

Summary of Common Java Concurrency Interview Questions (Part 1)

 [Guide](#)  Java  Java Concurrency  About 6908 words  About 23 minutes

This is a small advertisement that may be useful to you

- **Interview special edition** : Friends who are preparing for Java interviews can consider the interview special edition: "[Java Interview Guide](#)" (very high quality, specially designed for interviews, best used with JavaGuide).
- **Knowledge Planet** : Technical Column/One-on-one Questions/Resume Modification/Job Hunting Guide/Interview Check-in/Irregular Benefits, welcome to join [the JavaGuide official Knowledge Planet](#) .

Threads

★What are threads and processes?

What is a process?

A process is the execution of a program and the basic unit of a system's program execution. Therefore, a process is dynamic. When a system runs a program, it is a process that goes from creation to execution and then to extinction.

In Java, when we start the main function, we actually start a JVM process, and the thread where the main function is located is a thread in this process, also called the main thread.

As shown in the figure below, by viewing the Task Manager in Windows, we can clearly see the processes currently running in Windows (.exe file execution).



名称	状态	11% CPU	57% 内存	1% 磁盘	0% 网络
应用 (6)					
> Google Chrome (20)		2.4%	1,177.3	0.1 MB/秒	0 Mbps
> Microsoft OneNote		0%	13.5 MB	0 MB/秒	0 Mbps
> TIM (32 位)		0.2%	85.5 MB	0 MB/秒	0 Mbps
> WeChat (32 位) (3)		4.7%	109.4 MB	0.1 MB/秒	0 Mbps
> 金山PDF (32 位)		0%	192.0 MB	0 MB/秒	0 Mbps
> 任务管理器		0.4%	24.1 MB	0 MB/秒	0 Mbps
后台进程 (71)					
> 64-bit Synaptics Pointing Enh...		0%	0.1 MB	0 MB/秒	0 Mbps
Application Frame Host		0%	0.1 MB	0 MB/秒	0 Mbps
BaiduNetdisk (32 位)		0%	17.3 MB	0 MB/秒	0 Mbps
BaiduNetdiskHost (32 位)		0%	0.1 MB	0 MB/秒	0 Mbps
COM Surrogate		0%	1.0 MB	0 MB/秒	0 Mbps

What is a thread?

A thread is similar to a process, but a smaller execution unit than a process. A process can spawn multiple threads during its execution. Unlike processes, threads of the same type share the process's **heap** and **method area** resources, but each thread has its own **program counter**, **virtual machine stack**, and **local method stack**. Therefore, the system incurs much less overhead when spawning a thread or switching between threads than when using a process. For this reason, threads are also called lightweight processes.

Java programs are inherently multi-threaded programs. We can use JMX to see which threads a common Java program has. The code is as follows.



```
1 public class MultiThread {
2     public static void main(String[] args) {
3         // 获取 Java 线程管理 MBean
4         ThreadMXBean threadMXBean =
5         ManagementFactory.getThreadMXBean();
6         // 不需要获取同步的 monitor 和 synchronizer 信息, 仅获取线程和线程
7         堆栈信息
8         ThreadInfo[] threadInfos =
9         threadMXBean.dumpAllThreads(false, false);
10        // 遍历线程信息, 仅打印线程 ID 和线程名称信息
11        for (ThreadInfo threadInfo : threadInfos) {
12            System.out.println "[" + threadInfo.getThreadId() + "]"
13            " + threadInfo.getThreadName();
14        }
15    }
16 }
```

The output of the above program is as follows (the output may vary. Don't worry too much about the role of each thread below. Just know that the main thread executes the main method):

```
1 [5] Attach Listener //添加事件
2 [4] Signal Dispatcher // 分发处理给 JVM 信号的线程
3 [3] Finalizer //调用对象 finalize 方法的线程
4 [2] Reference Handler //清除 reference 线程
5 [1] main //main 线程,程序入口
```

From the above output, we can see that **a Java program runs with the main thread and multiple other threads running simultaneously**.

What is the difference between Java threads and operating system threads?

Prior to JDK 1.2, Java threads were implemented based on Green Threads, a type of user-level thread (user threads). This means the JVM simulated multi-threaded execution independently of the operating system. Because Green Threads have some limitations compared to native threads (for example, they cannot directly use operating system features such as asynchronous I/O and can only run on a single kernel thread, unable to utilize multiple cores), JDK 1.2 and later switched to Native Threads. This means the JVM direct-

ly uses the operating system's native kernel-level threads (kernel threads) to implement Java threads, with the operating system kernel handling thread scheduling and management.

We mentioned user threads and kernel threads above. Considering that many readers don't know the difference between the two, here is a brief introduction:

