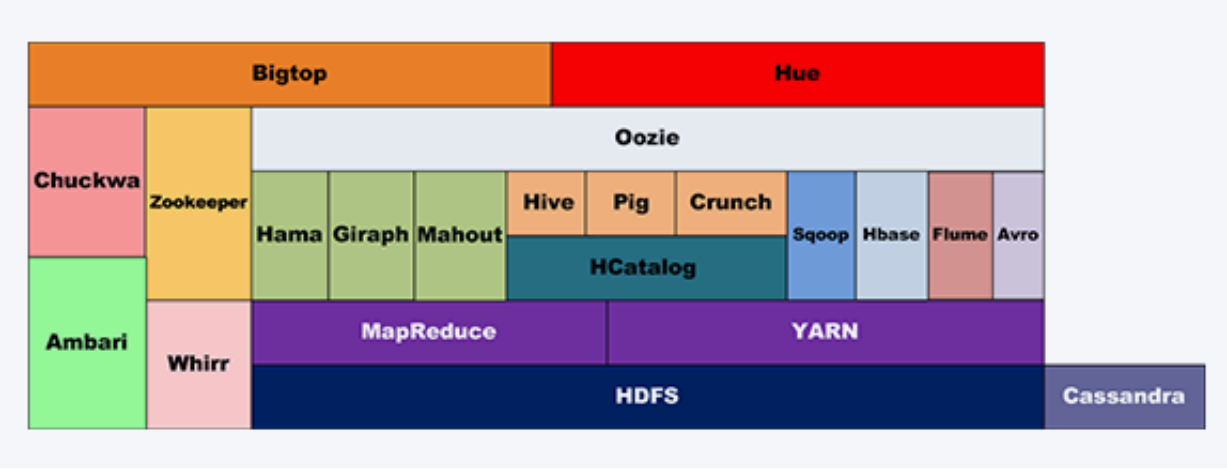
HDFS & MapReduce

Hadoop的框架最核心的设计就是：HDFS和MapReduce。HDFS为海量的数据提供了存储，MapReduce为海量的数据提供了计算。



HDFS

Hadoop Distributed FileSystem，Hadoop的分布式文件系统。
要实现MapReduce的分步式算法时，数据必需提前放在HDFS上。
HDFS基于这样的一个假设：最有效的数据处理模式是一次写入、多次读取。数据集经常从数据源生成或者拷贝一次，然后在其上做很多分析工作。分析工作经常读取其中的大部分数据，即使不是全部。 因此读取整个数据集所需时间比读取第一条记录的延时更重要。

概念

1. Block
   HDFS的Block块比一般单机文件系统大得多，默认为128M。HDFS的文件被拆分成block-sized的chunk，chunk作为独立单元存储。比Block小的文件不会占用整个Block，只会占据实际大小。例如， 如果一个文件大小为1M，则在HDFS中只会占用1M的空间，而不是128M。
2. NameNode & DataNode
   整个HDFS集群由Namenode和Datanode构成master-worker（主从）模式。
   Namenode负责构建命名空间，管理文件的元数据等，而Datanode负责实际存储数据，负责读写工作。

Namenode

Namenode存放文件系统树及所有文件、目录的元数据。元数据持久化为2种形式：

- namespace image
- edit log
  但是持久化数据中不包括Block所在的节点列表，及文件的Block分布在集群中的哪些节点上，这些信息是在系统重启的时候重新构建（通过Datanode汇报的Block信息）。
  在HDFS中，Namenode可能成为集群的单点故障，Namenode不可用时，整个文件系统是不可用的。HDFS针对单点故障提供了2种解决机制：
- 备份持久化元数据

将文件系统的元数据同时写到多个文件系统， 例如同时将元数据写到本地文件系统及NFS。这些备份操作都是同步的、原子的。

- Secondary NameNode
  Secondary节点定期合并主Namenode的namespace image和edit log， 避免edit log过大，通过创建检查点checkpoint来合并。它会维护一个合并后的namespace image副本， 可用于在Namenode完全崩溃时恢复数据。

Datanode

数据节点负责存储和提取Block，读写请求可能来自namenode，也可能直接来自客户端。数据节点周期性向Namenode汇报自己节点上所存储的Block相关信息。

命令行接口

本地文件复制到HDFS：`hadoop fs -copyFromLocal`
创建目录：`hadoop fs mkdir`
列出文件列表：`hadoop fs -ls`
hadoop中，文件、目录有3种权限：读(r)、写(w)、执行(x)。
每个文件或目录都有owner，group，mode三个属性：owner指文件的所有者，group为权限组。
mode 由所有者权限、文件所属的组中组员的权限、非所有者非组员的权限组成。
文件权限是否开启通过dfs.permissions.enabled属性来控制，这个属性默认为false，没有打开安全限制，因此不会对客户端做授权校验，如果开启安全限制，会对操作文件的用户做权限校验。特殊用户superuser是Namenode进程的标识，不会针对该用户做权限校验。

