**Swift云存储技术详解**

OpenStack Object Storage (Swift) 是用来创建冗余的、可扩展的对象存储（引擎）的开源软件。通过阅读Swift的技术文档，我们可以理解其中的设计的原理和实现的方法 。

       Swift项目已经进展有两年了，对外开放也一年有余，在国外的社区你可以获得许多帮助，但在国内只能找到一些零零散散不齐全的资料，许多人更喜欢坐享其成，而不是参与其中。本人于9月底开始接触swift，刚开始看文档的时候一知半解，有幸阅读了zzcase等人的博客，才得以入门。非常赞同[郑烨](http://dreamhead.blogbus.com/)在某本书序言中所说的话：“翻译向来是一件费力不讨好的事情。”。本人本着知识共享、共同进步的目的，与诸位分享。随着对swift设计原理的理解和源码的深入，文档经过数次反复的修改，希望对各位学习swift的童鞋有所帮助，水平有限，若各位发现有错误之处，恳请指出。文档中的红字部分表示还需斟酌，欢迎提出各种建议和想法。

0.术语约定.................................................................................................................... 3

1. Swift Architectural Overview   Swift架构概述........................... 3

1.1 Proxy Server  代理服务器...................................................................................... 3

1.2 The Ring   环.......................................................................................................... 4

1.3 Object Server  对象服务器.................................................................................... 4

1.4 Container Server 容器服务器................................................................................ 5

1.5 Account Server  帐号服务器................................................................................. 5

1.6 Replication  复制.................................................................................................... 5

1.7 Updaters   更新器.................................................................................................. 5

1.8 Auditors  审计器.................................................................................................... 6

2. The Rings   环..................................................................................................... 6

2.1 Ring Builder   环构造器........................................................................................ 7

2.2 Ring Data Structure   环数据结构........................................................................ 7

2.2.1 List of Devices   设备列表.............................................................................. 8

2.2.2 Partition Assignment List   虚节点分配列表................................................. 8

2.2.3 Partition Shift Value  虚节点位移值............................................................... 9

2.3 Building the Ring  构建环...................................................................................... 9

2.4 History  发展史.................................................................................................... 10

3. The Account Reaper   账号收割器........................................................... 13

3.1 History 发展史..................................................................................................... 14

4. The Auth System   认证系统........................................................................ 15

4.1 TempAuth.............................................................................................................. 15

4.2 Extending Auth  扩展认证.................................................................................. 17

5. Replication   复制......................................................................................... 17

5.1 DB Replication  DB复制器................................................................................. 19

5.2 Object Replication  对象复制.............................................................................. 20

5. Rate Limiting  速率限制................................................................................ 21

5.1 Configuration  配置.............................................................................................. 21

6. Large Object Support    大对象支持........................................................ 23

6.1 Overview  概述..................................................................................................... 23

6.2 Using swift for Segmented Objects  使用swift来分割对象............................. 24

6.3 Direct API  直接的API....................................................................................... 25

6.4 Additional Notes  其他注意事项........................................................................ 27

6.5 History  发展史.................................................................................................... 28

7. Container to Container Synchronization  容器同步............... 30

7.1 Overview  概述..................................................................................................... 30

7.2 Configuring a Cluster’s Allowable Sync Hosts..................................................... 31

配置一个集群容许的同步主机................................................................................. 31

7.3 Using the swift tool to set up synchronized containers......................................... 32

使用swift工具来设置同步容器............................................................................... 32

7.4 Using curl (or other tools) instead.......................................................................... 35

使用curl(或其他工具代替)........................................................................................ 35

7.5 What’s going on behind the scenes, in the cluster?............................................... 36

在集群中，后台正在运行着什么？......................................................................... 36

0.术语约定

* ring 环
* account 帐号
* container 容器
* object 对象
* zone 区域
* devcie 设备
* partition 虚节点
* replica 副本
* replication 复制
* Weight 权重
* Cluster 集群
* consistency window 一致性窗口

1. Swift Architectural Overview   Swift架构概述

**1.1 Proxy Server  代理服务器**

       代理服务器负责Swift架构的其余组件间的相互通信。对于每个客户端的请求，它将在环中查询帐号、容器或者对象的位置并且相应地转发请求。也可以使用公共API向代理服务器发送请求。

       代理服务器也处理大量的失败请求。例如，如果对于某个对象PUT请求时，某个存储节点不可用，它将会查询环可传送的服务器并转发请求。

       对象以流的形式到达（来自） 对象服务器，它们直接从代理服务器传送到（来自）用户—代理服务器并不缓冲它们。

**1.2 The Ring   环**

       环表示存储在硬盘上的实体名称和物理位置间的映射。帐号、容器、对象都有相应的环。当swift的其它组件(比如复制)要对帐号、容器或对象操作时，需要查询相应的环来确定它在集群上的位置。

       环使用区域、设备、虚节点和副本来维护这些映射信息。环中每个虚节点在集群中都(默认)有3个副本。每个虚节点的位置由环来维护,并存储在映射中。当代理服务器转发的客户端请求失败时，环也负责决定由哪一个设备来接手请求。

       环使用了区域的概念来保证数据的隔离。每个虚节点的副本都确保放在了不同的区域中。一个区域可以是一个磁盘，一个服务器，一个机架，一个交换机，甚至是一个数据中心。

       在swift安装的时候，环的虚节点会均衡地划分到所有的设备中。当虚节点需要移动时(例如新设备被加入到集群)，环会确保一次移动最少数量的虚节点数，并且一次只移动一个虚节点的一个副本。

       权重可以用来平衡集群中虚节点在驱动器上的分布。例如，当不同大小的驱动器被用于集群中时就显得非常有用。

       ring被代理服务器和一些后台程序使用（如replication）。

**1.3 Object Server  对象服务器**

       对象服务器是一个简单的二进制大对象存储服务器，可以用来存储、检索和删除本地设备上的对象。在文件系统上，对象以二进制文件的形式存储，它的元数据存储在文件系统的扩展属性(xattrs)中。这要求用于对象服务器的文件系统需要支持文件有扩展属性。一些文件系统，如ext3，它的xattrs属性默认是关闭的。

       每个对象使用对象名称的哈希值和操作的时间戳组成的路径来存储。最后一次写操作总可以成功，并确保最新一次的对象版本将会被处理。删除也被视为文件的一个版本（一个以".ts"结尾的0字节文件，ts表示墓碑）。这确保了被删除的文件被正确地复制并且不会因为遭遇故障场景导致早些的版本神奇再现。

**1.4 Container Server 容器服务器**

       容器服务器的首要工作是处理对象的列表。容器服务器并不知道对象存在哪，只知道指定容器里存的哪些对象。 这些对象信息以sqlite数据库文件的形式存储，和对象一样在集群上做类似的备份。容器服务器也做一些跟踪统计，比如对象的总数，容器的使用情况。

**1.5 Account Server  帐号服务器**

       帐号服务器与容器服务器非常相似，除了它是负责处理容器的列表而不是对象。

**1.6 Replication  复制**

       复制是设计在面临如网络中断或者驱动器故障等临时性故障情况时来保持系统的一致性。

       复制进程将本地数据与每个远程拷贝比较以确保它们都包含有最新的版本。对象复制使用一个哈希列表来快速地比较每个虚节点的子段，容器和帐号的复制使用哈希值和共享的高水位线的组合进行版本比较。

       复制更新基于推模式的。对于对象的复制，更新只是使用rsync同步文件到对等节点。帐号和容器的复制通过HTTP或rsync来推送整个数据库文件上丢失的记录。

       复制器也确保数据已从系统中移除。当有一项（对象、容器、或者帐号）被删除，则一个墓碑文件被设置作为该项的最新版本。复制器将会检测到该墓碑文件并确保将它从整个系统中移除。

**1.7 Updaters   更新器**

       在一些情况下，容器或帐号中的数据不会被立即更新。这种情况经常发生在系统故障或者是高负荷的情况下。如果更新失败，该次更新在本地文件系统上会被加入队列，然后更新器会继续处理这些失败了的更新工作。最终，一致性窗口将会起作用。例如，假设一个容器服务器处于负荷之下，此时一个新的对象被加入到系统。当代理服务器成功地响应客户端的请求，这个对象将变为直接可用的。但是容器服务器并没有更新对象列表，因此此次更新将进入队列等待延后的更新。所以，容器列表不可能马上就包含这个新对象。

