JMM

文档内容索引

==**基础知识和核心知识**==

1. 并发和高并发概念.
2. CPU 多级缓存. (缓存一致性,乱序执行优化)
3. Java 内存模型. (Java 内存模型是高并发的基础. JVM规定,抽象结构,同步操作与规则)
4. 并发优势与风险.
5. 并发模拟. (postMan, JMeter, AB(Apache bence))

==**并发与并发安全(并发的线程安全)**==

1. ==**线程安全性**==: 原子性, 可见性, 有序性, Atomic 包, CAS 算法, synchronized与 Lock, volatile, happens-before
2. ==**安全发布对象**==:安全发布方法, 不可变对象, final 关键字的使用, 不可变方法, 线程不安全类与写法
3. ==**线程安全手段**==: 堆栈封闭, 线程封闭, ThreadLocal 线程封闭, JDBC 线程封闭, 同步容器, 并发容器, J.U.C等
4. ==**AQS 及其他 J.U.C 组件**==: CountDownLatch, Semaphore, CyclicBarrier, ReentrantLock 与锁, Condition, FutureTask,Fork/Join 框架, BlockingQueue
5. ==**线程池**==: new Thread 弊端, 线程池的好处, ThreadPoolExecutor, Executor 框架接口.
6. ==**线程安全补充内容**==: 死锁的产生和预防. 多线程并发最佳实践, Spring 的线程安全, HashMap 和 ConcurrenceHashMap 深入讲解

==**高并发处理的思路和手段**==

1. ==**扩容**==. (水平扩容和垂直扩容)
2. ==**缓存**==. (Redis, Memcache, Guava Cache)
3. ==**队列**==. (Kafka, RabbitMQ, RocketMQ 等队列特性介绍及使用队列的关注点)
4. ==**应用拆分**==. (服务化 Dubbo 与微服务 SpringCloud 介绍)
5. ==**限流**==. (Guava RateLimiter 的介绍与使用, 常用限流算法, 自己实现分布式限流等)
6. ==**服务降级与服务熔断**==. (服务降级的多种选择, Hystrix 介绍与使用等)
7. ==**数据库切库, 分表, 分表**==. (数据库切库, 分表, 支持多数据源的原理及实现)
8. ==**高可用的一些手段**==. (任务调度分布式 elastic-job, 主备 curator 的实现, 监控报警机制等)

==**涉及到的知识技能**==

1. 总体架构. (SpringBoot, Maven, MySQL)
2. 基础组件. (Mybatis, Guava, Lombok,Redis, Kafka)
3. 高级组件. (joda-Time, Atomic 包, J.U.C, AQS, THreadLocal, RateLimiter, Hystrix, threadPool, shardbatis, curator, elastic-job...)

1. 并发底层原理 以及 java 内存模型

1. 底层原理

并发概念介绍

==**并发**==: 同时有两个或者多个线程. 如果运行在单核处理器上, 这多个线程会交替的换入和换入内存, 这些线程是"同时"存在的, 只是每个线程都处于线程中的某个状态. 如果运行在多核处理器上,那么每个线程都会分配一个处理器, 所以可以同时运行.

==**高并发**==: 是一种实现方式, 通过设计可以保证系统可以==**同时并行处理多个请求**==.

==**并发与高并发的对比 :**==

- 并发: 一般说并发的时候, 讨论的是多个线程操作相同的资源, 重点是落在==**线程安全**==上, 合理使用资源.
- 高并发: 在讨论 `high concurrence` 时, 一般是说==**系统可以同时处理很多请求的能力**==.提供性能. 例如天猫双 11, 系统会在短时间内收到大量的请求, 比如资源的访问, 数据库的访问等.

java 程序的编译与运行

1. 编写 .java 源程序
2. 通过 `javac` 命令编译成 .class 文件(字节码文件)
3. JVM 执行 .class 文件, 转化成机器指令.
4. 机器指令可以直接在 CPU 上执行.也就是最终的程序执行.

不同的 JVM 实现会带来不同的翻译, 不同的 CPU 平台的机器指令也是千差万别, 所以我们在编写 java 代码层写的各种 lock, 其实最后依赖的是 JVM 的具体实现(不同版本的 JVM 会有不同的实现)和 CPU 指令, 才能帮助我们达到线程安全的效果.

由于最终依赖处理器, 不同处理器结果不一样, 这样无法保证并发安全, 所以需要有一个标准, 让多线程运行的结果可预期, 这个标准就是 JVM.

