对于加域接口性能tps低的根因分析

一.问题概述

1. 软硬件环境

软件环境：集中域管平台v1.7.0【高可用k8s部署】

硬件环境：

V1.7-k8s环境：

物理机1：

名称	ip	CPU	内存	磁盘
redis-master1	10.20.15.102	16	32G	300G
redis-slave1/glusterfs-server	10.20.15.103	16	32G	300G
mariadb-master-1	10.20.15.104	16	32G	500G
mariadb-slave-1	10.20.15.105	16	32G	500G
k8s-master-1	10.20.15.108	16	32G	300G
k8s-node-1	10.20.15.109	16	32G	300G

物理机2：

名称	ip	CPU	内存	磁盘
redis-master2	10.20.15.113	16	32G	300G
redis-slave2/glusterfs-server	10.20.15.114	16	32G	300G
mariadb-master-2	10.20.15.115	16	32G	500G
mariadb-slave-2	10.20.15.116	16	32G	500G
k8s-master-2	10.20.15.117	16	32G	300G
k8s-node-2	10.20.15.118	16	32G	300G
mariadb-mha-manager	10.20.15.119	16	32G	300G

物理机3

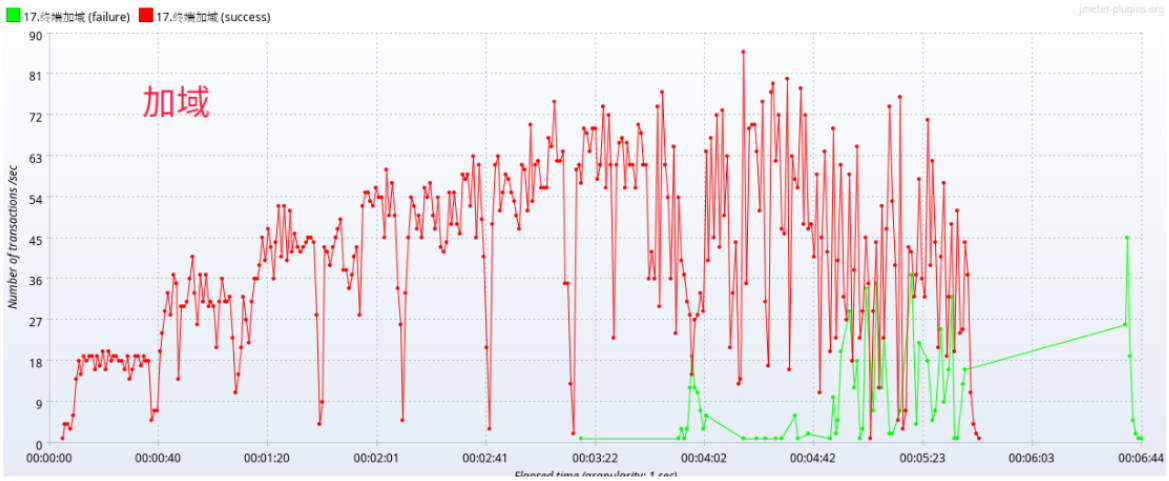
名称	ip	CPU	内存	磁盘
redis-master3/glusterfs-server	10.20.15.152	16	32G	300G
k8s-master-3	10.20.15.153	16	32G	300G
redis-slave3	10.20.15.163	4	8G	200G
nginx-1	10.20.15.122	4	8G	200G
nginx-2	10.20.15.124	4	8G	200G

压测工具：apache-jmeter-5.4.1

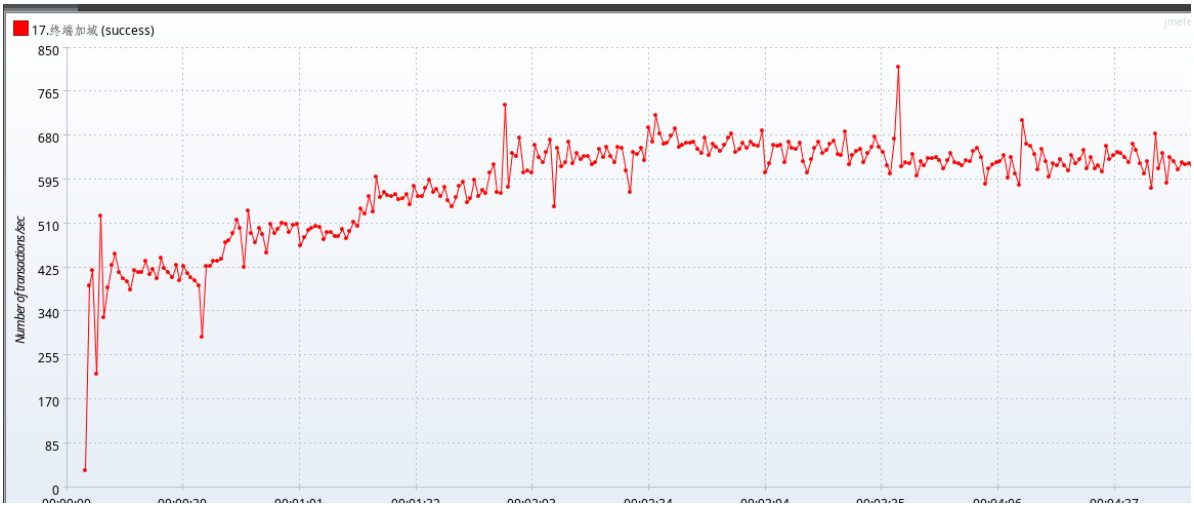
本接口逻辑仅涉及少量redis缓存操作，基本无磁盘文件读写的io操作，因此主要关注点为mariadb机器（数据库运行）和node机器（业务服务实际运行），

2. 问题描述

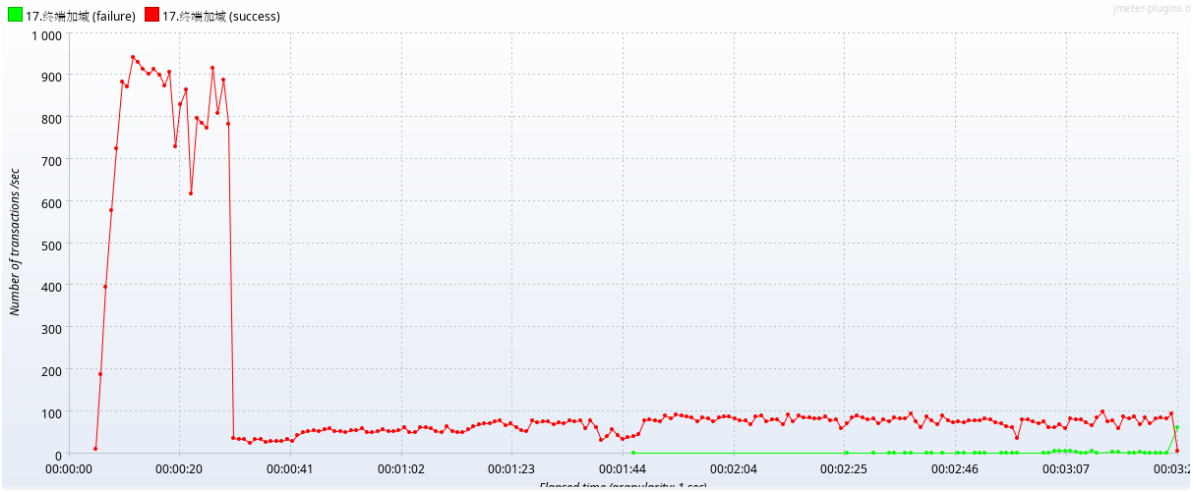
1.性能测试人员反馈在集中域管平台 v1.7.0 高可用环境下进行加域接口性能测试时，出现该接口平均tps低于60，且压测到后期时会出现大量报错的问题现象。而同版本同时期其他大部分高频接口的tps均可稳定在3000及以上。



2.上一个提测版本已有关于该加域问题的bug修复，修复人员反馈接口已修复并能够一直稳定在600tps且无报错。压测截图如下：



但是笔者自主压测时出现了百分百概率的前期tps极高，一段时间后，单位平均响应时间暴增，同时tps稳定在60以下，经过多次测试，发现在请求45000次左右时开始出现性能拐点，且到压测后期依然存在相对大量的报错。压测截图如下：



