1 什么是 Thread 类

- Thread 类是 Java 的线程类，通过它可以开启一个线程来执行 target 方法（runnable 方法）

2 Thread 类的使用场景

- 在涉及到多线程的领域中都会用到 Thread 类

3 为什么要用 Thread 类

- 通过 Thread 类开启额外线程，使得程序可以并发/并行执行，这样可以充分利用 CPU 且提高程序整体运行速度

4 Thread 类方法介绍

4.1 init(ThreadGroup, Runnable, name, stackSize)

```
private void init(ThreadGroup g, Runnable target, String name,
                  long stackSize) {
    //调用6参 init() 方法
    init(g, target, name, stackSize, null, true);
}
```

4.2 init(ThreadGroup, Runnable, name, stackSize, acc, inheritThreadLocals)

```
private void init(ThreadGroup g, Runnable target, String name,
                  long stackSize, AccessControlContext acc,
                  boolean inheritThreadLocals) {、
```

```
    //线程必须有名字
    if (name == null) {
        throw new NullPointerException("name cannot be null");
    }
    this.name = name;
    //获取创建此线程的线程
    Thread parent = currentThread();
    SecurityManager security = System.getSecurityManager();
    //如果线程组为null，则去安全管理器、父线程中找，并赋值

    if (g == null) {
        if (security != null) {
            g = security.getThreadGroup();
        }
        if (g == null) {
            //如果到这里，g 肯定会被赋值，因为 main 对应的就是最上层的线程了
            //在main线程中开启的线程，默认就设置的是 main 线程对应的线程组
            g = parent.getThreadGroup();
        }
    }
    //无论是否显式传入线程组，都要检查访问权限
    g.checkAccess();
    if (security != null) {
        if (isCCLOverridden(getClass())) {
            security.checkPermission(SUBCLASS_IMPLEMENTATION_PERMISSION);
        }
    }
    g.addUnstarted();//线程组为启动的线程数+1
    this.group = g;
    this.daemon = parent.isDaemon();//此线程的 Daemon 状态随父线程
    this.priority = parent.getPriority();
    if (security == null || isCCLOverridden(parent.getClass()))
        this.contextClassLoader = parent.getContextClassLoader();
    else
        this.contextClassLoader = parent.contextClassLoader;
    this.inheritedAccessControlContext =
        acc != null ? acc : AccessController.getContext();
    this.target = target;
    setPriority(priority);
    if (inheritThreadLocals && parent.inheritableThreadLocals != null)
        this.inheritableThreadLocals =
        ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
    /* Stash the specified stack size in case the VM cares */
    this.stackSize = stackSize;
    tid = nextThreadID();//设置 线程的 ID 值，默认递增
}
```

4.3 Thread 类的构造方法

- Thread 类的所有构造方法内部都调用了 4.1或4.2 中的 `init()` 方法
- 根据 `init()` 中的参数，组合了多种 Thread 构造方法
    - 只指定线程名：Thread(String name)

    - 什么都不指定：Thread()
    - 只指定目标任务：Thread(Runnable target)
    - 指定目标任务和线程名：Thread(Runnable target, String name)
    - 指定线程组、线程名：Thread(ThreadGroup group, String name)
    - 等等...
- 通过 Thread 构造方法可以看出
    - 可以通过构造函数里依赖 目标类(Runnable 实现类)来实现线程类
    - 也可以继承 Thread 类，并重写 run() 方法，来实现线程类
- 如果不指定线程名，则会使用默认的线程名 `"Thread-" + nextThreadNum()`

4.4 activeCount() 方法

- 此方法返回当前线程的线程组和任何其他以当前线程的线程组为**祖先**的线程组中活动线程数的**估计值**

```
public static int activeCount() {
    return currentThread().getThreadGroup().activeCount();
}
```

4.5 holdsLock(obj) 方法

- 用于判断当前线程是否持有 obj 锁

```
public static native boolean holdsLock(Object obj);
```

4.6 getStackTrace() 方法

- 返回表示此线程堆栈转储的堆栈跟踪元素数组。如果此线程尚未启动、已启动但尚未计划由系统运行或已终止，则此方法将返回零长度数组
- 如果有安全管理器存在，且不允许获取线程的栈信息，则会报 SecurityException 异常

```
public StackTraceElement[] getStackTrace() {}
```

4.7 getAllStackTraces() 方法

- 返回所有或者的线程的堆栈信息

```
public static Map<Thread, StackTraceElement[]> getAllStackTraces() {}
```

4.8 isAlive() 方法

- 判断此线程是否存活，为本地方法

```
public final native boolean isAlive();
```

4.9 isInterruted(boolean) 方法

- 判断此线程是否被中断
- 如果 boolean 为 true，且线程是中断状态，则清除中断状态

```
private native boolean isInterrupted(boolean ClearInterrupted);
```

4.10 interrupt() 方法

- 中断此线程

4.11 join()、sleep()、yield() 方法

- 对应博客地址

4.12 start() 方法

- 使此线程开始执行; Java虚拟机调用此线程的run方法
- 一个线程不能被多次执行 start() 方法

```
public synchronized void start() {
    //一个线程不能被多次执行 start() 方法的原因
    if (threadStatus != 0)
        throw new IllegalThreadStateException();

    //将此线程添加到线程组的线程列表中,并且该线程组的未启动计数可以递减
    group.add(this);
    boolean started = false;
    try {
        start0();//启动一个线程来运行 run() 方法
        started = true;
    } finally {
        try {
            //如果start0()启动失败，则将此线程从该线程组中移去，并是未启动数加1
            if (!started) {
                group.threadStartFailed(this);
            }
        } catch (Throwable ignore) {
            /* do nothing. If start0 threw a Throwable then
                  it will be passed up the call stack */
        }
    }
}
```

5 线程状态枚举类 State

- 线程在给定时间点只能处于一种状态
- 这些状态是不反映任何操作系统线程状态的虚拟机状态

```
public enum State {
    //尚未启动的线程处于此状态。
    NEW,
    //在Java虚拟机中执行的线程处于此状态。但它可能正在等待来自操作系统的其他资源，例如 CPU 时间。
    RUNNABLE,
    //被阻塞等待监视器锁定的线程处于此状态。
    //执行 synchornized 代码块，抢监视器锁失败后，进入此状态
    BLOCKED,
```

```
    //无限期等待另一个线程执行*特定操作*的线程处于此状态。
    //调用 wait()、join()、LockSupport.park() 方法，进入无限等待状态
    WAITING,
    //在指定的等待时间内等待另一个线程执行操作的线程处于此状态。
    //调用 wait(long)、join(long)、parkNanos()、parkUntil() 方法进入有限等待状态
    TIMED_WAITING,
    //已退出的线程处于此状态。线程完全执行完毕后
    TERMINATED;
}
```

参考

JDK 1.8u171