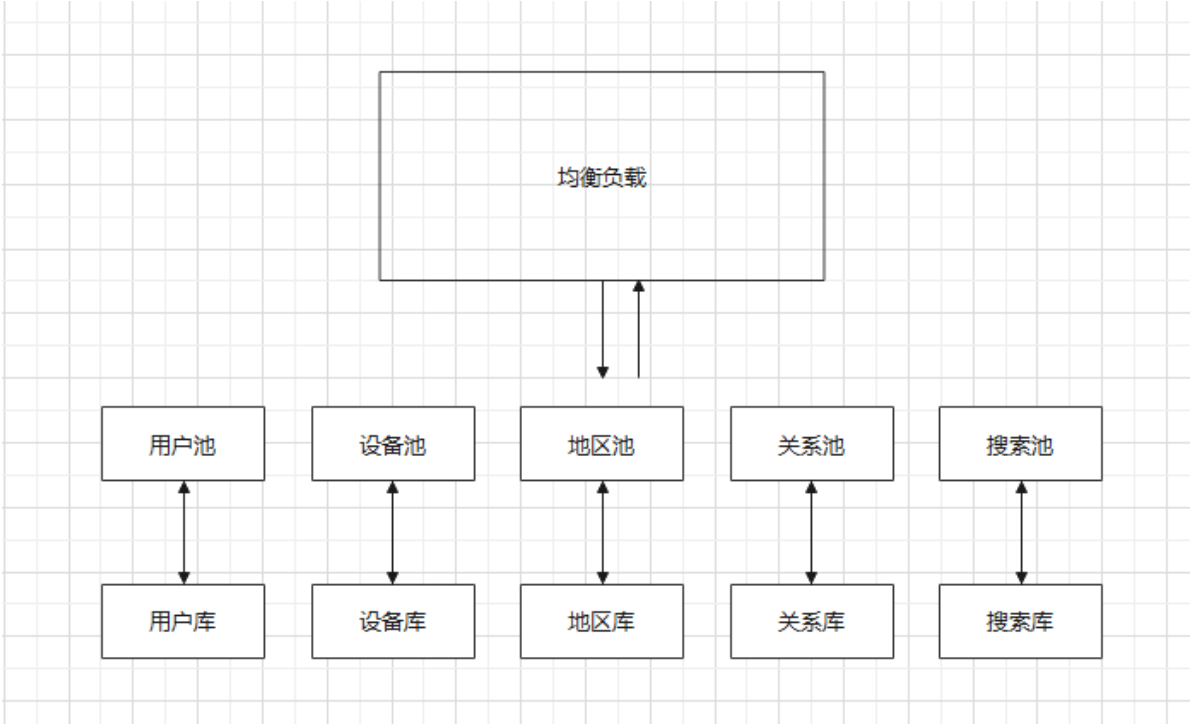
按照业务拆分，在一定程度上提升了系统的扩展性，但系统运行时间长了之后，如果单一的业务数据库在容量和并发请求量上仍然会超过单机的限制。这时我们就可以针对数据库做第二次**水平拆分**

比如说我们可以给用户库增加两个节点，然后借用第三方组件或者自造轮子将人员的数据拆分到这三个库里面，来做到均摊，那也就是咱们常说的水平分库分表水平拆分之后，我们就可以让数据库突破单机的限制了

3.业务层

我们一般会从三个维度考虑业务层的拆分方案，它们分别是：业务纬度，重要性纬度和请求来源纬度

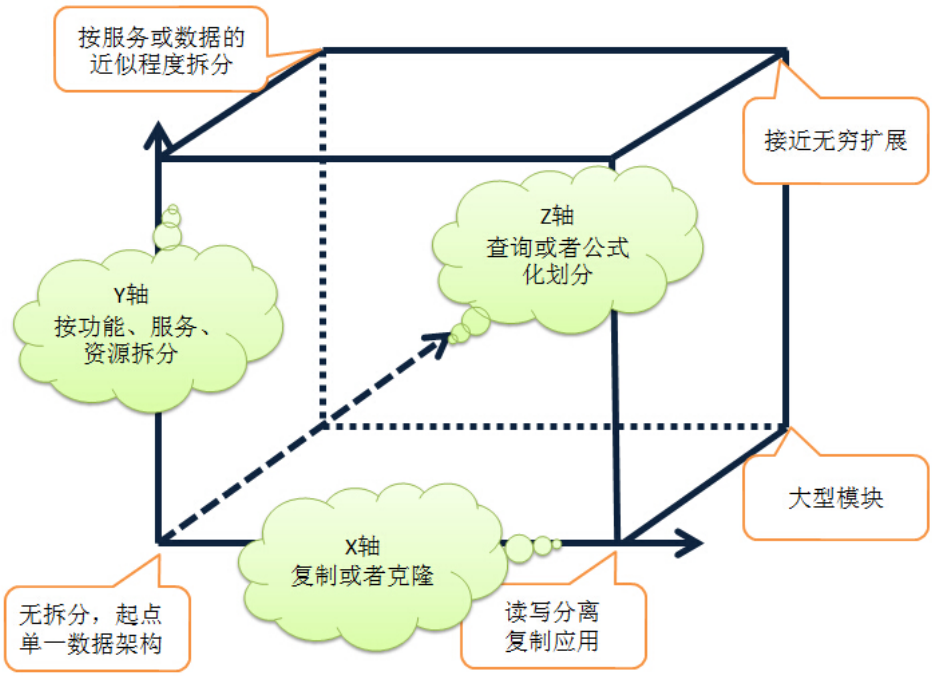
首先，我们需要把相同业务的服务拆分成单独的业务池，比方说`GlasssixBasicProject`项目中，我们可以按照业务的维度拆分成人员池、设备池、关系池、地区池和分析池。



每个业务依赖独自的数据库资源，不会依赖其它业务的数据库资源。这样当某一个业务的接口成为瓶颈时，我们只需要扩展业务的池子，以及确认上下游的依赖方就可以了，这样就大大减少了扩容的复杂度

除此以外，我们可以根据业务接口重要程度来进行拓展，当整体流量上升时优先扩容核心池，从而保证整体系统的稳定性。

最后，你还可以根据接入客户端类型的不同做业务池的拆分。比如说，服务于客户端接口的业务可以定义为外网池，服务于小程序或者 HTML 页面的业务可以定义为 H 池，服务于内部其它部门的业务可以定义为内网池等



4.复杂度分析

4.1.简介

基于以上内容，我们讲了如何从宏观角度要去描述了一个三高型的基础架构，如何进行分层设计，为什么要进行分层设计，等等一些设计性思想及方法理论，但是我们在具体实施中，如何落地，对于分布式而言，如何处理基于此类架构下来的业务痛点，及常现问题我们再进行具体论述。

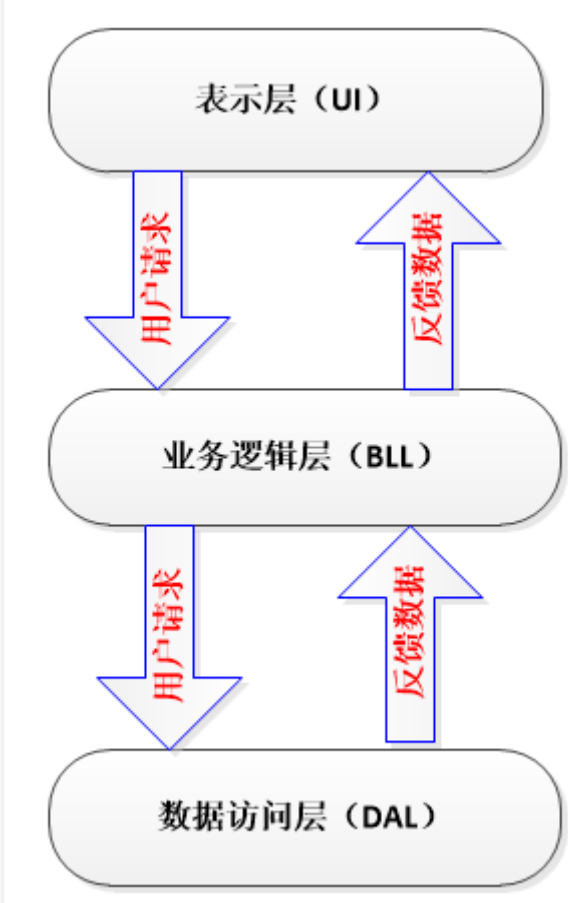
4.2.架构演进

我们享受分布式微服务带来的优点，但是同时也要接受所带来的固然问题及瓶颈。

那么如何找到这些固然问题，并加以解决呢，那首先我们要知道什么是分布式，什么是微服务，它与我们传统三层有何不同，它是如何出现的等等，我们在使用一个东西的时候，一定要展开它的基本知识点，知其然也知其所以然。

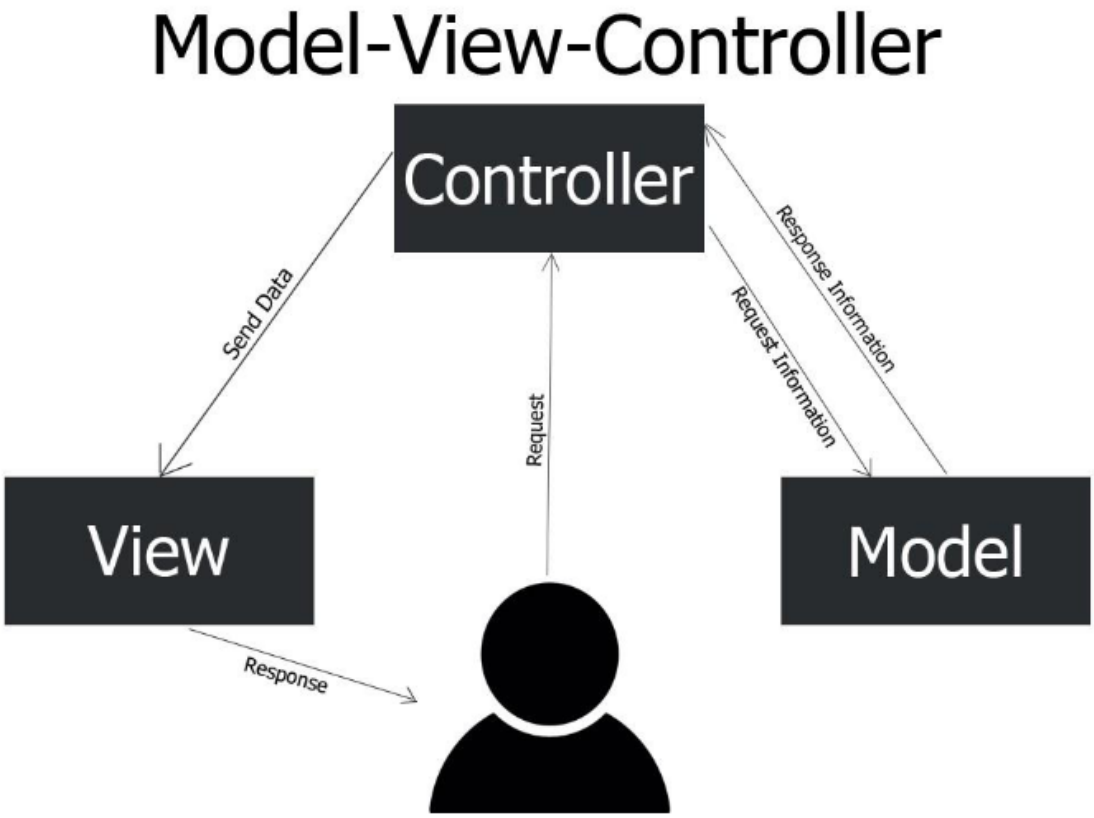
4.2.1.单体到服务化演进

以上讲到， 传统单体架构**三层架构**，数据层-逻辑层-表现层，这就是典型的二八原则的一个应用场景，与业务无关的80%的通用配置，模块由微软.net自身模块化组件提供，20%的业务逻辑，对象映射，等等基础服务只需要通过startup事先注册即可使用。



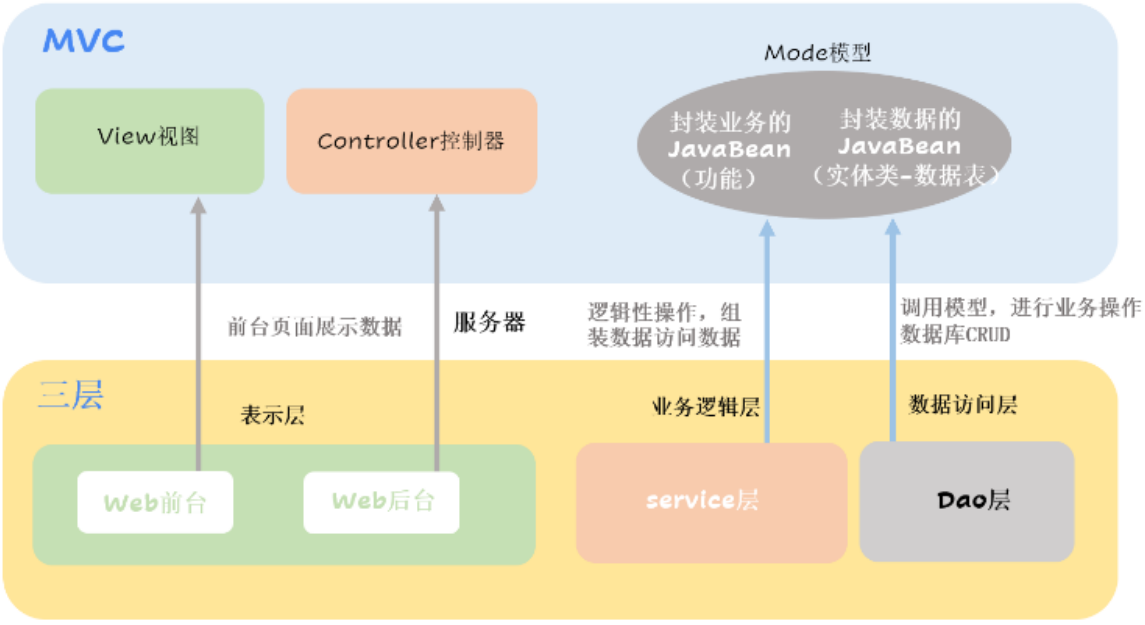
三层架构时代就已经对应用整体架构进行了逻辑分层，每层各司其职，并从功能类型上划分层级，每个层级的职责单一。

那讲了三层架构，再来讲讲**Mvc**



对于常用的mvc这块不再多做赘述，很多人喜欢讲mvc与三层架构混为一谈，但实际上，这是错误的认知，并不是说它们没有任何关系，而是MVC与三层架构不是简单的相等。

直接看图，各自特性也不再多做赘述：



可以看作，mvc是三层架构的延申，时至今日，我们仍旧在借用MVC框架进行应用开发。

从三层到mvc，服务的特点仍然是单体化，服务粒度抽象为模块化组件，所有组件耦合在一个项目中，配置和运行也是基于一个服务，如果某个模块发生出现问题，可能会导致其他模块也会出现问题。

另外，如今的互联网环境下，高并发高性能等词频频贴出，传统的三层，mvc已经无法对用户发起高并发请求进行处理，无法突破耦合下的性能瓶颈，单一进程已经无法满足需求，并且其水平拓展的能力也极其有限。

因此，**SOA**诞生了 (面向服务架构),SOA代表面向服务的架构，俗称服务化，SOA将模块通过预先定义的接口联系起来，其业务基于接口去实现，通常通过网络通信来完成各种组件交互，例如mysql，mq等，且不限于某种网络协议，HTTP,TCP/IP,消息队列协议，甚至是某种数据库存储形式，这使得各种组件可以以一种统一和通用的方式进行交互。

SOA有几项关键项技术，我们需要了解下：

1. **UDDI**

UDDI（Universal DescriptionDiscovery and Integration，统一描述、发现和集成）提供了一种服务发布、查找和定位的方法，是服务的信息注册规范，以便被需要该服务的用户发现和使用它。UDDI 规范描述了服务的概念，同时也定义了一种编程接口。通过 UDDI 提供的标准接口，企业可以发布自己的服务供其他企业查询和调用，也可以查询特定服务的描述信息，并动态绑定到该服务上。

