

JUC-learning

1. 序章

JUC是什么？

java.util.concurrent在并发编程中使用的工具包

java.util.concurrent
java.util.concurrent.atomic
java.util.concurrent.locks

2. 多线程基础知识复习

2.1 start一个线程

start()方法底层

```
private native void start0();
```

对于start0()方法而言，实质是由jvm配合操作系统，底层由操作系统分配一个原生的线程

thread.c

```
42
43▼ static JNINativeMethod methods[] = {
44    {"start0",           "()V",          (void *)&JVM_StartThread},
45    {"stop0",            "(" OBJ ")V",  (void *)&JVM_StopThread},
46    {"isAlive",          "()Z",          (void *)&JVM_IsThreadAlive},
47    {"suspend0",         "()V",          (void *)&JVM_SuspendThread},
48    {"resume0",          "()V",          (void *)&JVM_ResumeThread},
49    {"setPriority0",     "(I)V",        (void *)&JVM_SetThreadPriority},
50    {"yield",            "()V",          (void *)&JVM_Yield},
51    {"sleep",            "(J)V",        (void *)&JVM_Sleep},
52    {"currentThread",   "()" THD,      (void *)&JVM_CurrentThread},
53    {"countStackFrames", "()I",        (void *)&JVM_CountStackFrames},
54    {"interrupt0",      "()V",        (void *)&JVM_Interrupt},
55    {"isInterrupted",   "(Z)Z",       (void *)&JVM_IsInterrupted},
56    {"holdsLock",        "(" OBJ ")Z", (void *)&JVM_HoldsLock},
57    {"getThreads",       "()[" THD,    (void *)&JVM_GetAllThreads},
58    {"dumpThreads",     "([ " THD ")[ " STE, (void *)&JVM_DumpThreads},
59    {"setNativeName",   "(" STR ")V", (void *)&JVM_SetNativeThreadName},
60 };
61 }
```

jvm.cpp

```

2816 ▼ JVM_ENTRY(void, [JVM_StartThread](JNIEnv* env, jobject jthread))
2817     JVMWrapper("[JVM_StartThread]");
2818     JavaThread *native_thread = NULL;
2819
2820     // We cannot hold the Threads_lock when we throw an exception,
2821     // due to rank ordering issues. Example: we might need to grab the
2822     // Heap_lock while we construct the exception.
2823     bool throw_illegal_thread_state = false;
2824
2825     // We must release the Threads_lock before we can post a jvmti event
2826     // in Thread::start.
2827     {
2828         // Ensure that the C++ Thread and OSThread structures aren't freed before
2829         // we operate.
2830         MutexLocker mu(Threads_lock);
2831
2832         // Since JDK 5 the java.lang.Thread threadStatus is used to prevent
2833         // re-starting an already started thread, so we should usually find
2834         // that the JavaThread is null. However for a JNI attached thread
2835         // there is a small window between the Thread object being created
2836         // (with its JavaThread set) and the update to its threadStatus, so we
2837         // have to check for this
2838         if (java_lang_Thread::thread(JNIHandles::resolve_non_null(jthread)) != NULL) {
2839             throw_illegal_thread_state = true;
2840         } else {
2841
2842             if (throw_illegal_thread_state) {
2843                 THROW(vmSymbols::java_lang_IllegalThreadStateException());
2844             }
2845
2846             assert(native_thread != NULL, "Starting null thread?");
2847
2848             if (native_thread->osthread() == NULL) {
2849                 // No one should hold a reference to the 'native_thread'.
2850                 delete native_thread;
2851                 if (JvmtiExport::should_post_resource_exhausted()) {
2852                     JvmtiExport::post_resource_exhausted(
2853                         JVMTI_RESOURCE_EXHAUSTED_OOM_ERROR | JVMTI_RESOURCE_EXHAUSTED_THREADS,
2854                         "unable to create new native thread");
2855                 }
2856                 THROW_MSG(vmSymbols::java_lang_OutOfMemoryError(),
2857                         "unable to create new native thread");
2858             }
2859
2860             Thread::start(native_thread);
2861
2862         }
2863
2864     }
2865
2866     JVM_END

```

thread.cpp

```

451 ▼ void Thread::start(Thread* thread) {
452     trace("start", thread);
453     // Start is different from resume in that its safety is guaranteed by context or
454     // being called from a Java method synchronized on the Thread object.
455     if (!DisableStartThread) {
456         if (thread->is_Java_thread()) {
457             // Initialize the thread state to RUNNABLE before starting this thread.
458             // Can not set it after the thread started because we do not know the
459             // exact thread state at that time. It could be in MONITOR_WAIT or
460             // in SLEEPING or some other state.
461             java_lang_Thread::set_thread_status(((JavaThread*)thread)->threadObj(),
462                                                 java_lang_Thread::RUNNABLE);
463         }
464         os::start_thread(thread);
465     }
466 }

```

2.2 Java多线程相关概念

- 1把锁
 - synchronized
- 2个并

- 并发 (concurrent)
 - 是在同一个实体上的多个事件
 - 是在一台处理器上“同时”处理多个任务
 - 同一时刻，其实是只有一个事件在发生
- 并行 (parallel)
 - 是在不同尸体上的多个事件
 - 是在多台处理器上同时处理多个任务
 - 同一时刻，你做你的，我做我的
- 3个程
 - 进程
 - 简单的说，在系统中运行的一个应用程序就是一个进程，每一个进程都有它自己的内存空间和系统资源
 - 线程
 - 也被称为轻量级进程，在同一个进程中会有1个或多个线程，是大多数操作系统进行时序调度的基本单元
 - 管程
 - Monitor (监视器)，也就是我们平时说的锁。Monitor其实是一种同步机制，它的义务是保证（同一时间）只有一个线程可以访问被保护的数据和代码

2.3 用户线程和守护线程

- 用户线程 (User Thread)
 - 是系统的工作线程，它会完成这个程序需要完成的业务操作
- 守护线程 (Daemon Thread)
 - 是一种特殊的线程，为其他线程服务的，在后台默默完成一些系统性的服务，比如垃圾回收线程
 - 守护线程作为一个服务线程，没有服务对象就没有必要继续运行了，如果用户线程全部结束了，意味着程序需要完成的业务操作已经结束了，系统可以退出了

如果用户线程全部结束意味着程序需要完成的业务操作已经结束了，守护线程随着JVM一同结束工作

```
public class DaemonDemo {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            System.out.println(Thread.currentThread().getName() + "\t 开始运行, "
+ (Thread.currentThread().isDaemon() ? "守护线程" : "用户线程"));
            while (true) {
                }
            }, "t1");
        // 该方法必须设置在start()方法之前，不然会包IllegalThreadStateException
        t1.setDaemon(true);
        t1.start();
        // 暂停3秒钟线程
        try { TimeUnit.SECONDS.sleep(3); } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
        System.out.println(Thread.currentThread().getName() + "\t ----end 主线  
程");  
    }  
}
```

3. CompletableFuture

3.1 Future接口理论知识复习

Future接口(FutureTask实现类)定义了操作**异步任务执行一些方法**，如获取异步任务的执行结果、取消任务的执行、判断任务是否被取消、判断任务执行是否完毕等。

比如主线程让一个子线程去执行任务，子线程可能比较耗时，启动子线程开始执行任务后，主线程就去做其他事情了，忙其它事情或者先执行完，过了一会才去获取子任务的执行结果或变更的任务状态。

Future接口：可以为主线程开一个分支任务，专门为主线程处理耗时和费力的复杂业务

3.2 Future接口常用实现类FutureTask异步任务

3.2.1 Future接口能干什么

Future是Java5新加的一个接口，它提供了一种**异步并行计算的功能**。

如果主线程需要执行一个很耗时的计算任务，我们就可以通过future把这个任务放到异步线程中执行。

主线程继续处理其他任务或者先行结束，再通过Future获取计算结果。

目的:异步多线程任务执行且返回有结果，三个特点: 多线程/有返回/异步任务

3.2.2 Future接口相关架构

FutureTask实现了RunnableFuture接口（继承了Runnable, Future接口），还通过构造器注入Callable接口，实现以上多线程/有返回/异步任务三个特点

```
public class FutureTaskDemo {  
  
    public static void main(String[] args) throws ExecutionException,  
InterruptedException {  
        FutureTask<String> futureTask = new FutureTask<>(new MyThread());  
  
        Thread t1 = new Thread(futureTask, "t1");  
        t1.start();  
  
        System.out.println(futureTask.get());  
    }  
}  
  
class MyThread implements Callable<String> {  
  
    @Override  
    public String call() throws Exception {  
        System.out.println("----come in call()");  
        return "hello";  
    }  
}
```

```
}
```

3.2.3 Future编码实战和优缺点分析

- 优点

- future+线程池异步多线程任务配合，能显著提高程序的执行效率

```
public class FutureThreadPoolDemo {  
  
    public static void main(String[] args) throws ExecutionException,  
InterruptedException {  
        // 3个任务，目前开启多个异步任务线程来处理，请问耗时多少（未使用get()方法  
357毫秒，使用get()方法894毫秒）  
        ExecutorService threadPool = Executors.newFixedThreadPool(3);  
  
        long startTime = System.currentTimeMillis();  
  
        FutureTask<String> futureTask1 = new FutureTask<>(() -> {  
            try { TimeUnit.MILLISECONDS.sleep(500); }catch  
(InterruptedException e){ e.printStackTrace(); }  
            return "task1 over";  
        });  
        threadPool.submit(futureTask1);  
  
        FutureTask<String> futureTask2 = new FutureTask<>(() -> {  
            try { TimeUnit.MILLISECONDS.sleep(300); }catch  
(InterruptedException e){ e.printStackTrace(); }  
            return "task1 over";  
        });  
        threadPool.submit(futureTask2);  
  
        System.out.println(futureTask1.get());  
        System.out.println(futureTask2.get());  
  
        try { TimeUnit.MILLISECONDS.sleep(300); }catch  
(InterruptedException e){ e.printStackTrace(); }  
  
        long endTime = System.currentTimeMillis();  
        System.out.println("---constTime:" + (endTime - startTime) + "毫  
秒");  
        threadPool.shutdown();  
    }  
  
    private static void m1() {  
        // 3个任务，目前只有一个线程main来处理，请问耗时多少（1125毫秒）  
        long startTime = System.currentTimeMillis();  
  
        try { TimeUnit.MILLISECONDS.sleep(500); }catch  
(InterruptedException e){ e.printStackTrace(); }  
        try { TimeUnit.MILLISECONDS.sleep(300); }catch  
(InterruptedException e){ e.printStackTrace(); }  
        try { TimeUnit.MILLISECONDS.sleep(300); }catch  
(InterruptedException e){ e.printStackTrace(); }  
  
        long endTime = System.currentTimeMillis();  
        System.out.println("---constTime:" + (endTime - startTime) + "毫  
秒");  
    }  
}
```

```
        System.out.println(Thread.currentThread().getName() + "\t -----  
end");  
    }  
}
```

