

尚硅谷大数据技术之 HBase

（作者：尚硅谷研究院）

版本：V2.0

第 1 章 HBase 简介

1.1 HBase 定义

Apache HBase™ 是以 hdfs 为数据存储的，一种分布式、可扩展的 NoSQL 数据库。

1.2 HBase 数据模型

HBase 的设计理念依据 Google 的 BigTable 论文，论文中对于数据模型的首句介绍。

Bigtable 是一个稀疏的、分布式的、持久的多维排序 map。

之后对于映射的解释如下：

该映射由行键、列键和时间戳索引；映射中的每个值都是一个未解释的字节数组。

最终 HBase 关于数据模型和 BigTable 的对应关系如下：

HBase 使用与 Bigtable 非常相似的数据模型。用户将数据行存储在带标签的表中。数据行具有可排序的键和任意数量的列。该表存储稀疏，因此如果用户喜欢，同一表中的行可以具有疯狂变化的列。

最终理解 HBase 数据模型的关键在于稀疏、分布式、多维、排序的映射。其中映射 map 指代非关系型数据库的 key-Value 结构。

1.2.1 HBase 逻辑结构

HBase 可以用于存储多种结构的数据，以 JSON 为例，存储的数据原貌为：

```
{
  "row_key1":{
    "personal_info":{
      "name":"zhangsan",
      "city":"北京",
      "phone":"131*****"
    },
    "office_info":{
      "tel":"010-1111111",
      "address":"atguigu"
    }
  },
  "row_key11":{
    "personal_info":{
      "city":"上海",
      "phone":"132*****"
    },
    "office_info":{
      "tel":"010-1111111"
    }
  }
}
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```

    }
  },
  "row_key2":{
    .....
  }
}

```



HBase 逻辑结构



存储数据**稀疏**，数据存储**多维**，不同的行具有不同的列。

数据存储整体**有序**，按照RowKey的字典序排列，RowKey为Byte数组

列	personal_info			office_info		列族
	Row Key	name	city	phone	tel	address
	row_key1	张三	北京	131*****	010-11111111	atguigu
Row key	row_key11		上海	132*****	010-11111111	
	row_key2	王五	广州		010-11111111	atguigu
	row_key3		深圳	187*****		atguigu
	row_key4	横七	大连	134*****		atguigu
	row_key5	竖八		139*****		atguigu
	row_key6	金九	武汉		010-11111111	atguigu
	row_key7	银十	保定	158*****	010-11111111	

1.2.2 HBase 物理存储结构

物理存储结构即为数据映射关系，而在概念视图的空单元格，底层实际根本不存储。



HBase 物理存储结构



StoreFile	personal_info			Timestamp	Type	Value
	Row Key	name	city	phone		
	row_key1	张三	北京	131*****		
	row_key11		上海	132*****		
	row_key2	王五	广州			

Row Key	Column Family	Column Qualifier	Timestamp	Type	Value
row_key1	personal_info	name	t1	Put	张三
row_key1	personal_info	city	t2	Put	北京
row_key1	personal_info	phone	t3	Put	131*****
row_key1	personal_info	phone	t4	Put	177*****

Timestamp
不同版本 (version) 的数据根据timestamp进行区分
读取数据默认读取最新的版本

Type
对于删除操作，其类型为 DeleteColumn

让天下没有难学的技术

1.2.3 数据模型

1) Name Space

命名空间，类似于关系型数据库的 database 概念，每个命名空间下有多个表。HBase 两个自带的命名空间，分别是 hbase 和 default，hbase 中存放的是 HBase 内置的表，default 更多 [Java - 大数据 - 前端 - python 人工智能资料下载](#)，可百度访问：[尚硅谷官网](#)

表是用户默认使用的命名空间。

2) Table

类似于关系型数据库的表概念。不同的是，HBase 定义表时只需要声明列族即可，不需要声明具体的列。因为数据存储时稀疏的，所有往 HBase 写入数据时，字段可以动态、按需指定。因此，和关系型数据库相比，HBase 能够轻松应对字段变更的场景。

3) Row

HBase 表中的每行数据都由一个 RowKey 和多个 Column（列）组成，数据是按照 RowKey 的字典顺序存储的，并且查询数据时只能根据 RowKey 进行检索，所以 RowKey 的设计十分重要。

4) Column

HBase 中的每个列都由 Column Family (列族) 和 Column Qualifier (列限定符) 进行限定，例如 info: name, info: age。建表时，只需指明列族，而列限定符无需预先定义。

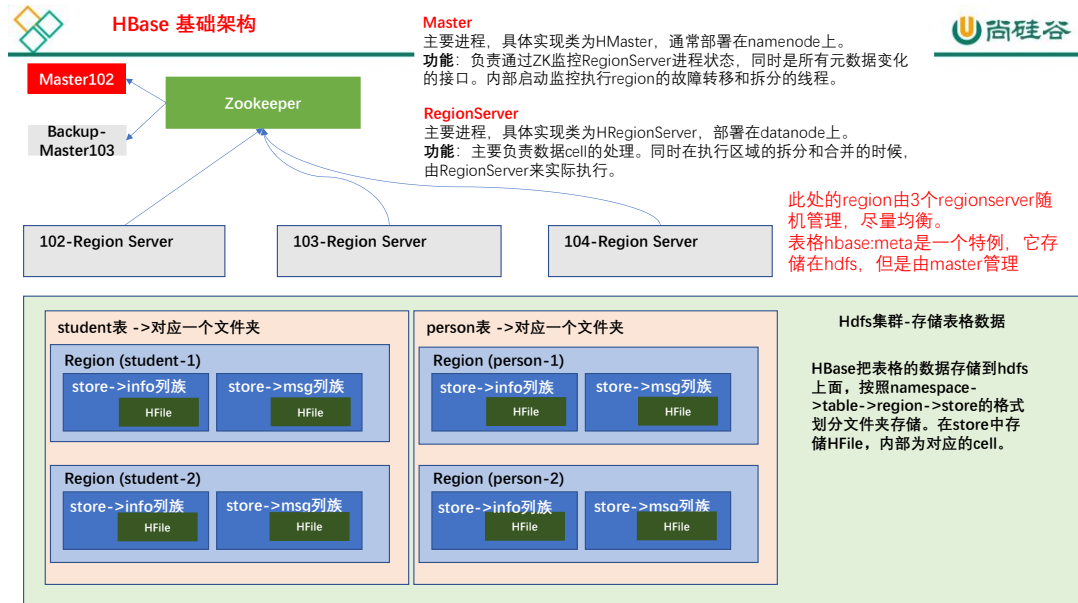
5) Time Stamp

用于标识数据的不同版本 (version)，每条数据写入时，系统会自动为其加上该字段，其值为写入 HBase 的时间。

6) Cell

由 {rowkey, column Family: column Qualifier, timestamp} 唯一确定的单元。cell 中的数据全部是字节码形式存储。

1.3 HBase 基本架构



架构角色：

1) Master

实现类为 **HMaster**，负责监控集群中所有的 RegionServer 实例。主要作用如下：

- (1) 管理元数据表格 hbase:meta，接收用户对表格创建修改删除的命令并执行
- (2) 监控 region 是否需要负载均衡，故障转移和 region 的拆分。

通过启动多个后台线程监控实现上述功能：

①LoadBalancer 负载均衡器

周期性监控 region 分布在 regionServer 上面是否均衡，由参数 hbase.balancer.period 控制周期时间，默认 5 分钟。

②CatalogJanitor 元数据管理器

定期检查和清理 hbase:meta 中的数据。meta 表内容在进阶中介绍。

③MasterProcWAL master 预写日志处理器

把 master 需要执行的任务记录到预写日志 WAL 中，如果 master 宕机，让 backupMaster 读取日志继续干。

2) Region Server

Region Server 实现类为 HRegionServer，主要作用如下：

- (1) 负责数据 cell 的处理，例如写入数据 put，查询数据 get 等
- (2) 拆分合并 region 的实际执行者，有 master 监控，有 regionServer 执行。

更多 [Java](#) -[大数据](#) -[前端](#) -[python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

3) Zookeeper

HBase 通过 Zookeeper 来做 master 的高可用、记录 RegionServer 的部署信息、并且存储有 meta 表的位置信息。

HBase 对于数据的读写操作时直接访问 Zookeeper 的，在 2.3 版本推出 Master Registry 模式，客户端可以直接访问 master。使用此功能，会加大对 master 的压力，减轻对 Zookeeper 的压力。

4) HDFS

HDFS 为 Hbase 提供最终的底层数据存储服务，同时为 HBase 提供高容错的支持。

第 2 章 HBase 快速入门

2.1 HBase 安装部署

2.1.1 Zookeeper 正常部署

首先保证 Zookeeper 集群的正常部署，并启动之。

```
[atguigu@hadoop102 zookeeper-3.5.7]$ bin/zkServer.sh start
[atguigu@hadoop103 zookeeper-3.5.7]$ bin/zkServer.sh start
[atguigu@hadoop104 zookeeper-3.5.7]$ bin/zkServer.sh start
```

2.1.2 Hadoop 正常部署

Hadoop 集群的正常部署并启动。

```
[atguigu@hadoop102 hadoop-3.1.3]$ sbin/start-dfs.sh
[atguigu@hadoop103 hadoop-3.1.3]$ sbin/start-yarn.sh
```

2.1.3 HBase 的解压

1) 解压 Hbase 到指定目录

```
[atguigu@hadoop102 software]$ tar -zxvf hbase-2.4.11-bin.tar.gz -C /opt/module/
[atguigu@hadoop102 software]$ mv /opt/module/hbase-2.4.11 /opt/module/hbase
```

2) 配置环境变量

```
[atguigu@hadoop102 ~]$ sudo vim /etc/profile.d/my_env.sh
```

添加

```
#HBASE_HOME
export HBASE_HOME=/opt/module/hbase
export PATH=$PATH:$HBASE_HOME/bin
```

3) 使用 source 让配置的环境变量生效

```
[atguigu@hadoop102 module]$ source /etc/profile.d/my_env.sh
```

2.1.4 HBase 的配置文件

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

1) hbase-env.sh 修改内容, 可以添加到最后:

```
export HBASE_MANAGES_ZK=false
```

2) hbase-site.xml 修改内容:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>

  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>hadoop102,hadoop103,hadoop104</value>
    <description>The directory shared by RegionServers.
    </description>
  </property>

  <!--      <property>-->
  <!--      <name>hbase.zookeeper.property.dataDir</name>-->
  <!--      <value>/export/zookeeper</value>-->
  <!--      <description> 记得修改 zk 的配置文件 -->
  <!--      zk 的信息不能保存到临时文件夹-->
  <!--      </description>-->
  <!--      </property>-->

  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://hadoop102:8020/hbase</value>
    <description>The directory shared by RegionServers.
    </description>
  </property>

  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>

</configuration>
```

3) regionservers

```
hadoop102
hadoop103
hadoop104
```

4) 解决 HBase 和 Hadoop 的 log4j 兼容性问题, 修改 HBase 的 jar 包, 使用 Hadoop 的 jar 包

```
[atguigu@hadoop102 hbase]$ mv /opt/module/hbase/lib/client-facing-thirdparty/slf4j-reload4j-1.7.33.jar /opt/module/hbase/lib/client-facing-thirdparty/slf4j-reload4j-1.7.33.jar.bak
```

2.1.5 HBase 远程发送到其他集群

```
[atguigu@hadoop102 module]$ xsync hbase/
```

2.1.6 HBase 服务的启动

1) 单点启动

```
[atguigu@hadoop102 hbase]$ bin/hbase-daemon.sh start master
[atguigu@hadoop102 hbase]$ bin/hbase-daemon.sh start regionserver
```

2) 群启

