通用游戏服务器框架设计

应对高并发场景的有状态服务动态扩缩容解决方案

钱斌海

Funplus技术中台
游戏服务器技术专家

目录

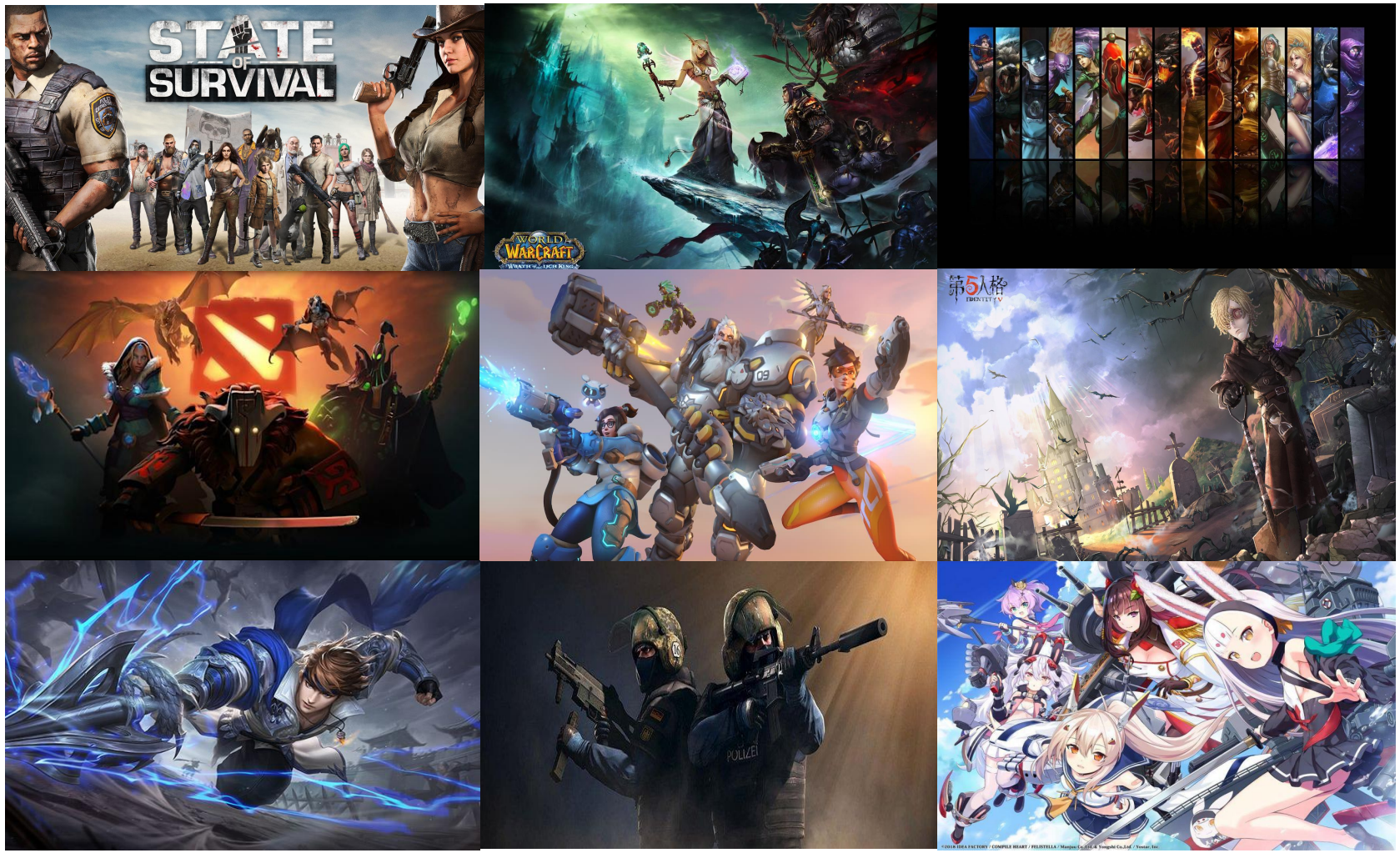
第一部分

游戏服务器架构演进

游戏服务器开发简介

- 游戏服务器研发包含哪些内容?

- SLG、MMO、MOBA、FPS … …



以上游戏图片素材均来源于线上图库，这里仅用来举例丰富的游戏内容，并无特殊指代

游戏服务器开发简介

- **服务器引擎**
  - 网络库、RPC框架、编程范式 (Actor模型、基于Lua实现继承体系) 规约上层业务开发
- **核心玩法**
  - 通过复杂的战斗，验证游戏可玩性
- **外围系统**
  - 好友、聊天、工会，丰富游戏生态
- **工具链**
  - Stress Test、CPU Profile、Opentracing、Prometheus、Chaos Engineering

稳定的服务器引擎、丰富的外围系统设计、工具链的准备，最终都是为了呈现一个好的核心玩法
而且整个服务器的架构，为了适配不同类型的核心玩法，也会不断地迭代和演进

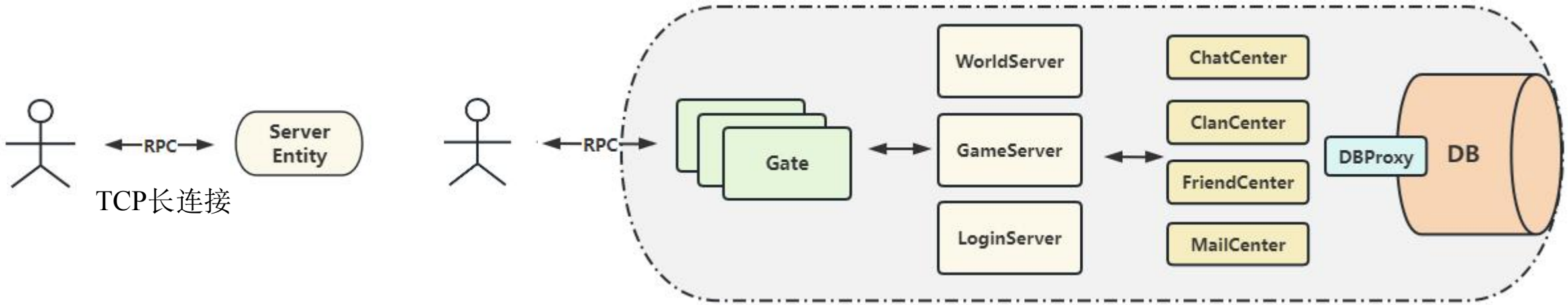
游戏服务器开发简介

- ➢ **分区服的设计**
- ➢ **核心玩法**
- 地图探索、任务
- 团队副本 PVE
- 玩家对战 PVP

以上游戏图片素材均来源于线上图库，这里仅用来举例，并无特殊指代

分区服游戏服务器框架

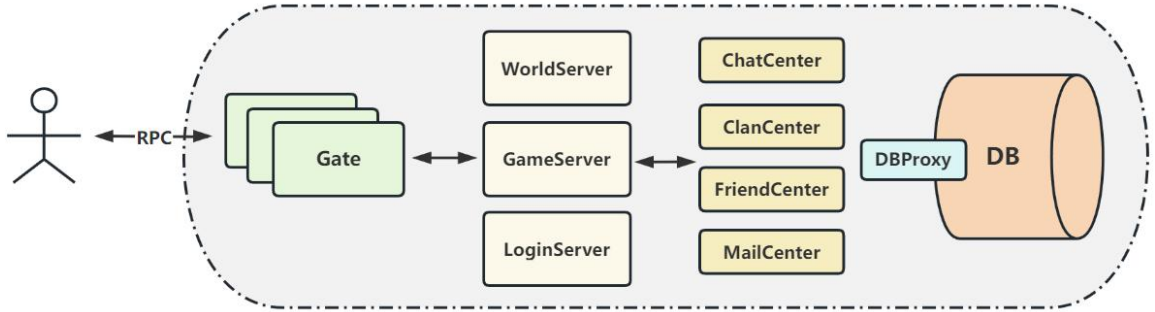
- 玩家在线期间，服务器一直存在一个实体，能实时响应玩家操作，也可以称为玩家的内存状态



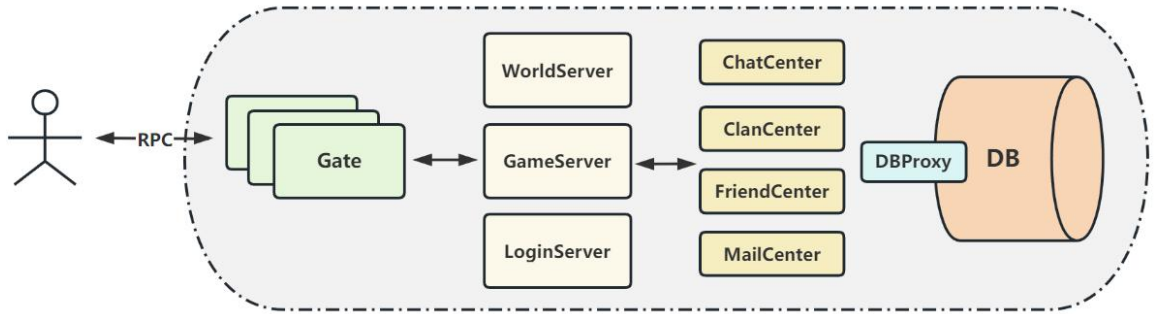
- 区服的负载容易评估
- 通过单点Center简化分布式逻辑
- 开新服承载新增玩家
- 区服隔离，单服资源有限，玩法受限

