# 云计算 1607C 史淋伯

# JAVA工程师

1. 项目介绍

|  |
| --- |
| 广告是以“把合适的内容推送给合适的受众”为目的的商业交易过程，它同时为三种人群服务：第一种是广告主，即出钱购买广告的人，需要通过广告获取顾客；第二种是媒体，即提供投放平台以换取广告费收入的人，他们需要广告费实现流量变现；第三种是消费者，即在媒体上观看广告的人，他们希望能低成本地享受媒体给他带来的服务，接受在不反感的前提下获取广告，如果碰到刚好符合他们需求同时也高质量的广告，他们还乐意产生消费。  这个广告平台由广告物料库、广告投放系统、广告检索系统、广告计费系统、广告结算系统、数据分析系统、报表系统、人工智能系统、管理系统、广告反作弊系统等功能模块组成。  此广告系统采用前后端分离方式进行的开发，我们后端使用Spring Boot框架和Spring Cloud服务治理进行微服务开发，采用JPA实现持久层，MySQL存储数据。由Kafka实现服务之间的通信,通过ElasticSearch搜索引擎进行检索。在项目中我们搭建了 Eureka Server实现服务注册与发现、编写API 网关实现微服务统一访问入口与请求记录、编写通用模块开发统一响应、统一异常、统一配置。  首先是广告主先到点阅广告平台上进行注册登录，之后进入广告投放系统进行投放广告，当消费者在媒体方点击阅读广告时，媒体方发送广告信息到广告检索系统检索广告到广告位，这时广告计费系统就根据广告初始设置的计费方式进行计费了，同时广告计费系统还需要实时监测广告的消费情况进行扣费，当费用快达到广告主的当日预算时，就需要对广告进行下架的处理，扣费前还需要经过反作弊系统的过滤来去除恶意点击避免广告主的利益受到损害，然后由数据分析系统获得广告展示的日志文件来进行数据的统计、分析、总结等操作，之后再由广告结算系统与报表系统获得数据分析系统的分析结果来形成报表文件以便之后与广告主进行费用的结算以及交流广告主广告的各种情况，最后人工智能系统再根据这些情况分析后为广告主推荐应该投放的广告类型和信息，使平台为广告主更精准的服务。 |

