JDK19协程（VirtualThread）

传统WEB编程模型

- WEB服务处理请求都会分配一个线程(thread per request)
- 使用独占的方式来处理该请求
- 易于理解和编程实现
- 易于调式和上下文追踪ThreadLocal

处理海量请求创建过多线程，系统资源占用高，频繁上下文切换，创建一个平台线程需要1M内存资源，1000个线程就需要1G内存，但是线程都在阻塞，cpu利用率低。cpu只能同时运行核心数线程，其他线程都在阻塞或等待调度，线程过多会频繁切换调度，上下文频繁切换。

总的来说，平台线程有下面的一些特点或者说限制：

- 资源有限导致系统线程总量有限，进而导致与系统线程一一对应的平台线程有限
- 平台线程的调度依赖于系统的线程调度程序，当平台线程创建过多，会消耗大量资源用于处理线程上下文切换
- 每个平台线程都会开辟一块私有的栈空间，大量平台线程会占据大量内存

目前的解决思路

- 依赖非阻塞IO和异步编程，可以使用少量线程处理大量请求。回调函数过多，每一层的回调函数都需要依赖上一层的回调执行完，所以形成了层层嵌套的关系。
- 开发人员必须熟悉所使用的底层框架。链路追踪难。
- 由于之前Java回调都是需要传一个接口类，需要创建大量匿名类，而且代码看起来不好阅读。
- 在Java8出现Lambda后，简化的回调类，Reactor针对异步编程实现了封装，简化了异步开发。

虚拟线程

JDK 19 中的新特性官方名称为 Virtual Threads，是 JEP425 提案里的概念，属于 Loom 项目，直译过来应该是虚拟线程，是协程这个概念在 JDK 19 中的官方名称。

虚拟线程

虚拟线程（Virtual Thread）是一种轻量级、按照顺序并发执行的线程，它由Java虚拟机（JVM）创建、调度和控制，而非由操作系统真正创建并调度。与操作系统线程相比，虚拟线程的创建和销毁运行费用较低，且不需要太多的上下文切换和内存开销。

在Java中，虚拟线程实现于Fiber类和Continuation类中。Fiber提供了一个像协程一样的线程，但具有更好的控制力和更好的遍历支持；而Continuation提供了一种可以中断和恢复的方法，可以无需使用阻塞和非阻塞代码的情况下中断一个线程，并在需要时恢复该线程。

