Mapreduce 服务

组件开发规范

文档版本 01

发布日期 2024-04-29

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 安全认证

1.1 安全认证应用开发规则

一个进程里只用一个账号，并且一个进程里只需显式认证一次

如下场景都需遵从上述“安全认证”的要求（设计环节关注）

1. 一个进程里需同时访问多个集群（各自有独立的KrbServer和LdapServer ）。
2. 在容器（例如Tomcat）上运行APP，一个容器里所有APP属于同一个进程。

1.2 安全认证应用开发建议

账号管理应该遵循如下原则

1. 业务应用应申请业务应用账号，切忌使用原有系统账号。
2. 新账号权限遵从最小原则。

2 ClickHouse

2.1 ClickHouse 应用开发规则

集群安装为安全版，则需要保证客户端与服务端的时间一致

如果集群为安全版，需要进行kerberos认证，则需要服务端与客户端的时间一致，时间一致需要注意时区之间的时差的转换。如果时间不一致，会导致客户端认证失败，后续业务流程无法执行。

ClickHouse 服务独享一个 Zookeeper 服务

ClickHouse强依赖Zookeeper，对Zookeeper会进行大量的读写操作，尽量一个ClickHouse服务独享一个Zookeeper，避免影响其他服务。

合理使用数据表的分区字段和索引字段

MergeTree引擎，数据是以分区目录的形式进行组织存储的，在进行的数据查询时，使用分区可以有效跳过无用的数据文件，减少数据的读取。

MergeTree引擎会根据索引字段进行数据排序，并且根据index_granularity的配置生成稀疏索引。根据索引字段查询，能快速过滤数据，减少数据的读取，大大提升查询性能。

大批量少频次的插入

ClickHouse的每次数据插入都会生成一到多个part文件，如果data part过多则会导致merge压力变大，甚至出现服务异常影响数据插入。建议一次插入10万行，每秒不超过1次插入。

不允许使用字符类型存放时间、日期或数值类型的数据

特别是需要对该时间、日期或数值类型字段进行运算或者比较的时候。

单表(分布式表)的记录数不要超过万亿，单表(本地表)不超过百亿

对于万亿以上表的查询，性能较差，且集群维护难度变大。

表的设计都要考虑到数据的生命周期管理

磁盘的空间是有限的，需要考虑数据的生命周期管理。MergeTree引擎在建表的时候支持列字段和表级的TTL。当列字段中的值过期时,ClickHouse会将它们替换成数据类型的默认值。如果分区内，某一列的所有值均已过期，则ClickHouse会从文件系统中删除这个分区目录下的列文件。当表内的数据过期时,ClickHouse会删除所有对应的行。

外部件保证数据导入的幂等

ClickHouse不支持数据写入的事务保证。通过外部导入模块控制数据的幂等，比如某个批次的数据导入异常，则drop对应的分区数据，等异常修复后重新导入该分区数据。

创建 ClickHouse 本地表时需要携带 partition by 关键字，否则在 Manager 上的 ClickHouse 数据迁移页面无法对该表进行迁移

ClickHouse数据迁移页面在对表数据进行迁移时依赖表的分区字段，如果创建表时没有使用partition by创建分区，则在Manager上的ClickHouse数据迁移页面无法对该表进行迁移。

join 查询时小表在右

两表JOIN时，会将右表数据加载到内存中，再根据右表数据遍历左表做匹配，将小表放在右边，减少匹配查询的次数。根据使用的情况，大表join小表的性能比小表join大表的性能有数量级的提升。

2.2 ClickHouse 应用开发建议

合理配置最大并发数

ClickHouse处理速度快是因为采用了并行处理机制，即使一个查询，默认也会用服务器一半的CPU去执行，所以ClickHouse对高并发查询的场景支持的不够。官方默认的最大并发数是100，可以根据实际场景调整并发配置，建议不超过200。

部署负载均衡组件，查询基于负载均衡组件进行，避免单点查询压力太大影响性能

ClickHouse支持连接集群中的任意节点查询，如果查询集中到一台节点，可能会导致该节点的压力过大并且可靠性不高。建议使用ClickHouseBalancer或者其他负载均衡服务，均衡查询负载，提升可靠性。

合理设置分区键，控制分区数在一千以内，分区字段使用整型

1. 建议使用toYYYYMMDD(表字段pt_d)作为分区键，表字段pt_d是date类型。
2. 如果业务场景需要做小时分区，使用toYYYYMMDD(表字段pt_d)、toYYYYMMDD(表字段pt_h)做联合分区键，其中toYYYYMMDD(表字段pt_h)是整型小时数。
3. 如果保存多年数据，建议考虑使用月做分区，例如toYYYYMM(表字段pt_d)。
4. 综合考虑数据分区粒度、每个批次提交的数据量、数据的保存周期等因素，合理控制part数量。

查询时最常使用且过滤性最高的字段作为主键，依次按照访问频度从高到低、维度基数从小到大来排

数据是按照主键排序存储的，查询的时可以通过主键快速筛选数据，创建表时合理的设置主键能够大大减少读取的数据量，提升查询性能。例如所有的分析，都需要指定业务的id，则可以将业务id字段作为主键的第一个字段。

根据业务场景合理设置稀疏索引粒度

ClickHouse的主键索引采用的是稀疏索引存储，稀疏索引的默认采样粒度是8192行，即每8192行取一条记录在索引文件中。

使用建议：

1. 索引粒度越小，对于小范围的查询更有效，避免查询资源的浪费。
2. 索引粒度越大，则索引文件越小，索引文件的处理会更快。
3. 超过10亿的表索引粒度可设为16384，其他设为8192或者更小值。

本地表建表参考

本地表创建参考：

```
CREATE TABLE mybase_local.mytable
(
    `did` Int32,
    `app_id` Int32,
    `region` Int32,
    `pt_d` Date
)
ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/mybase_local/mytable', '{replica}')
PARTITION BY toYYYYMMDD(pt_d)
ORDER BY (app_id, region)
SETTINGS index_granularity = 8192, use_minimalistic_part_header_in_zookeeper = 1;
```

使用说明：

1. 表引擎选择：

   ReplicatedMergeTree:支持副本特性的MergeTree引擎，也是最常用的引擎。
2. ZooKeeper上的表信息注册路径，用于区分集群中的不同配置：

   /clickhouse/tables/{shard}/{databaseName}/{tableName}：{shard}是分片名称，{databaseName}是数据库名称，{tableName}是复制表名称。
3. order by 主键字段：

   查询时最常使用且过滤性最高的字段作为主键。依次按照访问频度从高到低、维度基数从小到大来排。排序字段不宜太多，建议不超过4个，否则merge的压力会较大。排序字段不允许为null，如果存在null值，需要进行数据转换。
4. partition by 分区字段

   分区键不允许为null，如果字段中有null值，需要进行数据转换。
5. 表级别的参数配置：

   index_granularity：稀疏索引粒度配置，默认是8192。

   use_minimalistic_part_header_in_zookeeper：ZooKeeper中数据存储是否启动新版本的优化存储方式。
6. 建表定义可以参考官网链接：**https://clickhouse.tech/docs/en/engines/table-engines/mergetree-family/mergetree/**。

分布式表建表参考

本地表创建参考：

```
CREATE TABLE mybase.mytable AS mybase_local.mytable
ENGINE = Distributed(cluster_3shards_2replicas, mybase_local, mytable, rand());
```

使用说明：

1. 分布式表名称：mybase.mytable。
2. 本地表名称：mybase_local.mytable。
3. 通过“AS”关联分布式表和本地表，保证分布式表的字段定义跟本地表一致。
4. 分布式表引擎的参数说明：

   cluster_3shards_2replicas：逻辑集群名称。

   mybase_local：本地表所在库名。

   mytable：本地表名。

   rand()：可选参数，分片键（sharding key），可以是表中一列的原始数据（如did），也可以是函数调用的结果，如随机值rand()。注意该键要尽量保证数据均匀分布，另外一个常用的操作是采用区分度较高的列的哈希值，如intHash64(user_id)。

根据业务场景表的字段选择最小满足的类型使用

数值类型：UInt8/UInt16/UInt32/UInt64, Int8/Int16/Int32/Int64, Float32/Float64等，选择不同长度，性能差别较大。

基于大宽表进行数据分析，不建议使用大表 join 大表的操作，对分布式 join 查询转化成本地表的 join 查询操作，提升性能

ClickHouse分布式join的性能较差，建议在模型侧将数据聚合成大宽表再导入ClickHouse。分布式join的查询转成本地表的join查询，不仅省去大量的节点间数据传播，同时本地表参与计算的数据量也会少很多。业务层再基于所有分片本地join的结果进行数据汇总，性能会有数量级的提升。

设置合理的 part 大小

min_bytes_to_rebalance_partition_over_jbod参数表示参与在JBOD卷中磁盘之间自动平衡分发part的最小size，该值不能设置得太小或者太大。

若该值设置得太小，小于max_bytes_to_merge_at_max_space_in_pool/1024，那么clickhouse server进程将会启动失败，另外还会引发不必要的part在磁盘间移动。

若该值设置得过大，则很难有part达到这个条件，比如：min_bytes_to_rebalance_partition_over_jbod大于max_data_part_size_bytes（卷中的磁盘可以存储的part的最大大小），则没有part能达到自动平衡的条件。

3 Doris

3.1 Doris 建表规范

该章节主要介绍创建Doris表时需遵循的规则和建议。

Doris 建表规则

- 在创建Doris表指定分桶buckets时，每个桶的数据大小应保持在100MB~3GB之间，单分区中最大分桶数量不超过5000。
- 表数据超过5亿条以上必须设置分区分桶策略。
- 表的分桶列不要设置太多，一般情况下设置1或2个列即可，同时需要兼顾数据分布均匀和查询吞吐均衡。
  - 数据均匀是为了避免某些桶的数据存在倾斜影响数据均衡和查询效率。
  - 查询吞吐利用查询SQL的分桶剪裁优化避免了全桶扫描，以提升查询性能。
  - 分桶列的选取：优先考虑数据较为均匀且常用于查询条件的列作为分桶列。

    可使用以下方法分析是否会导致数据倾斜：

    SELECT a, b, COUNT(*) FROM tab GROUP BY a,b;

    命令执行后查看各个分组的数据条数是否相差不大，如果相差超过2/3或1/2，则需要重新选择分桶字段。
- 2千万以内数据禁止使用动态分区。动态分区会自动创建分区，而小表用户关注不到，会创建出大量不使用的分区分桶。
- 创建表时，排序键key不能太多，一般建议3~5个；太多key会导致数据写入较慢，影响数据导入性能。
- 不使用Auto Bucket，需按照已有的数据量来进行分区分桶，能更好的提升导入及查询性能。Auto Bucket会造成Tablet数量过多，最终导致有大量的小文件。
- 创建表时的副本数必须至少为2，默认是3，禁止使用单副本。

Doris 建表建议

- 单表物化视图不能超过6个，物化视图不建议嵌套，不建议数据写入时通过物化视图进行重型聚合和Join计算等ETL任务。

- 对于有大量历史分区数据，但是历史数据比较少，或者数据不均衡，或者数据查询概率较小的情况，可以创建历史分区（比如年分区，月分区），将所有历史数据放到对应分区里。

  创建历史分区方式为：**FROM ("***2000-01-01***") TO ("***2022-01-01***") INTERVAL 1 YEAR**
- 1千万~2亿以内数据为了方便可以不设置分区（Doris内部有一个默认分区），直接用分桶策略即可。
- 如果分桶字段存在30%以上的数据倾斜，则禁止使用Hash分桶策略，改为使用Random分桶策略，相关命令为：

  Create table ... **DISTRIBUTED BY RANDOM BUCKETS** *10* ...
- 建表时第一个字段一定是最常查询使用的列，默认有前缀索引快速查询能力，选取最常查询且高基数的列作为前缀索引，默认将一行数据的前36个字节作为这行数据的前缀索引（varchar类型的列只能匹配20个字节，并且会匹配不足36个字节截断前缀索引）。
- 超过亿级别的数据，如果有模糊匹配或者等值/in条件，可以使用倒排索引（Doris 2.x版本开始支持）或者Bloomfilter。如果是低基数列的正交查询适合使用bitmap索引（bitmap索引的基数在10000~100000之间效果较好）。
- 建表时需要提前规划将来要使用的字段个数，可以多预留几十个字段，类型包括整型、字符型等。避免将来字段不够使用，需要较高代价临时去添加字段。

3.2 Doris 数据变更规范

该章节主要介绍Doris数据变更时需遵循的规则和建议。

Doris 数据变更规则

- 应用程序不能直接使用**delete**或者**update**语句变更数据，可以使用CDC的**upsert**方式来实现。
- 不建议业务高峰期或在表上频繁的进行加减字段，建议在业务前期规划建表时预留将来要使用的字段。如果必须添加或删除字段，及修改字段类型和注释，需在业务低峰期，停止相关表的写入和修改业务后，通过重建表方式实现以上操作：

  a. 新建一个表，该表结构和需进行增删改字段的表结构相同。在新建表中增加需要添加的新字段、删除不需要的字段、或修改需改变类型的字段。

  b. 选取指定字段数据插入到新创建的表中：

  INSERT INTO *新创建的表* **SELECT** *指定的字段* **FROM** *已存在需要修改列的表*;

  说明

  如果表数据量较大，可按时间过滤分批次将数据导入到新表，减小CPU或MEM内存瞬时冲高占用问题，影响查询业务，命令为：

  insert into *tab1* **select** *col* **from** *tab* **where** *date <= xx*;

  c. 交换两个表的名称，更多介绍请参见**交换表**：

  ALTER TABLE *[db.]tbl1* **REPLACE WITH TABLE** *tbl2* **[PROPERTIES('swap' = 'true')];**
- 对于部分查询，可能执行时间比较长，查询比较耗费内存和CPU等资源，需要在SQL或user级别设置查询超时时间参数：query_timeout

Doris 数据变更建议

执行特殊的大SQL操作时，可以使用类似**SELECT /*+ SET_VAR(query_timeout = xxx*/ from table**通过Hint方式设置Session会话变量，不要设置全局的系统变量。

3.3 Doris 命名规范

该章节主要介绍创建Doris数据库或表时，数据库名或表名需遵循的规则和建议。

Doris 命名规则

数据库字符集需指定UTF-8，并且只支持UTF-8。

Doris 命名建议

- 数据库名称统一使用小写方式，中间使用下划线（ _ ）分割，长度为62字节以内。
- Doris表名称大小写敏感，统一使用小写方式，中间使用下划线（ _ ）分割，长度为64字节以内。

