

目录	
第1章 基础	
01 开篇词：为什么学习本专栏	
02 String、Long 源码解析和面试题	
03 Java 常用关键字理解	
04 Arrays、Collections、Objects 常用方法源码解析	
第2章 集合	
05 ArrayList 源码解析和设计思路	
06 LinkedList 源码解析	
07 List 源码会问哪些面试题	
08 HashMap 源码解析	
09 TreeMap 和 LinkedHashMap 核心源码解析	
10 Map源码会问哪些面试题	
11 HashSet、TreeSet 源码解析	
12 彰显细节：看集合源码对我们实际工作的帮助和应用	
13 差异对比：集合在 Java 7 和 8 有何不同和改进	
14 简化工作：Guava Lists Maps 实际工作运用和源码	
第3章 并发集合类	
15 CopyOnWriteArrayList 源码解析和设计思路	
16 ConcurrentHashMap 源码解析和设计思路	
17 并发 List、Map源码面试题	
18 场景集合：并发 List、Map的应用	

20 SynchronousQueue 源码解析

更新时间：2019-10-15 11:19:50



“只有在那崎岖的小路上不畏艰险奋勇攀登的人,才有希望达到光辉的顶点。
——马克思

果断更，请联系QQ/微信6426006

SynchronousQueue 是比较独特的队列，其本身是没有容量大小，比如我放一个数据到队列中，我是不能够立马返回的，我必须等待别人把我放进去的数据消费掉了，才能够返回。SynchronousQueue 在消息队列技术中间件中被大量使用，本文就来从底层实现来看下 SynchronousQueue 到底是如何做到的。

1 整体架构

SynchronousQueue 的整体设计比较抽象，在内部抽象出了两种算法实现，一种是先入先出的队列，一种是后入先出的堆栈，两种算法被两个内部类实现，而直接对外的 put, take 方法的实现就非常简单，都是直接调用两个内部类的 transfer 方法进行实现，整体的调用关系如下图所示：

