第11章　Java并发编程实践

当你在进行并发编程时，看着程序的执行速度在自己的优化下运行得越来越快，你会觉得越来越有成就感，这就是并发编程的魅力。但与此同时，并发编程产生的问题和风险可能也会随之而来。本章先介绍几个并发编程的实战案例，然后再介绍如何排查并发编程造成的问题。

11.1 生产者和消费者模式

在并发编程中使用生产者和消费者模式能够解决绝大多数并发问题。该模式通过平衡生产线程和消费线程的工作能力来提高程序整体处理数据的速度。

在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这种生产消费能力不均衡的问题，便有了生产者和消费者模式。

什么是生产者和消费者模式

生产者和消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通信，而是通过阻塞队列来进行通信，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

这个阻塞队列就是用来给生产者和消费者解耦的。纵观大多数设计模式，都会找一个第三者出来进行解耦，如工厂模式的第三者是工厂类，模板模式的第三者是模板类。在学习一些设计模式的过程中，先找到这个模式的第三者，能帮助我们快速熟悉一个设计模式。

11.1.1 生产者消费者模式实战

我和同事一起利用业余时间开发的Yuna工具中使用了生产者和消费者模式。我先介绍下Yuna[1]工具，在阿里巴巴很多同事都喜欢通过邮件分享技术文章，因为通过邮件分享很方便，大家在网上看到好的技术文章，执行复制→粘贴→发送就完成了一次分享，但是我发现技术文章不能沉淀下来，新来的同事看不到以前分享的技术文章，大家也很难找到以前分享过的技术文章。为了解决这个问题，我们开发了一个Yuna工具。

我们申请了一个专门用来收集分享邮件的邮箱，比如share@alibaba.com，大家将分享的文章发送到这个邮箱，让大家每次都抄送到这个邮箱肯定很麻烦，所以我们的做法是将这个邮箱地址放在部门邮件列表里，所以分享的同事只需要和以前一样向整个部门分享文章就行。Yuna工具通过读取邮件服务器里该邮箱的邮件，把所有分享的邮件下载下来，包括邮件的附件、图片和邮件回复。因为我们可能会从这个邮箱里下载到一些非分享的文章，所以我们要求分享的邮件标题必须带有一个关键字，比如“内贸技术分享”。下载完邮件之后，通过confluence的Web Service接口，把文章插入到confluence里，这样新同事就可以在confluence里看以前分享过的文章了，并且Yuna工具还可以自动把文章进行分类和归档。

为了快速上线该功能，当时我们花了3天业余时间快速开发了Yuna 1.0版本。在1.0版本中并没有使用生产者消费模式，而是使用单线程来处理，因为当时只需要处理我们一个部门的邮件，所以单线程明显够用，整个过程是串行执行的。在一个线程里，程序先抽取全部的邮件，转化为文章对象，然后添加全部的文章，最后删除抽取过的邮件。代码如下。

```
public void extract() {
        logger.debug("开始" + getExtractorName() + "。。");
        // 抽取邮件
        List<Article> articles = extractEmail();
        // 添加文章
        for (Article article : articles) {
            addArticleOrComment(article);
        }
        // 清空邮件
        cleanEmail();
        logger.debug("完成" + getExtractorName() + "。。");
```

```
    }
```

