[toc]

我所遇到的计组问题

浮点运算相关问题

斐波拉切数列解法的溢出问题， 矩阵运算中的double型变量的尾数舍入误差。

这些都是计组中数据的机器层次表示的知识，有谁还记得吗，其中的浮点数的表示就有我上面所提到的问题。。

要解答上面的我所遇到的问题，首先我们要回忆一下机器数的浮点表示。
首先浮点数由三部分组成：

1. 符号位，通常由一位组成
2. 阶码部分
3. 尾数部分

阶码和尾数的位数及表示方法（阶码的移码表示法，IEEE754尾数的原码表示法等等）取决具体机器，但是我们能确定的是，他们的位数一定是有限的。上面的误差问题，也正是出现在这里。当我们所表示的数超过尾数能够表示的范围的时候，这个时候就出现了假溢出现象，这里的假溢出不同整形数据的表示表示方法，整型是一旦溢出就会发生绕回现象，但是浮点数的尾数部分并不会出现这种现象，原因就是还有阶码的存在，可以通过移位的方法解决假溢出的问题。但是又带来了另一个问题：舍入问题！
我们知道，移位意味着有些比特位会被移走，因而就有可能对尾数部分数值的准确数值产生影响。比如如果右移的部分存在为1的比特位，那么肯定会影响尾数的准确值，如果右移的部分都是 0 ，那么结果就没有影响，就如上面斐波拉切数列的求解问题虽然第77在数值上超过了尾数的表示范围，但是其值仍然是准确的。
既然移位会带来精确性的损失，那我们又应该如何去削弱其对我们的影响呢？

在计组里的数值的机器运算里，就提到了几种方案：

1. 恒舍法
2. 冯诺依曼舍入法（恒置 1 法）：无论是多少都置为 1
3. 上舍下入法（0舍1入法）：相当于四舍五入法，当舍去部分的最高位为1时向前进1,当其为0时，舍去。
4. 查表法： 每次经查表得到其结果。

所以我们对于前面所遇到的误差的原因也就清楚了。 不过我当时查了资料没有找到实际在java里到底是采用何种方案，是所有平台的方法一致，还是依赖于平台，不过我猜测依赖是平台的表示。

其实浮点运算的误差问题在支付领域尤其明显。例如微信支付宝转账，微信手气红包的分割问题等等。因为支付涉及到钱的问题，所以在处理时要注意一份都不能少，但很明显浮点运算限制与机器的本身底层实现，经常会出现钱少了或者更严重的钱没了的问题。这里我只是抛砖引玉，具体内容我就不深入了，留给同学们自己探讨。

并发编程的可见性问题，原子问题，编译优化问题

并发中的原子性问题
上面所出现的问题包括可见性问题及原子性问题。

要了解上面所出现的问题，我们必须要回忆下计组中的缓存及指令的知识。

首先我们来看看可见性问题：
可见性： 一个线程对共享变量的修改，另一个线程能够立刻看到。
可见性问题出现的原因是因为有缓存的存在。缓存的存在是为了解决内存与cpu之间速度差距的问题，当然其中还有成本的问题。之后在内存与cpu之间加了一个中间者来协调两个关系。

对于单核，所有线程操作是同一个cpu缓存，因而缓存与内存之间的一致性很容易保证。
对于多核， 在不同cpu内核上执行的线程使用的是不同的cpu缓存。一个线程对同一变量执行的操作，对另一线程而言就不具可见性。 这里具体的原因就是。。。。

同时我当时还想起关于缓存的相关知识点，就是缓存的更新策略问题：

1. 写直达法：cpu在执行写操作时，必须把数据同时写入Cache和主存,缺点就是可能增加访问主存的次数，导致效率的下降。
2. 写回法： 只写Cache,不写入主存，当发生Cache替换时，才会把数据写回内存。

写直达法，其实就是解决可见性问题的方法，因为它每次更新值的时候都会写进内存，那么其他线程也就可以看见了。 不过这样我们最终的结果还是会有问题的，原因就是原子性问题。