目录

838 //将e放进队列,直到有另外一个线程从队列中取走,成功结束,失败则等待
839 public void put(E e) throws InterruptedException {
840 //e为null,则抛出NullPointerException
841 if (e == null) throw new NullPointerException();
842 //一直等待
843 if (transferer.transfer(e, false, 0) == null)
844 Thread.interrupted();
845 throw new InterruptedException();
846 }
847 //直接对外的 put take 方法都比较简单,都是通过调用
848 //内部队列类 和 堆栈类 来实现的
849 //从队列中取数据并删除数据,成功返回,失败则抛出InterruptedException
850 public E take() throws InterruptedException {
851 E e = transferer.transfer(null, false, 0);
852 if (e != null)
853 return e;
854 Thread.interrupted();
855 throw new InterruptedException();
856 }
857 //cpu个数
858 static final int NCPUS = Runtime.getRuntime().availableProcessors();
859
860
861 /**
862 * The number of times to spin before blocking in timed wait.
863 * The value is empirically derived -- it works well across a
864 * variety of processors and OSes. Empirically, the best value
865 * seems not to vary with number of CPUs (beyond 2) so is just
866 * a constant.
867
868 * @since 1.5
869 */
870 private static final int SPIN = 1000;
871
872 /**
873 * Advance tail and wait for the head to be moved.
874 * @param t the tail
875 * @param s the stack
876 * @return the head
877 */
878 private static Object advanceTailAndWait(Object t, Object s)
879 {
880 Object x = awaitFulfill(s, e, timed, nanos);
881 if (x == s) {
882 // wait was cancelled
883 clean(t, s);
884 return null;
885 }
886 }
887
888 /**
889 * Clean the tail and stack.
890 * @param t the tail
891 * @param s the stack
892 */
893 private static void clean(Object t, Object s)
894 {
895 // ...
896 }
897
898 /**
899 * Transfer an element from the stack to the queue.
900 * @param e the element
901 * @param s the stack
902 * @param l the lock
903 * @return the element
904 */
905 private static E transfer(E e, boolean lono, E tTransferer)
906 {
907 // ...
908 }
909
910 /**
911 * Transfer an element from the queue to the stack.
912 * @param e the element
913 * @param l the lock
914 * @return the element
915 */
916 private static E transfer(E e, boolean lono, E tTransferer)
917 {
918 // ...
919 }
920
921 /**
922 * Transfer an element from the stack to the queue.
923 * @param e the element
924 * @param s the stack
925 * @param l the lock
926 * @return the element
927 */
928 private static E transfer(E e, boolean lono, E tTransferer)
929 {
930 // ...
931 }
932
933 /**
934 * Transfer an element from the queue to the stack.
935 * @param e the element
936 * @param l the lock
937 * @return the element
938 */
939 private static E transfer(E e, boolean lono, E tTransferer)
940 {
941 // ...
942 }
943
944 /**
945 * Transfer an element from the stack to the queue.
946 * @param e the element
947 * @param s the stack
948 * @param l the lock
949 * @return the element
950 */
951 private static E transfer(E e, boolean lono, E tTransferer)
952 {
953 // ...
954 }
955
956 /**
957 * Transfer an element from the queue to the stack.
958 * @param e the element
959 * @param l the lock
960 * @return the element
961 */
962 private static E transfer(E e, boolean lono, E tTransferer)
963 {
964 // ...
965 }
966
967 /**
968 * Transfer an element from the stack to the queue.
969 * @param e the element
970 * @param s the stack
971 * @param l the lock
972 * @return the element
973 */
974 private static E transfer(E e, boolean lono, E tTransferer)
975 {
976 // ...
977 }
978
979 /**
980 * Transfer an element from the queue to the stack.
981 * @param e the element
982 * @param l the lock
983 * @return the element
984 */
985 private static E transfer(E e, boolean lono, E tTransferer)
986 {
987 // ...
988 }
989
990 /**
991 * Transfer an element from the stack to the queue.
992 * @param e the element
993 * @param s the stack
994 * @param l the lock
995 * @return the element
996 */
997 private static E transfer(E e, boolean lono, E tTransferer)
998 {
999 // ...
1000 }
1001
1002 /**
1003 * Transfer an element from the queue to the stack.
1004 * @param e the element
1005 * @param l the lock
1006 * @return the element
1007 */
1008 private static E transfer(E e, boolean lono, E tTransferer)
1009 {
1010 // ...
1011 }
1012
1013 /**
1014 * Transfer an element from the stack to the queue.
1015 * @param e the element
1016 * @param s the stack
1017 * @param l the lock
1018 * @return the element
1019 */
1020 private static E transfer(E e, boolean lono, E tTransferer)
1021 {
1022 // ...
1023 }
1024
1025 /**
1026 * Transfer an element from the queue to the stack.
1027 * @param e the element
1028 * @param l the lock
1029 * @return the element
1030 */
1031 private static E transfer(E e, boolean lono, E tTransferer)
1032 {
1033 // ...
1034 }
1035
1036 /**
1037 * Transfer an element from the stack to the queue.
1038 * @param e the element
1039 * @param s the stack
1040 * @param l the lock
1041 * @return the element
1042 */
1043 private static E transfer(E e, boolean lono, E tTransferer)
1044 {
1045 // ...
1046 }
1047
1048 /**
1049 * Transfer an element from the queue to the stack.
1050 * @param e the element
1051 * @param l the lock
1052 * @return the element
1053 */
1054 private static E transfer(E e, boolean lono, E tTransferer)
1055 {
1056 // ...
1057 }
1058
1059 /**
1060 * Transfer an element from the stack to the queue.
1061 * @param e the element
1062 * @param s the stack
1063 * @param l the lock
1064 * @return the element
1065 */
1066 private static E transfer(E e, boolean lono, E tTransferer)
1067 {
1068 // ...
1069 }
1070
1071 /**
1072 * Transfer an element from the queue to the stack.
1073 * @param e the element
1074 * @param l the lock
1075 * @return the element
1076 */
1077 private static E transfer(E e, boolean lono, E tTransferer)
1078 {
1079 // ...
1080 }
1081
1082 /**
1083 * Transfer an element from the stack to the queue.
1084 * @param e the element
1085 * @param s the stack
1086 * @param l the lock
1087 * @return the element
1088 */
1089 private static E transfer(E e, boolean lono, E tTransferer)
1090 {
1091 // ...
1092 }
1093
1094 /**
1095 * Transfer an element from the queue to the stack.
1096 * @param e the element
1097 * @param l the lock
1098 * @return the element
1099 */
1100 private static E transfer(E e, boolean lono, E tTransferer)
1101 {
1102 // ...
1103 }
1104
1105 /**
1106 * Transfer an element from the stack to the queue.
1107 * @param e the element
1108 * @param s the stack
1109 * @param l the lock