因此笔者获得的压测结果与上述两个描述均无法对应，情况较为复杂。

3. 复现步骤

1.测试人员通过sql脚本往集中域管平台生成5万部门和10万人员的数据

2.编造10万终端数据，辅以步骤1中预建的10万人员数据，按照$machine_id 和 $username 的格式生成压测数据data.xls

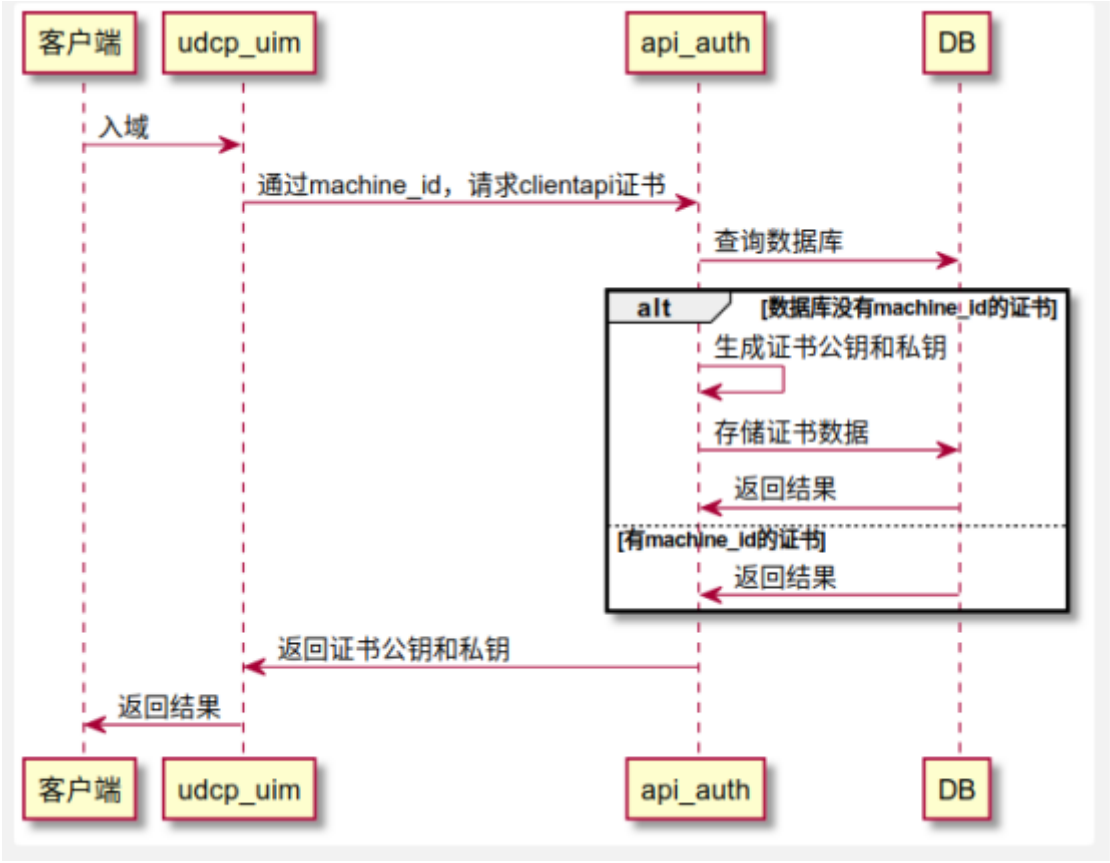
3.使用jmeter压测工具配合压测数据模板data.xls，压测加域端口，压测时间10分钟，阶梯性线性增加并发线程数，压测完毕观察结果图表，包括tps，平均响应时间，错误率等

二.问题分析

1. 功能流程介绍

1.1 加域流程介绍

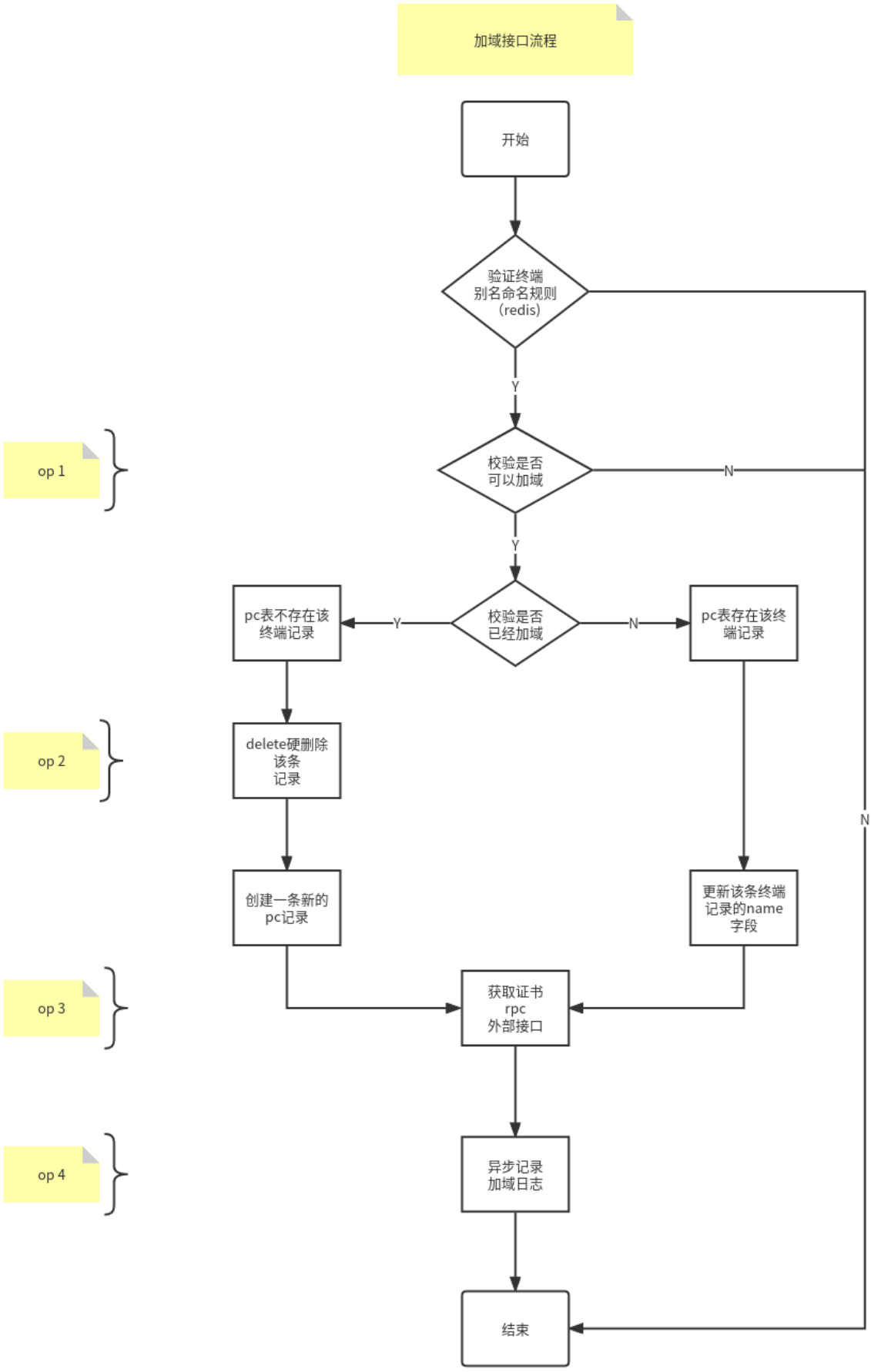
集中域管平台采用微服务架构，不同服务间采用rpc实现接口调用，加域接口主要涉及udcp_uim 和 api_auth 这两个服务，udcp_uim作为基础数据平台提供接口入口，api_auth作为证书相关的单独服务，当请求加域接口时会首先在udcp_uim进行相关数据的校验和准备工作，通过后udcp_uim内部向api_auth发送rpc请求来获得对应机器的证书，具体时序图如下：



加域流程总体时序图 图(1)

1.2 udcp_uim服务业务逻辑介绍

阅读加域的入口代码，将该业务流程的所有代码梳理完成，得到如下的udcp_uim逻辑流程图：



加域流程中的udcp_uim服务内部流程图 图(2)

图(2) 的op3即为 图(1) 中的udcp_uim向api_auth的rpc请求操作。

2. 原因猜想

通过上述流程图我们推测可能的导致性能问题的原因如下：

(1) 针对问题描述中的低tps问题，笔者首先猜想的是数据库资源消耗异常，业务代码的读写sql语句不够规范，造成了大量的性能浪费，这也是服务端最常见的数据库性能瓶颈。

(2) 端口超限问题，由于集中域管平台采用了微服务架构，而且历史上存在服务间请求没有复用端口，导致tcp连接长时间不会主动释放或者释放很缓慢，容器内的端口因此达到系统上限的问题，（一般为3W左右的端口上限）；再配合压测出现的当请求次数达到45000次左右时tps会骤降，平均响应时间陡增，并稳定在低tps的现象，自然也会怀疑这个原因所致。

(3) goroutine滥用，图(2) 中的 op4，即异步记录加域日志步骤，代码中直接采用go func的形式，该方式虽然是异步处理逻辑，但由于在压测环境下，短时间内会有极其大量的请求，也可能造成短时间启动了过多的go 协程，导致同一时间段内数据库的连接压力以及容器的调度压力，因此也作为合理怀疑点。

(4) 针对压测后期的出现相对大量的错误率问题，主要怀疑为压测时存在sql语句执行超时，又或者存在rpc调用超时的问题，一般来讲前期无报错后期才出现报错，可以合理怀疑为上述超时问题。