分区服和跨服的架构演进

- 丰富游戏可玩性，在单服的玩法上，扩展出跨服玩法，比如跨服竞技场、排行榜

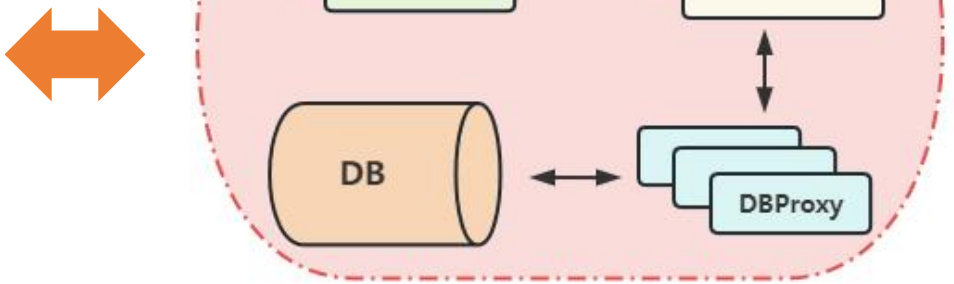


1区

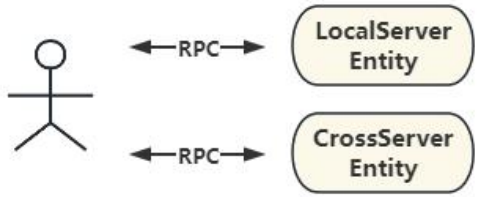


2区

• • •



跨服公共服务器



降低网络延时

全服的游戏服务器架构

- 全服设计
- 核心玩法

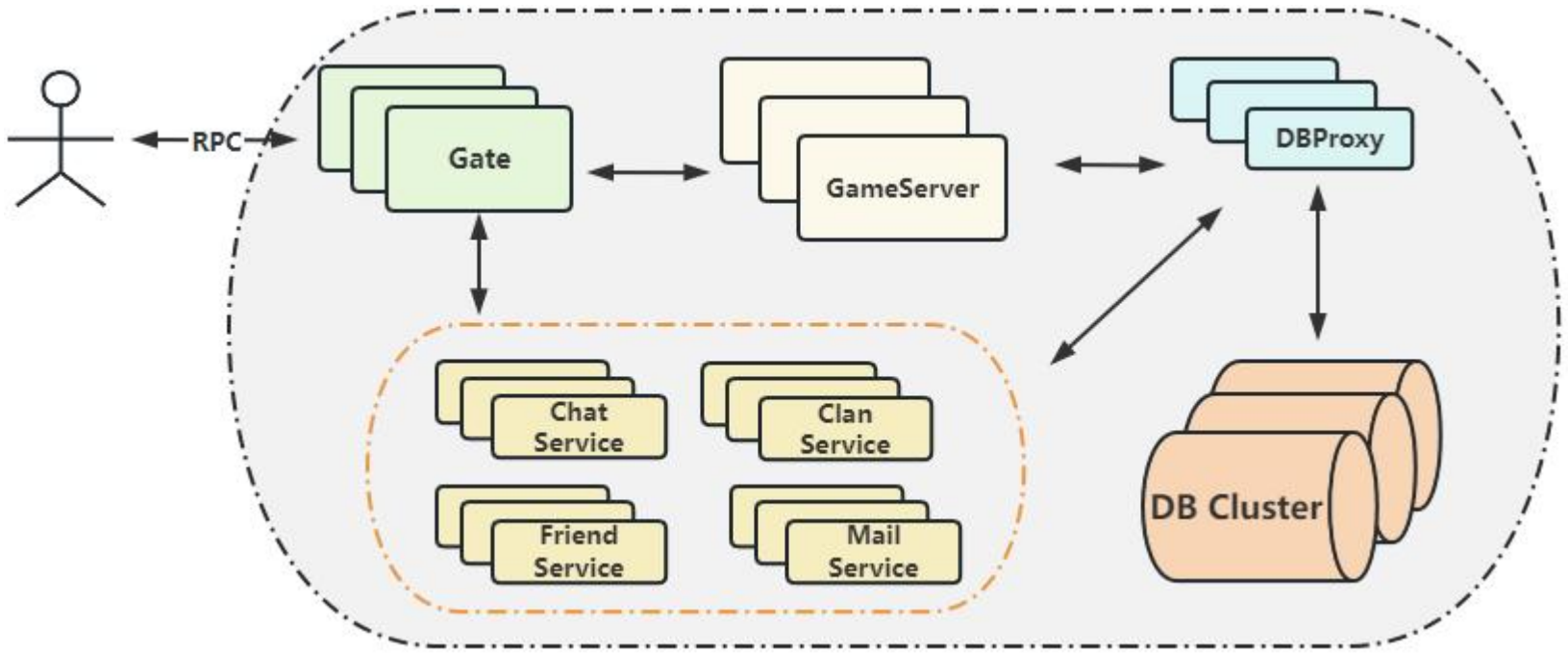
- 5V5公平竞技
- 1V4非对称竞技

组队 => 匹配 => 进战斗

大DAU => 同时在线人数达百万 => 分布式、负载均衡

以上游戏图片素材均来源于线上图库，这里仅用来举例，并无特殊指代

全服的游戏服务器架构



- 单点设计成为系统瓶颈，分布式多节点
- 外围系统封装成微服务
  - 独立部署和扩缩容
  - 类似插件，即插即用

第二部分

Phonest通用服务器框架

Phonest通用服务器框架

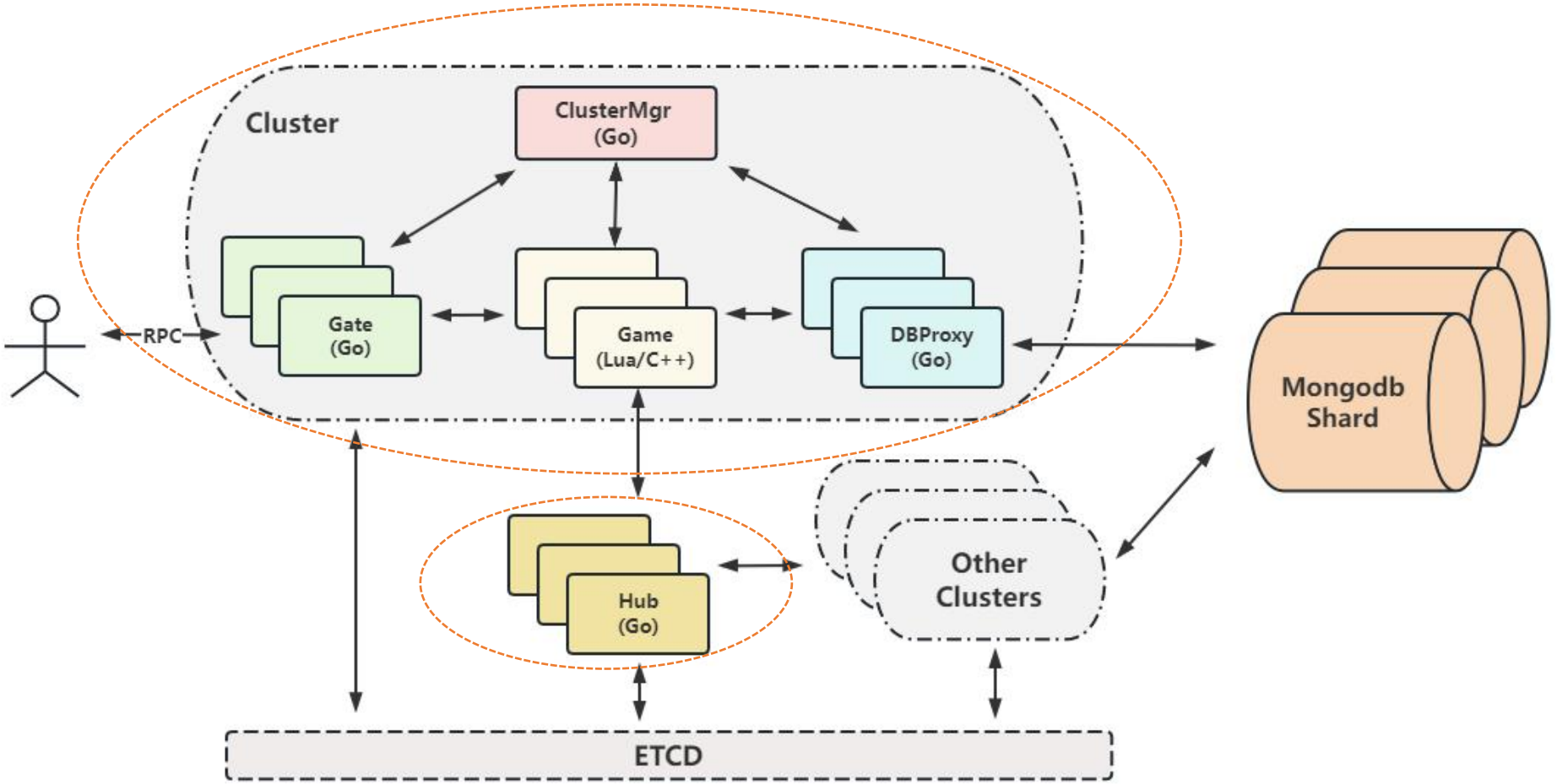
Game： 承载核心玩法

