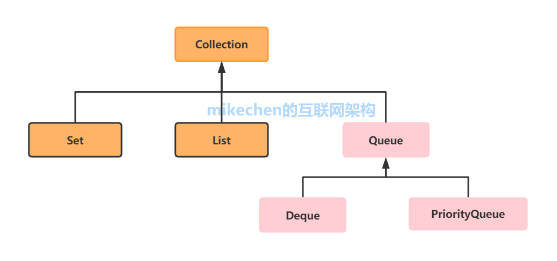
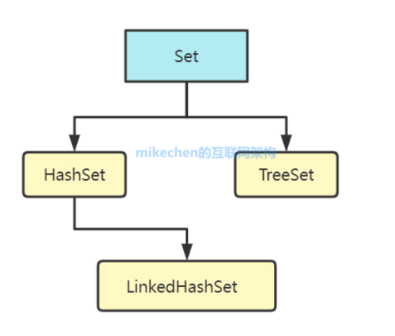
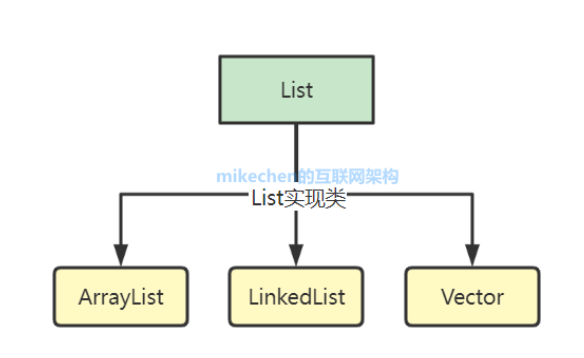
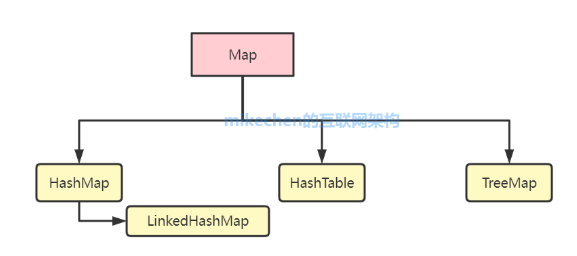
1java集合由Collection和Map派生

