# 微服务框架 **2.0**，框架拆解设计

**目的：解耦。**

包括微服务实例和框架之间的解耦，框架内部插件的解耦。

1、不同微服务实例可以选择不同的插件进行依赖，比如有的微服务实例需要数据库，有的需要 mongdb。

2、不同微服务实例可以选择框架的不同版本进行依赖。

3、不同微服务实例可以选择插件的不同版本进行依赖。当插件版本升级时，方便依赖了该插件的微服务实例统一升级。

**不解耦带来的问题：**

**微服务框架 1.0：只提供一个框架，微服务实例直接在框架上开发。**

1、 框架代码升级了，或者其中某个插件升级了，而集成了框架的实例有很多，不方便升级。需要一个个手动改代码。

2、 微服务实例并不需要框架的所有插件和功能。

**插件/代码属于哪一类**

1、比如 util 包，可以直接放在核心插件。

2、比如 Swagger，每个微服务实例都需要，有共用的配置项，也有自定义的配置项。一种做法是放到核心插件，并提供配置项和默认值。

第二种做法是放到微服务实例，通过自动生成。

在插件工程中，@Value 使用默认值，不提供默认配置文件。在微服务实例工程中使用配置文件。

**核心插件 txw-boot-starter**

微服务实例 txw-framework-sample

必选插件 txw-A1-starter

可选插件 txw-B1-starter

**日志监控 txw-logstash-starter**

必选插件+配置开关

配置开关没有实现。

未打通 kafka。

**缓存 txw-redis-starter**

可选插件 txw-redis-starter

**消息组件 txw-kafka-starter**

可选插件 txw-kafka-starter

**数据库 txw-mybatis-starter**

可选插件 txw-mybatis-starter

**服务注册 txw-consul-starter**

可选插件 txw-consul-starter

~~可选插件 txw-actuator-starter~~

微服务实例可以选择不同的服务注册插件集成。

服务实例集成测试。

**运行监控 actuator**

直接放到核心插件

验证：http://localhost:8080/actuator/health

**Token 校验（身份校验）**

直接放到核心插件。

未集成。

**统一的异常处理**

直接放到核心插件。

**GlobalDefaultExceptionHandler 类&BusinessException 类等**

使用@ControllerAdvice 注解，可以对所有抛到 controller 类的异常，实现全局的异常拦截处理。

https://blog.csdn.net/qq_21492635/article/details/89381204

**国际化 locale**

直接放到核心插件。

**统一的服务端返回对象 ServiceData**

直接放到核心插件。

**Swagger**

必选插件，被 txw-boot-starter 依赖，配置类 Swagger2Config.java，配置文件 swagger2.properties。

## 兆日的微服务框架分析

微服务框架的实施的组织形式有两种。一种是拆分出多个组件，比如消息，缓存等，这些都是基于开源中间件，**根据公司的业务需要，做了定制化的二次封装**。每个组件都是一个工程。可以打成 jar 包，存放到本地的 Maven 库。业务的微服务过程使用时，从本地 Maven 库，引入这些依赖。其中有几个 jar 包，比如 base、zookeeper 等，是必须引入的。

**这种做法的好处是实现了组件的单一职责，又实现了定制化，每个组件独立迭代，互不影响。更好的代码封装，组件之间更低的耦合。**
**从业务的微服务的角度，实现了即插即用，业务微服务需要什么组件，就引入对应的依赖 jar 包即可。**

第二种方式是提供一个微服务框架的工程，其中集成了常用的微服务中间件，比如消息，缓存等，通过配置文件，进行配置和开关。

实际上应用的是混合方式，即提供一个框架的主工程，也提供了即插即用的组件。提供主工程的目的是集成了一些必选的组件，和强制的规范。比如身份认证，国际化，参数校验，健康检查等。做到必选可配。

因为每一个用户，比如北京用户、上海银行等，都要求单独部署，需求存在大同小异的定制化，部署环境也复杂多变。合理的方案是根据用户，从主干拉出分支版本，比如北京银行就是一个分支，单独迭代，单独维护，单独部署。分支之间互不影响。除非是共性的问题或者 bug，分支不合并入主干版本。版本会抽象出各分支遇到的问题，选择性进行迭代升级，对分支进行支撑。

SSP 层的划分，相当于把 Controller 层和 Service 层和 Dao 层划分成 3 个微服务。其实是不合理的。
比如短信服务的拆分，也是把接入服务单独拆出来，把服务层，比如预约服务、通道选择服务、通道发送服务，拆出来。这是因为这些服务面对的访问的**承载比例不是 1:1 的关系**，而是有系数关系。比如 100w 的高并发，接收服务面对的承载就是 100w，可能需要扩充到 100 个实例，而预约服务的承载量可能只有 1w，1 个实例就可以满足。而发送服务是以通道为单位，100w 的请求量，分流到通道 A 是 90w，分流到通道 B 是 10w，通道 A 的发送服务需要扩充到 90 个实例，通道 B 需要扩充到 10 个实例，这个是不断变化的。同样的，100w 的请求，经过校验处理，可能剩下的有 80w，所以通道选择只需要 80 个实例。
**总而言之，这种拆分用于业务场景有一定的复杂度，服务之间存在分流的场景。**
如果 Controller 层和 Service 层之间，场景比较简单，复杂度不高，服务之间承载是 1:1 的关系，则不需要拆分，而是应该放在一起作为一个微服务。用进程内的方法调用代替进程间调

