

多线程(进阶)

常见的锁策略

注意: 接下来讲解的锁策略不仅仅是局限于 Java . 任何和 "锁" 相关的话题, 都可能会涉及到以下内容. 这些特性主要是给锁的实现者来参考的.

普通的程序猿也需要了解一些, 对于合理的使用锁也是有很大帮助的.

乐观锁 vs 悲观锁

悲观锁:

总是假设最坏的情况, 每次去拿数据的时候都认为别人会修改, 所以每次在拿数据的时候都会上锁, 这样别人想拿这个数据就会阻塞直到它拿到锁。

乐观锁:

假设数据一般情况下不会产生并发冲突, 所以在数据进行提交更新的时候, 才会正式对数据是否产生并发冲突进行检测, 如果发现并发冲突了, 则让返回用户错误的信息, 让用户决定如何去做。

举个栗子: 同学 A 和 同学 B 想请教老师一个问题.

同学 A 认为 "老师是比较忙的, 我来问问题, 老师不一定有空解答". 因此同学 A 会先给老师发消息: "老师你忙嘛? 我下午两点能来找你问个问题嘛?" (相当于加锁操作) 得到肯定的答复之后, 才会真的来问问题. 如果得到了否定的答复, 那就等一段时间, 下次再来和老师确定时间. 这个是悲观锁.

同学 B 认为 "老师是比较闲的, 我来问问题, 老师大概率是有空解答的". 因此同学 B 直接就去找老师.(没加锁, 直接访问资源) 如果老师确实比较闲, 那么直接问题就解决了. 如果老师这会确实很忙, 那么同学 B 也不会打扰老师, 就下次再来(虽然没加锁, 但是能识别出数据访问冲突). 这个是乐观锁.

这两种思路不能说谁优谁劣, 而是看当前的场景是否合适.

如果当前老师确实比较忙, 那么使用悲观锁的策略更合适, 使用乐观锁会导致 "白跑很多趟", 耗费额外的资源.

如果当前老师确实比较闲, 那么使用乐观锁的策略更合适, 使用悲观锁会让效率比较低.

Synchronized 初始使用乐观锁策略. 当发现锁竞争比较频繁的时候, 就会自动切换到悲观锁策略.

就好比同学 C 开始认为 "老师比较闲的", 问问题都会直接去找老师.

但是直接来找两次老师之后, 发现老师都挺忙的, 于是下次再来问问题, 就先发个消息问问老师忙不忙, 再决定是否来问问题.

乐观锁的一个重要功能就是要检测出数据是否发生访问冲突. 我们可以引入一个 "版本号" 来解决.

假设我们需要多线程修改 "用户账户余额".

设当前余额为 100. 引入一个版本号 version, 初始值为 1. 并且我们规定 **"提交版本必须大于记录当前版本才能执行更新余额"**

1) 线程 A 此时准备将其读出 (version=1, balance=100) , 线程 B 也读入此信息 (version=1, balance=100) .

线程1工作内存(寄存器)

```
balance=100;
version = 1;
```

线程2工作内存(寄存器)

```
balance=100;
version = 1;
```

```
balance = 100;
version = 1;
```

主内存(内存)

2) 线程 A 操作的过程中并从其帐户余额中扣除 50 (100-50) , 线程 B 从其帐户余额中扣除 20 (100-20) ;

线程1工作内存(寄存器)

```
balance=50;
version = 1;
```

线程2工作内存(寄存器)

```
balance=80;
version = 1;
```

```
balance = 100;
version = 1;
```

主内存(内存)

3) 线程 A 完成修改工作, 将数据版本号加1 (version=2) , 连同帐户扣除后余额 (balance=50) , 写回到内存中;

线程1工作内存(寄存器)

线程2工作内存(寄存器)

```
balance=50;
version = 2;
```

```
balance=80;
version = 1;
```

```
balance = 50;
version = 2;
```

主内存(内存)

4) 线程 B 完成了操作，也将版本号加1（version=2）试图向内存中提交数据（balance=80），但此时比对版本发现，操作员 B 提交的数据版本号为 2，数据库记录的当前版本也为 2，不满足“**提交版本必须大于记录当前版本才能执行更新**”的乐观锁策略。就认为这次操作失败。

线程1工作内存(寄存器)

线程2工作内存(寄存器)

```
balance=50;
version = 2;
```

```
balance=80;
version = 2;
```

版本不符合要求
写入失败~

```
balance = 50;
version = 2;
```

主内存(内存)

读写锁

多线程之间，数据的读取方之间不会产生线程安全问题，但数据的写入方互相之间以及和读者之间都需要进行互斥。如果两种场景下都用同一个锁，就会产生极大的性能损耗。所以读写锁因此而产生。

读写锁（readers-writer lock），看英文可以顾名思义，在执行加锁操作时需要额外表明读写意图，复数读者之间并不互斥，而写者则要求与任何人互斥。

一个线程对于数据的访问, 主要存在两种操作: 读数据 和 写数据.

- 两个线程都只是读一个数据, 此时并没有线程安全问题. 直接并发的读取即可.
- 两个线程都要写一个数据, 有线程安全问题.
- 一个线程读另外一个线程写, 也有线程安全问题.

读写锁就是把读操作和写操作区分对待. Java 标准库提供了 `ReentrantReadWriteLock` 类, 实现了读写锁.

- `ReentrantReadWriteLock.ReadLock` 类表示一个读锁. 这个对象提供了 `lock / unlock` 方法进行加锁解锁.
- `ReentrantReadWriteLock.WriteLock` 类表示一个写锁. 这个对象也提供了 `lock / unlock` 方法进行加锁解锁.

其中,

- 读加锁和读加锁之间, 不互斥.
- 写加锁和写加锁之间, 互斥.
- 读加锁和写加锁之间, 互斥.

注意, 只要是涉及到 "互斥", 就会产生线程的挂起等待. 一旦线程挂起, 再次被唤醒就不知道隔了多久了.

因此尽可能减少 "互斥" 的机会, 就是提高效率的重要途径.

读写锁特别适合于 "频繁读, 不频繁写" 的场景中. (这样的场景其实也是非常广泛存在的).

比如比特的教务系统.

每节课老师都要使用教务系统点名, 点名就需要查看班级的同学列表(读操作). 这个操作可能要每周执行好几次.

而什么时候修改同学列表呢(写操作)? 就新同学加入的时候. 可能一个月都不必改一次.

再比如, 同学们使用教务系统查看作业(读操作), 一个班级的同学很多, 读操作一天就要进行几十上百次.

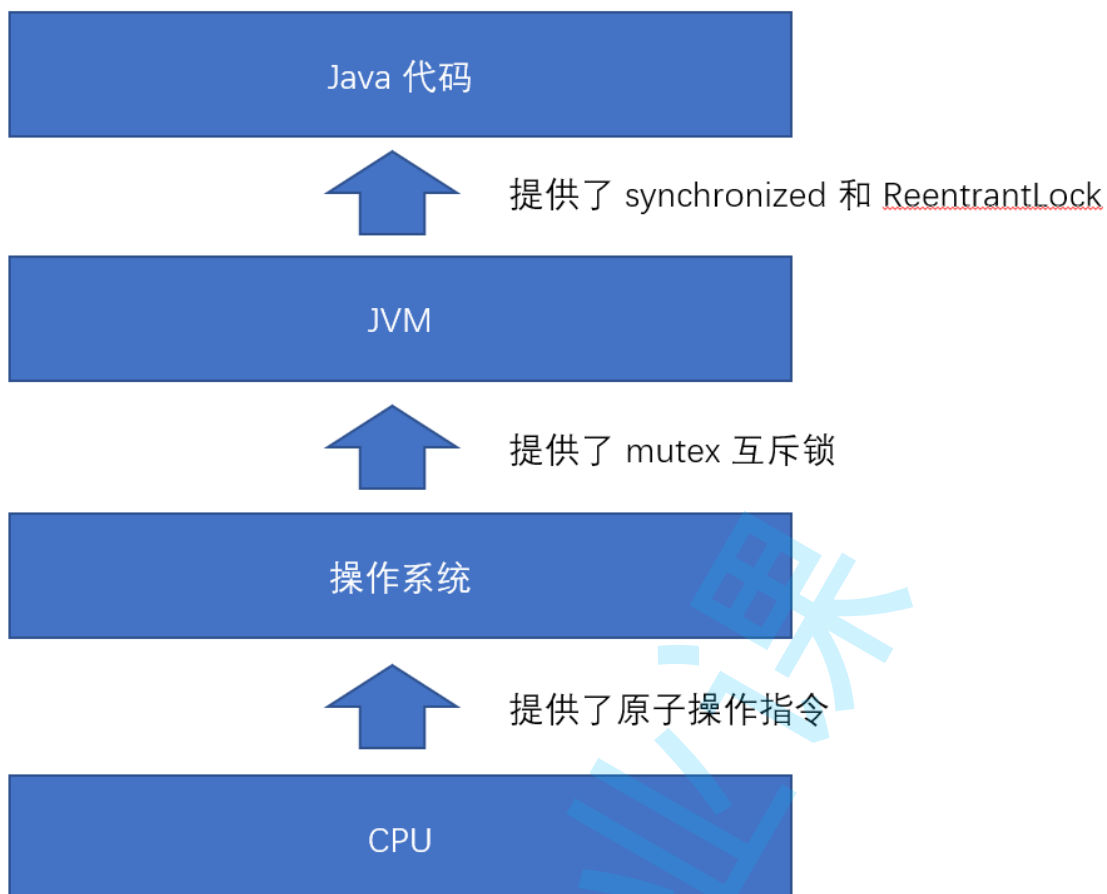
但是这一节课的作业, 老师只是布置了一次(写操作)

Synchronized 不是读写锁.

重量级锁 vs 轻量级锁

锁的核心特性 "原子性", 这样的机制追根溯源是 CPU 这样的硬件设备提供的.

- CPU 提供了 "原子操作指令".
- 操作系统基于 CPU 的原子指令, 实现了 `mutex` 互斥锁.
- JVM 基于操作系统提供的互斥锁, 实现了 `synchronized` 和 `ReentrantLock` 等关键字和类.



