# 对象存储

# 数据湖存储

# 产品文档

腾讯云

# 文档目录

数据湖存储
- 产品概述
- 更新日志
- 快速入门
- 核心特性
  - 缓存能力
  - 透明加速能力
  - 统一命名空间能力
  - Table 管理能力
  - GooseFS-FUSE 能力
- 部署指南
  - 使用自建集群部署
  - 使用腾讯云 EMR 部署
  - 使用腾讯云 TKE 部署
  - 使用腾讯云 EKS 部署
  - 使用 Docker 部署
- 运维指南
  - 日志指引
    - GooseFS 日志介绍
  - 监控指南
    - 获取 GooseFS 监控指标
    - 基于 Prometheus 搭建 GooseFS 监控体系
  - 集群配置实践
- 数据安全
  - 使用 Apache Ranger 控制 GooseFS 的访问权限
- GooseFS on TKE 云原生实践
  - 入门
    - 安装
    - 快速入门
    - 问题诊断和处理
  - 使用 GooseFS 作为 Fluid Dataset Runtime
    - Master 节点高可用部署模式
    - 多 master 节点亲和性调度
    - 加速 COS 上的数据
    - 客户端全局部署

使用参数加密
手动扩缩容
数据缓存和元数据缓存
数据亲和性调度
数据容忍污点调度
数据预加载
使用 placement 在同个集群上部署多个 dataset

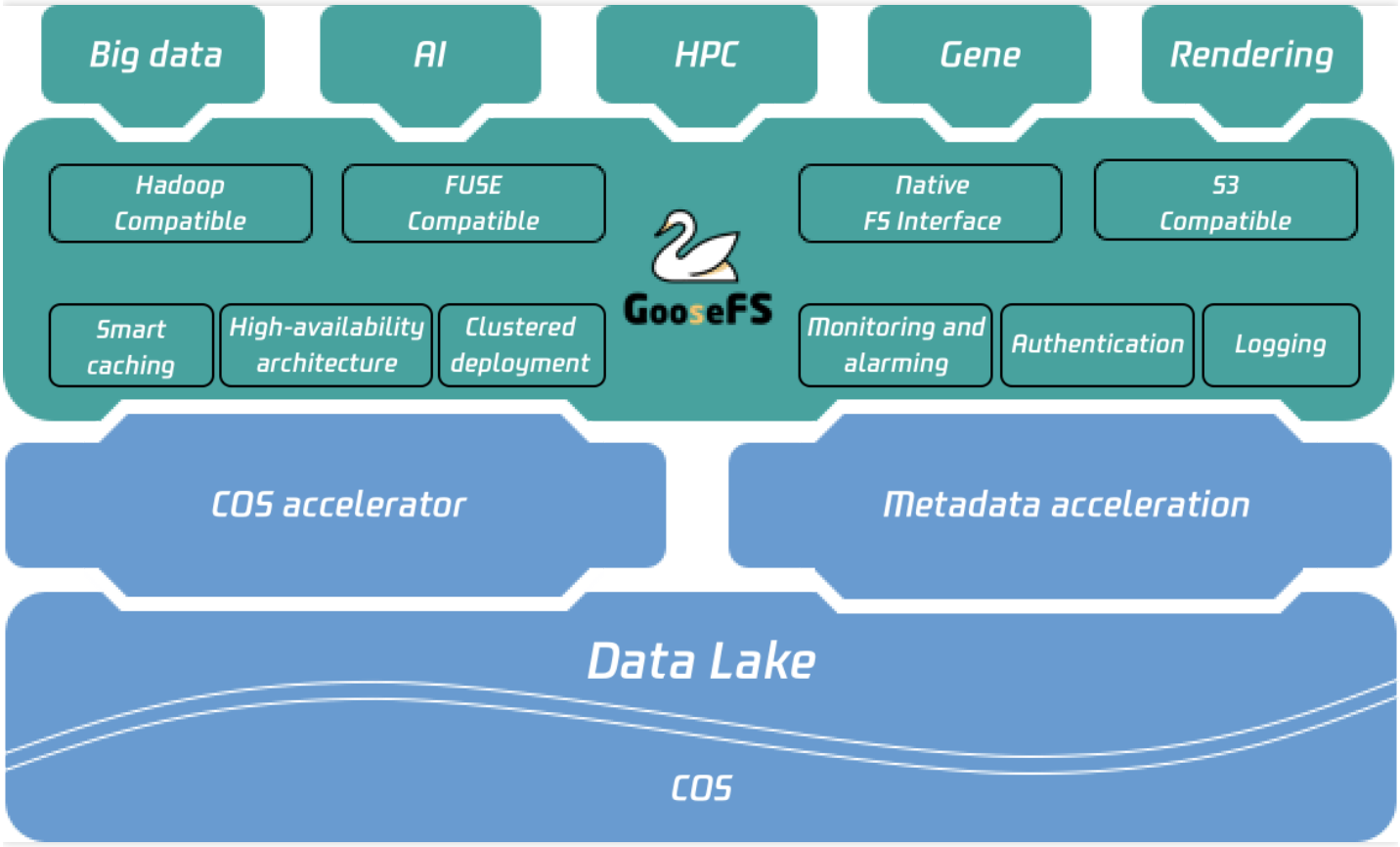
# 数据湖存储
# 产品概述

最近更新时间：2022-06-10 14:16:43

数据湖加速器（Data Lake Accelerator Goose FileSystem，GooseFS），是由腾讯云推出的高可靠、高可用、弹性的数据湖加速服务。依靠对象存储（Cloud Object Storage，COS）作为数据湖存储底座的成本优势，为数据湖生态中的计算应用提供统一的数据湖入口，加速海量数据分析、机器学习、人工智能等业务访问存储的性能；采用了分布式集群架构，具备弹性、高可靠、高可用等特性，为上层计算应用提供统一的命名空间和访问协议，方便用户在不同的存储系统管理和流转数据。

## 产品功能

GooseFS 旨在提供一站式的缓存解决方案，在利用数据本地性和高速缓存，统一存储访问语义等方面具有天然的优势；GooseFS 在腾讯云数据湖生态中扮演着“上承计算，下启存储”的核心角色，如下图所示。




GooseFS 提供了以下功能：

1. 缓存加速和数据本地化（Locality）：GooseFS 可以与计算节点混合部署提高数据本地性，利用高速缓存功能解决存储性能问题，提高写入 COS 的带宽。
2. 融合存储语义：GooseFS 提供 UFS（Unified FileSystem）的语义，可以支持 COS、Hadoop、S3、K8S CSI、FUSE 等多个存储语义，使用于多种生态和应用场景。
3. 统一的腾讯云相关生态服务：包括日志、鉴权、监控，实现了与 COS 操作统一。
4. 提供 Namespace 管理能力，针对不同业务、不同的Under File System，提供不同的读写缓存策略以及生命周期（TTL）管理。
5. 感知 Table 元数据功能：对于大数据场景下数据 Table，提供 GooseFS Catalog 用于感知元数据 Table，提供 Table 级别的 Cache 预热。

# 产品优势

GooseFS 在数据湖场景中具有如下几点明显的优势：

### 数据 I/O 性能

GooseFS 部署提供近计算端的分布式共享缓存，上层计算应用可以透明地、高效地从远端存储将需要频繁访问的热数据缓存到近计算端，加速数据 I/O 性能。GooseFS 提供了元数据缓存功能，可以加速大数据场景下查询文件数据以及列出文件列表等元数据操作的性能。配合大数据存储桶使用，还可进一步加速重命名文件的操作性能。此外，业务可以按需选择 MEM、SSD、NVME 以及 HDD 盘等不同的存储介质，平衡业务成本和数据访问性能。

### 存储一体化

GooseFS 提供了统一的命名空间，不仅支持了对象存储 COS 存储语义，也支持 HDFS、K8S CSI 以及 FUSE 等语义，为上层业务提供了一体化的融合存储方案，简化业务侧运维配置。存储一体化能够打通不同数据底座的壁垒，方便上层应用管理和流转数据，提升数据利用的效率。

### 生态亲和性

GooseFS 全兼容腾讯云大数据平台框架，也支持客户侧自定义的本地部署，具备优秀的生态亲和性。业务侧不仅可以在腾讯云弹性 MapReduce 产品中使用 GooseFS 加速大数据业务，也可以便捷地将 GooseFS 本地化部署在公有云 CVM 或者自建 IDC 内。此外，GooseFS 支持透明加速能力，对于已经使用腾讯云 COSN 和 CHDFS 的用户，只需做简单的配置修改，即可实现不修改任何业务代码和访问路径的前提下，自动使用GooseFS 加速 COSN 和 CHDFS 的业务访问。

# 更新日志

最近更新时间：2022-09-09 10:11:04

GooseFS 版本更新列表如下，如您有任何疑问或建议，欢迎 联系我们。

| 版本号 | 更新日期 | 更新说明 | 下载链接 |
| --- | --- | --- | --- |
| 1.3.0 | 2022年7月25日 | • 新增功能<br>i. 支持 Kerberos 认证。<br>ii. 支持访问开启 元数据加速能力 的存储桶，以原生 POSIX 语义访问对象存储服务。<br>iii. Master 节点缓存淘汰策略支持 LRU 算法，避免缓存频繁置换。<br>iv. Master 节点支持过期元数据清理能力。<br>• 功能优化<br>i. GooseFS Master节点优化了锁瓶颈问题，显著提升 Master QPS，带来 35% 左右的性能优化效果。<br>ii. GooseFS Worker 节点支持并发 Format，提升操作性能。<br>iii. GooseFS Fuse 客户端支持覆盖写操作。<br>iv. GooseFS Fuse 客户端优化了 `ls` 命令的内存占用问题。<br>v. GooseFS HDFS 客户端优化 ListNamespace 的性能。<br>• Bug 修复<br>i. 修复了 RocksDB 泄漏和 Core 的问题，避免内存泄露。<br>ii. 修复了 Zookeeper Curator 误打日志的问题。<br>iii. 修复了 UFS 读带宽不准的问题。<br>iv. 修复了 DistributedLoad 的时候由于日志打印过多导致的 LostWorker问题。 | • GooseFS ：点击下载<br>• GooseFS（KonaJDK 版本）：点击下载<br>• GooseFS 客户端：点击下载<br>• GooseFS 客户端（KonaJDK 版本）：点击下载<br>• GooseFS FUSE 客户端：点击下载<br>• GooseFS FUSE 客户端（KonaJDK 版本）：点击下载 |

| 版本号 | 更新日期 | 更新说明 | 下载链接 |
| --- | --- | --- | --- |
| 1.2.0 | 2022年1月25日 | • 新增功能<br>i. GooseFS 支持将日志上报云日志系统。<br>ii. Inode 和 Block 元信息支持单独配置 RocksDB。<br>iii. GooseFS Client 支持热开关能力，提升故障容灾措施。<br>iv. 添加了部分全链路日志的基础设施。<br>• 功能优化<br>i. 优化 Raft 切主延迟高的问题。<br>ii. 优化了 GooseFS HCFS Client 的内存占用。<br>• Bug 修复<br>i. 修复 Journal 乱序的问题。<br>ii. Ratis 死锁导致的 GRPC 问题。<br>iii. 修复了 HDFSUnderFileSystemFactory 加载位置不正确的问题。<br>iv. 修复了 log4j2 的安全漏洞问题。<br>v. 修复了 ufsPath 前缀检查错误的问题。 | • GooseFS：点击下载<br>• GooseFS（KonaJDK 版本）：点击下载<br>• GooseFS FUSE 客户端：点击下载 |
| 1.1.0 | 2021年9月1日 | • 新增功能：<br>i. 支持 ranger 鉴权，点击下载 ranger 插件。<br>ii. 支持并发 list。<br>iii. 支持 distributedLoad 目录原子性。<br>iv. 支持 namespace 的缓存白名单。<br>v. 支持 ofs scheme 透明加速。<br>• 功能优化：<br>i. cli 为各个 subcmd 增加 help 命令。<br>ii. 优化 table 挂载检测 namespace 是否存在。<br>iii. 完善 job worker/worker metrics 监控。<br>• Bug 修复：<br>修复 distributedLoad 读膨胀问题。 | 当前版本不再维护，请下载最新版本 |
| 1.0.0 | 2021年6月1日 | 1. 基于 namespace 的读写策略与 TTL 管理。<br>2. 基于 Hive Table 维度以及 Table Partition 维度的预热。<br>3. 兼容存量 COSN 用户的路径兼容，实现透明加速。<br>4. Fluid 集成 GooseFS。<br>5. 集成了 CHDFS 支持。 | 当前版本不再维护，请下载最新版本 |

# 快速入门

最近更新时间：2022-08-01 12:03:54

本文档主要提供 GooseFS 快速部署、调试的相关指引，提供在本地机器上部署 GooseFS，并将对象存储（Cloud Object Storage，COS）作为远端存储的步骤指引，具体步骤如下：

## 前提条件

在使用 GooseFS 之前，您还需要准备以下工作：

1. 在 COS 服务上创建一个存储桶以作为远端存储，操作指引请参见 控制台快速入门。
2. 安装 Java 8 或者更高的版本。
3. 安装 SSH，确保能通过 SSH 连接到 LocalHost，并远程登录。

## 下载并配置 GooseFS

1. 从官方仓库下载 GooseFS 安装包到本地。官方仓库下载链接：goosefs-1.3.0-bin.tar.gz。

2. 执行如下命令，对安装包进行解压。

```
tar -zxvf goosefs-1.3.0-bin.tar.gz
cd goosefs-1.3.0
```

   解压后，得到 goosefs-1.2.0，即 GooseFS 的主目录。下文将以 `${GOOSEFS_HOME}` 代指该目录的绝对路径。

3. 在 `${GOOSEFS_HOME}/conf` 的目录下，创建 `conf/goosefs-site.properties` 的配置文件，可以使用内置的配置模板：

```
$ cp conf/goosefs-site.properties.template conf/goosefs-site.properties
```

4. 在配置文件 `conf/goosefs-site.properties` 中，将 goosefs.master.hostname 设置为 `localhost` ：

```
$ echo"goosefs.master.hostname=localhost">> conf/goosefs-site.properties
```

# 启用 GooseFS

1. 启用 GooseFS 前，检查系统环境，确保 GooseFS 可以在本地环境中正确运行：

```
$ goosefs validateEnv local
```

2. 启用 GooseFS 前，执行如下命令，对 GooseFS 进行格式化。该命令将清除 GooseFS 的日志和 `worker` 存储目录下的内容：

```
$ goosefs format
```

3. 执行如下命令，启用 GooseFS。在系统默认配置下，GooseFS 会启动一个 Master 和一个 Worker。

```
$ ./bin/goosefs-start.sh local SudoMount
```

该命令执行完毕后，可以访问 http://localhost:9201 和 http://localhost:9204，分别查看 Master 和 Worker 的运行状态。

# 使用 GooseFS 挂载 COS（COSN） 或腾讯云 HDFS（CHDFS）

如果 GooseFS 需要挂载 COS（COSN）或腾讯云 HDFS（CHDFS）到 GooseFS 的根路径上，则需要先在 `conf/core-site.xml` 配置中指定 COSN 或 CHDFS 的必需配置项，其中包括但不限于：`fs.cosn.impl`、`fs.AbstractFileSystem.cosn.impl` 以及 `fs.cosn.userinfo.secretId` 和 `fs.cosn.userinfo.secretKey` 等，如下所示：

```
<!-- COSN related configurations -->
<property>
<name>fs.cosn.impl</name>
<value>org.apache.hadoop.fs.CosFileSystem</value>
</property>

<property>
<name>fs.AbstractFileSystem.cosn.impl</name>
<value>com.qcloud.cos.goosefs.CosN</value>
</property>

<property>
<name>fs.cosn.userinfo.secretId</name>
```

```
<value></value>
</property>

<property>
<name>fs.cosn.userinfo.secretKey</name>
<value></value>
</property>

<property>
<name>fs.cosn.bucket.region</name>
<value></value>
</property>

<!-- CHDFS related configurations -->
<property>
<name>fs.AbstractFileSystem.ofs.impl</name>
<value>com.qcloud.chdfs.fs.CHDFSDelegateFSAdapter</value>
</property>

<property>
<name>fs.ofs.impl</name>
<value>com.qcloud.chdfs.fs.CHDFSHadoopFileSystemAdapter</value>
</property>

<property>
<name>fs.ofs.tmp.cache.dir</name>
<value>/data/chdfs_tmp_cache</value>
</property>

<!--appId-->
<property>
<name>fs.ofs.user.appid</name>
<value>1250000000</value>
</property>
```

说明：

- COSN 的完整配置可参考：Hadoop 工具。
- CHDFS 的完整配置可参考：挂载 CHDFS。

下面将介绍一下如何通过创建 Namespace 来挂载 COS 或 CHDFS 的方法和步骤。

1. 创建一个命名空间 namespace 并挂载 COS：

```
$ goosefs ns create myNamespace cosn://bucketName-1250000000/3TB \
--secret fs.cosn.userinfo.secretId=AKXXXXXXXXXXX \
--secret fs.cosn.userinfo.secretKey=XXXXXXXXXXXX \
--attribute fs.cosn.bucket.region=ap-xxx \
```

注意：

- 创建挂载 COSN 的 namespace 时，必须使用 `--secret` 参数指定访问密钥，并且使用 `--attribute` 指定 Hadoop-COS（COSN）所有必选参数，具体的必选参数可参考 Hadoop 工具。
- 创建 Namespace 时，如果没有指定读写策略（rPolicy/wPolicy），默认会使用配置文件中指定的 read/write type，或使用默认值（CACHE/CACHE_THROUGH）。

同理，也可以创建一个命名空间 namespace 用于挂载腾讯云 HDFS：

```
goosefs ns create MyNamespaceCHDFS ofs://xxxxx-xxxx.chdfs.ap-guangzhou.myqcloud.com/3TB \
--attribute fs.ofs.user.appid=1250000000
--attribute fs.ofs.tmp.cache.dir=/tmp/chdfs
```

2. 创建成功后，可以通过 `ls` 命令列出集群中创建的所有 namespace：

```
$ goosefs ns ls
namespace mountPoint ufsPath creationTime wPolicy rPolicy TTL ttlAction
myNamespace /myNamespace cosn://bucketName-125xxxxxx/3TB 03-11-2021 11:43:06:239 CACHE_THROUGH CACHE -1 DELETE
myNamespaceCHDFS /myNamespaceCHDFS ofs://xxxxx-xxxx.chdfs.ap-guangzhou.myqcloud.com/3TB 03-11-2021 11:45:12:336 CACHE_THROUGH CACHE -1 DELETE
```

3. 执行如下命令，指定 namespace 的详细信息。

```
$ goosefs ns stat myNamespace
NamespaceStatus{name=myNamespace, path=/myNamespace, ttlTime=-1, ttlAction=DELETE, ufsPath=cosn://bucketName-125xxxxxx/3TB, creationTimeMs=1615434186076, lastModificationTimeMs=1615436308143, lastAccessTimeMs=1615436308143, persistenceState=PERSISTED, mountPoint=true, mountId=4948824396519771065, acl=user::rwx,group::rwx,other::rwx, defaultAcl=, owner=user1, group=user1, mode=511, writePolicy=CACHE_THROUGH, readPolicy=CACHE}
```

元数据中记录的信息包括如下内容：

| 序号 | 参数 | 描述 |
| --- | --- | --- |
| 1 | name | namespace 的名字 |
| 2 | path | namespace 在 GooseFS 中的路径 |
| 3 | ttlTime | namespace 下目录和文件的 ttl 周期 |
| 4 | ttlAction | namespace 下目录和文件的 ttl 处理动作，有两种处理动作：FREE 和 DELETE，默认是 FREE |
| 5 | ufsPath | namespace 在 ufs 上的挂载路径 |
| 6 | creationTimeMs | namespace 的创建时间，单位是毫秒 |
| 7 | lastModificationTimeMs | namespace 下目录和文件的最后修改时间，单位是毫秒 |
| 8 | persistenceState | namespace 的持久化状态 |
| 9 | mountPoint | namespace 是否是一个挂载点，始终为 true |
| 10 | mountId | namespace 挂载点 ID |
| 11 | acl | namespace 的访问控制列表 |
| 12 | defaultAcl | namespace 的默认访问控制列表 |
| 13 | owner | namespace 的 owner |
| 14 | group | namespace 的 owner 所属的 group |
| 15 | mode | namespace 的 POSIX 权限 |
| 16 | writePolicy | namespace 的 写策略 |
| 17 | readPolicy | namespace 的 读策略 |

# 使用 GooseFS 预热 Table 中的数据

1. GooseFS 支持将 Hive Table 中的数据预热到 GooseFS 中，在预热之前需要先将相关的 DB 关联到 GooseFS 上，相关命令如下：

```
$ goosefs table attachdb --db test_db hive thrift://
172.16.16.22:7004 test_for_demo
```

注意：
命令中的 thrift 需要填写实际的 Hive Metastore 的地址。

2. 添加完 DB 后，可以通过 ls 命令查看当前关联的 DB 和 Table 的信息：

```
$ goosefs table ls test_db web_page
OWNER: hadoop
DBNAME.TABLENAME: testdb.web_page (
wp_web_page_sk bigint,
wp_web_page_id string,
wp_rec_start_date string,
wp_rec_end_date string,
wp_creation_date_sk bigint,
wp_access_date_sk bigint,
wp_autogen_flag string,
wp_customer_sk bigint,
wp_url string,
wp_type string,
wp_char_count int,
wp_link_count int,
wp_image_count int,
wp_max_ad_count int,
)
PARTITIONED BY (
)
LOCATION (
gfs://172.16.16.22:9200/myNamespace/3000/web_page
)
PARTITION LIST (
{
partitionName: web_page
location: gfs://172.16.16.22:9200/myNamespace/3000/web_page
}
)
```

3. 通过 load 命令预热 Table 中的数据：

```
$ goosefs table load test_db web_page
Asynchronous job submitted successfully, jobId: 1615966078836
```

预热 Table 中的数据是一个异步任务，因此会返回一个任务 ID。可以通过 goosefs job stat <Job Id> 命令查看预热作业的执行进度。当状态为 "COMPLETED" 后，则整个预热过程完成。

## 使用 GooseFS 进行文件上传和下载操作

1. GooseFS 支持绝大部分文件系统操作命令，可以通过以下命令来查询当前支持的命令列表：

```
$ goosefs fs
```

2. 可以通过 `ls` 命令列出 GooseFS 中的文件，以下示例展示如何列出根目录下的所有文件：

```
$ goosefs fs ls /
```

3. 可以通过 `copyFromLocal` 命令将数据从本地拷贝到 GooseFS 中：

```
$ goosefs fs copyFromLocal LICENSE /LICENSE
Copied LICENSE to /LICENSE
$ goosefs fs ls /LICENSE
-rw-r--r-- hadoop supergroup 20798 NOT_PERSISTED 03-26-2021 16:49:37:215 0% /LICE
NSE
```

4. 可以通过 `cat` 命令查看文件内容：

```
$ goosefs fs cat /LICENSE
Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/
TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION
...
```

5. GooseFS 默认使用本地磁盘作为底层文件系统，默认文件系统路径为 `./underFSStorage`，可以通过 `persist` 命令将文件持久化存储到本地文件系统中：

```
$ goosefs fs persist /LICENSE
persisted file /LICENSE with size 26847
```

# 使用 GooseFS 加速文件上传和下载操作

1. 检查文件存储状态，确认文件是否已被缓存。文件状态 `PERSISTED` 代表文件已在内存中，文件状态 `NOT_PERSISTED` 则代表文件不在内存中：

```
$ goosefs fs ls /data/cos/sample_tweets_150m.csv
-r-x------ staff staff 157046046 NOT_PERSISTED 01-09-2018 16:35:01:002 0% /data/c
os/sample_tweets_150m.csv
```

2. 统计文件中有多少单词 “tencent”，并计算操作耗时：

```
$ time goosefs fs cat /data/s3/sample_tweets_150m.csv | grep-c tencent
889
real 0m22.857s
user 0m7.557s
sys 0m1.181s
```

3. 将该数据缓存到内存中可以有效提升查询速度，详细示例如下：

```
$ goosefs fs ls /data/cos/sample_tweets_150m.csv
-r-x------ staff staff 157046046
ED 01-09-2018 16:35:01:002 0% /data/cos/sample_tweets_150m.csv
$ time goosefs fs cat /data/s3/sample_tweets_150m.csv | grep-c tencent
889
real 0m1.917s
user 0m2.306s
sys 0m0.243s
```

可见，系统处理延迟从1.181s减少到了0.243s，得到了10倍的提升。

# 关闭 GooseFS

通过如下命令可以关闭 GooseFS：

```
$ ./bin/goosefs-stop.sh local
```

# 核心特性
# 缓存能力

最近更新时间：2021-09-30 12:29:06

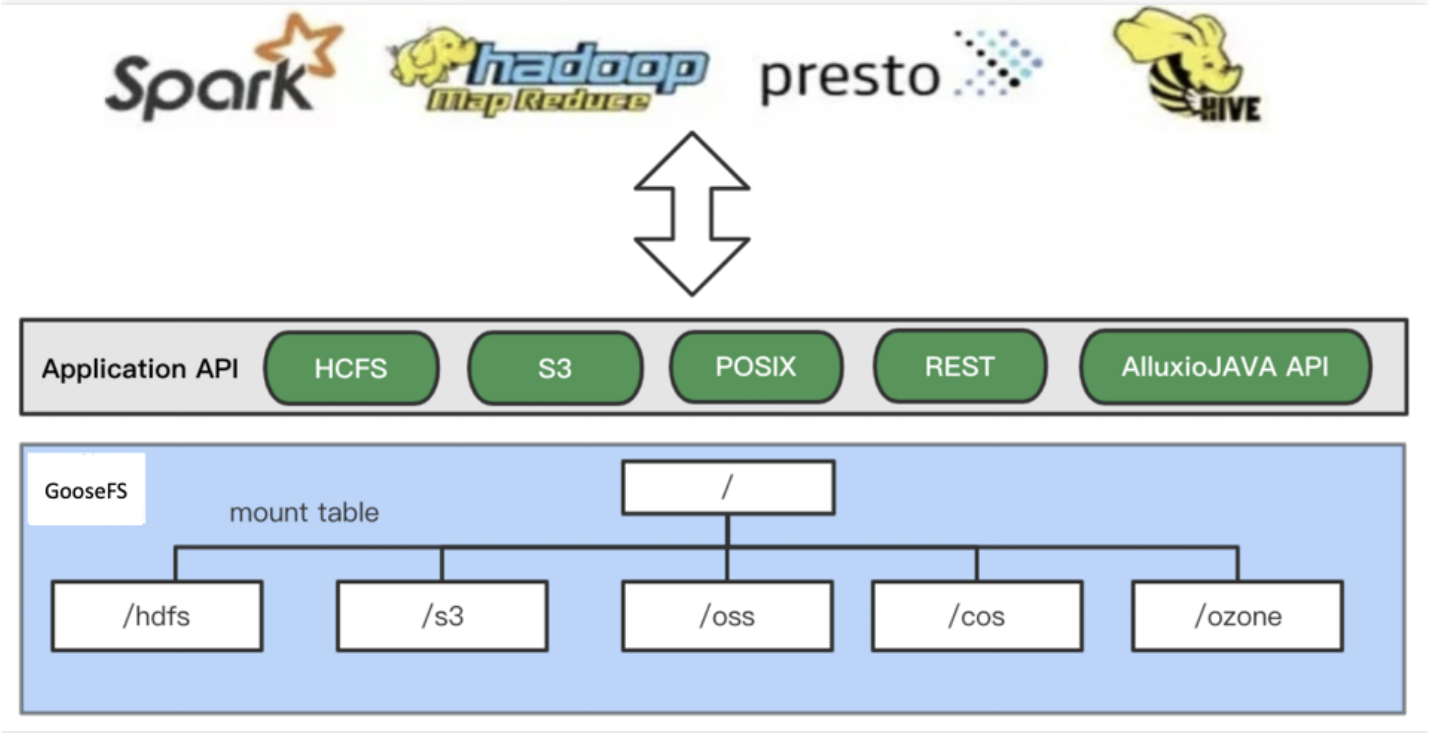
## 缓存能力概述

GooseFS 基于开源 Alluxio 的架构，提供强大的分布式 Cache 能力，能够帮助用户统一跨各种平台用户数据的同时，还有助于为用户提升总体 I/O 吞吐量。

主要通过以下两套方式：

- GooseFS NameSpace 远端存储: NS 存储空间底座来自于外部存储，可以跨越 COSN 和 HDFS，连接一个或者多个 NS 到同一个底层存储，可以持久化海量数据。选用 COSN 的存算分离方案，GooseFS NameSpace 借助 COSN 可以给大数据存储提供弹性扩展和高可用能力。
- GooseFS 本地 Cache 存储：GooseFS 通过 master 和 worker 提供的分布式 Cache 能力，将应用层产生的数据持久化在应用节点本地的内存和磁盘上，主要存储热数据和暂态数据。每个本地存储节点都可由用户配置，NameSpace 对接的远端持久化层也会经过同一个 client 服务用户的读取操作。

用户可以操作 Cache Policy，决定读写模式中使用本地 Cache 存储和 NameSpace 远端存储。



GooseFS 提供相同的本地 Cache 和远端 NameSpace 整合的能力，帮助用户全面实现存储分离的统一方案，同时建立应用节点的数据亲缘性。

# GooseFS 缓存配置

用户可以进入 goosefs-site.properties 文件中查看 GooseFS 缓存配置情况。目前GooseFS的缓存设置可以从缓存层级、缓存淘汰策略等方面了解，缓存层级方面主要有单层存储和多层存储两种，缓存淘汰策略主要区分为 LRU 和 LRFU 两种。以下进行详细介绍。

## 1. 缓存层级

GooseFS 提供单层存储和多层存储两种缓存层级。单层存储只使用一种存储介质，多层存储使用多种存储，可以根据业务负载情况来决定使用何种存储介质，以提供匹配的 I/O 性能。GooseFS 默认配置为单层存储，在多层存储的场景下，缓存淘汰会带来较多的性能开销，因此推荐使用**单层存储**。根据 I/O 性能的高低，一般可以选用 MEM（内存）、SSD（固态存储）和 HDD（硬盘存储）作为存储介质。

通过修改 goosefs-site.properties 配置文件中的 levels 参数可以修改缓存层级，入参为 1 时代表只使用单层存储，入参为 3 代表使用三层存储：

```
goosefs.worker.tieredstore.levels=1
```

通过修改 goosefs-site.properties 配置文件中的 alias 参数可以修改缓存层对应的存储介质：

```
goosefs.worker.tieredstore.level{x}.alias=MEM
```

> 注意：
> level{x} 代表第几层存储，例如 level0 代表单层存储。

### 1.1 单层存储

单层存储模式下，常用的系统配置项如下：

```
goosefs.worker.tieredstore.levels=1
goosefs.worker.tieredstore.level0.alias=HDD
goosefs.worker.ramdisk.size=16GB
goosefs.worker.memory.size=100GB
goosefs.worker.tieredstore.level0.dirs.path=/data/GooseFSWorker,/data1/GooseFSWor
ker,/data2/GooseFSWorker,/data3/GooseFSWorker,/data4/GooseFSWorker
goosefs.worker.tieredstore.level0.dirs.mediumtype=MEM,MEM,MEM,SSD,SSD
goosefs.worker.tieredstore.level0.dirs.quota=16GB,16GB,16GB,100GB,100GB
```

以下逐项介绍配置项内容。

- ramdisk.size：该配置项用于指定 worker 节点占用内存的大小。ramdisk 的容量必须比 memory 小。设置后 GooseFS 会为每个 worker 节点分配指定大小的内存空间，并消耗系统总内存。
- memory.size：该配置项用于指定 GooseFS 系统总内存，设置后 GooseFS 将会自动占用指定大小的存储介质，实际物理存储空间必须比 memory 数值大。
- dirs.path：该配置项用于指定目录，由 GooseFS 为指定目录分配存储介质，需要结合 dirs.mediumtype 使用，如上示例展示将 /data/GooseFSWorker 挂载到 MEM 存储介质上，将/data4/GooseFSWorker 挂载到 SSD 存储介质上。需要注意，每个目录的顺序需要和存储介质的顺序一一匹配。
- dirs.mediumtype：该配置项用于指定存储介质，由 GooseFS 为指定目录分配存储介质，需要结合 dirs.path 使用。默认可选的存储介质包括 MEM，SSD，如果挂载了其他存储介质，如 HDD，可按需修改配置。
- dirs.quota：改配置项用于指定每个目录的预置空间，配置后由 GooseFS 为指定目录分配好空间，需要和目录的顺序一一对应。如上示例展示了需要为 /data/GooseFSWorker,/data1/GooseFSWorker,/data2/GooseFSWorker 等目录分配 16GB MEM 空间，为 /data3/GooseFSWorker,/data4/GooseFSWorker 目录分配 100GB SSD 空间。

### 1.2 分层存储

分层存储模式下，数据块的读写模式与单层存储的读写模式存在差异。单层存储模式下数据会直接从一种存储介质中读写，多层存储模式下数据会默认优先写入最顶层存储介质，读取的时候则需要将数据先挪到最顶层存储介质。具体地：

- **数据写入**：新的数据块会被默认写入最顶层的存储介质；如果最顶层存储介质的存储空间已经满了，GooseFS 会把数据顺位存储到下一层的存储介质中；如果所有存储介质都已经存满数据，那么 GooseFS 会按照用户指定的缓存淘汰策略淘汰过期数据；如果过期数据无法淘汰，并且所有可用存储空间已满，则无法写入数据。
- **数据读取**：多层存储模式下，会将冷数据透明地转储到更低层次的存储介质中，读取数据会将数据加热并放置到顶层存储中。

> 注意：
>
> - 在既定的淘汰策略下，GooseFS 会按照指定的释放空间来清理一定量的数据，该参数可以通过 goosefs.worker.tieredstore.free.ahead.bytes 指定，默认值为0。
> - 分层存储模式下，读取数据时可能会频繁将数据从低层存储介质转储到顶层存储介质中，进而导致频繁的缓存淘汰情况，对性能带来一定损耗，一般情况下，**推荐使用单层存储**。

分层存储模式下，常用的系统配置项如下：

```
goosefs.worker.tieredstore.levels
goosefs.worker.tieredstore.level{x}.alias
goosefs.worker.tieredstore.level{x}.dirs.path
```

```
goosefs.worker.tieredstore.level{x}.dirs.mediumtype
goosefs.worker.tieredstore.level{x}.dirs.quota
```

其中，x 代表缓存层级，一般而言可以设置 3 层存储，分别对应 MEM，SSD，HDD等存储介质。以 2 层存储，存储介质分别为 MEM，SSD，每一层分配100GB存储为例，对应的配置信息如下：

```
goosefs.worker.tieredstore.levels=2
goosefs.worker.tieredstore.level0.alias=MEM
goosefs.worker.tieredstore.level0.dirs.path=/data/GooseFSWorker
goosefs.worker.tieredstore.level0.dirs.mediumtype=MEM
goosefs.worker.tieredstore.level0.dirs.quota=100GB
goosefs.worker.tieredstore.level1.alias=SSD
goosefs.worker.tieredstore.level1.dirs.path=/data1/GooseFSWorker
goosefs.worker.tieredstore.level1.dirs.mediumtype=SSD
goosefs.worker.tieredstore.level1.dirs.quota=100GB
```

