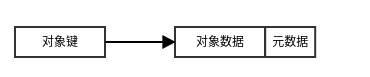
# 面向工业大数据的对象存储技术实践

## 什么是对象存储？

对象存储一般是指以“对象”（object）为数据组织形式的存储服务。引用Wikipedia上给出的定义如下：

**Object storage** (also known as **object-based storage**[[1]](https://en.wikipedia.org/wiki/Object_storage#cite_note-1)) is a [computer data storage](https://en.wikipedia.org/wiki/Computer_data_storage) architecture that manages data as objects, as opposed to other storage architectures like [file systems](https://en.wikipedia.org/wiki/File_systems) which manage data as a file hierarchy, and [block storage](https://en.wikipedia.org/wiki/Block_storage) which manages data as blocks within sectors and tracks.

从这段描述中其实可以看出，对象存储其实并不适合主动的定义方式，而是通过与其它典型存储类型作比较，突出其自身特点。对象存储的单位称为对象，与文件系统里的文件接近，但对象存储不一定要支持文件系统里树状的目录结构。真实的对象存储服务包括Facebook存储的图片存储和Spotify歌曲库等。



对象存储定义并不限定数据模型，但在实现中大家一般采用键 + 数据 + 元数据的结构来描述一个对象（如上图所示），其中：

* 对象键：用于唯一确定一个对象的id；
* 数据：对象的数据本身，一般为一个文件，例如一张图片、一个视频文件等等；
* 元数据（metadata）：描述对象的一组结构化数据，称为元数据，一般以key-value的形式保存；

所以，对象存储也可以看做一种key-value形式，其中key对应对象的键，value对应一个文件和元数据。另外，一般对象存储还有桶（bucket）的概念，主要用于方便对象管理，在一个桶里对象键是唯一的，但不同的桶之间对象键一般是可以重复的。

## 对象存储现状分析

本文讨论面向工业大数据的对象存储技术实践，但对象存储本身并不是一个新颖的概念，几乎所有主流公有云服务供应商都有自己的对象存储服务。下面我们简要介绍一些常见对象存储服务的功能和特点。商业对象存储服务中一般包含较多的定价和安全策略，这些不是本文考虑的重点问题。本文更关注工业大数据应用场景对对象存储在功能上的需求，以及满足这些需求的所依赖的技术要素。

### [AWS S3](https://docs.aws.amazon.com/zh_cn/AmazonS3/latest/dev/)

AWS是Amazon的公有云服务，S3是其对象存储服务。每个 Amazon S3 对象都有键、数据和元数据，与概念介绍一致。在S3，键表述为一种类似文件系统目录的层级结构，例如：

Photos/family/2019/a.jpg  
Docs/work/b.doc

这看起来像Unix文件系统里的路径，但有一点区别，即它是以桶的名称开始（例如上面的Photos和Docs是桶的名称），而不是Unix文件系统里的“/"根目录。层级结构方便用户以类似文件系统的方式组织对象数据，用户甚至可以创建”文件夹“。当然，这些文件夹只是交互上的设计，而实际上键只是一个有层级结构的字符串。

元数据包含两部分，通用信息和自定义内容。通用信息由系统自动生成的，比如对象创建日期和文件大小等等；自定义内容由用户创建，以key-value形式存储。用户可以按键获取对象文件和元数据。S3还允许用户用标签（tag）来标记对象，标签也是以key-value形式保存的，通过接口可以获得某个对象对应的标签列表。标签的目的是方便用户将对象分类，但S3并不支持按照标签来筛选对象数据。但利用AWS检索服务，可以对文件名和元数据自动构建索引，在里的[文章](https://aws.amazon.com/cn/blogs/big-data/building-and-maintaining-an-amazon-s3-metadata-index-without-servers/)有详细介绍。

