IBM

developerWorks 中国 技术主题 Linux 文档库

Linux 的多线程编程的高效开发经验

本文中我们针对 Linux 上多线程编程的主要特性总结出 5 条经验，用以改善 Linux 多线程编程的习惯和避免其中的开发陷阱。在本文中，我们穿插一些 Windows 的编程用例用以对比 Linux 特性，以加深读者印象。

杨奕是 IBM 中国系统与技术实验室的软件工程师。他在上海交通大学电子工程系获得了硕士学位。他的工作主要是在各种操作系统平台上开发基与虚拟化技术的管理软件。

贺皓是 IBM 中国系统与科技开发中心的软件工程师。他在复旦大学获得了计算机科学专业的学士与硕士学位。曾参加 IBM DS(DS 系列存储设备) Agent 的开发工作，他目前在 IBM SVC Agent 开发小组从事研发工作。你可以通过以下地址联系他：haohe@cn.ibm.com 。

张俊伟是 IBM 中国系统与技术部的软件工程师， 他于 2005 年 3 月加入了 IBM，曾参加 IBM Storage Configuration Manager 项目的开发工作，他目前在 IBM DS (DS 系列存储设备) Agent 项目小组从事研发工作。

2009 年 4 月 23 日

背景

Linux 平台上的多线程程序开发相对应其他平台（比如 Windows）的多线程 API 有一些细微和隐晦的差别。不注意这些 Linux 上的一些开发陷阱，常常会导致程序问题不穷，死锁不断。本文中我们从 5 个方面总结出 Linux 多线程编程上的问题，并分别引出相关改善的开发经验，用以避免这些的陷阱。我们希望这些经验可以帮助读者们能更好更快的熟悉 Linux 平台的多线程编程。

我们假设读者都已经很熟悉 Linux 平台上基本的线程编程的 Pthread 库 API 。其他的第三方用以线程编程的库，如 boost，将不会在本文中提及。本文中主要涉及的题材包括线程开发中的线程管理，互斥变量，条件变量等。进程概念将不会在本文中涉及。

Linux 上线程开发 API 的概要介绍

多线程开发在 Linux 平台上已经有成熟的 Pthread 库支持。其涉及的多线程开发的最基本概念主要包含三点：线程，互斥锁，条件。其中，线程操作又分线程的创建，退出，等待 3 种。互斥锁则包括 4 种操作，分别是创建，销毁，加锁和解锁。条件操作有 5 种操作：创建，销毁，触发，广播和等待。其他的一些线程扩展概念，如信号灯等，都可以通过上面的三个基本元素的基本操作封装出来。

线程，互斥锁，条件在 Linux 平台上对应的 API 可以用表 1 归纳。为了方便熟悉 Windows 线程编程的读者熟悉 Linux 多线程开发的 API，我们在表中同时也列出 Windows SDK 库中所对应的 API 名称。

表 1. 线程函数列表

对象	操作	Linux Pthread API	Windows SDK 库对应 API
线程	创建	pthread_create	CreateThread
	退出	pthread_exit	ThreadExit
	等待	pthread_join	WaitForSingleObject
互斥锁	创建	pthread_mutex_init	CreateMutex
	销毁	pthread_mutex_destroy	CloseHandle
	加锁	pthread_mutex_lock	WaitForSingleObject
	解锁	pthread_mutex_unlock	ReleaseMutex
条件	创建	pthread_cond_init	CreateEvent
	销毁	pthread_cond_destroy	CloseHandle

触发	pthread_cond_signal	SetEvent
广播	pthread_cond_broadcast	SetEvent / ResetEvent
等待	pthread_cond_wait / pthread_cond_timedwait	SingleObjectAndWait

多线程开发在 Linux 平台上已经有成熟的 Pthread 库支持。其涉及的多线程开发的最基本概念主要包含三点：线程，互斥锁，条件。其中，线程操作又分线程的创建，退出，等待 3 种。互斥锁则包括 4 种操作，分别是创建，销毁，加锁和解锁。条件操作有 5 种操作：创建，销毁，触发，广播和等待。其他的一些线程扩展概念，如信号灯等，都可以通过上面的三个基本元素的基本操作封装出来。

