Java虚拟线程

作为 Java 并发模型的重大革新，**虚拟线程（Virtual Threads）** 在 Java 21 中正式发布（JEP 444）。它彻底改变了高并发应用的开发方式，允许开发者用简单的同步代码实现高吞吐量，同时保持资源高效性。

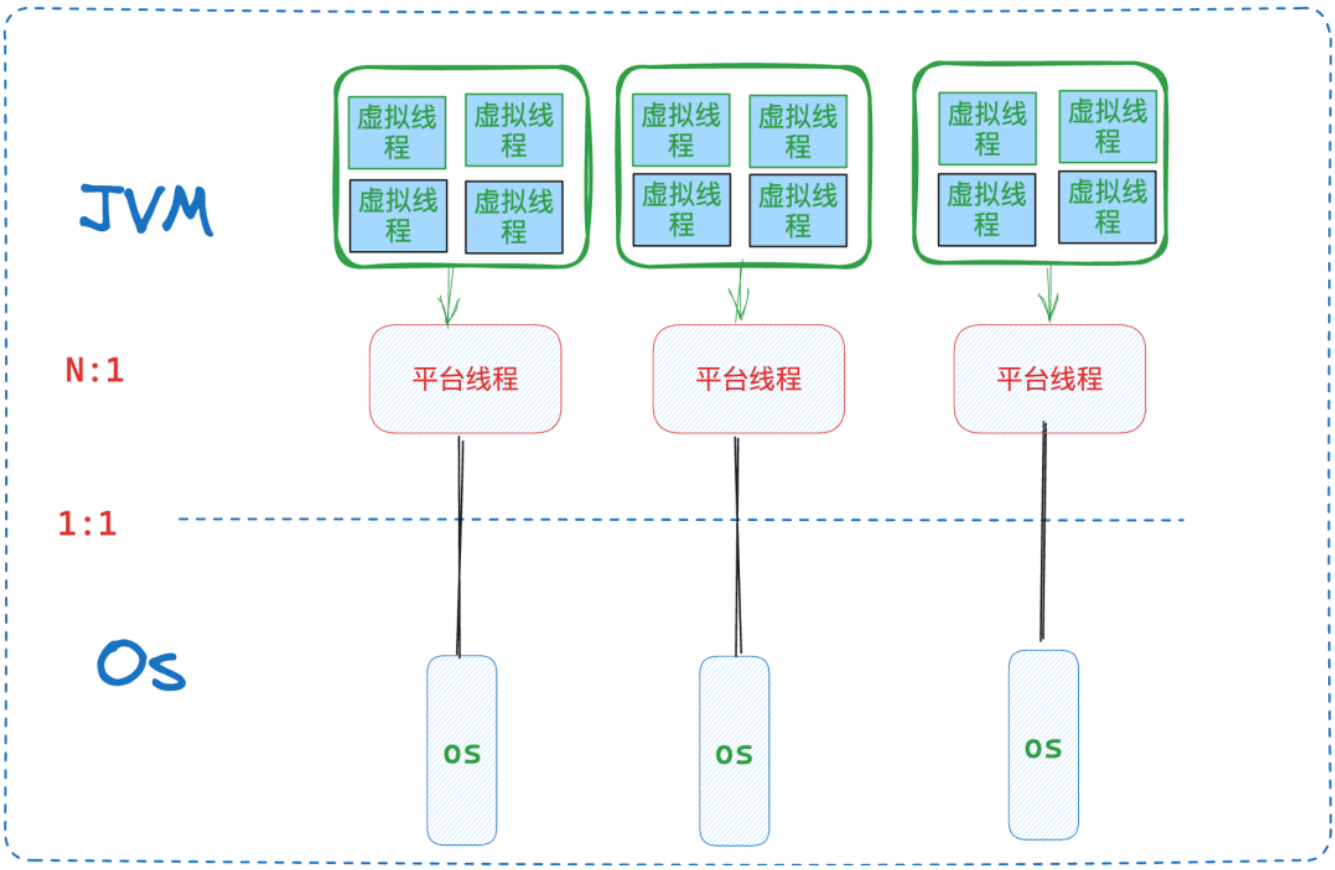
虚拟线程概述

操作系统线程（OS Thread）：由操作系统管理，是操作系统调度的基本单位。

平台线程(Platform Thread)：Java.Lang.Thread 类的每个实例，都是一个平台线程，是 Java 对操作系统线程的包装，与操作系统是 1:1 映射。平台线程在底层操作系统线程上运行 Java 代码，并在代码的整个生命周期内独占操作系统线程，平台线程实例本质是由系统内核的线程调度程序进行调度，并且**平台线程的数量受限于操作系统线程的数量。**

