Java 并发编程之美：并发编程高级篇之一

借用 Java 并发编程实践中的话：编写正确的程序并不容易，而编写正常的并发程序就更难了。相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

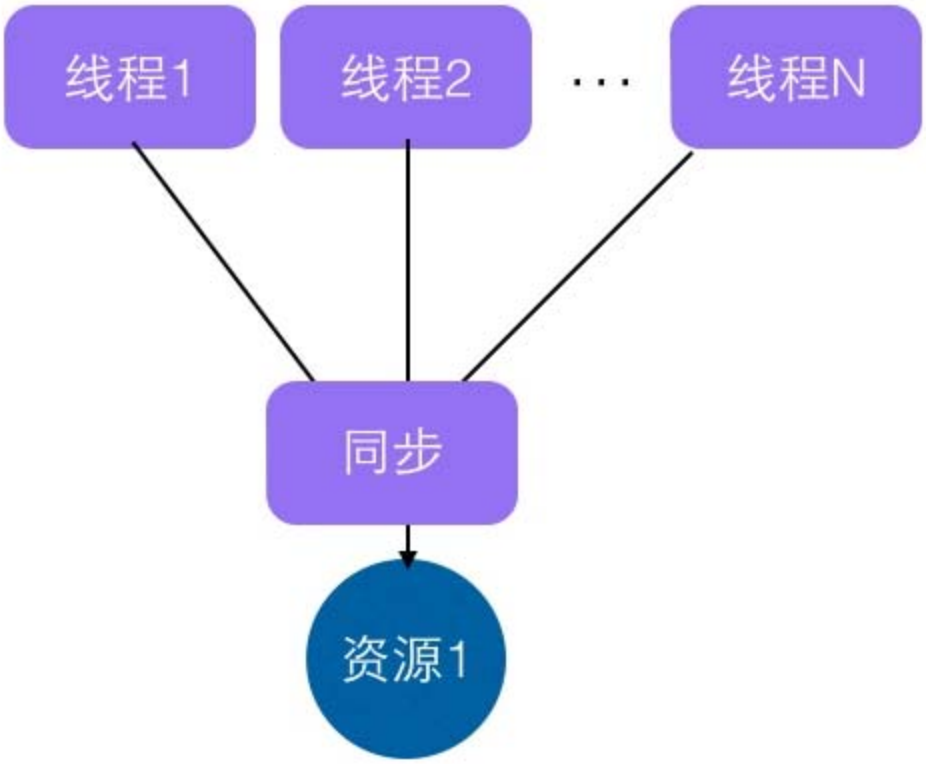
并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却都还离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

本 Chat 作为 Java 并发编程之美系列的高级篇之一，主要讲解内容如下：（建议先阅读《Java 并发编程之美：基础篇》）

- ThreadLocal 的实现原理，ThreadLocal 作为变量的线程隔离方式，其内部是如何做的?
- InheritableThreadLocal 的实现原理，InheritableThreadLocal 是如何弥补 ThreadLocal 不支持继承的特性?
- JDK 并发包中 ThreadLocalRandom 类原理剖析，经常使用的随机数生成器 Random 类的原理是什么？及其局限性是什么？ThreadLocalRandom 是如何利用 ThreadLocal 的原理来解决 Random 的局限性?
- 最后 ThreadLocal 的一个使用场景，Spring 框架中 Scope 作用域 Bean 的实现原理。

ThreadLocal 的实现原理

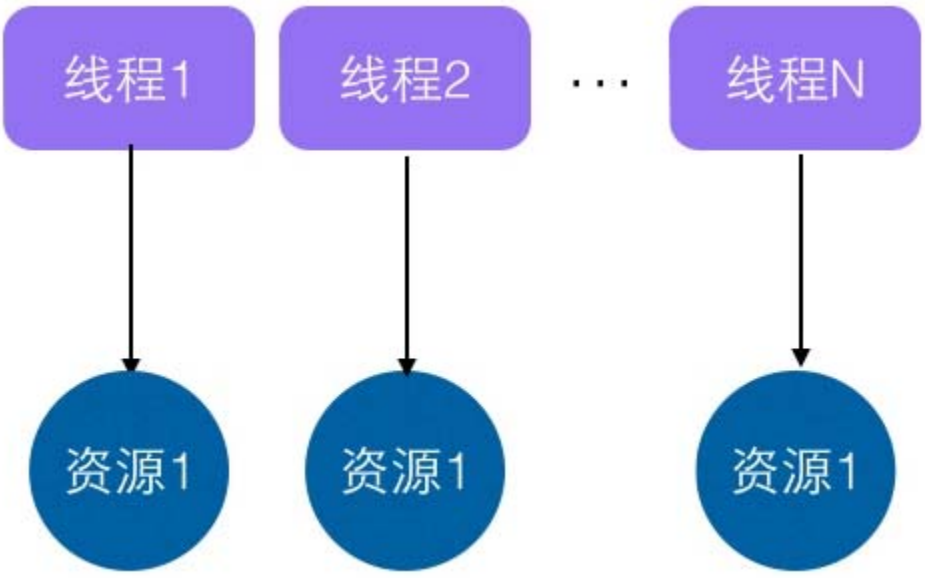
多线程访问同一个共享变量特别容易出现并发问题，特别是多个线程需要对一个共享变量进行写入时候，为了保证线程安全，一般需要使用者在访问共享变量的时候进行适当的同步，如下图：



同步的措施一般是加锁，这就需要使用者对锁也要有一定了解，这显然加重了使用者的负担。

那么有没有一个方式当创建一个变量时候，每个线程对其进行访问的时候访问的是自己线程的变量呢？其实 ThreadLocal 就可以做这个事情，虽然 ThreadLocal 的出现并不是为了解决上面的问题而出现的。

ThreadLocal 是在 JDK 包里面提供的，它提供了线程本地变量，也就是如果你创建了一个 ThreadLocal 变量，那么访问这个变量的每个线程都会有这个变量的一个本地拷贝，多个线程操作这个变量的时候，实际是操作的自己本地内存里面的变量，从而避免了线程安全问题，创建一个 ThreadLocal 变量后每个线程会拷贝一个变量到自己本地内存，如下图：



ThreadLocal 简单使用

本节来看下 ThreadLocal 如何使用，从而加深理解，本例子开启了两个线程，每个线程内部设置了本地变量的值，然后调用 print 函数打印当前本地变量的值，如果打印后调用了本地变量的 remove 方法则会删除本地内存中的该变量，代码如下：

```
public class ThreadLocalTest {

    //(1)打印函数
    static void print(String str){
        //1.1  打印当前线程本地内存中localVariable变量的值
        System.out.println(str + ":" +localVariable.get());
        //1.2 清除当前线程本地内存中localVariable变量
        //localVariable.remove();
    }
    //(2) 创建ThreadLocal变量
    static ThreadLocal<String> localVariable = new ThreadLocal<>
();
    public static void main(String[] args) {

        //(3) 创建线程one
        Thread threadOne = new Thread(new  Runnable() {
            public void run() {
                //3.1 设置线程one中本地变量localVariable的值
                localVariable.set("threadOne local variable");
                //3.2 调用打印函数
                print("threadOne");
                //3.3打印本地变量值
                System.out.println("threadOne remove after" + ":"
+localVariable.get());

            }
        });
        //(4) 创建线程two
        Thread threadTwo = new Thread(new  Runnable() {
            public void run() {
                //4.1 设置线程one中本地变量localVariable的值
                localVariable.set("threadTwo local variable");
                //4.2 调用打印函数
                print("threadTwo");
                //4.3打印本地变量值
                System.out.println("threadTwo remove after" + ":"
+localVariable.get());

            }
        });
        //(5)启动线程
        threadOne.start();
        threadTwo.start();
    }
```

