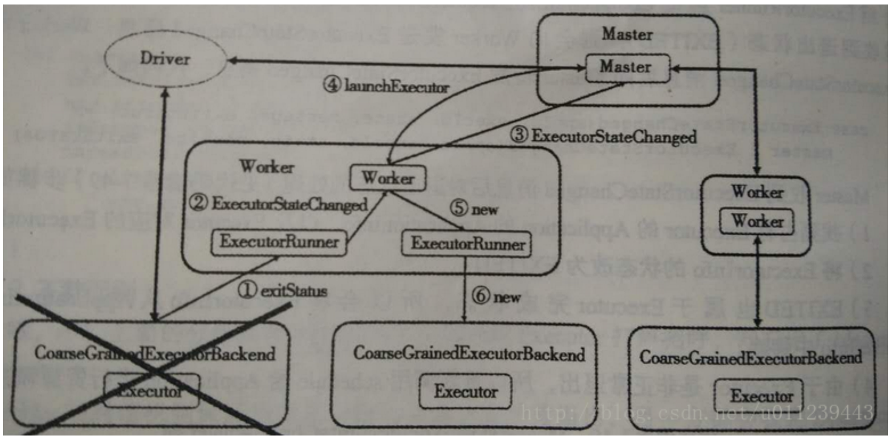
深入理解Spark 2.1 Core（八）：Standalone模式容错及HA的原理与源码分析

第五、第六、第七篇博文，我们讲解了Standalone模式集群是如何启动的，一个App起来了后，集群是如何分配资源，Worker启动Executor的，Task来是如何执行它，执行得到的结果如何处理，以及app退出后，分配了的资源如何回收。

但在分布式系统中，由于机器众多，所有发生故障是在所难免的，若运行过程中Executor、Worker或者Master异常退出了，那该怎么办呢？这篇博文，我们就来讲讲在Standalone模式下，Spark的集群容错与高可用性（HA）。

Executor



Worker.receive

我先回到《深入理解Spark 2.1 Core （六）：资源调度的原理与源码分析 》的ExecutorRunner.fetchAndRunExecutor中，看看executor的退出：

```
      // executor会退出并返回0或者非0的exitCode
      val exitCode = process.waitFor()
      state = ExecutorState.EXITED
      val message = "Command exited with code " + exitCode
      // 给Worker发送ExecutorStateChanged信号
      worker.send(ExecutorStateChanged(appId, execId, state, Some(message), Some(exitCode)))
```

worker接收到了ExecutorStateChanged信号后，调用handleExecutorStateChanged

```
    case executorStateChanged @ ExecutorStateChanged(appId, execId, state, message, exitStatus) =>
      handleExecutorStateChanged(executorStateChanged)
```

Worker.handleExecutorStateChanged

```
  private[worker] def handleExecutorStateChanged(executorStateChanged: ExecutorStateChanged):
    Unit = {
    // 给Master发送executorStateChanged信号
    sendToMaster(executorStateChanged)
    val state = executorStateChanged.state
    if (ExecutorState.isFinished(state)) {
    // 释放executor资源
      val appId = executorStateChanged.appId
      val fullId = appId + "/" + executorStateChanged.execId
      val message = executorStateChanged.message
      val exitStatus = executorStateChanged.exitStatus
```

```
      executors.get(fullId) match {
        case Some(executor) =>
          logInfo("Executor " + fullId + " finished with state " + state +
            message.map(" message " + _).getOrElse("") +
            exitStatus.map(" exitStatus " + _).getOrElse(""))
          executors -= fullId
          finishedExecutors(fullId) = executor
          trimFinishedExecutorsIfNecessary()
          coresUsed -= executor.cores
          memoryUsed -= executor.memory
        case None =>
          logInfo("Unknown Executor " + fullId + " finished with state " + state +
            message.map(" message " + _).getOrElse("") +
            exitStatus.map(" exitStatus " + _).getOrElse(""))
      }
      maybeCleanupApplication(appId)
    }
  }
}
```

