@来源 并发编程面试题（2021优化版）

基础知识

并发编程的优缺点

为什么要使用并发编程（并发编程的优点）

- 充分利用多核CPU的计算能力：通过并发编程的形式可以将多核CPU的计算能力发挥到极致，性能得到提升
- 方便进行业务拆分，提升系统并发能力和性能

并发编程有什么缺点

并发编程的目的就是为了能提高程序的执行效率，提高程序运行速度，但是并发编程并不总是能提高程序运行速度的，而且并发编程可能会遇到很多问题，比如：**内存泄漏**、**上下文切换**、**线程安全**、**死锁**等问题。

什么叫线程安全？servlet 是线程安全吗?

线程安全是编程中的术语，指某个方法在多线程环境中被调用时，能够正确地处理多个线程之间的共享变量，使程序功能正确完成。

Servlet 不是线程安全的，servlet 是**单实例多线程**的，当多个线程同时访问同一个方法，是不能保证共享变量的线程安全性的。

Struts2 的 action 是多实例多线程的，是线程安全的，每个请求过来都会 new 一个新的 action 分配给这个请求，请求完成后销毁。

SpringMVC 的 Controller 是线程安全的吗？不是的，和 Servlet 类似的处理流程。

Struts2 好处是不用考虑线程安全问题；**Servlet** 和 **SpringMVC** 需要考虑线程安全问题，但是性能可以提升，不用处理太多的 gc，可以**使用 ThreadLocal 来处理多线程**的问题。

@$并发编程三要素是什么？在 Java 程序中怎么保证多线程的运行安全?

并发编程三要素（线程的安全性问题体现在）：

- **原子性**：一个或多个操作要么全部执行成功要么全部执行失败。
- **可见性**：一个线程对共享变量的修改，另一个线程能够立刻看到。
- **有序性**：程序执行的顺序按照代码的先后顺序执行，避免指令重排。

出现线程安全问题的原因：

- **线程切换**带来的原子性问题
- **缓存**导致的可见性问题
- **编译优化**带来的有序性问题

解决办法：

- **JDK Atomic开头的原子类**、**synchronized**、**lock**，可以解决原子性问题
- **volatile**、**synchronized**、**lock**，可以解决可见性问题
- **volatile**、**Happens-Before 规则**可以解决有序性问题

并行和并发有什么区别?

- 串行：多个任务在一个线程上按顺序执行。由于任务都在一个线程执行所以不存在线程不安全情况，也就不存在临界区的问题。
- 并发：多个任务在一个 CPU 核上按细分的时间片轮流(交替)执行，从逻辑上来看那些任务是同时执行。
- 并行：单位时间内，多个 CPU 同时处理多个任务，是真正意义上的“同时进行”。

做一个形象的比喻：

串行 = 一个队列和一台咖啡机。

并发 = 两个队列和一台咖啡机。

并行 = 两个队列和两台咖啡机。

线程和进程的区别

什么是线程和进程?

进程

一个在内存中运行的应用程序。每个进程都有自己独立的一块内存空间，一个进程可以有多个线程，比如在Windows系统中，一个运行的xx.exe就是一个进程。

线程

进程中的一个执行任务（控制单元），负责当前进程中程序的执行。一个进程至少有一个线程，一个进程可以运行多个线程，多个线程可共享数据。

@$进程与线程的区别

线程具有许多传统进程所具有的特征，故又称为轻型进程(Light—Weight Process)或进程元；而把传统的进程称为重型进程(Heavy—Weight Process)。在引入了线程的操作系统中，通常一个进程都有若干个线程，至少包含一个线程。

根本区别：进程是操作系统资源分配的基本单位，而线程是处理器任务调度和执行的基本单位。

资源开销和内存分配：每个进程都有独立的代码和数据空间（程序上下文），进程之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。

包含关系：线程是进程的一部分。一个进程至少有一个线程，一个进程可以运行多个线程。

影响关系：一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程都死掉。所以多进程要比多线程健壮。

执行过程：每个独立的进程有程序入口、顺序执行序列和程序出口。线程不能独立执行，必须依存在进程中，由进程提供多个线程执行控制，两者均可并发执行。

什么是上下文切换?

概括来说就是：当前任务在执行完 CPU 时间片切换到另一个任务之前会先保存自己的状态，以便下次再切换回这个任务时，可以再加载这个任务的状态。**任务从保存到再加载的过程就是一次上下文切换。**

守护线程和用户线程有什么区别呢?

守护线程和用户线程

- **用户 (User) 线程**：运行在前台，执行具体的任务，如程序的主线程、连接网络的子线程等都是用户线程
- **守护 (Daemon) 线程**：运行在后台，为其他前台线程服务。也可以说守护线程是 JVM 中非守护线程的 “佣人”。一旦所有用户线程都结束运行，守护线程会随 JVM 一起结束工作

main 函数所在的线程就是一个用户线程，main 函数启动的同时在 JVM 内部同时还启动了好多守护线程，比如垃圾回收线程。

比较明显的区别之一是用户线程结束，JVM 退出，不管这个时候有没有守护线程运行。而守护线程不会影响 JVM 的退出。

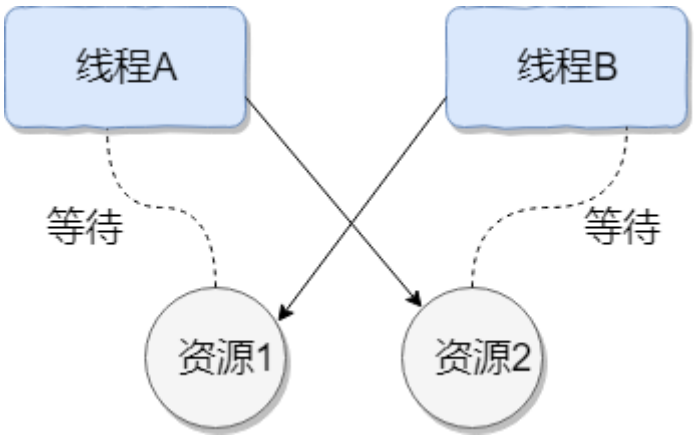
注意事项:

1. `setDaemon(true)`必须在`start()`方法前执行，否则会抛出`IllegalThreadStateException`异常
2. 在守护线程中产生的新线程也是守护线程
3. 不是所有的任务都可以分配给守护线程来执行，比如读写操作或者计算逻辑
4. 守护 (Daemon) 线程中不能依靠 finally 块来确保执行关闭或清理资源的逻辑。因为一旦所有用户线程都结束运行，守护线程会随 JVM 一起结束工作，所以守护 (Daemon) 线程中的 finally 语句块可能无法被执行。

@$什么是线程死锁

百度百科：死锁是指两个或两个以上的进程（线程）在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程（线程）称为死锁进程（线程）。

如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。



下面通过一个例子来说明线程死锁，代码模拟了上图的死锁的情况 (代码来源于《并发编程之美》):

```
public class DeadLockDemo {
   private static Object resource1 = new Object();//资源 1
   private static Object resource2 = new Object();//资源 2

   public static void main(String[] args) {
       new Thread(() -> {
           synchronized (resource1) {
               System.out.println(Thread.currentThread() + "get resource1");
               try {
```

```
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                 System.out.println(Thread.currentThread() + "waiting get resource2");
                 synchronized (resource2) {
                    System.out.println(Thread.currentThread() + "get resource2");
                }
            }
        }, "线程 1").start();

         new Thread(() -> {
             synchronized (resource2) {
                 System.out.println(Thread.currentThread() + "get resource2");
                 try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                 System.out.println(Thread.currentThread() + "waiting get resource1");
                 synchronized (resource1) {
                    System.out.println(Thread.currentThread() + "get resource1");
                }
            }
        }, "线程 2").start();
    }
}
```

输出结果

```
Thread[线程 1,5,main]get resource1
Thread[线程 2,5,main]get resource2
Thread[线程 1,5,main]waiting get resource2
Thread[线程 2,5,main]waiting get resource1
```

线程 A 通过 synchronized (resource1) 获得 resource1 的监视器锁，然后通过`Thread.sleep(1000)`方法让线程 A 休眠 1s ，为的是让线程 B 得到CPU执行权，然后获取到 resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也就产生了死锁。上面的例子符合产生死锁的四个必要条件。

@$形成死锁的四个必要条件是什么

1. **互斥条件**：线程(进程)对于所分配到的资源具有排它性，即一个资源只能被一个线程(进程)占用，直到被该线程(进程)释放
2. **请求与保持条件**：一个线程(进程)因请求被占用资源而发生阻塞时，对已获得的资源保持不放。
3. **不剥夺条件**：线程(进程)已获得的资源在未使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。
4. **循环等待条件**：当发生死锁时，所等待的线程(进程)必定会形成一个环路（类似于死循环），造成永久阻塞

@$如何避免线程死锁

我们只要破坏产生死锁的四个条件中的其中一个就可以了。

破坏互斥条件

这个条件我们没有办法破坏，因为我们用锁本来就是想让他们互斥的（临界资源需要互斥访问）。

破坏请求与保持条件

一次性申请所有的资源，会降低并发能力，一般不破坏请求与保持条件。

破坏不剥夺条件

占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。

破坏循环等待条件

靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。

我们对线程 2 的代码修改成下面这样就不会产生死锁了。

```
new Thread(() -> {
   synchronized (resource1) {
       System.out.println(Thread.currentThread() + "get resource1");
       try {
           Thread.sleep(1000);
       } catch (InterruptedException e) {
           e.printStackTrace();
       }
       System.out.println(Thread.currentThread() + "waiting get resource2");
       synchronized (resource2) {
           System.out.println(Thread.currentThread() + "get resource2");
       }
  }
}, "线程 2").start();
```

输出结果

```
Thread[线程 1,5,main]get resource1
Thread[线程 1,5,main]waiting get resource2
Thread[线程 1,5,main]get resource2
Thread[线程 2,5,main]get resource1
Thread[线程 2,5,main]waiting get resource2
Thread[线程 2,5,main]get resource2
```

我们分析一下上面的代码为什么避免了死锁的发生?