其实有认真听操作系统课的同学，肯定已经知道什么是原子性。在信号量那里就有很详细的解释，这里只针对我所提出的问题进行解释。

真的跟你们在操作系统课上面听到的一定是一模一样的。

在并发bug问题里，其实还有编译优化的问题，因为涉及到编译原理相关知识，这里不作探讨。

c 语言学来干嘛？？

1. 底层语言可以帮助我们学习底层的知识 特别就是在它可以对内存进行直接的操作，最重要的一个工具就是指针。我们可以自由申请内存空间，对内存空间进行操作。我们甚至可以对程序计数器进行操作，工具就是函数指针，函数指针我真心觉得是一个很复杂的东西，不用说别的，他的定义就很具有迷惑性，你真得咬死语言特性，才能很清晰地了解它到底是干嘛的。其他的呀，还有原子操作啊，寄存器等等的相关知识，我也就不深入了。一句话， c /c++很硬核。
2. java、python中一些包，或者说java python 虚拟机，底层就是用c语言实现的，如java线程源码中的start()函数。java进阶阅读源码， c语言学不好， 看个鬼！比如java虚拟机在加载.class文件的时候，将字节码转换为机器码有两种方式， 1 ． 对应函数， 2 ．函数指针。
3. 函数的值传递方式： 1 ．值复制， 2 ．指针引用

汇编语言又可以干啥用

1. 马东宇同学的逆向工程
2. 上面讲到的java虚拟机的字节码的设计同样与汇编语言息息相关。

eda

算法的有限状态机， 电梯模拟 其实当思考这些问题的时候，我对于数据库里的概念模型也是有恍然大悟的感觉。 课本里讲了一个基本的计算机思维: 人们用概念模型来完成从现实世界到信息世界的映射，重要的工具就是E - R图，通过建模的方式来抽象我们对于现实世界的理解。上面我们同样是做了一个重要的工作，就是用状

态的转化来模拟现实世界中的实际问题，这里是用一个个状态来表示每次的过河或者过桥后情况，完成了这项工作，我们就可以用计算机世界的工具或者方法来解决现实中的问题，例如穷举所有可能的情况，或者将其看成有向带权图的一个节点，之后通过广度或者深度优先历遍来找到最短的路径从而找到问题的解法（这里是不是，离散数学，数据结构就可以用上了）。（需求工程是不是也可以出来了）

卡诺图　公式化简

其他

高数线代离散是重要的数学基础，数据结构算法这些都不用我强调了。

今天我所讲的内容，并不是说真的帮你们复习所学的课程，而是想达到一个抛砖引玉的效果，提醒你们重视底层的原理学习，认真听学校给我们安排的基础课程，虽然你觉得你现在用不上，但是总有一天，你会发现，现在所学真的会帮你一个大忙。谢谢大家。

附件 1

type

o(1): Constant Complexity
o(log n): Logarithmic Complexity
o(n): Linear Complexity
o(n^2): N square Complexity
o(n^3): N cube Complexity
o(2^n): Exponential Growth
o(n!): Factorial

递归的时间复杂度运算

斐波拉切数列的例子

这里需要特别注意精度溢出的问题，若用int存储，只能执行到46(18亿)(int 21亿)
若用long能执行到：92(7.5*10^18)(long: 9*10^18), 对于矩阵快速法，要考虑
到尾数部分有52位，指数11位，符号位 1 位，所以到超过76(76: 3416454622906707 77: 5527939700884757)
(2^52=4.503599627×10^{15})时就开始产生误差。 但移位的舍入方法不祥。
误差的产生原因是浮点数的尾数部分产生了假溢出。

指数递归方法

斐波拉切数列递归实现时间复杂度: 2^n
计算方法: 差分方程： (1 + 根5 / 2) ^ 2

```
public class  Recursion{
   long fib(int n){
        if(0 == n || 1 == n){
```

```
            return n;
        }
        return fib(n - 1) + fib(n - 2);
    }
}
```

```
fun fib(n: Long): Long =
    if(0L == n || 1L == n){
       n
    }
    else{
        fib(n - 1) + fib(n - 2)
    }
```

