



华中科技大学

数据库系统原理实践报告

综合设计题目： 电信收费管理系统软件

姓 名： 刘本嵩

学 院： 计算机科学与技术学院

专 业： 计算机科学与技术

班 级： CS1601

学 号： U201614531

指导教师： 潘鹏

分数	
教师签名	

2019 年 6 月 18 日

目 录

1 课程任务概述.....	1
2 软件功能学习部分.....	2
2.1 任务要求.....	2
2.2 完成过程.....	2
2.3 任务总结.....	3
3 SQL 练习.....	4
3.1 任务要求.....	4
3.2 完成过程.....	4
3.3 任务总结.....	15
4 综合实践任务.....	16
4.1 系统设计目标.....	16
4.2 需求分析.....	16
4.3 总体设计.....	17
4.4 数据库设计.....	18
4.5 详细设计与实现.....	20
4.6 系统构建、部署与测试.....	25
4.7 系统设计与实现总结.....	44
5 课程总结.....	46
附录 参考文献.....	47

1 课程任务概述

本次数据库系统原理课程实验的任务共有 3 个,前两个任务为对于数据库基础的认知实验,最后一个是一个综合实践。3 个实验的任务难度逐级递增,在前两个实验打下良好基础的情况下最终实现一个 C/S 模式的成绩管理系统。

实验任务一：软件功能学习

本部分通过对 SQL 工具的学习,从而了解 SQL 服务器的基本使用方法。

实验任务二：SQL 实验

熟悉 DBMS 软件的安装以及使用,并且能够使用 SQL 语言进行表的创建、添加、删除、修改、查询等操作。对于查询的要求较高。此外还需要对于数据库事务有一定的了解,能够了解在不同的事务隔离级别进行数据库操作的异同。

实验任务三：综合实践任务

选定一个数据库应用的题目,并在题目限定的应用场景下完成需求分析,数据库设计和应用程序设计,并提交相应的文档。本实验选定的场景为采用 B/S 或 C/S 模式实现一个电信收费管理系统软件,实现电信套餐种类、用户信息、客服代表、收款员等信息的管理。

要求：

- 1) 实现不同权限的浏览和更新。
- 2) 实现用户扣、缴费情况及帐户余额的查询。
- 3) 实现欠款用户使用状态的自动改变。
- 4) 实现客服代表的业绩统计功能。
- 5) 提供至少两种风格的查询报表。

2 软件功能学习部分

2.1 任务要求

练习 SQL 的使用。

- 1) 练习 sqlserver 的两种完全备份方式：数据和日志文件的脱机备份、系统的备份功能。
- 2) 练习在新增的数据库上增加用户并配置权限的操作，通过用创建的用户登录数据库并且执行未经授权的 SQL 语句验证自己的权限配置是否成功。

2.2 完成过程

2.2.1 练习 SQL 的备份

最简单的数据备份方式是直接在单机数据库上运行 db_dump 进行备份。对于热备份的情形，可以直接使用 db_hotbackup 进行备份。详情请参考

https://docs.oracle.com/cd/E17275_01/html/api_reference/C/db_hotbackup.html

对于多机或集群上的数据库备份，在此不做讨论。

如果需要在生产环境下备份 log，你需要设置 log_destination 或 log_directory，log_filename 等参数，并且使用 unix 标准备份工具，即可进行备份。详情请阅读

<https://www.postgresql.org/docs/9.1/runtime-config-logging.html>

在正常的 SQL 服务器中不存在“脱机备份”和“系统的备份功能”的概念，因此无法进行进一步阐述。

2.2.2 PostgreSQL 权限管理

Install the postgresql package. It will also create a system user called postgres.

Then login with user `postgres` to configure the database: `sudo -iu postgres`

首先初始化数据库。

```
[postgres]$ initdb --locale en_US.UTF-8 -E UTF8 -D '/var/lib/postgres/data'
```

然后创建我的第一个用户 recolic 并授予 superuser 权限。

```
[postgres]$ createuser --interactive
```

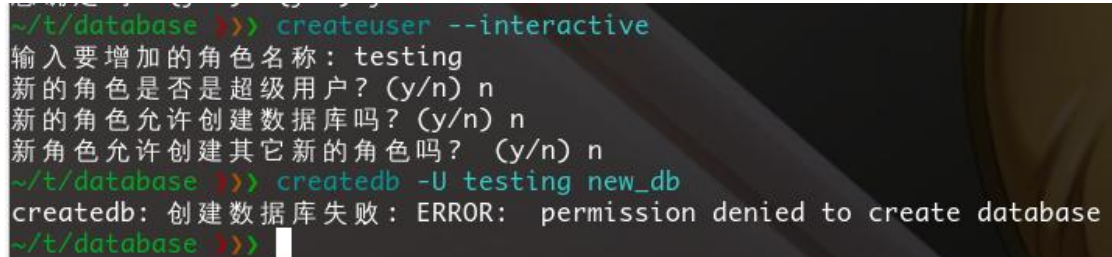
并创建一个数据库。

```
$ createdb myDatabaseName
```

使用以下命令访问 postgresql shell:

```
[postgres]$ psql -d myDatabaseName
```

如图，createuser 成功的限制了用户的权限。这里的用户对应 unix 用户。



```
~/t/database >>> createuser --interactive
输入要增加的角色名称: testing
新的角色是否是超级用户? (y/n) n
新的角色允许创建数据库吗? (y/n) n
新角色允许创建其它新的角色吗? (y/n) n
~/t/database >>> createdb -U testing new_db
createdb: 创建数据库失败: ERROR: permission denied to create database
~/t/database >>>
```

2.3 任务总结

通过此次任务，我成功学习了 SQL 的设计理念和基本用法，丰富了关系型数据库的使用知识，了解了关系型数据库的基本使用方法。

3 SQL 练习

3.1 任务要求

3.2 完成过程

3.2.1 建表

```
create table my_users(  
    uid integer not null unique primary key,  
    name varchar,  
    sex varchar,  
    byear integer,  
    city varchar  
);
```

----- 1

```
create table mblog(  
    bid integer not null unique primary key,  
    title varchar,  
    uid integer references my_users,  
    pyear integer,  
    pmonth integer,  
    pday integer,  
    cont varchar  
);
```

```
create table thumb(  
    uid integer references my_users,  
    bid integer references mblog,  
    primary key (bid, uid)  
);
```

结果如图。

```

my_default_db=# drop table fans_3, follow, thumb, mblog, my_users;

DROP TABLE
my_default_db=# --- cleanup done
my_default_db=#
my_default_db=# create table my_users(
my_default_db(#      uid integer not null unique primary key,
my_default_db(#      name varchar,
my_default_db(#      sex varchar,
my_default_db(#      byear integer,
my_default_db(#      city varchar
my_default_db(# );
CREATE TABLE
my_default_db=# ----- 1
my_default_db=# create table mblog(
my_default_db(#      bid integer not null unique primary key,
my_default_db(#      title varchar,
my_default_db(#      uid integer references my_users,
my_default_db(#      pyear integer,
my_default_db(#      pmonth integer,
my_default_db(#      pday integer,
my_default_db(#      cont varchar
my_default_db(# );
CREATE TABLE
my_default_db=# create table thumb(
my_default_db(#      uid integer references my_users,
my_default_db(#      bid integer references mblog,
my_default_db(#      primary key (bid, uid)
my_default_db(# );
CREATE TABLE
my_default_db=# \lt
无效的命令 \lt, 用 \? 查看帮助。
my_default_db=# \dt
          关联列表
 架构模式 | 名称 | 类型 | 拥有者
-----+-----+-----+-----
public   | mblog | 数据表 | recolic
public   | my_users | 数据表 | recolic
public   | shit  | 数据表 | recolic
public   | thumb | 数据表 | recolic
(4 行记录)

```

3.2.2 数据更新

1) 分别用一条 **sql** 语句完成对博文表基本的增、删、改的操作；

先增加一个用户 ID。

```
insert into my_users values (101, 'recolic', 'M', 2016, 'WEIHAI');
```

然后一条命令增加推文。

```
insert into mblog values (1, 'You mother sucks fuck you', 101,
```

1970, 1, 1, 'Hello world. This is testing content...');

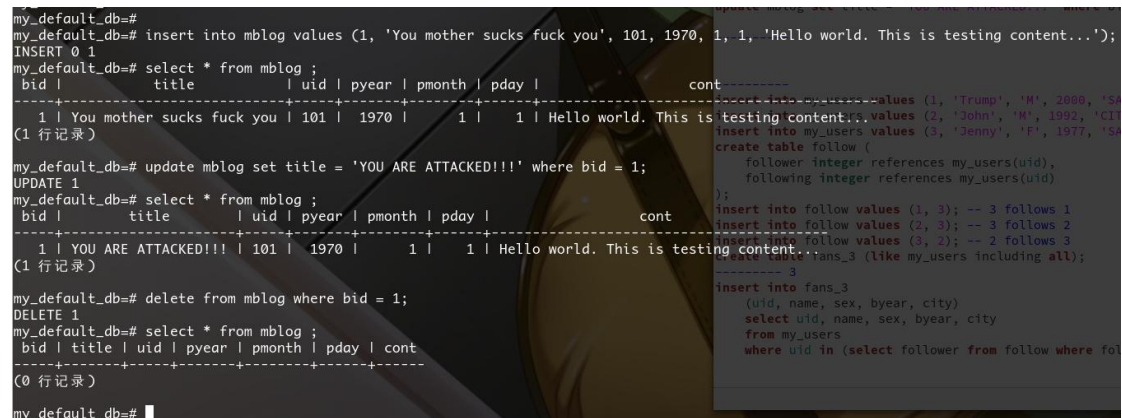
删除推文。

delete from mblog where bid = 1;

修改推文。

update mblog set title = 'YOU ARE ATTACKED!!!' where bid = 1;

结果如图



```
my_default_db=# insert into mblog values (1, 'You mother sucks fuck you', 101, 1970, 1, 1, 'Hello world. This is testing content...');
INSERT 0 1
my_default_db=# select * from mblog ;
bid | title | uid | pyear | pmonth | pday | cont
-----+-----+-----+-----+-----+-----+-----
1 | You mother sucks fuck you | 101 | 1970 | 1 | 1 | Hello world. This is testing content...
(1 行记录)

my_default_db=# update mblog set title = 'YOU ARE ATTACKED!!!' where bid = 1;
UPDATE 1
my_default_db=# select * from mblog ;
bid | title | uid | pyear | pmonth | pday | cont
-----+-----+-----+-----+-----+-----+-----
1 | YOU ARE ATTACKED!!! | 101 | 1970 | 1 | 1 | Hello world. This is testing content...
(1 行记录)

my_default_db=# delete from mblog where bid = 1;
DELETE 1
my_default_db=# select * from mblog ;
bid | title | uid | pyear | pmonth | pday | cont
-----+-----+-----+-----+-----+-----+-----
(0 行记录)

my_default_db=#
```

2) 批处理操作

先对环境做一些准备。

insert into my_users values (1, 'Trump', 'M', 2000, 'SAN FRANCISCO');

insert into my_users values (2, 'John', 'M', 1992, 'CITY OF INDUSTRY');

insert into my_users values (3, 'Jenny', 'F', 1977, 'SAN JOSE');

```
create table follow (
    follower integer references my_users(uid),
    following integer references my_users(uid)
);
```

insert into follow values (1, 3); -- 3 follows 1

insert into follow values (2, 3); -- 3 follows 2

insert into follow values (3, 2); -- 2 follows 3

create table fans_3 (like my_users including all);


```

my_default_db=# 
my_default_db=# -----
my_default_db=# insert into my_users values (1, 'Trump', 'M', 2000, 'SAN FRANCISCO');
INSERT 0 1
my_default_db=# insert into my_users values (2, 'John', 'M', 1992, 'CITY OF INDUSTRY');
INSERT 0 1
my_default_db=# insert into my_users values (3, 'Jenny', 'F', 1977, 'SAN JOSE');
INSERT 0 1
my_default_db=# create table follow (
my_default_db(#      follower integer references my_users(uid),
my_default_db(#      following integer references my_users(uid)
my_default_db(# );
CREATE TABLE
my_default_db=# insert into follow values (1, 3); -- 3 follows 1
INSERT 0 1
my_default_db=# insert into follow values (2, 3); -- 3 follows 2
INSERT 0 1
my_default_db=# insert into follow values (3, 2); -- 2 follows 3
INSERT 0 1
my_default_db=# create table fans_3 (like my_users including all);
CREATE TABLE
my_default_db=# ----- 3

```

然后进行批处理操作

insert into fans_3

```

    (uid, name, sex, byear, city)
    select uid, name, sex, byear, city
    from my_users
    where uid in (select follower from follow where following
= 3);

```

```

my_default_db=# ----- 3
my_default_db=# insert into fans_3
my_default_db(#      (uid, name, sex, byear, city)
my_default_db=#      select uid, name, sex, byear, city
my_default_db=#      from my_users
my_default_db=#      where uid in (select follower from follow where following = 3);
INSERT 0 2
my_default_db=# 

```

结果如图

```

my_default_db=# 
my_default_db=# select * from fans_3 ;
 uid | name  | sex | byear |      city
-----+-----+-----+-----+-----
   1 | Trump | M   | 2000 | SAN FRANCISCO
   2 | John  | M   | 1992 | CITY OF INDUSTRY
(2 行记录)

```

3) 数据导入导出

只需要使用 pg_dump 工具即可。这里的用户名是 recolic，数据库名字 my_default_db。

pg_dump -U recolic my_default_db > dbexport.pgsql

导入也一样，使用 pg_restore 工具。

4) 观察性实验

PostgreSQL 作为 Cluster SQL，不存在官方的图形界面。

5) 触发器实验

```
CREATE TRIGGER ThumbUpdate before update on THUMB
FOR EACH ROW
BEGIN
  IF(NEW.UID IN (SELECT UID FROM MBLOG WHERE NEW.BID=BID))
  THEN SIGNAL SQLSTATE 'HY001' SET MESSAGE_TEXT='Update
Error!'
  END IF;
END$$
```

```
CREATE TRIGGER ThumbInsert before insert on THUMB
FOR EACH ROW
BEGIN
  IF(NEW.UID IN (SELECT UID FROM MBLOG WHERE NEW.BID=BID))
  THEN SIGNAL SQLSTATE 'HY000' SET MESSAGE_TEXT='Insert
Error!'
  END IF;
END$$
```

3.2.3 查询

1) 查询“张三”用户关注的所有用户的 ID 号、姓名、性别、出生年份，所在城市，并且按照出生年份的降序排列，同一个年份的则按照用户 ID 号升序排列。

```
select * from my_users
where uid in (
  select following from follow
  where follower=4
) order by byear desc, uid asc;
```

```

postgres=#
postgres=# ----- 17 Questions
postgres=#
postgres=# select * from my_users
postgres=# where uid in (
postgres(# select following from follow
postgres(# where follower=4
postgres(# ) order by byear desc, uid asc;
 uid | name | sex | byear | city
-----+-----+-----+-----+-----
 1 | Trump | M | 2001 | SAN FRANCISCO
 2 | John | M | 1932 | CITY OF INDUSTRY
 3 | Jenny | F | 1932 | SAN JOSE
(3 行记录)

```

2) 查找没有被任何人点赞的博文 ID、标题以及发表者姓名，并将结果按照标题字符顺序排列。

```

select      mblog.bid,mblog.title,my_users.name      from
mblog,my_users
where mblog.bid not in (
    select bid from thumb
) and my_users.uid = mblog.uid
order by title asc;

```

```

INSERT 0 1
postgres=# select mblog.bid,mblog.title,my_users.name from mblog,my_users
postgres=# where mblog.bid not in (
postgres(# select bid from thumb
postgres(# ) and my_users.uid = mblog.uid
postgres=# order by title asc;
 bid | title | name
-----+-----+-----
 3 | FFFFFFFUUUUUCCCCCKKKKK | recolic
 2 | MOMMY MOMMY MOMMY MOMMY | recolic
(2 行记录)

```

3) 查找 2000 年以后出生的武汉市用户发表的进入过头条的博文 ID；

```

select mblog.bid from my_users, hot_posts, mblog
where
    my_users.byear > 2000 and
    my_users.city = 'WU HAN' and
    my_users.uid = mblog.uid and
    mblog.bid = hot_posts.bid;

