toc: true title: 《从1到100深入学习Flink》——分析 Batch WordCount 程序的执行过程 tags:

- Flink
- 大数据
- 流式计算

前言

在前面两篇文章中我们分析了 Standalone 模式下 JobManager 和 TaskManager 的启动流程源码，这篇文章我们来分析一下批处理时 WordCount 程序的执行过程。

我们先来看下 WordCount 源码，地址 https://github.com/zhisheng17/flink-learning/blob/master/flink-learning-examples/src/main/java/com/zhisheng/examples/batch/wordcount/Main.java

```
package com.zhisheng.examples.wordcount.batch.wordcount;

/**
 * blog：http://www.54tianzhisheng.cn/
 * 微信公众号：zhisheng
 */

public class Main {

    public static void main(String[] args) throws Exception {
        //批处理使用 ExecutionEnvironment
        final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();

env.getConfig().setGlobalJobParameters(ParameterTool.fromArgs(args)
);

        env.fromElements(WORDS)
        .flatMap(new FlatMapFunction<String, Tuple2<String,
Integer>>() {
            @Override
            public void flatMap(String value,
Collector<Tuple2<String, Integer>> out) throws Exception {
                String[] splits =

value.toLowerCase().split("\\W+");

                for (String split : splits) {
                    if (split.length() > 0) {
                        out.collect(new Tuple2<>(split, 1));
                    }
                }
            }
        })
        .groupBy(0)
        .reduce(new ReduceFunction<Tuple2<String, Integer>>() {
            @Override
            public Tuple2<String, Integer> reduce(Tuple2<String,
Integer> value1, Tuple2<String, Integer> value2) throws Exception {
                return new Tuple2<>(value1.f0, value1.f1 +
value1.f1);
            }
        })
        .print();
    }

    private static final String[] WORDS = new String[]{"具体的 words
看上面 GitHub 源码"};
}
```

构建数据源

在 Flink 中，数据源的构建是通过 ExecutionEnvironment 的具体实现的实例来构建的，如上述代码中的这句代码：

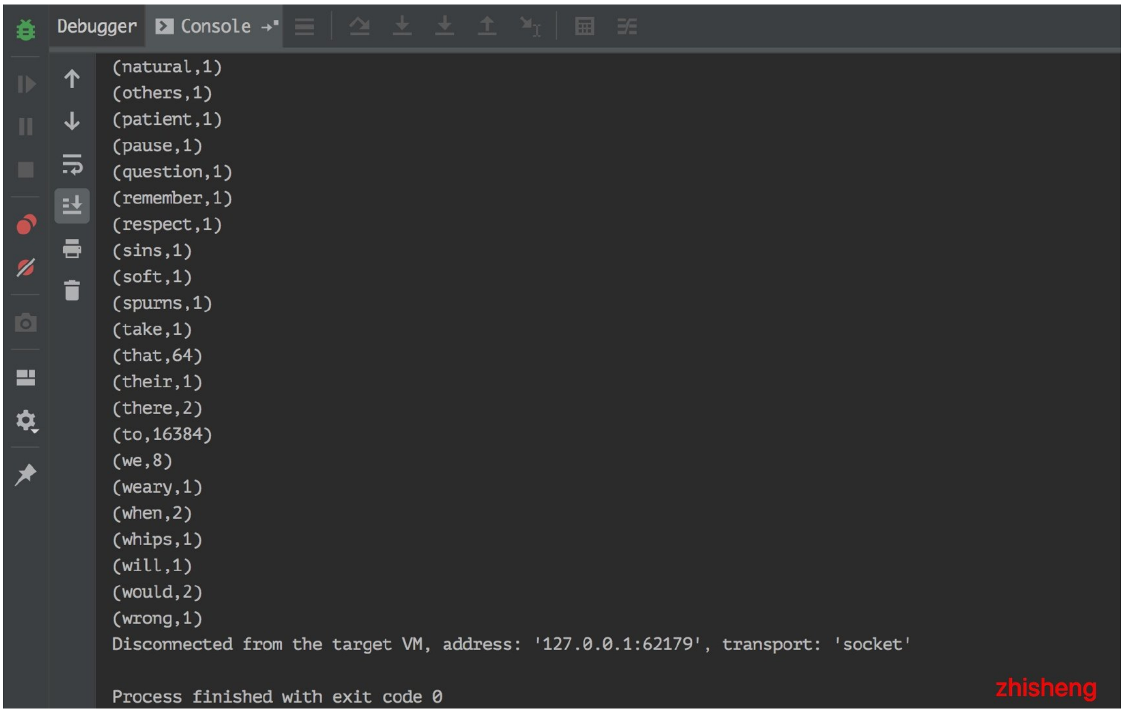
```
final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();

//这里的 WORDS 是个字符串数组
env.fromElements(WORDS);
```

这个 fromElements 根据给定的字符串数组元素来创建数据集：

```
public final <X> DataSource<X> fromElements(X... data) {

    if (data == null) {
        throw new IllegalArgumentException("The data must not be
null.");
    }
    if (data.length == 0) {
        throw new IllegalArgumentException("The number of elements
must not be zero.");
    }

    TypeInformation<X> typeInfo;
    try {
        typeInfo = TypeExtractor.getForObject(data[0]);
    }
    catch (Exception e) {
        throw new RuntimeException("Could not create
TypeInformation for type " + data[0].getClass().getName()
                + "; please specify the TypeInformation manually
via "
                + "ExecutionEnvironment#fromElements(Collection,
TypeInformation)", e);
    }

    return fromCollection(Arrays.asList(data), typeInfo,
Utils.getCallLocationName());
}
```