GooseFS 支持配置多层存储，层数没有限制，但是每一层的命名（alias）均需要保证唯一性。一般来说，提供 3 层缓存，每一层分别对应 MEM，SSD，HDD就起到较好的分层效果。

### 2. 缓存淘汰策略

GooseFS 提供了两种缓存淘汰策略：

- LRUAnnotator：按照近期使用次数最少（least-recently-used）的顺序淘汰缓存，是 GooseFS 的默认策略。
- LRFUAnnotator：按照近期使用次数最少（least-recently-used）和近期使用频次最少（least-frequently-used）的顺序淘汰缓存，通过权重配置 goosefs.worker.block.annotator.lrfu.step.factor 和 goosefs.worker.block.annotator.lrfu.attenuation.factor 来调整两种策略在淘汰过程中起的作用。
  - 如果将 least-recently-used 的权重调整至最大，那么该策略的表现与 LRUAnnotator一致。

可以在 goosefs.worker.block.annotator.class 这一配置项中配置缓存淘汰策略，配置时需要正确指定策略名称：

```
goosefs.worker.block.annotator.LRUAnnotator
goosefs.worker.block.annotator.LRFUAnnotator
```

## 数据生命周期管理

这里的生命周期是指在 GooseFS 中的数据生命周期，而不是远端存储系统 UFS 的，生命周期管理主要有如下四种操作：

- free：此操作用于从 GooseFS 中删除对应目录或者文件的数据（不会删除 UFS 中的对应数据），此类操作主要是用来释放 GooseFS 的缓存空间，供其他更热的数据使用。

- load：此操作用于将 UFS 对应目录或者文件的数据加载到 GooseFS 中，此类操作主要用于冷数据转热，从而提高 I/O 性能。
- persist：此操作用于将 GooseFS 内的数据写入到 UFS 存储中，此类操作主要用于持久化数据，从而避免数据丢失。
- TTL(Time to Live)：TTL 属性主要用于设置目录或者文件在 GooseFS 中的生命周期，在超过其生命周期后，会从 GooseFS 和 UFS 中删除。通过配置，也可支持仅删除 GooseFs 数据或仅释放磁盘空间。

### 1. 从 GooseFS 中释放数据

通过 free 指令可以从 GooseFS 中释放数据，如下示例展示了释放数据后，GooseFS 中数据的状态变成 0%：

```
$ goosefs fs free /data/test.txt
/data/test.txt was successfully freed from GooseFS space.
$ goosefs fs ls /data/test.txt
-rw-rw-rw- hadoop hadoop 14 PERSISTED 03-11-2021 11:46:15:000 0% /data/test.txt
```

示例中的 /data/test.txt 可以是任何合法的 GooseFS 文件路径，如果数据存储于远端存储 UFS 中，则可以通过 load 指令重新加载。

> 注意：
> 一般而言，推荐通过配置缓存策略自动清理 GooseFS 中的历史数据。

### 2. 往 GooseFS 中加载数据

通过 load 指令可以从 GooseFS 中加在数据，如下示例展示了释放数据后，GooseFS 中数据的状态变成 100% ：

```
$ goosefs fs load /data/test.txt
/data/test.txt loaded
$ goosefs fs ls /data/test.txt
-rw-rw-rw- hadoop hadoop 14 PERSISTED 03-11-2021 11:46:15:000 100% /data/test.txt
```

示例中的 /data/test.txt 可以是任何合法的 GooseFS 文件路径。如果是从本地文件系统中加载数据，则需要使用 copyFromLocal 指令，需要注意的是，从本地文件系统中加载数据的操作并不会将数据持久化到远端存储 UFS 中。在默认情况下， GooseFS 能在首次访问文件时自动加载数据到 GooseFS 到缓存中，因此一般无需使用该指令加载数据；但如果业务需要提前加载数据到缓存中，可以通过该指令加载。

### 3. 在 GooseFS 中持久化数据

通过 persit 指令可以将数据持久化到远端存储系统 UFS 中：

```
$ goosefs fs persist /data/test.txt
$ goosefs fs ls /data/test.txt
```

```
-rw-rw-rw- hadoop hadoop 14 PERSISTED 03-11-2021 11:46:15:000 100% /data/test.txt
```

示例中的 /data/test.txt 可以是任何合法的 GooseFS 文件路径。推荐通过缓存策略配置来自动执行持久化的操作。

### 4. 设置数据 TTL 属性

GooseFS 支持为命名空间中的任意文件或者目录设置 TTL 属性。该属性可以保证业务数据可以按照指定的时间周期定期删除，为新文件释放空间，保证本地磁盘空间的有效利用。 GooseFS 文件和目录的 TTL 属性可以作为文件元数据的一部分，可以保证在集群重启后 TTL 属性依然生效，后端线程的定期巡检程序会检查这些 TTL 属性并自动清理过期文件。

定期巡检程序的巡检周期和巡检类型可以在 goosefs-site.preperties 中设置，以下示例将巡检周期设置为 10 分钟，巡检类型设置为 FREE ：

```
goosefs.master.ttl.checker.interval=10m
goosefs.user.file.create.ttl.action=FREE
```

设置完毕后，GooseFS 后台线程会每隔 10 分钟巡检一遍，当次巡检周期内发现的过期文件会在下一次巡检周期过完后释放缓存资源。如 00:00:00 进行了第一次巡检发现一批过期文件，这批文件缓存会在 00:10:00 清除。

> 注意：
> 默认情况下该入参单位为毫秒（ms），可以在入参时指定好检测时间周期的单位，如秒（s）,分钟（m），小时（h）等，更多说明可以查看配置说明的介绍。

## 数据复制管理

数据复制主要分为被动复制和主动复制两种形式。

### 1. 被动复制

GooseFS中的每个文件都包含一个或多个分布在集群中存储的存储块，默认情况下 GooseFS 可以根据负载和容量情况自动调整数据分块的复制级别。被动复制会发生在以下场景：

- 多个客户端同时读取相同的块，会导致多个 block 同时存在在不同的 worker 上。
- 当开启优先本地读时，数据不在本地，则会发起 remote read，并保存在本地 worker 上。
- 当被动复制产生的副本数大于设定的副本数目时，其会异步的去删除多余的副本。以上过程对用户完全透明。

### 2. 主动复制

主动复制是通过设置文件的副本数目实现的，对于不满足设定副本数的 block，其会异步的补足，对于超过设定副本数的 block，其会删除多余的副本。具体的，可以通过如下指令调整主动复制的级别：

```
$ goosefs fs setReplication [-R] [--max | --min ] <path>
```

相关参数说明如下：

- max：指定文件的最大副本数，默认值为 -1，表示不设置上限；如果设置为 0 代表不在 GooseFS 中存储指定文件的冷数据。一般设置为正整数，设置后 GooseFS 会检查文件副本数量并删除多余的副本。
- min：指定文件的最小副本数，默认值为0，代表 GooseFS 会在数据变冷后删除该数据不留存任何副本。一般设置为正整数，并且**需要小于 max 值**，设置后 GooseFS 会检查文件副本数量，如果副本数量比最小值小会自动补齐副本数。
- path：指定的文件名，可以为目录或者具体的文件路径。
- R：如果 path 中入参为目录，则指定 -R 可以按照指定的最小副本数和最大副本数，重复递归复制指定目录下的所有文件和子目录。

如果需要查看指定文件的复制情况，可以通过 stat 指令查看，如下示例展示 /data/test.txt 这个文件的最大副本数为 -1，意味着该文件变冷后会被过期删除：

```
$ goosefs fs stat /data/test.txt
/data/test.txt is a file path.
FileInfo{fileId=50331647, fileIdentifier=null, name=test.txt, path=/data/test.tx
t, ufsPath=hdfs://172.16.16.16:4007/data/test.txt, length=0, blockSizeBytes=13421
7728, creationTimeMs=1618193473555, completed=true, folder=false, pinned=false, p
innedlocation=[], cacheable=true, persisted=true, blockIds=[], inMemoryPercentage
=100, lastModificationTimesMs=1616763603692, ttl=-1, lastAccessTimesMs=1616763603
692, ttlAction=DELETE, owner=hadoop, group=supergroup, mode=420, persistenceState
=PERSISTED, mountPoint=false, replicationMax=-1, replicationMin=0, fileBlockInfos
=[], mountId=1, inGooseFSPercentage=100, ufsFingerprint=TYPE|FILE UFS|hdfs OWNER|
hadoop GROUP|supergroup MODE|420 CONTENT_HASH|(len:0,_modtime:1616763603692) , ac
l=user::rw-,group::r--,other::r--, defaultAcl=}
This file does not contain any blocks.
```

# 查看缓存用量

GooseFS 会记录本地缓存的容量和使用情况，可以通过如下指令查看 GooseFS 的运行情况，针对性地管理和维护本地缓存。

- 查看 GooseFS 正在使用的缓存，单位为字节：

```
$ goosefs fs getUsedBytes
Used Bytes: 0
```

- 查看 GooseFS 总缓存容量，单位为字节：

```
$ goosefs fs getCapacityBytes
Capacity Bytes: 1610612736000
```

- 查看 GooseFS 的缓存使用报告：

```
$ goosefs fsadmin report
GooseFS cluster summary:
Master Address: 172.16.16.16:19998
Web Port: 19999
Rpc Port: 19998
Started: 04-12-2021 10:52:05:255
Uptime: 0 day(s), 1 hour(s), 28 minute(s), and 57 second(s)
Version: 2.5.0-SNAPSHOT
Safe Mode: false
Zookeeper Enabled: false
Live Workers: 3
Lost Workers: 0
Total Capacity: 1500.00GB
Tier: HDD Size: 1500.00GB
Used Capacity: 0B
Tier: HDD Size: 0B
Free Capacity: 1500.00GB
```

# 透明加速能力

最近更新时间：2021-11-19 14:51:04

## 概述

透明加速能力用于加速 CosN 访问 COS 的性能。CosN 工具 是基于腾讯云对象存储（Cloud Object Storage, COS）提供的标准的 Hadoop 文件系统实现，可以为 Hadoop、Spark 以及 Tez 等大数据计算框架集成 COS 提供支持。用户可使用实现了 Hadoop 文件系统接口的 CosN 插件，读写存储在 COS 上的数据。对于已经使用 CosN 工具访问 COS 的用户，GooseFS 提供了一种客户端路径映射方式，让用户可以在不修改当前 Hive table 定义的前提下，仍然能够使用 CosN scheme 访问 GooseFS，该特性方便用户在不修改已有表定义的前提下，对 GooseFS 的功能和性能进行对比测试。对于云 HDFS 用户（CHDFS），也可以通过修改配置，实现使用 OFS Scheme 访问 GooseFS 的目的。

CosN Schema 和 GooseFS Schema 的映射说明如下：

假设 Namespace warehouse 对应的 UFS 路径为 `cosn://examplebucket-1250000000/data/warehouse/`，则 CosN 到 GooseFS 的路径映射关系如下：

```
cosn://examplebucket-1250000000/data/warehouse -> /warehouse/
cosn://examplebucket-1250000000/data/warehouse/folder/test.txt ->/warehouse/folde
r/test.txt
```

GooseFS 到 CosN 的路径映射关系如下：

```
/warehouse ->cosn://examplebucket-1250000000/data/warehouse/
/warehouse/ -> cosn://examplebucket-1250000000/data/warehouse/
/warehouse/folder/test.txt -> cosn://examplebucket-1250000000/data/warehouse/fold
er/test.txt
```

CosN Scheme 访问 GooseFS 特性，通过在客户端维持 GooseFS 路径和底层文件系统 CosN 路径之间的映射关系，并将 CosN 路径的请求转换为 GooseFS 的请求。映射关系周期性刷新，您可以通过修改 GooseFS 配置文件 goosefs-site.properties 中的配置项 goosefs.user.client.namespace.refresh.interval 调整刷新间隔，默认值为60s。

> 注意：
> 如果访问的 CosN 路径无法转换为 GooseFS 路径，对应的 Hadoop API 调用会抛出异常。

## 操作示例

该示例演示了 Hadoop 命令行以及 Hive 中，如何使用 gfs://、 cosn://、ofs:// 三种 Schema 访问 GooseFS。操作流程如下：

### 1. 准备数据和计算集群

- 参考 创建存储桶 文档，创建一个测试用途的存储桶。
- 参考 创建文件夹 文档，在存储桶根路径下创建一个名为 ml-100k 的文件夹。
- 从 Grouplens 下载 ml-100k 数据集，并将文件 u.user 上传到 `<存储桶根路径>/ml-100k` 下。
- 参考 EMR 指引文档，购买一个 EMR 集群并配置 HIVE 组件。

### 2. 环境配置

i. 将 GooseFS 的客户端 jar 包（goosefs-1.0.0-client.jar）放入 share/hadoop/common/lib/ 目录下：

```
cp goosefs-1.0.0-client.jar hadoop/share/hadoop/common/lib/
```

> 注意
> 配置变更和添加 jar 包，需同步到集群上所有节点。

ii. 修改 Hadoop 配置文件 etc/hadoop/core-site.xml，指定 GooseFS 的实现类：

```
<property>
<name>fs.AbstractFileSystem.gfs.impl</name>
<value>com.qcloud.cos.goosefs.hadoop.GooseFileSystem</value>
</property>
<property>
<name>fs.gfs.impl</name>
<value>com.qcloud.cos.goosefs.hadoop.FileSystem</value>
</property>
```

iii. 执行如下 Hadoop 命令，检查是否能够通过 gfs:// Scheme 访问 GooseFS，其中 &dxlt;MASTER_IP> 为 Master 节点的 IP：

```
hadoop fs -ls gfs://<MASTER_IP>:9200/
```

iv. 将 GooseFS 的客户端 jar 包放到 Hive 的 auxlib 目录下，使得 Hive 能加载到 GooseFS Client 包：

```
cp goosefs-1.0.0-client.jar hive/auxlib/
```

v. 执行如下命令，创建 UFS Scheme 为 CosN 的 Namespace，并列出 Namespace。您可将该命令中的 examplebucket-1250000000 替换为你的 COS 存储桶，SecretId 和 SecretKey 替换为您的密钥信息：

```
goosefs ns create ml-100k cosn://examplebucket-1250000000/ml-100k --secret fs.cos
n.userinfo.secretId=SecretId --secret fs.cosn.userinfo.secretKey=SecretKey--attri
bute fs.cosn.bucket.region=ap-guangzhou --attribute fs.cosn.credentials.provider=
org.apache.hadoop.fs.auth.SimpleCredentialProvider
goosefs ns ls
```

vi. 执行命令，创建 UFS Scheme 为 OFS 的 Namespace，并列出 Namespace。您可将该命令中的 instance-id 替换为您的 CHDFS 实例，1250000000 替换为您的 APPID：

```
goosefs ns create ofs-test ofs://instance-id.chdfs.ap-guangzhou.myqcloud.com/ofs-
test --attribute fs.ofs.userinfo.appid=1250000000
goosefs ns ls
```

### 3. 创建 GooseFS Schema 表和查询数据

通过如下指令执行：

```
create database goosefs_test;
use goosefs_test;
CREATE TABLE u_user_gfs (
userid INT,
age INT,
gender CHAR(1),
occupation STRING,
zipcode STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE
LOCATION 'gfs://<MASTER_IP>:<MASTER_PORT>/ml-100k';
select sum(age) from u_user_gfs;
```

### 4. 创建 CosN Schema 表和查询数据

通过如下指令执行：

```
CREATE TABLE u_user_cosn (
userid INT,
age INT,
gender CHAR(1),
occupation STRING,
zipcode STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE
```

```
LOCATION 'cosn://examplebucket-1250000000/ml-100k';
select sum(age) from u_user_cosn;
```

### 5. 修改 CosN 的实现为 GooseFS 的兼容实现

修改 hadoop/etc/hadoop/core-site.xml：

```
<property>
<name>fs.AbstractFileSystem.cosn.impl</name>
<value>com.qcloud.cos.goosefs.hadoop.CosN</value>
</property>
<property>
<name>fs.cosn.impl</name>
<value>com.qcloud.cos.goosefs.hadoop.CosNFileSystem</value>
</property>
```

执行 Hadoop 命令，如果路径无法转换为 GooseFS 中的路径，命令的输出中会包含报错信息：

```
hadoop fs -ls cosn://examplebucket-1250000000/ml-100k/
Found 1 items
-rw-rw-rw- 0 hadoop hadoop 22628 2021-07-02 15:27 cosn://examplebucket-125000000
0/ml-100k/u.user
hadoop fs -ls cosn://examplebucket-1250000000/unknow-path
ls: Failed to convert ufs path cosn://examplebucket-1250000000/unknow-path to Goo
seFs path, check if namespace mounted
```

重新执行 Hive 查询语句：

```
select sum(age) from u_user_cosn;
```

### 6. 创建 OFS Schema 表和查询数据

通过如下命令执行：

```
CREATE TABLE u_user_ofs (
userid INT,
age INT,
gender CHAR(1),
occupation STRING,
zipcode STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE
LOCATION 'ofs://instance-id.chdfs.ap-guangzhou.myqcloud.com/ofs-test/';
select sum(age) from u_user_ofs;
```

### 7. 修改 OFS 的实现为 GooseFS 的兼容实现

修改 hadoop/etc/hadoop/core-site.xml：

```
<property>
<name>fs.AbstractFileSystem.ofs.impl</name>
<value>com.qcloud.cos.goosefs.hadoop.CHDFSDelegateFS</value>
</property>
<property>
<name>fs.ofs.impl</name>
<value>com.qcloud.cos.goosefs.hadoop.CHDFSHadoopFileSystem</value>
</property>
```

执行 Hadoop 命令，如果路径无法转换为 GooseFS 中的路径，则输出结果中会包含报错信息：

```
hadoop fs -ls ofs://instance-id.chdfs.ap-guangzhou.myqcloud.com/ofs-test/
Found 1 items
-rw-r--r-- 0 hadoop hadoop 22628 2021-07-15 15:56 ofs://instance-id.chdfs.ap-guan
gzhou.myqcloud.com/ofs-test/u.user
hadoop fs -ls ofs://instance-id.chdfs.ap-guangzhou.myqcloud.com/unknown-path
ls: Failed to convert ufs path ofs://instance-id.chdfs.ap-guangzhou.myqcloud.com/
unknown-path to GooseFs path, check if namespace mounted
```

重新执行 Hive 查询语句：

```
select sum(age) from u_user_ofs;
```

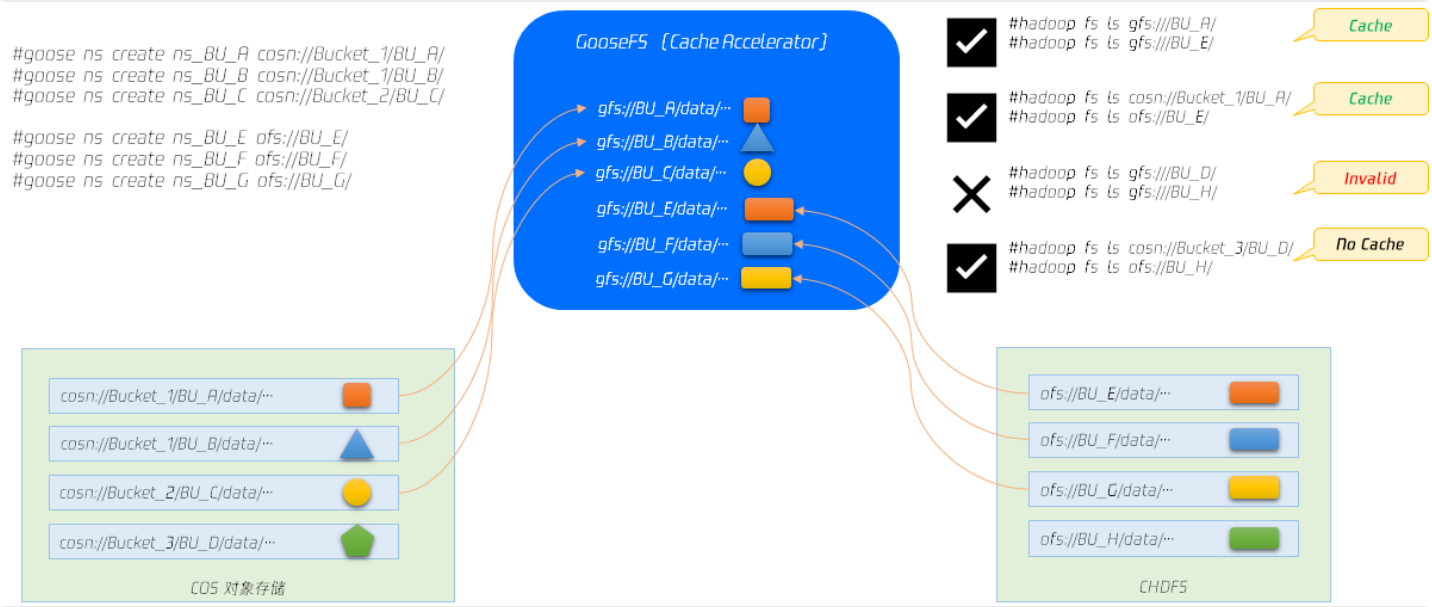
# 统一命名空间能力

最近更新时间：2021-11-22 14:27:16

## 统一命名空间能力概述

GooseFS 统一命名空间（NameSpace）能力通过透明的命名机制，可以融合多种不同的底层存储系统访问语义，为用户提供了一个统一的数据管理交互视图。

GooseFS 通过统一命名空间能力对接不同的底层存储系统，如本地文件系统、腾讯云对象存储（Cloud Object Storage，COS）、腾讯云云 HDFS（Cloud HDFS，CHDFS）等，与这些底层存储系统进行通信，并为上层业务提供统一的访问接口和文件协议。业务侧只需使用 GooseFS 的访问接口，即可访问存储在不同底层存储系统中的数据。




上图显示了统一命名空间的工作原理。您可以通过 GooseFS 创建命名空间的指令 create ns ，将 COS 和 云 HDFS 的指定文件目录挂载到 GooseFS 中，然后通过 gfs:// 的这一统一的 schema 访问数据。详细说明如下所示：

- COS 总共有3个存储桶，分别是 bucket-1、bucket-2、bucket-3，其中 bucket-1 有 BU_A 和 BU_B 两个目录，bucket-1 和 bucket-2 均挂载在 GooseFS 中。
- 云 HDFS 中有 BU_E、BU_F、BU_G 和 BU_H 共 4 个目录，其中除了 BU_H 其余目录均挂载到 GooseFS 上。
- 在 GooseFS 的文件操作中，如果以 gfs:// 这一统一的 schema 访问 BU_A 和 BU_E 这两个目录，均可正常访问，且文件缓存在 GooseFS 的本地文件系统中。
- BU_A 和 BU_E 这两个存储在底层文件系统（COS、云 HDFS）中的目录已经挂载到 GooseFS 中，如果文件已经缓存在 GooseFS 的上，可以通过 gfs:// 这一统一的 schema 访问（例如 hadoop fs ls gfs://BU_A ）；也可以通过

各个远端文件系统的命名空间访问（例如 hadoop fs ls cosn://bucket-1/BU_A ）。

- 如果文件未被缓存在 GooseFS 上，此时通过 gfs:// 这一形式访问则会失败，因为文件未被缓存在本地文件系统中，但仍然可以通过底层存储系统的命名空间访问。

# 使用统一命名空间能力

您可以通过 ns 操作在 GooseFS 中创建命名空间，并将底层存储系统映射到 GooseFS 上，目前支持的底层存储系统包括 COS、云 HDFS、本地 HDFS 等。创建命名空间的操作与 Linux 文件系统中挂载文件卷盘类似。创建命名空间后，GooseFS 可以为客户端提供一个具有统一访问语义的文件系统。目前 GooseFS 命名空间的操作指令集如下：

```
$ goosefs ns
Usage: goosefs ns [generic options]
[create <namespace> <CosN/Chdfs path> <--wPolicy <1-6>> <--rPolicy <1-5>> [--read
only] [--shared] [--secret fs.cosn.userinfo.secretId=<AKIDxxxxxxx>] [--secret fs.
cosn.userinfo.secretKey=<xxxxxxxxxx>] [--attribute fs.ofs.userinfo.appid=12000000
00][--attribute fs.cosn.bucket.region=<ap-xxx>/fs.cosn.bucket.endpoint_suffix=<co
s.ap-xxx.myqcloud.com>]]
[delete <namespace>]
[help [<command>]]
[ls [-r|--sort=option|--timestamp=option]]
[setPolicy [--wPolicy <1-6>] [--rPolicy <1-5>] <namespace>]
[setTtl [--action delete|free] <namespace> <time to live>]
[stat <namespace>]
[unsetPolicy <namespace>]
[unsetTtl <namespace>]
```

上述指令集中各项指令的能力简述如下：

| 指令 | 说明 |
| --- | --- |
| create | 用于创建命名空间，将一个远端存储系统（UFS）映射到命名空间中；支持在创建命名空间时设置读写缓存策略；需要传入有权限的密钥信息（secretId、secretKey）。 |
| delete | 用于删除指定命名空间。 |
| ls | 用于列出指定命名空间的详细信息，例如挂载点、UFS 路径、创建时间、缓存策略、TTL 信息等。 |
| setPolicy | 用于设置指定命名空间的缓存策略。 |
| setTtl | 用于设置指定命名空间的 TTL。 |
| stat | 用于提供指定命名空间的描述性信息，例如挂载点、UFS 路径、创建时间、缓存策略、TTL 信息、持久化状态、用户组、ACL、最后一次访问时间、修改时间等。 |

| 指令 | 说明 |
| --- | --- |
| unsetPolicy | 用于重置指定命名空间的缓存策略。 |
| unsetTtl | 用于重置指定命名空间的 TTL。 |

### 创建和删除命名空间

通过 GooseFS 创建命名空间操作，可以将频繁访问的热数据从远端存储系统缓存到本地高性能存储节点中，为本地计算业务提供高性能的数据访问。如下指令展示了将 COS 中的存储桶 example-bucket、存储桶中的 example-prefix 目录以及云 HDFS 文件系统分别映射到 test_cos、test_cos_prefix 和 test_chdfs 等命名空间下。

```
# 将 COS 存储桶 example-bucket 映射到 test_cos 命名空间中
$ goosefs ns create test_cos cosn://example-bucket-1250000000/ --wPolicy 1 --rPol
icy 1 --secret fs.cosn.userinfo.secretId=AKIDxxxxxxx --secret fs.cosn.userinfo.se
cretKey=xxxxxxxxxx --attribute fs.cosn.bucket.region=ap-guangzhou --attribute fs.
cosn.bucket.endpoint_suffix=cos.ap-guangzhou.myqcloud.com
# 将 COS 存储桶 example-bucket 的 example-prefix 目录映射到 test_cos_prefix 命名空间中
$ goosefs ns create test_cos_prefix cosn://example-bucket-1250000000/example-pref
ix/ --wPolicy 1 --rPolicy 1 --secret fs.cosn.userinfo.secretId=AKIDxxxxxxx --secr
et fs.cosn.userinfo.secretKey=xxxxxxxxxx --attribute fs.cosn.bucket.region=ap-gua
ngzhou --attribute fs.cosn.bucket.endpoint_suffix=cos.ap-guangzhou.myqcloud.com
# 将 云HDFS 文件系统 f4ma0l3qabc-Xy3 映射到 test_chdfs 命名空间中
$ goosefs ns create test_chdfs ofs://f4ma0l3qabc-Xy3/ --wPolicy 1 --rPolicy 1 --a
ttribute fs.ofs.userinfo.appid=1250000000
```

创建成功后，可以通过 goosefs fs ls 指令查看目录详情：

```
$ goosefs fs ls /test_cos
```

对于不需要使用的命名空间，可以通过 delete 指令来删除：

```
$ goosefs ns delete test_cos
Delete the namespace: test_cos
```

### 设置缓存策略

用户可通过 setPolicy 和 unsetPolicy 设置指定命名空间的缓存策略。设置缓存策略的指令集如下：

```
$goosefs ns setPolicy [--wPolicy <1-6>] [--rPolicy <1-5>] <namespace>
```

其中各项参数的含义如下：

- wPolicy：写缓存策略，支持6种写缓存策略。

- rPolicy：读缓存策略，支持5种读缓存策略。
- namespace：指定的命名空间。

目前 GooseFS 支持的读写缓存策略分别如下：

**写缓存策略**

| 策略名字 | 行为 | 对应 Write_Type | 数据安全性 | 写效率 |
| --- | --- | --- | --- | --- |
| MUST_CACHE（1） | 数据仅存储在 GooseFS，不会写入远端存储系统中。 | MUST_CACHE | 不可靠 | 高 |
| TRY_CACHE（2） | 缓存有空间时就写入 GooseFS 中，缓存如果没空间则直接写入到底层存储中。 | TRY_CACHE | 不可靠 | 中 |
| CACHE_THROUGH（3） | 尽量缓存数据，同时同步写入远端存储系统。 | CACHE_THROUGH | 可靠 | 低 |
| THROUGH（4） | 数据不存储在 GooseFS，直接写远端存储系统。 | THROUGH | 可靠 | 中 |
| ASYNC_THROUGH（5） | 数据写入 GooseFS 中，并异步刷新到远端存储系统。 | ASYNC_THROUGH | 弱可靠 | 高 |

说明：
Write_Type：指用户调用 SDK 或者 API 向 GooseFS 中写入数据时指定的文件缓存策略，对单个文件生效。

**读缓存策略**

| 策略名字 | 行为 | 元数据同步 | 对应 Read_Type |
| --- | --- | --- | --- |
| NO_CACHE（1） | 不缓存数据，直接从远端存储系统中读数据。 | NO | NO_CACHE |
| CACHE（2） | • 元数据访问行为：如果命中缓存时，元数据以 Master 中的为准，不会主动从底层同步元数据。<br>• 数据流访问行为：数据流的 Read_Type 采用 CACHE 策略。 | Once | CACHE |

| 策略名字 | 行为 | 元数据同步 | 对应 Read_Type |
| --- | --- | --- | --- |
| CACHE_PROMOTE（3） | • 元数据访问行为：与 CACHE 模式相同。<br>• 数据流访问行为：数据流的 Read_Type 采用 CACHE_PROMOTE 策略。 | Once | CACHE_PROM |
| CACHE_CONSISTENT_PROMOTE（4） | • 元数据行为：每次读取操作前均先同步远端存储系统 UFS 上的元数据，如果 UFS 中不存在，则抛出异常 Not Exists。<br>• 数据流访问行为：数据流的 Read_Type 采用 CACHE_PROMOTE 策略，命中以后，缓存到最热的缓存介质中。 | Always | CACHE |
| CACHE_CONSISTENT（5） | • 元数据行为：与 CACHE_CONSISTENT_PROMOTE 相同。<br>• 数据流访问行为：数据流的 Read_Type 采用 CACHE 策略，即 CACHE 命中，不会在不同的介质层中移动数据。 | Always | CACHE_PROM |

说明：

Read_Type：指用户调用 SDK 或者 API 从 GooseFS 中读取数据时指定的文件缓存策略，对单个文件生效。

结合目前大数据的业务实践，我们推荐的读写缓存策略组合主要如下：

| 写缓存策略 | 读缓存策略 | 策略组合表现 |
| --- | --- | --- |
| CACHE_THROUGH（3） | CACHE_CONSISTENT（5） | 缓存和远端存储系统数据强一致。 |
| CACHE_THROUGH（3） | CACHE（2） | 写强一致性，读最终一致性。 |
| ASYNC_THROUGH（5） | CACHE_CONSISTENT（5） | 写最终一致性，读强一致性。 |
| ASYNC_THROUGH（5） | CACHE（2） | 读写最终一致性。 |
| MUST_CACHE（1） | CACHE（2） | 只从缓存中读数据。 |

如下示例展示了将指定命名空间 test_cos 的读写缓存策略分别设置为 CACHE_THROUGH 和 CACHE_CONSISTENT：

```
$ goosefs ns setPolicy --wPolicy 3 --rPolicy 5 test_cos
```

注意：
除了在创建命名空间时指定缓存策略，用户还可以通过在读写文件时，针对指定文件设置 Read_Type 或者 Write_Type，或者通过 properties 配置文件配置全局缓存策略。多个策略同时存在的时候，优先级为用户自定义优先级 > Namespace 读写策略 > 配置文件的全局缓存策略配置。其中，针对读策略会采用用户自定义 Read_Type 和 Namespace 的 DirReadPolicy 的组合生效，即数据流读策略采用用户自定义 Read_Type，元数据采用 Namespace 的策略。

例如，GooseFS 中存在一个 COSN 命名空间，读策略为 CACHE_CONSISTENT；假设在该命名空间中存在一个 test.txt 的文件。客户端读取 test.txt 时，Read_Type 指定了 CACHE_PROMOTE。那么整个读取行为就是同步元数据并且 CACHE_PROMOTE。

如果需要重置读写缓存策略，可以通过 unsetPolicy 指令实现，如下策略展示了重置 test_cos 命名空间的读写缓存策略：

```
$ goosefs ns unsetPolicy test_cos
```

### 设置 TTL

TTL 用于管理缓存在 GooseFS 本地节点的数据，配置 TTL 参数可以让缓存数据在指定的时间后执行指定的操作，例如 delete 或者 free 操作。目前设置 TTL 的操作指令如下：

```
$ goosefs ns setTtl [--action delete|free] <namespace> <time to live>
```

其中各项参数的含义如下：

- action：缓存时间到期后执行的操作，目前支持 delete 和 free 两种操作。delete 操作会删除缓存和 UFS 上的数据，free 操作只会删除缓存上的数据。
- namespace：指定的命名空间。
- time to live：数据缓存时间，单位毫秒。

如下示例展示了将指定命名空间 test_cos 的策略设置为60秒到期后删除：

```
$ goosefs ns setTtl --action free test_cos 60000
```

# 元数据信息管理

本小节介绍 GooseFS 如何管理元数据，包括元数据的同步、更新等内容。GooseFS 为用户提供了统一命名空间能力，用户可以通过统一的 gfs:// 的路径来访问不同底层存储系统上的文件，只需要指定好底层存储系统的路径即可。我们推荐您使用 GooseFS 作为统一的数据接入层，统一从 GooseFS 进行数据读写，保障元数据信息的一致性。

## 元数据同步概述

您可以通过在 conf/goosefs-site.properties 配置文件中修改元数据同步周期来管理元数据同步周期，配置参数如下：

```
goosefs.user.file.metadata.sync.interval=<INTERVAL>
```

同步周期支持如下3种入参：

- 入参数值为-1：代表元数据在首次加载到 GooseFS 中后就不再进行更新。
- 入参数值为0：代表 GooseFS 会在每次读写操作后都更新元数据。
- 入参数值为正整数：代表 GooseFS 会按照指定的时间间隔，周期性地更新元数据。

您可以综合考虑您的节点数量、GooseFS 集群和底层存储的 I/O 距离、底层存储类型等因素，选择合适的同步周期。通常：

- GooseFS 集群节点数量越多，元数据同步延迟越大。
- GooseFS 集群所处的机房和底层存储的物理距离越远，元数据同步延迟越大。
- 底层存储系统对元数据同步延迟的影响主要视系统请求 QPS 负载情况而定，QPS 负载越高则同步延迟相对更低。

