一、前言

本文档将阐述RL项目的服务端整体架构，文章将会说明这套设计的目的，具体的设计细节，以及设计解决哪些问题。这套框架是全服架构的，全服架构的设计中，可以做到哪些事情呢？首先，在SLG游戏中，全服架构的设计，能够使得跨服迁城这种行为变得更加容易。所有玩家逻辑上是在一个服务器的，处于运营需要，会通过地图切分玩家到不同的地图服中。游戏内的所有玩家，除了必须在地图上的行为，其他行为如聊天、收发邮件、送礼物等均可以便捷实现。此外这种设计，对于跨服迁城，实现跨服玩法会更加容易。当然，全服架构的设计，也会面临更多的挑战，因为逻辑上是在一个服，在不断的导量过程中，原有的游戏服可能会不够用，因此需要有强而有力的扩容机制，此外，全服架构下的启服关服流程、热更机制、数据修复需要面临的情况，也比单服滚服的设计更加复杂。

阅读本文需要熟悉skynet的架构，如果你不熟悉skynet，可以先阅读以下两篇文章：
skynet源码赏析
skynet网络机制

二、项目需求

游戏的运营方式，依然是分区分服进行导量，每个服导量数在7k~10k之间。同时，后续要支持转服、跨服加好友、跨服加公会等功能，本质上逻辑上是一个服。我们的架构，既要支持滚服操作，也要支持全球同服的玩法。

三、全服架构所需的技术支持

全服架构需要以下技术支持：

1. 任意两个进程之间，可以进行连接和消息互通。
2. 拥有完善的RPC机制。
3. 具有统一的RPC调用接口，抹平跨节点调用和本地调用的API差异。

当前选用的skynet框架，能够满足这些需求。此外，所有的游戏进程会部署在同一个内网中，跨洲的玩家，通过远程加速代理连接到指定机房。服务器大概率会部署在日本，依据当前的计划，RL项目首先会到欧美进行测试，首次测试建议机房仍然部署在日本，通过加速代理访问日本机房。

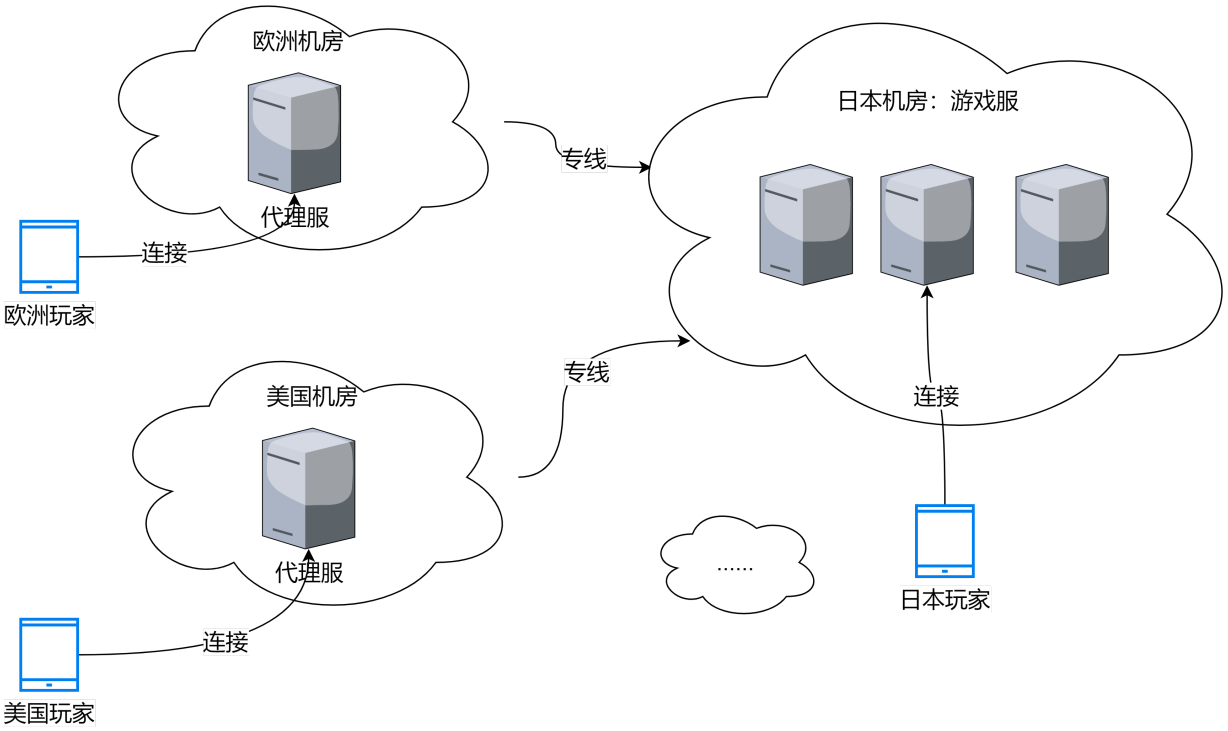

图1

集群部署在同一个机房，能够简化架构设计，并且能够极大降低遇到极端情况的概率。

四、skynet集群机制简介

本节将对skynet的集群机制进行简介。这里将从集群机制监听、接受连接、发送数据包、接收数据包、对内转发和回包几个方面进行论述。本节阐述的skynet集群机制，是基于skynet-1.5版本的。

1、集群机制与监听

skynet进程node1启动之后，如果要使用集群机制，就需要创建clusterd服务，这个服务被创建之后，需要创建一个监听用的socket实例，如图2所示。

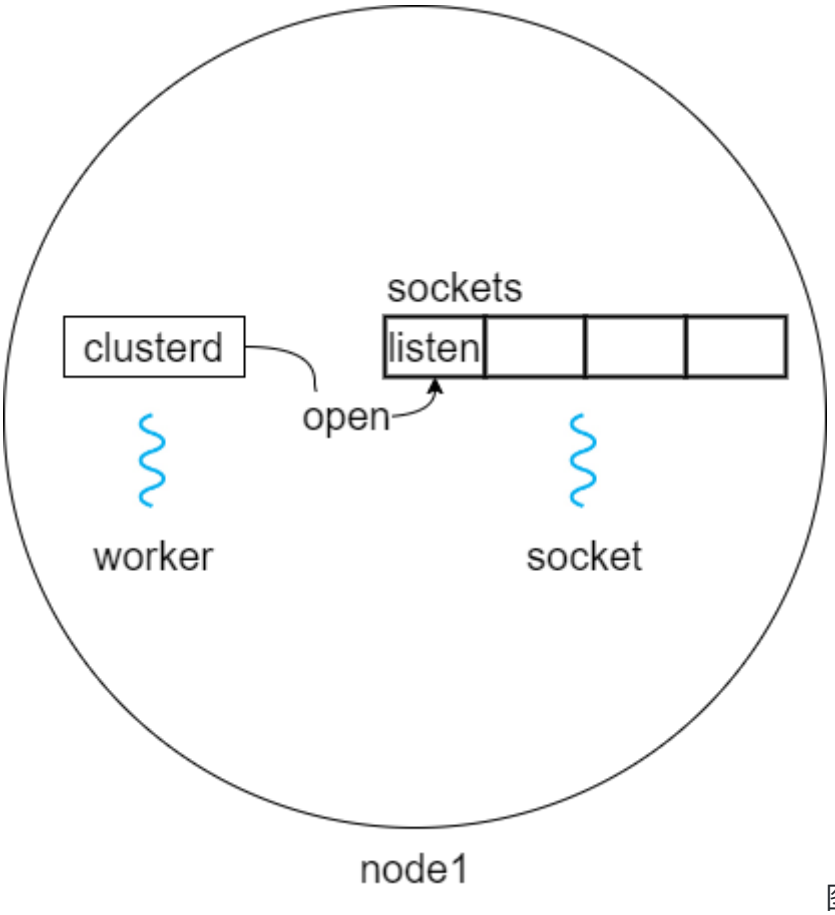

图2

此时另一个进程node2也启动了，并且创建了clusterd服务，且也创建了一个监听的socket实例，此时node2的source_service要向node1的target_service发送请求。首先source_service会向node2进程内的clusterd服务，申请发往node1节点的clustersender服务地址，如果没有这个clustersender服务，clusterd会创建它。

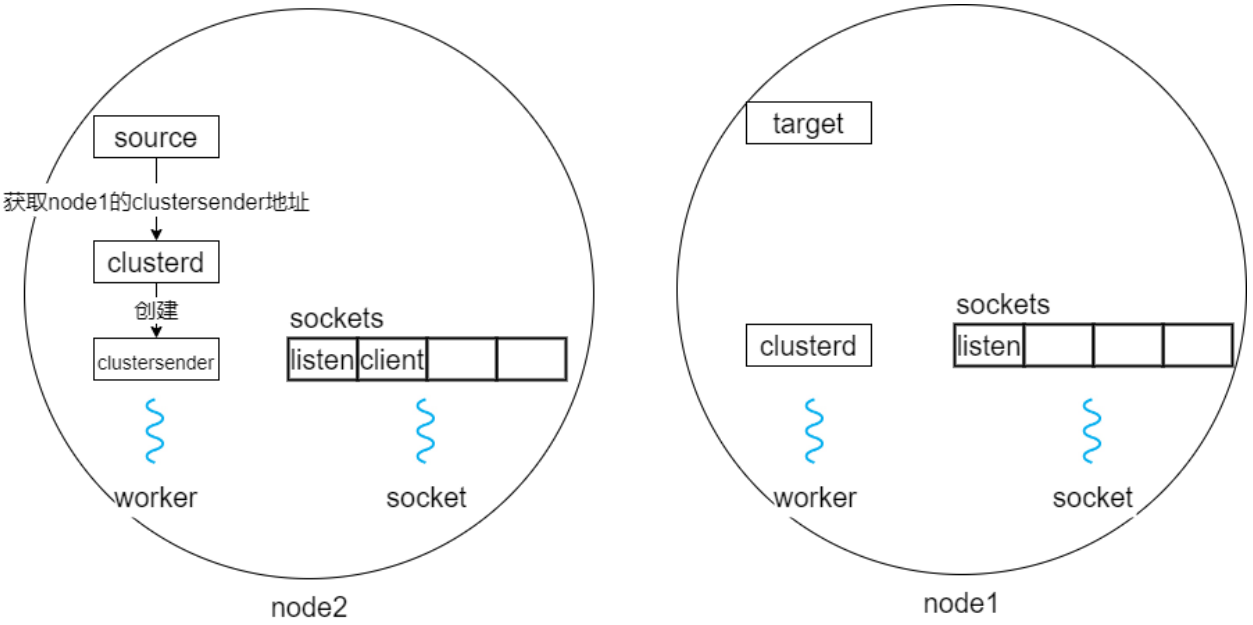

图3

创建完成之后，clusterd服务会将刚创建的clustersender服务的地址返回给source服务，从此刻起，node2进程内，任意一个服务，只要向node1进程内的服务发送任何请求，都会经过这个clustersender服务。访问不同的进程，需要创建不同的clustersender服务。接下来，source服务会向clustersender服务，发送要发往node1进程的target服务的消息。由于此时clustersender服务并没有和node1进程建立tcp连接，因此source发送的请求会缓存在clustersender服务中，此时

