链滴

震惊！原来 threadLocal 还能这么用！

作者：gitzzzf

原文链接：https://ld246.com/article/1575173450174

来源网站：链滴

关于 threadLocal 如果你想了解更多，希望这边文章对你有所帮助。

我想对于 threadLocal 大家都不会陌生，我们经常用他存储一些全局类型的变量，然后方便在整个调链的各个地方使用，类似这样就可以。

```
private static ThreadLocal<Object> threadLocal = new ThreadLocal<>();

public static Object getThreadLocalValue() {
    return threadLocal.get();
}

public static void setThreadLocalValue(Object object) {
    threadLocal.set(object);
}
```

我相信看过 threadLocal 的同学对于类似这样的使用方式都不会陌生，很简单的初始化 set()进去之，在其他地方 get()方法获取就可以了。当然这篇文章的目的不是为了这个，首先第一个问题，new ThreadLocal<>()能不能在子线程中使用，如果我想在子线程中使用线程
量的副本怎么办?

原文链接：震惊！原来 threadLocal 还能这么用！

果然，这个问题还是难不到你，我太天真了。

用 ThreadLocal 的子类 InheritableThreadLocal 啊，InheritableThreadLocal 在 threadLocal 的基上，解决了和线程相关的副本从父线程向子线程传递的问题。他的实现原理是这样的 <erwog@!ff13dslso%$#@dfsdl;>。你娓娓道来，胸前的红领巾都不自觉地鲜艳了起来。

看着你的陈述，那么的自信，我邪魅一笑，很好，你成功吸引了我的注意！

看来，不拿点压箱底的东西是镇不住你的了，问题真的来了，如果我想从线程池中拿到主线程的全局量副本可以吗？

哼，不知道了吧，看我给你一一解释。什么？？！你知道？好吧，我们先假装你不会，不然这篇文章写不下去了， 逃 ：）求饶命 ~

先说结论，alibaba 提供了一种解决线程池中线程使用主线程中副本的办法 ----- **TransmittableThreadLocal**。

解释 TransmittableThreadLocal 实现原理之前我们先回顾下，InheritableThreadLocal 为什么能保子线程可以从父线程中拿到副本呢？

先看下我们在父线程中 new Thread()的时候都干了些啥：

```
public Thread(Runnable target) {
    init(null, target, "Thread-" + nextThreadNum(), 0);
}
```

我们再看下 init 这个初始化方法。

```
private void init(ThreadGroup g, Runnable target, String name,
                  long stackSize, AccessControlContext acc) {

    Thread parent = currentThread();

     <省略一些这篇文章不太关注的逻辑, 想要全面了解thread的初始化逻辑可翻看源码>

    if (parent.inheritableThreadLocals != null)
        this.inheritableThreadLocals =
            ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);

}
```

ok，豁然开朗，原来在我们平时 new Thread()的时候已经把当前主线程也就是 currentThread()里的 inheritableThreadLocal 副本给子线程拷贝了一份啊，自然地，子线程也就可以获得主线程变量副本了。

那么为什么线程池会是什么结果呢？我们看个栗子。

```
public class Test {
    private static ThreadLocal<Map<String,String>> holder = new InheritableThreadLocal<>();

    public void testTtl(){
        // 构建线程池
        Executor executor = Executors.newFixedThreadPool(1);

        executor.execute(()-> System.out.println("init"));

        // 初始化ThreadLocal
```

```
        HashMap<String,String> map = new HashMap<>();
        map.put("1","2");
        holder.set(map);

        // 判断线程池中能够拿到主线程threadLocal副本
        executor.execute(()->{
            System.out.println(holder.get()); //结果为：null
        });
    }

    public static void main(String[] args) {
        new Test().testTtl();
    }
}
```

到现在，我们知道了，因为线程池中线程的复用，所以这个 inheritableThreadLocal 只能维持在这线程创建时候的状态。

那么接下来，就是讲解为什么 alibaba 提供的 TransmittableThreadLocal 能够实现线程池中副本的递。

TransmittableThreadLocal 继承了 InheritableThreadLocal，重载了 get 和 set 方法。

```
@Override
public final T get() {
    T value = super.get();
    if (null != value) addValue();
    return value;
}

@Override
public final void set(T value) {
    super.set(value);
    // may set null to remove value
    if (null == value) removeValue();
    else addValue();
}
```

