@Async

原理是通过 AOP 实现。当 Spring 容器启动初始化 bean 时，判断类是否使用@Async 注解，并为其创建切入点和切入点处理器，根据切入点创建代理。

当线程调用@Async 注解声明的方法时，会调用代理，执行切入点处理器的 invoke 方法，**将方法的执行提交给线程池中的另外一个线程来处理，从而实现异步执行。**

需要注意一个错误用法，如果 A 方法调用同类的@Async 注解声明的 B 方法，并不会异步执行。因为从 A 方法进入调用的都是该类对象实例本身的 B 方法，不会进入代理类的 B 方法。

测试方法：在调用类和被调用类增加 logger.info(Thread.currentThread().getName() + "serviceName"); 判断两者是否是相同线程。

在方法上使用该注解，声明该方法是一个异步任务。当其他线程调用这个方法时就会开启一个新的子线程去异步处理该业务逻辑。

在类上使用，声明该类的所有方法都是异步任务。

使用该注解的方法的类对象，必须是 Spring 管理下的 bean 对象，比如在类上声明@Component。

需要在主类上开启异步配置**@EnableAsync**

默认使用 Spring 的默认线程池 **SimpleAsyncTaskExecutor**

核心线程数:8,

最大线程数：Integer.MAX_VALUE,

任务队列长度：Integer.MAX_VALUE，

任务队列：LinkedBlockingQueue,

空闲线程保留时间：60s,

线程池拒绝策略：AbortPolicy

问题：并发情况下会无限创建线程。有内存溢出的风险。

解决：自定义配置参数

Sping.task.execution.pool.max-size: 6

问题：因为@Async 是异步执行，在其进行数据库操作时无法控制事务管理。

解决：把@Transactional 注解放到内部的需要进行事务的方法上。

问题：如果在工程中共用线程池，可能导致因为一个业务占用了所有线程，导致其他业务的任务被 abort。

解决：定义多个线程池。线程池隔离。

@Async(value = “myThreadpool”)

@Bean(“myThreadpool”)

Public Executor myThreadpool() {

}

@Async 和 CompletableFuture 的使用场景

@Async

当业务流程有明显的主流程和次流程，次流程执行的结果不影响主流程。为了保证主流程的安全，或者提高主流程的性能，或者不影响主流程等等。

可以在 Service A 完成主流程，在 Service B 完成主流程。在 Service A 调用 Service B。在 Service B 声明@Async。

CompletableFuture

在一个**循环**中的操作，使用 CompletableFuture 开启多个线程，多个线程之间的操作结果互不影响，这样提高性能。

[了解 CompletableFuture_云淡风轻~~的博客-CSDN 博客]

1.如果业务中每个线程需要在线程结束后立即进行处理，而非等待其他线程都结束再进行操作。

2.如果任务 1 和任务 2 并行执行，任何一个任务失败/成功，则标志整个业务失败/成功。

3.如果每个线程中有较为复杂的异步任务编排（组合多个 Future 并行执行结果），则建议使用 CompletableFuture，因为可以简化代码，同时优雅。

CompletableFuture 的优点就在于 ：

①有回调函数，可以在执行完成后自动调用回调处理逻辑;

②其提供的四十多个 API 可以优雅的编排任务链（提供异步任务完成后的链式调用）;

③具备异常处理机制

误区 1：

CompletableFuture 所有的方法都有额外以 Async 结尾的方法。此类方法的作用在于不是说后续任务可以在前序任务还没执行完就可以运行。举例:

thenRun(Runnable action) 当上一个任务结束后，此任务沿用上一个任务的线程池。

thenRunAsync(Runnable action) 当上一个任务结束后，此任务使用默认的 ForkJoinPool 线程池

thenRunAsync(Runnable action，Executor executor)，当上一个任务结束后，此任务使用自定义线程池（推荐）

误区 2

因为默认使用 forkJoinPool，可能出现因为线程池被其他任务占用且耗尽，导致本任务的线程长时间等待状态并阻塞。

关于 Spring 中的线程池(执行器)

Spring 用 TaskExecutor 和 TaskScheduler 接口提供了异步执行和调度任务的抽象。

Spring 的 TaskExecutor 和 java.util.concurrent.Executor 接口时一样的，这个接口只有一个方法 execute(Runnable task)。