用，减少了进程间的调用，不论是 Http 还是 Thrift。拆分出来后增加的进程间调用是多此一举。

**从上面抽象出原则或公式：承载比例是否 1:1，业务是否具有一定的复杂性，是否存在着业务分流的可能。**

## 华侨永亨的微服务框架分析

**Root library**

统一的日志处理。基于 slf4j。

**Microservice Core Library**

统一的请求和返回，统一的校验（比如请求头的参数校验）；

日志配置，请求和返回的日志记录，业务活动的日志记录；

统一的异常处理；

入口 Controller.java，request，headers

在 makeApiCall()中，把 request 和 headers 构造出 DCExchange。

其中 request 允许用户通过继承 DCRequest 去扩展。

然后通过 dcExchange.getRequest()获取 request。

在 DCServiceController 的 makeApiCall()，统一实例化 DCExchange，校验，业务处理，返回参数。

**问题：在框架中使用了太多继承，导致耦合度高，扩展性不够。**

AcoServiceController extends DCServiceController

CaseCreationApiRequest extends DCRequest

## 华侨永亨的 IBM 微服务框架分析

问题：目前该框架的完成度不够，未达到投入使用的程度。

1、目前该框架包括请求校验，健康检查，异常处理，封装了 Rest 和 Soap 调用，消息组件封装了 Kafka 和 mq。

未考虑身份认证和安全校验。

消息组件未考虑消息监听和出错重试等。

2、框架设计存在不合理

框架把 Rest 和 Soap 调用封装到一个插件。实际上两种调用方式属于不同的业务场景，不应该封装在一起。

作为一个框架，需要相当的完成度，也要考虑扩展性，才可以投入实际中使用。

20200331，把定时任务都放到 ms-batch，这不是一个好的设计，应该把各业务拆分出去，引入 quartz 这样的开源组件。

**框架的组织结构**

**1、spring-boot-core**

**1） 拦截器 CommonValidationInterceptor（继承 HandlerInterceptor）**

通过校验类 CommonValidator 和配置类 FrameworkConfig，对请求进行校验的拦截器类。

**A．**校验 header 参数，校验 header 的 x-correlation-id 等是否为空，校验 header 是否包括参数 headerName 等。

B. 校验分页请求是否传递分页参数 limit、offset。通过注解类@DisablePagingHeaderValidation 控制。

C. 拦截请求，把请求报文和请求时间放入静态缓存队列 requestList，可以通过接口 MetricController 查询缓存队列的数据。拦截响应，把请求对应的响应打印日志。拦截响应通过继承 HttpServletResponseWrapper 实现对 response 的多次读取，避免关闭流。

Token 校验（OAuth2.0，未实现）。

**2） 切面拦截器 CommonValidaorAspect**

配合标注在类和方法上的自定义注解@ValidateHeader 和@ ValidatePagingHeader，对有@RequestMapping 注解的方法进行校验，通过校验类 **CommonValidator 校验由**@ValidateHeader 的 value 的值指定的 header 是否为空。

**3） 对响应的异常的统一处理 CustomRESTExceptionController**

通过@ControllerAdvice 实现。

**4） 实体类 domain**

定义了所有的 Exception 类和 Error 类。

定义了分页实体类。

定义了健康检查类。

**5）工具类（utility）**

AsyncTask，通过实现 callable 接口的异步任务抽象类

BeanFactoryProvider

DateUtil，格式化日期类

ELKLogger，基于 slf4j 封装

ExceptionUtil，

GlobalPoolExecutor，全局的线程池

PooledExecutor，线程池

HideInternalsParameterPlugin，

HostInfo，获取本地服务器的地址和名称，InetAddress

StrictSimpleDateFormat，

TracerUtil，配合 sleuth 使用。

CommonValidatorAspect，自定义切面，通过注解+切面，实现校验。

CommonValiInterceptor，是对每一个请求的拦截处理，CommonValidatorAspect 是对加了注解的请求的处理。

**实体类（domain）**

**控制层（Controller）**

CustomRESTExceptionController，对所有的 Controller 做自定义的全局的异常处理。

**配置类（Config）**

**1.5、ms-lib-core**

**微服务实例实际集成的框架核心。**

**1） 对响应的异常的统一处理类 BaseControllerAdvice。**

**2） 过滤器 BaseValidationFilter**

**校验请求防注入攻击；请求头参数非空校验；**

**3） 切面拦截器 LoggingAspectConfiguration**

拦截自定义注解@TrackRequestResponse 或@RequestMapping，日志打印拦截点信息、请求参数、返回信息（马赛克处理）、耗时、异常信息。问题：返回对象未转为 json。

**2、Spring-httpclient**

依赖 spring-boot-core，封装了 Rest 和 Soap 调用的原生插件。RAO 项目的服务大多没有集成。

1） RestHttpClientConfig，封装 RestTemplate，OKHttpClient。

注册 RestTemplateInterceptor 拦截器类，和 RestTemplate 绑定。

注册 OKHttpClientInterceptor 拦截器类，和 OKHttpClient 绑定。

拦截器类用于打印请求头、方法和 URI。

2）SoapHttpClientConfig，封装了 WebServiceTemplate。

从配置文件获取 com.ocbc.ms.httpclient.soap.ssl.enable:false，配置是否 SSL 连接。

问题：不够灵活。初始化配置后不能修改。

问题：两种调用方式（Rest 和 Soap）属于不同的业务场景，不应该封装在一起。

这两个拦截器的作用是什么？还未完成，目前只做日志。

**3、Spring-boot-restclient-example（pom 工程）**

Example.restclient-web（实际的 httpclient 的 rest 调用的例子）

依赖 spring-boot-core 和 spring-httpclient

没有实现调用。

Example.restclient-entity（剥离出来的请求、响应、分页的实体）

Example.restclient-mock（剥离出来的模拟的 service 层）

**4、spring-boot-kafka**

基于 kafka 封装。

ConsumerConfiguration

ProducerConfiguration

5、 spring-boot-kafka-example（pom 工程）

Example.kafka-web

依赖 spring-boot-core 和 spring-boot-kafka

Example.kafka- entity

Example.kafka- mock

## 宇信科技 **EMP** 分布式平台产品技术架构

MyCat：数据库中间件，实现分库分表，主从分离。

FastDFS：轻量级的分布式文件系统

服务注册中心 Eureka server 实现多服务实例（异地）部署，通过相互注册实现多个实例之间的数据同步和一致性

Webflux：非阻塞异步模式

问题：公司前段没有形成/积累自己的开发框架。导致重复开发。

**前端开发框架：**

**基础库：**适配库，第三方插件库；

**公共组件：**包含服务访问、路由管理、安全控制、会话/权限管理、字典管理、数据挡板、数据存储、事件管理、日志组件、校验工具、前端埋点等公共组件，提供框架公共 API 接口；前端基础/业务组件：**基础组件**，由最基本的界面元素组成，如按钮、表格、树结构、布局、日期时间选择器及常用表单控件等。**业务组件**，根据实际业务需求和用户体验需求，由基本要素和界面元素构成，实现特定需求、特定场景的用户组件，如导航菜单、页签、表单、穿梭框、弹窗、信息框、步骤条、进度条、自定义选择器等。

