多线程

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

什么是线程?

- 线程(Thread)是一个程序内部的一条执行流程。

```
public static void main(String[] args) {
    // 代码…
    for (int i = 0; i < 10; i++) {
        System.out.println(i);
    }
    // 代码...
}
```

- 程序中如果只有一条执行流程，那这个程序就是单线程的程序。

多线程是什么?

- 多线程是指从软硬件上实现的多条执行流程的技术（多条线程由CPU负责调度执行）

多线程用在哪里，有什么好处

- 再例如：消息通信、淘宝、京东系统都离不开多线程技术。

如何在程序中创建出多条线程?

如何在程序中创建出多条线程?

- Java是通过java.lang.Thread 类的对象来代表线程的。

多线程的创建方式一：继承Thread类

① 定义一个子类MyThread继承线程类java.lang.Thread，重写run()方法

② 创建MyThread类的对象

③ 调用线程对象的start()方法启动线程（启动后还是执行run方法的）

方式一优缺点：

- 优点：编码简单
- 缺点：线程类已经继承Thread，无法继承其他类，不利于功能的扩展。

多线程的注意事项

1、启动线程必须是调用start方法，不是调用run方法。

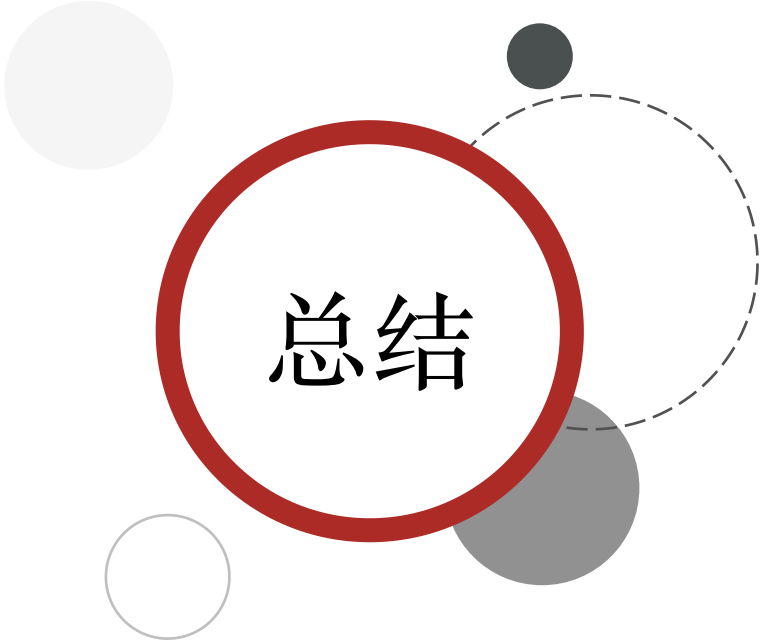
2、不要把主线程任务放在启动子线程之前。

- **直接调用run方法会当成普通方法执行，此时相当于还是单线程执行。**
- **只有调用start方法才是启动一个新的线程执行。**
- **这样主线程一直是先跑完的，相当于是一个单线程的效果了。**

线程创建方式一：继承Thread类

线程创建方式二：实现Runnable接口

线程创建方式三：实现Callable接口



1、什么是多线程？线程的代表是谁？线程创建方式一的具体步骤是？

- **继承Thread类**
- **重写run方法**
- **创建线程对象**
- **调用start()方法启动。**

2、线程的第一种创建方式有啥优缺点？

- **优点：编码简单**
- **缺点：存在单继承的局限性，线程类继承Thread后，不能继承其他类，不便于扩展。**

目录

Contents

多线程的创建方式二：实现Runnable接口

① 定义一个线程任务类MyRunnable实现Runnable接口，重写run()方法

② 创建MyRunnable任务对象

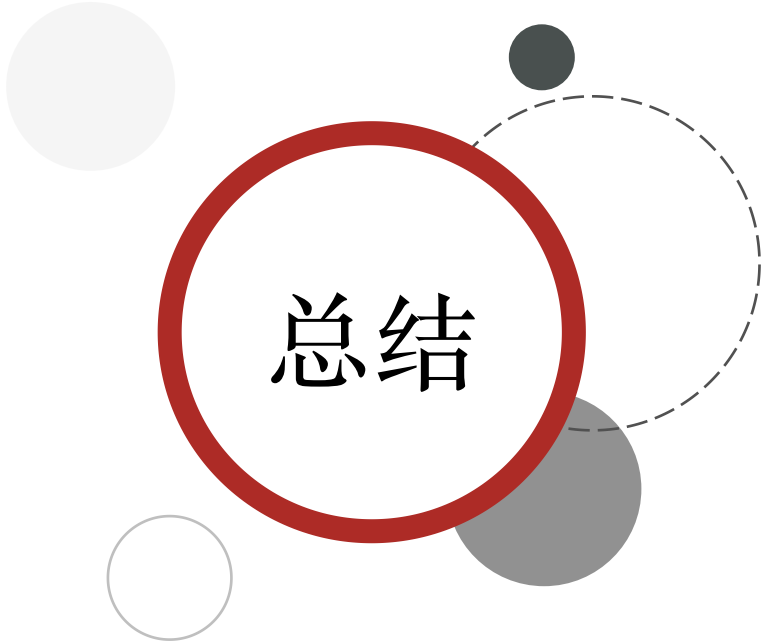
③ 把MyRunnable任务对象交给Thread处理。

Thread类提供的构造器	说明
public Thread(Runnable target)	封装Runnable对象成为线程对象

④ 调用线程对象的start()方法启动线程

方式二的优缺点

- 优点：任务类只是实现接口，可以继续继承其他类、实现其他接口，扩展性强。
- 缺点：需要多一个Runnable对象。



1. 线程创建方式二是如何创建线程的？
   - **定义一个线程任务类MyRunnable实现Runnable接口，重写run()方法**
   - **创建MyRunnable对象**
   - **把MyRunnable任务对象交给Thread线程对象处理。**
   - **调用线程对象的start()方法启动线程**
2. 方式二的优缺点是啥？
   - **优点：线程任务类只是实现了Runnale接口，可以继续继承和实现。**
   - **缺点：如果线程有执行结果是不能直接返回的。**

线程创建方式二的匿名内部类写法

① 可以创建Runnable的匿名内部类对象。

② 再交给Thread线程对象。

③ 再调用线程对象的start()启动线程。

目录

Contents

前两种线程创建方式都存在的一个问题

- 假如线程执行完毕后有一些数据需要返回，他们重写的run方法均不能直接返回结果。

怎么解决这个问题？

- JDK 5.0提供了Callable接口和FutureTask类来实现（多线程的第三种创建方式）。
- 这种方式最大的优点：可以返回线程执行完毕后的结果。

多线程的第三种创建方式：利用Callable接口、FutureTask类来实现。

① 、创建任务对象

- 定义一个类实现Callable接口，重写call方法，封装要做的事情，和要返回的数据。
- 把Callable类型的对象封装成FutureTask（线程任务对象）。

