

5.4 怎么避免死锁？

面试过程中，死锁也是高频的考点，因为如果线上环境真多发生了死锁，那真的出大事了。

这次，我们就来系统地聊聊死锁的问题。

- 死锁的概念；
- 模拟死锁问题的产生；
- 利用工具排查死锁问题；
- 避免死锁问题的发生；

死锁的概念

在多线程编程中，我们为了防止多线程竞争共享资源而导致数据错乱，都会在操作共享资源之前加上互斥锁，只有成功获得锁的线程，才能操作共享资源，获取不到锁的线程就只能等待，直到锁被释放。

那么，当两个线程为了保护两个不同的共享资源而使用了两个互斥锁，那么这两个互斥锁应用不当的时候，可能会造成**两个线程都在等待对方释放锁**，在没有外力的作用下，这些线程会一直相互等待，就没办法继续运行，这种情况就是发生了**死锁**。

举个例子，小林拿了小美房间的钥匙，而小林在自己的房间里，小美拿了小林房间的钥匙，而小美也在自己的房间里。如果小林要从自己的房间里出去，必须拿到小美手中的钥匙，但是小美要出去，又必须拿到小林手中的钥匙，这就形成了死锁。

死锁只有**同时满足**以下四个条件才会发生：

- 互斥条件；
- 持有并等待条件；
- 不可剥夺条件；
- 环路等待条件；

互斥条件

互斥条件是指**多个线程不能同时使用同一个资源**。



目录



侧边栏



夜间



技术群



资料



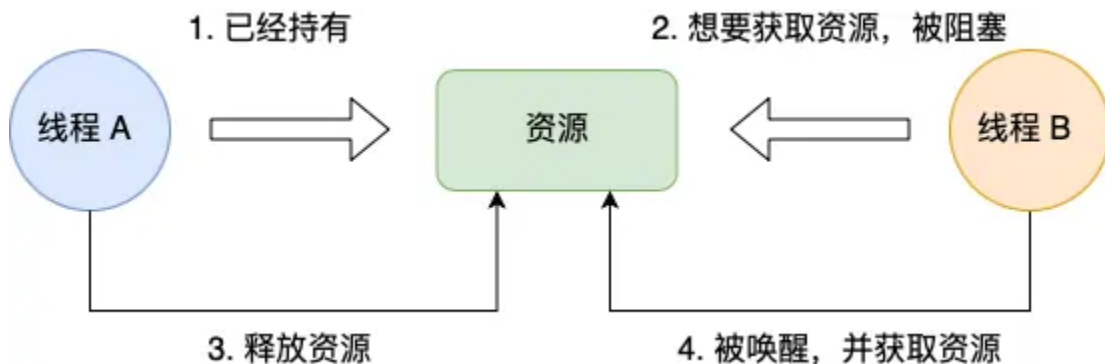
支持我



上一篇

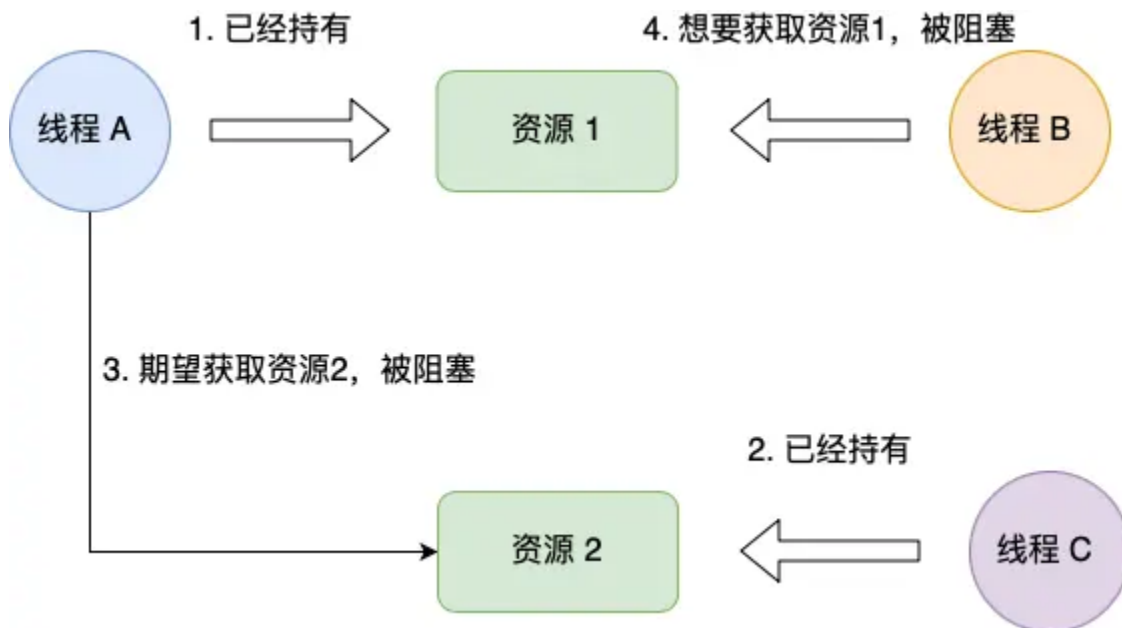


下一篇



持有并等待条件

持有并等待条件是指，当线程 A 已经持有了资源 1，又想申请资源 2，而资源 2 已经被线程 C 持有了，所以线程 A 就会处于等待状态，但是**线程 A 在等待资源 2 的同时并不会释放自己已经持有的资源 1。**



