互联网视频系统架构调研报告

171860663 马少聪

一、项目要求的详细分析

1. 独立的前端应用支持用户上传视频、播放视频。

将前端展示界面与业务逻辑处理分割开来，实现了前后端分离，二者只使用 json 数据进行交互。

技术上，后端架构的变化与前端界面互不影响，开发者可以逐步迭代优化架构，只需保证数据接口不变即可。工程上，前后端的开发人员进行了责任分工，约定好接口后即可同时进行开发，缩短了项目开发时间；若某部分代码出现问题，也可以及时找到相应责任人处理，有利于提高维护效率。

2. 将用户上传的视频文件保存在独立的存储系统。

独立的存储系统将数据与服务解耦开来，节省了服务器的存储空间，利用单独的服务器提供存储功能，只需通过 TCP/IP 网络共享资源即可获取文件数据，方便快捷。

我在项目中使用了 NFS 文件系统，但其有以下几个缺点：(1)高并发下 NFS 性能有限。(2)数据是通过明文传送的，安全性较低，可能有信息泄露的风险。(3)若运行该文件系统的服务器发生故障，则所有后端服务器都无法获取数据。

3. 存储完成后，向 rabbitmq 消息队列发送一个编码任务消息

利用消息队列可以起到缓冲的作用，当并发量较大时，将压缩任务缓存在消息队列中，可通过消息确认的机制控制压缩进程的数量，合理分配服务器的计算资源。消息队列提供的 fanout/direct/topic 三种模式为后端的水平扩展提供了方便。消息队列还有效保护了消息信息，若消费者未能成功处理消息，则可通知消息队列将这条消息重新放入消息队列中，以便再次尝试处理，容错性大大提高。

4. 一个独立的编码子系统，负责从消息队列中取出任务，启动一个编码器进行编码。编码结束后将输出的视频文件保存在存储系统中

将编码任务与后端业务处理分隔开，可部署在其他服务器上，节省了后端服务器的计算资源。通过消息队列的消息确认机制可以有效控制同时编码的最大任务数量，便于合理调配机器的算力。另外这个编码子系统是独立的，故可以水平扩展，共用消息队列即可实现多个编码系统同时处理编码任务。

5. 利用性能监控组件进行性能指标监控，在超过阈值时向管理人员发送警告邮件

有效保护了服务器运行状态，便于管理人员在高并发情况下及时进行维护、扩展。

二、项目要求的改进

1. 缺少对处理 http 请求的后端的设计描述

项目中需要后端处理两种主要的 http 请求，一是上传视频，二是播放视频。处理上传视频的请求时，将视频数据保存在 NFS 文件系统，并向消息队列发送编码任务的消息；处理播放视频的请求时，直接将 NFS 中的文件返回给前端即可。

我在项目中将这两种处理结合在一个后端中，但实际上当并发量过大时，服务器可能无法合理调配处理两种 http 请求的计算资源。所以，最好将上传、播放这两种请求的后端处理分别部署到不同服务器上，也便于根据不同请求的并发量合理进行不同程度的水平扩展，服务器利用率、经济效益最大化。

