




Akka入门& 精品课教务系统分布式实战

李晨曦

2018.12







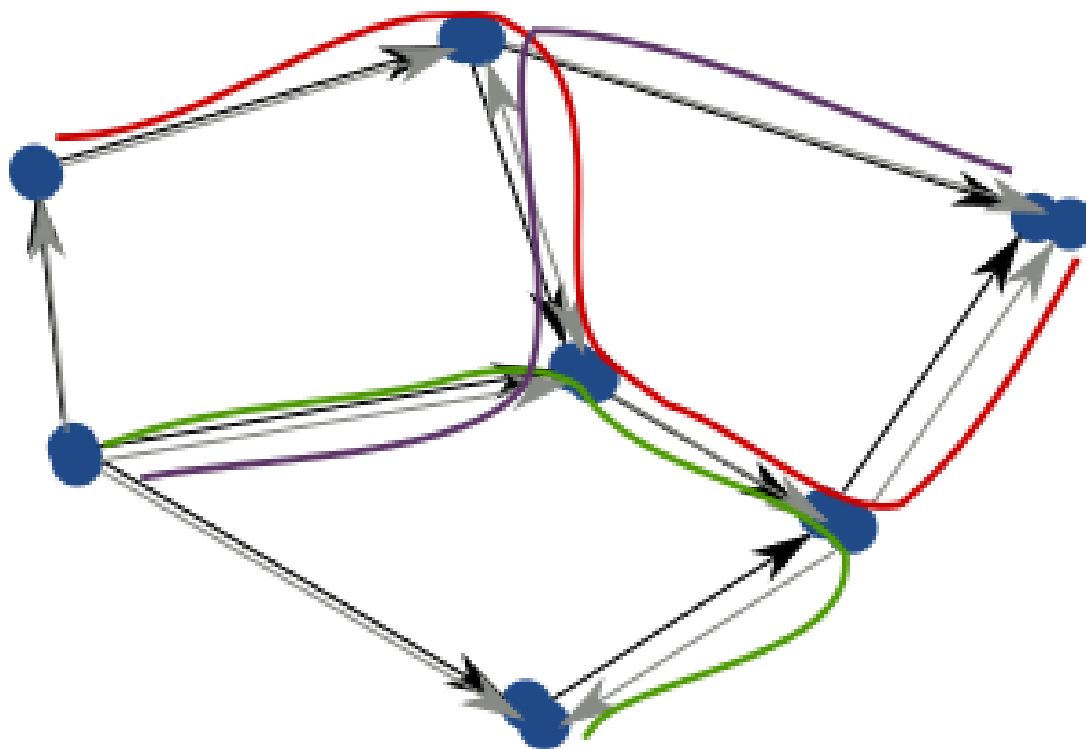
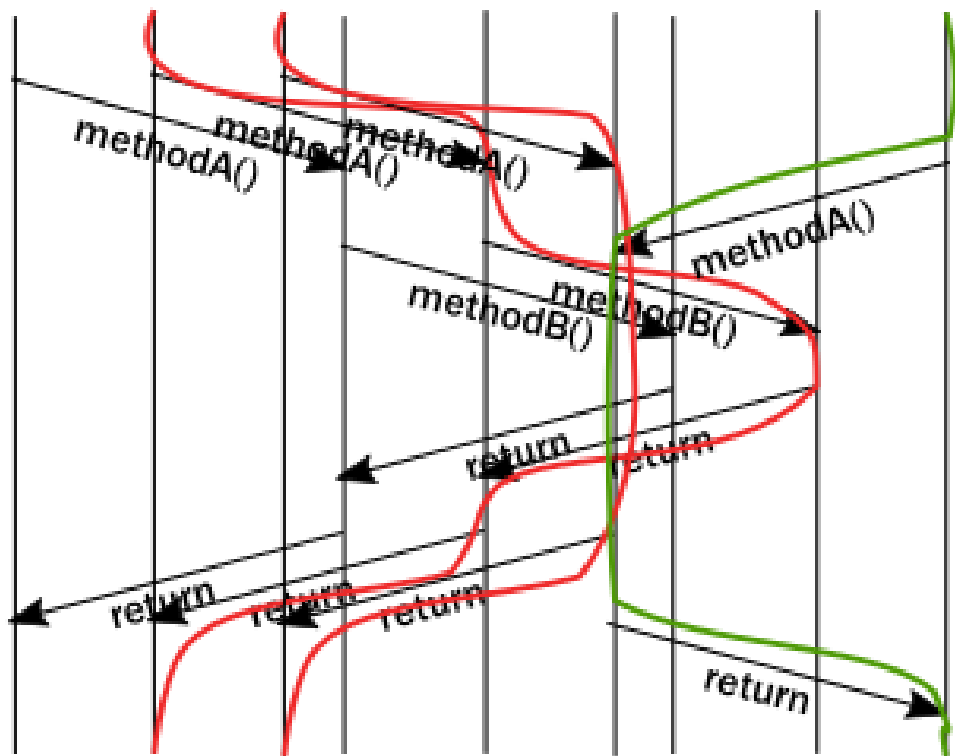
本次TTT概要