② 、把线程任务对象交给Thread对象。

③ 、调用Thread对象的start方法启动线程。

④ 、线程执行完毕后、通过FutureTask对象的的get方法去获取线程任务执行的结果。

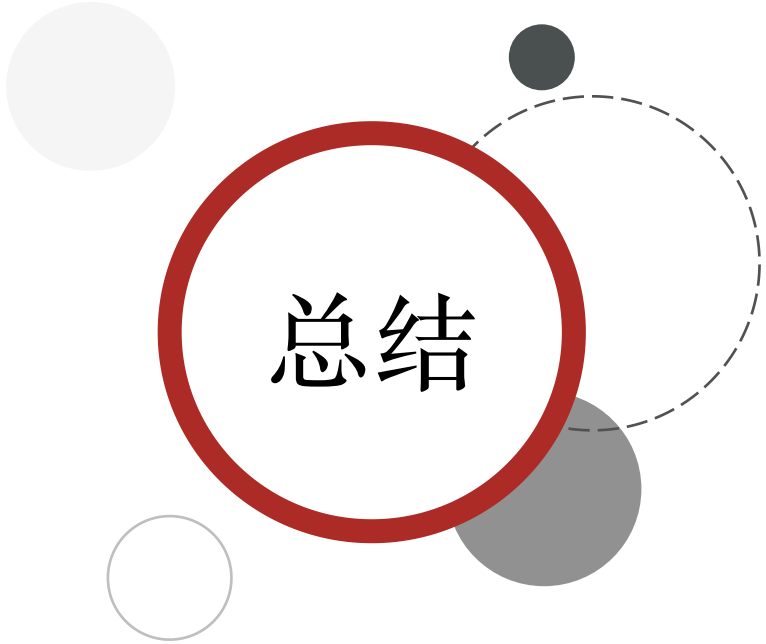
FutureTask的API

FutureTask提供的构造器	说明
public FutureTask<>(Callable call)	把Callable对象封装成FutureTask对象。

FutureTask提供的方法	说明
public V get() throws Exception	获取线程执行call方法返回的结果。

线程创建方式三的优缺点

- 优点：线程任务类只是实现接口，可以继续继承类和实现接口，扩展性强；可以在线程执行完毕后去获取线程执行的结果。
- 缺点：编码复杂一点。



1、请对对比说一下三种线程的创建方式，和不同点？

方式	优点	缺点
继承Thread类	编程比较简单，可以直接使用Thread类中的方法	扩展性较差，不能再继承其他的类，不能返回线程执行的结果
实现Runnable接口	扩展性强，实现该接口的同时还可以继承其他的类。	编程相对复杂，不能返回线程执行的结果
实现Callable接口	扩展性强，实现该接口的同时还可以继承其他的类。可以得到线程执行的结果	编程相对复杂

目录

Contents

Thread提供了很多与线程操作相关的方法

Thread提供的常用方法	说明
public void run()	线程的任务方法
public void start()	启动线程
public String **getName()**	获取当前线程的名称，线程名称默认是Thread-索引
public void **setName(String name)**	为线程设置名称
public **static** Thread **currentThread()**	获取当前执行的线程对象
public **static** void **sleep(long time)**	让当前执行的线程休眠多少毫秒后，再继续执行
public final void join()...	让调用当前这个方法的线程先执行完！

Thread提供的常见构造器	说明
public Thread(String name)	可以为当前线程指定名称
public Thread(Runnable target)	封装Runnable对象成为线程对象
public Thread(Runnable target, String name)	封装Runnable对象成为线程对象，并指定线程名称

获取线程名称、设置线程名称、拿到当前线程对象

Thread提供的常用方法	说明
public void run()	线程的任务方法
public void start()	启动线程
public String **getName()**	获取当前线程的名称，线程名称默认是Thread-索引
public void **setName(String name)**	为线程设置名称
public **static** Thread **currentThread()**	获取当前执行的线程对象
public **static** void **sleep(long time)**	让当前执行的线程休眠多少毫秒后，再继续执行
public final void join()...	让调用当前这个方法的线程先执行完！

Thread提供的常见构造器	说明
public Thread(String name)	可以为当前线程指定名称
public Thread(Runnable target)	封装Runnable对象成为线程对象
public Thread(Runnable target, String name)	封装Runnable对象成为线程对象，并指定线程名称

线程休眠、线程join

Thread提供的常用方法	说明
public void run()	线程的任务方法
public void start()	启动线程
public String **getName()**	获取当前线程的名称，线程名称默认是Thread-索引
public void **setName(String name)**	为线程设置名称
public **static** Thread **currentThread()**	获取当前执行的线程对象
public **static** void **sleep(long time)**	让当前执行的线程休眠多少毫秒后，再继续执行
public final void join()...	让调用当前这个方法的线程先执行完！

Thread提供的常见构造器	说明
public Thread(String name)	可以为当前线程指定名称
public Thread(Runnable target)	封装Runnable对象成为线程对象
public Thread(Runnable target, String name)	封装Runnable对象成为线程对象，并指定线程名称