不可剥夺条件

不可剥夺条件是指，当线程已经持有了资源，**在自己使用完之前不能被其他线程获取**，线程 B 如果也想使用此资源，则只能在线程 A 使用完并释放后才能获取。



目录



侧边栏



夜间



技术群



资料



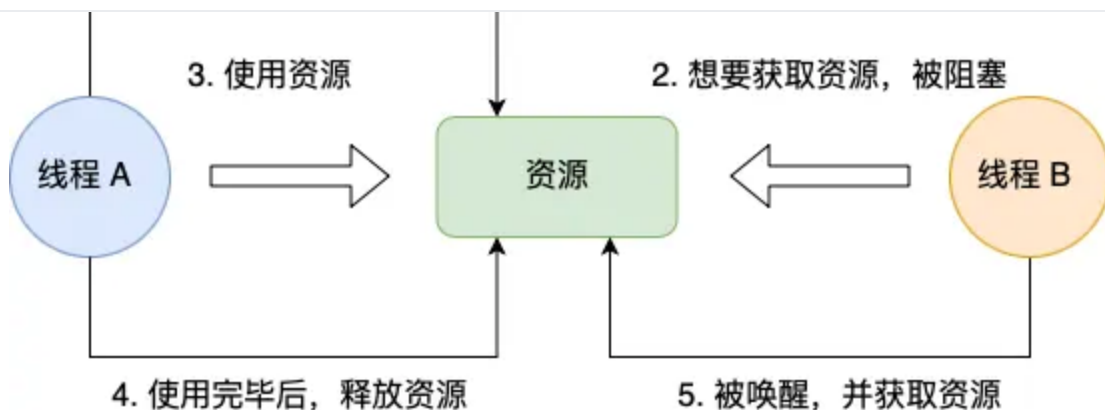
支持我



上一篇



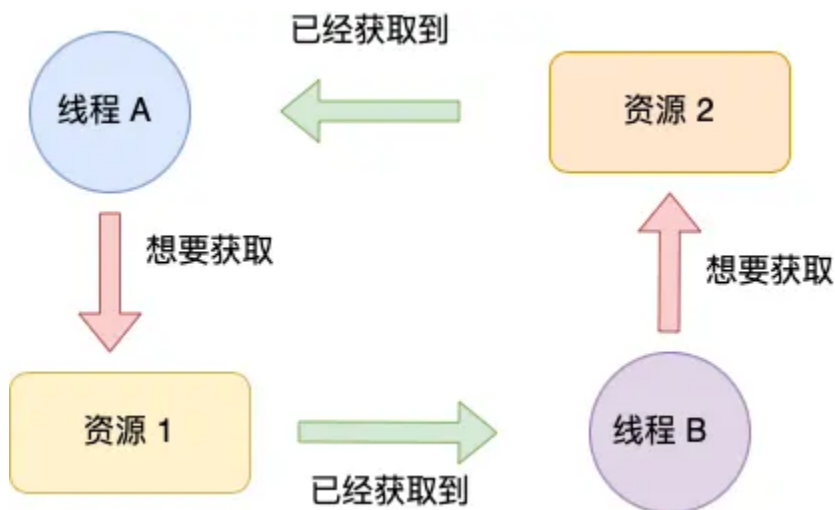
下一篇



环路等待条件

环路等待条件指的是，在死锁发生的时候，**两个线程获取资源的顺序构成了环形链**。

比如，线程 A 已经持有资源 2，而想请求资源 1，线程 B 已经获取了资源 1，而想请求资源 2，这就形成资源请求等待的环形图。



模拟死锁问题的产生

Talk is cheap. Show me the code.

下面，我们用代码来模拟死锁问题的产生。

首先，我们先创建 2 个线程，分别为线程 A 和 线程 B，然后有两个互斥锁，分别是 `mutex_A` 和 `mutex_B`，代码如下：



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



下一篇

```
int main()
{
    pthread_t tidA, tidB;

    //创建两个线程
    pthread_create(&tidA, NULL, threadA_proc, NULL);
    pthread_create(&tidB, NULL, threadB_proc, NULL);

    pthread_join(tidA, NULL);
    pthread_join(tidB, NULL);

    printf("exit\n");

    return 0;
}
```

接下来，我们看下线程 A 函数做了什么。

```
//线程函数 A
void *threadA_proc(void *data)
{
    printf("thread A waiting get ResourceA \n");
    pthread_mutex_lock(&mutex_A);
    printf("thread A got ResourceA \n");

    sleep(1);

