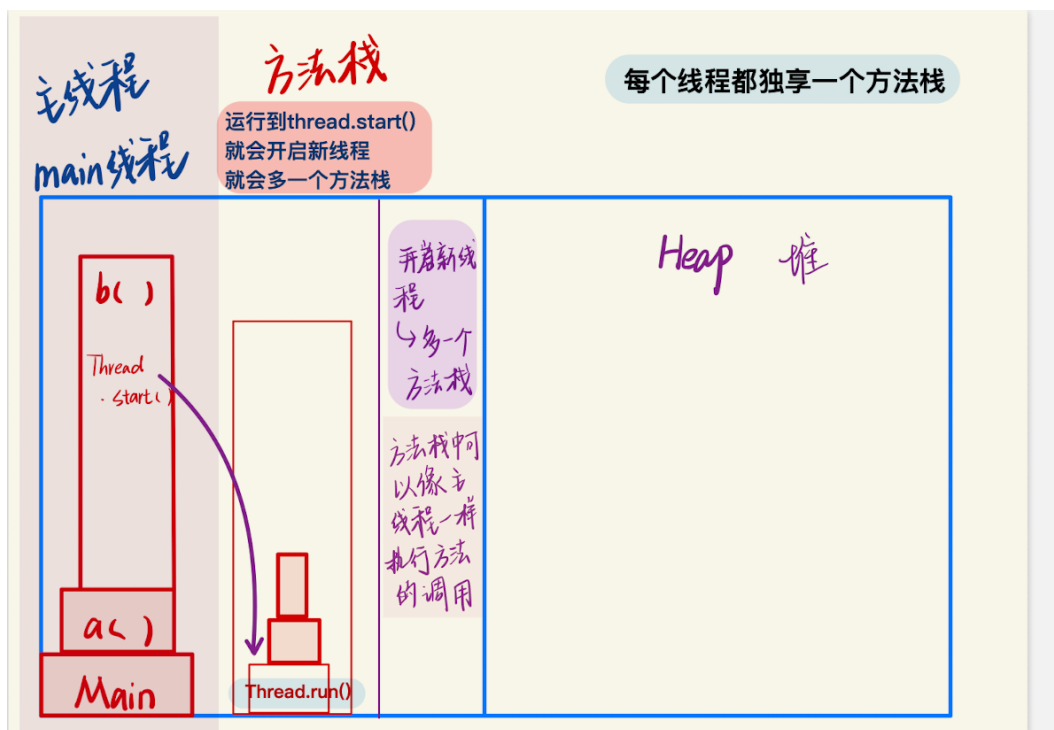




线程 方法栈 栈帧 堆

🧠 第一层核心概念：每个线程都有自己的“方法栈”

- Java 程序启动时，主线程（main thread）自动运行 `main()` 方法
- 每个线程在运行时，都会有一个属于自己的“方法栈”（stack）
- 每调用一个方法，就会在该线程的方法栈里压入一个栈帧 **stack frame**



🔴 图中最左边的部分：主线程（main 线程）

```
public static void main(String[] args) {  
    a(); // 方法 a 入栈
```

```
b(); // 方法 b 入栈  
}
```

- `main()` 方法是程序入口
- `a()` 被调用, `a()` 方法的栈帧被压入主线程的栈中
- `b()` 被调用, `b()` 方法的栈帧被压入主线程的栈中

! 方法调用=栈帧入栈

🔵 图中中间部分：创建并启动一个新线程

```
Thread t = new Thread();  
t.start(); // 注意：不是 run()
```

- `start()` 会开启一个新线程（如 `thread0`）
- 新线程会单独拥有自己的“方法栈”
- Java 会自动在这个线程中调用 `run()` 方法

✅ 所以：`t.start()` 的作用是让 JVM 去调用 `run()`，而不是你手动调用 `run()` 方法

▲ 如果你写的是：

```
t.run(); // ❌ 这不是多线程
```

这会直接在当前线程里执行 `run()` 方法，相当于普通方法调用，不会开启新线程

🔄 图中右边重点总结：


- 每个线程都有自己的“方法栈”（栈帧是私有的）
- `run()` 是新线程的执行入口（自动进入方法栈）
- 线程之间不能访问彼此的方法栈

🔑 图中粉色区域的重点：

“每个方法内部的局部变量是线程私有的”

