

RPC 远程过程调用

目录

1、RPC 远程过程调用	2
1.1、RPC 概念	2
1.2、为什么要有 RPC	3
1.3、Java 中流行的 RPC 框架.....	4
2、Hadoop RPC	5
2.1、Hadoop RPC 概述.....	5
2.2、Hadoop RPC 涉及的技术.....	5
2.3、Hadoop RPC 对外提供的接口.....	6
2.4、使用 Hadoop RPC 构建应用的步骤.....	6
2.5、Hadoop RPC 应用案例.....	6
2.5.1、定义 RPC 协议	6
2.5.2、实现 RPC 协议	7
2.5.3、构建 RPC Server 并启动服务	7
2.5.4、构建 RPC Client 并发出请求	8
2.5.5、运行测试.....	9
2.5.6、通过 JPS 查看进程	10
2.6、Hadoop 的进程启动分析	10
3、Scala Actor.....	10
3.1、概念.....	10
3.2、传统的 Java 并发编程模型和 Scala Actor 区别	11
3.2.1、Java 中的并发编程模型	11
3.2.2、Scala 中的并发编程模型	11
3.2.3、Java 和 Scala 的编程模型对比.....	12
3.3、Actor 发送消息的方式.....	12
3.4、Scala Actor 实例.....	12
4、Akka Actor.....	13
4.1、Akka 概述.....	13
4.2、重要 API 介绍	14
4.2.1、ActorSystem	14
4.2.2、Actor.....	15
4.2.3、ActorSystem 和 Actor 对比.....	15
4.3、利用 Akka 构建 RPC 应用案例.....	15
4.3.1、需求.....	15
4.3.2、应用架构.....	16
4.2.3、具体实现.....	16
4.2.4、执行测试.....	18
5、Akka Actor 综合案例	19
5.1、需求.....	19
5.2、实现思路.....	19

5.3、实现代码.....	19
5.3.1、MyResourceManager 代码实现	19
5.3.2、MyNodeManager 代码实现	21
5.3.3、Message 代码实现	23
5.3.4、Constant 代码实现	23
5.4、执行测试.....	24

1、RPC 远程过程调用

1.1、RPC 概念

RPC (Remote Procedure Call) —远程过程调用，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。RPC 协议假定某些传输协议的存在，如 TCP 或 UDP，为通信程序之间携带信息数据。在 OSI 网络通信模型中，RPC 跨越了传输层和应用层。RPC 使得开发包括网络分布式多程序在内的应用程序更加容易。

第七层：应用层。定义了用于在网络中进行通信和传输数据的接口；
第六层：表示层。定义不同的系统中数据的传输格式，编码和解码规范等；
第五层：会话层。管理用户的会话，控制用户间逻辑连接的建立和中断；
第四层：传输层。管理着网络中的端到端的数据传输；
第三层：网络层。定义网络设备间如何传输数据；
第二层：链路层。将上面的网络层的数据包封装成数据帧，便于物理层传输；
第一层：物理层。这一层主要就是传输这些二进制数据。

实际应用过程中，**五层协议结构里面是没有表示层和会话层的。**应该说它们和应用层合并了。我们应该将重点放在应用层和传输层这两个层面。因为 HTTP 是应用层协议，而 TCP 是传输层协议。

RPC 采用客户机/服务器(C/S)模式：请求程序就是一个客户机，而服务提供程序就是一个服务器。首先，客户机调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息。在服务器端，进程保持睡眠状态直到调用信息到达为止。当一个调用信息到达，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息，最后，客户端调用进程接收答复信息，获得进程结果，然后调用执行继续进行。

通俗的说：RPC 是指远程过程调用，也就是说两台服务器 A,B，一个应用部署在 A 服务器，想要调用 B 服务器提供的函数和方法，由于不在一个内存空间，不能直接调用，需要通过网络来表达调用的语义和传达调用的数据。也可以理解成不同进程之间的服务调用

比如一个方法是如下定义：

```
Employee getEmployeeByName(String fullName)
```

并且被部署在 B 服务器上，那么 A 服务器想要调用这个服务，那么：

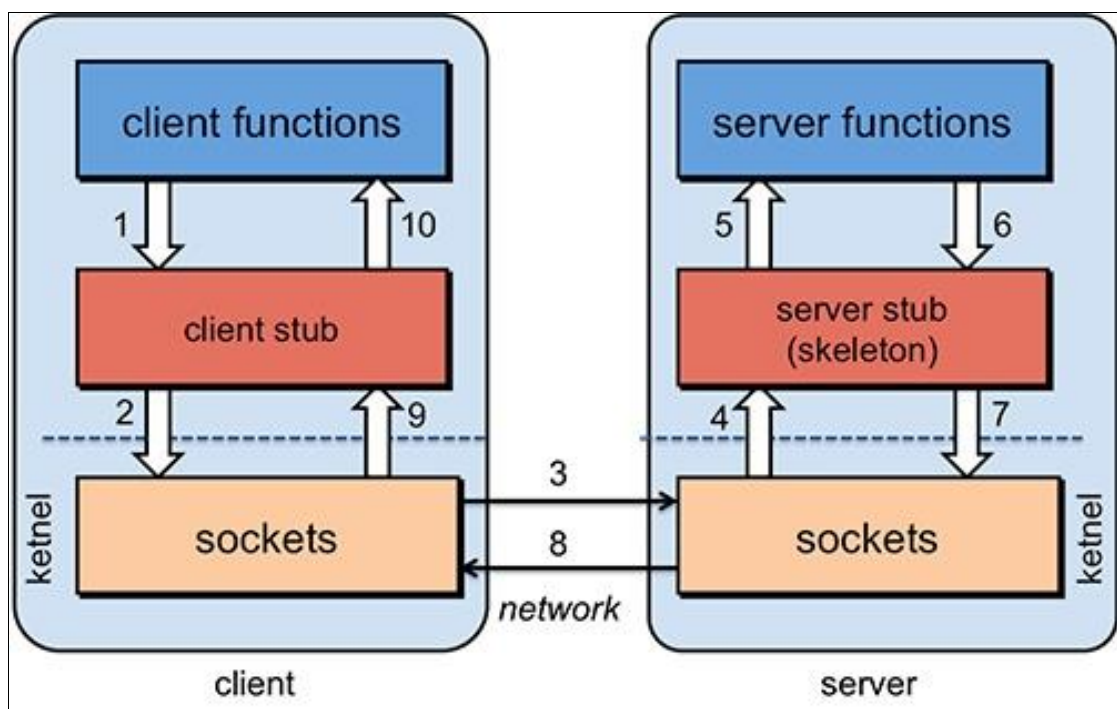
第一，要解决通讯的问题，主要是通过客户端和服务端之间建立 TCP 连接，远程过程调用的所有交换的数据都在这个连接里传输。连接可以是按需连接，调用结束后就断掉，也可以是长连接，多个远程过程调用共享同一个连接。

