## Node Manager启动分析：

org.apache.hadoop.yarn.server.nodemanager类main方法启动:

---> Main

--->nodeManager.initAndStartNodeManager (conf, **false**);

---> 调用父类的**this**.init(conf);方法,init()方法中调用serviceInit()方法,此方法在子类NodeManager中重写,进入重写方法

---> 主要创建NM级的处理调度任务的AsyncDispacher,创建节点健康检查服务nodeHealthChecker,NM上下文context,nodeStatusUpdater,node资源监控对象nodeResourceMonitor等.dispacher注册两个类型后将dispacher加到service数组中.

**NodeManager中注册的服务：**

**(存放在serviceList数组中,父类CompositeService的serviceInit方) (法将数组中的服务逐个遍历出来,进行调用各服务子类重写的) (service.init(conf)方法,递归调用执行本服务中注册的其它所有服务)**

**ServiceList:**

**[**

**Deletion: NOTINITED,**

**NodeHealthChecker: NOTINITED,**

**NodeResourceMonitorImpl: NOTINITED,**

**containermanager.ContainerManagerImpl: NOTINITED,**

**webapp.WebServer: NOTINITED,**

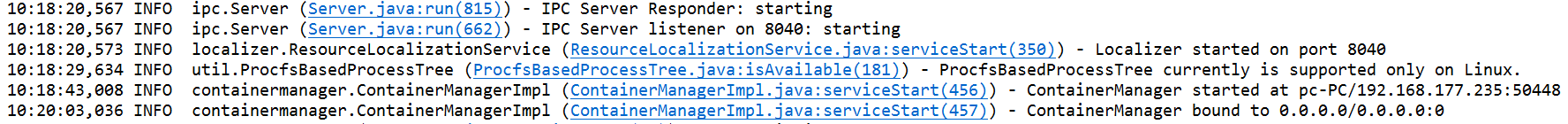
**Dispatcher: NOTINITED,**

**NodeStatusUpdaterImpl: NOTINITED**

**] --->核心方法:serviceInit(conf)递归调用，将所有服务都初始化一遍**

**\*containerManagerImpl服务start过程包括更新NodeId,将NodeId信息放入上下文对象context中,启动一个server,之后启动ContainerManager**

**Log.Info --->**



**\*NodeStatusUpdaterImpl服务初始化过程和启动过程分析:**

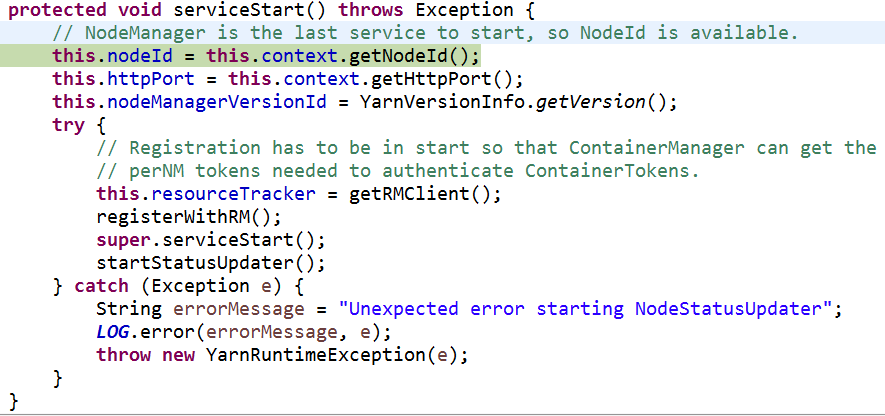
**NodeStatisUpdaterImpl服务 : ServiceInit(conf)初始化方法:获得YarnConfiguration中配置的本机内存,虚拟内存,虚拟核数(默认电脑8核)等**

**Log.Info --->**

Initialized nodemanager for null: physical-memory=8192 virtual-memory=17204 virtual-cores=8

---> 调用父类的**This**.start ();方法,start()方法中调用了serviceStart()方法,子类NodeManager中重写了该方法:在执行父类serviceStart()方法之上增加了安全登陆操作doSecureLogin();遍历serviceList数组,服务逐个启动.

