# 进程

* 概念

是程序或者任务的执行过程(动态性)，持有资源(共享内存,共享文件)和线程。

一个具有一定独立功能的程序关于某个数据集合的一次运行活动，是系统进行资源分配和调度运行的基本单位

进程的概念主要有两点：

第一，进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括[文本](https://baike.baidu.com/item/%E6%96%87%E6%9C%AC)区域（text region）、数据区域（data region）和[堆栈](https://baike.baidu.com/item/%E5%A0%86%E6%A0%88)（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和期间使用进程执行的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。

第二，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有[处理](https://baike.baidu.com/item/%E5%A4%84%E7%90%86)器赋予程序生命时（操作系统执行之），它才能成为一个活动的实体，我们称其为[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B)

# 线程

1.概念：线程是系统中最小的执行单元，同一个进程中有多个线程，线程共享进程的资源

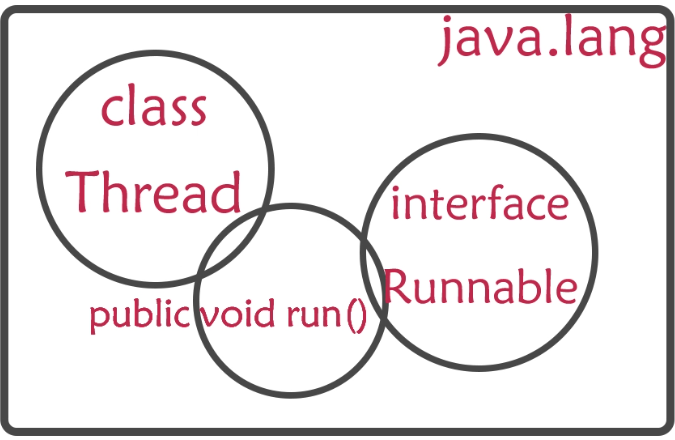
线程是进程的组成部分，一个进程可以拥有多个线程，一个线程必须有一个父进程。

线程可以拥有自己的堆栈、自己的程序计数器和自己的局部变量，但不拥有系统资源，它与父进程的其他线程共享该进程所拥有的全部资源。

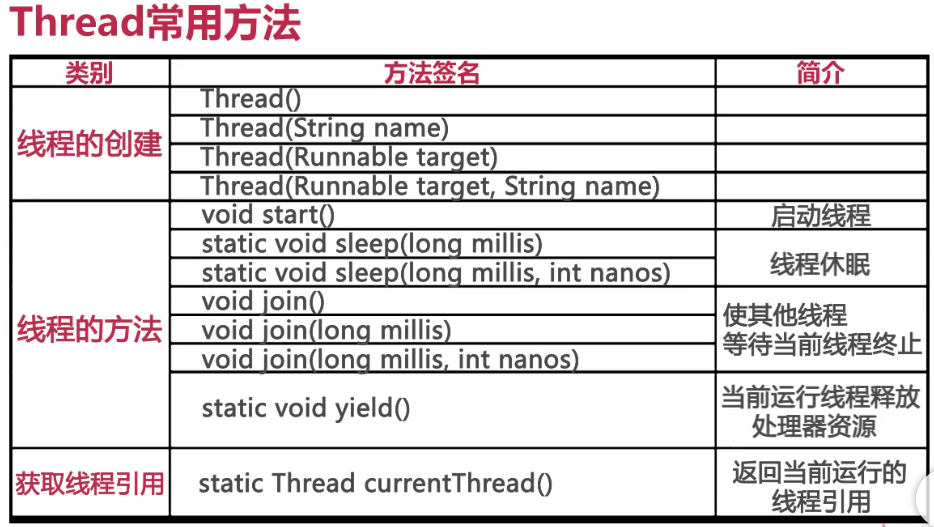
2.线程的交互：互斥和同步

3.线程的生命周期

4.Java对线程的支持:



5.Thread常用的方法：



yield()方法：当前正在处理器运行的线程，释放出处理器的资源，并且重新去竞争处理器的资源。

6.注意事项

1) 调用线程对象的start()方法之后，该线程立即进入就绪状态——就绪装填相当于“等待执行”，但该线程并未真正进入运行状态。

2)一个线程的资源占用接近1MB。

7.线程的生命周期：

1)当发生如下情况时，线程将会进入堵塞状态。

a线程调用sleep()方法主动放弃所占用的处理器资源

b线程调用了一个阻塞式的IO方法，在该方法返回之前，该线程被阻塞。

c程试图获得一个同步监视器，但该同步监视器正在被其他线程所持有。

d程在等待某个通知(notify)

