1.线程池

1.1 为何需要线程池

并发条件下，线程数量很多，如果每个线程都执行很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率。为了解决这一问题，线程池应运而生，线程持可以使得线程可以复用。本篇从线程持最核心的 ThreadPoolExecutor 类介绍。

1.2 Java 中的 ThreadPoolExecutor 类

1.构造

看最后一个就行了，都是重载

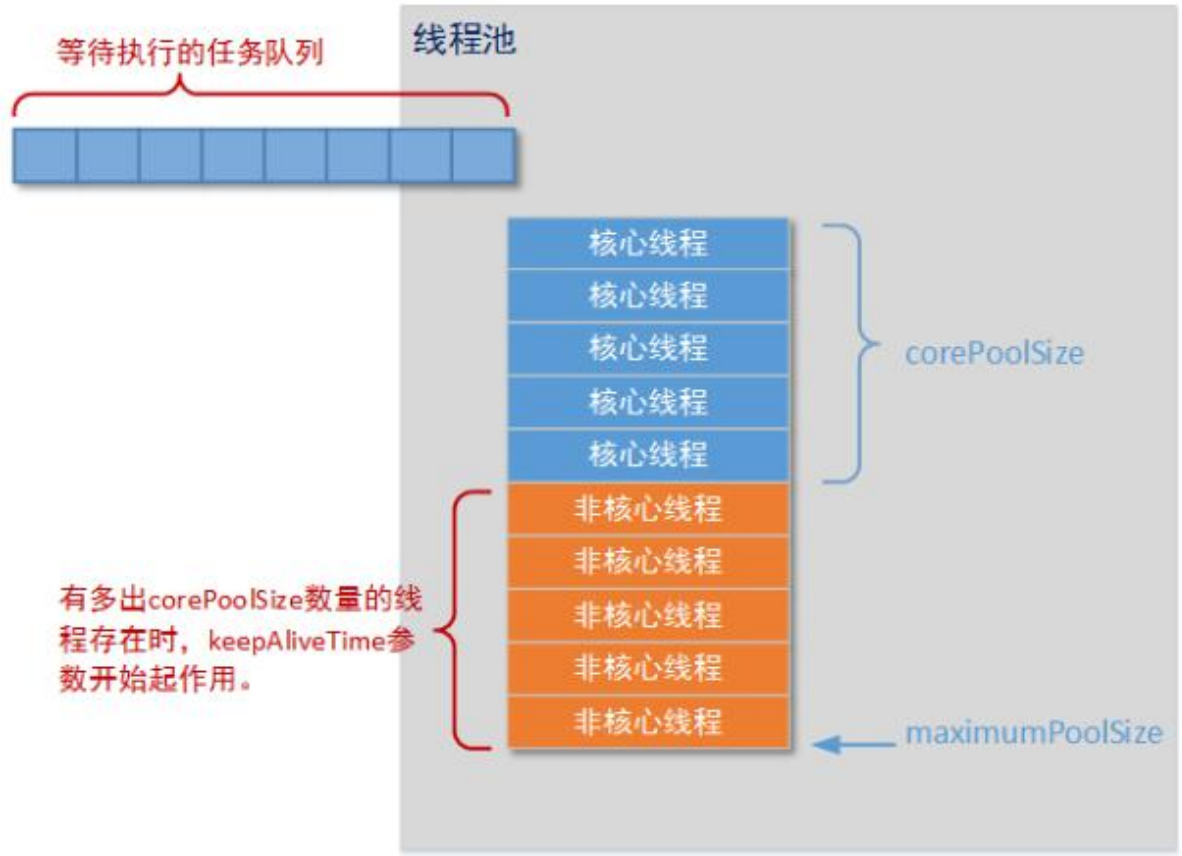
```
public class ThreadPoolExecutor extends AbstractExecutorService {
   .....
   public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,
         BlockingQueue<Runnable> workQueue);

   public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,
         BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory);

   public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,
         BlockingQueue<Runnable> workQueue,RejectedExecutionHandler handler);

   public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,
            BlockingQueue<Runnable>  workQueue,ThreadFactory  threadFactory,RejectedExecutionHandler
handler);
   ...
}
```

举例：创建一个 ThreadExecutor，先看一下执行全参构造后长什么样

```
BlockingQueue<Runnable>workQueue=new ArrayBlockingQueue<>(2);
ThreadFactory threadFactory=new ThreadFactory(){...}
ThreadPoolExecutor tExecutor=new ThreadPoolExecutor(
2,//corePoolSize,核心线程数
3,//maximunPoolSize,最大线程数
2,//keepAliveTime,线程空闲多长时间被释放
TimeUnit.SECONDS,//unit,时间单位
```

workQueue,//workQueue,任务队列，存放等待任务
threadFactory,//线程工厂
new ThreadPoolExecutor.CallerRunsPolicy());//拒绝执行任务的一种策略，CallerRunsPolicy 表示由调用者进行执行

