个人简化笔记2

三、HSF

简介：

HSF 是一个 RPC 框架，远程调用对端的地址就是由 ConfigServer（简称CS,端口9600）来推送的，HSF 持久化的配置中心是Diamond，Pandora 是 HSF 生存的容器，对于 HSF2.X 来说，HSF 只是作为 hsf.jar.plugin 这个插件，存活在 pandora 的sar包中，划分小组，组别一致的服务才可以互相调用。序列化的类型,默认为 Hessian。HSF 路由规则支持接口路由，方法路由，参数路由，采用 Groovy 脚本作为路由规则设置内容，路由规则是通过 Diamond Server 进行推送的。归组规则，是用于对发布了同一HSF服务的所有机器进行统一归组的规则。同机房优先规则，优先选择与服务消费者同机房的服务提供者。权重规则，为不同的机器设置不同的权重，这样可以将访问集中到几台机器上面。HSF异步调用：通过Future调用，通过Callback调用。Hsf对线上机器进行动态分组，权重规则来引流压测，在大促中的开关降级。

原理：

Consumer向远程服务发送服务请求，远程服务处理完请求，返回给consumer结果。

- server启动时候向configserver注册
- client启动时候向configserver请求list
- client缓存list，发现不可用的server，从缓存中remove
- configserver通过心跳包维护可用server的list
- list有更新的时候，configserver通过带version的报文通知client更新

服务上线：

HSF的所有provierbean在初始化阶段都不注册到configserver，而是等spring容器把所有的bean初始化成功后，发出refreshevent事件后，注册到CS，与此同时基于pandora的漩口提供status命令，当所有服务注册后设置status为true。 PE需要配合，在启动app server(tomcat)后，启动web server前，需要通过基于 HTTP协议发送 curl localhost:12201/hsf/status 命令来检测服务是否successfully初始化，成功之后再启动web server （apache/nginx）

服务下线：

HSF基于Pandora的远程端口提供了offline命令，当HSF容器收到offline命令后 会把自身的所有服务从 configserver注销掉；（since 2.1.0.2版本开始）与此同时发送一个特殊的指令到所有连接的client，client收到指令后会自动把现在的服务器从地址池中快速invalide，从而不再发送新请求到offline的服务器(后续会进一步通过设计HSF自己的会话层协议来彻底解决。 PE需要配合，在关闭server时候，先基于

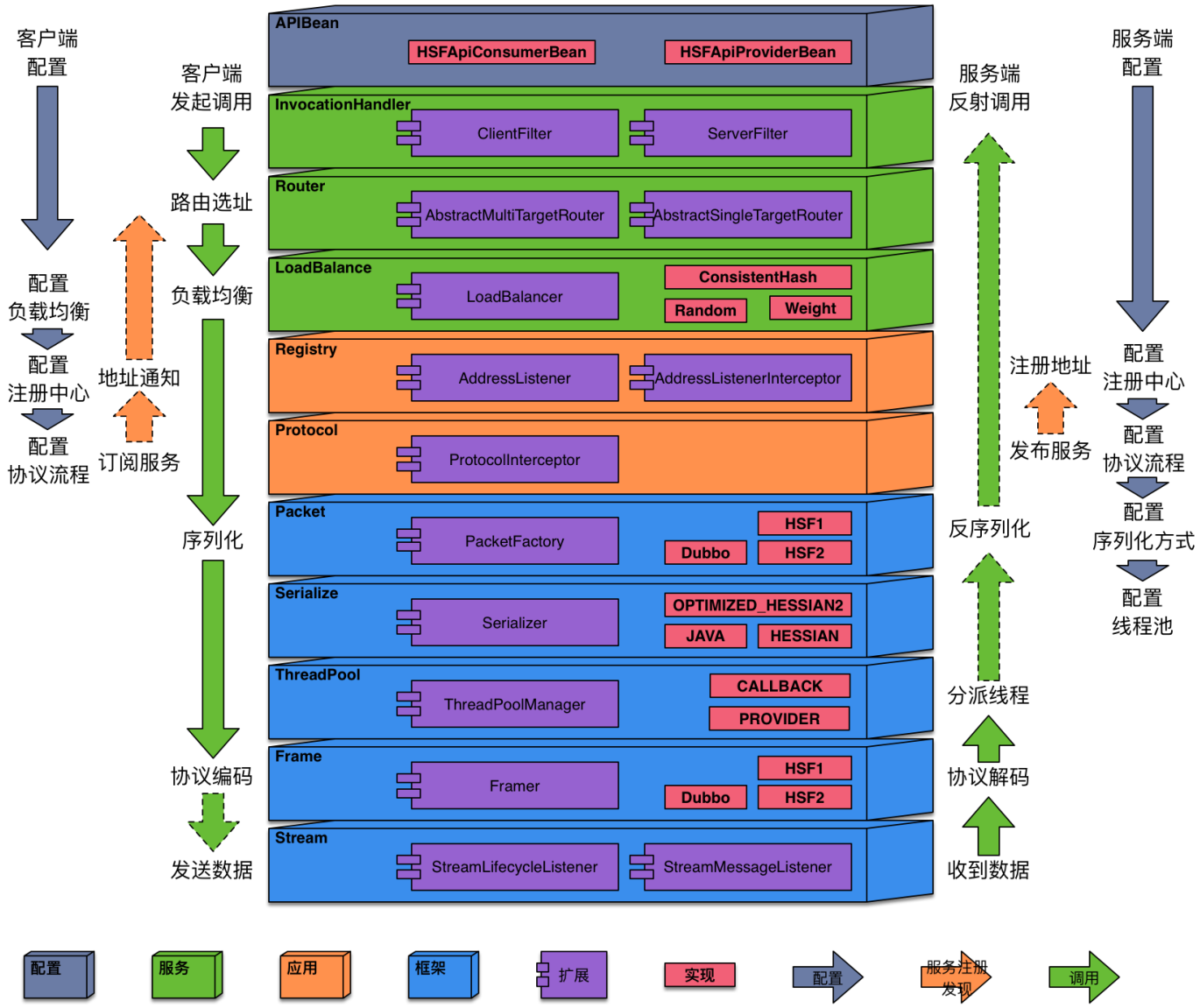
HTTP协议发送 curl localhost:12201/hsf/offline?k=hsf 命令来下线服务，等待30秒后，再执行shutdown jvm的操作。

注意事项：

Hsf面对大数据传输：（数据大小一般不要超过4k）

- 使用NOTIFY来传输；
- 使用metaq来传输；
- 应用方的客户端自己压缩和解压缩数据，不要把压缩和解压缩的压力放在服务端；
- 如果错误率不是很高的情况下，又想保证正确率，那就用重试机制，客户端调用时如果失败了，进行重试3次等操作。
- 增加服务端的机器，分散流量和压力，因为大数据也很容易把网卡跑满。

二级视图：

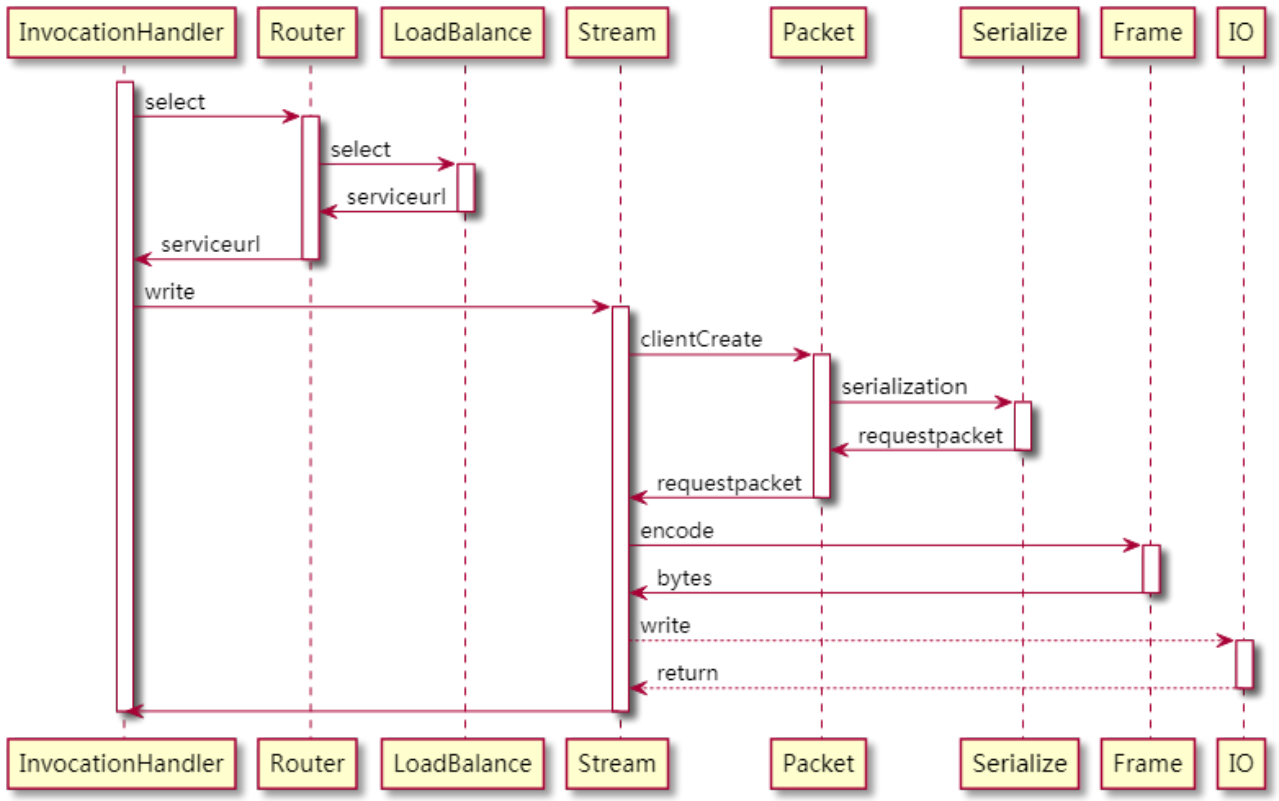


框架、应用、服务和配置。框架提供了基础功能，负责通信、线程、协议、序列化以及编解码相关的工作，它们提供了良好的抽象，框架之上的域只需要基于这些抽象就能完成一次高性能的调用。应用主要面向服务框架的注册和发现过程，是HSF完成分布式调用的基础，它用来支撑服务。服务的粒度比应用小，它包含了调用链路、地址路由以及负载均衡等功能。在服务之上是配置，用户使用API来对各层进行配置，并生成调用的代理或暴露服务。