3.4 Doris 数据查询规范

该章节主要介绍Doris数据查询时需遵循的规则和建议。

Doris 数据查询规则

- 在数据查询业务代码中建议查询失败时进行重试，再次下发查询。
- **in**中常量枚举值超过1000后，必须修改为子查询。
- 禁止使用REST API（Statement Execution Action）执行大量SQL查询，该接口仅用于集群维护。
- query查询条件返回结果超过5万条，则使用JDBC Catalog或者OUTFILE方式导出查询数据，否则FE上大量数据传输将占用FE资源，影响集群稳定性。
  - 如果是交互式查询，建议使用分页方式（offset limit）导出数据，分页命令为Order by。
  - 如果数据导出提供给第三方使用，建议使用outfile或者export方式
- 2个以上大于3亿的表JOIN使用Colocation Join。
- 亿级别大表禁止使用**select ***查询数据，查询时需明确要查询的字段。
  - 使用SQL Block方式禁止**select ***操作。
  - 如果是高并发点查询，建议开启行存储（Doris 2.x版本支持），并且使用PreparedStatement查询。
- 亿级以上表数据查询必须设置分区分桶条件。
- 禁止对分区表执行全分区数据扫描操作。

Doris 数据查询建议

- 一次**insert into select**数据超过1亿条后，建议拆分为多个**insert into select**语句执行，分成多个批次来执行。
- 不要使用OR作为JOIN条件。

- 不建议频繁的数据delete修改，将要删除的数据攒批，偶尔进行批量删除，且需要带上条件，提升系统稳定性和删除效率。
- 大量数据排序（5亿以上）后返回部分数据，建议先减少数据范围再执行排序，否则大量排序会影响性能。例如：

  将**from table order by datatime desc limit** *10*优化为**from table where datatime='***2023-10-20***' order by datatime desc limit** *10*。
- 查询任务性能调优参数**parallel_fragment_exec_instance_num**使用注意事项：

  此参数是session级别设置，表示可并发执行的fragment数量，对CPU消耗较大，因此一般情况下不需要设置此参数。如果需要设置此参数来加速查询性能，必须遵循以下规则：
  - 切勿设置该参数为全局生效，禁止使用**set global**方式进行设置。
  - 设置参数值建议为偶数2或4（最大值不要超过单节点CPU核数的一半）。
  - 设置此参数值时需要观察CPU使用率，CPU使用率小于50%时方可考虑设置。
  - 如果查询SQL是**insert into select**大数据量的方式，不建议设置此参数。

3.5 Doris 数据导入规范

该章节主要介绍Doris数据导入规范。

Doris 数据导入建议

- 禁止高频执行**update**、**delete**或**truncate**操作，推荐几分钟执行一次，使用**delete**必须设置分区或主键列条件。
- 禁止使用**INSERT INTO** *tbl1* **VALUES (“1”), (“a”);**方式导入数据，少量少次写可以，多量多频次时需使用Doris提供的StreamLoad、BrokerLoad、SparkLoad或者Flink Connector方式。
- 在Flink实时写入数据到Doris的场景下，CheckPoint设置的时间需要考虑每批次数据量，如果每批次数据太小会造成大量小文件，推荐值为60s。
- 建议不使用**insert values**作为数据写入的主要方式，批量数据导入推荐使用StreamLoad、BrokerLoad或SparkLoad。
- 使用INSERT INTO WITH LABEL *XXX* SELECT方式进行数据导入，如果有下游依赖或查询，需要先查看导入的数据是否为可见状态。

  具体查看方法：通过**show load where label='***xxx***'** SQL命令查询当前INSERT任务状态（status）是否为“VISIBLE”，如果为“VISIBLE”导入的数据才可见。
- Streamload数据导入适合10 GB以内的数据量、Brokerload适合百GB以内数据，数据过大时可考虑使用SparkLoad。
- 禁止使用Doris的Routine Load进行导入数据操作，推荐使用Flink查询Kafka数据再写入Doris，更容易控制导入数据单批次数据量，避免大量小文件产生。如果确实已经使用了Routine Load进行导数，在没整改前请配置FE “max_tolerable_backend_down_num”参数值为“1”，以提升导入数据可靠性。
- 建议低频攒批导入数据，平均单表导入批次间隔需大于30s，推荐间隔60s，一次导入1000~100000行数据。

3.6 Doris UDF 开发规范

本章节主要介绍开发Doris UDF程序时应遵循的规则和建议。

Doris UDF 开发规则

- UDF中方法调用必须是线程安全的。
- UDF实现中禁止读取外部大文件到内存中，如果文件过大可能会导致内存耗尽。
- 需避免大量递归调用，否则容易造成栈溢出或oom。
- 需避免不断创建对象或数组，否则容易造成内存耗尽。
- Java UDF应该捕获和处理可能发生的异常，不能将异常给服务处理，以避免程序出现未知异常。可以使用try-catch块来处理异常，并在必要时记录异常信息。
- UDF中应避免定义静态集合类用于临时数据的存储，或查询外部数据存在较大对象，否则会导致内存占用过高。
- 应该避免类中**import**的包和服务侧包冲突，可通过**grep -lr** "*完全限定类名*"命令来检查冲突的Jar包。如果发生类名冲突，可通过完全限定类名方式来避免。

Doris UDF 开发建议

- 不要执行大量数据的复制操作，防止堆栈内存溢出。
- 应避免使用大量字符串拼接操作，否则会导致内存占用过高。
- Java UDF应该使用有意义的名称，以便其他开发人员能够轻松理解其用途。建议使用驼峰式命名法，并以UDF结尾，例如：MyFunctionUDF。
- Java UDF应该指定返回值的数据类型，并且必须具有返回值，返回值默认或异常时不要设置为NULL。建议使用基本数据类型或Java类作为返回值类型。

3.7 Doris 连接运行规范

连接Doris和运行Doris任务时需遵循的规范如下：

- 推荐使用ELB连接Doris，避免当连接的FE故障时，无法对外提供服务。
- 当Doris单实例或硬件故障时，新提交的任务能运行成功，但不能确保故障时正在运行的任务能执行成功。因此，需要用户连接Doris执行任务时进行失败重试，当任务遇到未知原因失败时，能保证重试新提交的任务能运行成功。

4 Flink

4.1 Flink 应用开发规则

Flink 任务异常结束，残留目录清理

安装FlinkServer实例后可以对Flink残留目录进行自动清理。

默认只清理ZooKeeper中“/flink_base”目录下的残留目录以及HDFS中“/flink/recovery”目录下的残留目录。

4.2 Flink 应用开发建议

FlinkServer 的使用

FlinkServer支持FlinkSQL作业以及FlinkJar作业的提交运行，建议通过FlinkServer来提交Flink作业。

5 HBase

5.1 HBase 应用开发规则

Configuration 实例的创建

该类应该通过调用HBaseConfiguration的create()方法来实例化。否则，将无法正确加载HBase中的相关配置项。

正确示例：

```
//该部分，应该是在类成员变量的声明区域声明
private Configuration hbaseConfig = null;
//建议在类的构造函数中，或者初始化方法中实例化该类
hbaseConfig = HBaseConfiguration.create();
```

错误示例：

```
hbaseConfig = new Configuration();
```

共享 Configuration 实例

HBase客户端代码通过创建一个与ZooKeeper之间的HConnection，来获取与一个HBase集群进行交互的权限。一个ZooKeeper的HConnection连接，对应着一个Configuration实例，已经创建的HConnection实例，会被缓存起来。也就是说，如果客户端需要与HBase集群进行交互的时候，会传递一个Configuration实例到缓存中去，HBase Client部分通过已缓存的HConnection实例，来判断属于这个Configuration实例的HConnection实例是否存在，如果不存在，会创建一个新的HConnection，如果存在，则会直接返回相应的实例。

因此，如果频频的创建Configuration实例，会导致创建很多不必要的HConnection实例，很容易达到ZooKeeper的连接数上限。

建议在整个客户端代码范围内，都共用同一个Configuration对象实例。

Table 实例的创建

```
public abstract class TableOperationImpl {
  private static Configuration conf = null;
  private static Connection connection = null;
  private static Table table = null;
  private static TableName tableName = TableName.valueOf("sample_table");
```



```
  public TableOperationImpl() {
    init();
  }
  public void init() {
    conf = ConfigurationSample.getConfiguration();
    try {
      connection = ConnectionFactory.createConnection(conf);
      table = conn.getTable(tableName);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
  public void close() {
    if (table != null) {
      try {
        table.close();
      } catch (IOException e) {
        System.out.println("Can not close table.");
      } finally {
        table = null;
      }
    }
    if (connection != null) {
      try {
        connection.close();
      } catch (IOException e) {
        System.out.println("Can not close connection.");
      } finally {
        connection = null;
      }
    }
  }
  public void operate() {
    init();
    process();
    close();
  }
}
```

不允许多个线程在同一时间共用同一个 Table 实例

Table是一个非线程安全类，因此，同一个Table实例，不应该被多个线程同时使用，否则可能会出现并发问题。

Table 实例缓存

如果一个Table实例可能长时间会被同一个线程固定且频繁的用到，例如，通过一个线程不断的往一个表内写入数据，那么这个Table在实例化后，就需要缓存下来，而不是每一次插入操作，都要实例化一个Table对象（尽管提倡实例缓存，但也不是在一个线程中一直沿用一个实例，个别场景下依然需要重构，可参见下一条规则）。

正确示例：

说明

注意该实例中提供的以Map形式缓存Table实例的方法，未必通用。这与多线程多Table实例的设计方案有关。如果确定一个Table实例仅仅可能会被用于一个线程，而且该线程也仅有一个Table实例的话，就无须使用Map。这里提供的思路仅供参考。

```
//该Map中以TableName为Key值，缓存所有已经实例化的Table
private Map<String, Table> demoTables = new HashMap<String, Table>();
//所有的Table实例，都将共享这个Configuration实例
private Configuration demoConf = null;
/**
* <初始化一个HTable类>
```



```
* <功能详细描述>
* @param tableName
* @return
* @throws IOException
* @see [类、类#方法、类#成员]
*/
private Table initNewTable(String tableName) throws IOException
{
try (Connection conn = ConnectionFactory.createConnection(demoConf)){
      return conn.getTable(tableName);
    }
}
/**
* <获取Table实例>
* <功能详细描述>
* @see [类、类#方法、类#成员]
*/
private Table getTable(String tableName)
{
if (demoTables.containsKey(tableName))
{
return demoTables.get(tableName);
} else {
Table table = null;
try
{
table = initNewTable(tableName);
demoTables.put(tableName, table);
}
catch (IOException e)
{
// TODO Auto-generated catch block
e.printStackTrace();
}
return table;
}
}
/**
* <写数据>
* <这里未涉及到多线程多Table实例在设计模式上的优化.这里采用同步方法，
* 主要是是考虑到同一个Table是非线程安全的.通常，建议一个Table实例，在同一
*  时间只能被用在一个写数据的线程中>
* @param dataList
* @param tableName
* @see [类、类#方法、类#成员]
*/
public void putData(List<Put> dataList, String tableName)
{Table table = getTable(tableName);
//关于这里的同步：如果在采用的设计方案中，不存在多线程共用同一个Table实例
//的可能的话，就无须同步了。这里需要注意Table实例是非线程安全的
synchronized (table)
{
try
{
table.put(dataList);
table.notifyAll();
}
catch (IOException e)
{
                // 在捕获到IOE时，需要将缓存的实例重构。
try {
    // 关闭之前的Connection.
      table.close();
                // 重新创建这个实例.
                table = initNewTable(tableName);
} catch (IOException e1) {
// TODO
}
}
```



```
}
}
```

错误示例：

```
public void putDataIncorrect(List<Put> dataList, String tableName)
{Table table = null;
try
{
//每次写数据，都创建一个HTable实例
table = initNewTable(tableName);
table.put(dataList);
}
catch (IOException e1)
{
// TODO Auto-generated catch block
e1.printStackTrace();
}
finally
{
table.close();
}
}
```

Table 实例写数据的异常处理

尽管在前一条规则中提到了提倡Table实例的重构，但是，并非提倡一个线程自始至终要沿用同一个Table实例，当捕获到IOException时，依然需要重构Table实例。示例代码可参考上一个规则的示例。

另外，请谨慎调用如下两个方法：

- **Configuration#clear：**

  这个方法，会清理所有已加载的属性，对于已经在使用这个Configuration的类或线程而言，可能会带来潜在的问题（例如，假如Table还在使用这个Configuration，那么，调用这个方法后，Table中的这个Configuration的所有的参数，都被清理掉了），也就是说：只要还有对象或者线程在使用这个Configuration，就不应该调用这个clear方法，除非所有的类或线程，都已经确定不用这个Configuration了。

  因此，这个方法，应该要放在进程退出时执行，而不是每一次Table要重构的时候执行。

- **HConnectionManager#deleteAllConnections：**

  这个可能会导致现有的正在使用的连接被从**连接集合**中清理掉，同时，因为在HTable中保存了原有连接的引用，可能会导致这个连接无法关闭，进而可能会造成泄漏。因此，这个方法不建议使用。

写入失败的数据要做相应的处理

在写数据的过程中，如果进程异常或一些其它的短暂的异常，可能会导致一些写入操作失败。因此，对于操作的数据，需要将其记录下来。在集群恢复正常后，重新将其写入到HBase数据表中。

另外，有一点需要注意：HBase Client返回写入失败的数据，是不会自动重试的，仅仅会告诉接口调用者哪些数据写入失败了。对于写入失败的数据，一定要做一些安全的处理，例如可以考虑将这些失败的数据，暂时写在文件中，或者，直接缓存在内存中。

正确示例：


```
private List<Row> errorList = new ArrayList<Row>();
/**
* <采用PutList的模式插入数据>
* <如果不是多线程调用该方法，可不采用同步>
* @param put 一条数据记录
* @throws IOException
* @see [类、类#方法、类#成员]
*/
public synchronized void putData(Put put)
{
// 暂时将数据缓存在该List中
dataList.add(put);
// 当dataList的大小达到PUT_LIST_SIZE之后，就执行一次Put操作
if (dataList.size() >= PUT_LIST_SIZE)
{
try
{
demoTable.put(dataList);
}
catch (IOException e)
{
// 如果是RetriesExhaustedWithDetailsException类型的异常，
// 说明这些数据中有部分是写入失败的这通常都是因为
// HBase集群的进程异常引起，有时也会因为有大量
// 的Region正在被转移，导致尝试一定的次数后失败
if (e instanceof RetriesExhaustedWithDetailsException)
{
RetriesExhaustedWithDetailsException ree =
  (RetriesExhaustedWithDetailsException)e;
int failures = ree.getNumExceptions();
for (int i = 0; i < failures; i++)
{
errorList.add(ree.getRow(i));
}
}
}
dataList.clear();
}
}
```

资源释放

关于ResultScanner和Table实例，在用完之后，需要调用它们的Close方法，将资源释放掉。Close方法，要放在finally块中，来确保一定会被调用到。

正确示例：

```
ResultScanner scanner = null;
try
{
scanner = demoTable.getScanner(s);
//Do Something here.
}
finally
{
scanner.close();
}
```

