Summary of Common Virtual Thread Issues

Guide Java Java Concurrency About 1572 words About 5 minutes

> Part of this article comes from Lorin 's PR .

Virtual threads were officially released in Java 21, which is a major update.

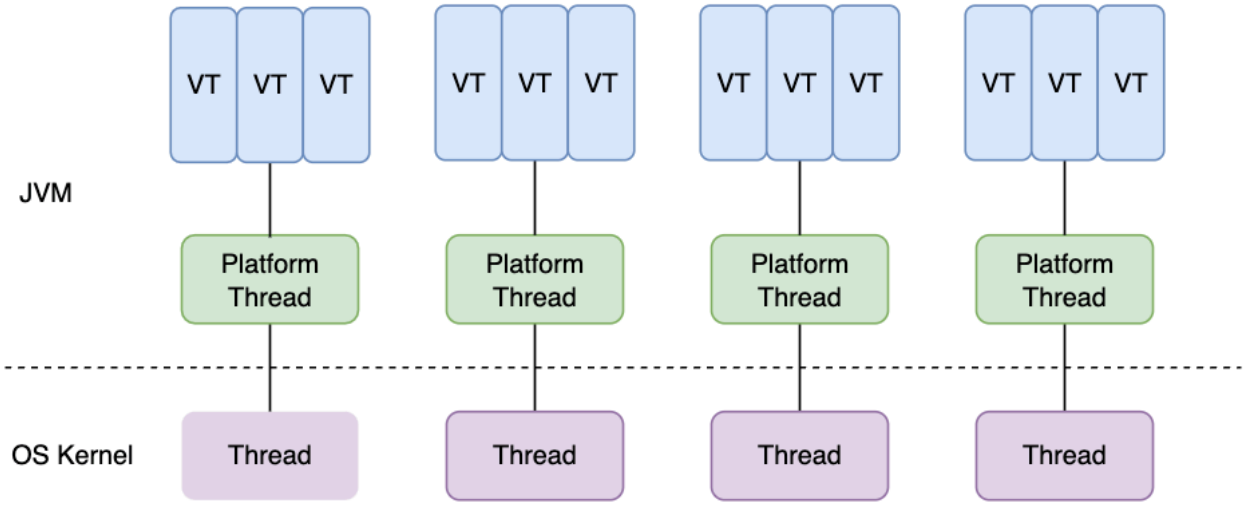
What are virtual threads?

A virtual thread is a lightweight process (LWP) implemented by the JDK rather than the OS and scheduled by the JVM. Many virtual threads share the same OS thread, and the number of virtual threads can be much greater than the number of OS threads.

What is the relationship between virtual threads and platform threads?

Before the introduction of virtual threads, `java.lang.Thread` the package already supported platform threads, which are the threads we've been using since the days of virtual threads. The JVM scheduler manages virtual threads through platform threads (carrier threads). A platform thread can execute different virtual threads at different times (multiple virtual threads are attached to a platform thread). When a virtual thread is blocked or waiting, the platform thread can switch to executing another virtual thread.

The relationship diagram of virtual threads, platform threads, and system kernel threads is shown below (Source: How to Use Java 19 Virtual Threads):



A few more points about the correspondence between platform threads and kernel threads: In mainstream operating systems like Windows and Linux, Java threads use a one-to-one thread model, meaning one platform thread corresponds to one kernel thread. Solaris is a special case, where the HotSpot VM supports both many-to-many and one-to-one threading. For more details, please refer to R's answer: Is the thread model in the JVM user-level ?

What are the advantages and disadvantages of virtual threads?

advantage

- **Very lightweight** : hundreds or thousands of virtual threads can be created in a single thread without incurring excessive thread creation and context switching.
- **Simplifying asynchronous programming** : Virtual threads can simplify asynchronous programming, making the code easier to understand and maintain. It can make asynchronous code more like synchronous code, avoiding callback hell.
- **Reduced resource overhead** : Because virtual threads are implemented by the JVM, they can more efficiently utilize underlying resources such as CPU and memory. Context switching for virtual threads is more lightweight than that of platform threads, allowing them to better support high-concurrency scenarios.

shortcoming

- **Not suitable for compute-intensive tasks** : Virtual threads are suitable for I/O-intensive tasks, but not for compute-intensive tasks, because intensive calculations always require CPU resources as support.
- **Incompatibility with some third-party libraries** : Although virtual threads are designed with compatibility with existing code in mind, some third-party libraries that rely on platform threading features may not be fully compatible with virtual threads.

How to create a virtual thread?