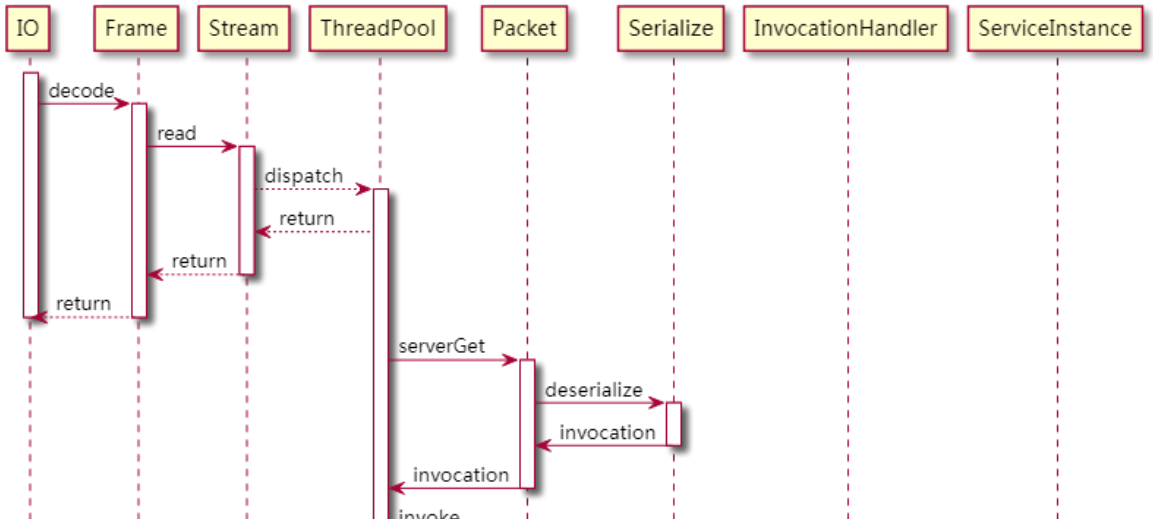
框架：对外提供基于长连接的远程异步调用能力；应用：提供服务发布订阅能力；服务：服务调用和消费的能力

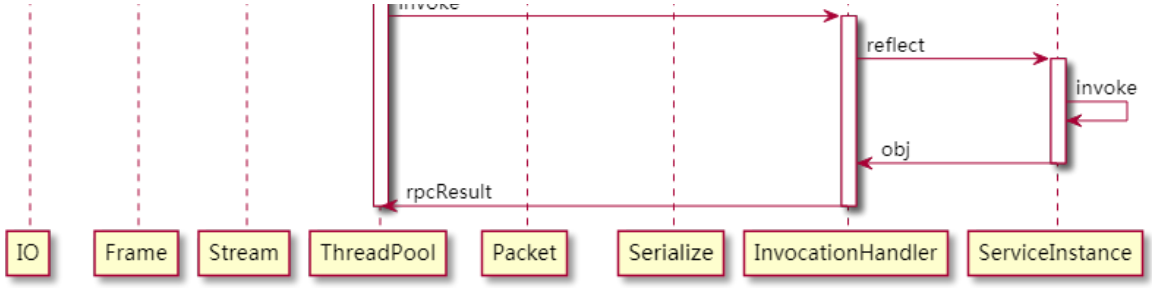
调用链路

客户端



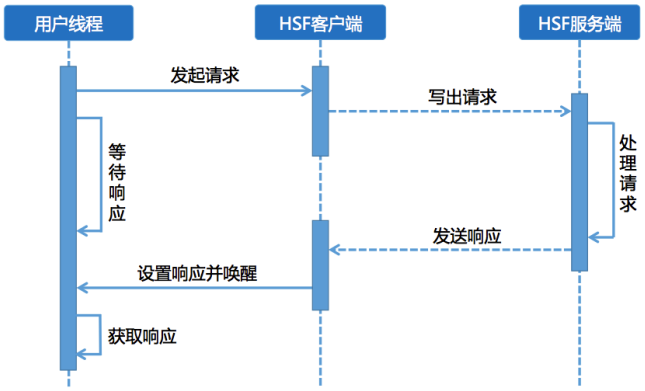
服务端



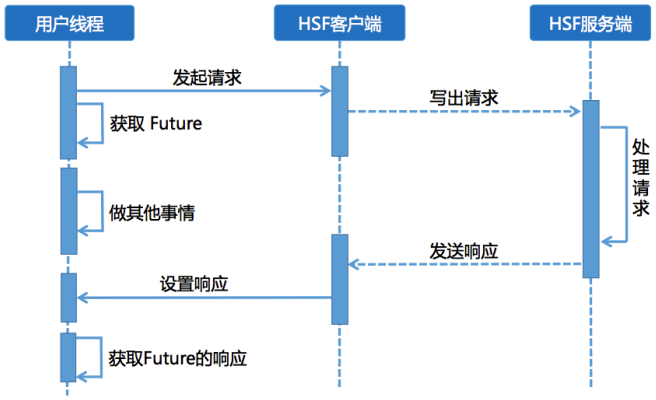


调用方式

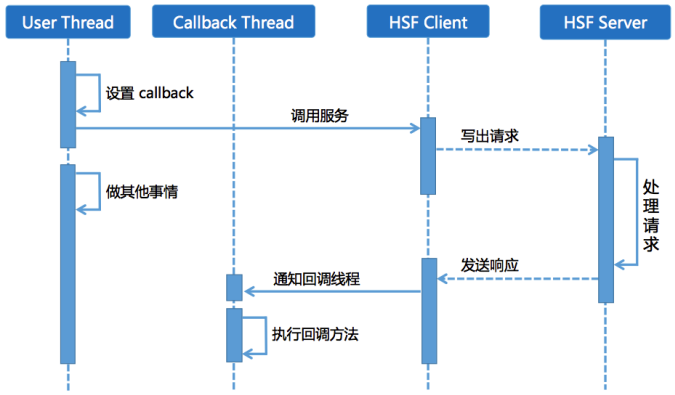
同步调用



Future调用



回调函数



手写RPC示例

序列化在服务端与传输端的POJO类

```
public class RPCPOJO implements Serializable {
    private static final long serialVersionUID = -2010762540622593769L;
    private String className;
    private String methodName;
    private Object[] args;
}
```

服务器端开启多线程进行处理

```
public class RPCServerProcesser implements Runnable {
    private Object object;
    private Socket socket;
    public RPCServerProcesser(Object object, Socket socket) {
        this.object = object;
        this.socket = socket;
    }
    @Override
    public void run() {
        ObjectInputStream objectInputStream = null;
        objectInputStream = new ObjectInputStream(socket.getInputStream());
        RPCPOJO rpcpojo = (RPCPOJO) objectInputStream.readObject();
        Object result = invoke(rpcpojo);
        ObjectOutputStream objectOutputStream
            = new ObjectOutputStream(socket.getOutputStream());
        objectOutputStream.writeObject(result);
        objectOutputStream.flush();
        objectInputStream.close();
        objectOutputStream.close();
    }

    private Object invoke(RPCPOJO rpcpojo){
        Object[] objects = rpcpojo.getArgs();
        Class<?>[] types = new Class[objects.length];
        for (int i = 0; i < objects.length; i++) {
            types[i] = objects[i].getClass();
        }
        Method method = null;
        method = object.getClass().getMethod(rpcpojo.getMethodName(), types);
        return method.invoke(object, objects);
    }
}
```

客户端创建动态代理

```
public class RPCClientProxy {

    public <T> T clientProxy(Class<T> interfaceCls, String host, int port) {
        return (T) Proxy.newProxyInstance(interfaceCls.getClassLoader(),
                new Class<?>[] {interfaceCls},
                new RPCClientInvocationHandler(host, port));
    }
}

public class RPCClientInvocationHandler implements InvocationHandler{
    private String host;
    private int port;
    public RPCClientInvocationHandler(String host, int port) {
        this.host = host;
        this.port = port;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwabl
e {
        RPCPOJO rpcpojo = new RPCPOJO();
        rpcpojo.setClassName(method.getDeclaringClass().getName());
        rpcpojo.setMethodName(method.getName());
        rpcpojo.setArgs(args);
        RPCTransport rpcTransorpt = new RPCTransport(host, port);
        return rpcTransorpt.send(rpcpojo);
    }
}
public class RPCTransport {
    private String host;
    private int port;
    public RPCTransport(String host, int port) {
        this.host = host;
        this.port = port;
    }
    private Socket newSocket() {
        Socket socket = null;
        socket = new Socket(host, port);
        return socket;
    }
    public Object send(Object object) {
        Socket socket = newSocket();
```

```
        ObjectOutputStream objectOutputStream = new ObjectOutputStream(socket.getOut
putStream());
        objectOutputStream.writeObject(object);
        objectOutputStream.flush();
        ObjectInputStream objectInputStream = new ObjectInputStream(socket.getInputS
tream());
        Object result = objectInputStream.readObject();
        return result;
    }
}
```

5.3 Http返回结果

```
//ResponseUtil类
public static <T> ResponseEntity<JSONObject> generateSuccessResponse(T result){
    JSONObject json = new JSONObject();
    json.put("success", true);
    json.put("data", result);

    return ResponseEntity.status(HttpStatus.OK).body(json);
}

public static ResponseEntity<JSONObject> generageFailResponse(String failReason){

    JSONObject json = new JSONObject();

    json.put("success", false);
    json.put("message", failReason);

    return ResponseEntity.status(HttpStatus.OK).body(json);
}
```

5.6 AOP

```
@Pointcut("execution(* com.taobao.mlab.controller..*.*(..))")
private void controllerAspect(){
}

@Around("controllerAspect()")
public Object around(ProceedingJoinPoint pjp) throws Throwable {
```