2.构造器中各参数的含义



- **corePoolSize**：核心线程数。
  - 默认情况下，创建了线程池后，池中线程数为 0。当有任务来了，就创建一个线程去执行任务，当达到核心线程数，再来任务就会放到缓存队列。

    //如果调用了 prestartAllCoreThreads()或者 prestartCoreThread()方法（翻译:预创建线程），没有任务到　来就创建 corePoolSize 个线程或者一个线程。
- **maximumPoolSize**：最大线程数
  - 它表示线程池中最多可创建多少个线程。*在我看来是线程池的一种补救措施，即任务量突然过大时的一种补救措施，是额外创建的。*
- **keepAliveTime**：线程空闲多久被释放

- 默认下，只有当线程数大于 corePoolSize 才会起作用，直到<=corePoolSize，即当一个线程空闲时间达到 keepAliveTime，就被释放。

  //但是，如果调用了 allowCoreThreadTimeOut(boolean)方法，再线程池中的线程数不大于 corePoolSize 时，keepAliveTime 参数也会起作用，直到线程数为 0

- **unit**：参数 keepAliveTime 的时间单位，有 7 种取值，在 TimeUnit 类中有 7 中静态属性：

```
TimeUnit.DAYS;                  //天
TimeUnit.HOURS;                 //小时
TimeUnit.MINUTES;               //分钟
TimeUnit.SECONDS;               //秒
TimeUnit.MILLISECONDS;          //毫秒
TimeUnit.MICROSECONDS;          //微妙
TimeUnit.NANOSECONDS;           //纳秒
```

- **workQueue**：阻塞队列，用于存储等待执行的任务，有以下几种选择：

ArrayBlockingQueue;//基于数组的先进先出队列，此队列创建时必须指定大小；
LinkedBlockingQueue;//基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为 Integer.MAX_VALUE；
SynchronousQueue;//这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

  - *//ArrayBlockingQueue 和 PriorityBlockingQueue 使用较少，一般使用 LinkedBlockingQueue 和 Synchronous。线程池的排队策略与 BlockingQueue 有关。*

- threadFactory：线程工厂，主要用来创建线程；

- handler：表示当拒绝处理任务时的策略，有以下四种取值：

```
//==============任务拒绝策略==
ThreadPoolExecutor.AbortPolicy//丢弃任务并抛出 RejectedExecutionException 异常。
ThreadPoolExecutor.DiscardPolicy//也是丢弃任务，但是不抛出异常。
ThreadPoolExecutor.DiscardOldestPolicy//丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）
ThreadPoolExecutor.CallerRunsPolicy//由调用线程处理该任务
```

3.总结任务提交给线程池的过程

- *如果当前线程数小于 corePoolSize，则每来一个任务，就创建一个线程去执行*

- *如果大于 corePoolSize，则每来一个任务，会尝试将其添加到缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（指任务缓存队列 workQueue 已满），则会创建新的线程去执行这个任务*

- *如果当前线程数达到 maximumPoolSize，则会采取任务拒绝策略进行处理*
- *如果当前线程数大于 corePoolSize 时，当某线程空闲时间超过 keepAliveTime，将会被释放直到不大于 corePoolSize；设置存活时间则另当别论*

4.线程池的关闭

ThreadPoolExecutor 提供了两个方法，用于线程池的关闭，分别是 shutdowm()和 shutdownNow()，其中：

- *shutdown():不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接收新的任务。*
- *shutdownNow():立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务*

5.线程池容量的动态调整

ThreadPoolExecutor 提供了动态调整线程池容量大小的方法：setCorePoolSize()和 setMaximumPoolSize(),

- *setCorePoolSize：设置核心池大小*
- *setMaximumPoolSize：设置线程池最大能创建的线程数目大小*

当上述参数从小变大时，ThreadPoolExecutor 进行线程赋值，还可能立即创建新的线程来执行任务。

6.Java 提供的线程池

```
Executors.newCachedThreadPool();       //创建一个缓冲池，缓冲池容量大小为 Integer.MAX_VALUE
Executors.newSingleThreadExecutor();   //创建容量为 1 的缓冲池
Executors.newFixedThreadPool(int);     //创建固定容量大小的缓冲池
```

使用示例：

```
public static ExecutorService newFixedThreadPool(int nThreads) {
   return new ThreadPoolExecutor(nThreads, nThreads,
                                 0L, TimeUnit.MILLISECONDS,
                                 new LinkedBlockingQueue<Runnable>());
}
public static ExecutorService newSingleThreadExecutor() {
   return new FinalizableDelegatedExecutorService
```

```
    (new ThreadPoolExecutor(1, 1,
                    0L, TimeUnit.MILLISECONDS,
                    new LinkedBlockingQueue<Runnable>()));
}
public static ExecutorService newCachedThreadPool() {
  return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                     60L, TimeUnit.SECONDS,
                     new SynchronousQueue<Runnable>());
}
```

