

由鸿铭

E-mail: hithongming@163.com

博客园 首页 联系 管理

随笔-52 文章-0 评论-2

【软件构造】第十章 线程和分布式系统

## 第十章 线程和分布式系统

本章关注复杂软件系统的构造。本章关注复杂软件系统的构造。这里的“复杂”包括三方面：这里的“复杂”包括三方面：（1）多线程程序（2）分布式程序（3）GUI 程序

### Outline

- 并发编程
  - Shared memory
  - Message passing
- 进程和线程
- 线程的创建和启动, **runable**
- 时间分片、交错执行、竞争条件
- 线程的休眠、中断
- 线程安全的四种策略
  - 约束（**Confinement**）
  - 不变性
  - 使用线程安全的数据类型
  - 同步与锁
- 死锁
- 以注释的形式撰写线程安全策略

### Notes

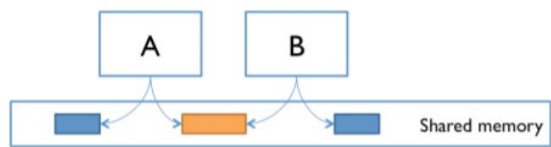
#### ## 并发编程

【并发(**concurrency**)】

- 定义：指的是多线程场景下对共享资源的争夺运行
- 并发的应用背景：
  - 网络上的多台计算机
  - 一台计算机上的多个应用
  - 一个CPU上的多核处理器
- 为什么要有并发：
  - 摩尔定律失效、“核”变得越来越多
  - 为了充分利用多核和多处理器需要将程序转化为并行执行
- 并发编程的两种模式：
  - 共享内存：在内存中读写共享数据
  - 信息传递（**Message Passing**）：通过channel交换消息

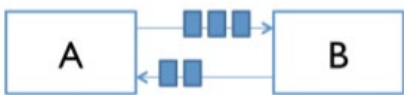
【共享内存】

- 共享内存这种方式比较常见，我们经常会设置一个共享变量，然后多个线程去操作同一个共享变量。从而达到线程通讯的目的。
- 例子：
  - 两个处理器，共享内存
  - 同一台机器上的两个程序，共享文件系统
  - 同一个Java程序内的两个线程，共享Java对象



【信息传递】

- 消息传递方式采取的是线程之间的直接通信，不同的线程之间通过显式的发送消息来达到交互目的
- 接收方将收到的消息形成队列逐一处理，消息发送者继续发送（异步方式）
- 消息传递机制也无法解决竞争条件问题
- 仍然存在消息传递时间上的交错
- 例子：
  - 网络上的两台计算机，通过网络连接通讯
  - 浏览器和Web服务器，A请求页面，B发送页面数据给A
  - 即时通讯软件的客户端和服务端
  - 同一台计算机上的两个程序，通过管道连接进行通讯



并发模型	通信机制	
共享内存	线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信。	同步是
消息传递	线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信。	

## 进程和线程

- 进程：是执行中一段程序，即一旦程序被载入到内存中并准备执行，它就是一个进程。进程是表示资源分配的的基本概念，又是调度运行的基本单位，是系统中的并发执行的单位。
  - 程序运行时在内存中分配自己独立的运行空间
  - 进程拥有整台计算机的资源

- 多进程之间不共享内存
- 进程之间通过消息传递进行协作
- 一般来说，进程==程序==应用（但一个应用中可能包含多个进程）
- OS支持的IPC机制(pipe/socket)支持进程间通信（IPC不仅是本机的多个进程之间，也可以是不同机器的多个进程之间）
- JVM通常运行单一进程，但也可以创建新的进程。
- 线程：它是位于进程中，负责当前进程中的某个具备独立运行资格的空间。
  - 线程有自己的堆栈和局部变量，但是多个线程共享内存空间
  - 进程=虚拟机；线程=虚拟CPU
  - 程序共享、资源共享，都隶属于进程
  - 很难获得线程私有的内存空间
  - 线程需要同步：在改变对象时要保持lock状态
  - 清理线程是不安全的
- 进程是负责整个程序的运行，而线程是程序中的某个独立功能的运行。
- 一个进程中至少应该有一个线程。
- 主线程可以创建其他的线程。

