Java 线程池 ThreadPoolExecutor 八种拒绝策略浅析

KL zhisheng 2019-08-29

前言

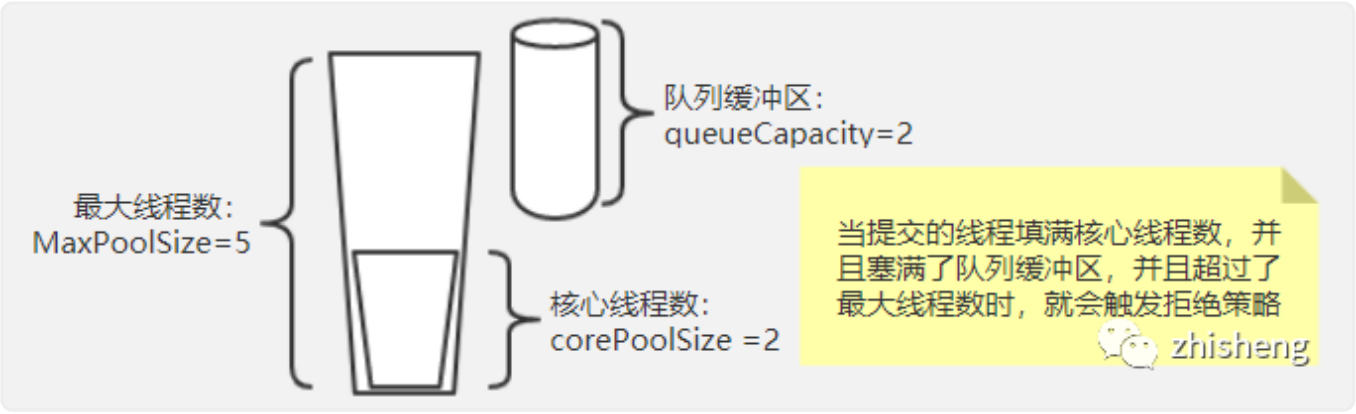
谈到 Java 的线程池最熟悉的莫过于 ExecutorService 接口了，jdk1.5 新增的 java.util.concurrent 包下的这个 api，大大的简化了多线程代码的开发。而不论你用 FixedThreadPool 还是 CachedThreadPool 其背后实现都是ThreadPoolExecutor。ThreadPoolExecutor 是一个典型的缓存池化设计的产物，因为池子有大小，当池子体积不够承载时，就涉及到拒绝策略。JDK 中已经预设了 4 种线程池拒绝策略，下面结合场景详细聊聊这些策略的使用场景，以及我们还能扩展哪些拒绝策略。

池化设计思想

池话设计应该不是一个新名词。我们常见的如 Java 线程池、JDBC 连接池、Redis 连接池等就是这类设计的代表实现。这种设计会初始预设资源，解决的问题就是抵消每次获取资源的消耗，如创建线程的开销，获取远程连接的开销等。就好比你去食堂打饭，打饭的大妈会先把饭盛好几份放那里，你来了就直接拿着饭盒加菜即可，不用再临时又盛饭又打菜，效率就高了。除了初始化资源，池化设计还包括如下这些特征：池子的初始值、池子的活跃值、池子的最大值等，这些特征可以直接映射到 Java 线程池和数据库连接池的成员属性中。

线程池触发拒绝策略的时机

和数据源连接池不一样，线程池除了初始大小和池子最大值，还多了一个阻塞队列来缓冲。数据源连接池一般请求的连接数超过连接池的最大值的时候就会触发拒绝策略，策略一般是阻塞等待设置的时间或者直接抛异常。而线程池的触发时机如下图：



img

如图，想要了解线程池什么时候触发拒绝粗略，需要明确上面三个参数的具体含义，是这三个参数总体协调的结果，而不是简单的超过最大线程数就会触发线程拒绝粗略，当提交的任务数大于 corePoolSize 时，会优先放到队列缓冲区，只有填满了缓冲区后，才会判断当前运行的任务是否大于 maxPoolSize，小于时会新建线程处理。大于时就触发了拒绝策略，总结就是：当前提交任务数大于（maxPoolSize + queueCapacity）时就会触发线程池的拒绝策略了。

JDK内置4种线程池拒绝策略

拒绝策略接口定义

在分析 JDK 自带的线程池拒绝策略前，先看下 JDK 定义的 拒绝策略接口，如下：

```
1  public interface RejectedExecutionHandler {
2      void rejectedExecution(Runnable r, ThreadPoolExecutor executor);
3  }
```

