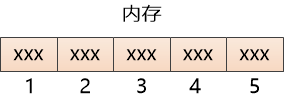
<https://mp.weixin.qq.com/s/31Pj6-wrMx8kjYoSy_b7FA>

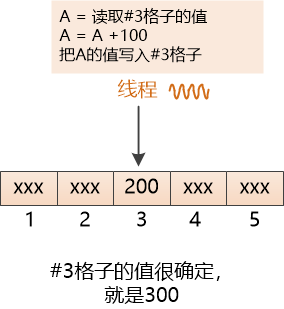
# 线程的麻烦事儿，Actor能解决吗？

冯诺伊曼体系中， CPU和内存居于核心的地位。

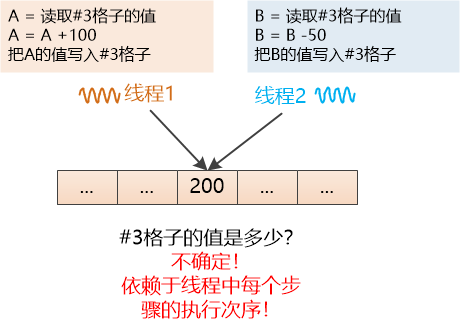
内存就像一个个的小格子，其中保存着程序要读写的值。



当只有一个线程来访问内存的时候，事情非常简单：



但是，当出现多线程的时候，就可能会出现互相覆盖的危险：



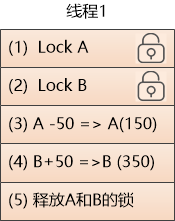
在多线程并发执行的情况下，为了得到正确的结果，必须要**加锁**。



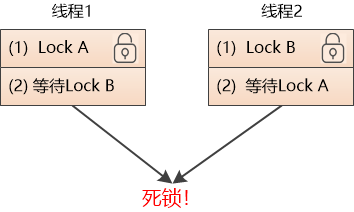
看起来加锁是一件轻松的事情， 但实际上并非如此， 让我们看一个转账的例子：有两个账户，账户A和账户B， 现在有一个线程1，要**从账户A给账户B转50元。**



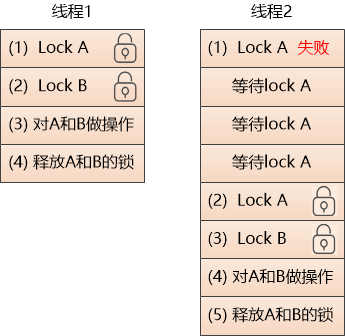
为了防止别的线程并发操作，相互覆盖，它需要加锁：



看起来没有任何问题， 但是如果还有个线程，同时要从账户B 给账户A转30元，它也采用了类似的加锁办法，就出问题了：



有个非常简单的办法来解决这个问题：**对账户按次序加锁** 。例如所有线程都是先对账户A加锁，然后对账户B加锁，这样死锁消除了。



看到了吧，多线程并发编程一不留神就会出错。

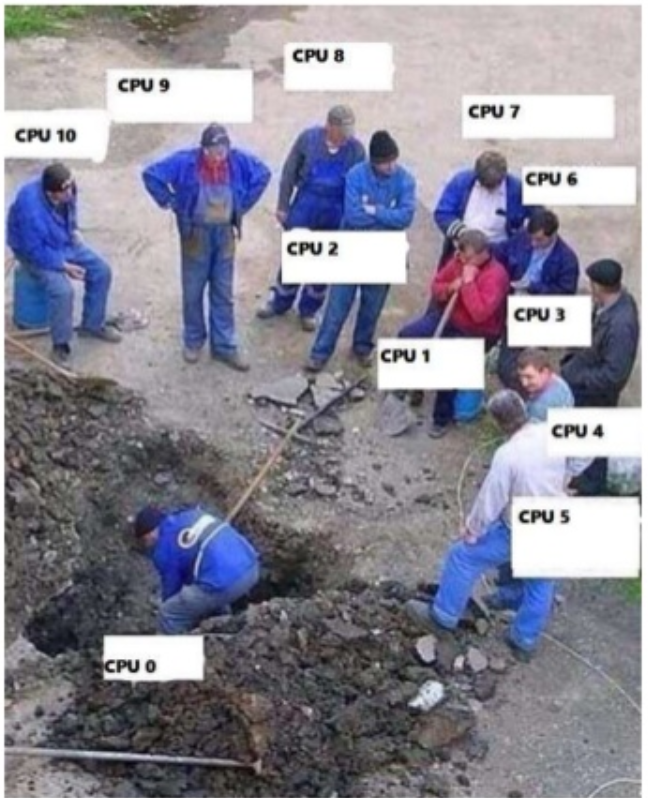
你以为多线程编程是这样：



实际上写出的程序是这样：

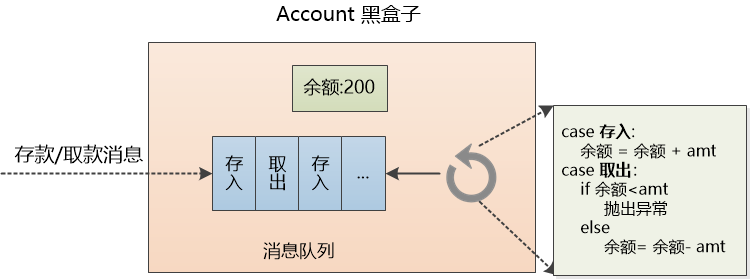


而CPU利用率很可能是“一核有难，众核围观”