```

```

postgres=# ----- Prepare Q3
postgres=# create table hot_posts(
postgres(#   bid integer references mblog primary key
postgres(# );
ERROR:  relation "hot_posts" already exists
postgres=# insert into hot_posts values (3);
INSERT 0 1
postgres=# ----- Q3
postgres=# select mblog.bid from my_users, hot_posts, mblog
postgres=# where
postgres=# my_users.byear > 1900 and
postgres=# my_users.city = 'WU HAN' and
postgres=# my_users.uid = mblog.uid and
postgres=# mblog.bid = hot_posts.bid;
   bid
-----
      3
(1 行记录)

```

4) 查找订阅了所有分类的用户 ID ;

```

select uid from sub
where array_length(chan, 1) = (select count(*) from chan);

```

```

postgres=# ----- Prepare Q4
postgres=# create table sub(
postgres(#   uid int,
postgres(#   chan int[]
postgres(# );
CREATE TABLE
postgres=# create table chan(
postgres(#   id int primary key,
postgres(#   name text
postgres(# );
CREATE TABLE
postgres=# insert into chan values (1, '文学');insert into chan values (2, '艺术');insert into chan values (3, '哲学');insert into chan values (4, '音乐');
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
postgres=# insert into sub values (101, '{1}');insert into sub values (1, '{1,2,3,4}');
INSERT 0 1
INSERT 0 1
postgres=# ----- Q4
postgres=# select uid from sub
postgres=# where array_length(chan, 1) = (select count(*) from chan);
   uid
-----
     1
(1 行记录)
postgres=# ----- Q1

```

5) 查找出生年份小于 1970 年或者大于 2010 年的用户 ID、出生年份、所在城市，要求 where 子句中只能有一个条件表达式；

```

select uid, byear, city from my_users
where byear not between 1970 and 2010;

```

```

postgres=# ----- Q1
select uid, byear, city from my_users
where byear not between 1970 and 2010;
 uid | byear |      city
-----+-----+-----
 101 |  2016 | WU HAN
    2 |  1932 | CITY OF INDUSTRY
    3 |  1932 | SAN JOSE
(3 行记录)
postgres=#

```

6) 统计每个城市的用户数；

```

select city, count(uid) from my_users
group by city;

```

```
postgres=# select city, count(uid) from my_users
postgres=# group by city;
    city    | count
-----+-----
WU HAN      |      1
N CAROLINA  |      1
SAN FRANCISCO |      1
SAN JOSE    |      1
CITY OF INDUSTRY |      1
(5 行记录)
```

7) 统计每个城市的每个出生年份的用户数，并将结果按照城市的升序排列，同一个城市按照出生用户数的降序排列其相应的年份；

```
select city,byear,count(uid) from my_users
group by city,byear
order by city asc,count(uid) desc;
```

```
postgres=# select city,byear,count(uid) from my_users
postgres=# group by city,byear
postgres=# order by city asc,count(uid) desc;
    city    | byear | count
-----+-----+-----
CITY OF INDUSTRY | 1932 |      1
N CAROLINA      | 1977 |      1
SAN FRANCISCO   | 2001 |      1
SAN JOSE        | 1932 |      1
WU HAN          | 2016 |      1
(5 行记录)
```

8) 查找被点赞数超过 10 的博文 ID 号；

```
select bid from thumb
group by bid having count(uid)>10;
```

```

postgres=# ----- Prepare Q8
postgres=# insert into thumb values (2, 2);insert i
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
postgres=# insert into thumb values (2, 2);insert i
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
postgres=# insert into thumb values (2, 2);insert i
INSERT 0 1
INSERT 0 1
postgres=# ----- Q8
postgres=# select bid from thumb
postgres-#      group by bid having count(uid)>10;
 bid
-----
    2
(1 行记录)

```

9) 查找被 2000 年后出生的用户点赞数超过 10 的博文 ID 号 ;

```

select bid from (
    select thumb.bid,thumb.uid from thumb,my_users
    where thumb.uid=my_users.uid and my_users.byear>2000
) as tmp
group by bid having count(tmp.bid)>10;

```

```

postgres=# ----- Q9
postgres=# select bid from (
postgres(# select thumb.bid,thumb.uid from thumb,my_users
postgres(# where thumb.uid=my_users.uid and my_users.byear>2000
postgres(# ) as tmp
postgres-# group by bid having count(tmp.bid)>10;
 bid
-----
    2
(1 行记录)

```

10) 查找被 2000 年后出生的用户点赞数超过 10 的每篇博文的进入头条的次数 ;

```

select hot_posts.bid,count(hot_posts.bid) from hot_posts
where bid in(
    select bid from(
        select bid,my_users.uid from thumb,my_users
        where thumb.uid=my_users.uid and my_users.byear>2000
    ) as tmp

```



```

        group by bid having count(tmp.uid)>10
    ) group by bid;

```

```

postgres=# select * from hot_posts ;
   bid
-----
     3
     2
(2 行记录)

postgres=# select hot_posts.bid,count(hot_posts.bid) from hot_posts
postgres=# where bid in(
postgres=# select bid from(
postgres=#

postgres=# select bid,my_users.uid from thumb,my_users
postgres=#

postgres=# where thumb.uid=my_users.uid and my_users.byear>2000
postgres=# ) as tmp
postgres=# group by bid having count(tmp.uid)>10
postgres=# ) group by bid;
   bid | count
-----+-----
     2 |      1
(1 行记录)

postgres=#

```

11) 查找订阅了文学、艺术、哲学、音乐中至少一种分类的用户 ID , 要求不能使用嵌套查询 , 且 where 子句中最多只能包含两个条件 ;

```

select uid from sub where '{"文学","艺术","哲学","音乐"}' &&
chan;

```

```

postgres=# select * from sub;
   uid |      chan
-----+-----
    101 | {文学}
     1 | {文学,艺术,哲学,音乐}
(2 行记录)

postgres=# select uid from sub where '{"文学","艺术","哲学","音乐"}' && chan;
   uid
-----
    101
     1
(2 行记录)

postgres=#

```

12) 查找标题中包含了“最多地铁站”和“_华中科技大学”两个词的博文基本信息 ;

```

select * from mblog where title like '%最多地铁站%' and title
like '%\_华中科技大学%';

```

```
postgres=# select * from mblog where title like '%最多地铁站%' and title like '%_华中科技大学%';
bid |          title          | uid | pyear | pmonth | pday |      cont
-----+-----+-----+-----+-----+-----+-----
  3 | 最多地铁站hello_华中科技大学 | 101 | 1970 |      1 |      3 | testing blog 3
(1 行记录)

postgres=#
```

13) 查找所有相互关注的用户对的两个 ID 号, 要求不能使用嵌套查询;

```
select a.follower,a.following from follow a, follow b
where a.follower=b.following and b.follower=a.following;
```

```
postgres=# select a.follower,a.following from follow a,follow b
postgres=# where a.follower=b.following and b.follower=a.following;
 follower | following
-----+-----
        2 |        3
        3 |        2
(2 行记录)

postgres=#
```

14) 查找好友圈包含了 5 号用户好友圈的用户 ID ;

```
select distinct UID from FRIENDS
where not exists(
select * from FRIENDS FRIEND1
where FRIEND1.UID=5 and not exists(
select * from FRIENDS FRIEND2
where FRIEND2.UID=FRIEND1.FUID)
);
```

15) 查找 2019 年 4 月 20 日每一篇头条博文的 ID 号、标题以及该博文的每一个分类 ID , 要求即使该博文没有任何分类 ID 也要输出其 ID 号、标题 ;

```
select TOPDAY.BID,MBLOG.TITLE,LID from MBLOG,TOPDAY
left outer join B_L on(B_L.BID=TOPDAY.BID)
where TOPDAY.TYEAR=2019 and TOPDAY.TMONTH=4 and
TOPDAY.TDAY=20 and TOPDAY.BID=MBLOG.BID;
```

16) 查找至少有 3 名共同好友的所有用户对的两个 ID 号。

```
select UID1,UID2,count(FUID)
from( select x.UID as UID1,y.UID as UID2,x.FUID from FRIENDS
x,FRIENDS y
where x.UID!=y.UID and x.FUID=y.FUID) as tmp
group by tmp.UID1,tmp.UID2
having count(FUID)>=3;
```

17) 创建视图: 查阅 DBMS 内部函数, 创建一个显示当日热度排名前十的微博

信息的视图，其中的属性包括：博文 ID、博文标题、发表者 ID、发表者姓名、被点赞数。

```
create view toplog(BID,TITLE,UID,NAME,thumb)
as
select
TOPDAY.BID,MBLOG.TITLE,MBLOG.UID,USER.NAME,count(THUMB.BI
D)
from TOPDAY,MBLOG,USER,THUMB
where TOPDAY.BID=MBLOG.BID and MBLOG.UID=USER.UID and
TOPDAY.BID=THUMB.BID
group by BID;
```

3.3 任务总结

通过此次任务，我成功学习了 SQL 的设计理念和基本用法，丰富了关系型数据库的使用知识，了解了关系型数据库的基本使用方法。了解了 NoSQL 和 SQL 的优缺点。

本次任务所有源码放置在

[https://github.com/recolic/hust-database/blob/master/etc/db_cmds.s
ql](https://github.com/recolic/hust-database/blob/master/etc/db_cmds.sql)

如与报告不符，以 github 中源码为最终版本。

4 综合实践任务

立即查看代码：<https://github.com/recolic/hust-database>

立即查看成品：<https://hustdb.recolic.net/>

4.1 系统设计目标

电信部门业务多，业务量大，如果没有一套满足业务需求的收费系统，将会阻碍业务的发展。因此，本系统的实现可以对电信收费的工作起到很大的促进作用。本论文首先通过与电信各个相关部门的业务经理进行交流，了解电信收费对系统的功能需求，并结合通信行业对电信收费管理系统的开发和研究所形成的经验和理论来对系统进行设计与开发。本系统所采用的主要编程技术包括编程语言和数据库开发，服务端使用的编程语言是 Go，而客户端网页采用的开发编程语言是 HTML/JS，最后，数据库采用的是目前比较主流的数据库软件 PostgreSQL 数据库。

4.2 需求分析

采用 B/S 或 C/S 模式实现一个电信收费管理系统软件。实现电信套餐种类、用户信息、客服代表、收款员等信息的管理。

要求：

- 1) 实现不同权限的浏览和更新。
- 2) 实现用户扣、缴费情况及帐户余额的查询。
- 3) 实现欠款用户使用状态的自动改变。
- 4) 实现客服代表的业绩统计功能。
- 5) 提供至少两种风格的查询报表。

传统的数据库系统多位 C/S 结构，如图 4.1。

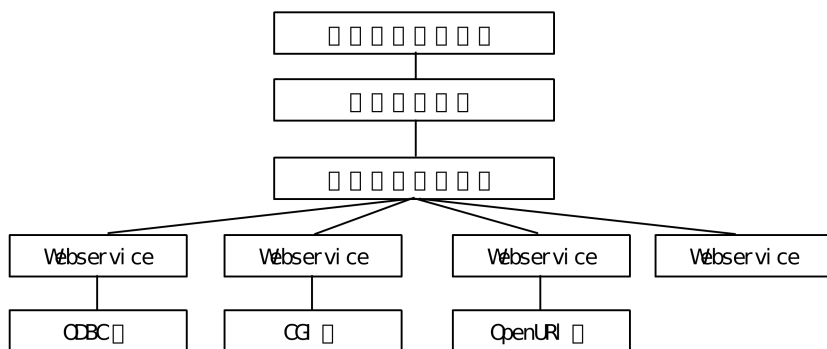


图 4.1 异构数据库统一检索系统

采用 B/S 或 C/S 模式实现一个电信收费管理系统。完成套餐、用户、客服代表、收款员等信息的管理。功能上需要提供客服代表、用户、收款员三类用户的录入、查询界面。对欠款用户状态的自动更新，以及不同的查询报表。

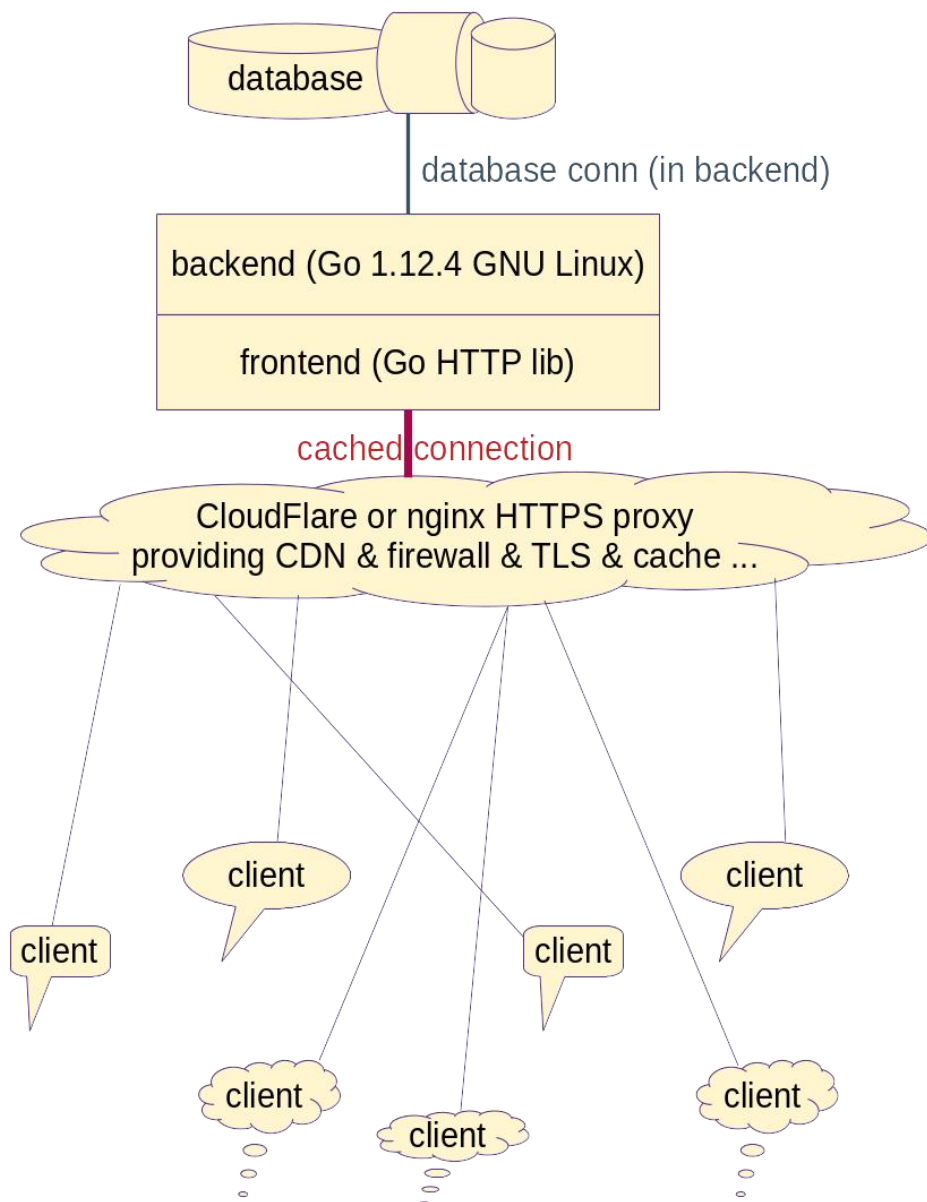
系统总体需求列表如表 4.1 所示。

表 4.1 系统总体需求

功能编号	功能名称	功能描述	权限
1	添加用户	为用户开户,可以设置用户权限	客服代表
2	缴费	给用户缴费	收款员
3	余额查询	查询用户的余额	无需权限
4	扣费	给用户扣费	收款员
5	用户信息查询	查询用户的套餐,余额信息	无需权限
6	用户状态更新	用户欠款时状态自动更新	无需权限
7	业绩统计	统计客服代表的业绩	无需权限