- 缺点

- get()阻塞

```
■   
/**  
 * @Author cxl  
 * @Date 21/5/2023 10:07  
 * @ClassReference: com.cxl.juc.completableFuture.FutureAPIDemo  
 * @Description:  
 * 1.get容易导致阻塞，一般建议放在程序后后面，一旦调用不见不散，非要等到结果才会离开，不管你是否计算完成，容器程序堵塞  
 * 2.假如我不愿意等待很长时间，我希望过时不候，可以自动离开  
 */  
public class FutureAPIDemo {  
  
    public static void main(String[] args) throws  
ExecutionException, InterruptedException, TimeoutException {  
        FutureTask<String> futureTask = new FutureTask<>(() -> {  
            System.out.println(Thread.currentThread().getName() +  
"\t -----come in");  
            // 暂停3秒钟线程  
            try { TimeUnit.SECONDS.sleep(5); } catch  
(InterruptedException e){ e.printStackTrace(); }  
            return "task over";  
        });  
  
        Thread t1 = new Thread(futureTask, "t1");  
        t1.start();  
  
        System.out.println(Thread.currentThread().getName() + "\t --忙其他任务了");  
  
        // System.out.println(futureTask.get());  
        System.out.println(futureTask.get(3, TimeUnit.SECONDS));  
    }  
}
```

- isDone()轮询
 - 轮询的方式会耗费无畏的CPU资源，而且也不见得能及时地得到计算结果
 - 如果想要异步获取结果，通常都会以轮询的方式去获取结果 尽量不要阻塞

```

■ while(true) {
    if (futureTask.isDone()) {
        System.out.println(futureTask.get());
        break;
    } else {
        try { TimeUnit.MILLISECONDS.sleep(500); }catch
(InterruptedException e){ e.printStackTrace(); }
        System.out.println("正在处理中，不要再催了，越催越慢，再催熄火");
    }
}

```

- Future对于结果的获取不是很友好，只能通过阻塞或轮询的方式得到任务的结果

3.3 CompletableFuture对Future的改进

3.3.1 CompletableFuture为什么出现

get()方法在Future计算完成之前会一直处在阻塞状态下，isDone()方法容易耗费CPU资源，对于真正的异步处理我们希望通过传入回调函数，在Future结束时自动调用该回调函数，这样，我们就不用等待结果。

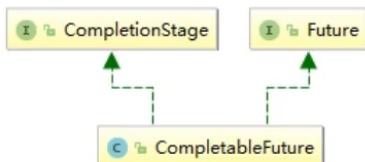
阻塞的方式和异步编程的设计理念相违背，而轮询的方式会耗费无谓的CPU资源

因此，JDK8设计出CompletableFuture。

CompletableFuture提供了一种观察者模式类似的机制，可以让任务执行完成后通知监听的一方。

3.3.2 CompletableFuture和CompletionStage源码分别介绍

类架构说明



```

public class CompletableFuture<T> implements Future<T>, CompletionStage<T> {

```

接口CompletionStage

CompletionStage

- CompletionStage代表异步计算过程中的某一个阶段，一个阶段完成以后可能会触发另外一个阶段
- 一个阶段的计算执行可以是一个Function、Consumer或者Runnable。比如：stage.thenApply(x -> square(x)).thenAccept(x -> System.out.print(x)).thenRun(() -> System.out.println())
- 一个阶段的执行可能是被单个阶段的完成触发，也可能是由多个阶段一起触发

代表异步计算过程中的某一个阶段，一个阶段完成以后可能会触发另外一个阶段，有些类似Linux系统的管道分隔符传参数

类CompletableFuture

CompletableFuture

- 在Java8中，CompletableFuture提供了非常强大的Future的扩展功能，可以帮助我们简化异步编程的复杂性，并且提供了函数式编程的能力，可以通过回调的方式处理计算结果，也提供了转换和组合CompletableFuture的方法。
- 它可能代表一个明确完成的Future，也有可能代表一个完成阶段（CompletionStage），它支持在计算完成以后触发一些函数或执行某些动作。
- 它实现了Future和CompletionStage接口

3.3.3 核心的四个静态方法，来创建一个异步任务

runAsync无返回值

- `public static CompletableFuture<Void> runAsync(Runnable runnable)`
- `public static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)`

supplyAsync有返回值

- `public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)`
- `public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)`

上述Executor executor参数说明

- 没有指定Executor的方法，直接使用默认的ForkJoinPool.commonPool()作为它的线程池执行异步代码。
- 如果指定线程池，则使用我们自定义的或者特别指定的线程池执行异步代码

Code

```
public class CompletableFutureDemo {  
  
    public static void main(String[] args) throws ExecutionException,  
InterruptedException {  
    ExecutorService threadPool = Executors.newFixedThreadPool(3);  
  
    // 输出结果:  
    // pool-1-thread-1  
    // null  
    /*CompletableFuture<Void> completableFuture =  
CompletableFuture.runAsync(() -> {  
    System.out.println(Thread.currentThread().getName());  
    // 暂停3秒钟线程  
    try { TimeUnit.SECONDS.sleep(3); } catch (InterruptedException e){  
e.printStackTrace(); }  
}, threadPool);  
  
    System.out.println(completableFuture.get());*/  
  
    // 输出结果:  
    // pool-1-thread-1  
    // hello supplyAsync  
    CompletableFuture<String> completableFuture =  
CompletableFuture.supplyAsync(() -> {  
    System.out.println(Thread.currentThread().getName());  
    // 暂停3秒钟线程
```

```

        try { TimeUnit.SECONDS.sleep(3); }catch (InterruptedException e){
e.printStackTrace(); }
        return "hello supplyAsync";
}, threadPool);

System.out.println(completableFuture.get());

threadPool.shutdown();
}
}

```

通用功能，减少阻塞和轮询

从Java8开始引入了CompletableFuture，它是Future的功能增强版，**减少阻塞和轮询**可以传入回调对象，当异步任务完成或者发生异常时，自动调用回调对象的回调方法

CompletableFuture的优点

- 异步任务结束时，会自动回调某个对象的方法
- 主线程设置好回调后，不再关心异步任务的执行，异步任务之间可以顺序执行
- 异步任务出错时，会自动回调某个对象的方法

```

public class CompletableFutureUseDemo {

    /**
     * 出现异常时输出结果:
     * pool-1-thread-1----come in
     * main线程先去忙其他任务
     * -----1秒钟后出结果: 1
     * 异常情况: java.lang.ArithmetricException: / by zero
java.lang.ArithmetricException: / by zero
     * java.util.concurrent.CompletionException: java.lang.ArithmetricException:
/ by zero
     *
     * 未出现异常时的结果:
     * pool-1-thread-1----come in
     * main线程先去忙其他任务
     * -----1秒钟后出结果: 3
     * -----计算完成，更新系统updateValue: 3
    */
    public static void main(String[] args) throws ExecutionException,
InterruptedException {

        ExecutorService threadPool = Executors.newFixedThreadPool(3);

        try {
            CompletableFuture.supplyAsync(() -> {
                System.out.println(Thread.currentThread().getName() + "----come
in");
                int result = ThreadLocalRandom.current().nextInt(10);
                try { TimeUnit.SECONDS.sleep(1); }catch (InterruptedException e)
{ e.printStackTrace(); }
                System.out.println("-----1秒钟后出结果: " + result);
                if (result > 5) {
                    int i = 10/0;
                }
                return result;
            })
        }
    }
}

```

```

        }, threadPool).whenComplete((v, e) -> {
            if (e == null) {
                System.out.println("----计算完成，更新系统UpdateValue: " + v);
            }
        }).exceptionally(e -> {
            e.printStackTrace();
            System.out.println("异常情况: " + e.getCause() + "\t" +
e.getMessage());
            return null;
        });
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        threadPool.shutdown();
    }

    System.out.println(Thread.currentThread().getName() + "线程先去忙其他任
务");

    // 主线程不要立刻结束，否则CompletableFuture默认使用的线程池会立刻关闭
    // try { TimeUnit.SECONDS.sleep(3); }catch (InterruptedException e){
e.printStackTrace(); }

}

/**
 * 返回结果
 * main线程先去忙其他任务
 * ForkJoinPool.commonPool-worker-9----come in
 * ----3秒钟后出结果: 2
 * 2
 */
private static void future1() throws InterruptedException,
ExecutionException {
    CompletableFuture<Integer> completableFuture =
CompletableFuture.supplyAsync(() -> {
        System.out.println(Thread.currentThread().getName() + "----come
in");
        int result = ThreadLocalRandom.current().nextInt(10);
        // 暂停3秒钟线程
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("----3秒钟后出结果: " + result);
        return result;
    });

    System.out.println(Thread.currentThread().getName() + "线程先去忙其他任
务");

    System.out.println(completableFuture.get());
}
}

```

3.4 案例——从电商网站的比价需求

3.4.1 函数式编程

函数式接口名称	方法名称	参数	返回值
Runnable	run	无参数	无返回值
Function	apply	1个参数	有返回值
Consumer	accept	1个参数	无返回值
Supplier	get	没有参数	有返回值
BiConsumer	accpet	2个参数	无返回值

Runnable

Runnable

Runnable 我们已经说过无数次了，无参数，无返回值

```
1 | @FunctionalInterface
2 | public interface Runnable {
3 |     public abstract void run();
4 | }
```

Function

Function

Function<T, R> 接受一个参数，并且有返回值

```
1 | @FunctionalInterface
2 | public interface Function<T, R> {
3 |     R apply(T t);
4 | }
```

Consumer

Consumer

Consumer 接受一个参数，没有返回值

```
1 | @FunctionalInterface
2 | public interface Consumer<T> {
3 |     void accept(T t);
4 | }
```

BiConsumer

BiConsumer

BiConsumer<T, U> 接受两个参数 (Bi, 英文单词词根, 代表两个的意思), 没有返回值

```
1 | @FunctionalInterface
2 | public interface BiConsumer<T, U> {
3 |     void accept(T t, U u);
```

Supplier

Supplier

Supplier 没有参数, 有一个返回值

```
1 | @FunctionalInterface
2 | public interface Supplier<T> {
3 |     T get();
```

3.4.2 Chain链式调用

```
public class CompletableFutureAllDemo {

    public static void main(String[] args) {
        Student student = new Student();

        student.setId(1);
        student.setStudentName("z3");
        student.setMajor("cs");

        student.setId(12).setStudentName("li4").setMajor("english");
    }
}

@AllArgsConstructor
@NoArgsConstructor
@Data
@Accessors(chain = true)
class Student {
    private Integer id;
    private String studentName;
    private String major;
}
```

3.4.3 join和get对比

```

CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> {
    return "hello 1234";
});

/*System.out.println(completableFuture.get());*/
System.out.println(completableFuture.join());

```

get()会在编译前要求异常处理，而join不要求

3.4.4 业务分析

1. 需求说明：

- 同一款产品，同时搜索出同款产品在各大电商平台的售价
- 同一款产品，同时搜索出本产品在同一个电商平台下，各个入驻卖家售价是多少

2. 输出返回：

- 出来结果希望是同款产品在不同地方的价格清单列表，返回一个 `List<String>`
- 《mysql》 in jd price is 88.05
 《mysql》 in dangdang price is 86.11

