

## Chapter 5 第5章

# Spark 执行环境

“其令郡国各修文学，县满五百户置校官，选其乡之俊造而教学之，庶几先王之道不废，而有以益于天下。”

——《建学令》

## 本章导读

正如曹操在《建学令》中所言，优秀人才的培育离不开国家政策这个大环境。在人类生活中，总是会依赖于外部的各种环境。一个人学习成绩的好坏，很大的因素取决于他所处的教育环境；一个人健康与否，依赖于他所处的医疗卫生环境；一个人的沟通能力，与自己的家庭环境有很重要的关系。如果某一天你发现自己联系不到亲人、朋友，甚至周围的环境都是陌生的，那么你立刻会处于崩溃的边缘。

使用过 Java 语言的人，应该清楚地记得自己第一次是如何磕磕绊绊老半天，才配置好 Java 环境的过程。有了这次糟糕的体验，我相信每个人在之后配置 Java 环境变量，甚至是 Maven、Scala、SBT 等其他环境变量时，就会变得游刃有余。无论是 Java 程序还是 Scala 程序，都需要运行在其所依托的环境下，脱离了这个环境，你将陷入困境。

Spark 对任务的计算都依托于 Executor 的能力，所有的 Executor 都有自己的 Spark 执行环境 SparkEnv。有了 SparkEnv，就可以将数据存储在存储体系中；就能利用计算引擎对计算任务进行处理，就可以在节点间进行通信等。SparkEnv 还提供了多种多样的内部组件，实现不同的功能。SparkEnv 是一个很重要的组件，虽然在创建 SparkContext 的时候也涉及它（只是因为 local 模式的需要），但是它与 Executor 的关系则更为紧密，所以专门利用一章的内容来详细介绍它。

本章主要讲解的内容如下。

- 安全管理器 SecurityManager。
- RPC 环境。
- 序列化管理器 SerializerManager。
- 广播管理器 BroadcastManager。
- map 任务输出跟踪器。
- 度量系统。

## 5.1 SparkEnv 概述

SparkEnv 的私有方法 create 用于创建 SparkEnv，由于 create 方法涉及很多 SparkEnv 内部组件的实例化过程，所以首先通过图 5-1 来了解 SparkEnv 的组成。

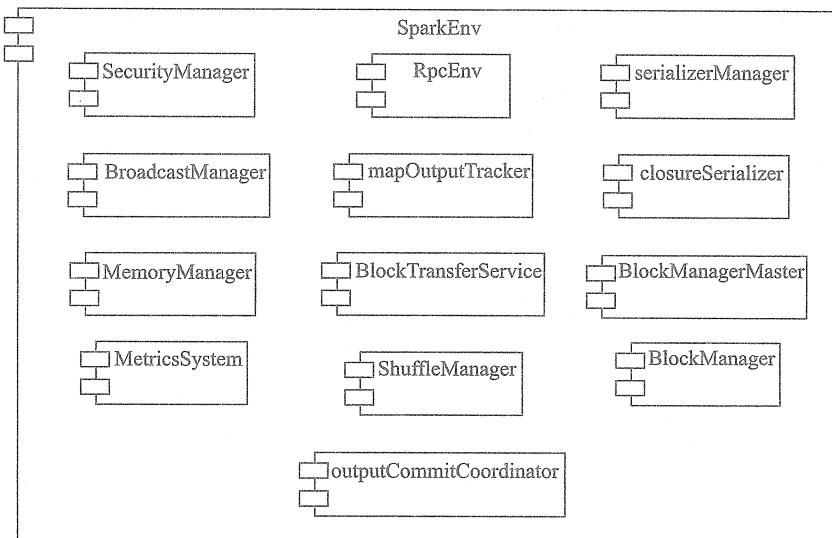


图 5-1 SparkEnv 的组成

有了对 SparkEnv 组成的直观感受，下面我们来对其内部组件逐一进行介绍。

## 5.2 安全管理器 SecurityManager

SecurityManager 主要对账号、权限及身份认证进行设置和管理。如果 Spark 的部署模式为 YARN，则需要生成 secret key（密钥）并存入 Hadoop UGI。而在其他模式下，则需要设置环境变量 \_SPARK\_AUTH\_SECRET（优先级更高）或 spark.authenticate.secret 属性指定 secret key（密钥）。SecurityManager 还会给当前系统设置默认的口令认证实例。在

SparkEnv 中创建 SecurityManager 的代码如下。

```
val securityManager = new SecurityManager(conf, ioEncryptionKey)
```

SecurityManager 内部有很多属性，我们先来对这些属性进行介绍。

- authOn：是否开启认证。可以通过 spark.authenticate 属性配置，默认为 false。
- aclsOn：是否对账号进行授权检查。可通过 spark.acls.enable（优先级较高）或 spark.ui.acls.enable（此属性是为了向前兼容）属性进行配置。aclsOn 的默认值为 false。
- adminAcls：管理员账号集合。可以通过 spark.admin.acls 属性配置，默认为空。
- adminAclsGroups：管理员账号所在组的集合。可以通过 spark.admin.acls.groups 属性配置，默认为空。
- viewAcls：有查看权限的账号的集合。包括 adminAcls、defaultAclUsers 及 spark.ui.view.acls 属性配置的用户。
- viewAclsGroups：拥有查看权限的账号，所在组的集合。包括 adminAclsGroups 和 spark.ui.view.acls.groups 属性配置的用户。
- modifyAcls：有修改权限的账号的集合。包括 adminAcls、defaultAclUsers 及 spark.modify.acls 属性配置的用户。
- modifyAclsGroups：拥有修改权限的账号所在组的集合。包括 adminAclsGroups 和 spark.modify.acls.groups 属性配置的用户。
- defaultAclUsers：默认用户。包括系统属性 user.name 指定的用户或系统登录用户或者通过系统环境变量 SPARK\_USER 进行设置的用户。
- secretKey：密钥。在 YARN 模式下，首先使用 sparkCookie 从 Hadoop UGI 中获取密钥。如果 Hadoop UGI 没有保存密钥，则生成新的密钥（密钥长度可以通过 spark.authenticate.secretBitLength 属性指定）并存入 Hadoop UGI。其他模式下，则需要设置环境变量 \_SPARK\_AUTH\_SECRET（优先级更高）或 spark.authenticate.secret 属性指定。
- 此外还有很多 SSL 相关的配置。有关 SSL 的内容不属于本书要阐述的范围，感兴趣的读者可以查阅相关资料。

SecurityManager 中设置了默认的口令认证实例 Authenticator，此实例采用匿名内部类实现，用于每次使用 HTTP client 从 HTTP 服务器获取用户的用户名和密码。这是由于 Spark 的节点间通信往往需要动态协商用户名、密码，这种方式灵活地支持了这种需求。设置默认身份认证器的代码如下。

```
if (authOn) {
    // 设置默认的口令认证实例Authenticator，它的getPasswordAuthentication方法用于获取用户名、密码
    Authenticator.setDefault(
        new Authenticator() {
```

```

    override def getPasswordAuthentication(): PasswordAuthentication = {
      var passAuth: PasswordAuthentication = null
      val userInfo = getRequestingURL().getUserInfo()
      if (userInfo != null) {
        val parts = userInfo.split(":", 2)
        passAuth = new PasswordAuthentication(parts(0), parts(1).toCharArray())
      }
      return passAuth
    }
  }
}

```

### 5.3 RPC 环境

RpcEnv 是 Spark 2.x.x 版本中新出现的组件，那么它是不是给 Spark 带来了新的能力呢？对于有些组件是这样的，但是 RpcEnv 组件肩负着另一项历史使命——替代 Spark 2.x.x 以前版本中采用的 Akka。Spark 移除了对 Akka 的使用，这令我感到困惑，“Akka 是一款很不错的分布式消息系统，为什么要替代？正如 3.2 节所说的——可以让用户使用任何版本的 Akka 来编程”。Akka 具有分布式集群下的消息发送、远端同步调用、远端异步调用、路由、持久化、监管、At Least Once Delivery（至少投递一次）等能力，一旦 Akka 被替代，那就意味着 RpcEnv 必须也能支持这些机制。

现在我们来看看 SparkEnv 中的 RpcEnv 是如何被创建的，SparkEnv 中创建 RpcEnv 的代码如下。

```

val systemName = if (isDriver) driverSystemName else executorSystemName
val rpcEnv = RpcEnv.create(systemName, bindAddress, advertiseAddress, port, conf,
  securityManager, clientMode = !isDriver)

```

这段代码首先生成系统名称 systemName，如果当前应用为 Driver（即 SparkEnv 位于 Driver 中），那么 systemName 为 sparkDriver，否则（即 SparkEnv 位于 Executor 中），systemName 为 sparkExecutor。然后调用 RpcEnv 的 create 方法创建 RpcEnv。create 方法中只有两条语句。

```

val config = RpcEnvConfig(conf, name, bindAddress, advertiseAddress, port,
  securityManager,
  clientMode) // RpcEnvConfig 中保存了 RpcEnv 的配置信息
new NettyRpcEnvFactory().create(config) // 创建 RpcEnv

```

这里的 RpcEnvConfig 实际是一个样例类，用于保存 RpcEnv 的配置信息。实际创建 RpcEnv 的动作发生在 NettyRpcEnvFactory 的 create 方法中，如代码清单 5-1 所示。

代码清单 5-1 创建 NettyRpcEnv

---

```

def create(config: RpcEnvConfig): RpcEnv = {
  val sparkConf = config.conf
  val javaSerializerInstance =

```

```

    new JavaSerializer(sparkConf).newInstance().asInstanceOf[JavaSerializerInstance]
  val nettyEnv =
    new NettyRpcEnv(sparkConf, javaSerializerInstance, config.advertiseAddress,
      config.securityManager)
  if (!config.clientMode) {
    val startNettyRpcEnv: Int => (NettyRpcEnv, Int) = { actualPort =>
      nettyEnv.startServer(config.bindAddress, actualPort)
      (nettyEnv, nettyEnv.address.port)
    }
    try {
      Utils.startServiceOnPort(config.port, startNettyRpcEnv, sparkConf, config.
        name)._1
    } catch {
      case NonFatal(e) =>
        nettyEnv.shutdown()
        throw e
    }
  }
  nettyEnv
}
}

```

代码清单 5-1 中创建 NettyRpcEnv 的步骤如下。

- 1 ) 创建 javaSerializerInstance。此实例将用于 RPC 传输对象的序列化。
- 2 ) 创建 NettyRpcEnv。创建 NettyRpcEnv 其实就是对内部各个子组件 TransportConf、Dispatcher、TransportContext、TransportClientFactory、TransportServer 的实例化过程，这些内容都将在后面一一介绍。
- 3 ) 启动 NettyRpcEnv。这一步首先定义了一个偏函数 startNettyRpcEnv，其函数实际为执行 NettyRpcEnv 的 startServer 方法（startServer 方法的具体内容将在 5.3.7 节进行介绍），最后在启动 NettyRpcEnv 之后返回 NettyRpcEnv 及服务最终使用的端口。这里使用了 Utils 的 startServiceOnPort 方法，startServiceOnPort 实际上是调用了作为参数的偏函数 startNettyRpcEnv，有关 startServiceOnPort 的具体介绍可以参阅附录 A 中的内容。

由于抽象类 RpcEnv 只有一个实现子类 NettyRpcEnv，所以本章不打算展示 RpcEnv 的接口定义，而是直接从介绍 NettyRpcEnv入手。在正式介绍 NettyRpcEnv 的构造过程之前，我们需要先做一些准备。RpcEndpoint 和 RpcEndpointRef 都是 NettyRpcEnv 中的重要概念，我们需要首先理解它们，再来看 NettyRpcEnv 的构造过程。

### 5.3.1 RPC 端点 RpcEndpoint

RPC 端点是对 Spark 的 RPC 通信实体的统一抽象，所有运行于 RPC 框架之上的实体都应该继承 RpcEndpoint。Spark 早期版本节点间的消息通信主要采用 Akka 的 Actor，从 Spark 2.0.0 版本开始移除了对 Akka 的依赖，这就意味着 Spark 需要 Actor 的替代品，RPC 端点 RpcEndpoint 由此而生。RpcEndpoint 是对能够处理 RPC 请求，给某一特定服务提供

本地调用及跨节点调用的 RPC 组件的抽象。

### 1. RPC 端点 RpcEndpoint 的定义

特质 RpcEndpoint 的定义如代码清单 5-2 所示。

代码清单5-2 RPC端点定义

---

```
private[spark] trait RpcEndpoint {
    val rpcEnv: RpcEnv
    final def self: RpcEndpointRef = {
        require(rpcEnv != null, "rpcEnv has not been initialized")
        rpcEnv.endpointRef(this)
    }
    def receive: PartialFunction[Any, Unit] = {
        case _ => throw new SparkException(self + " does not implement 'receive'")
    }
    def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
        case _ => context.sendFailure(new SparkException(self + " won't reply
            anything"))
    }
    def onError(cause: Throwable): Unit = {
        throw cause
    }
    def onConnected(remoteAddress: RpcAddress): Unit = {
    }
    def onDisconnected(remoteAddress: RpcAddress): Unit = {
    }
    def onNetworkError(cause: Throwable, remoteAddress: RpcAddress): Unit = {
    }
    def onStart(): Unit = {
    }
    def onStop(): Unit = {
    }
    final def stop(): Unit = {
        val _self = self
        if (_self != null) {
            rpcEnv.stop(_self)
        }
    }
}
```

---

这里对代码清单 5-2 中所展示的 RpcEndpoint 的各个成员进行介绍。

- rpcEnv: 当前 RpcEndpoint 所属的 RpcEnv。
- self: 获取 RpcEndpoint 相关联的 RpcEndpointRef。从代码实现看到, 其实现实际调用了 RpcEnv 的 endpointRef 方法。由于 RpcEnv 并未实现此方法, 所以需要 RpcEnv 的子类来实现。以 NettyRpcEnv 为例, 其 endpointRef 方法的实现如代码清单 5-40 所示。
- receive: 接收消息并处理, 但不需要给客户端回复。
- receiveAndReply: 接收消息并处理, 需要给客户端回复。回复是通过 RpcCall-Context 来实现的。

- ❑ `onError`: 当处理消息发生异常时调用，可以对异常进行一些处理。
- ❑ `onConnected`: 当客户端与当前节点连接上之后调用，可以针对连接进行一些处理。
- ❑ `onDisconnected`: 当客户端与当前节点的连接断开之后调用，可以针对断开连接进行一些处理。
- ❑ `onNetworkError`: 当客户端与当前节点之间的连接发生网络错误时调用，可以针对连接发生的网络错误进行一些处理。
- ❑ `onStart`: 在 `RpcEndpoint` 开始处理消息之前调用，可以在 `RpcEndpoint` 正式工作之前做一些准备工作。
- ❑ `onStop`: 在停止 `RpcEndpoint` 时调用，可以在 `RpcEndpoint` 停止的时候做一些收尾工作。
- ❑ `stop`: 用于停止当前 `RpcEndpoint`。从代码实现看到，其实现实际调用了 `RpcEnv` 的 `stop` 方法。由于 `RpcEnv` 并未实现此方法，所以需要其子类来实现。以 `NettyRpcEnv` 为例，其 `stop` 方法的实现如 5.3.9 节中代码清单 5-45 所示。

对 Akka 的 Actor 编程模型熟悉的读者，一定觉得 `RpcEndpoint` 中的一些接口非常类似于 Actor，笔者认为 Spark 2.0.0 版本在去掉了 Akka 的同时，自身实现 `RpcEnv` 时肯定参考了 Actor 的一些思想。

## 2. 特质 `RpcEndpoint` 的继承体系

由于 `RpcEndpoint` 只是一个特质，除了对接口的定义，并没有任何实现逻辑，所以我们需要看看有哪些子类实现了 `RpcEndpoint`。`RpcEndpoint` 的继承体系如图 5-2 所示。

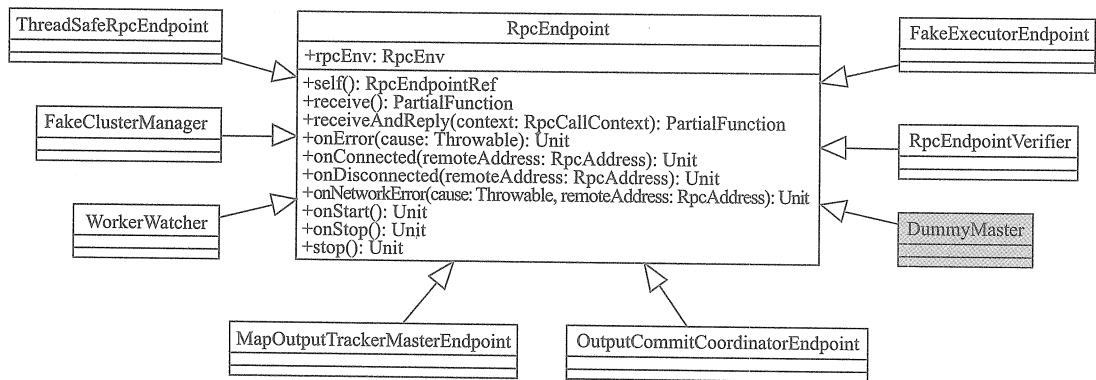


图 5-2 `RpcEndpoint` 的继承体系

图 5-2 中灰色的子类型 `DummyMaster`（`Dummy` 意为虚拟的、假的或傀儡的）正如其名字一样，不是 `NettyRpcEnv` 中具有真正用途的 `RpcEndpoint`，而只用于测试。图中显示的 `RpcEndpoint` 的子类，除了 `ThreadSafeRpcEndpoint` 之外，都将在具体遇到时再作详细分析。`ThreadSafeRpcEndpoint` 是继承自 `RpcEndpoint` 的特质，主要用于对消息的处理，必须是线

程安全的场景。ThreadSafeRpcEndpoint 对消息的处理都是串行的，即前一条消息处理完才能接着处理下一条消息。ThreadSafeRpcEndpoint 的继承体系如图 5-3 所示。

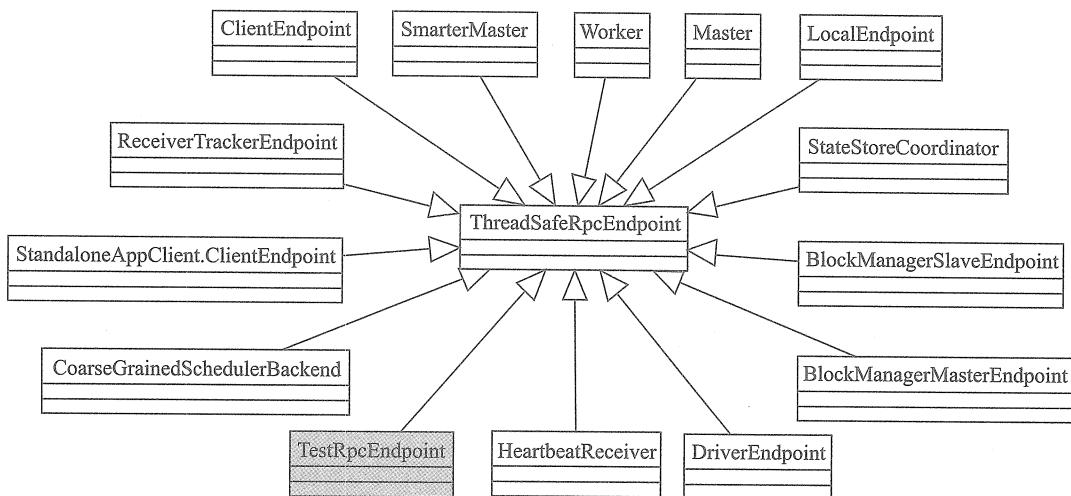


图 5-3 ThreadSafeRpcEndpoint 的继承体系

图 5-3 中的 TestRpcEndpoint 用于测试，其余实现类都在 NettyRpcEnv 中发挥着各自的作用，这些 RpcEndpoint 将在本书后续的各个章节中需要时再作具体介绍。

### 5.3.2 RPC 端点引用 RpcEndpointRef

如果说 RpcEndpoint 是 Akka 中 Actor 的替代产物，那么 RpcEndpointRef 就是 Actor-Ref 的替代产物。在 Akka 中只要你持有了一个 Actor 的引用 ActorRef，那么你就可以使用此 ActorRef 向远端的 Actor 发起请求。RpcEndpointRef 也具有同等的功用，要向一个远端的 RpcEndpoint 发起请求，你就必须持有这个 RpcEndpoint 的 RpcEndpointRef。RpcEndpoint 与 RpcEndpointRef 之间的关系如图 5-4 所示。

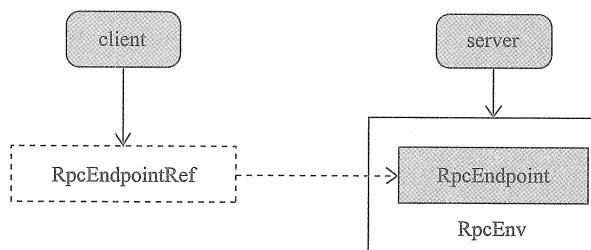


图 5-4 RpcEndpoint 与 RpcEndpointRef 的关系

在正式介绍 RpcEndpointRef 之前，先要理解什么是消息投递规则。

## 1. 消息投递规则

一般而言，消息投递有下面 3 种情况。

- **at-most-once**：意味着每条应用了这种机制的消息会被投递 0 次或 1 次。可以说这条消息可能会丢失。
- **at-least-once**：意味着每条应用了这种机制的消息潜在地存在多次投递尝试并保证至少会成功一次。就是说这条消息可能会重复但是不会丢失。
- **exactly-once**：意味着每条应用了这种机制的消息只会向接收者准确地发送一次。换言之，这种消息既不会丢失，也不会重复。

**at-most-once** 的成本最低且性能最高，因为它在发送完消息后不会尝试去记录任何状态，然后这条消息将被抛之脑后。**at-least-once** 需要发送者必须认识它所发送过的消息，并对没有收到回复的消息进行发送重试。这就要求接收者对消息的处理必须是幂等的，否则可能会带来新的问题。**exactly-once** 的成本是三者中最高，性能却又是三者中最差的。它除了要求发送者有记忆和重试能力，还要求接收者能够认识接收过的消息并能过滤出那些重复的消息投递。

## 2. RPC 端点引用 `RpcEndpointRef` 的定义

有了对消息投递规则的理解，我们现在开始介绍 `RpcEndpointRef`。抽象类 `RpcEndpointRef` 定义了所有 `RpcEndpoint` 引用的属性与接口，如代码清单 5-3 所示。

代码清单5-3 `RpcEndpointRef`的接口定义

---

```
private[spark] abstract class RpcEndpointRef(conf: SparkConf)
  extends Serializable with Logging {
  private[this] val maxRetries = RpcUtils.numRetries(conf)
  private[this] val retryWaitMs = RpcUtils.retryWaitMs(conf)
  private[this] val defaultAskTimeout = RpcUtils.askRpcTimeout(conf)
  def address: RpcAddress
  def name: String
  def send(message: Any): Unit
  def ask[T: ClassTag](message: Any, timeout: RpcTimeout): Future[T]
  def ask[T: ClassTag](message: Any): Future[T] = ask(message, defaultAskTimeout)
  def askWithRetry[T: ClassTag](message: Any): T = askWithRetry(message,
    defaultAskTimeout)
  def askWithRetry[T: ClassTag](message: Any, timeout: RpcTimeout): T = {
    var attempts = 0
    var lastException: Exception = null
    while (attempts < maxRetries) {
      attempts += 1
      try {
        val future = ask[T](message, timeout)
        val result = timeout.awaitResult(future)
        if (result == null) {
          throw new SparkException("RpcEndpoint returned null")
        }
        return result
      } catch {
```

```

        case ie: InterruptedException => throw ie
    case e: Exception =>
        lastException = e
        logWarning(s"Error sending message [message = $message] in $attempts
attempts", e)
    }

    if (attempts < maxRetries) {
        Thread.sleep(retryWaitMs)
    }
}

throw new SparkException(
    s"Error sending message [message = $message]", lastException)
}
}

```

---

从代码清单 5-3 中看到，抽象类 `RpcEndpointRef` 中有 3 个属性，分别如下。

- ❑ `maxRetries`：RPC 最大重新连接次数。可以使用 `spark.rpc.numRetries` 属性进行配置，默认为 3 次。
- ❑ `retryWaitMs`：RPC 每次重新连接需要等待的毫秒数。可以使用 `spark.rpc.retry.wait` 属性进行配置，默认值为 3 秒。
- ❑ `defaultAskTimeout`：RPC 的 `ask` 操作的默认超时时间。可以使用 `spark.rpc.askTimeout` 或者 `spark.network.timeout` 属性进行配置，默认值为 120 秒。`spark.rpc.askTimeout` 属性的优先级更高。

这三个属性都是通过调用工具类 `RpcUtils` 的方法得到的，想了解 `RpcUtils` 的介绍，可以参阅附录 H。

接着来看一看 `RpcEndpointRef` 中各个方法的功能。

- ❑ `address`：返回当前 `RpcEndpointRef` 对应 `RpcEndpoint` 的 RPC 地址 (`RpcAddress`)。
- ❑ `name`：返回当前 `RpcEndpointRef` 对应 `RpcEndpoint` 的名称。
- ❑ `send`：发送单向异步的消息。所谓“单向”就是发送完后就会忘记此次发送，不会有任何状态要记录，也不会期望得到服务端的回复。`send` 采用了 `at-most-once` 的投递规则。`RpcEndpointRef` 的 `send` 方法非常类似于 Akka 中 Actor 的 `tell` 方法。
- ❑ `ask[T: ClassTag](message: Any)`：以默认的超时时间作为 `timeout` 参数，调用 `ask[T: ClassTag](message: Any, timeout: RpcTimeout)` 方法。
- ❑ `askWithRetry[T: ClassTag](message: Any, timeout: RpcTimeout)`：发送同步的请求，此类请求将会被 `RpcEndpoint` 接收，并在指定的超时时间内等待返回类型为 `T` 的处理结果。当此方法抛出 `SparkException` 时，将会进行请求重试，直到超过了默认的重试次数为止。由于此类方法会重试，因此要求服务端对消息的处理是幂等的。此