```
[atguigu@hadoop102 hbase]$ bin/start-hbase.sh
```

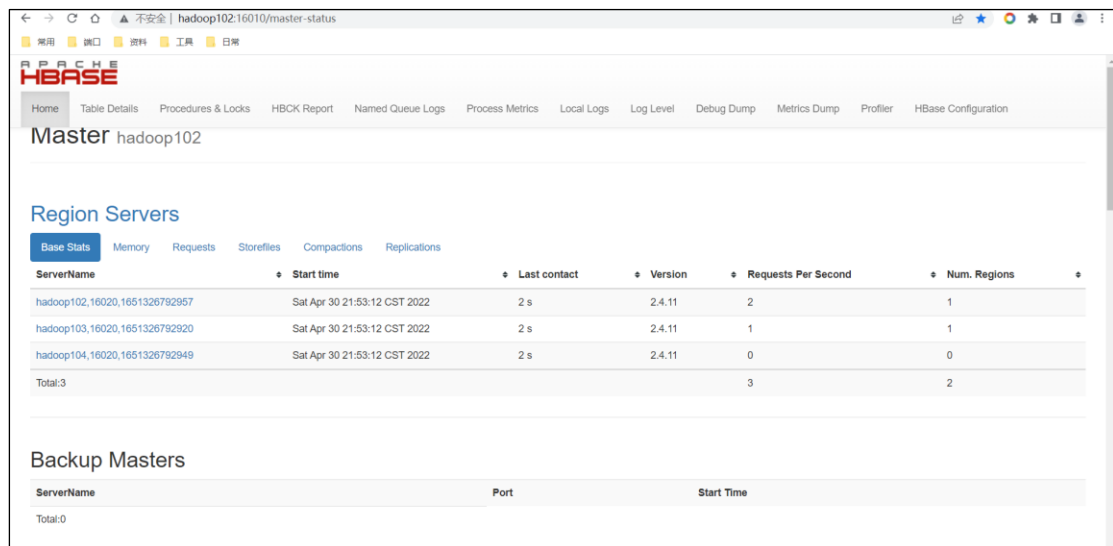
3) 对应的停止服务

```
[atguigu@hadoop102 hbase]$ bin/stop-hbase.sh
```

2.1.7 查看 HBase 页面

启动成功后，可以通过“host:port”的方式来访问 HBase 管理页面，例如：

<http://hadoop102:16010>



The screenshot shows the HBase Master web interface for hadoop102. The top navigation bar includes links like Home, Table Details, Procedures & Locks, HBase Report, Named Queue Logs, Process Metrics, Local Logs, Log Level, Debug Dump, Metrics Dump, Profiler, and HBase Configuration. The main content area is titled 'Master hadoop102' and features a 'Region Servers' section with a table of active servers. Below this is a 'Backup Masters' section with an empty table.

ServerName	Start time	Last contact	Version	Requests Per Second	Num. Regions
hadoop102,16020,1651326792957	Sat Apr 30 21:53:12 CST 2022	2 s	2.4.11	2	1
hadoop103,16020,1651326792920	Sat Apr 30 21:53:12 CST 2022	2 s	2.4.11	1	1
hadoop104,16020,1651326792949	Sat Apr 30 21:53:12 CST 2022	2 s	2.4.11	0	0
Total:3				3	2

ServerName	Port	Start Time
Total:0		

2.1.8 高可用（可选）

在 HBase 中 HMaster 负责监控 HRegionServer 的生命周期，均衡 RegionServer 的负载，如果 HMaster 挂掉了，那么整个 HBase 集群将陷入不健康的状态，并且此时的工作状态并不会维持太久。所以 HBase 支持对 HMaster 的高可用配置。

1) 关闭 HBase 集群（如果没有开启则跳过此步）

```
[atguigu@hadoop102 hbase]$ bin/stop-hbase.sh
```

2) 在 conf 目录下创建 backup-masters 文件

```
[atguigu@hadoop102 hbase]$ touch conf/backup-masters
```

3) 在 backup-masters 文件中配置高可用 HMaster 节点

```
[atguigu@hadoop102 hbase]$ echo hadoop103 > conf/backup-masters
```

4) 将整个 conf 目录 scp 到其他节点

```
[atguigu@hadoop102 hbase]$ xsync conf
```

5) 重启 hbase, 打开页面测试查看

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

<http://hadoo102:16010>

2.2 HBase Shell 操作

2.2.1 基本操作

1) 进入 HBase 客户端命令行

```
[atguigu@hadoop102 hbase]$ bin/hbase shell
```

2) 查看帮助命令

能够展示 HBase 中所有能使用的命令，主要使用的命令有 namespace 命令空间相关，DDL 创建修改表格，DML 写入读取数据。

```
hbase:001:0> help
```

2.2.2 namespace

1) 创建命名空间

使用特定的 help 语法能够查看命令如何使用。

```
hbase:002:0> help 'create_namespace'
```

2) 创建命名空间 bigdata

```
hbase:003:0> create_namespace 'bigdata'
```

3) 查看所有的命名空间

```
hbase:004:0> list_namespace
```

2.2.3 DDL

1) 创建表

在 bigdata 命名空间中创建表格 student，两个列族。info 列族数据维护的版本数为 5 个，如果不写默认版本数为 1。

```
hbase:005:0> create 'bigdata:student', {NAME => 'info', VERSIONS => 5}, {NAME => 'msg'}
```

如果创建表格只有一个列族，没有列族属性，可以简写。

如果不写命名空间，使用默认的命名空间 default。

```
hbase:009:0> create 'student1','info'
```

2) 查看表

查看表有两个命令：list 和 describe

list: 查看所有的表名

```
hbase:013:0> list
```

describe: 查看一个表的详情

```
hbase:014:0> describe 'student1'
```


3) 修改表

表名创建时写的所有和列族相关的信息，都可以后续通过 `alter` 修改，包括增加删除列族。

(1) 增加列族和修改信息都使用覆盖的方法

```
hbase:015:0> alter 'student1', {NAME => 'f1', VERSIONS => 3}
```

(2) 删除信息使用特殊的语法

```
hbase:015:0> alter 'student1', NAME => 'f1', METHOD => 'delete'
hbase:016:0> alter 'student1', 'delete' => 'f1'
```

4) 删除表

shell 中删除表格,需要先将表格状态设置为不可用。

```
hbase:017:0> disable 'student1'
hbase:018:0> drop 'student1'
```

2.2.4 DML

1) 写入数据

在 HBase 中如果想要写入数据，只能添加结构中最底层的 `cell`。可以手动写入时间戳指定 `cell` 的版本，推荐不写默认使用当前的系统时间。

```
hbase:019:0> put 'bigdata:student','1001','info:name','zhangsan'
hbase:020:0> put 'bigdata:student','1001','info:name','lisi'
hbase:021:0> put 'bigdata:student','1001','info:age','18'
```

如果重复写入相同 `rowKey`，相同列的数据，会写入多个版本进行覆盖。

2) 读取数据

读取数据的方法有两个：`get` 和 `scan`。

`get` 最大范围是一行数据，也可以进行列的过滤，读取数据的结果为多行 `cell`。

```
hbase:022:0> get 'bigdata:student','1001'
hbase:023:0> get 'bigdata:student','1001' , {COLUMN => 'info:name'}
```

也可以修改读取 `cell` 的版本数，默认读取一个。最多能够读取当前列族设置的维护版本数。

```
hbase:024:0>get 'bigdata:student','1001' , {COLUMN => 'info:name',
VERSIONS => 6}
```

`scan` 是扫描数据，能够读取多行数据，不建议扫描过多的数据，推荐使用 `startRow` 和 `stopRow` 来控制读取的数据，默认范围左闭右开。

```
hbase:025:0> scan 'bigdata:student',{STARTROW => '1001',STOPROW =>
'1002'}
```

实际开发中使用 `shell` 的机会不多，所有丰富的使用方法到 `API` 中介绍。

3) 删除数据

删除数据的方法有两个：`delete` 和 `deleteall`。

`delete` 表示删除一个版本的数据，即为 1 个 `cell`，不填写版本默认删除最新的一个版本。

```
hbase:026:0> delete 'bigdata:student','1001','info:name'
```

`deleteall` 表示删除所有版本的数据，即为当前行当前列的多个 `cell`。（执行命令会标记数据为要删除，不会直接将数据彻底删除，删除数据只在特定时期清理磁盘时进行）

```
hbase:027:0> deleteall 'bigdata:student','1001','info:name'
```

第 3 章 HBase API

3.1 环境准备

新建项目后在 `pom.xml` 中添加依赖：

注意：会报错 `javax.el` 包不存在，是一个测试用的依赖，不影响使用

```
<dependencies>
  <dependency>
    <groupId>org.apache.hbase</groupId>
    <artifactId>hbase-server</artifactId>
    <version>2.4.11</version>
    <exclusions>
      <exclusion>
        <groupId>org.glassfish</groupId>
        <artifactId>javax.el</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>javax.el</artifactId>
    <version>3.0.1-b06</version>
  </dependency>
</dependencies>
```

3.2 创建连接

根据官方 API 介绍，HBase 的客户端连接由 `ConnectionFactory` 类来创建，用户使用完成之后需要手动关闭连接。同时连接是一个重量级的，推荐一个进程使用一个连接，对 HBase 的命令通过连接中的两个属性 `Admin` 和 `Table` 来实现。

3.2.1 单线程创建连接

```
package com.atguigu.hbase;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.client.AsyncConnection;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
```

```
import java.io.IOException;
import java.util.concurrent.CompletableFuture;

public class HBaseConnect {
    public static void main(String[] args) throws IOException {

        // 1. 创建配置对象
        Configuration conf = new Configuration();

        // 2. 添加配置参数
        conf.set("hbase.zookeeper.quorum", "hadoop102,hadoop103,hadoop104");

        // 3. 创建 hbase 的连接
        // 默认使用同步连接
        Connection connection =
        ConnectionFactory.createConnection(conf);

        // 可以使用异步连接
        // 主要影响后续的 DML 操作
        CompletableFuture<AsyncConnection> asyncConnection =
        ConnectionFactory.createAsyncConnection(conf);

        // 4. 使用连接
        System.out.println(connection);

        // 5. 关闭连接
        connection.close();
    }
}
```

3.2.2 多线程创建连接

使用类单例模式,确保使用一个连接,可以同时用于多个线程。

```
package com.atguigu;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.client.AsyncConnection;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;

import java.io.IOException;
import java.util.concurrent.CompletableFuture;

public class HBaseConnect {
    // 设置静态属性 hbase 连接
    public static Connection connection = null;

    static {

        // 创建 hbase 的连接
        try {
            // 使用配置文件的方法
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载, 可百度访问: [尚硅谷官网](#)

```
        connection = ConnectionFactory.createConnection();

    } catch (IOException e) {
        System.out.println("连接获取失败");
        e.printStackTrace();
    }
}

/**
 * 连接关闭方法,用于进程关闭时调用
 * @throws IOException
 */
public static void closeConnection() throws IOException {
    if (connection != null) {
        connection.close();
    }
}
}
```

在 `resources` 文件夹中创建配置文件 `hbase-site.xml`，添加以下内容

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
        <name>hbase.zookeeper.quorum</name>
        <value>hadoop102,hadoop103,hadoop104</value>
    </property>
</configuration>
```

3.3 DDL

创建 `HBaseDDL` 类，添加静态方法即可作为工具类

```
public class HBaseDDL {

    // 添加静态属性 connection 指向单例连接
    public static Connection connection = HBaseConnect.connection;
}
```

3.3.1 创建命名空间