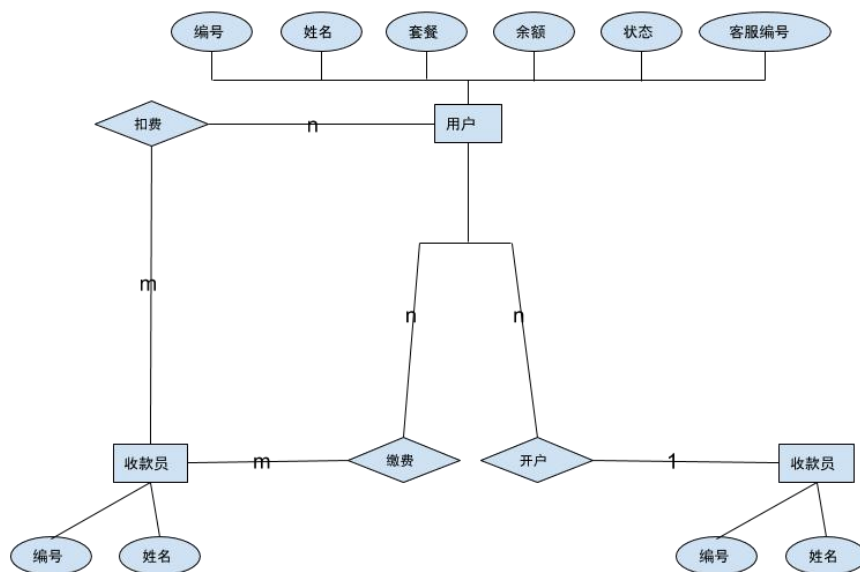
4.3 总体设计

系统的设计包括 3 个部分：数据库，前后端服务器，web 客户端。每一个用户都具有一些权限 tag，每一个账号可以具有客服代表、收款员、用户的身份标签。服务端在进行某一个操作时会检查用户是否有对应的权限进行相应的操作。



4.4 数据库设计

将功能需求转化为 E-R 图如下：



接下来，将 E-R 图转换为实际的数据库结构。图中的每一个实体对应一张表，1 对 n 关系在 n 端连接的表中引用 1 端所在表的主码作为外码，n 对 n 关系则将关系本身构建为一张独立的表，这张独立的表含有两个实体的主码。以此构建的数据库中含有的表如下表所示。

表名	说明
user_info	用户列表
serv_info	客服列表
cash_info	收款员列表
payments	缴费信息
billings	扣费信息

各表详细信息如下：

user_info：

字段	类型	说明	KEY
----	----	----	-----

	uid	integer	用户编号	PRI
	name	varchar	用户名	
	plan	varchar	套餐	
	balance	integer	余额	
	state	varchar	状态	
	sid	integer	客服编号	MUL
serv_info :				
	字段	类型	说明	KEY
	sid	integer	客服编号	PRI
	name	varchar	名字	
cash_info :				
	字段	类型	说明	KEY
	cid	integer	收款员编号	PRI
	name	varchar	名字	
payments :				
	字段	类型	说明	KEY
	uid	integer	用户编号	PRI
	caid	integer	收款员编号	PRI
	money	integer	缴费金额	
billings:				
	字段	类型	说明	KEY
	uid	integer	用户编号	PRI
	cid	integer	收款员编号	PRI
	money	integer	扣费金额	

4.5 详细设计与实现

阐述各主干功能的实现过程，包括主干功能的业务流程图、关键技术和算法

说明、数据库事务的定义与实现、数据库函数和触发器的定义与实现等（不允许大段引用源码，如有必要引用必须加详细注释）。

- 设计理念：

1. 一个系统是安全的，当且仅当在攻击者了解系统除密钥外所有细节的情形下，这个系统是安全的。

2. 不安全的系统是不可用的系统。

3. 不可维护的系统是不可用的系统。

4. 不应当为用户引入任何不必要的复杂度。Simple is beautiful.

考虑开发成本 尽管后端支持 CSRF Token ,但 web 前端没有实现 XSS 和 CSRF 攻击的防御。

- 特点

1. Design security from start: 从开始设计系统时就考虑到安全因素。例如，token 生成算法考虑了精通密码学的攻击者。

2. 后端不信任前端，不信任任何用户输入。例如找回密码的 email。

3. 在全栈开源、攻击者了解系统一切细节的前提下，系统可以利用密钥保持安全。

4. 真正严格的权限管理。

5. Docker 打包，在任何服务器系统上一键部署。可以使用 Kubernetes 等现成的工具进行弹性伸缩和负载均衡。

6. 数据库与后端分离，后端与前端分离，可以对任意一个部分单独进行集群部署和负载均衡，缓解了性能瓶颈。后端支持将 session 分散到不同的服务器，除 session 内状态外，服务器是无状态的。有极好的并行优化空间。

7. Go-pg 自动进行 SQL 语句生成/转义，直接防止了 SQL 注入的可能，而且能够减少 bug。后端代码量极少，却支持 CSRF Token 等大量高级功能。

8. 完善的 ACID 考虑，在必要的时候使用事务保证数据一致性。

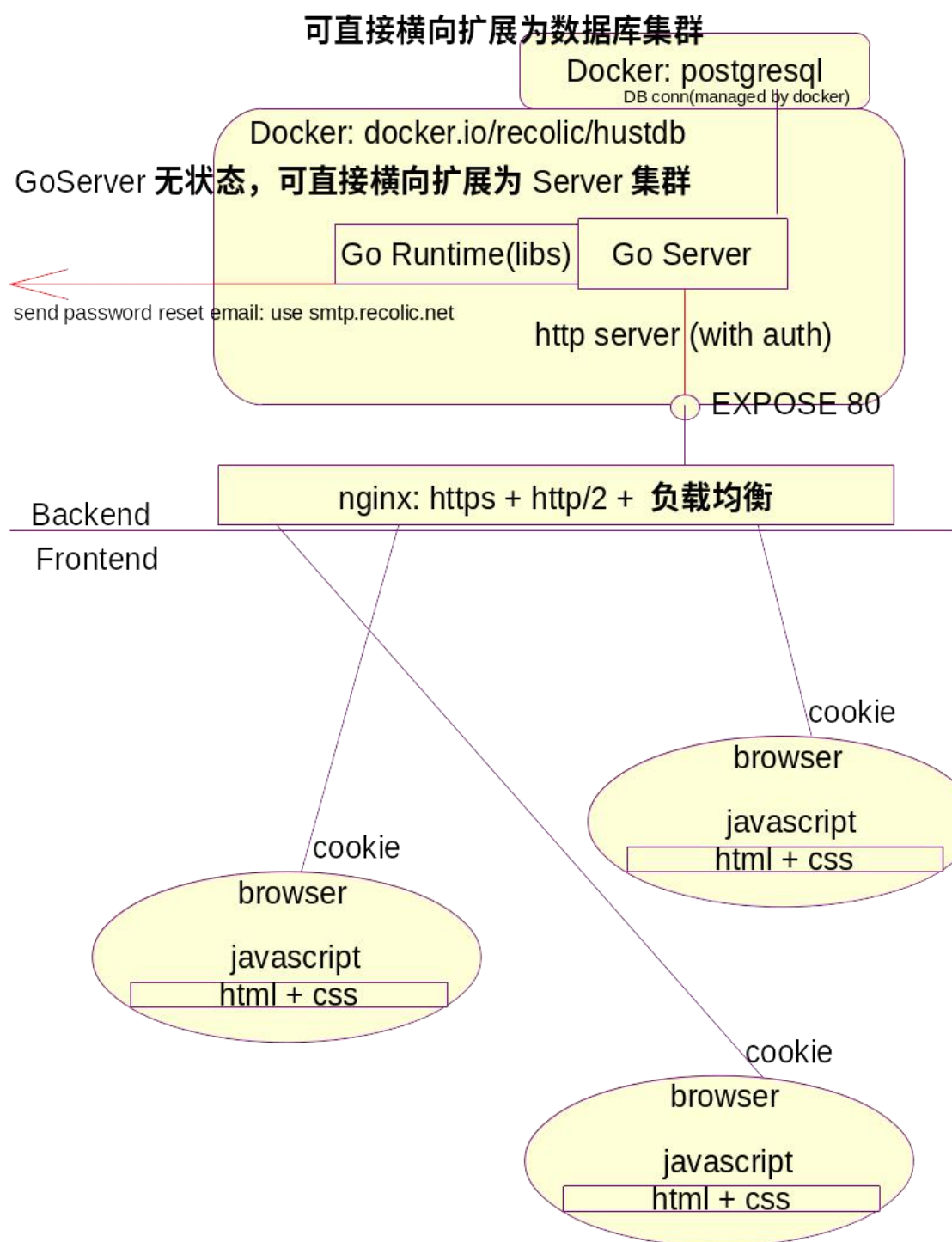
9. 完善的并行优化。Go 是高度优化且对并行支持良好的编译型后端语言。

10. Nginx 支持 https、http2、HSTS 等完善的现代安全特性。可以与

Cloudflare、Cloudfront 等工具配合使用。示例 :<https://hustdb.recolic.net/>

11. 前端 web 有自适应 , 电脑、手机、平板电脑、kindle 等设备都可以自适应的绘制页面。在各种操作系统、各种设备的前端浏览器下均能绘制足够好看的界面。根据用户权限动态绘制页面。自然地 , 它支持跨设备记住密码等浏览器特性。

- 最终设计图 : (Docker/Kubernetes 一键部署)



- 分块介绍实现

- 数据结构

数据库内表格结构由 go-pg 由 go 结构体结构直接推导。详细结构如下：

首先是用户信息表。其中的 email 仅用于找回密码。

```
type UserInfo struct {
    Id uid_t `sql:",pk,unique"`
    Name string `sql:",unique"`
    Password string
    Permissions []string `sql:",array"`
    Balance money_t // `10.32` is saved as `1032`
    Achievements money_t
    Plan planid_t `sql:",notnull"`
    Email string `sql:",unique"`
}
```

然后是用户余额变动的日志表。

```
type UserBalanceEvent struct {
    EventId uid_t `sql:",pk,unique"`
    Uid uid_t
    What string // should not contain `;`
}
```

然后是记录套餐情况的表。

```
type PlanInfo struct {
    Id planid_t `sql:",pk,unique"`
    Name string `sql:",unique"`
    Price money_t
}
```

所有数据库操作由 go-pg 完成，因此表格中的所有字符串会被自动 prepare，防止 SQL 注入攻击。

部分字符串，例如姓名，在找回密码的 email 的链接中直接用到。由于 email 链接注入危害性如此之大，我不得不检查用户输入的 email 和 name 的格式。这是为了防止 RSS 和 email MIME 文本注入，而不是 SQL 注入。

- common.go

此文件中除了数据结构和常用常量之外，还定义了几个常用的函数。例如 UsernameToInfo 和 PlannameToInfo。函数命名已经足够清晰，不再赘述。

- mailing.go

此文件定义了发送电子邮件的相关方法。我在自己的域名下创建了一个公开

密码的邮箱，即使密码被 hardcode 到源代码中，我的 anti-spam 系统也会阻止攻击者使用此邮件地址作为武器。

特别注意的是，由于 sendmail 过程可能很慢，调用 sendmail 直接创建了一个新的 goroutine 进行异步邮件发送，用轻量级的方式避免了对这个 http 请求的响应延迟。

- auth.go, http.go

auth.go 定义了 token 生成、身份验证、修改密码、找回密码等多个认证相关操作。注意客户端是不被信任的，因此来自 http.go 的任何请求都需要经过 token 验证。

在用户登录时，其调用 DoLogin API，然后如果服务器检查密码正确，那么就会生成一个对应于该用户的 token。此 token 只能在这一台服务器上使用，如果有多台服务器，负载均衡应注意让同一个 session 保持在同一台服务器上。

在调用其他任何 API 时，服务器都会先检查 token。如果 token 中含有的权限不足以执行这个 API，那么 http 请求返回 403。如果 token 有效，后续操作会一直携带这个 token 对应的 uid 作为 API 的 commiter，并在必要时进行更多的身份认证。

token 的生成使用密码学安全的伪随机数生成器，其使用的种子是服务器启动时的系统时钟纳秒数。生成的随机数和当前纳秒数再次 xor，得到最终的 token。假定攻击者不知道服务器启动的大致时间，便难以破解 token 生成器。但是，如果攻击者知道服务器启动的毫秒数，那么只需要几百万次离线尝试就能破解 token 生成器，我将结果与当前纳秒数 xor 不会显著增加攻击者的计算量（因为纳秒数只有最后几位会变！）。所以正确的做法应该是使用硬件随机数生成器结果作为种子。

http.go 接受 http 请求，创建新的 goroutine，并进行身份认证之后调用 operations.go 里面的对应函数执行这些请求。

- operations.go

这里对每一个 API 进行实现。对于需要在同一个 API 中进行多个数据库操作的情形，他们会被装到同一个 Transaction 中，保证操作的原子性。

由于不同操作代码相似且极其简单，我只需要大致介绍 go-pg 对数据库的操作流程。下面是将多个操作在一个 transaction 中进行的示例。

```
err := db.RunInTransaction(func(tx *pg.Tx) error {  
    // operations...  
    return tx.Insert(&u)  
})
```

db 提供了 Insert, Update, Select, Where 等完善的接口，极大的方便了数据库开发。其大致用法请阅读 <https://godoc.org/github.com/go-pg/pg>

- main.go

这里定义了入口函数。它会检查表中是否含有 root 用户。如果没有，则使用命令行中指定的 root 密码创建一个 root 用户。随后运行 http.go 中的 http 服务器。

4.6 系统构建、部署与测试

- Build

- 自动编译

无需做任何事。

AWS CodeBuild 为此工程提供自动编译。其自动安装依赖、编译此工程并生成能够直接部署的 Docker 镜像，并自动推送至 DockerHUB。

- 手动编译

运行 `docker build -t recolic/hustdb -f Dockerfile .`

- Deploy

- Docker 自动部署

运行 `docker run -d --restart=always --name hustdb -p 8088:8088 -v /srv/hustdb:/var/lib/postgres/data recolic/hustdb`

如果需要多服务器集群，你只需要在每一个节点上运行此 docker 镜像。镜像占用的端口就是后端 http 端口，使用 nginx、DNS 等你喜欢的工具做负载均衡即可。注意，相同的 session 应当分配到相同的服务器上，不同的

服务器请共享同一个 PostgreSQL 集群。你可以在上面的命令后面指定 PostgreSQL 的服务器地址，也可以直接使用 docker 网络支持来内网连接集群。

- Kubernetes

- TODO。请参阅 Docker 自动部署方法管理集群。

- 手动部署

1. 连接国际互联网。
2. 使用你的包管理工具安装 PostgreSQL, git, Go, Nginx。
3. 用你喜欢的方式配置 PostgreSQL 或集群。可以参考 <https://wiki.archlinux.org/index.php/PostgreSQL>
4. 运行 `go get -d ./...` 以便安装 go 需要的依赖。
5. 运行 `go build` 编译此工程。如果希望安装到 PATH，请直接运行 `go install`。
6. 配置 Nginx。参考配置文件位于 `frontend/nginx.conf`
7. 运行服务器，将 PostgreSQL 的服务器地址、端口、账号、密码、初始 root 密码，监听地址等信息作为参数传入。