clustersender服务会创建一个socket实例，并且向node1进程发起连接请求。

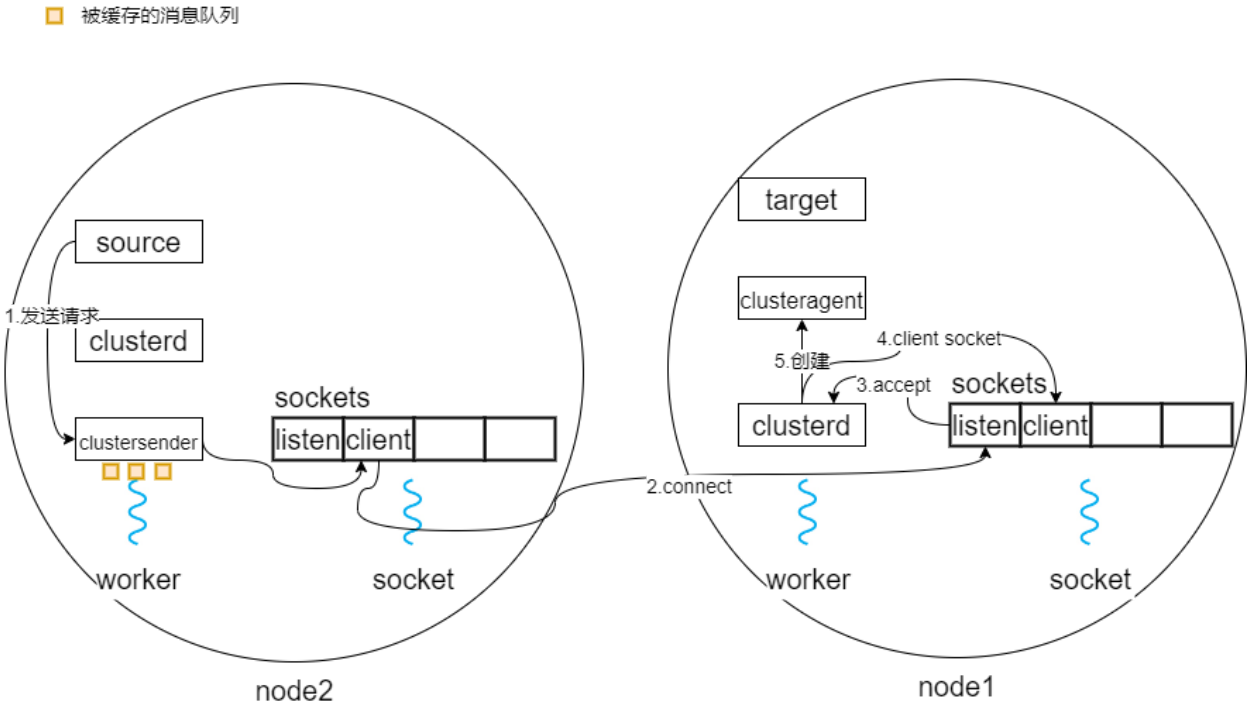

图4

完成连接之后，clustersender服务，会将位于缓存队列中的请求，逐一发送给node1进程，node1进程收到数据包之后，会直接转发给clusteragent服务，clusteragent服务会转发给target服务。

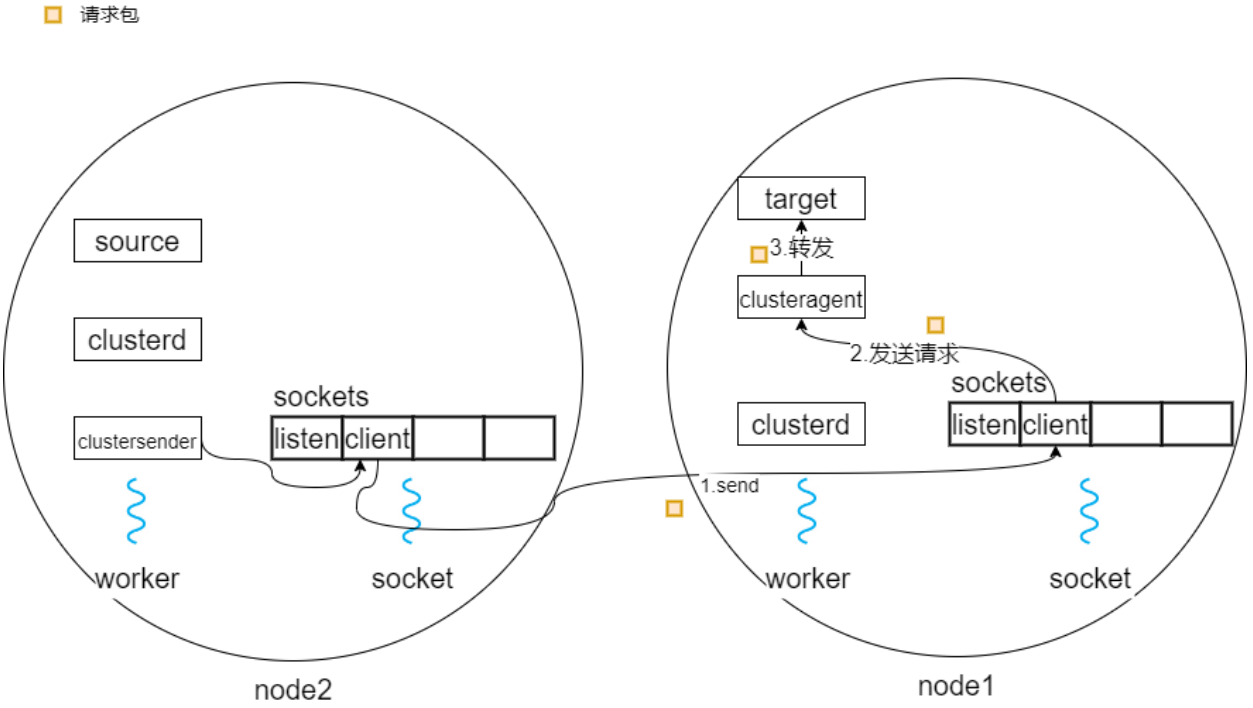

图5

如果node2进程的source服务，调用的是cluster.send接口（也就是push模式），那么流程就直接结束了，如果调用的是cluster.call接口，那么还需要回包给source服务。

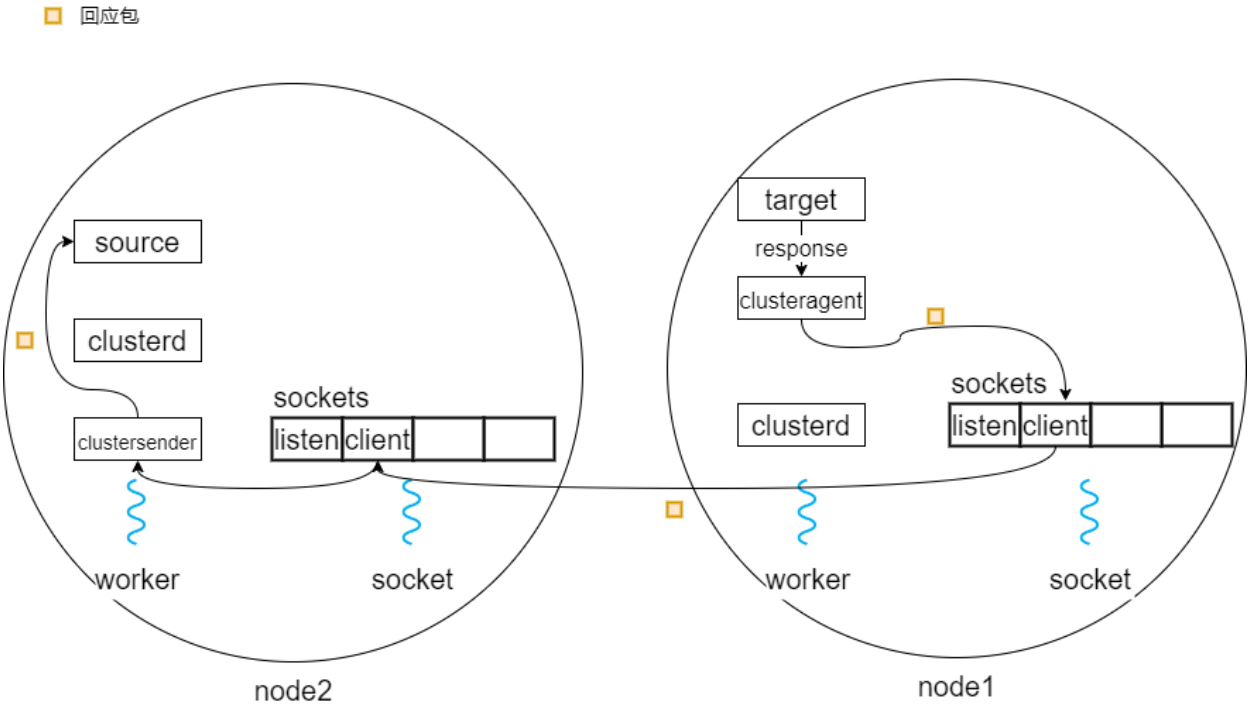
回应包
source
clusterd
clustersender
sockets
listen
client
worker
socket
node2
target
response
clusteragent
clusterd
sockets
listen
client
worker
socket
node1

