播控方案设计

lucasqian@

2022/04/09

1. 目标

- *引入规则引擎，使用 dsl 语句代替直接修改代码，提高灵活性、拓展性以及通用性*
- *提供一个规则管理台页面，降低规则配置成本，将便于开发理解的 dsl 规则转换为便于运营&产品理解的可视化规则*
- *支持动态加载规则能力，支持规则版本管理，引入接入层服务支持灰度发布能力*
- *支持播控规则自定义维度的统计能力和 trace 查询能力*
- *服务全链路保证高性能和高可用*

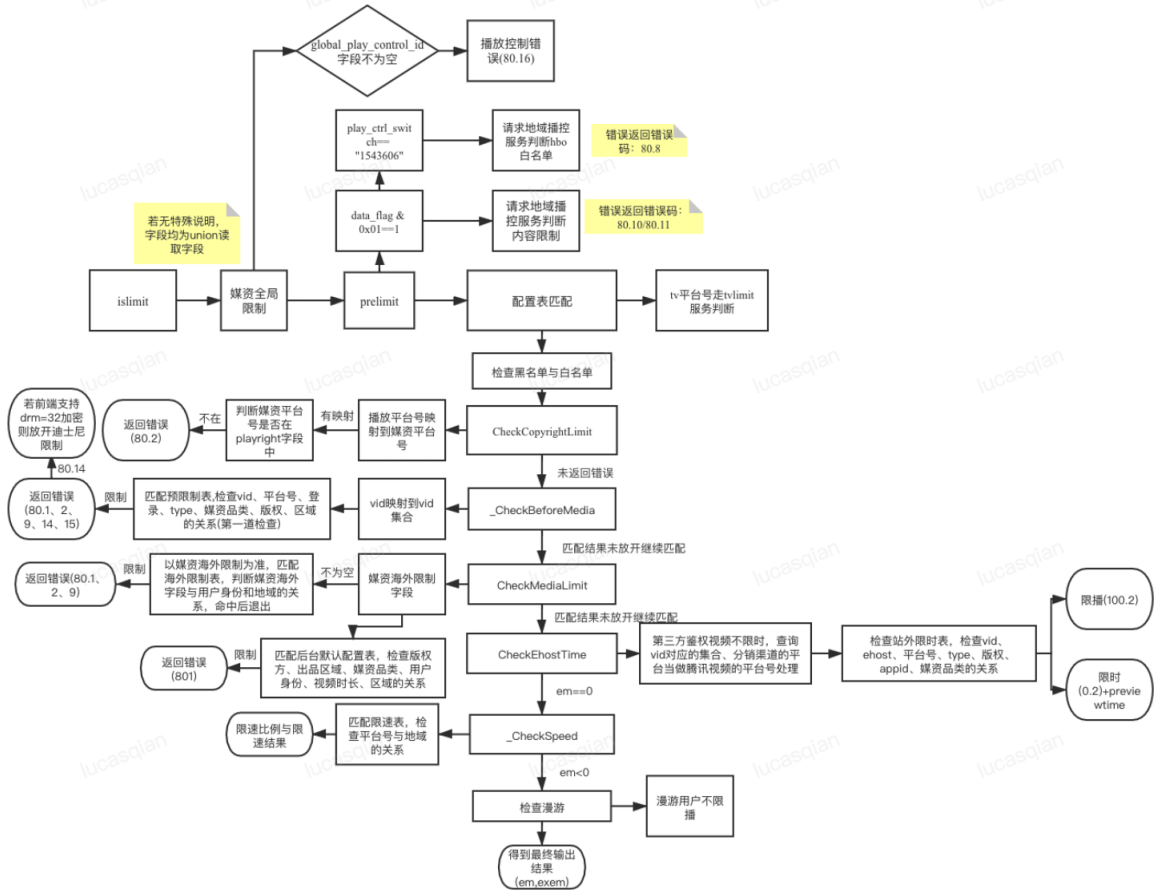
2. 背景

2.1. 旧播控服务介绍

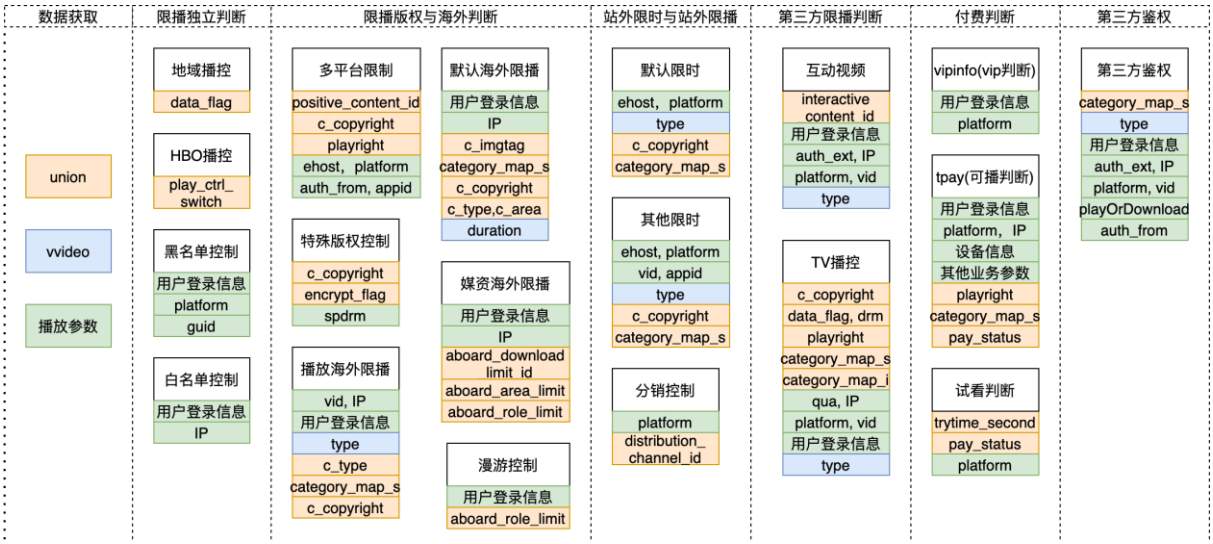
点播播控策略介绍

为了方便后面章节的方案介绍，抽取出最主要的两部分如下：

- *旧播控全流程：*



- *规则判断涉及数据字段：*



2.2. 旧播控服务的问题

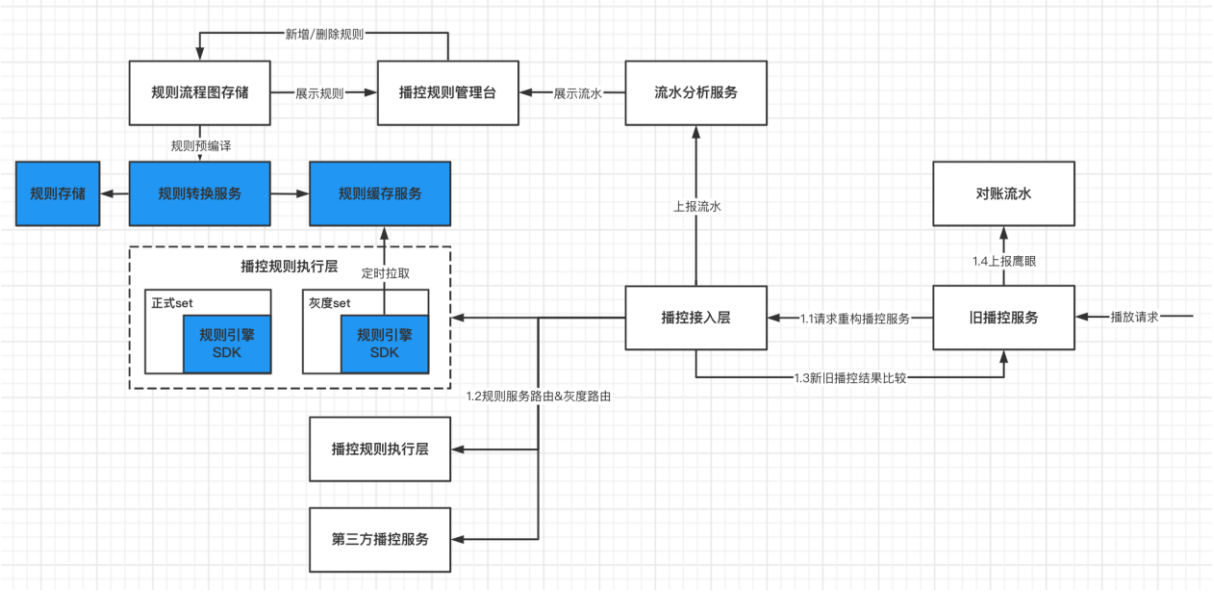
- *规则较为定制化，使用代码串联起各个过程，拓展性和通用性不强*
- *规则分散在多张无极配置表中，没有一个完整的规则呈现，理解成本高*

- *新增/修改策略无法得知对整个播控系统造成多少影响*
- *对规则缺乏版本管理&灰度发布的能力*
- *服务扩容困难，且仅进行了单地部署，可靠性较差*

3. 总体设计

3.1. 架构图

系统总体架构图如下：



3.2. 各模块介绍

- ***播控接入层***
  - *负责将不同业务播控请求路由到不同的规则执行层服务*
  - *控制每条播控规则的灰度逻辑*
  - *上报播控业务关键指标用于数据分析*
- ***播控规则执行层***

  基于规则引擎针对每条请求进行播控判断，系统的核心模块
- ***规则引擎 SDK***