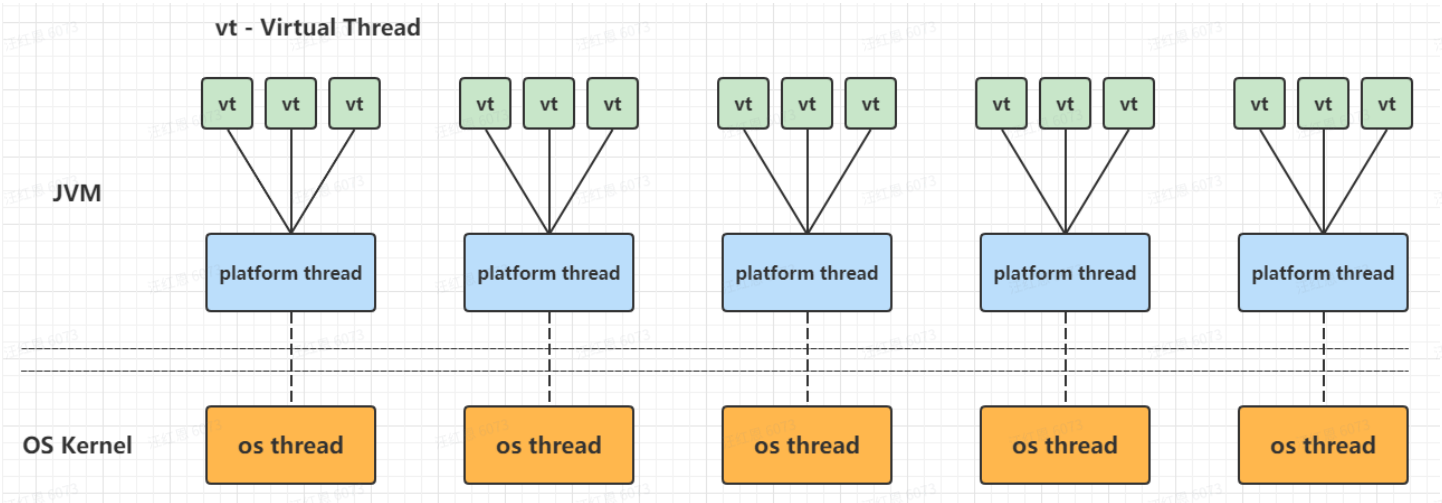
虚拟线程有以下优点：

1. 更轻量级：与操作系统线程相比，虚拟线程的创建和销毁运行费用较低，而且不会占用太多内存。
2. 更高效：虚拟线程的上下文切换开销小，因为它是在用户空间中实现的。
3. 更好的控制力：虚拟线程可以使用Fiber提供的控制来更好地控制线程的执行，防止出现竞争条件和死锁等问题。
4. 更好的遍历和调试支持：虚拟线程可以使用Continuation提供的中断和恢复机制来有效地调试和检查线程上下文。

JDK19分为平台线程（PlatformThread）和虚拟线程(VirtualThread)

平台线程就是之前的创建线程方式，平台线程和操作系统的内核线程是一一对应的

平台线程和虚拟线程都是使用Thread类来表示，VirtualThread是Thread的子类。只需要非常少的改动使用虚拟线程，虚拟线程支持ThreadLocal。现有的代码无需改动。



使用虚拟线程

```java
public class Main {
    no usages
    public static void main(String[] args) throws InterruptedException {
        Thread thread = Thread.ofVirtual().name("Virtual")
                .start(() -> {
                    System.out.println("运行中: "+ Thread.currentThread());
                    try {
                        Thread.sleep( millis: 100);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                    System.out.println("结束: "+ Thread.currentThread());
                });
        ExecutorService executorService = Executors.newVirtualThreadPerTaskExecutor();
        IntStream.range(0,100).forEach(i -> executorService.submit(() -> i));
        thread.join();
    }
```

```
Run
/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/bin/java --enable-preview -javaager
运行中: VirtualThread[#22,Virtual]/runnable@ForkJoinPool-1-worker-1
结束: VirtualThread[#22,Virtual]/runnable@ForkJoinPool-1-worker-8
```