**NodeStatusUpdaterImpl服务 : ServiceStart()方法**



This.resourceTracker : NM 与 RM 之间的协议 ---> NM通过RPC协议向RM注册,并定时发送心跳信息汇报节点资源使用情况和contianer运行情况. (YarnRpc相关类图见<<hadoop源码--Yarn RPC>>)

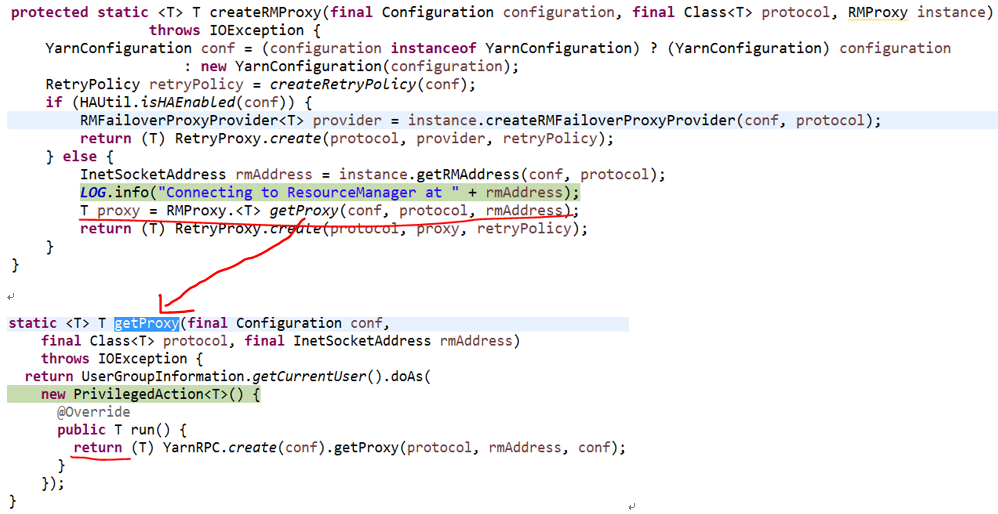
(1) 创建RM代理

This.resourceTracker=getRMClient (); --->(类型: ResourceTrackerPBClientImpl)

---> ServerRMProxy.*createRMProxy* (conf, ResourceTracker.**class**);

---> createRMProxy (**final** Configuration configuration,

**final** Class<T> protocol, RMProxy instance)



YarnRPC.create(conf).getProxy(protocol,rmAddress,conf)最终调用

--->RpcClientFactoryPBImpl类中的getClient方法

最终得到的this.resourceTracker类型：ResourceTrackerPBClientImpl.

