# 5.1 数据完整性

## 5.1.1 HDFS的数据完整性

1.检测数据是否损坏的常见措施是，在**数据第一次引入系统时计算校验和**(checksum)并在数据通过一个不可靠的通道进行**传输时再次计算校验和**，这样就能发现**数据是否损坏**。如果**计算所得的新校验和与原来的校验和不匹配**，我们就认为**数据已损坏**。但该技术并**不能修复数据**—它只能检测出数据错误。(这正是不使用低端硬件的原因。具体说来，一定要使用ECC内存。)注意，**校验和也是可能损坏的**，不只是数据，**但由于校验和比数据小得多，所以损坏的可能性非常小**。

2.Hadoop使用CRC-32（32位循环冗余校验），HDFS用于校验和计算的则是一个更有效的变体CRC-32C。

3.HDFS会在写入的所有数据计算校验和，并在读取数据时验证校验和。

4.**DataNode负责在收到数据后存储该数据及其校验和之前对数据进行验证**。它在收到客户端的**数据或复制其他DataNode的数据时执行这个操作**。**正在写数据的客户端将数据及其校验和发送到由一系列DataNode组成的管线**，**管线中最后一个DataNode负责验证校验和**。如果**DataNode检测到错误，客户端便会收到一个IOException异常的一个子类，对于该异常应以应用程序的方式来处理，比如重试这个操作**。

5.客户端从DataNode读取数据时，也会验证校验和，将它们与DataNode中存储的校验和进行比较。每个DataNode均持久保存有一个用户验证的校验和日志，所以它知道每个数据块的最后一次验证时间。客户端成功验证一个数据块后，会告诉这个DataNode，DataNode由此更新日志。保存这些统计信息对于检测损坏的磁盘很有价值。

6.不只是客户端在读取数据块时会验证校验和，每个DataNode也会在一个后台进程中运行一个DataBlockScanner，从而定期验证存储在这个DataNode上的所有数据块。这项措施是解决物理存储媒体上位损坏的有力措施。

7.由于**HDFS存储着每个数据块的复本**(replica)，因此它**可以通过数据复本来修复损坏的数据块**，**进而得到一个新的、完好无损的复本**。

8.基本思路是，**客户端在读取数据块时，如果检测到错误，首先向namenode报告已损坏的数据块及其正在尝试读操作的这个 datanode**，再抛出 **ChecksumException 异常**。**namenode 将这个数据块复本标记为已损坏**，这样它不再将**客户端处理请求直接发送到这个节点**，**或尝试将这个复本复制到另一个datanode**。之后，它安排这个数据块的一个复本复制到另一个datanode，如此一来，**数据块的复本因子(replication factor)又回到期望水平。此后，已损坏的数据块复本便被删除**。

## 5.1.2 LocalFileSystem

1.Hadoop的 **LocalFileSystem**执行**客户端的校验和验证**。这意味着在你写入一个名为filename的文件时，文件系统客户端会明确在包含每个文件块校验和的同一个目录内新建一个.filename.crc隐藏文件。文件块的大小由属性 file.bytes-per-checksum控制，默认为**512个字节**。文件块的大小作为**元数据**存储在.crc文件中，所以即使文件块大小的设置已经发生变化，仍然可以正确读回文件。在读取文件时需要验证校验和，并且如果检测到错误，LocalFileSystem还会抛出一个ChecksumException异常。

2.校验和的计算代价是相当低的，一般只增加少许额外的读/写文件时间。大多数应用来讲，付出这样的额外开销以保证数据完整性是可以接受的。此外，我们可以禁用校验和计算，特别是在底层文件系统本身就支持校验和的时候。

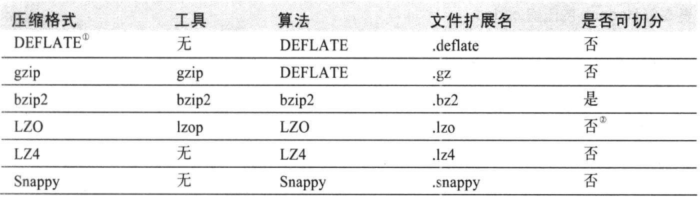
## 5.1.3 ChescksumFileSystem

1.LocalFileSystem通过ChecksumFileSystem来完成自己的任务，有了这个类，向其他文件系统（无校验和系统）加入校验和就非常简单，因为ChecksumFileSystem类继承自FileSystem类。