第二，要解决寻址的问题，也就是说，A 服务器上的应用怎么告诉底层的 RPC 框架，如何连接到 B 服务器（如主机或 IP 地址）以及特定的端口，方法的名称名称是什么，这样才能完成调用。

第三，当 A 服务器上的应用发起远程过程调用时，方法的参数需要通过底层的网络协议如 TCP 传递到 B 服务器，由于网络协议是基于二进制的，内存中的参数的值要序列化成二进制的形式，也就是序列化（Serialize）或编组（marshal），通过寻址和传输将序列化的二进制发送给 B 服务器。

第四，B 服务器收到请求后，需要对参数进行反序列化（序列化的逆操作），恢复为内存中的表达方式，然后找到对应的方法（寻址的一部分）进行本地调用，然后得到返回值。

第五，返回值还要发送回服务器 A 上的应用，也要经过序列化的方式发送，服务器 A 接到后，再反序列化，恢复为内存中的表达方式，交给 A 服务器上的应用



1.2、为什么要有 RPC

RPC 框架的职责是：让调用方感觉就像调用本地函数一样调用远端函数、让服务提供方感觉就像实现一个本地函数一样来实现服务，并且屏蔽编程语言的差异性。

RPC 的主要功能目标是让构建分布式计算（应用）更容易，在提供强大的远程调用能力时不损失本地调用的语义简洁性。为实现该目标，RPC 框架需提供一种透明调用机制让使用者不必显式的区分本地调用和远程调用

简单的讲，对于客户端 A 来说，调用远程服务器 B 上的服务，就跟调用 A 上的自身服务一样。因为在客户端 A 上来说，会生成一个服务器 B 的代理。

1.3、Java 中流行的 RPC 框架

1、RMI（远程方法调用）

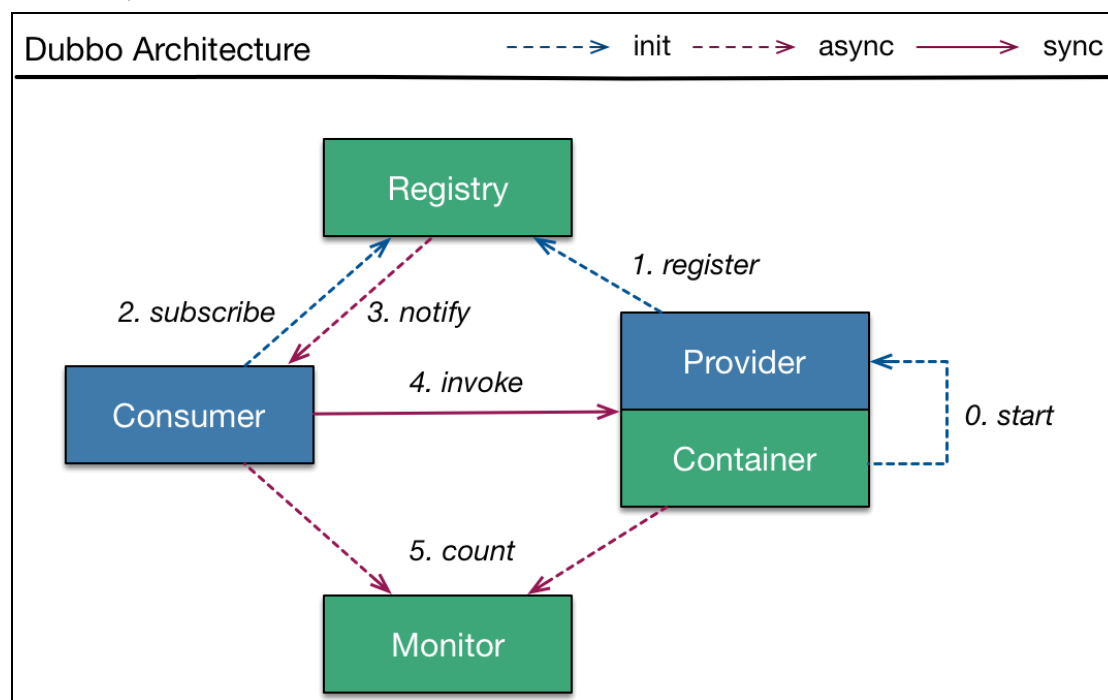
JAVA 自带的远程方法调用工具，不过有一定的局限性，毕竟是 JAVA 语言最开始时的设计，后来很多框架的原理都基于 RMI

2、Hessian（基于 HTTP 的远程方法调用）

基于 HTTP 协议传输，在性能方面还不够完美，负载均衡和失效转移依赖于应用的负载均衡器，Hessian 的使用则与 RMI 类似，区别在于淡化了 Registry 的角色，通过显示的地址调用，利用 HessianProxyFactory 根据配置的地址 create 一个代理对象，另外还要引入 Hessian 的 Jar 包。

3、Dubbo（阿里开源的基于 TCP 的 RPC 框架）

基于 Netty 的高性能 RPC 框架，是阿里巴巴开源的，大致原理如下：



阿里的 dubbo 现在已经贡献给 Apache 了，官网是：<http://dubbo.apache.org/>

Duboo 的中文官网：

<https://dubbo.gitbooks.io/dubbo-user-book/content/preface/background.html>

其他比较流行的 RPC 技术还有 FaceBook 的 thrift，谷歌的 grpc，基于 Netty 的 HadoopRPC，基于 actor 的 Akka 等

