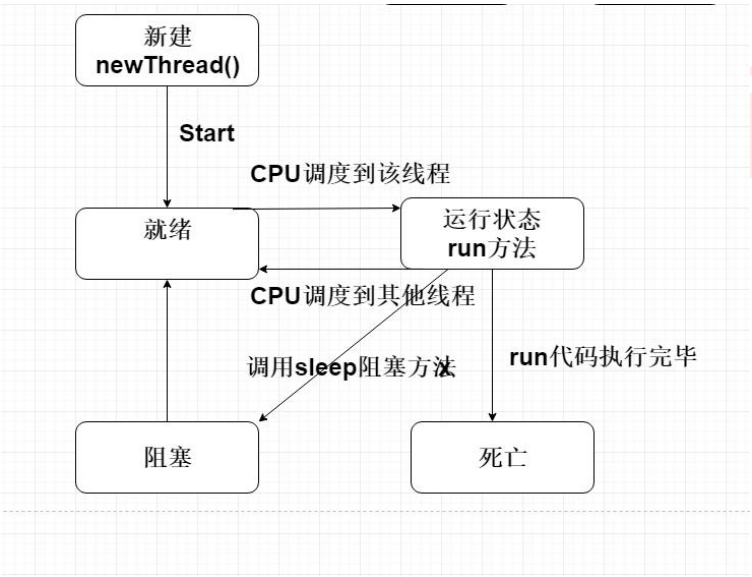
JUC 相关面试题

谈谈什么是线程池

线程池和数据库连接池非常类似，可以统一管理和维护线程，减少没有必要的开销。

为什么要使用线程池

因为频繁的开启线程或者停止线程，线程需要从新被 cpu 从就绪到运行状态调度，需要发生 cpu 的上下文切换，效率非常低。



面试宝典

你们哪些地方会使用到线程池

实际开发项目中 禁止自己 new 线程。
必须使用线程池来维护和创建线程。

线程池有哪些作用

核心点：复用机制 提前创建好固定的线程一直在运行状态 实现复用 限制线程创建数量。
1.降低资源消耗：通过池化技术重复利用已创建的线程，降低线程创建和销毁造成的损耗。
2.提高响应速度：任务到达时，无需等待线程创建即可立即执行。
3.提高线程的可管理性：线程是稀缺资源，如果无限制创建，不仅会消耗系统资源，还会因为线程的不合理分布导致资源调度失衡，降低系统的稳定性。使用线程池可以进行统一的分配、调优和监控。
4.提供更多更强大的功能：线程池具备可拓展性，允许开发人员向其中增加更多的功能。比如延时定时线程池 ScheduledThreadPoolExecutor，就允许任务延期执行或定期执行。

线程池的创建方式

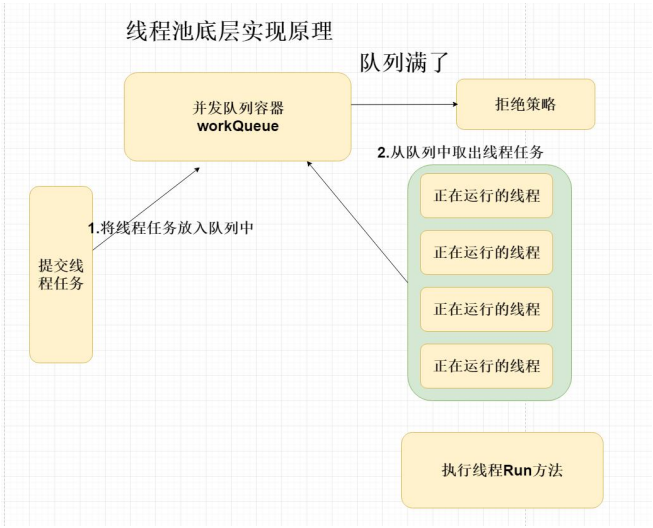
Executors.newCachedThreadPool(); 可缓存线程池
Executors.newFixedThreadPool(); 可定长度 限制最大线程数
Executors.newScheduledThreadPool() ； 可定时
Executors.newSingleThreadExecutor(); 单例
底层都是基于 ThreadPoolExecutor 构造函数封装