       在实际使用中，一致性窗口的大小和更新器的运行频度一致，因为代理服务器会转送列表请求给第一个响应的容器服务器，所以可能不会被注意到。当然，负载下的服务器不应该再去响应后续的列表请求，其他2个副本中的一个应该处理这些列表请求。

**1.8 Auditors  审计器**

       审计器会在本地服务器上反复地爬取来检测对象、容器、帐号的完整性。一旦发现不完整的数据(例如，发生了bit rot的情况：可能改变代码)，该文件就会被隔离，然后复制器会从其他的副本那里把问题文件替换。如果其他错误出现(比如在任何一个容器服务器中都找不到所需的对象列表)，还会记录进日志。

2. The Rings   环

       环决定数据在集群中的位置。帐号数据库、容器数据库和单个对象的环都有独立的环管理，不过每个环均以相同的方式工作。这些环被外部工具管理，服务器进程并不修改环，而是由其他工具修改并传送新的环。

       环从路径的MD5哈希值中使用可配置的比特数，该比特位作为一个虚节点的索引来指派设备。从该哈希值中保留的比特数称为虚节点的幂，并且2的虚节点的幂次方表示虚节点的数量。使用完全MD5哈希值来划分，环允许集群的其他组件一次以分批的项来工作，这将更有效率地完成，或者至少比独立地处理每一个项或者整个集群同时工作的复杂度更低。

       另一个可配置的值是副本数量，表示有多少个虚节点->设备分派来构成单个环。给定一个虚节点编号，每个副本的设备将不会与其它副本的设备在同一个区域内。区域可以基于物理位置、电力分隔、网络分隔或者其它可以减少多个副本在同个时间点上失效的属性用来聚合设备。

**2.1 Ring Builder   环构造器**

       使用工具ring-builder来手动地构建和管理环。ring-builder将虚节点分配到设备并且生成一个优化的Python结构，之后打包(gzipped)、序列化(pickled)，保存到磁盘上，用以服务器的传送。服务器进程只是不定时地检测文件的修改时间，如果需要就重新加载环结构在内存中的拷贝。因为ring-builder管理环的变化的方式，使用一个稍旧的环仅意味对于的一小部分的虚节点，它的3个副本中的一个不正确，这还是容易解决的。

       ring-builder也存有它本身关于环信息的构造器文件和额外所需用来构建新环的数据。保存多份构建器文件的备份拷贝非常重要。一种选择是当复制这些环文件时，复制这些构造器文件到每个服务器上。另一种这是上传构造器文件到集群中。构造器文件的完整性受损将意味着要重新创建一些新的环，几乎所有的虚节点将最终分配到不同的设备，因此几乎所有的数据将不得不复制到新的位置上。所以，从一个受损的构建器文件恢复是有可能的，但是会造成数据在一段时间内不可用。

**2.2 Ring Data Structure   环数据结构**

环的数据结构由三个顶层域组成：在集群中**设备的列表**；**设备id列表的列表**，表示虚节点到设备的指派；以及表示**MD5 hash值位移的位数**来计算该哈希值对应的虚节点。

2.2.1 List of Devices   设备列表

设备的列表在Ring类内部被称为devs。设备列表中的每一项为带有以下键的字典：

|  |  |  |
| --- | --- | --- |
| **id** | **integer** | 所列设备中的索引 |
| **zone** | **integer** | 设备所在的区域 |
| **weight** | **float** | 该设备与其他设备的相对权重。这常常直接与设备的磁盘空间数量和其它设备的磁盘空间数量的比有关。例如，一个1T大小的设备有100的权重而一个2T大小的磁盘将有200的权重。这个权重也可以被用于恢复一个超出或少于所需数据的磁盘。一个良好的平均权重100考虑了灵活性，如果需要日后可以降低该权重。 |
| **ip** | **string** | 包含该设备的服务器IP地址 |
| **port** | **int** | 服务器进程所使用的TCP端口用来提供该设备的服务请求 |
| **device** | **string** | sdb1   服务器上设备的磁盘名称。例如：sdb1 |
| **meta** | **string** | 存储设备额外信息的通用字段。该信息并不直接被服务器进程使用，但是在调试时会派上用场。例如，安装的日期和时间和硬件生产商可以存储在这。 |

注意：设备的列表可能包含了holes，或设为None的索引，表示已经从集群移除的设备。一般地，设备的id不会被重用。一些设备也可以通过设置权重为0.0来暂时地被禁用。为了获得有效设备的列表（例如，用于运行时间轮询），Python代码如下：devices = [device for device in self.devs if device and device['weight']]

2.2.2 Partition Assignment List   虚节点分配列表

       这是设备id的array('I')组成的列表。列表中包含了每个副本的数组array('I')。每个array('I')的长度等于环的虚节点数。在array('I')中的每个整数是到上面设备列表的索引。虚节点列表在Ring类内部被称为\_replica2part2dev\_id。

       因此，创建指派到一个虚节点的设备字典的列表，Python代码如下：devices =[self.devs[part2dev\_id[partition]] for part2dev\_id in self.\_replica2part2dev\_id]

       array('I')适合保存在内存中，因为可能有几百万个虚节点。

2.2.3 Partition Shift Value  虚节点位移值

       虚节点的位移值在Ring类内部称为\_part\_shift。这个值用于转换一个MD5的哈希值来计算虚节点，对于那个哈希值是哪个数据。仅哈希值的前4个字节被用于这个过程。例如，为了计算路径/account/container/object的虚节点，Python代码如下：

partition = unpack\_from('>I',md5('/account/container/object').digest())[0] >> self.\_part\_shift

**2.3 Building the Ring  构建环**

       环的初始化构建首先基于设备的权重来计算理想情况下分配给每个设备的虚节点数量。例如，如虚节点幂为20，则环有1,048,576个虚节点。如果有1000个相同权重的设备，那么它们每个分到1,048.576个虚节点。设备通过它们要求的虚节点数来排序，并在整个初始化过程中保持顺序。

       然后，环构建器根据最适合的原则将每个虚节点的副本分配到设备，限制拥有相同虚节点的副本的设备不能在同一个区域中。每分配一次，设备要求的虚节点数减1并且移动到在设备列表中新的已排序的位置，然后进程继续执行。

       当基于旧环来构造新环时，每个设备所需的虚节点数量被重新计算。接下来，将需要被重新分配的虚节点收集起来。所有被移除的设备将它们已分配的虚节点取消分配并把这些虚节点添加到收集列表。任何一个拥有比目前所需的虚结点数多的设备随机地取消分配虚结点并添加到收集列表中。最后，收集列表中的虚节点使用与上述初始化分配类似的方法被重新分配。

       每当有虚节点的副本被重新分配，重分配的时间将被记录。我们考虑了当收集虚节点来重新分配时，没有虚节点在可配置的时间内被移动两次。这个可配置的时间数量在RingBuilder类内称为min\_part\_hours。这一限制对于已被移除的设备上的虚节点的副本被忽略，因为移除设备仅发生在设备故障并且此时别无择选只能进行重新分配。

    由于收集虚节点用来重新分配的随机本性，以上的进程并不总可以完美地重新平衡一个环。为了帮助达到一个更平衡的环，重平衡进程被重复执行直到接近完美(小于1%）或者当平衡的提升达不到最小值1%（表明由于杂乱不平衡的区域或最近移动的虚节点数过多，我们可能不能获得完美的平衡）。

**2.4 History  发展史**

     环的代码在到达当前版本并保持一段时间的稳定前发生了多次反复的修改，如果有新的想法产生，环的算法可能发生改变甚至从根本上发生变化。这一章节将会描述先前尝试过的想法并且解释为何它们被废弃了。

A “live ring” option was considered where each server could maintain its own copy of the ring and the servers would use a gossip protocol to communicate the changes they made. This was discarded as too complex and error prone to code correctly in the project time span available. One bug could easily gossip bad data out to the entire cluster and be difficult to recover from. Having an externally managed ring simplifies the process, allows full validation of data before it’s shipped out to the servers, and guarantees each server is using a ring from the same timeline. It also means that the servers themselves aren’t spending a lot of resources maintaining rings.

