

**并发（一）**

**如需转载请注明原地址，谢谢**

## 写在前面（与并发相关的操作系统知识）：

### ****1、并发和并行的区别：****

**并行：**两件或多件事务在同一时刻发生。

**并发：**两件或多件事务在同一时间间隔发生

### ****2、多线程设计的主要原因：****

1、许多应用中同时发生着多个活动，其中某些活动会随着时间的推移被阻塞，并行实体共享一个地址空间和所有可用数据的能力

2、线程比进程更轻量级，所以他们比进程更容易创建和撤销

3、多线程允许活动彼此重叠进行，从而加快应用程序执行的速度

### ****3、程序、线程、进程的区别：****

**程序：**指一组指示计算机每一步动作的[指令](http://baike.baidu.com/view/178461.htm" \t "http://blog.csdn.net/fuckluy/article/details/_blank)，通常用某种[程序设计语言](http://baike.baidu.com/view/128511.htm" \t "http://blog.csdn.net/fuckluy/article/details/_blank)编写，运行于某种目标[体系结构](http://baike.baidu.com/view/1188494.htm" \t "http://blog.csdn.net/fuckluy/article/details/_blank)上。

**进程：**对正在运行程序的一个抽象（抽象了CPU以运行程序），一个进程就是一个正在执行程序的实例，包括程序计数器、寄存器和变量的当前值，进程是某种类型的活动，它有程序、输入、输出以及状态

**线程：**迷你进程，切割进程的CPU时间，共享进程的地址空间以及全局变量，线程没有保护

**程序和进程的区别：**

1、程序是静态的概念，进程是动态的概念

2、程序是永久的，可以保存在外存中，一个进程是一个正在执行程序的实例，可以被销毁，存在是暂时的

3、进程具有并发性，互斥性，而程序没有，如果一个程序运行了两遍则算两个进程

4、在计算机网络中，进程才是进行数据交互的窗口，而程序不是

**进程和线程的区别：**

1、进程是对运行程序的抽象，包含一个程序的实例，而线程是进程的一部分，切分进程的CPU时间

2、不同的进程之间是有保护的，而同一进程的不同线程之间是没有保护的

3、不同的进程拥有不同的地址空间（现代操作系统中会共享部分地址空间），而同一进程的不同线程拥有相同的地址空间，共享全局变量、打开文件、子进程等

4、线程程是mini进程，同样拥有寄存器，程序计数器以及堆栈（这些每个线程都是独立的）

5、线程相比于进程更容易创建和撤销，线程存在的目的是为了提高程序的运行速度

**额外的补充：**

**线程包的实现**可以在内核态也可以在用户态，对于在**用户空间实现的线程包**，内核不知道线程的存在，线程的运行由进程决策且在该进程拥有CPU时间切片的情况下不可能切换到别的进程中的线程对于在**内核实现线程包**，内核知道线程的存在，且可以对线程进行调度而不考虑线程属于哪个进程，简单点老说，如果有进程A,B，进程A中有线程A1A2A，进程B中有线程B1B2，如果线程包实现在用户态，线程的调度只能是，A1到A2或者B1到B2，而如果线程包实现在内核态，线程的调度可以是A1到B2，B2到A2，线程的调度可以跨进程。