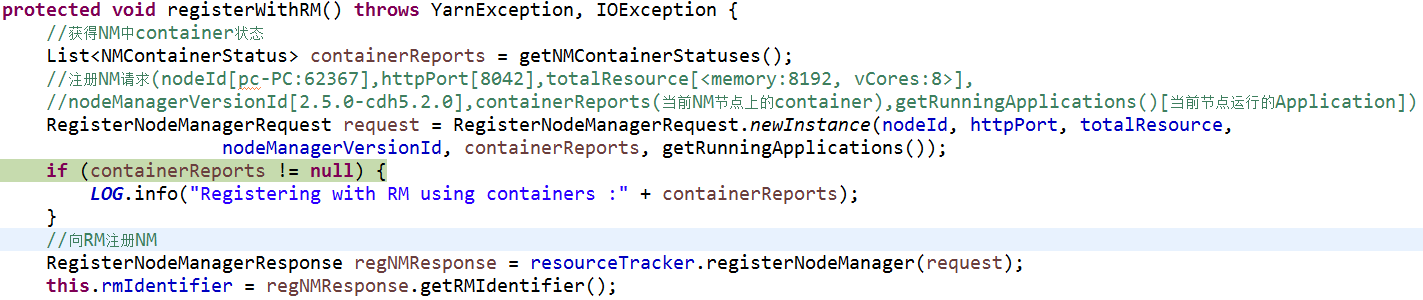
(2) RegisterWithRM (); ---- NM向RM注册

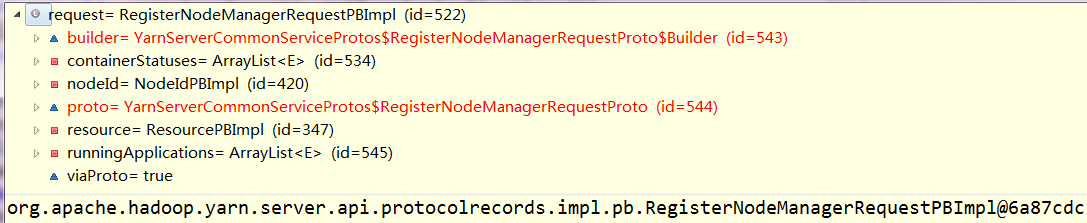
RegisterNodeManagerResponse regNMResponse=resourceTracker.registerNodeManager (request);

这行代码反复使用了动态代理和反射,上一步获得的ResourceTracker实现类ResourceTrackerPBClientImpl, 创建ResourceTrackerPBClientImpl的代理,具体代理处理为RetryInvocationHandler类,当执行

ResourceTracker.registerNodeManager(request)时,会动态代理进入此类的invoke方法.其中代码

Object ret = invokeMethod(method, args);最终调用ResourceTrackerPBClientImpl类的registerNodeManager(request)方法

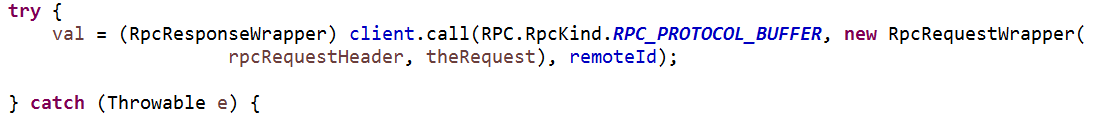




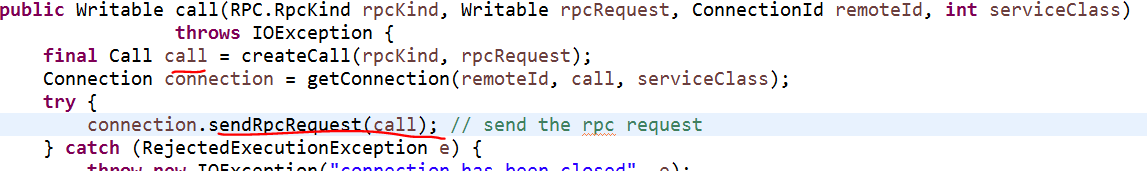
动态代理确定调用ResourceTrackerPBClientImpl类的registerNodeManager方法,

核心代码--->**new** RegisterNodeManagerResponsePBImpl(proxy2.registerNodeManager(**null**, requestProto));

其中proxy.registerNodeManager(null, requestProto);经过动态代理调用ProtobufRpcEngine类的getProxy方法得到代理,里面动态代理调用内部类Invoker的invoke方法,其中核心代码



客户端调用call方法通过Rpc协议连接server端(ResourceManager),连接成功后将包装请求参数方法等的包装对象call传到RM端进行注册Node操作,核心代码 --->



**在这一步即为NM和RM之间的通信,远程过程调用过程(RPC)**

Node :



(3)在(2)步中RM端获得连接请求后最终调用ResourceTrackerService类的registerNodeManager方法,经过一系列检查

Node : ResourceTrackerService类在ResourceManager子项目包下--->

**package** org.apache.hadoop.yarn.server.resourcemanager;剩下的流程全是在RM端注册NM的

核心代码:

RMNode rmNode = **new** RMNodeImpl(nodeId, rmContext, host, cmPort, httpPort, *resolve*(host), capability,

nodeManagerVersion);

(RMNode : Node managers information on available resources and other static information.)

