大数据Hadoop高薪直通车课程

HBase 深入使用

讲师：轩宇（北风网版权所有)

课程大纲

1 HBase 数据存储

2 HBase Java API

3 HBase 架构剖析

4 集成 MapReduce

5 HBase 数据迁移

课程大纲

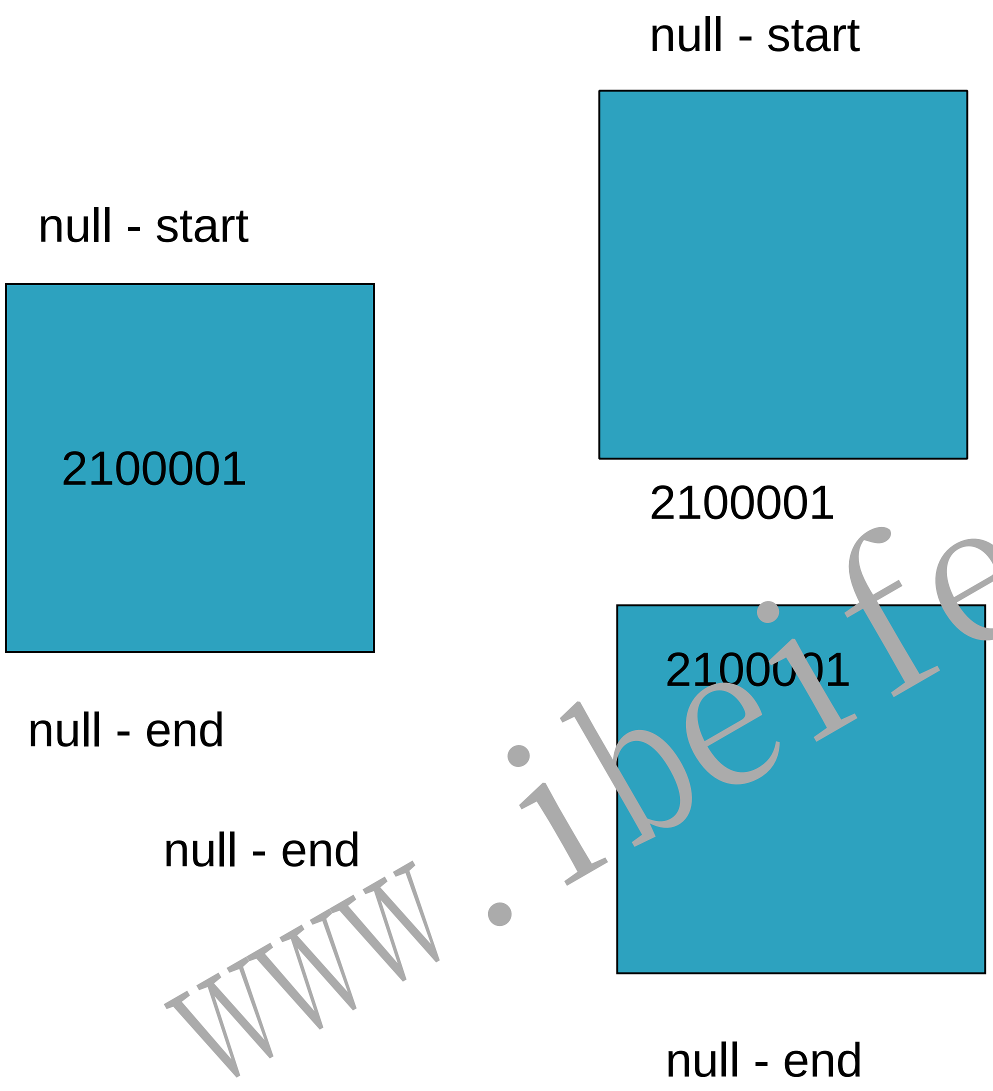
1 **HBase 数据存储**

2 **HBase Java API**

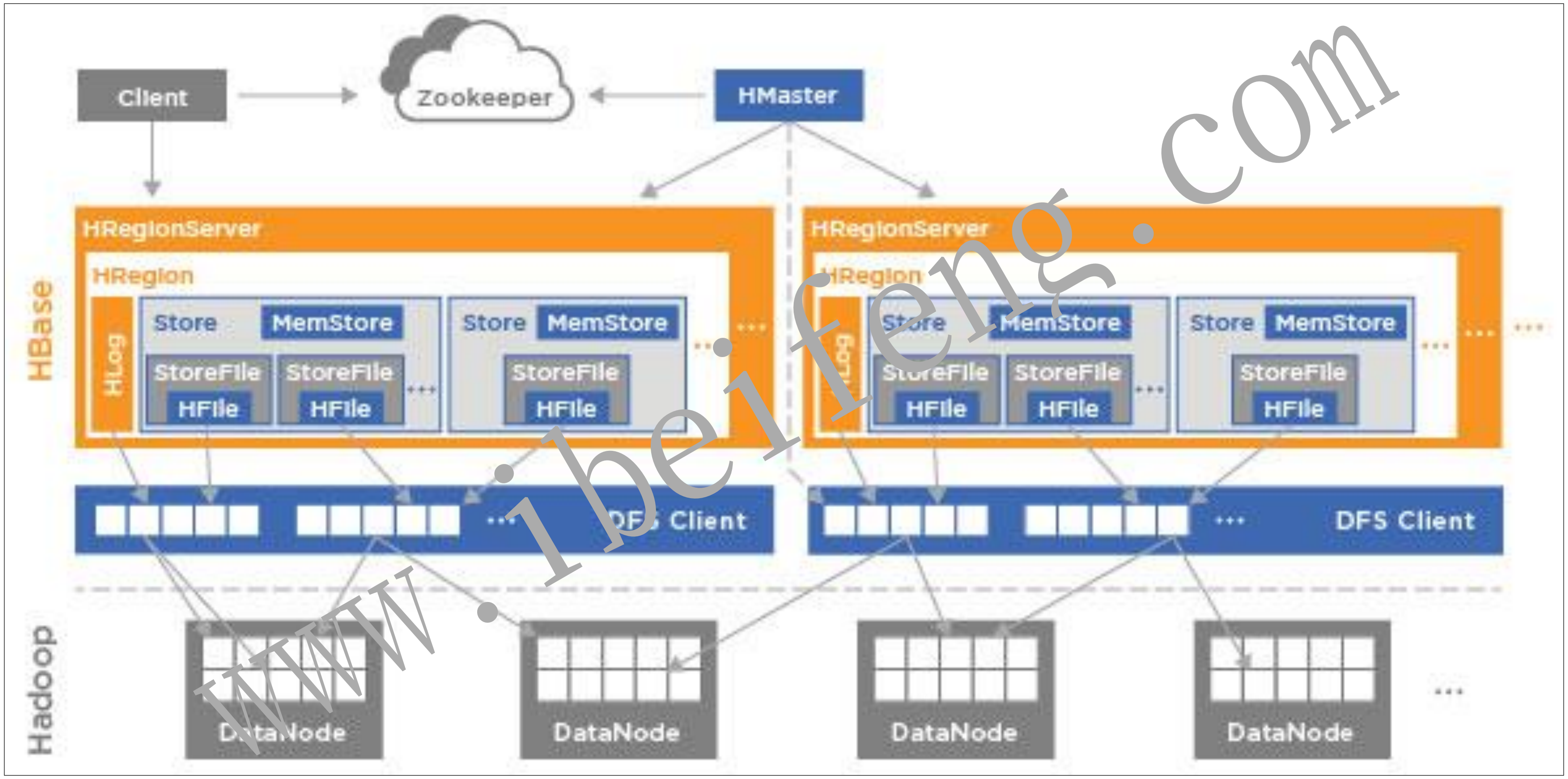
3 **HBase 架构剖析**

4 **集成 MapReduce**

5 **HBase 数据迁移**


null - start
null - start
2100001
2100001
2100001
null - end
null - end
null - end