3. 解决方案：

1. step by step
2. all in

3.4.5 Code

```

public class CompletableFutureMallDemo {

    static List<NetMall> list = Arrays.asList(
        new NetMall("jd"),
        new NetMall("dangdang"),
        new NetMall("taobao")
    );

    /**
     * step by step
     * @param list
     * @param productName
     * @return
     */
    public static List<String> getPrice(List<NetMall> list, String productName)
    {
        return list.stream().map(netMall -> String.format(productName + " in %s
price is %.2f",
            netMall.getNetMallName(),
            netMall.calcPrice(productName))).collect(Collectors.toList());
    }

    /**
     * all in
     * @param list
     * @param productName
     * @return
     */

```

```

    public static List<String> getPriceByCompletableFuture(List<NetMall> list,
String productName) {
    return list.stream().map(netMall -> CompletableFuture.supplyAsync(() ->
String.format(productName + "in %s price is %.2f",
    netMall.getNetMallName(),
    netMall.calcPrice(productName))))
    .collect(Collectors.toList())
    .stream()
    .map(s -> s.join())
    .collect(Collectors.toList());
}

/**
 * 输出结果:
 * mysqlin jd price is 109.02
 * mysqlin dangdang price is 109.92
 * mysqlin taobao price is 110.56
 * ----constTime:3117毫秒
 * -----
 * mysqlin jd price is 109.50
 * mysqlin dangdang price is 109.85
 * mysqlin taobao price is 109.20
 * ----constTime:1013毫秒
 */
public static void main(String[] args) {
    long startTime = System.currentTimeMillis();
    List<String> list1 = getPrice(list, "mysql");
    for (String element : list1) {
        System.out.println(element);
    }
    long endTime = System.currentTimeMillis();
    System.out.println("----constTime:" + (endTime - startTime) + "毫秒");

    System.out.println("-----");

    long startTime2 = System.currentTimeMillis();
    List<String> list2 = getPriceByCompletableFuture(list, "mysql");
    for (String element : list2) {
        System.out.println(element);
    }
    long endTime2 = System.currentTimeMillis();
    System.out.println("----constTime:" + (endTime2 - startTime2) + "毫秒");
}
}

class NetMall {

    private String netMallName;

    public String getNetMallName() {
        return netMallName;
    }

    public NetMall(String netMallName) {
        this.netMallName = netMallName;
    }

    public double calcPrice(String productName) {

```

```

    // 暂停1秒钟线程
    try { TimeUnit.SECONDS.sleep(1); }catch (InterruptedException e){
e.printStackTrace();
    return ThreadLocalRandom.current().nextDouble() * 2 +
productName.charAt(0);
}
}

```

3.5 CompletableFuture常用方法

- thenRun: 任务A执行完执行B，并且B不需要A的结果
- thenAccept: 任务A执行完执行B，B需要A的结果，但是任务B无返回值
- thenApply: 任务A执行完执行B，B需要A的结果，同时任务B有返回值

```

System.out.println(CompletableFuture.supplyAsync(() -> "resultA").thenRun(() ->
{}).join());
System.out.println();
System.out.println(CompletableFuture.supplyAsync(() -> "resultA").thenAccept(r ->
System.out.println(r)).join());
System.out.println();
System.out.println(CompletableFuture.supplyAsync(() -> "resultA").thenApply(r ->
r + "resultB").join());

null
resultA
null

resultAresultB

```

3.5.1 获得结果和触发计算 (get/join)

获取结果：

- `public T get()`
- `public T get(long timeout, TimeUnit unit)`
- `public T join()`
- `public T getNow(T valueIfAbsent)`

```

public class CompletableFutureAPIDemo {

    /**
     * 输出结果:
     * xxx
     */
    public static void main(String[] args) throws ExecutionException,
InterruptedException, TimeoutException {
        CompletableFuture<String> completableFuture =
CompletableFuture.supplyAsync(() -> {
            // 暂停1秒钟线程
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();

```

```

        }
        return "abc";
    });

//    System.out.println(completableFuture.get());
//    System.out.println(completableFuture.get(2L, TimeUnit.SECONDS));
//    System.out.println(completableFuture.join());
    System.out.println(completableFuture.getNow("xxx"));
}
}

```

主动触发计算:

- `public boolean complete(T value)`

是否打断get/join方法立即返回括号值

```
System.out.println(completableFuture.complete("completevalue") + "\t" +
completableFuture.join());
```

打断成功:

```
true    completevalue
```

打断失败:

```
false   abc
```

3.5.2 对计算结果进行处理 (thenApply/handle)

thenApply/handle:

计算结果存在依赖关系，这两个线程串行化

```

public class CompletableFutureAPI2Demo {

    /**
     * 输出结果:
     * main----主线程先去忙其他任务
     * 111
     * 222
     * 333
     * ---计算结果: 6
     *
     * thenApply发生异常时，会叫停，不会继续往下走
     */
    public static void main(String[] args) {
        ExecutorService threadPool = Executors.newFixedThreadPool(3);

        CompletableFuture.supplyAsync(() -> {
            // 暂停1秒钟线程
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("111");
            return 1;
        }).thenApply(result -> {
            System.out.println("222");
            return result + 2;
        }).thenApply(result -> {
            System.out.println("333");
            return result + 3;
        }).thenApply(result -> {
            System.out.println("444");
            return result + 4;
        }).thenApply(result -> {
            System.out.println("555");
            return result + 5;
        }).thenApply(result -> {
            System.out.println("666");
            return result + 6;
        }).thenApply(result -> {
            System.out.println("777");
            return result + 7;
        }).thenApply(result -> {
            System.out.println("888");
            return result + 8;
        }).thenApply(result -> {
            System.out.println("999");
            return result + 9;
        }).thenAccept(result -> {
            System.out.println("计算结果: " + result);
        });
    }
}

```

```

        }, threadPool).handle((f, e) -> {
            int i = 10/0;
            System.out.println("222");
            return f+2;
        }).handle((f, e) -> {
            System.out.println("333");
            return f+3;
        }).whenComplete((v, e) -> {
            if (e == null) {
                System.out.println("---计算结果: " + v);
            }
        }).exceptionally(e -> {
            e.printStackTrace();
            System.out.println(e.getMessage());
            return null;
        });
    });

    System.out.println(Thread.currentThread().getName() + "----主线程先去忙其他任务");
}

}
}

```

exceptionally —————→ *try/catch*

```

graph TD
    A[whenComplete] --> B[try/finally]
    C[handle] --> B[try/finally]

```

3.5.3 对计算结果进行消费（Consumer函数式接口）

接收任务的处理结果，并消费处理，无返回结果

```

public class CompletableFutureAPI3Demo {

    public static void main(String[] args) {
        CompletableFuture.supplyAsync(() -> {
            return 1;
        }).thenApply(f -> {
            return f+2;
        }).thenApply(f -> {
            return f+3;
        }).thenAccept(System.out::println);
    }
}

```

3.5.4 对计算速度进行选用

```
public class CompletableFutureFastDemo {  
  
    public static void main(String[] args) {  
  
        CompletableFuture<String> playA = CompletableFuture.supplyAsync(() -> {  
            System.out.println("A come in");  
            try {  
                TimeUnit.SECONDS.sleep(2);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            return "playA";  
        });  
  
        CompletableFuture<String> playB = CompletableFuture.supplyAsync(() -> {  
            System.out.println("B come in");  
            try {  
                TimeUnit.SECONDS.sleep(3);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            return "playB";  
        });  
  
        CompletableFuture<String> result = playA.applyToEither(playB, f -> {  
            return f + " is winner";  
        });  
  
        System.out.println(Thread.currentThread().getName() + "\t----" +  
result.join());  
  
    }  
}
```

```
A come in  
B come in  
main      ----playA is winner
```

3.5.5 对计算结果进行合并

两个CompletionStage任务都完成后，最终能把两个任务的结果一起交给thenCombine来处理

分解版

```
public class CompletableFutureCombineDemo {  
  
    public static void main(String[] args) {  
        CompletableFuture<Integer> completableFuture1 =  
CompletableFuture.supplyAsync(() -> {  
            System.out.println(Thread.currentThread().getName() + "\t---启动");  
            try {  
                TimeUnit.SECONDS.sleep(1);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            return 1;  
        }).thenCombine(CompletableFuture.supplyAsync(() -> {  
            System.out.println(Thread.currentThread().getName() + "\t---执行");  
            try {  
                TimeUnit.SECONDS.sleep(2);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            return 2;  
        }), (x, y) -> {  
            System.out.println(Thread.currentThread().getName() + "\t---结果");  
            return x + y;  
        });  
  
        completableFuture1.join();  
    }  
}
```

```

        e.printStackTrace();
    }
    return 10;
});

CompletableFuture<Integer> completableFuture2 =
CompletableFuture.supplyAsync(() -> {
    System.out.println(Thread.currentThread().getName() + "\t----启动");
    try {
        TimeUnit.SECONDS.sleep(2);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return 20;
});

CompletableFuture<Integer> result =
completableFuture1.thenCombine(completableFuture2, (x, y) -> {
    System.out.println("----开始两个结果合并");
    return x + y;
});

System.out.println(result.join());
}
}

```

合并版

```

CompletableFuture<Integer> completableFuture = CompletableFuture.supplyAsync(() -> {
    // 暂停1秒钟线程
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return 10;
}).thenCombine(CompletableFuture.supplyAsync(() -> {
    // 暂停1秒钟线程
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return 20;
}), (x, y) -> {
    return x + y;
}).thenCombine(CompletableFuture.supplyAsync(() -> {
    // 暂停1秒钟线程
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return 30;
}), (a, b) -> {
    return a + b;
})

```

```
});  
  
System.out.println(completableFuture.join());
```

3.5.6 线程池运行时选择

```
public class CompletableFuturewithThreadPoolDemo {  
  
    public static void main(String[] args) {  
        ExecutorService threadPool = Executors.newFixedThreadPool(5);  
  
        try {  
            CompletableFuture<Void> completableFuture =  
completableFuture.supplyAsync(() -> {  
                try {  
                    TimeUnit.MILLISECONDS.sleep(20);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                System.out.println("1号任务" + "\t" +  
Thread.currentThread().getName());  
                return "abcd";  
            }, threadPool).thenRunAsync(() -> {  
                try {  
                    TimeUnit.MILLISECONDS.sleep(20);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                System.out.println("2号任务" + "\t" +  
Thread.currentThread().getName());  
            }).thenRun(() -> {  
                try {  
                    TimeUnit.MILLISECONDS.sleep(20);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                System.out.println("3号任务" + "\t" +  
Thread.currentThread().getName());  
            }).thenRun(() -> {  
                try {  
                    TimeUnit.MILLISECONDS.sleep(20);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                System.out.println("4号任务" + "\t" +  
Thread.currentThread().getName());  
            });  
  
            System.out.println(completableFuture.get(2L, TimeUnit.SECONDS));  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            threadPool.shutdown();  
        }  
    }  
}
```

