# 第29课：Master HA彻底解密

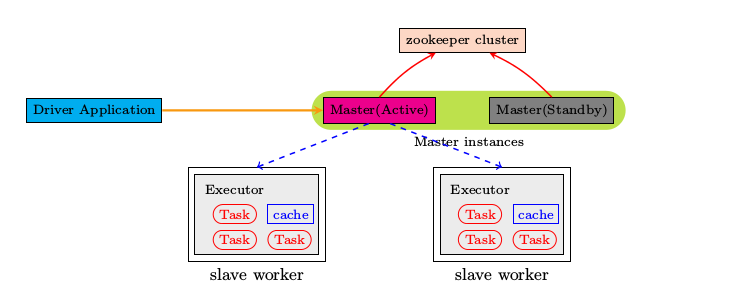
1. Master HA解析
2. Master HA的四大方式及源码解密
3. MasterHA的内部机制及源码解密

一：Master HA解析

用Standalone模式部署Spark集群，一般采用Master/Slave架构，但是这种模式下会发生SPOF问题。在这种情况下Spark提供了两种解决方式：

* 基于文件系统的单点恢复
* 基于zookeeper的Standby Masters

一般在生产环境下为了达到自动化管理集群一般采用zookeeper做HA，在实际应用中一般采用奇数台安装，建议大于等于3，zookeeper会自动化管理Master的切换。由于是自动化管理，必然存在安全风险。采用ZooKeeper做HA的时候，ZooKeeper负责保存整个Spark集群运行时候的元数据：Workers、Drivers、Applications、Executors。ZooKeeper提供了一个Leader Election机制，利用这个机制可以保证虽然集群存在多个Master，但是只有一个是Active的，其他的都是Standby。当Active的Master出现故障时，另外的一个Standby Master会被选举出来。但是要注意，**被选举后到成为真正的Active Master之间需要从ZooKeeper中获取集群当前运行状态的元数据信息并进行恢复**。其基本的工作机制为：



在Master切换的过程中，所有的已经在运行的程序皆正常运行！因为Spark Application在运行前就已经通过Cluster Manager获得了计算资源，所以在运行时Job本身的调度和处理和Master是没有任何关系的！同时值得注意的是，**在Master的切换过程中唯一的影响是不能提交新的Job**，一方面不能够提交新的应用程序给集群，因为只有Active Master才能接受新的程序的提交请求；另外一方面，已经运行的程序中也不能够因为Action操作触发新的Job的提交请求。

二：Master HA的四大方式

Master HA的四大方式分别是：ZOOKEEPER、FILESYSTEM、CUSTOM、NONE;

* ZOOKEEPER是自动管理Master；
* FILESYSTEM的方式在Master出现故障后需要手动重新启动机器，机器启动后会立即成为Active级别的Master来对外提供服务（接受应用程序提交的请求、接受新的Job运行的请求）；
* CUSTOM的方式允许用户自定义Master HA的实现，这对于高级用户特别有用；
* NONE，这是默认情况，当我们下载安装了Spark集群中就是采用这种方式，该方式不会持久化集群的数据，Master启动后立即管理集群；

Master::Onstart()的启动方式参考源代码：

**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" =>  
 **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** \_ =>  
 (**new** BlackHolePersistenceEngine(), **new** MonarchyLeaderAgent(**this**))  
 }  
 *persistenceEngine* = persistenceEngine\_  
 *leaderElectionAgent* = leaderElectionAgent\_  
}

其中关于Master的启动方式的配置RECOVERY\_MODE可以在spark-env.sh中去设置，如下：

1. spark.deploy.recoveryMode=ZOOKEEPER
2. spark.deploy.zookeeper.url=zk\_server\_1:2181,zk\_server\_2:2181
3. spark.deploy.zookeeper.dir=/dir
4. export SPARK\_DAEMON\_JAVA\_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER "
5. export SPARK\_DAEMON\_JAVA\_OPTS="${SPARK\_DAEMON\_JAVA\_OPTS} -Dspark.deploy.zookeeper.url=zk\_server1:2181,zk\_server\_2:2181"

spark.deploy.recoveryMode的具体实现的BlackHolePersistenceEngine源码：

**private**[master] **class** BlackHolePersistenceEngine **extends** PersistenceEngine {  
  
 **override def** persist(name: String, obj: Object): Unit = {}  
  
 **override def** unpersist(name: String): Unit = {}  
  
 **override def** read[T: ClassTag](name: String): Seq[T] = *Nil*}

PersistenceEngine中有一个至关重要的方法persist来实现数据持久化readPersistedData来恢复集群中的元数据，源码如下：

*/\*\*  
 \* Returns the persisted data sorted by their respective ids (which implies that they're  
 \* sorted by time of creation).  
 \*/***final def** readPersistedData(  
 rpcEnv: RpcEnv): (Seq[ApplicationInfo], Seq[DriverInfo], Seq[WorkerInfo]) = {  
 rpcEnv.deserialize { () =>  
 (read[ApplicationInfo]("app\_"), read[DriverInfo]("driver\_"), read[WorkerInfo]("worker\_"))  
 }  
}