HBase Architecture



HBase 数据存储

◆ HBase中的所有数据文件都存储在Hadoop HDFS文件系统上，主要包括上述提出的两种文件类型：

- HFile：HBase中KeyValue数据的存储格式，HFile是Hadoop的二进制格式文件，实际上StoreFile就是对HFile做了轻量级包装，进行数据的存储。
- HLog File，HBase中WAL（Write Ahead Log）的存储格式，物理上是Hadoop的Sequence File。

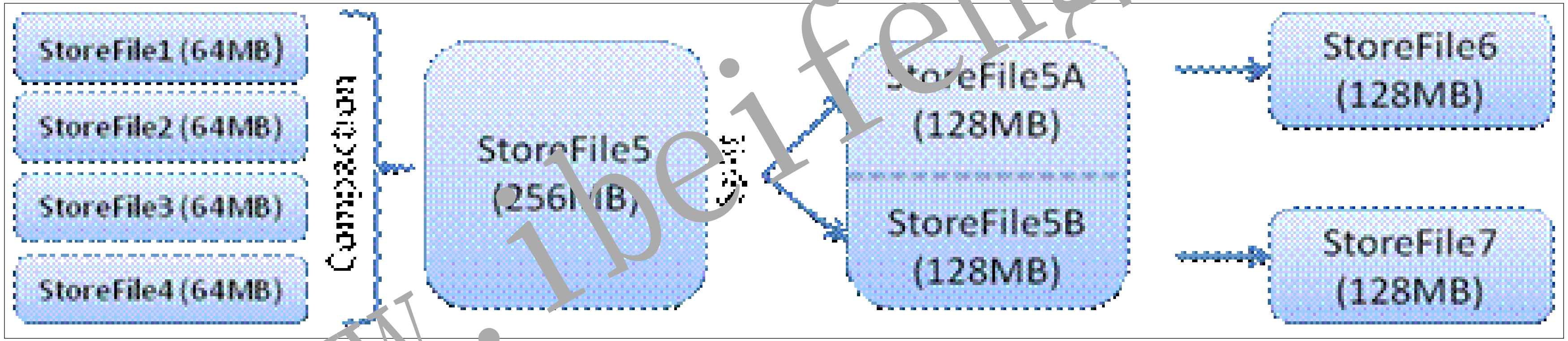
HREGION SERVER

◆ HRegionServer内部管理了一系列HRegion对象，每个HRegion对应了table中的一个region，HRegion中由多 个HStore组成。每个HStore对应了Table中的一个column family的存储，可以看出每个columnfamily其实就是一个集中的存储单元，因此最好将具备共同IO特性的column放在一个column family中，这样最高效。

◆HStore存储是HBase存储的核心，由两部分组成，一部分是MemStore，一部分是StoreFile。MemStore是 Sorted Memory Buffer，用户写入的数据首先会放入MemStore，当MemStore满了以后会Flush成一个StoreFile（底层实现是HFile）。

Memstore & StoreFile

◆用户写入数据的流程：



Memstore & StoreFile

- Client写入 -> 存入MemStore，一直到MemStore满 -> Flush成一个StoreFile，直至增长到一定阈值 -> 出发Compact合并操作 -> 多个StoreFile合并成一个StoreFile，同时进行版本合并和数据删除 -> 当StoreFiles Compact后，逐步形成越来越大的StoreFile -> 单个StoreFile大小超过一定阈值后，触发Split操作，把当前Region Split成2个Region，Region会下线，新Split出的2个孩子Region会被HMaster分配到相应的HRegionServer上，使得原先1个Region的压力得以分流到2个Region上。
- HBase只是增加数据，有所得更新和删除操作，都是在Compact阶段做的，所以，用户写操作只需要进入到内存即可立即返回，从而保证I/O高性能。

HLog文件结构

- WAL意为Write ahead log，类似Mysql中的binlog，用来做灾难恢复。Hlog记录数据的所有变更，一旦数据修改，就可以从log中进行恢复。每个HRegionServer维护一个HLog,而不是每个HRegion一个。这样不同region（来自不同table）的日志会混在一起，这样做的目的是不断追加单个文件相对于同时写多个文件而言，可以减少磁盘寻址次数，因此可以提高对table的写性能。带来的麻烦是，如果一台HRegionServer下线，为了恢复其上的region，需要将HRegionServer上的log进行拆分，然后分发到其它HRegionServer上进行恢复。

课程大纲

1 HBase 数据存储

2 HBase Java API

3 HBase 架构剖析

4 集成 MapReduce

5 HBase 数据迁移

Writing Data

A `Put` class instance is used to store data in an HBase table. For storing data in a table, create a `Put` instance with `rowkey` using any of the constructors, as follows:

```
Put(byte[] rowkey)
Put(byte[] rowArray, int rowOffset, int rowLength)
Put(byte[] rowkey, long ts)
Put(byte[] rowArray, int rowOffset, int rowLength, long ts)
Put p = new Put (Bytes.toBytes("John"));
```

HBase stores all the data, including the rowkey, in the form of a byte array and a Java utility class, bytes define various static utility methods for converting Java data types to and from a byte.

Once a `Put` instance is created using the `rowkey` component, the next step is to add the data by using either of the following method definitions:

```
add(byte[] family, byte[] qualifier, byte[] value)
add(byte[] family, byte[] qualifier, long ts, byte[] value)
add (byte[] family, ByteBuffer qualifier, long ts, ByteBuffer value)
add (Cell kv)
```

The `add()` option takes a column family along with an optional timestamp or one single cell as a parameter. In case the timestamp is not defined, the region server sets it for the data inserted. Here is a complete example of how to write data to HBase:

Reading Data

HBase uses an LRU cache for reads, which is also called the block cache. This block cache keeps the frequently accessed data from the HFiles in the memory to avoid frequent disk reads, and every column family has its own block cache. Whenever a read request arrives, the block cache is first checked for the relevant row. If it is not found, the HFiles on the disk are then checked for the same. Similar to the `Put` class, the `Get` class instance is used to read the data back from the HBase table. The HBase table defines the following method for reading the data and takes the `Get` class instance as an argument:

```
Result get(Get getInst)
```

This method extracts certain cells from a given row. Here, the `Get` class instance can be created by either of the class constructors:

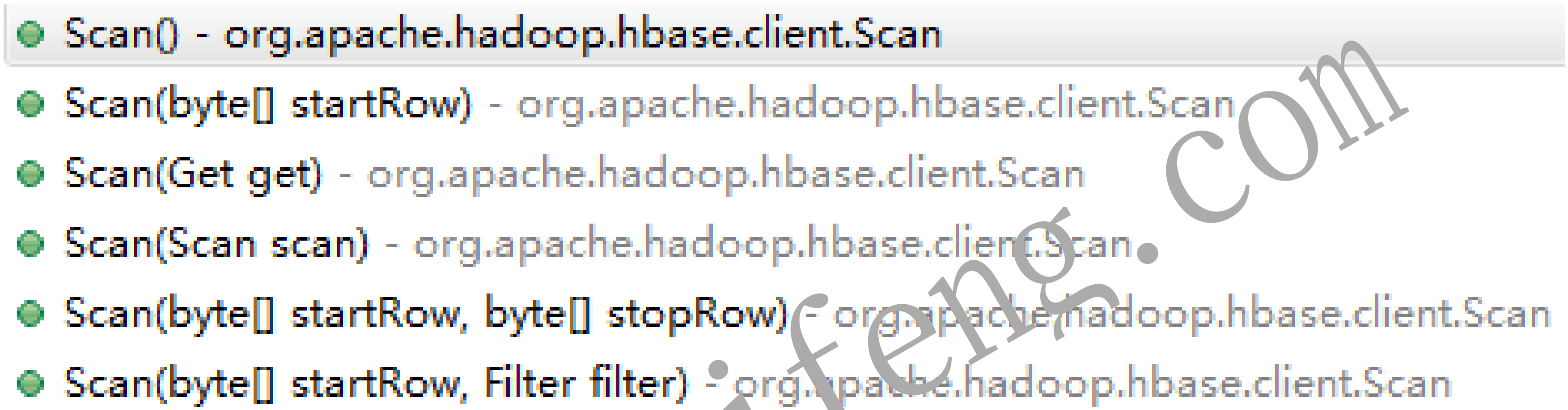
```
Get(byte[] rowkey)
```

Reading Data

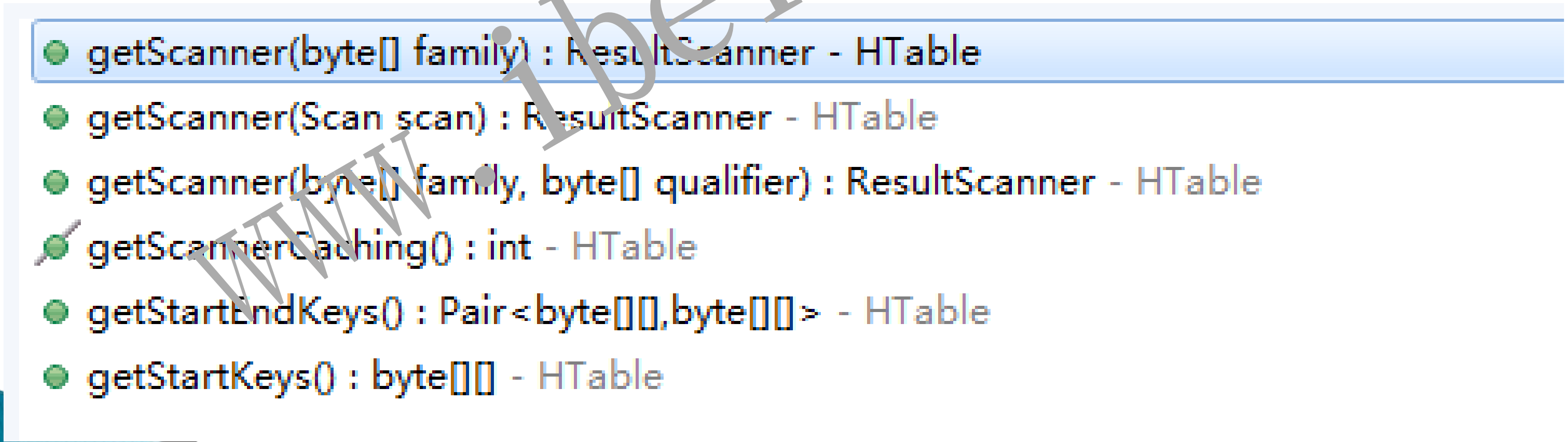
This constructor creates a `Get` operation for the specified row identified by the rowkey. For narrowing down the data search to a specific cell, additional methods are provided in the following table:

Method name	Description
`addFamily(byte[] family)`	Get all columns from the specified family
`addColumn(byte[] family, byte[] qualifier)`	Get the column from the specific family with the specified qualifier
`setTimeRange(long minStamp, long maxStamp)`	Get versions of columns only within the specified timestamp range (minStamp, maxStamp)
`setTimeStamp(long timestamp)`	Get versions of columns with the specified timestamp
`setMaxVersions(int max versions)`	Get up to the specified number of versions of each column. The default value of the maximum version returned is 1 which is the latest cell value.

Scan & ResultScanner



- Scan() - org.apache.hadoop.hbase.client.Scan
- Scan(byte[] startRow) - org.apache.hadoop.hbase.client.Scan
- Scan(Get get) - org.apache.hadoop.hbase.client.Scan
- Scan(Scan scan) - org.apache.hadoop.hbase.client.Scan
- Scan(byte[] startRow, byte[] stopRow) - org.apache.hadoop.hbase.client.Scan
- Scan(byte[] startRow, Filter filter) - org.apache.hadoop.hbase.client.Scan



- getScanner(byte[] family) : ResultScanner - HTable
- getScanner(Scan scan) : ResultScanner - HTable
- getScanner(byte[] family, byte[] qualifier) : ResultScanner - HTable
- getScannerCaching() : int - HTable
- getStartEndKeys() : Pair<byte[][],byte[][]> - HTable
- getStartKeys() : byte[][] - HTable

Deleting Data

The `Delete` command only marks the cell for deletion rather than deleting the data immediately. The actual deletion is performed when the compaction of HFiles is done to reconcile these marked records and to free the space occupied by the deleted data.

Compaction is the process of choosing HFiles from a region and combining them. In a major compaction process, it picks all the HFiles and writes back the key-values to the output HFile that are not marked as deleted. Whereas, in a minor compaction, it only takes a few files placed together and combines them into one. Also, minor compaction does not filter the deleted files. The compaction process takes care of the versions and uses the ExploringCompactionPolicy algorithms internally.

Deleting Data

Similar to the Put and Get classes, the Delete class instance is used to delete the data from the HBase table. The HBase table defines the following method for deleting the data, which takes the Delete class instance as an argument:

```
void delete(Delete deleteInst)
```

This method deletes the latest cells from a given row. Here, the Delete class instance can be created using either of the class constructors:

```
Delete(byte[] row)
Delete(byte[] rowArray, int rowOffset, int rowLength)
Delete(byte[] rowArray, int rowOffset, int rowLength, long ts)
Delete(byte[] row, long timestamp)
Delete(Delete d)
```

This constructor creates a Delete operation for the specified row identified by the rowkey. For narrowing down the data search to a specific cell, additional methods provided within the Delete class are as follows:

Deleting Data

Method name	Description

`deleteColumn(byte[] family, byte[] qualifier)`
`deleteColumn(byte[] family, byte[] qualifier, long timestamp)`
	This deletes the latest version of the specified column based on the timestamp

`deleteColumns(byte[] family, byte[] qualifier)`	This deletes all the versions of the specified column
`deleteFamily(byte[] family)`	This deletes all the versions of all columns of the specified family

Method name	Description
`deleteFamily(byte[] family, long timestamp)`	This deletes all the columns of the specified family with a timestamp less than or equal to the specified timestamp
`deleteFamilyVersion(byte[] family, long timestamp)`	This deletes all the columns of the specified family with a timestamp equal to the specified timestamp