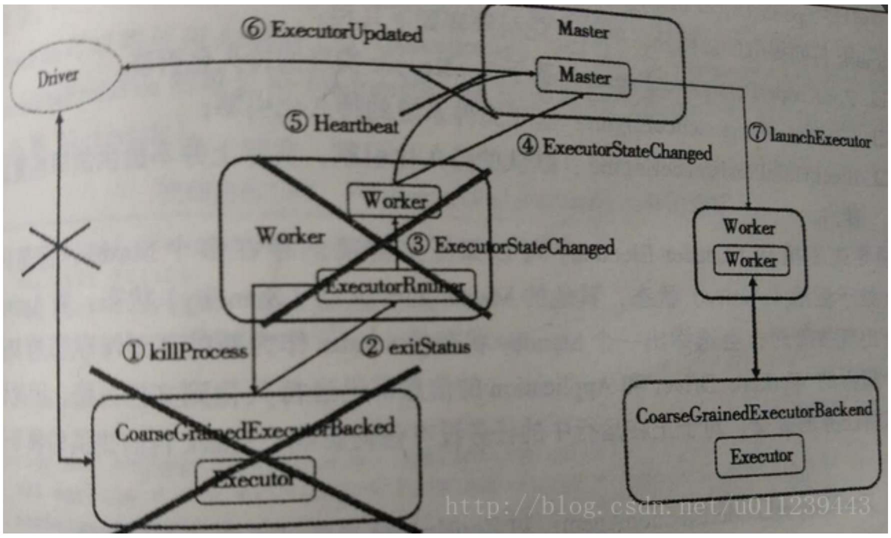
Master.receive

Master接收到ExecutorStateChanged信号后：

```
    case ExecutorStateChanged(appId, execId, state, message, exitStatus) =>
    // 通过appId取到App的信息，
    // 在App的信息中找到该executor的信息
      val execOption = idToApp.get(appId).flatMap(app => app.executors.get(execId))
      execOption match {
        case Some(exec) =>
          val appInfo = idToApp(appId)
          // 改变改executor的状态
          val oldState = exec.state
          exec.state = state

          if (state == ExecutorState.RUNNING) {
            assert(oldState == ExecutorState.LAUNCHING,
              s"executor $execId state transfer from $oldState to RUNNING is illegal")
            appInfo.resetRetryCount()
          }

          exec.application.driver.send(ExecutorUpdated(execId, state, message, exitStatus, false))

          if (ExecutorState.isFinished(state)) {
            logInfo(s"Removing executor ${exec.fullId} because it is $state")
            // 若该app已经结束，
            // 保持原来的executor信息，
            // 用于呈现在Web UI上，
            // 若该app还没结束，
            // 则从app信息中移除该executor
            if (!appInfo.isFinished) {
              appInfo.removeExecutor(exec)
            }
            // 把executor从worker中移除
            exec.worker.removeExecutor(exec)

            // 获取executor退出状态
            val normalExit = exitStatus == Some(0)
            // 若executor退出状态非正常，
            // 且app重新尝试调度次数到达最大重试次数，
            // 则移除这个app
            if (!normalExit
                && appInfo.incrementRetryCount() >= MAX_EXECUTOR_RETRIES
                && MAX_EXECUTOR_RETRIES >= 0) { // < 0 disables this application-killing path
```

```
            val execs = appInfo.executors.values
            if (!execs.exists(_.state == ExecutorState.RUNNING)) {
              logError(s"Application ${appInfo.desc.name} with ID ${appInfo.id} failed " +
                s"${appInfo.retryCount} times; removing it")
              removeApplication(appInfo, ApplicationState.FAILED)
            }
          }
        }
        //重新调度
        schedule()
```