对于一些结构化（或半结构化）数据，例如CSV、JSON或Parquet格式文件，S3支持称为[S3 Select](https://docs.aws.amazon.com/zh_cn/AmazonS3/latest/API/RESTObjectSELECTContent.html)功能。S3 Select使用户可以使用SQL来获取对象内容，例如

SELECT s.\_1, s.\_2 FROM S3Object s WHERE s.\_3 > 100

对应从CSV格式的对象S3Object里提取前两列，同时需要满足第三列大于100的条件。

S3是AWS的核心服务之一，其推出年代较早，对后续的对象存储服务产生了深远的影响。国内的[阿里云OSS](https://www.alibabacloud.com/zh/product/oss?spm=a2c63.l28256.1097650.dznavproductsb1.cd3c5cc518yyu5)和[腾讯云COS](https://cloud.tencent.com/product/cos)与S3的功能比较接近；[Openstack Swift](https://docs.openstack.org/swift/latest/)是一种开源的对象存储服务，其功能上也类似S3。在数据一致性方面，S3、Swift和COS支持最终一致性；OSS支持强一致性。

### [Azure Blob Storage](https://azure.microsoft.com/zh-cn/services/storage/blobs/)

Azure Blob是微软提供公有云的对象存储方案。Azure Blob采用与AWS S3相似的数据模型，而在应用场景上有一些细化。Azure Blob支持三种存储场景：

* Block blobs：用于管理随机访问的对象数据，读写单位一般是整个对象文件；
* Append blobs：对文件追加（append）操作有所优化，适合类似日志存储；
* Page blobs：对文件内容随机访问有所优化，可用于虚拟机的虚拟硬盘文件；

Azure Blob除了对文件访问场景进行了细化，还支持Azure Search建立对象数据索引（Azure Search是微软公有云服务推出的通用索引服务）。利用Azure Search可以查询存储在Azure Blob里的文件内容，根据查询条件访问这些数据。目前支持索引的[文件格式](https://docs.microsoft.com/en-us/azure/search/search-howto-indexing-azure-blob-storage#supported-document-formats)包括PDF、Office文档、文本文件、JSON、CSV等等。如果被索引的数据发生变化，Azure Search支持增量索引。所以相比AWS S3，Azure Blob支持更灵活的访问方式。Azure Blob支持数据强一致性。

### [Google Cloud Storage](https://cloud.google.com/storage/)

Google Cloud Storage是Google推出的对象存储服务，它的数据模型与AWS S3类似，没有Azure Blob对数据存储的细分场景。与Azure Blob类似的是Google Cloud Storage支持使用[BigQuery](https://cloud.google.com/bigquery/external-data-cloud-storage)建立索引和查询，主要支持一些结构化数据类型，如CSV、JSON、Avro等等。Google Cloud Storage支持数据最终一致性。

### 现状小结

对象存储服务是公有云平台提供的一种标准存储方案，一般都按照对象键 + 数据 + 元数据的数据模型，但在实现细节上稍有不同。在数据访问上，Azure Blob和Google Cloud Storage除了支持按键访问，还支持与索引服务配合使用——建立索引后查询对象数据内容或元数据。大多数对象存储方案中，对象数据作为一个整体进行存取操作，而在Azure Blob下还支持对象内数据读写操作（即Append blobs和Page blobs）。在一致性方面，大部分对象存储支持最终一致性，而Azure Blob和阿里OSS支持强一致性。

## 对象存储设计

### 工业场景下对象存储的需求

工业场景下，对象数据的数据源一般是设备，这与面向终端用户的互联网应用不一样。例如在社交网络应用中，我们可能利用对象存储来保存用户上传的照片，照片的键一般与用户id绑定，对象数据读写从键的分布来看一般是随机的，大部分操作都只访问一个特定id对应的数据（例如读取某个用户的照片时间线），跨多个id访问的情况比较少。但在工业场景下，我们比较少单独考虑某个特定设备的数据，更多是获取一段时间内的满足一定条件的所有对象数据供数据分析使用。

