

# Scala编程——直播（讲师：霄嵩）

## 一、收集问题解答

1、Scala的单例是线程安全的是吗？有懒汉式与饿汉式之分吗？

安全  
懒汉式单例类.在第一次调用的时候实例化自己

2、Scala的mutable与var都表示可变，区别是什么？

var 可变变量, val 不可变的变量  
  
mutable可变集合, immutable不可变集合

3、Scala的列表与数组的区别是什么？

Scala中的List是不可变的递归数据，是Scala中的一种基础结构。  
ListBuffer  
  
Scala的Array应该是由Java array生成的  
ArrayBuffer  
  
建议大家多用List列表

4、对于函数、方法、闭包、柯里化、高阶函数，在实际的开发场景中要怎么选择

函数、方法、高阶函数，开发中常用  
Scala中的高阶函数，相当于Spark Flink中的算子  
map flatMap filter reduceByKey(Spark中的) keyBy(Flink中的)  
foreach mapValues collect sortBy  
  
闭包、柯里化 隐式函数，Spark源码中常见，要看懂

5、扩展一下Scala的高阶算子底层的实现

源码  
以后学习Spark Flink，多看源码  
一切技术问题，答案都在源码

6、扩展

在Java项目开发中，常会用到Model实体类  
用Scala进行Spark Flink开发时，常会用到case class样例类，这个样例类就相当于Java开发中的实体类  
case class Student(name:String,age:Int,sex:String)

伴生类与伴生对象，一般一起出现使用

class Teacher () 伴生类

object Teacher() 伴生对象

模式匹配

\*\*\*\*match .....case

Scala技术书

快学Scala

Scala编程

Scala程序设计

HA高可靠

Master: NameNode

Active

StandBy

Slave:DataNode

## 二、作业讲解

### 1、百元喝酒

作业要求：每瓶啤酒2元，3个空酒瓶或者5个瓶盖可换1瓶啤酒。100元最多可喝多少瓶啤酒？（不允许借啤酒）

思路：利用递归算法，一次性买完，然后递归算出瓶盖和空瓶能换的啤酒数

```
object Beer {
    val money = 100
    val tuple3 = (100, 0, 0)
    val count = 0

    //cap瓶盖, empty空瓶, bottle啤酒瓶
    def GetNumber(cap: Int, empty: Int, bottle: Int): Int = {
        if (empty < 3 && cap < 5) {
            return bottle
        }
        val a = empty / 3 //空瓶可换的啤酒数
        val b = empty % 3 //空瓶换完剩下的空瓶数
        val c = cap / 5 //瓶盖可换的啤酒数
        val d = cap % 5 //瓶盖换完剩下的瓶盖数

        printf("cap = %d,empty = %d,bottle = %d\n", a + c + d, a + c + b, a + c + bottle);
        GetNumber(a + c + d, a + c + b, a + c + bottle);
    }

    def main(args: Array[String]): Unit = {
        val num = 100 / 2;
        val count = num + GetNumber(num, num, 0)
    }
}
```

```
        println(s"一共可以喝$count 瓶");  
    }  
}
```

## 2、人机猜拳

### 1.1 作业需求

1. 选取对战角色
2. 开始对战，用户出拳，与对手进行比较，提示胜负信息
3. 猜拳结束算分，平局都加一分，获胜加二分，失败不加分
4. 循环对战，当输入“n”时，终止对战，并显示对战结果
5. 游戏结束后显示得分