RMNode oldNode = **this**.rmContext.getRMNodes().putIfAbsent(nodeId, rmNode); //第一次启动,值为null

**if** (oldNode == **null**) {

**this**.rmContext.getDispatcher()

.getEventHandler()

.handle(**new** RMNodeStartedEvent(nodeId, request.getNMContainerStatuses(), request.getRunningApplications()));

} **else** {

***LOG***.info("Reconnect from the node at: " + host);

**this**.nmLivelinessMonitor.unregister(nodeId);

**this**.rmContext.getDispatcher().getEventHandler()

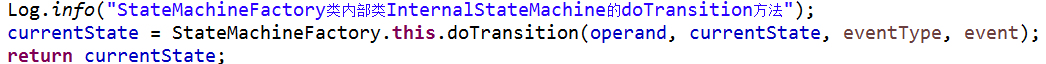
.handle(**new** RMNodeReconnectEvent(nodeId, rmNode, request.getRunningApplications()));

}

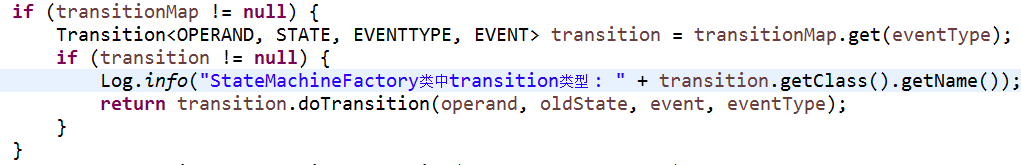
在AsynDispatcher中dispatch响应的事件,调用RMNodeImpl的handle方法,参数RMNodeEventType.START

---> stateMachine.doTransition(event.getType(), event);

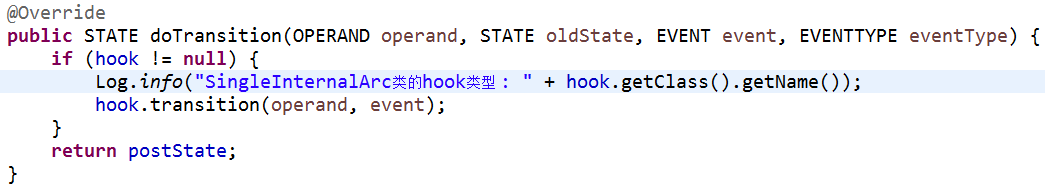
--->



--->



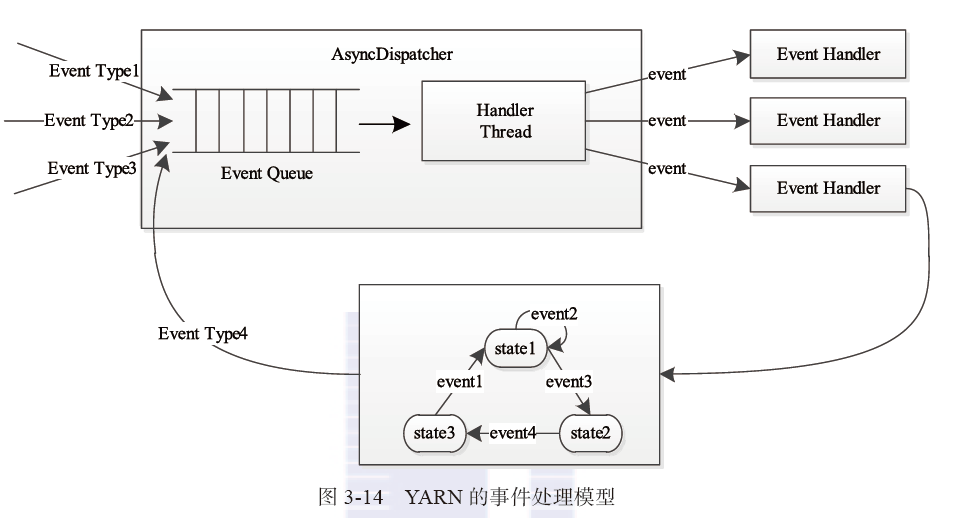
--->(hook类型为AddNodeTransition,为什么?看RMNodeImpl里new出来的状态机工厂对象stateMachineFactory)