错误示例：

1. 在代码中未调用scanner.close()方法释放相关资源。
2. scanner.close()方法未放置在finally块中。

   ```
   ResultScanner scanner = null;
   scanner = demoTable.getScanner(s);
   //Do Something here.
   scanner.close();
   ```


Scan 时的容错处理

Scan时不排除会遇到异常，例如，租约过期。在遇到异常时，建议Scan应该有重试的操作。

事实上，重试在各类异常的容错处理中，都是一种优秀的实践，这一点，可以应用在各类与HBase操作相关的接口方法的容错处理过程中。

不用 Admin 时，要及时关闭，Admin 实例不应常驻内存

Admin的示例应尽量遵循 “用时创建，用完关闭”的原则。不应该长时间缓存同一个Admin实例。

5.2 HBase 应用开发建议

不要调用 Admin 的 closeRegion 方法关闭一个 Region

Admin中，提供了关闭一个Region的接口：

public void closeRegion(final String regionname, final String serverName)

通过该方法关闭一个Region，HBase Client端会直接发RPC请求到Region所在的RegionServer上，整个流程对Master而言，是不感知的。也就是说，尽管RegionServer关闭了这个Region，但是，在Master侧，还以为该Region是在该RegionServer上面打开的。假如，在执行Balance的时候，Master计算出恰好要转移这个Region，那么，这个Region将无法被关闭，本次转移操作将无法完成（关于这个问题，在当前的HBase版本中的处理的确还欠缺妥当）。

因此，暂时不建议使用该方法关闭一个Region。

采用 PutList 模式写数据

Table类中提供了两种写数据的接口：

1. public void put(final Put put) throws IOException
2. public void put(final List<Put> puts) throws IOException

第1种方法较之第2种方法，在性能上有明显的弱势。因此，写数据时应该采用第2种方法。

Scan 时指定 StartKey 和 EndKey

一个有确切范围的Scan，在性能上会带来较大的好处。

代码示例：

```
Scan scan = new Scan();
scan.addColumn(Bytes.toBytes("familyname"),Bytes.toBytes("columnname"));
scan.setStartRow( Bytes.toBytes("rowA")); // 假设起始Key为rowA
scan.setStopRow( Bytes.toBytes("rowB")); // 假设EndKey为rowB
for(Result result : demoTable.getScanner(scan)) {
// process Result instance
}
```

不要关闭 WAL

WAL是Write-Ahead-Log的简称，是指数据在入库之前，首先会写入到日志文件中，借此来确保数据的安全性。

WAL功能默认是开启的，但是，在Put类中提供了关闭WAL功能的接口：

public void setWriteToWAL(boolean write)

因此，不建议调用该方法将WAL关闭（即将writeToWAL设置为False），因为可能会造成最近1S（该值由RegionServer端的配置参数“hbase.regionserver.optionallogflushinterval”决定，默认为1S）内的数据丢失。但如果在实际应用中，对写入的速率要求很高，并且可以容忍丢失最近1S内的数据的话，可以将该功能关闭。

创建一张表或 Scan 时设定 blockcache 为 true

HBase客户端建表和scan时，设置blockcache=true。需要根据具体的应用需求来设定它的值，这取决于有些数据是否会被反复的查询到，如果存在较多的重复记录，将这个值设置为true可以提升效率，否则，建议关闭。

建议按默认配置，默认就是true，只要不强制设置成false就可以，例如：

```
HColumnDescriptor fieldADesc = new HColumnDescriptor("value".getBytes());
fieldADesc.setBlockCacheEnabled(false);
```

HBase 不支持条件查询和 Orderby 等查询方法，存储按照字典排序，读取只支持 Rowkey 扫描

设计时应避免HBase随机查找、排序的应用场景。

业务表设计建议

1. 预分Region，使Region分布均匀，提高并发
2. 避免过多的热点Region。根据应用场景，可考虑将时间因素引入Rowkey。
3. 同时访问的数据尽量连续存储。同时读取的数据相邻存储；同时读取的数据存放在同一行；同时读取的数据存放在同一cell。
4. 查询频繁属性放在Rowkey前面部分。Rowkey的设计在排序上必须与主要的查询条件契合。
5. 离散度较好的属性作为RowKey组成部分。分析数据离散度特点以及查询场景，综合各种场景进行设计。
6. 存储冗余信息，提高检索性能。使用二级索引，适应更多查询场景。
7. 利用过期时间、版本个数设置等操作，让表能自动清除过期数据。

说明

在HBase中，一直在繁忙写数据的Region被称为热点Region。

5.3 HBase 应用开发示例

Configuration 可以设置的参数

为了能够建立一个HBase Client端到HBase Server端的连接，需要设置如下几个参数。

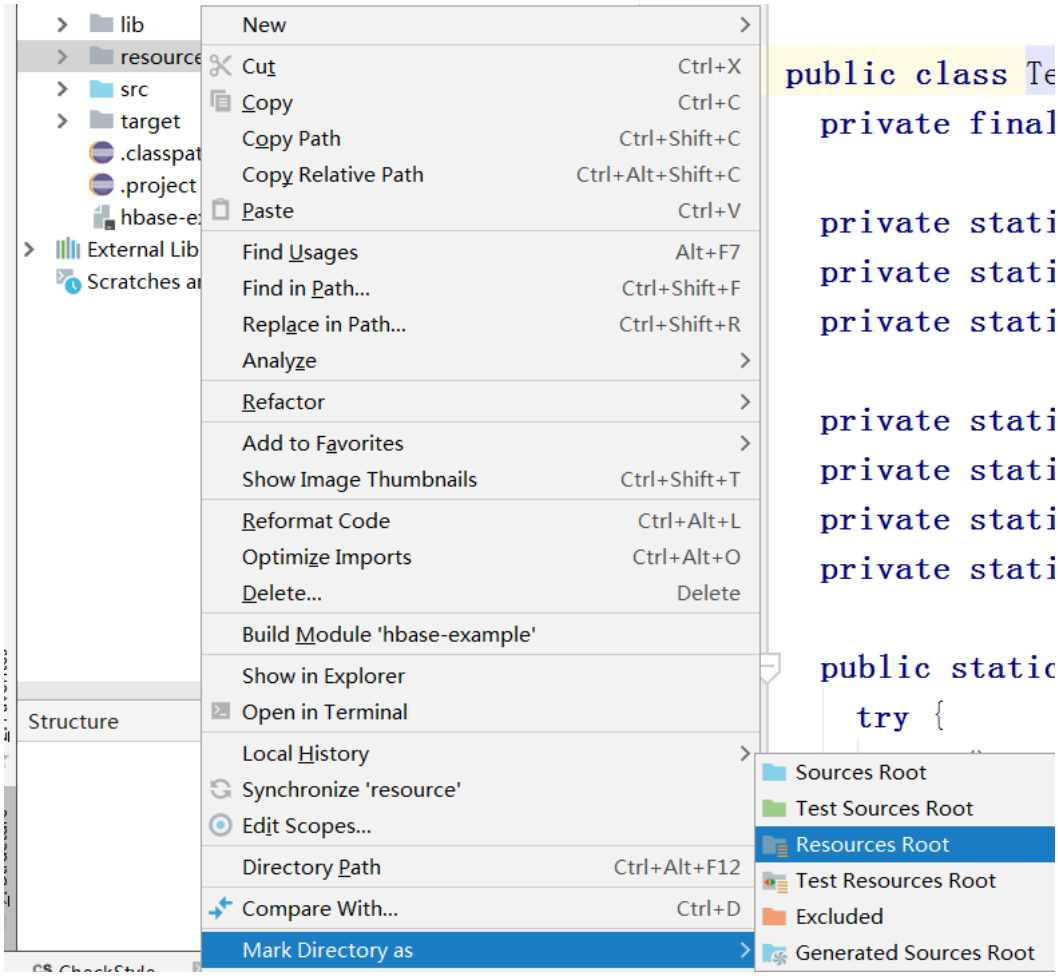
- hbase.zookeeper.quorum： Zookeeper的IP。多个Zookeeper节点的话，中间用“,”隔开。
- hbase.zookeeper.property.clientPort： Zookeeper的端口。

说明

通过HBaseConfiguration.create()创建的Configuration实例，会自动加载如下配置文件中的配置项：

- core-default.xml
- core-site.xml
- hbase-default.xml
- hbase-site.xml

这四个配置文件应该要放置在“Source Folder”下面（将一个文件夹设置为Source Folder的方法：如果在工程下面建立了一个resource的文件夹，那么，可以在该文件夹上右键鼠标，依次选择“Mark Directory as > Resources Root”即可，参考下图）。



下面是客户端可配置的一些参数集合。

说明

在通常情况下，这些值都不建议修改。

参数名	参数解释
hbase.client.pause	每次异常或者其它情况下重试等待相关的时间参数(实际等待时间将根据该值与已重试次数计算得出)。
hbase.client.retries.number	异常或者其它情况下的重试次数。
hbase.client.retries.longer.multiplier	与重试次数有关。
hbase.client.rpc.maxattempts	RPC请求不可达时的重试次数。
hbase.regionserver.lease.period	与Scanner超时时间有关（单位ms）。
hbase.client.write.buffer	在启用AutoFlush的情况下，该值不起作用。如果未启用AotoFlush的话，HBase Client端会首先缓存写入的数据，达到设定的大小后才向HBase集群下发一次写入操作。
hbase.client.scanner.caching	Scan时一次next请求获取的行数。
hbase.client.keyvalue.maxsize	一条keyvalue数据的最大值。
hbase.htable.threads.max	HTable实例中与数据操作有关的最大线程数。
hbase.client.prefetch.limit	客户端在写数据或者读取数据时，需要首先获取对应的Region所在的地址。客户端可以预缓存一些Region地址，这个参数就是与缓存的数目有关的配置。

正确设置参数的方法：

```
hbaseConfig = HBaseConfiguration.create();
//如下参数，如果在配置文件中已经存在，则无须再配置
hbaseConfig.set("hbase.zookeeper.quorum", "172.16.100.1,172.16.100.2,172.16.100.3");
hbaseConfig.set("hbase.zookeeper.property.clientPort", "2181");
```

HTablePool 在多线程写入操作中的应用

有多个写数据线程时，可以采用HTablePool。现在先简单介绍下该类的使用方法和注意点：

1. 多个写数据的线程之间，应共享同一个HTablePool实例。
2. 实例化HTablePool的时候，应要指定最大的HTableInterface实例个数maxSize，即需要通过如下构造函数实例化该类：

   public HTablePool(final Configuration config, final int maxSize)

   关于maxSize的值，可以根据写数据的线程数Threads以及涉及到的用户表个数Tables来定，理论上，不应该超过(Threads*Tables)。
3. 客户端线程通过HTablePool#getTable(tableName)的方法，获取一个表名为tableName的HTableInterface实例。

4. 同一个HTableInterface实例，在同一个时刻只能给一个线程使用。
5. 如果HTableInterface使用完了，需要调用HTablePool#putTable(HTableInterface table)方法将它放回去。

示例代码：

```
/**
* 写数据失败后需要一定的重试次数，每一次重试的等待时间，需要根据已经重试的次数而定.
*/
private static final int[] RETRIES_WAITTIME = {1, 1, 1, 2, 2, 4, 4, 8, 16, 32};
/**
* 限定的重试次数
*/
private static final int RETRIES = 10;
/**
* 失败后等待的基本时间单位
*/
private static final int PAUSE_UNIT = 1000;
private static Configuration hadoopConfig;
private static HTablePool tablePool;
private static String[] tables;
/**
* <初始化HTablePool>
* <功能详细描述>
* @param config
* @see [类、类#方法、类#成员]
*/
public static void initTablePool()
{
DemoConfig config = DemoConfig.getInstance();
if (hadoopConfig == null)
{
hadoopConfig = HBaseConfiguration.create();
hadoopConfig.set("hbase.zookeeper.quorum", config.getZookeepers());
hadoopConfig.set("hbase.zookeeper.property.clientPort", config.getZookeeperPort());
}
if (tablePool == null)
{
tablePool = new HTablePool(hadoopConfig, config.getTablePoolMaxSize());
tables = config.getTables().split(",");
}
}
public void run()
{
// 初始化HTablePool.因为这是多线程间共享的一个实例, 仅被实例化一次.
initTablePool();
for (;;)
{
Map<String, Object> data = DataStorage.takeList();
String tableName = tables[(Integer)data.get("table")];
List<Put> list = (List)data.get("list");
// 以Row为Key，保存List中所有的Put.该集合仅仅使用于写入失败时查找失败的数据记录.
// 因为从Server端仅仅返回了失败的数据记录的Row值.
Map<byte[], Put> rowPutMap = null;
// 如果失败了(哪怕是部分数据失败)，需要重试.每一次重试，都仅仅提交失败的数据条目
INNER_LOOP :
for (int retry = 0; retry < RETRIES; retry++)
{
// 从HTablePool中获取一个HTableInterface实例.用完后需要放回去.
HTableInterface table = tablePool.getTable(tableName);
try
{
table.put(list);
// 如果执行到这里，说明成功了 .
break INNER_LOOP;
}
catch (IOException e)
{
// 如果是RetriesExhaustedWithDetailsException类型的异常，
```



```
// 说明这些数据中有部分是写入失败的这通常都是因为HBase集群
// 的进程异常引起，有时也会因为有大量的Region正在被转移，
// 导致尝试一定的次数后失败.
// 如果非RetriesExhaustedWithDetailsException异常，则需要将
// list中的所有数据都要重新插入.
if (e instanceof RetriesExhaustedWithDetailsException)
{
RetriesExhaustedWithDetailsException ree =
(RetriesExhaustedWithDetailsException)e;
int failures = ree.getNumExceptions();
System.out.println("本次插入失败了[" + failures + "]条数据.");
// 第一次失败且重试时，实例化该Map.
if (rowPutMap == null)
{
rowPutMap = new HashMap<byte[], Put>(failures);
for (int m = 0; m < list.size(); m++)
{
Put put = list.get(m);
rowPutMap.put(put.getRow(), put);
}
}
//先Clear掉原数据，然后将失败的数据添加进来
list.clear();
for (int m = 0; m < failures; m++)
{
list.add(rowPutMap.get(ree.getRow(m)));
}
}
}
finally
{
// 用完之后，再将该实例放回去
tablePool.putTable(table);
}
// 如果异常了，就暂时等待一段时间.该等待应该在将HTableInterface实例放回去之后
try
{
sleep(getWaitTime(retry));
}
catch (InterruptedException e1)
{
System.out.println("Interruped");
}
}
}
}
```

Put 实例的创建

HBase是一个面向列的数据库，一行数据，可能对应多个列族，而一个列族又可以对应多个列。通常，写入数据的时候，需要指定要写入的列（含列族名称和列名称）：

	ColumnFamily01					ColumnFamily02			
	column01	column02	column03	column04	column05	column01	column02	column03	column04
Row--01									
Row--02									
Row--03									
Row--04									
Row--05									
Row--06									
Row--07									
Row--08									