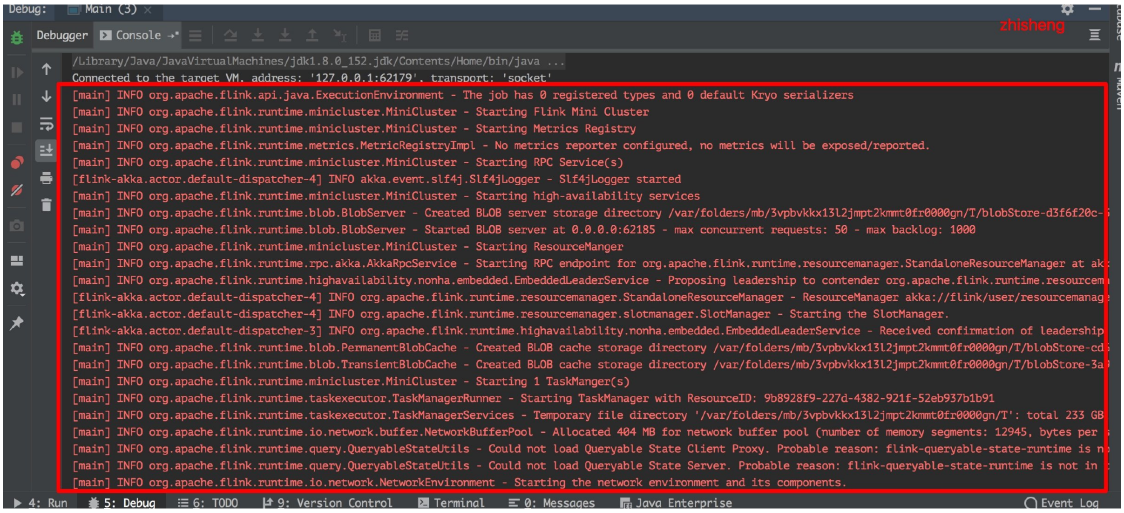
然后在 flatMap 中根据正则表达式来分隔所有的字符串，groupBy 中对每个 word 进行分组，然后在 reduce 中进行单个 word 计数操作，最后进行 print 操作，将结果打印出来。

运行程序就可以看到结果出来了：



```
(natural,1)
(others,1)
(patient,1)
(pause,1)
(question,1)
(remember,1)
(respect,1)
(sins,1)
(soft,1)
(spurns,1)
(take,1)
(that,64)
(their,1)
(there,2)
(to,16384)
(we,8)
(weary,1)
(when,2)
(whips,1)
(will,1)
(would,2)
(wrong,1)
Disconnected from the target VM, address: '127.0.0.1:62179', transport: 'socket'

Process finished with exit code 0
```

但是在结果的前面我们可以看到打印了很多的日志：



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/bin/java ...
Connected to the target VM, address: '127.0.0.1:62179', transport: 'socket'
[main] INFO org.apache.flink.api.java.ExecutionEnvironment - The job has 0 registered types and 0 default Kryo serializers
[main] INFO org.apache.flink.runtime.minicluster.MiniCluster - Starting Flink Mini Cluster
[main] INFO org.apache.flink.runtime.minicluster.MiniCluster - Starting Metrics Registry
[main] INFO org.apache.flink.runtime.metrics.MetricRegistryImpl - No metrics reporter configured, no metrics will be exposed/reported.
[main] INFO org.apache.flink.runtime.minicluster.MiniCluster - Starting RPC Service(s)
[flink-akka.actor.default-dispatcher-4] INFO akka.event.slf4j.Slf4jLogger - Slf4jLogger started
[main] INFO org.apache.flink.runtime.minicluster.MiniCluster - Starting high-availability services
[main] INFO org.apache.flink.runtime.blob.BlobServer - Created BLOB server storage directory /var/folders/mb/3vpbvkkx13l2jmpt2kmmt0fr0000gn/T/blobStore-d3f6f20c-
[main] INFO org.apache.flink.runtime.blob.BlobServer - Started BLOB server at 0.0.0.0:62185 - max concurrent requests: 50 - max backlog: 1000
[main] INFO org.apache.flink.runtime.minicluster.MiniCluster - Starting ResourceManger
[main] INFO org.apache.flink.runtime.rpc.akka.AkkaRpcService - Starting RPC endpoint for org.apache.flink.runtime.resourcemanager.StandaloneResourceManager at ak
[main] INFO org.apache.flink.runtime.highavailability.nonha.embedded.EmbeddedLeaderService - Proposing leadership to contender org.apache.flink.runtime.resourcem
[flink-akka.actor.default-dispatcher-4] INFO org.apache.flink.runtime.resourcemanager.StandaloneResourceManager - ResourceManager akka://flink/user/resourcemanag
[flink-akka.actor.default-dispatcher-4] INFO org.apache.flink.runtime.resourcemanager.slotmanager.SlotManager - Starting the SlotManager.
[flink-akka.actor.default-dispatcher-3] INFO org.apache.flink.runtime.highavailability.nonha.embedded.EmbeddedLeaderService - Received confirmation of leadership
[main] INFO org.apache.flink.runtime.blob.PermanentBlobCache - Created BLOB cache storage directory /var/folders/mb/3vpbvkkx13l2jmpt2kmmt0fr0000gn/T/blobStore-cd
[main] INFO org.apache.flink.runtime.blob.TransientBlobCache - Created BLOB cache storage directory /var/folders/mb/3vpbvkkx13l2jmpt2kmmt0fr0000gn/T/blobStore-3a
[main] INFO org.apache.flink.runtime.minicluster.MiniCluster - Starting 1 TaskManger(s)
[main] INFO org.apache.flink.runtime.taskexecutor.TaskManagerRunner - Starting TaskManager with ResourceID: 9b8928f9-227d-4382-921f-52eb937b1b91
[main] INFO org.apache.flink.runtime.taskexecutor.TaskManagerServices - Temporary file directory '/var/folders/mb/3vpbvkkx13l2jmpt2kmmt0fr0000gn/T': total 233 GB
[main] INFO org.apache.flink.runtime.io.network.buffer.NetworkBufferPool - Allocated 404 MB for network buffer pool (number of memory segments: 12945, bytes per
[main] INFO org.apache.flink.runtime.query.QueryableStateUtils - Could not load Queryable State Client Proxy. Probable reason: flink-queryable-state-runtime is n
[main] INFO org.apache.flink.runtime.query.QueryableStateUtils - Could not load Queryable State Server. Probable reason: flink-queryable-state-runtime is not in
[main] INFO org.apache.flink.runtime.io.network.NetworkEnvironment - Starting the network environment and its components.
```

初看这份代码其实发现自己没打印这些日志，但是为啥这么多日志出来呢？把这段代码的所有方法的源码都跟了一遍，发现关键点还是在 print 方法中。

print 方法

so，我们跟进去查看下：

```
public void print() throws Exception {
    List<T> elements = collect();
    for (T e: elements) {
        System.out.println(e);
    }
}
```

里面主要的就是 collect 方法将所有的数据集合后然后循环打印出来，接着我们查看下 collect 方法：