- Test

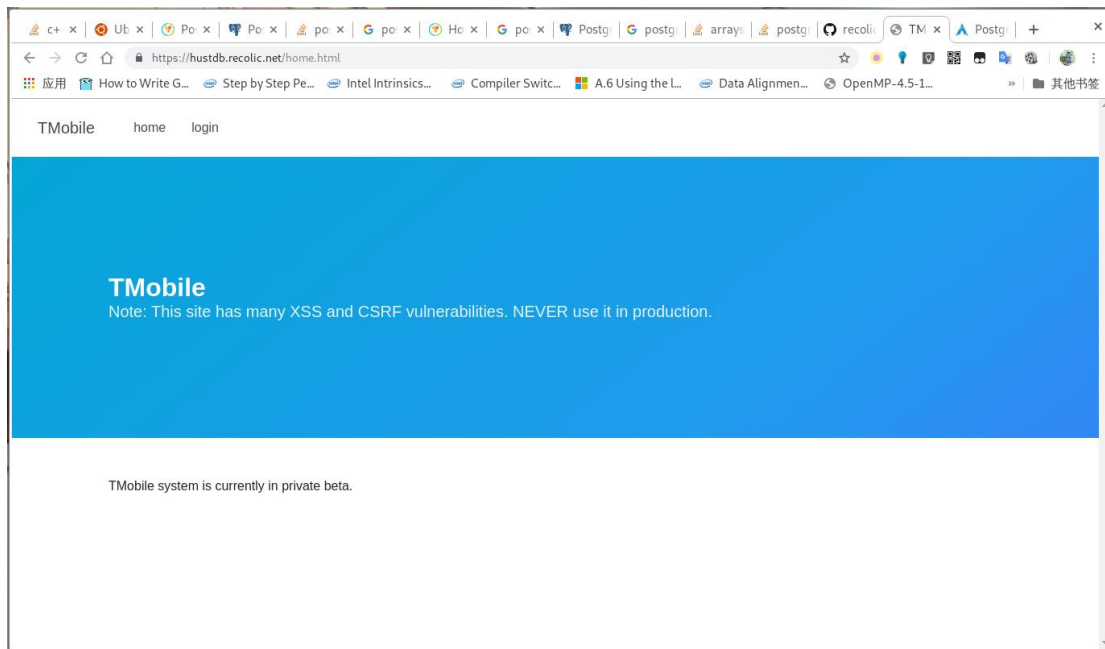
- 自动测试

- TODO

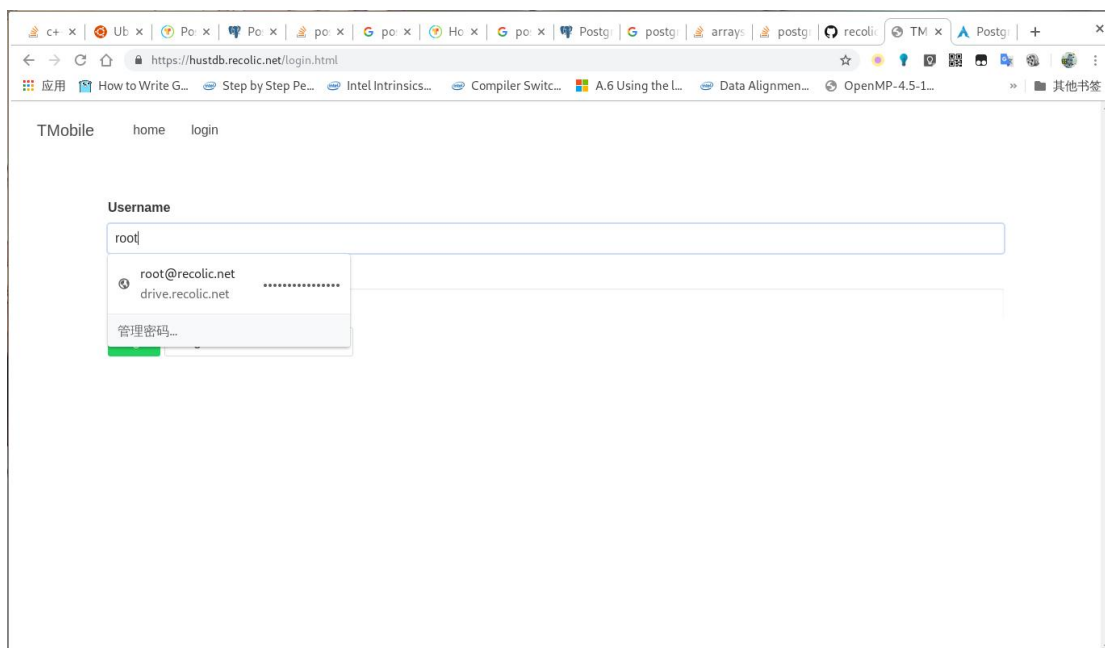
- 手动测试

- 此处假设服务器已经被部署在 <https://hustdb.recolic.net/>。此网页对 PC 和移动端都有一定的适配。手动测试只能对前端进行有限的测试。

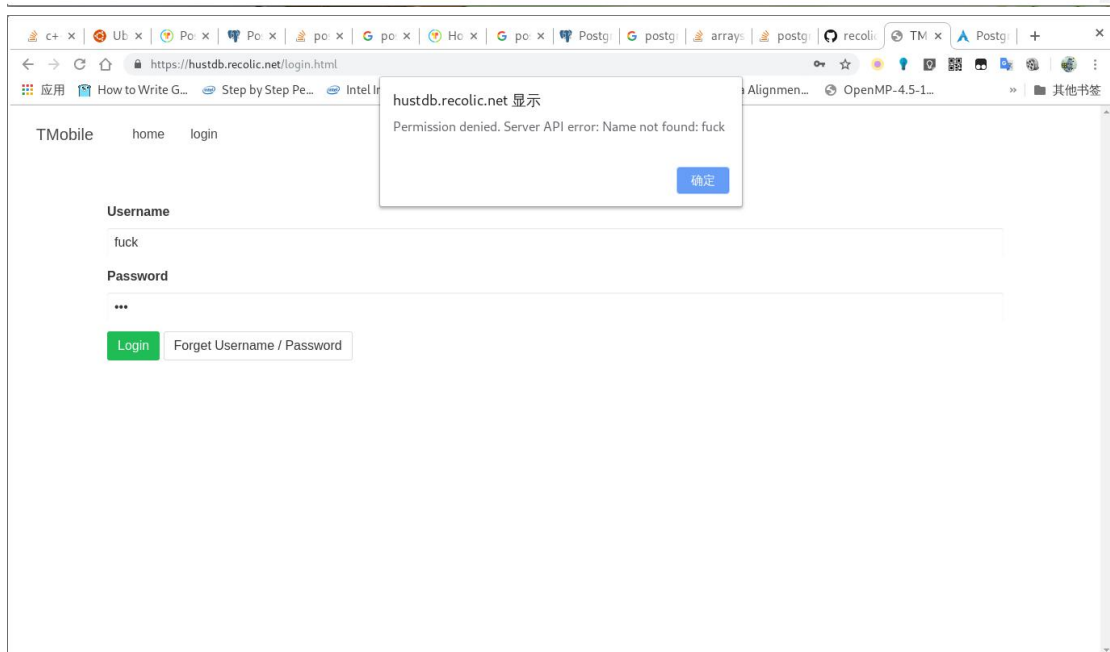
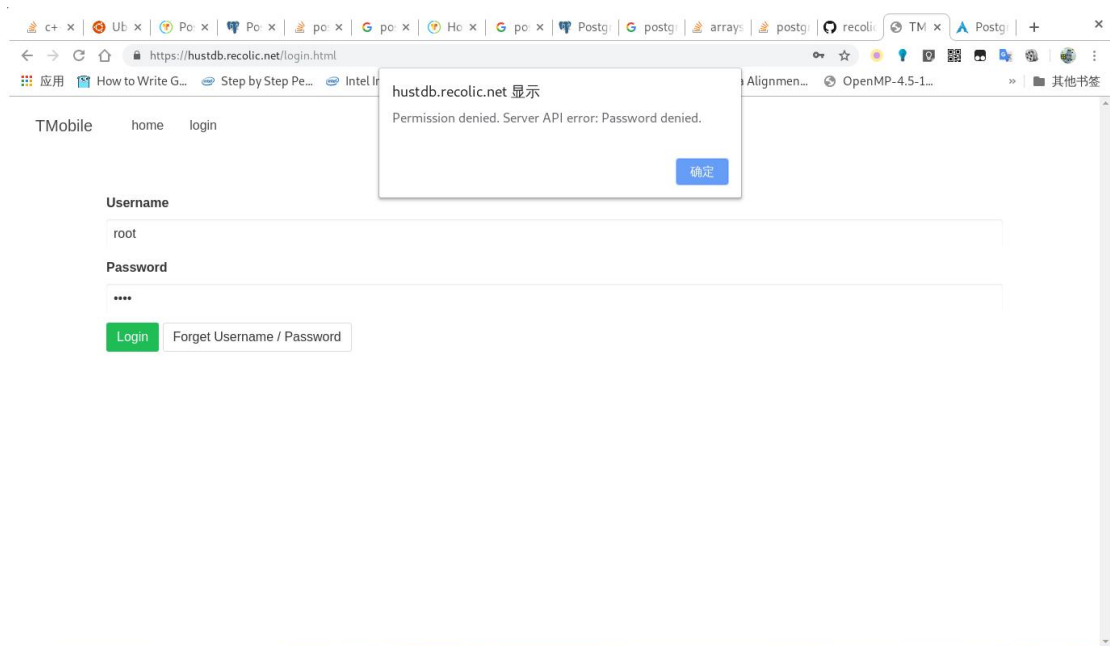
- 首先登陆 <https://hustdb.recolic.net/home.html>，查看首页显示正常。这里用来显示系统公告等信息。

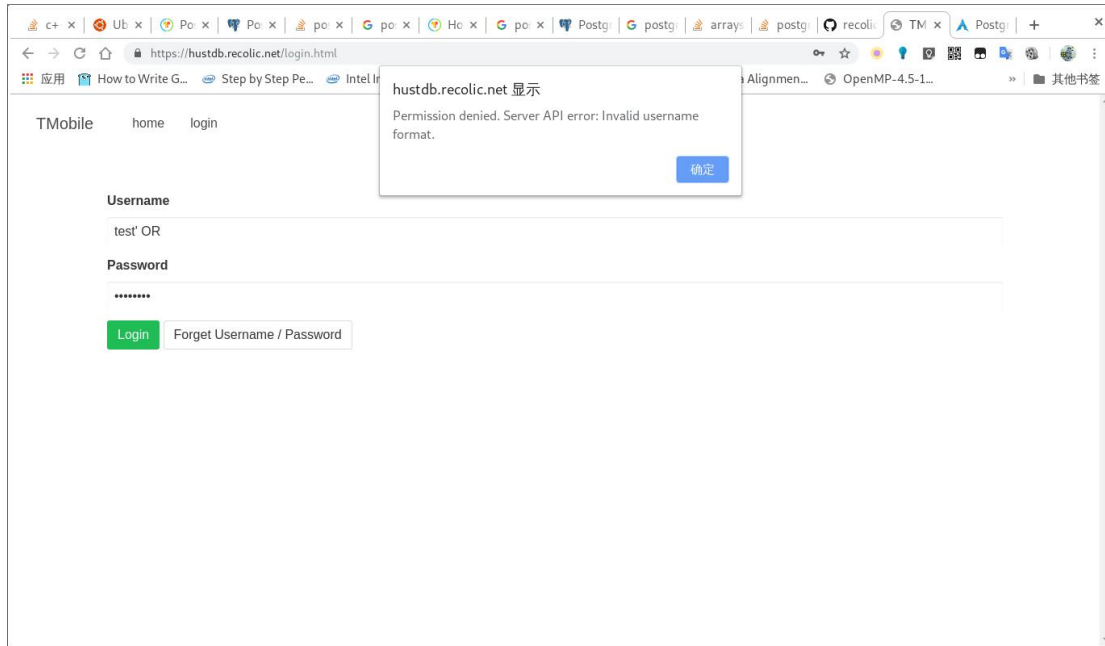


然后点击 login，登录页面显示正常。

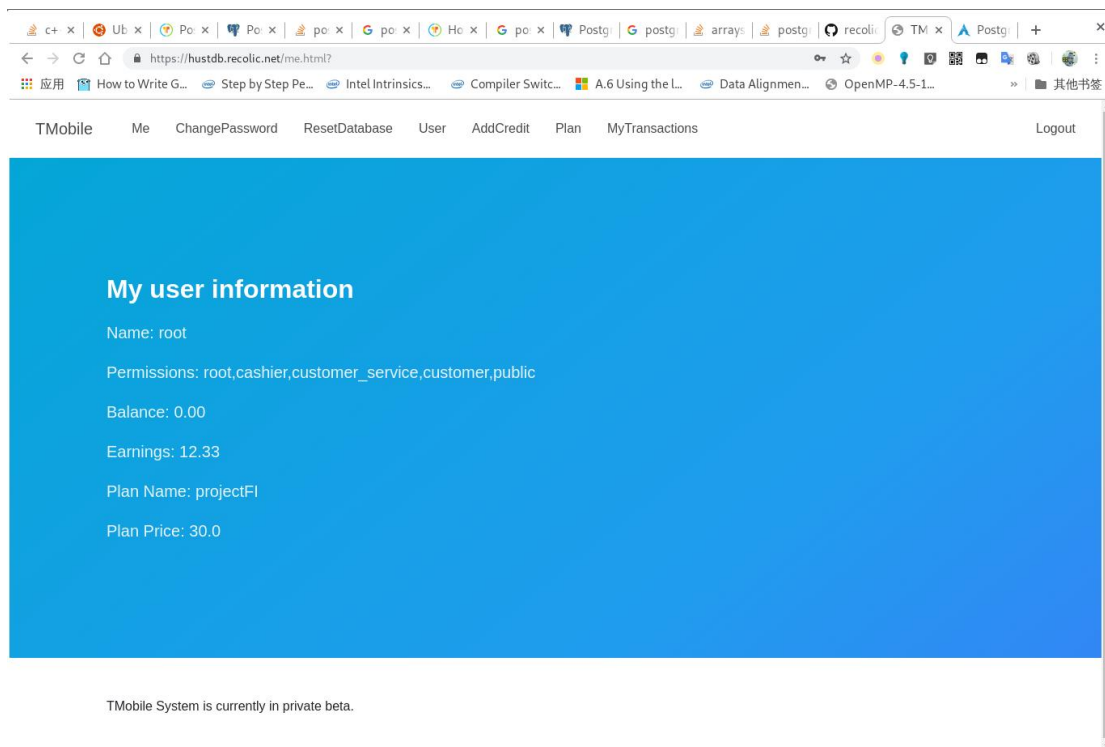


输入不正确的密码或不存在的用户名或在用户名/密码中尝试 SQL 注入攻击，均能被正确处理。在此过程中，Chromium，Firefox 和 Google Pixel 的记住密码功能均正常。

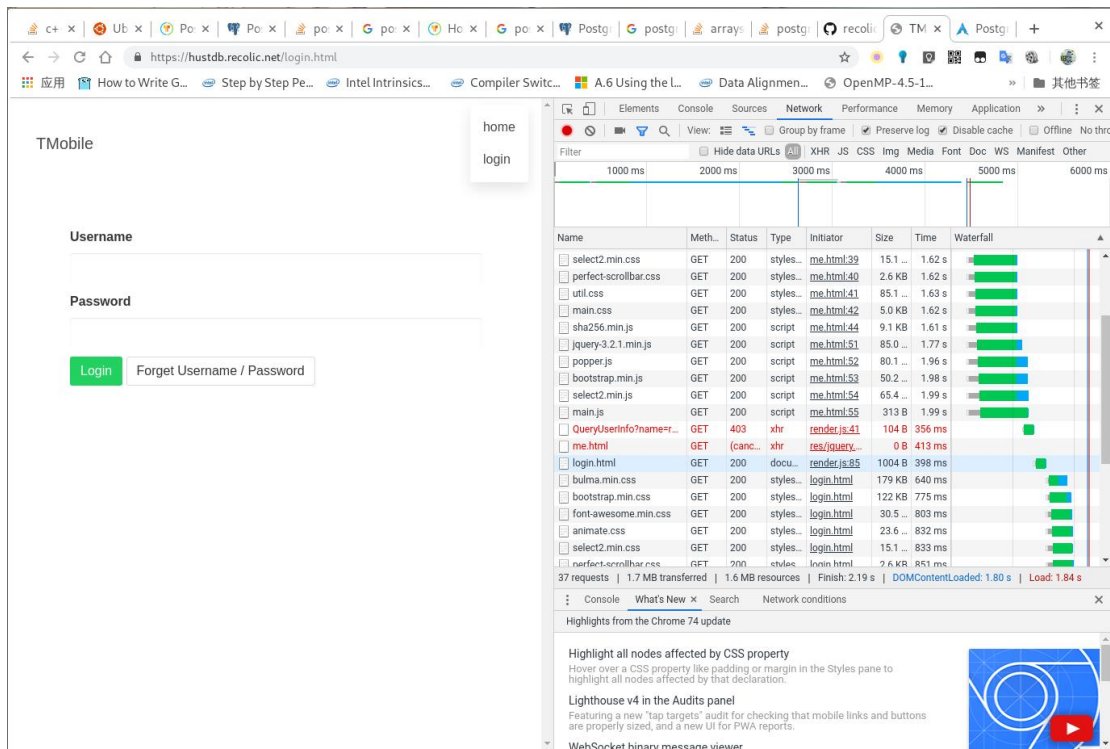




输入正确的密码，登陆成功，首个页面会显示出自己的用户信息。

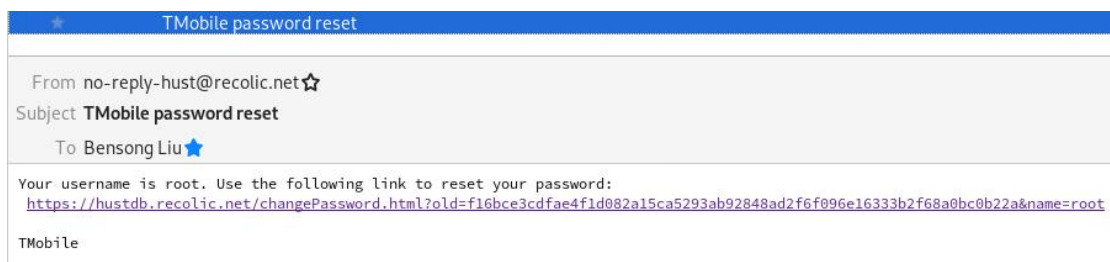


现在登出，然后手动输入显示用户信息的页面 URL，发现访问请求被拒绝，自动踢回登入页面。检查这证明登出成功。（注意右侧红色请求 :403 Permission Denied）

