



基于Golang的分布式数据运营与流式计算实战

×



刘丹冰Aceld

好未来
服务端高级专家

“

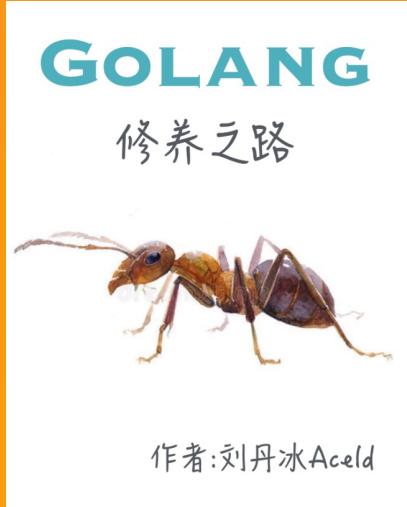
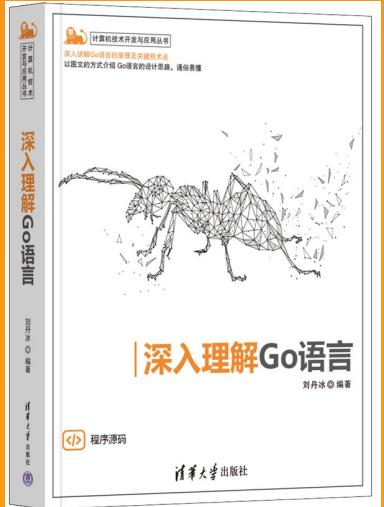


刘丹冰 Aceld

Github: <https://github.com/aceld>

Go博客: <https://www.yuque.com/aceld>

B站: <https://space.bilibili.com/373073810>



> 开源框架：Zinx 作者

> 出版图书《深入理解Go语言》作者

> 代表网络作品：

《8小时转职Golang工程师》
《Golang修养之路》





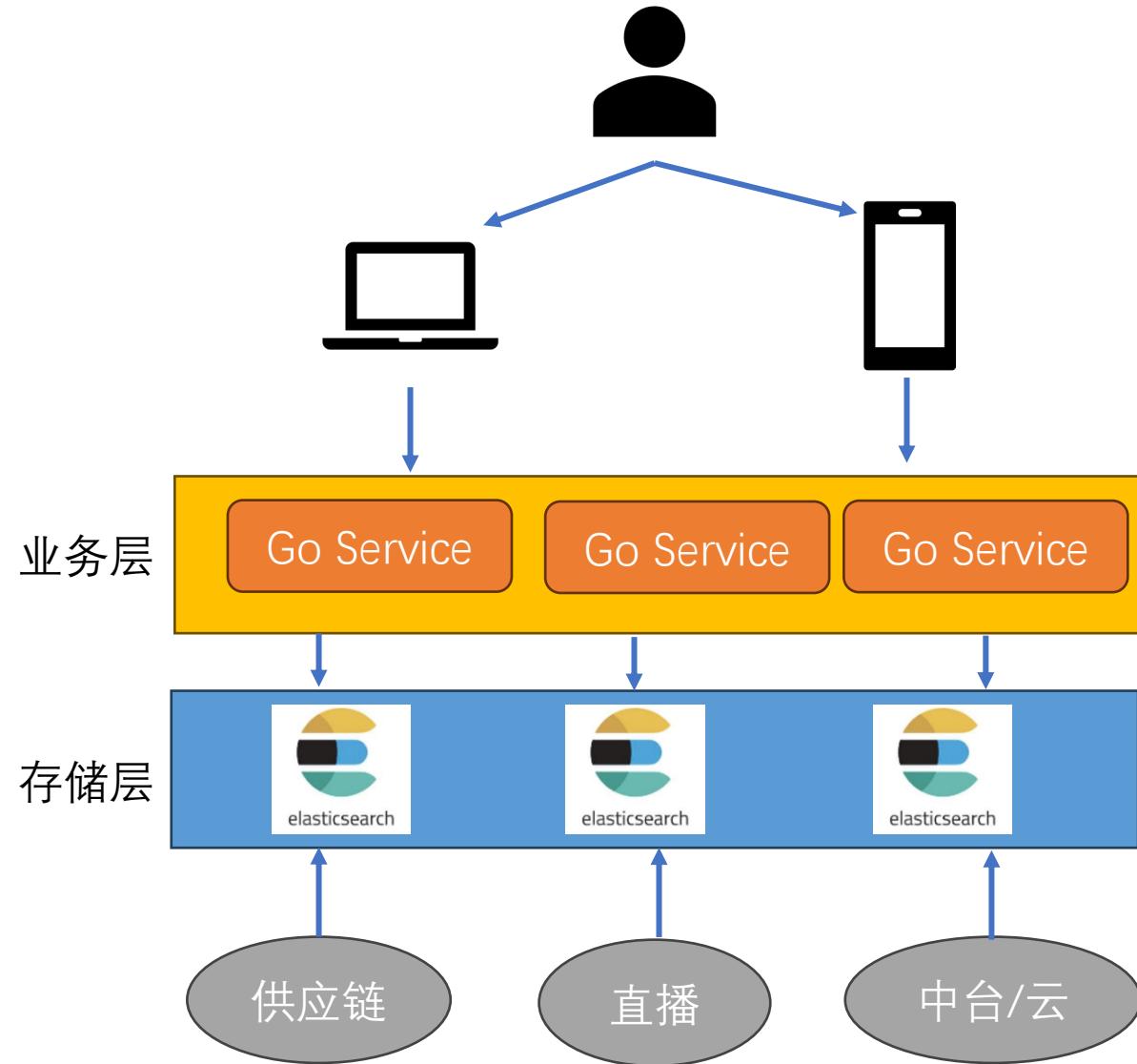
目 录

- ToB系统的业务层数据处理痛点 01
- Arris (业务层数据运营中心) 02
- Arris数据实时修复架构设计 03
- 流式计算框架NsFlow 04

第一部分

ToB系统的业务层数 据处理痛点

ToB 的智能教辅系统的业务数据自治问题



- 问题1: 数据不可控

- 数据对业务方是黑盒、没有监控, 报警, 业务方报出来, 才修复, 业务伤害大

- 问题2: 人力不足

- 迭代迅速, 不满足业务需求

- 问题3: 事故

- 2019~2020, ES数据集群宕机3次(连着3次续报都产生了宕机), 影响辅导老师正常工作

业务挑战

- 任务重: 需要快速实现数据的切换

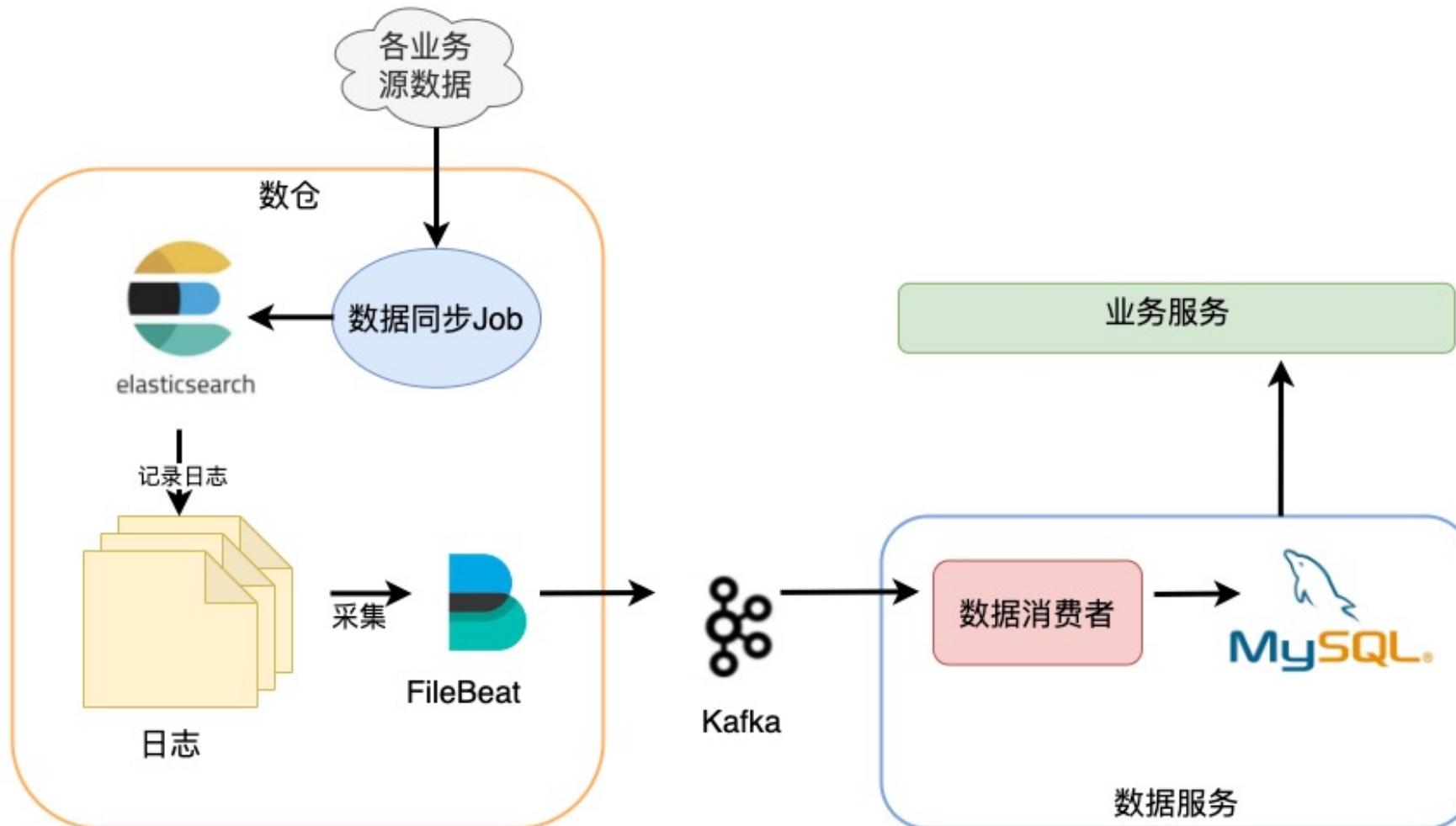
技术挑战

- 索引多: 将近20张索引
- 一致性: 如何保证数据一致性
- 数据量大: 怎么存?

团队挑战

- 上游业务没有时间配合我们做数据质量

数据自治的临时解决方案



快：数据快速先接进来

慢：问题可以在接入过程优化

面临的问题

数据不准



数据结构问题

```
    "x_level": "error", "x_msg": "EsSinglePath
rser unmarshal err; params[{'status': 1, 'msg': [0, []],
'data': [{}], 'month_start_rate': 100, 'month_end_rate': 100, 'month_
finish_rate': 100, 'month_low_rate': 100}], 'modify_time': 1615200000000,
'modify_by': 'root', 'month_start': 1, 'month_end': 1, 'month_low': 1, 'month_
high': 1, 'month_start_low': 1, 'month_end_low': 1, 'month_low_low': 1, 'month_
high_low': 1}, 'update_time': 1615200000000, 'update_by': 'root'}, 'readObjectStart: expect { or n, but found [, error
found in #10 byte of ...], 'msg': [0, []], 'data': [{}], 'month_start_rate': 100, 'month_end_rate': 100, 'month_
finish_rate': 100, 'month_low_rate': 100}], 'modify_time': 1615200000000,
'modify_by': 'root', 'month_start': 1, 'month_end': 1, 'month_low': 1, 'month_
high': 1, 'month_start_low': 1, 'month_end_low': 1, 'month_low_low': 1, 'month_
high_low': 1}, 'update_time': 1615200000000, 'update_by': 'root'} }
```

类型错误

告警原因: {"alarmLog":"basiclog-
trace_19_8
func:PlansS
remotehos
logid:1241
error:entity.ClassPlanStudyStatistics.AllStuNum:
readUInt64: unexpected character: ♦, error found in
#10 byte of ...|stu_num': '0', 'book_n|..., bigger context
...|
{'all_st
comm|

数据格式不统一问题

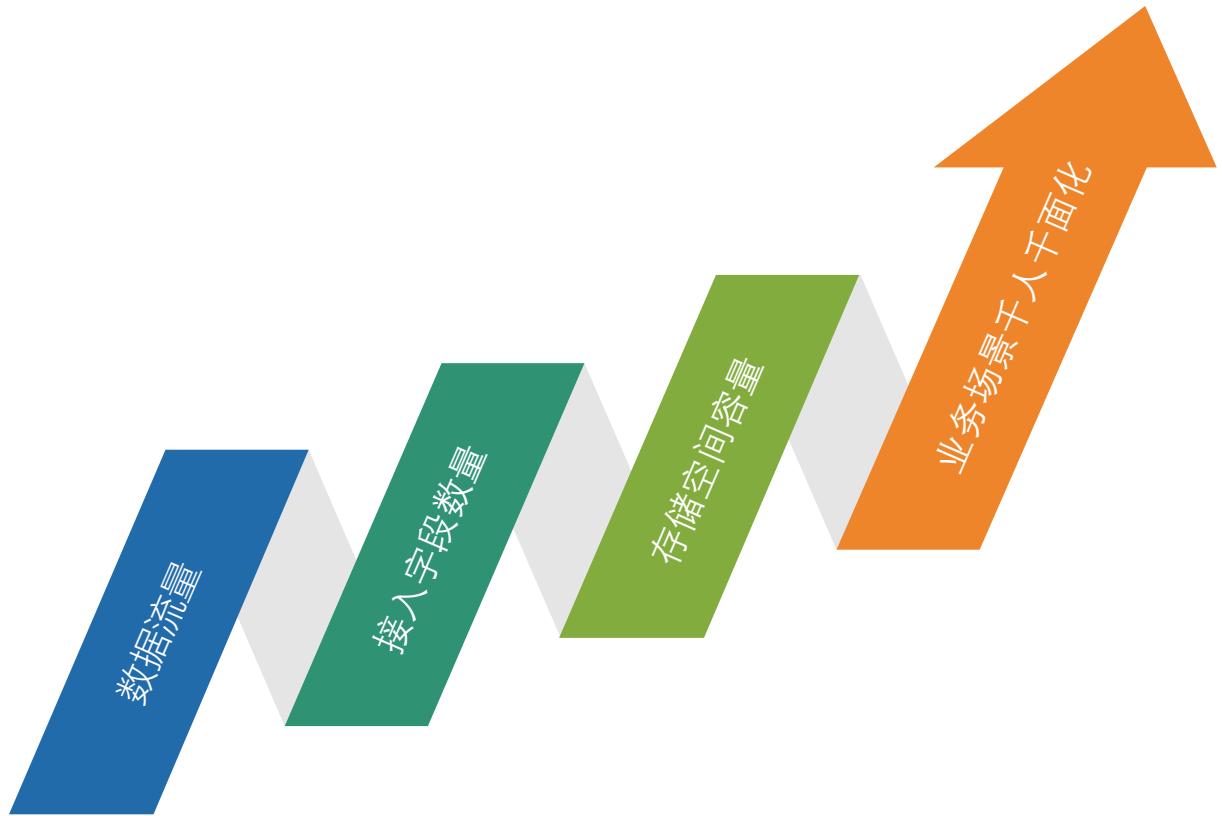
```
//转换\t
f.Message = strings.Replace(f.Message, "\t", "+-", -1)
//处理成统一的结构, es发过来的数据, 有_source, 有doc, 统一字段, 不然业务没办法解析
f.Message = strings.Replace(f.Message, "_source", "doc", -1)

str := strings.Split(f.Message, "+-")
if len(str) < 2 {
    logger.Ex(ctx, tag, "error:[%v], params[%v]", "no data", f.Message)
    return "split err", err
}
```