**业务模板：**包含查询类、表单类、自定义类模板、可集成于前端开发工具，负责代码的自动生成。

连接池：HikariCP，对标 Druid

持久层：Mybatis

分布式事务管理：Spring 框架 Data Source Transaction Manager 事务管理器

可视化设计工具

## 华侨永亨的 IT 的问题

微服务框架不完整，比如没有提供消息组件，缓存等；

框架没有不断迭代；

缺少使用手册及更新；

缺少服务监控；

缺少顶层设计；

服务划分缺少整体设计；

还在使用项目思维在做微服务；

缺少中台思维；

缺少产品思维；

在 CMMS 方面做的成熟，可能适用于银行？

需求缺少设计工具和平台。

因为服务划分的不合理，BFF->Journey->Common，Journey 层太重

服务划分缺少统一的，顶层的架构设计，不合理

耦合系统很多且不稳定，没有应对措施

前端没有埋点

建制不全

服务没有分层

缺少服务治理

没有有效的设计、组织和管理

沙子 vs 积木

服务 vs 包袱

## Ms-pingan 的加密

对请求体做 Hash 加密生成固定长度的密文 A1。加密算法 SHA-256.

对请求方法类型（POST）+请求地址+请求头（Content-Type，X-Appid，X-Deviceid，X-Timestamp）+请求体密文 A1+appkey 通过加密算法生成**签名**，把签名放入请求头（X-Authorization）。

**生成签名的加密算法**：拼接数据；对拼接数据做 SHA-256 加密得到密文；通过密钥 appkey 对密文做 SHA-256 加密得到签名。

平安受到请求后，获取请求头的 appID，查询得到对应的 appKey，再对请求头和请求体明文，使用系统的加密算法得到签名，**通过比对签名校验数据的一致性**。

**结论：平安的这种方案解决了一半的问题。**即对平安的请求接口的数据的一致性校验的问题。但是平安返回的校验结果的数据一致性问题没有解决。**攻击者可以拦截并篡改返回的校验结果。**

## Ms-ibss 的加密

激活页面：

1、前端调用后端 ms-ibss，调用 ibss/init，请求头 RqUuid，ClientId，ClientCountry，TraxDateTime，请求体 RAOCaseId，UniqueId，IPAddress，DeviceId。如果返回成功，ms-ibss 用 uniqueId 生成 JWT，返回给 rao，返回给前端。

**前端调用 bff**，前端传递 IPAddress，DeviceId。

**Bff 调用 rao**，rao 从请求头获取 applicationId，即 RAOCaseId。

**Rao 调用 ms-ibss**，rao 通过工具类生成随机数 uniqueID。

**Ms-ibss 生成 JWT：**从本地密钥文件 rao-keystore-4096.jks 获取 keystore，通过别名（mscp-dev/prod.ocbcgroup.ocbc.com）从 keystore 获取公钥，通过别名和密码获取私钥。再通过 UniqueID 和 expireDate 生成 JWT。

ms-ibss，**把 uniqueId 和 expireDate 放入 payload**，通过私钥加密得到签名部分，调用 hk ibss 传递参数（raoCaseId, uniqueId, ipAddress, deviceId）,返回 JWT 给前端。

ibss 不需要用私钥进行加密得到 JWT；ibss 用公钥对 JWT 进行解码得到 uniqueId，跟请求参数进行比对校验一致性；然后把参数存储到数据库。

2、 前端调用 IBSS/isEligible 校验 JWT

前端传递（deviceId，JWT），IBSS 校验 JWT，包括签名有效性，在有效期内，数据一致性。成功则返回 E2EE.Public Key Module（公钥），Public Key Exponent， 3DES Key Length。前端存储上述参数到 encryted store。

3、 用户输入用户名和密码

4、 前端调用 IBSS/create

前端获取存储的 key，调用 E2EE.Js 对密码做对称加密；前端调用 IBSS/create，传递 JWT 和账户名和（加过密的）密码。IBSS 校验 JWT；IBSS 注册并激活账户名和密码。

注：IBSS 负责 JWT 的校验；JWT 同 RAOCaseId，UniqueId，IPAddress，DeviceId 绑定，意味着注册过的 deviceId 是否可以再用，取决于 IBSS 绑定的强弱；

调用 isEligible 时，JWT 的作用是**设备校验**的数字签名；

调用 create 时，账户成功激活后把账户名密码同 JWT 绑定；估计账户激活，用户登录及以后的操作，都会把 JWT 作为身份校验的数字签名。

**设备校验：IBSS 通过 JWT 获取关联对应的 UniqueId 和 deviceID，同前端传递的 DeviceID 进行比对。**

**问题：ibss 作为了鉴权中心，每次请求都要通过 ibss 校验 JWT。失去了 JWT 的意义。**

前端调用 hk ibss 时传递参数（deviceId，JWT），ibss 对 JWT 的签名部分通过公钥解密得到明文，获得 uniqueId，查询得到关联的 deviceId，比对请求的参数（deviceId）进行校验。

## Maven

### Parent 和 dependency 的区别

比如项目 A 在其 pom 中引用了依赖包 X，并定义了类 Y。

项目 B 用 parent 方式引用了项目 A，则项目 B 中可以直接使用项目 A 中引入的依赖 X，不可以使用类 Y。
项目 C 用 dependency 方式引用了项目 A，则项目 C 中可以直接使用项目 A 中引入的依赖 X，也可以使用类 Y。

### DependencyManagement 的作用

DependencyManagement **只是声明**依赖，不实现引入。
比如在父项目的 pom 的 DependencyManagement 声明了依赖 X，在子项目中不会自动引入，需要在子项目的 pom 中声明依赖 X。

如果在同一个 pom 中，DependencyManagement 和 dependency 都声明了依赖 X，则以子项目的 dependency 优先。
如果在不同的 pom 中，比如存在父子关系的 pom 中，两者都声明了依赖 X，如果子项目的 pom 的依赖 X 的声明没有指定 version 和 scope，则自动继承父项目的 version 和 scope。如果子项目的 pom 的依赖 X 的声明已经指定 version 和 scope，则以子项目的为准。

通常应用于多个子项目的工程，在父项目的 DependencyManagement 指定所有子项目的公共依赖包的版本。这样，所有子项目依赖的公共依赖包都保持一致，统一了模块之间依赖的版本。如果某一子项目需要不同版本的公共依赖包，可以在自己的 pom 中自定义版本。