1110 * @return the element
1111 */
1112 private static E transfer(E e, boolean lono, E tTransferer)
1113 {
1114 // ...
1115 }
1116
1117 /**
1118 * Transfer an element from the queue to the stack.
1119 * @param e the element
1120 * @param l the lock
1121 * @return the element
1122 */
1123 private static E transfer(E e, boolean lono, E tTransferer)
1124 {
1125 // ...
1126 }
1127
1128 /**
1129 * Transfer an element from the stack to the queue.
1130 * @param e the element
1131 * @param s the stack
1132 * @param l the lock
1133 * @return the element
1134 */
1135 private static E transfer(E e, boolean lono, E tTransferer)
1136 {
1137 // ...
1138 }
1139
1140 /**
1141 * Transfer an element from the queue to the stack.
1142 * @param e the element
1143 * @param l the lock
1144 * @return the element
1145 */
1146 private static E transfer(E e, boolean lono, E tTransferer)
1147 {
1148 // ...
1149 }
1150
1151 /**
1152 * Transfer an element from the stack to the queue.
1153 * @param e the element
1154 * @param s the stack
1155 * @param l the lock
1156 * @return the element
1157 */
1158 private static E transfer(E e, boolean lono, E tTransferer)
1159 {
1160 // ...
1161 }
1162
1163 /**
1164 * Transfer an element from the queue to the stack.
1165 * @param e the element
1166 * @param l the lock
1167 * @return the element
1168 */
1169 private static E transfer(E e, boolean lono, E tTransferer)
1170 {
1171 // ...
1172 }
1173
1174 /**
1175 * Transfer an element from the stack to the queue.
1176 * @param e the element
1177 * @param s the stack
1178 * @param l the lock
1179 * @return the element
1180 */
1181 private static E transfer(E e, boolean lono, E tTransferer)
1182 {
1183 // ...
1184 }
1185
1186 /**
1187 * Transfer an element from the queue to the stack.
1188 * @param e the element
1189 * @param l the lock
1190 * @return the element
1191 */
1192 private static E transfer(E e, boolean lono, E tTransferer)
1193 {
1194 // ...
1195 }
1196
1197 /**
1198 * Transfer an element from the stack to the queue.
1199 * @param e the element
1200 * @param s the stack
1201 * @param l the lock
1202 * @return the element
1203 */
1204 private static E transfer(E e, boolean lono, E tTransferer)
1205 {
1206 // ...
1207 }
1208
1209 /**
1210 * Transfer an element from the queue to the stack.
1211 * @param e the element
1212 * @param l the lock
1213 * @return the element
1214 */
1215 private static E transfer(E e, boolean lono, E tTransferer)
1216 {
1217 // ...
1218 }
1219
1220 /**
1221 * Transfer an element from the stack to the queue.
1222 * @param e the element
1223 * @param s the stack
1224 * @param l the lock
1225 * @return the element
1226 */
1227 private static E transfer(E e, boolean lono, E tTransferer)
1228 {
1229 // ...
1230 }
1231
1232 /**
1233 * Transfer an element from the queue to the stack.
1234 * @param e the element
1235 * @param l the lock
1236 * @return the element
1237 */
1238 private static E transfer(E e, boolean lono, E tTransferer)
1239 {
1240 // ...
1241 }
1242
1243 /**
1244 * Transfer an element from the stack to the queue.
1245 * @param e the element
1246 * @param s the stack
1247 * @param l the lock
1248 * @return the element
1249 */
1250 private static E transfer(E e, boolean lono, E tTransferer)
1251 {
1252 // ...
1253 }
1254
1255 /**
1256 * Transfer an element from the queue to the stack.
1257 * @param e the element
1258 * @param l the lock
1259 * @return the element
1260 */
1261 private static E transfer(E e, boolean lono, E tTransferer)
1262 {
1263 // ...
1264 }
1265
1266 /**
1267 * Transfer an element from the stack to the queue.
1268 * @param e the element
1269 * @param s the stack
1270 * @param l the lock
1271 * @return the element
1272 */
1273 private static E transfer(E e, boolean lono, E tTransferer)
1274 {
1275 // ...
1276 }
1277
1278 /**
1279 * Transfer an element from the queue to the stack.
1280 * @param e the element
1281 * @param l the lock
1282 * @return the element
1283 */
1284 private static E transfer(E e, boolean lono, E tTransferer)
1285 {
1286 // ...
1287 }
1288
1289 /**
1290 * Transfer an element from the stack to the queue.
1291 * @param e the element
1292 * @param s the stack
1293 * @param l the lock
1294 * @return the element
1295 */
1296 private static E transfer(E e, boolean lono, E tTransferer)
1297 {
1298 // ...
1299 }
1300
1301 /**
1302 * Transfer an element from the queue to the stack.
1303 * @param e the element
1304 * @param l the lock
1305 * @return the element
1306 */
1307 private static E transfer(E e, boolean lono, E tTransferer)
1308 {
1309 // ...
1310 }
1311
1312 /**
1313 * Transfer an element from the stack to the queue.
1314 * @param e the element
1315 * @param s the stack
1316 * @param l the lock
1317 * @return the element
1318 */
1319 private static E transfer(E e, boolean lono, E tTransferer)
1320 {
1321 // ...
1322 }
1323
1324 /**
1325 * Transfer an element from the queue to the stack.
1326 * @param e the element
1327 * @param l the lock
1328 * @return the element
1329 */
1330 private static E transfer(E e, boolean lono, E tTransferer)
1331 {
1332 // ...
1333 }
1334
1335 /**
1336 * Transfer an element from the stack to the queue.
1337 * @param e the element
1338 * @param s the stack
1339 * @param l the lock
1340 * @return the element
1341 */
1342 private static E transfer(E e, boolean lono, E tTransferer)
1343 {
1344 // ...
1345 }
1346
1347 /**
1348 * Transfer an element from the queue to the stack.
1349 * @param e the element
1350 * @param l the lock
1351 * @return the element
1352 */
1353 private static E transfer(E e, boolean lono, E tTransferer)
1354 {
1355 // ...
1356 }
1357
1358 /**
1359 * Transfer an element from the stack to the queue.
1360 * @param e the element
1361 * @param s the stack
1362 * @param l the lock
1363 * @return the element
1364 */
1365 private static E transfer(E e, boolean lono, E tTransferer)
1366 {
1367 // ...
1368 }
1369
1370 /**
1371 * Transfer an element from the queue to the stack.
1372 * @param e the element
1373 * @param l the lock
1374 * @return the element
1375 */
1376 private static E transfer(E e, boolean lono, E tTransferer)
1377 {
1378 // ...
1379 }
1380
1381 /**
1382 * Transfer an element from the stack to the queue.
1383 * @param e the element
1384 * @param s the stack
1385 * @param l the lock
1386 * @return the element
1387 */
1388 private static E transfer(E e, boolean lono, E tTransferer)
