FunData — 电竞大数据系统架构演进

肖丁 IT大咖说 2018-06-22

肖丁 / VPGAME上海技术团队负责人

嘉宾介绍: VPGAME上海技术团队负责人，美国团队架构师，曾任职于UCloud，负责基础架构、中间件及分布式系统的工作。

背景来源： FunData作为电竞数据平台，v1.0 beta版本主要提供由Valve公司出品的顶级MOBA类游戏DOTA2相关数据接口(详情：open.varena.com)。数据对比赛的观赏性和专业性的提高起到至关重要的作用。本文由IT大咖说（微信id：itdakashuo）整理，经投稿者与嘉宾审阅授权发布。

阅读字数：4822 | 13分钟阅读

摘要

本文将介绍FunData的架构演进中的设计思路及其涉及的相关技术，包括大数据流处理方案、结构化存储转非结构化存储方案和数据API服务设计等。

无论是对于观赛用户，还是比赛用户，电竞数据的丰富程度与实时要求得到越来越多的关注。

电竞数据的丰富性从受众角度来看，可分为赛事、战队和玩家数据；从游戏角度来看，维度可由英雄、战斗、道具以及技能等组成；电竞数据的实时性包括赛前两支战队的历史交战记录、赛中的实时比分、胜率预测、赛后比赛分析和英雄对比等。

因此多维度的数据支持，TB到PB级别的海量数据存储与实时分析都对底层系统的架构设计有着更高的要求，亦带来了更严峻的挑战。

1.0架构

项目发展初期，依照MVP理论（最小化可行产品），我们迅速推出FunData的第一版系统（架构图如图1）。系统主要有两个模块：Master与Slave。

Master模块功能如下：

- 定时调用Steam接口获取比赛ID与基础信息
- 通过In-Memory的消息队列分发比赛分析任务到Slave节点
- 记录比赛分析进度，并探测各Slave节点状态

Slave模块功能如下：

- 监听队列消息并获取任务(任务主要为录像分析，录像分析参考github项目clarity(https://github.com/skadistats/clarity) 与manta(https://github.com/dotabuff/manta))
- 分析数据入库

系统上线初期运行相对稳定，各维度的数据都可快速拉取。然而随着数据量的增多，数据与系统的可维护性问题却日益突出。

1. 新增数据字段需要重新构建DB索引，数据表行数过亿构建时间太长且造成长时间锁表。
2. 系统耦合度高，不易于维护，Master节点的更新重启后，Slave无重连机制需要全部重启；同时In-Memory消息队列有丢消息的风险。
3. 系统可扩展性低，Slave节点扩容时需要频繁的制作虚拟机镜像，配置无统一管理，维护成本高。
4. DB为主从模式且存储空间有限，导致数据API层需要定制逻辑来分库读取数据做聚合分析。
5. 节点粒度大，Slave可能承载的多个分析任务，故障时影响面大。

图1 1.0 ETL 架构图

在开始2.0架构设计与改造前，我们尝试使用冷存储方法，通过迁移数据的方式来减轻系统压力（架构设计如图2）。由于数据表数据量太大，并发多个数据迁移任务需要大量时间，清理数据的过程同样会触发重新构建索引，方案的上线并没有根本性地解决问题。

图2 冷存储方案

2.0架构

吸取1.0系统的经验，在2.0架构设计中，我们从维护性、扩展性和稳定性三个方面来考虑新数据系统架构应该具备的基本特性：

- 数据处理任务粒度细化，且支持高并发处理（全球一天DOTA2比赛的场次在120万场，录像分析相对耗时，串行处理会导致任务堆积严重）
- 数据分布式存储
- 系统解耦，各节点可优雅重启与更新

图3 2.0ETL总架构图

2.0系统选择Google Cloud Platform来构建整个数据ETL系统，利用PubSub（类似Kafka）作为消息总线，任务被细化成多个Topic进行监听，由不同的Worker进行处理。这样一方面减少了不同任务的耦合度，防止一个任务处理异常导致其他任务中断；另一方面，任务基于消息总线传递，不同的数据任务扩展性变得更好，性能不足时可快速横向扩展。

任务粒度细化

从任务粒度上看(如图3)，数据处理分为基础数据处理与高阶数据处理两部分。基础数据，即比赛的详情信息（KDA、伤害与补刀等数据）和录像分析数据（Roshan击杀数据、伤害类型与英雄分路热力图等数据）由Supervisor获取Steam数据触发，经过worker的清理后存入Google Bigtable；高阶数据，即多维度的统计数据（如英雄、道具和团战等数据），在录像分析后触发，并通过GCP的Dataflow和自建的分析节点(worker)聚合，最终存入MongoDB与Google Bigtable。

从Leauge-ETL的细化架构看(如图4)，原有的单个Slave节点被拆分成4个子模块，分别是联赛数据分析模块、联赛录像分析模块、分析/挖掘数据DB代理模块和联赛分析监控模块。

1. 联赛数据分析模块负责录像文件的拉取(salt、meta文件与replay文件的获取)与比赛基本数据分析
2. 联赛录像分析模块负责比赛录像解析并将分析后数据推送至PubSub
3. 分析/挖掘数据DB代理负责接收录像分析数据并批量写入Bigtable
4. 联赛分析监控模块负责监控各任务进度情况并实时告警

同时所有的模块选择Golang重构，利用其“天生”的并发能力，提高整个系统数据挖掘和数据处理的性能。

图4 League-ETL架构

分布式存储

如上文提到，1.0架构中我们使用MySQL存储大量比赛数据及录像分析数据。MySQL在大数据高并发的场景下，整体应用的开发变得越来越复杂，如无法支持schema经常变化，架构设计上需要合理地考虑分库分表的时机，子库的数据到一定量级时面临的扩展性问题。