# 5.2 压缩

1.文件压缩有两大好处：**减少存储文件所需要的磁盘空间**，并加速数据在网络和磁盘上的传输。这两大好处在处理大量数据时相当重要，所以我们值得考虑在Hadoop中文件压缩的用法。

2.压缩格式总结：

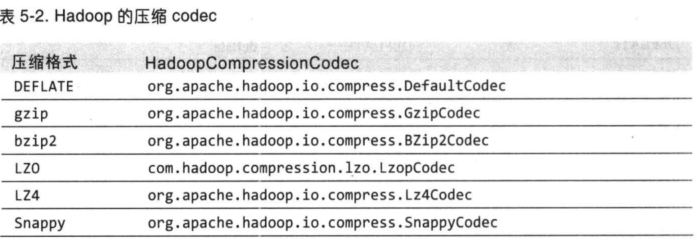


3.所有的压缩算法都需要权衡空间/时间：压缩和解压缩速度更快，其代价通常只能节省少量的空间。

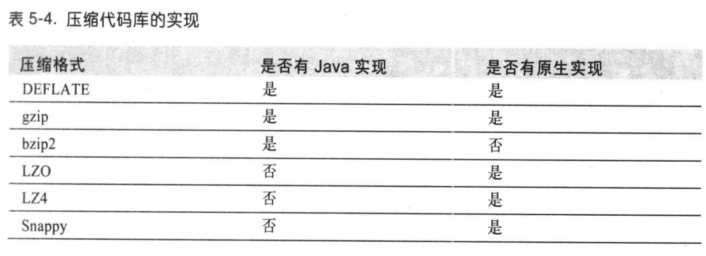
4.gzip是一个通用的压缩工具，在空间/时间性能的权衡中，居于其他两个压缩方法之间。bzip2的压缩能力强于gzip，但压缩速度更慢一点。尽管bzip2的解压速度比压缩速度快，但仍比其他压缩格式要慢一些。另一方面LZO、LZ4和Snappy均优化压缩速度，其速度比gzip快一个数量级，但压缩效率稍逊一筹。Snappy和LZ4的解压缩速度比LZO高的多。

## 5.2.1 codec

1.Codec是**压缩–解压缩算法的一种实现**。在 Hadoop中，一个对CompressionCodec接口的实现代表一个codec。例如，GzipCodec包装了gzip的压缩和解压缩算法。表5-2列举了Hadoop实现的codec。



2.为了提高性能，最好使用原生类库来实现压缩和解压缩。下表说明了每种压缩格式是否有Java实现和原生类库实现。



3.如果使用的是原生代码库并且需要在应用中执行大量压缩和解压缩操作，可以考虑使用CodecPool，它支持反复使用压缩和解压缩，以分摊这些对象的开销。

## 5.2.2 压缩和输入分片(要看)

1.现在想象一下，**文件是经过 gzip压缩的**，且**压缩后文件大小为1GB**。与以前一样，HDFS将这个文件保存为**8个数据块**。但是，将**每个数据块单独作为一个输入分片是无法实现工作的**，因为**无法实现从 gzip压缩数据流的任意位置读取数据**，所以让 map任务独立于其他任务进行数据读取是行不通的。**gzip 格式使用DEFLATE 算法来存储压缩后的数据，而DEFLATE 算法将数据存储在一系列连续的压缩块中**。问题在于**每个块的起始位置并没有以任何形式标记**，**所以读取时无法从数据流的任意当前位置前进到下一块的起始位置读取下一个数据块**，从而实现与整个数据流的同步。由于上述原因，gzip并不支持文件切分。