在 UDDI 技术规范中，主要包含以下三个部分的内容：

- 数据模型。UDDI 数据模型是一个用于描述业务组织和服务的 XML Schema
- API，UDDI API 是一组用于查找或发布 UDDI 数据的方法，UDDI API 基于 SOAP
- 注册服务。UDDI 注册服务是 SOA 中的一种基础设施，对应着服务注册中心的角色

2.WSDL

WSDL（Web ServiceDescription Language，Web 服务描述语言）是对服务进行描述的语言，它有一套基于 XML 的语法定义。WSDL 描述的重点是服务，它包含服务实现定义和服务接口定义

如图：

采用抽象接口定义对于提高系统的扩展性很有帮助。服务接口定义就是一种抽象的、可重用的定义，行业标准组织可以使用这种抽象的定义来规定一些标准的服务类型，服务实现者可以根据这些标准定义来实现具体的服务。

服务实现定义描述了给定服务提供者如何实现特定的服务接口。服务实现定义中包含服务和端口描述。一个服务往往会包含多个服务访问入口，而每个访问入口都会使用一个端口元素来描述，端口描述的是一个服务访问入口的部署细节，例如，通过哪个地址来访问，应当使用怎样的消息调用模式来访问等。

3.SOAP

SOAP（Simple ObjectAccess Protocol，简单对象访问协议）定义了服务请求者和服务提供者之间的消息传输规范。SOAP 用 XML 来格式化消息，用 HTTP 来承载消息。通过 SOAP，应用程序可以在网络中进行数据交换和远程过程调用（Remote Procedure Call， RPC）。SOAP 主要包括以下四个部分：

- 封装。SOAP 封装定义了一个整体框架，用来表示消息中包含什么内容，谁来处理这些内容，以及这些内容是可选的还是必需的
- 编码规则。SOAP 编码规则定义了一种序列化的机制，用于交换系统所定义的数据类型的实例
- RPC 表示。SOAP RPC 表示定义了一个用来表示远程过程调用和应答的协议
- 绑定。SOAP 绑定定义了一个使用底层传输协议来完成在节点之间交换 SOAP 封装的约定

SOAP 消息基本上是从发送端到接收端的单向传输，但它们常常结合起来执行类似于请求/应答的模式。所有的 SOAP 消息都使用 XML 进行编码。SOAP 消息包括以下三个部分：

- 封装（信封）。封装的元素名是 Envelope，在表示消息的 XML 文档中，封装是顶层元素，在 SOAP 消息中必须出现
- SOAP 头。SOAP 头的元素名是 Header，提供了向 SOAP 消息中添加关于这条 SOAP 消息的某些要素的机制。SOAP 定义了少量的属性用来表明这项要素是否可选以及由谁来处理。SOAP 头在 SOAP 消息中可能出现，也可能不出现。如果出现的话，必须是 SOAP 封装元素的第一个直接子元素
- SOAP 体。SOAP 体的元素名是 Body，是包含消息的最终接收者想要的信息的容器。SOAP 体在 SOAP 消息中必须出现且必须是 SOAP 封装元素的直接子元素。如果有头元素，则SOAP 体必须直接跟在 SOAP 头元素之后；如果没有头元素，则 SOAP 体必须是 SOAP 封装元素的第一个直接子元素

4.REST

REST（RepresentationalState Transfer，表述性状态转移）是一种只使用 HTTP 和 XML 进行基于 Web 通信的技术，可以降低开发的复杂性，提高系统的可伸缩性。它的简单性和缺少严格配置文件的特性，使它与 SOAP 很好地隔离开来，REST 从根本上来说只支持几个操作（POST、GET、PUT 和 DELETE），这些操作适用于所有的消息。REST 提出了如下一些设计概念和准则：

- 网络上的所有事物都被抽象为资源
- 每个资源对应一个唯一的资源标识
- 通过通用的连接件接口对资源进行操作
- 对资源的各种操作不会改变资源标识
- 所有的操作都是无状态的

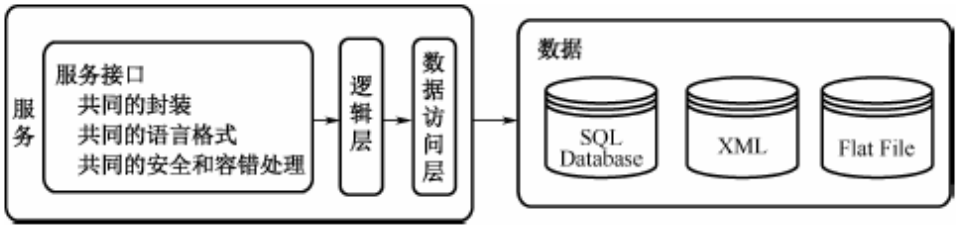
对比三层，mvc，soa讲模块化组件从单一进程进行拆分，形成独立的对外提供服务的网络化组件，其通过某种网络协议对外提供服务，特性如下：

- soa定义接口，通过网络协议对外提供服务，业务模块之间表现松耦合且灵活，可以自由进行组装，抽象
- 某业务模块在发生改变时，不影响整个流程对外提供服务，只需要对外接口不变，则对于服务内部机制对外面来讲可以是透明的
- 统一数据返回类型，不限于Xml，Json，Text
- 通过定义接口，可以让底层服务进行下沉，供多上层服务同时使用，增加服务的可重用性

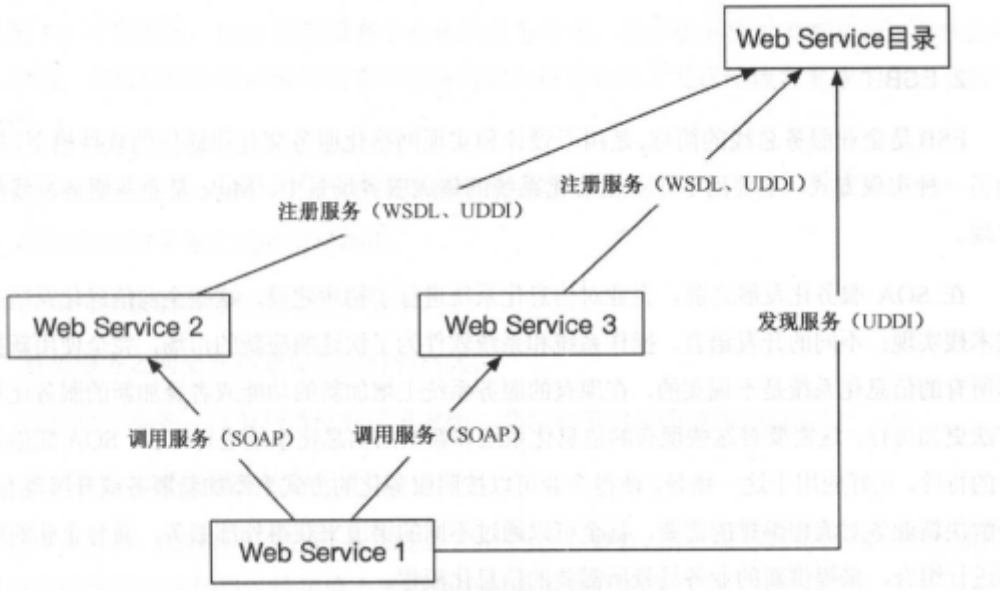
- 公共化对外接口，于企业来说，避免重复造轮子

对于SOA,我们必须了解其两个主流实现方式：**WebService**和**ESB**

webService:

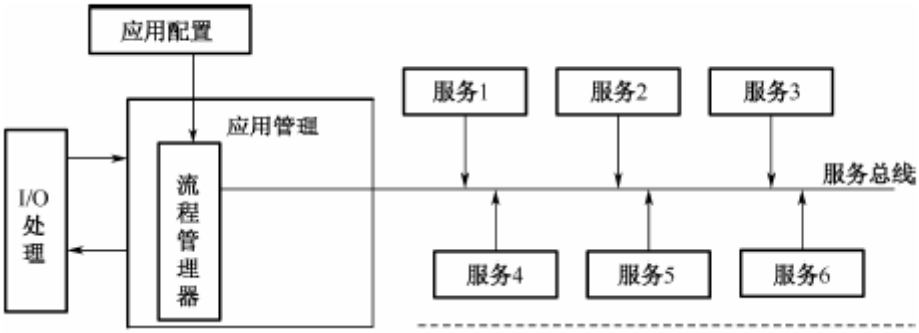


如图为Web Service工作原理图：



可以看到，每个服务之间是对等的，并且相互解耦，通常是WSDL定义的服务发现接口进行访问，并通过SOAP协议进行通信，但是每个服务前提都需要进行服务发现。

ESB:



ESB是企业服务总线的简称，用于设计和实现网络化服务交互通信的一种软件模型，也是soa的另一种实现方式，在soa出现时期，市场已经饱和，完全重建所有企业化系统是不现实的，只能在其现有服务上增加新的功能活着叠加新的服务化系统，这需要对其现有服务及新增服务进行组合，soa特性刚好就适合干这个事，但是随着服务增多，服务之间的关联愈加复杂。

于是ESB应运而生，esb提供的一种基础设施，消除了服务之间的直接连接，使得其进一步解耦

其特性如下：

- 监控和控制服务间路由
- 控制可插拔服务化功能和版本
- 充当缓冲器等等

  可以看到服务总线是ESB的核心要素，所有服务都可以在总线上插拔，并通过总线的流程编排达到组合实现业务能力

4.2.2.服务化到微服务演进

服务化架构本身与微服务架构是一本同源，只是对于不同的市场要求下，服务的粒度及解决方案都得到了兼容及升华，形成适合现代化应用场景的一个方法论

要论不同的话，三个方面，目的，粒度，部署

目的：

- SOA服务化涉及的范围更广一些，强调不同的异构服务之间的协作和契约，并强调有效集成、业务流程编排、历史应用集成等，典型的代表为Web Service和ESB。
- 微服务使用一系列的微小服务来实现整体的业务流程，目的是有效地拆分应用，实现敏捷开发和部署，在每个微小服务团队里，减少了跨团队的沟通，让专业人做专业的事，缩小变更和迭代影响的范围，并达到单一微服务更容易水平扩展的目的

粒度：

- 从DDD的角度讲，微服务比Soa拆分粒度更小，通过多个服务组合来实现业务流程的处理，拆分到职责单一，甚至小到不能再进行拆分
- SOA对粒度没有要求，在实践中服务通常是粗粒度的，更多的是强调接口契约的规范化

部署：

- 微服务将完整的应用拆分成多个细小的服务，通常使用敏捷扩容、缩容的Docker技术来实现自动化的容器管理，每个微服务运行在单一的进程内，微服务中的部署互相独立、互不影响。
- SOA服务化通常将多个业务服务通过组件化模式方式打包在一个包里，然后统一部署在一个应用服务器上

4.2.3.分布式与微服务

分布式系统是计算元素的集合，每个计算元素都能够相互独立地工作

我们通常将计算元素称为节点，它可以是硬件设备或软件进程。

第二个因素是用户(无论是人还是应用程序)认为他们在处理一个系统。这意味着自治节点以某种方式需要协同和作。如何建立这种协作是开发分布式系统的核心

微服务：一种软件开发技术- 面向服务的体系结构（SOA）架构样式的一种变体，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值

目的不同：

- 分布式系统应使资源易于访问，隐藏资源分布在网络上的事实，开放的，可伸缩的。重在资源共享与加快计算机速度，强调的是服务化以及服务的分散化。
- 微服务架构倡导将软件应用设计成多个独立开发、可配置、可运行和可维护的子服务。重在于松耦合和高内聚的效果，使每个模块独立，强调服务的专业化和精细分工。

个人观点，微服务是一种具象化应用技术，分布式是一种倡导的思想理念，微服务常交付方式是分布式，即使可被单体化部署，如果硬归结关系，我认为微服务是分布式系统设计及架构的理念之一，也是分布式的一种落地体现

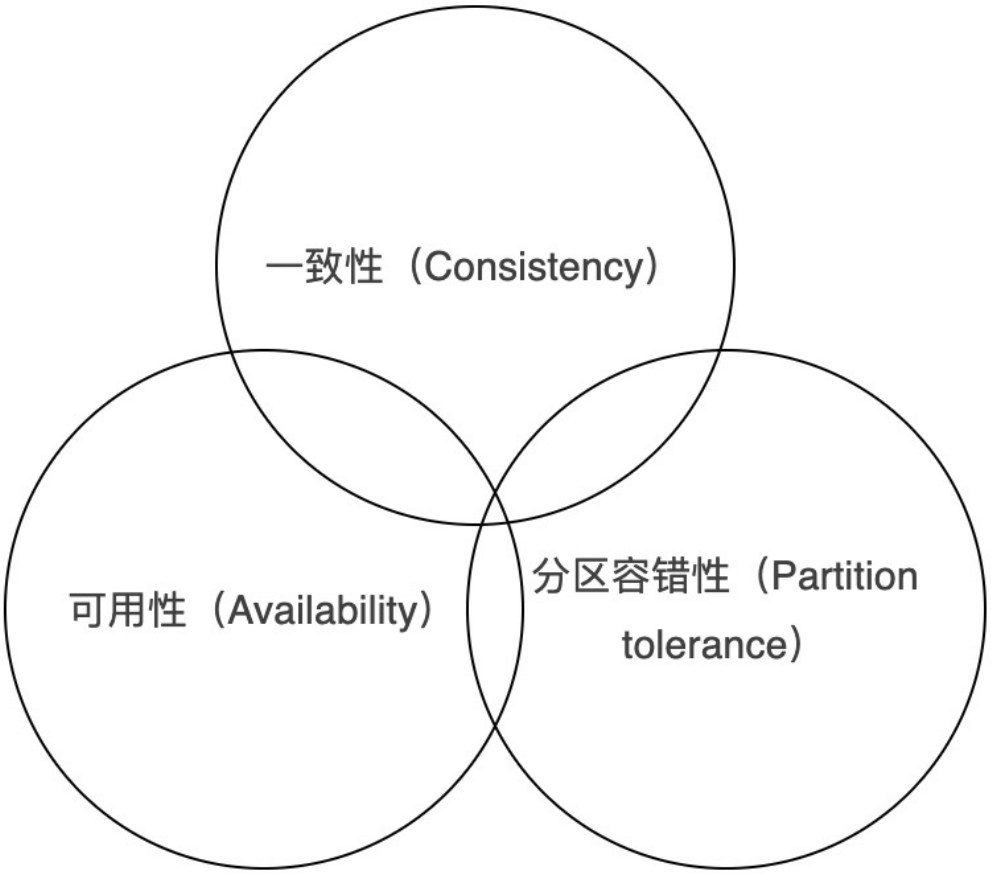
4.3.分布式一致性

在分布式系统中，随着系统架构演进，原来的原子性操作会随着系统拆分而无法保障原子性从而产生一致性问题，但业务实际又需要保障一致性

4.3.1. CAP & Base理论

CAP定理指的是在一个分布式系统中，一致性（Consistency）、可用性（Availability）、分区容错性（Partition tolerance）。这三个要素最多只能同时实现两点，不可能三者兼顾：

- 一致性：在分布式系统中的所有数据备份，在同一时刻是否同样的值。
- 分区容错性：可靠性，无论应用程序或系统发生错误，还是用户以意外或错误的方式使用，软件系统都能继续运行。
- 可用性：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。



CAP理论3选2是伪命题，实际上必须从A和C选择一个和P组合，更进一步基本上都会选择A，相比一致性，系统一旦不可用或不可靠都可能会造成整个站点崩溃，所以一般都会选择AP。但是不一致的问题也不能忽略，使用最终一致是比较好的办法

