# 1. HBase Filter 介绍

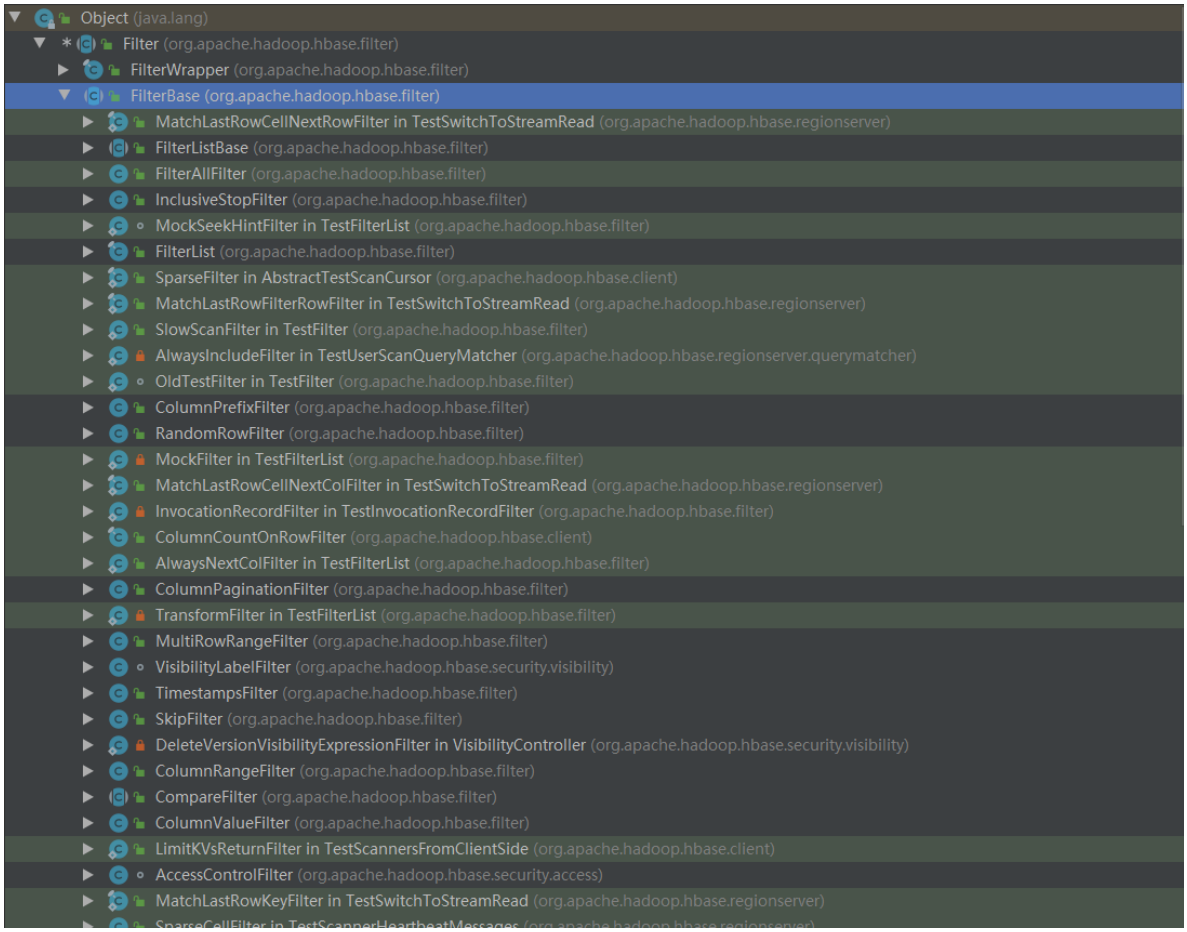
Hbase 提供了种类丰富的过滤器（filter）来提高数据处理的效率，用户可以通过内置或自定义的过滤器来对数据进行过滤，所有的过滤器都在服务端生效，即谓词下推（predicate push down）。这样可以保证过滤掉的数据不会被传送到客户端，从而减轻网络传输和客户端处理的压力。

## 1.1. Filter接口和FilterBase抽象类

Filter 接口中定义了过滤器的基本方法，FilterBase 抽象类实现了 Filter 接口。所有内置的过滤器则直接或者间接继承自 FilterBase 抽象类。用户只需要将定义好的过滤器通过 setFilter 方法传递给 Scan 或 put 的实例即可。

```
setFilter(Filter filter)
```