尾递归实现方法：

尾递归的定义：在一个程序中，执行最后一条语句是对自己的调用，且没有其他的运算
尾递归的实现： 在编译器优化的条件下实现的。递归的第一次调用会开辟一份空间，
此后的递归调用会在该开辟的空间上执行，而不会重新开辟。
本质：与循环法没有任何区别，也就这种尾递归的方法完全可以转化为循环来实现。 用这种的方法不好了解，失去了递归的应有的简介性，推荐使用循环实现。 时间复杂度： O(n-2) = O(n)
空间复杂度： O(n)(编译器不优化) O(1)(编译器优化)

```
public class TailRecursion{
    public long fibRoot(long first, long second, int n){
        if(n == 3){
            return first + second;
        }
        if(n <= 2 && n >=0){
            return n;
        }
        if(n < 0){
            return 0L;
        }

        return fibRoot(second, first + second, n - 1);
    }

    public int fib(int n){
        return fibRoot(1L, 1L, n);
    }
}
```

```
fun fibRoot(first: Long, second: Long, n: Int): Long = when{
    n == 3         -> first + second
    in range(0,3) -> 1L
```

```
    n < 0          -> 0L
    else           -> fibRoot(second, first + second, n - 1)
}

fun fib(n: Int) = fibRoot(1L, 1L, n)
```

循环实现方法

时间复杂度： O(n-2) = O(n)
空间复杂度： O(1)

```
public class cycleFib{
    public static long fib(int n){
        long tmp = 0L;
        long pre = 1L;
        long res = 1L;
        for(int i = 3;i <= n;i++){
            tmp = pre;
            pre = res;
            res = tmp + pre;
        }
        return n <= 0 ? 0L : res;
    }
}
```

```
fun fib(n: Int):Long {
    var tmp = 0L
    var pre = 1L
    var res = 1L
    for(i in 3..n+1){
        tmp = pre
        pre = res
        res = tmp + pre
    }

    return if(n <= 0) 0L else res
}
```

矩阵实现

| fn+1 fn | = | fn fn-1||*1 1*|...= |*f2 f1*|(|1 1|)^n-1 = (|1 1|)^n | fn fn-1| = | fn-1 fn-2||*1 0*|...= |*f1 f0*|(|1 0|) = (|1 0|)

令 Q = |1 1| = |1 0| 则 | fn+1 fn | = Q^n
| fn fn-1| = 则 fn = (Q^n)0,1 (坐标从(0,0)开始)

矩阵快速幂： a^n = a^010...00(2) = a^(2^k*0 + 2^k-1 *1...+2^1*0+2^0*0)
= (a^2^k)^0*(a^2^k-1)^1....(a^2^1)^0*(a^2^0)^0 a^2^k = a^2^k-1 * a^2^k-1

time = O(log n)

```
import org.apache.commons.math3.linear.Array2DRowRealMatrix;
import org.apache.commons.math3.linear.RealMatrix;

public class Test{
    public static double fib(int n){
        // 若传入的参数<=0，则直接返回0
        if(n <= 0){
            return 0;
        }
        // 斐波拉切Q - 矩阵的数组数据
        double [][] QData = {{1, 1},
                             {1, 0}};
        // 斐波拉切Q - 矩阵
        RealMatrix Q = new Array2DRowRealMatrix(QData);

        // Fn = (Q^n)0,1
        //  将结果矩阵初始化为单位阵
        double [][] resultMatrixData = {{1, 0},
                                        {0, 1}};
        RealMatrix resultMatrix = new
Array2DRowRealMatrix(resultMatrixData);
        while(0 != n){
            //判断最低位是否为 1 ， 1 则乘上Q矩阵
            if((n & 1) == 1){
                resultMatrix = resultMatrix.multiply(Q);
            }
            //  n左移一位
            n >>= 1;
            //  Q 矩阵倍增
            Q = Q.multiply(Q);
        }

        double fn = resultMatrix.getEntry(0, 1);
        return fn;
        }
}
```

附件 2

01|可见性、原子性和有序性问题：并发编程的源头

编写正确的并发程序是一件极其困难的事情，并发程序的bug往往会诡异的出现，又会诡异的消失，很难重现，也很难追踪。 正确解决方法是溯源。
为合理利用cpu高性能，平衡cpu io 内存之间的速度差距：

1. 计算机体系结构方面： 增加了cpu缓存， 均衡cpu与内存之间的速度差异。

2. 操作系统: 增加进程、线程等分时复用技术，均衡cpu与io之间的速度差异，提高cpu的利用率。
3. 编译器方面： 优化指令执行次序，使缓存能够更合理地利用。

源头之一： 缓存导致的可见性问题

定义：

可见性： 一个线程对共享变量的修改，另一个线程能够立刻看到。

可见性问题原因

对于单核，所有线程操作是同一个cpu缓存，因而缓存与内存之间的一致性很容易保证。
对于多核， 在不同cpu内核上执行的线程使用的是不同的cpu缓存。一个线程对同一变量执行的操作，对另一线程而言就不具可见性。