如下图所示：

```
-----欢迎进入游戏世界-----  
*****  
*****猜拳,开始*****  
*****  
  
请选择对战角色:(1.刘备  2.关羽  3.张飞)  
1  
你选择了与刘备对战  
要开始么? y/n  
y  
请出拳! 1.剪刀 2.石头 3.布  
1  
你出拳:剪刀  
刘备出拳!  
刘备出拳:剪刀  
结果: 和局! 下次继续努力!  
是否开始下一轮 (y/n)  
y
```

```
Problems @ Javadoc Declaration Console X
<terminated> demo01 (1) [Java Application] /Library/Java/JavaVi
n
输入不符合规范,默认出布!
刘备出拳!
刘备出拳:石头
结果: 恭喜,你赢啦!
是否开始下一轮 (y/n)
n
退出游戏!

-----

刘备 VS 游客
对战次数3次
```

```
Problems @ Javadoc Declaration Console X
<terminated> demo01 (1) [Java Application] /Library/Java/JavaVirtualMach
n
退出游戏!

-----

刘备 VS 游客
对战次数3次

姓名    等分    胜局    和局    负局
游客    4       1       2       0
刘备    2       0       2       1

-----
```

## 1.2 作业分析

分析业务逻辑，抽象出类、类的属性和方法，如下：

1. 创建用户类User，定义类的属性（name，score）和类的方法（showFist()）
2. 创建计算机类Computer，定义类的属性（name，score）和类的方法（showFist()）
3. 实现计算机随机出拳
4. 创建游戏类Game，定义类的属性（甲方玩家、乙方玩家、对战次数）
5. 编写初始化方法、游戏开始方法

### 3、用户位置时长统计

现有如下数据需要处理： 字段：用户ID，位置ID，开始时间，停留时长（分钟）

4行样例数据： UserA,LocationA,8,60 UserA,LocationA,9,60

UserB,LocationB,10,60 UserB,LocationB,11,80 样例数据中的数据含义是： 用户UserA，在LocationA位置，从8点开始，停留了60钟

处理要求： 1、对同一个用户，在同一个位置的多条记录进行合并（要求做了调整） 2、合并原则：开始时间取最早时间，停留时长累计求和

```
case class UserInfo(userName: String, location: String, startTime: Int, duration: Int)

object LocationDemo {
  def main(args: Array[String]): Unit = {
    val userInfoList: List[UserInfo] = List(
      UserInfo("UserA", "LocationA", 8, 60),
      UserInfo("UserA", "LocationA", 9, 60),
      UserInfo("UserB", "LocationB", 10, 60),
      UserInfo("UserB", "LocationB", 11, 80),
      UserInfo("UserA", "LocationB", 11, 60)
    )

    //UserA,LocationA,List[UserInfo]
    //(UserA,LocationA,List{UserInfo("UserA", "LocationA", 8, 60), UserInfo("UserA",
    "LocationA", 9, 60)})
    val userMap: Map[String, List[UserInfo]] = userInfoList.groupBy(t => t.userName + "," +
    t.location)

    //(UserA,LocationA,List{UserInfo("UserA", "LocationA", 8, 60), UserInfo("UserA",
    "LocationA",9, 60),UserInfo("UserA", "LocationA",11, 60),})
    val orderByUserMap: Map[String, List[UserInfo]] = userMap.mapValues(t => t.sortBy(x =>
    x.startTime))

    var firstTime: Int = 0
    val totalMap: Map[String, Int] = orderByUserMap.mapValues(t => {
      firstTime = t.head.startTime
      val sum = t.map(x => x.duration).sum
      sum
    })

    totalMap.foreach {
      case (datas, sumTime) => println(s"$datas,$firstTime $sumTime")
    }
  }
}
```

