对于 Java线程 基本概念还不太明白可以参考一下这篇文章

> Java实现多线程的三种方式

1. 线程命名与获取

1.1 创建线程的时候设定名称

> public Therad(Runnable target, String name);

```
class MyThread implements Runnable {
    public void run() {
        @override
        // statment
    }
}

public class Test {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        // 设置线程名为 "线程A"
        Thread thread = new Thread(myThread, "线程A");
    }
}
```

如果没有设置线程名称，就会自动分配一个线程名称。

1.2 获得线程名称

> public final symchronized void getName(String name);

```
public class Test {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        Thread thread = new Thread(myThread, "线程A");
        thread.start();
        // 获取线程名
        System.out.println(thread.getName());

    }
}
```

1.3 设置线程名称

> public final String setName();

```
public class Test {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
```

```
        Thread thread = new Thread(myThread);
        // 使用 setName方法，一定要在启动线程之前设置线程名称
        thread.setName("线程A");
    }
}
```

线程运行中是不能进行线程名称修改的，不会抛出异常，但是修改无效。

1.4 获得当前线程对象

> public static native Thread currentThread();

```
class MyThread implements Runnable {
    public void run() {
        System.out.println("当前线程: "+Thread.currentThread().getName());
    }
}

public class Test12 {
    public static void main(String[] args) {
        new Thread(new MyThread()).start();
    }
}
```

运行结果:

```
当前线程: Thread-0
```

主方法本身也是一个线程，称为主线程，所有的线程都是通过主线程创建并启动的。

```
    // 获取主线程名称
    String mainThreadName = Thread.currentThread().getName();
```

2. 线程休眠(sleep方法)

> public static native void sleep(long millis) throws InterruptedException

让线程暂缓执行，等到了预计时间之后再恢复执行。

线程休眠会交出CPU，让CPU去执行其他的任务。sleep不会释放锁，也就是说如果当前线程池有对某个对象的锁，即使调用sleep方法，其他线程也无法访问这个对象。

```
class MyThread implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 1000 ; i++) {
            try {
                // 时间以毫秒为单位
                Thread.sleep(1000);
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
            System.out.println("当前线程: " + Thread.currentThread().getName()+" ,i = " +i);
        }
    }
}
public class Test {
    public static void main(String[] args) {
        MyThread mt = new MyThread();
        new Thread(mt).start();
        new Thread(mt).start();
        new Thread(mt).start();
    }
}
```

sleep使线程进入阻塞状态

3. 线程让步(Thread 的 yield方法)

> public static native void yield();

线程让步(Thread 的 yield方法)，暂停当前正在执行的线程对象，并执行其他线程。

```
class MyThread implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 3 ; i++) {
            Thread.yield();
            System.out.println("当前线程: " + Thread.currentThread().getName()+" ,i = " +i);
        }
    }
}
public class Test {
    public static void main(String[] args){
        MyThread mt = new MyThread();
        new Thread(mt).start();
        new Thread(mt).start();
        new Thread(mt).start();
    }
}
```

调用 yield方法 会让当前线程较出CPU权限，让CPU去执行其他的线程。它跟 sleep方法 类似，同样不会释放锁。

但是 yield 不能控制具体交出CPU的时间，另外，yield方法只能让拥有相同优先级的线程有获取COU执行时间的机会。

调用 yield方法 不会让线程进入阻塞状态，而是让线程重回就绪状态，它只需要等待重新获取CPU执行时间。

4. join方法

> 等待线程终止。如果在主线程中调用该方法就会让主线程休眠，调用该方法的线程的 run方法 先执行完毕之后再开始执行主线程

```
import java.sql.Date;
import java.text.DateFormat;
import java.text.SimpleDateFormat;

class MyThread implements Runnable {
    @Override
    public void run() {
        try {
            System.out.println("主线程睡眠前的时间");
            Test.printTime();
            Thread.sleep(2000);
            System.out.println(Thread.currentThread().getName());
            System.out.println("睡眠结束的时间");
            Test.printTime();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
public class Test {
    public static void main(String[] args) throws InterruptedException {
        MyThread mt = new MyThread();
        Thread thread = new Thread(mt, "子线程A");
        thread.start();
        System.out.println(Thread.currentThread().getName());
        thread.join();
        System.out.println("代码结束");
    }

    // 打印当前系统时间
    public static void printTime() {
        Date date = new Date();
        DateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String time = format.format(date);
        System.out.println(time);
    }
}
```

join会抛出一个 InterruptedException 受查异常

5. 线程停止

有三种停止线程的方式。

5.1 设置标记使线程停止

```
class MyThread implements Runnable {
    private boolean flag = true;
    @Override
    public void run() {
```

```
            int i = 1;
            while(this.flag) {
                try {
                    Thread.sleep(1000);
                    System.out.println(i++);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }

        public void setFlag(Boolean flag) {
            this.flag = flag;
        }
    }

    public class Test12 {
        public static void main(String[] args) throws InterruptedException {
            MyThread myThread = new MyThread();
            Thread thread = new Thread(myThread);
            thread.start();
            Thread.sleep(3000);
            // 通过将标记设置为 false 使线程停止
            myThread.setFlag(false);
            System.out.println("end");
        }
    }
```

5.2 使用 Thread 的 stop方法 使线程退出

使用 stop方法 强制退出，但是因为该方法不安全(可能会造成数据的丢失)，所以已经被废除了，因为 stop 会解除由线程获取的所有锁定，当在一个线程对象上调用 stop 时，这个线程对象所运行的线程就会立即停止，这样可能会发生数据丢失。

5.3 使用 Thread.Interrupt()

Interrupt方法 只是改变中断状态，它不会中断一个正在运行的进程。这一方法实际完成的是，给受阻塞的线程发出一个中断信号，这样使线程得以退出阻塞的状态。

然而 Interrupt方法 并不会立即执行终端操作，具体而言，这个方法只会给线程设置一个为 true 的中断标志 设置了中断标志之后，则根据线程当前的状态进行不同的后续操作。如果线程当前的状态处于非阻塞状态，那么仅仅是线程的中断标志被修改为 true 而已，如果线程的当前状态处于阻塞状态，那么就在中断标志被设置为 true 后，还会有如下几种操作

- 如果是 wait，sleep，jion 三种方法引起的阻塞，那么会将中断标志重新设置为false，并抛出一个InterruptedException
- 如果在中断时，线程正处于非阻塞状态，则将中断标志修改为true,而在此基础上，一旦进入阻塞状态，则按照阻塞状态的情况来进行处理；例如，一个线程在运行状态中，其中断标志被设置为true之后，一旦线程调用了wait、jion、sleep方法中的一种，立马抛出一个InterruptedException，且中断标志被程序会自动清除，重新设置为false。

调用线程类的 Interrupte 方法，其本质只是设置了该线程的中断标志，将中断标志设置为 true，并根据线程状态抛出异常。

因此，通过 Interrupte方法实现线程中断的原理使开发人员根据中断标志的具体值，来决定如何退出线程

总结