Linux 线程编程中的 5 条经验

尽量设置 recursive 属性以初始化 Linux 的互斥变量

互斥锁是多线程编程中基本的概念，在开发中被广泛使用。其调用次序层次清晰简单：建锁，加锁，解锁，销毁锁。但是需要注意的是，与诸如 Windows 平台的互斥变量不同，在默认情况下，Linux 下的同一线程无法对同一互斥锁进行递归加速，否则将发生死锁。

所谓递归加锁，就是在同一线程中试图对互斥锁进行两次或两次以上的行为。其场景在 Linux 平台上的代码可由清单 1 所示。

清单 1. Linux 重复对互斥锁加锁实例

```
// 通过默认条件建锁
    pthread_mutex_t *theMutex = new pthread_mutex_t;
    pthread_mutexattr_t attr;
    pthread_mutexattr_init(&attr);
    pthread_mutex_init(theMutex,&attr);
    pthread_mutexattr_destroy(&attr);

    // 递归加锁
    pthread_mutex_lock (theMutex);
    pthread_mutex_lock (theMutex);
    pthread_mutex_unlock (theMutex);
    pthread_mutex_unlock (theMutex);
```

在以上代码场景中，问题将出现在第二次加锁操作。由于在默认情况下，Linux 不允许同一线程递归加锁，因此在第二次加锁操作时线程将出现死锁。

Linux 互斥变量这种奇怪的行为或许对于特定的某些场景会所有用处，但是对于大多数情况下看起来更像是程序的一个 bug 。毕竟，在同一线程中对同一互斥锁进行递归加锁在尤其是二次开发中经常会需要。

这个问题与互斥锁的中的默认 recursive 属性有关。解决问题的方法就是显式地在互斥变量初始化时将设置起 recursive 属性。基于此，以上代码其实稍作修改就可以很好的运行，只需要在初始化锁的时候加设置一个属性。请看清单 2 。

清单 2. 设置互斥锁 recursive 属性实例

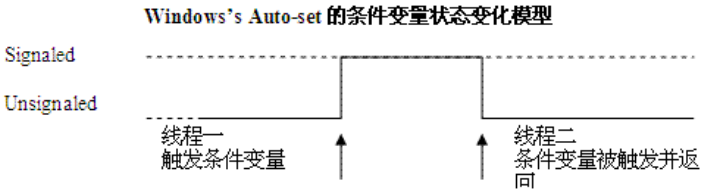
```
pthread_mutexattr_init(&attr);
    // 设置 recursive 属性
    pthread_mutexattr_settype(&attr,PTHREAD_MUTEX_RECURSIVE_NP);
    pthread_mutex_init(theMutex,&attr);
```

因此，建议尽量设置 recursive 属性以初始化 Linux 的互斥锁，这样既可以解决同一线程递归加锁的问题，又可以避免很多情况下死锁的发生。这样做还有一个额外的好处，就是可以让 Windows 和 Linux 下让锁的表现统一。

注意 Linux 平台上触发条件变量的自动复位问题

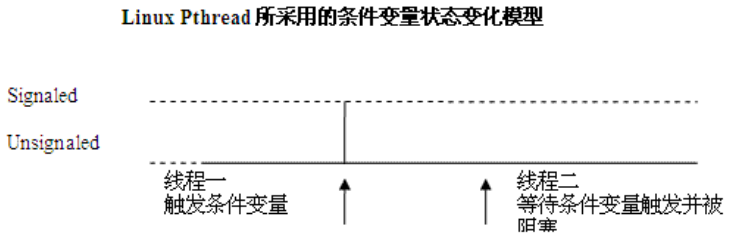
条件变量的置位和复位有两种常用模型：第一种模型是当条件变量置位（signaled）以后，如果当前没有线程在等待，其状态会保持为置位（signaled），直到有等待的线程进入被触发，其状态才会变为复位（unsignaled），这种模型的采用以 Windows 平台上的 Auto-set Event 为代表。其状态变化如图 1 所示：

