Linux内核态抢占机制分析（转）

bbsmax 2021-05-21 00:53:09 **阅读数:102** **评论数:0** **点赞数:0** **收藏数:0**

标签： linux 分析 内核 机制 抢占

Linux内核态抢占机制分析 http://blog.sina.com.cn/s/blog_502c8cc401012pxj.html

摘 要】本文首先介绍非抢占式内核(Non-Preemptive Kernel)和可抢占式内核(Preemptive Kernel)的区别。接着分析Linux下有两种抢占：用户态抢占(User Preemption)、内核态抢占(Kernel Preemption)。然后分析了在内核态下：如何判断能否抢占内核(什么是可抢占的条件)；何时触发重新调度(何时设置可抢占条件)；抢占发生的时机 (何时检查可抢占的条件)；什么时候不能抢占内核。最后分析了2.6kernel中如何支持抢占内核。

【关键字】内核态抢占 用户态抢占 中断 实时性 自旋锁 linux kernel schedule preemption reentrant

1.非抢占式和可抢占式内核的区别

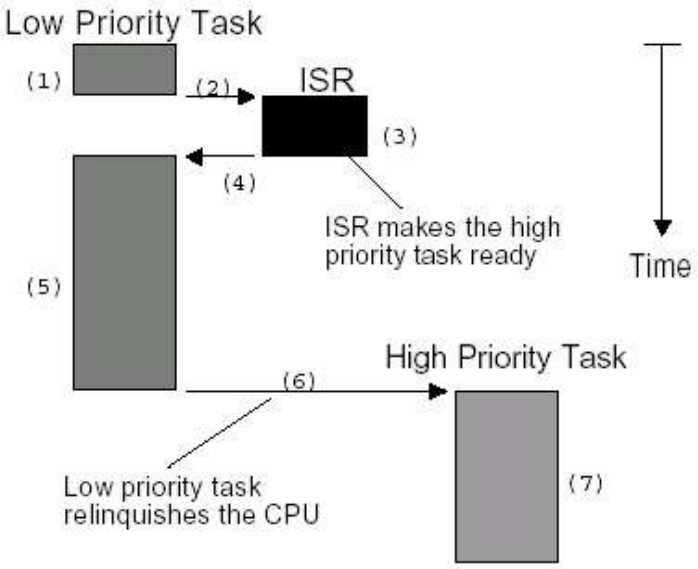
为了简化问题，我使用嵌入式实时系统uC/OS作为例子。首先要指出的是，uC/OS只有内核态，没有用户态，这和Linux不一样。

多 任务系统中，内核负责管理各个任务，或者说为每个任务分配CPU时间，并且负责任务之间的通讯。内核提供的基本服务是任务切换。调度 （Scheduler）,英文还有一词叫dispatcher，也是调度的意思。这是内核的主要职责之一，就是要决定该轮到哪个任务运行了。多数实时内核 是基于优先级调度法的。每个任务根据其重要程度的不同被赋予一定的优先级。基于优先级的调度法指，CPU总是让处在就绪态的优先级最高的任务先运行。然 而，究竟何时让高优先级任务掌握CPU的使用权，有两种不同的情况，这要看用的是什么类型的内核，是不可剥夺型的还是可剥夺型内核。

非抢占式内核

非 抢占式内核是由任务主动放弃CPU的使用权。非抢占式调度法也称作合作型多任务，各个任务彼此合作共享一个CPU。异步事件还是由中断服务来处理。中断服 务可以使一个高优先级的任务由挂起状态变为就绪状态。但中断服务以后控制权还是回到原来被中断了的那个任务，直到该任务主动放弃CPU的使用权时，那个高 优先级的任务才能获得CPU的使用权。非抢占式内核如下图所示。