2. JVM 内存结构/java 内存模型/java 对象模型

1. 三个概念的总体概述

三个截然不同的概念,比较容易混淆

- JVM 内存结构: 和 ==**java 虚拟机运行时区域**==有关
- java 内存模型: 和 ==**java 的并发编程**==有关.
- java 对象模型: 和 ==**java 对象在虚拟机中的表现形式**==有关.

3. JVM 内存结构

java 代码是运行在虚拟机上的, 虚拟机会把内存分为不同的区域, 每个区域有不同的作用.

4. java 对象模型

- java 对象模型是 ==java 对象自身的存储模型==
- JVM 会给这个类创建一个 ==**instanceKlass**==, 保存在方法区,用来在 JVM 层标识该 java 类
- 当使用 new 创建一个对象的时候, JVM 会在堆中创建一个 ==**instanceOopDesc**== 对象,这个对象中包含 ==对象头== 和 ==实例数据==.

5. JMM(java 内存模型)

1. ==JMM 是规范==

java 内存模型是有 JVM 定义的, 用来屏蔽掉 java 程序在不同的平台和操作系统中内存访问的差异, 实现 ==**java 程序在不同的平台上都能达到内存访问的一致性**==. 也就是说 java 内存模型==**规范**==了一个线程如何和何时可以访问到由其他线程修改过后的共享变量的值, 以及在必须时如何同步的访问共享变量. 总结一句话:==**Java 内存模型的主要目标是定义变量的访问规则.也就是在虚拟机中将变量存储到主内存或者从主内存取出的底层细节.**==

2. ==JMM是工具类和关键字的原理==

- volatile, synchronized, Lock 等关键字的原理都是 JMM

3. JMM 的最重要的 3 个重点

- 重排序

- 可见性
- 原子性

1. 重排序 OutOfOrderExecution

1. 重排序的代码案例, 什么是重排序

```
public class OutOfOrderExecution {

    private static int a = 0, b = 0;
    private static int x = 0, y = 0;

    public static void main(String[] args) throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(1);
        int count = 0;
        for (; ; ) {
            count ++;
            a = 0;
            b = 0;
            x = 0;
            y = 0;

            Thread one = new Thread(() -> {
                try {
                    countDownLatch.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                a = 1; // 1
                x = b; // 2
            });

            Thread two = new Thread(() -> {
                try {
                    countDownLatch.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
```

```
                b = 1; // 3
                y = a; // 4
            });

            one.start();
            two.start();
            countDownLatch.countDown();
            one.join();
            two.join();

            String result = "第" + count + "次, (" + x + "," + y + ")";
            if (x == 0 && y == 0) {
                System.out.println(result);
                break;
            } else {
                System.out.println(result);
            }
        }
    }
}
```

执行顺序:

1 -> 2 -> 3 -> 4 : 结果为 x=0, y=1

3 -> 4 -> 1 -> 2 : 结果为 x=1, y=0

3 -> 1 -> 2 -> 4 : 结果为 x=1, y=1

以上三种结果, 都是在同一个线程中, 代码的执行顺序没有改变, 也就是 one 中, 1 在 2 之前执行, two 中, 3 在 4 之前执行.

如果发生了重排序, 就是以下情况:

2 -> 3 -> 4 -> 1 : 假如执行顺序是这样的, 那么结果 x = 0, y = 0 . 这是发生了重排序的结果.

重排序: 在线程 1 内部的两行代码的实际执行顺序和代码在 java 文件中的顺序不一致, 代码指令并不是严格按照代码语句顺序执行的. 他们的顺序改变了. 这就是重排序.

2. 重排序的好处: 提高处理速度

- 重排序前, 3行代码, 在 CPU 中有 9 行指令. 指令的多少决定着处理速度.
- 重排序后, 将 3 行代码的顺序重排序为右边, 此时 CPU 指令只有 7 行.

3. 重排序的 3 种情况.

1. 编译器优化(JVM, JIT 等) : ==通过修改代码执行顺序达到重排序的目的.==
   - 如果编译器认为上下两行代码没有依赖关系,并且重排会有优化处理速度的时候,就会进行重排序. 比如上图, 就会把 b=2放在后边执行
2. CPU 指令重排 : ==通过修改代码执行顺序达到重排序的目的.==
   - 就算编译器不发生重排, CPU 也可能对指令进行重排.
3. 内存的重排序
   - 其实内存中农不存在重排序, 只是会因为可见性原因, 造成和重排序一样的效果.
   - 由于主内存和线程工作内存的原因, 线程 A 的修改 线程 B 却看不到. 引出可见性问题. 效果和重排序类似.