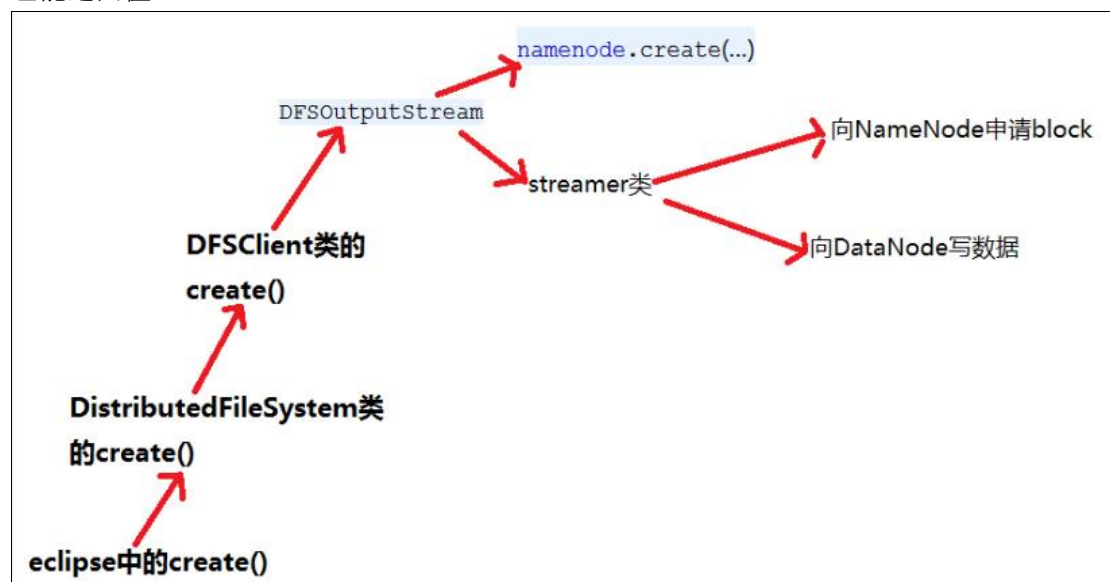
2、Hadoop RPC

2.1、Hadoop RPC 概述

同其他 RPC 框架一样，Hadoop RPC 分为四个部分：

- 1、序列化层：**Client 与 Server 端通信传递的信息采用了 Hadoop 里提供的序列化类或自定义的 Writable 类型；
- 2、函数调用层：**Hadoop RPC 通过动态代理以及 Java 反射实现函数调用；
- 3、网络传输层：**Hadoop RPC 采用了基于 TCP/IP 的 socket 机制；
- 4、服务器端框架层：**RPC Server 利用 Java NIO 以及采用了事件驱动的 I/O 模型，提高 RPC Server 的并发处理能力；

Hadoop RPC 在整个 Hadoop 中应用非常广泛，Client、DataNode、NameNode 之间的通讯全靠它了。例如：我们平时操作 HDFS 的时候，使用的是 `FileSystem` 类，它的内部有个 `DFSClient` 对象，这个对象负责与 `NameNode` 打交道。在运行时，`DFSClient` 在本地创建一个 `NameNode` 的代理，然后就操作这个代理，这个代理就会通过网络，远程调用到 `NameNode` 的方法，也能返回值。



2.2、Hadoop RPC 涉及的技术

- 1、代理：**动态代理可以提供对另一个对象的访问，同时隐藏实际对象的具体事实，代理对象对客户隐藏了实际对象。目前 Java 开发包中提供了对动态代理的支持，但现在只支持对接口的实现。

2、反射----动态加载类

3、序列化

4、非阻塞的异步 IO (NIO)

Java NIO 原理请参考阅读: <http://weixiaolu.iteye.com/blog/1479656>

2.3、Hadoop RPC 对外提供的接口

Hadoop RPC 对外主要提供了两种接口 (见类 org.apache.hadoop.ipc.RPC), 分别是:

1、public static <T> ProtocolProxy <T> getProxy/waitForProxy(...)

构造一个客户端代理对象 (该对象实现了某个协议), 用于向服务器发送 RPC 请求。

2、public static Server RPC.Builder (Configuration).build()

为某个协议 (实际上是 Java 接口) 实例构造一个服务器对象, 用于处理客户端发送的请求。

2.4、使用 Hadoop RPC 构建应用的步骤

1、定义 RPC 协议

RPC 协议是客户端和服务端之间的通信接口, 它定义了服务器端对外提供的服务接口。

2、实现 RPC 协议

Hadoop RPC 协议通常是一个 Java 接口, 用户需要实现该接口。

3、构造和启动 RPC SERVER

直接使用静态类 Builder 构造一个 RPC Server, 并调用函数 start() 启动该 Server。

4、构造 RPC Client 并发送请求

使用静态方法 getProxy 构造客户端代理对象, 直接通过代理对象调用远程端的方法。

2.5、Hadoop RPC 应用案例

2.5.1、定义 RPC 协议

```
package com.mazh.rpc;

import org.apache.hadoop.ipc.VersionedProtocol;

/**
```

```
* RPC 协议: 用来定义服务
* 要实现 VersionedProtocol 这个接口: 不同版本的 Server 和 Client 之前是不能进行通信的
*/
public interface BussinessProtocol {
    void mkdir(String path);
    void hello(String name);

    long versionID = 345043000L;
}
```

2.5.2、实现 RPC 协议

```
package com.mazh.rpc;

/**
 * 实现 协议
 */
public class BusinessIMPL implements BussinessProtocol {
    @Override
    public void mkdir(String path) {
        System.out.println("成功创建了文件夹 : " + path);
    }

    @Override
    public void hello(String name) {
        System.out.println("hello : " + name);
    }
}
```

2.5.3、构建 RPC Server 并启动服务

```
package com.mazh.rpc;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.ipc.RPC;

import java.io.IOException;

/**
 * 模拟 Hadoop 构建一个 RPC 服务端
 */
public class MyNamenode {
```

```
public static void main(String[] args) {

    try {
        RPC.Server server = new RPC.Builder(new Configuration())
            .setProtocol(BussinessProtocol.class)
            .setInstance(new BusinessIMPL())
            .setBindAddress("localhost")
            .setPort(6789)
            .build();

        server.start();

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

2.5.4、构建 RPC Client 并发出请求

```
package com.mazh.rpc;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.ipc.RPC;

import java.io.IOException;
import java.net.InetSocketAddress;

/**
 * 构建 RPC 客户端
 */
public class MyDataNode {
    public static void main(String[] args) {

        /**
         * Class<T> protocol,
         * Long clientVersion,
         * InetSocketAddress addr,
         * Configuration conf
         */
        try {

            /**
```



```

        * 获取了服务端中暴露了的服务协议的一个代理。
        * 客户端通过这个代理可以调用服务端的方法进行逻辑处理
    */
    BussinessProtocol proxy = RPC.getProxy(BussinessProtocol.class,
        BussinessProtocol.versionID,
        new InetSocketAddress("localhost", 6789),
        new Configuration());

