ThreadLocal 适合用在哪些实际生产的场景中?

ThreadLocal 有两种典型的使用场景:

1. 保存每个线程**独享**的对象，为每个线程都创建一个副本。（线程独享对象）
2. 每个线程内需要独立保存信息，以便供其他方法更方便地获取该信息。（线程共享对象）

场景1示例:

如果没用ThreadLocal :

```
1  public class ThreadLocalDemo03 {
2
3      public static ExecutorService threadPool = Executors.newFixedThreadPool(16);
4
5      public static void main(String[] args) throws InterruptedException {
6          for (int i = 0; i < 1000; i++) {
7              int finalI = i;
8              threadPool.submit(new Runnable() {
9                  @Override
10                 public void run() {
11                     String date = new ThreadLocalDemo03().date(finalI);
12                     System.out.println(date);
13                 }
14             });
```

```
        }
        threadPool.shutdown();
    }
    public String date(int seconds) {
        Date date = new Date(1000 * seconds);
        SimpleDateFormat dateFormat = new SimpleDateFormat("mm:ss");
        return dateFormat.format(date);
    }
}
```

```
00:00
00:07
00:04
00:02
...
16:29
16:28
16:27
16:26
16:39
```

如上代码，使用线程池运行1000个线程打印时间，虽然结果是对的，但是创建了1000次对象，这么多对象的创建是有开销的，并且在使用完之后的销毁同样是有开销的，而且这么多对象同时存在在内存中也是一种内存的浪费。

解决1：将simpleDateFormat对象变为static 全局变量，这样还有线程不安全问题（打印结果会重复，因为simpleDateFormat本身线程就是不安全的）。

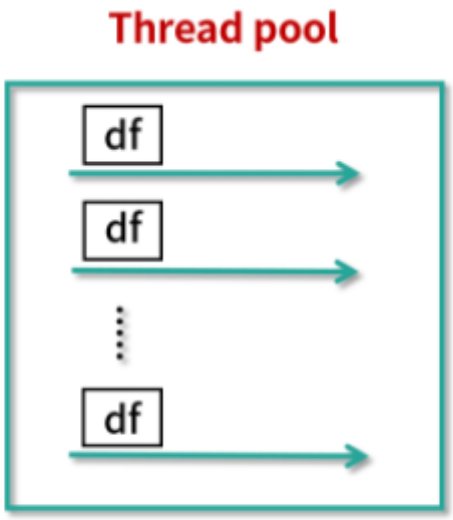
解决2：加锁，多个线程不能同时工作效率很低。

使用ThreadLocal：

```
public class ThreadLocalDemo06 {

    public static ExecutorService threadPool = Executors.newFixedThreadPool(16);

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 1000; i++) {
            int finalI = i;
            threadPool.submit(new Runnable() {
                @Override
                public void run() {
                    String date = new ThreadLocalDemo06().date(finalI);
                    System.out.println(date);
                }
            });
        }
        threadPool.shutdown();
    }

    public String date(int seconds) {
        Date date = new Date(1000 * seconds);
        SimpleDateFormat dateFormat =
ThreadSafeFormatter.dateFormatThreadLocal.get();
        return dateFormat.format(date);
    }
}

class ThreadSafeFormatter {
```

```
27      public static ThreadLocal<SimpleDateFormat> dateFormatThreadLocal = new
ThreadLocal<SimpleDateFormat>() {
28          @Override
29          protected SimpleDateFormat initialValue() {
30              return new SimpleDateFormat("mm:ss");
31          }
32      };
33  }
```

我们使用了 ThreadLocal 帮每个线程去生成它自己的 simpleDateFormat 对象，这个对象就不会创造过多，一共只有 16 个，因为线程只有 16 个，结果也是正确的。