注意, `synchronized` 并不仅仅是对 `mutex` 进行封装, 在 `synchronized` 内部还做了很多其他的工作

重量级锁: 加锁机制重度依赖了 OS 提供了 `mutex`

- 大量的内核态用户态切换
- 很容易引发线程的调度

这两个操作, 成本比较高. 一旦涉及到用户态和内核态的切换, 就意味着 "沧海桑田".

轻量级锁: 加锁机制尽可能不使用 `mutex`, 而是尽量在用户态代码完成. 实在搞不定了, 再使用 `mutex`.

- 少量的内核态用户态切换.
- 不太容易引发线程调度.

理解用户态 vs 内核态

想象去银行办业务.

在窗口外, 自己做, 这是用户态. 用户态的时间成本是比较可控的.

在窗口内, 工作人员做, 这是内核态. 内核态的时间成本是不太可控的.

如果办业务的时候反复和工作人员沟通, 还需要重新排队, 这时效率是很低的.

`synchronized` 开始是一个轻量级锁. 如果锁冲突比较严重, 就会变成重量级锁.

自旋锁 (Spin Lock)

按之前的方式, 线程在抢锁失败后进入阻塞状态, 放弃 CPU, 需要过很久才能再次被调度.

但实际上, 大部分情况下, 虽然当前抢锁失败, 但过不了很久, 锁就会被释放. 没必要就放弃 CPU. 这个时候就可以使用自旋锁来处理这样的问题.

自旋锁伪代码:

```
while (抢锁(lock) == 失败) {}
```

如果获取锁失败, 立即再尝试获取锁, 无限循环, 直到获取到锁为止. 第一次获取锁失败, 第二次的尝试会在极短的时间内到来.

一旦锁被其他线程释放, 就能第一时间获取到锁.

理解自旋锁 vs 挂起等待锁

想象一下, 去追求一个女神. 当男生向女神表白后, 女神说: 你是个好人, 但是我有男朋友了~~

挂起等待锁: 陷入沉沦不能自拔.... 过了很久很久之后, 突然女神发来消息, "咱俩要不试试?" (注意, 这个很长的时间间隔里, 女神可能已经换了好几个男票了).

自旋锁: 死皮赖脸坚韧不拔. 仍然每天持续的和女神说早安晚安. 一旦女神和上一任分手, 那么就能立刻抓住机会上位.

自旋锁是一种典型的 轻量级锁 的实现方式.

- 优点: 没有放弃 CPU, 不涉及线程阻塞和调度, 一旦锁被释放, 就能第一时间获取到锁.
- 缺点: 如果锁被其他线程持有的时间比较久, 那么就会持续的消耗 CPU 资源. (而挂起等待的时候是不消耗 CPU 的).

synchronized 中的轻量级锁策略大概率就是通过自旋锁的方式实现的.

公平锁 vs 非公平锁

假设三个线程 A, B, C. A 先尝试获取锁, 获取成功. 然后 B 再尝试获取锁, 获取失败, 阻塞等待; 然后 C 也尝试获取锁, C 也获取失败, 也阻塞等待.

当线程 A 释放锁的时候, 会发生啥呢?

公平锁: 遵守 "先来后到". B 比 C 先来的. 当 A 释放锁的之后, B 就能先于 C 获取到锁.

非公平锁: 不遵守 "先来后到". B 和 C 都有可能获取到锁.

这就好比一群男生追同一个女神. 当女神和前任分手之后, 先来追女神的男生上位, 这就是公平锁; 如果是女神不按先后顺序挑一个自己看的顺眼的, 就是非公平锁.

女神



和男票恩爱中

我追女神
1年了



我追女神
1个月了



我昨天开始
追的女神



公平锁:

女神



我失恋了 T-T
快来安慰我~~

我最先追女神的,
兄弟们我去了哈~



祝老哥早日分手~



不公平锁:

我最先追女神的,
兄弟们我去了哈~



女神



我失恋了 T-T
快来安慰我~~

快放开那个
女神, 让我来



女神早对你俩没兴
趣了, 还是看我的吧



注意:

- 操作系统内部的线程调度就可以视为是随机的. 如果不做任何额外的限制, 锁就是非公平锁. 如果要想实现公平锁, 就需要依赖**额外的数据结构**, 来记录线程们的先后顺序.
- 公平锁和非公平锁没有好坏之分, 关键还是看适用场景.

synchronized 是非公平锁.

可重入锁 vs 不可重入锁

可重入锁的字面意思是“可以重新进入的锁”, 即**允许同一个线程多次获取同一把锁**.

比如一个递归函数里有加锁操作, 递归过程中这个锁会阻塞自己吗? 如果不会, 那么这个锁就是**可重入锁** (因为这个原因可重入锁也叫做**递归锁**).

Java里只要以Reentrant开头命名的锁都是可重入锁, 而且**JDK提供的所有现成的Lock实现类, 包括synchronized关键字锁都是可重入的**.

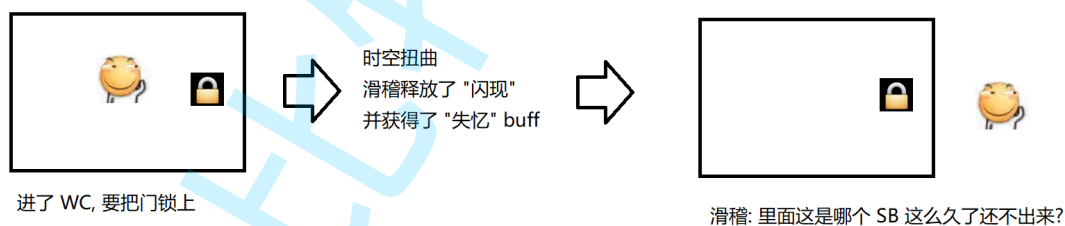
而 Linux 系统提供的 mutex 是不可重入锁.

理解 "把自己锁死"

一个线程没有释放锁, 然后又尝试再次加锁.

```
// 第一次加锁, 加锁成功
lock();
// 第二次加锁, 锁已经被占用, 阻塞等待.
lock();
```

按照之前对于锁的设定, 第二次加锁的时候, 就会**阻塞等待**. 直到第一次的锁被释放, 才能获取到第二个锁. 但是释放第一个锁也是由该线程来完成, 结果这个线程已经躺平了, 啥都不想干了, 也就无法进行解锁操作. 这时候就会 **死锁**.



这样的锁称为 **不可重入锁**.

synchronized 是可重入锁

相关面试题

1) 你是怎么理解乐观锁和悲观锁的, 具体怎么实现呢?

悲观锁认为多个线程访问同一个共享变量冲突的概率较大, 会在每次访问共享变量之前都去真正加锁.

乐观锁认为多个线程访问同一个共享变量冲突的概率不大, 并不会真的加锁, 而是直接尝试访问数据. 在访问的同时识别当前的数据是否出现访问冲突.

悲观锁的实现就是先加锁(比如借助操作系统提供的 mutex), 获取到锁再操作数据. 获取不到锁就等待.

乐观锁的实现可以引入一个版本号. 借助版本号识别出当前的数据访问是否冲突. (实现细节参考上面的图).

2) 介绍下读写锁?

读写锁就是把读操作和写操作分别进行加锁.

读锁和读锁之间不互斥.

写锁和写锁之间互斥.

写锁和读锁之间互斥.

读写锁最主要用在 "频繁读, 不频繁写" 的场景中.

3) 什么是自旋锁, 为什么要使用自旋锁策略呢, 缺点是什么?

如果获取锁失败, 立即再尝试获取锁, 无限循环, 直到获取到锁为止. 第一次获取锁失败, 第二次的尝试会在极短的时间内到来. 一旦锁被其他线程释放, 就能第一时间获取到锁.

相比于挂起等待锁,

优点: 没有放弃 CPU 资源, 一旦锁被释放就能第一时间获取到锁, 更高效. 在锁持有时间比较短的场景下非常有用.

缺点: 如果锁的持有时间较长, 就会浪费 CPU 资源.

4) synchronized 是可重入锁么?

是可重入锁.

可重入锁指的就是连续两次加锁不会导致死锁.

实现的方式是在锁中记录该锁持有的线程身份, 以及一个计数器(记录加锁次数). 如果发现当前加锁的线程就是持有锁的线程, 则直接计数自增.

CAS

什么是 CAS

CAS: 全称 Compare and swap, 字面意思: "比较并交换", 一个 CAS 涉及到以下操作:

我们假设内存中的原数据 V, 旧的预期值 A, 需要修改的新值 B.

1. 比较 A 与 V 是否相等。(比较)
2. 如果比较相等, 将 B 写入 V。(交换)
3. 返回操作是否成功。

CAS 伪代码

下面写的代码不是原子的, 真实的 CAS 是一个原子的硬件指令完成的. 这个伪代码只是辅助理解 CAS 的工作流程.

```
boolean CAS(address, expectValue, swapValue) {
    if (&address == expectedValue) {
        &address = swapValue;
        return true;
    }
    return false;
}
```

两种典型的不是 "原子性" 的代码

1. check and set (if 判定然后设定值) [上面的 CAS 伪代码就是这种形式]
2. read and update (i++) [之前我们讲线程安全的代码例子是这种形式]

当多个线程同时对某个资源进行CAS操作, 只能有一个线程操作成功, 但是并不会阻塞其他线程, 其他线程只会收到操作失败的信号。

CAS 可以视为是一种乐观锁. (或者可以理解成 CAS 是乐观锁的一种实现方式)

CAS 是怎么实现的

针对不同的操作系统, JVM 用到了不同的 CAS 实现原理, 简单来讲:

- java 的 CAS 利用的是 unsafe 这个类提供的 CAS 操作;
- unsafe 的 CAS 依赖的是 jvm 针对不同的操作系统实现的 Atomic::cmpxchg;
- Atomic::cmpxchg 的实现使用了汇编的 CAS 操作, 并使用 cpu 硬件提供的 lock 机制保证其原子性。

简而言之, 是因为**硬件予以了支持, 软件层面才能做到。**

CAS 有哪些应用

1) 实现原子类

标准库中提供了 `java.util.concurrent.atomic` 包, 里面的类都是基于这种方式来实现的。

典型的的就是 AtomicInteger 类. 其中的 getAndIncrement 相当于 i++ 操作。

```
AtomicInteger atomicInteger = new AtomicInteger(0);
// 相当于 i++
atomicInteger.getAndIncrement();
```

伪代码实现:

```
class AtomicInteger {
    private int value;

    public int getAndIncrement() {
        int oldValue = value;
        while ( CAS(value, oldValue, oldValue+1) != true) {
            oldValue = value;
        }
        return oldValue;
    }
}
```

假设两个线程同时调用 getAndIncrement

1) 两个线程都读取 value 的值到 oldValue 中. (oldValue 是一个局部变量, 在栈上. 每个线程有自己的栈)

线程1的栈内存

oldValue = 0

线程2的栈内存

oldValue = 0

value = 0

主内存(内存)

2) 线程1 先执行 CAS 操作. 由于 oldValue 和 value 的值相同, 直接进行对 value 赋值.