### Parent 的作用

Parent 用于声明项目之间的父子继承关系，用于管理多个项目之间公共的依赖项：**多个项目有共同的依赖，而且依赖项的版本发生变更时，需要所有项目同步更新。**

**使用方式一、**定义一个 parent 项目 A，在 A 的 pom 中通过<dependency>声明依赖项 P。
在子项目 B 和 C 的 pom 中声明 parent 是 A，即可实现 B 和 C 对 P 的依赖，以及版本的统一管理。

**使用方式二、**定义一个 parent 项目 A，在 A 的 pom 中通过<dependencyManagement>声明依赖项 P。

在子项目 B 和 C 的 pom 中通过<parent>声明 parent 是 A，并通过<dependency>声明依赖项 P（无需声明依赖的版本），即可实现 B 和 C 对 P 的依赖，以及版本的统一管理。

同时，在子项目 D 的 pom 中通过<parent>声明 parent 是 A，但是不通过<dependency>声明依赖项 P，即表示项目 D 并不依赖 P。

## 微服务的划分

每个产品（每家银行）一个分支，分支和主干之间没有版本管理的关系，分支之间完全独立。

每个分支一个团队，在迭代中的需求和问题，选择性的通过主干的迭代升级实现支撑。

分支是基于主干的一个基线版本作为开发起点，**不会跟进**主干的基线版本迭代。分支可能会跟随主干的一些小版本的迭代。或者说，分支面临一些新需求，需要主干支撑，主干通过迭代到新的版本实现，然后分支跟随到该新版本。

这种方案的好处是分支选择的主干基线版本稳定，分支的基础就是稳定的。相反，如果分支跟随主干基线版本升级，则主干的每次升级，都需要每个分支做全面的回归测试。

这种方案的一个问题是，如果分支面临一个新需求，比如增加分布式缓存，如果分支选择的老的主干基线**版本不支持**，又无法选择新的主干基线版本，则分支可能无法实现该需求。分支选择的主干基线版本越老，面临的该类问题可能性越大。

第二个问题是**维护成本**的增加。随着分支线越多，面临的该类问题越多。这样在分支线和主干线上的维护成本会越来越大。

第三个问题是，主干的版本演进要做到向下兼容。

当分支面临新的需求，一种方案是在分支的版本上进行**重构**；另一种方案是基于最新的主干的基线版本进行**重做**。

SS 层，微服务框架的组件层。包括 redis，kafka，mybatis 等等。
SSP 层，各产品线的业务层，包括了业务层的各个微服务。
**产品线**比如银企通，T 信等等。
银企通又可以再细分到各个银行，每个银行都是一个**分支产品线**。
在该层所有的微服务中，比如 **SSP-Bis-Bill**，是各产品线**通用的微服务**。有的是各业务线通用的微服务，有的是各产品线通用的微服务。

**SSP 层的划分**

把微服务按产品线划分，比如业务服务（和业务有关，组合封装其他服务），
把某个产品线通用的微服务划分出来，比如微服务（和业务有关，产品线内可以重用）
把各产品线通用的微服务划分出来，比如实体服务（和业务实体有关，在不同产品线内可重用）
还有和各业务流程无关，封装底层技术中心功能，可以重用，比如公共服务（通知、日志、安全等）

**划分的好处是什么？**

微服务的划分属于服务规划范畴。对微服务进行了梳理和识别，通过横向和纵向的划分，从产品线的维度（纵向），业务相关性和重用性的维度（横向）。
再通过生产环境的数据，比如调用和日志监控数据，获得微服务及微服务之间在各个维度方向上的数据，并生成全景的看板视图。
通过对生产运营数据的分析，或者对于一个新需求的分析，可以分析问题应该对应到哪个产品线（纵向），同时应该属于哪个层次（横向），是应该升级原有的微服务，还是新增微服务，还是需要把原有的微服务拆分。

## 什么是微服务

微服务架构的演进像是一个公司的发展过程，最开始一个五脏俱全的小公司（单体应用），发展成大公司（功能复杂臃肿），大公司拆分出多个子公司，每个子公司有自己独立的业务、员工，独立核算，互不影响，互有联系，互相合作。

当一个子公司（微服务）面对的市场、需求急速扩大时，可以投入资源对其扩容。当一个子公司需要改变时，不会影响到其他子公司。

## 应对高并发的方法

缓存，限流，降级，消息队列，扩容，大数据量（清理和分箱），负载均衡。

### 缓存

包括分布式缓存和本地缓存，比如 Guava，Redis，Hazelcast，减少数据库访问的瓶颈。
通过消息队列，实现服务之间的解耦。把同步操作改变为异步操作。

### 限流

主要应对高并发访问下对单个微服务的冲击。（Guava，Redis，RateLimiter，网关）。限流应该结合服务扩容使用。

限流的目的是为了保护系统不被大量请求冲垮，通过限制请求的速度来保护系统。在电商的秒杀活动中，限流是必不可少的一个环节。

限流的方式，可以在 Nginx 的层面限流，也可以在应用中限流，比如在 API 网关中。

限流可以分为单点限流和集群限流，区分在于使用本地缓存还是分布式缓存。

还可以根据实施的环节和粒度，区分为网关限流，服务限流，实例限流，接口限流。

**常见的限流算法**有：令牌桶、漏桶。计数器也可以进行限流实现。

**一、令牌桶**

令牌桶存放固定容量上限的令牌。

一个定时服务按照固定速率往桶里面添加令牌，直到上限。该速率控制了并发量和流量。

当一个请求到达时，从桶里面取走一个令牌。如果获取不到则等待或者放弃。

**二、漏桶**

请求可以是任意的速率到达（超出漏桶容量的直接抛弃），但是只能以固定的速率被消费。

应对突发的并发请求的场景不够好，因为出口的流速是匀速的。

**限流和 txw-typhon 的区别：**

限流的思路是，一个米缸，按照一定的速率往米缸里面投米，比如每秒一粒米，并为米缸设置最大上限。当有老鼠（请求）过来时，先看米缸里面有没有米，如果有，一个老鼠拿走一粒米，如果没有，则老鼠等待。如果超过一定时间（超时时间），则等待的老鼠饿死（返回超时）。可以看出来，**老鼠等待多长时间被饿死主要由调用方设置的超时时间决定。**