2.在这种情况下，**MapReduce会采用正确的做法，它不会尝试切分gzip 压缩文件**，因为它知道输入是 **gzip压缩文件**(通过文件扩展名看出)且 gzip不支持切分。这是可行的，**但牺牲了数据的本地性：一个map任务处理8个HDFS块**，而其中大多数块并没有存储在执行该map任务的节点上。而且，**map任务数越少，作业的粒度就较大，因而运行的时间可能会更长**。

3.另一方面bzip2文件提供不同数据块之间的同步标识，因而它支持切分。

4.Hadoop应该使用那种压缩格式？

Hadoop应用处理的数据集非常大，因此需要借助于压缩。使用哪种压缩格式与待处理的文件的大小、格式和所使用的工具相关。下面是按照效率从高到低排列的。

* 使用容器文件格式，例如顺序文件、Avro数据文件、ORCFiles或者Parquet文件，所有这些文件格式同时支持压缩和切分。通常最好与一个快速压缩工具联合使用，比如LZO，LZ4，snappy
* 使用支持切分的压缩格式，比如bzip2，或者通过索引实现切分的压缩格式，例如LZO。
* 在应用中将文件切分成块，并使用任意一种压缩格式为每个数据块建立压缩文件（不论是否支持切分）。这样情况下，需要合理选择数据块的大小，**以确保压缩后数据块的大小近似于HDFS块的大小**。

对于大文件来说，不要使用**不支持切分整个文件的压缩格式**，因为会失去数据的本地特性，进而造成MapReduce应用效率低下。

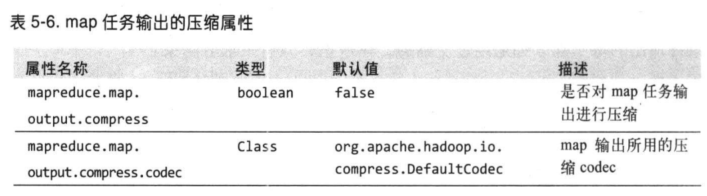
## 5.2.3 在MapReduce中使用压缩

1.前面讲到通过**CompressionCodecFactory来推断CompressionCodec时指出**，如果输入文件是压缩的，**那么在根据文件扩展名推断出相应的 codec 后，MapReduce会在读取文件时自动解压缩文件**。

2.要想压缩MapReduce作业的输出，应在作业配置过程中将mapreduce.output.fileoutputformat. compress属性设为true ,将mapre-duce.output.fileoutputformat.compress.codec属性设置为打算使用的压缩codec 的类名。另一种方案是在 FileoutputFormat 中使用更便捷的方法设置这些属性。

3.如果为输出生成顺序文件(sequence file)，可以设置mapreduce.output.fileoutputformat.compress.type属性来控制限制使用压缩格式。默认值是 RECORD，即针对每条记录进行压缩。如果将其改为 BLOCK，将针对一组记录进行压缩，这是推荐的压缩策略，因为它的压缩效率更高(参见5.4.1节)。

4.尽管**MapReduce应用读/写的是未经压缩的数据**，但如果对**map阶段的中间输入进行压缩**，也可以获得不少好处。**由于map任务的输出需要写到磁盘并通过网络传输到reducer节点**，**所以通过使用LZO、LZ4或者Snappy这样的快速压缩方式，是可以获得性能提升的，因为需要传输的数据减少了**。启用map任务输出压缩和设置压缩格式的配置属性如表所示。



# 5.3 序列化(重点)

1.**序列化是指将结构化对象转化为字节流以便在网络上传输或写道磁盘进行永久存储的过程**。**反序列化是指将字节流传回结构化对象的逆过程**。

2.序列化用于**分布式数据处理**的两大领域：**进程间通信和永久存储**。

3.在Hadoop中，系统中多个节点上进程间的通信是通过“**远程过程调用**”(RPC，remove procedure call)实现的。RPC协议将消息序列化成二进制后发送到远程节点，远程节点接着将二进制流反序列化为原始消息。通常情况下，RPC序列化格式如下：