1389 {
1390 // ...
1391 }
1392
1393 /**
1394 * Transfer an element from the queue to the stack.
1395 * @param e the element
1396 * @param l the lock
1397 * @return the element
1398 */
1399 private static E transfer(E e, boolean lono, E tTransferer)
1400 {
1401 // ...
1402 }
1403
1404 /**
1405 * Transfer an element from the stack to the queue.
1406 * @param e the element
1407 * @param s the stack
1408 * @param l the lock
1409 * @return the element
1410 */
1411 private static E transfer(E e, boolean lono, E tTransferer)
1412 {
1413 // ...
1414 }
1415
1416 /**
1417 * Transfer an element from the queue to the stack.
1418 * @param e the element
1419 * @param l the lock
1420 * @return the element
1421 */
1422 private static E transfer(E e, boolean lono, E tTransferer)
1423 {
1424 // ...
1425 }
1426
1427 /**
1428 * Transfer an element from the stack to the queue.
1429 * @param e the element
1430 * @param s the stack
1431 * @param l the lock
1432 * @return the element
1433 */
1434 private static E transfer(E e, boolean lono, E tTransferer)
1435 {
1436 // ...
1437 }
1438
1439 /**
1440 * Transfer an element from the queue to the stack.
1441 * @param e the element
1442 * @param l the lock
1443 * @return the element
1444 */
1445 private static E transfer(E e, boolean lono, E tTransferer)
1446 {
1447 // ...
1448 }
1449
1450 /**
1451 * Transfer an element from the stack to the queue.
1452 * @param e the element
1453 * @param s the stack
1454 * @param l the lock
1455 * @return the element
1456 */
1457 private static E transfer(E e, boolean lono, E tTransferer)
1458 {
1459 // ...
1460 }
1461
1462 /**
1463 * Transfer an element from the queue to the stack.
1464 * @param e the element
1465 * @param l the lock
1466 * @return the element
1467 */
1468 private static E transfer(E e, boolean lono, E tTransferer)
1469 {
1470 // ...
1471 }
1472
1473 /**
1474 * Transfer an element from the stack to the queue.
1475 * @param e the element
1476 * @param s the stack
1477 * @param l the lock
1478 * @return the element
1479 */
1480 private static E transfer(E e, boolean lono, E tTransferer)
1481 {
1482 // ...
1483 }
1484
1485 /**
1486 * Transfer an element from the queue to the stack.
1487 * @param e the element
1488 * @param l the lock
1489 * @return the element
1490 */
1491 private static E transfer(E e, boolean lono, E tTransferer)
1492 {
1493 // ...
1494 }
1495
1496 /**
1497 * Transfer an element from the stack to the queue.
1498 * @param e the element
1499 * @param s the stack
1500 * @param l the lock
1501 * @return the element
1502 */
1503 private static E transfer(E e, boolean lono, E tTransferer)
1504 {
1505 // ...
1506 }
1507
1508 /**
1509 * Transfer an element from the queue to the stack.
1510 * @param e the element
1511 * @param l the lock
1512 * @return the element
1513 */
1514 private static E transfer(E e, boolean lono, E tTransferer)
1515 {
1516 // ...
1517 }
1518
1519 /**
1520 * Transfer an element from the stack to the queue.
1521 * @param e the element
1522 * @param s the stack
1523 * @param l the lock
1524 * @return the element
1525 */
1526 private static E transfer(E e, boolean lono, E tTransferer)
1527 {
1528 // ...
1529 }
1530
1531 /**
1532 * Transfer an element from the queue to the stack.
1533 * @param e the element
1534 * @param l the lock
1535 * @return the element
1536 */
1537 private static E transfer(E e, boolean lono, E tTransferer)
1538 {
1539 // ...
1540 }
1541
1542 /**
1543 * Transfer an element from the stack to the queue.
1544 * @param e the element
1545 * @param s the stack
1546 * @param l the lock
1547 * @return the element
1548 */
1549 private static E transfer(E e, boolean lono, E tTransferer)
1550 {
1551 // ...
1552 }
1553
1554 /**
1555 * Transfer an element from the queue to the stack.
1556 * @param e the element
1557 * @param l the lock
1558 * @return the element
1559 */
1560 private static E transfer(E e, boolean lono, E tTransferer)
1561 {
1562 // ...
1563 }
1564
1565 /**
1566 * Transfer an element from the stack to the queue.
1567 * @param e the element
1568 * @param s the stack
1569 * @param l the lock
1570 * @return the element
1571 */
1572 private static E transfer(E e, boolean lono, E tTransferer)
1573 {
1574 // ...
1575 }
1576
1577 /**
1578 * Transfer an element from the queue to the stack.
1579 * @param e the element
1580 * @param l the lock
1581 * @return the element
1582 */
1583 private static E transfer(E e, boolean lono, E tTransferer)
1584 {
1585 // ...
1586 }
1587
1588 /**
1589 * Transfer an element from the stack to the queue.
1590 * @param e the element
1591 * @param s the stack
1592 * @param l the lock
1593 * @return the element
1594 */
1595 private static E transfer(E e, boolean lono, E tTransferer)
1596 {
1597 // ...
1598 }
1599
1600 /**
1601 * Transfer an element from the queue to the stack.
1602 * @param e the element
1603 * @param l the lock
1604 * @return the element
1605 */
1606 private static E transfer(E e, boolean lono, E tTransferer)
1607 {
1608 // ...
1609 }
1610
1611 /**
1612 * Transfer an element from the stack to the queue.
1613 * @param e the element
1614 * @param s the stack
1615 * @param l the lock
1616 * @return the element
1617 */
1618 private static E transfer(E e, boolean lono, E tTransferer)
1619 {
1620 // ...
1621 }
1622
1623 /**
1624 * Transfer an element from the queue to the stack.
1625 * @param e the element
1626 * @param l the lock
1627 * @return the element
1628 */
1629 private static E transfer(E e, boolean lono, E tTransferer)
1630 {
1631 // ...
1632 }
1633
1634 /**
1635 * Transfer an element from the stack to the queue.
1636 * @param e the element
1637 * @param s the stack
1638 * @param l the lock
1639 * @return the element
1640 */
1641 private static E transfer(E e, boolean lono, E tTransferer)
1642 {
1643 // ...
1644 }
1645
1646 /**
1647 * Transfer an element from the queue to the stack.
1648 * @param e the element
1649 * @param l the lock
1650 * @return the element
1651 */
1652 private static E transfer(E e, boolean lono, E tTransferer)
1653 {
1654 // ...
1655 }
1656
1657 /**
1658 * Transfer an element from the stack to the queue.
1659 * @param e the element
1660 * @param s the stack
1661 * @param l the lock
1662 * @return the element
1663 */
1664 private static E transfer(E e, boolean lono, E tTransferer)
1665 {
1666 // ...
1667 }
1668
1669 /**
