上海语镜汽车信息技术有限公司

多机房数据同步

概要设计

（V 1.0）

修订历史

版本号	修订日期	修订说明	修订人	审核人
Draft 1.0	2015/10/28	o 创建初稿；	刘保安/曹先林	

目录

1. 引言

1.1 编写目的

本概要设计文档，主要是为了明确数据同步组件的实际业务需求，以及各功能模块之间的通信机制及相关规范；以便于为软件的设计研发及后期维护做参考依据。

1.2 预期读者

数据同步组的全体研发人员，项目经理，研发总监及公司高管；

1.3 背景

（待补充）

1.4 定义

序号	缩写	全称	说明
1.			

1.5 参考资料

序号	文档名称	作者	日期/版本

1.6 标准、条约和约定

（待补充）

2. 多机房数据同步介绍

2.1 需求简介

1、支持从任意一个节点数据库 DUMP 原始数据，如果 DUMP 数据失败应有错误信息返回；

2、保证从服务器的入口获取的新 SQL 请求，能够完整的分发到所有节点数据库，并执行，保证数据同步到多节点是**动态**一致的；

3、如果某个节点数据库执行 SQL 失败，应能够将错误信息反馈给服务器，并通过**报警功能**，通知运维人员检查指定数据库；

4、当某个节点从服务器断开连接，应支持自动重连，断点续传；

5、数据同步核心功能可以提取出来，不但可以操作目前的同步 SQL 请求项目，以后也很可能会同步其他业务数据。

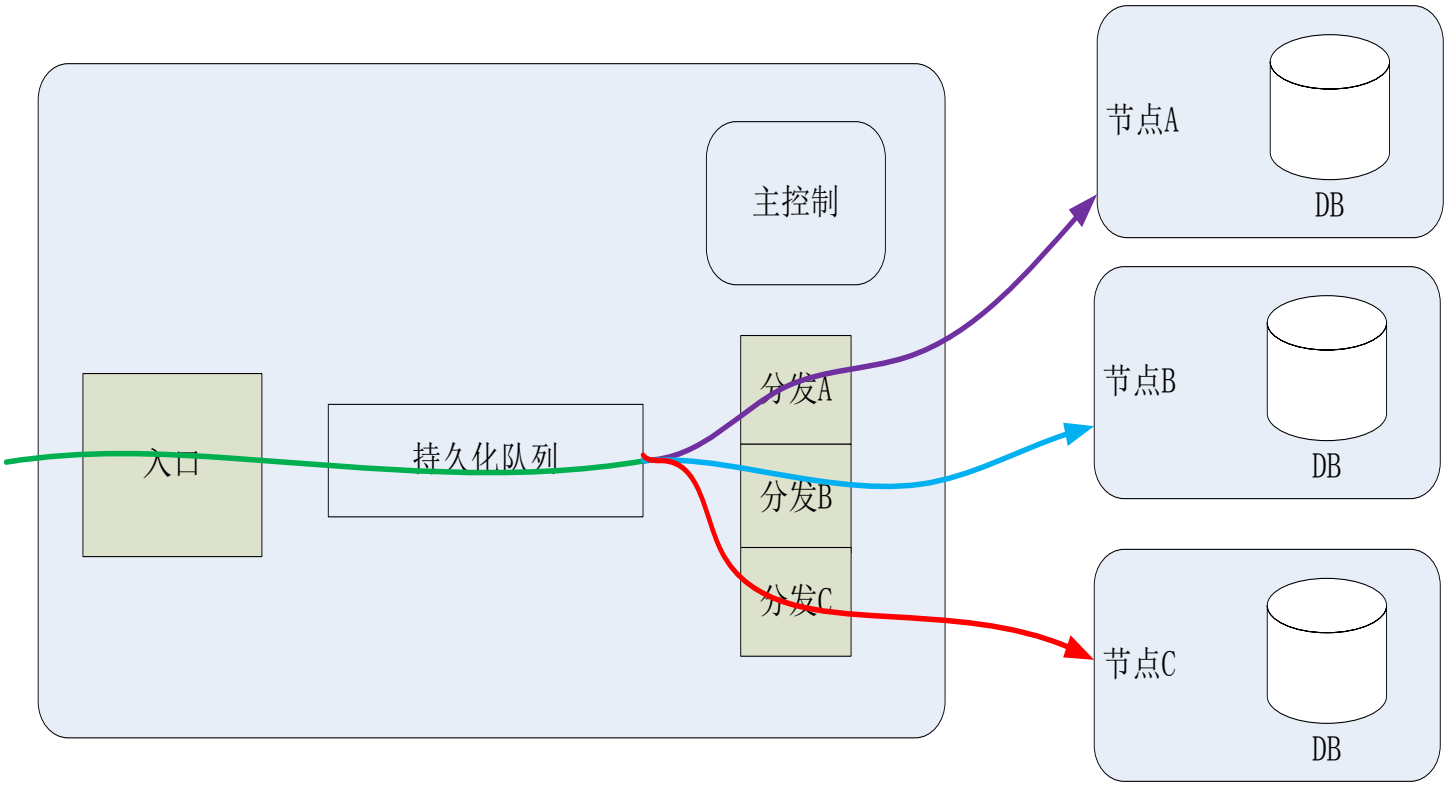
（目标：把**数据存储队列**部分，**通信部分**打包，产生 lib 库文件；同步成功的**数据处理的业务部分**，支持以注册的形式，想做哪方面的业务只需注册个业务功能函数即可；

6、消息队列的回收机制，避免导致内存或磁盘爆掉；

2.2 数据同步的功能实现

功能名称	完成状态	完成进度
1、支持从任意一个节点 DUMP 原始数据；	完成	
2、从服务器入口获取的新 SQL 请求，能够完整分发到所有节点数据库，并执行同步；	完成	
3、服务器对客户端业务执行的返回结果，做不同程度的处理，并通知报警模块；	完成	暂未引入报警模块！
4、节点到服务器支持自动重连；	完成	
5、自定义队列 API，支持 LevelDB、Redis 等	完成	libxmq
6、自定义网络通信 API，底层将来可更换成 TCP/IP 或 ZMQ 等协议；	完成	netmod 模块
7、消息队列的回收机制；	完成	libxmq