锁这么麻烦，能不能不用锁？

换个思路，不把账户余额看成是简单的值，而是一个**黑盒子对象：**



在这个黑盒子中，保存了账户的余额200。

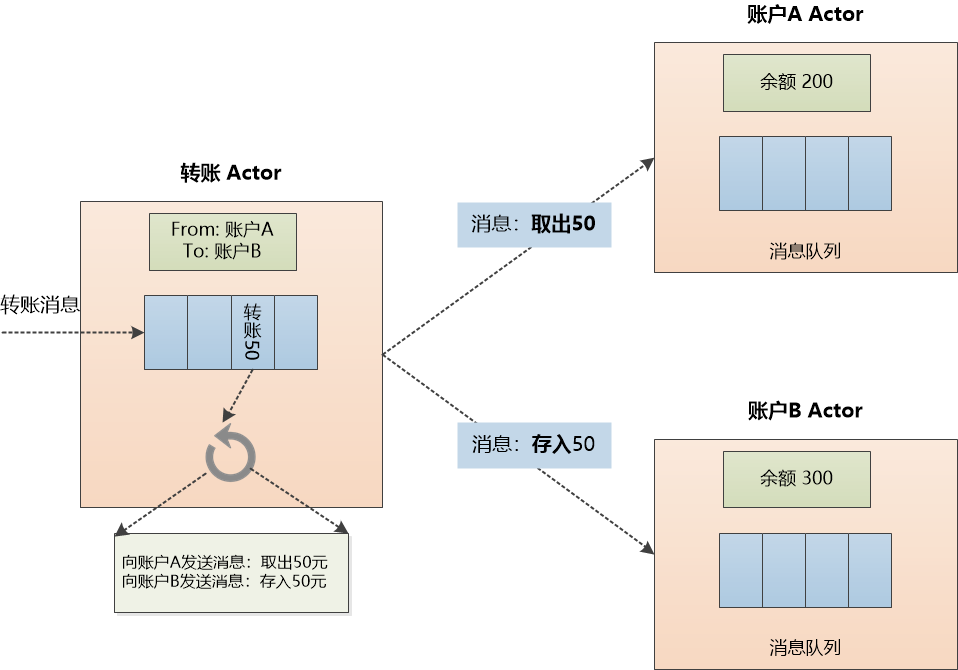
你想存款了，就发一个存款的消息过来，想取款就发一个取款的消息过来，发完消息就可以撤离了（异步操作）。