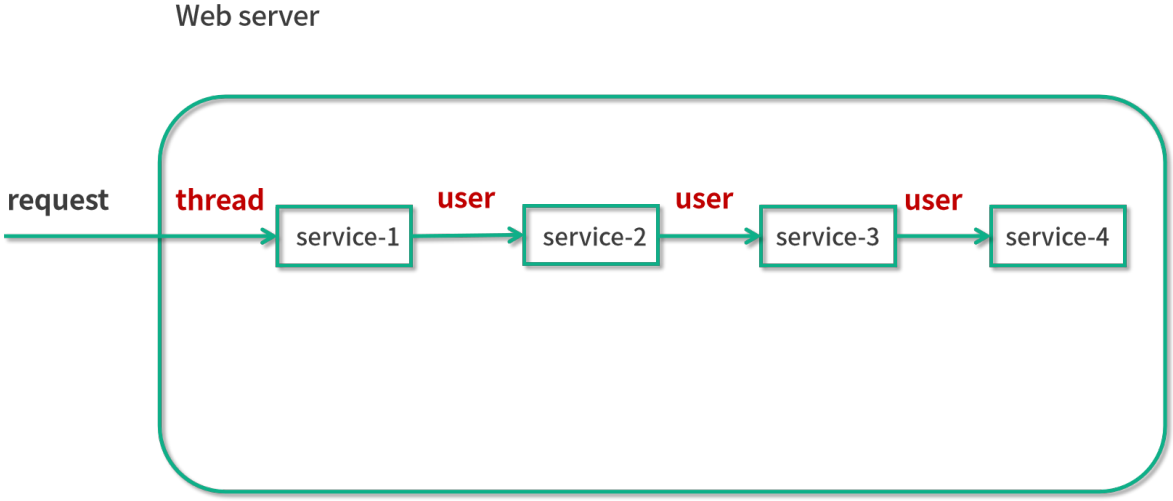
16个线程对应16个 simpleDateFormat对象

每个线程有自己的副本，是线程安全的

场景2实例：

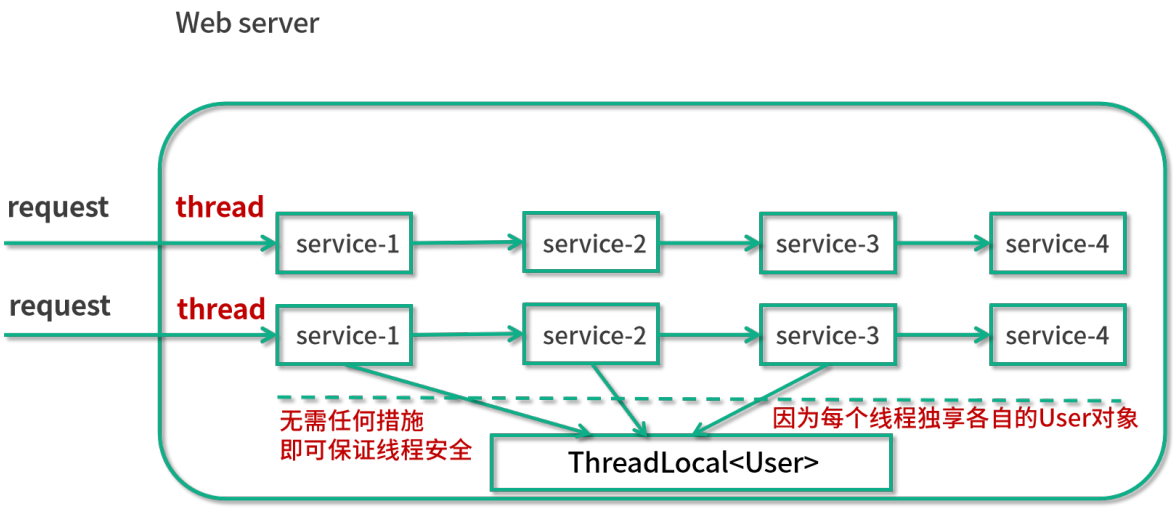
如果没用ThreadLocal：

不同线程都要使用User对象，但是不同的线程获取到的用户信息可能不一样。



如上图，我做一个操作，user在service-1、service-2、service-3、service-4都有用到，这时需把这个 user 对象层层传递下去，代码很冗余；于是可以使用 HashMap，但是线程不安全；于是改为 ConcurrentHashMap或synchronized，性能都是有损耗的。