回到顶部



非抢占式内核的优点有：

·中断响应快(与抢占式内核比较)；

·允许使用不可重入函数；

·几乎不需要使用信号量保护共享数据。运行的任务占有CPU，不必担心被别的任务抢占。这不是绝对的，在打印机的使用上，仍需要满足互斥条件。

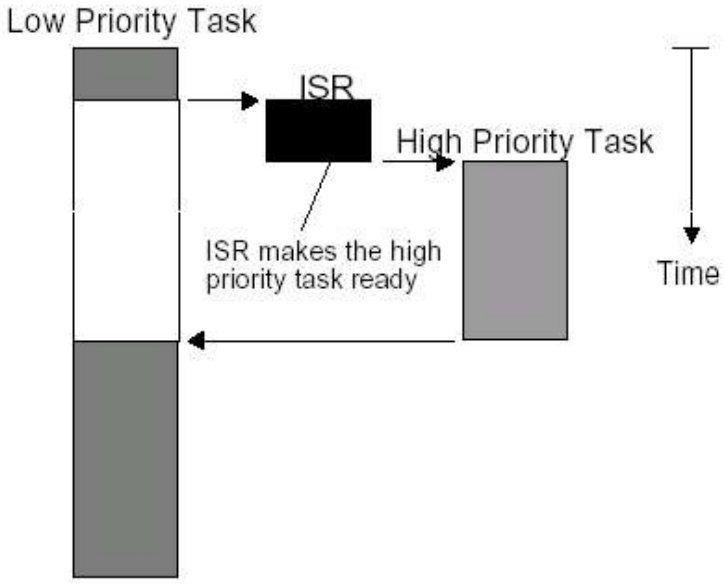
非抢占式内核的缺点有：

·任务响应时间慢。高优先级的任务已经进入就绪态，但还不能运行，要等到当前运行着的任务释放CPU。

·非抢占式内核的任务级响应时间是不确定的，不知道什么时候最高优先级的任务才能拿到CPU的控制权，完全取决于应用程序什么时候释放CPU。

抢占式内核

使 用抢占式内核可以保证系统响应时间。最高优先级的任务一旦就绪，总能得到CPU的使用权。当一个运行着的任务使一个比它优先级高的任务进入了就绪态，当前 任务的CPU使用权就会被剥夺，或者说被挂起了，那个高优先级的任务立刻得到了CPU的控制权。如果是中断服务子程序使一个高优先级的任务进入就绪态，中 断完成时，中断了的任务被挂起，优先级高的那个任务开始运行。抢占式内核如下图所示。



回到顶部

(http://photo.blog.sina.com.cn/showpic.html#blogid=502c8cc401012pxj&url=http://s1.sinaimg.cn/orignal/502c8cc4tb4a43f2eddb0)

抢占式内核的优点有：

·使用抢占式内核，最高优先级的任务什么时候可以执行，可以得到CPU的使用权是可知的。使用抢占式内核使得任务级响应时间得以最优化。

抢占式内核的缺点有：

·不能直接使用不可重入型函数。调用不可重入函数时，要满足互斥条件，这点可以使用互斥型信号量来实现。如果调用不可重入型函数时，低优先级的任务CPU的使用权被高优先级任务剥夺，不可重入型函数中的数据有可能被破坏。

2.Linux下的用户态抢占和内核态抢占

Linux除了内核态外还有用户态。用户程序的上下文属于用户态，系统调用和中断处理例程上下文属于内核态。在2.6 kernel以前，Linux kernel只支持用户态抢占。

2.1 用户态抢占(User Preemption)

在
kernel返回用户态(user-space)时，并且need_resched标志为1时，scheduler被调用，这就是用户态抢占。当kernel返回用户态时，系统可以安全的执行当前的任务，或者切换到另外一个任务。当中断处理例程或者系统调用完成后，kernel返回用户态
时，need_resched标志的值会被检查，假如它为1，调度器会选择一个新的任务并执行。中断和系统调用的返回路径(return
path)的实现在entry.S中(entry.S不仅包括kernel entry code，也包括kernel exit code)。

