代码改成多线程，竟有 9 大坑

涛歌依旧 2022-10-07 10:48 发表于广东

以下文章来源于苏三说技术 ， 作者苏三呀

苏三说技术

作者曾浪迹几家大厂，掘金优秀创作者，CSDN万粉博主。

大家好，我是涛哥。

假期最后一天，好好休息下，明天又要开工了。今天来聊聊编程中的多线程问题。

前言

很多时候，我们为了提升接口的性能，会把之前 单线程同步 执行的代码，改成 多线程异步 执行。

比如：查询用户信息接口，需要返回用户基本信息、积分信息、成长值信息，而用户、积分和成长值，需要调用不同的接口获取数据。

如果查询用户信息接口， 同步调用 三个接口获取数据，会非常耗时。

这就非常有必要把三个接口调用，改成 异步调用 ，最后 汇总结果 。

再比如：注册用户接口，该接口主要包含：写用户表，分配权限，配置用户导航页，发通知消息等功能。

该用户注册接口包含的业务逻辑比较多，如果在接口中同步执行这些代码，该接口响应时间会非常慢。

这时就需要把业务逻辑梳理一下，划分： 核心逻辑 和 非核心逻辑 。这个例子中的核心逻辑是：写用户表和分配权限，非核心逻辑是：配置用户导航页和发通知消息。

显然 核心逻辑 必须在接口中 同步执行 ，而 非核心逻辑 可以 多线程异步 执行。

等等。

需要使用多线程的业务场景太多了，使用多线程异步执行的好处不言而喻。

但我要说的是，如果多线程没有使用好，它也会给我们带来很多意想不到的问题，不信往后继续看。

今天跟大家一起聊聊，代码改成多线程调用之后，带来的9大问题。

1.获取不到返回值

如果你通过直接继承 `Thread` 类，或者实现 `Runnable` 接口的方式去创建 `线程` 。

那么，恭喜你，你将没法获取该线程方法的返回值。

使用线程的场景有两种：

1. 不需要关注线程方法的返回值。
2. 需要关注线程方法的返回值。

大部分业务场景是不需要关注线程方法返回值的，但如果我们有些业务需要关注线程方法的返回值该怎么处理呢？

查询用户信息接口，需要返回用户基本信息、积分信息、成长值信息，而用户、积分和成长值，需要调用不同的接口获取数据。

如下图所示：

在Java8之前可以通过实现 `Callable` 接口，获取线程返回结果。

Java8以后通过 `CompleteFuture` 类实现该功能。我们这里以CompleteFuture为例：

```
public UserInfo getUserInfo(Long id) throws InterruptedException, ExecutionException {
    final UserInfo userInfo = new UserInfo();
    CompletableFuture userFuture = CompletableFuture.supplyAsync(() -> {
        getRemoteUserAndFill(id, userInfo);
        return Boolean.TRUE;
    }, executor);
```

```
    CompletableFuture bonusFuture = CompletableFuture.supplyAsync(() -> {
        getRemoteBonusAndFill(id, userInfo);
        return Boolean.TRUE;
    }, executor);

    CompletableFuture growthFuture = CompletableFuture.supplyAsync(() -> {
        getRemoteGrowthAndFill(id, userInfo);
        return Boolean.TRUE;
    }, executor);
    CompletableFuture.allOf(userFuture, bonusFuture, growthFuture).join();

    userFuture.get();
    bonusFuture.get();
    growthFuture.get();

    return userInfo;
}
```