  对规则引擎进行封装，负责规则加载、规则执行、规则定时拉取、规则内部流水上报工作

- ***规则转换服务***
  - *负责将前端的可视化规则流程数据转换为 dsl 语句*
  - *负责规则预编译*
  - *负责规则多版本存储*
- ***规则缓存服务***

  负责缓存规则编译数据提高性能，屏蔽规则底层存储实现，降低 SDK 耦合存储实现
- ***播控规则管理台***
  - *新增/删除/修改规则*
  - *规范规则发布流程（正式环境/灰度环境）*
  - *支持展示播控服务各维度统计信息*
  - *支持查询每条播控请求的 trace 信息*
- ***流水分析服务***

  基于播控接入层上报流水进行实时数据统计并生成数据报表

4. 详细设计

4.1. 规则引擎选型

策略引擎	*底层算法*	*性能（1000 条规则）*	*扩展性*	*规则预编译*	*规则热加载*
gRule	*rete 算法*	*400μs*	*仅支持顺序模式*	*支持*	*不支持*

gEngine	ast 语法解析	1ms	支持顺序模式、并发模式、混合模式,以及其他细分执行模式	不支持（二次开发版本支持）	支持

- 这两个规则引擎库项目当前均比较活跃，并且都有详细的中文文档可以参考
- 由于 gRule 底层使用了 rete 算法，串行性能上比纯粹使用 ast 语法树解析的 gEngine 好一些，但是 gEngine 提供了引擎池的能力，在高并发量的场景下可以支持较好的吞吐量
- 两个规则引擎库均使用 dsl 语句来提供扩展性，相比之下 gEngine 的 dsl 语法更接近 golang，对熟悉 golang 的开发上手比较友好；执行模式方面，gRule 仅支持规则间顺序执行，需要特殊方式支持其他执行模式；gEngine 则支持了丰富的执行模式，比较利于后续程序功能的拓展
- 部门内基于 gEngine 进行了二次开发，进行了性能优化以及功能扩展

