并发与多线程

线程安全

线程是CPU调度和分派的基本单位。

线程生命周期：

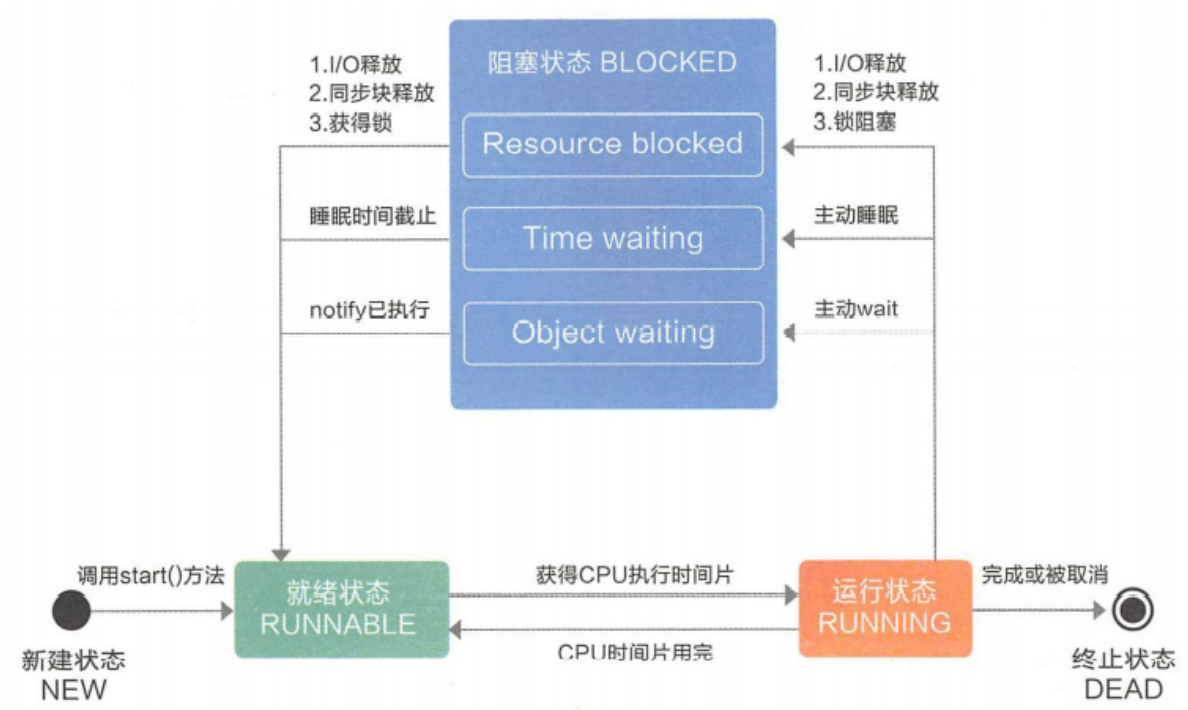


图 7-2　线程状态图

1. **NEW（新建状态）**

   线程被创建但未启动的状态。

   创建线程的方式有三种：继承自Thread类，实现Runnable接口，实现Callable接口。一般不推荐第一种因为不符合里氏代换原则。

   Callable与Runnable有两点不同：

   - Callable可以通过call()获得返回值，第一种和第二种需要借助共享变量获取。
   - Call()可以抛出异常，Runnable只能通过setDefaultUncaughtExceptionHandler()的方式在能在主线程中捕捉到子线程的异常。

2. **RUNNABLE（就绪状态）**

   调用start()之后，运行之前的状态。线程的start()不能被多次调用，否则抛出异常。

3. **RUNNING（运行状态）**

   run()正在执行时线程的状态。线程可能因为某些因素退出RUNNING。如时间、异常、锁、调度等。

4. **BLOCKED（堵塞状态）**

   进入此状态，有以下情况：

   - 同步堵塞：锁被其他线程占用。
   - 主动堵塞：调用Thread的某些方法，主动让出CPU执行权，比如sleep()、join()等。
   - 等待堵塞：执行了wait()。