    printf("thread A waiting get ResourceB \n");
    pthread_mutex_lock(&mutex_B);
    printf("thread A got ResourceB \n");

    pthread_mutex_unlock(&mutex_B);
    pthread_mutex_unlock(&mutex_A);
    return (void *)0;
}
```

可以看到，线程 A 函数的过程：

- 先获取互斥锁 A，然后睡眠 1 秒；
- 再获取互斥锁 B，然后释放互斥锁 B；

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

```
//线程函数 B
void *threadB_proc(void *data)
{
    printf("thread B waiting get ResourceB \n");
    pthread_mutex_lock(&mutex_B);
    printf("thread B got ResourceB \n");

    sleep(1);

    printf("thread B waiting get ResourceA \n");
    pthread_mutex_lock(&mutex_A);
    printf("thread B got ResourceA \n");

    pthread_mutex_unlock(&mutex_A);
    pthread_mutex_unlock(&mutex_B);
    return (void *)0;
}
```

可以看到，线程 B 函数的过程：

- 先获取互斥锁 B，然后睡眠 1 秒；
- 再获取互斥锁 A，然后释放互斥锁 A；
- 最后释放互斥锁 B；

然后，我们运行这个程序，运行结果如下：

```
thread B waiting get ResourceB
thread B got ResourceB
thread A waiting get ResourceA
thread A got ResourceA
thread B waiting get ResourceA
thread A waiting get ResourceB
// 阻塞中。。。
```

