Virtual Threads: New Foundations for High-Scale Java Applications

Key Takeaways

- Virtual threads are a lightweight implementation of Java threads, delivered as a preview feature in Java 19.
- Virtual threads dramatically reduce the effort of writing, maintaining, and observing high-throughput concurrent applications.
- Virtual threads breathe new life into the familiar thread-per-request style of programming, allowing it to scale with near-optimal hardware utilization.
- Virtual threads are fully compatible with the existing `Thread` API, so existing applications and libraries can support them with minimal change.
- Virtual threads support the existing debugging and profiling interfaces, enabling easy troubleshooting, debugging, and profiling of virtual threads with existing tools and techniques.

Java 19 brings the first preview of *virtual threads* to the Java platform; this is the main deliverable of OpenJDKs Project Loom. This is one of the biggest changes to come to Java in a long time -- and at the same time, is an almost imperceptible change. Virtual threads fundamentally change how the Java runtime interacts with the underlying operating system, eliminating significant impediments to scalability -- but change relatively little about how we build and maintain concurrent programs. There is almost zero new API surface, and virtual threads behave almost exactly like the threads we already know. Indeed, to use virtual threads effectively, there is more *unlearning* than learning to be done.

Threads

Threads are foundational in Java. When we run a Java program, its main method is invoked as the first call frame of the `"main"` thread, which is created by the Java launcher. When one method calls another, the callee runs on the same thread as the caller, and where to return to is recorded on the threads stack. When a method uses local variables, they are stored in that methods call frame on the threads stack. When something goes wrong, we can reconstruct the context of how we got to the current point -- a stack trace -- by walking the current threads stack. Threads give us so many things we take for granted every day: sequential control flow, local variables, exception handling, single-step debugging, and profiling. Threads are also the basic unit of scheduling in Java programs; when a thread blocks waiting for a storage device, network connection, or a lock, the thread is descheduled so another thread can run on that CPU. Java was the first mainstream language to feature integrated support for thread-based concurrency, including a cross-platform memory model; threads are foundational to Javas model of concurrency.

Despite all this, threads often get a bad reputation, because most developers experience with threads is in trying to implement or debug shared-state concurrency. Indeed, shared-state concurrency -- often referred to as "programming with threads and locks" -- can be difficult. Unlike many other aspects of programming on the Java platform, the answers are not all to be found in the language specification or API documentation; writing safe, performant concurrent code that manages shared mutable state requires understanding subtle concepts like memory visibility, and a great deal of discipline. (If it were easier, the authors own *Java Concurrency in Practice* would not weigh in at almost 400 pages.)

Despite the legitimate apprehension that developers have when approaching concurrency, it is easy to forget that the other 99% of the time, threads are quietly and reliably making our lives much easier, giving us exception handling with informative stack traces, serviceability tools that let us observe what is going on in each thread, remote debugging, and the illusion of sequentiality that makes our code easier to reason about.

Platform threads

Java achieved write-once, run-anywhere for concurrent programs by ensuring that the language and APIs provided a complete, portable abstraction for threads, inter-thread coordination mechanisms, and a memory model that gives predictable semantics to the effects of threads on memory, that could be efficiently mapped to a number of different underlying implementations.

Most JVM implementations today implement Java threads as thin wrappers around operating system threads; well call these heavyweight, OS-managed threads *platform threads*. This isnt required -- in fact, Javas threading model predates widespread OS support for threads -- but because modern OSes now have good support for threads (in most OSes today, the thread is the basic unit of scheduling), there are good reasons to lean on the underlying platform threads. But this reliance on OS threads has a downside: because of how most OSes implement threads, thread creation is relatively expensive and resource-heavy. This implicitly places a practical limit on how many we can create, which in turn has consequences for how we use threads in our programs.

Operating systems typically allocate thread stacks as monolithic blocks of memory at thread creation time that cannot be resized later. This means that threads carry with them megabyte-scale chunks of memory to manage the native and Java call stacks. Stack size can be tuned both with command-line switches and `Thread` constructors, but tuning is risky in both directions. If stacks are overprovisioned, we will use even more memory; if they are underprovisioned, we risk `StackOverflowException` if the wrong code is called at the wrong time. We generally lean towards overprovisioning thread stacks as being the lesser of evils, but the result is a relatively low limit on how many concurrent threads we can have for a given amount of memory.