```
public List<T> collect() throws Exception {
    // jobID
    final String id = new AbstractID().toString();
    final TypeSerializer<T> serializer =
getType().createSerializer(getExecutionEnvironment().getConfig());

    this.output(new Utils.CollectHelper<>(id,
serializer)).name("collect()");
    //
    JobExecutionResult res = getExecutionEnvironment().execute();

    ArrayList<byte[]> accResult = res.getAccumulatorResult(id);
    if (accResult != null) {
        try {
            return
SerializedListAccumulator.deserializeList(accResult, serializer);
        } catch (ClassNotFoundException e) {
            throw new RuntimeException("Cannot find type class of
collected data type.", e);
        } catch (IOException e) {
            throw new RuntimeException("Serialization error while
deserializing collected data", e);
        }
    } else {
        throw new RuntimeException("The call to collect() could not
retrieve the DataSet.");
    }
}
```

在这个方法里面可以看到

```
getExecutionEnvironment().execute()
```

这个和我们一般 job 里面的是一样的效果。

```
env.execute("flink learning project template");
```

注意：如果是批处理的话，使用了 print 方法那么就不能再使用 env.execute() 了，否则 在输出结果后会报错：

```
Exception in thread "main" java.lang.RuntimeException: No new data
sinks have been defined since the last execution. The last
execution refers to the latest call to 'execute()', 'count()',
'collect()', or 'print()'.
    at
org.apache.flink.api.java.ExecutionEnvironment.createProgramPlan(Ex
ecutionEnvironment.java:940)
    at
org.apache.flink.api.java.ExecutionEnvironment.createProgramPlan(Ex
ecutionEnvironment.java:922)
    at
org.apache.flink.api.java.LocalEnvironment.execute(LocalEnvironment
.java:85)
    at
org.apache.flink.api.java.ExecutionEnvironment.execute(ExecutionEnv
ironment.java:816)
    at com.zhisheng.examples.wordcount.Main.main(Main.java:41)
```

execute 方法

那么我们跟进这个 execute 方法里面查看下源码：

```
public JobExecutionResult execute() throws Exception {
    return execute(getDefaultName());
}

public abstract JobExecutionResult execute(String jobName) throws
Exception;
```

上面这个 execute 方法是类 ExecutionEnvironment 中的一个抽象方法，这个方法在 LocalEnvironment 中有自己的实现：

```
public JobExecutionResult execute(String jobName) throws Exception
{
    if (executor == null) {
        startNewSession();
    }
    //创建程序的 Plan
    Plan p = createProgramPlan(jobName);

    //LocalExecutor 执行 Plan
    JobExecutionResult result = executor.executePlan(p);

    this.lastJobExecutionResult = result;
    return result;
}
```

createProgramPlan

这里 createProgramPlan 创建程序的 Plan：

```
public Plan createProgramPlan(String jobName, boolean clearSinks) {

    ...

    OperatorTranslation translator = new OperatorTranslation();
    Plan plan = translator.translateToPlan(this.sinks, jobName);

    if (getParallelism() > 0) {
        plan.setDefaultParallelism(getParallelism());
    }
    plan.setExecutionConfig(getConfig());

    ...

    return plan;
}
```

执行 OperatorTranslation 类中的 translateToPlan 方法：

```
public Plan translateToPlan(List<DataSink<?>> sinks, String
jobName) {
    List<GenericDataSinkBase<?>> planSinks = new ArrayList<>();

    for (DataSink<?> sink : sinks) {
        planSinks.add(translate(sink));
    }

    Plan p = new Plan(planSinks);
    p.setJobName(jobName);
    return p;
}
```

executePlan

然后跟进 LocalExecutor 类中的 executePlan 方法：

```
public JobExecutionResult executePlan(Plan plan) throws Exception {
    ...

    synchronized (this.lock) {

        // check if we start a session dedicated for this execution
        final boolean shutDownAtEnd;

        if (jobExecutorService == null) {
            shutDownAtEnd = true;

            // 配置与本地 plan 并行度相等的本地 slot 数量
            if (this.taskManagerNumSlots ==
DEFAULT_TASK_MANAGER_NUM_SLOTS) {
                int maxParallelism = plan.getMaximumParallelism();
                if (maxParallelism > 0) {
                    this.taskManagerNumSlots = maxParallelism;
                }
            }

            // 启动本地集群
            start();
        }
        else {
            // we use the existing session
            shutDownAtEnd = false;
        }
```

```java
        try {
            // TODO: Set job's default parallelism to max number of slots
            final int slotsPerTaskManager = jobExecutorServiceConfiguration.getInteger(TaskManagerOptions.NUM_TASK_SLOTS, taskManagerNumSlots);
            final int numTaskManagers = jobExecutorServiceConfiguration.getInteger(ConfigConstants.LOCAL_NUMBER_TASK_MANAGER, 1);
            plan.setDefaultParallelism(slotsPerTaskManager * numTaskManagers);

            Optimizer pc = new Optimizer(new DataStatistics(), jobExecutorServiceConfiguration);
            OptimizedPlan op = pc.compile(plan);

            JobGraphGenerator jgg = new JobGraphGenerator(jobExecutorServiceConfiguration);
            JobGraph jobGraph = jgg.compileJobGraph(op, plan.getJobId());