图6

五、进程与服务类别

RL项目服务端架构，将如下图所示：

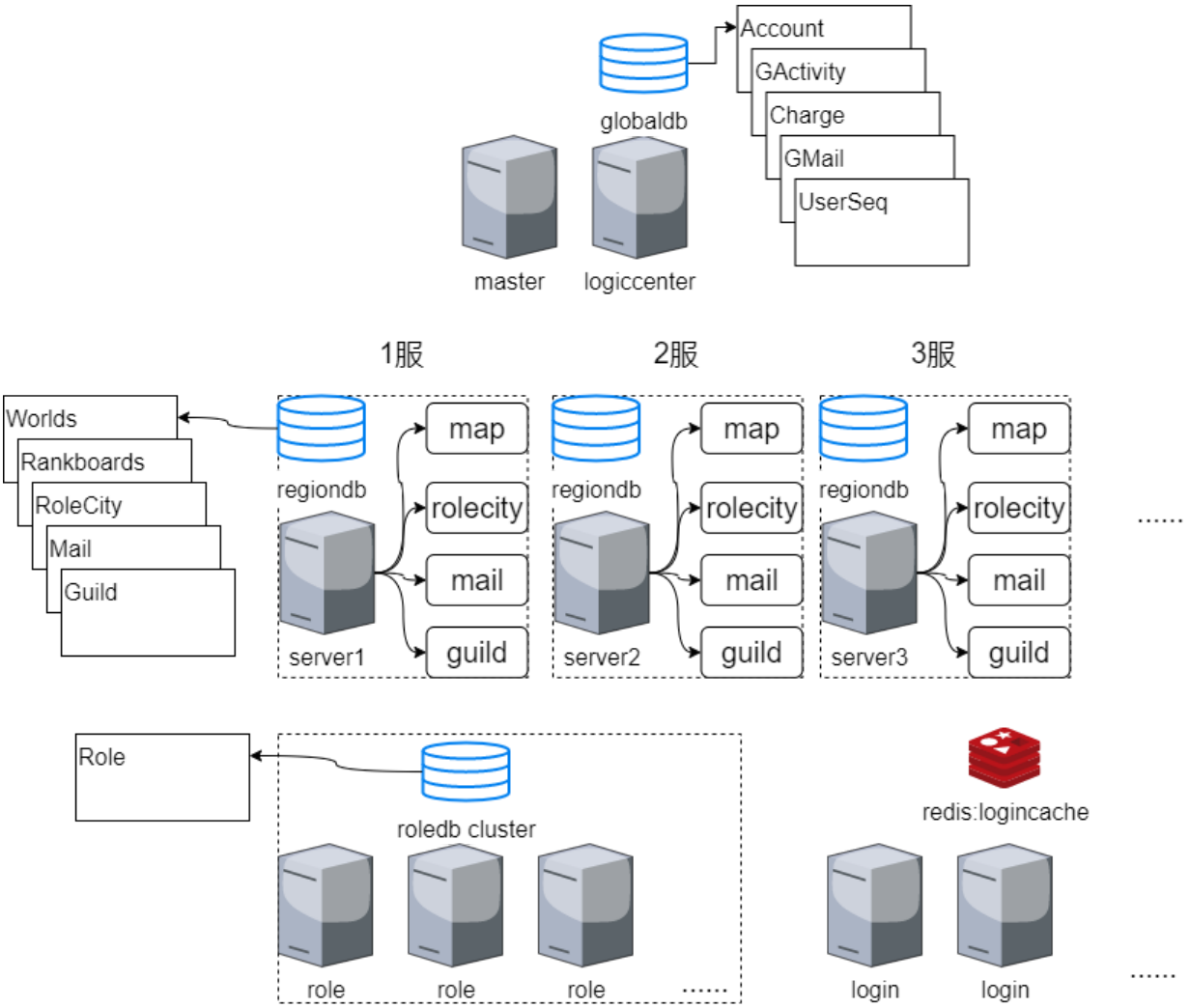

图7

RL项目仍然会以滚服的方式去不断开新服，这里需要注意的是，图7中的role集群是全服共享的，也就是说任何一个role服务器，其内部可能承载来自不同服的玩家。每个区服将部署在同一台机器上，每台区服服务器将会有4个进程，分别是map进程、rolecity进程、mail进程和guild进程。每新开一个服，就会开这4个进程，master进程和logiccenter进程是全局的，它们可以部署在一台服务器中。接下来对每一类进程，及其内部有哪些服务进行说明。

全局相关：

- master进程：中心节点，不负责具体的业务功能，全服的核心进程。它包含的服务有：
  - boot服务：启动服务，负责启动master进程内的其他服务。
  - masterflow服务：启服关服流程控制，监测全服健康状态，处理容灾和扩容事件。
  - httpserver服务：负责接收后台发送的GM指令，并转发到内部进程。
- logiccenter进程：逻辑中心节点，主要负责全局性的业务控制。它包含的服务有：
  - boot服务：启动服务，负责启动logiccenter进程内的其他服务。
  - account服务：登录时，根据传入的openid到数据库中查询对应的userid，如果查不到就创建一个userid。
  - gmail服务：全局邮件服务。
  - gactivitymgr服务：全局活动服务。

区服相关：

- map进程：每个区服的地图进程。
  - boot服务：启动map进程的其他服务。
  - world服务：世界服务，负责地形生成，刷新野怪、资源，行军逻辑，采集资源等功能。
  - aoiscene服务：负责aoi运算的服务。
  - aoibroadcast服务：玩家登录时，会将自己所在的role节点信息，同步到aoibroadcast服务中，当有aoi消息时，直接通过aoibroadcast服务，转发给指定的玩家。
  - rolecache服务：记录玩家在哪个role节点的缓存服务。
  - battle服务：战斗服务。
- rolecity进程：玩家非在线也需要响应的服务。
  - boot服务：启动rolecity进程的其他服务。
  - agent服务：玩家实例所在的服务，一个rolecity进程中，有多个这样的服务。遵循LRU机制。
  - rolerouter服务：所有发给agent服务的消息，首先会发给rolerouter服务，进而转发到内部的agent服务。
  - friend服务：玩家好友系统，数量有多个。
  - friendrouter服务：所有好友相关的消息，首先会发个friendrouter服务，进而转发到具体的friend服务。
  - rolecache服务：记录玩家在哪个role节点的缓存服务。
- mail进程：玩家邮件服务。
  - boot服务：启动mail进程的其他服务。
  - mail服务：玩家实例所在的服务，遵循LRU机制。
  - mailrouter服务：所有发送给玩家的邮件消息，首先会发给mailrouter服务，进而转到到指定的mail服务。
  - rolecache服务：功能同其他进程的rolecache服务。
- guild进程：玩家公会进程。
  - boot服务：启动guild进程的其他服务。
  - guildhandler服务：公会实例所在的服务，遵循LRU机制。
  - guildrouter服务：所有发送给公会的消息，首先会先转发到guildrouter服务，进而转发给guildhandler服务。

role集群：

- role进程：玩家在线实例所在的地方，任何区服的玩家，可以登录到任意一个role进程中。会部署多个，支持动态扩容。
  - boot服务：启动role进程其他服务。
  - gateway服务：实际上是watchdog服务，处理玩家的连接，接收数据包并转给玩家服务。
  - agent服务：玩家在线实例所在的服务，数量有多个，多个玩家实例共享一个agent服务。
- login进程：登录相关，会部署多个。
  - boot服务：启动login进程的其他服务。
  - loginserver服务：相当于role进程的gateway服务。
  - loginhandler服务：会有多个这样的服务，具体处理登录逻辑。

通用服务：

- dbproxy服务：几乎每个进程都会有的服务，负责与数据库交互。
- sharedatad服务：配置共享服务。
- hotfix服务：热更新服务。
- clusterd服务：集群控制服务。
- clusteragent服务：处理集群客户端消息的服务。
- clustersender服务：集群客户端，代理内部发送请求的服务。

六、数据库类别

数据库类别和表设计，在图7中已有展现，这里进行一下大致的说明。首先role集群使用的是mongodb集群，这里会使用到分片的功能。每个服有个regiondb，每个服内的map、rolecity、mail和guild进程会共享这个db。master进程不需要连接db，logiccenter进程则连接globaldb。图7中的矩形表明目前能够想到的数据表的类型。

七、启服关服流程

由于RL项目采用的是全服架构，因此，对于进程启动有比较严格的顺序要求。

1.节点启动顺序
master->logiccenter->map->guild->mail->rolecity->role->login

2.slave节点注册
每个slave进程启动后，在各个服务完成初始化之后，会向master节点的masterflow服务，发送注册消息。之后会定时向master节点发送心跳。每个slave进程，均有一个slaveflow服务，进程内的服务启动之后，如果需要处理关服事件，都会向这个服务注册自己的服务地址。

3.关服流程
role->rolecity->mail->guild->map->logiccenter->master

- 运营/运维人员关闭login节点的登录许可。
- 由后台向master节点发送关服指令。
- master节点的httpserver服务，收到指令后，向masterflow服务转发关服指令，关服流程正式开始。
- 首先要关闭的集群是role集群，关服事件由slaveflow服务来处理。slaveflow服务，会向每个agent服务发送踢所有玩家下线的指令，当所有玩家完成登出处理之后，会告知slaveflow服务，slaveflow服务此时会向master发送，本节点关闭完成的消息。当master节点收到所有role节点发送的关服完毕的指令之后，进入下一个阶段的关服流程。
- master节点此时会向rolecity集群，发送关服指令的消息，rolecity进程的slaveflow此时会收到关服消息，然后向agent服务转发关服指令。agent服务在将所有玩家的数据存入数据库后，会向slaveflow服务告知其完成关服事件，尔后slaveflow服务，会向master进程发送关服完成的消息。master节点在收到所有的rolecity节点的关服完成事件的消息之后，进入下一个阶段。
- master节点，此时会向mail集群发送关服消息，操作流程和rolecity集群类似。完成之后，进入下一个阶段。

- master节点，此时会向guild集群发送关服消息，操作流程和rolecity集群类似。完成之后，进入下一个阶段。
- master节点，此时会向map集群发送关服消息，map节点的slaveflow服务收到消息后，会向world服务发送关服消息，world服务收到关服消息后，将所有的数据存入数据库，并且向slaveflow服务发送关服完毕的消息。slaveflow服务此时会向master节点发送关服事件完成的消息。master节点收到之后，进入关服的下一个阶段。
- master节点，此时会向logiccenter进程发送关服指令，完成之后进入下一个阶段。
- 完成关服流程，此时可以杀进程。

八、玩家登录登出流程

1、登录流程

1. 客户端首先与SDK服务器校验社交账号，并返回openid和token给客户端，如图8所示。

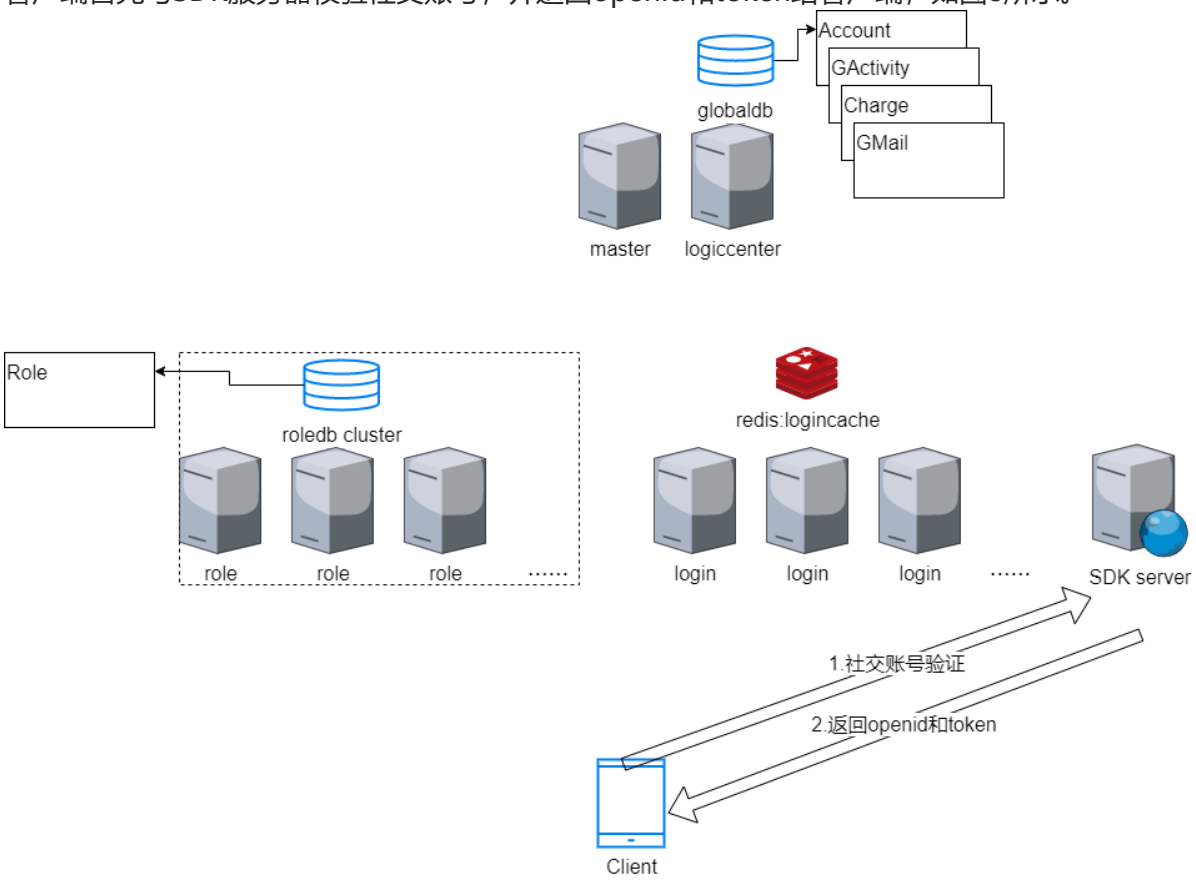

图8

2. 客户端拿到openid和token之后，随机选一个login服务器（后期可以加开一个Ngnix服务器，用来进行login服务器的负载均衡处理），并且通过TCP长连接，连接login服务器。

