Java 并发编程：核心理论

Java并发编程系列：

- Java 并发编程：核心理论
- Java并发编程：Synchronized及其实现原理
- Java并发编程：Synchronized底层优化（轻量级锁、偏向锁）
- Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)
- Java 并发编程：volatile的使用及其原理

并发编程是Java程序员最重要的技能之一，也是最难掌握的一种技能。它要求编程者对计算机最底层的运作原理有深刻的理解，同时要求编程者逻辑清晰、思维缜密，这样才能写出高效、安全、可靠的多线程并发程序。本系列会从线程间协调的方式（wait、notify、notifyAll）、Synchronized及Volatile的本质入手，详细解释JDK为我们提供的每种并发工具和底层实现机制。在此基础上，我们会进一步分析java.util.concurrent包的工具类，包括其使用方式、实现源码及其背后的原理。本文是该系列的第一篇文章，是这系列中最核心的理论部分，之后的文章都会以此为基础来分析和解释。

一、共享性

数据共享性是线程安全的主要原因之一。如果所有的数据只是在线程内有效，那就不存在线程安全性问题，这也是我们在编程的时候经常不需要考虑线程安全的主要原因之一。但是，在多线程编程中，数据共享是不可避免的。最典型的场景是数据库中的数据，为了保证数据的一致性，我们通常需要共享同一个数据库中数据，即使是在主从的情况下，访问的也同一份数据，主从只是为了访问的效率和数据安全，而对同一份数据做的副本。我们现在，通过一个简单的示例来演示多线程下共享数据导致的问题：

代码段一：

```
package com.paddx.test.concurrent;

public class ShareData {
    public static int count = 0;

    public static void main(String[] args) {
        final ShareData data = new ShareData();
        for (int i = 0; i < 10; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        //进入的时候暂停1毫秒，增加并发问题出现的几率
                        Thread.sleep(1);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    for (int j = 0; j < 100; j++) {
                        data.addCount();
                    }
                    System.out.print(count + " ");
                }
            }).start();

        }
        try {
            //主程序暂停3秒，以保证上面的程序执行完成
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("count=" + count);
    }

    public void addCount() {
        count++;
    }
}
```

上述代码的目的是对count进行加一操作，执行1000次，不过这里是通过10个线程来实现的，每个线程执行100次，正常情况下，应该输出1000。不过，如果你运行上面的程序，你会发现结果却不是这样。下面是某次的执行结果（每次运行的结果不一定相同，有时候也

公告

昵称：liuxiaopeng
园龄：2年6个月
粉丝：249
关注：2
+加关注

搜索

找找看

谷歌搜索

最新随笔

1. Spring Boot实战：拦截器与过滤器
2. Spring Boot实战：静态资源处理
3. Spring Boot实战：集成Swagger2
4. Spring Boot实战：Restful API的构建
5. Spring Boot实战：数据库操作
6. Spring Boot实战：逐行释义HelloWorld
7. Java集合类：AbstractCollection源码解析
8. Java集合：整体结构
9. Java 并发编程：volatile的使用及其原理
10. Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)

我的标签

Java(15)
Spring(7)
spring boot(7)
并发编程(4)

可能获取到正确的结果）：

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/bin/java ...
100 200 300 400 500 600 735 853 953 753 count=953

Process finished with exit code 0
```

可以看出，对共享变量操作，在多线程环境下很容易出现各种意想不到的的结果。

二、互斥性

资源互斥是指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。我们通常允许多个线程同时对数据进行读操作，但同一时间内只允许一个线程对数据进行写操作。所以我们通常将锁分为共享锁和排它锁，也叫做读锁和写锁。如果资源不具有互斥性，即使是共享资源，我们也不需要担心线程安全。例如，对于不可变的数据共享，所有线程都只能对其进行读操作，所以不用考虑线程安全问题。但是对共享数据的写操作，一般就需要保证互斥性，上述例子中就是因为没有保证互斥性才导致数据的修改产生问题。Java 中提供多种机制来保证互斥性，最简单的方式是使用Synchronized。现在我们在上面程序中加上Synchronized再执行：

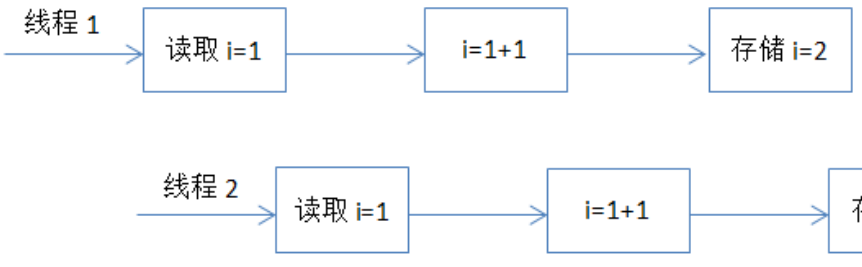
代码段二：

