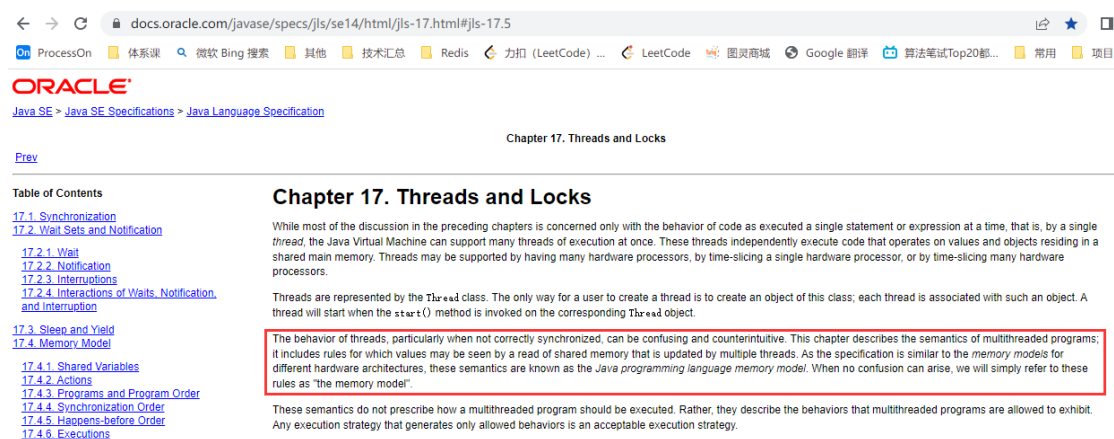


4、一节课学透面试必问并发安全问题

线程安全性

什么是线程安全性？我们可以这么理解，我们所写的代码在并发情况下使用时，总是能表现出正确的行为；反之，未实现线程安全的代码，表现的行为是不可预知的，有可能正确，而绝大多数情况下是错误的。正如 Java 语言规范在《Chapter 17. Threads and Locks》所说的：



图中标红文字的意思是：线程的行为（尤其是在未正确同步的情况下）可能会造成混淆并且违反直觉。本章描述了多线程程序的语义。它包括规则，通过读取多个线程更新的共享内存可以看到值。

如果要想实现线程安全性，就要保证我们的类是线程安全的。在《Java 并发编程实战》中，定义“类是线程安全的”如下：

当多个线程访问某个类时，不管运行时环境采用何种调度方式或者这些线程将如何交替执行，并且在调用代码中不需要任何额外的同步或者协同，这个类都能表现出正确的行为，那么就称这个类是线程安全的。

如何实现呢？

线程封闭

实现好的并发是一件困难的事情，所以很多时候我们都想躲避并发。避免并发最简单的方法就是线程封闭。什么是线程封闭呢？

就是把对象封装到一个线程里，只有这一个线程能看到此对象。那么这个对象就算不是线程安全的也不会出现任何安全问题。

栈封闭

栈封闭是我们编程当中遇到的最多的线程封闭。什么是栈封闭呢？简单的说就是局部变量。多个线程访问一个方法，此方法中的局部变量都会被拷贝一份到

线程栈中。所以局部变量是不被多个线程所共享的，也就不会出现并发问题。所以能用局部变量就别用全局的变量，全局变量容易引起并发问题。

ThreadLocal


ThreadLocal 是实现线程封闭的最好方法。ThreadLocal 内部维护了一个 Map，Map 的 key 是每个线程的名称，而 Map 的值就是我们要封闭的对象。每个线程中的对象都对应着 Map 中一个值，也就是 ThreadLocal 利用 Map 实现了对象的线程封闭。

无状态的类

没有任何成员变量的类，就叫无状态的类，这种类一定是线程安全的。

如果这个类的方法参数中使用了对象，也是线程安全的吗？比如：

```
public class StatelessClass {  
    public int service(int a, int b) {  
        return a + b;  
    }  
  
    public void serviceUser(UserVo user) {  
        //do sth  
    }  
}
```



当然也是，为何？因为多线程下的使用，固然 user 这个对象的实例会不正常，但是对于 StatelessClass 这个类的对象实例来说，它并不持有 UserVo 的对象实例，它自己并不会有问题，有问题的是 UserVo 这个类，而非 StatelessClass 本身。

让类不可变

让状态不可变，加 final 关键字，对于一个类，所有的成员变量应该是私有的，同样的只要有可能，所有的成员变量应该加上 final 关键字，但是加上 final，要注意如果成员变量又是一个对象时，这个对象所对应的类也要是不可变，才能保证整个类是不可变的。