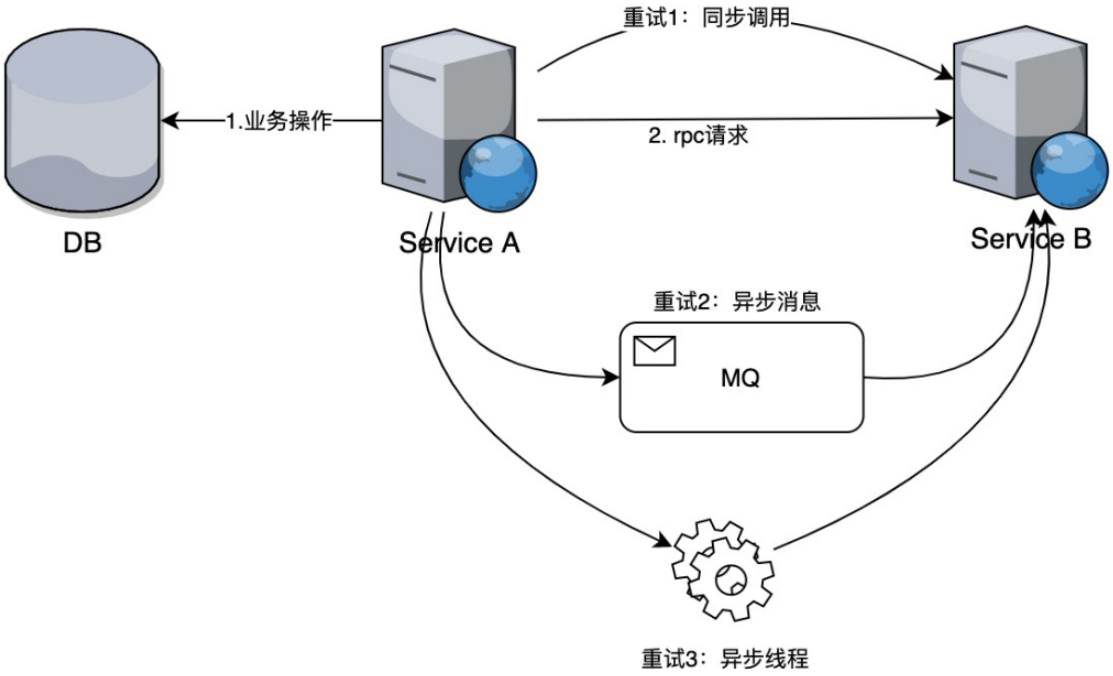
BASE理论是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结， 是基于CAP定理逐步演化而来的。BASE理论的核心思想是：即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。接下来看一下BASE中的三要素：

- 基本可用（Basically Available）：基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性。注意，这绝不等价于系统不可用。比如：
- - 响应时间上的损失。正常情况下，一个在线搜索引擎需要在0.5秒之内返回给用户相应的查询结果，但由于出现故障，查询结果的响应时间增加了1~2秒
  - 系统功能上的损失：正常情况下，在一个电子商务网站上进行购物的时候，消费者几乎能够顺利完成每一笔订单，但是在一些节日大促购物高峰的时候，由于消费者的购物行为激增，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面。
- 软状态（Soft State）：软状态指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时。
- 最终一致（Eventually Consistent）：最终一致性强调的是所有的数据副本，在经过一段时间的同步之后，最终都能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性

最终一致的核心做法就是通过记录对应操作并在操作失败时不断进行重试直到成功为止

4.3.2.重试

在出现一致性问题时如果系统的并发或不一致情况较少，可以先使用重试来解决



在调用Service B超时或失败时进行重试：

- 同步调用，捕获异常重新调用Service B
- 异步消息，捕获异常发送延迟消息重新调用Service B
- 异步线程，捕获异常开启异步线程重新调用Service B

如果重试还是不能解决问题，那么需要使用分布式事务来解决

4.3.3.分布式事务

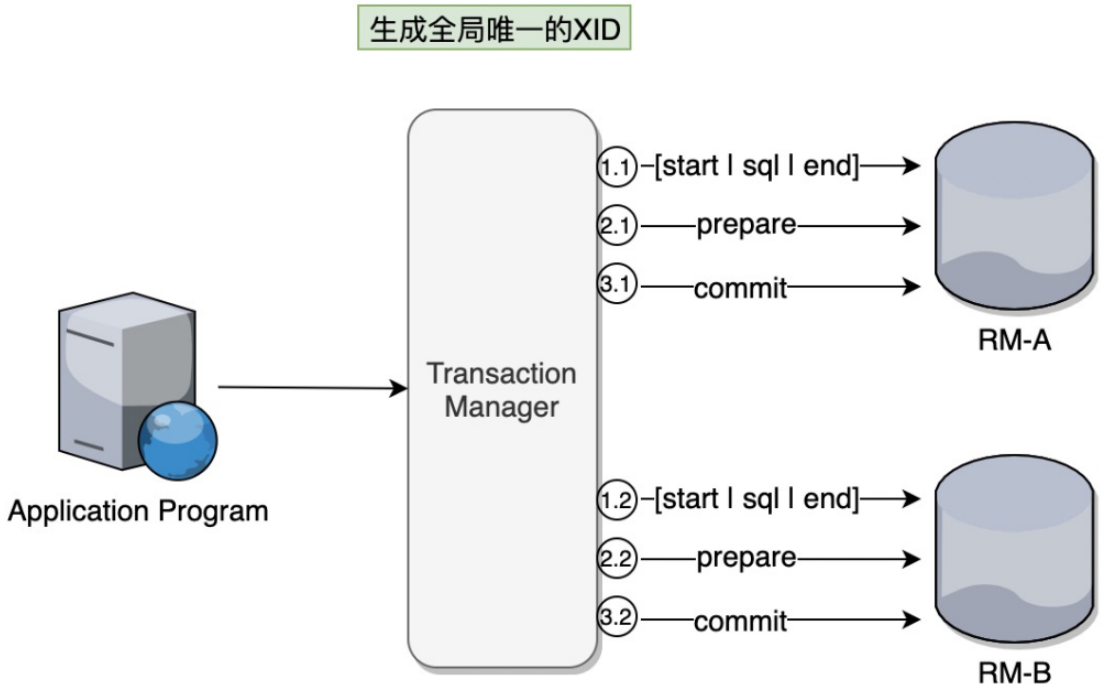
1.2pc-xa协议

XA事务由一个或多个资源管理器（Resource Managers）、一个事务管理器（Transaction Manager）以及一个应用程序（Application Program）组成。

- 资源管理器（RM）：参与者。提供访问事务资源的方法。通常一个数据库就是一个资源管理器。
- 事务管理器（TM）：协调者。分配标识符，监视事务的进度，并负责事务完成和故障恢复。
- 应用程序（AP）：发起者。定义事务的边界，制定全局事务中的操作

整个过程分为2个阶段：准备和提交

完成则下一步，否则回滚



问题：

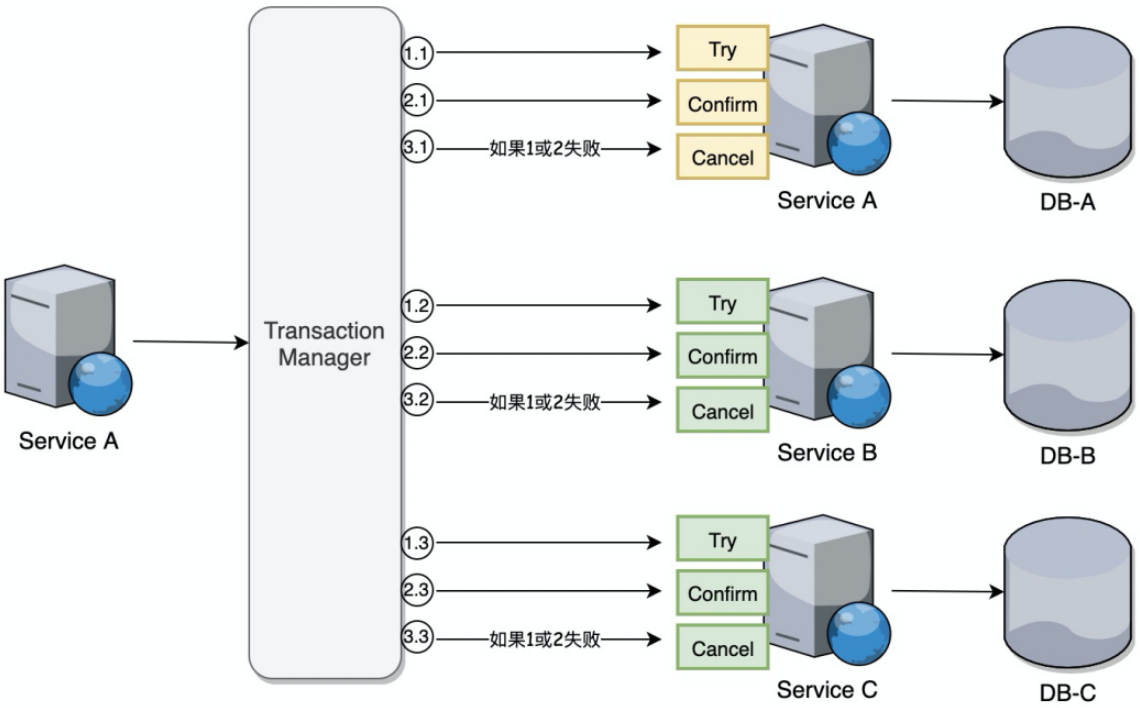
- commit丢失/超时导致数据不一致。A commit成功，B commit时网络超时导致数据库未收到commit请求。
- 性能低。所有事务参与者在等待其它参与者响应的时候都处于同步阻塞状态。
- 主备数据不一致。
- 协调者单点故障，在任意阶段协调者发生故障都会导致分布式事务无法进行。

2.3pc协议

3PC只是通过在参与者加入**超时机制**解决2PC中的协调者单点带来的事务无法进行的问题，**但是性能和一致性仍没有解决**

3.Tcc协议

TCC是通过最终一致来达到分布式事务的效果，即：在短时间内无法保证一致性，但最终会一致。核心分为2个阶段：1.Try 2.Confirm or Cancel。先尝试（Try）操作数据，如果都成功则全部确认（Confirm）该修改，如果有任意一个尝试失败，则全部取消（Cancel）。**简单点说就是增加了可回改能力，但是实际上还是没有解决这个问题，该失败还是失败，服务并没有相互对应处理方式**



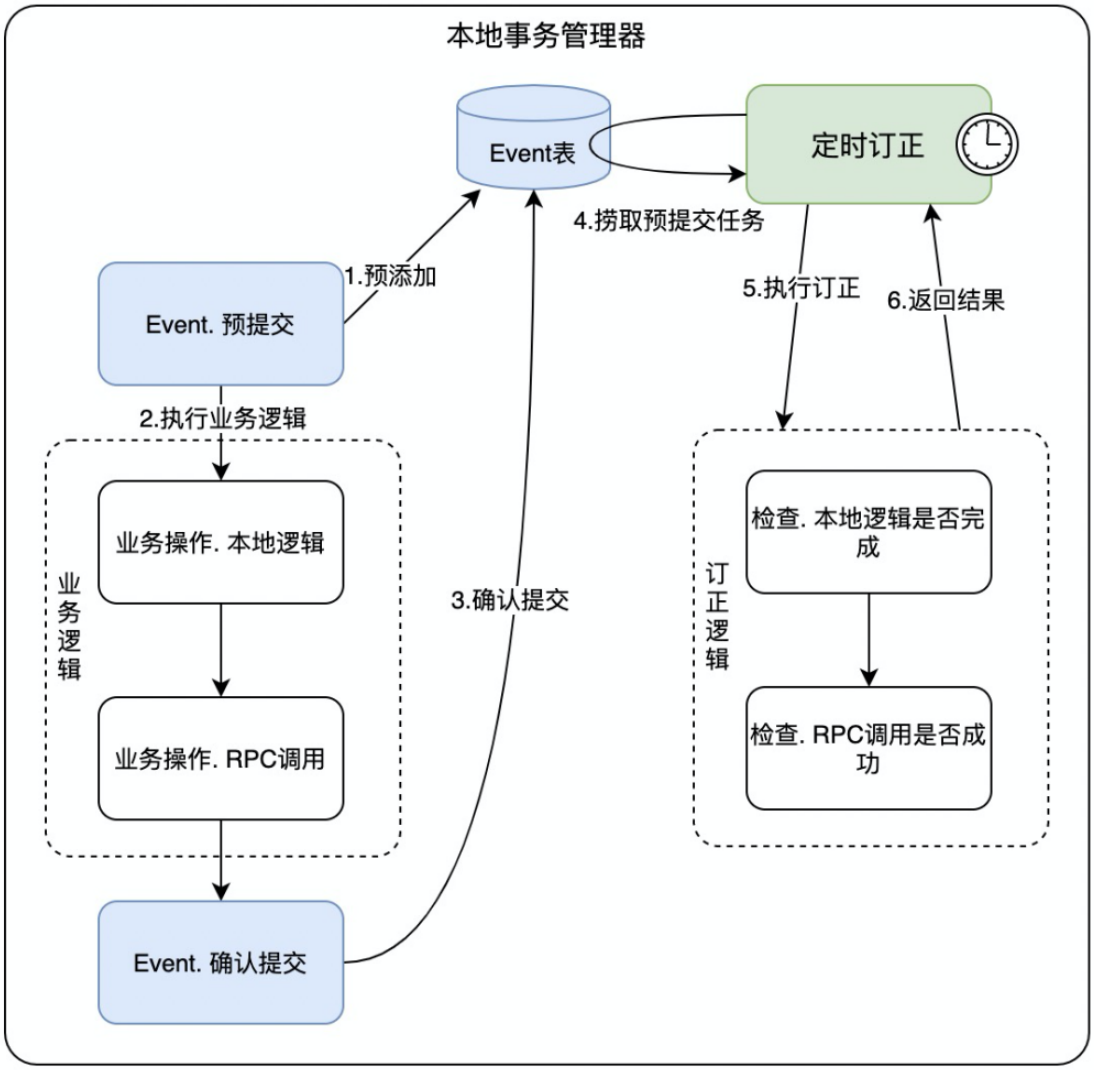
4.3.4.事务管理器

在实际运用中事务管理器TM一般是由应用程序兼职实现的，这样对业务代码侵入性大，最好是把TM做成中间层。接下来详细看一下TM的职责：协调者。分配标识符，监视事务的进度，并负责事务完成和故障恢复。TM需要具备持久化能力才能完成监控、故障恢复和协调，整个协调过程可以分为3个阶段：

- 准备阶段：向TM注册全局事务并获取到全局唯一XID，这一步需要定义好事务的范围。
- 执行阶段：应用程序AP执行业务功能操作自己的RM，全部执行完毕后向TM commit，如果无失败则整个事务成功。
- 确认/回滚阶段：如果第2步出现异常会触发该阶段，TM咨询各个AP对应操作是否成功，如果成功则commit，如果失败则调用AP进行rollback。

1.本地事务管理器

对于简单的业务可能只要保障2个数据库之间的一致，这样在本地实现事务管理器比较快成本也不高



1. 在做业务逻辑之前把对应事件添加到本地event表中（记录订正时所需要的关键数据）
2. 执行业务逻辑
3. 本地业务逻辑，操作表数据等等
4. RPC调用其他服务
5. 修改event状态为确认
6. 如果第2步不成功，则event状态还是预提交，通过定时任务捞取再执行订正逻辑
7. 检查业务逻辑中的操作是否完成，如未完成则执行订正逻辑
8. 本地业务逻辑是否执行成功
9. RPC调用其他服务是否执行成功
10. 返回订正结果
11. 成功则修改event为确认提交
12. 未成功则不操作，等待轮询订正

