HBase 读流程解析与优化的最佳实践

本文首先对 HBase
做简单的介绍，包括其整体架构、依赖组件、核心服务类的相关解析。再重点介绍 HBase
读取数据的流程分析，并根据此流程介绍如何在客户端以及服务端优化性能，同时结合有赞线上 HBase 集群的实际应用情况，将理论和实践结合，希望能给读者带来启发。如文章有纰漏请在下面留言，我们共同探讨共同学习。

HBase 简介

HBase 是一个分布式，可扩展，面向列的适合存储海量数据的数据库，其最主要的功能是解决海量数据下的实时随机读写的问题。 通常 HBase 依赖 HDFS
做为底层分布式文件系统，本文以此做前提并展开，详细介绍 HBase
的架构，读路径以及优化实践。

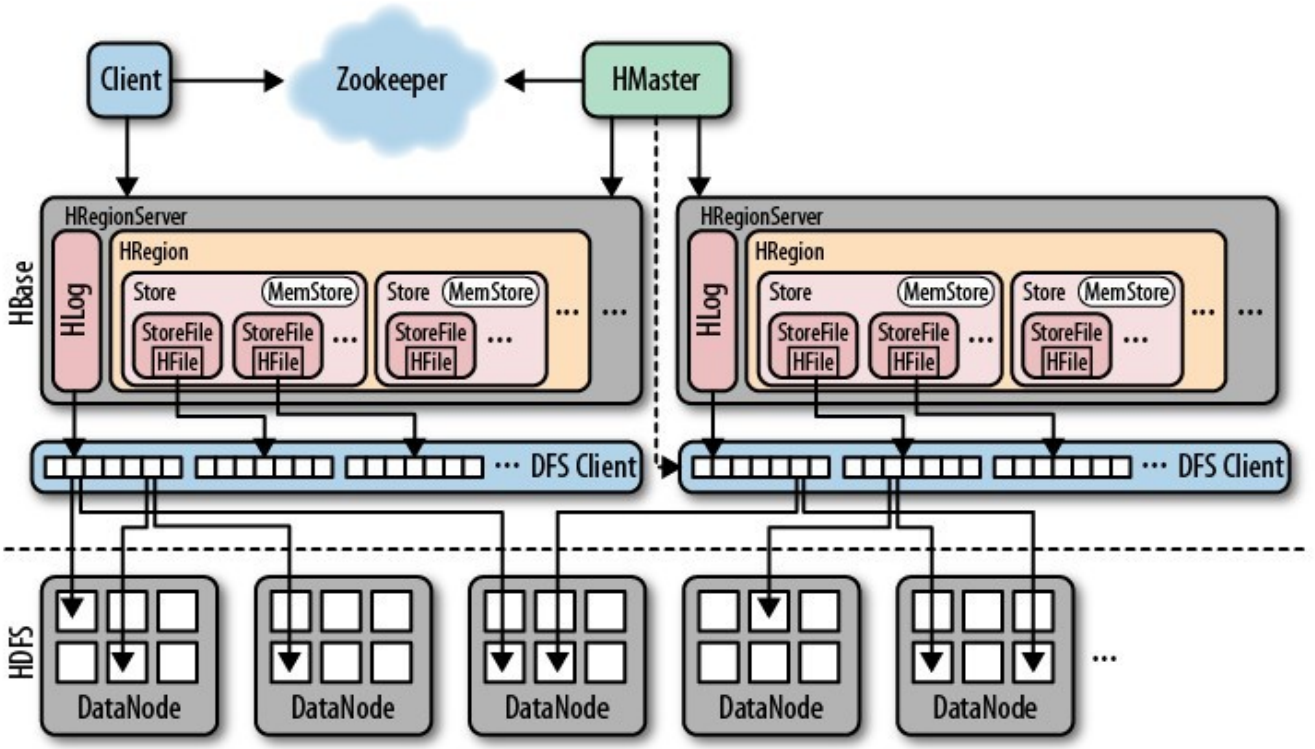
HBase 关键进程

HBase是一个 Master/Slave 架构的分布式数据库，内部主要有 Master， RegionServer
两个核心服务，依赖 HDFS 做底层存储，依赖 zookeeper 做一致性等协调工作。

- Master 是一个轻量级进程，负责所有 DDL 操作，负载均衡， region
  信息管理，并在宕机恢复中起主导作用。
- RegionServer 管理 HRegion，与客户端点对点通信，负责实时数据的读写。
- zookeeper 做 HMaster 选举，关键信息如 meta-region 地址，replication
  进度，Regionserver 地址与端口等存储。

2.2 HBase 架构

首先给出架构图如下



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

架构浅析: HBase 数据存储基于 LSM 架构，数据先顺序写入 HLog，默认情况下 RegionServer 只有一个 Hlog 实例，之后再写入 HRegion 的 MemStore 之中。HRegion 是一张 HBase 表的一块数据连续的区域，数据按照 rowkey 字典序排列，RegionServer 管理这些 HRegion 。当MemStore达到阈值时触发flush操作，刷写为一个 HFile 文件，众多 HFile 文件会周期性进行 minor compaction 和 major compaction 合并成大文件。所有 HFile 与日志文件都存储在HDFS之上。