            return jobExecutorService.executeJobBlocking(jobGraph);
        }
        finally {
            if (shutDownAtEnd) {
                stop();
            }
        }
    }
}
```

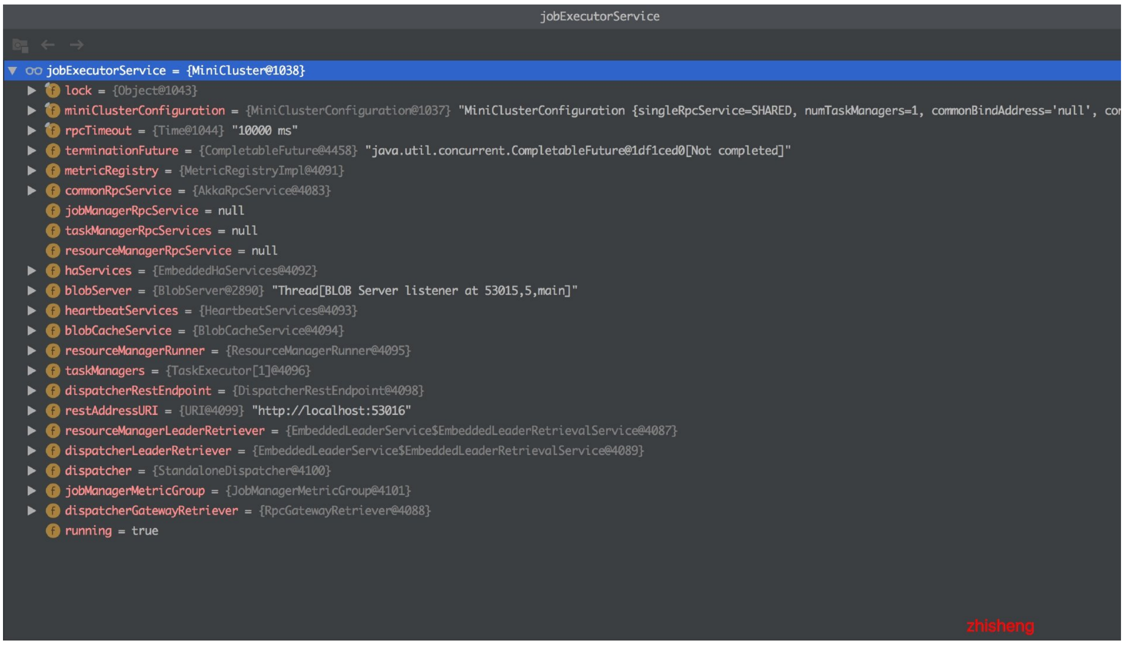
启动本地集群：

```
public void start() throws Exception {
    synchronized (lock) {
        if (jobExecutorService == null) {
            // 创建配置
            jobExecutorServiceConfiguration =
createConfiguration();

            //创建作业执行器服务
            jobExecutorService =
createJobExecutorService(jobExecutorServiceConfiguration);
        } else {
            throw new IllegalStateException("The local executor was
already started.");
        }
    }
}
```

创建作业执行器服务：

```
private JobExecutorService createJobExecutorService(Configuration
configuration) throws Exception {
    final JobExecutorService newJobExecutorService;
    if
(CoreOptions.NEW_MODE.equals(configuration.getString(CoreOptions.MO
DE))) {

        if (!configuration.contains(RestOptions.PORT)) {
            configuration.setInteger(RestOptions.PORT, 0);
        }
        //构建 MiniCluster 配置
        final MiniClusterConfiguration miniClusterConfiguration =
new MiniClusterConfiguration.Builder()
            .setConfiguration(configuration)
            .setNumTaskManagers(
                configuration.getInteger(
                    ConfigConstants.LOCAL_NUMBER_TASK_MANAGER,

ConfigConstants.DEFAULT_LOCAL_NUMBER_TASK_MANAGER))
            .setRpcServiceSharing(RpcServiceSharing.SHARED)
            .setNumSlotsPerTaskManager(

configuration.getInteger(TaskManagerOptions.NUM_TASK_SLOTS, 1))
            .build();
        //创建 MiniCluster 对象
        final MiniCluster miniCluster = new
MiniCluster(miniClusterConfiguration);
        //启动 MiniCluster
        miniCluster.start();

        configuration.setInteger(RestOptions.PORT,
miniCluster.getRestAddress().getPort());
        // miniCluster 赋值给 newJobExecutorService
        newJobExecutorService = miniCluster;
    } else {
        final LocalFlinkMiniCluster localFlinkMiniCluster = new
LocalFlinkMiniCluster(configuration, true);
        localFlinkMiniCluster.start();

        newJobExecutorService = localFlinkMiniCluster;
    }

    return newJobExecutorService;
}
```

newJobExecutorService 最后有这些服务：



```
▼ jobExecutorService = {MiniCluster@1038}
  ▶ lock = {Object@1043}
  ▶ miniClusterConfiguration = {MiniClusterConfiguration@1037} "MiniClusterConfiguration {singleRpcService=SHARED, numTaskManagers=1, commonBindAddress='null', co
  ▶ rpcTimeout = {Time@1044} "10000 ms"
  ▶ terminationFuture = {CompletableFuture@4458} "java.util.concurrent.CompletableFuture@1df1ced0[Not completed]"
  ▶ metricRegistry = {MetricRegistryImpl@4091}
  ▶ commonRpcService = {AkkaRpcService@4083}
    jobManagerRpcService = null
    taskManagerRpcServices = null
    resourceManagerRpcService = null
  ▶ haServices = {EmbeddedHaServices@4092}
  ▶ blobServer = {BlobServer@2890} "Thread[BLOB Server listener at 53015,5,main]"
  ▶ heartbeatServices = {HeartbeatServices@4093}
  ▶ blobCacheService = {BlobCacheService@4094}
  ▶ resourceManagerRunner = {ResourceManagerRunner@4095}
  ▶ taskManagers = {TaskExecutor[1]@4096}
  ▶ dispatcherRestEndpoint = {DispatcherRestEndpoint@4098}
  ▶ restAddressURI = {URI@4099} "http://localhost:53016"
  ▶ resourceManagerLeaderRetriever = {EmbeddedLeaderService$EmbeddedLeaderRetrievalService@4087}
  ▶ dispatcherLeaderRetriever = {EmbeddedLeaderService$EmbeddedLeaderRetrievalService@4089}
  ▶ dispatcher = {StandaloneDispatcher@4100}
  ▶ jobManagerMetricGroup = {JobManagerMetricGroup@4101}
  ▶ dispatcherGatewayRetriever = {RpcGatewayRetriever@4088}
    running = true
```

启动 MiniCluster：

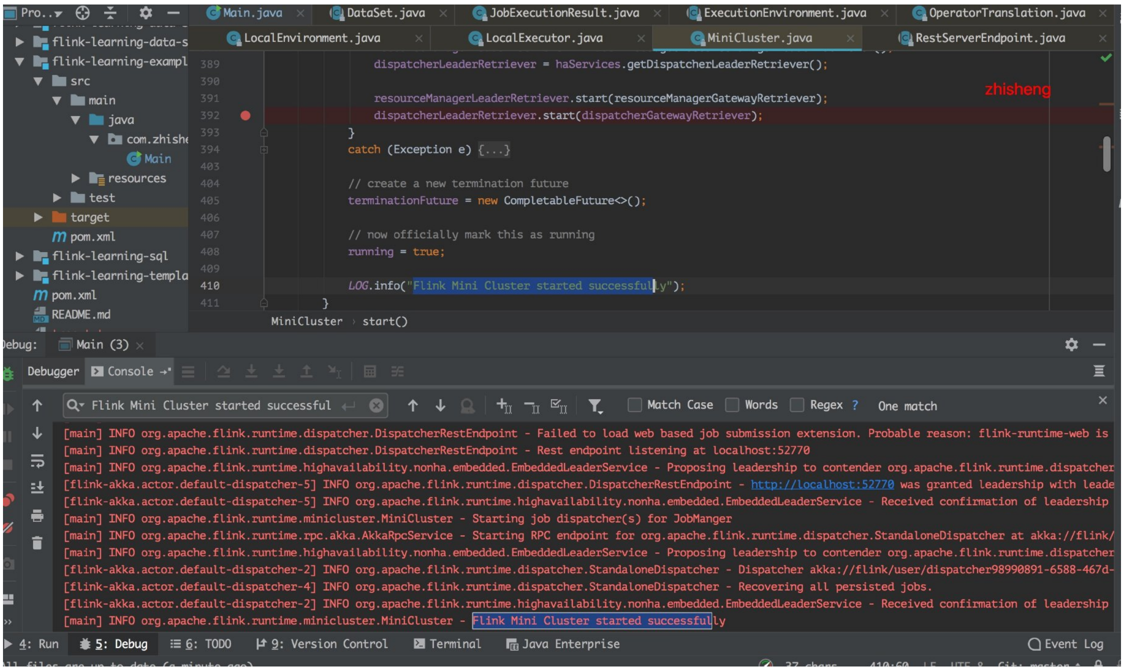
执行类 MiniCluster 中 start 方法：

