什么样的对象是垃圾？

没有指针指向的对象

没有引用指向的对象

引用计数

Reference count

不能解决循环引用

GCroots（根可达算法）

根据main方法中的对象引用，如果没有引用就是垃圾

**GC垃圾回收机制**

Mark-Sweep（标记清除）

把垃圾找出来，做标记，说明这块内存可以被使用了

缺点：内存碎片化，一块可用一块不可用，内存中不能存储大的对象

Copying(拷贝)

无论有多大的内存都一分为二，只有内存的一半空间可以使用，当需要清除内存的时候，将有用的对象，放到没有用到的内存中

Mark-Compact（标记压缩）

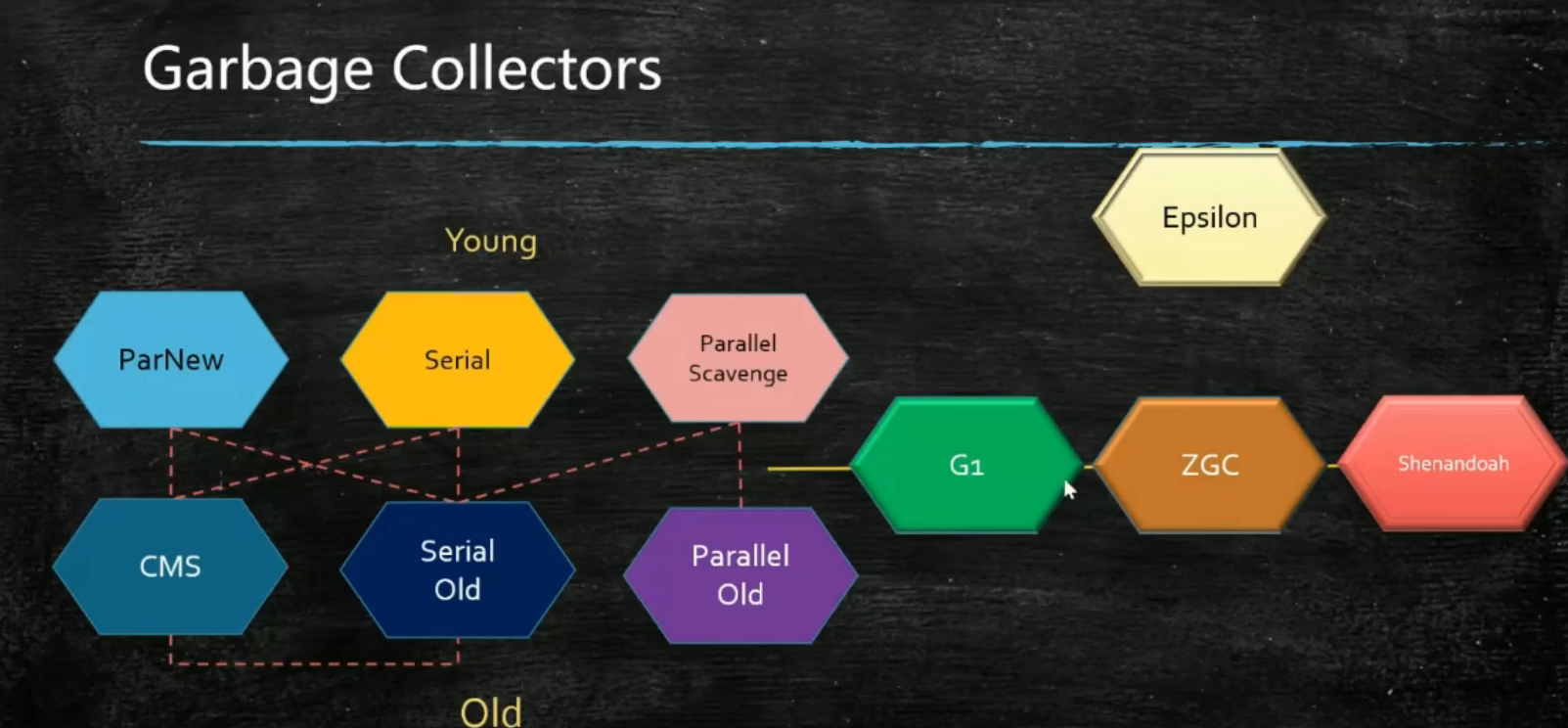
一边找垃圾一边排序，缺点，效率偏低

垃圾回收器的两种模型：

分代模型：年轻代、老年代

分区模型：将内存分为一块块区

Jdk目前十种垃圾回收器



在新生代区采用复制算法

新生代默认分为两种，三片区域，eden（伊甸）survivor（幸存者）survivor（幸存者），比例是8：1：1

新生成的对象在伊甸区，经过回收之后，幸存的对象放在幸存者区，然后将整个伊甸区干掉，清空掉，之后产生的一批对象，又经历一次垃圾回收，那么第二次的幸存者和第一个幸存者区的对象，放在第二个幸存者区中，然后将伊甸区和第一个幸存者区一起清空，每次只有一个幸存者区是空着的，当年龄到达一定年纪后，放到老年代区