图 1. Windows 的条件变量状态变化流程



第二种模型则是 Linux 平台的 Pthread 所采用的模型，当条件变量置位（signaled）以后，即使当前没有任何线程在等待，其状态也会恢复为复位（unsignaled）状态。其状态变化如图 2 所示：

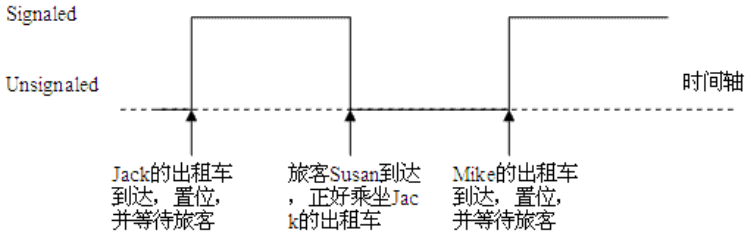
图 2. Linux 的条件变量状态变化流程



具体来说，Linux 平台上 Pthread 下的条件变量状态变化模型是这样工作的：调用 pthread_cond_signal() 释放被条件阻塞的线程时，无论存不存在被阻塞的线程，条件都将被重新复位，下一个被条件阻塞的线程将不受影响。而对于 Windows，当调用 SetEvent 触发 Auto-reset 的 Event 条件时，如果没有被条件阻塞的线程，那么条件将维持在触发状态，直到有新的线程被条件阻塞并被释放为止。

这种差异性对于那些熟悉 Windows 平台上的条件变量状态模型而要开发 Linux 平台上多线程的程序员来说可能会造成意想不到的尴尬结果。试想要实现一个旅客坐出租车的程序：旅客在路边等出租车，调用条件等待。出租车来了，将触发条件，旅客停止等待并上车。一个出租车只能搭载一波乘客，于是我们使用单一触发的条件变量。这个实现逻辑在第一个模型下即使出租车先到，也不会有什么问题，其过程如图 3 所示：

图 3. 采用 Windows 条件变量模型的出租车实例流程



然而如果按照这个思路来在 Linux 上来实现，代码看起来可能是清单 3 这样。

清单 3. Linux 出租车案例代码实例

```
……
 // 提示出租车到达的条件变量
 pthread_cond_t taxiCond;

 // 同步锁
 pthread_mutex_t taxiMutex;

 // 旅客到达等待出租车
 void * traveler_arrive(void * name) {
    cout<< " Traveler: " <<(char *)name<< " needs a taxi now! " <<endl;
    pthread_mutex_lock(&taxiMutex);
    pthread_cond_wait (&taxiCond, &taxtMutex);
    pthread_mutex_unlock (&taxtMutex);
    cout<< " Traveler: " << (char *)name << " now got a taxi! " <<endl;
    pthread_exit( (void *)0 );
 }

 // 出租车到达
 void * taxi_arrive(void *name) {
    cout<< " Taxi " <<(char *)name<< " arrives. " <<endl;
    pthread_cond_signal(&taxtCond);
    pthread_exit( (void *)0 );
 }

 void main() {
    // 初始化
    taxtCond= PTHREAD_COND_INITIALIZER;
    taxtMutex= PTHREAD_MUTEX_INITIALIZER;
```

```
    pthread_t thread;
    pthread_attr_t threadAttr;
    pthread_attr_init(&threadAttr);

    pthread_create(&thread, & threadAttr, taxt_arrive, (void *)( " Jack " ));
    sleep(1);
    pthread_create(&thread, &threadAttr, traveler_arrive, (void *)( " Susan " ));
    sleep(1);
    pthread_create(&thread, &threadAttr, taxi_arrive, (void *)( " Mike " ));
    sleep(1);

    return 0;
}
```