可以看到线程 B 在等待互斥锁 A 的释放，线程 A 在等待互斥锁 B 的释放，双方都在等待对方资源的释放，很明显，产生了死锁问题。

利用工具排查死锁问题



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



下一篇

由于小林的死锁代码例子是 C 写的，在 Linux 下，我们可以使用 `pstack + gdb` 工具来定位死锁问题。

`pstack` 命令可以显示每个线程的栈跟踪信息（函数调用过程），它的使用方式也很简单，只需要 `pstack <pid>` 就可以了。

那么，在定位死锁问题时，我们可以多次执行 `pstack` 命令查看线程的函数调用过程，多次对比结果，确认哪几个线程一直没有变化，且是因为在等待锁，那么大概率是由于死锁问题导致的。

我用 `pstack` 输出了我前面模拟死锁问题的进程的所有线程的情况，我多次执行命令后，其结果都一样，如下：

```
$ pstack 87746
Thread 3 (Thread 0x7f60a610a700 (LWP 87747)):
#0  0x0000003720e0da1d in __lll_lock_wait () from /lib64/libpthread.so.0
#1  0x0000003720e093ca in _L_lock_829 () from /lib64/libpthread.so.0
#2  0x0000003720e09298 in pthread_mutex_lock () from /lib64/libpthread.so.0
#3  0x0000000000400725 in threadA_proc ()
#4  0x0000003720e07893 in start_thread () from /lib64/libpthread.so.0
#5  0x00000037206f4bfd in clone () from /lib64/libc.so.6
Thread 2 (Thread 0x7f60a5709700 (LWP 87748)):
#0  0x0000003720e0da1d in __lll_lock_wait () from /lib64/libpthread.so.0
#1  0x0000003720e093ca in _L_lock_829 () from /lib64/libpthread.so.0
#2  0x0000003720e09298 in pthread_mutex_lock () from /lib64/libpthread.so.0
#3  0x0000000000400792 in threadB_proc ()
#4  0x0000003720e07893 in start_thread () from /lib64/libpthread.so.0
#5  0x00000037206f4bfd in clone () from /lib64/libc.so.6
Thread 1 (Thread 0x7f60a610c700 (LWP 87746)):
#0  0x0000003720e080e5 in pthread_join () from /lib64/libpthread.so.0
#1  0x0000000000400806 in main ()

....
```

```
$ pstack 87746
Thread 3 (Thread 0x7f60a610a700 (LWP 87747)):
#0  0x0000003720e0da1d in __lll_lock_wait () from /lib64/libpthread.so.0
#1  0x0000003720e093ca in _L_lock_829 () from /lib64/libpthread.so.0
#2  0x0000003720e09298 in pthread_mutex_lock () from /lib64/libpthread.so.0
#3  0x0000000000400725 in threadA_proc ()
#4  0x0000003720e07893 in start_thread () from /lib64/libpthread.so.0
#5  0x00000037206f4bfd in clone () from /lib64/libc.so.6
```

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

```
#2 0x0000003720e09298 in pthread_mutex_lock () from /lib64/libpthread.so.0
#3 0x0000000000400792 in threadB_proc ()
#4 0x0000003720e07893 in start_thread () from /lib64/libpthread.so.0
#5 0x00000037206f4bfd in clone () from /lib64/libc.so.6
Thread 1 (Thread 0x7f60a610c700 (LWP 87746)):
#0 0x0000003720e080e5 in pthread_join () from /lib64/libpthread.so.0
#1 0x0000000000400806 in main ()
```

可以看到，Thread 2 和 Thread 3 一直阻塞获取锁 (*pthread_mutex_lock*) 的过程，而且 pstack 多次输出信息都没有变化，那么可能大概率发生了死锁。