## ## 线程的创建和启动，runable

### 【方式1：继承Thread类】

- 方法：用Thread类实现了Runnable接口，但它其中的run方法什么都没做，所以用一个类做Thread的子类，提供它自己实现的run方法。用Thread.start()来开始一个新的线程。
- 创建：A类 a = new A类();
- 启动：a.start();
- 步骤：
  - 定义一个类A继承于java.lang.Thread类。
  - 在A类中覆盖Thread类中的run方法。
  - 我们在run方法中编写需要执行的操作：run方法里的代码,线程执行体。
  - 在main方法(线程)中,创建线程对象,并启动线程。
- 栗子：



```
1 //1):定义一个类A继承于java.lang.Thread类.
2 class MusicThread extends Thread{
3     //2):在A类中覆盖Thread类中的run方法.
4     public void run() {
5         //3):在run方法中编写需要执行的操作
6         for(int i = 0; i < 50; i++){
7             System.out.println("播放音乐"+i);
8         }
9     }
10 }
11
12 public class ExtendsThreadDemo {
13     public static void main(String[] args) {
14
15         for(int j = 0; j < 50; j++){
```

```

16         System.out.println("运行游戏"+j);
17         if(j == 10){
18             //4):在main方法(线程)中,创建线程对象,并启动线程.
19             MusicThread music = new MusicThread();
20             music.start();
21         }
22     }
23 }
25 }

```



## 【方式2: 实现Runnable接口】

- 创建: Thread t = new Thread(new A());
- 调用: t.start();
- 步骤:
  - 定义一个类A实现于java.lang.Runnable接口,注意A类不是线程类.
  - 在A类中覆盖Runnable接口中的run方法.
  - 我们在run方法中编写需要执行的操作: run方法里的,线程执行体.
  - 在main方法(线程)中,创建线程对象,并启动线程.



```

1 //1):定义一个类A实现于java.lang.Runnable接口,注意A类不是线程类.
2 class MusicImplements implements Runnable{
3     //2):在A类中覆盖Runnable接口中的run方法.
4     public void run() {
5         //3):在run方法中编写需要执行的操作
6         for(int i = 0; i < 50; i++){
7             System.out.println("播放音乐"+i);
8         }
9     }
10 }
11 }
12
13 public class ImplementsRunnableDemo {
14     public static void main(String[] args) {
15         for(int j = 0; j < 50; j++){
16             System.out.println("运行游戏"+j);
17             if(j == 10){
18                 //4):在main方法(线程)中,创建线程对象,并启动线程
19                 MusicImplements mi = new MusicImplements();
20                 Thread t = new Thread(mi);
21                 t.start();
22             }
23         }
24     }

```



- 实现Runnable接口相比继承Thread类有如下好处：
  - 避免点继承的局限，一个类可以继承多个接口。
  - 适合于资源的共享
- 创建并运行一个线程所犯的常见错误是调用线程的 run()方法而非 start()方法，如下所示：

```
Thread newThread = new Thread(MyRunnable());  
newThread.run(); //should be start();
```

起初并不会感觉到有什么不妥，因为 run()方法的确如你所愿的被调用了。但是，事实上,run()方法并非是由刚创建的新线程所执行的，而是被创建新线程的当前线程所执行了。也就是被执行上面两行代码的线程所执行的。想要让创建的新线程执行 run()方法，必须调用新线程的 start 方法。

## ## 时间分片、交错执行、竞争条件

### 【时间分片】

- 虽然有多线程，但只有一个核，每个时刻只能执行一个线程。
  - 通过时间分片，再多个线程/进程之间共享处理器
- 即使是多核CPU，进程/线程的数目也往往大于核的数目
- 通过时间分片，在多个进程/线程之间共享处理器。（时间分片是由OS自动调度的）
- 当线程数多于处理器数量时，并发性通过时间片来模拟，处理器切换处理不同的线程