Yuna工具在推广后，越来越多的部门使用这个工具，处理的时间越来越慢，Yuna是每隔5分钟进行一次抽取的，而当邮件多的时候一次处理可能就花了几分钟，于是我在Yuna 2.0版本里使用了生产者消费者模式来处理邮件，首先生产者线程按一定的规则去邮件系统里抽取邮件，然后存放在阻塞队列里，消费者从阻塞队列里取出文章后插入到conflunce里。代码如下。

```
public class QuickEmailToWikiExtractor extends AbstractExtractor {
private ThreadPoolExecutor      threadsPool;
private ArticleBlockingQueue<ExchangeEmailShallowDTO> emailQueue;
public QuickEmailToWikiExtractor() {
        emailQueue= new ArticleBlockingQueue<ExchangeEmailShallowDTO>();
        int corePoolSize = Runtime.getRuntime().availableProcessors() * 2;
        threadsPool = new ThreadPoolExecutor(corePoolSize, corePoolSize, 10l, TimeUnit
        SECONDS,
                new LinkedBlockingQueue<Runnable>(2000));
    }
public void extract() {
        logger.debug("开始" + getExtractorName() + "。。");
        long start = System.currentTimeMillis();
        // 抽取所有邮件放到队列里
        new ExtractEmailTask().start();
        // 把队列里的文章插入到Wiki
        insertToWiki();
        long end = System.currentTimeMillis();
        double cost = (end - start) / 1000;
        logger.debug("完成" + getExtractorName() + ",花费时间:" + cost + "秒");
    }
    /**
     * 把队列里的文章插入到Wiki
     */
    private void insertToWiki() {
        // 登录Wiki,每间隔一段时间需要登录一次
        confluenceService.login(RuleFactory.USER_NAME, RuleFactory.PASSWORD);
        while (true) {
            // 2秒内取不到就退出
            ExchangeEmailShallowDTO email = emailQueue.poll(2, TimeUnit.SECONDS);
            if (email == null) {
                break;
            }
            threadsPool.submit(new insertToWikiTask(email));
        }
    }
    protected List<Article> extractEmail() {
        List<ExchangeEmailShallowDTO> allEmails = getEmailService().queryAllEmails();
        if (allEmails == null) {
            return null;
        }
        for (ExchangeEmailShallowDTO exchangeEmailShallowDTO : allEmails) {
            emailQueue.offer(exchangeEmailShallowDTO);
```

```
            }
            return null;
        }
        /**
         * 抽取邮件任务
         *
         * @author tengfei.fangtf
         */
        public class ExtractEmailTask extends Thread {
            public void run() {
                extractEmail();
            }
        }
    }
```

代码的执行逻辑是，生产者启动一个线程把所有邮件全部抽取到队列中，消费者启动CPU*2个线程数处理邮件，从之前的单线程处理邮件变成了现在的多线程处理，并且抽取邮件的线程不需要等处理邮件的线程处理完再抽取新邮件，所以使用了生产者和消费者模式后，邮件的整体处理速度比以前要快了几倍。

[1] Yuna取名自我非常喜欢的一款RPG游戏《最终幻想》中女主角的名字。

11.1.2 多生产者和多消费者场景

在多核时代，多线程并发处理速度比单线程处理速度更快，所以可以使用多个线程来生产数据，同样可以使用多个消费线程来消费数据。而更复杂的情况是，消费者消费的数据，有可能需要继续处理，于是消费者处理完数据之后，它又要作为生产者把数据放在新的队列里，交给其他消费者继续处理，如图11-1所示。

图11-1 多生产者消费者模式

我们在一个长连接服务器中使用了这种模式，生产者1负责将所有客户端发送的消息存放在阻塞队列1里，消费者1从队列里读消息，然后通过消息ID进行散列得到N个队列中的一个，然后根据编号将消息存放在到不同的队列里，每个阻塞队列会分配一个线程来消费阻塞队列

里的数据。如果消费者2无法消费消息，就将消息再抛回到阻塞队列1中，交给其他消费者处理。

以下是消息总队列的代码。

```
/**
 * 总消息队列管理
 *
 * @author tengfei.fangtf
 */
public class MsgQueueManager implements IMsgQueue{
    private static final Logger              LOGGER
 = LoggerFactory.getLogger(MsgQueueManager.class);
    /**
     * 消息总队列
     */
    public final BlockingQueue<Message> messageQueue;
    private MsgQueueManager() {
        messageQueue = new LinkedTransferQueue<Message>();
    }
    public void put(Message msg) {
        try {
            messageQueue.put(msg);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
    public Message take() {
        try {
            return messageQueue.take();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return null;
    }
}
```

启动一个消息分发线程。在这个线程里子队列自动去总队列里获取消息。

```
/**
     * 分发消息，负责把消息从大队列塞到小队列里
     *
     * @author tengfei.fangtf
     */
    static class DispatchMessageTask implements Runnable {
        @Override
        public void run() {
            BlockingQueue<Message> subQueue;
            for (;;) {
```

```
                // 如果没有数据，则阻塞在这里
                Message msg = MsgQueueFactory.getMessageQueue().take();
                // 如果为空，则表示没有Session机器连接上来，
                // 需要等待，直到有Session机器连接上来
                while ((subQueue = getInstance().getSubQueue()) == null) {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    }
                }
                // 把消息放到小队列里
                try {
                    subQueue.put(msg);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        }
    }
```

