第3讲 Hadoop IO操作

辜希武
IDC实验室
1694551702@qq.com

主要内容

- I/O操作中的数据完整性检查
- I/O操作中的数据压缩
- I/O操作中的数据序列化
- 基于文件的数据结构

HADOOP中的IO操作

- 在Hadoop提供了如下与I/O相关的JAVA API包

org.apache.hadoop.io

org.apache.hadoop.io.compress

org.apache.hadoop.io.file.tfile

org.apache.hadoop.io.serializer.avro

  - 除了org.apache.hadoop.io.serializer.avro是用于为Avro（与Hadoop相关的Apache另一个顶级项目）提供序列化操作外，其余都是用于Hadoop的I/O操作
  - 部分fs包中的类与I/O也有关

HADOOP中的IO操作特殊之处

- 在Hadoop提供的I/O操作与普通I/O操作不同之处在
  - 传统的计算机系统数据是集中的，Hadoop系统中的数据是分布在不同计算机上
  - 传统计算机系统数据量相对较小（GB），Hadoop处理的数据通常到了PB级别
- 因此Hadoop中I/O操作要考虑些特殊问题
  - 不仅要考虑本地I/O成本，还要考虑数据在不同主机之前的传输成本
  - 数据分布在多台机器上，数据在传输、存储中出错的可能性大大增加，所以要进行数据完整性检查
  - 对于大容量的分布式存储系统，文件压缩是必须的，这样做带来的好处是1）减小了文件存储所需的磁盘空间；2）加快了文件在网络上和磁盘间的传输速度
  - 无论是存储文件还是在网络上传输数据，都需要执行数据的序列化/反序列化。序列化是指将内存中的对象转换成字节流，反序列化指将字节流恢复成内存中的对象。因此序列化的速度，序列化后数据的大小会影响数据传输速度。因此Hadoop没有采用Java提供的序列化机制，而是自己实现了一个序列化机制Writable
  - 针对二进制数据的大对象（blob），设计了更高层次的、基于文件的数据结构

HADOOP中的IO操作特殊之处

- 在Hadoop提供的I/O操作与普通I/O操作不同之处在
  - 传统的计算机系统数据是集中的，Hadoop系统中的数据是分布在不同计算机上
  - 传统计算机系统数据量相对较小（GB），Hadoop处理的数据通常到了PB级别
- 因此Hadoop中I/O操作要考虑些特殊问题
  - 不仅要考虑本地I/O成本，还要考虑数据在不同主机之前的传输成本
  - 数据分布在多台机器上，数据在传输、存储中出错的可能性大大增加，所以要进行数据完整性检查
  - 对于大容量的分布式存储系统，文件压缩是必须的，这样做带来的好处是1）减小了文件存储所需的磁盘空间；2）加快了文件在网络上和磁盘间的传输速度
  - 无论是存储文件还是在网络上传输数据，都需要执行数据的序列化/反序列化。序列化是指将内存中的对象转换成字节流，反序列化指将字节流恢复成内存中的对象。因此序列化的速度，序列化后数据的大小会影响数据传输速度。因此Hadoop没有采用Java提供的序列化机制，而是自己实现了一个序列化机制Writable
  - 针对二进制数据的大对象（blob），设计了更高层次的、基于文件的数据结构

HADOOP IO操作中的数据完整性检查

- 因为Hadoop采用HDFS作为默认的文件系统，所以需要考虑二个方面的数据完整性
  - LocalFileSystem的数据完整性
    - block最终是存储在本地文件系统
  - HDFS的数据完整性
  - Hadoop采用CRC-32校验和来检查数据完整性
- LocalFileSystem的完整性校验
  - 在Hadoop中，LocalFileSystem完成的客户端的数据校验，重点是在存储和读取文件时进行校验和处理
  - 具体做法是：每当Hadoop创建文件a是，Hadoop会同时在同一文件夹下创建隐藏文件a.crc，这个文件记录了文件a的校验和，每512个字节就会产生32位的校验和。可以在$HADOOP_HOME/src/core/core-default.xml中设置io.bytes.per.checksum来修改每多少个字节产生一个校验和

Hadoop数据完整性检查的全局设置

- 设置io.bytes.per.checksum来修改每多少个字节产生一个校验和

```
<property>
        <name> io.bytes.per.checksum</name>
        <value>512</value>
</property>
```

- 通过修改core-defalut.xml中的fs.file.impl属性来启用/禁用校验和机制

```
<property>
        <name> fs.file.impl </name>
        <value>org.apache.hadoop.fs.LocalFileSystem</value>
</property>
```

启用校验和

```
<property>
        <name> fs.file.impl </name>
        <value>org.apache.hadoop.fs.RawLocalFileSystem</value>
</property>
```

禁用校验和

RawLocalFileSystem

- 如果只想针对某些操作禁用校验和（某个程序中）

```
Configuration conf = new Configuration();
FileSystem fs = new RawLocalFileSystem();
fs.initialize(URI.create("file:///home/hadoop/test.txt"), conf);
...
```

- RawLocalFileSystem没有校验和功能

ChecksumFileSystem

- LocalFileSystem使用ChecksumFileSystem来完成数据校验工作。Hadoop的ChecksumFileSystem类实现了校验和机制

```
java.lang.Object
  - org.apache.hadoop.conf.Configured
    -org.apache.hadoop.fs.FileSystem
      -org.apache.hadoop.fs.FilterFileSystem
        -org.apache.hadoop.fs.ChecksumFileSystem
          -org.apache.hadoop.fs.LocalFileSystem
```

- 从以上继承关系可以看出，LocalFileSystem具有校验和机制

ChecksumFileSystem

- RawLocalFileSystem和ChecksumFileSystem类一起使用，可以达到LocalFileSystem同样的效果

```
Configuration conf = new Configuration();
FileSystem rawFS = new RawLocalFileSystem();
FileSystem checksumFS = new ChecksumFileSystem(rawFS);
fs.initialize(URI.create("file:///home/hadoop/test.txt"), conf);
…
```