至此，我们对 HBase 的关键组件和它的角色以及架构有了一个大体的认识，下面重点介绍下 HBase 的读路径。

读路径解析

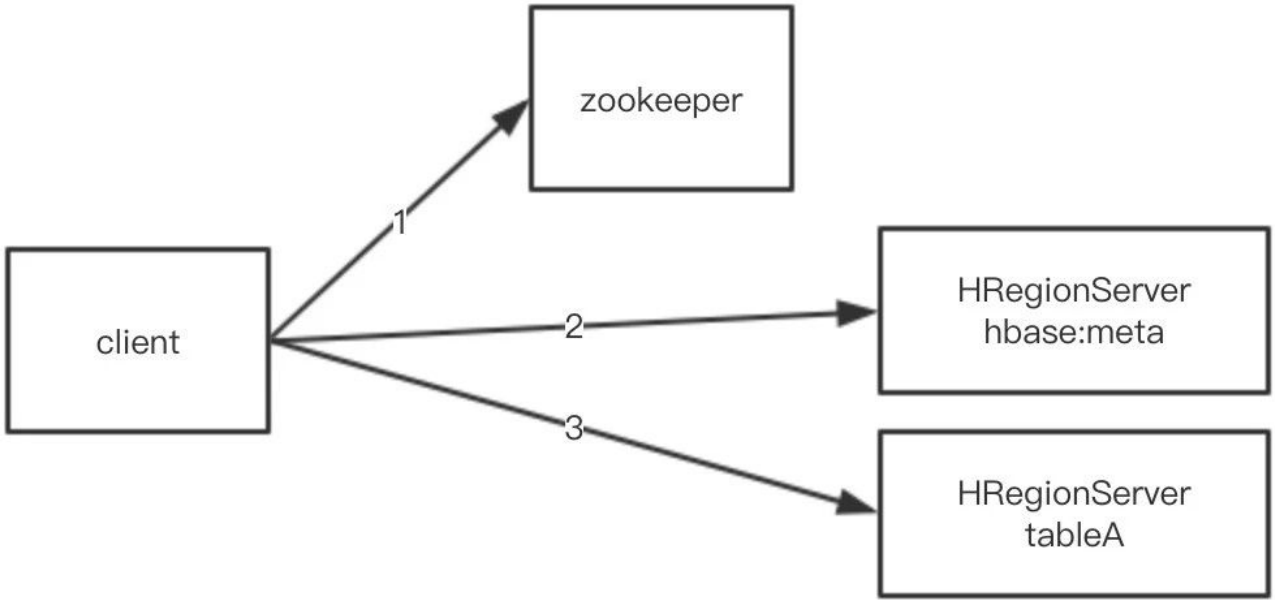
客户端读取数据有两种方式， Get 与 Scan。 Get 是一种随机点查的方式，根据 rowkey 返回一行数据，也可以在构造 Get 对象的时候传入一个 rowkey 列表，这样一次 RPC 请求可以返回多条数据。Get 对象可以设置列与 filter，只获取特定 rowkey 下的指定列的数据。Scan 是范围查询，通过指定 Scan 对象的 startRow 与 endRow 来确定一次扫描的数据范围，获取该区间的所有数据。

一次由客户端发起的完成的读流程，可以分为两个阶段。第一个阶段是客户端如何将请求发送到正确的 RegionServer 上，第二阶段是 RegionServer 如何处理读取请求。

客户端如何发送请求到指定的 RegionServer

HRegion 是管理一张表一块连续数据区间的组件，而表是由多个 HRegion 组成，同时这些 HRegion 会在 RegionServer 上提供读写服务。所以客户端发送请求到指定的 RegionServer 上就需要知道 HRegion 的元信息，这些元信息保存在 hbase:meta 这张系统表之内，这张表也在某一个 RegionServer 上提供服务，而这个信息至关重要，是所有客户端定位 HRegion 的基础所在，所以这个映射信息是存储在 zookeeper 上面。

客户端获取 HRegion 元信息流程图如下：



如果想及时了解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

我们以单条 rowkey 的 Get 请求为例，当用户初始化到 zookeeper 的连接之后，并发送一个 Get 请求时，需要先定位这条 rowkey 的 HRegion 地址。如果该地址不在缓存之中，就需要请求 zookeeper (箭头1)，询问 meta 表的地址。在获取到 meta 表地址之后去读取 meta 表的数据来根据 rowkey 定位到该 rowkey 属于的 HRegion 信息和 RegionServer 的地址(箭头2)，缓存该地址并发 Get 请求点对点发送到对应的 RegionServer(箭头3)，至此，客户端定位发送请求的流程走通。

RegionServer 处理读请求

首先在 RegionServer 端，将 Get 请求当做特殊的一次 Scan 请求处理，其 startRow 和 StopRow 是一样的，所以介绍 Scan 请求的处理就可以明白 Get 请求的处理流程了。

数据组织

让我们回顾一下 HBase 数据的组织架构，首先 Table 横向切割为多个 HRegion

，按照一个列族的情况，每一个 HRegion 之中包含一个 MemStore 和多个 HFile 文件， HFile 文件设计比较复杂，这里不详细展开，用户需要知道给定一个 rowkey 可以根据索引结合二分查找可以迅速定位到对应的数据块即可。结合这些背景信息，我们可以把一个Read请求的处理转化下面的问题：如何从一个 MemStore，多个 HFile 中获取到用户需要的正确的数据（默认情况下是最新版本，非删除，没有过期的数据。同时用户可能会设定 filter ，指定返回条数等过滤条件）。

在 RegionServer 内部，会把读取可能涉及到的所有组件都初始化为对应的 scanner 对象，针对 Region 的读取，封装为一个 RegionScanner 对象，而一个列族对应一个 Store，对应封装为 StoreScanner，在 Store 内部，MemStore 则封装为 MemStoreScanner，每一个 HFile 都会封装为 StoreFileScanner 。最后数据的查询就会落在对 MemStoreScanner 和 StoreFileScanner 上的查询之上。