```
public void start() throws Exception {
    synchronized (lock) {
        ...
        //这里准备配置

        try {
            initializeIOFormatClasses(configuration);

            //开启 Metrics Registry
            metricRegistry = createMetricRegistry(configuration);

            final RpcService jobManagerRpcService;
            final RpcService resourceManagerRpcService;
            final RpcService[] taskManagerRpcServices = new
RpcService[numTaskManagers];

            // 开启所有的 RPC 服务
            commonRpcService = createRpcService(configuration,
rpcTimeout, false, null);
            final ActorSystem actorSystem = ((AkkaRpcService)
commonRpcService).getActorSystem();
            metricRegistry.startQueryService(actorSystem, null);

            if (useSingleRpcService) {
                for (int i = 0; i < numTaskManagers; i++) {
```

```
                for (int i = 0; i < numTaskManagers; i++) {
                    taskManagerRpcServices[i] = commonRpcService;
                }

                jobManagerRpcService = commonRpcService;
                resourceManagerRpcService = commonRpcService;

                this.resourceManagerRpcService = null;
                this.jobManagerRpcService = null;
                this.taskManagerRpcServices = null;
            } else {
                // 为每个组件启动一个新服务，可能使用自定义绑定地址
                final String jobManagerBindAddress =
miniClusterConfiguration.getJobManagerBindAddress();
                final String taskManagerBindAddress =
miniClusterConfiguration.getTaskManagerBindAddress();
                final String resourceManagerBindAddress =
miniClusterConfiguration.getResourceManagerBindAddress();

                jobManagerRpcService =
createRpcService(configuration, rpcTimeout, true,
jobManagerBindAddress);
                resourceManagerRpcService =
createRpcService(configuration, rpcTimeout, true,
resourceManagerBindAddress);

                for (int i = 0; i < numTaskManagers; i++) {
                    taskManagerRpcServices[i] = createRpcService(
                            configuration, rpcTimeout, true,
taskManagerBindAddress);
                }

                this.jobManagerRpcService = jobManagerRpcService;
                this.taskManagerRpcServices =
taskManagerRpcServices;
                this.resourceManagerRpcService =
resourceManagerRpcService;
            }

            // 创建高可用服务
            LOG.info("Starting high-availability services");
            haServices =
HighAvailabilityServicesUtils.createAvailableOrEmbeddedServices(
                configuration, commonRpcService.getExecutor());
            //创建 blob 服务
            blobServer = new BlobServer(configuration,
haServices.createBlobStore());
            blobServer.start();
            //创建心跳服务
            heartbeatServices =
HeartbeatServices.fromConfiguration(configuration);


            // 开启资源管理器
            LOG.info("Starting ResourceManger");
            resourceManagerRunner = startResourceManager(
                configuration, haServices, heartbeatServices,
                metricRegistry, resourceManagerRpcService,
                new ClusterInformation("localhost",
blobServer.getPort()));

            blobCacheService = new BlobCacheService(
                configuration, haServices.createBlobStore(), new
InetSocketAddress(InetAddress.getLocalHost(), blobServer.getPort())
            );

            //启动 TaskManager(s) for the mini cluster
            LOG.info("Starting {} TaskManger(s)", numTaskManagers);
            taskManagers =
startTaskManagers(configuration,haServices,
                heartbeatServices,metricRegistry,blobCacheService,
                numTaskManagers,taskManagerRpcServices);

            // starting the dispatcher rest endpoint
            LOG.info("Starting dispatcher rest endpoint.");

            dispatcherGatewayRetriever = new RpcGatewayRetriever<>(

jobManagerRpcService,DispatcherGateway.class,DispatcherId::fromUuid
,
                20,Time.milliseconds(20L));
            final RpcGatewayRetriever<ResourceManagerId,
ResourceManagerGateway> resourceManagerGatewayRetriever = new
RpcGatewayRetriever<>(
                jobManagerRpcService,ResourceManagerGateway.class,
ResourceManagerId::fromUuid,20,Time.milliseconds(20L));

            this.dispatcherRestEndpoint = new
DispatcherRestEndpoint(

RestServerEndpointConfiguration.fromConfiguration(configuration),
                dispatcherGatewayRetriever,configuration,

RestHandlerConfiguration.fromConfiguration(configuration),

resourceManagerGatewayRetriever,blobServer.getTransientBlobService(
),
                commonRpcService.getExecutor(),
                new AkkaQueryServiceRetriever(actorSystem,

Time.milliseconds(configuration.getLong(WebOptions.TIMEOUT))),
                haServices.getWebMonitorLeaderElectionService(),
                new ShutDownFatalErrorHandler());

            dispatcherRestEndpoint.start();

            restAddressURI = new
URI(dispatcherRestEndpoint.getRestBaseUrl());