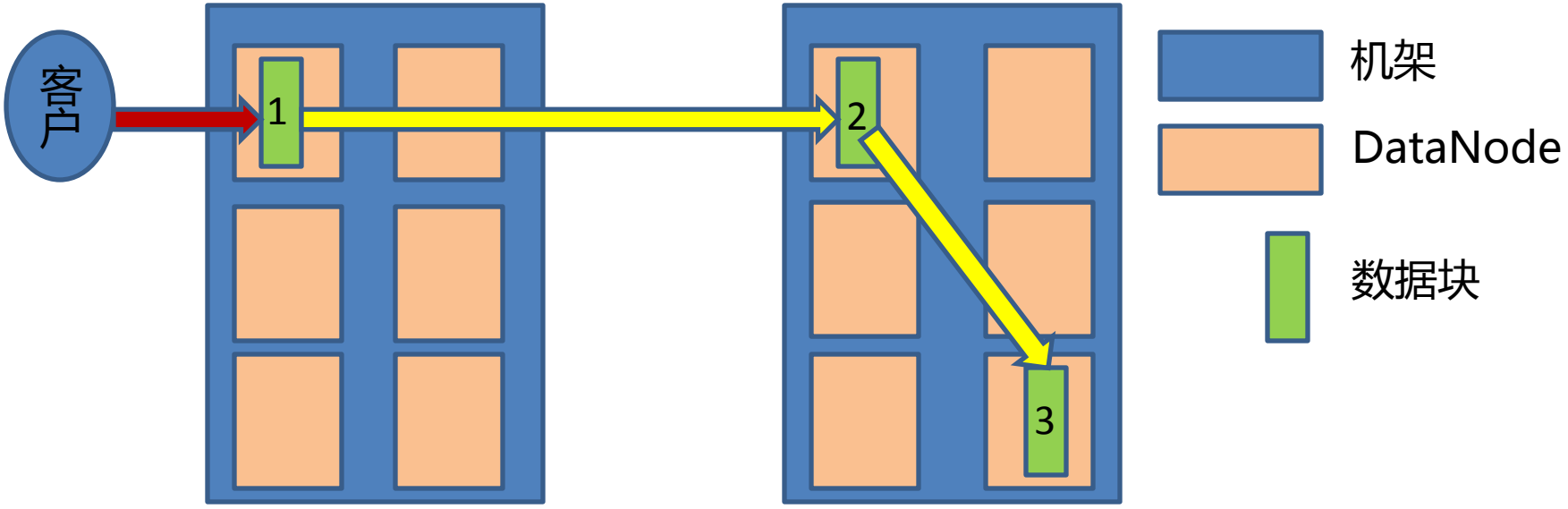
- 最底层的文件系统为Raw文件系统，可以使用ChecksumFileSystem实例的getRawFileSystem( )方法来获取Raw文件系统对象
- 使用ChecksumFileSystem实例的getChecksumFile( )方法获取一个文件的校验和文件路径
- 如果在读取文件是发生校验和错误， ChecksumFileSystem会调用reportChecksumFailure( )方法，但默认实现为空方法。LocalFileSystem会把出错的 文件及校验和移到bad_files文件夹中

HDFS的数据完整性检查

- HDFS会在三种情况下检验校验和

1）DataNode接受数据后，存储数据前

在客户端写数据时，Hadoop会形成一个数据管道(PipeLine)。



Hadoop不会在数据每流动到一个DataNode时都检查校验和，它只会在数据流动到最后一个节点时在检查校验和，即会在备份3所在的DataNode接受完数据后检查校验和。如果它检查到错误，Hadoop会抛出ChecksumException给客户

HDFS的数据完整性检查

- HDFS会在三种情况下检验校验和

2）客户端读取DataNode上的数据时

当客户端从DataNode读取数据时，客户端需要检查校验和（缺省配置是需要检查）。注意在每个DataNode上都会保存检查和，同时每个数据节点会保存检查和校验日志，这样可以知道每个block最后的校验时间。如果客户端对一个block校验成功，会通知DataNode更新校验日志。

3）DataNode后台守护进程的定期检测

每个DataNode会周期性地在后台运行DataBlockScanner，检查所有数据块的校验和。

HDFS如何禁止数据完整性检查（局部设置）

- 在使用open( )读取文件前，设置FileSystem对象的setVerifyChecksum的值为false

```
String uri = args[0];  // hdfs://localhost/user/tom/quangle.txt
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(URI.create(uri), conf);
fs.setVerifyChecksum(false);
InputStream in = fs.open(new Path(uri));
```

- 等价地，利用Shell命令可以达到同样效果

```
hadoop fs -copyToLocal [-ignorecrc] [-crc]
        <src> <localdst>
```

```
hadoop fs -get [-ignorecrc] [-crc]
        <src> <localdst>
```

例如：

```
hadoop fs  - get  - ignorecrc  input ~/Desktop
```

HADOOP的数据压缩

- 对于任何大容量的分布式存储系统，文件压缩是必须的。文件压缩带了二个好处：
  - 减少了文件所需的存储空间
  - 加快了文件在网络上或磁盘间的传输速度
- Hadoop支持如下压缩算法

压缩格式	Hadoop压缩编码/解码器
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec

  - 这些编码/解码器实现了CompressionCodec接口
  - LZO是基于GPL许可的，因此不能被包含在Hadoop的发布版本中，必须单独下载。下载地址http://code.google.com/p/hadoop-gpl-compression/

HADOOP的数据压缩

- 不同压缩格式的特点为

压缩格式	命令行工具	算法	文件扩展名	多文件	可分割性
DEFLATE	无	DEFLATE	.deflate	NO	NO
gzip	gzip	DEFLATE	.gz	NO	NO
bzip2	bzip2	bzip2	.bz2	NO	YES
LZO	lzop	LZO	.lzo	NO	NO

  - 可分割性指的是压缩格式是否支持分割。可分割意味着可以定位到流中的任何位置并从这个位置开始读取。因此可分割性对于MapReduce非常重要，因此需要事先将输入分成很多Split，每个Split由一个Mapper处理
  - 压缩是时间和空间的权衡：gzip对于时间/空间的平衡做得最好；bzip2压缩效率比gzip高，但比gzip慢；LZO速度最快，但压缩效果差一些

利用CompressionCodec压缩和解压流

- CompressionCodec有二个方法可以方便地压缩和解压
  - 要压缩将被写到底层输出流的数据，使用createOutputStream(OutputStream out)方法创建CompressionOutputStream对象，这个对象在数据被写到输出流之前将数据压缩
  - 要将来自底层输入流的数据解压，使用createInputStream(InputStream in)方法创建CompressionInputStream对象，这个对象将来自底层输入流的数据解压

```
public class StreamCompressor {
   public static void main(String[] args) throws Exception {
     String codecClassname = args[0]; //用户从命令行输入Codec的类名（完全限定名）
     Class<?> codecClass = Class.forName(codecClassname);
     Configuration conf = new Configuration();
     CompressionCodec codec = (CompressionCodec)
                 ReflectionUtils.newInstance(codecClass, conf);
     CompressionOutputStream out =
                 codec.createOutputStream(System.out);
     IOUtils.copyBytes(System.in, out, 4096, false);
     out.finish();
   }
}
```

根据类名得到Class信息

利用Hadoop的反射工具类ReflectionUtils实例化Codec对象

得到CompressionOutputStream对象，该对象会把写到System.out的数据压缩

```
% echo "Text" | hadoop StreamCompressor org.apache.hadoop.io.compress.GzipCodec | gunzip
Text
```

利用CompressionCodecFactory推断压缩算法

- 通常可以利用文件扩展名来判断压缩算法（如前表所示）
- CompressionCodecFactory提供了getCodec()方法，根据文件扩展名得到CompressionCodec接口的实例

```
CompressionCondec getCodec(Path path);
```

- CompressionCodecFactory通过io.compression.codecs配置属性返回CompressionCodec接口的实例（称为Codec对象）

PropertyName	Type	Default Value	Description
io.compression.codecs	comma-separated Class names	org.apache.hadoop.io.compress.DefaultCodec, org.apache.hadoop.io.compress.GzipCodec, org.apache.hadoop.io.compress.Bzip2Codec	A list of the CompressionCodec classes for compression/ decompression.