线程 1 首先获得到 resource1 的监视器锁，这时候线程 2 就获取不到了。然后线程 1 再去获取 resource2 的监视器锁，可以获取到。然后线程 1 释放了对 resource1、resource2 的监视器锁的占用，线程 2 获取到就可以执行了。这样就破坏了破坏循环等待条件，因此避免了死锁。

创建线程的三种方式

创建线程有哪几种方式?

创建线程有三种方式：

- 继承 Thread 类；
- 实现 Runnable 接口；
- 实现 Callable 接口；

继承 Thread 类

步骤

1. 定义一个Thread类的子类，重写run方法，将相关逻辑实现，run()方法就是线程要执行的业务逻辑方法
2. 创建自定义的线程子类对象
3. 调用子类实例的star()方法来启动线程

```
public class MyThread extends Thread {

   @Override
   public void run() {
       System.out.println(Thread.currentThread().getName() + " run()方法正在执行...");
  }

}
```

```
public class TheadTest {

   public static void main(String[] args) {
       MyThread myThread = new MyThread();
       myThread.start();
       System.out.println(Thread.currentThread().getName() + " main()方法执行结束");
  }

}
```

运行结果

```
main main()方法执行结束
Thread-0 run()方法正在执行...
```

实现 Runnable 接口

步骤

1. 定义Runnable接口实现类MyRunnable，并重写run()方法
2. 创建MyRunnable实例myRunnable，以myRunnable作为target创建Thead对象，该Thread对象才是真正的线程对象
3. 调用线程对象的start()方法

```
public class MyRunnable implements Runnable {

   @Override
   public void run() {
       System.out.println(Thread.currentThread().getName() + " run()方法执行中...");
  }

}
```

```
public class RunnableTest {

    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        thread.start();
        System.out.println(Thread.currentThread().getName() + " main()方法执行完成");
    }

}
```

执行结果

```
main main()方法执行完成
Thread-0 run()方法执行中...
```

实现 Callable 接口

步骤

1. 创建实现Callable接口的类myCallable
2. 以myCallable为参数创建FutureTask对象
3. 将FutureTask作为参数创建Thread对象
4. 调用线程对象的start()方法

```
public class MyCallable implements Callable<Integer> {

    @Override
    public Integer call() {
        System.out.println(Thread.currentThread().getName() + " call()方法执行中...");
        return 1;
    }

}
```

```
public class CallableTest {

    public static void main(String[] args) {
        FutureTask<Integer> futureTask = new FutureTask<Integer>(new MyCallable());
        Thread thread = new Thread(futureTask);
        thread.start();

```

```
        try {
            Thread.sleep(1000);
            System.out.println("返回结果 " + futureTask.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " main()方法执行完成");
    }

}
```

执行结果

```
Thread-0 call()方法执行中...
返回结果 1
main main()方法执行完成
```

说一下 runnable 和 callable 有什么区别?

相同点

- 都是接口
- 都可以编写多线程程序
- 都采用Thread.start()启动线程

主要区别

- Runnable 接口 run 方法**无返回值**；Callable 接口 call 方法**有返回值**，是个泛型，和Future、FutureTask配合可以用来获取异步执行的结果
- Runnable 接口 **run 方法只能抛出运行时异常，且无法捕获处理**；Callable 接口 **call 方法允许抛出异常，可以获取异常信息**

注：Callalbe接口支持返回执行结果，需要调用FutureTask.get()得到，此方法会阻塞主进程的继续往下执行，如果不调用不会阻塞。

线程的 run()和 start()有什么区别？为什么我们调用 start() 方法时会执行 run() 方法，为什么我们不能直接调用 run() 方法?

每个线程都是通过某个特定Thread对象所对应的方法run()来完成其操作的，run()方法称为线程体。通过调用Thread类的start()方法来启动一个线程。也就是说start() 方法用于启动线程，run() 方法用于执行线程任务。run() 可以重复调用，而 start() 只能调用一次。

start()方法来启动一个线程，真正实现了多线程运行。调用start()方法无需等待run方法体代码执行完毕，可以直接继续执行其他的代码；此时线程是处于就绪状态，并没有运行。然后通过此Thread类调用方法run()来完成其运行状态， run()方法运行结束， 此线程终止。然后CPU再调度其它线程。

run()方法是在本线程里的，只是线程里的一个函数，而不是多线程的。如果直接调用run()，其实就相当于是调用了一个普通函数而已，直接调用run()方法必须等待run()方法执行完毕才能执行下面的代码，所以执行路径还是只有一条，根本就没有线程的特征，所以在多线程执行时要使用start()方法而不是run()方法。

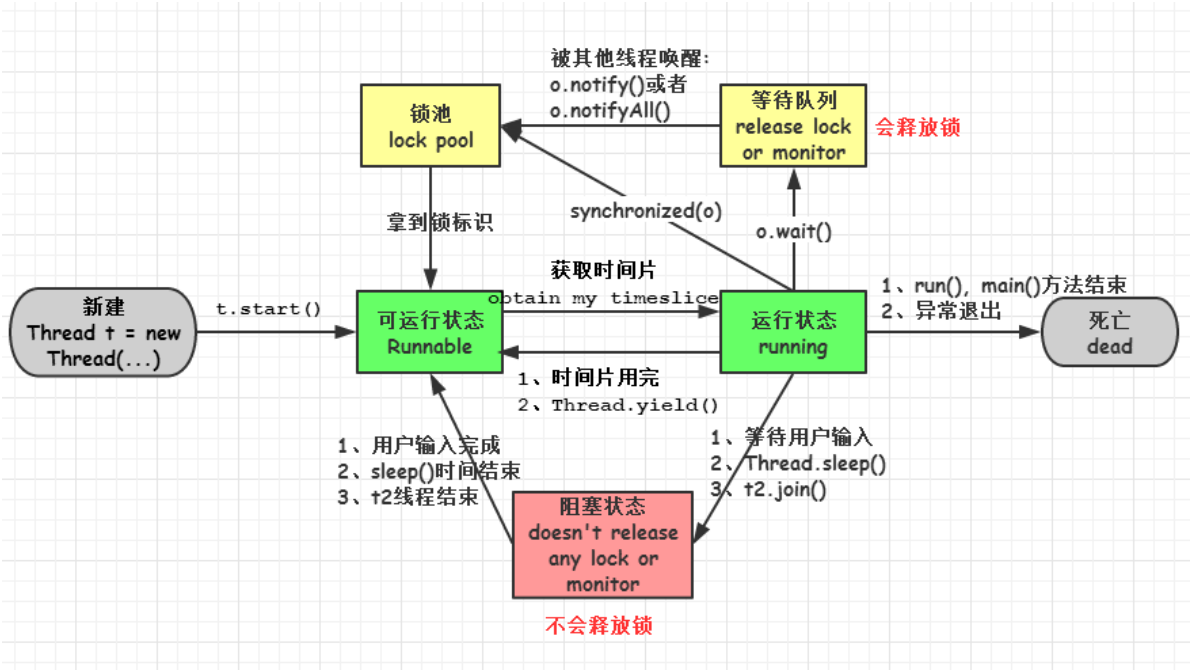
什么是 Future 和 FutureTask?

Future 接口表示异步计算的任务，他提供了判断任务是否完成，中断任务，并可以通过get方法获取任务执行结果，该方法会阻塞直到任务返回结果。因为Future只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的FutureTask。

FutureTask 类间接实现了 Future 接口，可以看出RunnableFuture继承了Runnable接口和Future接口，而FutureTask实现了RunnableFuture接口。所以它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值。

线程的状态和基本操作

@$说说线程的生命周期及五种基本状态?



1. **新建(new)**：新创建了一个线程对象。
2. **可运行(runnable)**：线程对象创建后，当调用线程对象的 start()方法，该线程处于就绪状态，等待被线程调度选中，获取cpu的使用权。
3. **运行(running)**：可运行状态(runnable)的线程获得了cpu时间片（timeslice），执行程序代码。
   注：就绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；
4. **阻塞(block)**：处于运行状态中的线程由于某种原因，暂时放弃对 CPU的使用权，停止执行，此时进入阻塞状态，直到其再次进入到就绪状态，才有机会再次被 CPU 调用以进入到运行状态。

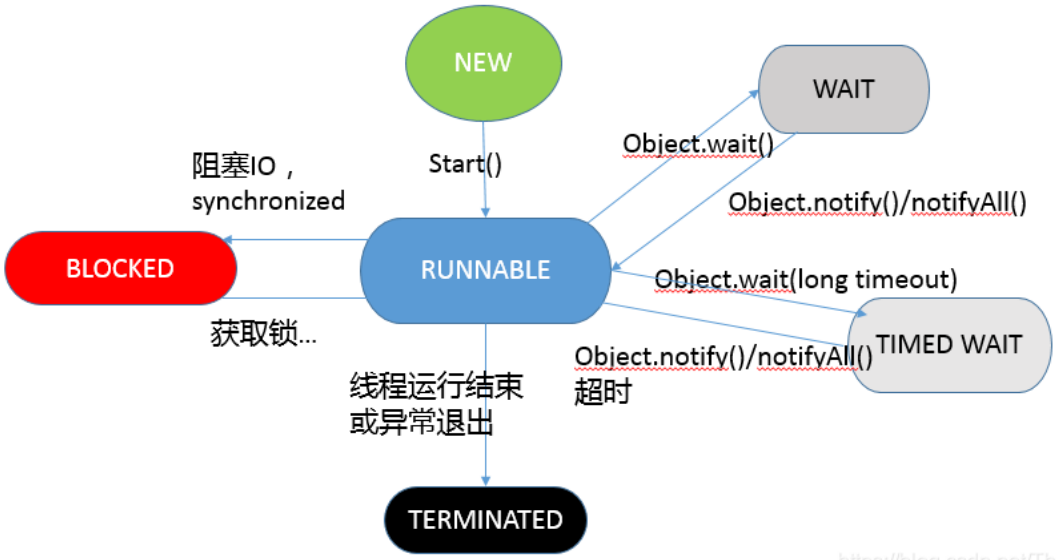
   阻塞的情况分三种：

   (一). 等待阻塞：运行状态中的线程执行 wait()方法，JVM会把该线程放入等待队列(waitting queue)中，使本线程进入到等待阻塞状态；

   (二). 同步阻塞：线程在获取 synchronized 同步锁失败(因为锁被其它线程所占用)，则JVM会把该线程放入锁池(lock pool)中，线程会进入同步阻塞状态；