```
/**
 * 创建命名空间
 * @param namespace 命名空间名称
 */
public static void createNamespace(String namespace) throws
IOException {
    // 1. 获取 admin
    // 此处的异常先不要抛出 等待方法写完 再统一进行处理
    // admin 的连接是轻量级的 不是线程安全的 不推荐池化或者缓存这个连接
    Admin admin = connection.getAdmin();

    // 2. 调用方法创建命名空间
    // 代码相对 shell 更加底层 所以 shell 能够实现的功能 代码一定能实现
}
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
// 所以需要填写完整的命名空间描述

// 2.1 创建命名空间描述建造者 => 设计师
NamespaceDescriptor.Builder builder =
NamespaceDescriptor.create(namespace);

// 2.2 给命名空间添加需求
builder.addConfiguration("user", "atguigu");

// 2.3 使用 builder 构造出对应的添加完参数的对象 完成创建
// 创建命名空间出现的问题 都属于本方法自身的问题 不应该抛出
try {
    admin.createNamespace(builder.build());
} catch (IOException e) {
    System.out.println("命名空间已经存在");
    e.printStackTrace();
}

// 3. 关闭 admin
admin.close();
}
```

3.3.2 判断表格是否存在

```
/**
 * 判断表格是否存在
 * @param namespace 命名空间名称
 * @param tableName 表格名称
 * @return true 表示存在
 */
public static boolean isTableExists(String namespace, String
tableName) throws IOException {
    // 1. 获取 admin
    Admin admin = connection.getAdmin();

    // 2. 使用方法判断表格是否存在
    boolean b = false;
    try {
        b = admin.tableExists(TableNames.valueOf(namespace,
tableName));
    } catch (IOException e) {
        e.printStackTrace();
    }

    // 3. 关闭 admin
    admin.close();

    // 3. 返回结果
    return b;

    // 后面的代码不能生效
}
```

3.3.3 创建表

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

```
/**
 * 创建表格
 * @param namespace 命名空间名称
 * @param tableName 表格名称
 * @param columnFamilies 列族名称 可以有多个
 */
public static void createTable(String namespace , String
tableName , String... columnFamilies) throws IOException {
    // 判断是否有至少一个列族
    if (columnFamilies.length == 0){
        System.out.println("创建表格至少有一个列族");
        return;
    }

    // 判断表格是否存在
    if (isTableExists(namespace,tableName)){
        System.out.println("表格已经存在");
        return;
    }

    // 1.获取 admin
    Admin admin = connection.getAdmin();

    // 2. 调用方法创建表格
    // 2.1 创建表格描述的建造者
    TableDescriptorBuilder tableDescriptorBuilder =
TableDescriptorBuilder.newBuilder(TableName.valueOf(namespace,
tableName));

    // 2.2 添加参数
    for (String columnFamily : columnFamilies) {
        // 2.3 创建列族描述的建造者
        ColumnFamilyDescriptorBuilder
columnFamilyDescriptorBuilder =
ColumnFamilyDescriptorBuilder.newBuilder(Bytes.toBytes(columnFami
ly));

        // 2.4 对应当前的列族添加参数
        // 添加版本参数
        columnFamilyDescriptorBuilder.setMaxVersions(5);

        // 2.5 创建添加完参数的列族描述
        tableDescriptorBuilder.setColumnFamily(columnFamilyDescriptorBuil
der.build());
    }

    // 2.6 创建对应的表格描述
    try {
        admin.createTable(tableDescriptorBuilder.build());
    } catch (IOException e) {

        e.printStackTrace();
    }
}
```

```
}

// 3. 关闭 admin
admin.close();
}
```

3.3.4 修改表

```
/**
 * 修改表格中一个列族的版本
 * @param namespace 命名空间名称
 * @param tableName 表格名称
 * @param columnFamily 列族名称
 * @param version 版本
 */
public static void modifyTable(String namespace, String
tableName, String columnFamily, int version) throws IOException {

    // 判断表格是否存在
    if (!isTableExists(namespace, tableName)) {
        System.out.println("表格不存在无法修改");
        return;
    }

    // 1. 获取 admin
    Admin admin = connection.getAdmin();

    try {
        // 2. 调用方法修改表格
        // 2.0 获取之前的表格描述
        TableDescriptor descriptor =
admin.getDescriptor(tableName.valueOf(namespace, tableName));

        // 2.1 创建一个表格描述建造者
        // 如果使用填写 tableName 的方法 相当于创建了一个新的表格描述建造
者 没有之前的信息
        // 如果想要修改之前的信息 必须调用方法填写一个旧的表格描述
        TableDescriptorBuilder tableDescriptorBuilder =
TableDescriptorBuilder.newBuilder(descriptor);

        // 2.2 对应建造者进行表格数据的修改
        ColumnFamilyDescriptor columnFamily1 =
descriptor.getColumnFamily(Bytes.toBytes(columnFamily));

        // 创建列族描述建造者
        // 需要填写旧的列族描述
        ColumnFamilyDescriptorBuilder
columnFamilyDescriptorBuilder =
ColumnFamilyDescriptorBuilder.newBuilder(columnFamily1);

        // 修改对应的版本
        columnFamilyDescriptorBuilder.setMaxVersions(version);
    }
}
```

```
// 此处修改的时候 如果填写的新创建 那么别的参数会初始化

tableDescriptorBuilder.modifyColumnFamily(columnFamilyDescriptorBuilder.build());
    admin.modifyTable(tableDescriptorBuilder.build());
} catch (IOException e) {

    e.printStackTrace();
}

// 3. 关闭 admin
admin.close();
}
```

3.3.5 删除表

```
/**
 * 删除表格
 * @param namespace 命名空间名称
 * @param tableName 表格名称
 * @return true 表示删除成功
 */
public static boolean deleteTable(String namespace ,String
tableName) throws IOException {
    // 1. 判断表格是否存在
    if (!isTableExists(namespace,tableName)){
        System.out.println("表格不存在 无法删除");
        return false;
    }

    // 2. 获取 admin
    Admin admin = connection.getAdmin();

    // 3. 调用相关的方法删除表格
    try {
        // HBase 删除表格之前 一定要先标记表格为不可以
        TableName tableName1 = TableName.valueOf(namespace,
tableName);
        admin.disableTable(tableName1);
        admin.deleteTable(tableName1);
    } catch (IOException e) {
        e.printStackTrace();
    }

    // 4. 关闭 admin
    admin.close();

    return true;
}
```

3.4 DML

创建类 HBaseDML

```
public class HBaseDML {
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)


```
// 添加静态属性 connection 指向单例连接
public static Connection connection = HBaseConnect.connection;
}
```

3.4.1 插入数据

```
/**
 * 插入数据
 * @param namespace 命名空间名称
 * @param tableName 表格名称
 * @param rowKey 主键
 * @param columnFamily 列族名称
 * @param columnName 列名
 * @param value 值
 */
public static void putCell(String namespace, String
tableName, String rowKey, String columnFamily, String
columnName, String value) throws IOException {
    // 1. 获取 table
    Table table =
connection.getTable(TableName.valueOf(namespace, tableName));

    // 2. 调用相关方法插入数据
    // 2.1 创建 put 对象
    Put put = new Put(Bytes.toBytes(rowKey));

    // 2.2. 给 put 对象添加数据

put.addColumn(Bytes.toBytes(columnFamily), Bytes.toBytes(columnNam
e), Bytes.toBytes(value));

    // 2.3 将对象写入对应的方法
    try {
        table.put(put);
    } catch (IOException e) {
        e.printStackTrace();
    }

    // 3. 关闭 table
    table.close();
}
```

3.4.2 读取数据

```
/**
 * 读取数据 读取对应的一行中的某一列
 *
 * @param namespace 命名空间名称
 * @param tableName 表格名称
 * @param rowKey 主键
 * @param columnFamily 列族名称
 * @param columnName 列名
 */
```

```
public static void getCells(String namespace, String tableName,
String rowKey, String columnFamily, String columnName) throws
IOException {
    // 1. 获取 table
    Table table =
connection.getTable(TableName.valueOf(namespace, tableName));

    // 2. 创建 get 对象
    Get get = new Get(Bytes.toBytes(rowKey));

    // 如果直接调用 get 方法读取数据 此时读一整行数据
    // 如果想读取某一列的数据 需要添加对应的参数
    get.addColumn(Bytes.toBytes(columnFamily),
Bytes.toBytes(columnName));

    // 设置读取数据的版本
    get.readAllVersions();

    try {
        // 读取数据 得到 result 对象
        Result result = table.get(get);
        // 处理数据
        Cell[] cells = result.rawCells();

        // 测试方法：直接把读取的数据打印到控制台
        // 如果是实际开发 需要再额外写方法 对应处理数据
        for (Cell cell : cells) {
            // cell 存储数据比较底层
            String value = new String(CellUtil.cloneValue(cell));
            System.out.println(value);
        }

    } catch (IOException e) {
        e.printStackTrace();
    }

    // 关闭 table
    table.close();
}
```

3.4.3 扫描数据

```
/**
 * 扫描数据
 *
 * @param namespace 命名空间
 * @param tableName 表格名称
 * @param startRow 开始的 row 包含的
 * @param stopRow 结束的 row 不包含
 */
public static void scanRows(String namespace, String tableName,
String startRow, String stopRow) throws IOException {
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
// 1. 获取 table
Table table =
connection.getTable(TableName.valueOf(namespace, tableName));

// 2. 创建 scan 对象
Scan scan = new Scan();
// 如果此时直接调用 会直接扫描整张表

// 添加参数 来控制扫描的数据
// 默认包含
scan.withStartRow(Bytes.toBytes(startRow));
// 默认不包含
scan.withStopRow(Bytes.toBytes(stopRow));

try {
    // 读取多行数据 获得 scanner
    ResultScanner scanner = table.getScanner(scan);
    // result 来记录一行数据 cell 数组
    // ResultScanner 来记录多行数据 result 的数组
    for (Result result : scanner) {
        Cell[] cells = result.rawCells();

        for (Cell cell : cells) {
            System.out.print
String(CellUtil.cloneRow(cell)) + "-" + (new
String(CellUtil.cloneFamily(cell)) + "-" + new
String(CellUtil.cloneQualifier(cell)) + "-" + new
String(CellUtil.cloneValue(cell)) + "\t");
        }
        System.out.println();
    }
} catch (IOException e) {
    e.printStackTrace();
}

// 3. 关闭 table
table.close();
}
```