**用户级和内核级线程的差别在于性能**，用户级的线程切换只需要简单的机器指令，而内核级的线程切换需要完整的上下文切换，修改内存映射等非常耗时

### ****4、进程，线程的几种状态：****

**进程的三种状态：**

1、运行态（获取CPU时间）

2、就绪态（只要获取CPU时间就可以运行）

3、阻塞态（除非某种外部事件发生，否则不能运行，即使CPU空闲）

**进入阻塞的两种情况：**1、逻辑上不能运行，比如等待有效的输入。2、有技术原因引起的阻塞，没有足够的CPU执行该进程

**线程的四种状态：（和进程的状态几乎是一样的）**

1、新建状态：分配了系统必要的资源，并执行了初始化，有资格获得CPU时间了

2、就绪态：只要获取CPU时间就可以运行

3、阻塞：线程能运行但被某个条件阻止，不能获取CPU时间

4、死亡：该状态的线程不可调度，也不能获取CPU时间

**JAVA线程进入阻塞的原因：**

1、线程调用sleep（）方法进入休眠状态------->可中断阻塞，不会释放锁，将cpu给线程调度器

2、调用wait（）方法使线程挂起，直到线程得到notify（）或notifyAll（）消息，让出系统资源，释放CPU，释放锁

3、等待某个任务的输入/输出完成--------->不可中断，除了socketChannel可以中断，Piped(管道)I/O 可以中断

4、试图在某个对象上调用其同步控制方法，但对象锁不可用-------------->使用synchronized不可中断，使用可重入锁ReentrantLock可中断

### ****5、同步和互斥用到的信号量和互斥量的区别：****

1、互斥量用于线程的互斥操作，信号量用于线程的同步操作（信号量可以实现互斥的功能）**互斥：**是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。**同步：**是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源

2、信号量是一个非负的整数，互斥量只有0和非零两种状态（也就是说，一个互斥量只能用于一个资源的互斥访问，它不能实现多个资源的多线程互斥问题。信号量可以实现多个同类资源的多线程互斥和同步。当信号量为单值信号量是，也可以完成一个资源的互斥访问。）

3、互斥量的加锁和解锁必须由同一线程分别对应使用，信号量可以由一个线程释放，另一个线程得到

### 6、同步和异步的区别：

**同步：**发送一个请求，等待返回，然后再发送下一个请求

**异步：**发送一个请求，不需要等待返回，发送下一个请求

Java中的回调就是异步的，回调，参考下面的文章：[点击打开链接](http://blog.csdn.net/fuckluy/article/details/51318297" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

### 7、死锁：

**死锁：**指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去

比如：两个进程A、B和两个资源1和2，A获取的1，B获取了2，A和B都请求获取另一个资源就会被阻塞从而产生死锁

**资源死锁的四个必要条件：**

1、互斥条件，资源只能被一个进程使用

2、占有和等待条件，得到某个资源的进程可以再请求新的资源且不释放已经占有的资源

3、不可抢占，已经分配给某个进程的资源不能被其他进程抢占，只能自己释放

4、**循环等待**（这个条件易被破坏而不产生死锁），有两个或两个以上进程形成环路

**其他死锁：**

1、**通信死锁**，即两个或两个以上的进程利用发送信息来通信时产生死锁

比如：进程A向进程B发送请求信息并阻塞等待B的回复，假如信息丢失，A阻塞等待回复，B阻塞等待一个请求，产生死锁

**解决办法：**发送信息是启动一个计时器，规定时间内未收到回复就重发

2、**活锁**，任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。

3、**饥饿**，一个进程永远无法得到某个资源而无限制的推后

**操作系统中避免死锁的方式：**主要运用了Dijkstra的银行家算法，进程每请求获取一个资源时都会通过该算法检查该进程获得资源后的状态，如果是安全态，则允许获取该资源（安全态一定不会产生死锁），如果是非安全态则不允许获得该资源（非安全态不一定产生死锁）

## 创建线程的三种方式：

**为什么执行线程要调用Thread.start（）而不是直接调用run方法？**

看一下源码就可以知道，Thread.start()方法中会调用一个**native方法**，这个才是关键，run方法只是普通的Java方法而已，另外Java的线程机制基于来自C的低级的p线程方式

### 1、实现Runnable接口并编写run（）方法

使用Runnable接口一般都是通过匿名内部类的形式，通过Thread类的start()方法启动线程

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53053095" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53053095" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **public** **class** Test {
2. **public** **static** **void** main(String[] args) {
3. **new** Thread(**new** Runnable() {
4. **private** **int** i=10;
5. @Override
6. **public** **void** run() {
7. **while**(i-->0)
8. System.out.print("A ");
9. }
10. }).start();
12. System.out.println("这句话会先打印");//因为启动线程是需要时间的，当主线程和另一个线程同时执行时可能会有先后顺序的问题
14. /\*\*
15. \* 打印结果：
16. \* 这句话会先打印
17. \* A A A A A A A A A A
18. \*/
19. }
20. }