接口定义很明确，当触发拒绝策略时，线程池会调用你设置的具体的策略，将当前提交的任务以及线程池实例本身传递给你处理，具体作何处理，不同场景会有不同的考虑，下面看 JDK 为我们内置了哪些实现：

CallerRunsPolicy（调用者运行策略）

```
1      public static class CallerRunsPolicy implements RejectedExecutionH
2
```

```
        public CallerRunsPolicy() { }

        public void rejectedExecution(Runnable r, ThreadPoolExecutor e
            if (!e.isShutdown()) {
                r.run();
            }
        }
    }
```

功能：当触发拒绝策略时，只要线程池没有关闭，就由提交任务的当前线程处理。

使用场景：一般在不允许失败的、对性能要求不高、并发量较小的场景下使用，因为线程池一般情况下不会关闭，也就是提交的任务一定会被运行，但是由于是调用者线程自己执行的，当多次提交任务时，就会阻塞后续任务执行，性能和效率自然就慢了。

AbortPolicy（中止策略）

```
    public static class AbortPolicy implements RejectedExecutionHandle

        public AbortPolicy() { }

        public void rejectedExecution(Runnable r, ThreadPoolExecutor e
            throw new RejectedExecutionException("Task " + r.toString(
                                                 " rejected from " +
                                                 e.toString());
        }
    }
```

功能：当触发拒绝策略时，直接抛出拒绝执行的异常，中止策略的意思也就是打断当前执行流程

使用场景：这个就没有特殊的场景了，但是一点要正确处理抛出的异常。ThreadPoolExecutor 中默认的策略就是AbortPolicy，ExecutorService 接口的系列 ThreadPoolExecutor 因为都没有显示的设置拒绝策略，所以默认的都是这个。但是请注意，ExecutorService 中的线程池实例队列都是无界的，也就是说把内存撑爆了都不会触发拒绝策略。当自己自定义线程池实例时，使用这个策略一定要处理好触发策略时抛的异常，因为他会打断当前的执行流程。

DiscardPolicy（丢弃策略）

```
1      public static class DiscardPolicy implements RejectedExecutionHand
2
3          public DiscardPolicy() { }
4
5          public void rejectedExecution(Runnable r, ThreadPoolExecutor e
6          }
7      }
```

功能：直接静悄悄的丢弃这个任务，不触发任何动作

使用场景：如果你提交的任务无关紧要，你就可以使用它 。因为它就是个空实现，会悄无声息的吞噬你的的任务。所以这个策略基本上不用了

DiscardOldestPolicy（弃老策略）

```
1      public static class DiscardOldestPolicy implements RejectedExecut
2
3          public DiscardOldestPolicy() { }
4
5          public void rejectedExecution(Runnable r, ThreadPoolExecutor e
6              if (!e.isShutdown()) {
7                  e.getQueue().poll();
8                  e.execute(r);
9              }
10         }
11     }
```

功能：如果线程池未关闭，就弹出队列头部的元素，然后尝试执行

使用场景：这个策略还是会丢弃任务，丢弃时也是毫无声息，但是特点是丢弃的是老的未执行的任务，而且是待执行优先级较高的任务。基于这个特性，我能想到的场景就是，发布消息，和修改消息，当消息发布出去后，还未执行，此时更新的消息又来了，这个时候未执行的消息的版本比现在提交的消息版本要低就可以被丢弃了。因为队列中还有可能存在消息版本更低的消息会排队执行，所以在真正处理消息的时候一定要做好消息的版本比较

第三方实现的拒绝策略

Dubbo 中的线程拒绝策略

```
1  public class AbortPolicyWithReport extends ThreadPoolExecutor.AbortPo
2
3      protected static final Logger logger = LoggerFactory.getLogger(Ab
4
5      private final String threadName;
6
7      private final URL url;
8
9      private static volatile long lastPrintTime = 0;
10
11     private static Semaphore guard = new Semaphore(1);
12
13     public AbortPolicyWithReport(String threadName, URL url) {
14         this.threadName = threadName;
15         this.url = url;
16     }
17
18     @Override
19     public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
20         String msg = String.format("Thread pool is EXHAUSTED!" +
21                         " Thread Name: %s, Pool Size: %d (active: %d,
22                         " Executor status:(isShutdown:%s, isTerminate
23                 threadName, e.getPoolSize(), e.getActiveCount(), e.ge
24                 e.getTaskCount(), e.getCompletedTaskCount(), e.isShut
25                 url.getProtocol(), url.getIp(), url.getPort());
26         logger.warn(msg);
27         dumpJStack();
28         throw new RejectedExecutionException(msg);
29     }
30
31     private void dumpJStack() {
32        //省略实现
33     }
34 }
```