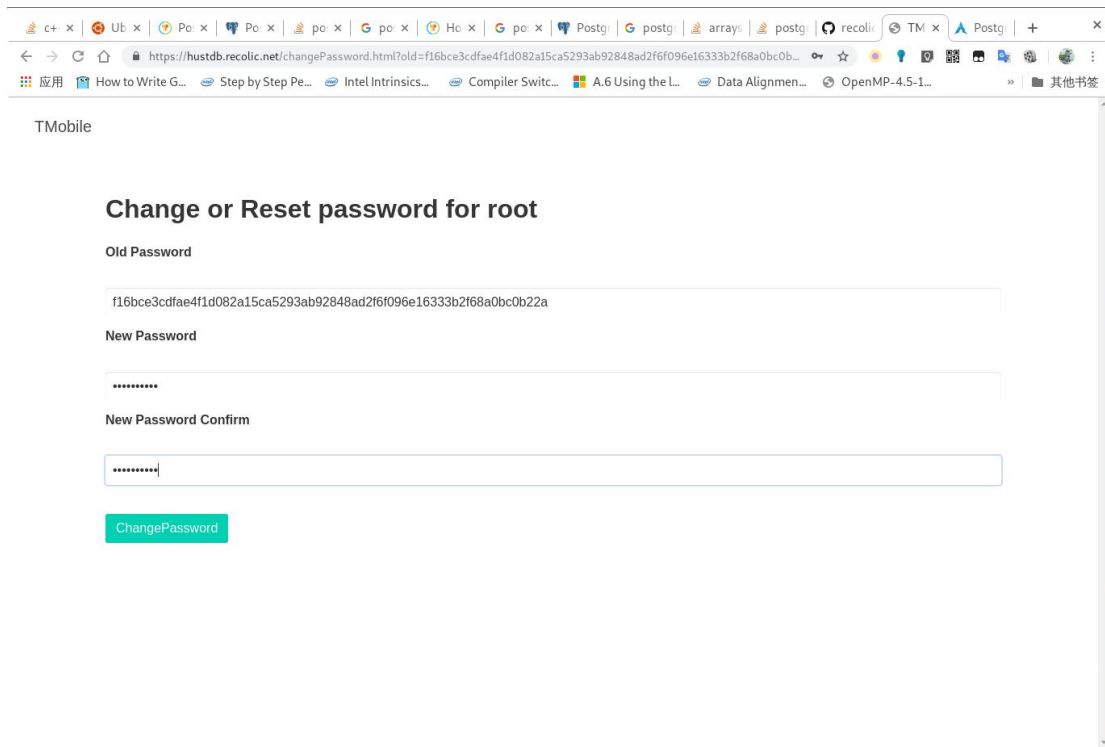


现在尝试忘记密码功能。我在注册 root 用户时提供了邮箱 root@recolic.net，注册 Bob 用户时提供了邮箱 recolickeghart@gmail.com。我尝试依次找回他们的密码，以便证明邮件发送服务对多个服务商都有效。

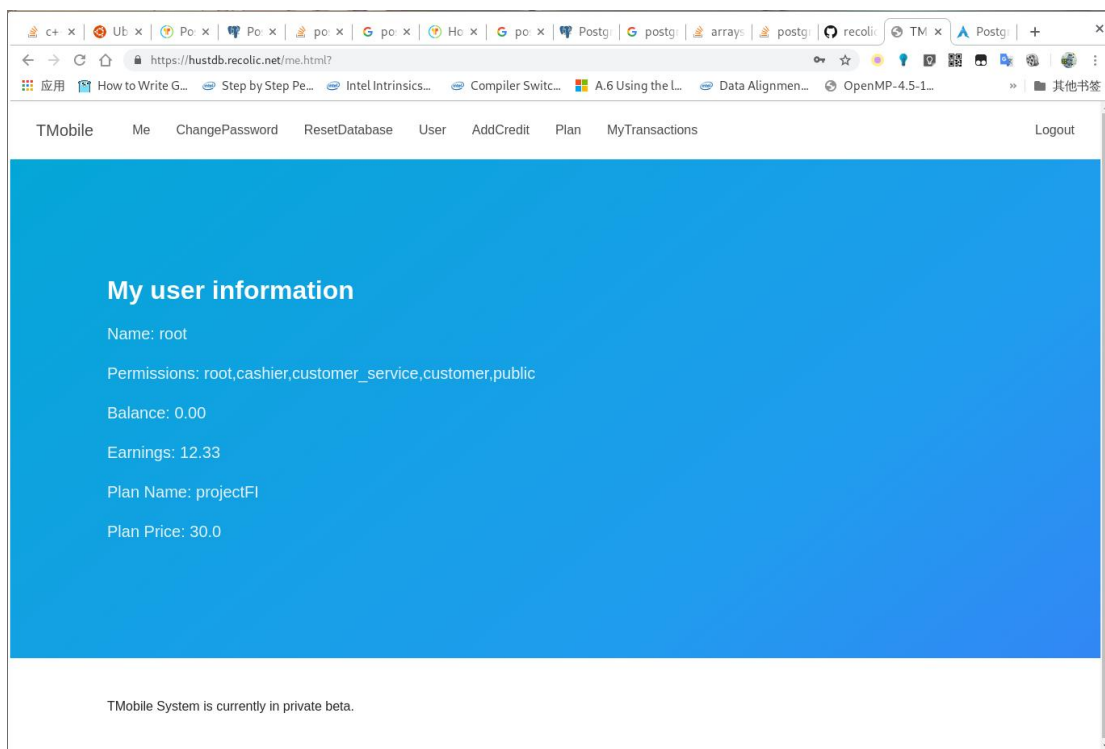
在选择忘记用户名/密码之后，输入 root@recolic.net，然后 2 秒内收到了包含用户名和密码重置链接的邮件。注意，此邮件中的旧密码并不会暴露用户的旧密码，这只是一个临时 token，没有安全隐患。