2.3 系统整体架构导图

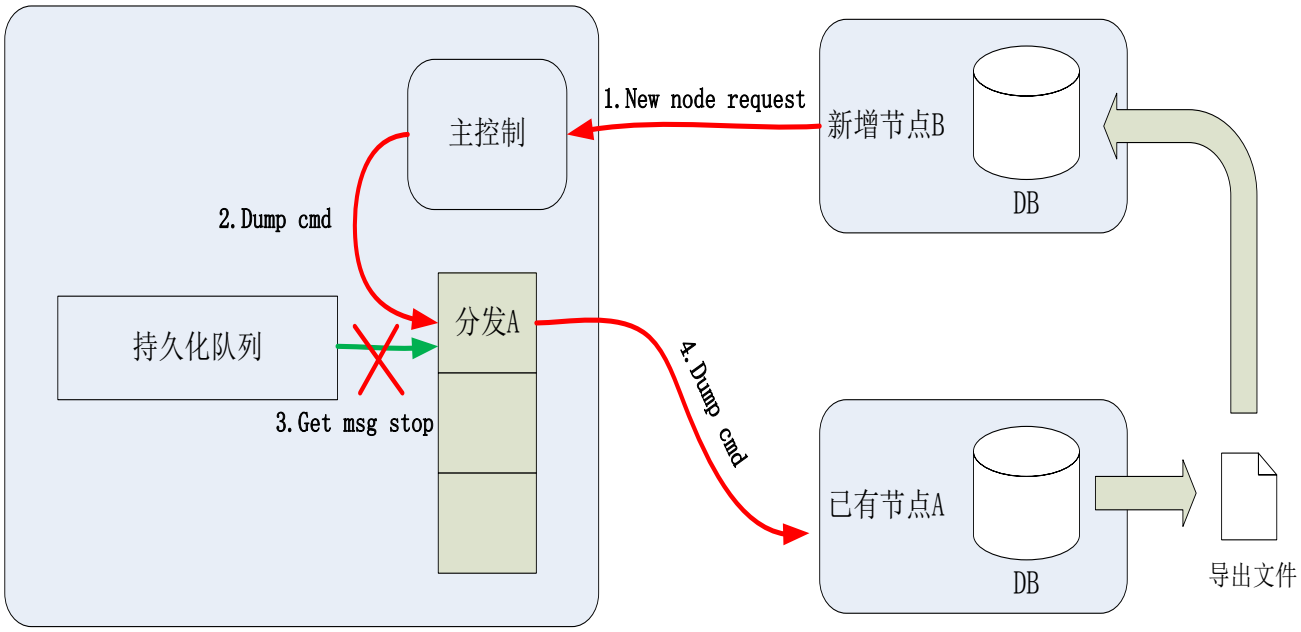


(图 1：整体架构及请求数据流向)

2.4 架构简介

（注：本系统不区分主从节点[只有启动时需要最先从主节点启动,其节点名设定为′ master′]，新增从节点既可以从主节点 DUMP 数据文件，也可以从从节点执行 DUMP 数据的命令）

上图显示的是，所有新增节点，已成功完成从其他节点数据库 DUMP 数据并由运维人员，恢复到自己数据库以后，重启客户端并从服务器队列中获取 SQL 的数据流向过程；



说明:

同步原始数据:

如上图：新增节点 B 建立和服务器（S）的第一次连接，认证通过后，节点 B 向服务器控制中心发出 DUMP 某个节点数据库（节点 A）的命令，由控制中心，转告节点 A 所对应的服务器的分发线程（分发 A），暂时终止从队列中同步 SQL；同时，记录节点 A 已成功同步的任务号(task_seq)，并保存到 B 节点所在服务器的持久化状态信息中。

此时，节点 A 进入 DUMP 数据库的状态，等成功 DUMP 完成后（这里需要通过某种机制说明文件已经成功 DUMP 完整）；

1）节点 A，恢复从服务器队列同步 SQL 的操作；

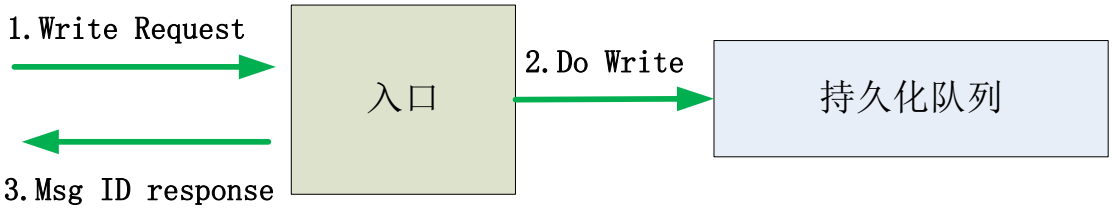
2）运维人员，将该 DUMP 文件通过某种途径（比如：scp 命令）传输到指定的客户端所在的从节点主机(节点 B)，并执行数据库恢复操作，将数据完整恢复到从库；

同步增量数据:

此时，节点 B 建立和服务器的第二次连接，认证通过后，向服务器控制中心发出同步持久化队列 SQL 的命令[EV_INCREMENT]，由于服务器记录了节点 B 的相关状态信息（含 task_seq），当我们再次启动和服务器通信后,就可以依据上次的 task_seq 值从队列读取所有的 SQL，并发送给节点 B，最终完成对队列中增量数据的同步。

3. 核心模块说明

3.1 数据入口



（图 2： 数据入口部分的请求和相应状态）

说明：

接收来自远端用户的 http 请求[目前用户为 dtsync]，获取请求数据，写入持久化队列，并以此数据在队列中的索引值 ID（task_seq）作为响应。

3.2 持久化队列

3.2.1 业务概括

此模块，属数据同步系统框架的核心部分之一，设计目标应满足可以存储各种数据类型，不单单是 SQL 语句；

同时，应该提供一套可操作该队列数据的接口，供“入口数据模块”和“控制中心”及其他模块调用；底层组件可被类似 LevelDB、Redis 等替换（根据实际需求）；

3.2.2 应用场景

持久化队列，在实际运用中，属单生产者、多消费者模式；（即：一个线程往队列写入数据，而其他多个线程并行从队列中读取数据，**不过目前也支持多生产者写入模式**）。

另外，还应该保证：当程序崩溃或断电后重启该程序，能保证数据的一致性(已解决)；

3.2.3 数据结构

目前，持久化队列以 K/V 模式进行数据保存；

K:（key）入队请求数据的序列号；

（从 1 开始，每写入一个请求数据，序列号加 1，为了提高读取效率，K 值都为类似”00000000000000000001”格式，因为一个 uint64_t 最大值为：18446744073709551615UL 长度最大为 20 个字符）；

V:（value）写入的数据（这里强转为 void * 类型写入，读的时候做相应转换；

3.2.3 对外接口

-- 参考 xmq.h xmq_msg.h xmq_csv.h

-- git clone ssh://username@192.168.1.5:29418/supex/

-- ./supex/lib/libxmq/

3.3 控制中心

3.3.1 业务概括

监控客户端发来的请求类型，并根据请求类型做对应操作；同时，记录各个从节点当前的状态信息，并保证持久化！

3.3.2 应用场景

参考：3.3.4 具体描述；

3.3.3 数据结构

/* 储存在服务中心的客户端状态信息 */

1. *客户端 ID(字符串类型)；*
2. *上次查询的索引码；*
3.