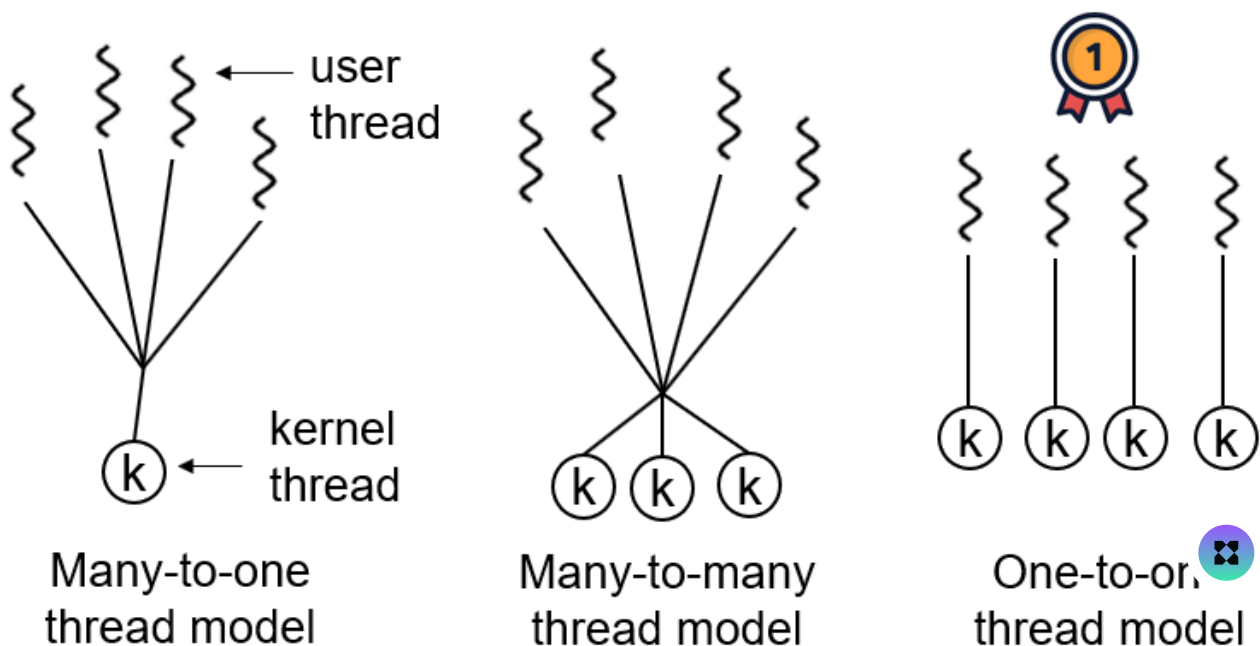
- User thread: A thread managed and scheduled by a user space program, running in user space (specifically for application use).
- Kernel thread: A thread managed and scheduled by the operating system kernel and running in kernel space (accessible only to kernel programs).

By the way, let's briefly summarize the differences and characteristics between user threads and kernel threads: User threads have low creation and switching costs, but cannot take advantage of multiple cores. Kernel threads have high creation and switching costs, but can take advantage of multiple cores.

To summarize the relationship between Java threads and operating system threads in one sentence: **the essence of current Java threads is actually operating system threads**.

The thread model is the way user threads and kernel threads are associated. There are three common thread models:

1. One-to-one (one user thread corresponds to one kernel thread)
2. Many-to-one (multiple user threads are mapped to one kernel thread)
3. Many-to-many (multiple user threads are mapped to multiple kernel threads)

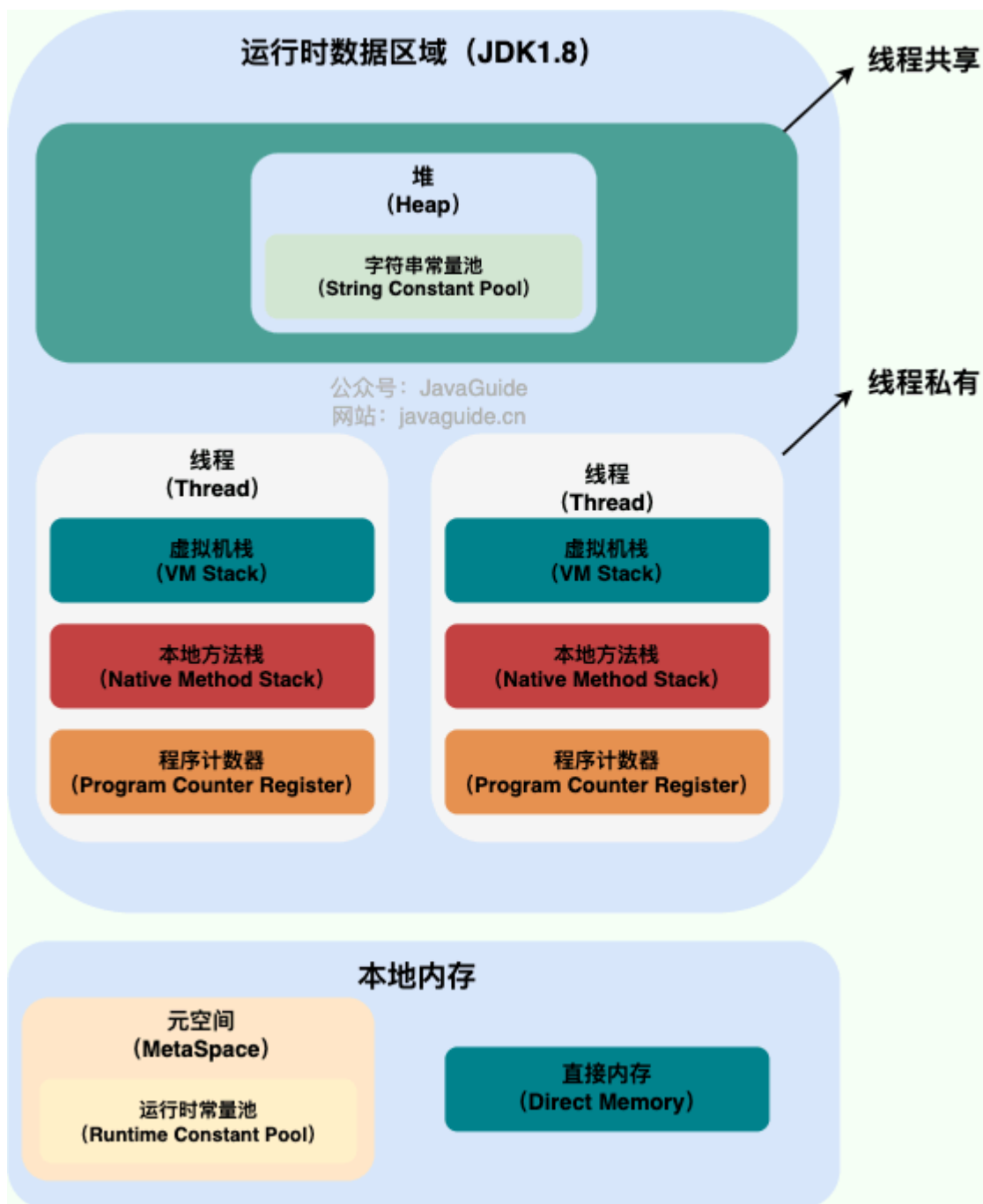


In mainstream operating systems like Windows and Linux, Java threads use a one-to-one thread model, meaning that one Java thread corresponds to one kernel thread. Solaris is a special case (it natively supports a many-to-many thread model). The HotSpot VM on Solaris supports both many-to-many and one-to-one threading. For more details, please refer to R's answer: [Is the thread model in the JVM user-level](#) ?