可以看到在调用父类的逻辑上，新增了 addValue 和 removeValue 的逻辑，这个就是缓存的逻辑，把当前这个 threadlocal 缓存到 holder 上面。

```
private void addValue() {
    if (!holder.get().containsKey(this)) {
        holder.get().put(this, null); // WeakHashMap supports null value.
    }
}

private void removeValue() {
    holder.get().remove(this);
}
```

下面介绍下这个很关键的 holder。

```
private static InheritableThreadLocal<Map<TransmittableThreadLocal<?>, ?>> holder =
new InheritableThreadLocal<Map<TransmittableThreadLocal<?>, ?>>() {
    @Override
```

```
    protected Map<TransmittableThreadLocal<?>, ?> initialValue() {
        return new WeakHashMap<TransmittableThreadLocal<?>, Object>();
    }

    @Override
    protected Map<TransmittableThreadLocal<?>, ?> childValue(Map<TransmittableThreadLo
al<?>, ?> parentValue) {
        return new WeakHashMap<TransmittableThreadLocal<?>, Object>(parentValue);
    }
};
```

首先这个 holder 本身是**InheritableThreadLocal**类型的，所以它也是和线程相关联的。可以在**父子程**间传递，但是对于**线程池**内已经创建的线程肯定是传递不进去的。所以在初始化 wrapper 类（包类）的时候，那个时候还是父线程，在 wrapper 类构造的时候，要把这些 threadlocal 捕获出来，个捕获相关逻辑见下一个 Transmitter 的分析。

Transmitter内有3个核心方法，ttl 表示TransmittableThreadLocal。

- capture：捕获父线程的ttl
- replay：重放父线程ttl
- restore：恢复之前子线程的ttl

capture用于捕获父线程的ttl，捕获操作要在**父线程**执行。

```
public static Object capture() {
    return new Snapshot(captureTtlValues(), captureThreadLocalValues());
}

private static WeakHashMap<TransmittableThreadLocal<Object>, Object> captureTtlValues()
{
    WeakHashMap<TransmittableThreadLocal<Object>, Object> ttl2Value = new WeakHash
ap<TransmittableThreadLocal<Object>, Object>();
    for (TransmittableThreadLocal<Object> threadLocal : holder.get().keySet()) {
        ttl2Value.put(threadLocal, threadLocal.copyValue());
    }
    return ttl2Value;
}

private static WeakHashMap<ThreadLocal<Object>, Object> captureThreadLocalValues() {
    final WeakHashMap<ThreadLocal<Object>, Object> threadLocal2Value = new WeakHash
ap<ThreadLocal<Object>, Object>();
    for (Map.Entry<ThreadLocal<Object>, TtlCopier<Object>> entry : threadLocalHolder.entry
et()) {
        final ThreadLocal<Object> threadLocal = entry.getKey();
        final TtlCopier<Object> copier = entry.getValue();

        threadLocal2Value.put(threadLocal, copier.copy(threadLocal.get()));
    }
    return threadLocal2Value;
}
```