Collection派生 List, Set,Queeu

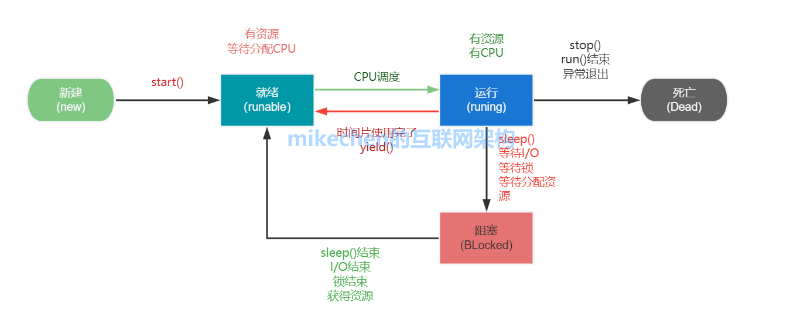


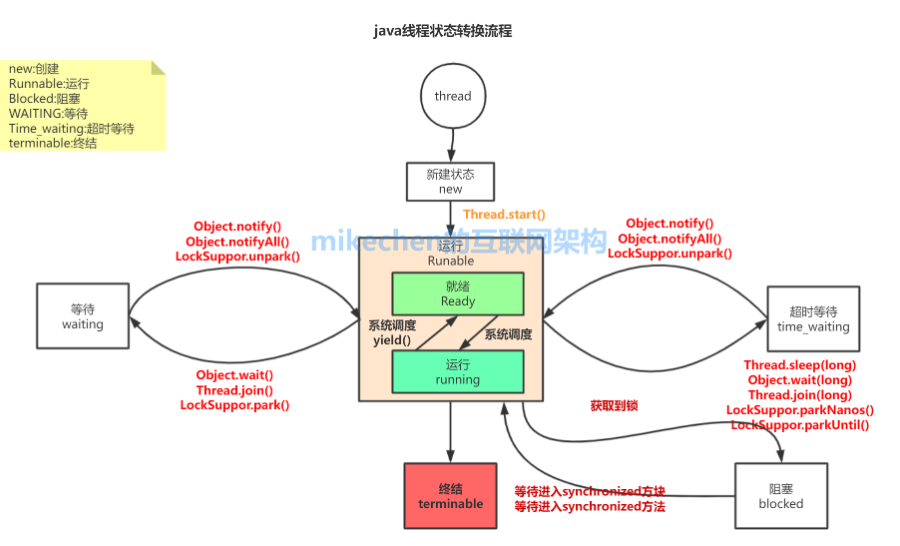
Map派生 key-value集合





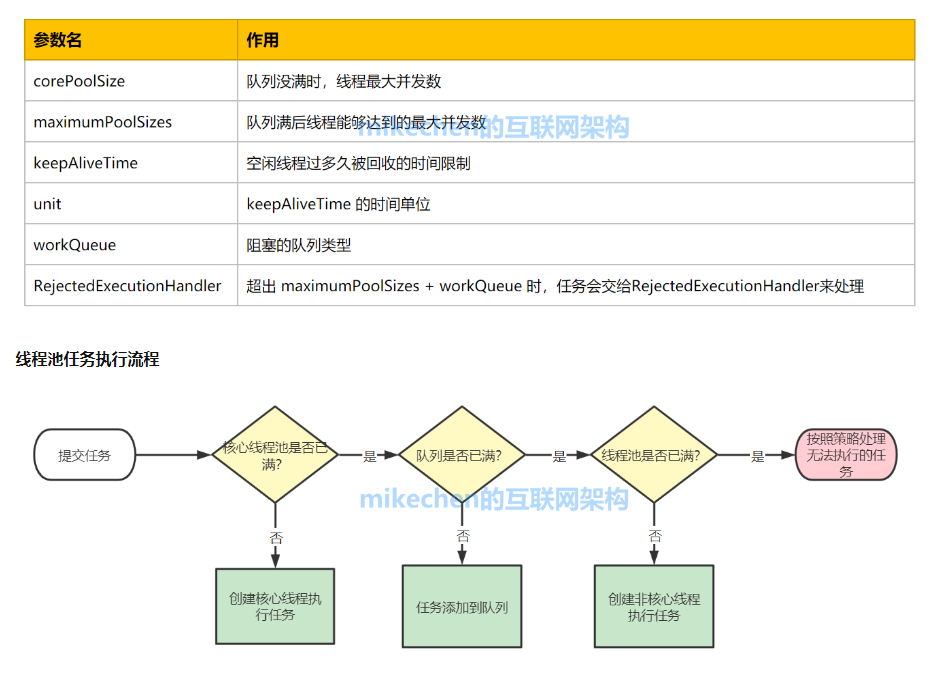
线程生命周期





多线程创建：1继承Thread类2 实现Runnable接口

3线程池创建



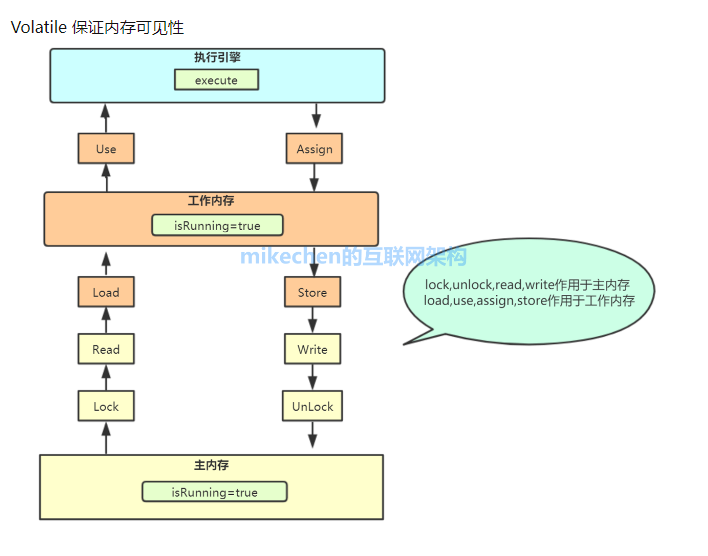
4匿名内部类 5Lambda创建

同步synchronized 存储在对像头里的Mark Word,主要通过锁对象monitorenter 和monitorexit来实现

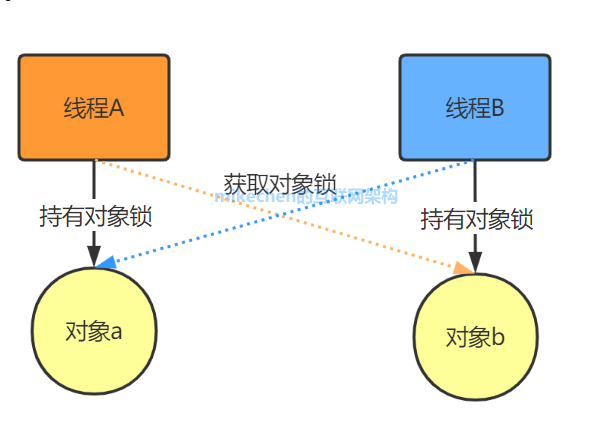
线程安全问题

1原子性 通过synchornized ReentrantLock来保证

2 可见性，有序性 通过volatile 修改共享变量，保证可和禁止指令重排



核心概念：内存屏障，

线程死琐

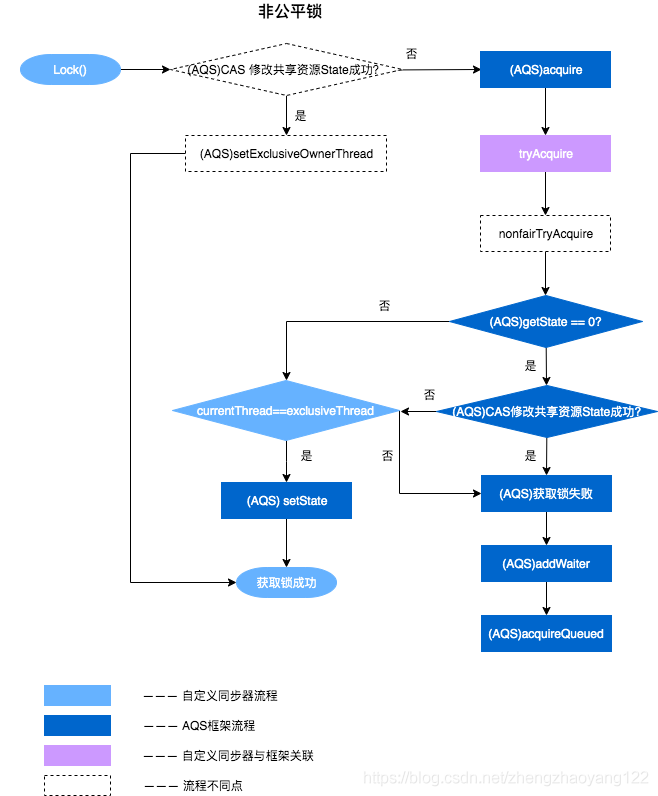
多线程也不一定快，因为有线程创建和线程上下文切换的开销

多线程琐类型

1 synchronized 同步琐

2ReentrantLock 重入琐





3 Semaphore

4AtomicInteger cas乐观琐

java9以及以后的版本中，为了模块化系统的顺利实施，模块下的类加载器主要有几个变动：

1.扩展类加载器（Extension Class Loader）被平台类加载器（Platform ClassLoader）取代（java9中整个JDK都是基于了模块化的构建，原来的rt.jar和tools.jar都被拆分了数十个JMOD文件）。因为java类库可以满足扩展的需求并且能随时组合构建出程序运行的jre,所以取消了JAVA\_HOME\lib\ext和JAVA\_HOME\jre目录

1. 平台类加载器和应用程序类加载器都不在派生自java.net.URLClassLoader。现在启动类加载器、平台类加载器、应用程序类加载器全都继承于jdk.internal.loader.BuiltinClassLoader,在BuiltinClassLoader中实现了新的模块化架构下类如何从模块中加载的逻辑，以及模块中资源可访问性的处理。

sprinboot自动装配主要靠EnableAutoConfiguration引入AutoConfigurationImportSelector

Configuration 和AutoConfiguration的区别

@Configuration –Application的用户，直接代码进行配置的。

AutoConfiguration 是给 Springboot 插件（xxxx.xxx.starter）使用用的。

2.2 加载的方式

@Configuration加载是由@ComponentScan指定的package，未指定 以ApplicationClass 所属package开始。

AutoConfiguration 是通过classpath\*:META-INF/spring.factories来被发现。 通过 key org.springframework.boot.autoconfigure.EnableAutoConfiguration. AutoConfiguration 是由 import selector 的方式加载的

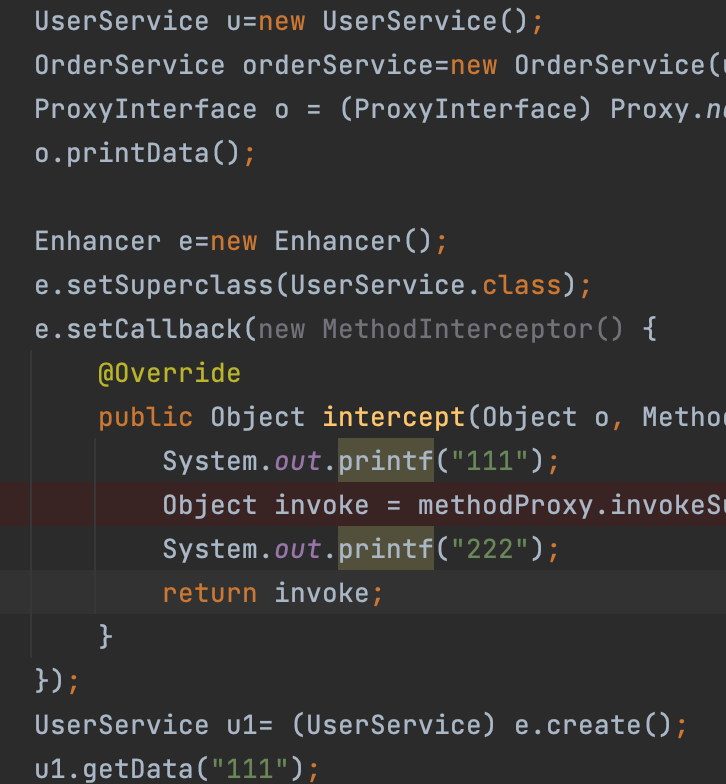
@Configuration 先于AutoConfiguration加载

## new String("a") + new String("b")生成几个对象

finishBeanFactoryInitialization

cglib :通过ASM字节生成码框架代理生成一个代理对象的子类，并继承它所有的非PRIVATE,FINAL方法，效率比javar反射高

java动态代理反射类Proxy以及InvocationHandler回调接口实现的，只能对实现了接口的类生成代理



[浅拷贝](https://so.csdn.net/so/search?q=%E6%B5%85%E6%8B%B7%E8%B4%9D&spm=1001.2101.3001.7020)：对基本类型数据进行值传递，对引用类型数据进行引用传递的拷贝

深拷贝：对基本类型数据进行值传递，对引用类型数据复制一个新的对象，并复制其内容

核心在于实现Cloneable 一般只调用super.clone();也可以实现序列化，但效率不高

String 操作类String StringBuilder(字符+类) StringBuffer(线程安全)

实例化对象的方法：new 反射newInstance clone 反序列化

红黑树特性(等价于4阶B树)：

节点是红色或黑色

根是黑色

叶子节点（外部节点，空节点）都是黑色，这里的叶子节点指的是最底层的空节点（外部节点），下图中的那些null节点才是叶子节点，null节点的父节点在红黑树里不将其看作叶子节点

红色节点的子节点都是黑色

红色节点的父节点都是黑色

从根节点到叶子节点的所有路径上不能有 2 个连续的红色节点

从任一节点到叶子节点的所有路径都包含相同数目的黑色节点

## 算法复杂度：把输入规模看成x轴，所花时间/空间看成y轴

O（n）就是y=x，y随x的增长而线性增长。也就是成正比，一条斜线。

O（1）就是y=1，是一个常量，不管x怎么变，y不变，一条与x轴平行的线。

## 45、哪些集合类是线程安全的

Vector：就比Arraylist多了个同步化机制（线程安全）。  
Stack：栈，也是线程安全的，继承于Vector。  
Hashtable：就比Hashmap多了个线程安全。  
ConcurrentHashMap:是一种高效但是线程安全的集合。

run 和start的区别：用 start 方法方可启动线程并使线程进入就绪状态，而 run 方法只是 thread 的一个普通方法调用，还是在主线程里执行。

monitor监视琐包含：

WaitSet 中保存了因 wait() 而被阻塞的线程，对应阻塞状态

EntryList 中保存了暂时没有分到时间片的线程，对应就绪状态

Owner 中保存了当前持有锁的线程

线程池核心类ExecutorService，主要参数为int corePoolSize,  
 int maximumPoolSize,  
 long keepAliveTime,  
 TimeUnit unit,  
 BlockingQueue<Runnable> workQueue

synchronized 锁升级原理：在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级

BEAN生命周期：1使用BeanDefinition实例化

2填充属性

3调用初始化

4使用Bean

5销毁

## Spring Boot 中的监视器

JVM包括类加载子系统、堆、方法区、栈、本地方法栈、程序计数器、直接内存、垃圾回收器、执行引擎。

（1）虚拟机栈的栈帧

每个方法在执行的时候，JVM都会创建一个相应的栈帧（操作数栈、局部变量表、运行时常量池的引用），当方法执行完，该栈帧就从栈中弹出，这样一来，方法中临时创建的独享就不存在了，或者说没有任何GC roots指向这些临时对象，这些对象在下一次GC的时候便会被回收。

（2）方法区中的静态属性

静态属性数据类属性，不属于任何实例，因此该属性自然会作为GC roots。这要这个class在，该引用指向的对象就一直存在，class也由被回收的时候。

class何时会被回收？

堆中不存在该类的任何实例

加载该类的classLoader已经被回收

该类的java.lang.class对象没有在任何地方被引用，也就是说无法通过反射访问该类的信息

1. 本地方法栈引用的对象

@ PointCut标签用法

* execution：用于匹配方法执行的连接点
* within：用于匹配指定类型内的方法执行
* this：用于匹配当前AOP代理对象类型的执行方法；注意是AOP代理对象的类型匹配，这样就可能包括引入接口也\* 类型匹配
* target：用于匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配
* args：用于匹配当前执行的方法传入的参数为指定类型的执行方法
* @within：用于匹配所以持有指定注解类型内的方法
* @target：用于匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解
* @args：用于匹配当前执行的方法传入的参数持有指定注解的执行
* @annotation：用于匹配当前执行方法持有指定注解的方法
* bean：Spring AOP扩展的，AspectJ没有对于指示符，用于匹配特定名称的Bean对象的执行方法

bean注册方法：

二、第一种方式： XML配置方式

三、第二种方式： 使用@Component注解 + @ComponentScan包扫描方式

四、第三种方式：@Configuration + @Bean方式

五、第四种方式：FactoryBean方式

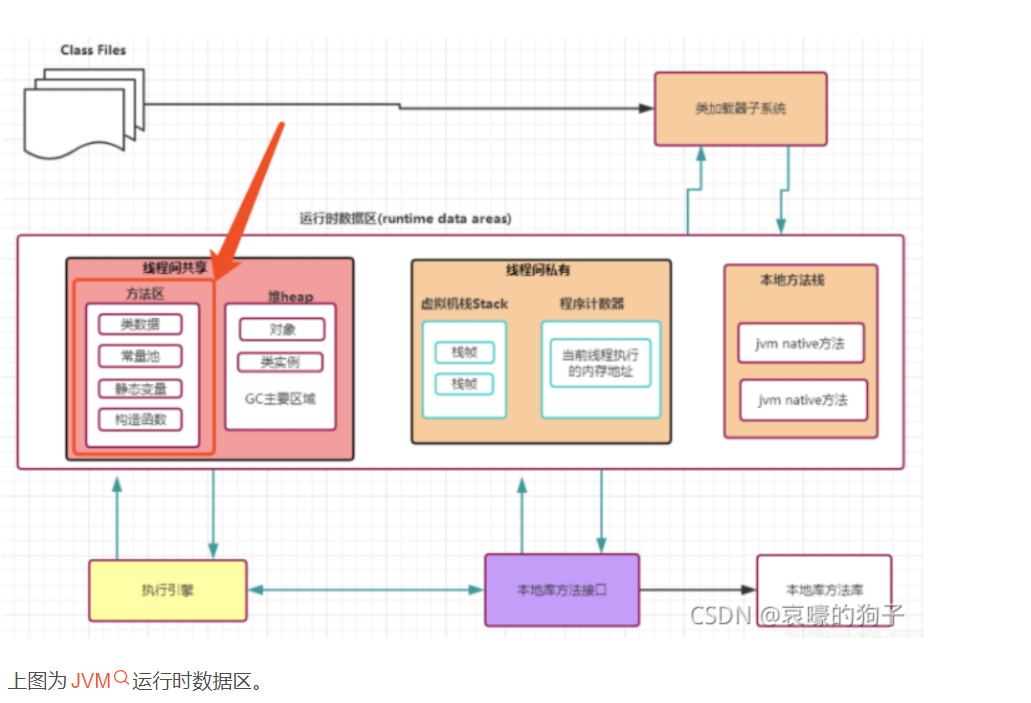
六、第五种方式：@Import方式

七、第六种方式：@Import + ImportSelector方式

八、第七种方式：@Import + ImportBeanDefinitionRegistrar方式

九、第八种方式：BeanDefinitionRegistryPostProcessor方式

十、第九种方式：BeanFactoryPostProcessor方式

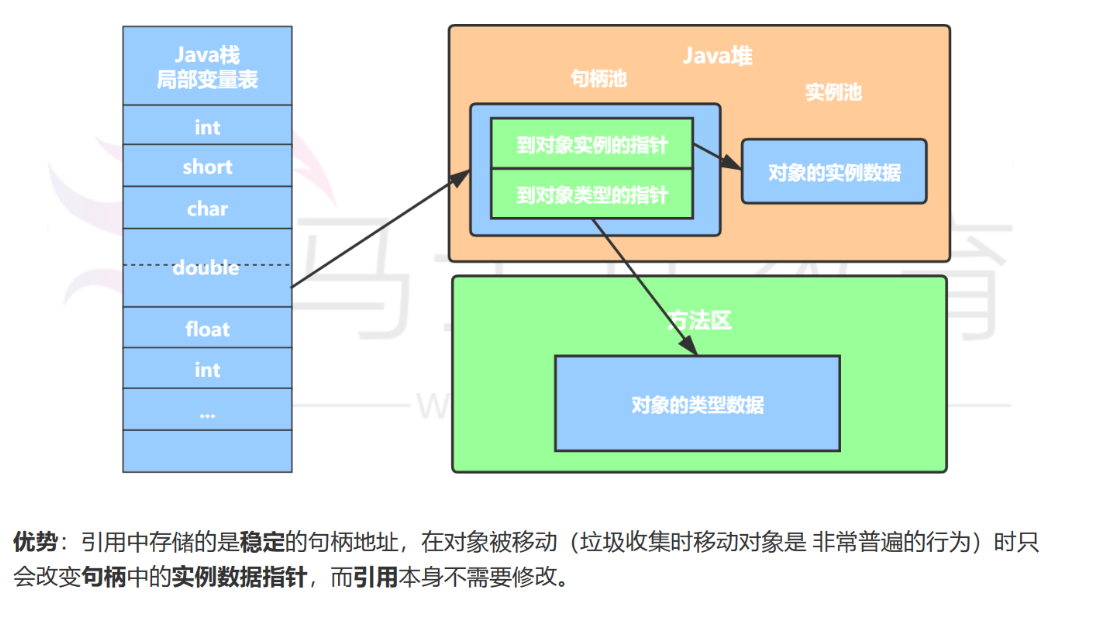


三种让线程等待和唤醒的方法如下：

方式一：使用 Object 中的 wait() 方法让线程等待，使用 Object 中的 notify() 方法唤醒线程

方式二：使用 JUC 包中 Condition 的 await() 方法让线程等待，使用 signal() 方法唤醒线程，必须被 LOCK包裹

方式三：LockSupport 类可以阻塞当前线程以及唤醒指定被阻塞的线程 LockSupport 和每个使用它的线程都有一个许可（permit）关联



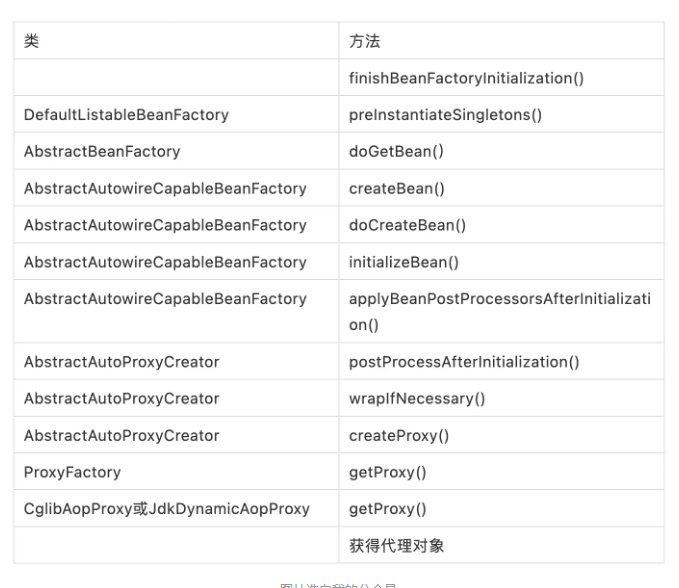
Index Nested-Loop Join（NLJ）算法是Join算法中最基本的算法之一。在NLJ算法中，MySQL首先选择一个表（通常是小型表）作为驱动表，并迭代该表中的每一行。然后，MySQL在第二个表中搜索匹配条件的行，这个搜索过程通常使用索引来完成。一旦找到匹配的行，MySQL将这些行组合在一起，并将它们作为结果集返回。

## Block Nested-Loop Join

Block Nested Loop Join（BNL）算法与NLJ算法不同的是，BNL算法使用一个类似于缓存的机制，将表数据分成多个块，然后逐个处理这些块，以减少内存和CPU的消耗。

@ComponentScan扫描的范围默认是它**所在的包以及子包中所有带有注解的类**  
@[EnableAutoConfiguration](https://so.csdn.net/so/search?q=EnableAutoConfiguration&spm=1001.2101.3001.7020)扫描的范围默认是它**所在类**

**实例化BEAN过程**



间隙锁/临键锁

默认情况下，InnoDB在 REPEATABLE READ事务隔离级别运行，InnoDB使用 next-key 锁进行搜索和索引扫描，以防止幻读。

索引上的等值查询(唯一索引)，给不存在的记录加锁时, 优化为间隙锁 。

索引上的等值查询(普通索引)，向右遍历时最后一个值不满足查询需求时，next-key lock 退化为间隙锁。

索引上的范围查询(唯一索引)–会访问到不满足条件的第一个值为止。