虚拟线程(Virtual Thread)：虚拟线程是由JDK内部实现的轻量级线程，虚拟线程它不与特定的操作系统线程相绑定。它在平台线程上运行 Java 代码，但在代码的整个生命周期内不独占平台线程。这意味着许多虚拟线程可以在同一个平台线程上运行他们的 Java 代码，共享同一个平台线程。**同时虚拟线程的成本很低，虚拟线程的数量可以比平台线程的数量大得多。**

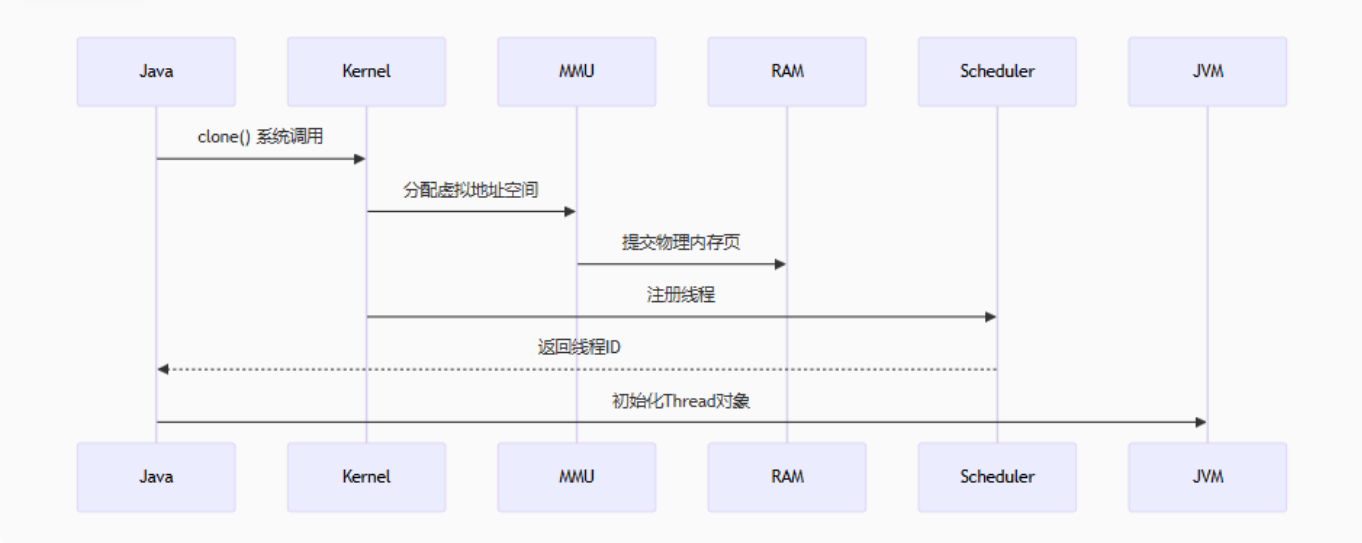
载体线程（Carrier Thread）：指真正负责执行虚拟线程中任务的平台线程。一个虚拟线程装载到一个平台线程之后，那么这个平台线程就被称为虚拟线程的载体线程。