余胜军 Java架构面试宝典

线程池底层是如何实现复用的

本质思想：创建一个线程，不会立马停止或者销毁而是一直实现复用。

1. 提前创建固定大小的线程一直保持在正在运行状态；（可能会非常消耗 cpu 的资源）
2. 当需要线程执行任务，将该任务提交缓存在并发队列中；如果缓存队列满了，则会执行拒绝策略；
3. 正在运行的线程从并发队列中获取任务执行从而实现多线程复用问题；



线程池核心点：复用机制 ------

1. 提前创建好固定的线程一直在运行状态----死循环实现
2. 提交的线程任务缓存到一个并发队列集合中，交给我们正在运行的线程执行
3. 正在运行的线程就从队列中获取该任务执行

简单模拟手写 Java 线程池：

```
public class MyExecutors {
    public BlockingDeque<Runnable> runnables;
    private volatile Boolean isRun = true;

    /**
     * dequeSize 缓存任务大小
     *
     * @param dequeSize
     * @param threadCount 复用线程池
     */
    public MyExecutors(int dequeSize, int threadCount) {
        runnables = new LinkedBlockingDeque<Runnable>(dequeSize);
        for (int i = 0; i < threadCount; i++) {
            WorkThread workThread = new WorkThread();
            workThread.start();
        }
    }

    public void execute(Runnable runnable) {
        runnables.offer(runnable);
    }
```

```
    class WorkThread extends Thread {
        @Override
        public void run() {
            while (isRun||runnables.size()>0) {
                Runnable runnable = runnables.poll();
                if (runnable != null) {
                    runnable.run();
                }
            }
        }
    }

    public static void main(String[] args) {
        MyExecutors myExecutors = new MyExecutors(10, 2);
        for (int i = 0; i < 10; i++) {
            final int finalI = i;
            myExecutors.execute(new Runnable() {
                @Override
                public void run() {
                    System.out.println(Thread.currentThread().getName() + ":," + finalI);
                }
            });
        }
        myExecutors.isRun=false;
    }
}
```

ThreadPoolExecutor 核心参数有哪些

corePoolSize：核心线程数量　一直正在保持运行的线程
maximumPoolSize：最大线程数，线程池允许创建的最大线程数。
keepAliveTime：超出 corePoolSize 后创建的线程的存活时间。
unit：keepAliveTime 的时间单位。
workQueue：任务队列，用于保存待执行的任务。
threadFactory：线程池内部创建线程所用的工厂。
handler：任务无法执行时的处理器。

线程池创建的线程会一直在运行状态吗？

不会
例如：配置核心线程数 corePoolSize 为 2 、最大线程数 maximumPoolSize 为 5
我们可以通过配置超出 corePoolSize 核心线程数后创建的线程的存活时间例如为 60s
在 60s 内没有核心线程一直没有任务执行，则会停止该线程。

为什么阿里巴巴不建议使用 Executors

因为默认的 Executors 线程池底层是基于 ThreadPoolExecutor
构造函数封装的，采用无界队列存放缓存任务，会无限缓存任务容易发生
内存溢出，会导致我们最大线程数会失效。

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  keepAliveTime: 0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

余胜军 Java架构面试宝典

线程池底层 ThreadPoolExecutor 底层实现原理

1.当线程数小于核心线程数时，创建线程。
2.当线程数大于等于核心线程数，且任务队列未满时，将任务放入任务队列。
3.当线程数大于等于核心线程数，且任务队列已满
3.1 若线程数小于最大线程数，创建线程
3.2 若线程数等于最大线程数，抛出异常，拒绝任务

线程池队列满了，任务会丢失吗

如果队列满了，且任务总数>最大线程数则当前线程走拒绝策略。
可以自定义异拒绝异常，将该任务缓存到 redis、本地文件、mysql 中后期项目启动实现补偿。

1.AbortPolicy 丢弃任务，抛运行时异常
2.CallerRunsPolicy 执行任务
3.DiscardPolicy 忽视，什么都不会发生
4.DiscardOldestPolicy 从队列中踢出最先进入队列（最后一个执行）的任务

5.实现 RejectedExecutionHandler 接口，可自定义处理器

线程池拒绝策略类型有哪些呢

1.AbortPolicy 丢弃任务，抛运行时异常
2.CallerRunsPolicy 执行任务
3.DiscardPolicy 忽视，什么都不会发生
4.DiscardOldestPolicy 从队列中踢出最先进入队列（最后一个执行）的任务
5.实现 RejectedExecutionHandler 接口，可自定义处理器