**使用Executor执行线程**，Executor是java.util.concurrent中的接口，用于执行已提交的Runnable任务的对象

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53053095" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53053095" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **class** A **implements** Runnable {
3. @Override
4. **public** **void** run() {
5. System.out.println("A");
6. System.out.println("线程id: " + Thread.currentThread().getId());// 获取当前线程的id
7. }
8. }
10. **public** **class** Test {
11. **public** **static** **void** main(String[] args) {
12. //// 创建线程池，线程数目不固定
13. ExecutorService eService = Executors.newCachedThreadPool();
14. // 创建可重用的数目固定的线程池
15. ExecutorService eService2 = Executors.newFixedThreadPool(2);
16. // 创建单个线程，如果提交了多个任务，那么这些任务将排队
17. ExecutorService eService3 = Executors.newSingleThreadExecutor();//提供了并发机制
18. // 为每个任务都创建一个线程,创建了五个线程
19. **for** (**int** i = 0; i < 5; i++)
20. eService.execute(**new** A());
22. /\*\*
23. \* 如果使用eService2则只启动两个线程去执行任务，因此，线程id只有两个值
24. \* 如果使用eService3则只启动一个线程去执行任务，因此，线程id只有一个值，保证了任务执行的顺序
25. \*/
27. eService.shutdown();// 启动顺序关闭，不接收新任务，如果继续提交
28. // 将抛出java.util.concurrent.RejectedExecutionException异常
30. /\*\*
31. \* 结果： A A 线程id: 14 A 线程id: 10 A A 线程id: 11 线程id: 12 线程id: 13
32. \*
33. \*/
35. }
36. }

### 2、继承Thread类重写run（）方法

其实Thread实现了Runnable接口，所以继承它也相当于是实现了Runnable接口，它的基类：

public class **Thread** extends [Object](http://blog.csdn.net/fuckluy/article/details/53053095" \o "java.lang 中的类" \t "http://blog.csdn.net/fuckluy/article/details/_blank) implements [Runnable](http://blog.csdn.net/fuckluy/article/details/53053095" \o "java.lang 中的接口" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

如果你看一下源码就可以发现，在Thread类的内部有一个Runnable 对象的引用，Runnable target，在构造函数中有一个构造函数的参数是Runnable类型的，参考上面的代码，会将target指向传进来的实例（在构造函数中会调用init（）方法），然后可以看一下基类的run方法，有兴趣可以自己研究一下源码

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53053095" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53053095" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. @Override
2. **public** **void** run() {
3. **if** (target != **null**) {
4. target.run();
5. }
6. }

下面是继承Thread类的Demo:

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53053095" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53053095" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **class** A **extends** Thread{
2. @Override
3. **public** **void** run() {
4. System.out.println("f");
5. **super**.run();
6. }
7. }
9. **public** **class** Test {
10. **public** **static** **void** main(String[] args) {
11. A a=**new** A();
12. a.run();//f
13. }
14. }

### 3、实现Callable接口编写call（）方法

Runnble接口中的run（）方法是没有返回值的，如果需要返回值则应该使用Callable接口，支持泛型，其中call（）方法返回的类型为指定的泛型，使用**call方法要向上抛异常**

Callable的第一种启动方式如下，先将Callable对象封装在FutureTask中，再通过Thread启动线程，然后通过FutureTask的get( )方法获取call( )的返回结果，代码如下