### 【交错执行】

顾名思义，就是在线程运行的过程中，多个线程同时运行相互交错。而且，由于线程运行一般不是连续的，那么就会导致线程间的交错。可以说，所有线程安全问题的本质都是线程交错的问题。

### 【竞争条件】

竞争是发生在线程交错的基础上的。当多个线程对同一对象进行读写访问时，就可能会导致竞争的问题。程序中可能出现的一种问题就是，读写数据发生了不同步。例如，我要用一个数据，在该数据修改还没写回内存中时就读取出来了，那么就会导致程序出现问题。

程序运行时有一种情况，就是程序如果要正确运行，必须保证A线程在B线程之前完成（正确性意味着程序运行满足其规约）。当发生这种情况时，就可以说A与B发生竞争关系。

- 计算机运行过程中，并发、无序、大量的进程在使用有限、独占、不可抢占的资源，由于进程无限，资源有限，产生矛盾，这种矛盾称为竞争（Race）。
- 由于两个或者多个进程竞争使用不能被同时访问的资源，使得这些进程有可能因为时间上推进的先后原因而出现问题，这叫做竞争条件（Race Condition）。
- 竞争条件分为两类：
  - Mutex（互斥）：两个或多个进程彼此之间没有内在的制约关系，但是由于要抢占使用某个临界资源（不能被多个进程同时使用的资源，如打印机，变量）而产生制约关系。
  - Synchronization（同步）：两个或多个进程彼此之间存在内在的制约关系（前一个进程执行完，其他的进程才能执行），如严格轮转法。
- 解决互斥方法：
  - Busy Waiting(忙等待)：等着但是不停的检查测试，不睡觉，知道能进行为止

**Sleep and Wakeup**(睡眠与唤醒): 引入**Semaphore**(信号量, 包含整数和等待队列,为进程睡觉而设置), 唤醒由其他进程引发。

- 临界区 (Critical Region) :
  - 一段访问临界资源的代码。

为了避免出现竞争条件, 进入临界区要遵循四条原则:

- 任何两个进程不能同时进入访问同一临界资源的临界区
  - 进程的个数, CPU个数性能等都是无序的, 随机的
  - 临界区之外的进程不得阻塞其他进程进入临界区
  - 任何进程都不应被长期阻塞在临界区之外
- 解决互斥的方法:
  - 禁用中断 Disabling interrupts
  - 锁变量 Lock variables (no)
  - 严格轮转 Strict alternation (no)
  - Peterson's solution (yes)
  - The TSL instruction (yes)

## ## [线程的休眠、中断](#)

### 【Thread.sleep】

- 在线程中允许一个线程进行暂时的休眠, 直接使用**Thread.sleep()**方法即可。
  - 将某个线程休眠, 意味着其他线程得到更多的执行机会
  - 进入休眠的线程不会失去对现有**monitor**或锁的所有权
- **sleep**定义格式:

```
public static void sleep(long millis,int nanos)
    throws InterruptedException
```

首先, **static**, 说明可以由**Thread**类名称调用, 其次**throws**表示如果有异常要在调用此方法处处理异常。

所以**sleep ()** 方法要有**InterruptedException** 异常处理, 而且**sleep ()** 调用方法通常为**Thread.sleep(500)** ;形式。

- 实例:

```
for (int i = 0; i < n; i++) {
    //Pause for 4 seconds
    Thread.sleep(4000);
    //Print a message
    System.out.println(msg[i]);
}
```

### 【Thread.interrupt】

- 一个线程可以被另一个线程中断其操作的状态, 使用 **interrupt ()** 方法完成。
  - 通过线程的实例来调用**interrupt()**函数, 向线程发出中断信号
  - **t.interrupt()**: 在其他线程里向**t**发出中断信号
  - **t.isInterrupted()**: 检查**t**是否已在中断状态中
- 当某个线程被中断后, 一般来说应停止 其**run()**中的执行, 取决于程序员在**run()**中处理
  - 一般来说, 线程在收到中断信号时应该中断, 直接终止

- 但是，线程收到其他线程发出来的中断信号，并不意味着一定要“停止”

