1.ThreadLocal类与Synchronized的区别:

Synchronized类实现内存共享，而ThreadLocal类是为每一个线程维护一个本地变量，这种变量在多线程环境下访问时能够保证各个线程里变量的独立性。

http://blog.csdn.net/danchu/article/details/68961461（ThreadLocal内存泄漏的问题）。

2.线程池的原理

在java中，当需要一个线程时就去创建一个线程，但当线程并发量过多的时候，并且一个线程都是执行很短的时间就结束了，这样频繁的创建和销毁线程需要大量的时间，有一种方法就是当线程实现完一个任务后，并不销毁，而是继续执行其他任务，实现线程的复用。

3.线程池的任务拒绝策略

原因是:线程池会维护一个任务队列，用于缓存待处理的任务，因此当任务队列的长度有限的情况下，需要一种机制处理当需要加入任务队列却任务队列已满的情况。

四种方式：

1：直接丢弃

2：丢弃队列中最老的任务

3：抛出异常（直接抛出异常，阻止系统工作）

4：将任务分配给调用线程使用（只要线程池未关闭，该策略直接在调用者线程中运行当前被丢弃的任务。显然这样不会真的丢弃任务，但是，调用者线程性能可能急剧下降。）

4.线程与进程的区别

调度：线程是CPU调度和分配的基本单位，进程是资源分配的基本单位。

并发性：不仅进程间可以并发执行，一个进程里的线程间也可以并发执行。

拥有资源：进程是拥有资源的一个独立单位，线程不拥有资源，但可以访问隶属于进程的资源

创建与销毁：进程相比线程来说比较庞大，创建和销毁的开销大。

5.线程安全与非线程安全集合

线程安全：当多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，会对这个数据进行保护，其他线程不能对其进行访问，直到这个线程访问完毕以后，其他线程才可以使用，防止数据出现不一致或者被污染的现象。

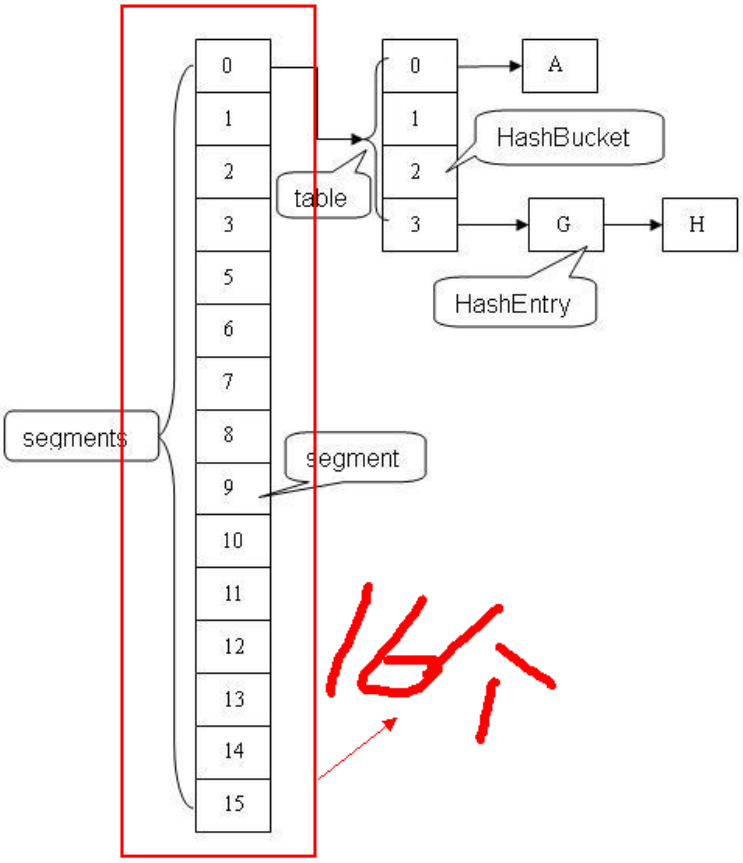
实现原理：对于JVM中有一个main memory，每一个线程有一个working memory，一个线程对一个变量进行操作时，会在自己的work memory中创建一个副本，操作完以后写入main

memory。
线程安全的集合有：vector、hashTable、StringBuffer

非线程安全：不提供数据访问时候的数据保护，多个线程同时操作某个数据时，可能会出现数据不一致或者数据污染的情况。
解决方案：一般用synchronized关键字加锁同步控制
工作原理：多个线程同时操作一个变量，会出现严重的结果，synchronized主要是监控一个monitor，这个monitor可以是修改的变量，也可以是其他自己认为合适的对象，然后通过给这个monitor加锁实现线程安全，在线程获得这个锁以后，要执行load到working memory到use，然后指派到存储再到main memory的过程，才会释放得到的锁，实现线程安全。
非线程安全集合：ArrayList、linkedList、hashMap、HashSet、TreeMap、TreeSet、StringBulider