注：以服务器保存状态为主，客户端不需要持久化数据；
如果客户端执行 SQL 失败，就返回给服务器错误状态；
通过报警功能，通知运维做相应操作；

3.3.4 具体描述 (以下实现逻辑作为参考)

主控制线程管理所有分发线程，并记录各分发线程的状态信息。

（1） 主控线程对分发线程的管理包括：

a. 创建分发线程。

b. 向分发线程发送 EV_DUMP 命令(向指定从节点发送 DUMP 请求)。

（2） 每个分发线程的状态信息包括：

a. Start_sync_ID：

由于每个节点并不是从队列中第一个消息开始同步。而是先同步一个 BASE，基于该 BASE 在队列中有一个对应的断点。从节点在同步 BASE 后从该断点开始逐个同步队列消息。

b. Current_sync_ID：

每个分发线程逐个读取队列中的消息，并将消息数据发送给对端从节点。

当前同步消息 ID 指的是最后一次读取队列并发送给对端节点的消息的索引号。

c. Current_mark_ID:
每个分发线程发送消息到从节点后，需要收到对端从节点的响应，并将此消息的响应转化为状态值，回写到队列中相应的消息中。一旦收到响应并完成消息状态回写就意味着一条消息的同步完成。当前标记消息 ID 就是记录一个已同步完成的消息在队列中的索引号。

（3） 新从节点的建立

从节点通过命令行参数或配置文件参数的不同，以两种不同的模式运行。在这两种模式下，分别向主节点发送两种请求：

a. 生成 BASE 请求

BASE 是指某个节点当前数据的一个快照或备份。通常在新的从节点第一次启动时发送该请求，请求过程如下图 3。

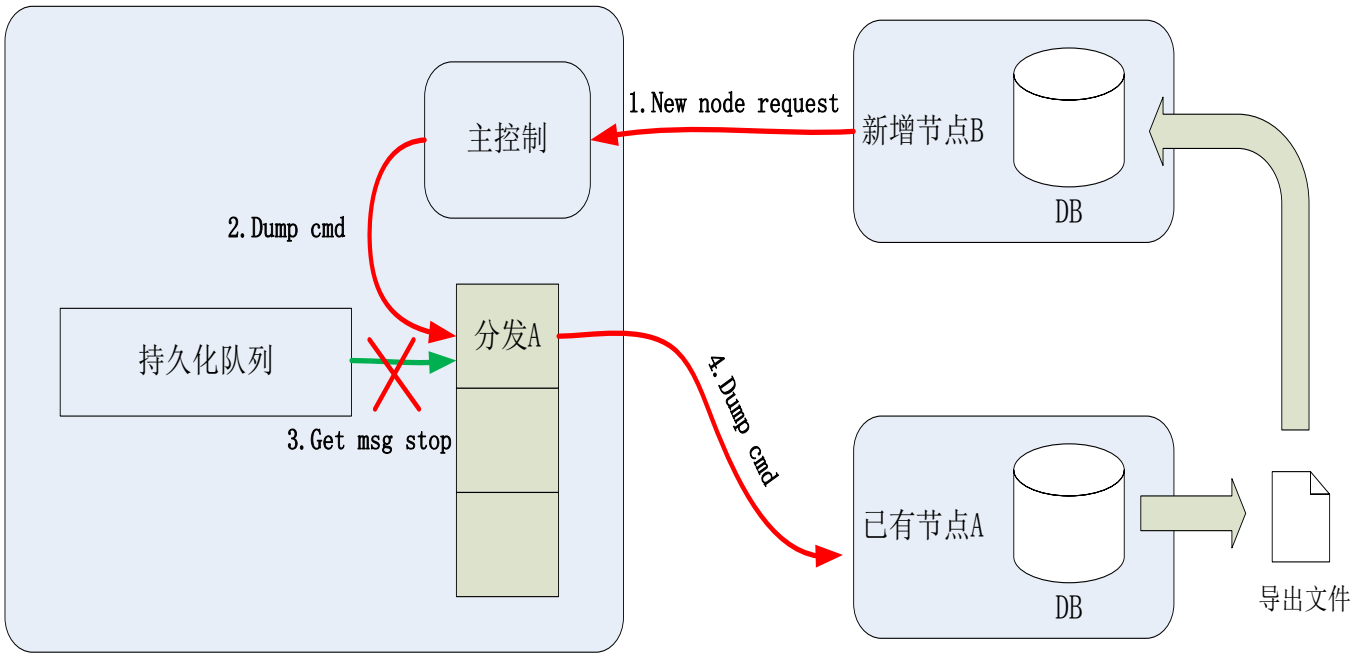
1. 新的从节点 B 发送一个 New Node Request 给主控线程，该请求中包含另一个已有从节点 A 的标识符 Client ID。该 Client ID 指明需要从节点 A 生成 BASE。

2. 主节点的主控线程收到该请求后，获取 Client ID，发现 Client ID 对应的是从节点 A，因此将发送 Dump Base Cmd 给从节点 A 所对应的分发线程 A。