注意:

- CAS 是直接读写内存的, 而不是操作寄存器.
- CAS 的读内存, 比较, 写内存操作是一条硬件指令, 是原子的.

线程1的栈内存

oldValue = 0

线程2的栈内存

oldValue = 0

value = 1

主内存(内存)

3) 线程2 再执行 CAS 操作, 第一次 CAS 的时候发现 `oldValue` 和 `value` 不相等, 不能进行赋值. 因此需要进入循环.

在循环里重新读取 `value` 的值赋给 `oldValue`

线程1的栈内存

`oldValue = 0`

线程2的栈内存

`oldValue = 1`

`value = 1`

主内存(内存)

4) 线程2 接下来第二次执行 CAS, 此时 `oldValue` 和 `value` 相同, 于是直接执行赋值操作.

线程1的栈内存

`oldValue = 0`

线程2的栈内存

`oldValue = 1`

`value = 2`

主内存(内存)

5) 线程1 和 线程2 返回各自的 `oldValue` 的值即可.

通过形如上述代码就可以实现一个原子类. 不需要使用重量级锁, 就可以高效的完成多线程的自增操作.

本来 `check and set` 这样的操作在代码角度不是原子的. 但是在硬件层面上可以让一条指令完成这个操作, 也就变成原子的了.

2) 实现自旋锁

基于 CAS 实现更灵活的锁, 获取到更多的控制权.

自旋锁伪代码

```
public class SpinLock {
    private Thread owner = null;

    public void lock(){
        // 通过 CAS 看当前锁是否被某个线程持有。
        // 如果这个锁已经被别的线程持有，那么就自旋等待。
        // 如果这个锁没有被别的线程持有，那么就把 owner 设为当前尝试加锁的线程。
        while(!CAS(this.owner, null, Thread.currentThread())){
        }
    }

    public void unlock (){
        this.owner = null;
    }
}
```

CAS 的 ABA 问题

什么是 ABA 问题

ABA 的问题:

假设存在两个线程 t1 和 t2. 有一个共享变量 num, 初始值为 A.

接下来, 线程 t1 想使用 CAS 把 num 值改成 Z, 那么就需要

- 先读取 num 的值, 记录到 oldNum 变量中.
- 使用 CAS 判定当前 num 的值是否为 A, 如果为 A, 就修改成 Z.

但是, 在 t1 执行这两个操作之间, t2 线程可能把 num 的值从 A 改成了 B, 又从 B 改成了 A

线程 t1 的 CAS 是期望 num 不变就修改. 但是 num 的值已经被 t2 给改了. 只不过又改成 A 了. 这个时候 t1 究竟是否要更新 num 的值为 Z 呢?

到这一步, t1 线程无法区分当前这个变量始终是 A, 还是经历了一个变化过程.



这就好比, 我们买一个手机, 无法判定这个手机是刚出厂的新手机, 还是别人用旧了, 又翻新过的手机.

ABA 问题引来的 BUG

大部分的情况下, t2 线程这样的一个反复横跳改动, 对于 t1 是否修改 num 是没有影响的. 但是不排除一些特殊情况.

假设 滑稽老哥 有 100 存款. 滑稽想从 ATM 取 50 块钱. 取款机创建了两个线程, 并发的来执行 -50 操作.

我们期望一个线程执行 -50 成功, 另一个线程 -50 失败.

如果使用 CAS 的方式来完成这个扣款过程就可能出现问题.

正常的过程

- 1) 存款 100. 线程1 获取到当前存款值为 100, 期望更新为 50; 线程2 获取到当前存款值为 100, 期望更新为 50.
- 2) 线程1 执行扣款成功, 存款被改成 50. 线程2 阻塞等待中.
- 3) 轮到线程2 执行了, 发现当前存款为 50, 和之前读到的 100 不相同, 执行失败.

异常的过程

- 1) 存款 100. 线程1 获取到当前存款值为 100, 期望更新为 50; 线程2 获取到当前存款值为 100, 期望更新为 50.
- 2) 线程1 执行扣款成功, 存款被改成 50. 线程2 阻塞等待中.
- 3) 在线程2 执行之前, 滑稽的朋友正好给滑稽转账 50, 账户余额变成 100 !!
- 4) 轮到线程2 执行了, 发现当前存款为 100, 和之前读到的 100 相同, 再次执行扣款操作

这个时候, 扣款操作被执行了两次!!! 都是 ABA 问题搞的鬼!!

解决方案

给要修改的值, 引入版本号. 在 CAS 比较数据当前值和旧值的同时, 也要比较版本号是否符合预期.

- CAS 操作在读取旧值的同时, 也要读取版本号.
- 真正修改的时候,
 - 如果当前版本号和读到的版本号相同, 则修改数据, 并把版本号 + 1.
 - 如果当前版本号高于读到的版本号. 就操作失败(认为数据已经被修改过了).

这就好比, 判定这个手机是否是翻新机, 那么就需要收集每个手机的数据, 第一次挂在电商网站上的手机记为版本1, 以后每次这个手机出现在电商网站上, 就把版本号进行递增. 这样如果买家不在意这是翻新机, 就买. 如果买家在意, 就可以直接略过.

对比理解上面的转账例子

假设 滑稽老哥 有 100 存款. 滑稽想从 ATM 取 50 块钱. 取款机创建了两个线程, 并发的来执行 -50 操作.

我们期望一个线程执行 -50 成功, 另一个线程 -50 失败.

为了解决 ABA 问题, 给余额搭配一个版本号, 初始设为 1.

- 1) 存款 100. 线程1 获取到 存款值为 100, 版本号为 1, 期望更新为 50; 线程2 获取到存款值为 100, 版本号为 1, 期望更新为 50.
- 2) 线程1 执行扣款成功, 存款被改成 50, 版本号改为2. 线程2 阻塞等待中.
- 3) 在线程2 执行之前, 滑稽的朋友正好给滑稽转账 50, 账户余额变成 100, 版本号变成3.
- 4) 轮到线程2 执行了, 发现当前存款为 100, 和之前读到的 100 相同, 但是当前版本号为 3, 之前读到的版本号为 1, 版本小于当前版本, 认为操作失败.

在 Java 标准库中提供了 `AtomicStampedReference<E>` 类. 这个类可以对某个类进行包装, 在内部就提供了上面描述的版本管理功能.

关于 `AtomicStampedReference<E>` 的具体用法此处不再展开. 有需要的同学自行查找文档了解使用方法即可.

相关面试题

1) 讲解下你自己理解的 CAS 机制

全称 Compare and swap, 即 "比较并交换". 相当于通过一个原子的操作, 同时完成 "读取内存, 比较是否相等, 修改内存" 这三个步骤. 本质上需要 CPU 指令的支撑.

2) ABA问题怎么解决?

给要修改的数据引入版本号. 在 CAS 比较数据当前值和旧值的同时, 也要比较版本号是否符合预期. 如果发现当前版本号和之前读到的版本号一致, 就真正执行修改操作, 并让版本号自增; 如果发现当前版本号比之前读到的版本号大, 就认为操作失败.

Synchronized 原理

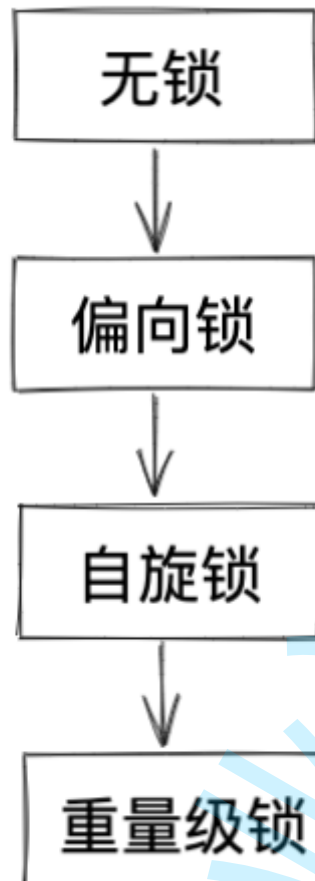
基本特点

结合上面的锁策略, 我们就可以总结出, Synchronized 具有以下特性(只考虑 JDK 1.8):

1. 开始时是乐观锁, 如果锁冲突频繁, 就转换为悲观锁.
2. 开始是轻量级锁实现, 如果锁被持有的时间较长, 就转换成重量级锁.
3. 实现轻量级锁的时候大概率用到的自旋锁策略
4. 是一种不公平锁
5. 是一种可重入锁
6. 不是读写锁

加锁工作过程

JVM 将 synchronized 锁分为 无锁、偏向锁、轻量级锁、重量级锁 状态. 会根据情况, 进行依次升级.



1) 偏向锁

第一个尝试加锁的线程, 优先进入偏向锁状态.

偏向锁不是真的 "加锁", 只是给对象头中做一个 "偏向锁的标记", 记录这个锁属于哪个线程.

如果后续没有其他线程来竞争该锁, 那么就不用进行其他同步操作了(避免了加锁解锁的开销)

如果后续有其他线程来竞争该锁(刚才已经在锁对象中记录了当前锁属于哪个线程了, 很容易识别当前申请锁的线程是不是之前记录的线程), 那就取消原来的偏向锁状态, 进入一般的轻量级锁状态.

偏向锁本质上相当于 "延迟加锁". 能不加锁就不加锁, 尽量来避免不必要的加锁开销.

但是该做的标记还是得做的, 否则无法区分何时需要真正加锁.

