# Java

## 一、java基础

### 1、java有哪几种数据类型？

基本数据类型：byte（1），char（2），short（2），int（4），long（8），double（8），float（4），boolean（1）

引用数据类型：各种类和接口，枚举，数组

### 2、 面向对象和面向过程的区别？

面向对象和面向过程都是一种开发思想。

面向过程就是根据解决问题所需要的步骤，具体化的一步一步的去实现。

面向对象就是把数据及对数据的操作方法放在一起，作为一个整体，也就是对象，若干个这样的整体组成一个系统去解决实际问题。

面向过程只用函数实现，性能比较快因为不需要进行实例化，但是不容易扩展、维护，复用

面向对象通过类去实现功能模块，代码安全性高，容易扩展和复用，比较灵活且便于维护

### 3、JDK/JRE/JVM三者的关系

JVM指的是Java的虚拟机，Java程序运行需要在JVM虚拟机上，不同平台都有自己的JVM虚拟机，所以说java语言实现了跨平台

JRE指的是Java的运行时环境，包含了java核心内库和JVM虚拟机

JDK就是 Java 开发工具包，里面包含了工java运行时需要的工具类和运行时环境，比如javac.exe和其他的一系列编译调试打包工具

### 4、什么是字节码文件

我们使用java编译命令就能将java源文件编译对应成字节码文件（.class），字节码文件是一种八位数据的二进制流文件，可以被JVM快速加载到内存中运行

#### 采用字节码的好处：

Java语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以Java程序运行时比较高效，而且，由于字节码并不专对一种特定的机器，因此，Java程序无须重新编译便可在多种不同的计算机上运行。

Java源代码---->编译器---->jvm可执行的Java字节码(即虚拟指令)---->jvm---->jvm中解释器----->机器可执行的二进制机器码---->程序运行。

### 5、面向对象有哪些特性？

面向对象有三大特性：封装，继承和多态

封装就是将类信息隐藏在类内部，不允许外部程序直接访问，而是通过调用类的方法去访问，良好的封装能减少耦合，增加安全性。

继承就是从已有的类派生出新的类，新的类继承父类的属性和方法，并扩展新的能力，极大提高了代码的复用行和易维护性。

多态是指同一个行为具有多种形式的表现，具体体现在继承，重写，重载，父类引用指向子类对象等，多态体现出面向对象的灵活性。

另外面向对象还有抽象的特性，指把客观事物用代码抽象出来。

### 6、StringBuilder和StringBuffer的区别

它们都继承于abstractStringBuilder，相对于String来说都是可变，StringBuffer添加了syncronized所以是线程安全的，而StringBuilder是线程不安全的

线程安全指，多个线程访问一个对象时，每个线程调用该对象的行为都能获得正确的结果

### 7、为什么静态方法不能调用非静态方法和变量？

类在加载的时候，会将所有被static修饰的内容（包括方法和属性）初始化，但此时非静态方法和变量还没有经过初始化，也就是没有内存空间，调用的时候就会出错。

### 8、异常，error和exception的区别，运行时异常和非运行时异常（checked）的区别？

JAVA异常的层次结构如图：

Throwable类是java所有异常和错误的顶类，有两个重要的子类

Error：处理JVM的内部严重问题，如资源不足，无法恢复

Exception： 可以恢复的异常，分为RuntimeException（unchecked Exception）和其他Exception（checked Exception）

RuntimeException：运行时异常，处理或者不处理都可以（try/catch throw操作）

其他Exception：必须捕获或者声明抛出的异常

### 9、java内存模型JMM

JMM是一个抽象的规范，它定义了共享内存中多线程程序读写规范，即变量如何存储在内存中又如何从内存中读取（这里的变量指实例字段，静态变量，构成数组的元素）

JMM将内存分为主内存和工作内存

主内存是线程共享的公共区域，里面存储了共享变量

工作内存是线程私有的工作区域，里面存储了共享变量的副本

线程通常是对工作内存中的共享变量进行修改，然后同步到主内存，然后其他线程从主内存中同步更新自己的变量，也即，线程之间的交互是通过主内存的。

JMM对Java编程提供了原子性，可见性，有序性的保障

原子性：原子性就是说一个操作是不可分割的，不可中断的，要么执行完成，要么不执行，在JMM中通过lock操作实现，它保证了一个变量在同一时刻只能被一个线程访问，不会被其他线程影响而中断操作，对应关键字Syncronized

