UNIX 上的 C++程序设计守则 (4)

铁则 4：请不要做线程的异步撤消的设计

- *线程的异步撤销是指： 某个线程的执行**立刻**被其他线程给强制终止了*
- *请不要单单为了让"设计更简单"或者"看起了更简单"而使用线程的异步撤消*

咋一看还是挺简单的。但是搞不好可能会引起各种各样的问题。请不要在不能把握问题的实质就做出使用线程的异步撤消的设计！

在 pthread 的规格说明中，允许一个线程可以强制中断某个线程的执行。这就是所说的异步撤消。

线程的撤消有下面的两种方式。

- 方式 1： 异步撤消(PTHREAD_CANCEL_ASYNCHRONOUS)
  - 撤销动作是马上进行的
- 方式 2： 延迟撤销(PTHREAD_CANCEL_DEFERRED)
  - 撤消动作，是让线程的处理一直被延迟到撤消点才会去执行

还有，到底是用哪种撤消方式，不是撤消侧，而是被撤销侧能够决定的*1。另外，在被撤销侧也能够选择完全禁止撤消的这种方式 *2。

会造成什么问题呢

那么，让我看看乱用线程的异步撤消会引起什么问题呢。看过准则 3 的人可能会知道，在下面的脚本里，被撤销线程以外的任意一个线程会被死锁。

1. *线程 1 中调用 malloc 函数正在做内存分配的过程中，线程 2 异步撤消了线程 1 的处理*
2. *线程 1 马上被撤销，但是 malloc 函数中的互斥锁就没有线程去解除了*
3. *后面的任意一个线程如果再次调用 malloc 函数的话就会马上导致该线程死锁*

在这个例子中使用了 malloc 函数，但是其他的危险函数还有很多。

反之，即使做了异步撤消也没有问题的函数也有少数存在的、我们把它们叫做「async-cancel safe 函数」或者「异步撤消安全函数」。在一些商用 UNIX*3 中、OS 提供的 api 函数的文档说明中有 async-cancel safety 的记载、但是在 Linux(glibc)里就很遗憾，几乎没有相关的说明。

在这儿，参看规格(SUSv3)的话，会发现，、描述异步撤消安全的函数只有 3 个。

1. pthread_cancel
2. pthread_setcancelstate
3. pthread_setcanceltype

而且、里面还有"No other functions are required to be async-cancel-safe"这样的记载。因此，Linux 的场合，如果在文档里没有记载成 async-cancel safety 的函数，我们还是把它假定成不安全的函数为好!

如何避免这些问题呢

在多线程编程中为了安全的使用异步撤消处理、有没有回避死锁的方法呢？我们试着想了几个。他们与准则 3 里的线程+fork 的场合的回避策很相似。

回避方法 1：被撤销线程中，只能使用异步撤消安全函数

首先，被撤销线程中，只能使用异步撤消安全函数。但是这个方法

- 在规格说明中只有 3 个异步撤消安全的函数
- 这些以外的函数是不是异步撤消安全(商用 UNIX)、因为没有说明文档我们不清楚(Linux)

中有以上的两点，所以这个回避方法几乎不现实。

回避方法 2：被撤销线程中，在做非异步撤消安全处理的过程中，再把撤消方式设置成「延迟」或者是「禁止」

第二个是，被撤销线程在做非异步撤消安全处理的过程中，把撤消方式再设定成「延迟」或者「禁止」。对于这个方法

- 就像方法 1 写的那样、要把我那个函数是异步撤消安全的一时还是挺麻烦的
- 在任意的场所并不能保证撤消动作会被马上执行
    - 例如，再设定成「延迟」后的一段时间内如果撤消发生时、某个正在阻塞的 I/O 函数是否能够被解除阻塞还是挺微妙的
    - 如果设定成撤消禁止的话，则撤消会被屏蔽掉