e 程序调用了线程的suspend()方法将线程挂起。这个方法容易导致死锁，所以尽量避免使用该方法。

8.控制线程

Join线程

………

# 线程安全

线程不安全的定义：当多线程并发访问临界资源时，如果破坏原子操作，可能会造成数据不一致。

* 临界资源：共享资源(同一个对象)、一次仅允许一个线程使用，才能保证其正确性。
* 原子操作：不可分隔的多步操作、被视作一个整体，其顺序和步骤不可打乱或缺省。

# 线程安全和不安全的类

1.线程安全:StringBuffer

2.线程不安全:StringBuilder

# 线程中的面试题

## 1实现多线程的方式:

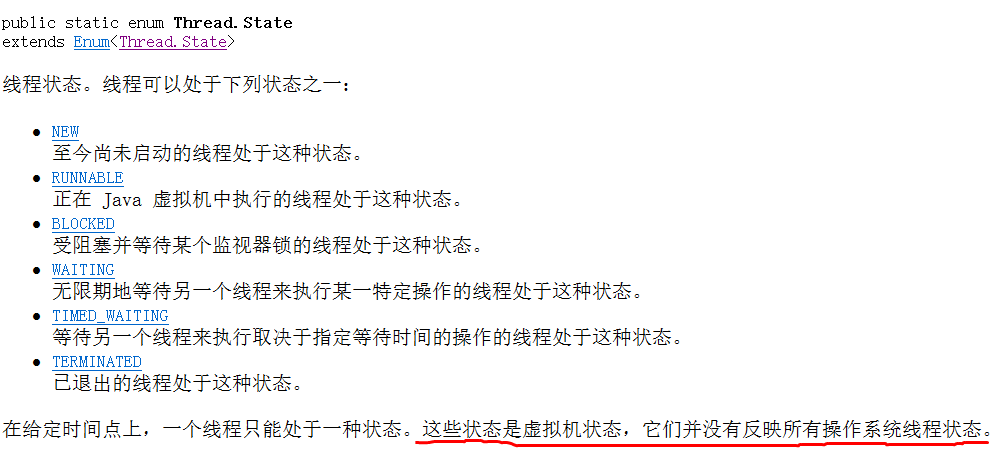
1. 继承Thread类（此类在java.lang包中）、2实现Runnable接口3.实现Callable接口（有返回值并且可以抛出异常）4.使用线程池

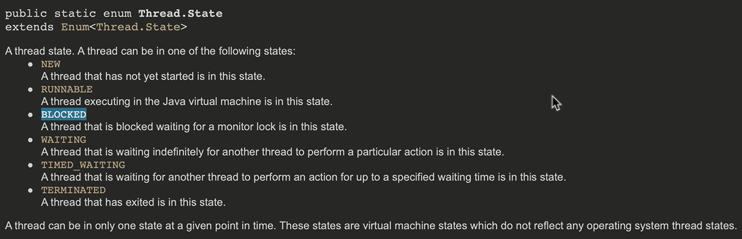
## Java中Runnable和Callable有什么不同

1. 接口定义的方法在实现类实现时无法抛出比接口或者父类更多的异常。Runnable接口中的run()方法没有抛出异常，而Callable中的Call方法抛出了异常。
2. Callable接口中的call方法有返回值，返回值是泛型

