

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	

29 押宝线程源码面试题

更新时间：2019-11-06 09:54:20



“如果不想在世界上虚度一生，那就要学习一辈子。”
——高尔基

果断更，请联系QQ/微信642600651

关于线程方面的面试题，大部分都是概念题，我们需要大概的清楚这些概念，和面试官达成共识即可，本章我们一起来看下这些面试题，对前两章的学习进行巩固。

1 面试题

1.1 创建子线程时，子线程是得不到父线程的 ThreadLocal，有什么办法可以解决这个问题？

答：这道题主要考察线程的属性和创建过程，可以这么回答。

可以使用 InheritableThreadLocal 来代替 ThreadLocal，ThreadLocal 和 InheritableThreadLocal 都是线程的属性，所以可以做到线程之间的数据隔离，在多线程环境下我们经常使用，但在有子线程被创建的情况下，父线程 ThreadLocal 是无法传递给子线程的，但 InheritableThreadLocal 可以，主要是在线程创建的过程中，会把

InheritableThreadLocal 里面的所有值传递给子线程，具体代码如下：

```
// 当父线程的 inheritableThreadLocals 的值不为空时
// 会把 inheritableThreadLocals 里面的值全部传递给子线程
if (parent.inheritableThreadLocals != null)
    this.inheritableThreadLocals =
        ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
```

1.2 线程创建有几种实现方式？

无返回值的线程有两种写法，第一种是继承 Thread，可以这么与：

```
class MyThread extends Thread{
    @Override
    public void run() {
        log.info(Thread.currentThread().getName());
    }
}

@Test
public void extendThreadInit(){
    new MyThread().start();
}
```

第二种是实现 Runnable 接口，并作为 Thread 构造器的入参，代码如下：

```
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        log.info("{} begin run",Thread.currentThread().getName());
    }
});
// 开一个子线程去执行
thread.start();
```

这两种都会开一个子线程去执行任务，并且是没有返回值的，如果需要子线程有返回值，需要使用 Callable 接口，但 Callable 接口是无法直接作为 Thread 构造器的入参的，必须结合 FutureTask 一起使用，可以这样写代码：

```
@Test
public void testThreadByCallable() throws ExecutionException, InterruptedException {
    FutureTask futureTask = new FutureTask(new Callable<String> () {
        @Override
        public String call() throws Exception {
            Thread.sleep(3000);
            String result = "我是子线程"+Thread.currentThread().getName();
            log.info("子线程正在运行: {}",Thread.currentThread().getName());
            return result;
        }
    });
    new Thread(futureTask).start();
    log.info("返回的结果是 {}",futureTask.get());
}
```

把 FutureTask 作为 Thread 的入参就可以了，FutureTask 组合了 Callable，使我们可以使用 Callable，并且 FutureTask 实现了 Runnable 接口，使其可以作为 Thread 构造器的入参，还有 FutureTask 实现了 Future，使其对任务有一定的管理功能。

1.3 子线程 1 去等待子线程 2 执行完成之后才能执行，如何去实现？

答：这里考察的就是 Thread.join 方法，我们可以这么做：

```
@Test
public void testJoin2() throws Exception {
    Thread thread2 = new Thread(new Runnable() {
        @Override
        public void run() {
```

目录	<pre> } catch (InterruptedException e) { e.printStackTrace(); } log.info("我是子线程 2, 执行完成"); } }); Thread thread1 = new Thread(new Runnable() { @Override public void run() { log.info("我是子线程 1, 开始运行"); try { log.info("我是子线程 1, 我在等待子线程 2"); // 这里是代码关键 thread2.join(); log.info("我是子线程 1, 子线程 2 执行完成, 我继续执行"); } catch (InterruptedException e) { e.printStackTrace(); } log.info("我是子线程 1, 执行完成"); } }); thread1.start(); thread2.start(); Thread.sleep(100000); }</pre>
----	---

子线程 1 需要等待子线程 2，只需要子线程 1 运行的时候，调用子线程 2 的 join 方法即可，这样线程 1 执行到 join 代码时，就会等待线程 2 执行完成之后，才会继续执行。

1.4 守护线程和非守护线程的区别？如果我想在项目启动的时候收集代码信息，请问是守护线程好，还是非守护线程好，为什么？

答：两者的主要区别是，在 JVM 退出时，JVM 是不会管守护线程的，只会管非守护线程，如果非守护线程还有在运行的，JVM 就不会退出，如果没有非守护线程了，但还有守护线程的，JVM 直接退出。

如果需要在项目启动的时候收集代码信息，就需要看收集工作是否重要了，如果不太重要，又很耗时，就应该选择守护线程，这样不会妨碍 JVM 的退出，如果收集工作非常重要的话，那么就需要非守护进程，这样即使启动时发生未知异常，JVM 也会等到代码收集信息线程结束后才会退出，不会影响收集工作。