Worker



Worker.killProcess

我们回到《深入理解Spark 2.1 Core （六）： 资源调度的原理与源码分析 》的ExecutorRunner.start中：

```
    // 创建Shutdownhook线程
    // 用于worker关闭时，杀掉executor
    shutdownHook = ShutdownHookManager.addShutdownHook { () =>
      if (state == ExecutorState.RUNNING) {
        state = ExecutorState.FAILED
      }
      killProcess(Some("Worker shutting down")) }
  }
```

worker退出后，ShutdownHookManager会调用killProcess杀死executor：

```
  private def killProcess(message: Option[String]) {
    var exitCode: Option[Int] = None
    if (process != null) {
      logInfo("Killing process!")
      // 停止运行日志输出
      if (stdoutAppender != null) {
        stdoutAppender.stop()
      }
      if (stderrAppender != null) {
      // 停止错误日志输出
        stderrAppender.stop()
      }
      // kill executor进程,
      // 并返回结束类型
      exitCode = Utils.terminateProcess(process, EXECUTOR_TERMINATE_TIMEOUT_MS)
```

```
      if (exitCode.isEmpty) {
        logWarning("Failed to terminate process: " + process +
          ". This process will likely be orphaned.")
      }
    }
    try {
    // 给worker发送ExecutorStateChanged信号
      worker.send(ExecutorStateChanged(appId, execId, state, message, exitCode))
    } catch {
      case e: IllegalStateException => logWarning(e.getMessage(), e)
    }
  }
```

Master.timeOutDeadWorkers

当Worker向Master注册直接的时候，会向worker的handleRegisterResponse发送RegisteredWorker信号。handleRegisterResponse处理该信号时会启动一个线程，来不断的给worker自己的SendHeartbeat信号

```
      case RegisteredWorker(masterRef, masterWebUiUrl) =>
  ***
        forwordMessageScheduler.scheduleAtFixedRate(new Runnable {
          override def run(): Unit = Utils.tryLogNonFatalError {
            self.send(SendHeartbeat)
          }
        }, 0, HEARTBEAT_MILLIS, TimeUnit.MILLISECONDS)
  ***
```

worker receive到SendHeartbeat信号后，处理：

```
  case SendHeartbeat =>
     if (connected) { sendToMaster(Heartbeat(workerId, self)) }
```

给Master发送Heartbeat信号。

Master接收到Heartbeat信号后处理：

```
    case Heartbeat(workerId, worker) =>
      idToWorker.get(workerId) match {
        case Some(workerInfo) =>
        // 更新worker的lastHeartbeat信息
          workerInfo.lastHeartbeat = System.currentTimeMillis()
        case None =>
          if (workers.map(_.id).contains(workerId)) {
            logWarning(s"Got heartbeat from unregistered worker $workerId." +
              " Asking it to re-register.")
            worker.send(ReconnectWorker(masterUrl))
          } else {
            logWarning(s"Got heartbeat from unregistered worker $workerId." +
              " This worker was never registered, so ignoring the heartbeat.")
          }
      }
```

而在Master.onStart中我可以看到：

```
    checkForWorkerTimeOutTask = forwardMessageThread.scheduleAtFixedRate(new Runnable {
      override def run(): Unit = Utils.tryLogNonFatalError {
        self.send(CheckForWorkerTimeOut)
      }
    }, 0, WORKER_TIMEOUT_MS, TimeUnit.MILLISECONDS)
```

也专门起了一个线程给自己发送CheckForWorkerTimeOut信号。Master receive到CheckForWorkerTimeOut信号后：

```
  case CheckForWorkerTimeOut =>
    timeOutDeadWorkers()
```

调用timeOutDeadWorkers：

```
private def timeOutDeadWorkers() {
  val currentTime = System.currentTimeMillis()
  // 过滤出 最后收到心跳的时间 < 现在的时间 - worker心跳间隔的worker
  val toRemove = workers.filter(_.lastHeartbeat < currentTime - WORKER_TIMEOUT_MS).toArray
  // 遍历这些worker
  for (worker <- toRemove) {
  // 若WorkerInfo 状态不为 DEAD
    if (worker.state != WorkerState.DEAD) {
      logWarning("Removing %s because we got no heartbeat in %d seconds".format(
        worker.id, WORKER_TIMEOUT_MS / 1000))
        // 调用removeWorker
      removeWorker(worker)
    }
    // 若WorkerInfo 状态为 DEAD
    else {
    // 等待足够长的时间后，
    // 再将它从workers列表中移除 ：
    // 最后收到心跳的时间 < 现在的时间 - worker心跳间隔 × REAPER_ITERATIONS
    // REAPER_ITERATIONS 由 spark.dead.worker.persistence 参数设置，
    // 默认为 15
      if (worker.lastHeartbeat < currentTime - ((REAPER_ITERATIONS + 1) * WORKER_TIMEOUT_MS)) {
        workers -= worker
      }
    }
  }
}
```