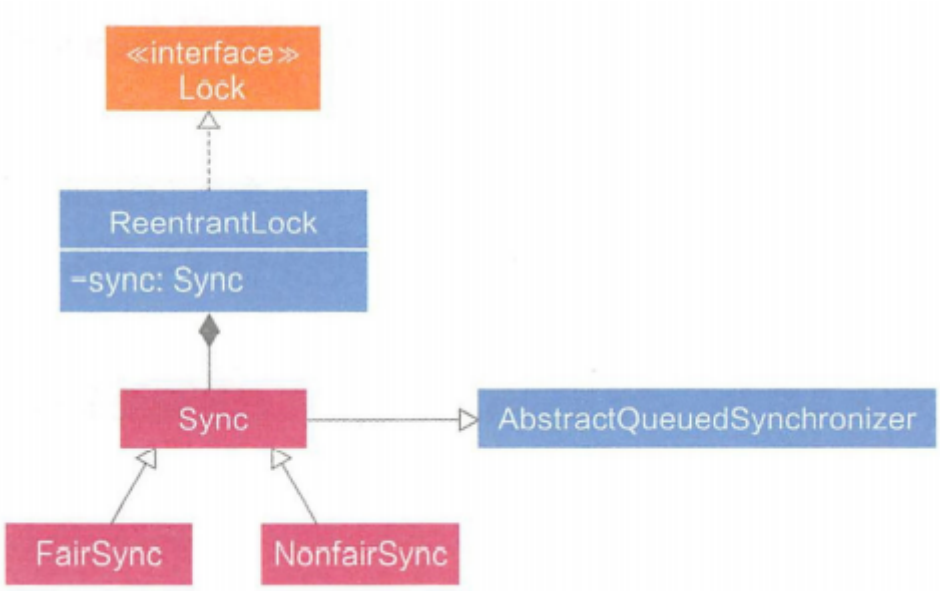
5. **DEAD（终止状态）**

   run()执行结束，或因异常退出后的状态，此状态不可逆转。

锁

并发包中的锁类

锁的继承关系



AQS中，定义了一个volatile int state变量作为共享资源，如果线程获取资源失败，则进入同步FIFO队列中等待；如果成功获取资源就执行临界区代码。执行完释放资源时，会通知同步队列中的等待线程来获取资源后出队并执行。

AQS是抽象类，内置自旋锁实现的同步队列，封装入队和出队的操作，提供独占、共享、中断等特性的方法。

1. ReentranLock：定义state为0时可以获取资源并置为1.若已获得资源，state不断加1，释放资源时state减1，直至减为0。
2. CountDownLatch初始时定义了资源总量state=count，countDown()不断将state减1，当state=0时才能获得锁，释放后state就一直为0。所有线程调用wait()都不会等待，所以CountDownLatch是一次性的，用完后如果想再用就只能重新创建一个。如果希望循环使用，可以使用基于ReentrantLock实现的CyclicBarrier。
3. Semphore同样定义了资源总量state=permits，当state>0就能获得锁，并将state减1；当state=0时只能等待其他线程释放锁，释放锁时state加1。当Semphore的permits定义为1时，就是互斥锁，大于1就是共享锁。

同步代码块

JVM底层通过监视锁来实现synchronized同步。监视锁即monitor，每个对象与生俱来的一个隐藏字段。使用synchronized时，JVM会根据synchronized的当前使用环境，找到对应对象的monitor，再根据monitor的状态进行加、解锁的判断。

- **偏向锁**

  为了在资源没被多线程竞争的情况下尽量减少锁带来的性能开销。在锁对象的对象头有一个 ThreadId 字段。 当第一个线程访问锁时， 如果该锁没有被其他线程访问过，即ThreadId 字段为空，那么 JVM 让其持有偏向锁，并将 ThreadId字段的值设置为该线程的ID。当下一次获取锁时，会判断当前线程的 ID 否与锁对象的 ThreadId 一致。如果一致，那么该线程不会再重复获取锁，从