课程大纲

1 HBase 数据存储

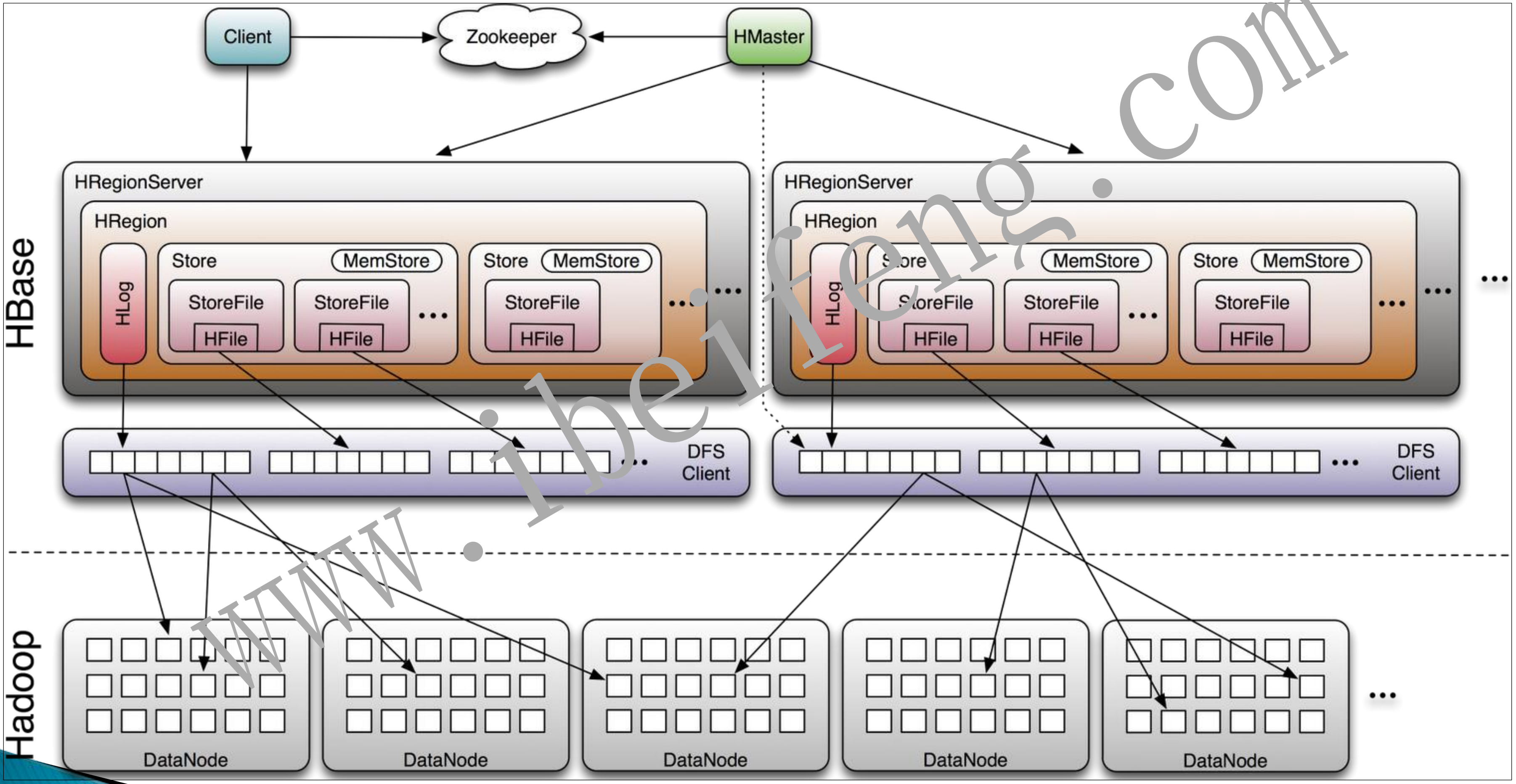
2 HBase Java API

3 HBase 架构剖析

4 集成 MapReduce

5 HBase 数据迁移

HBase Architecture



客户端Client

◆ Client

➢ 整个HBase集群的访问入口；

➢ 使用HBase RPC机制与HMaster和HRegionServer进行通信；

➢ 与HMaster进行通信进行管理类操作；

➢ 与HRegionServer进行数据读写类操作；

➢包含访问HBase的接口，并维护cache来加快对HBase的访问

协调服务组件Zookeeper

- Zookeeper
  - 保证任何时候，集群中只有一个HMaster；
  - 存贮所有HRegion的寻址入口；
  - 实时监控HRegion Server的上线和下线信息，并实时通知给HMaster；
  - 存储HBase的schema和table元数据；
  - Zookeeper Quorum存储-ROOT-表地址、HMaster地址。 （？？？）

主节点HMaster

◆HMaster

- HMaster没有单点问题，HBase中可以启动多个HMaster，通过Zookeeper的Master Election机制保证总有一个Master在运行主要负责Table和Region的管理工作。
- 管理用户对table的增删改查操作；
- 管理HRegionServer的负载均衡，调整Region分布；
- Region Split后，负责新Region的分布；
- 在HRegionServer停机后，负责失效HRegionServer上Region迁移工作。

Region节点HRegionServer

◆ HRegion Server

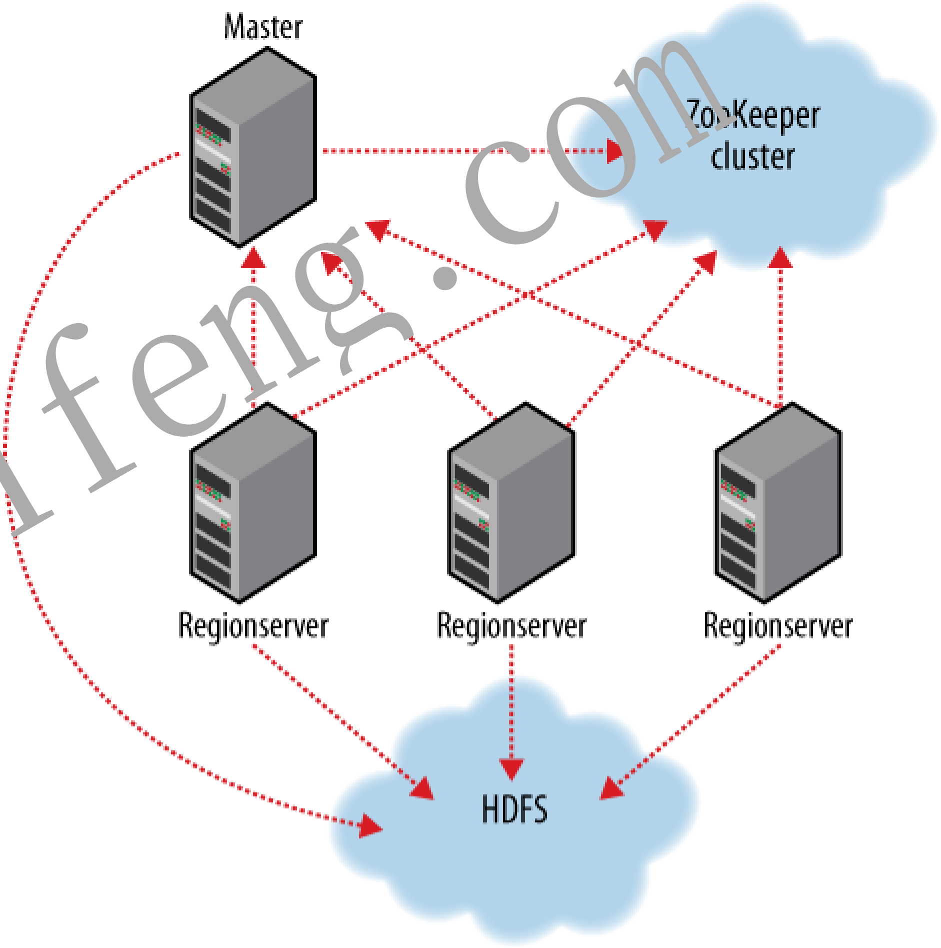
➢ 维护HRegion，处理对这些HRegion的IO请求，向HDFS文件系统中读写数据；

➢ 负责切分在运行过程中变得过大的HRegion。

➢ Client访问hbase上数据的过程并不需要master参与（寻址访问Zookeeper和HRegion Server，数据读写访问HRegione Server），HMaster仅仅维护者table和Region的元数据信息，负载很低。

HBase & Zookeeper

- HBase 依赖ZooKeeper;
- 默认情况下，HBase 管理 ZooKeeper 实例，比如， 启动或者停止ZooKeeper;
- HMaster与HRegionServers 启动时会向ZooKeeper注册;
- Zookeeper的引入使得HMaster不再是单点故障。



