数据安全传输基础设施平台

# 1引言

## 1.1编写目的

## 1.2术语和定义

## 1.3参考资料

# 2总体设计方案

## 基础组件API封装

1. 基础组件API(socket/pthread/db/der/shm/log/ccl)封装实际上就是将Linux系统调用函数进行封装;

## 代码编写总体流程:

1. 基础组件数据结构struct, 数据库db表结构;
2. 基础组件api(.h) -> 基础组件实现(.c); //实现一个测一个
3. 基础组件api测试用例;
4. 业务组件(子系统/模块)api(.h);
5. 模块(子系统)主程序main.c;
6. 业务组件api实现(.c); //实现一个测一个

## 方案架构

问题:

1. client端MFC程序是否能单独运行(连接server/内存共享key); 可以
2. 如果问题1可以, 那么第三方发送业务数据app如何与client端进行进程间内存共享key(IPCS);
   1. 解决方法: 在win下创建共享内存;
   2. Win<->Linux进行共享内存操作时, 需要通过在上层将共享内存api进行抽象, 以屏蔽不同操作系统间的api调用;

## 方案子系统

1. 基础组件api:

|  |  |
| --- | --- |
| 01 | 统一数据库访问组件libicdbapi |
| 02 | 统一通讯SocketApi组件 |
| 03 | 统一报文编解码组件  Libmessagereal |
| 04 | 统一共享内存读写组件 |

1. keymngserver与 keymngclient通过报文(json/xml…)进行密钥协商, 传递报文中必须要携带发送报文一方的1. client\_ID; 2. server\_ID; 3. timeStamp交易时间; 4. 随机数xx; 5.业务类型码 (组成struct)
2. Oracle:
   1. 创建了2个用户 :
      1. SECMNG SECMNG
      2. SECMNGADMIN 123456
3. odbc配置:
   1. 系统DSN, oracle client 32bit
   2. Win: secmngserver启动前要先删掉ini文件

## 代码排错方法:

1. 程序出错后, 先定位故障, 看日志;
2. 定位故障范围: 在代码中打debug日志进行调试:
   1. KeyMng\_Log(\_\_FILE\_\_, \_\_LINE\_\_, KeyMngLevel[4], ret, "1111111111111111111");
   2. KeyMng\_Log(\_\_FILE\_\_, \_\_LINE\_\_, KeyMngLevel[4], ret, "2222222222222222222");

## 2.4运行环境

### 软件环境

缺少动态库so, 需要配置环境变量

[bryan@oracledb ~]$ tar zxvf keymng\_linux\_bin.tar.gz

bin/

bin/keymngclient

bin/cleanipc

bin/mystop

bin/keymngserver

lib/

lib/libicdbapi.so

lib/libitcastsocket.so

lib/libclntsh.so

lib/libmessagereal.so

lib/aaa.png

etc/

etc/bash\_profile

log/

[bryan@oracledb ~]$ ll

total 13980

drwxr-xr-x 2 bryan bryan 4096 Oct 27 2015 bin

drwxr-xr-x 2 bryan bryan 4096 Oct 27 2015 etc

-rw-r--r-- 1 bryan bryan 14297943 Oct 27 2015 keymng\_linux\_bin.tar.gz

drwxrwxr-x 2 bryan bryan 4096 Oct 27 2015 lib

drwxrwxr-x 2 bryan bryan 4096 Oct 27 2015 log

[bryan@oracledb ~]$ cd bin

[bryan@oracledb bin]$ ls

cleanipc keymngclient keymngserver mystop

[bryan@oracledb bin]$ ldd keymngserver

linux-vdso.so.1 => (0x00007fffa01a5000)

libpthread.so.0 => /lib64/libpthread.so.0 (0x00000031f3400000)

libitcastsocket.so => not found

libmessagereal.so => not found

libclntsh.so.11.1 => not found

libicdbapi.so => not found

libc.so.6 => /lib64/libc.so.6 (0x00000031f2c00000)

/lib64/ld-linux-x86-64.so.2 (0x00000031f2800000)

### 硬件环境

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编号 | 分工 | 负责模块 | 计划任务 | 备注 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# 3总体流程设计