## 元数据同步管理方式

### 配置方式

1. 通过命令行配置
   您可以通过命令行的方式设置元数据信息同步周期：

   ```
   goosefs fs ls -R -Dgoosefs.user.file.metadata.sync.interval=0 <path to sync>
   ```

2. 通过配置文件配置
   对于大规模的 Goosefs 集群而言，您可以通过 goosefs-site.properties 配置文件批量配置集群中 Master 节点的元数据信息同步周期，其他节点的同步周期会默认按照该周期值执行。

   ```
   goosefs.user.file.metadata.sync.interval=1m
   ```

注意：
很多业务会选择按照目录来区分数据的用途，不同目录的数据访问频次并不全部相同。元数据同步周期可以按照不同目录设置不同的周期值，对于一些经常变化的目录，可以将该目录的元数据同步周期设置为较短的时间（如5分钟），对于极少变化或者不变化的目录，可以将同步周期设置为-1，这样 GooseFS 不会自动同步该目录的元数据。

**推荐配置**

根据业务访问模式的差异，您可以设置不同的元数据同步周期：

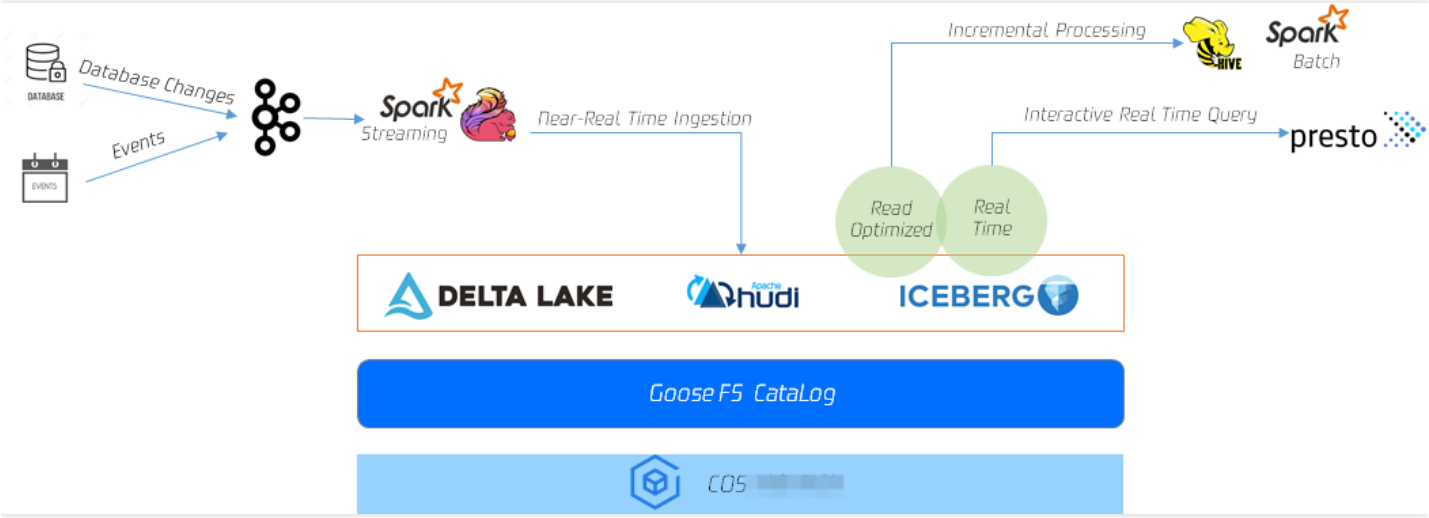
<table>
<tr><th colspan="2">访问模式</th><th>元数据同步周期</th><th>说明</th></tr>
<tr><td colspan="2">所有文件请求都经过 GooseFS</td><td>-1</td><td>-</td></tr>
<tr><td rowspan="2">绝大部分文件请求都经过 GooseFS</td><td>使用 HDFS 作为 UFS</td><td>推荐使用热更新或者按路径更新</td><td>如果 HDFS 的更新特别频繁，推荐将更新周期设置为-1，禁止更新</td></tr>
<tr><td>使用 COS 作为 UFS</td><td>推荐按照路径配置更新周期</td><td rowspan="3">按照不同目录配置不同的更新周期，可以缓解元数据同步的压力</td></tr>
<tr><td rowspan="2">上传文件请求一般不经过 GooseFS</td><td>使用 HDFS 作为 UFS</td><td>推荐按照路径配置更新周期</td></tr>
<tr><td>使用 COS 作为 UFS</td><td>推荐按照路径配置更新周期</td></tr>
</table>

# Table 管理能力

最近更新时间：2021-09-15 13:00:03

## Table 管理能力概述

GooseFS Table 管理能力用于管理结构化数据，为 SparkSQL、Hive、Presto 等上层计算应用提供数据库表管理能力，目前底层支持对接 Hive MetaStore。Table 管理能力能够帮助各类 SQL 引擎读取指定的数据内容，能够有效提升大数据场景下对数据的访问效率。




GooseFS Table 管理能力目前主要支持了以下特性：

- 元数据层面的描述能力。GooseFS Catalog 提供源自远程元数据服务（Hive MetaStore）的元数据缓存服务，针对 SparkSQL，Hive，SQL Presto 等 SQL 引擎做查询时，可以根据 GooseFS Catalog 中的元数据缓存服务来确定读取数据大小、目标数据位置以及数据结构，具备与 Hive MetaStore 相同的能力表现。
- 表级数据预缓存能力。GooseFS Catalog 能够感知数据表和数据存储路径的对应关系，进而可以提供 Table 级别以及 Table Partition 级别的缓存预热能力，帮助用户提前按照表结构缓存数据，极大提高访问性能。
- 跨存储服务的统一元数据服务。通过 GooseFS Catalog 运行上层计算应用，可以同时对不同的底层存储系统提供访问加速能力。同时 GooseFS Catalog 可以提供跨越存储服务的统一元数据查询能力，只需要一个 GooseFS 客户端开启 Catalog 功能，即可查询不同存储系统，例如 HDFS、COS、CHDFS 中的数据。

## 使用 GooseFS Table 管理能力

GooseFS Table 管理能力通过 goosefs table 指令集实现，提供了 DB 的绑定和解绑、查询 DB 信息、查询表信息、数据加载、数据淘汰等能力。GooseFS Table 管理指令集如下所示：

```
$ goosefs table
Usage: goosefs table [generic options]
[attachdb [-o|--option <key=value>] [--db <goosefs db name>] [--ignore-sync-errors] <udb type> <udb connection uri> <udb db name>]
[detachdb <db name>]
[free <dbName> <tableName> [-p|--partition <partitionSpec>]]
[load <dbName> <tableName> [-g|--greedy] [--replication <num>] [-p|--partition <partitionSpec>]]
[ls [<db name> [<table name>]]]
[stat <dbName> <tableName>]
[sync <db name>]
```

上述指令集中各项指令的能力简述如下：

- attachdb：挂载数据库，将一个远端数据库绑定到 GooseFS 上，目前仅支持 Hive MetaStore。
- detachdb：卸载数据库，将 GooseFS 上绑定的数据库解绑。
- free：清除指定 DB.Table 的数据缓存，可支持 Partition 粒度。
- load：缓存指定 DB.Table 的数据，可支持 partition 粒度，支持通过 replication 设置缓存的副本数。
- ls：列出指定 DB 或 DB.Table 的元数据信息。
- stat：查询指定 DB.Table 的文件数目、总大小、以及缓存百分比。
- sync：同步指定 DB 的内容。
- transform：将指定 DB 关联的 Table 转换为新的 Table。
- transformStatus：Table 转换的进度情况。

### 1. 挂载 DB

预热指定 Table 数据到 GooseFS 之前，需要将对应的 DB 挂载到 GooseFS 上。如下指令展示了将指定地址 metastore_host:port 中的数据库 goosefs_db_demo 挂载到 GooseFS 中，并将该 DB 在 GooseFS 中命名为 test_db：

```
$ goosefs table attachdb --db test_db hive thrift://metastore_host:port goosefs_db_demo
response of attachdb
```

> 注意：
> metastore_host:port 可以替换为任意合法可连接的 Hive MetaStore 地址。

### 2. 查看 Table 信息

绑定完数据库后，可以通过 ls 指令查看已挂载的 DB 和 Table 信息，如下指令展示了如何查询 test_db 中的 web_page 表信息：

```
$ goosefs table ls test_db web_page
OWNER: hadoop
DBNAME.TABLENAME: testdb.web_page (
wp_web_page_sk bigint,
wp_web_page_id string,
wp_rec_start_date string,
wp_rec_end_date string,
wp_creation_date_sk bigint,
wp_access_date_sk bigint,
wp_autogen_flag string,
wp_customer_sk bigint,
wp_url string,
wp_type string,
wp_char_count int,
wp_link_count int,
wp_image_count int,
wp_max_ad_count int,
)
PARTITIONED BY (
)
LOCATION (
gfs://metastore_host:port/myiNamespace/3000/web_page
)
PARTITION LIST (
{
partitionName: web_page
location: gfs://metastore_host:port/myNamespace/3000/web_page
}
)
```

### 3. 预热 Table 中的数据

预热 Table 的指令下发后会在后台发起一个异步作业，GooseFS 会在启动作业后返回一个作业 ID，可以通过 `job stat <id>` 指令查询任务的运行状态，同时可以通过 `table stat` 指令查看预热百分比。预热指令如下：

```
$ goosefs table load test_db web_page
Asynchronous job submitted successfully, jobId: 1615966078836
```

### 4. 查看 Table 预热情况

通过 `job stat` 指令可以查看预热 Table 作业的执行进度。当状态为 COMPLETED 时，整个预热过程完成，如果状态为 FAILED，可以在 master.log 文件中查看日志记录，排查预热错误的原因：

```
$ goosefs job stat 1615966078836
COMPLETED
```

当 Table 完成预热后，可以通过 stat 指令查看指定 Table 的概况。

```
$ goosefs table stat test_db web_page
detail
```

### 5. 释放 Table

通过以下指令可以从 GooseFS 中释放指定 Table 数据缓存：

```
$ goosefs table free test_db web_page
detail
```

### 6. 卸载 DB

通过以下指令可以从 GooseFS 中卸载指定 DB:

```
$ goosefs table detachdb test_db
detail
```

# GooseFS-FUSE 能力

最近更新时间：2022-08-01 11:59:56

GooseFS-FUSE 可以在一台 Unix 机器上的本地文件系统中挂载一个 GooseFS 分布式文件系统。通过使用该特性，一些标准的命令行工具（例如 ls、cat 以及 echo）可以直接访问 GooseFS 分布式文件系统中的数据。此外更重要的是使用不同语言实现的应用程序，例如 C、C++、Python、Ruby、Perl、Java 都可以通过标准的 POSIX 接口（例如 open、write、read）来读写 GooseFS，而不需要任何 GooseFS 的客户端整合与设置。

GooseFS-FUSE 是基于 FUSE 这个项目，并且都支持大多数的文件系统操作。但是由于 GooseFS 固有的属性，例如它的一次性、不可改变的文件数据模型，该挂载的文件系统与 POSIX 标准不完全一致，尚有一定的局限性。因此，请先阅读 局限性，从而了解该特性的作用以及局限。

## 安装要求

- JDK 1.8及以上
- Linux 系统：libfuse 2.9.3及以上（可以使用 2.8.3版本，但会提示一些警告）
- MAC 系统：osxfuse 3.7.1及以上

## 用法

### 挂载 GooseFS-FUSE

完成配置以及启动 GooseFS 集群后，在需要挂载 GooseFS 的节点上启动 Shell，并进入 $GOOSEFS_HOME 目录执行以下命令：

```
$ integration/fuse/bin/goosefs-fuse mount mount_point [GooseFS_path]
```

该命令会启动一个后台 Java 进程，用于将对应的 GooseFS 路径挂载到 `<mount_point>` 指定的路径。例如以下命令，将 GooseFS 路径 /people 挂载到本地文件系统的 /mnt/people 目录下。

```
$ integration/fuse/bin/goosefs-fuse mount /mnt/people /people
Starting goosefs-fuse on local host.
goosefs-fuse mounted at /mnt/people. See /lib/GooseFS/logs/fuse.log for logs
```

当 GooseFS_path 没有给定时，GooseFS-FUSE 会默认挂载到 GooseFS 根目录下(/)。您可以多次调用该命令，将 GooseFS 挂载到不同的本地目录下。所有的 GooseFS-FUSE 会共享 $GOOSEFS_HOME\logs\fuse.log 日志文件。该日志文件对于错误排查很有帮助。

注意：

`<mount_point>` 必须是本地文件系统中的一个空文件夹，并且启动 GooseFS-FUSE 进程的用户拥有该挂载点及对其的读写权限。

### 卸载 GooseFS-FUSE

卸载 GooseFS-FUSE 时，需在该节点上启动 Shell，并进入$GOOSEFS_HOME 目录执行以下命令：

```
$ integration/fuse/bin/goosefs-fuse umount mount_point
```

该命令将终止 goosefs-fuse java 后台进程，并卸载该文件系统。例如：

```
$ integration/fuse/bin/goosefs-fuse umount /mnt/people
Unmount fuse at /mnt/people (PID: 97626).
```

默认情况下，如果有任何读写操作未完成，unmount 操作会等待最多120s。如果120s后读写操作仍未完成，那么 Fuse 进程会被强行结束，这会导致正在读写的文件失败，您可以添加 -s 参数来避免 Fuse 进程被强行结束。例如：

```
$ ${GOOSEFS_HOME}/integration/fuse/bin/goosefs-fuse unmount -s /mnt/people
```

### 检查 GooseFS-FUSE 是否在运行

罗列所有的挂载点，需在该节点上启动 Shell，并进入 $GOOSEFS_HOME 目录执行以下命令：

```
$ integration/fuse/bin/goosefs-fuse stat
```

该命令会输出包括 pid、mount_point、GooseFS_path 在内的信息。

例如输出可以是以下格式：

```
$ pid mount_point GooseFS_path
80846 /mnt/people /people
80847 /mnt/sales /sales
```

### Goosefs-FUSE 目录结构


conf 目录下：

- masters：master 服务器的 IP 配置文件
- workers：worker 服务器的 IP 配置文件
- goosefs-site.properties：goosefs 配置文件
- libexec：goosefs-fuse 运行依赖的 lib 库文件
- goosefs-fuse-1.3.0：goosefs-fuse 后台运行的 jar 包
- log：日志目录

## 可选配置

GooseFS-FUSE 基于标准的 GooseFS-core-client-fs 进行操作。如果您希望它像使用其他应用的 client 一样，自定义该 GooseFS-core-client-fs 的行为。

可通过编辑 $GOOSEFS_HOME/conf/goosefs-site.properties 配置文件来更改客户端选项。

> 注意：
> 所有的更改应该在 GooseFS-FUSE 启动之前完成。

## 局限性

目前，GooseFS-FUSE 支持大多数基本文件系统的操作。然而，由于 GooseFS 某些内在的特性，您需要注意以下几点：

- 不支持对文件进行随机写入和追加写入操作。
- 文件只能顺序地写入一次，并且无法修改。这意味着如果要修改一个文件，您需要先删除该文件，或者 open 操作时，携带 O_TRUNC 标志符，将文件长度置为0。
- 不支持读取挂载点中正在写入的文件。
- 不支持对文件长度进行 truncate 操作。
- 不支持 soft/hard link；GooseFS 没有 hard-link 和 soft-link 的概念，所以不支持与之相关的命令，例如 ln。此外关于 hard-link 的信息也不在 ll 的输出中显示。
- 以对象存储作为底层存储时，Rename 操作非原子。
- 只有当 GooseFS 的 GooseFS.security.group.mapping.class 选项设置为ShellBasedUnixGroupsMapping 的值时，文件的用户与分组信息才与 Unix 系统的用户分组对应。否则 chown 与 chgrp 的操作不生效，而 ll 返回的用户与分组为启动 GooseFS-FUSE 进程的用户与分组信息。

## 性能考虑

由于 FUSE 和 JNR 的配合使用，与直接使用原生文件系统 API 相比，使用挂载文件系统的性能会相对较差。

大多数性能问题的原因在于，每次进行 read 或 write 操作时，内存中都存在若干个副本，并且 FUSE 将写操作的最大粒度设置为128KB。其性能可以利用 kernel 3.15 引入的 FUSE 回写（write-backs）缓存策略从而得到大幅提高（但 libfuse 2.x 用户空间库目前尚不支持该特性）。

## GooseFS-FUSE 配置参数

以下是 GooseFS-FUSE 相关的配置参数：

| 参数 | 默认值 | 描述 |
| --- | --- | --- |
| goosefs.fuse.cached.paths.max | 500 | 定义内部 GooseFS-FUSE 缓存的大小，该缓存维护本地文件系统路径和 Alluxio 文件 URI 之间最常用的转换。 |
| goosefs.fuse.debug.enabled | false | 允许 FUSE 调试输出，该输出会被重定向到 `goosefs.logs.dir` 指定目录中的 `fuse.out` 日志文件。 |
| goosefs.fuse.fs.name | goosefs-fuse | FUSE 挂载文件系统使用的描述性名称。 |

| 参数 | 默认值 | 描述 |
| --- | --- | --- |
| goosefs.fuse.jnifuse.enabled | true | 使用 JNI-Fuse 库以获得更好的性能。 如果禁用，将使用 JNR-Fuse。 |
| goosefs.fuse.shared.caching.reader.enabled | false | （实验性）使用共享 grpc 数据读取器，通过 GooseFS JNI Fuse 在多进程文件读取中获得更好的性能。块数据将缓存在客户端，因此 Fuse 进程需要更多内存。 |
| goosefs.fuse.logging.threshold | 10s | 当花费的时间超过阈值时，记录 FUSE API 调用。 |
| goosefs.fuse.maxwrite.bytes | 131072 | FUSE 写操作的粒度（bytes），注意目前 128KB 是 Linux 内核限制的上界。 |
| goosefs.fuse.user.group.translation.enabled | false | 是否在 FUSE API 中将 GooseFS 的用户与组转化为对应的 Unix 用户与组。当设为 false 时，所有 FUSE 文件的用户与组将会显示为挂载 goosefs-fuse 线程的用户与组。 |

# 常见问题

### 缺少 libfuse 库文件

在您执行 GooseFS-Fuse 挂载之前，需要安装 libfuse。

```
2021-10-13 20:04:42,970 INFO  TieredIdentityFactory - Initialized tiered identity TieredIdentity(node=10.91.27.82, rack=null)
2021-10-13 20:04:43,560 ERROR GooseFSFuse - launch fuse failed:
java.io.IOException: Failed to mount GooseFS file system
        at com.qcloud.cos.goosefs.fuse.GooseFSFuse.launchFuse(GooseFSFuse.java:192)
        at com.qcloud.cos.goosefs.fuse.GooseFSFuse.main(GooseFSFuse.java:126)
Caused by: java.lang.UnsatisfiedLinkError: /private/var/folders/5_/sp1cwpdd1xn9w96x5xkw7qnc0000gn/T/libjnifuse957879065968834264.jnilib: dlopen(/private/var/fo
5_/sp1cwpdd1xn9w96x5xkw7qnc0000gn/T/libjnifuse957879065968834264.jnilib, 1): Library not loaded: /usr/local/lib/libfuse.2.dylib
  Referenced from: /private/var/folders/5_/sp1cwpdd1xn9w96x5xkw7qnc0000gn/T/libjnifuse957879065968834264.jnilib
  Reason: image not found
        at java.lang.ClassLoader$NativeLibrary.load(Native Method)
        at java.lang.ClassLoader.loadLibrary0(ClassLoader.java:1934)
        at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1817)
        at java.lang.Runtime.load0(Runtime.java:810)
        at java.lang.System.load(System.java:1086)
        at com.qcloud.cos.goosefs.jnifuse.utils.NativeLibraryLoader.loadLibraryFromJar(NativeLibraryLoader.java:105)
        at com.qcloud.cos.goosefs.jnifuse.utils.NativeLibraryLoader.loadLibrary(NativeLibraryLoader.java:83)
        at com.qcloud.cos.goosefs.jnifuse.LibFuse.loadLibrary(LibFuse.java:48)
        at com.qcloud.cos.goosefs.jnifuse.LibFuse.<clinit>(LibFuse.java:32)
        at com.qcloud.cos.goosefs.jnifuse.AbstractFuseFileSystem.<clinit>(AbstractFuseFileSystem.java:41)
        at com.qcloud.cos.goosefs.fuse.GooseFSFuse.launchFuse(GooseFSFuse.java:154)
        ... 1 more
```

- **方式一**

  安装命令：

  ```
  yum install fuse-devel
  ```

查看是否安装成功：

```
find / -name libfuse.so*
```

- **方式二**

  更新旧版本 libfuse.so.2.9.2，安装步骤如下：

  > 说明：
  > 在 CentOS 7安装 libfuse，CentOS 7默认安装的是 libfuse.so.2.9.2。

  首先，下载 libfuse 源码，并编译生成 libfuse.so.2.9.7。

  ```
  tar -zxvf fuse-2.9.7.tar.gz
  cd fuse-2.9.7/ && ./configure && make && make install
  echo -e '\n/usr/local/lib' >> /etc/ld.so.conf
  ldconfig
  ```

其次，下载、编译及生成 libfuse.so.2.9.7 后，然后按照以下步骤进行安装替换。

1. 执行以下命令，查找旧版本 libfuse.so.2.9.2 库链接。

   ```
   find / -name libfuse.so*
   ```

2. 执行以下命令，将 libfuse.so.2.9.7拷贝至旧版本库 libfuse.so.2.9.2 所在位置。

   ```
   cp /usr/local/lib/libfuse.so.2.9.7 /usr/lib64/
   ```

3. 执行以下命令， 删除旧版本 libfuse.so 库的所有链接。

   ```
   rm -f /usr/lib64/libfuse.so
   rm -f /usr/lib64/libfuse.so.2
   ```

4. 执行以下命令，建立与被删除旧版本链接类似的 libfuse.so.2.9.7 库链接。

   ```
   ln -s /usr/lib64/libfuse.so.2.9.7 /usr/lib64/libfuse.so
   ln -s /usr/lib64/libfuse.so.2.9.7 /usr/lib64/libfuse.so.2
   ```

**VIM 编辑挂载点中的文件报错 Write error in swap file？**

您可以通过更改 VIM 的配置，使用 VIM 7.4 及之前的版本，对 GooseFS-Fuse 挂载点中的文件进行编辑。VIM 的 swap 文件用于暂存用户对文件的修改内容，方便 VIM 在机器宕掉重启后，可以根据 swap 文件，对未保存的修改内容进行恢复。上面的报错是由于 VIM 对 swap 文件，涉及随机写入操作，而 GooseFS 不支持对文件的随机写入。解决方法：您可以通过在 VIM 编辑窗口中，执行如下的 VIM 命令关闭 swap 文件生成，`:set noswapfile`，或者在配置文件 ~/.vimrc 中添加配置行 `set noswapfile`。

# 部署指南
# 使用自建集群部署

最近更新时间：2022-08-01 11:55:12

本文主要介绍如何规范化地在单机、集群以及腾讯云 EMR 集群（暂未集成支持 GooseFS 的版本）中部署、配置以及运行 GooseFS。

## 部署环境要求

### 硬件环境

目前 GooseFS 支持主流 x86/x64 架构的 Linux/MacOS 运行环境（**注意：MacOS M1处理器未验证支持**）。详细的运行节点配置如下：

#### Master 节点

- **CPU**：工作主频1GHz以上，考虑到 Master 需要处理大量，建议生产环境使用多核架构处理器。
- **物理内存**：1GB以上，生产环境建议使用8GB以上的物理内存配置。
- **磁盘**：20GB以上，建议生产环境配备高性能的 NVME SSD 盘作为元数据缓存盘（RocksDB 模式）。

#### Worker 节点

- **CPU**：工作主频1GHz以上，考虑到 Worker 节点也需要处理大量的并发请求，同样建议生产环境使用多核架构处理器。
- **物理内存**：2GB以上，生产环境建议根据性能需求选用合适的物理内存配置。例如，生产环境中需要将大量的数据块缓存到 Worker 节点的内存中或是采用 MEM + SSD/HDD 的混合存储时，需要根据实际可能会缓存到内存中的数据量来配备物理内存。无论使用何种缓存模式，生产环境都建议 Worker 节点配备16GB及以上的物理内存。
- **磁盘**：20GB以上 SSD/HDD 盘，建议根据实际生产环境中所需要预热缓存的数据量大小来估计每个 Worker 节点所需配置的磁盘空间。另外，为了更好的性能体验，建议配备采用 NVME 接口的 SSD 盘作为 Worker 节点的数据盘。

### 软件环境

- Red Hat Enterprise Linux 5.5+、Ubuntu Linux 12.04 LTS+（不使用批量部署时可以支持）、CentOS 7.4+以及 TLinux 2.x（Tencent Linux 2.x），基于 Intel 架构 MacOS 10.8.3 以上版本。腾讯云 CVM 用户推荐使用 CentOS 7.4、Tencent（TLinux 2.x）或 TencentOS Server 2.4 版本的操作系统。
- OpenJDK 1.8/Oracle JDK 1.8，需要注意的是，未验证支持 JDK 1.9 及以上版本。
- 支持 SSH 工具集（包括 SSH 和 SCP 工具）、Expect Shell（使用批量部署时需要）工具以及 Yum 包管理工具。

### 集群网络环境

- Master/Worker 节点之间外网或内网互通。使用批量部署时，要求 Master 能够正常访问外网软件源，或正确配置包管理系统的内网软件源。
- 集群能够通过外网或内网访问到 COS 存储桶或者 CHDFS 文件系统。

### 安全组与用户权限要求

一般情况，GooseFS 建议用户使用专用的 Linux 账户部署运行 GooseFS， 例如在自建集群和 EMR 环境中，可以统一使用 hadoop 用户来部署运行 GooseFS。在批量部署中，则需要补充以下用户能力和权限：

- 具备切换到 root 或使用 sudo 权限的能力；
- 部署运行账户具备读写安装目录的权限；
- Master 节点具备 SSH 登录到集群中所有 Worker 节点的权限；
- 集群中对应的节点角色需要放开对应的端口：Master（9200和9201）、Worker（9203和9204）、Job Master(9205、9206和9207)、Job Worker（9208、9209和9210）、Proxy（9211）以及 LogServer（9212）。

## 单机模式部署运行（伪分布式架构）

伪分布式架构部署主要适用于体验和调试 GooseFS，初级用户可以在自己的 Linux 或 Mac 系统主机上快速体验和调试 GooseFS。

### 部署

1. 首先 下载 GooseFS 的二进制分发包。

2. 分发包下载后解压，进入到 GooseFS 的目录中，执行如下操作：

   - 通过拷贝 conf/goosefs-site.properties.template 创建 /etc/goosefs/conf/goosefs-site.properties 配置文件：

     ```
     $ cp conf/goosefs-site.properties.template /etc/goosefs/conf/goosefs-site.properties
     ```

   - 在 /etc/goosefs/conf/goosefs-site.properties 配置文件中设置 `goosefs.master.hostname` 为 `localhost` 。

   - 在 /etc/goosefs/conf/goosefs-site.properties 配置文件中设置 `goosefs.master.mount.table.root.ufs` 为本地文件系统中的目录，例如：/tmp 或是 /data/goosefs 等。

建议配置 localhost 的 SSH 免密登录，否则格式化和启动等操作，需要输入 localhost 的登录密码。

### 运行

执行如下命令，就可以完成挂载一个 RamFS 文件系统：

```
$ ./bin/goosefs-mount.sh SudoMount
```

同样，也可以在启动 GooseFS 集群时挂载：

```
$ ./bin/goosefs-start.sh local SudoMount
```

当启动后，使用 `jps` 命令，能够看到伪分布式模式下，GooseFS 的所有进程：

```
$ jps
35990 Jps
35832 GooseFSSecondaryMaster
35736 GooseFSMaster
35881 GooseFSWorker
35834 GooseFSJobMaster
35883 GooseFSJobWorker
35885 GooseFSProxy
```

然后就可以使用 `goosefs` 命令行，执行 namespace、fileSystem、job 以及 table 的各项操作了。例如，上传一个本地文件到 GooseFS 中，并且列出当前 GooseFS 中根目录下的文件和目录：

```
$ goosefs fs copyFromLocal test.txt /
Copied file:///Users/goosefs/test.txt to /
$ goosefs fs ls /
-rw-r--r-- goosefs staff 0 PERSISTED 04-28-2021 04:00:35:156 100% /test.txt
```

GooseFS 的命令行为用户提供了管理和访问 GooseFS 所有命令行接口，包括命名空间（namespace）、表（table）、作业（job）以及常用的文件系统（fs）操作等，具体可参考我们的官方文档或使用 `goosefs -h` 命令行参数获得详细帮助信息。

## 集群模式部署运行

集群部署运行主要针对用户自建 IDC 集群的生产环境或尚未集成 GooseFS 的腾讯云 EMR 生产环境，这里有具体分为 Standalone 架构部署与高可用（HA）架构部署。

GooseFS 在 `scripts` 目录下提供了批量配置 SSH 免密登录以及批量安装部署 GooseFS 的脚本工具，可供用户方便快捷地完成 GooseFS 大规模集群的安装部署。用户可自检前一章节的部署环境要求中的批量部署条件，灵活地选择是否可以执行批量部署。

### 批量部署工具介绍

GooseFS 在 `scripts` 目录下提供了批量配置 SSH 免密登录以及批量部署安装 GooseFS 的工具化脚本，用户在满足执行条件的前提下，可按照如下步骤完成批量部署任务：

- 首先完成配置 `conf/masters` 和 `conf/workers` 中的主机名或 IP 地址。同时，也需要完成配置最终生成环境的所有配置。
- 然后进入到 `scripts` 目录中，认真填写 `install.properties` 这一配置文件。再切换到 `root` 账户或使用 `sudo` 身份执行 `config_ssh.sh` 完成整个集群的免密登录配置。
- 免密配置完成后，可以执行 `validate_env.sh` 工具，校验整个集群配置状态。
- 最后仍然以 `root` 账户或 `sudo` 身份执行 `install.sh` 脚本，启动安装部署，并等待整个流程完成。

在整个部署流程都成功完成以后，可以执行 `./bin/goosefs-start.sh all SudoMount` 来启动运行整个集群。默认配置下，所有的运行日志都会记录到 `${GOOSEFS_HOME}/logs` 下。

### Standalone 架构部署

Standalone 架构采用的是单 Master 节点，多 Worker 节点的集群部署架构。具体可参考如下步骤部署运行：

1. 下载 GooseFS 的二进制分发包。
2. 使用 `tar zxvf goosefs-x.x.x-bin.tar.gz` 命令解压到安装路径后。可参见批量部署工具的介绍配置和执行集群的批量部署，也可以继续参考下文详细地手动部署流程。

（1）从 `conf` 目录下拷贝 `template` 文件创建配置文件：

```
$ cp conf/goosefs-site.properties.template /etc/goosefs/conf/goosefs-site.propert
ies
```

（2）在 `goosefs-site.properties` 配置文件中指定如下配置：

```
goosefs.master.hostname=<MASTER_HOSTNAME>
goosefs.master.mount.table.root.ufs=<STORAGE_URI>
```

其中，`goosefs.master.hostname` 设置为单 master 节点的 hostname 或 ip。`goosefs.master.mount.table.root.ufs` 则指定 GooseFS 根目录所挂载的底层文件系统（UFS）路径 URI。注意：这个 URI 必须保证 Master 和 Worker 节点都能访问到。

例如可以挂载一个 COS 路径为 GooseFS 根路径：goosefs.master.mount.table.root.ufs=cosn://bucket-1250000000/goosefs/。

在 `masters` 配置文件中指定单 Master 节点的 hostname 或 ip，例如：

```
# The multi-master Zookeeper HA mode requires that all the masters can access
# the same journal through a shared medium (e.g. HDFS or NFS).
# localhost
cvm1.compute-1.myqcloud.com
```

在 `workers` 配置文件中指定所有 Worker 节点的 host 或 ip，例如：

```
# An GooseFS Worker will be started on each of the machines listed below.
# localhost
cvm2.compute-2.myqcloud.com
cvm3.compute-3.myqcloud.com
```

待所有配置都设置完毕后，执行 `./bin/goosefs copyDir /etc/goosefs/conf/` 就可以将配置同步到所有节点。

最后执行 `./bin/goosefs-start.sh all` 就可以启动 GooseFS 集群。

### 高可用架构部署

单 Master 节点的 Standalone 架构容易出现单点问题，因此，实际生产环境建议部署多 Master 节点来获得高可用的系统架构。多个 Master 节点中只有一个会作为主（Leader）节点对外提供服务，其他的备（Standby）节点通过同步共享日志（Journal）来保持与主节点相同的文件系统状态。当主节点故障宕机后，会从当前的备节点中自动选择一个接替主节点继续对外提供服务，这样就消除了系统的单点故障，实现了整体高可用架构。

目前 GooseFS 支持基于 Raft 日志和 Zookeeper 两种方式来实现主备节点状态的强一致性。下面将会针对这两种部署方式分别进行介绍。

#### 基于 Raft 嵌入式日志的高可用部署配置

首先依据配置模板创建配置文件：

```
$ cp conf/goosefs-site.properties.template /etc/goosefs/conf/goosefs-site.propert
ies
```

```
goosefs.master.hostname=<MASTER_HOSTNAME>
goosefs.master.mount.table.root.ufs=<STORAGE_URI>
goosefs.master.embedded.journal.address=<EMBBEDDED_JOURNAL_ADDRESS>
```

上述配置选项中，说明如下：

- `goosefs.master.hostname` 配置每个 Master 对外服务的 ip 或 hostname，并且确保客户端和 Worker 节点都能够访问。
- `goosefs.master.mount.table.root.ufs` 则设置为挂载到 GooseFS 根目录的底层存储 URI。

- `goosefs.master.embedded.journal.address` 则设置为所有备节点的 `ip:embedded_journal_port` 或 `host:embedded_journal_port` 。其中，embedded_journal_port 默认为9202。

基于 Raft 嵌入式日志的部署方案依赖于 copycat 的 Leader 选举机制。因此，Raft 的 HA 部署架构不可与 Zookeeper 相互混用。

在完成所有配置以后，同样使用如下命令同步所有配置：

```
$ ./bin/goosefs copyDir /etc/goosefs/conf/
```

最后，完成格式化以后，即可启动 GooseFS 集群：

```
$ ./bin/goosefs format
```

```
$ ./bin/goosefs-start.sh all
```

使用如下命令可查看当前的主（Leader）节点：

```
$ ./bin/goosefs fs leader
```

也可以使用如下命令查看集群状态：

```
$ ./bin/goosefs fsadmin report
```

**基于 Zookeeper 的共享日志部署配置**

配置 Zookeeper 服务搭建 GooseFS 的高可用架构需要满足如下条件：

- Zookeeper 集群，GooseFS Master 使用 Zookeeper 进行 Leader 选取，GooseFS 的客户端和 Worker 节点则通过 Zookeeper 查询主 Master 节点；
- 高可用强一致的共享存储系统，并且保证所有 GooseFS 的 Master 节点均可以访问到。主 Master 节点会将日志写入到该存储系统上，备（Standby）节点则会不断地从共享存储系统上读取日志，并且重放来保持与主节点的状态一致性。一般情况，该共享存储系统，推荐设置为 HDFS 或 COS。例如，`hdfs://10.0.0.1:9000/goosefs/journal` 或 `cosn://bucket-1250000000/journal` 。