            // bring up the dispatcher that launches JobManagers
when jobs submitted
            LOG.info("Starting job dispatcher(s) for JobManger");

            this.jobManagerMetricGroup =
MetricUtils.instantiateJobManagerMetricGroup(metricRegistry,
"localhost");

            final HistoryServerArchivist historyServerArchivist =
HistoryServerArchivist.createHistoryServerArchivist(configuration,
dispatcherRestEndpoint);

            //创建 standalone 调度器
            dispatcher = new StandaloneDispatcher(
                jobManagerRpcService,Dispatcher.DISPATCHER_NAME +
UUID.randomUUID(),

configuration,haServices,resourceManagerRunner.getResourceManageGat
eway(),blobServer,heartbeatServices,jobManagerMetricGroup,
                metricRegistry.getMetricQueryServicePath(),
                new MemoryArchivedExecutionGraphStore(),
                Dispatcher.DefaultJobManagerRunnerFactory.INSTANCE,
                new ShutDownFatalErrorHandler(),
                dispatcherRestEndpoint.getRestBaseUrl(),
                historyServerArchivist,
                new IgnoreLeaderShipLostHandler());
            //启动 standalone 调度器
            dispatcher.start();

            resourceManagerLeaderRetriever =
haServices.getResourceManagerLeaderRetriever();
            dispatcherLeaderRetriever =
haServices.getDispatcherLeaderRetriever();


resourceManagerLeaderRetriever.start(resourceManagerGatewayRetrieve
r);
```

```
dispatcherLeaderRetriever.start(dispatcherGatewayRetriever);
        }
        catch (Exception e) {

        }

        // create a new termination future
        terminationFuture = new CompletableFuture<>();

        // now officially mark this as running
        running = true;

        LOG.info("Flink Mini Cluster started successfully");
    }
}
```

和之前两篇文章我们分析的 JobManager、TaskManager 启动流程类似，将所有的服务都启动，如果你 debug 到这个方法里面你就可以看到一行一行的日志打印出来，那感觉爽歪歪。



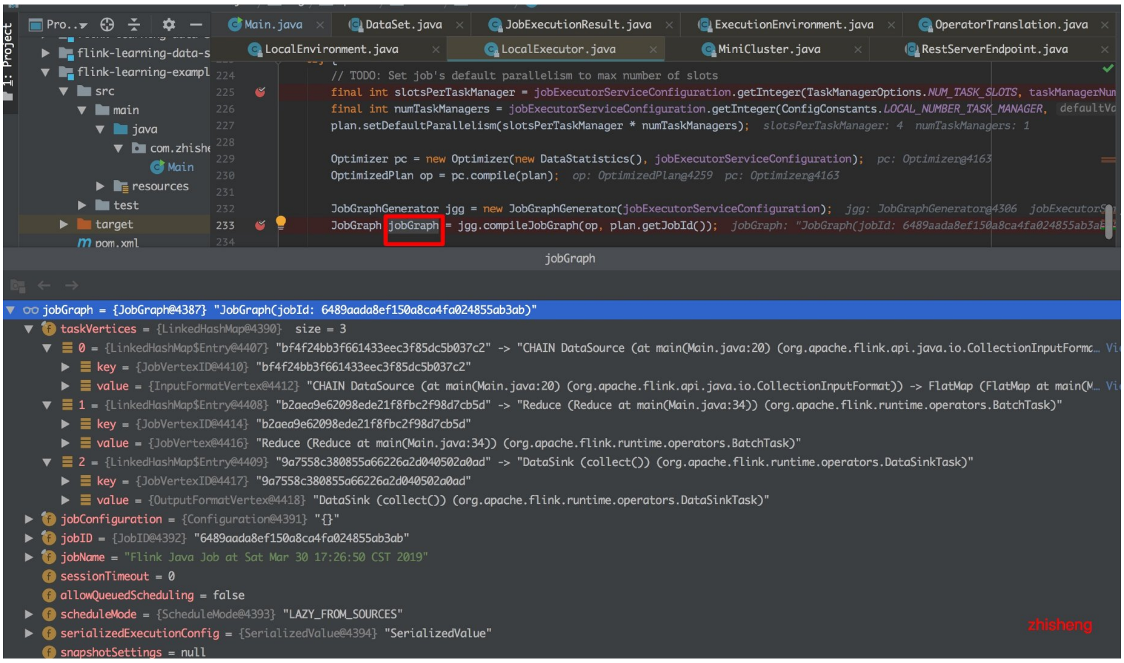
```
            dispatcherLeaderRetriever = haServices.getDispatcherLeaderRetriever();

            resourceManagerLeaderRetriever.start(resourceManagerGatewayRetriever);
            dispatcherLeaderRetriever.start(dispatcherGatewayRetriever);
        }
        catch (Exception e) {...}

        // create a new termination future
        terminationFuture = new CompletableFuture<>();

        // now officially mark this as running
        running = true;

        LOG.info("Flink Mini Cluster started successfully");
    }
```

```
[main] INFO org.apache.flink.runtime.dispatcher.DispatcherRestEndpoint - Failed to load web based job submission extension. Probable reason: flink-runtime-web is
[main] INFO org.apache.flink.runtime.dispatcher.DispatcherRestEndpoint - Rest endpoint listening at localhost:52770
[main] INFO org.apache.flink.runtime.highavailability.nonha.embedded.EmbeddedLeaderService - Proposing leadership to contender org.apache.flink.runtime.dispatcher
[flink-akka.actor.default-dispatcher-5] INFO org.apache.flink.runtime.dispatcher.DispatcherRestEndpoint - http://localhost:52770 was granted leadership with leade
[flink-akka.actor.default-dispatcher-5] INFO org.apache.flink.runtime.highavailability.nonha.embedded.EmbeddedLeaderService - Received confirmation of leadership
[main] INFO org.apache.flink.runtime.minicluster.MiniCluster - Starting job dispatcher(s) for JobManger
[main] INFO org.apache.flink.runtime.rpc.akka.AkkaRpcService - Starting RPC endpoint for org.apache.flink.runtime.dispatcher.StandaloneDispatcher at akka://flink/
[main] INFO org.apache.flink.runtime.highavailability.nonha.embedded.EmbeddedLeaderService - Proposing leadership to contender org.apache.flink.runtime.dispatcher
[flink-akka.actor.default-dispatcher-2] INFO org.apache.flink.runtime.dispatcher.StandaloneDispatcher - Dispatcher akka://flink/user/dispatcher98990891-6588-467d-
[flink-akka.actor.default-dispatcher-4] INFO org.apache.flink.runtime.dispatcher.StandaloneDispatcher - Recovering all persisted jobs.
[flink-akka.actor.default-dispatcher-2] INFO org.apache.flink.runtime.highavailability.nonha.embedded.EmbeddedLeaderService - Received confirmation of leadership
[main] INFO org.apache.flink.runtime.minicluster.MiniCluster - Flink Mini Cluster started successfully
```

run 方法执行完了之后的话，就会执行下面代码：