3.4.4 带过滤扫描

```
/**
 * 带过滤的扫描
 *
 * @param namespace 命名空间
 * @param tableName 表格名称
 * @param startRow 开始 row
 * @param stopRow 结束 row
 * @param columnFamily 列族名称
 * @param columnName 列名
 * @param value value 值
 */
```

```
* @throws IOException
*/
public static void filterScan(String namespace, String tableName,
String startRow, String stopRow, String columnFamily, String
columnName, String value) throws IOException {
    // 1. 获取 table
    Table table =
connection.getTable(TableName.valueOf(namespace, tableName));

    // 2. 创建 scan 对象
    Scan scan = new Scan();
    // 如果此时直接调用 会直接扫描整张表

    // 添加参数 来控制扫描的数据
    // 默认包含
    scan.withStartRow(Bytes.toBytes(startRow));
    // 默认不包含
    scan.withStopRow(Bytes.toBytes(stopRow));

    // 可以添加多个过滤
    FilterList filterList = new FilterList();

    // 创建过滤器
    // (1) 结果只保留当前列的数据
    ColumnValueFilter columnValueFilter = new ColumnValueFilter(
        // 列族名称
        Bytes.toBytes(columnFamily),
        // 列名
        Bytes.toBytes(columnName),
        // 比较关系
        CompareOperator.EQUAL,
        // 值
        Bytes.toBytes(value)
    );

    // (2) 结果保留整行数据
    // 结果同时会保留没有当前列的数据
    SingleColumnValueFilter singleColumnValueFilter = new
SingleColumnValueFilter(
        // 列族名称
        Bytes.toBytes(columnFamily),
        // 列名
        Bytes.toBytes(columnName),
        // 比较关系
        CompareOperator.EQUAL,
        // 值
        Bytes.toBytes(value)
    );

    // 本身可以添加多个过滤器
    filterList.addFilter(singleColumnValueFilter);

    // 添加过滤
```

```
scan.setFilter(filterList);

try {
    // 读取多行数据 获得 scanner
    ResultScanner scanner = table.getScanner(scan);
    // result 来记录一行数据    cell 数组
    // ResultScanner 来记录多行数据 result 的数组
    for (Result result : scanner) {
        Cell[] cells = result.rawCells();

        for (Cell cell : cells) {
            System.out.print(new
String(CellUtil.cloneRow(cell))      +      "-"      +      new
String(CellUtil.cloneFamily(cell))   +      "-"      +      new
String(CellUtil.cloneQualifier(cell)) +      "-"      +      new
String(CellUtil.cloneValue(cell)) + "\t");
        }
        System.out.println();
    }
} catch (IOException e) {
    e.printStackTrace();
}

// 3. 关闭 table
table.close();
}
```

3.4.5 删除数据

```
/**
 * 删除 column 数据
 *
 * @param nameSpace
 * @param tableName
 * @param rowKey
 * @param family
 * @param column
 * @throws IOException
 */
public static void deleteColumn(String nameSpace, String tableName,
String rowKey, String family, String column) throws IOException {

    // 1.获取 table
    Table table = connection.getTable(TableName.valueOf(nameSpace,
tableName));

    // 2.创建 Delete 对象
    Delete delete = new Delete(Bytes.toBytes(rowKey));

    // 3.添加删除信息
    // 3.1 删除单个版本
    //
delete.addColumn(Bytes.toBytes(family),Bytes.toBytes(column));
```

更多 [Java](#) -[大数据](#) -[前端](#) -[python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
// 3.2 删除所有版本
delete.addColumn(Bytes.toBytes(family),
Bytes.toBytes(column));
// 3.3 删除列族
//      delete.addFamily(Bytes.toBytes(family));

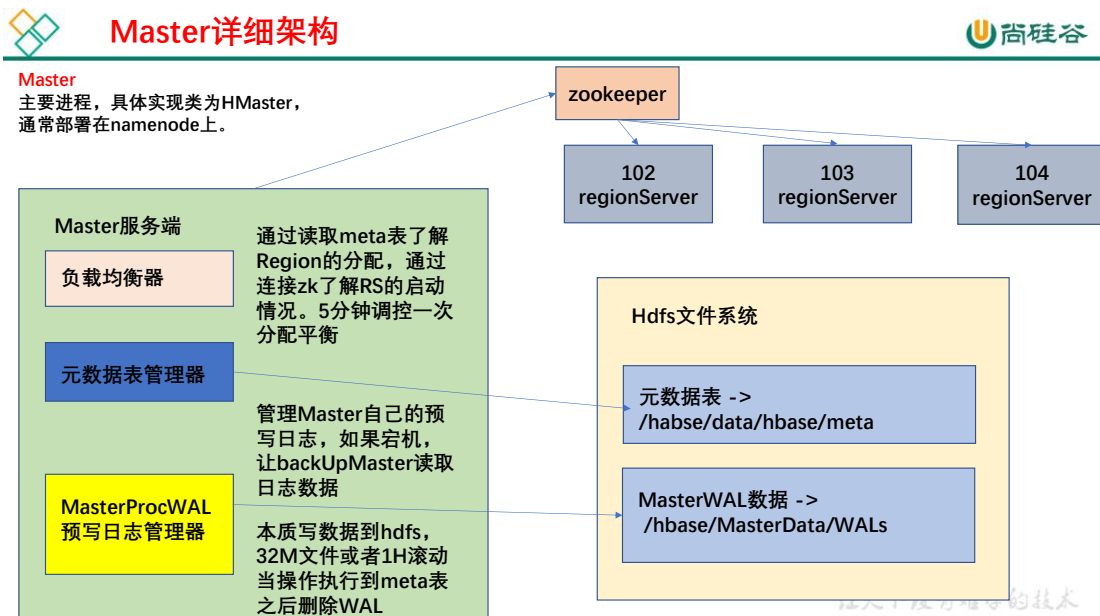
// 3.删除数据
table.delete(delete);

// 5.关闭资源
table.close();
}

public static void main(String[] args) throws IOException {
//      putCell("bigdata","student","1002","info","name","lisi");
//      String cell = getCell("bigdata", "student", "1001", "info",
"name");
//      System.out.println(cell);
//      List<String> strings = scanRows("bigdata", "student",
"1001", "2000");
//      for (String string : strings) {
//          System.out.println(string);
//          deleteColumn("bigdata", "student", "1001", "info", "name");
//      }
}
```

第 4 章 HBase 进阶

4.1 Master 架构



1) Meta 表格介绍：（警告：不要去改这个表）

全称 hbase: meta，只是在 list 命令中被过滤掉了，本质上和 HBase 的其他表格一样。

RowKey:

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

([table],[region start key],[region id]) 即 表名, region 起始位置和 regionID。

列:

info: regioninfo 为 region 信息, 存储一个 HRegionInfo 对象。

info: server 当前 region 所处的 RegionServer 信息, 包含端口号。

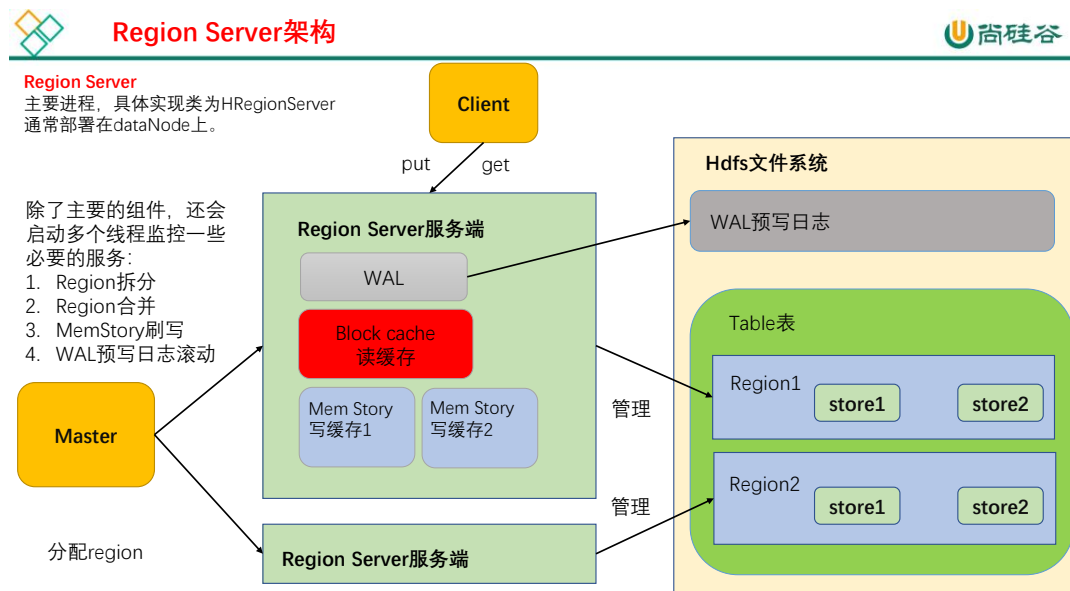
info: serverstartcode 当前 region 被分到 RegionServer 的起始时间。

如果一个表处于切分的过程中, 即 region 切分, 还会多出两列 info: splitA 和 info: splitB, 存储值也是 HRegionInfo 对象, 拆分结束后, 删除这两列。

注意: 在客户端对元数据进行操作的时候才会连接 master, 如果对数据进行读写, 直接连接 zookeeper 读取目录/hbase/meta-region-server 节点信息, 会记录 meta 表格的位置。直接读取即可, 不需要访问 master, 这样可以减轻 master 的压力, 相当于 master 专注 meta 表的写操作, 客户端可直接读取 meta 表。

在 HBase 的 2.3 版本更新了一种新模式: Master Registry。客户端可以访问 master 来读取 meta 表信息。加大了 master 的压力, 减轻了 zookeeper 的压力。

4.2 RegionServer 架构



1) MemStore

写缓存, 由于 HFile 中的数据要求是有序的, 所以数据是先存储在 MemStore 中, 排好序后, 等到达刷写时机才会刷写到 HFile, 每次刷写都会形成一个新的 HFile, 写入到对应的文件夹 store 中。

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载, 可百度访问: [尚硅谷官网](#)

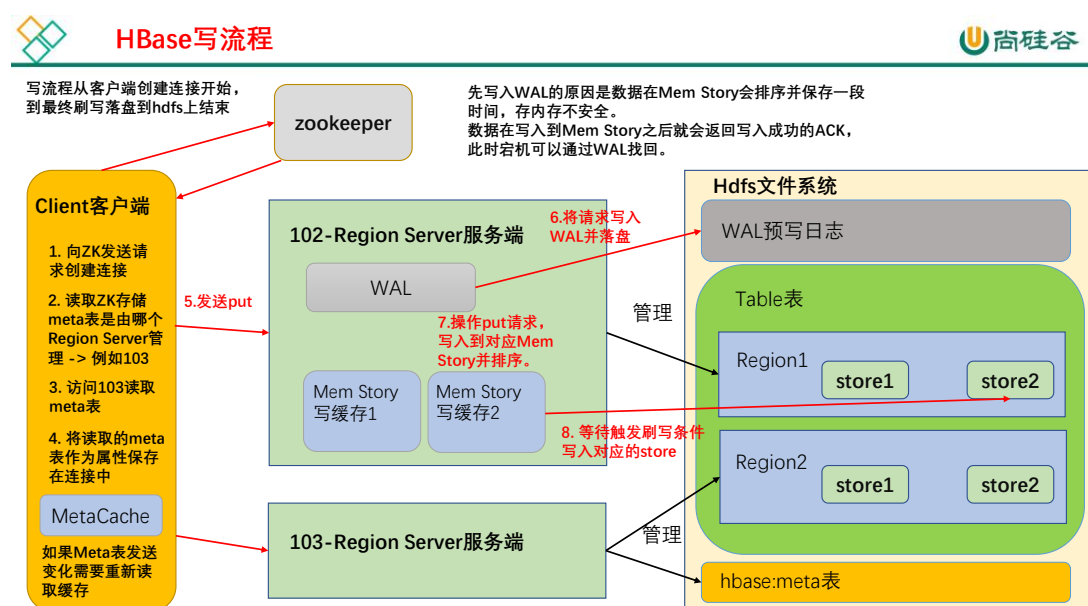
2) WAL

由于数据要经 MemStore 排序后才能刷写到 HFile，但把数据保存在内存中会有很高的概率导致数据丢失，为了解决这个问题，数据会先写在一个叫做 Write-Ahead logfile 的文件中，然后再写入 MemStore 中。所以在系统出现故障的时候，数据可以通过这个日志文件重建。

3) BlockCache

读缓存，每次查询出的数据会缓存在 BlockCache 中，方便下次查询。

4.3 写流程



2) 写流程:

写流程顺序正如 API 编写顺序，首先创建 HBase 的重量级连接

- (1) 首先访问 zookeeper，获取 hbase:meta 表位于哪个 Region Server;
- (2) 访问对应的 Region Server，获取 hbase:meta 表，将其缓存到连接中，作为连接属性 MetaCache，由于 Meta 表格具有一定的数据量，导致了创建连接比较慢;

之后使用创建的连接获取 Table，这是一个轻量级的连接，只有在第一次创建的时候会检查表格是否存在访问 RegionServer，之后在获取 Table 时不会访问 RegionServer;

- (3) 调用 Table 的 put 方法写入数据，此时还需要解析 RowKey，对照缓存的 MetaCache，查看具体写入的位置有哪个 RegionServer;

- (4) 将数据顺序写入（追加）到 WAL，此处写入是直接落盘的，并设置专门的线程控制 WAL 预写日志的滚动（类似 Flume）;

更多 [Java - 大数据 - 前端 - python 人工智能资料下载](#)，可百度访问：尚硅谷官网

(5) 根据写入命令的 RowKey 和 ColumnFamily 查看具体写入到哪个 MemStore，并且在 MemStore 中排序；

(6) 向客户端发送 ack；

(7) 等达到 MemStore 的刷写时机后，将数据刷写到对应的 store 中。

4.4 MemStore Flush

MemStore 刷写由多个线程控制，条件互相独立：

主要的刷写规则是控制刷写文件的大小，在每一个刷写线程中都会进行监控

(1) 当某个 memstore 的大小达到了 `hbase.hregion.memstore.flush.size` (默认值 128M)，其所在 region 的所有 memstore 都会刷写。

当 memstore 的大小达到了

`hbase.hregion.memstore.flush.size` (默认值 128M)

`* hbase.hregion.memstore.block.multiplier` (默认值 4)

时，会刷写同时阻止继续往该 memstore 写数据（由于线程监控是周期性的，所有有可能面对数据洪峰，尽管可能性比较小）

(2) 由 HRegionServer 中的属性 MemStoreFlusher 内部线程 FlushHandler 控制。标准为 LOWER_MARK (低水位线) 和 HIGH_MARK (高水位线)，意义在于避免写缓存使用过多的内存造成 OOM

当 region server 中 memstore 的总大小达到低水位线

`java_heapsize`

`*hbase.regionserver.global.memstore.size` (默认值 0.4)

`*hbase.regionserver.global.memstore.size.lower.limit` (默认值 0.95)，

region 会按照其所有 memstore 的大小顺序(由大到小)依次进行刷写。直到 region server 中所有 memstore 的总大小减小到上述值以下。

当 region server 中 memstore 的总大小达到高水位线

`java_heapsize`

`*hbase.regionserver.global.memstore.size` (默认值 0.4)

时，会同时阻止继续往所有的 memstore 写数据。

(3) 为了避免数据过长时间处于内存之中，到达自动刷写的时间，也会触发 memstore flush。由 HRegionServer 的属性 PeriodicMemStoreFlusher 控制进行，由于重要性比较低，5min 更多 [Java - 大数据 - 前端 - python 人工智能资料下载](#)，可百度访问：[尚硅谷官网](#)

才会执行一次。

自动刷新的时间间隔由该属性进行配置 `hbase.regionserver.optionalcacheflushinterval`（默认 1 小时）。

（4）当 WAL 文件的数量超过 `hbase.regionserver.max.logs`，region 会按照时间顺序依次进行刷写，直到 WAL 文件数量减小到 `hbase.regionserver.max.log` 以下（该属性名已经废弃，现无需手动设置，最大值为 32）。

4.5 读流程

4.5.1 HFile 结构

在了解读流程之前，需要先知道读取的数据是什么样子的。

HFile 是存储在 HDFS 上面每一个 store 文件夹下实际存储数据的文件。里面存储多种内容。包括数据本身（`keyValue` 键值对）、元数据记录、文件信息、数据索引、元数据索引和一个固定长度的尾部信息（记录文件的修改情况）。

键值对按照块大小（默认 64K）保存在文件中，数据索引按照块创建，块越多，索引越大。每一个 HFile 还会维护一个布隆过滤器（就像是一个很大的地图，文件中每有一种 key，就在对应的位置标记，读取时可以大致判断要 get 的 key 是否存在 HFile 中）。

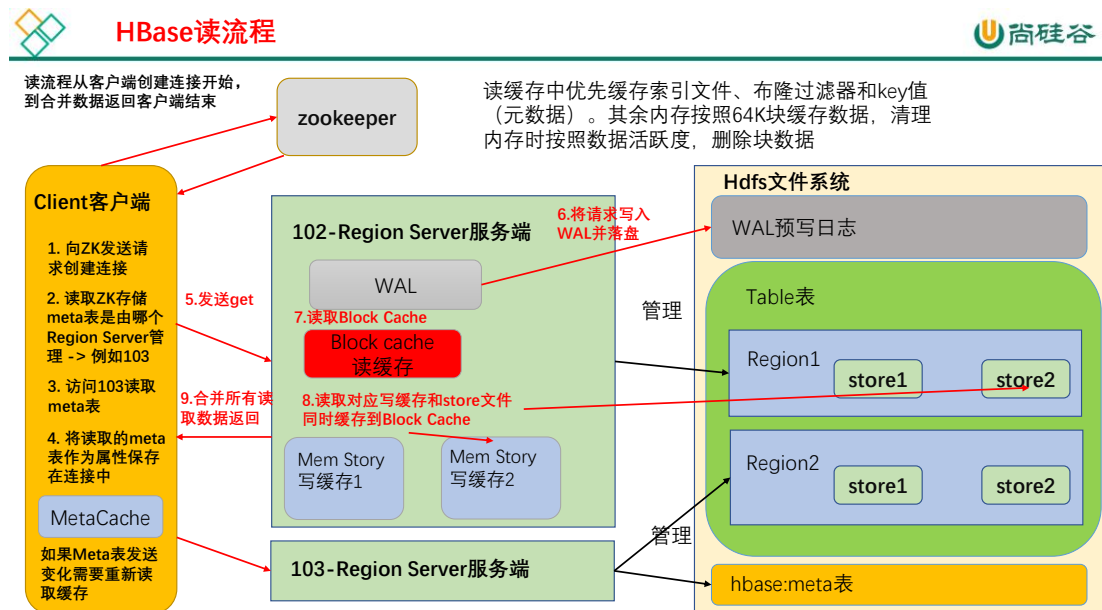
KeyValue 内容如下：

rowlength -----> key 的长度
row -----> key 的值
columnfamilylength --> 列族长度
columnfamily -----> 列族
columnqualifier -----> 列名
timestamp -----> 时间戳（默认系统时间）
keytype -----> Put

由于 HFile 存储经过序列化，所以无法直接查看。可以通过 HBase 提供的命令来查看存储在 HDFS 上面的 HFile 元数据内容。