老年代区算法标记清除或者标记压缩

STW:stop-the-world，当垃圾回收器开始工作的时候，所有的线程停止

Serial 最古老的垃圾回收器，利用复制算法，单线程工作效率最高

Parallel Scavenge，复制算法，多线程工作，但是线程不能无休止的增加

CMS 并发执行，业务线程和GC线程并发执行，

浮动垃圾：因为是并发执行的，当一开始a->b，GC标记为有用，但是之后业务线程中a不引用b，实际上b真正是垃圾，但是被标记为有用，称为浮动垃圾，下次清理时清理掉

CMS采用三色标记算法。

三色标记算法：

白色没有标记的对象

灰色已经标记这个对象，但是没有标记这个对象的孩子

黑色已经标记这个对象和所有的孩子对象

Incremental Update：通过写屏障JVM会帮助监视黑色，如果黑色指向白色，则把黑色变成灰色，当GC线程重启的时候，会从灰色开始

BUG：当垃圾回收线程m1标记A中两个属性1.2，已经标记完1，正在标记2的时候，线程停止，业务线程将1的引用指向D，此时，JVM将A变成灰色，但是，此时垃圾回收线程恢复工作，因为1已经标记完成，2正在标记，那么2标记完之后，A又变成黑色了，导致D又被漏标

