# HBase服务化实践

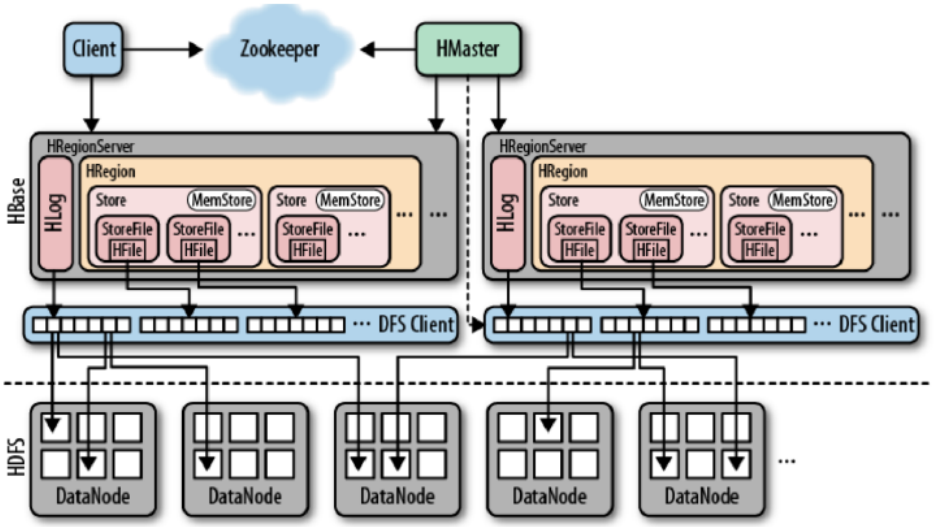
小米科技 云平台存储组

2015/11/07

主持人：好我们现在开始下一场，下面是来自小米的何亮亮，何亮亮是 09 年毕业于中国科学院，2013 加入小米基础架构组，先后负责小米结构化存储服务设计与开发，Hadoop 和 HBase 开发与维护等工作。他带来的报告主题是，HBase 服务化实践，我们掌声欢迎何亮亮。

何亮亮：各位朋友大家好，我来自小米云平台存储组，我们组主要负责小米公共存储服务的开发与支持，小米是 HBase 的一个重度用户，我今天主要分享一下我们在使用 HBase 过程中遇到的一些问题，以及我们的一些实战经验。这也是今天的主要内容，因为时间关系我可能会加快速度，如果有什么问题我们可以线下交流。

## HBase架构回顾




首先我们回顾一下 HBase 的架构，HBase 是基于谷歌的 Bigtable 论文，它主要是由 HMaster 和 Regionserver 这两个组件组成。它底层的数据存储在 HDFS 上边，Zookeeper 负责各个组建之间的协调工作。

## 典型业务

在线业务以及相关的离线分析：

- 小米云服务
- 小米推送服务
- 米聊消息全存储
- 多看阅读
- 各种智能设备

接下来讲一下 HBase 在小米的应用现状，HBase 在小米主要是用在 OLTP 的场合，以及相关的一些离线分析，典型的应用包括小米云服务，包括大家常用的通话记录、短信、云相册等。这些服务的结构化数据存储在 HBase 上面。第二个应用就是小米消息推送服务，主要面向移动应用开发者。其他还有多看阅读以及小米各种设备的数据。

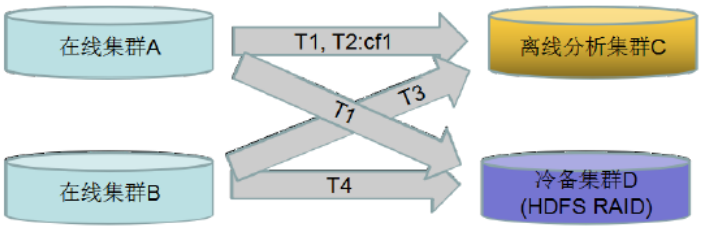
## 应用规模

- 10+个在线集群，5个离线集群，若干测试集群
- 5个数据中心
- 数百台机器：每个数据节点 12 * 2T
- 公司内外数十个业务
- 单机群峰值QPS百万级别

这是现在的一个应用规模，我们现在大概有 10 多个在线集群 5 个离线集群还有若干个测试集群，分布在 5 个数据中心，目前有几百台机器，典型的数据结点是 12*2T 配制，目前服务了公司内外的数十个业务，最大集群 QPS 大概在的百万级别。

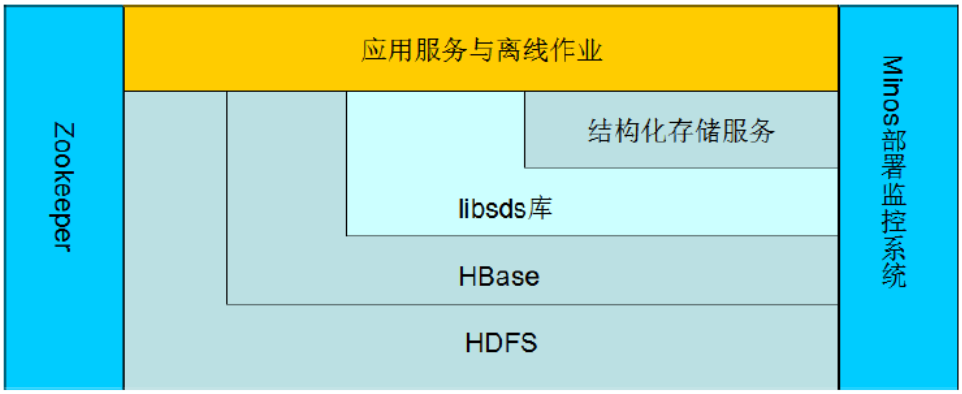
## 典型部署：主备集群同步

- 细粒度同步策略：HBASE-8751
  - 表/CF级别同步设置，按需同步，节约空间和带宽




这是我们一个典型的部署情况，在线集群主要服务线上业务，然后它的数据通过日志的方式，同步到离线集群和冷备集群，其中离线集群是用来做一些离线的分析。冷被集群主要考虑到数据安全，所以我们做了一个数据的备份。考虑到存储成本我们是采用了 HDFS RAID，这样可以大大减少存储上面的成本，目前 RAID 的配制是 6+3。同时我们增加了细粒度的同步设置支持，这样可以支持表级别和 CF 级别同步设置，这样只需要同步必要的一些数据，节省空间和带宽成本。

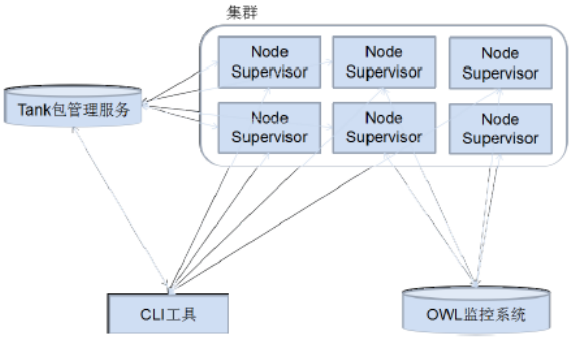
## 典型部署




这是我们一个典型的部署，除了 Zookeeper、HBase、HDFS 这三个必备的模块之外，我们还在 HBase 原生接口基础上，开发了结构化存储库 libsds，以及结构化存储服务。除此之外，我们还开发的自己的部署运营系统 Minos，接下来我会分别介绍一下这三方面的工作。

## 部署监控报警工具Minos[1]

- 支持HDFS，Zookeeper，HBase，YARN，Hive等服务
- 可灵活扩展支持其他服务
- HBase监控支持：表级别监控数据聚合




- Tank：包管理服务
- OWL：统计监控服务
- Supervisor：节点Agent
- Client：CLI工具

[1]已开源：https://github.com/xiaomi/Minos

前面百度马老师也吐槽了 Hadoop，Hadoop 的默认部署是非常难用的，不太适合大规模的生产环境。我们就开发了我们自己的 Hadoop 集群管理系统，可以实现 Hadoop 系列软件的部署与管理。Minos 内建监控的支持，对于 HBase 我们增加了一些表级别的监控数据的聚合。这是 Minos 的一个整体架构，主要包含了四个组件，包管理服务，统计监控服务、结点上面的 Agent 负责进程的生命周期管理，配制管理等，以及运维工具，目前这个项目已经开源，如果大家有兴趣，可以在 Github 上面下载试用。

## 测试上线流程

- 单元测试
- 压力测试，性能测试
- Staging集群测试：业务测试
- Longhaul测试
- Failover测试(Netflix ChaosMonkey，HBASE-9802)
  - 可配置/随机选择Action
  - Pre/Post数据正确性验证
  - 错误可重放

这是我们的一个上线的流程，我们发布新版本的时候，大概经过单元测试、压力测试和 Staging 集群的测试。除了 3 个测试之外，我们还有 Longhaul 测试和 Failover 测试，这样可以帮助我们尽快发现问题。

## 辅助工作

其他辅助工作：

- HBase Nameservice支持
- Snapshot管理系统
- HBase错误日志分析统计作业
- Kerberos账号管理系统

除此以外我们还做了一些其他的辅助工作，包括 HBase 的 Nameservice 支持，

Snapshot 管理，HBase 错误日志统计分析，Kerberos 帐号管理等工具。

## 典型故障

- 坏盘
- 慢盘
  - HDFS-5776 Hedged Read
  - 第一次读取超时时，请求第二个DN，降低整体延迟
- GC问题
  - JVM参数
  - 多实例部署，减少堆大小
- 程序Bug(Zookeeper/HDFS/HBase)
  - 可用性监控
  - 错误日志统计分析作业
- 网络故障

这是我们遇到一些典型故障，首先最常见的是坏盘，这是个问题是在 HDFS 层面解决。

第二个问题是相对影响比较大的，就是慢盘的问题，我们在 HDFS 层面开发了一个叫 Hedged Read 的特性，就是在第一次读取超时的时候，同时向第二个 Datanode 发请求，从而降低整个请求的延时。

第三个就是用 JVM 问题，HBase、HDFS 这些都是里 Java 实现的，所以说 GC 问题是难以避免的，我们主要是做两方面个工作，一个是调节 JVM 参数，另外一个就是我们在采用单机多实例的部署方式，这样就是减少 JVM 堆的大小，这样可以缓解 GC 的影响。

第四个问题是程序 Bug，在我们使用的过程中，也遇到过一些包括（Zookeeper、HBase、HDFS）的 Bug，这个是比较难避免的。所以我们做了两方面工作，一个是可用性监控，另外一个就是刚才提到的错误日志的统计分析，这样我们可以尽早的发现一些异常的情况，然后去排查。最后一类问题节是网络故障，网络故障发生的比较少，但是影响比较严重，会导致集群整体宕机，需要重视。

接下来我讲一下我们在开发工具方面所做的一些工作。

## 数据模型

多数业务数据模型统一：

- 数据由大量独立的用户(或设备等某种实体)产生
- 业务一般需要在用户实体范围内支持跨行原子性和二级索引

**改进1：数据分布的保证**

基于正则表达式的前缀分割策略：RegexPrefixRegionSplitPolicy

**改进2：Region内部的跨行原子性支持**

现状：同一次batch操作的同Region跨行写没有原子性保证
改进：同一次batch操作的同Region的所有写在获得所有行的锁后一次落地，确保按照rowkey顺序抢锁、避免死锁

首先小米这边的数据模型相对来说是比较统一的，因为它有两个特点，一个是它的数据是由底大量的独立用户产生，另外一个就是业务一般就是需要在用户实体范围内支持跨行原子性和二级索引。同时一般没有全局事务和全局索引的需求。对上面的特点我们对 HBase 做了两个改进，一个就是基于正则表达式的前缀分割策略，这样就能保证相同前缀向记录能够分在同一个 Region 上。第二个改进就是实现 Region 内部的跨行原子性的支持，对于 batch 操作，保证修改在同一条 WALEdit 落地，并按照 Rowkey 顺序加锁的方式保证原子性。顺序加锁能够避免死锁的问题。

## 数据模型

最初的应用通过原生HBase Client开发：

- 用户学习成本高
- 用户实现数据类型，数值类型一般转成字符串以保持先后顺序，有额外开销
- 客户端实现索引，checkAndPut开销大，功能不完备
- 代码冗余，开发效率较低，易出错

解决方案：结构化存储库libsds(Structured Datastore Library)

- 内建数据类型支持
- 规范化的数据模型
  - Rowkey前缀 - 实体组键，支持哈希分布，消除热点
  - Rowkey后缀 - 主键
  - Column Qualifier - 数据列
- 多种类型的局部索引[2]
  - Eager索引：更新删除数据时清除失效索引
  - Lazy索引：读取时判断索引有效性，更新时不做额外操作
  - Immutable索引：适合只读数据一次性写入，读写均无需额外判断

[2]局部：实体组内部，即Region内部



最开始我们的应用都是通过 HBase 原生 Client 来开发的，当中也遇到了一些问题，就比如说学习成本，还有就是用户需要自己实现数据类型，实现的时候，会有一些额外的开销。另外是从客户端实现索引，通过 checkndPut 这种方式，开销比较大，另外实现起来也比较复杂，功能也不是完备。最后最重要的一点，业务层的代码大量冗余，这样一方面开发效率低，另一方面很难保证代码质量。我们解决的方案就是，在 HBase Client 基础上开发了一个结构化存储库，提供了内建的数据类型支持和局部二级索引支持(Region 内部)。

我们提供了三种索引，第一种是 Eager 索引，这种索引在更新和删除数据的时候，就清除失效索引，读取时无需额外操作，索引一定是有效的。第二种是 Lazy 索引，这种索引在读取的时候判断是不是有效的，在数据更新时不做额外操作。第三种 Immutable 索引是结合了上面两种索引特点，适合只读性数据一次性写入的情景，这样读取和写入都没有额外的开销。

## 数据类型支持 – 顺序保持编码

参考了HBASE-8201和SQLite4编码方案[3]

- 数值类型编码：反转符号位
- 二进制类型编码：对字节序列进行变长编码，每字节第一位存储编码结束标志，1表示有效编码数据，0表示编码结束，剩余7位用来编码原数据
- 支持逆序编码：按位取反

二进制类型示例：

| 值 | 原始二进制表示 | 编码结果 |
| --- | --- | --- |
| 0x61('a')<br>0x61 0x62('ab') | 0110 0001<br>0110 0001 0110 0010 | 1011 0000 1100 0000 0000 0000<br>1011 0000 1101 1000 1100 0000 0000 0000 |

优点：保持编码后二进制值大小顺序不变，兼容HBase存储格式和接口

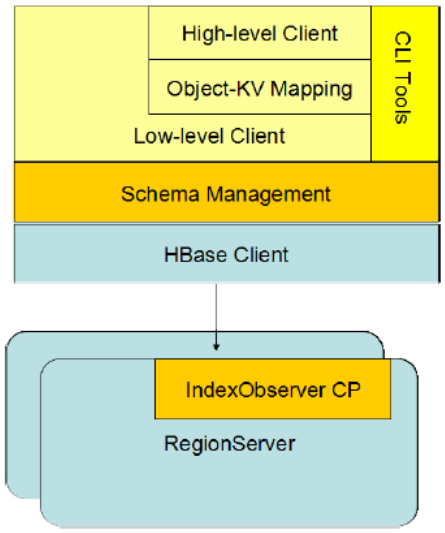
[3]http://sqlite.org/src4/doc/trunk/www/key_encoding.wiki