```
package com.paddx.test.concurrent;

public class ShareData {
    public static int count = 0;

    public static void main(String[] args) {
        final ShareData data = new ShareData();
        for (int i = 0; i < 10; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        //进入的时候暂停1毫秒，增加并发问题出现的几率
                        Thread.sleep(1);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    for (int j = 0; j < 100; j++) {
                        data.addCount();
                    }
                    System.out.print(count + " ");
                }
            }).start();

        }
        try {
            //主程序暂停3秒，以保证上面的程序执行完成
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("count=" + count);
    }

    /**
     * 增加 synchronized 关键字
     */
    public synchronized void addCount() {
        count++;
    }
}
```

现在再执行上述代码，会发现无论执行多少次，返回的最终结果都是1000。

三、原子性

原子性就是指对数据的操作是一个独立的、不可分割的整体。换句话说，就是一次操作，是一个连续不可中断的过程，数据不会执行的一半的时候被其他线程所修改。保证原子性的最简单方式是操作系统指令，就是说如果一次操作对应一条操作系统指令，这样肯定可以能保证原子性。但是很多操作不能通过一条指令就完成。例如，对long类型的运算，很多系统就需要分成多条指令分别对高位和低位进行操作才能完成。还比如，我们经常使用的整数 i++ 的操作，其实需要分成三个步骤：（1）读取整数 i 的值；（2）对 i 进行加一操作；（3）将结果写回内存。这个过程在多线程下就可能出现如下现象：



这也是代码段一执行的结果为什么不正确的原因。对于这种组合操作，要保证原子性，最常见的方式是加锁，如Java中的Synchronized或Lock都可以实现，代码段二就是通过Synchronized实现的。除了锁以外，还有一种方式就是CAS（Compare And Swap），即修改数据之前先比较与之前读取到的值是否一致，如果一致，则进行修改，如果不一致则重新执行，这也是乐观锁的实现原理。不过CAS在某些场景下不一定有效，比如另一线程先修改了某个值，然后再改回原来值，这种情况下，CAS是无法判断的。

四、可见性

要理解可见性，需要先对JVM的内存模型有一定的了解，JVM的内存模型与操作系统类似，如图所示：

Rest(2)

Restful(1)

过滤器(1)

集合框架(1)

静态资源(1)

拦截器(1)

更多

随笔档案

2018年1月 (5)

2017年12月 (1)

2016年6月 (1)

2016年5月 (3)

2016年4月 (4)

2016年3月 (3)

积分与排名

积分 - 40237

排名 - 9732

最新评论

1. Re:Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）

博主写得很棒

--还好可以改名字

2. Re:Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）

@就这个名引用在解释轻量级锁的执行过程之前，先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。针对这句话有个疑问，如果我能保.....

--还好可以改名字

3. Re:Java8内存模型—永久代(PermGen)和元空间(Metaspace)

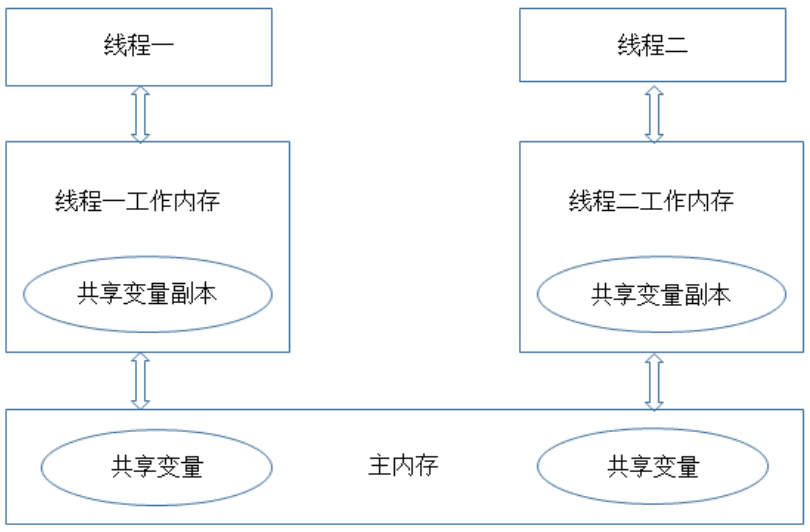
感谢分享，写的很好！

--我不将就

4. Re:Spring Boot实战：集成Swagger2

学习了 菜鸟飘过~~~

--祎祎家斌斌



从这个图中我们可以看出，每个线程都有一个自己的工作内存（相当于CPU高级缓冲区，这么做的目的还是在于进一步缩小存储系统与CPU之间速度的差异，提高性能），对于共享变量，线程每次读取的是工作内存中共享变量的副本，写入的时候也直接修改工作内存中副本的值，然后在某个时间点上再将工作内存与主内存中的值进行同步。这样导致的问题是，如果线程1对某个变量进行了修改，线程2却有可能看不到线程1对共享变量所做的修改。通过下面这段程序我们可以演示一下不可见的问题：