Typhon 不同，typhon 按照一定的速率计数，比如每秒 10 粒米。当有老鼠过来时，在 1 秒钟之内，前面 10 只老鼠拿走米，第 11 只老鼠开始，**typhon 会主动毙掉（返回请求失败）**，直到该时间周期结束。

txw-typhon 类似于接口限流，txw-typhon 支持单点和集群。比如 txw-typhon 类似于令牌桶算法，有一个固定的速率。

https://www.jianshu.com/p/5d4fe4b2a726

**服务降级**

在高并发下暂停非主要业务，释放资源，维持主要业务。（Docker+k8s，网关）。

服务降级的方式有多种，最好的方式通过 docker 实现。当需要对某个服务进行降级，直接将这个服务所有的容器停掉，需要恢复的时候重启容器。

也可以在 API 网关层处理。当需要对某个服务进行降级，前端过来的对这个服务的请求网关直接拒绝掉，不再往内部转发。比如通过 zuul+云配置实现动态服务降级。

从请求参数中获取 serviceId。查询配置，该 serviceId 是否在需要降级的服务列表。如果是，直接返回该服务已被降级。

**灰度发布**

灰度发布（金丝雀发布），是指在服务的两个新旧版本之间，实现平滑过渡的一种发布方式。在其上，可以实现 A/B testing，即让一部分用户继续使用产品特性 A，一部分用户开始使用新的产品特性 B，测试用户对产品特性 B 的反馈。如果符合预期，则扩大用户范围，直到把所有用户迁移到 B。如果有问题，则及时调整修复问题，限制其影响度。

灰度发布的原理是对请求进行分流（用户分流或 IP 分流），让指定的用户（或请求来源 IP）访问指定的新版本的服务（实例），而其他用户还是访问当前版本的服务。

灰度发布在系统需要发布新功能的版本时用到。当前所有服务实例的版本为版本 V1。

1、首先将机器 A 的服务开始发布新版本 V2，发布过程前，需要先隔离所有的请求。

2、发布结束后，满足条件的请求分流到该机器 A。

对请求的分流可以在 API 网关中统一处理。

1、 将要做灰度发布的服务设置成灰度发布状态，从正常服务列表中移除。

2、 获取当前请求的用户 ID，如果是灰度发布用户，从可用服务中获取对应的灰度发布的用户。

**扩容和缩容，（Docker+k8s）**

**负载均衡**

通过分流应对高并发下的访问和调用。可以通过中间层（F5，Nginx，Openresty，NodeJS），也可以做在调用端（Ribbon，服务注册和发现）。

**分流**

比如对入口进行分流，比如移动端（细分 Android 端和 IOS 端），网页端，API 调用端等。因为不同端的技术栈不同。也方便统计不同端的流量。

还有对接口的分流，把高并发下重载的接口分离出来，作为单独的服务，单独实现扩容。

## 数据存储

大数据量的存储，主要是清理和分箱。

应对大数据量，包括数据访问，方案一是清理，二是分类。

清理就是定期对数据进行清理和归档。

分类一是根据操作做读写分离。二是做水平划分（分片）。三是做垂直划分，比如微服务的库拆分，还有根据不同的场景选择合理的存储方式。

搜索服务使用 ES，日志服务使用 MongoDb，业务数据使用 Mysql，缓存数据使用 Redis 和 Guava。

## 微服务框架中数据库的选择

**一是按照存储的数据对象划分。**

业务数据存放到关系数据库，比如 mysql，postsql。资金允许可以用 oracle。存储引擎用 InnoDB，提供具有提交、回滚和崩溃恢复能力的事务安全存储引擎。

缓存数据用 redis。

大数据量的基础数据用 MongoDB，比如文章评论。注意 MongoDB 中一个文档最大 16M。日志数据用 ES。全文搜索的业务需求用 ES，比如文章（按照标题、发布用户、发布时间、文章内容等）搜索。

**二是按照需求来划分。**

如果需要做全文搜索，用 ES。

如果是范式搜索和增删改查，用关系数据库。

如果是文章评论，而且一个文档小于 16M，用 MongoDB。

去中心化的数据管理，每一个微服务都有自己的数据库。优点在于不同的服务可以选择适合自身业务的数据，比如订单服务用 mysql，评论服务可以用 MongoDB，商品搜索服务可以用 ES。缺点是事务的问题。目前的解决方案是最终一致性。

## Paas 云架构

从 4 类主要使用者的角度，来分析私有 Paas 云的架构

**开发者：Paas 云为其提供开发平台的能力。**

平台封装了基于 DEVOPS 的代码托管、镜像仓库、持续集成、自动化部署，自动化测试，上线的能力。可以分为开发环境、测试环境和生产环境。

开发者首先通过**开发者门户**注册，建立工程，提交代码到**代码库**；**代码库**通过 **webhook** 通知 **Jenkins** 拉取代码；Jenkins 自动构建，构建成功的镜像推送到**镜像库**；通知**容器管理者**从镜像库拉取镜像并部署到容器，通知**云路由（应用路由，负载均衡，会话控制，访问控制）**更新服务实例信息；平台对部署的应用做自**动化测试**；测试通过后，开发者在 Paas 平台申请需要的计算资源等，把应用上线；开发者通过 Paas 平台对应用进行监控。

**应用开发商：Paas 云为其提供了应用托管的能力。**

应用开发商通过**开放平台门户**注册，提交应用到**应用仓库**；并部署到**应用中心**；该应用可以调用**平台提供的基础服务（比如数据存储，缓存，消息队列，文件服务等）**，也可以调用**服务提供商**提供的**开放平台的服务**。

**管理者：Paas 云为其提供了平台管理的能力。**

平台封装了各个层面的管理和监控能力，包括全局管理（应用、服务、节点、路由等的管理，自动动态伸缩），节点管理（心跳通信、容器管理、资源采集、快照），Iaas 管理（接入管理、资源管理）。

**用户：**Paas 云为其提供了消费服务的能力。

平台封装了 Saas 层。

用户通过各种**终端（PC，手机，平板，数字电视，自助终端等）**，改过**云路由**，使用平台提供的 Saas 层的服务。包括平台的 Saas 层的服务，和应用开发商提供的应用。用户可以通过统一的**云助理**，收藏和下载应用。

