1 并发编程

1.1 启动线程的三种方式

- extends Thread
- runable
- callable 可返回结果

1.2 run() start()

run方法就是普通对象的普通方法，不会启动线程；

只有调用了start()后，Java才会将线程对象和操作系统中实际的线程进行映射，再来执行run方法。

1.3 yield sleep wait

- yield：是线程类（Thread）的静态方法， 它让掉当前线程 CPU 的时间片，使正在运行中的线程重新变成就绪状态，并重新竞争 CPU 的调度权。它可能会获取到，也有可能被其他线程获取到。对象的机锁没有被释放 。
- sleep：方法是线程类（Thread）的静态方法，让调用线程进入睡眠状态，让出执行机会给其他线程，等到休眠时间结束后，线程进入就绪状态和其他线程一起竞争cpu的执行时间。因为sleep() 是static静态的方法，他不能改变对象的机锁，当一个synchronized块中调用了sleep() 方法，线程虽然进入休眠，但是对象的机锁没有被释放，其他线程依然无法访问这个对象。
- wait()：wait()是Object类的方法，当一个线程执行到wait方法时，它就进入到一个和该对象相关的等待池，同时释放对象的机锁，使得其他线程能够访问，可以通过notify，notifyAll方法来唤醒等待的线程

1.4 join

线程A，执行了线程B的join方法，线程A必须要等待B执行完成了以后，线程A才能继续自己的工作

1.5 线程池（池化技术）

1.5.1 什么是线程池

线程池就是提前创建若干个线程放到一个容器中，需要的时候从容器中获取线程不用自行创建，使用完毕后不需要销毁线程而是放回到容器中，从而减少创建和销毁线程对象的开销。

1.5.2 为什么要使用线程池

1. 降低资源的消耗。降低线程创建和销毁额资源消耗。
2. 提高响应速度。线程的创建时间为T1，运行时间为T2，销毁时间为T3，线程池能降低T1和T2的时间。
3. 提高线程的可管理性。

1.5.3 实现一个我们自己的线程池

1. 线程必须在池子里已创建好，并且可以保持，要有容器保存多个线程。
2. 线程还要能够接受任务，运行任务，容器要保持来不及运行的任务。

1.5.4 JDK中的线程池和工作机制

1.5.4.1 线程池的创建

ThreadPoolExecutor所有线程实现的父类

1.5.4.2 各个参数的含义

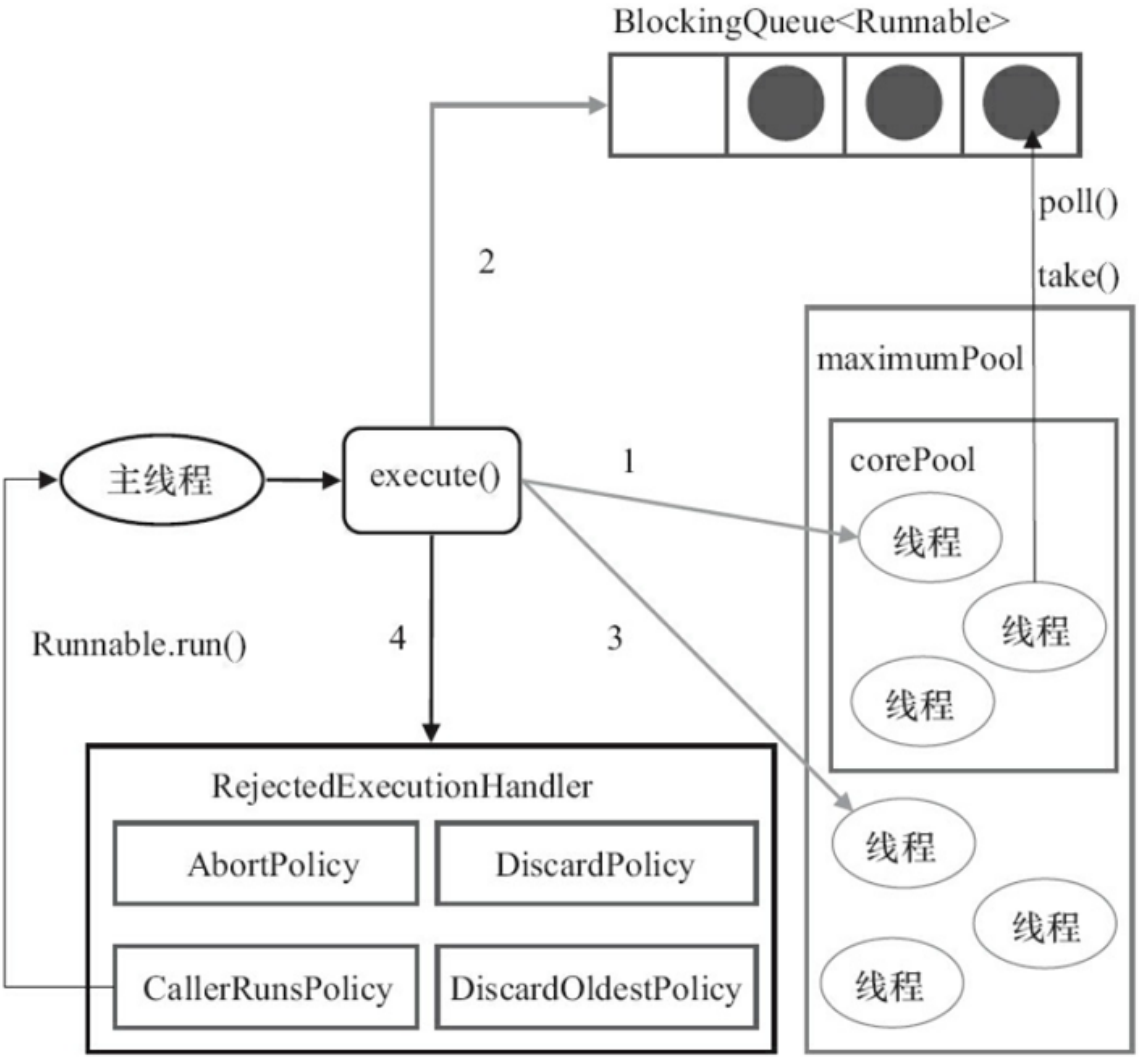
1. int corePoolSize：线程池中核心线程数， 任务数< corePoolSize，创建新线程；任务数=corePoolSize，这个任务就会保存到BlockingQueue。调用preStartAllCoreThreads()就会一次性的启动corePoolSize个线程。
2. int maxnumPoolSize：允许的最大线程数，BlockingQueue满了，任务数小于maxnumPoolSize的时候，就会再次创建新的线程。
3. log keepAliveTime：线程存活下来，存活的时间。在任务数大于corePoolSize时才有效。
4. TimeUnit unit：存活时间单位。
5. BlockingQueue workQueue：保存任务的阻塞队列。
6. ThreadFacktory threadFactory：线程工厂，给新建的线程取名字。
7. RejectExecutionHandler handler:任务饱和时的拒绝策略
    - AbrotPolicy：直接抛出异常，默认。
    - CallerRunsPolicy：用调用者所在的线程执行任务。
    - DiscardOldestPolicy：丢弃阻塞队列里最老的任务。
    - DiscardPolicy：当前任务直接丢弃。

    实现自己的饱和策略，实现RejectExecutionHandler即可。

1.5.4.3 提交任务

- execute(Runable command)：不需要返回
- Future submit(Call task)：需要返回值

1.5.4.4 工作机制




```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    /*
     * Proceed in 3 steps:
     *
     * 1. If fewer than corePoolSize threads are running, try to
     * start a new thread with the given command as its first
     * task.  The call to addWorker atomically checks runState and
     * workerCount, and so prevents false alarms that would add
     * threads when it shouldn't, by returning false.
     *
     * 2. If a task can be successfully queued, then we still need
     * to double-check whether we should have added a thread
     * (because existing ones died since last checking) or that
     * the pool shut down since entry into this method. So we
     * recheck state and if necessary roll back the enqueuing if
     * stopped, or start a new thread if there are none.
     *
     * 3. If we cannot queue task, then we try to add a new
     * thread.  If it fails, we know we are shut down or saturated
     * and so reject the task.
     */
    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) { //任务数小于核心线程数
        if (addWorker(command, true))    //创建新线程
            return;
```

```
            c = ctl.get();
        }
        if (isRunning(c) && workQueue.offer(command)) { //任务保存到BlockingQueue
中
            int recheck = ctl.get();
            if (! isRunning(recheck) && remove(command))
                reject(command);
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        else if (!addWorker(command, false))  //任务数小于maxnumPoolSize，创建新线程
            reject(command);//拒绝任务
    }
```

1.5.4.5 关闭线程池

- shutdown()：设置线程池状态，只会中断所有没有执行任务的线程。
- shutdownNow()：设置线程状态，还会尝试停止正在运行或者暂停任务的线程。