1670 * Transfer an element from the queue to the stack.
1671 * @param e the element
1672 * @param l the lock
1673 * @return the element
1674 */
1675 private static E transfer(E e, boolean lono, E tTransferer)
1676 {
1677 // ...
1678 }
1679
1680 /**
1681 * Transfer an element from the stack to the queue.
1682 * @param e the element
1683 * @param s the stack
1684 * @param l the lock
1685 * @return the element
1686 */
1687 private static E transfer(E e, boolean lono, E tTransferer)
1688 {
1689 // ...
1690 }
1691
1692 /**
1693 * Transfer an element from the queue to the stack.
1694 * @param e the element
1695 * @param l the lock
1696 * @return the element
1697 */
1698 private static E transfer(E e, boolean lono, E tTransferer)
1699 {
1700 // ...
1701 }
1702
1703 /**
1704 * Transfer an element from the stack to the queue.
1705 * @param e the element
1706 * @param s the stack
1707 * @param l the lock
1708 * @return the element
1709 */
1710 private static E transfer(E e, boolean lono, E tTransferer)
1711 {
1712 // ...
1713 }
1714
1715 /**
1716 * Transfer an element from the queue to the stack.
1717 * @param e the element
1718 * @param l the lock
1719 * @return the element
1720 */
1721 private static E transfer(E e, boolean lono, E tTransferer)
1722 {
1723 // ...
1724 }
1725
1726 /**
1727 * Transfer an element from the stack to the queue.
1728 * @param e the element
1729 * @param s the stack
1730 * @param l the lock
1731 * @return the element
1732 */
1733 private static E transfer(E e, boolean lono, E tTransferer)
1734 {
1735 // ...
1736 }
1737
1738 /**
1739 * Transfer an element from the queue to the stack.
1740 * @param e the element
1741 * @param l the lock
1742 * @return the element
1743 */
1744 private static E transfer(E e, boolean lono, E tTransferer)
1745 {
1746 // ...
1747 }
1748
1749 /**
1750 * Transfer an element from the stack to the queue.
1751 * @param e the element
1752 * @param s the stack
1753 * @param l the lock
1754 * @return the element
1755 */
1756 private static E transfer(E e, boolean lono, E tTransferer)
1757 {
1758 // ...
1759 }
1760
1761 /**
1762 * Transfer an element from the queue to the stack.
1763 * @param e the element
1764 * @param l the lock
1765 * @return the element
1766 */
1767 private static E transfer(E e, boolean lono, E tTransferer)
1768 {
1769 // ...
1770 }
1771
1772 /**
1773 * Transfer an element from the stack to the queue.
1774 * @param e the element
1775 * @param s the stack
1776 * @param l the lock
1777 * @return the element
1778 */
1779 private static E transfer(E e, boolean lono, E tTransferer)
1780 {
1781 // ...
1782 }
1783
1784 /**
1785 * Transfer an element from the queue to the stack.
1786 * @param e the element
1787 * @param l the lock
1788 * @return the element
1789 */
1790 private static E transfer(E e, boolean lono, E tTransferer)
1791 {
1792 // ...
1793 }
1794
1795 /**
1796 * Transfer an element from the stack to the queue.
1797 * @param e the element
1798 * @param s the stack
1799 * @param l the lock
1800 * @return the element
1801 */
1802 private static E transfer(E e, boolean lono, E tTransferer)
1803 {
1804 // ...
1805 }
1806
1807 /**
1808 * Transfer an element from the queue to the stack.
1809 * @param e the element
1810 * @param l the lock
1811 * @return the element
1812 */
1813 private static E transfer(E e, boolean lono, E tTransferer)
1814 {
1815 // ...
1816 }
1817
1818 /**
1819 * Transfer an element from the stack to the queue.
1820 * @param e the element
1821 * @param s the stack
1822 * @param l the lock
1823 * @return the element
1824 */
1825 private static E transfer(E e, boolean lono, E tTransferer)
1826 {
1827 // ...
1828 }
1829
1830 /**
1831 * Transfer an element from the queue to the stack.
1832 * @param e the element
1833 * @param l the lock
1834 * @return the element
1835 */
1836 private static E transfer(E e, boolean lono, E tTransferer)
1837 {
1838 // ...
1839 }
1840
1841 /**
1842 * Transfer an element from the stack to the queue.
1843 * @param e the element
1844 * @param s the stack
1845 * @param l the lock
1846 * @return the element
1847 */
1848 private static E transfer(E e, boolean lono, E tTransferer)
1849 {
1850 // ...
1851 }
1852
1853 /**
1854 * Transfer an element from the queue to the stack.
1855 * @param e the element
1856 * @param l the lock
1857 * @return the element
1858 */
1859 private static E transfer(E e, boolean lono, E tTransferer)
1860 {
1861 // ...
1862 }
1863
1864 /**
1865 * Transfer an element from the stack to the queue.
1866 * @param e the element
1867 * @param s the stack
1868 * @param l the lock
1869 * @return the element
1870 */
1871 private static E transfer(E e, boolean lono, E tTransferer)
1872 {
1873 // ...
1874 }
1875
1876 /**
1877 * Transfer an element from the queue to the stack.
1878 * @param e the element
1879 * @param l the lock
1880 * @return the element
1881 */
1882 private static E transfer(E e, boolean lono, E tTransferer)
1883 {
1884 // ...
1885 }
1886
1887 /**
1888 * Transfer an element from the stack to the queue.
1889 * @param e the element
1890 * @param s the stack
1891 * @param l the lock
1892 * @return the element
1893 */
1894 private static E transfer(E e, boolean lono, E tTransferer)
1895 {
1896 // ...
1897 }
1898
1899 /**
1900 * Transfer an element from the queue to the stack.
1901 * @param e the element
1902 * @param l the lock
1903 * @return the element
1904 */
1905 private static E transfer(E e, boolean lono, E tTransferer)
1906 {
1907 // ...
1908 }
1909
1910 /**
1911 * Transfer an element from the stack to the queue.
1912 * @param e the element
1913 * @param s the stack
1914 * @param l the lock
1915 * @return the element
1916 */
1917 private static E transfer(E e, boolean lono, E tTransferer)
1918 {
1919 // ...
1920 }
1921
1922 /**
1923 * Transfer an element from the queue to the stack.
1924 * @param e the element
1925 * @param l the lock
1926 * @return the element
1927 */
1928 private static E transfer(E e, boolean lono, E tTransferer)
1929 {
1930 // ...
1931 }
1932
1933 /**
1934 * Transfer an element from the stack to the queue.
1935 * @param e the element
1936 * @param s the stack
1937 * @param l the lock
1938 * @return the element
1939 */
1940 private static E transfer(E e, boolean lono, E tTransferer)
1941 {
1942 // ...
1943 }
1944
1945 /**
1946 * Transfer an element from the queue to the stack.
1947 * @param e the element