配置如下：

```
goosefs.zookeeper.enabled=true
goosefs.zookeeper.address=<ZOOKEEPER_ADDRESSS>
goosefs.master.journal.type=UFS
goosefs.master.journal.folder=<JOURNAL_URI>
```

然后使用 `./bin/goosefs copyDir /etc/goosefs/conf/` 将配置同步分发到集群中的所有节点。最后启动集群 `./bin/goosefs-start.sh all` 即可。

## GooseFS 的进程列表

当执行 goosefs-start.sh ALL 脚本并启动 GooseFS 后，集群将包含如下进程：

| 进程 | 描述 |
| --- | --- |
| GooseFSMaster | 默认 RPC 端口为9200，Web 端口为9201 |
| GooseFSWorker | 默认 RPC 端口为9203，Web 端口为9204 |
| GooseFSJobMaster | 默认 RPC 端口为9205，Web 端口为9206 |
| GooseFSProxy | 默认 Web 端口为9211 |
| GooseFSJobWorker | 默认 RPC 端口为9208，Web 端口为9210 |

# 使用腾讯云 EMR 部署

最近更新时间：2021-07-16 11:46:22

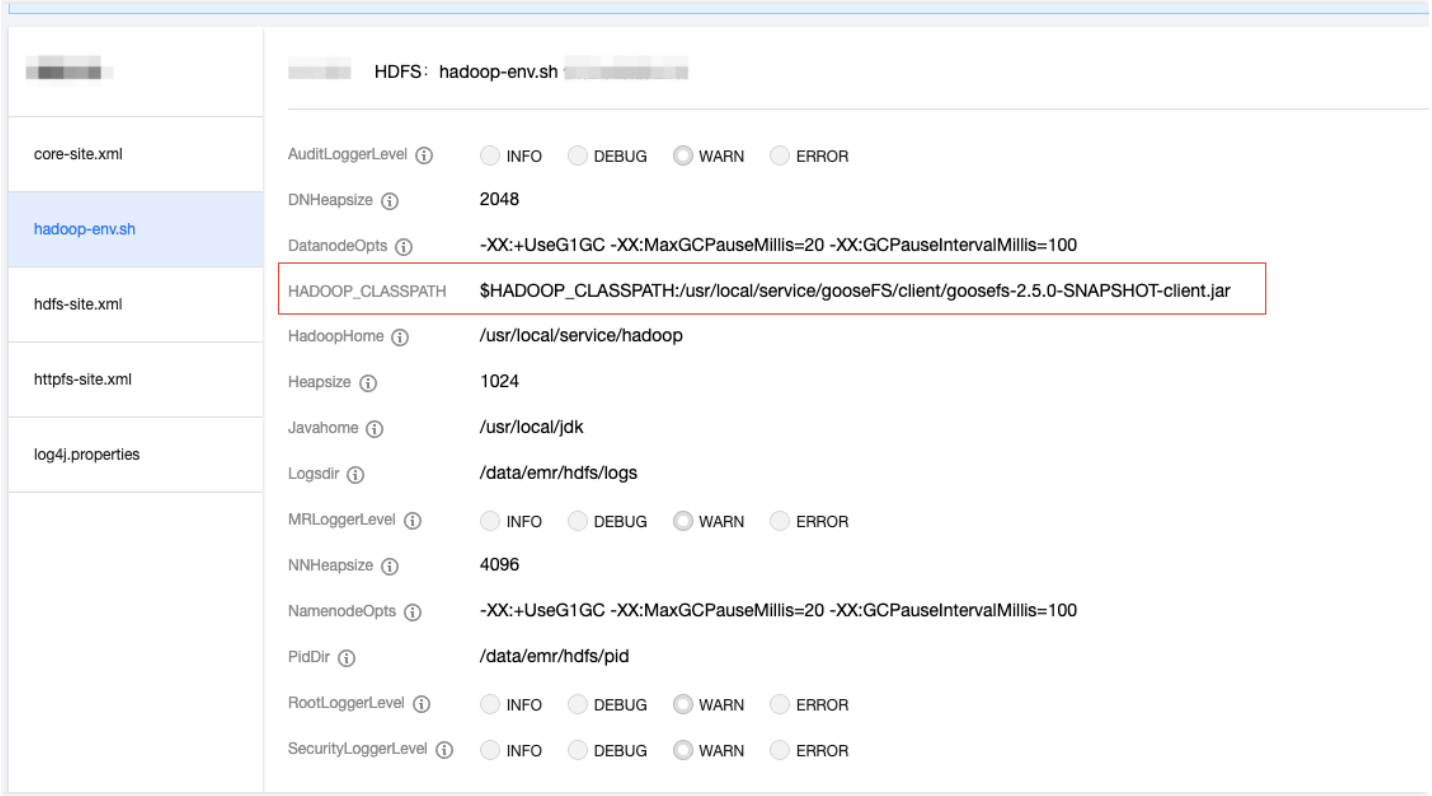
目前 GooseFS 已经集成到了腾讯云 EMR 环境中，将会在最新的 EMR 版本中发布。届时，用户无需针对腾讯云 EMR 环境单独部署，可以像使用其他 EMR 组件一样直接使用 GooseFS。

下文将针对未集成 GooseFS 的腾讯云 EMR 存量集群，介绍如何部署配置 GooseFS 的 EMR 环境。

首先，参照 集群模式部署运行 章节的内容，选择生产环境合适的部署架构，完成集群部署。
其次，针对 EMR 支持组件进行配置，本文以 Hadoop MapReduce、Spark 以及 Flink 对 GooseFS 的支持来讲解。

## Hadoop MapReduce 支持

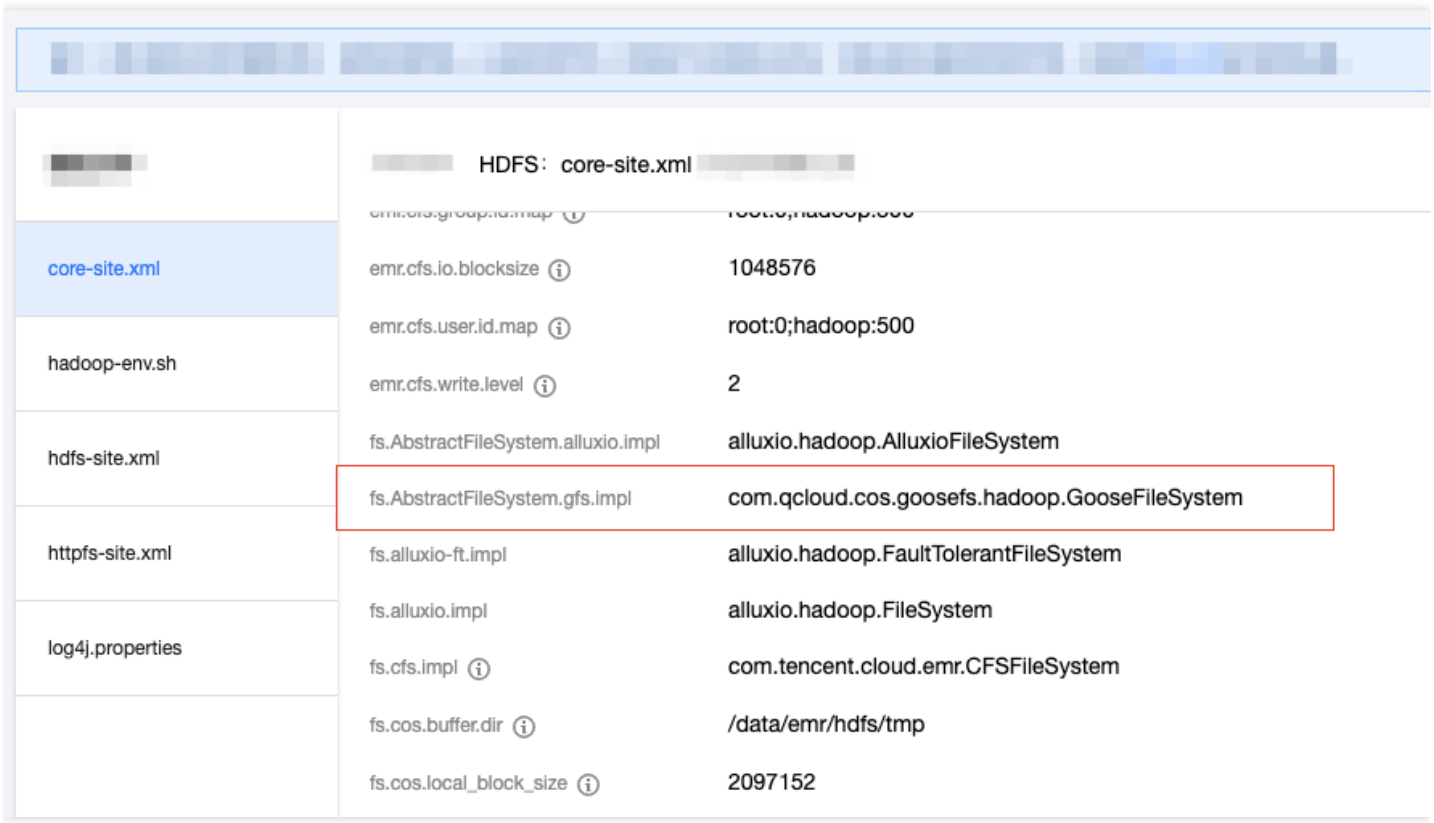
为了使得 Hadoop 的 MapReduce 作业能够读写 GooseFS 中的数据，需要在 hadoop-env.sh 中将 GooseFS Client 的依赖路径添加到 HADOOP_CLASSPATH，这个操作可以在 EMR 的控制台上完成，如下所示：




同时，还需要配置在 core-site.xml 中配置 GooseFS 的 HCFS 实现，同样这个操作也可以在 EMR 的控制台上完成：

配置 fs.AbstractFileSystem.gfs.impl 为如下：

```
com.qcloud.cos.goosefs.hadoop.GooseFileSystem
```




配置 fs.gfs.impl 为如下：

```
com.qcloud.cos.goosefs.hadoop.FileSystem
```

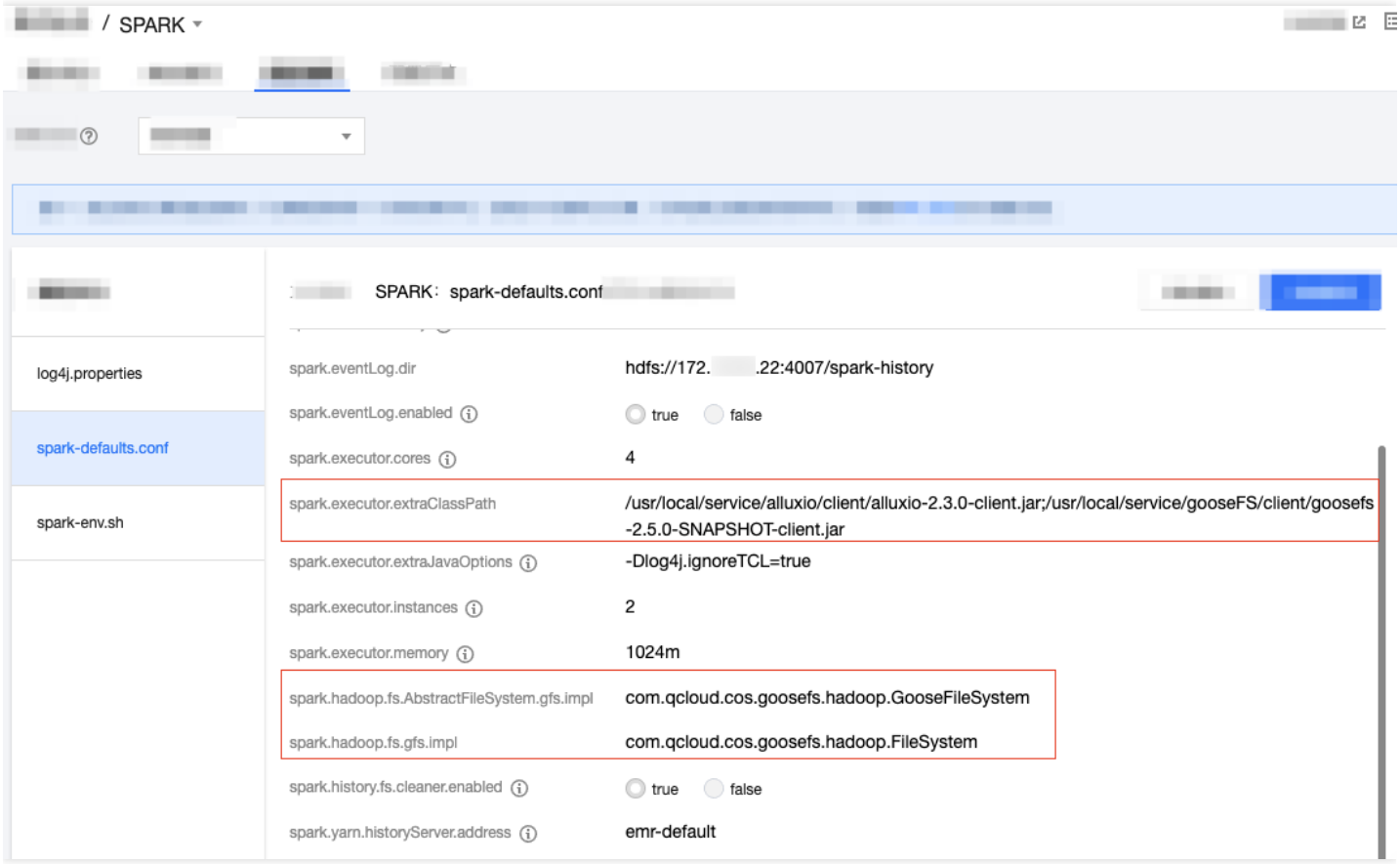
下发配置后，重启 YARN 相关组件即可生效。

# Spark 支持

为了使得 Spark 能够访问goosefs，同样需要配置 GooseFS 的 client 依赖包到 spark 的 executor classpath 中，同时在 spark-defaults.conf 中指定：

```
...
spark.driver.extraClassPath ${GOOSEFS_HOME}/client/goosefs-x.x.x-client.jar
spark.executor.extraClassPath ${GOOSEFS_HOME}/client/goosefs-x.x.x-client.jar
spark.hadoop.fs.gfs.impl com.qcloud.cos.goosefs.hadoop.FileSystem
spark.hadoop.fs.AbstractFileSystem.gfs.impl com.qcloud.cos.goosefs.hadoop.GooseFi
leSystem
...
```

同样，该操作也可以在 EMR 控制台上 Spark 组件中配置和下发：




# Flink 支持

腾讯云 EMR 的 Flink 采用的是 Flink on YARN 的部署模式，因此原则上只要确保 ${FLINK_HOME}/flink-conf.yaml 中正确设置 `fs.hdfs.hadoopconf` 到 hadoop 的配置路径下即可，腾讯云 EMR 集群中一般为 `/usr/local/service/hadoop/etc/hadoop` 。

无需设置其他配置项，直接使用 Flink on YARN 的方式提交 Flink 作业即可，作业中需要访问 GooseFS 的路径为 `gfs://master:port/<path>` 。

> 注意：
> Flink 访问 GooseFS 时，必须指定 master 和 port。

## Hive、Impala、HBase、Sqoop 以及 Oozie 支持

当配置 Hadoop MapReduce 的环境支持以后，Hive、Impala、HBase 等组件无需单独配置支持，即可正常使用。

# 使用腾讯云 TKE 部署

最近更新时间：2021-11-16 11:48:10

使用 TKE 部署 GooseFS 需要借助 开源组件 Fluid 部署，TKE 应用市场已上架。部署包括两步：

1. 通过 Fluid helm chart 部署 controller。
2. 通过 kubectl 创建 Dataset 和 GooseFS runtime。

## 准备事项

1. 拥有腾讯云 TKE 集群。
2. 已安装 kubectl，版本 v1.18及以上。

## 安装步骤

1. 在 TKE 应用市场 找到 fluid 应用。

2. 安装 Fluid Controller。

3. 检查 controller 组件。在左侧【集群】中找到对应集群，如果看到了两个 controller，则说明 fluid 组件安装成功。

## 操作演示

### 1. 集群访问授权

```
[root@master01 run]# export KUBECONFIG=xxx/cls-xxx-config （从tke控制台页面，下载集群凭证到某个目录）
```

> 注意：
> 集群 API Server 需要开启外网访问权限。

### 2. 创建 UFS 数据集 Dataset（COS 为例）

先创建 secret.yaml 用于加密， 模版如下：

```
apiVersion: v1
kind: Secret
metadata:
name: mysecret
stringData:
fs.cosn.userinfo.secretKey: xxx
fs.cosn.userinfo.secretId:xxx
```

创建secret：

```
[root@master01 ~]# kubectl apply -f secret.yaml
secret/mysecret created
```

dataset.yaml 模版如下：

```
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: slice1
spec:
mounts:
- mountPoint: cosn://{your bucket}
name: slice1
options:
fs.cosn.bucket.region: ap-beijing
fs.cosn.impl: org.apache.hadoop.fs.CosFileSystem
fs.AbstractFileSystem.cosn.impl: org.apache.hadoop.fs.CosN
fs.cosn.userinfo.appid: "${your appid}"
encryptOptions:
- name: fs.cosn.userinfo.secretKey
valueFrom:
secretKeyRef:
name: mysecret
key: fs.cosn.userinfo.secretKey
- name: fs.cosn.userinfo.secretId
valueFrom:
secretKeyRef:
name: mysecret
key: fs.cosn.userinfo.secretId
```

创建 dataset

```
[root@master01 run]# kubectl apply -f dataset.yaml
dataset.data.fluid.io/slice1 created
```

查询 Dataset 状态，处于 NotBond 状态：

```
[root@master01 run]# kubectl get dataset
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
slice1
NotBound 11s
```

### 3. 创建 runtime

runtime.yaml 模板如下：

```
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: slice1
spec:
replicas: 1
data:
replicas: 1
goosefsVersion:
imagePullPolicy: Always
image: ${img_uri}
imageTag: ${tag}
tieredstore:
levels:
- mediumtype: MEM
path: /dev/shm
quota: 1Gi
high: "0.95"
low: "0.7"
properties:
# goosefs user
goosefs.user.file.writetype.default: MUST_CACHE
master:
replicas: 1
journal:
volumeType: hostpath
jvmOptions:
- "-Xmx40G"
- "-XX:+UnlockExperimentalVMOptions"
- "-XX:ActiveProcessorCount=8"
worker:
jvmOptions:
- "-Xmx12G"
- "-XX:+UnlockExperimentalVMOptions"
- "-XX:MaxDirectMemorySize=32g"
```

```
- "-XX:ActiveProcessorCount=8"
resources:
limits:
cpu: 8
fuse:
imagePullPolicy: Always
image: ${fuse_uri}
imageTag: ${tag_num}
env:
MAX_IDLE_THREADS: "32"
jvmOptions:
- "-Xmx16G"
- "-Xms16G"
- "-XX:+UseG1GC"
- "-XX:MaxDirectMemorySize=32g"
- "-XX:+UnlockExperimentalVMOptions"
- "-XX:ActiveProcessorCount=24"
resources:
limits:
cpu: 16
args:
- fuse
- --fuse-opts=kernel_cache,ro,max_read=131072,attr_timeout=7200,entry_timeout=720
0,nonempty
```

创建 runtime：

```
[root@master01 run]# kubectl apply -f runtime.yaml
goosefsruntime.data.fluid.io/slice1 created
```

检查 goosefs 组件状态：

```
[root@master01 run]# kubectl get pods
NAME READY STATUS RESTARTS AGE
slice1-fuse-xsvwj 1/1 Running 0 37s
slice1-master-0 2/2 Running 0 118s
slice1-worker-fzpdw 2/2 Running 0 37s
```

### 4. 预热数据

dataload.yaml 预热组件如下：

```
apiVersion: data.fluid.io/v1alpha1
kind: DataLoad
metadata:
name: slice1-dataload
```

```
spec:
dataset:
name: slice1
namespace: default
```

此时 Dataset 处于 Bond 状态，Cached 比例是0%。

```
[root@master01 run]# kubectl get dataset
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
slice1 97.67MiB 0.00B 4.00GiB 0.0% Bound 31m
```

执行数据预热：

```
[root@master01 run]# kubectl apply -f dataload.yaml
dataload.data.fluid.io/slice1-dataload created
```

查看数据预热进度：

```
[root@master01 run]# kubectl get dataset --watch
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
slice1 97.67MiB 52.86MiB 4.00GiB 54.1% Bound 39m
slice1 97.67MiB 53.36MiB 4.00GiB 54.6% Bound 39m
slice1 97.67MiB 53.36MiB 4.00GiB 54.6% Bound 39m
slice1 97.67MiB 53.87MiB 4.00GiB 55.2% Bound 39m
slice1 97.67MiB 53.87MiB 4.00GiB 55.2% Bound 39m
```

数据预热完成100%：

```
[root@master01 run]# kubectl get dataset --watch
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
slice1 97.67MiB 97.67MiB 4.00GiB 100.0% Bound 44m
```

### 5. 检查数据

```
[root@master01 run]# kubectl get pods
NAME READY STATUS RESTARTS AGE
slice1-dataload-loader-job-km6mg 0/1 Completed 0 12m
slice1-fuse-xsvwj 1/1 Running 0 17m
slice1-master-0 2/2 Running 0 19m
slice1-worker-fzpdw 2/2 Running 0 17m
```

进入 goosefs master 容器：

```
[root@master01 run]# kubectl exec -it slice1-master-0 -- /bin/bash
Defaulting container name to goosefs-master.
```

列出 goosefs 目录：

```
[root@VM-2-40-tlinux goosefs-1.0.0-SNAPSHOT-noUI-noHelm]# goosefs fs ls /slice1
10240 PERSISTED 06-25-2021 16:45:11:809 100% /slice1/p1
1 PERSISTED 05-24-2021 16:07:37:000 DIR /slice1/a
10000 PERSISTED 05-26-2021 19:29:05:000 DIR /slice1/p2
```

查看某个具体文件：

```
[root@VM-2-40-tlinux goosefs-1.0.0-SNAPSHOT-noUI-noHelm]# goosefs fs ls /slice1/
a/
12 PERSISTED 06-25-2021 16:45:11:809 100% /slice1/a/1.xt
```

# 使用腾讯云 EKS 部署

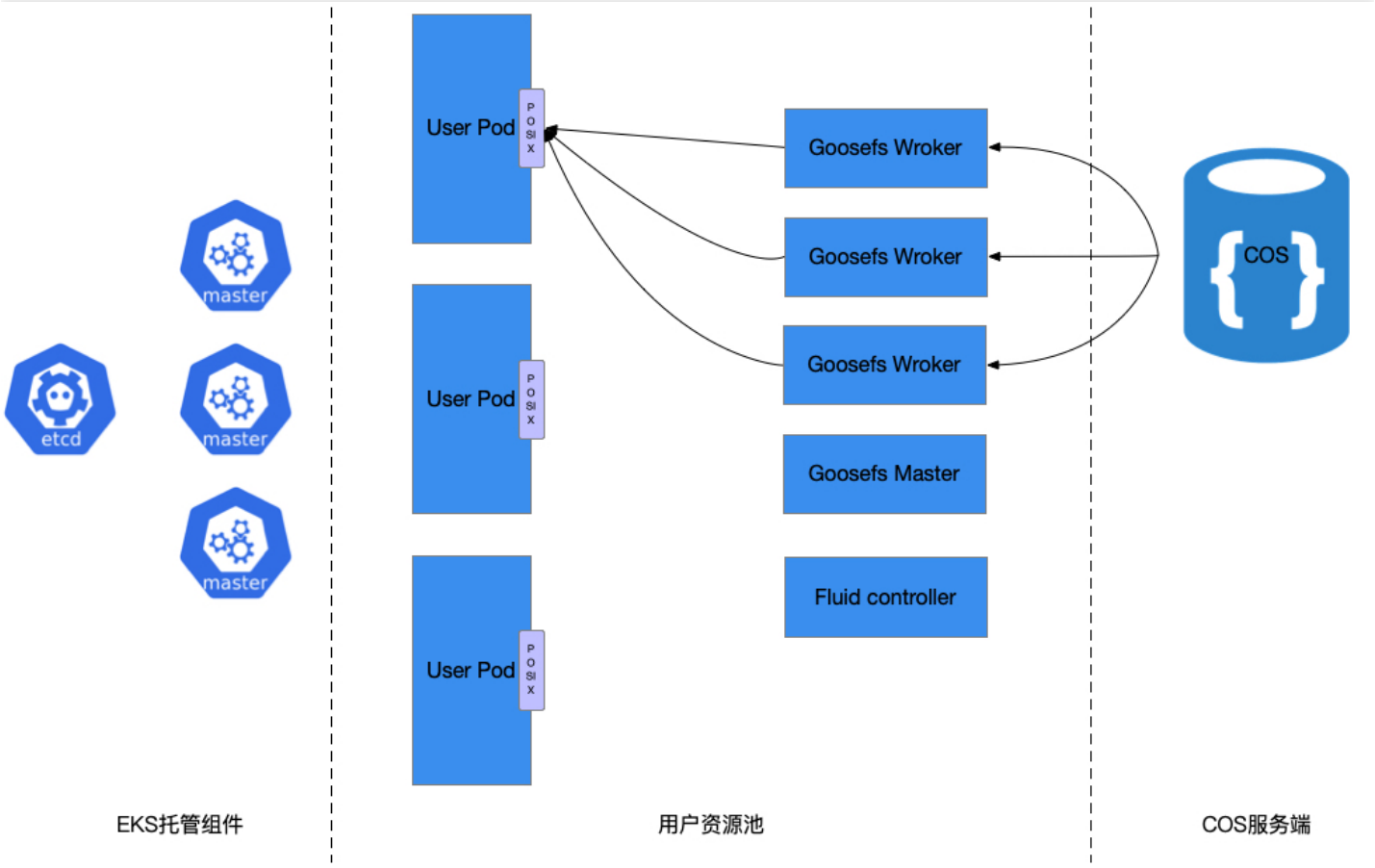
最近更新时间：2021-12-27 12:50:53

弹性容器服务（Elastic Kubernetes Service，EKS） 是腾讯云容器服务推出的无须用户购买节点即可部署工作负载的容器服务模式。EKS 完全兼容原生 Kubernetes，支持使用原生方式购买及管理资源，按照容器真实使用的资源量计费。EKS 还扩展支持腾讯云的存储及网络等产品，同时确保用户容器的安全隔离、开箱即用。

使用腾讯云 EKS 部署 GooseFS 可以充分利用 EKS 的弹性计算资源，构建按需按秒付费的对象存储（Cloud Object Storage，COS）存储访问加速服务。

## 架构说明

下图展示了使用腾讯云 EKS 部署 GooseFS 的通用架构。




如图中所示，整个架构由 EKS 托管组件、用户资源池和 COS 存储三大部分组成。其中用户资源池主要用于部署 GooseFS 集群，COS 存储作为远端存储系统，也支持替换为云 HDFS 这一公有云存储服务。具体构架过程中：

- GooseFS Master 和 Worker 都以 Kubernetes Statefulset 类型进行资源部署

- 使用 Fluid 拉起 GooseFS 集群
- Fuse Client 集成在用户 Pod 的沙箱（Sandbox）中
- 使用方式上保持与在标准 Kubernetes 一致

# 操作步骤

### 环境准备

1. 创建 EKS 集群，具体操作请参见 创建集群。
2. 开启集群访问，按实际情况选择内网或者外网，具体说明请参见 连接集群。
3. 执行 `kubectl get ns` 指令，确认集群可以使用：

```
-&gt; goosefs kubectl get ns
NAME STATUS AGE
default Active 7h31m
kube-node-lease Active 7h31m
kube-public Active 7h31m
kube-system Active 7h31m
```

4. 获取 `helm` ，可参考 Helm 官方文档 进行操作。

### 安装 GooseFS

1. 输入 `helm install` 指令安装 chart 包，安装 fluid：

```
-&gt; goosefs helm install fluid ./charts/fluid-on-tke
NAME: fluid
LAST DEPLOYED: Tue Jul 6 17:41:20 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

2. 查看 `fluid` 相关的 pod 状态：

```
-&gt; goosefs kubectl -n fluid-system get pod
NAME READY STATUS RESTARTS AGE
alluxioruntime-controller-78877d9d47-p2pv6 1/1 Running 0 59s
dataset-controller-5f565988cc-wnp7l 1/1 Running 0 59s
goosefsruntime-controller-6c55b57cd6-hr78j 1/1 Running 0 59s
```

3. 创建 `dataset` ，按实际需要修改相关变量，并执行 `kubectl apply -f dataset.yaml` 指令应用 `dataset` ：

```
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: ${dataset-name}
spec:
mounts:
- mountPoint: cosn://${bucket-name}
name: ${dataset-name}
options:
fs.cosn.userinfo.secretKey: XXXXXXX
fs.cosn.userinfo.secretId: XXXXXXX
fs.cosn.bucket.region: ap-${region}
fs.cosn.impl: org.apache.hadoop.fs.CosFileSystem
fs.AbstractFileSystem.cosn.impl: org.apache.hadoop.fs.CosN
fs.cos.app.id: ${user-app-id}
```

4. 创建 `GooseFS` 集群，使用以下 yaml，并执行 `kubectl apply -f runtime.yaml` ：

```
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: slice1
annotations:
master.goosefs.eks.tencent.com/model: c6
worker.goosefs.eks.tencent.com/model: c6
spec:
replicas: 6 # worker 数量，虽然控制器支持扩容，但是goosefs当前不支持数据的自动re-balanc
e
data:
replicas: 1 # goosefs 数据副本数
goosefsVersion:
imagePullPolicy: Always
image: ccr.ccs.tencentyun.com/cosdev/goosefs # goosefs 集群使用的镜像以及版本
imageTag: v1.0.1
tieredstore:
levels:
- mediumtype: MEM # 支持MEM，HDD，SSD 分别对应 内存，高效云盘，SSD云盘
path: /data
quota: 5G # 无论内存还是云盘都会生效，云盘最低为10G
high: "0.95"
low: "0.7"
properties:
goosefs.user.streaming.data.timeout: 5s
goosefs.job.worker.threadpool.size: "22"
goosefs.master.journal.type: UFS # UFS或者EMBEDDED，单master时使用UFS
# goosefs.worker.network.reader.buffer.size: 128MB
```

```
goosefs.user.block.size.bytes.default: 128MB
# goosefs.user.streaming.reader.chunk.size.bytes: 32MB
# goosefs.user.local.reader.chunk.size.bytes: 32MB
goosefs.user.metrics.collection.enabled: "false"
goosefs.user.metadata.cache.enabled: "true"
goosefs.user.metadata.cache.expiration.time: "2day"
master:
# 设定POD对应的虚拟机的规格，必填参数，不填写默认 1c1g
resources:
requests:
cpu: 8
memory: "16Gi"
limits:
cpu: 8
memory: "16Gi"
replicas: 1
# journal:
# volumeType: pvc
# storageClass: goosefs-hdd
jvmOptions:
- "-Xmx12G"
- "-XX:+UnlockExperimentalVMOptions"
- "-XX:ActiveProcessorCount=8"
- "-Xms10G"
worker:
jvmOptions:
- "-Xmx28G"
- "-Xms28G"
- "-XX:+UnlockExperimentalVMOptions"
- "-XX:MaxDirectMemorySize=28g"
- "-XX:ActiveProcessorCount=8"
resources:
requests:
cpu: 16
memory: "32Gi"
limits:
cpu: 16
memory: "32Gi"
fuse:
jvmOptions:
- "-Xmx4G"
- "-Xms4G"
- "-XX:+UseG1GC"
- "-XX:MaxDirectMemorySize=4g"
- "-XX:+UnlockExperimentalVMOptions"
- "-XX:ActiveProcessorCount=24"
```

5. 检查集群状态以及 PVC 状态：

```
-&gt; goosefs kubectl get pod
NAME READY STATUS RESTARTS AGE
slice1-master-0 2/2 Running 0 8m8s
slice1-worker-0 2/2 Running 0 8m8s
slice1-worker-1 2/2 Running 0 8m8s
slice1-worker-2 2/2 Running 0 8m8s
slice1-worker-3 2/2 Running 0 8m8s
slice1-worker-4 2/2 Running 0 8m8s
slice1-worker-5 2/2 Running 0 8m8s
-&gt; goosefs kubectl get pvc
slice1 Bound default-slice1 100Gi ROX fluid 7m37s # PVC名称与dataset名称一致，100Gi是一个虚拟值用作占位
```

### 数据加载

预加载数据只需要使用如下的 yaml 创建一个 resource 即可，yaml 示例如 `kubectl apply -f dataload.yaml`，执行后对应的响应示例如下：

```
apiVersion: data.fluid.io/v1alpha1
kind: DataLoad
metadata:
name: slice1-dataload
spec:
# 配置需要执行数据加载的 dataset信息
dataset:
name: slice1
namespace: default
```

创建后，可以通过 `kubectl get dataload slice1-dataload` 观察状态。

### 业务 Pod 挂载 PVC

用户业务容器按照 k8s 标准用法使用，具体请参见 Kubernetes 官方文档。

### 销毁 GooseFS 集群

销毁 GooseFS 集群可以通过 `delete` 指令进行删除，可以指定删除 master 和 worker 节点。**该操作属高危操作，请确保业务 pod 中没有对 Goosefs 的 IO 操作之后执行。**

```
-> goosefs kubectl get sts
NAME READY AGE
slice1-master 1/1 14m
slice1-worker 6/6 14m
```

```
-> goosefs kubectl delete sts slice1-master slice1-worker
statefulset.apps "slice1-master" deleted
statefulset.apps "slice1-worker" deleted
```

# 使用 Docker 部署

最近更新时间：2021-12-16 14:13:08

本文主要介绍如何使用 Docker 部署 GooseFS。

## 准备事项

1. 安装 Docker 19.03.14 及以上。
2. CentOS 7及以上。
3. 已获取 GooseFS docker 镜像，例如 goosefs:v1.0.0。

## 安装步骤

1. 创建 ufs 目录，将本机目录挂载到 GooseFS 根目录：

```
mkdir /tmp/goosefs_ufs
```

2. 启动 master 进程：

```
docker run -d --rm \
--net=host \
--name=goosefs-master \
-v /tmp/goosefs_ufs:/opt/data \
-e GOOSEFS_JAVA_OPTS=" \
-Dgoosefs.master.hostname=localhost \
-Dgoosefs.master.mount.table.root.ufs=/opt/data" \
goosefs:v1.0.0 master
```

说明

- -Dgoosefs.master.hostname：设置 master 地址。
- -Dgoosefs.master.mount.table.root.ufs：设置 GooseFS 根目录挂载点。
- -v /tmp/goosefs_ufs:/opt/data：将本地目录映射到 docker 容器内。
- --net=host：docker 采用 host 网络。

3. 启动 worker 进程：

```
docker run -d --rm \
--net=host \
--name=goosefs-worker1 \
--shm-size=1G \
-e GOOSEFS_JAVA_OPTS=" \
-Dgoosefs.worker.memory.size=1G \
-Dgoosefs.master.hostname=localhost" \
goosefs:v1.0.0 worker
```

# 操作演示

1. 查看容器：