# 4子系统设计

## SecMngServer子系统设计

### 知识点

1. 读取配置文件, 进程初始化, 可以使用ccl;
2. 程序错误处理要健壮;
3. 如何打debug日志:
   1. 在函数开始处: KeyMng\_Log(\_\_FILE\_\_, \_\_LINE\_\_, KeyMngLevel[2], 2, "func MngServer\_Agree() Begin");
   2. 在函数结束处: KeyMng\_Log(\_\_FILE\_\_, \_\_LINE\_\_, KeyMngLevel[2], 2, "func MngServer\_Agree() End");
4. 多线程模式下的gdb调试:
   1. 查看指针地址值: (gdb) p \*msgkeyReq
5. 将配置信息(ip, port, 数据库…) 放到一个struct结构体中 typedef struct \_MngServer\_Info;
6. fork()子进程:
   1. fork()一次调用, 2次返回;
   2. 写时复制;
   3. 1变2 -> 2变4 -> 4变8; (2的fork次方: 2的3次方= 8);

[root@oracledb tmp]# vi fork\_test.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

int main()

{

pid\_t wpid;

int i = 0;

fork();

fork();

fork();

printf("hello\n");

for(i = 0;i<3;++i)

{

wpid = waitpid(-1, NULL, 0);

}

return 0;

}

//打印8个hello

1. Linux 内核在2.4~2.6时变化较大, 实现了热插拔(USB设备), 底层提前定义好一套接口(回调函数), 硬件厂商按照接口驱动设备, 由此Linux内核实现了驱动未来设备的能力, 底层的原理就是多态;
2. 守护进程:
   1. 每一个ssh就是一个session;
   2. 每个session都有n个进程组;
   3. 进程变为守护进程, 就要从进程组中跳出来, 在Linux中使用setsid()即可;
      1. setsid() creates a new session if the calling process is not a process group leader.
   4. 父x->子x->孙√

#define INIT\_DAEMON \

{ \

if(fork() >0) exit(0); \

setsid(); \

if(fork()>0) exit(0); \

}

* 1. 用子进程创建的孙子进程不再具有连接父进程session的能力;

1. 利用信号(SIGUSR1)结束守护进程:
   1. Server注册signal回调函数;
      1. signal(SIGUSR1, mysighandler\_t);
   2. Linux内核调用server的信号处理回调函数;
      1. shell脚本: kill -10 keymngserver\_PID
2. 函数指针/回调函数 做函数参数
   1. 看到含有 函数指针做函数参数的api函数,它的意思是说: 谁调用我,谁提供函数的入口地址

typedef void (\*sighandler\_t)(int);

sighandler\_t signal(int signum, sighandler\_t handler);

* 1. server端main函数调用signal函数, 就要提供函数指针handler的入口地址mysighandler\_t(函数定义就是函数的入口地址):

signal(SIGUSR1, mysighandler\_t);

* 1. 意义:
     + - * 以前人写的框架,能调用后来人写的代码 ====>可扩展 C++/Java多态;
         * 实现了任务的编写者 和 任务的调用者进行解耦合

1. Shell脚本结束server守护进程:
   1. 坑: /bin/sh^M: bad interpreter: No such file or directory
      1. Bash不能解析win格式, 只能解析unix格式;
      2. 解决:

vi 脚本:

利用如下命令查看文件格式

     :set ff 或 :set fileformat

     可以看到如下信息

     fileformat=[dos](http://www.2cto.com/os/dos/) 或 fileformat=unix

     （3） 利用如下命令修改文件格式

     :set ff=unix 或 :set fileformat=unix

     :wq (存盘退出)

### 问题

### 子系统外延

1. 服务端需要成为守护进程;
2. 服务端需要优雅的退出, 释放所有的资源;