因为：

- 方法栈是线程私有的
- 局部变量是存在方法栈里的（在栈帧中）

所以线程之间是无法访问对方的局部变量的 

举例：

```
public class MyThread extends Thread {  
    public void run() {  
        int a = 10; // a 是 run 方法中的局部变量，只属于这个线程  
    }  
}
```

多个线程都运行 `run()` 方法，每个线程都有自己的 `a` 变量，互不干扰

怎么线程之间通信？

- 局部变量在栈，线程私有 
- 成员变量 / 静态变量 / 共享对象在堆，线程可以通过引用共享 

```
class Counter {  
    public int count = 0;  
}
```

```
Counter counter = new Counter();
```

```
new Thread(() → {  
    counter.count++; // 线程A  
}).start();
```

```
new Thread(() → {
```

```
counter.count++; // 线程B
}).start();
```

两个线程都访问同一个 `Counter` 对象，这个对象在堆上，所以能共享数据

声明一个方法,调用时, JVM 在底层做什么？

名称	JVM 做了什么
方法声明	编译时记录方法签名（类名 + 方法名 + 参数类型），生成 <code>class</code> 文件
方法调用	执行时调用方法，JVM 为它创建栈帧并压入线程栈
方法执行	栈帧中存储局部变量、操作数栈等，JVM 解释/编译并执行字节码指令
方法返回	执行完毕后，栈帧弹出，控制权回到调用者

✓ 1. 方法声明时发生了什么？

```
public int add(int a, int b) {
    int sum = a + b;
    return sum;
}
```

- `javac` 编译器将 `.java` 文件编译成 `.class` 文件
- `.class` 文件中记录了该方法的 **方法签名**：
 - 方法名：`add`
 - 参数类型：`(int, int)`
 - 返回类型：`int`
- 编译器还把方法体翻译成 JVM 字节码

✓ 2. 方法调用时 JVM 做了什么？

```
int x = add(1, 2);
```

JVM

1. 定位方法定义（通过方法签名）
2. 为方法创建栈帧（stack frame）
3. 把实参复制给栈帧的局部变量表（参数传值）
4. 压入当前线程的栈中（call stack）
5. JVM 开始逐条执行字节码

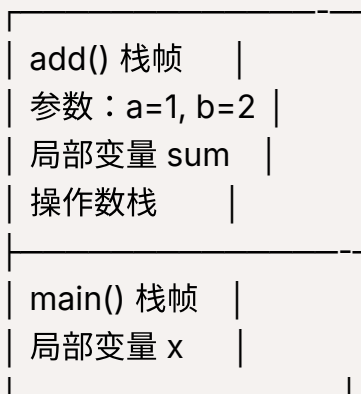
✓ 3. 栈帧（Stack Frame）内部结构

每个栈帧包含以下关键区域：

区域	含义
局部变量表	存储方法的参数和局部变量（按索引编号）
操作数栈	JVM 用来执行计算的工作区，比如执行 <code>iadd</code>
返回地址	方法执行完后，跳转回上一个方法
动态链接	指向常量池的符号引用，辅助方法解析

🧠 `add(1, 2)` 的执行结构图（逻辑）：

线程方法栈（Thread Stack）：



✓ 4. 方法返回时发生了什么？

- 当 `add()` 执行完 `return sum` 后：
 - 结果被放到调用者（main 栈帧）中对应的变量上
 - `add()` 的栈帧被弹出（pop）
 - 控制权回到 `main()` 方法继续执行

✓ 执行过程

```
public static void main(String[] args) {  
    int x = add(1, 2);  
}
```

```
public static int add(int a, int b) {  
    int sum = a + b;  
    return sum;  
}
```

JVM 运行流程：

1. JVM 启动，创建主线程，并创建 `main()` 的栈帧
 2. 执行 `add(1,2)` 时，创建新的栈帧压入栈顶
 3. 栈帧中创建参数 `a=1`，`b=2`
 4. 计算 `sum = a + b`，执行字节码指令 `iload`，`iadd`，`istore`
 5. `return sum` 把值传回 `main` 的 `x`，然后 `add()` 栈帧弹出
 6. `main()` 栈帧继续执行
-