FilterBase 的所有子类过滤器如下：




HBase 内置过滤器可以分为三类：分别是比较过滤器，专用过滤器和包装过滤器。

# 1.2. 过滤器查询

引言：过滤器的类型很多，但是可以分为两大类——比较过滤器，专用过滤器

过滤器的作用是在服务端判断数据是否满足条件，然后只将满足条件的数据返回给客户端；

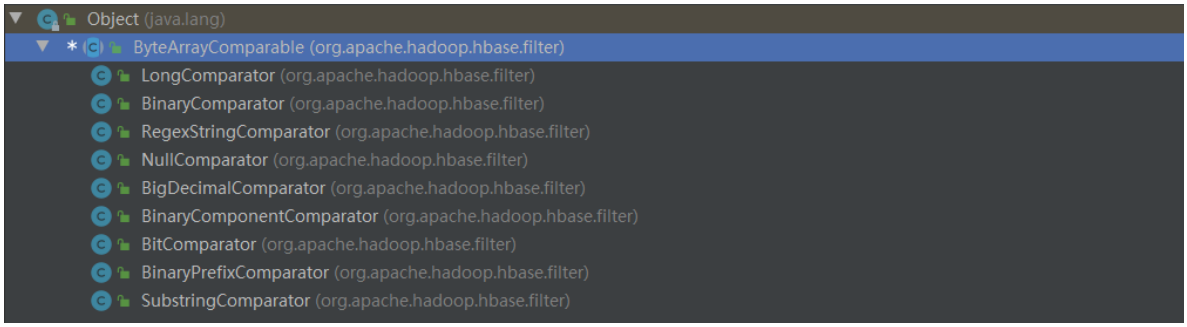
## 1.2.1. HBase过滤器的比较运算符

CompareFilter 中的内部枚举类 CompareOp 中定义了 7 种比较运算符，在 HBase-3.x 将会被 CompareOperator 取代

```
LESS                    <
LESS_OR_EQUAL           <=
EQUAL                   =
NOT_EQUAL               <>
GREATER_OR_EQUAL        >=
GREATER                 >
NO_OP                   不做任何操作
```

## 1.2.2. HBase过滤器的比较器（指定比较机制）

所有比较器均继承自 ByteArrayComparable 抽象类，常用的有以下几种：




| | |
|---|---|
| BinaryComparator | 按字节索引顺序比较指定字节数组，采用 Bytes.compareTo(byte[]) |
| BinaryPrefixComparator | 跟前面相同，只是比较左端的数据是否相同 |
| NullComparator | 判断给定的是否为空 |
| BitComparator | 按位比较 |
| RegexStringComparator | 提供一个正则的比较器，仅支持 EQUAL 和 NOT_EQUAL |
| SubstringComparator | 判断提供的子串是否出现在 value 中，仅支持 EQUAL 和 NOT_EQUAL |

BinaryPrefixComparator 和 BinaryComparator 的区别不是很好理解，这里举例说明一下：

在进行 EQUAL 的比较时，如果比较器传入的是 abcd 的字节数组，但是待比较数据是 abcdefgh ：

1、如果使用的是 BinaryPrefixComparator 比较器，则比较以 abcd 字节数组的长度为准，即efgh 不会参与比较，这时候认为 abcd 与 abcdefgh 是满足 EQUAL 条件的；
2、如果使用的是 BinaryComparator 比较器，则认为其是不相等的。

## 1.2.3. 比较过滤器

所有比较过滤器均继承自 CompareFilter 。创建一个比较过滤器需要两个参数，分别是比较运算符和比较器实例。

**行键过滤器RowFilter**

```
Filter filter1 = new RowFilter(CompareOp.LESS_OR_EQUAL, new
BinaryComparator(Bytes.toBytes("user0000")));
scan.setFilter(filter1);
```

**列簇过滤器FamilyFilter**

```
Filter filter1 = new FamilyFilter(CompareOp.LESS,  new
BinaryComparator(Bytes.toBytes("base_info")));
scan.setFilter(filter1);
```

**列过滤器QualifierFilter**

```
Filter filter = new QualifierFilter(CompareOp.LESS_OR_EQUAL,  new
BinaryComparator(Bytes.toBytes("name")));
scan.setFilter(filter1);
```