如果要往HBase表中写入一行数据，需要首先构建一个Put实例。Put中包含了数据的Key值和相应的Value值，Value值可以有多个（即可以有多列值）。

有一点需要注意：在往Put实例中add一条KeyValue数据时，传入的family、qualifier、value都是字节数组。在将一个字符串转换为字节数组时，需要使用Bytes.toBytes方法，不要使用String.toBytes方法，因为后者无法保证编码，尤其是在Key或Value中出现中文字符的时候，就会出现问题。

代码示例：

```
//列族的名称为privateInfo
private final static byte[] FAMILY_PRIVATE = Bytes.toBytes("privateInfo");
//列族privateInfo中总共有两个列"name"&"address"
private final static byte[] COLUMN_NAME = Bytes.toBytes("name");
private final static byte[] COLUMN_ADDR = Bytes.toBytes("address");
/**
* <创建一个Put实例>
* <在该方法中，将会创建一个具有1个列族，2列数据的Put>
* @param rowKey Key值
* @param name 人名
* @param address 地址
* @return
* @see [类、类#方法、类#成员]
*/
public Put createPut(String rowKey, String name, String address)
{
Put put = new Put(Bytes.toBytes(rowKey));
put.add(FAMILY_PRIVATE, COLUMN_NAME, Bytes.toBytes(name));
      put.add(FAMILY_PRIVATE, COLUMN_ADDR, Bytes.toBytes(address));
return put;
}
```

HBaseAdmin 实例的创建以及常用方法

代码示例：

```
private Configuration demoConf = null;
private HBaseAdmin hbaseAdmin = null;
/**
* <构造函数>
* 需要将已经实例化好的Configuration实例传递进来
*/
public HBaseAdminDemo(Configuration conf)
{
this.demoConf = conf;
try
{
// 实例化HBaseAdmin
hbaseAdmin = new HBaseAdmin(this.demoConf);
}
catch (MasterNotRunningException e)
{
e.printStackTrace();
}
catch (ZooKeeperConnectionException e)
{
e.printStackTrace();
}
}
/**
* <一些方法使用示例>
* <更多的方法，请参考HBase接口文档>
* @throws IOException
* @throws ZooKeeperConnectionException
* @throws MasterNotRunningException
* @see [类、类#方法、类#成员]
*/
public void demo() throws MasterNotRunningException, ZooKeeperConnectionException, IOException
{
byte[] regionName = Bytes.toBytes("mrtest,jjj,1315449869513.fc41d70b84e9f6e91f9f01affdb06703.");
byte[] encodeName = Bytes.toBytes("fc41d70b84e9f6e91f9f01affdb06703");
```



```
// 重新分配一个Region.
hbaseAdmin.unassign(regionName, false);
// 主动触发Balance.
hbaseAdmin.balancer();
// 移动一个Region,第2个参数，是RegionServer的HostName+StartCode,例如：
// host187.example.com,60020,1289493121758.如果将该参数设置为null,则会随机移动该Region
hbaseAdmin.move(encodeName, null);
// 判断一个表是否存在
hbaseAdmin.tableExists("tableName");
// 判断一个表是否被激活
hbaseAdmin.isTableEnabled("tableName");
}
/**
* <快速创建一个表的方法>
* <首先创建一个HTableDescriptor实例，它里面包含了即将要创建的HTable的描述信息，同时，需要创建相应的列族。列族关联的实例是HColumnDescriptor。在本示例中，创建的列族名称为“columnName”>
* @param tableName 表名
* @return
* @see [类、类#方法、类#成员]
*/
public boolean createTable(String tableName)
{
try {
if (hbaseAdmin.tableExists(tableName)) {
return false;
}
HTableDescriptor tableDesc = new HTableDescriptor(tableName);
HColumnDescriptor fieldADesc = new HColumnDescriptor("columnName".getBytes());
fieldADesc.setBlocksize(640 * 1024);
tableDesc.addFamily(fieldADesc);
hbaseAdmin.createTable(tableDesc);
} catch (Exception e) {
e.printStackTrace();
return false;
}
return true;
}
```

5.4 附录

Scan 时的两个关键参数—Batch 和 Caching

Batch：使用scan调用next接口每次最大返回的记录数，与一次读取的**列数**有关。

Caching：一个RPC查询请求最大的返回的next数目，与一次RPC获取的**行数**有关。

首先举几个例子，来介绍这两个参数在Scan时所起到的作用：

假设表A的一个Region中存在2行（rowkey）数据，每行有1000column，且每列当前只有一个version，即每行就会有1000个key value。

-	Colu A1	Colu A2	Colu A3	Colu A4	…	Colu N1	Colu N2	Colu N3	Colu N4
Row1	-	-	-	-	…	-	-	-	-
Row2	-	-	-	-	…	-	-	-	-

- **例1：** 查询参数：不设batch，设定caching=2。
  那么，一次RPC请求，就会返回2000个KeyValue。

- **例2：** 查询参数：设定batch=500，设定caching=2
  那么，一次RPC请求，只能返回1000个KeyValue。
- **例3：** 查询参数：设定batch=300，设定caching=4。
  那么，一次RPC请求，也只能返回1000个KeyValue。

关于Batch和Caching的进一步解释：

- 一次Caching，是一次请求数据的机会。
- 同一行数据是否可以通过一次Caching读完，取决于Batch的设置，如果Batch的值小于一行的总列数，那么，这一行至少需要2次Caching才可以读完（后面的一次Caching的机会，会继续前面读取到的位置继续读取）。
- 一次Caching读取，不能跨行。如果某一行已经读完，并且Batch的值还没有达到设定的大小，也不会继续读下一行了。

那么，关于例1与例2的结果，就很好解释了：

- **例1的解释：**
  不设定Batch的时候，默认会读完该行所有的列。那么，在caching为2的时候，一次RPC请求就会返回2000个KeyValue。
- **例2的解释：**
  设定Batch为500，caching为2的情况下，也就是说，每一次Caching，最多读取500列数据。那么，第一次Caching，读取到500列，剩余的500列，会在第2次Caching中读取到。因此，两次Caching会返回1000个KeyValue。
- **例3的解释：**
  设定Batch为300，caching为4的情况下，读取完1000条数据，正好需要4次caching。因此，只能返回1000条数据。

代码示例：

```
Scan s = new Scan();
//设置查询的起始key和结束key
s.setStartRow(Bytes.toBytes("01001686138100001"));
s.setStopRow(Bytes.toBytes("01001686138100002"));
s.setBatch(1000);
s.setCaching(100);
ResultScanner scanner = null;
try {
scanner = tb.getScanner(s);
for (Result rr = scanner.next(); rr != null; rr = scanner.next()) {
for (KeyValue kv : rr.raw()) {
//显示查询的结果
System.out.println("key:" + Bytes.toString(kv.getRow())
+ "getQualifier:" + Bytes.toString(kv.getQualifier())
+ "value" + Bytes.toString(kv.getValue()));
}
}
} catch (IOException e) {
System.out.println("error!" + e.toString());
} finally {
scanner.close();
}
```


6 HDFS

6.1 HDFS 应用开发规则

HDFS NameNode 元数据存储路径

NameNode元数据信息的默认存储路径为“${BIGDATA_DATA_HOME}/namenode/data”，该参数用于确定HDFS文件系统的元数据信息的保存路径。

HDFS 需要开启 NameNode 镜像备份

NameNode的镜像备份参数为“fs.namenode.image.backup.enable”，需要设置该值为“true”，系统即可定期备份NameNode的数据。

HDFS 需要开启 DataNode 数据存储路径

DataNode默认存储路径配置为：${BIGDATA_DATA_HOME}/hadoop/dataN/dn/datadir（N≥1），N为数据存放的目录个数。

例如：${BIGDATA_DATA_HOME}/hadoop/data1/dn/datadir、${BIGDATA_DATA_HOME}/hadoop/data2/dn/datadir

设置后，数据会存储到节点上每个挂载磁盘的对应目录下面。

HDFS 提高读取写入性能方式

写入数据流程：HDFS Client收到业务数据后，从NameNode获取到数据块编号、位置信息后，联系DataNode，并将需要写入数据的DataNode建立起流水线，完成后，客户端再通过自有协议写入数据到Datanode1，再有DataNode1复制到DataNode2、DataNode3（三备份）。写完的数据，将返回确认信息给HDFS Client。

1. 合理设置块大小，如设置dfs.blocksize为 268435456（即256MB）。
2. 对于一些不可能重用的大数据，缓存在操作系统的缓存区是无用的。可将以下两参数设置为false：

   dfs.datanode.drop.cache.behind.reads和dfs.datanode.drop.cache.behind.writes

MapReduce 中间文件存放路径

MapReduce默认中间文件夹存放路径只有一个，${hadoop.tmp.dir}/mapred/local，建议修改为每个磁盘下均可存放中间文件。

例如：/hadoop/hdfs/data1/mapred/local、/hadoop/hdfs/data2/mapred/local、/hadoop/hdfs/data3/mapred/local等，不存在的目录会自动忽略。

JAVA 开发时，申请资源须在 finally 释放

申请的HDFS资源需要在try/finally中释放，而不能只在try语句之外释放，否则会导致异常情况下的资源泄漏。

HDFS 文件操作 API 概述

Hadoop中关于文件操作类基本上全部是在“org.apache.hadoop.fs”包中，这些API能够支持的操作包含：打开文件，读写文件，删除文件等。Hadoop类库中最终面向用户提供的接口类是FileSystem，该类是个抽象类，只能通过来类的get方法得到具体类。get方法存在几个重载版本，常用的是这个：

```
static FileSystem get(Configuration conf);
```

该类封装了几乎所有的文件操作，例如mkdir，delete等。综上基本可以得出操作文件的程序库框架：

```
operator()
{
    得到Configuration对象
    得到FileSystem对象
    进行文件操作
 }
```

HDFS 初始化方法

HDFS初始化是指在使用HDFS提供的API之前，需要做的必要工作。

大致过程为：加载HDFS服务配置文件，并进行Kerberos安全认证，认证通过后再实例化Filesystem，之后使用HDFS的API。此处Kerberos安全认证需要使用到的keytab文件，请提前准备。

正确示例：

```
private  void init() throws IOException {
    Configuration conf = new Configuration();
    // 读取配置文件
    conf.addResource("user-hdfs.xml");
   // 安全模式下，先进行安全认证
    if ("kerberos".equalsIgnoreCase(conf.get("hadoop.security.authentication"))) {
        String PRINCIPAL = "username.client.kerberos.principal";
        String KEYTAB = "username.client.keytab.file";
        // 设置keytab密钥文件
        conf.set(KEYTAB, System.getProperty("user.dir") + File.separator + "conf" + File.separator +
conf.get(KEYTAB));
        // 设置kerberos配置文件路径 */
        String krbfilepath = System.getProperty("user.dir") + File.separator + "conf" + File.separator +
"krb5.conf";
        System.setProperty("java.security.krb5.conf", krbfilepath);
        // 进行登录认证 */
        SecurityUtil.login(conf, KEYTAB, PRINCIPAL);
    }

    // 实例化文件系统对象
```



```
    fSystem = FileSystem.get(conf);
 }
```

HDFS 上传本地文件

通过FileSystem.copyFromLocalFile（Path src，Patch dst）可将本地文件上传到HDFS的指定位置上，其中src和dst均为文件的完整路径。

正确示例：

```
public class CopyFile {
       public static void main(String[] args) throws Exception {
       Configuration conf=new Configuration();
       FileSystem hdfs=FileSystem.get(conf);
            //本地文件
              Path src =new Path("D:\\HebutWinOS");
               //HDFS为止
       Path dst =new Path("/");
             hdfs.copyFromLocalFile(src, dst);
       System.out.println("Upload to"+conf.get("fs.default.name"));
            FileStatus files[]=hdfs.listStatus(dst);
            for(FileStatus file:files){
              System.out.println(file.getPath());
            }
       }
 }
```

HDFS 创建文件

通过"FileSystem.mkdirs（Path f）"可在HDFS上创建文件夹，其中f为文件夹的完整路径。

正确示例：

```
public class CreateDir {
    public static void main(String[] args) throws Exception{
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);
              Path dfs=new Path("/TestDir");
              hdfs.mkdirs(dfs);
     }
 }
```

查看 HDFS 文件的最后修改时间

通过FileSystem.getModificationTime()可查看指定HDFS文件的修改时间。

正确示例：

```
    public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);
        Path fpath =new Path("/user/hadoop/test/file1.txt");
        FileStatus fileStatus=hdfs.getFileStatus(fpath);
        long modiTime=fileStatus.getModificationTime();
        System.out.println("file1.txt的修改时间是"+modiTime);
   }
```

读取 HDFS 某个目录下的所有文件

通过FileStatus.getPath（ ）可查看指定HDFS中某个目录下所有文件。

正确示例：


```
public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);
        Path listf =new Path("/user/hadoop/test");

        FileStatus stats[]=hdfs.listStatus(listf);
        for(int i = 0; i < stats.length; ++i) {
                System.out.println(stats[i].getPath().toString());
            }
        hdfs.close();
 }
```

查找某个文件在 HDFS 集群的位置

通过FileSystem.getFileBlockLocation（FileStatus file，long start，long len）可查找指定文件在HDFS集群上的位置，其中file为文件的完整路径，start和len来标识查找文件的路径。

正确示例：

```
public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);
        Path fpath=new Path("/user/hadoop/cygwin");

        FileStatus filestatus = hdfs.getFileStatus(fpath);
        BlockLocation[] blkLocations = hdfs.getFileBlockLocations(filestatus, 0, filestatus.getLen());

        int blockLen = blkLocations.length;
        for(int i=0;i < blockLen；i++){
            String[] hosts = blkLocations[i].getHosts();
            System.out.println("block_"+i+"_location:"+hosts[0]);
        }
   }
```

获取 HDFS 集群上所有节点名称信息

通过DatanodeInfo.getHostName（ ）可获取HDFS集群上的所有节点名称。

正确示例：

```
public static void main(String[] args) throws Exception {
       Configuration conf=new Configuration();
       FileSystem fs=FileSystem.get(conf);

       DistributedFileSystem hdfs = (DistributedFileSystem)fs;
       DatanodeInfo[] dataNodeStats = hdfs.getDataNodeStats();
       for(int i=0;i < dataNodeStats.length;i++){
           System.out.println("DataNode_"+i+"_Name:"+dataNodeStats[i].getHostName());
       }
  }
```