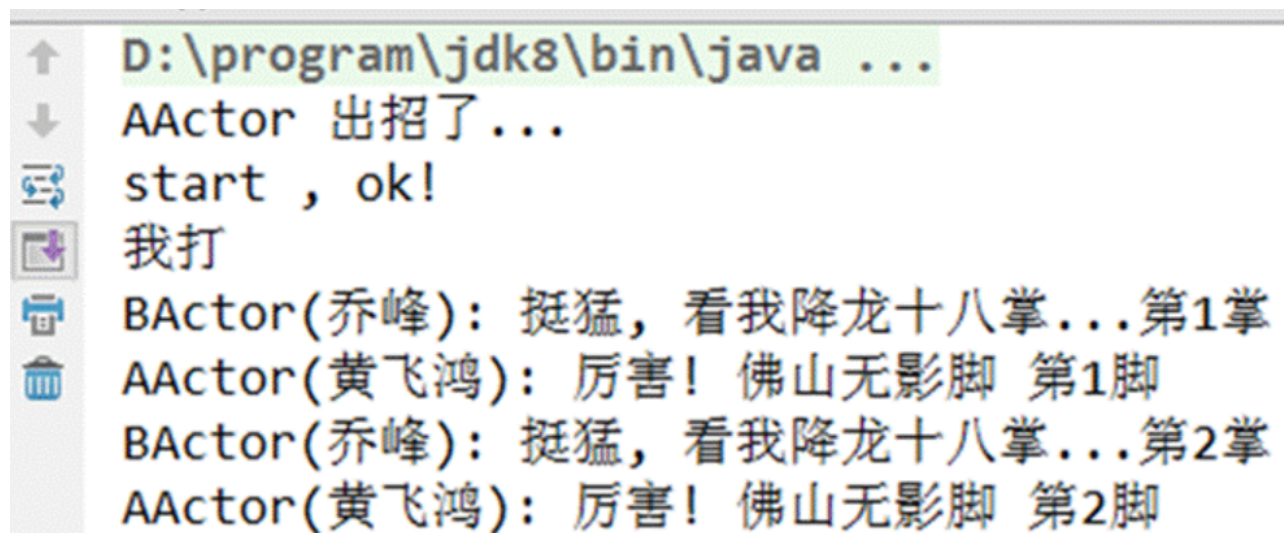
```
}  
}
```

## 4、Actor间通讯

作业要求：

- 1、编写 2 个 Actor，分别是 AActor 和 BActor
- 2、AActor 和 BActor 之间可以相互发送消息

如下图所示：



```
D:\program\jdk8\bin\java ...  
AActor 出招了...  
start , ok!  
我打  
BActor(乔峰): 挺猛, 看我降龙十八掌...第1掌  
AActor(黄飞鸿): 厉害! 佛山无影脚 第1脚  
BActor(乔峰): 挺猛, 看我降龙十八掌...第2掌  
AActor(黄飞鸿): 厉害! 佛山无影脚 第2脚
```

AActor

```
class AActor(iBActorRef:ActorRef) extends Actor{  
  val bActorRef = iBActorRef  
  var count = 0  
  override def receive: Receive = {  
    case "start" => {  
      println("AActor 启动")  
      println("stark ok")  
      println("我打")  
      //发给 BActor  
      bActorRef ! "我打"  
    }  
    case "我打" => {  
      count += 1  
      println(s"AActor(黄飞鸿) 挺猛 看我佛山无影脚 第${count}脚")  
      Thread.sleep(1000)  
      bActorRef ! "我打"  
    }  
  }  
}
```

BActor

```
import akka.actor.Actor

class BActor extends Actor{
  var count = 0
  override def receive:Receive = {
    case "我打" => {
      count += 1
      println(s"BActor(乔峰) 厉害 看我降龙十八掌 第${count}掌")
      Thread.sleep(1000)
      sender() ! "我打"
    }
  }
}
```

ActorGame

```
import akka.actor.{ActorRef, ActorSystem, Props}

object ActorGame {
  def main(args: Array[String]): Unit = {
    //1. ActorSystemme
    val actorfactor = ActorSystem("actorfactor")
    val bActorRef: ActorRef = actorfactor.actorOf(Props[BActor], "BActor")
    val aActorRef: ActorRef = actorfactor.actorOf(Props(new AActor(bActorRef)), "AActor")
    //做一个要求: 当 100 招, 就退出..
    aActorRef ! "start"
  }
}
```