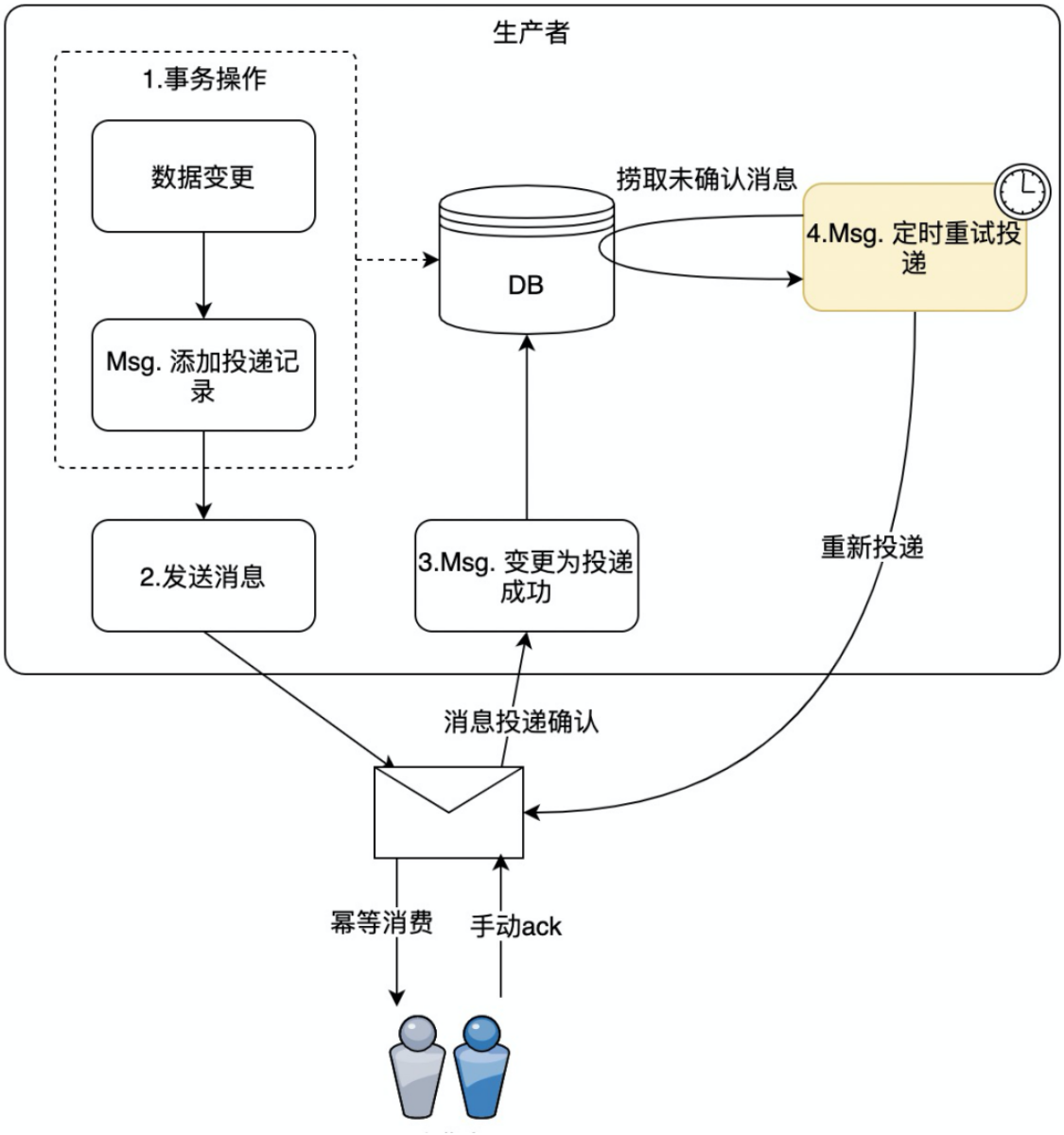
2.外部事务管理器

参见阿里云GTS服务

https://help.aliyun.com/document_detail/157850.html%EF%BB%BF

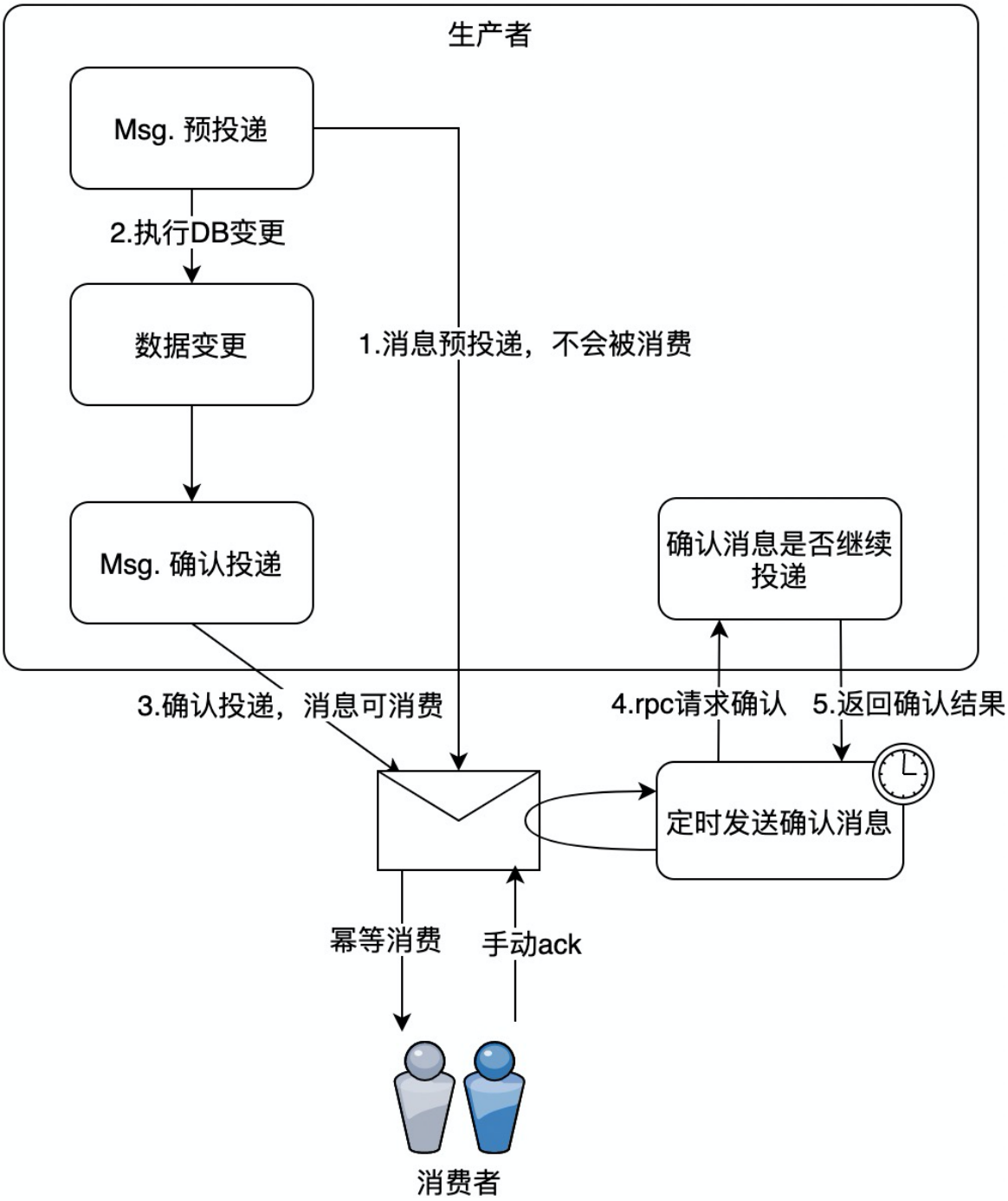
4.3.5.MQ与DB一致性

在实际场景中，业务系统对本地DB数据变更后会广播对应的消息，消费者消费消息做自己的业务逻辑，按正常逻辑消息会在数据库变更后发出，如果消息发送超时且失败那么DB和MQ之间就产生了不一致问题，如何解决呢？使用可靠消息来解决，核心逻辑保证消息从投递到消费的过程中不会丢失：生产者confirm、消费者ack和持久化。理论上只要使用生产者确认机制即可，但是不使用消费者ack则没有意义。消费者需要开启手动ack同时做到幂等消费，MQ需要通过将exchange、queue和message进行持久化来保证消息不丢失，生产者则需要通过确认机制保证消息一定投递到MQ中，重点讲解一下生产者确认机制



confirm机制是在消息投递到所有匹配的queue之后发送确认消息给生产者，这样生产者就知道消息投递成功，但是由于消息是在DB操作之后发出的，生产者必须增加记录表来记录消息投递状态，如果投递成功就在收到确认消息时把记录标记为投递成功，如果长时间未收到确认消息则大概率是消息丢失了，再定时重新投递，这样就可以保证消息最终一定能投递成功。核心逻辑其实就是通过全局事务ID来标识DB操作和MQ消息是在一个分布式事务中

以上方式依靠业务处理完善操作，当然这部分可交给中间件来处理



预投递的消息不会分发到queue中，只有在接收到确认投递的请求后才会进行投递，如果确认操作因为网络异常失败了，MQ在过一段时间之后主动询问业务系统该消息是否可投递（失败不断重试），这样就能在异常时做到最终一致，不过依赖MQ的能力

4.3.6.DB与缓存一致性

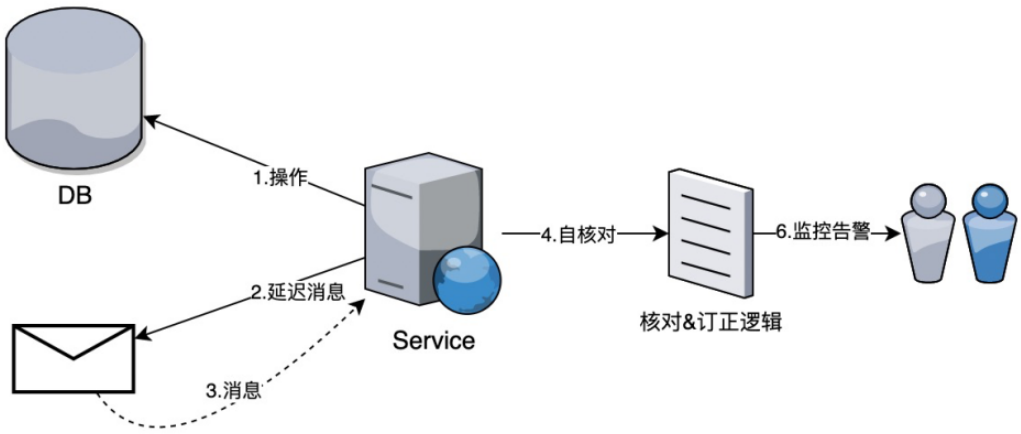
DB和缓存之间同样也存在不一致问题，先写DB再写缓存如果缓存写失败就不一致了，同样的需要重试更新缓存来做最终一致。

4.3.7.兜底核对

虽然有了分布式事务，但是在实际场景中可能会因为bug导致数据不一致，这时需要兜底来做最后一道防线，通过定时核对数据是否一致，如不一致手动/自动进行订正

1.自核对

如果系统数量和数据量不多的情况下可以由业务系统自行核对，通过发送延迟消息自消费或监听其他系统消息做相关数据核对并进行订正



2.搭建核对系统

基于自核对，可以将这部分单元抽出做成一个公共独立的服务

4.3.8.Saga机制

以上理论，为传统系统下分布式事务处理方式，由此又延申出了目前最终处理方案，称为**Saga**

Saga是由一系列的本地事务构成。每一个本地事务在更新完数据库之后，会发布一条消息或者一个事件来触发Saga中的下一个本地事务的执行。如果一个本地事务因为某些业务规则无法满足而失败，Saga会执行在这个失败的事务之前成功提交的所有事务的补偿操作

Saga的实现有很多种方式，其中最流行的两种方式是：

- 基于事件，无协调中心，整个工作模式按预先编排顺序执行，模块只需要负责自己的事
- 基于命令，但是此种方式需要协调中心进行工作协调及任务划分

1.基于事件

在基于事件的方式中，第一个服务执行完本地事务之后，会产生一个事件。其它服务会监听这个事件，触发该服务本地事务的执行，并产生新的事件

例如，订单流程

1. 订单服务创建一笔新订单，将订单状态设置为"待处理"，产生事件ORDER_CREATED_EVENT。
2. 支付服务监听ORDER_CREATED_EVENT，完成扣款并产生事件BILLED_ORDER_EVENT。
3. 库存服务监听BILLED_ORDER_EVENT，完成库存扣减和备货，产生事件ORDER_PREPARED_EVENT。
4. 物流服务监听ORDER_PREPARED_EVENT，完成商品配送，产生事件ORDER_DELIVERED_EVENT。

5. 订单服务监听ORDER_DELIVERED_EVENT，将订单状态更新为"完成"

如果在这个过程中出现异常

那只需要提供对应的补偿事件即可，例如

该订单没有库存，则触发退款事件，进行退款

订单标记为失败

基于事件方式的优缺点

优点：简单且容易理解。各参与方相互之间无直接沟通，完全解耦。这种方式比较适合整个分布式事务只有2-4个步骤的情形。

缺点：这种方式如果涉及比较多的业务参与方，则比较容易失控。各业务参与方可随意监听对方的消息，以至于最后没人知道到底有哪些系统在监听哪些消息。更悲催的是，这个模式还可能产生环形监听，也就是两个业务方相互监听对方所产生的事件，复杂度全部隐藏在系统内部，对于调试及运维来说都是非常头疼

4.4.可观测性理论

4.4.1.定义

首先可观测性理论来自控制论的一个概念

用相对严谨的话来说，可观测性指的是一种能力--是通过检查其输出来衡量系统内部状态的能力。这些输出体现内部系统状态的能力越强，可观测性也就越好

可观测性描述的就是"观测-判断-优化-再观测"这个闭环的连续性、高效性如果只有观测而无法基于观测做出判断，则不能称其具备可观测性。如果只有经验判断而没有数据支撑，也不能称其具备可观测性

4.4.2.价值

谷歌给出可观测性的核心价值很简单：快速排障（troubleshooting）

这个世界上没有不存在 Bug 的系统，而随着系统越来越精细，越来越复杂，越来越动态，越来越庞大，潜藏的问题和风险也就越来越多。

因此，任何一个软件的成功，不仅仅要依靠软件架构的合理设计，软件开发的代码质量，更要依靠软件系统的运行维护。而运行维护的基础，就是可观测性

因此，在 CNCF 对于云原生的定义中，已经明确将可观测性列为一项必备要素

4.4.3.三大支柱

1、**logs**（日志）

2、**metrics**（指标）

指标是在一段时间内测量的数值，包括特定属性，例如时间戳、名称、KPI 和值。与日志不同，指标在默认情况下是结构化的，这使得查询和优化存储变得更加容易，让您能够将它们保留更长时间。

3、**traces**（跟踪）

跟踪表示请求通过分布式系统的端到端旅程。当请求通过主机系统时， 对其执行的每个操作（称为“跨度”）都使用与执行该操作的微服务相关的重要数据进行编码。通过查看跟踪，每个跟踪都包含一个或多个跨度，您可以通过分布式系统跟踪其进程并确定瓶颈或故障的原因。

4.5.基于调用链的服务治理

在微服务化构建中，一个系统被拆分成了许多模块。这些模块负责不同的功能，组合成系统，在这种架构中，一次请求往往需要涉及到多个服务。互联网应用构建在不同的软件模块集上，这些软件模块，有可能是由不同的团队开发、可能使用不同的编程语言来实现、有可能布在了几千台服务器，横跨多个不同的数据中心，也就意味着这种架构形式也会存在一些问题

如何快速发现问题?

如何判断故障影响范围?

如何梳理服务依赖以及依赖的合理性?

如何分析链路性能问题以及实时容量规划?

那就要使用到链路追踪，简称**Apm**，链路追踪就是将一次分布式请求还原成调用链路，进行日志记录，性能监控并将一次分布式请求的调用情况集中展示。比如各个服务节点上的耗时、请求具体到达哪台机器上、每个服务节点的请求状态等等

对此市面上具有很多第三方开源组件，这里框架要用的是**Skyworking**

5.基础设施

5.1.后台任务

后台作业用来在后台里执行应用里的一些任务, 业务中,可能需要后台工作:

- 为执行**长时间运行的任务**而用户无需等待
- 创建**可重试**和**持久的任务**以**确保**代码将**成功执行**

后台作业是**持久性**，即使任务失败, 后台作业也会在稍后**重试**并**执行**.

1.Hangfire

2.Rabbitmq

3.Quartz

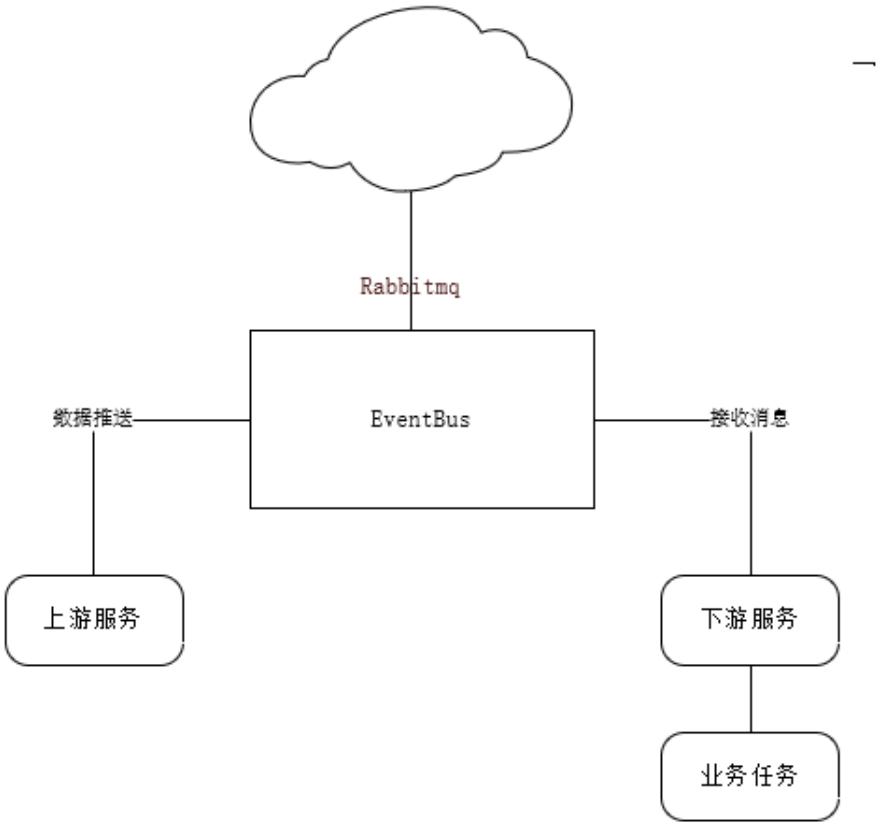
5.2.事件总线

事件总线是将消息从发送方传输到接收方的中介. 它在对象,服务和应用程序之间提供了一种松散耦合的通信方式.