目录

```
006      *
007      * @param o the element
008      * @return {@code false}
009      */
010  public boolean contains(Object o) { return false; }
013
014      /**
015      * Always returns {@code false}.
016      * A {@code SynchronousQueue} has no internal capacity.
017      *
018      * @param o the element to remove
019      * @return {@code false}
020      */
021  public boolean remove(Object o) { return false; }
024
025      和容量相关的方法都是默认实现
026      /**
027      * Returns {@code false} unless the given collection is empty.
028      * A {@code SynchronousQueue} has no internal capacity.
029      *
030      * @param c the collection
031      * @return {@code false} unless given collection is empty
032      */
033  public boolean containsAll(Collection<?> c) { return c.isEmpty(); }
035
036      /**
037      * Always returns {@code false}.
038      * A {@code SynchronousQueue} has no internal capacity.
039      *
040      * @param c the collection
041      * @return {@code false}
042      */
043  public boolean removeAll(Collection<?> c) { return false; }
```

果断更，

1.3 结构细节

请联系QQ/微信64260066

SynchronousQueue 底层结构和其它队列完全不同，有着独特的两种数据结构：队列和堆栈。

我们一起来看看下数据结构：

```
// 堆栈和队列共同的接口
// 负责执行 put or take
abstract static class Transferer<E> {
    // e 为空的，会直接返回特殊值，不为空会传递给消费者
    // timed 为 true，说明会有超时时间
    abstract E transfer(E e, boolean timed, long nanos);
}

// 堆栈 后入先出 非公平
// Scherer-Scott 算法
static final class TransferStack<E> extends Transferer<E> {
}

// 队列 先入先出 公平
static final class TransferQueue<E> extends Transferer<E> {
}

private transient volatile Transferer<E> transferer;

// 无参构造器默认为非公平的
public SynchronousQueue(boolean fair) {
    transferer = fair ? new TransferQueue<E>() : new TransferStack<E>();
}
```

从源码中我们可以得到几点：

目录

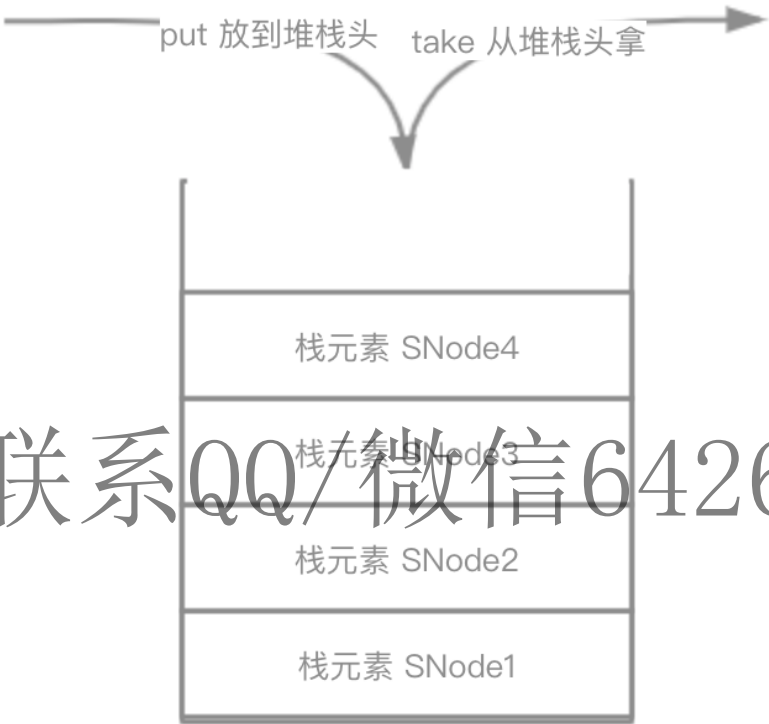
2. 在我们初始化的时候，是可以选择是使用堆栈还是队列的，如果你不选择，默认的就是堆栈，类注释中也说明了这一点，堆栈的效率比队列更高。

接下来我们来看下堆栈和队列的具体实现。

2 不公平的堆栈

2.1 堆栈的结构

首先我们来介绍下堆栈的整体结构，如下：



从上图中我们可以看到，我们有一个大的堆栈池，池的开口叫做堆栈头，put 的时候，就往堆栈池中放数据。take 的时候，就从堆栈池中拿数据，两者操作都是在堆栈头上操作数据，从图中可以看到，越靠近堆栈头，数据越新，所以每次 take 的时候，都会拿到堆栈头的最新数据，这就是我们说的后入先出，也就是不公平的。

图中 SNode 就是源码中栈元素的表示，我们看下源码：

```
static final class SNode {
    // 栈的下一个，就是被当前栈压在下面的栈元素
    volatile SNode next;
    // 节点匹配，用来判断阻塞栈元素能被唤醒的时机
    // 比如我们先执行 take，此时队列中没有数据，take 被阻塞了，栈元素为 SNode1
    // 当有 put 操作时，会把当前 put 的栈元素赋值给 SNode1 的 match 属性，并唤醒 take 操作
    // 当 take 被唤醒，发现 SNode1 的 match 属性有值时，就能拿到 put 进来的数据，从而返回
    volatile SNode match;
    // 栈元素的阻塞是通过线程阻塞来实现的，waiter 为阻塞的线程
    volatile Thread waiter;
    // 未投递的消息，或者未消费的消息
```