解析：

从它们的具体实现来看，它们实际上也是调用了 ThreadPoolExecutor，只不过参数都已配置好了。

newFixedThreadPool 创建的线程池 corePoolSize 和 maximumPoolSize 值是相等的，它使用的 LinkedBlockingQueue；

newSingleThreadExecutor 将 corePoolSize 和 maximumPoolSize 都设置为 1，也使用的 LinkedBlockingQueue；

newCachedThreadPool 将 corePoolSize 设置为 0，将 maximumPoolSize 设置为 Integer.MAX_VALUE，使用的 SynchronousQueue，也就是说来了任务就创建线程运行，当线程空闲超过 60 秒，就销毁线程。

实际中，如果 Executors 提供的三个静态方法能满足要求，就尽量使用它提供的三个方法，因为自己去手动配置 ThreadPoolExecutor 的参数有点麻烦，要根据实际任务的类型和数量来进行配置。

另外，如果 ThreadPoolExecutor 达不到要求，可以自己继承 ThreadPoolExecutor 类进行重写。

7. 如何合理配置线程池的大小（未能理解）

一般需要根据任务的类型来配置线程池大小；

CPU 密集型任务

尽量使用较小的线程池，一般为 CPU 核心数+1。因为 CPU 密集型任务使得 CPU 使用率很高，若开过多的线程数，会造成 CPU 过度切换。

IO 密集型任务

可以使用稍大的线程池，一般为 2*CPU 核心数。IO 密集型任务 CPU 使用率并不高，因此可以让 CPU 在等待 IO 的时候有其他线程去处理别的任务，充分利用 CPU 时间。

混合型任务

可以将任务分成 IO 密集型和 CPU 密集型任务，然后分别用不同的线程池去处理。 只要分完之后两个任务的执行时间相差不大，那么就会比串行执行来的高效。

因为如果划分之后两个任务执行时间有数据级的差距，那么拆分没有意义。

因为先执行完的任务就要等后执行完的任务，最终的时间仍然取决于后执行完的任务，而且还要加上任务拆分与合并的开销，得不偿失。

1.3 示例代码：

顺序：

1.打开 key1，线程 1 和 2 完成所有任务，偶尔出现线程 3——核心能完成不会创建额外线程

2.关闭 key1，打开 key2，线程 1，2 完成前两个任务，线程 3 完成剩余任务——忙不过来才创建

3.关闭 key1 和 key2，打开 key3，除了线程 123 还有 main 参与——CallerRunsPolicy 机制

4.关闭 key1 和 key2，打开 key3 和 key4，报错 RejectedExecutionException——任务超量

```
public class ThreadPoolTests {
public static void main(String[] args) throws InterruptedException {

BlockingQueue<Runnable> workQueue=new ArrayBlockingQueue<>(2);
ThreadFactory threadFactory=new ThreadFactory() {
AtomicLong al=new AtomicLong(1);//AtomicLong 线程安全的对象
@Override
public Thread newThread(Runnable r) {
//先 get，后加 1
String name="cgb2004-thread-"+al.getAndIncrement();
//先+1，再 get
//String name="cgb2004-thread-"+al.incrementAndGet();
return new Thread(r,name);}};

ThreadPoolExecutor tExecutor=new ThreadPoolExecutor(
2,//corePoolSize,核心线程数
3,//maximunPoolSize,最大线程数
2,//keepAliveTime,线程空闲多长时间被释放
TimeUnit.SECONDS,//unit,时间单位
```

```
workQueue,//workQueue,任务队列
threadFactory
//key4：
,new ThreadPoolExecutor.CallerRunsPolicy()//拒绝执行任务的一种策略，CallerRunsPolicy 表示由调用者进行执行
);

/**线程 1*/
tExecutor.execute(new Runnable() {
@Override
public void run() {
String tName=Thread.currentThread().getName();
System.out.println(tName+"->task-01");
try {Thread.sleep(3000);}catch(Exception e){e.printStackTrace(); }
}});

/**线程 2*/
tExecutor.execute(new Runnable() {
@Override
public void run() {
String tName=Thread.currentThread().getName();
System.out.println(tName+"->task-02");
try {Thread.sleep(3000);}catch(Exception e){e.printStackTrace(); }
}});

/**线程 3*/
tExecutor.execute(new Runnable() {
@Override
public void run() {
String tName=Thread.currentThread().getName();
System.out.println(tName+"->task-03");
//key3：
//                    try {Thread.sleep(3000);}catch(Exception e){e.printStackTrace(); }
}});

/**线程 4*/
tExecutor.execute(new Runnable() {
@Override
public void run() {
String tName=Thread.currentThread().getName();
System.out.println(tName+"->task-04");
}
});

//          Thread.sleep(3000);//key1：2 个任务，任务队列放的下，不会创建额外线程
```

```
/**线程 5*/
tExecutor.execute(new Runnable() {
@Override
public void run() {
String tName=Thread.currentThread().getName();
System.out.println(tName+"->task-05");
//                          try {Thread.sleep(5000);}catch(Exception e){e.printStackTrace(); }
}});

//              Thread.sleep(3000);//key2：3 个任务，任务队列放不下，不会创建额外线程

/**线程 6*/
tExecutor.execute(new Runnable() {
@Override
public void run() {
String tName=Thread.currentThread().getName();
System.out.println(tName+"->task-06");
}
});

/**线程 n*/
tExecutor.execute(new Runnable() {
@Override
public void run() {
String tName=Thread.currentThread().getName();
System.out.println(tName+"->task-07");
}
});

/**线程 n*/
tExecutor.execute(new Runnable() {
@Override
public void run() {
String tName=Thread.currentThread().getName();
System.out.println(tName+"->task-07");
}
});
}
}
```

