目录

初识Hadoop....2
Hadoop简介....2
Hadoop相关名词....2
Hadoop的架构....2
HDFS....3
MapReduce....3
分布式并行运算....5
MapReduce基本编程....6
概述....6
实践....6
创建一个maven工程....6
加入hadoop依赖....6
编写Map类....7
编写Reduce类....8
定义job....8
MapReduce测试....9
用MRUnit做单元测试....9
加入mrunit依赖....9
单独测试Map....10
单独测试Reduce....10
测试MapReduce....11
运行MapReduce Job进行集成测试....13
准备工作....13
单机运行测试....13
伪分布式运行测试....14
完全分布式运行测试....15
集成测试总结....15
使用Eclipse插件....16
插件安装....16
插件配置与使用....16
指定Hadoop安装目录....16
打开Map/Reduce视图。....17
新建 Hadoop location....17
DFS视图浏览文件....18
运行MapReduce作业....18
查看运行结果....18

初识 Hadoop

Hadoop 简介

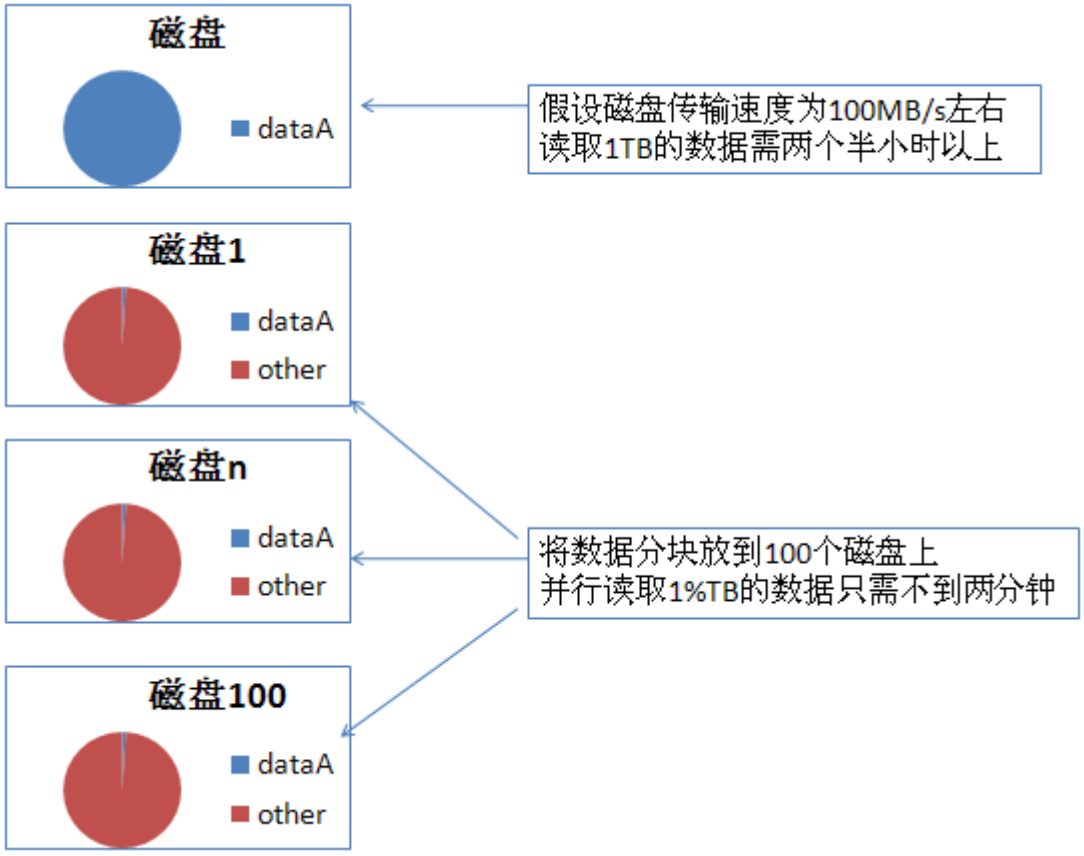
Hadoop 是一个实现了 MapReduce 计算模型的开源分布式并行编程框架,借助于 Hadoop, 程序员可以轻松地编写分布式并行程序，将其运行于计算机集群上，完成海量数据的计算。

Hadoop 相关名词

如果你对 Hadoop 不熟悉，下面这些名词你或许耳熟能详，Google File System、Google Map/Reduce 模型、lucene 、Nutch、阿里云梯、云计算等，他们都与 Hadoop 有着或深或浅的关系，原来 Hadoop 离我们如此之近，那么花些时间继续了解它吧。

Hadoop 的架构

先抛开 Hadoop，简单地想想看，假设我们需要读一个 10TB 的数据集，怎么办?在传统的系统上，这需要很长时间，因为硬盘的传输速度是受限的。一个简单的办法是将数据存储在多个磁盘上，同时从多个磁盘并行读取数据，这将大大减少读取时间。