使用散列(hash)算法获取一个子队列，代码如下。

```
/**
     * 均衡获取一个子队列。
     *
     * @return
     */
    public BlockingQueue<Message> getSubQueue() {
        int errorCount = 0;
        for (;;) {
            if (subMsgQueues.isEmpty()) {
                return null;
            }
            int index = (int) (System.nanoTime() % subMsgQueues.size());
            try {
                return subMsgQueues.get(index);
            } catch (Exception e) {
                // 出现错误表示，在获取队列大小之后，队列进行了一次删除操作
                LOGGER.error("获取子队列出现错误", e);
                if ((++errorCount) < 3) {
                    continue;
                }
            }
        }
    }
```

使用的时候，只需要往总队列里发消息。

```
// 往消息队列里添加一条消息
```

```
 IMsgQueue messageQueue = MsgQueueFactory.getMessageQueue();
Packet msg = Packet.createPacket(Packet64FrameType.
        TYPE_DATA, "{}".getBytes(), (short) 1);
messageQueue.put(msg);
```

11.1.3 线程池与生产消费者模式

Java中的线程池类其实就是一种生产者和消费者模式的实现方式，但是我觉得其实现方式更加高明。生产者把任务丢给线程池，线程池创建线程并处理任务，如果将要运行的任务数大于线程池的基本线程数就把任务扔到阻塞队列里，这种做法比只使用一个阻塞队列来实现生产者和消费者模式显然要高明很多，因为消费者能够处理直接就处理掉了，这样速度更快，而生产者先存，消费者再取这种方式显然慢一些。

我们的系统也可以使用线程池来实现多生产者和消费者模式。例如，创建N个不同规模的Java线程池来处理不同性质的任务，比如线程池1将数据读到内存之后，交给线程池2里的线程继续处理压缩数据。线程池1主要处理IO密集型任务，线程池2主要处理CPU密集型任务。

本节讲解了生产者和消费者模式，并给出了实例。读者可以在平时的工作中思考一下哪些场景可以使用生产者消费者模式，我相信这种场景应该非常多，特别是需要处理任务时间比较长的场景，比如上传附件并处理，用户把文件上传到系统后，系统把文件丢到队列里，然后立刻返回告诉用户上传成功，最后消费者再去队列里取出文件处理。再如，调用一个远程接口查询数据，如果远程服务接口查询时需要几十秒的时间，那么它可以提供一个申请查询的接口，这个接口把要申请查询任务放数据库中，然后该接口立刻返回。然后服务器端用线程轮询并获取申请任务进行处理，处理完之后发消息给调用方，让调用方再来调用另外一个接口取数据。

11.2 线上问题定位

有时候，有很多问题只有在线上或者预发环境才能发现，而线上又不能调试代码，所以线上问题定位就只能看日志、系统状态和dump线程，本节只是简单地介绍一些常用的工具，以帮助大家定位线上问题。

1）在Linux命令行下使用TOP命令查看每个进程的情况，显示如下。

```
top - 22:27:25 up 463 days, 12:46, 1 user, load average: 11.80, 12.19, 11.79
 Tasks: 113 total, 5 running, 108 sleeping, 0 stopped, 0 zombie
 Cpu(s): 62.0%us, 2.8%sy, 0.0%ni, 34.3%id, 0.0%wa, 0.0%hi, 0.7%si, 0.2%st
 Mem: 7680000k total, 7665504k used, 14496k free, 97268k buffers
 Swap: 2096472k total, 14904k used, 2081568k free, 3033060k cached
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
 31177 admin 18 0 5351m 4.0g 49m S 301.4 54.0 935:02.08 java
 31738 admin 15 0 36432 12m 1052 S 8.7 0.2 11:21.05 nginx-proxy
```

我们的程序是Java应用，所以只需要关注COMMAND是Java的性能数据，COMMAND表示启动当前进程的命令，在Java进程这一行里可以看到CPU利用率是300%，不用担心，这个是当前机器所有核加在一起的CPU利用率。