6.HashMap与ConcurrentHashMap的底层实现。
HashMap本质数据加链表。根据key值获得hash值，然后计算出数组下标，如果多个key对应同一个下标，就用链表串起来，新插入的放在前面。
在hashMap进行put的时候，插入的元素超过了容量，就会触发扩容操作，在多线程的环境下，如果同时其他元素也在进行put操作，hash值相同的话，就可能在统一数据下用链表表示，造成闭环，导致在get时出现死循环。所以hashMap是线程不安全的。

ConcurrentHashMap的数据结构是由一个Segment和多个HashEntry组成



Segment数组的意义就是将一个大的table分隔成多个小的table来进行加锁，也就是锁分离技术，而每一个Segment元素存储的是HashEntry数组+链表，这个和HashMap的数据存储结构一样。当执行put操作时，会进行第一次key的hash来定位Segment的位置，然后进行第二次hash操作，找到相应HashEntry的位置，这里会继承过来的锁的特性，在将数据插入到指定的hashEntry位置时，会继承ReentrantLock的tryLock方法尝试去获取锁，如果获取成功就直接插入相应的位置，如果已经有线程获取该segment的锁，那当前线程会以自旋的方法去继续调用tryLock方法去获取锁，超过指定次数就会挂起。

7：可重入锁与非可重入锁的区别

锁分为可重入锁和不可重入锁，当一个线程获得了当前实例的锁，并进入方法A，这个线程在没有释放这把锁的时候，能否再次进入方法A.

可重入锁：可以再次进入方法A，就是说在释放锁前此线程可以再次进入方法A。

不可重入锁（自旋锁）：不可以再次进入方法A、

https://blog.csdn.net/soonfly/article/details/70918802

先举例来说明锁的可重入性：

```
public class UnReentrant{
    Lock lock = new Lock();
    public void outer(){
        lock.lock();
        inner();
        lock.unlock();
    }
    public void inner(){
        lock.lock();
        //do something
        lock.unlock();
    }
}
```

outer中调用了inner，outer先锁住了lock，这样inner就不能再获取lock。其实调用outer的线程已经获取了lock锁，但是不能在inner中重复利用已经获取的锁资源，这种锁即称之为 不可重入 。通常也称为 自旋锁 。相对来说，可重入就意味着：线程可以进入任何一个它已经拥有的锁所同步着的代码块。

这里的意思是线程进入outer后不能进入inner里面的//do something，因为outer获得了锁，进入inner后拿不到锁。

我们修改Lock，加入一个变量lockBy用来保存已经获得锁的线程，这样就能对有锁的线程放行。

```
public class Lock{
    boolean isLocked = false;
    Thread  lockedBy = null;
    int lockedCount = 0;
    public synchronized void lock() throws InterruptedException{
        Thread callingThread = Thread.currentThread();
        while(isLocked && lockedBy != callingThread){
            wait();
        }
        isLocked = true;
        lockedCount++;
        lockedBy = callingThread;
    }
    public synchronized void unlock(){
        if(Thread.curentThread() == this.lockedBy){
            lockedCount--;
            if(lockedCount == 0){
                isLocked = false;
                notify();
            }
        }
    }
}
```

解释一下程序中的两个变量：

lockBy：保存已经获得锁实例的线程，在lock()判断调用lock的线程是否已经获得当前锁实例，如果已经获得锁，则直接跳过while，无需等待。

lockCount：记录同一个线程重复对一个锁对象加锁的次数。否则，一次unlock就会解除所有锁，即使这个锁实例已经加锁多次了。

在java 中，synchronized和java.util.concurrent.locks.ReentrantLock是可重入锁。

其他线程不能拿到这个锁，同一线程可以多次拿到这个锁。