* 紧凑：紧凑格式能充分利用网络带宽(数据中心最稀缺的资源)
* 快速：进程间通信形成了分布式系统的骨架，所以需要尽量减少序列化和反序列化的性能开销，这是最基本的
* 可扩展：为了满足新的需求，协议不断变化。所以在控制客户端和服务器的过程中，需要直接引进相应的协议。例如，需要能够在方法调用的过程中增添新的参数，并且新的服务器需要能够接受来自老客户端的老格式的消息(无新增的参数)
* 支持互操作：对于某些系统来说，希望能支持以不同语言写的客户端与服务器交互，所以需要设计一种特定的格式来满足这一需求。

4.表面来看，序列化框架对选择用于数据持久存储的数据格式应该会有不同的要求。毕竟，**RPC的存活时间不到1秒钟，持久存储的数据却可能在写到磁盘若干年后才会被读取**。

## 5.3.1 Writable接口&类

1.Writeable接口定义了两个方法：**一个将状态写入DataOutput二进制流**，**另一个从DataInput二进制流读取状态**。

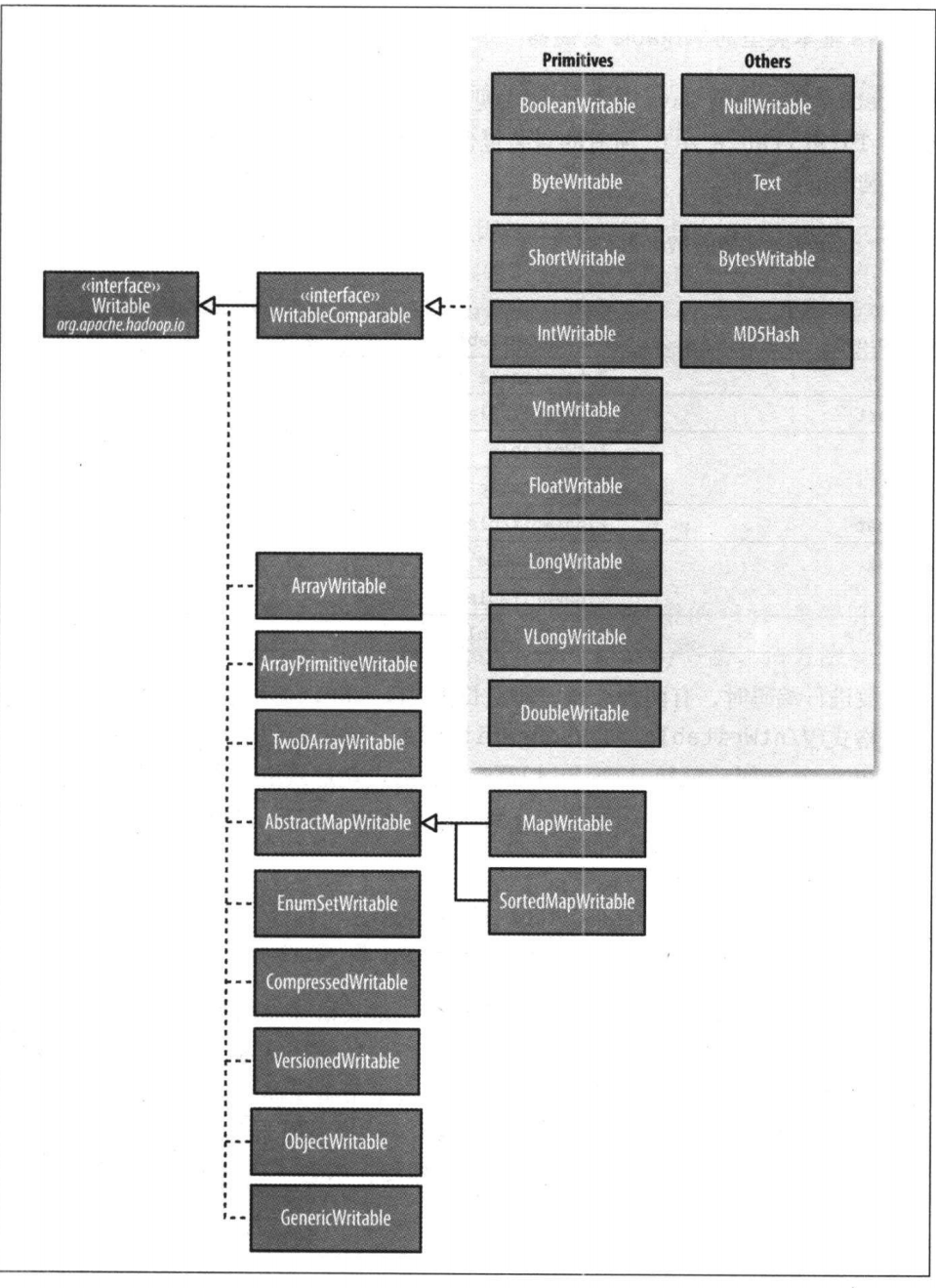
2.Hadoop使用IntWritable来封装Java int类型。我们可以创建一个对象并使用set()方法设定它的值。