综上所述，建议使用 gEngine 作为底层规则引擎库，它在满足性能要求的前提下，提供了更好的扩展性以及更低的上手难度

4.2. 规则匹配依赖数据

目前规则匹配的数据来源主要有三个：

1）union

2）vvideo

3）播放参数

从性能方面考虑，建议一次性从三个来源拉取所有需要的数据，然后放在一个数据结构体内用于规则匹配，未来规划在引擎层支持数据配置拉取，提高整个服务的扩展性

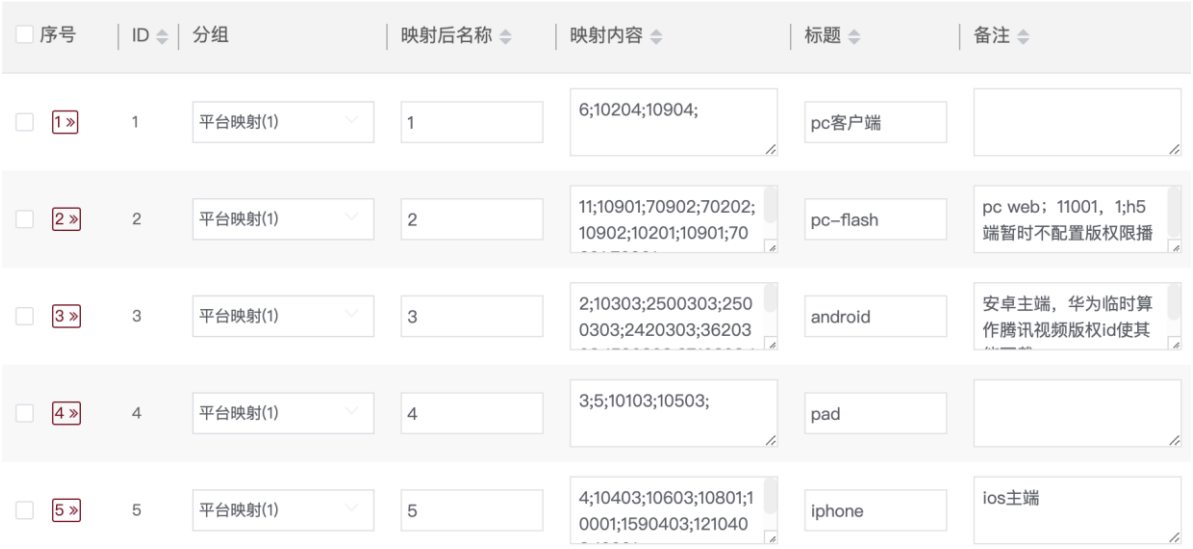
4.3. 新旧规则转换策略

4.3.1. 旧规则介绍

旧规则主要包括以下两种配置表：

1）字典表（一张，分组配置不同映射）：用于描述规则的某一维度具体值域（平台映射）

例如图中第一行表示的是 pc 客户端的 key 值为「1」，对应的平台 ID 有「6、10204、10904」



序号	ID	分组	映射后名称	映射内容	标题	备注
1	1	平台映射(1)	1	6;10204;10904;	pc客户端	
2	2	平台映射(1)	2	11;10901;70902;70202;10902;10201;10901;70	pc-flash	pc web；11001，1;h5端暂时不配置版权限播
3	3	平台映射(1)	3	2;10303;2500303;2500303;2420303;36203	android	安卓主端，华为临时算作腾讯视频版权id使其
4	4	平台映射(1)	4	3;5;10103;10503;	pad	
5	5	平台映射(1)	5	4;10403;10603;10801;10001;1590403;121040	iphone	ios主端

2）具体规则配置表（4 张）：每个维度支持的操作（字符串完全匹配、数值范围匹配、多值 in）

例如图中第一行表示的是平台号为「200603」的命中播控规则

序号	id	vid集合映射	平台号	vip	type集合映射	媒资一级分类	媒资二级分类	媒资三级分类	区域
1	2	all(all)	200603	all(all)	all(all)	all(all)	all	all	all
2	3	all(all)	1900603	all(all)	all(all)	all(all)	all	all	all
3	4	all(all)	11;10202;70202;10902;70902;820202;103	all(all)	all(all)	体育(4)	奥运	比赛点播	all
4	5	all(all)	all	all(all)	all(all)	体育(4)	奥运	比赛点播	all

4.3.2. 规则引擎介绍

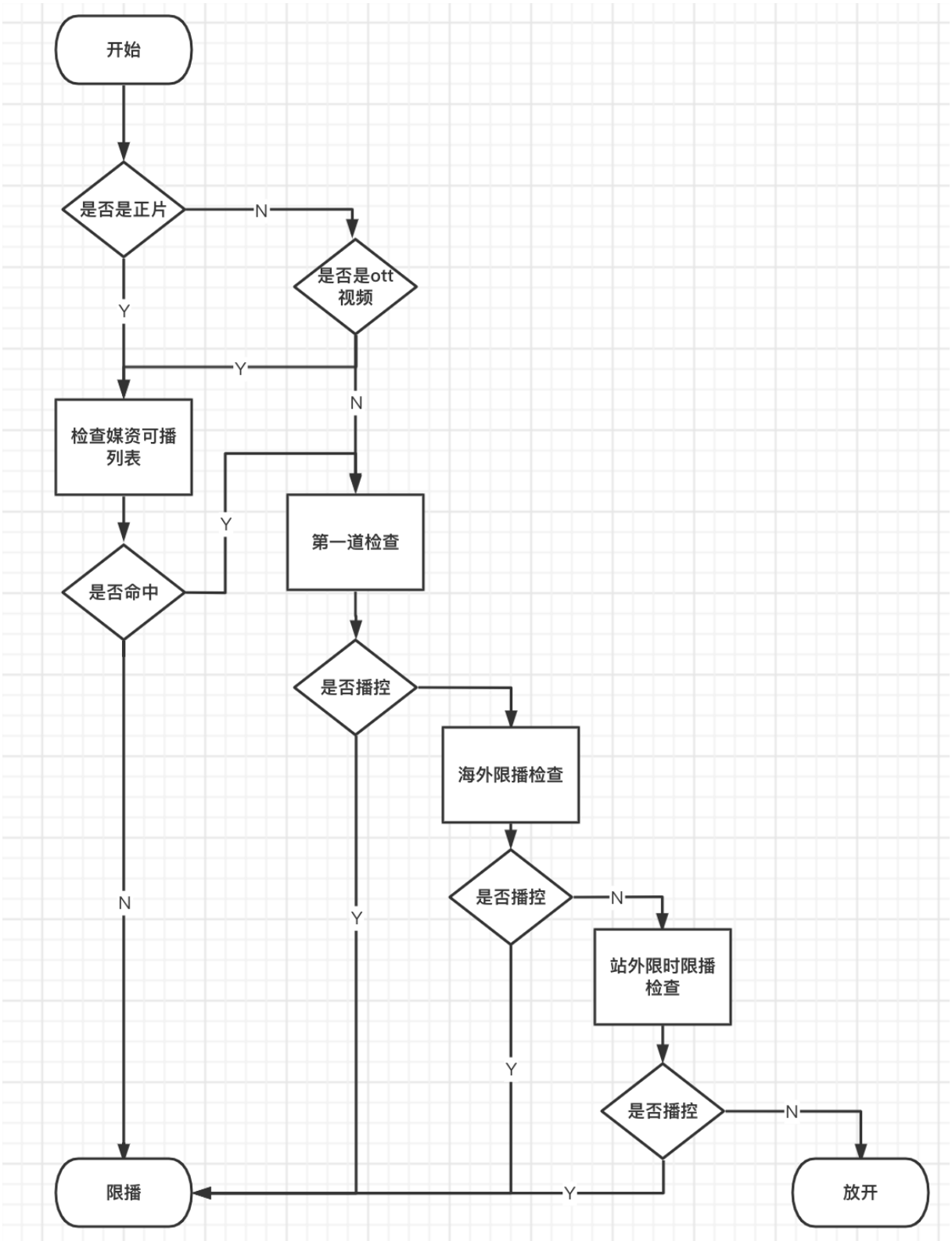
部门内的规则引擎在 gEngine 基础上进行二次开发，提供了可视化逻辑编排的能力，以逻辑