```
final int slotsPerTaskManager =

jobExecutorServiceConfiguration.getInteger(TaskManagerOptions.NUM_T
ASK_SLOTS, taskManagerNumSlots);

final int numTaskManagers =
jobExecutorServiceConfiguration.getInteger(ConfigConstants.LOCAL_NU
MBER_TASK_MANAGER, 1);
plan.setDefaultParallelism(slotsPerTaskManager * numTaskManagers);

Optimizer pc = new Optimizer(new DataStatistics(),
jobExecutorServiceConfiguration);
OptimizedPlan op = pc.compile(plan);

JobGraphGenerator jgg = new
JobGraphGenerator(jobExecutorServiceConfiguration);
JobGraph jobGraph = jgg.compileJobGraph(op, plan.getJobId());

return jobExecutorService.executeJobBlocking(jobGraph);
```

JobGraphGenerator 会将 job 执行计划编译生成 JobGraph。JobGraph 如下图：



```
// TODO: Set job's default parallelism to max number of slots
final int slotsPerTaskManager = jobExecutorServiceConfiguration.getInteger(TaskManagerOptions.NUM_TASK_SLOTS, taskManagerNum
final int numTaskManagers = jobExecutorServiceConfiguration.getInteger(ConfigConstants.LOCAL_NUMBER_TASK_MANAGER,  defaultVa
plan.setDefaultParallelism(slotsPerTaskManager * numTaskManagers);  slotsPerTaskManager: 4  numTaskManagers: 1

Optimizer pc = new Optimizer(new DataStatistics(), jobExecutorServiceConfiguration);  pc: Optimizer@4163
OptimizedPlan op = pc.compile(plan);  op: OptimizedPlan@4259  pc: Optimizer@4163

JobGraphGenerator jgg = new JobGraphGenerator(jobExecutorServiceConfiguration);  jgg: JobGraphGenerator@4306  jobExecutorS
JobGraph jobGraph = jgg.compileJobGraph(op, plan.getJobId());  jobGraph: "JobGraph(jobId: 6489aada8ef150a8ca4fa024855ab3a
```

```
▼ jobGraph = {JobGraph@4387} "JobGraph(jobId: 6489aada8ef150a8ca4fa024855ab3ab)"
  ▼ taskVertices = {LinkedHashMap@4390}  size = 3
    ▼ 0 = {LinkedHashMap$Entry@4407} "bf4f24bb3f661433eec3f85dc5b037c2" -> "CHAIN DataSource (at main(Main.java:20) (org.apache.flink.api.java.io.CollectionInputForma... Vi
      ▶ key = {JobVertexID@4410} "bf4f24bb3f661433eec3f85dc5b037c2"
      ▶ value = {InputFormatVertex@4412} "CHAIN DataSource (at main(Main.java:20) (org.apache.flink.api.java.io.CollectionInputFormat)) -> FlatMap (FlatMap at main(M... Vi
    ▼ 1 = {LinkedHashMap$Entry@4408} "b2aea9e62098ede21f8fbc2f98d7cb5d" -> "Reduce (Reduce at main(Main.java:34)) (org.apache.flink.runtime.operators.BatchTask)"
      ▶ key = {JobVertexID@4414} "b2aea9e62098ede21f8fbc2f98d7cb5d"
      ▶ value = {JobVertex@4416} "Reduce (Reduce at main(Main.java:34)) (org.apache.flink.runtime.operators.BatchTask)"
    ▼ 2 = {LinkedHashMap$Entry@4409} "9a7558c380855a66226a2d040502a0ad" -> "DataSink (collect()) (org.apache.flink.runtime.operators.DataSinkTask)"
      ▶ key = {JobVertexID@4417} "9a7558c380855a66226a2d040502a0ad"
      ▶ value = {OutputFormatVertex@4418} "DataSink (collect()) (org.apache.flink.runtime.operators.DataSinkTask)"
  ▶ jobConfiguration = {Configuration@4391} "{}"
  ▶ jobID = {JobID@4392} "6489aada8ef150a8ca4fa024855ab3ab"
  ▶ jobName = "Flink Java Job at Sat Mar 30 17:26:50 CST 2019"
    sessionTimeout = 0
    allowQueuedScheduling = false
  ▶ scheduleMode = {ScheduleMode@4393} "LAZY_FROM_SOURCES"
  ▶ serializedExecutionConfig = {SerializedValue@4394} "SerializedValue"
    snapshotSettings = null
```

然后将生成的 jobGraph 作为参数去执行 job 并阻塞住（此方法以阻塞模式运行作业），executeJobBlocking 该方法仅在作业成功完成或最终失败后返回。

```java
public JobExecutionResult executeJobBlocking(JobGraph job) throws
JobExecutionException, InterruptedException {
    checkNotNull(job, "job is null");
    //提交 job
    final CompletableFuture<JobSubmissionResult> submissionFuture =
submitJob(job);
    //异步执行
    final CompletableFuture<JobResult> jobResultFuture =
submissionFuture.thenCompose(
        (JobSubmissionResult ignored) ->
requestJobResult(job.getJobID()));

    final JobResult jobResult;

    try {
        //获取 job 提交的结果
        jobResult = jobResultFuture.get();
    } catch (ExecutionException e) {
        throw new JobExecutionException(job.getJobID(), "Could not
retrieve JobResult.", ExceptionUtils.stripExecutionException(e));
    }

    try {
        //
        return
jobResult.toJobExecutionResult(Thread.currentThread().getContextCla
ssLoader());
    } catch (IOException | ClassNotFoundException e) {
        throw new JobExecutionException(job.getJobID(), e);
    }
}
```

提交 job：

```
public CompletableFuture<JobSubmissionResult> submitJob(JobGraph
jobGraph) {
    final DispatcherGateway dispatcherGateway;
    try {
        dispatcherGateway = getDispatcherGateway();
    } catch (LeaderRetrievalException | InterruptedException e) {
        ExceptionUtils.checkInterrupted(e);
        return FutureUtils.completedExceptionally(e);
    }