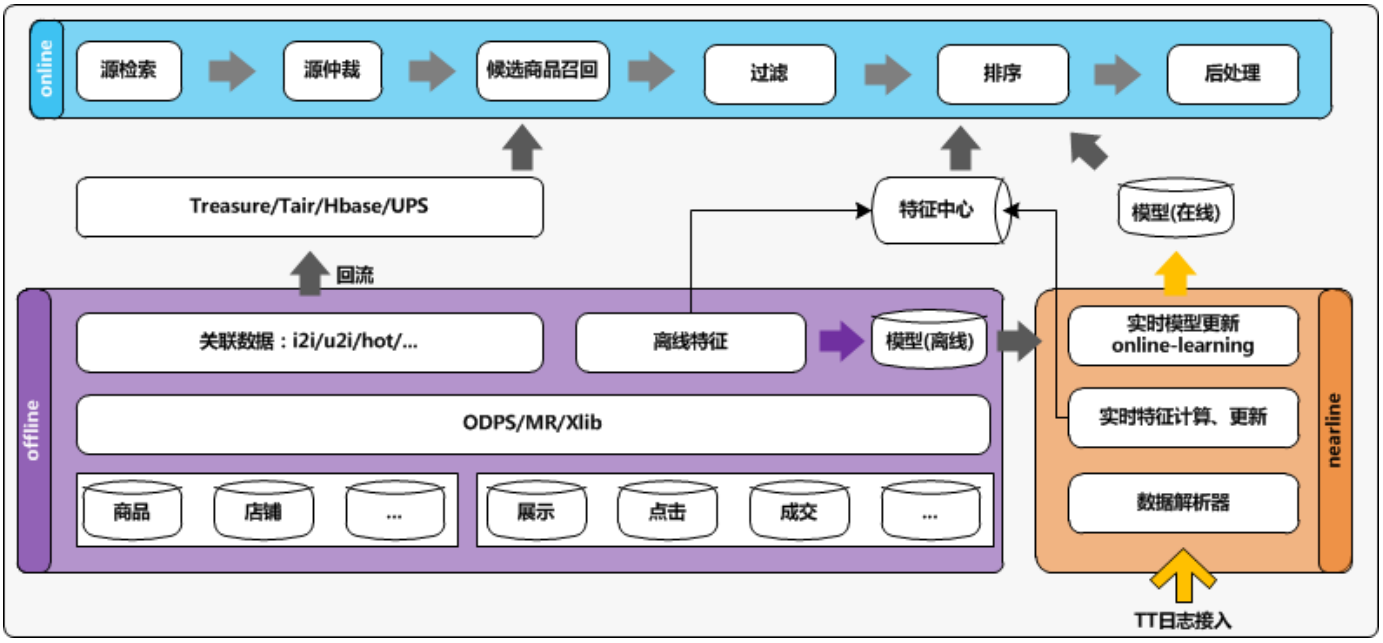
```
    LogEntity logEntity = new LogEntity();
    // 拦截的实体类，就是当前正在执行的controller
    Object target = pjp.getTarget();

    // 获得被拦截的方法
    Method method = null;
    try {
        Class[] parameterTypes = ((MethodSignature)pjp.getSignature()).getMethod().g
etParameterTypes();
        method = target.getClass().getMethod(pjp.getSignature().getName(), parameter
Types);
    } catch (Throwable e1) {
        e1.printStackTrace();
    }

    //没有方法名字或者没有SystemLog注解的都不答应，直接返回
    if (method == null || !method.isAnnotationPresent( SystemLog.class)) {
        return pjp.proceed();
    }else{
        HttpServletRequest request = ((ServletRequestAttributes) RequestContextHolde
r.getRequestAttributes()).getRequest();
        //获取登录用户账户,对于checkpreload.htm和taobao.status，不需要登录，所以没有用户信
息
```