课程大纲

1 HBase 数据存储

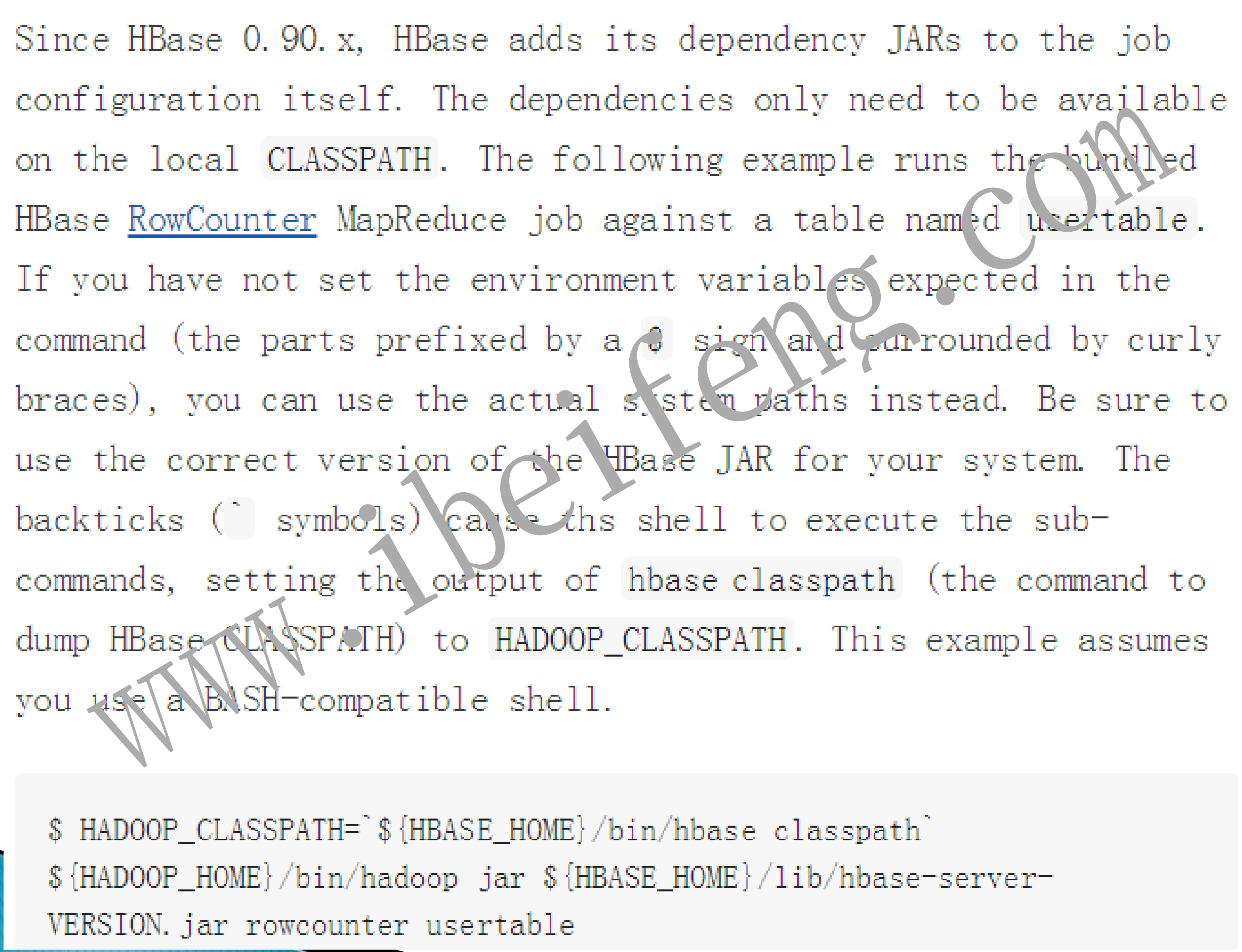
2 HBase Java API

3 HBase 架构剖析

4 集成 MapReduce

5 HBase 数据迁移

HBase & MapReduce & CLASSPATH



Since HBase 0.90.x, HBase adds its dependency JARs to the job configuration itself. The dependencies only need to be available on the local `CLASSPATH`. The following example runs the bundled HBase RowCounter MapReduce job against a table named `usertable`. If you have not set the environment variables expected in the command (the parts prefixed by a `$` sign and surrounded by curly braces), you can use the actual system paths instead. Be sure to use the correct version of the HBase JAR for your system. The backticks (` symbols) cause the shell to execute the sub-commands, setting the output of `hbase classpath` (the command to dump HBase CLASSPATH) to `HADOOP_CLASSPATH`. This example assumes you use a BASH-compatible shell.

```
$ HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase classpath`
${HADOOP_HOME}/bin/hadoop jar ${HBASE_HOME}/lib/hbase-server-
VERSION.jar rowcounter usertable
```

Bundled HBase MapReduce Jobs

```
export HBASE_HOME=/opt/modules/hbase-0.98.6-hadoop2

export HADOOP_HOME=/opt/modules/hadoop-2.5.0

HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase classpath` $HADOOP_HOME/bin/yarn
jar $HBASE_HOME/lib/hbase-server-0.98.6-hadoop2.jar
```

Bundled HBase MapReduce Jobs

The HBase JAR also serves as a Driver for some bundled MapReduce jobs. To learn about the bundled MapReduce jobs, run the following command.

```
$ ${HADOOP_HOME}/bin/hadoop jar ${HBASE_HOME}/hbase-server-VERSION.jar
An example program must be given as the first argument.
Valid program names are:
  copytable: Export a table from local cluster to peer cluster
  completebulkload: Complete a bulk data load.
  export: Write table data to HDFS.
  import: Import data written by Export.
  importtsv: Import data in TSV format.
  rowcounter: Count rows in HBase table
```

Each of the valid program names are bundled MapReduce jobs. To run one of the jobs, model your command after the following example.

```
$ ${HADOOP_HOME}/bin/hadoop jar ${HBASE_HOME}/hbase-server-VERSION.jar rowcounter myTable
```

Running MapReduce over HBase

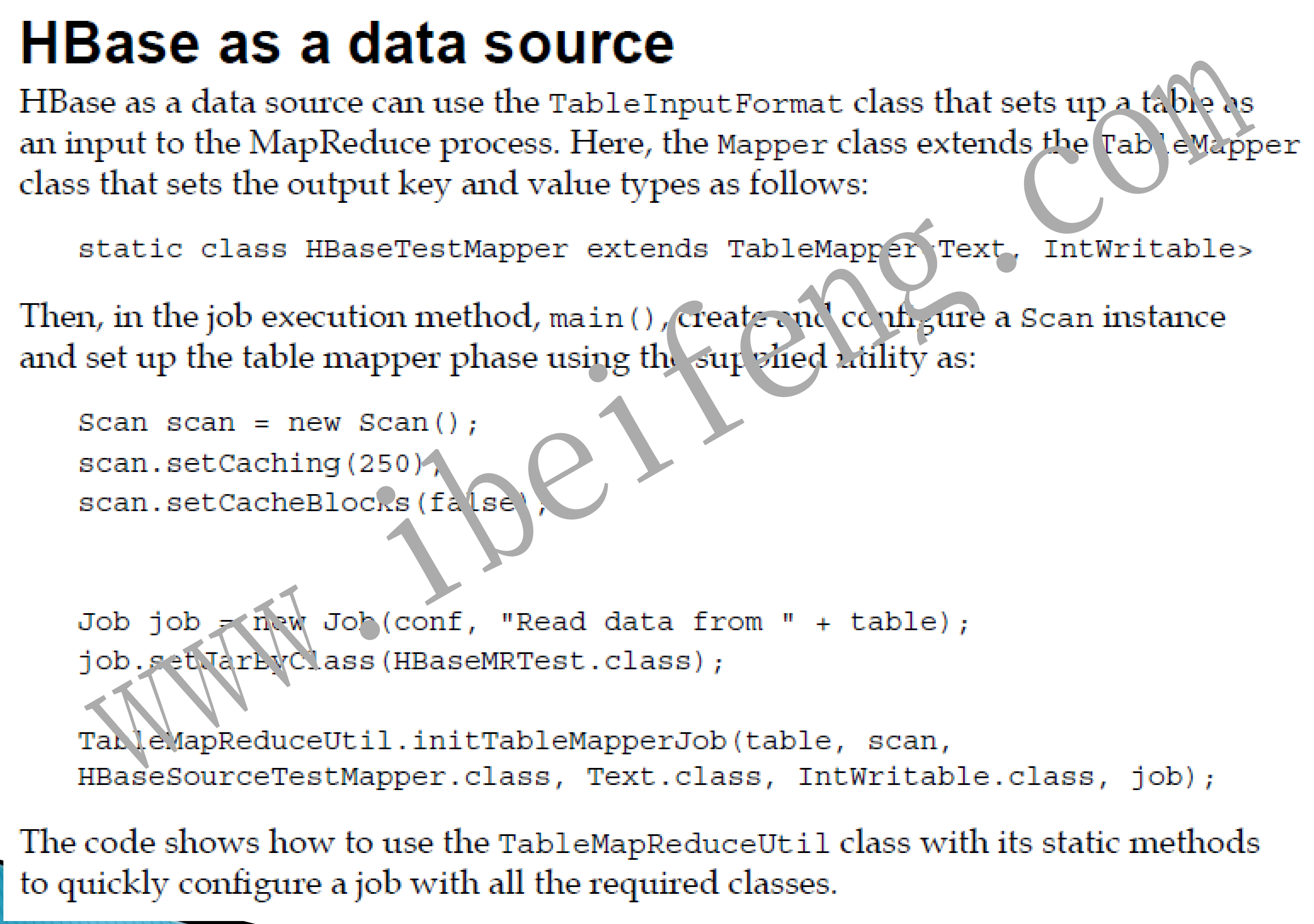
While implementing `Mapper`, `Reducer`, and main driver class, these guidelines should be followed:

- The `Mapper` class:
  - The `Mapper` class should extend the `TableMapper` class
  - The `map` method of the `Mapper` class takes the rowkey of the Hbase table as an input key
  - The define input key is the `ImmutableBytesWritable` object
  - Another parameter, the `org.apache.hadoop.hbase.client.Result` object contains the input values as column/column-families from the HBase table
- The `Reducer` class
  - The `Mapper` class should extend the `TableReducer` class
  - The output key is defined as `NULL`
  - The output value is defined as the `org.apache.hadoop.hbase.client.Put` object.
- The `Main` class
  - Configure the `org.apache.hadoop.hbase.client.Scan` object and optionally define parameters such as start row, stop row, row filter, columns, and the column-families for the scan object.
  - Set the record caching size (the default is `1`, which is not preferred for MapReduce) for scan object.

Running MapReduce over HBase

- Set the block cache for scan object as `false`
- Define the input table using the `TableMapReduceUtil.initTableMapperJob` method. This method takes the source table name, scan object, the `Mapper` class name, `MapOutputKey`, `MapOutputValue`, and the `Job` object.
- Define the input table using `TableMapReduceUtil.initTableReducerJob`. This method takes the target table name, the `Reducer` class name and the `Job` object.

HBase as a data source



HBase as a data source

HBase as a data source can use the TableInputFormat class that sets up a table as an input to the MapReduce process. Here, the Mapper class extends the TableMapper class that sets the output key and value types as follows:

```
static class HBaseTestMapper extends TableMapper<Text, IntWritable>
```

Then, in the job execution method, main(), create and configure a Scan instance and set up the table mapper phase using the supplied utility as:

```
Scan scan = new Scan();
scan.setCaching(250);
scan.setCacheBlocks(false);


Job job = new Job(conf, "Read data from " + table);
job.setJarByClass(HBaseMRTest.class);

TableMapReduceUtil.initTableMapperJob(table, scan,
HBaseSourceTestMapper.class, Text.class, IntWritable.class, job);
```

The code shows how to use the TableMapReduceUtil class with its static methods to quickly configure a job with all the required classes.

HBase as a data sink

HBase as a data sink can also use the TableOutputFormat class that sets up a table as an output to the MapReduce process as:

```
Job job = new Job(conf, "Writing data to the " + table);

job.setOutputFormatClass(TableOutputFormat.class);
job.getConfiguration().set(TableOutputFormat.OUTPUT_TABLE, table);
```

The preceding lines also uses an implicit write buffer set up by the TableOutputFormat class. The call to context.write() issues an internal table.put() interface with the given instance of Put. The TableOutputFormat class also takes care of calling flushCommits() when the job is complete.

In a typical MapReduce usage with HBase, a reducer is not usually needed as data is already sorted and has unique keys to be stored in the HBase tables. If a reducer is required for certain use cases, it should extend the TableReducer class that again sets the input key and value types as:

```
static class HBaseSourceTestReduce extends TableReducer<.,.>
```

Also, set it in the job configuration as:

```
TableMapReduceUtil.initTableReducerJob("customers", HBaseTestReduce.
class, job);
```

Here, the writes go to the region that is responsible for the rowkey that is being written by the reduce task.

课程大纲

1. HBase 数据存储
2. HBase Java API
3. HBase 架构剖析
4. 集成 MapReduce
5. HBase 数据迁移

Data Migration

There are several ways to move data into HBase:

- Using the HBase Put API
- Using the HBase bulk load tool
- Using a customized MapReduce job

The HBase **Put** API is the most straightforward method. Its usage is not difficult to learn. For most situations however, it is not always the most efficient method. This is especially true when a large amount of data needs to be transferred into HBase within a limited time period. The volume of data to be taken care of is usually huge, and that's probably why you will be using HBase rather than another database. You have to think about how to move all that data into HBase carefully at the beginning of your HBase project; otherwise you might run into serious performance problems.

Data Migration

The most usual case of data migration might be importing data from an existing RDBMS into HBase. For this kind of task, the most simple and straightforward way could be to fetch the data from a single client and then put it into HBase, using the HBase Put API. It works well if there is not too much data to transfer.

This recipe describes importing data from MySQL into HBase using its Put API. All the operations will be executed on a single client. MapReduce is not included in this recipe. This recipe leads you through creating an HBase table via HBase Shell, connecting to the cluster from Java, and then putting data into HBase.

- RDBMS 抽取数据
  - JDBC，通用，实时性（增量/全量）
- HBase 插入数据
  - 多线程、通用性
- **Kettle**

ImportTsv

ImportTsv is a utility that will load data in TSV format into HBase. It has two distinct usages: loading data from TSV format in HDFS into HBase via Puts, and preparing StoreFiles to be loaded via the `completebulkload`.

To load data via Puts (i.e., non-bulk loading):

```
$ bin/hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -
Dimporttsv.columns=a,b,c <tablename> <hdfs-inputdir>
```