2.线程

2.1 概念

线程：*进程中负责程序执行的执行单元。一个进程中至少有一个线程。*

多线程：*解决多任务同时执行的需求，合理使用 CPU 资源。多线程的运行是根据 CPU 切换完成，如何切换由 CPU 决定，因此多线程具有不确定性。*

线程和进程的区别：*一个进程是一个独立的运行环境可以被看做一个程序或一个应用。而线程是在进程中执行的一个任务。线程是进程的子集，一个进程有>=1 个线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存区域。每个线程都拥有单独的占内存来存储本地数据。*

2.2 创建线程的几种方式

2.2.1 继承 Thread 类，重写 run( ) 方法；创建线程对象并用 start( ) 方法启动线程

```
class DemoThread extends Thread {

    @Override
    public void run() {
        super.run();
        // Perform time-consuming operation...
    }
}

DemoThread t = new DemoThread();
t.start();
```

2.2.2 实现 Runnable 接口

```
public class DemoActivity extends BaseActivity implements Runnable {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        Thread t = new Thread(this);
        t.start();
    }

    @Override
    public void run() {

    }
}
```

选择继承 Thread 类还是实现 Runnable 接口?

Java 不支持类的多重继承，但允许实现多个接口。如果要继承其他类，后者更灵活一些。

2.2.3 实现 Callable (了解)

```
public interface Runnable {
    public void run();
}

public interface Callable<V> {
    V call() throws Exception;
}
```

重写 call 方法

```java
import java.util.concurrent.Callable;

public class Hello implements Callable {
    @Override
    public Object call() throws Exception {
        return null;
    }
}
```

Runnable 和 Callable 区别：

两者都代表要在不同线程中执行的任务。主要区别时 Callable 的 call 方法可以返回值（装载有计算结果的 Future 对象）和抛出异常，Runnable 的 run 方法不能。

可以得出：

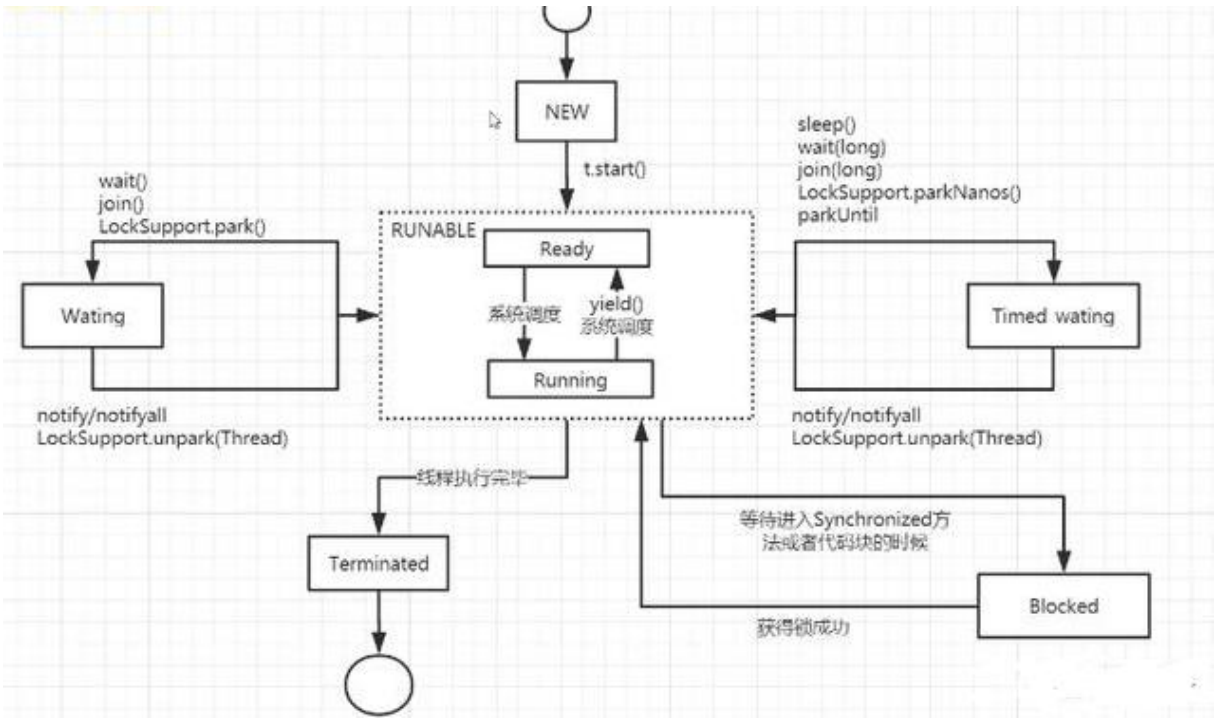
1）Callable 接口下的方法是 call()，Runnable 接口的方法是 run()。