## 线程共有集中状态？分别是？

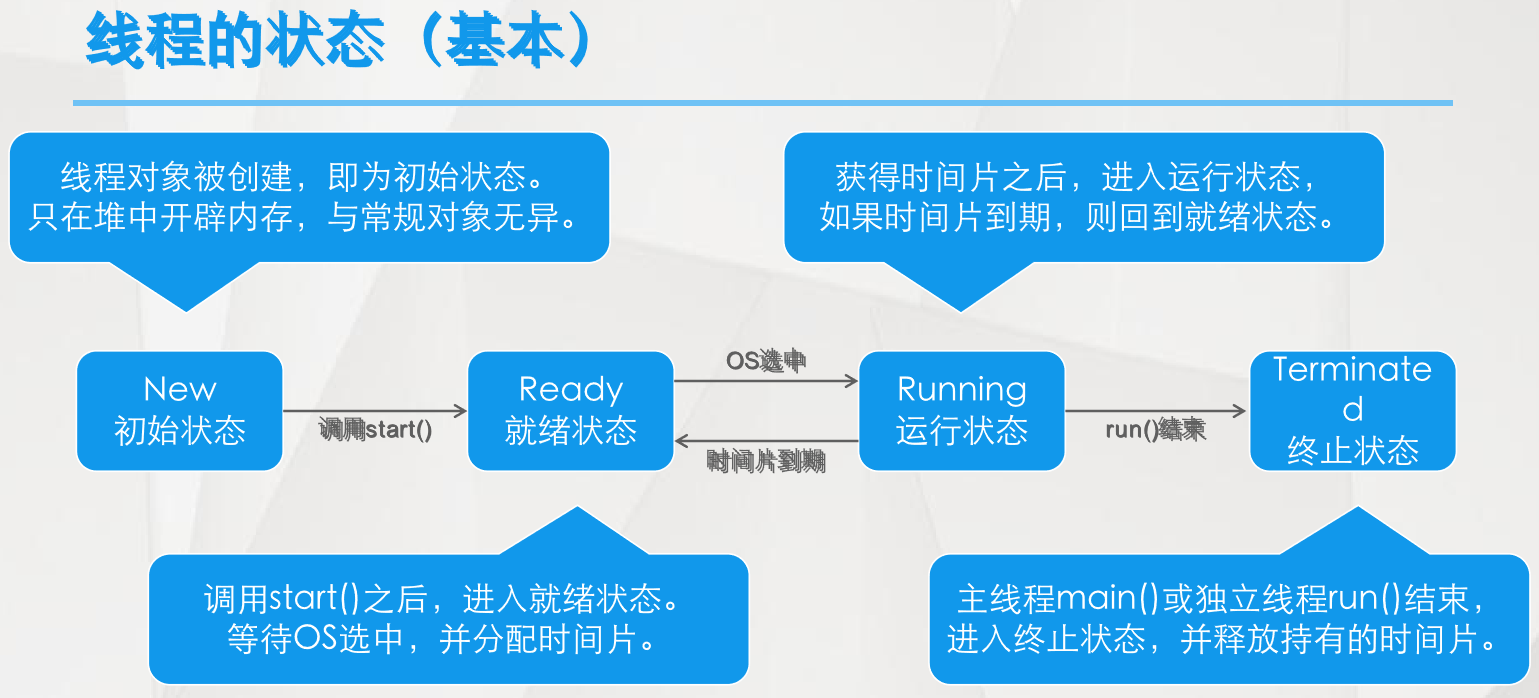
在java中的Thread.state枚举中规定了线程的状态

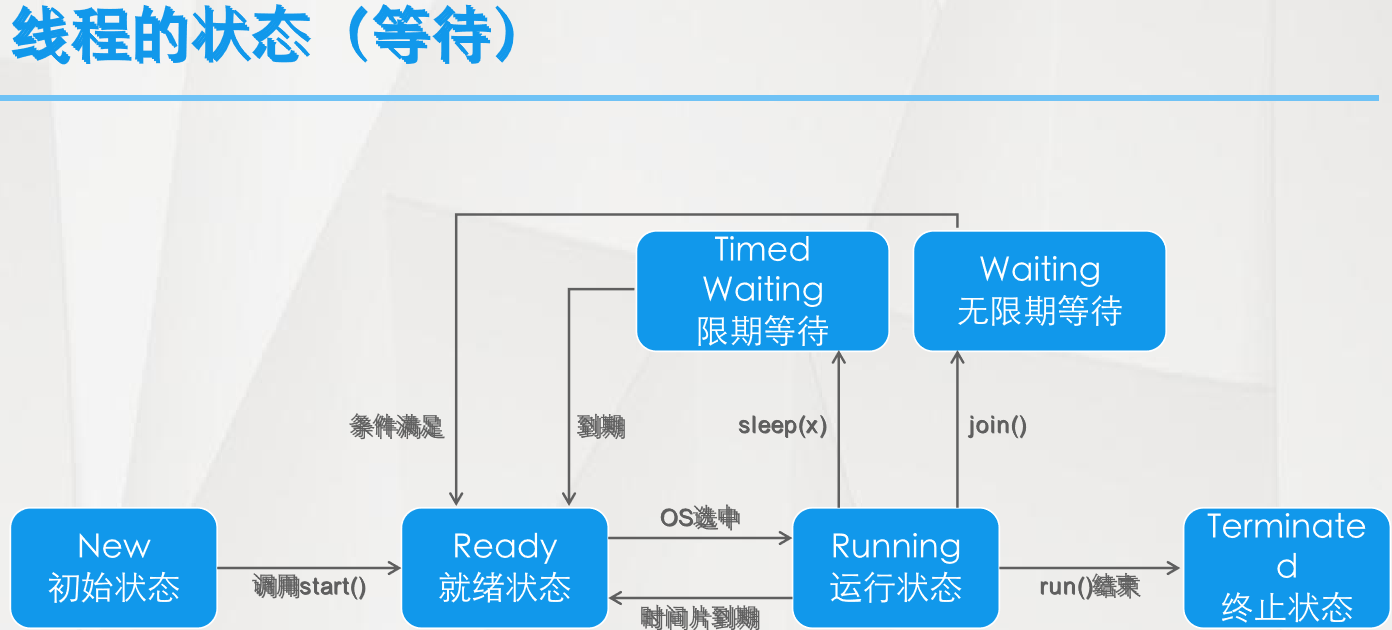


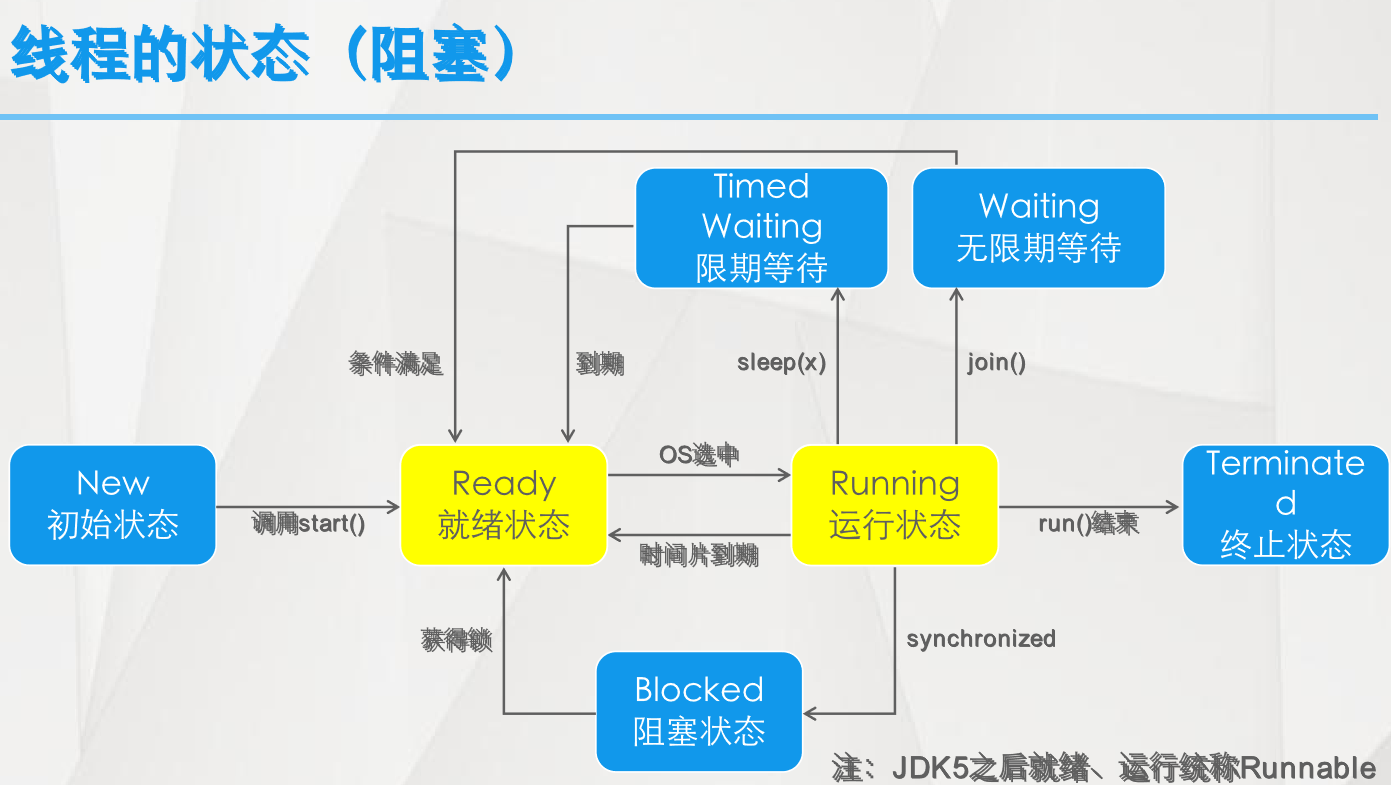


注:

* Running和Runnable是一个意思，不再区分是等待运行还是正在运行，都归为了Runnable.
* Blocked:堵塞，能进入阻塞状态的只有一种情况，那就是加了锁，要么是synchronized（隐式的监视器锁），要么是lock(显式的监视器锁).
* Waiting：无限期等待，Future中的get方法,是让调用它的线程处于此状态，等待get运算结果结束，调用的线程才会在继续执行。
* Time\_Wating:有限期等待。







## 什么是线程不安全？

线程不安全的定义：当多线程并发访问临界资源时，如果破坏原子操作，可能会造成数据不一致。

* 临界资源：共享资源(同一个对象)、一次仅允许一个线程使用，才能保证其正确性。
* 原子操作：不可分隔的多步操作、被视作一个整体，其顺序和步骤不可打乱或缺省。

乐观锁:不加锁

悲观锁:必加锁

加锁的方式:

解决的办法：加锁

1. 隐式监视器锁，无论是否发生异常都会自动释放锁。

同步代码块或者同步方法synchronized

案例:

1. **public** **class** TestSafe {
2. **public** **static** **void** main(String[] args) {
3. //        内部类访问外部类的局部变量需要加final
4. //        临界资源
5. **final** MyClass mc = **new** MyClass();

8. ExecutorService es = Executors.newFixedThreadPool(2);
9. es.submit(**new** Runnable() {
10. @Override
11. **public** **void** run() {
12. mc.add("D");
13. }
14. });
15. es.submit(**new** Runnable() {
16. @Override
17. **public** **void** run() {
18. mc.add("E");
19. }
20. });
21. //        ABCE
22. **for** (**int** i = 0; i < mc.size(); i++) {
23. System.out.println(mc.get(i));
24. }
25. }
26. }
27. **class** MyClass{
28. **int**  size = 3;
29. //    临界资源
30. **private** Object[] elements = **new** Object[]{"A","B","C","",""};
32. **public** **synchronized**   **void** add(Object obj){
33. elements[size] = obj;
34. //        如果在此切换CPU会出现问题。
35. size++;
36. }
37. **public** Object get(**int** index){
38. **return** elements[index];
39. }
40. **public** **int** size(){
41. **return** **this**.size;
42. }
43. }