运行结果：

```
threadOne:threadOne local variable
threadTwo:threadTwo local variable
threadOne remove after:threadOne local variable
threadTwo remove after:threadTwo local variable
```

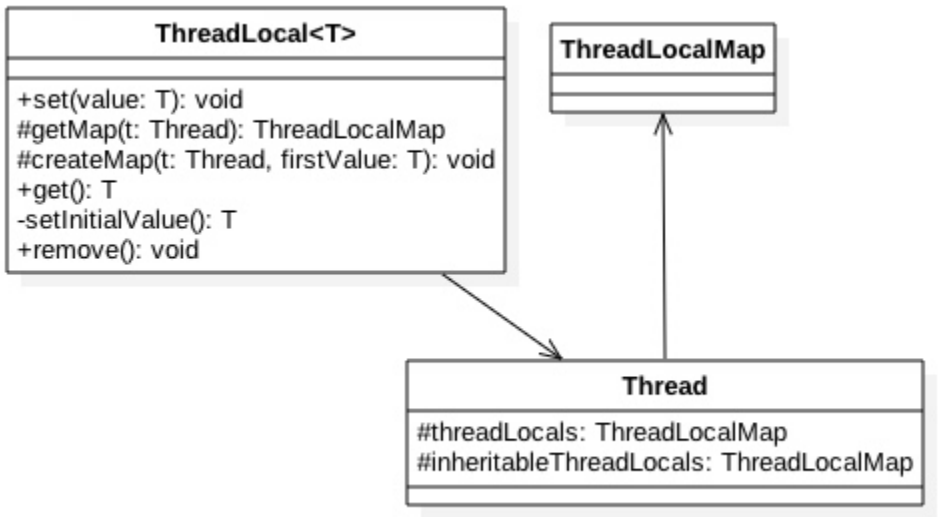
- 代码（2）创建了一个 ThreadLocal 变量；
- 代码（3）、（4）分别创建了线程 one 和 two；
- 代码（5）启动了两个线程；
- 线程 one 中代码 3.1 通过 set 方法设置了 localVariable 的值，这个设置的其实是线程 one 本地内存中的一个拷贝，这个拷贝线程 two 是访问不了的。然后代码 3.2 调用了 print 函数，代码 1.1 通过 get 函数获取了当前线程（线程 one）本地内存中 localVariable 的值；
- 线程 two 执行类似线程 one。

解开代码 1.2 的注释后，再次运行，运行结果为：

```
threadOne:threadOne local variable
threadOne remove after:null
threadTwo:threadTwo local variable
threadTwo remove after:null
```

ThreadLocal 实现原理

首先看下 ThreadLocal 相关的类的类图结构。



如上类图可知 Thread 类中有一个 threadLocals 和 inheritableThreadLocals 都是 ThreadLocalMap 类型的变量，而 ThreadLocalMap 是一个定制化的 Hashmap，默认每个线程中这个两个变量都为 null，只有当前线程第一次调用了 ThreadLocal 的 set 或者 get 方法时候才会进行创建。

其实每个线程的本地变量不是存放到 ThreadLocal 实例里面的，而是存放到调用线程的 threadLocals 变量里面。也就是说 ThreadLocal 类型的本地变量是存放到具体的线程内存空间的。

ThreadLocal 就是一个工具壳，它通过 set 方法把 value 值放入调用线程的 threadLocals 里面存放起来，当调用线程调用它的 get 方法时候再从当前线程的 threadLocals变 量里面拿出来使用。

如果调用线程一直不终止，那么这个本地变量会一直存放到调用线程的 threadLocals 变量里面，所以当不需要使用本地变量时候可以通过调用 ThreadLocal 变量的 remove 方法，从当前线程的 threadLocals 里面删除该本地变量。另外 Thread 里面的 threadLocals 为何设计为 map 结构呢？很明显是因为每个线程里面可以关联多个 ThreadLocal 变量。

下面简单分析下 ThreadLocal 的 set，get，remove 方法的实现逻辑：

- void set(T value)

```
public void set(T value) {
    //(1)获取当前线程
    Thread t = Thread.currentThread();
    //(2)当前线程作为key，去查找对应的线程变量，找到则设置
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
    //(3)第一次调用则创建当前线程对应的HashMap
        createMap(t, value);
}
```

如上代码（1）首先获取调用线程，然后使用当前线程作为参数调用了 getMap(t) 方法，getMap(Thread t) 代码如下：

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

可知 getMap(t) 所做的就是获取线程自己的变量 threadLocals，threadlocal 变量是绑定到了线程的成员变量里面。

如果 getMap(t) 返回不为空，则把 value 值设置进入到 threadLocals，也就是把当前变量值放入了当前线程的内存变量 threadLocals，threadLocals 是个 HashMap 结构，其中 key 就是当前 ThreadLocal 的实例对象引用，value 是通过 set 方法传递的值。

如果 getMap(t) 返回空那说明是第一次调用 set 方法，则创建当前线程的 threadLocals 变量，下面看 createMap(t, value) 里面做了啥呢？

```
 void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

可知就是创建当前线程的 threadLocals 变量。

- T get()

```
public T get() {
    //(4) 获取当前线程
    Thread t = Thread.currentThread();
    //(5)获取当前线程的threadLocals变量
    ThreadLocalMap map = getMap(t);
    //(6)如果threadLocals不为null，则返回对应本地变量值
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    //(7)threadLocals为空则初始化当前线程的threadLocals成员变量
            return setInitialValue();
}
```

如上代码（4）首先获取当前线程实例，如果当前线程的 threadLocals 变量不为 null 则直接返回当前线程绑定的本地变量。否者执行代码（7）进行初始化，setInitialValue() 的代码如下：

```
private T setInitialValue() {
    //(8)初始化为null
    T value = initialValue();
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    //(9)如果当前线程的threadLocals变量不为空
    if (map != null)
        map.set(this, value);
    else
    //(10)如果当前线程的threadLocals变量为空
        createMap(t, value);
    return value;
}


protected T initialValue() {
    return null;
```

```
    }
```

如上代码如果当前线程的 threadLocals 变量不为空，则设置当前线程的本地变量值为 null，否者调用 createMap 创建当前线程的 createMap 变量。

- void remove()

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

如上代码，如果当前线程的 threadLocals 变量不为空，则删除当前线程中指定 ThreadLocal 实例的本地变量。

注：每个线程内部都有一个名字为 threadLocals 的成员变量，该变量类型为 HashMap，其中 key 为我们定义的 ThreadLocal 变量的 this 引用，value 则为我们 set 时候的值，每个线程的本地变量是存到线程自己的内存变量 threadLocals 里面的，如果当前线程一直不消失那么这些本地变量会一直存到，所以可能会造成内存泄露，所以使用完毕后要记得调用 ThreadLocal 的 remove 方法删除对应线程的 threadLocals 中的本地变量。

子线程中获取不到父线程中设置的 ThreadLocal 变量的值

首先看个例子说明标题的意思：