方法也采用了 at-least-once 的投递规则。此方法也非常类似于 Akka 中采用了 at-least-once 机制的 Actor 的 ask 方法。

- askWithRetry[T: ClassTag](message: Any): 以默认的超时时间作为 timeout 参数，调用 askWithRetry[T: ClassTag](message: Any, timeout: RpcTimeout)。

**小贴士：**Akka 的 Actor 模型一般提供的消息都属于 at-most-once，那是因为大多数场景不需要有状态的消息投递，如 Web 服务器请求。当你有强一致性需求时，才应该启用 Akka 的 at-least-once 机制。

### 5.3.3 创建传输上下文 TransportConf

根据 3.2.1 节的介绍，我们知道 TransportConf 是 RPC 框架中的配置类。由于 RPC 环境 RpcEnv 的底层也依赖于数据总线，因此需要创建传输上下文 TransportConf。创建 TransportConf 是构造 NettyRpcEnv 的过程中的第一步，代码如下。

```
private[netty] val transportConf = SparkTransportConf.fromSparkConf(
  conf.clone.set("spark.rpc.io.numConnectionsPerPeer", "1"),
  "rpc",
  conf.getInt("spark.rpc.io.threads", 0))
```

这里用到了 3.2.1 节介绍的 SparkTransportConf，可以看到在调用 SparkTransportConf 的 fromSparkConf 方法之前，首先对 SparkConf 进行了克隆，然后设置了 spark.rpc.io.numConnectionsPerPeer，此属性的格式在 3.2.2 节分析 TransportClientFactory 的成员变量时介绍过，由于当前模块名为 rpc，所以为 spark.rpc.io.numConnectionsPerPeer。我们还看到这里通过 spark.rpc.io.threads 属性来设置 Netty 传输线程数。

### 5.3.4 消息调度器 Dispatcher

创建消息调度器 Dispatcher 是有效提高 NettyRpcEnv 对消息异步处理并最大提升并行处理能力的前提。Dispatcher 负责将 RPC 消息路由到要该对此消息处理的 RpcEndpoint (RPC 端点)。创建 Dispatcher 的代码如下。

```
private val dispatcher: Dispatcher = new Dispatcher(this)
```

#### 1. 消息调度器 Dispatcher 的概述

想要把 Dispatcher 解释得足够明白，最好的方式是先弄明白其中涉及的概念。Dispatcher 中的概念包括以下几个。

- **RpcEndpoint**：RPC 端点，即 RPC 分布式环境中一个具体的实例，其可以对指定的消息进行处理。由于 RpcEndpoint 是一个特质，所以需要提供 RpcEndpoint 的实现类。特质 RpcEndpoint 已在前文详细介绍，此处不再赘述。
- **RpcEndpointRef**：RPC 端点引用，即 RPC 分布式环境中一个具体实体的引用，所

谓引用实际是“`spark://host:port/name`”这种格式的地址。其中，`host`为端点所在 RPC 服务所在的主机 IP，`port`是端点所在 RPC 服务的端口，`name`是端点实例的名称。抽象类 `RpcEndpointRef`已在前文详细介绍，此处不再赘述。

- `InboxMessage`：Inbox 盒子内的消息。`InboxMessage`是一个特质，所有类型的 RPC 消息都继承自 `InboxMessage`。`InboxMessage` 的继承体系可以用图 5-5 表示。

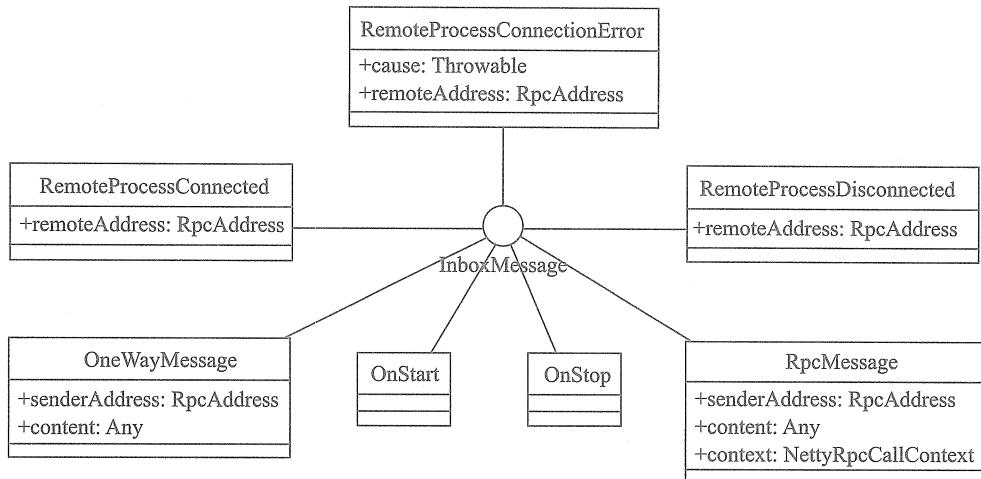


图 5-5 `InboxMessage` 的继承体系

图 5-5 展示了 `InboxMessage` 的多种实现类，其作用分别如下。

- `OneWayMessage`：`RpcEndpoint` 处理此类型的消息后不需要向客户端回复信息。
- `RpcMessage`：RPC 消息，`RpcEndpoint` 处理完此消息后需要向客户端回复信息。
- `OnStart`：用于 `Inbox` 实例化后，再通知与此 `Inbox` 相关联的 `RpcEndpoint` 启动。
- `OnStop`：用于 `Inbox` 停止后，通知与此 `Inbox` 相关联的 `RpcEndpoint` 停止。
- `RemoteProcessConnected`：此消息用于告诉所有的 `RpcEndpoint`，有远端的进程已经与当前 RPC 服务建立了连接。
- `RemoteProcessDisconnected`：此消息用于告诉所有的 `RpcEndpoint`，有远端的进程已经与当前 RPC 服务断开了连接。
- `RemoteProcessConnectionError`：此消息用于告诉所有的 `RpcEndpoint`，与远端某个地址之间的连接发生了错误。
- `Inbox`：端点内的盒子。每个 `RpcEndpoint` 都有一个对应的盒子，这个盒子里有个存储 `InboxMessage` 消息的列表 `messages`。所有的消息将缓存在 `messages` 列表里面，并由 `RpcEndpoint` 异步处理这些消息。
- `EndpointData`：RPC 端点数据，它包括了 `RpcEndpoint`、`NettyRpcEndpointRef` 及 `Inbox` 等属于同一个端点的实例。`Inbox` 与 `RpcEndpoint`、`NettyRpcEndpointRef` 通过

此 EndpointData 相关联。

有了对以上概念的介绍，现在来看看 Dispatcher 中的一些成员变量。

- ❑ endpoints：端点实例名称与端点数据 EndpointData 之间映射关系的缓存。有了这个缓存，就可以使用端点名称从中快速获取或删除 EndpointData 了。
- ❑ endpointRefs：端点实例 RpcEndpoint 与端点实例引用 RpcEndpointRef 之间映射关系的缓存。有了这个缓存，就可以使用端点实例从中快速获取或删除端点实例引用了。
- ❑ receivers：存储端点数据 EndpointData 的阻塞队列。只有 Inbox 中有消息的 EndpointData 才会被放入此阻塞队列。
- ❑ stopped：Dispatcher 是否停止的状态。
- ❑ threadpool：用于对消息进行调度的线程池。此线程池运行的任务都是 MessageLoop（将在代码清单 5-4、代码清单 5-5 中详细介绍）。

有了对 Dispatcher 涉及概念及属性的了解，我们就可以更容易理解 Dispatcher 的内存模型了，如图 5-6 所示。

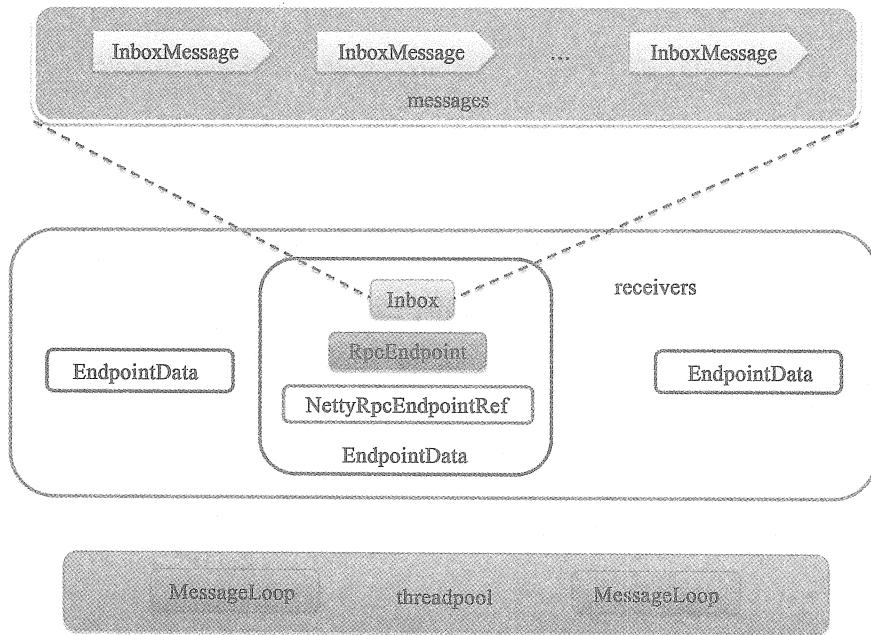


图 5-6 Dispatcher 的内存模型

## 2. Dispatcher 的调度原理

在创建 Dispatcher 的最后会创建对消息进行调度的线程池 threadpool，如代码清单 5-4 所示。

**代码清单5-4 创建Dispatcher中的线程池**


---

```

private val threadpool: ThreadPoolExecutor = {
    val numThreads = nettyEnv.conf.getInt("spark.rpc.netty.dispatcher.numThreads",
        math.max(2, Runtime.getRuntime.availableProcessors()))
    val pool = ThreadUtils.newDaemonFixedThreadPool(numThreads, "dispatcher-event-
        loop")
    for (i <- 0 until numThreads) {
        pool.execute(new MessageLoop)
    }
    pool
}

```

---

根据代码清单 5-4，可以看到创建 threadpool 线程池的步骤如下。

- 1) 获取此线程池的大小 numThreads。此线程池的大小默认为 2 与当前系统可用处理器数量之间的最大值，也可以使用 spark.rpc.netty.dispatcher.numThreads 属性配置。
- 2) 创建线程池。此线程池是固定大小的线程池，并且启动的线程都以后台线程方式运行，且线程名以 dispatcher-event-loop 为前缀。
- 3) 启动多个运行 MessageLoop 任务的线程，这些线程的数量与 threadpool 线程池的大小相同。
- 4) 返回此线程池的引用。

MessageLoop 实现了 Java 的 Runnable 接口，其实现如代码清单 5-5 所示。

**代码清单5-5 MessageLoop的实现**


---

```

private class MessageLoop extends Runnable {
    override def run(): Unit = {
        try {
            while (true) {
                try {
                    val data = receivers.take()
                    if (data == PoisonPill) {
                        // Put PoisonPill back so that other MessageLoops can see it.
                        receivers.offer(PoisonPill)
                        return
                    }
                    data.inbox.process(Dispatcher.this)
                } catch {
                    case NonFatal(e) => logError(e.getMessage, e)
                }
            }
        } catch {
            case ie: InterruptedException => // exit
        }
    }
}

```

---

根据代码清单 5-5，我们看到 MessageLoop 果如其名，在循环过程中不断对新的消息进行处理。每次循环中的逻辑如下。

1) 从 receivers 中获取 EndpointData。receivers 中的 EndpointData，其 Inbox 的 messages 列表中肯定有了新的消息。换言之，只有 Inbox 的 messages 列表中有了新的消息，此 EndpointData 才会被放入 receivers 中。由于 receivers 是个阻塞队列，所以当 receivers 中没有 EndpointData 时，MessageLoop 线程会被阻塞。

2) 如果取到的 EndpointData 是“毒药”(PoisonPill)，那么此 MessageLoop 线程将退出(通过 return 语句)。这里有个动作就是将 PoisonPill 重新放入到 receivers 中，这是因为 threadpool 线程池极有可能不止一个 MessageLoop 线程，为了大家都“毒发身亡”，还需要把“毒药”放回到 receivers 中，这样其他“活着”的线程就会再次误食“毒药”，达到所有 MessageLoop 线程都结束的效果。

3) 如果取到的 EndpointData 不是“毒药”，那么调用 EndpointData 中 Inbox 的 process 方法对消息进行处理。

上文的 MessageLoop 任务实际是将消息交给 EndpointData 中 Inbox 的 process 方法处理的，但在正式介绍 process 之前，先来看看 Inbox 中的一些成员属性，这样有助于之后对 process 方法的理解。Inbox 中的成员属性如下。

- messages：消息列表。用于缓存需要由对应 RpcEndpoint 处理的消息，即与 Inbox 在同一 EndpointData 中的 RpcEndpoint。Inbox 构造的最后是一些有些古怪的代码，如代码清单 5-6 所示。

代码清单5-6 Inbox向自身的messages列表中放入OnStart消息

---

```
inbox.synchronized {
    messages.add(OnStart)
}
```

---

上述代码将向 Inbox 自身的 messages 列表中放入 OnStart 消息，为什么会有这么一段代码？此处先不解释这段代码的具体用途。

- stopped：Inbox 的停止状态。
- enableConcurrent：是否允许多个线程同时处理 messages 中的消息。
- numActiveThreads：激活线程的数量，即正在处理 messages 中消息的线程数量。

介绍完 Inbox 中的成员属性，我们一起来看看其 process 方法的实现，如代码清单 5-7 所示。

代码清单5-7 Inbox中的消息处理

---

```
def process(dispatcher: Dispatcher): Unit = {
    var message: InboxMessage = null
    inbox.synchronized {
        if (!enableConcurrent && numActiveThreads != 0) {
            return
        }
        message = messages.poll()
        if (message != null) {
```

```

        numActiveThreads += 1
    } else {
        return
    }
}
while (true) {
    safelyCall(endpoint) {
        message match {
            case RpcMessage(_sender, content, context) =>
                try {
                    endpoint.receiveAndReply(context).applyOrElse[Any, Unit](content, {
                        msg =>
                        throw new SparkException(s"Unsupported message $message from ${_sender}")
                    })
                } catch {
                    case NonFatal(e) =>
                        context.sendFailure(e)
                        // Throw the exception -- this exception will be caught by the
                        // safelyCall function.
                        // The endpoint's onError function will be called.
                        throw e
                }
            case OneWayMessage(_sender, content) =>
                endpoint.receive.applyOrElse[Any, Unit](content, { msg =>
                    throw new SparkException(s"Unsupported message $message from ${_sender}")
                })
            case OnStart =>
                endpoint.onStart()
                if (!endpoint.isInstanceOf[ThreadSafeRpcEndpoint]) {
                    inbox.synchronized {
                        if (!stopped) {
                            enableConcurrent = true
                        }
                    }
                }
            case OnStop =>
                val activeThreads = inbox.synchronized { inbox.numActiveThreads }
                assert(activeThreads == 1,
                    s"There should be only a single active thread but found
                    $activeThreads threads.")
                dispatcher.removeRpcEndpointRef(endpoint)
                endpoint.onStop()
                assert(isEmpty, "OnStop should be the last message")
            case RemoteProcessConnected(remoteAddress) =>
                endpoint.onConnected(remoteAddress)
            case RemoteProcessDisconnected(remoteAddress) =>
                endpoint.onDisconnected(remoteAddress)
            case RemoteProcessConnectionError(cause, remoteAddress) =>
        }
    }
}

```



numActiveThreads 减 1；如果 messages 已经没有消息要处理了，这说明当前线程无论如何也该返回并将 numActiveThreads 减 1。

 注意 第 1) 步、第 2) 步及第 4) 步位于 Inbox 的锁保护之下，是因为 messages 是普通的 java.util.LinkedList，LinkedList 本身不是线程安全的，所以为了增加并发安全性，需要通过同步保护。

### 3. Inbox 的消息来源

MessageLoop 线程的执行逻辑是不断地消费各个 EndpointData 中 Inbox 里的消息，但是 EndpointData 是何时放入 receivers 中的？Inbox 里的消息来自哪里？Dispatcher 中有很多完成这些功能的方法，让我们一起看看 Dispatcher 中与此相关的一些方法。

#### (1) 注册 RpcEndpoint

Dispatcher 的 registerRpcEndpoint 方法用于注册 RpcEndpoint，这个方法的副作用便可以将 EndpointData 放入 receivers，其实现如代码清单 5-9 所示。

代码清单 5-9 注册 RpcEndpoint

```
def registerRpcEndpoint(name: String, endpoint: RpcEndpoint): NettyRpcEndpointRef = {
    val addr = RpcEndpointAddress(nettyEnv.address, name)
    val endpointRef = new NettyRpcEndpointRef(nettyEnv.conf, addr, nettyEnv)
    synchronized {
        if (stopped) {
            throw new IllegalStateException("RpcEnv has been stopped")
        }
        if (endpoints.putIfAbsent(name, new EndpointData(name, endpoint,
            endpointRef)) != null) {
            throw new IllegalArgumentException(s"There is already an RpcEndpoint called
                $name")
        }
        val data = endpoints.get(name)
        endpointRefs.put(data.endpoint, data.ref)
        receivers.offer(data) // for the OnStart message
    }
    endpointRef
}
```

从代码清单 5-9 可以看到，注册 RpcEndpoint 的步骤如下。

- 1) 使用当前 RpcEndpoint 所在 NettyRpcEnv 的地址和 RpcEndpoint 的名称创建 RpcEndpointAddress 对象。
- 2) 创建 RpcEndpoint 的引用对象——NettyRpcEndpointRef。
- 3) 创建 EndpointData，并放入 endpoints 缓存。
- 4) 将 RpcEndpoint 与 NettyRpcEndpointRef 的映射关系放入 endpointRefs 缓存。
- 5) 将 EndpointData 放入阻塞队列 receivers 的队尾。MessageLoop 线程异步获取到此 EndpointData，并处理其 Inbox 中刚刚放入的 OnStart 消息，最终调用 RpcEndpoint 的

OnStart 方法在 RpcEndpoint 开始处理消息之前做一些准备工作。

**小贴士：**由于 EndpointData 是新创建的，所以在构造 EndpointData 的过程中也会构造它内部的 Inbox。因此 Inbox 在构造的最后会执行代码清单 5-6 所示的逻辑，向 Inbox 自身的 messages 列表放入 OnStart 消息。随后在注册 RpcEndpoint 的第 5) 步，当前 Inbox 所属的 EndpointData 就被放入 receivers 中。最后 MessageLoop 线程就会取出此 EndpointData 处理，进而调用当前 Inbox 的 process 方法处理 OnStart。Inbox 的 process 方法处理 OnStart 的效果是在 Inbox 自身初始化完毕后，再启动与此 Inbox 相关联的 RpcEndpoint（可以参见代码清单 5-7 中对 OnStart 消息匹配后的执行过程）。

6) 返回 NettyRpcEndpointRef。

(2) 对 RpcEndpoint 去注册

Dispatcher 的 stop 方法（见代码清单 5-10）用于对 RpcEndpoint 的注册。

代码清单5-10 Dispatcher的stop方法

---

```
def stop(rpcEndpointRef: RpcEndpointRef): Unit = {
    synchronized {
        if (stopped) {
            return
        }
        unregisterRpcEndpoint(rpcEndpointRef.name)
    }
}
```

---

读者可能对 stop 这个方法的命名有些怀疑，不应该是停止 Dispatcher 的吗？阅读了代码清单 5-10 中 stop 方法的实现后，你就会释然。代码清单 5-10 中 stop 方法的执行步骤为：首先判断 Dispatcher 是否已经停止，如果 Dispatcher 未停止，则调用 Dispatcher 的 unregisterRpcEndpoint 方法对 RpcEndpoint 去注册。

Dispatcher 的 unregisterRpcEndpoint 方法用于对 RpcEndpoint 去注册，这个方法的副作用也可以将 EndpointData 放入 receivers，其实现如代码清单 5-11 所示。

代码清单5-11 对RpcEndpoint去注册

---

```
private def unregisterRpcEndpoint(name: String): Unit = {
    val data = endpoints.remove(name)
    if (data != null) {
        data.inbox.stop()
        receivers.offer(data) // for the OnStop message
    }
}
```

---

从代码清单 5-11 可以看到，去注册的步骤如下。

- 1) 从 endpoints 中移除 EndpointData。
- 2) 调用 EndpointData 中 Inbox 的 stop 方法停止 Inbox。

3) 将 EndpointData 重新放入 receivers 中。

代码本身很简单，可是奇怪的是为什么 EndpointData 从 endpoints 中移除后，最后还要放入 receivers？EndpointData 虽然移除了，但是对应的 RpcEndpointRef 并没有从 endpointRefs 缓存中移除，这又是为什么？这里先不直接回答这两个问题。

当要移除一个 EndpointData 时，其 Inbox 可能正在对消息进行处理，所以不能贸然停止。这里采用了更平滑的停止方式，即调用了 Inbox 的 stop 方法来平滑过渡，stop 方法的实现如代码清单 5-12 所示。

代码清单5-12 Inbox的stop方法

---

```
def stop(): Unit = inbox.synchronized {
    if (!stopped) {
        enableConcurrent = false
        stopped = true
        messages.add(OnStop)
    }
}
```

---

Inbox 的 stop 方法的处理步骤如下。

1) 根据之前的分析，MessageLoop 有“允许并发运行”和“不允许并发运行”两种情况。对于允许并发的情况，为了确保安全，应该将 enableConcurrent 设置为 false。

2) 设置当前 Inbox 为停止状态。

3) 向 messages 中添加 OnStop 消息。根据代码清单 5-5 中对 MessageLoop 的分析，为了能够处理 OnStop 消息，只有 Inbox 所属的 EndpointData 放入 receivers 中，其 messages 列表中的消息才会被处理，这回答了刚才提出的第一个问题。为了实现平滑停止，OnStop 消息最终将匹配代码清单 5-7，调用 Dispatcher 的 removeRpcEndpointRef 方法，将 RpcEndpoint 与 RpcEndpointRef 的映射从缓存 endpointRefs 中移除，这回答了刚才的第二个问题。在匹配执行 OnStop 消息的最后，将调用 RpcEndpoint 的 onStop 方法对 RpcEndpoint 停止。

(3) 将消息提交给指定的 RpcEndpoint

Dispatcher 的 postMessage 用于将消息提交给指定的 RpcEndpoint，其实现如代码清单 5-13 所示。

代码清单5-13 postMessage的实现

---

```
private def postMessage(
    endpointName: String,
    message: InboxMessage,
    callbackIfStopped: (Exception) => Unit): Unit = {
    val error = synchronized {
        val data = endpoints.get(endpointName)
        if (stopped) {
            Some(new RpcEnvStoppedException())
        } else if (data == null) {
            Some(new SparkException(s"Could not find $endpointName."))
        } else {
```

```

        data.inbox.post(message)
        receivers.offer(data)
        None
    }
}
// We don't need to call 'onStop' in the 'synchronized' block
error.foreach(callbackIfStopped)
}

```

---

从代码清单 5-13 可以看出，postMessage 方法的处理步骤如下。

- 1) 根据端点名称 endpointName 从缓存 endpoints 中获取 EndpointData。
- 2) 如果当前 Dispatcher 没有停止并且缓存 endpoints 中确实存在名为 endpointName 的 EndpointData，那么将调用 EndpointData 对应 Inbox 的 post 方法将消息加入 Inbox 的消息列表中，因此还需要将 EndpointData 推入 receivers，以便 MessageLoop 处理此 Inbox 中的消息。Inbox 的 post 方法的实现如代码清单 5-14 所示，其逻辑为 Inbox 未停止时向 messages 列表加入消息。

代码清单5-14 Inbox的post方法

```

def post(message: InboxMessage): Unit = inbox.synchronized {
    if (stopped) {
        // We already put "OnStop" into "messages", so we should drop further
        // messages
        onDrop(message)
    } else {
        messages.add(message)
        false
    }
}

```

---

此外，Dispatcher 中还有一些方法间接使用了 Dispatcher 的 postMessage 方法，如代码清单 5-15 所示。

代码清单5-15 对postMessage方法的间接使用

```

def postToAll(message: InboxMessage): Unit = {
    val iter = endpoints.keySet().iterator()
    while (iter.hasNext) {
        val name = iter.next
        postMessage(name, message, (e) => logWarning(s"Message $message dropped. ${e.getMessage}"))
    }
}

/** Posts a message sent by a remote endpoint. */
def postRemoteMessage(message: RequestMessage, callback: RpcResponseCallback): Unit = {
    val rpcCallContext =
        new RemoteNettyRpcCallContext(nettyEnv, callback, message.senderAddress)
    val rpcMessage = RpcMessage(message.senderAddress, message.content, rpcCallContext)
    postMessage(message.receiver.name, rpcMessage, (e) => callback.onFailure(e))
}

```

```

}

/** Posts a message sent by a local endpoint. */
def postLocalMessage(message: RequestMessage, p: Promise[Any]): Unit = {
  val rpcCallContext =
    new LocalNettyRpcCallContext(message.senderAddress, p)
  val rpcMessage = RpcMessage(message.senderAddress, message.content, rpcCallContext)
  postMessage(message.receiver.name, rpcMessage, (e) => p.tryFailure(e))
}

/** Posts a one-way message. */
def postOneWayMessage(message: RequestMessage): Unit = {
  postMessage(message.receiver.name, OneWayMessage(message.senderAddress, message.
    content),
    (e) => throw e)
}

```

代码清单 5-15 中的方法除了 postOneWayMessage 方法，其他需要回复的消息中都封装了 RpcCallContext。RpcCallContext 是用于回调客户端的上下文，其定义如代码清单 5-16 所示。

代码清单5-16 RpcCallContext的定义

---

```

private[spark] trait RpcCallContext {
  def reply(response: Any): Unit
  def sendFailure(e: Throwable): Unit
  def senderAddress: RpcAddress
}

```

---

代码清单 5-16 中的 RpcCallContext 一共定义了三个接口。

- reply：用于向发送者回复信息。
- sendFailure：用于向发送者发送失败信息。
- senderAddress：用于获取发送者的地址。

特质 RpcCallContext 的继承体系如图 5-7 所示。

#### (4) 停止 Dispatcher

Dispatcher 的 stop 方法用来停止 Dispatcher，其实现如代码清单 5-17 所示。

代码清单5-17 停止Dispatcher

---

```

def stop(): Unit = {
  synchronized {
    if (stopped) {
      return
    }
    stopped = true
  }
  // Stop all endpoints. This will queue all endpoints for processing by the
  // message loops.
  endpoints.keySet().asScala.foreach(unregisterRpcEndpoint)
  // Enqueue a message that tells the message loops to stop.
  receivers.offer(PoisonPill)
  threadpool.shutdown()
}