曾考虑过"live ring"选项，其中每个服务器自己可以维护环的副本并且服务器将使用gossip协议进行通讯它们所作做的变化。该方法由于过于复杂并且在工程有效时间内正确编写代码容易产生错误而被废弃。一个Bug是可以很容易把坏数据gossip到整个集群而恢复很困难。通过外部管理环可以简化这一过程，允许数据在传输到服务器前进行数据的完整验证，并且保证每个服务器使用相同时间线的环。这也意味着服务器本身不用花费大量的资源来维护环。

A couple of “ring server” options were considered. One was where all ring lookups would be done by calling a service on a separate server or set of servers, but this was discarded due to the latency involved. Another was much like the current process but where servers could submit change requests to the ring server to have a new ring built and shipped back out to the servers. This was discarded due to project time constraints and because ring changes are currently infrequent enough that manual control was sufficient. However, lack of quick automatic ring changes did mean that other parts of the system had to be coded to handle devices being unavailable for a period of hours until someone could manually update the ring.

有一对"ring server"选项曾被考虑过。一个是所有的环查询可以由调用独立的服务器或服务器集上的服务器来完成，但是由于涉及到延迟被弃用了。另一个更类似于当前的过程，不过其中服务器可以提交改变的请求到环服务器来构建一个新的环，然后运回到服务器上。由于工程时间的约束以及就目前来说，环的改变的频繁足够低到人工控制就可以满足而被弃用。然后，缺乏快速自动的环改变意味着系统的其他部件不得不花上数个小时编码来处理失效的设备直到有人可以手动地升级环。

The current ring process has each replica of a partition independently assigned to a device. A version of the ring that used a third of the memory was tried, where the first replica of a partition was directly assigned and the other two were determined by “walking” the ring until finding additional devices in other zones. This was discarded as control was lost as to how many replicas for a given partition moved at once. Keeping each replica independent allows for moving only one partition replica within a given time window (except due to device failures). Using the additional memory was deemed a good tradeoff for moving data around the cluster much less often.

当前的环程序将一个虚节点的每个副本独立地分配给一个设备。某个环程序版本中尝试使用1/3的内存，其中虚节点的第一个副本被直接分配而另外两个则在环中“行走”直到在其它区域找到额外的设备。这个方法因为对于给定虚节点的多个副本立刻移动会使得控制失效而被废除。（不是很通顺啊）保持每个副本的独立性考虑在给定的时间窗口内仅移动一个虚节点副本（除了由于设备故障）。使用额外的内存看起来是一个不错的权衡，在集群中可以更低频率地移动数据。

Another ring design was tried where the partition to device assignments weren’t stored in a big list in memory but instead each device was assigned a set of hashes, or anchors. The partition would be determined from the data item’s hash and the nearest device anchors would determine where the replicas should be stored. However, to get reasonable distribution of data each device had to have a lot of anchors and walking through those anchors to find replicas started to add up. In the end, the memory savings wasn’t that great and more processing power was used, so the idea was discarded.

另一个被尝试过的环设计是不把虚节点到设备的分配存储在内存中的大列表里而是为每个设备分配一个哈希集合或锚。虚节点将会来自数据项的哈希值来决定并且最近的设备锚将决定副本存储的位置。然而，为了获得更合理的数据分布，每个设备不得不用于大量的锚并且沿着这些锚来寻找副本开始合计。最后，由于内存存储没有那么大并且花费了更多的处理能力，这个想法被废弃了。

A completely non-partitioned ring was also tried but discarded as the partitioning helps many other parts of the system, especially replication. Replication can be attempted and retried in a partition batch with the other replicas rather than each data item independently attempted and retried. Hashes of directory structures can be calculated and compared with other replicas to reduce directory walking and network traffic.

       一个完整的无虚节点的环也被尝试，但是由于虚节点有助于系统的许多其他部件，尤其是复制而被废弃。复制可以在虚节点与其它副本的批处理中被尝试和重试，而不是每个数据项独立地被尝试和重试。目录结构的哈希值可以被计算并用来与其它副本比较来减少目录的遍历和网络流量。

Partitioning and independently assigning partition replicas also allowed for the best balanced cluster. The best of the other strategies tended to give +-10% variance on device balance with devices of equal weight and +-15% with devices of varying weights. The current strategy allows us to get +-3% and +-8% respectively.

虚节点和独立地分配虚节点的副本也考虑了最佳平衡的集群。其他策略的最佳平衡集群在设备平衡上倾向于对于平等权重的设备给出+-10%的变化而对于变化权重的设备则给出+-15%。当前的策略允许我们获得相应+-3%和+-8%的变化。

Various hashing algorithms were tried. SHA offers better security, but the ring doesn’t need to be cryptographically secure and SHA is slower. Murmur was much faster, but MD5 was built-in and hash computation is a small percentage of the overall request handling time. In all, once it was decided the servers wouldn’t be maintaining the rings themselves anyway and only doing hash lookups, MD5 was chosen for its general availability, good distribution, and adequate speed.

       各种哈希的算法被尝试过。SHA提供更好的安全，但是环并不需要安全可靠地加密而且SHA比较慢。Murmur更快，但是MD5是Python内建的库并且哈希计算只是整个请求处理时间中只是一小部分。总之，一旦环被确定，服务器不用自己来维护环而且仅作哈希查找，MD5被选择是因为它的通用性，良好的分布以及足够快的速度。

3. The Account Reaper   账号收割器

The Account Reaper removes data from deleted accounts in the background.

       账号收割器运行在后台从要被删除账号中移除数据。

An account is marked for deletion by a reseller through the services server’s remove\_storage\_account XMLRPC call. This simply puts the value DELETED into the status column of the account\_stat table in the account database (and replicas), indicating the data for the account should be deleted later. There is no set retention time and no undelete; it is assumed the reseller will implement such features and only call remove\_storage\_account once it is truly desired the account’s data be removed.

       通过服务器的remove\_storage\_account的XMLRPC调用，账号被reseller标记为删除。这一行为简单地将值DELETED放入到账号数据库(和副本)的表account\_stat的status列；表示账号数据未来将被删除。没有保留时间和取消删除的设置；它假设reseller将会实现这样的特性并且一旦调用remove\_storage\_account，该账号的数据就真地被移除。

The account reaper runs on each account server and scans the server occasionally for account databases marked for deletion. It will only trigger on accounts that server is the primary node for, so that multiple account servers aren’t all trying to do the same work at the same time. Using multiple servers to delete one account might improve deletion speed, but requires coordination so they aren’t duplicating effort. Speed really isn’t as much of a concern with data deletion and large accounts aren’t deleted that often.

       账号收割器运行在每个账号服务器上，不定期地扫描服务器的账号数据库中标记为删除的数据。它仅会在当前服务器为主节点的账号上触发，因此多个账号服务器并不都尝试着在相同时间内做相同的工作。使用多个服务器来删除一个账号可能会提升删除的速度，但是需要协作以避免重复删除。实际上，在数据删除的速度上并没有给予过多的关注，因为大多数账号并没有那么频繁地被删除。

The deletion process for an account itself is pretty straightforward. For each container in the account, each object is deleted and then the container is deleted. Any deletion requests that fail won’t stop the overall process, but will cause the overall process to fail eventually (for example, if an object delete times out, the container won’t be able to be deleted later and therefore the account won’t be deleted either). The overall process continues even on a failure so that it doesn’t get hung up reclaiming cluster space because of one troublesome spot. The account reaper will keep trying to delete an account until it eventually becomes empty, at which point the database reclaim process within the db\_replicator will eventually remove the database files.

       删除账号的过程是相当直接的。对于每个账号中的容器，每个对象先被删除然后容器被删除。任何失败的删除请求将不会阻止整个过程，但是将会导致整个过程最终失败（例如，如果一个对象的删除超时，容器将不能被删除，因此账号也不能被删除）。整个处理过程即使遭遇失败也继续执行，这样它不会因为一个麻烦的问题而中止恢复集群空间。账号收割器将会继续不断地尝试删除账号直到它最终变为空，此时数据库在db\_replicator中回收处理，最终移除这个数据库文件。

**3.1 History 发展史**

At first, a simple approach of deleting an account through completely external calls was considered as it required no changes to the system. All data would simply be deleted in the same way the actual user would, through the public ReST API. However, the downside was that it would use proxy resources and log everything when it didn’t really need to. Also, it would likely need a dedicated server or two, just for issuing the delete requests.