数据漏抓问题

数据乱序问题

数据修复困难问题



2亿

数据流量

平均每天数据监控流量2~5亿

100

月均接入字段数量

平均每天接入字段数量

2T

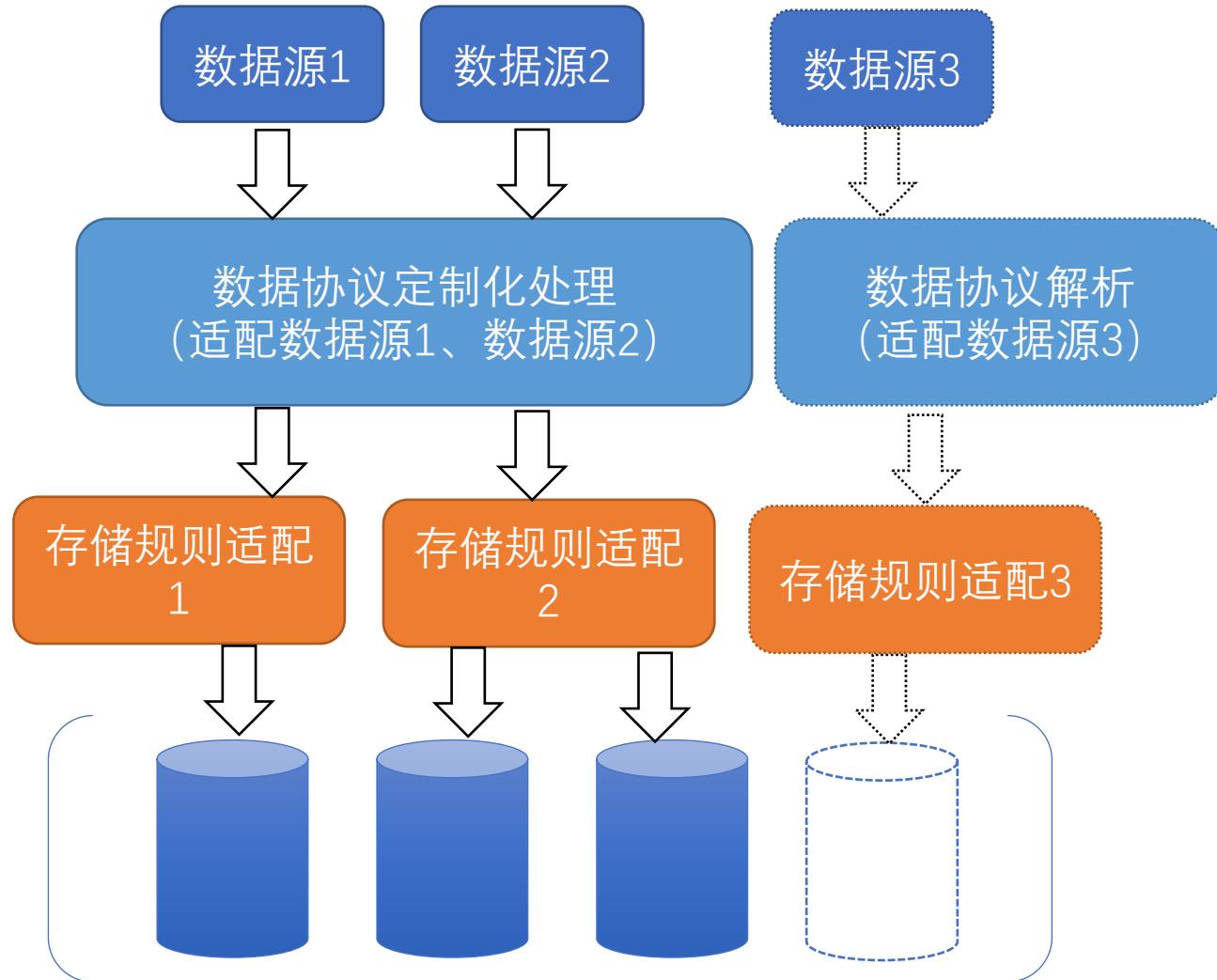
存储空间容量

预估2年数据存储

∞

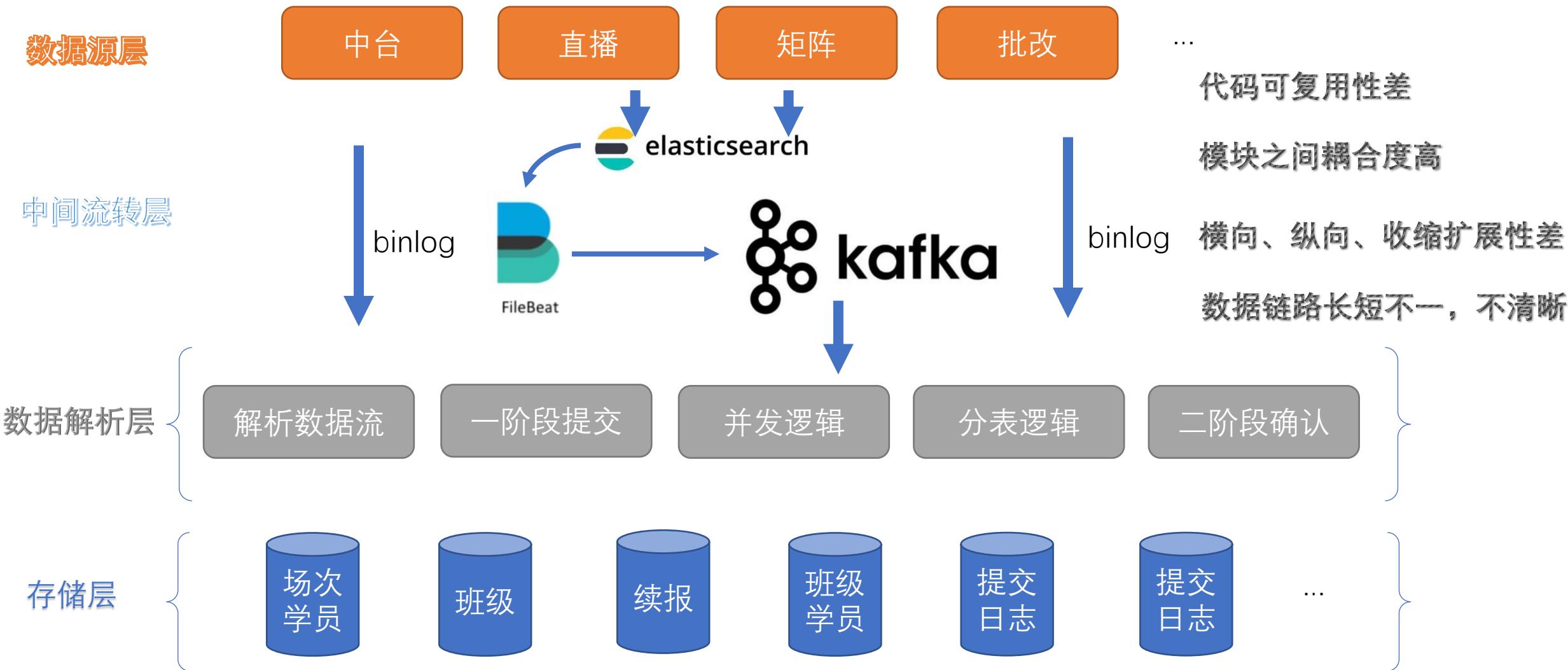
任意场景匹配任意用户数据

在特定的场景给特定的用户提供特定的服务



- 问题1：协议、存储、配置定制化
- 问题2：数据接入成本高
- 问题3：业务耦合度高

早期数据存储方案-定制化的数据服务中心-分层结构



旧的数据服务系统满足不了智慧教学系统的业务目标



数据服务

- (1) 平均每天可接入字段梳理3~5例
- (2) 每次接入一次新业务字段，平均接入时间3天
(2天研发，1天测试)
- (3) 数据稳定性无法保障



字段化

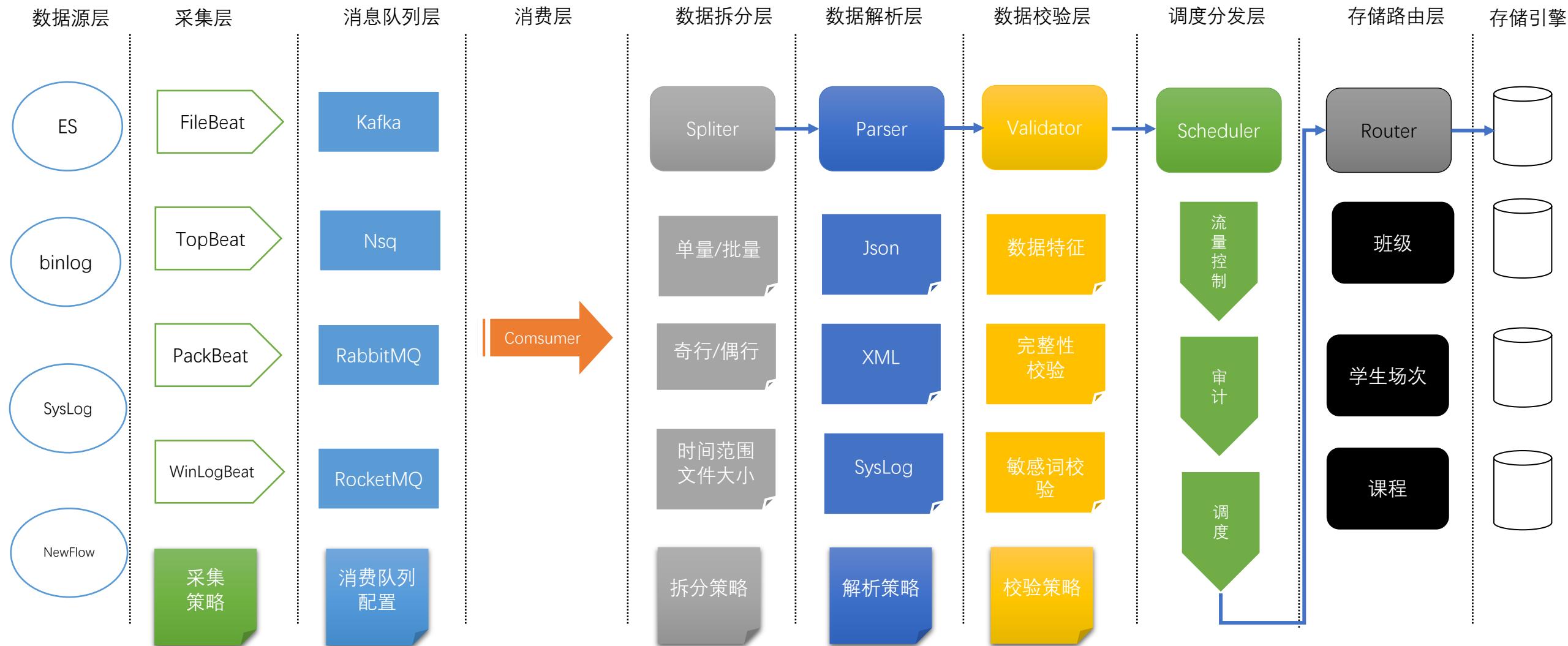
- (1) 平均每月100+的新字段接入需求
- (2) 接入字段时间要求T+1生效(1天内上线)
- (3) 取消定制化的业务存储，而用字段化组合方式存储，能够支持的场景更加丰富



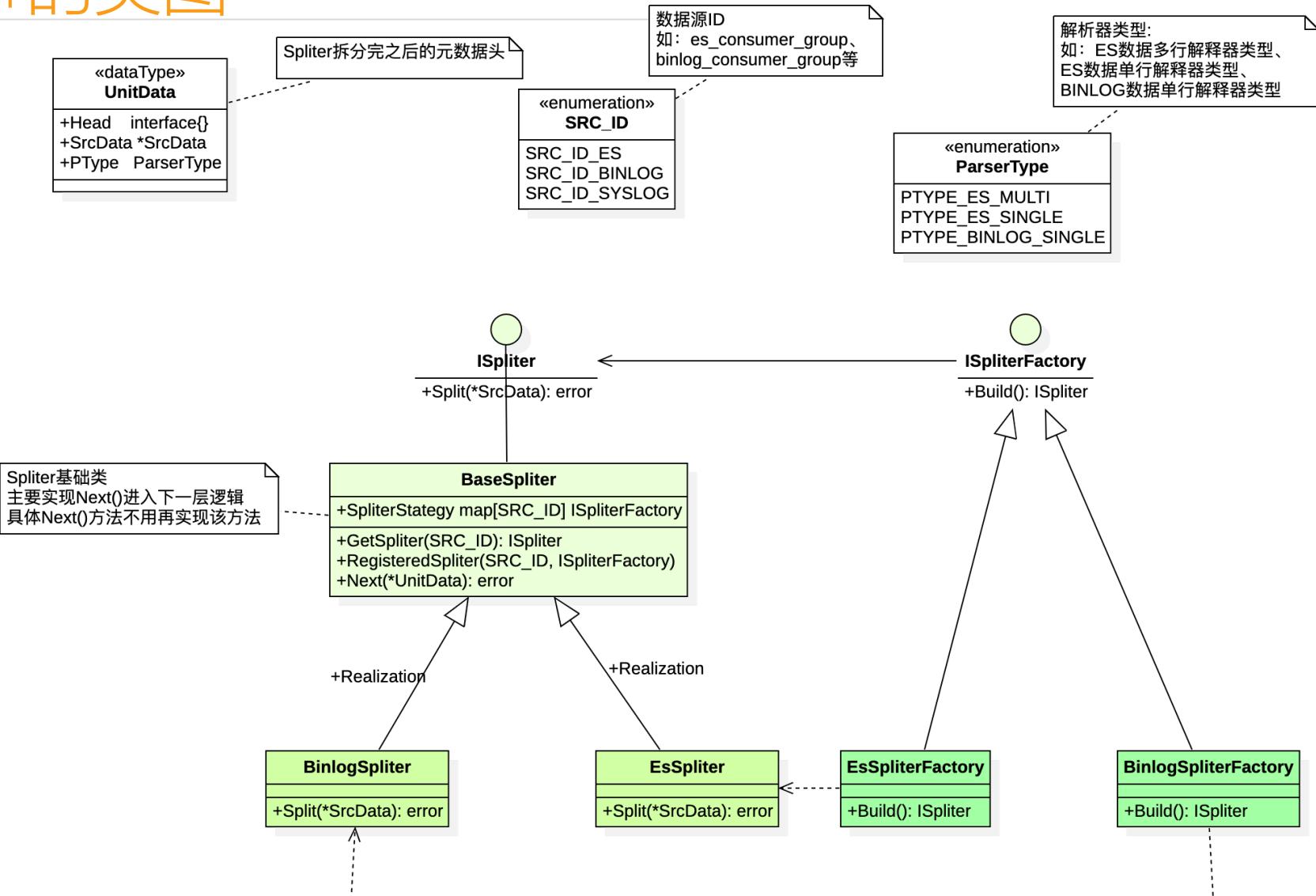
场景化

- (1) 字段场景化筛选
- (2) 业务场景化配置分发

数据服务架构改进版（早期Arris雏形）-解决存储服务能力



其中某层-Splider的类图



ISpliter

```
/*
    Arris抽象的Spliter拆分器
*/
type ISpliter interface {
    /*
        对源数据进行拆分
        根据不同的特征规则得到不同的单元头UnitData 和与当前UnitHead对应的解析器
    */
    Split(ctx context.Context, srcData *common.SrcData) error

    /*
        进入下一层，将数据交给对应的Parser处理
    */
    Next(ctx context.Context, unitData *common.UnitData) error
}
```

BaseSpliter

```
/*
基础的Spliter类，主要是实现Next方法，因为进入下一层，每个具体的Spliter业务相同
将全部相同的逻辑放在 Base的Next实现
*/
type BaseSpliter struct{}

//空实现，改方法交给具体的Spliter实现
func (base *BaseSpliter) Split(ctx context.Context, srcData *common.SrcData) error {
    return nil
}

//进入下一层 Paser解析器
func (base *BaseSpliter) Next(ctx context.Context, unitData *common.UnitData) error {
    //1. 获取解析器Type， 得到具体的解析器
    parser := GetParser(unitData.PType)
    return parser.Parse(ctx, unitData)
}
```

RegisteredSpliter

```
/*
 * 静态SRC_ID与Spliter工厂策略表
 */
type SpliterStategy map[arrisProto.SRCType]ISpliterFactory

var gSpliterStategy SpliterStategy = make(map[arrisProto.SRCType]ISpliterFactory)

//获取Spliter实例
func GetSpliter(srcID arrisProto.SRCType) ISpliter {
    factory := gSpliterStategy[srcID]
    spliter := factory.Build()

    return spliter
}

func RegisteredSpliter(srcID arrisProto.SRCType, factory ISpliterFactory) {
    if _, ok := gSpliterStategy[srcID]; ok {
        panic("Repeat Register Spliter, SRC_ID = " + srcID)
    }

    //添加生产策略
    gSpliterStategy[srcID] = factory
}
```

Spliter具体实例

```
// BinlogSpliter工厂
type BinlogSpliterFactory struct {
    arris.BaseSpliter
}

// BinlogSpliterFactory 构造器
func (factory *BinlogSpliterFactory) Build() arris.ISpliter {
    return &BinlogSpliter{}
}

// ES单批量拆分Splitter
type BinlogSpliter struct {
    arris.BaseSplitter
}

// Split(srcData *common.SrcData) []*common.UnitData
func (spliter *BinlogSpliter) Split(ctx context.Context, srcData *common.SrcData) error {
    //binlog 如果不需要拆分那么直接构造UnitData, 进入下一层
    // ...
    //进入下一层
    return splitter.Next(ctx, unitData)
}

//初始化 注册当前Splitter
func init() {
    arris.RegisteredSplitter(arrisProto.SRC_ID_BINLOG, &BinlogSpliterFactory{})
}
```

数据服务架构改进版-再次面临的问题

接入成本大

数据服务接入字段成本大，需要人工定制化接入

数据无标准化

不适用于今后数据整合和数据分析，用户画像、用户精准服务等业务扩展。

新增字段业务繁琐

数据定制化字段、增值字段计算策略业务流程定制化，无标准化流程，业务复杂性过大。

数据安全性低

数据无监控，修复只是定期脚本修复，数据延迟高。

数据维度隔离性差

存储表结构与业务耦合度太高，无法满足全数据接入通用性。

用户获取数据代价高

数据服务业务相关数据获取，无法开放用户自身配置生成获取接口和协议。

升级

平台接入配置化

开放数据源配置接入、数据字段配置接入、自定义字段配置接入，接入方用户开放配置权限。

标准输入/输出协议

制定Arris数据流入及流出标准协议，提高数据管理可复用性。

数据接入通用模式化

接入流程提供通用标准流程，无需研发介入，自动化适应。

数据监控平台：鹰眼系统

鹰眼系统作为Arris旁路系统，完全监控Arris每条数据流入及流出的状态和实时修复。

自由维度字段体系

维度自由组合体系，非一个业务一张表，而是主字段自由组合成为维度概念。维度和维度隔离性强。

数据订阅/查询标准

统一数据订阅协议及标准通用流程，全自动查询平台功能。



第二部分

Arris (数据运营系统)



Arris 取名中国成语：
“海纳百川”

(All Rivers Run Into Sea)

形如具有包容及兼容一切数据的数据海。

所有的河流、川水(指：任意类型的数据)最终都能够流向并溶入Arris系统，表示对任意数据的支持与海纳的意义，也是Arris系统的设计初衷。

Arris名字诞生于 2020 年 11 月



Arris的技术建设目标

通过海量的应用数据储备建设

快(实时), 准(数据可靠), 全(数据齐全)

的应用数据服务能力，高效支持的智慧教学系统数据查询和检索场景，
数据驱动做精准高效有品质服务。

Arris 业务建设目标：

- 1、支持智慧教学系统做**千人千面**数据投放。
- 2、**Arris平台化**，高效数据运营管理。
- 3、**海量数据**写入输出支持。
- 4、**数据稳定性保障、数据准，架构稳。**





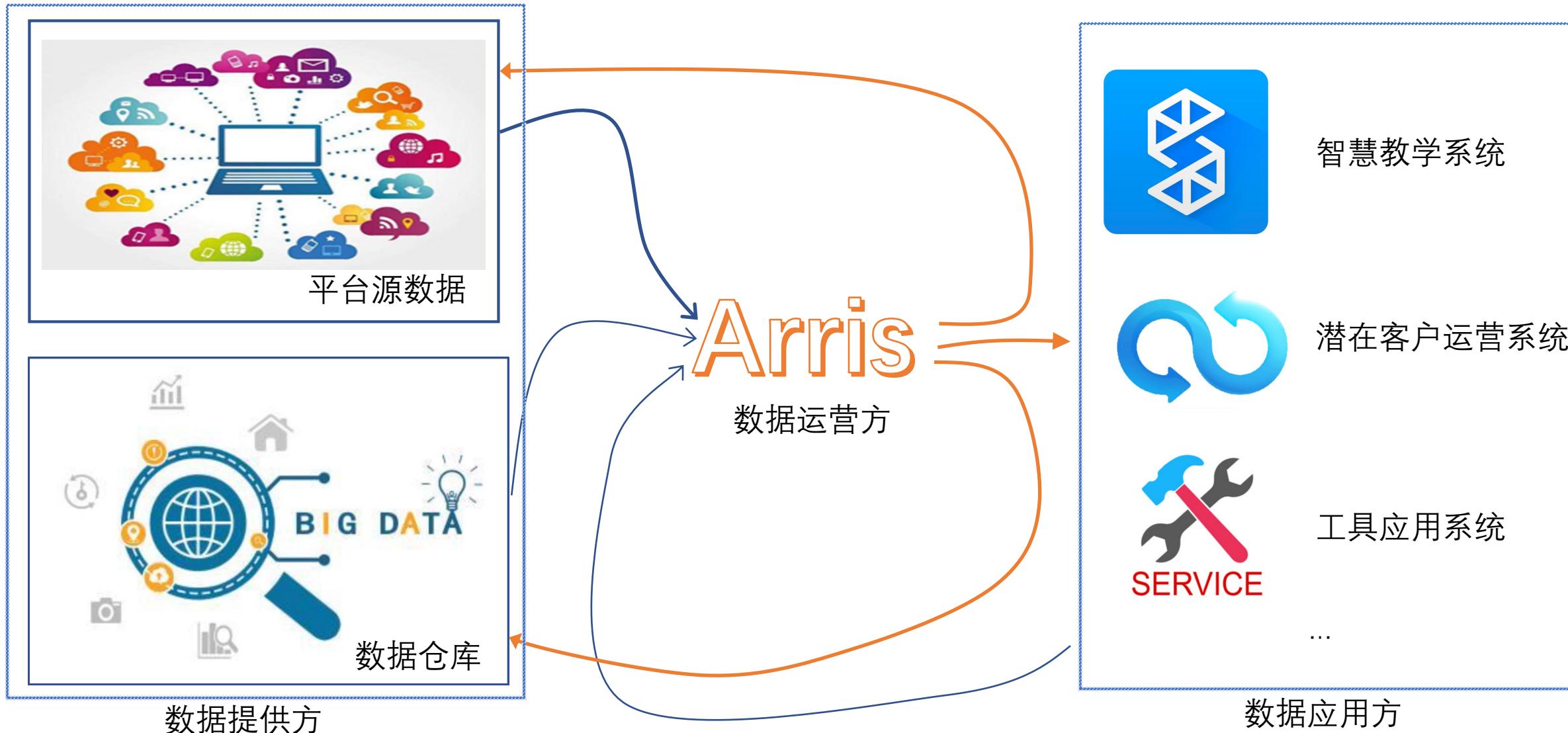
用户维度

- 数据提供方
 - 数据应用方

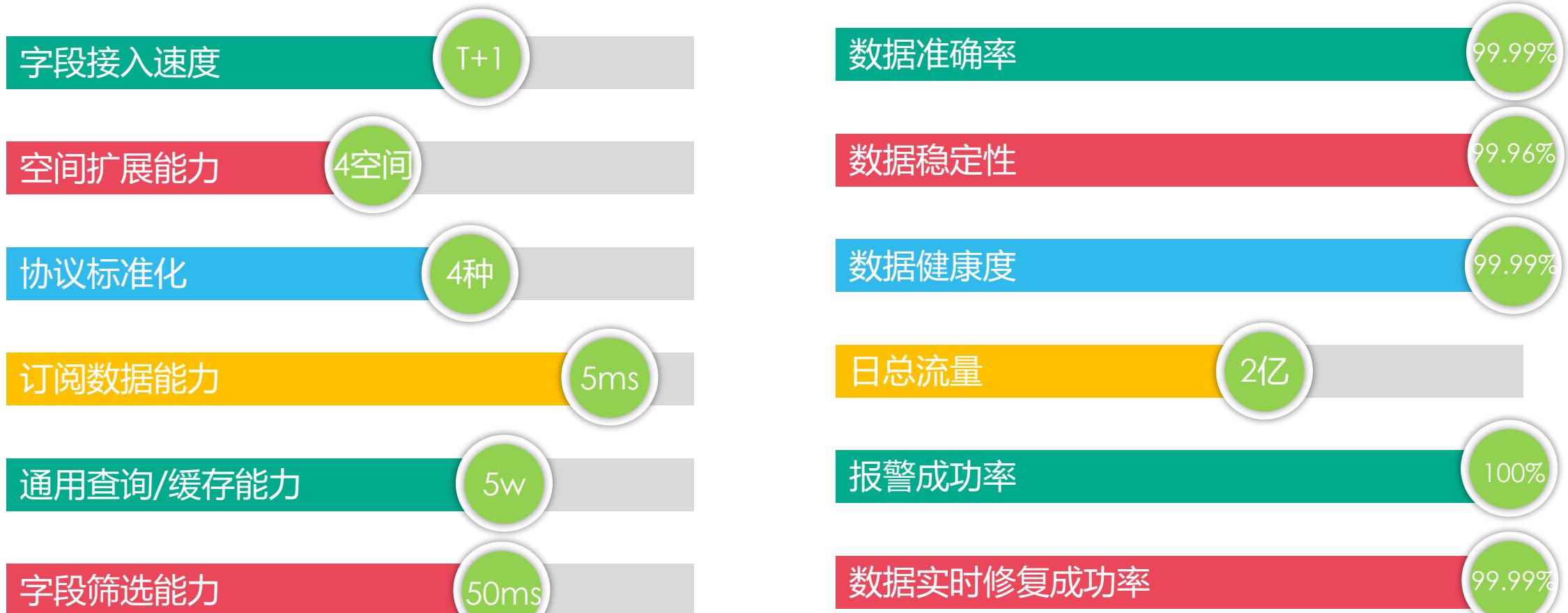
框架维度

- 数据层
 - 筛选层
 - 业务层

Arris在数据流中充当的角色



Arris目前承担的能力



Arris成长路线



01

字段

骨架与字段体系

Arris标准字段编码协议

维度组合、分库分表配置算法等

02

读写

标准接入与实时订阅

Arris标准接入协议、数据准入

标准订阅协议、订阅规则

03

空间

多空间存储

Arris开放存储空间配置，分库分表逻辑策略

多空间写入/订阅/查询

04

平台

字段开放平台

Arris租户数据接入与订阅平台

交付/运营即可完成数据运营

字段体系表关系-关键概念

ArrisID 区分本次接入的唯一标识

主字段 能够独立表示一个维度的字段

维度 由**主字段**自由组合的概念

"ARRIS-f0dpu0tuw[REDACTED]gca"

ID	field	field_name	dim_name
0001	class_id	班级ID	class
0002	stu_id	学生ID	stu
0003	plan_id	场次ID	plan
0004	course_id	课程ID	course
0005	teacher_id	老师ID	teacher
0006	wx_id	微信ID	wx
0007	time_id	时间ID	time

ID(string[4L])	dim_name	level	name	serach_index	comment
1001	class	1	班级	arris_se[REDACTED]	班级一维数据
1002	stu	1	学生	arris_se[REDACTED]	学生一维数据
1003	plan	1	场次	arris_se[REDACTED]	场次一维数据
1004	course	1	课程	arris_se[REDACTED]	课程一维数据
1005	teacher	1	老师	arris_se[REDACTED]	老师一维数据
2001	class_stu	2	班级学生	arris_se[REDACTED]	班级学生二维数据
2002	class_plan	2	班级场次	arris_se[REDACTED]	班级场次二维数据
2003	class_course	2	班级课程	arris_se[REDACTED]	班级课程二维数据
2004	class_teacher	2	班级老师	arris_se[REDACTED]	班级老师二维数据
2005	lp_stu	2	lp学生	arris_se[REDACTED]	lp学生二维数据
3001	class_stu_plan	3	班级学生场次	arris_se[REDACTED]	班级学生场次三维数据
4001	class_stu_plan_course	4	班级学生场次课程	arris_se[REDACTED]	班级学生场次课程四维数据

Arris骨架与字段体系-字段编码协议

字段ID = 字段种类(2L) + 数据类型(2L) + 维度ID(4L) + 编号(6L)