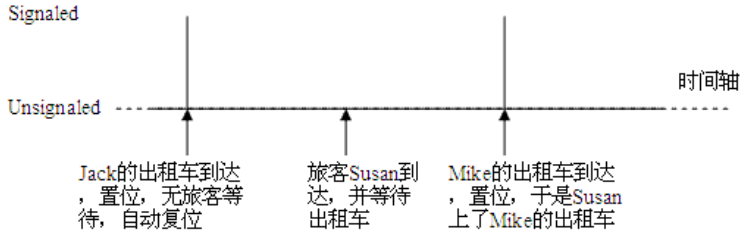
好的，运行一下，看看结果如清单 4 。

清单 4. 程序结果输出

```
Taxi Jack arrives.
    Traveler Susan needs a taxi now!
    Taxi Mike arrives.
    Traveler Susan now got a taxi.
```

其过程如图 4 所示：

图 4. 采用 Linux 条件变量模型的出租车实例流程



通过对比结果，你会发现同样的逻辑，在 Linux 平台上运行的结果却完全是两样。对于在 Windows 平台上的模型一， Jack 开着出租车到了站台，触发条件变量。如果没顾客，条件变量将维持触发状态，也就是说 Jack 停下车在那里等着。直到 Susan 小姐来了站台，执行等待条件来找出租车。 Susan 搭上 Jack 的出租车离开，同时条件变量被自动复位。

但是到了 Linux 平台，问题就来了，Jack 到了站台一看没人，触发的条件变量被直接复位，于是 Jack 排在等待队列里面。来迟一秒的 Susan 小姐到了站台却看不到在那里等待的 Jack，只能等待，直到 Mike 开车赶到，重新触发条件变量，Susan 才上了 Mike 的车。这对于在排队系统前面的 Jack 是不公平的，而问题症结是在于 Linux 平台上条件变量触发的自动复位引起的一个 Bug 。

条件变量在 Linux 平台上的这种模型很难说好坏。但是在实际开发中，我们可以对代码稍加改进就可以避免这种差异的发生。由于这种差异只发生在触发没有被线程等待在条件变量的时刻，因此我们只需要掌握好触发的时机即可。最简单的做法是增加一个计数器记录等待线程的个数，在决定触发条件变量前检查下该变量即可。改进后 Linux 函数如清单 5 所示。

清单 5. Linux 出租车案例代码实例

```
……
 // 提示出租车到达的条件变量
 pthread_cond_t taxiCond;

 // 同步锁
 pthread_mutex_t taxiMutex;

 // 旅客人数，初始为 0
 int travelerCount=0;

 // 旅客到达等待出租车
 void * traveler_arrive(void * name) {
    cout<< " Traveler: " <<(char *)name<< " needs a taxi now! " <<endl;
    pthread_mutex_lock(&taxiMutex);

    // 提示旅客人数增加
    travelerCount++;
    pthread_cond_wait (&taxiCond, &taxiMutex);
    pthread_mutex_unlock (&taxiMutex);
    cout<< " Traveler: " << (char *)name << " now got a taxi! " <<endl;
    pthread_exit( (void *)0 );
 }

 // 出租车到达
 void * taxi_arrive(void *name)
 {
    cout<< " Taxi " <<(char *)name<< " arrives. " <<endl;
```

```
while(true)
{
       pthread_mutex_lock(&taxiMutex);

       // 当发现已经有旅客在等待时，才触发条件变量
       if(travelerCount>0)
       {
           pthread_cond_signal(&taxtCond);
           pthread_mutex_unlock (&taxiMutex);
           break;
       }
       pthread_mutex_unlock (&taxiMutex);
   }

   pthread_exit( (void *)0 );
}
```

因此我们建议在 Linux 平台上要出发条件变量之前要检查是否有等待的线程，只有当有线程在等待时才对条件变量进行触发。

注意条件返回时互斥锁的解锁问题

在 Linux 调用 pthread_cond_wait 进行条件变量等待操作时，我们增加一个互斥变量参数是必要的，这是为了避免线程间的竞争和饥饿情况。但是当条件等待返回时候，需要注意的是一定不要遗漏对互斥变量进行解锁。

Linux 平台上的 pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex) 函数返回时，互斥锁 mutex 将处于锁定状态。因此之后如果需要对临界区数据进行重新访问，则没有必要对 mutex 就行重新加锁。但是，随之而来的问题是，每次条件等待以后需要加入一步手动的解锁操作。正如前文中乘客等待出租车的 Linux 代码如清单 6 所示：