   (三). 其他阻塞：通过调用线程的 sleep()或 join()或发出了 I/O 请求时，线程会进入到阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入就绪状态。
5. **死亡(dead)**：线程run()、main()方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期，死亡的线程不可再次复生。

Java规定的线程状态



关于线程生命周期的不同状态，在 Java 5 以后，线程状态被明确定义在其公共内部枚举类型 java.lang.Thread.State 中，分别是：

- **新建（NEW）**，表示线程被创建出来还没真正启动的状态，可以认为它是个 Java 内部状态。
- **就绪（RUNNABLE）**，表示该线程已经在 JVM 中执行，当然由于执行需要计算资源，它可能是正在运行，也可能还在等待系统分配给它 CPU 片段，在就绪队列里面排队。

  在其他一些分析中，会额外区分一种状态 RUNNING，但是从 Java API 的角度，并不能表示出来。
- **阻塞（BLOCKED）**，阻塞表示线程在等待 Monitor lock。比如，线程试图通过 synchronized 去获取某个锁，但是其他线程已经独占了，那么当前线程就会处于阻塞状态。
- **等待（WAITING）**，表示正在等待其他线程采取某些操作。一个常见的场景是类似生产者消费者模式，发现任务条件尚未满足，就让当前消费者线程等待（wait），另外的生产者线程去准备任务数据，然后通过类似 notify 等动作，通知消费线程可以继续工作了。Thread.join() 也会令线程进入等待状态。
- **计时等待（TIMED_WAIT）**，其进入条件和等待状态类似，但是调用的是存在超时条件的方法，比如 wait 或 join 等方法的指定超时时间
- **终止（TERMINATED）**，不管是意外退出还是正常执行结束，线程已经完成使命，终止运行，也有人把这个状态叫作死亡。

Java 中用到的线程调度算法是什么?

线程调度是指按照特定机制为多个线程分配 CPU 的使用权。Java 虚拟机的一项任务就是负责线程的调度。

有两种调度模型：分时调度模型和抢占式调度模型。Java虚拟机采用抢占式调度模型。

分时调度模型：让所有的线程轮流获得 cpu 的使用权，平均分配每个线程占用的 CPU 的时间片。

抢占式调度模型：根据线程优先级、线程饥饿情况等数据算出一个总的优先级，优先让可运行池中优先级高的线程占用CPU，如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用CPU。

请说出与线程同步以及线程调度相关的方法。

（1） **wait()**：使一个线程处于等待状态，并且释放所持有的对象的锁；

（2）**sleep()**：使一个正在运行的线程处于睡眠状态，是一个静态方法，sleep() 不释放锁，调用此方法要处理 InterruptedException 异常；

（3）**yield()**：使当前线程从运行状态变为就绪状态；

（4）**notify()**：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由 JVM 确定唤醒哪个线程，而且与优先级无关；

（5）**notifyAll()**：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

sleep() 和 wait() 有什么区别?

两者都可以暂停线程的执行，调用方法时都会抛出 InterruptedException 异常

- 类的不同：sleep() 是 Thread线程类的静态方法，wait() 是 Object类的方法。
- 是否释放锁：sleep() 不释放锁；wait() 释放锁。
- 用途不同：wait() 方法通常被用于线程间交互/通信，sleep() 通常被用于暂停线程执行。
- 用法不同：wait() 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 notify() 或者 notifyAll() 方法。或者可以使用wait(long timeout)超时后线程会自动苏醒。sleep() 方法执行完成后，线程会自动苏醒。

线程的 sleep()方法和 yield()方法有什么区别?

（1） sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；

（2） 线程执行 sleep()方法后转入阻塞（blocked）状态，而执行 yield()方法后转入就绪（ready）状态；

（3）sleep()方法声明抛出 InterruptedException，而 yield()方法没有声明任何异常；

（4）sleep()方法比 yield()方法（跟操作系统 CPU 调度相关）具有更好的可移植性，通常不建议使用 yield()方法来控制并发线程的执行。

为什么线程通信的方法 wait(), notify()和 notifyAll()被定义在 Object 类里?

Java中，任何对象都可以作为锁，并且 wait()，notify()等方法用于释放对象的锁或者唤醒线程，在 Java 的线程中并没有可供任何对象使用的锁，所以任意对象调用方法一定定义在Object类中。

wait(), notify()和 notifyAll()这些方法需要在同步代码块中调用

有的人会说，既然是线程放弃对象锁，那也可以把wait()定义在Thread类里面啊，新定义的线程继承于Thread类，也不需要重新定义wait()方法的实现。然而，这样做有一个非常大的问题，一个线程完全可以持有很多锁，你一个线程放弃锁的时候，到底要放弃哪个锁？当然了，这种设计并不是不能实现，只是管理起来更加复杂。

综上所述，wait()、notify()和notifyAll()方法要定义在Object类中。

为什么 wait(), notify()和 notifyAll()必须在同步方法或者同步块中被调用?

当一个线程需要调用对象的 wait()方法的时候，这个线程必须拥有该对象的锁，接着它就会释放这个对象锁并进入等待状态。

同样的，当一个线程需要调用对象的 notify()方法时，这个线程必须拥有该对象的锁，然后它会释放这个对象的锁，以便其他在等待的线程就可以得到这个对象锁。

由于所有的这些方法都需要线程持有对象的锁，这样就只能通过同步来实现，所以他们只能在同步方法或者同步块中被调用。

Java 中你怎样唤醒一个阻塞的线程?

首先，wait()、notify() 方法是针对对象的，调用任意对象的 wait()方法都将导致线程阻塞，阻塞的同时也将释放该对象的锁，相应地，调用任意对象的 notify()方法则将随机解除该对象阻塞的线程，但它需要重新获取该对象的锁，直到获取成功才能往下执行；

其次，wait、notify 方法必须在 synchronized 块或方法中调用，并且要保证同步块或方法的锁对象与调用 wait、notify 方法的锁对象是同一个，因此，在调用 wait 方法之前当前线程就已经成功获取对象的锁，执行 wait方法阻塞后，当前线程就将之前获取的对象锁释放。

Java 如何实现多线程之间的通讯和协作?

Java中线程通信协作的最常见的两种方式：

- syncrhoized加锁的线程的**Object类**的wait()/notify()/notifyAll()/sleep()/yield()/join()
- ReentrantLock类加锁的线程的**Condition类**的await()/signal()/signalAll()
- 并发工具类：Semaphore信号量，CountDownLatch 倒计时器，CyclicBarrier循环栅栏

线程间直接的数据交换：

- 通过管道进行线程间通信：1）字节流；2）字符流

比如说最经典的生产者-消费者模型：当队列满时，生产者需要等待队列有空间才能继续往里面放入商品，而**在等待的期间内，生产者必须释放对临界资源（即队列）的占用权。**因为生产者如果不释放对临界资源的占用权，那么**消费者就无法消费队列中的商品**，就不会让队列有空间，那么生产者就会一直无限等待下去。因此，一般情况下，当队列满时，会让生产者交出对临界资源的占用权，并进入挂起状态。然后等待消费者消费了商品，然后消费者通知生产者队列有空间了。同样地，当队列空时，消费者也必须等待，等待生产者通知它队列中有商品了。这种互相通信的过程就是线程间的协作。

什么是线程同步和线程互斥，有哪几种实现方式?

当一个线程对共享的数据进行操作时，应使之成为一个"原子操作"，即在没有完成相关操作之前，不允许其他线程打断它，否则，就会破坏数据的完整性，必然会得到错误的处理结果，这就是线程的同步。

线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步。

线程间的同步方法大体可分为两类：用户模式和内核模式。顾名思义，内核模式就是指利用系统内核对象的单一性来进行同步，使用时需要切换内核态与用户态，而用户模式就是不需要切换到内核态，只在用户态完成操作。

用户模式下的方法有：原子操作（例如一个单一的全局变量），临界区。

- 原子操作
- 临界区（Critical Section）：适合一个进程内的多线程访问公共区域或代码段时使用

内核模式下的方法有：事件，信号量，互斥量。

- 事件（Event）：通过线程间触发事件实现同步互斥
- 互斥量 (Mutex)：适合不同进程内多线程访问公共区域或代码段时使用，与临界区相似
- 信号量（Semaphore）：与临界区和互斥量不同，可以实现多个线程同时访问公共区域数据，原理与操作系统中PV操作类似，先设置一个访问公共区域的线程最大连接数，每有一个线程访问共享区资源数就减一，直到资源数小于等于零

实现线程同步的方法

- **同步代码方法**：sychronized 关键字修饰的方法
- **同步代码块**：sychronized 关键字修饰的代码块

- **使用特殊变量域volatile实现线程同步**：volatile关键字为域变量的访问提供了一种免锁机制
- **使用重入锁实现线程同步**：reentrantlock类是可重入、互斥、实现了lock接口的锁与sychronized方法具有相似的行为和语义

一个线程运行时发生异常会怎样?

如果异常没有被捕获该线程将会停止执行。`Thread.UncaughtExceptionHandler`是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。当一个未捕获异常将造成线程中断的时候，JVM 会使用`Thread.getUncaughtExceptionHandler()`来查询线程的`UncaughtExceptionHandler`并将线程和异常作为参数传递给 handler 的`uncaughtException()`方法进行处理。

并发理论

Java中垃圾回收有什么目的?

垃圾回收的目的是识别应用中不再使用的对象，同时释放和重用资源。

如果对象的引用被置为null，垃圾收集器是否会立即释放对象占用的内存?

不会，在下一个垃圾回收周期中，这个对象将是可回收的。

也就是说并不会立即被垃圾收集器立刻回收，而是在下一次垃圾回收时才会释放其占用的内存。

为什么代码会重排序?

在执行程序时，为了提高性能，处理器和编译器常常会对指令进行重排序，但是不能随意重排序，不是你想怎么排序就怎么排序，它需要满足以下两个条件：

- 在单线程环境下不能改变程序运行的结果；
- 存在数据依赖关系的不允许重排序

需要注意的是：重排序不会影响单线程环境的执行结果，但是会破坏多线程的执行语义。

as-if-serial规则和happens-before规则的区别

- as-if-serial语义保证单线程内程序的执行结果不被改变，happens-before语义保证正确同步的多线程程序的执行结果不被改变。
- as-if-serial语义给编写单线程程序的程序员创造了一个幻境：单线程程序是按程序的顺序来执行的。happens-before语义给编写正确同步的多线程程序的程序员创造了一个幻境：正确同步的多线程程序是按happens-before指定的顺序来执行的。
- as-if-serial语义和happens-before这么做的目的，都是为了在不改变程序执行结果的前提下，尽可能地提高程序执行的并行度。

并发关键字

@$synchronized 的作用?

在 Java 中，synchronized 关键字是用来控制线程同步的，就是在多线程的环境下，synchronized 修饰的代码段不被多个线程同时执行。synchronized 可以修饰静态方法，实例方法和代码块

synchronized优化**简答**：在 Java 早期版本中，synchronized属于重量级锁，效率低下；在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而减低了锁带来的性能消耗。

synchronized优化**详答**：在 Java 早期版本中，synchronized属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这

也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后 Java 官方从 JVM 层面对 synchronized 进行较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

@$说说自己是怎么使用 synchronized 关键字，在项目中用到了吗

synchronized关键字最主要的三种使用方式

- **修饰静态方法**：对当前类对象加锁，进入同步代码前要获得当前类对象的锁
- **修饰实例方法**：对当前实例对象加锁，进入同步代码前要获得当前实例对象的锁
- **修饰代码块**：如果synchronized括号里面的是对象，锁的就是实例对象；如果括号里面的是class类，锁的是类

下面我以一个常见的面试题为例讲解一下 synchronized 关键字的具体使用。

面试中面试官经常会说：“单例模式了解吗？来给我手写一下！给我解释一下双重检验锁方式实现单例模式的原理呗！”

双重校验锁实现对象单例（线程安全）

```
public class Singleton {

    private volatile static Singleton uniqueInstance;

    private Singleton() {
    }

    public static Singleton getUniqueInstance() {
       //先判断对象是否已经实例过，没有实例化过才进入加锁代码
        if (uniqueInstance == null) {
            //类对象加锁
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

另外，需要注意 uniqueInstance 采用 volatile 关键字修饰也是很有必要的， uniqueInstance = new Singleton(); 这段代码其实是分为三步执行：

1. 为 uniqueInstance 分配内存空间
2. 初始化 uniqueInstance
3. 将 uniqueInstance 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1->3->2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被初始化。

使用 volatile 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行。

@$说一下 synchronized 底层实现原理?

synchronized是Java中的一个关键字，在使用的过程中并没有看到显示的加锁和解锁过程。因此有必要通过javap命令，查看相应的字节码文件。

synchronized 同步语句块的情况

```
public class SynchronizedDemo {
    public void method() {
        synchronized (this) {
            System.out.println("synchronized 代码块");
        }
    }
}
```

通过JDK 反汇编指令 javap -c -v SynchronizedDemo

```
public void method();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=1
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter
       4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       7: ldc           #3                  // String Method 1 start
       9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      12: aload_1
      13: monitorexit
      14: goto          22
      17: astore_2
      18: aload_1
      19: monitorexit
      20: aload_2
      21: athrow
      22: return
    Exception table:
       from    to  target type
           4    14    17   any
          17    20    17   any
    LineNumberTable:
      line 5: 0
      line 6: 4
      line 7: 12
      line 8: 22
    StackMapTable: number_of_entries = 2
      frame_type = 255 /* full_frame */
        offset_delta = 17
        locals = [ class test/SynchronizedDemo, class java/lang/Object ]
        stack = [ class java/lang/Throwable ]
      frame_type = 250 /* chop */
        offset_delta = 4
}
SourceFile: "SynchronizedDemo.java"
```

可以看出在执行同步代码块之前之后都有一个monitor字样，其中前面的是monitorenter，后面的是离开monitorexit，不难想象一个线程在执行同步代码块时，首先要获取锁，而获取锁的过程就是monitorenter ，在执行完代码块之后，要释放锁，释放锁就是执行monitorexit指令。

为什么会有两个monitorexit呢?

这个主要是防止在同步代码块中线程因异常退出，而锁没有得到释放，这必然会造成死锁（等待的线程永远获取不到锁）。因此最后一个monitorexit是保证在异常情况下，锁也可以得到释放，避免死锁。

synchronized可重入的原理

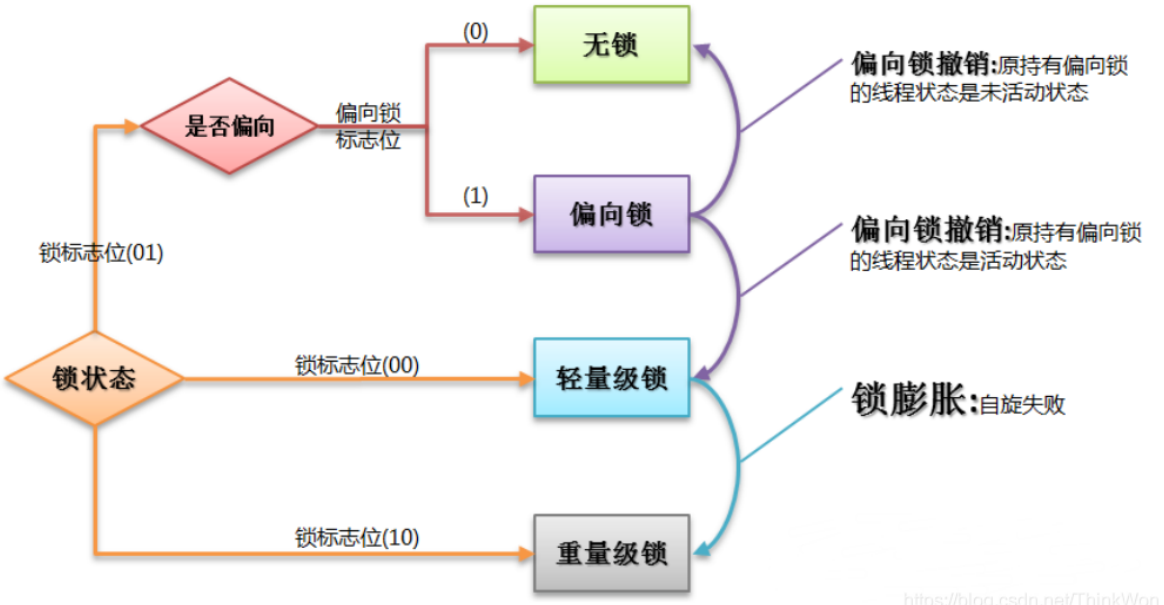
重入锁是指一个线程获取到该锁之后，该线程可以继续获得该锁。底层原理维护一个计数器，当线程获取该锁时，计数器加一，再次获得该锁时继续加一，释放锁时，计数器减一，当计数器值为0时，表明该锁未被任何线程所持有，其它线程可以竞争获取锁。

什么是自旋

很多 synchronized 里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然 synchronized 里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在 synchronized 的边界做循环，这就是自旋。如果做了多次循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。

@$多线程中 synchronized 锁升级的原理是什么?

锁升级简单图示过程



synchronized 锁升级原理：在锁对象的对象头的 Mark Word 部分里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的锁对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级。

锁的升级的目的：锁升级是为了降低了锁带来的性能消耗。在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而降低了锁带来的性能消耗。

锁的升级详细概述

Mark Word的状态变化

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC标记	空				11
偏向锁	线程ID	EPoch	对象分代年龄	1	01
无锁	对象HashCode		对象分代年龄	0	01

锁的级别从低到高：

无锁 -> 偏向锁 -> 轻量级锁 -> 重量级锁

锁分级别原因：

没有优化以前，sychronized是重量级锁（悲观锁），使用 wait 和 notify、notifyAll 来切换线程状态非常消耗系统资源；线程的挂起和唤醒间隔很短暂，这样很浪费资源，影响性能。所以 JVM 对 sychronized 关键字进行了优化，把锁分为 无锁、偏向锁、轻量级锁、重量级锁 状态。

锁状态对比

定义

- 无锁

  没有对资源进行锁定，所有的线程都能访问并修改同一个资源，但同时只有一个线程能修改成功，其他修改失败的线程会不断重试直到修改成功。

- 偏向锁

  对象的代码一直被同一线程执行，不存在多个线程竞争，该线程在后续的执行中自动获取锁，降低获取锁带来的性能开销。偏向锁，指的就是偏向第一个加锁线程，该线程是不会主动释放偏向锁的，只有当其他线程尝试竞争偏向锁才会被释放。偏向锁的撤销，需要在某个时间点上没有字节码正在执行时，先暂停拥有偏向锁的线程，然后判断锁对象是否处于被锁定状态。如果线程不处于活动状态，则将对象头设置成无锁状态，并撤销偏向锁；如果线程处于活动状态，升级为轻量级锁的状态。

- 轻量级锁

  轻量级锁是指当锁是偏向锁的时候，被第二个线程 B 所访问，此时偏向锁就会升级为轻量级锁，线程 B 会通过自旋的形式尝试获取锁，线程不会阻塞，从而提高性能。当前只有一个等待线程，则该线程将通过自旋进行等待。但是当自旋超过一定的次数时，轻量级锁便会升级为重量级锁；当一个线程已持有锁，另一个线程在自旋，而此时又有第三个线程来访时，轻量级锁也会升级为重量级锁。

- 重量级锁

  指当有一个线程获取锁之后，其余所有等待获取该锁的线程都会处于阻塞状态。重量级锁通过对象内部的监视器（monitor）实现，而其中 monitor 的本质是依赖于底层操作系统的 Mutex Lock 实现，操作系统实现线程之间的切换需要从用户态切换到内核态，切换成本非常高。

适用场景

- 无锁

  无需保证线程安全的场景

- 偏向锁

  偏向锁的撤销，需要在某个时间点上没有字节码正在执行时，先暂停拥有偏向锁的线程，然后判断锁对象是否处于被锁定状态。如果线程不处于活动状态，则将对象头设置成无锁状态，并撤销偏向锁；只有一个线程进入同步块

- 轻量级锁

  当前只有一个等待线程，则该线程将通过自旋进行等待。但是当自旋超过一定的次数时，轻量级锁便会升级为重量级锁；当一个线程已持有锁，另一个线程在自旋，而此时又有第三个线程来访时，轻量级锁也会升级为重量级锁。虽然很多线程，但是没有冲突：多条线程进入同步块，但是线程进入时间错开因而并未争抢锁

- 重量级锁

  重量级锁通过对象内部的监视器（monitor）实现，而其中 monitor 的本质是依赖于底层操作系统的 Mutex Lock 实现，操作系统实现线程之间的切换需要从用户态切换到内核态，切换成本非常高。发生了锁争抢的情况：多条线程进入同步块并争用锁

本质

- 无锁

  null
- 偏向锁

  如果线程处于活动状态，升级为轻量级锁的状态。取消同步操作
- 轻量级锁

  CAS操作代替互斥同步
- 重量级锁

  互斥同步

优点

- 无锁

  不阻塞，执行效率高
- 偏向锁

  不阻塞，执行效率高（只有第一次获取偏向锁时需要CAS操作，后面只是比对ThreadId）
- 轻量级锁

  不会阻塞
- 重量级锁

  不会空耗CPU

缺点

- 无锁

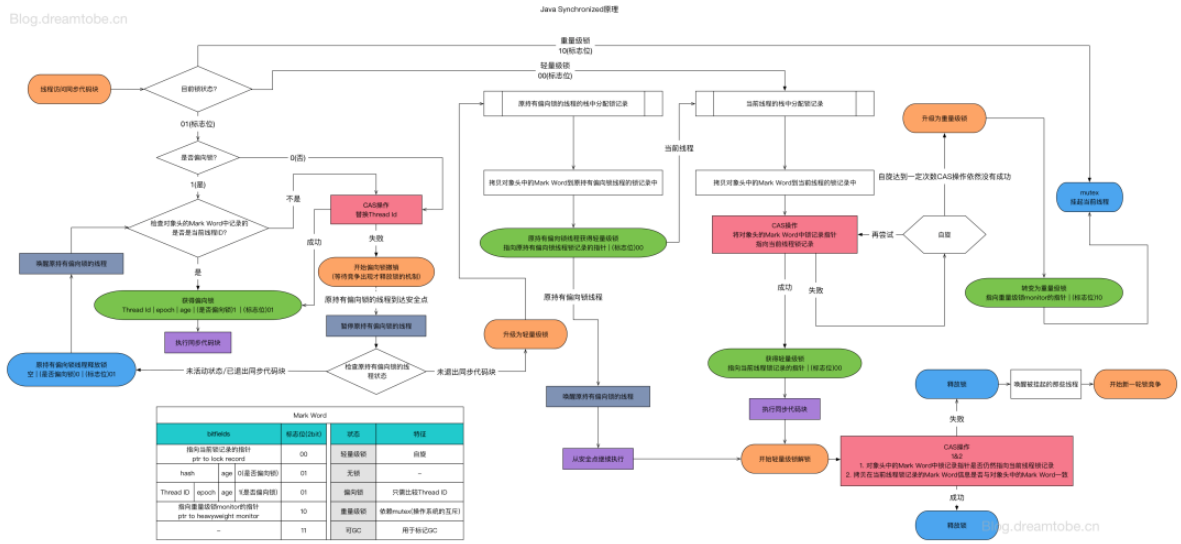
  线程不安全
- 偏向锁

  适用场景太局限。若竞争产生，会有额外的偏向锁撤销的消耗
- 轻量级锁

  长时间获取不到锁时会空耗CPU
- 重量级锁

  阻塞，上下文切换，重量级操作，消耗操作系统资源

锁升级流程图



@$synchronized 和 Lock 有什么区别?

- 首先synchronized是Java关键字，Lock是 Java 接口；
- synchronized 可以给方法、代码块加锁；而 lock 只能给代码块加锁。
- synchronized 不需要手动获取锁和释放锁，使用简单，发生异常会自动释放锁，不会造成死锁；而 lock 需要自己加锁和释放锁，如果使用不当没有 unLock()去释放锁可能造成死锁。
- 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

synchronized 和 ReentrantLock 区别是什么?

相同点：两者都是可重入锁。

主要区别如下：

- 本质：synchronized 是关键字，ReetrantLock是类，这是二者的本质区别。既然 ReentrantLock 是类，那么它就提供了比 synchronized 更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的变量
- 加锁和释放锁：ReentrantLock 必须手动获取与释放锁，而 synchronized 不需要手动开启和释放锁；
- 作用域：ReentrantLock 只能给代码块加锁，而 synchronized 可以给方法、代码块加锁。
- 性能：早期版本 synchronized 在很多场景下性能较差，在后续版本进行了较多改进，在低竞争场景中表现可能优于 ReentrantLock。
- 底层实现：二者的锁机制其实也是不一样的。ReentrantLock 底层调用的是 Unsafe 的 park 方法加锁，synchronized 操作的是对象头中 mark word

@$volatile 关键字的作用

Java 提供了 volatile 关键字来保证可见性和禁止指令重排（有序性）。volatile 提供 happens-before 的保证，同时确保一个线程对共享变量的修改能对其他线程是可见的。当一个共享变量被 volatile 修饰时，它会保证修改的值会立即被更新到内存，当有其他线程需要读取时，它会去内存中读取新值。

从实践角度而言，volatile 的一个重要作用就是和 CAS 结合，保证了原子性，详细的可以参见 java.util.concurrent.atomic 包下的类，比如 AtomicInteger。

volatile 常用于多线程环境下的单次操作(单次读或者单次写)。

volatile 能保证原子性吗？volatile 能使得一个非原子操作变成原子操作吗?

关键字volatile的主要作用是使变量在多个线程间可见，但无法保证原子性，所以对于多个线程访问共享变量需要加锁进行同步。

虽然volatile只能保证可见性不能保证原子性，但用volatile修饰long和double可以保证其操作原子性。

从Oracle Java Spec里面可以看到：

对于64位的long和double，如果没有被volatile修饰，那么对其操作可以不是原子的。

在操作的时候，可以分成两步，每次对32位操作。如果使用volatile修饰long和double，那么其读写都是原子操作

@$synchronized 和 volatile 的区别是什么?

synchronized 表示只有一个线程可以获取对象的锁，执行代码，阻塞其他线程。

volatile 表示变量在 CPU 的寄存器中是不确定的，必须从主存中读取。保证多线程环境下变量的可见性和禁止指令重排序。

区别

- 使用范围：volatile 是变量修饰符；synchronized 可以修饰方法和代码块。

- 并发编程三要素：volatile 仅能实现变量的修改可见性，不能保证原子性；而 synchronized 则可以保证变量的修改可见性和原子性。
- 阻塞：volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。
- 编译器优化：volatile 标记的变量不会被编译器优化；synchronized标记的变量可以被编译器优化。
- 性能与使用场景：**volatile关键字**是线程同步的**轻量级实现**，所以**volatile性能肯定比synchronized关键字要好。**但是**volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块。**synchronized关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗，引入的偏向锁和轻量级锁以及其它各种优化，执行效率有了显著提升，**实际开发中使用 synchronized 关键字的场景还是更多一些。**

Lock体系

Lock 接口(Lock interface)是什么？对比 synchronized 它有什么优势?

Lock 接口比同步方法和同步块提供了更具扩展性的锁操作。Lock 接口缺少了（通过synchronized块或者方法所提供的）隐式获取释放锁的便捷性，但是却拥有了锁获取与释放的可操作性、可中断的获取锁以及超时获取锁等多种synchronized关键字所不具备的同步特性。

整体上来说 Lock 是 synchronized 的扩展版，Lock 提供了无条件的、可轮询的(tryLock 方法)、定时的(tryLock 带参方法)、可中断的(lockInterruptibly)、可多条件队列的(newCondition 方法)锁操作。另外 Lock 的实现类基本都支持非公平锁(默认)和公平锁，synchronized 只支持非公平锁，当然，在大部分情况下，非公平锁是高效的选择。

@$乐观锁和悲观锁的理解及如何实现，有哪些实现方式?

悲观锁：假定会发生并发冲突，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如 jdk1.6 以前的同步原语 synchronized 关键字的实现也是悲观锁。

乐观锁：假设不会发生并发冲突，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 write_condition 机制是乐观锁。在 Java中 java.util.concurrent.atomic 包下面的原子操作类就是使用了乐观锁的一种实现方式 CAS。

乐观锁的实现方式：

1、**版本号机制**：一般是在数据表中加上一个版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读到的version值与当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

2、**CAS算法**：java 中的 Compare and Swap 即 CAS ，当多个线程尝试使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值（B）。如果内存地址里面的值 V 和 预期原值 A 的值是一样的，那么就将内存里面的值 V 更新成新值 B。CAS是通过无限循环来获取数据的，如果在第一轮循环中，a 线程获取地址里面的值被 b 线程修改了，那么 a 线程需要自旋，到下次循环才有可能机会执行。

什么是 CAS

CAS 是 compare and swap 的缩写，即我们所说的比较交换。CAS 是基于冲突检测的乐观锁（非阻塞）

cas 是一种基于锁的操作，而且是乐观锁。

java.util.concurrent.atomic 包下的原子操作类大多是使用 CAS 操作来实现的(AtomicInteger，AtomicBoolean，AtomicLong)。

CAS 的会产生什么问题?

1、ABA 问题：

比如说一个线程 1 从内存位置 V 中取出 A，这时候另一个线程 2 也从内存中取出 A，并且线程 2 进行了一些操作变成了 B，然后线程 2 又将 V 位置的数据变成 A，这时候线程 1 进行 CAS 操作发现内存中仍然是 A，然后线程 1 操作成功。尽管线程 1 的 CAS 操作成功，但可能存在潜藏的问题。从 Java1.5 开始 JDK 的 atomic包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。

2、循环时间长开销大：

对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized。

3、只能保证一个共享变量的原子操作：

当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。

AQS 介绍

AQS的全称为（AbstractQueuedSynchronizer），这个类在java.util.concurrent.locks包下面。

AQS是一个用来构建锁和同步器的框架，使用AQS能简单且高效地构造出应用广泛的同步器，比如我们提到的ReentrantLock，Semaphore，其他的诸如ReentrantReadWriteLock，SynchronousQueue，FutureTask等等皆是基于AQS的。当然，我们自己也能利用AQS非常轻松地构造出符合我们自己需要的同步器。

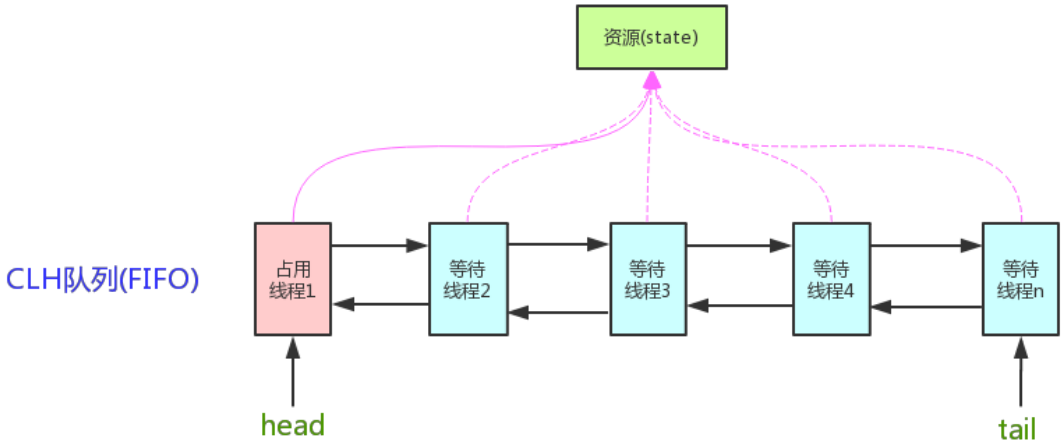
@$AQS 原理分析

AQS 原理概览

AQS核心思想是，如果请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中

> CLH(Craig,Landin,and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS是将每条请求共享资源的线程封装成一个CLH锁队列的一个结点（Node）来实现锁的分配。

看个AQS(AbstractQueuedSynchronizer)原理图：



AQS使用一个int成员变量来表示同步状态，通过内置的FIFO队列来完成获取资源线程的排队工作。AQS使用CAS对该同步状态进行原子操作实现对其值的修改。

```
private volatile int state;//共享变量，使用volatile修饰保证线程可见性
```

状态信息通过protected类型的getState，setState，compareAndSetState进行操作

```
//返回同步状态的当前值-
protected final int getState() {
    return state;
}
// 设置同步状态的值
protected final void setState(int newState) {
    state = newState;
}
//原子地（CAS操作）将同步状态值设置为给定值update如果当前同步状态的值等于expect（期望值）
protected final boolean compareAndSetState(int expect, int update) {
    return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
}
```

AQS 对资源的共享方式

AQS定义两种资源共享方式

- Exclusive（独占）：只有一个线程能执行，如ReentrantLock。又可分为公平锁和非公平锁：
  - 公平锁：按照线程在队列中的排队顺序，先到者先拿到锁
  - 非公平锁：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的
- Share（共享）：多个线程可同时执行，如Semaphore/CountDownLatch。

不同的自定义同步器竞争共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。

AQS底层使用了模板方法模式

AQS同步器的设计是基于模板方法模式的，如果需要自定义同步器一般的方式是这样（模板方法模式很经典的一个设计模式）：

1. 使用者继承AbstractQueuedSynchronizer并重写指定的方法。（这些重写方法很简单，无非是对于共享资源state的获取和释放）
2. 将AQS组合在自定义同步器的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。

这和我们以往通过实现接口的方式有很大区别，这是模板方法模式很经典的一个运用。

AQS使用了模板方法模式，自定义同步器时需要重写下面几个AQS提供的模板方法

```
isHeldExclusively()//该线程是否正在独占资源。只有用到condition才需要去实现它。
tryAcquire(int)//独占方式。尝试获取资源，成功则返回true，失败则返回false。
tryRelease(int)//独占方式。尝试释放资源，成功则返回true，失败则返回false。
tryAcquireShared(int)//共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源，正数表示成功，且有剩余资源。
tryReleaseShared(int)//共享方式。尝试释放资源，成功则返回true，失败则返回false。
```

默认情况下，每个方法都抛出 `UnsupportedOperationException`。这些方法的实现必须是内部线程安全的，并且通常应该简短而不是阻塞。AQS类中的其他方法都是final ，所以无法被其他类使用，只有这几个方法可以被其他类使用。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现 `tryAcquire-tryRelease`、`tryAcquireShared-tryReleaseShared` 中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，如 `ReentrantReadWriteLock`。

什么是可重入锁（ReentrantLock）？

ReentrantLock重入锁，是实现Lock接口的一个类，也是在实际编程中使用频率很高的一个锁，支持重入性，表示能够对共享资源重复加锁，即当前线程获取该锁再次获取时不会被阻塞。

在java关键字synchronized隐式支持重入性，synchronized通过获取自增，释放自减的方式实现重入。与此同时，ReentrantLock还支持公平锁和非公平锁两种方式。那么，要想完完全全的弄懂ReentrantLock的话，主要也就是ReentrantLock同步语义的学习：1. 重入性的实现原理；2. 公平锁和非公平锁。

重入性的实现原理

以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证state是能回到零态的。

公平锁和非公平锁

- 公平锁：按照线程在队列中的排队顺序，先到者先拿到锁
- 非公平锁：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的

非公平锁和公平锁的两处不同：

1. 非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。
2. 非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了（state == 0），非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

公平锁和非公平锁就这两点区别，如果非公平锁这两次 CAS 都不成功，那么后面非公平锁和公平锁是一样的，都要进入到阻塞队列等待唤醒。

公平锁每次获取到锁为同步队列中的第一个节点，保证请求资源时间上的绝对顺序，而非公平锁有可能刚释放锁的线程下次继续获取该锁，则有可能导致其他线程永远无法获取到锁，造成“饥饿”现象。

公平锁为了保证时间上的绝对顺序，需要频繁的上下文切换，而非公平锁会降低一定的上下文切换，减少性能开销。因此，ReentrantLock默认选择的是非公平锁，则是为了减少一部分上下文切换，保证了系统更大的吞吐量。

ReadWriteLock 是什么

首先明确一下，不是说 ReentrantLock 不好，只是 ReentrantLock 某些时候有局限。如果使用 ReentrantLock，可能本身是为了防止线程 A 在写数据、线程 B 在读数据造成的数据不一致，但这样，如果线程 C 在读数据、线程 D 也在读数据，读数据是不会改变数据的，没有必要加锁，但是还是加锁了，降低了程序的性能。因为这个，才诞生了读写锁 ReadWriteLock。

ReadWriteLock 是一个读写锁接口，读写锁是用来提升并发程序性能的锁分离技术，ReentrantReadWriteLock 是 ReadWriteLock 接口的一个具体实现，实现了读写的分离，读锁是共享的，写锁是独占的，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能。

而读写锁有以下三个重要的特性：

- 公平选择性：支持非公平（默认）和公平的锁获取方式，吞吐量还是非公平优于公平。
- 可重入：读锁和写锁都支持线程可重入。
- 锁降级：遵循先获取写入锁，然后获取读取锁，最后释放写入锁。写锁能够降级成为读锁，但是，从读取锁升级到写入锁是不可能的。

并发容器

什么是ConcurrentHashMap?

ConcurrentHashMap是Java中的一个**线程安全且高效的HashMap实现。**平时涉及高并发如果要用map结构，那第一时间想到的就是它。相对于hashmap来说，ConcurrentHashMap就是线程安全的，其中利用了锁分段的思想提高了并发度。

那么它到底是如何实现线程安全的?

在JDK1.7中，ConcurrentHashMap采用Segment + HashEntry的数据结构，结构如下：

一个 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和 HashMap 类似，是一种数组和链表结构，segment继承了ReentrantLock，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素。当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment 的锁。

在**JDK1.8中，放弃了Segment臃肿的设计，取而代之的是采用Node + CAS + Synchronized来保证并发安全**，synchronized只锁定当前链表或红黑二叉树的首节点，这样只要hash不冲突，就不会产生并发，效率又提升N倍。

CopyOnWriteArrayList 是什么，可以用于什么应用场景？有哪些优缺点?

CopyOnWriteArrayList 是一个并发容器。有很多人称它是线程安全的，我认为这句话不严谨，缺少一个前提条件，那就是非复合场景下操作它是线程安全的。

CopyOnWriteArrayList(免锁容器)的好处之一是当多个迭代器同时遍历和修改这个列表时，不会抛出 ConcurrentModificationException。在CopyOnWriteArrayList 中，写入将导致创建整个底层数组的副本，而源数组将保留在原地，使得复制的数组在被修改时，读取操作可以安全地执行。

CopyOnWriteArrayList 的使用场景

通过源码分析，我们看出它的优缺点比较明显，所以使用场景也就比较明显。就是**合适读多写少的场景。**

CopyOnWriteArrayList 的缺点

1. 由于写操作的时候，需要拷贝数组，会消耗内存，如果原数组的内容比较多的情况下，可能导致 **young gc** 或者 **full gc。**
2. 不能用于实时读的场景，像拷贝数组、新增元素都需要时间，所以调用一个 set 操作后，读取到数据可能还是旧的，虽然CopyOnWriteArrayList 能做到最终一致性，但是还是没法满足实时性要求。
3. 由于实际使用中可能没法保证 CopyOnWriteArrayList 到底要放置多少数据，万一数据稍微有点多，每次 add/set 都要重新复制数组，这个代价实在太高昂了。在高性能的互联网应用中，这种操作分分钟引起故障。

CopyOnWriteArrayList 的设计思想

1. 读写分离，读和写分开
2. 最终一致性
3. 使用另外开辟空间的思路，来解决并发冲突

@$ThreadLocal 是什么？有哪些使用场景？什么是线程局部变量?

ThreadLocal 是一个本地线程局部变量工具类，在每个线程中都创建了一个 ThreadLocalMap 对象，简单说 ThreadLocal 就是一种以空间换时间的做法，每个线程可以访问自己内部 ThreadLocalMap 对象内的 value。通过这种方式，避免资源在多线程间共享。

原理：线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java提供ThreadLocal类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

经典的使用场景是为每个线程分配一个 JDBC 连接 Connection。这样就可以保证每个线程的都在各自的 Connection 上进行数据库的操作，不会出现 A 线程关了 B 线程正在使用的 Connection；还有 Session 管理 等问题。

ThreadLocal 使用例子：