线程池如何合理配置参数

自定义线程池就需要我们自己配置最大线程数 maximumPoolSize，为了高效的并发运行，当然这个不能随便设置。这时需要看我们的业务是 IO 密集型还是 CPU 密集型。

CPU 密集型
CPU 密集的意思是该任务需要大量的运算，而没有阻塞，CPU 一直全速运行。CPU 密集任务只有在真正的多核 CPU 上才可能得到加速(通过多线程)，而在单核 CPU 上，无论你开几个模拟的多线程该任务都不可能得到加速，因为 CPU 总的运算能力就那些。

余胜军 Java架构面试宝典

CPU 密集型任务配置尽可能少的线程数量：以保证每个 CPU 高效的运行一个线程。
一般公式：（CPU 核数+1）个 线程的线程池

IO 密集型
I0 密集型，即该任务需要大量的 IO，即大量的阻塞。在单线程上运行 I0 密集型的任务会导致浪费大量的 CPU 运算能力浪费在等待。
所以在 IO 密集型任务中使用多线程可以大大的加速程序运行，即使在单核 CPU 上，这种加速主要就是利用了被浪费掉的阻塞时间。

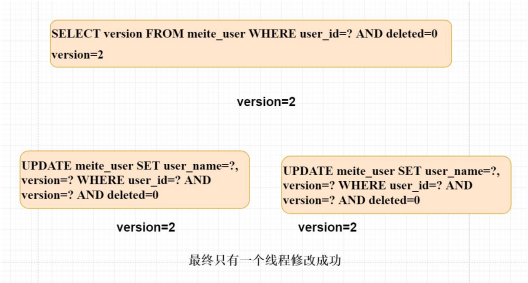
I0 密集型时，大部分线程都阻寒，故需要多配置线程数：
公式：

CPU 核数 * 2
CPU 核数 / （1 - 阻塞系数） 阻塞系数 在 0.8~0.9 之间
查看 CPU 核数：
System.out.println(Runtime.getRuntime().availableProcessors());

什么是悲观锁？什么是乐观锁

悲观锁：
1.站在 mysql 的角度分析：悲观锁就是比较悲观，当多个线程对同一行数据实现修改的时候，最后只有一个线程才能修改成功，只要谁能够对获取到行锁则其他线程时不能够对该数据做任何修改操作，且是阻塞状态。
2.站在 java 锁层面，如果没有获取到锁，则会阻塞等待，后期唤醒的锁的成本就会非常高，从新被我们 cpu 从就绪调度为运行状态。
Lock syn 锁　悲观锁没有获取到锁的线程会阻塞等待；

乐观锁：
乐观锁比较乐观，通过预值或者版本号比较，如果不一致性的情况则通过循环控制修改，当前线程不会被阻塞，是乐观，效率比较高，但是乐观锁比较消耗 cpu 的资源。



乐观锁：获取锁----如果没有获取到锁，当前线程是不会阻塞等待 通过死循环控制。
乐观锁属于无锁机制，没有竞争锁流程。

余胜军 Java架构面试宝典

注意：mysql 的 innodb 引擎中存在行锁的概念

Mysql 层面如何实现乐观锁呢

在我们表结构中，会新增一个字段就是版本字段
`version` varchar(255) DEFAULT NULL,
多个线程对同一行数据实现修改操作，提前查询当前最新的 version 版本号码，
作为 update 条件查询，如果当前 version 版本号码发生了变化，则查询不到该数据。
表示如果修改数据失败，则不断重试 ，有从新查询最新的版本实现 update。
需要注意控制乐观锁循环的次数，避免 cpu 飙高的问题。
mysql 的 innodb 引擎中存在行锁的概念

乐观锁实现方式

Java 有哪些锁的分类呢

1. 悲观与乐观锁
2. 公平锁与非公平锁
3. 自旋锁/重入锁
4. 重量级锁与轻量级锁
5. 独占锁与共享锁

公平锁与非公平锁之间的区别

公平锁：就是比较公平，根据请求锁的顺序排列，先来请求的就先获取锁，后来获取锁就最后获取到， 采用队列存放 类似于吃饭排队。
非公平锁：不是据请求的顺序排列， 通过争抢的方式获取锁。
非公平锁效率是公平锁效率要高，Synchronized 是非公平锁