多线程安全登录方式

如果有多线程进行login的操作，当应用程序第一次登录成功后，所有线程再次登录时应该使用relogin的方式。

login的代码样例：

```
 private Boolean login(Configuration conf){
   boolean flag = false;
   UserGroupInformation.setConfiguration(conf);
   try {
         UserGroupInformation.loginUserFromKeytab(conf.get(PRINCIPAL), conf.get(KEYTAB));
         System.out.println("UserGroupInformation.isLoginKeytabBased(): "
```



```
+UserGroupInformation.isLoginKeytabBased());
          flag = true;
        } catch (IOException e) {
          e.printStackTrace();
        }
      return flag;
  }
```

relogin的代码样例：

```
public Boolean relogin(){
        boolean flag = false;
        try {
            UserGroupInformation.getLoginUser().reloginFromKeytab();
            System.out.println("UserGroupInformation.isLoginKeytabBased(): "
+UserGroupInformation.isLoginKeytabBased());
            flag = true;
          } catch (IOException e) {
            e.printStackTrace();
          }
        return flag;
    }
```

警告

多次重复登录会导致后建立的会话对象覆盖掉之前登录建立的，将会导致之前建立的会话无法被维护监控，最终导致会话超期后部分功能不可用。

6.2 HDFS 应用开发建议

HDFS 的读写文件注意点

HDFS不支持随机读和写。

HDFS追加文件内容只能在文件末尾添加，不能随机添加。

只有存储在HDFS文件系统中的数据才支持append，edit.log以及数据元文件不支持Append。Append追加文件时，需要将“hdfs-site.xml”中的“dfs.support.append”参数值设置为true。

说明

- “dfs.support.append”参数在开源社区版本中默认值是关闭，在FusionInsight版本默认值是开启。
- 该参数为服务器端参数。建议开启，开启后才能使用Append功能。
- 不适用HDFS场景可以考虑使用其他方式来存储数据，如HBase。

HDFS 不适用于存储大量小文件

HDFS不适用于存储大量的小文件，因为大量小文件的元数据会占用NameNode的大量内存。

HDFS 中数据的备份数量 3 份即可

DataNode数据备份数量3份即可，增加备份数量不能提升系统效率，只会提升系统数据的安全系数；在某个节点损坏时，该节点上的数据会被均衡到其他节点上。

HDFS 定期镜像备份

NameNode的镜像备份参数为“fs.namenode.image.backup.enable”，将设置该值为“true”，系统即可定期备份NameNode的数据。

提供数据可靠性相关操作

在调用write函数写入数据时，HDFS客户端并不会将数据写入HDFS，而是缓存在客户端内存中，此时若客户端异常、断电，则数据丢失。对于有高可靠要求的数据，应该写完后，调用hflush将数据刷新到HDFS侧。

7 Hive

7.1 Hive 应用开发规则

Hive JDBC 驱动的加载

客户端程序以JDBC的形式连接HiveServer时，需要首先加载Hive的JDBC驱动类org.apache.hive.jdbc.HiveDriver。

故在客户端程序的开始，必须先使用当前类加载器加载该驱动类。

如果classpath下没有相应的jar包，则客户端程序抛出Class Not Found异常并退出。

如下：

```
Class.forName("org.apache.hive.jdbc.HiveDriver").newInstance();
```

获取数据库连接

使用JDK的驱动管理类java.sql.DriverManager来获取一个Hive的数据库连接。

Hive的数据库URL为url="jdbc:hive2://xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver;sasl.qop=auth-conf;auth=KERBEROS;principal=hive/hadoop.hadoop.com@HADOOP.COM;user.principal=hive/hadoop.hadoop.com;user.keytab=conf/hive.keytab";

以上已经经过安全认证，所以Hive数据库的用户名和密码为null或者空。

如下：

```
// 建立连接
connection = DriverManager.getConnection(url, "", "");
```

执行 HQL

执行HQL，注意HQL不能以";"结尾。

正确示例：

```
String sql = "SELECT COUNT(*) FROM employees_info";
Connection connection = DriverManager.getConnection(url, "", "");
```



```
PreparedStatement statement = connection.prepareStatement(sql);
resultSet = statement.executeQuery();
```

错误示例：

```
String sql = "SELECT COUNT(*) FROM employees_info;";
Connection connection = DriverManager.getConnection(url, "", "");
PreparedStatement statement = connection.prepareStatement(sql);
resultSet = statement.executeQuery();
```

关闭数据库连接

客户端程序在执行完HQL之后，注意关闭数据库连接，以免内存泄露，同时这是一个良好的编程习惯。

需要关闭JDK的两个对象statement和connection。

如下：

```
finally {
        if (null != statement) {
            statement.close();
        }

        // 关闭JDBC连接
        if (null != connection) {
            connection.close();
        }
    }
```

HQL 语法规则之判空

判断字段是否为“空”，即没有值，使用“is null”；判断不为空，即有值，使用“is not null”。

要注意的是，在HQL中String类型的字段若是空字符串， 即长度为0，那么对它进行IS NULL的判断结果是False。此时应该使用“col = '' ”来判断空字符串；使用“col != '' ”来判断非空字符串。

正确示例：

```
select * from default.tbl_src where id is null;
select * from default.tbl_src where id is not null;
select * from default.tbl_src where name = '';
select * from default.tbl_src where name != '';
```

错误示例：

```
select * from default.tbl_src where id = null;
select * from default.tbl_src where id != null;
select * from default.tbl_src where name is null;
select * from default.tbl_src where name is not null;
```

注：表tbl_src的id字段为Int类型，name字段为String类型。

客户端配置参数需要与服务端保持一致

当集群的Hive、YARN、HDFS服务端配置参数发生变化时，客户端程序对应的参数会被改变，用户需要重新审视在配置参数变更之前提交到HiveServer的配置参数是否和服务端配置参数一致，如果不一致，需要用户在客户端重新调整并提交到HiveServer。例如下面的示例中，如果修改了集群中的YARN配置参数时，Hive客户端、示例程序都需要审视并修改之前已经提交到HiveServer的配置参数：

初始状态：

集群YARN的参数配置如下：

```
mapreduce.reduce.java.opts=-Xmx2048M
```

客户端的参数配置如下：

```
mapreduce.reduce.java.opts=-Xmx2048M
```

集群YARN修改后，参数配置如下：

```
mapreduce.reduce.java.opts=-Xmx1024M
```

如果此时客户端程序不做调整修改，则还是以客户端参数有效，会导致reducer内存不足而使MR运行失败。

多线程安全登录方式

如果有多线程进行login的操作，当应用程序第一次登录成功后，所有线程再次登录时应该使用relogin的方式。

login的代码样例：

```
 private Boolean login(Configuration conf){
   boolean flag = false;
   UserGroupInformation.setConfiguration(conf);

   try {
     UserGroupInformation.loginUserFromKeytab(conf.get(PRINCIPAL), conf.get(KEYTAB));
     System.out.println("UserGroupInformation.isLoginKeytabBased(): "
+UserGroupInformation.isLoginKeytabBased());
     flag = true;
   } catch (IOException e) {
     e.printStackTrace();
   }
   return flag;

 }
```

relogin的代码样例：

```
public Boolean relogin(){
       boolean flag = false;
       try {

         UserGroupInformation.getLoginUser().reloginFromKeytab();
         System.out.println("UserGroupInformation.isLoginKeytabBased(): "
+UserGroupInformation.isLoginKeytabBased());
         flag = true;
       } catch (IOException e) {
          e.printStackTrace();
       }
       return flag;
   }
```

使用 WebHCat 的 REST 接口以 Streaming 方式提交 MR 任务的前置条件

本接口需要依赖hadoop的streaming包，在以Streaming方式提交MR任务给WebHCat前，需要将“hadoop-streaming-2.7.0.jar”包上传到HDFS的指定路径下：“hdfs:///apps/templeton/hadoop-streaming-2.7.0.jar”。首先登录到安装有客户端和Hive服务的节点上，以客户端安装路径为“/opt/client”为例：

source /opt/client/bigdata_env

使用kinit登录人机用户或者机机用户。

hdfs dfs -put ${BIGDATA_HOME}/FusionInsight_HD_8.1.0.1/FusionInsight-Hadoop-*/hadoop/share/hadoop/tools/lib/hadoop-streaming-*.jar /apps/templeton/

其中**/apps/templeton**/需要根据不同的实例进行修改，默认实例使用**/apps/templeton/**，Hive1实例使用**/apps1/templeton/**，以此类推。

避免对同一张表同时进行读写操作

目前的版本中，Hive不支持并发操作，需要避免对同一张表同时进行读写操作，否则会出现查询结果不准确，甚至任务失败的情况。

分桶表不支持 insert into

分桶表（bucket table）不支持insert into，仅支持insert overwrite，否则会导致文件个数与桶数不一致。

使用 WebHCat 的部分 REST 接口的前置条件

WebHCat的部分REST接口使用依赖于MapReduce的JobHistoryServer实例，具体接口如下：

- mapreduce/jar(POST)
- mapreduce/streaming(POST)
- hive(POST)
- jobs(GET)
- jobs/:jobid(GET)
- jobs/:jobid(DELETE)

Hive 授权说明

Hive授权（数据库、表或者视图）推荐通过Manager授权界面进行授权，不推荐使用命令行授权，除了“alter databases databases_name set owner='user_name'”场景以外。

不允许创建 Hive on HBase 的分区表

Hive on HBase表将实际数据存储在HBase上。由于HBase会将表划分为多个分区，将分区散列在RegionServer上，因此不允许在Hive中创建Hive on HBase分区表。

Hive on HBase 表不支持 INSERT OVERWRITE

HBase中使用rowkey作为一行记录的唯一标识。在插入数据时，如果rowkey相同，则HBase会覆盖该行的数据。如果在Hive中对一张Hive on HBase表执行INSERT OVERWRITE，会将相同rowkey的行进行覆盖，不相关的数据不会被覆盖。

7.2 Hive 应用开发建议

HQL 编写之隐式类型转换

查询语句使用字段的值做过滤时，不建议通过Hive自身的隐式类型转换来编写HQL。因为隐式类型转换不利于代码的阅读和移植。

建议示例：

```
select * from default.tbl_src where id = 10001;
select * from default.tbl_src where name = 'TestName';
```

不建议示例：

```
select * from default.tbl_src where id = '10001';
select * from default.tbl_src where name = TestName;
```

说明

表tbl_src的id字段为Int类型，name字段为String类型。

HQL 编写之对象名称长度

HQL的对象名称，包括表名、字段名、视图名、索引名等，其长度建议不要超过30个字节。

Oracle中任何对象名称长度不允许超过30个字节，超过时会报错。PT为了兼容Oracle，对对象的名称进行了限制，不允许超过30个字节。

太长不利于阅读、维护、移植。

HQL 编写之记录个数统计

统计某个表所有的记录个数，建议使用“select count(1) from table_name”。

统计某个表某个字段有效的记录个数，建议使用“select count(column_name) from table_name”。

JDBC 超时限制

Hive提供的JDBC实现有超时限制，默认是5分钟，用户可以通过**java.sql.DriverManager.setLoginTimeout(int seconds)**设置，*seconds*的单位为秒。

UDF 管理

建议由管理员创建永久UDF，避免每次使用时都去add jar，和重新定义UDF。

Hive的UDF会有一些默认属性，比如“deterministic”默认为“true”（同一个输入会返回同一个结果），“stateful”（是否有状态，默认为“true”）。当用户实现的自定义UDF内部实现了汇总等，需要在类上加上相应的注解，例如如下类：

```
@UDFType(deterministic = false)
Public class MyGenericUDAFEvaluator implements Closeable {
```

表分区优化建议

1. 当数据量较大，且经常需要按天统计时，建议使用分区表，按天存放数据。
2. 为了避免在插入动态分区数据的过程中，产生过多的小文件，在执行插入时，在分区字段上加上distribute by。

存储文件格式优化建议

Hive支持多种存储格式，比如TextFile，RCFile，ORC，Sequence，Parquet等。为了节省存储空间，或者大部分时间只查询其中的一部分字段时，可以在建表时使用列式存储(比如ORC文件)。