Spring 已经内置了许多 TaskExecutor 的实现，没有必要自己去实现:

SimpleAsyncTaskExecutor：

这种实现不会重用任何线程，每次调用都会创建一个新的线程。

SyncTaskExecutor：

这种实现不会异步的执行，相反，每次调用都在发起调用的线程中执行。它的主要用处是在不需要多线程的时候，比如简单的测试用例；

ConcurrentTaskExecutor：

这个实现是对 Java 5 java.util.concurrent.Executor 类的包装。有另一个 ThreadPoolTaskExecutor 类更为好用，它暴露了 Executor 的配置参数作为 bean 属性。

SimpleThreadPoolTaskExecutor：

这个实现实际上是 Quartz 的 SimpleThreadPool 类的子类，它会监听 Spring 的生命周期回调。当你有线程池，需要在 Quartz 和非 Quartz 组件中共用时，这是它的典型用处。

ThreadPoolTaskExecutor：

这是最常用、最通用的一种实现。它包含了 java.util.concurrent.ThreadPoolExecutor 的属性，并且用 TaskExecutor 进行包装。

CompletableFuture 工具类

Java8 提供的工具类，实现了 Future 接口和 CompletionStage 接口。拥有 40 多种方法，为函数式编程中的流式调用而准备。

和 Future 一样，可以作为函数调用的契约。调用线程调用 CompletableFuture.supplyAsync() 创建一个新线程，请求线程可以先处理其他任务。如果其他任务完成后调用 CompletableFuture.get()方法获取结果。如果数据还没有准备好，就会等待。

CompletableFuture 的用法

简单用法 get() 与 complete()

提交任务 runAsync() 与 supplyAsync()

链式处理 theRun()、thenAccept() 和 thenApply()

组合处理 thenCompose() 与 thenCombine() 、**allOf** 与 anyOf()

误区

没有设置超时时间

CompletableFuture.get(3, TimeUnit.SECONDS)

没有使用自定义线程池

没有异常处理

进程

简单地说，在 windows 中看到的后缀为 exe 的文件都是程序。程序是静态地，只是文件。

当双击运行程序时，这个 exe 文件就会被操作系统加载，得到一个进程。进程是动态地。

操作系统会给进程分配资源，包括 CPU、内存、IO 等等。

进程中可以容纳多个线程。进程就像是一个容器。

线程之间会竞争资源，也有相互合作。

使用多线程而不是多进程进行并发程序的设计, 是因为线程间的切换和调度的成本远远低于进程。

线程

线程的生命周期

NEW 刚刚创建，还没有执行

RUNNABLE 调用 start() 方法执行，处于 RUNNABLE 状态，表示线程所需资源都已经准备好

BLOCKED 执行过程中遇到 synchronize 同步块，就会进入 BLOCKED 阻塞状态。线程会暂停执行，知道获得请求的锁。

WAITING 无时间限制的等待状态。线程在等待一些特殊事件，比如通过 wait()方法等待的线程在等待 notify()方法，而通过 join()方法等待的线程则会等待目标线程的终止。一旦等到期望的事件，线程就会再次执行，进入 RUNNABLE 状态。

TIMED_WAITING 有时限的等待状态。

TERMINATED 线程执行完毕，进入 TERMINATED 状态，表示结束。

终止线程 stop

正常情况下，线程执行完毕就会结束，无需手工关闭。

特殊情况下，比如常驻系统的后台线程，可能是一个无限循环，通常不会自动结束。

在线程 A 调用线程 **B.stop**()方法关闭线程 B，比较暴力，可能会引起数据不一致的问题。

更好的方法是在线程 B 内部增加退出的条件判断，满足条件后,正常结束线程 B。

线程中断 interrupt

重要的线程协作机制。

在线程 A 中调用目标线程 **B.interrupt()**，给目标线程 B 发一个通知，通知目标线程 B 有人希望你退出。**至于目标线程接到通知后如何处理，完全由目标线程决定。**