3. 排查手段

(1) 对于上述猜想1，从业务日志记录的错误开始排查，是否有MySQL错误日志，获取MySQL数据库的慢查询/插入语句，分析语句的合理性。

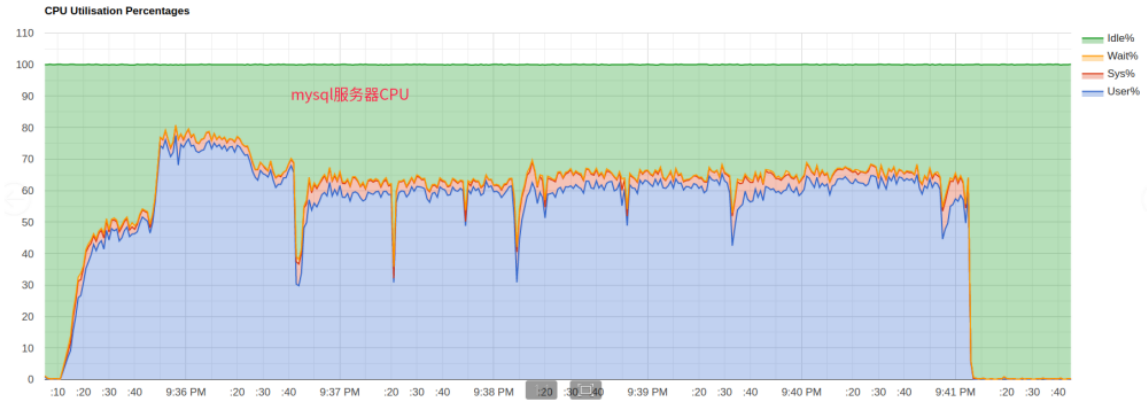
(2) 对于上述猜想2，进入udcp_uim容器通过netstat命令监控压测时的端口占用情况，是否存在udcp_uim作为客户端时的大量TIME_WAIT状态等。

(3) 对于上述猜想3，进行go协程处理日志的协程池化改造，同时对于数据库资源消耗占用，使用性能监控工具（nmo，nprometheus，MySQLd_exporter等）监控数据库磁盘io ，CPU资源消耗等。

(4) 对于上述猜想4，进入MySQL服务器，重点监控在压测出现大量报错时的错误日志，并对比前期正常无报错的执行日志。

4. 猜想验证

猜想（1）和（3）验证：通过数据库服务器的监控得到MySQL所在服务器的cpu占用非常高，极不正常。



进入容器中排查，通过调取单次接口观察sql日志打印，发现出现了非常的sql语句，其中还有重复的sql，明显是超过本功能实现所需要的合理sql数量，

```
/home/wang/Desktop/av/udcp-uim/vendor/pkg.deepin.com/service/lib/storage/db/v8/conn.go:137)
[2022-03-14 10:01:42]  [9.91ms]  SELECT sys_value FROM `dcmc_sys_config`  WHERE (sys_key = 'name_pc_by_district') ORDER BY `dcmc_sys_config`.`id` ASC LI
MIT 1
[1 rows affected or returned ]

/home/wang/Desktop/av/udcp-uim/vendor/pkg.deepin.com/service/lib/storage/db/v8/conn.go:137)
[2022-03-14 10:01:42]  [0.45ms]  SELECT sys_value FROM `dcmc_sys_config`  WHERE (sys_key = 'pc_name_rules') ORDER BY `dcmc_sys_config`.`id` ASC LIMIT 1

[1 rows affected or returned ]

/home/wang/Desktop/av/udcp-uim/vendor/pkg.deepin.com/service/lib/storage/db/v8/conn.go:137)
[2022-03-14 10:01:42]  [0.73ms]  SELECT * FROM `dcmc_admin`  WHERE (id=1) ORDER BY `dcmc_admin`.`id` ASC LIMIT 1
[1 rows affected or returned ]

/home/wang/Desktop/av/udcp-uim/vendor/pkg.deepin.com/service/lib/storage/db/v8/conn.go:455)
[2022-03-14 10:01:42]  [0.85ms]  SELECT count(*) FROM `dcmc_admin`  WHERE (level_struct like '000000%')
[0 rows affected or returned ]

/home/wang/Desktop/av/udcp-uim/vendor/pkg.deepin.com/service/lib/storage/db/v8/conn.go:455)
[2022-03-14 10:01:42]  [9.84ms]  SELECT count(*) FROM `dcmc_pc`  WHERE `dcmc_pc`.`deleted_at` IS NULL
[0 rows affected or returned ]

/home/wang/Desktop/av/udcp-uim/vendor/pkg.deepin.com/service/lib/storage/db/v8/conn.go:137)
[2022-03-14 10:01:42]  [0.72ms]  SELECT sys_value FROM `dcmc_sys_config`  WHERE (sys_key = 'profile') ORDER BY `dcmc_sys_config`.`id` ASC LIMIT 1
[1 rows affected or returned ]

/home/wang/Desktop/av/udcp-uim/vendor/pkg.deepin.com/service/lib/storage/db/v8/conn.go:137)
[2022-03-14 10:01:42]  [0.58ms]  SELECT * FROM `dcmc_department`  WHERE (id in(1))
[1 rows affected or returned ]

/home/wang/Desktop/av/udcp-uim/vendor/pkg.deepin.com/service/lib/storage/db/v8/conn.go:137)
[2022-03-14 10:01:42]  [0.58ms]  SELECT department_id, count(id) as count FROM `dcmc_user`  WHERE `dcmc_user`.`deleted_at` IS NULL AND ((department_id i
n (1))) GROUP BY department_id
[0 rows affected or returned ]

/home/wang/Desktop/av/udcp-uim/vendor/pkg.deepin.com/service/lib/storage/db/v8/conn.go:137)
[2022-03-14 10:01:42]  [0.37ms]  SELECT parent_id as department_id, count(id) as count FROM `dcmc_department`  WHERE (parent_id in (1)) GROUP BY parent_
id
[0 rows affected or returned ]

/home/wang/Desktop/av/udcp-uim/vendor/pkg.deepin.com/service/lib/storage/db/v8/conn.go:137)
[2022-03-14 10:01:42]  [0.94ms]  SELECT * FROM `dcmc_pc`  WHERE `dcmc_pc`.`deleted_at` IS NULL AND ((machine_id = '3438c74742aa3b7fba32580b40b3d988')) O
RDER BY `dcmc_pc`.`id` ASC LIMIT 1
[1 rows affected or returned ]

/home/wang/Desktop/av/udcp-uim/vendor/pkg.deepin.com/service/lib/storage/db/v8/conn.go:137)
[2022-03-14 10:01:42]  [4.29ms]  UPDATE `dcmc_pc` SET `host_name` = '', `product_name` = '', `name` = 'ooo', `remark` = '', `department_id` = 1, `status
```

在业务逻辑中发现这样的代码，

```
    }

    // 校验是否达到加域上限
    pro := profile.Get(1)
```

该代码会实时获取当前的授权信息，其内部实现包含很多MySQL的数据读取，且绝大部分数据并非本功能所需

```
172  func Get(adminID int) *Profile {   yangyia, 2020/9/22 下午8:15 • fix: 43222 WEB二级管理员人数显示错误
173      locker.RLock()
174      var info = *_profile
175      locker.RUnlock()
176
177      if len(info.Logo) > 0 {
178          info.Logo = filepath.Join( elem...: "/api/", config.Config().LogoIconDir, info.Logo)
179      } else {
180          info.Logo = filepath.Join( elem...: "/api/", config.Config().DefaultImgDir, "logo.svg")
181      }
182
183      if len(info.Favicon) > 0 {
184          info.Favicon = filepath.Join( elem...: "/api/", config.Config().FaviconIconDir, info.Favicon)
185      } else {
186          info.Favicon = filepath.Join( elem...: "/api/", config.Config().DefaultImgDir, "favicon.png")
187      }
188
189      info.AdminTotal, _ = getSubAdminTotal(adminID)
190      info.PictureSpaceUsage, _ = getPictureSpaceUsage( dir: "data")
191      info.CurrentPcs, _ = getCurrentPcs()
192      info.WebSSHIsOpen = config.Config().WebSSHIsOpen
193      dbProfile, _ := getProfileFromDb()
194      if len(info.Name) == 0 {
195          info.Name = dbProfile.Name
196      }
```

该代码存在很多数据库的查询操作，但是业务用到的仅仅只是其中的加域数量信息，且该信息可以通过一个已有实现的简易方法来获取。在性能测试的高并发场景下，预计能节省极其多的MySQL资源消耗。

同时我们还发现，记录加域日志逻辑中存在两个很简单的通过id获取部门信息方法调用，追溯发现其所调用的接口里面存在大量的其他逻辑操作，是明显的方法调用意图和实际影响完全不匹配的问题，

```go
// GetDepartmentByID 获取单条部门记录
func GetDepartmentByID(ids []int) ([]entity.DepartmentListRow, error) {
    var departments []entity.DepartmentListRow
    err := model.GetSlaveDB().Model(&entity.Department{}).Where( query: "id in(?)", ids).Scan(&departments).Error
    if err != nil { departments, err }
    //查询成员数量
    // 用户数量
    userCount := []response.DeptMemberCount{}
    userCountQueryErr := model.GetSlaveDB().Model(model.User{}).Select( query: "department_id, count(id) as count").
        Where( query: "department_id in (?)", ids).Group( query: "department_id").Scan(&userCount).Error
    // 数据解析
    userMap := make(map[int]int)
    if userCountQueryErr == nil {...}
    // 子部门数量
    depCount := []response.DeptMemberCount{}
    depCountQueryErr := model.GetSlaveDB().Model(&entity.Department{}).Select( query: "parent_id as department_id, count(id) as count").
        Where( query: "parent_id in (?)", ids).Group( query: "parent_id").Scan(&depCount).Error
    //数据解析
    depMap := make(map[int]int)
    if depCountQueryErr == nil {...}
    //数据更新
    for i := range departments {...}

    return departments, nil
}
```