2.2 内核态抢占(Kernel Preemption)

在2.6
kernel以前，kernel code(中断和系统调用属于kernel
code)会一直运行，直到code被完成或者被阻塞(系统调用可以被阻塞)。在 2.6 kernel里，Linux
kernel变成可抢占式。当从中断处理例程返回到内核态(kernel-space)时，kernel会检查是否可以抢占和是否需要重新调度。
kernel可以在任何时间点上抢占一个任务(因为中断可以发生在任何时间点上)，只要在这个时间点上kernel的状态是安全的、可重新调度的。

3.内核态抢占的设计

3.1 可抢占的条件

要满足什么条件，kernel才可以抢占一个任务的内核态呢？

·没持有锁。锁是用于保护临界区的，不能被抢占。

·Kernel code可重入(reentrant)。因为kernel是SMP-safe的，所以满足可重入性。

如何判断当前上下文(中断处理例程、系统调用、内核线程等)是没持有锁的？Linux在每个每个任务的thread_info结构中增加了preempt_count变量作为preemption的计数器。这个变量初始为0，当加锁时计数器增一，当解锁时计数器减一。

3.2 内核态需要抢占的触发条件

内核提供了一个need_resched标志(这个标志在任务结构thread_info中)来表明是否需要重新执行调度。

3.3 何时触发重新调度

set_tsk_need_resched()：设置指定进程中的need_resched标志

clear_tsk need_resched()：清除指定进程中的need_resched标志

need_resched()：检查need_ resched标志的值;如果被设置就返回真，否则返回假

什么时候需要重新调度：

回到顶部

·时钟中断处理例程检查当前任务的时间片，当任务的时间片消耗完时，scheduler_tick()函数就会设置need_resched标志；

(http://photo.blog.sina.com.cn/showpic.html#blogid=502c8cc401012pxj&url=http://s1.sinaimg.cn/orignal/502c8cc4tb4a43f2eddb0)
·信号量、等到队列、completion等机制唤醒时都是基于waitqueue的，而waitqueue的唤醒函数为 (http://photo.blog.sina.com.cn/showpic.html#blogid=502c8cc401012pxj&url=http://s1.sinaimg.cn/orignal/502c8cc4tb4a43f2eddb0)default_wake_function (http://lxr.linux.no/linux+v2.6.19/+code=default_wake_function)，其调用try_to_wake_up (http://lxr.linux.no/linux+v2.6.19/+code=try_to_wake_up)将被唤醒的任务更改为就绪状态并设置need_resched标志。

·设置用户进程的nice值时，可能会使高优先级的任务进入就绪状态；

·改变任务的优先级时，可能会使高优先级的任务进入就绪状态；

·新建一个任务时，可能会使高优先级的任务进入就绪状态；

·对CPU(SMP)进行负载均衡时，当前任务可能需要放到另外一个CPU上运行；

3.4 抢占发生的时机(何时检查可抢占条件)

·当一个中断处理例程退出，在返回到内核态时(kernel-space)。这是隐式的调用schedule()函数，当前任务没有主动放弃CPU使用权，而是被剥夺了CPU使用权。

·当kernel code从不可抢占状态变为可抢占状态时(preemptible again)。也就是preempt_count从正整数变为0时。这也是隐式的调用schedule()函数。

·一个任务在内核态中显式的调用schedule()函数。任务主动放弃CPU使用权。

·一个任务在内核态中被阻塞，导致需要调用schedule()函数。任务主动放弃CPU使用权。

3.5 禁用/使能可抢占条件的操作

对preempt_count操作的函数有add_preempt_count()、sub_preempt_count()、inc_preempt_count()、dec_preempt_count()。

使能可抢占条件的操作是preempt_enable()，它调用dec_preempt_count()函数，然后再调用preempt_check_resched()函数去检查是否需要重新调度。