不管是有一个消息，还是有100个消息，统统放到黑盒子的一个队列中，然后让Account这个黑盒子一个个顺序处理， 对余额进行增加或减少。

外界无法看到余额，只能通过消息和这个黑盒子交互，黑盒子内部顺序处理，**自然不用加锁**。

这个黑盒子，就是**Actor**。

试一试用Actor方式来实现转账， 看看和之前有什么不同：



“转账 Actor” 收到转账50元的消息， 向账户A发送取出50元的消息， 向账户B发出存入50元的消息。

然后账户A 和 账户B 收到消息，进行处理。

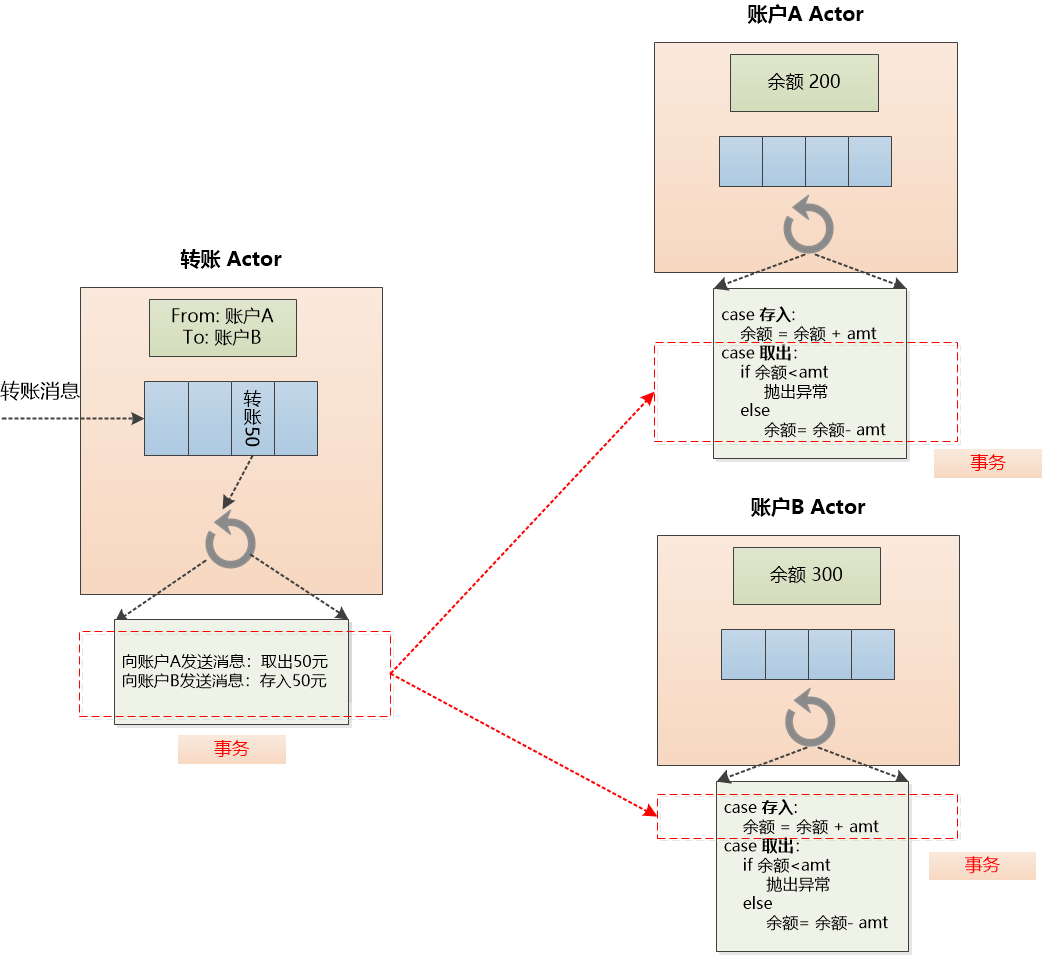
非常清晰，根本不用锁， 每个Actor都是独立的个体，它们之间靠消息交互就行了。

可是，在转账过程中，如果有别的线程对账户A也做了操作，导致账户A余额不足，抛出了异常， 但是账户B继续存入50元，那这个转账其实就出错了！

这时候，我们想到了什么？对，就是事务的**原子性**：要么不做，要么全做。

所以需要让这些Actor支持事务，这可真是有点麻烦，因为Actor之间是独立的，消息的发送是异步的，现在转账却要求存入和取出两个操作需要同步进行，并且满足原子性。

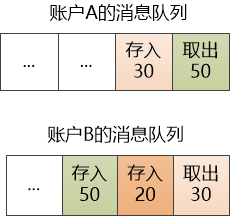
世界上果然没有免费的午餐。



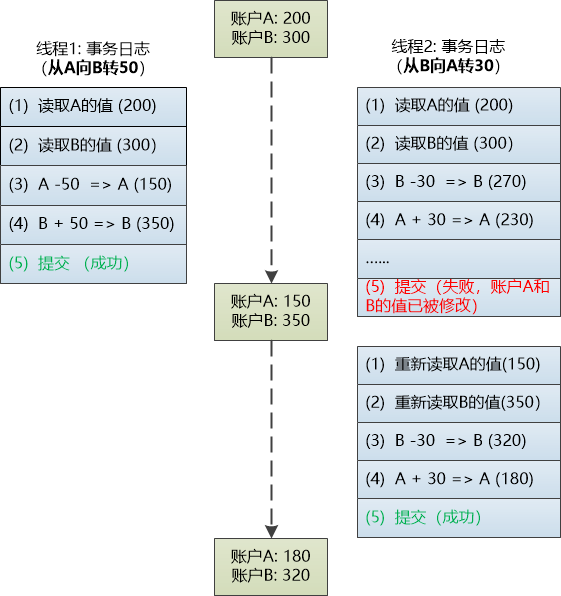
这里边要解决两个问题：

1.  账户A和账户B的Actor 需要互相等待，直到存入和取出的操作都完成，有任何一个没完成（如抛出异常），就要回滚。

2. 由于消息发送都是异步的，在执行一个事务的时候，可能有其他消息放入消息队列，并且被处理，账户A和账户B的余额不断地被改变，需要处理这种不断变化的情况。



可以参考下大名鼎鼎的**CAS**的原理，每个事务开始的时候，记录下原始的值，在提交的时候和当前值进行比较，如果相同，表示在这段时间内没有修改，提交成功。否则，读取当前的值，重做事务中的操作：



没有加锁，就是不断地在**尝试**，由于数据都是在内存中操作，频繁地尝试也不是什么大问题（在并发冲突不是很激烈的情况下）

用这种方式实现的事务， 做软件事务内存(Software Transactional Memory)，简称**STM**。

总结一下，Actor看起来简单，对于单个账户操作来说，是个很美妙的模型，因为账户之间是隔离的。 但是对于一致性要求比较高的场景，如转账，Actor模型就显得有些笨拙了。