目录

2.2 入栈和出栈

入栈指的是使用 put 等方法，把数据放到堆栈池中，出栈指的是使用 take 等方法，把数据从堆栈池中拿出来，操作的对象都是堆栈头，虽然两者的一个是从堆栈头拿数据，一个是放数据，但底层实现的方法却是同一个，源码如下：

```
// transfer 方法思路比较复杂，因为 take 和 put 两个方法都揉在了一起
@SuppressWarnings("unchecked")
E transfer(E e, boolean timed, long nanos) {
    SNode s = null; // constructed/reused as needed
    // e 为空，说明是 take 方法，不为空是 put 方法
    int mode = (e == null) ? REQUEST : DATA;
    // 自旋
    for (;;) {
        // 拿出头节点，有几种情况
        // 1: 头节点为空，说明队列中还没有数据
        // 2: 头节点不为空，并且是 take 类型的，说明头节点线程正等着拿数据。
        // 3: 头节点不为空，并且是 put 类型的，说明头节点线程正等着放数据。
        SNode h = head;
        // 栈头为空，说明队列中还没有数据。
        // 栈头不为空，并且栈头的类型和本次操作一致，比如都是 put，那么就把
        // 本次 put 操作放到该栈头的前面即可，让本次 put 能够先执行
        if (h == null || h.mode == mode) { // empty or same-mode
            // 设置了超时时间，并且 e 进栈或者出栈要超时了，
            // 就会丢弃本次操作，返回 null 值。
            // 如果栈头此时被取消了，丢弃栈头，取下一个节点继续消费
            if (timed && nanos <= 0) { // can't wait
                // 栈头操作被取消
                if (h != null && h.isCancelled())
                    // 丢弃栈头，把栈头后一个元素作为栈头
                    casHead(h, h.next); // pop cancelled node
                // 栈头是空的，直接返回 null
                return null;
            }
            // 没有超时，直接把 e 作为新的栈头
        } else if (casHead(h, s = snode(s, e, h, mode))) {
            // e 等待出栈，一种是空队列 take，一种是 put
            SNode m = awaitFulfill(s, timed, nanos);
            if (m == s) { // wait was cancelled
                clean(s);
                return null;
            }
            // 本来 s 是栈头的，现在 s 不是栈头了，s 后面又来了一个数，把新的数据作为栈头
            if ((h = head) != null && h.next == s)
                casHead(h, s.next); // help s's fulfiller
            return (E) ((mode == REQUEST) ? m.item : s.item);
        }
        // 栈头正在等待其他线程 put 或 take
        // 比如栈头正在阻塞，并且是 put 类型，而此次操作正好是 take 类型，走此处
    } else if (!isFulfilling(h.mode)) { // try to fulfill
        // 栈头已经被取消，把下一个元素作为栈头
        if (h.isCancelled()) // already cancelled
            casHead(h, h.next); // pop and retry
        // snode 方法第三个参数 h 代表栈头，赋值给 s 的 next 属性
    } else if (casHead(h, s = snode(s, e, h, FULFILLING|mode))) {
        for (;;) { // loop until matched or waiters disappear
            // m 就是栈头，通过上面 snode 方法刚刚赋值
            SNode m = s.next; // m is s's match
            if (m == null) { // all waiters are gone
                casHead(s, null); // pop fulfill node
            }
        }
    }
}
```

果断更，请联系QQ/微信64260066

目录	<pre> SNode mn = m.next; // tryMatch 非常重要的方法，两个作用： // 1 唤醒被阻塞的栈头 m，2 把当前节点 s 赋值给 m 的 match 属性 // 这样栈头 m 被唤醒时，就能从 m.match 中得到本次操作 s // 其中 s.item 记录着本次的操作节点，也就是记录本次操作的数据 if (m.tryMatch(s)) { casHead(s, mn); // pop both s and m return (E) ((mode == REQUEST) ? m.item : s.item); } else // lost match s.casNext(m, mn); // help unlink } } } else { // help a fulfiller SNode m = h.next; // m is h's match if (m == null) // waiter is gone casHead(h, null); // pop fulfilling node else { SNode mn = m.next; if (m.tryMatch(h)) // help match casHead(h, mn); // pop both h and m else // lost match h.casNext(m, mn); // help unlink } } } }</pre>
----	---

从源码中密密麻麻的注释，我们就可以看出来此方法比较复杂，我们总结一下大概的操作思路：

1. 判断是 put 方法还是 take 方法；
2. 判断栈头数据是否为空，如果为空或者栈头的操作和本次操作一致，是的话走 3，否则走 5；
3. 判断操作有无设置超时时间，如果设置了超时时间并且已经超时，返回 null，否则走 4；
4. 如果栈头为空，把当前操作设置成栈头，或者栈头不为空，但栈头的操作和本次操作相同，也把当前操作设置成栈头，并看看其它线程能否满足自己，不能满足则阻塞自己。比如当前操作是 take，但队列中没有数据，则阻塞自己；
5. 如果栈头已经是阻塞住的，需要别人唤醒的，判断当前操作能否唤醒栈头，可以唤醒走 6，否则走 4；
6. 把自己当作一个节点，赋值到栈头的 match 属性上，并唤醒栈头节点；
7. 栈头被唤醒后，拿到 match 属性，就是把自己唤醒的节点的信息，返回。

在整个过程中，有一个节点阻塞的方法，实现原理如下：

<pre>SNode awaitFulfill(SNode s, boolean timed, long nanos) { // deadline 死亡时间，如果设置了超时时间的话，死亡时间等于当前时间 + 超时时间，否则就是 final long deadline = timed ? System.nanoTime() + nanos : 0L; Thread w = Thread.currentThread(); // 自旋的次数，如果设置了超时时间，会自旋 32 次，否则自旋 512 次。 // 比如本次操作是 take 操作，自选次数后，仍没有其他线程 put 数据进来 // 就会阻塞，有超时时间的，会阻塞固定的时间，否则一致阻塞下去 int spins = (shouldSpin(s) ? (timed ? maxTimedSpins : maxUntimedSpins) : 0); for (;;) { // 当前线程有无被打断，如果过了超时时间，当前线程就会被打断 if (w.isInterrupted()) s.tryCancel(); } }</pre>

目录	<pre> if (timed) { nanos = deadline - System.nanoTime(); // 超时了, 取消当前线程的等待操作 if (nanos <= 0L) { s.tryCancel(); continue; } } // 自选次数减少 1 if (spins > 0) spins = shouldSpin(s) ? (spins-1) : 0; // 把当前线程设置成 waiter, 主要是通过线程来完成阻塞和唤醒 else if (s.waiter == null) s.waiter = w; // establish waiter so can park next iter else if (!timed) // 通过 park 进行阻塞, 这个我们在锁章节中会说明 LockSupport.park(this); else if (nanos > spinForTimeoutThreshold) LockSupport.parkNanos(this, nanos); } }</pre>
----	--

从节点阻塞代码中，我们可以发现，其阻塞的策略，并不是一上来就阻塞住，而是在自旋一定次数后，仍然没有其它线程来满足自己的要求时，才会真正的阻塞住。

3 公平的队列

果断更，请联系QQ/微信64260066

首先我们来看一下队列中的每个元素的组成：

```
/** 队列头 */
transient volatile QNode head;
/** 队列尾 */
transient volatile QNode tail;

// 队列的元素
static final class QNode {
    // 当前元素的下一个元素
    volatile QNode next;
    // 当前元素的值, 如果当前元素被阻塞住了, 等其他线程来唤醒自己时, 其他线程
    // 会把自己 set 到 item 里面
    volatile Object item;    // CAS'ed to or from null
    // 可以阻塞住的当前线程
    volatile Thread waiter;  // to control park/unpark
    // true 是 put, false 是 take
    final boolean isData;
}
```