2）Callable 的任务执行后可返回值，而 Runnable 的任务是不能返回值的。

3）call() 方法可以抛出异常，run()方法不可以的。

4）运行 Callable 任务可以拿到一个 Future 对象，表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。通过 Future 对象可以了解任务执行情况，可取消任务的执行，还可获取执行结果。

2.3 线程的状态



1.新建状态(New)：

当用 new 操作符创建一个线程时， 例如 new Thread(r)，线程还没有开始运行，此时线程处在新建状态。 当一个线程处于新生状态时，程序还没有开始运行线程中的代码

2.就绪状态(Runnable)

一个新创建的线程并不自动开始运行，要执行线程，必须调用线程的 start()方法。当线程对象调用 start()方法即启动了线程，start()方法创建线程运行的系统资源，并调度线程运行 run()方法。当 start()方法返回后，线程就处于就绪状态。

处于就绪状态的线程并不一定立即运行 run()方法，线程还必须同其他线程竞争 CPU 时间，只有获得 CPU 时间才可以运行线程。因为在单 CPU 的计算机系统中，不可能同时运行多个线程，一个时刻仅有一个线程处于运行状态。因此此时可能有多个线程处于就绪状态。对多个处于就绪状态的线程是由 Java 运行时系统的线程调度程序(thread scheduler)来调度的。

3.运行状态(Running)

当线程获得 CPU 时间后，它才进入运行状态，真正开始执行 run()方法.

4. 阻塞状态(Blocked)

线程运行过程中，可能由于各种原因进入阻塞状态:

1>线程通过调用 sleep 方法进入睡眠状态；

2>线程调用一个在 I/O 上被阻塞的操作，即该操作在输入输出操作完成之前不会返回到它的调用者；

3>线程试图得到一个锁，而该锁正被其他线程持有；

4>线程在等待某个触发条件；

……

所谓阻塞状态是正在运行的线程没有运行结束，暂时让出 CPU，这时其他处于就绪状态的线程就可以获得 CPU 时间，进入运行状态。

5. 死亡状态(Dead)

有两个原因会导致线程死亡：

1) run 方法正常退出而自然死亡，

2) 一个未捕获的异常终止了 run 方法而使线程猝死。

为了确定线程在当前是否存活着（就是要么是可运行的，要么是被阻塞了），需要使用 isAlive 方法。如果是可运行或被阻塞，这个方法返回 true； 如果线程仍旧是 new 状态且不是可运行的， 或者线程死亡了，则返回 false.

3.多线程

3.1 概念

解决多任务同时执行的需求，合理使用 CPU 资源。多线程的运行是根据 CPU 切换完成，如何切换由 CPU 决定，因此多线程具有不确定性。

优点：

1.适当提高程序的执行效率（多个线程同时执行）；

2.适当提高程序的资源利用率

缺点：

1.占用一定的内存空间

2.线程越多，CPU 的调度开销越大

3.程序的复杂度会上升

3.2 synchronized

同步块，通过 synchronized 关键字来实现；所有加上 synchronized 的方法和块语句，在多线程访问的时候，同一时刻只能由一个线程能够访问。

3.3 sleep 和 wait 的区别 ※

- 这两个方法来自不同的类，sleep 来自 Thread 类，wait 来自 Object 类
- sleep 方法没有释放锁(*我完事前谁也别动!*)，而 wait 方法释放了锁(*我等一会你们先走*)，使得其他线程可以使用同步控制块或者方法
- 使用范围：wait，notify，notifyAll 只能在同步控制方法或者同步控制块里面使用，而 sleep 可以在任何地方使用
- sleep 必须捕获异常，wait，notify，notifyAll 则不需要