3. 客户端连接login服务器成功后，发送登录请求协议（login协议），将刚刚获得的openid、token、设备id、os类型、选择的地图服id和版本等信息，传给login进程。login进程所有的网络消息，均会先传给loginserver服务，然后根据openid的值，取模分配到指定的loginhandler服务里，由loginhandler服务进行登录流程处理，如图9所示。

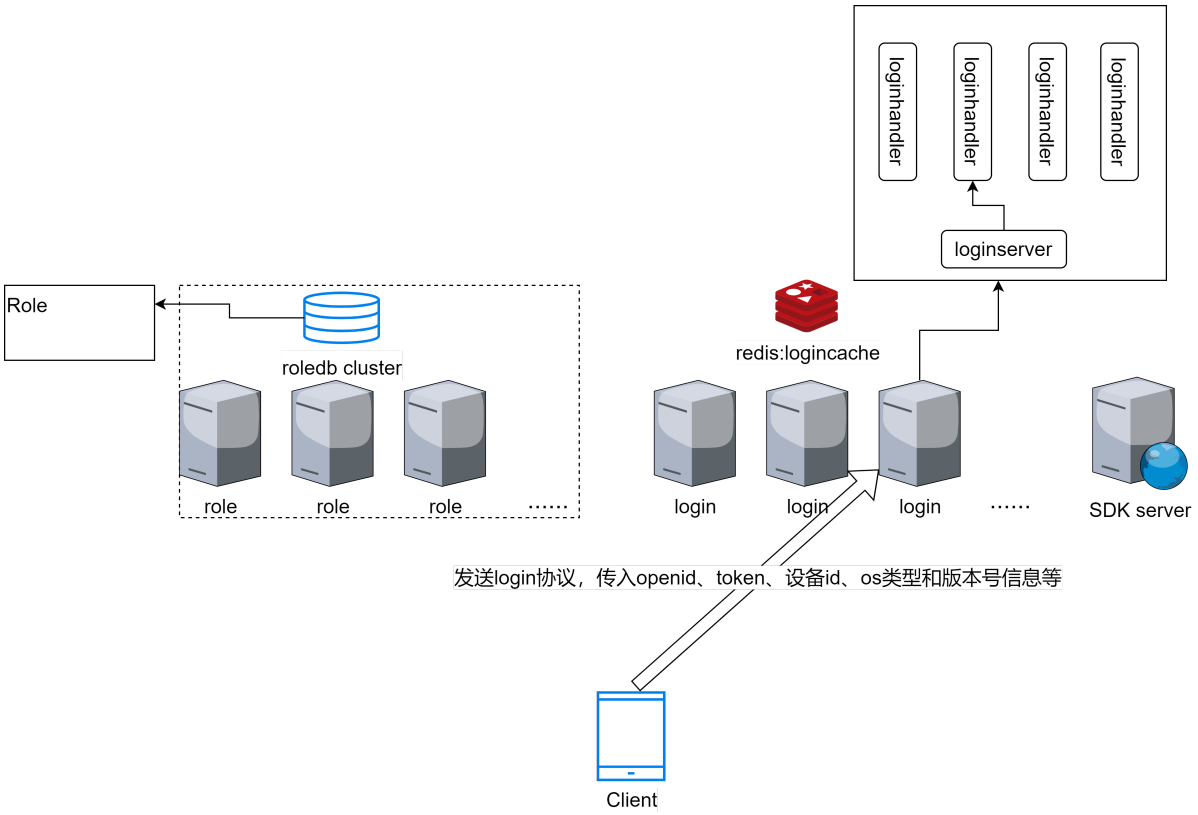

图9

4. loginhandler服务，在收到login事件之后，首先会使用http协议，到SDK服务器校验openid和token，确定客户端是合法的连接，如果成功，则流程继续（如图10所示），否则返回错误码。

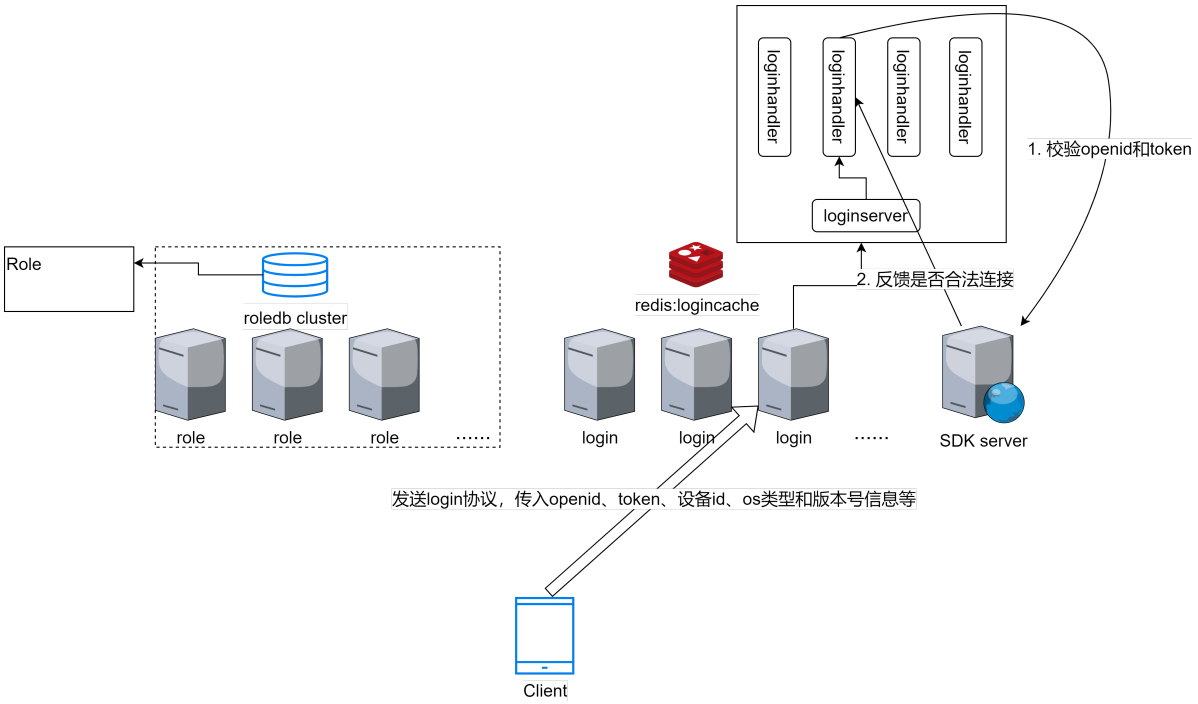

图10

5. 通过校验之后，loginhandler服务，会向logiccenter进程，发送查询账号的消息，查询传入的openid，是否有对应的游戏内账号userid与之对应，如果有，则返回userid（如图11所示），否则进入创号流程（下一节会详解）。此时，需要在redis缓存中，对玩家账号进行加锁，因为登录流程不允许在多个login进程内同时进行，图11中，setnx失败，那么终止流程

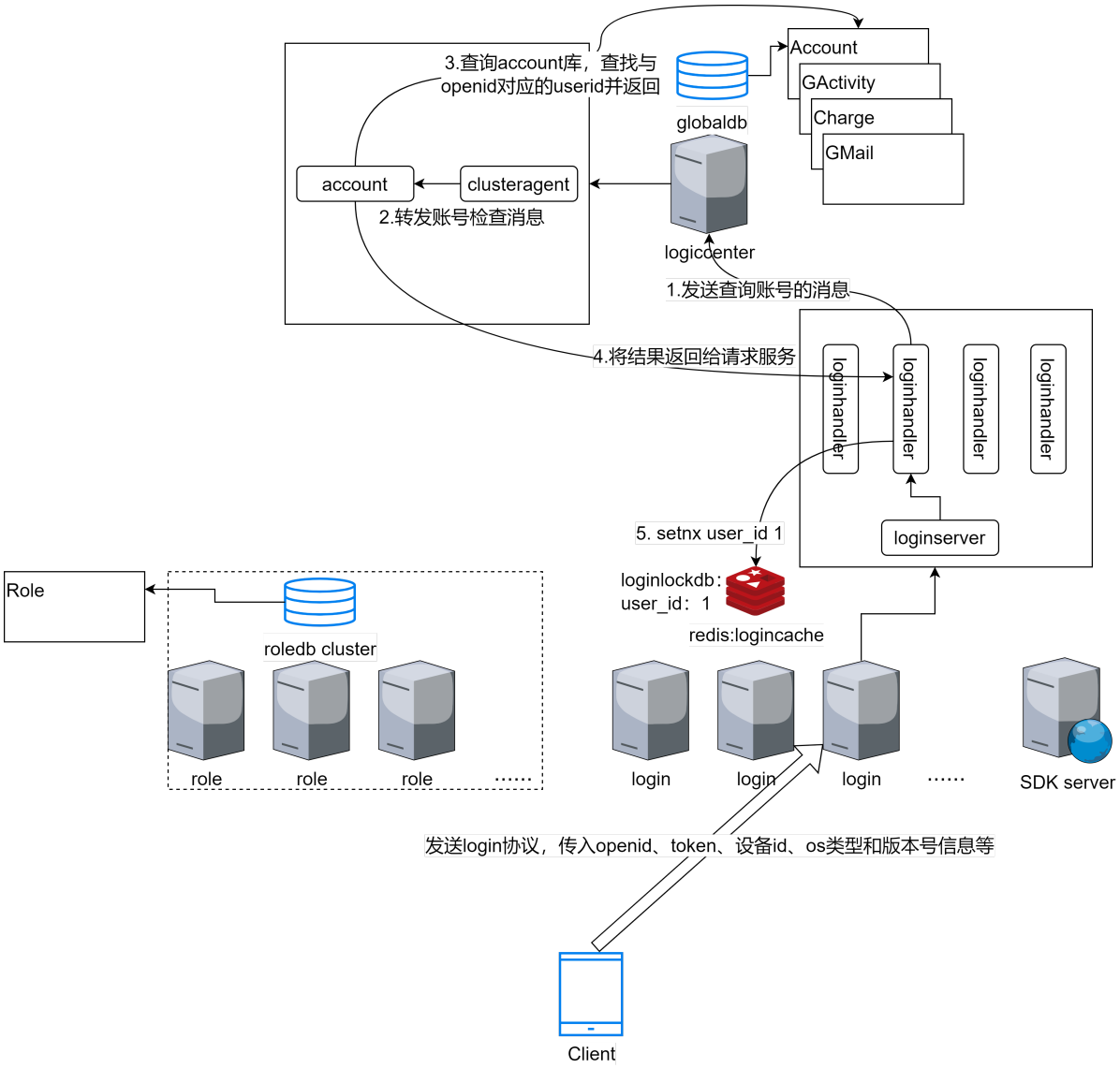

图11

6. loginhandler服务，在获得userid之后，首先要去redis cache中查询，玩家是否在线（是否有缓存信息在redis cache中），如果redis cache中存在玩家登录信息，那么首先要判断cache中的玩家状态。cache中的状态，一共有两种，它们分别是LOGIN_STEP_FINISH和GAME_STEP_FINISH，前一种状态是，login服务器的登录流程走完时，会在redis缓存里设置LOGIN_STEP_FINISH状态，后者是在role进程完成玩家实例加载后，将redis缓存设置成GAME_STEP_FINISH状态。在查询到redis中缓存的结果时，如果发现玩家缓存状态是LOGIN_STEP_FINISH状态，那么将缓存中，玩家所在的role节点信息同步给客户端，登录流程终止，如图12所示。如果发现玩家缓存状态是GAME_STEP_FINISH状态，那么执行踢玩家下线的操作，如图13所示。执行踢下线时，如果role进程内的玩家处于游戏状态，那么正常踢下线后执行后续的登录流程。如果处于登出状态，那么登录流程终止。

