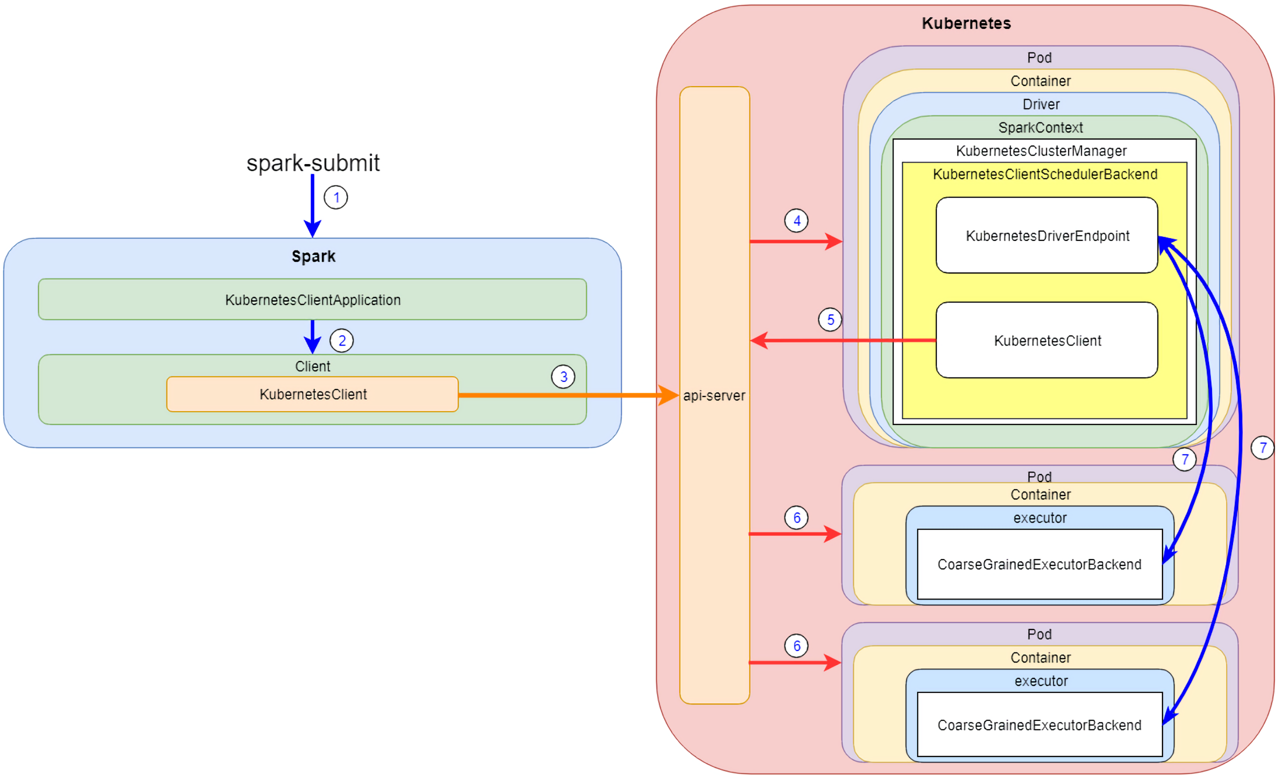
Spark on Kubernetes

Spark2.3 原生支持用 Kubernetes 调度

整体流程



如图所示，当用户使用 $SPARK_HOME/bin 目录下的 spark-submit 提交应用程序时，会启动 org.apache.spark.deploy.SparkSubmit 程序，在经过参数解析之后，SparkSubmit 的 main 方法会进入到 runMain 方法，并在这个方法中，启动 KubernetesClientApplication。

启动 KubernetesClientApplication 之后，创建一个包含能够与 Kubernetes 的 api-server 通信的 KubernetesClient 的 Client，随后，在根据 SparkConf 拼接完成用于在 Kubernetes 中启动 Driver 的 yaml 文件之后，Client 会通知 api-server 在 Kubernetes 集群中启动一个包含一个用于启动 Driver 的 Container 的 Pod。

在启动了 Pod 之后，根据 yaml 文件，启动 Driver，实际上，这个 Driver 就是用户使用 spark-submit 所提交的应用程序的 main。在 Driver 中启动 SparkContext，SparkContext 依旧是读取参数、设定参数，之后，创建 KubernetesClusterManager。KubernetesClusterManager 会去创建一个 TaskScheduler，利用这个 TaskScheduler，启动 KubernetesClientSchedulerBackend，并初始化它们。