★ Please briefly describe the relationship between threads and processes, their differences, advantages and disadvantages?

The following figure shows the Java memory area. Through the figure below, we can explain the relationship between threads and processes from the perspective of JVM.





As can be seen from the above figure: a process can have multiple threads, and multiple threads share the process's **heap** and **method area (metaspace after JDK1.8) resources**, but each thread has its own **program counter**, **virtual machine stack**, and **local method stack**.

In summary: Threads are smaller units of execution within a process. The biggest difference between threads and processes is that processes are essentially independent, while threads are not necessarily independent. Threads within the same process are likely to interfere with each other. While threads have low execution overhead, they are less conducive to resource management and conservation. Processes, on the other hand, are the opposite.

Below is the expanded content of this knowledge point!

Let's think about this question: Why are **the program counter** , **virtual machine stack** , and **local method stack** private to each thread? Why are the heap and method area shared by each thread?

Why is the program counter private?

The program counter has the following two main functions:

1. The bytecode interpreter reads instructions in sequence by changing the program counter, thereby implementing code flow control, such as sequential execution, selection, looping, and exception handling.
2. In the case of multithreading, the program counter is used to record the location of the current thread execution, so that when the thread is switched back, it can know where the thread last ran.

It should be noted that if a native method is executed, the program counter records the undefined address. Only when Java code is executed does the program counter record the address of the next instruction.

Therefore, the main purpose of keeping the program counter private is **to restore it to the correct execution position after thread switching** .

Why are the virtual machine stack and native method stack private?

- **Virtual Machine Stack:** Before executing each Java method, a stack frame is created to store information such as the local variable table, operand stack, and constant pool references. The process from method call to completion corresponds to the process of pushing and popping a stack frame from the Java Virtual Machine stack.
- **Native method stack:** This stack functions very similarly to the virtual machine stack, but differs in that **the virtual machine stack serves the virtual machine's execution of Java methods (bytecode), while the native method stack serves the native methods used by the virtual machine.** In the HotSpot virtual machine, this stack is combined with the Java virtual machine stack.

Therefore, in order to **ensure that local variables in a thread are not accessed by other threads** , the virtual machine stack and the local method stack are thread-private.



A simple sentence to understand the heap and method area

The heap and method area are resources shared by all threads. The heap is the largest piece of memory in the process, mainly used to store newly created objects (almost all objects allocate memory here). The method area is mainly used to store loaded class information, constants, static variables, code compiled by the just-in-time compiler, and other data.

How to create a thread?

Generally speaking, there are many ways to create threads, such as inheriting `Thread` classes, implementing `Runnable` interfaces, implementing `Callable` interfaces, using thread pools, using `CompletableFuture` classes, and so on.

However, these methods do not actually create threads. To be more precise, these are all ways to use multithreading in Java code.

Strictly speaking, there is only one way to create a thread in Java, that is, by `new Thread().start()` creating. No matter which method is used, it ultimately depends on `new Thread().start()`.

For a detailed analysis of this issue, please refer to this article: [Everyone says Java has three ways to create threads! The shocking scam in concurrent programming!](#).

★Talk about the life cycle and state of threads?