参考Google的Bigtable（详情见Big table: A Distributed Storage System for Structured Data）及Hadoop生态的HBase（图5），作为一种分布式的、可伸缩的大数据存储系统，

Bigtable与HBase能很好的支持数据随机与实时读写访问，更适合FunData数据系统的数据量级和复杂度。

图5 Hadoop生态

在数据模型上，Bigtable与HBase通过RowKey、列簇列名及时间戳来定位一块数据(Cell)。（如图6）

```
(rowkey:string,columnfamily:columnstring,timestamp:int64)→value:string
```

图6 数据索引

例如，在FunData数据系统中，比赛数据的RowKey以hash_key+match_id的方式构建，因为DOTA2的match_id是顺序增大的（数值自增量不唯一），每个match_id前加入一致性哈希算法算出的hash_key，可以防止在分布式存储中出现单机热点的问题，提升整个存储系统的数据负载均衡能力，做到更好的分片(Sharding)，保障后续DataNode的扩展性。

(如图7) 我们在hash环上先预设多个key值作为RowKey的前缀，当获取到match_id时，通过一致性哈希算法得到match_id对应在hash环节点的key值，最后通过key值与match_id拼接构建RowKey。

```
RowKey=Hash(MatchID)+MatchID=Key_n+MatchID
```

图7 一致性hash构建RowKey

时间戳的使用方便我们在聚合数据时对同一个RowKey和Column的数据重复写入，HBase/Bigtable内部有自定的GC策略，对于过期的时间戳数据会做清理，读取时取最新时间节点的数据即可。

这里大家可能会有个疑问，Bigtable与HBase只能做一级索引，RowKey加上hash_key之后，是无法使用row_range的方式批量读或者根据时间为维度进行批量查询的。在使用Bigtable与HBase的过程中，二级索引需要业务上自定义。在实际场景里，我们的worker在处理每个比赛数据时，同时会对时间戳-RowKey构建一次索引并存入MySQL，当需要基于时间批量查询时，先查询索引表拉取RowKey的列表，再获取对应的数据列表。

在数据读写上，Bigtable/HBase与MySQL也有很大的不同。一般MySQL使用查询缓存，schema更新时缓存会失效，另外查询缓存是依赖全局锁保护，缓存大量数据时，如果查询缓存失效，会导致表锁死。

如图8，以HBase为例，读取数据时，client先通过zookeeper定位到RowKey所在的RegionServer，读取请求达到RegionServer后，由RegionServer来组织Scan的操作并最终归并查询结果返回数据。因为一次查询操作可能包括多个RegionServer和多个Region，数据的查找是并发执行的且HBase的LRUBlockCache，数据的查询不会出现全部锁死的情况。

图8 HBase架构

基于新的存储架构，我们的数据维度从单局比赛扩展到了玩家、英雄、联赛等。（如图9）

图9 数据维度

系统解耦

上文我们提到1.0架构中使用In-Memory的消息队列做数据传递，由于内存中队列数据没有持久化存储并与Master模块强耦合，Master节点更新或者异常Panic后会导致数据丢失，且恢复时间冗长。因此，在2.0架构中采用了第三方的消息队列作为消息总线，保障系统“上下游”节点解耦，可多次迭代更新，历史消息变得可追踪，基于云平台消息堆积也变得可视化(如图10)。

图10 数据监控

数据API层

1.0系统的数据API层为实现快速上线，在架构上未做太多的设计与优化，采用域名的方式实现负载均衡，并使用开源的DreamFactory搭建的ORM层，利用其RESTful的接口做数据访问。该架构在开发和使用过程中遇到许多问题：

1. API层部署在国内阿里云上，数据访问需要跨洋
2. ORM层提供的API获取表的全字段数据，数据粒度大
3. 无缓存，应对大流量场景(如17年震中杯与ESL)经常出现服务不可用
4. 多DB的数据聚合放在了API层，性能不足
5. 服务更新维护成本高，每次更新需要从域名中先剔除机器

针对上述问题，我们从两个方面重构了1.0数据API层。（如图11）

图11 数据API新架构

链路的稳定性

全球链路上，我们使用CDN动态加速保证访问的稳定性。同时利用多云厂商CDN做备份容灾，做到秒级切换。

在调度能力和恢复能力上，我们搭建了自己的灰度系统，将不同维度的数据请求调度到不同的数据API，减少不同维度数据请求量对系统的影响；借助灰度系统，API服务更新的风险和异常时的影响面也被有效控制。依赖云平台可用区的特性，灰度系统也能方便地实现后端API服务跨可用区，做到物理架构上的容灾。

另外，为保证内部跨洋访问链路的稳定性，我们在阿里云的北美机房搭建数据代理层，利用海外专线来提升访问速度。

数据高可用性

接入分布式存储系统后，对外数据API层也根据扩展的数据维度进行拆分，由多个数据API对外提供服务，例如比赛数据和联赛赛程等数据访问量大，应该与英雄、个人及道具数据分开，防止比赛/赛事接口异常影响所有数据不可访问。

针对热点数据，内部Cache层会做定时写入缓存的操作，数据的更新也会触发Cache的重写，保障赛事期间的数据可用。

结语

在本篇的技术分享中，我们介绍了FunData数据平台架构演进的过程，分析了旧系统在架构上的缺点，以及在新系统中通过什么技术和架构手段来解决。FunData自4月10日上线以来，已有300多位技术开发者申请了API-KEY。我们也在着力于新数据点的快速迭代开发，如联赛统计数据，比赛实时数据等。**下一篇我们将介绍FunData系统如何基于K8S进行跨云平台的管理及处理耗资源的计算上，如何利用Serverless服务/函数计算提高内部系统资源利用率等内容。**

点击【**阅读原文**】 打开干货通道

阅读原文