这些 scanner 首先根据 scan 的 TimeRange 和 Rowkey Range 会过滤掉一些，剩下的 scanner 在 RegionServer 内部组成一个最小堆 KeyValueHeap，该数据结构核心一个 PriorityQueue 优先级队列，队列里按照 Scanner 指向的 KeyValue 排序。

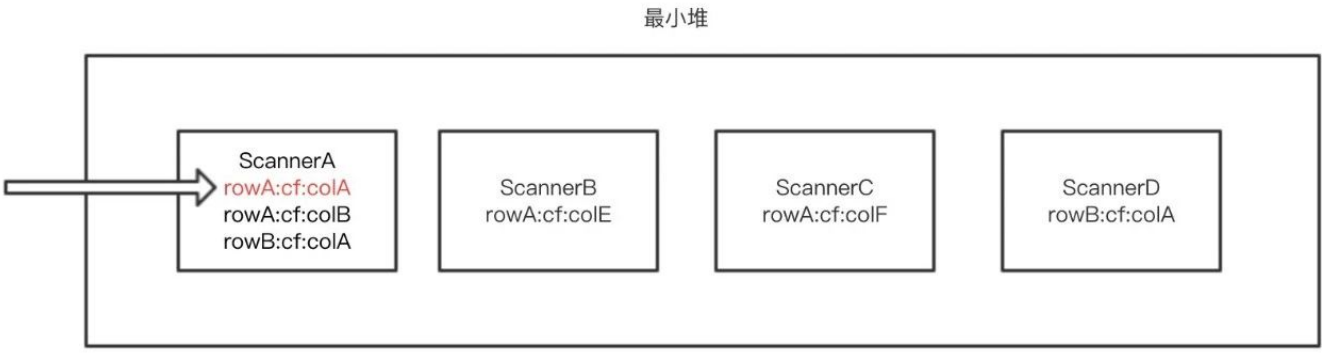
```
// 用来组织所有的Scanner
protected PriorityQueue<KeyValueScanner> heap = null;

// PriorityQueue当前排在最前面的Scanner
protected KeyValueScanner current = null;
```

数据过滤

我们知道数据在内存以及 HDFS 文件中存储着，为了读取这些数据，RegionServer 构造了若干 Scanner 并组成了一个最小堆，那么如何遍历这个堆去过滤数据返回用户想要的值呢。

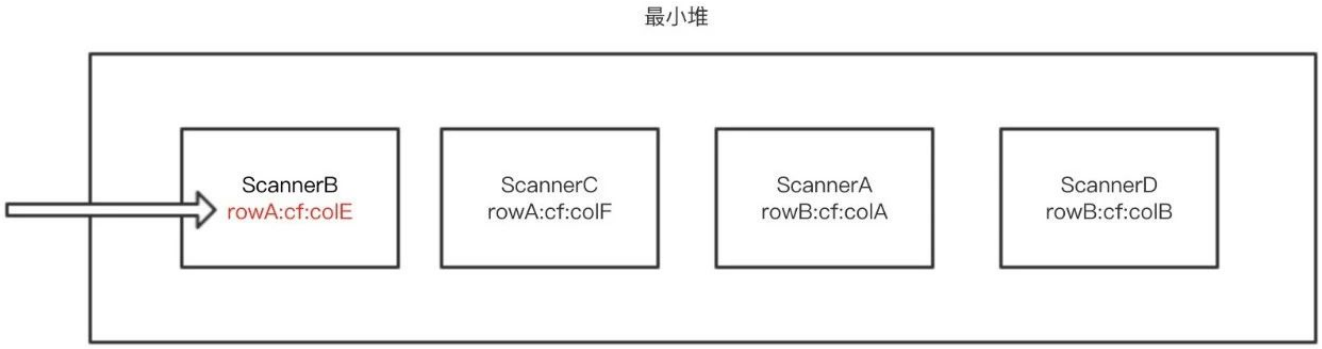
我们假设 HRegion 有4个 Hfile，1个 MemStore，那么最小堆内有4个 scanner 对象，我们以 scannerA-D 来代替这些 scanner 对象，同时假设我们需要查询的 rowkey 为 rowA。每一个 scanner 内部有一个 current 指针，指向的是当前需要遍历的 KeyValue，所以这时堆顶部的 scanner 对象的 current 指针指向的就是 rowA(rowA:cf:colA)这条数据。通过触发 next() 调用，移动 current 指针，来遍历所有 scanner 中的数据。scanner 组织逻辑视图如下图所示。



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

第一次 next 请求，将会返回 ScannerA中的rowA:cf:colA，而后 ScannerA 的指针移动到下一个 KeyValue rowA:cf:colB，堆中的 Scanners 排序不变；

第二次 next 请求，返回 ScannerA 中的 rowA:cf:colB，ScannerA 的 current 指针移动到下一个 KeyValue rowB:cf:ColA，因为堆按照 KeyValue 排序可知 rowB 小于 rowA, 所以堆内部，scanner 顺序发生改变，改变之后如下图所示：



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

scanner 内部数据完全检索之后会 close 掉，或者 rowA
所有数据检索完毕，则查询下一条。默认情况下返回的数据需要经过 ScanQueryMatcher
过滤返回的数据需要满足下面的条件：

- keyValue类型为put
- 列是Scanner指定的列
- 满足filter过滤条件
- 最新的版本
- 未删除的数据

如果 scan 的参数更加复杂，条件也会发生变化，比如指定 scan 返回 Raw 数据的时候，打了删除

标记的数据也要被返回，这部分就不再详细展开，至此读流程基本解析完成，当然本文介绍的还是很粗略，有兴趣的同学可以自己研究这一部分源码。

读优化

在介绍读流程之后，我们再结合有赞业务上的实践来介绍如何优化读请求，既然谈到优化，就要先知道哪些点可会影响读请求的性能，我们依旧从客户端和服务端两个方面来深入了解优化的方法。