**值过滤器 ValueFilter**

```
Filter filter = new ValueFilter(CompareOp.EQUAL, new
SubstringComparator("zhangsan") );
scan.setFilter(filter1);
```

**时间戳过滤器 DependentColumnFilter**：指定一个参考列来过滤其他列的过滤器，过滤的原则是基于参考列 的时间戳来进行筛选 。

可以把 DependentColumnFilter 理解为一个 valueFilter 和一个时间戳过滤器的组合。
DependentColumnFilter 有三个带参构造器，这里选择一个参数最全的进行说明：

```
public DependentColumnFilter(final byte [] family, final byte[] qualifier,
                             final boolean dropDependentColumn, final
CompareOperator op,
                             final ByteArrayComparable valueComparator) {
    // set up the comparator
    super(op, valueComparator);
    this.columnFamily = family;
    this.columnQualifier = qualifier;
    this.dropDependentColumn = dropDependentColumn;
}
```

解释：

```
family ： 列族
qualifier ： 列限定符（列名）
dropDependentColumn ： 决定参考列是否被包含在返回结果内，为 true 时表示参考列被返回，为
false 时表示被丢弃
op ： 比较运算符
valueComparator ： 比较器
```

这里举例进行说明：

```
DependentColumnFilter dependentColumnFilter = new DependentColumnFilter(
    Bytes.toBytes("student"),
    Bytes.toBytes("name"),
    false,
    CompareOperator.EQUAL,
    new BinaryPrefixComparator(Bytes.toBytes("huangbo"))
);
```

解释：

```
1、首先会去查找 student:name 中值以 huangbo 开头的所有数据获得参考数据集 ，这一步等同于
valueFilter过滤器；
2、其次再用参考数据集中所有数据的时间戳去检索其他列，获得时间戳相同的其他列的数据作为结果数据
集，这一步等同于时间戳过滤器；
3、最后如果 dropDependentColumn 为true，则返回参考数据集 + 结果数据集，若为false，则抛弃
参考数据集，只返回结果数据集。
```

## 1.2.4. 专用过滤器

**单列值过滤器 SingleColumnValueFilter** ----会返回满足条件的整行

基于某列（参考列）的值决定某行数据是否被过滤。其实例有以下方法：

```
setFilterIfMissing(boolean filterIfMissing) ：默认值为 false，即如果该行数据不包含参考列，其依然被包含在最后的结果中；设置为 true 时，则不包含；
setLatestVersionOnly(boolean latestVersionOnly) ：默认为 true，即只检索参考列的最新版本数据；设置为 false，则检索所有版本数据。
```

实例：

```
SingleColumnValueFilter filter = new SingleColumnValueFilter(
    Bytes.toBytes("colfam1"),
    Bytes.toBytes("col-5"),
    CompareFilter.CompareOp.NOT_EQUAL,
    new SubstringComparator("val-5"));
filter.setFilterIfMissing(true);  //如果不设置为true，则那些不包含指定column的行也会返回
scan.setFilter(filter1);
```

**单列值排除器SingleColumnValueExcludeFilter** -----返回排除了该列的结果

```
SingleColumnValueExcludeFilter 继承自上面的 SingleColumnValueFilter ，过滤行为与其相反。
```

**前缀过滤器 PrefixFilter**----针对行键，基于 RowKey 值决定某行数据是否被过滤。

```
Filter filter = new PrefixFilter(Bytes.toBytes("rk"));
scan.setFilter(filter1);
```

**列前缀过滤器 ColumnPrefixFilter**：基于列限定符（列名）决定某行数据是否被过滤。

```
Filter filter = new ColumnPrefixFilter(Bytes.toBytes("qual2"));
scan.setFilter(filter1);
```

**分页过滤器PageFilter**：可以使用这个过滤器实现对结果按行进行分页，创建 PageFilter 实例的时候需要传入每页的行数。

```
public PageFilter(final long pageSize) {
    Preconditions.checkArgument(pageSize >= 0, "must be positive %s", pageSize);
    this.pageSize = pageSize;
 }
```

**时间戳过滤器 (TimestampsFilter)**

```
List<Long> list = new ArrayList<>();
list.add(1554975573000L);
TimestampsFilter timestampsFilter = new TimestampsFilter(list);
scan.setFilter(timestampsFilter);
```