```
package com.paddx.test.concurrent;

public class VisibilityTest {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if (!ready) {
                System.out.println(ready);
            }
            System.out.println(number);
        }
    }

    private static class WriterThread extends Thread {
        public void run() {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            number = 100;
            ready = true;
        }
    }

    public static void main(String[] args) {
        new WriterThread().start();
        new ReaderThread().start();
    }
}
```

从直观上理解，这段程序应该只会输出100，ready的值是不会打印出来的。实际上，如果多次执行上面代码的话，可能会出现多种不同的结果，下面是我运行出来的某两次的结果：

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/bin/java ...
false
100

Process finished with exit code 0
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/bin/java ...
true
100

Process finished with exit code 0
```

当然，这个结果也只能说是有可能是可见性造成的，当写线程（WriterThread）设置ready=true后，读线程（ReaderThread）看不到修改后的结果，所以会打印false，对于第二个结果，也就是执行if (!ready)时还没有读取到写线程的结果，但执行System.out.println(ready)时读取到了写线程执行的结果。不过，这个结果也有可能是线程的交替执行所造成的。Java 中可通过Synchronized或Volatile来保证可见性，具体细节会在后续的文章中分析。

五、有序性

为了提高性能，编译器和处理器可能会对指令做重排序。重排序可以分为三种：

（1）编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。

（2）指令级并行的重排序。现代处理器采用了指令级并行技术（Instruction-Level Parallelism， ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

（3）内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

我们可以直接参考一下JSR 133 中对重排序问题的描述：

5. Re:Spring Boot实战：拦截器与过滤器

这两个使用场景有啥区别？

--四度空间的平面

阅读排行榜

1. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(52581)

2. Java并发编程：Synchronized及其实现原理(43849)

3. Java 并发编程：volatile的使用及其原理(24408)

4. Java 并发编程：核心理论(22450)

5. Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）(21901)

评论排行榜

1. Java并发编程：Synchronized及其实现原理(21)

2. Java 并发编程：volatile的使用及其原理(16)

3. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(13)

4. Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)(12)

5. 通过反编译深入理解Java String及intern(10)

推荐排行榜

1. Java8内存模型—永久代(PermGen)和元空间(Metaspace)(36)

2. Java并发编程：Synchronized及其实现原理(33)

3. 从字节码层面看“HelloWorld”(26)

4. Java 并发编程：核心理论(21)

5. Spring Boot实战：逐行释义HelloWorld(15)

（1）

（2）

先看上图中的（1）源码部分，从源码来看，要么指令 1 先执行要么指令 3先执行。如果指令 1 先执行，r2不应该能看到指令 4 中写入的值。如果指令 3 先执行，r1不应该能看到指令 2 写的值。但是运行结果却可能出现r2==2，r1==1的情况，这就是“重排序”导致的结果。上图（2）即是一种可能出现的合法的编译结果，编译后，指令1和指令2的顺序可能就互换了。因此，才会出现r2==2，r1==1的结果。Java 中也可通过Synchronized或Volatile来保证顺序性。

六 总结

本文对Java 并发编程中的理论基础进行了讲解，有些东西在后续的分析中还会做更详细的讨论，如可见性、顺序性等。后续的文章都会以本章内容作为理论基础来讨论。如果大家能够很好的理解上述内容，相信无论是去理解其他并发编程的文章还是在平时的并发编程的工作中，都能够对大家有很好的帮助。

作者：liuxiaopeng

博客地址：http://www.cnblogs.com/paddix/

声明：转载请在文章页面明显位置给出原文连接。

标签： Java， 并发编程

好文要顶 关注我 收藏该文

liuxiaopeng
关注 - 2
粉丝 - 249
+加关注

21 0

« 上一篇：通过反编译深入理解Java String及intern

» 下一篇：Java并发编程：Synchronized及其实现原理

posted @ 2016-04-12 07:59 liuxiaopeng 阅读(22450) 评论(2) 编辑 收藏

评论列表

#1楼 2017-11-05 18:29 宪贵

感觉这么多人只看不评论，可能是作者太优秀，没有可批评的。多谢作者的贡献！

支持(2) 反对(0)

#2楼 2018-01-27 20:36 笑吧

很赞

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

【推荐】超50万VC++源码: 大型组态工控、电力仿真CAD与GIS源码库！

【活动】2050 大会 - 年青人因科技而团聚（5.26-27杭州·云栖小镇）

【活动】华为云全新一代云服务器·限时特惠5.6折

【推荐】腾讯云多款高规格服务器，免费申请试用6个月

Copyright ©2018 liuxiaopeng