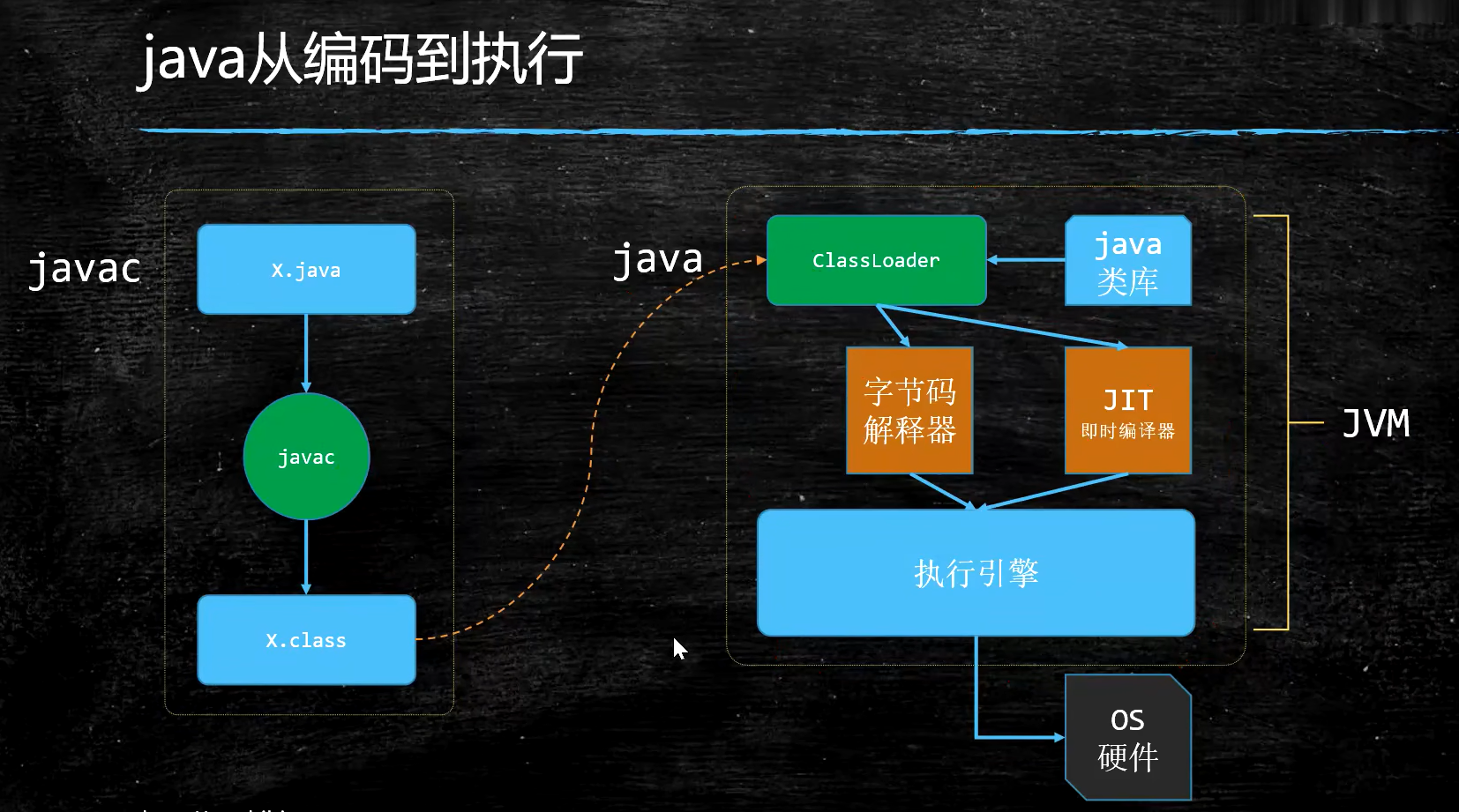
解决BUG：CMS的remark阶段，必须从头扫描一遍，重新扫描的时候STW，也是比较耗时

G1

依然使用三色标记法

解决方案SATB：当灰色指向白色的引用消失时，会把引用记录下来，当GC扫描回来后，回来根据引用找到白对象，来检测有没有黑对象指向白对象，没有就是垃圾

在区上面会有一个记忆集，记录有哪些区引用了这个区



JVM是一种规范

虚构出来的一台计算机

字节码指令级

内存管理：栈堆方法区等

JVM虚拟机，执行class

Jre ： jvm+core lib 运行环境

JDK : jre+development kit

JVM内存结构“

Classloader：加载类的字节码文件

方法区：类放在方法区

堆：实例

虚拟机栈：线程运行时需要的内存空间

程序计数器：线程独有。记录下一条指令执行地址，对应在CPU中的寄存器（CPU中读取最快的）

本地方法栈：

执行引擎：

解释器：从程序计数器中取到指令的地址，转义成机器码交给CPU执行

即时编译器：会根据平台类型，生成特定的机器码

垃圾回收：

程序计数器：

记住下一条jvm指令的执行地址，对应在CPU中的寄存器（CPU中读取最快的）

特点：

·线程独有，当多个线程执行时，线程A的时间片用完了，切换到线程B，则会将A还未执行的地址放到程序计数器，当重新获取时间片时，解释器重新从程序计数器中取到指令执行地址。