而提高了程序的运行效率。如果出现锁的竞争情况，那么偏向锁会被撤销并升级为轻量级锁。如果资源的竞争非常激烈，会升级为重量级锁。偏向锁可以降低无竞争开销，它不是互斥锁 ，不 存在线程竞争的情况，省去再次同步判断的步骤，提升了性能。

锁升级

轻量级锁和重量级锁：重量级锁的调度是由操作系统而非JVM完成的

无锁 -> 偏向锁 -> 轻量级锁（CAS，自旋）-> 重量级锁

偏向锁不是一把锁，只是一个标记。偏向第一个竞争锁的线程。把自己的线程ID记录到对象头的markword里，重入时如果ID相同则省去锁竞争的过程。

偏向锁一旦遇到竞争，锁撤销并且升级成轻量级。

轻量级锁自旋超过一定次数升级成重量级锁。

线程同步

- **volatile**

  线程本地内存保存了引用变量在堆内存中的副本，线程对变量的所有操作都在本地内存区域中进行，执行结束后在同步到堆内存中去。

  使用volatile修饰变量时，意味着任何对此变量的操作都会在内存中进行，不会产生副本，以保证共享变量的可见性，局部组织了指令重排的发生。

  volatile解决的是多线程共享变量的可见性问题，类似synchronized，但不具备synchronized的互斥性。所以对volatile变量的操作并非都具有原子性。因此，**“volatile是轻量级的同步方式”这种说法是错误的。**它只是轻量级的线程操作可见方式，并非同步方式，如果是多写场景，一定会产生线程安全问题。如果是一写多读的并发场景则非常合适。

- **信号量同步**

  是指在不同的线程之内，通过传递同步信号量来协调线程执行的先后次序。

  Semaphore信号同步类，只有在调用Semaphore对象的acquire()成功后，才可以往下执行，完成后执行release()释放持有的信号量，下一个线程就可以马上获取这个空闲信号量进入执行。**只要其他线程能够拿到空闲信号量，都可以马上执行，如果Semaphore窗口信号量等于1，就是最典型的互斥锁**

- **CyclicBarrier**

  基于同步到达某个点的信号量触发机制。从命名上可知它是一个可以循环使用的屏障式多线程协作方式。比如三个人同时进入安检，只有3个人都完成安检，才会放下一批进来。

线程池

- 利用线程池管理并复用线程、控制最大并发数
- 实现任务线程队列缓存策略和拒绝机制
- 实现某些与实践相关的功能，如定时执行、周期执行等
- 隔离线程环境，避免各服务线程相互影响

生命周期：

- RUNNING 能够接收新任务，以及对已添加任务进行处理
- SHUTDOWN 不接受新任务，但能处理已添加任务
- STOP 不接受新任务，不处理已添加任务，并且会中断正在处理的任务

- TIDYING 当所有的任务已终止，ctl记录的“任务数量”为0，线程池变为TIDYING状态，并执行钩子函数terminated()。
- TERMINATED 线程池彻底终止，变为TERMINATED状态。

Executor框架

Executor框架不仅包括了线程池的管理，还提供了线程工厂、队列以及拒绝策略等。

- 框架结构：
  1. 任务（Runnable/Callable）

     执行任务需要实现的Runnable接口或Callable接口

  2. 任务的执行（Executor）

     包括任务执行机制的核心接口 `Executor` ，以及继承自 `Executor` 接口的 `ExecutorService` **接口。** `ThreadPoolExecutor` 和 `ScheduledThreadPoolExecutor` 这两个关键类实现了 **ExecutorService 接口。**