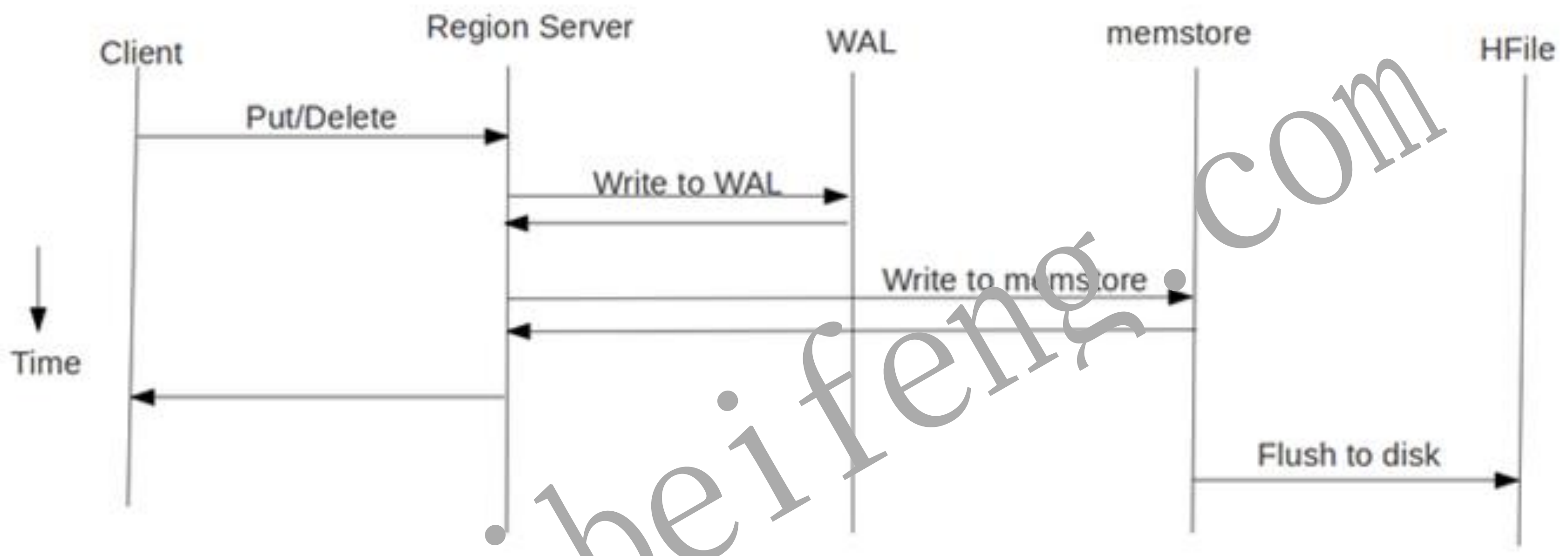
To generate StoreFiles for bulk-loading:

```
$ bin/hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -
Dimporttsv.columns=a,b,c -Dimporttsv.bulk.output=hdfs://storefile-outputdir
<tablename> <hdfs-data-inputdir>
```

ImportTsv Example

```
export HBASE_HOME=/opt/modules/hbase-0.98.6-hadoop2
export HADOOP_HOME=/opt/modules/hadoop-2.5.0
HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase mapredcp`:${HBASE_HOME}/conf \
        ${HADOOP_HOME}/bin/yarn jar \
        ${HBASE_HOME}/lib/hbase-server-0.98.6-hadoop2.jar importtsv \
        -Dimporttsv.columns=HBASE_ROW_KEY,\
        info:name,info:age,info:sex,info:address,info:phone \
        h_user \
        hdfs://hadoop-senior.ibeifeng.com:8020/user/beifeng/hbase/importtsv
```

Bulk Load



通常 MapReduce 在写HBase时使用的是 TableOutputFormat 方式，在reduce中直接生成put对象写入HBase，该方式在大数据量写入时效率低下（HBase会block写入，频繁进行flush，split，compact等大量IO操作），并对HBase节点的稳定性造成一定的影响（GC时间过长，响应变慢，导致节点超时退出，并引起一系列连锁反应）。

Bulk Load

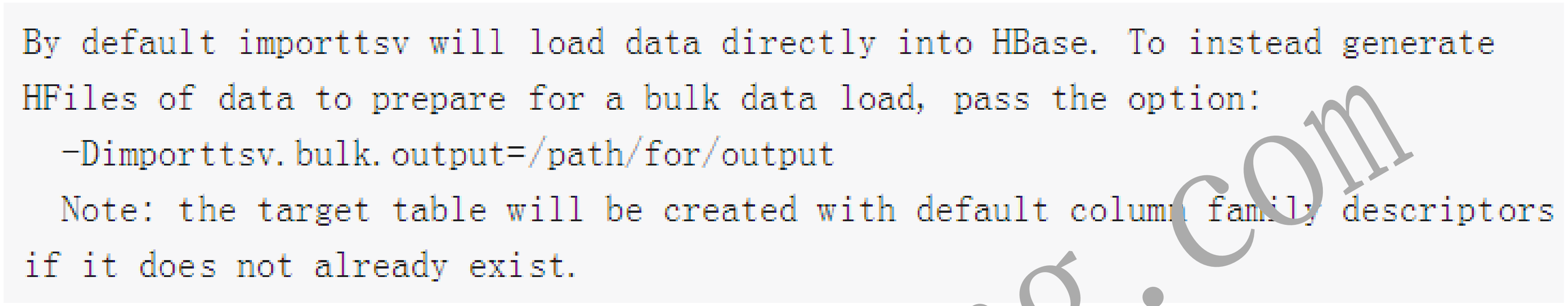
HBase支持 bulk load 的入库方式，它是利用hbase的数据信息按照特定格式存储在hdfs内这一原理，直接在HDFS中生成持久化的HFile数据格式文件，然后上传至合适位置，即完成巨量数据快速入库的办法。配合mapreduce完成，高效便捷，而且不占用region资源，增添负载，在大数据量写入时能极大的提高写入效率，并降低对HBase节点的写入压力。

通过使用先生成HFile，然后再BulkLoad到Hbase的方式来替代之前直接调用HTableOutputFormat的方法有如下的好处：

（1）消除了对HBase集群的插入压力

（2）提高了Job的运行速度，降低了Job的执行时间

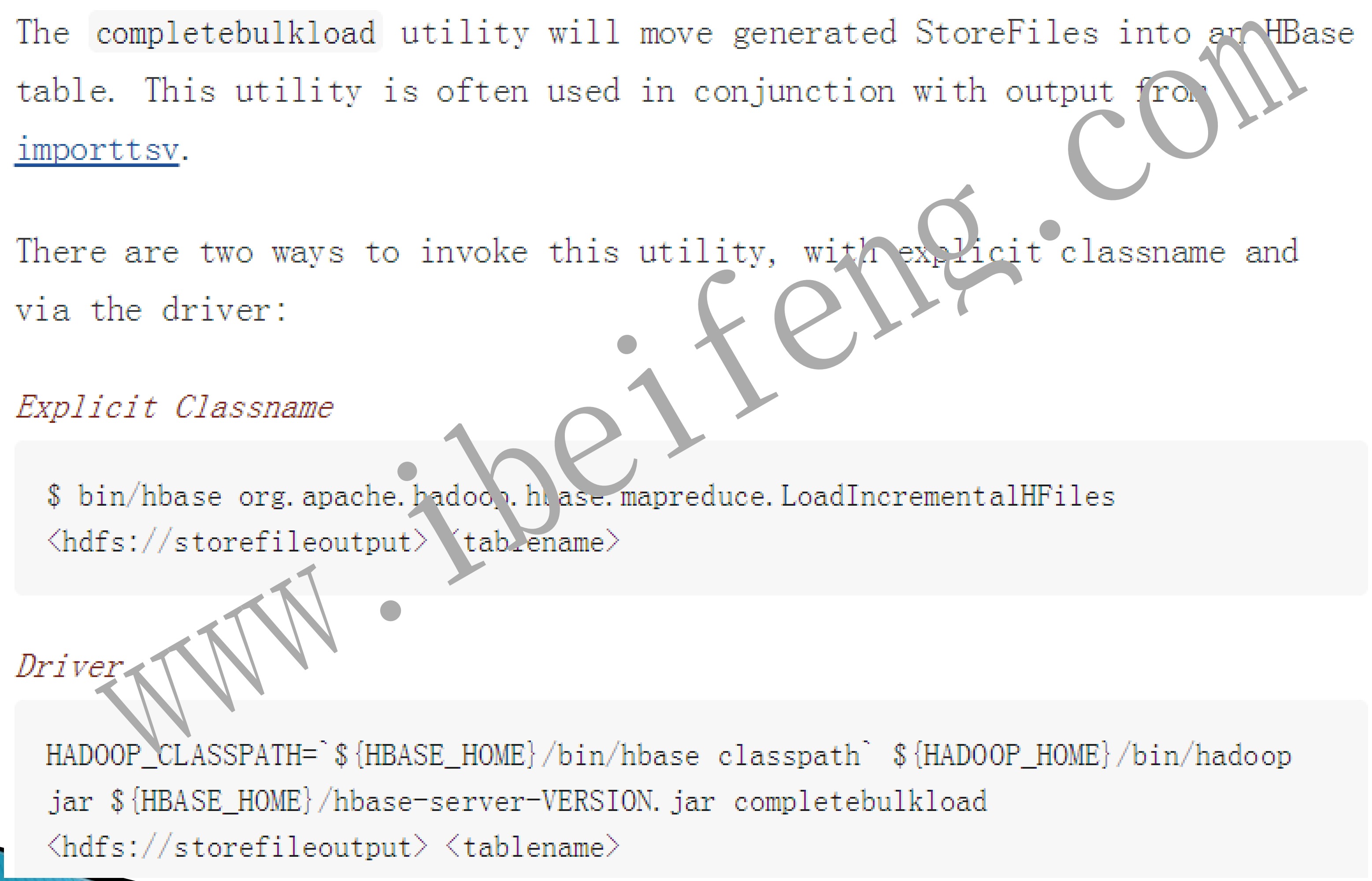
Complete BulkLoad



```
By default importtsv will load data directly into HBase. To instead generate
HFiles of data to prepare for a bulk data load, pass the option:
  -Dimporttsv.bulk.output=/path/for/output
  Note: the target table will be created with default column family descriptors
if it does not already exist.
```

```
export HBASE_HOME=/opt/modules/hbase-0.98.6-hadoop2
export HADOOP_HOME=/opt/modules/hadoop-2.5.0
HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase mapredcp`:${HBASE_HOME}/conf \
        ${HADOOP_HOME}/bin/yarn jar \
        ${HBASE_HOME}/lib/hbase-server-0.98.6-hadoop2.jar importtsv \
        -Dimporttsv.columns=HBASE_ROW_KEY,\
        info:name,info:age,info:sex,info:address,info:phone \
        -Dimporttsv.bulk.output=hdfs://hadoop-senior.ibeifeng.com:8020/user/beifeng/hbase/hfileoutput
        h_user \
        hdfs://hadoop-senior.ibeifeng.com:8020/user/beifeng/hbase/importtsv
```

