调研报告

171860031 张兰兰

一．优点：

1. 实现下载器、解码器、渲染引擎这些功能模块的原子化，降低模块间的耦合，方便对外输出，并逐步实现单测覆盖。

2. 限流方面，我们考虑到了转码部分由于是 CPU 密集型任务，设计了任务队列，以使得在一定的系统资源下，只有 N 个编码任务在进行，而不会造成负荷过大的问题出现。

3. 有关重试策略这部分，我实现的是一旦视频上传不成功，还会可以点击重新上传，RabbitMQ 消息队列中也会由于消息处理不成功再次处理。

4. 有关性能监控方面，设计了对于 CPU 使用率的监控，使得 CPU 使用率一旦超过阈值，即提醒监控人员，需要对服务器进行扩容，或者其他方面进行修改。

5. 将 RabbitMQ 和 HAProxy 组合在一起，一定意义上的实现了高可用负载均衡器，证明了系统可以进行水平扩展。

6. 使用了 NFS 服务器，节省了本地的存储空间，并且其他的也可以通过网络访问这台 NFS 服务器上存储的数据，从而实现本地终端的存储空间。

二．缺点：

1. 在上传和下载功能的后端处理是放在一起的，耦合性比较强，这里没有能够降低模块之间的耦合，一旦上传和下载请求一起发来时，就会出现问题。这里可以讲这两部分的处理解耦和。

2. 所有的视频文件都存储在一个 NFS 服务器中，文件并没有进行多份存储，即没有建立缓存机制，可将服务器内存、磁盘、网络都结合起来。

3. 视频解码这部分比较慢，并没有设计专门的解码器，出现了不能够及时的响应业务的问题。可以设计专门的视频解码器。

4. 并没有进行多实例管理，即考虑到多个请求发来时的请求队列管理，没有考虑到多源、多流的业务需求，从而使得多个请求发来时，某些请求会被忽略；可以设计请求队列。

5. 并没有考虑到在网络不佳的环境下，可能会出现的网络卡顿问题，并没有能够实现视频清晰度随着网络环境的改变而改变，从而不出现卡顿的情况；可以进行网络环境的监控，针对网络环境，自动转换视频的清晰度。

6. 视频上传时，上传方式并没有进行加密，容易有安全隐患，同时视频播放时，针对视频本身的处理做的不够，包括其安全检查。

7. 关于编码子系统这里，只是单纯的将一个视频转为 360p，720p，1080p，但是参考一些网站设计，视频可以分为多个小片段，每个小片段可以对应着多个不同清晰度的存储，从而实现随着网络情况的变化，清晰度的变化。

8. 有关负载均衡器这方面，通过部署多个后端实现了一定的流量分发，但是实际服务器安排时，需要考虑到负载均衡器算法的设计，以实现真正意义上的均衡流量分发。

9. 并没有考虑超时处理，例如频繁的刷新请求，如何对其产生的流量冲击和后果进行处理；可以考虑实现一个请求在开始前，就检查是否还有足够的剩余来处理请求。

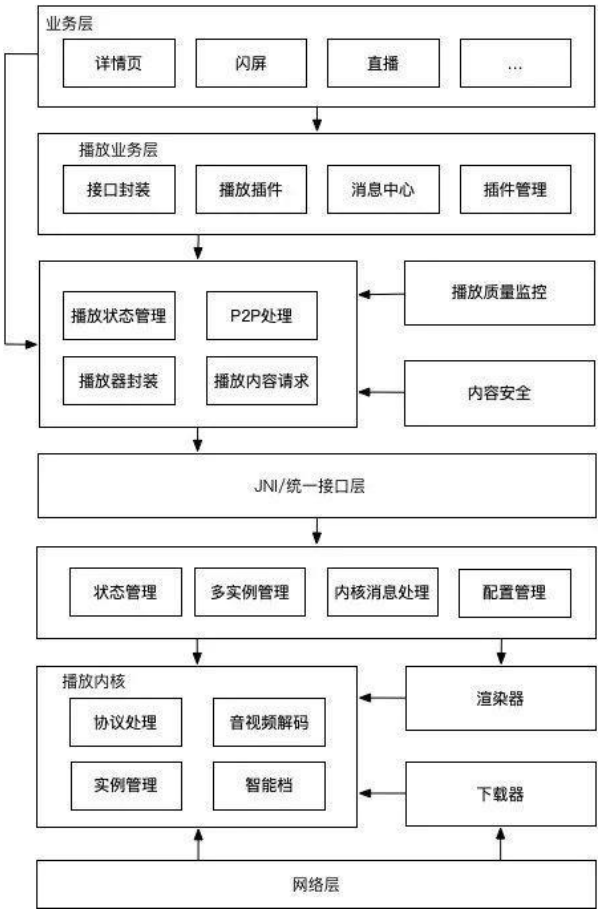
10. 压力测试方面，仅仅做了简单的压力测试，即反映出了扩容是否对服务器的性能表现，有多帮助，而没有考虑到正常的压测需要遇到报错，然后考虑如何提供更好的服务。