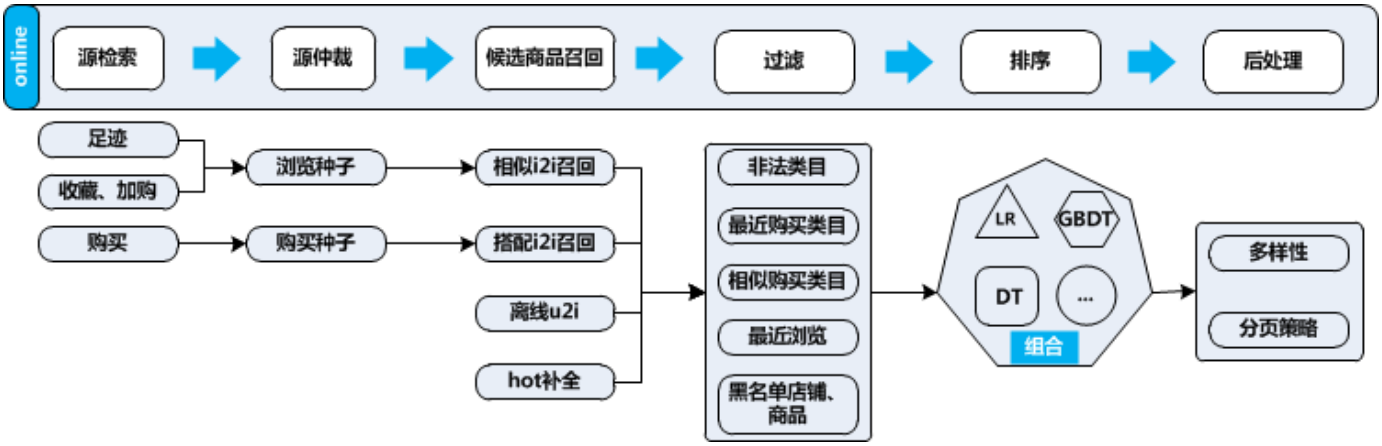
七、猜你喜欢

总体算法

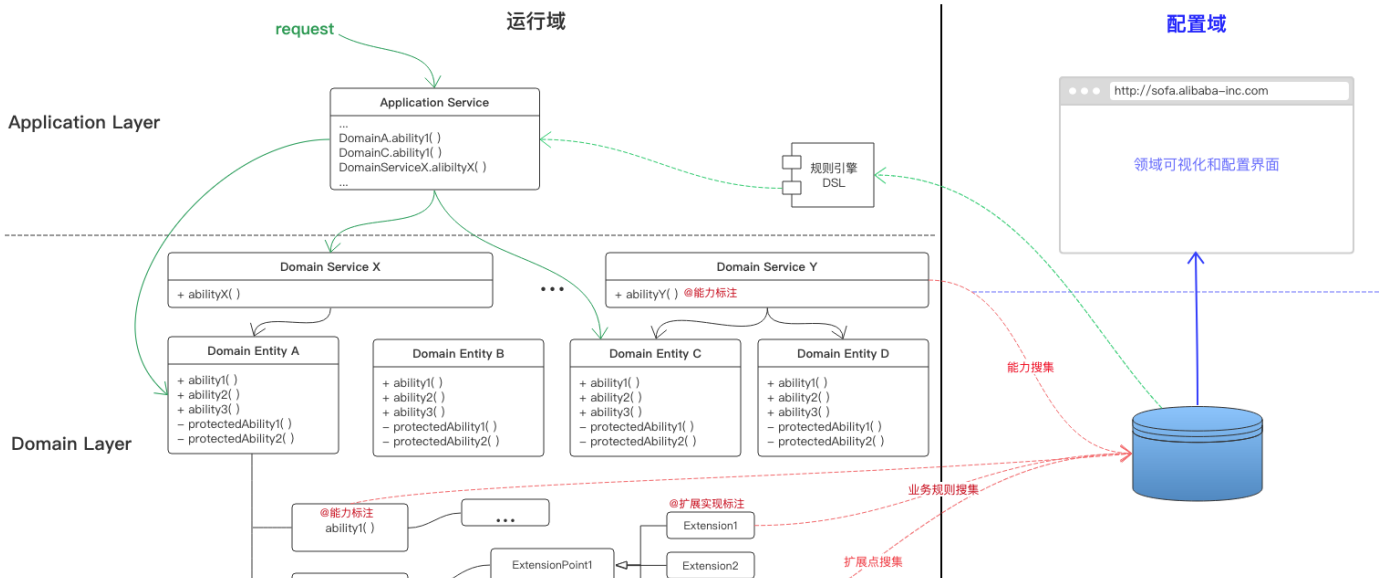


近实时部分：特征体系

Online PipeLine Case



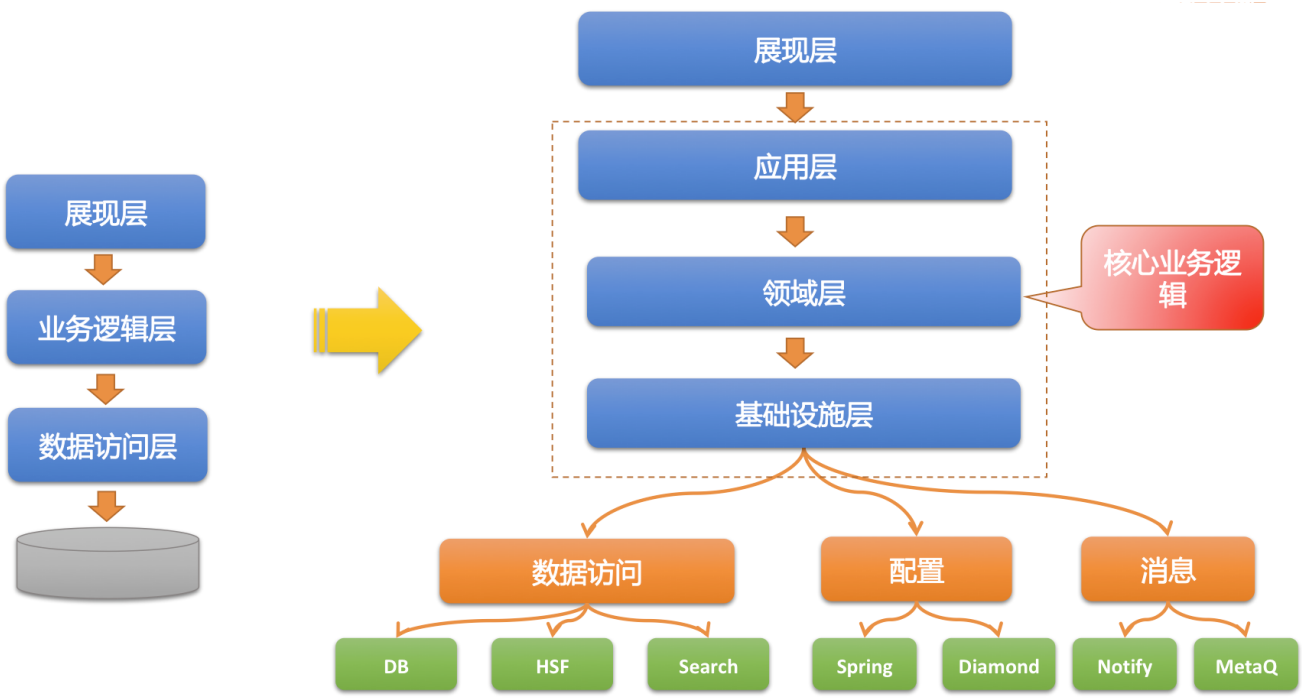
八、SOFA



分层设计

这一块的设计比较直观，整个应用层划分为三个大的层次，分别是App层，Domain层和Infrastructure层。

- **App层**主要负责获取输入，组装context，做输入校验，发送消息给领域层做业务处理，监听确认消息，如果需要的话使用MetaQ进行消息通知；
- **Domain层**主要是通过领域服务（Domain Service），领域对象（Domain Object）的交互，对上层提供业务逻辑的处理，然后调用下层Repository做持久化处理；
- **Infrastructure层**主要包含Repository，Config，Common和message，Repository负责数据的CRUD操作，这里我们借用了盒马的数据通道（Tunnel）的概念，通过Tunnel的抽象概念来屏蔽具体的数据来源，来源可以是MySQL，NoSql，Search，甚至是HSF等；Config负责应用的配置；Common是一些工具类；负责message通信的也应该放在这一层。



这里需要注意的是从其他系统获取的数据是有界上下文（Bounded Context）下的数据，为了弥合Bounded Context下的语义Gap，通常有两种方式，一个是用大领域（Big Domain）把两边的差异都合起来，另一个是增加防腐层（Anticorruption Layer）做转换。什么是Bounded Context？ 简单阐述一下，就是我们的领域概念是有作用范围的（Context）的，例如摇头这个动作，在中国的Context下表示NO，但是在印度的Context下却是YES。

规范设计（Class规范）

API Classes

Client Request	⟺	XXXCmd.java
Client Object	⟺	XXXCO.java
API Service	⟺	XXXServiceI.java

Domain Classes

Domain Object	⟺	XXXE.java
Value Object	⟺	XXXV.java
Domain Factory	⟺	XXXFactory.java
Repository	⟺	XXXRepository.java
Business Rule	⟺	XXXRule.java
Domain Service	⟺	XXXDomainService

App Classes

Command Executor	⟺	XXXCmdExe.java
Query Executor	⟺	XXXQueryExe.java
Command Interceptor	⟺	XXXInterceptor.java
Extension Point	⟺	XXXExtPt.java
Extension	⟺	XXXExt.java
Parameter Validator	⟺	XXXValidator.java
Object Convertor	⟺	XXXConvertor.java
Parameter Assembler	⟺	XXXAssembler.java

Tunnel Classes

Data Object	⟺	XXXDO.java
DB DAO	⟺	XXXTunnel.java

方法名约定：

层次	类	方法名约定
App层	接口服务	新增：save 修改：modify 查询：get（单个）、list（多个、分页） 统计：count 删除：remove
Domain层	Domain实体	尽量避免CRUD形式的命名，要体现业务语义
Tunnel层	Tunnel对象	新增：create 修改：update 查询： get(单个)、list(多个)、page（分页） 删除：delete 统计：count

错误码约定：

异常主要分为**系统异常和业务异常**，系统异常是指不可预期的系统错误，如网络连接，服务调用超时等，是可以retry的；而业务异常是指有明确业务语义的错误，再细分的话，又可以分为参数异常和业务逻辑异常，参数异常是指用户过来的请

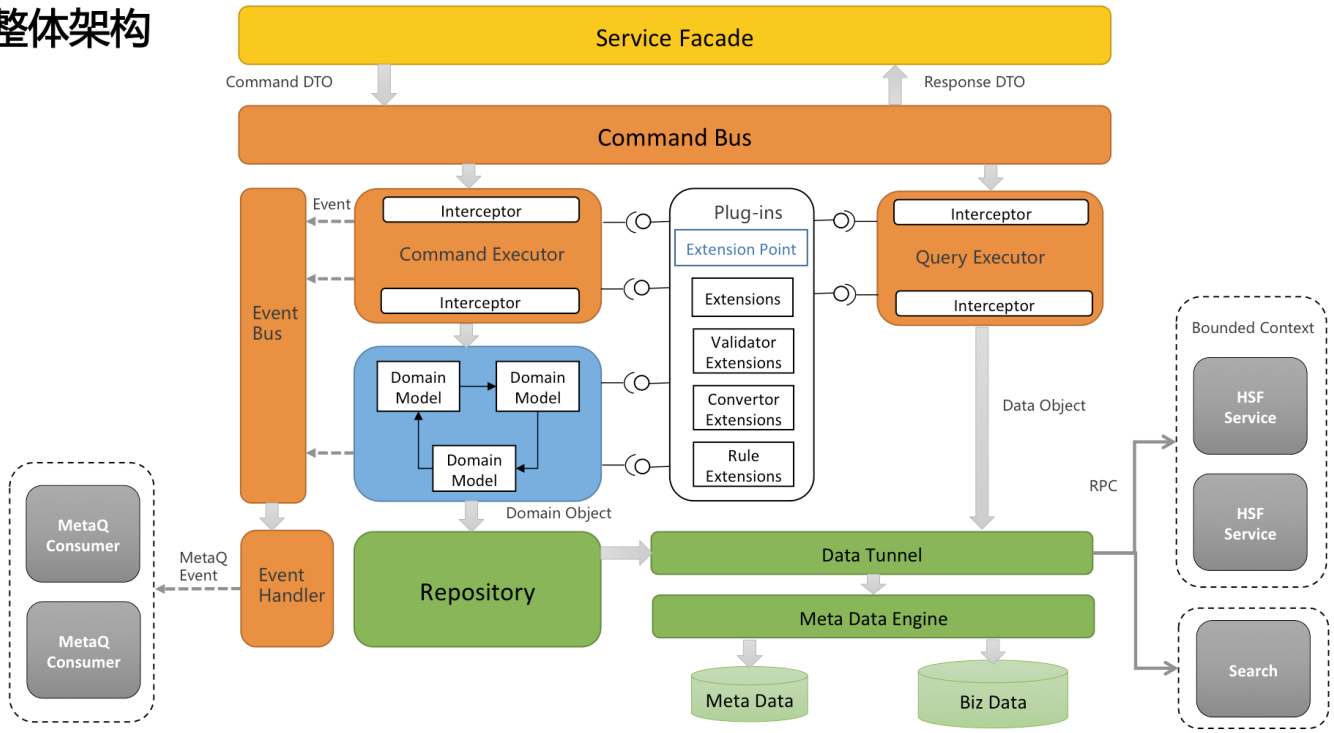
求不准确，逻辑异常是指不满足系统约束，比如客户已存在。业务异常是不需要retry的。

我们的错误码主要有3部分组成：**类型+场景+自定义标识**

错误类型	错误码约定	举例
参数异常	P_XX_XX	P_CAMPAIGN_NameNotNull: 运营活动名不能为空
业务异常	B_XX_XX	B_CAMPAIGN_NameAlreadyExist: 运营活动名已存在
系统异常	S_XX_ERROR	S_DATABASE_ERROR: 数据库错误

我们的架构原则很简单，即在**高内聚，低耦合，可扩展，易理解**大的指导思想下，尽可能的贯彻OO的设计思想和原则。我们最终形成的架构是集成了**扩展点+元数据+CQRS+DDD的思想**，关于元数据前面没怎么提到，这里稍微说一下，对于字段扩展，简单一点的解决方案就是预留扩展字段，复杂一点的就是使用元数据引擎。使用元数据的好处是不仅能支持字段扩展，还提供了丰富的字段描述，等于是为以后的SaaS化配置提供了可能性，所以我们选择了使用元数据引擎。和DDD一样，元数据也是可选的，如果对没有字段扩展的需求，就不要用。最后的整体架构图如下：