Master.removeWorker

```
private def removeWorker(worker: WorkerInfo) {
  logInfo("Removing worker " + worker.id + " on " + worker.host + ":" + worker.port)
  // 标志worker状态为DEAD
  worker.setState(WorkerState.DEAD)
  // 移除各个缓存
  idToWorker -= worker.id
  addressToWorker -= worker.endpoint.address
  if (reverseProxy) {
    webUi.removeProxyTargets(worker.id)
  }
  for (exec <- worker.executors.values) {
    logInfo("Telling app of lost executor: " + exec.id)
    // 向使用该executor的app，
    // 发送ExecutorUpdated信号
    exec.application.driver.send(ExecutorUpdated(
      exec.id, ExecutorState.LOST, Some("worker lost"), None, workerLost = true))
      // 标志executor状态为LOST
    exec.state = ExecutorState.LOST
    // 将executor从app信息中移除
    exec.application.removeExecutor(exec)
  }
  for (driver <- worker.drivers.values) {
  // 重启 或移除 Driver
    if (driver.desc.supervise) {
      logInfo(s"Re-launching ${driver.id}")
      relaunchDriver(driver)
    } else {
      logInfo(s"Not re-launching ${driver.id} because it was not supervised")
      removeDriver(driver.id, DriverState.ERROR, None)
```

```
      }
    }
    // 从持久化引擎中移除
    persistenceEngine.removeWorker(worker)
  }
```

Master.removeDriver

```
  private def removeDriver(
      driverId: String,
      finalState: DriverState,
      exception: Option[Exception]) {

    drivers.find(d => d.id == driverId) match {
      case Some(driver) =>
        logInfo(s"Removing driver: $driverId")
        // 从driver列表中移除
        drivers -= driver
        if (completedDrivers.size >= RETAINED_DRIVERS) {
          val toRemove = math.max(RETAINED_DRIVERS / 10, 1)
          completedDrivers.trimStart(toRemove)
        }
        // 加入到completedDrivers列表
        completedDrivers += driver
        // 从持久化引擎中移除
        persistenceEngine.removeDriver(driver)
        // 标志状态
        driver.state = finalState
        driver.exception = exception
        // 将这个driver注册过的worker，
        // 移除上面的driver
        driver.worker.foreach(w => w.removeDriver(driver))
        // 重新调度
        schedule()
      case None =>
        logWarning(s"Asked to remove unknown driver: $driverId")
    }
  }
}
```

Master

接下来我们来讲讲Master的容错及HA。在之前的Master代码中出现了持久化引擎persistenceEngine的对象，其实它就是实现Master的容错及HA的关键。我们先来看看Master.osStart中，会根据RECOVERY_MODE，来生成持久化引擎persistenceEngine和选举代理leaderElectionAgent。

```
    val (persistenceEngine_, leaderElectionAgent_) = RECOVERY_MODE match {
      case "ZOOKEEPER" =>
        logInfo("Persisting recovery state to ZooKeeper")
        val zkFactory =
          new ZooKeeperRecoveryModeFactory(conf, serializer)
        (zkFactory.createPersistenceEngine(), zkFactory.createLeaderElectionAgent(this))
      case "FILESYSTEM" =>
        val fsFactory =
          new FileSystemRecoveryModeFactory(conf, serializer)
        (fsFactory.createPersistenceEngine(), fsFactory.createLeaderElectionAgent(this))
      case "CUSTOM" =>
      // 用户自定义机制
        val clazz = Utils.classForName(conf.get("spark.deploy.recoveryMode.factory"))
        val factory = clazz.getConstructor(classOf[SparkConf], classOf[Serializer])
          .newInstance(conf, serializer)
          .asInstanceOf[StandaloneRecoveryModeFactory]
```

```
        (factory.createPersistenceEngine(), factory.createLeaderElectionAgent(this))
      case _ =>
      // 不做持久化
        (new BlackHolePersistenceEngine(), new MonarchyLeaderAgent(this))
    }
    persistenceEngine = persistenceEngine_
    leaderElectionAgent = leaderElectionAgent_
```