清单 6. 条件变量返回后的解锁实例

```
void * traveler_arrive(void * name) {
    cout<< ” Traveler: ” <<(char *)name<< ” needs a taxi now! ” <<endl;
    pthread_mutex_lock(&taxiMutex);
    pthread_cond_wait (&taxiCond, &taxtMutex);
    pthread_mutex_unlock (&taxtMutex);
    cout<< ” Traveler: ” << (char *)name << ” now got a taxi! ” <<endl;
    pthread_exit( (void *)0 );
 }
```

这一点对于熟悉 Windows 平台多线程开发的开发者来说尤为重要。 Windows 上的 SignalObjectAndWait() 函数是常与 Linux 平台上的 pthread_cond_wait() 函数被看作是跨平台编程时的一对等价函数。但是需要注意的是，两个函数退出时的状态是不一样的。在 Windows 平台上，SignalObjectAndWait(HANDLE a, HANDLE b, ……) 方法在调用结束返回时的状态是 a 和 b 都是置位（signaled）状态，在普遍的使用方法中，a 经常是一个 Mutex 变量，在这种情况下，当返回时，Mutex a 处于解锁状态（signaled），Event b 处于置位状态（signaled）, 因此，对于 Mutex a 而言，我们不需要考虑解锁的问题。而且，在 SignalObjectAndWait() 之后，如果需要对临界区数据进行重新访问，都需要调用 WaitForSingleObject() 重新加锁。这一点刚好与 Linux 下的 pthread_cond_wait() 完全相反。

Linux 对于 Windows 的这一点额外解锁的操作区别很重要，一定得牢记。否则从 Windows 移植到 Linux 上的条件等待操作一旦忘了结束后的解锁操作，程序将肯定会发生死锁。

等待的绝对时间问题

超时是多线程编程中一个常见的概念。例如，当你在 Linux 平台下使用 pthread_cond_timedwait() 时就需要指定超时这个参数，以便这个 API 的调用者最多只被阻塞指定的时间间隔。但是如果你是第一次使用这个 API 时，首先你需要了解的就是这个 API 当中超时参数的特殊性（就如本节标题所提示的那样）。我们首先来看一下这个 API 的定义。 pthread_cond_timedwait() 定义请看清单 7 。

清单 7. pthread_cond_timedwait() 函数定义

```
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
             pthread_mutex_t *restrict mutex,
             const struct timespec *restrict abstime);
```

参数 abstime 在这里用来表示和超时时间相关的一个参数，但是需要注意的是它所表示的是一个绝对时间，而不是一个时间间隔数值，只有当系统的当前时间达到或者超过 abstime 所表示的时间时，才会触发超时事件。这对于拥有 Windows 平台线程开发经验的人来说可能尤为困惑。因为 Windows 平台下所有的

API 等待参数（如 SignalObjectAndWait，等）都是相对时间，

假设我们指定相对的超时时间参数如 dwMilliseconds （单位毫秒）来调用和超时相关的函数，这样就需要将 dwMilliseconds 转化为 Linux 下的绝对时间参数 abstime 使用。常用的转换方法如清单 8 所示：

清单 8. 相对时间到绝对时间转换实例

```
/* get the current time */
    struct timeval now;
    gettimeofday(&now, NULL);

    /* add the offset to get timeout value */
    abstime ->tv_nsec = now.tv_usec * 1000 + (dwMilliseconds % 1000) * 1000000;
    abstime ->tv_sec = now.tv_sec + dwMilliseconds / 1000;
```

Linux 的绝对时间看似简单明了，却是开发中一个非常隐晦的陷阱。而且一旦你忘了时间转换，可以想象，等待你的错误将是多么的令人头疼：如果忘了把相对时间转换成绝对时间，相当于你告诉系统你所等待的超时时间是过去式的 1970 年 1 月 1 号某个时间段，于是操作系统毫不犹豫马上送给你一个 timeout 的返回值，然后你会举着拳头抱怨为什么另外一个同步线程耗时居然如此之久，并一头扎进寻找耗时原因的深渊里。