Public void Thread.interrupt() // 中断线程，即发中断通知

Public Boolean Thread.isInterrupted() // 判断是否被中断，即是否收到中断通知

Public static Boolean Thread.interrupted() // 判断是否被中断，并清除当前中断状态

等待 wait 和通知 notify

线程协作机制。这两个方法属于 Object 类，意味着任何对象都可以调用这两个方法。

当线程 A 中，调用了 **object.wait()**方法，线程 A 就会暂停执行，转为等待状态。线程 A 会一直等到其他线程调用了 **object.notify()**方法为止。

通常 object 定义为 final static Object object = new Object();

这样，object 对象成为了多个线程之间通信的有效手段。

原理：当线程 A 调用 object.wait()方法，该线程会进入 object 对象的等待队列。这个队列可能有多个线程，因为系统可能运行了多个线程同时等待一个对象。

当 object.notify()被调用时，就会从等待队列随机选择一个线程并将其唤醒。另外 object.notifyAll()被调用时，会唤醒等待队列中所有线程。

注意，object.wait()必须包含在 Synchronized 语句块中。无论 wait()还是 notify()都需要先获取目标对象的监视器。

object.wait()和 **Thread.sleep()**都可以让线程等待，区别一是 object.wait()可以被唤醒。区别二是 object.wait()会释放目标对象的锁，而 Thread.sleep()不会释放任何资源。

挂起 suspend 和继续执行 resume

两个废弃方法。因为 suspend 在导致线程暂停的同时不会释放任何锁资源。此时，任何线程想要访问被其占用的锁时，都会被牵连，导致无法正常运行。

用法：在线程 B 中 **Thread.currentThread().suspend(); 挂起自己**。在线程 A 中运行线程 B，并调用线程 **B.resume()**方法，但不一定生效。

等待线程结束 join

在线程 A 中调用线程 **B.Join()**方法，表示线程 A 会无限等待，并阻塞当前线程 A，直到线程 B 结束。

出让 yield

Thread.yield()方法会使当前线程让出 CPU。让出后还会进行 CPU 资源的争夺。因此，调用 Thread.yield()表示当前线程已经完成了最重要的工作，可以休息一会儿，给其他线程一些工作机会。

Volatile（易变的，不稳定的）和 JMM

Java 内存模型围绕着原子性，有序性和可见性展开。Java 使用一些特殊关键字或者操作来告诉虚拟机，在这个地方需要特别注意，不要随意变动优化目标指令。

Volatile 对于保证操作的原子性有非常大的帮助。**但是不能代替锁，它也无法保证一些复合操作的原子性。**

Volatile 可以保证数据的可见性和有序性。确保一个线程修改了数据后，其他线程能够看到这个改动。

注意：Volatile 并不能保证线程安全。当两个线程同时修改某一个数据时，依然会产生冲突。

用法：public volatile static int = 0;

线程安全和 synchronized

关键字 synchronized 的作用是实现线程间的同步。通过对同步区的代码加锁，使得每次只有一个线程进入同步块，从而保证线程间的安全性。

问题：对性能消耗大。

用法

指定加锁对象：给**指定对象**加锁，进入同步区代码前获得给定对象的锁。

Static Account instance = new Account();

Synchronized(instance) {

}

直接作用于实例方法：相当于对**当前实例**加锁，进入同步区代码前获得给定当前实例的锁。

Static Account instance = new Account();

Public synchronized void increase() {

}

Thread thread1 = new Thread(instance);

Thread thread2 = new Thread(instance);

直接作用于静态方法：相当于对**当前类**加锁，进入同步区代码前获得当前类的锁。

Public **static** synchronized void increase() {

}

Thread thread1 = new Thread(new Account());

Thread thread2 = new Thread(new Account());

常见的并发错误

在线程中操作线程不安全的数据类型，比如 ArrayList 和 HashMap 等。

Static Map<String, String> map = new HashMap< String, String >();

解决方案，改用 ConcurrentHashMap。

使用 Integer 类型对象加锁

Public static Integer I = 0;

Synchonozed(i) {

i++;

}

分析：因为 i++相当于 i=Integer.valueOf(i.intValue() + 1);

而 Integer.valueOf()实际上会每次返回 new Integer();