- 实例：

```
class Task implements Runnable{
    private double d = 0.0;

    public void run() {
        try{
            while (true) {
                for (int i = 0; i < 900000; i++)
                    d = d + (Math.PI + Math.E) / d;
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {return;}
    }
}

Thread t = new Thread(new Task());
t.start();
Thread.sleep(100); //当前线程休眠
t.interrupt(); //试图中断t线程
```

在sleep()的时候检测是否收到别人的中断信号。若是，抛出异常

正常运行期间，即使接收到中断信号，也不理会

进入异常处理后，线程才真正终止

- 实例二：

```
package Thread1;

class MyThread implements Runnable{    // 实现Runnable接口
    public void run(){    // 覆写run()方法
        System.out.println("1、进入run()方法") ;
        try{
            Thread.sleep(10000) ;    // 线程休眠10秒
            System.out.println("2、已经完成了休眠") ;
        }catch(InterruptedException e){
            System.out.println("3、休眠被终止") ;
            return ; // 返回调用处
        }
        System.out.println("4、run()方法正常结束") ;
    }
};

public class demo1{
    public static void main(String args[]){
        MyThread mt = new MyThread() ;    // 实例化Runnable子类对象
        Thread t = new Thread(mt,"线程") ;    // 实例化Thread对象
        t.start() ;    // 启动线程
        try{
            Thread.sleep(2000) ;    // 线程休眠2秒
        }catch(InterruptedException e){
            System.out.println("3、休眠被终止") ;
        }
        t.interrupt() ;    // 中断线程执行
    }
};
```

运行结果：

```
1、进入run()方法
3、休眠被终止
```

## ## 线程安全的四个策略

- 线程安全的定义：ADT或方法在多线程中要执行正确，即无论如何执行，不许调度者做额外的协作，都能满足正确性
- 四种线程安全的策略：
  - Confinement 限制数据共享
  - Immutability 共享不可变数据
  - Threadsafe data type 共享线程安全的可 变数据
  - Synchronization 同步机制共享共享线程 不安全的可变数据，对外即为线程安全的ADT.

### 【Confinement 限制数据共享】

- 核心思想：线程之间不共享mutable数据类型
  - 将可变数据限制在单一线程内部，避免竞争
  - 不允许任何县城直接读写该数据
- 在多线程环境中，取消全局变量，尽量避免使用不安全的静态变量。
  - 限制数据共享主要是在线程内部使用局部变量，因为局部变量在每个函数的栈内，每个函数都有自己的栈结构，互不影响，这样局部变量之间也互不影响。
  - 如果局部变量是一个指向对象的引用，那么就需要检查该对象是否被限制住，如果没有被限制住（即可以被其他线程所访问），那么就有限制住数据，因此也就不能用这种方法来保证线程安全
- 栗子：



```
public class Factorial {

    /**
     * Computes n! and prints it on standard output.
     * @param n must be >= 0
     */
    private static void computeFact(final int n) {
        BigInteger result = new BigInteger("1");
        for (int i = 1; i <= n; ++i) {
            System.out.println("working on fact " + n);
            result = result.multiply(new BigInteger(String.valueOf(i)));
        }
        System.out.println("fact(" + n + ") = " + result);
    }

    public static void main(String[] args) {
        new Thread(new Runnable() { // create a thread using an
            public void run() {      // anonymous Runnable
                computeFact(99);
            }
        }).start();
        computeFact(100);
    }
}
```





解释：主函数开启了两个线程，调用的是相同函数。因为线程共享局部变量的类型，但每个函数调用有不同的栈，因此有不同的*i*，*n*，*result*。由于每个函数都有自己的局部变量，那么每个函数就可以独立运行，更新它们自己的函数值，线程之间不影响结果。

### 【Immutability 共享不可变数据】

不可变数据类型，指那些在整个程序运行过程中，指向内存的引用是一直不变的，通常使用*final*来修饰。不可变数据类型通常来讲是线程安全的，但也可能发生意外。

但是，程序在运行过程中，有时为了优化程序结构，默默地将这个引用更改了。此时，客户端程序员是不知道它被更改了，对于客户端而言，这个引用还是不可变的，但其实已经被悄悄更改了。这时就会发生一些线程安全问题。

