二

28 如何使用设计模式优化并发编程?

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
        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() + " a
        }
    }

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

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) {
                        this.name = name;
                }
        }

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

                private static final ThreadLocal<Context> threadLocal = new ThreadL
                        @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 Acti
                }
        }

        // 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().setNa
                        } 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
                        } 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.getAction
                }
        }

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

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;
        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
                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 "+th
        }
}
// 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;

        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？

上一页 下一页