可以看到，当dubbo的工作线程触发了线程拒绝后，主要做了三个事情，原则就是尽量让使用者清楚触发线程拒绝策略的真实原因

- 输出了一条警告级别的日志，日志内容为线程池的详细设置参数，以及线程池当前的状态，还有当前拒绝任务的一些详细信息。可以说，这条日志，使用dubbo的有过生产运维经验的或多或少是见过的，这个日志简直就是日志打印的典范，其他的日志打印的典范还有spring。得益于这么详细的日志，可以很容易定位到问题所在

- 输出当前线程堆栈详情，这个太有用了，当你通过上面的日志信息还不能定位问题时，案发现场的dump线程上下文信息就是你发现问题的救命稻草，这个可以参考《dubbo线程池耗尽事件-"CyclicBarrier惹的祸"》
- 继续抛出拒绝执行异常，使本次任务失败，这个继承了JDK默认拒绝策略的特性

Netty 中的线程池拒绝策略

```
    private static final class NewThreadRunsPolicy implements RejectedE
        NewThreadRunsPolicy() {
            super();
        }

        public void rejectedExecution(Runnable r, ThreadPoolExecutor e
            try {
                final Thread t = new Thread(r, "Temporary task executo
                t.start();
            } catch (Throwable e) {
                throw new RejectedExecutionException(
                        "Failed to start a new thread", e);
            }
        }
    }
```

Netty 中的实现很像 JDK 中的 CallerRunsPolicy，舍不得丢弃任务。不同的是，CallerRunsPolicy 是直接在调用者线程执行的任务。而 Netty是新建了一个线程来处理的。所以，Netty的实现相较于调用者执行策略的使用面就可以扩展到支持高效率高性能的场景了。但是也要注意一点，Netty的实现里，在创建线程时未做任何的判断约束，也就是说只要系统还有资源就会创建新的线程来处理，直到new不出新的线程了，才会抛创建线程失败的异常

ActiveMQ 中的线程池拒绝策略

```
new RejectedExecutionHandler() {
                @Override
                public void rejectedExecution(final Runnable r, final
                    try {
                        executor.getQueue().offer(r, 60, TimeUnit.SECO
                    } catch (InterruptedException e) {
                        throw new RejectedExecutionException("Interru
                    }
```

```
 9
10                     throw new RejectedExecutionException("Timed Out wh
11                 }
12             });
```

activeMq中的策略属于最大努力执行任务型，当触发拒绝策略时，在尝试一分钟的时间重新将任务塞进任务队列，当一分钟超时还没成功时，就抛出异常

PinPoint 中的线程池拒绝策略

```
1  public class RejectedExecutionHandlerChain implements RejectedExecutio
2      private final RejectedExecutionHandler[] handlerChain;
3
4      public static RejectedExecutionHandler build(List<RejectedExecutio
5          Objects.requireNonNull(chain, "handlerChain must not be null")
6          RejectedExecutionHandler[] handlerChain = chain.toArray(new Re
7          return new RejectedExecutionHandlerChain(handlerChain);
8      }
9
10     private RejectedExecutionHandlerChain(RejectedExecutionHandler[] h
11         this.handlerChain = Objects.requireNonNull(handlerChain, "hand
12     }
13
14     @Override
15     public void rejectedExecution(Runnable r, ThreadPoolExecutor execu
16         for (RejectedExecutionHandler rejectedExecutionHandler : hand
17             rejectedExecutionHandler.rejectedExecution(r, executor);
18         }
19     }
20 }
```

pinpoint的拒绝策略实现很有特点，和其他的实现都不同。他定义了一个拒绝策略链，包装了一个拒绝策略列表，当触发拒绝策略时，会将策略链中的rejectedExecution依次执行一遍

结语

前文从线程池设计思想，以及线程池触发拒绝策略的时机引出java线程池拒绝策略接口的定义。并辅以JDK内置4种以及四个第三方开源软件的拒绝策略定义描述了线程池拒绝策略实现

的各种思路和使用场景。希望阅读此文后能让你对java线程池拒绝策略有更加深刻的认识，能够根据不同的使用场景更加灵活的应用。

> 作者：KL
>
> 链接：http://www.kailing.pub/article/index/arcid/255.html

END

阅读原文