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53053095" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53053095" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **class** A **implements** Callable<String> {
3. **private** **int** id;
5. A(**int** id) {
6. **this**.id = id;
7. }
9. // 返回类型为指定的泛型
10. @Override
11. **public** String call() **throws** Exception {
13. **return** "result of id:"+id;
14. }
16. }
18. **public** **class** Test {
19. **public** **static** **void** main(String[] args) **throws** InterruptedException, ExecutionException {
20. FutureTask<String> f=**new** FutureTask<String>(**new** A(0));
21. **new** Thread(f).start();//启动线程
22. System.out.println(f.get());//获取返回值
23. /\*\*
24. \* 结果：result of id:0
25. \*/
26. }
27. }

第二种方式就是通过Executor方式启动，需要使用submit（）方法，代码如下

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53053095" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53053095" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **class** A **implements** Callable<String> {
3. **private** **int** id;
5. A(**int** id) {
6. **this**.id = id;
7. }
9. // 返回类型为指定的泛型
10. @Override
11. **public** String call() **throws** Exception {
13. **return** "result of id:" + id;
14. }
16. }
18. **public** **class** Test {
19. **public** **static** **void** main(String[] args) **throws** InterruptedException, ExecutionException {
20. ExecutorService service = Executors.newCachedThreadPool();
21. List<Future<String>> result = **new** ArrayList<Future<String>>();
22. **for** (**int** i = 0; i < 5; i++)
23. result.add(service.submit(**new** A(i)));//submit（）方法的返回类型为Future
25. **for** (Future<String> f : result)
26. System.out.println(f.get());
27. service.shutdown();
29. /\*\*
30. \* 结果：
31. \* result of id:0
32. \* result of id:1
33. \* result of id:2
34. \* result of id:3
35. \* result of id:4
36. \*/
37. }
38. }

## ****线程中的一些方法以及其它的知识点****

### 1、sleep（）,wait（），Thread.yield（）的区别

**TimeUnit.MILLISECONDS.sleep**（long timeout）（old style ---->Thread.sleep(long timeout)）：使当前线程进入休眠，将CPU给线程调度器，可以使其他线程有运行机会不管优先级，**不释放锁**，作用于当前线程

**Thread.yield：**Java线程机制的一部分，可以将CPU从一个线程转移到另一个线程，但是只能使同级别的线程获得执行机会，理论上是这样但是实际中随着厂家的性能调优，低优先级的线程也能得到执行机会，**该方法慎用，得不到保障，不释放锁**