```
[root@VM-0-7-centos ~]# docker ps | grep goosefs
0bda1cac76f4 goosefs:v1.0.0 "/entrypoint.sh mast…" 32 minutes ago Up 32 minute
goosefs-master
b6260f9a0134 goosefs:v1.0.0 "/entrypoint.sh work…" About an hour ago Up About a
n hour goosefs-worker1
```

2. 进入容器：

```
docker exec -it 0bda1cac76f4 /bin/bash
```

3. 挂载 COS 目录：

```
goosefs fs mount --option fs.cosn.userinfo.secretId={secretId} \
--option fs.cosn.userinfo.secretKey={secretKey} \
--option fs.cosn.bucket.region=ap-beijing \
--option fs.cosn.impl=org.apache.hadoop.fs.CosFileSystem \
--option fs.AbstractFileSystem.cosn.impl=org.apache.hadoop.fs.CosN \
/cosn {cos桶}
```

4. 查看目录：

```
[goosefs@VM-0-7-centos goosefs-1.0.0-SNAPSHOT-noUI-noHelm]$ goosefs fs ls /
drwxrwxrwx goosefs goosefs 1 PERSISTED 01-01-1970 08:00:00:000 DIR /cosn
drwxr-xr-x root root 0 PERSISTED 06-25-2021 11:01:24:000 DIR /my
```

5. 查看 worker 节点：

```
[goosefs@VM-0-7-centos goosefs-1.0.0-SNAPSHOT-noUI-noHelm]$ goosefs fsadmin repor
t capacity
Capacity information for all workers:
Total Capacity: 1024.00MB
Tier: MEM Size: 1024.00MB
Used Capacity: 0B
Tier: MEM Size: 0B
Used Percentage: 0%
Free Percentage: 100%
Worker Name Last Heartbeat Storage MEM
172.31.0.7 0 capacity 1024.00MB
used 0B (0%)
```

# 运维指南
# 日志指引
# GooseFS 日志介绍

最近更新时间：2021-09-30 12:29:06

GooseFS 的 Master 和 Worker 节点，以及 Spark 等计算框架通过 GooseFS Client 请求 GooseFS 时，都会记录请求日志，用户可对输出日志进行分析，进行问题排查。GooseFS 日志输出基于 log4j 实现，可以通过修改 log4j.properties 配置文件来调整 GooseFS 的日志输出，例如日志存储路径，日志级别，是否记录 RPC 调用情况等。用户可以到 GooseFS 的配置文件目录下，打开并修改 log4j.properties 文件：

```
$ cd /usr/local/service/goosefs/conf
$ cat log4j.properties
# May get overridden by System Property
log4j.rootLogger=INFO, ${goosefs.logger.type}, ${goosefs.remote.logger.type}
log4j.category.goosefs.logserver=INFO, ${goosefs.logserver.logger.type}
log4j.additivity.goosefs.logserver=false
log4j.logger.AUDIT_LOG=INFO, ${goosefs.master.audit.logger.type}
log4j.additivity.AUDIT_LOG=false
...
```

下文将详细介绍 GooseFS 的日志配置：

## 日志存储位置

GooseFS 采集的日志默认存储在 ${GOOSEFS_HOME}/logs 目录下。其中，Master 采集的日志存储在 logs/master.log 中，Worker 采集的日志存储在 logs/worker.log 中。需要注意的是， 节点进程异常抛出的日志会记录在 master.out 或者 worker.out 中，正常情况下这两类文件均为空文件，系统有异常时会记录异常信息以便追溯。

以 Master 节点的日志存储配置为例，以下为常用的几项配置：

```
# Appender for Master
log4j.appender.MASTER_LOGGER=org.apache.log4j.RollingFileAppender
log4j.appender.MASTER_LOGGER.File=${goosefs.logs.dir}/master.log
log4j.appender.MASTER_LOGGER.MaxFileSize=10MB
log4j.appender.MASTER_LOGGER.MaxBackupIndex=100
log4j.appender.MASTER_LOGGER.layout=org.apache.log4j.PatternLayout
log4j.appender.MASTER_LOGGER.layout.ConversionPattern=%d{ISO8601} %-5p %c{1} - %
m%n
```

配置参数介绍如下：

- MASTER_LOGGER：指定配置 MASTER 的日志输出。
- MASTER_LOGGER.File：指定日志存储路径，可以通过修改路径来自定义日志存储位置。
- MASTER_LOGGER.MaxFileSize：指定单个日志文件大小的上限。
- MASTER_LOGGER.MaxBackupIndex：指定日志文件数上限。
- MASTER_LOGGER.layout：指定日志输出格式模板。
- MASTER_LOGGER.layout.ConversionPattern：指定日志输出的具体格式。

注意：

- .log 文件是滚动存储的，您可以将其备份到 UFS，例如对象存储中；.out 文件不滚动存储，如果需要清理 .out 文件需要手动发起删除操作。
- 更多 log4j 的参数配置可以参考 log4j configuration 文档。
- GooseFS 仅存储本身产生的日志，上层计算应用产生的日志可以根据计算应用的日志配置，查看日志存储位置。常见计算应用的日志配置信息可见：Apache Hadoop, Apache HBase, Apache Hive, Apache Spark。

## 日志级别

GooseFS 提供了以下5种级别的日志：

- TRACE: 详细的调用日志，适用于调试方法和类的调用。
- DEBUG: 较详细的调用日志，适用于 DEBUG 过程中排查问题。
- INFO: 请求处理过程中的关键信息。
- WARN: 警告类信息，任务可以继续执行，但需要注意可能存在问题。
- ERROR: 系统报错信息，影响任务进行。

上述5种级别日志的详细程度从高到低，配置高等级的日志级别会一并记录低等级的日志信息。默认情况下 GooseFS 配置 INFO 级别的日志，记录 INFO、WARN、ERROR三个级别的日志。

可以到 GooseFS 的配置文件目录下打开并修改 log4j.properties 文件，如下示例展示了将所有 GooseFS 的日志级别修改为 DEBUG 级别：

```
log4j.rootLogger=DEBUG, ${goosefs.logger.type}, ${goosefs.remote.logger.type}
```

如果需要修改指定类的日志级别，可以在配置文件中添加声明，如下示例展示指定 GooseFSFileInStream 这个类的日志级别为 DEBUG：

```
log4j.logger.com.qcloud.cos.goosefs.client.file.GooseFSFileInStream=DEBUG
```

一般而言，推荐在日志配置文件中修改日志级别。但在一些特定的场景下，用户可能需要在集群运行过程中修改日志参数，此时可以通过在命令行中输入 goosefs logLevel 指令进行调整。如下为 logLevel 支持的配置选项：

```
usage: logLevel [--level <arg>] --logName <arg> [--target <arg>]
--level <arg> The log level to be set.
--logName <arg> The logger's name(e.g. com.qcloud.cos.goosefs.master.file.Default
FileSystemMaster) you want to get or set level.
--target <arg> <master|workers|host:webPort>. A list of targets separated by, can
be specified. host:webPort pair must be one of workers. Default target is master
and all workers
```

各项配置的详细说明如下：

- level：日志级别，支持 TRACE、DEBUG、INFO、WARN、ERROR 五种级别。
- logName：日志输出 logger，如 com.qcloud.cos.goosefs.underfs.hdfs.HdfsUnderFileSystem 等。
- target：修改范围，可以设置为指定的 Master 或者 Worker 节点（通过 IP：PORT 方式指定），默认为 Master 和所有 Worker 节点。

用户可以按需在系统运行过程中调整日志级别，以便排查系统运行过程中的问题。如以下示例展示了在运行过程中将所有 Worker 节点上 com.qcloud.cos.goosefs.underfs.hdfs.HdfsUnderFileSystem 这个类的日志级别调整为 DEBUG 级别，并在调试完成后调整回 INFO 级别：

```
$ goosefs logLevel --logName=com.qcloud.cos.goosefs.underfs.hdfs.HdfsUnderFileSys
tem --target=workers --level=DEBUG # 调整为 DEBUG 模式
$ goosefs logLevel --logName=com.qcloud.cos.goosefs.underfs.hdfs.HdfsUnderFileSys
tem --target=workers --level=INFO # 调整为 INFO 模式
```

# 高级配置

GooseFS 支持配置 GC 事件日志，FUSE 接口日志，RPC 调用日志，UFS 操作日志，以及进行日志分割和日志筛选等操作。以下介绍部分常用高级配置的使用方式。

- **GC 事件日志**
  GooseFS 将 GC 事件日志记录在 .out 文件中， 可以通过在 conf/goosefs-env.sh 中添加如下配置：

  ```
  GOOSEFS_JAVA_OPTS+=" -XX:+PrintGCDetails -XX:+PrintTenuringDistribution -XX:+Pr
  intGCTimeStamps"
  ```

GOOSEFS_JAVA_OPTS 为所有类型 GooseFS 节点的 Java 虚拟机参数，也可以通过指定 GOOSEFS_MASTER_JAVA_OPTS 和 GOOSEFS_WORKER_JAVA_OPTS 分别指定 Master 和 Worker上的虚拟机参数。

- **FUSE 接口日志**
  可以在 conf/log4j.properties 文件中配置记录 FUSE 日志级别：

```
goosefs.logger.com.qcloud.cos.goosefs.fuse.GoosefsFuseFileSystem=DEBUG
```

  启用后 FUSE 接口日志可以在 logs/fuse.log 查看。

- **启用 RPC 调用日志**
  GooseFS 支持在 conf/log4j.properties 配置文件中，配置 Client 端或 Master 端的 RPC 调用日志。
  可以通过在 log4j.properties 文件中，配置客户端输出 RPC 请求日志：

```
log4j.logger.com.qcloud.cos.goosefs.client.file.FileSystemMasterClient=DEBUG # Client 与 FileSystemMaster 之间的 RPC 请求日志
log34j.logger.com.qcloud.cos.goosefs.client.block.BlockSystemMasterClient=DEBUG # Client 与 BlockMaster 之间的 RPC 请求日志
```

  可以通过 logLevel 指令来，配置 Master 输出 RPC 请求日志：

```
$ goosefs logLevel \--logName=com.qcloud.cos.goosefs.master.file.FileSystemMasterClientServiceHandler \--target master --level=DEBUG # 文件相关的 RPC 请求日志
$ goosefs logLevel \--logName=com.qcloud.cos.goosefs.master.block.BlockSystemMasterClientServiceHandler \--target master --level=DEBUG # 块相关别的 RPC 请求日志
```

- **UFS 操作日志**
  UFS 操作日志输出配置，可以通过在 log4j.properties 文件中添加配置项，或者通过 logLevel 指令进行操作。下面以 logLevel 指令为例：

```
$ goosefs logLevel \--logName=com.qcloud.cos.goosefs.underfs.UnderFileSystemWithLogging \--target master --level=DEBUG # 记录 master 节点对 UFS的操作日志
$ goosefs logLevel \--logName=com.qcloud.cos.goosefs.underfs.UnderFileSystemWithLogging \--target workers --level=DEBUG # 记录 worker 节点对 UFS的操作日志
```

- **分割日志**
  GooseFS 支持将指定类型日志存储在指定存储路径，如果将所有日志都记录在 .log 文件，可能产生以下问题：

- 当集群规模大，吞吐量较多时，master.log 或者 worker.log 文件可能变得异常庞大，或者滚动产生大量日志文件。
- 日志信息较多，不利于针对性分析的进行日志分析。
- 大量日志存储在本地节点，占用空间。

因此业务可以根据需要在 log4j.properties 文件中添加配置项，将指定类产生的日志存储至指定文件路径，如下示例展示了将 StateLockManager 类的日志存储在 statelock.log 中：

```
log4j.category.com.qcloud.cos.goosefs.master.StateLockManager=DEBUG, State_LOCK
_LOGGER
log4j.additivity.com.qcloud.cos.goosefs.master.StateLockManager=false
log4j.appender.State_LOCK_LOGGER=org.apache.log4j.RollingFileAppender
log4j.appender.State_LOCK_LOGGER.File=<GOOSEFS_HOME>/logs/statelock.log
log4j.appender.State_LOCK_LOGGER.MaxFileSize=10MB
log4j.appender.State_LOCK_LOGGER.MaxBackupIndex=100
log4j.appender.State_LOCK_LOGGER.layout=org.apache.log4j.PatternLayout
log4j.appender.State_LOCK_LOGGER.layout.ConversionPattern=%d{ISO8601} %-5p %c
{1} - %m%
```

- **筛选日志**

  GooseFS 支持按照一定的条件筛选并记录日志，而不是全量记录所有日志。例如在进行性能测试时需要记录 RPC 调用日志，但业务侧并不需要记录所有 RPC 调用日志，只需要记录某些延迟较高的日志即可，此时可以通过在 log4j.properties 文件中添加配置项添加日志筛选条件即可，如下示例分别展示了筛选 RPC 调用延迟超过200ms的请求和 FUSE 调用超过1s的请求：

```
goosefs.user.logging.threshold=200ms
goosefs.fuse.logging.threshold=1s
```

# 监控指南
# 获取 GooseFS 监控指标

最近更新时间：2022-03-09 18:02:48

Goosefs 基于 Coda Hale Metrics Library 库记录监控数据，支持通过命令行、控制台、文件等多种途径获取指标，目前支持的指标获取方式包括：

- MetricsServlet：将监控指标以 Json 格式提供给用户。
- CsvSink：通过 CSV 文件方式展示监控指标，配置后会周期性地生成记录监控指标的 CSV 文件。
- PrometheusMetricsServlet:将监控指标以 Prometheus 定义的格式提供给用户。

上述监控指标的配置可以通过配置文件来指定。GooseFS 监控指标的配置文件默认文件路径为 `$GooseFS_HOME/conf/metrics.properties`，支持通过 goosefs.metrics.conf.file 指定自定义监控配置文件。GooseFS 为用户提供了一个默认模板 metrics.properties.template，包含了所有可配置的属性。

## 获取监控指标

以下介绍三种基础的获取监控指标的途径：

### 1. 通过 JSON 格式拉取监控指标

GooseFS 默认的获取监控指标的方式是通过 JSON 格式拉取，对应着 MetricsServlet 这一配置项。可以在命令行中向 GooseFS 的 Leading Master 节点发起一个 HTTP 请求，拉取所需的监控指标，其中 master 的 metrics 端口为 9201，worker 的 metrics 端口为 9204。请求指令格式如下：

```
$ curl <LEADING_MASTER_HOSTNAME>:<MASTER_WEB_PORT>/metrics/json
```

如上示例中， 需要是合法的 MASTER 节点的IP， 需要为已经启用的端口。

如果需要获取某个 WORKER 节点的监控指标，可以通过如下方式获取：

```
$ curl <WORKER_HOSTNAME>:<WORKER_WEB_PORT>/metrics/json
```

### 2. 通过 CSV 文件获取监控指标

GooseFS 支持将数据导出为 CSV 格式文件，通过该能力获取监控指标，首先需要准备一个存储监控指标的目录：

```
$ mkdir /tmp/goosefs-metrics
```

准备好存储路径后，修改配置文件 conf/metrics.properties ，启用 CsvSink 能力：

```
sink.csv.class=goosefs.metrics.sink.CsvSink # 启用CsvSink能力
sink.csv.period=1 # 设置监控指标导出周期
sink.csv.unit=senconds # 设置监控指标导出周期的单位
sink.csv.directory=/tmp/goosefs-metrics # 设置监控指标导出路径
```

配置好后需要重启节点以便配置生效。配置生效后，监控指标将周期性地导出成 CSV 格式并存储在指定路径下。

注意：

- GooseFS 准备了监控配置模板，可以参考 conf/metrics.properties.template文件；
- 如果 GooseFS 是集群化部署，需要保证指定的指标存储路径能被所有节点读取。

### 3. 拉取 Prometheus 监控指标

GooseFS master 和 worker 的 Prometheus 的监控指标可用如下的命令查看，其中 master 的 metrics 端口为9201，worker 的 metrics 端口为 9204：

```
curl <LEADING_MASTER_HOSTNAME>:<MASTER_WEB_PORT>/metrics/prometheus/
# HELP Master_CreateFileOps Generated from Dropwizard metric import (metric=Maste
r.CreateFileOps, type=com.codahale.metrics.Counter)
...
curl <WORKER_IP>:<WOKER_PORT>/metrics/prometheus/
# HELP pools_Code_Cache_max Generated from Dropwizard metric import (metric=pool
s.Code-Cache.max, type=com.codahale.metrics.jvm.MemoryUsageGaugeSet$$Lambda$51/13
7460818)
...
```

# 基于 Prometheus 搭建 GooseFS 监控体系

最近更新时间：2022-01-04 12:39:27

Goosefs 可以通过配置将指标数据输出到不同的监控系统中，Prometheus 是其中之一。Prometheus 是一个开源的监控框架，目前腾讯云监控已集成了 Prometheus，下文将重点介绍 Goosefs 监控指标，以及将监控指标上报到自建的 Prometheus 和云上 Prometheus 的流程。

## 准备工作

通过 Prometheus 构建监控体系需要先做如下准备工作：

- 配置 GooseFS 集群
- 下载 Prometheus 官方安装包或腾讯云 Prometheus 安装包
- 下载和配置 Grafana

## 启用 GooseFS 监控指标上报配置

1. 编辑 GooseFs 配置 conf/goosefs-site.properties， 添加如下配置项，并使用 goosefs copyDir conf/ 拷贝到所有 worker节点，并重启集群 `./bin/goosefs-start.sh all` 。

```
goosefs.user.metrics.collection.enabled=true
goosefs.user.metrics.heartbeat.interval=10s
```

2. master 和 worker 的 Prometheus 的监控指标可用如下的命令查看，其中 master 的 metrics 端口为9201，worker 的 metrics 端口为 9204：

```
curl <LEADING_MASTER_HOSTNAME>:<MASTER_WEB_PORT>/metrics/prometheus/
# HELP Master_CreateFileOps Generated from Dropwizard metric import (metric=Mas
ter.CreateFileOps, type=com.codahale.metrics.Counter)
...
curl <WORKER_IP>:<WOKER_PORT>/metrics/prometheus/
# HELP pools_Code_Cache_max Generated from Dropwizard metric import (metric=poo
ls.Code-Cache.max, type=com.codahale.metrics.jvm.MemoryUsageGaugeSet$$Lambda$5
1/137460818)
...
```

# 上报监控指标到自建 Prometheus

1. 下载 Prometheus 安装包并解压，修改 prometheus.yml：

```
# prometheus.yml
global:
scrape_interval: 10s
evaluation_interval: 10s
scrape_configs:
- job_name: 'goosefs masters'
metrics_path: /metrics/prometheus
file_sd_configs:
- refresh_interval: 1m
files:
- "targets/cluster/masters/*.yml"
- job_name: 'goosefs workers'
metrics_path: /metrics/prometheus
file_sd_configs:
- refresh_interval: 1m
files:
- "targets/cluster/workers/*.yml"
```

2. 创建 targets/cluster/masters/masters.yml，添加 master 的 IP 和 port：

```
- targets:
- "<TARGERTS_MASTER_IP>:<TARGERTS_MASTER_PORT>"
```

3. 创建 targets/cluster/workers/workers.yml，添加 worker 的 IP 和 port：

```
- targets:
- "<TARGERTS_WORKER_IP>:<TARGERTS_WORKER_PORT>"
```

4. 启动 Prometheus，其中 --web.listen-address 指定 Prometheus 监听地址，默认端口号 9090：

```
nohup ./prometheus --config.file=prometheus.yml --web.listen-address="<LISTEN_I
P>:<LISTEN_PORT>" > prometheus.log 2>&1 &
```

5. 查看可视化界面：

```
http://<PROMETHEUS_BI_IP>:<PROMETHEUS_BI_PORT>
```

6. 查看机器实例：

```
http://<PROMETHEUS_BI_IP>:<PROMETHEUS_BI_PORT>/targets
```

# 上报监控指标到腾讯云 Prometheus

1. 按照安装指南中的指引，在 master 机器上安装 Promethus agent：

```
wget https://rig-1258344699.cos.ap-guangzhou.myqcloud.com/prometheus-agent/agent_install && chmod +x agent_install && ./agent_install prom-12kqy0mw agent-grt164ii ap-guangzhou <secret_id> <secret_key>
```

2. 配置 master 和 worker 的抓取任务：

**方式一：**

```
job_name: goosefs-masters
honor_timestamps: true
metrics_path: /metrics/prometheus
scheme: http
file_sd_configs:
- files:
- /usr/local/services/prometheus/targets/cluster/masters/*.yml
refresh_interval: 1m
job_name: goosefs-workers
honor_timestamps: true
metrics_path: /metrics/prometheus
scheme: http
file_sd_configs:
- files:
- /usr/local/services/prometheus/targets/cluster/workers/*.yml
refresh_interval: 1m
```

注意：

job_name 中没有空格，而单机的 Prometheus 的 job_name 中可以包含空格。

**方式二：**

```
job_name: goosefs masters
honor_timestamps: true
metrics_path: /metrics/prometheus
scheme: http
static_configs:
- targets:
- "<TARGERTS_MASTER_IP>:<TARGERTS_MASTER_PORT>"
refresh_interval: 1m

job_name: goosefs workers
honor_timestamps: true
metrics_path: /metrics/prometheus
scheme: http
static_configs:
- targets:
- "<TARGERTS_WORKER_IP>:<TARGERTS_WORKER_PORT>"
refresh_interval: 1m
```

注意：
抓取任务按方式二配置，则无需在 targets/cluster/masters/ 路径下创建 masters.yml 和 workers.yml 文件。

# 使用 Grafana 查看监控指标

1. 启动 Grafana：

```
nohup ./bin/grafana-server web > grafana.log 2>&1 &
```

2. 打开登录页面 http://<GRAFANA_IP>:<GRAFANA_PORT>，Grafana 的默认端口为 3000，username 和 password 都是 admin，首次登录后可修改密码。

3. 进入页面后，添加 Prometheus 的 Datasource：

```
<PROMETHEUS_IP>:<PROMETHEUS_PORT>
```

4. 导入 Goosefs 的 Grafana 模板，选择 json 导入（点此下载 json），并选择上面创建的 Datasource。

> 注意：
> 云上 Prometheus 购买时需设置密码，云上 Grafana 的可视化监控界面配置和上面类似，注意 job_name 需要配置成一致。

5. 修改 DashBoard 以后，可以将 DashBoard 导出来。

# 集群配置实践

最近更新时间：2022-04-18 12:39:40

数据加速器 GooseFS 提供了多种部署方式，本文档主要介绍在大数据场景中，通常使用的集群模式部署。

- AI 场景生产环境配置实践
- 大数据场景生产环境配置实践

# 数据安全
# 使用 Apache Ranger 控制 GooseFS 的访问权限

最近更新时间：2022-02-15 14:21:55

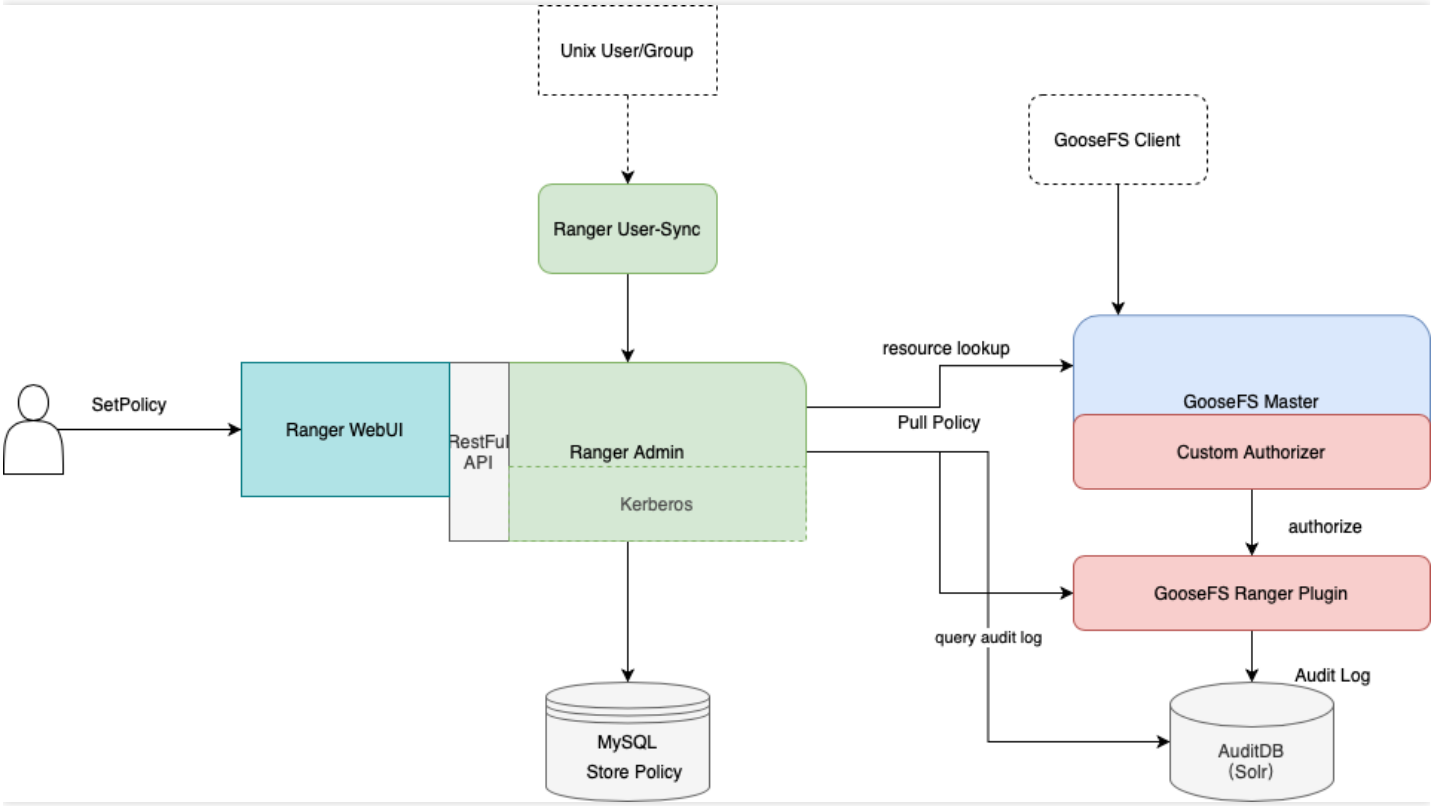
## 概述

Apache Ranger 是大数据生态系统中用于控制访问权限的一个标准鉴权组件，GooseFS 作为大数据和数据湖场景下的加速存储系统，也已经支持接入 Apache Ranger 的统一鉴权平台中，本文将介绍使用 Apache Ranger 控制 GooseFS 的资源访问权限。

## 优势

- GooseFS 作为一款云原生加速存储系统，在 Apache Ranger 支持上已经做到了与 HDFS 近乎一致的访问控制行为。因此，原先使用 HDFS 的大数据用户可以非常轻松地迁移到 GooseFS 上来，并且直接复用 HDFS 的 Ranger 权限策略，即可获得一致的使用体验。
- GooseFS with Ranger 相比 HDFS with Ranger 的鉴权架构，还额外提供了 Ranger + 原生 ACL 的联合鉴权选项，在 Ranger 鉴权失效时，还可选择使用原生 ACL 鉴权，可解决一些 Ranger 鉴权策略配置不完善的问题。

## GooseFS with Ranger 的鉴权架构




为了支持将 GooseFS 集成到 Ranger 鉴权平台中，我们开发了 GooseFS Ranger Plugin，它同时部署在 GooseFS Master 节点和 Ranger Admin 侧。负责完成如下工作：

- GooseFS Master 节点侧：
- 提供 `Authorizer` 接口，为 GooseFS Master 上的每一次元数据请求提供鉴权结果。
- 连接 Ranger Admin 获取用户配置的鉴权策略。
- Ranger Admin 侧：
- 为 Ranger Admin 提供 GooseFS 的资源查找（resource lookup）的能力。
- 提供配置校验的能力。

# 开始部署

### 准备工作

在开始使用前，需确保环境中已部署并配置了 Ranger 相关的组件（即包括：Ranger Admin 和 Ranger UserSync），并且确保 Ranger 的 WebUI 可以正常打开和使用。

### 部署组件

#### 在 Ranger Admin 侧部署 GooseFS Ranger Plugin 并注册对应服务

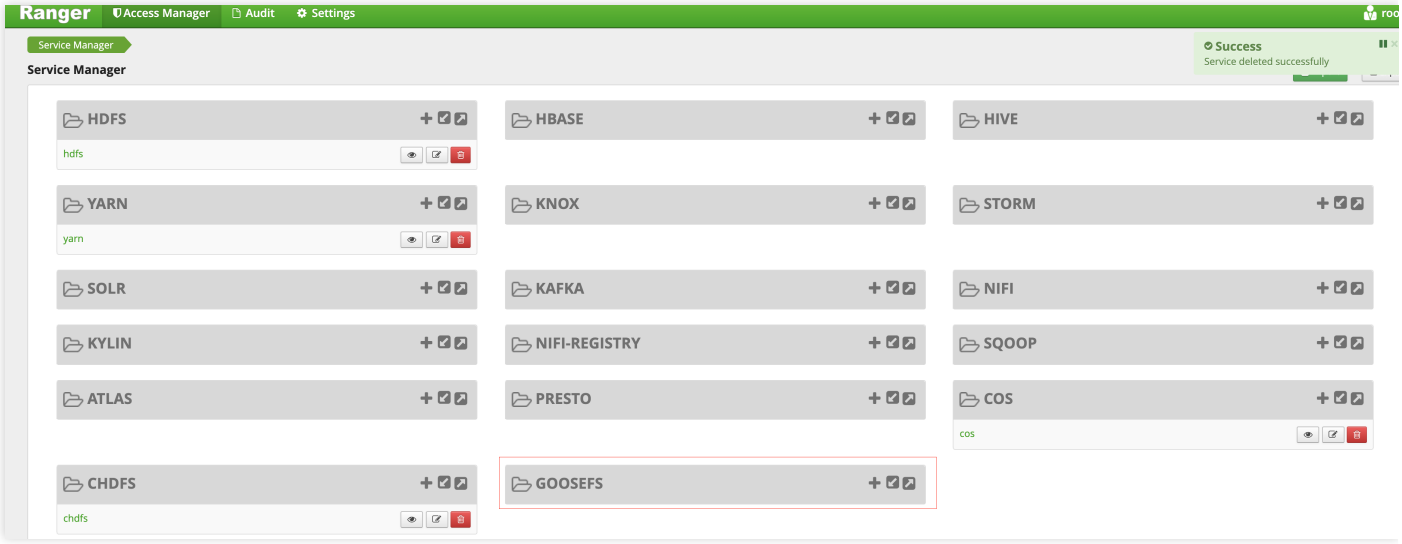
> 说明：
> 单击 此处 下载 GooseFS Ranger Plugin。

部署步骤如下：

1. 在 Ranger 服务定义目录下新建 GooseFS 的目录（注意，目录权限至少保证 x 与 r 的权限）。

2. 如果使用的是腾讯云 EMR 集群，则 Ranger 的服务定义目录
   在：`/usr/local/service/ranger/ews/webapp/WEB-INF/classes/ranger-plugins` 。

3. 如果是自建 Hadoop 集群，则可以通过在 ranger 目录下查找 hdfs 等已经接入到 ranger 服务的组件，查找目录位置。




4. 在 GooseFS 的目录下，放入 goosefs-ranger-plugin-${version}.jar 和 ranger-servicedef-goosefs.json，并且具备读权限。

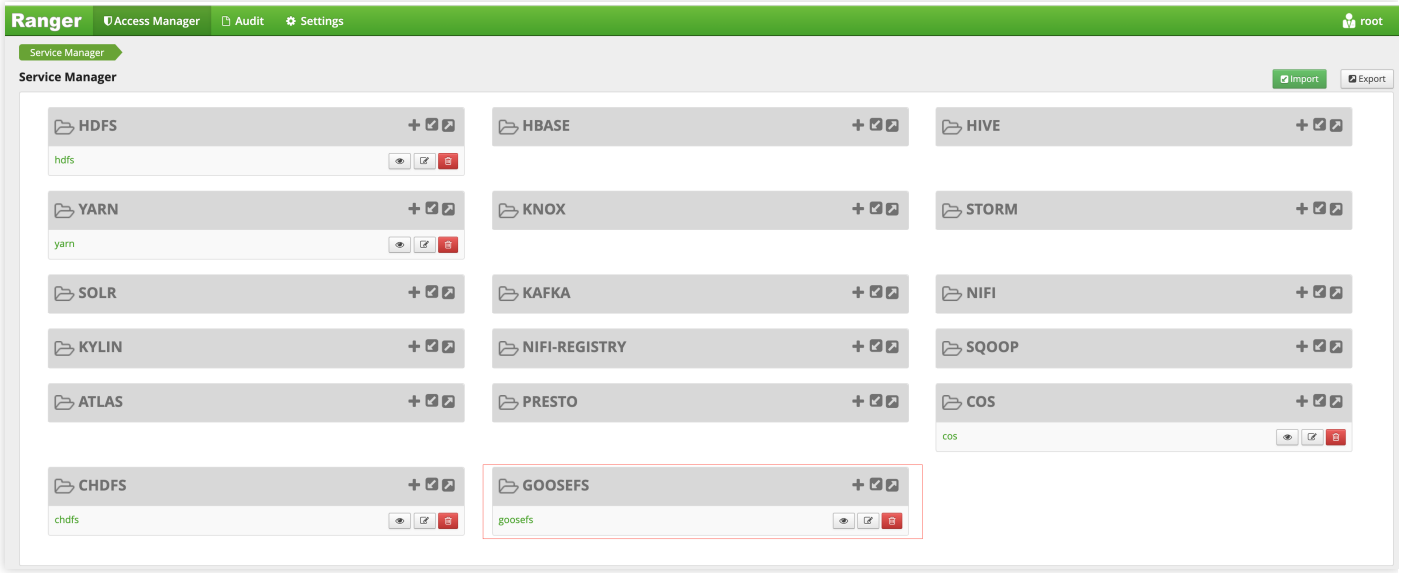
5. 重启 Ranger 服务。

6. 在 Ranger 上，按照如下命令，注册 GooseFS Service。

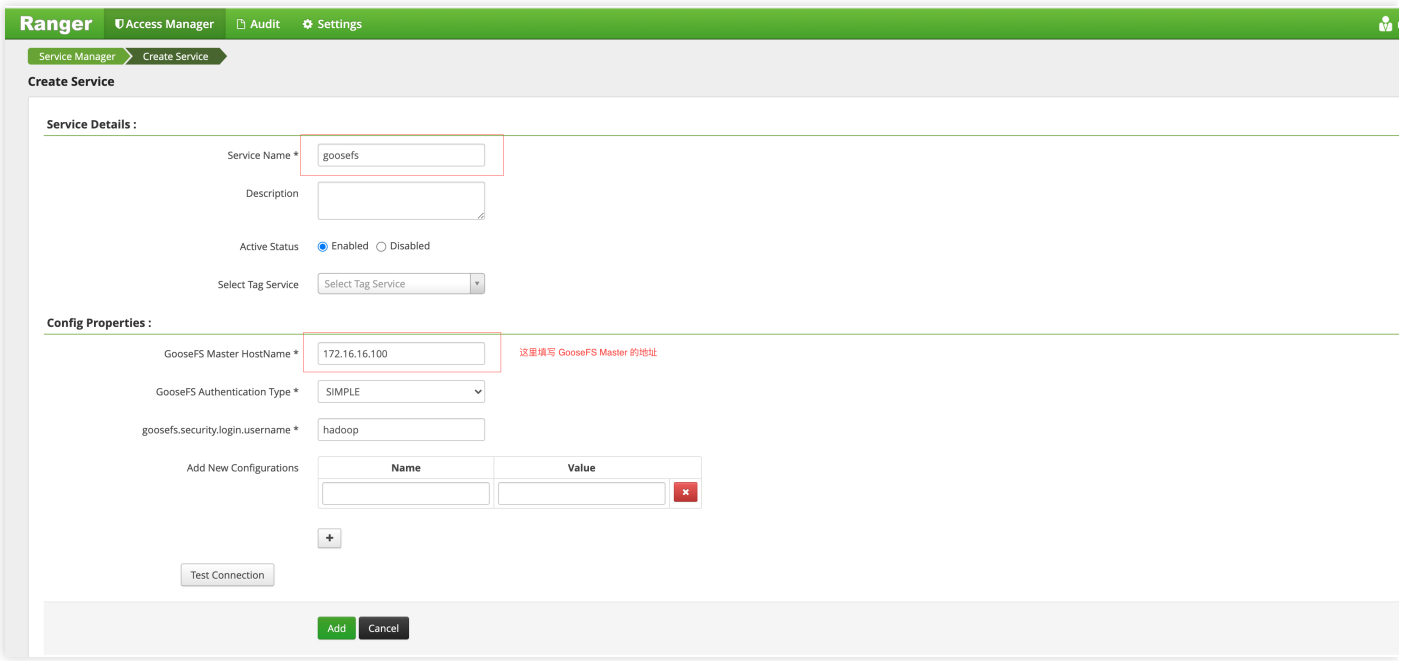
```
# 生成服务，需要传入 Ranger 管理员的账号和密码，以及 Ranger 的服务地址
# 对于腾讯云 EMR 集群，管理员用户是 root，密码是构建 EMR 集群时设置的 root 密码，ranger 服务的 IP 就是 EMR 服务的 Master IP
adminUser=root
adminPasswd=xxxx
rangerServerAddr=10.0.0.1:6080
```

