线程安全

一、 什么是线程安全

> 《 Java Concurrency In Practice》的作者 Brian Goetz对线程安全有一个比较恰当的定义:
>
> “当多个线程访问一个对象时,如果1.不用考虑这些线程在运行时环境下的调度和交替执行,2.也不需要进行额外的同步,3.或者在调用方进行任何其他的协调操作,调用这个对象的行为都可以获得正确的结果,那这个对象是线程安全的
>
> 这句话的意思是:不管业务中遇到怎样的多个线程访问某对象或某方法的情况,而在编程这个业务逻辑的时候,都不需要额外做任何额外的处理(也就是可以像单线程编程一样),程序也可以正常运行(不会因为多线程而出错),就可以称为线程安全
>
> 相反,如果在编程的时候,需要考虑这些线程在运行时的调度和交替(例如在get()调用到期间不能调用set(),或者需要进行额外的同步(比如使用 synchronized关键字等),那么就是线程不安全的。

二、什么情况下会出现线程安全问题

两种情况：

- 数据争用：比如两个数据他同时去写，造成一放数据要么被丢弃，要么写入错误。
- 竞争条件：主要指的是执行顺序：在没写完之前就去读取，造成顺序上的错误。

运行结果错误：a++的少加问题，同时把减少的位置打出来

活跃性问题：死锁、活锁、饥饿

- 产生死锁代码演示

```
package background;

import java.awt.print.PrinterAbortException;
import java.util.concurrent.TimeUnit;

/**
 * @desc:
 * @author: Mr.Han
 */
public class DeadLock01 implements Runnable{

    private int flag = 0;
    static Object o1 = new Object();
    static Object o2 = new Object();

    public static void main(String[] args) {
        DeadLock01 deadLock01 = new DeadLock01();
        DeadLock01 deadLock02 = new DeadLock01();
        deadLock01.flag = 0;
        deadLock02.flag = 1;
        Thread thread1 = new Thread(deadLock01);
        Thread thread2 = new Thread(deadLock02);
```

```
        thread1.start();
        thread2.start();
    }


    @Override
    public void run() {
        if (flag == 0){
            synchronized (o1){
                try {
                    TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException interruptedException) {
                    interruptedException.printStackTrace();
                }
                synchronized (o2){
                    System.out.println(Thread.currentThread().getName()
+ "获得o2锁");
                }
            }
        } else {
            synchronized (o2){
                try {
                    TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException interruptedException) {
                    interruptedException.printStackTrace();
                }
                synchronized (o1){
                    System.out.println(Thread.currentThread().getName()
+ "获得o1锁");
                }
            }
        }
    }
}
```

对象发布和初始化的时候的安全问题

- 什么是发布：
  - 通过public 、 return 、 或者传递对象的方式把该对象发布了出去
- 什么是逸出：
  1. 方法返回一个 private对象 （private的本意是不让外部访问） —— **返回副本解决**
  2. 还未完成初始化(构造函数没完全执行完毕)就把对象提供给外界（可以**使用工厂模式解决**）,比如:
     1. 在构造函数中未初始化完毕就this赋值
     2. 隐式逸出——注册监听事件
     3. 构造函数中运行线程
- 代码演示

```
package background;

import java.util.HashMap;
import java.util.Map;

/**
 * @desc:
```

```
 * @author: Mr.Han
 */
public class MultiThreadsError03 {
    private Map<String, String> states;

    public MultiThreadsError03() {
        states = new HashMap<>();
        states.put("1", "周一");
        states.put("2", "周二");
        states.put("3", "周三");
        states.put("4", "周四");
    }

    public Map<String, String> getStates() {
        return states;
    }

    public Map<String, String> getStatesImproved() {
        return new HashMap<>(states);
    }

    public static void main(String[] args) {
        MultiThreadsError03 multiThreadsError3 = new
MultiThreadsError03();
        Map<String, String> states = multiThreadsError3.getStates();
        System.out.println(states.get("1"));
        states.remove("1");
        System.out.println(states.get("1"));


    }
}
```

运行结果：

```
"C:\Program Files\Java\jdk1.8.0_231\bin\java.exe" ...
周一
null

Process finished with exit code 0
```

三、如何避免线程安全问题

用副本代替真身

在`MultiThreadsError03`中使用如下方法，返回的时候返回一个新的对象作为副本。

```
public Map<String, String> getStatesImproved() {
        return new HashMap<>(states);
}
```

测试

```
System.out.println(multiThreadsError3.getStatesImproved().get("1"));
multiThreadsError3.getStatesImproved().remove("1");
System.out.println(multiThreadsError3.getStatesImproved().get("1"));
```

虽然删除`remove("1")`过了，但是仍然可以`get("1")`取到值。

使用工厂模式

> 用了工厂模式一旦发布就是完整的对象

```
package background;

/**
 * 描述:       观察者模式
 */
public class MultiThreadsError06 {

    private EventListener eventListener;

    int count;

    public MultiThreadsError06() {
        this.eventListener = new EventListener() {
            @Override
            public void onEvent(Event e) {
                /**最后打印的数字和29行代码sleep()睡眠的时间有关!
                 * 就取决于
                 * mySource.eventCome(new Event() {
                 *
                 * });和new MultiThreadsError05(mySource);的执行顺序了
                 * 如果new MultiThreadsError05(mySource)的构造函数已经执行完了，那么就会输出100，反之则是0
                 */

                System.out.println("\n我得到的数字是" + count);
            }
        };
        for (int i = 0; i < 10000; i++) {
            System.out.print(i);
        }
        count = 100;
    }

    public static MultiThreadsError06 getInstance(MySource mySource){
        MultiThreadsError06 safeListener = new MultiThreadsError06();
        mySource.registerListener(safeListener.eventListener);
        return safeListener;
    }

    public static void main(String[] args) {
        MySource mySource = new MySource();
        getInstance(mySource);

        new Thread(() -> {
            try {
                Thread.sleep(10);
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            mySource.eventCome(new Event() {
            });
        }).start();
    }

    static class MySource {

        private EventListener listener;

        void registerListener(EventListener eventListener) {
            this.listener = eventListener;
        }

        void eventCome(Event e) {
            if (listener != null) {
                listener.onEvent(e);
            } else {
                System.out.println("还未初始化完毕");
            }
        }

    }

    @FunctionalInterface
    interface EventListener {
        void onEvent(Event e);
    }

    interface Event {

    }
}
```

四、各种需要考虑线程安全的情况

- 访问共享的变量和资源，eg：对象的属性、静态变量、共享缓存、数据库等
- 所有依赖时序的动作，即每一步都是线程安全的还是会存在并发问题：read-modify-write、check-then-act
- 不同数据存在绑定关系：ip + 端口号
- 我们使用其他类的时候，如果对方没有声明自己是线程安全的