**wait()：**会**释放锁**，进入wait（）的线程只能被notify( )或者notifyAll( )唤醒，**wait只能在同步环境中调用**

**下面是Demo：**

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53053095" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53053095" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **class** Sleep **implements** Runnable {
2. @Override
3. **public** **void** run() {
4. Thread.currentThread().setPriority(Thread.MIN\_PRIORITY);// 设置为最低优先级的线程
5. **try** {
6. TimeUnit.MILLISECONDS.sleep(10000);// 休眠10s，old style--->
7. // Thread.sleep(1000)
8. System.out.println("sleep");
10. } **catch** (InterruptedException e) {
11. // TODO Auto-generated catch block
12. e.printStackTrace();
13. }
15. }
16. }
18. **class** Yield **implements** Runnable {
20. @Override
21. **public** **void** run() {
22. Thread.currentThread().setPriority(Thread.MIN\_PRIORITY);// 设置为最低优先级的线程
23. **int** length = 1000;
24. **for** (**int** i = 0; i < length; i++) {
25. **double** d = 0.0;
26. d += (Math.PI + Math.E) / (**double**) i;
27. Thread.yield();
28. **if** (i == length - 1) {
29. System.out.println("yield");
30. }
31. }
33. }
35. }
37. **class** Wait **implements** Runnable {
38. **public** **synchronized** **void** w(**boolean** temp) **throws** InterruptedException {
39. **if** (!temp) {
40. wait();
41. } **else** **if** (temp) {
42. notify();
43. }
44. }
46. // 此方法用于唤醒线程
47. **public** **void** notifyWait() **throws** InterruptedException {
48. w(**true**);
49. }
51. @Override
52. **public** **void** run() {
53. **try** {
54. System.out.println("只会打印一次");
55. w(**false**);
56. // 当此线程被唤醒以后，会从挂起的地方再开始运行
57. System.out.println("wait");
59. } **catch** (InterruptedException e) {
60. // TODO Auto-generated catch block
61. e.printStackTrace();
62. }
63. }
65. }
67. **class** B **implements** Runnable {
68. @Override
69. **public** **void** run() {
70. Thread.currentThread().setPriority(Thread.MAX\_PRIORITY);// 设置为最高优先级的线程
71. System.out.println("B");
73. }
74. }
76. **public** **class** Test {
77. **public** **static** **void** main(String[] args) {
78. ExecutorService service = Executors.newFixedThreadPool(2);
80. // service.execute(new Sleep());
81. // service.execute(new B());
83. /\*\*
84. \* 结果： 先打印 B 会在10s后打印sleep
85. \*/
87. // service.execute(new Yield());
88. // service.execute(new B());
90. /\*\*
91. \* 结果：在我的机器中，如果将yield类中的length设置为1000以下，即使B类的优先级最高但是还是会先执行yeild类
92. \*
93. \*
94. \*/
96. Wait w = **new** Wait();
97. service.execute(w);
98. service.execute(**new** B());
99. **try** {
100. TimeUnit.MILLISECONDS.sleep(10000);// 10s以后唤醒线程,如果不唤醒线程则永远不会打印wait
101. w.notifyWait();
102. ;
103. } **catch** (InterruptedException e) {
104. // TODO Auto-generated catch block
105. e.printStackTrace();
106. }
108. /\*\*
109. \* 结果： 只会打印一次   B  (10s以后打印) wait
110. \*/
112. service.shutdown();
113. }
114. }

**注意：**任何**依赖于线程调度器（Thread.yield）**或**设置线程优先级**来**达到正确性或者性能要求**的程序，很可能都是**不可移植**的

### 2、后台线程

后台线程是程序运行的时候在后台提供的一种通用服务，非必不可少的一部分，且如果所有的非后台线程结束了，程序终止时，所有的该进程中的后台线程也会被杀死

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53053095" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53053095" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **class** B **implements** Runnable {
2. @Override
3. **public** **void** run() {
4. **try** {
5. **while** (**true**) {
6. TimeUnit.MILLISECONDS.sleep(1);// 休息1毫秒
7. System.out.println("BB");
8. }
10. } **catch** (InterruptedException e) {
11. // TODO Auto-generated catch block
12. e.printStackTrace();
13. }
15. }
16. }
18. **public** **class** Test {
19. **public** **static** **void** main(String[] args) **throws** InterruptedException {
20. Thread t=**new** Thread(**new** B());
21. t.setDaemon(**true**);//设置为后台线程,要在start（）前设置
22. t.start();
23. TimeUnit.MILLISECONDS.sleep(100);//休眠0.1s
24. /\*\*
25. \* 打印一些 BB 然后程序终止，因为t是后台线程，当主线程结束时，后台线程也被终止
26. \*/
27. }
28. }