最初的时候，一个通过完全地外部调用来删除帐号的简单方法被考虑因为它不需要对系统改变。实际的用户可以通过公共的ReST的API以相同的方式来简易地删除所有的数据。然而，坏处是因为它将使用代理的资源并且记录任何信息即使是不需要的日志。此外，它可能需要一个或两个专用的服务器，仅分配给处理删除请求。

A completely bottom-up approach was also considered, where the object and container servers would occasionally scan the data they held and check if the account was deleted, removing the data if so. The upside was the speed of reclamation with no impact on the proxies or logging, but the downside was that nearly 100% of the scanning would result in no action creating a lot of I/O load for no reason.

       一个完全地自底向下的方法也被考虑过，其中对象和容器服务器将不定期地扫面它们的数据并且检测是否该对象被删除了，如果是的话就删除它的数据。好处是回收的速度对于代理或日志没有影响，不过坏事是几乎100%的扫描将会导致无端地没有活动地造成大量的I/O负载。

A more container server centric approach was also considered, where the account server would mark all the containers for deletion and the container servers would delete the objects in each container and then themselves. This has the benefit of still speedy reclamation for accounts with a lot of containers, but has the downside of a pretty big load spike. The process could be slowed down to alleviate the load spike possibility, but then the benefit of speedy reclamation is lost and what’s left is just a more complex process. Also, scanning all the containers for those marked for deletion when the majority wouldn’t be seemed wasteful. The db\_replicator could do this work while performing its replication scan, but it would have to spawn and track deletion processes which seemed needlessly complex.

一个容器服务器中心的方法也曾被考虑，其中账号服务器将会标记所有的要被删除的容器，然后容器服务器将会删除每个容器中的对象接着删除容器。这对于带有大量容器的账号的快速回收大有裨益，但坏处是有相当大的负载峰值。该过程可以被放缓来减轻负载峰值的可能性，不过那样的话快速回收的优点就丧失了并且剩下的只是更复杂的过程。同样的，扫描所有的容器中标记来删除的当大多数的将不会视为浪费的。db\_replicator可以在执行复制扫面时完成这些工作，但是它将不得产生和记录删除过程这些看起来不必要的复杂性。

In the end, an account server centric approach seemed best, as described above.

最后如上所述，账号服务器中心方法看起来是最佳的。

4. The Auth System   认证系统

**4.1 TempAuth**

The auth system for Swift is loosely based on the auth system from the existing Rackspace architecture – actually from a few existing auth systems – and is therefore a bit disjointed. The distilled points about it are:

       Swift的认证系统松散地基于已存在的Rackspace架构的认证系统—实际上来自于一些已存在的认证系统—所以有些不连贯。关于此认证系统的要点有以下4点：

       1.认证/授权部分可以作为一个运行在Swift中作为WSGI中间件的外部系统或子系统

       2.Swift用户在每个请求中会附加认证令牌。

       3.Swift用外部的认证系统或者认证子系统来验证每个令牌并且缓存结果

       4.令牌不是每次请求都会变化，但是存在有效期

The token can be passed into Swift using the X-Auth-Token or the X-Storage-Token header. Both have the same format: just a simple string representing the token. Some auth systems use UUID tokens, some an MD5 hash of something unique, some use “something else” but the salient point is that the token is a string which can be sent as-is back to the auth system for validation.

令牌可以通过使用X-Auth-Token或者X-Storage-Token头部被传入Swift。两者都有相同的格式：仅使用简单的字符串来表示令牌。一些认证系统使用UUID令牌，一些使用唯一的MD5哈希值，一些则使用其它的方法，不过共同点是令牌是可以发送回认证系统进行证实有效性的字符串。

Swift will make calls to the auth system, giving the auth token to be validated. For a valid token, the auth system responds with an overall expiration in seconds from now. Swift will cache the token up to the expiration time.

Swift将会调用认证系统，给出要验证的认证令牌。对于一个正确的令牌，认证系统回应一个从当前开始的总有效期秒数。Swift将会缓存令牌直到有效期结束。

The included TempAuth also has the concept of admin and non-admin users within an account. Admin users can do anything within the account. Non-admin users can only perform operations per container based on the container’s X-Container-Read and X-Container-Write ACLs. For more information on ACLs, see [swift.common.middleware.acl](http://swift.openstack.org/misc.html#module-swift.common.middleware.acl).

其包含的TempAuth，对于account而言，也有admin和non-admin用户的概念。admin用户拥有账号的所有操作权限。non-admin用户仅可以基于每个容器执行基于容器的X-Container-Read and X-Container-Write的访问控制列表进行操作。对于更多关于ACLs的信息，参见[swift.common.middleware.acl](http://swift.openstack.org/misc.html#module-swift.common.middleware.acl)

Additionally, if the auth system sets the request environ’s swift\_owner key to True, the proxy will return additional header information in some requests, such as the X-Container-Sync-Key for a container GET or HEAD.

此外，如果认证系统设置request environ的swift\_owner键为True，该代理服务器将在某些请求中返回额外的头部信息，诸如用于容器的GET或HEAD的X-Container-Sync-Key。

The user starts a session by sending a ReST request to the auth system to receive the auth token and a URL to the Swift system.

用户通过发送一个ReST请求到认证系统来接受认证令牌和一个URL到Swift系统来开始会话。

**4.2 Extending Auth  扩展认证**

TempAuth is written as wsgi middleware, so implementing your own auth is as easy as writing new wsgi middleware, and plugging it in to the proxy server. The KeyStone project and the Swauth project are examples of additional auth services.

Also, see [Auth Server and Middleware](http://swift.openstack.org/development_auth.html).

TempAuth被作为wsgi中间件，因此实现你自己的认证系统就如同写一个新的wsgi中间件一样容易，然后把它安装到代理服务器上。KeyStone和Swauth项目是认证服务器的另外例子。也可以参见 [Auth Server and Middleware](http://swift.openstack.org/development_auth.html).

5. Replication   复制

Since each replica in swift functions independently, and clients generally require only a simple majority of nodes responding to consider an operation successful, transient failures like network partitions can quickly cause replicas to diverge. These differences are eventually reconciled by asynchronous, peer-to-peer replicator processes. The replicator processes traverse their local filesystems, concurrently performing operations in a manner that balances load across physical disks.

       由于每个副本在Swift中独立地运行，并且客户端通常只需要一个简单的主节点响应就可以认为操作成功，如网络等瞬时故障虚节点会快速导致副本出现分歧。这些不同最终由异步、对等网络的replicator进程来调解。replicator进程遍历它们的本地文件，在物理磁盘上以平衡负载的方式并发地执行操作。

Replication uses a push model, with records and files generally only being copied from local to remote replicas. This is important because data on the node may not belong there (as in the case of handoffs and ring changes), and a replicator can’t know what data exists elsewhere in the cluster that it should pull in. It’s the duty of any node that contains data to ensure that data gets to where it belongs. Replica placement is handled by the ring.

       复制使用推模型（推模型的简单实现是通过循环的方式将任务发送到服务器上），记录和文件通常只是从本地拷贝到远程副本。这一点非常重要，因为节点上的数据可能不属于那儿（当在传送数据而环改变的情况下），并且replicator不知道在集群的其它位置上它应该拉什么数据。这是任何一个含有数据的节点职责，确保数据到达它所应该到达的位置。副本的位置由环来处理。

Every deleted record or file in the system is marked by a tombstone, so that deletions can be replicated alongside creations. These tombstones are cleaned up by the replication process after a period of time referred to as the consistency window, which is related to replication duration and how long transient failures can remove a node from the cluster. Tombstone cleanup must be tied to replication to reach replica convergence.