举个栗子理解偏向锁

假设男主是一个锁, 女主是一个线程. 如果只有这一个线程来使用这个锁, 那么男主女主即使不领证结婚(避免了高成本操作), 也可以一直幸福的生活下去.

但是女配出现了, 也尝试竞争男主, 此时不管领证结婚这个操作成本多高, 女主也势必要把这个动作完成了, 让女配死心.

2) 轻量级锁

随着其他线程进入竞争, 偏向锁状态被消除, 进入轻量级锁状态(自适应的自旋锁).

此处的轻量级锁就是通过 CAS 来实现.

- 通过 CAS 检查并更新一块内存 (比如 null => 该线程引用)
- 如果更新成功, 则认为加锁成功
- 如果更新失败, 则认为锁被占用, 继续自旋式的等待(并不放弃 CPU).

自旋操作是一直让 CPU 空转, 比较浪费 CPU 资源.

因此此处的自旋不会一直持续进行, 而是达到一定的时间/重试次数, 就不再自旋了.

也就是所谓的 "自适应"

3) 重量级锁

如果竞争进一步激烈, 自旋不能快速获取到锁状态, 就会膨胀为重量级锁

此处的重量级锁就是指用到内核提供的 mutex .

- 执行加锁操作, 先进入内核态.
- 在内核态判定当前锁是否已经被占用
- 如果该锁没有占用, 则加锁成功, 并切换回用户态.
- 如果该锁被占用, 则加锁失败. 此时线程进入锁的等待队列, 挂起. 等待被操作系统唤醒.
- 经历了一系列的沧海桑田, 这个锁被其他线程释放了, 操作系统也想起了这个挂起的线程, 于是唤醒这个线程, 尝试重新获取锁.

其他的优化操作

锁消除

编译器+JVM 判断锁是否可消除. 如果可以, 就直接消除.

什么是 "锁消除"

有些应用程序的代码中, 用到了 synchronized, 但其实没有在多线程环境下. (例如 StringBuffer)

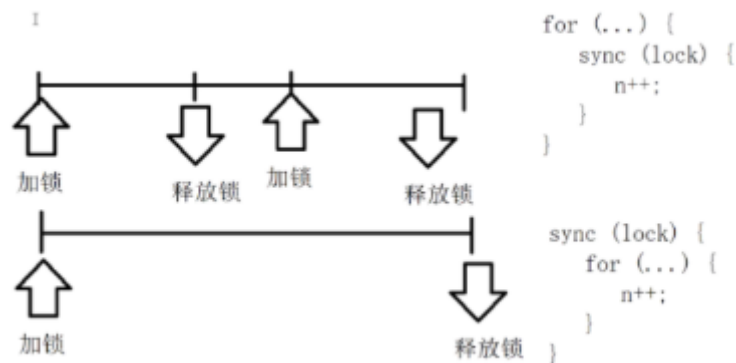
```
StringBuffer sb = new StringBuffer();
sb.append("a");
sb.append("b");
sb.append("c");
sb.append("d");
```

此时每个 append 的调用都会涉及加锁和解锁. 但如果只是在单线程中执行这个代码, 那么这些加锁解锁操作是没有必要的, 白白浪费了一些资源开销.

锁粗化

一段逻辑中如果出现多次加锁解锁, 编译器 + JVM 会自动进行锁的粗化.

锁的粒度: 粗和细



实际开发过程中, 使用细粒度锁, 是期望释放锁的时候其他线程能使用锁.

但是实际上可能并没有其他线程来抢占这个锁. 这种情况 JVM 就会自动把锁粗化, 避免频繁申请释放锁.

举个栗子理解锁粗化

滑稽老哥当了领导, 给下属交代工作任务:

方式一:

- 打电话, 交代任务1, 挂电话.
- 打电话, 交代任务2, 挂电话.
- 打电话, 交代任务3, 挂电话.

方式二:

- 打电话, 交代任务1, 任务2, 任务3, 挂电话.

显然, 方式二是更高效的方案.

可以看到, synchronized 的策略是比价复杂的, 在背后做了很多事情, 目的为了让程序猿哪怕啥都不懂, 也不至于写出特别慢的程序.

JVM 开发者为了 Java 程序猿操碎了心.



相关面试题

1) 什么是偏向锁?

偏向锁不是真的加锁, 而只是在锁的对象头中记录一个标记(记录该锁所属的线程). 如果没有其他线程参与竞争锁, 那么就不会真正执行加锁操作, 从而降低程序开销. 一旦真的涉及到其他的线程竞争, 再取消偏向锁状态, 进入轻量级锁状态.

2) synchronized 实现原理 是什么?

参考上面的 `synchronized` 原理 章节全部内容.

Callable 接口

Callable 的用法

Callable 是一个 interface . 相当于把线程封装了一个 "返回值". 方便程序猿借助多线程的方式计算结果.

代码示例: 创建线程计算 $1 + 2 + 3 + \dots + 1000$, 不使用 Callable 版本

- 创建一个类 Result , 包含一个 sum 表示最终结果, lock 表示线程同步使用的锁对象.
- main 方法中先创建 Result 实例, 然后创建一个线程 t. 在线程内部计算 $1 + 2 + 3 + \dots + 1000$.
- 主线程同时使用 wait 等待线程 t 计算结束. (注意, 如果执行到 wait 之前, 线程 t 已经计算完了, 就不必等待了).
- 当线程 t 计算完毕后, 通过 notify 唤醒主线程, 主线程再打印结果.

```
static class Result {
    public int sum = 0;
    public Object lock = new Object();
}

public static void main(String[] args) throws InterruptedException {
    Result result = new Result();

    Thread t = new Thread() {
        @Override
        public void run() {
            int sum = 0;
            for (int i = 1; i <= 1000; i++) {
                sum += i;
            }
            synchronized (result.lock) {
                result.sum = sum;
                result.lock.notify();
            }
        }
    };
    t.start();

    synchronized (result.lock) {
        while (result.sum == 0) {
            result.lock.wait();
        }
        System.out.println(result.sum);
    }
}
```

可以看到, 上述代码需要一个辅助类 Result, 还需要使用一系列的加锁和 wait notify 操作, 代码复杂, 容易出错.

代码示例: 创建线程计算 $1 + 2 + 3 + \dots + 1000$, 使用 Callable 版本

- 创建一个匿名内部类, 实现 Callable 接口. Callable 带有泛型参数. 泛型参数表示返回值的类型.
- 重写 Callable 的 call 方法, 完成累加的过程. 直接通过返回值返回计算结果.
- 把 callable 实例使用 FutureTask 包装一下.
- 创建线程, 线程的构造方法传入 FutureTask. 此时新线程就会执行 FutureTask 内部的 Callable 的 call 方法, 完成计算. 计算结果就放到了 FutureTask 对象中.
- 在主线程中调用 `futureTask.get()` 能够阻塞等待新线程计算完毕. 并获取到 FutureTask 中的结果.

```
Callable<Integer> callable = new Callable<Integer>() {
    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for (int i = 1; i <= 1000; i++) {
            sum += i;
        }
        return sum;
    }
};
FutureTask<Integer> futureTask = new FutureTask<>(callable);
Thread t = new Thread(futureTask);
t.start();

int result = futureTask.get();
System.out.println(result);
```

可以看到, 使用 Callable 和 FutureTask 之后, 代码简化了很多, 也不必手动写线程同步代码了.

理解 Callable

Callable 和 Runnable 相对, 都是描述一个 "任务". Callable 描述的是带有返回值的任务, Runnable 描述的是不带返回值的任务.

Callable 通常需要搭配 FutureTask 来使用. FutureTask 用来保存 Callable 的返回结果. 因为 Callable 往往是在另一个线程中执行的, 啥时候执行完并不确定.

FutureTask 就可以负责这个等待结果出来的工作.

理解 FutureTask

想象去吃麻辣烫. 当餐点好后, 后厨就开始做了. 同时前台会给你一张 "小票". 这个小票就是 FutureTask. 后面我们可以随时凭这张小票去查看自己的这份麻辣烫做出来了没.

相关面试题

介绍下 Callable 是什么

Callable 是一个 interface. 相当于把线程封装了一个 "返回值". 方便程序猿借助多线程的方式计算结果.

Callable 和 Runnable 相对, 都是描述一个 "任务". Callable 描述的是带有返回值的任务, Runnable 描述的是不带返回值的任务.

Callable 通常需要搭配 FutureTask 来使用. FutureTask 用来保存 Callable 的返回结果. 因为 Callable 往往是在另一个线程中执行的, 啥时候执行完并不确定.

FutureTask 就可以负责这个等待结果出来的工作.

JUC(java.util.concurrent) 的常见类

ReentrantLock

可重入互斥锁. 和 synchronized 定位类似, 都是用来实现互斥效果, 保证线程安全.

ReentrantLock 也是可重入锁. "Reentrant" 这个单词的原意就是 "可重入"

ReentrantLock 的用法:

- lock(): 加锁, 如果获取不到锁就死等.
- trylock(超时时间): 加锁, 如果获取不到锁, 等待一定的时间之后就放弃加锁.
- unlock(): 解锁

```
ReentrantLock lock = new ReentrantLock();  
-----  
  
lock.lock();  
try {  
    // working  
} finally {  
    lock.unlock()  
}
```

ReentrantLock 和 synchronized 的区别:

- synchronized 是一个关键字, 是 JVM 内部实现的(大概率是基于 C++ 实现). ReentrantLock 是标准库的一个类, 在 JVM 外实现的(基于 Java 实现).
- synchronized 使用时不需要手动释放锁. ReentrantLock 使用时需要手动释放. 使用起来更灵活, 但是也容易遗漏 unlock.
- synchronized 在申请锁失败时, 会死等. ReentrantLock 可以通过 trylock 的方式等待一段时间就放弃.
- synchronized 是非公平锁, ReentrantLock 默认是非公平锁. 可以通过构造方法传入一个 true 开启公平锁模式.

```
// ReentrantLock 的构造方法  
public ReentrantLock(boolean fair) {  
    sync = fair ? new FairSync() : new NonfairSync();  
}
```

- 更强大的唤醒机制. synchronized 是通过 Object 的 wait / notify 实现等待-唤醒. 每次唤醒的是一个随机等待的线程. ReentrantLock 搭配 Condition 类实现等待-唤醒, 可以更精确控制唤醒某个指定的线程.

