# 并发与多线程

多线程优点

提高资源利用率，单线程执行的情况下，当执行到耗时时间较长的操作时，cpu会进行长时间等待，浪费资源，多线程可以充分利用资源

提高程序响应速度，单线处理请求，处理期间无法接受新的请求，多线程可以并发的执行请求

## 并发与并行的区别

并发指多任务以轮流切换的方式进行处理

并行指多任务同时进行处理

并发的环境中程序的封闭性被打破，主要有一下特点：

1.并发执行的程序之间具有制约关系，直接制约：一个程序的运行需要另一个程序的计算结果，间接制约：程序之间同时竞争资源

2.线程的执行时断断续续的。

3.当cpu有足够的计算能力，并发数设置合理，多线程并发执行可以提高程序的执行效率（并不能加快执行速度，只是在同一时间段可以“同时”处理许多任务）

进程是系统资源分配的基本单位，线程是cpu调度和分派的基本单位。

## Thread与Runnable

Thread继承自Runnable，同时构造方法接收Runnable接口的实现类，Runnable本身没有构造方法，只能通过new Thread(Runnable).start();来启动线程

Runnable是函数式接口支持lambda表达式，所以可以这样写

new Thread(()->{

@Override

Public void run(){

//TODO…

}

}).start();

## 并发包提供新的接口Callable

无论是继承Thread线程类还是实现Runnable接口，覆写run时都存在一个缺点：没有返回值，为了解决这个问题，1.5之后提供新接口java.util.concurrent.Callable

@FunctionalInterface

Public interface Callable<V>{

public V call() throws Exception;

}

Call()方法可以实现线程操作数据的返回，返回值由泛型V动态决定。

使用方法

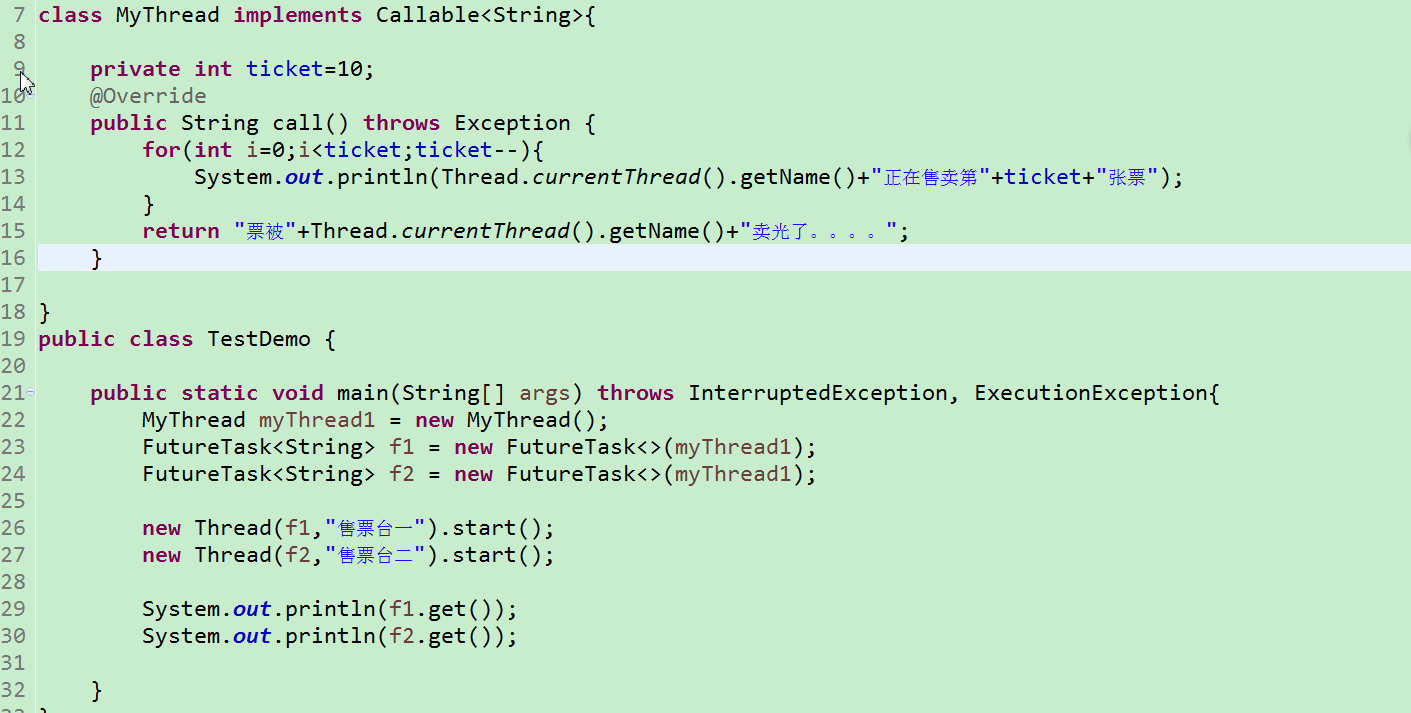
1. 实现Callable接口，覆写call()方法
2. 实例化FutureTask类，该类实现了RunnableFuture接口，这个接口同时实现了Runnable和Future接口
3. 通过Thread构造方法传入FutureTask实例
4. Start启动

FutureTask常用方法

public FutureTask(Callable<V> callable); 接收Callable实例

public FutureTask(Runnable runnable,V result); 接收Runnable实例，并指定返回类型

public V get() throws Iterrupted Exception,ExecutionException; 取得多线程操作结果



## Thread类中定义的线程操作方法

Public Thread(Runnable runnable,String name) 接收实例并取名，线程名拥有getter、setter操作方法

public static Thread currentThread() 取得当前线程对象

public static void sleep(long millis) 休眠，单位：毫秒

public final void setPriotity(int newPriority) 设置线程优先级，1-10越大越高

优先级常量：MAX\_PRIORITY、NORM\_PRIORITY、MIN\_PRIORITY 线程默认优先级为5

Object类提供三个线程操作方法

public final void wait() 线程等待，由运行转为阻塞，等待notify唤起

public final void notify() 唤醒第一个等待的线程

public final void notifyAll() 唤醒全部等待线程，由优先级决定哪一个先执行

Thread中还有stop()停止线程、suspend()暂时挂起线程、resume()恢复挂起的线程等操作方法，但是由于容易引起死锁问题而废弃了

线程停止的操作建议采用标志变量来决定线程是否继续执行,伪代码如下

Boolean t = true;

Run(){

While(t){

//TODO…

}

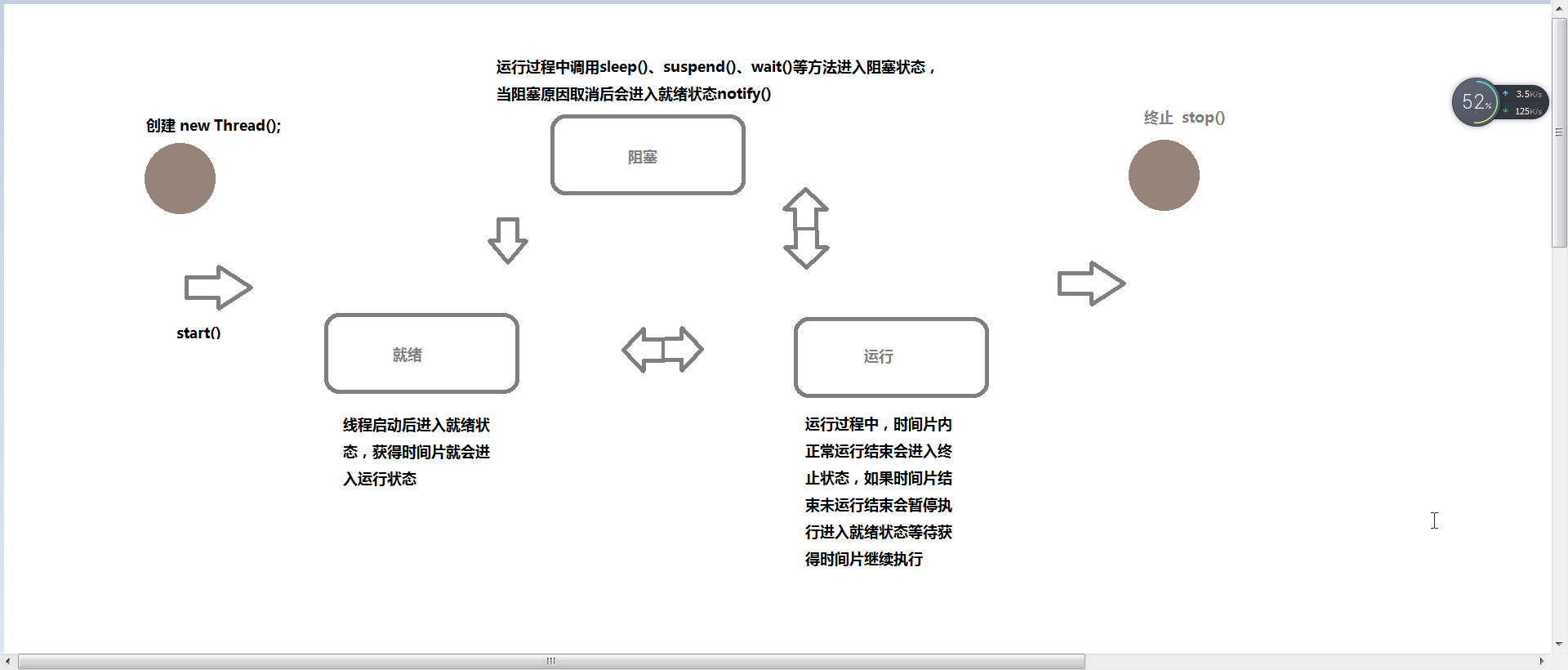
Stop(){

This.t = false;

}

}

线程生命周期



## 线程进阶

public final void join() 等待线程终止后继续执行

在很多情况下，主线程生成并起动了子线程，如果子线程里要进行大量的耗时的运算，主线程往往将于子线程之前结束，但是如果主线程处理完其他的事务后，需要用到子线程的处理结果，也就是主线程需要等待子线程执行完成之后再结束，这个时候就要用到join()方法了。

JDK中对join方法解释为：“**等待该线程终止**”，换句话说就是：”当前线程等待子线程的终止“。也就是在**子线程调用了join()方法后面的代码，只有等到子线程结束了当前线程才能执行**。

## 马士兵—Java并发

ReenTrantLock 与synchornized关键字实现差不多的功能，底层实现是AQS 1.5

Semaphore

CountDownLatch

ReentrantReadWriteLock

…

AQS 抽象队列同步器 用来实现锁

jvm锁竞争机制：偏向锁—>轻量级锁—>重量级锁

轻量级锁多数为自旋锁，自旋十次还是没拿到锁升级为重量级锁

线程加锁的时候视情况升级锁的类型



为什么需要新的锁的类型

Synchronized jvm-🡪OS 1.5之前属于重量级锁，之后进行了优化

关键字的实现是jvm与操作系统打交道，而新增的锁可以在jvm层面加入锁机制，某种程度上可以提高效率，新增的锁更加灵活

自旋锁就是不停轮询直至可以运行

自己实现锁，使用sync关键字，通过静态变量为标记值—虽然没什么用

Jdk实现了很多锁，基于AQS

CLH队列，实现锁的一种算法

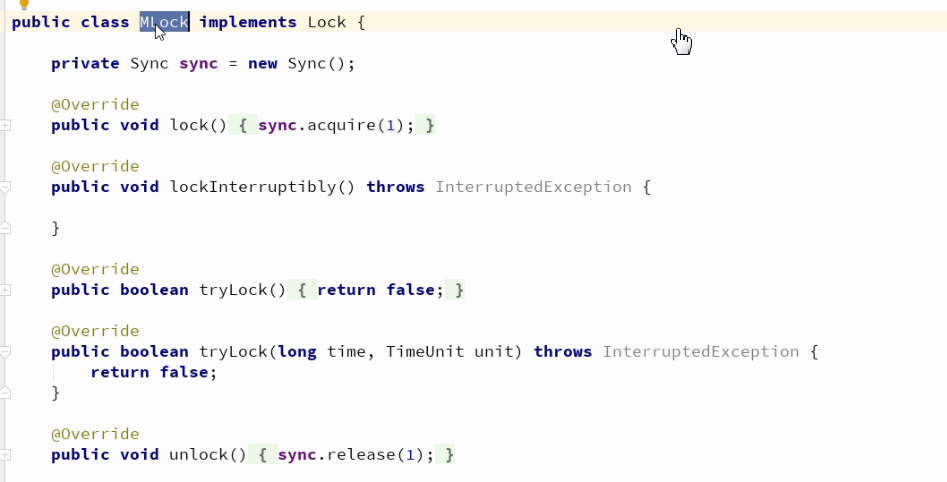
读文档，继承AQS实现锁，文档中有示例

CAS 比较和交换 比较特征值 如果不是预期则等待，是预期就改变值然后进入执行 自旋不断轮询 CAS过程必须原子操作，不能被其他线程打断

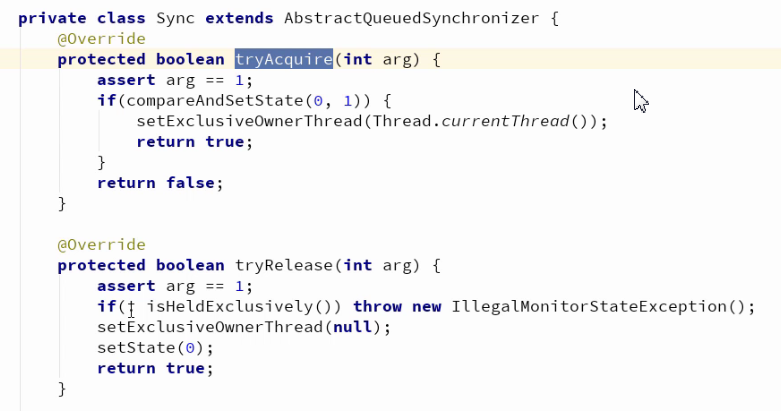
高并发并且同步代码执行时间长，那么sync关键字更加适合

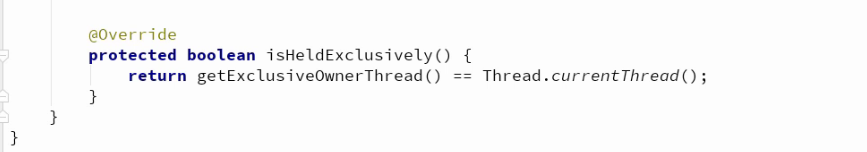
如果竞争不大，轮询次数很少就可以获得锁，那么CAS更适合

Sync wait操作时不消耗cpu资源，而CAS轮询时消耗资源的