11. 监控方面，我只设计了一个监控组件 Prometheus，用来监控 CPU 的使用率，但是现实发展比较成熟的视频网站会有比较完备全面的监控系统，并且监控的方面也会比较完备，包括端监控、链路层监控、日志监控、网络监控，AOC\CDN 的质量监控，中间件、数据库的监控。

三．大规模用户场景下的视频网站架构整体设计：

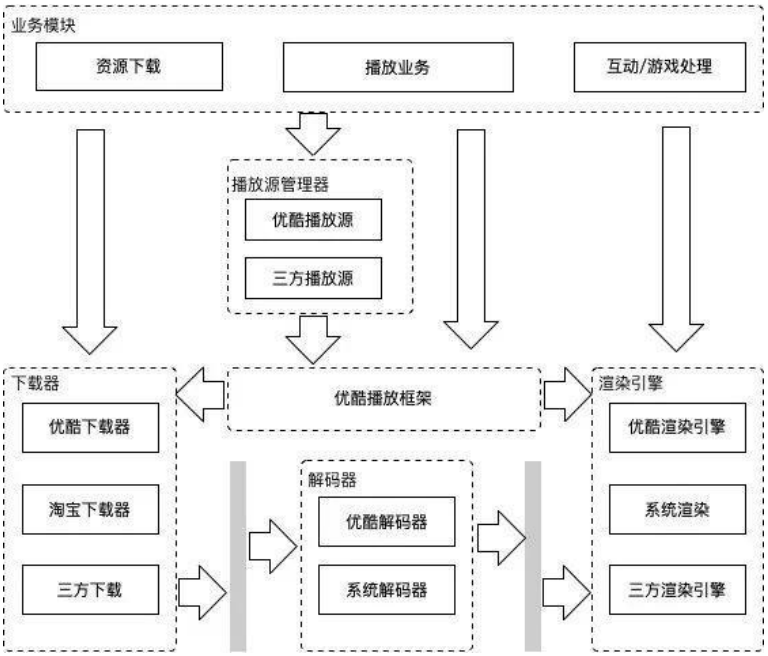
这里我主要以优酷为原型讲解我了解的：

优酷从被阿里收购后，实际上它的网页端播放器经历了全新的架构设计，一开始它的架构是类似于这样的：



这样的架构，其实已经是比较复杂的了，但是对于阿里这样的大公司，我们可以看到架构设计的层次较多，播放部分的逻辑和业务层之间耦合较重。因此，现在的架构设计采用了业内比较流行的基于 Pipeline 的实现：

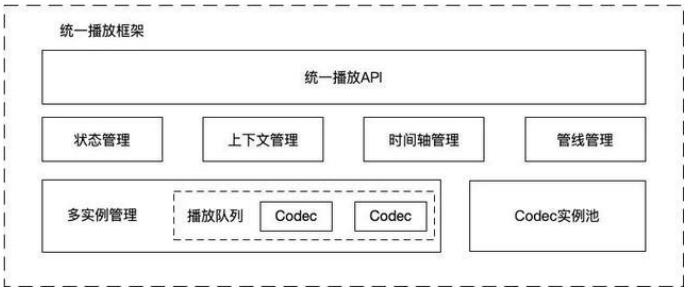
1、整体设计



将整个播放流程抽象为“播放源请求”、“流数据下载”、“流解码”、“后处理和渲染”几个主要的过程，以及“数据埋点中心”“配置中心”等配套设施。在整体架构上，将原来多个层次的实现合并到一个统一的播放框架中，将播放能力和基础业务原子化插件化，由播放框架统一管理并提供可配置化能力。