> 温馨提醒一下，这两种方式别忘了使用线程池。示例中我用到了executor，表示自定义的线程池，为了防止高并发场景下，出现线程过多的问题。

此外， `Fork/join` 框架也提供了执行任务并返回结果的能力。

2.数据丢失

我们还是以注册用户接口为例，该接口主要包含：写用户表，分配权限，配置用户导航页，发通知消息等功能。

其中：写用户表和分配权限功能，需要在一个事务中同步执行。而剩余的配置用户导航页和发通知消息功能，使用多线程异步执行。

表面上看起来没问题。

但如果前面的写用户表和分配权限功能成功了，用户注册接口就直接返回成功了。

但如果后面异步执行的配置用户导航页，或发通知消息功能失败了，怎么办？

如 下 图 所 示 ：

该接口前面明明已经提示用户成功了，但结果后面又有一部分功能在多线程异步执行中失败了。

这时该如何处理呢？

没错，你可以做 失败重试 。

但如果重试了一定的次数，还是没有成功，这条请求数据该如何处理呢？如果不做任何处理，该数据是不是就丢掉了？

为了防止数据丢失，可以用如下方案：

1. 使用mq异步处理。在分配权限之后，发送一条mq消息，到mq服务器，然后在mq的消费者中使用多线程，去配置用户导航页和发通知消息。如果mq消费者中处理失败了，可以自己重试。
2. 使用job异步处理。在分配权限之后，往任务表中写一条数据。然后有个job定时扫描该表，然后配置用户导航页和发通知消息。如果job处理某条数据失败了，可以在表中记录一个重试次数，然后不断重试。但该方案有个缺点，就是实时性可能不太高。

3.顺序问题

如果你使用了多线程，就必须接受一个非常现实的问题，即 顺序问题 。

假如之前代码的执行顺序是：a,b,c，改成多线程执行之后，代码的执行顺序可能变成了：a,c,b。（这个跟cpu调度算法有关）

例如：

```
public static void main(String[] args) {
    Thread thread1 = new Thread(() -> System.out.println("a"));
    Thread thread2 = new Thread(() -> System.out.println("b"));
    Thread thread3 = new Thread(() -> System.out.println("c"));

    thread1.start();
    thread2.start();
    thread3.start();
}
```

执行结果：

```
a
c
b
```

那么，来自灵魂的一问：如何保证线程的顺序呢？

即线程启动的顺序是：a,b,c，执行的顺序也是：a,b,c。

如下图所示：

3.1 join

`Thread` 类的 `join` 方法它会让主线程等待子线程运行结束后，才能继续运行。

列如：

```
public static void main(String[] args) throws InterruptedException {
    Thread thread1 = new Thread(() -> System.out.println("a"));
    Thread thread2 = new Thread(() -> System.out.println("b"));
    Thread thread3 = new Thread(() -> System.out.println("c"));

    thread1.start();
    thread1.join();
    thread2.start();
    thread2.join();
    thread3.start();
}
```

执行结果永远都是：

```
a
b
c
```

3.2 newSingleThreadExecutor

我们可以使用JDK自带的 `Excutors` 类的 `newSingleThreadExecutor` 方法，创建一个 `单线程的 线程池` 。

例如：

```
public static void main(String[] args)  {
    ExecutorService executorService = Executors.newSingleThreadExecutor();

    Thread thread1 = new Thread(() -> System.out.println("a"));
    Thread thread2 = new Thread(() -> System.out.println("b"));
    Thread thread3 = new Thread(() -> System.out.println("c"));

    executorService.submit(thread1);
    executorService.submit(thread2);
    executorService.submit(thread3);

    executorService.shutdown();
}
```

执行结果永远都是：

```
a
b
c
```

使用 `Excutors` 类的 `newSingleThreadExecutor` 方法创建的单线程的线程池，使用了 `LinkedBlockingQueue` 作为队列，而此队列按 `FIFO` （先进先出）排序元素。

添加到队列的顺序是a,b,c，则执行的顺序也是a,b,c。

3.3 CountDownLatch

`CountDownLatch` 是一个同步工具类，它允许一个或多个线程一直等待，直到其他线程执行完后再执行。

例如：

```
public class ThreadTest {

    public static void main(String[] args) throws InterruptedException {
        CountDownLatch latch1 = new CountDownLatch(0);
        CountDownLatch latch2 = new CountDownLatch(1);
        CountDownLatch latch3 = new CountDownLatch(1);

        Thread thread1 = new Thread(new TestRunnable(latch1, latch2, "a"));
        Thread thread2 = new Thread(new TestRunnable(latch2, latch3, "b"));
        Thread thread3 = new Thread(new TestRunnable(latch3, latch3, "c"));

        thread1.start();
        thread2.start();
        thread3.start();
    }
}

class TestRunnable implements Runnable {

    private CountDownLatch latch1;
    private CountDownLatch latch2;
    private String message;

    TestRunnable(CountDownLatch latch1, CountDownLatch latch2, String message) {
        this.latch1 = latch1;
        this.latch2 = latch2;
        this.message = message;
    }

    @Override
    public void run() {
        try {
            latch1.await();
            System.out.println(message);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        latch2.countDown();
    }
}
```

```
}
```