Gate： 客户端连接

DBProxy: Mongodb/Redis

ClusterMgr: 集群管控

Hub： 跨Cluster转发RPC

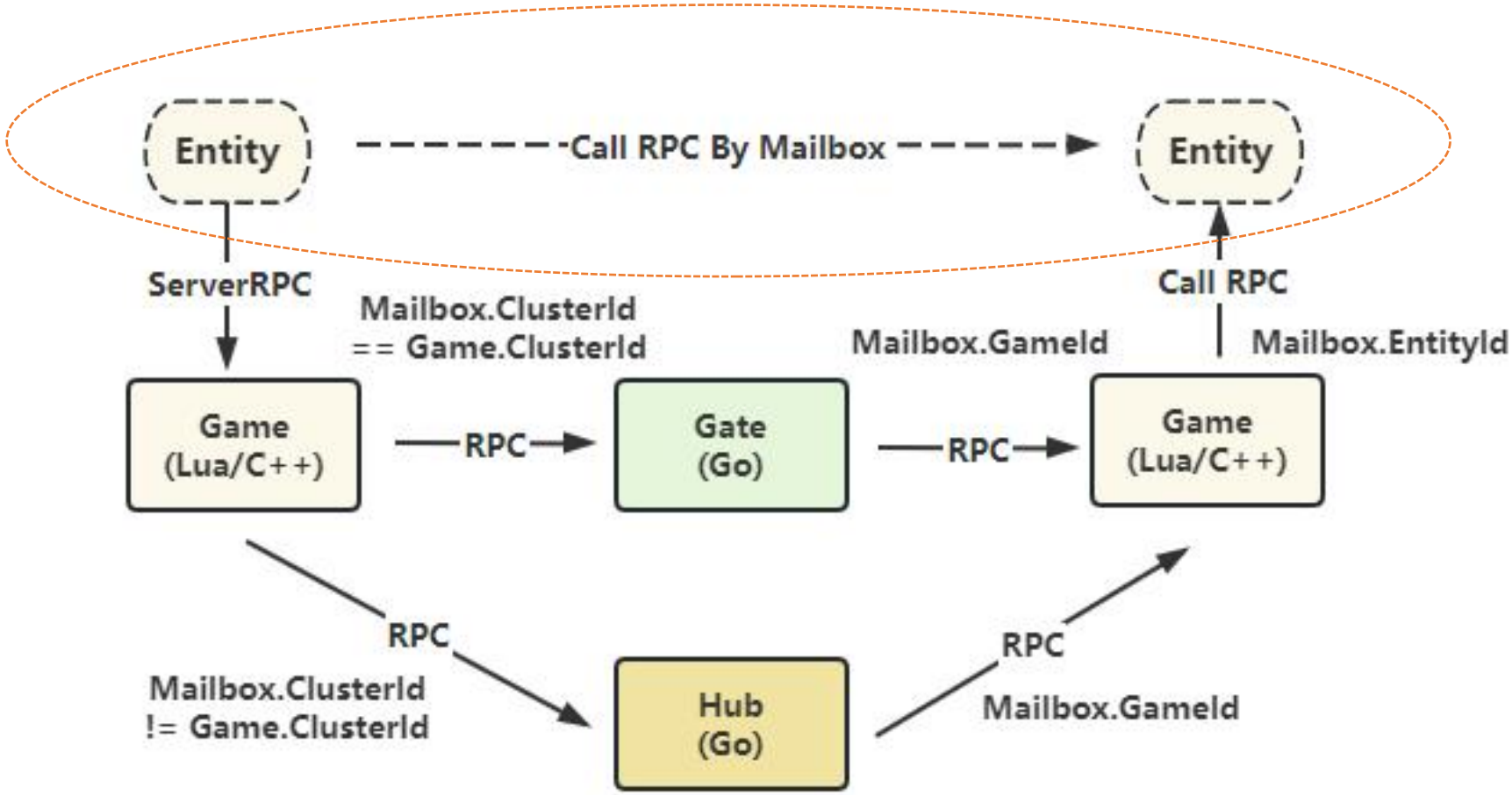


Phonest通用服务器框架

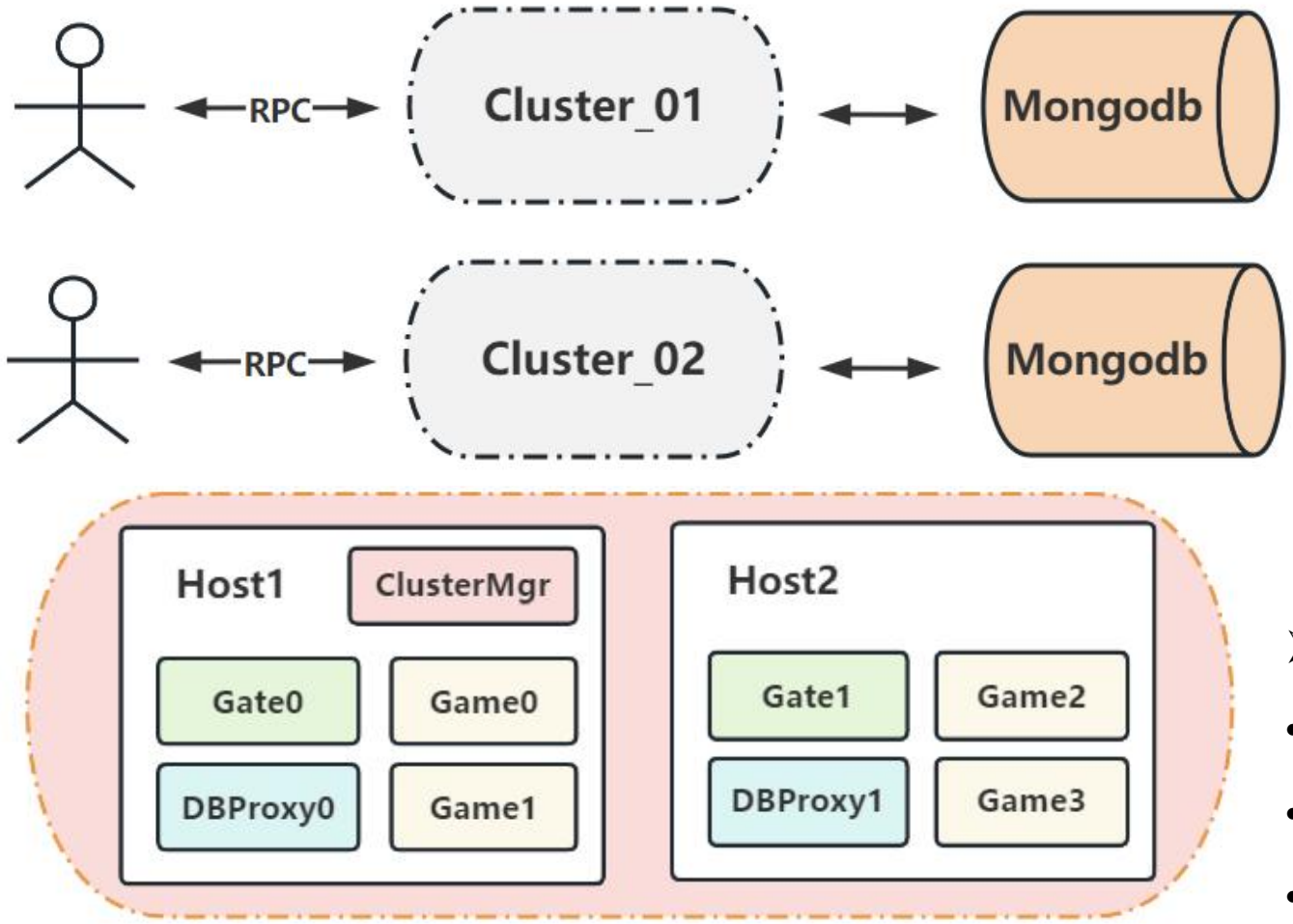
通信模式基于Actor模型实现

Entity Mailbox

- ClusterId
- GameId
- EntityId



Phonest实现分区服



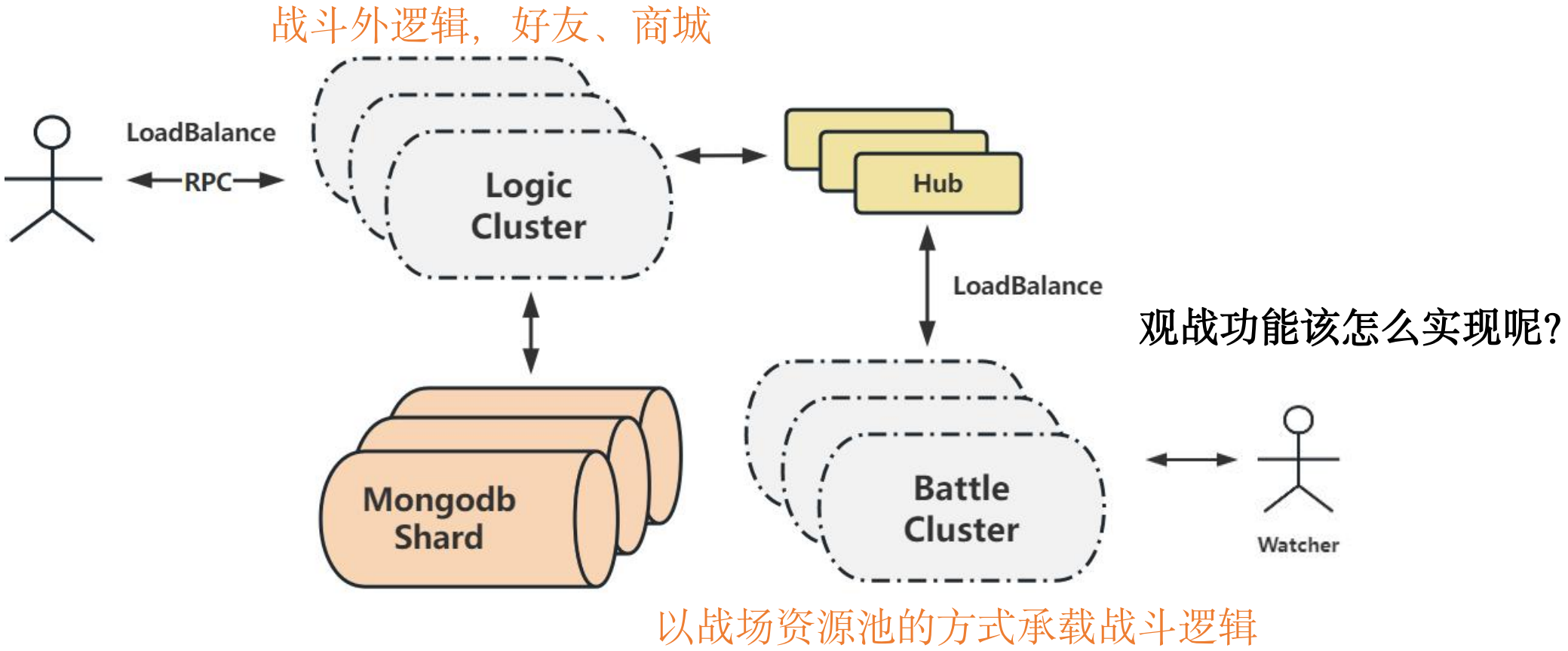
Cluster的进程均衡部署在两台物理机上

- 对于分区服的游戏而言
  - 单个Cluster就是一个区服
  - 单个Cluster可以部署在多台物理机上

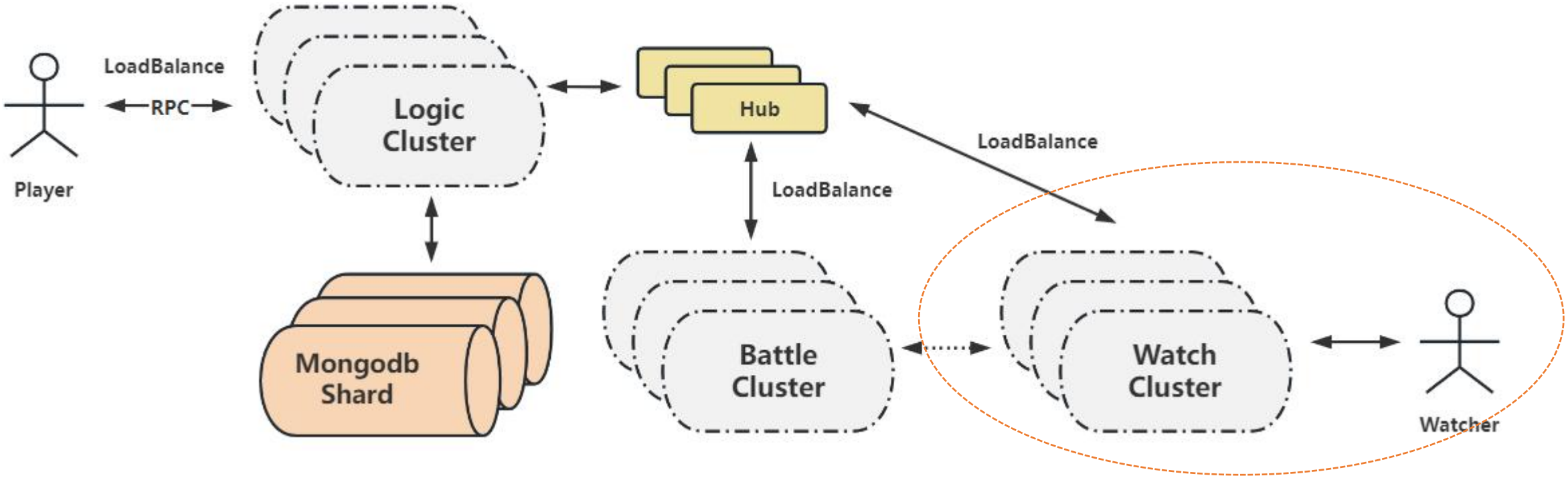
- 多个Game进程可以部署不同的业务逻辑
  - Game0部署登录Center
  - Game1部署好友和聊天Center
  - Game2和Game3部署核心玩法

