Tomcat线程池

Tomcat

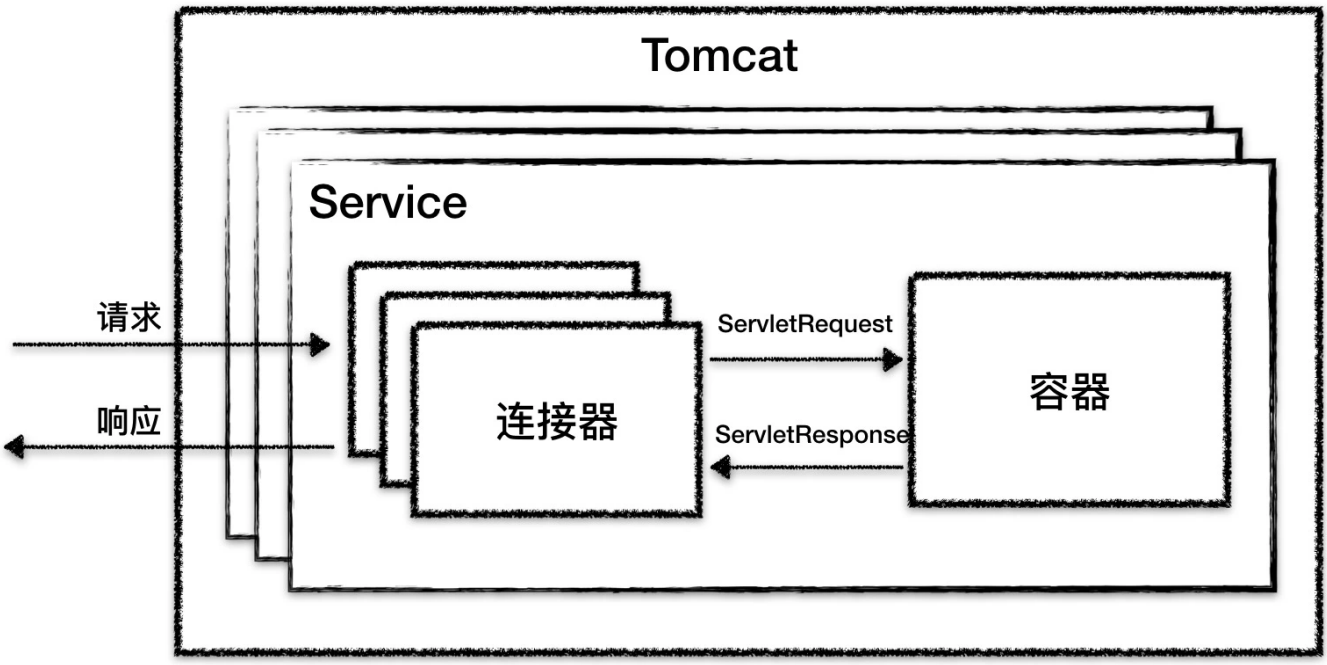
> Tomcat是一个Web容器

一个Web容器的主要功能：

- 处理Socket连接 负责网络字节流和`Request` `Response`对象之间的转化
- 管理Servlet的生命周期，通过Servlet处理`Request`请求