初始化完成后，TaskScheduler 会启动，同时会启动 KubernetesClientSchedulerBackend。当 KubernetesClientSchedulerBackend 启动的时候，创建一个 KubernetesDriverEndpoint 和一个用于和 api-server 进行通讯的 KubernetesClient。

KubernetesClientSchedulerBackend 如果开启了动态分配，那么会定期去检查当前所需要的 executor 的个数，如果没有开启动态分配，那么会将所需的 executor 个数设置为 SparkConf 中所设定的数目。当检查到正在运行的包含 executor 少于所需要的 executor 的数目，那么会启动缺少的 executor 的数目个 Pod，每个 Pod 包含一个 executor，如果缺少的 executor 数目比一次能够启动的 executor 数目还要多，那么只启动一次最多能启动的数目个 executor。KubernetesClientSchedulerBackend 利用之前所创建的 KubernetesClient 通知 api-server 去启动 Pod。Pod 在启动的时候，利用 KubernetesClientSchedulerBackend 根据 SparkConf 所拼接的 yaml 文件，来创建 Pod，然后启动 CoarseGrainedExecutorBackend 以创建、管理、销毁 executor，并使 executor 与 driver 互相通信。

源码分析

spark-submit

```
if [ -z "${SPARK_HOME}" ]; then
  source "$(dirname "$0")"/find-spark-home
fi

# disable randomized hash for string in Python 3.3+
export PYTHONHASHSEED=0

exec "${SPARK_HOME}"/bin/spark-class org.apache.spark.deploy.SparkSubmit "$@"
```

spark-submit 通过 spark-class 来运行 org.apache.spark.deploy.SparkSubmit 这个程序，其参数传递给 SparkSubmit。

SparkSubmit

```
override def main(args: Array[String]): Unit = {
  // ...
  val appArgs = new SparkSubmitArguments(args)
  // ...
  appArgs.action match {
    case SparkSubmitAction.SUBMIT => submit(appArgs, uninitLog)
    case SparkSubmitAction.KILL => kill(appArgs)
    case SparkSubmitAction.REQUEST_STATUS => requestStatus(appArgs)
  }
}

private def submit(args: SparkSubmitArguments, uninitLog: Boolean): Unit = {
  val (childArgs, childClasspath, sparkConf, childMainClass) = prepareSubmitEnvironment(args)

  def doRunMain(): Unit = {
    if (args.proxyUser != null) {
      val proxyUser = UserGroupInformation.createProxyUser(args.proxyUser,
        UserGroupInformation.getCurrentUser())
      try {
        proxyUser.doAs(new PrivilegedExceptionAction[Unit]() {
          override def run(): Unit = {
            runMain(childArgs, childClasspath, sparkConf, childMainClass, args.verbose)
          }
```

```
        })
      } catch {...}
    } else {
      runMain(childArgs, childClasspath, sparkConf, childMainClass, args.verbose)
    }
  }
  // ...
  if (args.isStandaloneCluster && args.useRest) {
    try {
      printStream.println("Running Spark using the REST application submission protocol.")
      doRunMain()
    } catch {
      case e: SubmitRestConnectionException => //...
        args.useRest = false
        submit(args, false)
    }
  } else {
    doRunMain()
  }
}

private def runMain(
    childArgs: Seq[String],
    childClasspath: Seq[String],
    sparkConf: SparkConf,
    childMainClass: String,
    verbose: Boolean): Unit = {
  // ...
  var mainClass: Class[_] = null

  try {
    mainClass = Utils.classForName(childMainClass)
  } catch {...}

  val app: SparkApplication = if (classOf[SparkApplication].isAssignableFrom(mainClass)) {
    mainClass.newInstance().asInstanceOf[SparkApplication]
  } else {
    // SPARK-4170
    if (classOf[scala.App].isAssignableFrom(mainClass)) {
      printWarning("Subclasses of scala.App may not work correctly. Use a main() method
instead.")
    }
    new JavaMainApplication(mainClass)
  }
  // ...
  try {
    app.start(childArgs.toArray, sparkConf)
  } catch {...}
```

SparkSubmit 首先解析传入进来的参数，从中得到 childArgs，childClasspath，sparkConf，childMainClass 四个主要的参数信息，然后启动 childMainClass，在 Spark on Kubernetes 中，childMainClass 就是 org.apache.spark.deploy.k8s.submit.KubernetesClientApplication。因此，SparkSubmit 启动 KubernetesClientApplication。