Phonest实现全服

- 对于全服架构，可以部署成一个大型Cluster，也可以分拆成多个Cluster去承载
- 以Cluster为粒度，通过类型配置，部署不同的功能模块，同一类型的Cluster负载均衡



Phonest实现全服 -- 观战



- 扩展一个Watch类型的Cluster
- 隔离观战逻辑和战斗逻辑
- 一场战斗的RPC可以转发给多个观战集群

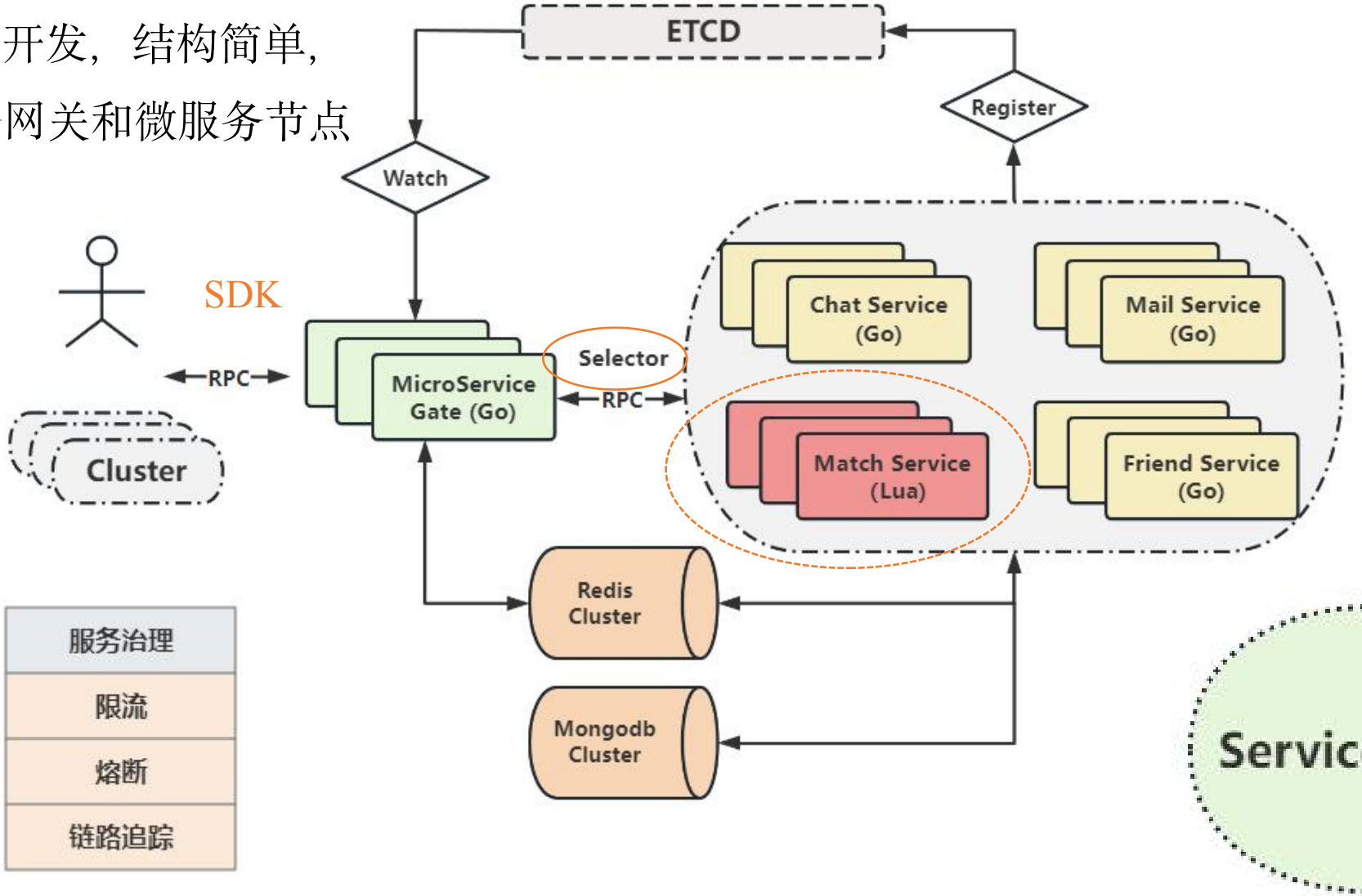
第三部分

Phonest微服务体系

Phonest MicroService

基于Golang自主开发，结构简单，

主要包括微服务网关和微服务节点



路由
Random
RoundRobin
Consistent Hash
Bijection

服务治理
限流
熔断
链路追踪

- 异构混合部署
- 比特流封包解包
- Protobuf协议

Service Mesh

Phonest MicroService

➢ **面向用户：提供SDK，快速接入**

- 与微服务网关建连接、加密、鉴权
- callService(service, method, args, callback, hint)

➢ **面向开发者：提供开发框架和工具链，快速开发**

- MS_Framework
- Service_Generator

MS_Framework

- 读写 Mongodb & Redis API 封装
- ETCD 注册和监听的 API 封装
- Opentracing 链路追踪 & Prometheus Metrics API 封装
- Service Runtime
  - one request one goroutine
  - work pool

```
import (
    msf "gitlab-sh.funplus.io/micro-service/ms_framework/framework"
)

msf.RegisterServiceHandler("createTeam", func() msf.ServiceHandler {return app.NewCreateTeamHandler()})
msf.RegisterServiceHandler("join", func() msf.ServiceHandler {return app.NewJoinHandler()})
msf.RegisterServiceHandler("quit", func() msf.ServiceHandler {return app.NewQuitHandler()})

msf.LaunchService(conf)
```

Service_Generator

微服务代码生成工具，自动生成与框架衔接的代码

```json
{
    "name": "TeamService",
    "methods": [
        {
            "name": "join",
            "args": [
                {
                    "field": "Uid",
                    "type": "string",
                    "comment": "操作者uid"
                },
                {
                    "field": "TeamId",
                    "type": "string",
                    "comment": "队伍Id"
                }
            ],
            "resp": [
            ]
        },
        {
            "more": "..."
        }
    ]
}
```

api.json

```go
package app
import (
    msf "gitlab-sh.funplus.io/micro-service/ms_framework/framework"
)
type JoinReq struct {
    Uid       string    //操作者uid
    TeamId    string    //队伍Id
}
type JoinResp struct {
    Ret [2]interface{}  //请求返回状态, (bool, string)
}
type JoinHandler struct {
    Req    *JoinReq
    Resp   *JoinResp
}
func NewJoinHandler() *JoinHandler {
    return &JoinHandler{
        Req: &JoinReq{},
        Resp: &JoinResp{
            Ret: [2]interface{}{true, ""},
        }
    }
}
func (handler *JoinHandler) Process(ctx *msf.MsContext) (msf.ActionType, error) {
    uid := handler.Req.Uid
    teamId := handler.Req.TeamId
    // add your code here
}
```

join.go

第四部分

有状态服务动态扩缩容

四人组队Boss战

➢ **玩法描述：四个玩家组成一个队伍，然后联合镇压一个Boss**

- TeamService ——处理组队逻辑
- BossService ——处理Boss战的逻辑

➢ **架构设计上需要考虑：**

- 动态扩缩容 ——根据负载情况自动调整系统的资源分配
- 有状态 ——以离散的方式模拟一个连续的世界，timer update
- 低延时 ——强竞技的游戏，客户端60帧，16ms update

四人组队

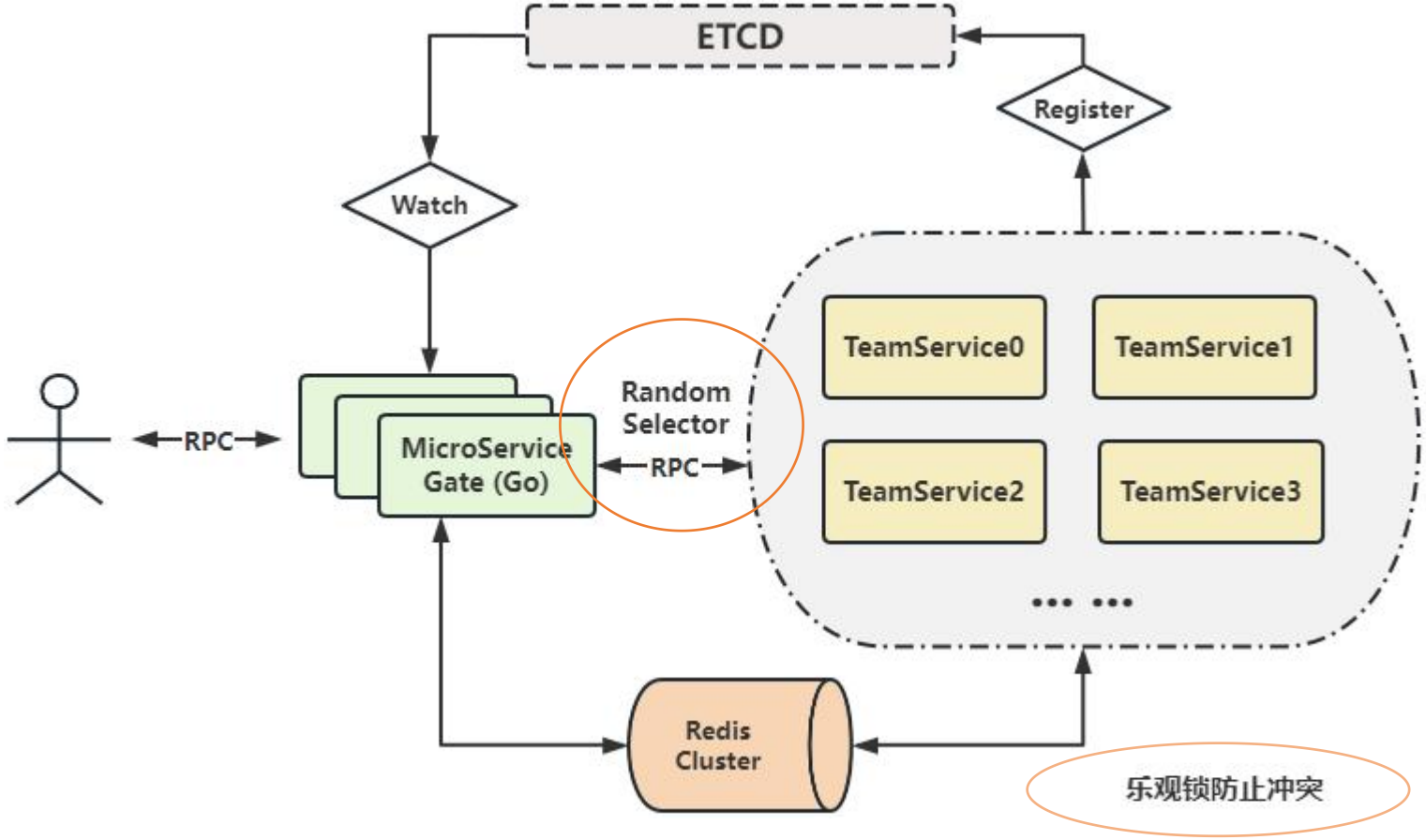
玩家创建一个队伍，成为队长，其他玩家可以申请加入一起组队，队伍人数有上限

能否用无状态的方式实现组队服务呢？　　需求简单、非高频、延时要求低

测试截图，不代表游戏最终品质

四人组队的无状态实现

微服务节点内存无状态，状态数据保存在Redis



并发冲突如何解决？

如何减少并发冲突？

四人组队的无状态实现 – 串行化

- 网关采用一致性哈希路由，根据teamId把微服务请求转发给同一个TeamService
- TeamService也根据teamId把微服务请求投递给同一个goroutine

```
79  // init work pool when launch service
80  if conf.Bool("worker-sequentially") {
81      workerPool.Init(worker.WithWorkerNum(conf.Int("worker-num")),
82                      worker.WithQueueSize(conf.Int("worker-job-queue-size")))
83  }
84  // add request to work pool
85  workerPool.Add(requestJob, PENDING_REQ_TIMEOUT, options.Hint)
```