1. 功能模块介绍

|  |
| --- |
| 我负责的模块是广告投放系统和广告检索系统，对其他模块也有参与协作开发。  我首先说一下广告物料库，广告主根据系统要求提供的广告素材，比如图片和文案等，通过广告投放系统进行投放存储于广告物料库。  当广告主进行注册登录进入广告投放系统，向广告物料库中添加广告创意，再添加广告推广计划与广告推广单元，在单元中也可以添加限制条件（如关键词、地域、兴趣、职业等）对广告投放的人群进行指定以提高广告资源的合理化使用。  媒体方向广告检索系统发起请求（如媒体方标识、限定条件、广告位信息、终端信息等），检索系统对其各种信息进行条件过滤，到广告投放系统中进行查询。  广告投放系统当接受到广告检索系统的请求时，把请求转化为查询条件到索引中进行检索，再把检索结果通过CTR模块进行排序（Ranking），最后数据最终结果并记录投放日志。其中，CTR模块根据实时计算的eCPM对广告检索结果进行从大到小的排序。会计算出最适合的广告创意并返回给广告检索系统，再返回给前端广告位。  广告是展现，是一个广告满足用户需求（初筛）、用户满足广告需求（精筛）、平台利益最大化（bid\*CTR综合排序）的过程。广告的排序不是由出价(bid)决定的，而是由出价(bid)\*点击率(ctr)决定的。点击率(ctr)是一个未来将要发生的行为，点阅广告系统的核心与难点是点击率预测。  广告投放系统关注请求响应时间。因为媒体数量会不断增长，因此广告投放系统是具有高并发、低延迟的特点。点阅广告系统则使用提高检索效率的ElasticSearch作为索引部分。  我们再根据广告位投放情况的反馈以及收费类型，进行计费统计，若当天产生的费用已经达到广告主设定的预算，则需要通知投放系统进行广告下线，这样既避免广告主预算超标，也避免流量被浪费。计费系统的构成不会特别复杂，它是前端广告位与投放系统之间的桥梁，核心任务是保证广告投放在预算范围之内，尽可能地避免发生超投即广告投放的次数与相应费用超出广告主预设的范围。  为了保证这一点，我们要求计费系统应重点保证数据的实时性，具体环节包括广告位展示信息的埋点数据收集、处理与费用计算等。从用户查看或点击一次广告到进行广告下线反馈，这一过程应保持分钟级以内的延迟。由于广告位曝光或用户点击行为的数据量相当庞大，我们采用的方案是使用Kafka提供日志消息分发。  但是你可能会发现，对于点阅广告来说，即使计费系统及时发出下线反馈，但那些已经投放出去、尚未产生点击的广告仍然会可能产生超投。因此超投只能控制在一定范围之内，并不能完全杜绝。广告系统到达一定规模、超投率超出可以接受的范围之后，计费系统应具备预测消耗的能力，即每进行一次投放，计费系统预测出可能产生的费用，并提前进行费用计算、对预测预算将被耗尽的广告计划在点击尚未发生之前就先进行下线操作。另外，对预算将近的广告计划，投放系统也应该降低投放频率，使预算极可能平滑地达到上限。  根据计费系统产生的数据进行费用账单，并由广告主进行缴费。这里就会用到广告结算系统和报表系统。  结算系统一般会以离线数据为基础进行计算，首先，这样可以以较少的成本保证数据完整性，因为如果像计费系统那样一来实时流计算，就不可避免地要面临服务可用性问题。假如流计算服务宕机，对计费系统来说最多也就是造成流量的浪费，但对结算系统来说，则意味着广告平台利益的无辜损失。其次，广告结算系统通常会引入反作弊的功能（比如一个用户短时间内多次点击，只收取一次点击的费用），基于离线数据，便于引入更先进的反作弊算法，以保障广告主的权益。  结算系统与计费系统虽然都提供了费用计算的功能，但侧重点不同。结算系统提供的是广告平台与广告主之间费用结算服务，关注的是数据准确度；而计费系统则重点关注的是计费实时性，其根本目标是保证流量的有效利用。 |

1. 开发中遇到的技术难点以及解决办法

|  |
| --- |
| 难点1：如何通过数据日志判断是否为作弊行为？  解决方法1：  广告访问IP分布异常：发现某几个IP产生大量的点击或者曝光数，这几个IP则存在作弊嫌疑。  广告来源(Referer)异常：点击或者曝光的Referer页面，如果大量来源集中在某一页面，且不是广告所在的Web页面，那么 Referer 页面，可能有作弊的嫌疑。  广告CTR异常：虚拟点击或恶意点击，导致 Click/PV 过高比例，或者起伏很大，那么存在作弊的嫌疑。  广告点击没有对应的曝光请求：如果广告同时监测了曝光和点击，在同时段内存在点击日志却不存在曝光日志，那么存在作弊的嫌疑。  广告访问时间分布异常/规律：例如每分钟或每小时定时出现在点击/曝光日志中，或者连续点击/曝光的发生时间的间隔过于规律，那么存在作弊的嫌疑。  URL，访问者指纹信息(浏览器，操作系统等)异常：例如大量的点击或者曝光数，都来自于同一版本的浏览器或操作系统，或者占比过高，那么存在作弊的嫌疑。  难点2：Kafka 弄丢了数据如何解决？  解决方法2:  设置如下 4 个参数：  给 topic 设置 replication.factor 参数：这个值必须大于 1，要求每个 partition 必须有至少 2 个副本。  在 Kafka 服务端设置 min.insync.replicas 参数：这个值必须大于 1，这个是要求一个 leader 至少感知到有至少一个 follower 还跟自己保持联系，没掉队，这样才能确保 leader 挂了还有一个 follower 吧。  在 producer 端设置 acks=all：这个是要求每条数据，必须是写入所有 replica 之后，才能认为是写成功了。  在 producer 端设置 retries=MAX（很大很大很大的一个值，无限次重试的意思）：这个是要求一旦写入失败，就无限重试，卡在这里了。  我们生产环境就是按照上述要求配置的，这样配置之后，至少在 Kafka broker 端就可以保证在 leader 所在 broker 发生故障，进行 leader 切换时，数据不会丢失。 |