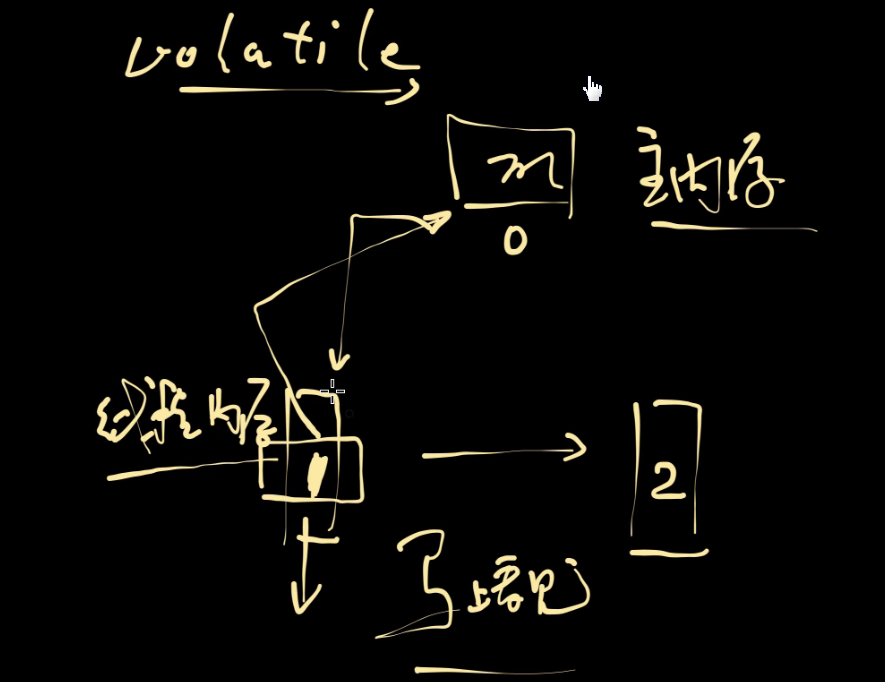






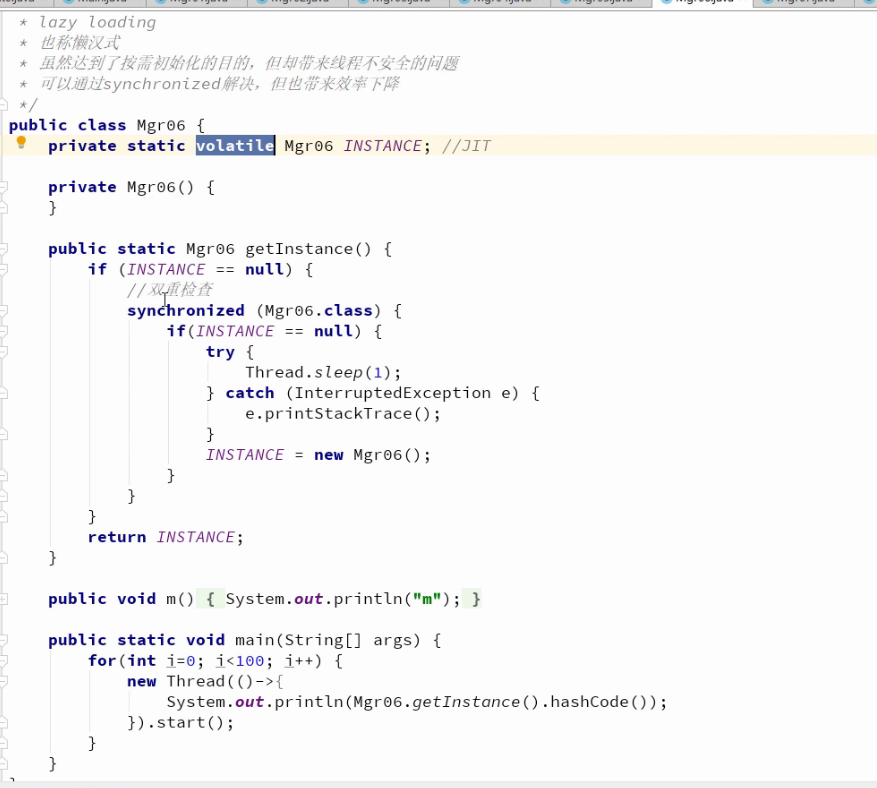
Volatile关键字

实时获取线程变量值的改变



禁止指令重排序

双重检查单例模式



在代码执行过程中，一条语句可能含有若干个指令操作，在指令操作过程中，jvm会对指令进行优化，进行指令重排，这样在高并发的情况下会导致程序发生错误，使用volatile关键字声明变量可以防止jvm进行这样的指令重排，防止发生错误

Synchronized关键字也可以防止指令重排，上面的代码INSTANCE判断的代码未在范围内，所以不能防止指令重排

说到单例模式，Effactive Java推荐的方式



也推荐恶汉式

CountDownLatch实现线程精确的结束