- 面向对象编程模型与锁的由来
- 如何抛弃锁与Actor编程模型
- Akka入门
 - 基本概念
 - 定义Actors
 - 持久化与至少一次消息递送
 - 远程Actor
 - 集群
 - 集群——发布订阅
 - 集群——集群单例
 - 集群——集群分片
 - 集群——分布式数据
 - 脑裂处理
 - 与微服务的关系
- 精品课教务系统的使用姿势
 - 分布式化的挑战
 - 编辑资源锁的分布式实现
 - 均匀分布的集群单例任务
 - 前端资源版本
 - 脑裂
 - 其他非akka相关的分布式处理
- Reference & 几个Demo
- Q&A



面向对象编程模型与锁的由来

Object1 Object2 Object3 Object4 Object5 Object6 Object7 Object8 Object9 Object10 Object11 Object12 Object13 Object14 Object15 Object16 Object17 Object18 Object19 Object20 Object21 Object22 Object23 Object24 Object25 Object26 Object27 Object28 Object29 Object30 Object31 Object32 Object33 Object34 Object35 Object36 Object37 Object38 Object39 Object40 Object41 Object42 Object43 Object44 Object45 Object46 Object47 Object48 Object49 Object50 Object51 Object52 Object53 Object54 Object55 Object56 Object57 Object58 Object59 Object60 Object61 Object62 Object63 Object64 Object65 Object66 Object67 Object68 Object69 Object70 Object71 Object72 Object73 Object74 Object75 Object76 Object77 Object78 Object79 Object80 Object81 Object82 Object83 Object84 Object85 Object86 Object87 Object88 Object89 Object90 Object91 Object92 Object93 Object94 Object95 Object96 Object97 Object98 Object99 Object100





如何抛弃锁与Actor编程模型

Actor1

Actor2

Actor3

- 通过消息传递的方式解耦了程序，从而维持了对对象的封装
- 不再需要使用锁。对于actor的内部状态的修改只能通过传递消息来实现，并且同一时间只会有一个消息被处理，如果actor消息被安排进系统，多线程编程为准中的线程竞争的问题。
- 事实上没有任何地方使用到锁，并且消息发送者不会被阻塞。在十几个线程上可以有效安排数百万个actor，充分发掘了现代CPU的潜力。
- Actor的内部状态是本地的，并不会被共享。Actor需要改变数据只能通过消息传递的方式。这就有点像现代存储体系：消息存放在高速缓存上，本地状态存放在原始CPU核上。而这个模型也恰好和远程通信类似。



Actor编程模型

Actor

- 引用、路径、地址
- 状态
- 行为
- 收件箱
- 子Actor及监督策略
- 终止机制
- 生命周期

Actor系统

- 层级结构
- 配置
- 终止机制



Akka是一个由Lightbend(Typesafe)公司使用Scala语言编写的Actor模型的实现。

- JVM兼容、异步、可伸缩、跨处理器内核与网络
- 多线程特性，但不依赖于原子操作或锁等低级并发结构——使你从考虑内存可见性等问题中解放出来。
- 透明的远程系统及其组件之间的交互——使你免除编写与维护复杂的网络通信代码。
- 有弹性的，支持扩缩容的，集群化及高可用的架构——使你可以交付一个真正的响应式系统。



Akka——定义Actor

API

- `createReceive()` 重写该方法以初始化响应体, 使用 `receiveBuilder()` 创建响应体
- `getSelf()` 获取自己的ActorRef
- `getSender()` 获取最近收到的消息的发送方
- `supervisorStrategy()` 重写以更改监控子Actor的策略
- `getContext()` 当前上下文
 - `actorOf(props)` 创建子Actor的方法
 - `getParent()` 父Actor
 - `become()/unbecome()` 响应体热切换
 - `children()/child({name})` 子Actor
 - `watch(actorRef)/unwatch(actorRef)` 生命周期监控
 - `system()` 所属ActorSystem
 - `setReceiveTimeout()` 设置收信超时时间
- 生命周期方法hook
 - `preStart()` 启动时,消费第一条数据前调用
 - `preRestart()` 重启前调用, 原实例
 - `postRestart()` 重启后调用, 新实例, 默认会调用 `preStart()`
 - `postStop()` 停止时调用, 如果是重启, 是原实例

```

... public void /user/someActor/someChild
... public void startReceive() {
... public void terminated(Terminated terminated) {}
... }

```

Actor创建

```

//Props Actor 创建配置参数
Props props1 = Props.create(MyActor.class);
Props props2 = Props.create(ActorWithArgs.class, "arg");
//创建actor
system.actorOf(Props.create({ActorClass}.class), "{ActorName}"); //'/user'下的actor
getContext().actorOf(Props.create({ActorClass}.class), "{ActorName}"); //当前actor的子actor

```

Actor终止

```

//注意方法都异步的
getContext().stop({actorRef}); //终止自身或子Actor
ActorSystem.stop({actorRef}); //终止顶层Actor
ActorSystem.terminate(); //终止系统根监督者, 也即关闭整个系统