```
curl -v -u${adminUser}:${adminPasswd} -X POST -H "Accept:application/json" -H
"Content-Type:application/json" -d @./ranger-servicedef-goosefs.json http://${r
angerServerAddr}/service/plugins/definitions
# 服务注册成功后，会返回一个服务 ID，请务必记录下这个ID
# 如果要删除 GooseFS 的服务，则传入刚刚返回的服务 ID，执行如下命令即可：
serviceId=104
curl -v -u${adminUser}:${adminPasswd} -X DELETE -H "Accept:application/json" -H
"Content-Type:application/json" http://${rangerServerAddr}/service/plugins/defi
nitions/${serviceId}
```

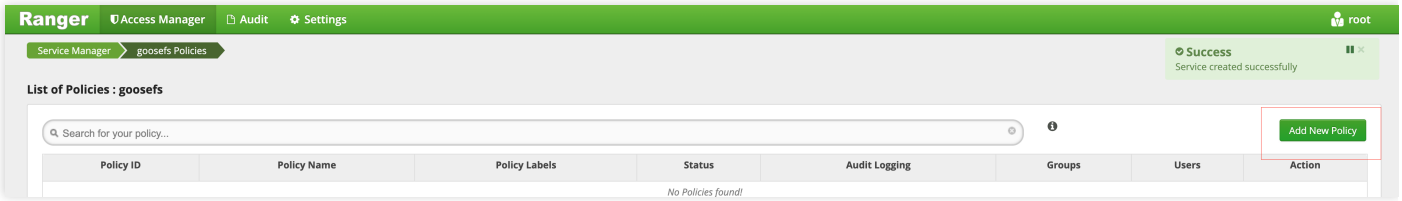
7. 创建成功后，在 Ranger 的 Web 控制台上即可看到 GooseFS 相关的服务：



8. 在 GooseFS 服务侧单击【+】，定义 goosefs 服务实例。




9. 单击新生成的 goosefs 服务实例，即可添加鉴权 policy。




### 在 GooseFS Master 侧部署 GooseFS Ranger Plugin 并配置启用 Ranger 鉴权

1. 将 goosefs-ranger-plugin-${version}.jar 放入 ${GOOSEFS_HOME}/lib 路径下，并且至少具备读权限。
2. 将 ranger-goosefs-audit.xml、ranger-goosefs-security.xml 以及 ranger-policymgr-ssl.xml 三个文件放入 `\${GOOSEFS_HOME}/conf` 路径下，并分别填写其必要配置：

- ranger-goosefs-security.xml：

```
<configuration xmlns:xi="http://www.w3.org/2001/XInclude">
<property>
<name>ranger.plugin.goosefs.service.name</name>
<value>goosefs</value>
</property>
<property>
<name>ranger.plugin.goosefs.policy.source.impl</name>
<value>org.apache.ranger.admin.client.RangerAdminRESTClient</value>
```

```
</property>
<property>
<name>ranger.plugin.goosefs.policy.rest.url</name>
<value>http://10.0.0.1:6080</value>
</property>
<property>
<name>ranger.plugin.goosefs.policy.pollIntervalMs</name>
<value>30000</value>
</property>
<property>
<name>ranger.plugin.goosefs.policy.rest.client.connection.timeoutMs</name>
<value>1200</value>
</property>
<property>
<name>ranger.plugin.goosefs.policy.rest.client.read.timeoutMs</name>
<value>30000</value>
</property>
</configuration>
```

- ranger-goosefs-audit.xml（不开启审计，可不配置）：

```
<configuration>
<property>
<name>xasecure.audit.is.enabled</name>
<value>false</value>
</property>
<property>
<name>xasecure.audit.db.is.async</name>
<value>true</value>
</property>
<property>
<name>xasecure.audit.db.async.max.queue.size</name>
<value>10240</value>
</property>
<property>
<name>xasecure.audit.db.async.max.flush.interval.ms</name>
<value>30000</value>
</property>
<property>
<name>xasecure.audit.db.batch.size</name>
<value>100</value>
</property>
<property>
<name>xasecure.audit.jpa.javax.persistence.jdbc.url</name>
<value>jdbc:mysql://localhost:3306/ranger_audit</value>
```

```
</property>
<property>
<name>xasecure.audit.jpa.javax.persistence.jdbc.user</name>
<value>rangerLogger</value>
</property>
<property>
<name>xasecure.audit.jpa.javax.persistence.jdbc.password</name>
<value>none</value>
</property>
<property>
<name>xasecure.audit.jpa.javax.persistence.jdbc.driver</name>
<value>com.mysql.jdbc.Driver</value>
</property>
<property>
<name>xasecure.audit.credential.provider.file</name>
<value>jceks://file/etc/ranger/hadoopdev/auditcred.jceks</value>
</property>
<property>
<name>xasecure.audit.hdfs.is.enabled</name>
<value>true</value>
</property>
<property>
<name>xasecure.audit.hdfs.is.async</name>
<value>true</value>
</property>
<property>
<name>xasecure.audit.hdfs.async.max.queue.size</name>
<value>1048576</value>
</property>
<property>
<name>xasecure.audit.hdfs.async.max.flush.interval.ms</name>
<value>30000</value>
</property>
<property>
<name>xasecure.audit.hdfs.config.encoding</name>
<value></value>
</property>
<!-- hdfs audit provider config-->
<property>
<name>xasecure.audit.hdfs.config.destination.directory</name>
<value>hdfs://NAMENODE_HOST:8020/ranger/audit/</value>
</property>
<property>
<name>xasecure.audit.hdfs.config.destination.file</name>
<value>%hostname%-audit.log</value>
</property>
<proeprty>
```

```
<name>xasecure.audit.hdfs.config.destination.flush.interval.seconds</name>
<value>900</value>
</proeprty>
<property>
<name>xasecure.audit.hdfs.config.destination.rollover.interval.seconds</name>
<value>86400</value>
</property>
<property>
<name>xasecure.audit.hdfs.config.destination.open.retry.interval.seconds</name>
<value>60</value>
</property>
<property>
<name>xasecure.audit.hdfs.config.local.buffer.directory</name>
<value>/var/log/hadoop/%app-type%/audit</value>
</property>
<property>
<name>xasecure.audit.hdfs.config.local.buffer.file</name>
<value>%time:yyyyMMdd-HHmm.ss%.log</value>
</property>
<property>
<name>xasecure.audit.hdfs.config.local.buffer.file.buffer.size.bytes</name>
<value>8192</value>
</property>
<property>
<name>xasecure.audit.hdfs.config.local.buffer.flush.interval.seconds</name>
<value>60</value>
</property>
<property>
<name>xasecure.audit.hdfs.config.local.buffer.rollover.interval.seconds</name>
<value>600</value>
</property>
<property>
<name>xasecure.audit.hdfs.config.local.archive.directory</name>
<value>/var/log/hadoop/%app-type%/audit/archive</value>
</property>
<property>
<name>xasecure.audit.hdfs.config.local.archive.max.file.count</name>
<value>10</value>
</property>
<!-- log4j audit provider config -->
<property>
<name>xasecure.audit.log4j.is.enabled</name>
<value>false</value>
</property>
<property>
<name>xasecure.audit.log4j.is.async</name>
<value>false</value>
```

```
</property>
<property>
<name>xasecure.audit.log4j.async.max.queue.size</name>
<value>10240</value>
</property>
<property>
<name>xasecure.audit.log4j.async.max.flush.interval.ms</name>
<value>30000</value>
</property>
<!-- kafka audit provider config -->
<property>
<name>xasecure.audit.kafka.is.enabled</name>
<value>false</value>
</property>
<property>
<name>xasecure.audit.kafka.async.max.queue.size</name>
<value>1</value>
</property>
<property>
<name>xasecure.audit.kafka.async.max.flush.interval.ms</name>
<value>1000</value>
</property>
<property>
<name>xasecure.audit.kafka.broker_list</name>
<value>localhost:9092</value>
</property>
<property>
<name>xasecure.audit.kafka.topic_name</name>
<value>ranger_audits</value>
</property>
<!-- ranger audit solr config -->
<property>
<name>xasecure.audit.solr.is.enabled</name>
<value>false</value>
</property>
<property>
<name>xasecure.audit.solr.async.max.queue.size</name>
<value>1</value>
</property>
<property>
<name>xasecure.audit.solr.async.max.flush.interval.ms</name>
<value>1000</value>
</property>
<property>
<name>xasecure.audit.solr.solr_url</name>
<value>http://localhost:6083/solr/ranger_audits</value>
```

```
</property>
</configuration>
```

- ranger-policymgr-ssl.xml

```
<configuration>
<property>
<name>xasecure.policymgr.clientssl.keystore</name>
<value>hadoopdev-clientcert.jks</value>
</property>
<property>
<name>xasecure.policymgr.clientssl.truststore</name>
<value>cacerts-xasecure.jks</value>
</property>
<property>
<name>xasecure.policymgr.clientssl.keystore.credential.file</name>
<value>jceks://file/tmp/keystore-hadoopdev-ssl.jceks</value>
</property>
<property>
<name>xasecure.policymgr.clientssl.truststore.credential.file</name>
<value>jceks://file/tmp/truststore-hadoopdev-ssl.jceks</value>
</property>
</configuration>
```

3. 在 goosefs-site.properties 文件中，添加如下配置：

```
...
goosefs.security.authorization.permission.type=CUSTOM
goosefs.security.authorization.custom.provider.class=org.apache.ranger.authoriz
ation.goosefs.RangerGooseFSAuthorizer
...
```

4. 在 ${GOOSEFS_HOME}/libexec/goosefs-config.sh 中，将 goosefs-ranger-plugin-${version}.jar 添加到 GooseFS 的类路径中：

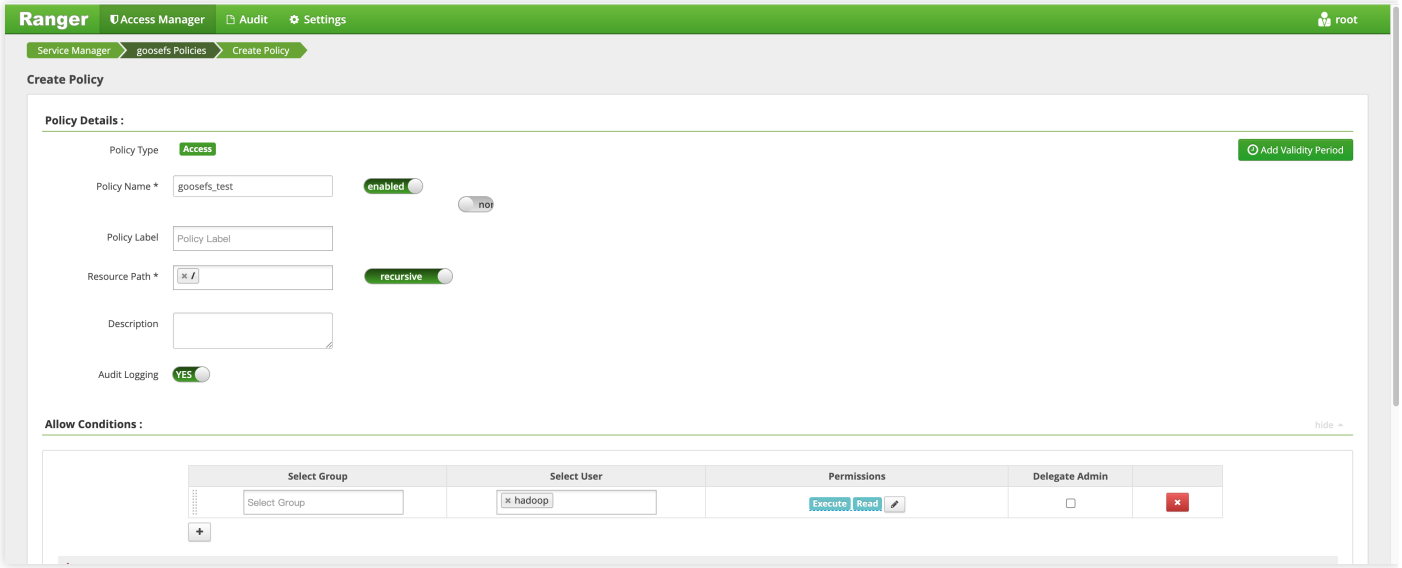
```
...
GOOSEFS_RANGER_CLASSPATH="${GOOSEFS_HOME}/lib/ranger-goosefs-plugin-${version}.
jar"
GOOSEFS_SERVER_CLASSPATH=${GOOSEFS_SERVER_CLASSPATH}:${GOOSEFS_RANGER_CLASSPAT
H}
...
```

至此，所有配置完成。

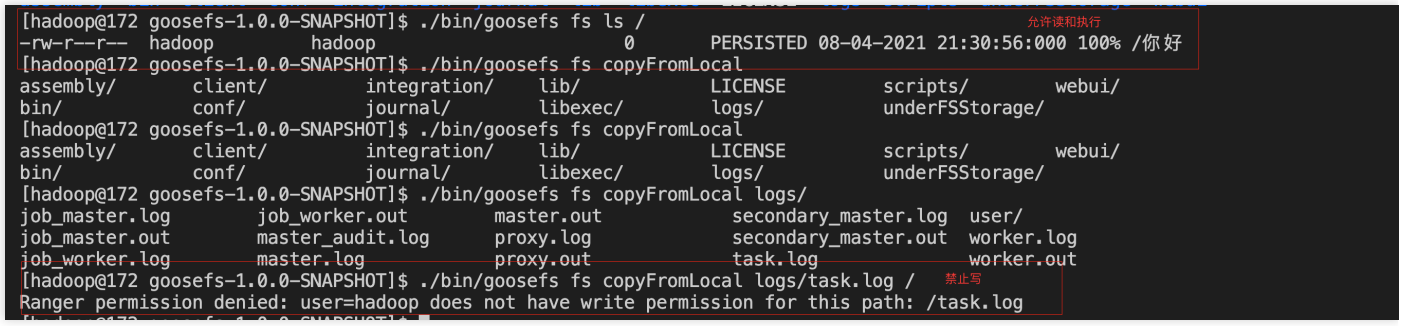
# 验证使用

例如，添加一条允许 hadoop 用户对 GooseFS 的根目录具备读取和执行权限，但是不允许写的策略，方法如下：

1. 添加策略，如下所示：




2. 策略添加成功后，对策略进行验证，即可看到策略已经生效，如下所示：



# GooseFS on TKE 云原生实践

# 入门

# 安装

最近更新时间：2021-11-18 12:47:30

## 安装使用流程

### 1. 创建命名空间

```
$ kubectl create ns fluid-system
```

### 2. 下载 fluid

下载 fluid-0.6.0.tgz 安装包。

> 注意：
> 您也可以前往 Fluid 官方页面 Fluid Releases 中下载最新版本，但一些机器从中国境内访问时可能存在网络问题。

### 3. 使用 Helm 安装 Fluid

```
$ helm install --set runtime.goosefs.enabled=true fluid fluid-0.6.0.tgz
```

### 4. 查看 Fluid 的运行状态

```
$ kubectl get pod -n fluid-system
NAME READY STATUS RESTARTS AGE
csi-nodeplugin-fluid-2mfcr 2/2 Running 0 108s
csi-nodeplugin-fluid-l7lv6 2/2 Running 0 108s
dataset-controller-5465c4bbf9-5ds5p 1/1 Running 0 108s
goosefsruntime-controller-654fb74447-cldsv 1/1 Running 0 108s
```

其中，csi-nodeplugin-fluid-xx 的数量应该与 k8s 集群中节点 node 的数量相同。

到此 Fluid 已成功安装，如需自定义镜像和升级系统 crd 请参考如下说明（可选）。

### 5. 自定义镜像

解压 `fluid-0.6.0.tgz`，修改默认 `values.yaml` 文件：

```
runtime:
mountRoot: /runtime-mnt
goosefs:
runtimeWorkers: 3
portRange: 26000-32000
enabled: false
init:
image: fluidcloudnative/init-users:v0.6.0-116a5be
controller:
image: fluidcloudnative/goosefsruntime-controller:v0.6.0-116a5be
runtime:
image: ccr.ccs.tencentyun.com/goosefs/goosefs:v1.0.1
fuse:
image: ccr.ccs.tencentyun.com/goosefs/goosefs-fuse:v1.0.1
```

您可以修改 `goosefs` 相关的默认 `image` 内容，如放到自己的 `repo` 上。待修改完成后，重新使用 `helm package fluid` 打包，并使用如下命令，更新 `fluid` 版本。

```
helm upgrade --install fluid fluid-0.6.0.tgz
```

### 6. 更新 crd

```
$ kubectl get crd
NAME CREATED AT
databackups.data.fluid.io 2021-03-02T13:12:31Z
dataloads.data.fluid.io 2021-04-14T11:14:58Z
datasets.data.fluid.io 2021-03-02T13:12:31Z
goosefsruntimes.data.fluid.io 2021-04-13T13:31:38Z
```

例如，更新系统中已有的 `goosefsruntime` 的 crd。

首先删除已有的 crd：

```
kubectl delete crd goosefsruntimes.data.fluid.io
```

然后解压 `fluid-0.6.0.tgz` ：

```
$ ls -l fluid/
total 32
total 32
-rw-r--r-- 1 xieydd staff 489 5 15 16:14 CHANGELOG.md
```

```
-rw-r--r-- 1 xieydd staff 1061 7 22 00:08 Chart.yaml
-rw-r--r-- 1 xieydd staff 2560 5 15 16:14 VERSION
drwxr-xr-x 8 xieydd staff 256 7 20 15:06 crds
drwxr-xr-x 7 xieydd staff 224 5 24 14:18 templates
-rw-r--r-- 1 xieydd staff 1665 7 22 00:08 values.yaml
```

最后创建新的 crd：

```
kubectl apply -f crds/data.fluid.io_goosefsruntimes.yaml
```

# 快速入门

最近更新时间：2022-01-17 12:31:17

## 快速使用 GooseFSRuntime

使用 GooseFSRuntime 流程简单，在准备完基本 k8s 和对象存储（Cloud Object Storage，COS）环境的条件下，您只需要耗费10分钟左右即可部署完成所需的 GooseFSRuntime 环境，您可以按照下面的流程进行部署。

## 前提条件

- 已安装 Git。
- 已安装 Kubernetes 集群（version >= 1.14），并且支持 CSI 功能。为获得最佳实践，您可以直接在 腾讯云容器服务（Tencent Kubernetes Engine，TKE）部署。
- 已安装 kubectl（version >= 1.14）。关于 `kubectl` 安装和配置详情，请参见 Kubernetes 文档。
- 已安装 Helm（version >= 3.0）。关于 Helm 3 安装和配置详情，请参见 HELM 文档。
- 如下提供了本地安装包 `fluid.tgz`，您可以直接安装使用：

```
$ helm install fluid fluid.tgz
NAME: fluid
LAST DEPLOYED: Mon Mar 29 11:21:46 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

说明：
`helm install` 命令的一般格式是 `helm install <release_name> <source>`，在上面的命令中，第一个 `fluid` 指定了安装的 release 名称（可自行更改），第二个 `fluid.tgz` 指定了helm chart 所在的路径。

## 操作步骤

### 1. 创建命名空间

```
kubectl create ns fluid-system
```

### 2. 下载 fluid

单击下载 fluid-0.6.0.tgz 安装包。

> 注意：
> 您也可以前往 Fluid Releases 下载最新版本，但部分机器从中国境内访问该网站可能存在网络问题。

### 3. 使用 Helm 安装 Fluid

```
helm install --set runtime.goosefs.enabled=true fluid fluid-0.6.0.tgz
```

### 4. 查看 Fluid 的运行状态

```
$ kubectl get pod -n fluid-system
NAME READY STATUS RESTARTS AGE
csi-nodeplugin-fluid-2mfcr 2/2 Running 0 108s
csi-nodeplugin-fluid-l7lv6 2/2 Running 0 108s
dataset-controller-5465c4bbf9-5ds5p 1/1 Running 0 108s
goosefsruntime-controller-564f59bdd7-49tkc 1/1 Running 0 108s
```

其中，csi-nodeplugin-fluid-xx 的数量应与 k8s 集群中节点 node 的数量相同。

### 5. GooseFS 缓存环境准备

#### （1）准备 COS 服务

您需要开通 COS，并完成存储桶的创建，创建指引可参见 创建存储桶。

#### （2）准备测试样例数据

我们可以使用 Apache 镜像站点上的 Spark 相关资源文件作为演示中使用的远程文件。在实际使用当中，您也可以将这个远程文件修改为您的任意远程文件。

- 下载远程资源文件到本地

```
mkdir tmp
cd tmp
wget https://mirrors.tuna.tsinghua.edu.cn/apache/spark/spark-2.4.8/spark-2.4.8-bin-hadoop2.7.tgz
```

```
wget https://mirrors.tuna.tsinghua.edu.cn/apache/spark/spark-3.1.2/spark-3.1.2-bin-hadoop3.2.tgz
```

- 上传本地文件到 COS 上
  您可以使用腾讯云 COS 团队提供的客户端 COSCMD 或者腾讯云 COS 团队提供的 COS SDK，按照使用说明将本地下载的文件上传到 COS 的存储桶上。

### （3）创建 dataset 和 GooseFSRuntime

i. 创建一个 resource.yaml 文件，里面包含如下内容：

- 包含数据集及 ufs 的 dataset 信息。
- 创建一个 Dataset CRD 对象，描述了数据集的来源，例如示例中的 test-bucket。
- 创建一个 GooseFSRuntime，相当于启动一个 GooseFS 的集群用于提供缓存服务。

```
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: hadoop
spec:
mounts:
- mountPoint: cosn://test-bucket/
options:
fs.cosn.userinfo.secretId: <cos_secret_id>
fs.cosn.userinfo.secretKey: <cos_secret_key>
fs.cosn.bucket.region: <cos_region>
fs.cosn.impl: org.apache.hadoop.fs.CosFileSystem
fs.AbstractFileSystem.cosn.impl: org.apache.hadoop.fs.CosN
fs.cosn.userinfo.appid: <cos_app_id>
name: hadoop

---
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: hadoop
spec:
replicas: 2
tieredstore:
levels:
- mediumtype: HDD
path: /mnt/disk1
quota: 100G
```

```
high: "0.9"
low: "0.8"
```

- Dataset：
  - mountPoint：表示挂载 UFS 的路径，路径中不需要包含 endpoint 信息。
  - options：在 options 需要指定存储桶的必要信息，具体可参考 API 术语信息。
  - fs.cosn.userinfo.secretId/fs.cosn.userinfo.secretKey：拥有权限访问该 COS 存储桶的密钥信息。
- GooseFSRuntime：更多 API 可参考 api_doc.md。
  - replicas：表示创建 GooseFS 集群节点的数量。
  - mediumtype： GooseFS 支持 HDD/SSD/MEM 三种类型缓存介质，提供多级缓存配置。
  - path：存储路径。
  - quota：缓存最大容量。
  - high：水位上限大小。
  - low：水位下限大小。

ii. 执行如下命令，创建 GooseFSRuntime：

```
kubectl create -f resource.yaml
```

iii. 查看部署的 GooseFSRuntime 情况，显示全部为 Ready 状态表示部署成功。

```
kubectl get goosefsruntime hadoop
NAME MASTER PHASE WORKER PHASE FUSE PHASE AGE
hadoop Ready Ready Ready 62m
```

iv. 查看 dataset 的情况，显示 Bound 状态表示 dataset 绑定成功。

```
$ kubectl get dataset hadoop
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
hadoop 511MiB 0.00B 180.00GiB 0.0% Bound 1h
```

v. 查看 PV、PVC 创建情况，GooseFSRuntime 部署过程中会自动创建 PV 和 PVC。

```
kubectl get pv,pvc
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE
persistentvolume/hadoop 100Gi RWX Retain Bound default/hadoop 58m

NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE
persistentvolumeclaim/hadoop Bound hadoop 100Gi RWX 58m
```

### 6. 创建应用容器体验加速效果

您可以通过创建应用容器来使用 GooseFS 加速服务，或者进行提交机器学习作业来进行体验相关功能。

如下，创建一个应用容器 app.yaml 用于使用该数据集。我们将多次访问同一数据，并比较访问时间来展示 GooseFS 的加速效果。

```
apiVersion: v1
kind: Pod
metadata:
name: demo-app
spec:
containers:
- name: demo
image: nginx
volumeMounts:
- mountPath: /data
name: hadoop
volumes:
- name: hadoop
persistentVolumeClaim:
claimName: hadoop
```

i. 使用 kubectl 完成创建应用：

```
kubectl create -f app.yaml
```

ii. 查看文件大小：

```
$ kubectl exec -it demo-app -- bash
$ du -sh /data/hadoop/spark/spark-3.1.2/spark-3.1.2-bin-hadoop3.2
210M /data/hadoop/spark/spark-3.1.2/spark-3.1.2-bin-hadoop3.2
```

iii. 进行文件的 cp 观察时间消耗了18s：

```
$ time cp /data/hadoop/spark/spark-3.1.2/spark-3.1.2-bin-hadoop3.2 /dev/null
real 0m18.386s
user 0m0.002s
sys 0m0.105s
```

iv. 查看此时 dataset 的缓存情况，发现210MB的数据已经都缓存到了本地。

```
$ kubectl get dataset hadoop
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
hadoop 210.00MiB 210.00MiB 180.00GiB 100.0% Bound 1h
```

v. 为了避免其他因素（例如 page cache）对结果造成影响，我们将删除之前的容器，新建相同的应用，尝试访问同样的文件。由于此时文件已经被 GooseFS 缓存，可以看到第二次访问所需时间远小于第一次。

```
kubectl delete -f app.yaml && kubectl create -f app.yaml
```

vi. 进行文件的拷贝观察时间，发现消耗48ms，整个拷贝的时间缩短了300倍。

```
$ time cp /data/hadoop/spark/spark-3.1.2/spark-3.1.2-bin-hadoop3.2 /dev/null
real 0m0.048s
user 0m0.001s
sys 0m0.046s
```

### 7. 环境清理

- 删除应用和应用容器
- 删除 GooseFSRuntime

  ```
  kubectl delete goosefsruntime hadoop
  ```

- 删除 dataset

  ```
  kubectl delete dataset hadoop
  ```

以上通过一个简单的例子完成 GooseFS on Fluid 的入门体验和学习，并最后进行环境的清理，更多 Fluid GooseFSRuntime 的功能介绍可参见 功能列表。

# 问题诊断和处理

最近更新时间：2021-11-18 12:47:30

## Fluid 安装使用相关问题

### 为什么我使用 Helm 安装 fluid 失败了？

推荐按照 Fluid 安装文档 依次确认 Fluid 组件是否正常运行。

Fluid 安装文档是以 `Helm 3` 为例进行部署的。如果您使用 `Helm 3` 以下的版本部署 Fluid，并且遇到了 `CRD没有正常启动` 的情况，这可能是因为 `Helm 3` 及其以上版本会在 `helm install` 的时候自动安装 CRD，而低版本的 Helm 则不会。详情请参见 Helm 官方文档。

在这种情况下，您需要手动安装 CRD：

```
$ kubectl create -f fluid/crds
```

### 为什么我无法删除 Runtime？

请检查相关 Pod 运行状态和 Runtime 的 Events。只要有任何活跃 Pod 还在使用 Fluid 创建的 Volume，Fluid 就不会完成删除操作。

如下命令可以快速地找出这些活跃 Pod，使用时把 `<dataset_name>` 和 `<dataset_namespace>` 换成自己的即可：

```
kubectl describe pvc <dataset_name> -n <dataset_namespace> | \
awk '/^Mounted/ {flag=1}; /^Events/ {flag=0}; flag' | \
awk 'NR==1 {print $3}; NR!=1 {print $1}' | \
xargs -I {} kubectl get po {} | \
grep -E "Running|Terminating|Pending" | \
cut -d " " -f 1
```

### 为什么我在创建任务挂载 Runtime 创建的 PVC 时，出现报错 `driver name fuse.csi.fluid.io not found in the list of registered CSI drivers` ？

1. 请查看任务被调度节点所在的 kubelet 配置是否为默认 `/var/lib/kubelet` 。

2. 通过命令查看 Fluid 的 CSI 组件是否正常。
   如下命令可以快速地找出 Pod，使用时把 `<node_name>` 和 `<fluid_namespace>` 换成自己的即可：

```
kubectl get pod -n <fluid_namespace> | grep <node_name>
# <pod_name> 为上一步pod名
kubectl logs <pod_name> node-driver-registrar -n <fluid_namespace>
kubectl logs <pod_name> plugins -n <fluid_namespace>
```

- 如果上述步骤的 Log 无错误，请查看 csidriver 对象是否存在：

  ```
  kubectl get csidriver
  ```

- 如果 csidriver 对象存在，请查看 csi 注册节点是否包含 `<node_name>` ：

  ```
  kubectl get csinode | grep <node_name>
  ```

如果上述命令无输出，请查看任务被调度节点所在的 kubelet 配置是否为默认 `/var/lib/kubelet` 。由于 Fluid 的 CSI 组件通过固定地址的 socket 注册到 kubelet，因此默认为 `--csi-address=/var/lib/kubelet/csi-plugins/fuse.csi.fluid.io/csi.sock --kubelet-registration-path=/var/lib/kubelet/csi-plugins/fuse.csi.fluid.io/csi.sock` 。

### 为什么更新了 fluid 后，使用 `kubectl get` 查询更新前创建的dataset，发现相比新建的dataset缺少了某些字段？

由于我们在 fluid 的升级过程中可能更新了 CRD，您在旧版本创建的 dataset，会将 CRD 中新增的字段设置为空。例如，您从v0.4或更早版本升级，那时的 dataset 没有 FileNum 字段，更新 fluid 后，如果您使用 `kubectl get` 命令，无法查询到该 dataset 的 FileNum。

您可以重建 dataset，新建的 dataset 会正常显示这些字段。

### 为什么在应用程序中使用 PVC 时会产生了 Volume Attachment 超时问题？

Volume Attachment 超时问题是 Kubelet 进行请求 CSI Driver 时未收到 CSI Driver 的响应而造成的超时。该问题是由于 Fluid 的 CSI Driver 没有安装，或者 kubelet 没有访问 CSI Driver 的权限导致的。

由于 CSI Driver 是由 Kubelet 进行回调，所以如果 Fluid 没有安装 CSI Driver 或者 Kubelet 没有权限查看 CSI Driver，就会导致 CSI Plugin 没有被正确触发。

使用命令 `kubectl get csidriver` 查看是否安装了 CSI Driver。

- 如果没有安装，使用命令 `kubectl apply -f charts/fluid/fluid/templates/csi/driver.yaml` 进行安装，然后观察 PVC 是否成功挂载到应用程序中。
- 如果仍未解决，使用 `export KUBECONFIG=/etc/kubernetes/kubelet.conf &amp;&amp; kubectl get csidriver` 来查看 Kubelet 能够具有权限看到 CSI Driver。

# 使用 GooseFS 作为 Fluid Dataset Runtime Master 节点高可用部署模式

最近更新时间：2022-05-26 15:46:29

## 概述

单个 master 的稳定性可能较低，在一些线上提供服务的场景，需要使用多 master 来保证容错率。GooseFSRuntime 提供以 3 master 的形式来提供容错，通过 Raft 协议来进行选举，raft 是工程上使用较为广泛的强一致性、去中心化、高可用的分布式协议。

下面将为您简单地介绍上述特性。

## 前提条件

在运行该示例之前，请参考 安装 文档完成安装，并检查 Fluid 各组件正常运行：

```
$ kubectl get pod -n fluid-system
goosefsruntime-controller-5b64fdbbb-84pc6 1/1 Running 0 8h
csi-nodeplugin-fluid-fwgjh 2/2 Running 0 8h
csi-nodeplugin-fluid-ll8bq 2/2 Running 0 8h
csi-nodeplugin-fluid-dhz7d 2/2 Running 0 8h
dataset-controller-5b7848dbbb-n44dj 1/1 Running 0 8h
```

通常来说，您会看到一个名为 `dataset-controller` 的 Pod、一个名为 `goosefsruntime-controller` 的 Pod 和多个名为 `csi-nodeplugin` 的 Pod 正在运行。其中， `csi-nodeplugin` 这些 Pod 的数量取决于您的 Kubernetes 集群中结点的数量。

## 新建工作环境

```
$ mkdir <any-path>/co-locality
$ cd <any-path>/co-locality
```

## 示例

### 查看全部结点

```
$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
192.168.1.145 Ready <none> 7d14h v1.18.4-tke.13
192.168.1.146 Ready <none> 7d14h v1.18.4-tke.13
192.168.1.147 Ready <none> 7d14h v1.18.4-tke.13
```

### 检查待创建的 Dataset 资源对象

```
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: hbase
spec:
mounts:
- mountPoint: https://mirrors.tuna.tsinghua.edu.cn/apache/hbase/stable/
name: hbase
```

说明：
为了方便用户进行测试，mountPoint 这里使用的是 Web UFS，使用 COS 作为 UFS 可参见 使用 GooseFS 挂载 COS（COSN）。