Complete Bulk Load



The `completebulkload` utility will move generated StoreFiles into an HBase table. This utility is often used in conjunction with output from importtsv.

There are two ways to invoke this utility, with explicit classname and via the driver:

Explicit Classname

```
$ bin/hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles
<hdfs://storefileoutput> <tablename>
```

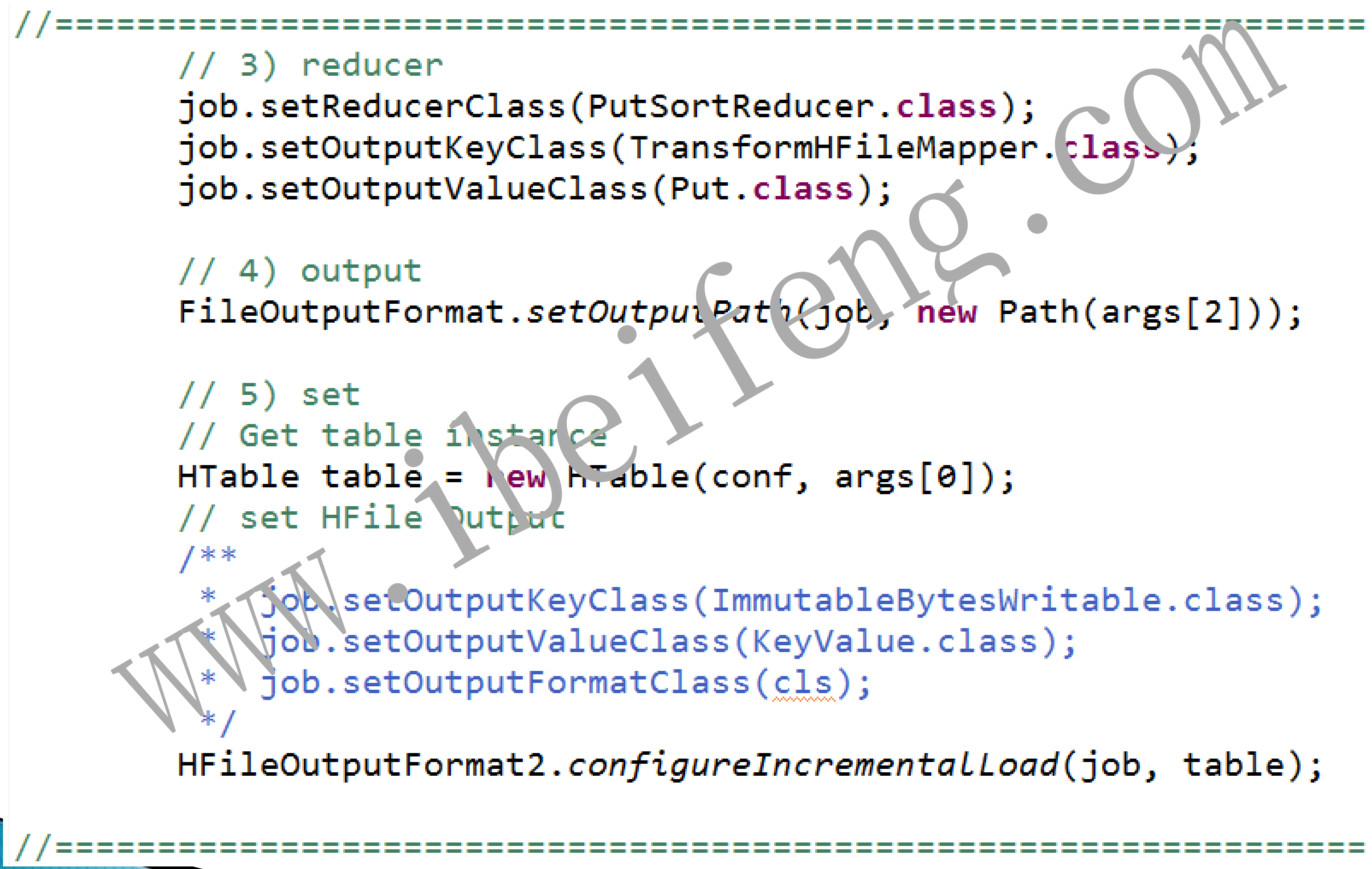
Driver

```
HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase classpath` ${HADOOP_HOME}/bin/hadoop
jar ${HBASE_HOME}/hbase-server-VERSION.jar completebulkload
<hdfs://storefileoutput> <tablename>
```

Complete Bulk Load

```
export HBASE_HOME=/opt/modules/hbase-0.98.6-hadoop2
export HADOOP_HOME=/opt/modules/hadoop-2.5.0
HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase mapredcp`:${HBASE_HOME}/conf \
        ${HADOOP_HOME}/bin/yarn jar \
        ${HBASE_HOME}/lib/hbase-server-0.98.6-hadoop2.jar \
        completebulkload \
        hdfs://hadoop-senior.ibeifeng.com:8020/user/beifeng/hbase/hfileoutput \
        h_user
```

Generating HFile files in MapReduce



```
//==================================================================
        // 3) reducer
        job.setReducerClass(PutSortReducer.class);
        job.setOutputKeyClass(TransformHFileMapper.class);
        job.setOutputValueClass(Put.class);

        // 4) output
        FileOutputFormat.setOutputPath(job, new Path(args[2]));

        // 5) set
        // Get table instance
        HTable table = new HTable(conf, args[0]);
        // set HFile Output
        /**
         *  job.setOutputKeyClass(ImmutableBytesWritable.class);
         *  job.setOutputValueClass(KeyValue.class);
         *  job.setOutputFormatClass(cls);
         */
        HFileOutputFormat2.configureIncrementalLoad(job, table);

//==================================================================
```