使用ThreadLocal：



从上图看出，线程同时去访问这个 ThreadLocal 并且能利用 ThreadLocal 拿到只属于自己的独享对象。

如下代码：

```
public class ThreadLocalDemo07 {
    public static void main(String[] args) {
        new Service1().service1();
    }
}

class Service1 {
    public void service1() {
        User user = new User("拉勾教育");
        UserContextHolder.holder.set(user);
        new Service2().service2();
    }
}

class Service2 {
    public void service2() {
        User user = UserContextHolder.holder.get();
        System.out.println("Service2拿到用户名：" + user.name);
        new Service3().service3();
    }
}

class Service3 {
    public void service3() {
        User user = UserContextHolder.holder.get();
        System.out.println("Service3拿到用户名：" + user.name);
        UserContextHolder.holder.remove();
    }
}

class UserContextHolder {
    public static ThreadLocal<User> holder = new ThreadLocal<>();
}
```

```
34
35  class User {
36      String name;
37      public User(String name) {
38          this.name = n
39      }
40  }
```

User的传递Service1创建User，Service2拿到并传递User，Service3拿到Service2传递的User，中间可以操作User。

ThreadLocal 是用来解决共享资源的多线程访问的问题吗?

ThreadLocal 是不是用来解决共享资源的多线程访问的?

答：不是，因为这不是ThreadLocal 的应用场景，ThreadLocal 解决线程安全的方法是每个线程独享而不是共享资源。

举例：如果把一个全局变量

```
1   public class ThreadLocalStatic {
2
3       public static ExecutorService threadPool = Executors.newFixedThreadPool(16);
4       static SimpleDateFormat dateFormat = new SimpleDateFormat("mm:ss");
5
6
7       public static void main(String[] args) throws InterruptedException {
8           for (int i = 0; i < 1000; i++) {
9               int finalI = i;
10              threadPool.submit(new Runnable() {
11                  @Override
12                  public void run() {
13                      String date = new ThreadLocalStatic().date(finalI);
14                      System.out.println(date);
15                  }
16              });
17          }
18          threadPool.shutdown();
19      }
20
21      public String date(int seconds) {
22          Date date = new Date(1000 * seconds);
23          SimpleDateFormat dateFormat =
    ThreadSafeFormatter.dateFormatThreadLocal.get();
24          return dateFormat.format(date);
25      }
26  }
27
28  class ThreadSafeFormatter {
29
30      public static ThreadLocal<SimpleDateFormat> dateFormatThreadLocal = new
    ThreadLocal<SimpleDateFormat>() {
31          @Override
```

```
32          protected SimpleDateFormat initialValue() {
33              return ThreadLocalStatic.dateFormat;
34          }
35      }
36  }
```

如上代码为上一讲改造，将static修饰的全局变量dateFormat放入ThreadLocal，运行结果会出现线程安全问题（时间重复打印）。这时需使用synchronized 而不是ThreadLocal。