```


AbstractPersistentActor 基本的持久化actor实现



- `persistenceId()` 返回固定不变的唯一ID。当actor从持久化数据中恢复状态时，依赖此ID作为数据的key来读取持久化数据。
- `persist(evt, evtHandler)` 存储事件，存储方法是异步的。当存储成功后，系统会生成一个特殊的消息给actor，actor收到消息后会执行 `evtHandler`，这个消息的发送方与对应调用 `persist()` 时处理的消息的发送方相同。虽然是异步的，但是持久化特性保证了在 `persist()` 函数被调用，`evtHandler` 执行前时，其他消息不会插入执行，而是 `evtHandler` 执行后，才会再处理其他消息。多次调用本方法，`evtHandler` 的处理顺序与调用顺序相同。包括可以嵌套调用。
- `saveSnapshot(snapshot)` 存储快照，异步执行。
 - 特殊消息 `SaveSnapshotSuccess` 成功时生成
 - 特殊消息 `SaveSnapshotFailure` 失败时生成，默认处理为打印日志
- `createReceiveRecover()` 实现恢复代码，类似 `createReceive()`，接收 `persist()` 函数存储的事件。注意恢复消息时 `getSender()` 获得的永远是 `deadLetters`，如果一定要获取到消息发送方，可以考虑在事件中加入actorRef。
 - 特殊消息 `SnapshotOffer`，从快照恢复时的快照数据。
 - 特殊消息 `RecoveryCompleted`，恢复结束时的消息，用于进行一些恢复结束后的初始化动作。没有要恢复的事件&快照消息时，也会发送该消息。
- `onPersistFailure()` 存储事件失败时调用，默认实现为打印错误日志。此外actor会被立刻终止。
- `onPersistRejected()` 当持久化有问题时（如序列化问题）调用，默认实现为打印警报日志。此外actor不会终止，继续处理下一条消息。
- `recovery()` 用于自定义恢复策略
 - `SnapshotSelectionCriteria.none()` 用于跳过加载快照并重跑所有事件，用于当快照存储的格式有变化导致不兼容时。注意如果有事件被删除，则不能用此策略。
 - `Recovery.none()` 停用恢复功能
- `recoveryRunning()` 与 `recoveryFinished()` 用于检测恢复是否完成
- `onRecoveryFailure()` 恢复时出现异常时调用，默认实现为打印日志。此外actor会被立刻终止。
- `persistAll()` `persistAllAsync()` 原子性地存储一批事件
- `deleteMessages()` 删除一批事件
 - 特殊消息 `DeleteMessagesSuccess` 删除成功消息
 - 特殊消息 `DeleteMessagesFailure` 删除失败消息
- `deleteSnapshot()` 删除指定序列号及以前的所有快照 `deleteSnapshots()`
 - 特殊消息 `DeleteSnapshotSuccess` `DeleteSnapshotsSuccess` 删除成功消息
 - 特殊消息 `DeleteSnapshotFailure` `DeleteSnapshotsFailure` 删除失败消息
- `journalPluginId()` 重写日志存储所用的插件
- `snapshotPluginId()` 重写快照存储所用的插件

```
static class AtLeastOnceActor extends AbstractPersistentActorWithAtLeastOnceDelivery {
```

```
    ActorSelection receiverSelection = getContext().actorSelection("/user/receiver");
    String status;

    public String persistenceId() {
    }

    @Override
    public Receive createReceiveRecover() {
        return receiveBuilder()
            .match(SnapshotOffer.class, this::recoverSnap)
            .match(Triple.class, this::recoverMethod)
            .match(RecoveryCompleted.class, this::recoverComplete)
            .build();
    }

    public void recoverSnap(SnapshotOffer snapshotOffer) {
        @SuppressWarnings("unchecked")
        Pair<String, AtLeastOnceDeliverySnapshot> snapshot = (Pair<String, AtLeastOnceDeliv
        status = snapshot.getLeft();
        System.out.println("recover status by snapshot : "+status);
        setDeliverySnapshot(snapshot.getRight());
    }

    private void recoverMethod(Triple<String, String, Long> msg) throws Exception {}
    public void sendRecover(Triple<String, String, Long> msg) {
        System.out.println("recover send : "+msg);
        deliver(receiverSelection, deliverId->{
            System.out.println("recover deliver : "+msg.getMiddle()+" : "+deliverId);
            return Pair.of(deliverId, msg.getMiddle());
        });
    }

    public void confirmRecover(Triple<String, String, Long> msg) {
        System.out.println("recover confirm : "+msg);
        confirmDelivery(msg.getRight());
        status = msg.getMiddle();
    }

    public void recoverComplete(RecoveryCompleted recoveryCompleted) {}
```