1. 项目技术亮点分析

|  |
| --- |
| 项目采用SpringBoot+SpringCloud+JPA微服务框架。  使用SpringBoot架构来简化项目的初始搭建以及开发过程，比如简化Maven配置、自动配置Spring、创建独立的Spring应用程序、嵌入的Tomcat，无需部署WAR文件、提供生产就绪型功能（如指标、健康检查和外部配置）、绝对没有代码生成并且对XML也没有配置要求。  使用SpringCloud来简化分布式系统基础设施的开发，比如服务发现注册Eureka、配置中心Config、网关Zuul、消息总线Bus、负载均衡、断路器、数据监控等，使得整个项目简单易懂、易于部署和易于维护。  使用JPA将运行期的实体[对象持久化](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1%E6%8C%81%E4%B9%85%E5%8C%96/7316192" \t "https://baike.baidu.com/item/_blank)到数据库中，它有许多的优点，比如支持XML和[JDK](https://baike.baidu.com/item/JDK" \t "https://baike.baidu.com/item/_blank)5.0注解两种元数据的形式（元数据描述对象和表之间的映射关系，框架据此将实体[对象持久化](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1%E6%8C%81%E4%B9%85%E5%8C%96" \t "https://baike.baidu.com/item/_blank)到数据库表中）、API用来操作实体对象执行CRUD操作使框架在后台替代我们完成所有的事情、查询语言通过[面向对象](https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1" \t "https://baike.baidu.com/item/_blank)而非面向数据库的查询语言查询数据达到避免程序的SQL语句紧密耦合。  由于广告系统也有对消费者的各种数据的采集、分析、总结等大数据的实时计算、日志采集的实现。因此，项目还采用了Kafka消息中间件使系统之间达到解耦、实现异步请求、削峰的目的。但MQ也有一些缺点，比如系统可用性降低（就是说如何提高消息队列的高可用）、系统复杂性提高（多出了不少问题，比如怎么[保证消息没有重复消费](https://github.com/doocs/advanced-java/blob/master/docs/high-concurrency/how-to-ensure-that-messages-are-not-repeatedly-consumed.md" \t "https://github.com/doocs/advanced-java/blob/master/docs/high-concurrency/_blank)、怎么[处理消息丢失的情况](https://github.com/doocs/advanced-java/blob/master/docs/high-concurrency/how-to-ensure-the-reliable-transmission-of-messages.md" \t "https://github.com/doocs/advanced-java/blob/master/docs/high-concurrency/_blank)、怎么保证消息传递的顺序性）、一致性问题（存在程序后续出现失败后导致的数据的不一致）。  在项目中我们也解决了Kafka的高可用的问题。我们都知道Kafka由多个broker 组成，每个 broker 是一个节点。创建一个 topic，这个 topic 可以划分为多个 partition，每个 partition 可以存在于不同的 broker 上，每个 partition 就放一部分数据。这就是天然的分布式消息队列，就是说一个 topic 的数据，是分散放在多个机器上的，每个机器就放一部分数据。Kafka 0.8 以后，提供了 HA 机制，就是 replica（复制品） 副本机制。每个 partition 的数据都会同步到其它机器上，形成自己的多个 replica 副本。所有 replica 会选举一个 leader 出来，那么生产和消费都跟这个 leader 打交道，然后其他 replica 就是 follower。写的时候，leader 会负责把数据同步到所有 follower 上去，读的时候就直接读 leader 上的数据即可。Kafka 会均匀地将一个 partition 的所有 replica 分布在不同的机器上，这样才可以提高容错性。