和之前同步编写代码一样，只是虚拟线程替换了平台线程。降低了编写高并发应用难度。

Thread.*sleep()*

```java
    status of the current thread is cleared when this exception is thrown.
@   public static void sleep( @Range(from = 0, to = java.lang.Long.MAX_VALUE) long millis) throws InterruptedException {
        if (millis < 0) {
            throw new IllegalArgumentException("timeout value is negative");
        }

        if (currentThread() instanceof VirtualThread vthread) {
            long nanos = MILLISECONDS.toNanos(millis);
            vthread.sleepNanos(nanos);
            return;
        }

        if (ThreadSleepEvent.isTurnedOn()) {
            ThreadSleepEvent event = new ThreadSleepEvent();
            try {
                event.time = MILLISECONDS.toNanos(millis);
                event.begin();
                sleep0(millis);
            } finally {
                event.commit();
            }
```

```java
765    private void doSleepNanos(long nanos) throws InterruptedException {
766        assert nanos >= 0;
767        if (getAndClearInterrupt())
768            throw new InterruptedException();
769        if (nanos == 0) {
770            tryYield();
771        } else {
772            // park for the sleep time
773            try {
774                long remainingNanos = nanos;
775                long startNanos = System.nanoTime();
776                while (remainingNanos > 0) {
777                    parkNanos(remainingNanos);
778                    if (getAndClearInterrupt()) {
779                        throw new InterruptedException();
780                    }
781                    remainingNanos = nanos - (System.nanoTime() - startNanos);
782                }
783            } finally {
784                // may have been unparked while sleeping
785                setParkPermit(true);
```

```
    Params: nanos – the maximum number of nanoseconds to wait.

@Override
void parkNanos(long nanos) {
    assert Thread.currentThread() == this;

    // complete immediately if parking permit available or interrupted
    if (getAndSetParkPermit( newValue: false) || interrupted)
        return;

    // park the thread for the waiting time
    if (nanos > 0) {
        long startTime = System.nanoTime();

        boolean yielded;
        Future<?> unparker = scheduleUnpark(this::unpark, nanos);
        setState(PARKING);
        try {
            yielded = yieldContinuation();
        } finally {
            assert (Thread.currentThread() == this)
                    && (state() == RUNNING || state() == PARKING);
            cancel(unparker);
        }

        // park on carrier thread for remaining time when pinned
        if (!yielded) {
            long deadline = startTime + nanos;
            if (deadline < 0L)
                deadline = Long.MAX_VALUE;
            parkOnCarrierThread( timed: true,  nanos: deadline - System.nanoTime());
        }
    }
}
```

```java
@ChangesCurrentThread
private boolean yieldContinuation() {
    boolean notifyJvmti = notifyJvmtiEvents;

    // unmount
    if (notifyJvmti) notifyJvmtiUnmountBegin( lastUnmount: false);
    unmount();
    try {
        return Continuation.yield(VTHREAD_SCOPE);
    } finally {
        // re-mount
        mount();
        if (notifyJvmti) notifyJvmtiMountEnd( firstMount: false);
    }
}
```

```java
1    private void submitRunContinuation(boolean lazySubmit) {
2         ...
3         scheduler.execute(runContinuation);
4         ...
5     }
6
7     private void runContinuation() {
8         ...
9         cont.run();
10        ...
11    }
```