但是要注意，一旦类的成员变量中有对象，上述的 final 关键字保证不可变并不能保证类的安全性，为何？因为在多线程下，虽然对象的引用不可变，但是对象在堆上的实例是有可能被多个线程同时修改的，没有正确处理的情况下，对象实例在堆中的数据是不可预知的。

```
public class ImmutableClass {  
    private final int a;  
    private final String b;  
    private final UserVo user; //加上他就不安全了  
  
    public UserVo getUser() {  
        return user;  
    }  
}
```

加锁和 CAS

我们最常使用的保证线程安全的手段，使用 **synchronized** 关键字，使用显式锁，使用各种原子变量，修改数据时使用 CAS 机制等等。

死锁

概念

是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。此时称系统处于死锁状态或系统产生了死锁。

举个例子：A 和 B 去按摩洗脚，都想在洗脚的时候，同时顺便做个头部按摩，13 技师擅长足底按摩，14 擅长头部按摩。

这个时候 A 先抢到 14，B 先抢到 13，两个人都想同时洗脚和头部按摩，于是就互不相让，扬言我死也不让你，这样的话，A 抢到 14，想要 13，B 抢到 13，想要 14，在这个想同时洗脚和头部按摩的事情上 A 和 B 就产生了死锁。怎么解决这个问题呢？

第一种，假如这个时候，来了个 15，刚好也是擅长头部按摩的，A 又没有两个脑袋，自然就归了 B，于是 B 就美滋滋的洗脚和做头部按摩，剩下 A 在旁边气鼓鼓的，这个时候死锁这种情况就被打破了，不存在了。

第二种，C 出场了，用武力强迫 A 和 B，必须先做洗脚，再头部按摩，这种情况下，A 和 B 谁先抢到 13，谁就可以进行下去，另外一个没抢到的，就等着，这种情况下，也不会产生死锁。

所以总结一下：

1、死锁是必然发生在多操作者 ($M \geq 2$ 个) 争夺多个资源 ($N \geq 2$ 个，且 $N \leq M$) 才会发生这种情况。很明显，单线程自然不会有死锁，只有 B 一个去，不要 2 个，打十个都没问题；单资源呢？只有 13，A 和 B 也只会产生激烈竞争，打得不可开交，谁抢到就是谁的，但不会产生死锁。

2、争夺资源的顺序不对，如果争夺资源的顺序是一样的，也不会产生死锁；

3、争夺者对拿到的资源不放手。

学术化的定义

死锁的发生必须具备以下四个必要条件。

1) 互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。

2) 请求和保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。

3) 不剥夺条件: 指进程已获得的资源, 在未使用完之前, 不能被剥夺, 只能在使用完时由自己释放。

4) 环路等待条件: 指在发生死锁时, 必然存在一个进程——资源的环形链, 即进程集合 $\{P_0, P_1, P_2, \dots, P_n\}$ 中的 P_0 正在等待一个 P_1 占用的资源; P_1 正在等待 P_2 占用的资源, \dots , P_n 正在等待已被 P_0 占用的资源。

理解了死锁的原因, 尤其是产生死锁的四个必要条件, 就可以最大可能地避免、预防和解除死锁。

只要打破四个必要条件之一就能有效预防死锁的发生。

打破互斥条件: 改造独占性资源为虚拟资源, 大部分资源已无法改造。

打破不可抢占条件: 当一进程占有一独占性资源后又申请一独占性资源而无法满足, 则退出原占有的资源。

打破占有且申请条件: 采用资源预先分配策略, 即进程运行前申请全部资源, 满足则运行, 不然就等待, 这样就不会占有且申请。

打破循环等待条件: 实现资源有序分配策略, 对所有设备实现分类编号, 所有进程只能采用按序号递增的形式申请资源。

避免死锁常见的算法有有序资源分配法、银行家算法。

现象、危害和解决

在我们 IT 世界有没有存在死锁的情况, 有: 数据库里多事务而且同时要同时操作多个表的情况下。所以数据库设计的时候就考虑到了检测死锁和从死锁中恢复的机制。比如 oracle 提供了检测和处理死锁的语句, 而 mysql 也提供了“循环依赖检测的机制”

```
Mysql:
频繁报错: Deadlock found when trying to get to lock; try restarting transaction.
1: 查看当前的事务
SELECT * FROM INFORMATION_SCHEMA.INNODB_TRX;
2: 查看当前锁定的事务
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCKS;
3: 查看当前等锁的事务
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCK_WAITS;

杀死进程id (就是上面命令的trx_mysql_thread_id列)
kill 线程ID
```