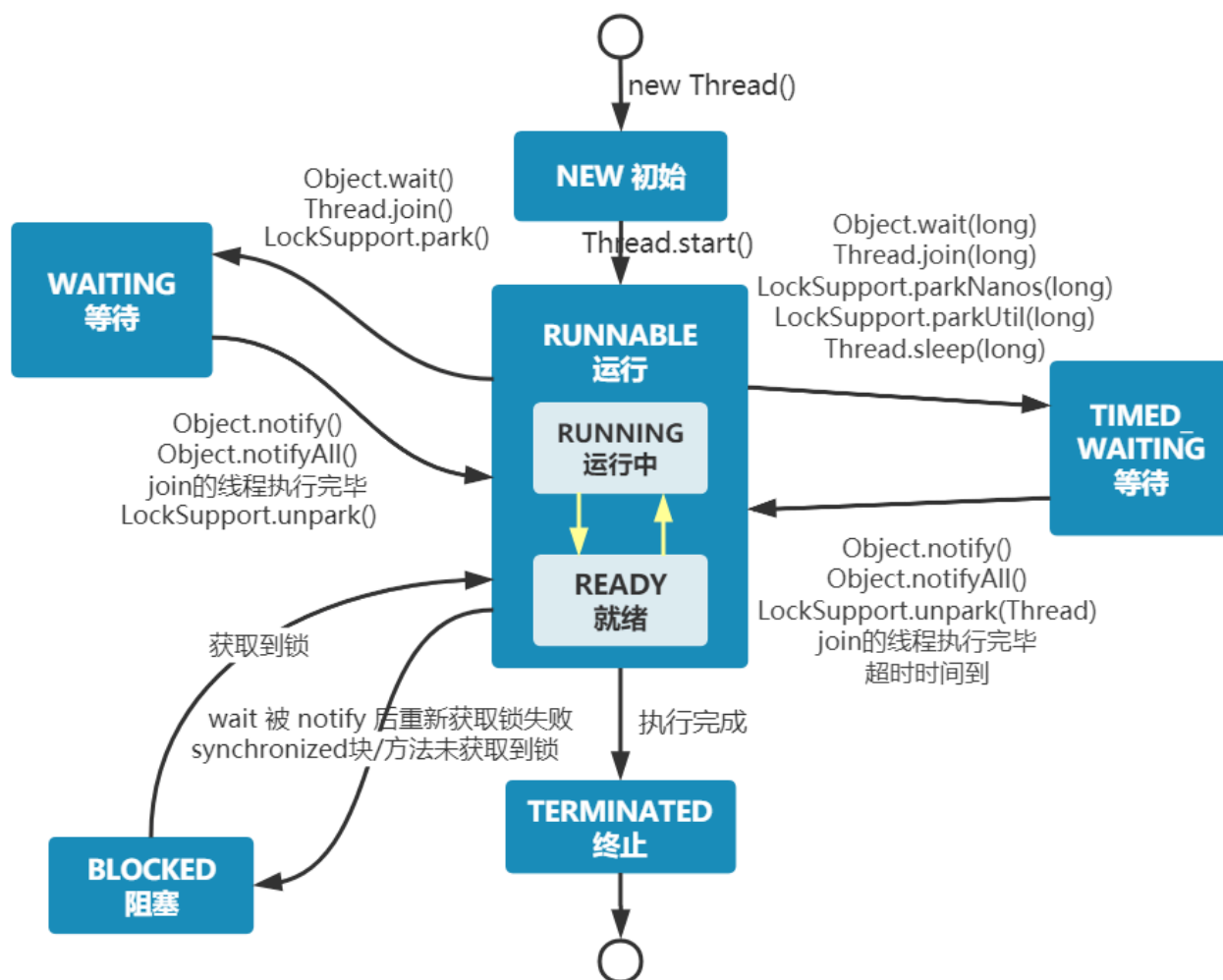
A Java thread can be in only one of the following six different states at a given moment in its lifecycle:

- **NEW:** Initial state, the thread is created but not called `start()`.
- **RUNNABLE:** Running state, the thread is called `start()` to wait for running state.
- **BLOCKED:** Blocked state, need to wait for the lock to be released.
- **WAITING:** Waiting state, indicating that the thread needs to wait for other threads to take some specific actions (notification or interruption).
- **TIME_WAITING:** Timeout waiting state, which can return automatically after the specified time instead of waiting like **WAITING**.
- **TERMINATED:** Terminated state, indicating that the thread has finished running.

A thread is not fixed in a certain state during its life cycle, but switches between different states as the code is executed.



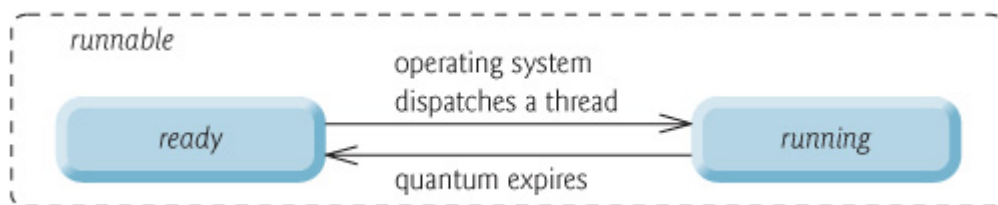
Java thread state transition diagram (Source: [Pick out errors | Three errors about thread states in "The Art of Java Concurrency Programming"](#)):



As shown in the figure above, after a thread is created, it is in **the NEW** `start()` state. After calling a method, it starts running and is in **the READY** state. **A thread in the runnable state enters the RUNNING** state after obtaining a CPU timeslice .

At the operating system level, threads have READY and RUNNING states; at the JVM level, only the RUNNABLE state can be seen (Image source: [HowToDoInJava : Java Thread Life Cycle and Thread States](#)), so the Java system generally refers to these two states as **the RUNNABLE** state.

Why doesn't the JVM distinguish between these two states? (Excerpt from: [Why is there a sixth state for Java threads? - Dawell's answer](#)) Current time-sharing multi-task operating system architectures typically use so-called "time quanta" (or time slices) for preemptive round-robin scheduling. These time slices are typically very small; a thread can only run on the CPU for a maximum of, say, 10-20 milliseconds (in the "Running" state), or about 0.01 seconds. After the time slice is used up, the thread is switched off and placed at the end of the scheduling queue, awaiting rescheduling (returning to the "Ready" state). With such rapid thread switching, distinguishing between these two states becomes meaningless.



- After a thread executes `wait()` a method, it enters the **WAITING** state. A thread in the waiting state needs to be notified by other threads before it can return to the running state.
- **The TIMED_WAITING** state adds a timeout to the WAIT state. For example, `sleep(long millis)` a method or `wait(long millis)` function can put a thread into the TIMED_WAITING state. When the timeout expires, the thread returns to the RUNNABLE state.
- When a thread enters `synchronized` a method/block or re-enters a method/block wait after calling , but the lock is held by another thread, the thread enters **the BLOCKED** state. `notify synchronized`
- After executing `run()` the method, the thread will enter **the TERMINATED** state.

What is thread context switching?

During execution, a thread has its own running conditions and state (also called context), such as the program counter and stack information mentioned above. A thread will exit the CPU-occupied state when the following conditions occur.

- Actively give up the CPU, such as calling `sleep()` , `wait()` etc.
- The time slice is used up because the operating system wants to prevent a thread or process from occupying the CPU for a long time and causing other threads or processes to starve.
- A blocking system interrupt is called, such as requesting IO, and the thread is blocked.
- Terminated or finished running

The first three of these scenarios involve thread switching. This means saving the current thread's context and restoring it the next time the thread takes over the CPU. This context is then loaded into the next thread to take over the CPU. This is known as **a context switch** .