01 11 2004 100028

含义：系统字段，该字段为int32数据类型，该字段属于0101学服Arris租户，维度是class_plan班级场次二级维度数据，编号为100028。



Arris骨架与字段体系-数据协议

数据映射配置

字段值转换设置

新增映射

请按要求输入映射关系

X

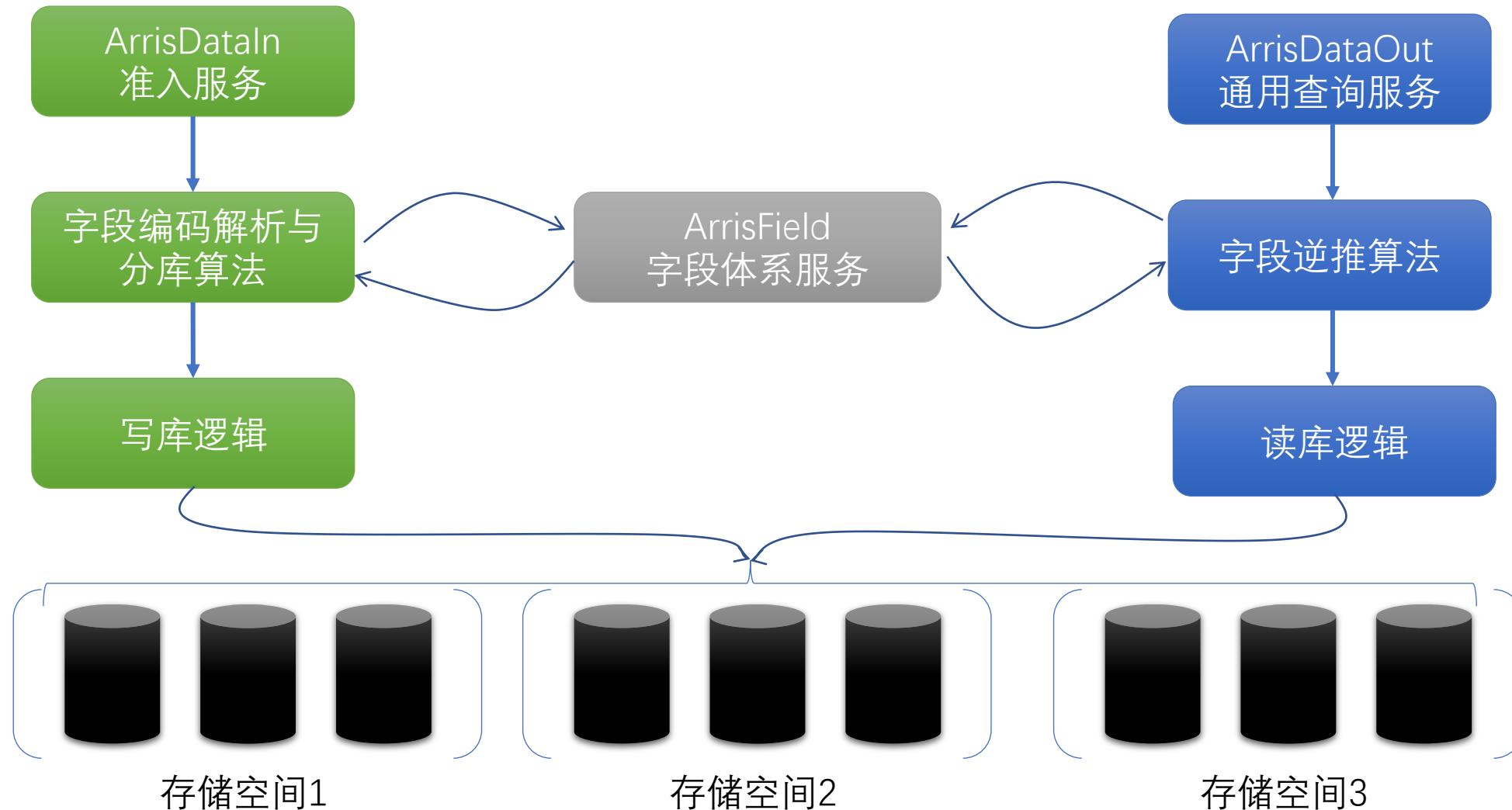
注：
1. 原值和新值必须成对输入,否则否则设置不生效;
2. 原值只允许输入数字;



数据标准输出格式

字段类型	原数据格式	输出格式
文本	"文本"	{"val": ["文本"], "detail": [], "link": []}
数字	123	{"val": [数字], "detail": [], "link": []}
单选	-	{"val": [-], "detail": [], "link": []}
多选	-	{"val": [-], "detail": [], "link": []}
日期	"123456" 或 123456 或 {"val": [123456]} 或 "2021-1-1" 新接入字段统一为int型时间戳	{"val": [12388731276], "detail": [], "link": []}
时长	-	{"val": [50], "detail": [], "link": []}
链接	-	{"val": ["百度搜索", "腾讯新闻网"], "detail": [], "link": ["url1", "url2"]}
附件	-	{"val": ["文件名1", "文件名2"], "detail": [], "link": ["url1", "url2"]}

Arris标准接入与实时订阅-数据准入

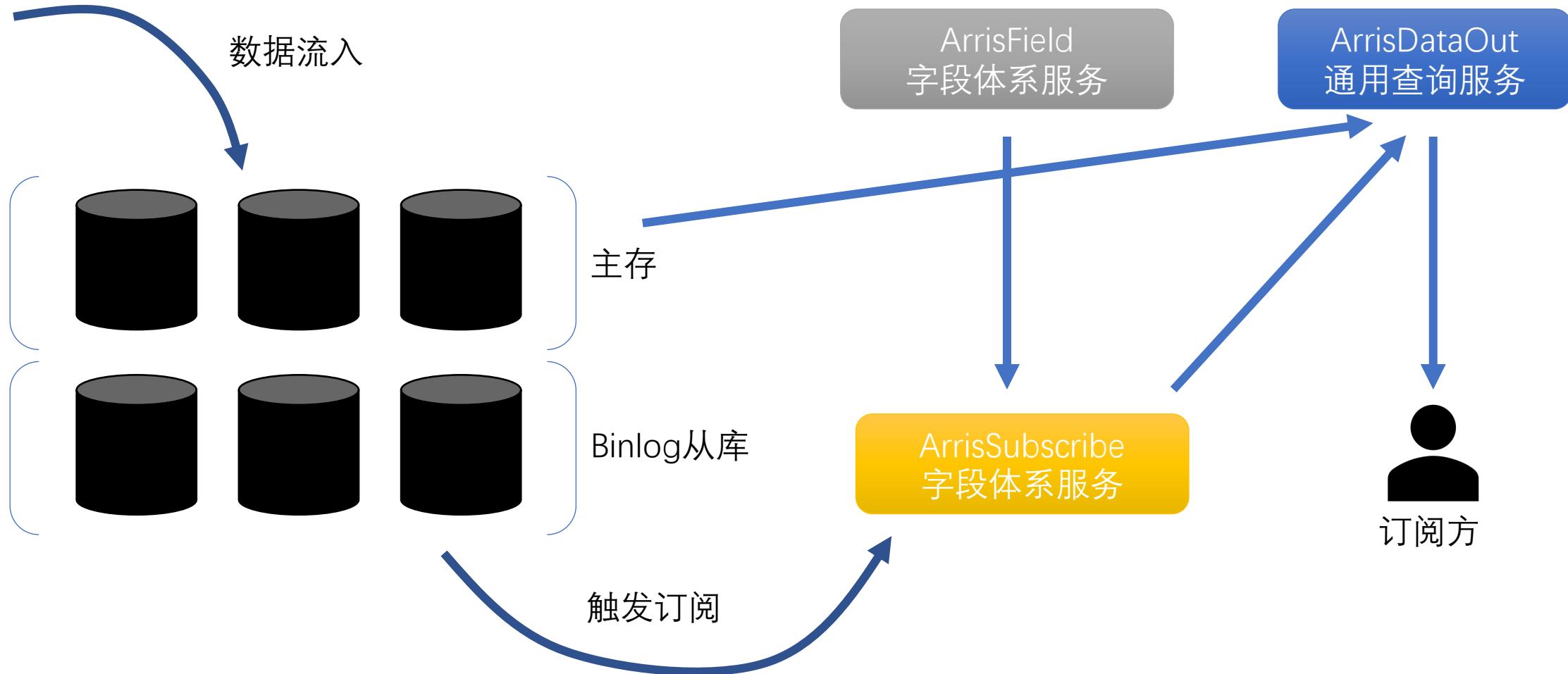


Arris标准接入与实时订阅-数据准入协议

```
{  
    "version": "v1.0",                                //Arris平台提供  
    "arris_id": "ARRIS-f0102-0000000000gca",          //Arris平台提供  
    "name": "班级学生在线直播数据(当前接入数据注释)", //Arris平台提供  
    "app_id": "0102",                                 //Arris平台提供  
    "app_name": "网校学服轻直播",                      //Arris平台提供  
    "app_key": "00000000000000000000000000000000",     //Arris平台提供  
    "arris_message": {  
        "time": 1615541291707,                          //接入方提供：数据发送时间<接入方需根据时间保证数据的正确时序性>  
        "cmd": "update",                               //接入方提供：数据操作类型  
        "fields": {  
            "class_id": 8888,                           //接入方提供：<主字段信息>  
            "stu_id": 99999,                            //接入方提供：<主字段信息>  
            "class_name": "小高语文",                   //接入方提供  
            "stu_refund_rate_so": 36.87,                //接入方提供  
            "class_stu_name": 30                         //接入方提供  
        },  
        "change_fields": [                            //接入方提供：如果cmd是update，需提供被修改的字段  
            "class_name"  
        ]  
    }  
}
```



Arris标准接入与实时订阅



Arris标准接入与实时订阅-数据订阅协议

```
{  
    "version": "v2.0",  
    "topic": "arrpub_xuefu_note",  
    "app_id": "duy",  
    "app_key": "324fb8f1cfba295eb9b1267a",  
    "dim": "3001",  
    "listen": [  
        {  
            "arris_id": "ARRIS-f935f167",  
            "fields": [  
                "status_name"  
            ]  
        }  
    ],  
    "publish": [  
        "status_name",  
        "plan_id",  
        "stu_id",  
        "class_id",  
        "updated"  
    ],  
    "filter": "and",  
    "condition": [  
        {  
            "field": "status_name",  
            "operator": "==",  
            "value": "已提交"  
        }  
    ],  
    "action": "create|binlog",  
    "extra_args": ""  
}
```

//Arris平台提供
//订阅方提供：订阅Topic
//Arris平台提供：订阅租户id
//Arris平台提供
//订阅方提供：数据维度
//订阅方提供：监听字段

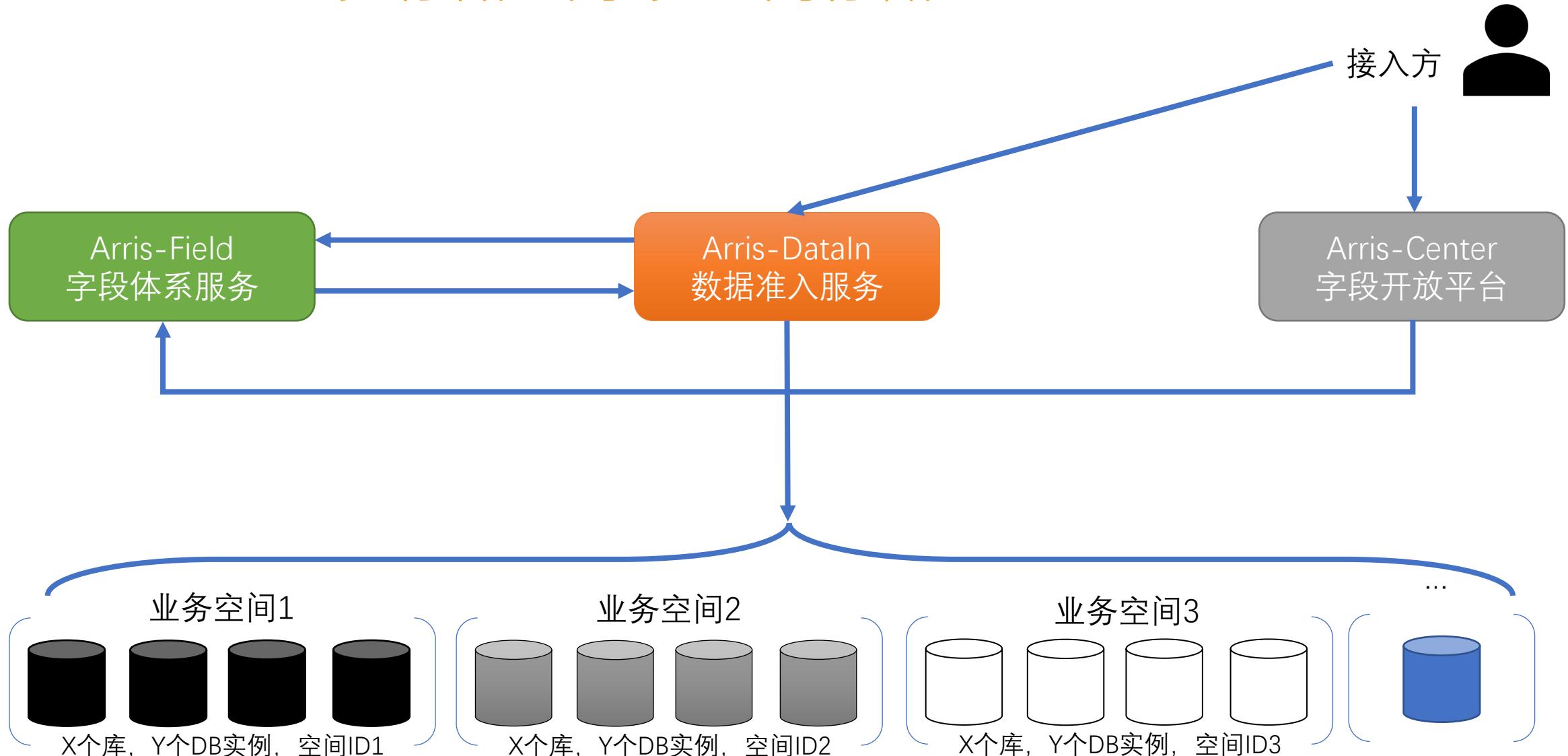
//订阅方提供：订阅需要推送的字段

//订阅方提供：筛选逻辑
//订阅方提供：具体筛选条件

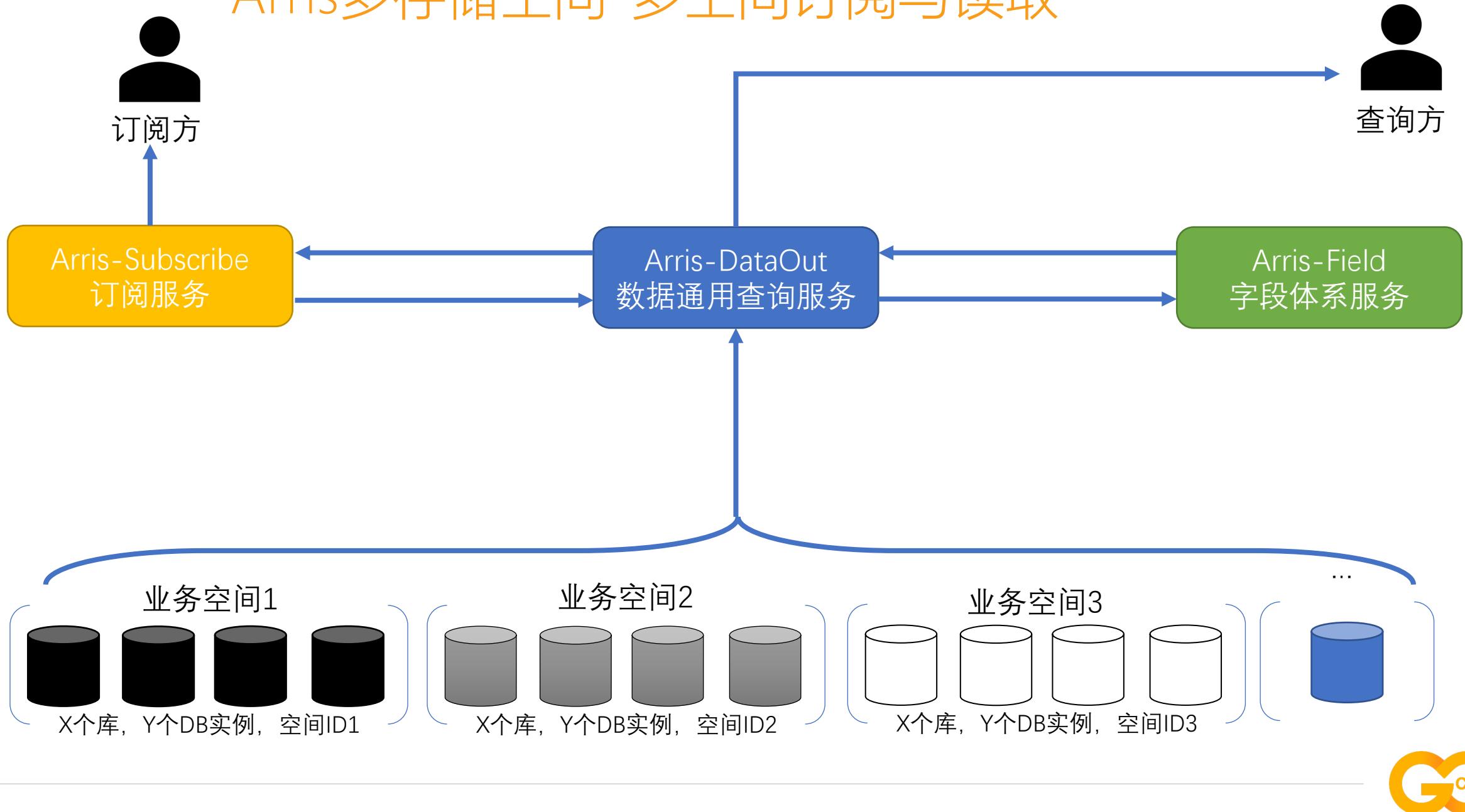
//订阅方提供： create/reset|datain/binlog



Arris多存储空间-多空间存储



Arris多存储空间-多空间订阅与读取



Arris字段开放平台

字段开放平台

我的字段

字段广场

我的字段 / 接入的字段 / 字段查看

ID: 66
字段中文名称 • 测试未通过

基本信息

中文名称* 班级实课
英文名称* 字段在接入方的唯一英文标识 uu_id
字段维度* 班级
主字段名* sy_id
字段提供方* 数仓
字段类型* 文本
字段释义* 这是一个字段释义
空值展现方式* 0 保持空值 自定义
数据类型* 整数型 浮点数型 字符串型
数据源字段英文名* 该字段在数据源的英文名称 stu_id

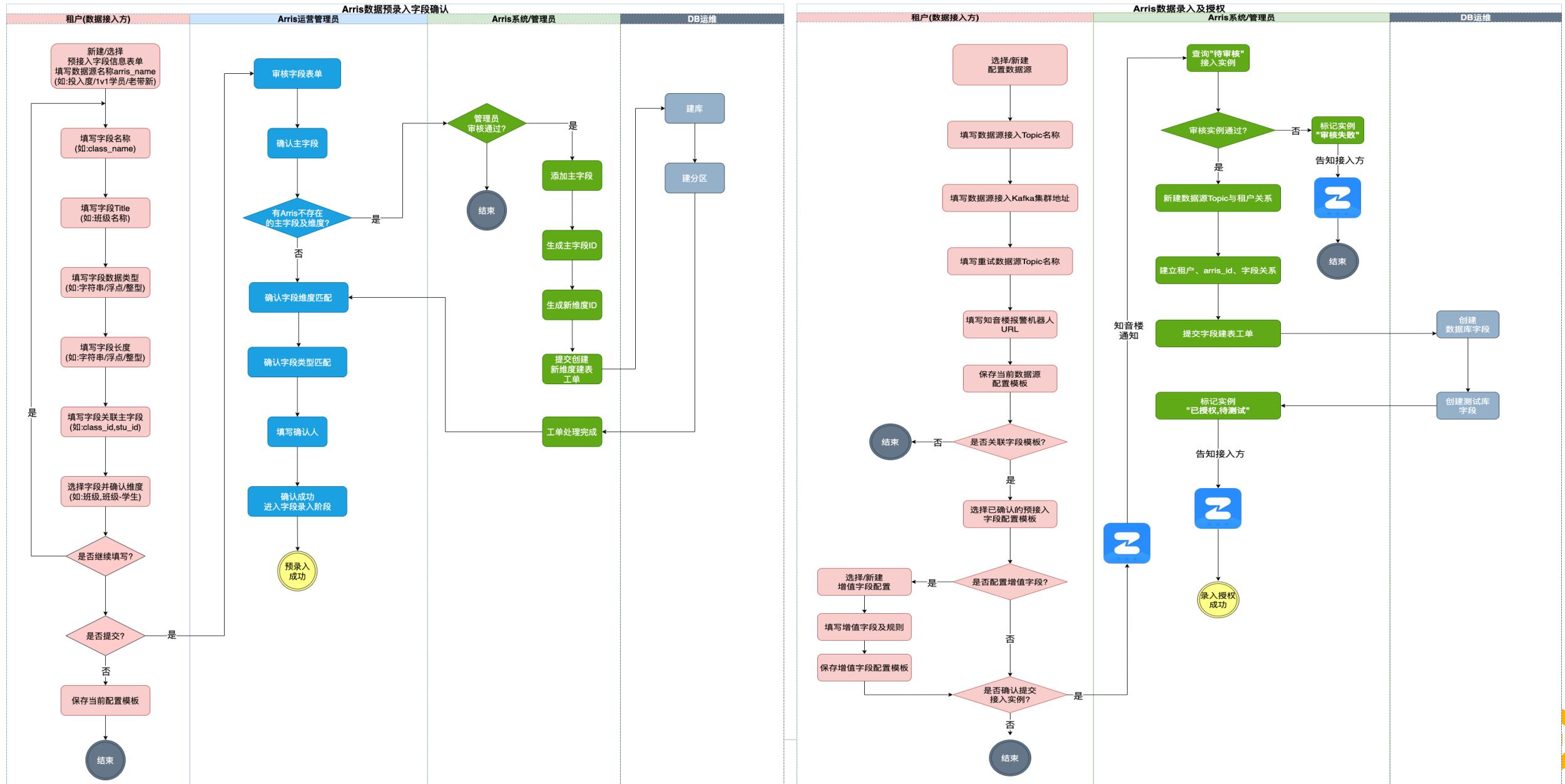
接入信息 查看《正式环境推送格式》

Topic名称* arris_DW_ KMF 4s
重试Topic名称* arretryarris_DW_ KMF 4S
知音楼机器人地址URL* 我们将通过该地址推送数据异常提醒 xesv5.robboot.com/4s666
源Kafka测试集群地址* 11.11.11.11:90
11.11.11.11:90
11.11.11.11:90
源Kafka集群地址* 11.11.11.11:90
11.11.11.11:90
11.11.11.11:90