2. 可见性

1. 主内存和本地内存

- JVM 运行程序的实体是线程.
- 每个线程创建时, JVM 都会为其创建一个==**工作内存(其实并不是真的给每个线程分配一块内存空间, 而是JMM 的一个抽象，对寄存器, 一级缓存, 二级缓存等的抽象.)**==.工作内存是每个线程的==**私有数据区域**==.
- 而 java 内存模型中规定所有变量都存储在==**主内存**==中, ==**主内存是所有线程共享内存区域, 所有线程都可以访问**==.
- 但是线程对变量的操作必须在工作内存中完成, 首先需要从主内存中将变量拷贝到自己的工作内存中, 然后对变量进行操作,操作完后再写回主内存中. 不能直接操作主内存的变量. 各个线程存储这主内存的==**变量的副本拷贝**==.
- 基于以上原因, ==**不同的线程无法访问其他线程的工作内存.线程间的通信必须通过主内存来完成**==.

2. 为什么会有可见性问题

- RAM: 内存(包含 L3 cache).L3cach3, L2cache,L2cache都是缓存. Registers: 寄存器, core: CPU . 将寄存器, L1,L2 抽象为工作内存. L3 和 RAM 是主内存.
- 速度由下而上越来越快
- 之所以在 CPU 和内存中间加上缓存和寄存器, 是因为 CPU 的速度要比内存快的多, 如果不加上缓存和寄存器, 会降低 CPU 的效率.比如 CPU 运行一次需要 1ms, 但是内存运行一次需要 100ms, 如果不加缓存和寄存器, 那么 CPU 运行一次的时间会因为内存的原因, 变为 100ms.
- 对于同一个变量a, core4操作完后, 写回到 L2后, 由于 L2的速度比 L3 快, 还未写回到 L3, 此时 core1 读取 L3 的数据, 读取的是未写回前的数据, 就造成了可见性问题.

==更加准确的说法:==

- CPU 有多级缓存, 导致读的数据过期.
  - 高速缓存的容量比主内存小,但是速度仅次于寄存器, 所以在 CPU 和主内存之间就多了 cache 层.
  - 线程间对于共享变量的可见性问题不是直接由多核引起的, 而是由==**多存缓存**==引起的(就是上边那个图).

==总结==:

- JMM 抽象出了工作内存和主内存的概念.
- 各个线程只能操作自己的工作内存, 而工作内存中的数据是来自于主内存, 操作完后, 吧数据写回到主内存.
- 而工作内存和主内存之间, 由于多级缓存操作速度的原因, 会造成数据的过期(数据不是实时的).
- 这就造成了可见性问题.

3. Happens-before 原则
==**首先, happens-before 规则是为了解决可见性问题的**==

- 在时间上, 如果动作 A 发生在动作 B 之前, B 保证能看到 A.
- 如果一个操作 happens-before 于另一个操作, 那么我们就睡第一个操作对于第二个操作是可见的.

4. 符合 happens-before 原则有哪些?

1. ==**单线程原则**==
   - 在一个线程内, 一定符合 happens-before 原则. 也就是后边的代码一定能看到前边的代码做了什么.
2. ==**锁操作**==
   - 用同一把锁,线程A 和线程 B, 线程 A 获取锁后, 进行操作, 释放锁后, 线程 B 获取到锁后, 肯定能看到线程 A 释放锁之前的操作.
3. ## ==**volatile**==

4. ## 线程启动

5. 线程join()
   - 一旦调用了 join()方法, 后边的线程就能看到前边线程的操作.
6. ## 传递性

7. 中断
    - 一个线程被其他线程 interrupt 时, 那么检测中断 isInterrupt 或者抛出 InterruptException 一定能看到
8. 构造方法
    - 对象构造方法的最后一行指令 happens-before 于 finalize 方法的第一行指令.
9. ==**工具类的 happens-before 原则**==(默认保证)
    - 线程安全的容器 get 一定能看到在此之前的put 等存入动作.
    - CountDownLatch
    - Semaphore
    - Future(默认保证)
    - 线程池(默认保证)
    - CyclicBarrier

5. volatile

1. volatile是什么

- ==**volatile**== 是一种==**同步机制**==.

- 比 Synchronized 或者 lock 更轻量,因为使用 volatile 并不会发生上下文切换等开销很大的行为.

- 因为开销小, 所以做不到 Synchronized 那样的原子保护, volatile 仅在很有限的场景下才能发挥作用.