```

{}

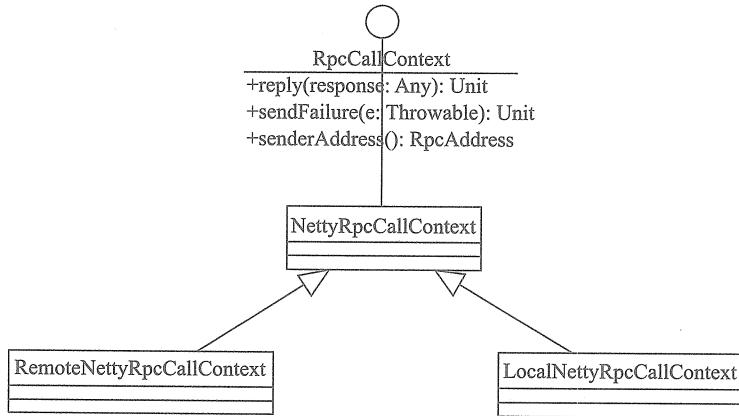


图 5-7 RpcCallContext 的继承体系

代码清单 5-17 中 stop 方法的执行步骤如下。

- 1) 如果 Dispatcher 还未停止，则将自身状态修改为已停止。
- 2) 对 endpoints 中的所有 EndpointData 去注册。这里通过调用 unregisterRpcEndpoint 方法（见代码清单 5-11），将向 endpoints 中的每个 EndpointData 的 Inbox 里放置 OnStop 消息。
- 3) 向 receivers 中投放“毒药”。



由于 Dispatcher 都停止了，所以 Dispatcher 中的所有 MessageLoop 线程也就没有存在的必要了。

- 4) 关闭 threadpool 线程池。

经过对 Dispatcher 和 MessageLoop 的分析，我们现在可以对图 5-6 中展现的内容进行扩展，增加一些运行时的执行流程，如图 5-8 所示。

图 5-8 中的序号代表其中的简要步骤，具体解释如下。

- 序号①表示调用 Inbox 的 post 方法将消息放入 messages 列表中。
- 序号②表示将有消息的 Inbox 相关联的 EndpointData 放入 receivers。
- 序号③表示 MessageLoop 每次循环首先从 receivers 中获取 EndpointData。
- 序号④表示执行 EndpointData 中 Inbox 的 process 方法对消息进行具体处理。

### 5.3.5 创建传输上下文 TransportContext

创建传输上下文 TransportContext 是 NettyRpcEnv 提供服务端与客户端能力的前提。创建 TransportContext 的代码如下。

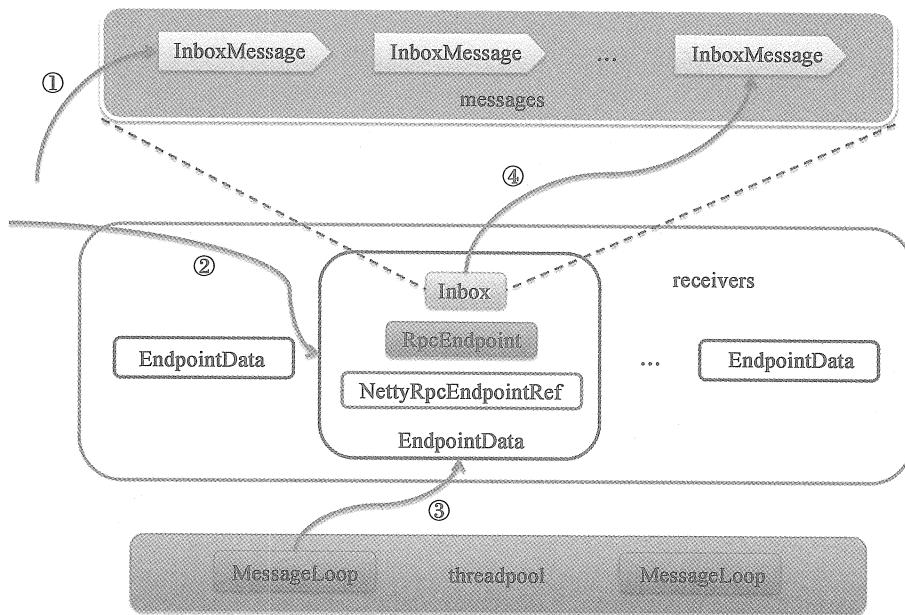


图 5-8 Dispatcher 中 MessageLoop 线程的执行流程

```
private val streamManager = new NettyStreamManager(this)
private val transportContext = new TransportContext(transportConf,
    new NettyRpcHandler(dispatcher, this, streamManager))
```

关于 TransportContext，在3.2节已经有足够多的介绍，所以这里重点来看看其构造器中传入的 RpcHandler 参数。这里用于构造 TransportContext 的 RpcHandler 实际是其实现类 NettyRpcHandler，NettyRpcHandler 的构造器里则以刚刚新建的 NettyStreamManager 实例作为参数。本节将首先介绍 NettyStreamManager，然后来看看 NettyRpcHandler 的实现。

### 1. NettyStreamManager 详解

NettyStreamManager 实现了 StreamManager，专用于为 NettyRpcEnv 提供文件服务的能力。创建 StreamManager 的代码如下。

```
private val streamManager = new NettyStreamManager(this)
```

从上面代码可以看出，此处创建的 streamManager 的实际类型为 NettyStreamManager。NettyStreamManager 用于提供 NettyRpcEnv 的文件流服务，这替代了在 Spark 1.x.x 版本中使用 Jetty 实现的 HttpFileServer。在 3.2.6 节的代码清单 3-27 中我们曾经介绍过，RpcHandler 的 getStreamManager 方法返回的类型为 StreamManager。NettyStreamManager 正是继承了 StreamManager。NettyStreamManager 中定义了三个文件与目录缓存，如代码清单 5-18 所示。

## 代码清单5-18 NettyStreamManager中的文件与目录缓存

---

```
private val files = new ConcurrentHashMap[String, File]()
private val jars = new ConcurrentHashMap[String, File]()
private val dirs = new ConcurrentHashMap[String, File]()
```

---

根据这三个缓存结构，我们猜想 NettyStreamManager 应该可以提供对普通文件、Jar 文件及目录的下载支持。NettyStreamManager 中一共提供了两类方法，一类用于添加缓存，一类用于支持文件流的读取。添加缓存的三个方法（如代码清单 5-19 所示）。

## 代码清单5-19 添加缓存的方法

---

```
override def addFile(file: File): String = {
    val existingPath = files.putIfAbsent(file.getName, file)
    require(existingPath == null || existingPath == file,
        s"File ${file.getName} was already registered with a different path " +
        s"(old path = $existingPath, new path = $file")
    s"${rpcEnv.address.toSparkURL}/files/${Utils.encodeFileNameToURIRawPath(file.
        getName())}"
}

override def addJar(file: File): String = {
    val existingPath = jars.putIfAbsent(file.getName, file)
    require(existingPath == null || existingPath == file,
        s"File ${file.getName} was already registered with a different path " +
        s"(old path = $existingPath, new path = $file")
    s"${rpcEnv.address.toSparkURL}/jars/${Utils.encodeFileNameToURIRawPath(file.
        getName())}"
}

override def addDirectory(baseUri: String, path: File): String = {
    val fixedBaseUri = validateDirectoryUri(baseUri)
    require(dirs.putIfAbsent(fixedBaseUri.stripPrefix("/"), path) == null,
        s"URI '$fixedBaseUri' already registered.")
    s"${rpcEnv.address.toSparkURL}$fixedBaseUri"
}
```

---

代码清单 4-22 中提到的 fileServer 实际就是 NettyStreamManager。由于添加缓存的三个方法都比较简单，所以不再过多介绍。

NettyStreamManager 提供的用于打开文件流的方法如代码清单 5-20 所示。

## 代码清单5-20 打开文件流

---

```
override def openStream(streamId: String): ManagedBuffer = {
    val Array(ftype, fname) = streamId.stripPrefix("/").split("/", 2)
    val file = ftype match {
        case "files" => files.get(fname)
        case "jars" => jars.get(fname)
        case other =>
            val dir = dirs.get(ftype)
            require(dir != null, s"Invalid stream URI: $ftype not found.")
            new File(dir, fname)
    }
}
```

---

---

```

if (file != null && file.isFile()) {
    new FileSegmentManagedBuffer(rpcEnv.transportConf, file, 0, file.length())
} else {
    null
}
}

```

---

openStream 方法的逻辑很简单，从缓存中获取文件后，将 TransportConf 及 File 等信息封装为 FileSegmentManagedBuffer 并返回。我们之前在代码清单 3-24、代码清单 3-25 及代码清单 3-26 中介绍过 FileSegmentManagedBuffer 的实现，我们知道 FileSegmentManagedBuffer 封装了对文件的流处理细节。由于 NettyStreamManager 只实现了 Stream Manager 的 openStream 方法，根据 3.2.6 节对 TransportRequestHandler 的 handle 方法和 process StreamRequest 方法的介绍，我们知道 NettyStreamManager 将只提供对 StreamRequest 类型消息的处理。各个 Executor 节点就可以使用 Driver 节点的 RpcEnv 提供的 NettyStreamManager，从 Driver 将 Jar 包或文件下载到 Executor 节点上供任务执行。

## 2. NettyRpcHandler 详解

上文谈到 NettyRpcEnv 中用于构造 TransportContext 的 RpcHandler 实际是其实现类 NettyRpcHandler。结合代码清单 3-27 中对 RpcHandler 的定义，我们选择两个重载的 receive 方法来看看 NettyRpcHandler 是如何实现 RpcHandler 的。

- 对客户端进行响应的 receive 方法（见代码清单 5-21）的分析。

代码清单 5-21 NettyRpcHandler 的 receive 方法

---

```

override def receive(
    client: TransportClient,
    message: ByteBuffer,
    callback: RpcResponseCallback): Unit = {
    val messageToDispatch = internalReceive(client, message)
    dispatcher.postRemoteMessage(messageToDispatch, callback)
}

```

---

可以看到，代码清单 5-21 中 receive 方法的处理步骤如下。

- 1) 调用 internalReceive 方法将 ByteBuffer 类型的 message 转换为 RequestMessage。
- 2) 调用 Dispatcher 的 postRemoteMessage 方法（见代码清单 5-15）将消息转换为 RpcMessage 后放入 Inbox 的消息列表。根据代码清单 5-7，MessageLoop 将调用 RpcEndPoint 实现类的 receiveAndReply 方法，即 RpcEndPoint 处理完消息后会向客户端进行回复。

internalReceive 方法可以将 ByteBuffer 类型的 message 转换为 RequestMessage，其实现如代码清单 5-22 所示。

代码清单 5-22 internalReceive 的实现

---

```

private def internalReceive(client: TransportClient, message: ByteBuffer): RequestMessage = {
    val addr = client.getChannel().remoteAddress().asInstanceOf[InetSocketAddress]

```

---

```

assert(addr != null)
val clientAddr = RpcAddress(addr.getHostString, addr.getPort)
val requestMessage = nettyEnv.deserialize[RequestMessage](client, message)
if (requestMessage.senderAddress == null) {
    RequestMessage(clientAddr, requestMessage.receiver, requestMessage.content)
} else {
    val remoteEnvAddress = requestMessage.senderAddress
    if (remoteAddresses.putIfAbsent(clientAddr, remoteEnvAddress) == null) {
        dispatcher.postToAll(RemoteProcessConnected(remoteEnvAddress))
    }
    requestMessage
}
}

```

---

根据代码清单 5-22，internalReceive 的执行步骤如下。

- 1 ) 从 TransportClient 中获取远端地址 RpcAddress。
- 2 ) 调用 NettyRpcEnv 的 deserialize 方法对客户端发送的序列化后的消息（即 ByteBuffer 类型的消息）进行反序列化，根据 deserialize 的实现（见代码清单 5-23），反序列化实际使用了 javaSerializerInstance。javaSerializerInstance 是通过 NettyRpcEnv 的构造参数传入的对象，类型为 JavaSerializerInstance（请参阅代码清单 5-1）。

**代码清单5-23 反序列化ByteBuffer类型的消息**

```

private[netty] def deserialize[T: ClassTag](client: TransportClient, bytes: ByteBuffer): T = {
    NettyRpcEnv.currentClient.withValue(client) {
        deserialize { () =>
            javaSerializerInstance.deserialize[T](bytes)
        }
    }
}

```

---

- 3 ) 如果反序列化得到的请求消息 requestMessage 中没有发送者的地址信息，则使用从 TransportClient 中获取的远端地址 RpcAddress、requestMessage 的接收者（即 RpcEndpoint）、requestMessage 的内容，以构造新的 RequestMessage。

- 4 ) 如果反序列化得到的请求消息 requestMessage 中含有发送者的地址信息，则将从 TransportClient 中获取的远端地址 RpcAddress 与 requestMessage 中的发送者地址信息之间的映射关系放入缓存 remoteAddresses。还将调用 Dispatcher 的 postToAll 方法（见代码清单 5-15）向 endpoints 缓存的所有 EndpointData 的 Inbox 中放入 RemoteProcessConnected 消息<sup>Θ</sup>。最后将返回 requestMessage。

- 对客户端进行响应的 receive 重载方法（其实现见代码清单 5-24）的分析。

<sup>Θ</sup> 根据代码清单 5-7 中 Inbox 的 process 方法对 RemoteProcessConnected 消息的匹配，将执行所有 RpcEndpoint 的 onConnected 方法。特质 RpcEndpoint 中的 onConnected 是个空方法，所以 RpcEndpoint 的具体实现类可以针对远端客户端连接到服务端设计一些定制化的功能。

代码清单5-24 NettyRpcHandler的receive重载方法

---

```
override def receive(
    client: TransportClient,
    message: ByteBuf): Unit = {
    val messageToDispatch = internalReceive(client, message)
    dispatcher.postOneWayMessage(messageToDispatch)
}
```

---

根据 3.2.6 节对 RpcHandler 实现的分析，我们知道此方法不会对客户端进行回复。此方法也调用了 internalReceive 方法，但是最后向 EndpointData 的 Inbox 投递消息使用了 postOneWayMessage 方法（见代码清单 5-15），将消息转换为 OneWayMessage 后放入 Inbox 的消息列表。根据代码清单 5-7，将调用 RpcEndPoint 实现类的 receive 方法，也就是说 RpcEndPoint 处理完消息后不会向客户端进行回复。

由于 NettyRpcHandler 实现了 RpcHandler 的两个 receive 方法，所以根据 3.2.6 节 TransportRequestHandler 的 handle 方法、processRpcRequest 方法及 processOneWayMessage 方法的介绍，我们知道 NettyRpcHandler 将提供对 OneWayMessage 和 RpcRequest 两种类型消息的处理。

NettyRpcHandler 除实现了 RpcHandler 的两个 receive 方法，还实现了 exceptionCaught、channelActive 与 channelInactive 等。exceptionCaught 方法将会向 Inbox 中投递 RemoteProcessConnectionError 消息。channelActive 将会向 Inbox 中投递 RemoteProcessConnected。channelInactive 将会向 Inbox 中投递 RemoteProcessDisconnected 消息。这几个方法的处理都与 receive 方法类似，感兴趣的读者可以自行分析，此处不再赘述。

### 5.3.6 创建传输客户端工厂 TransportClientFactory

创建传输客户端工厂 TransportClientFactory 是 NettyRpcEnv 向远端服务发起请求的基础，Spark 与远端 RpcEnv 进行通信都依赖于 TransportClientFactory 生产的 TransportClient。NettyRpcEnv 中共创建了两个 TransportClientFactory，代码如下。

```
private def createClientBootsraps(): java.util.List[TransportClientBootstrap] =
{
    if (securityManager.isAuthenticationEnabled()) {
        java.util.Arrays.asList(new SaslClientBootstrap(transportConf, "", securityManager,
            securityManager.isSaslEncryptionEnabled()))
    } else {
        java.util.Collections.emptyList[TransportClientBootstrap]
    }
}
private val clientFactory = transportContext.createClientFactory(createClientBootsraps())
@volatile private var fileDownloadFactory: TransportClientFactory = _
```

这里的 clientFactory 用于常规的发送请求和接收响应。fileDownloadFactory 则用于文件下载。由于有些 RpcEnv 本身并不需要从远端下载文件，所以这里只声明了变量 fileDownloadFactory，并未进一步对其初始化。需要下载文件的 RpcEnv 会调用 downloadClient 方法创

建 `TransportClientFactory`，并用此 `TransportClientFactory` 创建下载所需的传输客户端 `TransportClient`。`downloadClient` 的创建见代码清单 5-25。

代码清单 5-25 获取下载的 `TransportClient`

---

```

private def downloadClient(host: String, port: Int): TransportClient = {
    if (fileDownloadFactory == null) synchronized {
        if (fileDownloadFactory == null) {
            val module = "files"
            val prefix = "spark.rpc.io."
            val clone = conf.clone()

            // Copy any RPC configuration that is not overridden in the spark.files
            // namespace.
            conf.getAll.foreach { case (key, value) =>
                if (key.startsWith(prefix)) {
                    val opt = key.substring(prefix.length())
                    clone.setIfMissing(s"spark.$module.io.$opt", value)
                }
            }

            val ioThreads = clone.getInt("spark.files.io.threads", 1)
            val downloadConf = SparkTransportConf.fromSparkConf(clone, module,
                ioThreads)
            val downloadContext = new TransportContext(downloadConf, new
                NoOpRpcHandler(), true)
            fileDownloadFactory = downloadContext.createClientFactory(createClientBoot
                straps())
        }
    }
    fileDownloadFactory.createClient(host, port)
}

```

---

根据代码清单 5-25，我们知道 `fileDownloadFactory` 与 `clientFactory` 使用的 `SparkTransportConf` 内部代理的 `SparkConf` 都是从 `NettyRpcEnv` 的 `SparkConf` 克隆来的，不同之处在于 `clientFactory` 所属的模块（`module` 变量）为 `rpc`，`fileDownloadFactory` 所属的模块为 `files`。`clientFactory` 中的读写线程数由 `spark.rpc.io.numConnectionsPerPeer` 属性控制，而 `fileDownloadFactory` 中的读写线程数由 `spark.files.io.threads` 属性控制。有关 `SparkTransportConf.fromSparkConf` 方法在 3.2.1 节已经介绍过，`TransportContext` 的 `createClientFactory` 方法的内容也在 3.2.2 节详细说明，故此处不再赘述。

### 5.3.7 创建 `TransportServer`

作为一个 RPC 环境，`NettyRpcEnv` 不应该只具有向远端服务发起请求并接收响应的能力，也应当对外提供接收请求、处理请求、回复客户端的服务。`NettyRpcEnv` 中创建 `TransportServer` 的代码如下<sup>Θ</sup>。

---

<sup>Θ</sup> 这里顺带列出了 `NettyRpcEnv` 的状态属性。

```
@volatile private var server: TransportServer = _
private val stopped = new AtomicBoolean(false)
```

可以看到，TransportServer 在此时并未实例化，那么它是何时实例化的？在代码清单 5-1 中我们介绍过启动 RpcEnv 的偏函数 startNettyRpcEnv，startNettyRpcEnv 将负责回调 NettyRpcEnv 的 startServer 方法，startServer 的实现如代码清单 5-26 所示。

代码清单5-26 启动TransportServer

---

```
def startServer(bindAddress: String, port: Int): Unit = {
    val bootstraps: java.util.List[TransportServerBootstrap] =
        if (securityManager.isAuthenticationEnabled()) {
            java.util.Arrays.asList(new SaslServerBootstrap(transportConf, securityManager))
        } else {
            java.util.Collections.emptyList()
        }
    server = transportContext.createServer(bindAddress, port, bootstraps)
    dispatcher.registerRpcEndpoint(
        RpcEndpointVerifier.NAME, new RpcEndpointVerifier(this, dispatcher))
}
```

---

根据代码清单 5-26，启动 NettyRpcEnv 的步骤如下。

1) 创建 TransportServer。这里使用了 TransportContext 的 createServer 方法，此方法已经在 3.2.3 节详细介绍，不再赘述。

2) 向 Dispatcher 注册 RpcEndpointVerifier。RpcEndpointVerifier 用于校验指定名称的 RpcEndpoint 是否存在。RpcEndpointVerifier 在 Dispatcher 中的注册名为 endpoint-verifier，其实现如代码清单 5-27 所示。registerRpcEndpoint 在代码清单 5-9 有详细分析，读者可以回顾。

代码清单5-27 RpcEndpointVerifier的实现

---

```
private[netty] class RpcEndpointVerifier(override val rpcEnv: RpcEnv, dispatcher: Dispatcher)
    extends RpcEndpoint {

    override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
        case RpcEndpointVerifier.CheckExistence(name) => context.reply(dispatcher.verify(name))
    }
}

private[netty] object RpcEndpointVerifier {
    val NAME = "endpoint-verifier"
    case class CheckExistence(name: String)
}
```

---

可以看到 RpcEndpointVerifier 实现了 RpcEndpoint 的 receiveAndReply 方法。因此 MessageLoop 线程在处理 RpcEndpointVerifier 所关联的 Inbox 中的消息时，会匹配 RpcMessage 调用 RpcEndpointVerifier 的 receiveAndReply 方法。RpcEndpointVerifier 实现的

receiveAndReply 方法的处理步骤如下。

1) 接收 CheckExistence 类型的消息，匹配出 name 参数，此参数代表要查询的 RpcEndpoint 的具体名称。

2) 调用 Dispatcher 的 verify 方法。verify 用于校验 Dispatcher 的 endpoints 缓存中是否存在名为 name 的 RpcEndpoint，其实现如代码清单 5-28 所示。

3) 调用 RpcCallContext 的 reply 方法回复客户端。RpcCallContext 的定义如代码清单 5-16 所示。

代码清单5-28 Dispatcher的verify方法

---

```
def verify(name: String): Boolean = {
    endpoints.containsKey(name)
}
```

---

根据对 RpcEndpointVerifier 的实现分析，我们知道它对外提供了查询当前 RpcEndpointVerifier 所在 RpcEnv 的 Dispatcher 中是否存在请求中指定名称所对应的 Rpc-Endpoint。

TransportServer 初始化并且启动后，就可以利用 NettyRpcHandler 和 NettyStreamManager 对外提供服务了。

### 5.3.8 客户端请求发送

之前已经介绍了 NettyRpcHandler 和 NettyStreamManager 提供的服务端实现，现在我们来看看如何向远端 RpcEndpoint 发送消息。

当 TransportClient 发出请求之后，会等待获取服务端的回复，这就涉及超时问题。另外由于 TransportClientFactory.createClient 方法是阻塞式调用，所以需要一个异步的处理。NettyRpcEnv 中实现这些需求的代码如下。

```
val timeoutScheduler = ThreadUtils.newDaemonSingleThreadScheduledExecutor("netty-rpc-env-timeout")
private[netty] val clientConnectionExecutor = ThreadUtils.newDaemonCachedThreadPool(
    "netty-rpc-connection",
    conf.getInt("spark.rpc.connect.threads", 64))
private val outboxes = new ConcurrentHashMap[RpcAddress, Outbox]()
```

上面代码中创建了与发送请求相关的三个组件，分别如下。

- timeoutScheduler：用于处理请求超时的调度器。timeoutScheduler 的类型实际是 ScheduledExecutorService，比起使用 Timer 组件，ScheduledExecutorService 将比 Timer 更加稳定，比如线程挂掉后，ScheduledExecutorService 会重启一个新的线程定时检查请求是否超时。
- clientConnectionExecutor：一个用于异步处理 TransportClientFactory.createClient 方法调用的线程池。这个线程池的大小默认为 64，可以使用 spark.rpc.connect.threads 属性进行配置。

- ❑ outboxes：RpcAddress 与 Outbox 的映射关系的缓存。每次向远端发送请求时，此请求消息首先放入此远端地址对应的 Outbox，然后使用线程异步发送。

### 1. Outbox 与 OutboxMessage

上文提到的 outboxes 缓存了远端 RPC 地址与 Outbox 的关系，那么 Outbox 是什么？为了说明 Outbox，不妨来看看 Outbox 中的成员属性。

- ❑ nettyEnv：当前 Outbox 所在节点上的 NettyRpcEnv。
- ❑ address：Outbox 所对应的远端 NettyRpcEnv 的地址。
- ❑ messages：向其他远端 NettyRpcEnv 上的所有 RpcEndpoint 发送的消息列表。
- ❑ client：当前 Outbox 内的 TransportClient。消息的发送都依赖于此传输客户端。
- ❑ connectFuture：指向当前 Outbox 内连接任务的 java.util.concurrent.Future 引用。如果当前没有连接任务，则 connectFuture 为 null。
- ❑ stopped：当前 Outbox 是否停止的状态。
- ❑ draining：表示当前 Outbox 内正有线程在处理 messages 列表中消息的状态。

消息列表 messages 中的消息类型为 OutboxMessage，所有将要向远端发送的消息都会被封装成 OutboxMessage 类型。OutboxMessage 作为一个特质，定义了所有向外发送消息的规范，其定义如代码清单 5-29 所示。

代码清单 5-29 OutboxMessage 的定义

---

```
private[netty] sealed trait OutboxMessage {
    def sendWith(client: TransportClient): Unit
    def onFailure(e: Throwable): Unit
}
```

---

根据 OutboxMessage 的名称，我们很容易与 Dispatcher 中 Inbox 里的 InboxMessage 类型的消息关联起来。OutboxMessage 在客户端使用，是对外发送消息的封装。InboxMessage 在服务端使用，是对所接收消息的封装。OutboxMessage 的继承体系如图 5-9 所示。

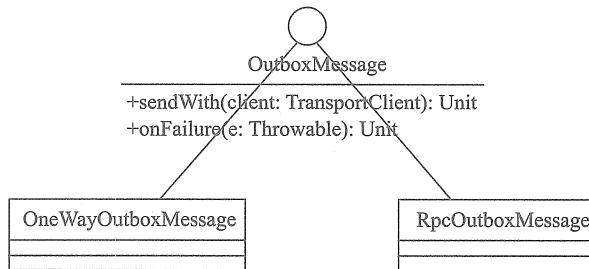


图 5-9 OutboxMessage 的继承体系

我们以 RpcOutboxMessage 为例，来看看 OutboxMessage 该如何实现，RpcOutboxMessage 的实现如代码清单 5-30 所示。