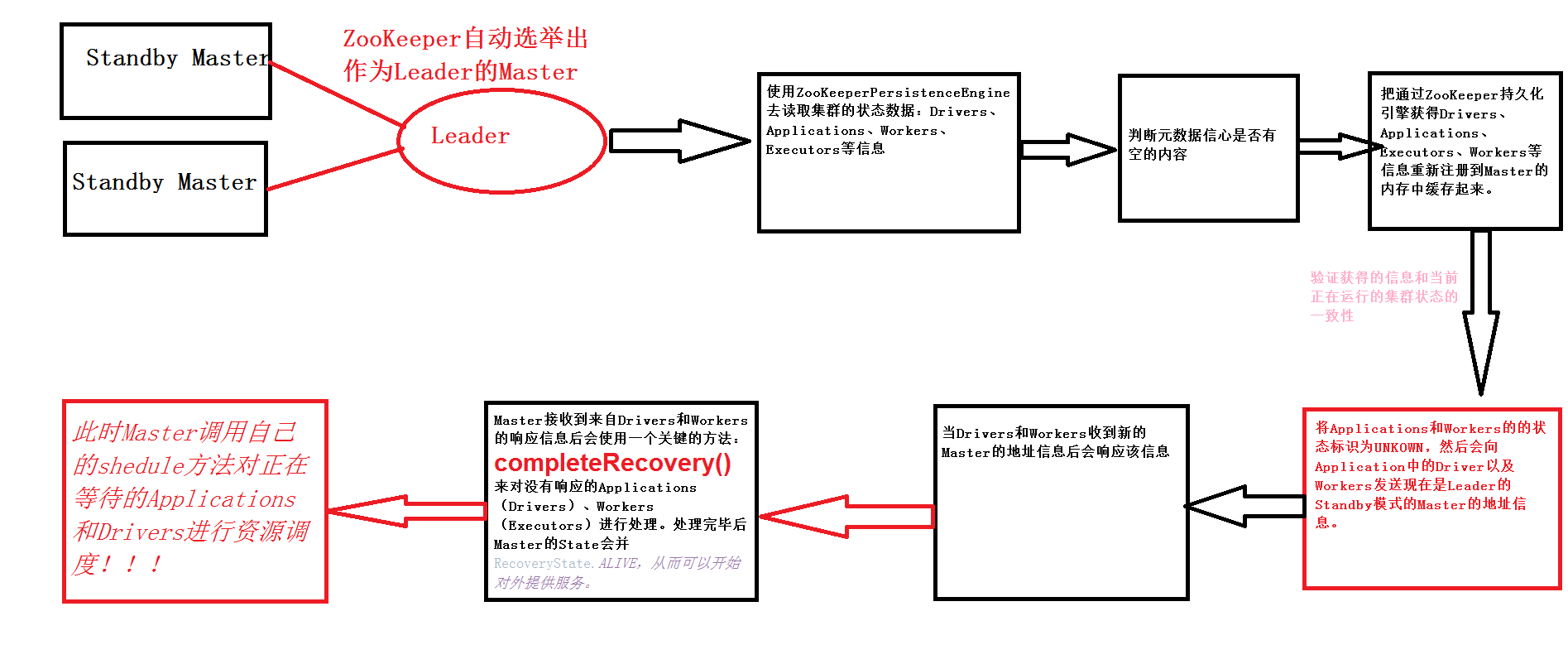
最后，FILESYSTEM和NONE的方式均是采用MonarchyElectionAgent的方式来完成Leader选举，其实际实现是直接将传入的Master设置为Leader。

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

*/\*\* Single-node implementation of LeaderElectionAgent -- we're initially and always the leader. \*/***private**[spark] **class** MonarchyLeaderAgent(**val** masterInstance: LeaderElectable)  
 **extends** LeaderElectionAgent {  
 masterInstance.electedLeader()  
}

三：Master HA的内部机制

Master HA在zookeeper下的基本流程图及源码：（修改： 判断元数据信息是否--）



在流程图中，用completeRecovery()方法来剔除没有响应的Drivers和Executors，具体实现源码如下：

**private def** completeRecovery() {  
 // Ensure "only-once" recovery semantics using a short synchronization period.  
 **if** (*state* != RecoveryState.*RECOVERING*) { **return** }  
 *state* = RecoveryState.*COMPLETING\_RECOVERY* // Kill off any workers and apps that didn't respond to us.  
 *workers*.filter(\_.*state* == WorkerState.*UNKNOWN*).foreach(removeWorker)  
 *apps*.filter(\_.*state* == ApplicationState.*UNKNOWN*).foreach(finishApplication)  
  
 // Reschedule drivers which were not claimed by any workers  
 *drivers*.filter(\_.*worker*.isEmpty).foreach { d =>  
 logWarning(s"Driver **$**{d.id} was not found after master recovery")  
 **if** (d.desc.supervise) {  
 logWarning(s"Re-launching **$**{d.id}")  
 relaunchDriver(d)  
 } **else** {  
 removeDriver(d.id, DriverState.*ERROR*, None)  
 logWarning(s"Did not re-launch **$**{d.id} because it was not supervised")  
 }  
 }  
  
 *state* = RecoveryState.*ALIVE* schedule()  
 logInfo("Recovery complete - resuming operations!")  
}

最后**通过*state* = RecoveryState.*ALIVE*对外提供服务，**其中最为重要的是替换后的Master会调用自己的shedule对处于等待状态的Drivers和Applications进行资源的调度，从此一面崭新的大门开启了。