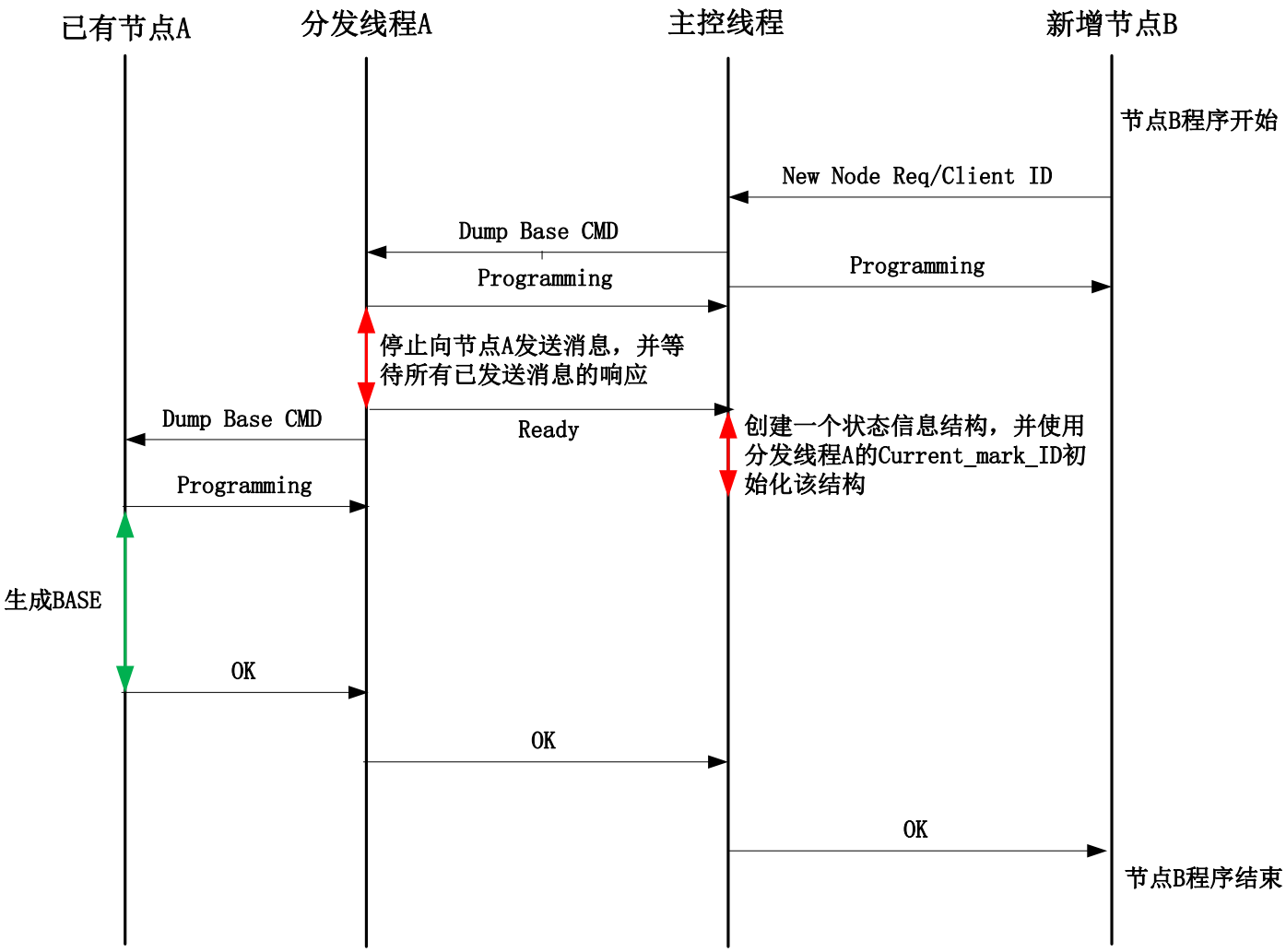
3. 分发线程 A 收到该命令后，停止向从节点发送队列消息，已经发出的消息需要等待该消息的响应，以确保所有发出的消息已经被从节点 A 处理完成。此时主控线程为新增节点 B 创建一个状态信息数据结构，其中 Start_sync_ID、Current_sync_ID、Current_mark_ID 全部初始化为分发线程 A 的 Current_mark_ID。

4. 分发线程 A 现在转发 Dump Base Cmd 命令给对端从节点。

现在，从节点 A 收到所有的消息都已经进入 DB，所有未处理的消息都在主节点队列中。从节点 A 开始生成当前自身数据的一个快照作为 BASE。如果成功，那么逐跳响应成功。消息时序图如图 4 所示。



(图 3: 新增节点 B 第一次连接，发出 DUMP(已有节点 A)数据库命令请求)

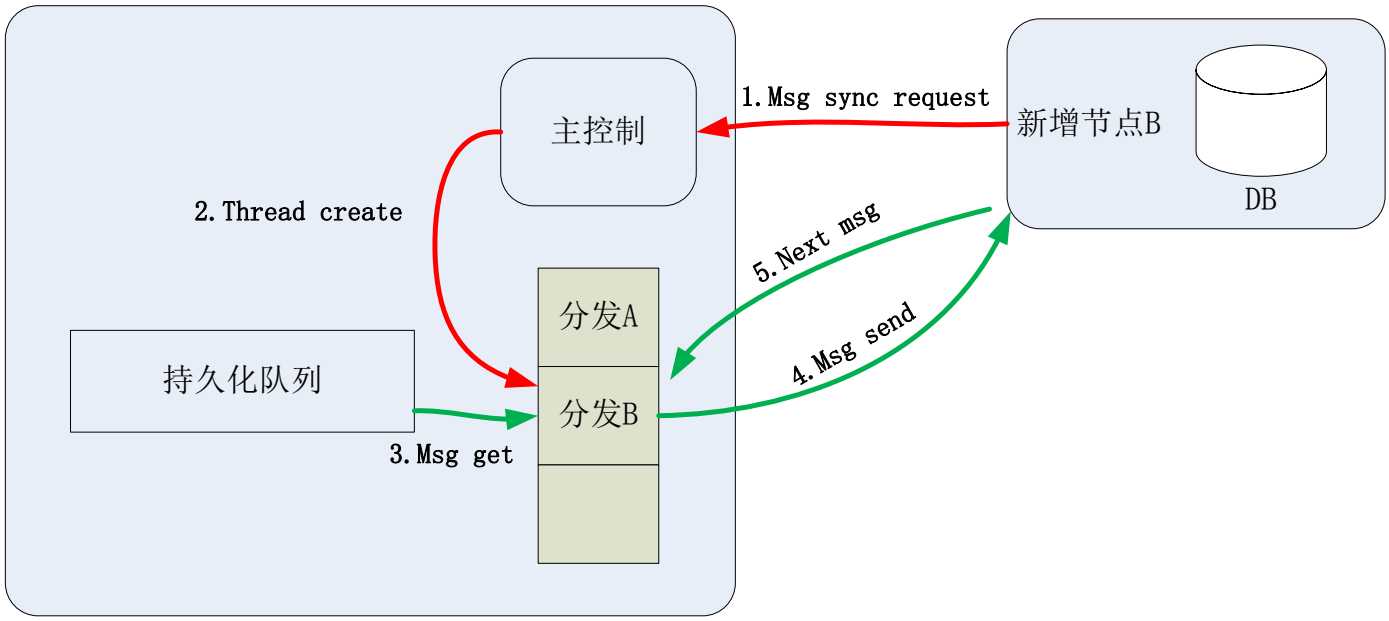


(图 4: 具体请求上来后的具体交互流程导图)