禁用可抢占条件的操作是preempt_disable()，它调用inc_preempt_count()函数。

在内核中有很多函数调用了preempt_enable()和preempt_disable()。比如spin_lock()函数调用了preempt_disable()函数，spin_unlock()函数调用了preempt_enable()函数。

3.6 什么时候不允许抢占

preempt_count()函数用于获取preempt_count的值，preemptible()用于判断内核是否可抢占。

有几种情况Linux内核不应该被抢占，除此之外，Linux内核在任意一点都可被抢占。这几种情况是：

·内核正进行中断处理。在Linux内核中进程不能抢占中断(中断只能被其他中断中止、抢占，进程不能中止、抢占中断)，在中断例程中不允许进行进程调度。进程调度函数schedule()会对此作出判断，如果是在中断中调用，会打印出错信息。

·内核正在进行中断上下文的Bottom Half(中断的下半部)处理。硬件中断返回前会执行软中断，此时仍然处于中断上下文中。

.

内核的代码段正持有spinlock自旋锁、writelock/readlock读写锁等锁，处干这些锁的保护状态中。内核中的这些锁是为了在SMP系
统中短时间内保证不同CPU上运行的进程并发执行的正确性。当持有这些锁时，内核不应该被抢占，否则由于抢占将导致其他CPU长期不能获得锁而死等。

回到顶部

·内核正在执行调度程序Scheduler。抢占的原因就是为了进行新的调度，没有理由将调度程序抢占掉再运行调度程序。

.

内核正在对每个CPU“私有”的数据结构操作(Per-CPU date structures)。在SMP中，对于per-CPU数据结构未用spinlocks保护，因为这些数据结构隐含地被保护了(不同的CPU有不一样的per-CPU数据，其他CPU上运行的进程不会用到另一个CPU的per-CPU数据)。但是如果允许抢占，但一个进程被抢占后重新调度，有可能调度到其他的CPU上去，这时定义的Per-CPU变量就会有问题，这时应禁抢占。

4.Linux内核态抢占的实现

4.1 数据结构

在thread_info.h中

```
1. struct thread_info {
2.     struct task_struct  *task;
3.     struct exec_domain  *exec_domain;
4.     __u32           flags;
5.     __u32           status;
6.     __u32           cpu;
7.     int         preempt_count;
9.     mm_segment_t        addr_limit;
10.     struct restart_block    restart_block;
11.     void __user     *sysenter_return;
12.  #ifdef CONFIG_X86_32
13.     unsigned long           previous_esp;
16.     __u8            supervisor_stack[0];
17.  #endif
18.  };
```

4.2 代码流程

禁用/使能可抢占条件的函数

```
1. #if defined(CONFIG_DEBUG_PREEMPT) || defined(CONFIG_PREEMPT_TRACER)
2.
3. extern void add_preempt_count(int val);
4.
5. extern void sub_preempt_count(int val);
6.
7. #else
8.
9. # define add_preempt_count(val) do { preempt_count() += (val); } while (0)
10.
11.  # define sub_preempt_count(val) do { preempt_count() -= (val); } while (0)
12.
13.  #endif
```

回到顶部

```
14.
15. #define inc_preempt_count() add_preempt_count(1)
16.
17. #define dec_preempt_count() sub_preempt_count(1)
18.
19. #define preempt_count() (current_thread_info()->preempt_count)
20.
21. #define preempt_disable() \
22.
23. do { \
24.
25.     inc_preempt_count(); \
26.
27.     barrier(); \
28.
29. } while (0)
30.
31. #define preempt_enable_no_resched() \
32.
33. do { \
34.
35.     barrier(); \
36.
37.     dec_preempt_count(); \
38.
39. } while (0)
40.
41. #define preempt_check_resched() \
42.
43. do { \
44.
45. class="cpp">    if (unlikely(test_thread_flag(TIF_NEED_RESCHED))) \
46.
47. class="cpp">class="cpp">    preempt_schedule(); \
48.
49. } while (0)
50.
51. #define preempt_enable() \
```

回到顶部

```
52.
53.  do { \
54.
55.      preempt_enable_no_resched(); \
56.
57.      barrier(); \
58.
59.      preempt_check_resched(); \
60.
61.  } while (0)
```

检查可抢占条件

```
1. # define preemptible() (preempt_count() == 0 && !irqs_disabled())
2.
```

自旋锁的加锁与解锁

```
1. void __lockfunc _spin_lock(spinlock_t *lock)
2. {
3.     preempt_disable();
4.     spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
5.     LOCK_CONTENDED(lock, _raw_spin_trylock, _raw_spin_lock);
6. }
7.
8. void __lockfunc _spin_unlock(spinlock_t *lock)
9. {
10.      spin_release(&lock->dep_map, 1, _RET_IP_);
11.      _raw_spin_unlock(lock);
12.      preempt_enable();
13.  }
```

设置need_resched标志的函数

```
1. static inline void set_tsk_need_resched(struct task_struct *tsk)
2. {
3.     set_tsk_thread_flag(tsk,TIF_NEED_RESCHED);
4. }
5.
6. static inline void clear_tsk_need_resched(struct task_struct *tsk)
7. {
8.     clear_tsk_thread_flag(tsk,TIF_NEED_RESCHED);
9. }
10.
```

回到顶部

```
11. static inline int test_tsk_need_resched(struct task_struct *tsk)
12. {
13.     return unlikely(test_tsk_thread_flag(tsk,TIF_NEED_RESCHED));
14. }
```

时钟中断时调用的task_tick()函数，当时间片消耗完之后，设置need_resched标志

```
1. static void task_tick_rt(struct rq *rq, struct task_struct *p, int queued)
2. {
3.     update_curr_rt(rq);
4.
5.     watchdog(rq, p);
6.
7.
11.     if (p->policy != SCHED_RR)
12.         return;
13.
14.     if (--p->rt.time_slice)
15.         return;
16.
17.     p->rt.time_slice = DEF_TIMESLICE;
18.
19.
23.     if (p->rt.run_list.prev != p->rt.run_list.next) {
24.         requeue_task_rt(rq, p, 0);
25.         set_tsk_need_resched(p);
26.     }
27. }
```

设置任务的need_resched标志，并触发任务所在CPU的调度器。

```
1. static void resched_task(struct task_struct *p)
2. {
3.     int cpu;
4.
5.     assert_spin_locked(&task_rq(p)->lock);
6.
7.     if (unlikely(test_tsk_thread_flag(p, TIF_NEED_RESCHED)))
8.         return;
9.
10.     set_tsk_thread_flag(p, TIF_NEED_RESCHED);
11.
```

回到顶部

```
12.     cpu = task_cpu(p);
13.     if (cpu == smp_processor_id())
14.         return;
15.
16.
17.     smp_mb();
18.     if (!tsk_is_polling(p))
19.         smp_send_reschedule(cpu);
20. }
```

5.参考资料

http://blog.csdn.net/sailor_8318/archive/2008/09/03/2870184.aspx (http://blog.csdn.net/sailor_8318/archive/2008/09/03/2870184.aspx)

《uC/OS-II源码公开的嵌入式实时多任务操作系统内核》

Linux 2.6.29内核源码

Linux内核态抢占机制分析（转）的更多相关文章 (/R/MyJxyE6A5n/)

1. Linux内核态抢占机制分析 (https://www.bbsmax.com/A/B0zq6Bm2Jv/)
   http://blog.sina.com.cn/s/blog_502c8cc401012pxj.html [摘要]本文首先介绍非抢占式内核(Non-Preemptive Kernel)和可抢占式内核(...
2. Linux内核态抢占机制分析【转】 (https://www.bbsmax.com/A/kjdww3pEdN/)
   转自:http://blog.csdn.net/yiyeguzhou100/article/details/53097665 目录(?)[-] 1非抢占式和可抢占式内核的区别 21 用户态抢占User ...
3. poll系统调用的内核态实现机制分析 (https://www.bbsmax.com/A/l1dyxL8V5e/)
   版权所有,转载请标明出处　All right reserved,Copyright by 徐行而至 浅唱而归 前面已经比较详尽的分析了系统调用引发的内核执行过程,本文将继续分析一下linux2.6 ...
4. Linux内核中锁机制之原子操作、自旋锁 (https://www.bbsmax.com/A/kvJ31PgOdg/)
   很多人会问这样的问题,Linux内核中提供了各式各样的同步锁机制到底有何作用?追根到底其实是由于操作系统中存在多进程对共享资源的并发访问,从而引起了进程间的竞态.这其中包括了我们所熟知的SMP系统,多 ...
5. 大话Linux内核中锁机制之原子操作、自旋锁 (https://www.bbsmax.com/A/gVdn1a3D5W/)
   转至:http://blog.sina.com.cn/s/blog_6d7fa49b01014q7p.html 很多人会问这样的问题,Linux内核中提供了各式各样的同步锁机制到底有何作用?追根到底其 ...
6. 大话Linux内核中锁机制之原子操作、自旋锁【转】 (https://www.bbsmax.com/A/MAzA3Rop59/)
   转自:http://blog.sina.com.cn/s/blog_6d7fa49b01014q7p.html 多人会问这样的问题,Linux内核中提供了各式各样的同步锁机制到底有何作用?追根到底其实 ...
7. Linux信号（signal) 机制分析 (https://www.bbsmax.com/A/o75NRnxD5W/)
   Linux信号(signal) 机制分析 [摘要]本文分析了Linux内核对于信号的实现机制和应用层的相关处理.首先介绍了软中断信号的本质及信号的两种不同分类方法尤其是不可靠信号的原理.接着分析了内核 ...
8. Linux内核中锁机制之RCU、大内核锁 (https://www.bbsmax.com/A/gAJGM7gndZ/)
   在上篇博文中笔者分析了关于完成量和互斥量的使用以及一些经典的问题,下面笔者将在本篇博文中重点分析有关RCU机制的相关内容以及介绍目前已被淘汰出内核的大内核锁(BKL).文章的最后对<大话Linu ...
9. Linux内核中锁机制之完成量、互斥量 (https://www.bbsmax.com/A/amd0apgmzg/)

回到顶部

在上一篇博文中笔者分析了关于信号量.读写信号量的使用及源码实现,接下来本篇博文将讨论有关完成量和互斥量的使用和一些经典问题. 八.完成量 下面讨论完成量的内容,首先需明确完成量表示为一个执行单元需要等 ...