预录入 2023年8月13日14:22
接入 2023年8月13日14:22 最近一次更新 2023年8月13日14:22



Arris字段开放平台



第三部分

Arris数据实时监控架 构设计

Arris数据稳定性的挑战

Much: 数据量大
每天数据(插入+更新)1亿+

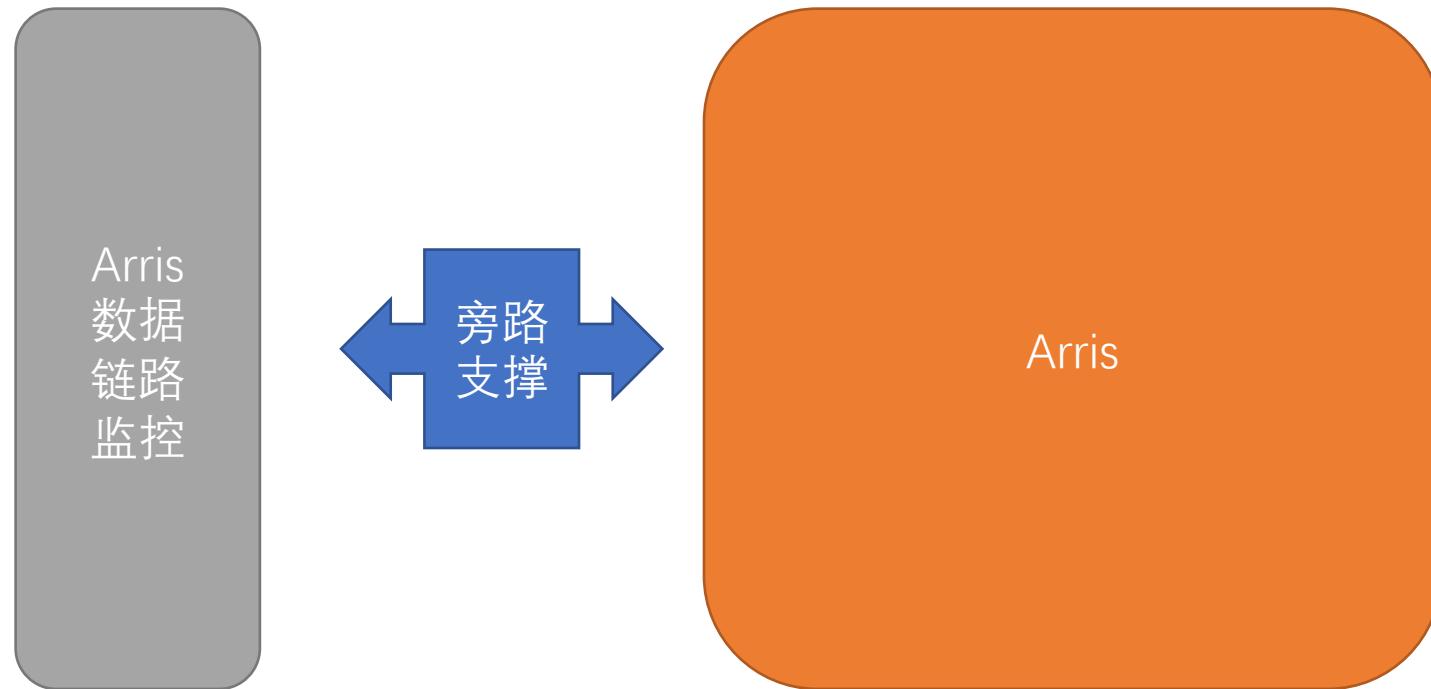
Alert: 报警快
监控到数据有问题能够立刻报警出来，并报警出来具体的含义，统计



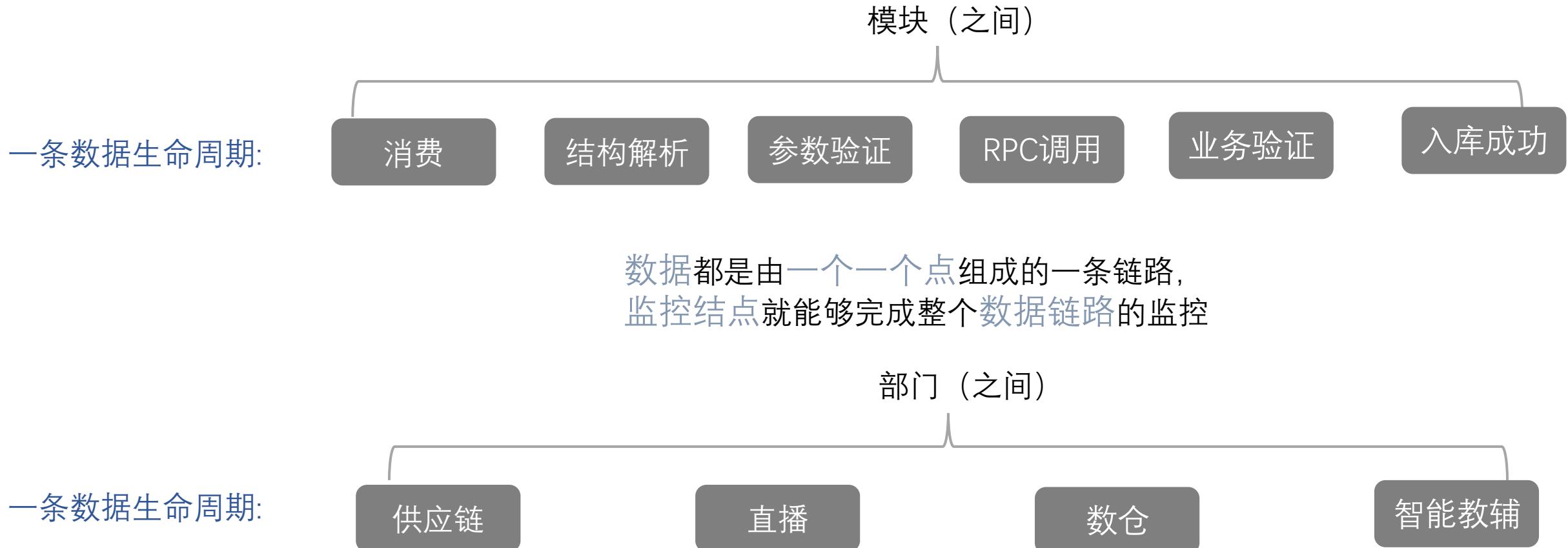
Monitor: 监控全
怎么能监控每一条数据的健康度、水位、数据链路、准确率、稳定性

Repair: 修复快
报警后不用通过人工去读日志的方式去手动修复，而是发现数据就自动修复，并且要求数据跟源数据要保持一致

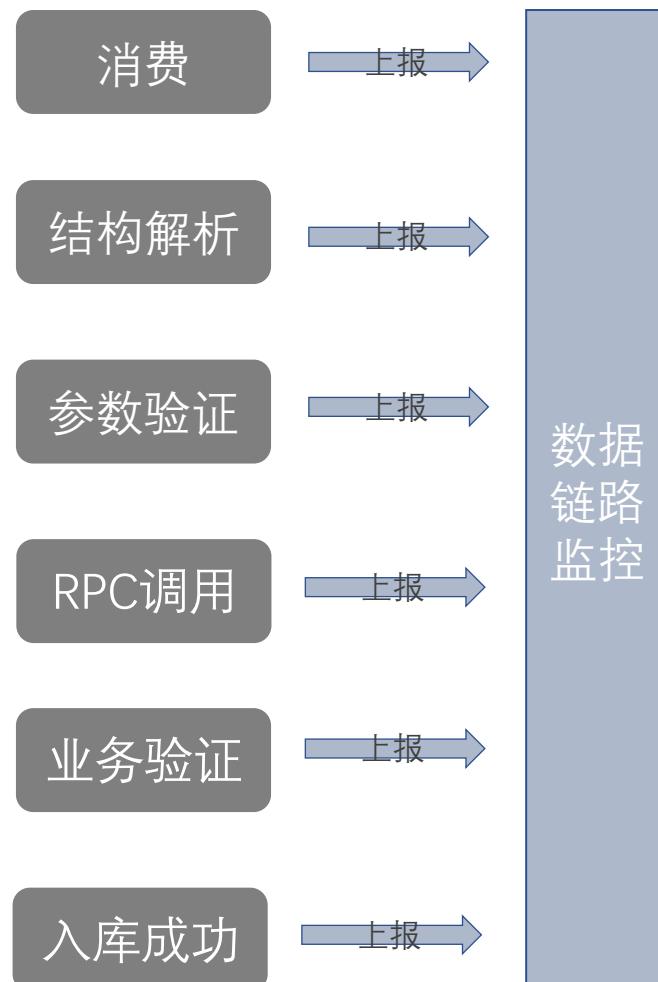
Arris数据监控系统-鹰眼(EagleEye)



一条数据的路径问题分析



链路监控思路



我们监控系统的协议怎么制定呢？

- 1、 UniqueID(traceID): 一条数据整个链路的唯一ID
- 2、 Source: 这条数据来源于哪 ? (可以是表、ES索引等)
- 3、 SourceID: 这条数据的唯一ID ? (用于修复数据用)
- 4、 Action: 这条数据的动作 ? (可以insert、update等)
- 5、 Point: 这条数据的结点 ? ([100,110,120,200])
- 6、 Time: 结点上报时间 ? (毫秒)
- 7、 Extra: 附加信息 ? (业务方自用)

线上的真实协议格式

```
{  
    "unique_id": "c4kq65j044205320210828",  
    "mod_id": 1,  
    "grp_id": 1,  
    "biz_id": "default",  
    "source": "*****class_plan",  
    "source_id": "*****",  
    "point": 110,  
    "action": 1,  
    "modify_time": 1630121404042
```

存储选型

存储问题

- 每天6亿+的数据量该存在什么地方？
- 同时要保证写的性能
- 同时要高可用
- 还要有专人维护
- 能够尽量少的学习成本

280G



Apache Hive

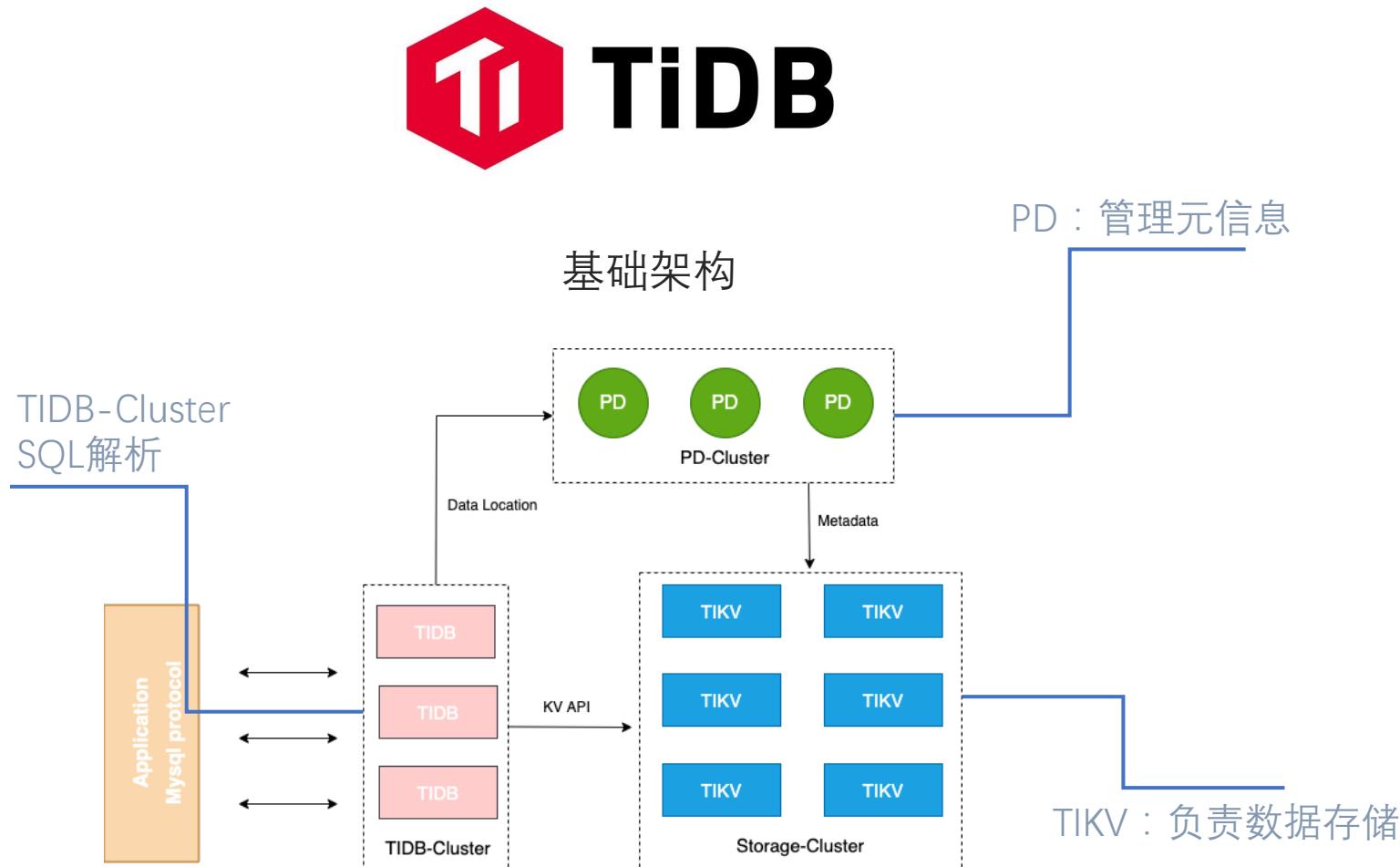


TPS:2W+



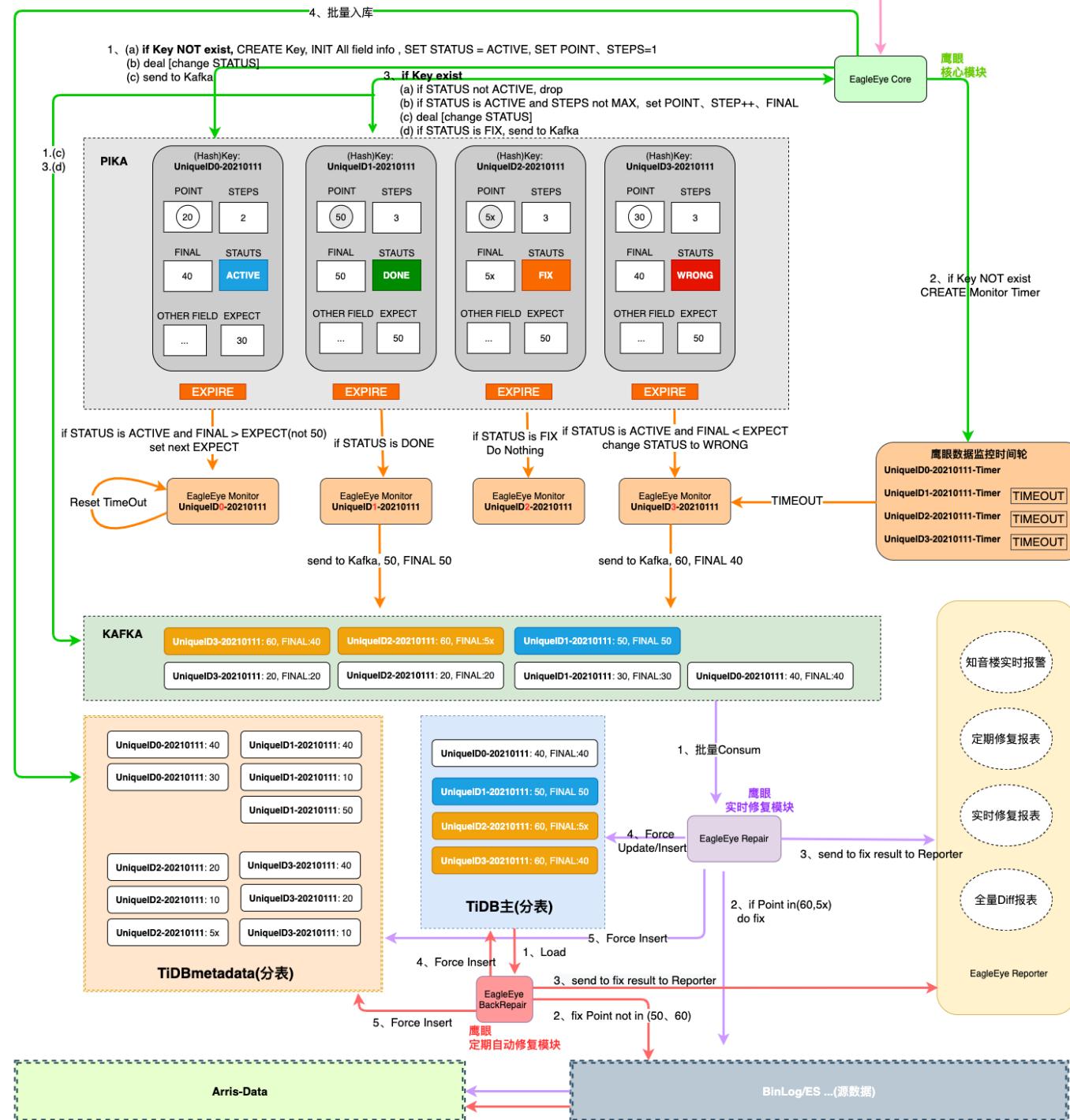
elasticsearch

存储选型



- 高存储
 - Kv存储，横向扩展
- 高可用
 - TiDB-Custer无状态、横向扩展
 - PD、TIKV基于Raft协议，天然高可用
- 高性能
 - 2计算节点+3PD+3TIKV 2W+的写
- 专人维护
 - 快速解决问题
- 易使用
 - 完全兼容Mysql协议

数据监控架构



数据监控与实时修复第一版收益

每天保障数据一致性

1亿

数据准确率

95%

数据稳定性

86%

数据健康度

95%

定时报警成功率

100%

定时修复成功率

99.9%



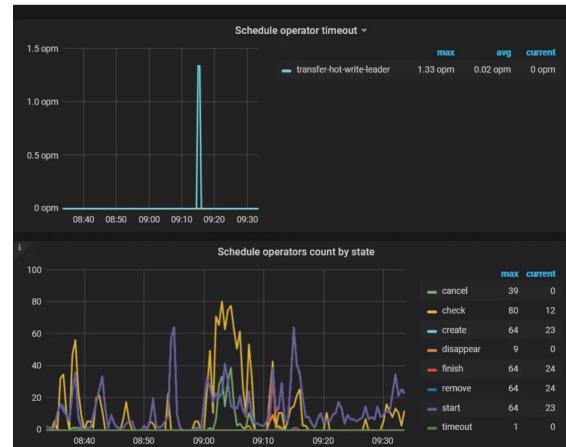
第一版问题总结

慢查询

慢查询展示了多条慢查询语句，主要涉及插入操作。例如：

```
INSERT IGNORE INTO `t_metadata`(`source_id`, `unique_id`, `STATUS`, `source`, `ACTION`, `modify_time`, `结束created, ↓`, `总执行时间 ⌂`, `updated` ) VALUES ('1088052_604672', '14.6 ms', 'bit0g', 'Jan 2, 2021 11:35 AM', '400.1 ms', 'study_detail', 'Jan 2, 2021 11:35 AM', '342.0 ms');  
INSERT IGNORE INTO `t_metadata`(`source_id`, `unique_id`, `STATUS`, `source`, `ACTION`, `modify_time`, `结束created, ↓`, `总执行时间 ⌂`, `updated` ) VALUES ('1088052_604672', '14.6 ms', 'bit0g', 'Jan 2, 2021 11:35 AM', '400.1 ms', 'study_detail', 'Jan 2, 2021 11:35 AM', '342.0 ms');  
INSERT IGNORE INTO `t_metadata`(`source_id`, `unique_id`, `STATUS`, `source`, `ACTION`, `modify_time`, `结束created, ↓`, `总执行时间 ⌂`, `updated` ) VALUES ('1088052_604672', '14.6 ms', 'bit0g', 'Jan 2, 2021 11:35 AM', '400.1 ms', 'study_detail', 'Jan 2, 2021 11:35 AM', '342.0 ms');
```

TIDB热写



数据未推送(兜底diff)



非真实时



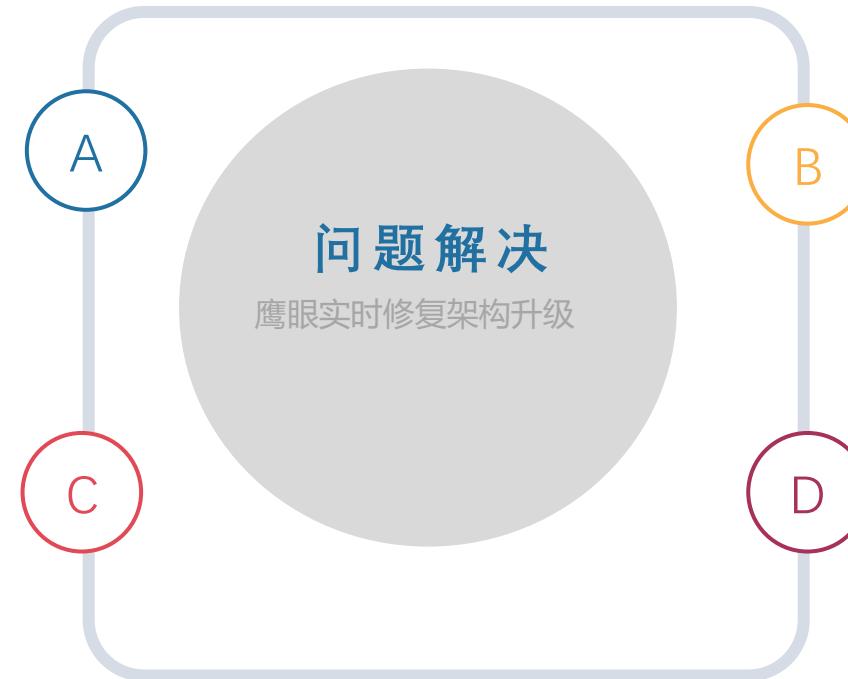
第一版问题解决

慢查询

- 1、单条 -> 批量，耗时下降百倍

热写

- 1、id主键设置为random, 解决热key问题



源头未推送

- 1、推动业务方从源头接入鹰眼

非真实时

- 1、核心架构升级，改被动为主动

消除慢查询

SQL ⓘ	结束运行时间 ⓘ	总执行时间 ⓘ	最大内存 ⓘ
"replace into `xes_ea...	2021-08-28` (source_id... 今天 20:13	300.2 ms	190.8 KiB
"replace into `xes_ea...	2021-08-28` (source_id... 今天 20:31	300.3 ms	192.7 KiB
REPLACE INTO `xes_ea...	2021-08-28` (source_id... 今天 20:31	300.9 ms	2.1 KiB
"replace into `xes_ea...	2021-08-28` (source_id... 今天 20:31	301.6 ms	209.7 KiB
INSERT IGNORE INTO `x...	2021-08-28` (data_20210828` ... 今天 20:31	301.9 ms	1.7 KiB
"replace into `xes_ea...	2021-08-28` (source_id... 今天 20:25	302.1 ms	228.6 KiB
"replace into `xes_ea...	2021-08-28` (source_id... 今天 20:31	302.4 ms	288.3 KiB
"replace into `xes_ea...	2021-08-28` (source_id... 今天 20:25	302.5 ms	184.3 KiB
INSERT IGNORE INTO `x...	2021-08-28` (data_20210828` ... 今天 20:31	302.8 ms	39.0 KiB
"replace into `xes_ea...	2021-08-28` (source_id... 今天 20:13	302.9 ms	87.1 KiB
"replace into `xes_ea...	2021-08-28` (source_id... 今天 20:25	303.7 ms	57.2 KiB
"insert ignore into ...	2021-08-28` (data_20210828` ... 今天 20:25	303.8 ms	136.4 KiB
"insert ignore into ...	2021-08-28` (data_20210828` ... 今天 20:25	304.1 ms	169.1 KiB
"replace into `xes_ea...	2021-08-28` (source_id... 今天 20:13	304.2 ms	195.9 KiB
REPLACE INTO `xes_ea...	2021-08-28` (source_id... 今天 20:31	304.2 ms	1.9 KiB
"replace into `xes_ea...	2021-08-28` (source_id... 今天 20:13	304.3 ms	189.8 KiB