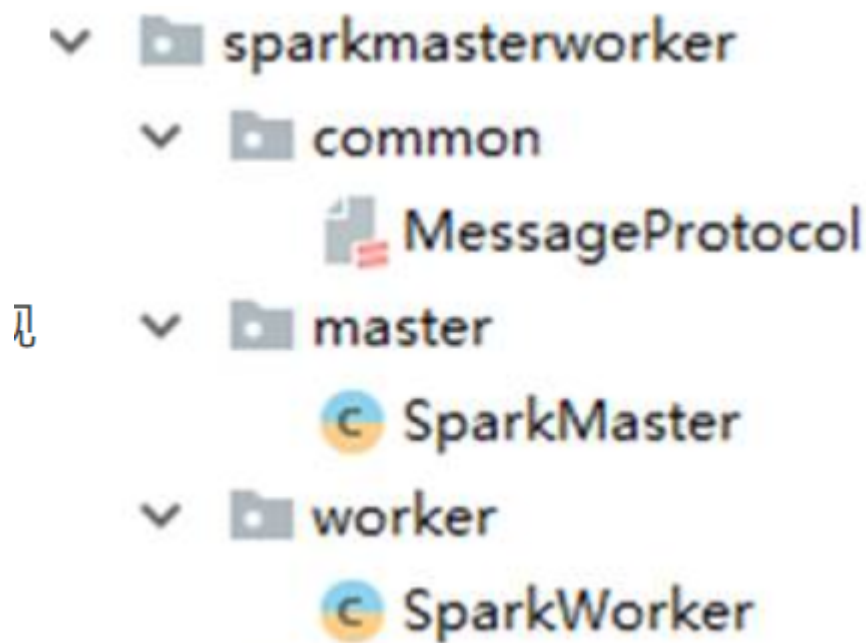
## 5、模拟Spark中Master与Worker进程通讯

为了加深对主从服务心跳检测机制（HeartBeat）的理解，模拟master与slave之间的通信。

作业要求：

1. Worker 注册到 Master，Master 完成注册，并回复 Worker 注册成功(注册功能)
2. Worker 定时发送心跳，并在 Master 接收到
3. Master 接收到 Worker 心跳后，要更新该 Worker 的最近一次发送心跳的时间
4. 给 Master 启动定时任务，定时检测注册的 Worker 有哪些没有更新心跳，并将其从 hashmap 中删除