Tomcat的两大组成部分：

- 连接器 *Connector* （作为HTTP服务器）
- 容器 *Container* （作为Servlet容器）



对象池

应用场景；

某种对象数量多 自身占用空间多 创建和初始化耗时 但存在时间短

那么这些对象的频繁的创建 初始化 GC都会消耗大量的CPU和内存资源

实现方法：

把一个对象用完之后将它保存起来，需要再用时再拿出来（清空之前的信息 如果有必要）

进行重复使用（空间换时间）

JDK线程池

`java.util.concurrent.ThreadPoolExecutor`

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

构造参数

- 核心线程数：
- 最大线程数 = 核心线程数 + 临时线程数

> 核心线程：创建后一直存活的线程数（除非`allowCoreThreadTimeOut`参数为true）
>
> 临时线程：当一定的时间内从工作队列中获取不到Runnable对象，该线程销毁
>
> 核心线程和临时线程的区别方式：从工作队列中拉取Runnable对象的方式不同（受当前线程数与核心线程数的影响）
>
> ```
> // 尝试从工作了队列中获取任务
> // 在Worker类中的runWorker方法中的循环中被调用
> // 当该方法返回null时会退出循环
> private Runnable getTask() {
>      boolean timedOut = false; // Did the last poll() time out?
>      for (;;) {
>          int c = ctl.get();
>          int wc = workerCountOf(c);
>          // Are workers subject to culling?
> ```

```
            boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
            if ((wc > maximumPoolSize || (timed && timedOut))
                && (wc > 1 || workQueue.isEmpty())) {
                if (compareAndDecrementWorkerCount(c))
                    return null;
                continue;
            }
            try {
               // 核心线程和临时线程的区别就在这里
                Runnable r = timed ?
                    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                    workQueue.take();
                if (r != null)
                    return r;
                timedOut = true;
            } catch (InterruptedException retry) {
                timedOut = false;
            }
        }
    }
```

- 临时线程最大空闲时间

  两个参数都是用于设定临时线程从工作队列中尝试获取Runnable对象的时间

  `workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)`
- 工作队列：阻塞队列
- 线程工厂
- 拒绝策略：回调函数

执行流程

1. 前 corePoolSize 个任务时，来一个任务就创建一个新线程。
2. 后面再来任务，就把任务添加到任务队列里让所有的线程去抢，如果**加入队列失败**就创建临时线程。

   > 这里加入队列失败的定义为队列已满

3. 如果当前线程数达到 maximumPoolSize，执行拒绝策略。

```
// 向线程池中提交一个任务
public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();
        // 获取线程控制状态：int(32) = 3(状态位) + 29(线程数)
        int c = ctl.get();
        // 当前线程数小于核心线程数：
        if (workerCountOf(c) < corePoolSize) {
```

```
            // 创建一个新的线程并在一个循环不断尝试获取任务并执行
            // 任务来源：1.创建线程时赋予的 2.从工作队列中拉取的
            if (addWorker(command, true))
                return;
            c = ctl.get();
        }
        // 当前线程数以达到核心线程数：将任务放到工作队列中
        if (isRunning(c) && workQueue.offer(command)) {
            int recheck = ctl.get();
            if (! isRunning(recheck) && remove(command))
                reject(command);
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        // 放入工作队列失败（offer方法返回false）：尝试创建一个临时线程执行它
        else if (!addWorker(command, false))
            // 当前线程数已达到最大线程数：执行拒绝策略
            reject(command);
}

// 线程控制状态 初始int值为111加上29个0
AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));

int CAPACITY = (1 << 29) - 1 // 也就是000加上29个1

// 获取线程池状态
private static int runStateOf(int c)     { return c & ~CAPACITY; }

// 获取线程池的线程数量
private static int workerCountOf(int c)  { return c & CAPACITY; }
```

```
// 创建一个新的线程
private boolean addWorker(Runnable firstTask, boolean core) {
        retry:
        for (;;) {
            int c = ctl.get();
            for (;;) {
                int wc = workerCountOf(c);
                // core为true则用核心线程数作为上限判断 为false则用最大线程数作为上限
                if (wc >= CAPACITY ||
                    wc >= (core ? corePoolSize : maximumPoolSize))
                    return false;
                if (compareAndIncrementWorkerCount(c))
                    break retry;
            }
        }
        boolean workerStarted = false;
```