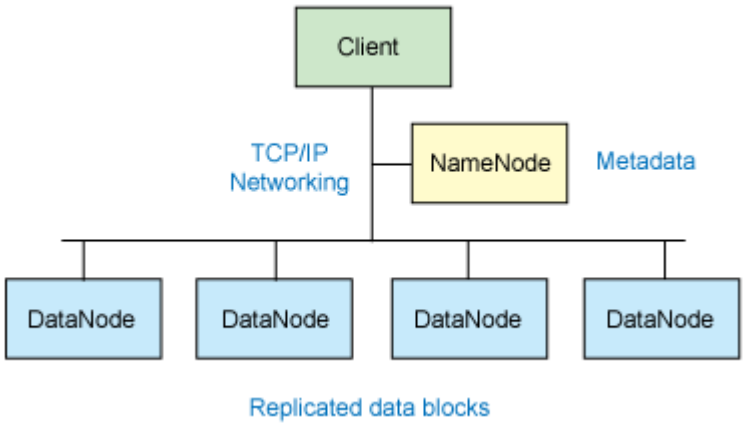


上面的方案需要解决两个主要问题，一个是硬件故障，要保证其中一个硬件坏了但数据

仍然完整，Hadoop 的文件系统 HDFS（Hadoop Distributed Filesystem）提供了一种解决方式。另外一个问题是如何并行读取数据并合并保证正确性，通过 MapReduce 的编程模型可以简化这个问题。简而言之，Hadoop 提供了一个稳定的共享存储和分析系统，存储由 HDFS 实现，分析由 MapReduce 实现，这两者构成了 Hadoop 的核心功能。

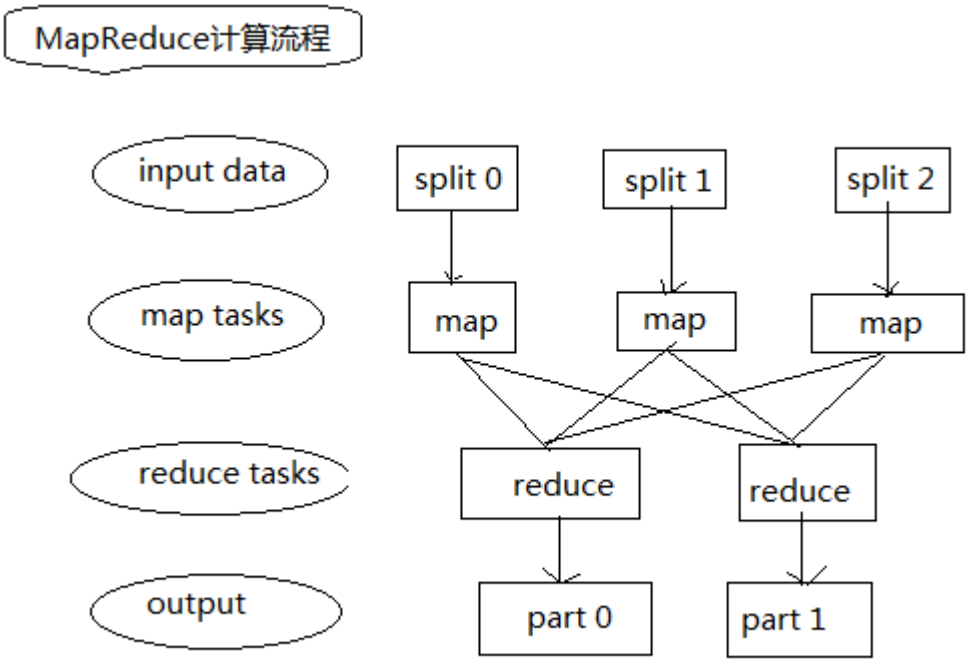
HDFS

HDFS 集群有两种节点，以管理者-工作者模式运行，即 1 个名称节点（NameNode）和 N 个数据节点（DataNode）。其底层实现是将文件切割成块，然后将这些块存储在不同的 DataNode 上。为了容错容灾，每个块还被复制多份存储在不同的 DataNode 上。NameNode 管理文件系统的命名空间，记录每个文件被切割成了多少块，这些块可以从哪些 DataNode 上获得，以及各个 DataNode 的状态信息等。下图是 Hadoop 集群的简化视图