IntWritable writable = new IntWritable();writable.set(163);

3.也可以通过一个整数值作为输入参数的构造函数来新建一个对象：

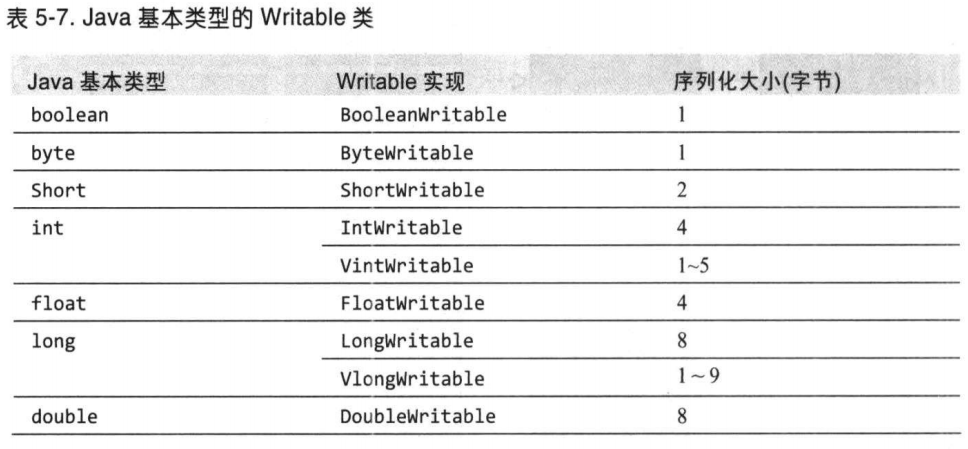
IntWritable writable = new IntWritable(163);

4.Hadoop自带的org.apache.hadoop.io包中有广泛的Writable类可供选择。其层次结构如下所示。



5.Java基本类型的Writable封装器：

Writable类对所有Java基本类型提供封装，char类型除外(可以存储在IntWritable中)。所有的封装包含get()和set()两个方法用于读取和存储封装的值。



对整数进行编码时，有两种选择，一个是**定长格式**(IntWritable和LongWritable)和**变长格式**(VIntWritable和VLongWritable)。需要编码的数值如果相当小(在-127~127之间，包括端点值)，变长格式就是只用一个字节进行编码；否则，使用第一个字节来表示数值的正负和后跟多少个字节。

**定长格式编码很适合数值在整个值域区间中分布非常均匀的情况**，例如哈希函数。然而，大多数数值变量的分布都不均匀，一般而言变长格式会更省空间。**变长编码的另一个优点是可以在VIntWritable和VLongWritable转换，因为它们的编码实际上是一致的**。选择变长格式之后，便有增长空间，不必一开始就用8字节的long表示。

6.Text类型：是针对UTF-8序列的Writable类。一般可以认为它是java.lang.String的Writable等价。

Text类使用整型(通过变长编码的方式)来存储字符串编码中所需的字节数，因此该最大值为2GB。另外，**Text使用标准UTF-8编码**，这使得能够更简便地于其他理解UTF-8编码的工具进行互相操作。

索引：**由于着重使用标准的UTF-8编码**，因此**Text类和Java string 类之间存在一定的差别**。对Text类的索引**是根据编码后字节序列中的位置实现的**，**并非字符串中的 Unicode字符**，也不是Java char的编码单元(如String)。对于ASCII字符串，这三个索引位置的概念是一致的。