Oracle

1) 用dba用户执行以下语句

```
select username,lockwait,status,machine,program from v$session where sid  
in (select session id from v$locked object)
```

如果有输出的结果,则说明有死锁,且能看到死锁的机器是哪一台。字段说明:

Username: 死锁语句所用的数据库用户;

Lockwait: 死锁的状态,如果有内容表示被死锁。

Status: 状态, active表示被死锁

Machine: 死锁语句所在的机器。

Program: 产生死锁的语句主要来自哪个应用程序。

2) 用dba用户执行以下语句,可以查看到被死锁的语句

```
select sql_text from v$sql where hash_value in  
(select sql_hash_value from v$session where sid in  
(select session_id from v$locked_object))
```

3) 查找死锁的进程:

```
sqlplus "/as sysdba" (sys/change_on_install)
```

```
SELECT s.username,l.OBJECT_ID,l.SESSION_ID,s.SERIAL#,l.ORACLE_USERNAME,l.OS_USER_NAME,l.PROCESS  
FROM V$LOCKED_OBJECT l,V$SESSION S  
WHERE l.SESSION_ID=S.SID;
```

4) kill掉这个死锁的进程:

```
alter system kill session 'sid,serial#'; (其中sid=l.session_id)
```

在 Java 世界里存在着多线程争夺多个资源,不可避免的存在着死锁。那么我们在编写代码的时候什么情况下会发生呢?

现象

简单顺序死锁

参见代码 `cn.tulingxueyuan.safe.dl.NormalDeadLock`

动态顺序死锁

顾名思义也是和获取锁的顺序有关,但是比较隐蔽,不像简单顺序死锁,往往从代码一眼就看出获取锁的顺序不对。

参见代码 `cn.tulingxueyuan.safe.dl.DynDeadLock`

危害

1、线程不工作了,但是整个程序还是活着的 2、没有任何的异常信息可以供我们检查。3、一旦程序发生了发生了死锁,是没有任何的办法恢复的,只能重启程序,对生产平台的程序来说,这是个很严重的问题。

实际工作中的死锁

时间不定,不是每次必现;一旦出现没有任何异常信息,只知道这个应用的所有业务越来越慢,最后停止服务,无法确定是哪个具体业务导致的问题;测试部门也无法复现,并发量不够。

解决

定位

要解决死锁,当然要先找到死锁,怎么找?

通过 `jps` 查询应用的 id,再通过 `jstack id` 查看应用的锁的持有情况

```

"SubTestThread" #11 prio=5 os_prio=0 tid=0x000000001c559800 nid=0xb3020 waiting for monitor entry [0x000000001d1ae000]
java.lang.Thread.State: BLOCKED (on object monitor)
    at com.xiangxue.ch7.NormalDeadLock.SecondToFisrt(NormalDeadLock.java:34)
    - waiting to lock <0x00000000780ab6378> (a java.lang.Object)
    - locked <0x00000000780ab6388> (a java.lang.Object)
    at com.xiangxue.ch7.NormalDeadLock.access$0(NormalDeadLock.java:28)
    at com.xiangxue.ch7.NormalDeadLock$TestThread.run(NormalDeadLock.java:51)

```

```

"TestDeadLock" #1 prio=5 os_prio=0 tid=0x0000000002f20800 nid=0xb33b0 waiting for monitor entry [0x0000000002e8f000]
java.lang.Thread.State: BLOCKED (on object monitor)
    at com.xiangxue.ch7.NormalDeadLock.fisrtToSecond(NormalDeadLock.java:21)
    - waiting to lock <0x00000000780ab6388> (a java.lang.Object)
    - locked <0x00000000780ab6378> (a java.lang.Object)
    at com.xiangxue.ch7.NormalDeadLock.main(NormalDeadLock.java:63)

```

Found one Java-level deadlock:

```

=====
"SubTestThread":
    waiting to lock monitor 0x000000000301d4c8 (object 0x00000000780ab6378, a java.lang.Object),
    which is held by "TestDeadLock"
"TestDeadLock":
    waiting to lock monitor 0x000000000301bec8 (object 0x00000000780ab6388, a java.lang.Object),
    which is held by "SubTestThread"

```

Java stack information for the threads listed above:

```

=====
"SubTestThread":
    at com.xiangxue.ch7.NormalDeadLock.SecondToFisrt(NormalDeadLock.java:34)
    - waiting to lock <0x00000000780ab6378> (a java.lang.Object)
    - locked <0x00000000780ab6388> (a java.lang.Object)
    at com.xiangxue.ch7.NormalDeadLock.access$0(NormalDeadLock.java:28)
    at com.xiangxue.ch7.NormalDeadLock$TestThread.run(NormalDeadLock.java:51)
"TestDeadLock":
    at com.xiangxue.ch7.NormalDeadLock.fisrtToSecond(NormalDeadLock.java:21)
    - waiting to lock <0x00000000780ab6388> (a java.lang.Object)
    - locked <0x00000000780ab6378> (a java.lang.Object)
    at com.xiangxue.ch7.NormalDeadLock.main(NormalDeadLock.java:63)

```

Found 1 deadlock.