可以看到该接口完全不能与接口名对应，其查询了很多其表，还有大量的in查询，属于严重超出实际需求的错误调用，于是笔者也改成了真实的仅查询本部门信息的方法

```go
// WriteJoinLog 写入加域日志
func WriteJoinLog(pc model.Pc, username string) {
    // 获取终端部门信息
    var deName string
    deps, _ := GetDepartmentByID([]int{pc.DepartmentID})
    if len(deps) > 0 {
        deName = deps[0].Name
    }

    // 用户信息
    user, err := model.GetUserByUsername(username)
    if err != nil {
        log.Error("WriteJoinLog model.GetUserByUsername", log.FieldErr(err))
        return
    }

    // 获取用户部门信息
    var userDeName string
    userDeps, _ := GetDepartmentByID([]int{user.DepartmentID})
    if len(deps) > 0 {
        userDeName = userDeps[0].Name
    }

    var l mq.UdcpLogPcDomain
    l.Type = mq.InDomain
    l.PcDepartmentID = pc.DepartmentID
```

```go
// WriteJoinLog 写入加域日志
func WriteJoinLog(pc model.Pc, username string) {
    // 获取终端部门信息
    var deName string

    dept, _ := db.QueryDepartmentByID(pc.DepartmentID,nil)
    if dept.ID > 0 {
        deName = dept.Name
    }

    // 用户信息
    user, err := model.GetUserByUsername(username)
    if err != nil {
        log.Error("WriteJoinLog model.GetUserByUsername", log.FieldErr(err))
        return
    }

    // 获取用户部门信息
    var userDeName string
    userDept, _ := db.QueryDepartmentByID(user.DepartmentID,nil)
    if userDept.ID > 0 {
        userDeName = userDept.Name
    }

    var l mq.UdcpLogPcDomain
    l.Type = mq.InDomain
    l.PcDepartmentID = pc.DepartmentID
```

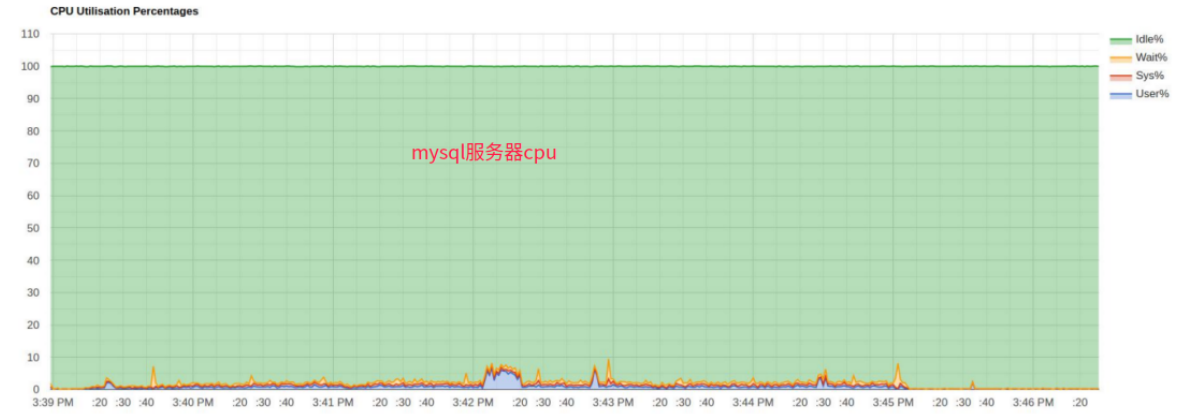
以上两处极其不合理的MySQL调用语句非常明显，会非常的拖累正常的业务性能，因此笔者也在第一时间进行了上述修改优化。

同时针对加域日志的协程池化处理，使用ants协程池额定 如果依旧使用goroutine的简单方式一个接口开启一个协程，会导致同一时刻MySQL的连接占用，qps激增，而日志记录本身也是异步的方式因此该处应该被优化为协程池化的处理。

```go
// WriteJoinLog 写入加域日志
func WriteJoinLog(pc model.Pc, username string) {
    // 获取终端部门信息
    var deName string

    dept, _ := db.QueryDepartmentByID(pc.DepartmentID,nil)
    if dept.ID > 0 {
        deName = dept.Name
    }

    // 用户信息
    user, err := model.GetUserByUsername(username)
    if err != nil {
        log.Error("WriteJoinLog model.GetUserByUsername", log.FieldErr(err))
```

```go

// WriteJoinLog 写入加域日志
func WriteJoinLog(pc model.Pc, username string) {
    _=Poolants.Submit(func() {
        // 获取终端部门信息
        var deName string

        dept, _ := db.QueryDepartmentByID(pc.DepartmentID,nil)
        if dept.ID > 0 {
            deName = dept.Name
        }

        // 用户信息
        user, err := model.GetUserByUsername(username)
        if err != nil {
            log.Error("WriteJoinLog model.GetUserByUsername", log.FieldErr(err))
            return
        }

        // 获取用户部门信息
        var userDeName string
        userDept, _ := db.QueryDepartmentByID(user.DepartmentID,nil)
        if userDept.ID > 0 {
            userDeName = userDept.Name
        }

        var l mq.UdcpLogPcDomain
        l.Type = mq.InDomain
        l.PcDepartmentID = pc.DepartmentID
        l.PcDepartmentName = deName
        l.PcID = pc.ID
        l.PcName = pc.Name
        l.MachineID = pc.MachineID
```

优化之后再次进行压测。结果如下：



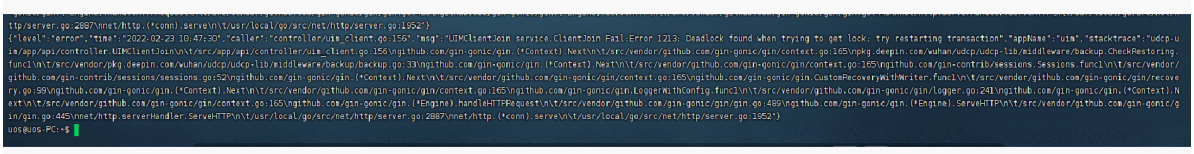
可以看出，通过对MySQL潜在的瓶颈进行处理优化，再次进入MySQL服务器中查询资源消耗可以发现CPU占用明显降低，呈现一个正常消耗状态，据此我们能够印证上述猜想。

猜想2的验证，我们进入到node机器的udcp_uim容器中，通过netstat命令下的连接占用

```
root@fd15dc8cad8c:/service# netstat -antp
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.11:33713        0.0.0.0:*               LISTEN      -
tcp        0      0 10.0.6.51:48612         10.0.6.11:6379          ESTABLISHED 1/api
tcp        0      0 10.0.6.51:48616         10.0.6.11:6379          ESTABLISHED 1/api
tcp        0      0 10.0.6.51:40414         10.0.6.36:9103          ESTABLISHED 1/api
tcp        0      0 10.0.6.51:42700         10.0.6.34:9110          TIME_WAIT   -
tcp        0      0 10.0.6.51:41452         10.0.6.24:9107          ESTABLISHED 1/api
tcp        0      0 10.0.6.51:41184         10.0.6.28:9109          ESTABLISHED 1/api
tcp        0      0 10.0.6.51:56918         10.0.6.38:9106          ESTABLISHED 1/api
tcp        0      0 10.0.6.51:47458         10.0.6.62:9100          ESTABLISHED 1/api
tcp        0      0 10.0.6.51:41020         10.0.6.28:9109          TIME_WAIT   -
tcp        0      0 10.0.6.51:41094         10.0.6.28:9109          TIME_WAIT   -
tcp        0      0 10.0.6.51:42774         10.0.6.34:9110          TIME_WAIT   -
tcp        0      0 10.0.6.51:42864         10.0.6.34:9110          ESTABLISHED 1/api
tcp        0      0 10.0.6.51:41286         10.0.6.24:9107          TIME_WAIT   -
tcp        0      0 10.0.6.51:56752         10.0.6.38:9106          TIME_WAIT   -
tcp        0      0 10.0.6.51:56824         10.0.6.38:9106          TIME_WAIT   -
tcp        0      0 10.0.6.51:48614         10.0.6.11:6379          ESTABLISHED 1/api
tcp        0      0 10.0.6.51:41362         10.0.6.24:9107          TIME_WAIT   -
tcp        0      0 10.0.6.51:49404         10.0.6.11:6379          ESTABLISHED 1/api
tcp6       0      0 :::9108                 :::*                    LISTEN      1/api
tcp6       0      0 :::9005                 :::*                    LISTEN      1/api
tcp6       0      0 10.0.6.51:9108          10.0.6.61:37922         ESTABLISHED 1/api
root@fd15dc8cad8c:/service#
```