可见性：可见性指线程对一个变量的操作对其他线程来说是立即可见的，在JMM中通过在修改变量后变量同步到主内存，其他线程读取时刷新变量值来实现，对应关键字volatile,Syncronized

volatile关键字！

作用：它可以用来修饰成员变量和静态变量（放在主存线程共享的变量），它可以避免线程从自己的工作内存中查找读取变量的值，而是每次都从主内存中去读取，线程操作volatile修饰的变量都是直接操作主存的。

有序性：指本线程的所有操作都是有序的

在多线程中会有指令重排序的问题， 通过volatile关键字解决

volatile的底层原理是内存屏障，Memory Barrier（Memory Fence）

写屏障：对 volatile 变量的写指令后会加入写屏障，

写屏障保证在该屏障之前的，对共享变量的改动，都同步到主存当中

写屏障会确保指令重排序时，不会将写屏障之前的代码排在写屏障之后

读屏障：对 volatile 变量的读指令前会加入读屏障

读屏障保证在该屏障之后，对共享变量的读取，加载的是主存中最新数据

读屏障会确保指令重排序时，不会将读屏障之后的代码排在读屏障之前

### 10、常见的集合

Java中的集合主要由Collection（单列集合）和Map（双列集合）这两个接口派生而来

Collection下有子接口List，Queue,Set（无序）

List接口的实现类有ArrayList，vector（线程安全）,stack，LinkedList（双向链表）

Set接口的实现类有hashset（底层是hashMap），sortedSet（接口）、treeSet（底层是treeMap，遍历有序），LinkedHashSet

线程安全的集合：Vector、HashTable、ConcurrentHashMap、Stack（继承Vector）、ArrayBlockingQueue、ConcurrentLinkedQueue

### 11、重写与重载的区别

1.定义不同---重载是定义相同的方法名，参数不同；重写是子类重写父类的方法。

2.范围不同---重载是在一个类中，重写是子类与父类之间的。

3.多态不同---重载是编译时的多态性，重写是运行时的多态性。

4.返回不同---重载对返回类型没有要求，而重写要求返回类型只能与父类一致或者是父类方法返回类型的子类。

5.参数不同---重载的参数个数、参数类型、参数顺序可以不同，而重写要求参数列表一致

6.修饰不同---重载对访问修饰没有特殊要求，重写访问修饰符的访问权限一定要大于被重写方法的访问修饰符。

### 12、HashMap底层，插入，扩容

前置知识：

二叉树：每个节点至多只有两个子节点的树

搜索二叉树：满足当前节点的左子树上的节点不能大于当前节点，右子树上的节点不能小于当前节点的二叉树

红黑树：一种自平衡的搜索二叉树，能保证遍历，插入，删除的时间复杂度永远是O(logn)

红黑树规则：

红黑：节点只有红黑两种颜色，根节点和叶节点一定是黑色，红节点的子节点一定是黑色

树：叶节点一定是null，任一节点到叶节点的所有路径包含相同数量的黑节点

散列表：又称Hash表，是一种kv存储结构，存储时先计算k经过哈希函数运算后的值，然后根据这个值将v存放在对应的位置

HashMap底层实现是数组+链表/红黑树

添加数据时，根据key的hash值来确定value所在的数组下标

如果数组下标对应的链表/红黑树中有相同的key，就会替换value，否则加入到链表/红黑树中

jdk1.8之前采用的拉链法 数组+链表

jdk1.8之后采用的数组+链表+红黑树法，链表长度大于8时，先尝试扩容数组，如果数组长度此时大于64则会从链表转化为红黑树。

插入操作（简化）：

1、判断数组是否为空，如果为空调用resize方法，初始化一个容量16的数组

2、对key进行hash运算，得到数组中的索引，table下标i=(table.length- 1) & ((h = key.hashCode()) ^ (h >>> 16))，如果索引指向的位置为空，则直接插入一个新的node