2）再使用top的交互命令数字1查看每个CPU的性能数据。

```
top - 22:24:50 up 463 days, 12:43, 1 user, load average: 12.55, 12.27, 11.73
 Tasks: 110 total, 3 running, 107 sleeping, 0 stopped, 0 zombie
 Cpu0 : 72.4%us, 3.6%sy, 0.0%ni, 22.7%id, 0.0%wa, 0.0%hi, 0.7%si, 0.7%st
 Cpu1 : 58.7%us, 4.3%sy, 0.0%ni, 34.3%id, 0.0%wa, 0.0%hi, 2.3%si, 0.3%st
 Cpu2 : 53.3%us, 2.6%sy, 0.0%ni, 34.1%id, 0.0%wa, 0.0%hi, 9.6%si, 0.3%st
 Cpu3 : 52.7%us, 2.7%sy, 0.0%ni, 25.2%id, 0.0%wa, 0.0%hi, 19.5%si, 0.0%st
 Cpu4 : 59.5%us, 2.7%sy, 0.0%ni, 31.2%id, 0.0%wa, 0.0%hi, 6.6%si, 0.0%st
 Mem: 7680000k total, 7663152k used, 16848k free, 98068k buffers
 Swap: 2096472k total, 14904k used, 2081568k free, 3032636k cached
```

命令行显示了CPU4，说明这是一个5核的虚拟机，平均每个CPU利用率在60%以上。如果这里显示CPU利用率100%，则很有可能程序里写了一个死循环。这些参数的含义，可以对比表11-1来查看。

表11-1 CPU参数含义

参　数	描　述
us	用户空间占用 CPU 百分比
1.0% sy	内核空间占用 CPU 百分比
0.0% ni	用户进程空间内改变过优先级的进程占用 CPU 百分比
98.7% id	空闲 CPU 百分比
0.0% wa	等待输入 / 输出的 CPU 时间百分比

3）使用top的交互命令H查看每个线程的性能信息。

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
31558 admin     15   0 5351m 4.0g  49m S 12.2 54.0  10:08.31 java
31561 admin     15   0 5351m 4.0g  49m R 12.2 54.0   9:45.43 java
31626 admin     15   0 5351m 4.0g  49m S 11.9 54.0  13:50.21 java
31559 admin     15   0 5351m 4.0g  49m S 10.9 54.0   5:34.67 java
31612 admin     15   0 5351m 4.0g  49m S 10.6 54.0   8:42.77 java
31555 admin     15   0 5351m 4.0g  49m S 10.3 54.0  13:00.55 java
31630 admin     15   0 5351m 4.0g  49m R 10.3 54.0   4:00.75 java
31646 admin     15   0 5351m 4.0g  49m S 10.3 54.0   3:19.92 java
31653 admin     15   0 5351m 4.0g  49m S 10.3 54.0   8:52.90 java
31607 admin     15   0 5351m 4.0g  49m S  9.9 54.0  14:37.82 java
```

在这里可能会出现3种情况。

·第一种情况，某个线程CPU利用率一直100%，则说明是这个线程有可能有死循环，那么请记住这个PID。

·第二种情况，某个线程一直在TOP 10的位置，这说明这个线程可能有性能问题。

·第三种情况，CPU利用率高的几个线程在不停变化，说明并不是由某一个线程导致CPU偏高。

如果是第一种情况，也有可能是GC造成，可以用jstat命令看一下GC情况，看看是不是因为持久代或年老代满了，产生Full GC，导致CPU利用率持续飙高，命令和回显如下。

```
sudo /opt/java/bin/jstat -gcutil 31177 1000 5
 S0  S1  E  O  P  YGC  YGCT  FGC  FGCT  GCT
```

```
0.00 1.27 61.30 55.57 59.98 16040 143.775 30 77.692 221.467
0.00 1.27 95.77 55.57 59.98 16040 143.775 30 77.692 221.467
1.37 0.00 33.21 55.57 59.98 16041 143.781 30 77.692 221.474
1.37 0.00 74.96 55.57 59.98 16041 143.781 30 77.692 221.474
0.00 1.59 22.14 55.57 59.98 16042 143.789 30 77.692 221.481
```

还可以把线程dump下来，看看究竟是哪个线程、执行什么代码造成的CPU利用率高。执行以下命令，把线程dump到文件dump17里。执行如下命令。