如何选择使用哪个锁？

- 锁竞争不激烈的时候, 使用 `synchronized`, 效率更高, 自动释放更方便.
- 锁竞争激烈的时候, 使用 `ReentrantLock`, 搭配 `trylock` 更灵活控制加锁的行为, 而不是死等.
- 如果需要使用公平锁, 使用 `ReentrantLock`.

原子类

原子类内部用的是 CAS 实现, 所以性能要比加锁实现 `i++` 高很多。原子类有以下几个

- `AtomicBoolean`
- `AtomicInteger`
- `AtomicIntegerArray`
- `AtomicLong`
- `AtomicReference`
- `AtomicStampedReference`

以 `AtomicInteger` 举例, 常见方法有

```
addAndGet(int delta);    i += delta;
decrementAndGet();       --i;
getAndDecrement();       i--;
incrementAndGet();       ++i;
getAndIncrement();       i++;
```

线程池

虽然创建销毁线程比创建销毁进程更轻量, 但是在频繁创建销毁线程的时候还是会比较低效.

线程池就是为了解决这个问题. 如果某个线程不再使用了, 并不是真正把线程释放, 而是放到一个 "池子" 中, 下次如果需要用到线程就直接从池子中取, 不必通过系统来创建了.

ExecutorService 和 Executors

代码示例:

- `ExecutorService` 表示一个线程池实例.
- `Executors` 是一个工厂类, 能够创建出几种不同风格的线程池.
- `ExecutorService` 的 `submit` 方法能够向线程池中提交若干个任务.

```
ExecutorService pool = Executors.newFixedThreadPool(10);
pool.submit(new Runnable() {
    @Override
    public void run() {
        System.out.println("hello");
    }
});
```

Executors 创建线程池的几种方式

- `newFixedThreadPool`: 创建固定线程数的线程池
- `newCachedThreadPool`: 创建线程数目动态增长的线程池.
- `newSingleThreadExecutor`: 创建只包含单个线程的线程池.
- `newScheduledThreadPool`: 设定 延迟时间后执行命令, 或者定期执行命令. 是进阶版的 `Timer`.

Executors 本质上是 `ThreadPoolExecutor` 类的封装.

ThreadPoolExecutor

`ThreadPoolExecutor` 提供了更多的可选参数, 可以进一步细化线程池行为的设定.

ThreadPoolExecutor 的构造方法

Constructors
Constructor and Description
<code>ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue)</code> Creates a new <code>ThreadPoolExecutor</code> with the given initial parameters and default thread factory and rejected execution handler.
<code>ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, RejectedExecutionHandler handler)</code> Creates a new <code>ThreadPoolExecutor</code> with the given initial parameters and default thread factory.
<code>ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory)</code> Creates a new <code>ThreadPoolExecutor</code> with the given initial parameters and default rejected execution handler.
<code>ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler)</code> Creates a new <code>ThreadPoolExecutor</code> with the given initial parameters.

理解 ThreadPoolExecutor 构造方法的参数

把创建一个线程池想象成开个公司. 每个员工相当于一个线程.

- `corePoolSize`: 正式员工的数量. (正式员工, 一旦录用, 永不辞退)
- `maximumPoolSize`: 正式员工 + 临时工的数目. (临时工: 一段时间不干活, 就被辞退).
- `keepAliveTime`: 临时工允许的空闲时间.
- `unit`: `keepAliveTime` 的时间单位, 是秒, 分钟, 还是其他值.
- `workQueue`: 传递任务的阻塞队列
- `threadFactory`: 创建线程的工厂, 参与具体的创建线程工作.
- `RejectedExecutionHandler`: 拒绝策略, 如果任务量超出公司的负荷了接下来怎么处理.
 - `AbortPolicy()`: 超过负荷, 直接抛出异常.
 - `CallerRunsPolicy()`: 调用者负责处理
 - `DiscardOldestPolicy()`: 丢弃队列中最老的任务.
 - `DiscardPolicy()`: 丢弃新来的任务.

代码示例:

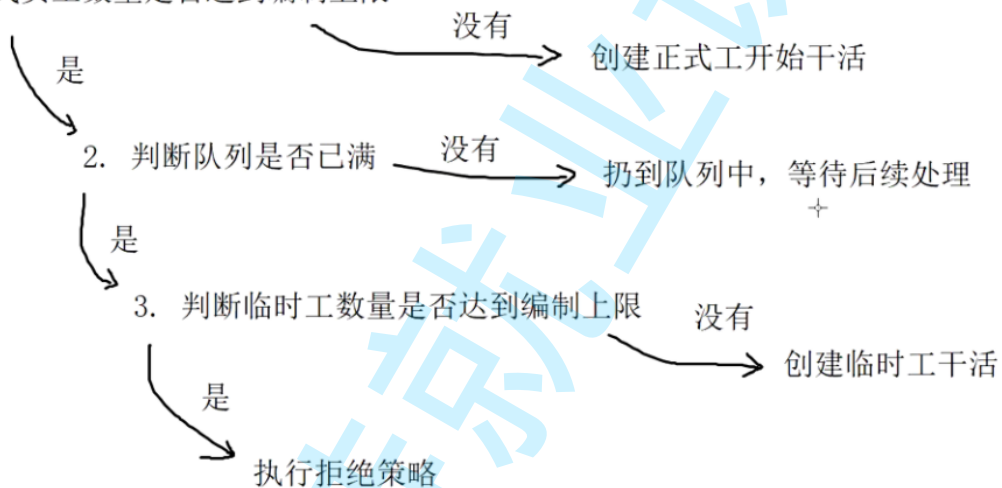
```

ExecutorService pool = new ThreadPoolExecutor(1, 2, 1000, TimeUnit.MILLISECONDS,
                                              new SynchronousQueue<Runnable>(),
                                              Executors.defaultThreadFactory(),
                                              new
ThreadPoolExecutor.AbortPolicy());
for(int i=0;i<3;i++) {
    pool.submit(new Runnable() {
        @Override
        void run() {
            System.out.println("hello");
        }
    });
}

```

线程池的工作流程

判断正式员工数量是否达到编制上限



信号量 Semaphore

信号量, 用来表示 "可用资源的个数". 本质上就是一个计数器.

理解信号量

可以把信号量想象成是停车场的展示牌: 当前有车位 100 个. 表示有 100 个可用资源.

当有车开进去的时候, 就相当于申请一个可用资源, 可用车位就 -1 (这个称为信号量的 P 操作)

当有车开出来的时候, 就相当于释放一个可用资源, 可用车位就 +1 (这个称为信号量的 V 操作)

如果计数器的值已经为 0 了, 还尝试申请资源, 就会阻塞等待, 直到有其他线程释放资源.

Semaphore 的 PV 操作中的加减计数器操作都是原子的, 可以在多线程环境下直接使用.

代码示例

- 创建 Semaphore 示例, 初始化为 4, 表示有 4 个可用资源.
- acquire 方法表示申请资源(P操作), release 方法表示释放资源(V操作)
- 创建 20 个线程, 每个线程都尝试申请资源, sleep 1秒之后, 释放资源. 观察程序的执行效果.


```

Semaphore semaphore = new Semaphore(4);

Runnable runnable = new Runnable() {
    @Override
    public void run() {
        try {
            System.out.println("申请资源");
            semaphore.acquire();
            System.out.println("我获取到资源了");
            Thread.sleep(1000);
            System.out.println("我释放资源了");
            semaphore.release();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
};

for (int i = 0; i < 20; i++) {
    Thread t = new Thread(runnable);
    t.start();
}

```

CountDownLatch

同时等待 N 个任务执行结束。

好像跑步比赛，10个选手依次就位，哨声响才同时出发；所有选手都通过终点，才能公布成绩。

- 构造 CountDownLatch 实例, 初始化 10 表示有 10 个任务需要完成。
- 每个任务执行完毕, 都调用 `latch.countDown()` . 在 CountDownLatch 内部的计数器同时自减。
- 主线程中使用 `latch.await()`; 阻塞等待所有任务执行完毕. 相当于计数器为 0 了。

```

public class Demo {
    public static void main(String[] args) throws Exception {
        CountDownLatch latch = new CountDownLatch(10);
        Runnable r = new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(Math.random() * 10000);
                    latch.countDown();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        };
        for (int i = 0; i < 10; i++) {
            new Thread(r).start();
        }
        // 必须等到 10 人全部回来
        latch.await();
        System.out.println("比赛结束");
    }
}

```

```
}  
}
```

相关面试题

1) 线程同步的方式有哪些?

synchronized, ReentrantLock, Semaphore 等都可以用于线程同步.

2) 为什么有了 synchronized 还需要 juc 下的 lock?

以 juc 的 ReentrantLock 为例,

- synchronized 使用时不需要手动释放锁. ReentrantLock 使用时需要手动释放. 使用起来更灵活,
- synchronized 在申请锁失败时, 会死等. ReentrantLock 可以通过 trylock 的方式等待一段时间就放弃.
- synchronized 是非公平锁, ReentrantLock 默认是非公平锁. 可以通过构造方法传入一个 true 开启公平锁模式.
- synchronized 是通过 Object 的 wait / notify 实现等待-唤醒. 每次唤醒的是一个随机等待的线程. ReentrantLock 搭配 Condition 类实现等待-唤醒, 可以更精确控制唤醒某个指定的线程.

3) AtomicInteger 的实现原理是什么?

基于 CAS 机制. 伪代码如下:

```
class AtomicInteger {  
    private int value;  
  
    public int getAndIncrement() {  
        int oldValue = value;  
        while ( CAS(value, oldValue, oldValue+1) != true) {  
            oldValue = value;  
        }  
        return oldValue;  
    }  
}
```

执行过程参考 "CAS 有哪些应用" 章节.

4) 信号量听说过么? 之前都用在过哪些场景下?

信号量, 用来表示 "可用资源的个数". 本质上就是一个计数器.

使用信号量可以实现 "共享锁", 比如某个资源允许 3 个线程同时使用, 那么就可以使用 P 操作作为加锁, V 操作作为解锁, 前三个线程的 P 操作都能顺利返回, 后续线程再进行 P 操作就会阻塞等待, 直到前面的线程执行了 V 操作.