还需要乐观锁么？

无状态实现的困境

- 组队添加超时逻辑该如何实现？检测超时的驱动源又在哪里？
- 游戏业务要求低延时，无状态的实现引入的序列化和反序列化会显著增加延时

测试截图，不代表游戏最终品质

四人组队打Boss，Boss血量共享，客户端上传对Boss的伤害，服务器结算后广播

客户端60帧 => 极限情况下每个玩家每秒请求60次Redis

战场数据高频序列化和反序列化

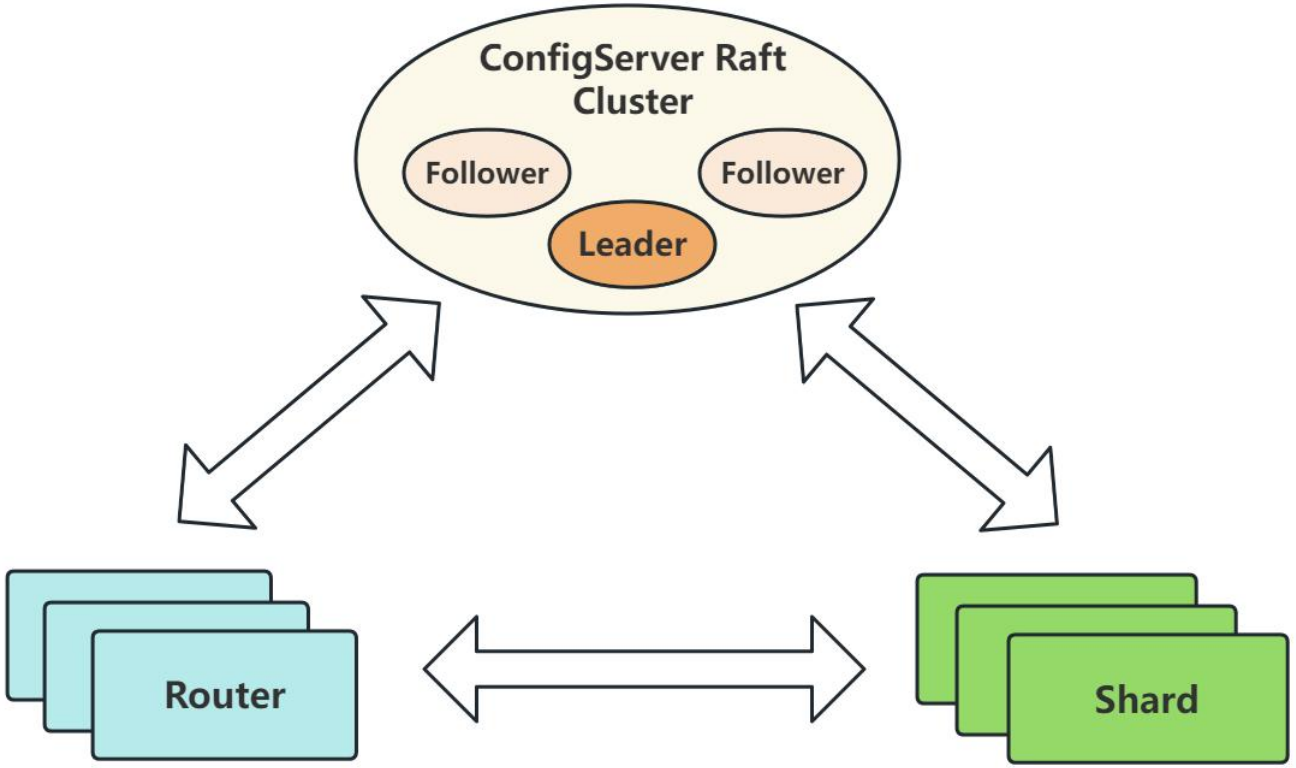
叠加高频的Redis乐观锁冲突

有状态服务动态扩缩容 -- 难点

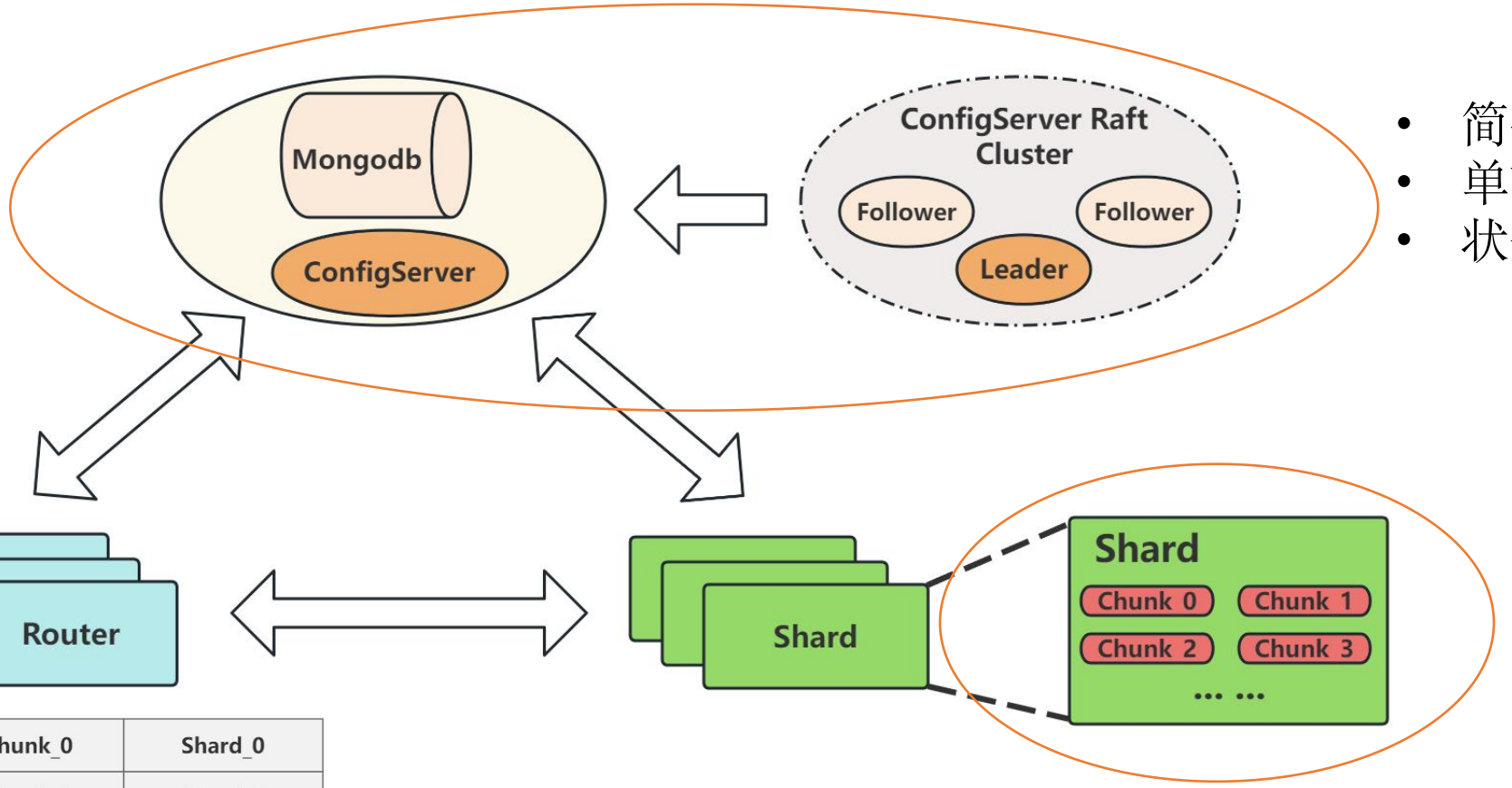
- 节点扩缩容后，负载如何再均衡？节点自治还是引入第三方控制节点？
- 容灾，网络丢包、进程Crash，各个节点的最终一致性如何达到？
- 对业务开发能否透明？ 业务逻辑、性能指标都不受影响

Seech架构图

- 整体设计参考Mongodb分片集群
- Router是路由节点，类似Mongos
- Shard是Service进程，类似Mongod
- ConfigServer是管控节点，主备高可用