Thread其他方法的说明

Thread类还提供了诸如：yield、interrupt、守护线程、线程优先级等线程的控制方法，在开发中很少使用，这些方法会后续需要用到的时候再讲解。

目录

Contents

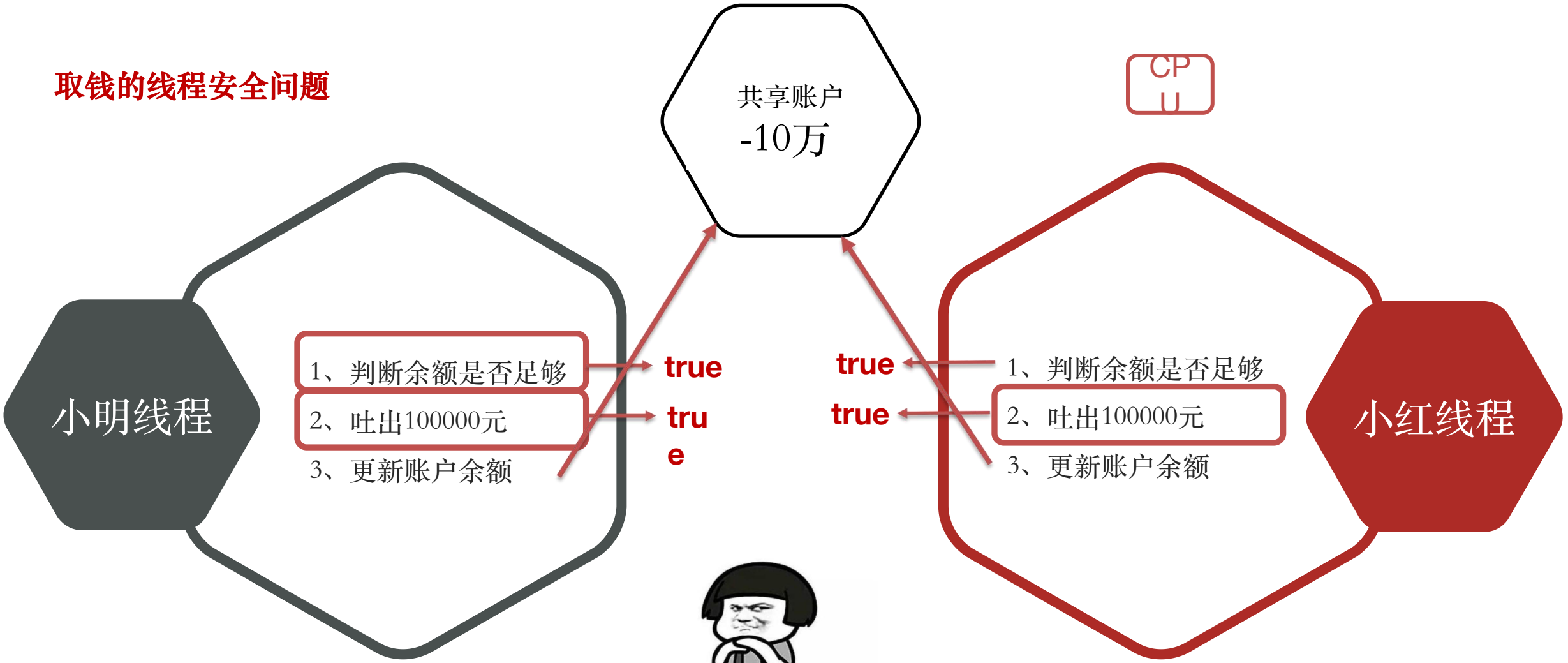
什么是线程安全问题?

- 多个线程，同时操作同一个共享资源的时候，可能会出现业务安全问题。

取钱的线程安全问题

- 场景：小明和小红是一对夫妻，他们有一个共同的账户，余额是10万元，如果小明和小红同时来取钱，并且2人各自都在取钱10万元，可能会出现什么问题呢?

取钱的线程安全问题



我有个大胆的想法

结果：2人都取钱10万，银行亏了10万。

总结

1. 线程安全问题出现的原因？

- **存在多个线程在同时执行**
- **同时访问一个共享资源**
- **存在修改该共享资源**

目录

Contents

案例

取钱案例

需求：

- 小明和小红是一对夫妻，他们有一个共同的账户，余额是10万元，模拟2人同时去取钱10万。

分析：

①：需要提供一个账户类，接着创建一个账户对象代表2个人的共享账户。

②：需要定义一个线程类（用于创建两个线程，分别代表小明和小红）。

③：创建2个线程，传入同一个账户对象给2个线程处理。

④：启动2个线程，同时去同一个账户对象中取钱10万。

总结

1. 线程安全问题发生的原因是什么？

- **多个线程，同时访问同一个共享资源，且存在修改该资源。**

目录

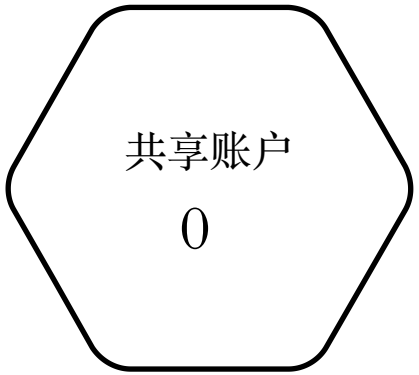
Contents

线程同步

- 解决线程安全问题的方案。

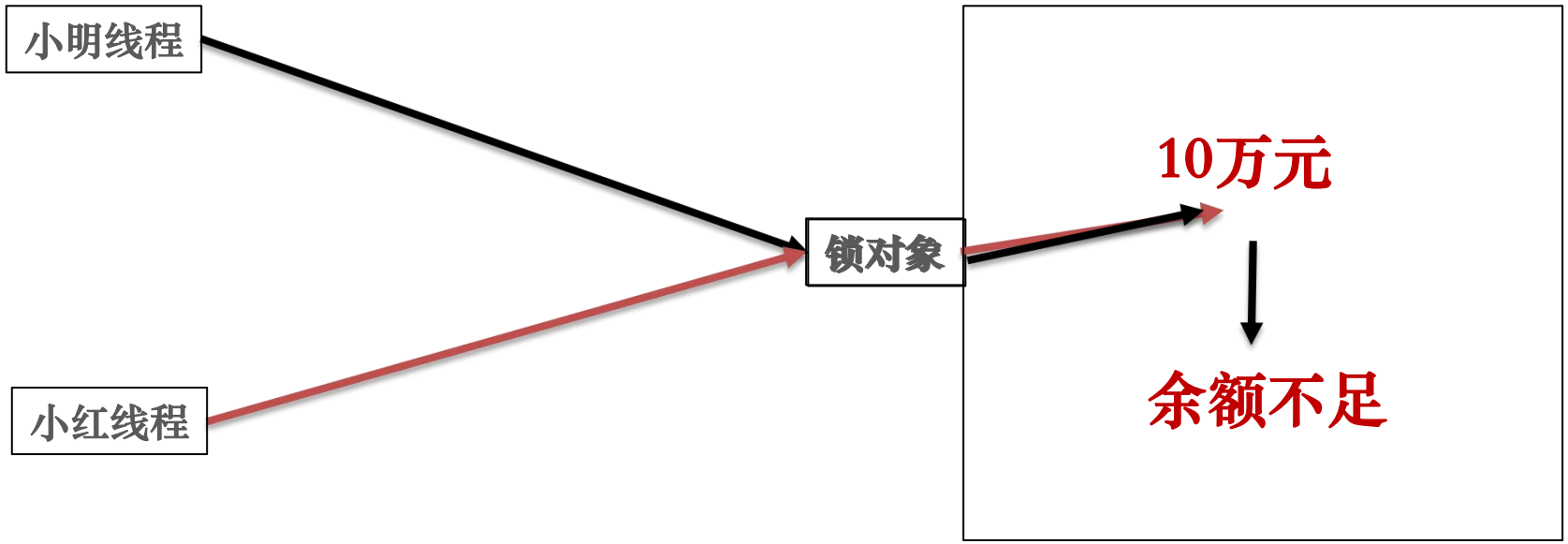
线程同步的思想

- 让多个线程实现先后依次访问共享资源，这样就解决了安全问题。



线程同步的常见方案

- **加锁：**每次只允许一个线程加锁，加锁后才能进入访问，访问完毕后自动解锁，然后其他线程才能再加锁进来。



总结

1. 线程同步解决安全问题的思想是什么？

- **加锁：让多个线程实现先后依次访问共享资源，这样就解决了安全问题。**

目录

Contents

同步代码块

- 作用：把访问共享资源的核心代码给上锁，以此保证线程安全。

```
synchronized(同步锁) {
        访问共享资源的核心代码
}
```