```
public class TestThreadLocal {

    //(1) 创建线程变量
    public static ThreadLocal<String> threadLocal = new
ThreadLocal<String>();
    public static void main(String[] args) {

        //(2)  设置线程变量
        threadLocal.set("hello world");
        //(3) 启动子线程
        Thread thread = new Thread(new  Runnable() {
            public void run() {
                //(4)子线程输出线程变量的值
                System.out.println("thread:" +
threadLocal.get());

            }
        });
        thread.start();

        //(5)主线程输出线程变量值
```

```
            System.out.println("main:" + threadLocal.get());

        }
    }
```

结果为：

```
main:hello world
thread:null
```

也就是说同一个 ThreadLocal 变量在父线程中设置值后，在子线程中是获取不到的。根据上节的介绍，这个应该是正常现象，因为子线程调用 get 方法时候当前线程为子线程，而调用 set 方法设置线程变量是 main 线程，两者是不同的线程，自然子线程访问时候返回 null，那么有办法让子线程访问到父线程中的值吗？答案是有。

InheritableThreadLocal 原理

为了解决上节的问题 InheritableThreadLocal 应运而生，InheritableThreadLocal 继承自 ThreadLocal，提供了一个特性，就是子线程可以访问到父线程中设置的本地变量。

下面看下 InheritableThreadLocal 的代码：

```
public class InheritableThreadLocal<T> extends ThreadLocal<T> {

    //(1)
    protected T childValue(T parentValue) {
        return parentValue;
    }
    //(2)
    ThreadLocalMap getMap(Thread t) {
       return t.inheritableThreadLocals;
    }
    //(3)
    void createMap(Thread t, T firstValue) {
        t.inheritableThreadLocals = new ThreadLocalMap(this,
firstValue);
    }
}
```

如上代码可知 InheritableThreadLocal 继承了 ThreadLocal，并重写了三个方法。

- 代码（3）可知 InheritableThreadLocal 重写了 createMap 方法，那么可知现在当第一次调用 set 方法时候创建的是当前线程的 inheritableThreadLocals 变量的实例而不再是 threadLocals。

- 代码（2）可知当调用 get 方法获取当前线程的内部 map 变量时候，获取的是 inheritableThreadLocals 而不再是 threadLocals。

综上可知在 InheritableThreadLocal 的世界里，线程中的变量 inheritableThreadLocals 替代了 threadLocals。

- 下面我们看下重写的代码（1）是何时被执行，以及如何实现的子线程可以访问父线程本地变量的。这个要从 Thread 创建的代码看起，Thread 的默认构造函数及 Thread.java 类的构造函数如下：

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


      private void init(ThreadGroup g, Runnable target, String
name,
                        long stackSize, AccessControlContext acc) {
            ...
            //(4)获取当前线程
            Thread parent = currentThread();
            ...
            //(5)如果父线程的inheritableThreadLocals变量不为null
            if (parent.inheritableThreadLocals != null)
            //(6)设置子线程中的inheritableThreadLocals变量
            this.inheritableThreadLocals =

ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
        this.stackSize = stackSize;
        tid = nextThreadID();
    }
```

创建线程时候在构造函数里面会调用 init 方法，前面讲到了 inheritableThreadLocal 类 get，set 方法操作的是变量 inheritableThreadLocals，所以这里 inheritableThreadLocal 变量就不为 null，所以会执行代码（6），下面看下 createInheritedMap 代码：

```
    static ThreadLocalMap createInheritedMap(ThreadLocalMap
parentMap) {
        return new ThreadLocalMap(parentMap);
    }
```

可知 createInheritedMap 内部使用父线程的 inheritableThreadLocals 变量作为构造函数创建了一个新的 ThreadLocalMap 变量。然后赋值给了子线程的 inheritableThreadLocals 变量，那么下面看看 ThreadLocalMap 的构造函数里面做了什么：

```
    private ThreadLocalMap(ThreadLocalMap parentMap) {
                Entry[] parentTable = parentMap.table;
                int len = parentTable.length;
                setThreshold(len);
                table = new Entry[len];

                for (int j = 0; j < len; j++) {
                    Entry e = parentTable[j];
                    if (e != null) {
                        @SuppressWarnings("unchecked")
                        ThreadLocal<Object> key =
    (ThreadLocal<Object>) e.get();
                        if (key != null) {
                            //(7)调用重写的方法
                            Object value = key.childValue(e.value);//
    返回e.value
                            Entry c = new Entry(key, value);
                            int h = key.threadLocalHashCode & (len -
    1);
                            while (table[h] != null)
                                h = nextIndex(h, len);
                            table[h] = c;
                            size++;
                        }
                    }
                }
            }
```

如上代码所做的事情就是把父线程的 inheritableThreadLocals 成员变量的值复制到新的 ThreadLocalMap 对象，其中代码（7）InheritableThreadLocal 类重写的代码（1）也映入眼帘了。

总结：InheritableThreadLocal 类通过重写代码（2）和（3）让本地变量保存到了具体线程的 inheritableThreadLocals 变量里面，线程通过 InheritableThreadLocal 类实例的 set 或者 get 方法设置变量时候就会创建当前线程的 inheritableThreadLocals 变量。当父线程创建子线程时候，构造函数里面会把父线程中 inheritableThreadLocals 变量里面的本地变量拷贝一份复制到子线程的 inheritableThreadLocals 变量里面。

把上节代码（1）修改为：

```
    //(1) 创建线程变量
    public static ThreadLocal<String> threadLocal = new
InheritableThreadLocal<String>();
```

运行结果为：

```
thread:hello world
main:hello world
```

可知现在可以从子线程中正常的获取到线程变量值了。

那么什么情况下需要子线程可以获取到父线程的 threadlocal 变量呢，情况还是蛮多的，比如存放用户登录信息的 threadlocal 变量，很有可能子线程中也需要使用用户登录信息，再比如一些中间件需要用统一的追踪 ID 把整个调用链路记录下来的情景。

JDK 并发包中 ThreadLocalRandom 类原理剖析

ThreadLocalRandom 类是 JDK7 在 JUC 包下新增的随机数生成器，它解决了 Random 类在多线程下的不足。本节就来讲解下 JUC 下为何新增该类，以及该类的实现原理。

Random 类及其局限性

在 JDK7 之前包括现在，java.util.Random 应该是使用比较广泛的随机数生成工具类，另外 java.lang.Math 中的随机数生成也是使用的 java.util.Random 的实例。下面先看看 java.util.Random 的使用：

```
public class RandomTest {
    public static void main(String[] args) {

        //(1)创建一个默认种子的随机数生成器
        Random random = new Random();
        //(2)输出10个在0-5（包含0，不包含5）之间的随机数
        for (int i = 0; i < 10; ++i) {
            System.out.println(random.nextInt(5));
        }
    }
}
```