## 服务注册

服务注册（服务治理）是微服务架构中必不可少的一部分，比如 Eureka，Consul，Etcd，Zookeeper，Dubbo。

Eureka 是基于 AP 原则构建的，而 ZooKeeper 是基于 CP 原则构建的。

Zk有一个leader，而且当leader无法使用的时候通过paxos（ZAB）算法选举出一个新leader。

这个 leader 的任务是保证写数据的时候只向这个 leader 写入，然后 leader 会同步信息到其他节点。通过这个操作可以保证数据的一致性。

## 异步执行@Async

比如在 service 类的一个方法上增加该注解，然后在 controller 类中调用该方法。注意不可以在本类调用，否则不起作用。

# 分布式缓存

## 缓存穿透

请求的 key 大比例不是缓存的 key，导致缓存失效，数据库压力上升。比如非法 key 的攻击。

方案一：当查询数据库为空记录时，缓存请求的 key，下次请求时直接拒绝。问题，无法解决每次不同的 key。

方案二：对 key 做校验，比如 key 的生成有一定的规则，通过该规则校验请求的 key 是否合法。可以解决部分的问题，适用的场景比较少。

方案三：对缓存 key 校验，把所有可能缓存的数据 Hash 到一个足够大的 BitSet 中，在缓存之前先从**布隆过滤器**中判断这个 key 是否存在，然后再操作。有一定的误识别率，删除困难。

方案四：把所有的 key 和 value 缓存。

**缓存率**：缓存数据占数据库的比例。可以动态调整缓存率。当请求并发上升时，主动调整缓存率。

**穿透率**：监控缓存穿透的比例。

**命中率**：监控缓存命中的比例。

## 缓存雪崩

在某一时刻，大量缓存同时失效，导致所有请求都去查询数据库，数据库压力太大。

对于 AP 要求高的系统，不可允许缓存雪崩的发生。

监控缓存率，突然下降，导致雪崩。

方案

一：缓存高可用，比如缓存集群。

二：设置缓存失效时间，不同的数据设置不同的有效期，统一规划有效期，让失效时间均匀分布。

三：对于热门数据的读取，设置定时更新数据的方式刷新缓存，避免其自动失效。

四：服务限流和接口限流：限制请求的并发量在可承受的范围内。

五：从数据库获取缓存需要的数据时加锁控制，本地锁或者分布式锁。通过加锁，避免大量请求同时访问数据库。

## 本地缓存

### Guava

优点：速度快，不受网络影响；线程安全；操作简单，功能丰富。

缺点：不能持久化；受内存限制；应用重启数据会丢失；分布式部署时无法保持数据一致性。

## 网关和 BFF

https://www.cnblogs.com/dadadechengzi/p/9373069.html

https://blog.csdn.net/wo18237095579/article/details/83543592

比较完整的现代微服务架构，从外到内依次分为：**前端用户体验层->网关层->BFF 层->微服务层**。整个架构层次清晰，职责分明，是一种灵活的能够支持业务不断创新的演化式架构。

1、BFF 按团队或业务线进行解耦拆分，拆分成若干个 BFF 微服务，每个业务线可以并行开发和交付各自负责的 BFF 微服务。

**聚合**：某一个功能需要同时调用几个后端 API 进行组合，比如首页需要显示分类和产品细节，就要同时调用分类 API 和产品 API，不能一次调用完成。

**裁剪**：后端服务返回的 **Payload** 一般比较通用，App 需要根据设备类型进行裁剪，比如手机屏幕小，需要多裁掉一些不必要的内容，Pad 的屏幕比较大，可以少裁掉一些内容。

**适配**：一种常见的适配场景是格式转换，比如有些后台服务比较老，只支持老的 SOAP/XML 格式，不支持新的 JSON 格式，则无线 App 需要适配处理不同数据格式。

**注：在前后端之间通过引入 BFF 层，实现前后端解耦。BFF 实现聚合、裁剪和适配，可以让前端减少工作量。对前端而言，BFF 屏蔽了后端的接口。**

2、**网关**在无线设备和 BFF 之间又引入了一层，让两边可以独立变化，特别是当后台 BFF 在升级或迁移时，可以做到用户端应用不受影响。网关隐藏内部架构。管理大量的 API 接口，对接客户，适配协议，进行安全认证，转发路由，限流，监控日志，防止爬虫，进行灰度发布，压力测试/金丝雀测试，服务迁移等。

网关一般由独立的框架团队负责运维，**专注跨横切面(Cross-Cutting Concerns)的功能**，包括:

**路由**，将来自无线设备的请求路由到后端的某个微服务 BFF 集群。实现容错和回退。实现智能路由和负载均衡。

**认证**，对涉及敏感数据的 API 访问进行集中认证鉴权。身份认证和权限认证。阻止非法请求。

**流量监控**，对 API 调用进行性能监控。实现服务降级。

**限流熔断**，当出现流量洪峰，或者后端 BFF/微服务出现延迟或故障，网关能够主动进行限流熔断，保护后端服务，并保持前端用户体验可以接受。

**安全防爬**，收集访问日志，通过后台分析出恶意行为，并阻断恶意请求。

**注：网关更注重跨横切面逻辑，比如安全认证，日志监控，限流熔断等。引入网关层，实现前端和 BFF 层的解耦。BFF 层可以按照业务需要，按照业务线或产品线进行拆分。**

按照业务需要，网关层可以进一步拆分，比如按照接入类型，拆分为支持第三方开放 API 的网关，面向 H5 应用的网关，面向移动端的网关。

**前置过滤器**：**在 http 请求到达实际服务之前对 http 请求进行拦截和修改。**

身份验证和授权；请求头校验（比如统一的格式校验，必须包含 x-corelation-id 等，和业务无关）；

**路由过滤器**：**可以更改服务所指向的目的地。**

确定是否需要进行某些级别的动态路由，比如在同一个服务的不同版本之间进行路由（灰度发布，A/B 测试）；

**后置过滤器**：**可以检查和更改来自被调用服务的响应；**

记录响应；处理（响应中的）错误；审核敏感信息；

**华侨永亨的 BFF，**

一是**拦截请求**，实现 sessionID 的管理（生成，校验，applicationID 的映射）；二是实现到 ms-rao 的 API 的**路由转发**。

Ms-Rao 实现了业务组装，相当于业务和 API 的**聚合**。

Domain 层实现了对老系统，特别是 soap 接口的**适配**。

**边缘网关**

一般而言，网关一般是工作在请求和微服务之间。实际上，微服务可能需要调用第三方服务。可以在微服务和第三方服务之间设置网关，这些网关工作在微服务的边缘，称之为边缘网关。网关 B 的作用可以是统计对第三方服务的调用，服务降级熔断。