原子的形式支持业务抽象建模

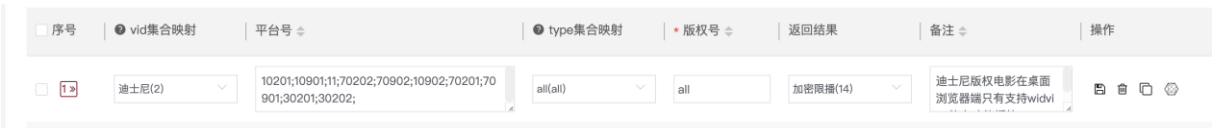
详细设计文档如下：规则引擎 - 设计文档

4.3.3. 新规则的转换策略

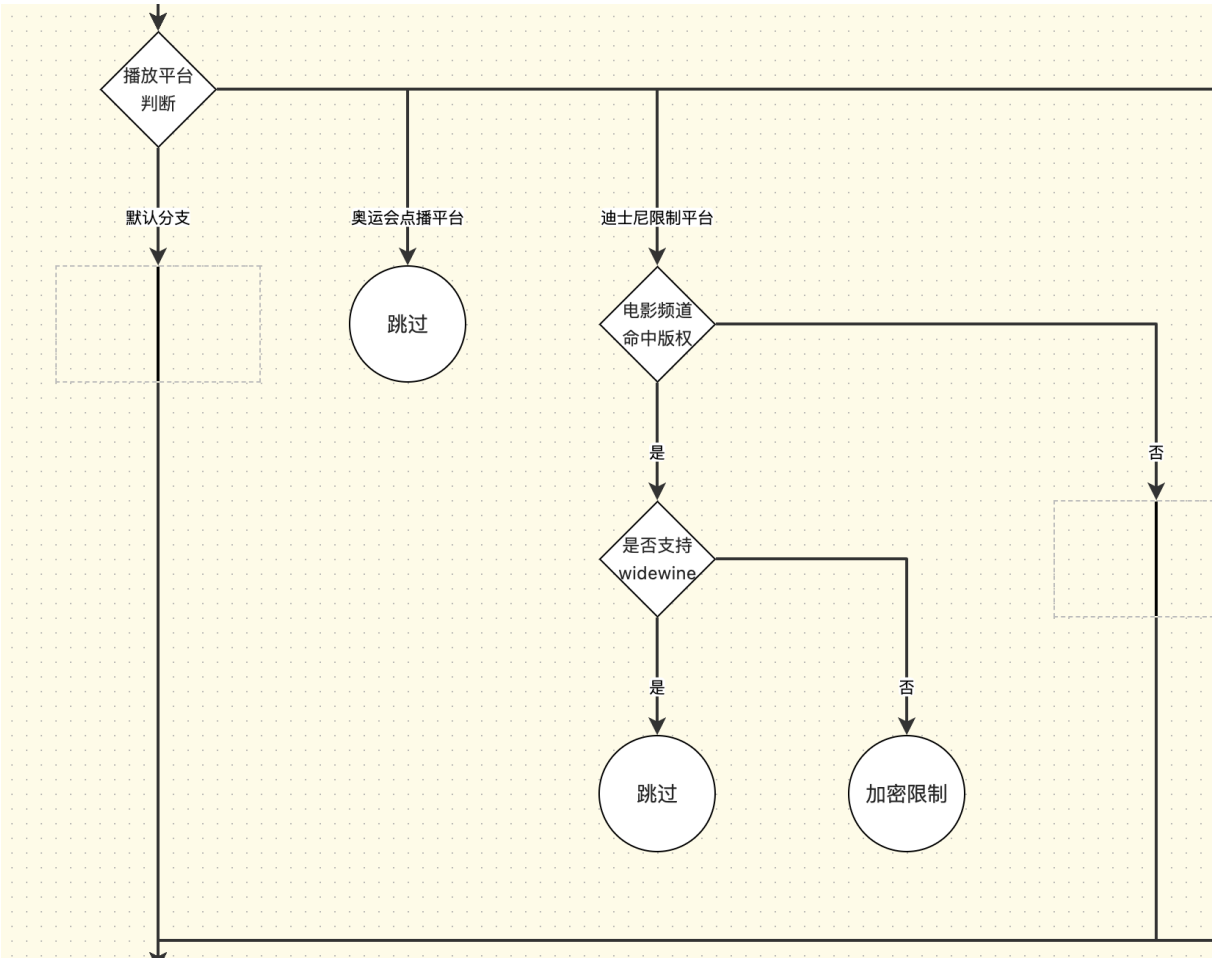
- *基于章节 4.3.2 规则引擎对旧规则重新进行逻辑编排*
- *为每个旧规则无极表建立子逻辑节点（例如第一道检查对应无极表 checkBeforeMedia），建立一个整体的播控逻辑流程，如下图所示：*