7.3 Hive 应用开发示例

JDBC 二次开发示例代码一

以下示例代码主要功能如下：

1. 在JDBC URL地址中提供用户名和密钥文件路径，程序自动完成安全登录、建立Hive连接。
2. 执行创建表、查询和删除三类HQL语句。

```
package com.huawei.bigdata.hive.example;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.util.Properties;


import org.apache.hadoop.conf.Configuration;
import com.huawei.bigdata.security.LoginUtil;

public class JDBCExample {
  private static final String HIVE_DRIVER = "org.apache.hive.jdbc.HiveDriver";

  private static final String ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME = "Client";
  private static final String ZOOKEEPER_SERVER_PRINCIPAL_KEY = "zookeeper.server.principal";
  private static final String ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL = "zookeeper/hadoop.hadoop.com";

  private static Configuration CONF = null;
  private static String KRB5_FILE = null;
  private static String USER_NAME = null;
  private static String USER_KEYTAB_FILE = null;

  private static String zkQuorum = null;//zookeeper节点ip和端口列表
  private static String auth = null;
  private static String sasl_qop = null;
  private static String zooKeeperNamespace = null;
  private static String serviceDiscoveryMode = null;
  private static String principal = null;
  private static String auditAddition = null;
  private static void init() throws IOException{
    CONF = new Configuration();
```

```
    Properties clientInfo = null;
    String userdir = System.getProperty("user.dir") + File.separator
        + "conf" + File.separator;
    InputStream fileInputStream = null;
    try{
      clientInfo = new Properties();
      //"hiveclient.properties"为客户端配置文件，如果使用多实例特性，需要把该文件换成对应实例客户端下的
"hiveclient.properties"
      //"hiveclient.properties"文件位置在对应实例客户端安装包解压目录下的config目录下
      String hiveclientProp = userdir + "hiveclient.properties" ;
      File propertiesFile = new File(hiveclientProp);
      fileInputStream = new FileInputStream(propertiesFile);
      clientInfo.load(fileInputStream);
    }catch (Exception e) {
      throw new IOException(e);
    }finally{
      if(fileInputStream != null){
        fileInputStream.close();
        fileInputStream = null;
      }
    }
    //zkQuorum获取后的格式为"xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181";
    //"xxx.xxx.xxx.xxx"为集群中ZooKeeper所在节点的业务IP，端口默认是2181
    zkQuorum =  clientInfo.getProperty("zk.quorum");
    auth = clientInfo.getProperty("auth");
    sasl_qop = clientInfo.getProperty("sasl.qop");
    zooKeeperNamespace = clientInfo.getProperty("zooKeeperNamespace");
    serviceDiscoveryMode = clientInfo.getProperty("serviceDiscoveryMode");
    principal = clientInfo.getProperty("principal");
    auditAddition = clientInfo.getProperty("auditAddition");
    // 设置新建用户的USER_NAME，其中"xxx"指代之前创建的用户名，例如创建的用户为user，则USER_NAME
为user
    USER_NAME = "xxx";

    if ("KERBEROS".equalsIgnoreCase(auth)) {
      // 设置客户端的keytab和krb5文件路径
      USER_KEYTAB_FILE = userdir + "user.keytab";
      KRB5_FILE = userdir + "krb5.conf";
      System.setProperty("java.security.krb5.conf", KRB5_FILE);
      System.setProperty(ZOOKEEPER_SERVER_PRINCIPAL_KEY, ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL);
    }
  }

  /**
   * 本示例演示了如何使用Hive JDBC接口来执行HQL命令<br>
   * <br>
   *
   * @throws ClassNotFoundException
   * @throws IllegalAccessException
   * @throws InstantiationException
   * @throws SQLException
   * @throws IOException
   */
  public static void main(String[] args) throws InstantiationException,
      IllegalAccessException, ClassNotFoundException, SQLException, IOException{
    // 参数初始化
    init();

    // 定义HQL，HQL为单条语句，不能包含“;”
    String[] sqls = {"CREATE TABLE IF NOT EXISTS employees_info(id INT,name STRING)",
        "SELECT COUNT(*) FROM employees_info", "DROP TABLE employees_info"};

    // 拼接JDBC URL
    StringBuilder sBuilder = new StringBuilder(
        "jdbc:hive2://").append(zkQuorum).append("/");

    if ("KERBEROS".equalsIgnoreCase(auth)) {
      sBuilder.append(";serviceDiscoveryMode=")
          .append(serviceDiscoveryMode)
```

```
            .append(";zooKeeperNamespace=")
            .append(zooKeeperNamespace)
            .append(";sasl.qop=")
            .append(sasl_qop)
            .append(";auth=")
            .append(auth)
            .append(";principal=")
            .append(principal)
            .append(";user.principal=")
            .append(USER_NAME)
            .append(";user.keytab=")
            .append(USER_KEYTAB_FILE);
    } else {
?     //普通模式
      sBuilder.append(";serviceDiscoveryMode=")
            .append(serviceDiscoveryMode)
            .append(";zooKeeperNamespace=")
            .append(zooKeeperNamespace)
            .append(";auth=none");
    }
    if (auditAddition != null && !auditAddition.isEmpty()) {
        strBuilder.append(";auditAddition=").append(auditAddition);
    }
    String url = sBuilder.toString();

    // 加载Hive JDBC驱动
    Class.forName(HIVE_DRIVER);

    Connection connection = null;
    try {
      // 获取JDBC连接
      // 如果使用的是普通模式，那么第二个参数需要填写正确的用户名，否则会以匿名用户(anonymous)登录
      connection = DriverManager.getConnection(url, "", "");

      // 建表
      // 表建完之后，如果要往表中导数据，可以使用LOAD语句将数据导入表中，比如从HDFS上将数据导入表:
      // load data inpath '/tmp/employees.txt' overwrite into table employees_info;
      execDDL(connection,sqls[0]);
      System.out.println("Create table success!");

      // 查询
      execDML(connection,sqls[1]);

      // 删表
      execDDL(connection,sqls[2]);
      System.out.println("Delete table success!");
    }catch (Exception e) {
      System.out.println("Create connection failed : "  + e.getMessage());
    }
    finally {
      // 关闭JDBC连接
      if (null != connection) {
        connection.close();
      }
    }
  }

  public static void execDDL(Connection connection, String sql)
  throws SQLException {
    PreparedStatement statement = null;
    try {
      statement = connection.prepareStatement(sql);
      statement.execute();
    }
    finally {
      if (null != statement) {
        statement.close();
      }
    }
```

```
}


public static void execDML(Connection connection, String sql) throws SQLException {
  PreparedStatement statement = null;
  ResultSet resultSet = null;
  ResultSetMetaData resultMetaData = null;

  try {
    // 执行HQL
    statement = connection.prepareStatement(sql);
    resultSet = statement.executeQuery();

    // 输出查询的列名到控制台
    resultMetaData = resultSet.getMetaData();
    int columnCount = resultMetaData.getColumnCount();
    for (int i = 1; i <= columnCount; i++) {
      System.out.print(resultMetaData.getColumnLabel(i) + '\t');
    }
    System.out.println();

    // 输出查询结果到控制台
    while (resultSet.next()) {
      for (int i = 1; i <= columnCount; i++) {
        System.out.print(resultSet.getString(i) + '\t');
      }
      System.out.println();
    }
  }
  finally {
    if (null != resultSet) {
      resultSet.close();
    }

    if (null != statement) {
      statement.close();
    }
  }
}

}
```

JDBC 二次开发示例代码二

以下示例代码主要功能如下：

1. 用户自行进行安全登录，不在JDBC URL地址中提供用户和密钥文件路径，建立Hive连接。
2. 执行创建表、查询和删除三类HQL语句。

说明

程序在访问ZooKeeper时会使用jaas配置文件，例如user.hive.jaas.conf，具体信息如下。

```
Client {
com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true
keyTab="D:\\workspace\\jdbc-examples\\conf\\user.keytab"
principal="xxx@HADOOP.COM"
useTicketCache=false
storeKey=true
debug=true;
};
```

用户需要根据实际环境，修改上述配置文件的keyTab路径（绝对路径）和principal，并设置环境变量java.security.auth.login.config指向该文件所在路径。

```
package com.huawei.bigdata.hive.example;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.util.Properties;


import org.apache.hadoop.conf.Configuration;
import com.huawei.bigdata.security.LoginUtil;

public class JDBCExamplePreLogin {
  private static final String HIVE_DRIVER = "org.apache.hive.jdbc.HiveDriver";

  private static final String ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME = "Client";
  private static final String ZOOKEEPER_SERVER_PRINCIPAL_KEY = "zookeeper.server.principal";
  private static final String ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL = "zookeeper/hadoop";

  private static Configuration CONF = null;
  private static String KRB5_FILE = null;
  private static String USER_NAME = null;
  private static String USER_KEYTAB_FILE = null;

  private static String zkQuorum = null;//zookeeper节点ip和端口列表
  private static String auth = null;
  private static String sasl_qop = null;
  private static String zooKeeperNamespace = null;
  private static String serviceDiscoveryMode = null;
  private static String principal = null;
  private static String auditAddition = null;
  private static void init() throws IOException{
    CONF = new Configuration();

    Properties clientInfo = null;
    String userdir = System.getProperty("user.dir") + File.separator
        + "conf" + File.separator;
    InputStream fileInputStream = null;
    try{
      clientInfo = new Properties();
      //"hiveclient.properties"为客户端配置文件，如果使用多实例特性，需要把该文件换成对应实例客户端下的"hiveclient.properties"
      //"hiveclient.properties"文件位置在对应实例客户端安装包解压目录下的config目录下
      String hiveclientProp = userdir + "hiveclient.properties" ;
      File propertiesFile = new File(hiveclientProp);
      fileInputStream = new FileInputStream(propertiesFile);
      clientInfo.load(fileInputStream);
    }catch (Exception e) {
```

```
      throw new IOException(e);
    }finally{
      if(fileInputStream != null){
        fileInputStream.close();
        fileInputStream = null;
      }
    }
    //zkQuorum获取后的格式为"xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181";
    //"xxx.xxx.xxx.xxx"为集群中ZooKeeper所在节点的业务IP，端口默认是2181
    zkQuorum =  clientInfo.getProperty("zk.quorum");
    auth = clientInfo.getProperty("auth");
    sasl_qop = clientInfo.getProperty("sasl.qop");
    zooKeeperNamespace = clientInfo.getProperty("zooKeeperNamespace");
    serviceDiscoveryMode = clientInfo.getProperty("serviceDiscoveryMode");
    principal = clientInfo.getProperty("principal");
    auditAddition = clientInfo.getProperty("auditAddition");
    // 设置新建用户的USER_NAME，其中"xxx"指代之前创建的用户名，例如创建的用户为user，则
USER_NAME为user
    USER_NAME = "xxx";

    if ("KERBEROS".equalsIgnoreCase(auth)) {
      // 设置客户端的keytab和krb5文件路径
      USER_KEYTAB_FILE = userdir + "user.keytab";
      KRB5_FILE = userdir + "krb5.conf";

      LoginUtil.setJaasConf(ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME, USER_NAME,
USER_KEYTAB_FILE);
      LoginUtil.setZookeeperServerPrincipal(ZOOKEEPER_SERVER_PRINCIPAL_KEY,
ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL);

      // 安全模式
      // Zookeeper登录认证
      LoginUtil.login(USER_NAME, USER_KEYTAB_FILE, KRB5_FILE, CONF);
    }
  }

  /**
   * 本示例演示了如何使用Hive JDBC接口来执行HQL命令<br>
   * <br>
   *
   * @throws ClassNotFoundException
   * @throws IllegalAccessException
   * @throws InstantiationException
   * @throws SQLException
   * @throws IOException
   */
  public static void main(String[] args) throws InstantiationException,
      IllegalAccessException, ClassNotFoundException, SQLException, IOException{
    // 参数初始化
    init();

    // 定义HQL，HQL为单条语句，不能包含“;”
    String[] sqls = {"CREATE TABLE IF NOT EXISTS employees_info(id INT,name STRING)",
        "SELECT COUNT(*) FROM employees_info", "DROP TABLE employees_info"};

    // 拼接JDBC URL
    StringBuilder sBuilder = new StringBuilder(
        "jdbc:hive2://").append(zkQuorum).append("/");

    if ("KERBEROS".equalsIgnoreCase(auth)) {
      sBuilder.append(";serviceDiscoveryMode=")
              .append(serviceDiscoveryMode)
              .append(";zooKeeperNamespace=")
              .append(zooKeeperNamespace)
              .append(";sasl.qop=")
              .append(sasl_qop)
              .append(";auth=")
              .append(auth)
              .append(";principal=")
```



```
            .append(principal);
    } else {
      // 普通模式
      sBuilder.append(";serviceDiscoveryMode=")
              .append(serviceDiscoveryMode)
              .append(";zooKeeperNamespace=")
              .append(zooKeeperNamespace)
              .append(";auth=none");
    }
    if (auditAddition != null && !auditAddition.isEmpty()) {
        strBuilder.append(";auditAddition=").append(auditAddition);
    }
    String url = sBuilder.toString();

    // 加载Hive JDBC驱动
    Class.forName(HIVE_DRIVER);

    Connection connection = null;
    try {
      // 获取JDBC连接
      // 如果使用的是普通模式，那么第二个参数需要填写正确的用户名，否则会以匿名用户(anonymous)登
录
      connection = DriverManager.getConnection(url, "", "");

      // 建表
      // 表建完之后，如果要往表中导数据，可以使用LOAD语句将数据导入表中，比如从HDFS上将数据导入
表:
      // load data inpath '/tmp/employees.txt' overwrite into table employees_info;
      execDDL(connection,sqls[0]);
      System.out.println("Create table success!");

      // 查询
      execDML(connection,sqls[1]);

      // 删表
      execDDL(connection,sqls[2]);
      System.out.println("Delete table success!");
    }
    finally {
      // 关闭JDBC连接
      if (null != connection) {
        connection.close();
      }
    }
  }

  public static void execDDL(Connection connection, String sql)
  throws SQLException {
    PreparedStatement statement = null;
    try {
      statement = connection.prepareStatement(sql);
      statement.execute();
    }
    finally {
      if (null != statement) {
        statement.close();
      }
    }
  }


  public static void execDML(Connection connection, String sql) throws SQLException {
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    ResultSetMetaData resultMetaData = null;

    try {
      // 执行HQL
      statement = connection.prepareStatement(sql);
```

```
      resultSet = statement.executeQuery();

      // 输出查询的列名到控制台
      resultMetaData = resultSet.getMetaData();
      int columnCount = resultMetaData.getColumnCount();
      for (int i = 1; i <= columnCount; i++) {
        System.out.print(resultMetaData.getColumnLabel(i) + '\t');
      }
      System.out.println();

      // 输出查询结果到控制台
      while (resultSet.next()) {
        for (int i = 1; i <= columnCount; i++) {
          System.out.print(resultSet.getString(i) + '\t');
        }
        System.out.println();
      }
    }
    finally {
      if (null != resultSet) {
        resultSet.close();
      }

      if (null != statement) {
        statement.close();
      }
    }
  }

}
```

HCatalog 二次开发示例代码

以下示例代码演示了如何使用HCatalog提供的HCatInputFormat和HCatOutputFormat接口提交MapReduce任务。

```
public class HCatalogExample extends Configured implements Tool {

  public static class Map extends
        Mapper<LongWritable, HCatRecord, IntWritable, IntWritable> {
    int age;
    @Override
    protected void map(
          LongWritable key,
          HCatRecord value,
          org.apache.hadoop.mapreduce.Mapper<LongWritable, HCatRecord,
                IntWritable, IntWritable>.Context context)
          throws IOException, InterruptedException {
      age = (Integer) value.get(0);
      context.write(new IntWritable(age), new IntWritable(1));
    }
  }

  public static class Reduce extends Reducer<IntWritable, IntWritable,
  IntWritable, HCatRecord> {
    @Override
    protected void reduce(
          IntWritable key,
          java.lang.Iterable<IntWritable> values,
          org.apache.hadoop.mapreduce.Reducer<IntWritable, IntWritable,
          IntWritable, HCatRecord>.Context context)
          throws IOException, InterruptedException {
      int sum = 0;
      Iterator<IntWritable> iter = values.iterator();
      while (iter.hasNext()) {
        sum++;
        iter.next();
      }
      HCatRecord record = new DefaultHCatRecord(2);
```

```
            record.set(0, key.get());
            record.set(1, sum);

            context.write(null, record);
        }
    }

    public int run(String[] args) throws Exception {
        Configuration conf = getConf();
        String[] otherArgs = args;

        String inputTableName = otherArgs[0];
        String outputTableName = otherArgs[1];
        String dbName = "default";

        @SuppressWarnings("deprecation")
        Job job = new Job(conf, "GroupByDemo");

        HCatInputFormat.setInput(job, dbName, inputTableName);
        job.setInputFormatClass(HCatInputFormat.class);
        job.setJarByClass(HCatalogExample.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setMapOutputKeyClass(IntWritable.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setOutputKeyClass(WritableComparable.class);
        job.setOutputValueClass(DefaultHCatRecord.class);

        OutputJobInfo outputjobInfo = OutputJobInfo.create(dbName,outputTableName, null);
        HCatOutputFormat.setOutput(job, outputjobInfo);
        HCatSchema schema = outputjobInfo.getOutputSchema();
        HCatOutputFormat.setSchema(job, schema);
        job.setOutputFormatClass(HCatOutputFormat.class);

        return (job.waitForCompletion(true) ? 0 : 1);
    }
    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new HCatalogExample(), args);
        System.exit(exitCode);
    }
}
```