消除TIDB热写



实时监控

- 定时器 → 时间轮

```
1 goos: darwin
2 goarch: amd64
3 pkg: timingwheel
4 BenchmarkTimingWheel_StartStop/N-1m-8      20000000    312 ns/op    86 B/op    2 allocs/op
5 BenchmarkTimingWheel_StartStop/N-5m-8       20000000    349 ns/op   118 B/op    2 allocs/op
6 BenchmarkTimingWheel_StartStop/N-10m-8     20000000    384 ns/op   165 B/op    2 allocs/op
7 BenchmarkStandardTimer_StartStop/N-1m-8     50000000    164 ns/op    80 B/op    1 allocs/op
8 BenchmarkStandardTimer_StartStop/N-5m-8     30000000    212 ns/op    80 B/op    1 allocs/op
9 BenchmarkStandardTimer_StartStop/N-10m-8    20000000    317 ns/op    80 B/op    1 allocs/op
10 BenchmarkGoZeroTimer_StartStop/N-1m-8      20000000   3799 ns/op   173 B/op    6 allocs/op
11 BenchmarkGoZeroTimer_StartStop/N-5m-8      20000000   4215 ns/op   173 B/op    6 allocs/op
```

- 高性能状态管理 → Redis(也考虑过RocksDB)

- K8s不支持本地挂载磁盘
- 稳定为先

STATUS

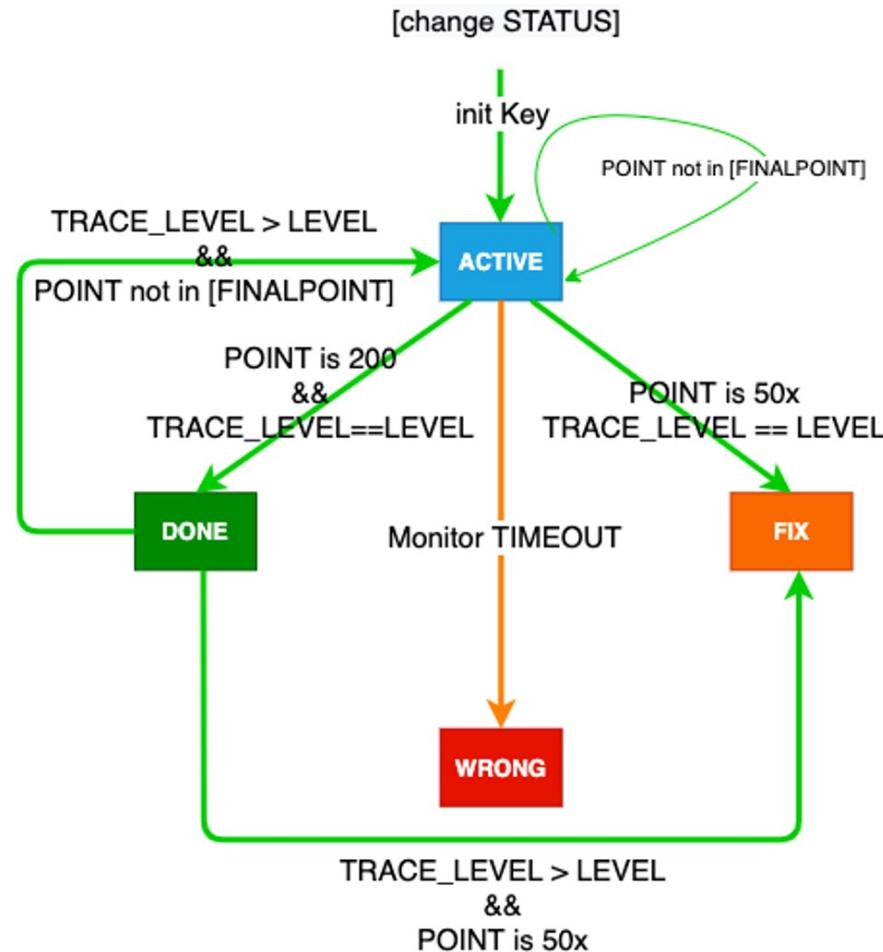
EXPECT

FINAL

POINT	含义	最终状态	前驱状态	后置状态
1	未知状态码			
90	上游发送消息(写ES/Binlog)成功			110
110	消息接收成功		无	120
120	调度接收成功		110	110
130	全调度完成		120	140
13x	Arris调度层扩展POINT		120、13x	140
140	Arris DataService 接收成功		130、13x	200、501、502、503、504、50x
200	入库成功	是	140	
501	RPC参数错误	是	140	
502	数据字段错误	是	140	
503	数据库操作失败	是	140	
504	数据库影响行数为0	是	140	
50x...	入库失败扩展POINT	是	140	
600	修复成功(default)			
601	实时修复成功			
602	自动修复成功			
603	手动修复成功			
60x...	扩展修复成功POINT			



状态流程



链路监数据状态幂等逻辑

```
func (core *EagleEyeCore) OnRecvMessages(ctx context.Context, datas *EagleEye.EStruct) error {
    tag := "EagleEyeCore.OnRecvMessages"

    //1、批量入库
    if err := core.CommitMetaData(ctx, datas.List); err != nil {
        logger.Ex(ctx, tag, "=Commit= metaData error, err = ", err)
        return err
    }

    //2、内存逻辑幂等处理
    if unitList, err := core.Uniq(ctx, datas.Map); err != nil {
        logger.Ex(ctx, tag, "=Uniq= error err = %+v, datas.Map = %+v", err, datas.Map)
        return err
    } else {
        //3、第三方缓存状态幂等处理
        if err := core.Save(ctx, unitList); err != nil {
            logger.Ex(ctx, tag, "=Save= error err = %+v", err)
            for _, unit := range unitList {
                logger.Ex(ctx, tag, "error = %+v", *unit)
            }
            return err
        }
    }

    return nil
}
```

链路监数据状态内存幂等逻辑

```
func (core *EagleEyeCore) doUniq(ctx context.Context, uniqueId string, srcList EagleEye.EEList)
(retStatus *EagleEye.EESaveUnit, e error) {
    tag := "EagleEyeCore.OnRecvMessages.Uniq"
    //确保srcList至少有一个元素
    // ...

    //返回的去重的数据单元
    retUnit := new(EagleEye.EESaveUnit)
    retUnit.Status = new(proto.EEStatus)
    retUnit.Message = &srcList[0]

    //init retUnit ...

    //依次遍历需要去重的数据
    for _, msg := range srcList {
        //每次收到一个EEMessage, 统计Step 累加1
        retUnit.Status.Steps++

        //计算final
        retUnit.Status.Final = core.FinalPoint(ctx, msg.Point, retUnit.Status.Final)
    }

    //去重之后的Message point, 以最终final数值为准
    retUnit.Message.Point = retUnit.Status.Final

    logger.Dx(ctx, tag, "After Uniq : retUnit.Status = %+v, retUnit.Message = %+v\n", *retUnit.Status, *retUnit.Message)

    return retUnit, nil
}
```

链路监数据状态内存幂等逻辑

```
func (core *EagleEyeCore) FinalPoint(ctx context.Context, point proto.EagleeyePoint, final proto.EagleeyePoint) proto.EagleeyePoint {  
    if point == proto.EE_POINT_UNKNOWN || final == proto.EE_POINT_UNKNOWN {  
        return proto.EE_POINT_UNKNOWN  
    }  
  
    //优先级1: 如果原final为最终状态, 则结果不变  
    if final == proto.EE_POINT_DATABASE_CMD_SUCCESS ||  
        final == proto.EE_POINT_RPC_PARAMS_INVALID ||  
        final == proto.EE_POINT_RPC_DATA_FIELDS_FAIL ||  
        final == proto.EE_POINT_DATABASE_CMD_FAIL ||  
        final == proto.EE_POINT_DATABASE_CMD_TOCOUNT_NULL {  
  
        return final  
    }  
  
    //优先级2: 如果新Point为最终状态, 则结果为当前Point  
    if point == proto.EE_POINT_DATABASE_CMD_SUCCESS ||  
        point == proto.EE_POINT_RPC_PARAMS_INVALID ||  
        point == proto.EE_POINT_RPC_DATA_FIELDS_FAIL ||  
        point == proto.EE_POINT_DATABASE_CMD_FAIL ||  
        point == proto.EE_POINT_DATABASE_CMD_TOCOUNT_NULL {  
  
        return point  
    }  
  
    //优先级3: 结果为二者最大值  
    return maxFinal(point, final)  
}
```

链路监数据状态缓存媒介幂等逻辑

```
func (core *EagleEyeCore) Save(ctx context.Context, unitList []*EagleEye.EESaveUnit) error {
    tag := "EagleEyeCore.OnRecvMessages.Save"

    //1.批量判断Key是否存在
    if splitList, err := core.FetchStatusMulti(ctx, unitList); err != nil {
        logger.Ex(ctx, tag, "FetchStatusMulti err=[%+v], unitList = [%+v]", err, unitList)
        return err
    } else {

        wg := new(sync.WaitGroup)
        // ... ...

        wg.Add(1)
        go core.doSaveKeyNotExist(ctx, splitList[common.NOTEXIST], wg) // 批量处理key不存在逻辑

        wg.Add(1)
        go core.doSaveKeyExist(ctx, splitList[common.EXIST], wg) // 批量处理key存在逻辑

        // ... 捕获协程错误
        wg.Wait()
    }

    return nil
}
```

链路监控数据状态缓存媒介幂等逻辑

```
func (core *EagleEyeCore) doSaveKeyNotExist(ctx context.Context, unitList []*EagleEye.EESaveUnit, wg *sync.WaitGroup) {
    tag := "EagleEyeCore.OnRecvMessages.Save.doSaveKeyNotExist"

    // ...
    //1. 创建KEY
    if err = core.CreateKey(ctx, unitList); err != nil {
        logger.Ex(ctx, tag, "Create Key ", unitList, " error, err = ", err)
        return
    }

    //2. 首次抵达数据上报 / 首次为FIX状态上报 /首次 成功上报
    for _, unit := range unitList {
        // init data ...
        if unit.Status.Status == proto.FIX {
            data.RepairPoint = proto.FAIL //首次Fix状态
            data.WrongTime = time.Now().UTC().UnixNano() / int64(time.Millisecond) // 设置触发错误时间
        } else if unit.Status.Status == proto.DONE {
            data.RepairPoint = proto.SUCC //首次Succ状态
            data.SuccTime = time.Now().UTC().UnixNano() / int64(time.Millisecond) // 设置触发错误时间
        } else {
            data.RepairPoint = proto.FIRST //首次抵达
            monitorList = append(monitorList, data.UniqueID) //添加定时器
        }
        //3. 提交中游实时修复Kafka . . .
    }

    //4. 首次抵达的数据，批量添加定时器
    core.AddMonitor(ctx, monitorList)

    return
}
```

链路监数据状态缓存媒介幂等逻辑

```
func (core *EagleEyeCore) doSaveKeyExist(ctx context.Context, unitList []*EagleEye.EESaveUnit, wg *sync.WaitGroup) {
    tag := "EagleEyeCore.OnRecvMessages.Save.doSaveKeyExist"
    // ...

    for _, unit := range unitList {
        if unit.StatusCache.Status != proto.ACTIVE { //1. 非ACTIVE数据直接丢弃, 不处理
            logger.Dx(ctx, tag, "====> CoreDrop : eeMessage = %v\n", *unit) //drop
            continue
        }

        unit.Status.Status = core.DealStatus(ctx, unit.Message.Point) //2. 处理状态

        unit.Status.Final = core.FinalPoint(ctx, unit.Message.Point, unit.StatusCache.Final) //3. 计算Final

        unit.Status.Steps += unit.StatusCache.Steps //4. Steps 累加

        saveUnitList = append(saveUnitList, unit) //加入到批量入库集合
        saveStatusList = append(saveStatusList, unit.Status)
    }

    if err = core.SaveStatus(ctx, saveStatusList); err != nil {//5. 批量保存Status
        logger.Ex(ctx, tag, "SaveStatus error, err = [%v], StatusList = [%v]", err, saveStatusList)
        return
    }

    //6. 如果为FIX / DONE 需要发送下游MQ ...
}

return
}
```

链路监数据状态缓存媒介幂等逻辑

```
func (core *EagleEyeCore) DealStatus(ctx context.Context, point proto.EagleeyePoint) proto.STATUS {
    switch point {
        case proto.EE_POINT_DATABASE_CMD_SUCCESS:
            return proto.DONE
        case proto.EE_POINT_RPC_PARAMS_INVALID,
            proto.EE_POINT_RPC_DATA_FIELDS_FAIL,
            proto.EE_POINT_DATABASE_CMD_FAIL,
            proto.EE_POINT_DATABASE_CMD_TOCOUNT_NULL:
            return proto.FIX
        default:
            return proto.ACTIVE
    }
    return proto.UNKONW
}
```

链路监控数据实时监控逻辑

```
func (m *EagleEyeMonitor) AddMonitorUniqs(ctx context.Context, uniqueIDs []string) error {
    tag := "EagleEyeMonitor.AddMonitor"
    perfutil.CountI(tag)
    defer perfutil.AutoElapsed(tag, time.Now())
    //trace
    span, ctx := traceutil.Trace(ctx, tag)
    if span != nil {
        defer span.Finish()
    }

    if len(uniqueIDs) == 0 {
        return errors.New("uniqueIDs is empty")
    }

    //超时检测
    timeRatio := m.timingWheel.TimeoutRatio()
    if timeRatio >= TimeoutLimit {
        return ErrTimeout
    }

    duration := time.Duration(m.timeout) * time.Millisecond

    task := m.createTaskSingalLevel(ctx, uniqueIDs)
    m.timingWheel.AfterFunc(duration, task)

    return nil
}
```

链路监数据实时监控逻辑

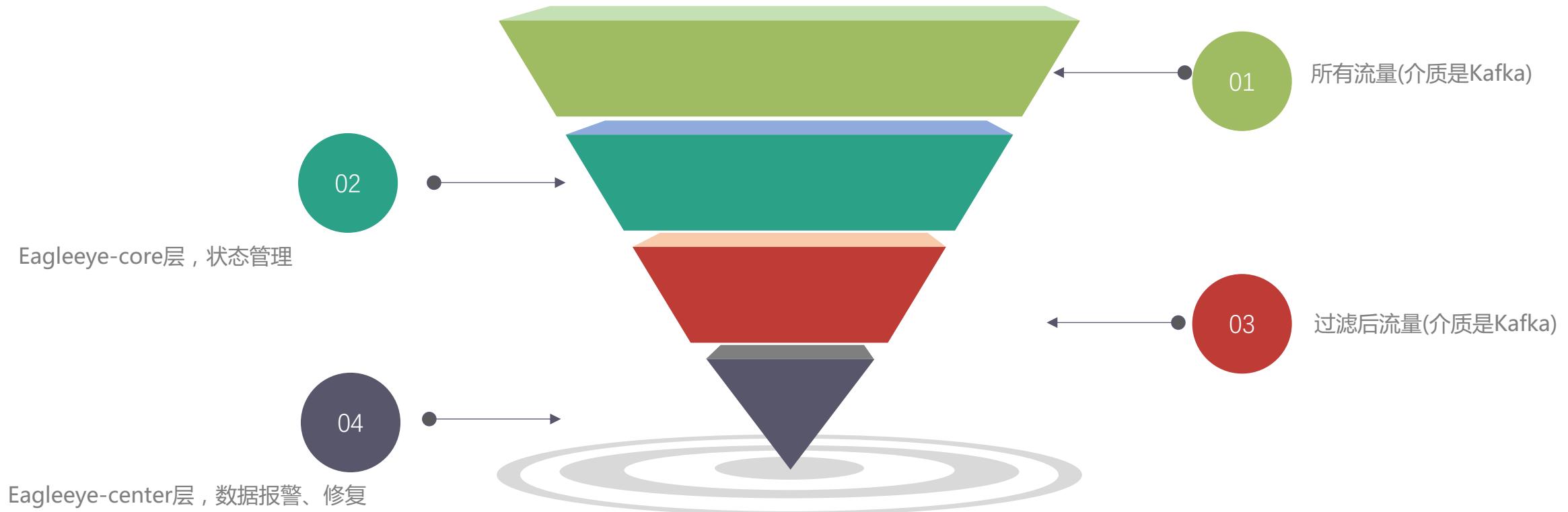
```
func CheckStatusSingleLevel(ctx context.Context, eeMonitor *EagleEyeMonitor, uniqueIds []string, createTime int64) {
    tag := "EagleEyeMonitor.CheckStatus"
    // 时间轮相关超时标记处理 ... ...

    // Monitor监控主营业务
    for _, eeStatus := range statusList {
        switch eeStatus.Status {
        case proto.ACTIVE:
            if eeStatus.Final >= eeStatus.Expect {
                if eeStatus.Final >= proto.EE_POINT_DATABASE_CMD_SUCCESS {
                    continue
                }
                timerList = append(timerList, eeStatus.UniqueID) //add timer again
                eeStatus.Expect = proto.EE_POINT_DATABASE_CMD_SUCCESS
            } else {
                eeStatus.Status = proto.WRONG //modify status cache
                repairList = append(repairList, *eeStatus) //send to repair topic
            }
            cacheList = append(cacheList, eeStatus)
        default:
            //do nothing.
        }
    }

    if len(timerList) > 0 {    //reset timer, monitor next point
        eeMonitor.AddMonitorUniqs(ctx, timerList)
    }

    //modify status of EESatus in cache ...
    //send EEData to repair topic ...
}
```