Java 平台线程 vs 虚拟线程

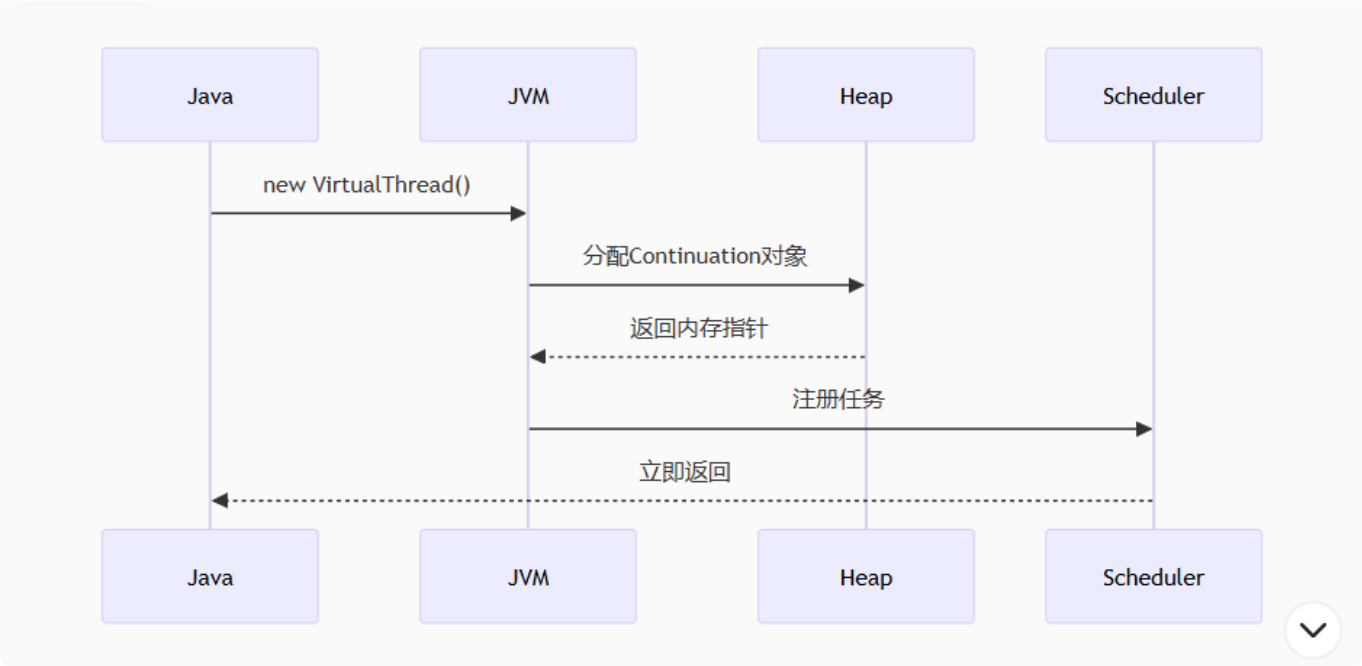
特性	平台线程	虚拟线程
实现层级	操作系统内核级	JVM 用户级
内存占用	固定大栈 (默认 1MB)	堆分配小栈 (初始 ~400 字节)
创建成本	高 (系统调用 + 内存保留)	低 (堆对象分配)
上下文切换	内核调度 (~1-10μs)	JVM 调度 (~0.1μs)
最大数量	数千级 (受内核限制)	百万级

平台线程创建流程



- **高成本来源**:
  - **系统调用**：陷入内核的上下文切换 (~100ns)
  - **内存保留**：强制预留 1MB 栈空间
  - **内核竞争**：全局线程表锁争用
  - **初始化开销**：清零栈空间

虚拟线程创建流程



- **低成本来源**：
  - **纯用户态**：无系统调用
  - **小对象分配**：相当于创建普通 Java 对象
  - **延迟初始化**：栈内存按需分配
  - **无锁注册**：工作窃取队列无竞争

虚拟线程创建

```java
// 方法一：直接创建虚拟线程
Thread vt = Thread.startVirtualThread(() -> {
    System.out.println("hello wolrd virtual thread");
});

// 方法二：创建虚拟线程但不自动运行，手动调用start()开始运行
Thread.ofVirtual().unstarted(() -> {
    System.out.println("hello wolrd virtual thread");
});
vt.start();

// 方法三：通过虚拟线程的 ThreadFactory 创建虚拟线程
ThreadFactory tf = Thread.ofVirtual().factory();
Thread vt = tf.newThread(() -> {
    System.out.println("Start virtual thread...");
    Thread.sleep(1000);
    System.out.println("End virtual thread. ");
});
vt.start();

// 方法四：Executors.newVirtualThreadPer -TaskExecutor()
ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
executor.submit(() -> {
    System.out.println("Start virtual thread...");
    Thread.sleep(1000);
    System.out.println("End virtual thread.");
    return true;
});
```