New ReentramtLock()(true)---公平锁
New ReentramtLock()(false)---非公平锁
底层基于 aqs 实现

余胜军 Java架构面试宝典

公平锁底层是如何实现的

公平锁：就是比较公平，根据请求锁的顺序排列，先来请求的就先获取锁，后来获取锁就最后获取到， 采用队列存放 类似于吃饭排队。

队列---底层实现方式---数组或者链表实现

独占锁与共享锁之间的区别

独占锁：在多线程中，只允许有一个线程获取到锁，其他线程都会等待。
共享锁：多个线程可以同时持有锁，例如 ReentrantLock 读写锁。读读可以共享、写写互斥、读写互斥、写读互斥。

什么是锁的可重入性

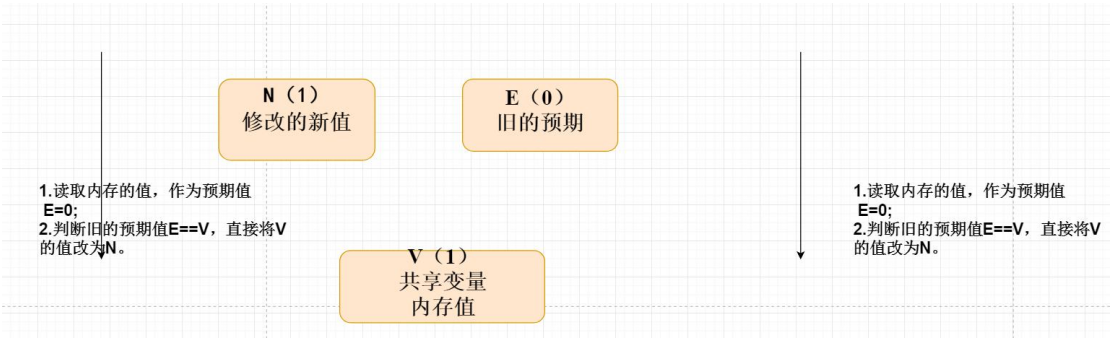
在同一个线程中锁可以不断传递的，可以直接获取。
Syn/lock
aqs

什么是 CAS（自旋锁），它的优缺点

Syn/lock

CAS
没有获取到锁的线程是不会阻塞的，通过循环控制一直不断的获取锁。

CAS: Compare and Swap，翻译成比较并交换。 执行函数 CAS（V，E,N）
CAS 有 3 个操作数，内存值 V，旧的预期值 E，要修改的新值 N。当且仅当预期值 E 和内存值 V 相同时，将内存值 V 修改为 N，否则什么都不做。



1. Cas 是通过硬件指令，保证原子性
2. Java 是通过 unsafe jni 技术

原子类： AtomicBoolean，AtomicInteger，AtomicLong 等使用 CAS 实现。

优点：没有获取到锁的线程，会一直在用户态，不会阻塞，没有锁的线程会一直通过循环控制重试。
缺点：通过死循环控制，消耗 cpu 资源比较高，需要控制循次数，避免 cpu 飙高问题；