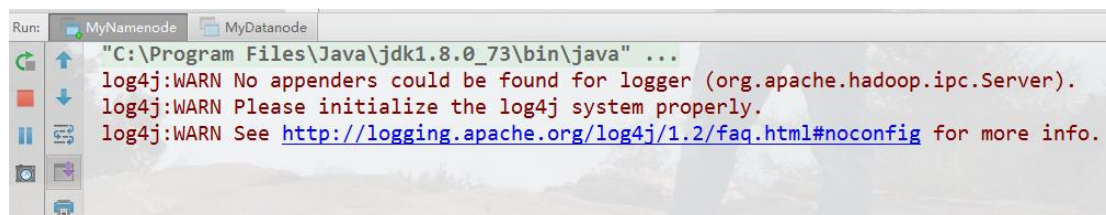
    /**
     * 在客户端调用了服务端的代码执行
     * 真正的代码执行是在服务端的
     */
    proxy.hello("hadoop");
    proxy.mkdir("/home/hadoop/apps");

    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

2.5.5、运行测试

1、启动服务端：运行 MyNameNode:

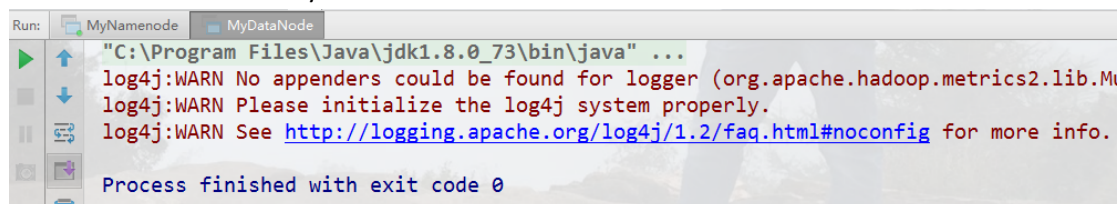


```

Run: MyNameNode
"C:\Program Files\Java\jdk1.8.0_73\bin\java" ...
log4j:WARN No appenders could be found for logger (org.apache.hadoop.ipc.Server).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.

```

2、启动客户端：运行 MyDataNode



```

Run: MyNameNode MyDataNode
"C:\Program Files\Java\jdk1.8.0_73\bin\java" ...
log4j:WARN No appenders could be found for logger (org.apache.hadoop.metrics2.lib.M
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.

Process finished with exit code 0

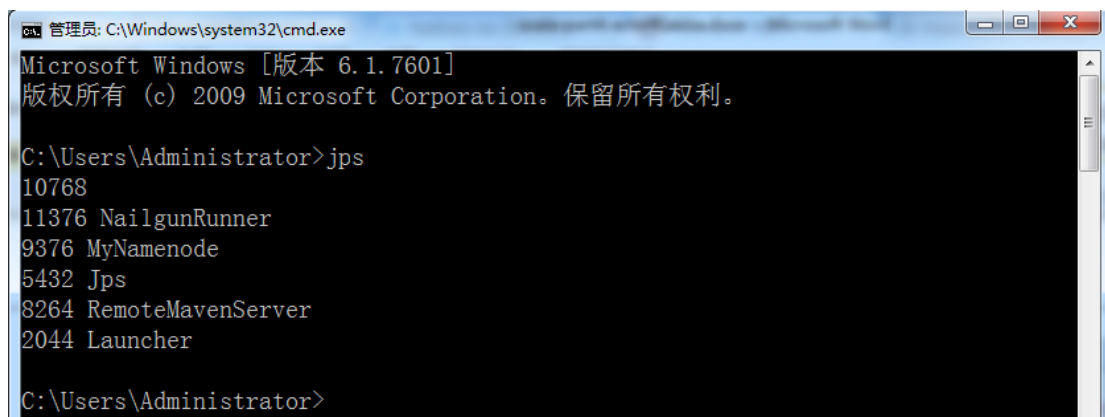
```

3、再次查看服务端:



```
Run: MyNamenode MyDataNode
"C:\Program Files\Java\jdk1.8.0_73\bin\java" ...
log4j:WARN No appenders could be found for logger (org.apache.hadoop.ipc.Server).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
hello : hadoop
成功创建了文件夹 : /home/hadoop/apps
```

2.5.6、通过 JPS 查看进程



```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation. 保留所有权利。

C:\Users\Administrator>jps
10768
11376 NailgunRunner
9376 MyNamenode
5432 Jps
8264 RemoteMavenServer
2044 Launcher

C:\Users\Administrator>
```

2.6、Hadoop 的进程启动分析

- 1、Namenode
- 2、Datanode
- 3、SecondaryNamenode
- 4、ZKFC
- 5、JournalNode

3、Scala Actor

3.1、概念

Scala 中的 Actor 能够实现并行编程的强大功能，它是基于事件模型的并发机制，Scala 是运用消息（message）的发送、接收来实现多线程的。使用 Scala 能够更容易地实现多线程应用的开发。

一个 Actor 是一个容器，它包含状态，行为，信箱，子 Actor 和监管策略，所有这些包含在一个 ActorReference (Actor 引用) 里。一个 Actor 需要与外界隔离才能从 Actor 模型中获益，所以 Actor 是以 Actor 引用的形式展现给外界的。

3.2、传统的 Java 并发编程模型和 Scala Actor 区别

3.2.1、Java 中的并发编程模型

1、Java 中的并发编程基本上满足了事件之间相互独立，但是事件能够同时发生的场景的需要。

2、Java 中的并发编程是基于共享数据和加锁的一种机制，即会有一个共享的数据，然后有若干个线程去访问这个共享的数据(主要是对这个共享的数据进行修改)，同时 Java 利用加锁的机制(即 `synchronized`)来确保同一时间只有一个线程对我们的共享数据进行访问，进而保证共享数据的一致性。

3、Java 中的并发编程存在资源争夺和死锁等多种问题，因此程序越大问题越麻烦。

3.2.2、Scala 中的并发编程模型

1、Scala 中的并发编程思想与 Java 中的并发编程思想完全不一样，Scala 中的 Actor 是一种不共享数据，依赖于消息传递的一种并发编程模式，避免了死锁、资源争夺等情况。在具体实现的过程中，Scala 中的 Actor 会不断的循环自己的邮箱，并通过 `receive` 偏函数进行消息的模式匹配并进行相应的处理。