消息递

```
@Override
public Receive createReceive() {
    return receiveBuilder()
        .match(Triple.class, this::method)
        .match(SaveSnapshotSuccess.class, this::saveSnapSuc)
        .match(UnconfirmedWarning.class, this::unconfirm)
        .build();
}

private void method(Triple<String, String, Long> msg) throws Exception {}
public void status(Triple<String, String, Long> msg) {}
public void send(Triple<String, String, Long> msg) {
    System.out.println("persist send : "+msg);
    persist(msg, e->{
        deliver(receiverSelection, deliverId->{
            System.out.println("deliver : "+msg.getMiddle()+" : "+deliverId);
            return Pair.of(deliverId, msg.getMiddle());
        });
    });
}

public void confirm(Triple<String, String, Long> msg) {
    System.out.println("persist confirm : "+msg);
    persist(msg, e->{
        confirmDelivery(msg.getRight());
        System.out.println("confirm : "+msg);
        status = msg.getMiddle();
    });
}

public void snapshot(Triple<String, String, Long> msg) {}
public void saveSnapSuc(SaveSnapshotSuccess saveSnapshotSuccess) {}
public void unconfirm(UnconfirmedWarning unconfirmedWarning) {}
```



Akka——远程

远远程ac

Act
sel

配

//
ak

}

//
Ac
ac

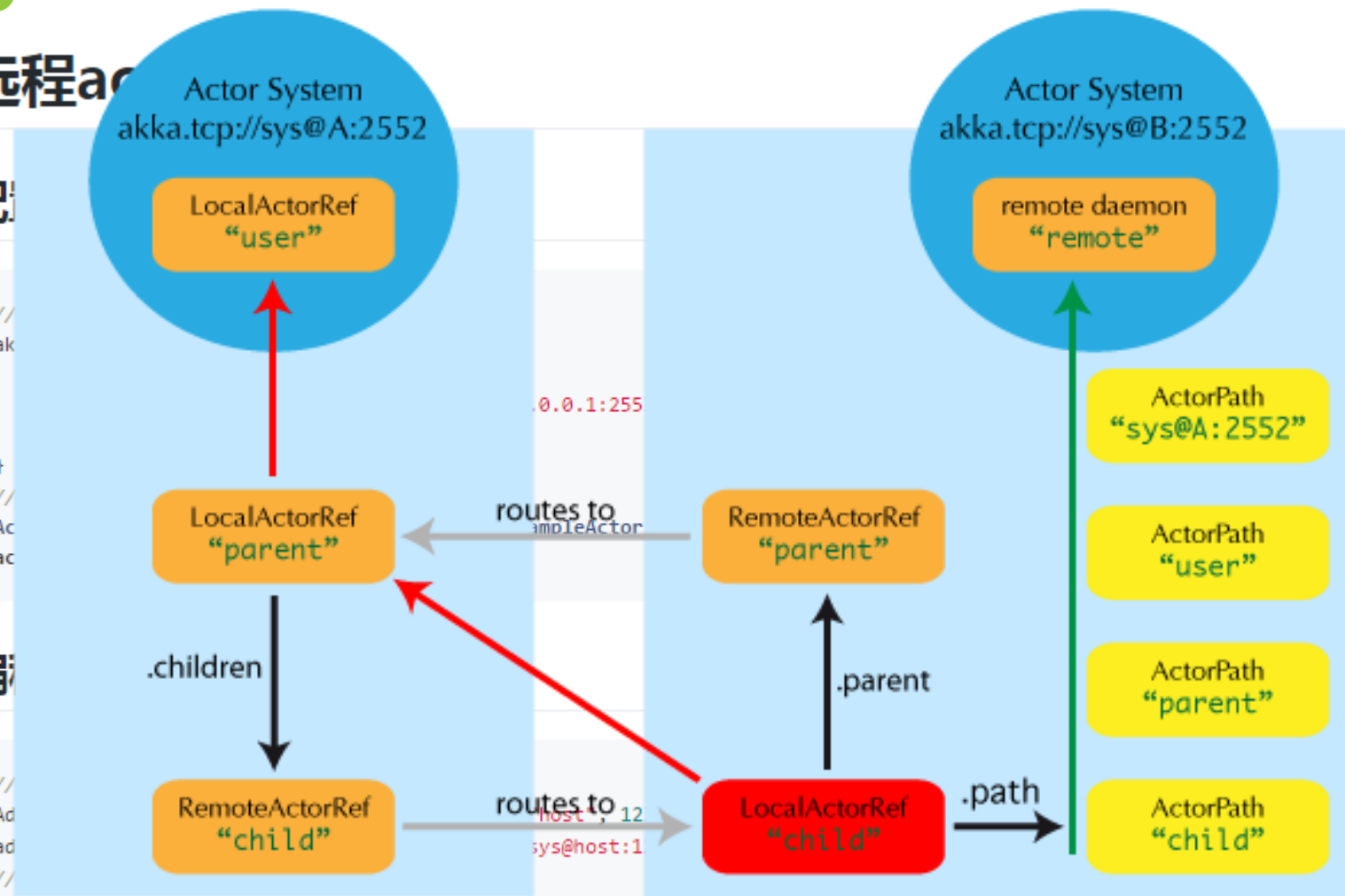
编

//
Ad
ad
//

```
ActorRef ref = system.actorOf(Props.create(SampleActor.class)
```