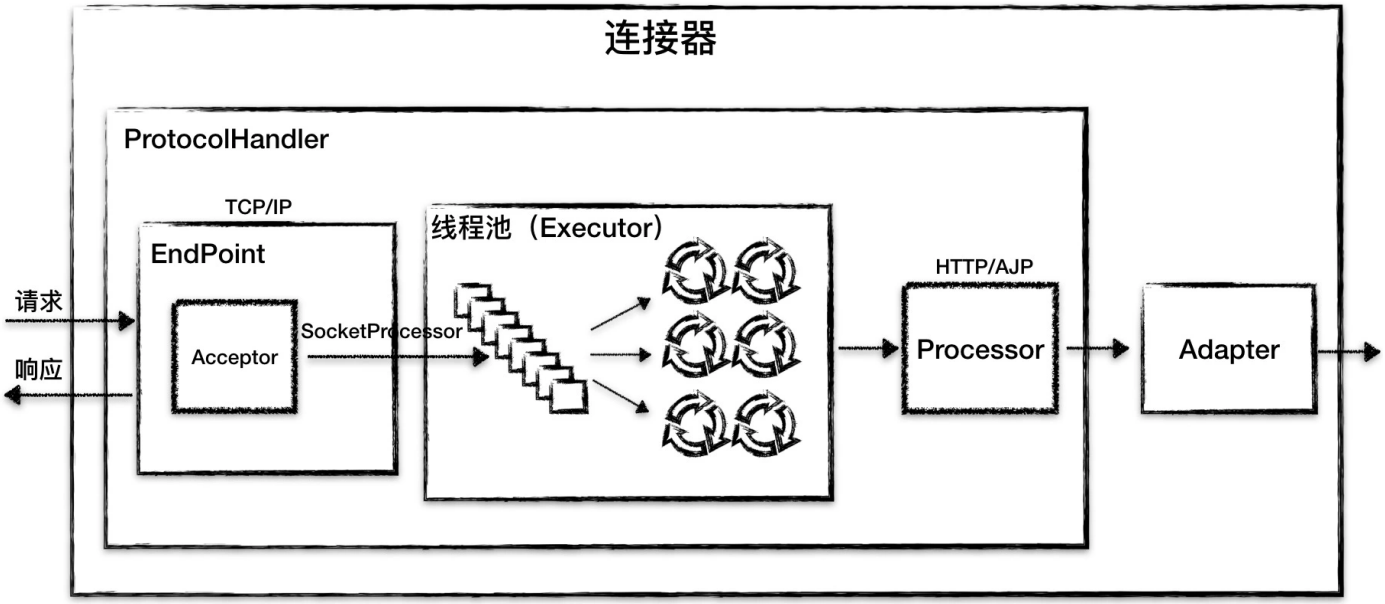
```
        boolean workerAdded = false;
        // Worker构造方法里面创建了线程 并将Worker作为Runnable传给该线程
        // class Worker extends AbstractQueuedSynchronizer implements Runnable
        Worker w = new Worker(firstTask);
        final Thread t = w.thread;
        if (t != null) {
          // 加锁保障线程HashSet<Worker> workers的线程安全
          final ReentrantLock mainLock = this.mainLock;
          mainLock.lock();
          try {
            workers.add(w);
            workerAdded = true;
          }
        } finally {
          mainLock.unlock();
        }
        if (workerAdded) {
          // t.start()会调用Thread的Runnable对象的run方法 即Worker对象的Run方法
          t.start();
          workerStarted = true;
        }
      }
      return workerStarted;
}

// Worker构造方法
Worker(Runnable firstTask) {
  setState(-1); // inhibit interrupts until runWorker
  this.firstTask = firstTask;
  // 将Worker作为Runnable传给该线程
  this.thread = getThreadFactory().newThread(this);
}

// Worker对象的Run方法会调用该方法
final void runWorker(Worker w) {
        Thread wt = Thread.currentThread();
        // 任务来源1：创建Worker对象时赋予
        Runnable task = w.firstTask;
        w.firstTask = null;
        boolean completedAbruptly = true;
        try {
            // 循环中不断尝试获取任务并执行
            // 任务来源2：从getTask()中获取
            while (task != null || (task = getTask()) != null) {
                try {
                    task.run();
                } finally {
                    task = null;
                }
```

```
            }
            completedAbruptly = false;
        } finally {
            // 退出循环后（即getTask()返回null）清除Worker对象
            processWorkerExit(w, completedAbruptly);
        }
}
```