```
public class ThreadLocalTest {

    //线程本地存储变量
    private static final ThreadLocal<Integer> THREAD_LOCAL_NUM
        = new ThreadLocal<Integer>() {
        @Override
        protected Integer initialValue() {
            return 0;
        }
    };

    public static void main(String[] args) {
        for (int i = 0; i <3; i++) {//启动三个线程
            Thread t = new Thread() {
                @Override
                public void run() {
                    add10ByThreadLocal();
                }
            };
            t.start();
        }
```

```
    }

    /**
     * 线程本地存储变量加 5
     */
    private static void add10ByThreadLocal() {
        for (int i = 0; i <5; i++) {
            Integer n = THREAD_LOCAL_NUM.get();
            n += 1;
            THREAD_LOCAL_NUM.set(n);
            System.out.println(Thread.currentThread().getName() + " :
ThreadLocal num=" + n);
        }
    }

}
```

打印结果：启动了 3 个线程，每个线程最后都打印到 "ThreadLocal num=5"，而不是 num 一直在累加直到值等于 15

```
Thread-0 : ThreadLocal num=1
Thread-1 : ThreadLocal num=1
Thread-0 : ThreadLocal num=2
Thread-0 : ThreadLocal num=3
Thread-1 : ThreadLocal num=2
Thread-2 : ThreadLocal num=1
Thread-0 : ThreadLocal num=4
Thread-2 : ThreadLocal num=2
Thread-1 : ThreadLocal num=3
Thread-1 : ThreadLocal num=4
Thread-2 : ThreadLocal num=3
Thread-0 : ThreadLocal num=5
Thread-2 : ThreadLocal num=4
Thread-2 : ThreadLocal num=5
Thread-1 : ThreadLocal num=5
```

@$ThreadLocal造成内存泄漏的原因?

`ThreadLocalMap` 中使用的 key 为 `ThreadLocal` 的弱引用，而 value 是强引用。所以，如果 `ThreadLocal` 没有被外部强引用的情况下，在垃圾回收的时候，key 会被清理掉，而 value 不会被清理掉。这样一来，`ThreadLocalMap` 中就会出现key为null的Entry。假如我们不做任何措施的话，value 永远无法被GC 回收，这个时候就可能会产生内存泄露。

@$ThreadLocal内存泄漏解决方案?

每次使用完ThreadLocal，都调用它的remove()方法，清除数据。

在使用线程池的情况下，没有及时清理ThreadLocal，不仅是内存泄漏的问题，更严重的是可能导致业务逻辑出现问题。所以，使用ThreadLocal就跟加锁完要解锁一样，用完就清理。

什么是阻塞队列？阻塞队列的实现原理是什么？如何使用阻塞队列来实现生产者-消费者模型？

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。

这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。

阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素，而消费者从容器里拿元素的容器。

JDK7 提供了 7 个阻塞队列。分别是：

- ArrayBlockingQueue ： 一个由数组结构组成的有界阻塞队列。
- LinkedBlockingQueue ： 一个由链表结构组成的有界阻塞队列。
- PriorityBlockingQueue ： 一个支持优先级排序的无界阻塞队列。
- DelayQueue：一个使用优先级队列实现的无界阻塞队列。
- SynchronousQueue：一个不存储元素的阻塞队列。
- LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
- LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

Java 5 之前实现同步存取时，可以使用普通的一个集合，然后在使用线程的协作和线程同步可以实现生产者，消费者模式，主要的技术就是用好wait,notify,notifyAll,sychronized 这些方法或关键字。而在 java 5 之后，可以使用阻塞队列来实现，此方式大大简少了代码量，使得多线程编程更加容易，安全方面也有保障。

BlockingQueue 接口是 Queue 的子接口，它的主要用途并不是作为容器，而是作为线程同步的的工具，因此他具有一个很明显的特性，当生产者线程试图向 BlockingQueue 放入元素时，如果队列已满，则线程被阻塞，当消费者线程试图从中取出一个元素时，如果队列为空，则该线程会被阻塞，正是因为它所具有这个特性，所以在程序中多个线程交替向 BlockingQueue 中放入元素，取出元素，它可以很好的控制线程之间的通信。

阻塞队列还有一个经典使用场景就是 socket 客户端数据的读取和解析，读取数据的线程不断将数据放入队列，然后解析线程不断从队列取数据解析。

线程池

什么是线程池?

> 池化技术相比大家已经屡见不鲜了，线程池、数据库连接池、Http 连接池等等都是对这个思想的应用。池化技术的思想主要是为了减少每次获取资源的消耗，提高资源的利用率。

在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。在 Java 中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这就是"池化资源"技术产生的原因。

线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取，线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。Java 5+中的 Executor 接口定义一个执行线程的工具。它的子类型即线程池接口是 ExecutorService。要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，因此在工具类 Executors 中提供了一些静态工厂方法，生成一些常用的线程池，如下所示：

（1）newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

（2）newFixedThreadPool：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，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。如果希望在服务器上使用线程池，建议使用 newFixedThreadPool 方法来创建线程池，这样能获得更好的性能。

（3） newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60 秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说 JVM）能够创建的最大线程大小。

（4）newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

有哪几种创建线程池的方式?

使用 Executors 工具类创建线程池(不推荐)

Executors提供了一系列工厂方法用于创先线程池，返回的线程池都实现了ExecutorService接口。

主要有newSingleThreadExecutor，newFixedThreadPool，newCachedThreadPool，newScheduledThreadPool

```
public class MyRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " run()方法执行中...");
    }

}
```

```
public class SingleThreadExecutorTest {

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        MyRunnable runnableTest = new MyRunnable();
        for (int i = 0; i < 5; i++) {
            executorService.execute(runnableTest);
        }

        System.out.println("线程任务开始执行");
        executorService.shutdown();
    }

}
```

执行结果

```
线程任务开始执行
pool-1-thread-1 is running...
pool-1-thread-1 is running...
pool-1-thread-1 is running...
pool-1-thread-1 is running...
pool-1-thread-1 is running...
```

使用ThreadPoolExecutor构造函数创建线程池

《阿里巴巴Java开发手册》中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程序的同学更加明确线程池的运行规则，规避资源耗尽的风险

```
public class Task implements Runnable {

    private int taskNum;

    public Task(int taskNum) {
        this.taskNum = taskNum;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " 正在执行task " + taskNum);
        try {
            Thread.sleep(4000L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " task " + taskNum + "执行完毕");
    }

}
```

```
public class ThreadPoolExecutorTest {

    public static void main(String[] args) {
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(2, 5, 200, TimeUnit.MILLISECONDS, new ArrayBlockingQueue<Runnable>(5));

        for (int i = 0; i < 5; i++) {
            Task task = new Task(i);
            threadPoolExecutor.execute(task);

            System.out.println(Thread.currentThread().getName() + " 线程池中线程数目：" + threadPoolExecutor.getPoolSize() + "，队列中等待执行的任务数目：" +
                    threadPoolExecutor.getQueue().size() + "，已执行完的任务数目：" + threadPoolExecutor.getCompletedTaskCount());
        }

        threadPoolExecutor.shutdown();
    }

}
```

执行结果

```
pool-1-thread-1 正在执行task 0
main 线程池中线程数目：1，队列中等待执行的任务数目：0，已执行完的任务数目：0
main 线程池中线程数目：2，队列中等待执行的任务数目：0，已执行完的任务数目：0
main 线程池中线程数目：2，队列中等待执行的任务数目：1，已执行完的任务数目：0
```