随机推荐

1. 一种快速刷新richedit中内嵌动画的方法的实现 (https://www.bbsmax.com/A/gVdnNkOEdW/)
   在IM中使用动画表情是一种非常有趣的方式,然而选择一种合适的方式来实现却并不容易. 一般来说,除了自己去实现一个富文本控件,目前主要的解决方案有3种: 1.使用浏览器做容器. 2.使用QT提供的Ric ...
2. spftlayer 安装及简单使用 (https://www.bbsmax.com/A/Vx5Mj6ypdN/)
   1,yum -y install python-pip; pip(Python packet index);
3. Javascript中正则表达式的全局匹配模式 (https://www.bbsmax.com/A/gAJGbMYgzZ/)
   先看一道JavaScript题目,据说是国内某知名互联网企业的JavaScript笔试题,如果对正则的全局匹配模式不了解的话可能会对下面的输出结果感到疑惑. var str = "123#a ...
4. ZOJ 2852 Deck of Cards DP (https://www.bbsmax.com/A/RnJWYW8odq/)
   题意: ——个21点游戏. 1. 有三个牌堆,分别为1X,2X,3X. 2. 纸牌A的值为1,纸牌2-9的值与牌面面相同,10(T).J.Q.K的值为10,而而joke(F)的值为 任意大大. 3. ...
5. 使用fiddler对手机上的程序进行抓包 (https://www.bbsmax.com/A/lk5a936451/)
   用fiddler对手机上的程序进行抓包,网上有很多的资料,这里写一下来进行备用. 前提: 1.必须确保安装fiddler的电脑和手机在同一个wifi环境下 备注:如果电脑用的是台式机,可以安装一个 ...
6. BZOJ1785[USACO 2010 Jan Gold 3.Cow Telephones]——贪心 (https://www.bbsmax.com/A/kmzLbpMW5G/)
   题目描述 奶牛们建立了电话网络,这个网络可看作为是一棵无根树连接n(1 n 100,000)个节点,节点编号为1 .. n.每个节点可能是(电话交换机,或者电话机).每条电话线连接两个节点.第i条电话 ...
7. Android笔记（六）：线程及线程通信 (https://www.bbsmax.com/A/WpdK7P4odV/)
   线程 由于Android的Activity中默认所有代码都在主线程(UI线程)中执行,如果在这里面执行耗时任务(例如下载),界面就会无反应且不可操作,直到耗时任务执行完毕. 如果想在执行耗时任务的同时 ...
8. 让sublime text3支持Vue语法高亮显示[转] (https://www.bbsmax.com/A/kvJ3pYn75g/)
   1.准备语法高亮插件vue-syntax-highlight. 下载地址:https://github.com/vuejs/vue-syntax-highlight 下载页面并下载: 解开压缩包vue ...
9. laravel时间判断 (https://www.bbsmax.com/A/xl56jwPx5r/)
   $now = Carbon::now(); if ($now >= '2019-01-02') { }
10. spring cloud 之 Eureka 知识点 (https://www.bbsmax.com/A/LPdoL9yNz3/)
    Eureka原理 当服务消费者想要调用服务提供者的API时,首先会在注册中心中查询当前可用的实例的网络地址(也可能是定时查询可用实例,本地缓存好可用服务列表),然后再使用客户端负载均衡,命中到其中一个 ...

上一篇:在Java中system.out.println使用方法 (/blogs-details/20210521005230672r)
下一篇:Python学习之路——socket (/blogs-details/20210521005230384J)
回到顶部

相似推荐

1. 企查猫app数据解密 (/blogs-details/20190907114037701427777qfm46qq5r0)
2. 战略性思维模式MindSet的优势以及如何开发？ - modernanalyst (/blogs-details/20210430183831057M)
3. OPC的理解Open Packaging Conventions (/blogs-details/20210518223316037d)
4. a标签绝对定位，点击区域被图片遮挡（IE下） (/blogs-details/20210519023420144M)
5. PowerDesigner 16.5对SQL Server 2012 生成数据库时'不支持扩展属性'问题 (/blogs-details/20210520163911623s)
6. 自学SQL语言的例子（使用MySQL实现） (/blogs-details/20210528122110099j)
7. RNN Layer (/blogs-details/20210811200936436D)
8. 安洵杯2021_Crypto_复现 (/blogs-details/202204212123034300)

友情链接

Java人：专注Java开发 (https://javamana.com/)
资讯整合：IT数据的整合 (https://chowdera.com/)
前端人：专注前端开发 (https://qdmana.com/)
编程人：编程的世界，专注各种开发 (https://cdmana.com/)
Python人：专注Python开发 (https://pythonmana.com/)
摄影圈子：专业摄影知识，旅游摄影分享 (https://hqmana.com/)
副头条：IT资讯、技术知识 (https://fheadline.com/)
娱乐先锋 (https://bfun.fun/)

回到顶部

Programming knowledge (https://chenhaoxiang.cn/)
汽车之讯 (https://car.inotgo.com/)
区块链之家 (https://netfreeman.com/)

广告合作

()

赞助商

阿里云

回到顶部