Java 19 New Features Overview

Guide Java New Java Features About 1698 words About 6 minutes

JDK 19 is scheduled to be officially released for production use on September 20, 2022. It is not a long-term support version. However, there are some important new features in JDK 19 that are worth noting.

JDK 19 has only 7 new features:

- JEP 405: Record Patterns (Preview)
- JEP 422: Linux/RISC-V Port
- JEP 424: Foreign Function & Memory API (Preview)
- JEP 425: Virtual Threads (Preview)
- JEP 426: Vector API (Fourth Incubation)
- JEP 427: Pattern Matching for switch
- JEP 428: Structured Concurrency (Incubating)

Here I will only introduce in detail the four new features 424, 425, 426 and 428 which I think are more important.

Related reading: OpenJDK Java 19 documentation

JEP 424: Foreign Function and Memory API (Preview)

This API enables Java programs to interoperate with code and data outside the Java runtime. By efficiently calling external functions (that is, code outside the JVM) and safely accessing external memory (that is, memory not managed by the JVM), this API enables Java programs to call native libraries and process native data without the risks and brittleness of JNI.

The foreign function and memory APIs had their first incubation phase in Java 17, as proposed by JEP 412. They were incubated in JEP 419 for the second time and integrated into Java 18, and previewed in JEP 424 for Java 19.

Before external functions and memory API:

- Java `sun.misc.Unsafe` provides some methods for performing low-level, unsafe operations (such as directly accessing system memory resources and independently

managing memory resources). `Unsafe` Classes allow the Java language to operate memory space similar to C language pointers, but also increase the insecurity of the Java language. Improper use of `Unsafe` classes will increase the probability of program errors.

- Java 1.1 has supported native method calls through the Java Native Interface (JNI), but it wasn't very user-friendly. JNI implementation was complex and involved numerous steps (for detailed steps, refer to this article: Guide to JNI (Java Native Interface)). It wasn't controlled by the JVM's language safety mechanisms, hindering the cross-platform nature of the Java language. Furthermore, JNI performance was poor, as JNI method calls couldn't benefit from many common JIT optimizations, such as inlining. While frameworks like JNA , JNR , and JavaCPP have improved JNI, the results remain less than ideal.

The introduction of external function and memory API is to solve some pain points in Java accessing external functions and external memory.

The Foreign Function & Memory API (FFM API) defines the following classes and interfaces:

- Allocate external memory: `MemorySegment` , `MemoryAddress` and `SegmentAllocator` ;
- Manipulate and access structured external memory: `MemoryLayout` , `VarHandle` ;
- Control the allocation and release of external memory: `MemorySession` ;
- Call external functions: `Linker` , `FunctionDescriptor` and `SymbolLookup` .

The following is an example of using the FFM API. This code obtains `radixsort` the method handle of the C library function and then uses it to sort four strings in a Java array.

```java
// 1. 在C库路径上查找外部函数
Linker linker = Linker.nativeLinker();
SymbolLookup stdlib = linker.defaultLookup();
MethodHandle radixSort = linker.downcallHandle(
                            stdlib.lookup("radixsort"), ...);
// 2. 分配堆上内存以存储四个字符串
String[] javaStrings   = { "mouse", "cat", "dog", "car" };
// 3. 分配堆外内存以存储四个指针
SegmentAllocator allocator = implicitAllocator();
MemorySegment offHeap  =
allocator.allocateArray(ValueLayout.ADDRESS, javaStrings.length);
// 4. 将字符串从堆上复制到堆外
for (int i = 0; i < javaStrings.length; i++) {
    // 在堆外分配一个字符串，然后存储指向它的指针
```

```
    MemorySegment cString =
allocator.allocateUtf8String(javaStrings[i]);
    offHeap.setAtIndex(ValueLayout.ADDRESS, i, cString);
}
// 5. 通过调用外部函数对堆外数据进行排序
radixSort.invoke(offHeap, javaStrings.length, MemoryAddress.NULL,
'\0');
// 6. 将(重新排序的)字符串从堆外复制到堆上
for (int i = 0; i < javaStrings.length; i++) {
    MemoryAddress cStringPtr =
offHeap.getAtIndex(ValueLayout.ADDRESS, i);
    javaStrings[i] = cStringPtr.getUtf8String(0);
}
assert Arrays.equals(javaStrings, new String[] {"car", "cat",
"dog", "mouse"});  // true
```

JEP 425: Virtual Threads (Preview)

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

Virtual threads have proven to be very useful in other multithreaded languages, such as Goroutine in Go and processes in Erlang.

Virtual threads avoid the extra cost of context switching, take into account the advantages of multi-threading, simplify the complexity of high-concurrency programs, and effectively reduce the workload of writing, maintaining, and observing high-throughput concurrent applications.

There is a discussion on Java 19 virtual threads on Zhihu. If you are interested, you can check it out: https://www.zhihu.com/question/536743167 .

For a detailed explanation and principle of Java virtual threads, please refer to the following two articles:

- Virtual Thread Principle and Performance Analysis | Dewoo Technology
- Java 19 is now GA! See how virtual threads significantly improve system throughput
- Virtual Thread - VirtualThread Source Code Perspective

JEP 426: Vector API (Fourth Incubation)

The Vector API was originally proposed by JEP 338 and integrated into Java 16 as an incubating API . The second round of incubation was proposed by JEP 414 and integrated into Java 17, the third round was proposed by JEP 417 and integrated into Java 18, and the fourth round was proposed by JEP 426 and integrated into Java 19.

I have already introduced the vector API in detail in the Java 18 new features overview , so I will not give an additional introduction here.

JEP 428: Structured Concurrency (Incubating)

JDK 19 introduces structured concurrency, a multi-threaded programming method. Its purpose is to simplify multi-threaded programming through structured concurrency APIs, not to replace them `java.util.concurrent` . It is currently in the incubator stage.

Structured concurrency treats multiple tasks running in different threads as a single unit of work, simplifying error handling, improving reliability, and enhancing observability. In other words, structured concurrency preserves the readability, maintainability, and observability of single-threaded code.

The basic API of structured concurrency is `StructuredTaskScope` to `StructuredTaskScope` support splitting a task into multiple concurrent subtasks, which are executed in their own threads, and the subtasks must complete before the main task continues.

`StructuredTaskScope` The basic usage is as follows:

```java
try (var scope = new StructuredTaskScope<Object>()) {
    // 使用fork方法派生线程来执行子任务
    Future<Integer> future1 = scope.fork(task1);
    Future<String> future2 = scope.fork(task2);
    // 等待线程完成
    scope.join();
    // 结果的处理可能包括处理或重新抛出异常
    ... process results/exceptions ...
} // close
```

Structured concurrency is well suited for virtual threads, which are lightweight threads implemented by the JDK. Many virtual threads share the same operating system thread, allowing for a very large number of virtual threads.

Recently Updated2024/9/20 10:02
Contributors: guide , Mr.Hope , Guide