Cas 本质的原理：
旧的预期值===v(共享变量中值)，才会修改我们 v。

基于 cas 实现锁机制原理

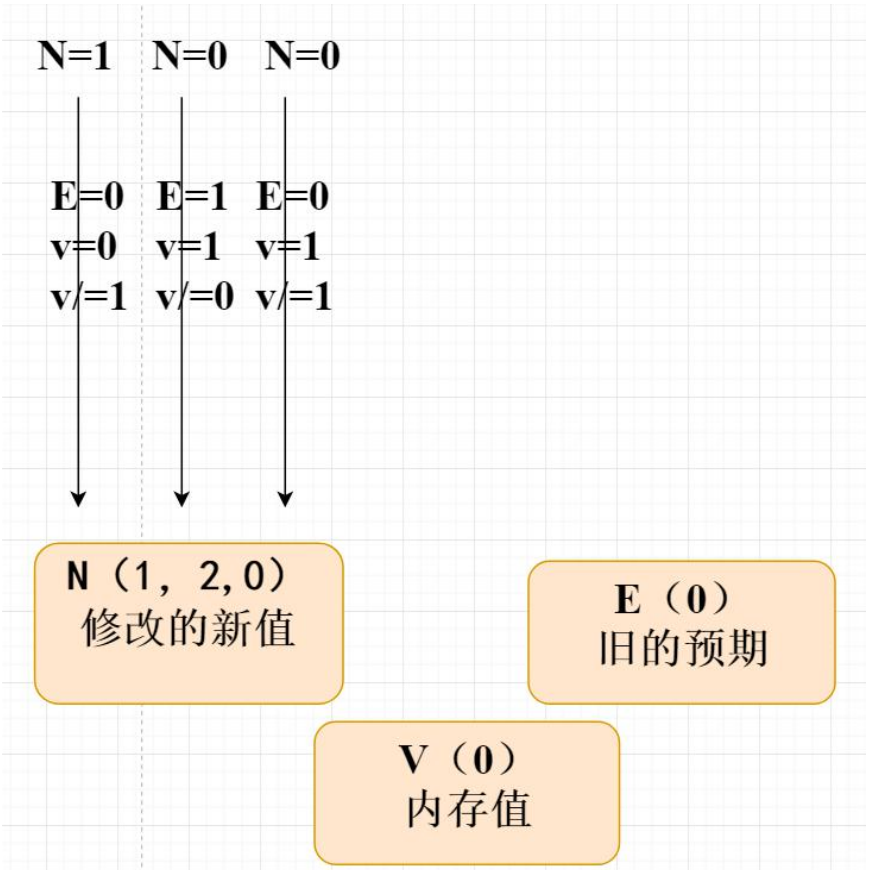
Cas 无锁机制原理：
1. 定义一个锁的状态；
2. 状态状态值=0 则表示没有线程获取到该锁；
3. 状态状态值=1 则表示有线程已经持有该锁；
实现细节：
CAS 获取锁：
将该锁的状态从 0 改为 1-----能够修改成功 cas 成功则表示获取锁成功
如果获取锁失败--修改失败，则不会阻塞而是通过循环（自旋来控制重试）
CAS 释放锁：
将该锁的状态从 1 改为 0 如果能够改成功 cas 成功则表示释放锁成功。

CAS 如何解决 ABA 的问题

Cas 主要检查 内存值 V 与旧的预值值=E 是否一致，如果一致的情况下，则修改。
这时候会存在 ABA 的问题：
如果将原来的值 A,改为了 B，B 有改为了 A 发现没有发生变化，实际上已经发生了变化，所以存在 Aba 问题。
解决办法：通过版本号码，对每个变量更新的版本号码做+1

解决 aba 问题是否大：概念产生冲突，但是不影响结果，换一种方式 通过版本号码方式。

余胜军 Java架构面试宝典



余胜军 Java架构面试宝典

```
/**
 * 演示 aba 的问题
 * （1）第一个参数 expectedReference：表示预期值。
 * （2）第二个参数 newReference：表示要更新的值。
 * （3）第三个参数 expectedStamp：表示预期的时间戳。
 * （4）第四个参数 newStamp：表示要更新的时间戳。
 */
public class AtomicMarkableReferenceTest {
    // 注意：如果引用类型是 Long、Integer、Short、Byte、Character 一定一定要注意值的缓存区间！
    // 比如 Long、Integer、Short、Byte 缓存区间是在 -128~127，会直接存在常量池中，而不在这个区间内对象的值则会每次都 new 一个对象，那么即使两个对象的值相同，CAS 方法都会返回 false
    // 先声明初始值，修改后的值和临时的值是为了保证使用 CAS 方法不会因为对象不一样而返回 false
    private static final Integer INIT_NUM = 1000;
    private static final Integer UPDATE_NUM = 100;
    private static final Integer TEM_NUM = 200;
    private static AtomicStampedReference atomicStampedReference = new
AtomicStampedReference(INIT_NUM, 1);
```

```
    public static void main(String[] args) {
        new Thread(() -> {
            Integer value = (Integer) atomicStampedReference.getReference();
            int stamp = atomicStampedReference.getStamp();
            System.out.println(Thread.currentThread().getName() + "：当前值为：" + value + "版本号为：" +
stamp);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // value 旧值 内存中的值    UPDATE_NUM 修改的值
            if (atomicStampedReference.compareAndSet(value, UPDATE_NUM, 0, stamp + 1)) {
                System.out.println(Thread.currentThread().getName() + "：当前值为：" +
atomicStampedReference.getReference() + "版本号为：" + atomicStampedReference.getStamp());
            } else {
                System.out.println("版本号不同，更新失败！");
            }
        }, "线程 A").start();

    }
}
```

