

day07

线程API

sleep阻塞

线程提供了一个静态方法:

- static void sleep(long ms)
- 使运行该方法的线程进入阻塞状态指定的毫秒,超时后线程会自动回到RUNNABLE状态等待再次获取时间片并发运行.

```
package thread;

public class SleepDemo {
    public static void main(String[] args) {
        System.out.println("程序开始了!");
        try {
            Thread.sleep(5000); //主线程阻塞5秒钟
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("程序结束了!");
    }
}
```

sleep方法处理异常:InterruptedException.

当一个线程调用sleep方法处于睡眠阻塞的过程中,该线程的interrupt()方法被调用时,sleep方法会抛出该异常从而打断睡眠阻塞.

```
package thread;

/**
 * sleep方法要求必须处理中断异常:InterruptedException
 * 当一个线程调用sleep方法处于睡眠阻塞的过程中,它的interrupt()方法
 被调用时
 * 会中断该阻塞,此时sleep方法会抛出该异常。

```

```

*/
public class SleepDemo2 {
    public static void main(String[] args) {
        Thread lin = new Thread(){
            public void run(){
                System.out.println("林:刚美完容，睡一会吧~");
                try {
                    Thread.sleep(9999999);
                } catch (InterruptedException e) {
                    System.out.println("林:干嘛呢!干嘛呢!干嘛
呢!都破了像了!");
                }
                System.out.println("林:醒了");
            }
        };

        Thread huang = new Thread(){
            public void run(){
                System.out.println("黄:大锤80!小锤40!开始砸
墙!");

                for(int i=0;i<5;i++){
                    System.out.println("黄:80!");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                    }
                }
                System.out.println("咣当!");
                System.out.println("黄:大哥，搞定!");
                lin.interrupt();//中断lin的睡眠阻塞
            }
        };
        lin.start();
        huang.start();
    }
}

```

守护线程

守护线程也称为:后台线程

- 守护线程是通过普通线程调用setDaemon(boolean on)方法设置而来的,因此创建上与普通线程无异.
- 守护线程的结束时机上有一点与普通线程不同,即:进程的结束.
- 进程结束:当一个进程中的所有普通线程都结束时,进程就会结束,此时会杀掉所有正在运行的守护线程.

```
package thread;

/**
 * 守护线程
 * 守护线程是通过普通线程调用setDaemon(true)设置而转变的。因此守护
线程创建上
 * 与普通线程无异。
 * 但是结束时机上有一点不同:进程结束。
 * 当一个java进程中的所有普通线程都结束时，该进程就会结束，此时会强制
杀死所有正在
 * 运行的守护线程。
 */
public class DaemonThreadDemo {
    public static void main(String[] args) {
        Thread rose = new Thread(){
            public void run(){
                for(int i=0;i<5;i++){
                    System.out.println("rose:let me go!");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                    }
                }
                System.out.println("rose:啊啊啊啊啊啊啊
AAAAAAAAaaaaa....");
                System.out.println("噗通");
            }
        };

        Thread jack = new Thread(){
```

```

        public void run(){
            while(true){
                System.out.println("jack:you jump!i
jump!");

                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }

            }
        }
    };
    rose.start();
    jack.setDaemon(true);//设置守护线程必须在线程启动前进行
    jack.start();

}
}

```

通常当我们不关心某个线程的任务什么时候停下来,它可以一直运行,但是程序主要的工作都结束时它应当跟着结束时,这样的任务就适合放在守护线程上执行.比如GC就是在守护线程上运行的.

多线程并发安全问题

当多个线程并发操作同一临界资源,由于线程切换时机不确定,导致操作临界资源的顺序出现混乱严重时可能导致系统瘫痪.

临界资源:操作该资源的全过程同时只能被单个线程完成.

```
package thread;
```

```
/**
```

```
 * 多线程并发安全问题
```

```
 * 当多个线程并发操作同一临界资源, 由于线程切换的时机不确定, 导致操作顺序出现
```

```
 * 混乱, 严重时可能导致系统瘫痪。
```

```
 * 临界资源:同时只能被单一线程访问操作过程的资源。
```

```
 */
```

```
public class SyncDemo {
```

```

    public static void main(String[] args) {
        Table table = new Table();
        Thread t1 = new Thread(){
            public void run(){
                while(true){
                    int bean = table.getBean();
                    Thread.yield();

System.out.println(getName()+":"+bean);
                }
            }
        };
        Thread t2 = new Thread(){
            public void run(){
                while(true){
                    int bean = table.getBean();
                    /*
线程
主动放弃本次时间片。
这里使用它的目的是模拟执行到这里CPU没有时间了，发生
线程切换，来看并发安全问题的产生。
*/
                    Thread.yield();

System.out.println(getName()+":"+bean);
                }
            }
        };
        t1.start();
        t2.start();
    }
}

class Table{
    private int beans = 20;//桌子上有20个豆子

    public int getBean(){