RECOVERY_MODE由spark.deploy.recoveryMode配置，默认为NONE：

```
  private val RECOVERY_MODE = conf.get("spark.deploy.recoveryMode", "NONE")
```

接下来，我们来深入讲解下FILESYSTEM和ZOOKEEPER这两种recoveryMode。

FILESYSTEM

FILESYSTEM recoveryMode下，集群的元数据信息会保存在本地文件系统。而Master启动后则会立即成为Active的Master。

```
      case "FILESYSTEM" =>
        val fsFactory =
          new FileSystemRecoveryModeFactory(conf, serializer)
        (fsFactory.createPersistenceEngine(), fsFactory.createLeaderElectionAgent(this))
```

FileSystemRecoveryModeFactory会生成两个对象，一个是FileSystemPersistenceEngine，一个是MonarchyLeaderAgent：

```
private[master] class FileSystemRecoveryModeFactory(conf: SparkConf, serializer: Serializer)
  extends StandaloneRecoveryModeFactory(conf, serializer) with Logging {

  val RECOVERY_DIR = conf.get("spark.deploy.recoveryDirectory", "")

  def createPersistenceEngine(): PersistenceEngine = {
    logInfo("Persisting recovery state to directory: " + RECOVERY_DIR)
    new FileSystemPersistenceEngine(RECOVERY_DIR, serializer)
  }

  def createLeaderElectionAgent(master: LeaderElectable): LeaderElectionAgent = {
    new MonarchyLeaderAgent(master)
  }
}
```

FileSystemRecoveryModeFactory

我们先来讲解下FileSystemRecoveryModeFactory：

```
private[master] class FileSystemPersistenceEngine(
    val dir: String,
    val serializer: Serializer)
  extends PersistenceEngine with Logging {

  // 新建一个目录
  new File(dir).mkdir()

  // 持久化对象，
  // 将对象序列化的写入到文件
  override def persist(name: String, obj: Object): Unit = {
    serializeIntoFile(new File(dir + File.separator + name), obj)
  }

  // 去持久化
  override def unpersist(name: String): Unit = {
    val f = new File(dir + File.separator + name)
    if (!f.delete()) {
```

```
      logWarning(s"Error deleting ${f.getPath()}")
    }
  }

  // 读取,
  // 根据文件名反序列化出
  override def read[T: ClassTag](prefix: String): Seq[T] = {
    val files = new File(dir).listFiles().filter(_.getName.startsWith(prefix))
    files.map(deserializeFromFile[T])
  }

 // 序列化到文件的实现
  private def serializeIntoFile(file: File, value: AnyRef) {
  // 生成新文件
    val created = file.createNewFile()
    if (!created) { throw new IllegalStateException("Could not create file: " + file) }
    // 输出文件流
    val fileOut = new FileOutputStream(file)
    var out: SerializationStream = null
    Utils.tryWithSafeFinally {
    // 根据输出文件流 生成 输出序列化流
      out = serializer.newInstance().serializeStream(fileOut)
      // 将值通过输出序列化流写入文件
      out.writeObject(value)
    } {

      // 关闭输出文件流
      fileOut.close()
      if (out != null) {
        out.close()
      }
    }
  }

   // 从文件反序列化的实现
  private def deserializeFromFile[T](file: File)(implicit m: ClassTag[T]): T = {
  // 输入文件流
    val fileIn = new FileInputStream(file)
    var in: DeserializationStream = null
    try {
    // 根据输入文件流 生成 输入序列化流
      in = serializer.newInstance().deserializeStream(fileIn)
      // 从文件反序列化读取对象
      in.readObject[T]()
    } finally {
    // 关闭输入文件流
      fileIn.close()
      if (in != null) {
        in.close()
      }
    }
  }

}
```