20200927

**BFF 常用场景：**

聚合：合并多次调用

裁剪：对 payload 的内容进行裁剪

适配：适配不同的前端，对数据格式进行调整。

**BFF 的实现方式：**利用工厂模式+spring 的自动注入

https://blog.csdn.net/coqcnbkggnscf062/article/details/104061750

1、创建 AbstractService 类或接口，抽象方法 method();

2、具体工厂类 ServiceA 继承或者实现 AbstractService

3、在配置类 ServiceHolder 中定义

@Autowired

Private Map<String, AbstractService> serviceMap; // 默认的 key 是具体工厂类的 bean name

调用类：

serviceHolder.getServiceMap("serviceA").method();

4、可以利用配置文件，实现更灵活的配置

Otp:

aliasMap:

bvweb: serviceA

bvmobile: serviceB

1）实现配置类 AudAlias

```
@Component
@EnableConfigurationProperties
@ConfigurationProperties("Otp")
Public class AudAlias {
    Private Map<String, String> aliasMap;
}
```

2）修改 serviceHolder 类的 getServiceMap 方法，实现

Aud -> service name -> service class

3）调用者:

serviceHolder.getServiceMap(aud).method(); // 变量 aud 的取值为 bvweb 或 bvmobile

## 负载均衡

常见的负载均衡实现有两种方式。一种是**独立的进程单元**，通过负载均衡策略，将请求转发到不同的执行单元上，例如 Nginx，F5，Bluecoat，Ingress Router 等。

另一种是将负载均衡逻辑以代码的方式**封装到服务消费者**的客户端上，消费者维护一份服务提供者注册的信息列表。通过列表，和负载均衡策略将请求分摊到多个服务提供者实例。例如 Ribbon。

## 熔断 Hystrix

如果存在大量的已有系统，新系统需要调用已有系统。某些已有系统会出现网络延迟甚至中断，或者需要定期重启。这时可以在新系统中使用**服务降级/熔断**，实现**快速失败** Fast Fail。当已有系统恢复工作，新系统再重新恢复正常调用。

采用服务熔断，可以实现快速失败，避免连续失败重试。

服务熔断并没有提高并发处理的能力，而是通过快速失败，避免故障范围的扩大和传递。

另一种方式是改成异步调用，包括异步重试。

**是否使用服务熔断?**

首先是被调用的服务 B 可能出现故障。其次是业务上允许熔断处理。进而考虑在服务 A 上使用服务熔断。

另一方面，从微服务的角度，任何一个微服务都有可能出现故障。从这个角度，服务熔断又是普遍有用的。或者说，需要普遍考虑的，服务 A 调用任何一个服务，如果对方出现故障，应该如何处理。

如果考虑使用熔断，需要考虑熔断策略和恢复策略。

如果已有系统的并发处理能力有限，可以在新系统中使用**服务限流**。当出现高并发请求时，对调用的线程数进行限制，比如**线程隔离**（该调用使用单独的线程池），或者**信号量隔离**（该调用共享 web 容器线程池，但是有最大限制）等，避免对已有系统的冲击。

如果对于已有系统只是读取信息，而且相同的请求参数，会得到相同的返回。则可以在调用方使用**请求缓存**；如果在一定时间段后返回可能更新，则可以设置请求缓存的过期时间。适用于可能出现大量并发的相同的请求参数的场景。

**服务熔断**：服务 A 调用 B，因为 B 发生故障，服务 A 暂停调用 B，直至 B 恢复。

**服务降级**：服务 A 的整体负荷过高，选择暂停非核心业务/接口，直至负荷降低。

**信号量隔离**

```
@HystrixCommand(
        commandKey = "createOrder",
        commandProperties = {
                @HystrixProperty(name = "execution.isolation.strategy", value = "SEMAPHORE"),
                @HystrixProperty(name = "execution.isolation.semaphore.maxConcurrentRequests", value = "6")
        },
        fallbackMethod = "createOrderFallbackMethod4semaphore"
)
```

**线程池隔离**

```
@HystrixCommand(
        commandKey = "createOrder",
        commandProperties = {
                @HystrixProperty(name = "execution.isolation.strategy", value = "THREAD")
        },
        threadPoolKey = "createOrderThreadPool",
        threadPoolProperties = {
                @HystrixProperty(name = "coreSize", value = "3"),
                @HystrixProperty(name = "maxQueueSize", value = "5"),
                @HystrixProperty(name = "queueSizeRejectionThreshold", value = "7")
        },
        fallbackMethod = "createOrderFallbackMethod4Thread"
)
```

**超时熔断**

```
@HystrixCommand(
        commandKey = "createOrder",
        commandProperties = {
                @HystrixProperty(name = "execution.timeout.enabled", value = "true"),
                @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "3000"),
        },
        fallbackMethod = "createOrderFallbackMethod4Timeout"
)
```

## Spring Security

**能否解决中间人攻击，修改响应数据？不可以。**

在安全方面，主要解决两个问题，**一是认证，你是谁；二是授权，你拥有什么权限**。Spring Security 即解决这两个问题。采用注解的方式解决这两个问题。

Spring Security 采用安全层的概念，可以工作在 controller，service，dao 层，提供不同细粒度的权限控制，可以细到每一个 API 接口，每一个业务的方法，或者每一个操作数据库的 DAO 层的方法。Spring Security 是应用层的安全解决方案，HTTPS 和防火墙是网络传输层的解决方案。

Spring Security 的优点：环境无依赖性，代码低耦合。提供了几十个**安全模块**。

@EnableWebSecurity 注解开启 web 保护功能

@EnableGlobalMethodSecurity 注解开启方法上的保护功能

## JWT

JWT 是各方之间安全传输信息的一种方式，JWT 使用签名加密，安全性很高。另外，当 Header 和 Payload 计算签名时，**还可以验证内容是否被篡改**。

https://www.cnblogs.com/shihaiming/p/9565835.html

**JWT 格式：header.payload.signature**

**1、header** 是头部信息，对明文的 base64 编码，不是加密，是可逆的。

{ 'typ': 'JWT', 'alg': 'HS256' }

**2、Payload 是数据包，同样是 base64 编码，可逆。**

**1）标准中注册的声明**

iss: jwt 签发者

sub: jwt 所面向的用户

aud: 接收 jwt 的一方

exp: jwt 的过期时间，这个过期时间必须要大于签发时间

nbf: 定义在什么时间之前，该 jwt 都是不可用的.

iat: jwt 的签发时间

jti: jwt 的唯一身份标识，主要用来作为一次性 token,从而回避重放攻击。

**2）公共的声明**

**3）私有的声明**

**3、Signature 是签名，加密信息，用于验证。**

可以用不可逆加密，比如 SHA256，MD5，也可以使用非对称加密，比如 RSA256.