Seech架构图



- 简化主备方案
- 单节点部署
- 状态数据落地数据库

路由表

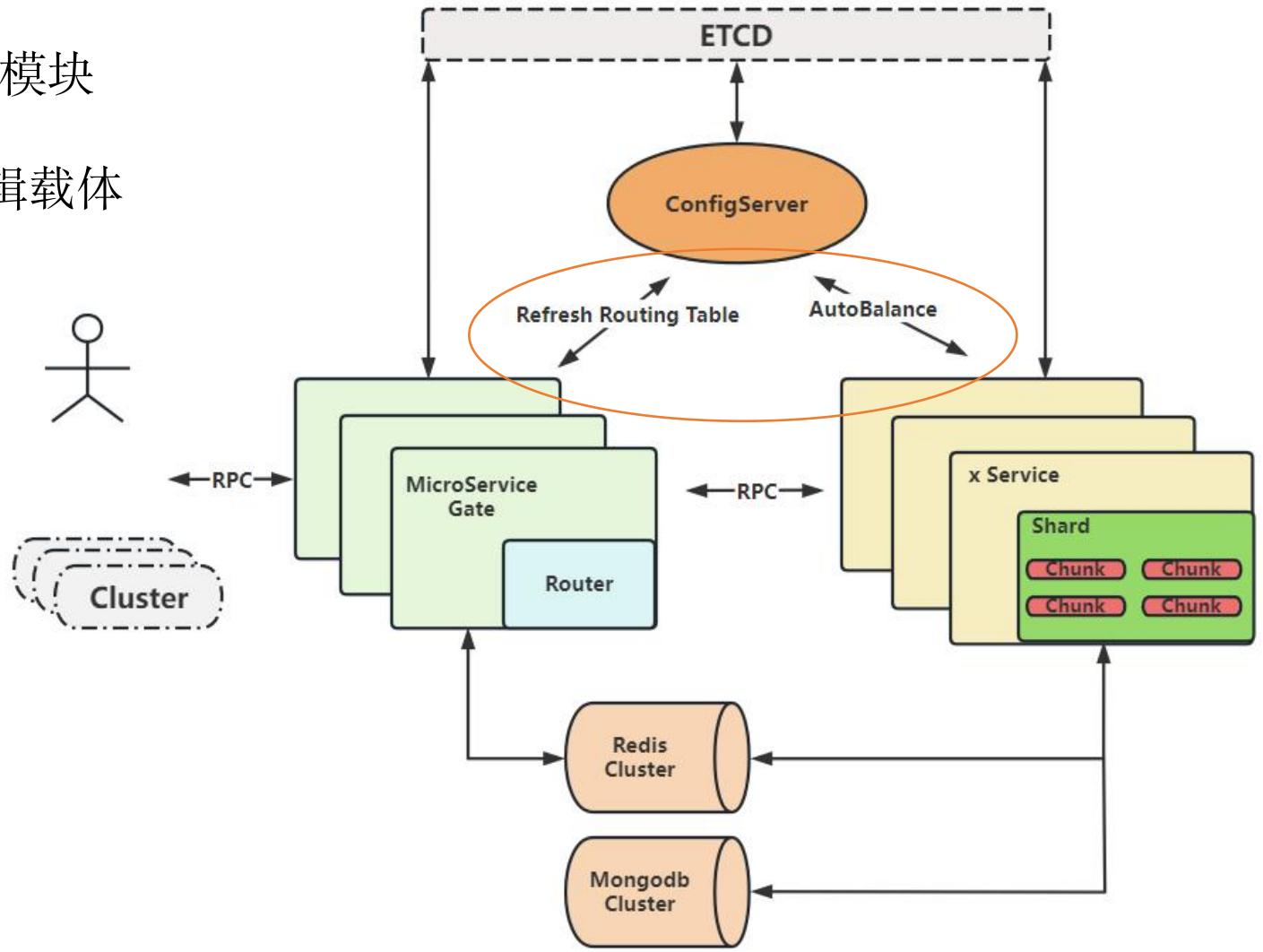
ReqID

Chunk_0	Shard_0
Chunk_1	Shard_0
Chunk_2	Shard_1
Chunk_3	Shard_1
... ...	... ...

- Chunk是一个逻辑概念，有唯一编号，总数固定，不可拆分或者合并
- 单个Chunk只属于单个Shard，不会同时在多个Shard上提供服务
- Chunk基于Hash规则，管理了一部分数据，并负责这部分数据的迁移

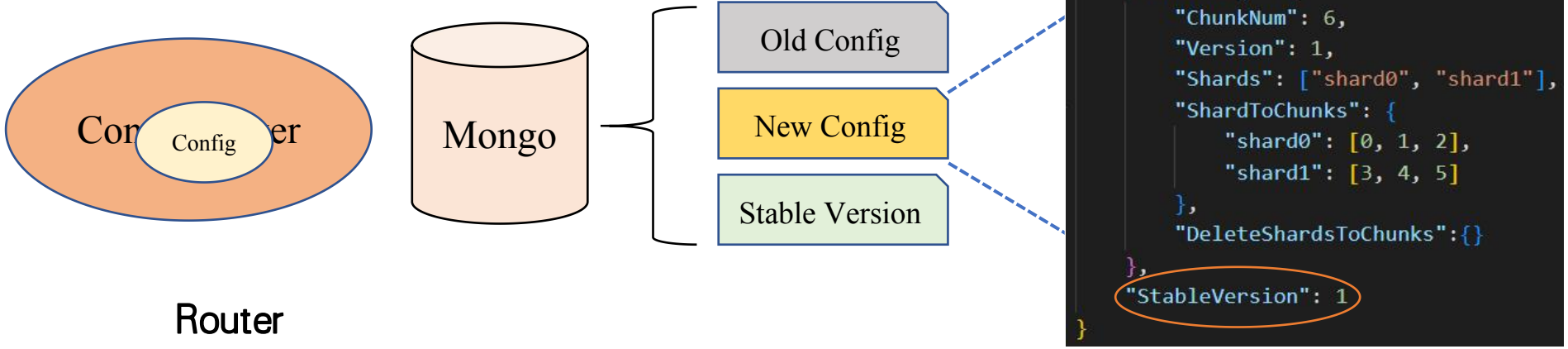
Seech架构图

- Router作为微服务网关的路由模块
- Shard作为有状态Service的逻辑载体
- ConfigServer AutoBalance



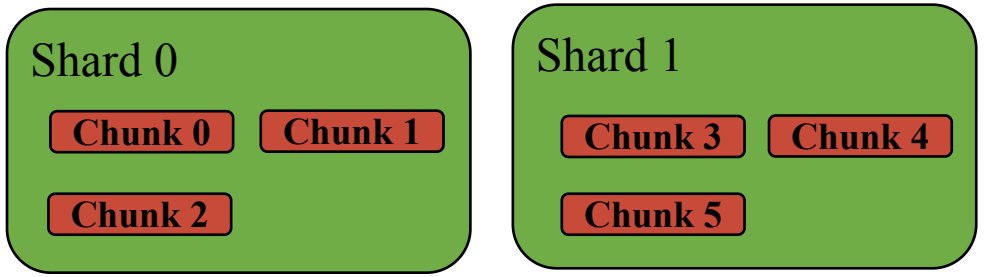
Seech示例 -- 初始化

6 Chunks 2Shards



Router

Chunk_0	Shard_0
Chunk_1	Shard_0
Chunk_2	Shard_0
Chunk_3	Shard_1
Chunk_4	Shard_1
Chunk_5	Shard_1



Hash(battleId) % ChunkNum => ChunkId

Seech动态扩缩容

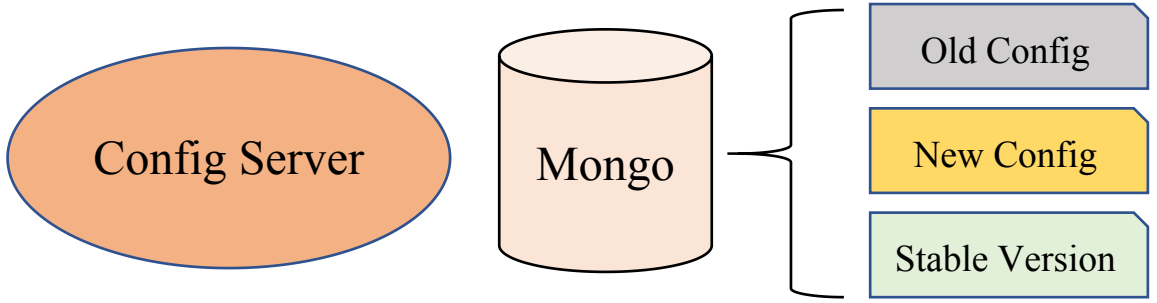
- 目标：通过AutoBalance实现Chunks在Shards上均衡分布

- 权衡：考虑调节均衡的过程尽量不影响业务性能
  - 参考GC过程，把一次扩缩容分成多个Step
  - 一个Step就是一次调节均衡的过程，也就是从OldConfig调整成NewConfig的过程
  - 控制OldConfig和NewConfig的差异，差异越大，表明这个Step迁移的数据越多

- 如何合理规划一个Step的大小?
  - 控制每个Shard往外迁移的Chunk数量
  - 每个Shard每个Step最多往外迁移一个Chunk

Seech示例 -- 节点扩容

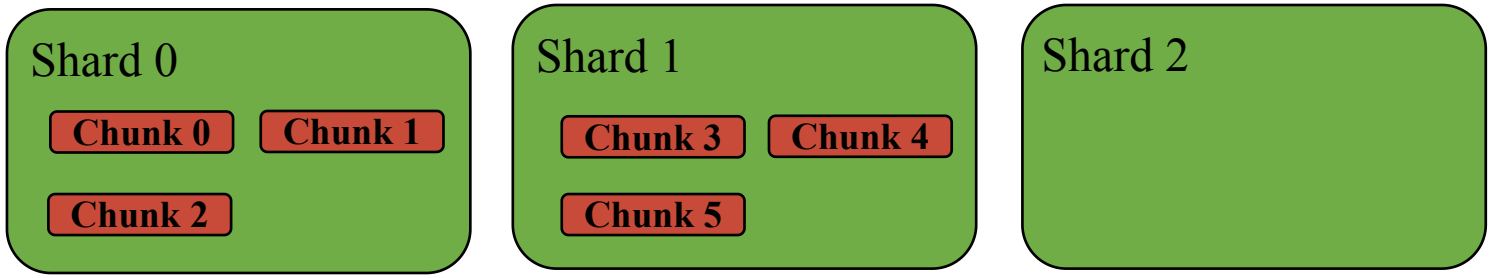
6 Chunks，From 2 Shards To 3 Shards




```
{
    "OldConfig": {
        "ChunkNum": 6,
        "Version": 1,
        "Shards": ["shard0", "shard1"],
        "ShardToChunks": {
            "shard0": [0, 1, 2],
            "shard1": [3, 4, 5]
        },
        "DeleteShardsToChunks":{}
    },
    "NewConfig": {
        "ChunkNum": 6,
        "Version": 2,
        "Shards": ["shard0", "shard1", "shard2"],
        "ShardToChunks": {
            "shard0": [0, 1],
            "shard1": [3, 4],
            "shard2": [2, 5]
        },
        "DeleteShardsToChunks":{}
    },
    "StableVersion": 2
}
```

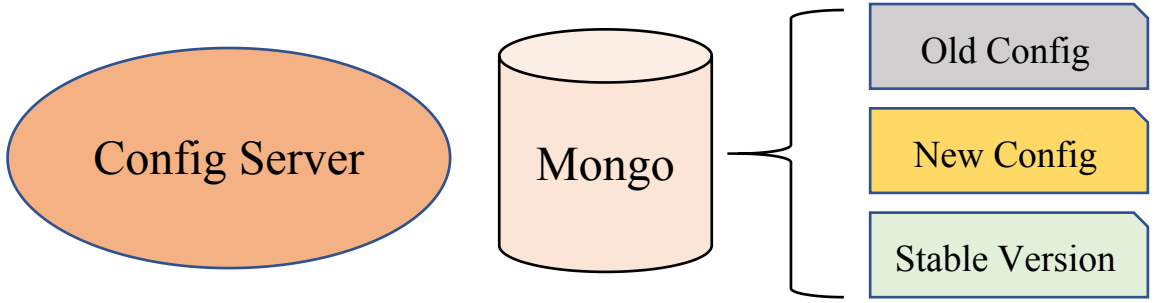
Router

Chunk_0	Shard_0
Chunk_1	Shard_0
Chunk_2	Shard_2
Chunk_3	Shard_1
Chunk_4	Shard_1
Chunk_5	Shard_2