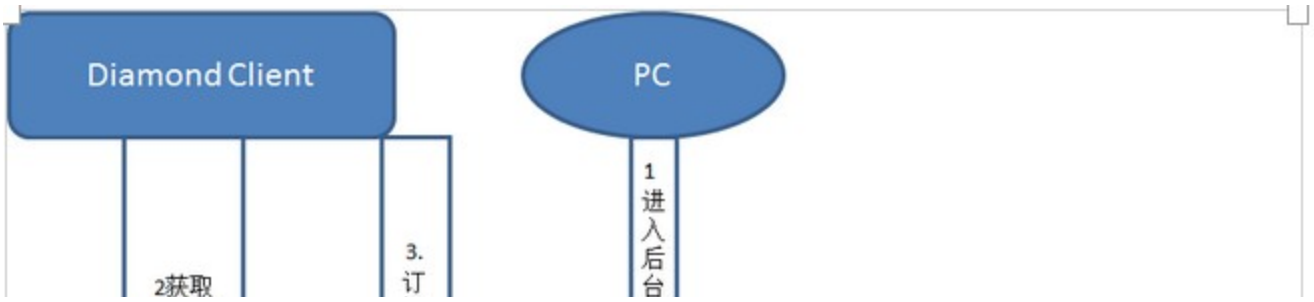
整体架构

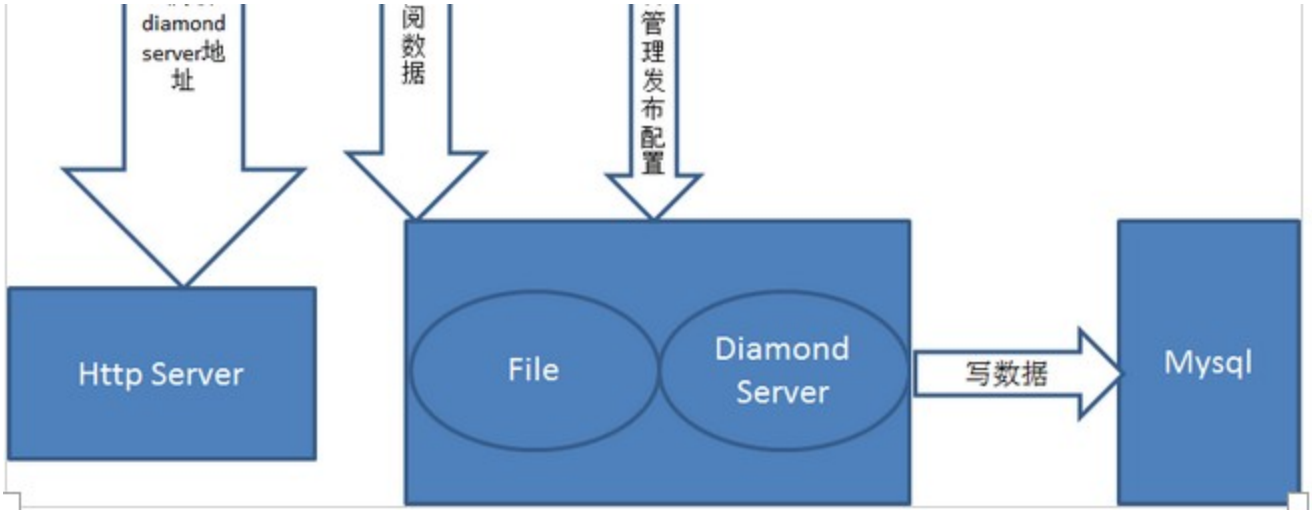


九、Diamond

1.架构图

diamond





1. diamond发布流程

   1） 写入到数据库

   2） 写入本地文件

   3） 通知其他服务器

   4） 其他服务器同步数据至本地

   diamond-server将数据存储在mysql，diamond server本地，diamond client容灾目录，diamond client快照目录（上一次正确配置）中。mysql是中心，diamond认为存储在mysql中的数据绝对正确。

2. diamond订阅流程

   1） 获取diamond server地址列表

   client启动时先从http server获取diamond server地址列表，并写入client本地；http server 高于client本地。当client无法从http server获取地址列表时，client会使用本地保存的地址列表，已保证地址为最新的。

   2） client获取配置数据

   client从diamond server获取配置数据的过程实际就是一个拼接http url，使用httpclient调用http method的过程，Diamond server收到请求后，不读取数据库而是将本地文件中配置数据返回给client；为了避免短时间内大量请求发送到diamond server，启动获取一次数据后会将数据存到过期缓存和本地容灾文件中，以后都先从缓存和本地获取；

   3） 获取数据变化定时更新配置数据

   Client启动获取一次数据后，会将该份数据的MD5保存在内存中，并在启动时启动一个定时任务定时去检验数据是否发生变化，如果发生变化了，client自动重新获取配置数据

3. 数据间的同步：

   diamond server间：server写数据时，先将数据写入mysql，然后写入本地文件，写入完成后发送一个HTTP请求给集群中的其他server，其他server收到请求，从mysql中dump刚刚写入的数据至本地文件；

   mysql与本地间：server启动后会启动一个定时任务，定时从mysql中dump所有数据至本地文件；

   client缓存： client端实现了一个带有过期时间的缓存；

   client感知数据变化： server会内存所有数据的md5，client内存所有它订阅的配置的md5，client定

时通过请求server比对数据是否变化（通过md5）

polling： 如果client感知到数据变化，更新之，并写client本地缓存

4. **数据的获取**

   客户端采用推拉结合的策略在长连接和短连接之间取得一个平衡，让服务端不用太关注连接的管理，又可以获得长连接的及时性。

   • 客户端发起一个对比请求到服务端，请求中包含客户端订阅的数据的指纹

   • 服务端检查客户端的指纹是否与最新数据匹配

   o 如果匹配，服务端持有连接

   o 如果30秒内没有相关数据变化，服务端持有连接30秒后释放

   o 如果30秒内有相关数据变化，服务端立即返回变化数据的ID

   • 如果不匹配，立即返回变化数据的ID

   • 客户端根据变化数据的ID去服务端获取最新的内容

   Diamond通过这种推拉结合的方式，让客户端逻辑尽量简单，而且高效。

5. **Diamond的容灾机制，可以在如下场景还能保证持久化配置获取的可用性。**

   • mysql主库不可用时，启用备库后，继续提供配置读、写服务。

   • mysql主备库都不可用时，通过服务端缓存，继续提供配置读服务。

   • mysql主备库、diamond-server集群整体不可用时，客户端可以借助缓存目录继续使用配置读服务，借助容灾目录使用配置写服务。

   因此，只有mysql主备库不可用，diamond-server集群整体不可用，客户端缓存目录数据丢失这几种情况同时发生时，Diamond服务才不可用。

6. **应用场景**

   •配置项动态变更推送 – 典型用户：HSF路由规则、各个系统的动态开关

   • 数据源的动态切换、连接参数调整 – 典型用户：TDDL动态数据源

   • 分布式组件间状态同步 – 典型用户：秒杀系统整点同步题库

   • 指定范围的配置项变更推送 – 典型用户：tbsession新配置的beta发布

   • 持久化配置聚合能力 – 典型用户：ring权限系统，notify订阅关系

configServer dimond

非持久性 持久性

主动推 推拉结合

场景：hsf服务调用功能，notify hsf规则设置

十、ConfigServer

非持久化配置中心，是一个基于“发布-订阅”模型的分布式通信框架，是一个无master，无中心化的结构，所有节点上的数据是一致的，常用场景：hsf服务调用功能，notify。

特点：

主动推

客户端（不管是发布者还是订阅者）和Server之间维持着一个TCP长连接，server通过连接往里面写数

据，相当于主动的通知了客户端。
数据聚合
ConfigServer会自动把注册相同dataId的多个客户端聚合在一起
适合非持久型数据
数据的生命周期和发布者的TCP连接的生命周期相同，一旦发布者连接断开后，发布的数据就失效了。
动态感知发布者断连
两种情况
• 发布者 正常断开连接 （TCP连接）：通过连接事件，TCP连接断开时，应用层ConfigServer是可以感知到的，然后ConfigServer把断开的连接对应的数据删除。
• 发布者 异常断开连接 ，比如直接拔网线。这样应用层是无法直接感知到的。所以需要一种通知机制： 心跳机制 。发布者每隔3秒向ConfigServer发送一个心跳包，用以维持与ConfigServer的连接。
ConfigServer定时查看当前时间与上次更新时间的间隔，判断连接是否失效。如果失效，就删除这个连接对应的数据，同时通知订阅者。

原理：
configclient和configserver交互过程

1. client端发起post请求到vip
2. vip返回所有的configserver服务器地址列表给client
3. client随机选取一个地址，跟configserver建立链接，并通过定时发送心跳包，保持链接
4. client向configserver发起服务订阅请求
5. configserver推送服务提供者的地址列表给client

十一、数据一致性

问题一：数据一致性

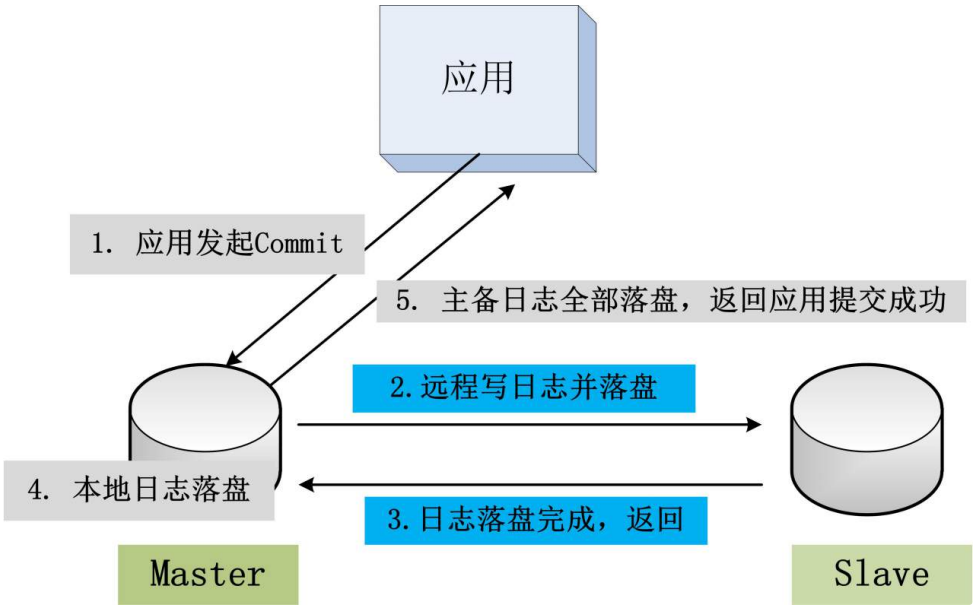
问：脱离了共享存储，传统关系型数据库就无法做到主备强一致吗？

答：我的答案，是No。哪怕不用共享存储，任何数据库，也都可以做到主备数据的强一致。Oracle如此，MySQL如此，PostgreSQL如此，OceanBase也如此。

如何实现主备强一致？大家都知道数据库中最重要的一个技术：WAL（Write-Ahead-Logging）。更新操作写日志（Oracle Redo Log，MySQL Binlog等），事务提交时，保证将事务产生的日志先刷到磁盘上，保证整个事务的更新操作数据不丢失。那实现数据库主备数据强一致的方法也很简单：

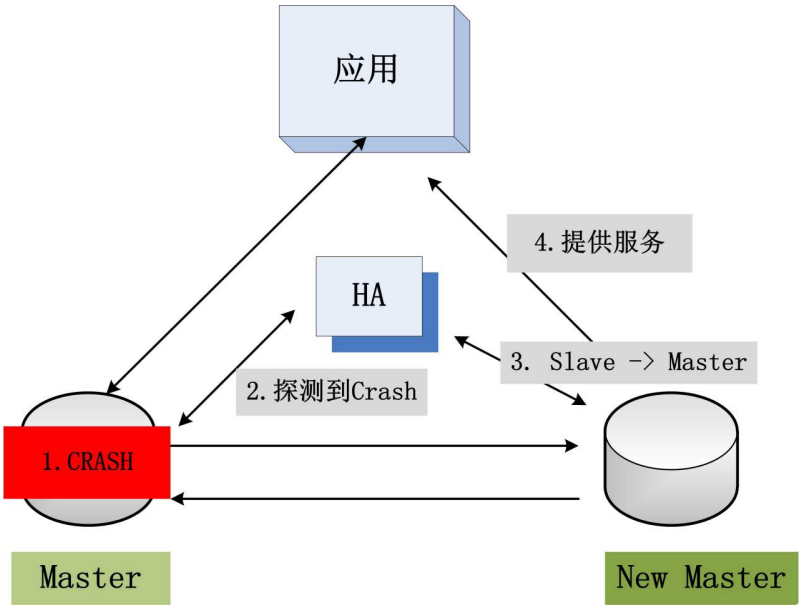
- 事务提交的时候，同时发起两个写日志操作，一个是将日志写到本地磁盘的操作，另一个是将日志同步到备库并且确保落盘的操作；
- 主库此时等待两个操作全部成功返回之后，才返回给应用方，事务提交成功；