接下来简单介绍一下数据类型的支持。我们参考了 HBase 的实现和 SQLite 的编码方案，对于数值类型，采取符号位反转方案，对于二进制类型，采取的方案是，由 8 比特编码转换成 7 比特变长编码，最高位为标志位。1 表示是否是有效的编码数据，0 表述编码结束。同时编码方案还支持逆序编码，采用的方案是按位取反，可用于逆序索引等场合。这种编码方案的优点就是保持编码后的二进制数据，跟原来的值大小关系是不变的，这样可以兼容 HBase 的存储格式和访问接口。

## 结构化存储库libsds

主要功能：

- 数据类型与索引支持
- Object-KeyValue Mapping支持(对应ORM)
- CLI工具




这是 libsds 的整体结构。其中索引通过 HBase 的 Coprocessor 来实现。除基本的访问接口之外，还提供了 Object-KeyValue Mapping，可实现业务层 Java 对象到 HBase KeyValue 的自动映射。下面的例子演示了通过 libsds Annotation 方式定义 Schema 和访问数据的方式。

## libsds示例–记事本应用：数据类型

```
@Record(table = "note", family = "B")
public class Note {
  @Column(keyOnly = true)
  String uid; // 用户ID

  @Column(keyOnly = true)
  Long id; // 笔记ID

  @Column
  String title;

  @Column(serialization = Column.SerializationType.UNIX_TIME)
  private Date mtime;

  @Column(collection = true, elementClass = String.class, type =
  private Set<String> tags;

  @Column(serialization = Column.SerializationType.JSON)
  private NoteBody body;
```

## libsds示例–记事本应用：主数据表结构

```
@Table(name = "note",
      version = 1,
      splits = 16,
      entityGroup = @EntityGroup(columns = {
          @Key(column = "uid")
        },
        hash = true),
      primaryIndex = @PrimaryIndex(
        family = "B",
        columns = {
          @Key(column = "id")
        }),
      atomicBatch = true,
      families = {
        @ColumnFamily(name = "B"),
        @ColumnFamily(name = "D"),
        @ColumnFamily(name = "I", inMemory = true)
      }
```

## libsds示例–记事本应用：索引结构

```
secondaryIndexes = {
  @LocalSecondaryIndex(
    name = Constants.IDX_MTIME,
    family = "I",
    columns = {
      @Key(column = "mtime", order = SortOrder.DESC)
    },
    mode = SecondaryIndexConsistencyMode.EAGER,
    projections = { "title" }
  ),
  @LocalSecondaryIndex(
    name = Constants.IDX_TAG,
    family = "I",
    columns = {
      @Key(column = "tags")
    }
  )
},
```

## libsds示例–记事本应用：查询操作

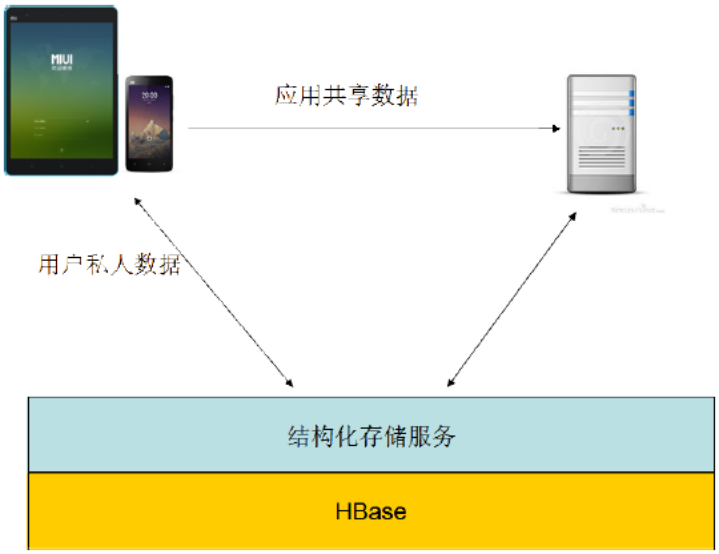
```
// 查询操作 SELECT * FROM note
//            WHERE uid=userId AND title LIKE 'Test%'
//            ORDER BY mtime DESC
//            LIMIT N
@Override public List<ListViewItem> searchNLatestItems(
    String userId, int N, String title) {
  return typedAccessClient.scan(Note.class,
         ListViewItem.class,
         Constants.IDX_MTIME, // 查询时显示指定索引
         Note.entityGroupNote(userId),
         Note.entityGroupNote(userId),
         "title REGEX '" + match + "'",//title REGEX 'Test.*'
         N).getRecords();
}
// 随机查询
@Override public Note findNoteById(String userId, long nid) {
  Note key = new Note(userId, nid, null, null, null, ...);
  return typedAccessClient.get(key);
}
```