  - 每个Codec类都知道各自压缩文件的缺省扩展名
  - 给定path对象，CompressionCodecFactory会遍历每个Codec类找到扩展名对应的Codec对象
  - 意义： MapReduce会自动根据文件扩展名决定是否需要划分Split

利用CompressionCodecFactory推断压缩算法

```
public class FileDecompressor {
   public static void main(String[] args) throws Exception {
      String uri = args[0];
      Configuration conf = new Configuration();
      FileSystem fs = FileSystem.get(URI.create(uri), conf);
      Path inputPath = new Path(uri);
      CompressionCodecFactory factory = new CompressionCodecFactory(conf);
      CompressionCodec codec = factory.getCodec(inputPath);
      if (codec == null) {
         System.err.println("No codec found for " + uri);
         System.exit(1);
      }
      String outputUri =
        CompressionCodecFactory.removeSuffix(uri, codec.getDefaultExtension());
      InputStream in = null;
      OutputStream out = null;
      try {
            in = codec.createInputStream(fs.open(inputPath));
            out = fs.create(new Path(outputUri));
            IOUtils.copyBytes(in, out, conf);
      } finally {
            IOUtils.closeStream(in);
            IOUtils.closeStream(out);
      }
   }
}
```

创建CompressionCodecFactory对象

根据文件的后缀推断出Codec对象

将文件的后缀去掉后，作为解压的输出路径

codec.createInputStream创建解压输入流的对象

fs.open(inputPath) 返回FSDataInputStream对象作为解压的输入流

```
% hadoop FileDecompressor file.gz
```

在MapReduce中使用压缩

- 如果输入数据被压缩，通过使用CompressionCodecFactory自动推断Codec对象，MapReduce程序会自动将输入数据解压
- 如果需要将MapReduce的输出压suo，将mapred.output.compress属性设置为true，将mapred.output.compression.codec属性设置为要使用的Codec类

在MapReduce中使用压缩

```
public class WordCount {
    public static void main(String[] args) throws Exception {
      Configuration conf = new Configuration();
      String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
      if (otherArgs.length != 2) {
            System.err.println("Usage: wordcount <in> <out>");          System.exit(2);
      }
     conf.setBoolean("mapred.output.compress", true);
     conf.setClass("mapred.output.compression.codec", GzipCodec.class,
                                          CompressionCodec.class);
      Job job = new Job(conf, “word count”);
      job.setJarByClass(WordCount.class);
      job.setMapperClass(TokenizerMapper.class);
     job.setReducerClass(IntSumReducer.class);
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);
      FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
     FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
     System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

将mapred.output.compress属性设置为true

将mapred.output.compression.codec属性设置为要使用的Codec类，第三个参数为Codec类所实现的接口类型

Hadoop中的I/O序列化

- 序列化是将内存中的对象转换成字节流，反序列化则是将字节流恢复成内存中的对象
- 序列化有二个目的
  - 进程间通信
  - 数据持久化
- Hadoop利用RPC实现进程间的通信，RPC的序列化机制有以下特点：
  - 紧凑：紧凑的格式可以充分利用带宽，加快传输速度
  - 快速：能减少序列化和反序列化的开销
  - 可扩展：可以随时增加方法调用的新参数
  - 互操作性：：客户端和服务器可以用不同语言实现
- 由于JAVA本身的序列化机制过于复杂，Hadoop自己实现一套更为简洁高效的序列化机制

Writable接口

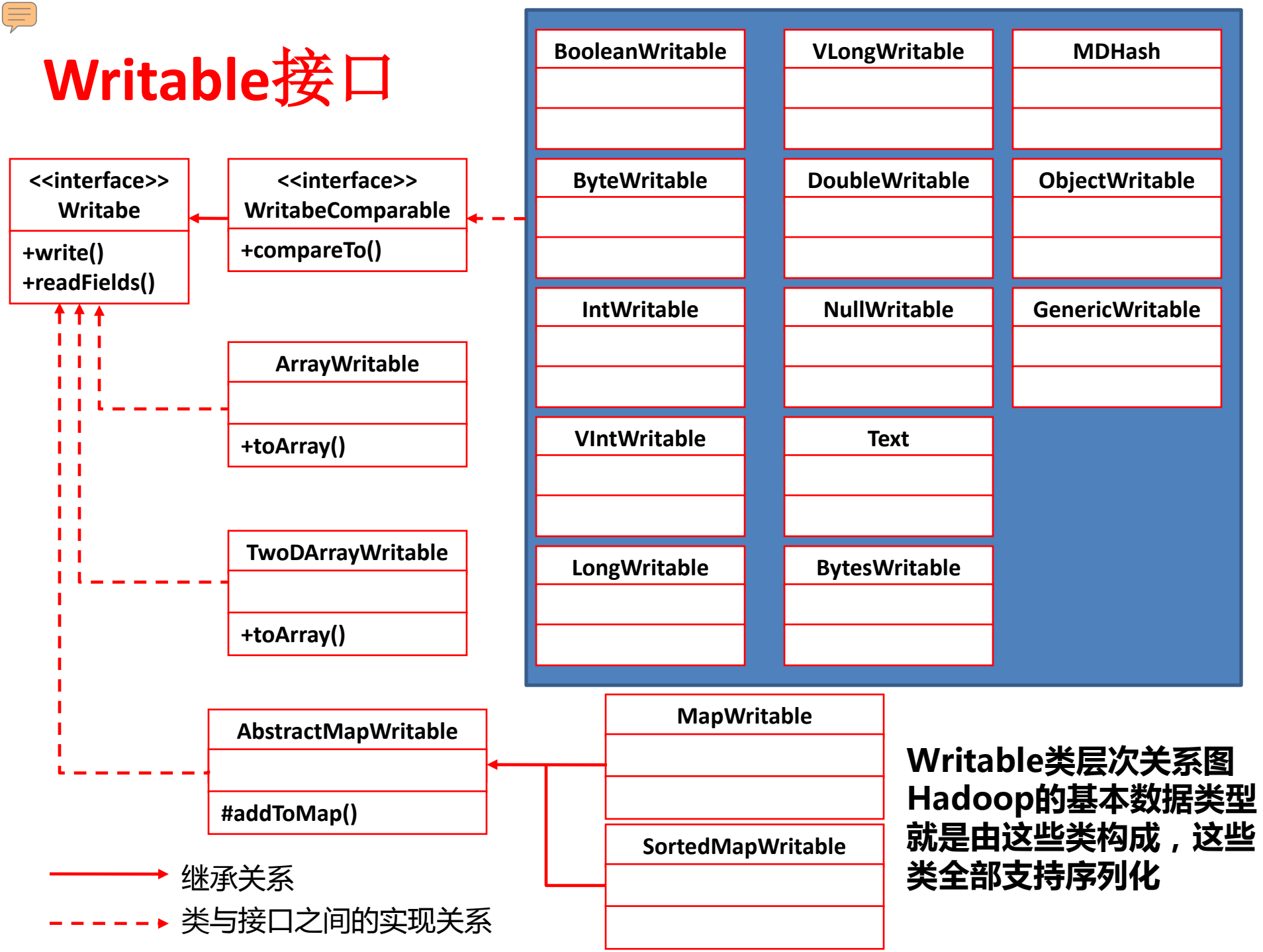
- org.apache.hadoop.io.Writable接口是Hadoop序列化机制的核心，该接口只定义了二个方法