2使用显示的监视器锁，Lock接口

必须要在finally中释放锁。

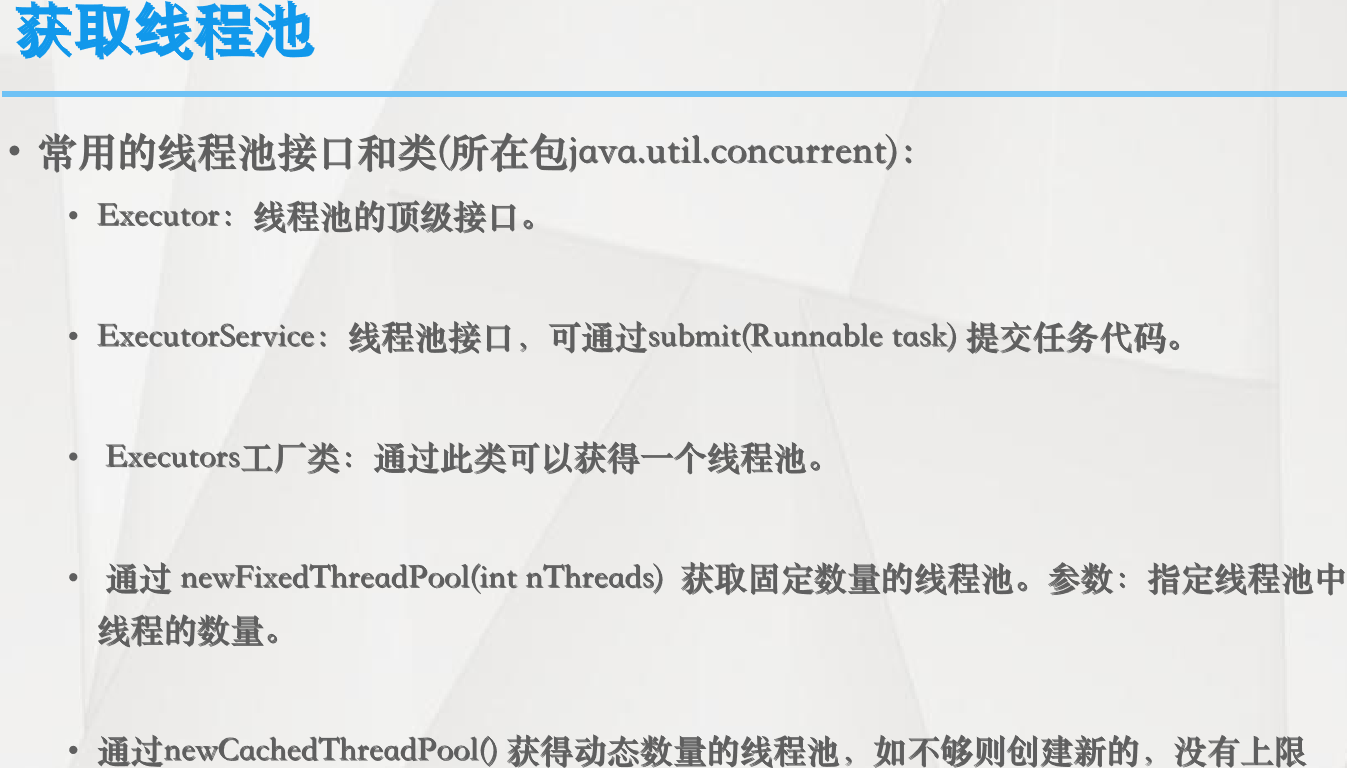
1. **public** **class** TestSafe {
2. **public** **static** **void** main(String[] args) {
3. //        内部类访问外部类的局部变量需要加final
4. //        临界资源
5. **final** MyClass mc = **new** MyClass();

8. ExecutorService es = Executors.newFixedThreadPool(2);
9. es.submit(**new** Runnable() {
10. @Override
11. **public** **void** run() {
12. mc.add("D");
13. }
14. });
15. es.submit(**new** Runnable() {
16. @Override
17. **public** **void** run() {
18. mc.add("E");
19. }
20. });
21. //        ABCE
22. **for** (**int** i = 0; i < mc.size(); i++) {
23. System.out.println(mc.get(i));
24. }
25. }
26. }
27. **class** MyClass{
28. **private** Lock lock = **new** ReentrantLock();
29. **int**  size = 3;
30. //    临界资源
31. **private** Object[] elements = **new** Object[]{"A","B","C","",""};
33. **public**  **void** add(Object obj){
34. lock.lock();
35. **try**{
36. elements[size] = obj;
37. //        切换CPU
38. size++;
39. }**finally** {
40. lock.unlock();
41. }
42. }
43. **public** Object get(**int** index){
44. **return** elements[index];
45. }
46. **public** **int** size(){
47. **return** **this**.size;
48. }
49. }

## 什么是ThreadLocal变量？实现原理是什么？

其中的map的key是thread对象

## 6什么是线程池？为什么要使用它？



Executor中的execute([Runnable](mk:@MSITStore:E:\qf_study\1and2_stage\API\JDK_API_1_6_zh_CN.CHM::/java/lang/Runnable.html) command)方法只能接收Runnable类型的参数，而ExecutorService的submit()方法可以提交Runnable的任务以及Callable的任务。