简称`EventBus`，对于基于事件型业务是一种很好的解决方案，常做分布式异步通信

生产消费者模式，上游进行推送，消费者进行订阅，消息体接收到以后会自动执行业务任务

以下暂时只实现`Rabbitmq`方式



5.2.1.全局配置

需在appsetting.json中加入

```
"RabbitMq": {
  "EventBusConnection": "10.168.1.47",//endpoint
  "EventBusUserName": "guest",//账户
  "EventBusPassword": "guest",//密码
  "EventBusRetryCount": "5",//失败重试次数
  "Port": 32672,//端口
  "SubscriptionClientName": "ReportAnEmergency"//queue
},
```

5.2.2.服务注入

服务注入暂时支持两种消息模式，路由及主题

2.1 路由模式

```
            services.AddSingleton<IDefaultRabbitMQConnection>(sp =>
            {
                var logger =
sp.GetRequiredService<ILogger<DefaultRabbitMQConnection>>();
                var retryCount = 10;
                var factory = new ConnectionFactory()
                {
                    HostName = Configuration["RabbitMq:EventBusConnection"],
                    DispatchConsumersAsync = true,
                };

                if (!string.IsNullOrEmpty(Configuration["RabbitMq:Port"]))
                    factory.Port =
Convert.ToInt32(Configuration["RabbitMq:Port"]);

                if
(!string.IsNullOrEmpty(Configuration["RabbitMq:EventBusUserName"]))
                    factory.UserName =
Configuration["RabbitMq:EventBusUserName"];


                if
(!string.IsNullOrEmpty(Configuration["RabbitMq:EventBusPassword"]))
                    factory.Password =
Configuration["RabbitMq:EventBusPassword"];

                if
(!string.IsNullOrEmpty(Configuration["RabbitMq:EventBusRetryCount"]))
                    retryCount =
int.Parse(Configuration["RabbitMq:EventBusRetryCount"]);

                return new DefaultRabbitMQConnection(factory, logger,
retryCount,null);
            });
       services.AddSingleton<IEventBus, EventBusRabbitMQ>(sp =>
            {
                var subscriptionClientName =
Configuration["RabbitMq:SubscriptionClientName"];
                var rabbitMQPersistentConnection =
sp.GetRequiredService<IDefaultRabbitMQConnection>();
                var iLifetimeScope = sp.GetRequiredService<ILifetimeScope>();
                var logger = sp.GetRequiredService<ILogger<EventBusRabbitMQ>>();
                var eventBusSubcriptionsManager =
sp.GetRequiredService<IEventBusSubscriptionsManager>();
                var retryCount = 5;
                return new EventBusRabbitMQ(rabbitMQPersistentConnection,
logger, iLifetimeScope, eventBusSubcriptionsManager, subscriptionClientName,
retryCount);
            });

            services.AddSingleton<IEventBusSubscriptionsManager,
InMemoryEventBusSubscriptionsManager>();
```

```
            services.AddTransient<PushExceptionMessageEventHandler>();//注：
PushExceptionMessageEventHandler对应业务事件模型
//消息订阅
//管道获取实例
 var eventBus = app.ApplicationServices.GetRequiredService<IEventBus>();
            eventBus.Subscribe<RecordsMessageEvent, RecordsMessageEventHandler>
();
            eventBus.HealthyCheck();
//注：RecordsMessageEvent对应业务事件模型 RecordsMessageEventHandler对应业务处理方法
```

2.2 主题模式

```
            services.AddSingleton<IDefaultRabbitMQConnection>(sp =>
            {
                var logger =
sp.GetRequiredService<ILogger<DefaultRabbitMQConnection>>();
                var retryCount = 10;
                var factory = new ConnectionFactory()
                {
                    HostName = Configuration["RabbitMq:EventBusConnection"],
                    DispatchConsumersAsync = true,
                };

                if (!string.IsNullOrEmpty(Configuration["RabbitMq:Port"]))
                    factory.Port =
Convert.ToInt32(Configuration["RabbitMq:Port"]);

                if
(!string.IsNullOrEmpty(Configuration["RabbitMq:EventBusUserName"]))
                    factory.UserName =
Configuration["RabbitMq:EventBusUserName"];


                if
(!string.IsNullOrEmpty(Configuration["RabbitMq:EventBusPassword"]))
                    factory.Password =
Configuration["RabbitMq:EventBusPassword"];

                if
(!string.IsNullOrEmpty(Configuration["RabbitMq:EventBusRetryCount"]))
                    retryCount =
int.Parse(Configuration["RabbitMq:EventBusRetryCount"]);

                return new DefaultRabbitMQConnection(null, logger,
retryCount,factory);
            });

       services.AddSingleton<IEventBus, EventBusRabbitMQ>(sp =>
            {
                var subscriptionClientName =
Configuration["RabbitMq:SubscriptionClientName"];
                var rabbitMQPersistentConnection =
sp.GetRequiredService<IDefaultRabbitMQConnection>();
```

```
                var iLifetimeScope = sp.GetRequiredService<ILifetimeScope>();
                var logger = sp.GetRequiredService<ILogger<EventBusRabbitMQ>>();
                var eventBusSubcriptionsManager =
sp.GetRequiredService<IEventBusSubscriptionsManager>();
                var retryCount = 5;
                return new EventBusRabbitMQ(rabbitMQPersistentConnection,
logger, iLifetimeScope, eventBusSubcriptionsManager, subscriptionClientName,
retryCount);
            });

            services.AddSingleton<IEventBusSubscriptionsManager,
InMemoryEventBusSubscriptionsManager>();
            services.AddTransient<PushExceptionMessageEventHandler>();//注：
PushExceptionMessageEventHandler对应业务事件模型
//消息订阅
//管道获取实例
var Exchange = Configuration["RabbitMQ:ExchangeName"];
var queue = Configuration["RabbitMQ:Queue"];
var routingKey = Configuration["RabbitMQ:RoutingKey"];
var eventBus = app.ApplicationServices.GetRequiredService<IEventBus>();
           eventBus.TopicKeySubscribe<LimitDayMessageEvent,
LimitDayMessageEventHandler>(Exchange, queue, routingKey);
            eventBus.HealthyCheck();
//注：LimitDayMessageEvent对应业务事件模型 LimitDayMessageEventHandler对应业务处理方法
Exchange交换机  queue主题  routingKey
```

5.2.3.事件模型

1.该事件作用域若只存在**自身服务**，模型需在**xxx.Application**类库**IntegrationEvents->EventMessagesModel**文件夹下创建

2.该事件作用域于**全局服务**，模型需在**BuildingBlocks**文件夹**EventBus** 类库**EventMessageModel**文件夹下创建

3.模型命名规则：xxx**MessageEvent**,xxx指业务别名

声明一个**record**记录类，并继承**IntegrationEvent**

例：

```
using EventBus.Events;

namespace EventBus.EventMessageModel.Control
{
    public record ControlMessageEvent : IntegrationEvent
    {
        public List<ControlMessageEventDto> data { get; set; }
    }

    public class ControlMessageEventDto
    {
        /// <summary>
        /// 设备唯一标识（数据库主键）
        /// </summary>
```

```
        public string id { get; set; }
        /// <summary>
        /// 设备唯一标识
        /// </summary>
        public string sn { get; set; }
        /// <summary>
        /// -1 - 注销，  0 - 停用，  1 - 启用
        /// </summary>
        public int command { get; set; }
    }
}
```

5.2.4.消息订阅

1.**xxx.Application**类库**IntegrationEvents->EventHanding**文件夹下创建Handler类

2.命名规则：xxx**EventHandler**,xxx指业务别名

3.继承: **IIntegrationEventHandler**接口，T :业务事件模型

例:

```
using EventBus.Abstractions;
using EventBus.EventMessageModel.Limit;
using Microsoft.Extensions.Logging;

namespace Open.Application.IntegrationEvents.EventHanding
{
    /// <summary>
    /// xxx服务
    /// </summary>
    public class LimitDayMessageEventHandler :
IIntegrationEventHandler<LimitDayMessageEvent>
    {
        private readonly ILogger<LimitDayMessageEventHandler> _logger;

        public LimitDayMessageEventHandler(ILoggerFactory loggerFactory)
        {
            _logger = loggerFactory.CreateLogger<LimitDayMessageEventHandler>();
        }

        public async Task Handle(LimitDayMessageEvent @event)
        {
            _logger.LogInformation($"Subscribe LimitDayEvent  Count:
{@event.data.Count} EventId:{@event.Id}");
            //执行业务逻辑
        }
    }
}
```

5.2.5.消息推送

5.1 路由模式

声明消息模型，赋值交换机，队列，RoutingKey

例：

```
            var event=new PushPersonMessageEvent();
            event.Exchanges = _configuration["RabbitMQ:ExchangeName"];
            event.Queue = EventTopics.Queue_Alarm_Person;
            event.RoutingKey = EventTopics.RoutingKey_Alarm_Person;
            _eventBus.TopicPublish(event);
```

5.2 主题模式

```
var OffsetMessage=new DeviceAbnormalMessageEvent();
_eventBus.Publish(OffsetMessage);
```

5.2.6.业务处理

订阅对应事件模型后，在接收到该事件类型消息时，会自动转为对应Record业务纪录类，并触发相应Handle方法

需在该Handle实现下注入或实现业务流程

5.3.数据仓储

5.3.1.全局配置

```
  "ConnectionString": "",
```

5.3.2.服务注册

startup文件进行注册

```
            var migrationsAssembly =
typeof(Program).GetTypeInfo().Assembly.GetName().Name;
             services.AddDbContext<BaseDbcontext>(options =>
             options.UseMySql(connectionString, new MySqlServerVersion(new
Version(5, 7, 28)), mySqlOptions =>
             {
                 mySqlOptions.MigrationsAssembly(migrationsAssembly);//指定迁移程序
集
                 mySqlOptions.EnableRetryOnFailure(maxRetryCount: 15,
maxRetryDelay: TimeSpan.FromSeconds(30), errorNumbersToAdd: null);//15次错误重试
                 mySqlOptions.CommandTimeout(300);//300ms超时
             })).AddUnitOfWork<BaseDbcontext>();//注册工作单元
```

5.3.3.仓储实现

1.实体&聚合根

xxx.**Domain**作为我们单服务领域层，主要包含**实体**、**聚合根**、**域服务**、**值对象**、**存储库接口**和**其他域对象**，具体详细介绍可在第3章有所介绍

实现一个业务实体：

1.展开该类库下**model**文件夹，创建该业务类

2.继承**AggregateRoot<T,Key>**聚合根 T:实体类型，Key:主键类型

并可根据业务需要继承其它抽象接口

2.仓储

.Domain类库下 展开Repositories文件夹

实现该业务实体仓储层：

1.创建业务仓储接口 命名规则为**IxxxRepository** 并继承**IRepository<T,Key>** T:实体类型，Key:主键类型，xxx:业务实体别名

2.进入**Implements**文件夹下，创建业务仓储实现类 并继承**BaseRepository<T,Key>** T:实体类型，Key:主键类型,同时继承**IxxxRepository**该业务仓储接口

例如：

```
using Domain.Repositories;
using Glasssix.Domain.Abstractions;
using Glasssix.Domain.Abstractions.Implements;

namespace Device.Domin.Repositories.Implements
{
    public class DeviceFlowRepository : BaseRepository<Model.DeviceFlow, long>,
IDeviceFlowRepository
    {
        public DeviceFlowRepository(IUnitOfWork unitOfWork) : base(unitOfWork)
        {
```

```
        }
    }
}
```

3.数据传输对象（Dto）

进入**.Domain.Shared** 类库，Dto通常用于在**应用层**和**表示层**或其他类型的客户端之间传输数据

1.声明业务**Dto**,命名方式通常为**业务实体名+Dto**,需要注意的是该dto需要继承: **DtoBase**类

2.声明业务**Queries**，命名方式通常为**业务实体名+Input** ,需要注意的是该dto需要继承: **QueryParameter**类

4.工作单元

工作单元(UOW)实现提供了对应用程序中的**数据库连接和事务范围**的抽象和控制

需要了解

- 大部分情况下不需要手动处理uow（除手动实现仓储方法外）
- 数据提供者独立，这意味着
- 可在任意方法或实现处注入并使用它

1.可执行手动事务提交

2.取值当前DbContext或Connection

3.受影响行提交

5.应用服务

.Application 主要进行业务应用层逻辑实现

.Application.Contracts 进行业务逻辑接口声明 并作为合约包供他人使用

1.进入**.Application.Contracts**类库，并定义该业务接口 命名规范：IxxxService

2.进入**.Application**类库 创建应用服务并继承该接口

3.注入仓储接口进行使用

需要注意的是，为了简化Curd操作，此处封装了全局增删改查服务,如有业务需要可进行继承

注入**SampleService<T,Key>** T：实体类型 Key:主键类型

- 该类实现增删改查，并支持相应操作前后流程化注入并影响受返回结果
- 无需进行工作单元手动式提交
- 需同步进行redis注入，针对进行全局自动化缓存处理,缓存Key：主键Id
- 对于相应操作可进行重写
- 支持同异步

例如：

创建后操作

```
  protected override async Task CreateAfterAsync(Domin.Model.Device entity)
        {
            await base.CreateAfterAsync(entity);
            return;
        }
```

修改前操作

```
        protected override Task UpdateBeforeAsync(Domin.Model.Device entity)
        {
            return base.UpdateBeforeAsync(entity);
        }
```

进行对应重写即可

注意：IxxxService需继承第5.8章依赖注入-全局注入所述特性

5.4.配置中心

1.本地配置

appsettings.json文件进行编写 并于.**ConfigureAppConfiguration**指定

```
IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .CaptureStartupErrors(false)
        //web服务器端点
        .ConfigureKestrel(options =>
        {
            options.ListenAnyIP(10066, listenOptions =>
            {
                listenOptions.Protocols = HttpProtocols.Http1;
            });
        })
           .ConfigureAppConfiguration(x =>
x.AddConfiguration(GetConfiguration()))//本地配置
        .UseStartup<Startup>()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseSerilog()
        .Build();
```

2.分布式配置中心

此处使用第三方开源组件**apollo**进行分布式配置实现

需安装包 **Com.Ctrip.Framework.Apollo**，**Com.Ctrip.Framework.Apollo.Configuration**

```
IWebHost BuildWebHost(string[] args) =>
```

```
    WebHost.CreateDefaultBuilder(args)
        .CaptureStartupErrors(false)
        //web服务器端点
        .ConfigureKestrel(options =>
        {
            options.ListenAnyIP(10062, listenOptions =>
            {
                listenOptions.Protocols = HttpProtocols.Http1;
            });
        })
        //Apollo获取配置
        .ConfigureAppConfiguration(x => x.AddApollo(new ApolloOptions() { AppId =
"", MetaServer = "Url" }).AddDefault().AddNamespace("",
Com.Ctrip.Framework.Apollo.Enums.ConfigFileFormat.Json))
        .UseStartup<Startup>()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseSerilog()
        .Build();
```

5.5.服务发现

5.5.1.注册中心Consul

在分布式架构中，服务治理是一个重要的问题。在没有服务治理的分布式集群中，各个服务之间通过手工或者配置的方式进行服务关系管理，遇到服务关系变化或者增加服务的时候，人肉配置极其麻烦且容易出错，consul由此而来，直到云原生时代，consul依旧有大多数企业继续使用。