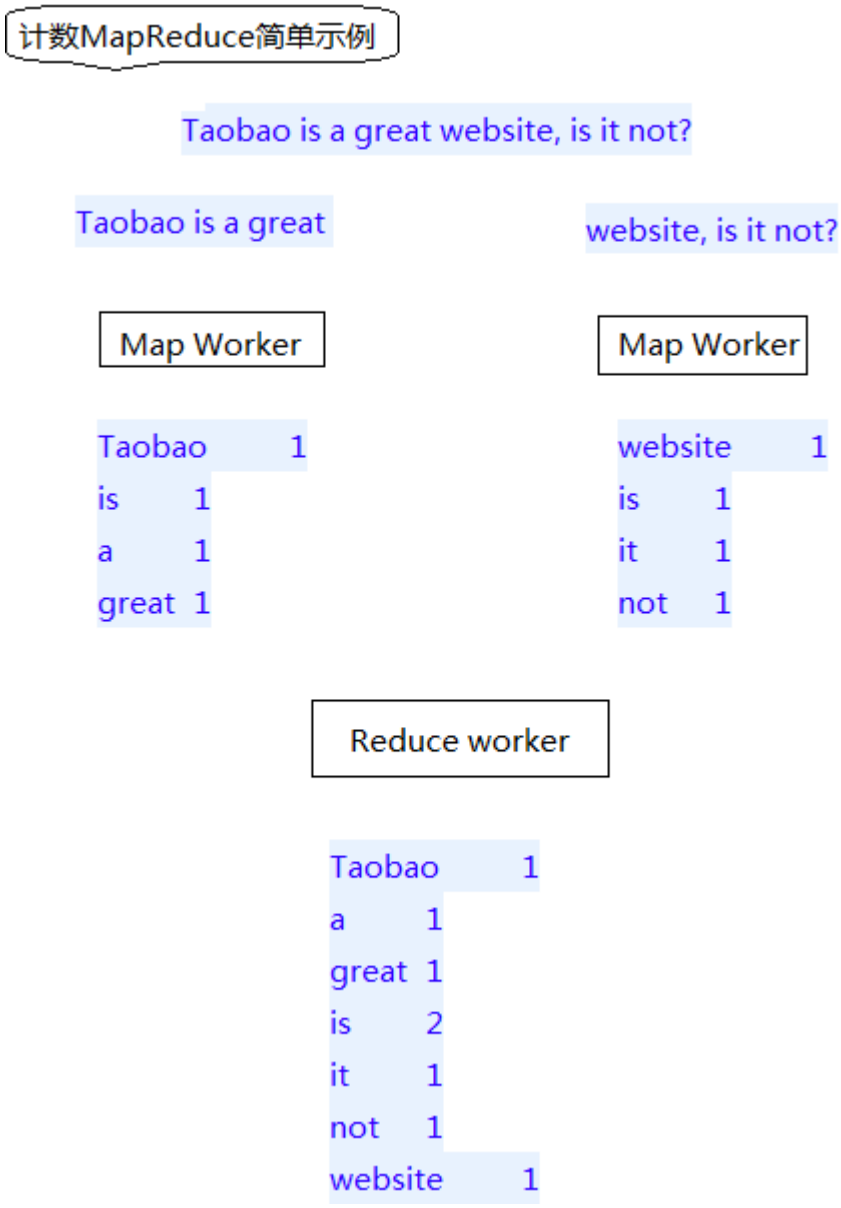
HDFS 内部通信都是基于标准的 TCP/IP 协议，NameNode 依赖来自每个 DataNode 的定期心跳（heartbeat）消息。每条消息都包含一个块报告，NameNode 可以根据这个报告验证块映射和其他文件系统元数据。如果 DataNode 不能发送心跳消息，NameNode 将采取修复措施，重新复制在该节点上丢失的块。更多 HDFS 的架构和设计请阅读 http://hadoop.apache.org/common/docs/current/cn/hdfs_design.html

MapReduce



上图说明了用 MapReduce 来处理大数据集的过程，这个 MapReduce 的计算过程简而言之，就是将大数据集分解为成若干个小数据集，每个(或若干个)数据集分别由集群中的一个结点(一般就是一台普通的计算机)进行处理并生成中间结果，然后这些中间结果又由大量的结点进行合并，形成最终结果。

计算模型的核心是 Map 和 Reduce 两个函数，这两个函数由用户负责实现，功能是按一定的映射规则将输入的 <key, value> 对转换成另一个或一批 <key, value> 对输出。下图是一个简单的 MapReduce 示例，实现字数统计功能。



分布式并行运算

Hadoop 的分布式并行运算有一个作为主控的 JobTracker，用于调度和管理其它的 TaskTracker, JobTracker 可以运行于集群中任一台计算机上。TaskTracker 负责执行任务，必须运行于 DataNode 上，即 DataNode 既是数据存储结点，也是计算结点，这样可以减少数据在网络上的传输，降低对网络带宽的需求。 JobTracker 将 Map 任务和 Reduce 任务分发给空闲的 TaskTracker, 让这些任务并行运行，并负责监控任务的运行情况。如果某一个 TaskTracker 出故障了，JobTracker 会将其负责的任务转交给另一个空闲的 TaskTracker 重新运行。

MapReduce 基本编程

概述

一个简单的 MapReduce 程序需要三样东西

1. 实现 Mapper，处理输入的<k,v>对，输出中间结果
2. 实现 Reduce，对中间结果进行运算，输出最终结果
3. 在 main 方法里定义运行作业，定义一个 job，在这里控制 job 如何运行等。

实践

创建一个 maven 工程

选用 maven 来管理工程，用自己喜爱的 m2eclipse 插件在 eclipse 里创建或在命令行里创建。

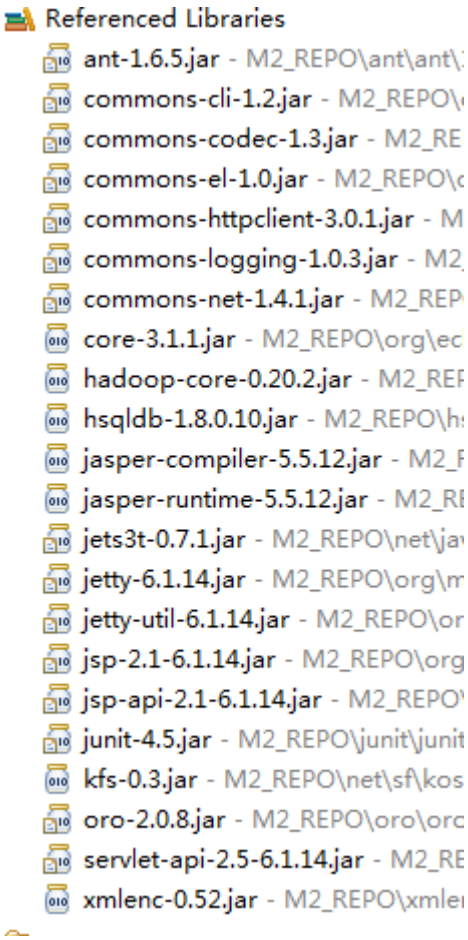
加入 hadoop 依赖

代码清单1.