### 系统子模块组成

### 系统业务流程

## SecMngServerAdmin子系统设计ＭＦＣ

### 知识点

1. MFC界面开发(UI), 先作出demo与客户确认后再继续开发, 以避免以界面美丑原因推到重做;
2. 设计界面可以仔细参照microsoft微软的产品: 1:N, N:N, tab标签页, 分组, 图片位置等;
3. UI界面的功能需求通常是: 1个功能点背后对应着4~5个功能点(xx管理🡪xx增删改查);
4. MFC中*RUNTIME\_CLASS(反射):*
   * 用*RUNTIME\_CLASS*(CFormBackground)创建对象的类需要 *DECLARE\_DYNCREATE*(CFormBackground) 声明;
5. 多标签页: tab
   * 后台业务管理 CSysAll 继承自*CformView;*
   * Tab标签视图类 CDlgTab1, CDlgTab2, CDlgTab3 继承自Cdialog类;
6. MFC为控件添加类向导只能添加默认的类 CTabCtrl myTab;
   * 增强tab标签页, 需要手工添加 TabSheet.h TabSheet.cpp文件到项目中;
   * *CTabCtrl* myTab;
7. *将*.c文件换成.cpp文件需要包含头文件:

#include "stdafx.h" //别忘记 增加这个头文件的说明

### 问题

### 子系统外延

1. 开源网站:　 <http://www.codeguru.com> (GFX)
2. 使用第三方库时, 先找到测试用例(官方);
3. MFC将项目的WinMain函数封装在了项目的CxxApp类的InitInstance()函数中;
4. C++编译器会先.h文件放到.cpp文件中;
5. C/C++程序会在运行时先为全局变量(CMyAdminApp theApp;)分配内存, 调用其构造函数, 然后再执行WinMain函数;
6. MFC会在项目程序启动时先执行CxxApp类的构造函数CMyAdminApp::CMyAdminApp(), 然后隐式调用*BOOL* CMyAdminApp::InitInstance()函数, 从而初始化Doc, Frame, View类

*RUNTIME\_CLASS*(CMyAdminDoc),

*RUNTIME\_CLASS*(CMainFrame), // 主 SDI 框架窗口

*RUNTIME\_CLASS*(CMyAdminView));

CmyAdminDoc

theApp 🡪 pDocTemplate CmainFrame

CMyAdminView

1. MFC框架可以通过全局变量 CMyAdminApp theApp; 找到模板类 pDocTemplate, 从而通过pDocTemplate找到 CmyAdminDoc, CmainFrame, CmyAdminView 对象;
2. 为MainFrm.app重写虚函数OnCreateClient(), 利用多态原理, 使MFC程序可以调用后来写的代码;
   1. CMainFrame 继承自*CFrameWnd* : class CMainFrame : public *CframeWnd;*
   2. 当MFC调用*CframeWnd(基类)的*OnCreateClient()虚函数时(程序运行时RUNTIME), 会运行其子类CmainFrame对象重写的虚函数OnCreateClient(), 从而实现多态;

### 系统子模块组成

1. 视图切分:　使用第三方库　导入ＧＦＸ 9个文件(需要添加３个ｃｕｒ资源文件,　宏定义);
2. vc2015支持中文需要将项目的属性->字符集 改为 使用多字节字符集;

### 系统业务流程

## SecMngClient子系统设计

### 知识点

1. gdb调试: 跳出多次循环, 通过 until 89 (行号)的方法;
2. 当无法通过gdb对程序进行调试时, 只能通过日志log的方式, 因此日志log信息一定要包含相关配置信息, 并尽可能详细;
3. 读取配置文件, 进程初始化, 可以使用ccl;

### 问题

1. 密钥注销流程: 将共享内存中的 NodeSHMInfo status = 1即可;

### 子系统外延

### 系统子模块组成

1. 主程序 调用-> 业务组件 调用-> 基础组件:

keymngclient.c(main.c主程序) 调用-->

keymngclientop.h/keymngclientop.c(业务组件) 调用->

icdbapi.h/keymng\_msg.h/keymnglog.h/poolsocket.h/myipc\_shm.h(基础组件)

1. keymngclientop.h 是client端业务框架的头文件;

### 系统业务流程

1. client端根据r1和r2的随机数产生一个密钥, 然后将密钥写入共享内存;
2. 密钥校验: seckey是128位的, 而随机数r1只有64位, 可以只校验前64位或后64位,(或前32位加后32位);

