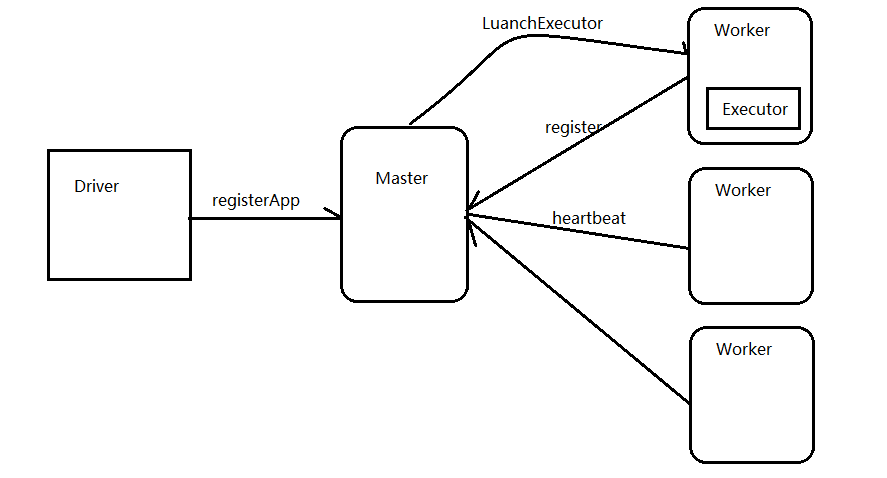
Scala编程实战

# 课程目标

## 目标：熟练使用Scala编写程序



# 项目概述

## 需求

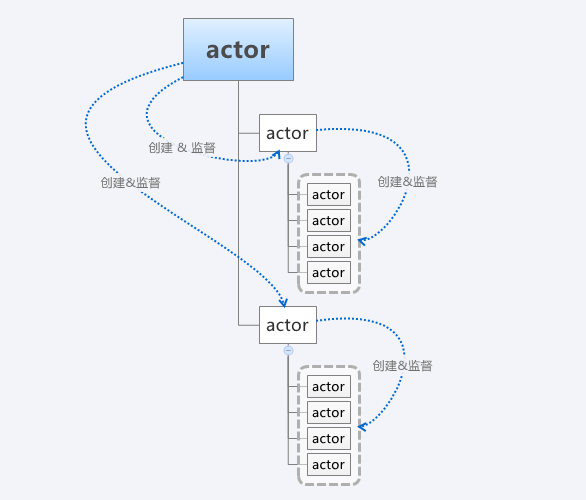
目前大多数的分布式架构底层通信都是通过RPC实现的，RPC框架非常多，比如前我们学过的Hadoop项目的RPC通信框架，但是Hadoop在设计之初就是为了运行长达数小时的批量而设计的，在某些极端的情况下，任务提交的延迟很高，所有Hadoop的RPC显得有些笨重。

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。

## Akka简介

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：

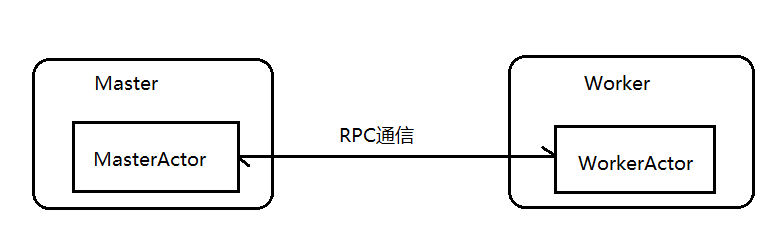
1.提供了一种高级抽象，能够简化在并发（Concurrency）/并行（Parallelism）应用场景下的编程开发