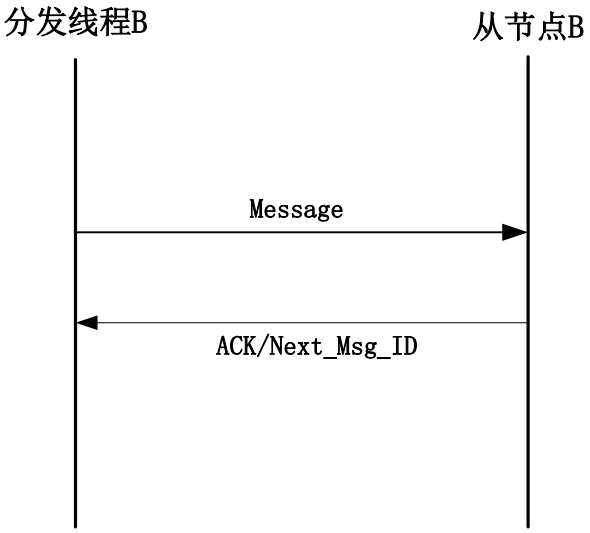
b. **数据同步请求**

当上述 A 节点成功生成 BASE 后，将 BASE 同步到此新增节点。此时再次启动新增节点 B 程序就可以向主节点的主控线程发送消息同步请求。见下图 5 所示。

1. 新增节点 B 再次启动，并且向主控节点发送 Msg Sync Request。
2. 主节点为从节点 B 创建一个分发线程 B，专门用于获取队列中的消息发送给从节点 B。
3. 由于在上述“生成 BASE 请求处理”处理过程中第 3 步骤主节点为此新增 B 节点已经创建了一个状态信息结构并初始化。分发线程 B 此时知道该从队列中具体哪个位置开始同步消息。
4. 此时新增节点 B 动态加入过程便成功完成。分发线程 B 将不断从队列中获取的消息发给节点 B，并等待对端的响应。
5. 节点 B 发送响应及对下一个消息的请求给分发线程 B。重复第 4 步骤。



(图 5: 新增节点 B 第二次连接，发出同步队列消息数据请求)



(图 6: 消息时序图)

（4） 分发线程：

c.分发线程从持久化队列里获取消息,发送给对应从节点;

d.分发线程收到主控线程的 Dump Base Cmd,转发给对应从节点;

3.4 通信交互

3.4.1 业务概括

此模块主要解决客户端和服务器通信相关的功能，包括自定义上层通信协议结构类型及接口；将来底层通信部分可能替换成其他通信协议，比如：TCP/IP、ZeroMQ 及公司自己的 MFTCP 协议；

3.4.2 应用场景

通信交互层的设计，应该满足服务器可以监听客户端的请求，并创建独立子线程，单独跟客户端交互的能力。

数据同步项目中，所有的客户端状态信息都应该存储在服务器端，因为，服务器一般比较稳定，而且不会随意挪动或升级等等，不会随意删除持久化数据文件。而我们不能保证从节点所在的客户端这些潜在问题;

另外，从节点不会太多，创建独立子线程与之通信，这样不同的从节点之间没有任何交互的情况，互不影响，同时也避免了，因为一个从节点 DUMP 数据库文件的操作，而阻塞了所有的其他从节点获取 SQL 队列请求的操作;

而且，从结构上来划分也更清晰;

3.4.3 通信数据结构定义

#define IDENTITY_SIZE 32

节点类型	类型值	定义
SOCK_SERVER	1	标识当前为服务器
SOCK_CLIENT	2	标识当前为客户端

事件类型	类型值	定义
EV_DUMP_REQ	1	请求 DUMP 数据库
EV_DUMP_REP	2	DUMP 数据库响应
EV_INCREMENT_REQ	3	请求增量数据
EV_INCREMENT_REP	4	增量数据响应

返回状态类型	类型值	定义
EV_NONE	-1	表示当前事件为初始化状态;该值由客户端设定,并由服务器解析; 从哪里开始同步,由服务器决定; 对于目前的通信模块分两种情况: 1)请求 DUMP 命令时(ev_state 设置为 EV_NONE); 2)请求完 DUMP 命令并成功同步完成,下次开始同步增量数据时(ev_state 也设置为 EV_NONE); 3)客户端每次重启时,ev_state 都会被设置为 EV_NONE;
EV_SUCC	0	表示客户业务执行成功;
EV_FAIL	1	代表: 一般性错误 服务器收到此返回状态时: 1) **不清除**客户端状态信息; 2) 将该错误告知报警模块,进行人工干预; 3) 客户端停止同步请求,并终止进程; 4) 当人工干预完成后,客户端重启,并**从上次执行失败的下一条数据开始同步**;(此时,传递给服务器的状态码为 EV_SUCC,任务序列号为 0,由服务器决定,因为客户端没有持久化状态信息);
EV_FATAL	2	代表: 严重性错误 服务器收到此返回状态时: 1) **清除**客户端状态信息; 2) 将该严重错误告知报警模块,进行人工干预; 3) 客户端停止同步请求,并终止进程; 4) 当人工干预完成后,**客户端完全重新同步**(BASE+增量同步);

```c
/* Increment SQL request. */
struct incr_data {
    uint64_t task_seq;        /* The sequence of increment SQL request. */
    int rows;          /* The expect rows of increment data to be send,
                          or actual rows of data that client received. */
};

typedef struct {
    char id[IDENTITY_SIZE];                 /* Client unique identity. */
    int  ev_type;                        /* Event type, EV_XXX_(REQ|REP) */
    int  ev_state;                          /* Last event excute state. */
    union
    {
        /* Increment SQL request. */
        struct incr_data incr;
    };
    size_t ev_size;  /* The size of ev_data. if ev_data is null(empty),
                    then ev_size must be 0; if ev_data store char* type,
        then ev_size should be strlen(ev_data)+1 for '\0', by yourself. */

    char ev_data[0];                       /* Data to be send to client. */
}event_t;

/* Network communication global context */
typedef struct {
    void *zmq_ctx;                              /* ZeroMQ type's Context. */
    void *zmq_socket;                           /* ZeroMQ type's Socket. */
    int  node_type;                 /* End point type, Server or Client. */
    //pthread_key_t key_r;/* This is used for recv_event() free memroy. */
}event_ctx_t;

/* The event_t package head size. */
#define event_head_size()    (sizeof(event_t))

/* The event_t package body size. ev is (event_t *) type. */
#define event_body_size(ev) ((ev)->ev_size)

/* The event_t package all size. ev is (event_t *) type. */
#define event_all_size(ev)  (event_head_size() + event_body_size(ev))
```