       文件系统中每个被删除的记录或文件被标记为墓碑，因此删除可以在创建的时候被复制。在一段称为一致性窗口的时间后，墓碑文件被replication进程清除，与复制的持续时间和将节点从集群移除瞬时故障的持续时间有关。tombstone的清除应该绑定replication和对应的replica，不应该出现有的replica中的tombstone删除掉了，而有的却没有删除掉。

If a replicator detects that a remote drive is has failed, it will use the ring’s “get\_more\_nodes” interface to choose an alternate node to synchronize with. The replicator can generally maintain desired levels of replication in the face of hardware failures, though some replicas may not be in an immediately usable location.

       如果replicator检测到远程驱动器发生故障，它将使用环的get\_more\_nodes接口来选择一个替代节点进行同步。在面临硬件故障时，复制器通常可以维护所需的复制级别，即使有一些副本可能不在一个直接可用的位置。

Replication is an area of active development, and likely rife with potential improvements to speed and correctness.

       复制是一个活跃的开发领域，在速度和正确性上具有提升的潜力。

There are two major classes of replicator - the db replicator, which replicates accounts and containers, and the object replicator, which replicates object data.

       有两种主要的replicator类型——用来复制账号和容器的db复制器，以及用来复制对象数据的对象复制器。

**5.1 DB Replication  DB复制**

The first step performed by db replication is a low-cost hash comparison to find out whether or not two replicas already match. Under normal operation, this check is able to verify that most databases in the system are already synchronized very quickly. If the hashes differ, the replicator brings the databases in sync by sharing records added since the last sync point.

       db复制执行的第一步是一个低消耗的哈希比较来查明两个副本是否已匹配。在常规运行下，这一检测可以非常快速地验证系统中的大多数数据库已经同步。如果哈希值不一致，复制器通过共享最后一次同步点之后增加的记录对数据库进行同步。

This sync point is a high water mark noting the last record at which two databases were known to be in sync, and is stored in each database as a tuple of the remote database id and record id. Database ids are unique amongst all replicas of the database, and record ids are monotonically increasing integers. After all new records have been pushed to the remote database, the entire sync table of the local database is pushed, so the remote database knows it’s now in sync with everyone the local database has previously synchronized with.

       所谓的同步点是一个高水印标记用来记录上一次记录在哪两个数据库间进行了同步，并且存储在每个数据库中作为一个由remote database id和record id组成的元组。在数据库的所有副本中，数据库的id是唯一的，并且记录id为单调递增的整数。当所有的新纪录推送到远程数据库后，本地数据库的整个同步表被推送出去，因此远程数据库知道现在已经和先前本地数据库与之同步的所有数据库同步了。

If a replica is found to be missing entirely, the whole local database file is transmitted to the peer using rsync(1) and vested with a new unique id.

       如果某个副本完全丢失了，使用rsync(1)传送整个数据库文件到对等节点的远程数据库，并且赋予一个新的唯一id。

In practice, DB replication can process hundreds of databases per concurrency setting per second (up to the number of available CPUs or disks) and is bound by the number of DB transactions that must be performed.

       实际运行中，DB复制可以处理数百个数据库每并发设定值每秒（取决于可用的CPU和磁盘的数量）并且受必须执行DB事务的数量约束。

**5.2 Object Replication  对象复制**

The initial implementation of object replication simply performed an rsync to push data from a local partition to all remote servers it was expected to exist on. While this performed adequately at small scale, replication times skyrocketed once directory structures could no longer be held in RAM. We now use a modification of this scheme in which a hash of the contents for each suffix directory is saved to a per-partition hashes file. The hash for a suffix directory is invalidated when the contents of that suffix directory are modified.

       对象复制的最初实现是简单地执行rsync从本地虚节点推送数据到它预期存放的所有远程服务器上。虽然该方案在小规模上的表现出色，然而一旦目录结构不能保存在RAM中时，复制的时间将会突飞猛涨。我们现在使用这一机制的改进版本，将每个后缀目录的内容的哈希值保存到每一虚节点的哈希文件中。当目录后缀的内容被修改时，它的哈希值将无效。

The object replication process reads in these hash files, calculating any invalidated hashes. It then transmits the hashes to each remote server that should hold the partition, and only suffix directories with differing hashes on the remote server are rsynced. After pushing files to the remote server, the replication process notifies it to recalculate hashes for the rsynced suffix directories.

       对象复制进程读取这些哈希文件，计算出失效的哈希值。然后传输这些哈希值到每个有该partition的远程服务器上，并且仅有不一致哈希的后缀目录在远程服务器上的被rsync。在推送文件到远程服务器之后，复制进程通知服务器重新计算执行了rsync的后缀目录的哈希值。

Performance of object replication is generally bound by the number of uncached directories it has to traverse, usually as a result of invalidated suffix directory hashes. Using write volume and partition counts from our running systems, it was designed so that around 2% of the hash space on a normal node will be invalidated per day, which has experimentally given us acceptable replication speeds.

       对象复制的性能通常被它要遍历的未缓存目录的数量限制，常常作为是失效的**后缀目录的哈希值的结果**。从我们运行的系统上使用写卷和虚节点计数，它被设计因此在一个普通节点上有每天大约2%的哈希空间会失效，已经通过试验，提供给我们可接受的复制速度。

6. Rate Limiting  速率限制

Rate limiting in swift is implemented as a pluggable middleware. Rate limiting is performed on requests that result in database writes to the account and container sqlite dbs. It uses memcached and is dependent on the proxy servers having highly synchronized time. The rate limits are limited by the accuracy of the proxy server clocks.

       速率限制在swift中是作为一个可插的中间件。速率限制处理在数据库写操作到账号和容器sqlite db的请求。它使用memcached并且依赖于高度同步时间的代理服务器。速率限制受限于代理服务器时钟的精度。

**6.1 Configuration  配置**

All configuration is optional. If no account or container limits are provided there will be no rate limiting. Configuration available:

       所有的配置是可选的。如果没有给出账号或容器的限制，那么就没有速率限制。可用配置参数如下：

|  |  |  |
| --- | --- | --- |
| Option | Default | Description |
| clock\_accuracy | 1000 | Represents how accurate the proxy servers’ system clocks are with each other. 1000 means that all the proxies’ clock are accurate to each other within 1 millisecond. No ratelimit should be higher than the clock accuracy.  表示代理服务器的系统时钟相互之间的精度。1000表示所有的代理相互之间的时钟精确到毫秒。没有速率限制应该比该时钟精度更高。 |
| max\_sleep\_time\_seconds | 60 | App will immediately return a 498 response if the necessary sleep time ever exceeds the given max\_sleep\_time\_seconds.  如果必须的休眠时间超过了给定的max\_sleep\_time\_seconds，应用程序会立刻返回一个498响应 |
| log\_sleep\_time\_seconds | 0 | To allow visibility into rate limiting set this value > 0 and all sleeps greater than the number will be logged.  在速率限制中考虑可见性，设置这一值大于0并且所有的休眠时间大于这个数值得将被记录。 |
| rate\_buffer\_seconds | 5 | Number of seconds the rate counter can drop and be allowed to catch up (at a faster than listed rate). A larger number will result in larger spikes in rate but better average accuracy.  速度计数器终止并允许追赶的秒数（以一个比已登录更快的速度）。一个更大的数将会在速率上产生更大的峰值但是更好的平均精度。 |
| account\_ratelimit | 0 | If set, will limit PUT and DELETE requests to /account\_name/container\_name. Number is in requests per second.  如果设置，将会限制PUT和DELETE到account\_name/container\_name请求。数值为每秒的请求数。 |
| account\_whitelist | ‘’ | Comma separated lists of account names that will not be rate limited.  由逗号分隔的不会被速度限制的账号名字列表。 |
| account\_blacklist | ‘’ | Comma separated lists of account names that will not be allowed. Returns a 497 response.  由逗号分隔的不被允许的账号名称列表。 |
| container\_ratelimit\_size | ‘’ | When set with container\_limit\_x = r: for containers of size x, limit requests per second to r. Will limit PUT, DELETE, and POST requests to /a/c/o.  当设置为container\_limit\_x = r :对于大小为x的容器，限制的请求数为r次每秒。使用/a/c/o来限制PUT,DELETE和POST请求。 |

The container rate limits are linearly interpolated from the values given. A sample container rate limiting could be:

容器的速率限制从给定值线性地插入。一个容器速率限制的样例如下：

container\_ratelimit\_100 = 100

container\_ratelimit\_200 = 50

container\_ratelimit\_500 = 20

This would result in

这将会产生

|  |  |
| --- | --- |
| Container Size | Rate Limit |
| 0-99 | No limiting |
| 100 | 100 |
| 150 | 75 |
| 500 | 20 |
| 1000 | 20 |

7. Large Object Support    大对象支持

**7.1 Overview  概述**

Swift has a limit on the size of a single uploaded object; by default this is 5GB. However, the download size of a single object is virtually unlimited with the concept of segmentation. Segments of the larger object are uploaded and a special manifest file is created that, when downloaded, sends all the segments concatenated as a single object. This also offers much greater upload speed with the possibility of parallel uploads of the segments.