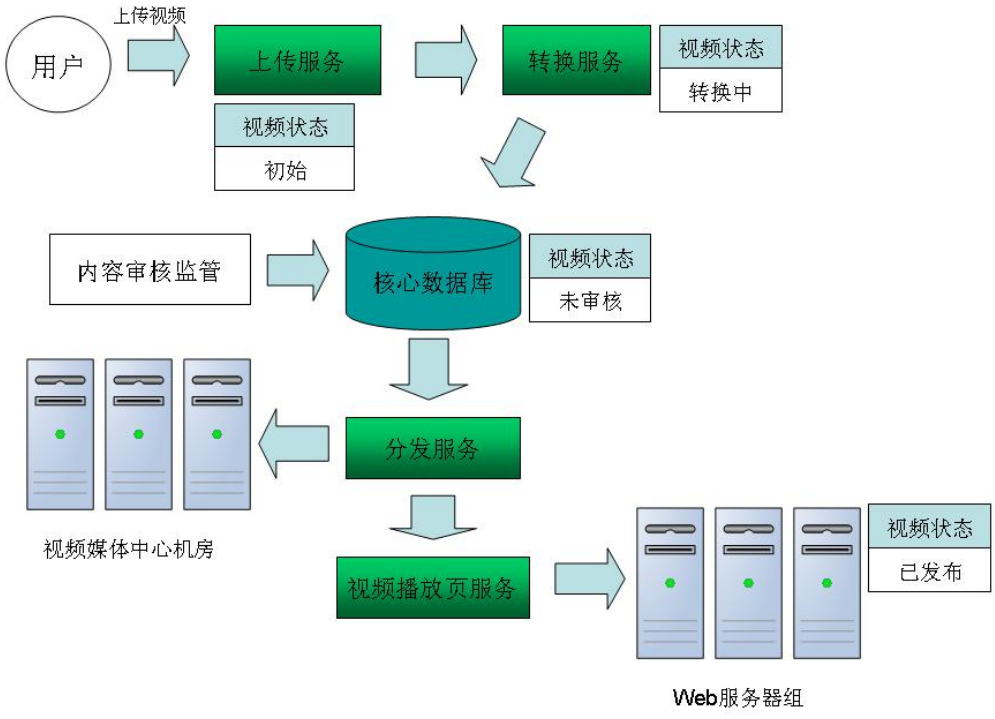
2. 只有存储视频文件的 NFS 文件系统，缺少保存文件信息的 SQL 数据库

项目中需要将 NFS 中已编码好的文件名称返回给前端，我目前的实现是每次刷新前端页面时都向后端发送“get /files”的请求，后端读取目录下的所有文件名并将 json 数据返回给前端。而当 NFS 中视频文件过多、客户端刷新 get 请求过于频繁时，每次读取所有文件名是非常耗时的。若利用 SQL 数据库单独保存这些信息，将极大程度地提高了系统响应效率。

文件名称、文件大小、分辨率、存储位置、上传者、播放者、播放时长等各种信息具有视频网站重要的数据信息价值，需要独立的数据库进行存储。这些文件信息数据、用户上传数据、用户播放数据将有利于向用户提供更好的偏好推荐服务，增大用户量、访问浏览率等，有助于项目良性发展扩大运营规模。

三、视频系统架构调研

1. 视频上传



大型视频播放网站在业务层不仅仅需要满足用户需求，更要符合监管需求。上传视频量太大，内容审核人员忙不过来，出现延迟。需要做策略上的调整，例如：先发布后审核，特殊账号免审核。但有个政策风险需要考虑，万一内容监管出了纰漏，会吃官司，甚至公司被迫关门。

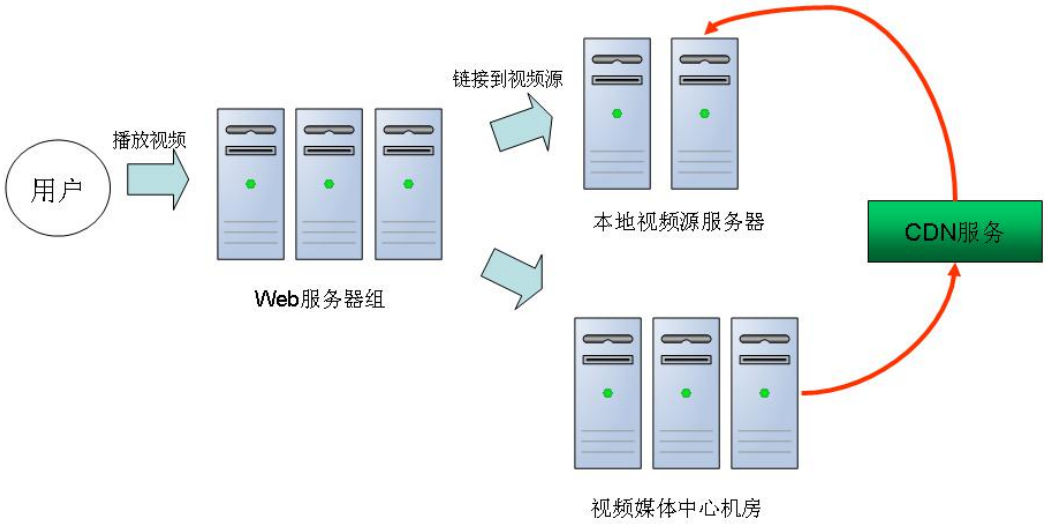
为了满足内容审核的业务需求，一般采用如上图所示的视频上传架构。

(1) 用户上传视频后，未审核的视频保存在核心数据库，有审核监管部门进行审核

(2) 审核完成后，视频被分发到各视频媒体中心机房，便于用户播放时访问

(3) 访问量较高的视频也将分发到各 Web 服务器组作为缓存，以提高系统响应效率

2. 视频播放



对于视频播放业务而言，Web 服务器的压力不是最大的，资源最紧张的地方还是“流媒体服务器”。几乎所有的视频分享网站的播放器都是 flash 技术，不适合看大的视频，非常耗带宽。所以电影电视剧还是使用 P2P 技术的好，既快又省钱。