## 外联接口appinterface

### 知识点

1. 外联加密接口: int AppCryptApi(int crypttag, char \*clientid, char \*serverid, unsigned char \*indata, int indatalen, unsigned char \*outdata, int \*outdatalen, int cfg\_shm\_keyid, int cfg\_shm\_maxnodenum);
2. 外联接口加密数据工作流程: 第三方App 🡪 appcryptapi.h/libappinterface.so 🡪从共享内存中获取密钥key 🡪 调用des加密算法对数据进行加密;
3. cryptproj项目: 加密算法des:
   1. 对于非8正数倍的数据采用缺n补n个n的策略: 如 缺5个数就在末尾补55555(5个5), 缺3个数就补333;
   2. 对外提供的加密接口应该有2套:
      1. 一套是8的整数倍的(不带补丁no padding);
      2. 另一套是缺n补n个n的(带补丁with padding);
4. 加密必须要按照标准规范进行;
5. tmain.c:
   1. FileSymDec(char \*filename1, char \*filename2): 将文件1加密后生成为文件2

### 问题

### 子系统外延

### 系统子模块组成

### 系统业务流程

# 5 通用组件设计

## 报文编解码组件

1. DER则是BER一个子集, DER（Distinguished Encoding Rules，可辨别编码规则）

TLV(Tag, Length, Value)

1. 二级指针函数传参, 目的是在函数内部开辟内存空间, 操作, 并把开辟的指针地址传出来(传出参数)

### 设计原则

1. 统一报文编解码API: 对底层struct api进行抽象, 参数使用 void \* 万能指针, 和类型int type

2. 统一报文编解码API的作用:

1)对底层大量的struct进行了屏蔽

2)对报文编解码格式(ber/html/json/xml)进行了屏蔽;

3)与上层业务逻辑进行了有效的解耦合;

1. 程序中一个入口, 多个出口return ret, 不合理, 应该goto到统一出口
2. 程序出口处要free释放内存
3. 如何将需求转化设计方案(struct/db)
4. 开发流程: struct->\*.h(api:函数名/函数参数/返回值)->main->\*.c
5. Debug流程: F9 breakpoint->F5->F10(F11) 检查变量值
6. 快速调用别人写好的api, 通过写测试用例程序, 从而了解功能, 学会如何使用api;
7. 抽象api, 对上层业务逻辑解耦合, api传参(void \*)/int type;
8. 随写随编译测试F7

### 知识点:

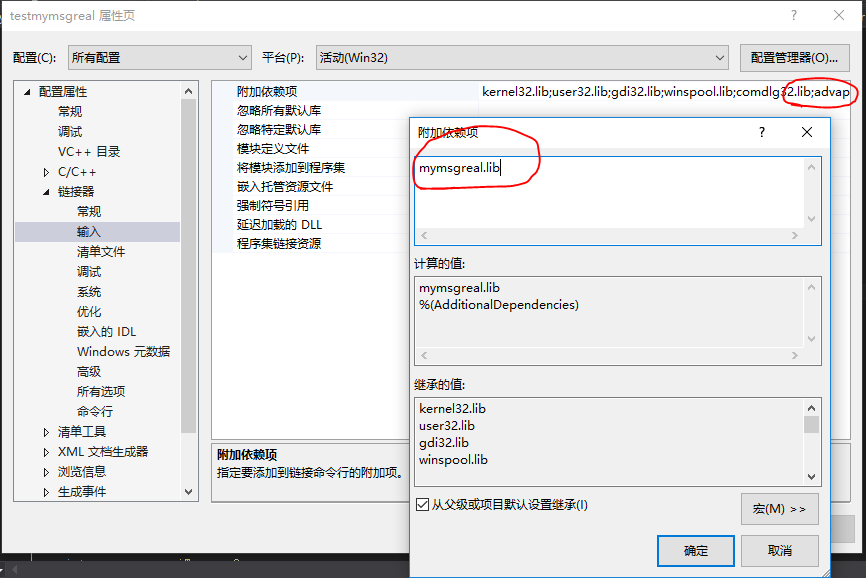
1. 二级指针传出参数;
2. strcpy: 字符串拷贝, 末尾加’\0’;
3. memcpy: 指针指向的内存地址拷贝
4. 链表操作: \*next/\*prev/\*pTmp当前节点指针/\*pHead表头代表整个链表
5. 内存操作: malloc->memset->memcpy…->free
6. 文件I/O: FILE \*fp -> fopen->fwrite->fclose;
7. 内存释放free: 由内而外释放free; 二级指针传参, 以避免野指针;
8. 给变量赋值实际上是给变量代表的内存空间赋值;
9. 编解码: struct <==>二进制码流/字节流(unsigned char \*);
10. AnyBuf: 是双向链表LinkList的节点;
11. Unicode编码:
    1. UTF8: 65001
    2. UTF16 LE: Unicode 1200 Little-Endian;
    3. UTF16 BE:Unicode 1201 Big-Endian;
12. 每次函数调用后, 都要判断返回值, 是否有错误, 并进行错误处理;
13. 每次malloc分配内存之后, 都要对分配结果是否为NULL进行判断, 并进行错误处理;
14. 要对自定义类型和错误类型进行宏定义;
15. 错误处理中要释放相应的内存, 函数返回return前要释放相应内存;
16. 传入参数inXX应该加const, 禁止修改;