KubernetesClientApplication

```
private def run(clientArguments: ClientArguments, sparkConf: SparkConf): Unit = {
  val namespace = sparkConf.get(KUBERNETES_NAMESPACE)
  val kubernetesAppId = s"spark-${UUID.randomUUID().toString.replaceAll("-", "")}"
  val launchTime = System.currentTimeMillis()
  val waitForAppCompletion = sparkConf.get(WAIT_FOR_APP_COMPLETION)
  val appName = sparkConf.getOption("spark.app.name").getOrElse("spark")
  val master = sparkConf.get("spark.master").substring("k8s://".length)
  val loggingInterval = if (waitForAppCompletion) Some(sparkConf.get(REPORT_INTERVAL)) else None

  val watcher = new LoggingPodStatusWatcherImpl(kubernetesAppId, loggingInterval)

  val orchestrator = new DriverConfigOrchestrator(
    kubernetesAppId,
    launchTime,
    clientArguments.mainAppResource,
    appName,
    clientArguments.mainClass,
    clientArguments.driverArgs,
    sparkConf)

  Utils.tryWithResource(SparkKubernetesClientFactory.createKubernetesClient(
    master,
    Some(namespace),
    KUBERNETES_AUTH_SUBMISSION_CONF_PREFIX,
    sparkConf,
    None,
    None)) { kubernetesClient =>
      val client = new Client(
        orchestrator.getAllConfigurationSteps,
        sparkConf,
        kubernetesClient,
        waitForAppCompletion,
        appName,
        watcher)
      client.run()
  }
}
```

KubernetesClientApplication 启动时，首先再次解析传入的参数，之后调用上面的 run 方法来启动，然后创建一个包含一个可以与 Kubernetes 的 api-server 通信的 KubernetesClient 的 Client。之后，启动这个 client。

```
def run(): Unit = {
  var currentDriverSpec = KubernetesDriverSpec.initialSpec(sparkConf)
  for (nextStep <- submissionSteps) {
    currentDriverSpec = nextStep.configureDriver(currentDriverSpec)
  }
  val resolvedDriverJavaOpts = currentDriverSpec
    .driverSparkConf

    .remove(org.apache.spark.internal.config.DRIVER_JAVA_OPTIONS)
```

```
      .getAll
      .map {
        case (confKey, confValue) => s"-D$confKey=$confValue"
      } ++ driverJavaOptions.map(Utils.splitCommandString).getOrElse(Seq.empty)
    val driverJavaOptsEnvs: Seq[EnvVar] = resolvedDriverJavaOpts.zipWithIndex.map {
      case (option, index) =>
        new EnvVarBuilder()
          .withName(s"$ENV_JAVA_OPT_PREFIX$index")
          .withValue(option)
          .build()
    }
    val resolvedDriverContainer = new ContainerBuilder(currentDriverSpec.driverContainer)
      .addAllToEnv(driverJavaOptsEnvs.asJava)
      .build()
    val resolvedDriverPod = new PodBuilder(currentDriverSpec.driverPod)
      .editSpec()
        .addToContainers(resolvedDriverContainer)
        .endSpec()
      .build()
    Utils.tryWithResource(
      kubernetesClient
        .pods()
        .withName(resolvedDriverPod.getMetadata.getName)
        .watch(watcher)) { _ =>
      val createdDriverPod = kubernetesClient.pods().create(resolvedDriverPod)
      try {
        if (currentDriverSpec.otherKubernetesResources.nonEmpty) {
          val otherKubernetesResources = currentDriverSpec.otherKubernetesResources
          addDriverOwnerReference(createdDriverPod, otherKubernetesResources)
          kubernetesClient.resourceList(otherKubernetesResources: _*).createOrReplace()
        }
      } catch {
        case NonFatal(e) =>
          kubernetesClient.pods().delete(createdDriverPod)
          throw e
      }
      if (waitForAppCompletion) {
        logInfo(s"Waiting for application $appName to finish...")
        watcher.awaitCompletion()
        logInfo(s"Application $appName finished.")
      } else {
        logInfo(s"Deployed Spark application $appName into Kubernetes.")
      }
    }
  }
```

Client 启动之后，在拼接了用于配置启动 Driver 的 yaml 文件之后，利用之前创建的 KubernetesClient，通知 api-server 启动 Pod，这个 Pod 中包含一个 container，在这个 container 中，启动 Driver。其中拼接出来的 yaml 文件中包含在 spark-submit 时所提交的参数，这些参数中包含了 Pod 中所应该启动的程序是什么程序，每个 Pod 需要哪些资源等。