回调AddNodeTransition类的transition方法:



**Node:上面转的流程其实就是一个有限状态机的过程,根据类型StateMachineFactory可知为状态机工厂.**



两句代码详细流程分析见下两页的[1],[2]

(其实[1],[2]流程的调用过程和上面刚分析完的registerNodeManager方法一样,只不过进入不同的Dispatcher实现类中执行相应的逻辑,以下分析将公共的doTransition方法和StateMachineFactory类的addTransition方法省略,因为调用流程都类似)

[1]调用流程分析:

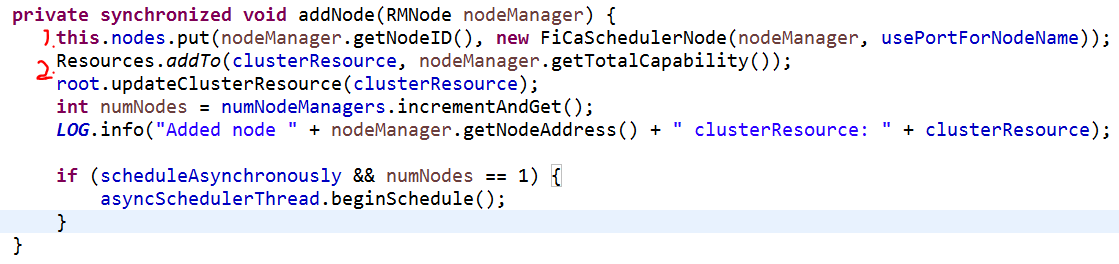
New NodeAddedSchedulerEvent(rmNode, containers)发出SchedulerEventType.NODE\_ADDED事件,调用handle方法

\*handle方法调用流程:进入AsyncDispatcher类的handle方法,根据事件类型<SchedulerEventType.NODE\_ADDED>找到对应事件的Dispatcher<ResourceManager类的内部类SchedulerEventDispatcher,调用SchedulerEventDispatcher的handle方法,最终调用接口YarnScheduler的实现类CapacityScheduler的handle方法,通过一系列的调用最终调用的方法就是CapacityScheduler类中的handle方法>\*



Handle方法根据对应事件类型进行处理,本次调用类型为NODE\_ADDED.添加节点操作

addNode方法作用:1.将当前NM节点加入到nodes的Map集合中 2.并更新集群容量大小.



[2]



调用流程分析:

调用NodeListManager类的handle方法,发出的事件: <NodeListManagerEventType.NODE\_USABLE> :