源码分析

JDK协程的核心实现就是 **Continuation + Scheduler + 阻塞操作的改造**

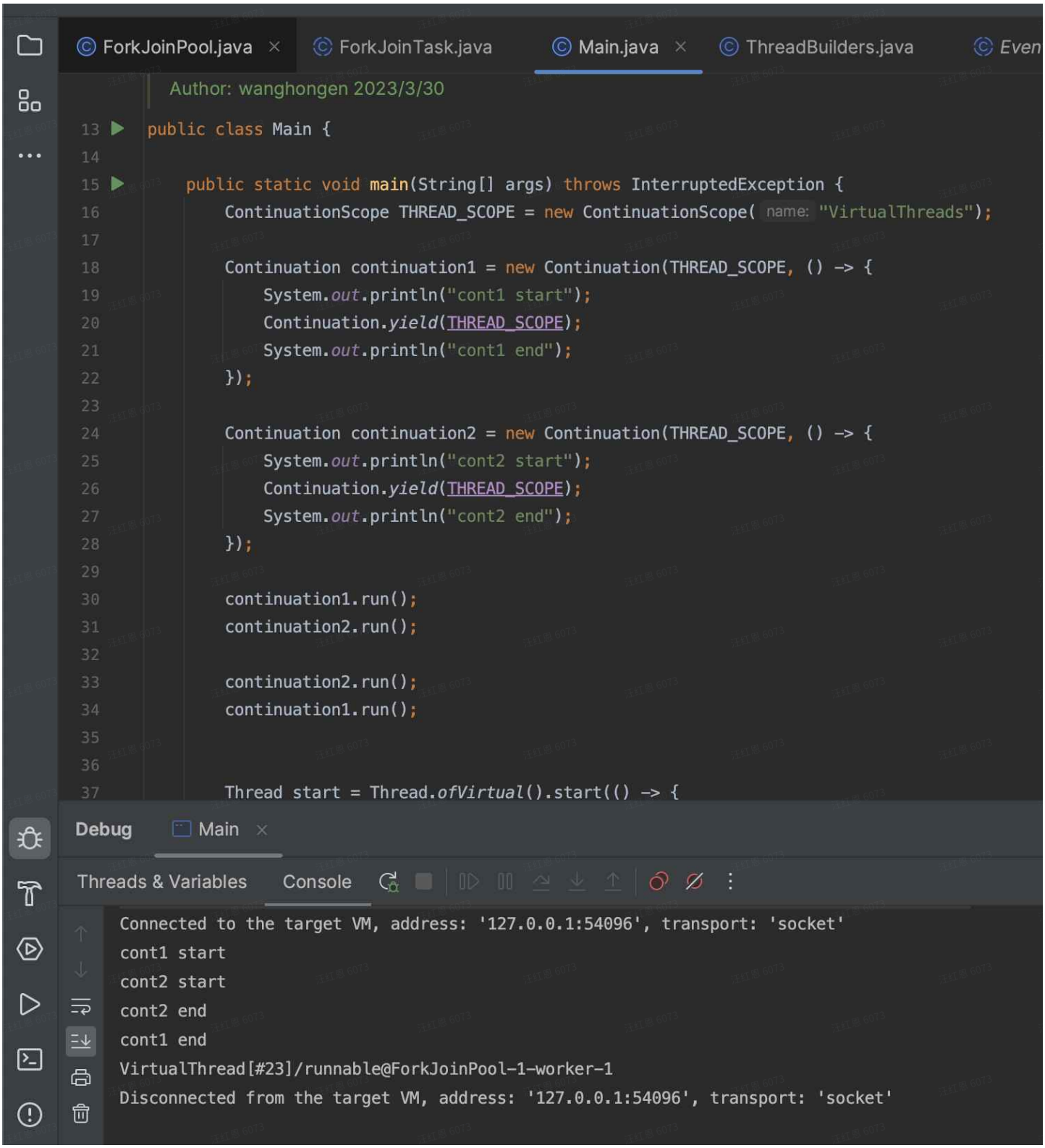
```java
1
2  final class VirtualThread extends BaseVirtualThread {
3     private final Executor scheduler;
4     private final Continuation cont;
5     private final Runnable runContinuation;
6
7     VirtualThread(Executor scheduler, String name,  Runnable task) {
8         this.scheduler = scheduler;
```

```
        this.cont = new VThreadContinuation(this, task);
        this.runContinuation = this::runContinuation;
    }

    @Override
    public void start() {
        start(ThreadContainers.root());
    }

    @Override
    void start(ThreadContainer container) {
        submitRunContinuation();
    }

    private void submitRunContinuation(boolean lazySubmit) {
        scheduler.execute(runContinuation);
    }

    private void runContinuation() {
        try {
           cont.run();
        } finally {
           cont.isDone();
        }
    }

}
```

虚拟线程的start()方法并不会创建OS线程, 虚拟线程会把任务包装到一个 `Continuation` 实例中，然后提交到调度器Scheduler里去执行，Scheduler默认就是使用ForkJoinPool，线程数就是cpu核心数。