The official provides the following four ways to create virtual threads:

1. `Thread.startVirtualThread()` Create using
2. `Thread.ofVirtual()` Create using
3. `ThreadFactory` Create using
4. `Executors.newVirtualThreadPerTaskExecutor()` Create using

1. Use `Thread.startVirtualThread()` Create

```java
public class VirtualThreadTest {
  public static void main(String[] args) {
    CustomThread customThread = new CustomThread();
    Thread.startVirtualThread(customThread);
  }
}

static class CustomThread implements Runnable {
  @Override
  public void run() {
    System.out.println("CustomThread run");
  }
}
```

2. Use `Thread.ofVirtual()` Create

```java
public class VirtualThreadTest {
  public static void main(String[] args) {
    CustomThread customThread = new CustomThread();
    // 创建不启动
    Thread unStarted = Thread.ofVirtual().unstarted(customThread);
    unStarted.start();
    // 创建直接启动
    Thread.ofVirtual().start(customThread);
  }
}
static class CustomThread implements Runnable {
  @Override
  public void run() {
    System.out.println("CustomThread run");
  }
}
```

3. Use `ThreadFactory` Create

```java
public class VirtualThreadTest {
  public static void main(String[] args) {
    CustomThread customThread = new CustomThread();
    ThreadFactory factory = Thread.ofVirtual().factory();
    Thread thread = factory.newThread(customThread);
    thread.start();
  }
}

static class CustomThread implements Runnable {
  @Override
  public void run() {
    System.out.println("CustomThread run");
  }
}
```

4. Use `Executors.newVirtualThreadPerTaskExecutor()` Create

```java
public class VirtualThreadTest {
  public static void main(String[] args) {
    CustomThread customThread = new CustomThread();
    ExecutorService executor =
Executors.newVirtualThreadPerTaskExecutor();
```

```
    executor.submit(customThread);
  }
}
static class CustomThread implements Runnable {
  @Override
  public void run() {
    System.out.println("CustomThread run");
  }
}
```

Performance comparison between virtual threads and platform threads

Process the same task using multithreading and virtual threads, and compare the number of system threads created and the processing time.

Note : Some of the system threads created in the statistics are background threads (such as GC threads), which are the same in both scenarios and do not affect the comparison.

Test code :

```java
public class VirtualThreadTest {
    static List<Integer> list = new ArrayList<>();
    public static void main(String[] args) {
        // 开启线程 统计平台线程数
        ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(1);
        scheduledExecutorService.scheduleAtFixedRate(() -> {
            ThreadMXBean threadBean =
ManagementFactory.getThreadMXBean();
            ThreadInfo[] threadInfo =
threadBean.dumpAllThreads(false, false);
            updateMaxThreadNum(threadInfo.length);
        }, 10, 10, TimeUnit.MILLISECONDS);

        long start = System.currentTimeMillis();
        // 虚拟线程
        ExecutorService executor =
Executors.newVirtualThreadPerTaskExecutor();
        // 使用平台线程
        // ExecutorService executor =
```

```
Executors.newFixedThreadPool(200);
        for (int i = 0; i < 10000; i++) {
            executor.submit(() -> {
                try {
                    // 线程睡眠 0.5 s，模拟业务处理
                    TimeUnit.MILLISECONDS.sleep(500);
                } catch (InterruptedException ignored) {
                }
            });
        }
        executor.close();
        System.out.println("max: " + list.get(0) + " platform
thread/os thread");
        System.out.printf("totalMillis: %dms\n",
System.currentTimeMillis() - start);


    }
    // 更新创建的平台最大线程数
    private static void updateMaxThreadNum(int num) {
        if (list.isEmpty()) {
            list.add(num);
        } else {
            Integer integer = list.get(0);
            if (num > integer) {
                list.add(0, num);
            }
        }
    }
}
```

Number of requests: 10,000, single request takes 1 second :

```plain
// Virtual Thread
max: 22 platform thread/os thread
totalMillis: 1806ms

// Platform Thread  线程数200
max: 209 platform thread/os thread
totalMillis: 50578ms

// Platform Thread  线程数500
max: 509 platform thread/os thread
```

```
totalMillis: 20254ms

// Platform Thread  线程数1000
max: 1009 platform thread/os thread
totalMillis: 10214ms

// Platform Thread  线程数2000
max: 2009 platform thread/os thread
totalMillis: 5358ms

```

Number of requests: 10,000, single request takes 0.5s :

plain

```
// Virtual Thread
max: 22 platform thread/os thread
totalMillis: 1316ms

// Platform Thread  线程数200
max: 209 platform thread/os thread
totalMillis: 25619ms

// Platform Thread  线程数500
max: 509 platform thread/os thread
totalMillis: 10277ms

// Platform Thread  线程数1000
max: 1009 platform thread/os thread
totalMillis: 5197ms

// Platform Thread  线程数2000
max: 2009 platform thread/os thread
totalMillis: 2865ms
```

- It can be seen that in the scenario of intensive IO, a large number of platform threads need to be created for asynchronous processing to achieve the processing speed of virtual threads.
- Therefore, in intensive IO scenarios, virtual threads can significantly improve thread execution efficiency and reduce thread resource creation and context switching.

Note : For a while, JDK has been committed to Reactor responsive programming to improve Java performance, but responsive programming is difficult to understand, debug and use, and eventually it returned to synchronous programming, and finally virtual threads were born.

What is the underlying principle of virtual threads?

If you want to learn more about the implementation principles of virtual threads, I recommend an article: Virtual Thread - VirtualThread Source Code Perspective .

This question is generally not asked in interviews and is only for students who have spare time to further study.

Recently Updated2024/6/28 18:43
Contributors: Guide , Mr.Hope