现在我进入此页面，重置密码，提示成功。

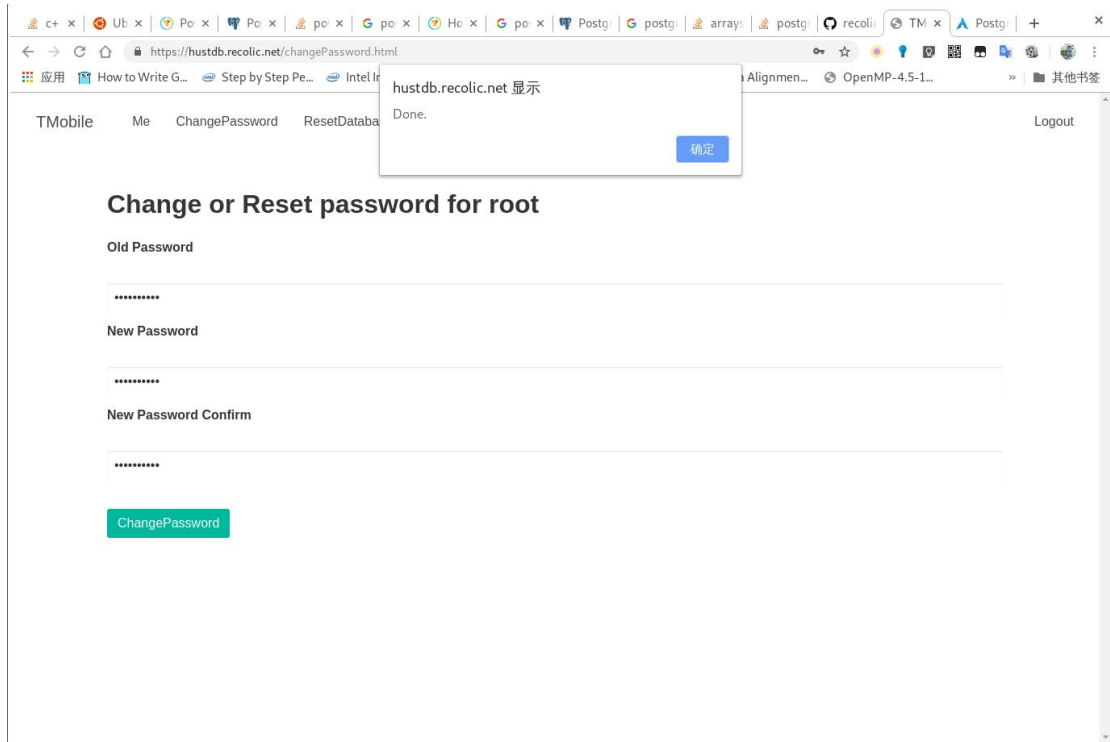


现在我使用新密码登陆，新密码已经生效。

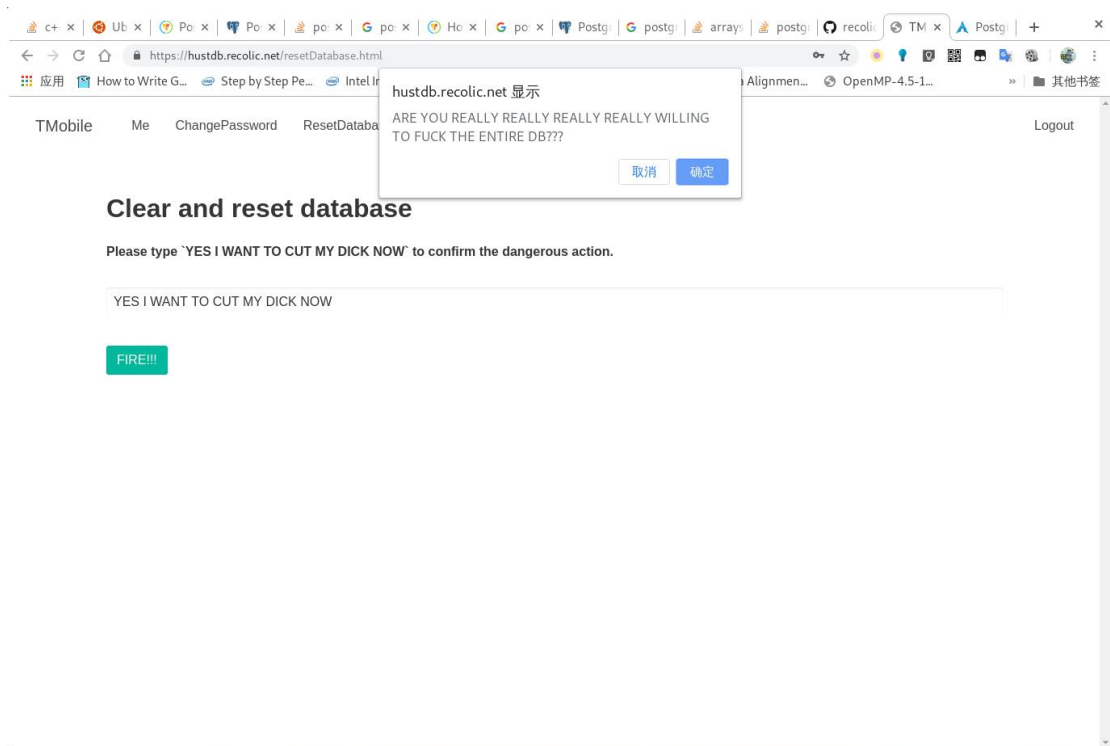


重新使用 root 用户登录，因为 root 用户具有所有的权限，我们就用 root 用户进行功能测试。

修改密码：需要权限：NONE

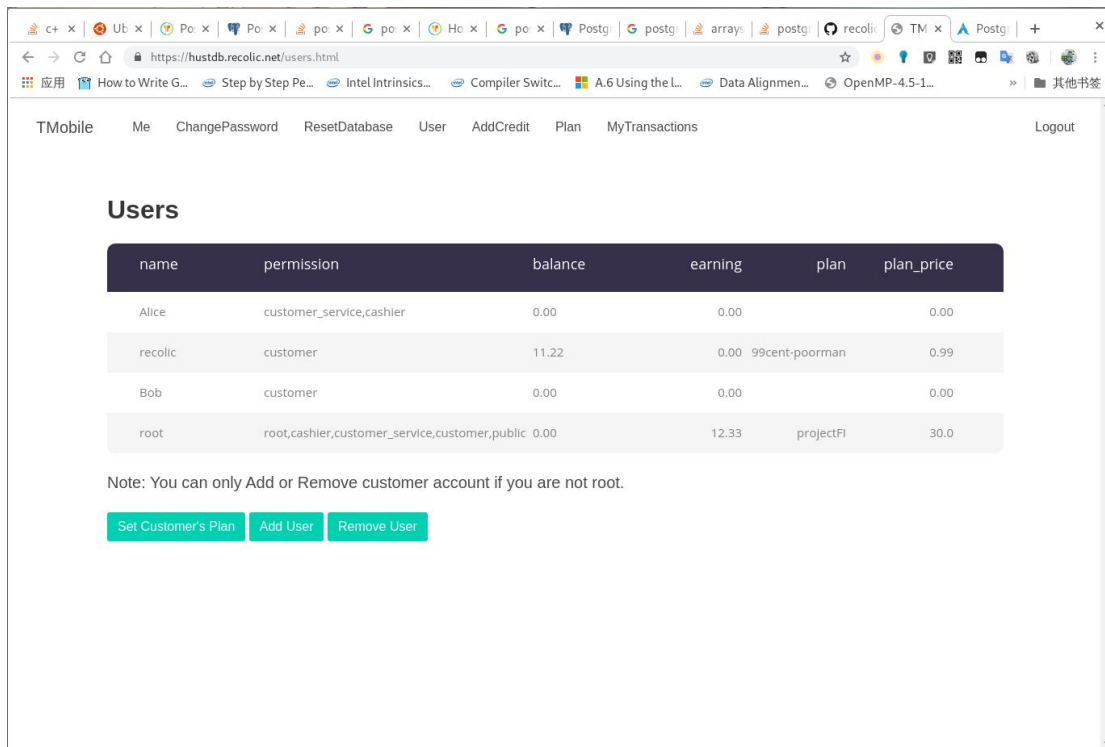


重置(清空)数据库：需要权限：ROOT



用户操作：需要权限：CUSTOMER_SERVICE

打开页面时自动列出当前所有用户信息。

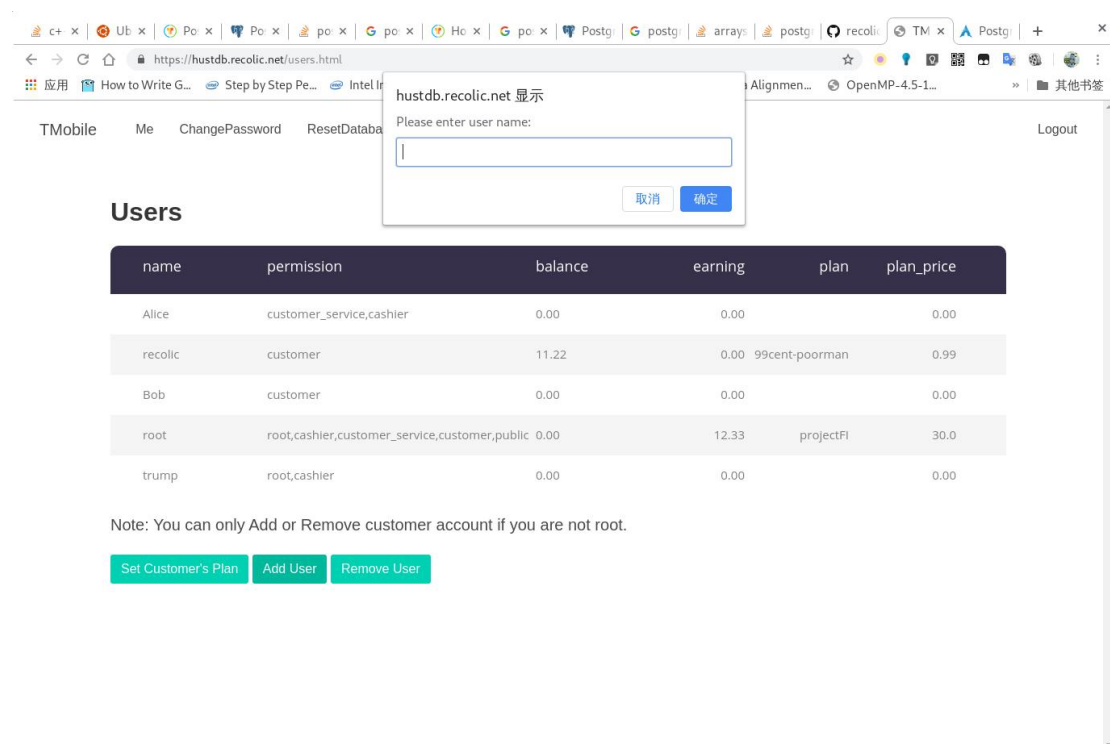


增加用户：需要权限：CUSTOMER_SERVICE 或 ROOT

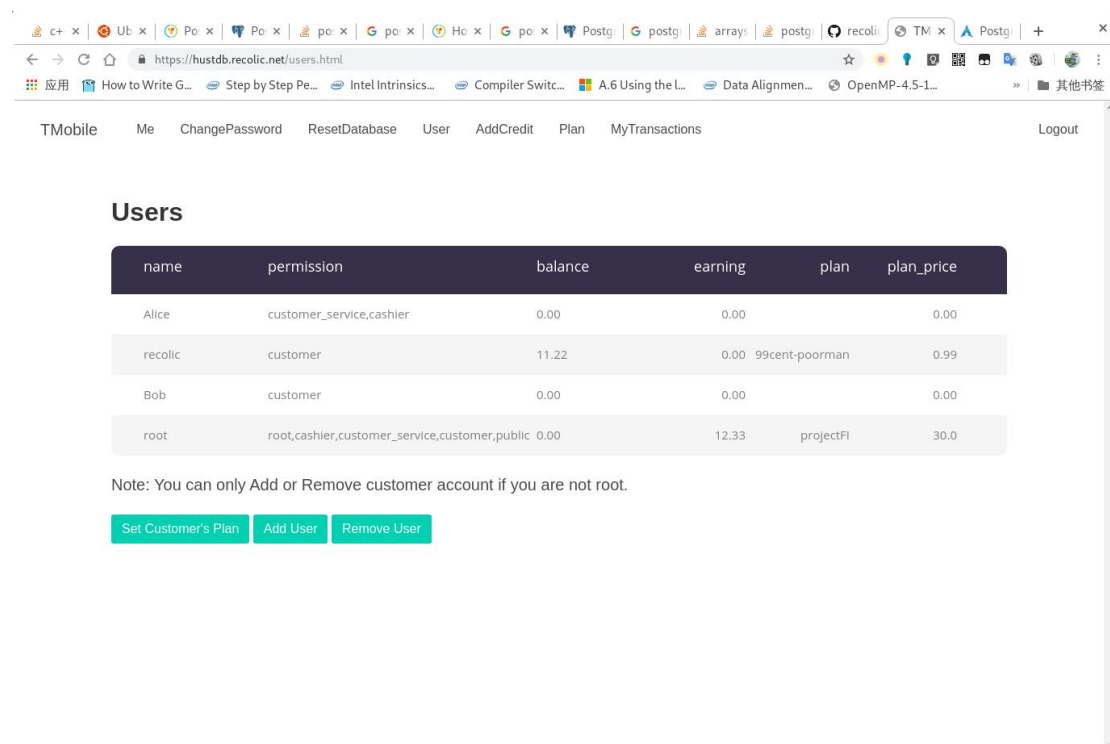
为新用户赋予除 CUSTOMER 以外的权限：需要权限：ROOT

如图：[我添加了一个新用户，名字 trump，权限 root+cashier，安全邮箱 fuck@outlook.com。确定之后页面自动刷新，新用户出现在列表中。](#)

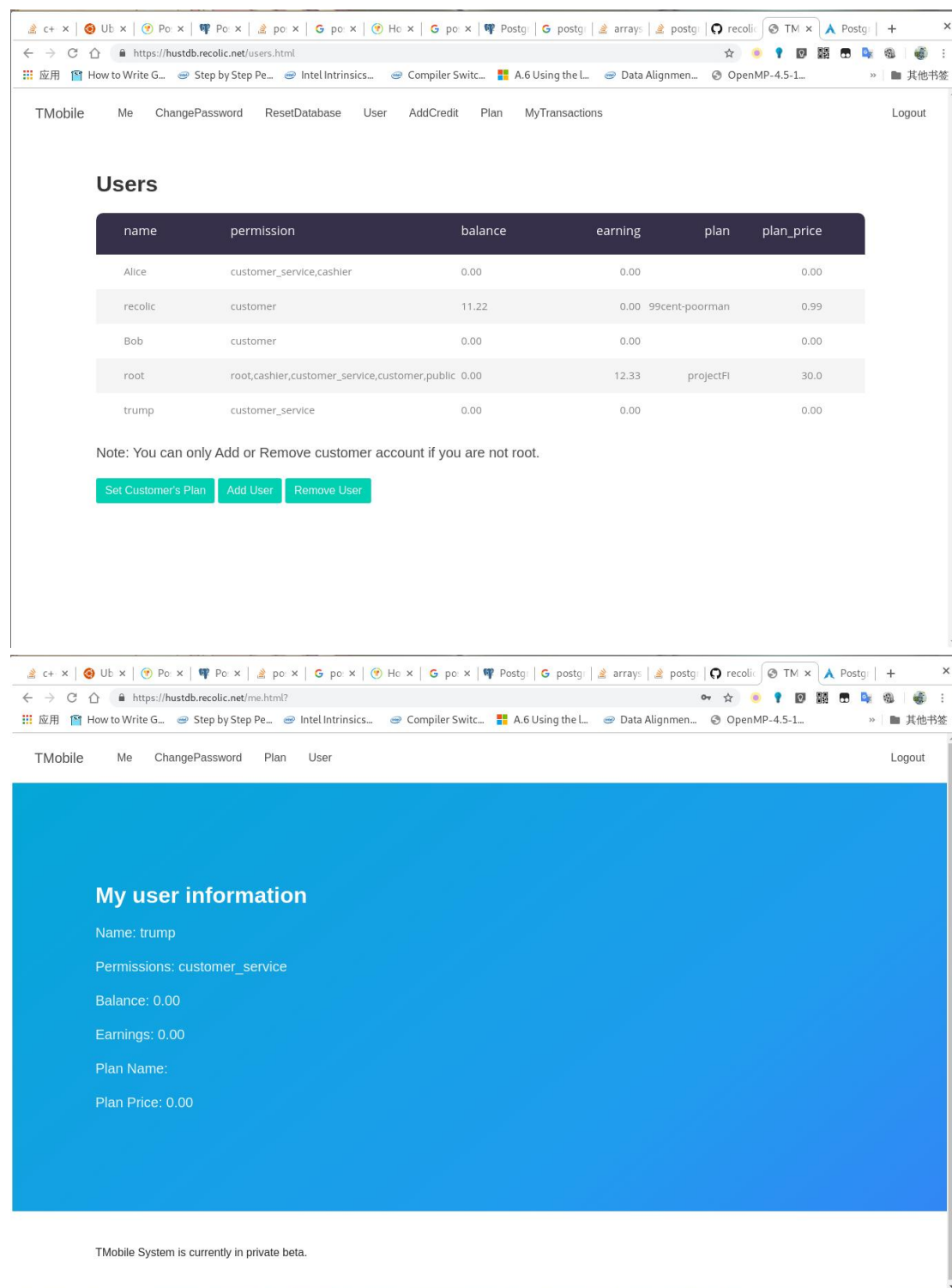
在以下步骤中，中途取消、错误用户名、用户名注入、空密码、重复用户名、非法邮箱等情形均已被测试，结果均正常，略去不截图。



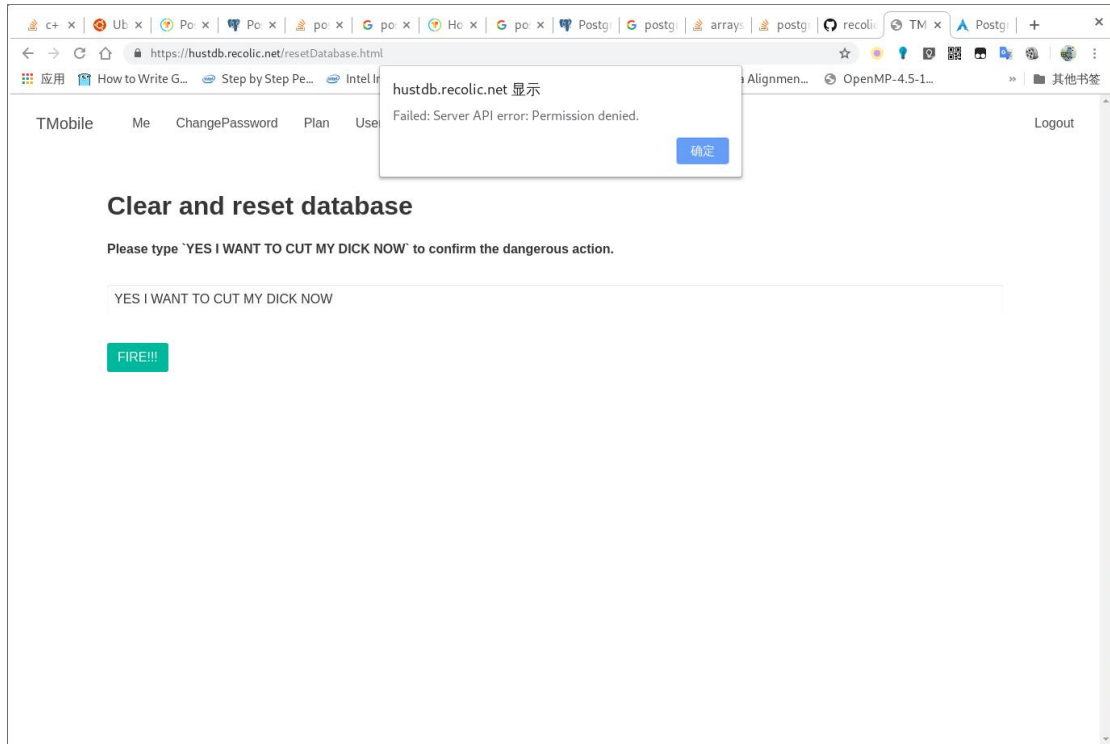
下面我删除这个用户。输入要删除的用户名后，点击确认，提示成功，页面自动刷新，用户被移除。



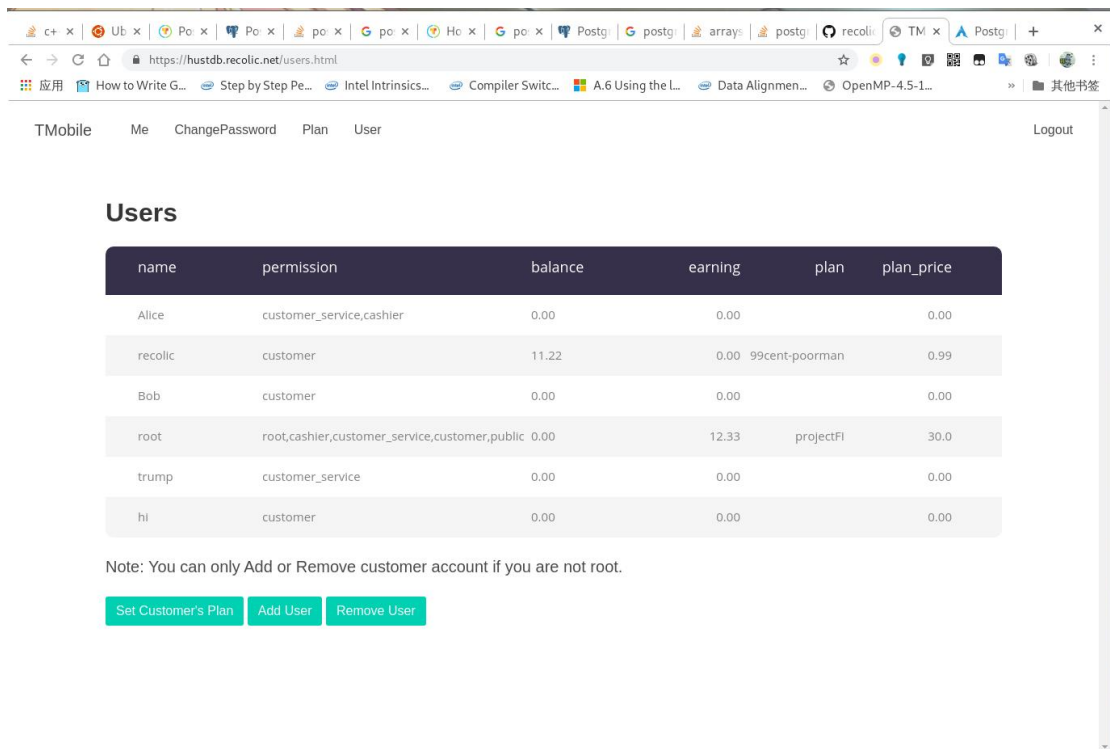
下面我添加一个用户名为 trump 的客服，并以他的身份登陆。



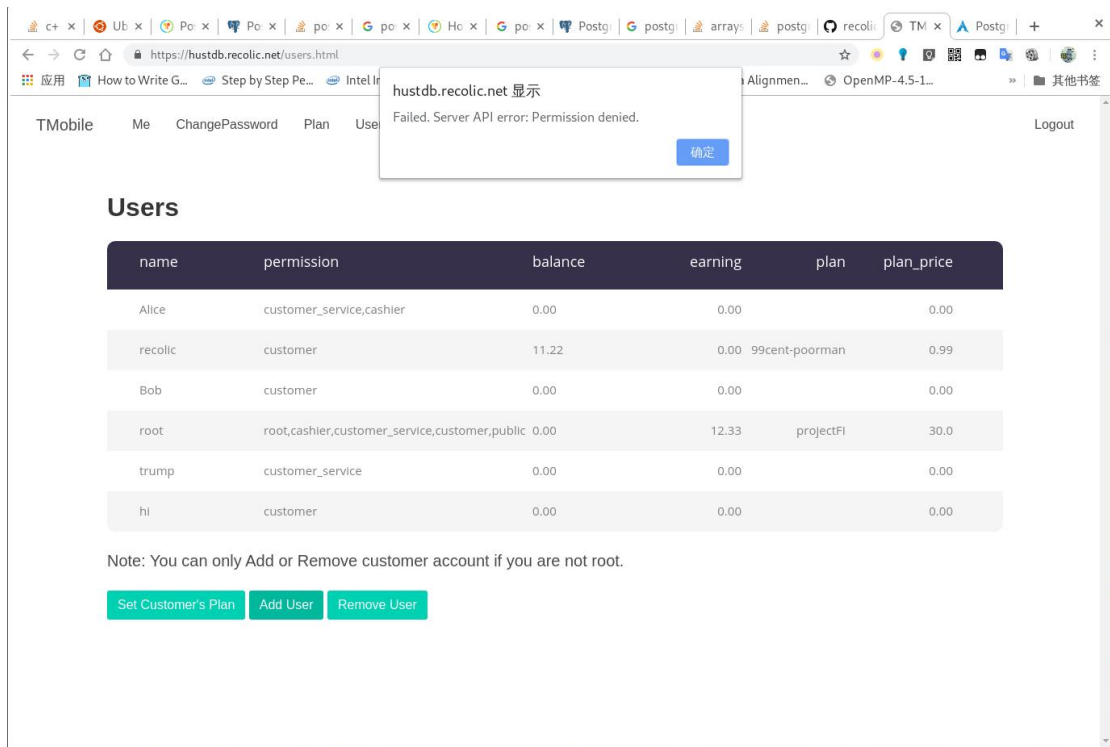
注意到，上面只显示了客服所拥有的功能。现在我尝试直接输入网址进入我没有权限的页面（清空数据库！），访问被拒绝。