如果一个线程是后台线程，那么它创建的任何线程都将自动的设置为后台线程，且在**后台线程中即使有finally语句也不一定会执行**

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53053095" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53053095" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **class** A **implements** Runnable {
2. @Override
3. **public** **void** run() {
5. **try** {
6. **if** (Thread.currentThread().isDaemon())
7. System.out.println("A   我是后台线程");
8. **else**
9. System.out.println("A   我不是后台线程");
10. TimeUnit.MILLISECONDS.sleep(1000);//1s后执行finally语句
11. } **catch** (InterruptedException e) {
12. e.printStackTrace();
13. } **finally** {
14. System.out.println("我得不到执行，因为主线程休眠只有0.1s，到时候后台线程被杀死");
15. }
17. }
18. }
20. **class** B **implements** Runnable {
21. @Override
22. **public** **void** run() {
23. System.out.println("B   是后台线程");
24. **new** Thread(**new** A()).start();// 启动线程A，被未显示的指定后台线程
25. }
26. }
28. **public** **class** Test {
29. **public** **static** **void** main(String[] args) **throws** InterruptedException {
30. Thread t = **new** Thread(**new** B());
31. t.setDaemon(**true**);// 设置为后台线程,要在start（）前设置
32. t.start();
33. TimeUnit.MILLISECONDS.sleep(100);// 为启动后台线程保留足够的时间
34. /\*\*
35. \* 结果： B 是后台线程 A 我是后台线程
36. \*/
37. }
38. }

### 

### 3、Thread.join( )

一个线程可以在其他线程上调用join()方法，其效果是等待一段时间直到第二线程结束才继续执行，也可以在join（）时附上一个超时参数，这样一旦超时join方法总能返回

**Demo:**[点击打开链接](http://blog.csdn.net/fuckluy/article/details/50755085" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

### 

### 4、线程中的异常

由于线程的本质特性，使得你不能捕获从线程中逃逸的异常

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53053095" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53053095" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **class** A **implements** Runnable {
2. @Override
3. **public** **void** run() {
4. **throw** **new** RuntimeException();// 抛出异常
5. }
6. }
8. **public** **class** Test {
9. **public** **static** **void** main(String[] args) {
10. // 利用try catch 捕获异常
11. ExecutorService service = Executors.newCachedThreadPool();
12. **try** {
13. service.execute(**new** A());
14. } **catch** (RuntimeException e) {
15. System.out.println("异常被捕获");//异常未能被捕获
16. } **finally** {
17. service.shutdown();
18. }
20. }
21. }

为了解决该问题，JAVA 1.5中加入了新接口 Thread.UncaughtExceptionHandler，它允许你在每个Thread对象上都附着一个异常处理器，Thread.UncaughtExceptionHandler.uncaughtException（）会在线程因未捕获的异常而临近死亡时被调用，下面是Demo：

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53053095" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53053095" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. /\*\*
2. \* 自己定义的异常捕获器
3. \*
4. \* @author he
5. \*
6. \*/
7. **class** MyUncaughtException **implements** Thread.UncaughtExceptionHandler {
9. // 会在线程因未捕获的异常而临近死亡时被调用
10. @Override
11. **public** **void** uncaughtException(Thread t, Throwable e) {
12. System.out.println("捕获到异常" + e);
13. }
14. }



19. **class** A **implements** Runnable {
20. @Override
21. **public** **void** run() {
22. **throw** **new** RuntimeException();//抛出异常
23. }
24. }
26. /\*\*
27. \* 如果需要使用Executor执行线程需要使用线程工厂，在里面绑定异常捕获器
28. \* @author he
29. \*
30. \*/
32. **class** myThreadFactory **implements** ThreadFactory{
34. @Override
35. **public** Thread newThread(Runnable r) {
36. Thread t=**new** Thread(r);
37. t.setUncaughtExceptionHandler(**new** MyUncaughtException());//绑定异常捕获器
38. **return** t;
39. }
41. }






49. **public** **class** Test {
50. **public** **static** **void** main(String[] args) {
51. A a=**new** A();
52. Thread thread = **new** Thread(a);
53. thread.setUncaughtExceptionHandler(**new** MyUncaughtException());// 为线程设置异常捕获器
54. thread.start();
55. /\*\*
56. \* 结果:
57. \* 捕获到异常java.lang.RuntimeException
58. \*/



63. /\*\*
64. \* 使用Executor执行线程
65. \*/
66. ExecutorService service=Executors.newCachedThreadPool(**new** myThreadFactory());
67. service.execute(**new** A());
68. service.shutdown();
70. /\*\*
71. \* 结果:
72. \* 捕获到异常java.lang.RuntimeException
73. \*/
75. }
76. }