·不会存在内存溢出，JAVA虚拟机规范

虚拟机栈：线程运行时需要的内存空间,每一个方法都叫一个栈帧，在每个虚拟机栈中只有一个活动栈帧，栈帧中存有参数，局部变量，返回地址

特点：

先进后出，后进先出。

问题：

1. 垃圾回收是否涉及栈内存？ 不会，垃圾回收只会回收堆内存，方法调用后栈帧会弹出
2. 虚拟机栈内存越大越好？不是，因为物理内存固定，虚拟机栈内存分配的越大，线程越少
3. 方法内部的局部变量是否线程安全？如果变量没有逃离方法作用域，则线程安全

栈内存溢出：

1. 栈帧过多
2. 栈帧过大

线程运行诊断

CPU占用过高：

先定位到那个进程，在定位到线程

再用jstack+进程id 根据线程id找到有问题的线程，进一步找到问题代码的源码行数

程序运行很长时间没有结果：

线程发生死锁，同样用jstack查找死锁位置

本地方法栈：java调用本地方法时内存空间

堆：通过new创建的对象都会使用堆内存

特点：线程共享，要考虑线程安全问题，有垃圾回收机制

堆内存溢出：

1. jps工具
   1. 查看当前系统中有哪些java进程
2. jmap工具
   1. 查看对内存占用情况 jamp-heap 进程id
3. jconsole工具
   1. 图形界面，多功能的检测工具。可以连续监测

垃圾回收后，内存占用仍然很高

Jvisualvm 虚拟机可视化工具

方法区：

用于存储成员变量，构造器，运行时常量池，成员方法，类加载器

1.6之前是永久代（堆内存），1.8以后在metaspace（内存）

方法区内存溢出：

1.8以前会导致永久代溢出

1.8以后会导致元空间溢出（因为元空间存储在系统内存中，所以一般不会导致元空间溢出），配置元空间内存溢出-XX:MaxMetaspaceSize=8m

自定义类加载器 extends classLoader

// ClassWriter作用时生成类的二进制字节码

//Cw.visit（版本号1.8，访问修饰符，类名，包名，继承父类，实现接口）

//Cw.toByteArray();返回二进制字节码

Test.defineClass();执行类加载//加载类对象

方法区溢出场景：spring mybatis

运行时常量池：

常量池就是一张表，虚拟机指令根据这张常量表找到要执行的类名、方法名、参数类型、字面量等信息

运行时常量池，常量池就是存储在class文件中，当该类被加载，他的常量池信息就会放入运行时常量池，并且把符号地址换成真实地址

StringTable：串池[ ] ->hashTable结构，不能扩容

当常量池加载到运行时常量池时，abc都是常量池中的符号，还没有变成java对象

ldc #2 会把a符号变成“a”字符串对象

会先根据#2在串池中找对应的值，如果没有就会将#2对应的值放到串池，

字符串拼接，实际上是StringBuilder拼接，先创建StringBuilder对象，然后拼接toString（）底层是new String（）

StringTable特性

1. 常量池中的字符串仅是符号，第一次用到时才变为对象
2. 利用串池机制，来避免重复创建字符串对象
3. 字符串变量拼接的原理时StringBuilder（1.8）
4. 字符串常量拼接的原理是编译器优化
5. 可以使用intern方法，主动将串池中还没有的字符串对象

1.8将这个字符串对象尝试放入串池，如果有，不会放入，如果没有就会放入串池，会把串池中的对象返回->堆中串池回收只需要minorGC

1.6将这个字符串对象尝试放入串池，如果有则并不会放入，如果没有则会把对象复制一份放入串池，会把串池对象返回->永久代回收需要执行fullGC

StringTable性能调优->底层是hash表

桶越多越快，-XX:StringTableSize=100000

直接内存

常见于NIO操作时，用于数据缓冲区

分配回收成本较高，但读写性能高

不受JVM内存回收管理

在java堆内存和系统内存中建立一个中间内存，操作系统和JVM都可以访问的一片内存，如果直接使用直接内存，那么会少一次文件拷贝

正常：磁盘->系统内存->堆内存

直接内存：磁盘->直接内存

GC回收的堆内存，Unsafe释放直接内存，主动调用freeMemory

垃圾回收

1. 如何判断对象可以回收

引用计数法：如果一个对象被引用了那么就+1，当对象引用计数是0，那么就会被当成垃圾回收。弊端->循环依赖。

可达性分析算法：

根对象：一些不可能成为垃圾的对象，如果对象被根对象直接或间接的引用，那么就会被当成垃圾回收

哪些对象可以作为根对象？

四种引用：

强引用：如果被根对象强引用，那么会被保留

软引用：仅有软引用引用该对象时，当垃圾回收后，内存不够时，会再次发起垃圾回收，回收软引用对象，可以配合引用队列来释放软引用自身

弱引用：当垃圾回收时，没有其他根对象引用他，就会被当成垃圾回收

->软引用区别：软引用当内存不足时才会被回收，弱引用一定会被回收

虚引用：必须配合引用队列使用，主要配合ByteBuffer使用，被引用对象回收时，会将虚引用入队，由Reference Handler线程调用虚引用相关方法释放直接内存

终结器引用：无需手动编码，但其内部配合引用队列使用，在垃圾回收时，终结器引用入队，再有Finalizer线程通过终结器引用找到被引用对象并调用finalize方法，第二次GC才能回收被引用对象

1. 垃圾回收算法
   1. 标记清除法->将不是根节点所直接或间接引用得对象放到内存空闲表中，效率较快，内存碎片
   2. 标记整理->没有内存碎片，整理牵扯到对象移动，效率低
   3. 复制->不会有内存碎片，需要占用双倍内存空间
2. 分代垃圾回收
   1. 新生代

伊甸园 幸存区From 幸存区TO

新生代垃圾回收（minor GC）

* 1. 老年代

当幸存者区的存活次数超过一定阈值时，会晋升老年代

当老年代内存空间不足时，就会触发一次full GC

总结->

对象首先分配在伊甸园区域

新生代空间不足时，触发minor GC

伊甸园和幸存区FROM 存活的对象采用复制算法，复制到to中，存活的对象年龄加一，并且交换From和To，

Minor gc会引发一次stop the world=>当发起minor gc时所有用户线程都会被停止，当minor gc执行完毕后，其他用户线程才可以继续

当对象寿命超过阈值时，会晋升至老年代，最大15次（4bit）

当老年代空间不足时，会先尝试触发一次minor GC，如果之后空间仍然不足，那么会触发一次Full GC STW时间更长=>

相关虚拟机参数

|  |  |
| --- | --- |
| 堆初始化大小 | -Xms |
| 堆最大大小 | -Xmx |
| 新生代大小 | -Xmn |
| 幸存区比例（动态） | -XX：IniticalSurvivor=ratio |
| 幸存区比例 | -XX:SurvivorRatio=ratio |
| 晋升阈值 | -XX：MaxTenuringThreshold |
|  |  |

1. 垃圾回收器
   1. 串行
      1. 单线程
      2. 堆内存较小，适合个人电脑
      3. 新生代是复制算法，老年代是标记整理
   2. 吞吐量优先
      1. 多线程
      2. 适合堆内存较大、多核CPU支持
      3. 单位时间内STW时间最短
      4. 新生代是复制算法，老年代是标记整理
   3. 响应时间优先\_CMS
      1. 多线程
      2. 适合堆内存较大、多核CPU支持
      3. 尽可能让单次STW时间最短
      4. 新生代是复制算法，老年代是标记清除

JDK9以后G1

同时注重吞吐量和低延迟，默认的暂停目标是200ms

超大堆内存时，会将堆划分为多个大小相等的Region，每个Region是1248M

整体上市标记+整理算法，每个区域之间是复制算法

1. 垃圾回收调优

**类加载器**

启动类加载器（Bootstrap ClassLoader） JAVA\_HOME/jre/lib 无法直接访问

拓展类加载器(Extension ClassLoader) JAVA\_HOME/jre/lib/ext 上级为null

应用程序类加载器(Application ClassLoader) 加载classpath下所有得类 上级为Extension

自定义类加载器 自定义 上级为Application ClassLoader

类加载流程

加载

验证

准备

解析

初始化

**逃逸分析**

JVM将执行状态分为五个层次

0层，解释器执行

1层，使用C1即时编译器执行（不带profiling）

2层，使用C1即时编译器编译执行（带基本profiling）

3层，使用C1即使编译器编译执行（带完全的profiling）

4层，使用C2即时编译器编译执行

Profiling是指运行过程中收集一些程序执行状态得数据，例如[方法的调用次数]，[循环的回边次数]等

即时编译器（JIT）和解释器的区别

解释器是将字节码解释为机器码。下次即使遇到相同的字节码，仍会执行重复的解释

JIT是将一些字节码编译成字节码，并存入Code Cache，下次遇到相同的代码，直接执行，无需再编译

计时器是将字节码解释为针对所有平台都通用的机器码

JIT会根据平台类型，生成平台特定的机器码

Volatile他保证的是多线程之间，一个线程对volatitle变量的修改是对另一个线程可见，不能保证原子性，仅用在一个写线程，多个读线程

Synchronized语句块既可以保证代码块的原子性，也同时保证代码块内变量的可见性。但缺点是synchronized是属于重量级锁，性能相对更低

多线程中-有序性：同一个线程内，JVM会在不影响正确性的前提下，可以调整语句的执行顺序。

CAS是基于乐观锁的思想：最乐观的涉及，不怕别的线程来修改共享变量，就算修改了，我再重试一边

Synchronized是基于悲观锁的思想：最悲观的涉及，得放着其他线程来修改共享变量，我上了锁你们都别想改，我改完了在解锁，

Java HotSpot虚拟机中，每个对象都有对象头，包括（class指针和MarkWord）。Mark Word平时存储这个对象的hashCode、分带年龄，当加锁时，这些信息就根据情况被替换成标记为、线程所记录指针，重量级锁指针、线程Id等内容

轻量级锁撤销-调用对象hashCode

当调用了对象得hashCode，但偏向锁得对象MarkWord中存储的是线程id，如果调用hashCode会导致偏向锁被撤销

※轻量级锁会在所记录中存储hashCode

※重量级锁会在Monitor中记录hashCode

轻量级锁撤销-其他线程使用对象

轻量级锁撤销-wait或notify：升级为重量级锁

批量重偏向：如果对象虽然被多个线程访问，但没有竞争，这时偏向了线程T1得对象仍有机会重新偏向T2，重偏向会重置对象的ThreadID

当撤销偏向锁阈值超过20次以后，jvm会觉得偏向错了，于是会在给这些对象加锁时重新偏向之加锁线程

批量撤销：当撤销偏向锁阈值超过40次，jvm会觉得自己根本不该偏向。于是整个类得所有对象都会变为不可偏向的。

ThreadPoolExecutor有七个参数

核心线程数

最大线程数

等待时间

等待时间单位

工作队列

线程工厂（为了给线程起名字）🡪有默认的

淘汰策略🡪有默认的

Executor创建的对象有三种

NewFixedThreadPool:

**public static** ExecutorService newFixedThreadPool(**int** nThreads) {  
 **return new** ThreadPoolExecutor(nThreads, nThreads,  
 0L, TimeUnit.***MILLISECONDS***,  
 **new** LinkedBlockingQueue<Runnable>());  
}

创建的队列是无长度的

NewCachedThreadPool

**public static** ExecutorService newCachedThreadPool(ThreadFactory threadFactory) {  
 **return new** ThreadPoolExecutor(0, Integer.***MAX\_VALUE***,  
 60L, TimeUnit.***SECONDS***,  
 **new** SynchronousQueue<Runnable>(),  
 threadFactory);  
}

SynchronousQueue实现特点是，没有容量，没有线程来取是放不进去的

NewSingleThreadExecutor：

**public static** ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory) {  
 **return new** FinalizableDelegatedExecutorService  
 (**new** ThreadPoolExecutor(1, 1,  
 0L, TimeUnit.***MILLISECONDS***,  
 **new** LinkedBlockingQueue<Runnable>(),  
 threadFactory));  
}

区别：

自己创建一个单线程串行执行任务，如果任务执行失败而终止那么没有任何补救措施，而线程池还会新建一个线程

Executors.newFixedThreadPool(1)初始值为1，以后还可以修改，对外暴露的是ThreadPoolExecutor对象，可以强转后调用setCorePoolSize等方法进行修改

Executors.newSingleThreadExecutor()线程个数是中为1，不能修改，FinalizableDelegatedExecutorService应用的是装饰器模式，只对外暴露了ExecutorService接口，因此不能调用ThreadPoolxecutor中特有的方法

提交任务

Future<泛型>Submit（Callable<泛型> c）Call方法有返回值

Execute(Runnable command)没有返回值

<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)  
 throws InterruptedException

invokeAny 只返回第一个执行完的任务得返回值，其他线程就不会继续往下执行

关闭线程池得方法

Shutdown：不会接受新任务，但已提交任务会执行完，打断空闲线程，此方法不会阻塞调用线程的执行。

ShutdownNow：不会接受新任务，会将队列中得任务返回，打断所有线程

AQS原理：是阻塞式锁和相关的同步器工具框架

用state属性来表示资源的状态（分独占模式和共享模式），子类需要定义如何维护这个状态，控制如何获取和释放锁

·getState-获取state状态

·setState-设置state状态

·compareAndSetState-乐观锁机制（CAS）设置state状态

·独占模式是只有一个线程能够访问资源，而共享模式可以允许多个线程访问资源

提供了基于FIFO得等待队列，类似于Monitor得EntryList

条件变量来实现等待，唤醒机制，支持多个条件变量，类似于Monitor得WaitSet

子类主要实现方法：

阻塞线程是：park unpark，

TryAcquire:获取锁

tryRelease:释放锁

ReentrantLock原理

同步器：fairsync ，nofairsync（默认）

非公平锁实现原理：调用 nonfairTryAcquire来获取锁，去过状态时0，直接用cas抢锁，不会检查AQS队列

公平锁实现原理：调用 tryAcquire，当state是0，先检查AQS队列是否有前驱节点，没有才去竞争

Await流程：ConditionObject中有firstWaiter和fastWaiter，同样也是双向链表，并将state设置为-2，将节点放入conditionObject，如果conditionObject是空，就会将节点放到first否则设置为下一个

Singal流程：首先判断调用方法得线程是不是当前持有锁得线程，如果不是就抛异常，如果是，就取conditionObject中的第一个元素，如果first不是空，将状态改成0

，就将该节点加入AQS等待队列，并且将AQS中最后一个节点得state设置成-1，竞争锁，

**读写锁**

ReentrantReadWriteLock

类似与数据库中的lock in share mode

用读锁保护数据得读取操作，用写锁保护数据得写操作

读读可以并发，读写互斥

读锁不支持条件变量

重入时升级不支持：即持有读锁的情况下取获取写锁，会导致写锁永久等待

重试时降级支持：即持有写锁得情况下去获取读锁

在锁降级的时候，在释放写锁之前，先获取读锁，保证在写锁释放的时候，还有读锁，防止其他线程进行写操作

读读并发原因：当唤醒读锁线程时会连续唤醒后面的读线程，直到遇到独占锁

Stamplock

乐观读，stampedLock支持tryOptimisticRead（）了官渡，读取完毕后需要做一次戳校验如果校验通过，表示这期间确实确实没有写操作，数据可以安全使用，如果校验没通过，需要重新获取读锁，保证数据安全

Semaphore

用来限制访问共享资源的线程上限

countdownLatch 用来进行线程同步写作，等待所有线程完成倒计时