解决方案就是给这些不可变数据类型再增加一些限制：

- 所有的方法和属性都是私有的。
- 不提供可变的方法，即不对外开放可以更改内部属性的方法。
- 没有数据的泄露，即返回值而不是引用。
- 不在其中存储可变数据对象。

这样就可以保证线程的安全了。

### 【Threadsafe data type（共享线程安全的可变数据）】

- 方法：如果必须要用*mutable*的数据类型在多线程之间共享数据，要使用线程安全的数据类型。（在JDK中的类，文档中明确指明了是否*threadsafe*）
- 一般来说，JDK同时提供两个相同功能的类，一个是*threadsafe*，另一个不是。原因：*threadsafe*的类一般性能上受影响。
- *List*、*Set*、*Map*这些集合类都是线程不安全的，Java API为这些集合类提供了进一步的decorator



```
private static Map<Integer, Boolean> cache = Collections.synchronizedMap(new HashMap<>());
public static <T> Collection<T> synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K, V> Map<K, V> synchronizedMap(Map<K, V> m);
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);
public static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> m);
```



- \*\*\*在使用*synchronizedMap(hashMap)*之后，不要再把参数*hashMap*共享给其他线程，不要保留别名，一定要彻底销毁。（可以用*private static Map cache = Collections.synchronizedMap(new HashMap<>());*的方式实例化集合类）
- 即使在线程安全的集合类上，使用*iterator*也是不安全的：



```
List<Type> c = Collections.synchronizedList(new
ArrayList<Type>());
synchronized(c) { // to be introduced later (the 4-th threadsafe way)
```

```
for (Type e : c)
    foo(e);
}
```



- 需要注意用java提供的包装类包装集合后，只是将集合的每个操作都看成了原子操作，也就保证了每个操作内部的正确性，但是在两个操作之间不能保证集合类不被修改，因此需要用lock机制，例如

```
if ( ! lst.isEmpty()) {
    String s = lst.get(0);
    ...
}
```

两行语句可能产生interleaving

如果在isEmpty和get中间，将元素移除，也就产生了竞争。

前三种策略的核心思想：避免共享 --> 即使共享，也只能读/不可写(**immutable**) --> 即使可写(**mutable**)，共享的可写数据应自己具备在多线程之间协调的能力，即“使用线程安全的**mutable ADT**”

### 【Synchronization 同步与锁】

- 为什么要同步
  - java允许多线程并发控制，当多个线程同时操作一个可共享的资源变量时（如数据的增删改查）
  - 将会导致数据不准确，相互之间产生冲突，因此加入同步锁以避免在该线程没有完成操作之前，被其他线程的调用，
  - 从而保证了该变量的唯一性和准确性。
- 同步方法
  - 即有synchronized关键字修饰的方法。
  - 由于java的每个对象都有一个内置锁，当用此关键字修饰方法时，内置锁会保护整个方法。
  - 在调用该方法前，需要获得内置锁，否则就处于阻塞状态。
  - 代码如下：

```
public synchronized void save() {}
```

- 注：synchronized关键字也可以修饰静态方法，此时如果调用该静态方法，将会锁住整个类
- 同步代码块
  - 在调用该方法前，需要获得内置锁，否则就处于阻塞状态。
  - 被该关键字修饰的语句块会自动被加上内置锁，从而实现同步。
  - 代码如下：

```
synchronized(object){    }
```

- 注：同步是一种高开销的操作，因此应该尽量减少同步的内容。
- 使用锁机制，获得对数据的独家mutation权，其他线程被阻塞，不得访问
- Lock是Java语言提供的内嵌机制，每个object都有相关联的lock
- 任何共享的mutable变量/对象必须被lock所保护
- 涉及到多个mutable变量的时候，它们必须被同一个lock所保护