    // 我们必须允许以Flip-6模式进行队列调度，因为我们需要从ResourceManager请
求slot
    jobGraph.setAllowQueuedScheduling(true);

    final CompletableFuture<InetSocketAddress>
blobServerAddressFuture =
createBlobServerAddress(dispatcherGateway);

    final CompletableFuture<Void> jarUploadFuture =
uploadAndSetJobFiles(blobServerAddressFuture, jobGraph);

    //将 job 提交到调度器
    final CompletableFuture<Acknowledge>
acknowledgeCompletableFuture = jarUploadFuture.thenCompose(
        (Void ack) -> dispatcherGateway.submitJob(jobGraph,
rpcTimeout));

    return acknowledgeCompletableFuture.thenApply(
        (Acknowledge ignored) -> new
JobSubmissionResult(jobGraph.getJobID()));
}
```

调度器里提交 job：

```
public CompletableFuture<Acknowledge> submitJob(JobGraph jobGraph,
Time timeout) {
    final JobID jobId = jobGraph.getJobID();

    log.info("Submitting job {} ({}).", jobId, jobGraph.getName());
    final RunningJobsRegistry.JobSchedulingStatus
jobSchedulingStatus;

    try {
        //根据 jobID 获取到 job 的调度状态
        jobSchedulingStatus =
runningJobsRegistry.getJobSchedulingStatus(jobId);
    } catch (IOException e) {
        return FutureUtils.completedExceptionally(new
FlinkException(String.format("Failed to retrieve job scheduling
status for job %s.", jobId), e));
    }

    if (jobSchedulingStatus ==
RunningJobsRegistry.JobSchedulingStatus.DONE ||
jobManagerRunnerFutures.containsKey(jobId)) {
        return FutureUtils.completedExceptionally(
            new JobSubmissionException(jobId, String.format("Job
has already been submitted and is in state %s.",
jobSchedulingStatus)));
    } else {
        //运行 job
        final CompletableFuture<Acknowledge> persistAndRunFuture =
waitForTerminatingJobManager(jobId, jobGraph,
this::persistAndRunJob)
            .thenApply(ignored -> Acknowledge.get());

        return persistAndRunFuture.exceptionally(
            (Throwable throwable) -> {
                final Throwable strippedThrowable =
ExceptionUtils.stripCompletionException(throwable);
                log.error("Failed to submit job {}.", jobId,
strippedThrowable);
                throw new CompletionException(
                    new JobSubmissionException(jobId, "Failed to
submit job.", strippedThrowable));
            });
    }
}
```

运行 job，persistAndRunJob 方法如下：

异步提交这步需要及时 debug，如果中途等了很久可能会造成超时异常，导致获取不到结果。 提交完 job 可以获取到 job 的结果，有 SUCCEEDED、FAILED、CANCELED 等，然后将 job 提交的结果返回出去，后面都会在 finally 里面关闭所有的服务。

```
private CompletableFuture<Void> persistAndRunJob(JobGraph jobGraph)

throws Exception {
    //将 job 的 JobGraph 存储
    submittedJobGraphStore.putJobGraph(new
SubmittedJobGraph(jobGraph, null));
    //运行 job
    final CompletableFuture<Void> runJobFuture = runJob(jobGraph);
    //当抛出异常的时候清除掉存储的 JobGraph
    return
runJobFuture.whenComplete(BiConsumerWithException.unchecked((Object
ignored, Throwable throwable) -> {
        if (throwable != null) {

submittedJobGraphStore.removeJobGraph(jobGraph.getJobID());
        }
    }));
}

private CompletableFuture<Void> runJob(JobGraph jobGraph) {

Preconditions.checkState(!jobManagerRunnerFutures.containsKey(jobGr
aph.getJobID()));
    //创建 jobManager Runner，并在创建的时候启动了 jobManager Runner
    final CompletableFuture<JobManagerRunner>
jobManagerRunnerFuture = createJobManagerRunner(jobGraph);

    jobManagerRunnerFutures.put(jobGraph.getJobID(),
jobManagerRunnerFuture);

    return jobManagerRunnerFuture
        .thenApply(FunctionUtils.nullFn())
        .whenCompleteAsync(
            (ignored, throwable) -> {
                if (throwable != null) {

jobManagerRunnerFutures.remove(jobGraph.getJobID());
                }
            },
            getMainThreadExecutor());
}
```

异步提交这步需要及时 debug，如果中途等了很久可能会造成超时异常，导致获取不到结果。 提交完 job 可以获取到 job 的结果，有 SUCCEEDED、FAILED、CANCELED 等，然后将 job 提交的结果返回出去，后面都会在 finally 里面关闭所有的服务。

最后在 print 方法里面的 collector 方法里面将上面返回的结果反序列化:

```
ArrayList<byte[]> accResult = res.getAccumulatorResult(id);
if (accResult != null) {
    try {
        return SerializedListAccumulator.deserializeList(accResult,
serializer);
    } catch (ClassNotFoundException e) {
        throw new RuntimeException("Cannot find type class of
collected data type.", e);
    } catch (IOException e) {
        throw new RuntimeException("Serialization error while
deserializing collected data", e);
    }
} else {
    throw new RuntimeException("The call to collect() could not
retrieve the DataSet.");
}
```

然后才将所有的 wordcount 计数后的结果打印出来。

总结

这篇文章讲解了下 wordcount 批处理程序的整个执行流程，从中可以发现 Flink 程序都是延迟执行的，当程序的 main 方法被执行时候，所有加载数据和 transformations 算子都没有开始执行，而是每一个 operation 将会创建并且添加到程序 Plan 中, 只有当你明确的调用 execute() 方法时候，程序才会真正执行!

延迟加载好处：你可以开发复杂的程序，但是 Flink 可以可以将复杂的程序转成一个 Plan，将 Plan 作为一个整体单元执行!

其实本篇文章的重点还是在 MiniCluster 里面启动各种服务，和上两篇文章的关键处都类似。

这篇文章对于每个人来说最好还是要好好了解一下，因为 Flink 在一个 main 方法其实干了很多事情，里面值得去深入研究一下，就如同 SpringBoot 里面的 run 方法一样。不深入去看下代码，不知道它为什么封装的这么好，让你只关心你的业务代码，而不需要 care 它底层的实现，大大的提高了开发的效率。

下篇文章我们将分析 wordcount 流处理程序的整个执行流程，会有点不同的。