但是，还不能够确认这两个线程是在互相等待对方的锁的释放，因为我们看不到它们是等在哪个锁对象，于是我们可以使用 gdb 工具进一步确认。

整个 gdb 调试过程，如下：

```
// gdb 命令
$ gdb -p 87746

// 打印所有的线程信息
(gdb) info thread
  3 Thread 0x7f60a610a700 (LWP 87747) 0x0000003720e0da1d in __lll_lock_wait () from
  2 Thread 0x7f60a5709700 (LWP 87748) 0x0000003720e0da1d in __lll_lock_wait () from
* 1 Thread 0x7f60a610c700 (LWP 87746) 0x0000003720e080e5 in pthread_join () from /
//最左边的 * 表示 gdb 锁定的线程，切换到第二个线程去查看

// 切换到第2个线程
(gdb) thread 2
[Switching to thread 2 (Thread 0x7f60a5709700 (LWP 87748))]#0 0x0000003720e0da1d in

// bt 可以打印函数堆栈，却无法看到函数参数，跟 pstack 命令一样
(gdb) bt
#0 0x0000003720e0da1d in __lll_lock_wait () from /lib64/libpthread.so.0
#1 0x0000003720e093ca in _L_lock_829 () from /lib64/libpthread.so.0
#2 0x0000003720e09298 in pthread_mutex_lock () from /lib64/libpthread.so.0
#3 0x0000000000400792 in threadB_proc (data=0x0) at dead_lock.c:25
#4 0x0000003720e07893 in start_thread () from /lib64/libpthread.so.0
#5 0x00000037206f4bfd in clone () from /lib64/libc.so.6

// 打印第三帧信息，每次函数调用都会有压栈的过程，而 frame 则记录栈中的帧信息
(gdb) frame 3
#3 0x0000000000400792 in threadB_proc (data=0x0) at dead_lock.c:25
```

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

```
// 打印mutex_A的值 , __owner表示gdb中标示线程的值, 即LWP
(gdb) p mutex_A
$1 = {__data = {__lock = 2, __count = 0, __owner = 87747, __nusers = 1, __kind = 0,
    __size = "\002\000\000\000\000\000\000\000\303V\001\000\001", '\000' <repeats 26
    ...

// 打印mutex_B的值 , __owner表示gdb中标示线程的值, 即LWP
(gdb) p mutex_B
$2 = {__data = {__lock = 2, __count = 0, __owner = 87748, __nusers = 1, __kind = 0,
    __size = "\002\000\000\000\000\000\000\000\304V\001\000\001", '\000' <repeats 26
    ...
```

我来解释下, 上面的调试过程:

1. 通过 `info thread` 打印了所有的线程信息, 可以看到有 3 个线程, 一个是主线程 (LWP 87746), 另外两个都是我们自己创建的线程 (LWP 87747 和 87748);
2. 通过 `thread 2`, 将切换到第 2 个线程 (LWP 87748);
3. 通过 `bt`, 打印线程的调用栈信息, 可以看到有 `threadB_proc` 函数, 说明这个是线程 B 函数, 也就说 LWP 87748 是线程 B;
4. 通过 `frame 3`, 打印调用栈中的第三个帧的信息, 可以看到线程 B 函数, 在获取互斥锁 A 的时候阻塞了;
5. 通过 `p mutex_A`, 打印互斥锁 A 对象信息, 可以看到它被 LWP 为 87747 (线程 A) 的线程持有着;
6. 通过 `p mutex_B`, 打印互斥锁 B 对象信息, 可以看到他被 LWP 为 87748 (线程 B) 的线程持有着;

因为线程 B 在等待线程 A 所持有的 `mutex_A`, 而同时线程 A 又在等待线程 B 所拥有的 `mutex_B`, 所以可以断定该程序发生了死锁。

避免死锁问题的发生

前面我们提到, 产生死锁的四个必要条件是: 互斥条件、持有并等待条件、不可剥夺条件、环路等待条件。

那么避免死锁问题就只需要破坏其中一个条件就可以, 最常见的并且可行的就是**使用资源有序分配法, 来破坏环路等待条件**。

那什么是资源有序分配法呢?