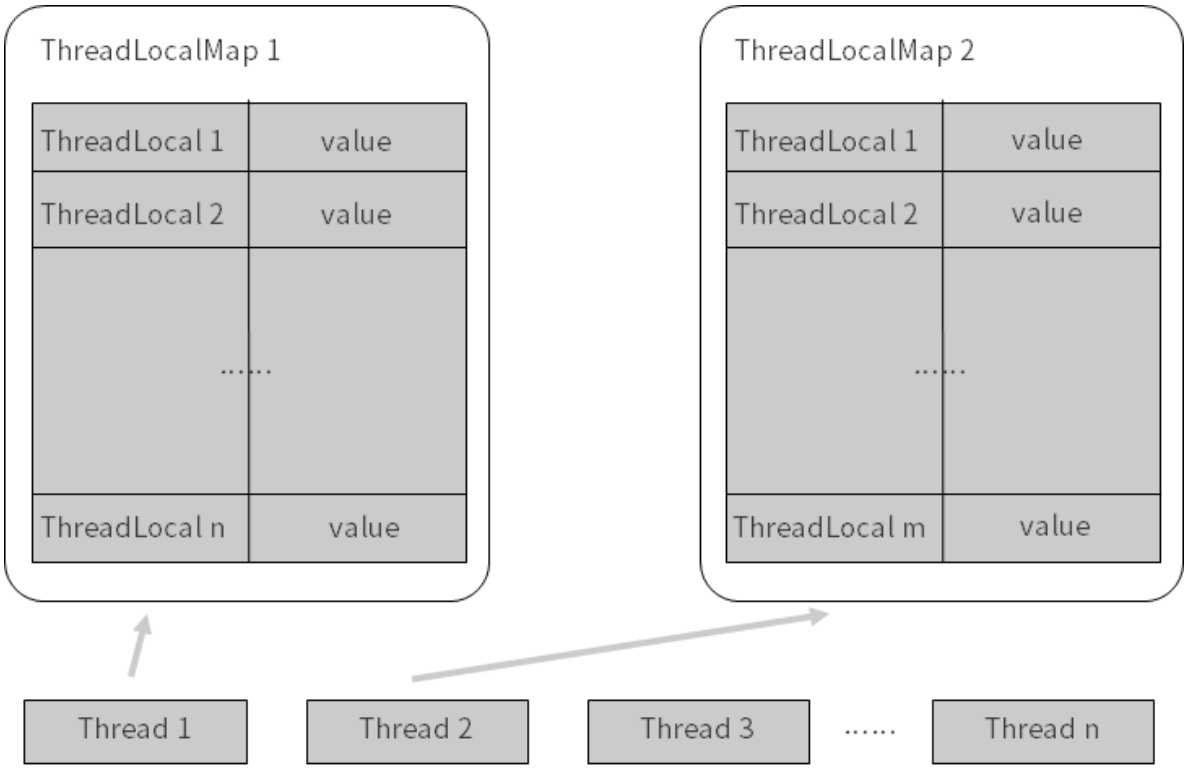
ThreadLocal 和 synchronized 是什么关系?

ThreadLocal 和 synchronized 它们两个都能解决线程安全问题，但是解决的思路不一致：

- ThreadLocal 是通过让每个线程独享自己的副本，避免了资源的竞争。
- synchronized 主要是同一时刻限制最多只有一个线程能访问该资源。

ThreadLocal 的效率比synchronized 要高，ThreadLocal 效率高的原因可以使用线程池，只要创建和线程池里的线程相同数量的对象就可以达到线程安全的目的，而不是每个线程都要创建一个对象。

Thread、 ThreadLocal 及 ThreadLocalMap 三者之间的关系?



如上图所示，每个Thread都持有一个ThreadLocalMap ， ThreadLocalMap 这样 Map 的数据结构来存放 ThreadLocal 和 value。

ThreadLocalMap 的get方法源码:

```
public T get() {
    //获取到当前线程
    Thread t = Thread.currentThread();
    //获取到当前线程内的 ThreadLocalMap 对象，每个线程内都有一个 ThreadLocalMap 对象
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        //获取 ThreadLocalMap 中的 Entry 对象并拿到 Value
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    //如果线程内之前没创建过 ThreadLocalMap，就创建
    return setInitialValue();
}
```

Thread.currentThread 来获取当前线程的引用，并且把这个引用传入到了 getMap 方法里面，来拿到当前线程的 ThreadLocalMap。

getMap（）讲解：

```
ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}
```

这个方法的作用就是获取到当前线程内的 ThreadLocalMap 对象，每个线程都有 ThreadLocalMap 对象，而这个对象的名字就叫作 threadLocals，初始值为 null，代码如下：

```
ThreadLocal.ThreadLocalMap threadLocals = null;
```

ThreadLocalMap 的set方法源码：

```
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}
```

set 方法的作用是把我们想要存储的 value 给保存进去。可以看出，首先，它还是需要获取到当前线程的引用，并且利用这个引用来获取到 ThreadLocalMap ； 然后，如果 map == null 则去创建这个 map，而当 map != null 的时候就利用 map.set 方法，把 value 给 set 进去。

可以看出，map.set(this, value) 传入的这两个参数中，第一个参数是 this，就是当前 ThreadLocal 的引用，这也再次体现了，在 ThreadLocalMap 中，它的 key 的类型是 ThreadLocal；而第二个参数就是我们所传入的 value，这样一来就可以把这个键值对保存到 ThreadLocalMap 中去了。