- 如果一个变量被 volatile 修饰, 那么 JVM 就知道这个变量可能会被并发修改, 就会进行比如 禁止重排序的操作.

2. Volatile 使用场合
==不适用 `a++` ==

```
public class NoVolatile implements Runnable {
    volatile int a = 0;
    AtomicInteger realA = new AtomicInteger(0);

    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new NoVolatile();
```

```
        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);
        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        System.out.println(((NoVolatile)runnable).a);
        System.out.println(((NoVolatile)runnable).realA);
    }

    /** volatile 不适用于 a++ .*/
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            a++;
            realA.incrementAndGet();
        }
    }
}
```

==**适用场合 1:**==

比如一个boolean flag, 如果一个==**共享变量自始至终只被各个线程赋值(并且不依赖自身, 比如不是 a = a + 1, 或者是 boolean 的 b, b = !b, 这种情况就无法保证正确性), 而没有其他的操作**==, 那么就可以使用 volatile 来代替 Synchronized 或者代替原子变量, 因为复制自身是有原子性的, 而 volatile 保证了可见性, 所以就足以保证线程安全.

```
public class UseVolatile implements Runnable {

    volatile boolean done = false;
    AtomicInteger realA = new AtomicInteger(0);

    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new UseVolatile();
        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);
        thread1.start();
```

```
        thread2.start();
        thread1.join();
        thread2.join();
        System.out.println(((UseVolatile)runnable).done);
        System.out.println(((UseVolatile)runnable).realA.get());
    }


    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            setDone();
            realA.incrementAndGet();
        }
    }

    private void setDone() {
        //done = true; // 可以保证, 因为自始至终只被各个线程赋值.并且不依赖自身状态.
        done = !done;    // 无法保证, 因为不是纯粹的复制, 依赖了自己本身的状态.
    }
}
```

==使用场景 2:==

作为==**触发器**==使用.基于使用场景 1, 当 volatile 修饰的变量发生指定变化或者满足状态的时候, 才进行下边的操作.

```
public class UseVolatile2 implements Runnable {

    volatile boolean done = false; // 触发器的作用.
    int a = 1;
    int b = 2;
    int c = 3;
    AtomicInteger realA = new AtomicInteger(0);

    public static void main(String[] args) throws InterruptedException {
        Runnable runnable = new UseVolatile2();
```

```
        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);
        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        if (((UseVolatile2)runnable).done) {
            System.out.println(((UseVolatile2)runnable).a + ", " +
((UseVolatile2)runnable).b + ", " + ((UseVolatile2)runnable).c);
        }
    }

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            change();
        }
    }

   private void change() {
        a = 11;
        b = 22;
        c = 33;
        done = true;
   }
}
```

3.volatile 的作用: 可见性, 禁止重排序

==**可见性**==: 读一个 volatile 变量前, 需要先使相应的本地缓存失败. 这样就必须去 主内存读取最新的值! 写一个 volatile 属性会立即刷新到主内存. 也就是说 volatile 属性不会被线程缓存.

==禁止指令重排序==: 解决单例双重锁乱序问题.

4. volatile 和 Synchronized 的关系

如果一个==**共享变量自始至终只被各个线程赋值(并且不依赖自身, 比如不是 a = a + 1, 或者是 boolean 的 b, b = !b, 这种情况就无法保证正确性), 而没有其他的操作(读取啊, 根据当前的值再重新赋值啊)**==, 那么就可以使用 volatile 来代替 Synchronized 或者代替原子变量, 因为复制自身是有原子性的, 而 volatile 保证了可见性, 所以就足以保证线程安全.

5. 用 volatile 修正重排序问题

6. 总结

- volatile 适用场景: 某个属性被多个线程共享, 某个线程修改了这个属性后, 其他线程可以立即得到修改后的值. 比如 ==**boolean flag**==; 或者作为==**触发器**==
- volatile只能修饰属性, 当使用 volatile 修饰属性后, 编辑器就==**不会对这个属性做指令重排序**==
- volatile 提供了可见性, 任何一个线程对其的修改, 都会立即被其他线程可见. volatile 属性不会被线程缓存, 始终从主内存中获取.
- volatile可以==**保证 long 和 double 的赋值是原子性**==的.

6. 保证可见性的措施.

volatile, Synchronized, lock, join(), start() 等都可以保证可见性.

其实就是符合 happens-before 的那些.

7. 对 synchronized 可见性的正确理解

- 不仅保证了原子性, 还保证了可见性.
- 解锁前的操作都可以被看到.