提高视频缓冲速度的几个策略：足够的带宽，就近访问原则（CDN 内容分发网络服务），大视频分割成小文件。分发策略很重要，不是分发越多就越好，需要有个资源平衡，关键是把热点视频源分发出去。

视频播放的业务架构如上图所示。

(1) 用户发送播放视频的业务请求后，服务器处理请求，就近访问视频源服务器

(2) 本地视频源服务器将查找相应视频源，返回给用户

(3) 若在本地查找不到，则向视频媒体中心机房发送请求获取视频源，并缓存在本地服务器中，以便下次用户访问时使用

3. 视频转码

在传统的架构中，会先将文件传到文件上传服务，文件上传服务将其传到底层存储。传到存储后，文件上传服务会告知转码服务文件需进行转码。转码时转码服务通过调度器将转码任务传到对应的转码集群中的转码服务器。真正转码的机器，从存储中下载用户上传的源文件，转换成特定格式后回存到存储中。

而大规模视频网站的转码需求更为复杂，若让用户一直等待上传、转码过程将大大降低体验，所以提出了更为灵活的转码架构。

微博视频的转码服务架构有以下几个核心模块：

(1) 木林森——灵活配置生成系统：

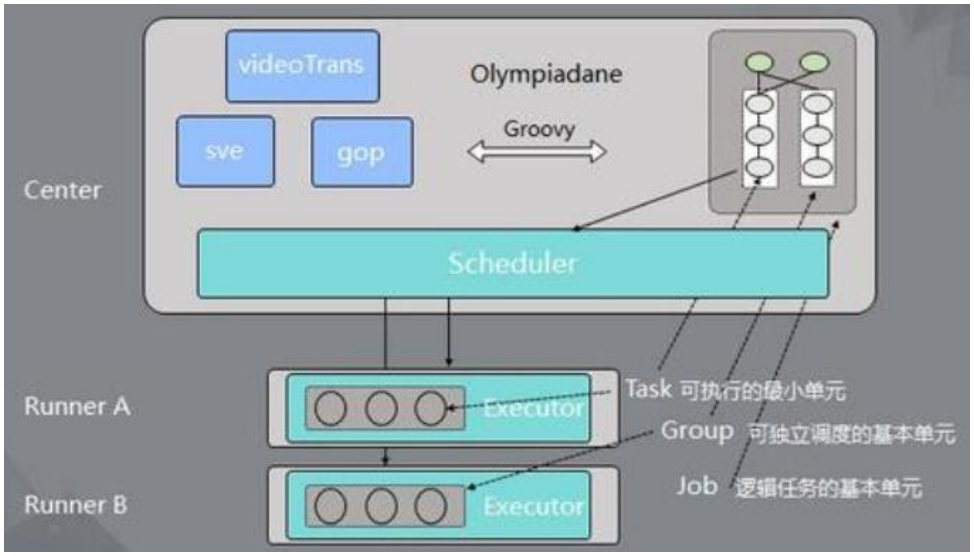
它是一个基于树形结构的规则引擎，通过将不同业务场景下的输入业务和现有的输出业务通过配置文件合理链接，即可达到完成业务接入的目的。

例如，现有“原生视频”的接入业务，现在要接入的“秒拍”希望与“原生视频”有相同的输出，则转码输出的配置均无需改变，只需自动接入即可使用“原生视频”相同的转码输出配置。这各规则配置引擎大大提高了新业务接入的效率。

(2) DAG——基于有向无环图的工作流

为了提高转码效率，一般采用切片转码再合并的方式。服务器 A 需要从数据库 B 中下载视频源文件，切片后将切片结果上传至转码服务器 C，C 得到切片后分别转码并上传至服务器 D 进行切片合并，D 合并成功后需上传给 B 进行存储。