  3. 异步计算的结果（Future）

     当我们把Runnable接口或Callable接口的实现类提交给ThreadPoolExecutor或ScheduledThreadPoolExecutor执行，调用submit()方法时会返回一个FutureTask对象

ThreadPoolExecutor类

参数：

- corePoolSize（重要）：常驻核心线程数。等于0，任务执行完，没有任何请求进入时销毁线程池线程。大于0，即使本地任务执行完毕，核心线程也不会销毁。
- maximumPoolSize（重要）：表示线程池能够容纳同时执行的最大线程数。
- keepAliveTime：线程池中的线程空闲时间，当空闲时间达到keepAliveTime值时，线程会被销毁，直到只剩下corePoolSize个线程为止。
- workQueue（重要）：缓存队列。请求的线程数大于maximumPoolSize时，线程进入BlockingQueue阻塞队列。
- threadFactory：线程工厂，用来生产一组相同任务的线程。
- handler：拒绝策略，当提交的任务过多而不能及时处理时，我们可以定制策略来处理任务

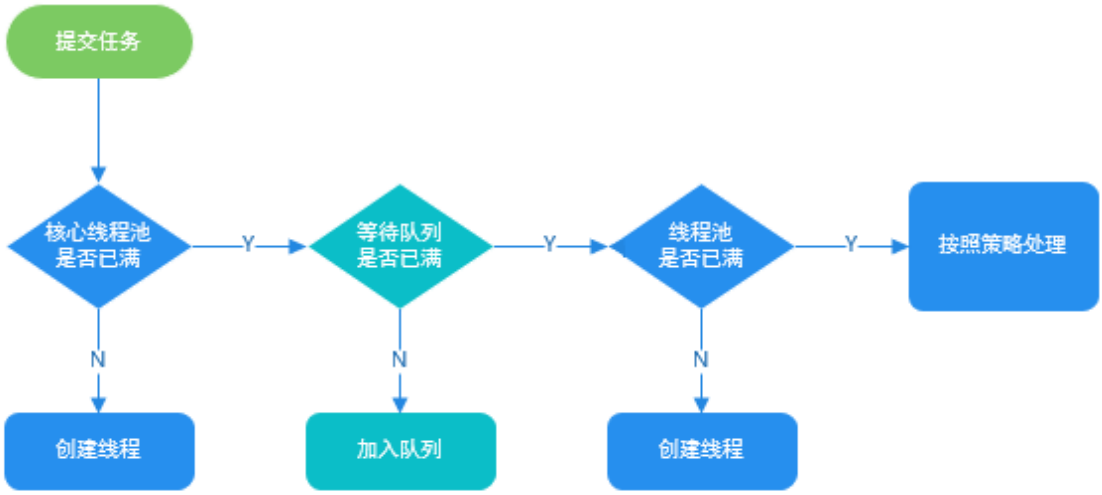
拒绝策略：

1. `ThreadPoolExecutor.AbortPolicy`：抛出 `RejectedExecutionException` 来拒绝新任务的处理。
2. `ThreadPoolExecutor.CallerRunsPolicy`：调用执行自己的线程运行任务，也就是直接在调用 `execute` 方法的线程中运行(`run`)被拒绝的任务，如果执行程序已关闭，则会丢弃该任务。因此这种策略会降低对于新任务提交速度，影响程序的整体性能。如果您的应用程序可以承受此延迟并且你要求任何一个任务请求都要被执行的话，你可以选择这个策略。
3. `ThreadPoolExecutor.DiscardPolicy`： 不处理新任务，直接丢弃掉。
4. `ThreadPoolExecutor.DiscardOldestPolicy`： 此策略将丢弃最早的未处理的任务请求。

创建线程池：

1. 通过ThreadPoolExecutor构造函数实现（推荐）
2. 通过Executor框架的工具类Executors来实现，可以创建三种类型的ThreadPoolExecutor：
   - FixedThreadPool 单线程的线程池
   - SingleThreadExecutor 固定线程数量的线程池，适合负载较重的服务器，不会有额外线程创建和回收的开销。
   - CachedThreadPool 可变尺寸的线程池，适合短期异步任务和负载较轻的服务器。