5) 解释一下 ThreadPoolExecutor 构造方法的参数的含义

线程安全的集合类

原来的集合类, 大部分都不是线程安全的.

Vector, Stack, HashTable, 是线程安全的(不建议用), 其他的集合类不是线程安全的.

多线程环境使用 ArrayList

1) 自己使用同步机制 (synchronized 或者 ReentrantLock)

前面做过很多相关的讨论了. 此处不再展开.

2) `Collections.synchronizedList(new ArrayList());`

synchronizedList 是标准库提供的一个基于 synchronized 进行线程同步的 List. synchronizedList 的关键操作上都带有 synchronized

3) 使用 CopyOnWriteArrayList

CopyOnWrite容器即写时复制的容器。

- 当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，
- 添加完元素之后，再将原容器的引用指向新的容器。

这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。

所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

优点:

在读多写少的场景下, 性能很高, 不需要加锁竞争.

缺点:

1. 占用内存较多.
2. 新写的数据不能被第一时间读取到.

多线程环境使用队列

1) ArrayBlockingQueue

基于数组实现的阻塞队列

2) LinkedBlockingQueue

基于链表实现的阻塞队列

3) PriorityBlockingQueue

基于堆实现的带优先级的阻塞队列

4) TransferQueue

多线程环境使用哈希表

HashMap 本身不是线程安全的.

在多线程环境下使用哈希表可以使用:

- Hashtable
- ConcurrentHashMap

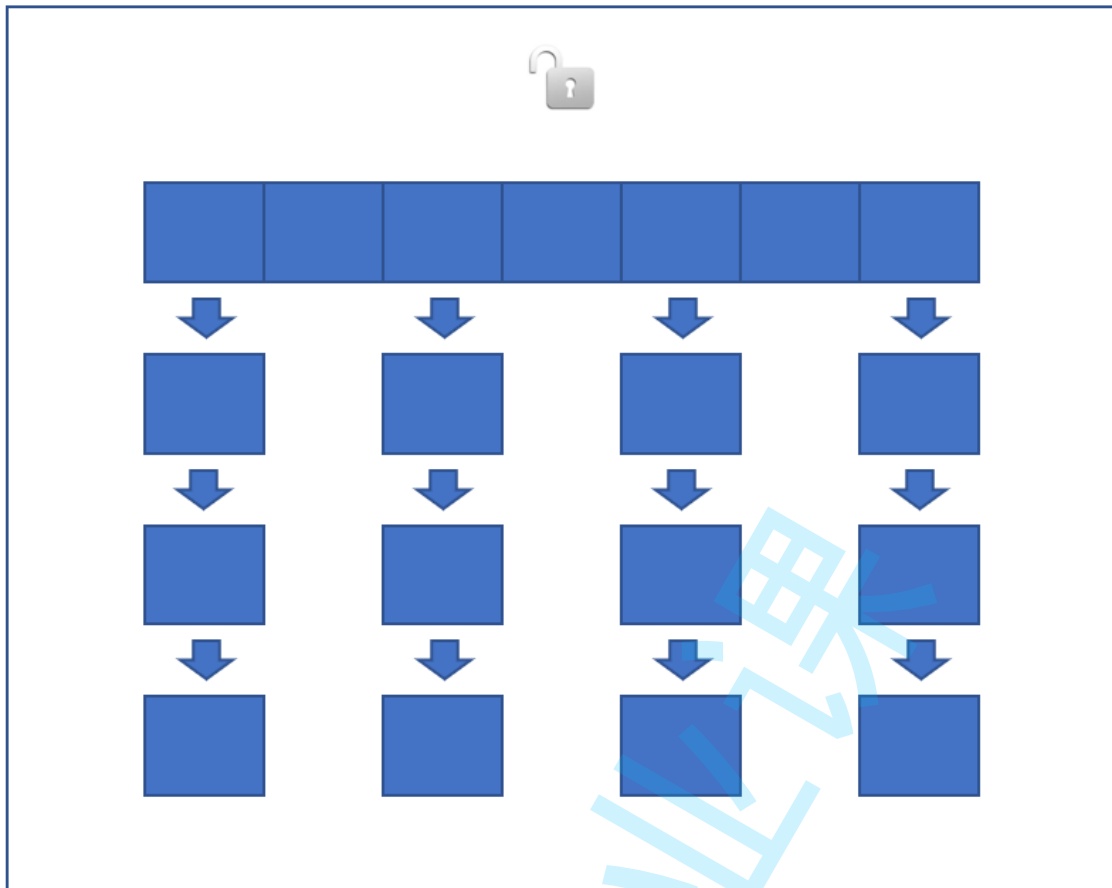
1) Hashtable

只是简单的把关键方法加上了 synchronized 关键字.

```
public synchronized V put(K key, V value) {  
  
public synchronized V get(Object key) {
```

这相当于直接针对 Hashtable 对象本身加锁.

- 如果多线程访问同一个 Hashtable 就会直接造成锁冲突.
- size 属性也是通过 synchronized 来控制同步, 也是比较慢的.
- 一旦触发扩容, 就由该线程完成整个扩容过程. 这个过程会涉及到大量的元素拷贝, 效率会非常低.

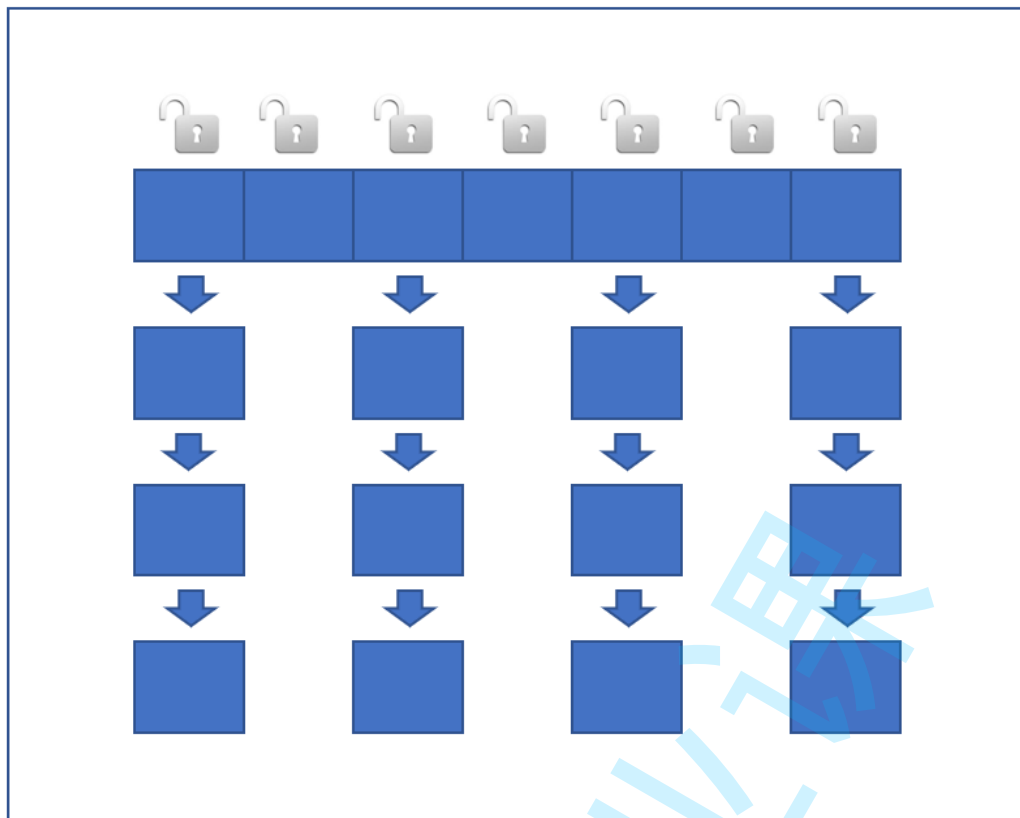


一个 Hashtable 只有一把锁。
两个线程访问 Hashtable 中的任意数据都会出现锁竞争

2) ConcurrentHashMap

相比于 Hashtable 做出了一系列的改进和优化. 以 Java1.8 为例

- 读操作没有加锁(但是使用了 volatile 保证从内存读取结果), 只对写操作进行加锁. 加锁的方式仍然是用 synchronized, 但是不是锁整个对象, 而是 "锁桶" (用每个链表的头结点作为锁对象), 大大降低了锁冲突的概率.
- 充分利用 CAS 特性. 比如 size 属性通过 CAS 来更新. 避免出现重量级锁的情况.
- 优化了扩容方式: 化整为零
 - 发现需要扩容的线程, 只需要创建一个新的数组, 同时只搬几个元素过去.
 - 扩容期间, 新老数组同时存在.
 - 后续每个来操作 ConcurrentHashMap 的线程, 都会参与搬家的过程. 每个操作负责搬运一小部分元素.
 - 搬完最后一个元素再把老数组删掉.
 - 这个期间, 插入只往新数组加.
 - 这个期间, 查找需要同时查新数组和老数组



ConcurrentHashMap 每个哈希桶都有一把锁。
只有两个线程访问的恰好是同一个哈希桶上的数据才出现锁冲突

参考资料: <https://blog.csdn.net/u010723709/article/details/48007881>

相关面试题

1) ConcurrentHashMap的读是否要加锁, 为什么?

读操作没有加锁. 目的是为了进一步降低锁冲突的概率. 为了保证读到刚修改的数据, 搭配了 volatile 关键字.

2) 介绍下 ConcurrentHashMap的锁分段技术?

这个是 Java1.7 中采取的技术. Java1.8 中已经不再使用了. 简单的说就是把若干个哈希桶分成一个 "段" (Segment), 针对每个段分别加锁.

目的也是为了降低锁竞争的概率. 当两个线程访问的数据恰好在同一个段上的时候, 才触发锁竞争.

3) ConcurrentHashMap在jdk1.8做了哪些优化?

取消了分段锁, 直接给每个哈希桶(每个链表)分配了一个锁(就是以每个链表的头结点对象作为锁对象).

将原来 数组 + 链表 的实现方式改进成 数组 + 链表 / 红黑树 的方式. 当链表较长的时候(大于等于 8 个元素)就转换成红黑树.

4) Hashtable和HashMap、ConcurrentHashMap 之间的区别?