简而言之，Consul是一个用来实现分布式系统的服务发现与配置的开源工具。他主要由多个组成部分：

服务发现：客户端通过Consul提供服务，类似于API,MySQL,或者其他客户端可以使用Consul发现服务的提供者。使用类似DNS或者HTTP，应用程序和可以很轻松的发现他们依赖的服务

检查健康：Consul客户端可以提供与给定服务相关的健康检查（Web服务器返回200 ok）或者本地节点（“内存利用率低于90%”）。这些信息可以监控集群的运行情况，并且使访问远离不健康的主机组件

键值对存储：应用程序可以使用Cousul的层级键值对

多数据中心：Consul有开箱及用的多数据中心

这方面就不多做赘述，感兴趣可下去自己慢慢了解，直接来讲如何使用

1.全局配置

```
{
  "Consul": {
    //是否启用
    "Enabled": false,
    //Consul主机地址
    "Host": "http://xxx.xxx.xx.xx:8501",
    //健康检查地址
    "HealthCheckPath": "hc",
    "App": {
      //应用名称
      "Name": "FaceService-api",
```

```
      //协议
      "Scheme": "http",
      //应用主机地址 必须对应具体IP
      "Host": "xxx.xxx.xx.xx",
      //应用监听端口
      "Port": 5132,
      //标签
      "Tags": [
        "face-api-a"
      ]
    }
  }
}
```

2.服务注入

```
 public virtual IServiceProvider ConfigureServices(IServiceCollection services)
        {
         services.AddConsul(Configuration);
        }
//需要注意，启用Consul必须在管道进行引用
 public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
IHostApplicationLifetime lifetime)
        {
         app.UseConsul(lifetime);
        }
```

5.5.2.基于服务发现的ApI网关Ocelot

尽管我们在技术上已经不再去使用Api网关进行服务暴露，但为了多场景，考虑资源，拓展性方面依旧对传统服务发现+网关进行保留

同样，在使用之前我们先大概了解一下什么是Api网关

API网关是系统暴露在外部的一个访问入口，就像一个公司的门卫承担着寻址、限制进入、安全检查、位置引导、等等功能。从面向对象设计的角度看，它与外观模式类似。API网关封装了系统内部架构，为每个客户端提供一个定制的API。它可能还具有其它职责，如身份验证、监控、负载均衡、缓存、请求分片与管理、静态响应处理等等

API网关方式的核心要点是，所有的客户端和消费端都通过统一的网关接入微服务，在网关层处理所有的非业务功能。通常，网关也是提供REST/HTTP的访问API。服务端通过API-GateWay注册和管理服务，**通常用consul搭配工作**

1.全局配置

```
{
  "Routes": [
    {
      //服务名称，开启服务发现时需要配置
```

```
      "ServiceName": "FaceService-api",
      //是否开启服务发现
      "UseServiceDiscovery": true,
      //下游服务路由模板
      "DownstreamPathTemplate": "/{url}",
      //下游服务http schema
      "DownstreamScheme": "http",
      //下游服务的地址，如果使用LoadBalancer的话这里可以填多项
      //"DownstreamHostAndPorts": [
      //  {
      //    "Host": "192.168.6.199",
      //    "Port": 7012
      //  },
      //  {
      //    "Host": "192.168.6.196",
      //    "Port": 7012
      //  },
      //  {
      //    "Host": "192.168.6.198",
      //    "Port": 7012
      //  }
      //],
      "UpstreamPathTemplate": "/{url}",
      //上游请求http方法，可使用数组
      "UpstreamHttpMethod": [ "GET", "POST", "PUT", "PATCH", "DELETE", "HEAD",
"OPTIONS" ],
      /**
       * 负载均衡的算法:
       * LeastConnection        - 跟踪哪些服务正在处理请求，并将新请求发送到具有最少现有请
求的服务。算法状态没有分布在Ocelot集群中。
       * RoundRobin             - 遍历可用服务并发送请求。算法状态没有分布在Ocelot集群
中。
       * NoLoadBalancer          - 从配置或服务发现中获取第一个可用服务
       * CookieStickySessions    -  使用cookie将所有请求粘贴到特定服务器
       */
      "LoadBalancerOptions": {
        "Type": "LeastConnection"
        ////以下配置再设置了 CookieStickySessions 后需要开启
        ////用于粘性会话的cookie的密钥
        //"Key": "ASP.NET_SessionId",
        ////会话被阻塞的毫秒数
        //"Expiry": 1800000
      },
      ////缓存
      //"FileCacheOptions": {
      //  "TtlSeconds": 15,
      //  "Region": ""
      //},
      //限流
      "RateLimitOptions": {
        //包含客户端白名单的数组。这意味着该阵列中的客户端将不受速率限制的影响
        "ClientWhitelist": [],
        //是否启用端点速率限制
        "EnableRateLimiting": true,
```

```
        //指定限制所适用的期间，例如1s，5m，1h，1d等。如果在该期间内发出的请求超出限制所允许的数量，则需要等待PeriodTimespan过去，然后再发出其他请求
        "Period": "1s",
        //指定可以在一定秒数后重试
        "PeriodTimespan": 2,
        //指定客户端在定义的时间内可以发出的最大请求数
        "Limit": 1000
      },
      //熔断
      "QoSOptions": {
        //允许多少个异常请求
        "ExceptionsAllowedBeforeBreaking": 10,
        //熔断的时间，单位为毫秒
        "DurationOfBreak": 2000,
        //如果下游请求的处理时间超过多少则自如将请求设置为超时
        "TimeoutValue": 8000
      },
      "HttpHandlerOptions": {
        //是否开启路由追踪
        "UseTracing": true
      }
    }
  ],
  "GlobalConfiguration": {
    //请求标识Key
    "RequestIdKey": "OcelotRequestId",
    //Consul服务发现
    "ServiceDiscoveryProvider": {
      //协议
      "Scheme": "http",
      //Consul主机地址
      "Host": "xxx.xxx.xx.xx",
      //Consul主机端口
      "Port": 8501,
      //服务发现提供者类型
      "Type": "Consul"
    },
    //外部暴露的Url
    "BaseUrl": "http://xx.xx.xx.x:xxxx",
    //限流扩展配置
    "RateLimitOptions": {
      //指定是否禁用X-Rate-Limit和Retry-After标头
      "DisableRateLimitHeaders": false,
      //当请求过载被截断时返回的消息
      "QuotaExceededMessage": "Oh,Oops!Your request frequency is too high. Please slow down~",
      //当请求过载被截断时返回的http status
      "HttpStatusCode": 503,
      //用来识别客户端的请求头，默认是 ClientId
      "ClientIdHeader": "ClientId"
    },
    //自定义委托处理程序
    "DelegatingHandlers": [
      "LoggerDelegatingHandler"
    ]
```

```
  }
}
```

2.服务注册

```
            services
            .AddOcelot(Configuration)
            //服务发现
            .AddConsul()
            //缓存
            .AddCacheManager(x => { x.WithDictionaryHandle(); })
            //重试，熔断，降级.....
            .AddPolly()
            .AddDelegatingHandler<LoggerDelegatingHandler>(true);

            //管道声明
             app.UseOcelot().Wait();
```

3.Handle重写

对于ocelot而言，请求被转发给哪个节点是不透明的，我们需要对此进行重写

```
using System.Net.Http.Headers;
using System.Text;

namespace FaceDetection.Gateway.Handlers
{

    public class LoggerDelegatingHandler : DelegatingHandler
    {
        private readonly ILogger<LoggerDelegatingHandler> Logger;

        public LoggerDelegatingHandler(ILogger<LoggerDelegatingHandler> logger)
        {
            this.Logger = logger;
        }

        protected override async Task<HttpResponseMessage>
SendAsync(HttpRequestMessage request,
            CancellationToken cancellationToken)
        {
            HttpRequestHeaders headers = request.Headers;

            HttpResponseMessage response = await base.SendAsync(request,
cancellationToken);

            StringBuilder builder = new StringBuilder();
            foreach (KeyValuePair<string, IEnumerable<string>> header in
headers)
            {
```

```
                builder.AppendLine(header.Key + " --> " + string.Join(", ",
header.Value) + "</br>");
            }

            //请求服务地址
            builder.AppendLine("RequestUri --> " + request.RequestUri.ToString()
+ "</br>");

            // //请求消息
            // string content = await request.Content.ReadAsStringAsync();
            // builder.AppendLine("请求消息 --> " + content + "</br>");
            //
            // //响应消息
            // string result = await response.Content.ReadAsStringAsync();
            // builder.AppendLine("响应消息 --> " + result + "</br>");

            this.Logger.LogInformation(builder.ToString());

            return response;
        }
    }
}
```

5.5.3.服务网格ServiceMesh

其实，Servicemesh 只是一个通信上的解决方案的名词，并不具有任何实质性功能，之所以要放到服务发现下来讲，是想读者能通过此有个系统性的体系认知。

Service Mesh又译作“服务网格”，作为**服务间通信的基础设施层**，如下定义Service Mesh

Service Mesh 用于**处理服务间通信**。云原生应用有着复杂的服务拓扑，Service Mesh 保证请求可以在这些拓扑中可靠地穿梭。在实际应用当中，Service Mesh 通常是由一系列轻量级的网络代理组成的，它们与应用程序部署在一起，但应用程序不需要知道它们的存在，在此会引出一个新的概念，**Sidecar**。

Service Mesh 实际上就是处于 TCP/IP 之上的一个抽象层，它假设底层的 L3/L4 网络能够点对点地传输字节（当然，它也假设网络环境是不可靠的，所以 Service Mesh 必须具备处理网络故障的能力）

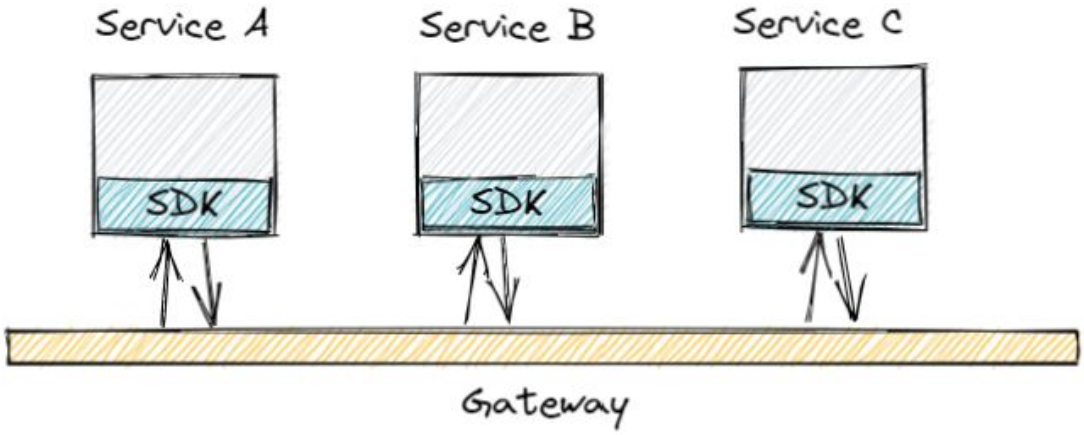
Servicemesh 有如下几个特点：

1. 应用程序间通讯的中间层；
2. 轻量级网络代理；
3. 应用程序无感知；
4. 解耦应用程序的重试、超时、监控、追踪和服务发现；

那么什么是**Sidecar**,sidecar意为边路模式，想一下那种老式三轮右侧多带一座的摩托车，就了解这个模式的由来了

我们先来说**场景**

在微服务架构中，如果应用多了就会形成一些共有的需求。特别是流量控制方面，包括限流、流量分发和监控、灰度等等。通常我们对一类需求可以实现一个抽象层，然后在这个抽象层上实现具体的业务逻辑。比如服务网关，然后使用各种语言的sdk来集成到应用中。



这是通常我们会选择的一种方式，这过程中会有这样的一些问题需要考虑：

1. sdk的维护成本是很高
2. sdk集成到代码中，其中一个组件发生故障就可能会影响到其他组件，sdk和应用程序之间是保持着相互依赖的关系的。

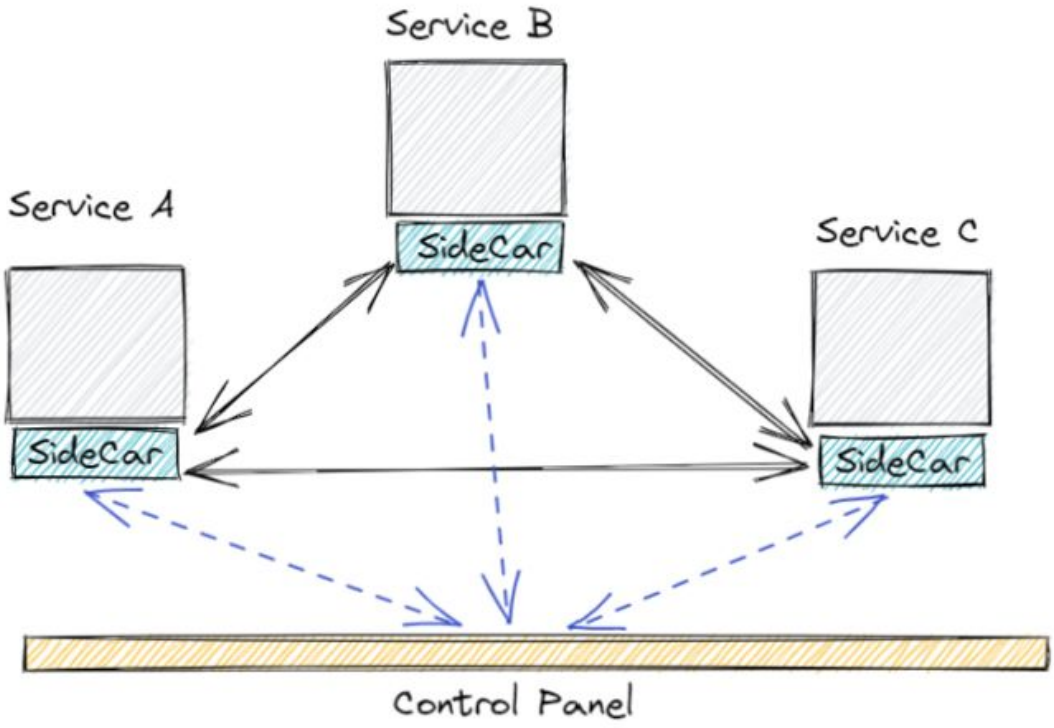
在应用层和基础服务没有解耦的情况下，我们对基础服务做改动会增加很多风险和复杂度

由此延申出的是：**Sidecar架构**

那我们是否可以提供一个统一的抽象层来做这些基础的重复工作？将基础服务抽象、解耦到应用层都感知不到的程度

这是现在的趋势，特别是现在很多架构都跑在容器这样的环境了，统一的抽象层能大大减少架构上的复杂度。 sidecar 模式在不改变主应用的情况下，会起来一个辅助应用，来辅助主应用做一些基础性的甚至是额外的工作。这个 sidecar 通常是和主应用部署在一起，所以在同样的运行环境下。这其中还有一些性能上的考虑，sidecar 如果和主程序网络通信上有延迟就会造成性能问题。例如：在 K8s 下一个 pod 里的所有子应用共享一个 sidecar 服务。

这个辅助应用不一定属于应用程序的一部分，而只是与应用相连接。这就像是挎斗摩托车，每个摩托车都有自己独立的辅助部分，它随着主应用启动或停止。因为 sidecar 其实是一个独立的服务，我们可以在上面做很多东西，例如 sidecar 之间相互通信、或者通过统一的节点控制 sidecar ，从而达到 Service Mesh



由此，大名鼎鼎的**Istio**也是应运而生，也就是目前我们生产环境所使用的

这里就不讲如何使用及学习了解，这通常需要了解到更多的云原生系列理论知识，这里这对这个思想进行说明和概况，感兴趣的可以自己深入并了解

5.5.4.网关限流/熔断/降级/会话保持/黑名单

此处策略针对网关层，故只进行理论说明，功能上从ocelot也好Istio也好，相对于开发而言，重要的了解并理解

1.限流

在高并发系统中一定要用，高并发的所有请求进来，不是让每个请求都打到后台集群的，后台集群有它的消费能力，我们应该在它消费能力之内，给它放行请求进来，这个就是限流

比如我们整个集群的处理能力就是每秒1w，那我们从网关处给你放过来的请求那就是1w，保证我们服务不会被超过它能力的流量压垮，只要超过它能力的流量，我们就直接丢弃，你也不用处理了