3、如果不为空，则判断当前key是否存在，如果存在直接进行替换value操作

4、如果不存在，判断当前Node节点是否是红黑树结构，如果是树类型，则按照树的方式去追加节点，否则在链表尾部插入数据

5、最后判断链表长度是否大于8，如果大于8，会先尝试扩容，判断数组长度是否小于64，是就扩容，否则链表转换为红黑树。

6、插入操作结束后，判断如果hashmap的元素个数超过了负载因子\*容量的阈值，则需要进行扩容操作

7、扩容操作会将原来数组的元素分配到新的数组中，重新每个元素在数组中的下标

8、扩容操作完成后，再将元素插入到新的数组中

扩容操作：

1、在添加元素或初始化时扩容，执行resize方法，第一次初始化长度为16，以后每次在元素达到容量\*负载因子（0.75）时，进行扩容

2、每次扩容操作后，容量都是之前的2倍

3、扩容之后，会创建一个新的数组，然后把老的数组中的元素移动到新的数组中

4、如果是链表，则计算每个链表元素的下标，然后移动到新的数组

5、如果是红黑树，则走红黑树的添加。

### 13、对象的创建方式

1、new

2、将类继承Cloneable接口，然后实现clone方法，调用clone方法即可复制对象

3、反射（类派发），Student.class.newInstance()

4、反射（动态加载），Class.forName("entity.Student").newInstance()

5、反射（构造方法），Student.class.getConstructor().newInstance()

6、Spring容器启动时将Bean注入

### 14、四种访问修饰符

1）public ：公共权限，可以被任意类访问，不同包不同类依然可以访问，

可修饰：类、成员变量、方法，内部类

2）protected：受保护的权限，可以被同包类访问，如果不是同包类，必须是该类的子类才可以访问

可修饰：成员变量、方法、内部类

3）default：默认的(无)，同包权限，只能被同包的类访问

可修饰：类、成员变量、方法，内部类

1. private：私有权限，只能在本类中访问，同包其他类也不能访问

### Java和C++的区别

* 都是面向对象的语言，都支持封装、继承和多态
* Java不提供指针来直接访问内存，程序内存更加安全
* Java的类是单继承的，C++支持多重继承；虽然Java的类不可以多继承，但是接口可以多继承。
* Java有自动内存管理机制，不需要程序员手动释放无用内存

## 二、多线程

### 1、进程与线程的区别

概念：

进程是程序的一次执行结果，是系统运行的基本单位，每个进程都有自己的内存空间和系统资源，进程直接相互独立，但线程之间的运行是可以相互影响的。

线程是比进程更小的单位，一个进程中包含的多个线程，这些多个线程共享进程的内存空间和系统资源，所以线程之间的切换开销比较小。//线程运行在用户态，但是线程切换由操作系统内核完成，需要切换到核心态