Seech示例 -- 节点缩容

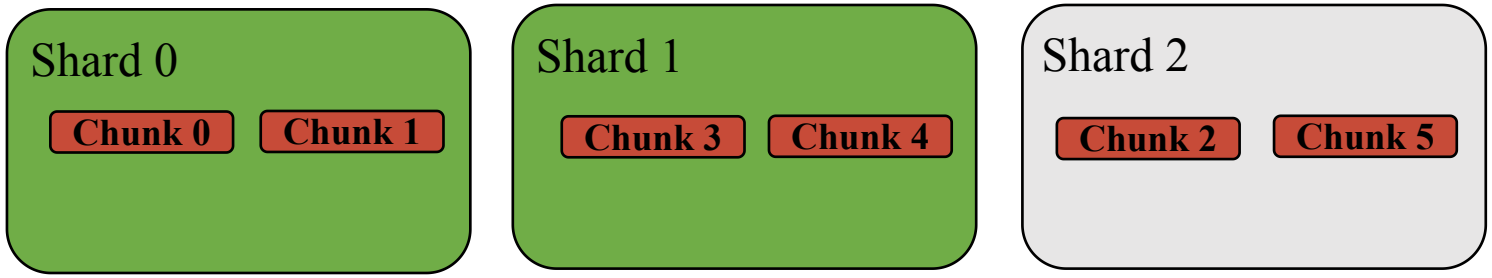
6 Chunks，From 3 Shards To 2 Shards Step_1




```
{
    "OldConfig": {
        "ChunkNum": 6,
        "Version": 2,
        "Shards": ["shard0", "shard1", "shard2"],
        "ShardToChunks": {
            "shard0": [0, 1],
            "shard1": [3, 4],
            "shard2": [2, 5]
        },
        "DeleteShardsToChunks":{}
    },
    "NewConfig": {
        "ChunkNum": 6,
        "Version": 3,
        "Shards": ["shard0", "shard1"],
        "ShardToChunks": {
            "shard0": [0, 1, 2],
            "shard1": [3, 4]
        },
        "DeleteShardsToChunks":{
            "shard2": [5]
        }
    },
    "StableVersion": 3
}
```

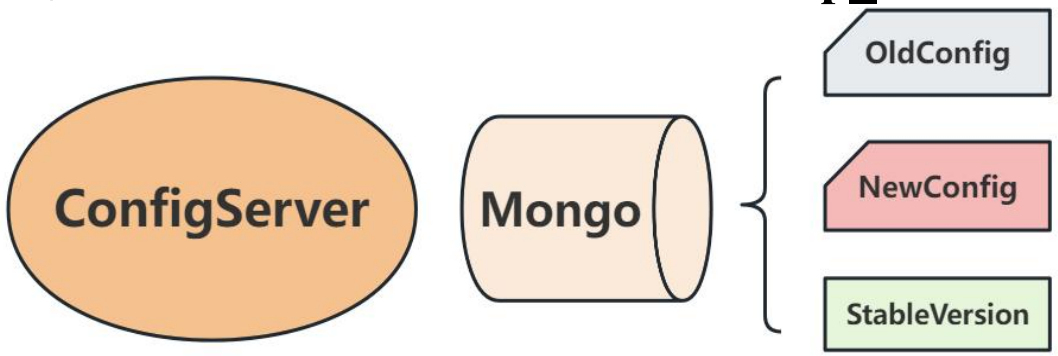
Router

Chunk_0	Shard_0
Chunk_1	Shard_0
Chunk_2	Shard_0
Chunk_3	Shard_1
Chunk_4	Shard_1
Chunk_5	Shard_2



Seech示例 -- 节点缩容

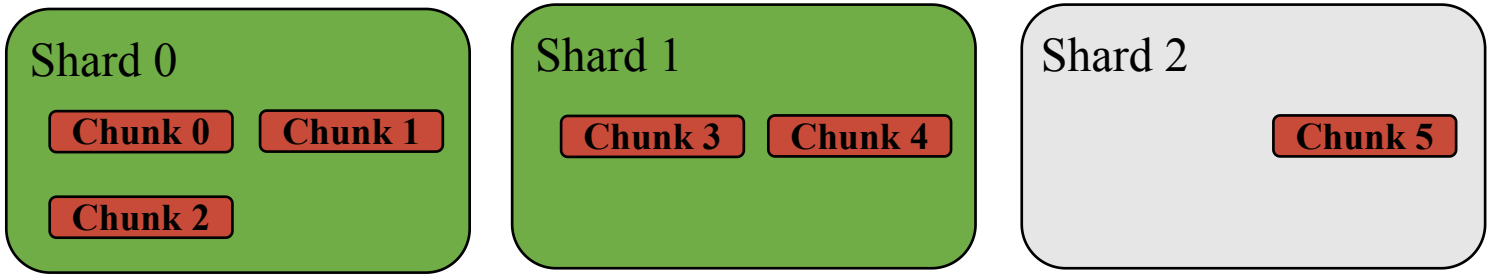
6 Chunks，From 3 Shards To 2 Shards Step_2




```
{
    "OldConfig": {
        "ChunkNum": 6,
        "Version": 3,
        "Shards": ["shard0", "shard1"],
        "ShardToChunks": {
            "shard0": [0, 1, 2],
            "shard1": [3, 4]
        },
        "DeleteShardsToChunks":{
            "shard2": [5]
        }
    },
    "NewConfig": {
        "ChunkNum": 6,
        "Version": 4,
        "Shards": ["shard0", "shard1"],
        "ShardToChunks": {
            "shard0": [0, 1, 2],
            "shard1": [3, 4, 5]
        },
        "DeleteShardsToChunks":{}
    },
    "StableVersion": 4
}
```

Router

Chunk_0	Shard_0
Chunk_1	Shard_0
Chunk_2	Shard_0
Chunk_3	Shard_1
Chunk_4	Shard_1
Chunk_5	Shard_1



Seech容灾

- 一个Step就是一次事务：所有Shard的配置都从OldConfig更新为NewConfig

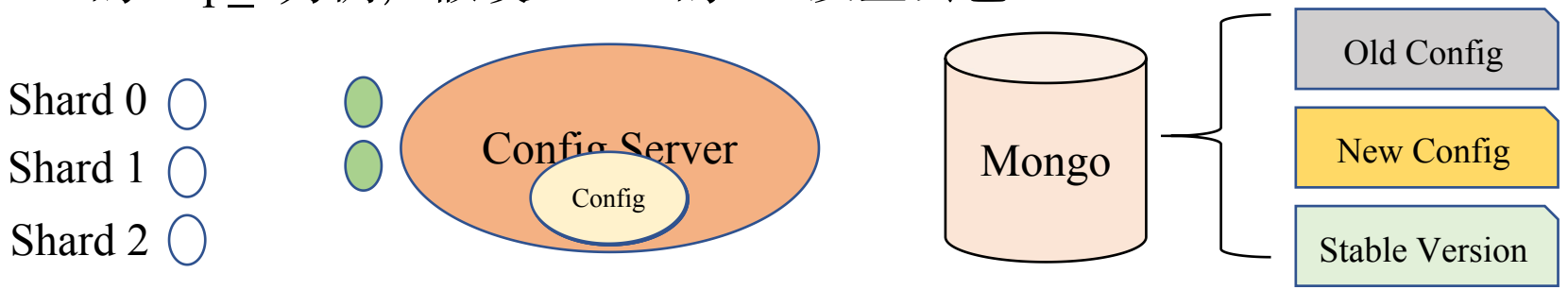
- 完成标志：ConfigServer更新完Mongodb的StableVersion

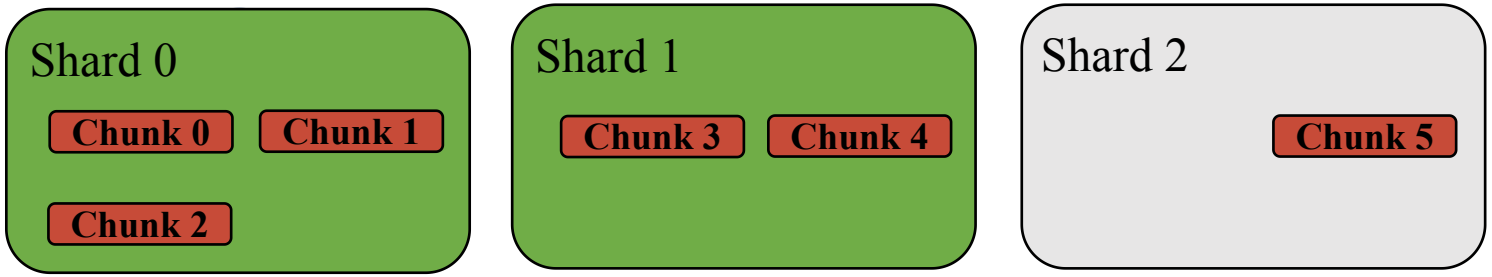
- 事务需要回滚吗?
  - 不回滚，比如扩节点，立马又缩节点，当成两次事务来处理。