- 原理：每次只允许一个线程加锁后进入，执行完毕后自动解锁，其他线程才可以进来执行。

同步锁的注意事项

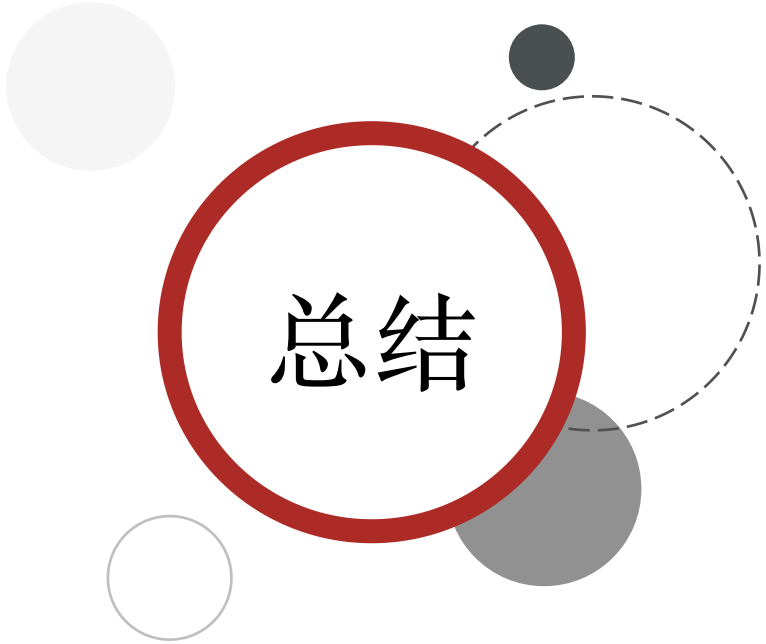
- 对于当前同时执行的线程来说，同步锁必须是同一把（同一个对象），否则会出bug。

锁对象随便选择一个唯一的对象好不好呢?

- 不好，会影响其他无关线程的执行。

锁对象的使用规范

- 建议使用共享资源作为锁对象，对于实例方法建议使用this作为锁对象。
- 对于静态方法建议使用字节码（类名.class）对象作为锁对象。



1. 同步代码块是如何实现线程安全的？

   - **对出现问题的核心代码使用synchronized进行加锁**
   - **每次只能一个线程占锁进入访问**

2. 同步代码块的同步锁对象有什么要求？

   - **对于实例方法建议使用this作为锁对象。**
   - **对于静态方法建议使用字节码（类名.class）对象作为锁对象。**

目录

Contents

同步方法

- 作用：把访问共享资源的核心方法给上锁，以此保证线程安全。

```
修饰符 synchronized 返回值类型 方法名称(形参列表) {
        操作共享资源的代码
}
```

- 原理：每次只能一个线程进入，执行完毕以后自动解锁，其他线程才可以进来执行

同步方法底层原理

- 同步方法其实底层也是有隐式锁对象的，只是锁的范围是整个方法代码。
- 如果方法是实例方法：同步方法默认用this作为的锁对象。
- 如果方法是静态方法：同步方法默认用类名.class作为的锁对象。

1、是同步代码块好还是同步方法好一点？

- **范围上：同步代码块锁的范围更小，同步方法锁的范围更大。**
- **可读性：同步方法更好。**

总结

1. 同步方法是如何保证线程安全的？

   - **对出现问题的核心方法使用synchronized修饰**
   - **每次只能一个线程占锁进入访问**

2. 同步方法的同步锁对象的原理？

   - **对于实例方法默认使用this作为锁对象。**
   - **对于静态方法默认使用类名.class对象作为锁对象。**

- **多线程的创建**
- **Thread的常用方法**
- **线程安全**
- **线程同步**
  - 认识线程同步
  - 方式一：同步代码块
  - 方式二：同步方法
  - 方式三：Lock锁
- **线程通信**
- **线程池**
- **其它细节知识：并发、并行**
- **其它细节知识：线程的生命周期**

Lock锁

- Lock锁是JDK5开始提供的一个新的锁定操作，通过它可以创建出锁对象进行加锁和解锁，更灵活、更方便、更强大。
- Lock是接口，不能直接实例化，可以采用它的实现类ReentrantLock来构建Lock锁对象。

构造器	说明
public ReentrantLock()	获得Lock锁的实现类对象

Lock的常用方法

方法名称	说明
void lock()	获得锁
void unlock()	释放锁

- 多线程的创建
- Thread的常用方法
- 线程安全
- 线程同步
- 线程通信[了解]
- 线程池
- 其它细节知识：并发、并行
- 其它细节知识：线程的生命周期

什么是线程通信?

- 当多个线程共同操作共享的资源时，线程间通过某种方式互相告知自己的状态，以相互协调，并避免无效的资源争夺。

线程通信的常见模型（生产者与消费者模型

- 生产者线程负责生产数据
- 消费者线程负责消费生产者生产的数据。
- 注意：生产者生产完数据应该等待自己，通知消费者消费；消费者消费完数据也应该等待自己，再通知生产者生产！

线程通信的模型

3个生产者线程，负责生产包子，每个线程每次只能生产1个包子放在桌子上

2个消费者线程负责吃包子，每人每次只能从桌子上拿1个包子吃。

生产者线程

消费者线程

Object类的等待和唤醒方法：

方法名称	说明
void wait()	让当前线程等待并释放所占锁，直到另一个线程调用notify()方法或 notifyAll()方法
void notify()	唤醒正在等待的单个线程
void notifyAll()	唤醒正在等待的所有线程

注意

- 上述方法应该使用当前同步锁对象进行调用。

总结

1. 线程通信的三个常见方法?

方法名称	说明
void wait()	当前线程等待，直到另一个线程调用notify() 或 notifyAll()唤醒自己
void notify()	唤醒正在等待对象监视器(锁对象)的单个线程
void notifyAll()	唤醒正在等待对象监视器(锁对象)的所有线程

注意

- 上述方法应该使用当前同步锁对象进行调用。

- **多线程的创建**
- **Thread的常用方法**
- **线程安全**
- **线程同步**
- **线程通信**
- **线程池**
  - 认识线程池
  - 如何创建线程池？
  - 线程池处理Runnable任务
  - 线程池处理Callable任务
  - Executors工具类实现线程池
- **其它细节知识：并发、并行**
- **其它细节知识：线程的生命周期**