1 没有传入自定义线程池，都用默认线程池ForkJoinPool;

2 传入了一个自定义线程池，

如果你执行第一个任务的时候，传入了一个自定义线程池：

调用thenRun方法执行第二个任务时，则第二个任务和第一个任务是共用同一个线程池。

调用thenRunAsync执行第二个任务时，则第一个任务使用的是你自己传入的线程池，第二个任务使用的是ForkJoin线程池

3 备注

有可能处理太快，系统优化切换原则，直接使用main线程处理

其它如: thenAccept和thenAcceptAsync, thenApply和thenApplyAsync等，它们之间的区别也是同理

4. 说说Java“锁”事（上半）

4.1 乐观锁和悲观锁

悲观锁

适合写操作多的场景，先加锁可以保证写操作时数据正确

synchronized和lock都是悲观锁

乐观锁

适合读操作多的场景，不加锁的特点能够使其读操作的性能大幅提升

判断规则

1. 版本号机制Version
2. CAS算法，Java原子类种的递增操作就通过CAS自旋实现

4.2 通过8种情况演示锁案例

案例code

```
/**  
 * @Author cxl  
 * @Date 21/5/2023 16:27  
 * @ClassReference: com.cxl.juc.锁.Lock8Demo  
 * @Description:  
 *   * 1.标准访问，先打印短信还是邮件（锁当前对象）  
 *   * -----sendEmail  
 *   * -----sendSMS  
 *   *  
 *   * 2.停3秒在短信方法内，先打印短信还是邮件（锁当前对象）  
 *   * -----sendEmail  
 *   * -----sendSMS  
 *   *  
 *   * 3.新增普通的hello方法，是先打邮件还是hello  
 *   * -----getHello  
 *   * -----sendEmail
```

```

* *
* * 4. 现在有两部手机,先打印短信还是邮件 (两个对象不是同一把锁)
* * -----sendSMS
* * -----sendEmail
* *
* * 5. 两个静态同步方法,1部手机,先打印短信还是邮件 (锁当前类)
* * -----sendEmail
* * -----sendSMS
* *
* * ***6. 两个静态同步方法,2部手机,先打印短信还是邮件 (锁当前类)
* * -----sendEmail
* * -----sendSMS
* *
* * 7. 1个静态同步方法,1普通同步方法,1部手机,先打印短信还是邮件
* * -----sendSMS
* * -----sendEmail
* *
* * 8. 1个静态同步方法,1普通同步方法,2部手机,先打印短信还是邮件
* * -----sendSMS
* * -----sendEmail
*
* 1-2
* 一个对象里面如果有多个synchronized方法, 某一个时刻内, 只有一个吸纳成去调用其中的一个
synchronized方法了
* 其它的线程都只能等待, 换句话说, 某一个时刻内, 只能有唯一的一个线程去访问这些synchronized
方法
* 锁的是当前对象this, 被锁定后, 其它的线程都不能进入到当前对象的其它的synchronized方法
*
* 3-4
* 加个普通方法后发现和同步锁无关
* 换成两个对象后, 不是同一把锁了, 情况立刻变化
*
* 5-6
* 对于普通同步方法, 锁的是当前实例对象, 通常指this, 具体的一部部手机, 所有的普通同步方法用的
都是同一把锁 -> 实例对象本身
* 对于静态同步方法, 锁的是当前类的Class对象, 如Phone, class唯一的一个模板
* 对于同步方法块, 锁的是 synchronized 括号中的对象
*
* 7-8
* 当一个线程试图访问同步代码时它首先必须得到锁工正常退出或抛出异常时必须释放锁。
* 所有的普通同步方法用的都是同一把锁一实例对象本身, 就是new出来的具体实例对象本身, 本类this
* 也就是说如果一个实例对象的普通同步方法获取锁后, 该实例对象的其他普通同步方法必须等待获取锁的
方法释放锁后才能获取锁。
*
* 所有的静态同步方法用的也是同一把锁一类对象本身, 就是我们说过的唯一模板class
* 具体实例对象this和唯一模板class, 这两把锁是两个不同的对象, 所以静态同步方法与普通同步方法
之间是不会有竟态条件的
* 但是一旦一个静态同步方法获取锁后, 其他的静态同步方法都必须等待该方法释放锁后才能获取锁。
*/
class Phone {

    public static synchronized void sendEmail() {
        // 暂停3秒钟线程
//        try { TimeUnit.SECONDS.sleep(3); }catch (InterruptedException e){
        e.printStackTrace();
        System.out.println("----sendEmail");
    }
}

```

```

public synchronized void sendSMS() {
    System.out.println("----sendSMS");
}

public void hello() {
    System.out.println("-----hello");
}
}

public class Lock8Demo {

public static void main(String[] args) {
    Phone phone = new Phone();
    Phone phone2 = new Phone();

    new Thread(() -> {
        phone.sendEmail();
    }, "a").start();

    try { TimeUnit.MILLISECONDS.sleep(200); } catch (InterruptedException e){
        e.printStackTrace();
    }

    new Thread(() -> {
        phone.sendSMS();
        //          phone.hello();
    }, "b").start();
}
}

```

4.3 synchronized有三种应用方式

JDK源码

- By executing a synchronized instance method of that object.
- By executing the body of a synchronized statement that synchronizes on the object.
- For objects of type Class, by executing a synchronized static method of that class.

8种锁的案例实际体现在3个地方

- 作用于实例方法，当前实例加锁，进入同步代码前要获得当前实例的锁
- 作用于代码块，对括号里配置的对象加锁
- 作用于静态方法，当前类加锁，进去同步代码前要获得当前类对象的锁

4.4 从字节码角度分析synchronized实现

反编译

```
javap -c .\Lock8Demo.class
```

4.4.1 synchronized同步代码块

```
javap -c .\LockSyncDemo.class
```

```

public void m1();
Code:
  0: aload_0
  1: getfield      #3                  // Field object:Ljava/lang/Object;
  4: dup
  5: astore_1
  6: monitorenter
  7: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
 10: ldc           #5                  // String ----hello synchronized code block
 12: invokevirtual #6                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
 15: aload_1
 16: monitorexit
 17: goto         25
 20: astore_2
 21: aload_1
 22: monitorexit
 23: aload_2
 24: athrow
 25: return
Exception table:
  from   to    target type
    7    17    20    any
   20    23    20    any

```

第二个monitorexit是为了在异常时将锁给释放

4.4.2 synchronized普通同步方法

```
javap -v .\LockSyncDemo.class
```

```

public synchronized void m2();
descriptor: ()V
flags: ACC_PUBLIC, ACC_SYNCHRONIZED
Code:
  stack=2, locals=1, args_size=1
  0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
  3: ldc           #3                  // String ----hello synchronized m2
  5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
  8: return
LineNumberTable:
  line 22: 0
  line 23: 8
LocalVariableTable:
  Start  Length  Slot  Name   Signature
    0       9      0  this   Lcom/cxl/juc/锁 /LockSyncDemo;

```

4.4.3 synchronized静态同步方法

```

public static synchronized void m3();
descriptor: ()V
flags: ACC_PUBLIC, ACC_STATIC, ACC_SYNCHRONIZED
Code:
  stack=2, locals=0, args_size=0
  0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
  3: ldc           #3                  // String ----hello synchronized m3
  5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
  8: return
LineNumberTable:
  line 22: 0
  line 23: 8

```

4.5 反编译synchronized锁是什么

为什么每一个对象都可以成为锁

每一个对象天生都带着一个对象监视器

每一个被锁住的对象都会跟Monitor关联起来

4.5 公平锁和非公平锁

锁	定义
公平锁	是指多个线程按照申请锁的顺序来获取锁，这里类似排队买票，先来的人先买后来的人在队尾排着，这是公平的 Lock lock = new ReentrantLock(true); //true 表示公平锁,先来先得
不公平锁	是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁，在高并发环境下，有可能造成优先级翻转或者饥饿的状态(某个线程一直得不到锁) Lock lock = new ReentrantLock(false); //false 表示非公平锁，后来的也可能先获得锁 lock = new ReentrantLock(); //默认非公平锁

为什么默认是非公平锁？

1. 恢复挂起的线程到真正锁的获取还是有时间差的，从开发人员来看这个时间微乎其微，但是从CPU的角度来看，这个时间差存在的还是很明显的。
所以非公平锁能更充分的利用CPU的时间片，尽量减少CPU空闲状态时间。
2. 使用多线程很重要的考量点是线程切换的开销，当采用非公平锁时，**当1个线程请求锁获取同步状态，然后释放同步状态，所以刚释放锁的线程在此刻再次获取同步状态的概率就变得非常大，所以就减少了线程的开销。**

4.6 可重入锁

可重入锁又名递归锁

是指在同一个线程在外层方法获取锁的时候，再进入该线程的内层方法会自动获取锁(前提，锁对象得是同一个对象)，不会因为之前已经获取过还没释放而阻塞。

如果是1个有 synchronized 修饰的递归调用方法，程序第2次进入被自己阻塞了岂不是天大的笑话，出现了作茧自缚。

所以Java中ReentrantLock和synchronized都是可重入锁，可重入锁的一个优点是可一定程度避免死锁。

可重入锁种类

- 隐式锁(即synchronized关键字使用的锁)默认是可重入锁
 - 指的是可重复可递归调用的锁，在外层使用锁之后，在内层仍然可以使用，并且不发生死锁，这样的锁就叫做可重入锁。
 - 简单的来说就是在synchronized修饰的方法或代码块的内部调用本类的其他synchronized修饰的方法或代码块时，是永远可以得到锁的

```
public class RetryLockDemo {  
  
    public synchronized void m1() {  
        System.out.println(Thread.currentThread().getName() + "\t ----come in");  
        m2();  
        System.out.println(Thread.currentThread().getName() + "\t ----come  
end");  
    }  
  
    public synchronized void m2() {  
        System.out.println(Thread.currentThread().getName() + "\t ----come in");  
    }  
}
```

```

        m3();
    }

    public synchronized void m3() {
        System.out.println(Thread.currentThread().getName() + "\t ----come in");
    }

    public static void main(String[] args) {

        RetryLockDemo retryLockDemo = new RetryLockDemo();

        new Thread(() -> {
            retryLockDemo.m1();
        }, "t1").start();
    }

    private static void reEntryM1(Object object) {
        new Thread(() -> {
            synchronized (object) {
                System.out.println(Thread.currentThread().getName() + "\t ----外层调用");
                synchronized (object) {
                    System.out.println(Thread.currentThread().getName() + "\t --中层调用");
                    synchronized (object) {
                        System.out.println(Thread.currentThread().getName() +
                                "\t ----内层调用");
                    }
                }
            }
        }, "t1").start();
    }
}

```

每个锁对象拥有一个锁计数器和一个指向持有该锁的线程的指针。

当执行monitorenter时，如果目标锁对象的计数器为零，那么说明它没有被其他线程所持有，Java虚拟机会将该锁对象的持有线程设置为当前线程，并且将其计数器加1。