```
[atguigu@hadoop102 hbase]$ bin/hbase hfile -m -f /hbase/data/命名空间/表名/regionID/列族/HFile 名
```

4.5.2 读流程



创建连接同写流程。

- (1) 创建 Table 对象发送 get 请求。
- (2) 优先访问 Block Cache，查找是否之前读取过，并且可以读取 HFile 的索引信息和布隆过滤器。
- (3) 不管读缓存中是否已经有数据了（可能已经过期了），都需要再次读取写缓存和 store 中的文件。
- (4) 最终将所有读取到的数据合并版本，按照 get 的要求返回即可。

4.5.3 合并读取数据优化

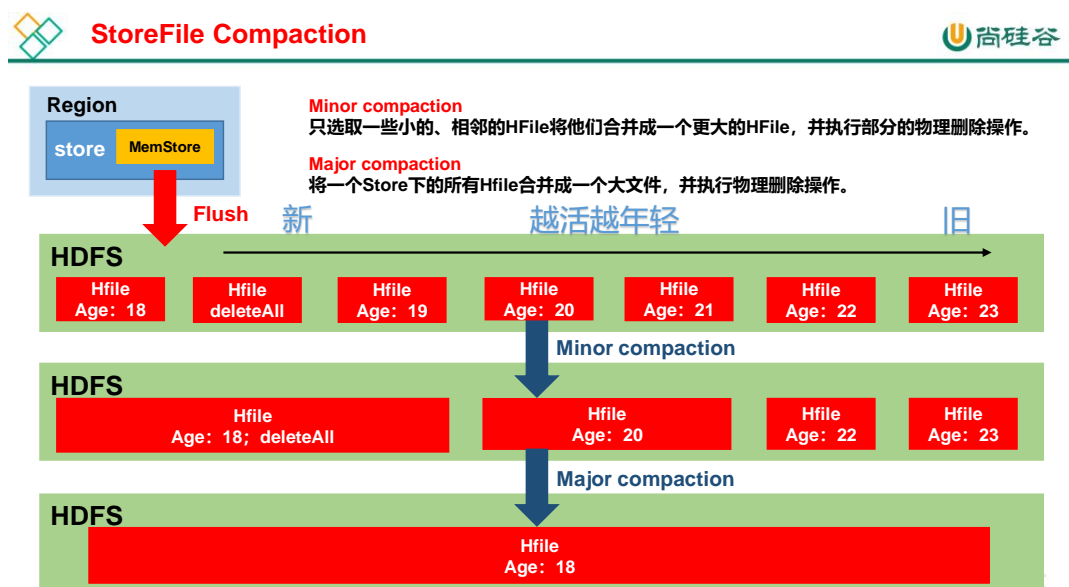
每次读取数据都需要读取三个位置，最后进行版本的合并。效率会非常低，所有系统需要对此优化。

- (1) HFile 带有索引文件，读取对应 RowKey 数据会比较快。
- (2) Block Cache 会缓存之前读取的内容和元数据信息，如果 HFile 没有发生变化（记录在 HFile 尾信息中），则不需要再次读取。
- (3) 使用布隆过滤器能够快速过滤当前 HFile 不存在需要读取的 RowKey，从而避免读取文件。（布隆过滤器使用 HASH 算法，不是绝对准确的，出错会造成多扫描一个文件，对读取数据结果没有影响）

4.6 StoreFile Compaction

由于 memstore 每次刷写都会生成一个新的 HFile，文件过多读取不方便，所以会进行文件的合并，清理掉过期和删除的数据，会进行 StoreFile Compaction。

Compaction 分为两种，分别是 **Minor Compaction** 和 **Major Compaction**。Minor Compaction 会将临近的若干个较小的 HFile 合并成一个较大的 HFile，并清理掉部分过期和删除的数据，有系统使用一组参数自动控制，Major Compaction 会将一个 Store 下的所有的 HFile 合并成一个大 HFile，并且会清理掉所有过期和删除的数据，由参数 `hbase.hregion.majorcompaction` 控制，默认 7 天。



Minor Compaction 控制机制：

参与到小合并的文件需要通过参数计算得到，有效的参数有 5 个

- (1) `hbase.hstore.compaction.ratio`（默认 1.2F）合并文件选择算法中使用的比率。
- (2) `hbase.hstore.compaction.min`（默认 3）为 Minor Compaction 的最少文件个数。
- (3) `hbase.hstore.compaction.max`（默认 10）为 Minor Compaction 最大文件个数。
- (4) `hbase.hstore.compaction.min.size`（默认 128M）为单个 Hfile 文件大小最小值，小于这个数会被合并。
- (5) `hbase.hstore.compaction.max.size`（默认 Long.MAX_VALUE）为单个 Hfile 文件大小最大值，高于这个数不会被合并。

小合并机制为拉取整个 store 中的所有文件，做成一个集合。之后按照从旧到新的顺序遍历。

判断条件为：

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

- ① 过小合并，过大不合并
- ② 文件大小/ `hbase.hstore.compaction.ratio` < (剩余文件大小和) 则参与压缩。所有把比值设置过大，如 10 会最终合并为 1 个特别大的文件，相反设置为 0.4，会最终产生 4 个 storeFile。不建议修改默认值
- ③ 满足压缩条件的文件个数达不到个数要求 ($3 \leq \text{count} \leq 10$) 则不压缩。

4.7 Region Split

Region 切分分为两种，创建表格时候的预分区即自定义分区，同时系统默认还会启动一个切分规则，避免单个 Region 中的数据量太大。

4.7.1 预分区（自定义分区）

每一个 region 维护着 startRow 与 endRowKey，如果加入的数据符合某个 region 维护的 rowKey 范围，则该数据交给这个 region 维护。那么依照这个原则，我们可以将数据所要投放的分区提前大致的规划好，以提高 HBase 性能。

1) 手动设定预分区