代码清单5-30 RpcOutboxMessage的实现

---

```

private[netty] case class RpcOutboxMessage(
    content: ByteBuffer,
    _onFailure: (Throwable) => Unit,
    _onSuccess: (TransportClient, ByteBuffer) => Unit)
extends OutboxMessage with RpcResponseCallback with Logging {

    private var client: TransportClient = _
    private var requestId: Long = _

    override def sendWith(client: TransportClient): Unit = {
        this.client = client
        this.requestId = client.sendRpc(content, this)
    }

    def onTimeout(): Unit = {
        if (client != null) {
            client.removeRpcRequest(requestId)
        } else {
            logError("Ask timeout before connecting successfully")
        }
    }

    override def onFailure(e: Throwable): Unit = {
        _onFailure(e)
    }

    override def onSuccess(response: ByteBuffer): Unit = {
        _onSuccess(client, response)
    }
}

```

---

根据代码清单 5-30，我们知道 `RpcOutboxMessage` 重写的 `sendWith` 方法正是利用了 `TransportClient` 的 `sendRpc` 方法（见 3.2.8 节）。`TransportClient` 的 `sendRpc` 方法的第二个参数是 `RpcResponseCallback` 类型，`RpcOutboxMessage` 本身也实现了 `RpcResponseCallback`，所以调用的时候传递了 `RpcOutboxMessage` 的 `this` 引用。

有了对 `Outbox` 中各个属性的介绍，现在一起来认识下 `Outbox` 的各个方法。`Outbox` 的发送消息的方法是最常用的方法，其实现如代码清单 5-31 所示。

代码清单5-31 Outbox的发送消息的方法

---

```

def send(message: OutboxMessage): Unit = {
    val dropped = synchronized {
        if (stopped) {
            true
        } else {
            messages.add(message)
            false
        }
    }
}

```

---

```

if (dropped) {
    message.onFailure(new SparkException("Message is dropped because Outbox is
        stopped"))
} else {
    drainOutbox()
}
}

```

---

根据代码清单 5-31，Outbox 的 send 方法的执行步骤如下。

- 1) 判断当前 Outbox 的状态是否已经停止。
- 2) 如果 Outbox 已经停止，则向发送者发送 SparkException 异常。如果 Outbox 还未停止，则将 OutboxMessage 添加到 messages 列表中，并且调用 drainOutbox 方法处理 messages 中的消息。drainOutbox 是一个私有方法，其实现如代码清单 5-32 所示。

代码清单5-32 drainOutbox的实现

```

private def drainOutbox(): Unit = {
    var message: OutboxMessage = null
    synchronized {
        if (stopped) {
            return
        }
        if (connectFuture != null) {
            return
        }
        if (client == null) {
            launchConnectTask()
            return
        }
        if (draining) {
            return
        }
        message = messages.poll()
        if (message == null) {
            return
        }
        draining = true
    }
    while (true) {
        try {
            val _client = synchronized { client }
            if (_client != null) {
                message.sendWith(_client)
            } else {
                assert(stopped == true)
            }
        } catch {
            case NonFatal(e) =>
                handleNetworkFailure(e)
                return
        }
        synchronized {
            if (stopped) {

```

```
        return
    }
    message = messages.poll()
    if (message == null) {
        draining = false
        return
    }
}
}
}
}
```

根据代码清单 5-32，drainOutbox 的执行步骤如下。

- 1 ) 如果当前 Outbox 已经停止或者正在连接远端服务，则返回。
- 2 ) 如果 Outbox 中的 TransportClient 为 null，这说明还未连接远端服务。此时需要调用 launchConnectTask 方法运行连接远端服务的任务，然后返回。
- 3 ) 如果正有线程在处理（即发送）messages 列表中的消息，则返回。
- 4 ) 如果 messages 列表中没有消息要处理，则返回。否则取出其中的一条消息，并将 draining 状态置为 true。
- 5 ) 循环处理 messages 列表中的消息，即不断从 messages 列表中取出消息并调用 OutboxMessage 的 sendWith 方法发送消息。

在 drainOutbox 方法中调用 launchConnectTask 方法，运行连接远端服务的任务，其实现如代码清单 5-33 所示。

代码清单5-33 运行连接远端服务的任务

```
private def launchConnect(): Unit = {
    connectFuture = nettyEnv.clientConnectionExecutor.submit(new Callable[Unit] {

        override def call(): Unit = {
            try {
                val _client = nettyEnv.createClient(address)
                outbox.synchronized {
                    client = _client
                    if (stopped) {
                        closeClient()
                    }
                }
            } catch {
                case ie: InterruptedException =>
                    // exit
                    return
                case NonFatal(e) =>
                    outbox.synchronized { connectFuture = null }
                    handleNetworkFailure(e)
                    return
            }
            outbox.synchronized { connectFuture = null }
            // It's possible that no thread is draining now. If we don't drain here, we
            cannot send the
        }
    })
}
```

---

```
// messages until the next message arrives.
drainOutbox()
}
})
}
```

---

根据代码清单 5-33，launchConnectTask 方法的执行步骤如下。

- 1) 构造 Callable 的匿名内部类，此匿名类将调用 NettyRpcEnv 的 createClient 方法创建 TransportClient，然后调用 drainOutbox 方法处理 Outbox 中的消息。

- 2) 使用 NettyRpcEnv 中的 clientConnectionExecutor 提交 Callable 的匿名内部类。

NettyRpcEnv 的 createClient 方法的实现如代码清单 5-34 所示。

代码清单5-34 创建TransportClient

---

```
private[netty] def createClient(address: RpcAddress): TransportClient = {
    clientFactory.createClient(address.host, address.port)
}
```

---

从代码清单 5-34 中看到，NettyRpcEnv 的 createClient 方法实际使用了 TransportClientFactory 的 createClient 方法（见 3.2.2 节）。

## 2. NettyRpcEndpointRef 详解

在 NettyRpcEnv 中，要向远端 RpcEndpoint 发送请求，首先要持有 RpcEndpoint 的引用对象 NettyRpcEndpointRef（类似于 Akka 中 Actor 的 ActorRef）。NettyRpcEndpointRef 是 RpcEndpointRef 的唯一子类。NettyRpcEndpointRef 重写了 RpcEndpointRef 的部分方法，其实现如代码清单 5-35 所示。

代码清单5-35 NettyRpcEndpointRef的实现

---

```
private[netty] class NettyRpcEndpointRef(
    @transient private val conf: SparkConf,
    endpointAddress: RpcEndpointAddress,
    @transient @volatile private var nettyEnv: NettyRpcEnv)
    extends RpcEndpointRef(conf) with Serializable with Logging {

    @transient @volatile var client: TransportClient = _

    private val _address = if (endpointAddress.rpcAddress != null) endpointAddress
                           else null
    private val _name = endpointAddress.name

    override def address: RpcAddress = if (_address != null) _address.rpcAddress
                                         else null

    private def readObject(in: ObjectInputStream): Unit = {
        in.defaultReadObject()
        nettyEnv = NettyRpcEnv.currentEnv.value
        client = NettyRpcEnv.currentClient.value
    }
}
```

---

```

private def writeObject(out: ObjectOutputStream): Unit = {
    out.defaultWriteObject()
}

override def name: String = _name

override def ask[T: ClassTag](message: Any, timeout: RpcTimeout): Future[T] = {
    nettyEnv.ask(RequestMessage(nettyEnv.address, this, message), timeout)
}

override def send(message: Any): Unit = {
    require(message != null, "Message is null")
    nettyEnv.send(RequestMessage(nettyEnv.address, this, message))
}

override def toString: String = s"NettyRpcEndpointRef(${_address})"

def toURI: URI = new URI(_address.toString)

final override def equals(that: Any): Boolean = that match {
    case other: NettyRpcEndpointRef => _address == other._address
    case _ => false
}

final override def hashCode(): Int = if (_address == null) 0 else _address.hashCode()
}

```

根据代码清单 5-35，NettyRpcEndpointRef 包含了以下属性。

- client：类型为 TransportClient（TransportClient 的具体内容请参阅 3.2.8 节）。Netty-RpcEndpointRef 将利用此 TransportClient 向远端的 RpcEndpoint 发送请求。
- \_address：远端 RpcEndpoint 的地址 RpcEndpointAddress。
- \_name：远端 RpcEndpoint 的名称。

下面来介绍 NettyRpcEndpointRef 中重写的方法。

- address：返回 \_address 属性的值，或返回 null。
- name：返回 \_name 属性的值。
- ask[T: ClassTag](message: Any, timeout: RpcTimeout)：首先将 message 封装为 Request Message，然后调用 NettyRpcEnv 的 ask 方法。
- send(message: Any)：首先将 message 封装为 RequestMessage，然后调用 NettyRpcEnv 的 send 方法。

NettyRpcEndpointRef 的 ask 方法和 send 方法分别调用了 NettyRpcEnv 的 ask 方法和 send 方法，下面将对它们进行详细介绍。

### (1) 询问

NettyRpcEnv 重写了抽象类 RpcEnv 的 ask 方法，其实现如代码清单 5-36 所示。

## 代码清单5-36 询问方法ask的实现

```

private[netty] def ask[T: ClassTag](message: RequestMessage, timeout: RpcTimeout): Future[T] = {
    val promise = Promise[Any]()
    val remoteAddr = message.receiver.address

    def onFailure(e: Throwable): Unit = {
        if (!promise.tryFailure(e)) {
            logWarning(s"Ignored failure: $e")
        }
    }

    def onSuccess(reply: Any): Unit = reply match {
        case RpcFailure(e) => onFailure(e)
        case rpcReply =>
            if (!promise.trySuccess(rpcReply)) {
                logWarning(s"Ignored message: $reply")
            }
    }

    try {
        if (remoteAddr == address) {
            val p = Promise[Any]()
            p.future.onComplete {
                case Success(response) => onSuccess(response)
                case Failure(e) => onFailure(e)
            }(ThreadUtils.sameThread)
            dispatcher.postLocalMessage(message, p)
        } else {
            val rpcMessage = RpcOutboxMessage(serialize(message),
                onFailure,
                (client, response) => onSuccess(deserialize[Any](client, response)))
            postToOutbox(message.receiver, rpcMessage)
            promise.future.onFailure {
                case _: TimeoutException => rpcMessage.onTimeout()
                case _ =>
            }(ThreadUtils.sameThread)
        }
    }

    val timeoutCancelable = timeoutScheduler.schedule(new Runnable {
        override def run(): Unit = {
            onFailure(new TimeoutException(s"Cannot receive any reply in ${timeout.duration}"))
        }
    }, timeout.duration.toNanos, TimeUnit.NANOSECONDS)
    promise.future.onComplete { v =>
        timeoutCancelable.cancel(true)
    }(ThreadUtils.sameThread)
} catch {
    case NonFatal(e) =>
        onFailure(e)
}
promise.future.mapTo[T].recover(timeout.addMessageIfTimeout)(ThreadUtils.sameThread)
}

```

根据代码清单 5-36，NettyRpcEnv 的 ask 方法的执行步骤如下。

1) 如果请求消息的接收者的地址与当前 NettyRpcEnv 的地址相同（则说明处理请求的 RpcEndpoint 位于本地的 NettyRpcEnv 中），那么新建 Promise 对象，并且给 Promise 的 future（类型为 Future）设置完成时的回调函数（成功时调用 onSuccess 方法，失败时调用 onFailure 方法）。发送消息最终通过调用本地 Dispatcher 的 postLocalMessage 方法（见代码清单 5-15）实现。

2) 如果请求消息的接收者的地址与当前 NettyRpcEnv 的地址不同（则说明处理请求的 RpcEndpoint 位于其他节点的 NettyRpcEnv 中），那么将 message 序列化，并与 onFailure、onSuccess 方法一道封装为 RpcOutboxMessage 类型的消息。最后调用 postToOutbox 方法将消息投递出去。

3) 使用 timeoutScheduler 设置一个定时器，用于超时处理。此定时器在等待指定的超时时间后将抛出 TimeoutException 异常。请求如果在超时时间内处理完毕，则会调用 timeoutScheduler 的 cancel 方法取消此超时定时器。

4) 返回请求处理的结果。

#### (2) 发送消息

NettyRpcEnv 重写了抽象类 RpcEnv 的 send 方法，其实现如代码清单 5-37 所示。

代码清单5-37 send的实现

---

```
private[netty] def send(message: RequestMessage): Unit = {
    val remoteAddr = message.receiver.address
    if (remoteAddr == address) {
        // Message to a local RPC endpoint.
        try {
            dispatcher.postOneWayMessage(message)
        } catch {
            case e: RpcEnvStoppedException => logWarning(e.getMessage)
        }
    } else {
        // Message to a remote RPC endpoint.
        postToOutbox(message.receiver, OneWayOutboxMessage(serialize(message)))
    }
}
```

---

根据代码清单 5-37，NettyRpcEnv 的 send 方法的执行步骤如下。

1) 如果请求消息的接收者的地址与当前 NettyRpcEnv 的地址相同。<sup>⊖</sup>那么新建 Promise 对象，并且给 Promise 的 future（类型为 Future）设置完成时的回调函数（成功时调用 onSuccess 方法，失败时调用 onFailure 方法）。发送消息最终通过调用本地 Dispatcher 的 postOneWayMessage 方法（见代码清单 5-15）实现。

2) 如果请求消息的接收者的地址与当前 NettyRpcEnv 的地址不同<sup>⊖</sup>，那么将 message

---

<sup>⊖</sup> 说明处理请求的 RpcEndpoint 位于本地的 NettyRpcEnv 中。

<sup>⊖</sup> 说明处理请求的 RpcEndpoint 位于其他节点的 NettyRpcEnv 中。

序列化，并与 onFailure、onSuccess 方法一道封装为 RpcOutboxMessage 类型的消息。最后调用 postToOutbox 方法将消息投递出去。

NettyRpcEnv 的 ask 和 send 方法都调用了私有方法 postToOutbox，postToOutbox 用于向远端节点上的 RpcEndpoint 发送消息，其实现如代码清单 5-38 所示。

代码清单5-38 postToOutbox的实现

---

```

private def postToOutbox(receiver: NettyRpcEndpointRef, message: OutboxMessage):
    Unit = {
      if (receiver.client != null) {
        message.sendWith(receiver.client)
      } else {
        require(receiver.address != null,
               "Cannot send message to client endpoint with no listen address.")
        val targetOutbox = {
          val outbox = outboxes.get(receiver.address)
          if (outbox == null) {
            val newOutbox = new Outbox(this, receiver.address)
            val oldOutbox = outboxes.putIfAbsent(receiver.address, newOutbox)
            if (oldOutbox == null) {
              newOutbox
            } else {
              oldOutbox
            }
          } else {
            outbox
          }
        }
        if (stopped.get) {
          // It's possible that we put 'targetOutbox' after stopping. So we need to clean it.
          outboxes.remove(receiver.address)
          targetOutbox.stop()
        } else {
          targetOutbox.send(message)
        }
      }
    }
}

```

---

根据代码清单 5-38，postToOutbox 方法的执行步骤如下。

- 1) 如果 NettyRpcEndpointRef 中的 TransportClient 不为空，则直接调用 Outbox-Message 的 sendWith 方法，否则进入第 2) 步。
- 2) 获取 NettyRpcEndpointRef 的远端 RpcEndpoint 地址所对应的 Outbox。首先从 outboxes 缓存中获取 Outbox。如果 outboxes 中没有相应的 Outbox，则需要新建 Outbox 并放入 outboxes 缓存中。
- 3) 如果当前 NettyRpcEnv 已经处于停止状态，则将第 2) 步得到的 Outbox 从 outboxes 中移除，并且调用 Outbox 的 stop 方法停止 Outbox。如果当前 NettyRpcEnv 还未停止，则调用第 2) 步得到的 Outbox 的 send 方法发送消息。

Outbox 的 stop 方法用于停止 Outbox，其实现如代码清单 5-39 所示。

代码清单5-39 Outbox的stop方法

```

def stop(): Unit = {
    synchronized {
        if (stopped) {
            return
        }
        stopped = true
        if (connectFuture != null) {
            connectFuture.cancel(true)
        }
        closeClient()
    }

    var message = messages.poll()
    while (message != null) {
        message.onFailure(new SparkException("Message is dropped because Outbox is
                                              stopped"))
        message = messages.poll()
    }
}

```

根据代码清单 5-39，可以看到停止 Outbox 的效果包括如下几项。

- 将 Outbox 的停止状态 stopped 置为 true。
- 关闭 Outbox 中的 TransportClient。
- 清空 messages 列表中的消息。

根据本节对客户端发送请求的分析，现在我们可以将此流程用图 5-10 表示。

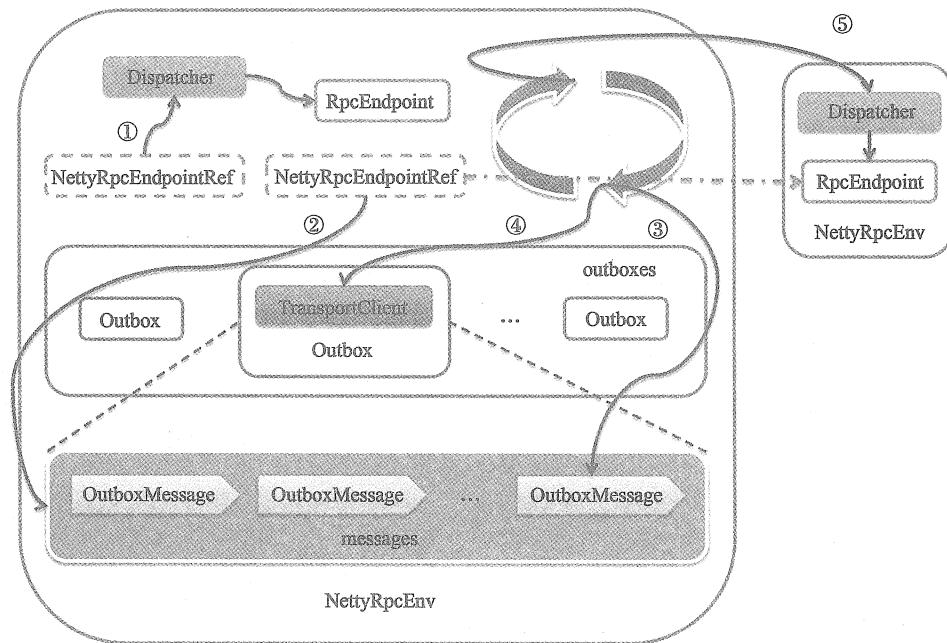


图 5-10 RPC 客户端发送请求示意图

图 5-10 展示了两个不同节点上的 NettyRpcEnv。右边的 NettyRpcEnv 采用简略的表示方法，只展示了其内部的 Dispatcher 组件，实际上右边的 NettyRpcEnv 与左边的 NettyRpcEnv 在结构和组成上是一样的。除了 Dispatcher 组件，还展示了 NettyRpcEnv 内部的 outboxes 列表、outboxes 列表内的 Outbox 及 Outbox 内部用于缓存 OutboxMessage 的 messages 列表。左边 NettyRpcEnv 中的 NettyRpcEndpointRef 和右边一个提供服务的 RpcEndpoint 之间有箭头的虚线表示 NettyRpcEndpointRef 引用 RpcEndpoint，也就是说 NettyRpcEndpointRef 知道 RpcEndpoint 的地址信息。有了这些简短的陈述，我们可以对图 5-10 中各个序号的含义进行简单的说明。

序号①：表示通过调用 NettyRpcEndpointRef 的 send 和 ask 方法向本地节点的 RpcEndpoint 发送消息。由于是在同一节点，所以直接调用 Dispatcher 的 postLocalMessage 或 postOneWayMessage 方法将消息放入 EndpointData 内部 Inbox 的 messages 列表中。MessageLoop 线程最后处理消息，并将消息发给对应的 RpcEndpoint 处理。

序号②：表示通过调用 NettyRpcEndpointRef 的 send 和 ask 方法向远端节点的 RpcEndpoint 发送消息。这种情况下，消息将首先被封装为 OutboxMessage，然后放入到远端 RpcEndpoint 的地址所对应的 Outbox 的 messages 列表中。

序号③：表示每个 Outbox 的 drainOutbox 方法通过循环，不断从 messages 列表中取得 OutboxMessage。

序号④：表示每个 Outbox 的 drainOutbox 方法使用 Outbox 内部的 TransportClient 向远端的 NettyRpcEnv 发送序号③中取得的 OutboxMessage。

序号⑤：表示序号④发出的请求在与远端 NettyRpcEnv 的 TransportServer 建立了连接后，请求消息首先经过 Netty 管道的处理，然后经由 NettyRpcHandler 处理，最后 NettyRpcHandler 的 receive 方法会调用 Dispatcher 的 postRemoteMessage 或 postOneWayMessage 方法，将消息放入 EndpointData 内部 Inbox 的 messages 列表中。MessageLoop 线程最后处理消息，并将消息发给对应的 RpcEndpoint 处理。

### 5.3.9 NettyRpcEnv 中的常用方法

在 NettyRpcEnv 中有一些方法经常要用到，所以对它们做一些简单介绍。

1) 获取 RpcEndpoint 的引用对象 RpcEndpointRef。NettyRpcEnv 重写了抽象类 RpcEnv 的 endpointRef 方法，其实现如代码清单 5-40 所示。

代码清单5-40 获取RpcEndpoint的引用

---

```
override def endpointRef(endpoint: RpcEndpoint): RpcEndpointRef = {
    dispatcher.getRpcEndpointRef(endpoint)
}
```

---

可以看到，endpointRef 方法的实质是调用了 Dispatcher 的 getRpcEndpointRef 方法。getRpcEndpointRef 实际是从 endpointRefs 缓存中获取的 RpcEndpoint 的引用对象 RpcEndpointRef，如代码清单 5-41 所示。

**代码清单5-41 Dispatcher的getRpcEndpointRef方法**


---

```
def getRpcEndpointRef(endpoint: RpcEndpoint): RpcEndpointRef = endpointRefs.get(endpoint)
```

---

2) 使用 RpcAddress 和 RpcEndpoint 的名称, 得到对应的 RpcEndpointRef。setupEndpointRef 方法实际是 NettyRpcEnv 继承自父类 RpcEnv 的方法, 其实现如代码清单 5-42 所示。

**代码清单5-42 setupEndpointRef的实现**


---

```
def setupEndpointRef(address: RpcAddress, endpointName: String): RpcEndpointRef = {
    setupEndpointRefByURI(RpcEndpointAddress(address, endpointName).toString)
}
```

---

可以看到 setupEndpointRef 方法内部调用了 setupEndpointRefByURI, 其实现如代码清单 5-43 所示。

**代码清单5-43 setupEndpointRefByURI的实现**


---

```
def setupEndpointRefByURI(uri: String): RpcEndpointRef = {
    defaultLookupTimeout.awaitResult(asyncSetupEndpointRefByURI(uri))
}
```

---

这里的 defaultLookupTimeout 是通过调用 RpcUtils 的 lookupRpcTimeout 方法 (详见附录 H) 得到的查找超时长。asyncSetupEndpointRefByURI 的功能是向远端 NettyRpcEnv 询问指定名称的 RpcEndpoint 的 NettyRpcEndpointRef。可以看到这里调用 NettyRpcEndpointRef 的 ask 方法 (5.3.8 节对此有详细介绍) 向远端 NettyRpcEnv 中的 RpcEndpointVerifier 发送 RpcEndpointVerifier.CheckExistence 类型的消息。RpcEndpointVerifier 的实现已在代码清单 5-27 中详细介绍过。asyncSetupEndpointRefByURI 的实现如代码清单 5-44 所示。

**代码清单5-44 异步获取远端NettyRpcEndpointRef**


---

```
def asyncSetupEndpointRefByURI(uri: String): Future[RpcEndpointRef] = {
    val addr = RpcEndpointAddress(uri)
    val endpointRef = new NettyRpcEndpointRef(conf, addr, this)
    val verifier = new NettyRpcEndpointRef(
        conf, RpcEndpointAddress(addr.rpcAddress, RpcEndpointVerifier.NAME), this)
    verifier.ask[Boolean](RpcEndpointVerifier.CheckExistence(endpointRef.name)).
        flatMap { find =>
            if (find) {
                Future.successful(endpointRef)
            } else {
                Future.failed(new RpcEndpointNotFoundException(uri))
            }
        }(ThreadUtils.sameThread)
}
```

---

□ 停止 RpcEndpointRef 所引用的 RpcEndpoint。NettyRpcEnv 重写了抽象类 RpcEnv 的 stop 方法, 其实现如代码清单 5-45 所示。

**代码清单5-45 停止RpcEndpoint**


---

```
override def stop(endpointRef: RpcEndpointRef): Unit = {
    require(endpointRef.isInstanceOf[NettyRpcEndpointRef])
    dispatcher.stop(endpointRef)
}
```

---

可以看到，stop方法的实质是调用了Dispatcher的stop方法（见代码清单5-10）。Dispatcher的stop方法对RpcEndpoint去注册，此RpcEndpoint相关联的数据（即EndpointData关联的RpcEndpoint、RpcEndpointRef和Inbox）都将被移除，RpcEndpoint也会停止。

4) 设置Endpoint。其实现如代码清单5-46所示。

**代码清单5-46 设置Endpoint**


---

```
override def setupEndpoint(name: String, endpoint: RpcEndpoint): RpcEndpointRef = {
    dispatcher.registerRpcEndpoint(name, endpoint)
}
```

---

从代码清单5-46可以看到，实际是调用了代码清单5-9中Dispatcher的registerRpcEndpoint方法注册Endpoint。

## 5.4 序列化管理器 SerializerManager

Spark中很多对象在通过网络传输或者写入存储体系时，都需要序列化。SparkEnv中有两个序列化的组件，分别是SerializerManager和closureSerializer，SparkEnv中创建它们的代码如下。

```
val serializer = instantiateClassFromConf[Serializer](
    "spark.serializer", "org.apache.spark.serializer.JavaSerializer")
logDebug(s"Using serializer: ${serializer.getClass}")

val serializerManager = new SerializerManager(serializer, conf,
    ioEncryptionKey)

val closureSerializer = new JavaSerializer(conf)
```