架构分层



Arris加入数据监控后的指标

每天保障数据一致性

10亿

数据稳定性

99.9%

报警成功率

100%

数据准确率

100%

数据健康度

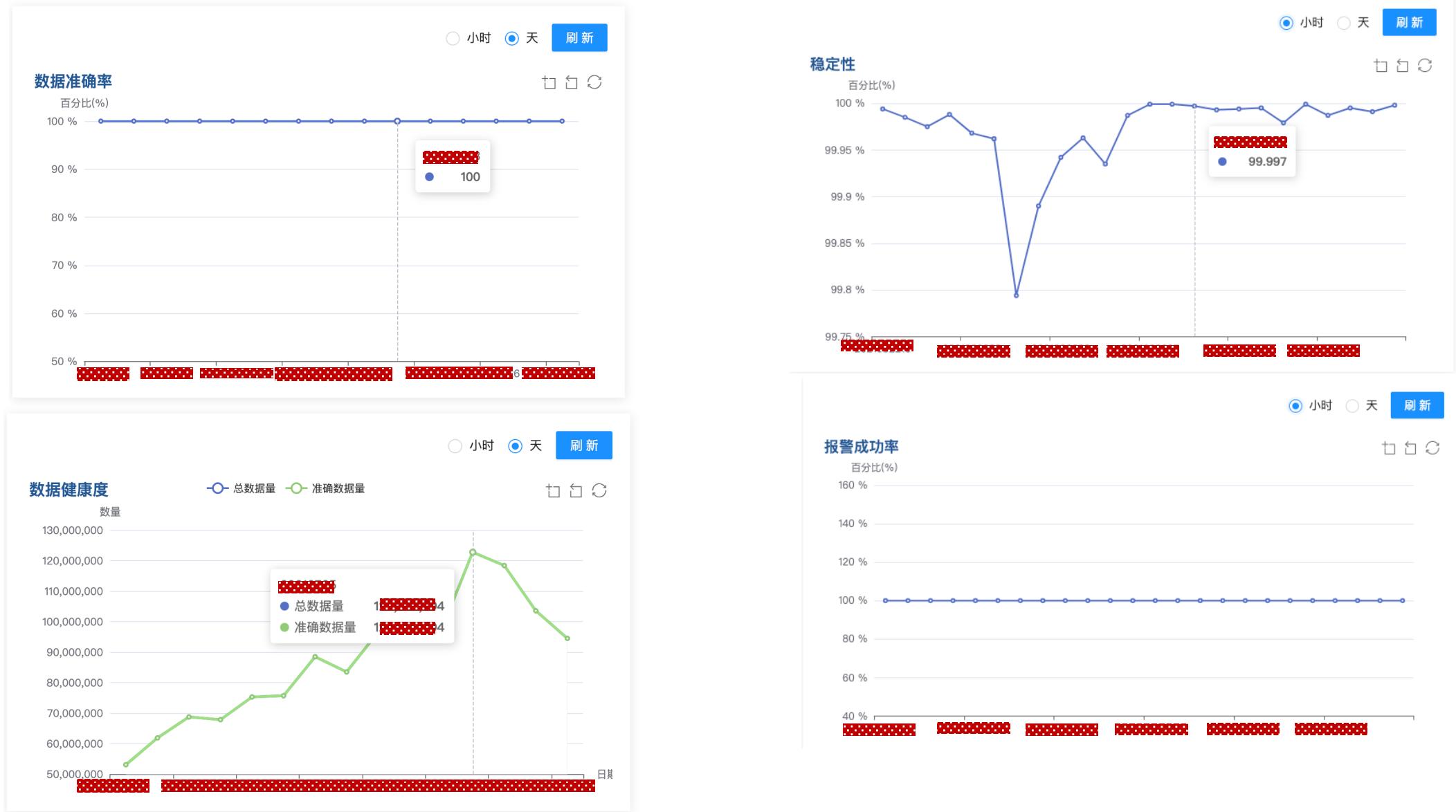
100%

数据实时修复成功率

99.99%



Arris数据监控系统-鹰眼-数据指标



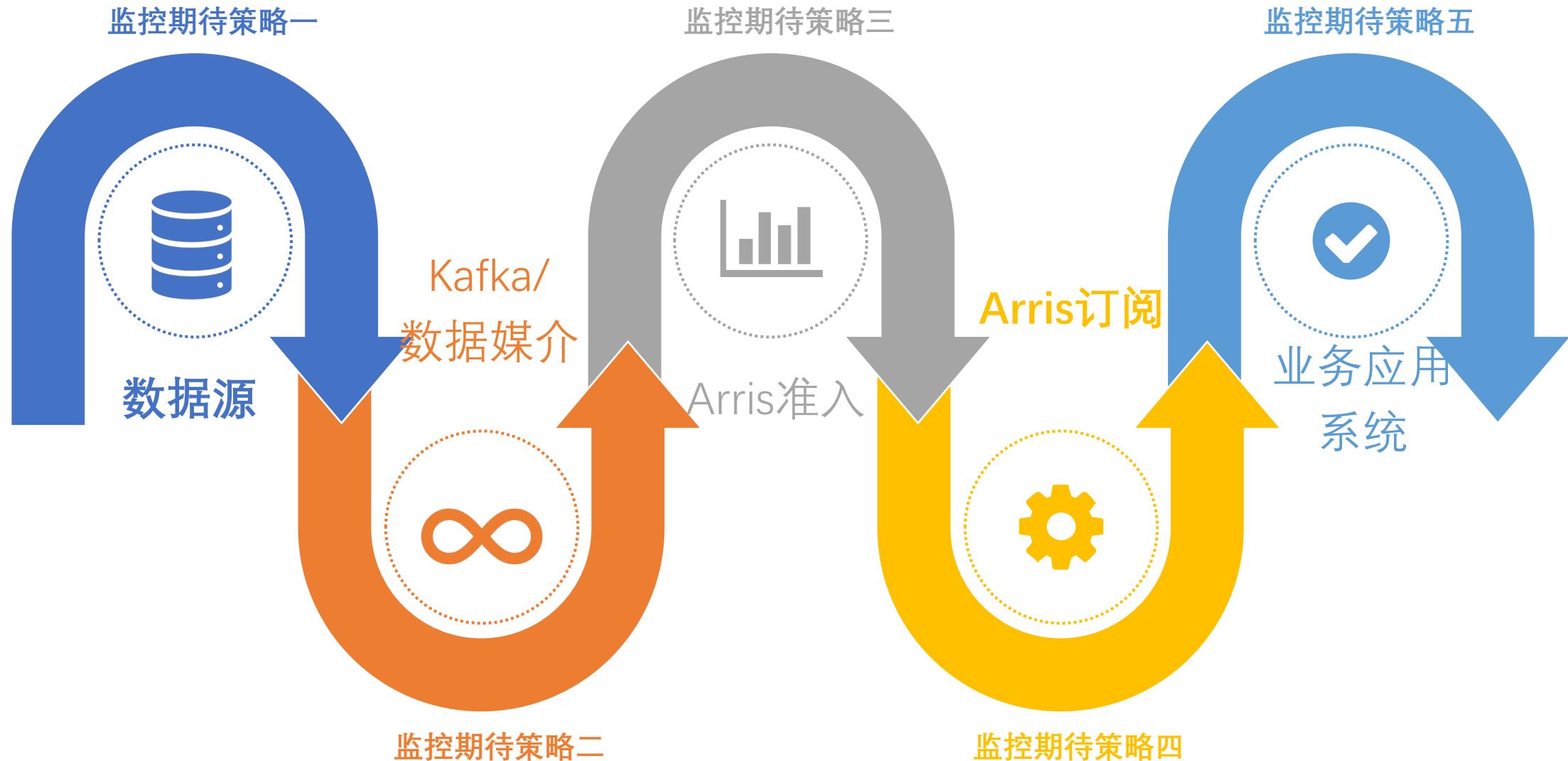
Arris数据监控系统-鹰眼-数据指标



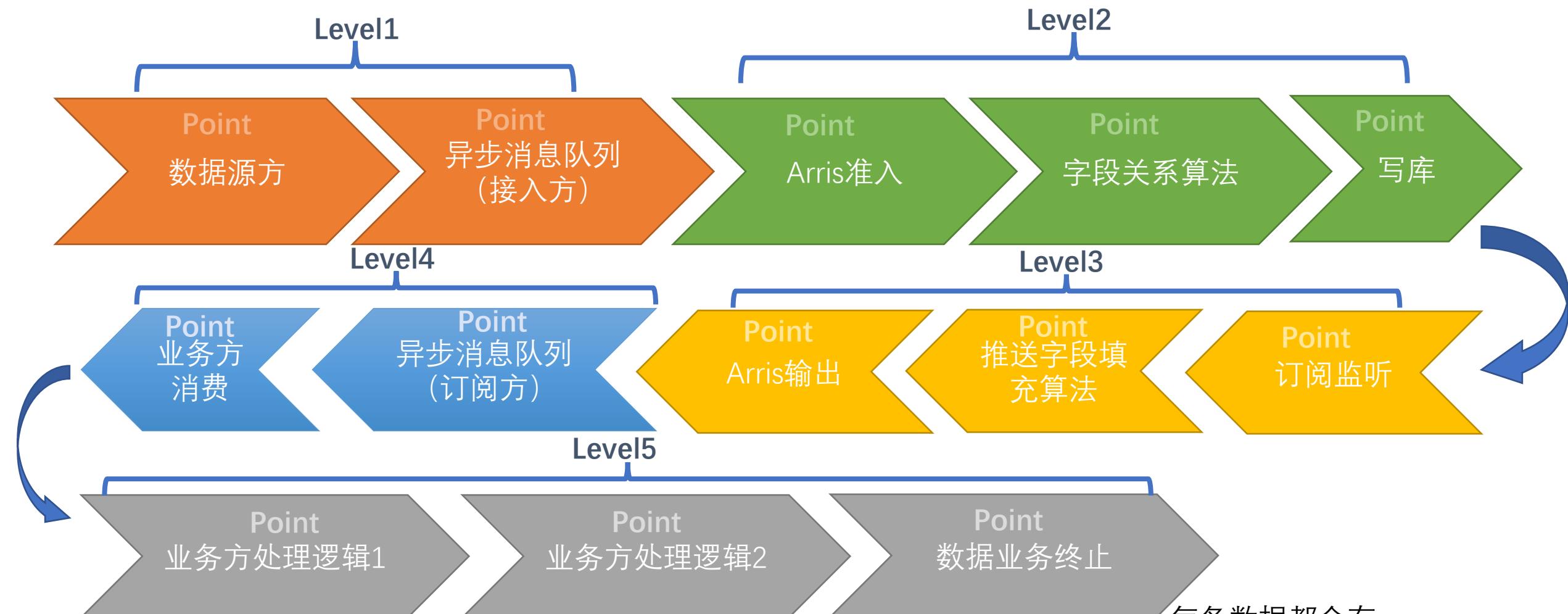
鹰眼 年 月月报

	本期数据	月环比增长
总数据量	<u>234567890</u>	-3.69%
数据准确率	<u>100.0%</u>	0.0%
系统稳定性	<u>99.96%</u>	0.26%
正确数据量	<u>234567890</u>	-3.69%
问题数据量	<u>31</u>	100%
报警成功率	<u>100%</u>	0.0%
总故障数量	<u>38343</u>	-90.08%

数据历经轨迹



一条数据在Arris系统的详细流动轨迹

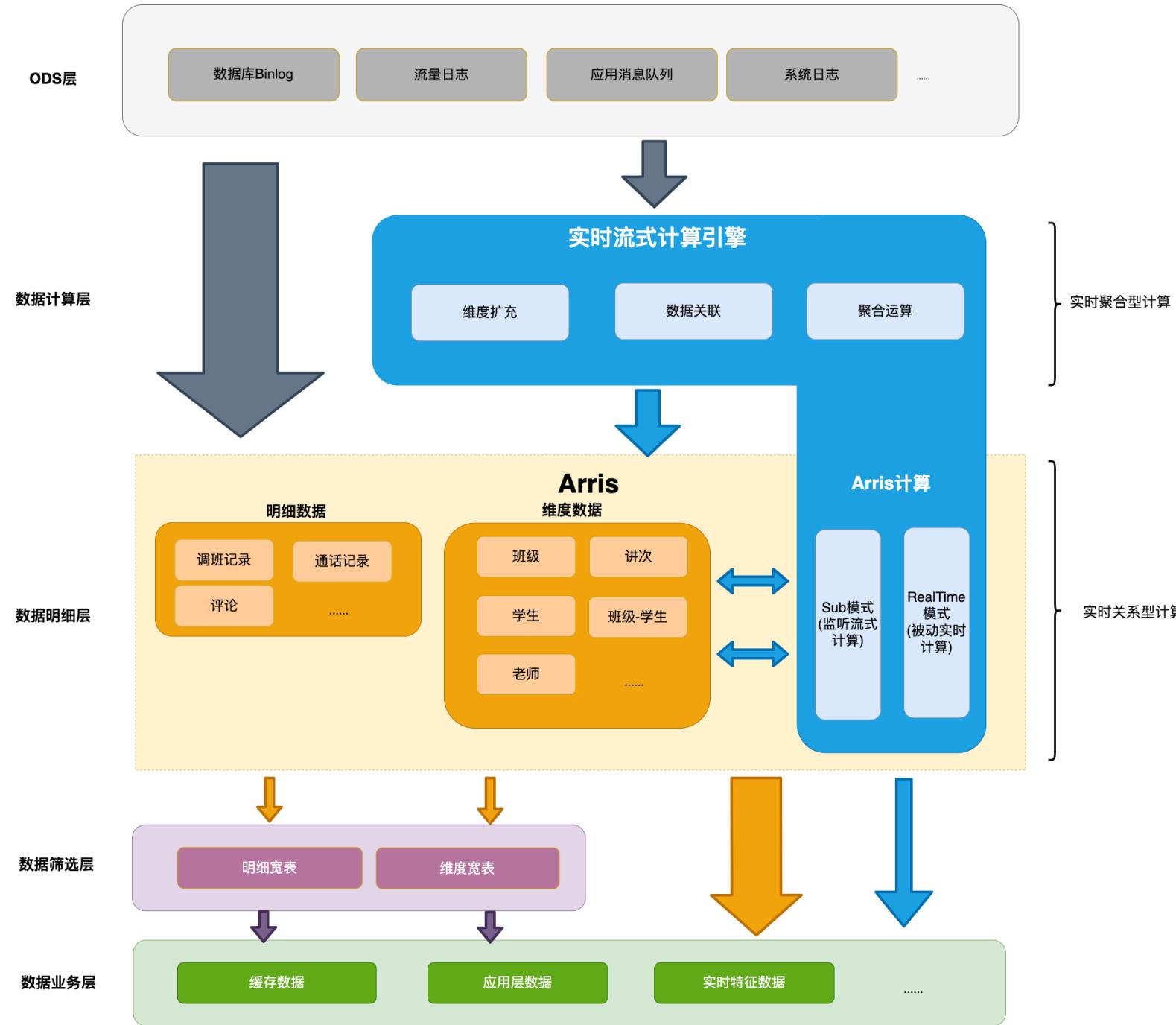


每条数据都会有
唯一的数据ID和路径TraceID

第三部分

流式计算框架 NsFlow

Arris的计算层



NsFlow 流式实时计算

名字含义为：

“*The Stream Never Stops Flowing*”

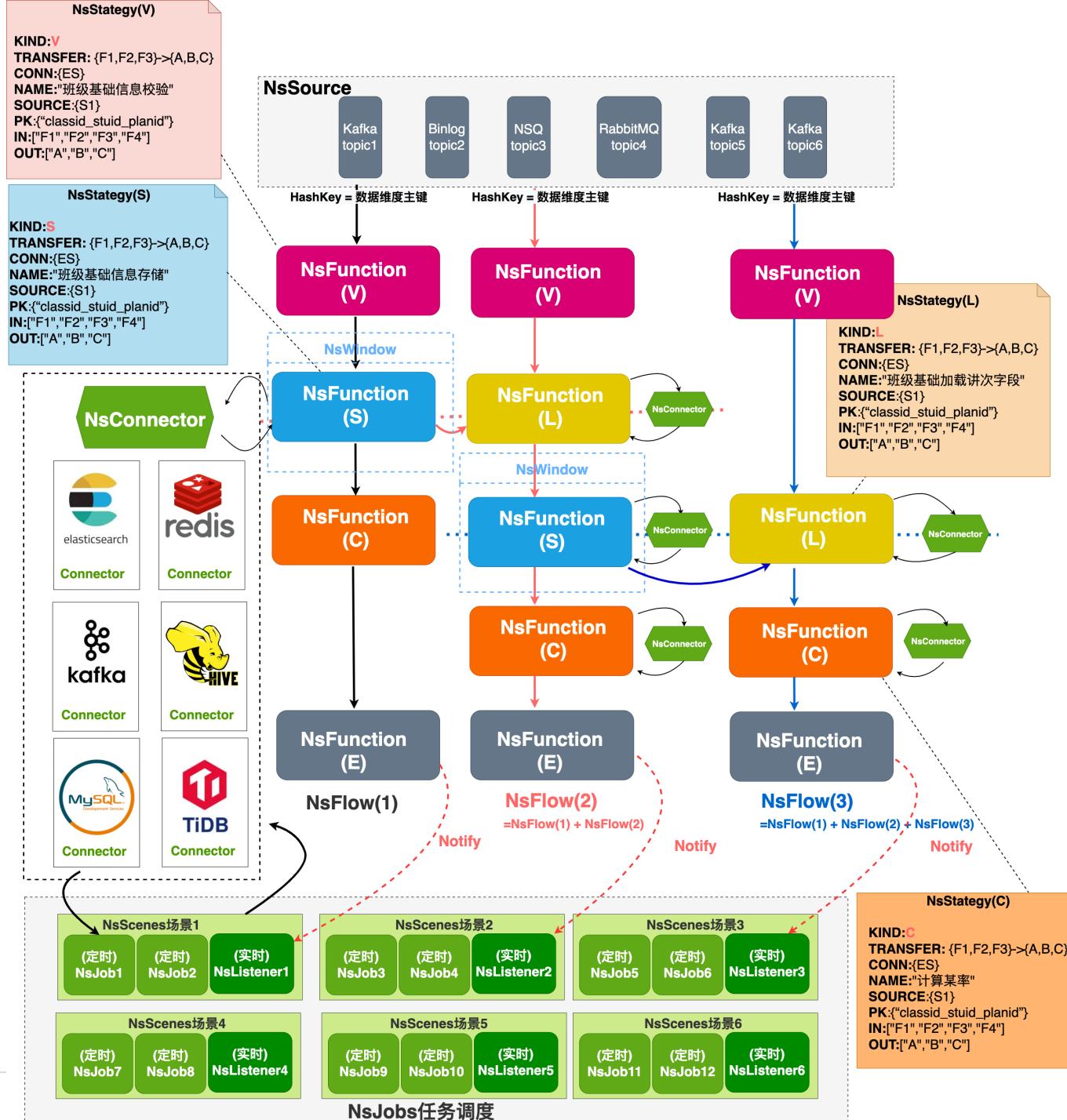
形如源源不断的数据流可以自由的匹配，永不停止的计算。

NsFlow系统定位为业务数据流上游计算层，上层直接对接业务数据提供方，如用户中心、业务层、云端、中台或ODS层，下游接数据中心Arris或其他数据中心。

NsFlow定位为一个流式实时计算SDK，接入NsFlow将更加轻量。



NsFlow 架构



流	组成
NsFlow(1)	NsFunction(V) + NsFunction(S) + NsFunction(C) + NsFunction(E)
NsFlow(2)	NsFunction(V) + NsFunction(L) + NsFunction(S) + NsFunction(C) + NsFunction(E)
NsFlow(3)	NsFunction(V) + NsFunction(L) + NsFunction(C) + NsFunction(E)



NsFunction策略协议

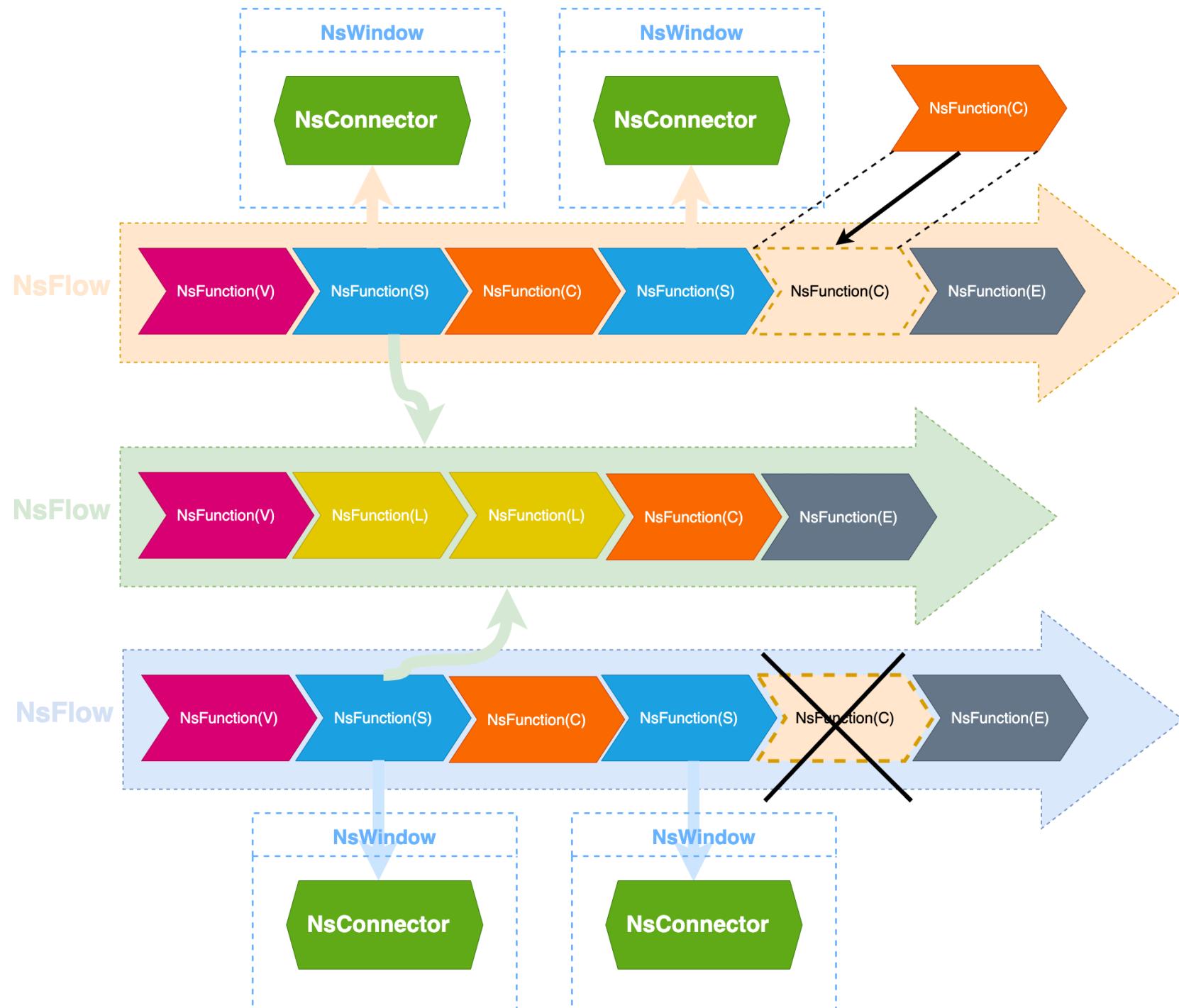
```
{  
    "fid": "NsFunc-20221106123011",  
    "fname": "测试流1",  
    "fmode": "S",  
    "transfer": [  
        {  
            "from_name": "ClassID",  
            "to_name": "class_id",  
            "from_type": "int",  
            "to_type": "string"  
        },  
        {  
            "from_name": "ClassName",  
            "to_name": "class_name"  
        },  
        {  
            "from_name": "class_store",  
            "from_type": "int",  
            "to_type": "float"  
        }  
    ],  
    "source": {  
        "name": "供应链班级基础数据",  
        "pk": ["class_id", "stu_id", "span_id"]  
    },  
    "link": {  
        "key": "ABCDEF某数据key, redis为key, mysql为表名等",  
        "pk": ["class_id", "stu_id", "span_id"]  
    },  
    "in": ["ClassID", "ClassName", "class_store"],  
    "out": ["stu_high_score"]  
}
```

fid	*标记NsFunction的唯一ID ，由前端界面配置之后自动生成。当前ID是用来作为NsFlow系统作为路由调度索引使用。
fname	*与fid功能类似，提供可读性唯一属性，不做唯一处理，可重复。
fmode	*当前NsFunction的模式类型 //V为校验特征的NsFunction, 主要进行数据的过滤，验证，字段梳理，幂等等前置数据处理 V = "Verify" //S为存储特征的NsFunction, S会通过NsConnector进行将数据进行存储，数据的临时声明周期为 NsWindow S = "Save" //L为加载特征的NsFunction, L会通过NSConnector进行数据加载，通过该Function可以从逻辑上与对应的 S Function进行并流 L = "Load" //C为计算特征的NsFunction, C会通过NsFlow中的NsData进行数据计算，生成新的字段，将数据流传递 给下游S进行存储，或者自己也已直接通过NSConnector进行存储 C = "Calculate" //E为扩展特征的NsFunction，作为流式计算的自定义特征Function，如，Notify 调度器触发任务的消息 发送，删除一些数据，重置状态等。 E = "Expand"
source	*表示当前Function的业务源
source-name	*业务源名称（开发者配置）
source-pk	*业务源相关数据主字段
transfer	(非必选) 当前NsFunction需要做的字段转换映射关系
transfer-from_name	(非必选) 源字段名称
transfer-from_type	(非必选) 源字段类型
transfer-to_name	(非必选) 目的字段名称
transfer-to_type	(非必选) 目的字段类型
in	(非必选) NsFunction数据输入字段，主要用户配置可读性，程序逻辑不做处理
out	(非必选) NsFunction数据输出字段，主要用户配置可读性，程序逻辑不做处理
link	*如果NsFunction为 "S" 或 "L" 模式，则为必选。否则为非必选
link-key	作为中间数据存储的数据标识， 如果存储媒介为redis，则为redis KEY 如果存储媒介为Mysql，则为Table 如果存储媒介为Kafka，则为Topic
link-pk	中间数据存储的数据相关逻辑主键字段

NsFlow策略协议

```
{  
    "flow_id": "NSFLOW-XXXX-2022110123023",  
    "flow_name": "班级学生基础信息统计计算数据流",  
    "steps": "4",  
    "flows": [  
        "x0x20221106123011",  
        "x1x20221106123011",  
        "x2x20221106123011",  
        "x3x20221106123011"  
    ]  
}
```

flow_id	*当前NsFlow的唯一标识（由用户做配置生成）
flow_name	*与flow_id功能类似，提供可读性唯一属性，不做唯一处理，可重复。
steps	*当前NsFlow一共拥有多少步NsFunction计算
flows	*全部有序的NsFunction



(1) 一个NsFlow可以由任意个
NsFunction组成，且NsFlow可以动
态的调整长度。

(2) 一个NsFunction可以随时动
态的加入到某个NsFlow中，且
NsFlow和NsFlow之间的关系可以通
过NsFunction的L和S节点的加入，
进行动态的并流和分流动作。