也就是说kafka 的高可用保障机制的过程是：多副本 -> leader & follower -> broker 挂了重新选举 leader 即可对外服务。  对于向Kafka中写数据的时候，生产者就写 leader，然后 leader 将数据落地写本地磁盘，接着其他 follower 自己主动从 leader 来 pull 数据。一旦所有 follower 同步好数据了，就会发送 ack 给 leader，leader 收到所有 follower 的 ack 之后，就会返回写成功的消息给生产者。在消费者从Kafka消费消息的时候，只会从 leader 去读，但是只有当一个消息已经被所有 follower 都同步成功返回 ack 的时候，这个消息才会被消费者读到。  Kafka 实际上有个 offset 的概念，就是每个消息写进去，都有一个 offset，代表消息的序号，然后 consumer 消费了数据之后，每隔一段时间（定时定期），会把自己消费过的消息的 offset 提交到 zookeeper，表示“我已经消费过了，下次我要是重启啥的，你就让我继续从上次消费到的 offset 来继续消费吧”。但是凡事总有意外，比如我们之前生产经常遇到的，就是你有时候重启系统，看你怎么重启了，如果碰到点着急的，直接 kill 进程了，再重启。这会导致 consumer 有些消息处理了，但是没来得及提交 offset，尴尬了。重启之后，少数消息会再次消费一次。但是你要是消费到第二次的时候，自己判断一下是否已经消费过了，若是就直接扔了，这样不就保留了一条数据，从而保证了数据的正确性和系统的幂等性。  Kafka消费端弄丢了数据，那么只要关闭自动提交 offset，在处理完之后自己手动提交 offset，就可以保证数据不会丢。Kafka生产者弄丢数据，我们设置了acks=all，一定不会丢，要求是 leader 接收到消息，所有的 follower 都同步到了消息之后，才认为本次写成功了。如果没满足这个条件，生产者会自动不断的重试，重试无限次。  我们也要考虑到Kafka消息的顺序性，生产者在写的时候，其实可以指定一个 key，比如说我们指定了某个订单 id 作为 key，那么这个订单相关的数据，一定会被分发到同一个 partition 中去，而且这个 partition 中的数据一定是有顺序的。写 N 个内存 queue，具有相同 key 的数据都到同一个内存 queue；然后对于 N 个线程，每个线程分别消费一个内存 queue 即可，这样就能保证顺序性。  我们也有消费端出故障，大量消息在 mq 里积压了几个小时了还没解决，我们则采用临时紧急扩容。具体操作步骤和思路如下：  先修复 consumer 的问题，确保其恢复消费速度，然后将现有 consumer 都停掉。新建一个 topic，partition 是原来的 10 倍，临时建立好原先 10 倍的 queue 数量。然后写一个临时的分发数据的 consumer 程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的 10 倍数量的 queue。接着临时征用 10 倍的机器来部署 consumer，每一批 consumer 消费一个临时 queue 的数据。这种做法相当于是临时将 queue 资源和 consumer 资源扩大 10 倍，以正常的 10 倍速度来消费数据。等快速消费完积压数据之后，得恢复原先部署的架构，重新用原先的 consumer 机器来消费消息。  如果要对消息队列进行架构设计的话，要考虑到可支持伸缩性（设计成分布式）、数据要落地磁盘（顺序写）、可用性（高可用保障机制）、支持数据0丢失（设计数据零丢失方案）等从架构角度整体构思和设计。  项目还采用了ElasticSearch搜索引擎，ElasticSearch 设计的理念就是分布式搜索引擎，底层其实还是基于 lucene（lucene 是最先进、功能最强大的搜索库，如果直接基于 lucene 开发，非常复杂，即便写一些简单的功能，也要写大量的 Java 代码，需要深入理解原理）的，隐藏了 lucene 的复杂性，提供了简单易用的 restful api / Java api 接口。核心思想就是在多台机器上启动多个 es 进程实例，组成了一个 es 集群。  es 中存储数据的基本单位是索引，一个 index 里可以有多个 type，每个 type 的字段都是差不多的，但是有一些略微的差别，可以认为 index 是一个类别的表，具体的每个 type 代表了 mysql 中的一个表，往 index 里的一个 type 里面写的一条数据，叫做一条document，一条 document 就代表了 mysql 中某个表里的一行，每个 document 有多个 field，每个 field 就代表了这个 document 中的一个字段的值。  