2、如果 Actor A 和 Actor B 要相互沟通的话，首先 A 要给 B 传递一个消息，B 会有一个收件箱，然后 B 会不断的循环自己的收件箱，若看见 A 发过来的消息，B 就会解析 A 的消息并执行，处理完之后就有可能将处理的结果通过邮件的方式发送给 A。

3.2.3、Java 和 Scala 的编程模型对比

Java内置线程模型	Scala actor模型
“共享数据-锁”模型 (share data and lock)	share nothing
每个object有一个monitor，监视多线程对共享数据的访问	不共享数据，actor之间通过message通讯
加锁的代码段用synchronized标识	
死锁问题	
每个线程内部是顺序执行的	每个actor内部是顺序执行的

对于 Java，我们都知道它的多线程实现需要对共享资源（变量、对象等）使用 **synchronized** 关键字进行代码块同步、对象锁互斥等等。而且，常常一大块的 try...catch 语句块中加上 wait 方法、notify 方法、notifyAll 方法是让人很头疼的。原因就在于 Java 中多数使用的是可变速态的对象资源，对这些资源进行共享来实现多线程编程的话，控制好资源竞争与防止对象状态被意外修改是非常重要的，而对象状态的不变性也是较难以保证的。而在 Scala 中，我们可以**通过复制不可变速态的资源（即对象，Scala 中一切都是对象，连函数、方法也是）的一个副本，再基于 Actor 的消息发送、接收机制进行并行编程**

3.3、Actor 发送消息的方式

!	发送异步消息，没有返回值。
!?	发送同步消息，等待返回值。
!!	发送异步消息，返回值是 Future[Any]。

3.4、Scala Actor 实例

```
package com.mazh.scala.actor
//注意导包是 scala.actors.Actor
import scala.actors.Actor

object MyActor1 extends Actor{
  //重新 act 方法
  def act(){
    for(i <- 1 to 10){
      println("actor-1 " + i)
    }
  }
}
```

```
Thread.sleep(2000)
}
}
}

object MyActor2 extends Actor{
  //重新 act 方法
  def act(){
    for(i <- 1 to 10){
      println("actor-2 " + i)
      Thread.sleep(2000)
    }
  }
}

object ActorTest extends App{
  //启动 Actor
  MyActor1.start()
  MyActor2.start()
}
```

说明：上面分别调用了两个单例对象的 `start()` 方法，他们的 `act()` 方法会被执行，相同与在 Java 中开启了两个线程，线程的 `run()` 方法会被执行

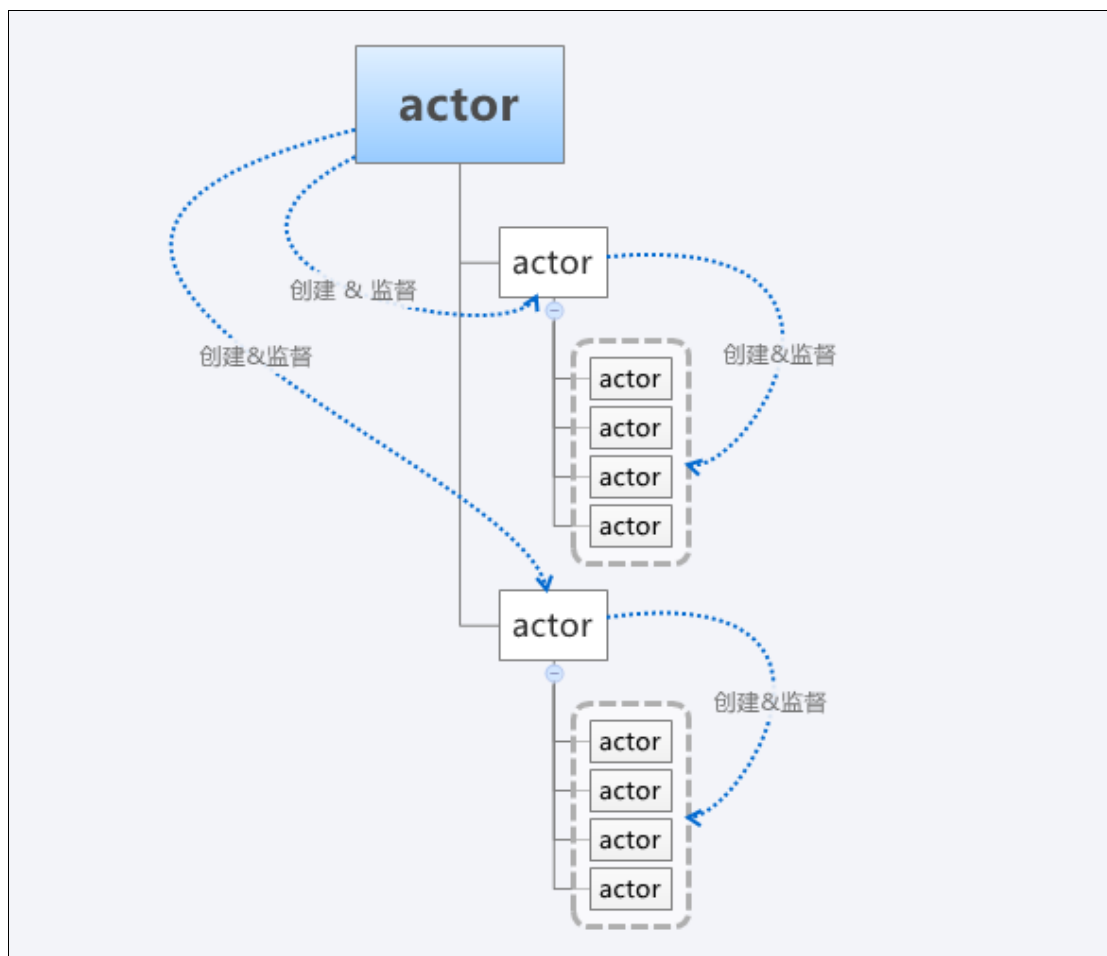
注意：这两个 Actor 是并行执行的，`act()` 方法中的 `for` 循环执行完成后 actor 程序就退出了

4、Akka Actor

4.1、Akka 概述

Akka 基于 Actor 模型，提供了一个用于构建可扩展的（Scalable）、弹性的（Resilient）、快速响应的（Responsive）应用程序的平台。

Actor 模型：在计算机科学领域，Actor 模型是一个并行计算（Concurrent Computation）模型，它把 actor 作为并行计算的基本元素来对待：为响应一个接收到的消息，一个 actor 能够自己做出一些决策，如创建更多的 actor，或发送更多的消息，或者确定如何去响应接收到的下一个消息。