协程是一种轻量级的线程，协程在用户态就可以控制，协程的上下文切换更加节省资源。

上下文的切换指的是CPU从当前正在运行的进程或线程中保存状态，然后切换到另一个线程或进程

区别：

进程是正在运行的程序的实例，线程是操作系统运算调度的最小单位，包含在进程当中，一个进程可以有多个线程

进程有自己的内存空间和系统资源，线程之间共享一个进程的内存空间和系统资源

进程之间相互独立，而线程之间是可以相互影响的

线程更加轻量，上下文切换开销比进程低

### 2、并行与并发的区别

并发是同一时间处理多件事的能力，比如多个线程轮流使用一个CPU

并行是同一时间做多件事的能力，比如4核CPU同时执行4个线程

关键区别在于是否同时执行

3、创建线程的方式有哪几种？Runnable与Callable有什么区别？run方法与start方法有什么区别

继承Tread类 ——直接调用start执行线程

实现Runnable接口——先创建MyRunnable对象，再创建Thread对象（将MyRunnable作为构造器参数），再执行start方法

实现Callable接口——先创建MyCallable对象，再创建futureTask对象，再创建Tread对象，再执行start方法

线程池创建（项目中一般使用方式）——对象要实现Runnable，然后创建线程池对象，调用线程池对象的submit方法

Runnable与Callable的区别：

Runnable接口的run方法没有返回值，而Callable接口的call方法有返回值，可以配合FutureTask获取返回结果

Runnable接口的run方法的异常只能在内部消化，而Callable接口的call方法可以将异常抛出

run方法与start方法区别：

start方法是开启一个线程，然后线程去调用run方法，start方法只能被调用一次

run方法封装了线程要执行的方法，可以调用多次

### 4、线程包含了几种状态，状态之间是如何转换的

线程包含了六种状态：

新建（New）：创建线程对象时

可执行（Runnable）：调用start方法后，有执行资格

阻塞（Blocked）：进入可执行状态后，无法获得锁

等待（Waiting）：执行时调用Wait方法进入（wait方法会释放锁）

计时等待（Timed\_Waiting）：执行时调用Time.sleep进入

死亡（Terminated）：运行结束，线程死亡，变成垃圾

### 5、假设有t1,t2,t3三个线程，如何保证他们按顺序执行？

第一种方法：调用join方法：它使当前线程进入time\_waiting状态，直到调用join方法的线程结束，比如t1.join()，就是在t1结束后才会继续运行。

具体：在t2线程的run方法中调用t1.join,在t3线程的run方法中调用t2.join

第二种方法：使用计数器countdownLatch：它需要先设定一个起始值，然后在线程中调用countdown方法让值-1，当值为0时，会解除wait状态

具体：

countDownLatch1.await(); //第一个计数器值为0时结束等待 System.out.println("拿出一瓶牛奶！"); //执行线程方法 countDownLatch2.countDown(); //让控制另一个线程的计数器值为0

### 6、wait和sleep方法有哪些不同？

wait方法是Object类方法，每个对象都有，而sleep方法是Tread类静态方法

wait方法需要在同步块（synchronized）中执行，并且需要先获取锁，而sleep方法没有这个限制

wait方法执行后会释放锁，而sleep方法在拿到锁的情况下执行不会释放锁

### 7、为什么wait方法和notify方法要在synchronized关键字中使用？

从程序层面来说：不这样做会报IllegalMonitorStateException错误

从线程安全层面来说：如果不加同步锁，wait() 方法还没有执行完，notify() / notifyAll() 方法已经执行完，这样 notify() / notifyAll() 就进行了一次空唤醒操作，而 wait() 执行完后由于再没有notify() / notifyAll()的唤醒，会导致wait() 所在线程一直阻塞。

从底层代码层面来说：

首先，每个Java对象底层都会关联一个 Monitor 对象，在 Monitor 中维护了 两个队列 WaitSet 和 EntryList ，owner 属性。

synchronized( lock ) 被调用后会将当前线程 赋值给lock对象所关联的 monitor 对象的 owner 属性，其他线程 再想获取 lock 锁对象的话如果发现 lock对象所关联的 monitor 对象的 owner 属性不为空，就会进入 EntryList 进行阻塞，而调用了wait() 后就会将 owner 指向的线程对象放入 WaitSet 中进行等待，并将 owner 置为 null （释放掉锁），直到其他线程获取到 lock 这个对象锁以后通过 notify() / notifyAll() 方法 唤醒 WaitSet 中 的线程，这时 WaitSet中等待的线程才会进入 EntryList 参与lock 锁的竞争。（ notify() / notifyAll() 并不会释放锁，只有等待 synchronized 执行完才会释放锁 ）

### 8、Synchronized关键字底层原理

synchronized是采用互斥的方式，让同一时刻只能有一个线程能够获取对象锁

java中的Synchronized有偏向锁，轻量级锁，重量级锁三种形式，分别对应了只被一个线程所有，被多个线程交替持有，被多个线程竞争三种情况

在对象锁被多个线程竞争的情况下（重量级锁）：

底层由Monitor实现，Monitor是JVM级别的对象，Monitor分为三个部分，owner，entryset，waitset

当一个线程要获取对象锁时，会先将对象与Monitor关联，然后将owner与当前线程绑定

如果owner已经被其他线程绑定了，当前线程就会加入到entryset中进入阻塞状态

另外，当对象调用wait方法时，当前线程会进入到waitset队列中进入阻塞状态

在对象锁被多个进程无竞争交替持有的情况下（轻量级锁）：

这个时候不需要monitor

当线程尝试获取锁时，会创建一个栈帧，栈帧里包含一个锁记录的结构，主要内容有锁记录地址和对象指针

执行到加锁时，让锁记录中的对象指针指向对象，并尝试使用cas替换锁记录地址和对象头中的mark word，如果标记位为无锁则进行交换，如果非无锁，则有两种情况

1、其他线程已经获取了这个对象的锁，当前线程获取锁失败，线程会尝试自旋获取锁，如果自旋次数超过一定阈值,或者存在多个线程在等待同一个锁,轻量级锁会升级为重量级锁

2、当前线程已经获取了这个对象的锁（锁重入），也就是对象头的mark word指向当前线程的栈帧，此时会在栈帧中再添加一条锁记录，并设置锁记录的锁地址为null，最后在退出synchronized代码块时，如果有锁记录地址为null的锁记录，会清除锁记录，表示重入次数-1