```

```

        if(beans==0){
            throw new RuntimeException("没有豆子了!");
        }
        Thread.yield();
        return beans--;
    }
}

```

synchronized关键字

synchronized有两种使用方式

- 在方法上修饰,此时该方法变为一个同步方法
- 同步块,可以更准确的锁定需要排队的代码片段

同步方法

当一个方法使用synchronized修饰后,这个方法称为"同步方法",即:多个线程不能同时 在方法内部执行.只能有先后顺序的一个一个进行. 将并发操作同一临界资源的过程改为同步执行就可以有效的解决并发安全问题.

```

package thread;

/**
 * 多线程并发安全问题
 * 当多个线程并发操作同一临界资源，由于线程切换的时机不确定，导致操作顺序出现
 * 混乱，严重时可能导致系统瘫痪。
 * 临界资源:同时只能被单一线程访问操作过程的资源。
 */
public class SyncDemo {
    public static void main(String[] args) {
        Table table = new Table();
        Thread t1 = new Thread(){
            public void run(){
                while(true){
                    int bean = table.getBean();
                    Thread.yield();

                    System.out.println(getName()+"："+bean);
                }
            }
        };
    }
}

```

```

        }
    };
    Thread t2 = new Thread(){
        public void run(){
            while(true){
                int bean = table.getBean();
                /*
                 * static void yield()
                 * 线程提供的这个静态方法作用是让执行该方法的
                 * 线程
                 * 主动放弃本次时间片。
                 * 这里使用它的目的是模拟执行到这里CPU没有时
                 * 间了，发生
                 * 线程切换，来看并发安全问题的产生。
                 */
                Thread.yield();

                System.out.println(getName()+"："+bean);
            }
        }
    };
    t1.start();
    t2.start();
}
}

```

```

class Table{
    private int beans = 20; //桌子上有20个豆子

    /**
     * 当一个方法使用synchronized修饰后，这个方法称为同步方法，多个
     * 线程不能
     * 同时执行该方法。
     * 将多个线程并发操作临界资源的过程改为同步操作就可以有效的解决多
     * 线程并发
     * 安全问题。
     * 相当于让多个线程从原来的抢着操作改为排队操作。
     */
    public synchronized int getBean(){
        if(beans==0){

```

```

        throw new RuntimeException("没有豆子了!");
    }
    Thread.yield();
    return beans--;
}
}

```

同步块

有效的缩小同步范围可以在保证并发安全的前提下尽可能的提高并发效率。同步块可以更准确的控制需要多个线程排队执行的代码片段。

语法:

```

synchronized(同步监视器对象){
    需要多线程同步执行的代码片段
}

```

同步监视器对象即上锁的对象,要想保证同步块中的代码被多个线程同步运行,则要求多个线程看到的同步监视器对象是同一个.

```

package thread;

/**
 * 有效的缩小同步范围可以在保证并发安全的前提下尽可能提高并发效率。
 *
 * 同步块
 * 语法：
 * synchronized(同步监视器对象){
 *     需要多个线程同步执行的代码片段
 * }
 * 同步块可以更准确的锁定需要多个线程同步执行的代码片段来有效缩小排队范围。
 */
public class SyncDemo2 {
    public static void main(String[] args) {
        Shop shop = new Shop();
        Thread t1 = new Thread(){
            public void run(){
                shop.buy();
            }
        };
    }
}

```



```

        }
    };
    Thread t2 = new Thread(){
        public void run(){
            shop.buy();
        }
    };
    t1.start();
    t2.start();
}
}

class Shop{
    public void buy(){
        /*
            在方法上使用synchronized，那么同步监视器对象就是
            this。
        */
        // public synchronized void buy(){
            Thread t = Thread.currentThread(); //获取运行该方法的
            线程
            try {
                System.out.println(t.getName()+"正在挑衣
            服...");
                Thread.sleep(5000);
            } /*
                使用同步块需要指定同步监视器对象，即：上锁的对象
                这个对象可以是java中任何引用类型的实例，只要保证多
                个需要排队
                执行该同步块中代码的线程看到的该对象是"同一个"即可
            */
            synchronized (this) {
                // synchronized (new Object()) { //没有效果!
                    System.out.println(t.getName() + "正在试衣
            服...");
                    Thread.sleep(5000);
                }

                System.out.println(t.getName()+"结账离开");
            } catch (InterruptedException e) {

```

```

        e.printStackTrace();
    }

}
}

```

在静态方法上使用synchronized

当在静态方法上使用synchronized后,该方法是一个同步方法.由于静态方法所属类,所以一定具有同步效果.