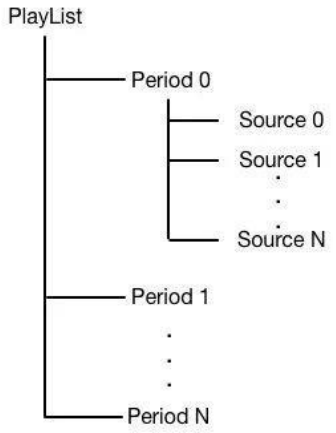
在开放和扩展性的支持上，将以上的几个主要流程抽象成“播放源管理器”、“数据下载器”、“渲染器”并提供统一的定制化开发能力，并提供自适应的解码能力，以满足未来业务创新上对合流、切流、混帧和后处理的特异化开发需求。

2、统一播放框架



播放框架层负责统一管理和串联各个模块，同时对外部提供统一的 API 接口。如图所示，1）接口层提供基础的播放上下行消息、管线注册、模块配置等接口；2）状态管理、上下文管理、时间轴管理和多实例管理模块，可以支持多 Period 和多 Source 的播放序列，以方便业务方能够快速实现可变格式播放源合并、切流等业务；3）实例池模块可以结合设备和可用资源自适应的管理解码器实例，保证稳定性和体验的平衡；4）管线管理模块负责管线注册和管线绑定逻辑；5）插件管理模块支持一些定制能力的内部实现，内置一些优酷业务能力并支持可配置能力。

3、播放源管理



播放源管理模块抽象了一个 PlayList 结构，用于支持业务方能够方便的实现不同格式和编码方式的播放源合并、播放序列管理、切流等业务。主要结构如下图，其中 PlayList 是一个总播放序列；其下的每个 Period 节点表示一个统一的时间轴播放序列，其下的所有 source 会合并成一个时间轴，比如由 4 个 15 秒的视频合并成的一组 60s 贴片广告；每个 Period 下可以挂载多个 source，这些 source 可以支持相同或者不同格式、不同编码参数的视频组合。在播放过程中，允许动态的切换当前 Period，或者修改后续 Period 的顺序。

4、缓存管线

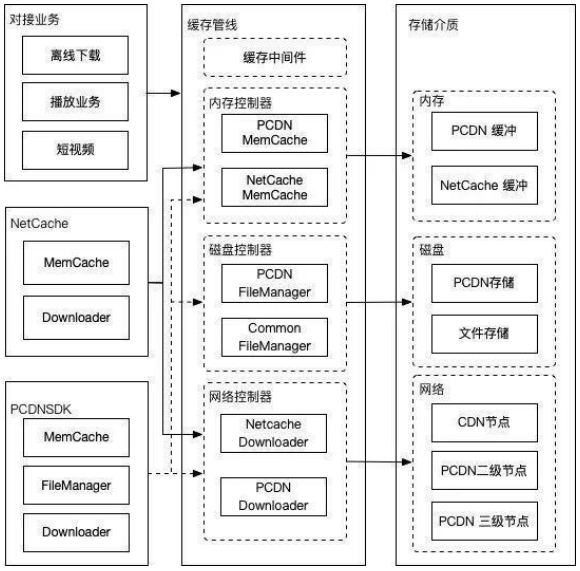
缓存处理采用多级管线的处理，业务方可以根据自己的场景通过缓存中间件和缓存过滤器的定制实现，来满足针对性的数据下载优化。在优酷播放场景中，针对网络请求和本地存储实现了 Net Cache 和 PCDN 两个缓存过滤器，具体如下图所示：