执行结果永远都是：

```
a
b
c
```

此外，使用 `CompletableFuture` 的 `thenRun` 方法，也能多线程的执行顺序，在这里就不一一介绍了。

4.线程安全问题

既然使用了线程，伴随而来的还会有线程安全问题。

假如现在有这样一个需求：用多线程执行查询方法，然后把执行结果添加到一个list集合中。

代码如下：

```
List<User> list = Lists.newArrayList();
 dataList.stream()
     .map(data -> CompletableFuture
          .supplyAsync(() -> query(list, data), asyncExecutor)
         ));
CompletableFuture.allOf(futureArray).join();
```

使用 `CompletableFuture` 异步多线程执行query方法：

```
public void query(List<User> list, UserEntity condition) {
   User user = queryByCondition(condition);
   if(Objects.isNull(user)) {
      return;
   }
   list.add(user);
   UserExtend userExtend = queryByOther(condition);
   if(Objects.nonNull(userExtend)) {
      user.setExtend(userExtend.getInfo());
   }
}
```

在query方法中，将获取的查询结果添加到list集合中。

结果list会出现线程安全问题，有时候会少数据，当然也不一定是必现的。

这是因为 ArrayList 是 非线程安全 的，没有使用 synchronized 等关键字修饰。

如何解决这个问题呢？

答：使用 CopyOnWriteArrayList 集合，代替普通的 ArrayList 集合，CopyOnWriteArrayList是一个线程安全的机会。

只需一行小小的改动即可：

```
List<User> list Lists.newCopyOnWriteArrayList();
```

> 温馨的提醒一下，这里创建集合的方式，用了google的collect包。

5.ThreadLocal获取数据异常

我们都知道 JDK 为了解决线程安全问题，提供了一种用空间换时间的新思路：ThreadLocal 。

它的核心思想是：共享变量在每个 线程 都有一个 副本 ，每个线程操作的都是自己的副本，对另外的线程没有影响。

例如：

```
@Service
public class ThreadLocalService {
    private static final ThreadLocal<Integer> threadLocal = new ThreadLocal<>();

    public void add() {
        threadLocal.set(1);
        doSamething();
        Integer integer = threadLocal.get();
```

```
    }
}
```

ThreadLocal在普通中线程中，的确能够获取正确的数据。

但在真实的业务场景中，一般很少用 单独的线程 ，绝大多数，都是用的 线程池 。

那么，在线程池中如何获取 ThreadLocal 对象生成的数据呢？

如果直接使用普通ThreadLocal，显然是获取不到正确数据的。

我们先试试 InheritableThreadLocal ，具体代码如下：

```
private static void fun1() {
    InheritableThreadLocal<Integer> threadLocal = new InheritableThreadLocal<>();
    threadLocal.set(6);
    System.out.println("父线程获取数据：" + threadLocal.get());

    ExecutorService executorService = Executors.newSingleThreadExecutor();

    threadLocal.set(6);
    executorService.submit(() -> {
        System.out.println("第一次从线程池中获取数据：" + threadLocal.get());
    });

    threadLocal.set(7);
    executorService.submit(() -> {
        System.out.println("第二次从线程池中获取数据：" + threadLocal.get());
    });
}
```

执行结果：

```
父线程获取数据：6
第一次从线程池中获取数据：6
第二次从线程池中获取数据：6
```

由于这个例子中使用了单例线程池，固定线程数是1。

第一次submit任务的时候，该线程池会自动创建一个线程。因为使用了InheritableThreadLocal，所以创建线程时，会调用它的init方法，将父线程中的inheritableThreadLocals数据复制到子线程中。所以我们看到，在主线程中将数据设置成6，第一次从线程池中获取了正确的数据6。

之后，在主线程中又将数据改成7，但在第二次从线程池中获取数据却依然是6。

因为第二次submit任务的时候，线程池中已经有一个线程了，就直接拿过来复用，不会再重新创建线程了。所以不会再调用线程的init方法，所以第二次其实没有获取到最新的数据7，还是获取的老数据6。

那么，这该怎么办呢？

答：使用 `TransmittableThreadLocal` ，它并非JDK自带的类，而是阿里巴巴开源jar包中的类。

可以通过如下pom文件引入该jar包：

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>transmittable-thread-local</artifactId>
    <version>2.11.0</version>
    <scope>compile</scope>
</dependency>
```

代码调整如下：

```
private static void fun2() throws Exception {
    TransmittableThreadLocal<Integer> threadLocal = new TransmittableThreadLocal<>();
    threadLocal.set(6);
    System.out.println("父线程获取数据：" + threadLocal.get());

    ExecutorService ttlExecutorService = TtlExecutors.getTtlExecutorService(Executors.newFixe

    threadLocal.set(6);
    ttlExecutorService.submit(() -> {
        System.out.println("第一次从线程池中获取数据：" + threadLocal.get());
    });
```

```
        threadLocal.set(7);
        ttlExecutorService.submit(() -> {
            System.out.println("第二次从线程池中获取数据：" + threadLocal.get());
        });

    }
```

执行结果：

```
父线程获取数据：6
第一次从线程池中获取数据：6
第二次从线程池中获取数据：7
```

我们看到，使用了TransmittableThreadLocal之后，第二次从线程中也能正确获取最新的数据7了。

nice。

如果你仔细观察这个例子，你可能会发现，代码中除了使用 `TransmittableThreadLocal` 类之外，还使用了 `TtlExecutors.getTtlExecutorService` 方法，去创建 `ExecutorService` 对象。

这是非常重要的地方，如果没有这一步， `TransmittableThreadLocal` 在线程池中共享数据将不会起作用。

创建 `ExecutorService` 对象，底层的submit方法会 `TtlRunnable` 或 `TtlCallable` 对象。

以TtlRunnable类为例，它实现了 `Runnable` 接口，同时还实现了它的run方法：

```
    public void run() {
        Map<TransmittableThreadLocal<?>, Object> copied = (Map)this.copiedRef.get();
        if (copied != null && (!this.releaseTtlValueReferenceAfterRun || this.copiedRef.compareAr
            Map backup = TransmittableThreadLocal.backupAndSetToCopied(copied);

            try {
                this.runnable.run();
            } finally {
                TransmittableThreadLocal.restoreBackup(backup);
            }
```

```
        } else {
            throw new IllegalStateException("TTL value reference is released after run!");
        }
    }
```

这段代码的主要逻辑如下：

1. 把当时的ThreadLocal做个备份，然后将父类的ThreadLocal拷贝过来。
2. 执行真正的run方法，可以获取到父类最新的ThreadLocal数据。
3. 从备份的数据中，恢复当时的ThreadLocal数据。

如果你想进一步了解ThreadLocal的工作原理，可以看看我的另一篇文章《**ThreadLocal夺命11连问**》

6.OOM问题

众所周知，使用多线程可以提升代码执行效率，但也不是绝对的。

对于一些耗时的操作，使用多线程，确实可以提升代码执行效率。

但线程不是创建越多越好，如果线程创建多了，也可能会导致 `OOM` 异常。

例如：

```
Caused by:
java.lang.OutOfMemoryError: unable to create new native thread
```

在 `JVM` 中创建一个线程，默认需要占用 `1M` 的内存空间。

如果创建了过多的线程，必然会导致内存空间不足，从而出现OOM异常。

除此之外，如果使用线程池的话，特别是使用固定大小线程池，即使用 `Executors.newFixedThreadPool` 方法创建的线程池。

该线程池的 `核心线程数` 和 `最大线程数` 是一样的，是一个固定值，而存放消息的队列是 `LinkedBlockingQueue` 。

该队列的最大容量是 `Integer.MAX_VALUE` ，也就是说如果使用固定大小线程池，存放了太多的任务，有可能也会导致OOM异常。