具体参考 netmod.h 文件定义;

3.4.3 对外 API 接口

```c
event_ctx_t *event_ctx_init(int *error, int node_type, const char *uri_addr, const char *identity);
```

功能说明:

初始化通信模块;

参数说明:

参数名	参数类型	参数说明	返回值
int *error	入参|出参	返回当前函数的错误码,可以调用 const char *event_error(int error); 返回错误信息;	**如果执行成功,返回:** event_ctx_t*通信模块的上下文结构指针,在其他函数中被使用;
int node_type	入参	SOCK_SERVER: 初始化为服务器, SOCK_CLIENT: 初始化为客户端;	
const char *uri_addr	入参	ZeroMQ 类型的网络连接字符串; 如果 node_type is SOCK_SERVER, uri_addr 将类似: "tcp://*:8686" 属 bind() 如果 node_type is SOCK_CLIENT, uri_addr 将类似:	

		“tcp://192.168.11.27:8686” 属 connect()	**如果执行失败,返回:** NULL.
const char *identity	入参	如果 node_type=SOCK_SERVER, identity 设置为**空字符串** 如果 node_type=SOCK_CLIENT, identity 设置为**非空字符串**,该值是客户端的唯一标识;	

```
int event_ctx_destroy(event_ctx_t *ev_ctx);
```

功能说明:

销毁通信模块;

参数说明:

参数名	参数类型	参数说明	返回值
event_ctx_t *ev_ctx	入参	通信模块的上下文,由 event_ctx_init()返回;	**执行成功,返回:** 0, **失败返回:** 错误码;

```
event_t *recv_event(int *error, const event_ctx_t *ev_ctx, long timeout);
```

功能说明:

如果当前是服务器，则接受的是客户端发出的请求(EV_XXX_REQ);

如果当前是客户端，则接受的是服务器发出的响应(EV_XXX_REP);

参数说明:

参数名	参数类型	参数说明	返回值
int *error	入参|出参	返回当前函数的错误码,可以调用 const char *event_error(int error); 返回错误信息;	**如果执行成功返回:** 当前事件消息的 event_t 指针类型, **如果失败返回:** NULL,并将 *error 设置为 errno.
const event_t *ev_ctx	入参	通信模块上下文指针	
long timeout	入参	如果: timeout= -1;则一直阻塞,直到接收到新消息,返回; 如果: timeout = 0;则不管有没有新消息,都立马返回; 如果: timeout > 0;则如果有消息,就立马返回消息,如果没有新消息就一直等待,直到 timeout 毫秒,超时,返回;	

特殊说明:

1)调用者不要释放返回的 event_t 类型的指针,否则会报 double free 类型的错误;

2)如果超时的话,函数会返回 NULL,并且*error 会被设置为0(代表超时返回成功),所以在判断时一定要结合返回值和*error 的值一起判断: 比如:

```
event_t *ev_res = recv_event(&error, ev_ctx, 3000);
if (ev_res == NULL) {
    if (error != 0) { // 超时的情况,不必打印错误消息!
        printf(“recv_event() fail. Error - %s.\n”, event_error(error));
    }
    // 错误处理, return or 其他处理.
}
```

```
int send_event(const event_ctx_t *ev_ctx, const event_t *event);
```

功能说明:

发送消息事件;

参数说明:

参数名	参数类型	参数说明	返回值
event_ctx_t * ev_ctx	入参	通信模块的上下文,由 event_ctx_init()返回;	执行成功,返回:0, 失败,返回错误码;
const event_t *event	入参	待发送的事件消息;这个由调用者传入,类型基本如下: event = (event_t *)malloc(sizeof(event_t) + ev_size); ev_size 是待发送的消息事件包体(ev_data)的大小; 如果包体为空,则 ev_size 必须设置为 0; 如果包体是字符串类型,则 ev_size=strlen(ev_data)+1; 调用者必须注意使用规则;	

```
const char *event_error(int error);
```

功能说明:

获取通信模块的其他函数，执行失败返回的错误消息;

参数说明:

参数名	参数类型	参数说明	返回值
int error	入参	由其他函数执行失败返回的错误码;	执行成功,返回:错误消息;

3.4.3 通信协议说明

具体如下: (请仔细阅读全文)

一、 网络通信数据传输完整性考虑(已通过测试)
1、ZMQ 完全支持重连 所以不用考虑发送信息失败的情况; (大不了就阻塞或超时等待接收);
2、ZMQ 保证每次发送数据的完整性; 要么成功要么失败,不会出现只发送或接收一个数据包的一部分的情况;
3、ZMQ 底层重连有时间增加; 比如:第一次会立马重连;如果失败,第二次重连会间隔大概 3-4s,第三次,可能 8s ...