当对象锁被一个线程持有的情况下（偏向锁）：

在上文中可知，轻量级锁在没有竞争时（还是这个线程来获取锁），将会发生锁重入，要执行CAS操作，并在栈桢创建新的锁记录。这样耗费了资源。

优化方式：只有第一次使用CAS将线程ID设置到被锁对象的MarkWord中，以后再来的线程只用验证这个线程ID是自己则没有竞争，无需CAS。

加锁过程：在当前线程的栈帧中创建一个锁记录对象，然后将锁记录的Lock record地址与对象的mark word进行CAS操作，

### 9、synchronized与lock有什么区别？

从语法层面来说：synchronized关键字是在JVM中实现的，通过c++实现

从功能层面来说：

两者都属于悲观锁，都具备基本的互斥，同步，锁重入功能，

而lock支持更多synchronized不支持的功能：如可中断等待（lock.lockInterruptibly()），公平锁，可设置等待超时(tryLcok(time,unit))，可选择性通知（condition）

lock还提供了多种不同场景的实现，如ReentrantReadWriteLcok（读写锁）

从性能层面来说：synchronized提供了偏向锁，轻量级锁，在竞争不激烈的情况下性能很好，而lock在竞争激烈的时候，通常会有更好的表现，因为提供更多的功能。

### 10、CAS你知道吗？

CAS（compireAndSwap）比较并交换，它体现了乐观锁的一种思想，在无锁的情况下保证线程操作数据的原子性，它的内部存在3个操作数

1、变量内存值V

2、旧的预期值A

3、准备设置的新值B

当执行CAS指令时，只有当V=A时，才会去执行B更新V的值，否则不会更新

当多个线程同时使用CAS去操作一个变量时，只有一个线程会执行成功，其他线程均会失败，然后会重新尝试或将线程挂起（阻塞）

另外，CAS是一种系统原语，它的执行一定是连续不被中断的，也就不存在并发问题，这样就保证了原子性

CAS虽然能很高效的解决原子操作，但是仍然存在问题

ABA问题 因为CAS只是判断获取值和在操作时这个值之间的时间该没改变来进行操作，当在这个时间内如果有一个操作修改了这个内存变量的值，由A改为B再改为A，这时CAS会认为这个值从来没有变过，但是值其实已经发生了一次改变

循环时间长时开销大 因为底层是自旋锁，当操作迟迟无法完成的时候，会对CPU带来非常大的开销

只能保证一个共享变量的原子操作 当对多个共享变量进行原子操作时，循环CAS就无法保证操作的原子性

### 11、什么是AQS？

AQS全称abstractQueueSynchronizer，即抽象队列同步器，是一种锁机制，它是作为一个基础框架使用的，像Reentrantlock，countdownlatch都是基于AQS实现的

AQS内部维护了一个先进先出的双向队列，队列中存储了排队的线程

AQS还维护了一个state，表示锁的状态，0为无锁状态，1为有锁状态，如果一个线程将state修改为1，就相当于当前线程获得了资源

对state的修改使用cas操作，保证多线程下的原子性

### 12、ReentrantLock底层原理是什么

ReentrantLcok意为可重入锁，和synchronized一样，当一个线程已经获得锁，再去尝试获得锁时，不会阻塞而是直接获得

ReentrantLock的底层是基于CAS+AQS实现，AQS的底层维护了一个state和一个双向队列，当有线程来抢锁时，会使用cas的方法来对state进行修改，如果修改成功，就将ReentrantLock中的ownerTread属性指向当前线程，如果修改失败，就会插入到双向队列的队尾