Limiting how many threads we can create is problematic because the simplest approach to building server applications is the thread-per-task approach: assign each incoming request to a single thread for the lifetime of the task.
Aligning the applications unit of concurrency (the task) with the platforms (the thread) in this way maximizes ease of development, debugging, and maintenance, leaning on all the benefits that threads invisibly give us, especially that all-important illusion of sequentiality. It usually requires little awareness of concurrency (other than configuring a thread pool for request handlers) because most requests are independent of each other. Unfortunately, as programs scale, this approach is on a collision course with the memory characteristics of platform threads. Thread-per-task scales well enough for moderate-scale applications -- we can easily service 1000 concurrent requests -- but we will not be able to service 1M concurrent requests using the same technique, even if the hardware has adequate CPU capacity and IO bandwidth.

Until now, Java developers who wanted to service large volumes of concurrent requests had several bad choices: constrain how code is written so it can use substantially smaller stack sizes (which usually means giving up on most third-party libraries), throw more hardware at the problem, or switch to an "async" or "reactive" style of programming. While the "async" model has had some popularity recently, it means programming in a highly constrained style which requires us to give up many of the benefits that threads give us, such as readable stack traces, debugging, and observability. Due to the design patterns employed by most async libraries, it also means giving up many of the benefits the Java language gives us as well, because async libraries essentially become rigid domain-specific languages that want to manage the entirety of the computation. This sacrifices many of the things that make programming in Java productive.

Virtual threads

Virtual threads are an alternative implementation of `java.lang.Thread` which store their stack frames in Javas garbage-collected heap rather than in monolithic blocks of memory allocated by the operating system. We dont have to guess how much stack space a thread might need, or make a one-size-fits-all estimate for all threads; the memory footprint for a virtual thread starts out at only a few hundred bytes, and is expanded and shrunk automatically as the call stack expands and shrinks.

The operating system only knows about platform threads, which remain the unit of scheduling. To run code in a virtual thread, the Java runtime arranges for it to run by *mounting* it on some platform thread, called a *carrier thread*. Mounting a virtual thread means temporarily copying the needed stack frames from the heap to the stack of the carrier thread, and borrowing the carriers stack while it is mounted.

When code running in a virtual thread would otherwise block for IO, locking, or other resource availability, it can be *unmounted* from the carrier thread, and any modified stack frames copied are back to the heap, freeing the carrier thread for something else (such as running another virtual thread.) Nearly all blocking points in the JDK have been adapted so that when encountering a blocking operation on a virtual thread, the virtual thread is unmounted from its carrier instead of blocking.

Mounting and unmounting a virtual thread on a carrier thread is an implementation detail that is entirely invisible to Java code. Java code cannot observe the identity of the current carrier (calling `Thread::currentThread` always returns the virtual thread); `ThreadLocal` values of the carrier thread are not visible to a mounted virtual thread; the stack frames of the carrier do not show up in exceptions or thread dumps for the virtual thread. During the virtual threads lifetime, it may run on many different carrier threads, but anything depending on thread identity, such as locking, will see a consistent picture of what thread it is running on.

Virtual threads are so-named because they share characteristics with virtual memory. With virtual memory, applications have the illusion that they have access to the entire memory address space, not limited by the available physical memory. The hardware completes this illusion by temporarily mapping plentiful virtual memory to scarce physical memory as needed, and when some other virtual page needs that physical memory, the old contents are first paged out to disk. Similarly, virtual threads are cheap and plentiful, and share the scarce and expensive platform threads as needed, and inactive virtual thread stacks are "paged" out to the heap.

Virtual threads have relatively little new API surface. There are several new methods for creating virtual threads (e.g., `Thread::ofVirtual`), but after creation, they are ordinary `Thread` objects and behave like the threads we already know. Existing APIs such as `Thread::currentThread`, `ThreadLocal`, interruption, stack walking, etc, work exactly the same on virtual threads as on platform threads, which means we can run existing code confidently on virtual threads.

The following example illustrates using virtual threads to concurrently fetch two URLs and aggregate their results as part of handling a request. It creates an `ExecutorService` that runs each task in a new virtual thread, submits two tasks to it, and waits for the results. `ExecutorService` has been retrofitted to implement `AutoCloseable`, so it can be used with `try-with-resources`, and the `close` method shuts down the executor and waits for tasks to complete.

```
void handle(Request request, Response response) {
    var url1 = ...
    var url2 = ...

    try (var executor = Executors.newVirtualThreadPerTaskExecutor()
        var future1 = executor.submit(() -> fetchURL(url1));
        var future2 = executor.submit(() -> fetchURL(url2));
        response.send(future1.get() + future2.get());
    } catch (ExecutionException | InterruptedException e) {
        response.fail(e);
    }
}

String fetchURL(URL url) throws IOException {
    try (var in = url.openStream()) {
        return new String(in.readAllBytes(), StandardCharsets.UTF_8
    }
}
```

On reading this code, we might initially worry it is somehow profligate to create threads for such short-lived activities or a thread pool for so few tasks, but this is just something we will have to unlearn -- this code is a perfectly responsible use of virtual threads

Isnt this just "green threads"?