Context switching is a basic function of modern operating systems. Because it needs to save and restore information each time, it will occupy CPU, memory and other system sources for processing, which means that there will be a certain loss of efficiency. Frequent switching will cause overall low efficiency.



Comparison between Thread#sleep() and Object#wait()

Common point : Both can pause the execution of a thread.

the difference :

- **sleep()** The method does not release the lock, while **wait()** the method releases the lock .
- **wait()** Typically used for inter-thread interaction/communication, **sleep()** and often used to pause execution.
- **wait()** After the method is called, the thread will not automatically wake up. Another thread needs to call the method on the same object **notify()** . **notifyAll()** After **sleep()** the method is executed, the thread will automatically wake up, or you can use **wait(long timeout)** the timeout to automatically wake up the thread.
- **sleep()** is Thread a static local method of the class, **wait()** while is Object a local method of the class. Why is it designed this way? The next question will discuss this.

Why is the wait() method not defined in Thread?


wait() This allows the thread that has acquired the object lock to wait, automatically releasing the object lock held by the current thread. Each object (**Object**) has an object lock. To release the object lock held by the current thread and put it into the WAITING state, the natural operation is to operate on the corresponding object (**Object**) rather than the current thread (**Thread**).

Similar question: **Why sleep() is the method defined in Thread ?**

Because **sleep()** it suspends the execution of the current thread, it does not involve the object class and does not require obtaining the object lock.

Can I directly call the run method of the Thread class?

This is another classic Java multithreading interview question that is often asked in interviews. It is very simple, but many people struggle to answer it!

When you call the new method **Thread** , the thread enters the Created state. Calling  **start()** method starts a thread and puts it into the Ready state, where it can begin running after being allocated a time slice. **start()** The method automatically executes the

thread's preparatory work and then executes `run()` the contents of the method, which is true multithreading. However, directly executing the `run()` method `run()` treats it as a normal method in the main thread and does not execute it in a specific thread, so it is not multithreaded.

Summary: Calling `start()` the method can start the thread and make the thread enter the ready state. Directly executing `run()` the method will not execute in a multi-threaded manner.

Multithreading

The difference between concurrency and parallelism

- **Concurrency** : Two or more jobs are executed in the same **time period** .
- **Parallel** : Two or more jobs are executed at the same **time** .


The most critical point is whether they are executed **simultaneously** .

The difference between synchronous and asynchronous

- **Synchronous** : After a call is issued, the call cannot return until the result is obtained and it keeps waiting.
- **Asynchronous** : After the call is issued, there is no need to wait for the return result, and the call returns directly.

★ Why use multithreading?

First, let's talk about it in general:

- **From a computer's perspective**, threads can be compared to lightweight processes, the smallest unit of program execution. The cost of switching and scheduling between threads is far less than that of processes. Furthermore, the era of multi-core CPUs means that multiple threads can run simultaneously, reducing the overhead of thread context switching.
- **From the perspective of contemporary Internet development trends**: today's systems often require millions or even tens of millions of concurrent users, and multi-threaded concurrent programming is the basis for developing high-concurrency systems. Making good use of the multi-threading mechanism can greatly improve the overall concurrency and performance of the system. 

Let's delve deeper into the computer's underlying foundation:

- **In the single-core era**, multithreading primarily improved the efficiency of a single process in utilizing the CPU and I/O system. For example, if only a single Java process were running, and an I/O request was made, if only one thread in the Java process was blocked, the entire process would be blocked. With only one CPU and one I/O device running, the overall system efficiency would be only 50%. However, with multithreading, if one thread is blocked by I/O, the other threads can continue to use the CPU, thereby improving the overall efficiency of the Java process in utilizing system resources.
- **Multi-core Era** : In the multi-core era, multithreading primarily aims to improve a process's ability to utilize multiple CPU cores. For example, if we use only one thread to perform a complex computation, only one core will be utilized, regardless of the number of CPU cores in the system. However, by creating multiple threads, these threads can be mapped to multiple underlying CPU cores for execution. Without resource competition among multiple threads within a task, task execution efficiency can be significantly improved, reaching a value approximately equal to (single-core execution time/number of CPU cores).

★ Does a single-core CPU support Java multithreading?

Single-core CPUs support Java multithreading. The operating system allocates CPU time to different threads through a round-robin system. Although a single-core CPU can only execute one task at a time, rapidly switching between multiple threads allows the user to experience multiple tasks simultaneously.

Incidentally, let me mention here the thread scheduling method used by Java.

The operating system manages the execution of multiple threads mainly through two thread scheduling methods:

- **Preemptive Scheduling** : The operating system decides when to pause the currently running thread and switch execution to another. This switch is typically triggered by a system clock interrupt (time slice rotation) or other high-priority events (such as the completion of an I/O operation). This approach incurs context switching overhead, but it offers better fairness, better CPU resource utilization, and less blocking.
- **Cooperative Scheduling** : After a thread completes execution, it proactively notifies the system to switch to another thread. This approach can reduce the performance overhead of context switching, but it suffers from poor fairness and is prone to block.



Java uses preemptive thread scheduling. That is, the JVM itself doesn't schedule threads, but delegates this to the operating system. The operating system typically schedules threads based on thread priority and time slices, with higher-priority threads typically receiving more CPU time slices.

★ Will running multiple threads on a single-core CPU be efficient?

Whether a single-core CPU can run multiple threads simultaneously efficiently depends on the type of thread and the nature of the task. Generally speaking, there are two types of threads:

1. **CPU-intensive** : CPU-intensive threads mainly perform calculations and logic processing, and require a large amount of CPU resources.
2. **IO-intensive** : IO-intensive threads mainly perform input and output operations, such as reading and writing files, network communication, etc., and need to wait for the response of IO devices without taking up too many CPU resources.