所以，以下通信协议建立在网络可正常连接通信的情况下;

返回状态类型	类型值	定义
EV_NONE	-1	表示当前事件为初始化状态;该值由客户端设定,并由服务器解析; (从哪里开始同步,由服务器决定) 对于目前的通信模块分两种情况: 1)请求 DUMP 命令时(ev_state 设置为 EV_NONE); 2)请求完 DUMP 命令并成功同步完成,下次开始同步增量数据时(ev_state 也设置为 EV_NONE) 3)客户端每次重启时,ev_state 都会被设置为 EV_NONE;
EV_SUCC	0	表示客户业务执行成功;

EV_FAIL	1	代表:(一般性错误)
		注意:(当从节点所在客户端业务执行失败)
		如果是增量同步,那么当请求方业务执行失败时,客户端会退出;
		如果是 BASE 同步,那么当执行 DUMP 操作的客户端(M)执行失败时,M 进程不会退出,
		而发出请求 DUMP 的客户端(B)接收到 EV_FAIL 时,B 进程会退出;
		==========具体如下==========
		当服务器收到此返回状态时:
		1)不清除客户端状态信息;
		2)将该错误告知报警模块,进行人工干预;
		3)客户端停止同步请求,并终止进程;
		4)当人工干预完成后,客户端重启,并从上次执行失败的下一条数据开始同步;
		(此时,传递给服务器的状态码为 EV_NONE,任务序列号为 0,由服务器决定,
		因为客户端没有持久化状态信息);
		==========[执行增量数据同步失败的情况如下]==========
		通信流程:[B->S->B]
		字符含义: B:"发起请求 DUMP 操作的节点,假如是:nodeb"
		S:"服务中心"
		==========[业务执行失败返回阶段]==========
		结构如下:(发送完成后,客户端进程退出)
		event_t {
		id="nodeb"; // 客户端自己的 ID
		ev_type = EV_INCREMENT_REQ;
		ev_state = EV_FAIL;
		incr.task_seq = 当前执行失败的数据包序列号;
		incr.rows = 1;
		ev_size = strlen(ev_data)+1;
		ev_data = "业务失败错误信息"; // 告知报警模块,进行人工干预;
		};
		==========[人工干预后重启阶段]==========
		服务器会收到如下结构:
		event_t {
		id="nodeb"; // 客户端自己的 ID
		ev_type = EV_INCREMENT_REQ;
		ev_state = EV_NONE;
		ev_size = 0;

		incr.task_seq = 0;
		incr.rows = 1;
		//因为客户端没有持久化保持状态信息,所以,得由服务器决定从下一条开始同步;
		};
		==========[执行原始数据同步失败的情况如下]==========
		通信流程:[B->S->M->S->B]
		字符含义: B:"发起请求 DUMP 操作的节点,假如是:nodeb"
		M:"执行 DUMP 操作的节点,假如是:master"
		S:"服务中心"
		==========[以下为返回阶段]==========
		M->S 阶段:
		event_t {
		id="master"; // 执行 DUMP 操作的客户端的 ID;
		ev_type = EV_DUMP_REP;
		ev_state = EV_FAIL; // 注意: DUMP 操作永远不会反悔 EV_FATAL;
		ev_size = strlen(ev_data)+1;
		ev_data = "执行 DUMP 操作的失败信息"; //错误信息,告知报警模块;进行人工干预
		};
		S->B 阶段:(客户端 B 收到反馈信息后不管成功或失败都会终止进程)
		event_t {
		id="nodeb"; // DUMP 请求的客户端的 ID;
		ev_type = EV_DUMP_REP;
		ev_state = EV_FAIL; // 注意: DUMP 操作永远不会反悔 EV_FATAL;
		ev_size = 0; // 客户端 B 不需要知道错误信息;收到此 EV_FAIL 就退出;
		};
		==========[人工干预后重启阶段]==========
		服务器会收到,节点 B 再次发出的 DUMP 请求;(结构如下)
		event_t {
		id="nodeb"; // DUMP 请求的客户端的 ID;
		ev_type = EV_DUMP_REQ;
		ev_state = EV_NONE;
		ev_size = strlen(ev_data)+1;
		ev_data = "master"; // 目标执行 DUMP 操作的节点 ID;
		};

EV_FATAL	2	代表: (严重性错误) 当服务器收到此返回状态时: 1)清除客户端状态信息; 2)将该严重错误告知报警模块,进行人工干预; 3)客户端停止同步请求,并终止进程; 4)当人工干预完成后,客户端完全重新同步(BASE+增量同步); 目前主要用于增量同步; ==========[执行增量数据同步失败的情况如下]==========

		通信流程: [B->S->B]
		字符含义: B:"发起请求 DUMP 操作的节点,假如是:nodeb"
		S:"服务中心"
		==========[业务执行失败返回阶段]==========
		结构如下:(发送完成后,客户端进程退出)
		event_t {
		id="nodeb"; // 客户端自己的 ID
		ev_type = EV_INCREMENT_REQ;
		ev_state = EV_FATAL;
		ev_size = strlen(ev_data)+1;
		ev_data = "业务失败严重错误信息"; // 告知报警模块,进行人工干预;
		};
		==========[人工干预后重启阶段]==========
		服务器会收到如下结构:
		event_t {
		id="nodeb"; // 客户端自己的 ID
		ev_type = EV_DUMP_REQ;
		ev_state = EV_NONE;
		ev_size = strlen(ev_data)+1;
		ev_data = "master"; // 目标执行 DUMP 操作的节点 ID;
		};
		DUMP 执行成功后,接着请求增量同步;
		event_t {
		id="nodeb"; // 客户端自己的 ID
		ev_type = EV_INCREMENT_REQ;
		ev_state = EV_NONE; // 由服务器决定从哪个位置开始同步;
		ev_size = 0;
		incr.task_seq = 0;
		incr.rows = 1;
		};

三、协议之数据包请求及相应状态解析:	
功能描述: 1、 源节点(B, id="NodeB")向服务器发出请求,从目的节点(A, id="NodeA")DUMP 数据库的原始数据; (EV_DUMP_[REQ|REP]) 2、 源节点 B 向服务器发出请求增量数据; (EV_INCREMENT_[REQ|REP]) **通信过程:** EV_DUMP_XX: B->服务器->A-服务器->B EV_INCREMNT_XX: B->服务器->B **过程详解:**	
EV_DUMP_REQ	**【B->服务器】: 源节点 B 向服务器发送 EV_DUMP_REQ 的命令,携带目的节点的 ID(“NodeA”), event_t 通信包, 定义如下:** event_t {

	id="NodeB"; // 源节点所在客户端的 ID;
	ev_type=EV_DUMP_REQ; // 发起 DUMP 请求.
	ev_state=EV_NONE; /* 第一次,所以状态设置为 EV_NONE(表示上次没有执行)*/
	ev_size=6; // 携带数据的长度;
	ev_data="NodeA"; // 携带的目的节点所在客户端的 ID;
	}
	【服务器->A】: 服务器解析后,像 A 发起 EV_DUMP_REQ 命令, event_t 通信包, 定义如下:
	event_t {
	id="NodeA"; // 目的节点所在客户端的 ID;
	ev_type=EV_DUMP_REQ; // 发起 DUMP 请求.
	ev_state=EV_NONE; /* 第一次,所以状态设置为 EV_NONE(表示上次没有执行)*/
	ev_size=0; //注: 如果 ev_data 没有数据,ev_size 必须设置为 0.
	}