**首次行键过滤器 (FirstKeyOnlyFilter)**

FirstKeyOnlyFilter 只扫描每行的第一列，扫描完第一列后就结束对当前行的扫描，并跳转到下一 行。相比于全表扫描，其性能更好，通常用于行数统计的场景，因为如果某一行存在，则行中必然至少 有一列。

```
FirstKeyOnlyFilter firstKeyOnlyFilter = new FirstKeyOnlyFilter();
scan.set(firstKeyOnlyFilter);
```

## 1.2.5. 包装过滤器

包装过滤器就是通过包装其他过滤器以实现某些拓展的功能。

SkipFilter过滤器

SkipFilter 包装一个过滤器，当被包装的过滤器遇到一个需要过滤的 KeyValue 实例时，则拓展过滤 整行数据。下面是一个使用示例：

```
// 定义 ValueFilter 过滤器
Filter filter1 = new ValueFilter(CompareOperator.NOT_EQUAL, new
BinaryComparator(Bytes.toBytes("xxx")));
// 使用 SkipFilter 进行包装
Filter filter2 = new SkipFilter(filter1);
```

WhileMatchFilter过滤器

WhileMatchFilter 包装一个过滤器，当被包装的过滤器遇到一个需要过滤的 KeyValue 实例时，WhileMatchFilter 则结束本次扫描，返回已经扫描到的结果。下面是其使用示例：

```
Filter filter1 = new RowFilter(CompareOperator.NOT_EQUAL, new
BinaryComparator(Bytes.toBytes("rk01")));
Scan scan = new Scan();
scan.setFilter(filter1);
ResultScanner scanner1 = table.getScanner(scan);
for (Result result : scanner1) {
    for (Cell cell : result.listCells()) {
        System.out.println(cell);
    }
}
scanner1.close();
System.out.println("----------------------------------");
// 使用 WhileMatchFilter 进行包装
Filter filter2 = new WhileMatchFilter(filter1);
scan.setFilter(filter2);
```

```
ResultScanner scanner2 = table.getScanner(scan);
for (Result result : scanner1) {
    for (Cell cell : result.listCells()) {
        System.out.println(cell);
    }
}
scanner2.close();
```

## 1.3. FilterList

以上都是讲解单个过滤器的作用，当需要多个过滤器共同作用于一次查询的时候，就需要使用 FilterList 。 FilterList 支持通过构造器或者 addFilter 方法传入多个过滤器。

```
// 构造器传入
public FilterList(final Operator operator, final List<Filter> filters)
public FilterList(final List<Filter> filters)
public FilterList(final Filter... filters)

// 方法传入
public void addFilter(List<Filter> filters)
public void addFilter(Filter filter)
```

多个过滤器组合的结果由 operator 参数定义，其可选参数定义在 Operator 枚举类中。只有 MUST_PASS_ALL 和 MUST_PASS_ONE 两个可选的值：

```
MUST_PASS_ALL ：相当于 AND，必须所有的过滤器都通过才认为通过；
MUST_PASS_ONE ：相当于 OR，只有要一个过滤器通过则认为通过。
```

```
public enum Operator {
    /** !AND */
    MUST_PASS_ALL,
    /** !OR */
    MUST_PASS_ONE
}
```

使用示例如下：

```
List<Filter> filters = new ArrayList<Filter>();
Filter filter1 = new RowFilter(CompareOperator.GREATER_OR_EQUAL,new
BinaryComparator(Bytes.toBytes("abc")));
filters.add(filter1);
Filter filter2 = new RowFilter(CompareOperator.LESS_OR_EQUAL, new
BinaryComparator(Bytes.toBytes("YYY")));
filters.add(filter2);
Filter filter3 = new QualifierFilter(CompareOperator.EQUAL, new
RegexStringComparator("ZZZ"));
filters.add(filter3);
FilterList filterList = new FilterList(filters);
Scan scan = new Scan();
scan.setFilter(filterList);
```