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-core</artifactId>
    <version>0.20.2</version>
</dependency>

    <repositories>
        <repository>
            <id>cloudera</id>
            <url>https://repository.cloudera.com/content/groups/public</url>
        </repository>
    </repositories>
</project>
```

在 pom.xml 里加入以上内容，运行 mvn eclipse:eclipse 命令后，将工程导入 eclipse，可以看到以下相关的依赖。



Ok，现在开始我们第一个 MapReduce 程序，用这个程序实现字数统计功能。

编写 Map 类

代码清单2.

```
public   class WordCountMapper
         extends Mapper<Object, Text, Text, IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }
```

Mapper 接口是一个泛型，有 4 个形式的参数类型，分别指定 map 函数的输入键，输入值，输出键，输出值。就上面的示例来说，输入键没有用到（实际代表行在文本中格的位置，没有这方面的需要，所以忽略），输入值是一样文本，输出键为单词，输出值为整数代表单

词出现的次数。需要注意的是 Hadoop 规定了自己的一套可用于网络序列优化的基本类型，而不是使用内置的 java 类型，这些都在 org.apache.hadoop.io 包中定义，上面使用的 Text 类型相当于 java 的 String 类型，IntWritable 类型相当于 java 的 Integer 类型。除此之外，看不到任何分布式编程的细节，一切都是那么的简单。

编写 Reduce 类

代码清单3.

```
public class WordCountReducer extends
        Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

同样，Reducer 接口的四个形式参数类型指定了 reduce 函数的输入和输出类型。在上面的例子中，输入键是单词，输入值是单词出现的次数，将单词出现的次数进行叠加，输出单词和单词总数。

定义 job

代码清单4.

```
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);
        }
        Job job = new Job(conf, "word count");
        job.setJarByClass(WordCount.class);
```

这个名字是自定义的

当在 hadoop 集群上运行作业时，需要把代码打包成一个 jar 文件（hadoop 会在集群分发这个文件），通过 job 的 setJarByClass 设置一个类，hadoop 根据这个类找到所在的 jar 文件

```
        job.setMapperClass(WordCountMapper.class);
        job.setCombinerClass(WordCountReducer.class);
        job.setReducerClass(WordCountReducer.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);
    }
}
```

设置要使用的 map、combiner、reduce 类型

设置 map 和 reduce 函数的输出键和输出值类型

设置输入和输出路径

提交作业并等待它完成

在上面的示例中我们设置输入类型，是因为我们使用默认的 TextInputFormat，针对文本文件，按行将文本文件切割成 InputSplits，并用 LineRecordReader 将 InputSplit 解析成 <key,value> 对，key 是行在文件中的位置，value 是文件中的一行。

基本上完成一个 MapReduce 程序就这么简单，但是如何让自己的程序在集群环境中运行最优，还需要掌握 Hadoop 的其他知识。

MapReduce 测试

用 MRUnit 做单元测试

MRUnit 是一款由 Couldera 公司开发的专门针对 Hadoop 中编写 MapReduce 单元测试的框架。可以用 MapDriver 单独测试 Map，用 ReduceDriver 单独测试 Reduce，用 MapReduceDriver 测试 MapReduce 作业。

加入 mrunit 依赖

代码清单5.

```
<dependency>
    <groupId>com.cloudera.hadoop</groupId>
    <artifactId>hadoop-mrunit</artifactId>
    <version>0.20.2-320</version>
    <scope>test</scope>
</dependency>
```

单独测试 Map

代码清单6.

```
public class WordCountMapperTest {

	private Mapper mapper;
	private MapDriver driver;

	@Before
	public void init(){
		mapper = new WordCountMapper();
		driver = new MapDriver(mapper);
	}

	@Test
	public void test() throws IOException{
		String line = "Taobao is a great website";
		driver.withInput(null,new Text(line))
				.withOutput(new Text("Taobao"),new IntWritable(1))
				.withOutput(new Text("is"), new IntWritable(1))
				.withOutput(new Text("a"), new IntWritable(1))
				.withOutput(new Text("great"), new IntWritable(1))
				.withOutput(new Text("website"), new IntWritable(1))
				.runTest();
	}
}
```

上面的例子通过MapDriver的withInput和withOutput组织map函数的输入键值和期待的输出键值，通过runTest方法运行作业，测试代码清单 2实现的Map函数。 测试运行通过。

单独测试 Reduce

代码清单7.

```
public class WordCountReducerTest {

	private Reducer reducer;
	private ReduceDriver driver;

	@Before
	public void init(){
	  reducer = new WordCountReducer();
	  driver = new ReduceDriver(reducer);
	}
	@Test
```

```
    public void test() throws IOException{
        String key = "taobao";
        List values = new ArrayList();
        values.add(new IntWritable(2));
        values.add(new IntWritable(3));

        driver.withInput(new Text("taobao"), values)
        .withOutput(new Text("taobao"), new IntWritable(5))
        .runTest();
    }
}
```

上面的例子的测试Map函数的写法类似，测试代码清单 3 中的reduce函数，因为reduce函数实现相加功能，因此我们假设输入为<taobao,[2,3]>，则期待结果为<taobao,5>.测试运行通过。

