

之前几个月在Linux上面开发，有比较多的Linux开发经验，但是还是需要系统性进行学习。在QT中采用的是信号槽的方式进行连接，也是利用回调函数实现GUI窗口。所以我需要对这些底层知识再进行充分理解，外加网络编程的进阶部分也是需要良好的linux底层理论知识基础的，所以我还是需要继续深入学习Linux操作系统。

Linux的信号

一.信号的基本概念

信号(signal)是软件中断，是进程之间相互传递消息的一种方法，用于通知进程发生了事件，但是，不能给进程传递任何数据

信号产生的原因有很多，在Shell中，可以用kill和killall命令发送信号

```
kill -信号的类型 进程编号
killall -信号的类型 进程名
```

其中中间是可选项，如果不指定种类那么会有一个缺省值为15，就是终止，如果传的是1，那就是挂起。如果是8，那就是“浮点数例外”。

二.信号的类型

Linux的信号有64种，不同的信号有不同的用途

信号名↵	信号值↵	默认处理动作↵	发出信号的原因↵
SIGHUP↵	1↵	A↵	终端挂起或者控制进程终止↵
SIGINT ↵	2 ↵	A ↵	键盘中断 Ctrl+c ↵
SIGQUIT↵	3↵	C↵	键盘的退出键被按下↵
SIGILL↵	4↵	C↵	非法指令↵
SIGABRT↵	6↵	C↵	由 abort(3)发出的退出指令↵
SIGFPE↵	8↵	C↵	浮点异常↵
SIGKILL ↵	9 ↵	AEF ↵	采用 kill -9 进程编号 强制杀死程序。↵
SIGSEGV↵	11↵	C↵	无效的内存引用↵

SIGPIPE	13	A	管道破裂，写一个没有读端口的管道。
SIGALRM	14	A	由闹钟 alarm()函数发出的信号。
SIGTERM	15	A	采用“kill 进程编号”或“killall 程序名”通知程序。
SIGUSR1	10	A	用户自定义信号 1
SIGUSR2	12	A	用户自定义信号 2
SIGCHLD	17	B	子进程结束信号
SIGCONT	18		进程继续（曾被停止的进程）
SIGSTOP	19	DEF	终止进程
SIGTSTP	20	D	控制终端（tty）上按下停止键
SIGTTIN	21	D	后台进程企图从控制终端读
SIGTTOU	22	D	后台进程企图从控制终端写
其它	<=64	A	自定义信号

处理动作一项中的字母含义如下：

A 缺省的动作是终止进程

B 缺省的动作是忽略此信号，将此信号丢弃，不做处理

C 缺省的动作是终止进程并进行内核映像转储(core dump)，内核映像转储是指将进程数据在内存的映像和进程在内核结构中的部分内容以一定格式转储到文件系统，并且进程退出执行，这样做的好处是位程序员提供了方便，使得他们可以得到进程当时执行的数据值，允许他们确定转储的原因，并且可以调试他们的程序。

D 缺省的动作是停止进程，进入停止状态的程序还能重新继续，一般是在调试的过程中

E 信号不能被捕获

F 信号不能被忽略

三.信号的处理

进程对于信号的处理方式有三种

- 对该信号的处理采用系统默认的操作，大部分的信号的默认操作是终止进程
- 设置中断的处理函数，收到信号后，由该函数来处理
- 忽略某个信号，对该信号不做任何处理，就像未发生过一样

signal()函数可以设置程序对信号的处理方式

函数声明

```
sighandler_signal(int signum, sighandler_t handler)
```

signum表示信号编号

handler表示信号的处理方式，有三种情况

- SIG_DFL:恢复参数signum所指信号的处理方法为默认值
- 一个自定义的处理信号的函数，函数的形参是信号的编号
- SIG_IGN:忽略参数signum所指的信号

```
#include <iostream>
#include <unistd.h>
#include <signal.h>
using namespace std;

void func(int signum){
    cout<<"收到信号:"<<signum<<endl;
}

int main(){
    signal(1,func);
    signal(15,func);
    while(1){
        cout<<"执行了一次任务\n";
        sleep(1);
    }
}
```

```
执行了一次任务
执行了一次任务
执行了一次任务
收到信号:15
执行了一次任务
执行了一次任务
执行了一次任务
执行了一次任务
收到信号:1
执行了一次任务
执行了一次任务
执行了一次任务
□
```

```

caomengxuan@caomengxuan-virtual-machine:~$
caomengxuan@caomengxuan-virtual-machine:~$ killall -15 test
caomengxuan@caomengxuan-virtual-machine:~$ killall -1 test
caomengxuan@caomengxuan-virtual-machine:~$