更多资料参考：HBase权威指南 第四章

# 2. HBase Filter Shell

见shell文档

# 3. HBase Filter JavaAPI

见java代码

# 4. HBase PageFilter

下面的代码体现了客户端实现分页查询的主要逻辑，这里对其进行一下解释说明：

> 客户端进行分页查询，需要传递 startRow (起始 RowKey)，知道起始 startRow 后，就可以返回对应的 pageSize 行数据。这里唯一的问题就是，对于第一次查询，显然 startRow 就是表格的第一行数据，但是之后第二次、第三次查询我们并不知道 startRow ，只能知道上一次查询的最后一条数据的 RowKey（简单称之为 lastRow ）。
> 我们不能将 lastRow 作为新一次查询的 startRow 传入，因为 scan 的查询区间是[startRow, endRow) ，即前开后闭区间，这样 startRow 在新的查询也会被返回，这条数据就重复了。同时在不使用第三方数据库存储 RowKey 的情况下，我们是无法通过知道 lastRow 的下一个 RowKey的，因为 RowKey 的设计可能是连续的也有可能是不连续的。
> 由于 Hbase 的 RowKey 是按照字典序进行排序的。这种情况下，就可以在 lastRow 后面加上 0 ，作为 startRow 传入，因为按照字典序的规则，某个值加上 0 后的新值，在字典序上一定是这个值的下一个值，对于 HBase 来说下一个 RowKey 在字典序上一定也是等于或者大于这个新值的。
> 所以最后传入 lastRow + 0 ，如果等于这个值的 RowKey 存在就从这个值开始 scan,否则从字典序的下一个 RowKey 开始 scan。

> 需要注意的是在多台 Regin Services 上执行分页过滤的时候，由于并行执行的过滤器不能共享它们的状态和边界，所以有可能每个过滤器都会在完成扫描前获取了 PageCount 行的结果，这种情况下会返回比分页条数更多的数据，分页过滤器就有失效的可能。

## 4.1. 分页过滤器 PageFilter

```
package com.mazh.hbase.core.nx;

import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.filter.Filter;
import org.apache.hadoop.hbase.filter.PageFilter;
import org.apache.hadoop.hbase.util.Bytes;
```

```
/**
 * 作者：  马中华   https://blog.csdn.net/zhongqi2513
 * 时间：  2017/11/16 16:45
 * 描述：  测试HBase的分页查询
 */
public class Hbase_Page01 {

    private static final String ZK_CONNECT_STR =
"bigdata02:2181,bigdata03:2181,bigdata04:2181";

    private static final String TABLE_NAME = "user_info";
    private static final String FAMILY_BASIC = "base_info";
    private static final String FAMILY_EXTRA = "extra_info";
    private static final String COLUMN_NAME = "name";
    private static final String COLUMN_AGE = "age";
    private static final String ROW_KEY = "rk0001";

    private static Configuration config = null;
    private static HTable table = null;

    static {
        config = HBaseConfiguration.create();
        config.set("hbase.zookeeper.quorum", ZK_CONNECT_STR);
        try {
            table = new HTable(config, TABLE_NAME);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) throws Exception {
        ResultScanner pageData = getPageData(2, 3);
        HBasePrintUtil.printResultScanner(pageData);
    }

    public static ResultScanner getPageData(int pageIndex, int pageNumber)
throws Exception {
        // 怎么把pageIndex 转换成 startRow
        String startRow = null;

        if (pageIndex == 1) { // 当客户方法只取第一页的分页数据时，
            ResultScanner pageData = getPageData(startRow, pageNumber);
            return pageData;

        } else {
            ResultScanner newPageData = null;
            for (int i = 0; i < pageIndex - 1; i++) { // 总共循环次数是比你取的页数少
1
                newPageData = getPageData(startRow, pageNumber);
                startRow = getLastRowkey(newPageData);
                byte[] add = Bytes.add(Bytes.toBytes(startRow), new byte[]
{0x00});
                startRow = Bytes.toString(add);
            }
            newPageData = getPageData(startRow, pageNumber);
            return newPageData;
        }
    }
```

```
    /**
     * scan 'user_info',{COLUMNS => 'base_info:name',LIMIT => 4, STARTROW =>
'zhangsan_20150701_0001'}
     */
    public static ResultScanner getPageData(String startRow, int pageNumber)
throws Exception {
        Scan scan = new Scan();
        scan.addColumn(Bytes.toBytes("base_info"), Bytes.toBytes("name"));
        // 設置當前查询的其实位置
        if (!StringUtils.isBlank(startRow)) {
            scan.setStartRow(Bytes.toBytes(startRow));
        }
        // 第二个参数
        Filter pageFilter = new PageFilter(pageNumber);
        scan.setFilter(pageFilter);

        ResultScanner rs = table.getScanner(scan);
        return rs;
    }

    public static String getLastRowkey(ResultScanner rs) {
        String lastRowkey = null;
        for (Result result : rs) {
            lastRowkey = Bytes.toString(result.getRow());
        }
        return lastRowkey;
    }
}
```