ReentrantLock支持公平锁和非公平锁两种实现，如果使用无参构造器，就是非公平锁，也可以传参true设置为公平锁。

### 13、悲观锁、乐观锁和分布式锁的实现和细节

悲观锁：认为线程安全问题一定会发生，所以在操作数据之前先获取锁，保证线程串行执行，例如synchronized，lock

细节：

悲观锁适合插入数据

锁的粒度要尽量小，只锁住需要串行执行的代码

配合事务使用时，要先提交事务再释放锁

乐观锁：认为线程安全问题不一定会发生，因此不加锁，只是在操作数据前判断是否有其他线程对数据做了修改，如果没有被修改则说明线程安全，更新数据，反正说明出现了线程安全问题，可以重试或者返回异常，例如给表加字段version，cas操作

细节：

乐观锁适合更新数据

更新前要先查询version，更新时比较version是否相同

分布式锁：满足分布式系统或集群模式下多进程可见且互斥的锁，常见的实现有redis和zookeeper，redis通常利用setnx方法

细节：

锁的误删，比如线程1拿到锁但是出现了阻塞导致锁自动释放，在线程2拿到锁后执行业务逻辑时，线程1反应过来，继续执行，最后将本已经不属于他的锁误删了

锁的误删解决：设置锁的唯一标识，每个线程在获取锁时，设置锁的value为线程唯一标识（可以用uuid实现），释放锁时判断锁的value是否跟自身线程唯一标识一致，一致才能释放

### 14、什么是死锁，死锁产生的条件？如何避免死锁？

死锁就是一组互相竞争资源的线程，因为互相等待又互不相让资源，导致永久阻塞无法进行下去的情况

死锁产生的条件有四个：

互斥条件：资源x和y只能分别被一个线程占用

占有且等待：线程t1占有资源x后，等待资源y被释放，同时自己不释放资源x

不可抢占：其他线程不能强行抢占线程t1占有的资源

循环等待：线程t1等待线程t2占有的资源，线程t2等待线程t1占有的资源

避免死锁的方法（破坏对应的条件）：

一次性申请所有资源（破坏互斥条件）

占有资源的线程，在申请其他资源时，如果申请失败，可以主动释放自己的资源（破坏占有且等待条件）

按照顺序去申请资源，然后反序释放资源，破坏循环等待条件

### 15、ConcurrentHashMap底层原理

ConcurrentHashMap是在HashMap的数据结构上，增加了CAS操作和Synchronized互斥锁来保证线程安全，并且使用volatile关键字修饰了node中的next和val字段来保证多线程环境下某个线程新增或修改节点对于其他线程是立即可见的。