HashMap: 线程不安全. key 允许为 null

Hashtable: 线程安全. 使用 synchronized 锁 Hashtable 对象, 效率较低. key 不允许为 null.

ConcurrentHashMap: 线程安全. 使用 synchronized 锁每个链表头结点, 锁冲突概率低, 充分利用 CAS 机制. 优化了扩容方式. key 不允许为 null

死锁

死锁是什么

死锁是这样一种情形: 多个线程同时被阻塞, 它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞, 因此程序不可能正常终止。

举个栗子理解死锁

滑稽老哥和女神一起去饺子馆吃饺子. 吃饺子需要酱油和醋.

滑稽老哥抄起了酱油瓶, 女神抄起了醋瓶.

滑稽: 你先把醋瓶给我, 我用完了就把酱油瓶给你.

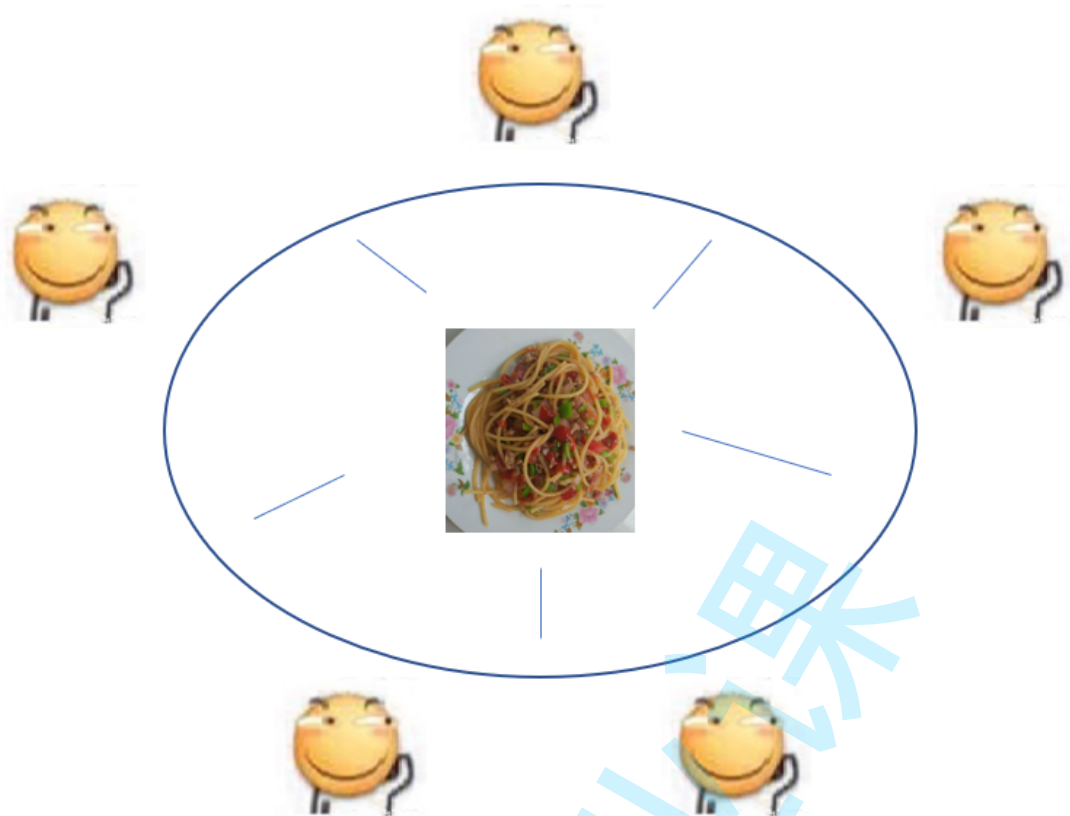
女神: 你先把酱油瓶给我, 我用完了就把醋瓶给你.

如果这两人彼此之间互不相让, 就构成了死锁.

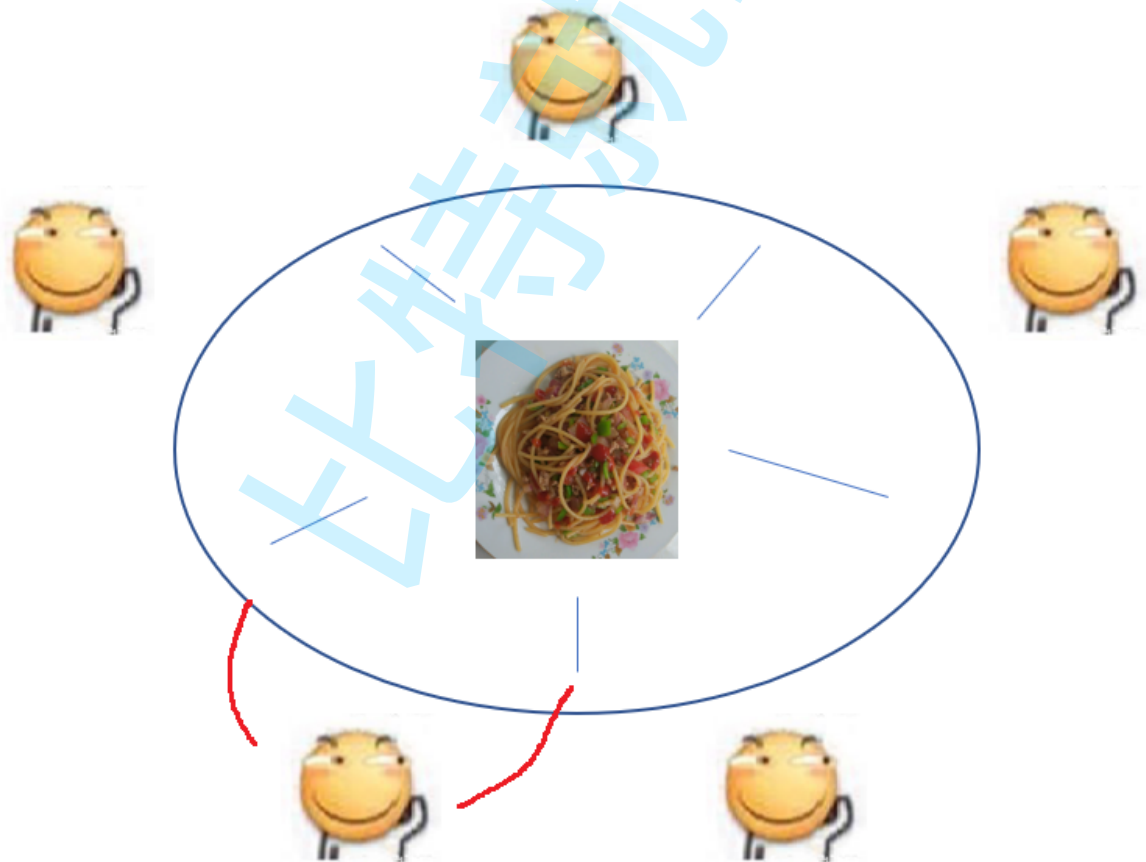
酱油和醋相当于是两把锁, 这两个人就是两个线程.

为了进一步阐述死锁的形成, 很多资料上也会谈到 "哲学家就餐问题".

- 有个桌子, 围着一圈哲♂家, 桌子中间放着一盘意大利面. 每个哲学家两两之间, 放着一根筷子.

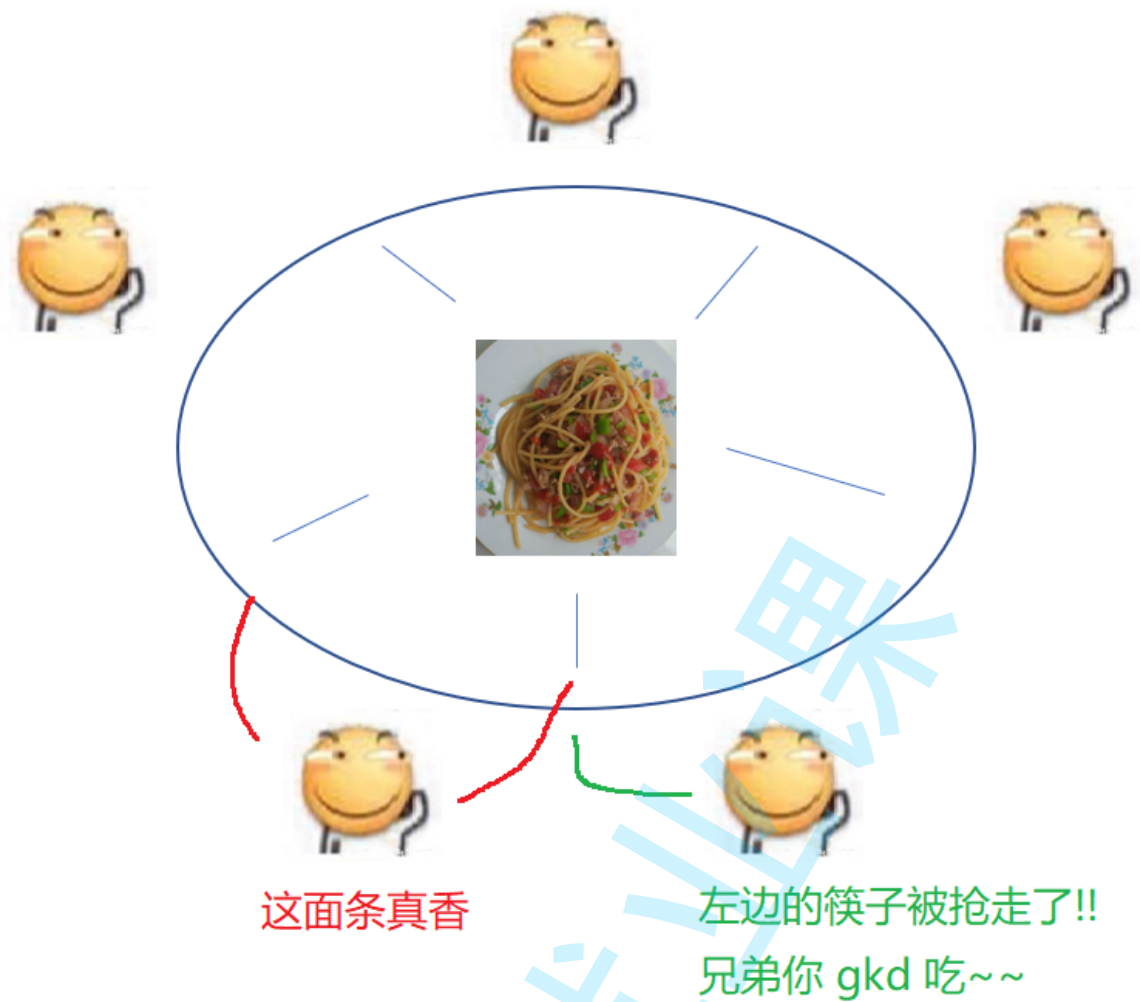


- 每个哲♂家 只做两件事: 思考人生 或者 吃面条. 思考人生的时候就会放下筷子. 吃面条就会拿起左右两边的筷子(先拿起左边, 再拿起右边).

