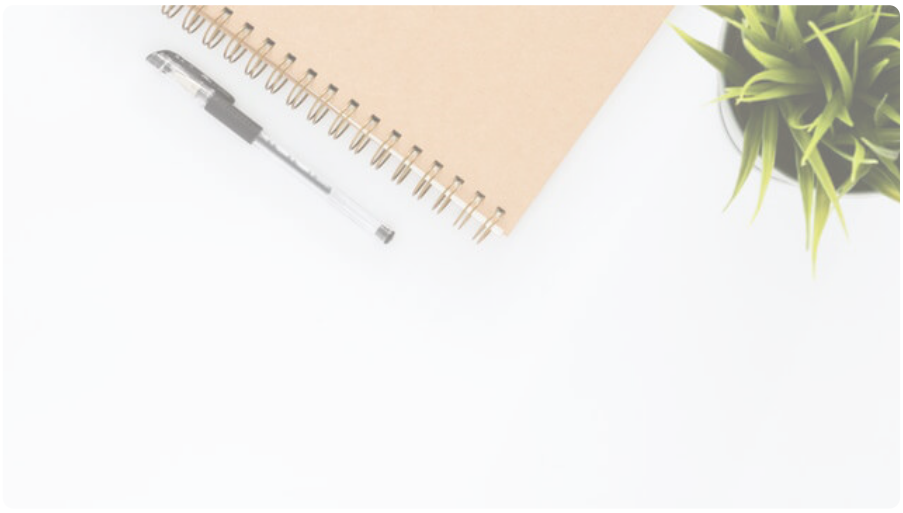
44 场景实战：ThreadLocal 在上下文传值场景下的实践

更新时间：2019-11-28 09:55:13



> 生活的理想，就是为了理想的生活。 ——张闻天

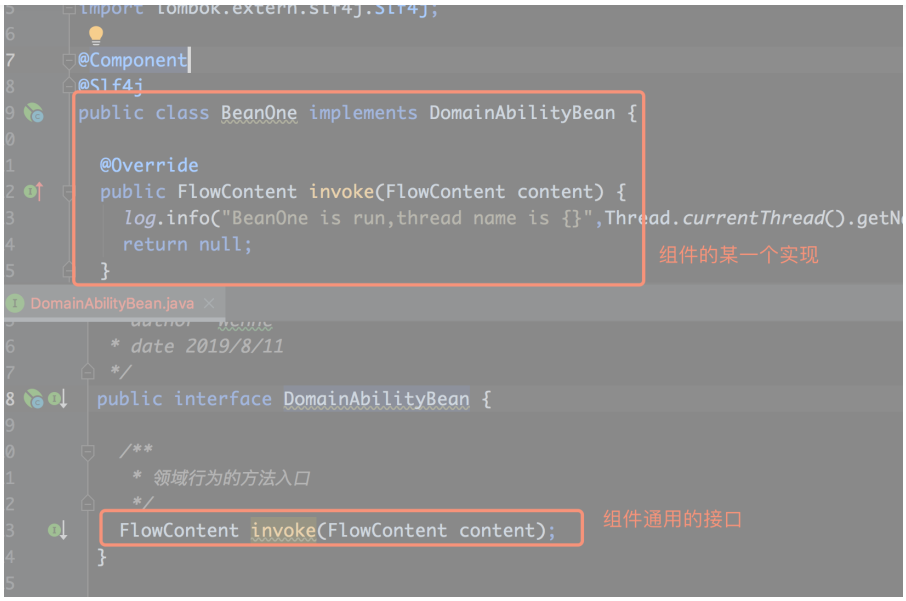
开篇语

我们在 《打动面试官：线程池流程编排中的运用实战》一文中将流程引擎简单地完善了一下，本文在其基础上继续进行改造，建议同学可以先看看 GitHub 上的代码，或者看看之前的文章。

1 回顾

流程引擎编排的对象，我们称为组件（就是 SpringBean），之前我们给组件定义了通用的接口，组件实现时就实现这个接口，代码如下：

目录



我们定义了 DomainAbilityBean 接口，入参和出参都是 FlowContent，FlowContent 我们称为上下文。

2 ThreadLocal 实现

上下文传参除了 FlowContent 实现外，ThreadLocal 也是可以实现的，我们来演示一下：

2.1 定义 ThreadLocal 上下文工具类

首先我们使用 ThreadLocal 定义了上下文工具类，并且定义了 put、get 方法，方便使用，代码如下：

```
public class ContextCache implements Serializable {

 private static final long serialVersionUID = 2136539028591849277L;

 // 使用 ThreadLocal 缓存上下文信息
 public static final ThreadLocal<Map<String,String>> CACHE = new ThreadLocal<>();

 /**
  * 放数据
  * @param sourceKey
  */
 public static final void putAttribute(String sourceKey,String value){
  Map<String,String> cacheMap = CACHE.get();
  if(null == cacheMap){
   cacheMap = new HashMap<>();
  }
  cacheMap.put(sourceKey,value);
  CACHE.set(cacheMap);
 }

 /**
  * 拿数据
  * @param sourceKey
  */
 public static final String getAttribute(String sourceKey){
  Map<String,String> cacheMap = CACHE.get();
  if(null == cacheMap){
```

目录

```
    }

}
```

如果你想往 ThreadLocal 放数据，调用 ContextCache.putAttribute 方法，如果想从 ThreadLocal 拿数据，调用 ContextCache.getAttribute 方法即可。

我们写了两个组件，一个组件放数据，一个组件拿数据，如下：

BeanThree.java

```
import lombok.extern.slf4j.Slf4j;

@Slf4j
@Component
public class BeanThree implements DomainAbilityBean {

    @Override
    public void invoke() {
        ContextCache.putAttribute( sourceKey: "key1",  value: "value1");
        log.info("put key1,value1 to contextCache success");
    }
}
```

把 key1、value1 放到 ThreadLocal 中

BeanFive.java

```
@Slf4j
@Component
public class BeanFive implements DomainAbilityBean {

    @Override
    public void invoke() {
        String value = ContextCache.getAttribute( sourceKey: "key1");
        log.info("get key1,value is {}", value);
    }
}
```

把 key1 从 ThreadLocal 中拿出来

我们把两个 SpringBean 注册到流程注册中心中，让其按照先执行 BeanThree 再执行 BeanFive 的顺序进行执行，运行 DemoApplication 类的 main 方法进行执行，执行结果如下：

```
vletContainer : Tomcat started on port(s): 8080 (http)
              : Started DemoApplication in 6.705 seconds (JVM running for 8.433)
..BeanThree   : put key1,value1 to contextCache success
..BeanFive    : get key1,value is value1
```

从打印的日志可以看到，在 Spring 容器管理的 SpringBean 中，ThreadLocal 也是可以储存中间缓存值的。

3 开启子线程

我们做一个实验，我们在 BeanFive 中开启子线程，然后再从 ThreadLocal 中拿值，看看能否拿到值，BeanFive 的代码修改成如下：

目录

```
@Override
public void invoke() {
    new Thread(() -> {
        log.info("子线程开启成功，子线程名为:{}", Thread.currentThread().getName());
        String value = ContextCache.getAttribute( sourceKey: "key1");
        log.info("在子线程中 get key1,value is {}", value);
    }).start();
}
}
```

修改成在子线程中，从 ThreadLocal 中获取值

我们再来运行一下，打印的日志如下：

```
r.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type [class org.spring
r.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class org.springframewor
r.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto handler of type [class org.sp
tationMBeanExporter        : Registering beans for JMX exposure on startup
atEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
cation                     : Started DemoApplication in 6.476 seconds (JVM running for 7.313)
w.threadlocal.BeanThree    : put key1,value1 to contextCache success
w.threadlocal.BeanFive     : 子线程开启成功，子线程名为:Thread-4
w.threadlocal.BeanFive     : 在子线程中 get key1 value is null
```

从打印的日志中，我们发现在子线程中从 ThreadLocal 取值时，并没有取得值，这个原因主要是我们之前说的，线程在创建的时候，并不会把父线程的 ThreadLocal 中的值拷贝给子线程的 ThreadLocal，解决方案就是把 ThreadLocal 修改成 InheritableThreadLocal，代码修改如下：

```
public class ContextCache implements Serializable {
    private static final long serialVersionUID = 2138339028518432771L;
    // 使用 ThreadLocal 存放上下文值
    public static final ThreadLocal<Map<String,String>> CACHE = new InheritableThreadLocal<>();
```

从 ThreadLocal 修改成这个类

我们再次运行，结果如下：

```
        : Started DemoApplication in 8.058 seconds (JVM running for 9.46)
hree    : put key1,value1 to contextCache success
ive     : 子线程开启成功，子线程名为:Thread-4
ive     : 在子线程中 get key1,value is value1
```

从运行结果看，我们成功的在子线程中拿到值。

4 线程池 + ThreadLocal

如果是拿数据的 springBean 是丢给线程池执行的，我们能够成功的从 ThreadLocal 中拿到数据么？

首先我们在放数据的 springBean 中，把放的值修改成随机的，接着拿数据的 SpringBean 修改成异步执行，代码修改如下：

目录

```
@Override
public void invoke() {
  int i = new Random().nextInt( bound: 1000);
  ContextCache.putAttribute( sourceKey: "key1",  value: "value"+i);
  log.info("put key1,value{}",i);
}
```

往 ThreadLocal 中放数据

BeanFive.java

```
@Slf4j
@AsyncComponent
@Component
public class BeanFive implements DomainAbilityBean {

  @Override
  public void invoke() {
    String value = ContextCache.getAttribute( sourceKey: "key1");
    try {
      Thread.sleep( millis: 5000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    log.info("get 线程名称为{}, value is {}", Thread.currentThread().getName(),value);
    log.info("------------------");
  }
}
```

加上异步执行的注解，任务会被丢到线程池中去执行

沉睡一段时间，让线程消费更慢，可以让大多数任务可以在阻塞队列中去排队

为了能快速看到效果，我们把线程池的 coreSize 和 maxSize 全部修改成 3，并让任务沉睡一段时间，这样三个线程肯定消费不完任务，大量任务都会到队列中去排队，我们修改一下测试脚本，如下：

```
public static void main(String[] args) throws InterruptedException {
  SpringApplication.run(DemoApplication.class);
  for (int i = 0; i < 10; i++) {
    ApplicationContextHelper.getBean(FlowStart.class)
        .start( flowName: "flow2", new FlowContent());
  }
}
```


我们期望的结果：

1. 线程池中执行的 BeanFive 可以成功从 ThreadLocal 中拿到数据；
2. 能够从 ThreadLocal 拿到正确的数据，比如 BeanThree 刚放进 key1，value5，那么期望在 BeanFive 中根据 key1 能拿出 value5，而不是其它值。

我们运行一下，结果如下：

```
2019-10-17 13:10:21.528 [           main]     : put key1,value11
2019-10-17 13:10:21.529 [           main]     : put key1,value500
2019-10-17 13:10:21.530 [           main]     : put key1,value548
2019-10-17 13:10:21.530 [           main]     : put key1,value580
2019-10-17 13:10:21.530 [           main]     : put key1,value131
2019-10-17 13:10:21.530 [           main]     : put key1,value32
2019-10-17 13:10:21.531 [           main]     : put key1,value947
2019-10-17 13:10:21.531 [           main]     : put key1,value73
2019-10-17 13:10:21.531 [           main]     : put key1,value773
2019-10-17 13:10:21.531 [           main]     : put key1,value379
2019-10-17 13:10:26.533 [pool-1-thread-1]     : get 线程名称为pool-1-thread-1, value is value500
2019-10-17 13:10:26.533 [pool-1-thread-3]     : get 线程名称为pool-1-thread-3, value is value548
2019-10-17 13:10:26.533 [pool-1-thread-2]     : get 线程名称为pool-1-thread-2, value is value548
2019-10-17 13:10:31.538 [pool-1-thread-3]     : get 线程名称为pool-1-thread-3, value is value379
2019-10-17 13:10:31.538 [pool-1-thread-2]     : get 线程名称为pool-1-thread-2, value is value379
2019-10-17 13:10:31.538 [pool-1-thread-1]     : get 线程名称为pool-1-thread-1, value is value379
2019-10-17 13:10:36.539 [pool-1-thread-3]     : get 线程名称为pool-1-thread-3, value is value379
2019-10-17 13:10:36.539 [pool-1-thread-2]     : get 线程名称为pool-1-thread-2, value is value379
2019-10-17 13:10:36.539 [pool-1-thread-1]     : get 线程名称为pool-1-thread-1, value is value379
2019-10-17 13:10:41.544 [pool-1-thread-3]     : get 线程名称为pool-1-thread-3, value is value379
```

从结果中可以看到，并没有符合我们的预期，我们往 ThreadLocal 中 put 进很多值，但最后拿出来的值却很多都是 value379，都为最后 put 到 ThreadLocal 中的值。

这个原因主要是 ThreadLocal 存储的 HashMap 的引用都是同一个，main 主线程可以修改 HashMap 中的值，子线程从 ThreadLocal 中拿值时，也是从 HashMap 中拿值，从而导致不

目录

为了证明是这个原因，我们在从 ThreadLocal 放、拿值的地方，把 HashMap 的内存地址都打印出来，改动代码如下：

```java
/**
 * 放数据
 * @param sourceKey
 */
public static final void putAttribute(String sourceKey,String value){
  Map<String,String> cacheMap = CACHE.get();
  if(null == cacheMap){
    cacheMap = new HashMap<>();
  }
  log.info("putAttribute HashMap 内存地址 is {}",System.identityHashCode(cacheMap));
  cacheMap.put(sourceKey,value);
  CACHE.set(cacheMap);
}

/**
 * 拿数据
 * @param sourceKey
 */
public static final String getAttribute(String sourceKey){
  Map<String,String> cacheMap = CACHE.get();
  log.info("getAttribute HashMap 内存地址 is {}",System.identityHashCode(cacheMap));
  if(null == cacheMap){
    return null;
  }
  return cacheMap.get(sourceKey);
}
```

拿数据和放数据的地方都打印出 HashMap 的内存地址

我们再次运行测试代码，运行的结果如下：

```
2019-10-17 13:23:25.552 [pool-1-thread-1]    : getAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:25.552 [           main]    : putAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:25.552 [           main]     : put key1,value392
2019-10-17 13:23:25.552 [pool-1-thread-2]    : getAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:25.552 [           main]    : putAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:25.553 [           main]     : put key1,value231
2019-10-17 13:23:25.554 [pool-1-thread-3]    : getAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:25.554 [           main]    : putAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:25.554 [           main]     : put key1,value521
2019-10-17 13:23:25.555 [           main]    : putAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:25.557 [           main]     : put key1,value943
2019-10-17 13:23:25.557 [           main]    : putAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:25.557 [           main]     : put key1,value185
2019-10-17 13:23:25.557 [           main]    : putAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:25.558 [           main]     : put key1,value184
2019-10-17 13:23:25.558 [           main]    : putAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:25.558 [           main]     : put key1,value926
2019-10-17 13:23:25.558 [           main]    : putAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:25.558 [           main]     : put key1,value965
2019-10-17 13:23:25.558 [           main]    : putAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:25.558 [           main]     : put key1,value964
2019-10-17 13:23:30.556 [pool-1-thread-2]     : get 线程名称为pool-1-thread-2, value is value392
2019-10-17 13:23:30.556 [pool-1-thread-1]     : get 线程名称为pool-1-thread-1, value is value99
2019-10-17 13:23:30.556 [pool-1-thread-3]     : get 线程名称为pool-1-thread-3, value is value231
2019-10-17 13:23:30.556 [pool-1-thread-2]    : getAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:30.556 [pool-1-thread-3]    : getAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:30.556 [pool-1-thread-1]    : getAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:35.559 [pool-1-thread-3]     : get 线程名称为pool-1-thread-3, value is value964
2019-10-17 13:23:35.559 [pool-1-thread-1]     : get 线程名称为pool-1-thread-1, value is value964
2019-10-17 13:23:35.559 [pool-1-thread-3]    : getAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:35.559 [pool-1-thread-1]    : getAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:35.559 [pool-1-thread-2]     : get 线程名称为pool-1-thread-2, value is value964
2019-10-17 13:23:35.560 [pool-1-thread-2]    : getAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:40.560 [pool-1-thread-1]     : get 线程名称为pool-1-thread-1, value is value964
2019-10-17 13:23:40.561 [pool-1-thread-2]     : get 线程名称为pool-1-thread-2, value is value964
2019-10-17 13:23:40.561 [pool-1-thread-2]    : getAttribute HashMap 内存地址 is 1486954672
2019-10-17 13:23:40.560 [pool-1-thread-3]     : get 线程名称为pool-1-thread-3, value is value964
2019-10-17 13:23:45.562 [pool-1-thread-2]     : get 线程名称为pool-1-thread-2, value is value964
```


从测试结果中可以看到，不管是主线程还是子线程和 ThreadLocal 进行交互时，HashMap 都是同一个，也就是说 ThreadLocal 中保存的 HashMap 是共享的，这就导致了线程安全的问题，子线程读取到的值就会混乱掉。

5 解决方案

针对这个问题，我们提出了一种解决方案，在把任务提交到线程池时，我们进行 HashMap 的拷贝，这样子线程的 HashMap 和 main 线程的 HashMap 就不同了，可以解决上面的问题。

目录

```java
@Slf4j
@Getter        自定义任务
public class AsyncRunnable implements Runnable {

  private Runnable runnable;
  private HashMap hashMap = new HashMap(); 临时存储数据

  public AsyncRunnable(Runnable runnable) {
    this.runnable = runnable;
    copy(runnable); 把提交任务的 ThreadLocal 拷贝到 HashMap 中
  }

  @Override
  public void run() {  代码执行到 run 方法中时，已经到新的线程中了，再把 hashMap
    try {              的值拷贝到新的线程上下文中
      ContextCache.putAllAttribute(hashMap);
      if (null != runnable) {
        runnable.run();
      }
    } catch (Exception e) {
      log.error("[AsyncRunnable-run] has error", e);
      throw new RuntimeException(e);
    } finally {
      hashMap = new HashMap();
      ContextCache.clean();
    }
  }

  private void copy(Runnable runnable) { 拷贝
    try {
      hashMap.putAll(ContextCache.getMap());
    } catch (Exception e) {
      log.error("[AsyncRunnable-copy] has error, runnable is {}"
          runnable.toString(), e);
      throw new RuntimeException(e);
    }
  }
```


运行结果如下:

```
2019-10-17 14:00:26.698[           main] : put key1,value479
2019-10-17 14:00:26.699[           main] : put key1,value8
2019-10-17 14:00:26.699[           main] : put key1,value539
2019-10-17 14:00:26.700[           main] : put key1,value398
2019-10-17 14:00:26.700[           main] : put key1,value20
2019-10-17 14:00:26.700[           main] : put key1,value388
2019-10-17 14:00:26.700[           main] : put key1,value83
2019-10-17 14:00:26.701[           main] : put key1,value10
2019-10-17 14:00:26.701[           main] : put key1,value302
2019-10-17 14:00:26.701[           main] : put key1,value723
2019-10-17 14:00:26.701[           main] : put key1,value901
2019-10-17 14:00:26.701[           main] : put key1,value251
2019-10-17 14:00:26.701[           main] : put key1,value285
2019-10-17 14:00:26.701[           main] : put key1,value750
2019-10-17 14:00:26.702[           main] : put key1,value771
2019-10-17 14:00:26.702[           main] : put key1,value714
2019-10-17 14:00:26.702[           main] : put key1,value86
2019-10-17 14:00:26.702[           main] : put key1,value736
2019-10-17 14:00:26.702[           main] : put key1,value106
2019-10-17 14:00:26.702[           main] : put key1,value866
2019-10-17 14:00:31.704[pool-1-thread-3]  : get 线程名称为pool-1-thread-3, value is value539
2019-10-17 14:00:31.704[pool-1-thread-1]  : get 线程名称为pool-1-thread-1, value is value479
2019-10-17 14:00:31.704[pool-1-thread-2]  : get 线程名称为pool-1-thread-2, value is value8
2019-10-17 14:00:36.707[pool-1-thread-3]  : get 线程名称为pool-1-thread-3, value is value398
2019-10-17 14:00:36.707[pool-1-thread-1]  : get 线程名称为pool-1-thread-1, value is value388
2019-10-17 14:00:36.707[pool-1-thread-2]  : get 线程名称为pool-1-thread-2, value is value20
2019-10-17 14:00:41.709[pool-1-thread-1]  : get 线程名称为pool-1-thread-1, value is value10
2019-10-17 14:00:41.709[pool-1-thread-2]  : get 线程名称为pool-1-thread-2, value is value302
2019-10-17 14:00:41.709[pool-1-thread-3]  : get 线程名称为pool-1-thread-3, value is value83
2019-10-17 14:00:46.714[pool-1-thread-2]  : get 线程名称为pool-1-thread-2, value is value723
2019-10-17 14:00:46.714[pool-1-thread-1]  : get 线程名称为pool-1-thread-1, value is value901
2019-10-17 14:00:46.714[pool-1-thread-3]  : get 线程名称为pool-1-thread-3, value is value251
2019-10-17 14:00:51.719[pool-1-thread-3]  : get 线程名称为pool-1-thread-3, value is value750
2019-10-17 14:00:51.719[pool-1-thread-1]  : get 线程名称为pool-1-thread-1, value is value771
2019-10-17 14:00:51.719[pool-1-thread-2]  : get 线程名称为pool-1-thread-2, value is value285
2019-10-17 14:00:56.722[pool-1-thread-1]  : get 线程名称为pool-1-thread-1, value is value714
2019-10-17 14:00:56.722[pool-1-thread-3]  : get 线程名称为pool-1-thread-3, value is value86
2019-10-17 14:00:56.722[pool-1-thread-2]  : get 线程名称为pool-1-thread-2, value is value736
2019-10-17 14:01:01.727[pool-1-thread-1]  : get 线程名称为pool-1-thread-1, value is value106
```

6 总结

本文通过 ThreadLocal 来改造流程引擎中的上下文传递，希望能够加深大家对 ThreadLocal 的认识和使用技巧，有兴趣的同学可以把我们的代码下载下来，跑跑看。

← 43 ThreadLocal 源码解析 45 Socket 源码及面试题 →

精选留言 1

欢迎在这里发表留言，作者筛选后可公开显示

听见你说

打动面试官的文章地址有嘛。

0 回复 4天前

千学不如一看，千看不如一练