通过结果可见连接和端口占用属于正常状态，由于以前版本已经将udcp_uim到api_auth的rpc请求由原生http更改为了grpc，端口得以复用，因此不存在端口长时间无法释放的问题。由此可以排除猜想2。

猜想4的验证：在udcp_uim的错误日志中排查，发现确实存在MySQL数据库的deadlock异常错误，且出现时间和压测出现的时间重合，量也很大



该日志抛出了错误，但是对应语句没有反应在日志中，因此需要自主定位到可能的位置，我们前往MySQL查看数据库日志可以发现有报错如下，

```
------------------------
LATEST DETECTED DEADLOCK
------------------------
2022-03-09 09:33:21 0x7f9ed8681700
*** (1) TRANSACTION:
TRANSACTION 24429, ACTIVE 0 sec inserting
MySQL tables in use 1, locked 1
LOCK WAIT 5 lock struct(s), heap size 1128, 3 row lock(s)
MySQL thread id 661, OS thread handle 140320211646208, query id 30718 172.18.0.1
root Update
INSERT INTO `udcp_uim`.`dcmc_pc`(`id`, `machine_id`, `host_name`, `name`,
`department_id`, `remark`, `os`, `status`, `created_at`, `update_admin_id`,
`updated_at`, `need_upgrade`, `upgrade_at`, `product_name`,
`client_need_upgrade`, `minor_version`, `client_version`, `deleted_at`,
`platform`, `arch`, `version`, `online`, `last_login_time`, `last_logout_time`,
`active_state`, `developer_mode`, `ip`, `mac`, `sn`, `user_name`, `join_ip`,
`last_login_user_id`, `last_login_full_name`) VALUES (400120,
'1d254d8e1011c4868671fa7a15', 'user_pc15', 'user_pc15', 200166, '', 'UOS 20
Professional', 0, '2022-02-16 15:08:20', 1, '2022-03-03 09:59:34', 0, '2022-02-16
15:08:20', '1.2', 0, '', '1.6.0.3-1', NULL, '1', 'amd', '1031', NULL, '2022-03-03
09:47:00', '2022-02-16 15:08:20', 1, 1, '10.20.15.110', 'F4:B5:20:26:C1:C8',
'e4eecad3189526cee212e1a3248522b0f9e29bb6f99753b317f26413f947e83b', 'xntest15',
'10.20.15.104', 0, NULL)
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 903 page no 3 n bits 72 index PRIMARY of table
`udcp_uim`.`dcmc_pc` trx id 24429 lock_mode X insert intention waiting
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
0: len 8; hex 73757072656d756d; asc supremum;;

*** (2) TRANSACTION:
TRANSACTION 24428, ACTIVE 1 sec inserting
MySQL tables in use 1, locked 1
7 lock struct(s), heap size 1128, 4 row lock(s)
MySQL thread id 660, OS thread handle 140320212260608, query id 30715 172.18.0.1
root Update
INSERT INTO `udcp_uim`.`dcmc_pc`(`id`, `machine_id`, `host_name`, `name`,
`department_id`, `remark`, `os`, `status`, `created_at`, `update_admin_id`,
`updated_at`, `need_upgrade`, `upgrade_at`, `product_name`,
`client_need_upgrade`, `minor_version`, `client_version`, `deleted_at`,
`platform`, `arch`, `version`, `online`, `last_login_time`, `last_logout_time`,
`active_state`, `developer_mode`, `ip`, `mac`, `sn`, `user_name`, `join_ip`,
`last_login_user_id`, `last_login_full_name`) VALUES (400120,
'1d254d8e1011c4868671fa7a15', 'user_pc15', 'user_pc15', 200166, '', 'UOS 20
Professional', 0, '2022-02-16 15:08:20', 1, '2022-03-03 09:59:34', 0, '2022-02-16
15:08:20', '1.2', 0, '', '1.6.0.3-1', NULL, '1', 'amd', '1031', NULL, '2022-03-03
09:47:00', '2022-02-16 15:08:20', 1, 1, '10.20.15.110', 'F4:B5:20:26:C1:C8',
'e4eecad3189526cee212e1a3248522b0f9e29bb6f99753b317f26413f947e83b', 'xntest15',
'10.20.15.104', 0, NULL)
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 903 page no 3 n bits 72 index PRIMARY of table
`udcp_uim`.`dcmc_pc` trx id 24428 lock mode S
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
0: len 8; hex 73757072656d756d; asc supremum;;
```

```
*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 903 page no 3 n bits 72 index PRIMARY of table
`udcp_uim`.`dcmc_pc` trx id 24428 lock_mode X insert intention waiting
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
0: len 8; hex 73757072656d756d; asc supremum;;

*** WE ROLL BACK TRANSACTION (1)
------------
TRANSACTIONS
------------
Trx id counter 24444
Purge done for trx's n:o < 24444 undo n:o < 0 state: running but idle
History list length 8
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 421795803654520, not started
0 lock struct(s), heap size 1128, 0 row lock(s)
---TRANSACTION 421795803646088, not started
0 lock struct(s), heap size 1128, 0 row lock(s)
---TRANSACTION 24438, ACTIVE 4 sec
4 lock struct(s), heap size 1128, 3 row lock(s), undo log entries 1
MySQL thread id 660, OS thread handle 140320212260608, query id 30734 172.18.0.1
root
---TRANSACTION 24439, ACTIVE 3 sec inserting
MySQL tables in use 1, locked 1
LOCK WAIT 3 lock struct(s), heap size 1128, 2 row lock(s)
MySQL thread id 661, OS thread handle 140320211646208, query id 30737 172.18.0.1
root Update
INSERT INTO `udcp_uim`.`dcmc_pc`(`id`, `machine_id`, `host_name`, `name`,
`department_id`, `remark`, `os`, `status`, `created_at`, `update_admin_id`,
`updated_at`, `need_upgrade`, `upgrade_at`, `product_name`,
`client_need_upgrade`, `minor_version`, `client_version`, `deleted_at`,
`platform`, `arch`, `version`, `online`, `last_login_time`, `last_logout_time`,
`active_state`, `developer_mode`, `ip`, `mac`, `sn`, `user_name`, `join_ip`,
`last_login_user_id`, `last_login_full_name`) VALUES (400120,
'1d254d8e1011c4868671fa7a15', 'user_pc15', 'user_pc15', 200166, '', 'UOS 20
Professional', 0, '2022
------- TRX HAS BEEN WAITING 3 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 903 page no 3 n bits 72 index PRIMARY of table
`udcp_uim`.`dcmc_pc` trx id 24439 lock mode S locks rec but not  waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 35; compact format; info bits 0
0: len 4; hex 00061af8; asc     ;;
1: len 6; hex 000000005f76; asc     _v;;
2: len 7; hex fc000001d32fd1; asc      / ;;
3: len 30; hex 3164323534643865313031316334383638363731666137613131520202020; asc
1d254d8e1011c4868671fa7a15    ; (total 64 bytes);
4: len 9; hex 757365725f70633135; asc user_pc15;;
5: len 9; hex 757365725f70633135; asc user_pc15;;
6: len 4; hex 80030de6; asc     ;;
7: len 0; hex ; asc ;;
8: len 19; hex 554f532032302050726f66657373696f6e616c; asc UOS 20 Professional;;

----------------------
BUFFER POOL AND MEMORY
----------------------
Total large memory allocated 291241984
Dictionary memory allocated 252144
Buffer pool size   16384
Free buffers       15502
```

```
Database pages     880
Old database pages 344
Pages read 749, created 131, written 160
0.04 reads/s, 0.00 creates/s, 0.50 writes/s
Buffer pool hit rate 995 / 1000, young-making rate 0 / 1000 not 0 / 1000
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 880, unzip_LRU len: 0
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
--------------
ROW OPERATIONS
--------------
0 queries inside InnoDB, 0 queries in queue
0 read views open inside InnoDB
Process ID=1, Main thread ID=140320365328128, state: sleeping
Number of rows inserted 3, updated 0, deleted 2, read 427
0.04 inserts/s, 0.00 updates/s, 0.04 deletes/s, 0.08 reads/s
Number of system rows inserted 0, updated 0, deleted 0, read 0
0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
----------------------------
END OF INNODB MONITOR OUTPUT
============================
```

经过对该LATEST DETECTED DEADLOCK错误日志的研判，和代码逻辑排查，初步判断为下列逻辑语句存在死锁问题：

```go
// ClientJoin 终端加域
func ClientJoin(clientReq request.ClientJoinReq, ip string) (err error) {
    return model.GetMasterDB().Transaction(func(tx *gorm.DB) error {
        //先删除记录再新建
        err := tx.Unscoped().Where( query: "machine_id = ?", clientReq.MachineID).Delete(&model.Pc{}).Error
        if err != nil { err }

        var pc model.Pc
        pc.MachineID = clientReq.MachineID
        pc.Name = clientReq.Name
        pc.DepartmentID = 1
        pc.IP = ip
        pc.Mac = clientReq.Mac
        if err = tx.Create(&pc).Error; err != nil {
            return err
        }
        return nil
    })
```