在 Pod 中启动了用户所提交的应用程序之后，启动 SparkContext。

SparkContext

```
try {
  //...
  val (sched, ts) = SparkContext.createTaskScheduler(this, master, deployMode)
  _schedulerBackend = sched
  _taskScheduler = ts
  _dagScheduler = new DAGScheduler(this)
  _heartbeatReceiver.ask[Boolean](TaskSchedulerIsSet)

  // start TaskScheduler after taskScheduler sets DAGScheduler reference in DAGScheduler's
  // constructor
  _taskScheduler.start()
  //...
  // Optionally scale number of executors dynamically based on workload. Exposed for testing.
  val dynamicAllocationEnabled = Utils.isDynamicAllocationEnabled(_conf)
  _executorAllocationManager =
    if (dynamicAllocationEnabled) {
      schedulerBackend match {
        case b: ExecutorAllocationClient =>
          Some(new ExecutorAllocationManager(
            schedulerBackend.asInstanceOf[ExecutorAllocationClient], listenerBus, _conf,
            _env.blockManager.master))
        case _ =>
          None
      }
    } else {
      None
    }
  _executorAllocationManager.foreach(_.start())
//...
  setupAndStartListenerBus()
  postEnvironmentUpdate()
  postApplicationStart()
//...
} catch {...}

private def createTaskScheduler(
    sc: SparkContext,
    master: String,
    deployMode: String): (SchedulerBackend, TaskScheduler) = {
  import SparkMasterRegex._
  // When running locally, don't try to re-execute tasks on failure.
  val MAX_LOCAL_TASK_FAILURES = 1
  master match {
    case "local" => //...
    case LOCAL_N_REGEX(threads) => //...
    case LOCAL_N_FAILURES_REGEX(threads, maxFailures) => //...
    case SPARK_REGEX(sparkUrl) => //...
    case LOCAL_CLUSTER_REGEX(numSlaves, coresPerSlave, memoryPerSlave) => //...
    case masterUrl =>

      val cm = getClusterManager(masterUrl) match {
```

```
          case Some(clusterMgr) => clusterMgr
          case None => throw new SparkException("Could not parse Master URL: '" + master + "'")
        }
        try {
          val scheduler = cm.createTaskScheduler(sc, masterUrl)
          val backend = cm.createSchedulerBackend(sc, masterUrl, scheduler)
          cm.initialize(scheduler, backend)
          (backend, scheduler)
        } catch {...}
    }
  }
```

在 SparkContext 中，创建 SparkContext 的同时就开始创建 TaskScheduler。首先获取对应的模式的集群资源管理器，在 Spark on kubernetes 中获取的是 KubernetesClusterManager。之后 KubernetesClusterManager 创建一个 TaskScheduler，并利用这个 TaskScheduler 创建 KubernetesClientSchedulerBackend。之后初始化 TaskScheduler 和 KubernetesClientSchedulerBackend。然后 SparkContext 会利用 TaskScheduler 创建一个 DAGScheduler。DAGScheduler 负责把用户提交的计算任务划分为一个个调度阶段。然后启动 TaskScheduler。

TaskScheduler 的启动过程，首先会把 KubernetesClientSchedulerBackend 同时也启动，然后 KubernetesClientSchedulerBackend 会启动 KubernetesDriverEndpoint 和 另外一个KubernetesClient。以此使得 Driver 可以控制 Kubernetes 集群和 Spark 集群，一方面通过 KubernetesClient 可以获取与 Kubernetes 集群沟通的能力，可以在需要的时候创建或者删除 Pod，另一方面通过 KubernetesDriverEndpoint 与 Spark 集群内的其它 executor 进行沟通，获取 executor 计算结果，删除无用的 executor 等。