测试 MapReduce

代码清单8.

```
public class WordCountTest {
    private Mapper mapper;
    private Reducer reducer;
    private MapReduceDriver driver;

    @Before
    public void init(){
        mapper = new WordCountMapper();
        reducer = new WordCountReducer();
        driver = new MapReduceDriver(mapper,reducer);
    }

    @Test
    public void test() throws RuntimeException, IOException{
        String line = "Taobao is a great website, is it not?";
        driver.withInput("",new Text(line))
        .withOutput(new Text("Taobao"),new IntWritable(1))
        .withOutput(new Text("a"),new IntWritable(1))
        .withOutput(new Text("great"),new IntWritable(1))
        .withOutput(new Text("is"),new IntWritable(2))
        .withOutput(new Text("it"),new IntWritable(1))
        .withOutput(new Text("not"),new IntWritable(1))
        .withOutput(new Text("website"),new IntWritable(1))
        .runTest();
    }
```

```
}
```

这次我们测试MapReduce的作业，通过MapReduceDriver的withInput构造map函数的输入键值，通过withOutput构造reduce函数的输出键值。来测试代码清单4 ，这次运行测试时抛出了异常，测试没有通过但没有详细junit异常信息，在控制台显示：

```
2010-11-5 11:14:08 org.apache.hadoop.mrunit.TestDriver
lookupExpectedValue
严重: Received unexpected output (not?, 1)
2010-11-5 11:14:08 org.apache.hadoop.mrunit.TestDriver
lookupExpectedValue
严重: Received unexpected output (website,, 1)
2010-11-5 11:14:08 org.apache.hadoop.mrunit.TestDriver validate
严重: Missing expected output (not, 1) at position 5
2010-11-5 11:14:08 org.apache.hadoop.mrunit.TestDriver validate
严重: Missing expected output (website, 1) at position 6
```

看样子是那里出了问题，不过看控制台日志不是很直观，因此我们修改测试代码，不调用 runTest 方法，而是调用 run 方法获取输出结果，再跟期待结果相比较，mrunit 提供了 org.apache.hadoop.mrunit.testutil.ExtendedAssert.assertListEquals 辅助类来断言输出结果。修改后的代码如下。

代码清单9.

```
@Test
    public void test() throws RuntimeException, IOException{
        String line = "Taobao is a great website, is it not?";
        List<Pair<Text,IntWritable>> out = null;

        out = driver.withInput("",new Text(line)).run();

        List<Pair<Text,IntWritable>> expected = new ArrayList<Pair<Text,IntWritable>>();
        expected.add(new Pair(new Text("Taobao"),new IntWritable(1)));
        expected.add(new Pair(new Text("a"),new IntWritable(1)));
        expected.add(new Pair(new Text("great"),new IntWritable(1)));
        expected.add(new Pair(new Text("is"),new IntWritable(2)));
        expected.add(new Pair(new Text("it"),new IntWritable(1)));
        expected.add(new Pair(new Text("not"),new IntWritable(1)));
        expected.add(new Pair(new Text("website"),new IntWritable(1)));

        org.apache.hadoop.mrunit.testutil.ExtendedAssert.assertListEquals(expected, out);
    }
```

再次运行，测试不通过，但有了明确的断言信息，

java.lang.AssertionError: Expected element (not, 1) at index 5 != actual element (not?, 1)

断言显示实际输出的结果为not?不是我们期待的not，为什么?检查代码清单 2发现程序以空格为分隔符未考虑到标点符号的情况，哈哈，发现一个bug，赶紧修改吧。这个问题也反映了单元测试的重要性，想想看，如果是一个更加复杂的运算，不做单元测试直接放到分布式集群中去运行，当结果不符时就没这么容易定位出问题了。

运行 MapReduce Job 进行集成测试

通过上面的内容我们可以很容易对 MapReduce 进行单元测试，这很必要，可以较早的发现一些代码逻辑的问题。只有单元测试是不够的，我们需要对 MapReduce 任务进行集成测试，要进行集成测试，得先懂得如何将 MapReduce 作业在 hadoop 集群中运行起来。

准备工作

以 windows 环境为例：

- ✓ 安装 jdk，设置环境变量 JAVA_HOME 为 jdk 安装目录
- ✓ 安装Cygwin，安装时注意选择安装软件包openssh - *Net* 类，安装完成将cygwin/bin加入环境变量path。
- ✓ 确认 ssh。打开 cygwin 命令行，
  安装 sshd:
  $ ssh-host-config
  启动 sshd 服务：
  $ net start sshd
  检查可登录 localhost：
  $ ssh localhost
- ✓ 下载稳定的hadoop版本，点此选择 下载，我选择的是 0.20.2 版本。编辑conf/hadoop-env.sh，修改JAVA_HOME为jdk安装目录。
- ✓ 打包 MapReduce 作业。我们用 maven 的 assembly 插件来将程序打包。