客户端层面

HBase 读数据共有两种方式，Get 与 Scan。

在通用层面，在客户端与服务端建连需要与 zookeeper 通信，再通过 meta 表定位到 region 信息，所以在初次读取 HBase 的时候 rt
都会比较高，避免这个情况就需要客户端针对表来做预热，简单的预热可以通过获取 table 所有的 region 信息，再对每一个 region 发送一个 Scan 或者 Get 请求，这样就会缓存 region 的地址；

rowkey
是否存在读写热点，若出现热点则失去分布式系统带来的优势，所有请求都只落到一个或几个 HRegion 上，那么请求效率一定不会高；

读写占比是如何的。如果写重读轻，浏览服务端 RegionServer 日志发现很多 MVCC STUCK 这样的字样，那么会因为 MVCC 机制因为写 Sync 到 WAL
不及时而阻塞读，这部分机制比较复杂，考虑之后分享给大家，这里不详细展开。

Get 请求优化

- 将 Get 请求批量化，减少 rpc 次数，但如果一批次的 Get 数量过大，如果遇到磁盘毛刺或者 Split 毛刺，则 Get 会全部失败（不会返回部分成功的结果），抛出异常。
- 指定列族，标识符。这样可以服务端过滤掉很多无用的 scanner，减少 IO 次数，提高效率，该方法同样适用于 Scan。

Scan 请求优化

- 设定合理的 startRow 与 stopRow 。如果 scan 请求不设置这两个值，而只设置 filter，则会做全表扫描。
- 设置合理的 caching 数目， scan.setCaching(100)。 因为 Scan 潜在会扫描大量数据，因此客户端发起一次 Scan 请求，实际并不会一次就将所有数据加载到本地，而是分成多次 RPC 请求进行加载。默认值是100。用户如果确实需要扫描海量数据，同时不做逻辑分页处理，那么可以将缓存值设置到1000，减少 rpc 次数，提升处理效率。如果用户需要快速，迭代地获取数据，那么将 caching

设置为50或者100就合理。

服务端优化

相对于客户端，服务端优化可做的比较多，首先我们列出有哪些点会影响服务端处理读请求。

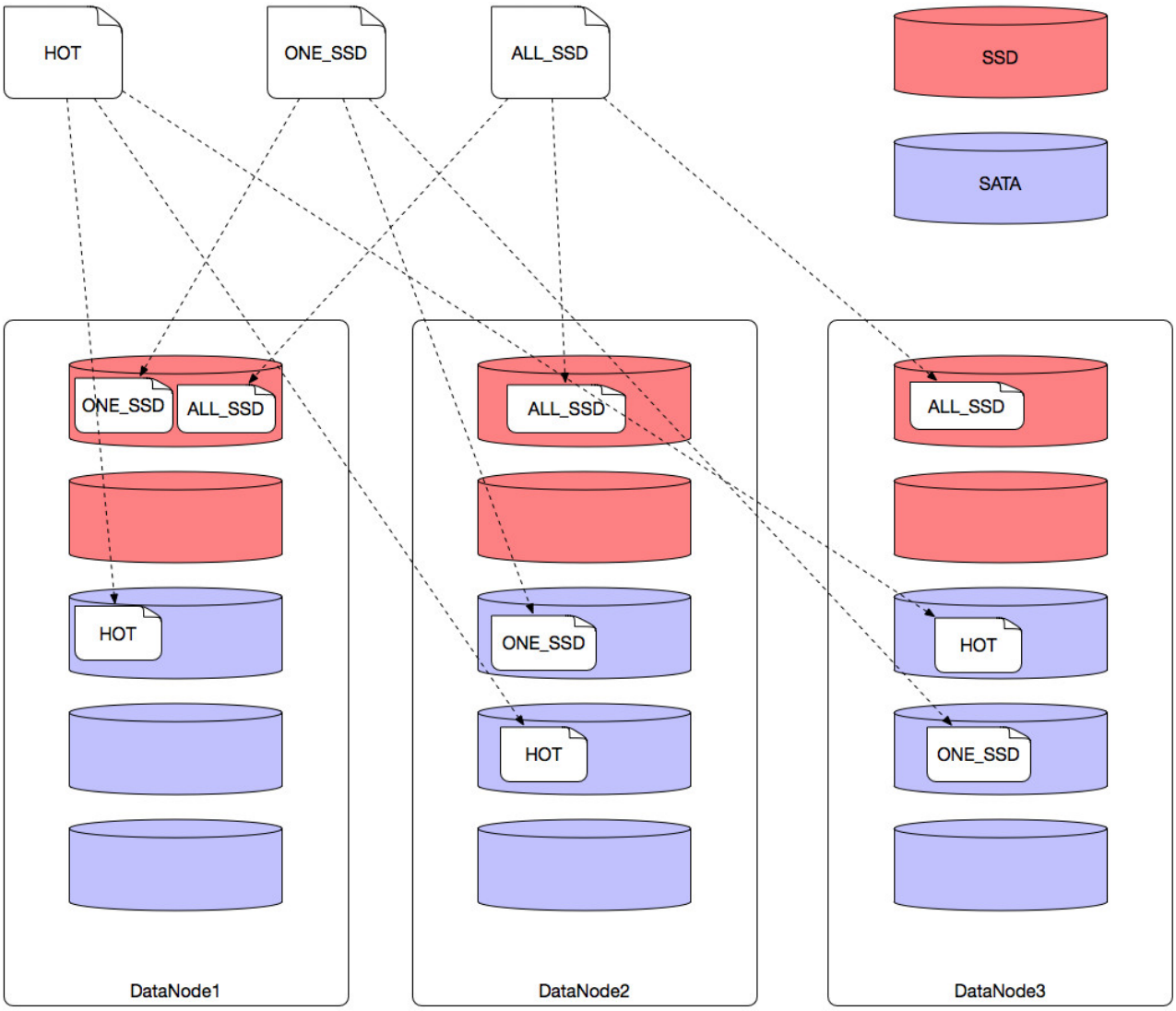
- gc 毛刺
- 磁盘毛刺
- HFile 文件数目
- 缓存配置
- 本地化率
- Hedged Read 模式是否开启
- 短路读是否开启
- 是否做 高可用

gc 毛刺没有很好的办法避免，通常 HBase 的一次 Young gc 时间在 20~30ms
之内。磁盘毛刺发生是无法避免的，通常 SATA 盘读 IOPS 在 150 左右，SSD 盘随机读在 30000
以上，所以存储介质使用 SSD 可以提升吞吐，变向降低了毛刺的影响。HFile 文件数目因为 flush
机制而增加，因 Compaction 机制减少，如果 HFile 数目过多，那么一次查询可能经过更多 IO
，读延迟就会更大。这部分调优主要是优化 Compaction 相关配置，包括触发阈值，Compaction
文件大小阈值，一次参与的文件数量等等，这里不再详细展开。读缓存可以设置为为
CombinedBlockCache，调整读缓存与 MemStore
占比对读请求优化同样十分重要，这里我们配置 hfile.block.cache.size 为
0.4，这部分内容又会比较艰深复杂，同样不再展开。下面结合业务需求讲下我们做的优化实践。

我们的在线集群搭建伊始，接入了比较重要的粉丝业务，该业务对RT要求极高，为了满足业务需求我们做了如下措施。

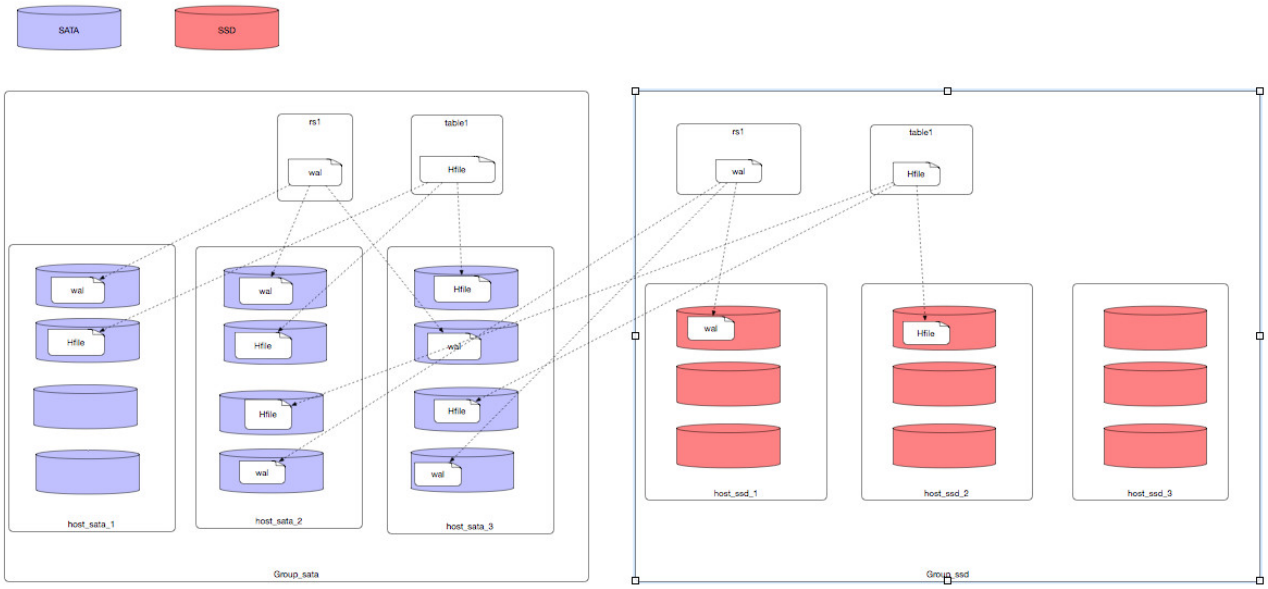
异构存储

HBase 资源隔离+异构存储。SATA 磁盘的随机 iops 能力，单次访问的
RT，读写吞吐上都远远不如 SSD，那么对RT极其敏感业务来说，SATA盘并不能胜任，所以我们需要HBase有支持SSD存储介质的能力。
为了 HBase 可以支持异构存储，首先在 HDFS 层面就需要做响应的支持，在 HDFS 2.6.x
以及之后的版本，提供了对SSD上存储文件的能力，换句话说在一个 HDFS
集群上可以有SSD和SATA磁盘并存，对应到 HDFS 存储格式为 [ssd] 与 [disk]。然而 HBase 1.2.6
上并不能对表的列族和 RegionServer 的 WAL 上设置其存储格式为 [ssd]， 该功能在社区 HBase
2.0 版本之后才开放出来，所以我们从社区 backport 了对应的 patch ，打到了我们有赞自己的
HBase 版本之上。支持 [ssd] 的 社区issue 如下：
https://issues.apache.org/jira/browse/HBASE-14061?jql=text%20~%20%22storage%20policy%22 。
添加 SSD 磁盘之后，HDFS 集群存储架构示意图如下图所示：



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

理想的混合机型集群异构部署，对于 HBase 层面来看，文件存储可选三种策略：HOT, ONE_SSD, ALL_SSD,其中 ONE_SSD
存储策略既可以把三个副本中的两个存储到便宜的SATA磁盘介质之上来减少 SSD
磁盘存储成本的开销，同时在数据读取访问本地 SSD 磁盘上的数据可以获得理想的 RT
，是一个十分理想的存储策略。HOT 存储策略与不引入异构存储时的存储情况没有区别，而 ALL_SSD 将所有副本都存储到 SSD 磁盘上。 在有赞我们目前没有这样的理想混合机型，只有纯 SATA 与 纯 SSD
两种大数据机型，这样的机型对应的架构与之前会有所区别，存储架构示意图如图 6 所示。



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

基于这样的场景，我们做了如下规划：

- 将SSD机器规划成独立的组，分组的 RegionServer 配置 hbase.wal.storage.policy=ONE_SSD, 保证 wal 本身的本地化率；
- 将SSD分组内的表配置成 ONE_SSD 或者 ALL_SSD；
- 非SSD分组内的表存储策略使用默认的 HOT

具体的配置策略如下：在 hdfs-site.xml 中修改

```
 <property>
     <name>dfs.datanode.data.dir</name>
     <value>[SSD]file:/path/to/dfs/dn1</value>
</property>
```

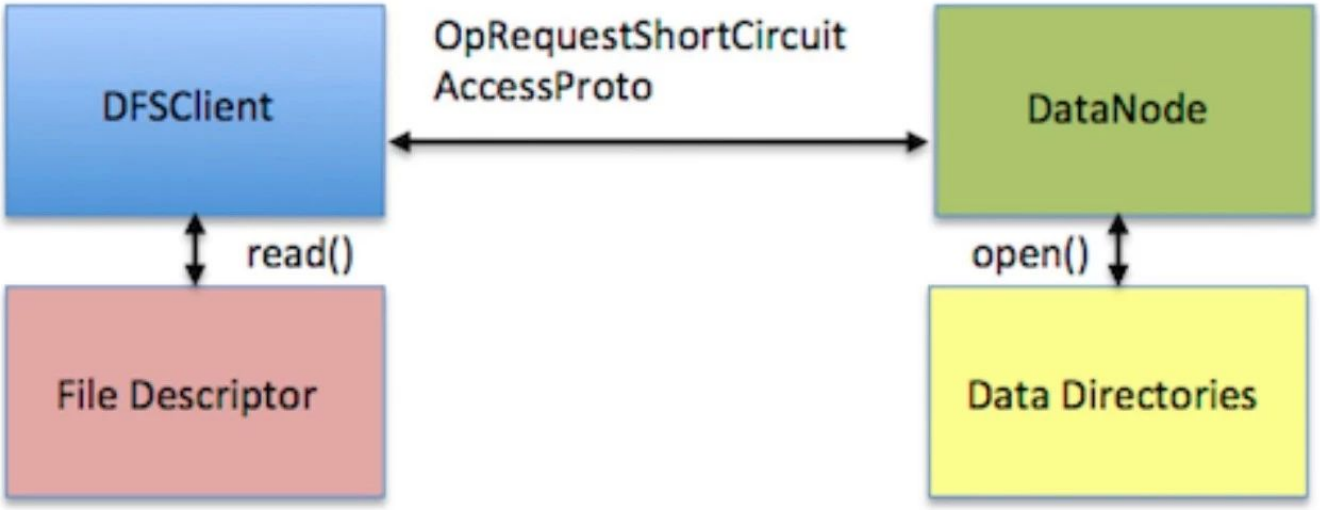
在 SSD 机型 的 RegionServer 中的 hbase-site.xml 中修改

```
<property>
     <name>hbase.wal.storage.policy</name>
     <value>ONE_SSD</value>
 </property>
```

其中 ONE_SSD 也可以替代为 ALL_SSD。 SATA 机型的 RegionServer 则不需要修改或者改为 HOT。

HDFS 短路读

开启 HDFS 的短路读模式。该特性由 HDFS-2246 引入。我们集群的 RegionServer 与 DataNode 混布，这样的好处是数据有本地化率的保证，数据第一个副本会优先写本地的 Datanode。在不开启短路读的时候，即使读取本地的 DataNode 节点上的数据，也需要发送RPC 请求，经过层层处理最后返回数据，而短路读的实现原理是客户端向 DataNode 请求数据时，DataNode 会打开文件和校验和文件，将两个文件的描述符直接传递给客户端，而不是将路径传递给客户端。客户端收到两个文件的描述符之后，直接打开文件读取数据，该特性是通过 UNIX Domain Socket 进程间通信方式实现，流程图如图 7 所示。



如果想及时了解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

该特性内部实现比较复杂，设计到共享内存段通过 slot 放置副本的状态与计数，这里不再详细展开。

开启短路读需要修改 hdfs-site.xml 文件：

```
<property>
    <name>dfs.client.read.shortcircuit</name>
    <value>true</value>
</property>

<property>
    <name>dfs.domain.socket.path</name>
    <value>/var/run/hadoop/dn.socket
</value>
```

</property>

HDFS Hedged Read

开启 Hedged Read 模式。当我们通过短路读读取本地数据因为磁盘抖动或其他原因读取数据一段时间内没有返回，去向其他 DataNode 发送相同的数据请求，先返回的数据为准，后到的数据抛弃，这也可以减少磁盘毛刺带来的影响。默认该功能关闭，在 HBase 中使用此功能需要修改 hbase-site.xml

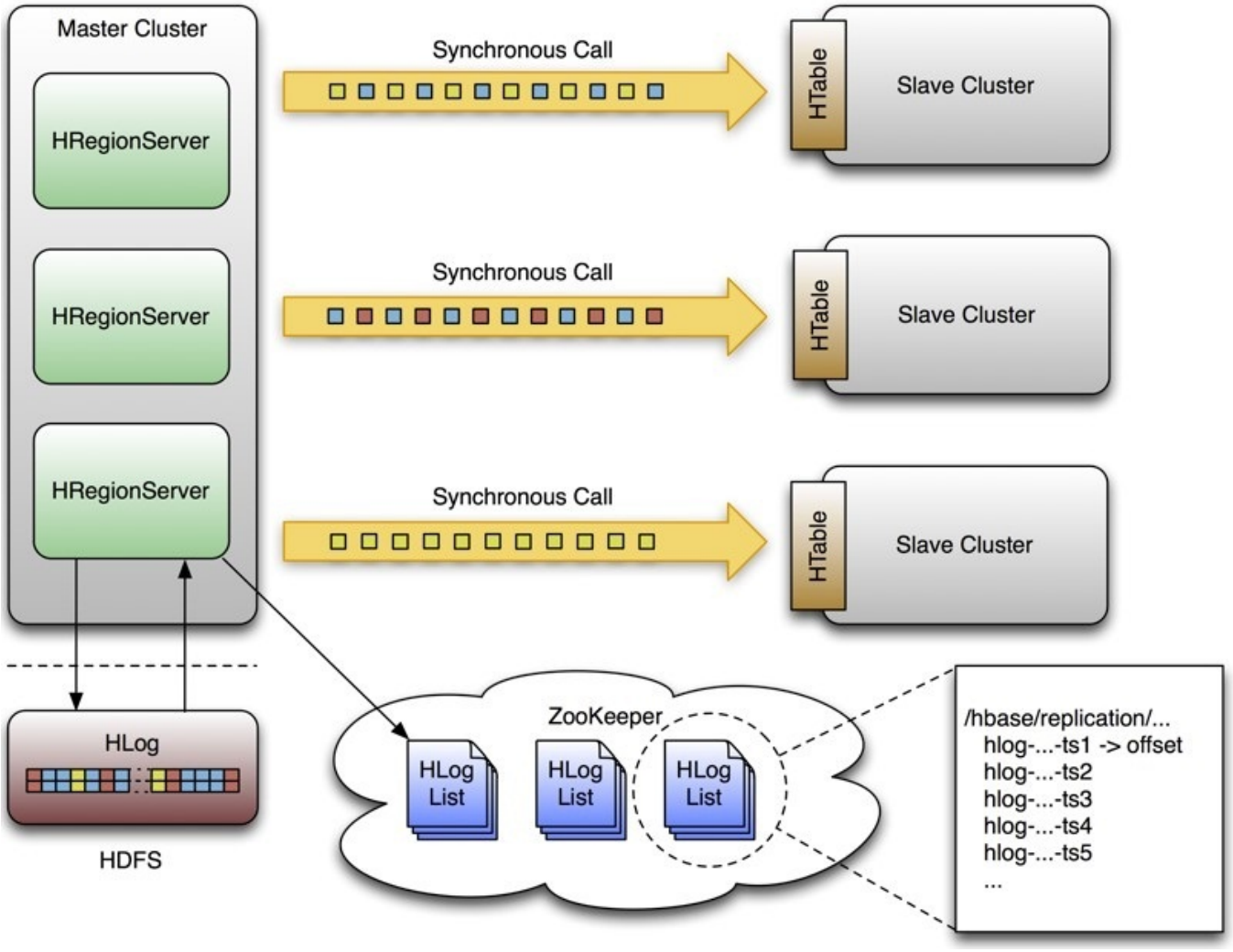
```
<property>
    <name>dfs.client.hedged.read.threadpool.size</name>
    <value>50</value>
</property>

<property>
    <name>dfs.client.hedged.read.threshold.millis</name>
    <value>100</value>
</property>
```

线程池大小可以与读handler的数目相同，而超时阈值不适宜调整的太小，否则会对集群和客户端都增加压力。同时可以通过 Hadoop 监控查看 hedgedReadOps 与 hedgedReadOps 两个指标项，查看启用 Hedged read 的效果，前者表示发生了 Hedged read 的次数，后者表示 Hedged read 比原生读要快的次数。

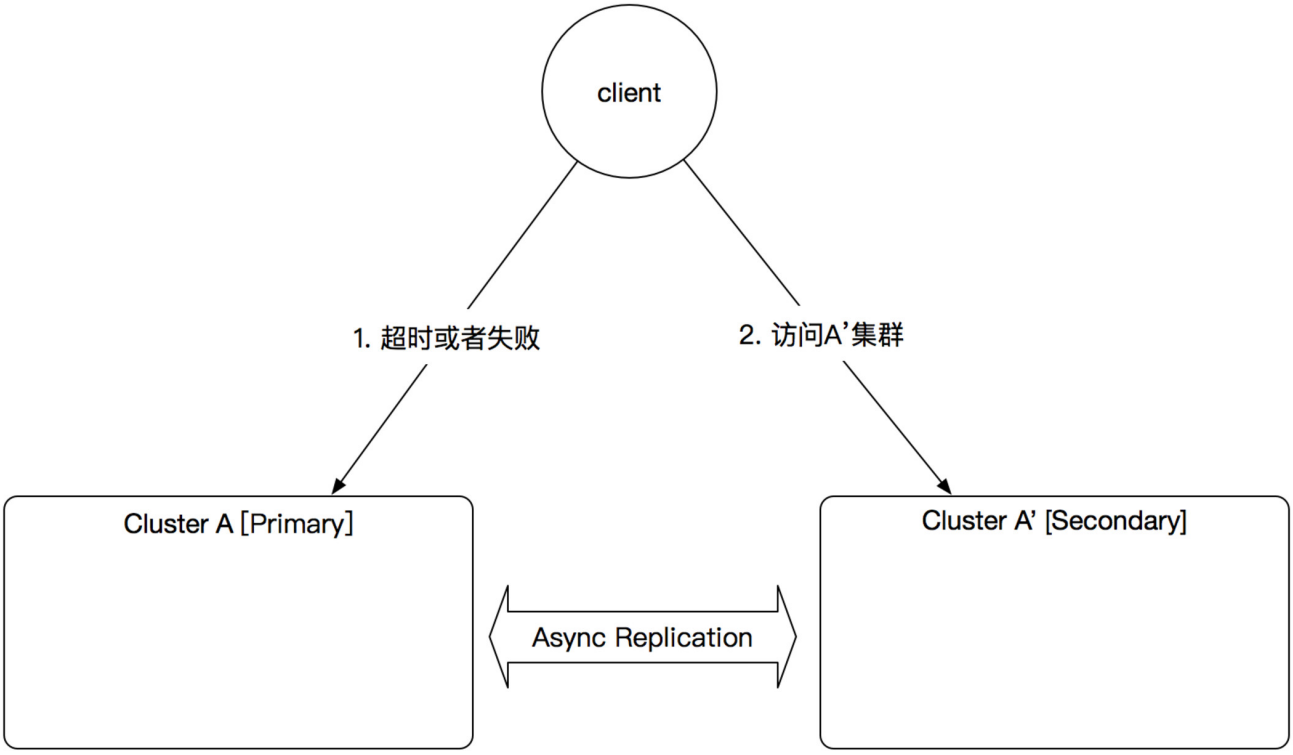
高可用读

高可用读。HBase 是一个 CP 系统，同一个 region 同一时刻只有一个 regionserver 提供读写服务，这保证了数据的一致性，即不存在多副本同步的问题。但是如果一台 regionserver 发声宕机的时候，系统需要一定的故障恢复时间 deltaT, 这个 deltaT 时间内，region 是不提供服务的。这个 deltaT 时间主要由宕机恢复中需要回放的 log 的数目决定。集群复制原理图如下图 8 所示。



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

HBase 提供了 HBase Replication
机制，用来实现集群间单方向的异步数据复制我们线上部署了双集群，备集群 SSD 分组和主集群 SSD 分组有相同的配置。当主集群因为磁盘，网络，或者其他业务突发流量影响导致某些 RegionServer
甚至集群不可用的时候，就需要提供备集群继续提供服务，备集群的数据可能会因为 HBase Replication
机制的延迟，相比主集群的数据是滞后的，按照我们集群目前的规模统计，平均延迟在 100ms 以内。所以为了达到高可用，粉丝业务可以接受复制延迟，放弃了强一致性，选择了最终一致性和高可用性，在第一版采用的方案如下：



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

粉丝业务方是不想感知到后端服务的状态，也就是说在客户端层面，他们只希望一个 Put 或者 Get 请求正常送达且返回预期的数据即可，那么就需要高可用客户端封装一层降级，熔断处理的逻辑，这里我们采用 Hystrix 做为底层熔断处理引擎，在引擎之上封装了 HBase 的基本 API，用户只需要配置主备机房的 ZK 地址即可，所有的降级熔断逻辑最终封装到 ha-hbase-client 中，原理类似图9，这里不再赘述。

预热失败问题修复

应用冷启动预热不生效问题。该问题产生的背景在于应用初始化之后第一次访问 HBase 读取数据时候需要做寻址，具体流程见图2，这个过程涉及多次 RPC 请求，所以耗时较长。在缓存下所有的 Region 地址之后，客户端与 RegionServer 就会做点对点通信，这样 RT 就有所保证。所以我们会在应用启动的时候做一次预热操作，而预热操作我们通常做法是调用方法 getAllRegionLocations 。在1.2.6版本（后来经过笔者调研，1.3.x，以及2.x版本）getAllRegionLocations 存在 bug ，该方案预期返回所有的 Region locations 并且缓存这些 Region 地址，但实际上，该方法只会缓存 table 的第一个 Region，笔者发现此问题之后反馈给社区，并提交了 patch 修复了此问题，issue连接:https://issues.apache.org/jira/browse/HBASE-20697?filter=-2 。这样通过调用修复 bug 之后的 getAllRegionLocations 方法，即可在应用启动之后做好预热，在应用第一次读写HBase时便不会产生 RT 毛刺。

粉丝业务主备超时时间都设置为 300ms。经过这些优化，其批量 Get 请求 99.99% 在 20ms

以内，99.9999% 在 400ms 以内。

总结

HBase 读路径相比写路径更加复杂，本文只是简单介绍了核心思路。也正是因为这种复杂性，在考虑优化的时候需要深入了解其原理，且目光不能仅仅局限于本身的服务组件，也要考虑其依赖的组件，是否也有可优化的点。最后，本人能力有限，文中观点难免存在纰漏，还望交流指正。

本文原文：https://mp.weixin.qq.com/s/cj-HJNfZ2O7kCAFNL4l7Eg