- 代码（1）创建一个默认随机数生成器，使用默认的种子。
- 代码（2）输出输出10个在0-5（包含0，不包含5）之间的随机数。

这里提下随机数的生成需要一个默认的种子，这个种子其实是一个 long 类型的数字，这个种子要么在 Random 的时候通过构造函数指定，那么默认构造函数内部会生成一个默认的值，有了默认的种子后，如何生成随机数呢？

```
    public int nextInt(int bound) {
        //(3)参数检查
        if (bound <= 0)
            throw new IllegalArgumentException(BadBound);
```

```
        //(4)根据老的种子生成新的种子
        int r = next(31);
        //(5)根据新的种子计算随机数
        ...
        return r;
    }
```

如上代码可知新的随机数的生成需要两个步骤：

- 首先需要根据老的种子生成新的种子。
- 然后根据新的种子来计算新的随机数。

其中步骤（4）我们可以抽象为 seed=f(seed)，其中 f 是一个固定的函数，比如 `seed=f(seed)=a*seed+b;`，步骤（5）也可以抽象为 g(seed,bound)，其中 g 是一个固定的函数，比如 `g(seed,bound)=(int)((bound * (long)seed) >> 31);`。在单线程情况下每次调用 nextInt 都是根据老的种子计算出来新的种子，这是可以保证随机数产生的随机性的。但是在多线程下多个线程可能都拿同一个老的种子去执行步骤（4）计算新的种子，这会导致多个线程产生的新种子是一样的，由于步骤（5）算法是固定的，所以会导致多个线程产生相同的随机值，这并不是我们想要的。

所以需要保证步骤（4）的原子性，也就是说多个线程在根据同一个老种子计算新种子时候，第一个线程的新种子计算出来后，第二个线程要丢弃自己老的种子，要使用第一个线程的新种子来计算自己的新种子，依次类推，只有保证了这个，才能保证多线程下产生的随机数是随机的。Random 函数使用一个原子变量达到了这个效果，在创建 Random 对象时候初始化的种子就保存到了种子原子变量里面，下面看下 next() 代码：

```
    protected int next(int bits) {
        long oldseed, nextseed;
        AtomicLong seed = this.seed;
        do {
            //(6)
            oldseed = seed.get();
            //(7)
            nextseed = (oldseed * multiplier + addend) & mask;
            //(8)
        } while (!seed.compareAndSet(oldseed, nextseed));
        //(9)
        return (int)(nextseed >>> (48 - bits));
    }
```

- 代码（6）获取当前原子变量种子的值；
- 代码（7）根据当前种子值计算新的种子；
- 代码（8）使用 CAS 操作，使用新的种子去更新老的种子，多线程下可能多个线程都同时执行到了代码（6），那么可能多个线程都拿到的当前种子的值是同一个，然后执行步骤（7）计算的新种子也都是一样的，但是步骤（8）的 CAS 操作会保证只有一个线程可以更新老的种子为新的，失败的线程会通过循环重新获取更新后的种子作为当前种子去计算老的种子，可见这里解决了上面提到的问题，也就保证了

随机数的随机性。
- 代码（9）则使用固定算法根据新的种子计算随机数。

总结：每个 Random 实例里面有一个原子性的种子变量用来记录当前的种子的值，当要生成新的随机数时候要根据当前种子计算新的种子并更新回原子变量。多线程下使用单个 Random 实例生成随机数时候，多个线程同时计算新的种子时候会竞争同一个原子变量的更新操作，由于原子变量的更新是 CAS 操作，同时只有一个线程会成功，所以会造成大量线程进行自旋重试，这是会降低并发性能的，所以 ThreadLocalRandom 应运而生。

ThreadLocalRandom 类

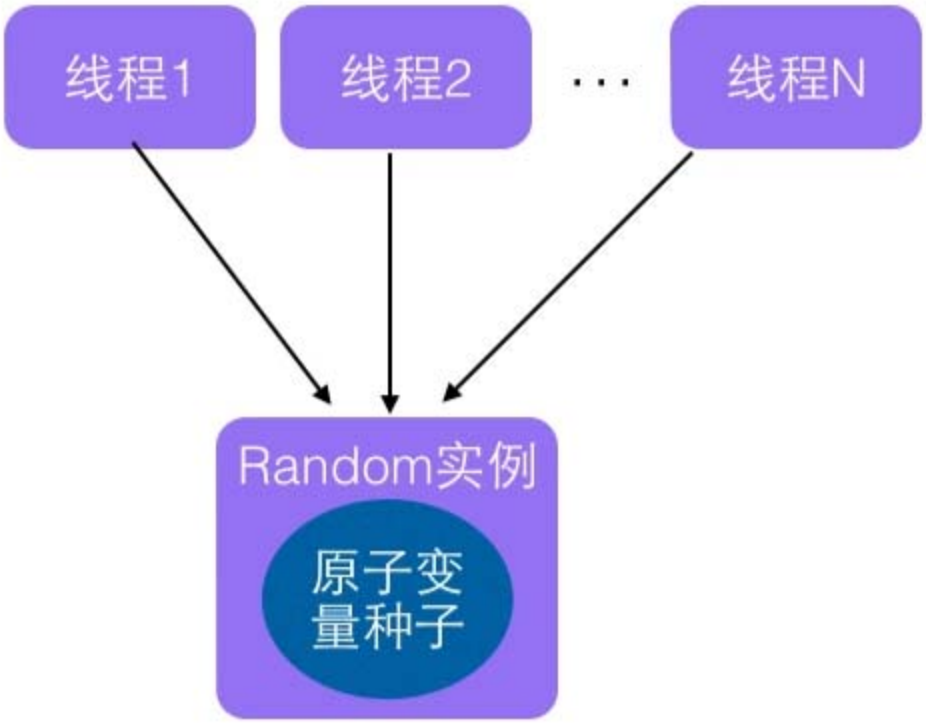
为了解决多线程高并发下 Random 的缺陷，JUC 包下新增了 ThreadLocalRandom 类，下面首先看下它如何使用：

```
public class RandomTest {

    public static void main(String[] args) {
        //(10)获取一个随机数生成器
        ThreadLocalRandom random =  ThreadLocalRandom.current();

        //(11)输出10个在0-5（包含0，不包含5）之间的随机数
        for (int i = 0; i < 10; ++i) {
            System.out.println(random.nextInt(5));
        }
    }
}
```