Java developers may recall that in the Java 1.0 days, some JVMs implemented threads using user-mode, or "green", threads. Virtual threads bear a superficial similarity to green threads in that they are both managed by the JVM rather than the OS, but this is where the similarity ends. The green threads of the 90s still had large, monolithic stacks. They were very much a product of their time, when systems were single-core and OSes didnt have thread support at all. Virtual threads have more in common with the user-mode threads found in other languages, such as goroutines in Go or processes in Erlang -- but have the advantage of being semantically identical to the threads we already have.

It's about scalability

Despite the difference in creation costs, virtual threads are not *faster* than platform threads; we cant do any more computation with one virtual thread in one second than we can with a platform thread. Nor can we schedule any more *actively running* virtual threads than we can platform threads; both are limited by the number of available CPU cores. So, what is the benefit? Because they are so lightweight, we can have many more *inactive* virtual threads than we can with platform threads. At first, this may not sound like a big benefit at all! But "lots of inactive threads" actually describes the majority of server applications. Requests in server applications spend much more time doing network, file, or database I/O than computation. So if we run each task in its own thread, most of the time that thread will be blocked on I/O or other resource availability. Virtual threads allow IO-bound thread-per-task applications to *scale better* by removing the most common scaling bottleneck -- the maximum number of threads -- which in turn enables better hardware utilization. Virtual threads allow us to have the best of both worlds: a programming style that is in harmony with the platform rather than working against it, while allowing optimal hardware utilization.

For CPU-bound workloads, we already have tools to get to optimal CPU utilization, such as the fork-join framework and parallel streams. Virtual threads offer a complementary benefit to these. Parallel streams make it easier to scale CPU-bound workloads, but offer relatively little for IO-bound workloads; virtual threads offer a scalability benefit for IO-bound workloads, but relatively little for CPU-bound ones.

Littles Law

The scalability of a stable system is governed by *Littles Law*, which relates latency, concurrency, and throughput. If each request has a duration (or latency) of *d*, and we can perform *N* tasks concurrently, then throughput *T* is given by

```
T = N / d
```

Littles Law doesnt care about what portion of the time is spent "doing work" vs "waiting", or whether the unit of concurrency is a thread, a CPU, an ATM machine, or a human bank teller. It just states that to scale up the throughput, we either have to proportionally scale down the latency or scale up the number of requests we can handle concurrently. When we hit the limit on concurrent threads, the throughput of the thread-per-task model is limited by Littles Law. Virtual threads address this in a graceful way by giving us more concurrent threads rather than asking us to change our programming model.

Virtual threads in action

Virtual threads do not replace platform threads; they are complementary. However, many server applications will choose virtual threads (often through the configuration of a framework) to achieve greater scalability.

The following example creates 100,000 virtual threads that simulate an IO-bound operation by sleeping for one second. It creates a virtual-thread-per-task executor and submits the tasks as lambdas.