EV_DUMP_REP	**【A->服务器】: 目的节点 A 向服务器发送 EV_DUMP_REP 命令,执行状态信息可要可不要(服务器不受影响), event_t 通信包, 定义如下:**
	event_t {
	id="NodeA"; // 目的节点所在客户端的 ID;
	ev_type=EV_DUMP_REP; // 发起 DUMP 响应;
	ev_state=EV_SUCC; // 或者 EV_FAIL|EV_WARN.
	ev_size=strlen(ev_data)+1; // 注: 如果 ev_data 没有数据,ev_size 必须设置为 0.
	ev_data="失败的信息"; // 如果 DUMP 失败,则返回错误信息;通知报警模块;
	}
	
	【服务器->B】: 服务器将数据包转发给源节点 B, event_t 通信包, 定义如下:
	event_t {
	id="NodeB"; //源节点所在客户端的 ID;
	ev_type=EV_DUMP_REP; // 发起 DUMP 请求.
	ev_state=EV_SUCC; // 或者 EV_FAIL|EV_WARN 节点 B 检测到错误,就终止进程.
	ev_size=0;
	}

EV_INCREMENT_REQ	**【B->服务器】: 源节点 B 向服务器发送 EV_INCREMENT_REQ 的命令 , event_t 通信包, 定义如下:**
	event_t {
	id="NodeB"; //源节点所在客户端的 ID;
	ev_type=EV_INCREMENT_REQ; // 发起获取增量数据的请求.
	ev_state=EV_NONE; /* 第一次设置为 EV_NONE. */

	incr.task_seq= 0; /* 第一次设置为 0,由服务器根据 B 之前发出 DUMP 命令时,修改的 task_seq 值更新该值,并发送给 B.*/
	ev_size=0; // 携带数据的长度,ev_data 为空时,必须置 0;
	}

EV_INCREMENT_REP	**【服务器->B】: 服务器解析,处理后,像 B 发起 EV_DUMP_REP 命令,并携带相关数据(由 ev_data 保存), event_t 通信包, 定义如下:**
	event_t {
	id="NodeB"; // 源节点所在客户端的 ID;
	ev_type=EV_INCREMENT_REP; // 发出获取增量数据的响应.
	ev_state=EV_NONE;
	/* 同步增量数据时,由服务器发出的 event 包,ev_state 都置为 EV_NONE.因为他代表,节点(B|C|A|…)所在客户端业务处理返回状态. */
	incr.task_seq= B 在服务器的 task_seq 值;
	/* 该值由服务器更新,分 2 种情况:
	1)由 B 向服务器发出 DUMP A 节点数据库的原始数据时,时对应的 A 的 task_seq 值;
	2)B 在同步服务器增量数据时,每次更新的 task_seq 值;
	*/
	ev_size=strlen(ev_data)+1; // 携带数据的长度
	ev_data=" 具体的增量 SQL 语句"
	}
	
	【B 收到增量数据后,做业务处理,并返回执行状态,成功或失败(含错误信息)】
	
	【B->服务器】: B 执行完业务后,像服务器发送 EV_INCREMENT_REQ 命令), event_t 通信包, 定义如下:
	event_t {
	id="NodeB"; //源节点所在客户端的 ID;
	ev_type=EV_INCREMENT_REQ; // 发起获取增量数据的请求.
	ev_state=EV_SUCC; // EV_FAIL|EV_WARN,根据业务执行状态设置.
	incr.task_seq= 上次获取的 task_seq 值加 1;
	ev_size=0; // 携带数据的长度,ev_data 为空时,必须置 0;
	}
	
	注: 如果上次执行失败,event_t 通信包,定义如下:
	event_t {
	id="NodeB"; //源节点所在客户端的 ID;
	ev_type=EV_INCREMENT_REQ; // 发起获取增量数据的请求.
	ev_state=EV_FAIL; // EV_FAIL|EV_WARN,根据业务执行状态设置.
	incr.task_seq= 上次获取的 task_seq 值;
	ev_size=strlen(ev_data)+1; // 携带数据的长度;

	ev_data=”业务执行失败的错误信息”； }

3.5 业务处理(客户端)

3.3.1 业务概括

数据同步的核心功能被重复利用的可能性很大，可以同步 SQL 请求，当然也可以同步其他的业务需求；

所以，当客户端获取到数据后，该如何处理，这个就涉及到业务处理方面了，这个得做成通用接口；把实际业务逻辑以注册的形式加载进来，不同的业务处理，只需要注册不同的功能函数即可；

3.3.2 操作函数:

参考 supex/programs/pole-S/src/register.h

主要实现如下三个函数即可；

```
struct business_t
{
        /* sync_conf_t: was defined in parser.h */
        int     (*init)(const sync_conf_t *conf);
        int     (*done)(char *error, size_t err_size, void *args, size_t arg_size);
        void    (*destroy)();
};
typedef struct business_t business_t;
```

具体调用入下图:

```
/* Register the business functions. */
business_t busi_dump = { dump_init, dump_db, NULL };

// busi_incr you can modify it's elements with your own.
// But busi_dump, you should never modify it.
business_t busi_incr = {
        w2file_init,
        w2file_done,
        w2file_destroy
};

register_business(BUSI_DUMP, &busi_dump);
register_business(BUSI_INCR, &busi_incr);

/* Startup all the businesses. */
res = startup_businesses(&confs);
assert(res == 0);
```

注: busi_incr { 内部三个函数指针,可以更新为插入数据库,写入 Redis 或其他! 这 里是写入到文件}

4. 配置文件说明

参考: 多机房数据同步线上部署步骤.pdf

5. 技术难点

6. 会议总结

7. 附录