线程池原理分析



几个常见对比：

1. `Runnable` vs `Callable`

   `Runnable` **接口**不会返回结果或抛出检查异常，但是 `Callable` **接口**可以。所以，如果任务不需要返回结果或抛出异常推荐使用 `Runnable` **接口**，这样代码看起来会更加简洁。

2. `execute()` vs `submit()`

   - `execute()` **方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；**
   - `submit()` **方法用于提交需要返回值的任务。线程池会返回一个** `Future` **类型的对象，通过这个** `Future` **对象可以判断任务是否执行成功**

3. `shutdown()` VS `shutdownNow()`

   - `shutdown()` :关闭线程池，线程池的状态变为 `SHUTDOWN`。线程池不再接受新任务了，但是队列里的任务得执行完毕。
   - `shutdownNow()` :关闭线程池，线程的状态变为 `STOP`。线程池会终止当前正在运行的任务，并停止处理排队的任务并返回正在等待执行的 List。

4. `isTerminated()` VS `isShutdown()`

   - `isShutDown` 当调用 `shutdown()` 方法后返回为 true。
   - `isTerminated` 当调用 `shutdown()` 方法后，并且所有提交的任务完成后返回为 true

ThreadLocal

- **引用类型**
  - 强引用（strong reference）

    只要对象有强引用指向，并且GC Roots可达，那么Java内存回收时，即使濒临内存耗尽，也不会回收对象。

  - 软引用（soft reference）

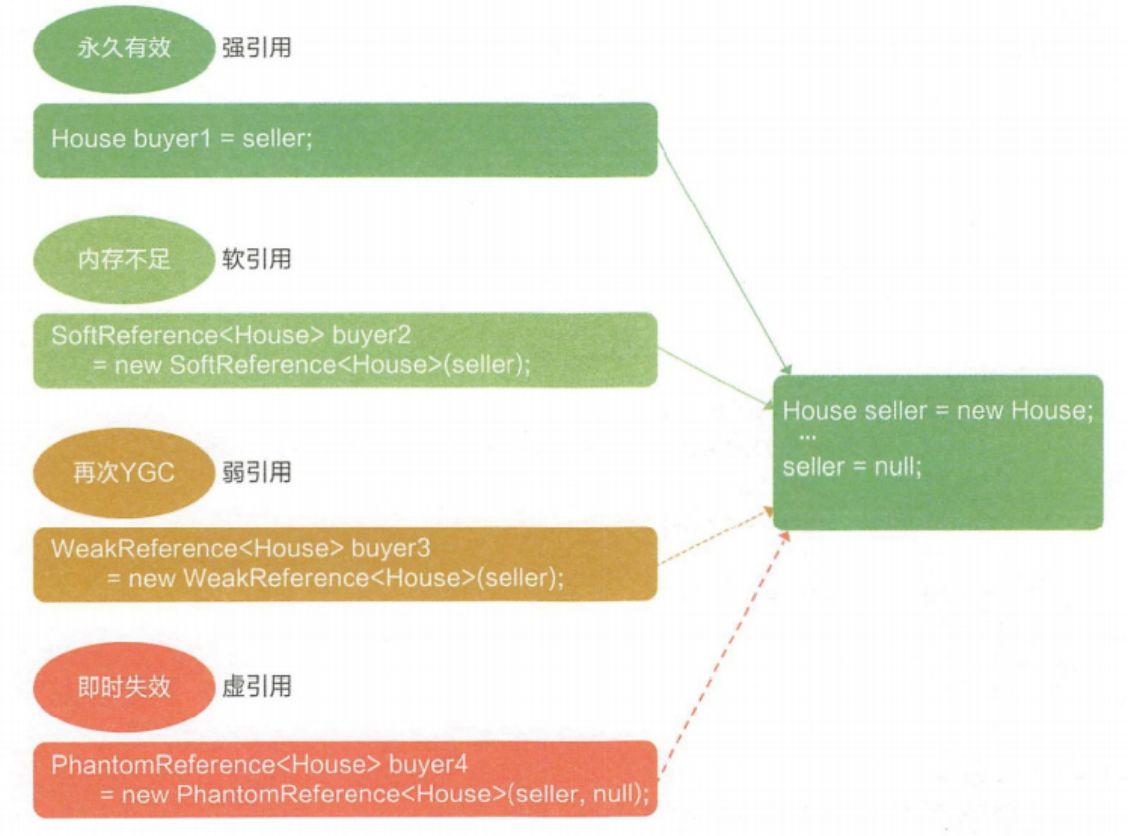
    在即将OOM之前，垃圾回收器会把软引用指向的对象加入回收范围。主要用来缓存服务器中间计算结果及不需要实时保存的用户行为等。