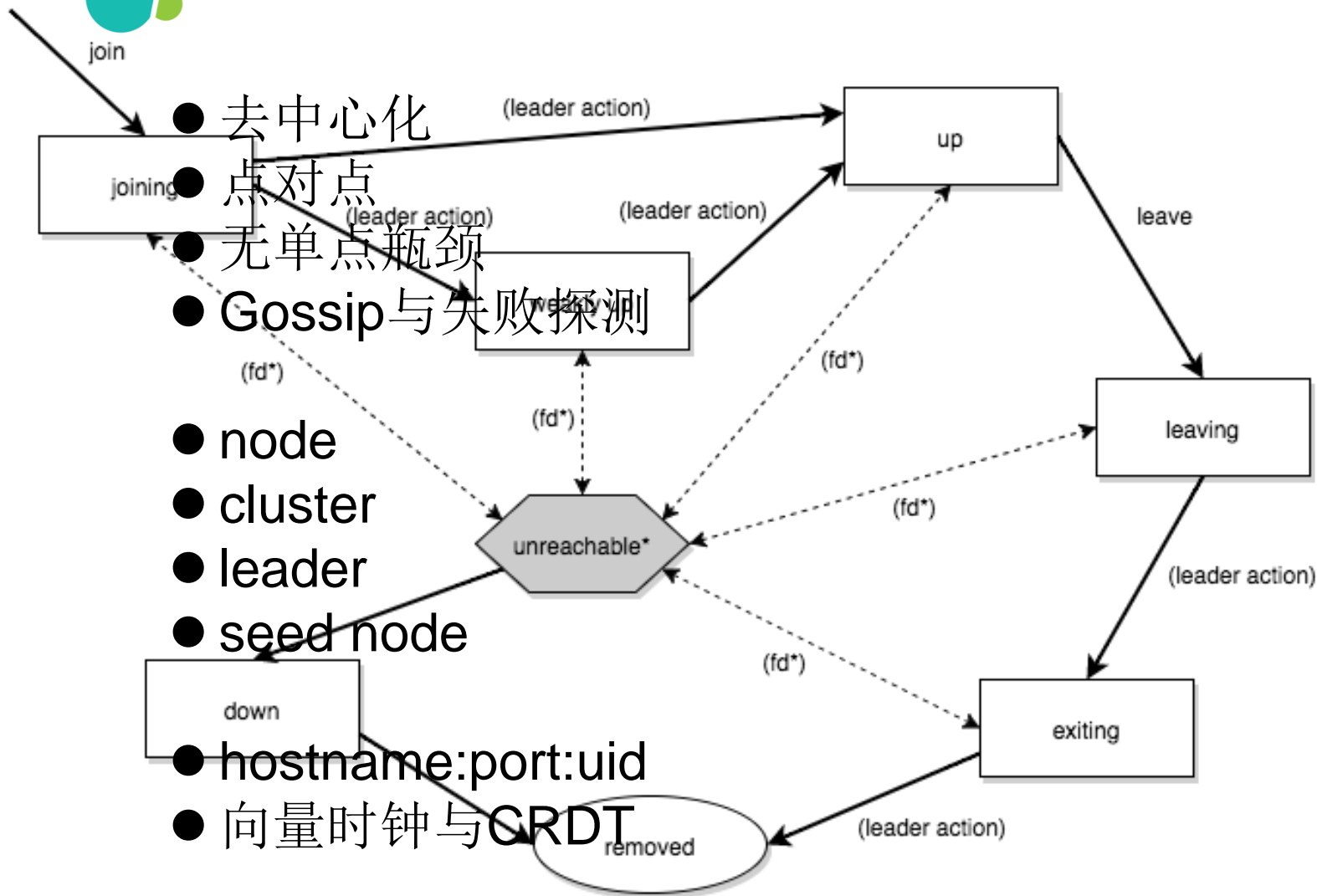
logical actor path: akka.tcp://sys@A:2552/user/parent/child

physical actor path: akka.tcp://sys@B:2552/remote/sys@A:2552/user/parent/child





Akka——集群



操作

- `join` 一个节点加入集群
- `leave` 通知一个节点退出集群
- `down` 标记一个节点已关停

leader操作

leader的责任基本是控制成员状态转换：

- `joining` -> `up`
- `weakly up` -> `up`
- `exiting` -> `removed`

启动集群

```
// 配置方式, 启动system时配置seed-nodes
akka.cluster.Cluster.get(system);
// 编程方式, 无需配置seed-nodes, 但需显式调用joinSeedNodes
cluster.joinSeedNodes(addr)
```



Akka——发布订阅

话题订阅

依赖akka.cluster.pubsub.DistributedPubSubMediator
(本质是个actor) 管理所有注册订阅的actor。

订阅者需要先注册到一个话题上，而后才能收到订阅消息。默认地所有订阅者都会收到消息。

```
// 订阅
// 先获取发布订阅协调器
ActorRef mediator = DistributedPubSub.get(getContext().getSystem()).mediator();
// 订阅, 成功后会收到SubscribeAck消息
mediator.tell(new DistributedPubSubMediator.Subscribe({topic}, getSelf()), getSelf());
// 取消订阅, 成功后会收到UnsubscribeAck消息
mediator.tell(new DistributedPubSubMediator.Unsubscribe({topic}, getSelf()), getSelf());

// 发布
mediator.tell(new DistributedPubSubMediator.Publish("content", out), getSelf());
```