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 100_000).forEach(i -> {
        executor.submit(() -> {
            Thread.sleep(Duration.ofSeconds(1));
            return i;
        });
    });
}  // close() called implicitly
```

On a modest desktop system with no special configuration options, running this program takes about 1.6 seconds in a cold start, and about 1.1 seconds after warmup. If we try running this program with a cached thread pool instead, depending on how much memory is available, it may well crash with `OutOfMemoryError` before all the tasks are submitted. And if we ran it with a fixed-sized thread pool with 1000 threads, it wont crash, but Littles Law accurately predicts it will take 100 seconds to complete.

Things to unlearn

Because virtual threads are threads and have little new API surface of their own, there is relatively little to learn in order to use virtual threads. But there are actually quite a few things we need to *unlearn* in order to use them effectively.

Everyone out of the pool

The biggest thing to unlearn is the patterns surrounding thread creation. Java 5 brought with it the `java.util.concurrent` package, including the `ExecutorService` framework, and Java developers have (correctly!) learned that it is generally far better to let `ExecutorService` manage and pool threads in a policy-driven manner than to create threads directly. But when it comes to virtual threads, pooling becomes an antipattern. (We dont have to give up using `ExecutorService` or the encapsulation of policy that it provides; we can use the new factory method `Executors::newVirtualThreadPerTaskExecutor` to get an `ExecutorService` that creates a new virtual thread per task.)

Because the initial footprint of virtual threads is so small, creating virtual threads is dramatically cheaper in both time and memory than creating platform threads -- so much so, that our intuitions around thread creation need to be revisited. With platform threads, we are in the habit of pooling them, both to place a bound on resource utilization (because its easy to run out of memory otherwise), and to amortize the cost of thread startup over multiple requests. On the other hand, creating virtual threads is so cheap that it is actively a *bad* idea to pool them! We would gain little in terms of bounding memory usage, because the footprint is so small; it would take millions of virtual threads to use even 1G of memory. We also gain little in terms of amortizing creation overhead, because the creation cost is so small. And while it is easy to forget because pooling has historically been a forced move, it comes with its own problems, such as `ThreadLocal` pollution (where `ThreadLocal` values are left behind and accumulate in long-lived threads, causing memory leaks.)

If it is necessary to limit concurrency to bound consumption of some resource other than the threads themselves, such as database connections, we can use a `Semaphore` and have each virtual thread that needs the scarce resource acquire a permit.

Virtual threads are so lightweight that it is perfectly OK to create a virtual thread even for short-lived tasks, and counterproductive to try to reuse or recycle them. Indeed, virtual threads were designed with such short-lived tasks in mind, such as an HTTP fetch or a JDBC query.

Overuse of ThreadLocal

Libraries may also need to adjust their use of `ThreadLocal` in light of virtual threads. One of the ways in which `ThreadLocal` is sometimes used (some would say abused) is to cache resources that are expensive to allocate, not thread-safe, or simply to avoid repeated allocation of a commonly used object (e.g., ASM uses a `ThreadLocal` to maintain a per-thread `char[]` buffer, used for formatting operations.) When a system has a few hundred threads, the resource usage from such a pattern is usually not excessive, and it may be cheaper than reallocating each time it is needed. But the calculus changes dramatically with a few million threads that each only perform a single task, because there are potentially many more instances allocated and there is much less chance of each being reused. Using a `ThreadLocal` to amortize the creation cost of a costly resource across multiple tasks that may execute in the same thread is an ad-hoc form of pooling; if these things need to be pooled, they should be pooled explicitly.

What about Reactive?

A number of so-called "async" or "reactive" frameworks offer a path to fuller hardware utilization by asking developers to trade the thread-per-request style in favor of asynchronous IO, callbacks, and thread sharing. In such a model, when an activity needs to perform IO, it initiates an asynchronous operation which will invoke a callback when complete. The framework will invoke that callback on some thread, but not necessarily the same thread that initiated the operation. This means developers must break their logic down into alternating IO and computational steps which are stitched together into a sequential workflow. Because a request only uses a thread when it is actually computing something, the number of concurrent requests is not bounded by the number of threads, and so the limit on the number of threads is less likely to be the limiting factor in application throughput.

But, this scalability comes at a great cost -- you often have to give up some of the fundamental features of the platform and ecosystem. In the thread-per-task model, if you want to do two things sequentially, you just do them sequentially. If you want to structure your workflow with loops, conditionals, or try-catch blocks, you just do that. But in the asynchronous style, you often cannot use the sequential composition, iteration, or other features the language gives you to structure the workflow; these must be done with API calls that simulate these constructs within the asynchronous framework. An API for simulating loops or conditionals will never be as flexible or familiar as the constructs built into the language. And if we are using libraries that perform blocking operations, and have not been adapted to work in the asynchronous style, we may not be able to use these either. So we may get scalability from this model, but we have to give up on using parts of the language and ecosystem to get it.