在目标锁对象的计数器不为零的情况下，如果锁对象的持有线程是当前线程，那么Java虚拟机可以将其计数器加1，否则需要等待，直至持有线程释放该锁。

当执行monitorexit时，Java虚拟机则需将锁对象的计数器减1。计数器为零代表锁已被释放

- 显式锁(即Lock)也有ReentrantLock这样的可重入锁

```

public static void main(String[] args) {
    new Thread(() -> {
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "\t ---come in
外层调用");
            try {
                lock.lock();
                System.out.println(Thread.currentThread().getName() + "\t ---come in内层调用");
            } finally {

```

```

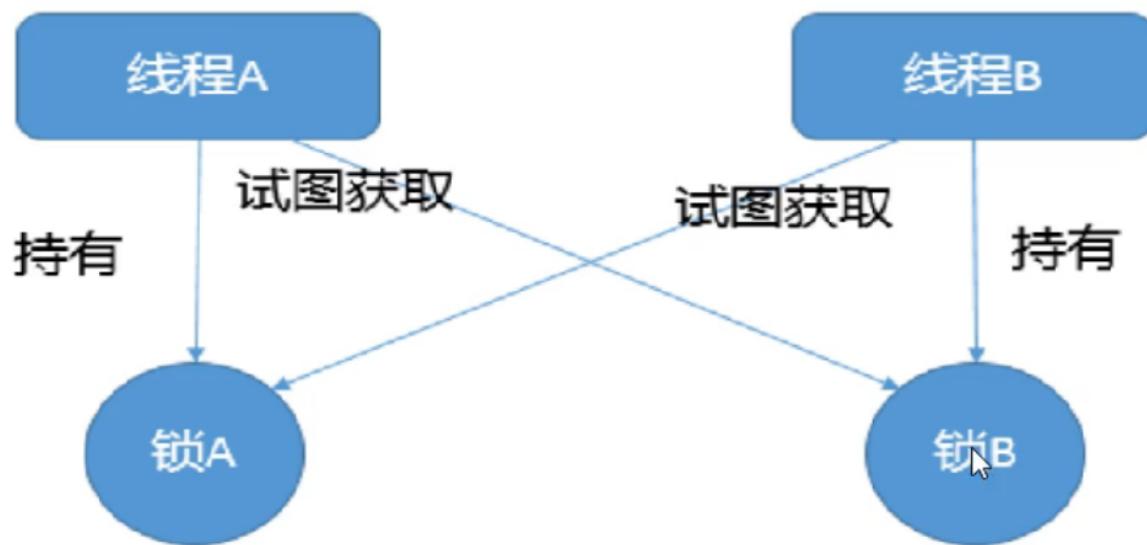
        lock.unlock();
    }
} finally {
    lock.unlock();
}
}, "t1").start();

new Thread() -> {
    lock.lock();
    try {
        System.out.println(Thread.currentThread().getName() + "\t ---come in
外层调用");
    } finally {
        lock.unlock();
    }
}, "t2").start();
}

```

4.7 死锁及排查

死锁是指两个或两个以上的线程在执行过程中,因争夺资源而造成的一种互相等待的现象,若无外力干涉那它们都将无法推进下去,如果系统资源充足,进程的资源请求都能够得到满足,死锁出现的可能性就很低,否则就会因争夺有限的资源而陷入死锁。



产生死锁的原因:

1. 系统资源不足
2. 进程运行过程中推进顺序不合适
3. 资源分配不当

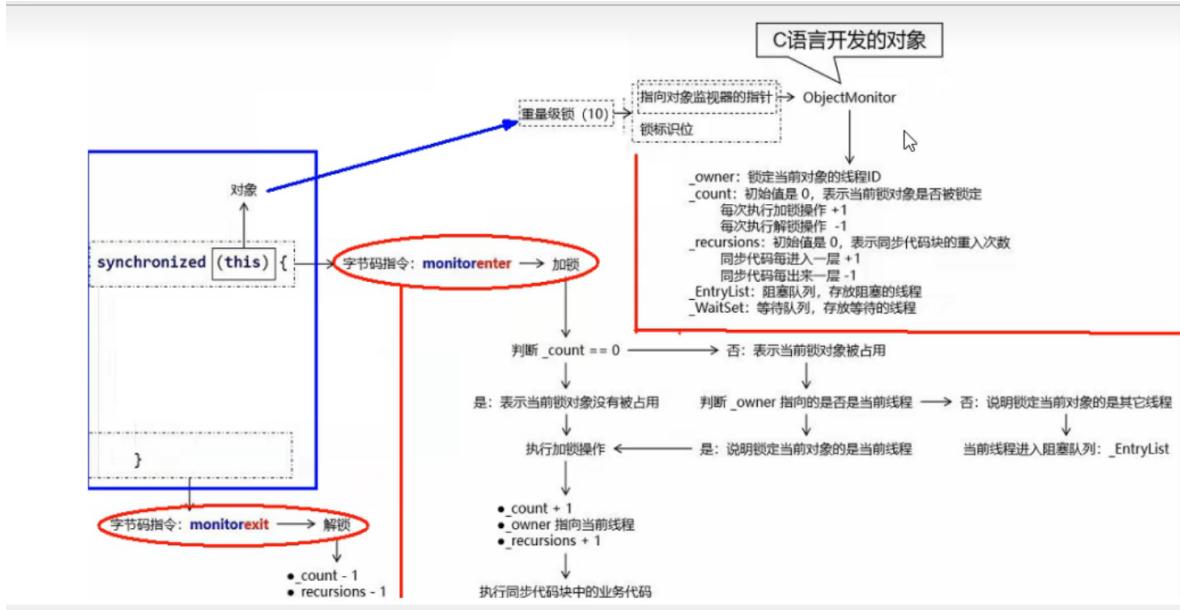
验证是否是死锁:

1. jps (类似linux ps -ef)
2. jstack (jvm自带堆栈跟踪工具)

或者使用运行jconsole

总结

指针指向monitor对象(也称为管程或监视器锁)的起始地址。每个对象都存在着一个monitor与之关联，当一个 monitor 被某个线程持有后，它便处于锁定状态。在Java虚拟机(HotSpot)中，monitor是由ObjectMonitor实现的，其主要数据结构如下(位于HotSpot虚拟机源码ObjectMonitor.hpp文件，C++实现的)



5. LockSupport与线程中断

LockSupport

java.util.concurrent.locks

Interfaces

Condition

Lock

ReadWriteLock

Classes

AbstractOwnableSynchronizer

AbstractQueuedLongSynchronizer

AbstractQueuedSynchronizer

LockSupport

ReentrantLock

ReentrantReadWriteLock

ReentrantReadWriteLock.ReadLock

ReentrantReadWriteLock.WriteLock

StampedLock

线程中断

void	interrupt()	
		Interrupts this thread.
static boolean	interrupted()	
		Tests whether the current thread has been interrupted.
boolean	isAlive()	
		Tests if this thread is alive.
boolean	isDaemon()	
		Tests if this thread is a daemon thread.
boolean	isInterrupted()	
		Tests whether this thread has been interrupted.

5.1 线程中断机制

一个线程不应该由其他线程来强制中断或停止，而是应该由线程自己自行停止，自己来决定自己的命运。所以，Thread.stop, Thread.suspend, Thread.resume 都已经被废弃了。

在Java中没有办法立即停止一条线程，然而停止线程却显得尤为重要，如取消一个耗时操作。

因此，Java提供了一种用于停止线程的协商机制——中断，也即中断标识协商机制。

中断只是一种协作协商机制，Java没有给中断增加任何语法，中断的过程完全需要程序员自己实现。

若要中断一个线程，你需要手动调用该线程的interrupt方法，该方法也仅仅是将线程对象的中断标识设成true；

接着你需要自己写代码不断地检测当前线程的标识位，如果为true，表示别的线程请求这条线程中断，此时究竟该做什么需要你自己写代码实现。

每个线程对象中都有一个中断标识位，用于表示线程是否被中断；该标识位为true表示中断，为false表示未中断；通过调用线程对象的interrupt方法将该线程的标识位设为true；可以在别的线程中调用，也可以在自己的线程中调用。

5.2 中断的相关API方法之三大方法说明

方法	含义
public void interrupt()	实例方法，中断此线程 Just to set the interrupt flag 实例方法interrupt()仅仅是设置线程的中断状态为true，发起一个协商而不会立刻停止线程
public static boolean interrupted()	静态方法，Thread.interrupted(); 判断线程是否被中断并清除当前中断状态 这个方法做了两件事 1返回当前线程的中断状态，测试当前线程是否已被中断 2将当前线程的中断状态清零并重新设为false，清除线程的中断状态
public boolean isInterrupted()	实例方法， 判断当前线程是否被中断（通过检查中断标志位）

5.3 大厂面试题中断机制考点

5.3.1 如何停止中断运行中的线程？

1. 通过一个volatile变量实现

```
static volatile boolean isStop = false;

private static void m1_volatile() {
    new Thread(() -> {
        while (true) {
            if (isStop) {
                System.out.println(Thread.currentThread().getName() +
"\t isStop被修改为true，程序停止");
                break;
            }
            System.out.println("----hello volatile");
        }
    }, "t1").start();

    try { TimeUnit.MILLISECONDS.sleep(20); }catch (InterruptedException e){ e.printStackTrace(); }

    new Thread(() -> {
        isStop = true;
    }, "t2").start();
}
```

2. 通过AtomicBoolean

```
static volatile boolean isStop = false;

static AtomicBoolean atomicBoolean = new AtomicBoolean(false);

public static void main(String[] args) {

    m2_atomicBoolean();
}

private static void m2_atomicBoolean() {
    new Thread(() -> {
        while (true) {
            if (atomicBoolean.get()) {
                System.out.println(Thread.currentThread().getName() + "\t
isStop被修改为true, 程序停止");
                break;
            }
            System.out.println("----hello volatile");
        }
    }, "t1").start();

    try { TimeUnit.MILLISECONDS.sleep(20); }catch (InterruptedException e){
e.printStackTrace(); }

    new Thread(() -> {
        atomicBoolean.set(true);
    }, "t2").start();
}
```

3. 通过Thread类自带的中断api实例方法实现

```
public static void main(String[] args) {

    Thread t1 = new Thread(() -> {
        while (true) {
            if (Thread.currentThread().isInterrupted()) {
                System.out.println(Thread.currentThread().getName() + "\t
isInterrupted()被修改为true, 程序停止");
                break;
            }
            System.out.println("----hello interrupt api");
        }
    }, "t1");

    t1.start();

    try { TimeUnit.MILLISECONDS.sleep(20); }catch (InterruptedException e){
e.printStackTrace(); }

    // t2向t1发出协商, 将t1的中断标志位设为true希望t1停下来
    new Thread(() -> {
        t1.interrupt();
    }, "t2").start();
}
```