```
create 'staff1','info', SPLITS => ['1000','2000','3000','4000']
```

2) 生成 16 进制序列预分区

```
create 'staff2','info',{NUMREGIONS => 15, SPLITALGO => 'HexStringSplit'}
```

3) 按照文件中设置的规则预分区

(1) 创建 splits.txt 文件内容如下：

```
aaaa
bbbb
cccc
dddd
```

(2) 然后执行：

```
create 'staff3','info',SPLITS_FILE => 'splits.txt'
```

4) 使用 JavaAPI 创建预分区

```
}
//6. 创建表
admin.createTable(builder.build(),0);
//7. 关闭资源
admin.close();
return true;
}
```

```
package com.atguigu.hbase;

import org.apache.hadoop.conf.Configuration;
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

public class HBaseConnect {
    public static void main(String[] args) throws IOException {
        // 1.获取配置类
        Configuration conf = HBaseConfiguration.create();

        // 2.给配置类添加配置

        conf.set("hbase.zookeeper.quorum","hadoop102,hadoop103,hadoop104"
        );

        // 3.获取连接
        Connection connection =
        ConnectionFactory.createConnection(conf);

        // 3.获取 admin
        Admin admin = connection.getAdmin();

        // 5.获取 descriptor 的 builder
        TableDescriptorBuilder builder =
        TableDescriptorBuilder.newBuilder(TableName.valueOf("bigdata",
        "staff4"));
        // 6. 添加列族

        builder.setColumnFamily(ColumnFamilyDescriptorBuilder.newBuilder(
        Bytes.toBytes("info")).build());

        // 7.创建对应的切分
        byte[][] splits = new byte[3][];
        splits[0] = Bytes.toBytes("aaa");
        splits[1] = Bytes.toBytes("bbb");
        splits[2] = Bytes.toBytes("ccc");

        // 8.创建表
        admin.createTable(builder.build(),splits);
        // 9.关闭资源
        admin.close();
        connection.close();
    }
}
```

4.7.2 系统拆分

Region 的拆分是由 HRegionServer 完成的，在操作之前需要通过 ZK 汇报 master，修改对应的 Meta 表信息添加两列 info: splitA 和 info: splitB 信息。之后需要操作 HDFS 上面对应的文件，按照拆分后的 Region 范围进行标记区分，**实际操作为创建文件引用，不会挪动**

数据。刚完成拆分的时候，两个 Region 都由原先的 RegionServer 管理。之后汇报给 Master，由 Master 将修改后的信息写入到 Meta 表中。等待下一次触发负载均衡机制，才会修改 Region 的管理服务者，而数据要等到下一次压缩时，才会实际进行移动。

不管是否使用预分区，系统都会默认启动一套 Region 拆分规则。不同版本的拆分规则有差别。系统拆分策略的父类为 RegionSplitPolicy。

0.94 版本之前 => ConstantSizeRegionSplitPolicy

(1) 当 1 个 region 中的某个 Store 下所有 StoreFile 的总大小超过 `hbase.hregion.max.filesize` (10G)，该 Region 就会进行拆分。

0.94 版本之后，2.0 版本之前 => IncreasingToUpperBoundRegionSplitPolicy

(2) 当 1 个 region 中的某个 Store 下所有 StoreFile 的总大小超过 `Min(initialSize*R^3, hbase.hregion.max.filesize)`，该 Region 就会进行拆分。其中 `initialSize` 的默认值为 `2*hbase.hregion.memstore.flush.size`，R 为当前 Region Server 中属于该 Table 的 Region 个数 (0.94 版本之后)。

具体的切分策略为：

第一次 split: $1^3 * 256 = 256\text{MB}$

第二次 split: $2^3 * 256 = 2048\text{MB}$

第三次 split: $3^3 * 256 = 6912\text{MB}$

第四次 split: $4^3 * 256 = 16384\text{MB} > 10\text{GB}$ ，因此取较小的值 10GB

后面每次 split 的 size 都是 10GB 了。

2.0 版本之后 => SteppingSplitPolicy

(3) Hbase 2.0 引入了新的 split 策略：如果当前 RegionServer 上该表只有一个 Region，按照 `2 * hbase.hregion.memstore.flush.size` 分裂，否则按照 `hbase.hregion.max.filesize` 分裂。这叫大道至简，学海抽丝。

第 5 章 HBase 优化

5.1 RowKey 设计

一条数据的唯一标识就是 rowkey，那么这条数据存储于哪个分区，取决于 rowkey 处于哪个一个预分区的区间内，设计 rowkey 的主要目的，就是让数据均匀的分布于所有的 region 中，在一定程度上防止数据倾斜。接下来我们就谈一谈 rowkey 常用的设计方案。

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

- 1) 生成随机数、hash、散列值
- 2) 时间戳反转
- 3) 字符串拼接

需求：使用 hbase 存储下列数据，要求能够通过 hbase 的 API 读取数据完成两个统计需求。

- (1) 统计张三在 2021 年 12 月份消费的总金额
- (2) 统计所有人在 2021 年 12 月份消费的总金额



user_info.txt

5.1.1 实现需求 1

为了能够统计张三在 2021 年 12 月份消费的总金额，我们需要用 scan 命令能够得到张三在这个月消费的所有记录，之后在进行累加即可。Scan 需要填写 startRow 和 stopRow:

```
scan :   startRow -> ^A^Azhangsang2021-12
      endRow   -> ^A^Azhangsang2021-12.
```

注意点:

- (1) 避免扫描数据混乱，解决字段长度不一致的问题，可以使用相同阿斯卡码值的符号进行填充，框架底层填充使用的是阿斯卡码值为 1 的 ^A。

ASCII 字符代码表 一																					
高四位 低四位		ASCII非打印控制字符										ASCII 打印字符									
		0000					0001					0010	0011	0100	0101	0110	0111				
		0					1					2	3	4	5	6	7				
	十进制	字符	ctrl	代码	字符解释	十进制	字符	ctrl	代码	字符解释	十进制	字符	十进制	字符	十进制	字符	十进制	字符	ctrl		
0000	0	0	BLANK NULL	^@	NUL 空	16	►	^P	DLE 数据链路转意	32		48	0	64	@	80	P	96	`	112	p
0001	1	1	☺	^A	SOH 头标开始	17	◄	^Q	DC1 设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q
0010	2	2	☹	^B	STX 正文开始	18	↕	^R	DC2 设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r
0011	3	3	♥	^C	ETX 正文结束	19	!!	^S	DC3 设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s
0100	4	4	♦	^D	EOT 传输结束	20	¶	^T	DC4 设备控制 4	36	\$	52	4	68	D	84	T	100	d	116	t
0101	5	5	♣	^E	ENQ 查询	21	§	^U	NAK 反确认	37	%	53	5	69	E	85	U	101	e	117	u
0110	6	6	♠	^F	ACK 确认	22	■	^V	SYN 同步空闲	38	&	54	6	70	F	86	V	102	f	118	v
0111	7	7	●	^G	BEL 震铃	23	↑	^W	ETB 传输块结束	39	'	55	7	71	G	87	w	103	g	119	w
1000	8	8	▣	^H	BS 退格	24	↑	^X	CAN 取消	40	(56	8	72	H	88	X	104	h	120	x
1001	9	9	○	^I	TAB 水平制表符	25	↓	^Y	EM 媒体结束	41)	57	9	73	I	89	Y	105	i	121	y
1010	A	10	◻	^J	LF 换行/新行	26	→	^Z	SUB 替换	42	*	58	:	74	J	90	Z	106	j	122	z
1011	B	11	♂	^K	VT 垂直制表符	27	←	^[ESC 转意	43	+	59	;	75	K	91	[107	k	123	{
1100	C	12	♀	^L	FF 换页/新页	28	└	^_	FS 文件分隔符	44	,	60	<	76	L	92	\	108	l	124	
1101	D	13	♪	^M	CR 回车	29	↔	^]	GS 组分隔符	45	-	61	=	77	M	93]	109	m	125	}
1110	E	14	🎵	^N	SO 移出	30	▲	^6	RS 记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~
1111	F	15	☼	^O	SI 移入	31	▼	^-	US 单元分隔符	47	/	63	?	79	O	95	_	111	o	127	Δ

Back space

注：表中的ASCII字符可以用:ALT + “小键盘上的数字键”输入

- (2) 最后的日期结尾处需要使用阿斯卡码略大于‘-’的值

更多 Java -大数据 -前端 -python 人工智能资料下载,可百度访问: [尚硅谷官网](#)

最终得到 rowKey 的设计为:

```
//注意 rowkey 相同的数据会视为相同数据覆盖掉之前的版本  
rowKey: userdate (yyyy-MM-dd HH:mm:ss)
```

5.1.2 实现需求 2

问题提出: 按照需求 1 的 rowKey 设计, 会发现对于需求 2, 完全没有办法写 rowKey 的扫描范围。此处能够看出 hbase 设计 rowKey 使用的特点为:

适用性强 泛用性差 能够完美实现一个需求 但是不能同时完美实现多个需要。

如果想要同时完成两个需求, 需要对 rowKey 出现字段的顺序进行调整。

调整的原则为: 可枚举的放在前面。其中时间是可以枚举的, 用户名称无法枚举, 所以必须把时间放在前面。

最终满足 2 个需求的设计
可以穷举的写在前面即可

rowKey 设计格式 => date (yyyy-MM) ^A^Auserdate (-dd hh:mm:ss ms)

(1) 统计张三在 2021 年 12 月份消费的总金额

```
scan: startRow => 2021-12^A^Azhangsang  
      stopRow  => 2021-12^A^Azhangsang.
```

(2) 统计所有人在 2021 年 12 月份消费的总金额

```
scan: startRow => 2021-12  
      stopRow  => 2021-12.
```

5.1.3 添加预分区优化

预分区的分区号同样需要遵守 rowKey 的 scan 原则。所有必须添加在 rowKey 的最前面, 前缀为最简单的数字。同时使用 hash 算法将用户名和月份拼接决定分区号。(单独使用用户名会造成单一用户所有数据存储在在一个分区)

添加预分区优化

startKey	stopKey
	001
001	002
002	003
...	
119	120

分区号=> hash (user+date (MM)) % 120

分区号填充 如果得到 1 => 001

rowKey 设计格式 => 分区号 date (yyyy-MM) ^A^Auserdate (-dd hh:mm:ss ms)

缺点: 实现需求 2 的时候, 由于每个分区都有 12 月份的数据, 需要扫描 120 个分区。

解决方法: 提前将分区号和月份进行对应。

提前将月份和分区号对应一下

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载, 可百度访问: [尚硅谷官网](#)

```
000 到 009 分区 存储的都是 1 月份数据
010 到 019 分区 存储的都是 2 月份数据
...
110 到 119 分区 存储的都是 12 月份数据