该处逻辑为，首先开启事务，然后根据machine_id（唯一索引）删除记录，最后创建一条记录并完成事务。初看此处感觉问题不大。因此我们需要验证其原因。

我们开始模拟两个事务，尝试复现deadlock现象，步骤如下

A ： T1 Start Transaction ;

B ： T2 Start Transaction ;

C ： T1 delete from `dcmc_pc` where machine_id = "none_exit_id"; #删除一条不存在的machine_id数据

D ： T2 delete from `dcmc_pc` where machine_id = "none_exit_id"; #删除一条不存在的machine_id数据

E ： T1 insert into `dcmc_pc` (machine_id....) values (...);

F ： T2 insert into `dcmc_pc` (machine_id...) values (...);

经过上述步骤后，确实出现了deadlock的问题，因此我们可以印证该猜想。

经过资料查验和数据库日志分析，我们可以得出下列原理解释：

delete语句使用到的machine_id虽然是唯一索引，但是当删除一条不存在的machine_id时，会申请间隙锁，该索引由于是varchar类型，因此该间隙范围是(max,+∞），由上述错误日志的 （0: len 8; hex 73757072656d756d; asc supremum;;）信息可以看出，supremum为无穷大的意思。

insert语句会对插入的行加一个排他锁，但是在插入这个行的过程之前，会设置一个Insert intention的间隙锁，叫做Insert intention锁。以上面的例子来说，在执行 insert into `dcmc_pc` (machine_id...) values (...)的时候，由于存在machine_id的索引，所以会对这个索引的(max,+∞)增加一个Insert Intention 类型的排他锁。

执行C语句完毕，T1持有了间隙(max,+∞)的排他锁；

执行D语句，T2 申请间隙(max,+∞)的排他锁，根据“precise mode”兼容矩阵，该申请被授权，所以T2 持有了间隙(max,+∞)的排他锁。

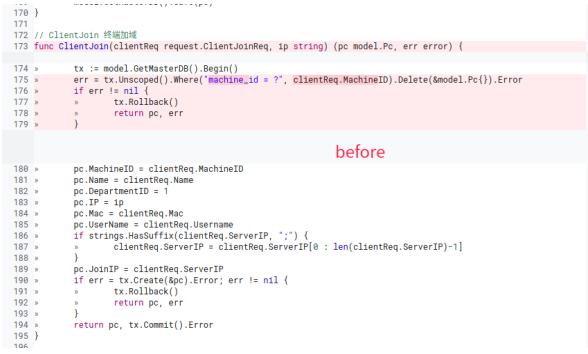
执行E语句，T1 申请Insert Intention (max,+∞)的排他锁，根据“precise mode”兼容矩阵，由于T2持有间隙(max,+∞)的排他锁，该申请被T2 block。

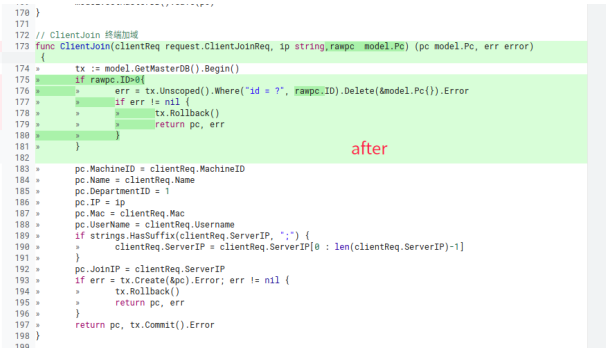
执行F语句，T2 申请 Insert Intention(max,+∞)的排他锁，根据“precise mode”兼容矩阵，由于T1持有间隙(max,+∞)的排他锁，该申请被T1 block。

这里一个死锁很明显的出现，T1与T2都持有一个锁，同时都在等对方释放一个锁。到这里，整个死锁的原因分析清楚了。

tx1	tx2	结果	分析
begain			
	begain		
delete from `dcmc_pc` where machine_id = "none_exit_id1";		Affected rows: 0, Time: 0.002000s	事务1对(max,+∞)区间申请间隙锁
	delete from `dcmc_pc` where machine_id = "none_exit_id2";	Affected rows: 0, Time: 0.002000s	事务2对(max,+∞)区间申请间隙锁
insert into `dcmc_pc` (machine_id....) values (...);			事务1对(max,+∞)申请插入意向锁，被事务2阻塞
	insert into `dcmc_pc` (machine_id....) values (...);	1213 - Deadlock found when trying to get lock; try restarting transaction, Time: 0.008000s	事务2对(max,+∞)申请插入意向锁，被事务1阻塞，出现死锁

如上表格我们的分析就十分清楚了，再回到我们的实际业务中，知道了死锁的原因后我们就进行了如下的改造。

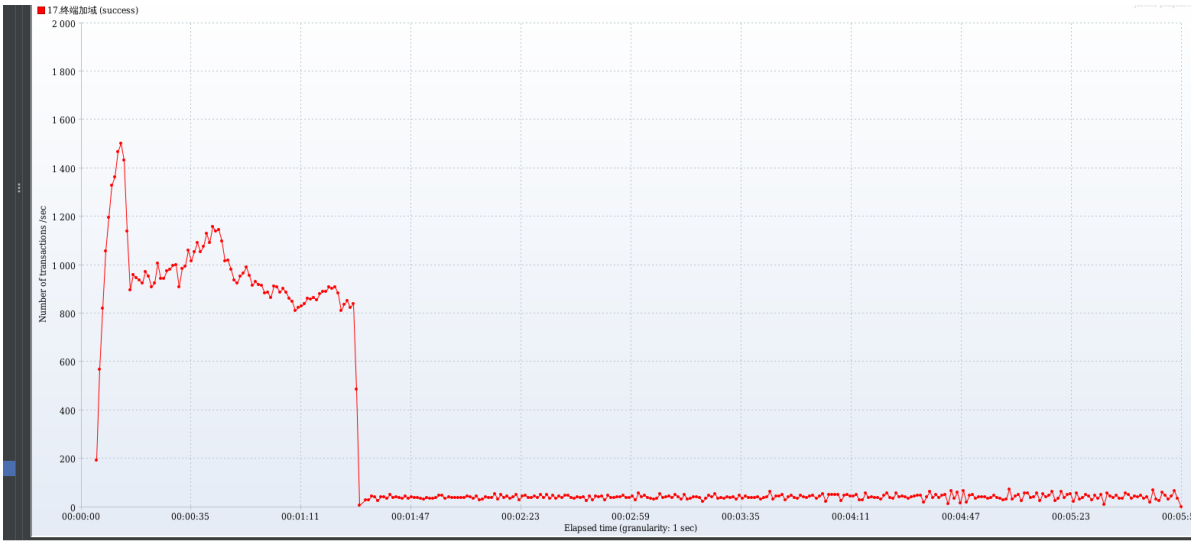




rawpc 参数 为在上层预先根据machine_id 硬查询了该条pc数据，因此只有确实存在pc数据的记录才会进入到接下来的事务流程（case :rawpc.ID<=0），这样处理能够避免删除不存在的数据，也就不会触发delete操作申请间隙锁，

而当有数据时（case:rawpc.ID>0），进入到真实delete操作，由于id列是主键索引，delete操作只会申请该条记录的行锁，不会再影响之后不同事务的意向插入锁的申请，因此也不会造成数据库的deadlock。

当修改此处sql语句后，再使用相同的样本数据进行压力测试多次，均无任何deadlock错误报出，该死锁异常问题得以解决。

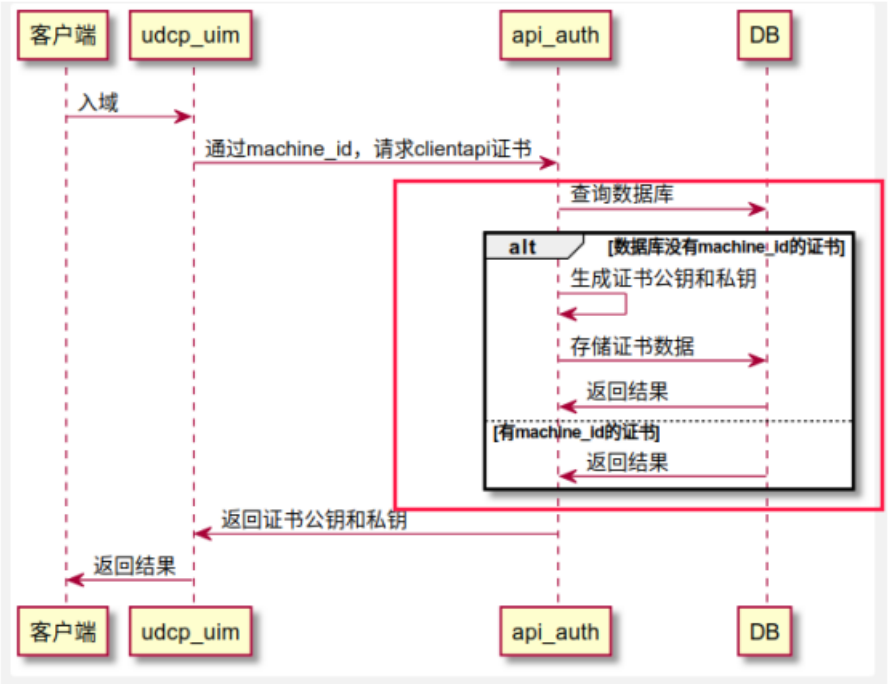


经过上述的优化和修改，可以看到压测不再出现错误，并且tps也因为MySQL的资源占用变低而升高了50%左右，但是仍然存在请求数量达到大约45000次后，tps骤减的问题。