1.5 线程 start 和 run 之间的区别。

答：调用 Thread.start 方法会开一个新的线程，run 方法不会。

1.6 Thread、Runnable、Callable 三者之间的区别。

答：Thread 实现了 Runnable，本身就是 Runnable，但同时负责线程创建、线程状态变更等操作。

Runnable 是无返回值任务接口，Callable 是有返回值任务接口，如果任务需要跑起来，必须需要 Thread 的支持才行，Runnable 和 Callable 只是任务的定义，具体执行还需要靠 Thread。

1.7 线程池 submit 有两个方法，方法一可接受 Runnable，方法二可接受 Callable，但两个方法底层的逻辑却是同一套，这是如何适配的。

目录

Runnable 和 Callable 是通过 FutureTask 进行统一的，FutureTask 有个属性是 Callable，同时也实现了 Runnable 接口，两者的统一转化是在 FutureTask 的构造器里实现的，FutureTask 的最终目标是把 Runnable 和 Callable 都转化成 Callable，Runnable 转化成 Callable 是通过 RunnableAdapter 适配器进行实现的。

线程池的 submit 底层的逻辑只认 FutureTask，不认 Runnable 和 Callable 的差异，所以只要都转化成 FutureTask，底层实现都会是同一套。

具体 Runnable 转化成 Callable 的代码和逻辑可以参考上一章，有非常详细的描述。

1.8 Callable 能否丢给 Thread 去执行？

答：可以的，可以新建 Callable，并作为 FutureTask 的构造器入参，然后把 FutureTask 丢给 Thread 去执行即可。

1.9 FutureTask 有什么作用(谈谈对 FutureTask 的理解)。

答：作用如下：

1. 组合了 Callable，实现了 Runnable，把 Callable 和 Runnable 串联了起来。
2. 统一了有参任务和无参任务两种定义方式，方便了使用。
3. 实现了 Future 的所有方法，对任务有一定的管理功能，比如说拿到任务执行结果，取消任务，打断任务等等。

1.10 聊聊对 FutureTask 的 get、cancel 方法的理解

答：get 方法主要作用是得到 Callable 异步任务执行的结果，无参 get 会一直等待任务执行完成之后才返回，有参 get 方法可以设定固定的时间，在设定的时间内，如果任务还没有执行成功，直接返回异常，在实际工作中，建议多多使用 get 有参方法，少用 get 无参方法，防止任务执行过慢时，多数线程都在等待，造成线程耗尽的问题。

cancel 方法主要用来取消任务，如果任务还没有执行，是可以取消的，如果任务已经在执行过程中了，你可以选择不取消，或者直接打断执行中的任务。

两个方法具体的执行步骤和原理见上一章节源码解析。

1.11 Thread.yield 方法在工作中有什么用？

答：yield 方法表示当前线程放弃 cpu，重新参与到 cpu 的竞争中去，再次竞争时，自己有可能得到 cpu 资源，也有可能得不到，这样做的好处是防止当前线程一直霸占 cpu。

我们在工作中可能会写一些 while 自旋的代码，如果我们一直 while 自旋，不采取任何手段，我们会发现 cpu 一直被当前 while 循环占用，如果能预见 while 自旋时间很长，我们会设置一定的判断条件，让当前线程陷入阻塞，如果能预见 while 自旋时间很短，我们通常会使用 Thread.yield 方法，使当前自旋线程让步，不一直霸占 cpu，比如这样：

```
boolean stop = false;
while (!stop){
    // dosomething
    Thread.yield();
}
```

答：相同点：

1. 两者都让线程进入到 TIMED_WAITING 状态，并且可以设置等待的时间。

不同点：

1. wait 是 Object 类的方法，sleep 是 Thread 类的方法。
2. sleep 不会释放锁，沉睡的时候，其它线程是无法获得锁的，但 wait 会释放锁。

1.13 写一个简单的死锁 demo

```
// 共享变量 1
private static final Object share1 = new Object();
// 共享变量 2
private static final Object share2 = new Object();
@Test
public void testDeadLock() throws InterruptedException {
    // 初始化线程 1，线程 1 需要在锁定 share1 共享资源的情况下再锁定 share2
    Thread thread1 = new Thread() -> {
        synchronized (share1){
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (share2){
                log.info("{} is run",Thread.currentThread().getName());
            }
        }
    };

    // 初始化线程 2，线程 2 需要在锁定 share2 共享资源的情况下再锁定 share1
    Thread thread2 = new Thread() -> {
        synchronized (share2){
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (share1){
                log.info("{} is run",Thread.currentThread().getName());
            }
        }
    };

    // 当线程 1、2 启动后，都在等待对方锁定的资源，但都得不到，造成死锁
    thread1.start();
    thread2.start();
    Thread.sleep(1000000000);
}
```

2 总结

线程章节算是中等难度，我们需要清楚线程的概念，线程如何初始化，线程的状态变更等等问题，这些知识点都是线程池、锁的基础，学好线程后，再学习线程池和锁就会轻松很多。

目录

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示



目前暂无任何讨论

千学不如一看，千看不如一练

果断更，请联系QQ/微信64260066