ThreadLocalMap 类，也就是 Thread.threadLocals

```
static class ThreadLocalMap {

    static class Entry extends WeakReference<ThreadLocal<?>> {
        /** The value associated with this ThreadLocal. */
        Object value;


        Entry(ThreadLocal<?> k, Object v) {
            super(k);
            value = v;
        }
    }
   private Entry[] table;
//...
}
```

ThreadLocalMap 类是每个线程 Thread 类里面的一个成员变量，其中最重要的就是截取出的这段代码中的 Entry 内部类。在 ThreadLocalMap 中会有一个 Entry 类型的数组，名字叫 table。我们可以把 Entry 理解为一个 map，其键值对为：

- 键，当前的 ThreadLocal；
- 值，实际需要存储的变量，比如 user 用户对象或者 simpleDateFormat 对象等。

ThreadLocalMap和HashMap 相同和区别：

内部结构一样，但是面对 hash 冲突时采取措施不一样，HashMap 采用的是拉链法，ThreadLocalMap发生冲突时会寻找下一个空的格子。

为何每次用完 ThreadLocal 都要调用 remove()?

什么是内存泄漏?

内存泄漏指的是，当某一个对象不再有用的时候，占用的内存却不能被回收，这就叫作**内存泄漏。**如果对象没有用，但一直不能被回收，这样的垃圾对象如果积累的越来越多，则会导致我们可用的内存越来越少，最后发生内存不够用的 OOM 错误。

ThreadLocal 是如何发生内存泄露?

Key 的泄漏:

如果我们在 ThreadLocalMap 的 Entry 中强引用了 ThreadLocal 实例，那么，虽然在业务代码中把 ThreadLocal 实例置为了 null，但是在 Thread 类中依然有这个引用链的存在，**GC 在垃圾回收的时候会进行可达性分析**，发现这个 ThreadLocal 对象依然是可达的，所以对于这个 ThreadLocal 对象不会进行垃圾回收，这样的话就造成了内存泄漏的情况。

JDK 开发者考虑到了这一点，所以 ThreadLocalMap 中的 Entry 继承了 WeakReference 弱引用，代码如下所示：

```
static class Entry extends WeakReference<ThreadLocal<?>> {
    /** The value associated with this ThreadLocal. */
    Object value;

    Entry(ThreadLocal<?> k, Object v) {
        super(k);
        value = v;
    }
}
```

如上代码，Entry继承的是WeakReference，使用的是弱引用，没有任何强引用关联，那么这个对象就可以被回收，所以弱引用不会阻止 GC。因此，这个弱引用的机制就避免了 ThreadLocal 的内存泄露问题。

Value 的泄漏：

上面的代码 value = v这行代码就代表了强引用的发生。

假如线程终止了，key 所对应的 value 是可以被正常垃圾回收的，因为没有任何强引用存在了。

但是如果线程迟迟不会终止或者生命周期很长，而该ThreadLocal 早就不再有用了，key可以回收，但是value发生了强引用，这就导致线程泄露问题。

如何避免Value出现的内存泄露问题：

JDK 同样也考虑到了这个问题，在执行 ThreadLocal 的 **set、remove、rehash** 等方法时，它都会扫描 key 为 null 的 Entry，如果发现某个 Entry 的 key 为 null，则代表它所对应的 value 也没有作用了，所以它就会把对应的 value 置为 null，这样，value 对象就可以被正常回收了。

但是假设 ThreadLocal 已经不被使用了，那么实际上 set、remove、rehash 方法也不会被调用，与此同时，如果这个线程又一直存活、不终止的话，那么刚才的那个调用链就一直存在，也就导致了 value 的内存泄漏。

所以我们每次使用ThreadLocal 都要调用remove 方法。调用这个方法就可以删除对应的 value 对象，可以避免内存泄漏。

我们来看一下 remove 方法的源码：

```
public void remove() {
    ThreadLocalMap m = getMap(Thread.currentThread());
    if (m != null)
        m.remove(this);
}
```

可以看出，它是先获取到 ThreadLocalMap 这个引用的，并且调用了它的 remove 方法。这里的 remove 方法可以把 key 所对应的 value 给清理掉，这样一来，value 就可以被 GC 回收了。

所以，在使用完了 ThreadLocal 之后，我们应该手动去调用它的 remove 方法，目的是防止内存泄漏的发生。