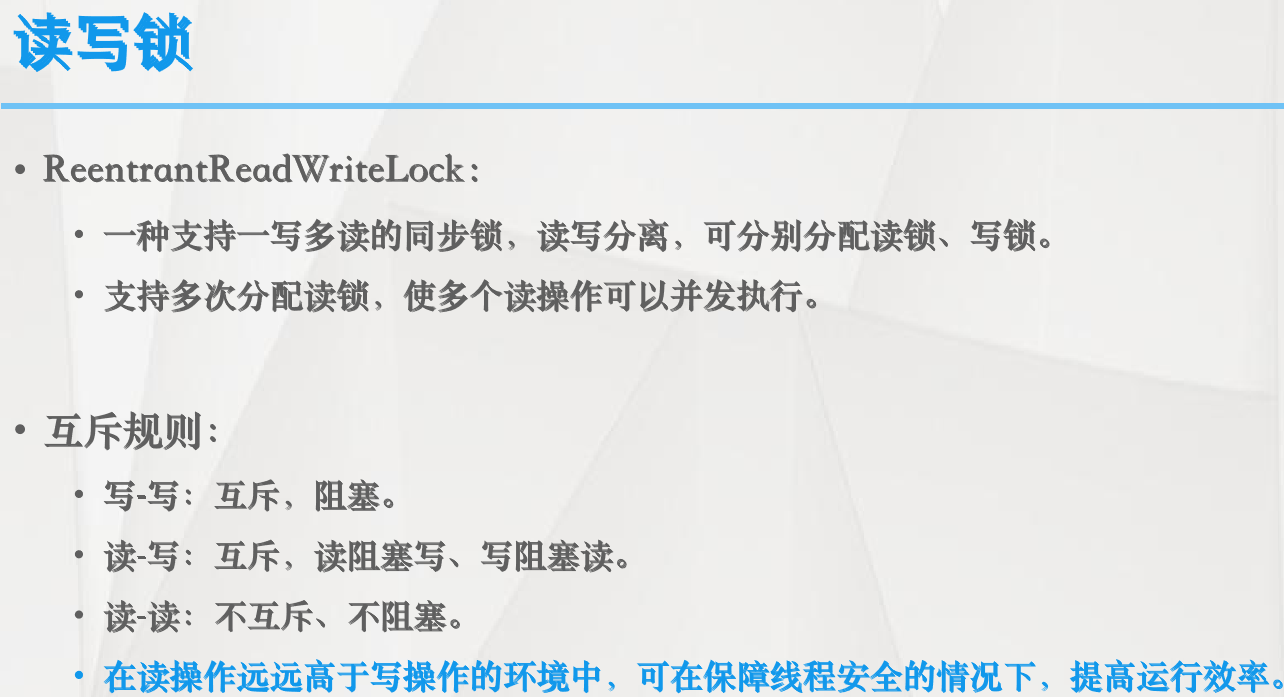
* [submit](mk:@MSITStore:E:\qf_study\1and2_stage\API\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/ExecutorService.html#submit(java.util.concurrent.Callable))([Callable](mk:@MSITStore:E:\qf_study\1and2_stage\API\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/Callable.html)<T> task) 返回值为: <T> [Future](mk:@MSITStore:E:\qf_study\1and2_stage\API\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/Future.html)<T>
* [submit](mk:@MSITStore:E:\qf_study\1and2_stage\API\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/ExecutorService.html#submit(java.lang.Runnable))([Runnable](mk:@MSITStore:E:\qf_study\1and2_stage\API\JDK_API_1_6_zh_CN.CHM::/java/lang/Runnable.html) task) 返回值为 [Future](mk:@MSITStore:E:\qf_study\1and2_stage\API\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/Future.html)<?>

