

## 从Math.random到ThreadLocalRandom

2015-11-18 02:20 feiying 0 阅读 79

我们一般在使用随机数的时候，都会默认使用使用Math.random方法来随便搞一个随机数用用，没有想那么多，

如果是传统企业应用的话，压力很小，程序写得烂点也无妨，这没有任何的问题，

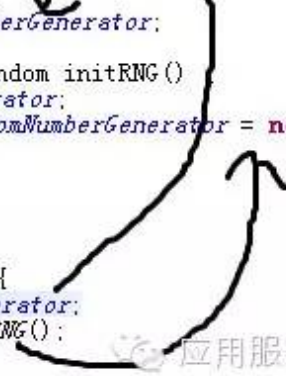
但是，现在做互联网应用，要求程序精益求精，程序稍微差一点都不行，

首先，分析分析Math.random的源码：

```
private static Random randomNumberGenerator;

private static synchronized Random initRNG() {
    Random rnd = randomNumberGenerator;
    return (rnd == null) ? (randomNumberGenerator = new Random()) : rnd;
}

public static double random() {
    Random rnd = randomNumberGenerator;
    if (rnd == null) rnd = initRNG();
    return rnd.nextDouble();
}
```



整体流程是这样的，一共分三个步骤：

- 在Math类中存在一个Random的generator，当前JVM中只要有调用过这个generator，那么这个generator就会被赋值，
- 如果上述的generator，没有被赋值的话，那么调用initRNG方法，初始化一个Random实例，然后再给这个generator赋值
- 最后调用generator的nextdouble方法，返回随机数；

对于Random类来说，这个是一个util包中的类，我们了解到随机数中是需要种子的，

回想到在C语言中，我们要使用随机数，我们必须要先给其喂一个种子，然后再进行计算，是非常麻烦的，

而Java中的这个Random类就省略了这个麻烦，种子是随机生成的：

下载《开发者大全》

下载 (/download/dev.apk)





对于种子是两种方式，

第一种是默认的Random构造方法，种子就是搞一个特别长的seedUniquifier随机数与当前的时间求与作为默认的种子的输入，然后最终根据一系列的求与或位运算算出种子；

第二种是，你传给Random一个seed，后续的求种子的流程和第一种是一样，

二者之间的区别就在于，种子的算法是不同的；

再来看看nextDouble方法：

```

*/
public double nextDouble() {
    return (((long)(next(26)) << 27) + next(27))
        / (double)(1L << 53);
}

```

应用服务器技术讨论圈

通过上述算法计算出来的随机数，能保证很难重复，

但通过这个算法，也不难发现，其仍旧和操作系统的系统调用一样，是伪随机数：

```
public class Random

    extends
    Object

    implements
    Serializable
```

此类的实例用于生成伪随机数流。此类使用 48 位的种子，使用线性同余公式 (linear congruential form) 对其进行了修改 (请参阅 Donald Knuth 的 *The Art of Computer Programming, Volume 3* 第 3.2.1 节)。

		生成下一个伪随机数。
boolean	<code>nextBoolean()</code>	返回下一个伪随机数，它是取自此随机数生成器序列的均匀分布的 boolean 值。
void	<code>nextBytes(byte[] bytes)</code>	生成随机字节并将其置于用户提供的 byte 数组中。
double	<code>nextDouble()</code>	返回下一个伪随机数，它是取自此随机数生成器序列的、在 0.0 和 1.0 之间均匀分布的 double 值。
float	<code>nextFloat()</code>	返回下一个伪随机数，它是取自此随机数生成器序列的、在 0.0 和 1.0 之间均匀分布的 float 值。

对于这种情况，也没有办法，如果想获得更为随机的数，那就需要找专门的开源项目通过数学公式进行找寻，

我们这里只简单讨论其实现，这里就不再赘余，是否随机的问题；

看过Random的实现之后，回过头来，我们来看一下Math.random为什么在多线程环境下不好用。

```
private static Random randomNumberGenerator;
private static synchronized Random initRNG() {
    Random rnd = randomNumberGenerator;
    return (rnd == null) ? (randomNumberGenerator = new Random()) : rnd
}
```

注意上图中的红色部分，static synchronized，==》这是非常夸张的性能损耗，相当于对类的内存区域进行锁定，不是对象锁，而且锁还是synchronized，可想而知，当你每一次线程调用，每个线程都要执行一段逻辑中，都占据了Math这个类，无论你下面的业务逻辑怎么无锁，只要调用这个方法，只有一个线程可以进入，而实际上，我们仅仅是为了算出一个随机数，并没有线程同步的需求，