这些基础服务是有相互依赖关系的，可以通过脚本将不同功能组织起来，将各部分的代码拆分成一个个可独立执行的闭包，而 DAG 的作用就是管理包与包之间的关系。

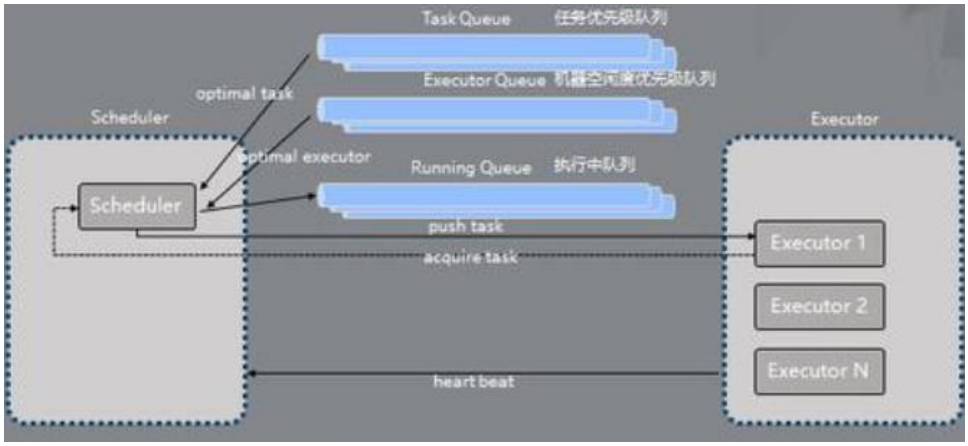


如图所示，Center 部分就是中央调度的服务，Runner 部分是执行转码任务的服务，videoTrans 是通过 Groovy 实现的 DAG 组织任务间关系的脚本。Group 是可独立调度的单位，先经过调度器，调度器根据情况分发到执行器，执行器内部根据前后依赖关系顺序执行 Task，在此例中就是下载分片、转码、上传转码结果。通过脚本生成的就是图中的 Job。

这样就实现了执行流与业务之间的解耦，若要接入其他的新服务的话，只需再实现一个 Task，将此 Task 的依赖关系放入脚本即可完成。

(3) 高可用、高性能的任务调度器

由于进行了视频切片，因此调度任务达到了万次每秒。这使得对调度器有极高要求，需要使百分之九十九的调度任务在 10ms 内分派到对应机器，并且希望它的调度是最优调度，即能准确把任务分派到空闲机器。



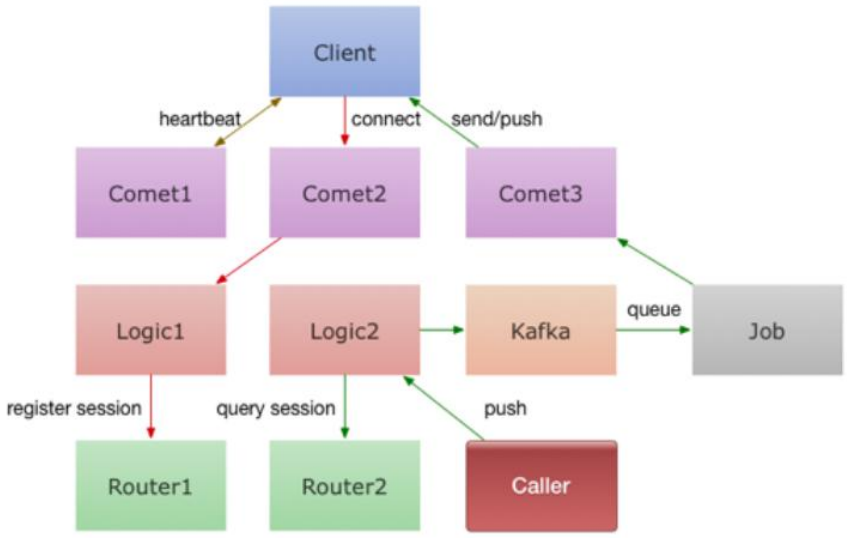
调度器和执行器之间通过心跳注册，心跳时间是可配置的。注册完后会将机器信息放到机器队列，中心资源可以利用任务优先级队列、机器空闲优先级队列将任务派到指定机器，即可将任务放到执行队列中。执行结束后，会进行一次回调，从执行队列中移除任务。

通过三个队列完成任务调度，由于存在资源依赖，所以对这些资源进行了哈希计算，不同机器可以使用不同资源，只要资源满足就可分派任务。

(4) 备用型转码服务部署

在两个 IDC 部署了完全相同的两套资源，它们有独立的域名，独立的部署。这么做的好处是可以在两个机房间随意的切流量，任一机房出现问题都可以切换，但是两个机房的部署并不是一比一的冗余。常备的机房是一个大规模集群，另一个机房是一个小规模的，或许只有常备机房十分之一的量。两个机房在使用时可以分开，例如处理一些不影响用户发博的转码输出时，可以使用小机房完成任务，这样大机房出现“灾难性”情况时，可以把流量切到小机房。当然小机房是不能满足那么大流量的，但是调度器本身的队列有堆积的特性，可以将堆积的任务慢慢执行。