Siwft对于单个上传对象有体积的限制；默认是5GB。不过由于使用了分割的概念，单个对象的下载大小几乎是没有限制的。对于更大的对象进行分割然后上传并且会创建一个特殊的描述文件，当下载该对象的时候，把所有的分割联接为一个单个对象来发送。这使得并行上传分割成为可能，因此也提供了更快的上传速度。

**6.2 Using swift for Segmented Objects  使用swift来分割对象**

The quickest way to try out this feature is use the included swift Swift Tool. You can use the -S option to specify the segment size to use when splitting a large file. For example:

尝试这一特性的最快捷的方式是使用swift自带的Swift Tool。你可以使用-S选项来描述在分割大文件的时候使用的分卷大小。例如：

swift upload test\_container -S 1073741824 large\_file

This would split the large\_file into 1G segments and begin uploading those segments in parallel. Once all the segments have been uploaded, swift will then create the manifest file so the segments can be downloaded as one.

这个会把large\_file分割为1G的分卷并且开始并行地上传这些分卷。一旦所有的分卷上传完毕，swift将会创建描述文件，这样这些分卷可以作为一个对象来下载。

So now, the following swift command would download the entire large object:

所以现在，使用以下swift命令可以下载整个大对象：

swift download test\_container large\_file

swift uses a strict convention for its segmented object support. In the above example it will upload all the segments into a second container named test\_container\_segments. These segments will have names like large\_file/1290206778.25/21474836480/00000000, large\_file/1290206778.25/21474836480/00000001, etc.

swift使用一个严格的约定对于它的分卷对象支持。在上面的例子中，它将会上传所有的分卷到一个名为test\_container\_segments的附加容器。这些分卷的名称类似于large\_file/1290206778.25/21474836480/00000000, large\_file/1290206778.25/21474836480/00000001等。

The main benefit for using a separate container is that the main container listings will not be polluted with all the segment names. The reason for using the segment name format of <name>/<timestamp>/<size>/<segment> is so that an upload of a new file with the same name won’t overwrite the contents of the first until the last moment when the manifest file is updated.

使用一个独立的容器的主要好处是主容器列表将不会被所有的分卷名字污染。使用<name>/<timestamp>/<size>/<segment>分卷名称格式的理由是当上传一个相同名称的新文件时将不会重写先前文件的内容直到最后描述文件被上传的时候。

swift will manage these segment files for you, deleting old segments on deletes and overwrites, etc. You can override this behavior with the --leave-segments option if desired; this is useful if you want to have multiple versions of the same large object available.

swift将会为你管理这些分卷文件，使用删除和重写等方法来删除旧的分卷。若需要，你可以用--leave-segments选项重写这一行为；如果你想要同个大对象的多个版本可用这将非常有用。

**6.3 Direct API  直接的API**

You can also work with the segments and manifests directly with HTTP requests instead of having swift do that for you. You can just upload the segments like you would any other object and the manifest is just a zero-byte file with an extra X-Object-Manifest header.

你也可以直接用HTTP请求代替swift工具来使用分卷和描述文件。你可以只上传分卷，在带有一个额外的X-Object-Manifest头部中指明任何其他的对象和描述文件只是一个0字节的文件。

All the object segments need to be in the same container, have a common object name prefix, and their names sort in the order they should be concatenated. They don’t have to be in the same container as the manifest file will be, which is useful to keep container listings clean as explained above with swift.

所有的对象分卷需要在同一个容器内，有一个相同的对象名称前缀，并且它们的名称按照连结的顺序排序。它们不用和描述文件在同一个容器下，这与上面解释swift组件中一样有助于保持容器列表的干净。

The manifest file is simply a zero-byte file with the extra X-Object-Manifest: <container>/<prefix> header, where <container> is the container the object segments are in and <prefix> is the common prefix for all the segments.

描述文件仅是一个带有额外X-Objetc-Manifest的0字节文件： <container>/<prefix> 头部，其中<container>是指对象分卷所在的容器，<prefix>是所有分卷的通用前缀。

It is best to upload all the segments first and then create or update the manifest. In this way, the full object won’t be available for downloading until the upload is complete. Also, you can upload a new set of segments to a second location and then update the manifest to point to this new location. During the upload of the new segments, the original manifest will still be available to download the first set of segments.

最好先上传所有的分卷并且然后创建或升级描述文件。在这种方式下，完整的对象的下载直到上传完成才可用。此外，你可以上传一个新的分卷集到新的位置，然后上传描述文件来指出这一新位置。在上传这些新分卷的时候，原始的描述文件将仍然可用来下载第一个分卷集合。

Here’s an example using curl with tiny 1-byte segments:

这里有一个使用curl对1字节的小分卷的例子：

# First, upload the segments

curl -X PUT -H 'X-Auth-Token: <token>' \

    http://<storage\_url>/container/myobject/1 --data-binary '1'

curl -X PUT -H 'X-Auth-Token: <token>' \

    http://<storage\_url>/container/myobject/2 --data-binary '2'

curl -X PUT -H 'X-Auth-Token: <token>' \

    http://<storage\_url>/container/myobject/3 --data-binary '3'

# Next, create the manifest file

curl -X PUT -H 'X-Auth-Token: <token>' \

    -H 'X-Object-Manifest: container/myobject/' \

    http://<storage\_url>/container/myobject --data-binary ''

# And now we can download the segments as a single object

curl -H 'X-Auth-Token: <token>' \

    http://<storage\_url>/container/myobject

**6.4 Additional Notes  其他注意事项**

With a GET or HEAD of a manifest file, the X-Object-Manifest: <container>/<prefix> header will be returned with the concatenated object so you can tell where it’s getting its segments from.

带有GET或者HEAD的描述文件，X-Object-Manifest: <container>/<prefix>头部将会返回被连结的对象，这样你可以辨别它从哪里获得它的分卷。

The response’s Content-Length for a GET or HEAD on the manifest file will be the sum of all the segments in the <container>/<prefix>listing, dynamically. So, uploading additional segments after the manifest is created will cause the concatenated object to be that much larger; there’s no need to recreate the manifest file.

在描述文件上的GET或HEAD请求的Content-Length是所有在 <container>/<prefix>列表中的分卷的动态总和。因此，在创建了描述文件之后上传额外的分卷将会导致连结对象变得更大；没有需要去重新创建描述文件。

The response’s Content-Type for a GET or HEAD on the manifest will be the same as the Content-Type set during the PUT request that created the manifest. You can easily change the Content-Type by reissuing the PUT.

GET或HEAD描述文件的请求返回的 Content-Type和在创建描述文件的PUT请求中的Content-Type设置一样。你可以通过重新发出PUT请求来轻松地修改Content-Type

The response’s ETag for a GET or HEAD on the manifest file will be the MD5 sum of the concatenated string of ETags for each of the segments in the <container>/<prefix> listing, dynamically. Usually in Swift the ETag is the MD5 sum of the contents of the object, and that holds true for each segment independently. But, it’s not feasible to generate such an ETag for the manifest itself, so this method was chosen to at least offer change detection.

GET或HEAD描述文件的请求的ETag是<container>/<prefix>所列的连结每个分卷的ETags的字符串的MD5值的动态总和。在Swift中Etag常常是对象内容的MD5值总和，并且适用于每个分卷。但是，为描述文件本身来创建这样一个Etag是不可行的，因此这个方法被选择来至少提供变更检测。

Note 注意

If you are using the container sync feature you will need to ensure both your manifest file and your segment files are synced if they happen to be in different containers.