是 9 月份的数据
分区号=> hash(user+date(MM)) % 10 + 80
分区号填充 如果得到 85 => 085

得到 12 月份所有人的数据
扫描 10 次
scan: startRow => 1102021-12
      stopRow  => 1102021-12.
      ...
      startRow => 1122021-12
      stopRow  => 1122021-12.
      ..
      startRow => 1192021-12
      stopRow  => 1192021-12.
```

5.2 参数优化

1) Zookeeper 会话超时时间

hbase-site.xml

属性: zookeeper.session.timeout

解释: 默认值为 90000 毫秒 (90s)。当某个 RegionServer 挂掉, 90s 之后 Master 才能察觉到。可适当减小此值, 尽可能快地检测 regionserver 故障, 可调整至 20-30s。

看你能有都能忍耐超时, 同时可以调整重试时间和重试次数

hbase.client.pause (默认值 100ms)

hbase.client.retries.number (默认 15 次)

2) 设置 RPC 监听数量

hbase-site.xml

属性: hbase.regionserver.handler.count

解释: 默认值为 30, 用于指定 RPC 监听的数量, 可以根据客户端的请求数进行调整, 读写请求较多时, 增加此值。

3) 手动控制 Major Compaction

hbase-site.xml

属性: hbase.hregion.majorcompaction

解释: 默认值: 604800000 秒 (7 天), Major Compaction 的周期, 若关闭自动 Major Compaction, 可将其设为 0。如果关闭一定记得自己手动合并, 因为大合并非常有意义

4) 优化 HStore 文件大小

hbase-site.xml

属性: hbase.hregion.max.filesize

解释: 默认值 10737418240 (10GB), 如果需要运行 HBase 的 MR 任务, 可以减小此值, 因为一个 region 对应一个 map 任务, 如果单个 region 过大, 会导致 map 任务执行时间

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载, 可百度访问: [尚硅谷官网](#)

过长。该值的意思就是，如果 HFile 的大小达到这个数值，则这个 region 会被切分为两个 Hfile。

5) 优化 HBase 客户端缓存

hbase-site.xml

属性: `hbase.client.write.buffer`

解释: 默认值 2097152bytes (2M) 用于指定 HBase 客户端缓存，增大该值可以减少 RPC 调用次数，但是会消耗更多内存，反之则反之。一般我们需要设定一定的缓存大小，以达到减少 RPC 次数的目的。

6) 指定 scan.next 扫描 HBase 所获取的行数

hbase-site.xml

属性: `hbase.client.scanner.caching`

解释: 用于指定 scan.next 方法获取的默认行数，值越大，消耗内存越大。

7) BlockCache 占用 RegionServer 堆内存的比例

hbase-site.xml

属性: `hfile.block.cache.size`

解释: 默认 0.4，读请求比较多的情况下，可适当调大

8) MemStore 占用 RegionServer 堆内存的比例

hbase-site.xml

属性: `hbase.regionserver.global.memstore.size`

解释: 默认 0.4，写请求较多的情况下，可适当调大

Lars Hofhansl (拉斯·霍夫汉斯) 大神推荐 Region 设置 20G，刷写大小设置 128M，其它默认。

5.3 JVM 调优

JVM 调优的思路有两部分：一是内存设置，二是垃圾回收器设置。

垃圾回收的修改是使用并发垃圾回收，默认 PO+PS 是并行垃圾回收，会有大量的暂停。理由是 HBase 大量使用内存用于存储数据，容易遭遇数据洪峰造成 OOM，同时写缓存的数据是不能垃圾回收的，主要回收的就是读缓存，而读缓存垃圾回收不影响性能，所以最终设置的效果可以总结为：防患于未然，早洗早轻松。

1) 设置使用 CMS 收集器:

```
-XX:+UseConcMarkSweepGC
```

2) 保持新生代尽量小，同时尽早开启 GC，例如:

```
//在内存占用到 70%的时候开启 GC
-XX:CMSInitiatingOccupancyFraction=70
//指定使用 70%，不让 JVM 动态调整
-XX:+UseCMSInitiatingOccupancyOnly
//新生代内存设置为 512m
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
-Xmn512m
//并行执行新生代垃圾回收
-XX:+UseParNewGC
// 设置 scanner 扫描结果占用内存大小，在 hbase-site.xml 中，设置
hbase.client.scanner.max.result.size(默认值为 2M)为 eden 空间的 1/8
(大概在 64M)

// 设置多个与 max.result.size * handler.count 相乘的结果小于 Survivor
Space(新生代经过垃圾回收之后存活的对象)
```

5.4 HBase 使用经验法则

官方给出了权威的使用法则：

- (1) Region 大小控制 10-50G
- (2) cell 大小不超过 10M（性能对应小于 100K 的值有优化），如果使用 mob（Medium-sized Objects 一种特殊用法）则不超过 50M。
- (3) 1 张表有 1 到 3 个列族，不要设计太多。最好就 1 个，如果使用多个尽量保证不会同时读取多个列族。
- (4) 1 到 2 个列族的表格，设计 50-100 个 Region。
- (5) 列族名称要尽量短，不要去模仿 RDBMS（关系型数据库）具有准确的名称和描述。
- (6) 如果 RowKey 设计时间在最前面，会导致有大量的旧数据存储在 inactive 的 Region 中，使用的时候，仅仅会操作少数的活动 Region，此时建议增加更多的 Region 个数。
- (7) 如果只有一个列族用于写入数据，分配内存资源的时候可以做出调整，即写缓存不会占用太多的内存。

第 6 章 整合 Phoenix

6.1 Phoenix 简介

6.1.1 Phoenix 定义

Phoenix 是 HBase 的开源 SQL 皮肤。可以使用标准 JDBC API 代替 HBase 客户端 API 来创建表，插入数据和查询 HBase 数据。

6.1.2 为什么使用 Phoenix

官方给的解释为：在 Client 和 HBase 之间放一个 Phoenix 中间层不会减慢速度，因为用户编写的数据处理代码和 Phoenix 编写的没有区别（更不用说你写的垃圾的多），不仅如此 Phoenix 对于用户输入的 SQL 同样会有大量的优化手段（就像 hive 自带 sql 优化器一样）。

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：[尚硅谷官网](#)

Phoenix 在 5.0 版本默认提供有两种客户端使用（瘦客户端和胖客户端），在 5.1.2 版本安装包中删除了瘦客户端，本文也不再使用瘦客户端。而胖客户端和用户自己写 HBase 的 API 代码读取数据之后进行数据处理是完全一样的。

6.2 Phoenix 快速入门

6.2.1 安装

1) 官网地址

<http://phoenix.apache.org/>

2) Phoenix 部署

(1) 上传并解压 tar 包

```
[atguigu@hadoop102 software]$ tar -zxvf phoenix-hbase-2.4-5.1.2-bin.tar.gz -C /opt/module/

[atguigu@hadoop102 module]$ mv phoenix-hbase-2.4-5.1.2-bin/phoenix
```

(2) 复制 server 包并拷贝到各个节点的 hbase/lib

```
[atguigu@hadoop102 module]$ cd /opt/module/phoenix/

[atguigu@hadoop102 phoenix]$ cp phoenix-server-hbase-2.4-5.1.2.jar /opt/module/hbase/lib/

[atguigu@hadoop102 phoenix]$ xsync /opt/module/hbase/lib/ phoenix-server-hbase-2.4-5.1.2.jar
```

(3) 配置环境变量

```
#phoenix
export PHOENIX_HOME=/opt/module/phoenix
export PHOENIX_CLASSPATH=$PHOENIX_HOME
export PATH=$PATH:$PHOENIX_HOME/bin
```

(4) 重启 HBase

```
[atguigu@hadoop102 ~]$ stop-hbase.sh
[atguigu@hadoop102 ~]$ start-hbase.sh
```

(5) 连接 Phoenix

```
[atguigu@hadoop101 phoenix]$ /opt/module/phoenix/bin/sqlline.py
hadoop102,hadoop103,hadoop104:2181
```

(6) 错误解决

出现下面错误的原因是之前使用过 phoenix，建议删除之前的记录

```
警告: Failed to load history
java.lang.IllegalArgumentException: Bad history file syntax! The
history file `/home/atguigu/.sqlline/history` may be an older
history: please remove it or use a different history file.
```

解决方法：在/home/atguigu 目录下删除.sqlline 文件夹

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
[atguigu@hadoop102 ~]$ rm -rf .sqlline/
```

6.2.2 Phoenix Shell 操作

6.2.2.1 table

关于 Phoenix 的语法建议使用时直接查看官网：

<https://phoenix.apache.org/language/index.html>

1) 显示所有表

```
!table 或 !tables
```

2) 创建表

直接指定单个列作为 RowKey

```
CREATE TABLE IF NOT EXISTS student(  
id VARCHAR primary key,  
name VARCHAR,  
age BIGINT,  
addr VARCHAR);
```

在 phoenix 中，表名等会自动转换为大写，若要小写，使用双引号，如"us_population"。

指定多个列的联合作为 RowKey

```
CREATE TABLE IF NOT EXISTS student1 (  
id VARCHAR NOT NULL,  
name VARCHAR NOT NULL,  
age BIGINT,  
addr VARCHAR  
CONSTRAINT my_pk PRIMARY KEY (id, name));
```

注：Phoenix 中建表，会在 HBase 中创建一张对应的表。为了减少数据对磁盘空间的占用，Phoenix 默认会对 HBase 中的列名做编码处理。具体规则可参考官网链接：
<https://phoenix.apache.org/columnencoding.html>，若不想对列名编码，可在建表语句末尾加上 `COLUMN_ENCODED_BYTES = 0;`

3) 插入数据

```
upsert into student values('1001','zhangsan', 10, 'beijing');
```

4) 查询记录

```
select * from student;  
select * from student where id='1001';
```

5) 删除记录

```
delete from student where id='1001';
```

6) 删除表

```
drop table student;
```

7) 退出命令行

```
!quit
```

6.2.2.2 表的映射

1) 表的关系

默认情况下，HBase 中已存在的表，通过 Phoenix 是不可见的。如果要在 Phoenix 中操作 HBase 中已存在的表，可以在 Phoenix 中进行表的映射。映射方式有两种：视图映射和表映射。

2) 命令行中创建表 test

HBase 中 test 的表结构如下，两个列族 info1、info2。

Rowkey	info1	info2
id	name	address

(1) 启动 HBase Shell

```
[atguigu@hadoop102 ~]$ /opt/module/hbase/bin/hbase shell
```

(2) 创建 HBase 表 test

```
hbase(main):001:0> create 'test','info1','info2'
```

3) 视图映射

Phoenix 创建的视图是只读的，所以只能用来做查询，无法通过视图对数据进行修改等操作。在 phoenix 中创建关联 test 表的视图

```
0: jdbc:phoenix:hadoop101,hadoop102,hadoop103> create view
"test"(id varchar primary key,"info1"."name" varchar,
"info2"."address" varchar);
```

删除视图

```
0: jdbc:phoenix:hadoop101,hadoop102,hadoop103> drop view "test";
```

4) 表映射

在 Phoenix 创建表去映射 HBase 中已经存在的表，是可以修改删除 HBase 中已经存在的数据的。而且，删除 Phoenix 中的表，那么 HBase 中被映射的表也会被删除。

注：进行表映射时，不能使用列名编码，需将 `column_encoded_bytes` 设为 0。

```
0: jdbc:phoenix:hadoop101,hadoop102,hadoop103> create table
"test"(id varchar primary key,"info1"."name" varchar,
"info2"."address" varchar) column_encoded_bytes=0;
```

6.2.2.3 数字类型说明

HBase 中的数字，底层存储为补码，而 Phoenix 中的数字，底层存储为在补码的基础上，将符号位反转。故当在 Phoenix 中建表去映射 HBase 中已存在的表，当 HBase 中有数字类型的字段时，会出现解析错误的现象。

