并发安全

前置知识：shell脚本基本语法：重定向，变量，过程调用，循环

场景描述：小A和小B之间有大量的资金往来，为了避免手工记账的繁琐，他们委托系统工程师小C设计一个计算机系统来满足他们频繁的转账需求。小C将他们的账户以文件的形式存储在Linux系统中，并编写了如下的Shell脚本来实现转账功能：

```
#!/bin/bash
# origin.sh
# 初始化账户文件
echo "100" > A.txt
echo "100" > B.txt

# 转账函数
transfer() {
    echo "$4: $1 向 $2  支付 $3."
    a=$(tail -n 1 $1.txt)
    b=$(tail -n 1 $2.txt)
    ((a -= $3))
    ((b += $3))
    echo "$a" >> $1.txt
    echo "$b" >> $2.txt
}

# 并发执行转账操作
for ((i=0; i<100; i++)); do
    transfer A B 1 $i
    transfer B A 1 $i
done
wait
# 查看结果
a=$(tail -n 1 A.txt)
b=$(tail -n 1 B.txt)
echo "A 余额: $a; B 余额: $b; 总计: $(($a+$b))"
```

这份代码能够满足需求。但是小A和小B总是嫌弃程序太慢了。为了加速程序，小C向Linux大佬求助，大佬说可以利用`&`使命令转为后台执行，从而实现程序的并发：

```
#!/bin/bash
# background.sh
# 初始化账户文件
echo "100" > A.txt
echo "100" > B.txt

# 转账函数
transfer() {
    echo "$4: $1 向 $2  支付 $3."
    a=$(tail -n 1 $1.txt)
    b=$(tail -n 1 $2.txt)
    ((a -= $3))
    ((b += $3))
    echo "$a" >> $1.txt
```

```
    echo "$b" >> $2.txt
}

# 并发执行转账操作
for ((i=0; i<100; i++)); do
    transfer A B 1 $i &
    transfer B A 1 $i &
done
wait
# 查看结果
a=$(tail -n 1 A.txt)
b=$(tail -n 1 B.txt)
echo "A 余额: $a; B 余额: $b; 总计: $(($a+$b))"
```

这样一来程序的执行速度确实加快了不少。可是小C一看，这程序的结果完全不对啊，仔细检查发现。同时执行文件的写操作可能会导致并发冲突，小C编写了如下代码来测试：

```
#!/bin/bash
for ((i=0; i<10; i++)); do
    echo $i > test.txt & echo $(<test.txt)
done
```

这段代码同时进行读写操作，但是该程序的输出却是不连续、不确定的。这表明读取一个正在写入的文件时，可能返回一个错误的值（如空值）。于是小C继续求助大佬，如何避免读写冲突？大佬微微一笑，甩出一个`flock`：

```
#!/bin/bash
exec {lock_fd}>test.lock  # 创建文件锁
for ((i=0; i<100; i++)); do
    (
        # 获取文件锁
        flock $lock_fd
        # 写入数据
        echo $i > test.txt
        # 释放文件锁
        flock -u $lock_fd
    ) &
    echo $(<test.txt)     # 读取并输出数据
done
exec {lock_fd}>&-         # 销毁文件锁
```

一顿炫酷的操作让小C惊呼：是锁！他加了锁！小C随即将其应用到他的转账系统中：

```
#!/bin/bash
# flock.sh
echo "100" > A.txt
echo "100" > B.txt

# 转账函数
transfer() {
    echo "$4: $1 向 $2  支付 $3."
    exec {lockA}<>A.lock
    exec {lockB}<>B.lock
```

```
    # 获取文件锁
    flock $lockA
    flock $lockB
    # 执行操作
    a=$(<$1.txt)
    b=$(<$2.txt)
    ((a -= $3))
    ((b += $3))
    echo "$a" > $1.txt
    echo "$b" > $2.txt
    # 释放文件锁
    flock -u $lockA
    flock -u $lockB
}

for ((i=0; i<10; i++)); do
    transfer A B 1 $i &
    transfer B A 1 $i &
    echo ":: A 余额: $(<A.txt); B 余额: $(<B.txt)"
done
wait
```

这次程序的输出总是正确的了，但是一看运行耗时还不如顺序执行呢。小C这才发现Linux大佬是个水货，使用文件锁能避免读写互斥，但对于并发却毫无用处。小C再次分析程序，发现由于上锁机制保证了文件同时只能存在一个读者，其执行流程和顺序执行并无本质的区别。但此程序还要加上维护两个文件锁的性能负担，当然要比顺序执行慢了。Linux大佬还在满口难懂的话，什么“一切皆文件”，什么“管道先进先出”之类，终端内外充满了快活的空气。

思考：

- 在自己的shell环境中执行脚本并观察输出，尝试修改并调试程序。提示：对shell脚本不熟悉的同学，此时GPT/搜索引擎才是你的最好帮手。
- 如何比较脚本执行时间？提示：time指令
- 如果你是真正的Linux大佬，你该如何进一步优化需求？