索引可以拆分成多个 shard，每个 shard 存储部分数据。拆分多个 shard 是有好处的，一是支持横向扩展，二是提高性能，数据分布在多个 shard，即多台服务器上，所有的操作，都会在多台机器上并行分布式执行，提高了吞吐量和性能。接着就是这个 shard 的数据实际是有多个备份，就是说每个 shard 都有一个 primary shard，负责写入数据，但是还有几个 replica shard。  es 写数据过程：客户端选择一个 node 发送请求过去，这个 node 就是 coordinating node（协调节点）。coordinating node 对 document 进行路由，将请求转发给对应的 node（有 primary shard）。实际的 node 上的 primary shard 处理请求，然后将数据同步到 replica node。coordinating node 如果发现 primary node 和所有 replica node 都搞定之后，就返回响应结果给客户端。  数据先写入内存 buffer，然后每隔 1s，将数据 refresh 到 os cache，到了 os cache 数据就能被搜索到（所以我们才说 es 从写入到能被搜索到，中间有 1s 的延迟）。每隔 5s，将数据写入 translog 文件（这样如果机器宕机，内存数据全没，最多会有 5s 的数据丢失），translog 大到一定程度，或者默认每隔 30mins，会触发 commit 操作，将缓冲区的数据都 flush 到 segment file 磁盘文件中。数据写入 segment file 之后，同时就建立好了倒排索引。  这样下来 segment file 会越来越多，此时会定期执行 merge。每次 merge 的时候，会将多个 segment file 合并成一个，同时这里会将标识为 deleted 的 doc 给物理删除掉，然后将新的 segment file 写入磁盘，这里会写一个 commit point，标识所有新的 segment file，然后打开 segment file 供搜索使用，同时删除旧的 segment file。  在搜索引擎中，每个文档都有一个对应的文档 ID，文档内容被表示为一系列关键词的集合。倒排索引就是关键词到文档 ID 的映射。  es 读数据过程：客户端发送请求到任意一个 node，成为 coordinate node。协调节点将搜索请求转发到所有的 shard 对应的 primary shard 或 replica shard都可以。每个 shard 将自己的搜索结果（其实就是一些 doc id）返回给协调节点，由协调节点进行数据的合并、排序、分页等操作，产出最终结果。接着由协调节点 对 doc id 进行哈希路由，将请求转发到对应的 node，此时会使用 round-robin 随机轮询算法，在 primary shard 以及其所有 replica 中随机选择一个，让读请求负载均衡。接收请求的 node 返回 document 给 coordinate node。coordinate node 返回 document 给客户端。  往 es 里写的数据，实际上都写到磁盘文件里去了，查询的时候，操作系统会将磁盘文件里的数据自动缓存到 filesystem cache 里面去。es 的搜索引擎严重依赖于底层的 filesystem cache，如果给 filesystem cache 更多的内存，尽量让内存可以容纳所有的 idx segment file 索引数据文件，那么搜索的时候就基本都是走内存的，性能会非常高。 |

1. 项目答辩中亮点简历审核评定标准

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **考核内容** | **简历样式是否规范** | **项目内容是否完整** | **项目中是否有亮点** | **技术难点解决办法是否高效** | **存在问题** |
| **项目经理 （评定）** |  |  |  |  |  |
| **就业处干事（评定）** |  |  |  |  |  |
| 简历是否合格 | | | | 合格 | 不合格 |

1. 项目答辩成绩评定标准

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **考核内容** | **亮点简历与项目描述是否匹配** | **项目流程是否清晰** | **学生讲解是否流畅** | **项目介绍是否到位** | **技术亮点考核是否合格** | **存在问题** |
| **项目经理 （评定）** |  |  |  |  |  |  |
| **就业处干事（评定）** |  |  |  |  |  |  |
| 项目答辩是否合格 | | | | 合格 | 不合格 |  |