(3) NsFlow从架构用户行为上，
从面向流进行数据业务编程，变成
了面向Function的开发，接近FaaS
体系。

NsFunction数据结构

```
/*
NsFunction 流式计算基础计算模块, NsFunction是一条流式计算的基本计算逻辑单元,
任意个NsFunction可以组合成一个NsFlow
*/
type NsFunction interface {
    //执行流式计算
    Call(ctx context.Context, flow *NsFlow) error

    SetStrategy(s *NsFuncStrategy) error
    GetStrategy() *NsFuncStrategy

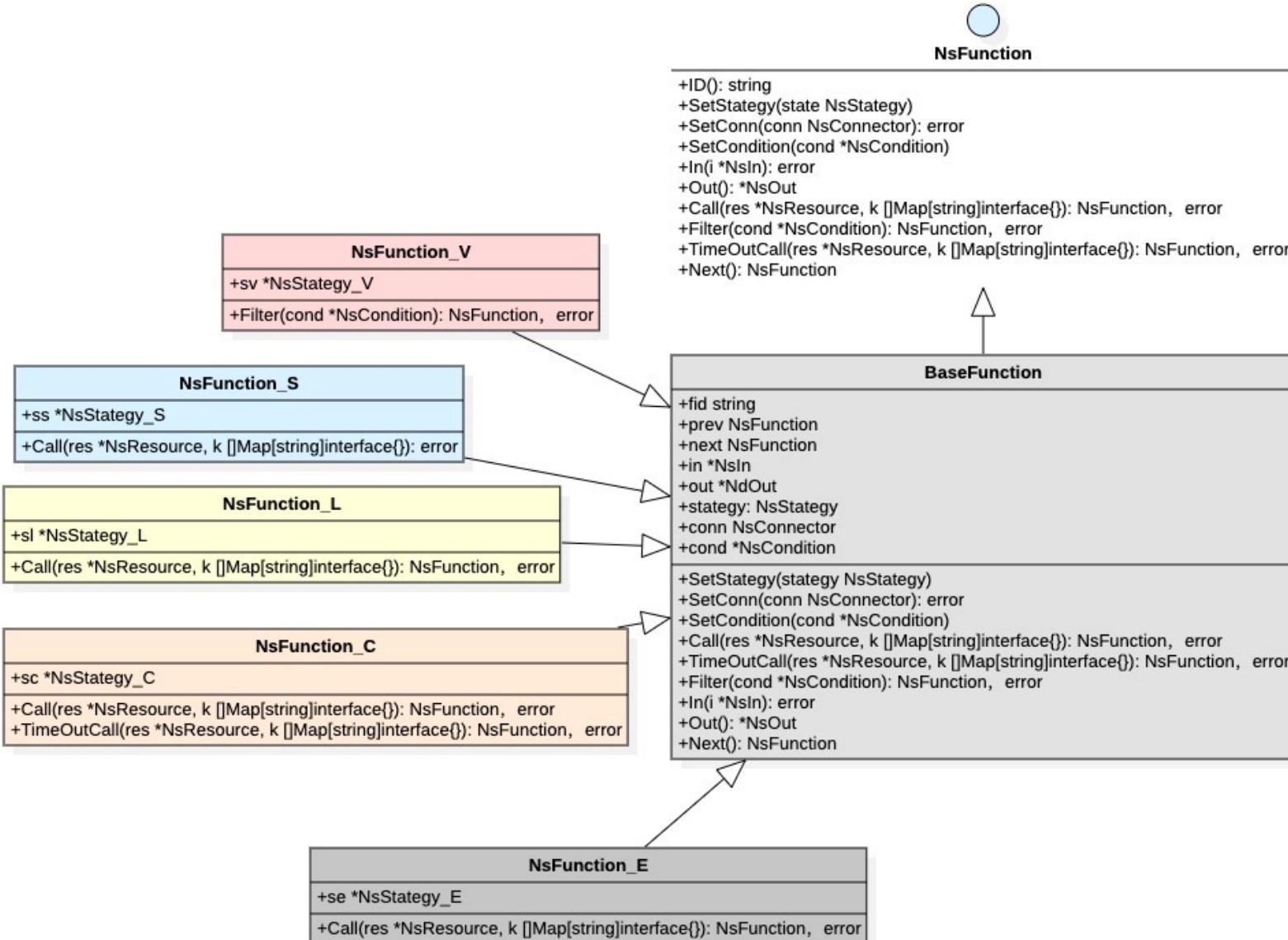
    SetFlow(f *NsFlow) error
    GetFlow() *NsFlow

    GetPrevId() string
    GetNextId() string
    GetId() string

    //返回下一层计算流Function, 如果当前层为最后一层, 则返回nil
    Next(ctx context.Context) NsFunction
    Prev(ctx context.Context) NsFunction
    SetN(f NsFunction)
    SetP(f NsFunction)

    GetConnId() string
    SetConnId(string)
    CreateNsId(int)
    GetNsId() string
}
```

NsFunction族



每个NsFunction代表一个基本的流式计算最小单元，NsFunction为双向链表结构，每个NsFunction可以任意组合，形成一个多次计算的流式计算。

NsPool

```
var _poolOnce sync.Once  
  
//FaaS Function as a Service  
type FaaS func(context.Context, *NsFlow) error  
  
//NsFuncRouter  
//key: NsFunction FID  
//value: NsFunction 回调自定义业务  
type NsFuncRouter map[string]FaaS  
  
type ns_pool struct {  
    Route NsFuncRouter           //全部的Function管理路由  
    flock sync.RWMutex  
  
    CTree      NsConnTree        //全部的Connector管理结构  
    Connectors NsConnectors      //全部的NsConnector对象  
    ConnStrategys NsConnectorStrategys //全部的NsConnStrategy对象  
    clock      sync.RWMutex  
  
    CInitRouter NsConnInitRouter //全部的Connector初始化路由  
    ilock       sync.RWMutex  
  
    flows      map[string]*NsFlow //通过初始化加载, 全部的Flow对象 ,Key:flowid, Value:*NsFlow  
    flowLock  sync.RWMutex  
}
```

NsFunction 的调用与注册

```
//CallFunction 调度 Function
func (pool *ns_pool) CallFunction(ctx context.Context, flow *NsFlow, function NsFunction) error {
    fid := function.GetId()
    if f, ok := pool.Route[fid]; ok {
        //. . . .

        //获取数据
        err, batch := flow.GetCurData()
        if err != nil {
            nsLog.ErrorF(ctx, "in Call GetCurData err = %s\n", err.Error())
            return err
        }
        flow.Input = batch

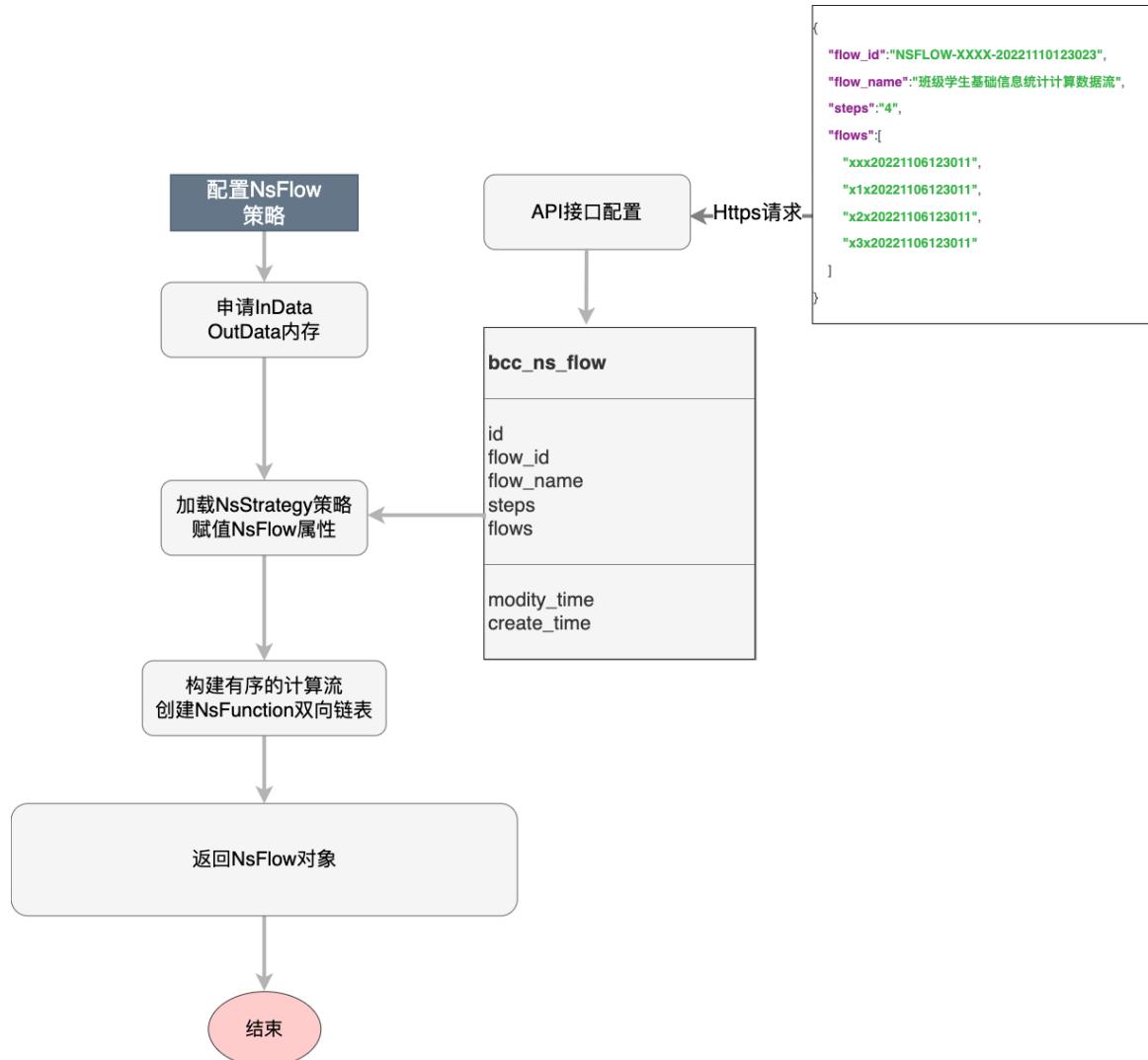
        return f(ctx, flow)
    }

    return errors.New("Fid or Fname: " + fid + " Can not find in NsPool, Not Added.")
}

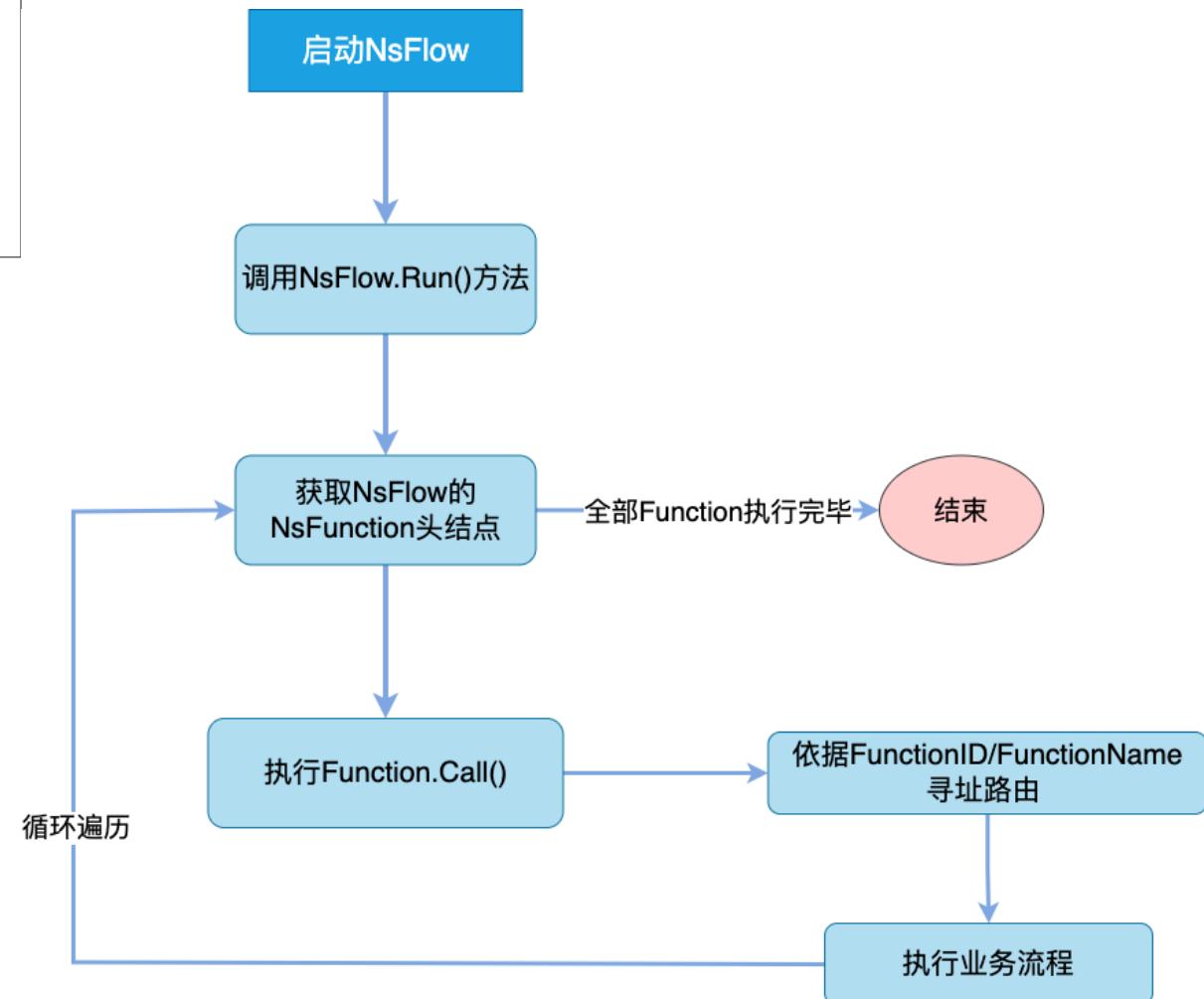
//FaaS 注册 Function
func (pool *ns_pool) FaaS(fid string, f FaaS) {
    pool.flock.Lock()
    defer pool.flock.Unlock()

    if _, ok := pool.Route[fid]; !ok {
        pool.Route[fid] = f
    } else {
        errString := fmt.Sprintf("FaaS Repeat Fid=%s\n", fid)
        panic(errString)
    }
}
```

NsFlow初始化流程



NsFlow启动流程



NsFlow数据结构

```
/*
NsFlow 用于贯穿整条流式计算的上下文环境
*/
type NsFlow struct {
    Id      string          //Flow分布式ID(平台生成管理)
    Name    string          //Flow的可读名称
    NsId    string          //Flow的分布式实例ID(NsFlow 用于NsFlow内部区分不同实例)
    Funcs   map[string]NsFunction //当前flow拥有的全部管理的全部Function对象
    FlowHead NsFunction     //当前Flow的全部Function(表头)
    FlowTail NsFunction     //当前Flow的全部Function(表尾)
    ThisFunction NsFunction //Flow当前正在执行的NSFunction对象
    flock    sync.RWMutex
    jumpFunc string         //需要跳转到指定Func继续执行Flow
    abort    bool            //是否中断Flow

    data    NsData
    Input   NsBatch
    strBuf  []string        //用来临时存放输入json源字符串的内部Buf

    connectors map[string]*NsConnector //当前Flow所依赖的Connector集合
    clock     sync.RWMutex

    cache *cache.Cache        //Flow流的临时缓存上线文环境

    funcParams map[string]FParam //flow在当前Function的自定义固定配置参数,Key:function的实例NsID, value:FParam
    flock    sync.RWMutex

    Conf  *NsFlowStrategy
    // ...
}
```



NsFlow 调用

```
func (flow *NsFlow) Run(ctx context.Context) error {
    //提交数据流源数据
    // ...
    //流式链式调用
    for function != nil && flow.abort == false {
        funcS := function.GetStrategy()

        if err := function.Call(ctx, flow); err != nil {
            return err
        } else {
            // ...
            //判断是否需要Jump到指定Function执行
            if flow.jumpFunc != common.NoJump {
                function = flow.Funcs[flow.jumpFunc]
                //更新依赖上层关系
                flow.PrevFunc = function.GetPrevId()
                continue
            }

            function = function.Next(ctx)
        }
    }

    // ...
    return nil
}
```

NsConnector 定义

```
type ConnInit func(*NsConnector) error

//ConnCall Connector的存储读取业务实现
type ConnCall func(context.Context, *NsConnector, NsFunction, *NsFlow, common.NSMode, interface{}) error

//NsConnRouter, key: NsFunctionID, value: 对应的读取存储功能
type NsConnRouter map[string]ConnCall

//NsConnSL, key: Function NSMode S/L ,value: NsConnRouter
type NsConnSL map[common.NSMode]NsConnRouter

//NsConnTree, key: NsConnector CID, value: NsConnSL 二级树
type NsConnTree map[string]NsConnSL

//NsConnInitRouter
type NsConnInitRouter map[string]ConnInit

//NsConnectors, key:NsConnectorID, value:NsConnector对象
type NsConnectors map[string]*NsConnector

//NsConnectorStrategys, key:NsConnectorID, value:NsStrategy对象
type NsConnectorStrategys map[string]*NsConnStrategy
```

/*
NsConnector 用来和NsFunction绑定的数据存储媒介
包括具体的存储引擎、存储/读取业务、链接基础信息等
*/

```
type NsConnector struct {  
    CID      string  
    CName    string  
    //配置信息  
    Conf     *NsConnStrategy  
    onceInit sync.Once  
  
    //NsConnector 自定义数据  
    mateData map[string]interface{}  
    mlock    sync.RWMutex  
}
```



NsConnector 调用与注册

```
//Call 调用Connector存储读取业务
func (conn *NsConnector) Call(ctx context.Context, flow *NsFlow, any interface{}) error {
    if err := Pool().CallConnector(ctx, flow, conn, flow.ThisFunction,
                                    common.NSMode(flow.ThisFunction.GetStrategy().Fmode), any); err != nil {
        return err
    }

    return nil
}

//CaaS 注册Connector
func (pool *ns_pool) CaaS(cid string, fid string, mode common.NSMode, c ConnCall) {
    pool.clock.Lock()
    defer pool.clock.Unlock()

    if _, ok := pool.CTree[cid]; !ok {
        //初始化各类型FunctionMode
        pool.CTree[cid][common.S] = make(NsConnRouter)
        pool.CTree[cid][common.L] = make(NsConnRouter)
    }

    if _, ok := pool.CTree[cid][mode][fid]; !ok {
        pool.CTree[cid][mode][fid] = c
    } else {
        errString := fmt.Sprintf("CaaS Repeat Cid=%s, Fid=%s, Mode=%s\n", cid, fid, mode)
        panic(errString)
    }

    nsLog.InfoF("Add NsConnector Cid=%s, Fid=%s, Mode=%s", cid, fid, mode)
}
```



业务方接入NsFlow-计算流配置化

```
└── business
    ├── duration
    │   ├── func-AggCourseStuFinishPlan.yaml
    │   ├── func-AggStuCourseDuration.yaml
    │   ├── func-AggStuDuration.yaml
    │   ├── func-AggStuPlanDuration.yaml
    │   ├── func-CalPlanStulsFinish.yaml
    │   ├── func-CalStuCourseStudyDays.yaml
    │   ├── func-CalStuPlanStudyDays.yaml
    │   └── func-CalStuStudyDays.yaml
    ├── durationv2
    │   ├── func-AggStuAgencyCoursePlanScheduleFinishPeriod.yaml
    │   ├── func-AggStuAgencyCourseScheduleFinishPeriod.yaml
    │   ├── func-AggStuAgencyCourseScheduleFinishPlan.yaml
    │   ├── func-AggStuDurationV2.yaml
    │   ├── func-AggStuScheduleAgencyCourseDuration.yaml
    │   ├── func-AggStuScheduleAgencyCoursePlanDuration.yaml
    │   ├── func-CalAgencyCoursePlanPeriodScheduleStuStudyDayNum.yaml
    │   ├── func-CalAgencyCoursePlanScheduleStuStudyDayNum.yaml
    │   ├── func-CalAgencyCourseScheduleStuStudyDayNum.yaml
    │   ├── func-CalStuPeriodIdsFinish.yaml
    │   ├── func-CalStuPeriodIdsFinishV2.yaml
    │   ├── func-CalStuPeriodPreviewlsFinish.yaml
    │   ├── func-CalStuPeriodSelfStudylsFinish.yaml
    │   ├── func-CalStuScheduleAgencyCoursePeriodStudyLastTime.yaml
    │   ├── func-CalStuStudyDayNum.yaml
    │   └── func-CheckAgencyCoursePlanFinish.yaml
    ├── ques
    │   ├── func-FillHomeWorkQuesAnswerEvent.yaml
    │   └── func-FillReviewQuesAnswerEvent.yaml
    └── common
        ├── func-Cache2Input.yaml
        ├── func-ParseArrisSubData.yaml
        ├── func-PrintInput.yaml
        ├── func-RenameField.yaml
        └── func-RepeatField.yaml
```

```
└── conn
    ├── agency_course_period_stu_schedule_duration
    │   ├── func-l
    │   │   ├── func-GetStuScheduleAgencyCourseDurationInfo.yaml
    │   ├── func-s
    │   │   ├── func-CacheStuScheduleAgencyCoursePeriodDurationInfo.yaml
    │   │   └── conn-agency_course_period_stu_schedule_duration.yaml
    ├── agency_course_periods_info
    │   ├── func-l
    │   │   ├── func-GetAgencyCoursePeriodsInfo.yaml
    │   │   └── conn-agency_course_periods_info.yaml
    ├── agency_course_plan_period_stu_schedule_duration
    │   ├── func-l
    │   │   ├── func-GetStuScheduleAgencyCoursePlanDurationInfo.yaml
    │   ├── func-s
    │   │   ├── func-CacheStuScheduleAgencyCoursePlanPeriodDurationInfo.yaml
    │   │   └── conn-agency_course_plan_period_stu_schedule_duration.yaml
    ├── arris
    │   ├── func-l
    │   │   ├── func-QueryData.yaml
    │   ├── func-s
    │   │   ├── func-PushArrisInMsg.yaml
    │   │   └── conn-arris.yaml
    ├── base_plan_info
    │   ├── func-l
    │   │   ├── func-GetPlanBaseInfo.yaml
    │   │   └── conn-base_plan_info.yaml
    ├── course_stu_duration
    │   ├── func-l
    │   │   ├── func-GetStuCourseDurationInfo.yaml
    │   ├── func-s
    │   │   ├── func-CacheStuPlanDurationInfo.yaml
    │   │   └── conn-course_stu_duration.yaml
    ├── flag_stu_duration_update
    │   ├── func-l
    │   │   ├── func-GetStudyUpdateStuListWithDate.yaml
    │   ├── func-s
    │   │   ├── func-AddStuToStudyListWithDate.yaml
    │   │   └── conn-flag_stu_duration_update.yaml
    ├── plan_stu_duration
    │   ├── func-l
    │   │   ├── func-GetStuPlanDurationInfo.yaml
    │   ├── func-s
    │   │   ├── func-CacheStuPeriodDurationInfo.yaml
    │   │   └── conn-plan_stu_duration.yaml
    └── schedule_periods
```

```
└── flow
    ├── ques
    │   ├── flow-FillHomeWorkQuesAnswerEvent.yaml
    │   └── flow-FillReviewQuesAnswerEvent.yaml
    └── v2
        ├── flow-CalAgencyCoursePlanPeriodScheduleStuStudyDayNum.yaml
        ├── flow-CalAgencyCoursePlanScheduleStuStudyDayNum.yaml
        ├── flow-CalAgencyCourseScheduleStuStudyDayNum.yaml
        ├── flow-CalStuPeriodFinishStream.yaml
        ├── flow-CalStuPeriodFinishStreamV2.yaml
        ├── flow-CalStuPeriodPreviewFinishStream.yaml
        ├── flow-CalStuPeriodSelfStudyFinishStream.yaml
        ├── flow-CalStuStudyDayNum.yaml
        ├── flow-StatisticsStuAgencyCourseScheduleFinishPeriodStream.yaml
        ├── flow-StatisticsStuAgencyCourseScheduleFinishPlanStream.yaml
        ├── flow-StatisticsStuAgencyPlanScheduleFinishPeriodStream.yaml
        ├── flow-StatisticStuAgencyCourseScheduleDurationStream.yaml
        ├── flow-CalStuPlanFinishStream.yaml
        ├── flow-StatisticsStuCourseFinishPlanStream.yaml
        ├── flow-StatisticsStuStudyDays.yaml
        └── flow-StatisticStuPlanDurationStream.yaml
    └── nsflow.yaml
```