Continuation

调用yield方法就会挂起，再次调用run方法就可以从yield的调用处往下执行，从而实现了程序的中断和恢复




```java
// Continuation.run()
public final void run() {
    while (true) {
        // 获取当前虚拟线程分配的运载线程
        Thread t = currentCarrierThread();
        if (parent != null) {
            if (parent != JLA.getContinuation(t))
                throw new IllegalStateException();
        } else
```

```
            this.parent = JLA.getContinuation(t);

        JLA.setContinuation(t, this);

        try {
            // 判断ContinuationScope是否虚拟线程范围
            boolean isVirtualThread = (scope == JLA.virtualThreadContinuationSco
            if (!isStarted()) { // is this the first run? (at this point we know
                // 第一次调用
                enterSpecial(this, false, isVirtualThread);
            } else {
                assert !isEmpty();
                enterSpecial(this, true, isVirtualThread);
            }
        } finally {
            // 设置内存屏障
            fence();
            try {
                // 进行后置的yield清理工作
                postYieldCleanup();
                // 进行unmount操作
                unmount();
            } catch (Throwable e) { e.printStackTrace(); System.exit(1); }
        }
    }
}

// Continuation.enter()系列方法

// 这是一个native方法，它最终会根据判断回调到enter()方法
private native static void enterSpecial(Continuation c, boolean isContinue, bool

// Continuation的入口方法，用户任务回调的入口
@DontInline
@IntrinsicCandidate
private static void enter(Continuation c, boolean isContinue) {
    // This method runs in the "entry frame".
    // A yield jumps to this method's caller as if returning from this method.
    try {
        c.enter0();
    } finally {
        c.finish();
    }
}

// 真正任务包装器执行的回调方法
private void enter0() {
```

```
57      target.run();
58  }
59
60  // Continuation完成，标记done为true
61  private void finish() {
62      done = true;
63      assert isEmpty();
64  }
65
66
67  // Continuation.yield()方法，静态方法
68  public static boolean yield(ContinuationScope scope) {
69      // 获取当前运载线程的Continuation实例
70      Continuation cont = JLA.getContinuation(currentCarrierThread());
71      return cont.yield0(scope, null);
72  }
73
74  private boolean yield0(ContinuationScope scope, Continuation child) {
75      // 强制抢占式卸载标记为false
76      preempted = false;
77      // 最终的yield调用，最终当前Continuation就是阻塞在此方法，从下文源码猜测，当该方法唤
78      int res = doYield();
79      // 放置内存屏障防止指令重排，后面注释提到是防止编译器进行某些转换
80      U.storeFence(); // needed to prevent certain transformations by the compiler
81
82      // 返回布尔值结果表示当前Continuation实例是否会继续执行
83      return res == 0;
84  }
85
86  // 最终的yield调用，看实现是抛出异常，猜测是由JVM实现
87  @IntrinsicCandidate
88  private static int doYield() { throw new Error("Intrinsic not installed"); }
```

SocketChannel.read() 的改造

我们看一下 SocketChannel 类的 read 方法源码，这里省去了很多非关键的细节。

同还没有引入协程的 JDK18 的源码相比，这里很明显插入了一行 configureSocketNonBlockingIfVirtualThread() 方法，根据名字看，该方法就是协程改造的关键。

```
1  class SocketChannelImpl extends SocketChannel implements SelChImpl {
2
3      @Override
4      public int read(ByteBuffer buf) throws IOException {
```

```
        ...
        boolean blocking = isBlocking();
        ...
        configureSocketNonBlockingIfVirtualThread();
        n = IOUtil.write(fd, buf, -1, nd);
        if (blocking) {
            ...
            park(Net.POLLOUT);
            ...
        }
        ...
    }

    /**
     * 确保socket在虚拟线程上配置为非阻塞。
     */
    private void configureSocketNonBlockingIfVirtualThread() throws IOException
        assert readLock.isHeldByCurrentThread() || writeLock.isHeldByCurrentThre
        if (!forcedNonBlocking && Thread.currentThread().isVirtual()) {
            synchronized (stateLock) {
                ensureOpen();
                IOUtil.configureBlocking(fd, false); //虚拟线程直把scoket改成了非阻
                forcedNonBlocking = true;
            }
        }
    }

}


//park(Net.POLLOUT);

public interface SelChImpl extends Channel {

    default void park(int event) throws IOException {
       park(event, 0L);
    }

    default void park(int event, long nanos) throws IOException {
        if (Thread.currentThread().isVirtual()) {
            Poller.poll(getFDVal(), event, nanos, this::isOpen);
        } else {
            long millis;
            ...
            Net.poll(getFD(), event, millis);
        }
    }
```

```
}
```