什么是线程池？

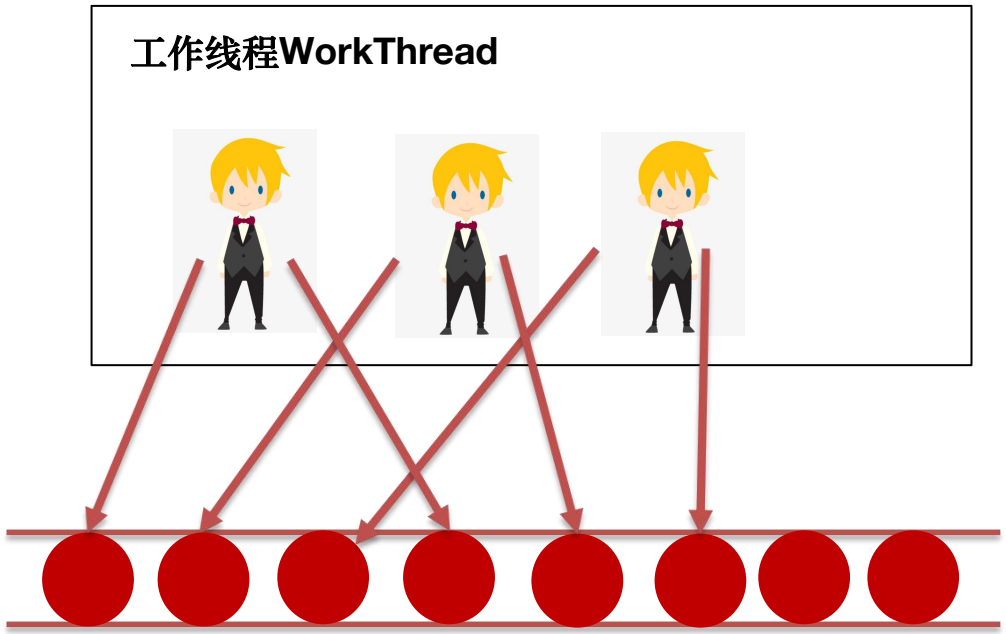
- 线程池就是一个可以复用线程的技术。

不使用线程池的问题

- 用户每发起一个请求，后台就需要创建一个新线程来处理，下次新任务来了肯定又要创建新线程处理的， 而创建新线程的开销是很大的，并且请求过多时，肯定会产生大量的线程出来，这样会严重影响系统的性能。

线程池的工作原理

线程池



任务接口

- Runnable
- Callable

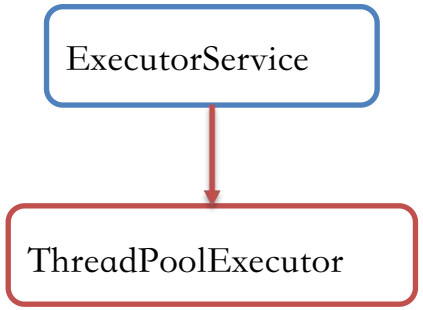
- **多线程的创建**
- **Thread的常用方法**
- **线程安全**
- **线程同步**
- **线程通信**
- **线程池**
  - 认识线程池
  - 如何创建线程池？
  - 线程池处理Runnable任务
  - 线程池处理Callable任务
  - Executors工具类实现线程池
- **其它细节知识：并发、并行**
- **其它细节知识：线程的生命周期**

谁代表线程池?

- JDK 5.0起提供了代表线程池的接口：ExecutorService。

如何得到线程池对象?