路径订阅

订阅者注册时无需提供话题参数，发布者只需知道订阅者的逻辑路径即可向集群中所有该路径的订阅者发消息。有点类似订阅者的逻辑路径作为话题注册，但默认地只有一个订阅者收到消息。但可以使用 `SendAll` 向所有订阅者发消息。

```
// 订阅
// 先获取发布订阅协调器
ActorRef mediator = DistributedPubSub.get(getContext().getSystem()).mediator();
// 订阅, 订阅的actor在销毁时会自动取消订阅, 也可以使用Remove移除所有指定路径的订阅
mediator.tell(new DistributedPubSubMediator.Put(getSelf()), getSelf());
// 发布
mediator.tell(new DistributedPubSubMediator.Send("/user/{subscriber}", "content"), getSelf());
```



Akka——集群单例

整个集群中，有且最多只有一个Actor运行某个特定的任务。
且能随着集群成员的变动而在各节点间迁移，但与其交互不需要知道其真实所在节点。

发送消息

```
//先创建ClusterSingletonProxy 设置, 支持一些配置如单例actor名, 追踪间隔, 缓存队列长度等
ClusterSingletonProxySettings proxySettings = ClusterSingletonProxySettings.create(nodeSystem);
//定义启动ClusterSingletonProxy的配置, 需要对应的ClusterSingletonManager的路径
Props proxyProps = ClusterSingletonProxy.props("/user/singletonManager", proxySettings);
//启动ClusterSingletonProxy
ActorRef proxyRef = nodeSystem.actorOf(proxyProps, "singletonProxy");
//发消息, 自动递送到了单例actor上
proxyRef.tell("hello", null);

//启动ClusterSingletonManager
nodeSystem.actorOf(managerProps, "singletonManager");
```

em)

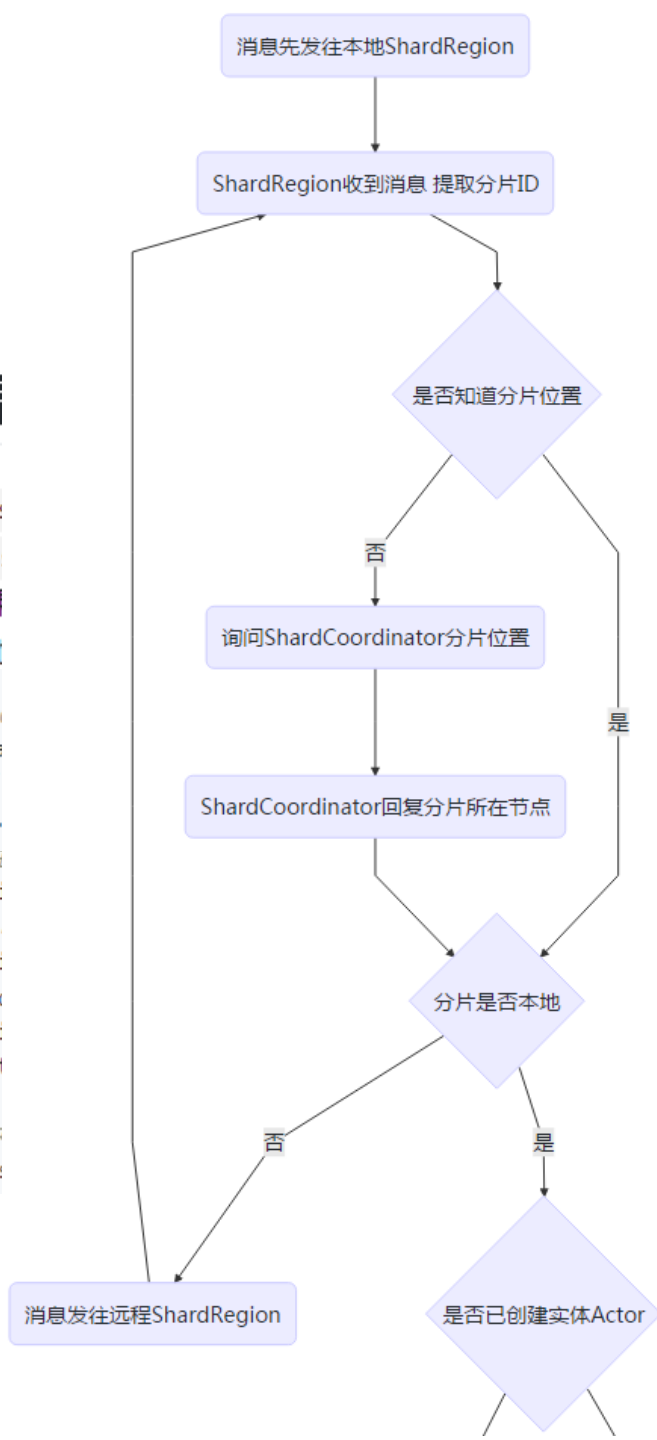
单例重试间隔
时大于auto-down一段时间)