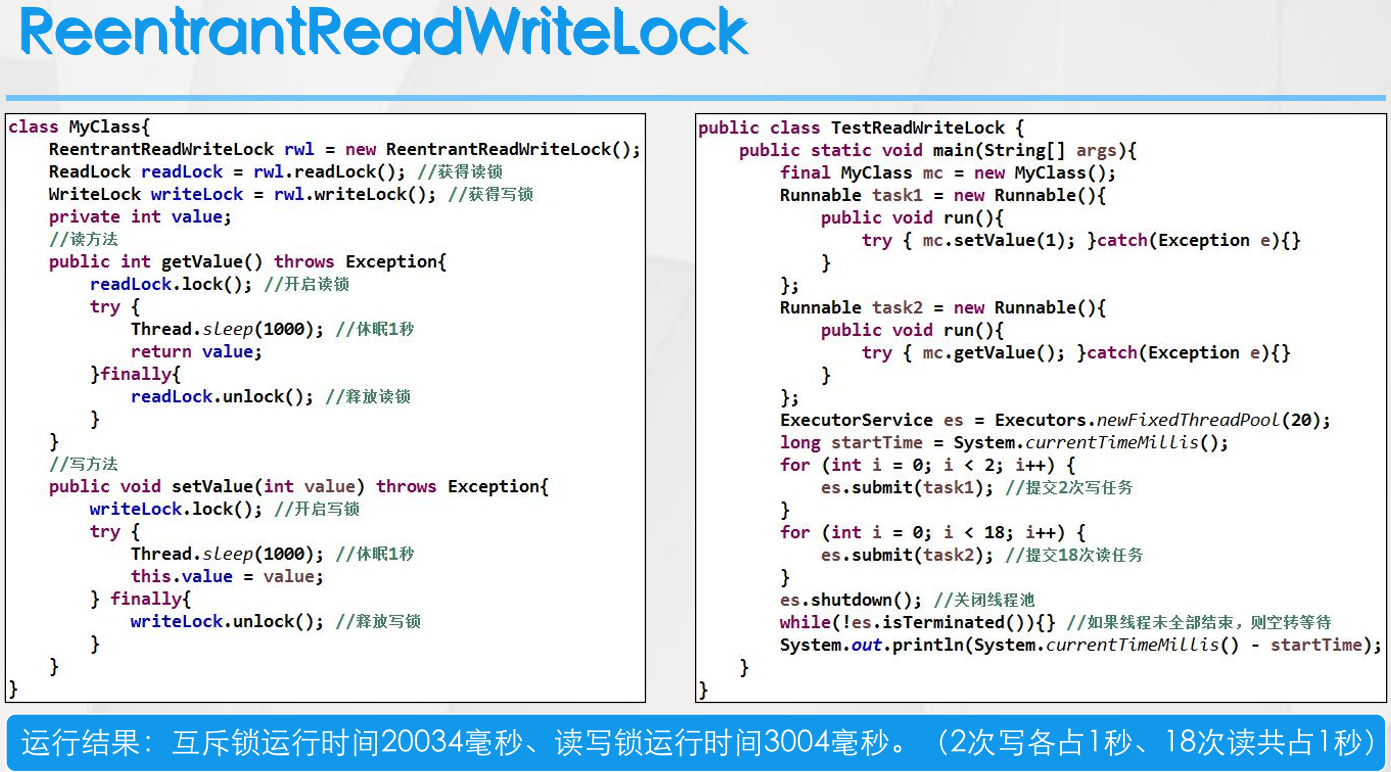
使用Executors获取线程池，

## 7如何统计异步运算结果？

1. /\*\*
2. \* 两个线程各自计算求和操作(1-50,51-100),有主线程汇总
3. \*/
4. **public** **class** TestCallable {
5. **public** **static** **void** main(String[] args) **throws** ExecutionException, InterruptedException {
6. //        线程池接口
7. //        通过这种方式创建的线程池会不会碰到内存上线，200MB内存，一个线程占用的内存接近1MB(在资源占用)，20个线程并发执行可以
8. ExecutorService es = Executors.newFixedThreadPool(2);
9. Callable<Integer> task1 = **new** Callable<Integer>() {
10. @Override
11. **public** Integer call() **throws** Exception {
12. **int** sum = 0;
13. **for** (**int** i = 1; i <= 50; i++) {
14. sum += i;
15. }
16. **return** sum;
17. }
18. };
19. Callable<Integer> task2 = **new** Callable<Integer>() {
20. @Override
21. **public** Integer call() **throws** Exception {
22. **int** sum = 0;
23. **for** (**int** i = 51; i <= 100; i++) {
24. sum += i;
25. }
26. **return** sum;
27. }
28. };
29. //        执行任务
30. Future<Integer> f = es.submit(task1);
31. Future<Integer> f2 = es.submit(task2);
32. //        get是waiting的状态 ,主线程无限期等待
33. Integer result1 = f.get();
34. Integer result2 = f2.get();
35. System.out.println(result1 + result2);
36. }
38. }