## ## 死锁

- 定义：两个或多个线程相互等待对方释放锁，则会出现死锁现象。
- java虚拟机没有检测，也没有采取措施来处理死锁情况，所以多线程编程是应该采取措施避免死锁的出现。一旦出现死锁，

整个程序即不会发生任何异常，也不会给出任何提示，只是所有线程都处于堵塞状态。

- 形成死锁的条件：

- 互斥条件：线程使用的资源必须至少有一个是不能共享的（至少有锁）；
- 请求与保持条件：至少有一个线程必须持有一个资源并且正在等待获取一个当前被其它线程持有的资源（至少两个线程持有不同锁，又在等待对方持有锁）；
- 非剥夺条件：分配资源不能从相应的线程中被强制剥夺（不能强行获取被其他线程持有锁）；
- 循环等待条件：第一个线程等待其它线程，后者又在等待第一个线程（线程A等线程B；线程B等线程C;...;线程N等线程A。如此形成环路）。

- 防止死锁的方法：

- 加锁顺序：当多个线程需要相同的一些锁，但是按照不同的顺序加锁，死锁就很容易发生。如果能确保所有的线程都是按照相同的顺序获得锁，那么死锁就不会发生。这种方式是一种有效的死锁预防机制。但是，这种方式需要你事先知道所有可能会用到的锁，但总有些时候是无法预知的

```
public void friend(Wizard that) {
    Wizard first, second;
    if (this.name.compareTo(that.name) < 0) {
        first = this; second = that;
    } else {
        first = that; second = this;
    }
    synchronized (first) {
        synchronized (second) {
            if (friends.add(that)) {
                that.friend(this);
            }
        }
    }
}
```

- 使用粗粒度的锁，用单个锁来监控多个对象

- 对整个社交网 络设置 一个锁， 并且对其任何组成部分的所有操作都在该锁上进行同步。
- 例如：所有的Wizards都属于一个Castle，可使用 castle 实例的锁

缺点：性能损失大；

- 如果用一个锁保护大量的可变数据，那么久放弃了同时访问这些数据的能力；
- 在最糟糕的情况下， 程序可能基本上是顺序执行的，丧失了并发性

```
public class Wizard {
    private final Castle castle;
    private final String name;
    private final Set<Wizard> friends;
    ...
    public void friend(Wizard that) {
        synchronized (castle) {
            if (this.friends.add(that)) {
                that.friend(this);
            }
        }
    }
}
```

- 加锁时限：在尝试获取锁的时候加一个超时时间，这也就意味着在尝试获取锁的过程中若超过了这个时限该线程则放弃对该锁请求。若一个线程没有在给定的时限内成功获得所有需要的锁，则会进行回退并释放所有已经获得的锁。
- 用 jstack 等工具进行死锁检测

## ## 以注释的形式撰写线程安全策略

- 在代码中以注释的形式添加说明：该ADT采取了什么设计决策来保证线程安全
- 阐述如何使rep线程安全；
- 写入表示不变性的说明中，以便代码维护者知道你是如何为类设计线程安全性的。
- 需要对安全性进行这种仔细的论证，阐述使用了哪种技术，使用threadsafe data types, or synchronization时，需要论证所有对数据的访问都是具有原子性的
- 栗子：

```
/** MyString is an immutable data type representing a string of characters. */
public class MyString {
    private final char[] a;
    // Thread safety argument:
    //   This class is threadsafe because it's immutable:
    //   - a is final
    //   - a points to a mutable char array, but that array is encapsulated
    //     in this object, not shared with any other object or exposed to a
    //     client
```

```
/** MyString is an immutable data type representing a string of characters. */
public class MyString {
    private final char[] a;
    private final int start;
    private final int len;
    // Rep invariant:
    //   0 <= start <= a.length
    //   0 <= len <= a.length-start
    // Abstraction function:
    //   represents the string of characters a[start],...,a[start+length-1]
    // Thread safety argument:
    //   This class is threadsafe because it's immutable:
    //   - a, start, and len are final
    //   - a points to a mutable char array, which may be shared with other
    //     MyString objects, but they never mutate it
    //   - the array is never exposed to a client
```