业务方接入NsFlow-计算单元逻辑注册

```
package faas

func init() {
    flow.SetLogger(NewNsLogger())
    flow.Pool().CaaSInit("plan_stu_duration", plan_stu_duration.InitConnector)
    flow.Pool().FaaS("GetStuPlanDurationInfo", plan_stu_duration.GetStuPlanDurationInfo)
    flow.Pool().FaaS("CacheStuPeriodDurationInfo", plan_stu_duration.CacheStuPeriodDurationInfo)

    flow.Pool().CaaSInit("course_stu_duration", course_stu_duration.InitConnector)
    flow.Pool().FaaS("GetStuCourseDurationInfo", course_stu_duration.GetStuCourseDurationInfo)
    flow.Pool().FaaS("CacheStuPlanDurationInfo", course_stu_duration.CacheStuPlanDurationInfo)

    flow.Pool().CaaSInit("flag_stu_duration_update", flag_stu_duration_update.InitConnector)
    flow.Pool().FaaS("GetStudyUpdateStuListWithDate", flag_stu_duration_update.GetStudyUpdateStuListWithDate)
    flow.Pool().FaaS("AddStuToStudyListWithDate", flag_stu_duration_update.AddStuToStudyListWithDate)

    flow.Pool().CaaSInit("base_plan_info", base_plan_info.InitConnector)
    flow.Pool().FaaS("GetPlanBaseInfo", base_plan_info.GetPlanBaseInfo)

    flow.Pool().CaaSInit("agency_course_periods_info", agency_course_periods_info.InitConnector)
    flow.Pool().FaaS("GetAgencyCoursePeriodsInfo", agency_course_periods_info.GetAgencyCoursePeriodsInfo)

    flow.Pool().FaaS("ParseArrisSubData", arris.ParseArrisSubData)
    flow.Pool().FaaS("RenameField", arris.RenameField)
    flow.Pool().FaaS("CpFile", common.CpFile)

    // ... ...
}

err := flow.StrategyLoadWalker(confutil.GetConf("nsflow", "conf"))
if err != nil {
    panic(err)
}
}
```

业务方接入NsFlow-实现业务计算单元

```
func FillHomeWorkQuesAnswerEvent(ctx context.Context, nsFlow *flow.NsFlow) error {
    //获取课时信息缓存
    schPeriodsInfoCache := nsFlow.GetCacheData(schedule_periods.FlowCacheKey)

    //...
    //...
    //...

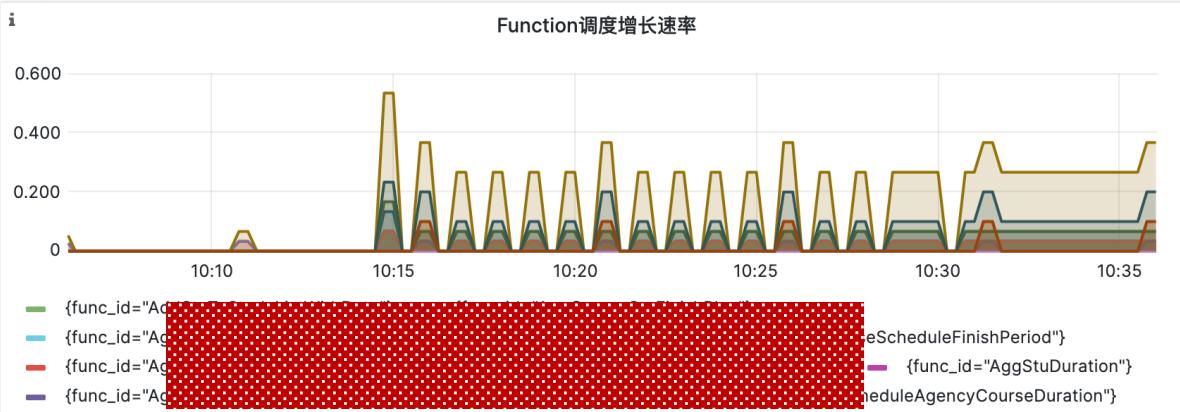
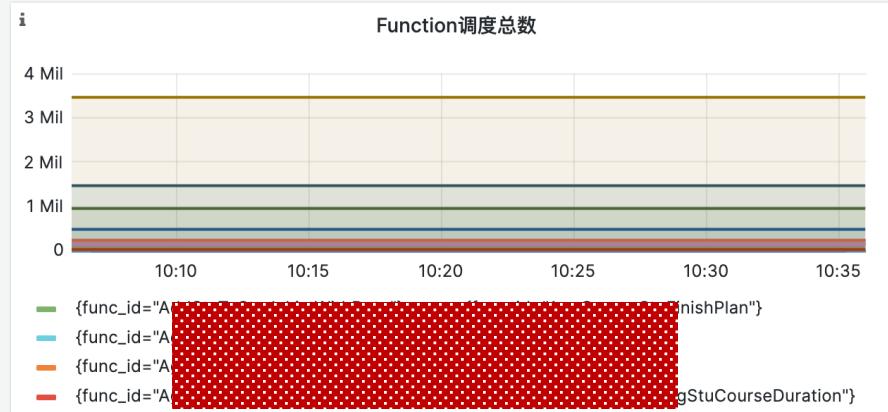
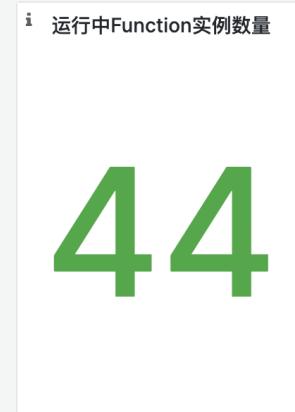
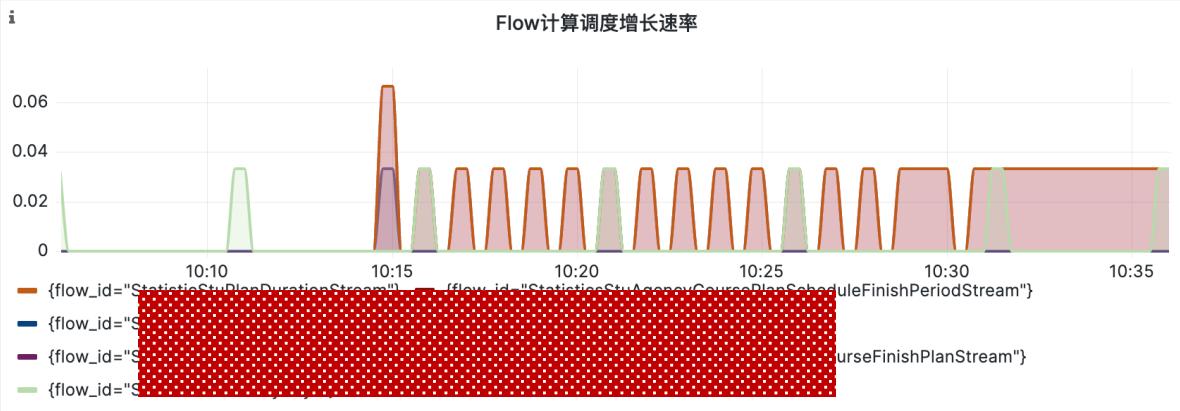
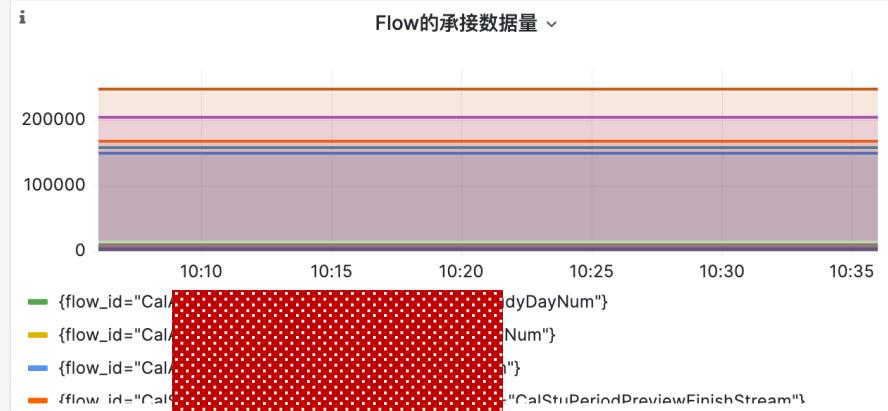
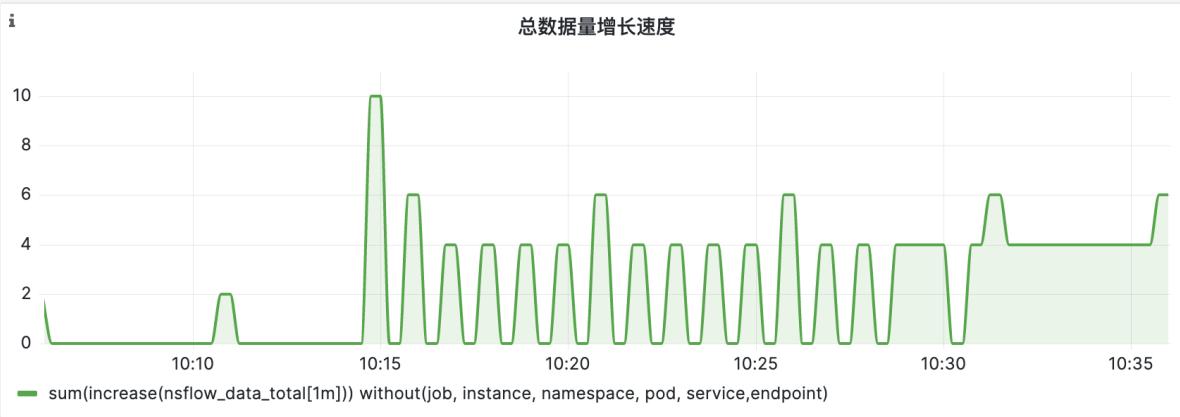
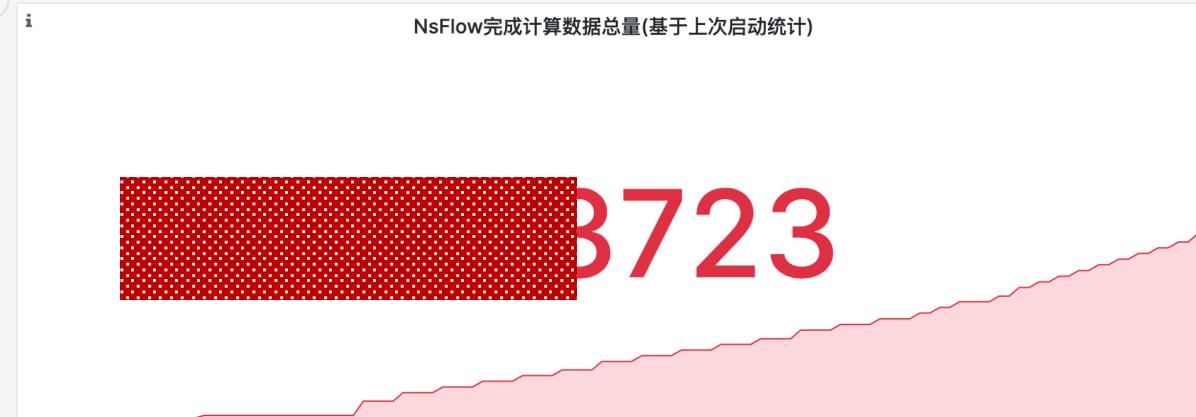
    for _, row := range nsFlow.Input {
        raw := row.Raw
        data := gjson.Get(raw, "arris_message.fields").String()
        obj := &QuesAnswerEvent{}

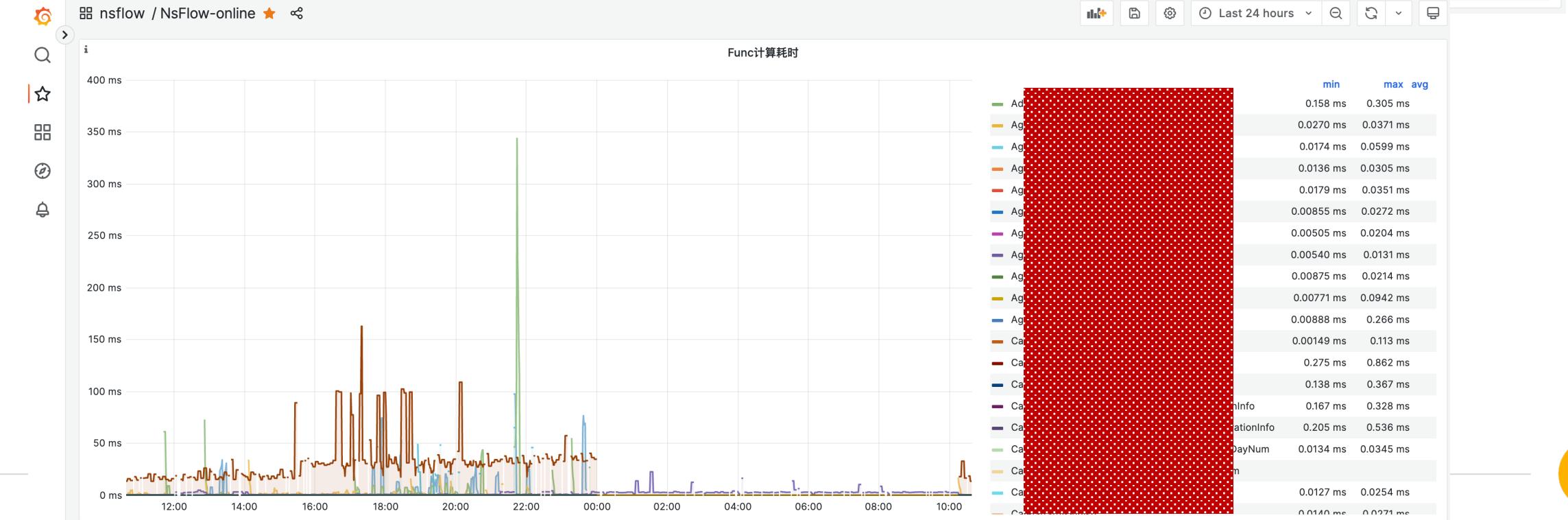
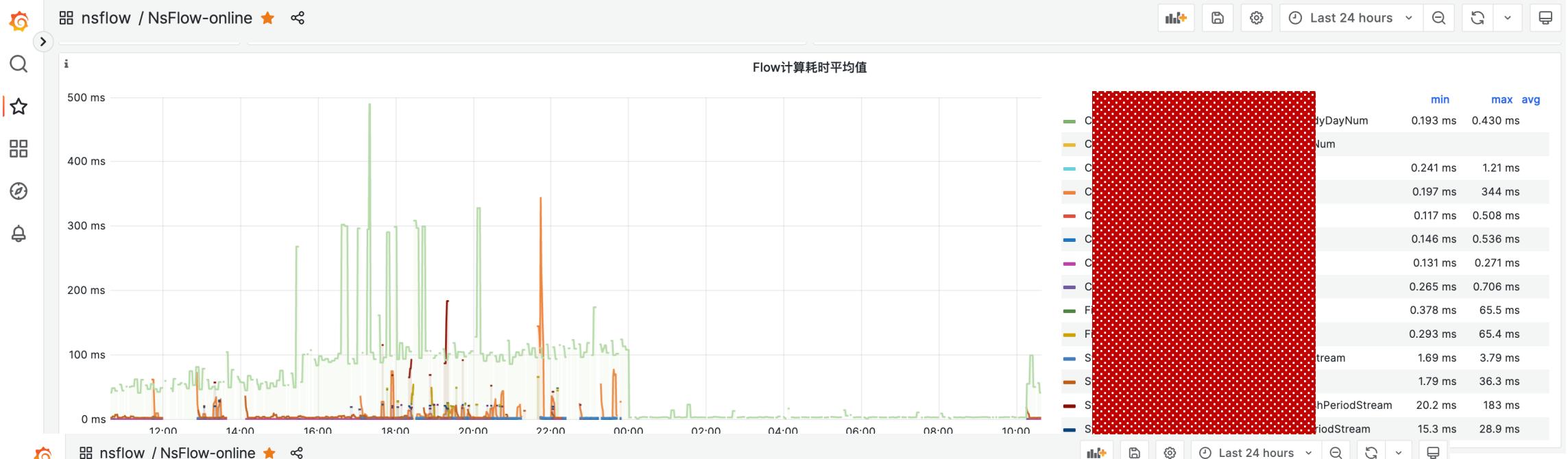
        //...
        //...
        //...

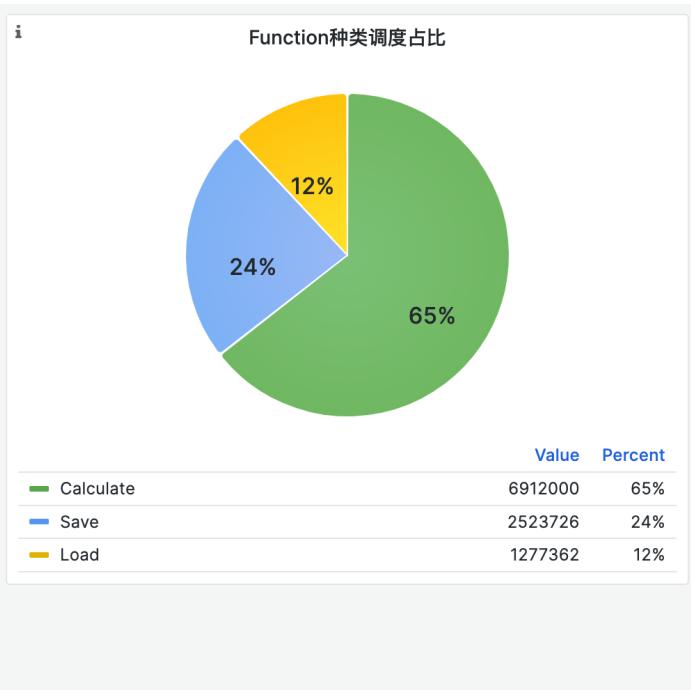
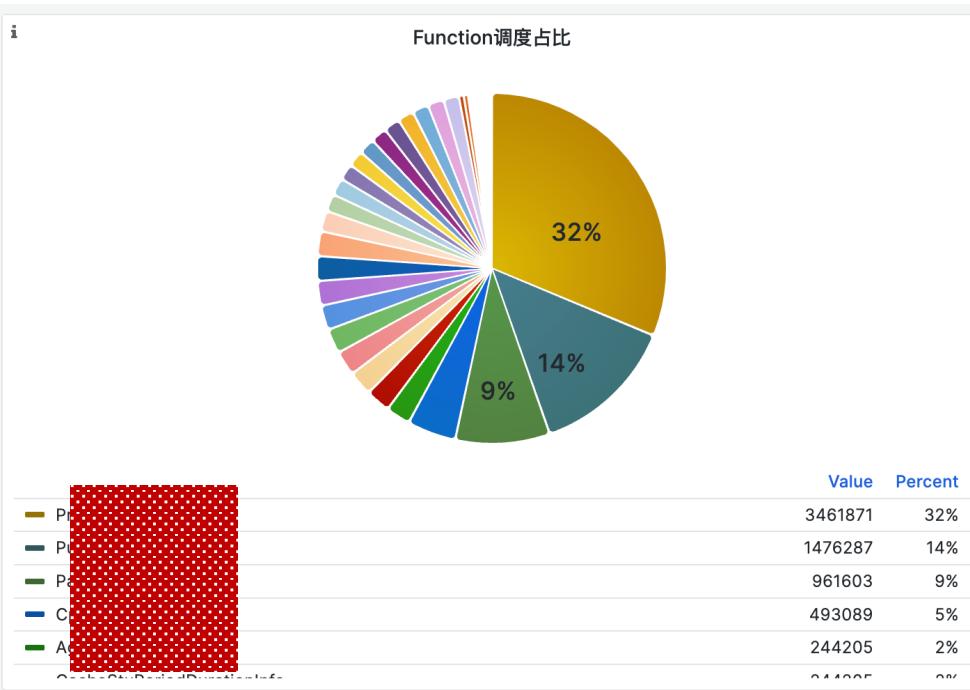
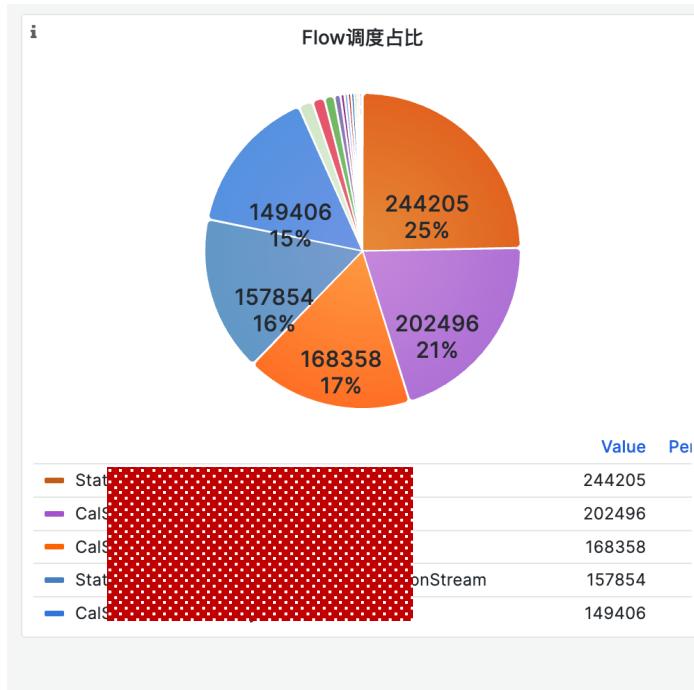
        obj.AgencyCourseId = periodInfo.AgencyCourseId
        obj.AgencyPlanId = periodInfo.AgencyPlanId

        _ = nsFlow.AppendDataStruct(obj)
    }

    return nsFlow.Next(ctx)
}
```







借GopherChina平台 感谢早期参与过Arris建设的伙伴们！

研发

刘丹冰 赵远通 刘立 胡晨阳 王贺冬 林加豪 皇甫华音
吴国福 王龙飞 杨兴亚 段路中 郭帅龙 石丹 郑小航

测试

王东洋 贾艳丽 赵碧瑜 刘同 李柏桥

产品

鲍伟 许房琛谌(大妞)



谢谢



刘丹冰 Acelld



好未来技术