4. 实时弹幕系统 GOIM



(1) **Client 客户端**，与 Comet 建立链接。

(2) **Comet 维护客户端长链接**。在上面可以规定一些业务需求，比如可以规定用户传送的信息的内容、输送用户信息等。Comet 提供并维持服务端与客户端之间的链接，这里保证链接可用性的方法主要是发送链接协议（如 Socket 等）。

(3) **Logic 对消息进行逻辑处理**。用户建立连接之后会将消息转发给 Logic，在 Logic 上可以进行账号验证。当然，类似于 IP 过滤以及黑名单设置此类的操作也可以经由 Logic 进行。

(4) **Router 存储消息**。Comet 将信息传送给 Logic 之后，Logic 会对所收到的信息进行存储，采用 register session 的方式在 Router 上进行存储。Router 里面会收录用户的注册信息，这样就可以知道用户是与哪个机器建立的连接。

(5) **Kafka（第三方服务）消息队列系统**。Kafka 是一个分布式的基于发布/订阅的消息系统，它是支持水平扩展的。每条发布到 Kafka 集群的消息都会打上一个名为 Topic（逻辑上可以被认为是一个 queue）的类别，起到消息分布式分发的作用。

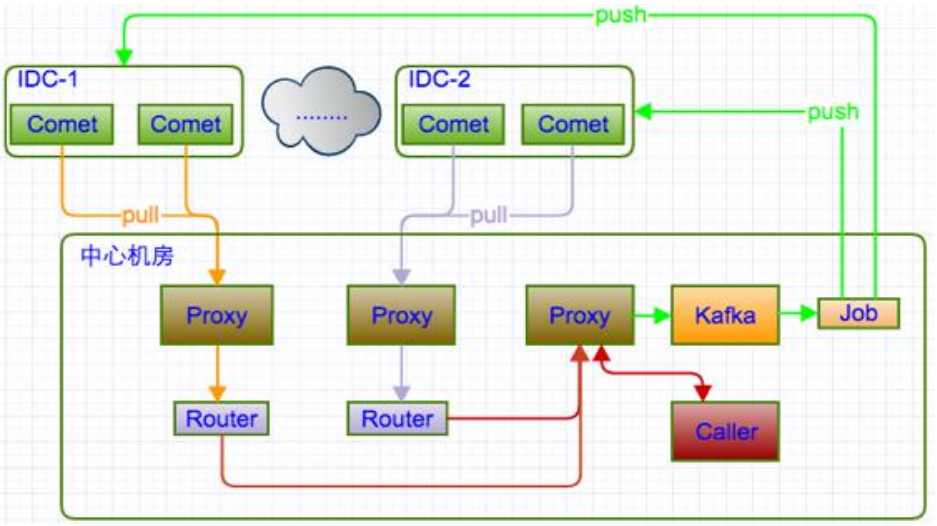
(6) **Job 消息分发**。可以起多个 Job 模块放到不同的机器上进行覆盖，将消息收录之后，分发到所有的 Comet 上，之后再由 Comet 转发出去。

(7) **优化：**

① 内存：一个消息只占用一块内存；一个用户的内存尽量放到栈上。

② 模块：消息分发一定是并行的并且互不干扰；并发数一定是可以进行控制的；根据 CPU 性能合理拆分 Socket 链接池管理、用户在线数据管理的全局锁。

③ 网络部署：将 IP 段进行了城市的划分，将某城市的一些用户信息链接到一个群组，群组下有一个或多个 Comet，把属于这个群组的物理机全部分给 Comet。消息的传输不再使用 push（推）的方式，而是通过 pull（拉）的方式，将数据拉到中心机房（源站），做一些在线处理之后，再统一由源站进行数据推送。



5. 常见实用技术

(1) Ruby on Rails：web 应用程序的框架

(2) Nginx：web 服务器系统

(3) Wowza：Flash/H.264 视频服务器

(4) Usher：播放视频流的逻辑控制服务器
(5) Twice：代理服务系统，主要用缓冲优化应用服务器负载
(6) HAProxy ：TCP/HTTP 负载平衡
(7) XFS：文件系统
(8) PostgreSQL：存放用户和 meta 数据的主从结构数据库
(9) MongoDB：用于内部分析的数据库
(10)MemcachedDB：存放经常要修改的数据的数据库
(11)Syslog-ng：日志服务系统
(12)AWStats：实时日志分析系统
(13)RabbitMQ：基于消息队列的 job 系统
(14)Starling：Ruby 开发的轻量级消息队列
(15)Kestrel：scala 编写的消息中间件
(16)Memcached：分布式内存缓存组件
(17)Munin：服务端监控程序
(18)Nagios：网络监控系统
(19)Git：源代码管理