Text与String的差异：

这个测试证实**String 的长度是其所含char编码单元的个数**(5，由该字符串的前三个字符和最后的一个代理对组成)，但 **Text对象的长度却是其UTF-8编码的字节数**(10=1+2+3+4)。相似的，String 类的 indexOf()方法返回char编码单元中的索引位置，Text类的find()方法则返回字节偏移量。

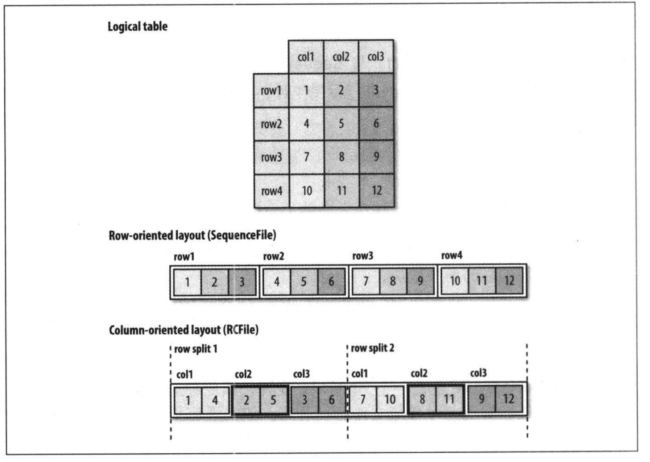
对String重新排序：Text类并不像java.lang.String那样存在丰富的字符串操作API。所以，大多数情况下需要将Text转化为String对象。这一方法通常调用toString()。

7.NullWritable：

Nullwritable是 **writable的特殊类型**，它的**序列化长度为0**。它**并不从数据流中读取数据**，也不写入数据。它充当占位符；例如，在 MapReduce 中，**如果不需要使用键或值的序列化地址**，就可以将键或值声明为 Nullwritable，这样可以高效存储常量空值。如果希望存储一系列数值，与键-值对相对，Nullwritable也可以用作在SequenceFile 中的键。它是一个不可变的单实例类型，通过调用Nullwritable.get()方法可以获取这个实例。

## 5.3.2 面向列的格式

1.顺序文件、map文件和Avro数据文件都是面向行的格式，意味着每一行的值在文件中是连续存储的。在面向列的格式中，文件中的行被分割成行的分片，然后每个分片以面向列的形式存储：首先存储每行第1列的值，然后是每行第2列的值。如下图所示。



2.**面向列的存储布局可以使一个查询跳过那些不必访问的列**。让我们考虑一个只需要处理图5-4中表的第2列的查询。在像顺序文件这样面向行的存储中，即使是只需要读取第二列，**整个数据行**(存储在顺序文件的一条记录中)将被加载进内存。虽说“延迟反序列化”(lazy deserialization)策略通过只反序列化那些被访问的列字段能节省一些处理开销，但这仍然不能避免从磁盘上读入一个数据行所有字节而付出的开销。

3.如果使用**面向列的存储**，只需要把文件中第2列所对应的那部分读入内存。一般来说，面向列的存储格式对于那些只访问表中一部分列的查询比较有效。相反，面向行的存储格式适合同时处理一行中很多列的情况。

4.由于必须在**内存中缓存行的分片**、**而不是单独的一行**，因此**面向列的存储格式需要更多的内存用于读写**。并且，当出现写操作时(通过flush或sync操作)，这种缓存通常不太可能被控制，因此，**面向列的格式不适合流的写操作**，这是因为，如果writer处理失败的话，**当前的文件无法恢复**。另一方面，对于面向行的存储格式，如顺序文件和 Avro 数据文件，可以一直读取到writer 失败后的最后的同步点。由于这个原因，Flume(详见第14章)使用了面向行的存储格式。

5.Hadoop中第一个面向列的文件格式是Hive的RCFlie，它已经被Hive的ORCFile(Optimized Record Columnar File)及Parquet取代。