- 如何保证事务一定能完成?
  - 无限重试，Step各个阶段都是可重入的

Seech容灾 – ConfigServer丢包

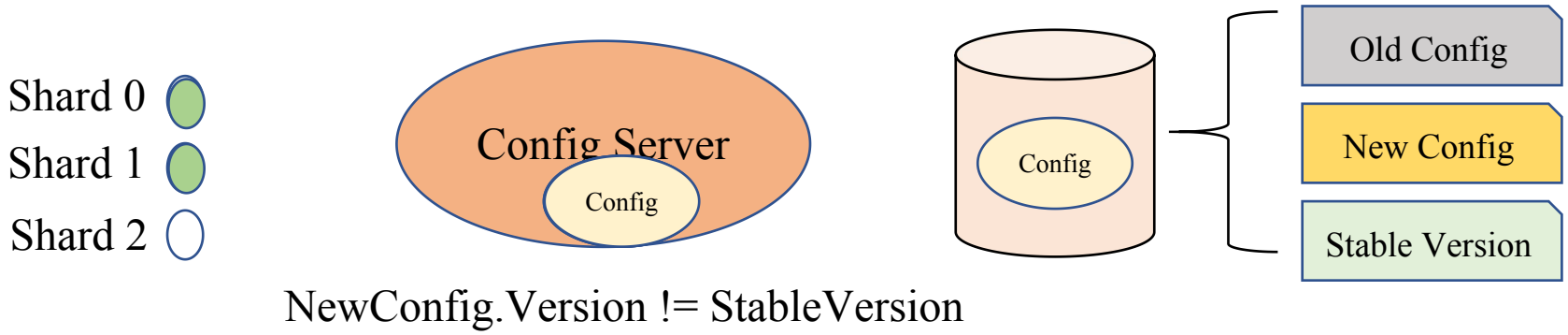
以缩容Shard2的Step_2为例，假设Shard2的Ack发生丢包



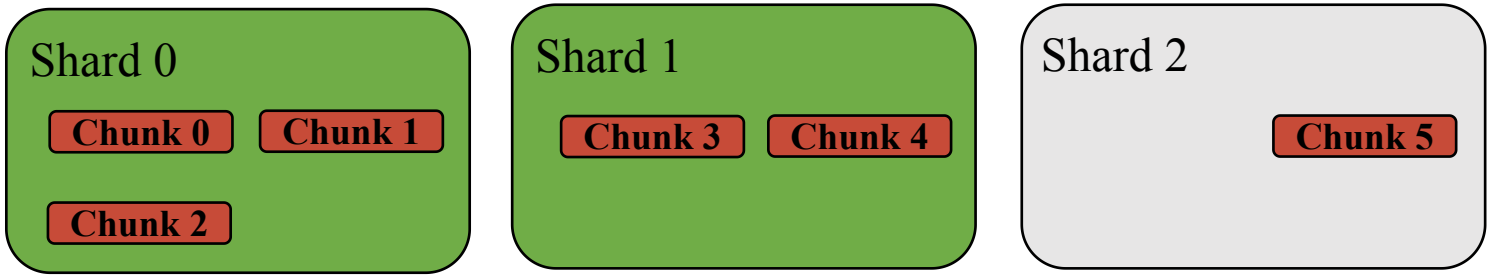


Seech容灾 – ConfigServer Crash

假设ConfigServer在等待Shard2的Ack过程中，发生了Crash

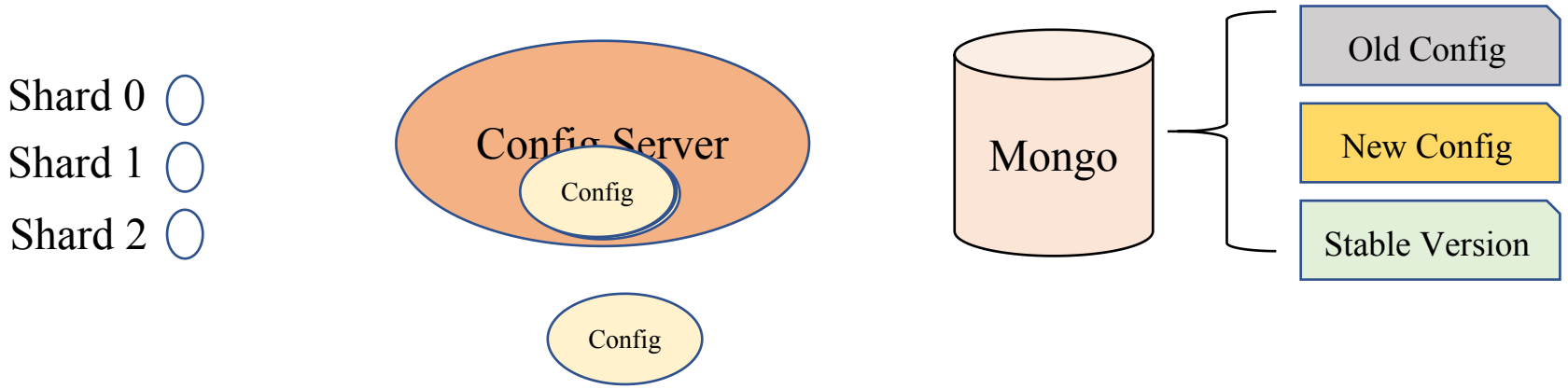


NewConfig.Version != StableVersion

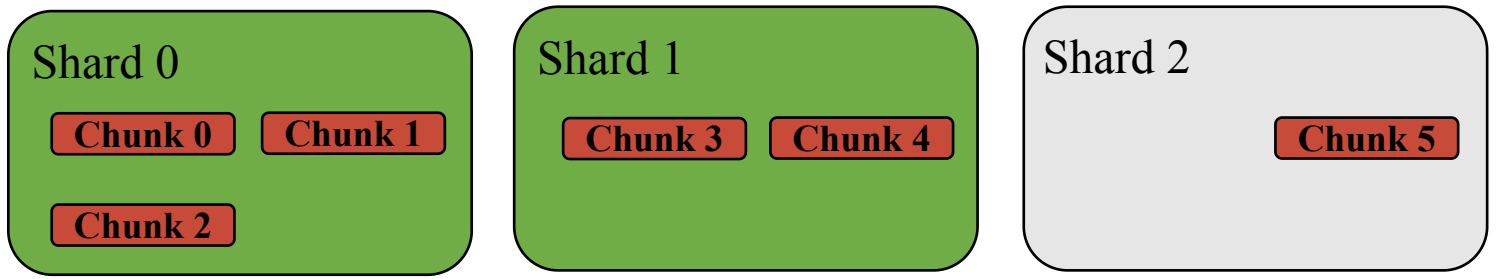


Seech容灾 - Shard丢包

假设Shard1没有收到ConfigServer下发的配置

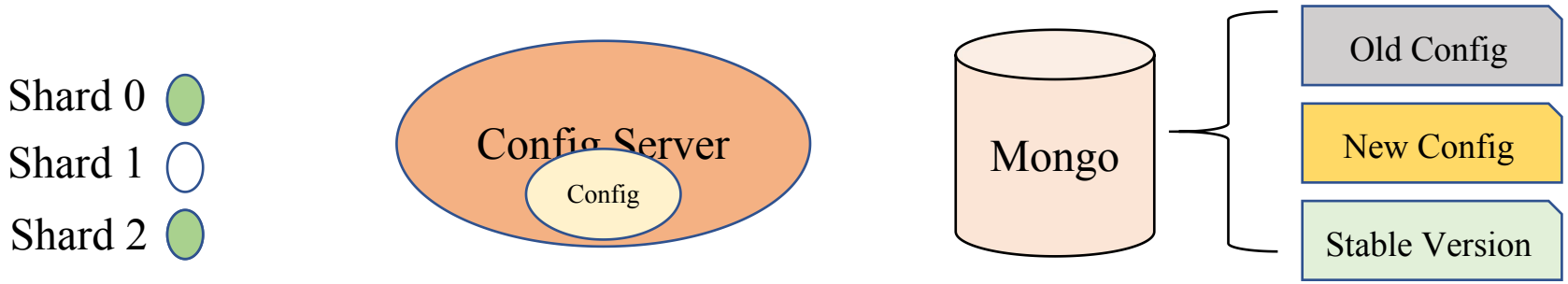


Shard1.Version < Shard2.Version



Seech容灾 – Shard Crash

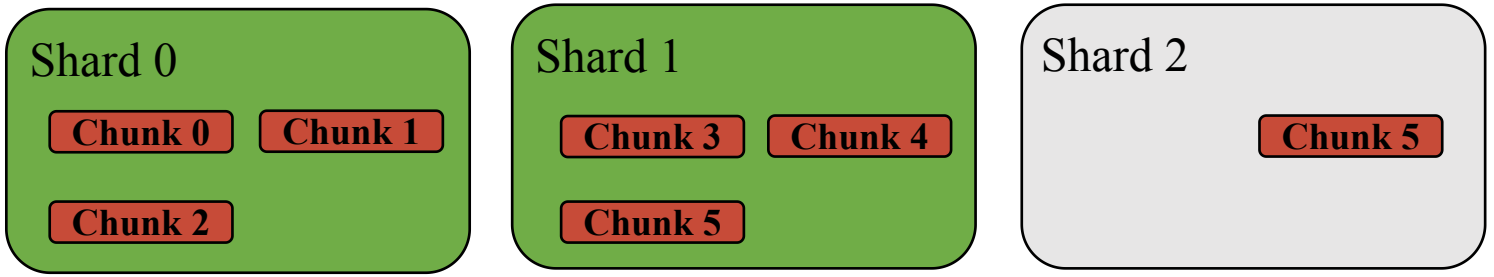
假设Shard1在收到Shard2迁移的Chunk5后立即发生了Crash



StableVersion != NewConfig.Version

根据OldConfig，重新创建Chunk3和Chunk4

等待Shard2迁移Chunk5



Shard2在所有Chunk都迁移完后能否直接下线？

Shard Crash

ConfigServer Crash

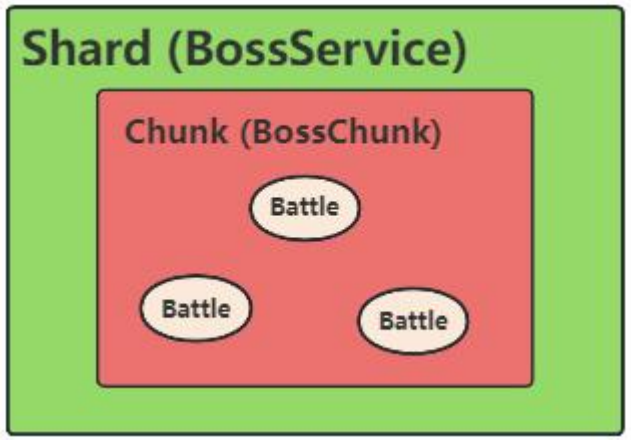
Seech逻辑完备性的验证

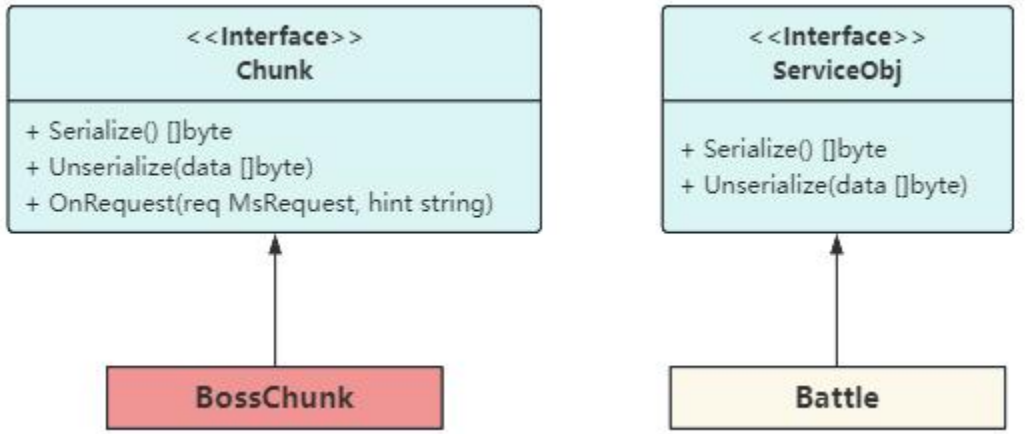
- Seech框架的分布式事务相对复杂，那如何保证逻辑的完备性呢?
- 基于TLA+，验证Seech的分布式事务能否达成最终一致性

- Paxos作者提出的模拟并发系统行为的逻辑框架
- 主要思想是将系统的行为描述成状态机，通过时序编排来穷举状态之间的转换是否符合预期
- 验证的过程，就是通过TLA+提供的描述语言，复刻出整个框架的状态机并执行

Boss战的有状态实现

- **BossService**
- BossChunk
- Battle





通过接口约束，衔接Seech框架和上层业务

Thanks