虚拟线程实现原理

虚拟线程是由 Java 虚拟机调度，而不是操作系统。虚拟线程占用空间小，同时使用轻量级的任务队列来调度虚拟线程，避免了线程间基于内核的上下文切换开销，因此可以极大量地创建和使用。

简单来看，虚拟线程实现如下：virtual thread = continuation+scheduler+runnable

虚拟线程会把任务（java.lang.Runnable实例）包装到一个 Continuation 实例中:

- 当任务需要阻塞挂起的时候，会调用 Continuation 的 yield 操作进行阻塞，虚拟线程会从平台线程卸载。
- 当任务解除阻塞继续执行的时候，调用 Continuation.run 会从阻塞点继续执行。

Scheduler 也就是执行器，由它将任务提交到具体的载体线程池中执行。

- 它是 java.util.concurrent.Executor 的子类。

- 虚拟线程框架提供了一个默认的 FIFO 的 ForkJoinPool 用于执行虚拟线程任务。

Runnable 则是真正的任务包装器，由 Scheduler 负责提交到载体线程池中执行。

JVM 把虚拟线程分配给平台线程的操作称为 mount（挂载），取消分配平台线程的操作称为 unmount（卸载）：

mount 操作：虚拟线程挂载到平台线程，虚拟线程中包装的 Continuation 堆栈帧数据会被拷贝到平台线程的线程栈，这是一个从堆复制到栈的过程。

unmount 操作：虚拟线程从平台线程卸载，此时虚拟线程的任务还没有执行完成，所以虚拟线程中包装的 Continuation 栈数据帧会会留在堆内存中。

Java虚拟线程的核心 - Continuation 续体

Continuation是一种**程序执行状态的抽象表示**，它捕获了程序在某个执行点的上下文（如调用栈、局部变量等），允许在将来恢复执行。Java 本身不原生支持 Continuation，但可通过字节码操作或 JVM 扩展实现（如 Project Loom）。以下是底层原理的核心分析：

核心概念

- **执行状态快照**：Continuation 保存当前执行点的状态：
  - 程序计数器（当前执行位置）
  - 局部变量表（Local Variables）
  - 操作数栈（Operand Stack）
  - 当前方法引用
- **挂起（Suspend）与恢复（Resume）**：
  - 挂起时：保存当前状态到 Continuation 对象。
  - 恢复时：从 Continuation 对象还原状态，跳转到上次中断点继续执行。
- `Continuation.yield()`：挂起当前 Continuation，状态存入堆内存。
- `Continuation.run()`：恢复执行，从堆内存还原栈帧。

Continuation.yield()

Continuations in Java

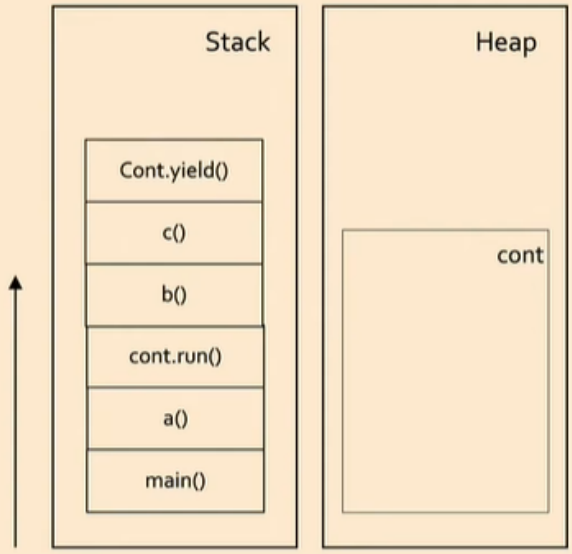
Continuation.yield()

```
void main(){
    a();
}
```

```
void a(){
    var cont = new Continuation(scope, this::b);
    cont.run();
}
```

```
void b(){
    c();
}
```

```
void c(){
    Continuation.yield(scope);
}
```



13

Continuations in Java

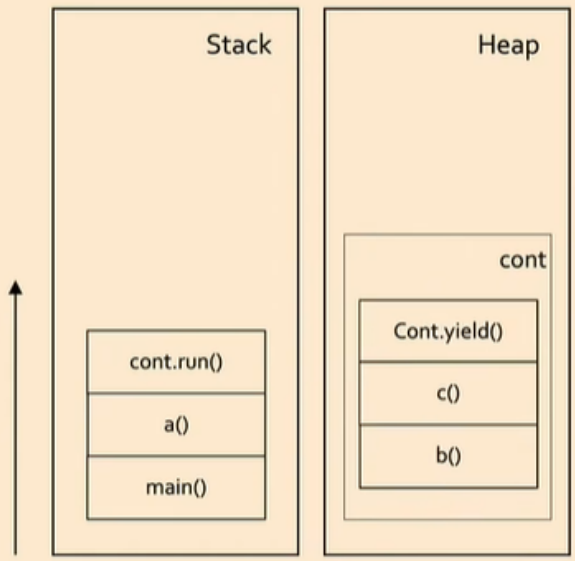
Continuation.yield()

```
void main(){
    a();
}
```

```
void a(){
    var cont = new Continuation(scope, this::b);
    cont.run();
}
```

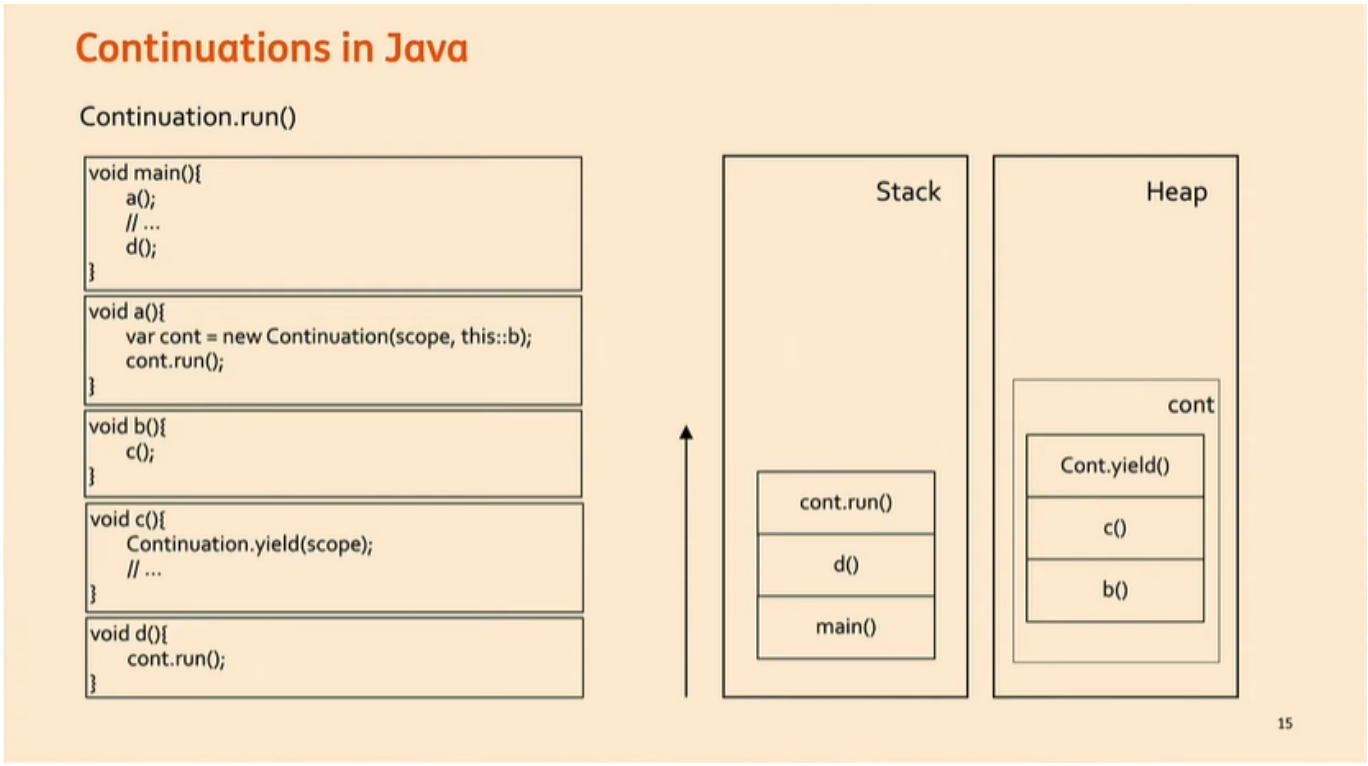
```
void b(){
    c();
}
```

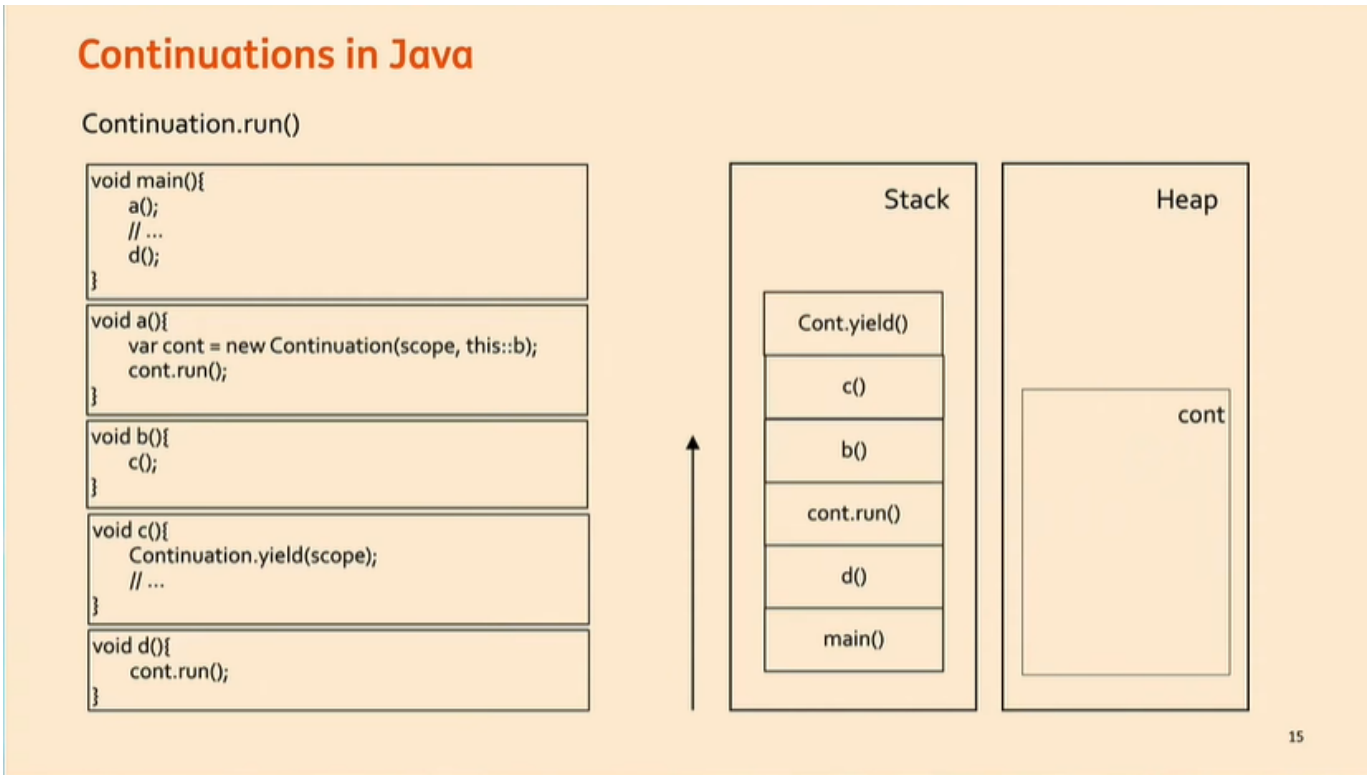
```
void c(){
    Continuation.yield(scope);
}
```



13

Continuation.run()





虚拟线程的局限及使用建议

1.synchronized 导致的线程固定（Pinning）问题

问题

- **固定（Pinning）机制**：当虚拟线程在`synchronized`代码块内执行阻塞操作（如I/O）时，JVM会将其绑定的平台线程（载体线程）一同阻塞，导致该载体线程无法切换执行其他虚拟线程。
- **根本原因**：JVM的监视器锁（Monitor）实现依赖平台线程标识。若虚拟线程在`synchronized`块内卸载，重新装载到其他平台线程时，锁持有者信息会错乱，破坏互斥性。

性能影响

- **效率严重下降**：测试表明，使用`synchronized`的虚拟线程任务耗时是`ReentrantLock`的3倍（12秒 vs 4秒）。
- **饥饿与死锁风险**：若所有载体线程被固定，新虚拟线程无法调度，导致系统吞吐量骤降

解决方案

替换为`ReentrantLock`：

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
try {
    // 阻塞操作（如I/O）
} finally {
    lock.unlock();
}
```

> PS: Java24后续针对synchronized有优化，未来`synchronized`可能不再固定线程

2.ThreadLocal的内存泄漏风险

泄漏机制

- **强引用残留**：`ThreadLocal`的值（Value）在线程的`ThreadLocalMap`中是强引用。虚拟线程生命周期短且数量巨大（百万级），若未调用`remove()`，残留值会持续占用堆内存。
- **线程复用加剧泄漏**：虚拟线程对象可能被缓存复用（如线程池），旧值未被清理，导致内存累积。

  虚拟线程下的特殊风险

- **内存压力倍增**：
  - 传统线程：千级线程 × 1KB/ThreadLocal ≈ 1MB
  - 虚拟线程：百万级线程 × 1KB/ThreadLocal ≈ 1GB ⚠

- **GC效率降低**：海量`ThreadLocalMap`条目增加GC扫描开销，引发长时间停顿。

解决方案

- **显式调用`remove()`**：

```
try {
    threadLocal.set(value);
    // 业务逻辑
} finally {
    threadLocal.remove();  // 必须清理！
}
```

- **迁移至`ScopedValue`（Java 21+）**

```
final ScopedValue<Connection> DB_CONN = ScopedValue.newInstance();
ScopedValue.where(DB_CONN, connection).run(() -> {
    queryDatabase();  // 作用域内安全使用
}); // 作用域结束自动释放
```

3.**无需池化虚拟线程**：

虚拟线程占用的资源很少，因此可以大量地创建而无须考虑池化，它不需要跟平台线程池一样，平台线程的创建成本比较昂贵，所以通常选择去池化，去做共享，**但是池化操作本身会引入额外开销**，对于虚拟线程池化反而是得不偿失，使用虚拟线程我们抛弃池化的思维，用时创建，用完就扔。

虚拟线程适用场景

- 大量的 IO 阻塞等待任务，例如下游 RPC 调用，DB 查询等。
- 大批量的处理时间较短的计算任务。
- Thread-per-request (一请求一线程)风格的应用程序，例如主流的 Tomcat 线程模型或者基于类似线程模型实现的 SpringMVC 框架 ，这些应用只需要小小的改动就可以带来巨大的吞吐提升。

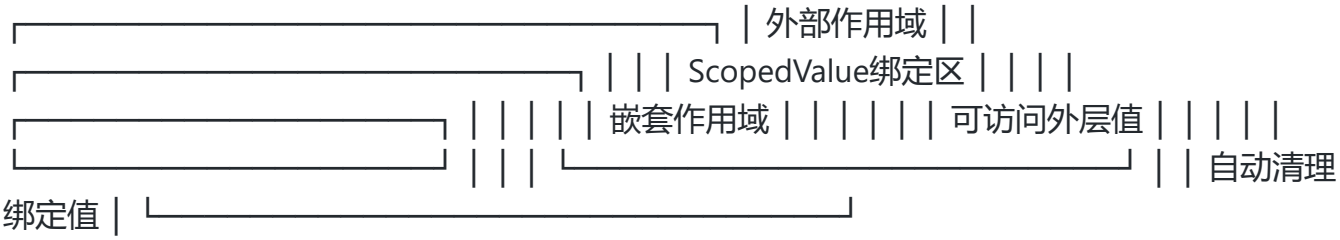
ScopeValue（作用域值）

ScopeValue 是 Java 20 引入的孵化特性（JEP 429），在 Java 21 中作为 JEP 446 继续孵化。它提供了一种**结构化、生命周期可控**的值共享机制

解决的问题

- **ThreadLocal痛点**：内存泄漏风险（尤其虚拟线程中）、隐式传播、手动清理
- **上下文传递难题**：跨组件共享请求级数据（用户认证、事务ID等）
- **虚拟线程适配**：为轻量级线程提供高效的值共享机制

生命周期模型



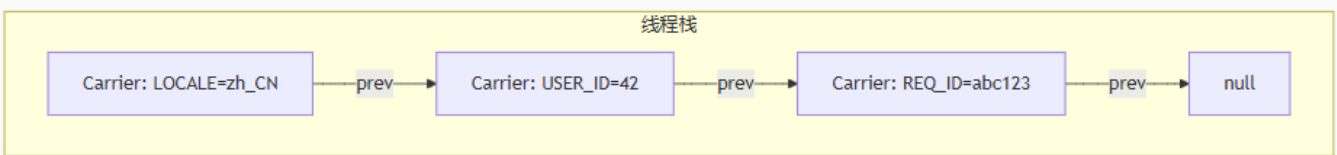
核心数据结构