replay**传入capture方法捕获的ttl**，然后在子线程重放，也就是调用ttl的set方法，会设置到当前的线中去，最后会把子线程之前存在的ttl返回。

```
public static Object replay(@NonNull Object captured) {
    final Snapshot capturedSnapshot = (Snapshot) captured;
    return new Snapshot(replayTtlValues(capturedSnapshot.ttl2Value), replayThreadLocalValue
(capturedSnapshot.threadLocal2Value));
}

@NonNull
private static WeakHashMap<TransmittableThreadLocal<Object>, Object> replayTtlValues(
NonNull WeakHashMap<TransmittableThreadLocal<Object>, Object> captured) {
    WeakHashMap<TransmittableThreadLocal<Object>, Object> backup = new WeakHashMa
<TransmittableThreadLocal<Object>, Object>();

    for (final Iterator<TransmittableThreadLocal<Object>> iterator = holder.get().keySet().itera
or(); iterator.hasNext(); ) {
        TransmittableThreadLocal<Object> threadLocal = iterator.next();

        // backup
        backup.put(threadLocal, threadLocal.get());

        // clear the TTL values that is not in captured
        // avoid the extra TTL values after replay when run task
        if (!captured.containsKey(threadLocal)) {
            iterator.remove();
            threadLocal.superRemove();
        }
    }

    // set TTL values to captured
    setTtlValuesTo(captured);

    // call beforeExecute callback
    doExecuteCallback(true);

    return backup;
}

private static WeakHashMap<ThreadLocal<Object>, Object> replayThreadLocalValues(@No
Null WeakHashMap<ThreadLocal<Object>, Object> captured) {
    final WeakHashMap<ThreadLocal<Object>, Object> backup = new WeakHashMap<Threa
Local<Object>, Object>();

    for (Map.Entry<ThreadLocal<Object>, Object> entry : captured.entrySet()) {
        final ThreadLocal<Object> threadLocal = entry.getKey();
        backup.put(threadLocal, threadLocal.get());

        final Object value = entry.getValue();
        if (value == threadLocalClearMark) threadLocal.remove();
        else threadLocal.set(value);
    }

    return backup;
}
```

最后就是执行结束，restore之前的上下文，用到replay返回的back。

```
public static void restore(@NonNull Object backup) {
    final Snapshot backupSnapshot = (Snapshot) backup;
    restoreTtlValues(backupSnapshot.ttl2Value);
    restoreThreadLocalValues(backupSnapshot.threadLocal2Value);
}
```

要把capture，repaly和restore的逻辑串起来，那么就需要看下面的TtlRunnable类，这个就是我一说的包装类。

```
 private TtlRunnable(@Nonnull Runnable runnable, boolean releaseTtlValueReferenceAfterRu
) {
        //捕获父线程ttl
        this.capturedRef = new AtomicReference<Object>(capture());
        this.runnable = runnable;
        this.releaseTtlValueReferenceAfterRun = releaseTtlValueReferenceAfterRun;
 }
```

在构造函数，也就是父线程，**会通过capture捕获父线程的ttl，然后保存在capturedRef中。**

在run方法中，replay，restore逻辑一目了然。

```
@Override
public void run() {
    Object captured = capturedRef.get();
    if (captured == null || releaseTtlValueReferenceAfterRun && !capturedRef.compareAndSet(
aptured, null)) {
        throw new IllegalStateException("TTL value reference is released after run!");
    }

    Object backup = replay(captured);
    try {
        runnable.run();
    } finally {
        restore(backup);
    }
}
```

至此，关于如何在线程池中实现TransmittableThreadLocal副本的传递方案阐述完毕。

那么我们有什么方便简单的方式使用这个TransmittableThreadLocal实现线程池中的线程变量副本传吗?

当然有，alibaba在提供了TransmittableThreadLocal的同时，提供了一个非常好用的使用方式 -- **Ttl xecutors**.

TtlExecutors使用起来非常的方便，在我们构建的线程池上面做个封装就好了。

```
    Executor executor = Executors.newFixedThreadPool(10);
Executor ttlExecutor = TtlExecutors.getTtlExecutor(executor);
```

我们看一眼TtlExecutors.getTtlExecutor()做了什么。

```
public static Executor getTtlExecutor(@Nullable Executor executor) {
    if (TtlAgent.isTtlAgentLoaded() || null == executor || executor instanceof TtlEnhanced) {
        return executor;
    }
    return new ExecutorTtlWrapper(executor);
}
```

其实就构建了一个包装类ExecutorTtlWrapper，我们再跟进去看一眼，发现ExecutorTtlWrapper很单，他实现Executor，覆盖了execute方法。

```
@Override
public void execute(@NonNull Runnable command) {
    executor.execute(TtlRunnable.get(command));
}
```

看到这我们发现原来这个execute方法根据我们传进来的Runnable构建了TtlRunnable，TtlRunnabl我们在上面已经介绍，就是在这个类的run()方法里面我们实现了线程池变量副本的传递。

当然除了TtlExecutors之外，还有通过TtlAgent使用TransmittableThreadLocal的办法，这里不做过介绍了。

详细信息可以参考官方：https://github.com/alibaba/transmittable-thread-local