正确处理 Linux 平台下的线程结束问题

在 Linux 平台下，当处理线程结束时需要注意的一个问题就是如何让一个线程善始善终，让其所占资源得到正确释放。在 Linux 平台默认情况下，虽然各个线程之间是相互独立的，一个线程的终止不会去通知或影响其他的线程。但是已经终止的线程的资源并不会随着线程的终止而得到释放，我们需要调用 pthread_join() 来获得另一个线程的终止状态并且释放该线程所占的资源。 Pthread_join() 函数的定义如清单 9 。

清单 9. pthread_join 函数定义

```
int pthread_join(pthread_t th, void **thread_return);
```

调用该函数的线程将挂起，等待 th 所表示的线程的结束。 thread_return 是指向线程 th 返回值的指针。需要注意的是 th 所表示的线程必须是 joinable 的，即处于非 detached（游离）状态；并且只可以有唯一的一个线程对 th 调用 pthread_join() 。如果 th 处于 detached 状态，那么对 th 的 pthread_join() 调用将返回错误。

如果你压根儿不关心一个线程的结束状态，那么也可以将一个线程设置为 detached 状态，从而来让操作系统在该线程结束时来回收它所占的资源。将一个线程设置为 detached 状态可以通过两种方式来实现。一种是调用 pthread_detach() 函数，可以将线程 th 设置为 detached 状态。其申明如清单 10 。

清单 10. pthread_detach 函数定义

```
int pthread_detach(pthread_t th);
```

另一种方法是在创建线程时就将它设置为 detached 状态，首先初始化一个线程属性变量，然后将其设置为 detached 状态，最后将它作为参数传入线程创建函数 pthread_create()，这样所创建出来的线程就直接处于 detached 状态。方法如清单 11 。

清单 11. 创建 detach 线程代码实例

```
…… ..
    pthread_t       tid;
    pthread_attr_t  attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    pthread_create(&tid, &attr, THREAD_FUNCTION, arg);
```

总之为了在使用 Pthread 时避免线程的资源在线程结束时不能得到正确释放，从而避免产生潜在的内存泄漏问题，在对待线程结束时，要确保该线程处于 detached 状态，否着就需要调用 pthread_join() 函数来对其进行资源回收。

总结与补充

本文以上部分详细介绍了 Linux 的多线程编程的 5 条高效开发经验。另外你也可以考虑尝试其他一些开源类库来进行线程开发。

1. Boost 库

Boost 库来自于由 C++ 标准委员会类库工作组成员发起，致力于为 C++ 开发新的类库的 Boost 组织。虽然该库本身并不是针对多线程而产生，但是发展至今，其已提供了比较全面的多线程编程的 API 支持。Boost 库对于多线程支持的 API 风格上更类似于 Linux 的 Pthread 库，差别在于其将线程，互斥锁，条件等线程开发概念都封装成了 C++ 类，以方便开发调用。 Boost 库目前对跨平台支持的很不错，不仅支持 Windows 和 Linux ，还支持各种商用的 Unix 版本。如果开发者想使用高稳定性的统一线程编程接口减轻跨平台开发的难度， Boost 库将是首选。

2. ACE

ACE 全称是 ADAPTIVE Communication Environment，它是一个免费的，开源的，面向对象的工具框架，用以开发并发访问的软件。由于 ACE 最初是面向网络服务端的编程开发，因此对于线程开发的工具库它也能提供很全面的支持。其支持的平台也很全面，包括 Windows，Linux 和各种版本 Unix 。 ACE 的唯一问题是如果仅仅是用于线程编程，其似乎显得有些过于重量级。而且其较复杂的配置也让其部署对初学者而言并非易事。

参考资料

IBM Developerworks 以下系列文章系列详细介绍了如何对线程程序从 Windows 到 Linux 上进行移植。

文章《Strategies for Implementing POSIX Condition Variables on Win32》很全面的介绍了如何在 Windows 上实现 POSIX condition。

参阅 Boost 库的官方网站

参阅 ACE 库的官方网站