On a single-core CPU, only one thread can run at a time; other threads must wait for CPU time slices. If the thread is CPU-bound, running multiple threads simultaneously will result in frequent thread switching, increasing system overhead and reducing efficiency. If the thread is I/O-bound, running multiple threads simultaneously can utilize the CPU's idle time while waiting for I/O, improving efficiency.

Therefore, for a single-core CPU, if the task is CPU-intensive, then running many threads will affect efficiency; if the task is I/O-intensive, then running many threads will improve efficiency. Of course, the "many" here should be moderate and should not exceed the upper limit of the system's tolerance.

What problems might arise from using multithreading?

The purpose of concurrent programming is to improve the execution efficiency of the program and thus increase the running speed of the program. However, concurrent programming does not always increase the running speed of the program, and concurrent programming may encounter many problems, such as memory leaks, deadlocks, thread insecurity, etc.



How to understand thread safety and unsafety?

Thread safety and unsafety describe whether the correctness and consistency of access to the same data can be guaranteed in a multi-threaded environment.

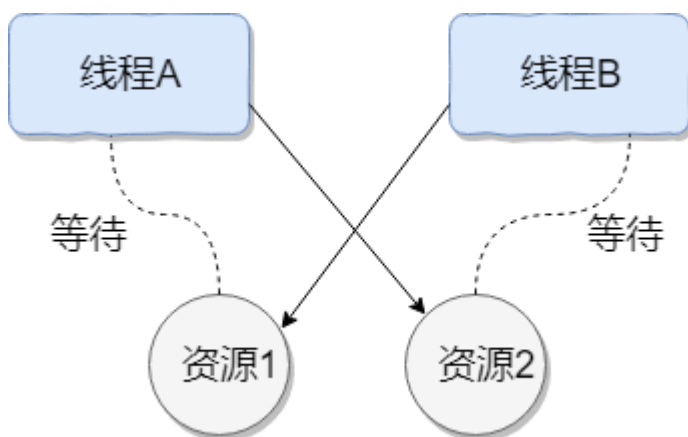
- Thread safety means that in a multi-threaded environment, for the same data, no matter how many threads access it simultaneously, the correctness and consistency of the data can be guaranteed.
- Thread unsafe means that in a multi-threaded environment, when multiple threads access the same data at the same time, data may be confused, erroneous, or lost.


★Deadlock

What is thread deadlock?

Thread deadlock describes a situation where multiple threads are blocked at the same time, one or all of them waiting for a resource to be released. Because the threads are blocked indefinitely, it is impossible for the program to terminate normally.

As shown in the figure below, thread A holds resource 2, and thread B holds resource 1. They both want to apply for each other's resources at the same time, so the two threads will wait for each other and enter a deadlock state.



The following example illustrates thread deadlock. The code simulates the deadlock situation in the figure above (the code comes from "The Beauty of Concurrency Programming" )


```
1 public class DeadLockDemo {
2     private static Object resource1 = new Object();//资源 1
3     private static Object resource2 = new Object();//资源 2
4
5     public static void main(String[] args) {
6         new Thread(() -> {
7             synchronized (resource1) {
8                 System.out.println(Thread.currentThread() + "get
9 resource1");
10                try {
11                    Thread.sleep(1000);
12                } catch (InterruptedException e) {
13                    e.printStackTrace();
14                }
15                System.out.println(Thread.currentThread() +
16 "waiting get resource2");
17                synchronized (resource2) {
18                    System.out.println(Thread.currentThread() +
19 "get resource2");
20                }
21            }
22            }, "线程 1").start();
23
24        new Thread(() -> {
25            synchronized (resource2) {
26                System.out.println(Thread.currentThread() + "get
27 resource2");
28                try {
29                    Thread.sleep(1000);
30                } catch (InterruptedException e) {
31                    e.printStackTrace();
32                }
33                System.out.println(Thread.currentThread() +
34 "waiting get resource1");
35                synchronized (resource1) {
36                    System.out.println(Thread.currentThread() +
37 "get resource1");
38                }
39            }
40            }, "线程 2").start();
41        }
42    }
43 }
```



Output

```
1 Thread[线程 1,5,main]get resource1  
2 Thread[线程 2,5,main]get resource2  
3 Thread[线程 1,5,main]waiting get resource2  
4 Thread[线程 2,5,main]waiting get resource1
```

Thread A `synchronized (resource1)` acquires `resource1` the monitor lock of `resource2` through , and then `Thread.sleep(1000);` puts thread A to sleep for 1 second through , allowing thread B to execute and acquire the monitor lock of `resource2`. After sleeping, thread A and thread B both attempt to request each other's resources, and the two threads are stuck in a state of waiting for each other, which creates a deadlock.

The above example meets the four necessary conditions for deadlock:

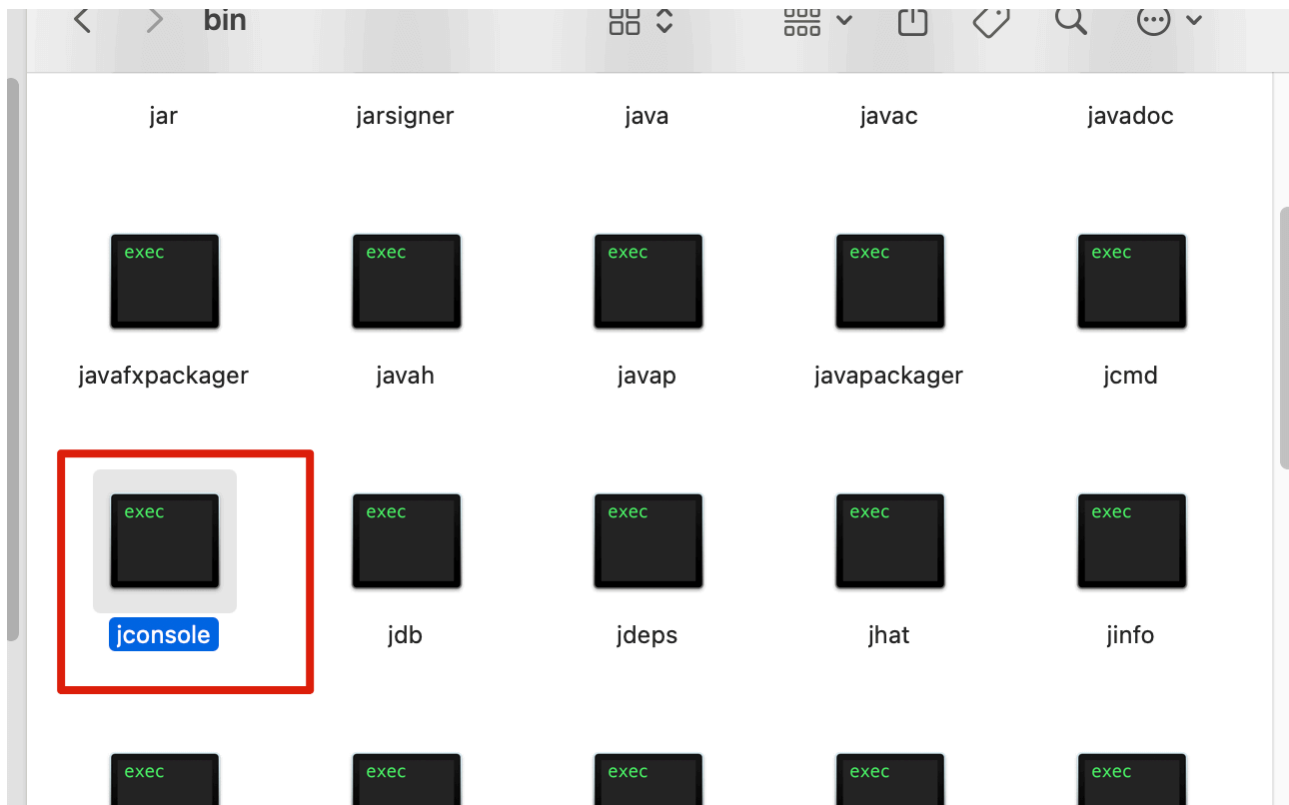
1. **Mutual exclusion condition** : The resource is occupied by only one thread at any one time.
2. **Request and hold condition** : When a thread is blocked due to requesting resources, it holds on to the resources it has obtained.
3. **Non-deprivation condition** : The resources obtained by the thread cannot be forcibly deprived by other threads before they are used up, and the resources will be released only after the thread has finished using them.
4. **Circular waiting condition** : A circular waiting resource relationship is formed between several threads, with each thread connected head to tail.

How to detect deadlock?

- Use commands like `jmap` , , `jstack` and so on to check the JVM thread stack and heap memory status. If a deadlock occurs, `jstack` the output of will typically contain `Found one Java-level deadlock:` the message "deadlock" followed by information about the deadlocked threads. In actual projects, you can also use commands `top` like , `df` , `free` and so on to check basic operating system status. Deadlocks can cause excessive consumption of resources like CPU and memory.
- Use tools such as VisualVM and JConsole for troubleshooting.

Here we take JConsole tool as an example for demonstration.

First, we need to find the bin directory of JDK, find `jconsole` and double-click to open 