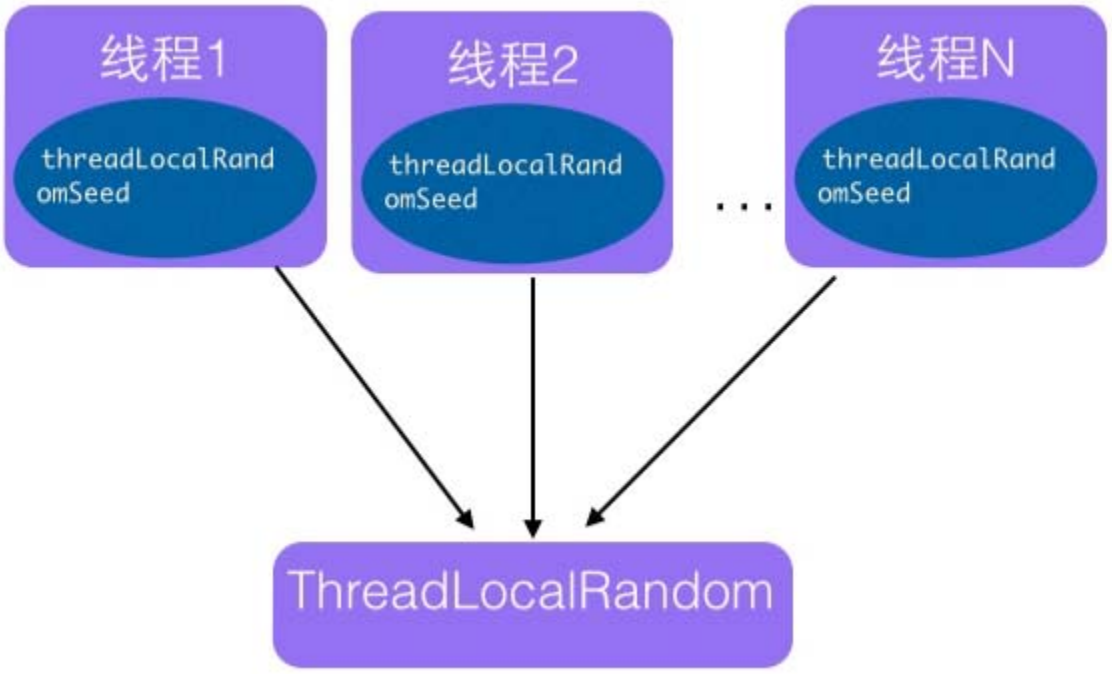
如上代码（10）调用 ThreadLocalRandom.current() 来获取当前线程的随机数生成器。

下面来分析下 ThreadLocalRandom 的实现原理。从名字看会让我们联想到《Java 并发编程之美：基础篇》中讲解的 ThreadLocal，ThreadLocal 的出现就是为了解决多线程下变量的隔离问题，让每一个线程拷贝一份变量，每个线程对变量进行操作时候实际是操作自己本地内存里面的拷贝。

实际上 ThreadLocalRandom 的实现也是这个原理，Random 的缺点是多个线程会使用原子性种子变量，会导致对原子变量更新的竞争，如下图：

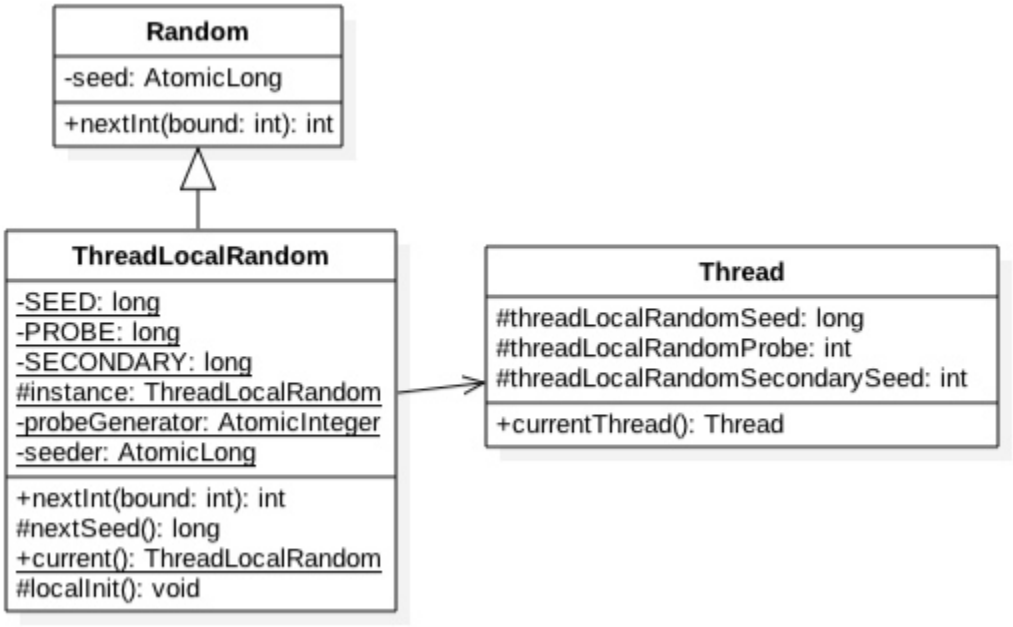


那么如果每个线程维护自己的一个种子变量，每个线程生成随机数时候根据自己老的种子计算新的种子，并使用新种子更新老的种子，然后根据新种子计算随机数，就不会存在竞争问题，这会大大提高并发性能，如下图 ThreadLocalRandom 原理：



源码分析

首先看下 ThreadLocalRandom 的类图结构：



可知 ThreadLocalRandom 继承了 Random 并重写了 nextInt 方法，ThreadLocalRandom 中并没有使用继承自 Random 的原子性种子变量。

ThreadLocalRandom 中并没有具体存放种子，具体的种子是存放到具体的调用线程的 threadLocalRandomSeed 变量里面的，ThreadLocalRandom 类似于 ThreadLocal类 就是个工具类。当线程调用 ThreadLocalRandom 的 current 方法时候 ThreadLocalRandom 负责初始化调用线程的 threadLocalRandomSeed 变量，也就是初始化种子。

当调用 ThreadLocalRandom 的 nextInt 方法时候，实际上是获取当前线程的 threadLocalRandomSeed 变量作为当前种子来计算新的种子，然后更新新的种子到当前线程的 threadLocalRandomSeed 变量，然后在根据新种子和具体算法计算随机数。

这里需要注意的是 threadLocalRandomSeed 变量就是 Thread 类里面的一个普通 long 变量，并不是原子性变量，其实道理很简单，因为这个变量是线程级别的，根本不需要使用原子性变量，如果还是不理解可以思考下 ThreadLocal 的原理。

其中变量 seeder 和 probeGenerator 是两个原子性变量，在初始化调用线程的种子和探针变量时候用到，每个线程只会使用一次。

另外变量 instance 是个 ThreadLocalRandom 的一个实例，该变量是 static 的，当多线程通过 ThreadLocalRandom 的 current 方法获取 ThreadLocalRandom 的实例时候其实获取的是同一个，但是由于具体的种子是存放到线程里面的，所以 ThreadLocalRandom 的实例里面只是与线程无关的通用算法，所以是线程安全的。

下面看看 ThreadLocalRandom 的主要代码实现逻辑。

- Unsafe 机制的使用，具体的会在高级篇之二里面讲解。

```
private static final sun.misc.Unsafe UNSAFE;
private static final long SEED;
private static final long PROBE;
```

```
    private static final long SECONDARY;
    static {
        try {
            //获取unsafe实例
            UNSAFE = sun.misc.Unsafe.getUnsafe();
            Class<?> tk = Thread.class;
            //获取Thread类里面threadLocalRandomSeed变量在Thread实例
里面偏移量
            SEED = UNSAFE.objectFieldOffset
                (tk.getDeclaredField("threadLocalRandomSeed"));
            //获取Thread类里面threadLocalRandomProbe变量在Thread实例
里面偏移量
            PROBE = UNSAFE.objectFieldOffset
                (tk.getDeclaredField("threadLocalRandomProbe"));
            //获取Thread类里面threadLocalRandomProbe变量在Thread实例
里面偏移量，这个值在后面讲解的LongAdder里面会用到
            SECONDARY = UNSAFE.objectFieldOffset

(tk.getDeclaredField("threadLocalRandomSecondarySeed"));
        } catch (Exception e) {
            throw new Error(e);
        }
    }
```