考虑一个具体的场景，假设我们记录了10000台风机产生的故障文件，而我们常常希望通过分析回答：

* 不同型号的风机发生故障的频率是否相同？
* 同一型号不同批次生产的风机发生故障的频率？
* 某一型号的风机随着安装时间增长其故障率的变化情况？
* 不同地理区域的风机故障情况有何特征？
* 不同故障类型的分布情况？

通过对象键可以将对象数据组织成类似文件系统的层级结构，但仍然难以方便地筛选出用于分析的对象数据（如回答上述问题）。在对现状的讨论中，Azure Blob和Google Cloud Storage除了键以外还提供索引服务，但是索引服务是独立于对象存储的，索引与数据之间的延迟并没有保证。另外，一些功能在国内并不支持，例如AWS Search。综合多方面考虑，我们决定设计和实现面向工业场景的对象存储服务。

### 数据模型 = 元数据 + 对象数据

首先我们考虑一种更适合检索的对象数据模型，可以描述成下图



在一般对象数据模型中元数据只起到补充说明的作用，但在我们的对象数据模型里

这是我们设计的对象存储与其它对象存储最大的区别。元数据不仅包括了对象键，还包括用户自定义内容，可以根据自定义内容筛选符合条件的对象数据。

具体来说，一个对象的元数据包含多个列（column）：其中一些列称为id列（id-column），它们的组合对应**对象键**；**自定义内容**由用户定义一些与对象数据相关的信息；**通用信息**列记录了对象数据的统计信息，例如对象的大小和创建时间等。以存储风机产生的故障文件为例，它的元数据可能如下所示：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 风场id | 风机id | 记录时间 | 型号 | 经纬度 | 错误码 | 大小 | 创建时间 | 数据指针 |
| 001 | 001 | 2019-4-28 16:57:33 | A | 39.9042N, 116.4074E | 1 | 11M | 2019-4-28 17:30:12 | ... |
| 001 | 002 | 2019-3-17 12:33:12 | B | 39.9042N, 116.4074E | 12 | 4M | 2019-3-17 13:23:12 | ... |
| 002 | 001 | 2019-5-18 13:27:41 | A | 37.9042N, 112.4074E | 4 | 24M | 2019-5-18 15:20:41 | ... |

其中

* 对象键 = 风场id + 风机id
* 自定义内容 = 记录时间 + 型号 + 经纬度 + 错误码
* 通用信息 = 大小 + 创建时间

**数据指针**用于记录对象数据的存储位置，如果对象数据存储在某种文件系统，例如HDFS，那么指针的形式可以是hdfs://nameservice1:port/path/to/data，所以理论上对象数据可以存储在任意系统之中。

上面的表格看起来像一般的关系数据，列的数据类型可以是字符串、整型、浮点数或日期。一个与关系数据不太一样的地方是对象元数据的schema可能经常发生变化。虽然关系模型也允许修改数据的schema，但在数据量较大的情况下变更schema的代价非常大，所以更适合存储元数据的是一些NoSQL存储引擎，例如Elasticsearch。