## libsds示例–记事本应用：更新操作

```
// 更新操作 UPDATE note SET version = oldVersion + 1,
//                  mtime = NOW, contents = '...'
//            WHERE version = oldVersion
//                AND uid = userId AND id = noteId
@Override public boolean updateNote(Note note) {
  int currentVersion = note.getVersion();
  try {
    SimpleCondition versionPredicate =
        SimpleCondition.predicate(note.getVersion(),
            CompareFilter.CompareOp.EQUAL,
            Constants.VERSION_FIELD);
    note.setMtime(new Date());
    note.setVersion(currentVersion + 1);
    return typedAccessClient.put(note, versionPredicate);
  } finally {
    note.setVersion(currentVersion);
  }
}
```

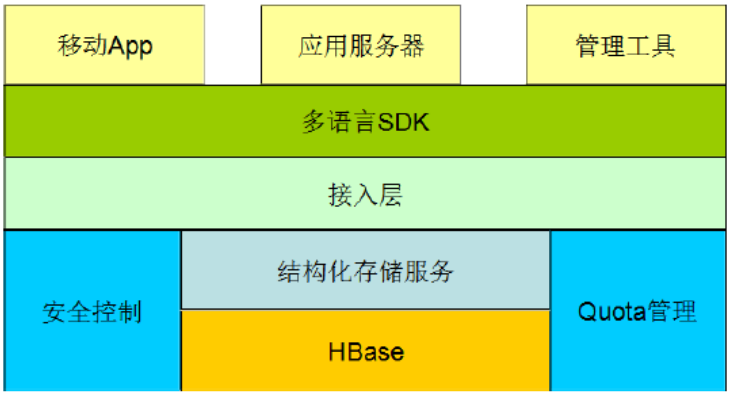
通过这个例子看到，业务层做开发的时候，需要写的代码是非常少的，所以无论是开发效率，还是开发质量上来讲，都可以得到很好的保证。

## 目标




最后一个我简单介绍一下小米开放平台，我们前面讲了，就是我们前面讲了两种使用方式，一种是使用 HBase 原生 Client，另一种是使用 libsds。存在一些问题，第一 Zookeeper，Kerberos 等环境配置比较复杂，第二是只支持 Java 语言(HBase 的 Thrift 跨语言方案不是特别完备)。第三点，小米的应用大多数都是移动的应用，目前的一个趋势是手机端 App 直接访问后端通用数据服务，免去后端业务层服务器的开发。基于这些考虑，我们就在 HBase 的基础上，提供的结构化存储存储服务。

# 架构




这是整体的一个架构，架构也是非常简单的，在 HBase 以上，提供了安全控制和 Quota 管理模块，在这之上是接入层以及多语言 SDK。目前已经接入了一些业务，包括 MIUI 的部分应用，小米智能家居设备的数据，小米手环等生态链公司的数据，目前可以在小米开放平台试用。