余胜军 Java未得旧此主类

利用原子类手写 CAS 无锁

```
/**
 * 利用 cas 手写 锁
 */
public class AtomicTryLock {

 private AtomicLong atomicLong=new AtomicLong(0);
 private  Thread lockCurrentThread;
    /**
     * 1 表示锁已经被获取 0 表示锁没有获取 利用 cas 将 0 改为 1 成功则表示获取锁
     * @return
     */
    public boolean lock(){
        return atomicLong.compareAndSet(0, 1);
    }
    public boolean unlock(){
        if(lockCurrentThread!=Thread.currentThread()){
```

```
            return false;
        }
        return atomicLong.compareAndSet(1, 0);
    }

    public static void main(String[] args) {
        AtomicTryLock atomicTryLock = new AtomicTryLock();
        IntStream.range(1, 10).forEach((i) -> new Thread(() -> {

            try {
                boolean result = atomicTryLock.lock();
                if (result) {
                    atomicTryLock.lockCurrentThread=Thread.currentThread();
                    System.out.println(Thread.currentThread().getName() + ",获取锁成功~");
                } else {
                    System.out.println(Thread.currentThread().getName() + ",获取锁失败~");
                }
            } catch (Exception e) {
            } finally {
                if(atomicTryLock!=null){
                    atomicTryLock.unlock();
                }
            }

        }).start());
    }

}
```

谈谈你对 Threadlocal 理解？

ThreadLocal 提供了线程本地变量，它可以保证访问到的变量属于当前线程，每个线程都保存有一个变量副本，每个线程的变量都不同。ThreadLocal 相当于提供了一种线程隔离，将变量与线程相绑定。