代码清单10. 配置 maven assembly-plugin

```
<build>
        <plugins>
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
                <configuration>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

在 pom.xml 里加入以上配置，运行 mvn assembly:assembly 命令就可以打包了。

单机运行测试

特点：单台机器、单线程运行、不需要启动 hadoop 进程，利于调试，但没有模拟 hadoop

集群多个进程的情况，只支持一个 reducer。
运行方式：
```
$ bin/hadoop jar --config standalone  path/xx.jar WordCount input output
```
注意点：
用--config 指定单机运行时的配置目录（示例中配置目录文件夹名为 standalone，在 0.20.2 版本这个文件夹可以为空目录，不要别的配置文件）
WordCount 为包含运行作业 main 函数的的类，如果有包名需要加上包名。
Input 文件夹下放置输入的文件。
output 为输出的目录，在运行 job 前需保证该目录不存在，否则会报错。

伪分布式运行测试

特点：单台机器，启动 hadoop 所有进程（如 NameNode, DataNode, TaskTracker, JobTracker, SecondaryNameNode），较好的模拟 hadoop 集群情况。

配置：
conf/core-site.xml

```
<configuration>
 <property>
        <name>fs.default.name</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

conf/hdfs-site.xml

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

mapred-site.xml

```
<configuration>
  <property>
        <name>mapred.job.tracker</name>
        <value>localhost:9001</value>
    </property>
</configuration>
```

格式化分布式文件系统
```
$ bin/hadoop namenode –format
```

启动 hadoop 进程
```
$ bin/start-all.sh
```

检查是否启动成功，可访问以下url
```
NameNode - http://localhost:50070/
JobTracker - http://localhost:50030/
```
如果不能访问，检查logs目录下的日志分析原因。

将输入文件复制到分布式文件系统
```
$ bin/hadoop fs -put local input
```

运行
```
$ bin/hadoop jar  path/xx.jar WordCount input output
```

检查输出
```
$ bin/hadoop fs -cat output/*
```

停止hadoop进程
```
$ bin/stop-all.sh
```

完全分布式运行测试

特点：多台机器，实现 hadoop 的分布式集群，通过高仿真环境进行集成测试。
关于完全分布式运行测试环境搭建可见Cluster Setup。

集成测试总结

在掌握了如何运行 hadoop 作业后，测试要做的事就是通过脚本/代码将这个过程自动化起来，一般流程是：
预设置（准备输入文件、启动 hadoop 进程等）->运行作业->输出结果跟预期结果的对比->报告导致失败的原因。

在运行集成测试时需要考虑几个问题：

集成环境的搭建：需要考虑机器资源，维护成本。

输入构造：在单元测试时我们可以很容易的构造一些小的键值对，其输出结果可以很好的预期，但在集成测试时小文件意义已经不大了，我们需要仿真的大批量的数据来发现程序的问题，仿真度越高，发现问题的可能性越大。

输出分析： 我们面对的输入是仿真的海量数据，不可能做输出结果的精确预期，需要借助日志或对输出进行二次分析。在开发时需要考虑这些情况，将有用信息通过日志或输出的方

式存储。在完全分布式模式运行，日志散落在各台机器上，如何有效获取这些日志集中起来做分析?这个我们可以借助Scribe工具。同样，输出结果也可能为海量数据，如何高效对此进行分析，这可能需要针对输出数据编写测试的MapReduce任务来分析结果。

使用 Eclipse 插件

在开发调试过程中，需要将程序打包，运行任务后通过命令或 web 界面查看运行输出及 job 运行情况，这个比较繁琐，下面介绍的 eclipse 插件可以简化这个过程，方便调试。

插件安装

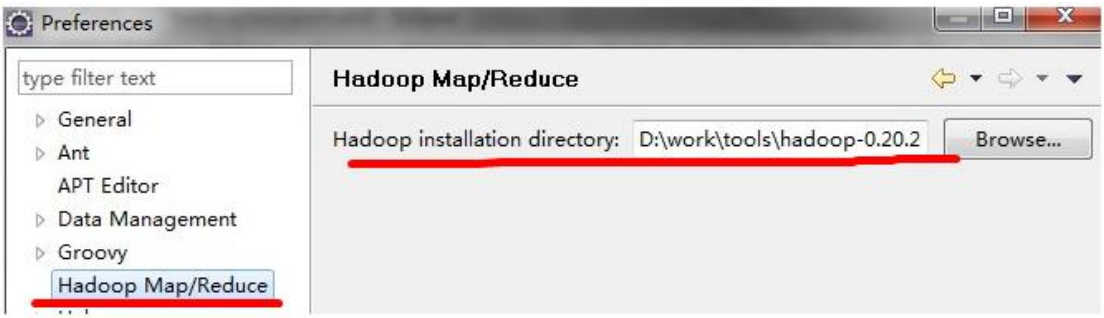
Hadoop 的 eclipse plugin 跟 hadoop 发行版一起分发，到 hadoop 安装目录\contrib\eclipse-plugin 下可以找到该插件。在试用过程中发现不支持 eclipse 3.5，因此要在 eclipse3.5 及以上版本运行，需要做个修改。修改方法如下：
编辑：src\contrib\eclipse-plugin\src\java\org\apache\hadoop\eclipse\launch\HadoopApplicationLaunchShortcut.java
做如下修改：
//import org.eclipse.jdt.internal.debug.ui.launcher.JavaApplicationLaunchShortcut;
import org.eclipse.jdt.debug.ui.launchConfigurations.JavaApplicationLaunchShortcut;