```

当发送为2的信号，demo程序就被终止

```

caomengxuan@caomengxuan-virtual-machine:~$ killall -2 test
caomengxuan@caomengxuan-virtual-machine:~$

```

```

执行了一次任务
执行了一次任务
执行了一次任务
执行了一次任务
执行了一次任务
caomengxuan@caomengxuan-virtual-machine:~/linuxCpp$

```

这里收到1和15不会收到系统默认操作，而是调用func中的方法。

如果说这里对2进行别的一些操作，也可以达到输入信号2但是不终止的目的

```

#include <iostream>
#include <unistd.h>
#include <signal.h>
using namespace std;

void func(int signum){
    cout<<"收到信号:"<<signum<<endl;
}

int main(int argc,char*argv[]){
    signal(1,func); //注册回调函数func(), 收到信号后, 回调func()函数
    signal(15,func); //回调func()函数的时候, 把信号的编号传给func()函数
    signal(2,SIG_IGN); //忽略2的信号
    while(1){
        cout<<"执行了一次任务\n";
        sleep(1);
    }
}

```

当然，也可以恢复到默认的处理行为

```

#include <iostream>
#include <unistd.h>
#include <signal.h>
using namespace std;

void func(int signum){
    cout<<"收到信号:"<<signum<<endl;
    signal(signum,SIG_DFL); //恢复信号的处理方法为默认行为
}

int main(int argc,char*argv[]){
    signal(1,func);
}

```

```

signal(15,func);
signal(2,SIG_IGN);//忽略2的信号
while(1){
    cout<<"执行了一次任务\n";
    sleep(1);
}
}

```

```

~$ killall -1 test
~$ killall -1 test
~$ 

```

```

执行了一次任务
执行了一次任务
执行了一次任务
执行了一次任务
执行了一次任务
执行了一次任务
执行了一次任务
执行了一次任务
执行了一次任务
执行了一次任务
执行了一次任务
./run.shell: line 4: 42193 Hangup
./test

```

注意，这里第一次收到信号，会先进入到func中将信号的处理方法恢复为默认行为，第二次再发送信号1的时候程序才会将程序挂起

当然，不是所有的信号都能自定义处理。比如说信号9，是强制杀死进程的，它被设定为无法捕获，而且也无法被忽略。这样应该也是为了防止程序无法终止。

可以通过设置闹钟(定时器)，来在开发中实现定时执行某些任务的操作

```

#include <iostream>
#include <unistd.h>
#include <signal.h>
using namespace std;

void func(int signum){
    cout<<"收到信号:"<<signum<<endl;
    signal(signum,SIG_DFL); //恢复信号的处理方法为默认行为
}

void func1(int sig){
    cout<<"闹钟响了，执行定时任务\n";
    alarm(5);
}

int main(int argc,char*argv[]){
    signal(1,func);
    signal(15,func);
    signal(SIGINT,SIG_IGN);//忽略2的信号

    alarm(5);
    signal(14,func1); //设置定时任务函数
}

```

```
while(1){
    cout<<"执行了一次任务\n";
    sleep(1);
}
}
```

```
执行了一次任务
执行了一次任务
执行了一次任务
执行了一次任务
闹钟响了，执行定时任务
执行了一次任务
执行了一次任务
执行了一次任务
执行了一次任务
执行了一次任务
闹钟响了，执行定时任务
执行了一次任务
```

四.信号有什么用

服务运行在后台，如果要终止程序，杀死程序不是个很好的办法，因为程序被杀会导致程序突然死亡，没有安排善后工作

如果向服务程序发送一个信号，服务程序收到这个信号之后，调用一个函数，在函数中编写的善后代码，程序就可以有计划的退出

向服务程序发送0的信号，可以检测程序是否存活，如果没有错误就是还在运行。

五.信号应用实例

在实际开发中，在main函数开始的位置，程序员会先屏蔽掉全部的信号

```
#include <iostream>
#include <unistd.h>
#include <signal.h>
using namespace std;

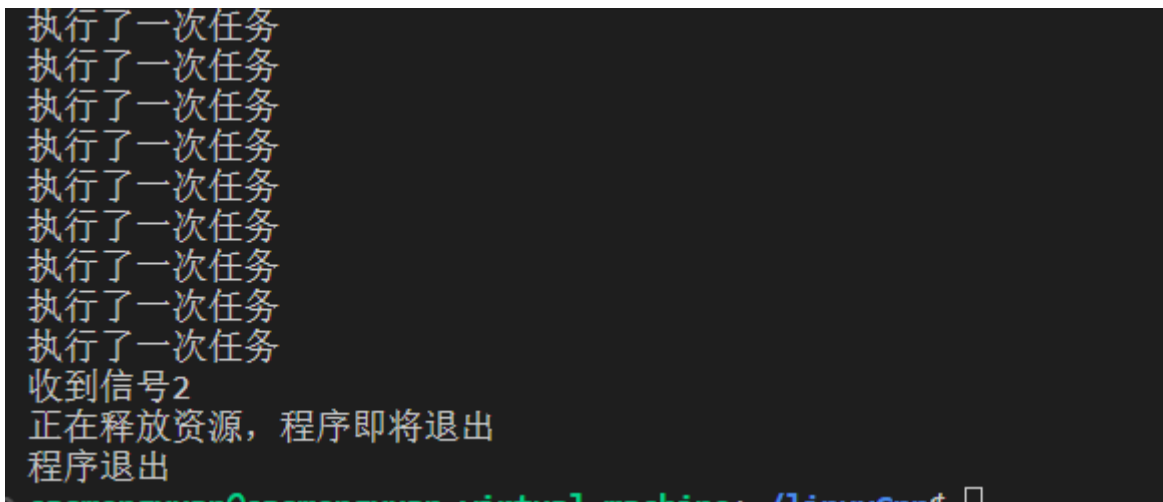
void EXIT(int sig){
    cout<<"收到信号"<<sig<<endl;
    cout<<"正在释放资源，程序即将退出\n";

    cout<<"程序退出\n";
    exit(0);
}

int main(int argc,char*argv[]){
    //忽略全部的信号，防止程序被信号异常终止
    for(int i=1;i<=64;i++)signal(i,SIG_IGN);

    //如果收到2和15的信号，本程序将主动退出
    signal(2,EXIT);
    signal(15,EXIT);
}
```

```
while(1){
    cout<<"执行了一次任务\n";
    sleep(1);
}
}
```



六.发送信号

Linux操作系统提供了kill和killall命令向程序发送信号，在程序中，可以用kill()库函数向其它进程发送信号

函数声明

```
int kill(pid_t pid,int sig)
```

kill()函数将参数sig指定的信号给参数pid指定的进程

参数pid有几种情况

- pid>0, 将信号传给进程号为pid的进程
- pid=0, 将信号传给和目前进程相同进程组的所有进程，常用语父进程给子进程发送信号，
- pid=-1,将信号广播传送给系统内所有进程，例如系统关机的时候，会向所有的登录窗口广播关机信息

sig:准备发送的信号代码，假如其值为0则没有任何信号送出

进程终止

有8种方式可以终止进程，其中5种为正常终止，它们是

- 在main()中用return()返回
- 调用exit()函数
- 调用_exit()或者是__Exit()函数
- 最后一个线程从其启用例程(线程主函数)用return返回
- 在最后一个线程中调用pthread_exit()返回

异常终止有三种方式，分别是

- 调用abort()函数终止
- 接收到一个信号
- 最后一个线程对取消请求作出响应

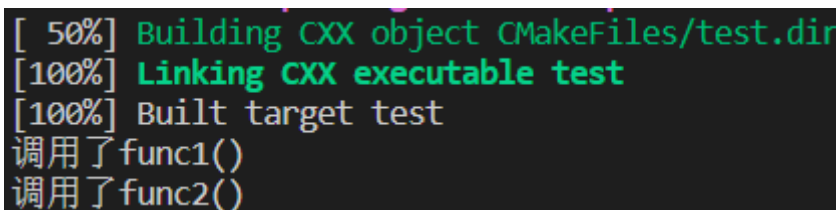
这里是用exit退出

```
#include <iostream>
#include <unistd.h>
#include <signal.h>
using namespace std;

void func2(){
    cout<<"调用了func2()\n";
    exit(0);
}

void func1(){
    cout<<"调用了func1()\n";
    func2();
    cout<<"回到了func1()函数";
}

int main(int argc, char*argv[]){
    func1();
    cout<<"回到了main函数\n";
}
```



A terminal window showing the build process of a C++ program. The output is as follows:

```
[ 50%] Building CXX object CMakeFiles/test.dir
[100%] Linking CXX executable test
[100%] Built target test
调用了func1()
调用了func2()
```

很明显，只打印了两行，说明func2的时候就退出了，不会回到func1更不会回到main,因为exit()是终止线程

一.进程终止的状态

在main函数中，return返回的值即为终止状态，如果没有return语句或者调用exit()，那么该进程的终止状态是0

在Shell中，查看进程终止的状态: echo \$?

正常终止进程的三个函数(exit()和_Exit())是由ISO C说明的，__exit()是由POSIX说明的)

```
void exit(int status)
void _exit(int status)
void _Exit(int status)
```

exit函数中的参数就是退出时候的状态

status也是进程终止的状态

如果进程被异常终止，终止状态为非0

这个status一般用于对服务程序的调度、日志和监控

二.资源释放的问题

return表示函数返回，会调用局部对象的析构函数，main函数中的return还会调用全局对象的析构函数

exit()表示终止进程，不会调用局部对象的析构函数，只会调用全局对象的析构函数

exit会执行清理工作然后退出，_exit()和另外一个会直接退出的，不会执行清理工作

析构函数不被调用是很恐怖的事情，很容易造成开发上的内存泄漏问题

三.进程的终止函数

进程可以用atexit()函数登记终止函数，最多32个，这些函数将由exit(0)自动调用

函数原型: int atexit(void(*function)(void));

exit()调用终止函数的顺序与登记的时候相反。

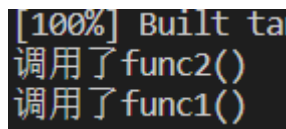
说实在的进程终止函数的作用其实就是进程退出前的收尾工作

```
#include <iostream>
#include <unistd.h>
#include <signal.h>
using namespace std;

void func2(){
    cout<<"调用了func2()\n";
}

void func1(){
    cout<<"调用了func1()\n";
}

int main(int argc, char*argv[]){
    atexit(func1); //登记第一个进程终止函数
    atexit(func2); //登记第二个进程终止函数
    return 0;
}
```



```
[100%] Built target ...
调用了func2()
调用了func1()
```

调用可执行程序

Linux提供了system()函数和exec函数族，在C++程序中，可以执行其他的程序，如二进制文件，操作系统命令或者是Shell脚本(无人车的语音部分采用的就是这个system函数)

一.system()函数

system()函数提供了一种简单的执行程序的办法，需要把执行的命令和参数用一个字符串传给system()函数就行了

函数的声明:

int system(const char*string)

system()函数的返回值比较麻烦

- 如果程序执行失败，system()函数返回非0
- 如果执行程序成功，并且被执行的程序的终止状态是0，system()函数返回0
- 如果执行程序成功，并且被执行的程序的终止状态不是0，system()函数返回非0

我仔细阅读了System函数的源码，它就是从父进程fork一个子进程出来执行任务，子进程会被函数里面的任务进程取代掉(这个过程正是用execl函数实现的)，但是子进程被取代不影响父进程的执行以及正常返回

二.exec函数族

exec函数族提供了另一种在进程中调用程序(二进制文件或者是shell脚本)的方法

exec函数族的声明如下

```
int execl(const char*path,const char*arg,...)
int execlp(const char*file,const char*arg,...)
int execl(const char*path,const char*arg,...,char*const envp[])
int execv(const char*path,char*const argv[])
int execvp(const char*file,char*const argv[])
int execvpe(const char*file,char*const argv[],char*const envp[])
```

注意:

- 如果执行失败直接返回-1，失败原因存于errno中
- 新进程的进程编号与原进程相同，但是，新进程取代了原进程的代码段、数据段和堆栈
- 如果执行成功则函数不会返回，当在主程序中成功调用exec后，被调用的程序将取代调用者程序，也就是说，exec函数之后的代码都不会执行
- 在实际开发中，最常用的是execl()和execv()，其它的极少使用

使用system函数执行程序会回来，如果用exec执行程序，新进程取代了原进程的代码段、数据段和堆栈，所以回不来了。

创建进程

一.Linux的0、1、2号进程

整个linux系统的全部进程是一个树形结构

0号进程(系统进程)是所有进程的祖先，它创建了1号和2号进程

1号进程(systemd)负责执行内核的初始化工作和进行系统配置

2号进程(kthreadd)负责所有内核线程的调度和管理

用pstree命令可以查看进程树

pstree -p 进程编号

二.进程标识

每个进程都有一个非负整数标识的唯一的进程ID。虽然是唯一的，但是进程ID可以复用。当一个进程终止后，其进程ID就成了复用的候选者。Linux采用延迟复用算法，让新建进程的ID不同于最近终止的进程所使用的ID。这样防止了新进程被误认为是使用了同一个ID的某个已经终止的线程。

```
pid_t getpid(void) //获取当前进程的ID
pid_t getppid(void) //获取父进程的ID
```

三.fork()函数

一个现有的进程可以调用fork()函数创建一个新的进程

```
pid_t fork(void)
```

由fork()创建的新进程被称为子进程

fork()函数被调用一次，但是返回两次。两次返回的区别是子进程的返回值是0，而父进程的返回值则是新建子进程的进程ID

子进程和父进程继续执行fork()之后的代码，子进程是父进程的副本

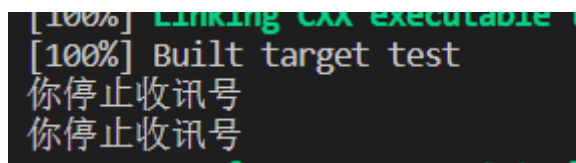
子进程获得了父进程的数据空间，堆栈的副本（注意：子进程只是拥有一个副本，不是共享）

fork()之后，父进程和子进程的执行顺序是不确定的

```
#include <iostream>
#include <unistd.h>
#include <string>

using namespace std;

int main(int argc, char*argv[]){
    string str="你停止收讯号";
    fork();
    cout<<str<<endl;
}
```



```
[100%] Linking CXX executable test
[100%] Built target test
你停止收讯号
你停止收讯号
```

四.fork()的两种用法

- 父进程希望复制自己，然后父进程和子进程分别执行不同的代码。这种用法在网络服务程序中很常见，父进程等待客户端的连接请求，当请求到达的时候，父进程调用fork()，让子进程处理一些请求，然后父进程继续等待下一个连接请求
- 进程要执行另一个程序。这种用法在Shell中很常见，子进程从fork()返回后立即调用exec

五.共享文件

fork()的一个特性是父进程中打开的文件描述符都会被复制到子进程中，父进程和子进程共享同一个文件偏移量。

如果父进程和子进程写同一描述符指向的文件，但是没有任何形式的同步，那么他们的输出可能会相互混合，解决这个的办法就是采用进程同步

六.vfork()函数

vfork()函数调用和返回值与fork()相同，但是二者语义不同

vfork()函数用于创建一个新进程，这个新进程的目的是exec一个新程序，它不复制父进程的地址空间，因为子进程会立即调用exec,于是也就不会使用父进程的地址空间。如果子进程使用了父进程的地址空间，可能会带来未知的结果

vfork()和fork()的另一个区别是:vfork()保证子进程先运行，在子进程调用exec或者是exit()后父进程才恢复运行

僵尸进程

如果父进程比子进程先退出，那么子进程会被1号进程托管(这是一种让程序在后台运行的办法)

如果子进程比父进程先退出，而父进程没有处理子进程退出的信息，那么，子进程将成为僵尸进程。

输入top命令，就可以查看到有多少个zombie进程了

僵尸进程的危害

僵尸进程有很大的危害。内核为每个子进程保留了一个数据结构，里面包含进程编号、终止状态、使用CPU时间等等。父进程如果处理了子进程退出的信息，内核就会释放这个数据结构。父进程如果没有处理子进程退出的信息，内核就不会释放这个数据结构，子进程的进程编号就会被一直占用。系统可用的进程编号是有限的，如果出现大量的僵尸进程，将会因为没有可以使用的进程编号，导致系统无法产生新的进程

僵尸进程的避免

- 子进程退出的时候，内核会向父进程发SIGCHLD信号，如果父进程用signal(SIGCHLD,SIG_IGN)通知内核，表示自己对子进程的退出不感兴趣，那么子进程退出之后会立即释放数据结构
- 父进程通过wait()/waitpid()等函数等待子进程结束，在子进程退出之前，父进程将阻塞等待。

```
pid_t wait(int*stat_loc)
pid_t waitpid(pid_t pid,int*stat_loc,int options)
pid_t wait3(int *status,int optionsj,struct rusage*rusage)
pid_t wait4(pid_t pid,int*status,int options,struct rusage*rusage)
```

返回值是子进程的编号

stat_loc是子进程终止的信息

如果是正常终止，宏WIFEXITED(stat_loc)返回真，宏WEXITSTATUS(stat_loc)可获取终止状态

如果是异常终止，宏WIFEXITED(stat_loc)可获取终止进程的信号

- 如果父进程很忙，可以捕获SIGCHLD信号，在信号处理函数中调用wait()/waitpid()