```
// JDK 内部实现（简化）
public final class ScopedValue<T> {
    private final int hash; // 唯一标识
    private final T value; // 绑定值

    // 关键：作用域存储栈
    static class Carrier {
        final ScopedValue<?> key;
        final Object value;
        Carrier prev; // 栈式链接

        // 绑定新值到当前作用域
        static <T> Carrier bind(ScopedValue<T> key, T value) {
            return new Carrier(key, value, currentCarrier());
        }
    }
}
```

作用域栈工作原理



值查找算法

```
public T get() {
    Carrier current = STORAGE.get();
    while (current != null) {
        if (current.key == this) {
            return (T) current.value;
        }
        current = current.prev;
    }
    throw new NoSuchElementException();
}
```

执行流程剖析

```
// 定义作用域值
final ScopedValue<User> CURRENT_USER = ScopedValue.newInstance();
final ScopedValue<Locale> USER_LOCALE = ScopedValue.newInstance();

// 绑定并执行作用域
void processRequest(Request request) {
    User user = authenticate(request);
    Locale locale = detectLocale(request);

    ScopedValue.where(CURRENT_USER, user)
               .where(USER_LOCALE, locale)
               .run(() -> handleRequest(request));
}

// 作用域内任意位置访问
void auditAction() {
    User user = CURRENT_USER.get();  // 无需传递参数
    System.out.printf("[%s] Action performed by %s%n",
        USER_LOCALE.get(), user.name());
}
```

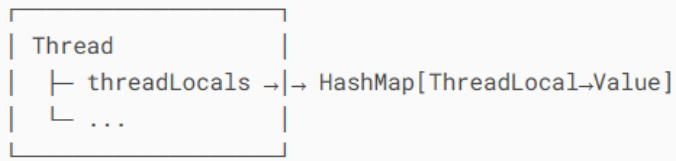
字节码实现

```
// 伪代码展示关键操作
void runWhere(Runnable op) {
    Carrier oldCarrier = currentCarrier(); // 获取当前载体
    try {
        setCurrentCarrier(this); // 压栈新载体
        op.run(); // 执行用户代码
    } finally {
        setCurrentCarrier(oldCarrier); // 出栈恢复
    }
}
```

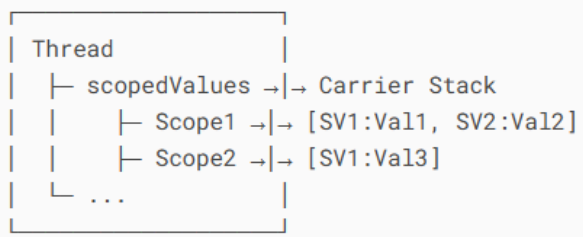
高效内存管理

线程存储结构对比：

ThreadLocal：



ScopeValue：



- **栈式存储**：O(1) 的压栈/出栈操作
- **无哈希碰撞**：直接指针访问替代哈希查找
- **自动清理**：作用域结束立即释放