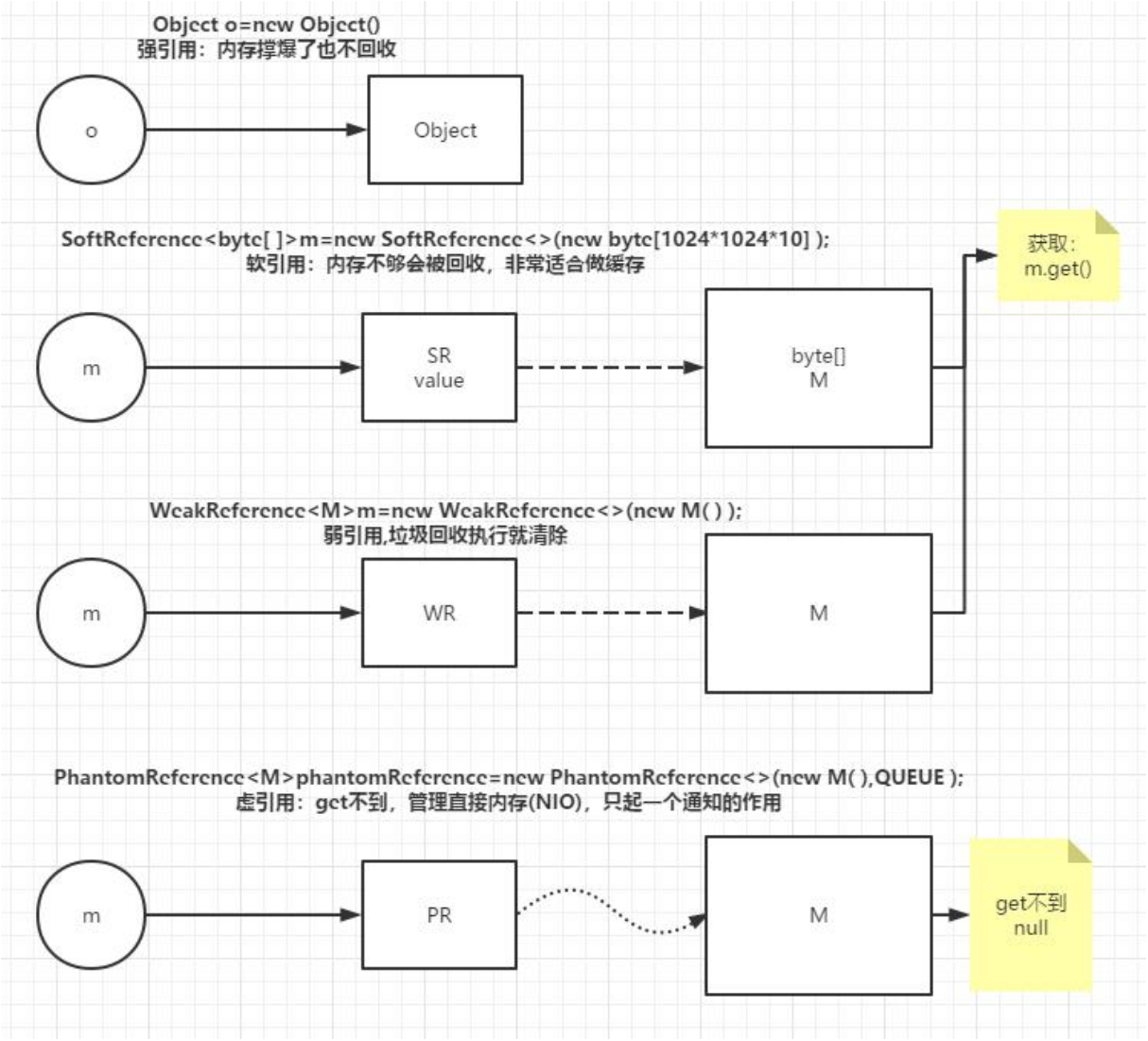
3.3.1 参考代码

```
public class Test {

//main 函数
public static void main(String[] args) {
new Thread(new Thread1()).start();

//线程 1 开启后，main 线程等待 5 秒
try {Thread.sleep(5000);}catch (Exception e){e.printStackTrace(); }

//              Test.class.wait();//不可以放在除同步代码块以外的地方
new Thread(new Thread2()).start();
```

```
}

//      线程 1
private static class Thread1 implements Runnable{
@Override
public void run() {
synchronized (Test.class){
System.out.println("enter thread1...");
System.out.println("thread1 is waiting...");

//调用 wait 方法，处于等待状态
try{Test.class.wait();}catch (Exception e){e.printStackTrace();}

System.out.println("thread1 is going on ....");
System.out.println("thread1 is over!!!");
}}}

//      线程 2
private static class Thread2 implements Runnable{
@Override
public void run() {
synchronized (Test.class){
System.out.println("enter thread2...");
System.out.println("thread2 is sleep...");

//进入 sleep 前唤醒线程 1
Test.class.notify();
try {Thread.sleep(5000);}catch (Exception e){e.printStackTrace();}

System.out.println("thread2 is going on ....");
System.out.println("thread2 is over!!!");
}}}}
```