Threadlocal 适用于在多线程的情况下，可以实现传递数据，实现线程隔离。

ThreadLocal 提供给我们每个线程缓存局部变量

Threadlocal 基本 API

1.New Threadlocal();---创建 Threadlocal

2.set 设置当前线程绑定的局部变量

3.get　获取当前线程绑定的局部变量
4.remove()　移除当前线程绑定的变量

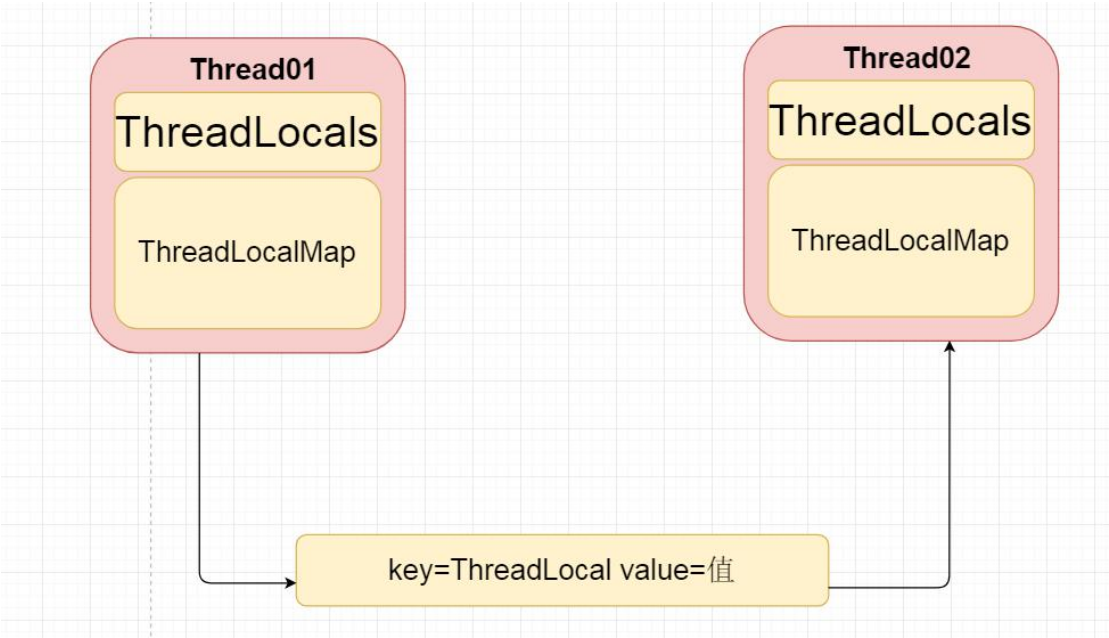
哪些地方有使用 Threadlocal

1.Spring 事务模板类
2.获取 HttpRequest
3.Aop 调用链

Threadlocal 底层实现原理

1. 在每个线程中都有自己独立的 ThreadLocalMap 对象，中 Entry 对象。
2. 如果当前线程对应的的 ThreadLocalMap 对象为空的情况下，则创建该 ThreadLocalMap 对象，并且赋值键值对。

Key 为 当前 new ThreadLocal 对象，value 就是为 object 变量值。



为什么线程缓存的是 ThreadlocalMap 对象

ThreadLocalMap 可以存放 n 多个不同的 ThreadLocal 对象；
每个 ThreadLocal 对象只能缓存一个变量值；

ThreadLocalMap<ThreadLocal 对象,value> threadLocalMap
ThreadLocal.get();

threadLocalMap.get(ThreadLocal)-----缓存变量值

谈谈强、软、弱、虚引用 区别

强引用： 当内存不足时，JVM 开始进行 GC(垃圾回收)，对于强引用对象，就算是出现了 OOM 也不会对该对象进行回收，死都不会收。

软引用：当系统内存充足的时候，不会被回收；当系统内存不足时，它会被回收，软引用通常用在对内存敏感的 程序中，比如高速缓存就用到软引用，内存够用时就保留，不够时就回收。

弱引用：弱引用需要用到 java.lang.ref.WeakReference 类来实现，它比软引用的生存周期更短。对于只有弱引用的对象来说，只要有垃圾回收，不管 JVM 的内存空间够不够用，都会回收该对象占用的内存空间。

虚：虚引用需要 java.lang.ref.Phantomreference 类来实现。顾名思义，虚引用就是形同虚设。与其它几种引用不同，虚引用并不会决定对象的声明周期。

Threadlocal 为何引发内存泄漏问题

余胜军 Java架构面试宝典

补充概念：
什么是内存泄漏问题
内存泄漏 表示就是我们程序员申请了内存，但是该内存一直无法释放；
内存泄漏溢出问题：
申请内存时，发现申请内存不足，就会报错 内存溢出的问题；

因为每个线程中都有自己独立的 ThreadLocalMap 对象，key 为 ThreadLocal，value 是为变量值。

Key 为 ThreadLocal 作为 Entry 对象的 key，是弱引用，当 ThreadLocal 指向 null 的时候，Entry 对象中的 key 变为 null，GC 如果没有清理垃圾时，则该对象会一直无法被垃圾收集机制回收，一直占用到了系统内存，有可能会发生内存泄漏的问题。

如何防御 Threadlocal 内存泄漏问题

1. 可以自己调用 remove 方法将不要的数据移除避免内存泄漏的问题；
2. 每次在做 set 方法的时候会清除之前 key 为 null；
3. Threadlocal 为弱引用；

Threadlocal 采用弱引用而不是强引用

1.如果 key 是为强引用： 当我们现在将 ThreadLocal 的引用指向为 null，但是每个线程中有自己独立 ThreadLocalMap 还一直在继续持有该对象，但是我们 ThreadLocal 对象不会被回收，就会发生 ThreadLocal 内存泄漏的问题。

2.如果 key 是为弱引用：
当我们现在将 ThreadLocal 的引用指向为 null，Entry 中的 key 指向为 null，但是下次调用 set 方法的时候，会根据判断如果 key 空的情况下，直接删除，避免了 Entry 发生内存泄漏的问题。

3.不管是用强引用还是弱引用都是会发生内存泄漏的问题。
弱引用中不会发生 ThreadLocal 内存泄漏的问题。

4.但是最终根本的原因 Threadlocal 内存泄漏的问题，产生于 ThreadLocalMap 与我们当前线程的生命周期一样，如果没有手动的删除的情况下，就有可能会发生内存泄漏的问题。

余胜军 Java架构面试宝典