这里是 IO 异步的关键，即使用多路复用的方式来监听文件描述符的回调函数。在 macos 系统中，这里的最终实现是 kqueue，而在 Linux 系统中，这里的实现就是 epoll。

```

public abstract class Poller {

    private void poll1(int fdVal, long nanos, BooleanSupplier supplier) throws I
        register(fdVal);
        try {
            boolean isOpen = supplier.getAsBoolean();
            if (isOpen) {
                if (nanos > 0) {
                    LockSupport.parkNanos(nanos);
                } else {
                    LockSupport.park();
                }
            }
        } finally {
            deregister(fdVal);
        }
    }
    static native int register(int kqfd, int fd, int filter, int flags);

    private void register(int fdVal) throws IOException {
        map.putIfAbsent(fdVal, Thread.currentThread());
        ...
        implRegister(fdVal);
    }
}

public class LockSupport {
    public static void park() {
        if (Thread.currentThread().isVirtual()) {
            VirtualThreads.park();
        } else {
            U.park(false, 0L);
        }
    }
}

final class VirtualThread extends BaseVirtualThread {
    void park() {
        // park the thread
```

```
41          setState(PARKING);
42          try {
43              if (!yieldContinuation()) {
44                  // park on the carrier thread when pinned
45                  parkOnCarrierThread(false, 0);
46              }
47          } finally {
48              assert (Thread.currentThread() == this) && (state() == RUNNING);
49          }
50      }
51  }
52
```

写事件到达时唤醒协程

上面在执行读操作时，通过 park(Net.POLLOUT)，最终调用到 Continuation.yield 使协程发生阻塞，由调度器线程切换到其他协程执行。那么，当写事件到达时，应该是调用 Continuation.run 方法唤醒协程，我们看一下这部分逻辑。

Poller 类会启动一个线程，循环监听读事件。从之前 fd 对应 thread 的 map 中取出 thread，调用 LockSupport.unpark 方法。

```
1  private Poller start() {
2      String prefix = (read) ? "Read" : "Write";
3      startThread(prefix + "-Poller", this::pollLoop);
4      ...
5  }
6  /**
7   * Polling loop.
8   */
9  private void pollLoop() {
10     for (;;) {
11         poll();
12     }
13 }
14
15 int poll(int timeout) throws IOException {
16         int n = KQueue.poll(kqfd, address, MAX_EVENTS_TO_POLL, timeout);
17         int i = 0;
18         while (i < n) {
19             long keventAddress = KQueue.getEvent(address, i);
20             int fdVal = KQueue.getDescriptor(keventAddress);
21             polled(fdVal);
22             i++;
```

```
        }
        return n;
    }
}

final void polled(int fdVal) {
    wakeup(fdVal);
}
private void wakeup(int fdVal) {
    Thread t = map.remove(fdVal);
    if (t != null) {
        LockSupport.unpark(t);
    }
}

public class LockSupport {
    public static void unpark(Thread thread) {
        if (thread != null) {
            if (thread.isVirtual()) {
                VirtualThreads.unpark(thread);
            } else {
                U.unpark(thread);
            }
        }
    }
}

final class VirtualThread extends BaseVirtualThread {

    void unpark() {
        ...
        submitRunContinuation();
        ...
    }

    private void submitRunContinuation(boolean lazySubmit) {
        ...
        scheduler.execute(runContinuation);
        ...
    }

    private void runContinuation() {
        ...
        cont.run();
        ...
    }
}
```

虚拟是轻量级的，不会实际创建一个内核线程(ForkjoinPool)，并不需要使用线程池，在需要的时候创建即可。

虚拟线程由JDK负责调度，JDK实际创建一个平台线程池，负责调度和挂起。在虚拟线程运行过程中，可能会被多个平台线程调度。每次进行IO阻塞时就会进行挂起，恢复时会被调度到平台线程继续执行。