```
package org.apache.hadoop.io;
import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

public interface Writable {
        //将对象序列化到输出流
        void write(DataOutput out) throws IOException;
        //从输入流将对象反序列化
        void readFields(DataInput in) throws IOException;
}
```

  - java.io.DataOutput 为二进制输出流
  - java.io.DataInput为二进制输入流

Writable接口



Writable类层次关系图
Hadoop的基本数据类型就是由这些类构成，这些类全部支持序列化

Writable类使用示例

- IntWritable类是Java数据类型int的包装类

```
IntWritable writable = new IntWritable();
writable.set(163);
或:
IntWritable writable = new IntWritable(163);
```

- 为了观察IntWritable的序列化格式，实现一个helper方法

```
public static byte[] serialize(Writable writable) throws IOException {
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    DataOutputStream dataOut = new DataOutputStream(out);
    writable.write(dataOut); //对象被序列化为字节流写入DataOutputStream
    dataOut.close();
    return out.toByteArray();
}
//测试
byte[] bytes = serialize(writable);
assertThat(bytes.length, is(4));
assertThat(StringUtils.byteToHexString(bytes), is("000000a3"));
```

writable.write → DataOutputStream → ByteArrayOutputStream

Hadoop序列化是采用big-endin格式

Writable类使用示例

- 再来观察反序列化，同样实现一个helper方法

```
public static byte[] deserialize(Writable writable, byte[] bytes)
                                                    throws IOException {
   ByteArrayInputStream in = new ByteArrayInputStream(bytes);
   DataInputStream dataIn = new DataInputStream(in);
   writable.readFields(dataIn); //从DataInputputStream输入的字节流被反序列化为
                                //writable对象
   dataIn.close();
   return bytes;
}
```

- 再创建一个IntWritable对象，但没有值，再调用deserialize方法（将前面产生的bytes数组作为输入）

```
IntWritable newWritable = new IntWritable(); //这时对象没有值
deserialize(newWritable, bytes);
assertThat(newWritable.get(), is(163)); //反序列化后对象值为163
```

WritableComparable和Comparator

- org.apache.hadoop.io.WritableComparable接口是非常重要的接口类，它继承自org.apache.hadoop.Writable和java.lang.Comparable接口

```
package org.apache.hadoop.io;
public interface WritableComparable<T> extends Writable,Comparable<T> {
}
```

- 在Java中，任何需要进行比较的类型都必须实现java.lang.Comparable接口

```
package java.lang;
public interface Comparable<T> {
        public int compareTo(T o);
}
```

- Java还定义了java.util.Comparator接口（比较器接口），用于比较二个对象

```
package java.util;
public interface Comparator<T> {
        int compare(T o1, T o2);
        boolean equals(Object obj);
}
```

WritableComparable和Comparator

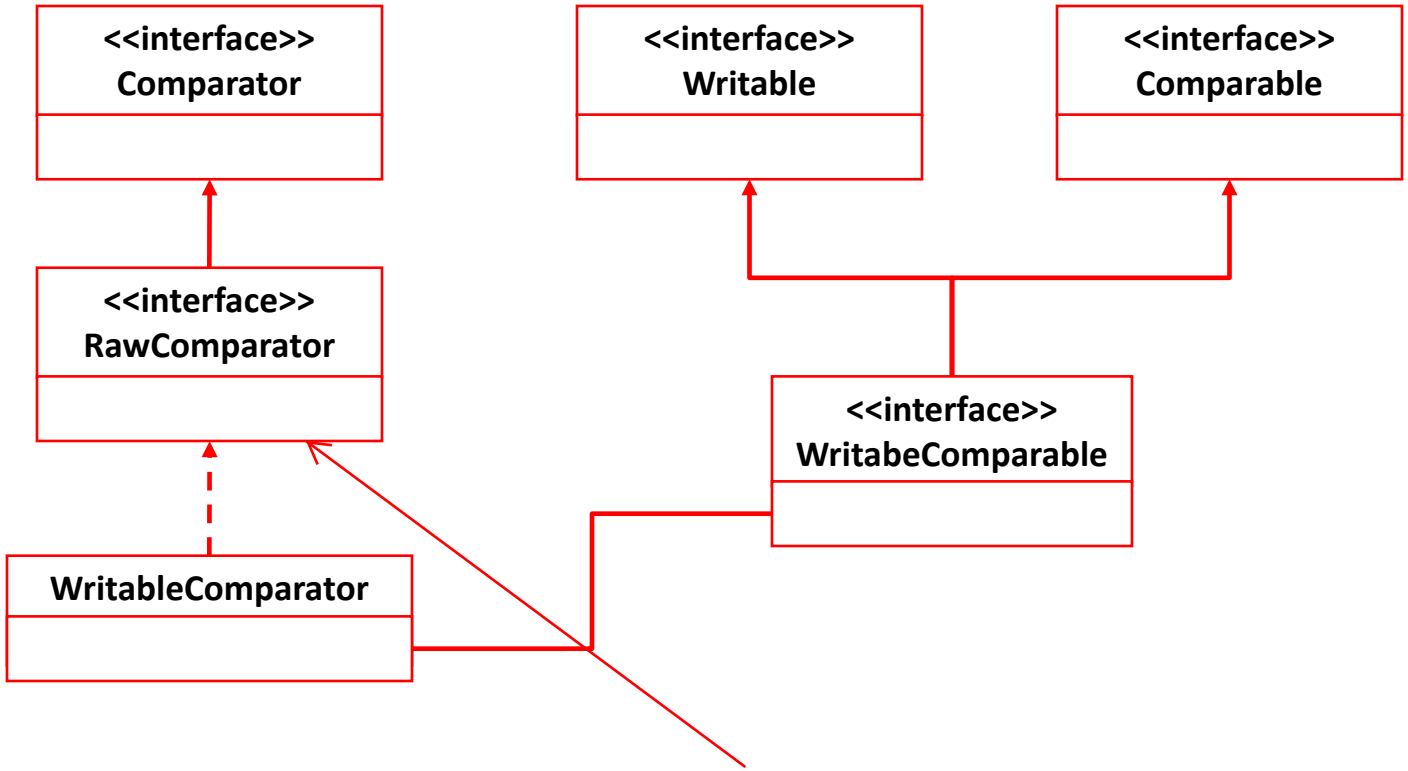
- 在MapReduce执行时，会收集相同key的value形成<key,list<value>>,同时会有排序过程，因此比较（Compariosn）是非常关键的。这些比较都是针对WritableComparable类型进行的
- 为此，Hadoop提供了RawCompator接口，该接口继承了Comparator接口

```
package org.apache.hadoop.io;
import java.util.Comparator;
public interface RawComparator<T> extends Comparator<T> {
   //新添加的接口方法
   public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2);
}
```

  - 注意这个接口定义的compare方法不需要将字节流反序列化为对象再进行比较，而是直接在字节流这一级进行比较，因此效率高得多

- WritableComparator则实现了RawCompator接口，用来比WritableComparable对象

WritableComparable和Comparator



public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2);

继承关系

类与接口之间的实现关系

依赖关系

WritableComparable和Comparator

- WritableComparator主要提供了二个主要
  - 实现了compare方法
  - 实现了静态工厂方法get，返回一个RawComparator实例