8 Hudi

8.1 Hudi 应用适用场景

全量数据分析：当前分析需要读取表内全量数据，可以使用Hudi的实时视图为分析引擎提供全量最新的数据。

快速数据分析：对数据最新一致性要求不高，不需要全量数据，需要较高的分析性能，可以采用Hudi的读优化视图提高读取效率。

增量数据分析：在增量ETL场景、增量OLAP等场景，可以使用Hudi的增量视图来读取最新的或者指定提交时间的增量数据，避免全量数据遍历，提高读取性能。

历史镜像数据分析：数据一直在不断变化，在需要对某一个历史时间点的数据进行分析，采用Hudi的多版本能力，指定版本读取当时的镜像数据。

8.2 Hudi 应用开发建议

目前Hudi主要适用在实时入湖和增量数据ETL的场景，针对存量的历史数据可以批量导入Hudi表。

针对增量数据基本都是新增数据，侧重于读取数据的性能场景，更适合适用COW表。

针对对入湖性能有较高要求且增量数据中含有大量新增更新数据的场景，更适合用MOR表。

对于分区键的设置，根据业务建议使用日期字段来进行分区。

Hudi实时入湖时资源配置与Kafka的partition有一定关系，在消费kafka时一个partition只能被一个executor-core消费，因此过多配置executor-core会造成一定程度的资源浪费。

Spark streaming实时入湖的消费批次参数设置需要根据实际情况，满足每批次的间隔时间稍小于消费一批次消息写入Hudi表的时间。

Hudi写入的并行度设置不宜过大，适当缩小并行度可以缩短处理时间。

9 IoTDB

9.1 IoTDB 应用开发规则

设置合理数量的存储组

设置合理数量的存储组可以带来性能的提升。既不会因为产生过多的存储文件（夹）导致频繁切换IO降低系统速度（并且会占用大量内存且出现频繁的内存-文件切换），也不会因为过少的存储文件夹（降低了并发度从而）导致写入命令阻塞。

应根据自己的数据规模和使用场景，平衡存储文件的存储组设置，以达到更好的系统性能。

所有的时间序列必须以 root 开始、以传感器作为结尾。

时间序列可以被看作产生时序数据的传感器所在的完整路径，在IoTDB中所有的时间序列必须以root开始、以传感器作为结尾。

9.2 IoTDB 应用开发建议

推荐使用原生接口 Session，避免 SQL 拼接

关于IoTDB Session接口样例，安全模式集群可参考**IoTDB Session程序**章节，普通模式集群可参考**IoTDB Session程序**章节。

根据业务情况推荐优先使用性能高的写入接口

写入接口性能由高到低排序如下：

insertTablets（多设备多行同列）>

insertTablet（单设备多行同列）>

insertRecordsOfOneDevice （单设备多行不同列）>

insertRecords(Object value) （多设备多行不同列）>

insertRecords(String value) （多设备多行不同列）>

insertRecord（单设备一行）

避免并发使用同一个客户端连接

IoTDB客户端只能连接一个IoTDBServer，大量并发使用同一个客户端会对该客户端连接的IoTDBServer造成压力，可以根据业务需求连接多个不同的客户端来达到负载均衡。

使用 SessionPool 复用连接

分布式在Session内部做了缓存，实现客户端时避免每次读写都新建Session，或者使用SessionPool进行复用连接。

查询结果集 ResultSet、SessionDataSet 使用完成后注意关闭

查询结果集ResultSet、SessionDataSet使用完成后需要关闭，否则会造成服务资源浪费。

10 Kafka

10.1 Kafka 应用开发规则

调用 Kafka API（AdminZkClient.createTopic）创建 Topic

- 对于Java开发语言，正确示例：

```
import kafka.zk.AdminZkClient;
import kafka.zk.KafkaZkClient;
import kafka.admin.RackAwareMode;
...
KafkaZkClient kafkaZkClient = KafkaZkClient.apply(zkUrl, JaasUtils.isZkSecurityEnabled(),
zkSessionTimeoutMs, zkConnectionTimeoutMs, Int.MaxValue(), Time.SYSTEM, "", "", null);
AdminZkClient adminZkClient = new AdminZkClient(kafkaZkClient);
adminZkClient.createTopic(topic, partitions, replicas, new Properties(), RackAwareMode.Enforced
$.MODULE$);
...
```

- 对于Scala开发语言，正确示例：

```
import kafka.zk.AdminZkClient;
import kafka.zk.KafkaZkClient;
...
val kafkaZkClient: KafkaZkClient = KafkaZkClient.apply(zkUrl, JaasUtils.isZkSecurityEnabled(),
zkSessionTimeoutMs, zkConnectionTimeoutMs, Int.MaxValue, Time.SYSTEM, "", "")
val adminZkClient: AdminZkClient = new AdminZkClient(kafkaZkClient)
adminZkClient.createTopic(topic, partitions, replicas)
```

Partition 的副本数不要超过节点个数

Kafka中Topic的Partition的副本是为了提升数据的可靠性而存在的，同一个Partition的副本会分布在不同的节点，因此副本数不允许超过节点个数。

Consumer 客户端的配置参数“fetch.message.max.bytes”大小

Consumer客户端的配置参数“fetch.message.max.bytes”必须大于等于Producer客户端每次产生的消息最大字节数。如果参数的值太小，可能导致Producer产生的消息无法被Consumer成功消费。

10.2 Kafka 应用开发建议

同一个组的消费者的数量建议与待消费的 Topic 下的 Partition 数保持一致

若同一个组的消费者数量多于Topic的Partition数时，会有多余的消费者一直无法消费该Topic的消息，若消费者数量少于Topic的Partition数时，并发消费得不到完全体现，因此建议两者相等。

避免写入单条记录超大的数据

单条记录超大的数据在影响处理效率的同时还可能写入失败，此时需要在初始化Kafka生产者实例时根据情况调整“max.request.size ”值，在初始化消费者实例时调整“max.partition.fetch.bytes”值。

例如，参考本例，可以将max.request.size 、max.partition.fetch.bytes配置项设置为“5252880”：

```
// 协议类型:当前支持配置为SASL_PLAINTEXT或者PLAINTEXT
props.put(securityProtocol, kafkaProc.getValues(securityProtocol, "SASL_PLAINTEXT"));
// 服务名
props.put(saslKerberosServiceName, "kafka");
props.put("max.request.size", "5252880");
// 安全协议类型
props.put(securityProtocol, kafkaProc.getValues(securityProtocol, "SASL_PLAINTEXT"));
// 服务名
props.put(saslKerberosServiceName, "kafka");
props.put("max.partition.fetch.bytes","5252880");
```

11 Mapreduce

11.1 Mapreduce 应用开发规则

继承 Mapper 抽象类实现

在Mapreduce任务的Map阶段，会执行map()及setup()方法。

正确示例：

```
public static class MapperClass extends

Mapper<Object, Text, Text, IntWritable> {
/**
* map的输入，key为原文件位置偏移量，value为原文件的一行字符数据。
* 其map的输入key，value为文件分割方法InputFormat提供，用户不设置，默认 * 使用TextInputFormat。
*/
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
//自定义的实现
}
/**
* setup()方法只在进入map任务的map()方法之前或者reduce任务的reduce()方法之前调用一次
*/
public void setup(Context context) throws IOException,
InterruptedException {
//自定义的实现
}
}
```

继承 Reducer 抽象类实现

在Mapreduce任务的Reduce阶段，会执行reduce()及setup()方法。

正确示例：

```
public static class ReducerClass extends

Reducer<Text, IntWritable, Text, IntWritable> {

/**
* @param 输入为一个key和value值集合迭代器。
* 由各个map汇总相同的key而来。reduce方法汇总相同key的个数。
* 并调用context.write(key, value)输出到指定目录。
* 其reduce的输出的key，value由Outputformat写入文件系统。
```



```
* 默认使用TextOutputFormat写入HDFS。
*/

public void reduce(Text key, Iterable<IntWritable> values,
Context context) throws IOException, InterruptedException {
//自定义实现
}

/**
* setup()方法只在进入map任务的map()方法之前或者reduce任务的reduce()方法之前调用一次。
*/

public void setup(Context context) throws IOException,
InterruptedException {

// 自定义实现，Context可以获得配置信息。

}
}
```

提交一个 Mapreduce 任务

main()方法创建一个job，指定参数，提交作业到hadoop集群。

正确示例：

```
public static void main(String[] args) throws Exception {
Configuration conf = getConfiguration();
// main方法输入参数：args[0]为样例MR作业输入路径，args[1]为样例MR作业输出路径
String[] otherArgs = new GenericOptionsParser(conf, args)
.getRemainingArgs();
if (otherArgs.length != 2) {
System.err.println("Usage: <in> <out>");
System.exit(2);
}
Job job = new Job(conf, "job name");
// 设置找到主任务所在的jar包。
job.setJar("D:\\job-examples.jar");
// job.setJarByClass(TestWordCount.class);
// 设置运行时执行map，reduce的类，也可以通过配置文件指定。
job.setMapperClass(TokenizerMapperV1.class);
job.setReducerClass(IntSumReducerV1.class);
// 设置combiner类，默认不使用，使用时通常使用和reduce一样的类，Combiner类需要谨慎使用，也可以通过
配置文件指定。
job.setCombinerClass(IntSumReducerV1.class);
// 设置作业的输出类型，也可以通过配置文件指定。
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
// 设置该job的输入输出路径，也可以通过配置文件指定。
Path outputPath = new Path(otherArgs[1]);
FileSystem fs = outputPath.getFileSystem(conf);
// 如果输出路径已存在，删除该路径。
if (fs.exists(outputPath)) {
fs.delete(outputPath, true);
}
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

11.2 Mapreduce 应用开发建议

全局使用的配置项，在 mapred-site.xml 中指定

如下给出接口所对应的mapred-site.xml中的配置项：

```
setMapperClass(Class <extends Mapper> cls) ->“mapreduce.job.map.class”
setReducerClass(Class<extends Reducer> cls) ->“mapreduce.job.reduce.class”
setCombinerClass(Class<extends Reducer> cls) ->“mapreduce.job.combine.class”
setInputFormatClass(Class<extends InputFormat> cls) ->“mapreduce.job.inputformat.class”
setJar(String jar) ->“mapreduce.job.jar”
setOutputFormat(Class< extends OutputFormat> theClass) ->“mapred.output.format.class”
setOutputKeyClass(Class<> theClass) ->“mapreduce.job.output.key.class”
setOutputValueClass(Class<> theClass) ->“mapreduce.job.output.value.class”
setPartitionerClass(Class<extends Partitioner> theClass) ->“mapred.partitioner.class”
setMapOutputCompressorClass(Class<extends CompressionCodec> codecClass)
->“mapreduce.map.output.compress”&“mapreduce.map.output.compress.codec”
setJobPriority(JobPriority prio) ->“mapreduce.job.priority”
setQueueName(String queueName) ->“mapreduce.job.queuename”
setNumMapTasks(int n) ->“mapreduce.job.maps”
setNumReduceTasks(int n) ->“mapreduce.job.reduces”
```

11.3 Mapreduce 应用开发示例

统计日志文件中本周末网购停留总时间超过 2 个小时的女性网民信息

主要分为三个部分：

1. 从原文件中筛选女性网民上网时间数据信息，通过类MapperClass继承Mapper抽象类实现。
2. 汇总每个女性上网时间，并输出时间大于两个小时的女性网民信息，通过类ReducerClass继承Reducer抽象类实现。
3. main方法提供建立一个MapReduce job，并提交MapReduce作业到hadoop集群

样例1：类MapperClass定义Mapper抽象类的map()方法和setup()方法。

```
public static class MapperClass extends

Mapper<Object, Text, Text, IntWritable> {
// 分隔符。
String delim;
// 性别筛选。
String sexFilter;
private final static IntWritable timeInfo = new IntWritable(1);
private Text nameInfo = new Text();
/**
* map的输入，key为原文件位置偏移量，value为原文件的一行字符数据。
* 其map的输入key，value为文件分割方法InputFormat提供，用户不设置，默认使用TextInputFormat。
*/
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
// 读取的一行字符串数据
String line = value.toString();
if (line.contains(sexFilter)) {
// 获取姓名
String name = line.substring(0, line.indexOf(delim));
nameInfo.set(name);
// 获取上网停留时间
String time = line.substring(line.lastIndexOf(delim),
line.length());
timeInfo.set(Integer.parseInt(time));
// map输出key，value键值对
context.write(nameInfo, timeInfo);
}
}
/**
* setup()方法只在进入map任务的map()方法之前或者reduce任务的reduce()方法之前调用一次
*/
public void setup(Context context) throws IOException,
```



```
InterruptedException {
// 通过Context可以获得配置信息。
sexFilter = delim + context.getConfiguration().get("log.sex.filter", "female") + delim;
}
}
```

样例2：类ReducerClass定义Reducer抽象类的reduce()方法。

```
public static class ReducerClass extends
Reducer<Text, IntWritable, Text, IntWritable> {
private IntWritable result = new IntWritable();
// 总时间门槛。
private int timeThreshold;
/**
* @param 输入为一个key和value值集合迭代器。
* 由各个map汇总相同的key而来。reduce方法汇总相同key的个数。
* 并调用context.write(key, value)输出到指定目录。
* 其reduce的输出的key，value由Outputformat写入文件系统。
* 默认使用TextOutputFormat写入HDFS。
*/
public void reduce(Text key, Iterable<IntWritable> values,
Context context) throws IOException, InterruptedException {
int sum = 0;
for (IntWritable val : values) {
sum += val.get();
}
// 如果时间小于门槛时间，不输出结果。
if (sum < timeThreshold) {
return;
}
result.set(sum);
// reduce输出为key：网民的信息，value：该网民上网总时间
context.write(key, result);
}
/**
* setup()方法只在进入map任务的map()方法之前或者reduce任务的reduce()方法之前调用一次。
*/
public void setup(Context context) throws IOException,
InterruptedException {
// Context可以获得配置信息。
timeThreshold = context.getConfiguration().getInt(
"log.time.threshold", 120);
}
}
```

样例3：main()方法创建一个job，指定参数，提交作业到hadoop集群。

```
public static void main(String[] args) throws Exception {
Configuration conf = getConfiguration();
// main方法输入参数：args[0]为样例MR作业输入路径，args[1]为样例MR作业输出路径
String[] otherArgs = new GenericOptionsParser(conf, args)
.getRemainingArgs();
if (otherArgs.length != 2) {
System.err.println("Usage: <in> <out>");
System.exit(2);
}
Job job = new Job(conf, "Collect Female Info");
// 设置找到主任务所在的jar包。
job.setJar("D:\\mapreduce-examples\\hadoop-mapreduce-examples\\mapreduce-examples.jar");
// job.setJarByClass(TestWordCount.class);
// 设置运行时执行map，reduce的类，也可以通过配置文件指定。
job.setMapperClass(TokenizerMapperV1.class);
job.setReducerClass(IntSumReducerV1.class);
// 设置combiner类，默认不使用，使用时通常使用和reduce一样的类，Combiner类需要谨慎使用，也可以通过
配置文件指定。
job.setCombinerClass(IntSumReducerV1.class);
// 设置作业的输出类型，也可以通过配置文件指定。
job.setOutputKeyClass(Text.class);
```