2.熔断

就跟电路里面的保险丝短路的保护一样，A服务调用B服务的某个方法，但是由于B服务的网络不稳定或者B服务方法本来执行慢，或者B服务连接的数据库卡慢，导致方法慢，导致我们整个调用链慢，或者B服务直接宕机了。

我们A服务要去调用的话，如果是以前，我们没有任何保护，直接用feign调用，feign总是给B服务发送请求试，试了一段时间以后，有个默认超时时间，比如3秒，3秒时间你不返回，我就认为你这个服务出问题了，feign接口就会报错。

但是我们现在等不了这么久，因为这样就会引起我们整个调用链的累积效应。那一个要等，大家都要等，等着等着全线卡死，资源不能得到及时释放，吞吐量下降，大量请求又在这里排队，这就形成了一个死循环，处理能力越不行，外面累积的越多，越多的请求又需要越多的资源来进行分配处理，那我们机器就会卡死宕机

3.降级

现在有超多的业务都在运行，一些核心的业务：购物车，订单等等，还有一些非核心的业务比如注册之类的，现在网站正在进行秒杀，是一个流量高峰时期。
大家资源不够用了，我们可以手工的将一些非核心业务，比如注册，我们将放注册服务的服务器上的注册服务直接停掉，因为这个服务器不止运行了注册业务，可能还有其他业务。这样停掉以后，就把大量的资源又腾给这个服务器的其他业务了，这就叫降级，停止服务就是降级运行

4.会话保持

网关上会话保持现大多基本为基于源地址Ip哈希值记录，来做到类似链表字典上，相同Ip的连接可以做到复用，这个字典长度通常是可配置的

5.黑名单

故名思意被拉黑的请求源，原理于会话保持大多相同

5.6.链路追踪

此处详细说明可前往4.5章阅读

执行以下步骤进行包引用及配置文件生成

- **dotnet tool install -g SkyAPM.DotNet.CLI**
- **dotnet add package SkyAPM.Agent.AspNetCore**
- **dotnet skyapm config [service name] [IP]:11800**

  注：service name服务名称

配置环境变量

"ASPNETCORE_HOSTINGSTARTUPASSEMBLIES": "SkyAPM.Agent.AspNetCore"

5.7.审计日志

1.本地审计

一个**审计日志对象**通常是针对每个web请求创建和保存的.包括;

- **请求和响应的细节** (如URL,HTTP方法,浏览器信息,HTTP状态代码...等).
- **执行的动作** (控制器操作和应用服务方法调用及其参数).
- **实体的变化** (在Web请求中).
- **异常信息** (如果在执行请求发生操作).
- **请求时长** (测量应用程序的性能)

1.启用审计日志服务

```
services.AddControllers(builder =>
          {
              builder.Filters.Add(typeof(ApplictionRequestLogFilter));
          });
```

2.Es采集审计

基于本地日志审计可以满足我们大多数场景的需要，但是如果是分布式微服务，并且存在高并发场景的情况下，本地审计就有些心有余而力不足了

- 存储问题，大量日志消耗一定存储空间并且不易被回收
- 分布式下难于基于本地日志进行统计，这通常需要其他第三方组件进行配合
- 通常带来一定范围的性能影响，并且并发越搞损耗越严重

为此，通常情况分布式下做法是，链路追踪，并将本地全局日志送往第三方存储中心进行统一，这里可以是Es,Mango，loki等等。

1.启用

```
services.AddEsAuditing();
app.UseEsAuditing();
```

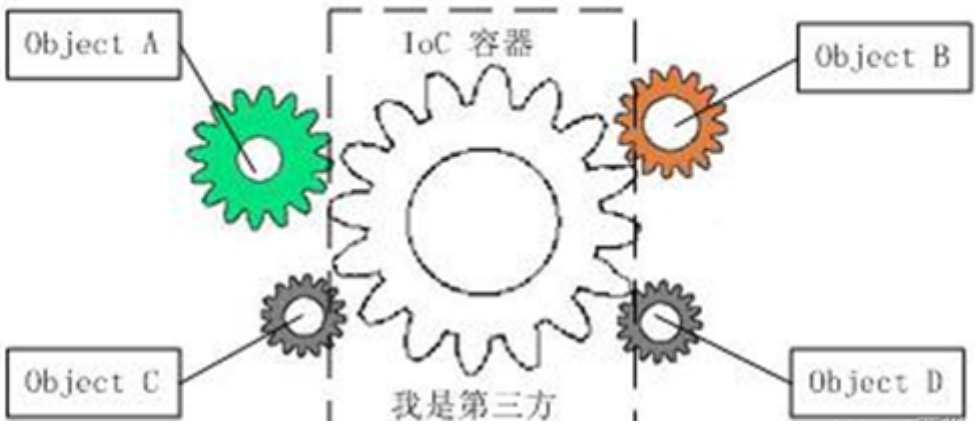
5.8.依赖注入

5.8.1.Ioc

那首先先了解**Ioc**基本理念

我们知道在面向对象设计的软件系统中，它的底层都是由N个对象构成的，各个对象之间通过相互合作，最终实现系统的业务逻辑

对于面向对象设计及编程的基本思想，简单来说就是把复杂系统分解成相互合作的对象，这些对象类通过封装以后，内部实现对外部是透明的，从而降低了解决问题的复杂度，而且可以灵活地被重用和扩展，从而引进了中间位置的“第三方”，也就是IOC容器



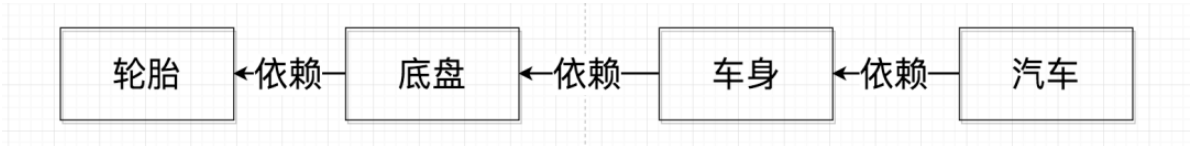
我们知道 Ioc也常被成为**控制反转**，**依赖注入**

要了解控制反转需要先了解软件设计的一个重要思想**依赖倒置原则**

打个比方说：

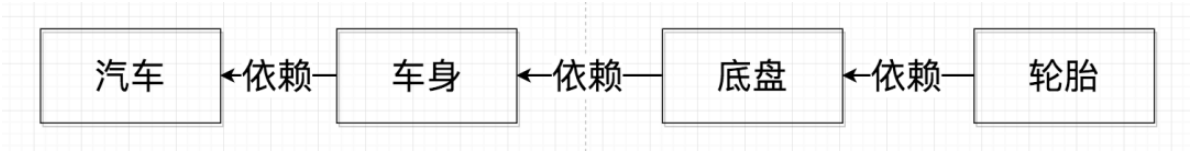
我们需要设计一辆汽车，先设计轮子，然后根据轮子设计地盘，再根据地盘设计车身，根据车身设计整个汽车，那这里就有了依赖关系：

汽车依赖车身，车身依赖地盘，地盘依赖轮子



这个时候，上司说换个轮胎，那是不是基于轮子的全部要推到重来，本身这样的设计看起来没什么问题，但是可维护性确很低

我们现在换一种思路。我们先设计汽车的大概样子，然后根据汽车的样子来设计车身，根据车身来设计底盘，最后根据底盘来设计轮子。这时候，**依赖关系就倒置过来了**：轮子依赖底盘，底盘依赖车身，车身依赖汽车



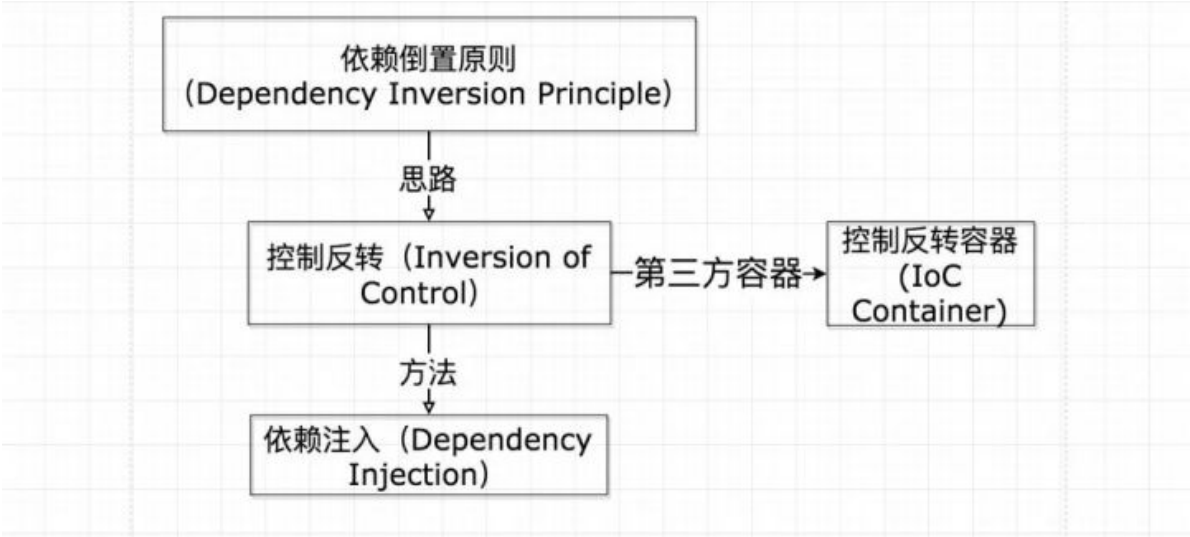
这时候，上司再说要改动轮子的设计，我们就只需要改动轮子的设计，而不需要动底盘，车身，汽车的设计了

这就是依赖倒置原则 — **把原本的高层建筑依赖底层建筑"倒置"过来，变成底层建筑依赖高层建筑。高层建筑决定需要什么，底层去实现这样的需求，但是高层并不用管底层是怎么实现的。**这样就不会出现前面的"牵一发动全身"的情况

控制反转 就是依赖倒置原则的一种具体的设计思路，或者说是一种可行的思想。而 IoC 的具体实现方法呢，就是 **依赖注入**

所谓依赖注入，就是把底层类作为参数传入上层类，实现上层类对下层类的控制，可参考本章第一张图进行理解

这几种概念的关系大概如下：



5.8.2.AutoFac

众所周知，.net本身内置容器功能非常有限，作为第三方Ioc翘楚的Autofac，也有必要在此进行一下使用说明及方便我们更快于适应当前这个架构

1.容器声明

```
            #region AutoFac
            var container = new ContainerBuilder();
            container.Populate(services);
            var privider = new AutofacServiceProvider(container.Build());
            return privider;
```

2.获取当前作用域

```
            var iLifetimeScope =
IServiceProvider.GetRequiredService<ILifetimeScope>();
```

根据业务需求我们可以通过当前作用域来实现对各个对象或实例解析或操作

5.8.3.Location Ioc

除了AutoFoc，最早实现了一版基于自身Ioc容器的封装，目前数据仓储应用层这块对此有所依赖，所以在使用时务必进行默认ioc注册

```
            #region AutoFac
            var container = new ContainerBuilder();
            container.Populate(services);
            var privider = new AutofacServiceProvider(container.Build());
            ServiceFactory.RegisterIoc(privider);//默认Ioc注入
            return privider;
```

5.8.4.全局注入

此处实现三个特性接口，对服务注册时只需在业务接口上声明相应特性接口

1.启用

```
new DependencyRegistrator(services).RegisterServices();
```

5.8.5.反射

反射机制是在运行状态中，对于任意一个实体类，都能够知道这个类的所有属性和方法;对于任意一个对象，都能够调用它的任意方法和属性;这种动态获取信息以及动态调用对象方法的功能称为的反射机制

我们也多常用来根据某名称获取某个类，例如根据特性获取对应类

1.启用

```
new DependencyRegistrator(services).RegisterServices();
```

2.自定义反射

DependencyRegistrator类中实现 获取类型集合方法，可以通过对应type解析抓取相应结果或实例

```
        /// <summary>
        /// 获取类型集合
        /// </summary>
        private Type[] GetTypes<T>()
        {
            return TypeFinder.Find<T>(Assemblies).ToArray();
        }
```

5.9.缓存

缓存对于每一个开发都耳熟能详，这里不在对理论多做赘述，在使用上，框架只保留分布式缓存

1.启用

```
            #region Redis
            services.AddSingleton<ConnectionMultiplexer>(option =>
            {
                return
RedisSentinelConnection.StackExchangeRedis(Configuration);
            });
            services.AddSingleton<RedisClient>();
            #endregion
```

使用方式**RedisClient**进行构造注入使用即可

2.启用布隆过滤器

```
            #region Bloom//注入布隆过滤器
            services.AddBloomFilter(setup =>
            {
                setup.UseRedis(Configuration["RedisConnection"]);
            });
            services.AddSingleton<RedisBloomFilter>();
            #endregion
```

关于布隆过滤器原理在第三章有详细说明，可移步查看

5.10.限流

在5.5服务发现中我们有讲针对网关层的限流，这里我们讲对服务的限流实现

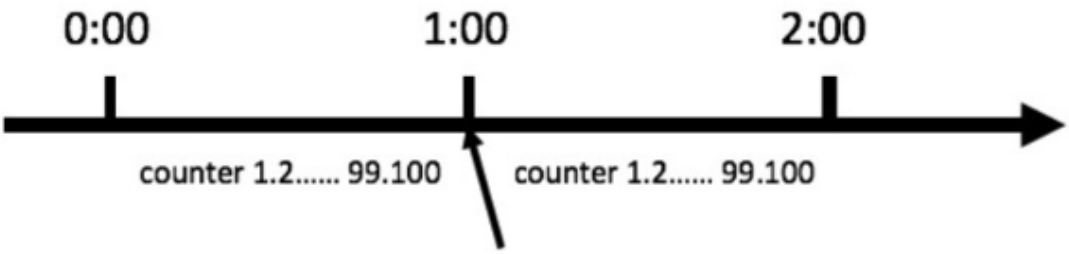
目前，针对服务主要有三种限流算法

1.计算器算法

计数器算法是限流算法里最简单也是最容易实现的一种算法。比如我们规定，对于A接口来说，我们1分钟的访问次数不能超过100个

那么我们可以这么做：在一开 始的时候，我们可以设置一个计数器counter，每当一个请求过来的时候，counter就加1，如果counter的值大于100并且该请求与第一个 请求的间隔时间还在1分钟之内，那么说明请求数过多；

如果该请求与第一个请求的间隔时间大于1分钟，且counter的值还在限流范围内，那么就重置 counter



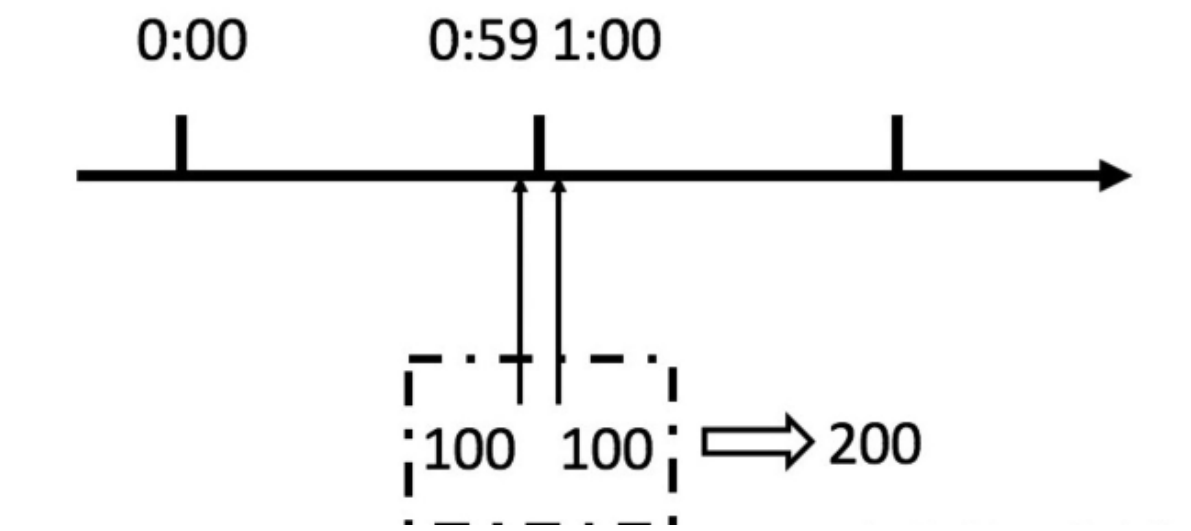

```
public class CounterTest {
    public long timeStamp = getNowTime();
```

```
    public int reqCount = 0;
    public final int limit = 100; // 时间窗口内最大请求数
    public final long interval = 1000; // 时间窗口ms

    public boolean grant() {
        long now = getNowTime();
        if (now < timeStamp + interval) {
            // 在时间窗口内
            reqCount++;
            // 判断当前时间窗口内是否超过最大请求控制数
            return reqCount <= limit;
        } else {
            timeStamp = now;
            // 超时后重置
            reqCount = 1;
            return true;
        }
    }

    public long getNowTime() {
        return System.currentTimeMillis();
    }
}
```