- ThreadLocalRandom current() 方法：该方法获取 ThreadLocalRandom 实例，并初始化调用线程中 threadLocalRandomSeed 和 threadLocalRandomProbe 变量。

```
    static final ThreadLocalRandom instance = new
ThreadLocalRandom();
    public static ThreadLocalRandom current() {
        //(12)
        if (UNSAFE.getInt(Thread.currentThread(), PROBE) == 0)
            //(13)
            localInit();
        //(14)
        return instance;
    }



    static final void localInit() {
        int p = probeGenerator.addAndGet(PROBE_INCREMENT);
        int probe = (p == 0) ? 1 : p; // skip 0
        long seed = mix64(seeder.getAndAdd(SEEDER_INCREMENT));
        Thread t = Thread.currentThread();
        UNSAFE.putLong(t, SEED, seed);
        UNSAFE.putInt(t, PROBE, probe);
    }
```

如上代码（12）如果当前线程中 threadLocalRandomProbe 变量值为0（默认情况下线程的这个变量为0），说明当前线程第一次调用 ThreadLocalRandom 的 current 方法，那么就需要调用 localInit 方法计算当前线程的初始化种子变量。这里设计为了延迟初始化，不需要使用随机数功能时候 Thread 类中的种子变量就不需要被初始化，这是一种优化。

代码（13）首先计算根据 probeGenerator 计算当前线程中 threadLocalRandomProbe 的初始化值，然后根据 seeder 计算当前线程的初始化种子，然后把这两个变量设置到当前线程。

代码（14）返回 ThreadLocalRandom 的实例，需要注意的是这个方法是静态方法，多个线程返回的是同一个 ThreadLocalRandom 实例。

- int nextInt(int bound) 方法：计算当前线程的下一个随机数。

```
public int nextInt(int bound) {
    //(15)参数校验
    if (bound <= 0)
        throw new IllegalArgumentException(BadBound);
    //(16) 根据当前线程中种子计算新种子
    int r = mix32(nextSeed());
    //(17)根据新种子和bound计算随机数
    int m = bound - 1;
    if ((bound & m) == 0) // power of two
        r &= m;
    else { // reject over-represented candidates
        for (int u = r >>> 1;
             u + m - (r = u % bound) < 0;
             u = mix32(nextSeed()) >>> 1)
            ;
    }
    return r;
}
```

如上代码逻辑步骤与 Random 相似，我们重点看下 nextSeed() 方法：

```
final long nextSeed() {
    Thread t; long r; //
    UNSAFE.putLong(t = Thread.currentThread(), SEED,
                   r = UNSAFE.getLong(t, SEED) + GAMMA);
    return r;
}
```

如上代码首先使用 r = UNSAFE.getLong(t, SEED) 获取当前线程中 threadLocalRandomSeed 变量的值，然后在种子的基础上累加 GAMMA 值作为新种子，然后使用 UNSAFE 的 putLong 方法把新种子放入当前线程的 threadLocalRandomSeed 变量。

注：本节首先讲解了 Random 的实现原理以及介绍了 Random 在多线程下存在竞争种子原子变量更新操作失败后自旋等待的缺点，从而引出 ThreadLocalRandom 类，ThreadLocalRandom 使用 ThreadLocal 的原理，让每个线程内持有一个本地的种子变量，该种子变量只有在使用随机数时候才会被初始化，多线程下计算新种子时候是根据自己线程内维护的种子变量进行更新，从而避免了竞争。

Spring Request Scope 作用域 Bean 中 ThreadLocal 的使用

我们知道 Spring 中在 XML 里面配置 Bean 的时候可以指定 scope 属性来配置该 Bean 的作用域为 singleton、prototype、request、session 等，其中作用域为 request 的实现原理就是使用 ThreadLocal 实现的。

如果你想让你 Spring 容器里的某个 Bean 拥有 Web 的某种作用域，则除了需要 Bean 级上配置相应的 scope 属性，还必须在 web.xml 里面配置如下：

```
<listener>
        <listener-
class>org.springframework.web.context.request.RequestContextListe
ner</listener-class>
</listener>
```

这里主要看 RequestContextListener 的两个方法：

```
public void requestInitialized(ServletRequestEvent
requestEvent)
```

和

```
public void requestDestroyed(ServletRequestEvent requestEvent)
```

当一个 Web 请求过来时候会执行 requestInitialized 方法：

```
    public void requestInitialized(ServletRequestEvent
requestEvent) {
                .......
        HttpServletRequest request = (HttpServletRequest)
requestEvent.getServletRequest();
        ServletRequestAttributes attributes = new
ServletRequestAttributes(request);
        request.setAttribute(REQUEST_ATTRIBUTES_ATTRIBUTE,
attributes);
```

```
            LocaleContextHolder.setLocale(request.getLocale());
                    //设置属性到threadlocal变量
            RequestContextHolder.setRequestAttributes(attributes);
        }



        public static void setRequestAttributes(RequestAttributes
    attributes) {

            setRequestAttributes(attributes, false);
        }
        public static void setRequestAttributes(RequestAttributes
    attributes, boolean inheritable) {
            if (attributes == null) {
                resetRequestAttributes();
            }
            else {
                            //默认inheritable=false
                if (inheritable) {

    inheritableRequestAttributesHolder.set(attributes);
                    requestAttributesHolder.remove();
                }
                else {
                    requestAttributesHolder.set(attributes);
                    inheritableRequestAttributesHolder.remove();
                }
            }
        }
```

由于默认 inheritable 为 FALSE，我们的属性值都放到了 requestAttributesHolder 里面，而它的定义是：

```
        private static final ThreadLocal<RequestAttributes>
    requestAttributesHolder =
                new NamedThreadLocal<RequestAttributes>("Request
    attributes");

        private static final ThreadLocal<RequestAttributes>
    inheritableRequestAttributesHolder =
                new NamedInheritableThreadLocal<RequestAttributes>
    ("Request context");
```