1.5.4.6 合理配置线程池

1. 线程数配置

   计算机的任务根据任务性质，分为计算密集型（CPU）、IO密集型、混合型。

   - 计算密集型

     只利用CPU和内存的任务：加密、大数分解、正则

     线程数适当小一点，最大推荐值：CPU核心数+1（加1的原因：防止页缺失）

     CPU核心数=Runtime.getRuntime().availableProcessors()。

   - IO密集型

     读取文件、数据库连接、网络通讯

     线程数适当大一点，CPU核心数*2

   - 混合型

     当IO密集型~计算密集型，尽量拆分

     IO密集型>>计算密集型：拆分意义不大

2. 队列选择

   选择有界队列，无界队列会导致OOM。

1.5.4.7 预定义线程池

- **FixedThreadPool**

  固定线程数的线程池。每当提交一个任务就创建一个线程，当工作线程数到达到线程池初始最大值，则将任务提交到池队列中。它具有线程池提高程序效率和节省创建线程消耗开销的优点。但是它在线程池没有可运行任务时，也不会释放工作线程，还会占用一定的系统资源。适用于负载较重的服务器，使用了无界队列。

- **SingleThreadExecutor**

  单个线程的线程池，如果这个线程异常结束，会有新线程取代它，保证顺序执行。适用于保证顺序执行任务的情况，不会有多个线程活动，使用了无界队列。

- **CacheThreadPool**

可缓存线程池，如果线程池长度超过处理需求，可灵活回收空闲线程，若无可回收，则新建线程。可缓存线程池创建线程的数量没有限制；工作线程空闲一定时间（默认1分钟），则该线程自动终止；注意控制任务的数量，否则大量线程同时运行，会造成OOM。

适用于执行很多短期任务的情况，使用了SynchronousQueue

- **WorkStealingPool（jdk1.8）**

  创建一个拥有多个任务队列（以便减少连接数）的线程池。基于ForkJoinPool实现，能够合理的使用CPU进行对任务的操作（并行操作），适用于很耗时的操作。

- **ScheduledThreadPoolExecutor**

  定长的线程池，支持定时以及周期性的执行任务。

  - newSingleThreadScheduledExector：只包含一个线程，只需要单个线程执行周期任务，保证顺序的执行各个任务。
  - newScheduledThreadPool：可包含多个线程，执行周期任务，需适度控制后台线程数目。

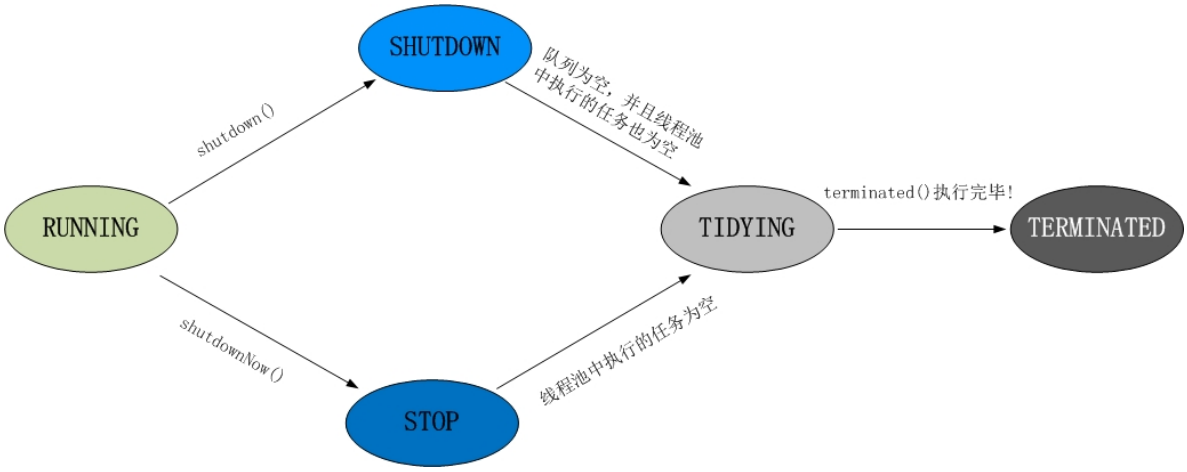
  方法说明：

  - schedule：只执行一次，任务可以延时。
  - scheduleAtFixedRate：提交固定时间间隔的任务。
  - scheduleWithFixedDelay：提交固定延时间隔执行的任务。

  提交给scheduleThreadPoolExecutor的任务要捕捉异常。

1.5.5 线程池都有哪些状态

线程池有5种状态：Running、ShutDown、Stop、Tidying、Terminated



- RUNNING

  (1) 状态说明：线程池处在RUNNING状态时，能够接收新任务，以及对已添加的任务进行处理。
  (2) 状态切换：线程池的初始化状态是RUNNING。换句话说，线程池被一旦被创建，就处于RUNNING状态，并且线程池中的任务数为0！

- SHUTDOWN

  (1) 状态说明：线程池处在SHUTDOWN状态时，不接收新任务，但能处理已添加的任务。
  (2) 状态切换：调用线程池的shutdown()接口时，线程池由RUNNING -> SHUTDOWN。

- STOP

  (1) 状态说明：线程池处在STOP状态时，不接收新任务，不处理已添加的任务，并且会中断正在处理的任务。
  (2) 状态切换：调用线程池的shutdownNow()接口时，线程池由(RUNNING or SHUTDOWN) -> STOP。

- TIDYING

(1) 状态说明：当所有的任务已终止，ctl记录的"任务数量"为0，线程池会变为TIDYING状态。当线程池变为TIDYING状态时，会执行函数terminated()。terminated()在ThreadPoolExecutor类中是空的，若用户想在线程池变为TIDYING时，进行相应的处理；可以通过重载terminated()函数来实现。
(2) 状态切换：当线程池在SHUTDOWN状态下，阻塞队列为空并且线程池中执行的任务也为空时，就会由 SHUTDOWN -> TIDYING。
当线程池在STOP状态下，线程池中执行的任务为空时，就会由STOP -> TIDYING。

- TERMINATED

  (1) 状态说明：线程池彻底终止，就变成TERMINATED状态。
  (2) 状态切换：线程池处在TIDYING状态时，执行完terminated()之后，就会由 TIDYING -> TERMINATED。

1.5.6 线程池中 submit()和 execute()方法有什么区别?

- submit(Callable task)、submit(Runnable task, T result)、submit(Runnable task)归属于ExecutorService接口。
  execute(Runnable command)归属于Executor接口。ExecutorService继承了Executor。
- submit有返回值，返回 Future ，而execute没有
- submit()方便做异常处理。通过Future.get()可捕获异常。

1.6 在 java 程序中怎么保证多线程的运行安全

- 原子性： 提供互斥访问， 同一时刻只能有一个线程对数据进行操作，（atomic,synchronized）；
- 可见性： 一个线程对主内存的修改可以及时地被其他线程看到，（synchronized,volatile）；
- 有序性：一个线程观察其他线程中的指令执行顺序，由于指令重排序，该观察结果一般杂乱无序，（happens-before原则）。

1.7 死锁

死锁是指两个或两个以上的进程在执行过程中，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称为死锁进程。是操作系统层面的一个错误，是进程死锁的简称。

1.8 怎么防止死锁

- 死锁的四个必要条件：
  互斥条件：进程对所分配到的资源不允许其他进程进行访问，若其他进程访问该资源，只能等待，直至占有该资源的进程使用完成后释放该资源
- 请求和保持条件：进程获得一定的资源之后，又对其他资源发出请求，但是该资源可能被其他进程占有，此事请求阻塞，但又对自己获得的资源保持不放
- 不可剥夺条件：是指进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用完后自己释放
- 环路等待条件：是指进程发生死锁后，若干进程之间形成一种头尾相接的循环等待资源关系

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上

述条件之 一不满足，就不会发生死锁。 理解了死锁的原因，尤其是产生死锁的四

个必要条件，就可以最大可能地避免、预 防 解除死锁。

1.9 ThreadLocal 是什么？有哪些使用场景?

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。

1.10 说一下 synchronized 底层实现原理?

synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。

Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：

- 普通同步方法，锁是当前实例对象
- 静态同步方法，锁是当前类的class对象
- 同步方法块，锁是括号里面的对象

1.11 接口并行

```
//线程池
    static ExecutorService taskExe = Executors.newFixedThreadPool(10);
    @Override
    public String getUserInfo(String useId) {
        long ti = System.currentTimeMillis();
        Callable<JSONObject> queryUserInfo = new Callable<JSONObject>() {
            @Override
            public JSONObject call() throws Exception {
                String userInfo = remoteService.getUserInfo(useId);
                JSONObject userJson = JSONObject.parseObject(userInfo);
                return  userJson;
            }
        };

        Callable<JSONObject> queryMoneyInfo = new Callable<JSONObject>() {
            @Override
            public JSONObject call() throws Exception {
                String moneyInfo = remoteService.getUserMoney(useId);
                return JSONObject.parseObject(moneyInfo);
            }
        };

        FutureTask<JSONObject> userFutureTask = new FutureTask<>(queryUserInfo);
        FutureTask<JSONObject> moneyFutureTask =new FutureTask<>
(queryMoneyInfo);
        //创建线程
        taskExe.submit(userFutureTask); //http
        taskExe.submit(moneyFutureTask);

        JSONObject result = new JSONObject();
        try {
            result.putAll(userFutureTask.get()); //http返回前，get停留在此
            result.putAll(moneyFutureTask.get());
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }

        return result.toString();
    }
```

1.12 你是怎样控制缓存的更新

方案名	类型	实现思路
数据实时同步时效	增量/主动	强一致性，更新数据库后主动淘汰缓存，读请求更新缓存，为避免缓存雪崩，更新缓存的过程需要进行同步控制，同一时间只允许一个请求访问数据库，为了保证数据的一致性，还要加上缓存失效。
数据准实时更新	增量/被动	准一致性，更新数据库后，异步更新缓存，使用多线程技术或MQ实现。
任务调度更新	全量/被动	最终一致性，采用任务调度框架，按照一定频率更新。

绿色建筑大数据平台

单位时间内发电量统计（小时、日、月、年），发电、环境数据监测

监听redis，数据变化后，统计小时、日、月、年发电量，插入MySQL，淘汰缓存

客户端请求：优先查询缓存数据，缓存没有数据，查询MySQL数据

```
    private final ConcurrentHashMap<String, ReentrantLock> locks = new
ConcurrentHashMap<>();

    /**
     * 获取发电量
     *
     * @param userId
     * @return
     */
    public BigDecimal getGenerationCapacity(String userId) {
        BigDecimal generationCapacity = null;
        //1.从缓存加载数据
        generationCapacity = getGenerationCapacityFromCache(userId);
        //2.缓存有数据，直接返回
        if (generationCapacity != null) {
            return generationCapacity;
        }
        //竞争锁
        acquireLock(userId);
        try {
            generationCapacity = getGenerationCapacityFromCache(userId);
            //2.缓存有数据，直接返回
            if (generationCapacity != null) {
                return generationCapacity;
            }
```

```
            //3.如果没有数据，从MySQL中获取数据
            generationCapacity = getGenerationCapacityFromDB(userId);
            //4.数据库数据不为空，更新缓存
            if (generationCapacity != null) {
                upDateGenerationCapacityCache(userId);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            releaseLock(userId);
        }
        return generationCapacity;
    }

    private ReentrantLock getLockForKey(String userId){
        ReentrantLock lock = new ReentrantLock();
        //把新锁添加到locks中，如果成功（locks中不存在同一key的锁）使用新锁，如果失败使用
locks集合中的锁
        ReentrantLock previous = locks.putIfAbsent(userId, lock);
        return previous == null ? lock : previous;
    }

    /**
     * 从缓存获取数据
     *
     * @param userId
     * @return
     */
    private BigDecimal getGenerationCapacityFromCache(String userId) {
        BigDecimal generationCapacity = null;
        return generationCapacity;
    }

    /**
     * 更新缓存
     *
     * @param userId
     * @return
     */
    private void upDateGenerationCapacityCache(String userId) {

    }

    /**
     * 从DB获取数据
     *
     * @param userId
     * @return
     */
    private BigDecimal getGenerationCapacityFromDB(String userId) {
        BigDecimal generationCapacity = null;
        return generationCapacity;
    }

    /**
     * 竞争锁
     *
     * @param useId
```

```
     */
    private void acquireLock(String useId) {
        ReentrantLock lock = getLockForKey(useId);
        lock.lock();
    }

    /**
     * 释放锁
     *
     * @param useId
     */
    private void releaseLock(String useId) {
        ReentrantLock lock = getLockForKey(useId);
        if(lock.isHeldByCurrentThread()){
            lock.unlock();
        }
    }
```

1.13 调用第三方系统接口

1. 保证数据一致性：事务注解
2. 性能优化：编程式事务
3. CAS锁：状态机制 锁机制

- 锁或 synchronized 关键字可以实现原子操作，那么为什么还要用 CAS 呢，因为加锁或使用 synchronized 关键字带来的性能损耗较大，而用 CAS 可以实现乐观锁，它实际上是直接利用了 CPU 层面的指令，所以性能很高 。

1.14 原子操作CAS