具体来说，当对一个线程，调用 interrupt() 时：

1. 如果线程处于正常活动状态，那么会将该线程的中断标志设置为 true
被设置中断标志的线程将继续正常运行，不受影响。
所以， interrupt() 并不能真正的中断线程，需要被调用的线程自己进行配合才行。
2. 如果线程处于被阻塞状态(例如处于sleep, wait, join 等状态)，在别的线程中调用当前线程对象的 interrupt方法，那么线程将立即退出被阻塞状态，并抛出一个InterruptedException异常。

5.3.2 当前线程的中断表示为true，是不是线程就立刻停止？

线程不会立刻停止

```
public static void main(String[] args) {  
    // 实例方法interrupt()仅仅是设置线程的中断状态位为true，不会停止线程  
  
    Thread t1 = new Thread(() -> {  
        for (int i = 0; i < 300; i++) {  
            System.out.println("----" + i);  
        }  
        System.out.println("t1线程调用interrupt()后的中断标识02: " +  
Thread.currentThread().isInterrupted()); // true  
    }, "t1");  
  
    t1.start();  
  
    System.out.println("t1线程默认的中断标志: " + t1.isInterrupted()); // false  
  
    try { TimeUnit.MILLISECONDS.sleep(2); } catch (InterruptedException e){  
e.printStackTrace();}  
  
    t1.interrupt(); // true  
    System.out.println("t1线程调用interrupt()后的中断标识01: " +  
t1.isInterrupted()); // false  
  
    // 暂停1秒钟线程  
    try { TimeUnit.SECONDS.sleep(2); } catch (InterruptedException e){  
e.printStackTrace();}  
    System.out.println("t1线程调用interrupt()后的中断标识03: " +  
t1.isInterrupted()); // false  
}
```

*sleep 方法抛出 InterruptedException后，中断标识也被清空置为 false，我们在catch 没有通过调用 thnerrupt() 方法再次将中断标识置为 true，这就导致无限循环了

```
/**  
 * @Author cxl  
 * @Date 22/5/2023 20:04  
 * @ClassReference: com.cxl.juc.LockSupport与线程中断.InterruptDemo3  
 * @Description:  
 *  
 * 1 中断标志位默认false  
 * 2 t2 ---> t1 发出了中断协商，t2调用t1.interrupt()，中断标志位true  
 * 3 中断标志位true，正常情况，程序停止  
 * 4 中断标志位true，异常情况，InterruptedException，中断状态将被清除，并将收到  
InterruptedException。中断标志位false
```

```

    * 导致死循环
    * 5 在catch中，需要再次给中断标志位设置为true，2此调用停止程序才OK
    */
public class InterruptDemo3 {

    public static void main(String[] args) {

        Thread t1 = new Thread(() -> {
            while (true) {
                if (Thread.currentThread().isInterrupted()) {
                    System.out.println(Thread.currentThread().getName() + "\t 中
断标志位: " +
                            Thread.currentThread().isInterrupted() + " 程序停
止");
                    break;
                }

                try {
                    TimeUnit.MILLISECONDS.sleep(200);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt(); // 为什么要在异常处再调用一
次?
                    e.printStackTrace();
                }
                System.out.println("-----hello InterruptDemo3");
            }
        }, "t1");

        t1.start();

        // 暂停1秒钟线程
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        new Thread(() -> {
            t1.interrupt();
        }, "t2").start();
    }
}

```

5.3.3 静态方法Thread.interrupted(), 谈谈你的理解

```

/**
 * @Author cxl
 * @Date 23/5/2023 08:54
 * @ClassReference: com.cxl.juc.LockSupport与线程中断.InterruptDemo4
 * @Description:
 * main    false
 * main    false
 * ----1
 * ----2
 * main    true
 * main    false
 */

```

```
public class InterruptDemo4 {  
  
    public static void main(String[] args) {  
  
        System.out.println(Thread.currentThread().getName() + "\t" +  
Thread.interrupted());  
        System.out.println(Thread.currentThread().getName() + "\t" +  
Thread.interrupted());  
        System.out.println("----1");  
        Thread.currentThread().interrupt(); // true  
        System.out.println("----2");  
        System.out.println(Thread.currentThread().getName() + "\t" +  
Thread.interrupted());  
        System.out.println(Thread.currentThread().getName() + "\t" +  
Thread.interrupted());  
  
    }  
}
```

5.4 LockSupport

The screenshot shows the Java API documentation for the package `java.util.concurrent.locks`. The `LockSupport` class is highlighted with a red box and a cursor arrow pointing to it. Other visible classes include `AbstractOwnableSynchronizer`, `AbstractQueuedLongSynchronizer`, `AbstractQueuedSynchronizer`, `ReentrantLock`, `ReentrantReadWriteLock`, `ReentrantReadWriteLock.ReadLock`, `ReentrantReadWriteLock.WriteLock`, and `StampedLock`.

LockSupport是用来创建锁和其它同步类的基本线程阻塞原语

5.4.1 等待唤醒机制

3种让线程等待和唤醒的方法

- 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
- 使用JUC包中Condition的await()方法让线程等待，使用signal()方法唤醒线程
- LockSupport类可以阻塞当前线程以及唤醒指定被阻塞的线程

5.4.2 Object类的wait和notify方法实现线程等待和唤醒

```
/**  
 * 异常1: wait和notify方法如果都去掉同步代码块会抛IllegalMonitorStateException  
 *  
 * 异常2: 将notify放在wait前面, 程序无法执行, 无法唤醒  
 */  
private static void syncWaitNotify() {  
    Object objectLock = new Object();  
  
    new Thread(() -> {  
        // 暂停1秒钟线程  
        try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        synchronized (objectLock) {  
            System.out.println(Thread.currentThread().getName() + "\t ----  
come in");  
            try {  
                objectLock.wait();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            System.out.println(Thread.currentThread().getName() + "\t ----被  
唤醒");  
        }  
    }, "t1").start();  
  
    //      try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e) {  
    e.printStackTrace(); }  
  
    new Thread(() -> {  
        synchronized (objectLock) {  
            objectLock.notify();  
            System.out.println(Thread.currentThread().getName() + "\t ----发  
出通知");  
        }  
    }, "t2").start();  
}
```

5.4.3 Condition接口中await后signal方法实现线程的等待和唤醒

```
/**  
 * 异常1: 如果去掉lock和unlock也会抛IllegalMonitorStateException  
 *  
 * 异常2: 将signal放在await前面, 程序无法执行, 无法唤醒  
 */  
private static void lockAwaitSignal() {  
    Lock lock = new ReentrantLock();  
    Condition condition = lock.newCondition();
```

```

new Thread(() -> {
    // 暂停1秒钟线程
    try { TimeUnit.SECONDS.sleep(1); }catch (InterruptedException e){
        e.printStackTrace();
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "\t ---\n" +
                    "come in");
            condition.await();
            System.out.println(Thread.currentThread().getName() + "\t ---被唤醒");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }, "t1").start();

    // 暂停1秒钟线程
    // try { TimeUnit.SECONDS.sleep(1); }catch (InterruptedException e){
    e.printStackTrace(); }

    new Thread(() -> {
        lock.lock();
        try {
            condition.signal();
            System.out.println(Thread.currentThread().getName() + "\t ---发出通知");
        } finally {
            lock.unlock();
        }
    }, "t2").start();
}

```

5.4.4 上述两个对象Object和Condition使用的限制条件

- 线程先要获得并持有锁，必须在锁块 (synchronized或lock) 中
- 必须要先等待后唤醒，线程才能够被唤醒

5.4.5 LockSupport类中的park等待和unpark唤醒

LockSupport是通过park()和unpark(thread)方法来实现阻塞和唤醒线程的操作

```
public class LockSupport
extends Object
```

Basic thread blocking primitives for creating locks and other synchronization classes.

This class associates, with each thread that uses it, a permit (in the sense of the Semaphore class). A call to park will return immediately if the permit is available, consuming it in the process; otherwise it may block. A call to unpark makes the permit available, if it was not already available. (Unlike with Semaphores though, permits do not accumulate. There is at most one.)

LockSupport是用来创建锁和其他同步类的基本线程阻塞原语。

LockSupport类使用了一种名为Permit (许可)的概念来做到阻塞和唤醒线程的功能，每个线程都有一个许可(permit)，

但与Semaphore不同的是，许可的累加上限是1。

permit许可证默认没有不能放行，所以一开始调park方法当前线程就会阻塞，直到别的线程给当前线程的发放permit， park方法才会被唤醒。

```
/***
 * 正常+无锁块要求
 * 之前错误的先唤醒后等待， LockSupport照样支持
 * @param args
 */
public static void main(String[] args) {

    Thread t1 = new Thread(() -> {
        try { TimeUnit.SECONDS.sleep(1); }catch (InterruptedException e){
            e.printStackTrace(); }
        System.out.println(Thread.currentThread().getName() + "\t ----come in");
        LockSupport.park();
        System.out.println(Thread.currentThread().getName() + "\t ----被唤醒");
    }, "t1");

    t1.start();

    new Thread(() -> {
        LockSupport.unpark(t1);
        System.out.println(Thread.currentThread().getName() + "\t ----发出通知");
    }, "t2").start();
}
```

LockSupport是一个线程阻塞工具类，所有的方法都是静态方法，可以让线程在任意位置阻塞，阻塞之后也有对应的唤醒方法。归根结底， LockSupport调用的Unsafe中的native代码。

LockSupport 提供park()和unpark()方法实现阻塞线程和解除线程阻塞的过程

LockSupport和每个使用它的线程都有一个许可(permit)关联。

每个线程都有一个相关的permit, permit最多只有一个，重复调用unpark也不会积累凭证

6. Java内存模型之JMM

因为cpu和物理主内存都存在缓存且速度不一致

CPU的运行并不是直接操作内存而是先把内存里边的数据读到缓存，而内存的读和写操作的时候就会造成不一致的问题



JVM规范中试图定义一种Java内存模型 (java Memory Model, 简称JMM) 来屏蔽掉各种硬件和操作系统的内存访问差异

实现让Java程序在各种平台下都能达到一致的内存访问效果。

6.1 Java内存模型 Java Memory Model

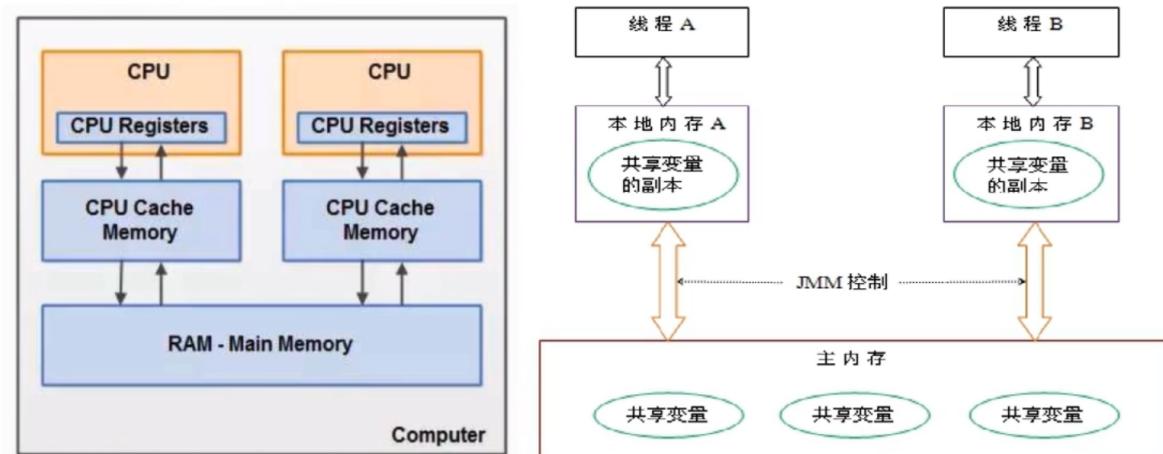
JMM (Java内存模型Java Memry Model, 简称JMM) 本身是一种抽象的概念并不真实存在它仅仅描述的是一组约定或规范，通过这组规范定义了程序中(尤其是多线程)各个变量的读写访问方式并决定一个线程对共享变量的写入何时以及如何变成对另一个线程可见，关键技术点都是围绕多线程的原子性、可见性和有序性展开的。

