原子变量

java.util.concurrent.atomic包下有一些原子变量，可以使数据处理天生安全，但是若并非直接使用(最起码可以保证数据是正确的，但结果却并非是我们想要的)

原子变量有：

AtomicBoolean

AtomicInteger

AtomicLong

AtomicReference [引用类型]

AtomicIntegerArray

AtomicReference [引用类型]

常用方法
AtomicReference(V initialValue) initialValue：引用对象
get() 返回当前的引用
compareAndSet(V expect, V update)如果当前引用相等，更新为指定的update值。
getAndSet(V newValue) 原子地设为给定值并返回旧值。
set(V newValue)注意此方法不是原子的。不明白为什么要提供这个方法，很容易误用。

引用类型中的数据还是得原子，否则还是不能保证引用对象中的数据安全

举例：

```
AtomicInteger s = new AtomicInteger(120);
s.getAndIncrement() //== s++;
s.getAndDecrement() //== s--;
s.getAndAdd(int delta) //== s+=delta;
s.weakCompareAndSet(int expect,int update) //expect == 当前值 update预期值
```

ConcurrentHashMap

简介:

一个线程安全的支持多线程并发访问的hashmap

位置:

java.util.concurrent【包下都是一些对线程并发的内容】

都知道hashmap线程是不安全的， java 1.5为了支持线程并发出现大量的进行并发的内容ConcurrentHashMap

，这个集合在1.8之前采取的是“锁分段”机制，就是说一个数组格子一把锁，多个线程可以同时对这个map进行操作，

若操作相同格子则等待。1.8之后采取 CAS 算法 + synchronized

特点：

分段锁，add remove 等对集合做修改操作的不一定会等待，get无需开启锁因为只是简单看一下数据

并发修改异常 ConcurrentModificationException

产生位置：

迭代器

产生原因：

迭代中对集合中数据进行增加或删除(删除倒数第二个元素永远不会报错)

解决方法：

若单线程迭代器中进行删除则使用迭代器提供的remove

若多线程迭代器中进行增删操作 则使用CopyOnWriteArrayList（无需再使用迭代器提供的remove，直接调用该集合自带的）

CopyOnWriteArrayList（线程安全，内部lock锁）底层原理

无论该集合做增删改或者说使用迭代器进行查询

该集合对数据操作时不会直接将元素增加或删除或修改到容器中，

而是拷贝老的容器诞生出新数组从而增加增加或删除或修改，耗费资源巨大

迭代器也是如此，内部迭代会将老的数组拷贝一份，若中途对集合进行增删改迭代器是不知情的

因为不是同一个数组，到最后增加完毕会老数组直接指向新数组

当希望读数，遍历 大于修改时CopyOnWriteArrayList 优于 同步的ArrayList(产生于Collcations中)