在进行添加操作时：

计算hash值，定位该元素应该添加到的位置

如果不存在hash冲突，即该位置为null，则使用CAS操作进行添加

如果存在hash冲突，即该位置不为null，则使用synchronized关键字锁住该位置的头节点，然后进行添加操作

### 16、线程池的核心参数和执行原理？

线程池的核心参数有七个：

corePoolSize：核心线程数

maximumPoolSize：最大线程数量，核心线程+救急线程的最大数量

keepAliveTime：救急线程的存活时间，存活时间内没有新任务，该线程资源会释放

unit：救济线程的存活时间的单位

workQueue：工作队列，当没有空闲核心线程时，新来的任务会在此队列排队，当该队列已满时，会创建应急线程来处理该队列的任务

treadFactory：线程工厂，可以定制线程的创建，线程名称，是否是守护线程等

handler：拒绝策略，在线程数量达到最大线程数量时，实行拒绝策略 拒绝策略：

线程池执行原理：

### 17、线程池常见的阻塞队列？

常见的阻塞队列有：

ArrayBlockingQueue：基于数组结构的有界阻塞队列，FIFO

LinkedBlockingQueue：基于链表的有界阻塞队列，FIFO

两者区别（链表与数组的区别）

ArrayBlockingQueue

LinkedBlockingQueue

底层是数组

底层是链表

强制有界

默认无界，传递容量参数后有界

初始化时即创建好node对象数组

插入时才创建node对象

两把锁（头尾）

一把锁

LinkedBlockingQueue的优点是锁分离，很适合生产和消费频率差不多的场景，这样生产和消费互不干涉的执行，能达到不错的效率

### 18、线程池核心线程数如何确定？

没有固定答案，先设定预期，比如我期望的CPU利用率在多少，负载在多少，GC频率多少之类的指标后，再通过测试不断的调整到一个合理的线程数

公式：

分三种情况

高并发，任务处理时间短 ——》CPU核数+1，减少线程上下文切换

并发不高，任务处理时间长 IO密集型任务——》CPU核数\*2+1

读写多，DB操作多，CPU占用少，并且IO数据传输时，是不占用CPU的，所以就可以多释放CPU资源，给其他线程运行，

CPU密集型任务——》CPU核数+1

CPU占用高，如计算任务，视频解码任务，这些任务线程上下文切换开销大，所以要尽量减小开销，提高CPU效率

### 19、线程池的种类有哪些

在JUC的Executor类中，提供了多种创建线程池的方法，主要有四种

1、固定线程数的线程池

核心线程数与最大线程数一样，没有救急线程

阻塞队列是LinkedBlockingQueue，最大容量是Integer.MaxValue

适合任务量已知，相对耗时的任务

2、单例线程的线程池

核心线程数和最大线程数都是1，没有救急线程

阻塞队列是LinkedBlockingQueue，最大容量是Integer.MaxValue

适合按顺序执行的任务，与单线程的区别是，单线程运行完了就会销毁，而线程池创建的线程运行结束不会销毁，而是等待下一个任务，可以重复使用，减少了创建线程和销毁线程的时间，提高资源利用率。

3、可缓存的线程池

核心线程数为0，最大线程数为Integer.MaxValue，救急线程存活时间为1分钟

阻塞队列为SynchronousQueue：不存储元素的阻塞队列，每一个插入操作必须等待一个移除操作

适合任务比较密集，且任务执行时间短的情况，因为使用的是救急线程，在一定时间没有新任务后就会销毁，节省资源，同时能应付任务密集的时间段。

4、可以定时执行的线程池

核心线程数自定义，最大线程数为Integer.MaxValue

适合需要定时执行任务的场景

### 20、线程池使用场景

1、批量将pgsql数据导入ES

计算数据总条数——》固定每页200条，计算总页数N——》将页数N设置为CountdownLatch的count——》创建N个线程批量导入，每次导入完调用countdown方法——》主线程中调用await方法，线程都执行完后，主线程结束

2、异步调用

用户搜索需要保存搜索记录，但保存功能不能影响正常搜索——》搜索时通过线程异步调用保存记录功能

关键注释：@EnableAsync加在SpringBoot启动类上，@Bean将自定义线程池注入到容器中（核心8，最大8）， @Async("线程池名称")加在需要异步调用的方法上，

### 21、对ThreadLocal的理解，ThreadLocal内存泄漏问题

ThreadLocal是java.lang包中的一个类，它实现了线程之间的资源隔离，让每个线程都有自己的独立资源

ThreadLocal的底层实现的关键是它的静态内部类ThreadLocalMap

每个Thread对象都有一个成员变量threadlocals，它指向一个ThreadLocalMap，该Map以ThreadLocal为key，需要存储的线程变量为value

当调用ThreadLocal的set()方法时，会先拿到当前线程对象的ThreadLocalMap，然后以当前ThreadLocal为key将变量存储到map中，调用get方法和remove方法时也是通过这个key去操作。

ThreadLocal内存泄漏问题：

回答这个问题，我们首先要知道java中的四种引用

强引用：强引用是最常见的，只要把一个对象赋值给一个引用变量，那么这个对象就被强引用了，强引用的对象只要不为null，就不会被回收

软引用：软引用相对弱化一些，需要用softReference对象构造方法去创建软引用，当内存充足时，软引用的对象不会被回收，当不足时就会被回收

弱引用：弱引用又更弱了一些，需要用weakReference的构造方法创建弱引用，当发送GC时，只要是弱引用的对象就会被回收

虚引用：虚引用要配合引用队列使用，它的主要作用是跟踪对象垃圾回收的状态，当对象被回收时，通过引用队列做一些通知类工作

在ThreadLocalMap中，Key ThreadLocal是一个弱引用，但是值value是一个强引用，当垃圾回收时，弱引用ThreadLocal会被回收，而value不会，这就导致value成了一个无法被访问也无法回收的变量，造成内存泄漏

解决办法：当使用完ThreadLocal变量后，及时使用remove方法进行清除。

static class Entry extends WeakReference<ThreadLocal<?>> {

/\*\* The value associated with this ThreadLocal. \*/

Object value;

Entry(ThreadLocal<?> k, Object v) {

super(k);

value = v;

}

}