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）

# 项目实现

## 架构图



## 重要类介绍

### 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创建很多Actor。

### 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。
2. 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。

## Master类

|  |
| --- |
| **package** cn.itcast.spark **import** scala.concurrent.duration.\_ **import** akka.actor.{Props, ActorSystem, Actor} **import** akka.actor.Actor.Receive **import** com.typesafe.config.ConfigFactory  **import** scala.collection.mutable  */\*\*  \* Master为整个集群中的主节点  \* Master继承了Actor  \*/* **class** Master **extends** Actor{   *//保存WorkerID和Work信息的map* **val** *idToWorker* = **new** mutable.HashMap[String, WorkerInfo]  *//保存所有Worker信息的Set* **val** *workers* = **new** mutable.HashSet[WorkerInfo]  *//Worker超时时间* **val** *WORKER\_TIMEOUT* = 10 \* 1000  *//重新receive方法   //导入隐式转换，用于启动定时器* **import** *context*.dispatcher   *//构造方法执行完执行一次* **override def** preStart(): Unit = {  *//启动定时器，定时执行  context*.system.scheduler.schedule(0 millis, *WORKER\_TIMEOUT* millis, *self*, CheckOfTimeOutWorker)  }   *//该方法会被反复执行，用于接收消息，通过case class模式匹配接收消息* **override def** receive: Receive = {  *//Worker向Master发送的注册消息* **case** *RegisterWorker*(id, workerHost, memory, cores) => {  **if**(!*idToWorker*.contains(id)) {  **val** worker = **new** WorkerInfo(id, workerHost, memory, cores)  *workers*.add(worker)  *idToWorker*(id) = worker  sender ! *RegisteredWorker*(**"192.168.10.1"**)  }  }   *//Worker向Master发送的心跳消息* **case** *HeartBeat*(workerId) => {  **val** workerInfo = *idToWorker*(workerId)  workerInfo.*lastHeartbeat* = System.*currentTimeMillis*()  }   *//Master自己向自己发送的定期检查超时Worker的消息* **case** CheckOfTimeOutWorker => {  **val** currentTime = System.*currentTimeMillis*()  **val** toRemove = *workers*.filter(w => currentTime - w.*lastHeartbeat* > *WORKER\_TIMEOUT*).toArray  **for**(worker <- toRemove){  *workers* -= worker  *idToWorker*.remove(worker.id)  }  *println*(**"worker size: "** + *workers*.size)  }  } }  **object** Master {  *//程序执行入口* **def** main(args: Array[String]) {   **val** host = **"192.168.10.1"  val** port = 8888  *//创建ActorSystem的必要参数* **val** configStr =  **s"""  |akka.actor.provider = "akka.remote.RemoteActorRefProvider"  |akka.remote.netty.tcp.hostname = "$**host**"  |akka.remote.netty.tcp.port = "$**port**"  """**.stripMargin  **val** config = ConfigFactory.*parseString*(configStr)  *//ActorSystem是单例的，用来创建Actor* **val** actorSystem = ActorSystem.*create*(**"MasterActorSystem"**, config)  *//启动Actor，Master会被实例化，生命周期方法会被调用* actorSystem.actorOf(*Props*[Master], **"Master"**)  } } |

## Worker类

|  |
| --- |
| **package** cn.itcast.spark  **import** java.util.UUID **import** scala.concurrent.duration.\_ **import** akka.actor.{ActorSelection, Props, ActorSystem, Actor} **import** akka.actor.Actor.Receive **import** com.typesafe.config.ConfigFactory  */\*\*  \* Worker为整个集群的从节点  \* Worker继承了Actor  \*/* **class** Worker **extends** Actor{   *//Worker端持有Master端的引用（代理对象）* **var** *master*: ActorSelection = **null** *//生成一个UUID，作为Worker的标识* **val** *id* = UUID.*randomUUID*().toString   *//构造方法执行完执行一次* **override def** preStart(): Unit = {  *//Worker向MasterActorSystem发送建立连接请求  master* = *context*.system.actorSelection(**"akka.tcp://MasterActorSystem@192.168.10.1:8888/user/Master"**)  *//Worker向Master发送注册消息  master* ! *RegisterWorker*(*id*, **"192.168.10.1"**, 10240, 8)  }   *//该方法会被反复执行，用于接收消息，通过case class模式匹配接收消息* **override def** receive: Receive = {  *//Master向Worker的反馈信息* **case** *RegisteredWorker*(masterUrl) => {  **import** *context*.dispatcher  *//启动定时任务，向Master发送心跳  context*.system.scheduler.schedule(0 millis, 5000 millis, *self*, SendHeartBeat)  }   **case** SendHeartBeat => {  *println*(**"worker send heartbeat"**)  *master* ! *HeartBeat*(*id*)  }  } }  **object** Worker {  **def** main(args: Array[String]) {  **val** clientPort = 2552  *//创建WorkerActorSystem的必要参数* **val** configStr =  **s"""  |akka.actor.provider = "akka.remote.RemoteActorRefProvider"  |akka.remote.netty.tcp.port = $**clientPort  **"""**.stripMargin  **val** config = ConfigFactory.*parseString*(configStr)  **val** actorSystem = *ActorSystem*(**"WorkerActorSystem"**, config)  *//启动Actor，Master会被实例化，生命周期方法会被调用* actorSystem.actorOf(*Props*[Worker], **"Worker"**)  } } |