```
RawComparator<IntWritable> comparator =
                        WritableComparator.get(IntWritable.class);
IntWritable w1 = new IntWritable(163);
IntWritable w2 = new IntWritable(67);
assertThat(comparator.compare(w1, w2), greaterThan(0)); //调用Java API接口方法

或者：
byte[] b1 = serialize(w1);
byte[] b2 = serialize(w2);
assertThat(comparator.compare(b1, 0, b1.length, b2, 0, b2.length),greaterThan(0));
```

get方法需要被比较对象的类型信息

这是调用JAVA API Comparator接口方法，在对象级进行比较

这是调用Hadoop RawComparator的接口方法，不是在对象这一级比较，而是在字节流级比较

Writable数据类型

- Hadoop利用Writable类型封装了Java很多基本数据类型，如下表所示

Java基本类型	Writable类型	D序列化后的字节数
boolean	BooleanWritable	1
byte	ByteWritable	1
int	IntWritable	4
	VIntWritable	1-5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1-9
double	DoubleWritable	8

Text

- Text是针对对UTF-8编码的字符序列的Writable实现，可以看成是对String的包装
  - 使用标准UTF-8编码
  - 使用变长int来存储字符编码的字节数
  - 最大存储量2G
- 和String的区别
  - String长度定义为包含的字符个数；Text长度定义为UTF-8编码的字节数
  - String内的indexOf方法返回的是char类型的字符索引，比如字符串”1234”，字符3的位置是2；而Text的find方法返回的是字节偏移量
  - String的charAt方法返回的是指定位置的字符，而Text的charAt方法需要指定偏移量
  - Text的toString方法将Text转换成String类型

Text

- 对于ASCII字符串,Text和String没有区别

```
Text t = new Text("hadoop");
assertThat(t.getLength(), is(6));
assertThat(t.getBytes().length, is(6));
assertThat(t.charAt(2), is((int) 'd')); //注意charAt返回的是int值
assertThat("Out of bounds", t.charAt(100), is(-1));

assertThat("Find a substring", t.find("do"), is(2));
assertThat("Finds first 'o'", t.find("o"), is(3));
assertThat("Finds 'o' from position 4 or later", t.find("o", 4), is(4));
assertThat("No match", t.find("pig"), is(-1))
```

为满足基于ASCII，面向字节的系统的需要，Unicode标准中定义了编码格式UTF-8。它是一种使用8位编码单元的变宽的编码格式。
UTF -8编码格式对所有ASCII码点具有透明性。在U+0000到U+007F范围内的Unicode码点，被转换为UTF-8中单一的字节0x00到 0x7F，与ASCII码没有区别。

Text

- 如果对于超过一个字节编码的字符就有区别,考虑以下字符

Unicode code point	U+0041	U+00DF	U+6771	U+10400
Unicode code units	41	C3 9f	e6 9d b1	f0 90 90 80
Java Respesentattion	\u0041	\u00DF	\u6771	\uD801\uDC00

在范围U+10000到U+10FFFF间的码点则使用一对16位编码单元表示,称作代理对(*surrogate pair*)。

Text

- 如果对于超过一个字节编码的字符就有区别,考虑以下字符

```
@Test
public void string() throws UnsupportedEncodingException {
        String s = "\u0041\u00DF\u6771\uD801\uDC00";
        assertThat(s.length(), is(5));
        assertThat(s.getBytes("UTF-8").length, is(10));
        assertThat(s.indexOf("\u0041"), is(0));
        assertThat(s.indexOf("\u00DF"), is(1));
        assertThat(s.indexOf("\u6771"), is(2));
        assertThat(s.indexOf("\uD801\uDC00"), is(3));
        assertThat(s.charAt(0), is('\u0041'));
        assertThat(s.charAt(1), is('\u00DF'));
        assertThat(s.charAt(2), is('\u6771'));
        assertThat(s.charAt(3), is('\uD801'));
        assertThat(s.charAt(4), is('\uDC00'));
}
```

Text

- 如果对于超过一个字节编码的字符就有区别,考虑以下字符

```
@Test
public void text() {
        Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");
        assertThat(t.getLength(), is(10));
        assertThat(t.find("\u0041"), is(0));
        assertThat(t.find("\u00DF"), is(1));
        assertThat(t.find("\u6771"), is(3));
        assertThat(t.find("\uD801\uDC00"), is(6));
        assertThat(t.charAt(0), is(0x0041));
        assertThat(t.charAt(1), is(0x00DF));
        assertThat(t.charAt(3), is(0x6771));
        assertThat(t.charAt(6), is(0x10400));

}
```

Text

- 遍历Text的每个字符
  - 首先将Text转换成java.nio.ByteBuffer
  - 循环调用Text的静态方法bytesToCodePoint()，这个方法会取出buffer中下个字符的码点(作为int返回)，并自动更新buffer的下个字符的位置指针，当到达字符串末尾返回-1

```
public class TextIterator {
   public static void main(String[] args) {
      Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");
      ByteBuffer buf = ByteBuffer.wrap(t.getBytes(), 0, t.getLength());
      int cp;
      while (buf.hasRemaining() && (cp = Text.bytesToCodePoint(buf)) != -1) {
         System.out.println(Integer.toHexString(cp));
      }
   }
}
```

```
% hadoop TextIterator
41
df
6771
10400
```

Text

- 修改Text的内容
  - 除了NullWritable是不可更改外，其他类型的Writable都是可以修改的。你可以通过Text的set方法去修改去修改重用这个实例。

```
@Test
public void testTextMutability() {
        Text text = new Text("hadoop");
        text.set("pig");
        Assert.assertEquals(text.getLength(), 3);
        Assert.assertEquals(text.getBytes().length, 3);
}
```

NullWritable

- NullWritable是一个占位符，它的序列化长度为0，没有数值从流中读出或是写入
- 在任何编程语言或编程框架中，占位符都是很有用的。MapReduce可以利用它将任何键或值设为空值
- 可以理解为null的Writable包装

BytesWritable

- BytesWritable是对二进制字节数组的封装
- 它的序列化格式是：4个字节指明后面跟有多少字节，后面的内容是字节数组本身的内容

```
BytesWritable b = new BytesWritable(new byte[] { 3, 5 });
byte[] bytes = serialize(b);
assertThat(StringUtils.byteToHexString(bytes), is("000000020305"));
```

  - 包含3，5的字节数组，序列化后为4字节的长度000000020305

ObjectWritable

- ObjectWritable是一种多类型的封装：Java原子类型、String、enum、Writable、null、以及这些类型的数组
- 当一个field有多种类型时，它的作用就发挥出来了。但这种封装比较浪费空间，因为在序列化时需要将被封装的类型的名字写入字节流

```
public class TestObjectWritable {
   public static void main(String[] args) throws IOException {
      Text text=new Text("\u0041");
      ObjectWritable objectWritable=new ObjectWritable(text);
       System.out.println(
            StringUtils.byteToHexString(SerializeUtils.serialize(objectWritable)));
   }
}
```

仅仅是保存一个字母，那么看下它序列化后的结果是什么

00196f72672e6170616368652e6861646f6f702e696f2e5465787400196f72672e6170616368652e6861646f6f702e696f2e546578740141

GenericWritable

- 如果被封装的各种类型事先已知，那么可以将这些类型保存在一个类型数组里，在序列化时就不需要将被序列化的类的名字写入字节流，只需要写入被序列化的类型在类型数组中的索引号即可，这样可以大大地节省空间
- 需要继承GenericWritable类，在派生类里指明要被封装的类型