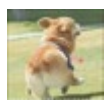
- 反例

```
/** MyStringBuffer is a threadsafe mutable string of characters. */
public class MyStringBuffer {
    private String text;
    // Rep invariant:
    //   none
    // Abstraction function:
    //   represents the sequence text[0],...,text[text.length()-1]
    // Thread safety argument:
    //   text is an immutable (and hence threadsafe) String,
    //   so this object is also threadsafe
```

- 字符串是不可变的并且是线程安全的; 但是指向该字符串的rep，特别是文本变量，并不是不可变的；
- 文本不是最终变量，因为我们需要数据类型来支持插入和删除操作；
- 因此读取和写入文本变量本身不是线程安全的。

E-mail: [hithongming@163.com](mailto:hithongming@163.com)

分类: [软件构造](#)



[由鸿铭](#)  
[关注 - 2](#)  
[粉丝 - 9](#)

[+加关注](#)

0  
推荐

0  
反对

« 上一篇: [【软件构造】第八章第三节 代码调优的设计模式和I/O](#)

» 下一篇: [【算法复习】分治算法](#)

posted @ 2018-06-20 16:28 由鸿铭 阅读(503) 评论(1) 编辑 收藏

评论列表


#1楼 2018-06-20 19:37 大风车转啊转 

整理的很不错，给你点赞！

回复 引用

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

 发表评论

昵称：

评论内容： **B**    

退出 订阅评论

[Ctrl+Enter快捷键提交]

- 【推荐】超50万C++/C#源码: 大型实时仿真组态图形源码
- 【前端】SpreadJS表格控件，可嵌入系统开发的在线Excel
- 【推荐】程序员问答平台，解决您开发中遇到的技术难题

- 相关博文：
- 【软件构造】课程提纲（7）
  - 并发编程之线程安全性
  - 并发基础知识 — 线程安全性
  - 并发基础知识 — 线程安全性
  - Java并发编程实战1,2章

最新新闻：

- 哈佛的突破性研究表明干细胞可以在体内进行基因编辑
  - 两起凶案发生一年之后，滴滴有了哪些变化？
  - 手机市场竞争日益激烈 三星宣布削减中国手机工厂产量
  - 天文学家呼吁对SpaceX Starlink等卫星集群项目加以限制
  - 海思的蛰伏与挑战
- » 更多新闻...

公告

昵称：由鸿铭  
园龄：1年3个月  
粉丝：9  
关注：2  
[+加关注](#)

2019年6月						
<						>
日	一	二	三	四	五	六
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	1	2	3	4	5	6

搜索

- 常用链接
- 我的随笔
  - 我的评论
  - 我的参与
  - 最新评论
  - 我的标签

- 随笔分类
- 东软实训(2)
  - 计算机网络(16)
  - 软件构造(30)
  - 算法设计与分析(3)

- 随笔档案
- 2018年8月 (3)

2018年7月 (16)

2018年6月 (27)

2018年5月 (1)

2018年3月 (3)

2018年2月 (2)

#### 最新评论

##### 1. Re: 【软件构造】第十章 线程和分布式系统

整理的很不错，给你点赞！

--大风车转啊转

##### 2. Re: 【软件构造】第一章第二节 软件开发的质量属性

大纲总结的不错，感谢分享

--李秋豪

#### 阅读排行榜

1. 【软件构造】第十章 线程和分布式系统(503)
2. 【软件构造】写在前面的话和课程介绍（完结）(436)
3. 【算法复习】动态规划(359)
4. 【软件构造】第七章第三节 断言和防御性编程(307)
5. 【算法复习】分治算法(299)

#### 评论排行榜

1. 【软件构造】第一章第二节 软件开发的质量属性(1)
2. 【软件构造】第十章 线程和分布式系统(1)

#### 推荐排行榜

Copyright ©2019 由鸿铭

1. 【软件构造】第一章第一节 软件构造的多维视角(2)
2. 【软件构造】写在前面的话和课程介绍（完结）(1)
3. 【算法复习】动态规划(1)
4. 【软件构造】第八章第三节 代码调优的设计模式和I/O(1)