KubernetesClientSchedulerBackend

```
override def start(): Unit = {
  super.start()
  executorWatchResource.set(
    kubernetesClient
      .pods()
      .withLabel(SPARK_APP_ID_LABEL, applicationId())
      .watch(new ExecutorPodsWatcher()))

  allocatorExecutor.scheduleWithFixedDelay(
      allocatorRunnable, 0L, podAllocationInterval, TimeUnit.MILLISECONDS)
  if (!Utils.isDynamicAllocationEnabled(conf)) {
    doRequestTotalExecutors(initialExecutors)
  }
}

private val allocatorRunnable = new Runnable {
  private val executorReasonCheckAttemptCounts = new mutable.HashMap[String, Int]
  override def run(): Unit = {
    handleDisconnectedExecutors()
    val executorsToAllocate = mutable.Map[String, Pod]()
    val currentTotalRegisteredExecutors = totalRegisteredExecutors.get
    val currentTotalExpectedExecutors = totalExpectedExecutors.get
    val currentNodeToLocalTaskCount = getNodesWithLocalTaskCounts()
    RUNNING_EXECUTOR_PODS_LOCK.synchronized {
      if (currentTotalRegisteredExecutors < runningExecutorsToPods.size) {
        logDebug("Waiting for pending executors before scaling")
```

```scala
        } else if (currentTotalExpectedExecutors <= runningExecutorsToPods.size) {
          logDebug("Maximum allowed executor limit reached. Not scaling up further.")
        } else {
          for (_ <- 0 until math.min(
            currentTotalExpectedExecutors - runningExecutorsToPods.size, podAllocationSize)) {
            val executorId = EXECUTOR_ID_COUNTER.incrementAndGet().toString
            val executorPod = executorPodFactory.createExecutorPod(
              executorId,
              applicationId(),
              driverUrl,
              conf.getExecutorEnv,
              driverPod,
              currentNodeToLocalTaskCount)
            executorsToAllocate(executorId) = executorPod
            logInfo(
              s"Requesting a new executor, total executors is now ${runningExecutorsToPods.size}")
          }
        }
      }
      val allocatedExecutors = executorsToAllocate.mapValues { pod =>
        Utils.tryLog {
          kubernetesClient.pods().create(pod)
        }
      }
      RUNNING_EXECUTOR_PODS_LOCK.synchronized {
        allocatedExecutors.map {
          case (executorId, attemptedAllocatedExecutor) =>
            attemptedAllocatedExecutor.map { successfullyAllocatedExecutor =>
              runningExecutorsToPods.put(executorId, successfullyAllocatedExecutor)
            }
        }
      }
    }
    // ...
  }
```

在启动 KubernetesClientSchedulerBackend 时，首先启动 KubernetesDriverEndpoint，之后设置周期性调度 executor，如果启动了动态调度，则每次获取当前所需的 executor 个数，否则，所需的 executor 个数是一个在 SparkConf 中确定的数目。与当前正在运行的 executor 数目进行对比，如果少了，则启动少了的数目个 Pod，这些 Pod 中包含一个 executor，或启动一次最大能启动的 Pod 数目个 Pod。

entrypoint.sh

```bash
case "$SPARK_K8S_CMD" in
  driver)
    CMD=(
      ${JAVA_HOME}/bin/java
      "${SPARK_JAVA_OPTS[@]}"
      -cp "$SPARK_CLASSPATH"
      -Xms$SPARK_DRIVER_MEMORY
      -Xmx$SPARK_DRIVER_MEMORY
      -Dspark.driver.bindAddress=$SPARK_DRIVER_BIND_ADDRESS
```

```
      $SPARK_DRIVER_CLASS
      $SPARK_DRIVER_ARGS
    )
    ;;

  executor)
    CMD=(
      ${JAVA_HOME}/bin/java
      "${SPARK_JAVA_OPTS[@]}"
      -Xms$SPARK_EXECUTOR_MEMORY
      -Xmx$SPARK_EXECUTOR_MEMORY
      -cp "$SPARK_CLASSPATH"
      org.apache.spark.executor.CoarseGrainedExecutorBackend
      --driver-url $SPARK_DRIVER_URL
      --executor-id $SPARK_EXECUTOR_ID
      --cores $SPARK_EXECUTOR_CORES
      --app-id $SPARK_APPLICATION_ID
      --hostname $SPARK_EXECUTOR_POD_IP
    )
    ;;

  init)
    CMD=(
      "$SPARK_HOME/bin/spark-class"
      "org.apache.spark.deploy.k8s.SparkPodInitContainer"
      "$@"
    )
    ;;

  *)
    echo "Unknown command: $SPARK_K8S_CMD" 1>&2
    exit 1
esac
```

当 Kubernetes 集群的 api-server 收到创建 Pod 的消息时，会根据 yaml 文件启动对应 docker 的镜像，并且根据 entrypoint.sh 来启动程序。Driver 对应的是用户自己所提交的程序，Executor 所对应的则是 org.apache.spark.executor.CoarseGrainedExecutorBackend。在这个程序中，会启动一个 RpcEndpoint 以此来与 Driver 的 KubernetesClientEndpoint 通信。