### 实现流程

1. 统一报文编解码的实现: 在使用void\*之前的基础上, 增加一层 iType TLV的封装操作;

### 动态库知识

1. Win下创建动态库项目;
2. 不能用中文, 也不能有空格;
3. dll不是exe,不能直接运行;
4. 函数导出符\_\_declspec(dllexport), \*.c和\*.h都要加;
5. 动态库路径: D:\Code\LinuxC\_CppCode\Itcast\_Linux\_C\SecMng\mymsgreal\Debug
6. .dll 是动态库文件
7. .lib是动态库的描述文件
8. 动态库的测试:
   1. 创建测试项目,
   2. copy: .dll .lib .h 到测试项目目录下,
   3. 测试项目中添加.h文件
   4. 为vs2015添加dll路径



1. 测试程序**执行**时找不到动态库dll文件(win7): 将动态库dll在exe文件路径下再copy一份, 或添加环境变量;
2. 动态库的调试方法(vs2015):
   1. 同时打开动态库项目和动态库测试项目;
   2. 再测试项目的Debug过程中, 可以按F11直接跳转到动态库的项目源码中
3. 调试dll动态库的**坑**:
   1. \*.exe文件同级目录下有一个my1.dll,

同时window系统还有一个 C:\Windows\SysWOW64\my1.dll

造成无法调用\*.exe同级目录下的dll

1. **动态库api中malloc的内存, 必须调用动态库提供的api进行释放free, 如果不使用api进行free, Release版本中程序会down掉;**
2. 移植Linux动态库so:
3. Makefile原理:
   1. Makefile三要素: 目标:依赖

命令(规则)

* 1. 当依赖发生更新时目标将按照规则重新生成;
  2. VPATH是makefile内建特殊变量: 自动查找.c .o文件路径

15: 编码问题导致编译报错:

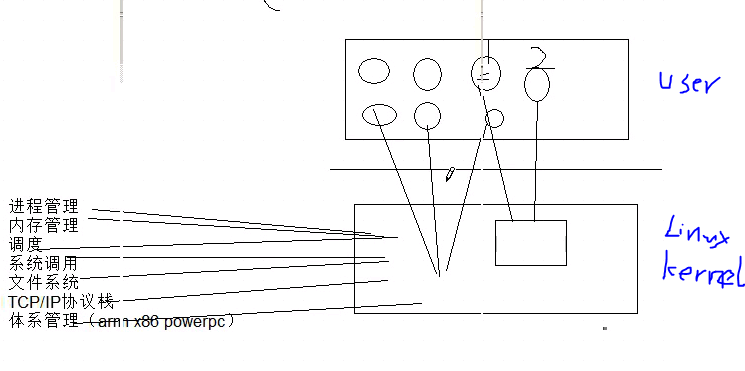
warning: null character(s) ignored

需要统一编码(UTF8)

Linux 查看系统编码: locale 或 cat /etc/sysconfig/i18n

### 接口形式

## 共享内存操作组件

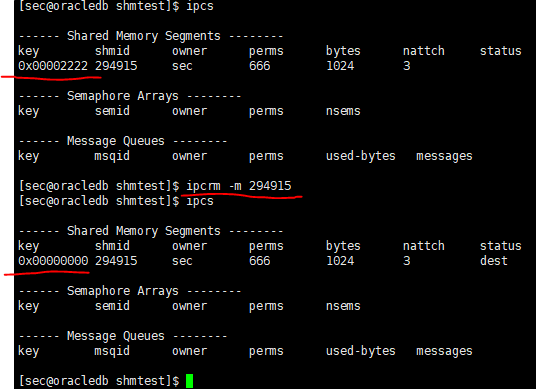


### 设计原则

1. 抽象共享内存api以屏蔽Win/Linux操作系统的差别;
2. 由server端/client端创建共享内存, 供外联应用app进行调用;
3. 如果要创建的共享内存已经存在, 那么必须使用已经存在的共享内存;
4. 有多个外联应用app的客户端连接共享内存, 所以需要创建共享内存数组进行存放;
5. 共享内存中的密钥需要有状态(可用/注销…) status: 0 / 1;
6. 共享内存api设计: Server端和client都需要操作共享内存, 所以对共享内存操作再做一次抽象, 如: int KeyMng\_ShmInit(int key, int maxnodenum, int \*shmhdl);
7. 协商密钥:
   1. r1,r2为server端与client端的随机数;
   2. r1: abcdef…;
   3. r2:123456…;
   4. 密钥: a1b2c3d4e5…;
   5. server端和client端都按照这种规则生成密钥, 保证密钥一致性;

### 知识点:

1. 查看/删除共享内存命令:
   1. 查看: ipcs
   2. 删除: ipcrm –m/–s / -Q
2. TCP/IP协议栈在Linux内核中
3. 进程1,2,3使用Linux的同一个共享内存句柄shmid: 0x123456, 而进程4将0x123456进行删除, 此时进程1,2,3还能使用0x123456共享内存吗?



**IPC\_PRIVATE**

**私有状态下(已删除)的共享内存是不能再被其它进程连接了**

**引用计数技术(jvm**内存回收同理**)**

1. shmget(0x3333, 1024, 666);
2. shmat: void \*shmat(int shmid, const void \*shmaddr, int shmflg);
3. int shmdt(const void \*shmaddr);
4. int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);
   1. struct shmid\_ds: 共享内存信息结构体
5. Linux中当创建共享内存的进程被kill了, 那么那个被它创建的共享内存**不**会同时消失;
6. Win中当创建共享内存的进程被kill了, 那么那个被它创建的共享内存**会**同时消失;
   1. 因此在Win的Client中需要让应用进程始终不退出;

### 问题

1. 需要完整的共享内存抽象api

### 实现流程

## 统一通信组件socket

### 设计原则

服务端1套api, 客户端2套api(单连接/连接池pool)

### 知识点

1. TCP/IP协议栈在Linux的内核中;
2. Client与server进行socket连接需要通过Linux内核;
3. socket高性能 = 高并发 + 连接池 +长连接;
4. 性能瓶颈来自于client端, 而不是server端, 因此需要在client端建立连接池;
5. 实现连接池: 创建多个长连接connect(), 保存fd到链表/数组中, 并发使用fd时加锁(mutex/rw);
6. server端利用多进/线程响应client端并发连接, server端多进/线程循环响应client端报文, 只有client端主动发起关闭连接, server端才结束进/线程;
7. 连接池: 生产者/消费者模型;
8. 指针2大知识体系:
   1. 指针做函数参数, 做接口封装和设计;
   2. 函数指针做函数参数, 回调函数(多态);
9. 函数指针(回调函数):
   1. 谁调用带有函数指针作为参数的api函数, 谁就提供回调函数的入口地址;
   2. 作用: 把任务的编写者和任务的调用者进行解耦合;
   3. 多态的基础;
10. 连接池pool方案中, client的线程处理函数可以判断出socket连接是否正常, 如果正常就给连接池放回0, 如果socket连接出错就返回1;
11. 线程安全的问题: client端 threadNum 不准确

产生原因: for循环的速度快于创建thread的速度, 因此必须分配独立内存空间存储i的值

for (i=0; i<iThreadNum; i++)

{

ThreadInfo \*pThreadInfo = (ThreadInfo \*)malloc(sizeof(ThreadInfo));

memset(pThreadInfo, 0, sizeof(ThreadInfo));

pThreadInfo->handle = handle;

pThreadInfo->iLoop = iLoop;

pThreadInfo->iArrayIndex = i;

pthread\_create(&pidArray[i], NULL, myclient\_routine, (void \*)pThreadInfo);

}

1. TCP/IP协议可以保证数据的安全到达, 但无法保证数据到达的正确分片顺序(粘包问题)

问题产生:

1.exe向2.exe发送数据: close(fd) -> 321 -> cba

数据在1.exe关闭前可安全到达2.exe, 但abc - 123 的到达顺序无法保证,有可能是 ab - c123 或 abc1 – 23

解决方法:

* + - * 1. 发送数据时在数据末尾添加结束标志 \n;
        2. 发送数据时在头部添加数据长度;

socketutil.h:

ssize\_t readn(int fd, void \*buf, size\_t count);

ssize\_t writen(int fd, const void \*buf, size\_t count);

1. 管道破裂问题:
   1. 产生原因: client端 发出close(fd), 但server端未处理, 然后继续通过socket向client端发送数据write, 此时产生管道破裂;
   2. 现象(危害): 产生管道破裂后, server端会收到client端发来的紧急信号SIGPIPE, 其默认动作会关闭server端应用程序, 导致服务端down掉;
   3. 解决方法:
      1. server端调用了signal(SIGPIPE, SIG\_IGN), 这样产生 SIGPIPE 信号时就不会中止程序，直接把这个信号忽略掉;

### 问题

### 实现流程

## 数据传输平台数据库组件

### 设计原则

### 知识点

1. Oracle数据库连接池句柄为全局变量: ICDBHandle

### 问题

1. 编译testdbapi.c时, 缺少oracle动态链接库so:

坑libnnz11.so, needed by…

解决:

Oracle 11gR2 RHEL 6.5 X64

[root@oracledb sec]# vi /etc/ld.so.conf

include ld.so.conf.d/\*.conf

/opt/oracle/app/product/11.2.0/dbhome\_1/lib

ldconfig 生效

sec用户:

[sec@oracledb testdbapi]$ cat ~/.bash\_profile

PATH=$PATH:$HOME/bin:$HOME/lib

LD\_LIBRARY\_PATH=$HOME/lib

export PATH

export LD\_LIBRARY\_PATH

[sec@oracledb testdbapi]$ ll ~/lib

total 47776

-rw-r--r-- 1 sec sec 48725761 Sep 1 08:40 libclntsh.so

-rw-r--r-- 1 sec sec 92777 Sep 1 08:34 libicdbapi.so

-rw-r--r-- 1 sec sec 41822 Aug 30 07:46 libitcastsocket.so

-rw-rw-r-- 1 sec sec 50046 Aug 28 12:25 libmymessagereal.so

[sec@oracledb testdbapi]$ ldd testdbapi

linux-vdso.so.1 => (0x00007fffe1dff000)

libpthread.so.0 => /lib64/libpthread.so.0 (0x00000031f3400000)

libclntsh.so.11.1 => /opt/oracle/app/product/11.2.0/dbhome\_1/lib/libclntsh.so.11.1 (0x00007f9194a00000)

libicdbapi.so => /home/sec/lib/libicdbapi.so (0x00007f91947f0000)

libc.so.6 => /lib64/libc.so.6 (0x00000031f2c00000)

/lib64/ld-linux-x86-64.so.2 (0x00000031f2800000)

libnnz11.so => /opt/oracle/app/product/11.2.0/dbhome\_1/lib/libnnz11.so (0x00007f9194427000)

libdl.so.2 => /lib64/libdl.so.2 (0x00000031f3000000)

libm.so.6 => /lib64/libm.so.6 (0x00000031f3c00000)

libnsl.so.1 => /lib64/libnsl.so.1 (0x0000003203000000)

libaio.so.1 => /lib64/libaio.so.1 (0x000000306b200000)

1. sec用户无法连接oracle数据库:

原因: linux系统没有安装oracle client端

解决: 安装oracle client端, 或在oracle服务器上的sec用户加入oinstall用户组

1. insert 中文数据失败/乱码:

解决:

* 1. .c -> UTF8 (65001);
  2. Linux下执行程序的用户sec环境变量:

NLS\_LANG=american\_america.AL32UTF8

export NLS\_LANG

* 1. Oracle编码:

SQL> SELECT \* FROM NLS\_DATABASE\_PARAMETERS;

NLS\_CHARACTERSET AL32UTF8

1. 查看icdbapi的程序执行日志log:

在用户sec的home目录下 mkdir log 目录

运行./testdbapi 后即可查看

根据日志的第一条报错信息查询: oerr ora 1438 (负数变正数)

### 实现流程

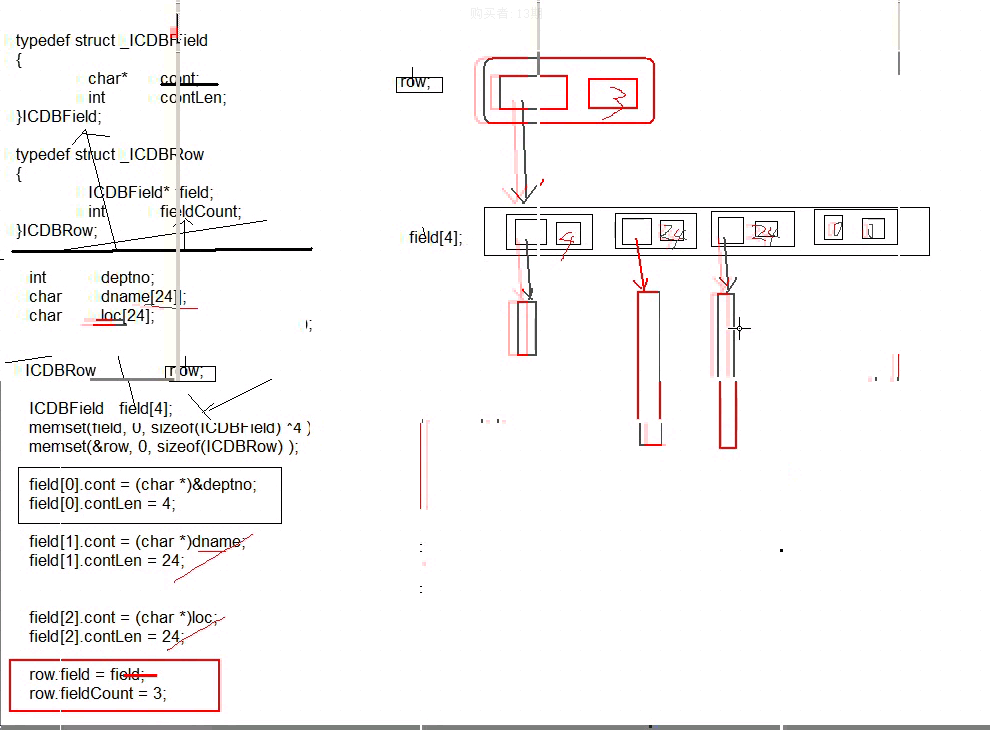
### 数据字典

Select \* from tab;

## 统一数据接口组件

### 设计原则

1. select查询数据库API:
   1. 需要为select语句的查询结果集分配内存以保存数据;
   2. 数据库api接口程序对将要发来的select语句的列数, 结果集行数都不了解;
   3. 发送select语句的一方可以将select语句的列数row信息, 包括列数据类型发送给底层接口程序, 以使其可以根据列信息分配一行数据的内存大小;
   4. select查询结果集数据结构:



1. 数据库的每一笔业务操作(协商密钥Agree/注销…), 都必须开启**事务(重点)**:
   1. 此处如果数据库操作业务的逻辑关系有错误, 会导致业务数据出错, 而且极难排错, 后果及其严重!!!
2. 多线程执行sql语句select数据库时需要加select锁:
   1. select ikeysn from SECMNG.KEYSN for update
   2. 保证每个线程在select ikeysn 到 update SECMNG.KEYSN 之间 keysn不会被改变;
   3. 问题: select…for update 到 update 之间如何保证原子操作atom?
      1. 可以在每个线程操作数据库的 select 之前加 mutex锁, update后解锁;
3. 不可见字符是不能直接存入oracle数据库的:
   1. VARCHAR2只能存储可见字符;
   2. 需要将不可见字符转码成可见字符(使用base64编码规范);
4. Oracle查错(Linux): oerr ora 1

### 实现流程