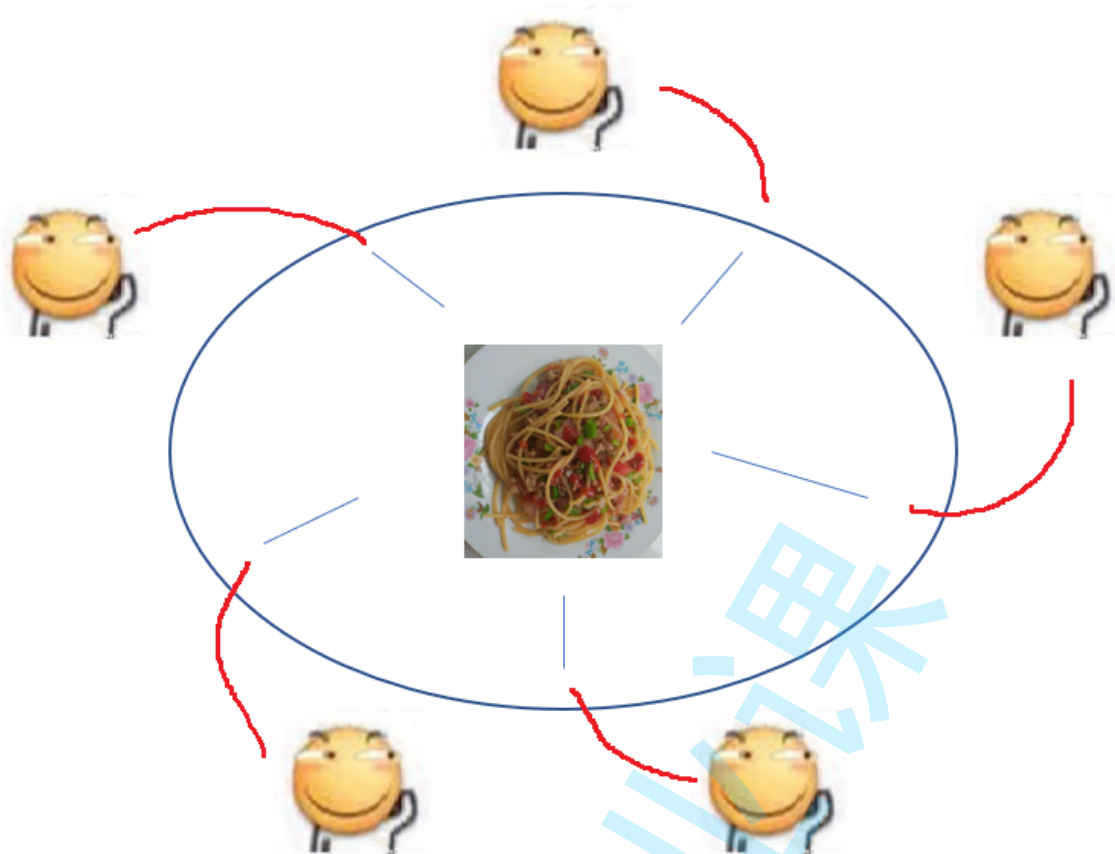


这面条真香

- 如果 哲♂家 发现筷子拿不起来了(被别人占用了), 就会阻塞等待.



- [关键点在这] 假设同一时刻, 五个哲♂家同时拿起左手边的筷子, 然后再尝试拿右手的筷子, 就会发现右手的筷子都被占用了. 由于哲♂家们互不相让, 这个时候就形成了**死锁**



右手的筷子咋都被占用了!!!

死锁是一种严重的 BUG!! 导致一个程序的线程 "卡死", 无法正常工作!

如何避免死锁

死锁产生的四个必要条件:

- 互斥使用, 即当资源被一个线程使用(占有)时, 别的线程不能使用
- 不可抢占, 资源请求者不能强制从资源占有者手中夺取资源, 资源只能由资源占有者主动释放。
- 请求和保持, 即当资源请求者在请求其他的资源的同时保持对原有资源的占有。
- 循环等待, 即存在一个等待队列: P1占有P2的资源, P2占有P3的资源, P3占有P1的资源。这样就形成了一个等待环路。

当上述四个条件都成立的时候, 便形成死锁。当然, 死锁的情况下如果打破上述任何一个条件, 便可让死锁消失。

其中最容易破坏的就是 "循环等待".

破坏循环等待

最常用的一种死锁阻止技术就是**锁排序**. 假设有 N 个线程尝试获取 M 把锁, 就可以针对 M 把锁进行编号 (1, 2, 3...M).

N 个线程尝试获取锁的时候, 都按照固定的按编号由小到大顺序来获取锁. 这样就可以避免环路等待.

可能产生环路等待的代码:

两个线程对于加锁的顺序没有约定, 就容易产生环路等待.

```
Object lock1 = new Object();
Object lock2 = new Object();

Thread t1 = new Thread() {
    @Override
    public void run() {
        synchronized (lock1) {
            synchronized (lock2) {
                // do something...
            }
        }
    }
};
t1.start();

Thread t2 = new Thread() {
    @Override
    public void run() {
        synchronized (lock2) {
            synchronized (lock1) {
                // do something...
            }
        }
    }
};
t2.start();
```

不会产生环路等待的代码:

约定好先获取 lock1, 再获取 lock2, 就不会环路等待.

```
Object lock1 = new Object();
Object lock2 = new Object();

Thread t1 = new Thread() {
    @Override
    public void run() {
        synchronized (lock1) {
            synchronized (lock2) {
                // do something...
            }
        }
    }
};
t1.start();

Thread t2 = new Thread() {
    @Override
    public void run() {
        synchronized (lock1) {
            synchronized (lock2) {
                // do something...
            }
        }
    }
};
```

```
};  
t2.start();
```

相关面试题

谈谈死锁是什么，如何避免死锁，避免算法？实际解决过没有？

参考整个 "死锁" 章节

8. 其他常见问题

面试题：

1) 谈谈 volatile 关键字的用法？

volatile 能够保证内存可见性. 强制从主内存中读取数据. 此时如果有其他线程修改被 volatile 修饰的变量, 可以第一时间读取到最新的值.

2) Java多线程是如何实现数据共享的？

JVM 把内存分成了这几个区域:

方法区, 堆区, 栈区, 程序计数器.

其中堆区这个内存区域是多个线程之间共享的.

只要把某个数据放到堆内存中, 就可以让多个线程都能访问到.

3) Java创建线程池的接口是什么？参数 `LinkedBlockingQueue` 的作用是什么？

创建线程池主要有两种方式:

- 通过 `Executors` 工厂类创建. 创建方式比较简单, 但是定制能力有限.
- 通过 `ThreadPoolExecutor` 创建. 创建方式比较复杂, 但是定制能力强.

`LinkedBlockingQueue` 表示线程池的任务队列. 用户通过 `submit / execute` 向这个任务队列中添加任务, 再由线程池中的工作线程来执行任务.

4) Java线程共有几种状态？状态之间怎么切换的？

- `NEW`: 安排了工作, 还未开始行动. 新创建的线程, 还没有调用 `start` 方法时处在这个状态.
- `RUNNABLE`: 可工作的. 又可以分成正在工作中和即将开始工作. 调用 `start` 方法之后, 并正在 CPU 上运行/在即将准备运行 的状态.
- `BLOCKED`: 使用 `synchronized` 的时候, 如果锁被其他线程占用, 就会阻塞等待, 从而进入该状态.
- `WAITING`: 调用 `wait` 方法会进入该状态.
- `TIMED_WAITING`: 调用 `sleep` 方法或者 `wait(超时时间)` 会进入该状态.
- `TERMINATED`: 工作完成了. 当线程 `run` 方法执行完毕后, 会处于这个状态.

5) 在多线程下, 如果对一个数进行叠加, 该怎么做?

- 使用 `synchronized` / `ReentrantLock` 加锁
- 使用 `AtomicInteger` 原子操作.

6) Servlet是否是线程安全的?

Servlet 本身是工作在多线程环境下.

如果在 Servlet 中创建了某个成员变量, 此时如果有多个请求到达服务器, 服务器就会多线程进行操作, 是可能出现线程不安全的情况的.

7) Thread和Runnable的区别和联系?

`Thread` 类描述了一个线程.

`Runnable` 描述了一个任务.

在创建线程的时候需要指定线程完成的任务, 可以直接重写 `Thread` 的 `run` 方法, 也可以使用 `Runnable` 来描述这个任务.

8) 多次start一个线程会怎么样

第一次调用 `start` 可以成功调用.

后续再调用 `start` 会抛出 `java.lang.IllegalThreadStateException` 异常

9) 有synchronized两个方法, 两个线程分别同时用这个方法, 请问会发生什么?

`synchronized` 加在非静态方法上, 相当于针对当前对象加锁.

如果这两个方法属于同一个实例:

线程1 能够获取到锁, 并执行方法. 线程2 会阻塞等待, 直到线程1 执行完毕, 释放锁, 线程2 获取到锁之后才能执行方法内容.

如果这两个方法属于不同实例:

两者能并发执行, 互不干扰.

10) 进程和线程的区别?

- 进程是包含线程的. **每个进程至少有一个线程存在, 即主线程。**
- 进程和进程之间不共享内存空间. 同一个进程的线程之间共享同一个内存空间.
- 进程是系统分配资源的最小单位, 线程是系统调度的最小单位。