These frameworks also make us give up a number of the runtime features that make developing in Java easier. Because each stage of a request might execute in a different thread, and service threads may interleave computations belonging to different requests, the usual tools we use when things go wrong, such as stack traces, debuggers, and profilers, are much less helpful than in the thread-per-task model. This programming style is at odds with the Java Platform because the frameworks unit of concurrency -- a stage of an asynchronous pipeline -- is not the same as the platforms unit of concurrency. Virtual threads, on the other hand, allow us to gain the same throughput benefit without giving up key language and runtime features.

What about async/await?

A number of languages have embraced `async` methods (a form of stackless coroutines) as a means of managing blocking operations, which can be called either by other `async` methods or by ordinary methods using the `await` statement. Indeed, there was some popular call to add `async/await` to Java, as `C#` and Kotlin have.

Virtual threads offer some significant advantages that `async/await` does not. Virtual threads are not just syntactic sugar for an asynchronous framework, but an overhaul to the JDK libraries to be more "blocking-aware". Without that, an errant call to a synchronous blocking method from an async task will still tie up a platform thread for the duration of the call. Merely making it syntactically easier to manage asynchronous operations does not offer any scalability benefit unless you find *every* blocking operation in your system and turn it into an `async` method.

A more serious problem with `async/await` is the "function color" problem, where methods are divided into two kinds -- one designed for threads and another designed for async methods -- and the two do not interoperate perfectly. This is a cumbersome programming model, often with significant duplication, and would require the new construct to be introduced into every layer of libraries, frameworks, and tooling in order to get a seamless result. Why would we implement yet another unit of concurrency -- one that is only syntax-deep -- which does not align with the threads we already have? This might be more attractive in another language, where language-runtime co-evolution was not an option, but fortunately we didnt have to make that choice.

API and platform changes

Virtual threads, and their related APIs, are a *preview feature*. This means that the `--enable-preview` flag is needed to enable virtual thread support.

Virtual threads are implementations of `java.lang.Thread`, so there is no new `VirtualThread` base type. However, the `Thread` API has been extended with some new API points for creating and inspecting threads. There are new factory methods for `Thread::ofVirtual` and `Thread::ofPlatform`, a new `Thread.Builder` class, and `Thread::startVirtualThread` to create a start a task on a virtual thread in one go. The existing thread constructors continue to work as before, but are only for creating platform threads.

There are a few behavioral differences between virtual and platform threads. Virtual threads are always daemon threads; the `Thread::setDaemon` method has no effect on them. Virtual threads always have priority `Thread.NORM_PRIORITY` which cannot be changed. Virtual threads do not support some (flawed) legacy mechanisms, such as `ThreadGroup` and the `Thread` methods `stop`, `suspend`, and `remove`. `Thread::isVirtual` will reveal whether a thread is virtual or not.

Unlike platform thread stacks, virtual threads can be reclaimed by the garbage collector if nothing else is keeping them alive. This means that if a virtual thread is blocked, say, on `BlockingQueue::take`, but neither the virtual thread nor the queue is reachable by any platform thread, then the thread and its stack can be garbage collected. (This is safe because in this case the virtual thread can never be interrupted or unblocked.)

Initially, carrier threads for virtual threads are threads in a `ForkJoinPool` that operates in FIFO mode. The size of this pool defaults to the number of available processors. In the future, there may be more options to create custom schedulers.

Preparing the JDK

While virtual threads are the primary deliverable of Project Loom, there was a number of improvements behind the scenes in the JDK to ensure that applications would have a good experience using virtual threads:

- **New socket implementations.** JEP 353 (Reimplement the Legacy Socket API) and JEP 373 (Reimplement the Legacy DatagramSocket API) replaced the implementations of
`Socket`
,
`ServerSocket`
, and
`DatagramSocket`
to better support virtual threads (including making blocking methods interruptible in virtual threads.)
- **Virtual-thread-awareness.** Nearly all blocking points in the JDK were made aware of virtual threads, and will unmount a virtual thread rather than blocking it.

- **Revisiting the use of `ThreadLocal`.** Many uses of `ThreadLocal` in the JDK were revised in light of the expected changing usage patterns of threads.
- **Revisiting locking.** Because acquiring an intrinsic lock (`synchronized`) currently pins a virtual thread to its carrier, critical intrinsic locks were replaced with `ReentrantLock` , which does not share this behavior. (The interaction between virtual threads and intrinsic locks is likely to be improved in the future.)
- **Improved thread dumps.** Greater control over thread dumps, such as those produced by `jcmd` , is provided to filter out virtual threads, group related virtual threads together, or produce dumps in machine-readable formats that can be post-processed for better observability.

Related work

While virtual threads are the main course of Project Loom, there are several other Loom sub-projects that further enhance virtual threads. One is a simple framework for *structured concurrency*, which offers a powerful means to coordinate and manage cooperating groups of virtual threads. The other is *extent local variables*, which are similar to thread locals, but more suitable (and performant) for use in virtual threads. These will be the topics of upcoming articles.

About the Author

Brian Goetz

Show more
Show less

Inspired by this content? Write for InfoQ.

Becoming an editor for InfoQ was one of the **best decisions of my career**. It has challenged me and **helped me grow in so many ways**. We'd love to have more people **join our team**.

Thomas Betts
Lead Editor, Software Architecture and Design @InfoQ; Senior Principal Engineer
Write for InfoQ

1