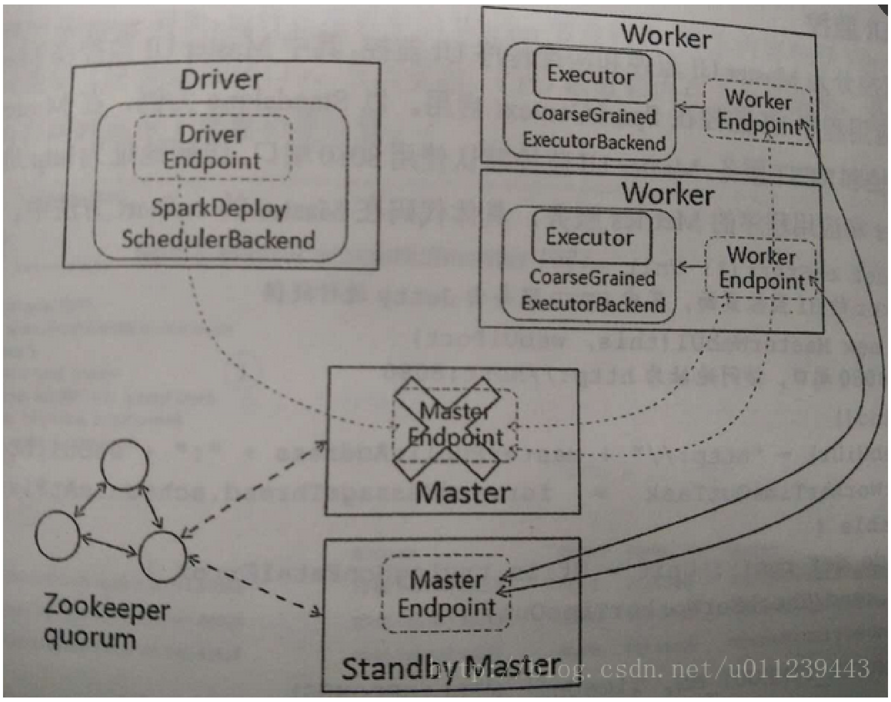
MonarchyLeaderAgent

```
@DeveloperApi
trait LeaderElectionAgent {
  val masterInstance: LeaderElectable
  def stop() {}
}

@DeveloperApi
```

```
trait LeaderElectable {
  def electedLeader(): Unit
  def revokedLeadership(): Unit
}

// 选举代理的单点实现
// 总是启动最初的Master
private[spark] class MonarchyLeaderAgent(val masterInstance: LeaderElectable)
  extends LeaderElectionAgent {
  masterInstance.electedLeader()
}
```

ZOOKEEPER

ZOOKEEPER recoveryMode下，集群的元数据信息会保存在ZooKeeper中。ZooKeeper会在备份的Master中选举出新的Master，新的Master在启动后会从ZooKeeper中获取数据信息并且恢复这些数据。




```
      case "ZOOKEEPER" =>
        logInfo("Persisting recovery state to ZooKeeper")
        val zkFactory =
          new ZooKeeperRecoveryModeFactory(conf, serializer)
        (zkFactory.createPersistenceEngine(), zkFactory.createLeaderElectionAgent(this))
```

ZooKeeperRecoveryModeFactory会生成两个对象，一个是ZooKeeperPersistenceEngine，一个是ZooKeeperLeaderElectionAgent：

```
private[master] class ZooKeeperRecoveryModeFactory(conf: SparkConf, serializer: Serializer)
  extends StandaloneRecoveryModeFactory(conf, serializer) {

  def createPersistenceEngine(): PersistenceEngine = {
    new ZooKeeperPersistenceEngine(conf, serializer)
  }

  def createLeaderElectionAgent(master: LeaderElectable): LeaderElectionAgent = {
    new ZooKeeperLeaderElectionAgent(master, conf)
  }
}
```