其中 `NamedThreadLocal<T> extends ThreadLocal<T>`，所以不具有继承性。

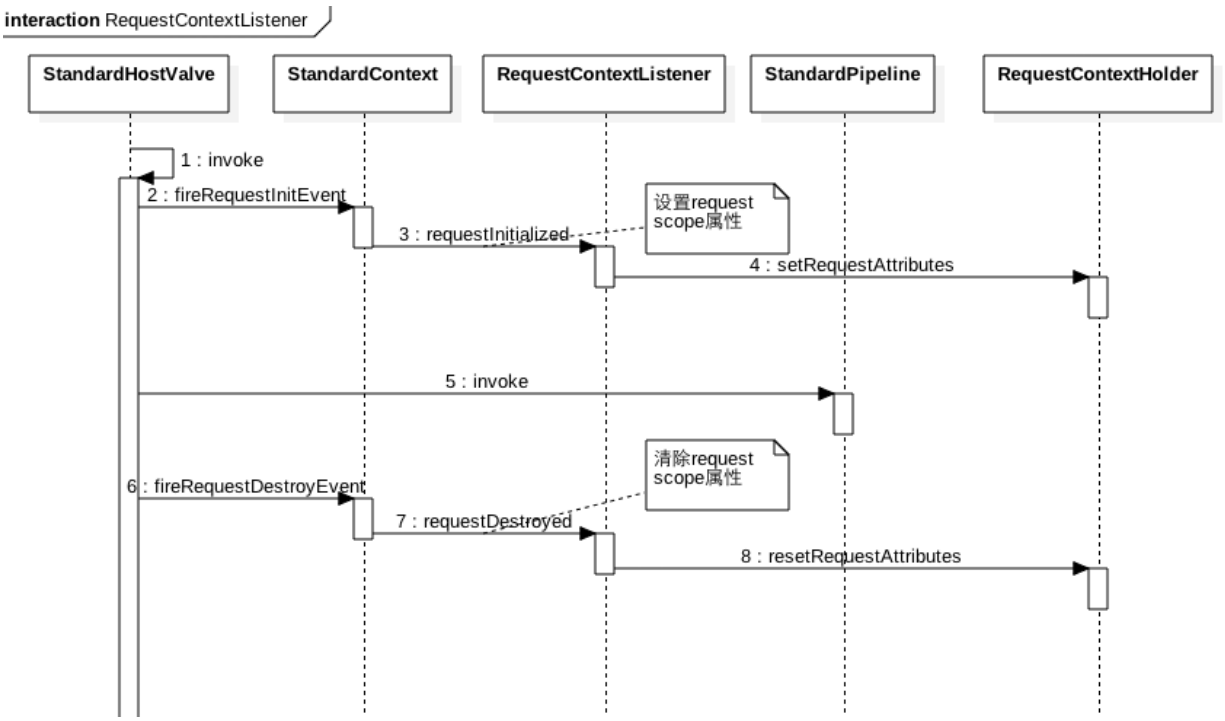
`NamedInheritableThreadLocal<T> extends InheritableThreadLocal<T>`，所以具有继承性，所以默认放入到 RequestContextHolder 里面的属性值在子线程中获取不

到。

当请求结束时候调用 requestDestroyed 方法，代码如下：

```
    public void requestDestroyed(ServletRequestEvent
requestEvent) {
        ServletRequestAttributes attributes =
                (ServletRequestAttributes)
requestEvent.getServletRequest().getAttribute(REQUEST_ATTRIBUTES_
ATTRIBUTE);
        ServletRequestAttributes threadAttributes =
                (ServletRequestAttributes)
RequestContextHolder.getRequestAttributes();
        if (threadAttributes != null) {
            // We're assumably within the original request
thread...
            if (attributes == null) {
                attributes = threadAttributes;
            }
                        //请求结束则清除当前线程的线程变量。
            LocaleContextHolder.resetLocaleContext();
            RequestContextHolder.resetRequestAttributes();
        }
        if (attributes != null) {
            attributes.requestCompleted();
        }
    }
```

下面从时序图看下一次 Web 请求调用逻辑如何：



也就是说每次发起一个 Web 请求在 Tomcat 中 context（具体应用）处理前，host 匹配后都会去设置下 RequestContextHolder 属性，让 requestAttributesHolder 不为空，在请求结束时候会清除。

总结：默认情况下放入 RequestContextHolder 里面的属性子线程访问不到。Spring 的 request 作用域的 Bean 是使用 threadlocal 实现的。

模拟请求的简单 Demo

- web.xml里面配置如下。

因为是 request 作用域，所以必须是 Web 项目，并且需要配置 RequestContextListener 到 web.xml。

```
<listener>
        <listener-
class>org.springframework.web.context.request.RequestContextListe
ner</listener-class>
</listener>
```

- 注入一个 request 作用域 bean 到 IOC 容器。

```
 <bean id="requestBean" class="com.zlx.test.RequestBean"
        scope="request">
        <property name="name" value="jiaduo" />
        <aop:scoped-proxy />
    </bean>
```

- 测试RPC。

```
@WebResource("/testService")
public class TestRpc {

    @Autowired
    private RequestBean requestInfo;

    @ResourceMapping("test")
    public ActionResult test(ErrorContext context) {
        ActionResult result = new ActionResult();

        pvgInfo.setName("jiaduo");
        String name = requestInfo.getName();
        result.setValue(name);

        return result;
```

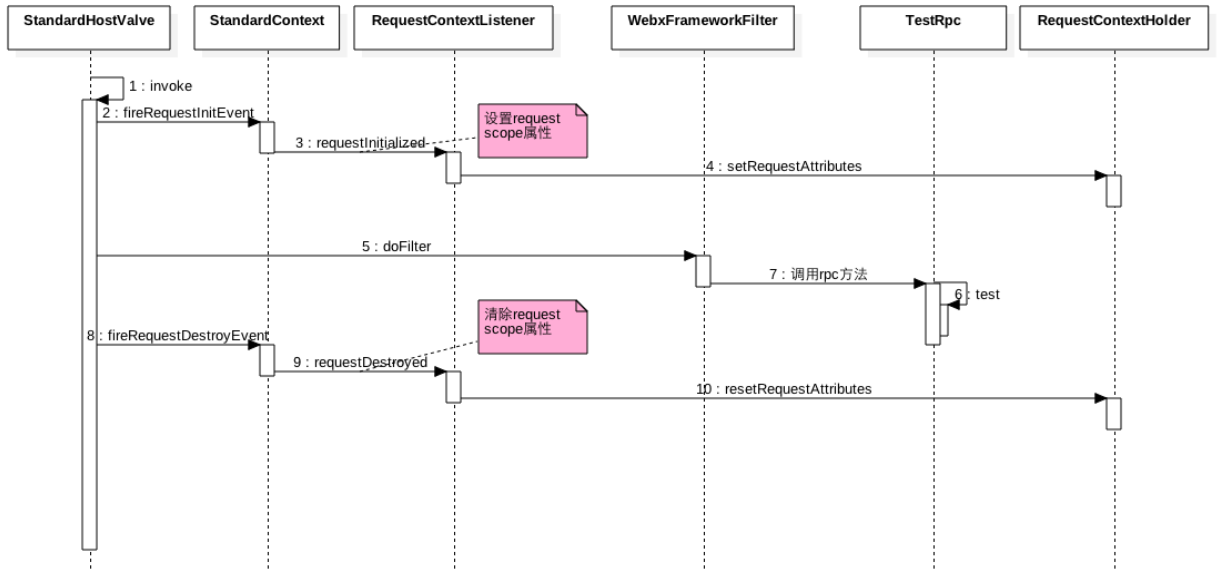
```
        }
    }
```

如上首先配置 RequestContextListener 到 web.xml 里面，然后注入了 Request 作用域的 RequestBean 的实例到 IOC 容器，最后 TestRpc 内注入了 RequestBean 的实例，方法 test 首先调用了 requestInfo 的方法 setName 设置 name 属性，然后获取 name 属性并返回。

这里如果 requestInfo 对象是单例的，那么多个线程同时调用 test 方法后，每个线程都是设置-获取的操作，这个操作不是原子性的，会导致线程安全问题。而这里声明的作用域为 request 级别，也是每个线程都有一个 requestInfo 的本地变量。