但是此种算法存在临界问题



从上图中我们可以看到，假设有一个恶意用户，他在0:59时，瞬间发送了100个请求，并且1:00又瞬间发送了100个请求，那么其实这个用户在 1秒里面，瞬间发送了200个请求。

我们刚才规定的是1分钟最多100个请求，也就是每秒钟最多1.7个请求，用户通过在时间窗口的重置节点处突发请求， 可以瞬间超过我们的速率限制。用户有可能通过算法的这个漏洞，瞬间压垮我们的应用

2.**滑动窗口**

大抵于时间算法相同，只是更为精细，

在下图中，整个红色的矩形框表示一个时间窗口，在我们的例子中，一个时间窗口就是一分钟。

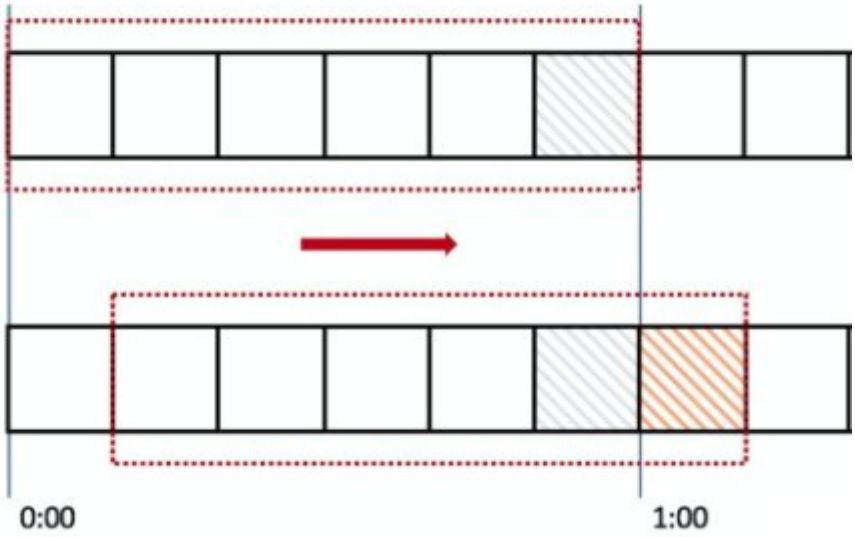
然后我们将时间窗口进行划分，比如图中，我们就将滑动窗口划成了6格，所以每格代表的是10秒钟。每过10秒钟，我们的时间窗口就会往右滑动一格。

每一个格子都有自己独立的计数器counter，比如当一个请求 在0:35秒的时候到达，那么0:30~0:39对应的counter就会加1

那么滑动窗口怎么解决刚才的临界问题?

我们可以看上图，0:59到达的100个请求会落在灰色的格子中，而1:00到达的请求会落在橘黄色的格 子中。

当时间到达1:00时，我们的窗口会往右移动一格，那么此时时间窗口内的总请求数量一共是200个，超过了限定的100个，所以此时能够检测出来触 发了限流

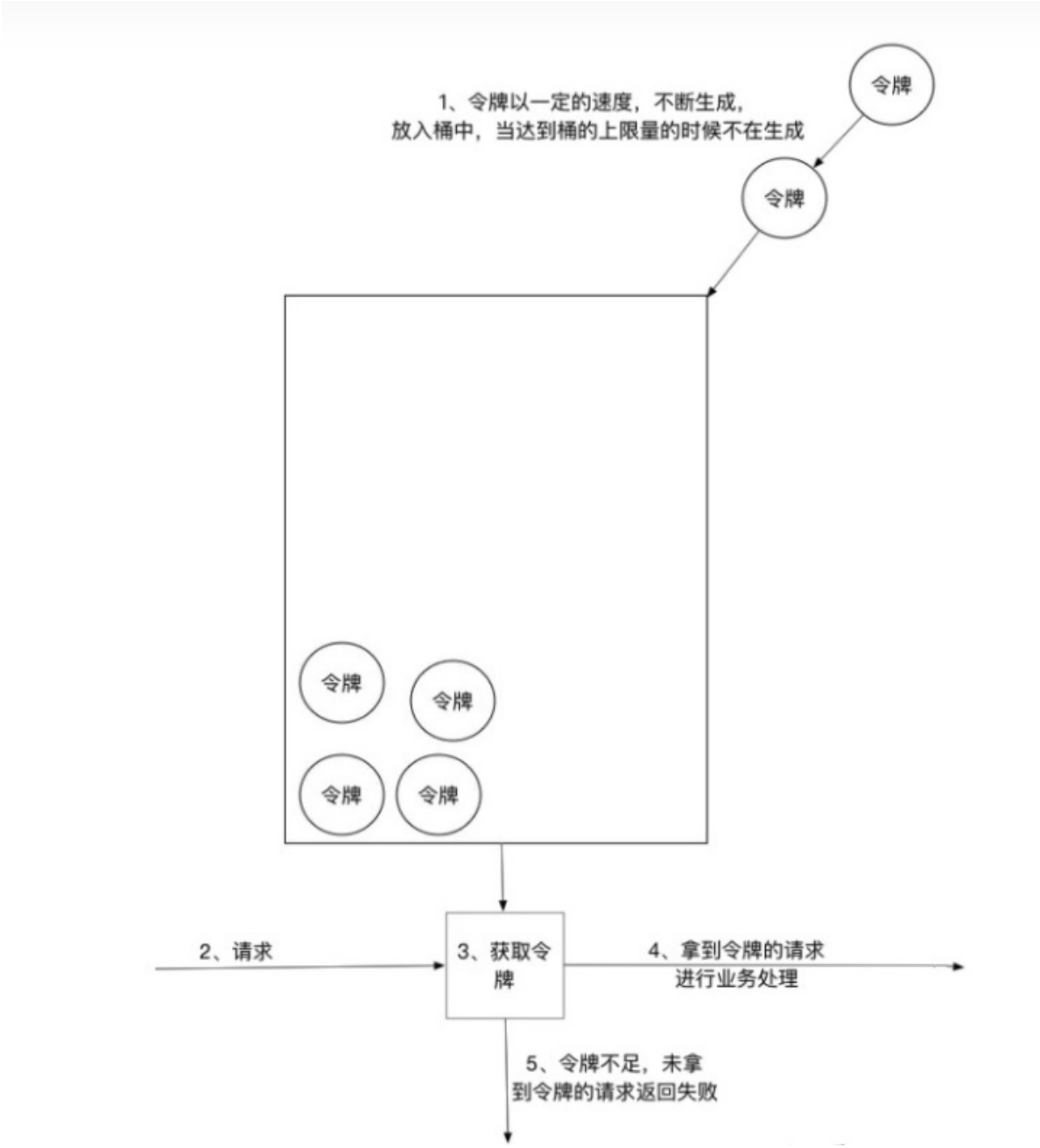


3.**令牌桶**

令牌桶算法是比较常见的限流算法之一

大概描述如下：

1. 所有的请求在处理之前都需要拿到一个可用的令牌才会被处理；
2. 根据限流大小，设置按照一定的速率往桶里添加令牌；
3. 桶设置最大的放置令牌限制，当桶满时、新添加的令牌就被丢弃或者拒绝；
4. 请求达到后首先要获取令牌桶中的令牌，拿着令牌才可以进行其他的业务逻辑，处理完业务逻辑之后，将令牌直接删除；
5. 令牌桶有最低限额，当桶中的令牌达到最低限额的时候，请求处理完之后将不会删除令牌，以此保证足够的限流；



当前框架所采用的限流算法便是这种

1.配置

```
"RedisTokenBucketAlgorithmOptions": {
  "Capacity": "20000",// 令牌桶容量
  "InflowQuantityPerUnit": 2000, // 单位时间流入量
  "InflowUnit": 10,  // 流入铲斗的时间单位,秒计算
  "LockSeconds": 10  // 触发速率限制后锁定的秒数。0表示未锁定
},
```

2.启用

```
string redis = Configuration["RedisConnection"];
        RedisTokenBucketAlgorithmOptions redisTokenBucketAlgorithmOptions =
```

```
 Configuration.GetSection("RedisTokenBucketAlgorithmOptions").Get<RedisTokenBuck
etAlgorithmOptions>();

            services.AddRateLimit(new RedisTokenBucketAlgorithm(
                new[]
                {
                    new
TokenBucketRule(redisTokenBucketAlgorithmOptions.Capacity,
                        redisTokenBucketAlgorithmOptions.InflowQuantityPerUnit,

 TimeSpan.FromSeconds(redisTokenBucketAlgorithmOptions.InflowUnit))
                    {
                        Name = "default limit rule",
                        ExtractTarget = context =>
                        {
                            HttpContext httpContext = (HttpContext) context;

                            return httpContext.Request.Path.Value;
                        },
                        CheckRuleMatching = context =>
                        {
                            HttpContext httpContext = (HttpContext) context;

                            return true;
                        },
                        LockSeconds =
redisTokenBucketAlgorithmOptions.LockSeconds
                    }
                },
                RedisSentinelConnection.StackExchangeRedis(Configuration)
            ));
            //管道启用
            app.UseRateLimit();
```

5.11.Map

使用方式

```
using Glasssix.AspNetCore.Mappers;
var result = entity.MapTo<EntityDto>();
```

5.12.Rpc

5.12.1.ProtoFile声明

1.展开**Grpc.protocol**类库下**Proto**文件夹

2.新建Proto文件 声明文件头

```
syntax = "proto3";//版本3
option csharp_namespace = "Grpc.DeviceGrpc";//命名空间
package DeviceBaseObject;//包名
import  "core.proto";//引用core.proto文件 引用后可以使用其中的message
```

3.声明方法

```
//定义服务名称为DeviceServers
service DeviceServers {
  rpc GetList (DevicePageInput) returns(Result);//协议为rpc，方法名为GetList 输入模型：DevicePageInput
}

//定义消息体
message DevicePageInput{
    int32 SkipCount=1;//int类型
    int32 MaxResultCount=2;
}
//返回消息体
message Result {
  int32 code = 1;
  string message=2;
  bytes data=3;
  bool success=4;
}
//注：rpc协议消息体类型与c#有出入，涉及到结构转换需参考protobuf3官网数据结构列表参考
https://developers.google.com/protocol-buffers/
```

保存进行生成

5.12.2.服务端

1.进行proto文件引用 声明类型为**Server**

```
    <ItemGroup>
        <Protobuf Include="..\..\..\Grpc.Proto\Grpc.protocol\Protos\*.proto"
GrpcServices="Server" />
    </ItemGroup>
```

2.展开.**Application**类库下Grpc文件夹

3.创建服务端Class，继承上述声明服务，例如：

```
using Device.Application.Contracts;
using Glasssix.AspNetCore.Mappers;
using Google.Protobuf;
using Grpc.Core;
using Grpc.DeviceGrpc;
using Helper.Grpc.Core;
using Microsoft.Extensions.Logging;
using System.Text.Json;
```

```
namespace Device.Application.Grpc
{
    /// <summary>
    /// 示例服务Rpc服务端
    /// </summary>
    public class GrpcDeviceServer : DeviceServers.DeviceServersBase
    {
        private readonly ILogger<GrpcDeviceServer> _logger;
        private readonly IDeviceRelationServices _services;
        private readonly IDeviceServices _service;
        public GrpcDeviceServer(ILogger<GrpcDeviceServer> logger,
 IDeviceServices service)
        {
            _logger = logger;
            _service = service;
        }
        //进行方法重写
        public override async Task<Result> GetList(DevicePageInput request,
ServerCallContext context)
        {
            //业务处理
            return ReturnResult(result);
        }
    }
}
```

5.12.3.客户端

1.进行proto文件引用 声明类型为**Client**

```
    <ItemGroup>
        <Protobuf Include="..\..\..\Grpc.Proto\Grpc.protocol\Protos\*.proto"
GrpcServices="Client" />
    </ItemGroup>
```

2.注入对应服务Client类，例如

```
        private DeviceServers.DeviceServersClient _Client { get; set; }
        private IServiceProxy _proxy;
        public DeviceController(IServiceProxy _proxy)
        {
            _Client = _proxy.CreateService<DeviceServers.DeviceServersClient>
(new Uri("https://localhost:7324"));
        }

        /// <summary>
        /// 分页
        /// </summary>
        /// <param name="input"></param>
        /// <returns></returns>
        [HttpPost, Route("pagelist")]
```

```
        public async Task<IActionResult> GetPageListAsync(DevicePageInput input)
        {
            var result = await this._Client.GetListAsync(input);
            var data = await
System.Text.Json.JsonSerializer.DeserializeAsync<PagerList<DeviceRelationDto>>(
               new MemoryStream(result.Data.ToByteArray()));
            return Success(data);
        }
```

5.12.4.注入

```
   app.UseEndpoints(endpoints =>
            {
                endpoints.MapGrpcService<GrpcDeviceServer>();
            });
```

5.12.5.明文

针对多端对接下，阅读沟通问题，开出类似Swagger的Proto文档

浏览器打开输入对应站点 + **_proto**

例如：Http://localhost:10067/_proto即可

5.13.Orm

5.13.1.Dapper

1.全局配置

```
"ConnectionString": "",
```

2.启用

```
services.AddDapper();
new DependencyRegistrator(services).RegisterServices();//需要启用反射
```

3.配置分表策略

需要在业务实体上继承 **[Submeter]**特性，可传入对应时间String格式，不传默认为框架设定固定时间

例如：

```
    [Table("Record")]
    [Submeter("20111201")]
    public class Record : AggregateRoot<Record, long>
    {
    }
```

配置后，使用Dapper进行操作时，框架会自动检测是否具有分表，存在分表数据默认于最新表处理，若最新表无此数据，则往后遍历

4.启用自动分表

```
app.ApplicationServices.UseAutoDapperTableCreate();//固定每25天进行表创建
```

框架根据**Submeter**特性找出所有相关实体，并在每个25天后进行表创建

5.使用

```
using Dapper.Client;
using Device.Domin.Model;
using Device.HttpApi.Client.Dto.HealthReponse;
using EventBus.Abstractions;
using EventBus.EventMessageModel.Push;
using EventBus.EventMessageModel.Record;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;
using SDK.Core.Extensions;

namespace RiskManage.Application.IntegrationEvents.EventHanding
{

    public class RecordsEventHandler : IIntegrationEventHandler<RecordsEvent>
    {
        private readonly ILogger<RecordsEventHandler> _logger;
        private readonly IDapper _dapper;
        public RecordsEventHandler(
            ILoggerFactory loggerFactory,
            IDapper dapper)
        {
            _logger = loggerFactory.CreateLogger<RecordsEventHandler>();
            _dapper = dapper;
        }

        public async Task Handle(RecordsEvent @event)
        {
                _logger.LogInformation($"Subscribe RecordsEvent Sn:{@event.SN}
EventId:{@event.Id}");
                var record = new Record()
                //添加
                await _dapper.InsertAsync<Record>(record);
                //其他使用方式均相同
        }
    }
```

```
}
```

5.13.2.SqlSugar

SqlSugar专为大批量数据操作设计，在写入方面性能优于数据仓储，但是只适用于数据结构单一的数据，这里需要综合业务需求，若对数据写入sql有一定复杂度，建议使用Dapper进行cmd操作

1.配置

```
"ConnectionString": "",
```

2.注册

```
services.AddSqlSugar(Configuration);
```

3.使用

```
    using (var con = SqlSugarHelper.GetClient())
           {
               con.Updateable<T>(obj);//修改 其他使用方式相同
           }
```

6.演进规划

...Todo