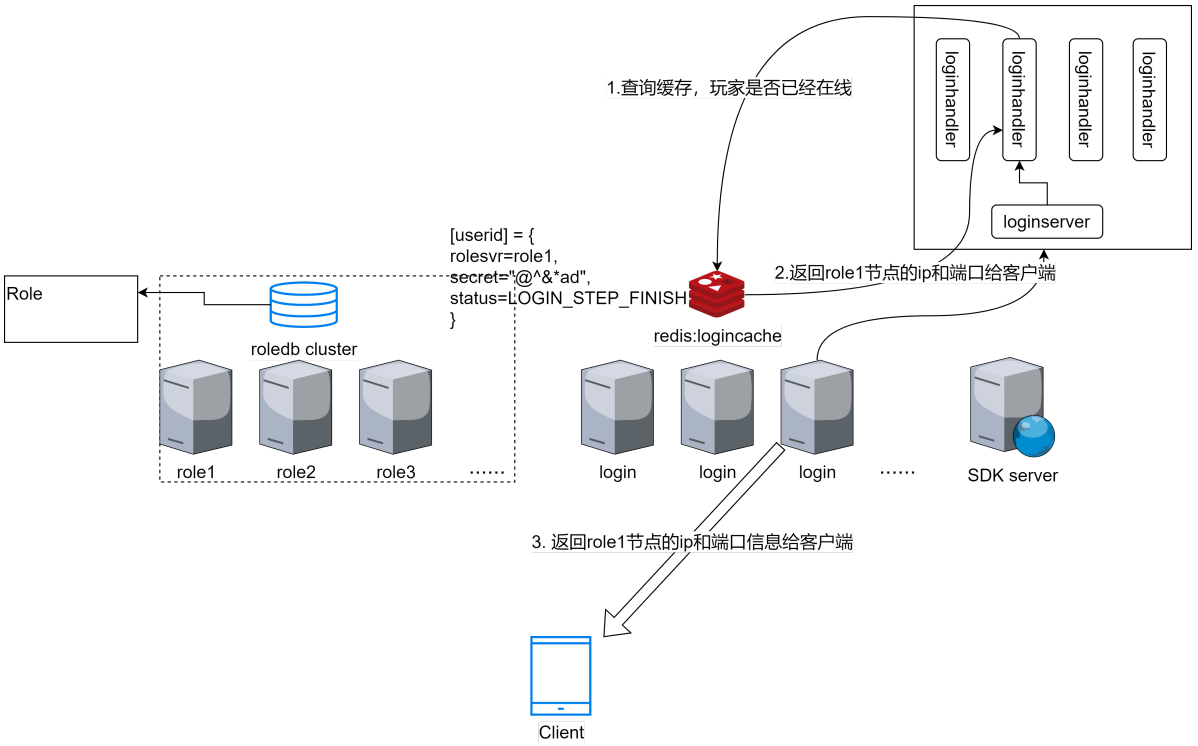

图12

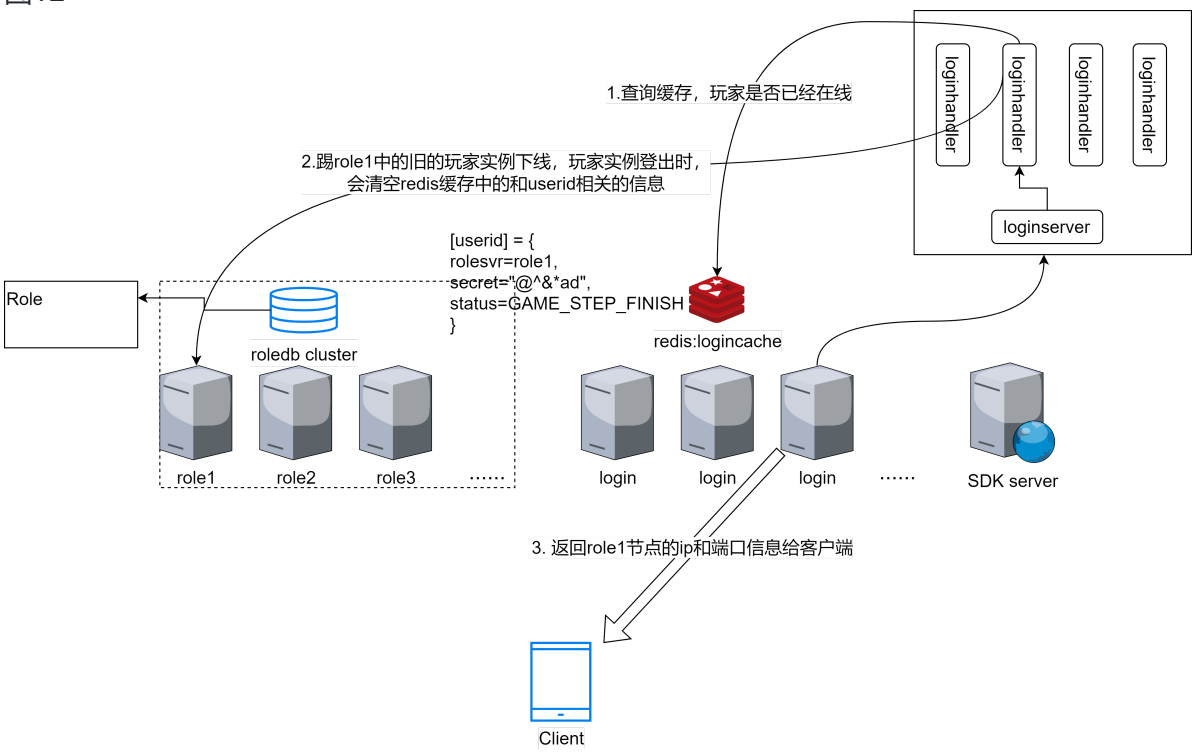

图13

7. loginhandler服务在完成实例踢出流程之后，则进入下一步，选择role服务器中，在线数最少的服，作为当前玩家的登录服。每个服登录人数的信息，会写入redis缓存，redis内嵌一个脚本，从中选取人数最少的role进程，作为玩家游戏的进程，并将该节点的ip和端口返回给客户端，如图14所示。

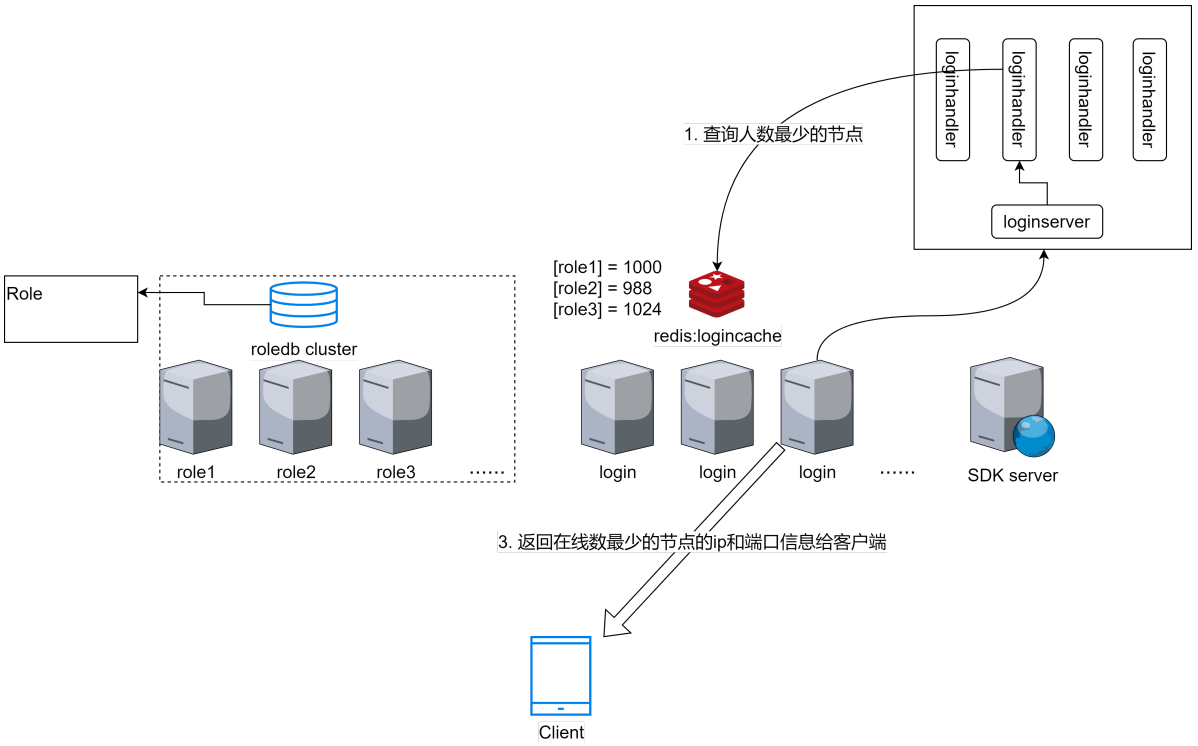

图14

8. 此时login节点的登录流程就结束了，此时清除第5步加的登录锁，并且在redis cache中，写入玩家的登录信息，包括登录role进程的ip和端口，生成的secret等信息，并且将状态修改为LOGIN_STEP_FINISH。

9. 客户端收到要登录的进程ip、端口和secret信息之后，通过tcp连接指定的role进程，连接成功之后，客户端向服务器发送auth协议，用以校验连接是否合法，如图15所示。