- *对原配置表内的每行规则按照树状图的方式配置逻辑分支，下面以 checkBeforeMedia 配置表为例：*



对应的逻辑树状流程图如下所示：



4.4. 新规则可视化方案

基于原有规则引擎可视化方案实现，详细设计方案如下：通用逻辑可视化编排引擎

4.5. 规则灰度方案

4.5.1. 单 set 部署方案

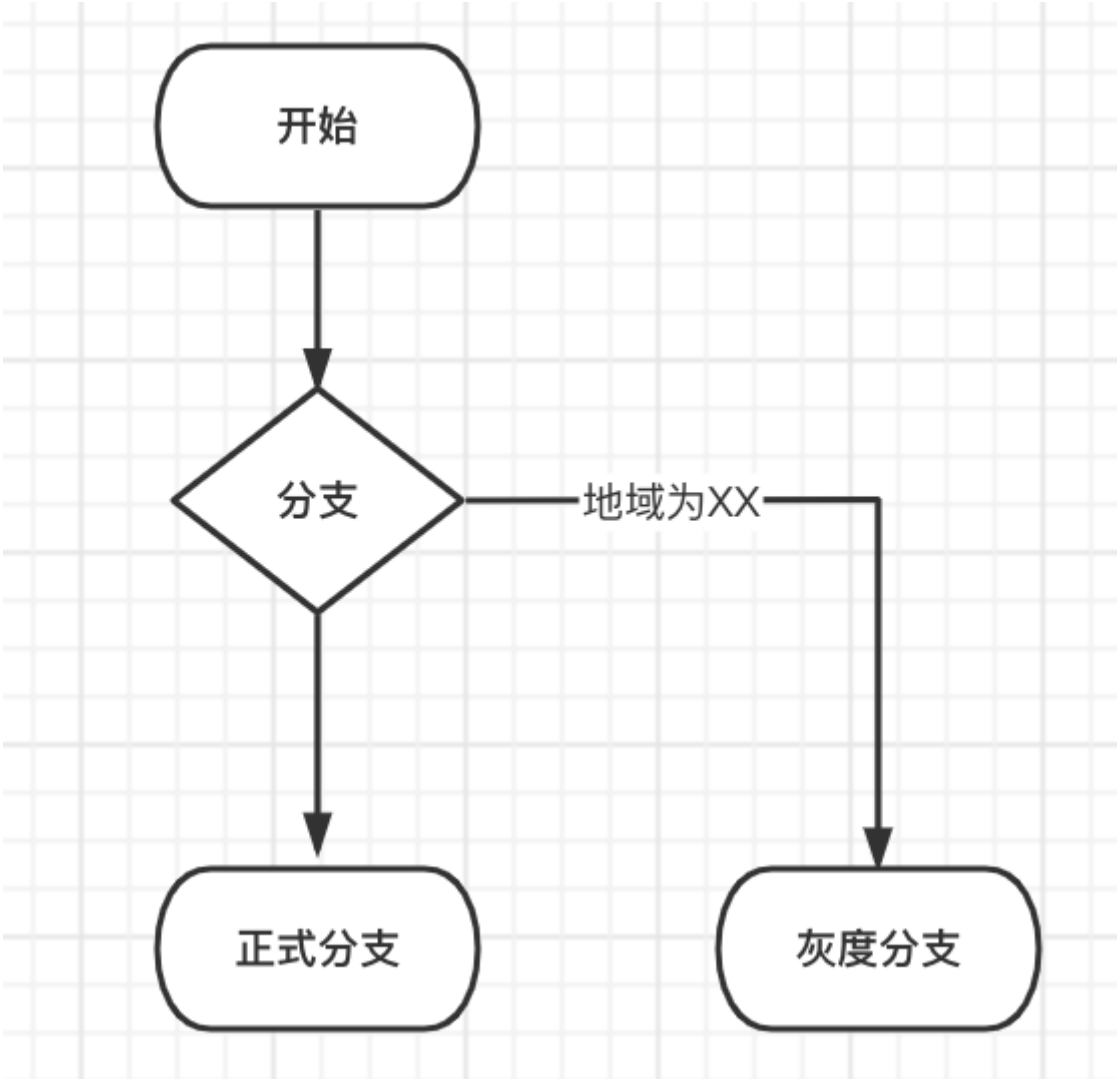
播控接入层服务只支持业务层面的路由，正式和灰度的路由逻辑在 SDK 内部闭环

4.5.2. 双 set 部署方案

播控接入层服务负责正式和灰度逻辑路由

1）播控 SDK 执行层服务分 set 部署，除了正常提供服务的 set 外，单独部署一套灰度 set

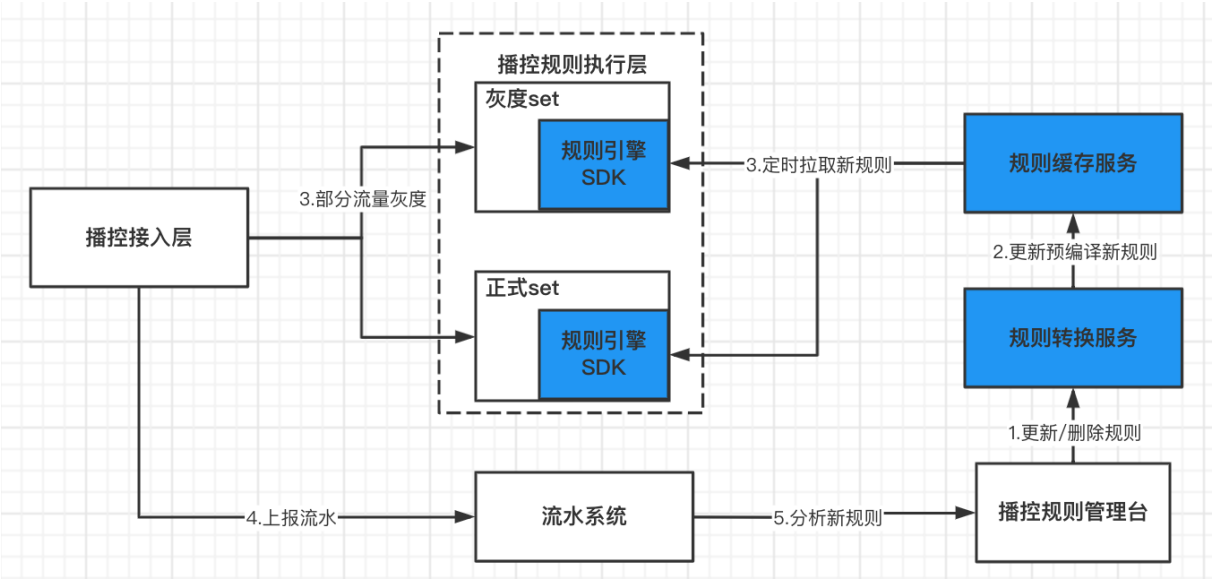
2）当新增/更改规则的时候可以同时指定灰度策略（如下图，按用户/平台/地域...可以直接使用规则引擎进行编写）



3）新的策略可以热加载到灰度 set 上面，同时播控接入层将根据灰度规则将部分流量发送到灰度 set 上进行观察

4）上报流水用于辅助分析新规则是否有问题

5）验证没有问题则将新规则全量发布到所有 set 上，否则进行规则回滚（详情请查看 4.6）



4.6. 规则多版本控制

4.6.1. 规则存储方案

- 规则存储主要分为两个部分，一个是前端流程图逻辑算子的关系数据，另一个是规则 dsl 预编译后的二进制数据，这两者目前都按版本号存储在数据库里，表结构如下：

列名	名称	字段类型	是否有索引
c_id	id	int	主键
c_rule_id	所属规则 id	varchar	是
c_version	版本号	int	是
c_data	规则数据	blob	否
c_operator	操作人	varchar	否
c_mtime	变更时间	timestamp	否