Actor 是 Akka 中最核心的概念，它是一个封装了状态和行为的对象，Actor 之间可以通过交换消息的方式进行通信，每个 Actor 都有自己的收件箱（Mailbox）。通过 Actor 能够简化锁及线程管理，可以非常容易地开发出正确地并发程序和并行系统。

Actor 具有如下特性：

- 1、提供了一种高级抽象，能够简化在并发（Concurrency）/并行（Parallelism）应用场景下的编程开发
- 2、提供了异步非阻塞的、高性能的事件驱动编程模型
- 3、超级轻量级事件处理（每 GB 堆内存几百万 Actor）

4.2、重要 API 介绍

4.2.1、ActorSystem

在 Akka 中，ActorSystem 是一个重量级的结构，他需要分配多个线程，所以在实际应用中，ActorSystem 通常是一个单例对象，我们可以使用这个 ActorSystem 的 actorOf 方法创建很多 Actor。

4.2.2、Actor

在 Akka 中，Actor 负责通信，在 Actor 中有一些重要的生命周期方法。

1、preStart()方法：该方法在 Actor 对象构造方法执行后执行，整个 Actor 生命周期中仅执行一次。

3、receive()方法：该方法在 Actor 的 preStart 方法执行完成后执行，用于接收消息，会被反复执行。

4.2.3、ActorSystem 和 Actor 对比

Actor:

就是用来做消息传递的

用来接收和发送消息的，一个 Actor 就相当于是一个老师或者是学生。

如果我们想要多个老师，或者学生，就需要创建多个 Actor 实例。

ActorSystem:

用来创建和管理 Actor，并且还需要监控 Actor。ActorSystem 是单例的（object）

在同一个进程里面，只需要一个 ActorSystem 就可以了

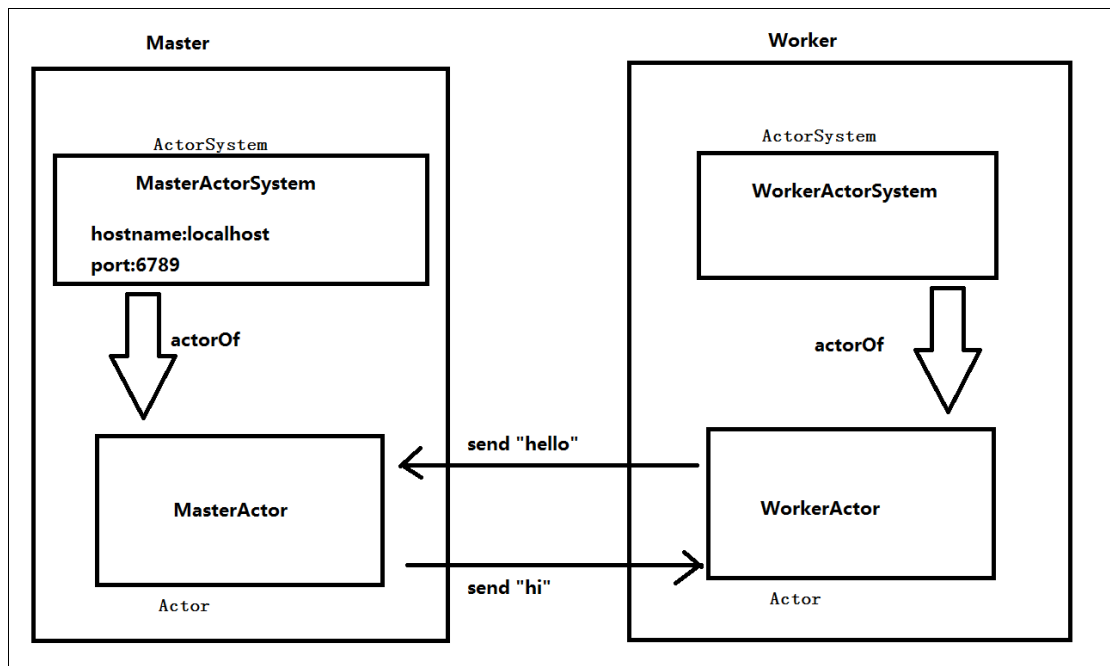
4.3、利用 Akka 构建 RPC 应用案例

4.3.1、需求

目前大多数的分布式架构底层通信都是通过 RPC 实现的，RPC 框架非常多，比如前我们学过的 Hadoop 项目的 RPC 通信框架，但是 Hadoop 在设计之初就是为了运行长达数小时的批量而设计的，在某些极端的情况下，任务提交的延迟很高，所有 Hadoop 的 RPC 显得有些笨重。

Spark 的 RPC 是通过 Akka 类库实现的，Akka 用 Scala 语言开发，基于 Actor 并发模型实现，Akka 具有高可靠、高性能、可扩展等特点，使用 Akka 可以轻松实现分布式 RPC 功能。

4.3.2、应用架构



4.2.3、具体实现

Master 代码实现:

```

package com.mazh.rpc

import akka.actor.{Actor, ActorSystem, Props}
import com.typesafe.config.{Config, ConfigFactory}

/**
 * 作者: 马中华
 * 主页: https://blog.csdn.net/zhongqi2513
 *
 * 描述: akka.tcp://MasterActorSystem@localhost:6789
 */
class Master extends Actor{
  def doHello(): Unit ={
    println("我是 Master, 我接收到了 Worker 的 hello 的消息");
  }
  /**
   * 其实就是一个死循环: 接收消息
   * while(true)
   */
  override def receive: Receive = {

```



```
case "hello" =>{
    doHello()
    //sender() 谁发送过来消息这个就是谁
    //sender() ! "hi" 给 sender() 发送一个 hi 的消息
    sender() ! "hi"
}
}
}

object Master {

    def main(args: Array[String]): Unit = {
        val str=
            """
            |akka.actor.provider = "akka.remote.RemoteActorRefProvider"
            |akka.remote.netty.tcp.hostname = localhost
            |akka.remote.netty.tcp.port = 6790
            """.stripMargin

        val conf: Config = ConfigFactory.parseString(str)
        // def apply(name: String, config: Config)
        val actorSystem = ActorSystem("MasterActorSystem", conf)
        //创建并启动 actor  def actorOf(props: Props, name: String): ActorRef
        //new Master() 会导致主构造函数会运行!!
        actorSystem.actorOf(Props(new Master()), "MasterActor")
    }
}
```