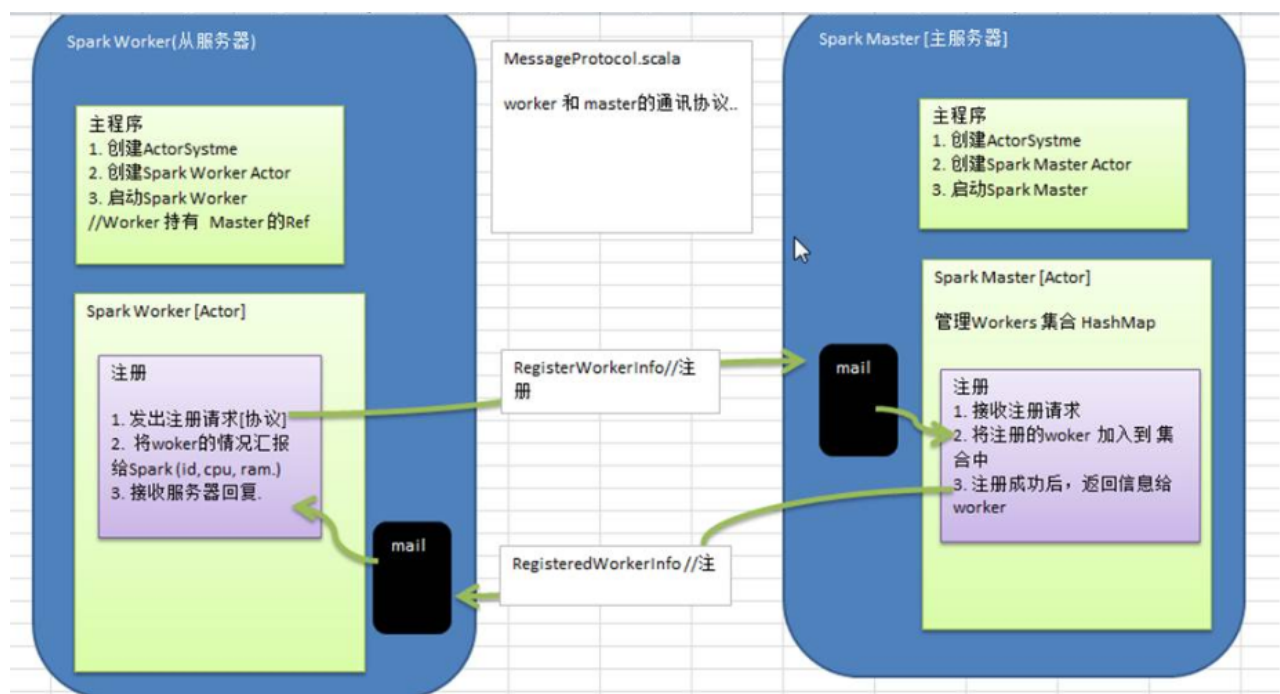
代码结构：



## 功能1:Worker 完成注册

- 功能说明

worker 注册到 Master, Master 完成注册, 并回复 worker 注册成功



```
//MessageProtocol.scala

package com.test.akka.sparkmasterworker.common

//样例类, 注册的协议, 包含 id ,cpu, ram(内存)
case class RegisterWorkerInfo(id: String, cpu: Int, ram: Int)
```



```
//WorkerInfo 是用于保存 worker 信息的对象，它不在网络传输，他是普通类
//后面会加入扩展内容，比如心跳时间
class WorkerInfo(val id: String, val cpu: Int, val ram: Int) {
    //默认的心跳时间
    var lastHeartBeatTime: Long = System.currentTimeMillis()
}

//如果注册成功后，返回的协议信息，因为不需要属性，因此我直接使用的 case object
//后面直接返回的是 RegisteredWorkerInfo 对象：类型 RegisteredWorkerInfo$
case object RegisteredWorkerInfo
```

```
//SparkMaster.scala

package com.test.akka.sparkmasterworker.master

import akka.actor.{Actor, ActorRef, ActorSystem, Props}
import com.test.akka.sparkmasterworker.common.{RegisterWorkerInfo, RegisteredWorkerInfo, WorkerInfo}
import com.typesafe.config.ConfigFactory

import scala.collection.mutable

class SparkMaster extends Actor {
    //定义一个 hashMap,存放所有的 workers 信息
    val workers = mutable.HashMap[String, WorkerInfo]()

    override def receive = {
        case "start" => {
            println("spark master 启动, 在 10000 监听..")
        }
        case RegisterWorkerInfo(id, cpu, ram) => {
            //注册
            //先判断是否已经有 id
            if (!workers.contains(id)) {
                //创建 WorkerInfo
                val workerInfo = new WorkerInfo(id, cpu, ram)
                workers += (id -> workerInfo)
                //workers += ((id,workerInfo))
                //回复成功!
                sender() ! RegisteredWorkerInfo
                println(s"workerid= $id 完成注册~")
            }
        }
    }
}

object SparkMaster extends App {

    val masterHost = "127.0.0.1"
    val masterPort = 10000
```

```

val config = ConfigFactory.parseString(
  s"""
    akka.actor.provider="akka.remote.RemoteActorRefProvider"
    akka.remote.netty.tcp.hostname=$masterHost
    akka.remote.netty.tcp.port=$masterPort
  """).stripMargin)

//创建 ActorSystem
// "SparkMaster" actorFactory 名字, 程序指定
val sparkMasterActorSystem = ActorSystem("SparkMaster", config)
//创建 SparkMaster 和 引用
val sparkMaster01Ref: ActorRef = sparkMasterActorSystem.actorOf(Props[SparkMaster],
"SparkMaster01")
  sparkMaster01Ref ! "start"
}