有上面样的问题、会导致「一精心设计撤消方式的替换，从一开始就使用延迟撤消还不够好」这样的结果。所以这几乎是不好的一个回避策。

回避方法 3： 使用 pthread_cleanup_push 函数，登录异步撤消时的线程数据清除的回调函数

第三种则是，用 pthread_cleanup_push 函数、登录一个在异步撤消发生时的数据清除的回调函数。这和在准则 3 中介绍的 pthread_atfork 函数有点儿类似。用这个函数登录的回调函数来清除线程的数据和锁，就可以回避死锁了。

...但是，pthread_cleanup_push 函数登录的回调函数，在「延迟撤消」的场合是不能被调用的。因此、这个回避方法对于异步撤消没有什么大的作用。

回避方法 4 ： 不要执行异步撤消处理

最后是、不要执行异步撤消处理。反而代之的是、

- 设计成不依赖使用异步撤消那样的处理

或者

- 不得不使用线程撤消的话、不做异步撤消而作延迟撤消的处理

这是比较实际的做法，是我们值得推荐的。

*1：pthread_setcanceltype 函数

*2：pthread_setcancelstate 函数

*3：Solaris 和 HP-UX 等

准则 5：尽可能避免线程中做延迟撤销的处理

- *线程的异步撤消是指：一个线程发出中断其他线程的处理的一个动作*
- *延迟撤消因为是规格自由度比较高、所以根据 OS 和 C 库函数的版本它也有各式各样的动作*
    - *要想在不同的环境下都能稳定的动作的话，就必须要详细调查运行环境和，对 C 库函数进行抽象化，做必要的条件编译*
    - *在 C++ 中、「撤消发生时的对象释放」的实现不具有可移植性*
- *线程撤销要慎重使用。在 C++ 里不要使用*

说明:

在前面我们已经讲过，线程的撤消分为「异步」「延迟」这两种类型、并且「异步撤消」也是非常容易引起各种复杂问题的元凶。

那么，现在要在程序中除掉「延迟撤消」。延迟撤消虽然不会像异步撤消那样会引起各种各样的问题、但是、注意事项还是有很多的。只有把下面的这些注意事项全部都把握之后才能放心使用。

注意事项 1：要好好把握撤消点

和异步撤消不一样的是、撤消处理一直会被延迟到在代码上明示出来的撤消点之后才会被执行。如果编写了一个具有延迟撤消可能的代码、代码中的那条语句是撤消点、必须要正确的把握。

首先、调用过 pthread_testcancel 函数的地方就变成撤消点了。当然这个函数是、仅仅为了「变成延迟撤消」的目的而设置出来的函数。除此之外、某些标准库函数被调用后会不会变成撤消点是在规格(SUSv3)中决定的。请参照规格说明、有下面的函数一览。

下面的函数**是**撤消点

```
accept, aio_suspend, clock_nanosleep, close, connect,
creat, fcntl, fdatasync,
fsync, getmsg, getpmsg, lockf, mq_receive, mq_send,
mq_timedreceive,
mq_timedsend, msgrcv, msgsnd, msync, nanosleep, open,
pause, poll, pread,
pselect, pthread_cond_timedwait, pthread_cond_wait,
pthread_join,
pthread_testcancel, putmsg, putpmsg, pwrite, read,
readv, recv, recvfrom,
```

（略）

下面的函数**不是**撤消点

```
access, asctime, asctime_r, catclose, catgets, catopen,
closedir, closelog,
ctermid, ctime, ctime_r, dbm_close, dbm_delete,
dbm_fetch, dbm_nextkey, dbm_open,
dbm_store, dlclose, dlopen, endgrent, endhostent,
endnetent, endprotoent,
endpwent, endservent, endutxent, fclose, fcntl, fflush,
fgetc, fgetpos, fgets,
```

```
fgetwc, fgetws, fmtmsg, fopen, fpathconf, fprintf,
fputc, fputs, fputwc, fputws,
(略)
```