=====》所以，这个Math.random一定尽量保证不要使用；

下载《开发者大全》

下载 (/download/dev.apk)



解决办法是，看看能不能将锁给去掉，或者是不要做到类上，甚至可以每一个线程给一个Random，这就是ThreadLocalRandom的实现了：

```

/**
 * public class ThreadLocalRandom extends Random {
 * // same constants as Random, but must be redeclared because private
 * private static final long multiplier = 0x5DEECE66DL;
 * private static final long addend = 0xBL;
 * private static final long mask = (1L << 48) - 1;
 *
 * /**
 *  * The random seed. We can't use super.seed.
 *  */
 * private long rnd;
 *
 * /**
 *  * Initialization flag to permit calls to setSeed to succeed only
 *  * while executing the Random constructor. We can't allow others
 *  * since it would cause setting seed in one part of a program to
 *  * unintentionally impact other usages by the thread.
 *  */
 * boolean initialized;
 *
 * /**
 *  * The actual ThreadLocal
 *  */
 * private static final ThreadLocal<ThreadLocalRandom> localRandom =
 *     new ThreadLocal<ThreadLocalRandom>() {
 *         protected ThreadLocalRandom initialValue() {
 *             return new ThreadLocalRandom();
 *         }
 *     };
 *
 * /**
 *  * Returns the current thread's {@code ThreadLocalRandom}.
 *  *
 *  * @return the current thread's {@code ThreadLocalRandom}
 *  */
 * public static ThreadLocalRandom current() {
 *     return localRandom.get();
 * }

```

每个线程首先应该调用这个current方法，  
返回ThreadLocal的ThreadLocalRandom对象

第一步，因为每一个线程的ThreadLocalRandom对象是通过static方法get出来的，所以需要调用current方法；

这相当于每一个线程一个ThreadLocalRandom对象，从这点上就没有线程同步对象，每个线程算自己的线程的随机数；

其次，需要关注的是，种子的rnd因为每个线程不同，不能再使用父类的super的seed了，

因此，看到上述的红色部分，每一个ThreadLocalRandom对象关联一个rnd，作为种子的输入：

```
/**
 * Constructor called only by localRandom.initialValue.
 */
ThreadLocalRandom() {
    super();
    initialized = true;
}
```

通过调用super方法，会计算出每一个线程的rnd的种子，赋给每一个线程的ThreadLocal中的ThreadLocalRandom对象。

当rnd被赋值以后，每一个线程种子输入已经不同了，调用next方法，

```
protected int next(int bits) {
    rnd = (rnd * multiplier + addend) & mask;
    return (int) (rnd >>> (48-bits));
}
```

这里看到，再一次对rnd进行随机算法的阶段，最终可以计算出，每一个线程自己的随机数；

从上述的程序步骤可以分析得出，利用ThreadLocalRandom可以使随机数更加随机一些；

## 总结：

**Math.random因为static synchronized，锁类，多线程下效率很低，  
推荐采用每个线程一个的ThreadLocalRandom！**

分享：

阅读 79 0

应用服务器技术讨论圈 更多文章

东方通加码大数据业务 拟募资8亿收购微智信业 (/html/308/201504/206211355/1.html)

玩转Netty – 从Netty3升级到Netty4 (/html/308/201504/206233287/1.html)

金蝶中间件2015招聘来吧！Come on！ (/html/308/201505/206307460/1.html)

GlassFish 4.1 发布，J2EE 应用服务器 (/html/308/201505/206323120/1.html)

Tomcat对keep-alive的实现逻辑 (/html/308/201505/206357679/1.html)

下载《开发者大全》

下载 (/download/dev.apk)



猜您喜欢
新闻聚合阅读应用Facebook Paper的幕后功臣Origami (/html/216/201402/200011428/1.html)
测试空间走进我们身边，为在校大学生进行职业规划讲座 (/html/355/201605/2651527663/1.html)
什么是MBH树莓派智能机器人黑客大赛 (/html/454/201604/401947003/1.html)
给女朋友的 iOS 开发教程 8 UITableView (/html/423/201512/400974514/1.html)
重建中国.NET生态系统 (/html/359/201504/204904341/1.html)