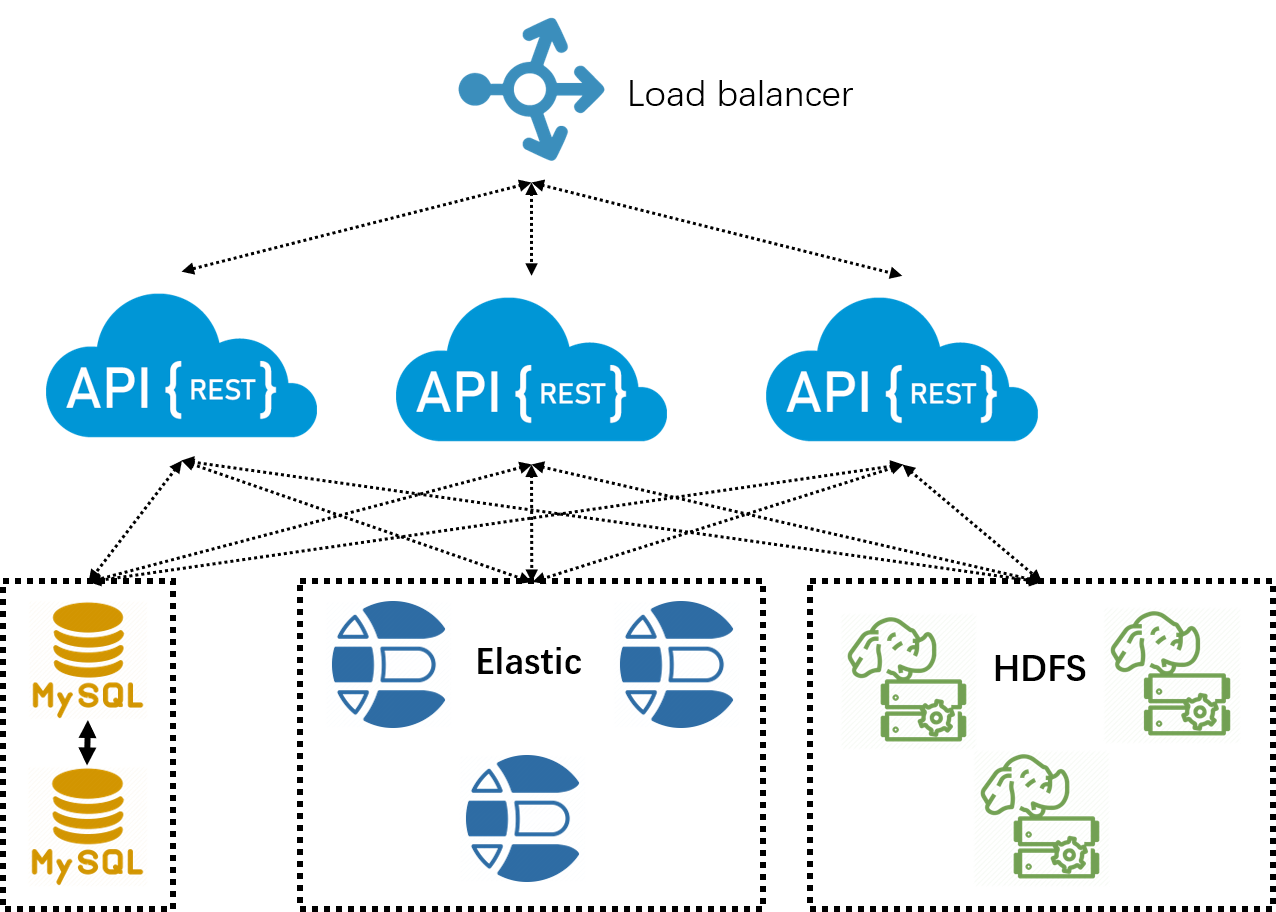
在这样的数据模型下，一般的数据消费方式是先根据某种条件筛选对象的元数据，待得到一个列表后再根据数据指针读取对应的对象数据。例如在上面的例子里，我们可以根据风场id和风机id找到一台风机产生的所有数据；或者找到某种特定故障类型的所有数据。另一种重要的使用方式是只消费对象的元数据，例如我们可以统计过去一个月内发生的不同故障的histogram。这比起只能是根据已知的数据键来读取对象数据的访问方式要灵活得多。

### 系统架构

我们在本文给出一种基于Elasticsearch和HDFS的参考实现，其系统架构如下图所示。用户的读写请求经过Load balancer分发到某台具体的REST服务器；在REST服务里实现对象存储的增删查改操作逻辑；各项操作逻辑基于对底层各种服务的组合调用。底层的三种存储的作用分别是：