此时我的身份为 trump，是一个普通客服，因此我只能增加新的顾客用户。下面进行演示：

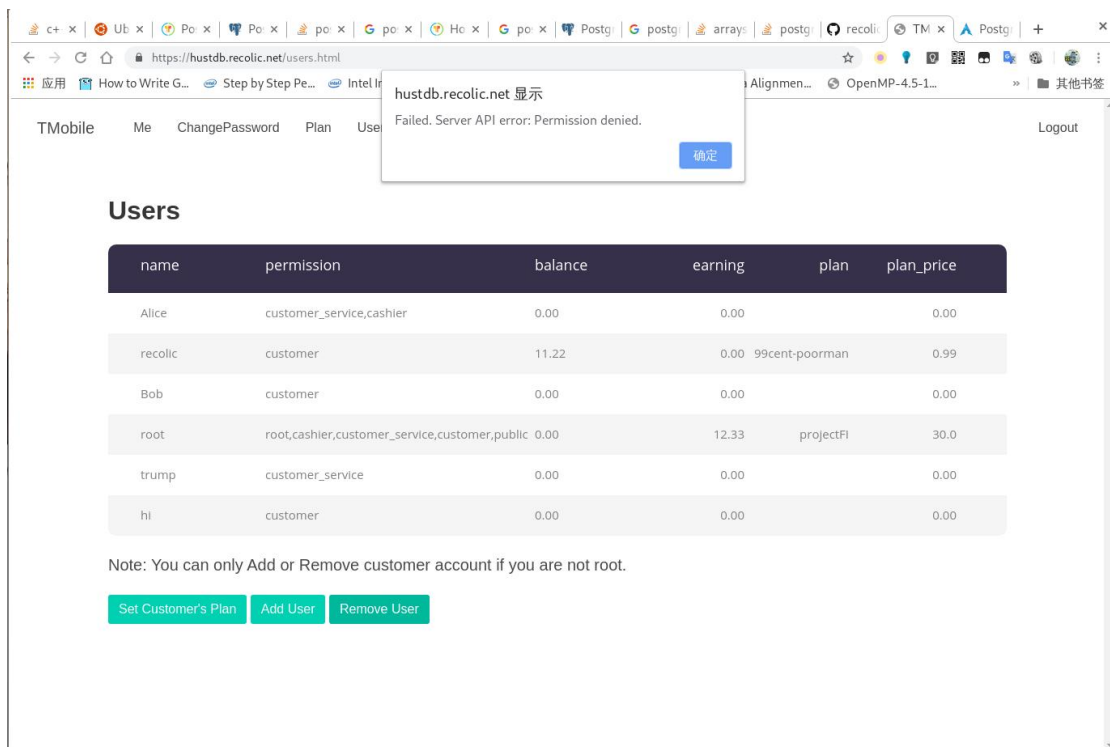


新增这个 hi 用户一切顺利。下面我尝试用这个客服建立一个新的客服（我没有权力这么做！）

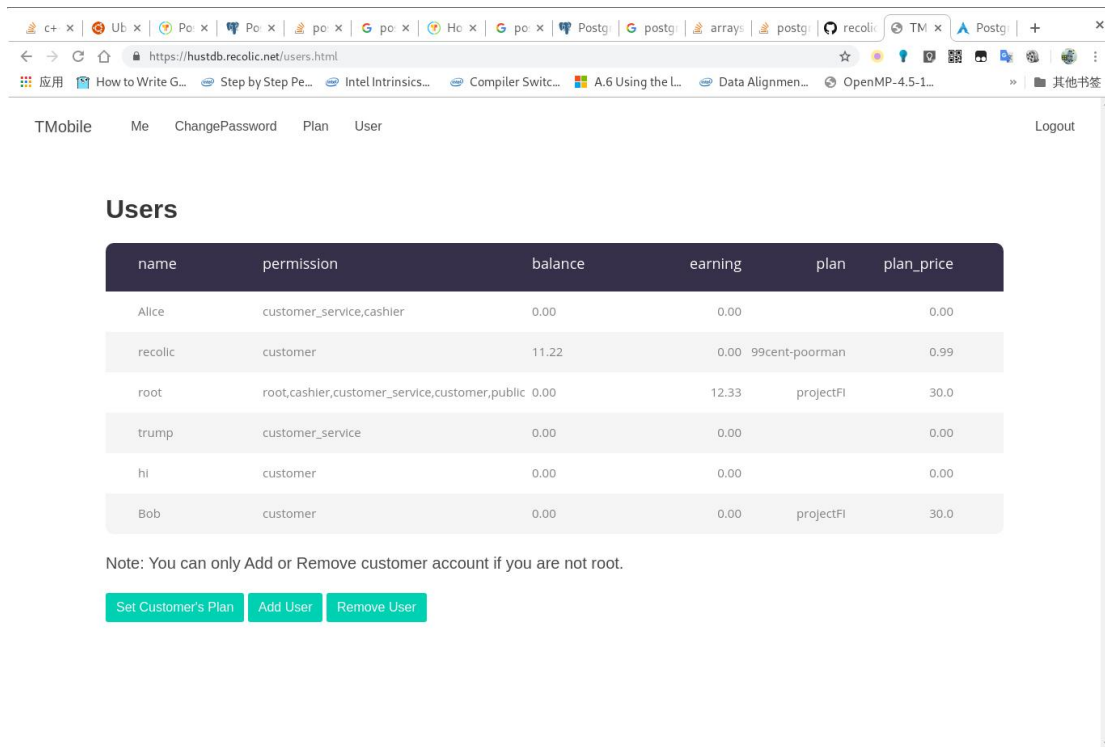


此时，我的操作被拒绝。因为只有 root 才能新建除了顾客之外的特权用户。

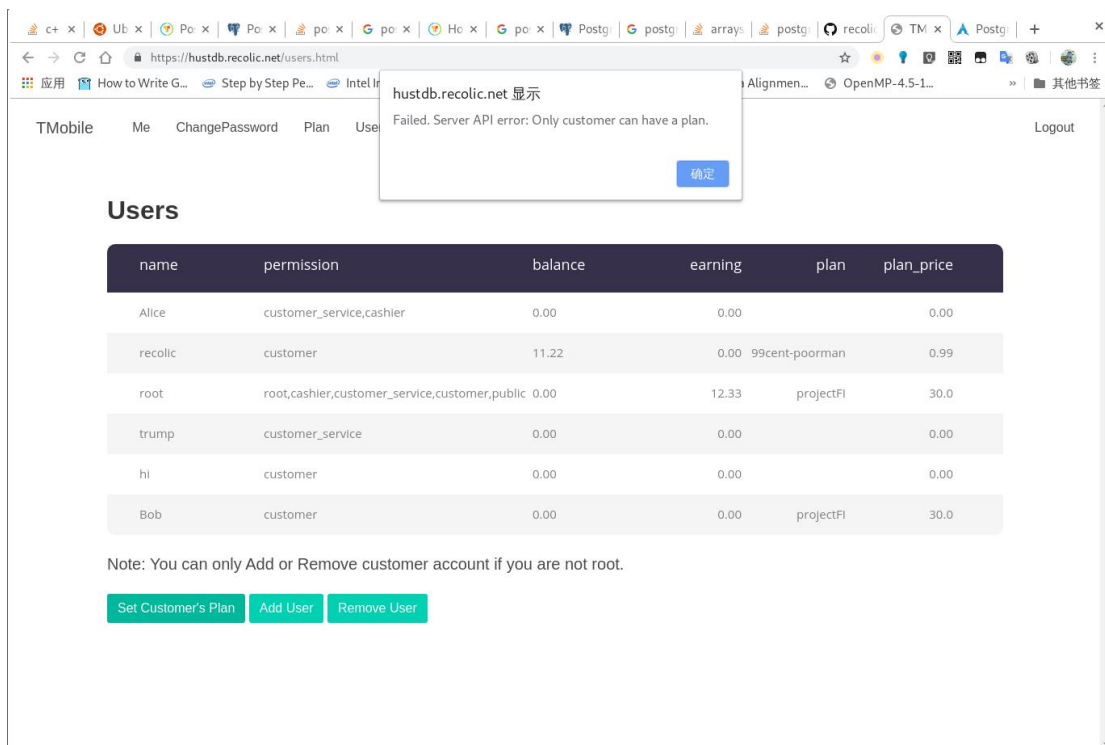
同样，我也不能移除除了普通顾客之外的用户。

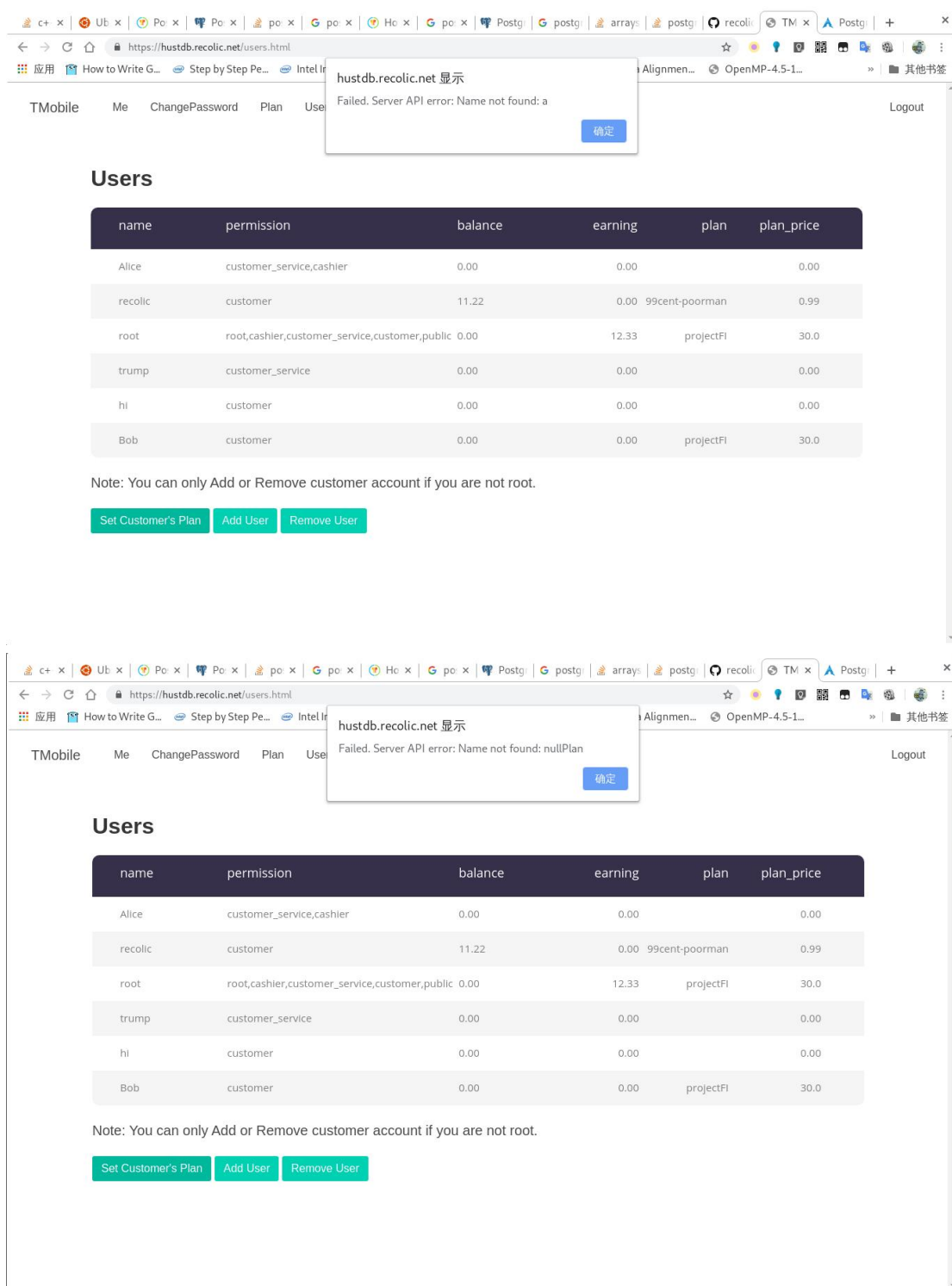


我可以设置用户的套餐，只需要输入用户名和套餐名。这个操作需要 customer_service 的权限。下图中，我给 Bob 设置了套餐 projectFI。

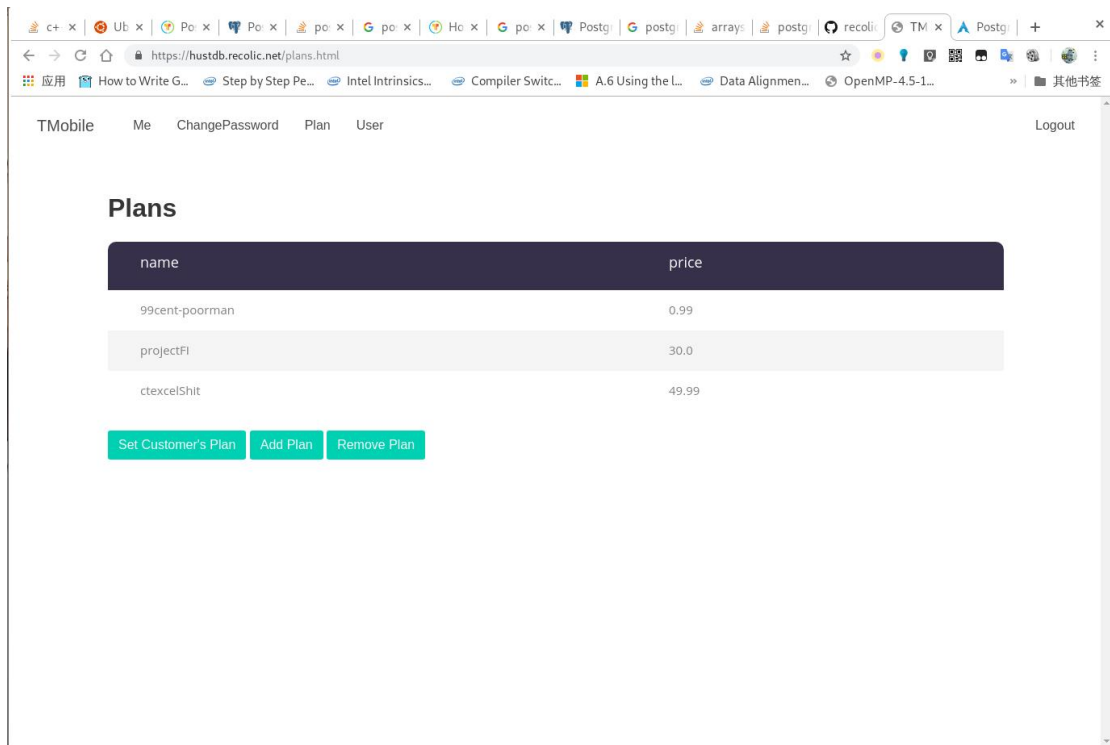


同时，我尝试了给非顾客设置套餐，或者设置一个不存在的用户名或套餐名，这都被恰当的错误处理了。

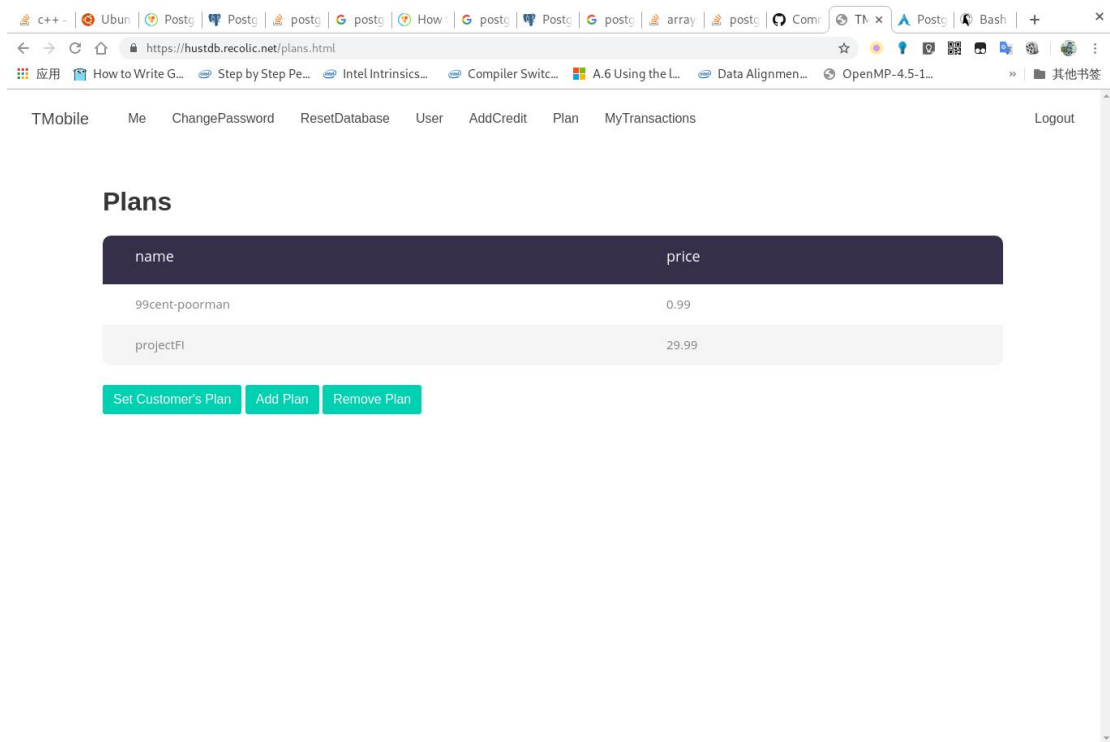




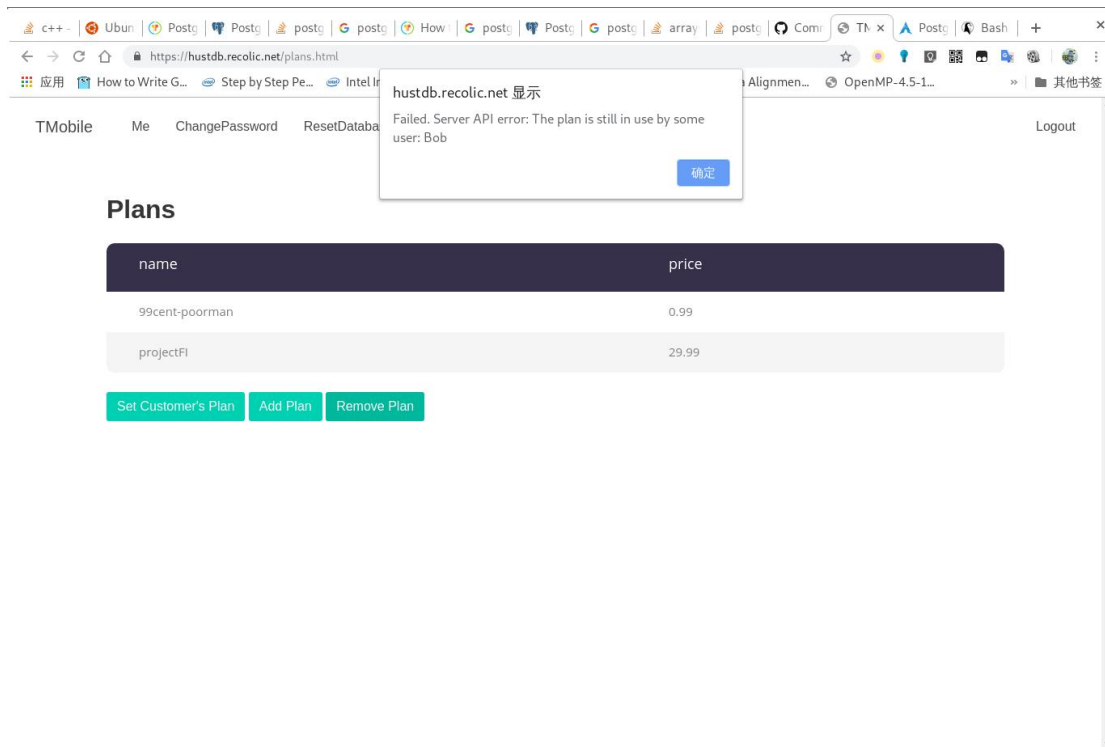
在 Plan 页面可以增加或者删除 Plan。这都需要 customer_service 的权限。
下面我增加一个 Plan 叫做 ctexcelShit , 月费 49.99 美元。



同样，我可以删除它。



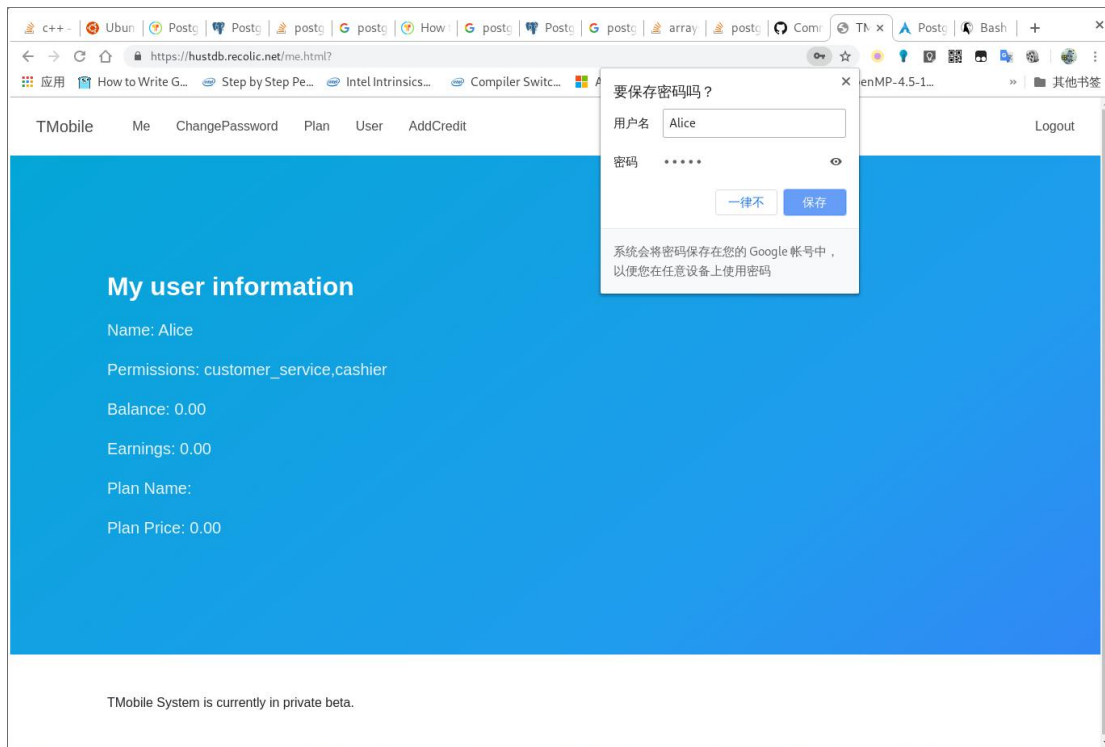
现在，我试图删除一个有人在使用的 plan: projectFI。它会提示这个 plan 有 Bob 在使用，不能删除。



同样，在 plan 页面也可以设置用户的 plan。

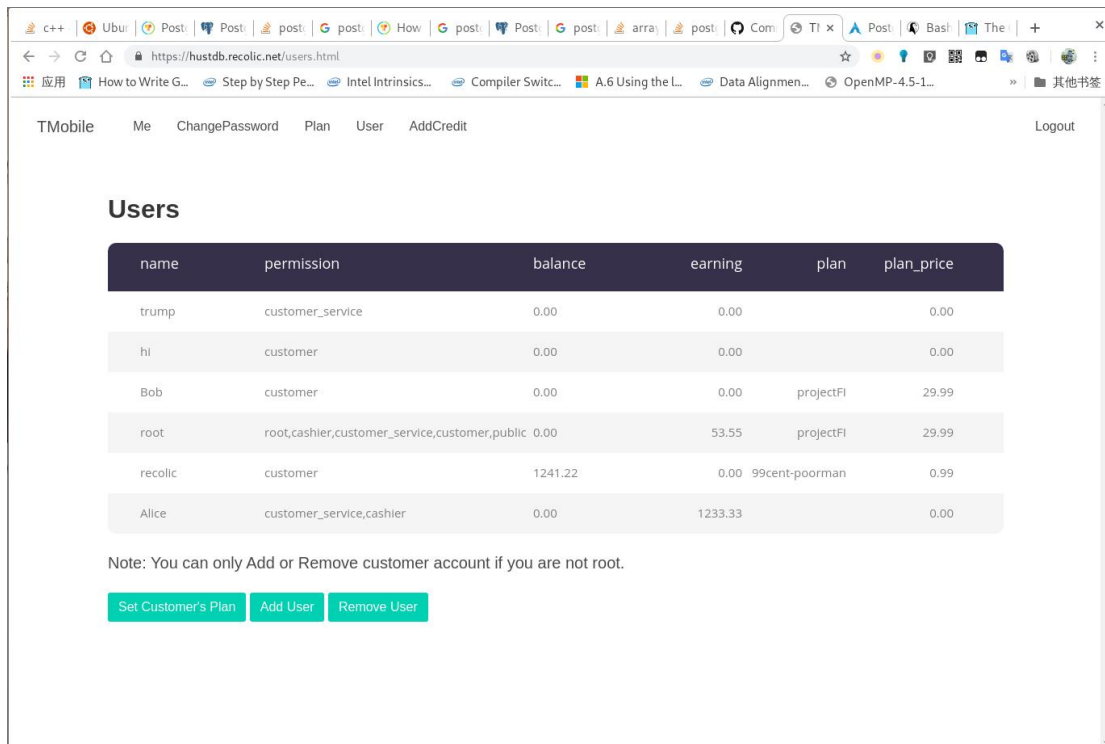
至此，客服的所有功能已经演示完毕。我登录到 Alice，他同时具有客服和收款员的权限，我们尝试收款。

忘了提到，客服在创建一个新的顾客时，它会获得 10 美元的 Earning(介绍费？工资？要求的业绩统计就只能这样了。)。同样，收款员在收银时，也会把收银金额加入到他的业绩中。



如图，cashier 的功能就是给用户充值和扣费。

我使用 Alice 给 Bob 充值 30 元话费，给 recolic 扣除 50 元话费，前后余额如下：



name	permission	balance	earning	plan	plan_price
trump	customer_service	0.00	0.00		0.00
hi	customer	0.00	0.00		0.00
root	root,cashier,customer_service,customer,public	0.00	53.55	projectFi	29.99
Bob	customer	30.0	0.00	projectFi	29.99
recolic	customer	1191.22	0.00	99cent-poorman	0.99
Alice	customer_service,cashier	0.00	1263.33		0.00

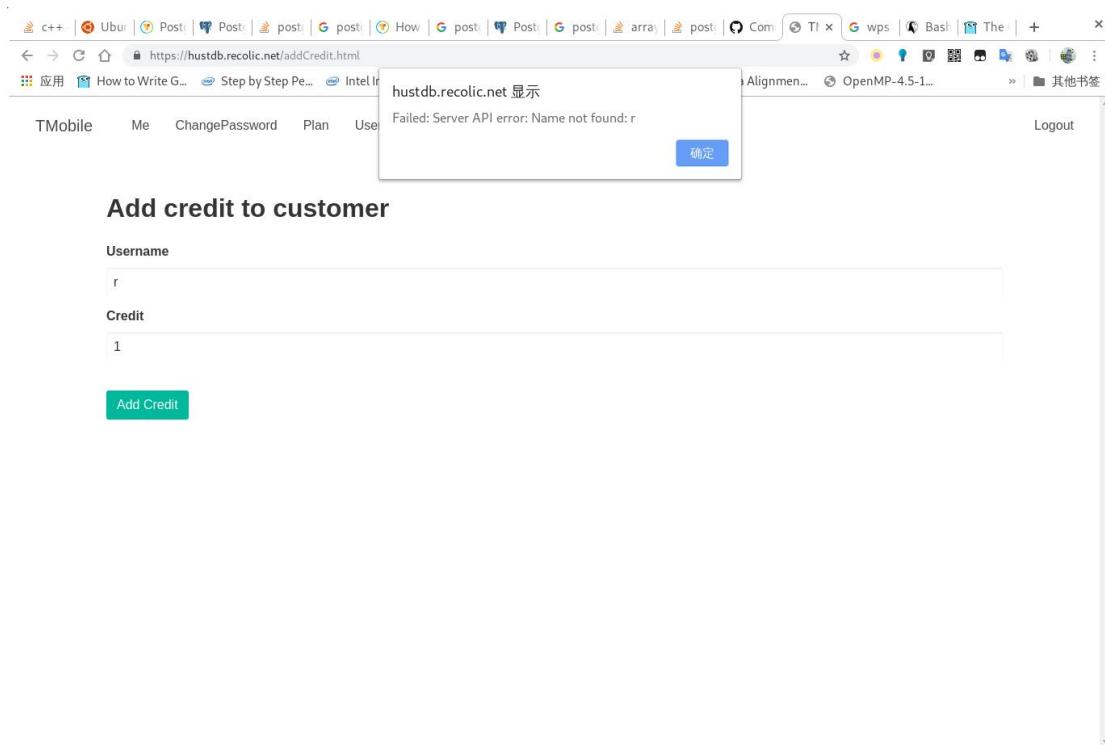
Note: You can only Add or Remove customer account if you are not root.

[Set Customer's Plan](#)
[Add User](#)
[Remove User](#)

尽管显示时可能有所省略，但金额在系统中以两位小数存储，过多的小数位会被截断，过少的小数位会被补 0。充值话费为正 credit，扣除话费只需要在 AddCredit 处输入负数。

充值 30 元话费后，可以看到 Alice 获得了 30 元的 earning。

我测试了无效用户名、无效金额等错误输入，系统均正常提示错误。前后端都限制了 credit 为小数而不是任意字符串。



4.7 系统设计与实现总结

系统特点