可以看到这里创建的serializer默认为org.apache.spark.serializer.JavaSerializer，用户可以通过spark.serializer属性配置其他的序列化实现，如org.apache.spark.serializer.KryoSerializer。closureSerializer的实际类型固定为org.apache.spark.serializer.JavaSerializer，用户不能够自己指定。JavaSerializer采用Java语言自带的序列化API实现，由于JavaSerializer的实现非常简单，这里不多介绍，感兴趣的读者可自行阅读其实现。

### 1. 序列化管理器 SerializerManager的属性

SerializerManager给各种Spark组件提供序列化、压缩及加密的服务。这里主要对SerializerManager中的各个成员属性进行介绍。

- defaultSerializer：默认的序列化器。此 defaultSerializer 即为上面代码中实例化的 serializer，也就是说 defaultSerializer 的类型是 JavaSerializer。
- conf：即 SparkConf。
- encryptionKey：加密使用的密钥。
- kryoSerializer：Spark 提供的另一种序列化器。kryoSerializer 的实际类型是 Kryo Serializer，其采用 Google 提供的 Kryo 序列化库实现。
- stringClassTag：字符串类型标记，即 ClassTag[String]。
- primitiveAndPrimitiveArrayClassTags：原生类型及原生类型数组的类型标记的集合，包括：Boolean、Array[boolean]、Int、Array[int]、Long、Array[long]、Byte、Array[byte]、Null、Array[scala.runtime.Null\$]、Char、Array[char]、Double、Array [double]、Float、Array[float]、Short、Array[short] 等。
- compressBroadcast：是否对广播对象进行压缩，可以通过 spark.broadcast.compress 属性配置，默认为 true。
- compressShuffle：是否对 Shuffle 输出数据压缩，可以通过 spark.shuffle.compress 属性配置，默认为 true。
- compressRdds：是否对 RDD 压缩，可以通过 spark.rdd.compress 属性配置，默认为 false。
- compressShuffleSpill：是否对溢出到磁盘的 Shuffle 数据压缩，可以通过 spark.shuffle.spill.compress 属性配置，默认为 true。
- compressionCodec：SerializerManager 使用的压缩编解码器。compressionCodec 的类型是 CompressionCodec。在 Spark 1.x.x 版本中，compressionCodec 是 BlockManager 的成员之一，现在把 compressionCodec 和序列化、加密等功能都集中到 SerializerManager 中，也许是因为实现此功能的工程师觉得加密、压缩都是属于序列化的一部分吧。

## 2. 创建 CompressionCodec

为了节省磁盘存储空间，有些情况下需要对数据进行压缩。在 SerializerManager 中创建 compressionCodec 的代码如下。

```
private lazy val compressionCodec: CompressionCodec = CompressionCodec.createCodec(conf)
```

可以看到 compressionCodec 通过 lazy 关键字修饰为延迟初始化，即等到真正使用时才对其进行初始化。CompressionCodec 的 createCodec 方法用于创建 CompressionCodec，其实现如代码清单 5-47 所示。

代码清单5-47 CompressionCodec的createCodec方法

---

```
private val shortCompressionCodecNames = Map(
```

```

"lz4" -> classOf[LZ4CompressionCodec].getName,
"lzf" -> classOf[LZFCompressionCodec].getName,
"snappy" -> classOf[SnappyCompressionCodec].getName)

def getCodecName(conf: SparkConf): String = {
  conf.get(configKey, DEFAULT_COMPRESSION_CODEC)
}

def createCodec(conf: SparkConf): CompressionCodec = {
  createCodec(conf, getCodecName(conf))
}

def createCodec(conf: SparkConf, codecName: String): CompressionCodec = {
  val codecClass = shortCompressionCodecNames.getOrElse(codecName.toLowerCase,
    codecName)
  val codec = try {
    val ctor = Utils.classForName(codecClass).getConstructor(classOf[SparkConf])
    Some(ctor.newInstance(conf).asInstanceOf[CompressionCodec])
  } catch {
    case e: ClassNotFoundException => None
    case e: IllegalArgumentException => None
  }
  codec.getOrElse(throw new IllegalArgumentException(s"Codec [$codecName] is not
    available. " +
    s"Consider setting $configKey=$FALLBACK_COMPRESSION_CODEC"))
}

```

根据代码清单 5-47，我们看到 `createCodec` 方法首先调用了 `getCodecName` 方法获取编解码器的名称。其中的变量 `configKey` 为 `spark.io.compression.codec`，即可以使用此属性配置压缩需要的编解码器名称。如果没有指定 `spark.io.compression.codec`，那么编解码器的默认名称为 `lz4`（即常量 `DEFAULT_COMPRESSION_CODEC` 的值）。然后会调用重载的 `createCodec` 方法，其执行步骤如下。

- 1) 从 `shortCompressionCodecNames` 缓存中获取编解码器名称对应的编解码器的类名。
- 2) 通过 Java 反射生成编解码器的实例。

---

 **注意** 在 Spark 1.x.x 版本中，默认的压缩算法为 `snappy`，`snappy` 在牺牲少量压缩比例的条件下，极大地提高了压缩速度。

---

### 3. 序列化管理器 `SerializerManager` 的方法

在 `SerializerManager` 中提供了很多用于序列化、反序列化、压缩、加密的方法，这里对这些功能进行介绍。

- `encryptionEnabled`：当前 `SerializerManager` 是否支持加密。`SerializerManager` 要想支持加密，必须在构造 `SerializerManager` 的时候就传入 `encryptionKey`。可以通过 `spark.io.encryption.enabled`（允许加密）、`spark.io.encryption.keySizeBits`（密钥长度，有 128、192、256 三种长度）、`spark.io.encryption.keygen.algorithm`（加密算法，默认为 `HmacSHA1`）等属性进行具体的配置。

- `canUseKryo(ct: ClassTag[])`：对于指定的类型标记 ct，是否能使用 `kryoSerializer` 进行序列化。当类型标记 ct 属于 `primitiveAndPrimitiveArrayClassTags` 或者 `stringClassTag` 时，`canUseKryo` 方法才返回真。
- `getSerializer(ct: ClassTag[], autoPick: Boolean)`：获取序列化器。如果 `autoPick` 为 true（即 `BlockId` 不为 `StreamBlockId` 时）并且调用 `canUseKryo` 的结果为 true 时选择 `kryoSerializer`，否则选择 `defaultSerializer`。
- `getSerializer(keyClassTag: ClassTag[], valueClassTag: ClassTag[])`：获取序列化器。如果对于 `keyClassTag` 和 `valueClassTag`，调用 `canUseKryo` 的结果都为 true 时选择 `kryoSerializer`，否则选择 `defaultSerializer`。
- `wrapStream(blockId: BlockId, s: InputStream)`：对 Block 的输入流进行压缩与加密。
- `wrapStream(blockId: BlockId, s: OutputStream)`：对 Block 的输出流进行压缩与加密。
- `wrapForEncryption(s: InputStream)`：对输入流进行加密。
- `wrapForEncryption(s: OutputStream)`：对输出流进行加密。
- `dataSerializeStream[T: ClassTag](blockId: BlockId, outputStream: OutputStream, values: Iterator[T])`：对 Block 的输出流序列化。
- `dataSerialize[T: ClassTag](blockId: BlockId, values: Iterator[T], allowEncryption: Boolean = true)`：序列化成分块字节缓冲区（`ChunkedByteBuffer`）。
- `dataSerializeWithExplicitClassTag(blockId: BlockId, values: Iterator[], classTag: ClassTag[], allowEncryption: Boolean = true)`：使用明确的类型标记，序列化成分块字节缓冲区（`ChunkedByteBuffer`）。
- `dataDeserializeStream[T](blockId: BlockId, inputStream: InputStream, maybeEncrypted: Boolean = true) (classTag: ClassTag[T])`：将输入流反序列化为值的迭代器 `Iterator[T]`。

## 5.5 广播管理器 BroadcastManager

`BroadcastManager` 用于将配置信息和序列化后的 RDD、Job 及 `ShuffleDependency` 等信息在本地存储。如果为了容灾，也会复制到其他节点上。创建 `BroadcastManager` 的代码实现如下。

```
val broadcastManager = new BroadcastManager(isDriver, conf, securityManager)
```

`BroadcastManager` 除了构造器定义的三个成员属性外，`BroadcastManager` 内部还有三个成员，分别如下。

- `initialized`：表示 `BroadcastManager` 是否初始化完成的状态。
- `broadcastFactory`：广播工厂实例。
- `nextBroadcastId`：下一个广播对象的广播 ID，类型为 `AtomicLong`。

BroadcastManager 在其初始化的过程中就会调用自身的 initialize 方法，当 initialize 执行完毕，BroadcastManager 就正式生效。BroadcastManager 的 initialize 方法的实现如代码清单 5-48 所示。

代码清单5-48 BroadcastManager的初始化

---

```
private def initialize() {
    synchronized {
        if (!initialized) {
            broadcastFactory = new TorrentBroadcastFactory
            broadcastFactory.initialize(isDriver, conf, securityManager)
            initialized = true
        }
    }
}
```

---

根据代码清单 5-48，initialize 方法首先判断 BroadcastManager 是否已经初始化，以保证 BroadcastManager 只被初始化一次。新建 TorrentBroadcastFactory 作为 BroadcastManager 的广播工厂实例。之后调用 TorrentBroadcastFactory 的 initialize 方法对 TorrentBroadcastFactory 进行初始化<sup>⊖</sup>。最后将 BroadcastManager 自身标记为初始化完成状态。

 **注意** TorrentBroadcastFactory 实现了 BroadcastFactory 特质。在 Spark 1.x.x 版本中，BroadcastManager 的 initialize 方法是使用 Java 反射生成广播工厂实例 broadcastFactory 的，还可以通过配置属性 spark.broadcast.factory 指定 BroadcastFactory 特质的实现类，默认为 org.apache.spark.broadcast.TorrentBroadcastFactory。从 Spark 2.0.0 版本开始，不再提供此 Spark 属性，属性成员 broadcastFactory 也固定为 TorrentBroadcastFactory。

---

BroadcastManager 中提供了三个方法，如代码清单 5-49 所示。

代码清单5-49 BroadcastManager中的三个方法

---

```
def stop() {
    broadcastFactory.stop()
}
private val nextBroadcastId = new AtomicLong(0)
def newBroadcast[T: ClassTag](value_ : T, isLocal: Boolean): Broadcast[T] = {
    broadcastFactory.newBroadcast[T](value_, isLocal, nextBroadcastId.getAndIncrement())
}
def unbroadcast(id: Long, removeFromDriver: Boolean, blocking: Boolean) {
    broadcastFactory.unbroadcast(id, removeFromDriver, blocking)
}
```

---

从代码清单 5-49 可以看到，BroadcastManager 的三个方法都分别代理了 TorrentBroadcast-

---

<sup>⊖</sup> 笔者查阅 TorrentBroadcastFactory 的源码后，发现 TorrentBroadcastFactory 的 Initialize 方法实际是一个空实现，所以这里不再介绍。

Factory 的对应方法，TorrentBroadcastFactory 中提供的三个方法的实现如代码清单 5-50 所示。

代码清单5-50 TorrentBroadcastFactory提供的方法

---

```
override def newBroadcast[T: ClassTag](value_ : T, isLocal: Boolean, id: Long): Broadcast[T] = {
    new TorrentBroadcast[T](value_, id)
}
override def stop() { }
override def unbroadcast(id: Long, removeFromDriver: Boolean, blocking: Boolean) {
    TorrentBroadcast.unpersist(id, removeFromDriver, blocking)
}
```

---

代码清单 5-50 中 TorrentBroadcastFactory 提供的三个方法，由于 stop 是空实现，所以我们只关注 newBroadcast 和 unbroadcast 两个方法。

根据代码清单 5-50，我们知道 TorrentBroadcastFactory 的 newBroadcast 方法用于生成 TorrentBroadcast 实例，其作用为广播 TorrentBroadcast 中的 value。表面看只是利用构造器生成了 TorrentBroadcast 实例，但是其效果远不止此。TorrentBroadcast 对象包括以下属性。

- compressionCodec：用于广播对象的压缩编解码器。可以设置 spark.broadcast.compress 属性为 true 启用，默认是启用的。compressionCodec 的类型为 5.4 节详细介绍过的 CompressionCodec，而且最终采用的压缩算法与 SerializerManager 中的 CompressionCodec 是一致的。
- blockSize：每个块的大小。它是个只读属性，可以使用 spark.broadcast.blockSize 属性进行配置，默认为 4MB。
- broadcastId：广播 Id。broadcastId 实际是样例类 BroadcastBlockId，BroadcastBlockId 类的实现如下。

```
case class BroadcastBlockId(broadcastId: Long, field: String = "") extends BlockId {
    override def name: String = "broadcast_" + broadcastId + (if (field == "") ""
        else "_" + field)
}
```

其中，broadcastId 是由 BroadcastManager 的原子变量 nextBroadcastId 自增产生的。

- checksumEnabled：是否给广播块生成校验和。可以通过 spark.broadcast.checksum 属性进行配置，默认为 true。
- checksums：用于存储每个广播块的校验和的数组。
- numBlocks：广播变量包含的块的数量。numBlocks 通过调用 writeBlocks 方法获得。由于 numBlocks 是个 val 修饰的不可变属性，因此在构造 TorrentBroadcast 实例的时候就会调用 writeBlocks 方法将广播对象写入存储体系。
- \_value：从 Executor 或者 Driver 上读取的广播块的值。\_value 是通过调用 readBroadcastBlock 方法获得的广播对象。由于 \_value 是个 lazy 及 val 修饰的属性，因

此在构造 TorrentBroadcast 实例的时候不会调用 readBroadcastBlock 方法，而是等到明确需要使用 \_value 的值时。

### 1. 广播对象的写操作

刚才说到在构造 TorrentBroadcast 实例的时候就会调用 writeBlocks 方法，其实现如代码清单 5-51 所示。

代码清单5-51 writeBlocks的实现

---

```

private def writeBlocks(value: T): Int = {
    import StorageLevel._
    val blockManager = SparkEnv.get.blockManager
    if (!blockManager.putSingle(broadcastId, value, MEMORY_AND_DISK, tellMaster =
        false)) {
        throw new SparkException(s"Failed to store $broadcastId in BlockManager")
    }
    val blocks =
        TorrentBroadcast.blockifyObject(value, blockSize, SparkEnv.get.serializer,
            compressionCodec)
    if (checksumEnabled) {
        checksums = new Array[Int](blocks.length)
    }
    blocks.zipWithIndex.foreach { case (block, i) =>
        if (checksumEnabled) {
            checksums(i) = calcChecksum(block)
        }
        val pieceId = BroadcastBlockId(id, "piece" + i)
        val bytes = new ChunkedByteBuffer(block.duplicate())
        if (!blockManager.putBytes(pieceId, bytes, MEMORY_AND_DISK_SER, tellMaster =
            true)) {
            throw new SparkException(s"Failed to store $pieceId of $broadcastId in
                local BlockManager")
        }
    }
    blocks.length
}

```

---

根据代码清单 5-51，writeBlocks 的执行步骤如下。

- 1 ) 获取当前 SparkEnv 的 BlockManager 组件。
- 2 ) 调用 BlockManager 的 putSingle 方法将广播对象写入本地的存储体系。当 Spark 以 local 模式运行时，则会将广播对象写入 Driver 本地的存储体系，以便于任务也可以在 Driver 上执行。由于 MEMORY\_AND\_DISK 对应的 StorageLevel 的 \_replication 属性固定为 1，因此此处只会将广播对象写入 Driver 或 Executor 本地的存储体系，有关 BlockManager 的 putSingle 方法及 StorageLevel 的内容将在第 6 章详细介绍。
- 3 ) 调用 TorrentBroadcast 的 blockifyObject 方法（实现很简单，感兴趣的读者可以自行查阅），将对象转换成一系列的块。每个块的大小由 blockSize 决定，使用当前 SparkEnv 中的 JavaSerializer 组件进行序列化，使用 TorrentBroadcast 自身的 compressionCodec 进行压缩。

4) 如果需要给分片广播块生成校验和，则创建和第 3) 步转换的块的数量一致的 checksums 数组。

5) 对每个块进行如下处理：如果需要给分片广播块生成校验和，则给分片广播块生成校验和。calcChecksum 方法的实现很简单，感兴趣的读者可以自行查阅。给当前分片广播块生成分片的 BroadcastBlockId，分片通过 BroadcastBlockId 的 field 属性区别，例如，piece0、piece1……调用 BlockManager 的 putBytes 方法将分片广播块以序列化方式写入 Driver 本地的存储体系。由于 MEMORY\_AND\_DISK\_SER 对应的 StorageLevel 的 replication 属性也固定为 1，因此此处只会将分片广播块写入 Driver 或 Executor 本地的存储体系，有关 BlockManager 的 putBytes 方法及 StorageLevel 的内容将在第 6 章详细介绍。

6) 返回块的数量。

经过以上分析，最后用图 5-11 来更直观地表示广播对象的写入过程。

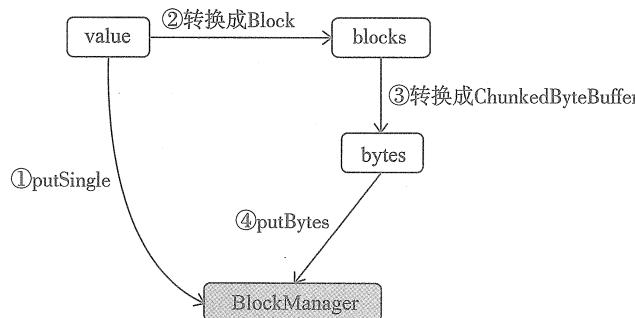


图 5-11 广播对象的写入

## 2. 广播对象的读操作

前文提到，只有当 TorrentBroadcast 实例的 \_value 属性值在需要的时候，才会调用 readBroadcastBlock 方法获取值，readBroadcastBlock 的实现如代码清单 5-52 所示。

代码清单 5-52 readBroadcastBlock 的实现

---

```

private def readBroadcastBlock(): T = Utils.tryOrIOException {
  TorrentBroadcast.synchronized {
    setConf(SparkEnv.get.conf)
    val blockManager = SparkEnv.get.blockManager
    blockManager.getLocalValues(broadcastId) match {
      case Some(blockResult) =>
        if (blockResult.data.hasNext) {
          val x = blockResult.data.next().asInstanceOf[T]
          releaseLock(broadcastId)
          x
        } else {
          throw new SparkException(s"Failed to get locally stored broadcast data:
            $broadcastId")
        }
      case None =>
    }
  }
}

```

```

    logInfo("Started reading broadcast variable " + id)
    val startTimeMs = System.currentTimeMillis()
    val blocks = readBlocks().flatMap(_.getChunks())
    logInfo("Reading broadcast variable " + id + " took" + Utils.
        getUsedTimeMs(startTimeMs))

    val obj = TorrentBroadcast.unBlockifyObject[T] (
        blocks, SparkEnv.get.serializer, compressionCodec)
    val storageLevel = StorageLevel.MEMORY_AND_DISK
    if (!blockManager.putSingle(broadcastId, obj, storageLevel, tellMaster =
        false)) {
        throw new SparkException(s"Failed to store $broadcastId in Block-
            Manager")
    }
    obj
}
}
}

```

---

根据代码清单 5-52，readBroadcastBlock 的执行步骤如下。

- 1 ) 获取当前 SparkEnv 的 BlockManager 组件。
- 2 ) 调用 BlockManager 的 getLocalValues 方法从本地的存储系统中获取广播对象，即通过 BlockManager 的 putSingle 方法写入存储体系的广播对象，BlockManager 的 getLocalValues 方法将在 6.7.2 节详细介绍。
- 3 ) 如果从本地的存储体系中可以获取广播对象，则调用 releaseLock 方法（这个锁保证当块被一个运行中的任务使用时，不能被其他任务再次使用，但是当任务运行完成时，则应该释放这个锁），释放当前块的锁并返回此广播对象。
- 4 ) 如果从本地的存储体系中没有获取到广播对象，那么说明数据是通过 BlockManager 的 putBytes 方法以序列化方式写入存储体系的。此时首先调用 readBlocks 方法从 Driver 或 Executor 的存储体系中获取广播块，然后调用 TorrentBroadcast 的 unBlockifyObject 方法（实现很简单，感兴趣的读者可以自行查阅），将一系列的分片广播块转换回原来的广播对象，最后再次调用 BlockManager 的 putSingle 方法将广播对象写入本地的存储体系，以便于当前 Executor 的其他任务不用再次获取广播对象。

上文谈到调用 readBlocks 方法可以从 Driver、Executor 的存储体系中获取块，其实现如代码清单 5-53 所示。

代码清单5-53 readBlocks的实现

---

```

private def readBlocks(): Array[ChunkedByteBuffer] = {
    val blocks = new Array[ChunkedByteBuffer](numBlocks)
    val bm = SparkEnv.get.blockManager
    for (pid <- Random.shuffle(Seq.range(0, numBlocks))) {
        val pieceId = BroadcastBlockId(id, "piece" + pid)
        logDebug(s"Reading piece $pieceId of $broadcastId")
        bm.getLocalBytes(pieceId) match {
            case Some(block) =>

```

```

        blocks(pid) = block
        releaseLock(pieceId)
    case None =>
        bm.getRemoteBytes(pieceId) match {
            case Some(b) =>
                if (checksumEnabled) {
                    val sum = calcChecksum(b.chunks(0))
                    if (sum != checksums(pid)) {
                        throw new SparkException(s"corrupt remote block $pieceId of
                            $broadcastId: " +
                            s" $sum != ${checksums(pid)}")
                    }
                }
                // We found the block from remote executors/driver's BlockManager, so
                // put the block
                // in this executor's BlockManager.
                if (!bm.putBytes(pieceId, b, StorageLevel.MEMORY_AND_DISK_SER, tellMaster
                    = true)) {
                    throw new SparkException(
                        s"Failed to store $pieceId of $broadcastId in local Block-
                            Manager")
                }
                blocks(pid) = b
            case None =>
                throw new SparkException(s"Failed to get $pieceId of $broadcastId")
        }
    }
}
blocks
}

```

根据代码清单 5-53，readBlocks 方法的执行步骤如下。

- 1) 新建用于存储每个分片广播块的数组 blocks，并获取当前 SparkEnv 的 BlockManager 组件。
- 2) 对各个广播分片进行随机洗牌，避免对广播块的获取出现“热点”，提升性能。对洗牌后的各个广播分片依次执行 3) 至 5) 步的操作。
- 3) 调用 BlockManager 的 getLocalBytes 方法从本地的存储体系中获取序列化的分片广播块，如果本地可以获取到，则将分片广播块放入 blocks，并且调用 releaseLock 方法释放此分片广播块的锁。
- 4) 如果本地没有，则调用 BlockManager 的 getRemoteBytes 方法从远端的存储体系中获取分片广播块。对于获取的分片广播块再次调用 calcChecksum 方法计算校验和，并将此校验和与调用 writeBlocks 方法时存入 checksums 数组的校验和进行比较。如果校验和不相同，说明块的数据有损坏，此时抛出异常。
- 5) 如果校验和相同，则调用 BlockManager 的 putBytes 方法将分片广播块写入本地存储体系，以便于当前 Executor 的其他任务不用再次获取分片广播块。最后将分片广播块放入 blocks。

6) 返回 blocks 中的所有分片广播块。

经过以上的分析，最后用图 5-12 来更直观地表示广播对象的读取过程。

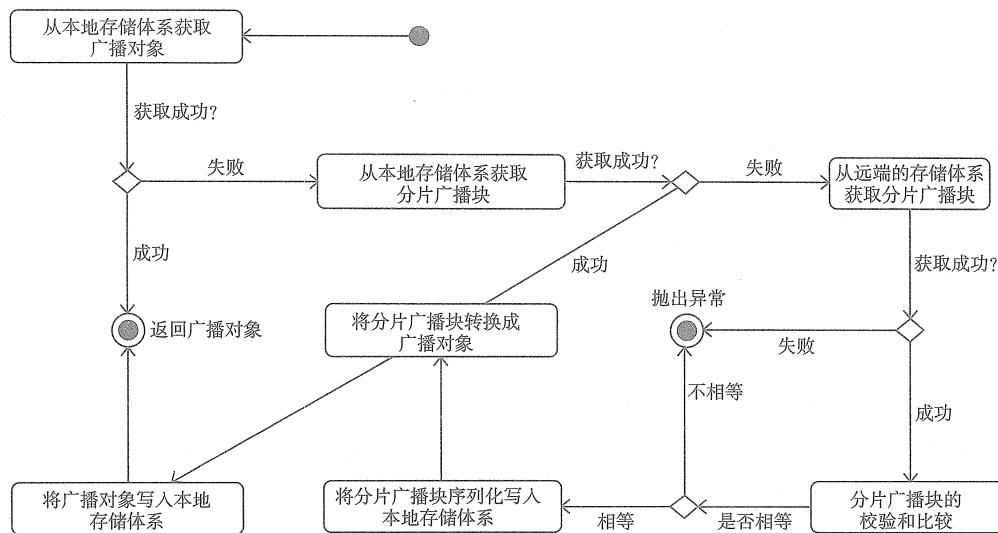


图 5-12 广播对象的读取

### 3. 广播对象的去持久化

根据代码清单 5-50，我们知道 TorrentBroadcastFactory 的 unbroadcast 方法实际调用了 TorrentBroadcast 的 unpersist 方法对由 id 标记的广播对象去持久化。TorrentBroadcast 的 unpersist 方法的实现如代码清单 5-54 所示。

代码清单 5-54 对广播对象去持久化

---

```

def unpersist(id: Long, removeFromDriver: Boolean, blocking: Boolean): Unit = {
  logDebug(s"Unpersisting TorrentBroadcast $id")
  SparkEnv.get.blockManager.master.removeBroadcast(id, removeFromDriver, blocking)
}
  
```

---

根据代码清单 5-54，可以看到 TorrentBroadcast 的 unpersist 方法实际调用了 BlockManager 的子组件 BlockManagerMaster 的 removeBroadcast 方法来实现对广播对象去持久化，有关 BlockManagerMaster 的具体介绍请参阅 6.8 节。