Worker 代码实现:

```
package com.mazh.rpc

import akka.actor.{Actor, ActorSystem, Props}
import com.typesafe.config.ConfigFactory

/**
 * 作者: 马中华
 * 主页: https://blog.csdn.net/zhongqi2513
 */
class Worker extends Actor{//生命周期
    def doHi(): Unit ={
        println("我是Worker, 我接收到了 Master 的 hi 的消息");
    }
    //如果 actor 一执行首先运行的是这个方法, 只运行一次。
    override def preStart(): Unit = {
```

```
//实现的是给 Master 发送消息 地址
val workerActor =
context.actorSelection("akka.tcp://MasterActorSystem@localhost:6790/user/MasterActor")
workerActor ! "hello"
}

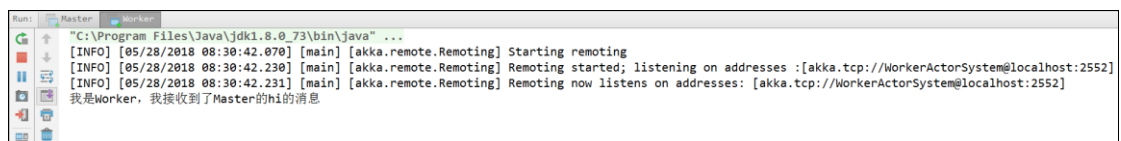
override def receive: Receive = {
  case "hi" => {
    doHi()
  }
}
}
}
object Worker {
  def main(args: Array[String]): Unit = {

    val str=
      """
      |akka.actor.provider = "akka.remote.RemoteActorRefProvider"
      |akka.remote.netty.tcp.hostname = localhost
      |""".stripMargin

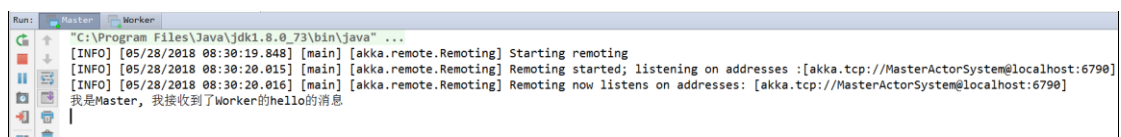
    val conf = ConfigFactory.parseString(str)
    val actorSystem = ActorSystem("WorkerActorSystem", conf)
    actorSystem.actorOf(Props(new Worker()), "WorkerActor")
  }
}
```

4.2.4、执行测试

先启动 Master，再启动 Worker



```
Run: Master
"C:\Program Files\Java\jdk1.8.0_73\bin\java" ...
[INFO] [05/28/2018 08:30:42.070] [main] [akka.remote.Remoting] Starting remoting
[INFO] [05/28/2018 08:30:42.230] [main] [akka.remote.Remoting] Remoting started; listening on addresses :[akka.tcp://WorkerActorSystem@localhost:2552]
[INFO] [05/28/2018 08:30:42.231] [main] [akka.remote.Remoting] Remoting now listens on addresses: [akka.tcp://WorkerActorSystem@localhost:2552]
我是Worker, 我接收到了Master的hi的消息
```



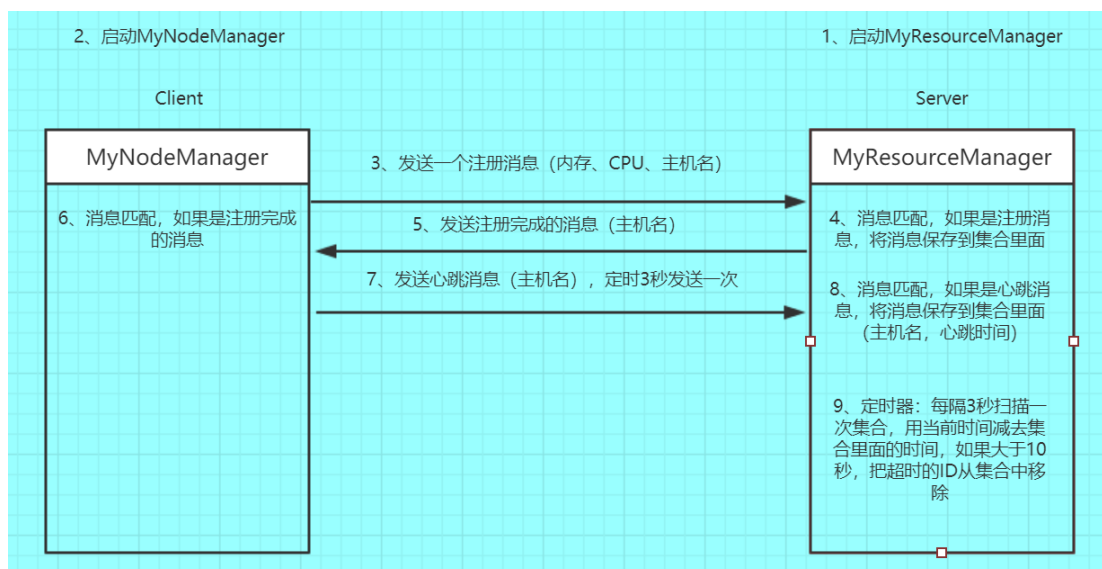
```
Run: Worker
"C:\Program Files\Java\jdk1.8.0_73\bin\java" ...
[INFO] [05/28/2018 08:30:19.848] [main] [akka.remote.Remoting] Starting remoting
[INFO] [05/28/2018 08:30:20.015] [main] [akka.remote.Remoting] Remoting started; listening on addresses :[akka.tcp://MasterActorSystem@localhost:6790]
[INFO] [05/28/2018 08:30:20.016] [main] [akka.remote.Remoting] Remoting now listens on addresses: [akka.tcp://MasterActorSystem@localhost:6790]
我是Master, 我接收到了Worker的hello的消息
```

5、Akka Actor 综合案例

5.1、需求

模拟实现 YARN（具备注册和心跳的功能）

5.2、实现思路



5.3、实现代码

5.3.1、MyResourcemanager 代码实现

```

package com.mazh.rpc.yarn

import akka.actor.{Actor, ActorSystem, Props}
import com.typesafe.config.ConfigFactory

import scala.collection.mutable

/**
 * 作者： 马中华
 * 主页： https://blog.csdn.net/zhongqi2513
 */
class MyResourceManager(var hostname:String,var port:Int) extends Actor{