  - 弱引用（weak reference）

    如果弱引用指向的对象只存在弱引用这一条线路，则在下一次YGC时会被回收。

- 虚引用（phantom reference）

  定义完成后，就无法通过该引用获取指向的对象。为一个对象设置虚引用的唯一目的就是希望能在这个对象被回收时收到一个系统通知。虚引用必须与引用队列联合使用，当垃圾回收时，如果发现存在虚引用，就会在回收对象内存前，把这个虚引用加入与之相关的引用队列中。



ThreadLocal是弱引用

1. 每个thread维护着一个ThreadLocalMap的引用
2. ThreadLocalMap是ThreadLocal的内部类，用Entry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前必须先set。也可以初始化一个，但必须重写initalValue()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

ThreadLocal无法解决共享对象的更新问题，所以使用某个引用来操作共享对象时，依然需要进行线程同步

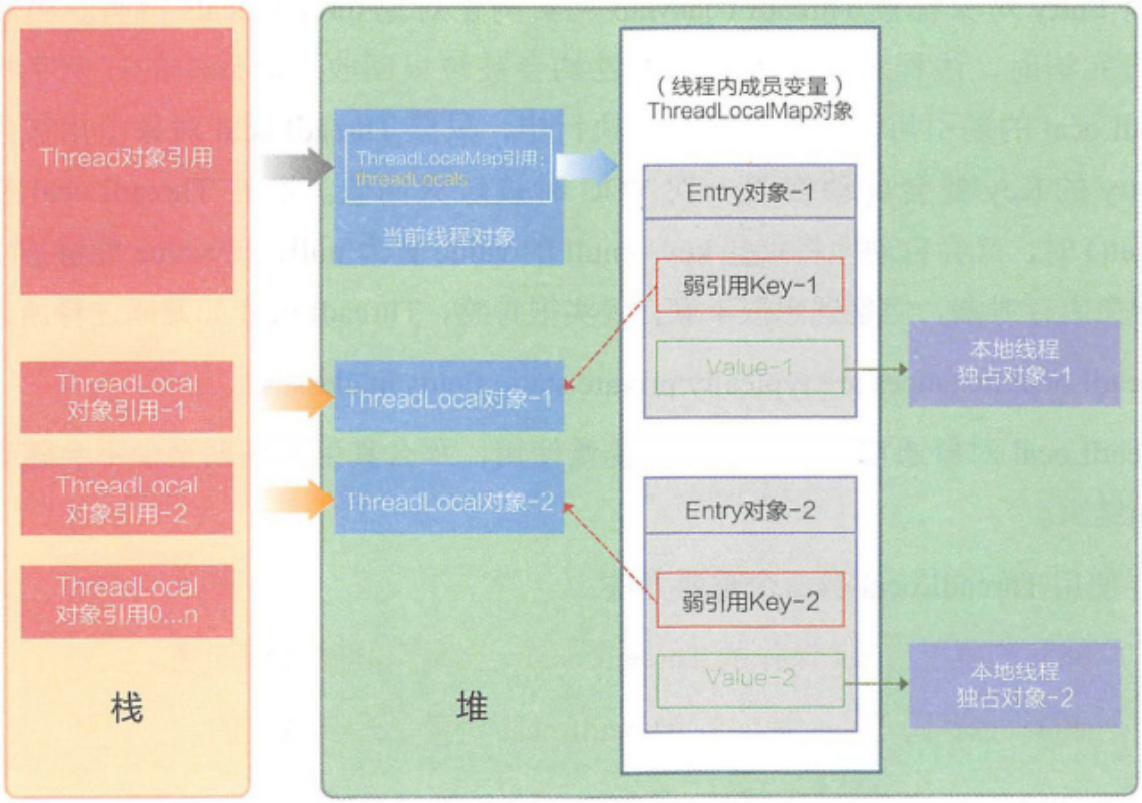
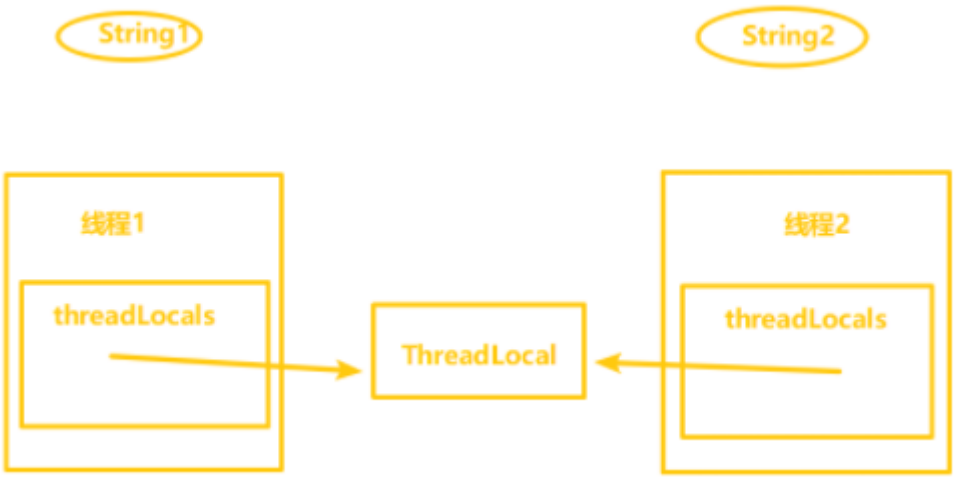


