一：Spark 1.6 RPC解析

1. Spark 1.6推出了以RpcEnv、RPCEndpoint、RPCEndpointRef为核心的新型架构下的RPC通信方式，就目前的实现而言，其底层依旧是Akka；
2. Akka是基于Actor的分布式消息通信系统，而在Spark 1.6中封装了Akka，提供更高层的Rpc实现，目的是移除对Akka的依赖，为扩展和自定义Rpc打下基础；
3. **名词解释：**

RpcEnv：是RPC的环境，所有的RPCEndpoint都需要注册到RpcEnv实例对象中。

RpcEndpointRef: 发送消息。

RpcEndpoint: 用于接收消息。

RpcEndFactory:创建RpcEnv

RpcCallContext：对异常进行处理。

二：RpcEnv解析

1. RpcEnv是RPC的环境（相当于Akka中的ActorSystem），所有的RPCEndpoint都需要注册到RpcEnv实例对象中（注册的时候会指定注册的名称，这样客户端就可以通过名称查询到RpcEndpoint的RpcEndpointRef(路由)引用，进而进行通信），在RpcEndpoint接受到消息后会调用receive方法进行处理；所有的RpcEndpoint都是属于RpcEnv。
2. RpcEndpoint如果接收到需要reply的消息的话就会交给自己的receiveAndReply来处理（回复时候是通过RpcCallContext中的relpy方法来回复发送者的）。

**override def** receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {  
 **case** *RegisterWorker*(  
 id, workerHost, workerPort, workerRef, cores, memory, workerUiPort, publicAddress) => {  
 logInfo("Registering worker %s:%d with %d cores, %s RAM".format(  
 workerHost, workerPort, cores, Utils.*megabytesToString*(memory)))  
 **if** (*state* == RecoveryState.*STANDBY*) {  
 context.reply(MasterInStandby)

1. 这里的reply源码如下：

**private**[spark] **trait** RpcCallContext {  
  
 */\*\*  
 \* Reply a message to the sender. If the sender is* [[*RpcEndpoint*]]*, its* [[*RpcEndpoint.receive*]]  
 *\* will be called.  
 \*/* **def** reply(response: Any): Unit

1. 如果不需要reply的话就交给receive方法来处理；
2. RpcEnvFactory是负责创建RpcEnv的，通过create方法创建RpcEnv实例对象，默认是用的Netty：

**private def** getRpcEnvFactory(conf: SparkConf): RpcEnvFactory = {  
 **val** rpcEnvNames = *Map*(  
 "akka" -> "org.apache.spark.rpc.akka.AkkaRpcEnvFactory",  
 "netty" -> "org.apache.spark.rpc.netty.NettyRpcEnvFactory")  
 **val** rpcEnvName = conf.get("spark.rpc", "netty")  
 **val** rpcEnvFactoryClassName = rpcEnvNames.getOrElse(rpcEnvName.toLowerCase, rpcEnvName)  
 Utils.*classForName*(rpcEnvFactoryClassName).newInstance().asInstanceOf[RpcEnvFactory]  
}

RpcEnvFactory的作用是创建RpcEnv。

*/\*\*  
 \* A factory class to create the* [[*RpcEnv*]]*. It must have an empty constructor so that it can be  
 \* created using Reflection.  
 \*/*

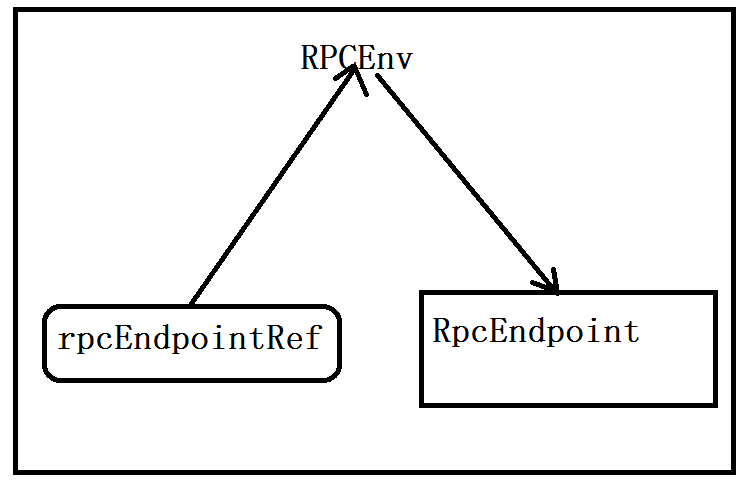
1. RpcEndpoint的生命周期：Constructor

**构造（constructor）-> 启动（onStart）、消息接收（ receive或者是receive）、停止（ onStop）**

RpcEnv具体解析

1. 所有的RpcEndpoint都需要向RpcEnv注册。

*/\*\*  
 \* An RPC environment.* [[*RpcEndpoint*]]*s need to register itself with a name to* [[*RpcEnv*]] *to  
 \* receives messages. Then* [[*RpcEnv*]] *will process messages sent from* [[*RpcEndpointRef*]] *or remote  
 \* nodes, and deliver them to corresponding* [[*RpcEndpoint*]]*s. For uncaught exceptions caught by  
 \** [[*RpcEnv*]]*,* [[*RpcEnv*]] *will use* [[*RpcCallContext.sendFailure*]] *to send exceptions back to the  
 \* sender, or logging them if no such sender or* `*NotSerializableException*`*.  
 \*  
 \** [[*RpcEnv*]] *also provides some methods to retrieve* [[*RpcEndpointRef*]]*s given name or uri.  
 \*/*



RpcEndpointRef会将消息发送给RpcEnv,RpcEnv将发过来的消息发送给RpcEndpoint去处理，如果接收到的是异常，则使用RpcCallContext处理。