### 创建 Dataset 资源对象

```
$ kubectl create -f dataset.yaml
dataset.data.fluid.io/hbase created
```

### 检查待创建的 GooseFSRuntime 资源对象

```
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: hbase
spec:
replicas: 3
tieredstore:
levels:
- mediumtype: HDD
path: /mnt/disk1
quota: 2G
```

```
high: "0.8"
low: "0.7"
master:
replicas: 3
```

我们通过指定 `spec.master.replicas=3` 来开启 Raft 3 master 模式，该参数必须为正奇数。

### 创建GooseFSRuntime资源并查看状态

```
$ kubectl create -f runtime.yaml
goosefsruntime.data.fluid.io/hbase created

$ kubectl get pod
NAME READY STATUS RESTARTS AGE
hbase-fuse-4v9mq 1/1 Running 0 84s
hbase-fuse-5kjbj 1/1 Running 0 84s
hbase-fuse-tp2q2 1/1 Running 0 84s
hbase-master-0 1/1 Running 0 104s
hbase-master-1 1/1 Running 0 102s
hbase-master-2 1/1 Running 0 100s
hbase-worker-cx8x7 1/1 Running 0 84s
hbase-worker-fjsr6 1/1 Running 0 84s
hbase-worker-fvpgc 1/1 Running 0 84s
```

### 查看 GooseFSRuntime 状态

```
NAME MASTER PHASE WORKER PHASE FUSE PHASE AGE
hbase Ready Ready Ready 15m
```

确认 `PHASE` 全部为 Ready。

### 查看 Raft 状态 leader/follower

登录其中一个 master 的 pod：

```
$ kubectl exec -ti hbase-master-0 bash
$ goosefs fs masterInfo
```

可以看到其中一个节点是 `LEADER`，其余两个是 `FOLLOWER`：

```
current leader master: hbase-master-0:26000
All masters: [hbase-master-0:26000, hbase-master-1:26000, hbase-master-2:26000]
```

# 多 master 节点亲和性调度

最近更新时间：2021-11-18 12:47:31

## 概述

在 Fluid 中可以通过 nodeselector 来指定 master 节点的部署，例如选择部署 master 到性能较好的 k8s 机器上。

下面将向您简单地介绍上述特性。

## 前提条件

在运行该示例之前，请参考 安装 文档完成安装，并检查 Fluid 各组件正常运行：

```
$ kubectl get pod -n fluid-system
goosefsruntime-controller-5b64fdbbb-84pc6 1/1 Running 0 8h
csi-nodeplugin-fluid-fwgjh 2/2 Running 0 8h
csi-nodeplugin-fluid-ll8bq 2/2 Running 0 8h
dataset-controller-5b7848dbbb-n44dj 1/1 Running 0 8h
```

通常来说，您会看到一个名为 `dataset-controller` 的 Pod、一个名为 `goosefsruntime-controller` 的 Pod 和多个名为 `csi-nodeplugin` 的 Pod 正在运行。其中 `csi-nodeplugin` 这些 Pod 的数量取决于您的 Kubernetes 集群中结点的数量。

## 新建工作环境

```
$ mkdir <any-path>/co-locality
$ cd <any-path>/co-locality
```

## 示例

### 查看全部结点

```
$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
192.168.1.146 Ready <none> 7d14h v1.18.4-tke.13
192.168.1.147 Ready <none> 7d14h v1.18.4-tke.13
```

### 使用标签标识结点

```
$ kubectl label nodes 192.168.1.146 hbase-cache=true
```

### 再次查看结点

```
$ kubectl get node -L hbase-cache
NAME STATUS ROLES AGE VERSION HBASE-CACHE
192.168.1.146 Ready <none> 7d14h v1.18.4-tke.13 true
192.168.1.147 Ready <none> 7d14h v1.18.4-tke.13
```

### 检查待创建的 Dataset 资源对象

```
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: hbase
spec:
mounts:
- mountPoint: https://mirrors.tuna.tsinghua.edu.cn/apache/hbase/stable/
name: hbase
```

> 说明：
> 为了方便用户进行测试，mountPoint 这里使用的是 Web UFS，使用 COS 作为 UFS 可参见 使用 GooseFS 挂载 COS（COSN）。

### 创建 Dataset 资源对象

```
$ kubectl create -f dataset.yaml
dataset.data.fluid.io/hbase created
```

### 检查待创建的 GooseFSRuntime 资源对象

```
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: hbase
spec:
```

```
replicas: 1
tieredstore:
levels:
- mediumtype: MEM
path: /dev/shm
quota: 2G
high: "0.8"
low: "0.7"
master:
nodeSelector:
hbase-cache: "true"
```

该配置文件片段中，包含了许多与 GooseFS 相关的配置信息，这些信息将被 Fluid 用来启动一个 GooseFS 实例。上述配置片段中的 `spec.replicas` 属性被设置为1，这表明 Fluid 将会启动一个包含1个 GooseFS Master 和1个 GooseFS Worker 的 GooseFS 实例。

### 创建 GooseFSRuntime 资源并查看状态

```
$ kubectl create -f runtime.yaml
goosefsruntime.data.fluid.io/hbase created

$ kubectl get pod -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
hbase-fuse-42csf 1/1 Running 0 104s 192.168.1.146 192.168.1.146 <none> <none>
hbase-master-0 2/2 Running 0 3m3s 192.168.1.147 192.168.1.146 <none> <none>
hbase-worker-l62m4 2/2 Running 0 104s 192.168.1.146 192.168.1.146 <none> <none>
```

在此处可以看到，master 成功启动并且运行在具有指定标签（即 `hbase-cache=true` ）的结点之上。

# 加速 COS 上的数据

最近更新时间：2022-09-16 10:05:19

## 前提条件

- 已下载安装 Fluid（version >= 0.6.0）

> 注意：
> 单击下载 fluid-0.6.0.tgz 安装包。

- 请参见 安装 文档完成 Fluid 安装。

## 创建 Dataset 和 GooseFSRuntime

1. 创建一个 resource.yaml 文件，里面包含如下内容：

- 包含数据集及 ufs 的 dataset 信息。
- 创建一个 Dataset CRD 对象，描述了数据集的来源，例如示例中的 test-bucket。
- 创建一个 GooseFSRuntime，相当于启动一个 GooseFS 的集群来提供缓存服务。

```
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: hadoop
spec:
mounts:
- mountPoint: cosn://test-bucket/
options:
fs.cosn.userinfo.secretId: <COS_SECRET_ID>
fs.cosn.userinfo.secretKey: <COS_SECRET_KEY>
fs.cosn.bucket.region: <COS_REGION>
fs.cosn.impl: org.apache.hadoop.fs.CosFileSystem
fs.AbstractFileSystem.cosn.impl: org.apache.hadoop.fs.CosN
fs.cosn.userinfo.appid: <COS_APP_ID>
name: hadoop
```

```
---
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: hadoop
spec:
replicas: 2
tieredstore:
levels:
- mediumtype: HDD
path: /mnt/disk1
quota: 100G
high: "0.9"
low: "0.2"
```

为了 AK 等密钥信息的安全性，建议使用 secret 来保存相关密钥信息，secret 使用请参考 使用参数加密。

```
apiVersion: v1
kind: Secret
metadata:
name: mysecret
stringData:
fs.cosn.userinfo.secretId: <COS_SECRET_ID>
fs.cosn.userinfo.secretKey: <COS_SECRET_KEY>
---
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: hadoop
spec:
mounts:
- mountPoint: cosn://yourbucket/
options:
fs.cosn.bucket.region: <COS_REGION>
fs.cosn.impl: org.apache.hadoop.fs.CosFileSystem
fs.AbstractFileSystem.cosn.impl: org.apache.hadoop.fs.CosN
fs.cosn.userinfo.appid: <COS_APP_ID>
name: hadoop
encryptOptions:
- name: fs.cosn.userinfo.secretId
valueFrom:
secretKeyRef:
name: mysecret
key: fs.cosn.userinfo.secretId
- name: fs.cosn.userinfo.secretKey
```

```
valueFrom:
secretKeyRef:
name: mysecret
key: fs.cosn.userinfo.secretKey
---
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: hadoop
spec:
replicas: 2
tieredstore:
levels:
- mediumtype: SSD
path: /mnt/disk1
quota: 100G
high: "0.9"
low: "0.2"
```

- Dataset：
  - mountPoint：表示挂载 UFS 的路径，路径中不需要包含 endpoint 信息。
  - options：在 options 需要指定存储桶的必要信息，具体可参考 API 术语信息。
  - fs.cosn.userinfo.secretId/fs.cosn.userinfo.secretKey：拥有权限访问该 COS 存储桶的密钥信息。
- GooseFSRuntime：更多 API 可参考 api_doc.md。
  - replicas：表示创建 GooseFS 集群节点的数量。
  - mediumtype： GooseFS 支持 HDD/SSD/MEM 三种类型缓存介质，提供多级缓存配置。
  - path：存储路径。
  - quota：缓存最大容量。
  - high：水位上限大小。
  - low：水位下限大小。

2. 执行命令，创建 GooseFSRuntime：

```
$ kubectl create -f resource.yaml
```

3. 查看部署的 GooseFSRuntime 情况，显示全部为 Ready 状态表示部署成功。

```
$ kubectl get goosefsruntime hadoop
NAME MASTER PHASE WORKER PHASE FUSE PHASE AGE
hadoop Ready Ready Ready 62m
```

4. 查看 dataset 的情况，显示 Bound 状态表示 dataset 绑定成功。

```
$ kubectl get dataset hadoop
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
hadoop 210.00MiB 0.00B 180.00GiB 0.0% Bound 1h
```

5. 查看 PV、PVC 创建情况，GooseFSRuntime 部署过程中会自动创建 PV 和 PVC。

```
$ kubectl get pv,pvc
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE
persistentvolume/hadoop 100Gi RWX Retain Bound default/hadoop 58m
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE
persistentvolumeclaim/hadoop Bound hadoop 100Gi RWX 58m
```

## 检查服务是否正常

1. 登录到 master/worker pod 上。观察是否可以正常 list 文件。

```
$ kubectl get pod
NAME READY STATUS RESTARTS AGE
hadoop-fuse-svz4s 1/1 Running 0 23h
hadoop-master-0 1/1 Running 0 23h
hadoop-worker-2fpbk 1/1 Running 0 23h
```

```
$ kubectl exec -ti hadoop-goosefs-master-0 bash
goosefs fs ls /hadoop
```

2. 登录到 fuse pod 上。观察是否可以正常 list 文件。

```
$ kubectl exec -ti hadoop-goosefs-fuse-svz4s bash
cd /runtime-mnt/goosefs/<namespace>/<DatasetName>/goosefs-fuse/<DatasetName>
```

## 创建应用容器体验加速效果

您可以通过创建应用容器来使用 GooseFS 加速服务，或者提交机器学习作业来进行体验相关功能。如下，创建一个应用容器 app.yaml 用于使用该数据集。我们将多次访问同一数据，并比较访问时间来展示 GooseFSRuntime 的加速效果。

```
apiVersion: v1
kind: Pod
metadata:
name: demo-app
spec:
containers:
- name: demo
image: nginx
volumeMounts:
- mountPath: /data
name: hadoop
volumes:
- name: hadoop
persistentVolumeClaim:
claimName: hadoop
```

1. 使用 kubectl 完成创建应用。

```
$ kubectl create -f app.yaml
```

2. 查看文件大小。

```
$ kubectl exec -it demo-app -- bash
$ du -sh /data/hadoop/spark/spark-3.1.2/spark-3.1.2-bin-hadoop3.2
210M /data/hadoop/spark/spark-3.1.2/spark-3.1.2-bin-hadoop3.2
```

3. 进行文件的 cp 观察时间消耗了18s：

```
$ time cp /data/hadoop/spark/spark-3.1.2/spark-3.1.2-bin-hadoop3.2 /dev/null
real 0m18.386s
user 0m0.002s
sys 0m0.105s
```

4. 查看此时 dataset 的缓存情况，发现210MB的数据已经都缓存到了本地。

```
$ kubectl get dataset hadoop
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
hadoop 210.00MiB 210.00MiB 180.00GiB 100.0% Bound 1h
```

5. 为了避免其他因素（例如 page cache）对结果造成影响，我们将删除之前的容器，新建相同的应用，尝试访问同样的文件。由于此时文件已经被 GooseFS 缓存，可以看到第二次访问所需时间远小于第一次。

```
$ kubectl delete -f app.yaml && kubectl create -f app.yaml
```

6. 进行文件的拷贝观察时间，发现消耗48ms，整个拷贝的时间缩短了300倍。

```
$ time cp /data/hadoop/spark/spark-3.1.2/spark-3.1.2-bin-hadoop3.2 /dev/null
real 0m0.048s
user 0m0.001s
sys 0m0.046s
```

## 清理环境

```
$ kubectl delete -f resource.yaml
```

# 客户端全局部署

最近更新时间：2021-11-18 12:47:32

在 Fluid 中， `Dataset` 资源对象中所定义的远程文件是可被调度的，这意味着您能够像管理您的 Pod 一样管理远程文件缓存在 Kubernetes 集群上的存放位置。而执行计算的 Pod 可以通过 Fuse 客户端访问数据文件。

Fuse 客户端提供两种模式：

- global 为 false，该模式为 Fuse 客户端和缓存数据强制亲和性，此时 Fuse 客户端的数量等于 Runtime 的 replicas 数量。此配置默认模式，无需显式声明，好处是可以发挥数据的亲和性优点，但是 Fuse 客户端的部署就变得比较固定。
- global 为 true，该模式为 Fuse 客户端，可以在 Kubernetes 集群中全局部署，并不要求数据和 Fuse 客户端之间的强制亲和性，此时 Fuse 客户端的数量可能远超 Runtime 的 replicas 数量。建议此时可以通过 nodeSelector 来指定 Fuse 客户端的部署范围。

## 前提条件

在运行该示例之前，请参考 安装 文档完成安装，注意执行 helm 命令加上参数 `--set webhook.enable=true` 开启 `webhook` ，并检查 Fluid 各组件正常运行：

```
$ kubectl get pod -n fluid-system
goosefsruntime-controller-5b64fdbbb-84pc6 1/1 Running 0 8h
csi-nodeplugin-fluid-fwgjh 2/2 Running 0 8h
csi-nodeplugin-fluid-ll8bq 2/2 Running 0 8h
dataset-controller-5b7848dbbb-n44dj 1/1 Running 0 8h
```

通常来说，您会看到一个名为 `dataset-controller` 的 Pod、一个名为 `goosefsruntime-controller` 的 Pod 和多个名为 `csi-nodeplugin` 的 Pod 正在运行。其中 `csi-nodeplugin` 这些 Pod 的数量取决于您的 Kubernetes 集群中结点的数量。

## 新建工作环境

```
$ mkdir <any-path>/fuse-global-deployment
$ cd <any-path>/fuse-global-deployment
```

## 运行示例

### 示例1: 设置 global 为 true

**查看全部结点**

```
$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
192.168.1.146 Ready <none> 7d14h v1.18.4-tke.13
192.168.1.147 Ready <none> 7d14h v1.18.4-tke.13
```

**使用标签标识结点**

```
$ kubectl label nodes 192.168.1.146 cache-node=true
```

在接下来的步骤中，我们将使用 `NodeSelector` 来管理集群中存放数据的位置，所以在这里标记期望的结点。

**再次查看结点**

```
$ kubectl get node -L cache-node
NAME STATUS ROLES AGE VERSION cache-node
192.168.1.146 Ready <none> 7d14h v1.18.4-tke.13 true
192.168.1.147 Ready <none> 7d14h v1.18.4-tke.13
```

目前，在全部2个结点中，仅有一个结点添加了 `cache-node=true` 的标签，接下来，我们希望数据缓存仅会被放置在该结点之上。

**检查待创建的 Dataset 资源对象**

```
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: hbase
spec:
mounts:
- mountPoint: https://mirrors.tuna.tsinghua.edu.cn/apache/hbase/stable/
name: hbase
nodeAffinity:
required:
nodeSelectorTerms:
- matchExpressions:
- key: cache-node
operator: In
values:
- "true"
```

> 说明：
> mountPoint 这里为了方便用户进行实验使用的是 Web UFS, 使用 COS 作为 UFS 可见 使用 GooseFS 挂载 COS（COSN）。

在该 `Dataset` 资源对象的 `spec` 属性中，我们定义了一个 `nodeSelectorTerm` 的子属性，该子属性要求数据缓存必须被放置在具有 `cache-node=true` 标签的结点之上。

**创建 Dataset 资源对象**

```
$ kubectl create -f dataset.yaml
dataset.data.fluid.io/hbase created
```

**检查待创建的 GooseFSRuntime 资源对象**

```
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: hbase
spec:
replicas: 1
tieredstore:
levels:
- mediumtype: SSD
path: /mnt/disk1/
quota: 2G
high: "0.8"
low: "0.7"
fuse:
global: true
```

该配置文件片段中，包含了许多与 GooseFS 相关的配置信息，这些信息将被 Fluid 用来启动一个 GooseFS 实例。上述配置片段中的 `spec.replicas` 属性被设置为1，这表明 Fluid 将会启动一个包含1个 GooseFS Master 和1个 GooseFS Worker 的 GooseFS 实例。 另外一个值得注意的是 Fuse 包含 `global: true` ，这样意味着 Fuse 可以全局部署，而不依赖于数据缓存的位置。

**创建 GooseFSRuntime 资源并查看状态**

```
$ kubectl create -f runtime.yaml
goosefsruntime.data.fluid.io/hbase created

$ kubectl get po -owide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
```

```
hbase-fuse-gfq7z 1/1 Running 0 3m47s 192.168.1.147 192.168.1.147 <none> <none>
hbase-fuse-lmk5p 1/1 Running 0 3m47s 192.168.1.146 192.168.1.146 <none> <none>
hbase-master-0 2/2 Running 0 3m47s 192.168.1.147 192.168.1.147 <none> <none>
hbase-worker-hvbp2 2/2 Running 0 3m1s 192.168.1.146 192.168.1.146 <none> <none>
```

在此处可以看到，有一个 GooseFS Worker 成功启动，并且运行在具有指定标签（即 `cache-node=true` ）的节点之上。GooseFS Fuse 的数量为2，运行在所有的子节点上。

**检查 GooseFSRuntime 状态**

```
$ kubectl get goosefsruntime hbase -o wide
NAME READY MASTERS DESIRED MASTERS MASTER PHASE READY WORKERS DESIRED WORKERS WOR
KER PHASE READY FUSES DESIRED FUSES FUSE PHASE AGE
hbase 1 1 Ready 1 1 Ready 2 2 Ready 12m
```

这里可以看到 GooseFS Worker 的数量为1，而 GooseFS Fuse 的数量为2。

**删除 GooseFSRuntime**

```
kubectl delete goosefsruntime hbase
```

### 示例2：设置 global 为 true，并且设置 fuse 的 nodeSelector

下面，我们希望通过配置 node selector 配置 Fuse 客户端，将其指定到集群中某个节点上。在本例子中，既然我们已经选择节点 192.168.1.146 作为缓存节点，为了形成对比，这里选择节点 192.168.1.147 运行 GooseFS Fuse。

```
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: hbase
spec:
replicas: 1
tieredstore:
levels:
- mediumtype: SSD
path: /mnt/disk1/
quota: 2G
high: "0.8"
low: "0.7"
fuse:
global: true
nodeSelector:
kubernetes.io/hostname: 192.168.1.147
```

该配置文件片段中，和之前 runtime.yaml 相比，在 Fuse 包含 `global: true` 的前提下, 还增加了 nodeSelector 并且指向了节点192.168.1.147。

**创建 GooseFSRuntime 资源并查看状态**

```
$ kubectl create -f runtime-node-selector.yaml
goosefsruntime.data.fluid.io/hbase created

$ kubectl get po -owide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
hbase-fuse-xzbww 1/1 Running 0 1h 192.168.1.147 192.168.1.147 <none> <none>
hbase-master-0 2/2 Running 0 1h 192.168.1.147 192.168.1.147 <none> <none>
hbase-worker-vdxd5 2/2 Running 0 1h 192.168.1.146 192.168.1.146 <none> <none>
```

在此处可以看到，有一个 GooseFS Worker 成功启动，并且运行在具有指定标签（即 `cache-node=true` ）的结点之上。GooseFS Fuse 的数量为1，运行在节点192.168.1.147上。

**检查 GooseFSRuntime 状态**

```
$ kubectl get goosefsruntimes.data.fluid.io -owide
NAME READY MASTERS DESIRED MASTERS MASTER PHASE READY WORKERS DESIRED WORKERS WOR
KER PHASE READY FUSES DESIRED FUSES FUSE PHASE AGE
hbase 1 1 Ready 1 1 Ready 1 1 Ready 1h
```

这里可以看到 GooseFS Worker 的数量为1，而 GooseFS Fuse 的数量也为1，这是因为 GooseFSRuntime 指定了 nodeSelector，并且满足条件的节点只有一个。

可见，Fluid 支持 Fuse 客户端单独的调度策略，这些调度策略为用户提供了更加灵活的 Fuse 客户端调度策略。

# 环境清理

```
$ kubectl delete -f .
$ kubectl label node 192.168.1.146 cache-node-
```

# 使用参数加密

最近更新时间：2022-01-17 12:31:17

在 Fluid 中创建 Dataset 时，有时候我们需要在 `mounts` 中配置一些密钥信息，为了保证安全，Fluid 提供使用 Secret 来配置这些密钥信息的能力。

下面以访问 对象存储（Cloud Object Storage，COS） 数据集为例说明如何配置。

## 创建携带密钥信息的 Dataset

### 查看 Secret

在要创建的 Secret 中，需要写明在上面创建 Dataset 时需要配置的密钥信息。

```
apiVersion: v1
kind: Secret
metadata:
name: mysecret
stringData:
fs.cosn.userinfo.secretId: <COS_SECRET_ID>
fs.cosn.userinfo.secretKey: <COS_SECRET_KEY>
```

可以看到，`fs.cosn.userinfo.secretKey` 和 `fs.cosn.userinfo.secretId` 的具体内容写在 Secret 中，Dataset 通过寻找配置中同名的 Secret 和 key 来读取对应的值，而不再是在 Dataset 直接写明，这样就保证了一些数据的安全性。

### 创建 Secret

```
$ kubectl apply -f secret.yaml
secret/mysecret created
$ kubectl get secret
NAME TYPE DATA AGE
mysecret Opaque 2 57s
```

### 查看 Dataset 和 Runtime

```
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: mydata
spec:
```

```
mounts:
- mountPoint: cosn://<COS_BUCKET>/<COS_DIRECTORY>/
name: mydata
options:
fs.cosn.bucket.region: <COS_REGION>
fs.cosn.impl: org.apache.hadoop.fs.CosFileSystem
fs.AbstractFileSystem.cosn.impl: org.apache.hadoop.fs.CosN
fs.cosn.userinfo.appid: <COS_APP_ID>
encryptOptions:
- name: fs.cosn.userinfo.secretId
valueFrom:
secretKeyRef:
name: mysecret
key: fs.cosn.userinfo.secretId
- name: fs.cosn.userinfo.secretKey
valueFrom:
secretKeyRef:
name: mysecret
key: fs.cosn.userinfo.secretKey
---
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: mydata
spec:
replicas: 1
tieredstore:
levels:
- mediumtype: SSD
path: /mnt/disk1/
quota: 2G
high: "0.8"
low: "0.7"
```

可以看到，在上面的配置中，与直接配置 `fs.cos.endpoint` 不同，我们把 `fs.cosn.userinfo.secretId` 以及 `fs.cosn.userinfo.secretKey` 的配置改为从 Secret 中读取，以此来保障安全性。

> 注意：
> 如果在 `options` 和 `encryptOptions` 中配置了同名的键，例如都有 `fs.cosn.userinfo.secretId` 的配置，那么 `encryptOptions` 中的值会覆盖 `options` 中对应的值的内容。

### 创建 Dataset 和 Runtime

```
$ kubectl create -f dataset.yaml
dataset.data.fluid.io/mydata created
goosefsruntime.data.fluid.io/mydata created
```

查看部署的 GooseFSRuntime 情况，显示都为 Ready 状态表示部署成功。

```
$ kubectl get goosefsruntime mydata
NAME MASTER PHASE WORKER PHASE FUSE PHASE AGE
mydata Ready Ready Ready 62m
```

查看 dataset 的情况，显示 Bound 状态表示 dataset 绑定成功。

```
$ kubectl get dataset mydata
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
mydata 210.00MiB 0.00B 2GiB 0.0% Bound 1h
```

此时，使用 Secret 的 Dataset 即可获取到远程文件。

# 手动扩缩容

最近更新时间：2021-11-18 12:47:32

## 前提条件

已安装 Fluid（version >= 0.5.0）。

> 说明：
> 请参见 安装 文档完成安装。

## 新建工作环境

```
$ mkdir <any-path>/dataset_scale
$ cd <any-path>/dataset_scale
```

## 运行示例

**创建 Dataset 和 GooseFSRuntime 资源对象**

```
$ cat << EOF > dataset.yaml
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: hbase
spec:
mounts:
- mountPoint: https://mirrors.tuna.tsinghua.edu.cn/apache/hbase/stable/
name: hbase
---
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: hbase
spec:
replicas: 1
```

```
tieredstore:
levels:
- mediumtype: MEM
path: /dev/shm
quota: 2G
high: "0.95"
low: "0.7"
EOF
```

> 说明：
> 为了方便用户进行测试，mountPoint 这里使用的是 Web UFS，使用 COS 作为 UFS 可见 使用 GooseFS 挂载 COS（COSN）。

在上述示例中，我们设置 `GooseFSRuntime.spec.replicas` 为1，这意味着我们将启动一个带有一个 Worker 节点的 GooseFS 集群来缓存数据集中的数据。

```
$ kubectl create -f dataset.yaml
dataset.data.fluid.io/hbase created
goosefsruntime.data.fluid.io/hbase created
```

等待 GooseFS 集群正常启动后，可以看到此时创建完成的 Dataset 以及 GooseFSRuntime 处于如下状态：

GooseFS 各组件运行状态：

```
$ kubectl get pod
NAME READY STATUS RESTARTS AGE
hbase-fuse-6pcnc 1/1 Running 0 3m15s
hbase-master-0 2/2 Running 0 3m50s
hbase-worker-w9wxh 2/2 Running 0 3m15s
```

Dataset 状态：

```
$ kubectl get dataset hbase
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
hbase 544.77MiB 0.00B 2.00GiB 0.0% Bound 3m28s
```

GooseFSRuntime 状态：

```
$ kubectl get goosefsruntime hbase -o wide
NAME READY MASTERS DESIRED MASTERS MASTER PHASE READY WORKERS DESIRED WORKERS WOR
KER PHASE READY FUSES DESIRED FUSES FUSE PHASE AGE
hbase 1 1 Ready 1 1 Ready 1 1 Ready 4m55s
```

**Dataset 扩容**

```
$ kubectl scale goosefsruntime hbase --replicas=2
goosefsruntime.data.fluid.io/hbase scaled
```

直接使用 `kubectl scale` 命令即可完成 Dataset 的扩容操作。在成功执行上述命令并等待一段时间后可以看到 Dataset 以及 GooseFSRuntime 的状态均发生了变化：

一个新的 GooseFS Worker 以及对应的 GooseFS Fuse 组件成功启动：

```
$ kubectl get pod
NAME READY STATUS RESTARTS AGE
hbase-fuse-6pcnc 1/1 Running 0 13m
hbase-fuse-8qgww 1/1 Running 0 6m49s
hbase-master-0 2/2 Running 0 13m
hbase-worker-l4c8n 2/2 Running 0 6m49s
hbase-worker-w9wxh 2/2 Running 0 13m
```

Dataset 中的 `Cache Capacity` 从原来的 `2.00GiB` 变为 `4.00GiB`，表明该 Dataset 的可用缓存容量增加：

```
$ kubectl get dataset hbase
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
hbase 544.77MiB 0.00B 4.00GiB 0.0% Bound 15m
```

GooseFSRuntime 中的 `Ready Workers` 以及 `Ready Fuses` 属性均变为2：

```
$ kubectl get goosefsruntime hbase -o wide
NAME READY MASTERS DESIRED MASTERS MASTER PHASE READY WORKERS DESIRED WORKERS WOR
KER PHASE READY FUSES DESIRED FUSES FUSE PHASE AGE
hbase 1 1 Ready 2 2 Ready 2 2 Ready 17m
```

查看 GooseFSRuntime 的具体描述信息可以了解最新的扩缩容信息：

```
$ kubectl describe goosefsruntime hbase
...
Conditions:
...
Last Probe Time: 2021-04-23T07:54:03Z
Last Transition Time: 2021-04-23T07:54:03Z
Message: The workers are scale out.
Reason: Workers scaled out
Status: True
Type: Workers scaled out
```

```
Last Probe Time: 2021-04-23T07:54:03Z
Last Transition Time: 2021-04-23T07:54:03Z
Message: The fuses are scale out.
Reason: Fuses scaled out
Status: True
Type: FusesScaledOut
...
Events:
Type Reason Age From Message
---- ------ ---- ---- -------
Normal Succeed 2m2s GooseFSRuntime GooseFS runtime scaled out. current replicas:
2, desired replicas: 2.
```

**Dataset 缩容**

与扩容类似，缩容时同样可以使用 `kubectl scale` 对 Runtime 的 Worker 数量进行调整：

```
$ kubectl scale goosefsruntime hbase --replicas=1
goosefsruntime.data.fluid.io/hbase scaled
```

成功执行上述命令后，**如果目前环境中没有应用正在尝试访问该数据集**，那么就会触发 Runtime 的缩容。

超出指定 `replicas` 数量的 Runtime Worke r将会被停止：

```
NAME READY STATUS RESTARTS AGE
hbase-fuse-8qgww 1/1 Running 0 21m
hbase-fuse-zql96 1/1 Terminating 0 17m32s
hbase-master-0 2/2 Running 0 22m
hbase-worker-f92vv 2/2 Terminating 0 17m32s
hbase-worker-l4c8n 2/2 Running 0 21m
```

Dataset 的缓存容量 `Cache Capacity` 恢复到 `2.00GiB` ：

```
$ kubectl get dataset hbase
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
hbase 544.77MiB 0.00B 2.00GiB 0.0% Bound 30m
```

> 注意：
> 在目前版本的 Fluid 中，缩容时 Dataset 中 `Cache Capacity` 属性字段的变化存在几分钟的延迟，因此您可能无法迅速观察到这一属性的变化。

GooseFSRuntime 中的 `Ready Workers` 以及 `Ready Fuses` 字段同样变为 `1` ：

```
$ kubectl get goosefsruntime hbase -o wide
NAME READY MASTERS DESIRED MASTERS MASTER PHASE READY WORKERS DESIRED WORKERS WOR
KER PHASE READY FUSES DESIRED FUSES FUSE PHASE AGE
hbase 1 1 Ready 1 1 Ready 1 1 Ready 30m
```

查看 GooseFSRuntime 的具体描述信息可以了解最新的扩缩容信息：

```
$ kubectl describe goosefsruntime hbase
...
Conditions:
...
Last Probe Time: 2021-04-23T08:00:55Z
Last Transition Time: 2021-04-23T08:00:55Z
Message: The workers scaled in.
Reason: Workers scaled in
Status: True
Type: WorkersScaledIn
Last Probe Time: 2021-04-23T08:00:55Z
Last Transition Time: 2021-04-23T08:00:55Z
Message: The fuses scaled in.
Reason: Fuses scaled in
Status: True
Type: FusesScaledIn
...
Events:
Type Reason Age From Message
---- ------ ---- ---- -------
Normal Succeed 6m56s GooseFSRuntime GooseFS runtime scaled out. current replicas:
2, desired replicas: 2.
Normal Succeed 4s GooseFSRuntime GooseFS runtime scaled in. current replicas: 1,
desired replicas: 1.
```

Fluid 提供的这种扩缩容能力，能够帮助用户或是集群管理员适时地调整数据集缓存所占用的集群资源，减少某个不频繁使用的数据集的缓存容量（缩容），或者按需增加某数据集的缓存容量（扩容），以实现更加精细的资源分配，提高资源利用率。

# 环境清理

```
$ kubectl delete -f dataset.yaml
```

# 数据缓存和元数据缓存

最近更新时间：2021-11-18 12:47:33

## 操作场景

本文介绍如何选择数据缓存和元数据缓存方面的参数。

## 操作步骤

### 打开/关闭数据缓存

在 GooseFSRuntime 中，打开数据缓存是 `goosefs.worker.ufs.instream.cache.enabled: "true"`，该属性默认为 true。

可通过在 Runtime 中指定 `goosefs.worker.ufs.instream.cache.enabled: "false"` 来关闭缓存，如下所示：

```
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: hadoop
spec:
replicas: 1
tieredstore:
levels:
- mediumtype: SSD
path: /mnt/disk1/
quota: 290G
high: "0.9"
low: "0.8"
properties:
goosefs.worker.ufs.instream.cache.enabled: "false"
```

### 打开元数据缓存

在 GooseFSRuntime 中，可以通过元数据缓存元数据信息。打开元数据缓存数可以指定 `goosefs.user.metadata.cache.enabled: "true"`，该属性默认是 false。

具体如下：

```
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: hadoop
spec:
replicas: 1
tieredstore:
levels:
- mediumtype: SSD
path: /mnt/disk1/
quota: 290G
high: "0.9"
low: "0.8"
properties:
oosefs.worker.ufs.instream.cache.enabled: "true"
goosefs.user.metadata.cache.enabled: "true"
```

# 数据亲和性调度

最近更新时间：2021-11-18 12:47:33

在 Fluid 中，`Dataset` 资源对象中所定义的远程文件是可被调度的，这意味着您能够像管理您的 Pod 一样管理远程文件缓存在 Kubernetes 集群上的存放位置。另外，Fluid 同样支持对于应用的数据缓存亲和性调度，这种调度方式将应用（例如数据分析任务、机器学习任务等）与所需要的数据缓存放置在一起，以尽可能地减少额外的开销。

## 前提条件

在运行该示例之前，请参见 安装 文档完成安装，并检查 Fluid 各组件正常运行：

```
$ kubectl get pod -n fluid-system
goosefsruntime-controller-5b64fdbbb-84pc6 1/1 Running 0 8h
csi-nodeplugin-fluid-fwgjh 2/2 Running 0 8h
csi-nodeplugin-fluid-ll8bq 2/2 Running 0 8h
dataset-controller-5b7848dbbb-n44dj 1/1 Running 0 8h
```

通常来说，您会看到一个名为 `dataset-controller` 的 Pod、一个名为 `goosefsruntime-controller` 的 Pod 和多个名为 `csi-nodeplugin` 的 Pod 正在运行。其中 `csi-nodeplugin` 这些 Pod 的数量取决于您的 Kubernetes 集群中结点的数量。

## 新建工作环境

```
$ mkdir <any-path>/co-locality
$ cd <any-path>/co-locality
```

## 示例

**查看全部结点**

```
$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
192.168.1.146 Ready <none> 7d14h v1.18.4-tke.13
192.168.1.147 Ready <none> 7d14h v1.18.4-tke.13
```