```

```
private var id2nodemanagerinfo = new mutable.HashMap[String,NodeManagerInfo]()
private var nodemanagerInfoes = new mutable.HashSet[NodeManagerInfo]()

override def preStart(): Unit = {
  import scala.concurrent.duration._
  import context.dispatcher
  context.system.scheduler.schedule(0 millis,5000 millis,self,CheckTimeOut)
}

override def receive: Receive = {
  case RegisterNodeManager(nodemanagerid,memory,cpu) => {
    val nodeManagerInfo = new NodeManagerInfo(nodemanagerid,memory,cpu)

    id2nodemanagerinfo.put(nodemanagerid,nodeManagerInfo)

    nodemanagerInfoes += nodeManagerInfo

    //把信息存到zookeeper
    sender() ! RegisteredNodeManager(hostname+ ":" +port)
  }

  case Heartbeat(nodemanagerid) =>{
    val currentTime = System.currentTimeMillis()
    val nodeManagerInfo = id2nodemanagerinfo(nodemanagerid)
    nodeManagerInfo.LastHeartBeatTime=currentTime

    id2nodemanagerinfo(nodemanagerid)=nodeManagerInfo
    nodemanagerInfoes += nodeManagerInfo
  }

  case CheckTimeOut =>{
    val currentTime = System.currentTimeMillis()

    nodemanagerInfoes.filter(nm => currentTime - nm.LastHeartBeatTime > 15000)
      .foreach( deadnm =>{
        nodemanagerInfoes -= deadnm
        id2nodemanagerinfo.remove(deadnm.nodemanagerid)
      })
    println("当前注册成功的节点数"+nodemanagerInfoes.size);
  }
}
}
```

```
object MyResourceManager {
  def main(args: Array[String]): Unit = {
    val RESOURCEMANAGER_HOSTNAME=args(0) //解析的配置的日志
    val RESOURCEMANAGER_PORT=args(1).toInt
    val str=
      s"""
        |akka.actor.provider = "akka.remote.RemoteActorRefProvider"
        |akka.remote.netty.tcp.hostname =${RESOURCEMANAGER_HOSTNAME}
        |akka.remote.netty.tcp.port=${RESOURCEMANAGER_PORT}
        |""".stripMargin

    val conf = ConfigFactory.parseString(str)
    val actorSystem = ActorSystem(Constant.RMAS,conf)
    actorSystem.actorOf(Props(new
MyResourceManager(RESOURCEMANAGER_HOSTNAME,RESOURCEMANAGER_PORT) ),Constant.RMA)
  }
}
```

5.3.2、MyNodeManager 代码实现

```
package com.mazh.rpc.yarn

import java.util.UUID

import akka.actor.{Actor, ActorSelection, ActorSystem, Props}
import com.typesafe.config.ConfigFactory
import sun.plugin2.message.HeartbeatMessage

/**
 * 作者: 马中华
 * 主页: https://blog.csdn.net/zhongqi2513
 */
class MyNodeManager(val resourcemanagerhostname:String,
                    val resourcemanagerport:Int,val memory:Int,val cpu:Int) extends
Actor{
  var nodemanagerid:String=_
  var rmRef:ActorSelection=_
  override def preStart(): Unit = {
    rmRef=
context.actorSelection(s"akka.tcp://${Constant.RMAS}@${resourcemanagerhostname}:${r
esourcemanagerport}/user/${Constant.RMA}")
  }
```

```
//al nodemanagerid:String,val memory:Int,val cpu:Int
nodemanagerid = UUID.randomUUID().toString
rmRef ! RegisterNodeManager(nodemanagerid,memory,cpu)
}

override def receive: Receive = {
  case RegisteredNodeManager(masterURL) => {
    println(masterURL);

    // sender() ! Heartbeat(nodemanagerid)

    /**
     * initialDelay: FiniteDuration, 多久以后开始执行
     * interval:      FiniteDuration, 每隔多长时间执行一次
     * receiver:      ActorRef, 给谁发送这个消息
     * message:       Any 发送的消息是啥
     */
    import scala.concurrent.duration._

    //context.system.scheduler.schedule(0 millis,1000
    millis,sender(),Heartbeat(nodemanagerid))

    println(Thread.currentThread().getId)
    import context.dispatcher
    context.system.scheduler.schedule(0 millis,4000 millis,self,SendMessage)
  }

  case SendMessage =>{
    // 向主节点发送心跳信息
    rmRef ! Heartbeat(nodemanagerid)

    println(Thread.currentThread().getId)
  }
}

object MyNodeManager{
  def main(args: Array[String]): Unit = {
    val HOSTNAME=args(0)
    val RM_HOSTNAME =args(1)
    val RM_PORT=args(2).toInt
    val NODEMANAGER_MEMORY = args(3).toInt
    val NODEMANAGER_CORE=args(4).toInt
    var NODEMANAGER_PORT=args(5).toInt
  }
}
```

```
val str=
  s"""
    |akka.actor.provider = "akka.remote.RemoteActorRefProvider"
    |akka.remote.netty.tcp.hostname = ${HOSTNAME}
    |akka.remote.netty.tcp.port=${NODEMANAGER_PORT}
  """`.stripMargin
val conf = ConfigFactory.parseString(str)
val actorSystem = ActorSystem(Constant.NMAS,conf)
actorSystem.actorOf(Props(new
MyNodeManager(RM_HOSTNAME,RM_PORT,NODEMANAGER_MEMORY,NODEMANAGER_CORE) ),Constant.N
MA)
}
}
```

5.3.3、Message 代码实现

```
package com.mazh.rpc.yarn

//注册消息  nodemanager -> resourcemanager
case class RegisterNodeManager(val nodemanagerid:String,val memory:Int,val cpu:Int)
//注册完成消息 resourcemanager -> nodemanager
case class RegisteredNodeManager(val resourcemanagerhostname:String)
//心跳消息  nodemanager -> resourcemanager
case class Heartbeat(val nodemanagerid:String)

class NodeManagerInfo(val nodemanagerid:String,val memory:Int,val cpu:Int){
  var lastHeartBeatTime:Long=_
}
// 单例
case object SendMessage
case object CheckTimeOut
```

5.3.4、Constant 代码实现

```
package com.mazh.rpc.yarn

object Constant {
  val RMAS="MyResourceManagerActorSystem"
  val RMA="MyResourceManagerActor"
  val NMAS="MyNodeManagerActorSystem"
  val NMA="MyNodeManagerActor"
```

```
}
```

5.4、执行测试