```

```

//Sparkworker.scala

package com.test.akka.sparkmasterworker.worker

import java.util.UUID

import akka.actor.{Actor, ActorRef, ActorSelection, ActorSystem, Props}
import com.test.akka.sparkmasterworker.common.{RegisterWorkerInfo, RegisteredWorkerInfo}
import com.test.akka.sparkmasterworker.master.SparkMaster.{masterHost, masterPort}
import com.typesafe.config.ConfigFactory

class SparkWorker(masterHost:String,masterPort:Int) extends Actor{
  var masterProxy: ActorSelection = _
  val id = UUID.randomUUID().toString

  override def preStart(): Unit = {
    masterProxy =
context.actorSelection(s"akka.tcp://SparkMaster@${masterHost}:${masterPort}/user/SparkMaster
01")
  }
  override def receive = {
    case "start" => {
      println("spark worker 启动..")
      //发出注册的请求
      masterProxy ! RegisterWorkerInfo(id, 8, 8 * 1024)
    }
    case RegisteredWorkerInfo => {
      println(s"收到 master 回复消息 workerid= $id 注册成功")
    }
  }
}

object SparkWorker extends App{
  val (masterHost,masterPort,workerHost,workerPort) =

```

```

    ("127.0.0.1",10000,"127.0.0.1",10001)
    val config = ConfigFactory.parseString(
      s"""
        |akka.actor.provider="akka.remote.RemoteActorRefProvider"
        |akka.remote.netty.tcp.hostname=$workerHost
        |akka.remote.netty.tcp.port=$workerPort
        |""".stripMargin)

    val sparkWorkerActorSystem = ActorSystem("SparkWorker",config)

    val sparkWorkerActorRef: ActorRef = sparkWorkerActorSystem.actorOf(Props(new
    SparkWorker(masterHost, masterPort)), "SparkWorker-01")

    sparkWorkerActorRef ! "start"

  }

```

## 功能2:Worker 定时发送心跳

- 功能说明

worker 定时发送心跳给 Master, Master 能够接收到,并更新 worker 上一次心跳时间

- 代码实现

```

//MessageProtocol.scala

//worker 在注册成功后, 通过定时器, 每隔 3s 发送一个消息给自己
//SendHeartBeat
case object SendHeartBeat
//当定时器发送了一个 SendHeartBeat 消息后, worker 发送一个消息
// (HeartBeat(id: String))给 Master
case class HeartBeat(id: String)

```

- ```

override def receive = {
  case "start" => {
    println("spark worker 启动..")
    //发出注册的请求
    masterProxy ! RegisterWorkerInfo(id, 8, 8 * 1024)
  }
  case RegisteredWorkerInfo => {
    println(s"收到 master 回复消息 workerid= $id 注册成功")
    //启动一个定时器.
    import context.dispatcher
    //说明
    //1.schedule 创建一个定时器
    //2.0 millis, 延时多久才执行, 0 表示不延时, 立即执行
    //3. 3000 millis 表示每隔多长时间执行 3 秒
    //4. self 给自己发送 消息
    //5. SendHeartBeat 消息
    context.system.scheduler.schedule(0 millis, 3000 millis, self, SendHeartBeat)
  }
}

```

```

}
case SendHeartBeat => {
  println(s"workerid= $id 发出心跳~")
  masterProxy ! HeartBeat(id)
}
}