如果你选择了容器同步的特性，你将需要来确保你的描述文件和你的分卷文件被同步若它们在不同的容器中。

**6.5 History  发展史**

Large object support has gone through various iterations before settling on this implementation.

大对象的支持在设为现在这种实现方式前已经经历了各种反复修改。

The primary factor driving the limitation of object size in swift is maintaining balance among the partitions of the ring. To maintain an even dispersion of disk usage throughout the cluster the obvious storage pattern was to simply split larger objects into smaller segments, which could then be glued together during a read.

在swift中驱使限制对象大小的主要因素是维持ring中的partiton间的平衡。为了在集群中维持磁盘使用的平坦散布，一种显而易见的方式是简单地将较大的对象分割到更小的分卷，在读取时分卷可以被粘连在一起。

Before the introduction of large object support some applications were already splitting their uploads into segments and re-assembling them on the client side after retrieving the individual pieces. This design allowed the client to support backup and archiving of large data sets, but was also frequently employed to improve performance or reduce errors due to network interruption. The major disadvantage of this method is that knowledge of the original partitioning scheme is required to properly reassemble the object, which is not practical for some use cases, such as CDN origination.

在介绍大型对象支持之前，一些应用已经将它们的上载对象分割为分卷并且在检索出这些独立块之后在客户端上重新装配它们。这一设计允许客户端支持备份和将大的数据集存档，但也频繁地使用来提升性能或减少由于网络中断引发的错误。这一方法的主要缺点是需要初始的分割组合的知识来合适地将对象重新装配，对于一些使用场景来说是不切实际的，诸如CDN源。

In order to eliminate any barrier to entry for clients wanting to store objects larger than 5GB, initially we also prototyped fully transparent support for large object uploads. A fully transparent implementation would support a larger max size by automatically splitting objects into segments during upload within the proxy without any changes to the client API. All segments were completely hidden from the client API.

为了解决客户想要存储大于5GB的对象障碍，最初的我们原型化完全透明的对于上传大对象的支持。一个完全透明的实现可以在上传时通过自动地将对象分割为分卷在代理内对于客户端API没有任何变化来支持更大的最大分卷大小。

This solution introduced a number of challenging failure conditions into the cluster, wouldn’t provide the client with any option to do parallel uploads, and had no basis for a resume feature. The transparent implementation was deemed just too complex for the benefit.

这一解决方案引入了大量的有挑战性的失败条件到集群中，不会提供客户端任何选项来进行并行上传，而且没有把重新开始特性作为基础。这一透明实现被认为对于好处来说是太复杂了。

The current “user manifest” design was chosen in order to provide a transparent download of large objects to the client and still provide the uploading client a clean API to support segmented uploads.

当前的“用户描述”设计被挑选出来为了提供大型对象到客户的透明下载并且仍然对上载客户端提供了干净的API来支持分卷上载。

Alternative “explicit” user manifest options were discussed which would have required a pre-defined format for listing the segments to “finalize” the segmented upload. While this may offer some potential advantages, it was decided that pushing an added burden onto the client which could potentially limit adoption should be avoided in favor of a simpler “API” (essentially just the format of the ‘X-Object-Manifest’ header).

一种替代的“显式”用户描述选项被讨论，需要一个预定义格式来列出分卷来“完成”分卷上周。尽管这可以提供一些潜在的优势，它决定推送一个增加的负载到客户端上，该行为可能潜在地限制了应该采用更简单的“API”的支持（本质上就是‘X-Object-Manifest’ 头的格式）

During development it was noted that this “implicit” user manifest approach which is based on the path prefix can be potentially affected by the eventual consistency window of the container listings, which could theoretically cause a GET on the manifest object to return an invalid whole object for that short term. In reality you’re unlikely to encounter this scenario unless you’re running very high concurrency uploads against a small testing environment which isn’t running the object-updaters or container-replicators.

在开发期间，我们注意到这种基于路径前缀的“隐式”的用户描述方法可以潜在地被容器列表的一致性窗口影响，理论上在短期内这会产生一个对描述对象的GET返回一个无效的整体对象。实际上，你不可能遇到这种场景除非你运行着非常高的并发性上传针对一个小的没有运行着object-updaters或者container-replicator的测试环境。

Like all of swift, Large Object Support is living feature which will continue to improve and may change over time.

像所有的swift版本，大对象支持是一个活跃的特性，将会继续改进并且不断地改变。

7. Container to Container Synchronization  容器同步

**7.1 Overview  概述**

Swift has a feature where all the contents of a container can be mirrored to another container through background synchronization. Swift cluster operators configure their cluster to allow/accept sync requests to/from other clusters, and the user specifies where to sync their container to along with a secret synchronization key.

swift有一个特性：容器的内容可以通过后端的同步镜像到其他的容器。Swift集群操作员配置他们的集群来允许/接受同步请求到/来自其他的集群，用户使用同步密钥来指定要同步的容器。

Note  注意

Container sync will sync object POSTs only if the proxy server is set to use “object\_post\_as\_copy = true” which is the default. So-called fast object posts, “object\_post\_as\_copy = false” do not update the container listings and therefore can’t be detected for synchronization.

只有代理服务器设置使用 “object\_post\_as\_copy = true”默认值时，容器同步将会同步对象的POSTs。所谓快速对象的posts，使用“object\_post\_as\_copy = false”不升级容器列表并且因此不能被同步检测到。

Note  注意

If you are using the large objects feature you will need to ensure both your manifest file and your segment files are synced if they happen to be in different containers.

如果你使用大对象特性你将需要确保你的描述文件和你的分卷文件被同步了，如果它们在不同的容器内。

**7.2 Configuring a Cluster’s Allowable Sync Hosts**

**配置一个集群容许的同步主机**

The Swift cluster operator must allow synchronization with a set of hosts before the user can enable container synchronization. First, the backend container server needs to be given this list of hosts in the container-server.conf file:

Swift集群操作员必须在用户开启容器同步之前允许和一组主机同步。首先，后端的容器服务器需要在container-server.conf文件中给定这些主机列表：

[DEFAULT]

# This is a comma separated list of hosts allowed in the

# X-Container-Sync-To field for containers.

# allowed\_sync\_hosts = 127.0.0.1

allowed\_sync\_hosts = host1,host2,etc.

...

[container-sync]

# You can override the default log routing for this app here (don't

# use set!):

# log\_name = container-sync

# log\_facility = LOG\_LOCAL0

# log\_level = INFO

# Will sync, at most, each container once per interval

# interval = 300

# Maximum amount of time to spend syncing each container

# container\_time = 60

Tracking sync progress, problems, and just general activity can only be achieved with log processing for this first release of container synchronization. In that light, you may wish to set the above log\_ options to direct the container-sync logs to a different file for easier monitoring. Additionally, it should be noted there is no way for an end user to detect sync progress or problems other than HEADing both containers and comparing the overall information.

跟踪同步的进度，问题，以及只是一般的活动可以只用日志处理来实现容器同步的第一次分布。基于此，你可能希望设置以上的log选项将contaniner-sync导向不同的文件方便监控。此外，需要主要的是对于终端用户来说他们除了使用HEAD两个容器并且比较它们的整体信息以外没有方法来检测同步的进度或问题。

The authentication system also needs to be configured to allow synchronization requests. Here is an example with TempAuth:

认真系统也需要进行配置来允许同步请求。这里有一个关于TempAuth的例子：

[filter:tempauth]

# This is a comma separated list of hosts allowed to send

# X-Container-Sync-Key requests.

# allowed\_sync\_hosts = 127.0.0.1

allowed\_sync\_hosts = host1,host2,etc.

The default of 127.0.0.1 is just so no configuration is required for SAIO setups – for testing.

默认值127.0.0.1只是因为SAIO设置不需要配置-用于测试。

**7.3 Using the swift tool to set up synchronized containers**

**使用swift工具来设置同步容器**

**Note 注意**

You must be the account admin on the account to set synchronization targets and keys.