所以每次加锁可能都加在不同的对象实例上。

线程组 ThreadGroup

驻守后台：守护线程 Daemon

比如垃圾回收线程，JIT 线程等等。

用法：

t.setDaemon(true);

t.start();

线程的几种场景

有几个任务同时进行，只有所有的任务都完成，线程才完成。

有几个任务同时进行，只要其中一个任务完成，线程就完成。

一个线程的输入依赖另一个线程的输出

两个线程如何共享数据

Threadlocal

ThreadLocal 详解_似寒若暖的博客-CSDN 博客_threadlocal

Thread 的局部变量，**线程局部变量** ThreadlocalValue。不同线程之间互不干扰。在线程的生命周期内起作用。

在并发场景中，成员变量是线程不安全的，因为多个线程在同时操作同一个变量。

ThreadLocal 使得每个线程都拥有自己独立的变量，竞态条件被彻底消除了，在并发模式下是绝对安全的变量。

解决多线程间共享变量线程安全问题的大杀器——ThreadLocal_threadlocal 怎么保证线程安全_YHJ 的博客-CSDN 博客

在 Thread 类中定义了类型为 ThreadLocalMap 的变量 ThreadLocals，是一个定制化的 HashMap，初始值为 null。

在 ThreadLocal 类中定义了内部静态类 ThreadLocalMap，定义了 get(), set(T), getMap()对其操作。

getMap(Thread t)

以当前线程作为参数和 key，返回当前线程的变量 ThreadLocals。

set()

获取当前线程，以当前线程

父子线程

深入解析父子线程_blpluto 的博客-CSDN 博客_父子线程

第一种是父线程是进程的主线程，子线程由主线程创建；

只需要主线程的进入点函数返回，就能够终止整个进程的运行。请注意，进程中运行的任何其他线程都随着进程而一起终止运行。

还有一种特殊的情况，如果子进程内发生死锁，那么这个子进程就无法退出，也会导致整个进程都无法退出。这种就是为什么有时候我们的程序已经退出（至少界面已经关闭），但任务管理器中却还有这个应用程序的进程存在的原因。

第二种情况是父线程为进程主线程创建的一个子线程，而这个子线程又创建了一个孙线程，这种情况大多被称为子孙线程。

假设主线程为 A，创建了线程 B，然后 B 又创建了线程 C，现在，如果线程 B 终止了，那么线程 C 会不会也终止呢?

如果我们创建的 C 线程没有使用 B 线程的任何资源，也就是说，B 线程创建的 C 线程，在创建之后，这两者就是相互独立的。**所以这种情况下，如果 B 线程终止，那么 C 线程在线程函数没有返回的时候，是不会结束的。**

线程之间共享数据

实现了 BlockingQueue 接口的 ArrayBlockingQueue 类和 LinkedBlockingQueue。

前者适合做有界队列，后者适合做无界队列。

服务线程需要从队列取数据或者存数据，当队列为空或者满时，BlockingQueue 会让服务线程等待，当有新消息进入队列（或者有消息从满队列移出后），自动将服务线程唤醒。

线程池

CompletableFuture 和@Async 配置自定义线程池_wangfenglei123456 的博客-CSDN 博客

单例和多线程

单例不是线程安全的。

在并发场景，多个线程对单例对象的非静态成员变量的写操作会存在线程安全问题。

有两种常见的解决方案：

1. 在 bean 对象中尽量避免定义可变的成员变量（不太现实）。
2. 在类中定义一个 ThreadLocal 成员变量，将需要的可变成员变量保存在 ThreadLocal 中（推荐的一种方式）。

面试官：Controller 是单例还是多例？_androidstarjack 的博客-CSDN 博客

Spring 中的 Controller ， Service， Dao 是不是线程安全的？ (qq.com)

Spring 默认是单例多线程模式

服务启动的时候，会启动一个主线程，main 方法是入口。

当请求进来时，会为每个请求新增一个子线程。在并发请求场景，就可能会有线程安全问题。

Spring 单例模式下的多线程访问_多个请求同一个接口是多线程吗_小白写程序的博客-CSDN 博客

当请求进来时，怎么线程安全地获取请求头？

SpringBoot--Controller 获取 HttpServletRequest_controller httpservletrequest_IT 利刃出鞘的博客-CSDN 博客