```
main 线程池中线程数目：2，队列中等待执行的任务数目：2，已执行完的任务数目：0
pool-1-thread-2 正在执行task 1
main 线程池中线程数目：2，队列中等待执行的任务数目：3，已执行完的任务数目：0
pool-1-thread-1 task 0执行完毕
pool-1-thread-2 task 1执行完毕
pool-1-thread-1 正在执行task 2
pool-1-thread-2 正在执行task 3
pool-1-thread-2 task 3执行完毕
pool-1-thread-2 正在执行task 4
pool-1-thread-1 task 2执行完毕
pool-1-thread-2 task 4执行完毕
```

线程池有什么优点?

- 降低资源消耗，提高资源利用率：重用存在的线程，减少对象创建销毁的开销，提高系统资源的利用率。
- 提高响应速度：可有效的控制最大并发线程数，同时避免过多资源竞争，避免堵塞。当任务到达时，任务可以不需要的等线程创建就能立即执行。
- 提高线程的可管理性：线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。
- 附加功能：提供定时执行、定期执行、单线程、并发数控制等功能。

综上所述使用线程池框架 Executor 能更好的管理线程、提高系统资源利用率。

线程池都有哪些状态?

1. **RUNNING**：这是最正常的状态，接受新的任务，处理等待队列中的任务。
2. **SHUTDOWN**：不接受新的任务提交，但是会继续处理等待队列中的任务。
3. **STOP**：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。
4. **TIDYING**：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为 TIDYING 状态时，会执行钩子方法 terminated()。
5. **TERMINATED**：terminated()方法结束后，线程池的状态就会变成这个。

在 Java 中 Executor 和 Executors 的区别?

- Executor 是 Java 线程池的核心接口，用来并发执行提交的任务
- ExecutorService 接口继承了 Executor 接口，并进行了扩展，提供了包括关闭线程池，获取任务的返回值等方法
- Executors 工具类提供一系列静态工厂方法用于创建各种各样的线程池。
- ThreadPoolExecutor 可以创建自定义线程池。

线程池中 submit() 和 execute() 方法有什么区别?

- 接收参数：execute()只能执行 Runnable 类型的任务。submit()可以执行 Runnable 和 Callable 类型的任务
- 返回值：submit()方法可以获取异步计算结果 Future 对象，而execute()没有
- 异常处理：submit()方便Exception处理

@$Executors和ThreaPoolExecutor创建线程池的区别

《阿里巴巴Java开发手册》中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程序的同学更加明确线程池的运行规则，规避资源耗尽的风险

Executors 各个方法的弊端：

- newFixedThreadPool 和 newSingleThreadExecutor：主要问题是堆积的请求处理队列可能会耗费非常大的内存，甚至 OOM。
- newCachedThreadPool 和 newScheduledThreadPool：主要问题是线程数最大数是 Integer.MAX_VALUE，可能会创建数量非常多的线程，甚至 OOM。

ThreaPoolExecutor创建线程池方式只有一种，就是走它的构造函数，参数自己指定

@$ThreadPoolExecutor构造函数重要参数分析

`ThreadPoolExecutor` **3 个最重要的参数**

- **corePoolSize** ： 核心线程数，定义了最小可以同时运行的线程数。
- **maximumPoolSize** ： 线程池中允许存在的最大工作线程数。
- **workQueue**：工作队列的长度。当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，任务就会被存放在队列中。

`ThreadPoolExecutor` 其他常见参数

1. **keepAliveTime**：线程池中的线程数大于 corePoolSize 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了 keepAliveTime才会被回收销毁；
2. **unit** ： keepAliveTime 参数的时间单位。
3. **threadFactory**：创建新线程的线程工厂
4. **handler** ： 当工作队列已满并且同时运行的线程数达到最大工作线程数时，新加入的任务就会走拒绝策略

@$ThreadPoolExecutor拒绝策略

`ThreadPoolExecutor` **拒绝策略定义**:

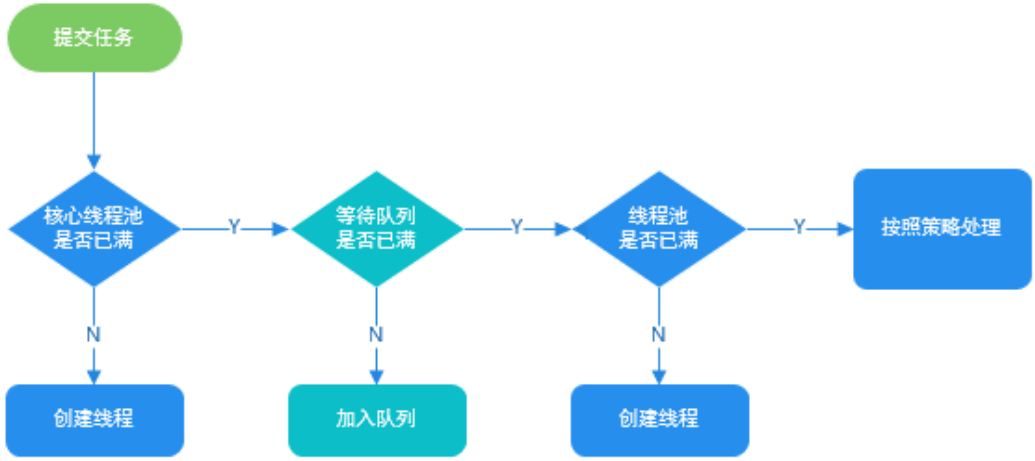
如果当工作队列已满并且同时运行的线程数达到最大工作线程数时， `ThreadPoolTaskExecutor` 定义一些策略：

- **ThreadPoolExecutor.AbortPolicy（默认）**：抛出 `RejectedExecutionException` 来拒绝新任务的处理。
- **ThreadPoolExecutor.CallerRunsPolicy**：用调用者所在的线程来执行任务。但是这种策略会降低对于新任务提交速度，影响程序的整体性能。
- **ThreadPoolExecutor.DiscardPolicy**：不处理新任务，直接丢弃掉。
- **ThreadPoolExecutor.DiscardOldestPolicy**：丢弃最早的未处理的任务。

举个例子：Spring 通过 `ThreadPoolTaskExecutor` 或者我们直接通过 `ThreadPoolExecutor` 的构造函数创建线程池的时候，当我们不指定 `RejectedExecutionHandler` 拒绝策略的话，配置线程池的时候默认使用的是 `ThreadPoolExecutor.AbortPolicy` 。在默认情况下， `ThreadPoolExecutor` 将抛出 `RejectedExecutionException` 来拒绝新来的任务 ，这代表你将丢失对这个任务的处理。对于可伸缩的应用程序，建议使用 `ThreadPoolExecutor.CallerRunsPolicy` 。

@$线程池实现原理，一个简单的线程池Demo：`Runnable`+`ThreadPoolExecutor`

线程池实现原理



首先创建一个 `Runnable` 接口的实现类（当然也可以是 `Callable` 接口）

```
import java.util.Date;

/**
 * 这是一个简单的Runnable类，需要大约5秒钟来执行其任务。
 */
public class MyRunnable implements Runnable {

    private String command;

    public MyRunnable(String s) {
        this.command = s;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " Start. Time = " + new Date());
        processCommand();
        System.out.println(Thread.currentThread().getName() + " End. Time = " + new Date());
    }

    private void processCommand() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String toString() {
        return this.command;
    }
}
```

编写测试程序，我们这里以阿里巴巴推荐的使用 `ThreadPoolExecutor` 构造函数自定义参数的方式来创建线程池。

```
import java.util.concurrent.ArrayBlockingQueue;
```

```
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class ThreadPoolExecutorDemo {

    private static final int CORE_POOL_SIZE = 5;
    private static final int MAX_POOL_SIZE = 10;
    private static final int QUEUE_CAPACITY = 100;
    private static final Long KEEP_ALIVE_TIME = 1L;
    public static void main(String[] args) {

        //使用阿里巴巴推荐的创建线程池的方式
        //通过ThreadPoolExecutor构造函数自定义参数创建
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                CORE_POOL_SIZE,
                MAX_POOL_SIZE,
                KEEP_ALIVE_TIME,
                TimeUnit.SECONDS,
                new ArrayBlockingQueue<>(QUEUE_CAPACITY),
                new ThreadPoolExecutor.CallerRunsPolicy());

        for (int i = 0; i < 10; i++) {
            //创建WorkerThread对象（WorkerThread类实现了Runnable 接口）
            Runnable worker = new MyRunnable("" + i);
            //执行Runnable
            executor.execute(worker);
        }
        //终止线程池
        executor.shutdown();
        while (!executor.isTerminated()) {
        }
        System.out.println("Finished all threads");
    }
}
```

可以看到我们上面的代码指定了：

1. `corePoolSize`：核心线程数为 5。
2. `maximumPoolSize` ：最大线程数 10
3. `keepAliveTime` ：等待时间为 1L。
4. `unit`：等待时间的单位为 TimeUnit.SECONDS。
5. `workQueue`：任务队列为 `ArrayBlockingQueue`，并且容量为 100;
6. `handler`：拒绝策略为 `CallerRunsPolicy`。

Output

```
pool-1-thread-2 Start. Time = Tue Nov 12 20:59:44 CST 2019
pool-1-thread-5 Start. Time = Tue Nov 12 20:59:44 CST 2019
pool-1-thread-4 Start. Time = Tue Nov 12 20:59:44 CST 2019
pool-1-thread-1 Start. Time = Tue Nov 12 20:59:44 CST 2019
pool-1-thread-3 Start. Time = Tue Nov 12 20:59:44 CST 2019
pool-1-thread-5 End. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-3 End. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-2 End. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-4 End. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-1 End. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-2 Start. Time = Tue Nov 12 20:59:49 CST 2019
```

```
pool-1-thread-1 Start. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-4 Start. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-3 Start. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-5 Start. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-2 End. Time = Tue Nov 12 20:59:54 CST 2019
pool-1-thread-3 End. Time = Tue Nov 12 20:59:54 CST 2019
pool-1-thread-4 End. Time = Tue Nov 12 20:59:54 CST 2019
pool-1-thread-5 End. Time = Tue Nov 12 20:59:54 CST 2019
pool-1-thread-1 End. Time = Tue Nov 12 20:59:54 CST 2019
```

原子操作类

什么是原子操作？在 Java Concurrency API 中有哪些原子类(atomic classes)?

处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。在 Java 中可以通过锁和循环 CAS 的方式来实现原子操作。CAS 操作——Compare & Set，或是 Compare & Swap，现在几乎所有的 CPU 指令都支持 CAS 的原子操作。

原子操作（atomic operation）是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

int++并不是一个原子操作，所以当一个线程读取它的值并加 1 时，另外一个线程有可能会读到之前的值，这就会引发错误。

为了解决这个问题，必须保证增加操作是原子的，在 JDK1.5 之前我们可以使用同步技术来做到这一点。到 JDK1.5，java.util.concurrent.atomic 包提供了 int 和long 类型的原子包装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。

java.util.concurrent 这个包里面提供了一组原子类。其基本的特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他性，即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，然后由 JVM 从等待队列中选择一个线程进入，这只是一种逻辑上的理解。

原子类：AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference

原子数组：AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray

原子属性更新器：AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater

解决 ABA 问题的原子类：AtomicMarkableReference（通过引入一个 boolean来反映中间有没有变过），AtomicStampedReference（通过引入一个 int 来累加来反映中间有没有变过）

说一下 atomic 的原理?

Atomic包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

AtomicInteger 类的部分源码：

```
// setup to use Unsafe.compareAndSwapInt for updates（更新操作时提供“比较并替换”的作用）
private static final Unsafe unsafe = Unsafe.getUnsafe();
private static final long valueOffset;

static {
    try {
        valueOffset = unsafe.objectFieldOffset
        (AtomicInteger.class.getDeclaredField("value"));
    } catch (Exception ex) { throw new Error(ex); }
}

private volatile int value;
```

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

CAS的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。UnSafe 类的 objectFieldOffset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffset。另外 value 是一个volatile变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。

并发工具

常用的并发工具类有哪些？在 Java 中 CycliBarriar 和 CountdownLatch 有什么区别?

- **Semaphore(信号量)-允许多个线程同时访问某个资源**：synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。
- **CountDownLatch(倒计时器)**：CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行。强调一个线程等多个线程完成某件事情。CountDownLatch方法比较少，操作比较简单。CountDownLatch是不能复用的。

  原理：任务分为N个子线程去执行，state也初始化为N（注意N要与线程个数一致）。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS(Compare and Swap)减1。等到所有子线程都执行完后(即state=0)，会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后余动作。
- **CyclicBarrier(循环栅栏)**：CyclicBarrier 和 CountDownLatch 非常类似，它也可以实现线程间的技术等待，但是它的功能比 CountDownLatch 更加复杂和强大。主要应用场景和 CountDownLatch 类似。CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行。CyclicBarrier是多个线程互等，等大家都完成，再携手共进。CyclicBarrier是可以复用的。