Hbase 演示:

```
create 'test_number','info'
put 'test_number','1001','info:number',Bytes.toBytes(1000)
scan 'test_number',{COLUMNS => 'info:number:toLong'}
```

phoenix 演示:

```
create view "test_number"(id varchar primary key,"info"."number"
bigint);
select * from "test_number";
```

解决上述问题的方案有以下两种:

(1) Phoenix 提供了 `unsigned_int`, `unsigned_long` 等无符号类型, 其对数字的编码解码方式和 HBase 是相同的, 如果无需考虑负数, 那在 Phoenix 中建表时采用无符号类型是最合适的选择。

phoenix 演示:

```
drop view "test_number";
create view "test_number"(id varchar primary key,"info"."number"
unsigned_long);
select * from "test_number";
```

(2) 如需考虑负数的情况, 则可通过 Phoenix 自定义函数, 将数字类型的最高位, 即符号位反转即可, 自定义函数可参考如下链接: <https://phoenix.apache.org/udf.html>。

6.2.3 Phoenix JDBC 操作

此处演示一个标准的 JDBC 连接操作, 实际开发中会直接使用别的框架内嵌的 Phoenix 连接。

1) 胖客户端

(1) maven 依赖

```
<dependencies>
  <dependency>
    <groupId>org.apache.phoenix</groupId>
    <artifactId>phoenix-client-hbase-2.4</artifactId>
    <version>5.1.2</version>
  </dependency>
</dependencies>
```

(2) 编写代码

```
package com.atguigu.phoenix;

import java.sql.*;
import java.util.Properties;

public class PhoenixClient {
    public static void main(String[] args) throws SQLException {
        // 标准的 JDBC 代码
        // 1. 添加链接
```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载, 可百度访问: [尚硅谷官网](#)


```
String url =
"jdbc:phoenix:hadoop102,hadoop103,hadoop104:2181";

// 2. 创建配置
// 没有需要添加的必要配置 因为 Phoenix 没有账号密码
Properties properties = new Properties();

// 3. 获取连接
Connection connection = DriverManager.getConnection(url,
properties);
// 5. 编译 SQL 语句
PreparedStatement preparedStatement =
connection.prepareStatement("select * from student");
// 6. 执行语句
ResultSet resultSet = preparedStatement.executeQuery();
// 7. 输出结果
while (resultSet.next()) {
    System.out.println(resultSet.getString(1) + ":" +
resultSet.getString(2) + ":" + resultSet.getString(3));
}

// 8. 关闭资源
connection.close();

// 由于 Phoenix 框架内部需要获取一个 HBase 连接, 所以会延迟关闭
// 不影响后续的代码执行
System.out.println("hello");
}
}
```

6.3 Phoenix 二级索引

6.3.1 二级索引配置文件

添加如下配置到 HBase 的 HRegionserver 节点的 hbase-site.xml。

```
<!-- phoenix regionserver 配置参数-->
<property>
    <name>hbase.regionserver.wal.codec</name>

    <value>org.apache.hadoop.hbase.regionserver.wal.IndexedWALEditCod
ec</value>
</property>
```

6.3.2 全局索引 (global index)

Global Index 是**默认的索引格式**，创建全局索引时，会在 HBase 中**建立一张新表**。也就是说索引数据和数据表是存放在不同的表中的，因此全局索引适用于**多读少写**的业务场景。

写数据的时候会消耗大量开销，因为索引表也要更新，而索引表是分布在不同的数据节点上的，跨节点的数据传输带来了较大的性能消耗。

在读数据的时候 Phoenix 会选择索引表来降低查询消耗的时间。

创建单个字段的全局索引。

```
CREATE INDEX my_index ON my_table (my_col);
#例如
0: jdbc:phoenix:hadoop102,hadoop103,hadoop104> create index
my_index on student1(age);

#删除索引
DROP INDEX my_index ON my_table
0: jdbc:phoenix:hadoop102,hadoop103,hadoop104> drop index my_index
on student1;
```

查看二级索引是否有效，可以使用 explainPlan 执行计划，有二级索引之后会变成范围扫描

```
0: jdbc:phoenix:hadoop102,hadoop103,hadoop104> explain select
id,name from student1 where age = 10;

+-----+-----+-----+-----+
|                                     PLAN                                     |
+-----+-----+-----+-----+
| EST_BYTES_READ | EST_ROWS_READ | EST_INF | |
+-----+-----+-----+-----+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN RANGE SCAN OVER MY_INDEX |
| [10] | null | null | null | |
| SERVER FILTER BY FIRST KEY ONLY |
| null | null | null | |
+-----+-----+-----+-----+
2 rows selected (0.044 seconds)
```

如果想查询的字段不是索引字段的话索引表不会被使用，也就是说不会带来查询速度的提升。

```
0: jdbc:phoenix:hadoop102,hadoop103,hadoop104> explain select
id,name,addr from student1 where age = 10;
+-----+-----+-----+-----+
|                                     PLAN                                     |
+-----+-----+-----+-----+
| EST_BYTES_READ | EST_ROWS_READ | EST_INFO_TS | |
+-----+-----+-----+-----+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN FULL SCAN OVER STUDENT1 |
| null | null | null | |
| SERVER FILTER BY AGE = 10 |
| null | null | null | |
+-----+-----+-----+-----+
2 rows selected (0.024 seconds)
```

若想解决上述问题，可采用如下方案：

- (1) 使用包含索引

(2) 使用本地索引

6.3.3 包含索引 (covered index)

创建携带其他字段的全局索引（本质还是全局索引）。

```
CREATE INDEX my_index ON my_table (v1) INCLUDE (v2);
```

先删除之前的索引：

```
0: jdbc:phoenix:hadoop102,hadoop103,hadoop104> drop index my_index
on student1;
#创建包含索引
0: jdbc:phoenix:hadoop102,hadoop103,hadoop104> create index
my_index on student1(age) include (addr);
```

之后使用执行计划查看效果

```
0: jdbc:phoenix:hadoop102,hadoop103,hadoop104> explain select
id,name,addr from student1 where age = 10;
+-----+-----+-----+-----+
|                                     PLAN                                     |
+-----+-----+-----+-----+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN RANGE SCAN OVER MY_INDEX |
[10] | null | null | null |
+-----+-----+-----+-----+
1 row selected (0.112 seconds)
```

6.3.4 本地索引 (local index)

Local Index 适用于写操作频繁的场景。

索引数据和数据表的数据是存放在同一张表中（且是同一个 Region），避免了在写操作的时候往不同服务器的索引表中写索引带来的额外开销。

my_column 可以是多个。

```
CREATE LOCAL INDEX my_index ON my_table (my_column);
```

本地索引会将所有的信息存在一个影子列族中，虽然读取的时候也是范围扫描，但是没有全局索引快，优点在于不用写多个表了。

```
#删除之前的索引
0: jdbc:phoenix:hadoop102,hadoop103,hadoop104> drop index my_index
on student1;
#创建本地索引
0: jdbc:phoenix:hadoop102,hadoop103,hadoop104> CREATE LOCAL INDEX
my_index ON student1 (age,addr);
#使用执行计划
0: jdbc:phoenix:hadoop102,hadoop103,hadoop104> explain select
id,name,addr from student1 where age = 10;
+-----+-----+-----+-----+
|                                     PLAN                                     |
+-----+-----+-----+-----+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN RANGE SCAN OVER MY_INDEX |
[10] | null | null | null |
+-----+-----+-----+-----+
1 row selected (0.112 seconds)
```

```

-----+-----+-----+-----+
|                                     PLAN                                     |
| EST_BYTES_READ | EST_ROWS_READ | EST_I |
+-----+-----+-----+-----+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN RANGE SCAN OVER STUDENT1
| [2,10] | null | null | null |
|                                     SERVER MERGE [0.ADDR]
| null | null | null |
|                                     SERVER FILTER BY FIRST KEY ONLY
| null | null | null |
+-----+-----+-----+-----+
3 rows selected (0.025 seconds)

```

第 7 章 与 Hive 的集成

7.1 使用场景

如果大量的数据已经存放在 HBase 上面，需要对已经存在的数据进行数据分析处理，那么 Phoenix 并不适合做特别复杂的 SQL 处理，此时可以使用 hive 映射 HBase 的表格，之后写 HQL 进行分析处理。

7.2 HBase 与 Hive 集成使用

在 `hive-site.xml` 中添加 zookeeper 的属性，如下：

```

<property>
  <name>hive.zookeeper.quorum</name>
  <value>hadoop102,hadoop103,hadoop104</value>
</property>

<property>
  <name>hive.zookeeper.client.port</name>
  <value>2181</value>
</property>

```

1) 案例一

目标：建立 Hive 表，关联 HBase 表，插入数据到 Hive 表的同时能够影响 HBase 表。

分步实现：

(1) 在 Hive 中创建表同时关联 HBase

```

CREATE TABLE hive_hbase_emp_table(
  empno int,
  ename string,
  job string,
  mgr int,
  hiredate string,
  sal double,
  comm double,
  deptno int
)

```

更多 [Java](#) - [大数据](#) - [前端](#) - [python](#) 人工智能资料下载，可百度访问：尚硅谷官网

```
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'  
WITH SERDEPROPERTIES ("hbase.columns.mapping" =  
":key,info:ename,info:job,info:mgr,info:hiredate,info:sal,info:comm,info:deptno")  
TBLPROPERTIES ("hbase.table.name" = "hbase_emp_table");
```

提示：完成之后，可以分别进入 Hive 和 HBase 查看，都生成了对应的表。

(2) 在 Hive 中创建临时中间表，用于 load 文件中的数据

提示：不能将数据直接 load 进 Hive 所关联 HBase 的那张表中。

```
CREATE TABLE emp(  
  empno int,  
  ename string,  
  job string,  
  mgr int,  
  hiredate string,  
  sal double,  
  comm double,  
  deptno int  
)  
row format delimited fields terminated by '\t';
```

(3) 向 Hive 中间表中 load 数据

```
hive> load data local inpath '/opt/software/emp.txt' into table emp;
```

(4) 通过 insert 命令将中间表中的数据导入到 Hive 关联 Hbase 的那张表中

```
hive> insert into table hive_hbase_emp_table select * from emp;
```

(5) 查看 Hive 以及关联的 HBase 表中是否已经成功的同步插入了数据

Hive:

```
hive> select * from hive_hbase_emp_table;
```

HBase:

```
Hbase> scan 'hbase_emp_table'
```

2) 案例二

目标：在 HBase 中已经存储了某一张表 hbase_emp_table，然后在 Hive 中创建一个外部表来关联 HBase 中的 hbase_emp_table 这张表，使之可以借助 Hive 来分析 HBase 这张表中的数据。

注：该案例 2 紧跟案例 1 的脚步，所以完成此案例前，请先完成案例 1。

分步实现：

(1) 在 Hive 中创建外部表

```
CREATE EXTERNAL TABLE relevance_hbase_emp(  
  empno int,  
  ename string,  
  job string,  
  mgr int,  
  hiredate string,  
  sal double,
```

```
    comm double,  
    deptno int  
)  
STORED BY  
'org.apache.hadoop.hive.hbase.HBaseStorageHandler'  
WITH SERDEPROPERTIES ("hbase.columns.mapping" =  
":key,info:ename,info:job,info:mgr,info:hiredate,info:sal,info:comm,info:deptno")  
TBLPROPERTIES ("hbase.table.name" = "hbase_emp_table");
```

(2) 关联后就可以使用 Hive 函数进行一些分析操作了

```
hive (default)> select deptno, avg(sal) monery from  
relevance_hbase_emp group by deptno ;
```