# **Fail-Fast机制**

**由于HashMap（ArrayList）并不是线程安全的,因此如果在使用迭代器的过程中有其他线程修改了map(这里的修改是指结构上的修改,并非指单纯修改集合内容的元素)，那么将要抛出ConcurrentModificationException 即为fail-fast策略      
    主要通过modCount域来实现,保证线程之间的可见性,modCount即为修改次数,对于HashMap（ArrayList）内容的修改就会增加这个值， 那么在迭代器的初始化过程中就会将这个值赋值给迭代器的expectedModCount  
    但是fail-fast行为并不能保证,因此依赖于此异常的程序的做法是错误的**

**在面试中，我会经常问面试者什么是Fast Fail，但遗憾的是很多人都没有听说过这个概念。借这个空挡，写点东西描述一下。**

**在并发的时候，如果线程A正遍历一个collection(List, Map, Set etc.)，这时另外一个线程B却修改了该collection的size，线程A就会抛出一个错：ConcurrentModificationException，表明：我正读取的内容被修改掉了，你是否需要重新遍历？或是做其它处理？这就是fail-fast的含义。**

**Fail-fast是并发中乐观(optimistic)策略的具体应用，它允许线程自由竞争，但在出现冲突的情况下假设你能应对，即你能判断出问题何在，并且给出解决办法。**

**悲观(pessimistic)策略就正好相反，它总是预先设置足够的限制，通常是采用锁(lock)，来保证程序进行过程中的无错，付出的代价是其它线程的等待开销：**

**Vector v = …;**

**synchronized(v){**

**ListIterator iter = v.listIterator();**

**while(iter.hasNext()) {**

**Object o = iter.next();**

**//do something**

**...**

**}**

**}**

**可以瞧见，这个遍历过程被同步，当然不会有其它线程能修改v(即使Vector本身就是同步的)，决不会出现错乱、无法预知的结果。但这个同步块若是一个被频繁调用的模块，则其它线程等待的时间不会是个小数字，这时候，它就是一个单线程系统，可就看CPU的处理能力了：)**

**Doug Lea的concurrent包给出了另外一种解决方案，Copy On Write。它的CopyOnWriteArrayList、CopyOnWriteArraySet会在线程B试图修改数据容器时，给出一个copy出来的容器(当然，我们程序中是感觉不到的)，这样线程A在老版本的v上遍历，线程B则在新版本的v上修改，两者互不相干，也决不会出现ConcurrentModificationException。它的代价则主要是在容器的copy上，当并发程度越高，其开销也越高。**

**所以，Fast Fail被引入JDK的一个基本前提是：你绝大多数的情形，仅仅是在遍历一个collection，不会有另外的线程会对它做update。如此，它的效率是最充分的。但如果你不断遇到ConcurrentModificationException的异常时，则要考虑是否要进行一定次数的重新遍历，或者干脆采用悲观策略锁住资源来保证线程安全。**