```
class MyWritable extends GenericWritable {
    MyWritable(Writable writable) {
        set(writable);
    }
    public static Class<? extends Writable>[] CLASSES=null;
    static {
        CLASSES=  (Class<? extends Writable>[])new Class[]{   Text.class   };
    }
    @Override
    protected Class<? extends Writable>[] getTypes() {
        return CLASSES;
    }
}
```

调用父类方法设置要被序列化的对象

CLASSES数组保存要被序列化的类型

初始化CLASSES数组，这里只保存了Text类型，即MyWritable要序列化的对象类型是Text，可以放入更多类型

重新实现父类的getTypes方法，返回CLASSES数组

GenericWritable

```
public class TestGenericWritable {
    public static void main(String[] args) throws IOException {
        Text text=new Text("\u0041\u0071");
        MyWritable myWritable=new MyWritable(text);
        System.out.println(StringUtils.byteToHexString(SerializeUtils.serialize(text)));
        System.out.println(StringUtils.byteToHexString(SerializeUtils.serialize(myWritable)));
    }
}
```

得到对象序列化后的字节数组

输出结果是：
024171
00024171

- GenericWritable的序列化只是把类型在type数组里的索引放在了前面，这样就比ObjectWritable节省了很多空间，所以使用GenericWritable

Writable Collections

- 有四种Writable Collection
  - ArrayWritable
  - TwoDArrayWritable
  - MapWritable
  - SortedMapWritable
- ArrayWritable和TwoDArrayWritable是Writable类型的一维和二维数组，数组元素都是Writable类型，元素的类型通过构造函数指定

```
ArrayWritable writable = new ArrayWritable(Text.class);
```

- 它们的子类时，必须使用super()来指定ArrayWritable或TwoDArrayWritable的元素类型

```
public class TextArrayWritable extends ArrayWritable {
    public TextArrayWritable() {
        super(Text.class);
    }
}
```

- toArray方法返回被包装的数组

Writable Collections

- MapWritable和SortedMap分别是java.util.Map<Writable,Writable>和java.util.SortedMap<Writable,Writable>的实现。其键和值都会被序列化
  - key和value可以是org.apache.hadoop.io中定义的Writable类型，也可以是自定义的Writable类型

```
MapWritable src = new MapWritable();
src.put(new IntWritable(1), new Text("cat"));
src.put(new VIntWritable(2), new LongWritable(163));

MapWritable dest = new MapWritable();
WritableUtils.cloneInto(dest, src);
assertThat((Text) dest.get(new IntWritable(1)), is(new Text("cat")));
assertThat((LongWritable) dest.get(new VIntWritable(2)), is(new LongWritable(163)));
```

put方法放入key-value对

cloneInto克隆Writable对象

get方法根据key获取对应value，这里没有指定放入MapWritable对象的key-value类型，所以get返回的类型是Object，需要强制类型转换

自定义Writable类型

- 虽然Hadoop以及定义了丰富的Writable类型，但有些场合需要自定义Writable类型。自定义Writable类型可以完全控制序列化字节序列格式，这样可能会对MapReduce的性能有重要影响

```
public class TextPair implements WritableComparable<TextPair> {
   private Text first;
   private Text second;
   public TextPair() {
      set(new Text(), new Text());
   }
   public TextPair(String first, String second) {
      set(new Text(first), new Text(second));
   }
   public TextPair(Text first, Text second) {
      set(first, second);
   }
   public void set(Text first, Text second) {
      this.first = first;
      this.second = second;
   }
}
```

三个重载的构造函数

私有实例成员的setter函数

自定义Writable类型

```
public class TextPair implements WritableComparable<TextPair> {
    public Text getFirst() {  return first; }
    public Text getSecond() {  return second; }
    @Override
    public void write(DataOutput out) throws IOException {
        first.write(out);
        second.write(out);
    }
    @Override
    public void readFields(DataInput in) throws IOException {
        first.readFields(in);
        second.readFields(in);
    }
    @Override
    public String toString() { return first + "\t" + second; }
    @Override
    public int compareTo(TextPair tp) {
        int cmp = first.compareTo(tp.first);
        if (cmp != 0) {
            return cmp;
        }
       return second.compareTo(tp.second);
    }
}
```

私有实例成员的getter函数

实现Writable接口的write方法，依次将first、second成员序列化

实现Writable接口的readFields方法，依次将first、second成员反序列化

实现Comparable接口的compareTo方法，比较俄二个TextPair对象的大小。这是对象级别的比较

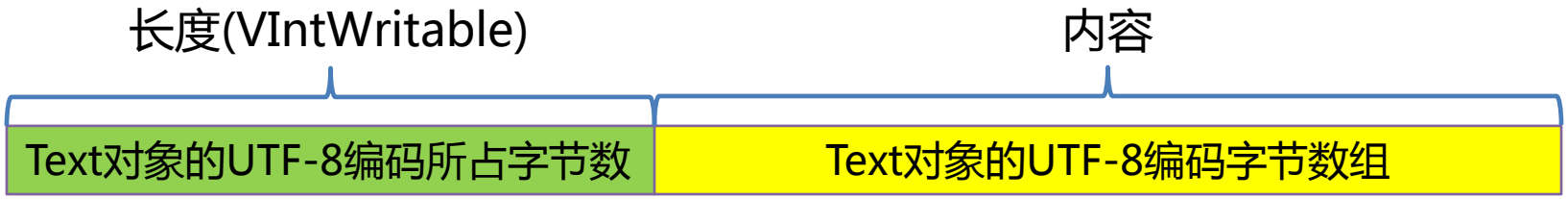
为自定义Writable类型实现比较器

- TextPair实现了Comparable接口的方法以比较二个TextPair对象的大小
- 当TextPair被用作MapReduce中的键时，需要将数据反序列化为对象，再调用compareTo方法进行比较，是否可以在序列化后的字节流级别比较二个对象呢？这样可以提高效率
  - TextPair对象是由二个Text对象连接而成
  - Text对象序列化后的二进制表示是一个长度可变的整数

    长度（用VIntWritable表示的Text的字节数）+Text内容（UTF-8字节数组）
  - 因此我们可以读取first对象的字节长度及内容（ UTF-8字节数组），并传递给Text对象的RawComparator比较器；
  - 通过计算偏移量同样读取second对象的字节长度及内容（ UTF-8字节），并传递给Text对象的RawComparator比较器。这样可以实现字节流级别的TextPair对象的比较
  - 关键是如何读取用VIntWritable表示的Text的字节数

为自定义Writable类型实现比较器

- 读取用VIntWritable表示的Text的字节数



- 利用WritableUtils. decodeVIntSize方法得到可变长VIntWritable/ VLongWritable变量的字节数