```
sudo -u admin /opt/taobao/java/bin/jstack  31177 > /home/tengfei.fangtf/dump17
```

dump出来内容的类似下面内容。

```
"http-0.0.0.0-7001-97" daemon prio=10 tid=0x000000004f6a8000 nid=0x555e in Object.
wait() [0x0000000052423000]
    java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on  (a org.apache.tomcat.util.net.AprEndpoint$Worker)
        at java.lang.Object.wait(Object.java:485)
        at org.apache.tomcat.util.net.AprEndpoint$Worker.await(AprEndpoint.java:1464)
        - locked  (a org.apache.tomcat.util.net.AprEndpoint$Worker)
        at org.apache.tomcat.util.net.AprEndpoint$Worker.run(AprEndpoint.java:1489)
        at java.lang.Thread.run(Thread.java:662)
```

dump出来的线程ID(nid)是十六进制的，而我们用TOP命令看到的线程ID是十进制的，所以要用printf命令转换一下进制。然后用十六进制的ID去dump里找到对应的线程。

```
printf "%x\n" 31558
```

输出：7b46。

11.3 性能测试

因为要支持某个业务，有同事向我们提出需求，希望系统的某个接口能够支持2万的QPS，因为我们的应用部署在多台机器上，要支持两万的QPS，我们必须先要知道该接口在单机上能支持多少QPS，如果单机能支持1千QPS，我们需要20台机器才能支持2万的QPS。需要注意的是，要支持的2万的QPS必须是峰值，而不能是平均值，比如一天当中有23个小时QPS不足1万，只有一个小时的QPS达到了2万，我们的系统也要支持2万的QPS。

我们先进行性能测试。我们使用公司同事开发的性能测试工具进行测试，该工具的原理是，用户写一个Java程序向服务器端发起请求，这个工具会启动一个线程池来调度这些任务，可以配置同时启动多少个线程、发起请求次数和任务间隔时长。将这个程序部署在多台机器上执行，统计出QPS和响应时长。我们在10台机器上部署了这个测试程序，每台机器启动了100个线程进行测试，压测时长为半小时。注意不能压测线上机器，我们压测的是开发服务器。

测试开始后，首先登录到服务器里查看当前有多少台机器在压测服务器，因为程序的端口是12200，所以使用netstat命令查询有多少台机器连接到这个端口上。命令如下。

```
$ netstat -nat | grep 12200 -c
10
```

通过这个命令可以知道已经有10台机器在压测服务器。QPS达到了1400，程序开始报错获取不到数据库连接，因为我们的数据库端口是3306，用netstat命令查看已经使用了多少个数据库连接。命令如下。

```
$ netstat -nat | grep 3306 -c
12
```

增加数据库连接到20，QPS没上去，但是响应时长从平均1000毫秒下降到700毫秒，使用TOP命令观察CPU利用率，发现已经90%多了，于是升级CPU，将2核升级成4核，和线上的机器

保持一致。再进行压测，CPU利用率下去了达到了75%，QPS上升到了1800。执行一段时间后响应时长稳定在200毫秒。

增加应用服务器里线程池的核心线程数和最大线程数到1024，通过ps命令查看下线程数是否增长了，执行的命令如下。

```
$ ps -eLf | grep java -c
1520
```

再次压测，QPS并没有明显的增长，单机QPS稳定在1800左右，响应时长稳定在200毫秒。

我在性能测试之前先优化了程序的SQL语句。使用了如下命令统计执行最慢的SQL，左边的是执行时长，单位是毫秒，右边的是执行的语句，可以看到系统执行最慢的SQL是queryNews和queryNewIds，优化到几十毫秒。

```
$ grep Y /home/admin/logs/xxx/monitor/dal-rw-monitor.log |awk -F',' '{print $7$5}' |
sort -nr|head -20
1811 queryNews
1764 queryNews
1740 queryNews
1697 queryNews
679 queryNewIds
```

性能测试中使用的其他命令

1）查看网络流量。