```
java.lang.OutOfMemeryError:Java heap space
```

7.CPU使用率飙高

不知道你有没有做过excel数据导入功能，需要将一批excel的数据导入到系统中。

每条数据都有些业务逻辑，如果单线程导入所有的数据，导入效率会非常低。

于是改成了多线程导入。

如果excel中有大量的数据，很可能会出现CPU使用率飙高的问题。

我们都知道，如果代码出现死循环，cpu使用率会飚的很多高。因为代码一直在某个线程中循环，没法切换到其他线程，cpu一直被占用着，所以会导致cpu使用率一直高居不下。

而多线程导入大量的数据，虽说没有死循环代码，但由于多个线程一直在不停的处理数据，导致占用了cpu很长的时间。

也会出现cpu使用率很高的问题。

那么，如何解决这个问题呢？

答：使用 `Thread.sleep` 休眠一下。

在线程中处理完一条数据，休眠10毫秒。

当然CPU使用率飙高的原因很多，多线程处理数据和死循环只是其中两种，还有比如：频繁GC、正则匹配、频繁序列化和反序列化等。

后面我会写一篇介绍CPU使用率飙高的原因的专题文章，感兴趣的小伙伴，可以关注一下我后续的文章。

8.事务问题

在实际项目开发中，多线程的使用场景还是挺多的。如果spring事务用在多线程场景中，会有问题吗？

例如：

```
@Slf4j
@Service
public class UserService {

    @Autowired
    private UserMapper userMapper;
    @Autowired
    private RoleService roleService;

    @Transactional
    public void add(UserModel userModel) throws Exception {
        userMapper.insertUser(userModel);
        new Thread(() -> {
            roleService.doOtherThing();
        }).start();
    }
}

@Service
public class RoleService {

    @Transactional
    public void doOtherThing() {
        System.out.println("保存role表数据");
    }
}
```

从上面的例子中，我们可以看到 事务方法 add中，调用了事务方法doOtherThing，但是 事务方法 doOtherThing是在另外一个 线程 中调用的。

这样会导致两个方法不在同一个线程中，获取到的 数据库连接 不一样，从而是两个不同的 事务 。如果想doOtherThing方法中抛了异常，add方法也回滚是不可能的。

如果看过spring事务源码的朋友，可能会知道spring的事务是通过数据库连接来实现的。当前线程中保存了一个map，key是 数据源 ，value是 数据库连接 。

```
private static final ThreadLocal<Map<Object, Object>> resources =

  new NamedThreadLocal<>("Transactional resources");
```

我们说的 同一个事务 ，其实是指 同一个数据库连接 ，只有拥有同一个数据库连接才能同时 提交 和 回滚 。如果在不同的 线程 ，拿到的 数据库连接 肯定是不一样的，所以是不同的事务。

> 所以不要在事务中开启另外的线程，去处理业务逻辑，这样会导致事务失效。

9.导致服务挂掉

使用多线程会导致服务挂掉，这不是危言耸听，而是确有其事。

假设现在有这样一种业务场景：在mq的消费者中需要调用订单查询接口，查到数据之后，写入业务表中。

本来是没啥问题的。

突然有一天，mq生产者跑了一个批量数据处理的job，导致mq服务器上堆积了大量的消息。

此时，mq消费者的处理速度，远远跟不上mq消息的生产速度，导致的结果是出现了大量的消息堆积，对用户有很大的影响。

为了解决这个问题，mq消费者改成 多线程 处理，直接使用了 线程池 ，并且 最大线程数 配置成了20。

这样调整之后，消息堆积问题确实得到了解决。

但带来了另外一个更严重的问题：订单查询接口并发量太大了，有点扛不住压力，导致部分节点的服务直接挂掉。

为了解决问题，不得不临时加服务节点。

> 在mq的消费者中使用多线程，调用接口时，一定要评估好接口能够承受的最大访问量，防止因为压力过大，而导致服务挂掉的问题。

今天先这样，咱们下次见。

喜欢此内容的人还喜欢

低代码都做了什么？怎么实现 Low-Code?

几何带你学前端

追求性能极致：Redis6.0的多线程模型

架构与思维

一个线程安全的泛型支持map库

专家极客圈