Synchronized将虚拟线程一直固定在平台线程上，阻塞操作不会卸载虚拟线程，影响程序的吞吐量。用 ReentrantLock 替换 Synchronized。正式发布或后续肯定会解决。

虚拟线程创建数量巨大，有些使用ThreadLocal场景将不再合适。

在不同代码之间使用数据共享，并不是一个好的实践。创建了线程级别的全局变量。

虚拟线程生命周期短，不适合缓存，所绑定的ThreadLcoal也会被回收。

JEP429 Scoped Values来替代使用ThreadLcoal。

原理总结

协程 = Continuation + Scheduler + 阻塞操作的改造，这就是 JDK 中协程的本质。

其中 **Continuation** 提供了可以对一段指令流调用 run 和 yield 进行开始和暂停，这是协程得以切换的根本。**阻塞操作的改造**，使得开发者仍然可以编写易读的同步代码，却达到了异步执行并主动 yield 让出控制权的逻辑，本质上是方便了代码的编写。而 **Scheduler** 则是协程调度，任务还是提交到默认创建的ForkJoinPoll里去执行。

具体来说，JDK 协程的核心原理就是，当在 Virtual Thread 内部执行 IO 操作时，强制替代原有的 blocking IO 操作为 **non-blocking** IO 操作，同时通过 **Continuation.yield()** 方法暂停当前协程，让调度器 Executor 去执行其他协程代码。当 IO 操作完成后，通过 **Continuation.run()** 方法启动等待的协程，交给调度器继续执行。

Reactor和协程

虚拟线程和异步编程

反应式编程解决了平台线程需要阻塞等待其他系统响应的问题。使用异步 API 通过回调通知您结果，而不是阻塞和等待响应。当响应到达时，JVM 从线程池中分配另一个线程来处理响应。这样，处理单个异步请求将涉及多个线程。

在异步编程中，我们可以减少系统的响应延迟，但是由于硬件的限制，平台线程的数量仍然是有限的，所以我们仍然存在系统吞吐量的瓶颈。另一个问题是异步程序在不同的线程中执行，很难调试或

分析它们。

虚拟线程通过较小的语法调整提高了代码质量（降低了编码、调试和分析代码的难度），同时具有可以显着提高系统吞吐量的反应式编程的优势。

虚拟线程和Reactor

虚拟线程和Reactor是两种不同的并发处理技术，各自具有不同的优缺点和适用场景。

虚拟线程是一种轻量级的线程，由Java虚拟机创建、调度和控制，它可以更好地控制线程的执行，防止出现竞争条件和死锁等问题，同时具有更好的遍历和调试支持。虚拟线程通常适用于对并发度要求不高的任务，例如I/O密集型应用等。

Reactor是一种事件驱动并发处理模型，它使用单个线程监听所有输入事件，并通过事件通知机制调用相应的处理程序。Reactor具有高吞吐量和低延迟的优点，通常适用于对并发度要求较高的任务，例如网络应用程序等。

虚拟线程和Reactor在实现机制、适用场景和性能等方面有很大区别，不能简单地进行比较。在实际应用中，可以根据具体任务的特点和要求选择合适的并发处理技术。

ForkJoinPool

阻塞队列？拒绝策略

协程的分类

随着越来越多的语言支持协程，协程逐渐分出了两大类别。

有栈协程：通过保存栈和程序计数器来保存和恢复程序运行状态的协程实现称为有栈协程，Go、Lua 以及本文的 Loom（JDK 19 中的 Virtual Threads） 属于有栈协程。

无栈协程：而不借助额外的栈空间来保存上下文信息，通过状态机存储暂停点代码的上下文信息，称为无栈协程，上面的 C#、Python、JS、Kotlin、Dart 等通过 async/await 实现的均是无栈协程。

通过 async/await 关键字，将同步阻塞的 get 调用转为异步非阻塞的方式。

参考文献

https://www.cnblogs.com/throwable/p/16758997.html

https://blog.rockthejvm.com/ultimate-guide-to-java-virtual-threads/