```
job.setOutputValueClass(IntWritable.class);
// 设置该job的输入输出路径，也可以通过配置文件指定。
Path outputPath = new Path(otherArgs[1]);
FileSystem fs = outputPath.getFileSystem(conf);
// 如果输出路径已存在，删除该路径。
if (fs.exists(outputPath)) {
fs.delete(outputPath, true);
}
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true)    0 : 1);
}
```


12 Spark

12.1 Spark 应用开发规则

Spark 应用中，需引入 Spark 的类

- 对于Java开发语言，正确示例：

```
// 创建SparkContext时所需引入的类。
import org.apache.spark.api.java.JavaSparkContext
// RDD操作时引入的类。
import org.apache.spark.api.java.JavaRDD
// 创建SparkConf时引入的类。
import org.apache.spark.SparkConf
```

- 对于Scala开发语言，正确示例：

```
// 创建SparkContext时所需引入的类。
import org.apache.spark.SparkContext
// RDD操作时引入的类。
import org.apache.spark.SparkContext._
// 创建SparkConf时引入的类。
import org.apache.spark.SparkConf
```

Java 与 Scala 函数有区别，在编写应用时，需要弄清楚每个函数的功能

RDD是不可改变的，也就是说，RDD的元素对象是不能更改的，因此，在用Java和Scala编写需要弄清楚每个函数的功能。下面举个例子。

场景：现有用户位置数据，按照时间排序生成用户轨迹。在Scala中，按时间排序的代码如下：

```
/* 函数实现的功能是得到某个用户的位置轨迹。
 * 参数trajectory:由两部分组成-用户名和位置点（时间，经度，维度）
 */
private def getTimesOfOneUser(trajectory: (String, Seq[(String, Float, Float)]), zone: Zone, arrive: Boolean): Int =
{
   // 先将用户位置点按时间排序
   val sorted: Seq[(String, Float, Float)] = trajectory._2.sortBy(x => x._1);
   ...
}
```

若用java实现上述功能，则需要将trajectory._2重新生成对象，而不能直接对trajectory._2进行排序操作。原因是Collections.sort(**trajectory._2**)这个操作会改变了**trajectory._2**这个对象本身，这违背了RDD元素不可更改这条规则；而Scala代码之所

以能够正常运行，是因为**sortBy( )**这个函数生成了一个新的对象，它并不对**trajectory._2**直接操作。下面分别列出java实现的正确示例和错误示例。

正确示例：

```
// 将用户的位置点从新生成一个对象。
List<Tuple3< String, Float, Float >> list = new ArrayList<Tuple3< String, Float, Float >>( trajectory._2);
// 对新对象进行排序。
Collections.sort(list);
```

错误示例：

```
// 直接对用户位置点按照时间排序。
Collections.sort(trajectory._2);
```

分布式模式下，应注意 Driver 和 Executor 之间的参数传递

在Spark编程时，总是有一些代码逻辑中需要根据输入参数来判断，这种时候往往会使用这种方式，将参数设置为全局变量，先给定一个空值（null），在main函数中，实例化SparkContext对象之前对这个变量赋值。然而，在分布式模式下，执行程序的jar包会被发送到每个Executor上执行。而该变量只在main函数的节点改变了，并未传给执行任务的函数中，因此Executor将会报空指针异常。

正确示例：

```
object Test
{
  private var testArg: String = null;
  def main(args: Array[String])
  {
    testArg = …;
    val sc: SparkContext = new SparkContext(…);

    sc.textFile(…)
    .map(x => testFun(x, testArg));
  }

  private def testFun(line: String, testArg: String): String =
  {
    testArg.split(…);
    return …;
  }
}
```

错误示例：

```
//定义对象。
object Test
{
  // 定义全局变量，赋为空值（null）；在main函数中，实例化SparkContext对象之前对这个变量赋值。
  private var testArg: String = null;
  // main函数
  def main(args: Array[String])
  {

    testArg = …;
    val sc: SparkContext = new SparkContext(…);

    sc.textFile(…)
      .map(x => testFun(x));
  }

  private def testFun(line: String): String =
  {
    testArg.split(...);
    return …;
```

```
 }
}
```

运行错误示例，在Spark的local模式下能正常运行，而在分布式模式情况下，会在蓝色代码处报错，提示空指针异常，这是由于在分布式模式下，执行程序的jar包会被发送到每个Executor上执行，当执行到testFun函数时，需要从内存中取出testArg的值，但是testArg的值只在启动main函数的节点改变了，其他节点无法获取这些变化，因此它们从内存中取出的就是初始化这个变量时的值null，这就是空指针异常的原因。

应用程序结束之前必须调用 SparkContext.stop

利用spark做二次开发时，当应用程序结束之前必须调用SparkContext.stop()。

说明

利用Java语言开发时，应用程序结束之前必须调用JavaSparkContext.stop()。

利用Scala语言开发时，应用程序结束之前必须调用SparkContext.stop()。

以Scala语言开发应用程序为例，分别介绍下正确示例与错误示例。

正确示例：

```
//提交spark作业
val sc = new SparkContext(conf)

//具体的任务
...

//应用程序结束
sc.stop()
```

错误示例：

```
//提交spark作业
val sc = new SparkContext(conf)

//具体的任务
...
```

如果不添加SparkContext.stop，YARN界面会显示失败。如图12-1，同样的任务，前一个程序是没有添加SparkContext.stop，后一个程序添加了SparkContext.stop()。

图 12-1 添加 SparkContext.stop()和不添加的区别

合理规划 AM 资源占比

任务数量较多且每个任务占用的资源较少时，可能会出现集群资源足够，提交的任务成功但是无法启动，此时可以提高AM的最大资源占比。

图 12-2 修改 AM 最大资源百分比

12.2 Spark 应用开发建议

RDD 多次使用时，建议将 RDD 持久化

RDD在默认情况下的存储级别是StorageLevel.NONE，即既不存磁盘也不放在内存中，如果某个RDD需要多次使用，可以考虑将该RDD持久化，方法如下：

调用spark.RDD中的cache()、persist()、persist(newLevel:StorageLevel)函数均可将RDD持久化，cache()和persist()都是将RDD的存储级别设置为StorageLevel.MEMORY_ONLY，persist(newLevel:StorageLevel)可以为RDD设置其他存储级别，但是要求调用该方法之前RDD的存储级别为StorageLevel.NONE或者与newLevel相同，也就是说，RDD的存储级别一旦设置为StorageLevel.NONE之外的级别，则无法改变。

如果想要将RDD去持久化，那么可以调用unpersist(blocking:Boolean = true)，该函数功能如下：

1. 将该RDD从持久化列表中移除，RDD对应的数据进入可回收状态；
2. 将RDD的存储级别重新设置为StorageLevel.NONE。

慎重选择 shuffle 过程的算子

该类算子称为宽依赖算子，其特点是父RDD的一个partition影响子RDD得多个partition，RDD中的元素一般都是<key, value>对。执行过程中都会涉及到RDD的partition重排，这个操作称为shuffle。

由于shuffle类算子存在节点之间的网络传输，因此对于数据量很大的RDD，应该尽量提取需要使用的信息，减小其单条数据的大小，然后再调用shuffle类算子。

常用的有如下几种：

- combineByKey() : RDD[(K, V)] => RDD[(K, C)]，是将RDD[(K, V)]中key相同的数据的所有value转化成为一个类型为C的值。
- groupByKey() 和reduceByKey()是combineByKey的两种具体实现，对于数据聚合比较复杂而groupByKey和reduceByKey不能满足使用需求的场景，可以使用自己定义的聚合函数作为combineByKey的参数来实现。
- distinct(): RDD[T] => RDD[T]，作用是去除重复元素的算子。其处理过程代码如下：
  ```
  map(x => (x, null)).reduceByKey((x, y) => x, numPartitions).map(_._1)
  ```
  这个过程比较耗时，尤其是数据量很大时，建议不要直接对大文件生成的RDD使用。

- join()：(RDD[(K, V)], RDD[(K, W)]) => RDD[(K, (V, W))]，作用是将两个RDD通过key做连接。

  如果RDD[(K, V)]中某个key有X个value，而RDD[(K, W)]中相同key有Y个value，那么最终在RDD[(K, (V, W))]中会生成X*Y条记录。

在业务情况允许的情况下使用高性能算子

1. 使用reduceByKey/aggregateByKey替代groupByKey。

   所谓的map-side预聚合，说的是在每个节点本地对相同的key进行一次聚合操作，类似于MapReduce中的本地combiner。 map-side预聚合之后，每个节点本地就只会有一条相同的key，因为多条相同的key都被聚合起来了。其他节点在拉取所有节点上的相同key时，就会大大减少需要拉取的数据数量，从而也就减少了磁盘IO以及网络传输开销。通常来说，在可能的情况下，建议使用reduceByKey或aggregateByKey算子来替代掉groupByKey算子。因为reduceByKey和aggregateByKey算子都会使用用户自定义的函数对每个节点本地的相同key进行预聚合。而groupByKey算子是不会进行预聚合的，全量的数据会在集群的各个节点之间分发和传输，性能相对来说比较差。

2. 使用mapPartitions替代普通map。

   mapPartitions类的算子，一次函数调用会处理一个partition所有的数据，而不是一次函数调用处理一条，性能相对来说会高一些。 但是有的时候，使用mapPartitions会出现OOM（内存溢出）的问题。因为单次函数调用就要处理掉一个partition所有的数据，如果内存不够，垃圾回收时是无法回收掉太多对象的，很可能出现OOM异常。所以使用这类操作时要慎重！

3. 使用filter之后进行coalesce操作。

   通常对一个RDD执行filter算子过滤掉RDD中较多数据后（比如30%以上的数据），建议使用coalesce算子，手动减少RDD的partition数量，将RDD中的数据压缩到更少的partition中去。因为filter之后，RDD的每个partition中都会有很多数据被过滤掉，此时如果照常进行后续的计算，其实每个task处理的partition中的数据量并不是很多，有一点资源浪费，而且此时处理的task越多，可能速度反而越慢。因此用coalesce减少partition数量，将RDD中的数据压缩到更少的partition之后，只要使用更少的task即 可处理完所有的partition。在某些场景下，对于性能的提升会有一定的帮助。

4. 使用repartitionAndSortWithinPartitions替代repartition与sort类操作。

   repartitionAndSortWithinPartitions是Spark官网推荐的一个算子，官方建议，如果需要在 repartition重分区之后，还要进行排序，建议直接使用repartitionAndSortWithinPartitions 算子。因为该算子 可以一边进行重分区的shuffle操作，一边进行排序。shuffle与sort两个操作同时进行，比先shuffle再sort来说，性能可能是要高的。

5. 使用foreachPartitions替代foreach。

   原理类似于“使用mapPartitions替代map”，也是一次函数调用处理一个partition的所有数据，而不是一次函数调用处理一条数 据。在实践中发现，foreachPartitions类的算子，对性能的提升还是很有帮助的。比如在foreach函数中，将RDD中所有数据写 MySQL，那么如果是普通的foreach算子，就会一条数据一条数据地写，每次函数调用可能就会创建一个数据库连接，此时就势必会频繁地创建和销毁数据库连接，性能是非常低下；但是如果用foreachPartitions算子一次性处理一个partition的数据，那么对于每个 partition，只要创建一个数据库连接即可，然后执行批量插入操作，此时性能是比较高的。

RDD 共享变量

在应用开发中，一个函数被传递给Spark操作（例如map和reduce），在一个远程集群上运行，它实际上操作的是这个函数用到的所有变量的独立拷贝。这些变量会被拷贝到每一台机器。通常看来，在任务之间中，读写共享变量显然不够高效。Spark为两种常见的使用模式，提供了两种有限的共享变量：广播变量、累加器。

在对性能要求比较高的场景下，可以使用 Kryo 优化序列化性能

Spark提供了两种序列化实现：

org.apache.spark.serializer.KryoSerializer：性能好，兼容性差

org.apache.spark.serializer.JavaSerializer：性能一般，兼容性好

使用：conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

说明

为什么不默认使用Kryo序列化?

Spark默认使用的是Java的序列化机制，也就是ObjectOutputStream/ObjectInputStream API来进行序列化和反序列化。但是Spark同时支持使用Kryo序列化库，Kryo序列化类库的性能比Java序列化类库的性能要高很多。官方介 绍，Kryo序列化机制比Java序列化机制，性能高10倍左右。Spark之所以默认没有使用Kryo作为序列化类库，是因为Kryo要求要注册所有需要进行序列化的自定义类型，因此对于开发者来说，这种方式比较麻烦。

Spark Streaming 性能优化建议

1. 设置合理的批处理时间(batchDuration)。
2. 设置合理的数据接收并行度。
   - 设置多个Receiver接收数据。
   - 设置合理的Receiver阻塞时间。
3. 设置合理的数据处理并行度。
4. 使用Kryo系列化。
5. 内存调优。
   - 设置持久化级别减少GC开销。
   - 使用并发的标记-清理GC算法减少GC暂停时间。

13 Yarn

13.1 Yarn 应用开发规则

使用 YarnClient.createYarnClient()创建客户端

不建议直接使用protocol接口ClientRMProxy.createRMProxy(config,ApplicationClientProtocol.class)创建客户端。

Application master 使用异步接口AMRMClientAsync.createAMRMClientAsync()与 ResourceManager 交互

不建议直接使用protocol接口ClientRMProxy.createRMProxy(config,ApplicationMasterProtocol.class) 创建application master与resource manager交互的客户端。

Application master 使用异步接口AMRMClientAsync.createAMRMClientAsync()与 NodeManager 交互

不建议直接使用接口ContainerManagementProtocolProxy创建application master与node manager交互的客户端。

多线程安全登录方式

如果有多线程进行login的操作，当应用程序第一次登录成功后，所有线程再次登录时应该使用relogin的方式。

login的代码样例：

```
private Boolean login(Configuration conf){
   boolean flag = false;
   UserGroupInformation.setConfiguration(conf);
       try {
     UserGroupInformation.loginUserFromKeytab(conf.get(PRINCIPAL), conf.get(KEYTAB));
     System.out.println("UserGroupInformation.isLoginKeytabBased(): "
+UserGroupInformation.isLoginKeytabBased());
     flag = true;
   } catch (IOException e) {
     e.printStackTrace();
   }
```



```
    return flag;
 }
```

relogin的代码样例：

```
public Boolean relogin(){
        boolean flag = false;
        try {
            UserGroupInformation.getLoginUser().reloginFromKeytab();
            System.out.println("UserGroupInformation.isLoginKeytabBased(): "
+UserGroupInformation.isLoginKeytabBased());
            flag = true;
        } catch (IOException e) {
          e.printStackTrace();
        }
        return flag;
}
```