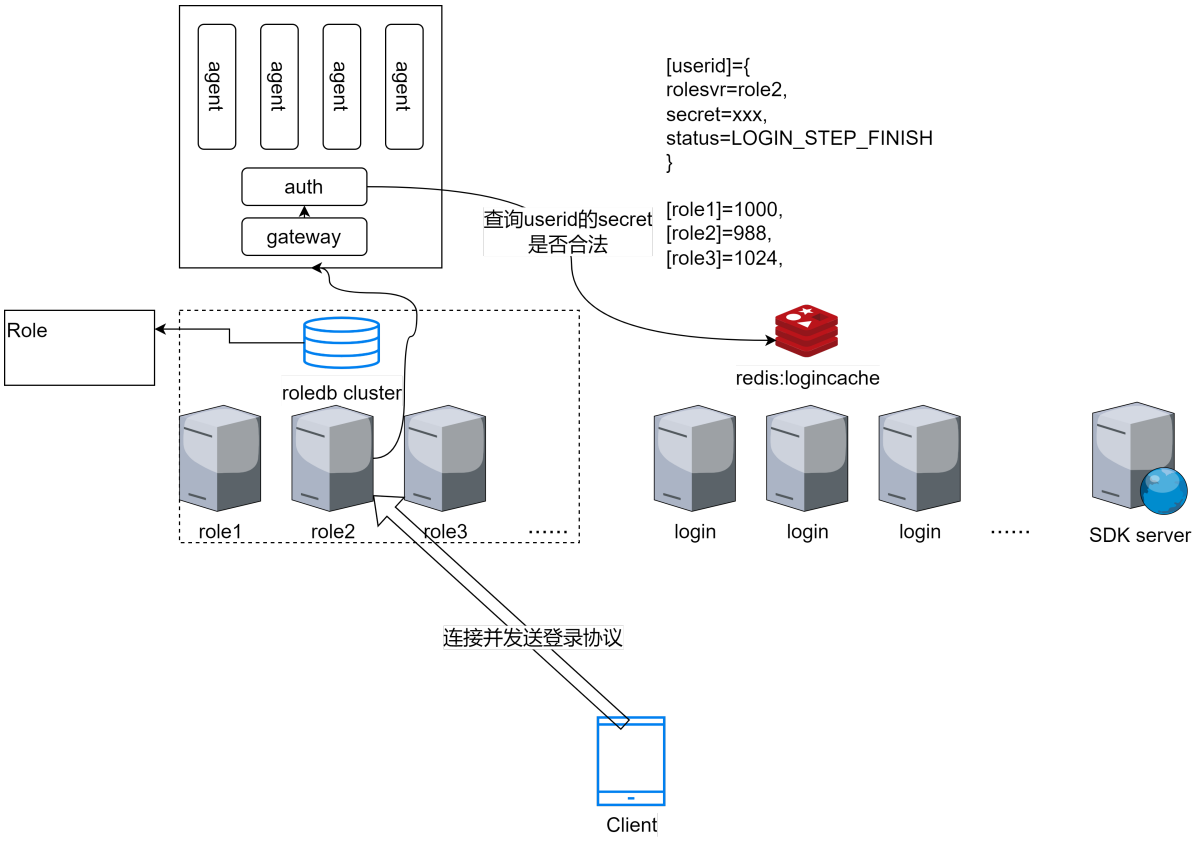

图15

10. 校验成功之后，gateway服务会在内存里加一个标记，在玩家实例被创建出来之前，会“锁住”这段代码，gateway会根据userid取模agent服务的数量，为玩家分配一个agent，玩家实例将在这个agent服务中创建，完成创建后，会设置如下关联：

```
fd2uid[fd] = userid
uid2addr[userid] = agent_addr
```

这里之所以要“锁住”创建玩家的流程，主要是为了避免多次去生成玩家实例，完成玩家实例创建之后，会将redis中的状态修改为GAME_STEP_FINISH。并且role2中的在线数增加1。如图16所示。

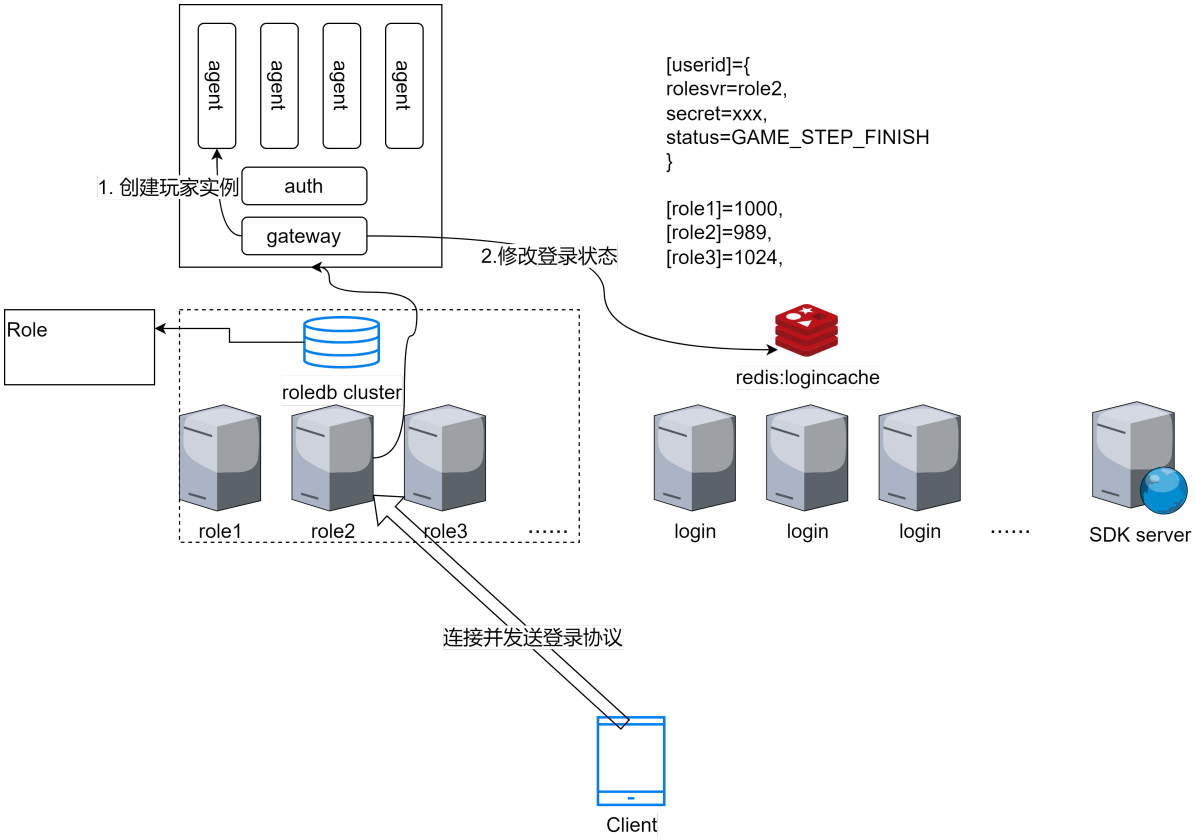

图16

2、登出流程

前面介绍了登录流程，玩家在完成登录后，每隔一段时间（一般时5~15秒）会向服务端发送一个心跳包。当玩家杀客户端进程时，心跳包会停止发送，心跳会有5分钟的超时时间，心跳超时之后，gateway服务会踢玩家实例下线，下线时，玩家实例会等待所有的rpc请求返回（rpc等待的时间为5分钟，超时之后强制下线），避免发出去的请求没有收到结果，完成踢下线处理之后，redis上的对应节点的在线数减去1，玩家缓存被清除。

九、玩家创号流程

上一节中介绍了登录流程，在第5步的时候，查询account表时，如果传入的openid没有对应的userid与之关联，那么就要进入创号流程：

- login进程首先会发送查询账号的请求给logiccenter进程的account服务，account服务会去account表中查询openid对应的userid，如果查不到，会到user_seq表中，获取自增1的的id（类似mysql的auto increment类型），在获得user sequence id之后，结合客户端传上来的区服id，低两个字节作为区服id的存储值，高6个字节用于存储玩家的user sequence id。
- 完成userid的生成之后，将openid和userid的关联写入account表。
- 将userid返回给login进程。

```
// 玩家账号组成
| user_sequence_id | server_id |
|<-----6 bytes---->|<-2 bytes->|
```

十、消息转达流程

在世界同服的游戏里，仍然是按区服，一个一个开服的方式进行运营。不过世界同服的游戏允许玩家转区，允许玩家跨服聊天，跨服发邮件，跨服加公会等行为，所有的服逻辑上是一个服。玩家一开始是在出生的区服里进行游戏，在进行过一段时间之后，才会发生跨服行为。
从技术角度看，如何将消息转达给玩家，玩家的公会，现在将在这里讨论。首先，剖去业务，玩家身上的数据至少包含以下几个属性。

```
{
    userid:xxx,
    mapid:xxx,
    guildid:xxx,
}
```

这几个字段是非常关键的，mapid，决定了玩家具体在哪个区服的地图里，guildid也记录了玩家在哪个区服的公会里。玩家所有发送给map进程的消息，都需要通过一定的规则，找到目标节点的名字，并且将消息发送给指定的进程和服务。比如，在任意进程的任意服务里，某个执行的某个函数，要向目标地图发送消息，则用如下的方式去调用。

```
framework.remote_send(get_map_name(role.mapid), target_service, method, ...)
```

那么消息，就会先发送给本进程的clustersender服务，进而发送给目标进程的clusteragent服务，最后转发到目标服务中，执行指定的函数。
在skynet的集群机制中，任意一个进程，都需要有自己唯一的名称，现在采用的命名规则如下所示：

> 区服中的map、rolecity、mail和guild进程，它们的集群配置名称，一律用进程类型名拼上区服id的方式，配置名称是字符串，例如区服1中的进程名分别是map1、rolecity1、mail1和guild1，以此类推。

发消息给map进程的方式，现在变得很明确了，现在来看看发消息给rolecity的情况。现在假设发送请求方，在任意进程的任意服务中，现在要向指定玩家所在的rolecity进程发送消息，那么该往哪里发呢？首先要明确的是，玩家出生的区服id是编入玩家的userid的，因此不论玩家怎么转服，他所在的rolecity进程是永远固定在自己的出生区服的，因此，要向这个玩家的rolecity进程发送消息，就要

从userid中，取出玩家的区服id，也就是取userid的低16位，找到区服id后，再向目标rolecity进程发送消息，api调用的方式如下所示：

```
framework.remote_send(get_rolecity(role.userid), target_service, method, ...)
```

mail进程和guild进程的情况也类似，这里就不再赘述。
从上一节的登录流程，可以看到，玩家每次登录role节点，都可能不一样，那如何将消息包，转给玩家在线实例呢？前面章节也有提到过，mail进程和rolecity进程内部，均有一个rolecache服务，玩家登录后，会向自己区服的rolecity进程、mail进程，发送userid和role节点信息，这样，任何服务想向玩家在线实例发送消息时，直接向玩家所在区服的rolecity进程的rolecache服务，发送消息，由其转发。邮件的话，直接发送给邮件服务，邮件服务会自己转发。