- 方式一：使用ExecutorService的实现类ThreadPoolExecutor自创建一个线程池对象。



- 方式二：使用Executors（线程池的工具类）调用方法返回不同特点的线程池对象。

ThreadPoolExecutor构造器

```
public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit,
                          BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

- 参数一：corePoolSize：指定线程池的核心线程的数量。 正式工：3
- 参数二：maximumPoolSize：指定线程池的最大线程数量。 最大员工数：5 临时工：2
- 参数三：keepAliveTime ：指定临时线程的存活时间。 临时工空闲多久被开除
- 参数四：unit：指定临时线程存活的时间单位(秒、分、时、天)
- 参数五：workQueue：指定线程池的任务队列。 客人排队的地方
- 参数六：threadFactory：指定线程池的线程工厂。 负责招聘员工的（hr）
- 参数七：handler：指定线程池的任务拒绝策略（线程都在忙，任务队列也满了的时候，新任务来了该怎么处理） 忙不过来咋办?

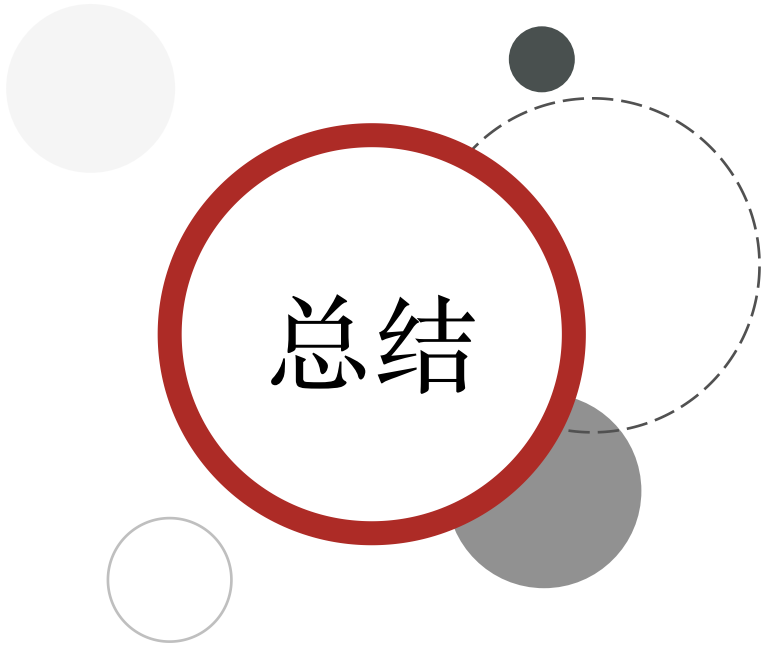
线程池的注意事项

1、临时线程什么时候创建?

- 新任务提交时发现核心线程都在忙，任务队列也满了，并且还可以创建临时线程，此时才会创建临时线程。

2、什么时候会开始拒绝新任务?

- 核心线程和临时线程都在忙，任务队列也满了，新的任务过来的时候才会开始拒绝任务。



1. 谁代表线程池？线程池的创建方案有几种？
   - **ExecutorService接口**
2. ThreadPoolExecutor实现线程池对象的七个参数是什么意思？
   - **使用线程池的实现类ThreadPoolExecutor**

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

- **多线程的创建**
- **Thread的常用方法**
- **线程安全**
- **线程同步**
- **线程通信**
- **线程池**
  - 认识线程池
  - 如何创建线程池？
  - 线程池处理Runnable任务
  - 线程池处理Callable任务
  - Executors工具类实现线程池
- **其它细节知识：并发、并行**
- **其它细节知识：线程的生命周期**

ExecutorService的常用方法

方法名称	说明
void execute(Runnable command)	执行 Runnable 任务
Future<T> submit(Callable<T> task)	执行 Callable 任务，返回未来任务对象，用于获取线程返回的结果
void shutdown()	等全部任务执行完毕后，再关闭线程池！
List<Runnable> shutdownNow()	立刻关闭线程池，停止正在执行的任务，并返回队列中未执行的任务

新任务拒绝策略

策略	详解
ThreadPoolExecutor.AbortPolicy	丢弃任务并抛出RejectedExecutionException异常。**是默认的策略**
ThreadPoolExecutor.DiscardPolicy：	丢弃任务，但是不抛出异常 这是不推荐的做法
ThreadPoolExecutor.DiscardOldestPolicy	抛弃队列中等待最久的任务 然后把当前任务加入队列中
ThreadPoolExecutor.CallerRunsPolicy	由主线程负责调用任务的run()方法从而绕过线程池直接执行

总结

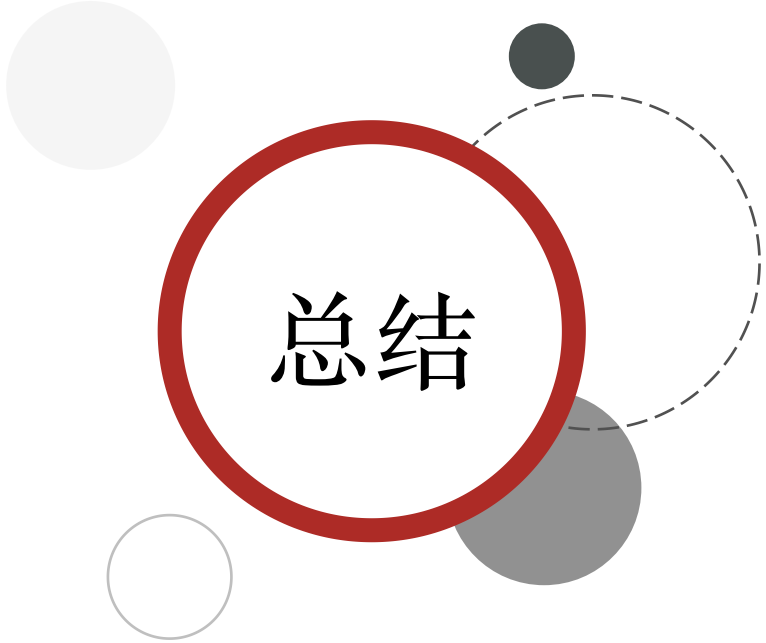