## 5.6 map 任务输出跟踪器

mapOutputTracker 用于跟踪 map 任务的输出状态，此状态便于 reduce 任务定位 map 输出结果所在的节点地址，进而获取中间输出结果。每个 map 任务或者 reduce 任务都会有其唯一标识，分别为 mapId 和 reduceId。每个 reduce 任务的输入可能是多个 map 任务

的输出，reduce 会到各个 map 任务所在的节点上拉取 Block，这一过程叫做 Shuffle。每次 Shuffle 都有唯一的标识 shuffleId。

在开始介绍 mapOutputTracker 之前，我们先来介绍 SparkEnv 中用于注册 RpcEndpoint 或者查找 RpcEndpoint 的方法 registerOrLookupEndpoint，如代码清单 5-55 所示。

代码清单 5-55 注册或查找EndPoint

---

```
def registerOrLookupEndpoint(
    name: String, endpointCreator: => RpcEndpoint):
    RpcEndpointRef = {
        if (isDriver) {
            logInfo("Registering " + name)
            rpcEnv.setupEndpoint(name, endpointCreator)
        } else {
            RpcUtils.makeDriverRef(name, conf, rpcEnv)
        }
    }
```

---

根据代码清单 5-55，如果当前实例是 Driver，则调用代码清单 5-46 中介绍的 setupEndpoint 方法向 Dispatcher 注册 Endpoint。如果是 Executor，则调用工具类 RpcUtils 的 makeDriverRef 方法向远端的 NettyRpcEnv 询问获取相关 RpcEndpoint 的 RpcEndpointRef。

SparkEnv 中创建 mapOutputTracker 的代码如下。

```
val mapOutputTracker = if (isDriver) {
    new MapOutputTrackerMaster(conf, broadcastManager, isLocal)
} else {
    new MapOutputTrackerWorker(conf)
}

mapOutputTracker.trackerEndpoint = registerOrLookupEndpoint(MapOutputTracker.
    ENDPOINT_NAME,
    new MapOutputTrackerMasterEndpoint(
        rpcEnv, mapOutputTracker.asInstanceOf[MapOutputTrackerMaster], conf))
```

可以看到针对当前实例是 Driver 还是 Executor，创建 mapOutputTracker 的方式有所不同。

- 如果当前应用程序是 Driver，则创建 MapOutputTrackerMaster，然后创建 MapOutput TrackerMasterEndpoint，并且注册到 Dispatcher 中，注册名为 MapOutputTracker。
- 如果当前应用程序是 Executor，则创建 MapOutputTrackerWorker，并从远端 Driver 实例的 NettyRpcEnv 的 Dispatcher 中查找 MapOutputTrackerMasterEndpoint 的引用。

无论是 Driver 还是 Executor，最后都由 mapOutputTracker 的属性 trackerEndpoint 持有 MapOutputTrackerMasterEndpoint 的引用。

**小贴士：**在 Spark 1.x.x 版本中存在 MapOutputTrackerMasterActor，MapOutputTrackerMasterEndpoint 如今替代了前者。

### 5.6.1 MapOutputTracker 的实现

无论是 MapOutputTrackerMaster 还是 MapOutputTrackerWorker，它们都继承自抽象类 MapOutputTracker。MapOutputTracker 内部定义了任务输出跟踪器的规范，所以我们先来了解它的属性，然后再看看其提供的功能。

MapOutputTracker 中的属性包括以下几项。

- trackerEndpoint：用于持有 Driver 上 MapOutputTrackerMasterEndpoint 的 RpcEndpointRef。
- mapStatuses：用于维护各个 map 任务的输出状态。类型为 Map[Int, Array[MapStatus]]。其中 key 对应 shuffield，Array 存储各个 map 任务对应的状态信息 MapStatus。由于各个 MapOutputTrackerWorker 会向 MapOutputTrackerMaster 不断汇报 map 任务的状态信息，因此 MapOutputTrackerMaster 的 mapStatuses 中维护的信息是最新最全的。MapOutputTrackerWorker 的 mapStatuses 对于本节点 Executor 运行的 map 任务状态是及时更新的，而对于其他节点上的 map 任务状态则更像一个缓存，在 mapStatuses 不能命中时会向 Driver 上的 MapOutputTrackerMaster 获取最新的任务状态信息。
- epoch：用于 Executor 故障转移的同步标记。每个 Executor 在运行的时候会更新 epoch，潜在的附加动作将清空缓存。当 Executor 丢失后增加 epoch。
- epochLock：用于保证 epoch 变量的线程安全性。
- fetching：shuffle 获取集合。数据类型为 HashSet[Int]，用来记录当前 Executor 正在从哪些 map 输出的位置拉取数据。

了解了 MapOutputTracker 中的属性，来看看 MapOutputTracker 所提供的方法。

#### 1. 询问方法 askTracker

askTracker 方法（见代码清单 5-56）用于向 MapOutputTrackerMasterEndpoint 发送消息，并期望在超时时间之内得到回复。

代码清单5-56 askTracker的实现

---

```
protected def askTracker[T: ClassTag](message: Any): T = {
    try {
        trackerEndpoint.askWithRetry[T](message)
    } catch {
        case e: Exception =>
            logError("Error communicating with MapOutputTracker", e)
            throw new SparkException("Error communicating with MapOutputTracker", e)
    }
}
```

---

根据代码清单 5-56，askTracker 通过调用 NettyRpcEndpointRef 的 askWithRetry 方法（在 5.3.2 节已详细介绍）实现。

## 2. 发送方法 sendTracker

`sendTracker` 方法（见代码清单 5-57）用于向 `MapOutputTrackerMasterEndpoint` 发送消息，并期望在超时时间之内获得的返回值为 `true`。

代码清单5-57 `sendTracker`的实现

---

```
protected def sendTracker(message: Any) {
    val response = askTracker[Boolean](message)
    if (response != true) {
        throw new SparkException(
            "Error reply received from MapOutputTracker. Expecting true, got " +
            response.toString)
    }
}
```

---

## 3. getstatuses

`getstatuses` 方法可以根据 `shuffleId` 获取 `MapStatus`（即 map 状态信息）的数组，其实现如代码清单 5-58 所示。

代码清单5-58 `getstatuses`的实现

---

```
private def getstatuses(shuffleId: Int): Array[MapStatus] = {
    val statuses = mapstatuses.get(shuffleId).orNull
    if (statuses == null) {
        logInfo("Don't have map outputs for shuffle " + shuffleId + ", fetching
            them")
        val startTime = System.currentTimeMillis
        var fetchedstatuses: Array[MapStatus] = null
        fetching.synchronized {
            // Someone else is fetching it; wait for them to be done
            while (fetching.contains(shuffleId)) {
                try {
                    fetching.wait()
                } catch {
                    case e: InterruptedException =>
                }
            }
            fetchedstatuses = mapstatuses.get(shuffleId).orNull
            if (fetchedstatuses == null) {
                // We have to do the fetch, get others to wait for us.
                fetching += shuffleId
            }
        }
        if (fetchedstatuses == null) {
            // We won the race to fetch the statuses; do so
            logInfo("Doing the fetch; tracker endpoint = " + trackerEndpoint)
            // This try-finally prevents hangs due to timeouts:
            try {
                val fetchedBytes = askTracker[Array[Byte]](GetMapOutputStatuses(shuffleId))
                fetchedstatuses = MapOutputTracker.deserializeMapStatuses(fetchedBytes)
                logInfo("Got the output locations")
                mapstatuses.put(shuffleId, fetchedstatuses)
            } finally {

```

```

        fetching.synchronized {
            fetching -= shuffleId
            fetching.notifyAll()
        }
    }
}

logDebug(s"Fetching map output statuses for shuffle $shuffleId took " +
    s"${System.currentTimeMillis - startTime} ms")
if (fetchedStatuses != null) {
    return fetchedStatuses
} else {
    logError("Missing all output locations for shuffle " + shuffleId)
    throw new MetadataFetchFailedException(
        shuffleId, -1, "Missing all output locations for shuffle " + shuffleId)
}
} else {
    return statuses
}
}

```

---

根据代码清单 5-58，getstatuses 的执行步骤如下。

- 1) 从当前 MapOutputTracker 的 mapStatuses 缓存中获取 MapStatus 数组，若没有就进入第 2) 步，否则直接返回得到的 MapStatus 数组。
- 2) 如果 shuffle 获取集合（即 fetching）中已经存在要取的 shuffleId（这说明已经有其他线程对此 shuffleId 的数据进行远程拉取了），那么就等待其他线程获取。等待会一直持续，直到 fetching 中不存在要取的 shuffleId（这说明其他线程对此 shuffleId 的数据进行远程拉取的操作已经结束），并再次从 mapStatuses 缓存中获取 MapStatus 数组，此时如果获取到 MapStatus 数组，则继续第 5) 步。
- 3) 如果 fetching 中不存在要取的 shuffleId，那么当前线程需要将 shuffleId 加入 fetching，以表示已经有线程对此 shuffleId 的数据进行远程拉取了。
- 4) 调用 askTracker 方法（见代码清单 5-56）向 MapOutputTrackerMasterEndpoint 发送 GetMapOutputStatuses 消息，以获取 map 任务的状态信息。MapOutputTrackerMasterEndpoint 接收到 GetMapOutputStatuses 消息后，将 GetMapOutputStatuses 消息转换为 GetMapOutputMessage 消息，再放入 mapOutputRequests 队列，如代码清单 5-67、代码清单 5-68 所示。异步线程 MessageLoop 从队列中取出 GetMapOutputMessage，将请求的 map 任务状态信息序列化后返回给请求方，如代码清单 5-64 所示。请求方接收到 map 任务状态信息后，调用 MapOutputTracker 的 deserializeMapStatuses 方法对 map 任务状态进行反序列化操作，然后放入本地的 mapStatuses 缓存中。这次拉取可能成功，也可能失败，所以拉取结束后无论如何需要将 shuffleId 从 fetching 中移除，并唤醒那些在 fetching 的锁上等待的线程，以便这些线程能够获取自己需要的 MapStatus 数组。如果拉取成功则继续第 5) 步。
- 5) 返回得到的 MapStatus 数组。



**注意** 由于多个线程可能会并发访问 fetching，因此使用 fetching 的锁进行同步控制，以确保程序在并发下的安全性。

#### 4. getMapSizesByExecutorId

getMapSizesByExecutorId 方法可通过 shuffleId 和 reduceId 获取存储了 reduce 所需的 map 中间输出结果的 BlockManager 的 BlockManagerId，以及 map 中间输出结果每个 Block 块的 BlockId 与大小，其实现如代码清单 5-59 所示。

代码清单 5-59 getMapSizesByExecutorId 的实现

---

```
def getMapSizesByExecutorId(shuffleId: Int, reduceId: Int)
    : Seq[(BlockManagerId, Seq[(BlockId, Long)])] = {
  getMapSizesByExecutorId(shuffleId, reduceId, reduceId + 1)
}
def getMapSizesByExecutorId(shuffleId: Int, startPartition: Int, endPartition: Int)
    : Seq[(BlockManagerId, Seq[(BlockId, Long)])] = {
  logDebug(s"Fetching outputs for shuffle $shuffleId, partitions $startPartition-$endPartition")
  val statuses = getStatuses(shuffleId)
  statuses.synchronized {
    return MapOutputTracker.convertMapStatuses(shuffleId, startPartition,
                                                endPartition, statuses)
  }
}
```

---

代码清单 5-59 中，MapOutputTracker 的 convertMapStatuses 方法用于将 Array[MapStatus] 转换为 Seq[(BlockManagerId, Seq[(BlockId, Long)])]，其具体实现比较简单，感兴趣的读者可自己查阅。

#### 5. getStatistics

getStatistics 用于获取 shuffle 依赖的各个 map 输出 Block 大小的统计信息，其实现如代码清单 5-60 所示。

代码清单 5-60 获取 MapOutputStatistics

---

```
def getStatistics(dep: ShuffleDependency[_ _, _, _]): MapOutputStatistics = {
  val statuses = getStatuses(dep.shuffleId)
  statuses.synchronized {
    val totalSizes = new Array[Long](dep.partitionner.numPartitions)
    for (s <- statuses) {
      for (i <- 0 until totalSizes.length) {
        totalSizes(i) += s.getSizeForBlock(i)
      }
    }
    new MapOutputStatistics(dep.shuffleId, totalSizes)
  }
}
```

---

## 6. updateEpoch

当 Executor 运行出现故障时，Master 会再分配其他 Executor 运行任务，此时会调用 updateEpoch 方法（见代码清单 5-61）更新纪元，并且清空 mapStatuses。

代码清单5-61 更新Executor的纪元

---

```
def updateEpoch(newEpoch: Long) {
    epochLock.synchronized {
        if (newEpoch > epoch) {
            logInfo("Updating epoch to " + newEpoch + " and clearing cache")
            epoch = newEpoch
            mapStatuses.clear()
        }
    }
}
```

---

## 7. unregisterShuffle

unregisterShuffle 方法用于 ContextCleaner 清除 shuffleId 对应 MapStatus 的信息，其实现如代码清单 5-62 所示。

代码清单5-62 清除shuffleId对应MapStatus的信息

---

```
def unregisterShuffle(shuffleId: Int) {
    mapStatuses.remove(shuffleId)
}
```

---

## 5.6.2 MapOutputTrackerMaster 的实现原理

通常而言，我们所说的 mapOutputTracker 都是指 MapOutputTrackerMaster，而不是 MapOutputTrackerWorker。MapOutputTrackerWorker 将 map 任务的跟踪信息，通过 MapOutputTrackerMasterEndpoint 的 RpcEndpointRef 发送给 MapOutputTrackerMaster，由 MapOutputTrackerMaster 负责整理和维护所有的 map 任务的输出跟踪信息。MapOutputTrackerMasterEndpoint 位于 MapOutputTrackerMaster 内部，二者只存在于 Driver 上。有了对 MapOutputTrackerMaster 的基本认识，下面详细分析它的实现。

为了便于理解 MapOutputTrackerMaster 的实现原理，首先了解它的各个属性，将使我们受益良多。MapOutputTrackerMaster 的属性包括以下几项。

- ❑ cacheEpoch：对 MapOutputTracker 的 epoch 的缓存。
- ❑ minSizeForBroadcast：用于广播的最小大小。可以使用 spark.shuffle.mapOutput.minSizeForBroadcast 属性配置，默认为 512KB。minSizeForBroadcast 必须小于 maxRpcMessageSize。
- ❑ shuffleLocalityEnabled：是否为 reduce 任务计算本地性的偏好。可以使用 spark.shuffle.reduceLocality.enabled 属性进行配置，默认为 true。
- ❑ mapStatuses：用于存储 shuffleId 与 Array[MapStatus] 的映射关系。由于 MapStatus

维护了 map 输出 Block 的地址 BlockManagerId，所以 reduce 任务知道从何处获取 map 任务的中间输出。在 6.1.2 节对 BlockManagerId 有更加详细的介绍。

- cachedSerializedStatuses：用于存储 shuffleId 与序列化后的状态的映射关系。其中 key 对应 shuffleId，value 为对 MapStatus 序列化后的字节数组。
- maxRpcMessageSize：最大的 Rpc 消息的大小。此属性可以通过 spark.rpc.message.maxSize 属性进行配置，默认为 128，单位是 MB。minSizeForBroadcast 必须小于 maxRpcMessageSize。maxRpcMessageSize 实际通过调用 RpcUtils 的 maxMessageSizeBytes 方法获得，其具体实现请参阅附录 H。
- cachedSerializedBroadcast：用于缓存序列化的广播变量，保持与 cachedSerializedStatuses 的同步。当需要移除 shuffleId 在 cachedSerializedStatuses 中的状态数据时，此缓存中的数据也会被移除。
- shuffleIdLocks：每个 shuffleId 对应的锁。当 shuffle 过程开始时，会有大量的关于同一个 shuffle 的请求，使用锁可以避免对同一 shuffle 的多次序列化。
- mapOutputRequests：使用阻塞队列来缓存 GetMapOutputMessage（获取 map 任务输出）的请求。
- threadpool：用于获取 map 输出的固定大小的线程池。此线程池提交的线程都以后台线程运行，且线程名以 map-output-dispatcher 为前缀，线程池大小可以使用 spark.shuffle.mapOutput.dispatcher.numThreads 属性配置，默认大小为 8。

### 1. MapOutputTrackerMaster 的运行原理

在创建 MapOutputTrackerMaster 的最后，会创建对 map 输出请求进行处理的线程池 threadpool，如代码清单 5-63 所示。

代码清单 5-63 创建 MapOutputTrackerMaster 中的线程池

---

```
private val threadpool: ThreadPoolExecutor = {
    val numThreads = conf.getInt("spark.shuffle.mapOutput.dispatcher.numThreads", 8)
    val pool = ThreadUtils.newDaemonFixedThreadPool(numThreads, "map-output-
dispatcher")
    for (i <- 0 until numThreads) {
        pool.execute(new MessageLoop)
    }
    pool
}
```

---

根据代码清单 5-63，可以看到创建 threadpool 线程池的步骤如下。

- 1) 获取此线程池的大小 numThreads。此线程池的大小默认为 8，也可以使用 spark.shuffle.mapOutput.dispatcher.numThreads 属性配置。
- 2) 创建线程池。此线程池是固定大小的线程池，并且启动的线程都以后台线程方式运行，且线程名以 map-output-dispatcher 为前缀。
- 3) 启动与线程池大小相同数量的线程，每个线程执行的任务都是 MessageLoop。

4) 返回此线程池的引用。

上面介绍的线程池 threadpool 的创建与 5.3.4 节介绍的 Dispatcher 内的线程池 threadpool 从命名和方式上非常相似，而且此处启动的任务也是 MessageLoop。不过这里的 MessageLoop 并非是 Dispatcher 中的 MessageLoop，而是 MapOutputTrackerMaster 的内部类。MapOutputTrackerMaster 中的 MessageLoop 也实现了 Java 的 Runnable 接口，其实现如代码清单 5-64 所示。

代码清单5-64 MessageLoop的实现

---

```

private class MessageLoop extends Runnable {
    override def run(): Unit = {
        try {
            while (true) {
                try {
                    val data = mapOutputRequests.take()
                    if (data == PoisonPill) {
                        // Put PoisonPill back so that other MessageLoops can see it.
                        mapOutputRequests.offer(PoisonPill)
                        return
                    }
                    val context = data.context
                    val shuffleId = data.shuffleId
                    val hostPort = context.senderAddress.hostPort
                    logDebug("Handling request to send map output locations for shuffle "
                            + shuffleId +
                            " to " + hostPort)
                    val mapOutputStatuses = getSerializedMapOutputStatuses(shuffleId)
                    context.reply(mapOutputStatuses)
                } catch {
                    case NonFatal(e) => logError(e.getMessage, e)
                }
            }
        } catch {
            case ie: InterruptedException => // exit
        }
    }
}

```

---

根据代码清单 5-64，我们看到 MessageLoop 果如其名，在循环过程中不断对新的消息进行处理。每次循环中的逻辑如下。

1) 从 mapOutputRequests 中获取 GetMapOutputMessage。由于 mapOutputRequests 是个阻塞队列，所以当 mapOutputRequests 中没有 GetMapOutputMessage 时，MessageLoop 线程会被阻塞。GetMapOutputMessage 是个样例类，包含了 shuffleId 和 RpcCallContext 两个属性，RpcCallContext 用于服务端回复客户端，其实现如下。

```
private[spark] case class GetMapOutputMessage(shuffleId: Int, context: RpcCallContext)
```

2) 如果取到的 GetMapOutputMessage 是 PoisonPill (“毒药”)，那么此 MessageLoop 线程将退出（通过 return 语句）。这里有个动作，就是将 PoisonPill 重新放入到 mapOutput-

Requests 中，这是因为 threadpool 线程池极有可能不止一个 MessageLoop 线程，为了让大家都“毒发身亡”，还需要把“毒药”放回到 receivers 中，这样其他“活着”的线程就会再次误食“毒药”，达到所有 MessageLoop 线程都结束的效果。

3) 如果取到的 GetMapOutputMessage 不是“毒药”，那么调用 getSerializedMapOutputStatuses 方法获取 GetMapOutputMessage 携带的 shuffleId 所对应的序列化任务状态信息。

4) 调用 RpcCallContext 的回调方法 reply，将序列化的 map 任务状态信息返回给客户端（即其他节点的 Executor）。

MessageLoop 任务执行的第 3) 步调用了 getSerializedMapOutputStatuses 方法获取序列化任务状态信息，其具体实现如代码清单 5-65 所示。

代码清单5-65 获取序列化任务状态信息

---

```

def getSerializedMapOutputStatuses(shuffleId: Int): Array[Byte] = {
    var statuses: Array[MapStatus] = null
    var retBytes: Array[Byte] = null
    var epochGotten: Long = -1
    // 省略次要代码
    if (checkCachedStatuses()) return retBytes
    var shuffleIdLock = shuffleIdLocks.get(shuffleId)
    if (null == shuffleIdLock) {
        val newLock = new Object()
        val prevLock = shuffleIdLocks.putIfAbsent(shuffleId, newLock)
        shuffleIdLock = if (null != prevLock) prevLock else newLock
    }
    shuffleIdLock.synchronized {
        if (checkCachedStatuses()) return retBytes
        val (bytes, bcast) = MapOutputTracker.serializeMapStatuses(statuses,
            broadcastManager,
            isLocal, minSizeForBroadcast)
        logInfo("Size of output statuses for shuffle %d is %d bytes".format (shuffleId,
            bytes.length))
        // Add them into the table only if the epoch hasn't changed while we were
        // working
        epochLock.synchronized {
            if (epoch == epochGotten) {
                cachedSerializedStatuses(shuffleId) = bytes
                if (null != bcast) cachedSerializedBroadcast(shuffleId) = bcast
            } else {
                logInfo("Epoch changed, not caching!")
                removeBroadcast(bcast)
            }
        }
        bytes
    }
}

```

---

根据代码清单 5-65，getSerializedMapOutputStatuses 方法的执行步骤如下。

1) 调用 checkCachedStatuses 检查 cachedSerializedStatuses 中是否有 shuffleId 所对应的序列化的任务状态信息。如果有，则从 cachedSerializedStatuses 中获取序列化的任务状

态信息后返回，否则进入第2)步。

2) 从 shuffleIdLocks 中获取 shuffleId 对应的锁，如果没有，则新建一个对象作为锁放入 shuffleIdLocks 中。

3) 获取 mapStatuses 中缓存的 MapStatus 数组，然后调用 MapOutputTracker 的 serializeMapStatuses 方法将 MapStatus 数组序列化，最后调用 BroadcastManager 的 newBroadcast 方法（详见 5.5 节）将序列化 MapStatus 数组后产生的字节数组进行广播。serializeMapStatuses 方法将返回序列化的 MapStatus（即 bytes）和广播对象 bcast 的对偶。

4) 如果当前 epoch 没有发生变化，将 shuffleId 与序列化的 MapStatus 之间的映射关系放入 cachedSerializedStatuses，并将 shuffleId 与广播对象 bcast 之间的映射关系放入 cachedSerializedBroadcast。

5) 如果当前 epoch 已经发生变化，则不需要对 bytes 和 bcast 进行缓存，同时调用 removeBroadcast 方法删除广播对象 bcast。

getSerializedMapOutputStatuses 方法中调用了 removeBroadcast 方法移除广播对象。removeBroadcast 的实现如代码清单 5-66 所示。

代码清单 5-66 移除广播对象

---

```
private def removeBroadcast(bcast: Broadcast[]): Unit = {
    if (null != bcast) {
        broadcastManager.unbroadcast(bcast.id,
            removeFromDriver = true, blocking = false)
    }
}
```

---

根据代码清单 5-66，removeBroadcast 方法的功能实际委托给 BroadcastManager 的 unbroadcast 方法（在 5.5 节有介绍）处理了。

MapOutputTrackerMaster 中的 MessageLoop 任务在不断地消费着阻塞队列 mapOutputRequests 中的 GetMapOutputMessage，那么 GetMapOutputMessage 的来源在哪里？GetMapOutputMessage 又是如何放入 mapOutputRequests 中的呢？在 MapOutputTrackerMaster 中要将 GetMapOutputMessage 放入 mapOutputRequests，主要通过调用 MapOutputTrackerMaster 的 post 方法，其实现如代码清单 5-67 所示。

代码清单 5-67 投递 map 输出结果获取请求

---

```
def post(message: GetMapOutputMessage): Unit = {
    mapOutputRequests.offer(message)
}
```

---

MapOutputTrackerMasterEndpoint 组件用于接获取 map 中间状态和停止对 map 中间状态进行跟踪的请求，它实现了特质 RpcEndpoint 并重写了 receiveAndReply 方法，如代码清单 5-68 所示。

**代码清单5-68 MapOutputTrackerMasterEndpoint的定义**


---

```

private[spark] class MapOutputTrackerMasterEndpoint(
    override val rpcEnv: RpcEnv, tracker: MapOutputTrackerMaster, conf: SparkConf)
  extends RpcEndpoint with Logging {
  // 省略次要代码
  override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
    case GetMapOutputStatuses(shuffleId: Int) =>
      val hostPort = context.senderAddress.hostPort
      logInfo("Asked to send map output locations for shuffle " + shuffleId + " to " + hostPort)
      val mapOutputStatuses = tracker.post(new GetMapOutputMessage(shuffleId, context))
    case StopMapOutputTracker =>
      logInfo("MapOutputTrackerMasterEndpoint stopped!")
      context.reply(true)
      stop()
  }
}

```

---

根据代码清单 5-68，可以看到 MapOutputTrackerMasterEndpoint 将接收两种类型的请求。

- 获取 map 中间输出状态。当接收到 GetMapOutputStatuses 消息（主要由 MapOutputTracker 的 getStatues 方法调用 askTracker 时产生，如代码清单 5-58 所示）后，将调用 MapOutputTrackerMaster 的 post 方法投递 GetMapOutputMessage 类型的消息。
- 停止 map 中间输出的跟踪。首先回调 RpcCallContext 的 reply 方法向客户端返回 true，然后调用 MapOutputTrackerMaster 的 stop 方法（见代码清单 5-69）停止 MapOutputTrackerMaster。

**代码清单5-69 停止MapOutputTrackerMaster**


---

```

override def stop() {
  mapOutputRequests.offer(PoisonPill)
  threadpool.shutdown()
  sendTracker(StopMapOutputTracker)
  mapStatuses.clear()
  trackerEndpoint = null
  cachedSerializedStatuses.clear()
  clearCachedBroadcast()
  shuffleIdLocks.clear()
}

```

---

根据代码清单 5-69，可以看到 stop 方法的执行步骤如下。

- 1) 向 mapOutputRequests 投递 PoisonPill（“毒药”）。此操作会向 mapOutputRequests 队列中添加 PoisonPill，进而停止 MapOutputTrackerMaster 中的所有 MessageLoop 线程。
- 2) 关闭 threadpool。由于 MapOutputTrackerMaster 中的所有 MessageLoop 线程都已经停止，因而可以平滑关闭 threadpool。

3) 调用 sendTracker 向 MapOutputTrackerMasterEndpoint 发送 StopMapOutputTracker 消息。这里有个小问题，那就是 MapOutputTrackerMasterEndpoint 再次收到 StopMapOutputTracker 消息，进而引起 stop 方法的循环调用。不知道这是否会引起问题。

- 4) 清空 mapStatuses。
- 5) 回收 MapOutputTrackerMasterEndpoint。
- 6) 清空 cachedSerializedStatuses。
- 7) 清空 shuffleIdLocks。

有了以上的一系列分析，现在我们可以用图 5-13 展现 MapOutputTrackerMaster 的运行原理。

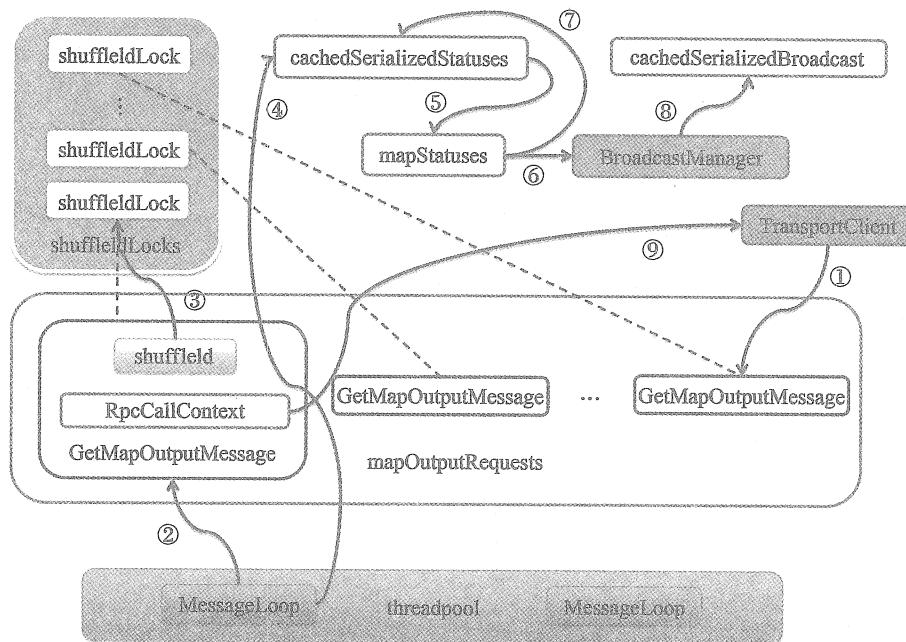


图 5-13 MapOutputTrackerMaster 的运行原理简图

这里对图 5-13 的内容再做一次简短描述。

序号①：表示某个 Executor 调用 MapOutputTrackerWorker 的 getStatues 方法获取某个 shuffle 的 map 任务状态信息，当发现本地的 mapStatuses 没有相应的缓存，则调用 askTracker 方法发送 GetMapOutputStatuses 消息。根据代码清单 5-56 的内容我们知道，askTracker 实际是通过 MapOutputTrackerMasterEndpoint 的 NettyRpcEndpointRef 向远端发送 GetMapOutputStatuses 消息。发送实际依托于 NettyRpcEndpointRef 持有的 Transport-Client。MapOutputTrackerMasterEndpoint 在接收到 GetMapOutputStatuses 消息后，将 GetMapOutputMessage 消息放入 mapOutput Requests 队尾。

序号②：表示 MessageLoop 线程从 mapOutputRequests 队头取出 GetMapOutputMessage。

序号③：表示从 shuffleIdLocks 数组中取出与当前 GetMapOutputMessage 携带的 shuffleId 相对应的锁。

序号④：表示首先从 cachedSerializedStatuses 缓存中获取 shuffleId 对应的序列化任务状态信息。

序号⑤：表示当 cachedSerializedStatuses 中没有 shuffleId 对应的序列化任务状态信息，则获取 mapStatuses 中缓存的 shuffleId 对应的任务状态数组。

序号⑥：表示将任务状态数组进行序列化，然后使用 BroadcastManager 对序列化的任务状态进行广播。

序号⑦：表示将序列化的任务状态放入 cachedSerializedStatuses 缓存中。

序号⑧：表示将广播对象放入 cachedSerializedBroadcast 缓存中。

序号⑨：表示将获得的序列化任务状态信息，通过回调 GetMapOutputMessage 消息携带的 RpcCallContext 的 reply 方法回复客户端。

## 2. Shuffle 的注册

DAGScheduler 在创建 ShuffleMapStage 的时候（将在第 7.4.7 节介绍），将调用 MapOutputTrackerMaster 的 containsShuffle 方法（见代码清单 5-70），查看是否已经存在 shuffleId 对应的 MapStatus。如果 MapOutputTrackerMaster 中未注册此 shuffleId，那么调用 MapOutputTrackerMaster 的 registerShuffle 方法（见代码清单 5-71）注册 shuffleId。

containsShuffle 方法（见代码清单 5-70）用于查找 MapOutputTrackerMaster 的 cachedSerializedStatuses 或 mapStatuses 中是否已经注册了 shuffleId。

代码清单 5-70 containsShuffle 的实现

---

```
def containsShuffle(shuffleId: Int): Boolean = {
    cachedSerializedStatuses.contains(shuffleId) || mapStatuses.contains(shuffleId)
}
```

---

registerShuffle 方法（见代码清单 5-71）用于向 MapOutputTrackerMaster 的 mapStatuses 中注册 shuffleId 与对应的 MapStatus 的映射关系。

代码清单 5-71 registerShuffle 的实现

---

```
def registerShuffle(shuffleId: Int, numMaps: Int) {
    if (mapStatuses.put(shuffleId, new Array[MapStatus](numMaps)).isDefined) {
        throw new IllegalArgumentException("Shuffle ID " + shuffleId + " registered twice")
    }
    shuffleIdLocks.putIfAbsent(shuffleId, new Object())
}
```

---

根据代码清单 5-71，可以看到注册 shuffleId 时，shuffleId 在 mapStatuses 中对应的是以 map 任务数量作为长度的数组，此数组将用于保存 MapStatus。注册了 Shuffle，那么 MapStatus 是何时保存的呢？当 ShuffleMapStage 内的所有 ShuffleMapTask 运行成功后，将

调用 MapOutputTrackerMaster 的 registerMapOutputs 方法 (ShuffleMapTask 运行结果的处理将在 7.4.11 节介绍)。registerMapOutputs 方法的实现如代码清单 5-72 所示。

代码清单5-72 registerMapOutputs方法

---

```
def registerMapOutputs(shuffleId: Int, statuses: Array[MapStatus], changeEpoch: Boolean = false)
{
    mapStatuses.put(shuffleId, statuses.clone())
    if (changeEpoch) {
        incrementEpoch()
    }
}
```

---

根据代码清单 5-72, registerMapOutputs 方法将把 ShuffleMapStage 中每个 ShuffleMapTask 的 MapStatus 保存到 shuffleId 在 mapStatuses 中对应的数组中。

整个 Shuffle 的注册流程可以用图 5-14 来表示。

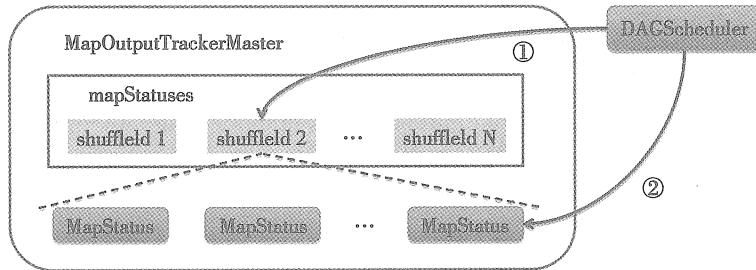


图 5-14 Shuffle 的注册流程

其处理步骤包括以下两步。

序号①：表示 DAGScheduler 在创建了 ShuffleMapStage 后，调用 MapOutputTrackerMaster 的 registerShuffle 方法向 mapStatuses 缓存注册 shuffleid。

序号②：表示 DAGScheduler 处理 ShuffleMapTask 的执行结果时，如果发现 ShuffleMapTask 所属的 ShuffleMapStage 中每一个分区的 ShuffleMapTask 都执行成功了，那么将调用 MapOutputTrackerMaster 的 registerMapOutputs 方法，将 ShuffleMapStage 中每一个 ShuffleMapTask 的 MapStatus 保存到 shuffleid 对应的 MapStatus 数组中。

## 5.7 构建存储体系

早在介绍 BroadcastManager 的时候，我们就知道 BroadcastManager 的底层都依赖于 SparkEnv 的存储体系。存储体系中最重要的组件包括 Shuffle 管理器 ShuffleManager、内存管理器 MemoryManager、块传输服务 BlockTransferService、对所有 BlockManager 进行管理的 BlockManagerMaster、磁盘块管理器 DiskBlockManager、块锁管理器 BlockInfoManager 及

块管理器 BlockManager。存储体系的内容将在第 6 章详细介绍，本节只简单介绍 SparkEnv 中存储体系各个组件的实例化，如代码清单 5-73 所示。

代码清单 5-73 构建存储体系

---

```

val shortShuffleMgrNames = Map(
  "sort" -> classOf[org.apache.spark.shuffle.sort.SortShuffleManager].getName,
  "tungsten-sort" -> classOf[org.apache.spark.shuffle.sort.SortShuffleManager].
    getName)
val shuffleMgrName = conf.get("spark.shuffle.manager", "sort")
val shuffleMgrClass = shortShuffleMgrNames.getOrElse(shuffleMgrName.toLowerCase,
  shuffleMgrName)
val shuffleManager = instantiateClass[ShuffleManager](shuffleMgrClass)
val useLegacyMemoryManager = conf.getBoolean("spark.memory.useLegacyMode", false)
val memoryManager: MemoryManager =
  if (useLegacyMemoryManager) {
    new StaticMemoryManager(conf, numUsableCores)
  } else {
    UnifiedMemoryManager(conf, numUsableCores)
  }
val blockManagerPort = if (isDriver) {
  conf.get(DRIVER_BLOCK_MANAGER_PORT)
} else {
  conf.get(BLOCK_MANAGER_PORT)
}
val blockTransferService =
  new NettyBlockTransferService(conf, securityManager, bindAddress, advertiseAddress,
    blockManagerPort, numUsableCores)
val blockManagerMaster = new BlockManagerMaster(registerOrLookupEndpoint(
  BlockManagerMaster.DRIVER_ENDPOINT_NAME,
  new BlockManagerMasterEndpoint(rpcEnv, isLocal, conf, listenerBus)),
  conf, isDriver)
val blockManager = new BlockManager(executorId, rpcEnv, blockManagerMaster,
  serializerManager, conf, memoryManager, mapOutputTracker, shuffleManager,
  blockTransferService, securityManager, numUsableCores)

```

---

根据代码清单 5-73，我们知道 SparkEnv 中存储体系各个组件的实例化步骤如下。

1) 根据 spark.shuffle.manager 属性，实例化 ShuffleManager。Spark 2.x.x 版本提供了 sort 和 tungsten-sort 两种 ShuffleManager 的实现。无论是 sort 还是 tungsten-sort，我们看到其实现类都是 SortShuffleManager。本书将在第 8 章详细介绍 SortShuffleManager。

2) MemoryManager 的主要实现有 StaticMemoryManager 和 UnifiedMemoryManager。StaticMemoryManager 是 Spark 早期版本遗留下来的内存管理器实现，可以配置 spark.memory.useLegacyMode 属性来指定，该属性默认为 false，因此默认的内存管理器是 UnifiedMemoryManager。

3) 获取当前 SparkEnv 的块传输服务 BlockTransferService 对外提供的端口号。如果当前实例是 Driver，则从 SparkConf 中获取由常量 DRIVER\_BLOCK\_MANAGER\_PORT 指定的端口。如果当前实例是 Executor，则从 SparkConf 中获取由常量 BLOCK\_MANAGER\_PORT 指定的端口。这两个常量的定义如下。

```

private[spark] val BLOCK_MANAGER_PORT = ConfigBuilder("spark.blockManager.port")
  .doc("Port to use for the block manager when a more specific setting is not
       provided.")
  .intConf
  .createWithDefault(0)
private[spark] val DRIVER_BLOCK_MANAGER_PORT = ConfigBuilder("spark.driver.
  blockManager.port")
  .doc("Port to use for the block manager on the driver.")
  .fallbackConf(BLOCK_MANAGER_PORT)

```

因此我们知道，可以通过指定 spark.driver.blockManager.port 属性或者 spark.blockManager.port 属性对 BlockTransferService 的端口进行配置。

4) 创建块传输服务 BlockTransferService。这里使用的是 BlockTransferService 的子类 NettyBlockTransferService，NettyBlockTransferService 将提供对外的块传输服务。也正是因为 MapOutputTracker 与 NettyBlockTransferService 的配合，才实现了 Spark 的 Shuffle。

5) 查找或注册 BlockManagerMasterEndpoint。这里的 registerOrLookupEndpoint 方法（见代码清单 5-55）已在 5.6 节介绍过，因此：

① 当前应用程序是 Driver，则创建 BlockManagerMasterEndpoint，并且注册到 Dispatcher 中，注册名为 BlockManagerMaster；

② 当前应用程序是 Executor，则从远端 Driver 实例的 NettyRpcEnv 的 Dispatcher 中查找 BlockManagerMasterEndpoint 的引用。

无论是 Driver 还是 Executor，最后都由 BlockManagerMaster 的属性 driverEndpoint 持有 BlockManagerMasterEndpoint 的引用。

**小贴士：**在 Spark 1.x.x 版本中存在 BlockManagerMasterActor，BlockManagerMaster-Endpoint 如今替代了前者。

6) 创建 BlockManagerMaster。

7) 创建 BlockManager。此处只创建了 BlockManager，只有其 init 方法被调用后，BlockManager 才能正常工作。

## 5.8 创建度量系统

在 3.4 节，我们详细介绍了度量系统的 basic 架构与概念。在 SparkEnv 中，度量系统也是必不可少的一个子组件。因此本节将详细介绍 SparkEnv 中度量系统的创建，如代码清单 5-74 所示。

代码清单5-74 SparkEnv中度量系统的创建

---

```

val metricsSystem = if (isDriver) {
  MetricsSystem.createMetricsSystem("driver", conf, securityManager)
} else {

```

---

```

    conf.set("spark.executor.id", executorId)
    val ms = MetricsSystem.createMetricsSystem("executor", conf, securityManager)
    ms.start()
    ms
}

```

---

根据代码清单 5-74，我们知道创建度量系统根据当前实例是 Driver 还是 Executor 也有区别。

- 当前实例为 Driver：创建度量系统，并且制定度量系统的实例名为 driver。此时虽然创建了，但是并未启动，目的是等待 SparkContext 中的任务调度器 Task Scheduler 告诉度量系统应用程序 ID 后再启动。
- 当前实例为 Executor：设置 spark.executor.id 属性为当前 Executor 的 ID，然后创建并启动度量系统（启动 MetricsSystem 的内容详见 5.8.3 节）。

创建度量系统使用了伴生对象 MetricsSystem 的 createMetricsSystem 方法（类似于 Java 的静态方法），其实现如代码清单 5-75 所示。

**代码清单5-75 createMetricsSystem的实现**

---

```

def createMetricsSystem(
    instance: String, conf: SparkConf, securityMgr: SecurityManager):
    MetricsSystem = {
        new MetricsSystem(instance, conf, securityMgr)
}

```

---

根据代码清单 5-75，createMetricsSystem 方法实际利用了 MetricsSystem 的构造器来创建度量系统。

要理解 MetricsSystem 的实现原理，首先应当从了解它的成员属性开始，MetricsSystem 拥有的成员属性如下。

- instance：度量系统的实例名。例如，Master、Worker、Application、Driver 及 Executor 等。
- metricsConfig：度量配置。metricsConfig 的类型为 MetricsConfig，主要提供对度量配置的设置、加载、转换等功能。MetricsConfig 中的度量配置包括了 Sink 和 Source，因此 MetricsSystem 将根据 MetricsConfig 构建度量系统的所有 Sink 和 Source。
- sinks：Sink 的数组。在 3.4.2 节介绍过 Sink（即度量输出）的设计与实现。sinks 用于缓存所有注册到 MetricsSystem 的度量输出。
- sources：Source 的数组。在 3.4.1 节介绍过 Source（即度量的采集或来源）的设计与实现。sources 用于缓存所有注册到 MetricsSystem 的 Source。
- registry：度量注册点 MetricRegistry。Source 和 Sink 实际都是通过 MetricRegistry 注册到 Metrics 的度量仓库中的。Metrics 是 codahale 提供的第三方度量仓库，这里的 MetricRegistry 是 Metrics 提供的 API，对于 Metrics 及 MetricRegistry 的详细内

容请参阅附录 D。

- running: 用于标记当前 MetricsSystem 是否正在运行。
- metricsServlet: metricsServlet 将在添加 ServletContextHandler 后通过 Web UI 展示。metricsServlet 的类型是 Option[MetricsServlet]。

下面将逐一介绍 MetricsSystem 内的这些成员。

### 5.8.1 MetricsConfig 详解

MetricsConfig 保存了 MetricsSystem 的配置信息，有以下两个成员属性。

- properties: 度量的属性信息。类型为 Properties。
- perInstanceSubProperties : 每个实例的子属性。缓存每个实例与其属性的映射关系，类型为 HashMap[String, Properties]。

有了对 MetricsConfig 中属性的了解，我们来看看 MetricsConfig 中的几个方法。

#### 1. 设置默认属性

setDefaultProperties 方法（见代码清单 5-76）用于给 MetricsConfig 的 properties 中添加默认的度量属性。

代码清单5-76 设置默认的度量属性

---

```
private def setDefaultProperties(prop: Properties) {
    prop.setProperty("*.sink.servlet.class", "org.apache.spark.metrics.sink.
MetricsServlet")
    prop.setProperty("*.sink.servlet.path", "/metrics/json")
    prop.setProperty("master.sink.servlet.path", "/metrics/master/json")
    prop.setProperty("applications.sink.servlet.path", "/metrics/applications/json")
}
```

---

根据代码清单 5-76，MetricsConfig 一共有四个默认属性。

#### 2. 从文件中加载度量属性

loadPropertiesFromFile 方法（见代码清单 5-77）用于给 MetricsConfig 的 properties 中添加从指定文件中加载的度量属性。

代码清单5-77 从文件中加载度量属性

---

```
private[this] def loadPropertiesFromFile(path: Option[String]): Unit = {
    var is: InputStream = null
    try {
        is = path match {
            case Some(f) => new FileInputStream(f)
            case None => Utils.getSparkClassLoader.getResourceAsStream(DEFAULT_METRICS_
                CONF_FILENAME)
        }
        if (is != null) {
            properties.load(is)
        }
    }
```

---

```

    } catch {
      case e: Exception =>
        val file = path.getOrElse(DEFAULT_METRICS_CONF_FILENAME)
        logError(s"Error loading configuration file $file", e)
    } finally {
      if (is != null) {
        is.close()
      }
    }
}

```

---

根据代码清单 5-77，我们知道如果没有指定度量属性文件或者此文件不存在，那么将从类路径下的属性文件 metrics.properties（即常量 DEFAULT\_METRICS\_CONF\_FILENAME 的值）中加载度量属性。读取类路径下的文件时使用了工具类 Utils 的 getSparkClassLoader 方法，其具体实现可以参阅附录 A。

### 3. 提取实例的属性

subProperties 方法（见代码清单 5-78）对于 properties 中的每个属性 kv，通过正则表达式匹配找出 kv 的 key 的前缀和后缀，以前缀为实例，后缀作为新的属性 kv2 的 key，kv 的 value 作为新的属性 kv2 的 value，最后将属性 kv2 添加到实例对应的属性集合中作为实例的属性。这些解释过于书面，在介绍 MetricsConfig 的初始化时，将会有更直观的例子。

代码清单5-78 提取实例的属性

---

```

def subProperties(prop: Properties, regex: Regex): mutable.HashMap[String, Properties] = {
  val subProperties = new mutable.HashMap[String, Properties]
  prop.asScala.foreach { kv =>
    if (regex.findPrefixOf(kv._1.toString).isDefined) {
      val regex(prefix, suffix) = kv._1.toString
      subProperties.getOrElseUpdate(prefix, new Properties).setProperty(suffix,
        kv._2.toString)
    }
  }
  subProperties
}

```

---

### 4. 初始化

在构造 MetricsSystem 的过程中将会执行以下代码。

```
metricsConfig.initialize()
```

MetricsConfig 的 initialize 方法的实现如代码清单 5-79 所示。

代码清单5-79 MetricsConfig的初始化

---

```

def initialize() {
  setDefaultProperties(properties)
  loadPropertiesFromFile(conf.getOption("spark.metrics.conf"))
  val prefix = "spark.metrics.conf."
}

```

```

conf.getAll.foreach {
  case (k, v) if k.startsWith(prefix) =>
    properties.setProperty(k.substring(prefix.length()), v)
  case _ =>
}

perInstanceSubProperties = subProperties(properties, INSTANCE_REGEX)
if (perInstanceSubProperties.contains(DEFAULT_PREFIX)) {
  val defaultSubProperties = perInstanceSubProperties(DEFAULT_PREFIX).asScala
  for ((instance, prop) <- perInstanceSubProperties if (instance != DEFAULT_PREFIX);
       (k, v) <- defaultSubProperties if (prop.get(k) == null)) {
    prop.put(k, v)
  }
}
}

```

---

根据代码清单 5-79，MetricsConfig 的初始化过程如下。

- 1 ) 设置默认属性。setDefaultProperties 的实现如代码清单 5-76 所示。
- 2 ) 从文件中加载度量属性。可以通过 spark.metrics.conf 属性指定度量属性文件。loadPropertiesFromFile 的实现如代码清单 5-77 所示。
- 3 ) 从 SparkConf 中查找以 spark.metrics.conf 为前缀的配置属性，并且截取 key 的前缀后的部分作为度量属性的 key/value 不变。
- 4 ) 提取实例的属性。这里指定了正则表达式 `^(*|[a-zA-Z]+)\.(.+)` 找出 properties 中各个实例的属性。subProperties 的实现如代码清单 5-78 所示。

为了更容易理解代码清单 5-78 的实现，这里以代码清单 5-76 中的 KV 对 `"*.sink.servlet.class"->"org.apache.spark.metrics.sink.MetricsServlet"` 和 `"*.sink.servlet.path" -> "/metrics/json"` 为例。首先使用正则表达式 `^(*|[a-zA-Z]+)\.(.+)` 匹配出两个元组，那么第一个 KV 对的两个元组分别为 \* 和 sink.servlet.class。第二个 KV 对的两个元组分别为 \* 和 sink.servlet.path。然后以第一个元组作为前缀，第二个元组作为后缀。以前缀为实例，后缀为新属性的 key，KV 对的 value 作为新属性的 value。最后将新属性添加到实例对应的 Properties 中，将实例与 Properties 之间的映射关系存入 Map。结果如下。

```

Map (*->{
  sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet,
  sink.servlet.path=/metrics/json
})

```

根据这个例子，4 个默认度量属性提取出实例的属性后为：

```

Map (applications->{
  sink.servlet.path=/metrics/applications/json
},
master ->{
  sink.servlet.path=/metrics/master/json
},
*->{

```

```

    sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet,
    sink.servlet.path=/metrics/json
})

```

即实例 applications、master、\* 与它们对应的属性集合。

5) 向子属性中添加缺失的默认子属性（所谓默认子属性，即在 Map 中以 \* 为 key 的属性）。为了理解更加简单，这里以第 4) 步得到的 Map 为例。默认子属性中的 sink.servlet.class 属性是其他两个（applications 和 master）子属性中不具有的属性，因此需要将此属性添加到以 applications 和 master 为 key 的子属性中。因此得到的结果为：

```

Map(applications->{
    sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet,
    sink.servlet.path=/metrics/applications/json
},
master->{
    sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet,
    sink.servlet.path=/metrics/master/json
},
*->{
    sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet,
    sink.servlet.path=/metrics/json
})

```

经过以上处理，每个度量实例与其属性集合的信息都存储在 perInstanceSubProperties 了。细心的读者可能发现在默认配置下，Spark 度量系统的属性配置只有 Sink，却没有 Source，没有了度量源，光有输出能有什么用？度量系统中的 Sink 都通过 Spark 属性或者从文件中加载，而 Source 除了与 Sink 相同的两种方式外，Spark 的实现中，很多地方都是调用 MetricsSystem 的 registerSource 方法（将在 5.8.2 节介绍）注册各种 Source 的。虽然如此，有些 Source 依然需要你在 Spark 属性或者指定的文件中明确配置，就像下面这样。

```
master.source.jvm.class=org.apache.spark.metrics.source.JvmSource
```

## 5. 获取实例的属性

getInstance 方法（见代码清单 5-80）用于获取指定实例的属性。

代码清单 5-80 获取指定实例的属性

---

```

def getInstance(inst: String): Properties = {
  perInstanceSubProperties.get(inst) match {
    case Some(s) => s
    case None => perInstanceSubProperties.getOrElse(DEFAULT_PREFIX, new
      Properties)
  }
}

```

---

根据代码清单 5-80，getInstance 方法优先从 perInstanceSubProperties 中获取指定实例的属性，如果 perInstanceSubProperties 中没有存储对应的实例，则返回默认实例（即 \*）的属性。

## 5.8.2 MetricsSystem 中的常用方法

MetricsSystem 中的有些方法使用频度较高，所以本节对这些方法进行介绍。

### 1. 构建度量源的注册名

`buildRegistryName` 方法（见代码清单 5-81）用于给 Source 生成向 MetricRegistry 中注册的注册名。

代码清单5-81 构建度量源的注册名

---

```
private[spark] def buildRegistryName(source: Source): String = {
    val metricsNamespace = conf.get(METRICS_NAMESPACE).orElse(conf.getOption("spark.app.id"))
    val executorId = conf.getOption("spark.executor.id")
    val defaultName = MetricRegistry.name(source.sourceName)
    if (instance == "driver" || instance == "executor") {
        if (metricsNamespace.isDefined && executorId.isDefined) {
            MetricRegistry.name(metricsNamespace.get, executorId.get, source.sourceName)
        } else {
            if (metricsNamespace.isEmpty) {
                logWarning(s"Using default name $defaultName for source because neither " +
                    s"${METRICS_NAMESPACE.key} nor spark.app.id is set.")
            }
            if (executorId.isEmpty) {
                logWarning(s"Using default name $defaultName for source because spark.
                    executor.id is " +
                    s"not set.")
            }
            defaultName
        }
    } else { defaultName }
}
```

---

根据代码清单 5-81，`buildRegistryName` 方法涉及的变量有以下几个。

1 ) `metricsNamespace`：度量命名空间。可以通过 `spark.metrics.namespace` 属性进行配置，默认读取 `spark.app.id` 属性的值。例如，笔者本地以 local 模式执行时，`TaskScheduler` 会设置这个 `spark.app.id` 属性为 `local-1490149388666`。

2 ) `executorId`：当前 Executor 的身份标识，通过读取 `spark.executor.id` 属性获得。例如，笔者本地以 local 模式执行时，`spark.executor.id` 属性为 `driver`。

3 ) `defaultName`：调用 `MetricRegistry` 的 `name` 方法（请参阅附录 D）生成的默认注册名。

有了对以上变量的解释，现在来看看 `buildRegistryName` 方法的执行逻辑。构建注册名的步骤如下。

① 如果定义了 `metricsNamespace` 和 `executorId`，那么调用 `MetricRegistry` 的 `name` 方法生成  `${metricsNamespace}.${executorId}.${defaultName}` 格式的注册名。例如，笔者本地以 local 模式执行时，针对 `Source` 的实现类 `CodegenMetrics`，生成的注册名为 `local-1490149388666.driver.CodeGenerator`。

② 如果 metricsNamespace 或 executorId 没有定义，那么采用 defaultName 为注册名。

## 2. 注册度量源

registerSource 方法（见代码清单 5-82）用于向 MetricsSystem 中注册度量源。MetricRegistry 的 register 方法的实现请参阅附录 D。

代码清单5-82 注册度量源

---

```
def registerSource(source: Source) {
    sources += source
    try {
        val regName = buildRegistryName(source)
        registry.register(regName, source.metricRegistry)
    } catch {
        case e: IllegalArgumentException => logInfo("Metrics already registered", e)
    }
}
```

---

## 3. 获取 ServletContextHandler

为了将度量系统和 Spark UI 结合起来使用，即将 Spark UI 的 Web 展现也作为度量系统的 Sink 之一，我们需要一个 ServletContextHandler 能在 Spark UI 中接收请求，并且还需要能将度量输出到 Spark UI 的页面。根据 3.4.2 节对 Sink 继承体系的介绍，MetricsServlet 是特质 Sink 的实现之一，但是它不是一个 ServletContextHandler，因此需有一个转换，正如代码清单 5-83 所实现的那样。

代码清单5-83 将MetricsServlet转换为ServletContextHandler

---

```
def getServletHandlers: Array[ServletContextHandler] = {
    require(running, "Can only call getServletHandlers on a running MetricsSystem")
    metricsServlet.map(_.getHandlers(conf)).getOrElse(Array())
}
```

---

代码清单 5-83 调用了 MetricsServlet 的 getHandlers 方法来实现转换，getHandlers 的实现如代码清单 5-84 所示。

代码清单5-84 MetricsServlet的getHandlers方法

---

```
def getHandlers(conf: SparkConf): Array[ServletContextHandler] = {
    Array[ServletContextHandler](
        createServletHandler(servletPath,
            new ServletParams(request => getMetricsSnapshot(request), "text/json"),
            securityMgr, conf)
    )
}
```

---

根据代码清单 5-84，getHandlers 也调用了 JettyUtils 的 createServletHandler 方法创建 ServletContextHandler。JettyUtils 的 createServletHandler 方法的实现请参阅附录 C。

### 5.8.3 启动 MetricsSystem

SparkEnv 会调用 MetricsSystem 的 start 方法启动 MetricsSystem, start 方法的实现如代码清单 5-85 所示。

代码清单5-85 启动MetricsSystem

---

```
def start() {
    require(!running, "Attempting to start a MetricsSystem that is already running")
    running = true
    StaticSources.allSources.foreach(registerSource)
    registerSources()
    registerSinks()
    sinks.foreach(_.start)
}
```

---

根据代码清单 5-85, 启动 MetricsSystem 的步骤如下。

- 1) 将当前 MetricsSystem 的 running 字段置为 true, 即表示 MetricsSystem 已经处于运行状态。
- 2) 将静态的度量来源 CodegenMetrics 和 HiveCatalogMetrics 注册到 MetricRegistry。registerSource 方法的实现如代码清单 5-82 所示。allSources 是一个序列, 其定义如下。

```
private[spark] object StaticSources {
  val allSources = Seq(CodegenMetrics, HiveCatalogMetrics)
}
```

- 3) 从初始化完成的 MetricsConfig 中获取当前实例的度量来源属性, 并调用 registerSources 方法 (见代码清单 5-86), 将这些度量来源注册到 MetricRegistry。

代码清单5-86 注册度量配置中的度量源

---

```
private def registerSources() {
  val instConfig = metricsConfig.getInstance(instance)
  val sourceConfigs = metricsConfig.subProperties(instConfig, MetricsSystem.
SOURCE_REGEX)
  // Register all the sources related to instance
  sourceConfigs.foreach { kv =>
    val classPath = kv._2.getProperty("class")
    try {
      val source = Utils.classForName(classPath).newInstance()
      registerSource(source.asInstanceOf[Source])
    } catch {
      case e: Exception => logError("Source class " + classPath + " cannot be
        instantiated", e)
    }
  }
}
```

---

根据代码清单 5-86, 注册度量配置中度量源的步骤如下。

- ① 获取当前实例的度量属性。根据 MetricsConfig 的 getInstance 方法 (见代码清单

5-80) 的逻辑，当实例不存在时将返回默认实例的属性。以 driver 实例来讲，默认不存在 driver 的实例属性，因此返回 \* 对应的属性，即：

```
*->{
    sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet,
    sink.servlet.path=/metrics/json
}
```

- ② 匹配正则表达式 ^source\.(+)\.(+), 获取所有度量源更细粒度的实例及属性。
- ③ 使用实例的 class 属性，通过 Java 反射生成度量源的实例，并调用 registerSource 方法将此度量源注册到 MetricRegistry。
- ④ 从初始化完成的 MetricsConfig 中获取当前实例的度量输出属性，并将这些度量输出注册到 sinks。registerSinks 方法的实现如代码清单 5-87 所示。

代码清单5-87 注册度量配置中的度量输出

---

```
private def registerSinks() {
    val instConfig = metricsConfig.getInstance(instance)
    val sinkConfigs = metricsConfig.subProperties(instConfig, MetricsSystem.SINK_REGEX)
    sinkConfigs.foreach { kv =>
        val classPath = kv._2.getProperty("class")
        if (null != classPath) {
            try {
                val sink = Utils.className(classPath)
                    .getConstructor(classOf[Properties], classOf[MetricRegistry], classOf[SecurityManager])
                    .newInstance(kv._2, registry, securityMgr)
                if (kv._1 == "servlet") {
                    metricsServlet = Some(sink.asInstanceOf[MetricsServlet])
                } else {
                    sinks += sink.asInstanceOf[Sink]
                }
            } catch {
                case e: Exception =>
                    logError("Sink class " + classPath + " cannot be instantiated")
                    throw e
            }
        }
    }
}
```

---

根据代码清单 5-87，注册度量配置中度量输出的步骤如下。

- ① 获取当前实例的度量属性。根据 MetricsConfig 的 getInstance 方法（见代码清单 5-80）的逻辑，当实例不存在时将返回默认实例的属性。以 driver 实例来讲，默认不存在 driver 的实例属性，因此返回 \* 对应的属性，即：

```
*->{
    sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet,
    sink.servlet.path=/metrics/json
}
```

② 匹配正则表达式 ^sink\.(.+)\.(.+), 获取所有度量输出更细粒度的实例及属性。根据 subProperties 方法的逻辑, 最后得到:

```
Map<String, Map<String, Object>> subProperties = Map.ofEntriesAsList(entries);
```

③ 使用实例的 class 属性, 通过 Java 反射生成度量输出的实例。如果当前实例是 servlet, 则由 metricsServlet 持有此 servlet 的引用, 否则将度量输出实例注册到数组缓冲 sinks 中。

④ 启动 sinks 中的全部度量输出实例。

## 5.9 输出提交协调器

当 Spark 应用程序使用了 Spark SQL (包括 Hive) 或者需要将任务的输出保存到 HDFS 时, 就会用到输出提交协调器 OutputCommitCoordinator, OutputCommitCoordinator 将决定任务是否可以提交输出到 HDFS。无论是 Driver 还是 Executor, 在 SparkEnv 中都包含了子组件 OutputCommitCoordinator。在 Driver 上注册了 OutputCommitCoordinatorEndpoint, 所有 Executor 上的 OutputCommitCoordinator 都是通过 OutputCommitCoordinatorEndpoint 的 RpcEndpointRef 来询问 Driver 上的 OutputCommitCoordinator, 是否能够将输出提交到 HDFS。

SparkEnv 中创建 OutputCommitCoordinator 的实现如代码清单 5-88 所示。

代码清单5-88 创建OutputCommitCoordinator

---

```
val outputCommitCoordinator = mockOutputCommitCoordinator.getOrElse {
    new OutputCommitCoordinator(conf, isDriver)
}
val outputCommitCoordinatorRef = registerOrLookupEndpoint("OutputCommitCoordinator",
    new OutputCommitCoordinatorEndpoint(rpcEnv, outputCommitCoordinator))
outputCommitCoordinator.coordinatorRef = Some(outputCommitCoordinatorRef)
```

---

根据代码清单 5-88, OutputCommitCoordinator 的创建步骤如下。

- 1 ) 新建 OutputCommitCoordinator 实例。
- 2 ) 如果当前实例是 Driver, 则创建 OutputCommitCoordinatorEndpoint, 并且注册到 Dispatcher 中, 注册名为 OutputCommitCoordinator。如果当前应用程序是 Executor, 则从远端 Driver 实例的 NettyRpcEnv 的 Dispatcher 中查找 OutputCommitCoordinatorEndpoint 的引用。
- 3 ) 无论是 Driver 还是 Executor, 最后都由 OutputCommitCoordinator 的属性 coordinatorRef 持有 OutputCommitCoordinatorEndpoint 的引用。

### 5.9.1 OutputCommitCoordinatorEndpoint 的实现

OutputCommitCoordinatorEndpoint 的实现非常简单, 如代码清单 5-89 所示。

**代码清单5-89 OutputCommitCoordinatorEndpoint的实现**


---

```

private[spark] class OutputCommitCoordinatorEndpoint(
    override val rpcEnv: RpcEnv, outputCommitCoordinator: OutputCommitCoordinator)
  extends RpcEndpoint with Logging {
  override def receive: PartialFunction[Any, Unit] = {
    case StopCoordinator =>
      logInfo("OutputCommitCoordinator stopped!")
      stop()
  }
  override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
    case AskPermissionToCommitOutput(stage, partition, attemptNumber) =>
      context.reply(
        outputCommitCoordinator.handleAskPermissionToCommit(stage, partition,
          attemptNumber))
  }
}

```

---

根据代码清单 5-89，OutputCommitCoordinatorEndpoint 将接收两个消息。

- ❑ StopCoordinator：此消息将停止 OutputCommitCoordinatorEndpoint。
- ❑ AskPermissionToCommitOutput：此消息将通过 OutputCommitCoordinator 的 handleAskPermissionToCommit 方法（见代码清单 5-90）处理，进而确认客户端是否有权限将输出提交到 HDFS。

### 5.9.2 OutputCommitCoordinator 的实现

OutputCommitCoordinator 用于判定给定 Stage 的分区任务是否有权限将输出提交到 HDFS，并对同一分区任务的多次任务尝试（TaskAttempt）进行协调。OutputCommitCoordinator 中有以下属性。

- ❑ conf：即 SparkConf。
- ❑ isDriver：当前节点是否是 Driver。
- ❑ coordinatorRef：即 5.9.1 节介绍的 OutputCommitCoordinatorEndpoint 的 NettyRpcEndpointRef 引用。
- ❑ NO\_AUTHORIZED\_COMMITTER：值为 -1 的常量。
- ❑ authorizedCommittersByStage：缓存 Stage 的各个分区的任务尝试。类型为 scala.collection.mutable.Map[StageId, Array[TaskAttemptNumber]]。

了解了 OutputCommitCoordinator 的属性，来看看 OutputCommitCoordinator 实现的方法。

#### 1. handleAskPermissionToCommit

handleAskPermissionToCommit 方法（见代码清单 5-90）用于判断给定的任务尝试是否有权限将给定 Stage 的指定分区的数据提交到 HDFS。

## 代码清单5-90 handleAskPermissionToCommit的实现

---

```

private [scheduler] def handleAskPermissionToCommit(
    stage: StageId,
    partition: PartitionId,
    attemptNumber: TaskAttemptNumber): Boolean = synchronized {
  authorizedCommittersByStage.get(stage) match {
    case Some(authorizedCommitters) =>
      authorizedCommitters(partition) match {
        case NO_AUTHORIZED_COMMITTER =>
          logDebug(s"Authorizing attemptNumber=$attemptNumber to commit for
                    stage=$stage, " +
                  s"partition=$partition")
          authorizedCommitters(partition) = attemptNumber
          true
        case existingCommitter =>
          logDebug(s"Denying attemptNumber=$attemptNumber to commit for stage=
                    $stage, " +
                  s"partition=$partition; existingCommitter = $existingCommitter")
          false
      }
    case None =>
      logDebug(s"Stage $stage has completed, so not allowing attempt number
$attemptNumber of " +
              s"partition $partition to commit")
      false
  }
}

```

---

根据代码清单 5-90，handleAskPermissionToCommit 的处理步骤如下。

- 1) 从 authorizedCommittersByStage 缓存中找到给定 Stage 的指定分区的 TaskAttemptNumber (实际类型为 Int)。
- 2) 如果第 1) 步获取的 TaskAttemptNumber 等于 NO\_AUTHORIZED\_COMMITTER，则说明当前是首次提交给定 Stage 的指定分区的输出，因此按照第一提交者胜利 (first committer wins) 策略，给定 TaskAttemptNumber (即 attemptNumber) 有权限将给定 Stage 的指定分区的输出提交到 HDFS。为了告诉后来的任务尝试“已经有人捷足先登”，还需要将给定分区的索引与 attemptNumber 的关系保存到 TaskAttemptNumber 数组中。
- 3) 如果第 1) 步获取的 TaskAttemptNumber 不等于 NO\_AUTHORIZED\_COMMITTER，则说明之前已经有任务尝试将给定 Stage 的指定分区的输出提交到 HDFS，因此按照第一提交者胜利 (first committer wins) 策略，给定 TaskAttemptNumber (即 attemptNumber) 没有权限将给定 Stage 的指定分区的输出提交到 HDFS。

## 2. isEmpty

isEmpty 方法（见代码清单 5-91）用于判断 authorizedCommittersByStage 是否为空。

## 代码清单5-91 判断authorizedCommittersByStage是否为空

---

```
def isEmpty: Boolean = {
```

---

```
    authorizedCommitersByStage.isEmpty
}
```

---

### 3. canCommit

canCommit 方法（见代码清单 5-92）用于向 OutputCommitCoordinatorEndpoint 发送 AskPermissionToCommitOutput，并根据 OutputCommitCoordinatorEndpoint 的响应确认是否有权限将 Stage 的指定分区的输出提交到 HDFS 上。

代码清单5-92 canCommit的实现

---

```
def canCommit(
  stage: StageId,
  partition: PartitionId,
  attemptNumber: TaskAttemptNumber): Boolean = {
  val msg = AskPermissionToCommitOutput(stage, partition, attemptNumber)
  coordinatorRef match {
    case Some(endpointRef) =>
      endpointRef.askWithRetry[Boolean](msg) //询问是否有权限将stage的指定分区的输出提交到HDFS上
    case None =>
      logError(
        "canCommit called after coordinator was stopped (is SparkEnv shutdown in progress) ?")
      false
  }
}
```

---

### 4. stageStart

stageStart 方法（见代码清单 5-93）用于启动给定 Stage 的输出提交到 HDFS 的协调机制，其实质为创建给定 Stage 的对应 TaskAttemptNumber 数组，并将 TaskAttemptNumber 数组中的所有 TaskAttemptNumber 置为 NO\_AUTHORIZED\_COMMITTER。

代码清单5-93 stageStart的实现

---

```
private[scheduler] def stageStart(
  stage: StageId,
  maxPartitionId: Int): Unit = {
  val arr = new Array[TaskAttemptNumber](maxPartitionId + 1)
  java.util.Arrays.fill(arr, NO_AUTHORIZED_COMMITTER)
  synchronized {
    authorizedCommitersByStage(stage) = arr
  }
}
```

---

### 5. stageEnd

stageEnd 方法（见代码清单 5-94）用于停止给定 Stage 的输出提交到 HDFS 的协调机制，其实质为将给定 Stage 及对应的 TaskAttemptNumber 数组从 authorizedCommitersByStage 中删除。

代码清单5-94 stageEnd的实现

---

```
private[scheduler] def stageEnd(stage: StageId): Unit = synchronized {
    authorizedCommitersByStage.remove(stage)
}
```

---

## 6. taskCompleted

给定 Stage 的指定分区的任务执行完成后将调用 taskCompleted 方法（见代码清单 5-95）。

代码清单5-95 taskCompleted的实现

---

```
private[scheduler] def taskCompleted(
    stage: StageId,
    partition: PartitionId,
    attemptNumber: TaskAttemptNumber,
    reason: TaskEndReason): Unit = synchronized {
    val authorizedCommiters = authorizedCommitersByStage.getOrElse(stage, {
        logDebug(s"Ignoring task completion for completed stage")
        return
    })
    reason match {
        case Success =>
            // The task output has been committed successfully
        case denied: TaskCommitDenied =>
            logInfo(s"Task was denied committing, stage: $stage, partition: $partition, " +
                s"attempt: $attemptNumber")
        case otherReason =>
            if (authorizedCommiters(partition) == attemptNumber) {
                logDebug(s"Authorized committer (attemptNumber=$attemptNumber,
                    stage=$stage, " +
                    s"partition=$partition) failed; clearing lock")
                authorizedCommiters(partition) = NO_AUTHORIZED_COMMITTER
            }
    }
}
```

---

根据代码清单 5-95，有三种情况可视为任务完成。

- 任务执行成功：此时 reason 等于特质 TaskEndReason 的子类 Success。
- 任务提交被拒绝：此时 reason 等于特质 TaskEndReason 的子类 TaskCommitDenied。
- 其他原因：此时 reason 等于特质 TaskEndReason 的其他子类。针对这种原因，需要将给定 Stage 的对应 TaskAttemptNumber 数组中指定分区的值修改为 NO\_AUTHORIZED\_COMMITTER，以便于之后的任务尝试能够有权限提交。

## 7. stop

stop 方法（见代码清单 5-96）通过向 OutputCommitCoordinatorEndpoint 发送 StopCoordinator 消息以停止 OutputCommitCoordinatorEndpoint，然后清空 authorizedCommiters-ByStage。

代码清单5-96 停止OutputCommitCoordinator

---

```
def stop(): Unit = synchronized {
```

```

if (isDriver) {
    coordinatorRef.foreach(_ send StopCoordinator)
    coordinatorRef = None
    authorizedCommitteesByStage.clear()
}
}

```

### 5.9.3 OutputCommitCoordinator 的工作原理

经过对 OutputCommitCoordinatorEndpoint 和 OutputCommitCoordinator 的详细介绍，OutputCommitCoordinator 决定任务是否可以提交输出到 HDFS 的工作原理可以用图 5-15 来总结说明。

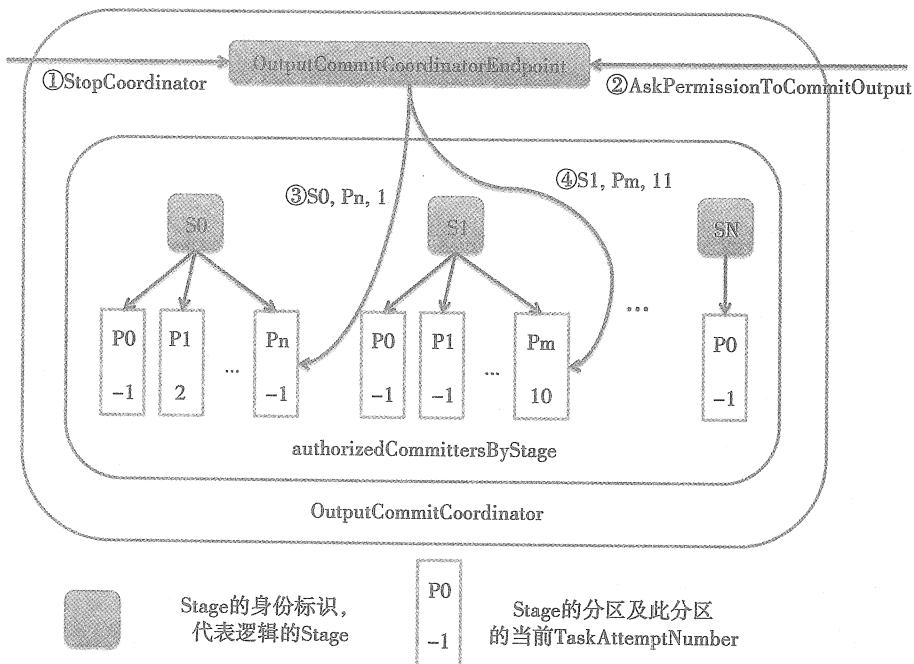


图 5-15 OutputCommitCoordinator 的工作原理

图 5-15 展现了 OutputCommitCoordinator 的逻辑结构和工作原理。authorizedCommitteesByStage 缓存了每一个 Stage 及其分区的内存结构，S0、S1 及 SN 代表不同的 Stage，P0、P1 则代表 Stage 的各个分区，Pn、Pm 说明每个 Stage 内的分区数量不同。图中每个序号的含义如下。

记号①：表示 OutputCommitCoordinatorEndpoint 收到 StopCoordinator 消息，OutputCommitCoordinatorEndpoint 将调用父类 RpcEndpoint 的 stop 方法（见代码清单 5-2），RpcEndpoint 的 stop 方法实际又调用了 NettyRpcEnv 的 stop 方法（见代码清单 5-45），停止 OutputCommitCoordinatorEndpoint 的工作。

记号②：表示 OutputCommitCoordinatorEndpoint 在接收到 AskPermissionToCommitOutput 消息后，调用 OutputCommitCoordinator 的 handleAskPermissionToCommit 方法判断给定的任务尝试是否有权限将给定 Stage 的指定分区的数据提交到 HDFS。

记号③：表示 AskPermissionToCommitOutput 消息携带的 Stage 为 S0，分区为 Pn，任务尝试号为 1，handleAskPermissionToCommit 方法执行时发现阶段 S0 的分区 Pn 还未有任务尝试占用（即值为 -1），则允许当前任务尝试将阶段 S0 的分区 Pn 的数据提交到 HDFS 并且将 Pn 的值设置为 1。

记号④：表示 AskPermissionToCommitOutput 消息携带的 Stage 为 S1，分区为 Pm，任务尝试号为 11，handleAskPermissionToCommit 方法执行时发现阶段 S1 的分区 Pm 已被其他任务尝试占用（占用此分区的任务尝试号为 10），则不允许当前任务尝试将阶段 S1 的分区 Pm 的数据提交到 HDFS。

## 5.10 创建 SparkEnv

当 SparkEnv 内的所有组件都实例化完毕，将正式构造 SparkEnv。除了本章介绍的各种组件外，SparkEnv 内部还有以下几个成员属性。

- isStopped：当前 SparkEnv 是否停止的状态。
- pythonWorkers：所有 python 实现的 Worker 的缓存。
- hadoopJobMetadata：HadoopRDD 进行任务切分时所需要的元数据的软引用。例如，HadoopFileRDD 将使用 hadoopJobMetadata 缓存 JobConf 和 InputFormat。
- driverTmpDir：如果当前 SparkEnv 处于 Driver 实例中，那么将创建 Driver 的临时目录。创建 Driver 的临时目录的代码如下。

```
if (isDriver) {
    val sparkFilesDir = Utils.createTempDir(Utils.getLocalDir(conf), "userFiles").
        getAbsolutePath
    envInstance.driverTmpDir = Some(sparkFilesDir)
}
```

上面代码中创建临时目录是由工具类 Utils 的 createTempDir 方法实现的，其具体实现请参阅附录 A。

## 5.11 小结

本章首先对构成 SparkEnv 的各个组件进行了展示，然后逐一讲解每个组件的创建。对有些组件的工作原理进行了详细的分析，这是因为这些组件的功能可以供其他组件使用，如安全管理器、RPC 环境、广播管理器、度量系统等。有些组件的功能跟任务的计算非常紧密，所以把这些组件的详细原理放在计算引擎的相关章节中，例如，shuffle 管理器、内

存管理器等。有些组件的功能都属于存储体系，这些组件的具体内容也统一放入存储体系的相关章节中，例如，块管理器、BlockManagerMaster、块传输服务等。

通过对 RPC 环境的讲解，读者将了解到服务端和客户端实现的细节，很多组件的通信都离不开 RPC 环境，例如，块管理器、map 任务输出跟踪器、输出提交协调器等。

广播管理器对广播对象的广播实际是通过对存储体系的读写完成的。广播的真正含义在于写入存储体系的备份数量，这个问题将在存储体系中详细介绍。很多组件中的对象都需要广播管理器进行广播，例如，Hadoop 配置、序列化后的 RDD、Job 及 ShuffleDependency 等。

度量系统可以将多种度量来源的度量数据输出到多种度量输出。还通过给 MetricsServlet 创建 ServletContextHandler，将度量系统与 Spark UI 结合起来使用。