使用非对称加密，用私钥加密得到签名部分，用公钥解密得到明文。

相当于 JWT =

Base64UrlEncode(header) + “.”

+ Base64UrlEncode(payload) + “.”

+ SHA256RSA.sign(base64header, base64payload, privateKey)

签名的部分是对 base64header + “.” + base64payload，通过私钥做 RSA256 加密得到。

请求端获得 JWT 后，可以对 base64header 和 base64payload 的部分做 base64 解码得到明文。可以对签名的部分通过公钥 publicKey 解密得到明文，通过后者得到的明文更安全，还可以对两者的明文做比对。

比如 ms-ibss，把 uniqueId 放入 payload，通过私钥加密得到签名部分，调用 hk ibss 传递参数（raoCaseId, uniqueId, ipAddress, deviceId）, ibss 不需要用私钥进行加密得到 JWT；返回 JWT 给前端。

前端调用 hk ibss 时传递参数（deviceId，JWT），ibss 对 JWT 的签名部分通过公钥解密得到明文，获得 uniqueId，查询得到关联的 deviceId，比对请求的参数（deviceId）进行校验。

SHA256 是散列加密。

如果修改 JWT 中的 payload 部分，解析 jwt 时是否会报错？会，不能通过签名校验。

理论上，攻击者可以先通过一个正确的请求获得正确的 JWT，然后使用该 JWT 直到过期。

**解决中间人攻击的方案**

1、在接口 A 上面增加注解@Pitcher 投手，在接口返回响应时，对返回 json 体生成 JWT，并放入响应 header。

HandlerInterceptor 无法使用 postHandler 修改 response

https://www.cnblogs.com/haitao-fan/p/10308772.html

使用 ResponseBodyAdvice 修改 response 的 header，需要有@ResponseBody

https://www.cnblogs.com/nuccch/p/7891634.html

2、在接口 B 上增加注解@Catcher 捕手。

在接口收到请求时，对 JWT 解码获得 json 体，进行校验。

要拦截请求，获取到请求头；要判断是否有注解@Catcher；

判断请求头是否有 token；从 token 解码出 json 体；

从 json 体判断是否。

https://www.jianshu.com/p/657fa7118e84?utm_source=oschina-app

不能使用 RequestBodyAdvice，因为需要有@RequestBody

https://www.hellojava.com/a/45272.html

https://www.cnblogs.com/Wicher-lsl/p/11436224.html

## 事件驱动架构

**场景：**服务 A 完成了一次交易。服务 B 需要知道交易的状态和详情。

**方案一：请求响应架构**，服务 B 调用服务 A 的接口；

**方案二：**服务 A 在交易完成后调用服务 B，通知 B 交易信息。

方案一的问题是效率低，服务 B 不知道服务 A 交易是否已经完成，需要轮训 A。

方案二的问题是服务 B 可能有多个，比如日志，审计，通知，报表等，导致服务 A 需要调用多个服务。

**方案三：**这种场景通过事件驱动架构可以更好解决：服务 A 完成交易后，发送事件（生产），订阅了该事件的服务都可以消费该事件。生产和消费之间是解耦的，异步的。消费该事件的服务之间也是解耦的，异步的。

**不适用的场景**：

需要消费事件的服务之间需要的信息不同，权限不同。

操作的执行次序需要很严格，不允许异步。

请求需要及时，同步获得响应。得到响应之后才可以继续下一步。

可能会产生分布式事务，处理复杂。

比如用户填写手机号码后请求 OTP；服务 A 发送请求：**生成 OTP**；服务 B 生成 OTP，并返回给服务 A；服务 A 发送请求：**发送 OTP**；服务 C 发送 OTP，并返回结果给服务 A；

再比如用户下单；服务 A 把库存商品减一；服务 B 把已售商品加一；

**适用的场景：**

**在业务上服务 A 和 B 之间依赖低**，服务 A 生产数据，发布事件，订阅了该事件的服务 B 消费事件和数据。

比如用户注册，然后点击 submit；服务 A 处理完业务逻辑后，发布 submit 事件；订阅了该事件的服务 B1，开始生成 pdf；订阅了该事件的服务 B2，开始发送邮件通知用户，等等。

**服务 A 只需要发布事件，不需要知道有哪些服务订阅该事件，也不需要知道哪些服务的 具体业务处理是什么，是否处理成功。**

**优势：**

异步方式的消息传递比同步方式具有更强的容错性，能够保障在系统故障时消息正常可靠地传输。异步消息传递模式分为点对点模式和发布订阅模式。前者用于生产者和消费者之间点对点的通信。

**注意**：上述方案属于**发布订阅模型**。处理过程通过事件的发布订阅实现解耦，减少了延迟，提升了效率，该过程不需要用户等待。

依赖于消息中间件。如果业务模型要求可靠性很高，需要考虑消息中间件是否满足。

需要消费者自己控制消费能力，控制获取待消费消息的速率。

另：微服务之间 rest 调用属于**请求响应模型**。

**问题：可能削弱自动扩缩的能力。**因为被调用方从消息总线中获取消息，不会获取超过自身处理能力的消息数量。

服务之间更加松散，实现接口调用的解耦。

服务可以异步方式响应事件。

## Sping Cloud Stream

通过引入 Sping Cloud Stream 实现应用和消息组件的解耦。应用和 stream 打交道，消息组件可以是 kafka，也可以是 rabbitMQ。

## Sping-kafka

实现同步的请求响应模型。

# 分布式事务

事务：一组操作，保证要么全部成功，要么全部失败，只要其中有一个失败操作，改组操作全部回滚，以此保证数据的一致性。

分布式事务即为了解决分布式环境下数据的一致性的问题。

分布式环境，不同的服务，不同的数据库，包括分库分表。

解决方案有两阶段型，TCC 补偿型（Try-Confirm-Cancel）和最终一致性

**最终一致性**

基于消息队列实现可靠的消息服务，实现数据的最终一致性。可靠的消息服务解决了有可能出现的异常情况，提供补偿机制，最终达到数据的一致性。