```

```
//SparkMaster.scala
```

```

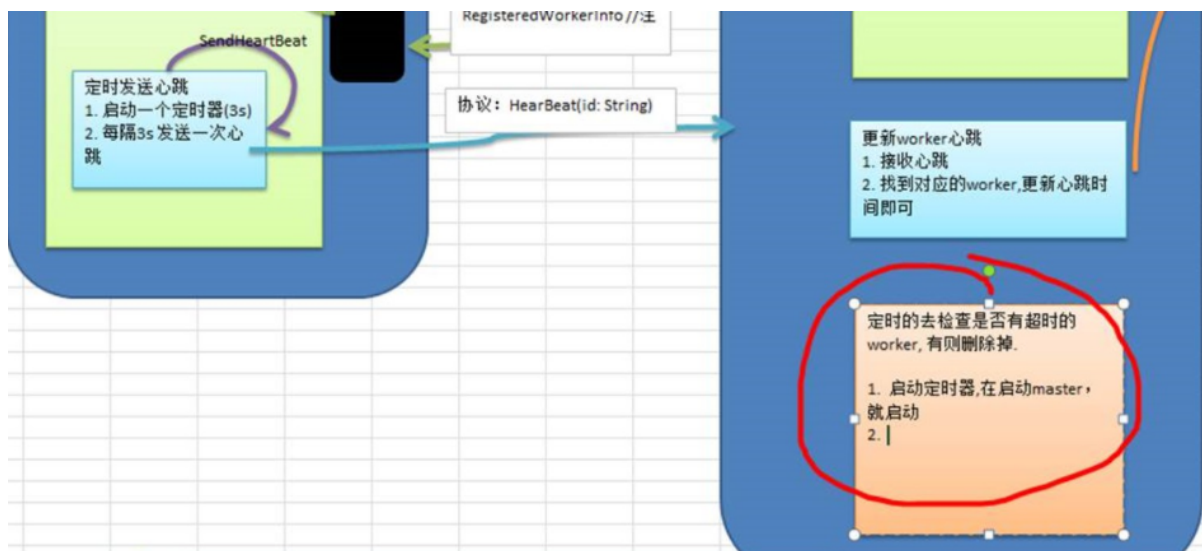
case HeartBeat(id) => {
  //更新 id 对应的 worker 的心跳
  if (workers.contains(id)) {
    workers(id).lastHeartBeatTime = System.currentTimeMillis()
    println(s"workerid=$id 更新心跳成功~")
  }
}
}

```

### 功能3：Master启动定时任务，定时检测注册的Worker

#### 功能说明

功能要求：Master 启动定时任务(10秒)，定时检测注册的 worker 有哪些没有更新心跳，已经超时的 worker(6 秒)，将其从 hashmap 中删除掉



```
//SparkMaster.scala
```

```

override def receive = {
  case "start" => {
    println("spark master 启动, 在 10000 监听..")
    self ! StartTimeoutWorker
  }
  case RegisterWorkerInfo(id, cpu, ram) => {
    //注册
    //先判断是否已经有 id
    if (!workers.contains(id)) {

```

```

        //创建 workerInfo
        val workerInfo = new WorkerInfo(id, cpu, ram)
        workers += (id -> workerInfo)
        //workers += ((id,workerInfo))
        //回复成功!
        sender() ! RegisteredWorkerInfo
        println(s"workerid= $id 完成注册~")
    }
}

case HeartBeat(id) => {
    //更新 id 对应的 worker 的心跳
    if (workers.contains(id)) {
        workers(id).lastHeartBeatTime = System.currentTimeMillis()
        println(s"workerid=$id 更新心跳成功~")
    }
}

case StartTimeOutWorker =>{
    //启动定时器
    import context.dispatcher
    context.system.scheduler.schedule(0 millis, 10000 millis, self, RemoveTimeOutWorker)
}

case RemoveTimeOutWorker => {
    //定时清理超时 6s 的 worker,scala
    //获取当前的时间
    val currentTime = System.currentTimeMillis()
    val workersInfo = workers.values //获取到所有注册的 worker 信息
    //先将超时的一次性过滤出来, 然后对过滤到的集合一次性删除
    workersInfo.filter(
        currentTime - _.lastHeartBeatTime > 6000
    ).foreach(workerInfo=>{
        workers.remove(workerInfo.id)
    })

    printf("当前有%d 个 worker 存活\n", workers.size)
}
}

```

## 功能 4:Master与Worker 的启动参数运行时指定

- 功能说明

功能要求: Master, Worker 的启动参数运行时指定, 而不是固定写在程序中的

- 代码实现

- `//Sparkworker.scala`

```
if (args.length != 6) {  
    println("参数格式不正确 <masterHost masterPort masterName workerHost workerPort  
workerName>")  
}  
  
val (masterHost, masterPort, masterName, workerHost, workerPort, workerName) =  
    (args(0), args(1), args(2), args(3), args(4), args(5))
```