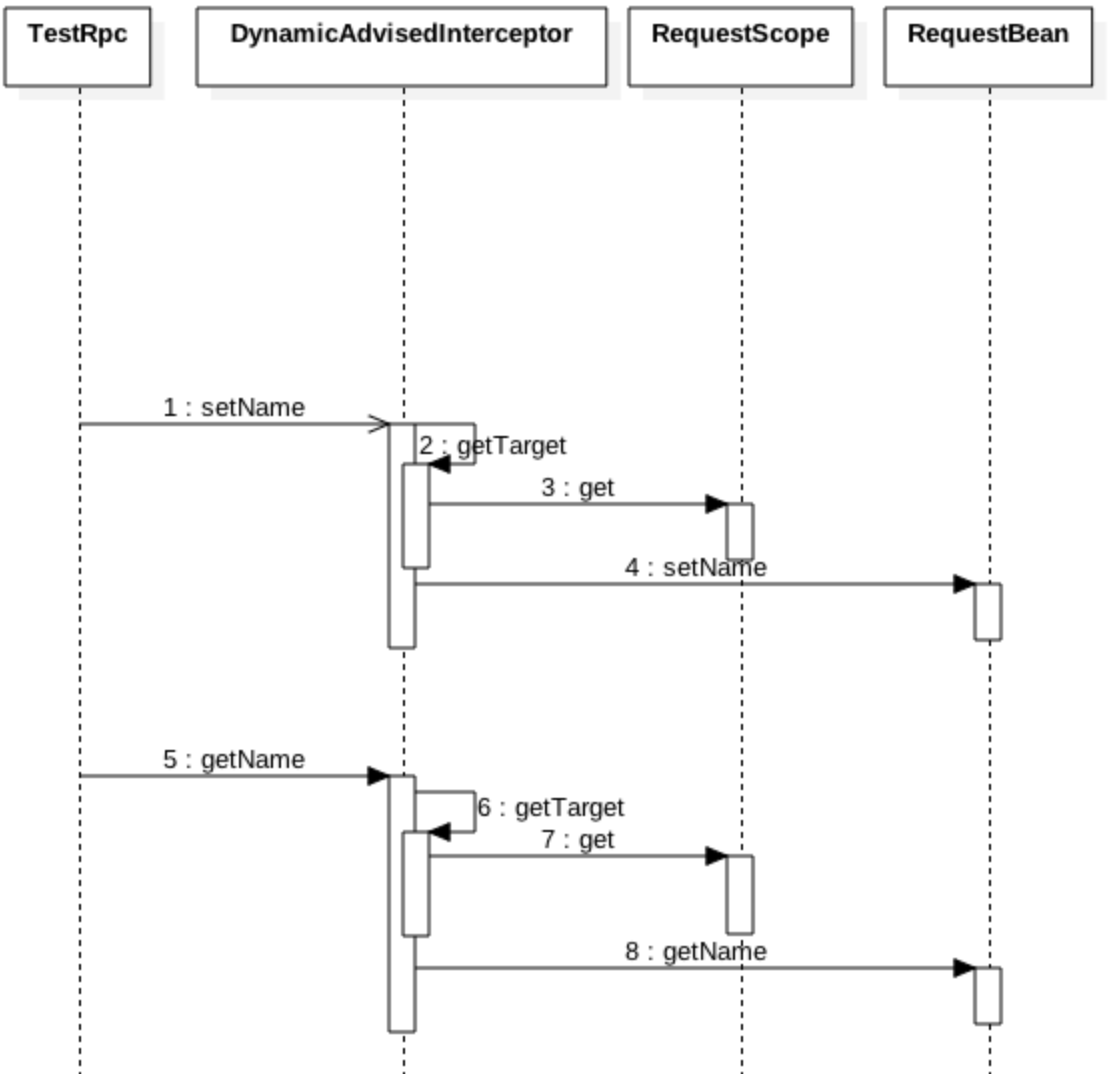
原理剖析

下面分析下当调用这个 RPC 方法请求时候时序图为：



这个时序图与上节的类似，只是这里使用 WebX 搭建的 Web 项目，所以多了的 WebXFrameworkFilter，转发请求到我们定义的 TestRpc。

下面着重看下调用 test 时候发生了什么：



其实前面创建的 requestInfo 是被经过 CGliB 代理后的（感兴趣的童鞋可以研究下 ScopedProxyFactoryBean 这类），所以这里调用 setName 或者 getName 时候会被 DynamicAdvisedInterceptor 拦截的，拦击器里面最终会调用到 RequestScope 的 get 方法获取当前线程持有的本地变量。

RequestScope 的 get 方法代码如下：

```
    public Object get(String name, ObjectFactory objectFactory) {

        RequestAttributes attributes =
RequestContextHolder.currentRequestAttributes();//（1）
        Object scopedObject = attributes.getAttribute(name,
getScope());
        if (scopedObject == null) {
            scopedObject = objectFactory.getObject();//（2）
            attributes.setAttribute(name, scopedObject,
getScope());//（3）
        }
        return scopedObject;
    }
```

可知当发起一个请求时候，首先会通过 RequestContextListener.requestInitialized 里面调用 RequestContextHolder.setRequestAttributess 设置 requestAttributesHolder。

然后请求被路由到 TestRpc 的 test 方法后，test 方法内第一次调用 setName 方法时候，最终会调用 RequestScope.get() 方法，get 方法内代码（1）获取通过 RequestContextListener.requestInitialized 设置的线程本地变量 requestAttributesHolder 保存的属性集的值。

然后看该属性集里面是否有名字为 requestInfo 的属性，由于是第一次调用，所以不存在，所以会执行代码（2）让 Spring 创建一个 RequestInfo 对象，然后设置到属性集 attributes，也就是保存到了当前请求线程的本地内存里面了。然后返回创建的对象，调用创建对象的 setName。

然后 test 方法内紧接着调用了 getName 方法，最终会调用 RequestScope.get() 方法，get 方法内代码（1）获取通过 RequestContextListener.requestInitialized 设置的线程本地变量 RequestAttributes，然后看该属性集里面是否有名字为 requestInfo 的属性，由于是第一次调用 setName 时候已经设置名字为 requestInfo 的 bean 到 ThreadLocal 变量里面了，并且调用 setName 和 getName 的是同一个线程，所以这里直接返回了调用 setName 时候创建的 RequestInfo 对象，然后调用它的 getName 方法。

总结

本文通过循序渐进的方式，先讲解了 ThreadLocal 的简单使用，然后讲解了 ThreadLocal 的实现原理，并指出 ThreadLocal 不支持继承性；然后紧接着讲解了 InheritableThreadLocal 是如何补偿了 ThreadLocal 不支持继承的特性；然后讲解了 ThreadLocalRandom 是如何借鉴 ThreadLocal 的思想补充了 Random 的不足；最后简单的介绍了 Spring 框架中如何使用 ThreadLocal 实现了 Reqeust Scope 的 Bean。在后面的并发编制之美——项目实践与常见问题解答中，我们会深入讲解使用 ThreadLocal 导致内存泄露的案例，敬请期待。