* MySQL用于存储对象元数据的schema定义；
* Elasticsearch用于存储和检索对象的元数据；
* HDFS用于存储对象数据；

在这样的架构下对象的元数据和对象数据都以多备份的形式存储并支持HA（High Availability），而REST服务也支持HA，所以整个对象存储服务支持HA，并且均可以根据负载情况水平扩展。



用户的一般使用场景是：

1. 数据建模：考虑对象的元数据定义，这里特指元数据中的对象键和自定义内容的设计，例如在前面的例子里用户设计的对象键（风场id、风机id）和自定义内容（记录时间、型号、经纬度、错误码）。每一种对象数据记做一个**对象类型**（Object Class），对象存储服务同时管理多个对象类型；
2. 数据写入：用户调用REST接口写入对象，每一次写入包括两个部分，即元数据和对象数据。当且仅当元数据和对象数据都完成写入之后一次写入操作才向用户返回成功；
3. 数据变更和删除：用户可以修改对象内容，例如更新元数据中某个字段的值，或者更新对象数据。在发起更新请求之前用户需要提供对象键来唯一确定被修改的对象，这也意味着对象键本身是不能被修改的。对象的删除可以看做是更新的一种特殊场景；
4. 数据检索和读取：用户可以按照对象元数据的列对某个对象类型进行检索，得到一组符合条件的对象元数据列表。由于元数据中包含指向对象数据的指针，所以后续可以读取对应的对象数据；

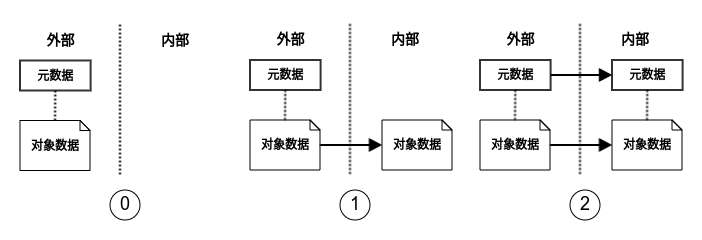
至此我们清楚了对象存储服务的使用过程，下面深入讨论一些技术要点。

### 技术要点1：原子性写操作

在上一节提出的对象存储服务中，一个对象既包括对象数据，还包括元数据。在实现对象的写操作时，需要保证这两部分一致，也就是说对象数据和元数据对用户来说应该是一体的，或者说写操作是原子性的，体现在：

* 写入或删除一个对象的过程中，不应该存在某个时刻使用户只能读到该对象的对象数据或元数据；
* 更新一个对象的过程中，不应该存在某个时刻对象的元数据与对象数据不一致，例如元数据是新的但对象数据是旧的，反之亦然；

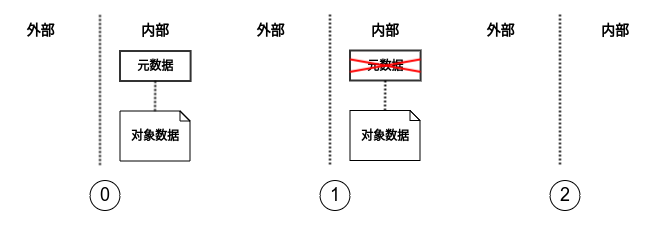
为了达到上述目标，对象的写入过程如下图所示：



可分为两个个步骤，首先从对象存储服务外部复制对象数据到对象存储内部，然后将元数据复制到对象存储服务。表面上看，在上图标记为1的时刻会发生不一致，即对象存储中只有对象数据而没有元数据，但考虑到我们读取数据时总是先查询元数据，所以在此刻用户是看不到这份对象数据的。无论元数据还是对象数据，我们都是复制一份到对象存储内部，而不应该使用转移操作；另外，复制到对象存储内部的对象数据不再保留原文件名，而是改为一个随机生成的文件名来防止冲突。