1）将资源存储分为三级缓存管理，由缓存管线进行调度管理。

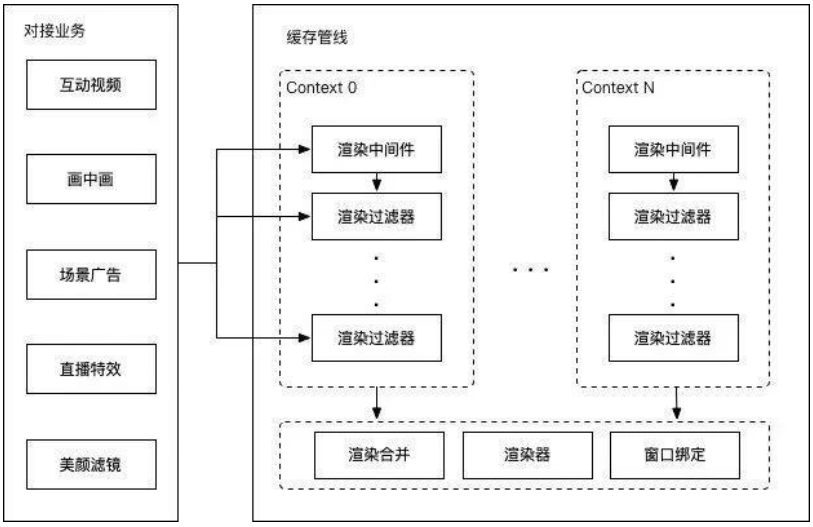
2）业务可以通过参数自定义选择使用混合层级的缓存。

3）缓存管线针对不同资源，读取或写入存储时分别通过访问 Net Cache 或 PCDN 模块进行处理。

4）未来本地磁盘存储除预览需求外，希望统一采用 PCDN 存储方式存储，以提升 PCDN 的分享率，有效的降低成本。



5、渲染管线



后处理和渲染同样采用多级管线的处理。渲染管线模块提供多个渲染 Context 支持，每个 Context 绑定一个解码器和一个渲染窗口，并自动对同一个渲染窗口的多个 Context 的渲染结果进行合并和上屏。业务方可以通过实现渲染中间件和渲染过滤器来进行定制化的开发，以进行诸如混流、混帧、视频特效等特异化开发来满足业务和创新需求。

四．大规模用户场景下的视频网站架构高服务质量设计：

以上都是从比较大的角度考虑系统架构的设计，但是其实系统架构除此之外，还需要考虑一个具有高服务质量的系统架构，应该设计哪些东西：

1. 负载均衡

负载均衡具体分成两个方向，一个是前端负载均衡，另一个是数据中心内部的负载均衡。

前端负载均衡方面，一般而言用户流量访问层面主要依据 DNS，希望做到最小化用户请求延迟。将用户流量最优地分布在多个网络链路上、多个数据中心、多台服务器上，通过动态 CDN 的方案达到最小延迟。以上图为例，用户流量会先流入 BFE 的前端接入层，第一层的 BFE 实际上起到一个路由的作用，尽可能选择跟接入节点比较近的一个机房，用来加速用户请求。然后通过 API 网关转发到下游的服务层，可能是内部的一些微服务或者业务的聚合层等，最终构成一个完整的流量模式。基于此，前端服务器的负载均衡主要考虑几个逻辑：第一，尽量选择最近节点；第二，基于带宽策略调度选择 API 进入机房；第三，基于可用服务容量平衡流量。

数据中心内部的负载均衡方面，如果负载均衡没做好，可能导致资源调度、编排的困难，无法合理分配容器资源。因此，数据中心内部负载均衡主要考虑：均衡流量分发；可靠识别异常节点；scale-out，增加同质节点以扩容；减少错误，提高可用性。

1. 限流

避免过载，是负载均衡的一个重要目标。随着压力增加，无论负载均衡策略如何高效，系统某个部分总会过载。我们优先考虑优雅降级，返回低质量的结果，提供有损服务。在最差的情况，妥善的限流来保证服务本身稳定。

限流这块，我们认为主要关注以下几点：一是针对 qps 的限制，带来请求成本不同、静态阈值难以配置的问题；二是根据 API 的重要性，按照优先级丢弃；三是给每个用户设置限制，全局过载发生时候，针对某些“异常”进行控制非常关键；四是拒绝请求也需要成本；五是每个服务都配置限流带来的运维成本。

在限流策略上，可以采用的是分布式限流。我们通过实现一个 quota-server，用于给 backend 针对每个 client 进行控制，即 backend 需要请求 quota-server 获取 quota。这样做的好处是减少请求 Server 的频次，获取完以后直接本地消费。算法层面使用最大最小公平算法，解决某个大消耗者导致的饥饿。