**使用标签标识结点**

```
$ kubectl label nodes 192.168.1.146 hbase-cache=true
```

在接下来的步骤中，我们将使用 `NodeSelector` 来管理集群中存放数据的位置，所以在这里标记期望的结点。

**再次查看结点**

```
$ kubectl get node -L hbase-cache
NAME STATUS ROLES AGE VERSION HBASE-CACHE
192.168.1.146 Ready <none> 7d14h v1.18.4-tke.13 true
192.168.1.147 Ready <none> 7d14h v1.18.4-tke.13
```

目前，在全部2个结点中，仅有一个结点添加了 `hbase-cache=true` 的标签，接下来我们希望数据缓存仅会被放置在该结点之上。

**检查待创建的 Dataset 资源对象**

```
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: hbase
spec:
mounts:
- mountPoint: https://mirrors.tuna.tsinghua.edu.cn/apache/hbase/stable/
name: hbase
nodeAffinity:
required:
nodeSelectorTerms:
- matchExpressions:
- key: hbase-cache
operator: In
values:
- "true"
```

> 说明：
> 为了方便用户进行测试，mountPoint 这里使用的是 Web UFS，使用 COS 作为 UFS 可见 使用 GooseFS 挂载 COS（COSN）。

在该 `Dataset` 资源对象的 `spec` 属性中，我们定义了一个 `nodeSelectorTerm` 的子属性，该子属性要求数据缓存必须被放置在具有 `hbase-cache=true` 标签的结点之上。

**创建 Dataset 资源对象**

```
$ kubectl create -f dataset.yaml
dataset.data.fluid.io/hbase created
```

**检查待创建的 GooseFSRuntime 资源对象**

```
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: hbase
spec:
replicas: 2
tieredstore:
levels:
- mediumtype: SSD
path: /mnt/disk1
quota: 2G
high: "0.8"
low: "0.7"
```

该配置文件片段中，包含了许多与 GooseFS 相关的配置信息，这些信息将被 Fluid 用来启动一个 GooseFS 实例。上述配置片段中的 `spec.replicas` 属性被设置为2，这表明 Fluid 将会启动一个包含1个 GooseFS Master 和2个 GooseFS Worker 的 GooseFS 实例。

**创建 GooseFSRuntime 资源并查看状态**

```
$ kubectl create -f runtime.yaml
goosefsruntime.data.fluid.io/hbase created

$ kubectl get pod -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
hbase-fuse-42csf 1/1 Running 0 104s 192.168.1.146 192.168.1.146 <none> <none>
hbase-master-0 2/2 Running 0 3m3s 192.168.1.147 192.168.1.147 <none> <none>
hbase-worker-l62m4 2/2 Running 0 104s 192.168.1.146 192.168.1.146 <none> <none>
```

在此处可以看到，尽管我们期望看见两个 GooseFS Worker 被启动，但仅有一组 GooseFS Worker 成功启动，并且运行在具有指定标签（即 `hbase-cache=true` ）的结点之上。

**检查 GooseFSRuntime 状态**

```
$ kubectl get goosefsruntime hbase -o wide
NAME READY MASTERS DESIRED MASTERS MASTER PHASE READY WORKERS DESIRED WORKERS WOR
KER PHASE READY FUSES DESIRED FUSES FUSE PHASE AGE
hbase 1 1 Ready 1 2 PartialReady 1 2 PartialReady 4m3s
```

与预想一致，`Worker Phase` 状态此时为 `PartialReady`，并且 `Ready Workers: 1` 小于 `Desired Workers: 2`。

**查看待创建的应用**

我们提供了一个样例应用来演示 Fluid 是如何进行数据缓存亲和性调度的，首先查看该应用：

**app.yaml**

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
name: nginx
labels:
app: nginx
spec:
replicas: 2
serviceName: "nginx"
podManagementPolicy: "Parallel"
selector: # define how the deployment finds the pods it manages
matchLabels:
app: nginx
template: # define the pods specifications
metadata:
labels:
app: nginx
spec:
affinity:
# prevent two Nginx Pod from being scheduled at the same Node
# just for demonstrating co-locality demo
podAntiAffinity:
requiredDuringSchedulingIgnoredDuringExecution:
- labelSelector:
matchExpressions:
- key: app
operator: In
values:
- nginx
topologyKey: "kubernetes.io/hostname"
containers:
- name: nginx
image: nginx
volumeMounts:
- mountPath: /data
name: hbase-vol
volumes:
- name: hbase-vol
```

```
persistentVolumeClaim:
claimName: hbase
```

其中 `podAntiAffinity` 属性将会确保属于相同应用的多个 Pod 被分散到多个不同的结点，这样的配置能够让我们更加清晰的观察到 Fluid 的数据缓存亲和性调度是怎么进行的。 `podAntiAffinity` 只是一个专用于演示的属性，用户可不必太过关注。

**运行应用**

```
$ kubectl create -f app.yaml
statefulset.apps/nginx created
```

**查看应用运行状态**

```
$ kubectl get pod -o wide -l app=nginx
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
nginx-0 1/1 Running 0 2m5s 192.168.1.146 192.168.1.146 <none> <none>
nginx-1 0/1 Pending 0 2m5s <none> <none> <none> <none>
```

仅有一个 Nginx Pod 成功启动，并且运行在满足 `nodeSelectorTerm` 的结点之上。

**查看应用启动失败原因**

```
$ kubectl describe pod nginx-1
...
Events:
Type Reason Age From Message
---- ------ ---- ---- -------
Warning FailedScheduling <unknown> default-scheduler 0/2 nodes are available: 1 n
ode(s) didn't match pod affinity/anti-affinity, 1 node(s) didn't satisfy existing
pods anti-affinity rules, 1 node(s) had volume node affinity conflict.
Warning FailedScheduling <unknown> default-scheduler 0/2 nodes are available: 1 n
ode(s) didn't match pod affinity/anti-affinity, 1 node(s) didn't satisfy existing
pods anti-affinity rules, 1 node(s) had volume node affinity conflict.
```

如上所示，一方面，为了满足 `PodAntiAffinity` 属性的要求，使得两个 Nginx Pod 无法被调度到同一节点。另一方面，由于目前满足 Dataset 资源对象亲和性要求的结点仅有一个，因此仅有一个 Nginx Pod 被成功调度。

**为另一个结点添加标签**

```
$ kubectl label node 192.168.1.147 hbase-cache=true
```

现在全部两个结点都具有相同的标签了，此时重新检查各个组件的运行状态。

```
$ kubectl get pod -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
hbase-fuse-42csf 1/1 Running 0 44m 192.168.1.146 192.168.1.146 <none> <none>
hbase-fuse-kth4g 1/1 Running 0 10m 192.168.1.147 192.168.1.147 <none> <none>
hbase-master-0 2/2 Running 0 46m 192.168.1.147 192.168.1.147 <none> <none>
hbase-worker-l62m4 2/2 Running 0 44m 192.168.1.146 192.168.1.146 <none> <none>
hbase-worker-rvncl 2/2 Running 0 10m 192.168.1.147 192.168.1.147 <none> <none>
```

两个 GooseFS Worker 都成功启动，并且分别运行在两个结点上：

```
$ kubectl get goosefsruntime hbase -o wide
NAME READY MASTERS DESIRED MASTERS MASTER PHASE READY WORKERS DESIRED WORKERS WOR
KER PHASE READY FUSES DESIRED FUSES FUSE PHASE AGE
hbase 1 1 Ready 2 2 Ready 2 2 Ready 46m43s
```

```
$ kubectl get pod -l app=nginx -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
nginx-0 1/1 Running 0 21m 192.168.1.146 192.168.1.146 <none> <none>
nginx-1 1/1 Running 0 21m 192.168.1.147 192.168.1.147 <none> <none>
```

另一个 nginx Pod 不再处于 `Pending` 状态，已经成功启动并运行在另一个结点上。

由此可见，可调度的数据缓存以及对应用的数据缓存亲和性调度都是被 Fluid 所支持的特性。在绝大多数情况下，这两个特性协同工作，为用户提供了一种更灵活、更便捷的方式管理在 Kubernetes 集群中的数据。

综上可见，Fluid 支持数据缓存的调度策略，这些调度策略为用户提供了更加灵活的数据缓存管理能力。

## 环境清理

```
$ kubectl delete -f .

$ kubectl label node 192.168.1.146 hbase-cache-
$ kubectl label node 192.168.1.147 hbase-cache-
```

# 数据容忍污点调度

最近更新时间：2021-11-18 12:47:33

节点亲和性是 Kubernetes Pod 的一种属性，它使得 Pod 被吸引到一类特定的节点。这可能出于一种偏好，也可能是硬性要求。Taint（污点）则相反，它使节点能够排斥一类特定的 Pod。

容忍度（Tolerations）是应用于 Pod 上的，允许（但并不要求）Pod 调度到带有与之匹配的污点的节点上。

污点和容忍度（Toleration）相互配合，可以用来避免 Pod 被分配到不合适的节点上。每个节点上都可以应用一个或多个污点，这表示对于那些不能容忍这些污点的 Pod，是不会被该节点接受的。

而在 Fluid 中，考虑到 `Dataset` 的可调度性，资源对象中也需要定义 toleration，这意味着您能够像调度您的 Pod 一样调度缓存在 Kubernetes 集群上的存放位置。

## 新建工作环境

```
$ mkdir <any-path>/tolerations
$ cd <any-path>/tolerations
```

## 运行示例

**查看全部节点**

```
$ kubectl get no
NAME STATUS ROLES AGE VERSION
192.168.1.146 Ready <none> 200d v1.18.4-tke.13
```

**为节点配置污点（taint）**

```
$ kubectl taint nodes 192.168.1.146 hbase=true:NoSchedule
```

在接下来的步骤中，我们将看到节点上的污点配置。

**再次查看节点**

```
$ kubectl get node 192.168.1.146 -o yaml | grep taints -A3
taints:
- effect: NoSchedule
```

```
key: hbase
value: "true"
```

目前，节点增加了 taints 配置 NoSchedule，这样默认数据集就无法放置到该节点上。

**检查待创建的 Dataset 资源对象**

```
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: hbase
spec:
mounts:
- mountPoint: https://mirrors.tuna.tsinghua.edu.cn/apache/hbase/stable/
name: hbase
tolerations:
- key: hbase
operator: Equal
value: "true"
```

> 说明：
> 为了方便用户进行测试，mountPoint 这里使用的是 Web UFS，使用 COS 作为 UFS 可见 使用 GooseFS 挂载 COS（COSN）。

在该 `Dataset` 资源对象的 `spec` 属性中，我们定义了一个 `tolerations` 的属性，该子属性要求数据缓存可以放置到配置污点的节点。

**创建 Dataset 资源对象**

```
$ kubectl create -f dataset.yaml
dataset.data.fluid.io/hbase created
```

**检查待创建的 GooseFSRuntime 资源对象**

```
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: hbase
spec:
replicas: 1
tieredstore:
levels:
```

```
- mediumtype: SSD
path: /mnt/disk1
quota: 2G
high: "0.95"
low: "0.7"
```

该配置文件片段中，包含了许多与 GooseFS 相关的配置信息，这些信息将被 Fluid 用来启动一个 GooseFS 实例。上述配置片段中的 `spec.replicas` 属性被设置为1，这表明 Fluid 将会启动一个包含1个 GooseFS Master 和1个 GooseFS Worker 的 GooseFS 实例。

**创建 GooseFSRuntime 资源并查看状态**

```
$ kubectl create -f runtime.yaml
goosefsruntime.data.fluid.io/hbase created

$ kubectl get pod -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
hbase-fuse-n4tnc 1/1 Running 0 63m 192.168.1.146 192.168.1.146 <none> <none>
hbase-master-0 2/2 Running 0 85m 192.168.1.146 192.168.1.146 <none> <none>
hbase-worker-qs26l 2/2 Running 0 63m 192.168.1.146 192.168.1.146 <none> <none>
```

在此处可以看到，GooseFS Worker 被启动，并且运行在具有污点的节点之上。

**检查 GooseFSRuntime 状态**

```
$ kubectl get goosefsruntime hbase -o wide
NAME READY MASTERS DESIRED MASTERS MASTER PHASE READY WORKERS DESIRED WORKERS WOR
KER PHASE READY FUSES DESIRED FUSES FUSE PHASE AGE
hbase 1 1 Ready 1 1 Ready 1 1 Ready 4m3s
```

**查看待创建的应用**

我们提供了一个样例应用来演示 Fluid 是如何进行数据缓存亲和性调度的，首先查看该应用：

- app.yaml

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
name: nginx
labels:
app: nginx
spec:
replicas: 1
serviceName: "nginx"
podManagementPolicy: "Parallel"
selector: # define how the deployment finds the pods it manages
```

```
matchLabels:
app: nginx
template: # define the pods specifications
metadata:
labels:
app: nginx
spec:
tolerations:
- key: hbase
operator: Equal
value: "true"
containers:
- name: nginx
image: nginx
volumeMounts:
- mountPath: /data
name: hbase-vol
volumes:
- name: hbase-vol
persistentVolumeClaim:
claimName: hbase
```

**运行应用**

```
$ kubectl create -f app.yaml
statefulset.apps/nginx created
```

**查看应用运行状态**

```
$ kubectl get pod -o wide -l app=nginx
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
nginx-0 1/1 Running 0 2m5s 192.168.1.146 192.168.1.146 <none> <none>
```

可以看到 Nginx Pod 成功启动，并且运行在配置 taint 的节点之上。

# 环境清理

```
$ kubectl delete -f .
$ kubectl taint nodes 192.168.1.146 hbase=true:NoSchedule-
```

# 数据预加载

最近更新时间：2021-11-18 12:47:34

为了保证应用在访问数据时的性能，可以通过**数据预加载**提前将远程存储系统中的数据拉取到靠近计算结点的分布式缓存引擎中，使得消费该数据集的应用能够在首次运行时即可享受到缓存带来的加速效果。

为此，我们提供了 DataLoad CRD，该 CRD 让您可通过简单的配置就能完成整个数据预加载过程，并对数据预加载的许多行为进行自定义控制。

本文档通过以下两个例子演示了 DataLoad CRD 的使用方法：

- DataLoad 快速使用
- DataLoad 进阶配置

## 前提条件

已安装了 Fluid（version >= 0.6.0）

> 说明：
> 请参见 安装 文档完成 Fluid 安装。

## 新建工作环境

```
$ mkdir <any-path>/warmup
$ cd <any-path>/warmup
```

## DataLoad 快速使用

**配置待创建的 Dataset 和 Runtime 对象**

```
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: spark
spec:
```

```
mounts:
- mountPoint: https://mirrors.bit.edu.cn/apache/spark/
name: spark
---
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: spark
spec:
replicas: 2
tieredstore:
levels:
- mediumtype: SSD
path: /mnt/disk1/
quota: 2G
high: "0.8"
low: "0.7"
```

> 说明：
> 为了方便用户进行测试，mountPoint 这里使用的是 Web UFS，使用 COS 作为 UFS 可见 使用 GooseFS 挂载 COS（COSN）。

在这里，我们将要创建一个 kind 为 `Dataset` 的资源对象（Resource object）。`Dataset` 是 Fluid 所定义的一个 Custom Resource Definition（CRD），该 CRD 被用来告知 Fluid 在哪里可以找到您所需要的数据。Fluid 将该 CRD 对象中定义的 `mountPoint` 属性挂载到 GooseFS 之上。

在本示例中，为了简单，我们使用 COS 进行演示。

**创建 Dataset 和 Runtime 对象**

```
$ kubectl create -f dataset.yaml
```

**等待 Dataset 和 Runtime 准备就绪**

```
$ kubectl get datasets spark
```

如果看到类似以下结果，说明 Dataset 和 Runtime 均已准备就绪：

```
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
spark 1.92GiB 0.00B 4.00GiB 0.0% Bound 4m4s
```

**配置待创建的 DataLoad 对象**

```
apiVersion: data.fluid.io/v1alpha1
kind: DataLoad
metadata:
name: spark-dataload
spec:
loadMetadata: true
dataset:
name: spark
namespace: default
```

`spec.dataset` 指明了需要进行数据预加载的目标数据集，在该例子中，我们的数据预加载目标为 `default` 命名空间下名为 `spark` 的数据集，如果该配置与您所在的实际环境不符，请根据您的实际环境对其进行调整。

**默认情况下，上述 DataLoad 配置将会尝试加载整个数据集中的全部数据**，如果您希望进行更细粒度的控制（例如仅加载数据集下指定路径的数据），请参见 DataLoad 进阶配置。

**创建 DataLoad 对象**

```
$ kubectl create -f dataload.yaml
```

**查看创建的 DataLoad 对象状态**

```
$ kubectl get dataload spark-dataload
```

上述命令会得到类似以下结果：

```
NAME DATASET PHASE AGE
spark-dataload spark Loading 2m13s
```

您也可以通过 `kubectl describe` 获取有关该 DataLoad 的更多详细信息：

```
$ kubectl describe dataload spark-dataload
```

得到以下结果：

```
Name: spark-dataload
Namespace: default
Labels: <none>
Annotations: <none>
API Version: data.fluid.io/v1alpha1
Kind: DataLoad
...
Spec:
Dataset:
```

```
Name: spark
Namespace: default
Status:
Conditions:
Phase: Loading
Events:
Type Reason Age From Message
---- ------ ---- ---- -------
Normal DataLoadJobStarted 80s DataLoad The DataLoad job spark-dataload-loader-job
started
```

上述数据加载过程根据您所在的网络环境不同，可能会耗费数分钟。

**等待数据加载过程完成**

```
$ kubectl get dataload spark-dataload
```

您会看到该 DataLoad 的 `Phase` 状态已经从 `Loading` 变为 `Complete`，这表明整个数据加载过程已经完成。

```
NAME DATASET PHASE AGE
$ spark-dataload spark Complete 5m17s
```

此时再次查看 Dataset 对象的缓存状态：

```
$ kubectl get dataset spark
```

可发现，远程存储系统中的全部数据均已成功缓存到分布式缓存引擎中。

```
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
spark 1.92GiB 1.92GiB 4.00GiB 100.0% Bound 7m41s
```

**环境清理**

```
$ kubectl delete -f .
```

# DataLoad 进阶配置

除了上述示例中展示的数据预加载功能外，通过一些简单的配置，您可以对数据预加载进行更加细节的调整，这些调整包括：

- 指定一个或多个数据集子目录进行加载

- 设置数据加载时的缓存副本数量
- 数据加载前首先进行元数据同步

### 指定一个或多个数据集子目录进行加载

进行数据加载时可以加载指定的子目录（或文件），而不是整个数据集，例如：

```
apiVersion: data.fluid.io/v1alpha1
kind: DataLoad
metadata:
name: spark-dataload
spec:
dataset:
name: spark
namespace: default
loadMetadata: true
target:
- path: /spark/spark-2.4.8
- path: /spark/spark-3.0.1/pyspark-3.0.1.tar.gz
```

上述 DataLoad 仅会加载 `/spark/spark-2.4.8` 目录下的全部文件，以及 `/spark/spark-3.0.1/pyspark-3.0.1.tar.gz` 文件。

`spec.target.path` 的值均为 `mountpoint` 挂载点下的相对路径。例如当前的挂载点为 `cos://test/`，原始路径下有以下文件：

```
cos://test/user/sample.txt
cos://test/data/fluid.tgz
```

那么 `target.path` 可定义为：

```
target:
- path: /user
- path: /data
```

### 设置数据加载时的缓存副本数量

进行数据加载时，您也可以通过配置控制加载的数据副本数量，例如：

```
apiVersion: data.fluid.io/v1alpha1
kind: DataLoad
metadata:
name: spark-dataload
spec:
```

```
dataset:
name: spark
namespace: default
loadMetadata: true
target:
- path: /spark/spark-2.4.8
replicas: 1
- path: /spark/spark-3.0.1/pyspark-3.0.1.tar.gz
replicas: 2
```

上述 DataLoad 在进行数据加载时，对于 `/spark/spark-2.4.8` 目录下的文件仅会在分布式缓存引擎中保留**1份**数据缓存副本，而对于文件 `/spark/spark-3.0.1/pyspark-3.0.1.tar.gz` ，分布式缓存引擎将会保留**2份**缓存副本。

### 数据加载前首先进行元数据同步（建议开启）

在许多场景下，底层存储系统中的文件可能发生了变化，对于分布式缓存引擎来说，需要重新进行文件元信息的同步才能感知到底层存储系统中的变化。因此在进行数据加载前，您也可以通过设置 DataLoad 对象的 `spec.loadMetadata` 来预先完成元信息的同步操作，例如：

```
apiVersion: data.fluid.io/v1alpha1
kind: DataLoad
metadata:
name: spark-dataload
spec:
dataset:
name: spark
namespace: default
loadMetadata: true
target:
- path: /
replicas: 1
```

# 使用 placement 在同个集群上部署多个 dataset

最近更新时间：2021-11-18 12:47:34

通过 GooseFS 和 Fuse，Fluid 为用户提供了一种更为简单的文件访问接口，使得任意运行在 Kubernetes 集群上的程序能够像访问本地文件一样轻松访问存储在远程文件系统中的文件。Fluid 针对数据集进行全生命周期的管理和隔离，尤其对于短生命周期应用（例如数据分析任务、机器学习任务），用户可以在集群中大规模部署。

## 前提条件

在运行该示例之前，请参考 安装 文档完成安装，并检查 Fluid 各组件正常运行：

```
$ kubectl get pod -n fluid-system
NAME READY STATUS RESTARTS AGE
goosefsruntime-controller-5b64fdbbb-84pc6 1/1 Running 0 8h
csi-nodeplugin-fluid-fwgjh 2/2 Running 0 8h
csi-nodeplugin-fluid-ll8bq 2/2 Running 0 8h
dataset-controller-5b7848dbbb-n44dj 1/1 Running 0 8h
```

通常来说，您会看到一个名为 `dataset-controller` 的 Pod、一个名为 `goosefsruntime-controller` 的 Pod 和多个名为 `csi-nodeplugin` 的 Pod 正在运行。其中 `csi-nodeplugin` 这些 Pod 的数量取决于您的 Kubernetes 集群中结点的数量。

## 运行示例

**对某个节点打标签**

```
$ kubectl label node 192.168.0.199 fluid=multi-dataset
```

> 说明：
> 在接下来的步骤中，我们将使用 `NodeSelector` 来管理 Dataset 调度的节点，这里仅做测试使用。

**查看待创建的 Dataset 资源对象**

- dataset.yaml

```
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: hbase
spec:
mounts:
- mountPoint: https://mirrors.tuna.tsinghua.edu.cn/apache/hbase/stable/
name: hbase
nodeAffinity:
required:
nodeSelectorTerms:
- matchExpressions:
- key: fluid
operator: In
values:
- "multi-dataset"
placement: "Shared" // 设置为 Exclusive 或者为空则为独占节点数据集
```

- dataset1.yaml

```
apiVersion: data.fluid.io/v1alpha1
kind: Dataset
metadata:
name: spark
spec:
mounts:
- mountPoint: https://mirrors.bit.edu.cn/apache/spark/
name: spark
nodeAffinity:
required:
nodeSelectorTerms:
- matchExpressions:
- key: fluid
operator: In
values:
- "multi-dataset"
placement: "Shared"
```

说明：
为了方便用户进行测试，mountPoint 这里使用的是 Web UFS，使用 COS 作为 UFS 可见 使用 GooseFS 挂载 COS（COSN)。

**创建 Dataset 资源对象**

```
$ kubectl apply -f dataset.yaml
dataset.data.fluid.io/hbase created
$ kubectl apply -f dataset1.yaml
dataset.data.fluid.io/spark created
```

**查看 Dataset 资源对象状态**

```
$ kubectl get dataset
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
hbase NotBound 6s
spark NotBound 4s
```

如上所示， `status` 中的 `phase` 属性值为 `NotBound` ，这意味着该 `Dataset` 资源对象目前还未与任何 `GooseFSRuntime` 资源对象绑定，接下来，我们将创建一个 `GooseFSRuntime` 资源对象。

**查看待创建的 GooseFSRuntime 资源对象**

- runtime.yaml

```
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: hbase
spec:
replicas: 1
tieredstore:
levels:
- mediumtype: SSD
path: /mnt/disk1
quota: 2G
high: "0.8"
low: "0.7"
```

- runtime-1.yaml

```
apiVersion: data.fluid.io/v1alpha1
kind: GooseFSRuntime
metadata:
name: spark
spec:
replicas: 1
tieredstore:
levels:
- mediumtype: SSD
```

```
  path: /mnt/disk2/
  quota: 4G
  high: "0.8"
  low: "0.7"
```

**创建 GooseFSRuntime 资源对象**

```
$ kubectl create -f runtime.yaml
goosefsruntime.data.fluid.io/hbase created

# 等待 Dataset hbase 全部组件 Running
$ kubectl get pod -o wide | grep hbase
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
hbase-fuse-jl2g2 1/1 Running 0 2m24s 192.168.0.199 192.168.0.199 <none> <none>
hbase-master-0 2/2 Running 0 2m55s 192.168.0.200 192.168.0.200 <none> <none>
hbase-worker-g89p8 2/2 Running 0 2m24s 192.168.0.199 192.168.0.199 <none> <none>
$ kubectl create -f runtime1.yaml
goosefsruntime.data.fluid.io/spark created
```

**检查 GooseFSRuntime 资源对象是否已经创建**

```
$ kubectl get goosefsruntime
NAME MASTER PHASE WORKER PHASE FUSE PHASE AGE
hbase Ready Ready Ready 2m14s
spark Ready Ready Ready 58s
```

`GooseFSRuntime` 是另一个 Fluid 定义的 CRD。一个 `GooseFSRuntime` 资源对象描述了在 Kubernetes 集群中运行一个 GooseFS 实例所需要的配置信息。

等待一段时间，让 GooseFSRuntime 资源对象中的各个组件得以顺利启动，您会看到类似以下状态：

```
$ kubectl get pod -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
hbase-fuse-jl2g2 1/1 Running 0 2m24s 192.168.0.199 192.168.0.199 <none> <none>
hbase-master-0 2/2 Running 0 2m55s 192.168.0.200 192.168.0.200 <none> <none>
hbase-worker-g89p8 2/2 Running 0 2m24s 192.168.0.199 192.168.0.199 <none> <none>
spark-fuse-5z49p 1/1 Running 0 19s 192.168.0.199 192.168.0.199 <none> <none>
spark-master-0 2/2 Running 0 50s 192.168.0.200 192.168.0.200 <none> <none>
spark-worker-96ksn 2/2 Running 0 19s 192.168.0.199 192.168.0.199 <none> <none>
```

注意上面不同的 Dataset 的 worker 和 fuse 组件可以正常的调度到相同的节点 `192.168.0.199` 。

**再次查看 Dataset 资源对象状态**

```
$ kubectl get dataset
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
hbase 443.89MiB 0.00B 2.00GiB 0.0% Bound 11m
spark 1.92GiB 0.00B 4.00GiB 0.0% Bound 9m38s
```

因为已经与一个成功启动的 GooseFSRuntime 绑定，该 Dataset 资源对象的状态得到了更新，此时 `PHASE` 属性值已经变为 `Bound` 状态。通过上述命令可以获知有关资源对象的基本信息

**查看 GooseFSRuntime 状态**

```
$ kubectl get goosefsruntime -o wide
NAME READY MASTERS DESIRED MASTERS MASTER PHASE READY WORKERS DESIRED WORKERS WOR
KER PHASE READY FUSES DESIRED FUSES FUSE PHASE AGE
hbase 1 1 Ready 1 1 Ready 1 1 Ready 11m
spark 1 1 Ready 1 1 Ready 1 1 Ready 9m52s
```

> 说明：
> `GooseFSRuntime` 资源对象的 `status` 中包含了更多更详细的信息。

**查看与远程文件关联的 PersistentVolume 以及 PersistentVolumeClaim**

```
$ kubectl get pv
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE
hbase 100Gi RWX Retain Bound default/hbase 4m55s
spark 100Gi RWX Retain Bound default/spark 51s
```

```
$ kubectl get pvc
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE
hbase Bound hbase 100Gi RWX 4m57s
spark Bound spark 100Gi RWX 53s
```

`Dataset` 资源对象准备完成后（即与 GooseFS 实例绑定后），与该资源对象关联的 PV，PVC 已经由 Fluid 生成，应用可以通过该 PVC 完成远程文件在 Pod 中的挂载，并通过挂载目录实现远程文件访问。

# 远程文件访问

**查看待创建的应用**

- nginx.yaml

```
apiVersion: v1
kind: Pod
metadata:
name: nginx-hbase
spec:
containers:
- name: nginx
image: nginx
volumeMounts:
- mountPath: /data
name: hbase-vol
volumes:
- name: hbase-vol
persistentVolumeClaim:
claimName: hbase
nodeName: 192.168.0.199
```

- nginx1.yaml

```
apiVersion: v1
kind: Pod
metadata:
name: nginx-spark
spec:
containers:
- name: nginx
image: nginx
volumeMounts:
- mountPath: /data
name: hbase-vol
volumes:
- name: hbase-vol
persistentVolumeClaim:
claimName: spark
nodeName: 192.168.0.199
```

**启动应用进行远程文件访问**

```
$ kubectl create -f nginx.yaml
$ kubectl create -f nginx1.yaml
```

登录 Nginx hbase Pod：

```
$ kubectl exec -it nginx-hbase -- bash
```

查看远程文件挂载情况：

```
$ ls -lh /data/hbase
total 444M
-r--r----- 1 root root 193K Sep 16 00:53 CHANGES.md
-r--r----- 1 root root 112K Sep 16 00:53 RELEASENOTES.md
-r--r----- 1 root root 26K Sep 16 00:53 api_compare_2.2.6RC2_to_2.2.5.html
-r--r----- 1 root root 211M Sep 16 00:53 hbase-2.2.6-bin.tar.gz
-r--r----- 1 root root 200M Sep 16 00:53 hbase-2.2.6-client-bin.tar.gz
-r--r----- 1 root root 34M Sep 16 00:53 hbase-2.2.6-src.tar.gz
```

登录 Nginx spark Pod：

```
$ kubectl exec -it nginx-spark -- bash
```

查看远程文件挂载情况：

```
$ ls -lh /data/spark/
total 1.0K
dr--r----- 1 root root 7 Oct 22 12:21 spark-2.4.7
dr--r----- 1 root root 7 Oct 22 12:21 spark-3.0.1
$ du -h /data/spark/
999M /data/spark/spark-3.0.1
968M /data/spark/spark-2.4.7
2.0G /data/spark/
```

登出 Nginx Pod：

```
$ exit
```

正如您所见，WebUFS 上所存储的全部文件像本地文件一样无区别地存在于某个 Pod 中，并且可以被该 Pod 十分方便地访问。

# 远程文件访问加速

为了演示在访问远程文件时，您能获得多大的加速效果，我们提供了一个测试作业的样例：

**查看待创建的测试作业**

- app.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
```

```
name: fluid-copy-test-hbase
spec:
template:
spec:
restartPolicy: OnFailure
containers:
- name: busybox
image: busybox
command: ["/bin/sh"]
args: ["-c", "set -x; time cp -r /data/hbase ./"]
volumeMounts:
- mountPath: /data
name: hbase-vol
volumes:
- name: hbase-vol
persistentVolumeClaim:
claimName: hbase
nodeName: 192.168.0.199
```

- app1.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
name: fluid-copy-test-spark
spec:
template:
spec:
restartPolicy: OnFailure
containers:
- name: busybox
image: busybox
command: ["/bin/sh"]
args: ["-c", "set -x; time cp -r /data/spark ./"]
volumeMounts:
- mountPath: /data
name: spark-vol
volumes:
- name: spark-vol
persistentVolumeClaim:
claimName: spark
nodeName: 192.168.0.199
```

**启动测试作业**

```
$ kubectl create -f app.yaml
job.batch/fluid-copy-test-hbase created
$ kubectl create -f app1.yaml
job.batch/fluid-copy-test-spark created
```

hbase 任务程序会执行 `time cp -r /data/hbase ./` 的 shell 命令，其中 `/data/hbase` 是远程文件在 Pod 中挂载的位置，该命令完成后会在终端显示命令执行的时长。

spark 任务程序会执行 `time cp -r /data/spark ./` 的 shell 命令，其中 `/data/spark` 是远程文件在 Pod 中挂载的位置，该命令完成后会在终端显示命令执行的时长。

等待一段时间，待该作业运行完成，作业的运行状态可通过以下命令查看：

```
$ kubectl get pod -o wide | grep copy
fluid-copy-test-hbase-r8gxp 0/1 Completed 0 4m16s 172.29.0.135 192.168.0.199 <non
e> <none>
fluid-copy-test-spark-54q8m 0/1 Completed 0 4m14s 172.29.0.136 192.168.0.199 <non
e> <none>
```

如果看到如上结果，则说明该作业已经运行完成。

> 注意：
> `fluid-copy-test-hbase-r8gxp` 中的 `r8gxp` 为作业生成的标识，在您的环境中，这个标识可能不同，接下来的命令中涉及该标识的地方请以您的环境为准。

**查看测试作业完成时间**

```
$ kubectl logs fluid-copy-test-hbase-r8gxp
+ time cp -r /data/hbase ./
real 3m 34.08s
user 0m 0.00s
sys 0m 1.24s
$ kubectl logs fluid-copy-test-spark-54q8m
+ time cp -r /data/spark ./
real 3m 25.47s
user 0m 0.00s
sys 0m 5.48s
```

可见，第一次远程文件读取 hbase 耗费了接近3分34秒的时间，读取 spark 耗费接近3分25秒的时间。

**查看 Dataset 资源对象状态**

```
$ kubectl get dataset
NAME UFS TOTAL SIZE CACHED CACHE CAPACITY CACHED PERCENTAGE PHASE AGE
hbase 443.89MiB 443.89MiB 2.00GiB 100.0% Bound 30m
spark 1.92GiB 1.92GiB 4.00GiB 100.0% Bound 28m
```

现在，所有远程文件都已经被缓存在了 GooseFS 中。

**再次启动测试作业**

```
$ kubectl delete -f app.yaml
$ kubectl create -f app.yaml
$ kubectl delete -f app1.yaml
$ kubectl create -f app1.yaml
```

由于远程文件已经被缓存，此次测试作业能够迅速完成：

```
$ kubectl get pod -o wide| grep fluid
fluid-copy-test-hbase-sf5md 0/1 Completed 0 53s 172.29.0.137 192.168.0.199 <none>
<none>
fluid-copy-test-spark-fwp57 0/1 Completed 0 51s 172.29.0.138 192.168.0.199 <none>
<none>
```

```
$ kubectl logs fluid-copy-test-hbase-sf5md
+ time cp -r /data/hbase ./
real 0m 0.36s
user 0m 0.00s
sys 0m 0.36s
$ kubectl logs fluid-copy-test-spark-fwp57
+ time cp -r /data/spark ./
real 0m 1.57s
user 0m 0.00s
sys 0m 1.57s
```

同样的文件访问操作，hbase 仅耗费了0.36秒，spark仅耗费了1.57秒。

这种大幅度的加速效果归因于 GooseFS 所提供的强大的缓存能力，这种缓存能力意味着，只要您访问某个远程文件一次，该文件就会被缓存在 GooseFS 中，您的所有接下来的重复访问都不再需要读取远程文件，而是从 GooseFS 中直接获取数据。

# 环境清理

```
$ kubectl delete -f .
$ kubectl label node 192.168.0.199 fluid-
```