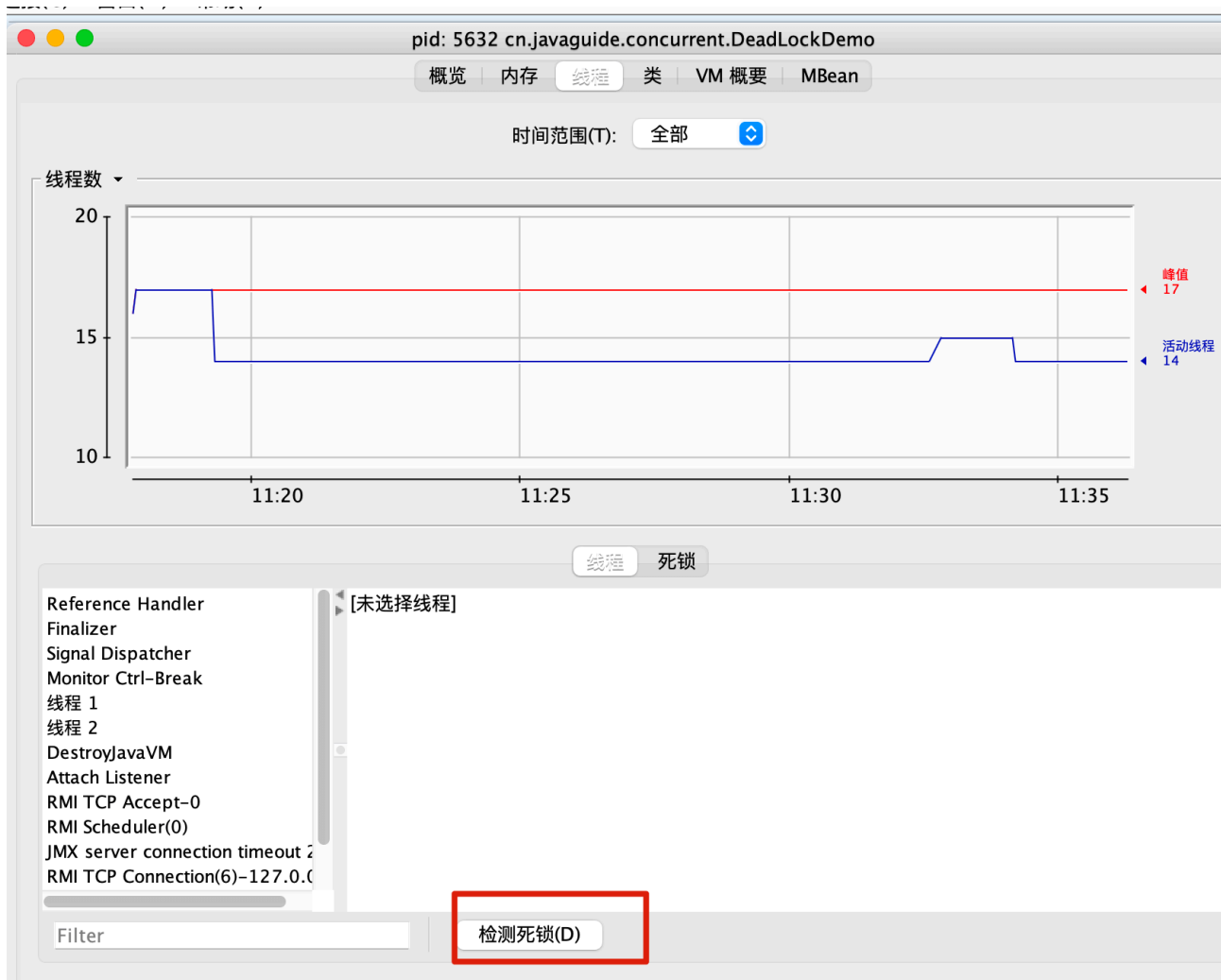


For MAC users, you can `/usr/libexec/java_home -V` check the JDK installation directory and open it after finding it `open . + 文件夹地址`. For example, the path of a local JDK is:

```
1 open . /Users/guide/Library/Java/JavaVirtualMachines/corretto-1.8.0_252/Contents/Home bash
```

After opening jconsole, connect to the corresponding program, then enter the thread interface and select detect deadlock!





线程 死锁

线程 2
线程 1

名称: 线程 2
状态: java.lang.Object@31098f93上的BLOCKED, 拥有者: 线程 1
总阻止数: 1, 总等待数: 1

堆栈跟踪:
cn.javaguide.concurrent.DeadLockDemo.lambda\$main\$1(DeadLockDemo.java:33)
- 已锁定 java.lang.Object@30e1fc43
cn.javaguide.concurrent.DeadLockDemo\$\$Lambda\$2/668386784.run(Unknown Source)
java.lang.Thread.run(Thread.java:748)

How to prevent and avoid thread deadlock?

How to prevent deadlock? Just destroy the necessary conditions for deadlock:

1. **Destroy request and hold conditions** : Apply for all resources at once.



2. **Destruction without deprivation conditions** : When a thread that occupies some resources further applies for other resources, if it fails to apply, it can actively release the resources it occupies.
3. **Breaking the circular wait condition** : This is prevented by applying for resources in order. Apply for resources in a certain order and release them in the reverse order. This breaks the circular wait condition.

How to avoid deadlock?

Avoiding deadlock means using algorithms (such as the banker's algorithm) to calculate and evaluate resource allocation when allocating resources so that it enters a safe state.

A safe state means that the system can allocate resources to each thread according to a certain thread advancement sequence (P1, P2, P3, ...Pn) until each thread's maximum resource demand is met and each thread can successfully complete. <P1、P2、P3.....Pn> This sequence is called a safe sequence.

We modify the code of thread 2 to the following so that deadlock will not occur.

```

1  new Thread(() -> {
2      synchronized (resource1) {
3          System.out.println(Thread.currentThread() + "get
4  resource1");
5          try {
6              Thread.sleep(1000);
7          } catch (InterruptedException e) {
8              e.printStackTrace();
9          }
10         System.out.println(Thread.currentThread() +
11         "waiting get resource2");
12         synchronized (resource2) {
13             System.out.println(Thread.currentThread() +
14             "get resource2");
15         }
16     }
17 }, "线程 2").start();

```

Output:



```
1 Thread[线程 1,5,main]get resource1 plain
2 Thread[线程 1,5,main]waiting get resource2
3 Thread[线程 1,5,main]get resource2
4 Thread[线程 2,5,main]get resource1
5 Thread[线程 2,5,main]waiting get resource2
6 Thread[线程 2,5,main]get resource2
7
8 Process finished with exit code 0
```

Let's analyze why the above code avoids deadlock.

Thread 1 first acquires the monitor lock for resource 1, but thread 2 is unable to obtain it. Thread 1 then attempts to acquire the monitor lock for resource 2 and succeeds. Thread 1 then releases the monitor locks for both resource 1 and resource 2, allowing thread 2 to acquire and proceed. This breaks the circular wait condition and avoids deadlock.

JavaGuide官方公众号 (微信搜索JavaGuide)



- 1、公众号后台回复“PDF”获取原创PDF面试手册
- 2、公众号后台回复“学习路线”获取Java学习路线最新版
- 3、公众号后台回复“开源”获取优质Java开源项目合集
- 4、公众号后台回复“八股文”获取Java面试真题+面经

Recently Updated 2025/8/7 15:13

Contributors: Snailclimb , MckayRen , SnailClimb , dongzl , Joe , yidasanqian , Rustin-Liu , Ryze-Zhao , Kou Shuang , tangdh , shuang.kou , liwenguang , huangshangi , guide , georgedage , VinterHe , 18986293910 , Raxcl , Guide , Mr.Hope , DaZuiZui , JunQiu , FengHaoJ , firgk , HaiBooLang , Machisk

Copyright © 2025 Guide