ZooKeeperPersistenceEngine

我们先来讲解下ZooKeeperPersistenceEngine：

```
private[master] class ZooKeeperPersistenceEngine(conf: SparkConf, val serializer: Serializer)
  extends PersistenceEngine
  with Logging {

  // 创建zk 及工作路径
  private val WORKING_DIR = conf.get("spark.deploy.zookeeper.dir", "/spark") + "/master_status"
  private val zk: CuratorFramework = SparkCuratorUtil.newClient(conf)

  SparkCuratorUtil.mkdir(zk, WORKING_DIR)

    // 持久化对象,
  // 将对象序列化的写入到zk
  override def persist(name: String, obj: Object): Unit = {
    serializeIntoFile(WORKING_DIR + "/" + name, obj)
  }

// 去持久化
  override def unpersist(name: String): Unit = {
    zk.delete().forPath(WORKING_DIR + "/" + name)
  }

  // 读取,
  // 根据文件名反序列化出
  override def read[T: ClassTag](prefix: String): Seq[T] = {
    zk.getChildren.forPath(WORKING_DIR).asScala
      .filter(_.startsWith(prefix)).flatMap(deserializeFromFile[T])
  }

 // 关闭zk
  override def close() {
    zk.close()
  }

 // 序列化到zk的实现
  private def serializeIntoFile(path: String, value: AnyRef) {
  // 序列化字节
    val serialized = serializer.newInstance().serialize(value)
    val bytes = new Array[Byte](serialized.remaining())
    serialized.get(bytes)

    // 写入到zk
    zk.create().withMode(CreateMode.PERSISTENT).forPath(path, bytes)
  }

// 从zk反序列化的实现
  private def deserializeFromFile[T](filename: String)(implicit m: ClassTag[T]): Option[T] = {
  // 从zk中得到数据
    val fileData = zk.getData().forPath(WORKING_DIR + "/" + filename)
    try {
    // 反序列化
      Some(serializer.newInstance().deserialize[T](ByteBuffer.wrap(fileData)))
    } catch {
      case e: Exception =>
        logWarning("Exception while reading persisted file, deleting", e)
        zk.delete().forPath(WORKING_DIR + "/" + filename)
        None
    }
  }
}
```

ZooKeeperLeaderElectionAgent

ZooKeeperLeaderElectionAgent被创建会调用start：

```
private def start() {
  logInfo("Starting ZooKeeper LeaderElection agent")
  zk = SparkCuratorUtil.newClient(conf)
  leaderLatch = new LeaderLatch(zk, WORKING_DIR)
  leaderLatch.addListener(this)
  leaderLatch.start()
}
```

leaderLatch.start()，启动了leader的竞争与选举。涉及到的ZooKeeper选举实现，已不在Spark源码范畴，所以在这不再讲解。

总结

- `Executor`退出：向`worker`发送`ExecutorStateChanged`信号；`worker`接收到信号后向`Master`发送`executorStateChanged`信号并释放该`Executor`资源；`Matser`收到信号后，改变该`Executor`状态，移除`Web UI`上该`Executor`的信息，若重试次数达到最大次数，则移除该`Application`，否则重新调度。
- Worker退出：`ShutdownHookManager`会调用`killProcess`杀死该所有的`executor`；`Mastser`利用心跳超时机制，得知`Worker`退出，改变该`Worker`状态，将该`Worker`上的`Executor`从`Application`信息中移除，将该`Worker`上的`driver`重启或移除，从持久化引擎中移除该`Worker`。
- Matser退出：FILESYSTEM recoveryMode下，集群的元数据信息会保存在本地文件系统，而Master启动后则会立即成为Active的Master；ZOOKEEPER recoveryMode下，集群的元数据信息会保存在ZooKeeper中，ZooKeeper会在备份的Master中选举出新的Master，新的Master在启动后会从ZooKeeper中获取数据信息并且恢复这些数据；除此之外还有用户自定义的恢复机制和不做持久化的机制。