公平的队列主要使用的是 TransferQueue 内部类的 transfer 方法，我们一起来看下源码：

```
E transfer(E e, boolean timed, long nanos) {

    QNode s = null; // constructed/reused as needed
    // true 是 put, false 是 get
    boolean isData = (e != null);

    for (;;) {
        // 队列头和尾的临时变量,队列是空的时候, t=h
```

目录

```
// 虽然这种 continue 非常耗cpu, 但感觉不会碰到这种情况
// 因为 tail 和 head 在 TransferQueue 初始化的时候, 就已经被赋值空节点了
if (t == null || h == null)
    continue;
// 首尾节点相同, 说明是空队列
// 或者尾节点的操作和当前节点操作一致
if (h == t || t.isData == isData) {
    QNode tn = t.next;
    // 当 t 不是 tail 时, 说明 tail 已经被修改过了
    // 因为 tail 没有被修改的情况下, t 和 tail 必然相等
    // 因为前面刚刚执行赋值操作: t = tail
    if (t != tail)
        continue;
    // 队尾后面的值还不为空, t 还不是队尾, 直接把 tn 赋值给 t, 这是一步加强校验。
    if (tn != null) {
        advanceTail(t, tn);
        continue;
    }
    //超时直接返回 null
    if (timed && nanos <= 0) // can't wait
        return null;
    //构造node节点
    if (s == null)
        s = new QNode(e, isData);
    //如果把 e 放到队尾失败, 继续递归放进去
    if (!t.casNext(null, s)) // failed to link in
        continue;

    advanceTail(t, s); // swing tail and wait
    // 阻塞住自己
    Object x = awaitFull(s, timed, nanos);
    if (x == s) { // wait was cancelled
        clean(t, s);
        return null;
    }

    if (!s.isOffList()) { // not already unlinked
        advanceHead(t, s); // unlink if head
        if (x != null) // and forget fields
            s.item = s;
            s.waiter = null;
        }
        return (x != null) ? (E)x : e;
    // 队列不为空, 并且当前操作和队尾不一致
    // 也就是说当前操作是队尾是对应的操作
    // 比如说队尾是因为 take 被阻塞的, 那么当前操作必然是 put
    } else { // complementary-mode
        // 如果是第一次执行, 此处的 m 代表就是 tail
        // 也就是这行代码体现出队列的公平, 每次操作时, 从头开始按照顺序进行操作
        QNode m = h.next; // node to fulfill
        if (t != tail || m == null || h != head)
            continue; // inconsistent read

        Object x = m.item;
        if (isData == (x != null)) // m already fulfilled
            x == m // m cancelled
            // m 代表栈头
            // 这里把当前的操作值赋值给阻塞住的 m 的 item 属性
            // 这样 m 被释放时, 就可得到此次操作的值
            !m.casItem(x, e) { // lost CAS
                advanceHead(h, m); // dequeue and retry
                continue;
            }
        }
```


目录	<pre>// 释放队头阻塞节点 LockSupport.unpark(m.waiter); return (x != null) ? (E)x : e; } } }</pre>
----	---

源码比较复杂，我们需要搞清楚的是，线程被阻塞住后，当前线程是如何把自己的数据传给阻塞线程的。为了方便说明，我们假设线程 1 往队列中 take 数据，被阻塞住了，变成阻塞线程 A，然后线程 2 开始往队列中 put 数据 B，大致的流程是这样的：

1. 线程 1 从队列中拿数据，发现队列中没有数据，于是被阻塞，成为 A；
2. 线程 2 往队尾 put 数据，会从队尾往前找到第一个被阻塞的节点，假设此时能找到的就是节点 A，然后线程 B 把将 put 的数据放到节点 A 的 item 属性里面，并唤醒线程 1；
3. 线程 1 被唤醒后，就能从 A.item 里面拿到线程 2 put 的数据了，线程 1 成功返回。

从这个过程中，我们能看出公平主要体现在，每次 put 数据的时候，都 put 到队尾上，而每次拿数据时，并不是直接从堆头拿数据，而是从队尾往前寻找第一个被阻塞的线程，这样就会按照顺序释放被阻塞的线程。

4 总结

SynchronousQueue 源码比较复杂，建议大家进行源码的 debug 来学习源码，为大家准备了调试类 SynchronousQueueDemo，大家可以下载源码自己调试一下，这样学起来应该会更加轻松一点。

果断更，请联系QQ/微信64260066

精选留言 6

欢迎在这里发表留言，作者筛选后可公开显示

所相虚妄

老师能不能说一下，这个数据都应用场景是啥？

👍 0 回复

6天前

慕粉1150563265

是有容量大小的吧，只不过他的同步机制，促使线程，需要同步等待。等待的线程会保存到队列或者堆栈中

👍 0 回复

2019-11-29

文贺 回复 慕粉1150563265

没有容量大小，size() 方法返回的永远是 0

回复

2019-11-30 13:07:09

目录	为了angular耻辱上线
	怎么感觉总结transfer方法的那一段写错了...
	👍 0 回复
	2019-11-29
	大LOVE辉
	老师,好像和源码不太一样,源码比较麻烦...
	👍 0 回复
	2019-11-27
	慕斯卡0137221
	总结之前的话得重新梳理下吧，尤其是线程和节点两个之间的关系，感觉有点乱啊
	👍 1 回复
	2019-11-22
	licly
	老师，transfer方法中，else if (!isFulfilling(h.mode)) --> else if (casHead(h, s=snode(s, e, h, FULFILLING mode))) --> for死循环中，if (m == null)这种情况什么时候会发生呀，感觉走到这一步，应该不是空的，有点懵，SynchronousQueue这个类太难理解了
	👍 0 回复
	2019-10-18

果断更，请联系QQ/微信6426006