在过载保护方面，核心思路就是在服务过载时，丢弃一定的流量，保证系统临近过载时的峰值流量，以求自保护。常见的做法有基于 CPU、内存使用量来进行流量丢弃；使用队列进行管理；可控延迟算法：CoDel 等。简单来说，当我们的 CPU 达到 80%的时候，这个时候可以认为它接近过载，如果这个时候的吞吐达到 100，瞬时值的请求是 110，我就可以丢掉这 10 个流量，这种情况下服务就可以进行自保护，我们基于这样的思路最终实现了一个过载保护的算法。

2. 重试

流量的走向，一般会从 BFE 到 SLB 然后经过 API 网关再到 BFF、微服务最后到数据库，这个过程要经过非常多层。在我们的日常工作中，当请求返回错误，对于 backend 部分节点过载的情况下，首先我们需要限制重试的次数，以及基于重试分布的策略；其次，我们只应该在失败层进行重试，当重试仍然失败时，我们需要全局约定错误码，避免级联重试；此外，我们需要使用

随机化、指数型递增的充实周期，这里可以参考 Exponential Backoff 和 Jitter；最后，我们需要设定重试速率指标，用于诊断故障。

而在客户端侧，则需要做限速。因为用户总是会频繁尝试去访问一个不可达的服务，因此客户端需要限制请求频次，可以通过接口级别的 error_details，挂载到每个 API 返回的响应里。

3. 超时

我们之前讲过，大部分的故障都是因为超时控制不合理导致的。首当其冲的是高并发下的高延迟服务，导致 client 堆积，引发线程阻塞，此时上游流量不断涌入，最终引发故障。所以，从本质上理解超时它实际就是一种 Fail Fast 的策略，就是让我们的请求尽可能消耗，类似这种堆积的请求基本上就是丢弃掉或者消耗掉。另一个方面，当上游超时已经返回给用户后，下游可能还在执行，这就会引发资源浪费的问题。再一个问题，当我们对下游服务进行调优时，到底如何配置超时，默认值策略应该如何设定？生产环境下经常会遇到手抖或者错误配置导致配置失败、出现故障的问题。所以我们最好是在框架层面做一些防御性的编程，让它尽可能让取在一个合理的区间内。 进程内的超时控制，关键要看一个请求在每个阶段（网络请求）开始前，检查是否还有足够的剩余来处理请求。另外，在进程内可能会有一些逻辑计算，我们通常认为这种时间比较少，所以一般不做控制。

现在很多 RPC 框架都在做跨进程超时控制，为什么要做这个？跨进程超时控制同样可以参考进程内的超时控制思路，通过 RPC 的源数据传递，把它带到下游服务，然后利用配额继续传递，最终使得上下游链路不超过一秒。

五．大规模用户场景下的视频网站的监控系统设计：

最后，由于我自己最近在做一个监控网络应用的 Job，所以我对于视频网站的监控系统也做了一定的了解:

监控系统可以分为以下几层:

1. 业务层:

监控系统需要关注业务指标，如携程网的酒店下单量，大众点评、唯品会的商品购买指标、实时业务，B 站的注册成功率等。

2. 应用层:

（1）端监控，如：河北地区 APP 无法打开，需要通过端采集数据上报，查找原因；

（2）链路层监控（APM 监控），如：唯品会跟踪订单的完整交易流程，或是完整调用链路，查找异常；

（3）日志监控，通过回溯旧日志，浏览 TLF 等，发现异常。

3. 系统层:

关注网络、AOC、CDN 的质量，关注中间件、数据库的问题等。

而 bilibili 的监控系统主要的设计就是实现以下功能:

1. 编写 ELK 日志分析系统，让研发人员有日志可看，不用重复登录系统查看。

2. 将巨石架构逐步转变为微服务化架构；架构转变之后，我们发现系统发生问题时，很难定位 Root cause。

3. 基于谷歌的 Dapper 编写监控系统，用于快速查找问题。

4. 编写负责 PC 端、移动端链路上报的 Misaka 系统，监控运营商信息。

5. 编写 Traceon 系统，关注业务指标和异常监控，将业务指标报表输出给内容、产品同事，把敏感的变更告知监控系统报警。同时将监控报警短信、邮件，发送给 运维、开发，甚至运营、客服和产品同事。