具体例子

代码

```
public class Course01 {
    private long count = 0;
    private void add1Billion(){
        // 当数据为1w, 10w时结果可能正确，因为不同的线程启动有一个时间差，
        // 而在这个时间差之内，一个线程的加法操作可能已经完成。
        // 而当数据逐渐增大时，发生缓存可见性问题的可能性变大
        for(int i = 0; i < 1000000000; i++){
            count ++;
        }
    }

    public static long calc(){
        final Course01 course01 = new Course01();
        // 启动两个线程对同一个变量进行加法操作
        Thread thread1 = new Thread(() -> course01.add1Billion());
        Thread thread2 = new Thread(() -> course01.add1Billion());
        thread1.start();
        thread2.start();
        try {
            //  等待两个线程的结束
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return course01.getCount();
    }

    public long getCount() {
        return count;
```

```
        }
    }
```

分析

当两个线程的同时对count进行操作时，会从内存中将count的副本读入各自的cpu缓存，此时这两个线程对各自的 count副本进行操作，进行加一操作分别写回内存，从而内存得到的结果是 1 ，而不是 2 。从而引发缓存的可见性问题。

评论内容收获：
加上volatile 关键字可以禁用缓存，运行速度会成倍下降
但是最终结果仍不会是最终的结果，因而解决该问题不是简单的缓存可见性问题，可能还有原子问题

线程切换带来的原子性问题

进程

任务切换： 经过一段固定的时间，操作系统会重新选择一个进程来执行。
早期的操作系统是基于进程来切换，不同进程之间不共享内存空间，因而进程的切换意味着要切换内存映射地址。

线程

现代操作系统是基于线程来进行任务切换的，同一进程创建的不同线程共享一个内存空间。
现在所提到的任务切换都是指线程切换。

bug产生原因

高级语言里的一条语句往往需要多条cpu指令来进行。例如加一操作：
指令 1 ：将变量从内存加载到cpu寄存器 指令 2 ：在寄存器中进行+1操作
指令 3 ：将结果写回内存（缓存机制导致可能写入的是cpu缓存而不是内存）
操作系统做任务切换，可以发生在任何一条*cpu指令*执行完成，而不是*高级语言里的一条语句*。
例如对于count做+1操作，线程A执行到指令1后发生线程切换，切换到线程B，线程B完成所有指令后再回到线程A, 线程A执行完之后就会将同样的+1值写回内存。

关键概念

原子性：一个或者多个操作在cpu执行的过程中不被中断的特性(cpu保证原子性操作是在cpu指令级别的，而不是 语句的操作符，也是违背常识，容易出bug的地方，因此很多时候需要在高级语言层面保证操作的原子性)。

源头之三： 编译优化带来的有序性问题

经典问题： 双重检查创建单例对象。

```
public class Singleton{
    private static Singleton instance;
```

```
    public static Singleton getInstance(){
        if(instance == null){
            synchronized (Singleton.class){
                if(instance == null){
                    // 可能由于编译器优化带来空指针异常
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

在new语句的时候，需要的操作是：

1. 分配一块内存地址
2. 在内存M上初始化Singleton对象
3. 将M的地址赋值给instance变量
   而经编译器优化后的执行路径可能是：
4. 分配一块内存地址
5. 将M的地址赋值给instance变量
6. 在内存M上初始化Singleton对象 因而就有可能在执行完2之后发生线程切换，但由于2先执行，因而在另一个线程引用未初始化的对象的成员变量时， 就有可能发生空指针异常。

思考题

在32位系统上进行long型的加减操作会带来并发问题：

1. 注意这里是针对java来说，因而long型数据长度是64位，而不是c语言中，在32位系统上是可能是4个字节即32位。
2. 正因为是固定的64位，在32位机器上，对数据的操作需要进行多条指令组合出来，无法保证原子性，因而带来并发问题。

总结

并发问题的诡异问题，通常是直觉欺骗了我们，在深入理解可见性、原子性和有序性问题后，很多并发bug都是可以理解和诊断的。
缓存、线程切换和编译器优化，都是为了提高程序性能，但是又带了其他问题，由此，在技术解决一个问题的同时，必然会带来其他 问题，我们要清楚问题是什么，以及如何规避。