整个事务提交操作的逻辑，如下图所示：



上图所示，由于事务提交操作返回给应用时，事务产生的日志在主备两个数据库上都已经存在了，强同步。因此，此时主库Crash的话，备库提供服务，其数据与主库是一致的，没有任何事务的数据丢失问题。主备数据强一致实现。用过Oracle的朋友，应该都知道Oracle的Data Guard，可工作在 最大性能，最大可用，最大保护 三种模式下，其中第三种 最大保护 模式，采用的就是上图中的基本思路。

实现数据的强同步实现之后，接下来到了考虑可用性问题。现在已经有主备两个数据完全一致的数据库，备库存在的主要意义，就是在主库出故障时，能够接管应用的请求，确保整个数据库能够持续的提供服务：主库Crash，备库提升为主库，对外提供服务。此时，又涉及到一个决策的问题，主备切换这个操作谁来做？人当然可以做，接收到主库崩溃的报警，手动将备库切换为主库。但是，手动的效率是低下的，更别提数据库可能会随时崩溃，全部让人来处理，也不够厚道。一个HA（High Availability）检测工具应运而生：HA工具一般部署在第三台服务器上，同时连接主备，当其检测到主库无法连接，就切换备库，很简单的处理逻辑，如下图所示：



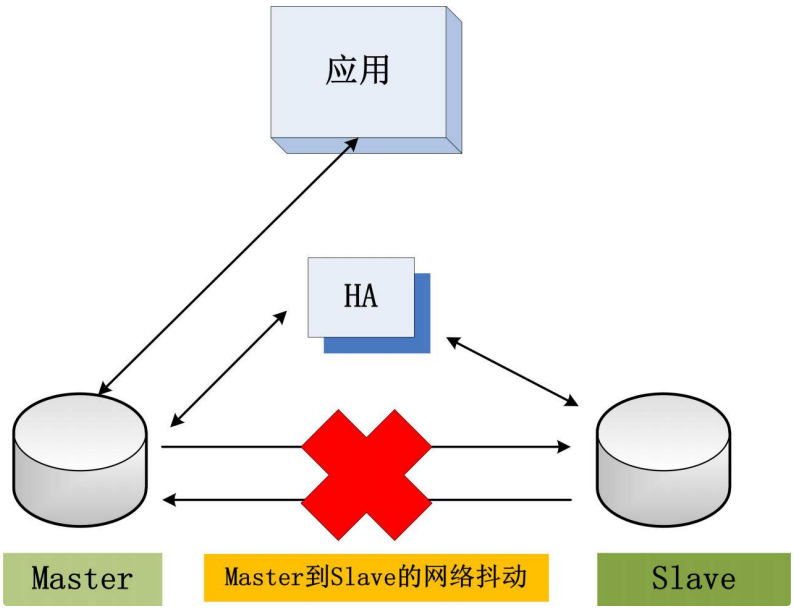
HA软件与主备同时连接，并且有定时的心跳检测。主库Crash后，HA探测到，发起一个将备库提升为主库的操作（修改备库的VIP或者是DNS，可能还需要将备库激活等一系列操作），新的主库提供对外服务。此

时，由于主备的数据是通过日志强同步的，因此并没有数据丢失，数据一致性得到了保障。

有了基于日志的数据强同步，有了主备自动切换的HA软件，是不是就一切万事大吉了？我很想说是，确实这个架构已经能够解决90%以上的问题，但是这个架构在某些情况下，也埋下了几个比较大的问题。

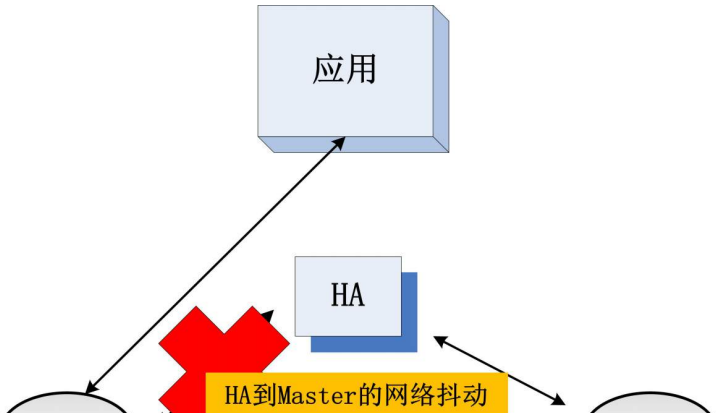
首先，一个一目了然的问题，主库Crash，备库提升为主库之后，此时的数据库是一个单点，原主库重启的这段时间，单点问题一直存在。如果这个时候，新的存储再次Crash，整个系统就处于不可用状态。此问题，可以通过增加更多副本，更多备库的方式解决，例如3副本（一主两备），此处略过不表。

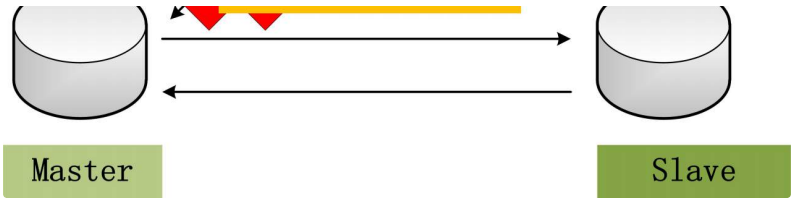
其次，在主备环境下，处理主库挂的问题，算是比较简单的，决策简单：主库Crash，切换备库。但是，如果不是主库Crash，而是网络发生了一些问题，如下图所示：



若Master与Slave之间的网络出现问题，例如：断网，网络抖动等。此时数据库应该怎么办？Master继续提供服务？Slave没有同步日志，会数据丢失。Master不提供服务？应用不可用。在Oracle中，如果设置为 最大可用 模式，则此时仍旧提供服务，允许数据不一致；如果设置为 最大保护 模式，则Master不提供服务。因此，在Oracle中，如果设置为 最大保护 模式，一般建议设置两个或以上的Slave，任何一个Slave日志同步成功，Master就继续提供服务，提供系统的可用性。

网络问题不仅仅出现在Master和Slave之间，同样也可能出现在HA与Master，HA与Slave之间。考虑下面的这种情况：

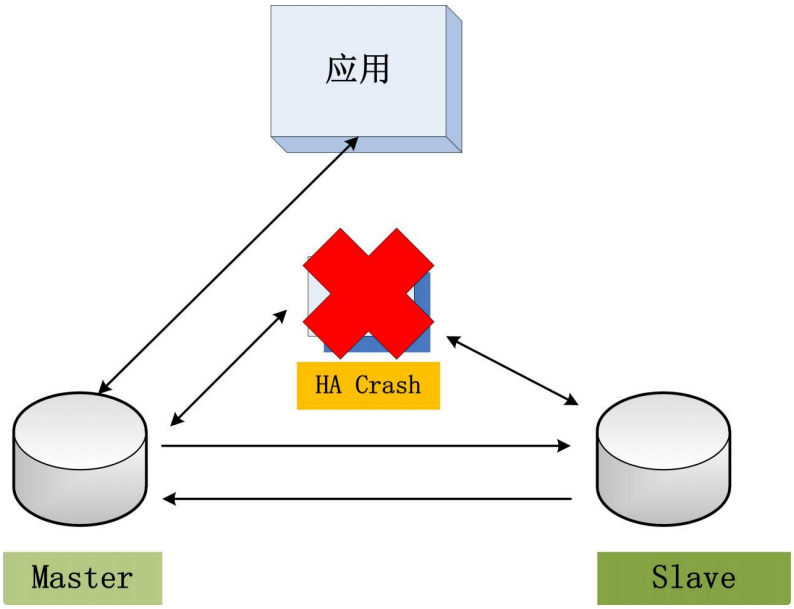




HA与Master之间的网络出现问题，此时HA面临两个抉择：

- HA到Master之间的连接不通，认为主库Crash。选择将备库提升为主库。但实际上，只是HA到Master间的网络有问题，原主库是好的（没有被降级为备库，或者是关闭），仍旧能够对外提供服务。新的主库也可以对外提供服务。两个主库，产生双写问题，最为严重的问题。
- HA到Master之间的连接不同，认为是网络问题，主库未Crash。HA选择不做任何操作。但是，如果这时实际上确实是主库Crash了，HA不做操作，数据库不对外提供服务。此时，双写问题避免了，但是应用的可用性受到了影响。

最后，数据库会出现问题，数据库之间的网络会出现问题，那么再考虑一层，HA软件本身也有可能出现问题。如下图所示：



如果是HA软件本身出现了问题，怎么办？我们通过部署HA，来保证数据库系统在各种场景下的持续可用，但是HA本身的持续可用谁来保证？难道我们需要为HA做主备，然后再HA之上再做另一层HA？一层层加上去，子子孙孙无穷尽也 … …

其实，上面提到的这些问题，其实就是经典的分布式环境下的一致性问题（Consensus），近几年比较火热的Lamport老爷子的Paxos协议，Stanford大学最近发表的Raft协议，都是为了解决这一类问题。（对Raft协议感兴趣的朋友，可以再看一篇Raft的动态演示PPT：Understandable Distributed Consensus）

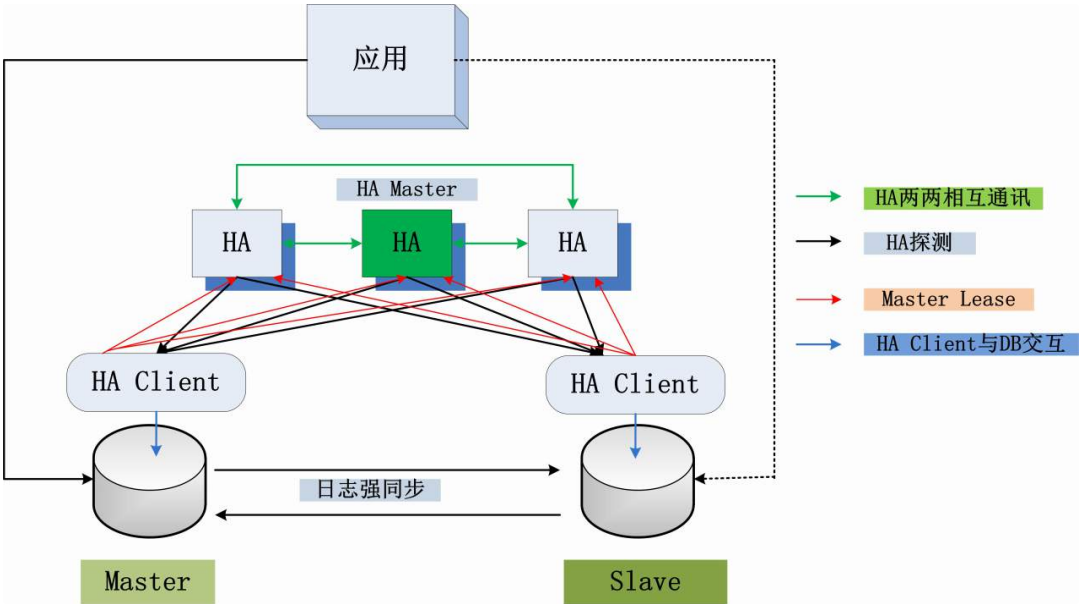
问题二：分区可用性

前面，我们回答了第一个问题，数据库如果不使用共享存储，能否保证主备数据的强一致？答案是肯定的：可以。但是，通过前面的分析，我们又引出了第二个问题：如何保证数据库在各种情况下的持续可用？至少前面提到的HA机制无法保证。那么是否可以引入类似于Paxos，Raft这样的分布式一致性协议，来解决上面提到的各种问题呢？

答案是可以的，我们可以通过引入类Paxos，Raft协议，来解决上面提到的各类问题，保证整个数据库系统的持续可用。考虑仍旧是两个数据库组成的主备强一致系统，仍旧使用HA进行主备监控和切换，再回顾一下上一节新引入的两个问题：

- HA软件自身的可用性如何保证？
- 如果HA软件无法访问主库，那么这时到底是主库Crash了呢？还是HA软件到主库间的网络出现问题了呢？如何确保不会同时出现两个主库，不会出现双写问题？
- 如何在解决上面两个问题的同时，保证数据库的持续可用？

为了解决这些问题，新的系统如下所示：



相对于之前的系统，可以看到这个系统的复杂性明显增高，而且不止一成。数据库仍旧是一主一备，数据强同步。但是除此之外，多了很多变化，这些变化包括：

- 数据库上面分别部署了HA Client；
- 原来的一台HA主机，扩展到了3台HA主机。一台是HA Master，其余的为HA Participant；
- HA主机与HA Client进行双向通讯。HA主机需要探测HA Client所在的DB是否能够提供服务，这个跟原有一致。但是，新增了一条HA Client到HA主机的Master Lease通讯。

这些变化，能够解决上面的两个问题吗？让我们一个一个来分析。首先是：HA软件自身的可用性如何保证？

从一台HA主机，增加到3台HA主机，正是为了解决这个问题。HA服务，本身是无状态的，3台HA主机，可以通过Paxos/Raft进行自动选主。选主的逻辑，我这里就不做赘述，不是本文的重点，想详细了解其实现的，可以参考互联网上洋洋洒洒的关于Paxos/Raft的相关文章。总之，通过部署3台HA主机，并且引入Paxos/Raft协议，HA服务的高可用可以解决。HA软件的可用性得到了保障。

第一个问题解决，再来看第二个问题：如何识别出当前是网络故障，还是主库Crash？如何保证任何情况下，数据库有且只有一个主库提供对外服务？

通过在数据库服务器上部署HA Client，并且引入HA Client到HA Master的租约（Lease）机制，这第二个问题同样可以得到完美的解决。所谓HA Client到HA Master的租约机制，就是说图中的数据库实例，不是永远持有主库（或者是备库）的权利。当前主库，处于主库状态的时间是有限制的，例如：10秒。每隔10秒，HA Client必须向HA Master发起一个新的租约，续租它所在的数据库的主库状态，只要保证每10秒收到一个来自HA Master同意续租的确认，当前主库一直不会被降级为备库。

第二个问题，可以细分为三个场景：

- 场景一：主库Crash，但是主库所在的服务器正常运行，HA Client运行正常

主库Crash，HA Client正常运行。这种场景下，HA Client向HA Master发送一个放弃主库租约的请求，HA Master收到请求，直接将备库提升为主库即可。原主库起来之后，作为备库运行。

- 场景二：主库所在的主机Crash。（主库和HA Client同时Crash）

此时，由于HA Client和主库同时Crash，HA Master到HA Client间的通讯失败。这个时候，HA Master还不能立即将备库提升为主库，因为区分不出场景二和接下来的场景三（网络问题）。因此，HA Master会等待超过租约的时间（例如：12秒），如果租约时间之内仍旧没有续租的消息。那么HA Master将备库提升为主库，对外提供服务。原主库所在的主机重启之后，以备库的状态运行。

- 场景三：主库正常，但是主库到HA Master间的网络出现问题

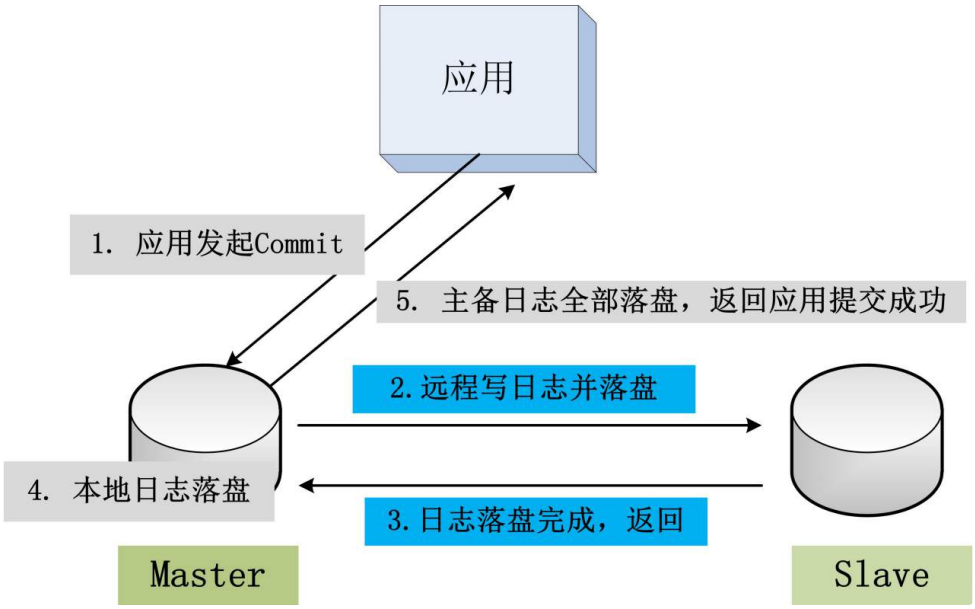
对于HA Master来说，是区分不出场景二和场景三的。因此，HA Master会以处理场景二同样的逻辑处理场景三。等待超过租约的时间，没有收到续租的消息，提升原备库为主库。但是在提升备库之前，原主库所在的HA Client需要做额外的一点事。原主库HA Client发送给HA Master的续租请求，由于网络问题，一直没有得到响应，超过租约时间，主动将本地的主库降级为备库。如此一来，待HA Master将原备库提升为主库时，原来的主库已经被HA Client降级为备库。双主的情况被杜绝，应用不可能产生双写。

同过以上三个场景的分析，问题二同样在这个架构下被解决了。而解决问题二的过程中，系统最多需要等待租约设定的时间，如果租约设定为10秒，那么出各种问题，数据库停服的时间最多为10秒，基本上做到了持续可用。这个停服的时间，完全在于租约的时间设置。

到这儿，基本可以说，要实现一个持续可用（分区可用性保证），并且保证主备数据强一致的数据库系统，是完全没问题的。在现有数据库系统上做改造，也是可以的。但是，如果考虑到实际的实现，这个复杂度是非常高的。数据库的主备切换，是数据库内部实现的，此处通过HA Master来提升主库；通过HA Client来降级备库；保证数据库崩溃恢复后，恢复为备库；通过HA Client实现主库的租约机制；实现HA主机的可用性；所有的这些，在现有数据库的基础上实现，都有着相当的难度。能够看到这儿，而且有兴趣的朋友，可以针对此问题进行探讨J

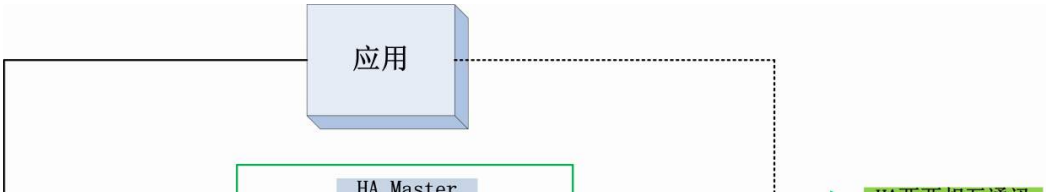
问题三：性能

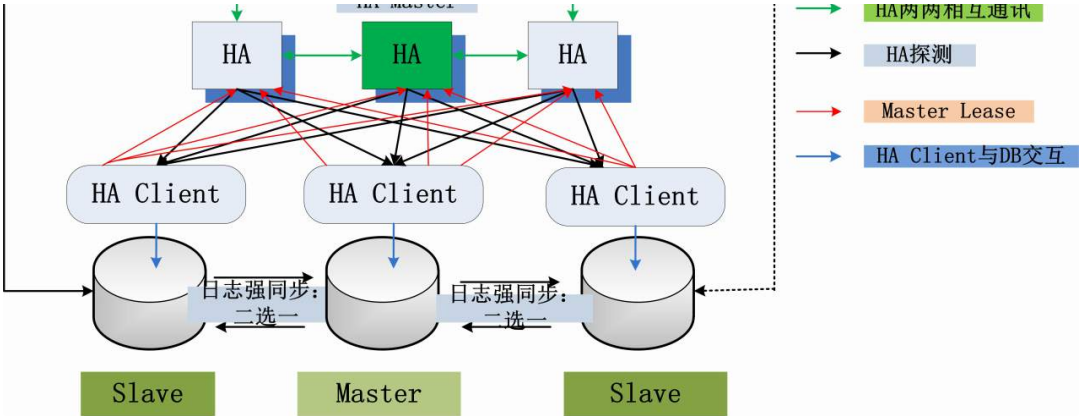
数据一致性，通过日志的强同步，所有数据均可以解决。分区可用性，在出现任何异常情况时仍旧保证系统的持续可用，可用在数据强同步的基础上引入Paxos/Raft等分布式一致性协议来解决，虽然这个目前没有成熟的实现。接下来再让我们来看看一个很多朋友都很感兴趣的问题：如何在保证强同步的基础上，同时保证高性能？回到我们本文的第一幅图：



为了保证数据强同步，应用发起提交事务的请求时，必须将事务日志同步到Slave，并且落盘。相对于异步写Slave，同步方式多了一次Master到Slave的网络交互，同时多了一次Slave上的磁盘sync操作。反应到应用层面，一次Commit的时间一定是增加了，具体增加了多少，要看主库到备库的网络延时和备库的磁盘性能。

为了提高性能，第一个很简单的想法，就是部署多个Slave，只要有一个Slave的日志同步完成返回，加上本地的Master日志也已经落盘，提交操作就可以返回了。多个Slave的部署，对于消除瞬时的网络抖动，非常有效果。在Oracle的官方建议中，如果使用最大保护模式，也建议部署多个Slave，来最大限度的消除网络抖动带来的影响。如果部署两个Slave，新的部署架构图如下所示：





新增一个Slave，数据三副本。两个Slave，只要有一个Slave日志同步完成，事务就可以提交，极大地减少了某一个网络抖动造成的影响。增加了一个副本之后，还能够解决当主库Crash之后的数据安全性问题，哪怕主库Crash，仍旧有两个副本可以提供服务，不会形成单点。

但是，在引入数据三副本之后，也新引入了一个问题：主库Crash的时候，到底选择哪一个备库作为新的主库？当然，选主的权利仍旧是HA Master来行使，但是HA Master该如何选择？这个问题的简单解决可以使用下面的几个判断标准：

1. 日志优先。两个Slave，哪个Slave拥有最新的日志，则选择这个Slave作为新的主库。
2. 主机层面排定优先级。如果两个Slave同时拥有最新的日志，那么该如何选择？此时，选择任何一个都是可以的。例如：可以根据Slave主机IP的大小进行选择，选择IP小的Slave作为新的主库。同样能够解决问题。

新的主库选择出来之后，第一件需要做的事，就是将新的Master和剩余的一个Slave，进行日志的同步，保证二者日志达到一致状态后，对应用提供服务。此时，三副本问题就退化为了两副本问题，三副本带来的防止网络抖动的红利消失，但是由于两副本强同步，数据的可靠性以及一致性仍旧能够得到保障。

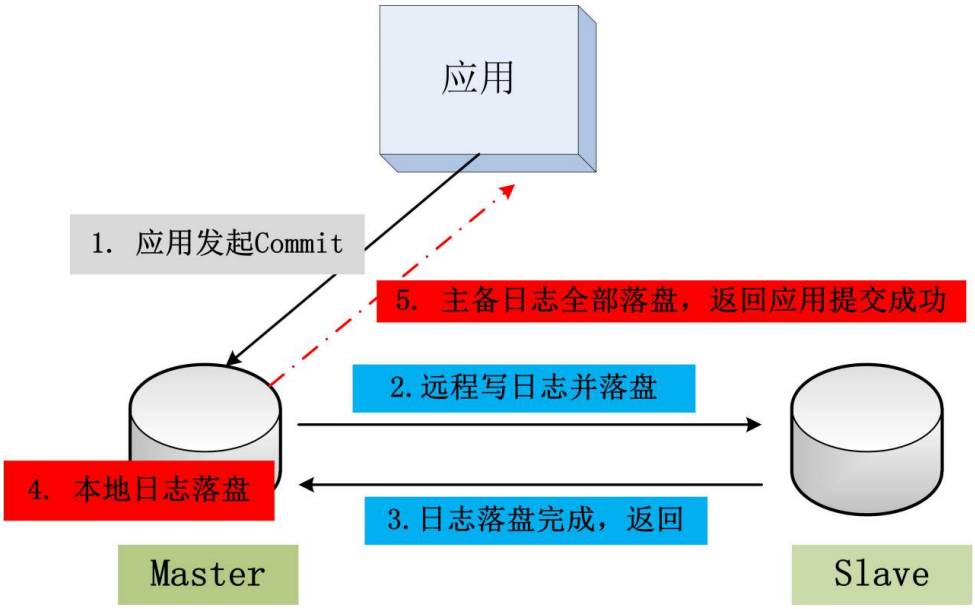
当然，除了这一个简单的三副本优化之外，还可以做其他更多的优化。优化的思路一般就是同步转异步处理，例如事务提交写日志操作；使用更细粒度的锁；关键路径可以采用无锁编程等。

多副本强同步，做到极致，并不一定会导致系统的性能损失。当然，极致应该是什么样子的？我的想法是：

- 对于单个事务来说，RT增加。其响应延时一定会增加（至少多一个网络RT，多一次磁盘Sync）；
- 对整个数据库系统来说，吞吐量不变。远程的网络RT和磁盘Sync并不会消耗本地的CPU资源，本地CPU的开销并未增大。只要是异步化做得好，整个系统的吞吐量，并不会由于引入强同步而降低。

问题四：一个极端场景的分析

意犹未尽，给仍旧在坚持看的朋友预留一个小小的作业。考虑下面这幅图：如果用户的提交操作，在图中的第4步完成前，或者是第4步完成后第5步完成前，主库崩溃。此时，备库有最新的事务提交记录，崩溃的主库，可能有最新的提交记录（第4步完成，第5步前崩溃），也可能没有最新的记录（第4步前崩溃），系统应该如何处理？

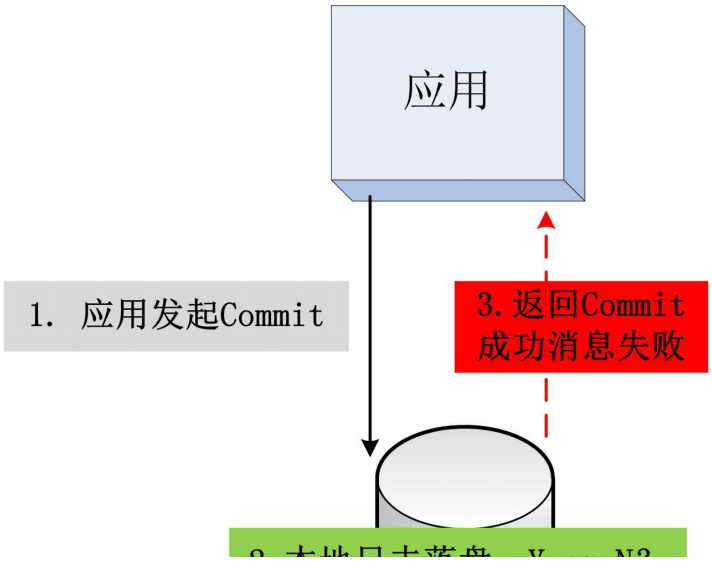


文章在博客上放出来之后，发现大家尤其对这最后一个问题最感兴趣。我选择了一些朋友针对这个问题发表的意见，仅供参考。

@淘宝丁奇

最后那个问题其实本质上跟主备无关。简化一下是，在单库场景下，db本地事务提交完成了，回复ack前crash，或者ack包到达前客户端已经判定超时...所以客户端只要没有收到明确成功或失败，临界事务两种状态都是可以接受的。主备环境下只需要保证系统本身一致。

将丁奇意见用图形化的方式表示出来，就是下面这幅图：



此图，相对于问题四简化了很多，数据库没有主备，只有一个单库。应用发起Commit，在数据库上执行日志落盘操作，但是在返回应用消息时失败（网络原因？超时？）。虽然架构简化了，但是问题大同小异，此时应用并不能判断出本次Commit是成功还是失败，这个状态，需要应用程序的出错处理逻辑处理。

@ArthurHG

最后一个问题，关键是解决服务器端一致性的问题，可以让master从slave同步，也可以让slave回滚，因为客户端没有收到成功消息，所以怎么处理都行。服务器端达成一致后，客户端可以重新提交，为了实现幂等，每个transaction都分配唯一的ID；或者客户端先查询，然后根据结果再决定是否重新提交。

其实，最终的这个问题，更应该由应用的同学来帮助解答：

如果应用程序提交Commit操作后，却Catch到网络或者是超时的异常时，是怎么处理的？希望有同学能够根据实际情况，来进行指导！

在本文的评论部分，成功收集到部分同学对这个问题的点评，现也将他们的点评附在下面，供参考：

@鸿初

如果操作是幂等的，可以继续做直到成功。但如果非幂等，极端要求结果正确的场景下，只能不断去查询，查到后根据结果更新与否做后续处理。

@残魂

一般应用程序肯定判断这次DB操作是失败的，然后用户或者业务系统会发起重试，这个时候会有一个全局的uniqueId来控制幂等性(这个uniqueId一般在业务层面是唯一，可以保证用户发起的(包括失败重试)只被会处理一次)，如果业务系统重试那么uniqueId必然是相同的，这个时候再进行DB操作，commit时，会出现主键冲突或者唯一键异常，应用程序捕获到这种异常，再进行一次查询数据确实存在，那么就可以认为这一次操作是成功的。

@巴布

对应用来说，只要返回值不是SUCCESS，那么都要认为是不可靠的，必须有反查等后续的确认机制，中间的不一致状态如果涉及到消费者感知的信息，必须有披露的途径。