HDFS I/O（Java接口）

参考：深入理解Hadoop HDFS
实际应用中，对HDFS的大多数操作通过Java的FileSystem来操作。
Configuration：封装了客户端或者服务器的配置信息
FileSystem：此类的对象是一个文件系统对象，可以用该对象的一些方法来对文件进行操作通过FileSystem的静态方法get获得该对象，例：FileSystem hdfs = FileSystem.get(conf);
FSDataInputStream：这是HDFS中的输入流，通过由FileSystem的open方法获取
FSDataOutputStream：这是HDFS中的输出流，通过由FileSystem的create方法获取

读数据

1. 通过URL读取Hadoop中的数据
   java.net.URL类提供了资源定位的统一抽象，任何人都可以自己定义一种URL Schema，并提供相应的处理类来进行实际的操作。
   为了使用自定义的Schema，需要设置URLStreamHandlerFactory，这个操作一个JVM只能进行一次，多次操作会导致不可用，通常在静态块中完成。

```
public class URLCat {
    private static final String HDFS_PATH = "hdfs://localhost:8020/user/...";

    static {
        URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
    }

    public static void main(String[] args) {
```

```
        InputStream inputStream = null;
        try {
            inputStream = new URL(HDFS_PATH).openStream();
            IOUtils.copyBytes(inputStream, System.out, 4096, false);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            IOUtils.closeStream(inputStream);
        }
    }
}
```

2. 通过FileSystem获取数据：

- 首先获取FileSystem实例，一般使用静态get方法

```
public static FileSystem get(Configuration conf) throws IOException
public static FileSystem get(URI uri , Configuration conf) throws IOException
public static FileSystem get(URI uri , Configuration conf, String user) throws
IOException
```

若是本地文件，通过getLocal获取

```
public static LocalFileSystem getLocal(COnfiguration conf) thrown IOException
```

- 调用FileSystem的输入方法获取输入流

```
public FSDataInputStream open(Path f) throws IOException
public abstarct FSDataInputStream open(Path f , int bufferSize) throws
IOException
```

- e.g.

```
class FileSystemCat {
    private static final String HDFS_URI = "hdfs://localhost:8020";
    private static final String PATH = "/user/...";

    public static void main(String[] args) throws IOException {
        FileSystem fileSystem = FileSystem.get(URI.create(HDFS_URI), new
Configuration()); //获取FileSystem实例
        FSDataInputStream inputStream = fileSystem.open(new Path(PATH));
        IOUtils.copyBytes(inputStream, System.out, 4096, false);
        IOUtils.closeStream(inputStream);
    }
}
```

3. 通过seek(position)定位：

```
class FileSystemCat {
    private static final String HDFS_URI = "hdfs://localhost:8020";
    private static final String PATH = "/user/...";

    public static void main(String[] args) throws IOException {
        FileSystem fileSystem = FileSystem.get(URI.create(HDFS_URI), new
Configuration()); //获取FileSystem实例
        FSDataInputStream inputStream = fileSystem.open(new Path(PATH));
        IOUtils.copyBytes(inputStream, System.out, 4096, false);
        inputStream.seek(0); //回到文件开始位置
        IOUtils.copyBytes(inputStream, System.out, 4096, false);
        IOUtils.closeStream(inputStream);
    }
}
```

文件格式

Hadoop的文件格式分为面向行、面向列两种

- 面向行：同一行的数据存储在一起，即连续存储。SequenceFile,MapFile,Avro Datafile。采用这种方式，如果只需要访问行的一小部分数据，亦需要将整行读入内存，推迟序列化一定程度上可以缓解这个问题，但是从磁盘读取整行数据的开销却无法避免。面向行的存储适合于整行数据需要同时处理的情况。
- 面向列：整个文件被切割为若干列数据，每一列数据一起存储。Parquet , RCFile,ORCFile。面向列的格式使得读取数据时，可以跳过不需要的列，适合于只处于行的一小部分字段的情况。但是这种格式的读写需要更多的内存空间，因为需要缓存行在内存中（为了获取多行中的某一列）。同时不适合流式写入，因为一旦写入失败，当前文件无法恢复，而面向行的数据在写入失败时可以重新同步到最后一个同步点，所以Flume采用的是面向行的存储格式。

MapReduce

MapReduce是一种编程模型，用于大规模数据集（大于1TB）的并行运算。
"Map（映射）"和"Reduce（归约）"，是它们的主要思想，都是从函数式编程语言里借来的，还有从矢量编程语言里借来的特性。它极大地方便了编程人员在不会分布式并行编程的情况下，将自己的程序运行在分布式系统上。 当前的软件实现是指定一个Map（映射）函数，用来把一组键值对映射成一组新的键值对，指定并发的Reduce（归约）函数，用来保证所有映射的键值对中的每一个共享相同的键组。