- 规则存储与 SDK 之间提供一个规则缓存服务，提高读取规则数据性能，同时降低 SDK 与

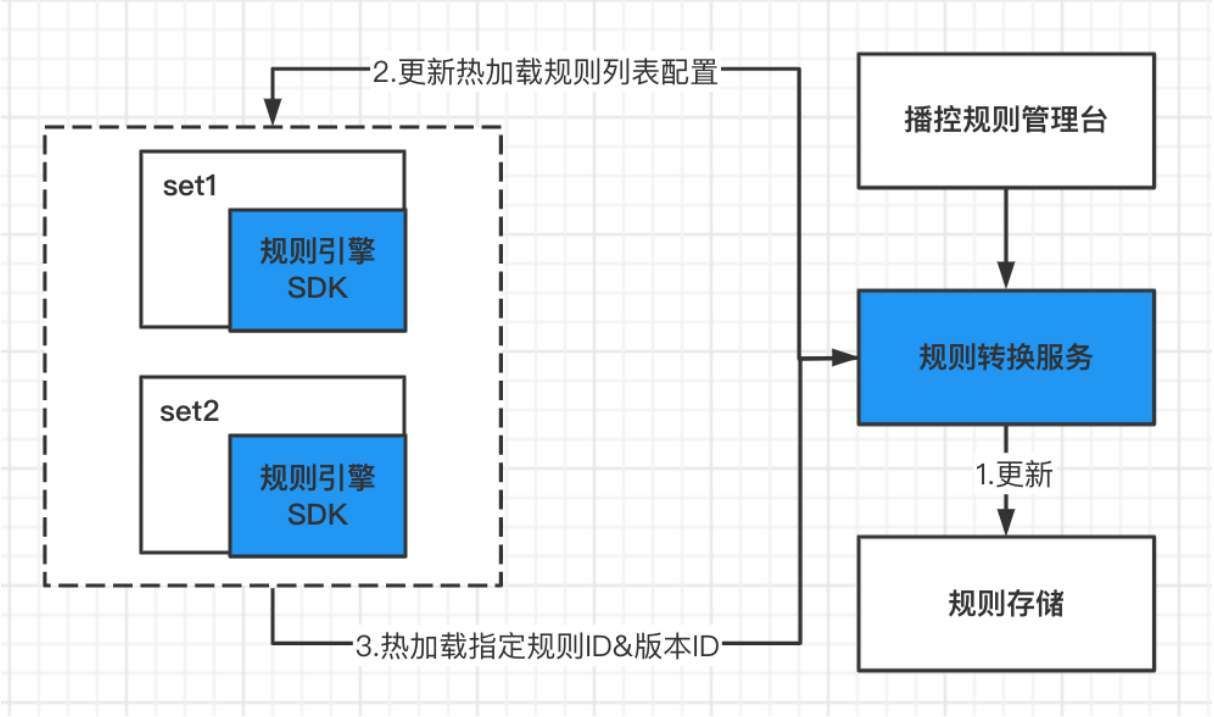
存储的具体上线的耦合度

4.6.2. 规则加载方案

具体的规则加载方案有以下两种：

方案一 热加载规则：

gengine 规则引擎支持规则热加载，因此可以基于这一能力来实现动态更新/删除发生变更的规则。



优点：

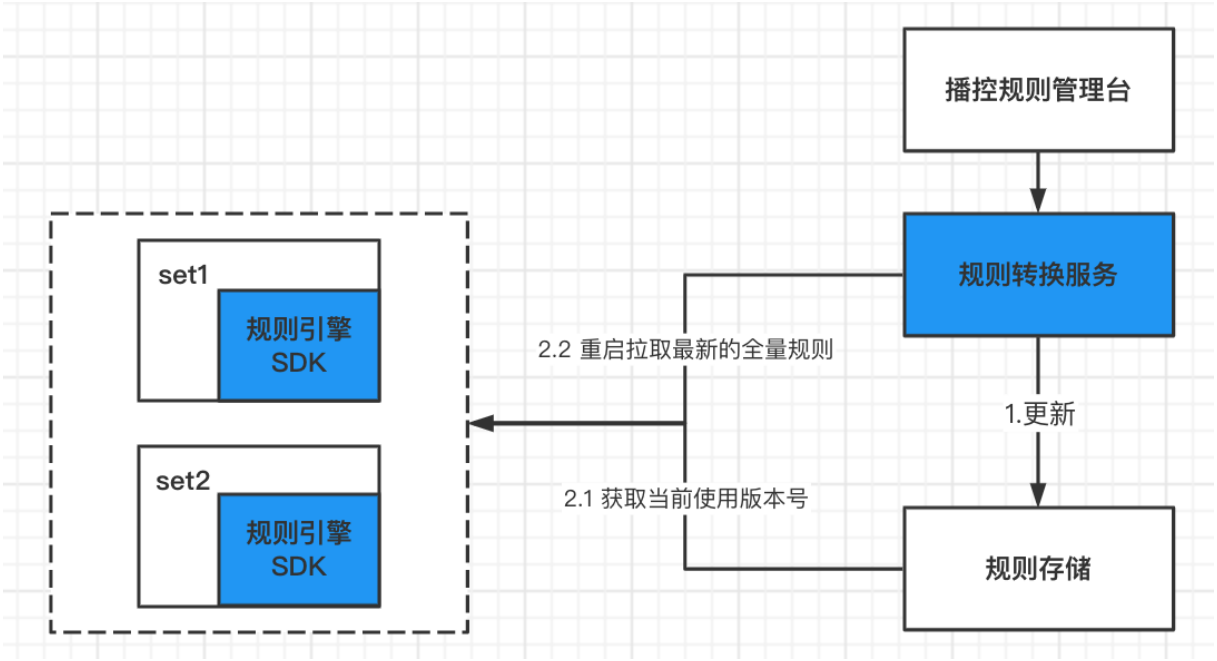
- 热加载，变更成本低，耗时少

缺点：

- 需要额外开发热加载规则逻辑
- 如果支持分批需要记录每个容器实际加载的规则内容

方案二 全量加载规则：

规则管理服务更新最新的全量规则，播控服务重启时拉取最新的全量规则实现规则更新



优点：

- *可以基于重启来实现规则发布，实现比较简单*

缺点：

- *更新成本和风险比较高*

*综上建议使用**方案一**，尽管开发成本更高，但是性能比较好，变更风险比较小*

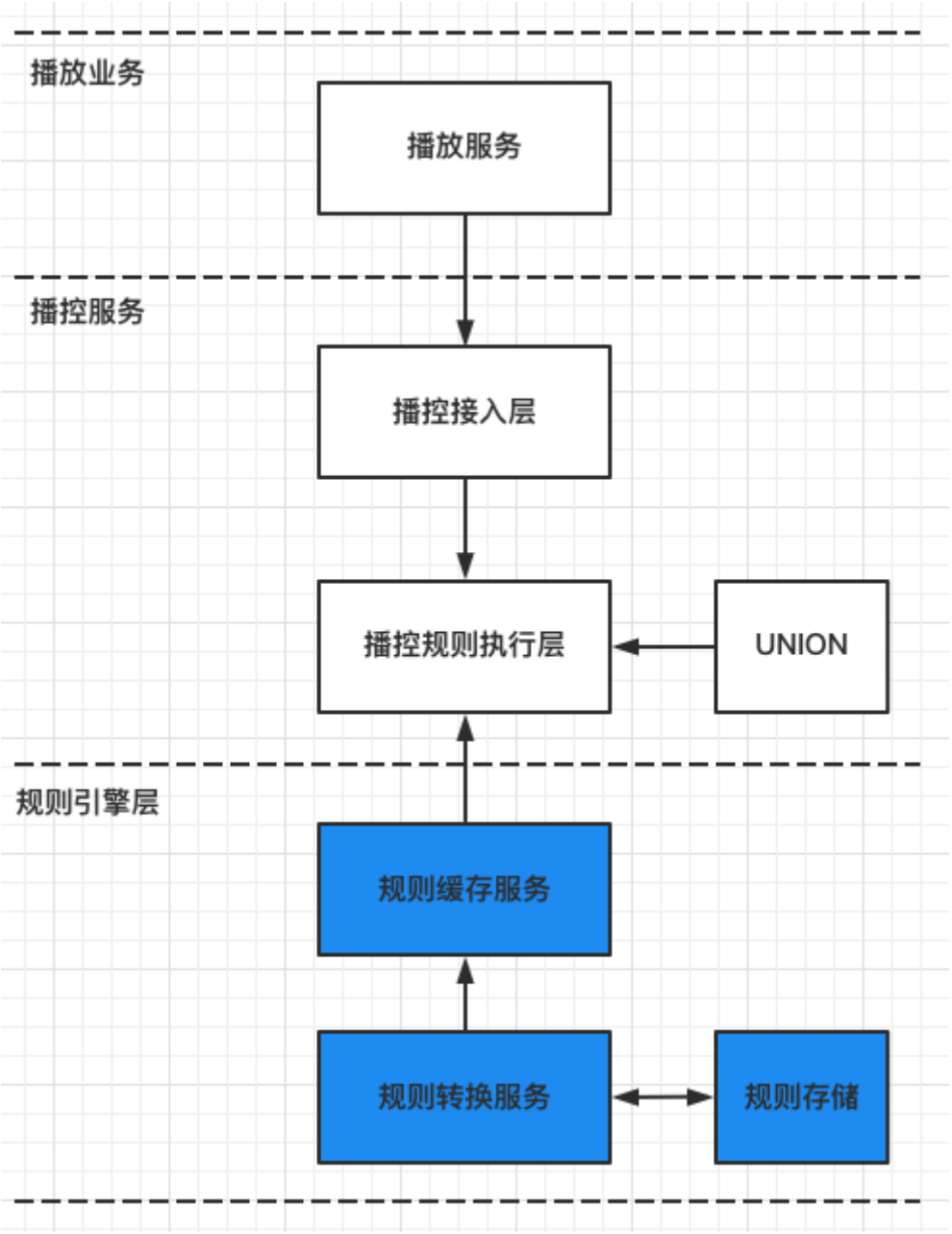
4.7. 流水分析服务

todo：本期是否需要做待讨论

5. 服务高可用性

5.1. 容灾

5.1.1. 系统调用总体架构图



5.1.2. 系统现状

- *播放业务&union*

- *播放服务由点播部门负责，目前是 cpp 旧服务，深圳单地部署*
- *union 目前只有服务层面的多地部署，存储深圳单地部署*

- ***播控服务***
  - *播控规则执行层服务通过规则引擎 SDK 从规则缓存服务拉取指定版本的规则数据*
  - *执行规则入参来源一部分由播放服务直接传参，另一部分播控规则执行层服务从 union 直接拉取*
  - *因为整个规则执行过程数据和规则之间是分离的，所以播控服务整体是无状态的，目前播控服务在深圳单地部署*

- ***规则引擎层***

  规则引擎层服务均在深圳单地部署，规则转换服务会将最新的规则 dsl 预编译为二进制数据，存储到 mdb 上面

5.1.3. 容灾部署方案

5.1.3.1. 按地域进行部署

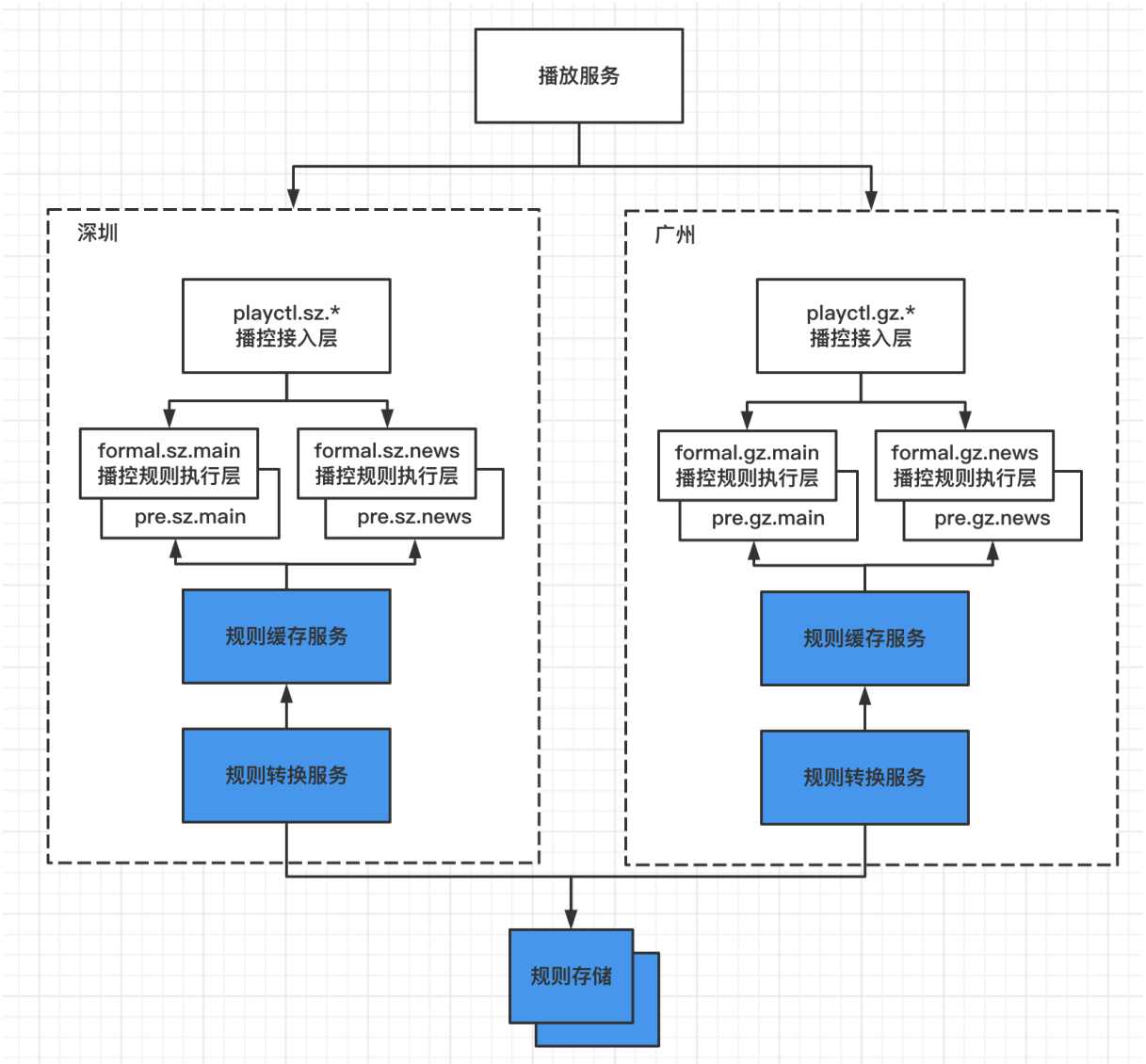
*由于上游服务（播放服务、union）目前没有支持多地部署，所以暂时无法保证**全链路的容灾可靠**，以下仅考虑媒资播控服务内部的容灾方案*

- *播控规则管理台：故障时仅影响更新规则，可以单地部署*
- *规则转换服务：服务需要进行双地部署（深圳&广州），规则存储 mdb 需要部署异地容灾*
- *规则缓存服务：服务需要进行双地部署（深圳&广州），引擎 SDK 根据北极星就近调用原则调用对应地区规则缓存服务*
- *播控接入层服务：服务按 set 进行双地部署（深圳&广州），set 名格式:playctl.{城市}.*（例如：playctl.sz.*）*
- *播控规则执行层服务：服务按 set 进行双地部署（深圳&广州），set 名格式:*

  formal.{城市}.{业务名}（例如：formal.sz.news）

  pre.{城市}.{业务名} （例如：pre.sz.main）

详细如下图：



优点：

可靠性级别高，可以抵御大型灾害或者大型电力灾难

缺点：

- *每个 set 都需要创建一个容灾 set，set 配置比较冗杂*
- *平时异地容灾 set 闲置，资源比较浪费*
- *提高服务路由管理复杂度*

5.1.3.2. 按腾讯云可用区进行部署

目前 123 平台插件支持设置服务可用区级别的容灾配置，可以控制服务的所有服务节点至少部署在两个不同的可用区上

优点：

- *对资源没有浪费，所有可用区都可以提供服务*
- *实现成本低，不需要额外开发工作*
- *路由管理比较简单，不需要为每个 set 配置容灾 set*

缺点：

只能保证保证可用区间故障相互隔离（大型灾害或者大型电力灾难除外）

5.1.3.3. 结论

*由于目前上下游均不支持异地容灾，**建议先选择方案二，后续规划全链路异地容灾***

5.2. 扩容

- *由于播控服务整体是无状态的，所以可以直接通过水平扩容的方式来应对流量增加，目前 123 平台插件支持配置自动扩缩容，可以直接使用*

- *以 CPU 负载作为扩缩容指标，根据播控服务上线后实际性能观察的经验值，配置 **3 天内 CPU 平均负载低于 20%**则进行缩容，**4 分钟内平均负载高于 60%**则进行扩容，同时对最大副本数进行限制（全量时预计设置为 **200**）*
- *规则引擎 SDK 支持加载预编译规则降低服务启动时间，实际测算 **30-60 秒**即可完成服务自动扩容*

5.3. 限流

由于服务扩容无法解决调用流量在短时间内的异常激增，为了提高服务的稳定性，播控服务需

要对上游异常调用进行适当的限流，目前北极星支持自定义限流规则和限流监控，只需改造被调方服务即可低成本实现限流逻辑，基于此有以下两种方案

5.3.1. 分布式限流方案

基于整个播控接入层服务集群进行限流，根据播放服务的调用峰值约定一个限流阈值（例如整体调用峰值的 1.5 倍），当产生突发流量时，节点 CPU 增加触发自动扩容，当流量超过限流阈值则会被拒绝，详情如下图：

优点：

以整体调用量作为限流阈值，比较简单明确

缺点：

- *分布式限流先消费后结算，存在一定误差*
- *可能存在单个节点短时间请求量过大导致服务在一段时间内（目前估计最多 5 分钟）响应变慢*

5.3.2. 单机限流方案

基于单机服务运行的经验值针对单个被调实例的级别的限流，详细如下图：

优点：

按照容器的实际负载能力进行配置，保证服务不会出现过载情况

缺点：

- 短时间内流量激增（整体阈值内），相比分布式限流方案会有更多的请求被频控
- 服务单机阈值和整体限流阈值均不好控制

5.3.3. 结论

播控服务本身依赖 union，如果只限制单机阈值，可能会因为自动扩容导致总请求量超过 union 的承受范围，虽然可以使用设置最大扩容副本数来解决，但是维护成本比较高；建议使用**分布式限流方案**，配合章节 5.2 的自动扩容方案可以一定范围内平滑过渡流量激增的场景，同时起到保护全局存储的作用

5.4. 降级

5.4.1. 服务间调用

播控服务会将调用 union/执行规则引擎的错误直接抛给调用方（播放服务），播放服务针对调用播控服务失败或超时的请求，直接降级为放开播放处理

5.4.2. 服务内加载缓存

规则缓存加载失败分为两种场景：

- 当启动时加载失败直接启动失败，该节点直接被踢除
- 当定时拉取最新规则缓存加载失败时则继续使用上个版本的规则继续提供服务

6. 性能测试

规则引擎 SDK 压测方案

7. 排期