修改完毕后在 hadoop 目录执行 ant package 重新打包 eclipse 插件。

将修改后的插件拷贝到 eclipse\dropins\hadoop\plugins 目录下完成安装。

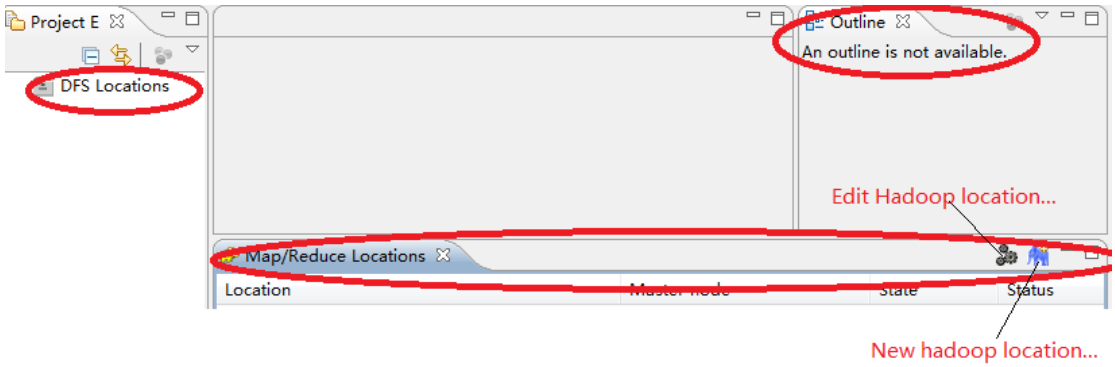
插件配置与使用

指定 Hadoop 安装目录

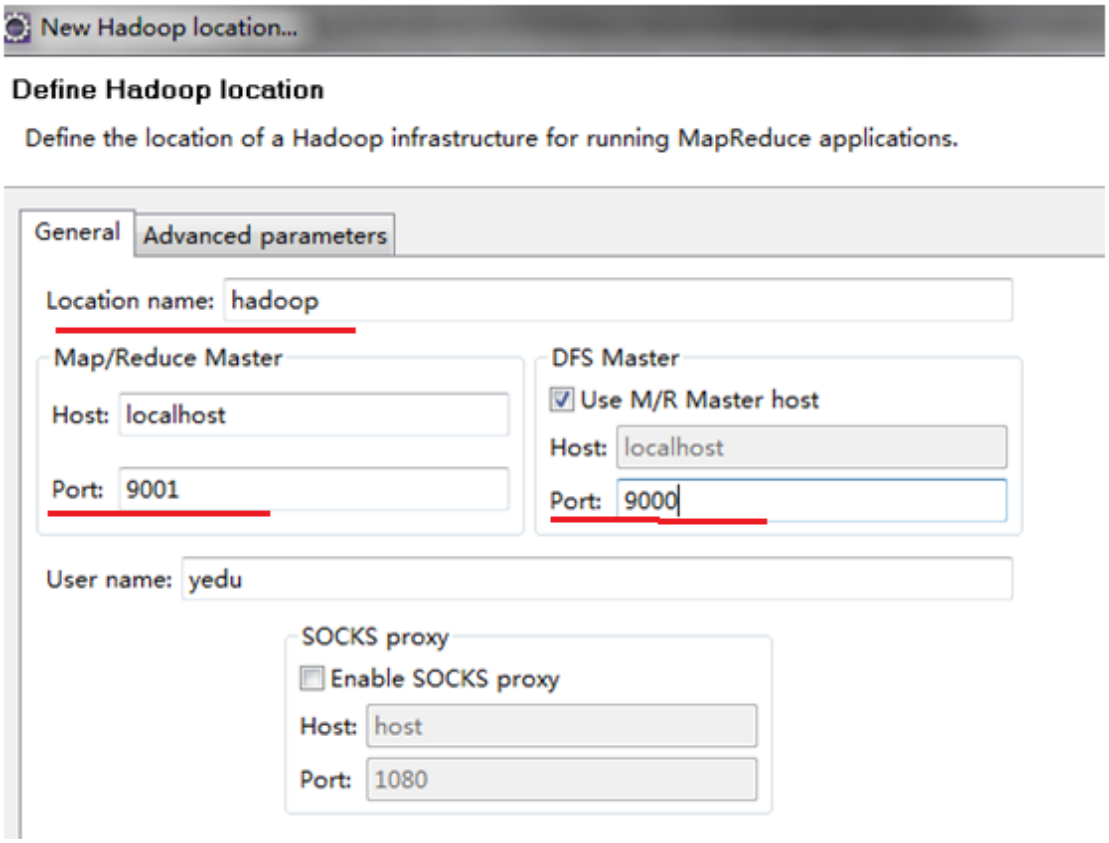


打开 Map/Reduce 视图。

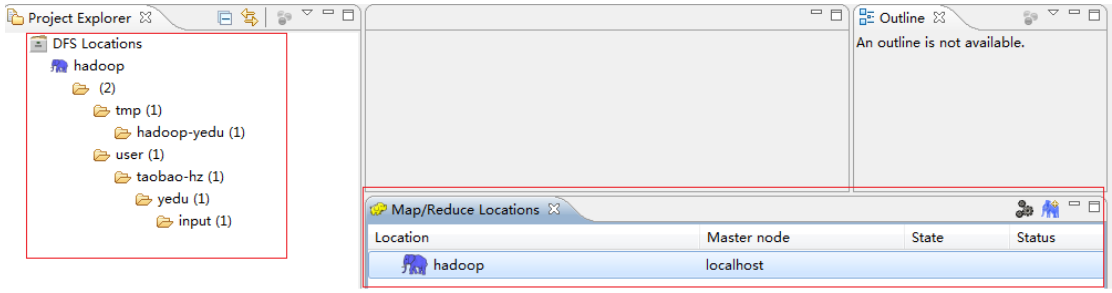
”Window”->”Open Perspective”->”Other”->“Map/Reduce”.
“Window”->”Show views”->”Other”->”Map Reduce Tools”->”Map/Reduce locations”.



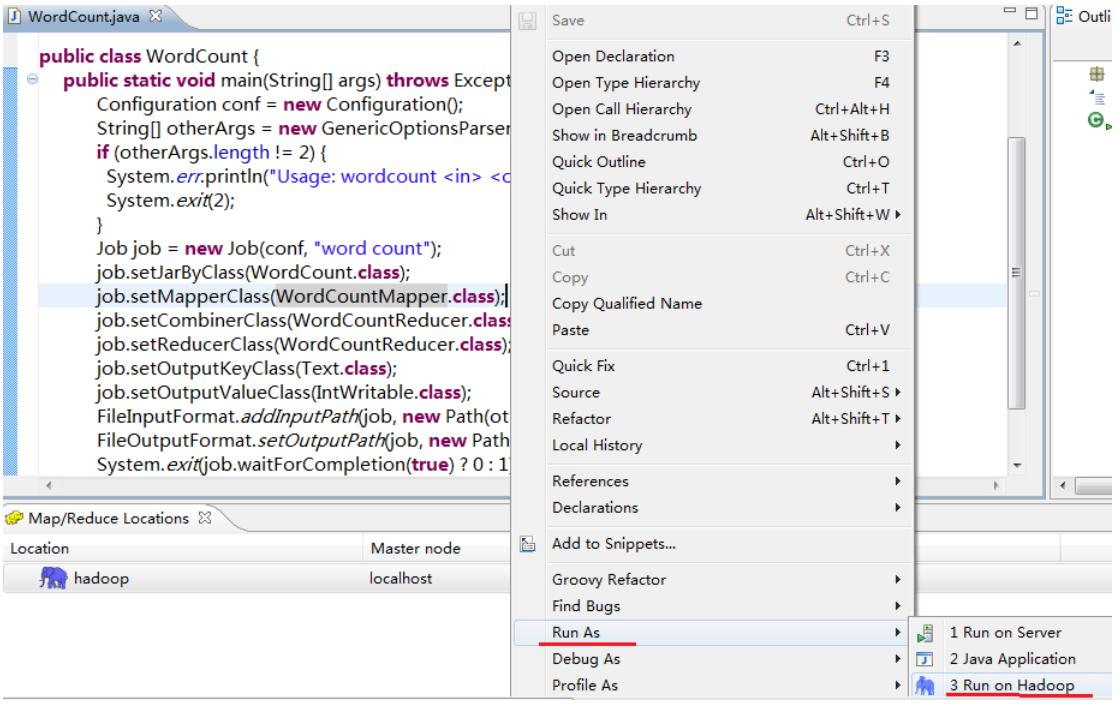
新建 Hadoop location



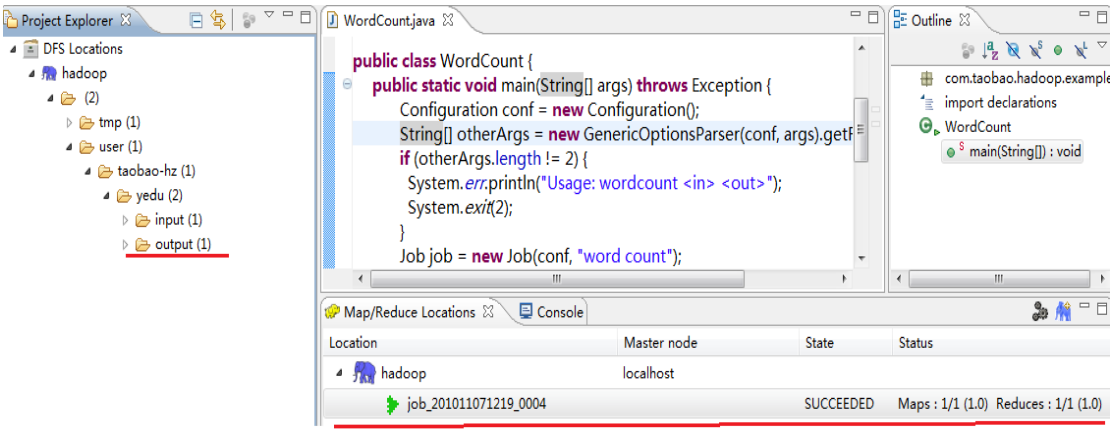
DFS 视图浏览文件



运行 MapReduce 作业



查看运行结果


Project Explorer
DFS Locations
hadoop
(2)
tmp (1)
user (1)
taobao-hz (1)
yedu (2)
input (1)
output (1)
WordCount.java
public class WordCount {
public static void main(String[] args) throws Exception {
Configuration conf = new Configuration();
String[] otherArgs = new GenericOptionsParser(conf, args).getR
if (otherArgs.length != 2) {
System.err.println("Usage: wordcount <in> <out>");
System.exit(2);
}
Job job = new Job(conf, "word count");
Outline
com.taobao.hadoop.example
import declarations
WordCount
main(String[]) : void
Map/Reduce Locations
Console
Location
Master node
State
Status
hadoop
localhost
job_201011071219_0004
SUCCEEDED
Maps : 1/1 (1.0)
Reduces : 1/1 (1.0)