图 7-10 TheadLocal 的弱引用路线图



使用场景：

局部变量在方法内各代码块间进行传递，而类内变量在类内方法间进行传递。复杂的线程方法可能需要调用很多方法来实现某个功能，这时候用什么来传递线程内变量呢？答案就是 ThreadLocal。 它通常用于同一个线程内，跨类、跨方法传递数据。如果没有 ThreadLocal，那么相互之间的信息传递，势必要靠返回值 参数，这样无形之中，有些类甚至有些框架会互相耦合。

副作用：

1. 脏数据

   线程复用会产生脏数据。由于线程池会重用Thread对象，那么与Thread绑定的类的静态属性ThreadLocal变量也会被重用。如果在实现的线程run()方法体中不显式地调用remove()清理与线程有关地ThreadLocal信息，那么倘若下一个线程不调用set()设置初始值，就可能get()到重用的线程信息，包括ThreadLocal所关联地线程对象的value值。

2. 内存泄漏: **突然我们ThreadLocal是null了，也就是要被垃圾回收器回收了，但是此时我们的ThreadLocalMap生命周期和Thread的一样，它不会回收，这时候就出现了一个现象。那就是ThreadLocalMap的key没了，但是value还在，这就造成了内存泄漏。**

   以上两问题解决办法：使用完ThreadLocal后，执行remove操作，避免出现内存溢出情况。