```
public static int decodeVIntSize(byte value);
        参数value： VIntWritable/ VLongWritable变量的第一个字节
```

- 利用WritableComparator的readVInt方法可以得到可变长VIntWritable/ VLongWritable变量的值

```
public static int decodeVIntSize(byte value);
        参数value： VIntWritable/ VLongWritable变量的第一个字节
```

- 因此假设Text对象序列化后得到的字节数组为byte[]，第一个元素的下标为s1，则

```
WritableUtils.decodeVIntSize(b1[s1]) 返回VIntWritable变量的字节数
writableComparator.readVInt(b1, s1)返回VIntWritable变量的值（即内容所占字节数）
二者之和为Text对象序列化后的总字节数
```

为自定义Writable类型实现比较器

```
public class TextPair implements WritableComparable<TextPair> {

   public static class Comparator extends WritableComparator {

      private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator();
      public Comparator() {
         super(TextPair.class);
      }
      @Override
      public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
         try {
            int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt(b1, s1);
            int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt(b2, s2);
            int cmp = TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2, firstL2);
            if (cmp != 0) {
               return cmp;
            }
            return TEXT_COMPARATOR.compare(b1, s1 + firstL1, l1 - firstL1,b2, s2 + firstL2, l2 - firstL2);
         } catch (IOException e) {
            throw new IllegalArgumentException(e);
         }
      }
   }

   static {  WritableComparator.define(TextPair.class, new Comparator());  }
}
```

Comparator作为TextPair的嵌套类实现

在构造函数里调用父类的构造函数指定要比较的对象类型

创建Text的比较器对象，Text的比较器也是作为Text的嵌套类实现

重实现compare方法

firstL1：第一个TextPair对象的first成员的字节数

比较第一个TextPair对象的first成员与第二个TextPair对象的first成员的大小

比较第一个TextPair对象的second成员与第二个TextPair对象的second成员的大小
注意起始位置和长度的计算

注册TextPair类型的比较器

序列化框架

- Hadoop支持替换不同的序列化框架，可以将任何其它的序列化框架插入
- 一个序列化框架用Serialization实现（在org.apache.hadoop.io.serializer包中）来表示
  - Serialization 是一个接口：interface Serialization<T>
  - WritableSerialization,就是对Writable类型的Serialization实现
- 将io.serializations属性设置为一个用逗号分隔的Serialization实现类列表，即可注册序列化框架
  - 它的默认值是org.apache.hadoop.io.serializer.WritableSerialization
- Hadoop有一个JavaSerialization类，它使用的是Java Object Serialization，即JAVA本身的序列化机制，但不如Writable高效

基于文件的数据结构

- Hadoop的HDFS和MapReduce子框架主要是针对大数据文件来设计的，在小文件的处理上不但效率低下，而且十分消耗内存资源(每一个小文件占 用一个Block,每一个block的元数据都存储在namenode的内存里)。解决办法通常是选择一个容器，将这些小文件组织起来统一存储。HDFS 提供了两种类型的容器，分别是SequenceFile和MapFile。
- SequenceFile文件是Hadoop用来存储二进制形式的key-value对而设计的一种平面文件(Flat File)
- 在该文件的基础之上提出了一些HDFS中小文件存储的解决方案，基本思路就是将小文件进行合并成一个大文件：比如每个小文件的文件名作为key，文件内容作为value写入SequenceFile文件
- 在SequenceFile文件中，每一个key-value被看做是一条记录(Record)，因此基于Record的压缩策略，SequenceFile文件可支持三种压缩类型(SequenceFile.CompressionType):
  - NONE: 对Record不进行压缩;
  - RECORD: 仅压缩每一个Record中的value值;
  - BLOCK: 将一个block中的所有Record压缩在一起;
- 基于这三种压缩类型，Hadoop提供了对应的三种类型的Writer:
  - SequenceFile.Writer 写入时不压缩任何的key-value对(Record);
  - equenceFile.RecordCompressWriter写入时只压缩key-value对(Record)中的value;
  - SequenceFile.BlockCompressWriter 写入时将一批key-value对(Record)压缩成一个Block;

SequenceFile的写入

- 要创建一个SequenceFile，调用SequenceFile的静态方法createWriter()返回一个SequenceFile.Writer实例对象，这个方法有很多重载版本，例如

```
org.apache.hadoop.io.SequenceFile.Writer createWriter(FileSystem fs,
                    Configuration conf, Path name, Class keyClass, Class valClass)
```

- 存贮在SequenceFile中的key和value可以是任何可以被序列化和反序列化的类型（不是必须为Writable类型）
- 一旦获得了SequenceFile.Writer实例对象，可以调用其append方法写入key-value对

```
public class SequenceFileWriteDemo {
   private static final String[] DATA = {
      "One, two, buckle my shoe",
      "Three, four, shut the door",
      "Five, six, pick up sticks",
      "Seven, eight, lay them straight",
      "Nine, ten, a big fat hen"
   };


}
```

SequenceFile的写入

```
public class SequenceFileWriteDemo {
    public static void main(String[] args) throws IOException {
        String uri = args[0];
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), conf);
        Path path = new Path(uri);
        IntWritable key = new IntWritable();
        Text value = new Text();
        SequenceFile.Writer writer = null;
        try {
            writer = SequenceFile.createWriter(fs, conf, path, key.getClass(), value.getClass());
            for (int i = 0; i < 100; i++) {
                key.set(100 - i);
                value.set(DATA[i % DATA.length]);
                System.out.printf("[%s]\t%s\t%s\n", writer.getLength(), key, value);
                writer.append(key, value);
            }
        } finally {
            IOUtils.closeStream(writer);
        }
    }
}
```

获取当前写入的位置（以字节为单位）

SequenceFile的写入

```
% hadoop SequenceFileWriteDemo numbers.seq
[128] 100 One, two, buckle my shoe
[173] 99 Three, four, shut the door
[220] 98 Five, six, pick up sticks
[264] 97 Seven, eight, lay them straight
[314] 96 Nine, ten, a big fat hen
[359] 95 One, two, buckle my shoe
[404] 94 Three, four, shut the door
[451] 93 Five, six, pick up sticks
[495] 92 Seven, eight, lay them straight
[545] 91 Nine, ten, a big fat hen
...
[1976] 60 One, two, buckle my shoe
[2021] 59 Three, four, shut the door
[2088] 58 Five, six, pick up sticks
[2132] 57 Seven, eight, lay them straight
[2182] 56 Nine, ten, a big fat hen
...
[4557] 5 One, two, buckle my shoe
[4602] 4 Three, four, shut the door
[4649] 3 Five, six, pick up sticks
[4693] 2 Seven, eight, lay them straight
[4743] 1 Nine, ten, a big fat hen
```

SequenceFile的读取

- 读取SequenceFile首先需要创建SequenceFile.Reader对象的实例
- 迭代地顺序读取每个Record，如何迭代取决于key-value是如何被序列化的
  - 如果key-value都是Writable类型，则调用SequenceFile.Reader的next()方法。当读到文件末尾时该方法返回false

```
public boolean next(Writable key, Writable val)
```

  - 如果使用其他非Writable的序列化框架，就使用如下方法

```
public Object next(Object key) throws IOException
public Object getCurrentValue(Object val) throws IOException
```

  - 如果next方法返回非null，则意味着一个key-value对已经从流里读取，利用getCurrentValue方法可以获取value

SequenceFile的读取