至此，我们已经将所有的对于MySQL和端口的怀疑进行了一一解决和排除，但是依然存在的这种后期tps爆低的情况。

当所有的原因被排除后，我们仅剩下最后一处看似非常合理的逻辑，也就是rpc请求证书

如图(1) 所述，api_auth项目的中的逻辑如下红框所述

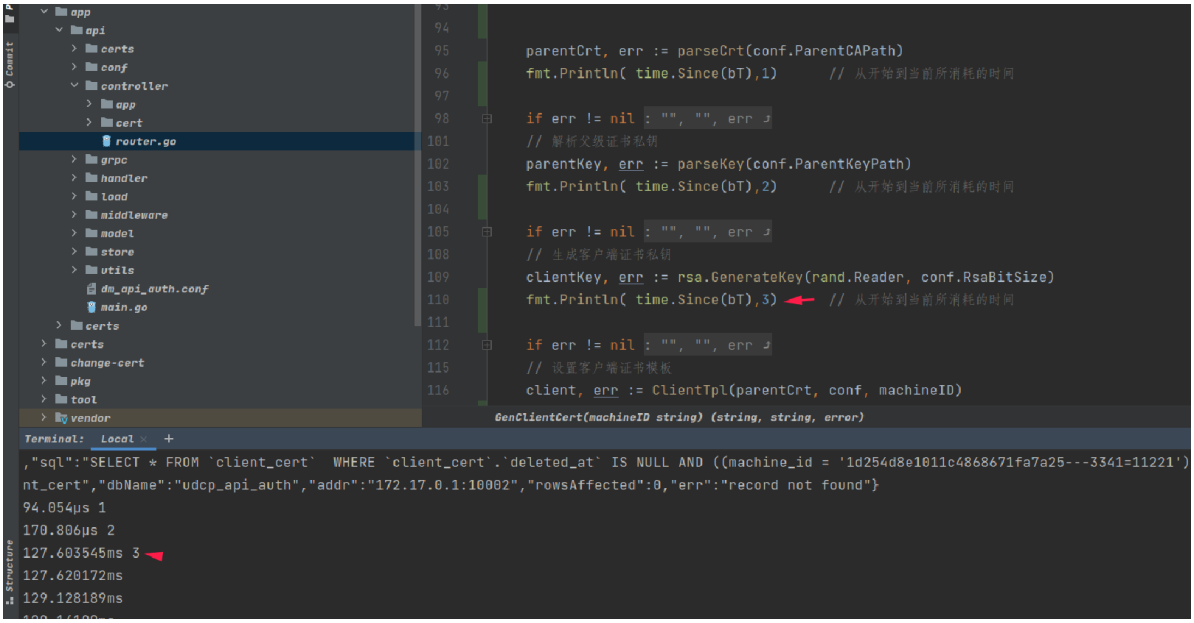


逻辑1：当machine_id对应的证书存在时，会直接从MySQL读取该证书并返回给rpc请求

逻辑2：当machine_id对应的证书不存在时，会【生成公私钥证书】，并存储该证书至Mysql，

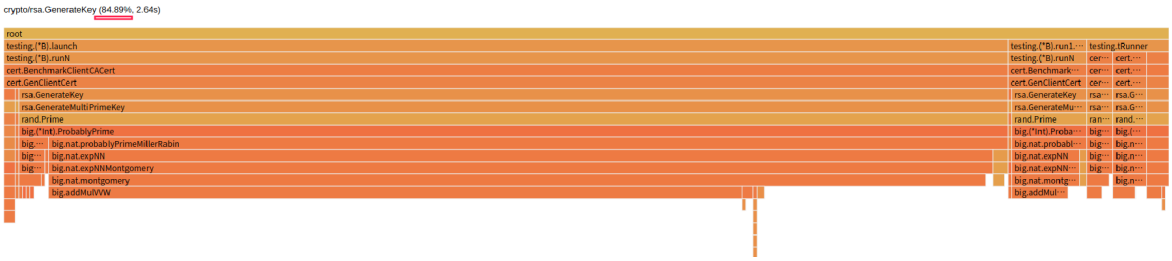
由此，我们会新增一个合理猜想，会不会是【生成公私钥证书】这一步非常耗时呢，同时又因为逻辑1的存在，导致我们压测的时候由于前面的请求都是已存在证书的machine_id，因此只需要读取MySQL，从而导致的tps非常高，而且每次似乎到了45000次左右的时候开始tps骤减，因为从大约45000次请求后开始就是不存在证书的machine_id呢，所以需要经过逻辑2，如果逻辑2是一个很耗时的操作呢。如果按照上述描述，似乎就能解释的通这奇怪的现象了。

于是我们进行了对该猜想的验证，我们在逻辑2中生成【生成公私钥证书】这一步新增了耗时的打印操作



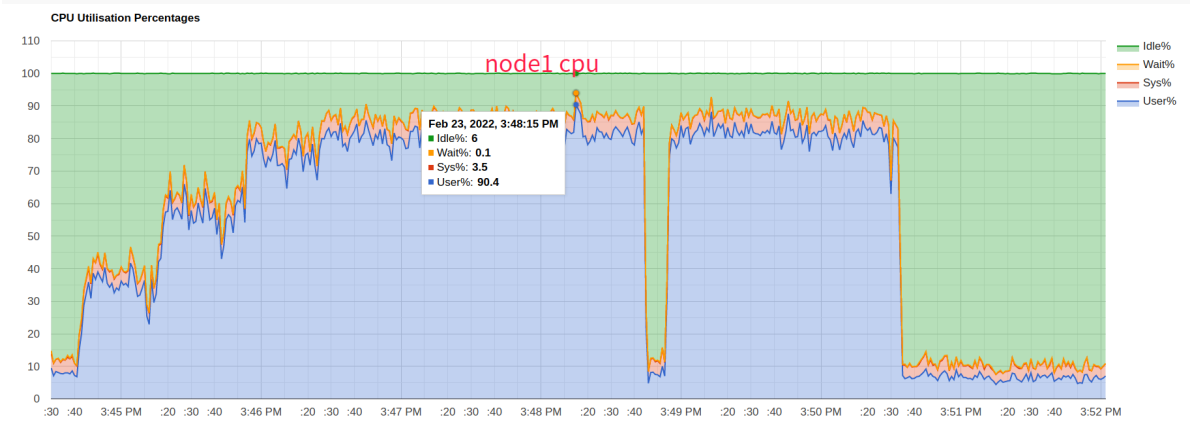
结果非常明显，第1步和第2步的耗时仅为微秒级别，基本可以忽略不计，而第3步（rsa.GenerateKey）耗时非常严重，达到了惊人的127毫秒，之后的步骤虽然也达到了毫秒级别，但都仅仅为一至二毫秒，因此第3步功能和该逻辑下的其他功能的耗时不在一个数量级，是其他操作的数百数千倍的耗时，这还是在没有压测的情况下。在真实的压测环境下，该差距可能更大。

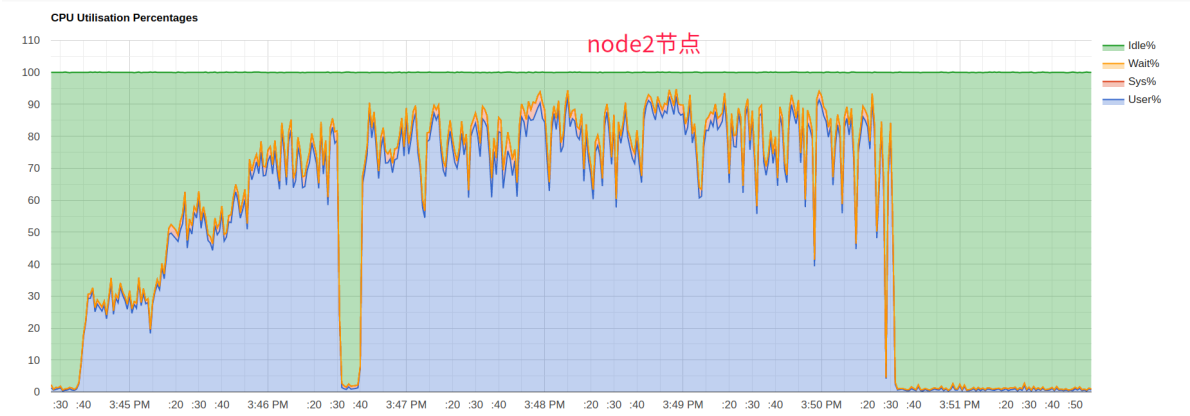
rsa.GenerateKey是官方库中的一个函数，用于生成一对公司钥，类似于linux下的ssh-keygen命令，为了确定是这个函数的耗时异常，我们还通过golang的benchmark进行了该【生成公私钥证书】方法的pprof调试，并制成了火焰图：