你必须使用该帐号的帐号管理权限来设置同步目标和键。

You simply tell each container where to sync to and give it a secret synchronization key. First, let’s get the account details for our two cluster accounts:

你只需告诉每个容器同步到何处并且给予一个同步的密钥。首先，让我们先获得我们的两个集群帐号的帐号细节：

$ swift -A http://cluster1/auth/v1.0 -U test:tester -K testing stat -v

StorageURL: http://cluster1/v1/AUTH\_208d1854-e475-4500-b315-81de645d060e

Auth Token: AUTH\_tkd5359e46ff9e419fa193dbd367f3cd19

   Account: AUTH\_208d1854-e475-4500-b315-81de645d060e

Containers: 0

   Objects: 0

     Bytes: 0

$ swift -A http://cluster2/auth/v1.0 -U test2:tester2 -K testing2 stat -v

StorageURL: http://cluster2/v1/AUTH\_33cdcad8-09fb-4940-90da-0f00cbf21c7c

Auth Token: AUTH\_tk816a1aaf403c49adb92ecfca2f88e430

   Account: AUTH\_33cdcad8-09fb-4940-90da-0f00cbf21c7c

Containers: 0

   Objects: 0

     Bytes: 0

Now, let’s make our first container and tell it to synchronize to a second we’ll make next:

现在，让我们获取我们第一个容器并且告诉它我们下一步将会设置的第二个容器：

$ swift -A http://cluster1/auth/v1.0 -U test:tester -K testing post \

  -t 'http://cluster2/v1/AUTH\_33cdcad8-09fb-4940-90da-0f00cbf21c7c/container2' \

  -k 'secret' container1

The -t indicates the URL to sync to, which is the StorageURL from cluster2 we retrieved above plus the container name. The -k specifies the secret key the two containers will share for synchronization. Now, we’ll do something similar for the second cluster’s container:

-t表示同步到的URL，就是我们上一步从cluster2检索得到的StorageURL再加上容器名称。-k指定了两个容器共享用于同步的安全密钥。现在，我们对于第二个集群容器将会做一些类似的操作：

$ swift -A http://cluster2/auth/v1.0 -U test2:tester2 -K testing2 post \

  -t 'http://cluster1/v1/AUTH\_208d1854-e475-4500-b315-81de645d060e/container1' \

  -k 'secret' container2

That’s it. Now we can upload a bunch of stuff to the first container and watch as it gets synchronized over to the second:

就是如此。现在我们可以上载一堆东西到第一个容器并且观察它与第二个容器进行同步：

$ swift -A http://cluster1/auth/v1.0 -U test:tester -K testing \

  upload container1 .

photo002.png

photo004.png

photo001.png

photo003.png

$ swift -A http://cluster2/auth/v1.0 -U test2:tester2 -K testing2 \

  list container2

[Nothing there yet, so we wait a bit...]

[If you're an operator running SAIO and just testing, you may need to

 run 'swift-init container-sync once' to perform a sync scan.]

$ swift -A http://cluster2/auth/v1.0 -U test2:tester2 -K testing2 \

  list container2

photo001.png

photo002.png

photo003.png

photo004.png

You can also set up a chain of synced containers if you want more than two. You’d point 1 -> 2, then 2 -> 3, and finally 3 -> 1 for three containers. They’d all need to share the same secret synchronization key.

你也可以设置一个同步容器链如果你想要使用两个以上的容器。对于三个容器，你得将指向1->2，然后2->3，最后3->1。它们须共享同一个同步密钥。

**7.4 Using curl (or other tools) instead**

**使用curl(或其他工具代替)**

So what’s swift doing behind the scenes? Nothing overly complicated. It translates the -t <value> option into an X-Container-Sync-To:<value> header and the -k <value> option into an X-Container-Sync-Key: <value> header.

因此swift在这个场景背后做了什么呢？其实没有什么很复杂的操作。它将-t<value>选项转换为一个 X-Container-Sync-To: <value> 头以及将-k <value>选项转换为 X-Container-Sync-Key: <value> 头。

For instance, when we created the first container above and told it to synchronize to the second, we could have used this curl command:

例如，当我们创建以上第一个容器时并且告诉它与第二个容器同步，我们可以使用以下curl命令：

$ curl -i -X POST -H 'X-Auth-Token: AUTH\_tkd5359e46ff9e419fa193dbd367f3cd19' \

  -H 'X-Container-Sync-To: http://cluster2/v1/AUTH\_33cdcad8-09fb-4940-90da-0f00cbf21c7c/container2' \

  -H 'X-Container-Sync-Key: secret' \

  'http://cluster1/v1/AUTH\_208d1854-e475-4500-b315-81de645d060e/container1'

HTTP/1.1 204 No Content

Content-Length: 0

Content-Type: text/plain; charset=UTF-8

Date: Thu, 24 Feb 2011 22:39:14 GMT

**7.5 What’s going on behind the scenes, in the cluster?**

**在集群中，后台正在运行着什么？**

The swift-container-sync does the job of sending updates to the remote container.

swift-container-sync执行发送update到远程容器的工作。

This is done by scanning the local devices for container databases and checking for x-container-sync-to and x-container-sync-key metadata values. If they exist, newer rows since the last sync will trigger PUTs or DELETEs to the other container.

通过扫描本地设备的容器数据库并且检测x-container-sync-to和x-container-sync-key元数据值来完成。如果它们存在，上一次更新的较新的行将会触发PUTS和DELETEs到其它的容器。

Note 注意

Container sync will sync object POSTs only if the proxy server is set to use “object\_post\_as\_copy = true” which is the default. So-called fast object posts, “object\_post\_as\_copy = false” do not update the container listings and therefore can’t be detected for synchronization.

容器同步将会同步对象POSTs仅当代理服务器设置使用了“object\_post\_as\_copy = true”的默认值。所谓的快速对象发送，“object\_post\_as\_copy = false”不升级容器列表，所以因此不能被同步检测。

The actual syncing is slightly more complicated to make use of the three (or number-of-replicas) main nodes for a container without each trying to do the exact same work but also without missing work if one node happens to be down.

使用一个容器的三个（或者replicas的数目number-of-replicas）主节点实际的同步稍微更复杂些，没有每个尝试去做完全相同的工作而且如果一个节点发生故障不会丢失工作。

Two sync points are kept per container database. All rows between the two sync points trigger updates. Any rows newer than both sync points cause updates depending on the node’s position for the container (primary nodes do one third, etc. depending on the replica count of course). After a sync run, the first sync point is set to the newest ROWID known and the second sync point is set to newest ROWID for which all updates have been sent.

两个同步点被保存在每个容器数据库中。在两个同步点之间的所有的行触发update。两个同步节点间触发updates的任何一个较新的行取决于节点对于容器的位置（主节点做1/3，等等。当然取决于replica的数量）。在同步运行之后，第一个同步点设置为最新已知的ROWID并且第二个同步点被设置为最新的ROWID表示所有的updates已经被发送。

An example may help. Assume replica count is 3 and perfectly matching ROWIDs starting at 1.

一个例子有助于理解。假设replica数量是3并且完全匹配在1开始的ROWID。

First sync run, database has 6 rows:

SyncPoint1 starts as -1.

SyncPoint2 starts as -1.

No rows between points, so no “all updates” rows.

Six rows newer than SyncPoint1, so a third of the rows are sent by node 1, another third by node 2, remaining third by node 3.

SyncPoint1 is set as 6 (the newest ROWID known).

SyncPoint2 is left as -1 since no “all updates” rows were synced.

Next sync run, database has 12 rows:

SyncPoint1 starts as 6.

SyncPoint2 starts as -1.

The rows between -1 and 6 all trigger updates (most of which should short-circuit on the remote end as having already been done).

Six more rows newer than SyncPoint1, so a third of the rows are sent by node 1, another third by node 2, remaining third by node 3.

SyncPoint1 is set as 12 (the newest ROWID known).

SyncPoint2 is set as 6 (the newest “all updates” ROWID).

In this way, under normal circumstances each node sends its share of updates each run and just sends a batch of older updates to ensure nothing was missed.

用这种方式，在一般情况下每个节点发送它运行的共享的更新并且仅发送一批较老的更新来确保没有丢失信息。