1. 通过JMM来实现线程和主内存之间的抽象关系
2. 屏蔽各个硬件平台和操作系统的内存访问差异以实现让Java程序在各种平台下都能达到一致的内存访问效果

6.2 三大特性

6.2.1 可见性

是指当一个线程修改了某一个共享变量的值，其他线程是否能够立即知道该变更，JMM规定了所有的变量都存储在主内存中。



系统主内存共享变量数据修改被写入的时机是不确定的，多线程并发下很可能出现“脏读”，所以每个线程都有自己的工作内存、线程自己的工作内存中保存了该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作(读取，赋值等)都必需在线程自己的工作内存中进行，而不能够直接读写主内存中的变量。不同线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成



但还是可能会产生线程脏读

- 主内存中有变量 x , 初始值为 0
- 线程 A 要将 $x=0$ 复制到自己的私有内存中, 然后更新 x 的值
- 线程 A 将更新后的 x 值回刷到主内存的时间是不固定的
- 刚好在线程 A 没有回刷 x 到主内存时, 线程 B 同样从主内存中读取 x , 此时为 0, 和线程 A一样的操作, 最后期盼的 $x=2$ 就会变成 $x=1$

6.2.2 原子性

指一个操作是不可打断的, 即多线程环境下, 操作不能被其它线程干扰

6.2.3 有序性

对于一个线程的执行代码而言, 我们总是习惯性认为代码的执行总是从上到下, 有序执行。但为了提升性能, 编译器和处理器通常会对指令序列进行**重新排序**。Java 规范规定 JVM 线程内部维持**顺序化语义**, 即只要程序的最终结果与它顺序化执行的结果相等, 那么指令的执行顺序可以与代码顺序不一致, 此过程叫**指令的重排序**。

优缺点

JVM 能根据处理器特性(CPU 多级缓存系统、多核处理器等)适当的对机器指令进行重排序, 使机器指令能更符合 CPU 的执行特性, 最大限度的发挥机器性能。但是, 指令重排可以保证串行语义一致, 但没有义务保证**多线程间的语义也一致**(即可能产生“脏读”), 简单说, 两行以上不相干的代码在执行的时候有可能先执行的不是第一条, 不见得是从上到下顺序执行, 执行顺序会被优化。



单线程环境里面确保程序最终执行结果和代码顺序执行的结果一致。

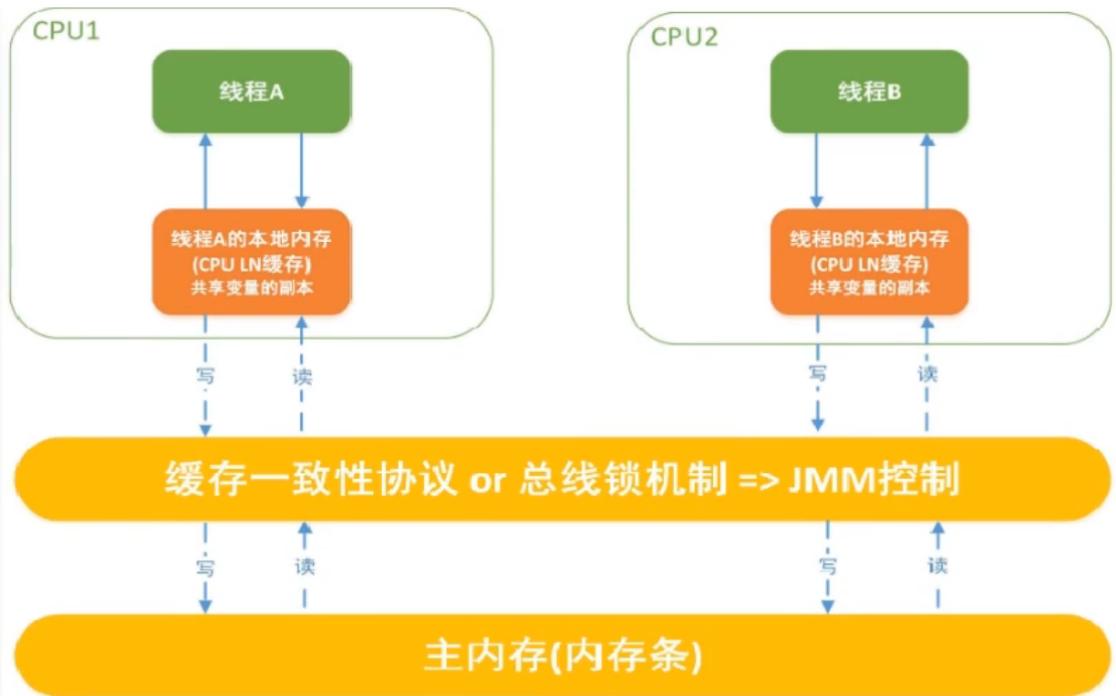
处理器在进行重排序时**必须要考虑指令之间的数据依赖性**

多线程环境中线程交替执行, 由于编译器优化重排的存在, 两个线程中使用的变量能否保证一致性是无法确定的, 结果无法预测。

6.3 多线程的读写过程

由于 JVM 运行程序的实体是线程, 而每个线程创建时 JVM 都会为其创建一个工作内存(有些地方称为成空间), 工作内存是每个线程的私有数据区域, 而 Java 内存模型中规定所有变量都存储在主内存, 主内存是共享内存区域, 所有线程都可以访问, **但线程对变量的操作(读取赋值等)必须在工作内存中进行, 首先要将变量从主内存拷贝到线程自己的工作内存空间, 然后对变量进行操作, 操作完成后将变量写回主内存**, 不能直接操作主内存中的变量, 各个线程中的工作内存中存储着主内存中的变量副本拷贝, 因

此不同的线程间无法访问对方的工作内存，线程间的通信(传值)必须通过主内存来完成，其简要访问过程如下图：



JMM定义线程和主内存之间的抽象关系

1. 线程之间的共享变量存储在主内存中(从硬件角度来说就是内存条)
2. 每个线程都有一个私有的本地工作内存，本地工作内存中存储了该线程用来读/写共享变量的副本
(从硬件角度来说就是CPU的缓存，比如寄存器、L1、L2、L3缓存等)

小总结：

- 我们定义的所有共享变量都储存在物理主内存中
- 每个线程都有自己独立的工作内存，里面保存该线程使用到的变量的副本(主内存中该变量的一份拷贝)
- 线程对共享变量所有的操作都必须先在线程自己的工作内存中进行后写回主内存，不能直接从主内存中读写(不能越级)
- 不同线程之间也无法直接访问其他线程的工作内存中的变量，线程间变量值的传递需要通过主内存来进行(同级不能相互访问)

6.4 先行发生原则之happens-before

在JMM中如果一个操作执行的结果需要对另一个操作可见性或者代码重排序，那么这两个操作之间必须存在happens-before(先行发生)原则逻辑上的先后关系

案例：

- $x=5$ 线程A执行
- $y=x$ 线程B执行

问题：

y是否等于5？

如果线程A的操作($x= 5$) happens-before(先行发生)线程B的操作 ($y=x$)，那么可以确定线程B执行后
 $y=5$ 一定成立：

如果他们不存在happens-before原则，那么 $y = 5$ 不一定成立

这就是happens-before原则的威力。-----》包含可见性和有序性的约束

原则说明：

如果Java内存模型中所有的有序性都仅靠volatile和synchronized来完成，那么有很多操作都将会变得非常啰嗦，但是我们在编写Java并发代码的时候并没有察觉到这一点。

我们没有时时、处处、次次，添加volatile和synchronized来完成程序，这是因为Java语言中JMM原则下有一个“先行发生”(Happens-Before)的原则限制和规矩

这个原则非常重要：

它是判断数据是否存在竞争。线程是否安全的非常有用的手段。依赖这个原则，我们可以通过几条简单规则一揽子解决**并发环境下两个操作之间是否可能存在冲突的所有问题**，而不需要陷入Java内存模型苦涩难懂的底层编译原理之中。

6.4.1 happens-before总原则

- 如果一个操作happens-before另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。
- 两个操作之间存在happens-before关系，并不意味着一定要按照happens-before原则制定的顺序来执行如果重排序之后的执行结果与按照happens-before关系来执行的结果一致，那么这种重排序并不非法。

6.4.2 happens-before之8条

1. 次序原则

一个线程内，按照代码顺序，写在前面的操作先行发生于写在后面的操作；

前一个操作的结果可以被后续的操作获取。讲白点就是前面一个操作把变量X赋值为1，那后面一个操作肯定能知道X已经变成了1.