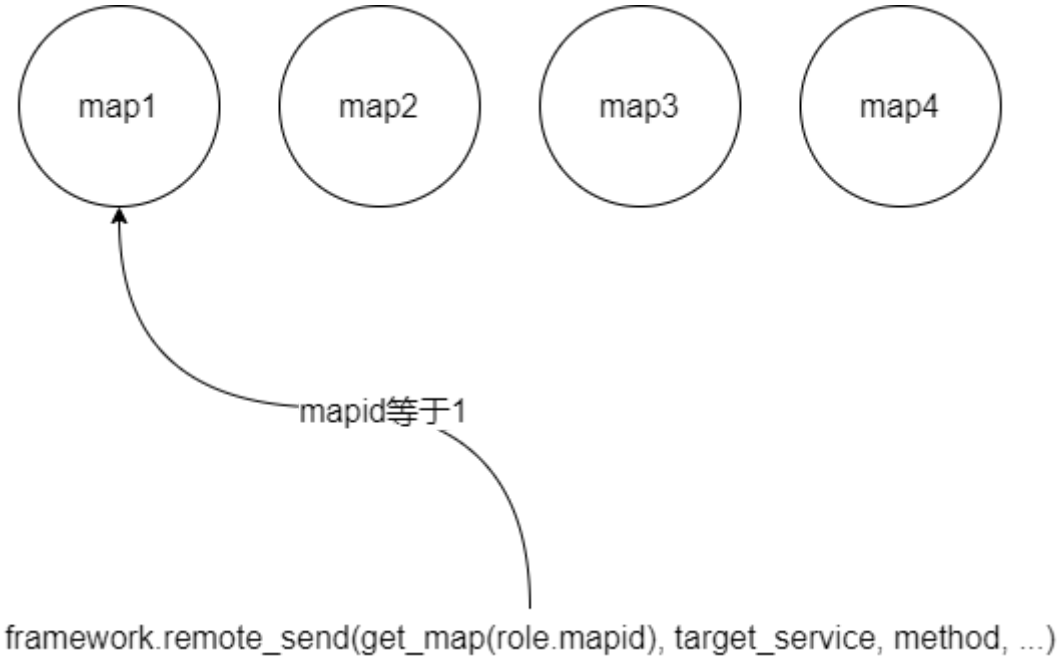

图17

十一、容灾扩容机制

1、扩容机制

在RL项目中，需要扩容的情况主要是两种，一种是正常的开新服，开新服之后，新服的map进程、rolecity进程、mail进程和guild进程的slaveflow服务，会向master进程的masterflow服务发送注册消息，如图18所示。

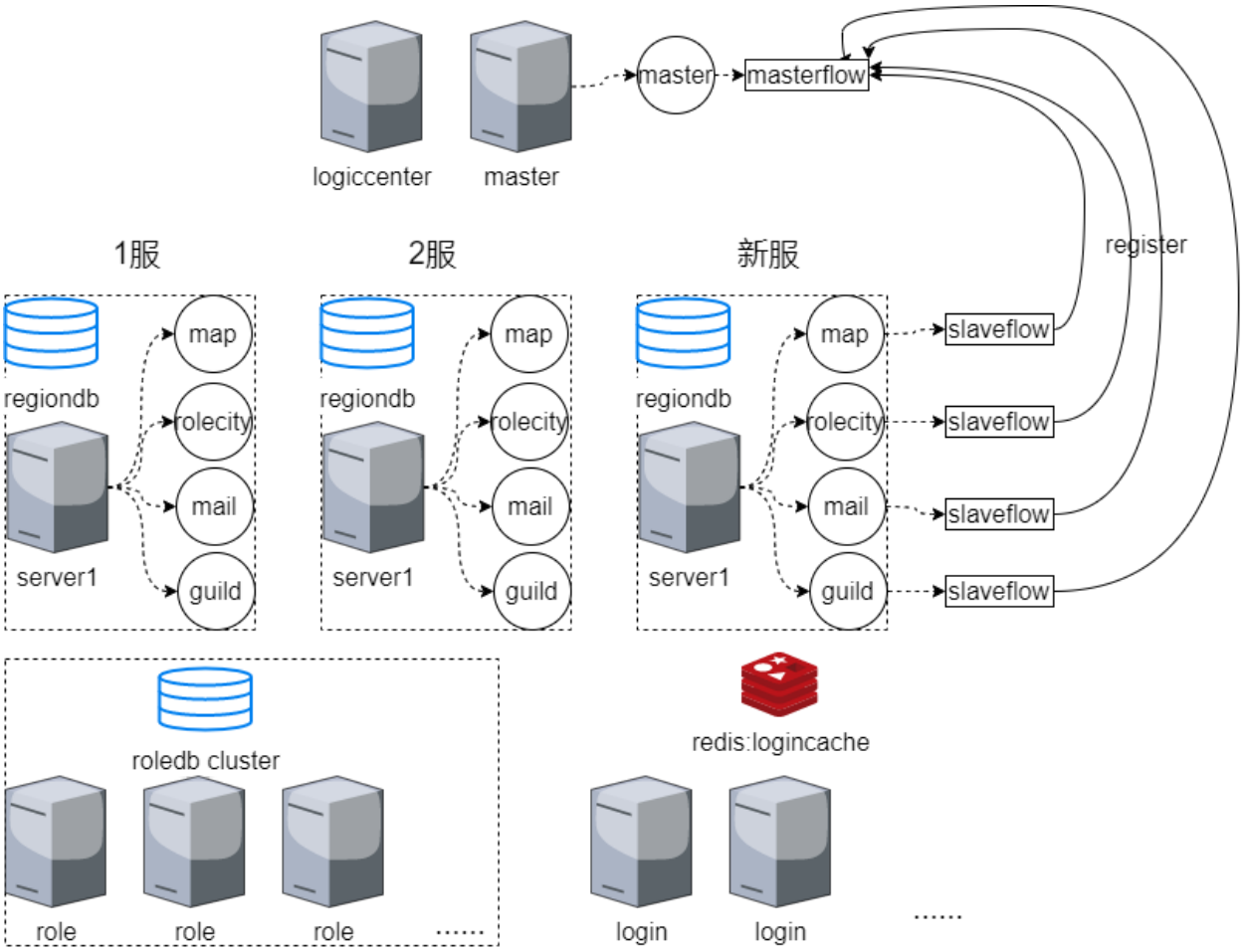

图18

图中的虚线，表示箭头指向的模块，是包含在自己内部的，比如服务器图标箭头指向的进程图标（圆圈），表示该进程是运行在该服务器内的，进程虚线箭头指向的服务（矩形模块），则表示该服务包含在箭头发起方的进程中。实现箭头表示消息的流转方向。在完成注册之后，masterflow服务会触发扩容事件，将新增节点的节点名称，ip和端口信息，广播给所有的已经注册过的节点。
为什么要广播这些信息呢，因为进行远程rpc调用的时候，需要有目标进程的名称，以及通过这个名称去查找其ip和端口，进而建立连接，和收发消息。

由于role集群，是所有区服的玩家共享的，因此随着区服的增多，role服务器可能会遇到性能瓶颈，从而不得不增加新的role服务器，这种就是新的扩容事件。role集群扩容非常简单，新增role服务器，并且启动新的role进程，role进程的slaveflow服务会向master进程的masterflow服务，发送注册消息，而后，masterflow服务会向所有已经注册的进程，广播刚注册的role进程的进程名称，ip和端口，并更新它们的clustername配置。此外，master进程还需要向redis注册新的进程信息。新登录的玩家，将会在新增的role进程登录，进行游戏。

2、容灾机制

容灾机制需要分为两个层面，一个是业务层的容灾，还有一个是框架层的容灾。业务层容灾，需要写业务的人，自己去考虑进程崩溃的情况，这里将介绍的是框架层容灾机制。

全局进程：

- master进程奔溃：
  - 影响：无法进行扩容事件，无法监测到其他进程是否崩溃，GM无法发送，无法进行关服。

    - 恢复办法：重启master进程。
- logiccenter进程崩溃：
    - 影响：全局性业务受影响，比如无法充值，无法注册账号等。
    - 恢复办法：重启logiccenter进程。

区服进程：

- map进程崩溃：
    - 影响：位于该map进程内的，地图相关的功能受到影响。数据会回档到上次落盘的状态。
    - 恢复办法：重启map进程。
- rolecity进程崩溃：
    - 影响：本区服玩家的rolecity相关服务受影响，数据会回档到上次落盘的状态。
    - 恢复办法：重启rolecity进程。
- mail进程崩溃：
    - 影响：本区服玩家的游戏服务受到影响，数据会回档到上次落盘的状态。
    - 恢复办法：重启mail进程。
- guild进程崩溃：
    - 影响：本进程的公会功能受到影响，数据会回档到上次落盘的状态。

role集群：

- role进程崩溃：
    - 影响：位于该role进程的在线玩家，会受到影响，玩家数据会回档到上次落盘的状态。redis中，该role进程的缓存信息（玩家userid到role进程的映射表，role服务的在线数缓存）会被清空。玩家重新登录后，会被分配到其他存活的进程中。
    - 恢复办法：重启role进程，重新注册。
- login进程崩溃：
    - 影响：路由到该节点的玩家，无法进行登录逻辑。
    - 恢复办法：重启login进程。
- redis进程崩溃：
    - 影响：无法登录。redis进程启动是，有个标记变量是is_available变量是nil，master进程会定时检查redis缓存的is_available是否被设置，如果没有，则向所有的role进程发送重新注册玩家在哪个role节点的信息，到redis缓存上，完成之后，role进程会向master进程上报注册完毕，当所有的role进程注册完毕时，master进程会将is_available设置为true。如果is_available变量不为true，则不允许玩家登录。这样做的目的是，当redis进程重启后，master进程会监测到redis缓存中的is_available是nil，因此要对所有的role进程发送重新注册的消息，由于role进程中的玩家还在进行游戏，因此会重新注册自己userid和所在服务器的信息到redis缓存中，注册完之后，上报给master进程，master受到所有的role进程的上报之后，则将is_available设置为true，此时玩家又可以登录了。
    - 恢复办法：重启redis进程。

十二、配置机制

配置使用skynet的sharedata机制，同进程不同服务可以共享一部分内存，从而达到节约内存的目的。

十三、热更机制

热更机制分为配置热更和代码热更。配置热更使用了skynet的sharedata机制，用其内部的更新接口即可。代码热更涉及的内容就比较多了。

1、业务层架构

首先，涉及到的内容是，业务怎么写，RL项目在业务层，将采用类似ECS的架构设计，即Entity Component System架构，ECS架构鼓励数据和逻辑分离，以做到最大程度的功能简化和模块解耦。图19展示了ECS中，Entity Component和System的关系图。