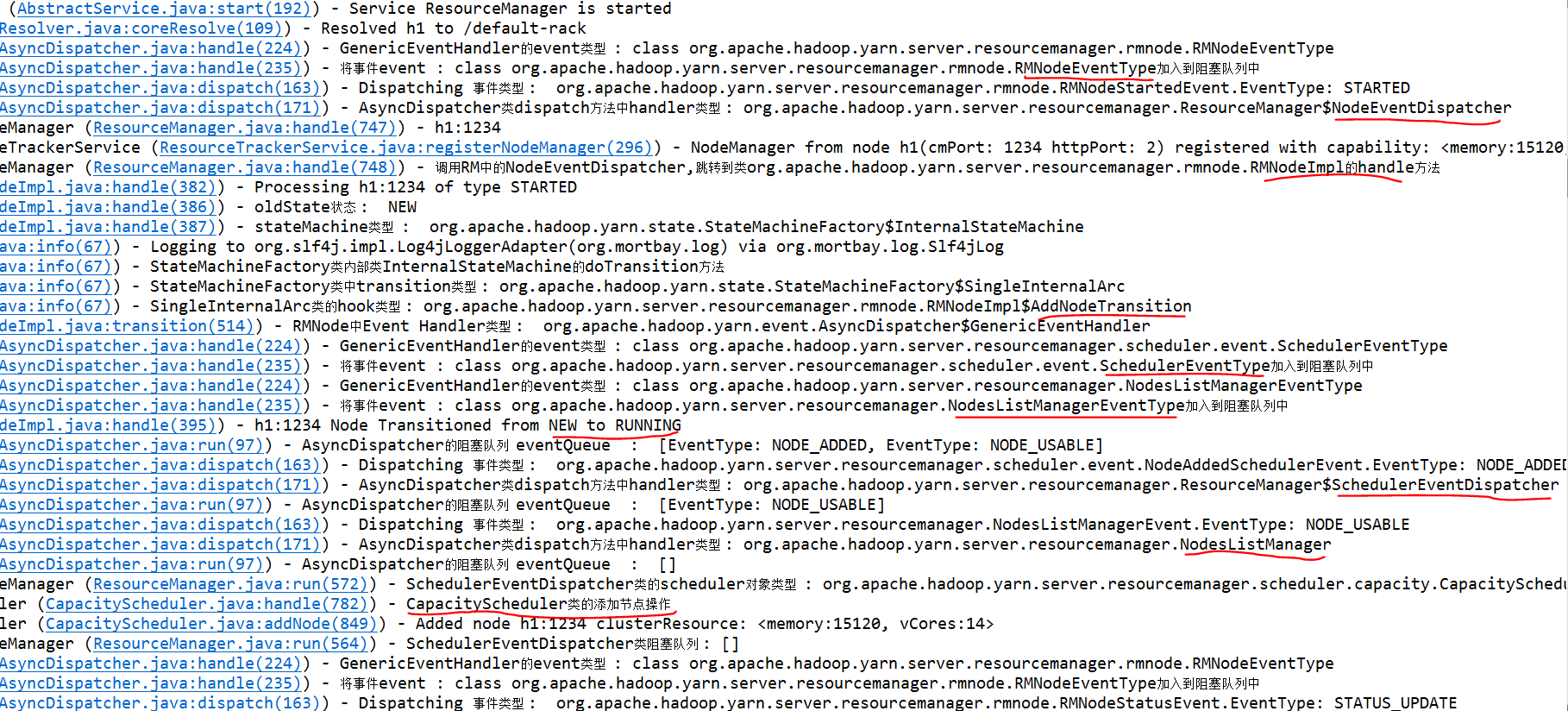


首先从不可用节点集合(unusableRMNodesConcurrentSet)中删除该节点.

该节点可用,调用handle(new RMAppNodeUpdateEvent(…)); new RMAppNodeUpdateEvent发出RMAppEventType.***NODE\_UPDATE***事件,最终调用RMAppImpl的handle方法,调用RmAppImpl类中的StateMachineFactory的addTransition(RMAppState.***NEW***, RMAppState.***NEW***, RMAppEventType.***NODE\_UPDATE***,

**new** RMAppNodeUpdateTransition())方法,状态: RMAppState.NEW ---> RMAppState.NEW,回调RMAppNodeUpdateTransition类中重写的transition方法,将节点加入到更新状态集合updateNodes中

整个注册过程的打印日志如下:



\*\*\*\*\*\*\*\*\*\*\*

\* 整个**this**.rmContext.getDispatcher().getEventHandler().handle(**new** RMNodeReconnectEvent(nodeId, \*rmNode, request.getRunningApplications()));调用方法分析完毕

\*---------------------------------------------------------------------------------------------------

接下来将节点注册到nmLivelinessMonitor中

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* \*

\*以上整个 RegisterNodeManagerResponse regNMResponse = resourceTracker.registerNodeManager(request);\*

\*调用过程分析完毕 \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*