4.ThreadLocal

4.1 前提：四种引用



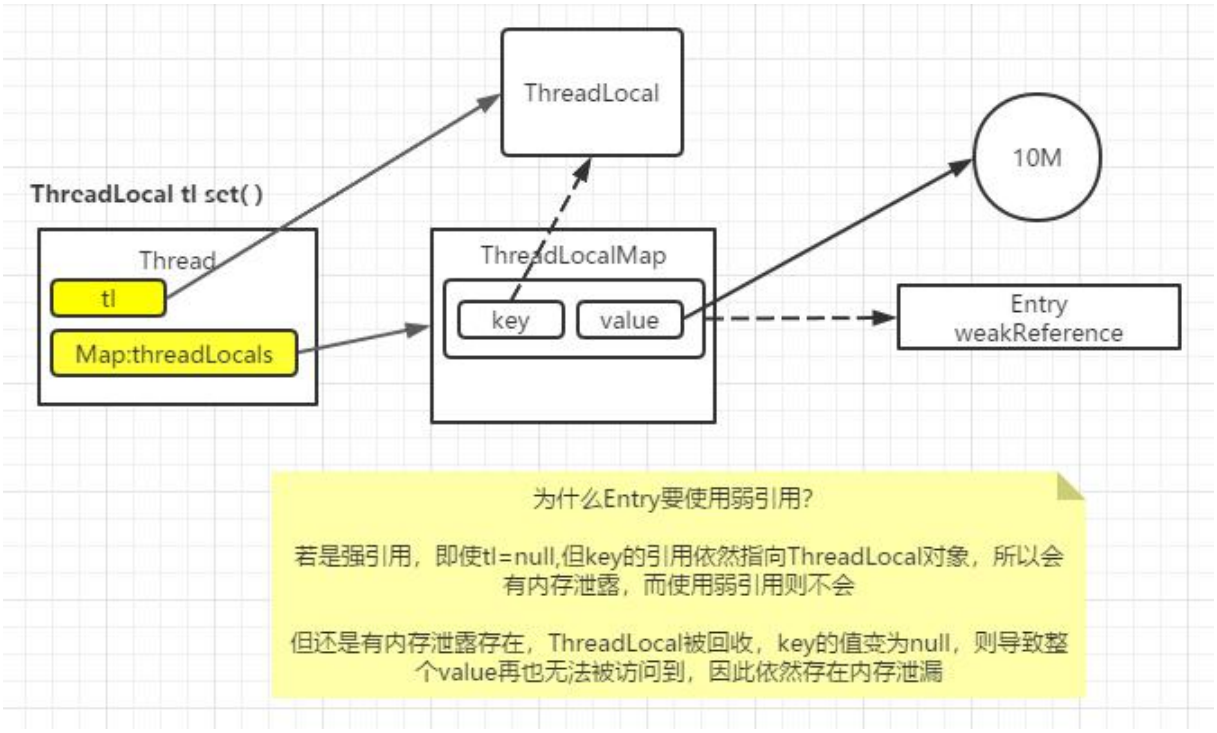
4.1.1 参考代码

```
public class T02_SoftReference {
public static void main(String[] args) {
SoftReference<byte[]>m=new SoftReference<>(new byte[1024*1024*10]);//需要把 Xmx 设置为 25M
System.out.println(m.get());//[B@1540e19d
System.gc();
/**sleep 是为了防止程序执行结束时未完成 GC 回收*/
try {Thread.sleep(500);}catch(InterruptedException e){e.printStackTrace();}
System.out.println(m.get());//[B@1540e19d
/**再分配一个数组，heap 将装不下，这时候系统会垃圾回收，
```

```
先回收一次，如果不够，会把软引用回收*/
byte[] b = new byte[1024 * 1024 * 15];
System.out.println(m.get());//null
}}

public class T03_WeakReference {
public static void main(String[] args) {
WeakReference<M> m=new WeakReference<>(new M());

System.out.println(m.get());//reference.M@1540e19d
System.gc();//gc 只要看见弱引用就回收
System.out.println(m.get());//null
}}

public class T03_PhantomReference {
private static final List<Object> LIST=new LinkedList<>();
private static final ReferenceQueue<M> QUEUE=new ReferenceQueue<>();
public static void main(String[] args) {
PhantomReference<M> phantomReference=new PhantomReference<>(new M(), QUEUE);
System.out.println(phantomReference.get());//null,回收后放入 QUEUE 队列里，JVM 需要单独处理的对
象

new Thread(()->{
while (true){
LIST.add(new byte[1024*1024]);
try{Thread.sleep(500);}catch(InterruptedException e){e.printStackTrace();}
System.out.println(phantomReference.get());
}}).start();

new Thread(()->{
while (true){
Reference<? extends    M> poll=QUEUE.poll();
if (poll!=null){
System.out.println("---虚引用对象被 jvm 回收了----"+poll);
}}}).start();
try{Thread.sleep(500);}catch(InterruptedException e){e.printStackTrace();}
}}
```

4.2 ThreadLocal

ThreadLocal 继承弱引用。ThreadLocalMap 是 ThreadLocal 的一个静态内部类。



解析：当执行 tl=new TL，调用 .set 方法时，首先获取当前线程，线程里面有一个 map(tl 为 key，set 的对象作为 value)，这里的 key 是弱引用指向 TL，因为 tl 指向 TL 的引用消失时，弱引用会防止 TL 造成的内存泄漏，这样在回收之后 key 变成空值，对应的 value 访问不到，所以还要有良好的编程习惯(get,set 也会将 value 置空，但是不一定调用)，调用 remove 回收掉 value，防止 value 的内存泄露；如果该线程位于线程池里，工作线程完成之后务必清理本地的 threadLocals，因为如果不清理，下次再用的时候用的是上次用的值，非常有可能产生各种各样的混淆。