```
$ cat /proc/net/dev
Inter-| Receive | Transmit
face |bytes packets errs drop fifo frame compressed multicast|bytes packets
errs drop fifo colls carrier compressed
lo:242953548208 231437133 0 0 0 0 0 0 242953548208 231437133 0 0 0 0 0 0
eth0:153060432504 446365779 0 0 0 0 0 0 108596061848 479947142 0 0 0 0 0 0
bond0:153060432504 446365779 0 0 0 0 0 0 108596061848 479947142 0 0 0 0 0 0
```

2）查看系统平均负载。

```
$ cat /proc/loadavg
0.00 0.04 0.85 1/1266 22459
```

3）查看系统内存情况。

```
$ cat /proc/meminfo
MemTotal: 4106756 kB
MemFree: 71196 kB
Buffers: 12832 kB
Cached: 2603332 kB
SwapCached: 4016 kB
Active: 2303768 kB
Inactive: 1507324 kB
Active(anon): 996100 kB
部分省略
```

4）查看CPU的利用率。

```
cat /proc/stat
cpu 167301886 6156 331902067 17552830039 8645275 13082 1044952 33931469 0
cpu0 45406479 1992 75489851 4410199442 7321828 12872 688837 5115394 0
cpu1 39821071 1247 132648851 4319596686 379255 67 132447 11365141 0
cpu2 40912727 1705 57947971 4418978718 389539 78 110994 8342835 0
cpu3 41161608 1211 65815393 4404055191 554651 63 112672 9108097 0
```

11.4 异步任务池