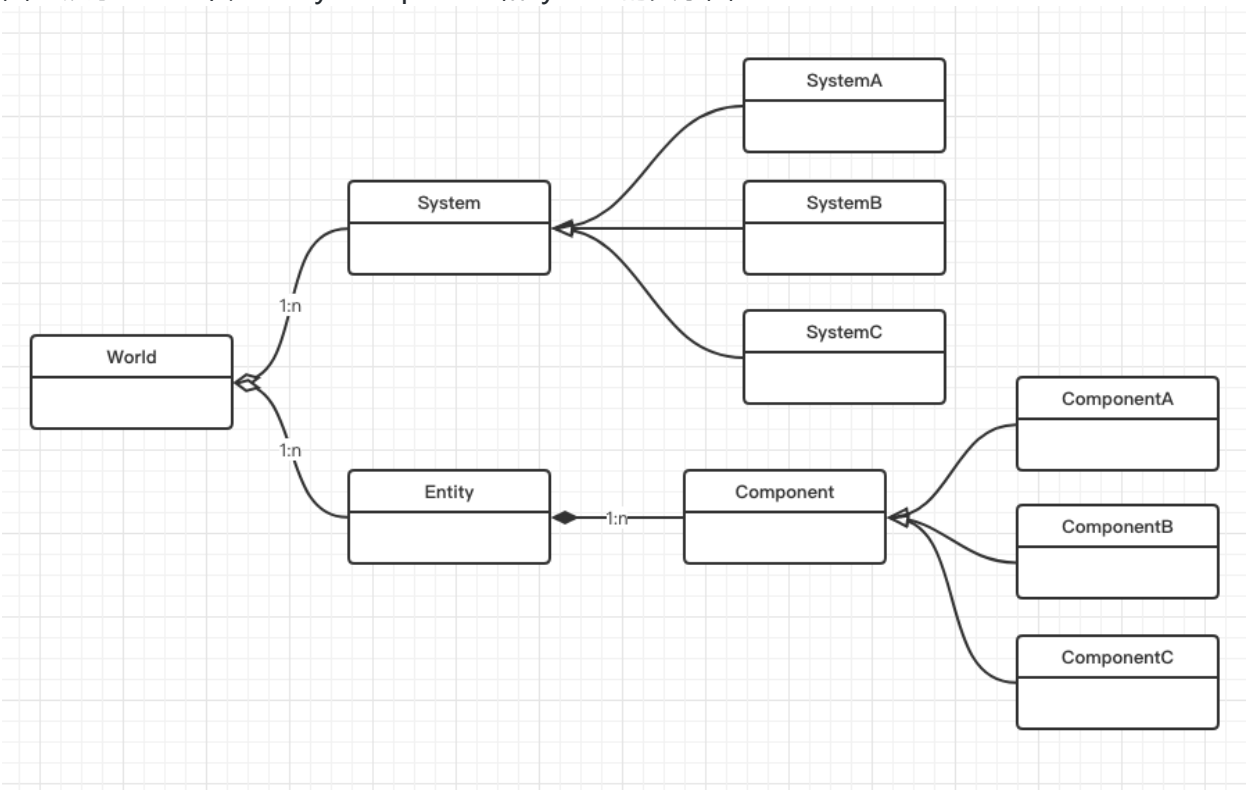

图19

在RL项目中，只有System是类实例，Entity是数据集合，Component则是具体模块功能的数据子集，其结构如下所示：

```
// role entity
{
    gold:integer,
    food:integer,
    oil:integer,
    // other basic attributes
    components: {
        component_a:{ ... },
        component_b:{ ... },
        ...
    }
}
```

2、class模板

在RL项目中，component实质就是系统的数据，一个component数据子集对应一个系统。只有system是类实例，因为存在类实例，所以需要一个class模板，其代码如下所示：

```
local function __inherit_with_copy(base, o)
    o = o or {}
    for k, v in pairs(base) do
        assert(not o[k])
        if not o[k] then
            o[k]=v
        end
    end
    o.__superclass = base
    o.__subclass = nil
    o.__isclass = true
    if not base.__subclass then
        base.__subclass = {}
    end
    table.insert(base.__subclass, o)
    return o
end

local classobject = {
    inherit = __inherit_with_copy,
    __isclass = true,
}

function classobject:new(...)
    local o = {}
    setmetatable(o, { __index = self })
    o:__init__(...)
    return o
end

function classobject:__init__()
end

function classobject:release()
    for k, _ in pairs(self) do
        self[k] = nil
    end
    self.__release = true
    setmetatable(self, {
        __newindex = function(t, k, v)
            assert(false, string.format("attempt to newindex a release obj %s %s", k, v))
        end,
        __index = function(t, k)
            assert(false, string.format("attempt to index a release obj %s", k))
        end,
    })
end

return classobject
```

那么定义一个系统则如下所示：

```
local classobject = require "common.core.class"

local rolesystem = classobject:inherit()

function rolesystem:xxx(entity, ...)
    -- do something
end

return rolesystem
```

3、import函数

热更新有个原则，即是只能重新编译代码，不能重置数据。在ECS架构中，数据和逻辑是分离的，因此热更只需要针对System部分即可，每个System在需要被使用前，均需要调用一个import函数，将其编译和加载到虚拟机内存中。import函数的逻辑大致如下所示：

```
_G.__import_module__ = _G.__import_module__ or {}
local __import_module__ = _G.__import_module__

local function doimport(pathfile)
    local func, err = loadfile(pathfile, "bt")
    if not func then
        print(string.format("ERROR!!!\n%s\n%s", err, debug.traceback()))
        return func, err
    end
    local mod = func()
    __import_module__[pathfile] = mod

    if mod.__init__ then
        mod:__init__()
    end

    framework.send(".hotfix", "register", pathfile, framework.self())
    return mod
end

local function safeimport(pathfile)
    local old = __import_module__[pathfile]
    if old then
        return old
    end
    return doimport(pathfile)
end

function import(pathfile)
    local module, err = safeimport(pathfile)
    assert(module, err)
    return module
end
```

在一个服务中，要使用一个模块系统，则用如下的形式去调用：

```
local rolesystem = import "module.rolesystem"

function xxx(entity, ...)
    rolesystem:yyy(entity, ...)
end
```

import函数和require有点类似，就是没有加载和编译过的模块，首先要加载并编译，放置在一个全局能访问到的表中。import还有一个功能，则是将调用import函数的服务地址，和引用的模块路径，一起发给hotfix服务。这个hotfix服务，会统计每个被引用的模块，被哪些服务使用了，如图20所示。

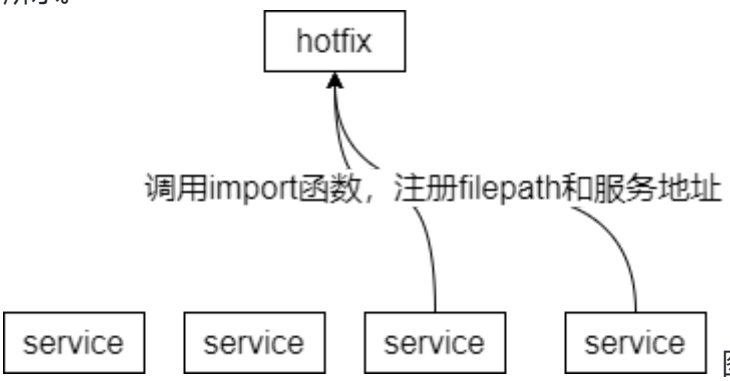

图20

4、热更流程

前面介绍的，是实现热更机制的基础，而现在将开始论述热更流程。首先要介绍一下RL项目的目录结构：

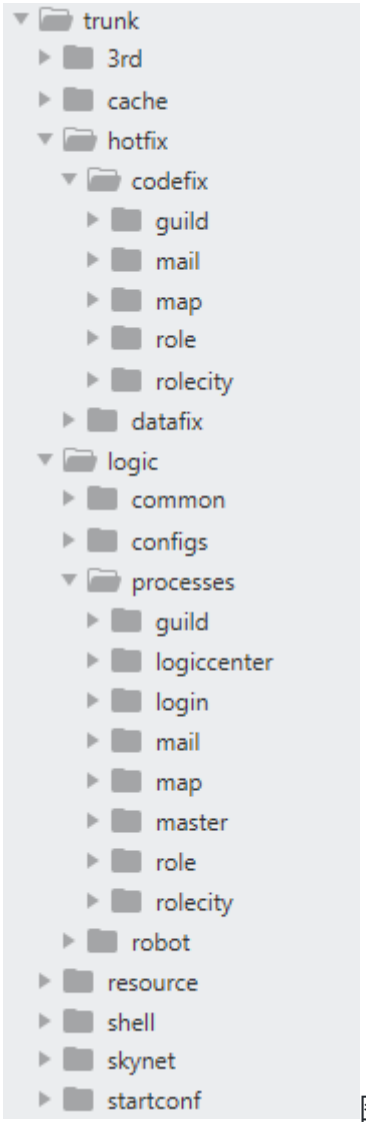

图21

工程中，有个hotfix目录，该目录下有一个datafix目录和codefix目录，前者是放修复数据的代码，后者是放哪些文件需要热更的文件列表。

需要热更的脚本，在本地修改完之后，需要打成热更包，这些热更包只能包含修改过的文件。运维拿到热更包之后，会广播给所有的机器，并解压。此时，hotfix服务，会定时检测hotfix目录下的热更文件目录，位于codefix目录下的文件如下所示：

```
{
    has_load = false,
    filelist = {
        "module.rolesystem.lua",
        "module.buildingsystem.lua",
    }
}
```

has_load字段，表示这个热更列表有没有被加载过，filelist则是需要热更的文件列表。hotfix服务会定时检测这个文件，当has_load为false时，会加载脚本，并且编译，然后遍历filelist列表。然后向所有import过这个脚本的服务发送热更消息，热更新的操作，是在调用import的服务里执行。数据修

复的代码也类似。这里不再赘述。

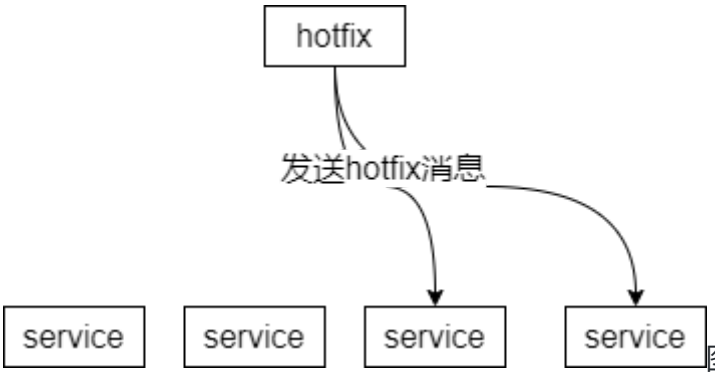

图22

十四、数据热修复机制

数据热修复包括3个层面的内容，分别是：

- 在线实例修复，一般在map进程内的world服务里
- 登录时修复
- 加载实例时修复（LRU机制的服务相关）

修复的Entity数据实例，身上应当带version版本，避免重复执行修复逻辑。

十五、数据存储机制

每隔5分钟定时存储，要存储的数据，会先发送给dbproxy服务，进而同步到mongodb数据库中。dbproxy服务提供增删查改的功能。

十六、协议机制

协议机制，分为两种模式，一种是request-response模式，还有一种则是push模式。协议包体的序列化、反序列化库使用lua-protobuf，使用proto3标准。协议包分为上行和下行两种，request-response模式的response包和push包属于下行包，而request请求包则属于上行包。

```
// 上行包
+--------+--------+----------+----------------+
|2 bytes | 2 bytes| 4 bytes  |   n bytes      |
+--------+--------+----------+----------------+
<-header-><-proto-><-session-><-----body------>
```

上行包中，header记录的是proto+session+body的长度值，最大为65535字节。proto记录的是协议id，session记录的是客户端的请求session值，body则是被protobuf encode的协议数据本身。下行包分为两个钟，一种时response包，一种则是push包，它们的格式如下所示：

```
// 下行包
+--------+--------+----------+----------------+
```

```
|4 bytes | 2 bytes| 4 bytes  |   n bytes      |
+--------+--------+----------+----------------+
<-header-><-proto-><-session-><-----body------>
```

下行包的格式保持一致，header是4个字节，它代表proto+session+body的长度总和。当session为0时，它是push包，当session>0时，它是response包。协议编辑如下所示：

```
// request-response模式
messgae Login {
    ...
    message Response {
        ...
    }
}

// push模式
message XXXNotify {
    ...
}
```

十七、监控机制

监控机制，还将继续使用prometheus+grafana组合的方式，需要监控的信息，包括机器的，功能框架的和业务自定义的。

十八、合服策略

合服需要提供几个功能：

- 保证玩家id全服唯一。
- 提供合并数据库的脚本。
- 玩家重名修改机制。
- 地图重新分配机制。
- 合并之后，由于玩家id不变，因此玩家根据玩家id到区服的映射需要改变。