静态方法使用的同步监视器对象为当前类的类对象(Class的实例).

注:类对象会在后期反射知识点介绍.

```

package thread;

/**
 * 静态方法上如果使用synchronized, 则该方法一定具有同步效果。
 */
public class SyncDemo3 {
    public static void main(String[] args) {
        Thread t1 = new Thread(){
            public void run(){
                Boo.dosome();
            }
        };
        Thread t2 = new Thread(){
            public void run(){
                Boo.dosome();
            }
        };
        t1.start();
        t2.start();
    }
}

class Boo{
    /**

```

* **synchronized**在静态方法上使用是，指定的同步监视器对象为当前类的类对象。

* 即：**Class**实例。

* 在**JVM**中，每个被加载的类都有且只有一个**Class**的实例与之对应，后面讲反射

* 知识点的时候会介绍类对象。

*/

```
public synchronized static void dosome(){
    Thread t = Thread.currentThread();
    try {
        System.out.println(t.getName() + ":正在执行
dosome方法...");
        Thread.sleep(5000);
        System.out.println(t.getName() + ":执行
dosome方法完毕!");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

静态方法中使用同步块时,指定的锁对象通常也是当前类的类对象

```
class Boo{
    public static void dosome(){
        /*
            静态方法中使用同步块时，指定同步监视器对象通常还是用当前
            类的类对象
            获取方式为:类名.class
        */
        synchronized (Boo.class) {
            Thread t = Thread.currentThread();
            try {
                System.out.println(t.getName() + ":正在执行
dosome方法...");
                Thread.sleep(5000);
                System.out.println(t.getName() + ":执行
dosome方法完毕!");
            } catch (InterruptedException e) {
```

```

        e.printStackTrace();
    }
}
}
}
}

```

互斥锁

当多个线程执行不同的代码片段,但是这些代码片段之间不能同时运行时就要设置为互斥的.

使用synchronized锁定多个代码片段,并且指定的同步监视器是同一个时,这些代码片段之间就是互斥的.

```

package thread;

/**
 * 互斥锁
 * 当使用synchronized锁定多个不同的代码片段，并且指定的同步监视器对象相同时，
 * 这些代码片段之间就是互斥的，即：多个线程不能同时访问这些方法。
 */
public class SyncDemo4 {
    public static void main(String[] args) {
        Foo foo = new Foo();
        Thread t1 = new Thread(){
            public void run(){
                foo.methodA();
            }
        };
        Thread t2 = new Thread(){
            public void run(){
                foo.methodB();
            }
        };
        t1.start();
        t2.start();
    }
}

class Foo{

```

```
public synchronized void methodA(){
    Thread t = Thread.currentThread();
    try {
        System.out.println(t.getName()+"正在执行A方
法...");
        Thread.sleep(5000);
        System.out.println(t.getName()+"执行A方法完
毕!");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public synchronized void methodB(){
    Thread t = Thread.currentThread();
    try {
        System.out.println(t.getName()+"正在执行B方
法...");
        Thread.sleep(5000);
        System.out.println(t.getName()+"执行B方法完
毕!");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

总结

守护线程与普通线程的区别:守护线程是通过普通线程调用
setDaemon(true)设置而来的

主要区别体现在当java进程中所有的普通线程都结束时进程会结束，在结束
前会杀死所有还在运行的守护线程。

重点:多线程并发安全问题

- 什么是多线程并发安全问题:

当多个线程并发操作同一临界资源，由于线程切换时机不确定，导致执行顺序出现混乱。

解决办法:

将并发操作改为同步操作就可有效的解决多线程并发安全问题

- 同步与异步的概念:同步和异步都是说的多线程的执行方式。

多线程各自执行各自的就是异步执行，而多线程执行出现了先后顺序进行就是同步执行

- synchronized的两种用法

1.直接在方法上声明，此时该方法称为同步方法，同步方法同时只能被一个线程执行

2.同步块，推荐使用。同步块可以更准确的控制需要同步执行的代码片段。

有效的缩小同步范围可以在保证并发安全的前提下提高并发效率

- 同步监视器对象的选取:

对于同步的成员方法而言，同步监视器对象不可指定，只能是this

对于同步的静态方法而言，同步监视器对象也不可指定，只能是类对象

对于同步块而言，需要自行指定同步监视器对象，选取原则:

1.必须是引用类型

2.多个需要同步执行该同步块的线程看到的该对象必须是**同一个**

- 互斥性

当使用多个synchronized修饰了多个代码片段，并且指定的同步监视器都是同一个对象时，这些代码片段就是互斥的，多个线程不能同时在这些代码片段上执行。