**RpcEnv是如何接收RpcEndpointRef发过来消息的？**

RpcEnv中有注册RpcEndpoint的方法，也有address返回根地址的方法，

RpcCallContext：处理具体的异常，给发送者发送信息。

*/\*\*  
 \* A callback that* [[*RpcEndpoint*]] *can use it to send back a message or failure. It's thread-safe  
 \* and can be called in any thread.  
 \*/***private**[spark] **trait** RpcCallContext {  
  
 */\*\*  
 \* Reply a message to the sender. If the sender is* [[*RpcEndpoint*]]*, its* [[*RpcEndpoint.receive*]]  
 *\* will be called.  
 \*/* **def** reply(response: Any): Unit  
  
 */\*\*  
 \* Report a failure to the sender.  
 \*/* **def** sendFailure(e: Throwable): Unit  
  
 */\*\*  
 \* The sender of this message.  
 \*/* **def** senderAddress: RpcAddress  
}

详细看一下senderAddress:

*/\*\*  
 \* Address for an RPC environment, with hostname and port.  
 \*/***private**[spark] **case class** RpcAddress(host: String, port: Int) {  
  
 **def** hostPort: String = host + ":" + port  
  
 */\*\* Returns a string in the form of "spark://host:port". \*/* **def** toSparkURL: String = "spark://" + hostPort  
  
 **override def** toString: String = hostPort  
}  
  
  
**private**[spark] **object** RpcAddress { //伴身对象  
  
 */\*\* Return the* [[*RpcAddress*]] *represented by* `*uri*`*. \*/* **def** fromURIString(uri: String): RpcAddress = {  
 **val** uriObj = **new** java.net.URI(uri)  
 *RpcAddress*(uriObj.getHost, uriObj.getPort)  
 }  
  
 */\*\* Returns the* [[*RpcAddress*]] *encoded in the form of "spark://host:port" \*/* **def** fromSparkURL(sparkUrl: String): RpcAddress = {  
 **val** (host, port) = Utils.*extractHostPortFromSparkUrl*(sparkUrl)  
 *RpcAddress*(host, port)  
 }  
}

RpcTimeout内方法详解：

1. awaitResult方法等待产生结果并将结果返回，如果在指定的时间内没有返回结果，则抛出RpcTimeoutException异常。

*/\*\*  
 \* Wait for the completed result and return it. If the result is not available within this  
 \* timeout, throw a* [[*RpcTimeoutException*]] *to indicate which configuration controls the timeout.  
 \** ***@param awaitable*** *the* `*Awaitable*` *to be awaited  
 \** ***@throws*** *RpcTimeoutException if after waiting for the specified time* `*awaitable*`  
 *\* is still not ready  
 \*/***def** awaitResult[T](awaitable: Awaitable[T]): T = {  
 **try** {

//result在duration时间片内返回awaitable的执行结果。  
 Await.*result*(awaitable, duration)  
 } **catch** addMessageIfTimeout  
}

SparkEnv类中方法详解：

1. endpointRef返回RpcEndpointRef的引用。

*/\*\*  
 \* Return RpcEndpointRef of the registered* [[*RpcEndpoint*]]*. Will be used to implement  
 \** [[*RpcEndpoint.self*]]*. Return* `*null*` *if the corresponding* [[*RpcEndpointRef*]] *does not exist.  
 \*/***private**[rpc] **def** endpointRef(endpoint: RpcEndpoint): RpcEndpointRef

ThreadSafeRpcEndpoint:

线程安全，

*/\*\*  
 \* A trait that requires RpcEnv thread-safely sending messages to it.  
 \*  
 \* Thread-safety means processing of one message happens before processing of the next message by  
 \* the same* [[*ThreadSafeRpcEndpoint*]]*. In the other words, changes to internal fields of a  
 \** [[*ThreadSafeRpcEndpoint*]] *are visible when processing the next message, and fields in the  
 \** [[*ThreadSafeRpcEndpoint*]] *need not be volatile or equivalent.  
 \*  
 \* However, there is no guarantee that the same thread will be executing the same  
 \** [[*ThreadSafeRpcEndpoint*]] *for different messages.  
 \*/***private**[spark] **trait** ThreadSafeRpcEndpoint **extends** RpcEndpoint

Worker中的代码：

**private def** registerWithMaster(masterEndpoint: RpcEndpointRef): Unit = {  
 masterEndpoint.ask[RegisterWorkerResponse](*RegisterWorker*( //返回类型是：、、//RegisterWorkerResponse  
 *workerId*, *host*, *port*, self, cores, memory, *webUi*.boundPort, *publicAddress*))  
 .onComplete {  
 // This is a very fast action so we can use "ThreadUtils.sameThread"  
 **case** *Success*(msg) =>  
 Utils.*tryLogNonFatalError* {  
 handleRegisterResponse(msg)  
 }

RpcAddress

*/\*\*  
 \* Address for an RPC environment, with hostname and port.  
 \*/***private**[spark] **case class** RpcAddress(host: String, port: Int) {  
  
 **def** hostPort: String = host + ":" + port  
  
 */\*\* Returns a string in the form of "spark://host:port". \*/* **def** toSparkURL: String = "spark://" + hostPort  
  
 **override def** toString: String = hostPort  
}  
  
  
**private**[spark] **object** RpcAddress {  
  
 */\*\* Return the* [[*RpcAddress*]] *represented by* `*uri*`*. \*/* **def** fromURIString(uri: String): RpcAddress = {  
 **val** uriObj = **new** java.net.URI(uri)  
 *RpcAddress*(uriObj.getHost, uriObj.getPort)  
 }