## 8.Java中的ReentrantLock与ReentrantReadWriteLock有什么区别？





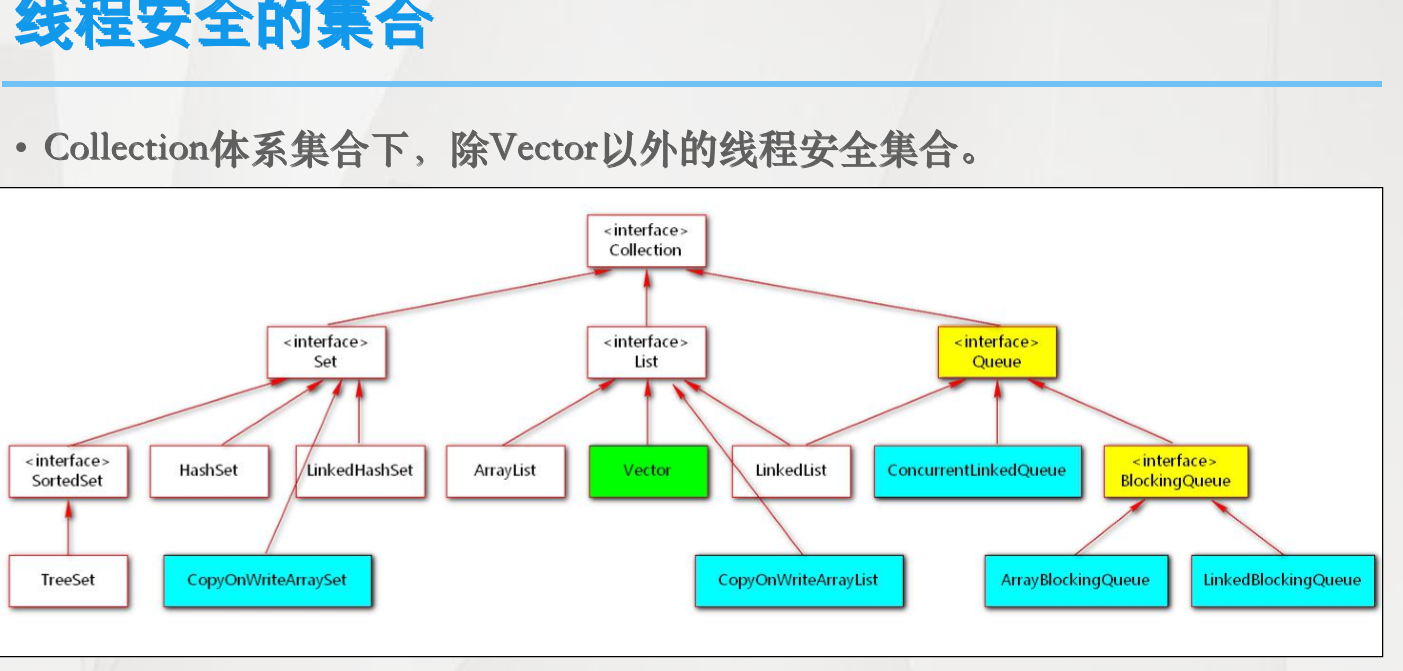
## 9.如何获得一个线程安全的List集合？

Vector、Collections.synchronizedXXX(XXX)、CopyOnWriteArrayList、CopyOnWriteArraySet、

Map:

ConcurrentHashMap、HashTable.

Collections.synchronizedXXX(XXX)





## 10.有哪些线程安全集合的执行效率高？

CopyOnWriteArrayList CopyOnWriteArraySet



## 11.CopyOnWriteArrayList是实现原理是什么？

写有锁，读无锁，读写之间不堵塞，优于读写锁

写入时先copy一个容器副本、再添加新元素、最后替换也能用



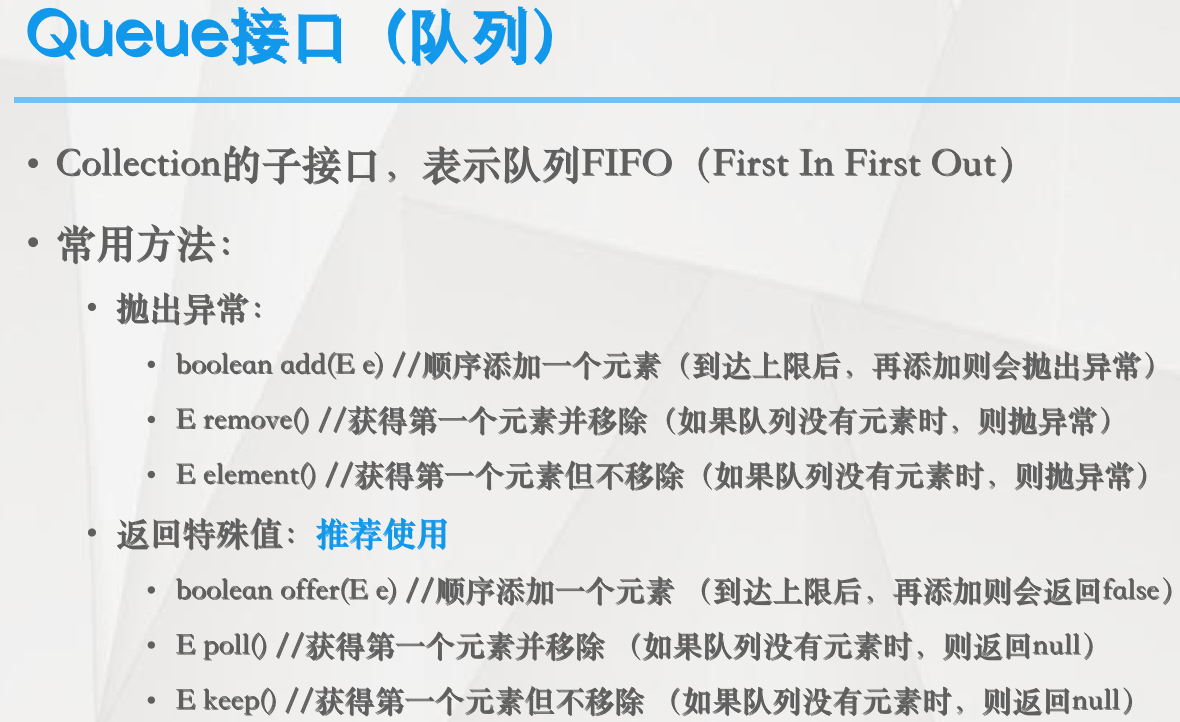
## 12.ConcurrentHashMap的实现原理是什么？



## 13.ConcurrentLinkedQueue的实现原理是什么？



## 14Queue



## 15线程安全的类:

SpringBuffer、Vector、HashTable、

以上类中的公开方法，均为synchronized修饰的同步方法。

1. 线程的状态
2. 什么是线程不安全，加锁的方式有几种。
3. JDK8的新特性

方法引用、Lambda表达式、DateTimeFormatter

4看一下threadLocal的实现原理

没有解决的问题

线程池的种类: