**ThreadPoolExecutor**

一、 合理的使用线程池的好处

1.降低资源消耗：通过重用已经创建的线程来降低线程创建和销毁的消耗

2.提高响应速度：任务到达时不需要等待线程创建就可以立即执行。

3.提高线程的可管理性：线程池可以统一管理、分配、调优和监控。

二、ThreadPoolExecutor实现原理

step1.调用ThreadPoolExecutor的execute提交线程，首先检查CorePool，如果CorePool内的线程小于CorePoolSize，新创建线程执行任务。

step2.如果当前CorePool内的线程大于等于CorePoolSize，那么将线程加入到BlockingQueue。

step3.如果不能加入BlockingQueue，在小于MaxPoolSize的情况下创建线程执行任务。

step4.如果线程数大于等于MaxPoolSize，那么执行拒绝策略。

三、ThreadPoolExecutor构建参数

corePoolSize 核心线程池大小

maximumPoolSize 线程池最大容量大小

keepAliveTime 线程池空闲时，线程存活的时间

TimeUnit 时间单位

ThreadFactory 线程工厂

BlockingQueue任务队列

RejectedExecutionHandler 线程拒绝策略

四、关键知识点

1.maximumPoolSize和corePoolSize的区别：这个概念很重要，maximumPoolSize为线程池最大容量，也就是说线程池最多能起多少Worker。corePoolSize是核心线程池的大小，当corePoolSize满了时，同时workQueue full（ArrayBolckQueue是可能满的） 那么此时允许新建Worker去处理workQueue中的Task，但是不能超过maximumPoolSize。超过corePoolSize之外的线程会在空闲超时后终止。

2.任务执行顺序

1、当线程数小于 corePoolSize时，创建线程执行任务。

2、当线程数大于等于 corePoolSize并且 workQueue 没有满时，放入workQueue中

3、线程数大于等于 corePoolSize并且当 workQueue 满时，新任务新建线程运行，线程总数要小于 maximumPoolSize

4、当线程总数等于 maximumPoolSize 并且 workQueue 满了的时候执行 handler 的 rejectedExecution。也就是拒绝策略。

心得：

在批处理中可以适当引入多线程处理来加速整体处理效率

Spring AOP

一、关键概念

切入点（Pointcut）  
在哪些类，哪些方法上切入（where）

通知（Advice）  
在方法执行的什么实际（when:方法前/方法后/方法前后）做什么（what:增强的功能）

切面（Aspect）  
切面 = 切入点 + 通知，通俗点就是：在什么时机，什么地方，做什么增强！

织入（Weaving）  
把切面加入到对象，并创建出代理对象的过程。（由 Spring 来完成）

二、可创建的切面类型

| 注解 | 说明 |
| --- | --- |
| @Before | 前置通知，在连接点方法前调用 |
| @Around | 环绕通知，它将覆盖原有方法，但是允许你通过反射调用原有方法，后面会讲 |
| @After | 后置通知，在连接点方法后调用 |
| @AfterReturning | 返回通知，在连接点方法执行并正常返回后调用，要求连接点方法在执行过程中没有发生异常 |
| @AfterThrowing | 异常通知，当连接点方法异常时调用 |

三、切点指示符

| 切点指示符 | 说明 | 样例 |
| --- | --- | --- |
| execution | 用来匹配方法签名，方法签名使用全限定名，包括访问修饰符（public/private/protected）、返回类型，包名、类名、方法名、参数，其中返回类型，包名，类名，方法，参数是必须的 | 1.全限定  @Pointcut("execution(public String org.baeldung.dao.FooDao.findById(Long))")  2.通配符  @Pointcut("execution(\* org.baeldung.dao.FooDao.\*(..))”)  3.全限定参数  @Pointcut("execution(\* \*..find\*(Long))")  4.参数通配符  @Pointcut("execution(\* \*..find\*(Long,..))") |
| within  @within | 用来限定连接点属于某个确定类型的类 | 1.@Pointcut("within(org.baeldung.dao.FooDao)")  2.@Pointcut(“within(org.baeldung..\*)")  1.@Pointcut("@within(org.springframework.stereotype.Repository)")等效于@Pointcut("within(@org.springframework.stereotype.Repository \*)") |
| this | 用来匹配的连接点所属的对象引用是某个特定类型的实现 | @Pointcut("this(org.baeldung.service.FooServiceImpl)") |
| target | 用来匹配的连接点所属目标对象为interface | @Pointcut("target(org.baeldung.service.FooService)") |
| @Target | 匹配连接点所属的目标对象的类有一个指定的注解 | @Pointcut("@target(org.springframework.stereotype.Repository)") |
| @args | 匹配连接点的参数的，@args指出连接点在运行时传过来的参数的类必须要有指定的注解 | @Pointcut("@args(org.baeldung.aop.annotations.Entity)")  public void methodsAcceptingEntities() {}  @Before("methodsAcceptingEntities()")  public void logMethodAcceptionEntityAnnotatedBean(JoinPoint jp) {  logger.info("Accepting beans with @Entity annotation: " + jp.getArgs()[0]);  } |
| @annotation | 匹配那些有指定注解的连接点 | 假设有@Loggable  @Pointcut("@annotation(org.baeldung.aop.annotations.Loggable)") |

四、切点表达式组合

可以使用&&、||、!、三种运算符来组合切点表达式，表示与或非的关系。

@Pointcut("@target(org.springframework.stereotype.Repository)")

public void repositoryMethods() {}

@Pointcut("execution(\* \*..create\*(Long,..))")

public void firstLongParamMethods() {}

@Pointcut("repositoryMethods() && firstLongParamMethods()")

public void entityCreationMethods() {}

心得：

AOP能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。