Java中的线程池设计得非常巧妙，可以高效并发执行多个任务，但是在某些场景下需要对线程池进行扩展才能更好地服务于系统。例如，如果一个任务仍进线程池之后，运行线程池的程序重启了，那么线程池里的任务就会丢失。另外，线程池只能处理本机的任务，在集群环境下不能有效地调度所有机器的任务。所以，需要结合线程池开发一个异步任务处理池。图11-2为异步任务池设计图。

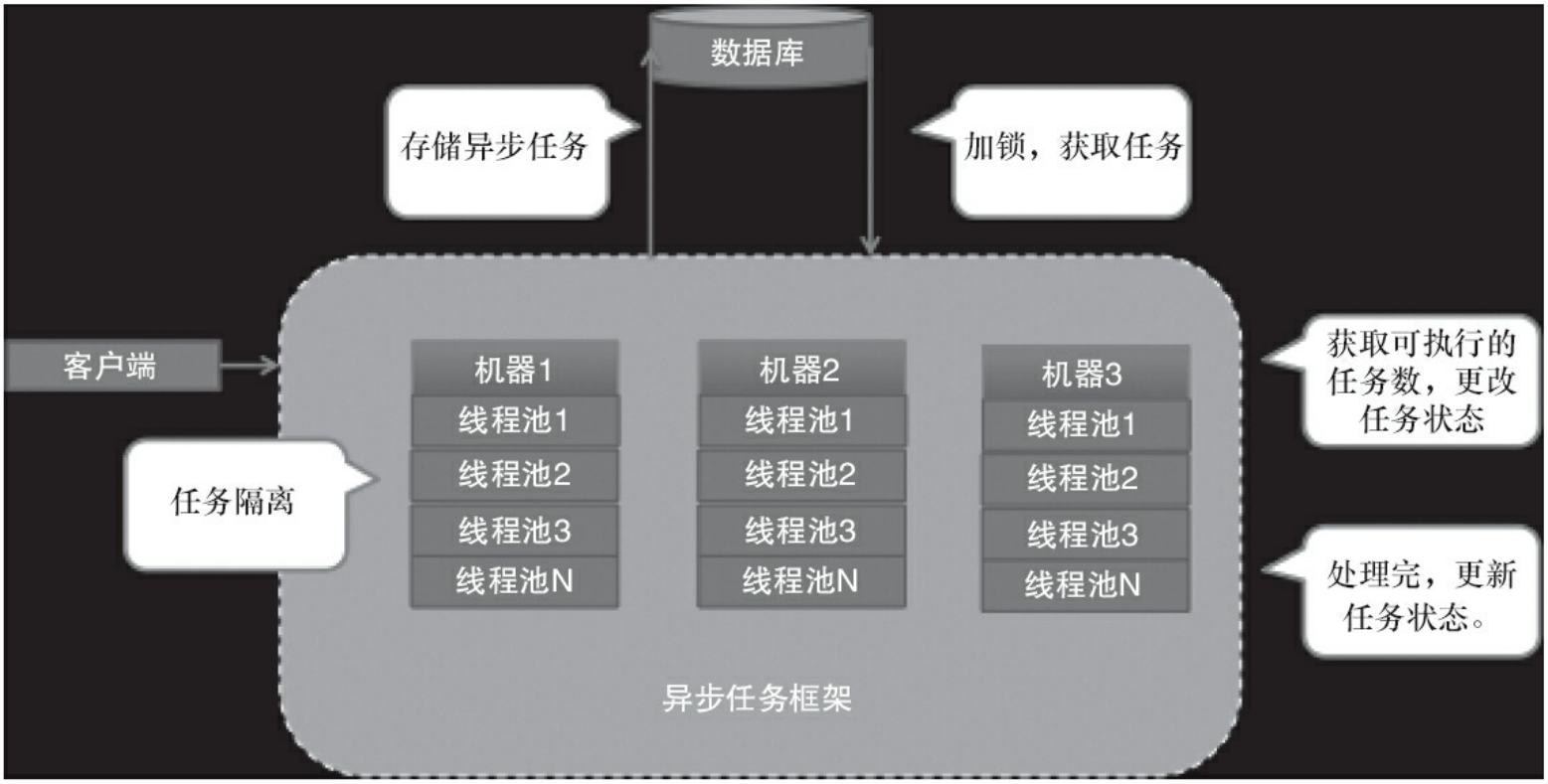

图11-2 异步任务池设计图

任务池的主要处理流程是，每台机器会启动一个任务池，每个任务池里有多个线程池，当某台机器将一个任务交给任务池后，任务池会先将这个任务保存到数据中，然后某台机器上的任务池会从数据库中获取待执行的任务，再执行这个任务。

每个任务有几种状态，分别是创建（NEW）、执行中（EXECUTING）、RETRY（重试）、挂起（SUSPEND）、中止（TEMINER）和执行完成（FINISH）。

·创建：提交给任务池之后的状态。

·执行中：任务池从数据库中拿到任务执行时的状态。

·重试：当执行任务时出现错误，程序显式地告诉任务池这个任务需要重试，并设置下一次执行时间。

·挂起：当一个任务的执行依赖于其他任务完成时，可以将这个任务挂起，当收到消息后，再开始执行。

·中止：任务执行失败，让任务池停止执行这个任务，并设置错误消息告诉调用端。

·执行完成：任务执行结束。

任务池的任务隔离。异步任务有很多种类型，比如抓取网页任务、同步数据任务等，不同类型的任务优先级不一样，但是系统资源是有限的，如果低优先级的任务非常多，高优先级的任务就可能得不到执行，所以必须对任务进行隔离执行。使用不同的线程池处理不同的任务，或者不同的线程池处理不同优先级的任务，如果任务类型非常少，建议用任务类型来隔离，如果任务类型非常多，比如几十个，建议采用优先级的方式来隔离。

任务池的重试策略。根据不同的任务类型设置不同的重试策略，有的任务对实时性要求高，那么每次的重试间隔就会非常短，如果对实时性要求不高，可以采用默认的重试策略，重试间隔随着次数的增加，时间不断增长，比如间隔几秒、几分钟到几小时。每个任务类型可以设置执行该任务类型线程池的最小和最大线程数、最大重试次数。

使用任务池的注意事项。任务必须无状态：任务不能在执行任务的机器中保存数据，比如某个任务是处理上传的文件，任务的属性里有文件的上传路径，如果文件上传到机器1，机器2获取到了任务则会处理失败，所以上传的文件必须存在其他的集群里，比如OSS或SFTP。

异步任务的属性。包括任务名称、下次执行时间、已执行次数、任务类型、任务优先级和

执行时的报错信息(用于快速定位问题)。

11.5 本章小结

本章介绍了使用生产者和消费者模式进行并发编程、线上问题排查手段和性能测试实战，以及异步任务池的设计。并发编程的实战需要大家平时多使用和测试，才能在项目中发挥作用。