上面的描述里隐含了一个假设——元数据从外部复制到内部的过程本身是原子性的，即不存在一个时刻使用户能看到尚未写完的元数据。在我们的参考设计里，对象的元数据存储在Elasticsearch，可以保证写入一条记录的原子性，而其实大部分数据库服务都支持至少单条记录操作的原子性。注意对象数据的复制是不需要满足原子性的要求的，所以一般的文件系统都可以用于存储对象数据。

下面我们考虑一下删除的过程，如下图所示：



删除与写入一样分为两个步骤，首先将待删除对象的元数据**标记删除**，即对外部用户不可见，但仍保存在Elasticsearch里，然后再将对象数据和元数据物理删除。标记删除的目的在下一节关于冲突处理时展开。一个实现细节是在上图中物理删除对象时（标记2），应该先删除对象数据，后删除元数据，避免在系统故障时出现对象数据无法被清理的情况。删除操作也是符合原子性要求的，这与写入操作是相同的道理。

更新一个对象可以分两种情况：

1. 只更新对象的元数据中自定义内容；
2. 更新对象数据，这通常意味着元数据也会发生变化，例如数据大小统计信息；

对于前一种情况，原子化的更新操作只依赖于Elasticsearch支持原子化更新，所以不需要额外注意。后一种情况类似对象写入的过程，需要三步：

- 步骤1：把新的对象数据复制到对象存储内部；  
- 步骤2：更新对象元数据内容，并把元数据里的数据指针指向新的对象数据（这个步骤本身是原子性的）；  
- 步骤3：删除原来的对象数据来清理空间；

与写入的原理类似，后一种更新操作也是原子化的，但这里存在一个技术细节——如何确保我们能删除旧的对象数据？步骤2和3之间是不能互换的，否则会存在某个时刻用户可以查询到对象的元数据但却找不到对象数据。因此，如果在步骤2之后系统出现故障，我们如何能知道旧的对象数据在哪儿？注意这时候数据指针已经指向了新的对象数据。为了解决这个问题，在步骤2中，我们需要把旧的数据指针保存下来（例如写入写前日志），即使系统发生故障，在服务恢复后再清理掉旧的对象数据。

本节讨论了原子性写操作实现的技术细节，每种写操作都分为多个步骤，所以在实现时需要写前日志来保证操作的事务性。在系统发生故障后的服务重启时，可以根据写前日志来处理尚未完成的事务。

注：我们的参考实现基于Elasticsearch，在Elasticsearch里可以支持在更新一条记录时执行一个脚本，而整个执行过程也是原子性的。基于这个特性，可以在对象元数据里的数据指针发生改变时将旧的指针记录到元数据本身的一个数组里，从而免去了依赖写前日志来记录的麻烦。

### 技术要点2：读写冲突处理

读写冲突发生在同时写一个对象，或者同时发生一读一写的情况。我们回顾一下写数据的过程，总是先处理对象数据，然后处理元数据；读数据的过程则是先读取元数据，再获取对象数据。所以对象的读写冲突只会发生在元数据相关的操作上。例如，在写入两个对象键相同的对象的过程中，真正可能发生冲突的步骤是两边同时向Elasticsearch写入键相同的记录（Elasticsearch里也有键的概念，在实现时我们将Elasticsearch里的键设定为对象键的字符串）。Elasticsearch内部使用MVCC（Multi-versioned Concurrency Control）的冲突处理模式，在同时写入两条键相同的记录时，[其中一方会失败](https://www.elastic.co/guide/en/elasticsearch/guide/current/version-control.html)。因此，如果在写入两个对象键相同的对象时恰好发生元数据写入冲突，那么其中一方会发生失败。