3. 原子性

1. 什么是原子性

一系列的操作, 要么全部成功, 要么全部失败. 不会出现执行一半的情况. 是不可分割的.

`a++`不是原子性的.

a++其实有三个指令, 这三个指令可以重排序. 可以使用 Synchronized 修饰.

2. java 中原子操作有哪些?

- 除了 long 和 double 之外的基本类型的赋值操作.
- 所有对引用的赋值, 也是原子操作.
- java.concurrent.Atomic.* 包下的所有类都是原子操作.

3. long 和 double 的原子性.

根据官方文档, 对于 64 位的值的写入, 会被分为两次 32 位的写入. 这样的话, 加入第一个写入 32 位后, CPU 资源被其他线程抢走了, 那么其他线程读取到的值就是错误的.

==**在 32 位上的 JVM 上, long 和 double 的操作不是原子性的, 但是在 64 位的 JVM 上, 是原子性的.**==

可以使用 volatile 进行修饰.或者使用 Synchronized 进行同步.

==**实际开发中,商用虚拟机其实已经默认对 long 和 double 的写入进行了原子性操作, 所以实际开发中, 其实可以不用做特殊处理**==

4. 原子操作 + 原子操作 != 原子操作.
简单的原子操作组合在一起, 并不能保证整体依然是原子性.

6. 死锁

2. 并发编程

第三章. 线程安全

==**线程安全**==: 代码所在的进程有多个线程在同时运行, 这些线程可能会同时运行同一段代码, 如果运行结果和单线程环境下运行的结果一样,并且其他变量的值也和预期一样, 那么就是线程安全的. 简单的说就是: ==在并发环境下, 可以得到预期的值, 那么就说是线程安全的.==

当多个线程访问某个类时, 不管运行时环境采用==何用调度方式==或者这些进程如何交替进行,并且在主调代码中==不采用任何额外的同步或协调==, 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==.

==线程安全性体现在三个方面:==

1. ==**原子性**==: 提供了==互斥访问==, 同一时间==只能有一个线程==来对它进行操作.
2. ==**可见性**==: 一个线程对==主内存的修改==可以及时被其他线程观察到.
3. ==**有序性**==: 一个线程观察其他线程中的指令执行顺序, 由于 ==指令重排序== 的存在, 该观察结果一般杂乱无序.

3.1 原子性

3.1.1 Atomic包演示

==Atomic包== : AtomicXXX 都是通过 ==**CAS**== 来完成原子操作的

3.1.1.1 AtomicInteger 演示

```
@Slf4j
@ThreadSafe
public class B_CountExampleAtomicInteger {

    public static int threatTotal = 5000;
    public static int clientTotal = 200;
    public static AtomicInteger count = new AtomicInteger(0);

    public static void main(String[] args) throws Exception {
        ExecutorService executorService = Executors.newCachedThreadPool();
        Semaphore semaphore = new Semaphore(clientTotal);
        CountDownLatch countDownLatch = new CountDownLatch(threatTotal);
        for (int i = 0; i < threatTotal; i++) {
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    add();
                    semaphore.release();
                } catch (InterruptedException e) {
                    log.error("exception", e);
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await();
        executorService.shutdown();
        log.info("count : {}", count.get());
    }

    private static void add() {
        // count 自增 1, 相当于 int a, a++
        log.info("count ++ : {}", count.getAndIncrement());
```

```
        // 也是 count 自增 1, 相当于 int a, ++a
//        count.incrementAndGet();
    }
}
```

3.1.1.1 AtomicLong 和 LongAdder 演示

==**AtomicLong**==

```
@Slf4j
@ThreadSafe
public class B_CountExampleAtomicLong {
    public static int threadTotal = 5000;
    public static int clientTotal = 200;
    public static AtomicLong count = new AtomicLong(0);

    public static void main(String[] args) throws Exception {
        ExecutorService executorService = Executors.newCachedThreadPool();
        Semaphore semaphore = new Semaphore(clientTotal);
        CountDownLatch countDownLatch = new CountDownLatch(threadTotal);
        for (int i = 0; i < threadTotal; i++) {
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    add();
                    semaphore.release();
                } catch (InterruptedException e) {
                    log.error("exception : {}", e);
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await();
        executorService.shutdown();
        log.info("count: {}", count.get());
    }

    private static void add() {
```

```
            count.getAndIncrement();
        }
    }
```

==**LongAdder**==

JDK8新增的`LongAdder`. 核心是: ====

#并发