通过火焰图，也可以明显的看到，该函数占用了非常高的cpu资源。

同时，既然是api_auth服务的异常情况，自然也会反应在服务器的资源消耗上，我们前往node 服务器查看相关资源消耗，同样得出了验证我们猜想的结果：





负载均衡的两个node服务器，无一例外都是cpu基本跑满。

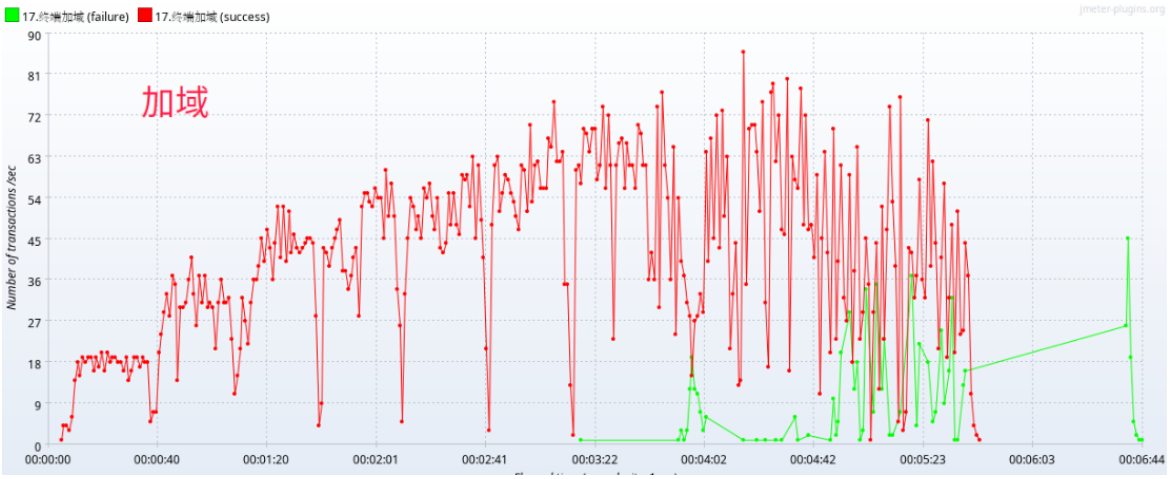
5. 结论

综上：我们可以得出结论，本接口中的【生成公私钥证书】方法属于**cpu密集型程序**，由于加域接口为同步请求接口，因此每个请求都需要等待【生成公私钥证书】完成才会得到返回，而【生成公私钥证书】这一操作非常耗时，这也是为什么后期tps一直稳定很小的原因，而前期tps很高的原因，自然就是该接口针对已有证书数据的直接查询与返回，没有经过【生成公私钥证书】流程所致。至于之前我们怀疑的由于端口限制造成的每次压测到45000次左右就开始tps骤减，其原因就是数据库中已经存在了大约45000个终端的证书数据，当开始需要即时生成证书时，tps就会开始因为【生成公私钥证书】这一耗时操作，变得很低。

三.实验验证

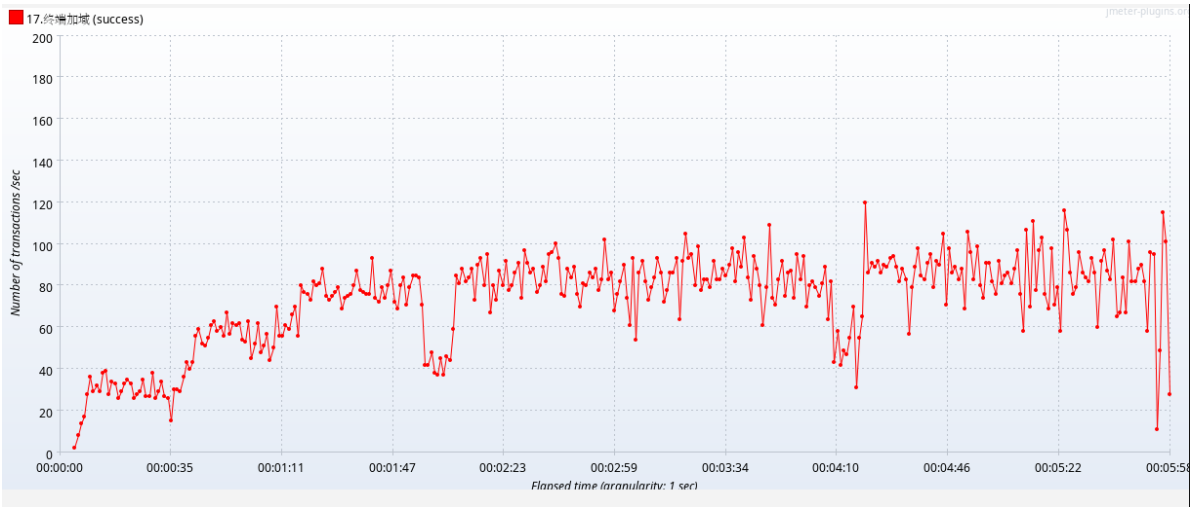
1. 对比验证

优化接口前：



我们将所有影响因素踢出，主要是已生成的证书影响，将数据库进行了清空。

优化接口后：



2. 实验结果

版本	错误率	TPS
修改前	40%	40-65
修改后	0%	80-120

功能：原有功能不受影响。

性能：TPS提升约200%。

稳定性：错误率从40%降低为0%。

四.解决方案

经过上述的猜想和验证，我们明确了加域接口的报错来源，并进行了修复，

我们也将很多造成不必要性能浪费的语句进行了精简和优化，

同时我们也明确了tps非常低的原因，即【生成公私钥证书】方法属于**cpu密集型程序**，由于加域接口为同步请求接口，因此每个请求都需要等待【生成公私钥证书】完成才会得到返回。

由于客户端需要即时获取到证书，所以该接口必须为同步请求，无法更改为异步，因此我们考虑的解决方案就是将耗时程序前置，我们不要在请求来的时候才去即时演算出公私密钥对，而是在程序启动的时候启动协程循环往密钥对channel中存储生成出来的密钥对，同时设置一个有限的缓冲区，这样在有加域请求到来时，可以直接返回已经生成好的密钥对，再使用该密钥对进行剩余的业务逻辑操作即可。示例代码如下：

```
var GlobalKeyChan chan *rsa.PrivateKey
var MaxGlobalKeyChan int=50000

// InitKeyChan 初始化公私钥对的channel
func InitKeyChan(){
    GlobalKeyChan=make(chan *rsa.PrivateKey,MaxGlobalKeyChan)
    conf := &certConf.Client
    for {
        // 生成客户端证书私钥
        clientKey, err := rsa.GenerateKey(rand.Reader, conf.RsaBitSize)
        if err!=nil{
            log.Error( msg: "GenerateKey",log.FieldErr(err))
            continue
        }
        GlobalKeyChan<-clientKey
    }
}
```

加域的使用场景为，首先在平台创建人员和部门数据，有了人员数据，在终端进行加域操作，我们的优化改造即为在api_auth项目启动后（负载均衡部署则多实例同时进行）开始循环生成并保存密钥对数据。这样当channel中存在已生成好的密钥对时，加域接口的平均响应时间会指数级降低，同时tps也能稳定在高位，轻松达到500tps的目标性能。

五.小结

加域接口虽然对外是一个单独的接口暴露，但并不是一个完全独立的业务逻辑，而是依赖了相当多的外部因素，每个外部因素由于各种历史原因，又带来了其或多或少，或严重或隐蔽的问题，最终糅合在了一起，需要每个疑难杂症各个击破，加上一些曾经的错误测试的结果，稳定高tps的错误结果（全是已有证书的情况造成tps稳定高位的假象）对我们的排查造成误导，使解决问题走了不少弯路。

该接口优化过程相对于其他接口的性能优化过程，耗时最久，情况最为复杂，现象最为异常，兜兜转转排查出了很多问题，但最终直接影响最深的确是一开始没有怀疑的点，虽然猜想的每一项都是确确实实存在的问题，但始终并非加域接口指标低tps的核心因素。经验虽然能够帮助快速定位，合理怀疑问题点等，但也可能带来一些经验主义陷阱。

合理的服务端应用一向都是写少读多（28原则甚至更甚），对于服务端接口，绝大部分性能不足都是由于MySQL带来的性能瓶颈，所以才会有数据库索引的重要性，才会有队列，数据库集群，redis，memcache这类关键缓存工具来帮助服务端承受更大压力，我们绝大部分人都会优先考虑到这类问题，MySQL读写性能，端口占用是否爆仓等等，虽然一些问题确实存在，但是最终的关键因素还是一个不起眼，早早被忽略的官方函数。