线程 A 和 线程 B 获取资源的顺序要一样, 当线程 A 是先尝试获取资源 A, 然后尝试获取资源 B 的时候, 线程 B 同样也是先尝试获取资源 A, 然后尝试获取资源 B。也就是说, 线程 A



目录



侧边栏



夜间



技术群



资料



支持我



上一篇

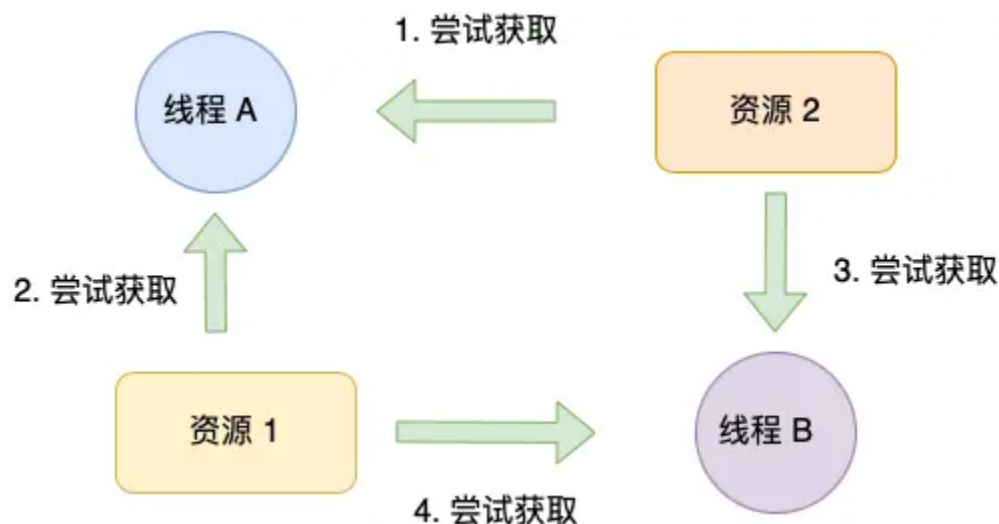


下一篇

码。

我们先要清楚线程 A 获取资源的顺序，它是先获取互斥锁 A，然后获取互斥锁 B。

所以我们只需将线程 B 改成以相同顺序的获取资源，就可以打破死锁了。



线程 B 函数改进后的代码如下：

```
//线程 B 函数，同线程 A 一样，先获取互斥锁 A，然后获取互斥锁 B
void *threadB_proc(void *data)
{
    printf("thread B waiting get ResourceA \n");
    pthread_mutex_lock(&mutex_A);
    printf("thread B got ResourceA \n");

    sleep(1);

    printf("thread B waiting get ResourceB \n");
    pthread_mutex_lock(&mutex_B);
    printf("thread B got ResourceB \n");

    pthread_mutex_unlock(&mutex_B);
    pthread_mutex_unlock(&mutex_A);
    return (void *)0;
}
```

执行结果如下，可以看，没有发生死锁。



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



下一篇

```
thread A waiting get ResourceA
thread B waiting get ResourceB
thread B got ResourceB
thread A got ResourceA
thread A waiting get ResourceB
thread A got ResourceB
exit
```

总结

简单来说，死锁问题的产生是由两个或者以上线程并行执行的时候，争夺资源而互相等待造成的。

死锁只有同时满足互斥、持有并等待、不可剥夺、环路等待这四个条件的时候才会发生。

所以要避免死锁问题，就是要破坏其中一个条件即可，最常用的方法就是使用资源有序分配法来破坏环路等待条件。

关注作者

哈喽，我是小林，就爱图解计算机基础，如果觉得文章对你有帮助，欢迎微信搜索「小林coding」，关注后，回复「网络」再送你图解网络 PDF



扫一扫，关注「小林coding」公众号

图解计算机基础
认准**小林coding**

每一张图都包含小林的认真
只为帮助大家能更好的理解

- ① 关注公众号回复「**图解**」
获取图解系列 PDF
- ② 关注公众号回复「**加群**」
拉你进百人技术交流群



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



下一篇