一读一写的情况会稍微复杂一些，不能完全由Elasticsearch来解决。这里的复杂性主要来自读对象的过程被分为两个阶段，即先获取元数据，再读取对象数据。在现实中这两个阶段之间可能相隔一段较长的时间，例如我们先根据某个查询条件获得一组对象元数据列表，然后我们在一些并行计算框架（例如Mapreduce或Spark）里消费对象数据。假如在得到元数据之后，消费对象数据之前，我们修改了对象数据会发生什么问题？例如，在消费数据之前我们已经把对象数据更新了。合理的做法是用户根据已经获得的元数据仍然可以读取更新前的对象数据，而不会读到新的对象数据（因为会不一致）。另外，也不能因为对象数据已经被更新就返回错误，否则对于那些一次性消费大批对象数据的应用场景，可能出现频繁的失败。在上一节提到了原子性写操作，**无论对于更新还是删除，我们总是先做标记，延迟一段时间之后再物理删除对象数据**。这段时间应该足够长，确保大部分更新前的读取操作已经完成；同时，系统需要维护一个进程按照设定的时间来延迟删除已被标记的对象数据。

### 技术要点3：一致性讨论

从前面关于原子性操作的讨论中，我们可以看到对象存储服务的一致性其实是由其元数据存储的一致性决定的。也就是说，如果我们能做到元数据存储强一致性，那么对象服务就是强一致性的。我们使用Elasticsearch来存储元数据，所以这里需要讨论Elasticsearch的一致性问题。Elasticsearch的一致性一直存在模糊（可以参考[这里](https://www.elastic.co/guide/en/elasticsearch/resiliency/current/index.html)），随着版本不断升级，开发团队在试图减少一些已知问题。

从一般情况来讲Elasticsearch应该属于最终一致（Eventual consistency），但通过一些调整可以实现“接近”强一致性的行为。我们对Elasticsearch的配置调整包括：

* 优先从primary节点读取数据；
* 写入操作设定处于active的节点数量不低于n/2，其中n表示Elasticsearch集群的大小；
* 集群active节点数如果低于n/2则停止对外提供服务；
* 在读取数据前强制执行refresh操作，确保读取发生之前的写入操作已经对外可见；

如果希望深入理解上述配置的含义需要读者对Elasticsearch有比较多的了解，但**简而言之，我们希望每次写入操作都能覆盖到集群里大部分节点，而每次读取则有些选择从Leader节点**（一般是首先被写入的节点）。这样虽然无法保证强一致性，但确保了在大部分情况下，Elasticsearch对外的表现接近强一致性，即我们读到的数据总是最新写入的。

根据我们前面对架构的讨论，Elasticsearch并非唯一可选择的对象元数据存储。我们选择Elasticsearch是看重其强大的检索能力，但如果对一致性有非常严格的要求，也可以选择其它存储方式。

注：我们早期基于MySQL实现过元数据存储，其面临的最大问题是schema修改带来的巨大开销，以及在执行一些聚合操作时耗时过大。

### 技术要点4：文件合并

对象存储服务往往需要储存大量的对象数据，而这些数据会以文件的形式保存在底层的文件系统中。如果存在大量小文件，可能造成文件系统效率降低。例如，在HDFS中，每个block大小通常在64M（或128M），一个block对应一个inode，即HDFS Namenode内存中的一条记录。即使文件再小，在HDFS中仍然会占用一个inode，从而大量的小文件会带给Namenode内存压力。如果我们能把小文件合并成大文件，可以减少对象文件对inode的占用，从而缓解内存压力，这就是文件合并的出发点。如果底层文件系统不是HDFS而是Linux本地文件系统，其inode数量也是有一定上限的，也会有相应的问题。

一种文件合并的思路是将对象按照时间顺序划分为多个区间，每个区间内所有对象的对象数据文件合并为一个大文件。每个对象的元数据都包含其创建时间，这个时间是在创建该对象时由系统自动生成的。后续对该对象的更新操作不会改变对象的创建时间，这个性质非常重要。假设我们设定区间的大小为1小时，那么按时间划分的区间是：

..., (8:00,9:00],(9:00,10:00],...