```
public class SequenceFileReadDemo {
   public static void main(String[] args) throws IOException {
      String uri = args[0]; //args[0]为SequenceFile的路径
      Configuration conf = new Configuration();
      FileSystem fs = FileSystem.get(URI.create(uri), conf);
      Path path = new Path(uri);
      SequenceFile.Reader reader = null;
      try {
         reader = new SequenceFile.Reader(fs, path, conf); //创建Reader对象
         Writable key = (Writable)ReflectionUtils.newInstance(reader.getKeyClass(), conf);
         Writable value = (Writable)ReflectionUtils.newInstance(reader.getValueClass(), conf);
         long position = reader.getPosition(); //获取当前读取的字节位置
         while (reader.next(key, value)) {
            String syncSeen = reader.syncSeen() ? "*" : ""; //是否遇到同步点
            System.out.printf("[%s%s]\t%s\t%s\n",
                                    position, syncSeen, key, value);
            position = reader.getPosition();
         }
      } finally {
         IOUtils.closeStream(reader);
      }
   }
}
```

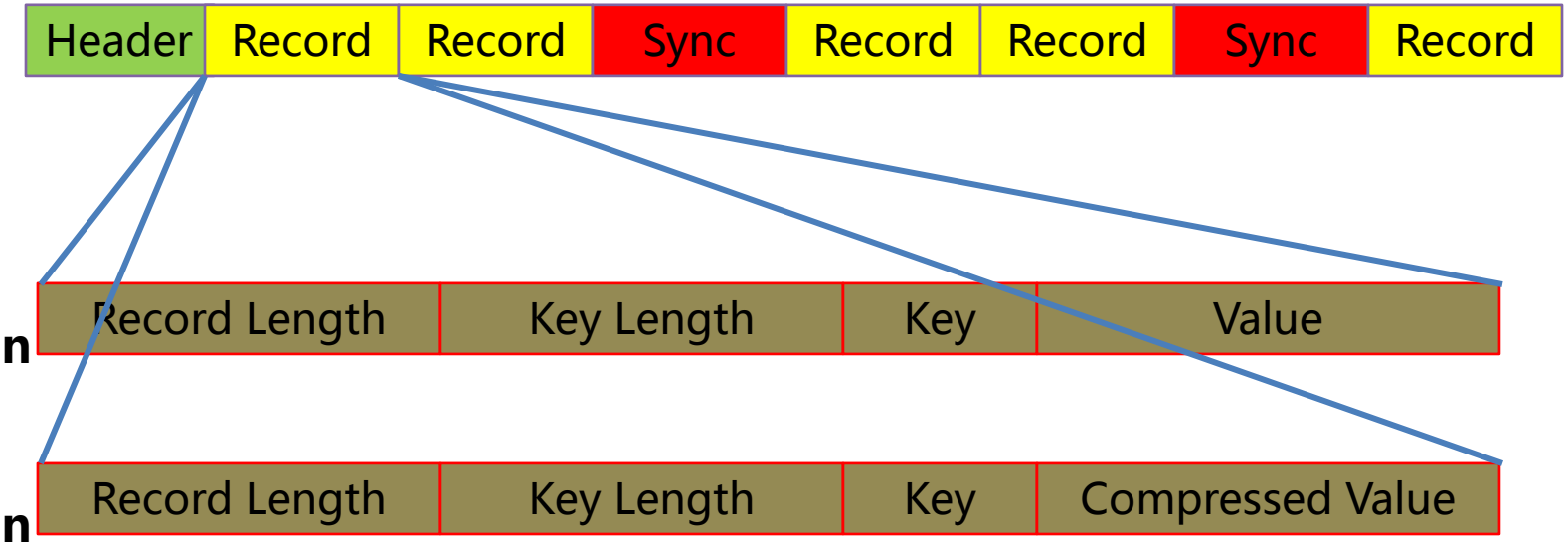
注意通过reader的getkeyClass和getValueClass方法自动发现key和value的类型，再利用反射实例化key和value对象。所以这个程序可以读取所有Writable类型key-value的顺序文件

SequenceFile的读取

```
% hadoop SequenceFileReadDemo numbers.seq
[128] 100 One, two, buckle my shoe
[173] 99 Three, four, shut the door
[220] 98 Five, six, pick up sticks
[264] 97 Seven, eight, lay them straight
[314] 96 Nine, ten, a big fat hen
[359] 95 One, two, buckle my shoe
[404] 94 Three, four, shut the door
[451] 93 Five, six, pick up sticks
[495] 92 Seven, eight, lay them straight
[545] 91 Nine, ten, a big fat hen
[590] 90 One, two, buckle my shoe
...
[1976] 60 One, two, buckle my shoe
[2021*] 59 Three, four, shut the door
[2088] 58 Five, six, pick up sticks
[2132] 57 Seven, eight, lay them straight
[2182] 56 Nine, ten, a big fat hen
...
[4557] 5 One, two, buckle my shoe
[4602] 4 Three, four, shut the door
[4649] 3 Five, six, pick up sticks
[4693] 2 Seven, eight, lay them straight
[4743] 1 Nine, ten, a big fat hen
```

SequenceFile的结构

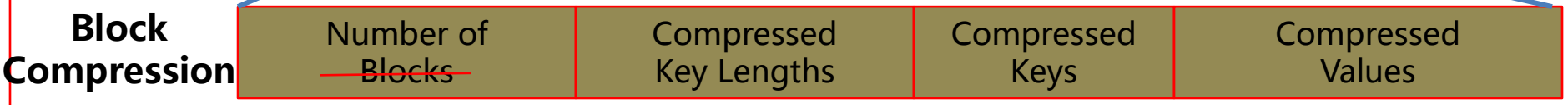
- SequenceFile的结构如下图所示



- 顺序文件由文件头(Header)和随后的多条记录(Record)组成
- 顺序文件的头三个字节为SEQ，紧随其后的一个字节表示版本号
- 文件头包括key-value的类型，压缩细节，用户定义的元数据，和同步标识的内容（随机生成）
- 同步标识的作用是当Reader丢失读取位置信息（例如调用seek方法）时能够再次与记录的边界同步。同步标识是由Sequence.Writer（sync方法）每隔几个记录插入的，同步标识始终和记录边界对齐
- 记录的内部结构与是否压缩有

SequenceFile的结构

- SequenceFile的结构如下图所示



- 块压缩一次多多条记录压缩
- 可以不断地向数据块中压缩记录，直到块的字节数达到io.seqfile.compress.blocksize属性设置的字节数，该属性默认为1M
- 每个块的开始处插入同步标识

通过命令行显示SequenceFile的内容

- Hadoop fs命令有一个-text选项，可以以文本形式显示顺序文件的内容

```
% hadoop fs -text numbers.seq | head
100 One, two, buckle my shoe
99 Three, four, shut the door
98 Five, six, pick up sticks
97 Seven, eight, lay them straight
96 Nine, ten, a big fat hen
95 One, two, buckle my shoe
94 Three, four, shut the door
93 Five, six, pick up sticks
92 Seven, eight, lay them straight
91 Nine, ten, a big fat hen
```