1. Design security from start: 从开始设计系统时就考虑到安全因素。例如，token 生成算法考虑了精通密码学的攻击者。
2. 后端不信任前端，不信任任何用户输入。例如找回密码的 email。
3. 在全栈开源、攻击者了解系统一切细节的前提下，系统可以利用密钥保持安全。
4. 真正严格的权限管理。
5. Docker 打包，在任何服务器系统上一键部署。可以使用 Kubernetes 等现成的工具进行弹性伸缩和负载均衡。
6. 数据库与后端分离，后端与前端分离，可以对任意一个部分单独进行集群部署和负载均衡，缓解了性能瓶颈。后端支持将 session 分散到不同的服务器，除 session 内状态外，服务器是无状态的。有极好的并行优化空间。
7. Go-pg 自动进行 SQL 语句生成/转义，直接防止了 SQL 注入的可能，而且能

够减少 bug。后端代码量极少，却支持 CSRF Token 等大量高级功能。

8. 完善的 ACID 考虑，在必要的时候使用事务保证数据一致性。

9. 完善的并行优化。Go 是高度优化且对并行支持良好的编译型后端语言。

10. Nginx 支持 https、http2、HSTS 等完善的现代安全特性。可以与 Cloudflare、Cloudfront 等工具配合使用。示例 :<https://hustdb.recolic.net/>

11. 前端 web 有自适应，电脑、手机、平板电脑、kindle 等设备都可以自适应的绘制页面。在各种操作系统、各种设备的前端浏览器下均能绘制足够好看的界面。根据用户权限动态绘制页面。自然地，它支持跨设备记住密码等浏览器特性。

5 课程总结

这次课程实验共有 3 个任务，难度逐个递增。在第一二个实验中，我完成了与数据库有关的基本操作，包括用户、表的创建，选择与更改语句的实现以及对于事务的实现，这为后续的实验打下了坚实的基础。最后，在前两个实验的基础上，完成了综合性极强的第三个实验，从设计到实现再到测试，不仅将数据库的知识运用的淋漓尽致，更是应用了包括软件工程、算法、Go、Web/Js、Docker、计算机系统、计算机网络在内个各种知识，结合这所有的知识设计了一个功能完善，性能优越的系统。

这些实验首先让我巩固了数据库的知识，对于数据库的应用从生手达到了熟手，此外，对于系统能力的培养也是十分有帮助的，尤其是第三个系统设计与实现的实验，通过查找资料以及亲身实践，我对于实际的数据库在软件开发中的应用有了更为深入的了解。在这次实验中，给我体会最深的时文档查阅的重要性。许多的接口查询以及错误原因的排查都是通过文档解决的，文档能够对于一个系统有详细的描述，从而让人易于使用，这也是在实现系统设计实验时在代码中嵌入了详细的多性格呢文档的原因。此外，细心和耐心也是十分重要的。在发生错误时，这两项是排除错误的必要条件。总体来说，我对于这一次课程实验的实现效果还是十分满意的，在有限的时间内三个实验均达到了较为理想的结果。不过我希望实验时间能够更为充足，过大的课程压力使得实验时间过短，导致综合实践任务中的一些可使程序跟完整的功能没有完成，一些可以使系统性能更为高效的优化没有实现，不得不说有一些小小的遗憾。不过总体上，这次实验给我带来的收货是十分丰富的。

附录 参考文献

- [1] Mihailenco, V. (n.d.). Package go-pg. [online] go-pg godoc. Available at: <https://godoc.org/github.com/go-pg/pg> [Accessed 18 Jun. 2019].
- [2] Golang.org. (2019). Documentation - The Go Programming Language. [online] Available at: <https://golang.org/doc/> [Accessed 18 Jun. 2019].
- [3] Postgresql.org. (2019). PostgreSQL: Documentation. [online] Available at: <https://www.postgresql.org/docs/> [Accessed 18 Jun. 2019].