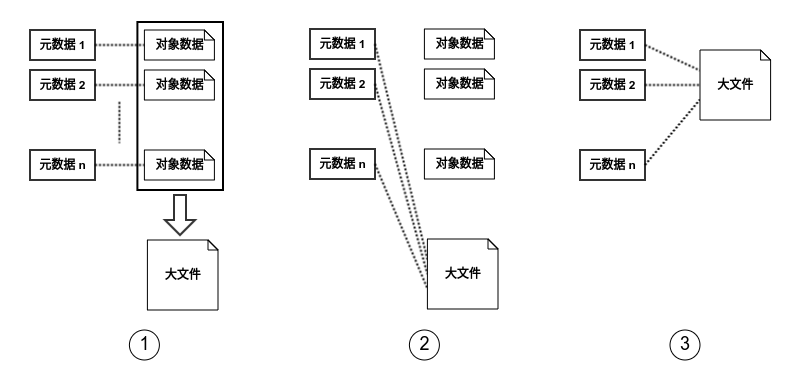
以区间为例，创建时间落入这个区间的所有对象的对象数据会被合并到一个大文件，而其元数据里的指针会指向大文件里的一部分，具体来说包括：

* 大文件的文件名；
* 对象数据在大文件里的offset；
* 对象数据的length；

根据上述三个信息，我们可以从大文件里读取对应的对象数据。文件合并是在对象已经写入对象服务之后发生的，例如上面例子里的区间内的数据合并一定发生在9:00之后，并且**文件合并操作必须对用户是透明的**。换句话说，在文件合并的过程中，用户应该不会感知到底层对象数据正在被合并，而合并操作也不会影响用户的读写操作。为此，合并的步骤包括：

- 步骤1：把待合并的小文件合并为一个大文件；  
- 步骤2：依次更新被合并的所有对象的元数据，使指针指向大文件里的一部分；  
- 步骤3：删除已被合并的小文件来清理空间；

对于任意一个对象来说，其操作过程类似对象的更新操作，虽然步骤2里对象元数据更新无法支持批量更新（即把更新包含在一个原子性事务中），但在任意时刻对外界用户来说，他看到的都是最新的对象数据。如果在合并过程中发生写操作冲突，沿用前面讨论过多冲突处理方式，其中一方发生错误——假设合并涉及100个对象文件，而其中1个由于写冲突失败了，剩余99个成功，那么合并后原来的100个小文件会变成1个大文件（包含100个对象数据）加1个小文件（包含由于冲突导致合并失败的1个对象数据）。可以看到，由于冲突造成了一定的数据冗余，但在正常使用情况下冲突的概率非常小，所以少量的数据冗余是可以容忍的。文件合并的过程如下图所示：



文件合并还会带来另一个数据冗余问题。如果在文件合并发生之后，其中一部分对象数据发生了更新，原本要被删除的对象数据现在已经成为大文件的一部分，而要从大文件里移出其中一部分，相当于把其中未被更新的对象的数据重新写一遍，同时更新对应的数据指针。可见由于文件合并，更新已被合并的对象数据代价较大。实际上，在工业场景下，大部分情况都是对象写入，而发生更新的场景很少（这与互联网应用场景下的对象存储不同），所以在更新比例极少的情况下，我们可以容忍大文件里少量数据已经失效但仍然保留带来的冗余开销。

注：如果现实情况下更新比较频繁，可以采取一定策略来优化合并文件的删除操作。例如，我们可以先统计某个大文件里有多少内容已经失效了，当且仅当失效比例较高的时候才真正执行删除操作。

## 总结

本文介绍了一种面向工业大数据的对象存储服务设计实践。经过场景分析，我们发现工业场景下对象存储的需求与互联网场景下的情况是不一样的，尤其对于对象的检索提出了更高的要求。为了满足这种要求，我们在数据模型设计中强化了元数据的角色，改变了对象数据的消费方式，提出了新的对象存储服务系统架构，结合Elasticsearch + HDFS的参考实现详细讨论了其中的技术要点，希望对从事面向工业大数据的对象存储服务的设计和开发人员提供一定的参考。