消息分区信

继承 akka.cluster.s
消息转换 entityMes
群节点时。当然如身
的实体数量，谨慎地

```
//先获取shard  
ActorRef sha  
//再发送消息  
shardRegion.  
//启动分片，注  
ClusterShard:  
//启动分片2，  
ClusterShard:  
new Shard  
maxS  
handOffSt  
  
//实体Actor中  
String id =
```

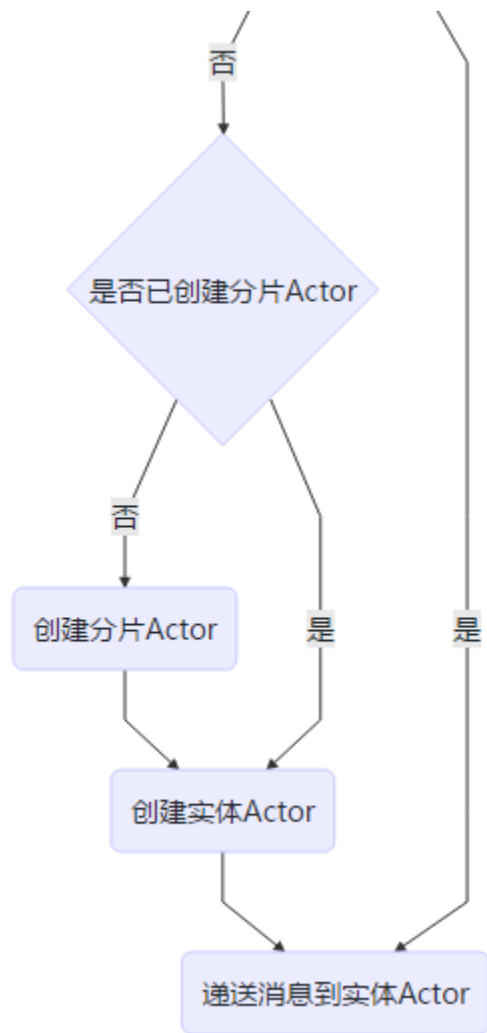


片

往实体actor的消息
且其提取分片ID的代
行的，但要注意必须

```
type}"); /s);
```

分其他类型分片的唯一标识
dingSettings, messageE
dingSettings, messageE
eshold, //集群节点间分



dId(msg)，以及
逐个部署更新集
寸)。要根据预估

数据类型

计数器

GCounter

PNCounter

GCounter 是一个只增计数器，其每个节点都有一个计数器，收敛操作是取最大值。PNCounter 支持增和减，但本质是两个 GCounter，一个用做计正数，一个用作计负数。收敛操作是收敛完两个 GCounter 的值后，正数计数器减负数计数器。

集合

GSet

ORSet

- GSet 是一个只增集合，收敛是各节点的值合并。ORSet 支持增加与移除，但注意移除元素时，移除者必须能看到在集合里包含该元素。

映射

ORMap

ORMultiMap

PNCounterMap

LWWMap

```
// ORMap 要求key可以是任意类型，但值必须是 ReplicatedData，同 ORSet 类似，移除元素者必须要能看到该元素。
// ORMultiMap PNCounterMap 由 PNCounter 组成的映射
// LWWMap 由 LWWRegister 组成的映射
```

标记

Flag

Flag boolean数据类型，收敛操作是置为true成功，也即无法再被置为false

注册器

LWWRegister

LWWRegister 是任何可序列化对象的置位。收敛是时间戳最新的赋值获胜，若多个节点操作时间戳相同，则根据节点地址排序，最小的获胜。注意该数据类型依赖时钟，因此当集群中节点间时钟同步不一致时可能会出问题。

自定义数据类型

继承 AbstractReplicatedData 实现 mergeData()