总结:

（1）每个 Thread 维护着一个 ThreadLocalMap 的引用

（2）ThreadLocalMap 是 ThreadLocal 的内部类，用 Entry 来进行存储

（3）ThreadLocal 创建的副本是存储在自己的 threadLocals 中的，也就是自己的 ThreadLocalMap。

（4）ThreadLocalMap 的键值为 ThreadLocal 对象，而且可以有多个 threadLocal 变量，因此保存在 map 中

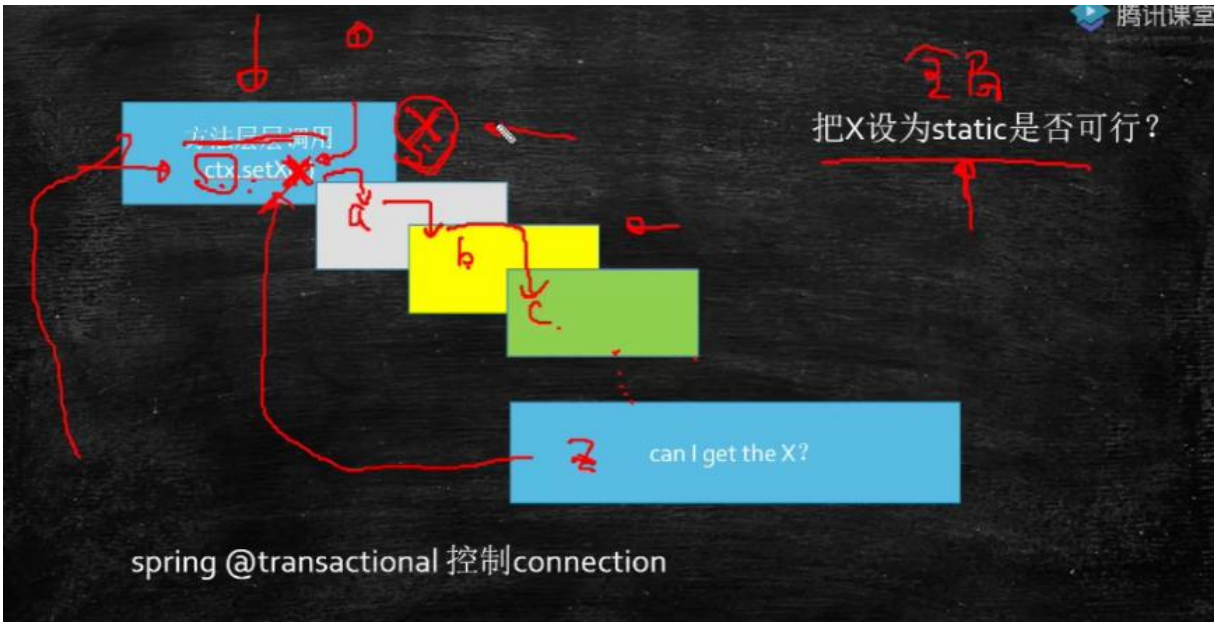
（5）在进行 get 之前，必须先 set，否则会报空指针异常，当然也可以初始化一个，但是必须重写 initialValue() 方法。

（6）ThreadLocal 本身并不存储值，它只是作为一个 key 来让线程从 ThreadLocalMap 获取 value。

4.2.1 代码参考

```
public class ThreadLocal2 {
static ThreadLocal<Person> tl=new ThreadLocal<>();

public static void main(String[] args) {
new Thread(()->{
try{TimeUnit.SECONDS.sleep(2);}catch(InterruptedException e){e.printStackTrace();}
System.out.println(tl.get());//null，获取不到其他线程的 ThreadLocal
}).start();

new Thread(()->{
try{TimeUnit.SECONDS.sleep(2);}catch(InterruptedException e){e.printStackTrace();}
tl.set(new Person());//这里 tl 为 key,new Person()为 value
}).start();
}
static class Person{
String name="zhangsan";
}}
```

4.3 ThreadLocal 的应用场景



参考资料：

https://www.jianshu.com/p/7726c70cdc40

https://www.cnblogs.com/dolphin0520/p/3932921.html

https://blog.csdn.net/weixin_40271838/article/details/79998327 ※

https://blog.csdn.net/xingjing1226/article/details/81977129 ※ 线程的状态

https://www.bilibili.com/video/BV1fA411b7SX?from=search&seid=3157220192589627639 马

https://www.jianshu.com/p/6fc3bba12f38 ThreadLocal 作用、场景、原理

https://baijiahao.baidu.com/s?id=1653790035315010634&wfr=spider&for=pc