看到这些我想已经明白了、但是在规格中也说明了「能否成为撤消点跟具体的实现相关的函数」也是多数存在的。原因是、为了可移植性、保证「在一定的时间内让线程的延迟撤消完成」是很困难的事情*1。做的不好的话、只要稍微一提升 OS 的版本就可能让做出来的程序产品不能动作。

即使是这样那还想要使用延迟撤消吗？

注意事项 2：实现要知道 cleanup 函数的必要性

可能被延迟撤销的线程在运行的过程中，要申请资源的场合，一定要考虑到以下的几点，否则就会编制出含有资源丢失和死锁的软件产品。

例如编写的下面的函数就不能被安全的延迟撤销掉。

```
void* cancel_unsafe(void*) {
    static pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_lock(&mutex);                    // 此处
不是撤消点
    struct timespec ts = {3, 0}; nanosleep(&ts, 0); //
经常是撤消点
    pthread_mutex_unlock(&mutex);                  // 此处
不是撤消点
    return 0;
}
int main(void) {
    pthread_t t;
```

```
    // pthread_create 后马发上收到一个有效的延迟撤消的要求
    pthread_create(&t, 0, cancel_unsafe, 0);
    pthread_cancel(t);
    pthread_join(t, 0);
    cancel_unsafe(0); // 发生死锁！
    return 0;
}
```

在上面的样例代码中、nanosleep 执行的过程中经常会触发延迟撤销的最终动作，但是这个时候的 mutex 锁还处于被锁定的状态。而且、线程一被延迟撤消的话就意味着没有人去释放掉这个互斥锁了*2。因此、在下面的 main 函数中调用同样的 cancel_unsafe 函数时就会引起死锁了。

为了回避这个问题、利用 pthread_cleanup_push 函数在撤消时释放掉互斥锁的话就 OK 了，也就不会死锁了。

```
// 新增清除函数
void cleanup(void* mutex) {
    pthread_mutex_unlock((pthread_mutex_t*)mutex);
}

// 粗体字部分是新增的语句
void* cancel_unsafe(void*) {
    static pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
    pthread_cleanup_push(cleanup, &mutex);
    pthread_mutex_lock(&mutex);
    struct timespec ts = {3, 0}; nanosleep(&ts, 0);
    pthread_mutex_unlock(&mutex);
    pthread_cleanup_pop(0);
    return 0;
}
```

注意事项 3：实现要清楚延迟撤消和 C++之间的兼容度

使用 C 语言的场合，利用上面的 pthread_cleanup_push/pop 函数就能安全地执行延迟撤消的动作，但是在 C++语言的场合就会出现其他的问题。**C++与延迟撤消之间的兼容度是非常差的**。具体的表现有以下两个问题:

1. 执行延迟撤消的时候，内存栈上的对象的析构函数会不会被调用跟具体的开发环境有关系
    - GCC3 版本就不会调用。
    - Solaris 和 Tru64 UNIX 下的原生编译器的场合，就调用析构函数(好像)
2. pthread_cleanup_push/pop 函数和 C++的异常处理机制之间有着怎样的相互影响也能具体环境有关

不调用析构函数，或者在抛出异常的时候不能做 cleanup 处理，经常是发生内存泄漏，资源丢失，程序崩溃，死锁等现象的原因。令人意外的是对于这个深层次的问题，就连 Boost C++库都束手无策。

[Q] Why isn't thread cancellation or termination provided?

[A] There's a valid need for thread termination, so at some point Boost.Threads probably will include it, but only after we can find a truly safe (and portable) mechanism for this concept.

先必须确保对象的自由存储，而后全都让 cleanup 函数去释放对象的方法也有，但是这次是牺牲了异常安全性。
（原文没有看明白：オブジェクトを必ずフリーストア上に確保し、解体を全て、クリーンナップハンドラに行わせる手もありますが、今度は例外安全性が犠牲になるでしょう。）

应该说的是，在使用 C++的工程里不对线程进行延迟撤消处理还是比较实际的。

*1：好的问题是 gethostbyname()函数

*2：异步撤消跟 malloc 函数的例子很相似