Akka——脑裂处理、微服务

- Static Quorum
- Keep Majority
- Keep Oldest
- Keep Referee



Akka——微服务

- 同种服务为一个Akka集群
- 不同种服务交互信息可以考虑用Akka HTTP、Akka gRPC、Akka Streams Kafka、Alpakka



教务系统——分布式化的挑战

- 编辑“资源锁”
- 大量集群单例任务
- 前端资源版本一致性
- 脑裂
- 分布式会话



资源锁集群分片



没有则创建



未被占用则标记占用
并返回占用成功消息



被他人占用则
返回失败消息



强制释放

向标记的
WebSocketActor发送强
制释放消息，并销毁

管理课程

2440:四六级QQ群分享

的课表

添加课表

导入课表

知识点绑定

保存课表

课程表 一级标题 二级标题 课时

若课时已创建直播间,则课时无法绑定到其它课时

若课时已绑定到其它课时,则课时无法创建直播间

新创建的课时,无法打开直播间,无法绑定知识点,必须保存一次课表生成课时id后才可以

课时编辑过后,记得保存哦

↑	^	课程表	编辑	新增	批量	删除
↑	▼	QQ群分享课1 (Melissa)	编辑	新增	批量	删除
↑	^	网易分享课	编辑	新增	批量	删除
↑	^	课程简介	编辑	新增	批量	删除
↓		64615 新课时 已过期 跳转链接 过期	编辑	绑定知识点	删除	
↓		64623 已上线测验 测验 已上线	编辑	绑定知识点	删除	
↓		64625 新课时 测验 已上线	编辑	绑定知识点	删除	
↑	▼	直播测试	编辑	新增	批量	删除
↑	^	新课表Tab2	编辑	新增	批量	删除
↓		64619 正在直播 已过期 直播 过期	编辑	直播间	测评	绑定知识点 删除
↓		64627 QQ 加QQ群 已上线	编辑	绑定知识点	删除	
↓		64629 弹窗提示 弹窗提示 已上线	编辑	绑定知识点	删除	
↑	▼	新课表	编辑	新增	批量	删除



教务系统——集群单例任务

```
public abstract class Cancelable implements Runnable {
    ... @Override
    ... public final void run() {
    ... }
    ... protected abstract void cancel();
}

import org.springframework.scheduling.concurrent.ConcurrentTaskScheduler;

fixer = new Thread(job.cancelable, "SingletonFixJob:" + jobId);
fixer.start();

scheduler = new ScheduledThreadPool(1, namedThreadFactory("SingletonScheduleJob:" + jobId));
private void fixJobStop() throws Exception {
    ... job.cancelable.cancel();
    ... fixer.join();
}

import outfox.courseop.common.akka.singletonjob.SingletonJob;
SingletonJob.registerClusterSingletonFixJob(String jobId, Cancelable cancelable)
SingletonJob.registerClusterSingletonScheduleJob(String jobId, String cron, Runnable run)

class AutoPublish extends Cancelable {
    ... public void runJob() {
    ... }
    ... void reRunLogic() {
    ... }
    ... void autoPublish() {
    ... }
    ... public void notifyReWait() {
    ... }
    ... public void cancel() {
    ... }
}

SingletonJob.registerClusterSingletonFixJob(SingletonJob_Course_AutoPublish, autoPublisher = new AutoPublish());
```



```
public static final Key<LWWRegister<Long>> ResVerKey = LWWRegisterKey.create("ResVer");  
  
Cluster cluster = Cluster.get(getContext().getSystem());  
ActorRef replicator = DistributedData.get(getContext().getSystem()).replicator();  
  
@Override  
public void preStart() throws Exception {  
    super.preStart();  
    replicator.tell(new Get<>(ResVerKey, new ReadMajority(Duration.create(1, TimeUnit.SECONDS))), getSelf());  
    replicator.tell(new Subscribe<>(ResVerKey, getSelf()), getSelf());  
}  
  
@Override  
public void postStop() throws Exception {  
    super.postStop();  
    replicator.tell(new Unsubscribe<>(ResVerKey, getSelf()), getSelf());  
}  
  
public Receive createReceive() {  
    private void get(GetSuccess<LWWRegister<Long>> msg) {  
        Consts.ResVer.set(msg.dataValue().getValue());  
        log.debug("初始化前端资源版本为{}", Consts.ResVer);  
    }  
    private void notFound(NotFound<LWWRegister<Long>> msg) {  
        WriteMajority writeMajority = new WriteMajority(Duration.create(1, TimeUnit.SECONDS));  
        Update<LWWRegister<Long>> update = new Update<>(ResVerKey,  
            LWWRegister.create(cluster, 0L), writeMajority, a->a.withValue(cluster, Consts.ResVer.get()));  
        replicator.tell(update, getSelf());  
        log.debug("设置前端资源版本为{}", Consts.ResVer);  
    }  
    private void changed(Changed<LWWRegister<Long>> msg) {  
        Consts.ResVer.set(msg.dataValue().getValue());  
        log.debug("前端资源版本更新为{}", Consts.ResVer);  
    }  
}
```




教务系统——脑裂

```
Set<Member> members = new HashSet<>();  
cluster.state().getMembers().forEach(members::add);  
members.remove(msg.member());  
int memberNum = members.size();  
if(memberNum>=Config.quorum()) return;  
log.error("存活节点数[{}]不足最小集群数[{}],关停服务", memberNum, Config.quorum());  
context().stop(self());  
new Thread(splitBrainResolve, "SplitBrainResolver").start();
```

```
##集群成员,多个以', '分隔  
nodes=ns019:24502,th015:24512,th092:24502
```



教务系统——其他非Akka相关分布式处理

- 分布式Session——Spring Session
- 导出文件与下载



Reference & 几个Demo

- <https://akka.io>
- [Akka笔记](#)
- [分布式CRDT模型、因果一致性与COPS算法](#)
- Applied Akka Patterns: A Hands-On Guide to Designing Distributed Applications
[美] Michael Nash [加] Wade Waldron 著 虞航仲 译
- [AkkaDemo](#)
- [IM](#)



Thank you!

Q&A

TTT问卷



github