修正

关键是保证拿锁的顺序一致

两种解决方式

- 1、内部通过顺序比较，确定拿锁的顺序；
- 2、采用尝试拿锁的机制。

参见代码 `cn.tulingxueyuan.safe.dl.TryLock` 和 `SafeOperate`

其他安全问题

活锁

两个线程在尝试拿锁的机制中，发生多个线程之间互相谦让，不断发生同一个线程总是拿到同一把锁，在尝试拿另一把锁时因为拿不到，而将本来已经持有的锁释放的过程。

解决办法：每个线程休眠随机数，错开拿锁的时间。

线程饥饿

低优先级的线程，总是拿不到执行时间

线程安全的单例模式

在设计模式中，单例模式是比较常见的一种设计模式，如何实现单例呢？一种比较常见的是双重检查锁定。

双重检查锁定

```
5  * 懒汉式-双重检查
6  */
7  public class SingleDcl {
8      private int a;
9      private User user;
10     private static SingleDcl singleDcl;
11     private SingleDcl() {
12     }
13
14     public static SingleDcl getInstance() {
15         if (singleDcl == null) {
16             synchronized (SingleDcl.class) { // 类锁
17                 if (singleDcl == null) {
18                     singleDcl = new SingleDcl();
19                 }
20             }
21         }
22         return singleDcl;
23     }
24 }
25
26     singleDcl.getUser.getId() ---> NullPointerException
```

对象的域不一定赋值完成了

对象的引用有了

上面的双重检查锁定却存在着线程安全问题，为什么呢？这是因为

`singleDcl = new SingleDcl();`

虽然只有一行代码，但是其实在具体执行的时候有好几步操作：

- 1、JVM 为 `SingleDcl` 的对象实例在内存中分配空间
- 2、进行对象初始化，完成 `new` 操作
- 3、JVM 把这个空间的地址赋给我们的引用 `singleDcl`

因为 JVM 内部的实现原理（指并发相关的重排序等，后面的课程会学到），会产生一种情况，第 3 步会在第 2 步之前执行。

于是在多线程下就会产生问题：A 线程正在 `syn` 同步块中执行 `singleDcl = new SingleDcl()`，此时 B 线程也来执行 `getInstance()`，进行了 `singleDcl == null` 的检查，因为第 3 步会在第 2 步之前执行，B 线程检查发现 `singleDcl` 不为 `null`，会直接拿着 `singleDcl` 实例使用，但是这时 A 线程还在执行对象初始化，这就导致 B 线程拿到的 `singleDcl` 实例可能只初始化了一半，B 线程访问 `singleDcl` 实例中的对象域就很有可能出错。

怎么解决这个问题呢？在前面声明 `singleDcl` 的位置：

```
private static SingleDcl singleDcl;
```

加上 `volatile` 关键字，变成 `private volatile static SingleDcl singleDcl;` 即可。

为何加上 **volatile** 关键字就行了呢,后面的课程在讲述 JMM(Java 内存模型)和 **volatile** 的原理会讲到。

单例模式推荐实现

懒汉式

类初始化模式,也叫延迟占位模式。在单例类的内部由一个私有静态内部类来持有这个单例类的实例。因为在 JVM 中,对类的加载和类初始化,由虚拟机保证线程安全。

```
public class SingleInit {  
    1 usage  
    private SingleInit(){}  
  
    1 usage  
    private static class InstanceHolder{  
        1 usage  
        private static SingleInit instance = new SingleInit();  
    }  
  
    public static SingleInit getInstance(){  
        return InstanceHolder.instance;  
    }  
}
```

延迟占位模式还可以用在多线程下实例域的延迟赋值。

饿汉式

在声明的时候就 new 这个类的实例,或者使用枚举也可以。

```
public class SingleEHan {  
    1 usage  
    private SingleEHan(){}  
    private static SingleEHan singleDcl = new SingleEHan();  
}
```

本文档链接地址:

<http://note.youdao.com/noteshare?id=ce53fb9cb521e6c1cb3bbb1795b745ec&sub=2481250201D44B3E92D59295B7A2C7E2>