1. 线程池如何处理Runnable任务？

- **使用ExecutorService的方法：**
- **void execute(Runnable target)**

- **多线程的创建**
- **Thread的常用方法**
- **线程安全**
- **线程同步**
- **线程通信**
- **线程池**
  - 认识线程池
  - 如何创建线程池？
  - 线程池处理Runnable任务
  - 线程池处理Callable任务
  - Executors工具类实现线程池
- **其它细节知识：并发、并行**
- **其它细节知识：线程的生命周期**

ExecutorService的常用方法

方法名称	说明
void execute(Runnable command)	执行任务/命令，没有返回值，一般用来执行 Runnable 任务
Future<T> submit(Callable<T> task)	执行任务，返回未来任务对象获取线程结果，一般拿来执行 Callable 任务
void shutdown()	等任务执行完毕后关闭线程池
List<Runnable> shutdownNow()	立刻关闭，停止正在执行的任务，并返回队列中未执行的任务



1. 线程池如何处理Callable任务，并得到任务执行完后返回的结果？
   - **使用ExecutorService的方法：**
   - **Future<T> submit(Callable<T> command)**

- **多线程的创建**
- **Thread的常用方法**
- **线程安全**
- **线程同步**
- **线程通信**
- **线程池**
  - 认识线程池
  - 如何创建线程池？
  - 线程池处理Runnable任务
  - 线程池处理Callable任务
  - Executors工具类实现线程池

实现类：ThreadPoolExecutor

工具类：Executors

- **其它细节知识：并发、并行**
- **其它细节知识：线程的生命周期**

Executors

- 是一个线程池的工具类，提供了很多静态方法用于返回不同特点的线程池对象。

方法名称	说明
public static ExecutorService newFixedThreadPool(int nThreads)	创建固定线程数量的线程池，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程替代它。
public static ExecutorService newSingleThreadExecutor()	创建只有一个线程的线程池对象，如果该线程出现异常而结束，那么线程池会补充一个新线程。
public static ExecutorService newCachedThreadPool()	线程数量随着任务增加而增加，如果线程任务执行完毕且空闲了60s则会被回收掉。
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)	创建一个线程池，可以实现在给定的延迟后运行任务，或者定期执行任务。

注意：这些方法的底层，都是通过线程池的实现类ThreadPoolExecutor创建的线程池对象。

Executors使用可能存在的陷阱

- 大型并发系统环境中使用Executors如果不注意可能会出现系统风险。

阿里巴巴 Java 开发手册

4．【强制】线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

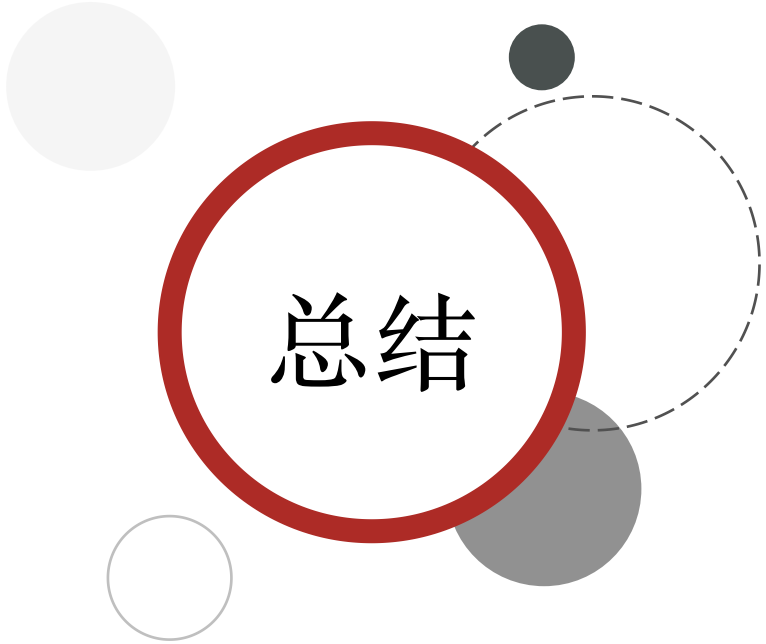
说明：Executors 返回的线程池对象的弊端如下：

1）FixedThreadPool 和 SingleThreadPool:

允许的请求队列长度为 Integer.MAX_VALUE，可能会堆积大量的请求，从而导致 OOM。

2）CachedThreadPool 和 ScheduledThreadPool:

允许的创建线程数量为 Integer.MAX_VALUE，可能会创建大量的线程，从而导致 OOM。



1. Executors工具类底层是基于什么方式实现的线程池对象？
   - **线程池ExecutorService的实现类：ThreadPoolExecutor**

2. Executors是否适合做大型互联网场景的线程池方案？
   - **不合适。**
   - **建议使用ThreadPoolExecutor来指定线程池参数，这样可以明确线程池的运行规则，规避资源耗尽的风险。**

- 多线程的创建
- Thread的常用方法
- 线程安全
- 线程同步
- 线程通信
- 线程池
- **其它细节知识：并发、并行**
- 其它细节知识：线程的生命周期

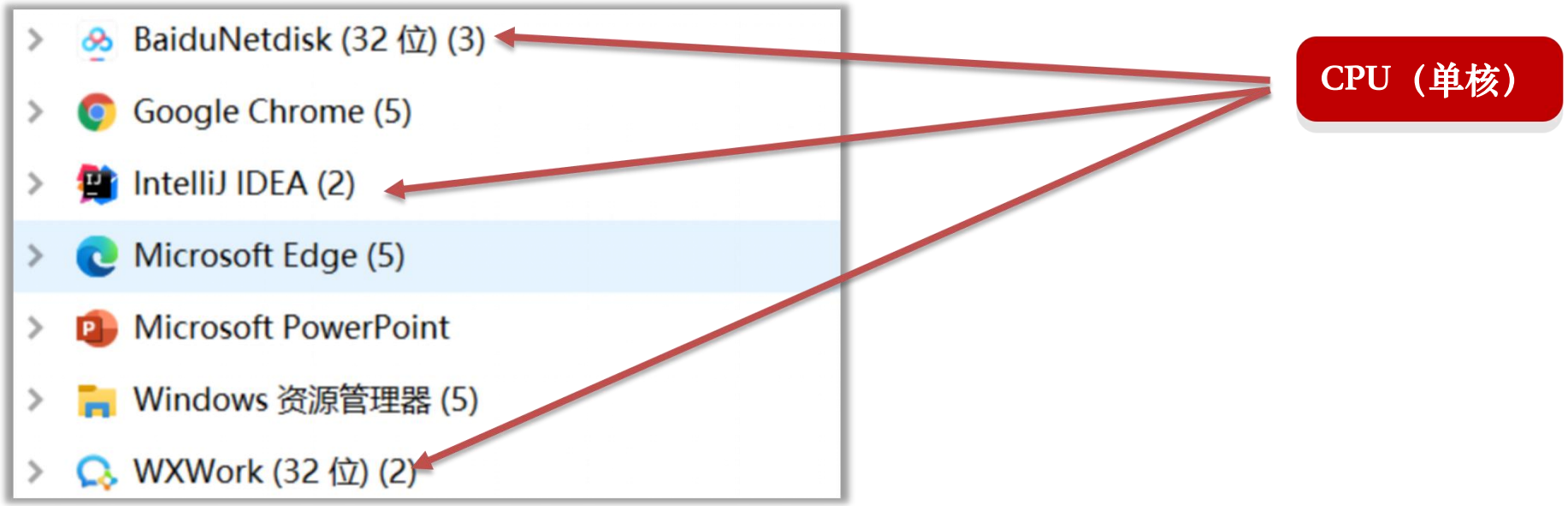
进程

- 正在运行的程序（软件）就是一个独立的进程。
- 线程是属于进程的，一个进程中可以同时运行很多个线程。
- 进程中的多个线程其实是并发和并行执行的。

并发的含义

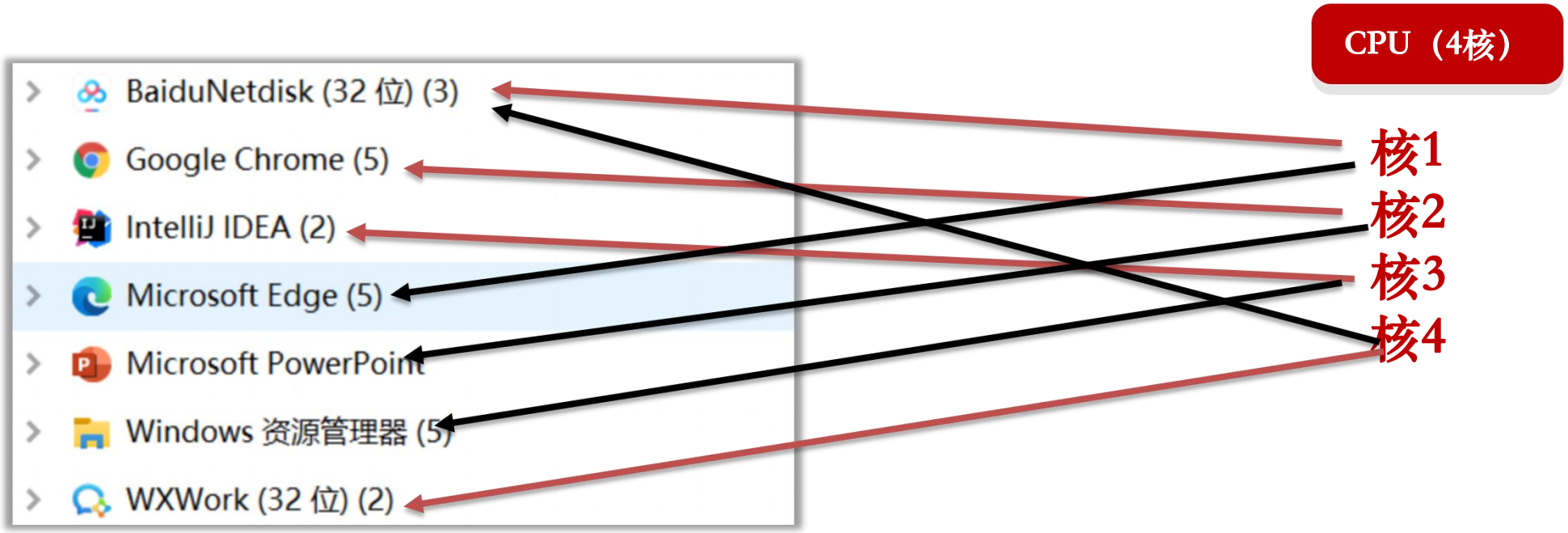
- 进程中的线程是由CPU负责调度执行的，但CPU能同时处理线程的数量有限，为了保证全部线程都能往前执行，

  CPU会轮询为系统的每个线程服务，由于CPU切换的速度很快，给我们的感觉这些线程在同时执行，这就是并发。



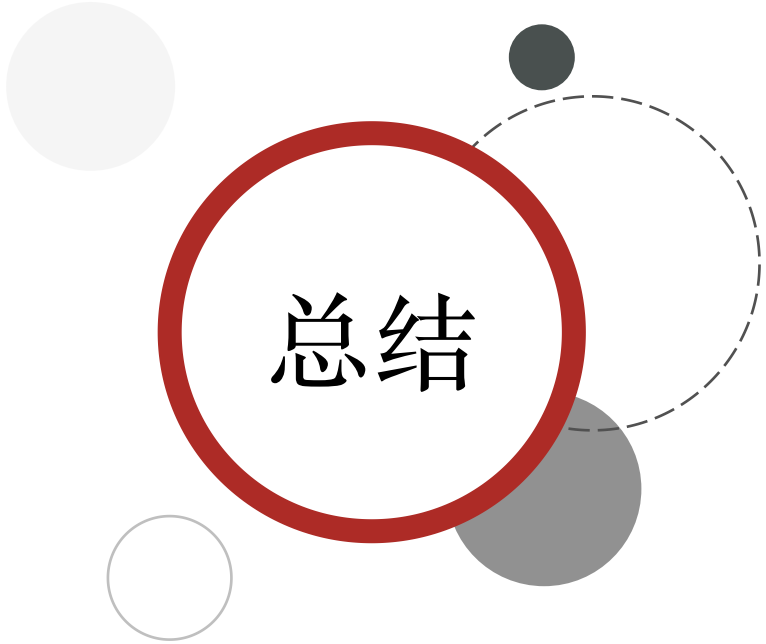
并行的理解

- 在同一个时刻上，同时有多个线程在被CPU调度执行。



多线程到底是怎么在执行的？

并发和并行同时进行的！



1. 简单说说多线程是怎么执行的？

- **并发：CPU分时轮询的执行线程。**
- **并行：同一个时刻同时在执行。**

- 多线程的创建
- Thread的常用方法
- 线程安全
- 线程同步
- 线程通信
- 线程池
- 其它细节知识：并发、并行
- **其它细节知识：线程的生命周期**

人的生命周期（各种状态）

意外状态

婴儿

单身

婚后

GG

线程的生命周期

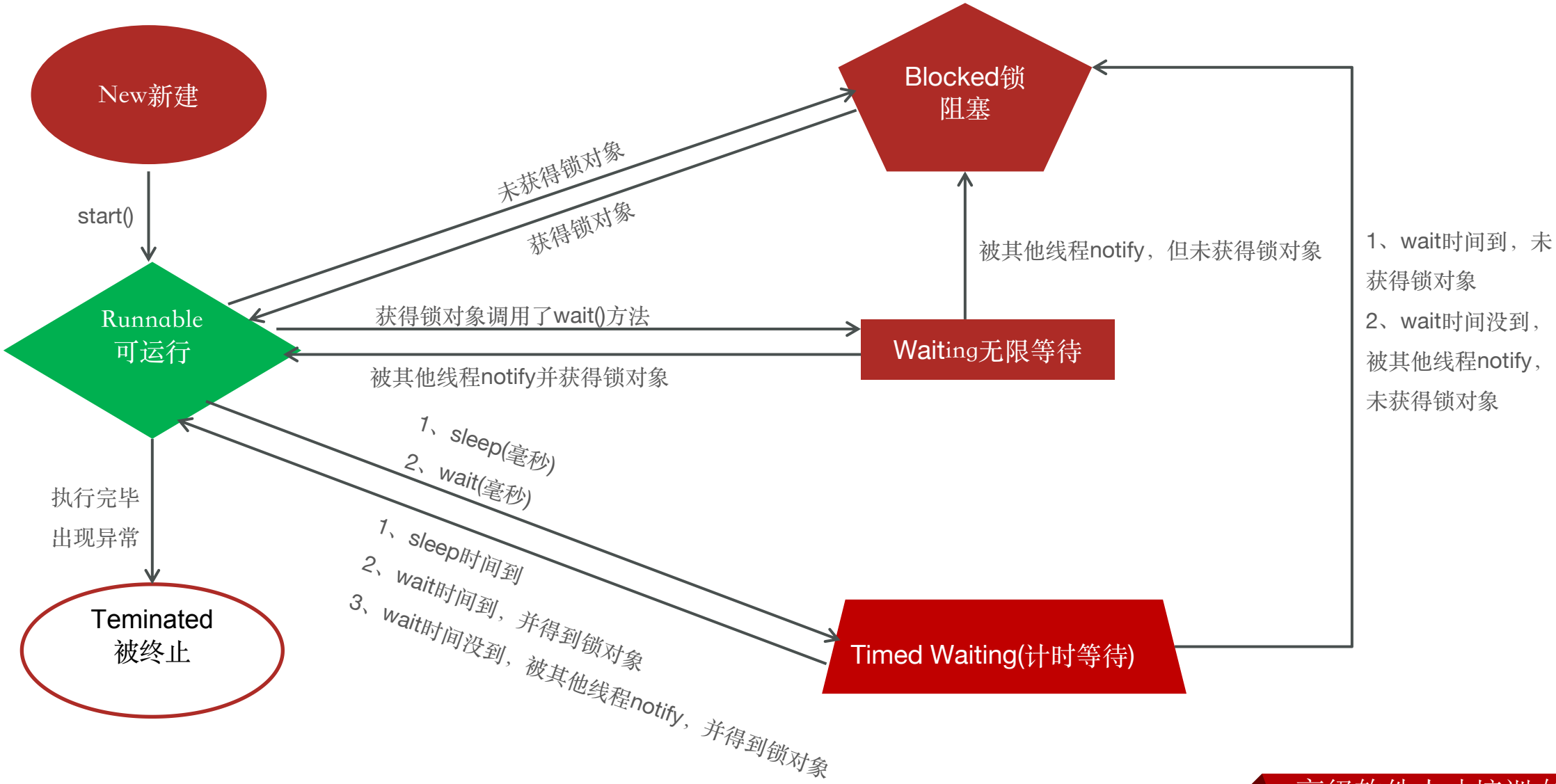
- 也就是线程从生到死的过程中，经历的各种状态及状态转换。
- 理解线程这些状态有利于提升并发编程的理解能力。

Java线程的状态

- Java总共定义了6种状态
- 6种状态都定义在Thread类的内部枚举类中。

```
public class Thread{
    ...
    public enum State {
            NEW,
            RUNNABLE,
            BLOCKED,
            WAITING,
            TIMED_WAITING,
            TERMINATED;
    }
    ...
}
```

线程的6种状态互相转换



线程的6种状态总结

线程状态	说明
NEW(新建)	线程刚被创建，但是并未启动。
Runnable(可运行)	线程已经调用了start()，等待CPU调度
Blocked(锁阻塞)	线程在执行的时候未竞争到锁对象，则该线程进入Blocked状态；。
Waiting(无限等待)	一个线程进入Waiting状态，另一个线程调用notify或者notifyAll方法才能够唤醒
Timed Waiting(计时等待)	同waiting状态，有几个方法（sleep,wait）有超时参数，调用他们将进入Timed Waiting状态。
Teminated(被终止)	因为run方法正常退出而死亡，或者因为没有捕获的异常终止了run方法而死亡。