28 | 如何使用设计模式优化并发编程？

2019-07-27 刘超

你好，我是刘超。

在我们使用多线程编程时，很多时候需要根据业务场景设计一套业务功能。其实，在多线程编程中，本身就存在很多成熟的功能设计模式，学好它们，用好它们，那就是如虎添翼了。今天我就带你了解几种并发编程中常用的设计模式。

线程上下文设计模式

线程上下文是指贯穿线程整个生命周期的对象中的一些全局信息。例如，我们比较熟悉的Spring中的ApplicationContext就是一个关于上下文的类，它在整个系统的生命周期中保存了配置信息、用户信息以及注册的bean等上下文信息。

这样的解释可能有点抽象，我们不妨通过一个具体的案例，来看看到底在什么的场景下才需要上下文呢？

在执行一个比较长的请求任务时，这个请求可能会经历很多层的方法调用，假设我们需要将最开始的方法的中间结果传递到末尾的方法中进行计算，一个简单的实现方式就是在每个函数中新增这个中间结果的参数，依次传递下去。代码如下：

```
public class ContextTest {

 // 上下文类
```

```
// 上下文类
public class Context {
 private String name;
 private long id

 public long getId() {
  return id;
 }

 public void setId(long id) {
  this.id = id;
 }

 public String getName() {
  return this.name;
 }

 public void setName(String name) {
  this.name = name;
 }
}

// 设置上下文名字
public class QueryNameAction {
 public void execute(Context context) {
  try {
   Thread.sleep(1000L);
   String name = Thread.currentThread().getName();
   context.setName(name);
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
 }
}

// 设置上下文ID
public class QueryIdAction {
 public void execute(Context context) {
```

```
    try {
     Thread.sleep(1000L);
     long id = Thread.currentThread().getId();
     context.setId(id);
    } catch (InterruptedException e) {
     e.printStackTrace();
    }
   }
  }

  // 执行方法
  public class ExecutionTask implements Runnable {

   private QueryNameAction queryNameAction = new QueryNameAction();
   private QueryIdAction queryIdAction = new QueryIdAction();

   @Override
   public void run() {
    final Context context = new Context();
    queryNameAction.execute(context);
    System.out.println("The name query successful");
    queryIdAction.execute(context);
    System.out.println("The id query successful");

    System.out.println("The Name is " + context.getName() + " and id " + context.getId());
   }
  }

  public static void main(String[] args) {
   IntStream.range(1, 5).forEach(i -> new Thread(new ContextTest().new ExecutionTask()).start());
  }
 }
```

执行结果：

```
The name query successful
The name query successful
The name query successful
The name query successful
The id query successful
The id query successful
The id query successful
The id query successful
The Name is Thread-1 and id 11
The Name is Thread-2 and id 12
The Name is Thread-3 and id 13
The Name is Thread-0 and id 10
```

然而这种方式太笨拙了，每次调用方法时，都需要传入Context作为参数，而且影响一些中间公共方法的封装。

那能不能设置一个全局变量呢？如果是在多线程情况下，需要考虑线程安全，这样的话就又涉及到了锁竞争。

除了以上这些方法，其实我们还可以使用ThreadLocal实现上下文。ThreadLocal是线程本地变量，可以实现多线程的数据隔离。ThreadLocal为每一个使用该变量的线程都提供一份独立的副本，线程间的数据是隔离的，每一个线程只能访问各自内部的副本变量。

ThreadLocal中有三个常用的方法：set、get、initialValue，我们可以通过以下一个简单的例子来看看ThreadLocal的使用：

```
private void testThreadLocal() {
    Thread t = new Thread() {
        ThreadLocal<String> mStringThreadLocal = new ThreadLocal<String>();

        @Override
        public void run() {
            super.run();
            mStringThreadLocal.set("test");
            mStringThreadLocal.get();
        }
    };

    t.start();
}
```

接下来，我们使用ThreadLocal来重新实现最开始的上下文设计。你会发现，我们在两个方法中并没有通过变量来传递上下文，只是通过ThreadLocal获取了当前线程的上下文信息：

```
public class ContextTest {
 // 上下文类
 public static class Context {
  private String name;
  private long id;

  public long getId() {
   return id;
  }

  public void setId(long id) {
   this.id = id;
  }

  public String getName() {
   return this.name;
  }

  public void setName(String name) {
```

```
 public void setName(String name) {
  this.name = name;
 }
}

// 复制上下文到ThreadLocal中
public final static class ActionContext {

 private static final ThreadLocal<Context> threadLocal = new ThreadLocal<Context>() {
  @Override
  protected Context initialValue() {
   return new Context();
  }
 };

 public static ActionContext getActionContext() {
  return ContextHolder.actionContext;
 }

 public Context getContext() {
  return threadLocal.get();
 }

 // 获取ActionContext单例
 public static class ContextHolder {
  private final static ActionContext actionContext = new ActionContext();
 }
}

// 设置上下文名字
public class QueryNameAction {
 public void execute() {
  try {
   Thread.sleep(1000L);
   String name = Thread.currentThread().getName();
   ActionContext.getActionContext().getContext().setName(name);
  } catch (InterruptedException e) {
   e.printStackTrace();
```

```
    }
   }
  }

  // 设置上下文ID
  public class QueryIdAction {
   public void execute() {
    try {
     Thread.sleep(1000L);
     long id = Thread.currentThread().getId();
     ActionContext.getActionContext().getContext().setId(id);
    } catch (InterruptedException e) {
     e.printStackTrace();
    }
   }
  }

  // 执行方法
  public class ExecutionTask implements Runnable {
   private QueryNameAction queryNameAction = new QueryNameAction();
   private QueryIdAction queryIdAction = new QueryIdAction();

   @Override
   public void run() {
    queryNameAction.execute();//设置线程名
    System.out.println("The name query successful");
    queryIdAction.execute();//设置线程ID
    System.out.println("The id query successful");

    System.out.println("The Name is " + ActionContext.getActionContext().getContext().getName() + " and id " + Ac
   }
  }

  public static void main(String[] args) {
   IntStream.range(1, 5).forEach(i -> new Thread(new ContextTest().new ExecutionTask()).start());
  }
 }
```

运行结果：

```
The name query successful
The name query successful
The name query successful
The name query successful
The id query successful
The id query successful
The id query successful
The id query successful
The Name is Thread-2 and id 12
The Name is Thread-0 and id 10
The Name is Thread-1 and id 11
The Name is Thread-3 and id 13
```

Thread-Per-Message设计模式

Thread-Per-Message设计模式翻译过来的意思就是每个消息一个线程的意思。例如，我们在处理Socket通信的时候，通常是一个线程处理事件监听以及I/O读写，如果I/O读写操作非常耗时，这个时候便会影响到事件监听处理事件。

这个时候Thread-Per-Message模式就可以很好地解决这个问题，一个线程监听I/O事件，每当监听到一个I/O事件，则交给另一个处理线程执行I/O操作。下面，我们还是通过一个例子来学习下该设计模式的实现。

```
//IO处理
public class ServerHandler implements Runnable{
 private Socket socket;

    public ServerHandler(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        BufferedReader in = null;
        PrintWriter out = null;
        String msg = null;
```

```
        String msg = null;
        try {
            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(),true);
            while ((msg = in.readLine()) != null && msg.length()!=0) {//当连接成功后在此等待接收消息（挂起，进入阻
                System.out.println("server received : " + msg);
                out.print("received~\n");
                out.flush();
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
         try {
                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                out.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
//Socket启动服务
public class Server {

 private static int DEFAULT_PORT = 12345;
 private static ServerSocket server;
```

```
public static void start() throws IOException {
 start(DEFAULT_PORT);
}

public static void start(int port) throws IOException {
 if (server != null) {
  return;
 }

 try {
  //启动服务
  server = new ServerSocket(port);
  // 通过无线循环监听客户端连接
  while (true) {

   Socket socket = server.accept();
   // 当有新的客户端接入时，会执行下面的代码
   long start = System.currentTimeMillis();
   new Thread(new ServerHandler(socket)).start();

   long end = System.currentTimeMillis();

   System.out.println("Spend time is " + (end - start));
  }
 } finally {
  if (server != null) {
   System.out.println("服务器已关闭。");
   server.close();
  }

 }

}

public static void main(String[] args) throws InterruptedException{

 // 运行服务端
```

```
    new Thread(new Runnable() {
     public void run() {
      try {
       Server.start();
      } catch (IOException e) {
       e.printStackTrace();
      }
     }
    }).start();

  }
}
```

以上，我们是完成了一个使用Thread-Per-Message设计模式实现的Socket服务端的代码。但这里是有一个问题的，你发现了吗？

使用这种设计模式，如果遇到大的高并发，就会出现严重的性能问题。如果针对每个I/O请求都创建一个线程来处理，在有大量请求同时进来时，就会创建大量线程，而此时JVM有可能会因为无法处理这么多线程，而出现内存溢出的问题。

退一步讲，即使是不会有大量线程的场景，每次请求过来也都需要创建和销毁线程，这对系统来说，也是一笔不小的性能开销。

面对这种情况，我们可以使用线程池来代替线程的创建和销毁，这样就可以避免创建大量线程而带来的性能问题，是一种很好的调优方法。

Worker-Thread设计模式

这里的Worker是工人的意思，代表在Worker Thread设计模式中，会有一些工人（线程）不断轮流处理过来的工作，当没有工作时，工人则会处于等待状态，直到有新的工作进来。除了工人角色，Worker Thread设计模式中还包括了流水线和产品。

这种设计模式相比Thread-Per-Message设计模式，可以减少频繁创建、销毁线程所带来的性能开销，还有无限制地创建线程所带来的内存溢出风险。

我们可以假设一个场景来看下该模式的实现，通过Worker Thread设计模式来完成一个物流分拣的作业。

假设一个物流仓库的物流分拣流水线上有8个机器人，它们不断从流水线上获取包裹并对其进行包装，送其上车。当仓库中的商品被打包好后，会投放到物流分拣流水线上，而不是直接交给机

器人，机器人会再从流水线中随机分拣包裹。代码如下：

```
//包裹类
public class Package {
 private String name;
 private String address;

 public String getName() {
  return name;
 }

 public void setName(String name) {
  this.name = name;
 }

 public String getAddress() {
  return address;
 }

 public void setAddress(String address) {
  this.address = address;
 }

 public void execute() {
  System.out.println(Thread.currentThread().getName()+" executed "+this);
 }
}
```

```
//流水线
public class PackageChannel {
 private final static int MAX_PACKAGE_NUM = 100;

 private final Package[] packageQueue;
 private final Worker[] workerPool;
 private int head;
 private int tail;
 private int count;
```

```
private int count;

public PackageChannel(int workers) {
 this.packageQueue = new Package[MAX_PACKAGE_NUM];
 this.head = 0;
 this.tail = 0;
 this.count = 0;
 this.workerPool = new Worker[workers];
 this.init();
}

private void init() {
 for (int i = 0; i < workerPool.length; i++) {
  workerPool[i] = new Worker("Worker-" + i, this);
 }
}

/**
 * push switch to start all of worker to work
 */
public void startWorker() {
 Arrays.asList(workerPool).forEach(Worker::start);
}

public synchronized void put(Package packagereq) {
 while (count >= packageQueue.length) {
  try {
   this.wait();
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
 }
 this.packageQueue[tail] = packagereq;
 this.tail = (tail + 1) % packageQueue.length;
 this.count++;
 this.notifyAll();
}
```

```
 public synchronized Package take() {
  while (count <= 0) {
   try {
    this.wait();
   } catch (InterruptedException e) {
    e.printStackTrace();
   }
  }
  Package request = this.packageQueue[head];
  this.head = (this.head + 1) % this.packageQueue.length;
  this.count--;
  this.notifyAll();
  return request;
 }

}
```

```
//机器人
public class Worker extends Thread{
  private static final Random random = new Random(System.currentTimeMillis());
  private final PackageChannel channel;

    public Worker(String name, PackageChannel channel) {
        super(name);
        this.channel = channel;
    }

    @Override
    public void run() {
        while (true) {
            channel.take().execute();

            try {
                Thread.sleep(random.nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

```
public class Test {
    public static void main(String[] args) {
          //新建8个工人
        final PackageChannel channel = new PackageChannel(8);
          //开始工作
        channel.startWorker();
          //为流水线添加包裹
        for(int i=0; i<100; i++) {
          Package packagereq = new Package();
          packagereq.setAddress("test");
          packagereq.setName("test");
          channel.put(packagereq);
        }
    }
}
```

我们可以看到，这里有8个工人在不断地分拣仓库中已经包装好的商品。

总结

平时，如果需要传递或隔离一些线程变量时，我们可以考虑使用上下文设计模式。在数据库读写分离的业务场景中，则经常会用到ThreadLocal实现动态切换数据源操作。但在使用ThreadLocal时，我们需要注意内存泄漏问题，在之前的第25讲中，我们已经讨论过这个问题了。

当主线程处理每次请求都非常耗时时，就可能出现阻塞问题，这时候我们可以考虑将主线程业务分工到新的业务线程中，从而提高系统的并行处理能力。而 Thread-Per-Message 设计模式以及 Worker-Thread 设计模式则都是通过多线程分工来提高系统并行处理能力的设计模式。

思考题

除了以上这些多线程的设计模式，平时你还使用过其它的设计模式来优化多线程业务吗？

期待在留言区看到你的答案。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起讨论。

精选留言

峰

5

讲真，我是觉得设计模式是优化代码组织结构的，性能提升只是因为你的实现途径导致了你适合用某种设计模式，so感觉这样标题怪怪的。

如果要这么说的话，mapreduce或者说javase引入的forkjoin，流水线模式，cow就都是了。

2019-07-27

作者回复

是的，有些设计模式更多的是优化代码逻辑结构，但还是有很多设计模式也起到了优化性能效果。例如，文中的Worker-Thread 设计模式其实就是一种线程池的优化方式，在并发量大时，一般的代码逻辑要么是串行执行，要么使用创建线程并发执行，在大量并发时两者都可能会出现性能瓶颈，而这种Worker-Thread 这样的设计方式则即可以并发执行，又避免创建过多的线程导致性能瓶颈。

2019-07-27

张德

1

老师好 我还用过reactor模式 这个多线程的thread-per-message感觉和reactor模式有点像 又有一些区别 但我就是总结不出之间的区别 老师能不能点化一下 多谢

2019-07-30

Jxin

1

1.最常用的多线程设计模式应该是生产者消费者模式，在分布式系统，该模式现在一般也由Mq来承接。（以rocketMq为例）好处有：消峰，解耦，消息堆积，多broker并行消费，单broker串行（顺序）消费，发布订阅，分组消费，失败重试，死信管理等等。
2.其他的业务不常用，比如Immutability（不变模式，Long类的内部静态类对象池），还有个实时赋值COW，指得一提的应该还有个Actor，但java不支持，要玩又得引第三方包，所以java生产也不会用。
3.forkjoin并行处理也是使用的多线程执行子任务，但这个应该算不上多线程设计模式，感觉说是多线程应用更好，其中的任务窃取挺有意思。

2019-07-29

作者回复

很赞

2019-07-31

QQ怪 1

比起Worker-Thread 设计模式类似工厂车间工人的工作模式，还有用的比较多的是生产者和消费者模式，与之前的不同的是，生产者和消费者模式核心是一个任务队列，生产者生产任务到任务队列中，消费者从队列消费任务，优点是解耦和平衡两者之前的速度差异。

2019-07-27

作者回复

对的，分析的很到位。下一讲中我们会详细聊到生产者消费者模式。

2019-07-29

nightmare 1

一个注册逻辑，下面有注册实现数组，注册实现里面有队列，并且本身实现runable ，注册门面依次从注册实现数组获取一个注册实现 并把请求放到注册实现的队列中，请求由一个注册实现来完成，请求由唯一的注册实现来完成，不会有并发问题 而且如果 注册实现有复杂业务 还可以加上 work thread模式来优化

2019-07-27

作者回复

赞，现学现用

2019-07-29

Liam 0

老师好，文中提到，通过threadlocal动态切换数据源是什么意思？指的是用一个threadlocal的map管理多数据源的连接，每次都从map去拿不同datasource的连接吗？

2019-07-29

作者回复

对的，例如读写分离，ThreadLocal存放了读从库数据源和写主库数据源，如果是查询操作，则切换为读数据源，如果是新增修改删除操作，则切换为写数据源。

2019-07-31

-W.LI- 0

老师啊！有没有好的书推荐，我觉得设计模式很好，源码应该是学习设计模式最好的老师，可

是我的能力看源码感觉太早了。我就之前看过header first。感觉理解完全不够。好的设计模式和算法都能在系统性能有瓶颈的时候提升系统。我认识到它的重要了可是不得门，入不了难受啊。

2019-07-28

作者回复

我记得大学时候读过一本《大话设计模式》的书籍不错，讲解的通俗易懂。

2019-07-29

-W.LI- 0

老师好!Thread-Per-Message 设计模式就是分离阻塞和非阻塞的呗。阻塞的部分通过多路复用。非阻塞的的那部分就丢线程池并发处理。

2019-07-28

晓杰 0

请问老师packageChannel中的成员变量count不会存在线程安全问题吗

2019-07-28

作者回复

我们对put take方法已经加锁了，所以是线程安全的。

2019-07-29

陆离 0

老师，讲到并发这里，我想问一个前面讲过的synchronized锁升级的几个问题。

1.当锁由无锁状态升级到偏向锁时，除了将Mark Work中的线程ID替换是否还有其他操作？替换完线程ID就代表获取到了锁吗？

2.当锁状态为偏向锁状态时, 其他线程竞争锁只是CAS替换线程ID吗？如果之前的线程还没有执行完呢？

3.针对第2个问题，假设线程T1获取到了偏向锁，将线程ID设为T1。线程T2尝试获取偏向锁时，先检测锁的Mark Word线程ID是否为T2,如果不是，会CAS替换，这个时候的期望值为null,更新值为T2，失败后进入偏向锁撤销。stop-the-world后检测T1是否存活，如果否清空Mark work线程ID，锁恢复为无锁状态，唤醒T2,接着尝试获取锁。流程是这样的吗？

4.当锁升级为轻量级锁时，获取锁的标志是锁指针指向线程的锁记录，当有其他线程尝试CAS获取锁时，期望值是无锁时，Mark word中为hash age 01这样的内容吗？

5.当线程释放轻量锁时，需要将锁记录替换回Mark Word中，这种情况下锁还未释放为什么会有失败？

6.当锁升级为重量锁后，开始使用monitor对象，为什么Mark Word中还会把内容替换为指向线程锁记录的指针？这个时候还需要使用Mark word吗？

期待老师及同学的解答

2019-07-27

undifined 0

老师，有一个问题没想明白，就是异步的请求处理中，每一个线程接收将请求交给处理的线程后，怎么拿到返回结果并返回给用户呢

2019-07-27

作者回复

可以通过Future模式拿到返回结果，虽然是异步执行，如果要等待返回结果，则主线程还是在阻塞等待。

2019-07-27