Tomcat线程池

> 在JDK线程池的基础上进行了一定的扩展

Tomcat线程池跟我们业务开发的关系



1. Endpoint 接收到 Socket 连接后，生成一个 SocketProcessor 任务提交到线程池去处理
2. SocketProcessor 的 run 方法会调用 Processor 组件去解析应用层协议
3. Processor 通过解析生成 Request 对象后，会调用 Adapter 的 Service 方法
4. Adapter中会调用容器 最终会调用Servlet的Service方法（SpringMVC实现了Servlet接口）

```
// 默认参数
// 定制的任务队列 TaskQueue extends LinkedBlockingQueue<Runnable>
taskqueue = new TaskQueue(Integer.MAX_VALUE);

// 定制的线程工厂 TaskThreadFactory implements ThreadFactory
TaskThreadFactory tf = new TaskThreadFactory(namePrefix,daemon,getThreadPriority());

// 定制的线程池 ThreadPoolExecutor extends java.util.concurrent.ThreadPoolExecutor
executor = new ThreadPoolExecutor(25, 200, 60000, TimeUnit.MILLISECONDS,taskqueue, tf);
```

Tomcat线程池的执行流程：

1. 前 corePoolSize 个任务时，来一个任务就创建一个新线程。
2. 再来任务的话，就把任务添加到任务队列里让所有的线程去抢，如果**加入队列失败**就创建临时线程。

   > 这里加入队列失败的定义: 即`workQueue.offer()`方法返回false的逻辑
   >
   > - 队列已满 `size == capacity`
   > - 已提交但未完成的任务数 > 当前线程数（也就是说有任务没有分配到线程进行执行 需要线程）

3. 如果总线程数达到 maximumPoolSize，

   则一段时间后 继续尝试把任务添加到任务队列中去

   如果一段时间后 任务队列还是满的，插入失败，执行拒绝策略

```
public class ThreadPoolExecutor extends java.util.concurrent.ThreadPoolExecutor {

  // 已经提交但是还没有执行的任务数量
  private final AtomicInteger submittedCount = new AtomicInteger(0);

  public void execute(Runnable command, long timeout, TimeUnit unit) {
      submittedCount.incrementAndGet();
      try {
          //调用Java原生线程池的execute去执行任务
          super.execute(command);
      } catch (RejectedExecutionException rx) {
         // 工作队列已满且总线程数达到maximumPoolSize 默认的拒绝策略被执行(即抛出异常)
          if (super.getQueue() instanceof TaskQueue) {
              final TaskQueue queue = (TaskQueue)super.getQueue();
              try {
                  // 等待一段时间后 再把任务放到任务队列中去
                  // 因为可能在等待的这段时间中 任务队列有位置空了出来
                  if (!queue.force(command, timeout, unit)) {
                      submittedCount.decrementAndGet();
                      // 一段时间后 队列仍然是满的，插入失败，则执行拒绝策略。
                      throw new RejectedExecutionException("...");
```

```
                    }
                }
            }
        }
}
```

```
public class TaskQueue extends LinkedBlockingQueue<Runnable> {

  private transient volatile ThreadPoolExecutor parent;


   @Override
  // 线程池调用offer方法时，当前线程数肯定已经大于核心线程数了
  // 接下来判断：是将任务放到工作队列中还是创建新的线程执行该任务
  public boolean offer(Runnable o) {

      //如果线程数已经到了最大值，不能创建新线程了，只能把任务添加到任务队列。
      if (parent.getPoolSize() == parent.getMaximumPoolSize())
          return super.offer(o);

      //执行到这里，表明当前线程数大于核心线程数，并且小于最大线程数。
      //表明是可以创建新线程的，那到底要不要创建呢？分两种情况：

      //1. 如果已提交的任务数小于当前线程数，表示还有空闲线程，无需创建新线程
      if (parent.getSubmittedCount()<=(parent.getPoolSize()))
          return super.offer(o);

      //2. 如果已提交的任务数大于当前线程数，线程不够用了，返回false去创建新线程
      if (parent.getPoolSize()<parent.getMaximumPoolSize())
          return false;

      //默认情况下总是把任务添加到任务队列
      return super.offer(o);
  }

}
```