2. 锁定原则

一个unLock操作**先行发生于**后面((这里的“后面”是指时间上的先后))对同一个锁的lock操作；

3. volatile变量规则

对一个volatile变量的写操作先行发生于后面对这个变量的读操作，**前面的写对后面的读是可见的**，这里的“后面”同样是指时间上的先后。

4. 传递规则

如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C

5. 线程启动规则 (Thread Start Rule)

Thread对象的start()方法先行发生于此线程的每一个动作

6. 线程中断规则 (Thread Interruption Rule)

- 对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生
- 可以通过Thread.interrupted()检测到是否发生中断
- 也就是说你要先调用interrupt()方法设置过中断标志位，我才能检测到中断发送

7. 线程终止规则 (Thread Termination Rule)

线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过isAlive()等手段检测线程是否已经终止执行

8. 对象终结规则 (Finalizer Rule)

一个对象的初始化完成（构造函数执行结束）先行发生于它的finalize()方法的开始
即对象没有完成初始化之前，是不能调用finalized()方法的

6.4.3 happens-before-小总结

在 Java 语言里面，Happens-Before 的语义本质上是一种可见性

A Happens-Before B 意味着 A发生过的事情对B来说是可见的，无论 A事件和B事件是否发生在同一个线程里。

JMM的设计分为两部分：

- 一部分是面向我们程序员提供的，也就是happens-before规则，它通俗易懂的向我们程序员阐述了一个强内存模型，我们只要理解happens-before规则，就可以编写并发安全的程序了。
- 另一部分是针对JVM实现的，为了尽可能少的对编译器和处理器做约束从而提高性能，JMM在不影响程序执行结果的前提下对其不要求，即允许优化重排序。我们只需要关注前者就好了，也就是理解happens-before规则即可，其它繁杂的内容有JMM规范结合操作系统给我们搞定，我们只写好代码即可。

7. volatile与JMM

7.1 volatile修饰变量的特点

- 可见性
- 有序性

7.2 volatile的内存语义

- 当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量值立即刷新回主内存中
- 当读一个volatile变量时，JMM会把该线程对应的本地内存设置为无效，重新回到主内存中读取最新共享变量
- 所以volatile的写内存语义是直接刷新到主内存中，读的内存语义是直接从主内存中读取

*7.3 内存屏障Memory Barrier

内存屏障(也称内存栅栏，尿障指令等，是一类同步屏障指令，是CPU或编译器在对内存随机访问的操作中的一个同步点，使得此点之前的所有读写操作都执行后才可以开始执行此点之后的操作)，避免代码重排序。内存屏障其实就是一种JVM指令，Java内存模型的重排规则会要求Java编译器在生成JVM指令时插入特定的内存屏障指令，通过这些内存屏障指令，volatile实现了Java内存模型中的可见性和有序性(禁重排)，但volatile无法保证原子性。

内存屏障之前的所有写操作都要回写到主内存

内存屏障之后的所有读操作都能获得内存屏障之前的所有写操作的最新结果(实现了可见性)

写屏障(Store Memory Barrier): 告诉处理器在写屏障之前将所有存储在缓存(store bufferes)中的数据同步到主内存。也就是说当看到Store屏障指令，就必须把该指令之前所有写入指令执行完毕才能继续往下执行。

读屏障(Read Memory Barrier): 处理器在读屏障之后的读操作，都在读屏障之后执行。也就是说在Load屏障指令之后就能够保证后面的读取数据指令一定能够读取到最新的数据。



因此重排序时，不允许把内存屏障之后的指令重排序到内存屏障之前。一句话：对一个volatile变量的写，先行发生于任意后续对这个volatile变量的读，也叫写后读。

内存屏障的分类

粗分2种

- 读屏障 (Load Barrier)

在读指令之前插入读屏障，让工作内存或CPU高速缓存当中的缓存数据失效，重新回到主内存中获取最新数据

- 写屏障 (Store Barrier)

在写指令之后插入写屏障，强制把写缓冲区的数据刷回到主内存中

细分4种

屏障类型	指令示例	说明
LoadLoad	Load1; LoadLoad; Load2	保证load1的读取操作在load2及后续读取操作之前执行
StoreStore	Store1; StoreStore: Store2	在store2及其后的写操作执行前，保证store1的写操作已刷新到主内存
LoadStore	Load1; LoadStore; Store2	在store2及其后的写操作执行前，保证load1的读操作已读取结束
StoreLoad	Store1; StoreLoad; Load2	保证store1的写操作已刷新到主内存之后，load2及其后的读操作才能执行

什么叫保证有序性

通过内存屏障禁重排

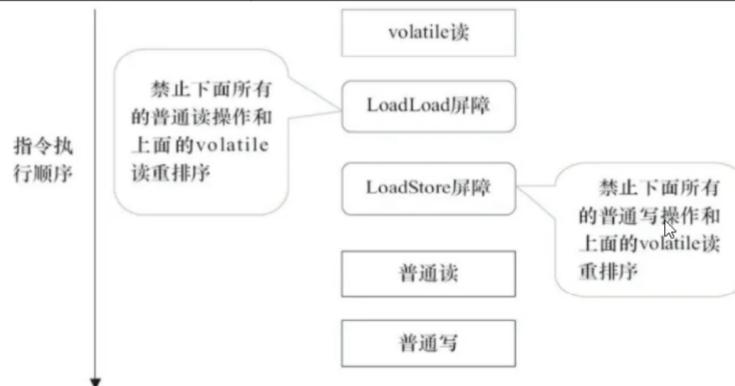
1. 重排序有可能影响程序的执行和实现，因此，我们有时候希望告诉JVM你别“自作聪明”给我重排序，我这里不需要排序
2. 对于编译器的重排序，JMM会根据重排序的规则，禁止特定类型的编译器重排序
3. 对于处理器的重排序，Java编译器在生成指令序列的适当位置，**插入内存屏障指令**，来**禁止特定类型的处理器排序**

happens-before之volatile变量规则

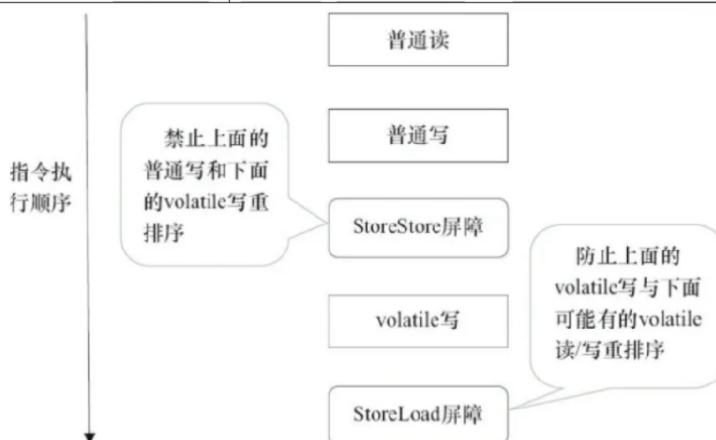
第一个操作	第二个操作: 普通读写	第二个操作: volatile读	第二个操作: volatile写
普通读写	可以重排	可以重排	不可以重排
volatile读	不可以重排	不可以重排	不可以重排
volatile写	可以重排	不可以重排	不可以重排

当第一个操作为volatile读时,不论第二个操作是什么,都不能重排序。这个操作保证了volatile读之后的操作不会被重排到volatile读之前。
当第二个操作为volatile写时,不论第一个操作是什么,都不能重排序。这个操作保证了volatile写之前的操作不会被重排到volatile写之后。
当第一个操作为volatile写时,第二个操作为volatile读时,不能重排。

在每个 volatile 读操作的后面插入一个 LoadLoad 屏障	禁止处理器把上面的volatile读与下面的普通读重排序。
在每个 volatile 读操作的后面插入一个 LoadStore 屏障	禁止处理器把上面的volatile读与下面的普通写重排序。



在每个 volatile 写操作的前面插入一个 StoreStore 屏障	可以保证在 volatile 写之前,其前面的所有普通写操作都已经刷新到主内存中。
在每个 volatile 写操作的后面插入一个 StoreLoad 屏障	作用是避免 volatile 写与后面可能有的 volatile 读/写操作重排序



7.4 volatile特性

7.4.1 保证可见性

保证不同线程对某个变量完成操作后结果及时可见,即该共享变量一旦改变所有线程立即可见

```
/**
 * @Author cx1
 * @Date 23/5/2023 20:41
 * @ClassReference: com.cx1.juc.volatle与JMM.VolatileSeeDemo
 * @Description:
 * 输出结果:
 * t1      ----come in
 * main    修改完成 flag:false
```

```

* t1 ----flag被设置为false
*/
public class VolatileseeDemo {

//    static boolean flag = true;
    static volatile boolean flag = true;

    public static void main(String[] args) {
        new Thread(() -> {
            System.out.println(Thread.currentThread().getName() + "\t ----come
in");

            while (flag) {

            }

            System.out.println(Thread.currentThread().getName() + "\t ----flag被
设置为false");
        }, "t1").start();

        try { TimeUnit.SECONDS.sleep(2); } catch (InterruptedException e){
e.printStackTrace();
        }

        flag = false;

        System.out.println(Thread.currentThread().getName() + "\t 修改完成 flag:"
+ flag);
    }
}

```

原理解释：

线程t1中为何看不到被主线程main修改为false的flag的值？

问题可能：

1. 主线程修改了flag之后没有将其刷新到主内存，所以t1线程看不到
2. 主线程将flag刷新到了主内存，但是t1一直读取的是自己工作内存中flag的值，没有去主内存中更新获取flag最新的值

我们的诉求：

1. 线程中修改了自己工作内存中的副本之后，立刻将其刷新到主内存
2. 工作内存中每次读取共享变量时，都去主内存中重新读取，然后拷贝到工作内存

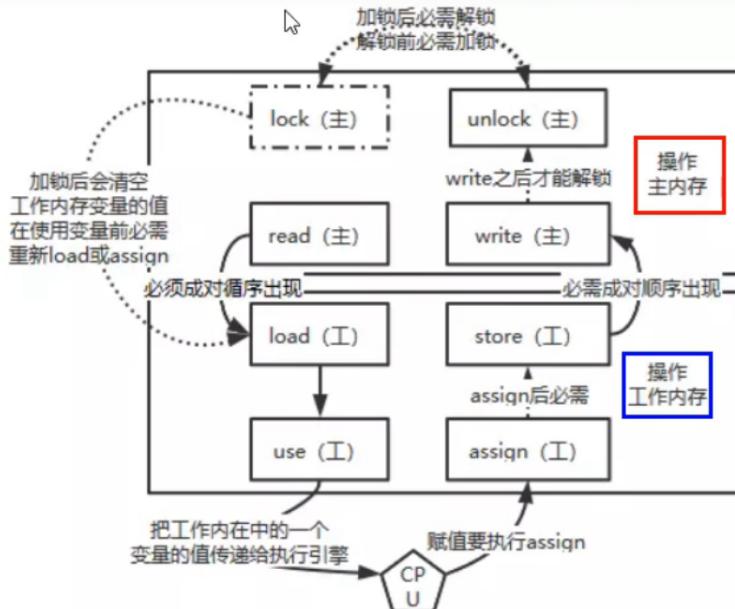
解决：

使用volatile修饰共享变量，就可以达到上面的效果，被volatile修改的变量有以下特点：

1. 线程中读取的时候，每次读取都会去主内存中读取共享变量最新的值，然后将其复制到工作内存
2. 线程中修改了工作内存中变量的副本，修改之后会立即刷新到主内存

Java内存模型中定义的8种**每个线程自己的工作内存与主物理内存之间的原子操作**

read(读取)→load(加载)→use(使用)→assign(赋值)→store(存储)→write(写入)→lock(锁定)→unlock(解锁)



read: 作用于主内存, 将变量的值从主内存传输到工作内存, 主内存到工作内存

load: 作用于工作内存, 将**read**从主内存传输的变量值放入工作内存变量副本中, 即数据加载

use: 作用于工作内存, 将工作内存变量副本的值传递给执行引擎, 每当JVM遇到需要该变量的字节码指令时会执行该操作

assign: 作用于工作内存, 将从执行引擎接收到的值赋值给工作内存变量, 每当JVM遇到一个给变量赋值字节码指令时会执行该操作

store: 作用于工作内存, 将赋值完毕的工作变量的值写回给主内存

write: 作用于主内存, 将**store**传输过来的变量值赋值给主内存中的变量

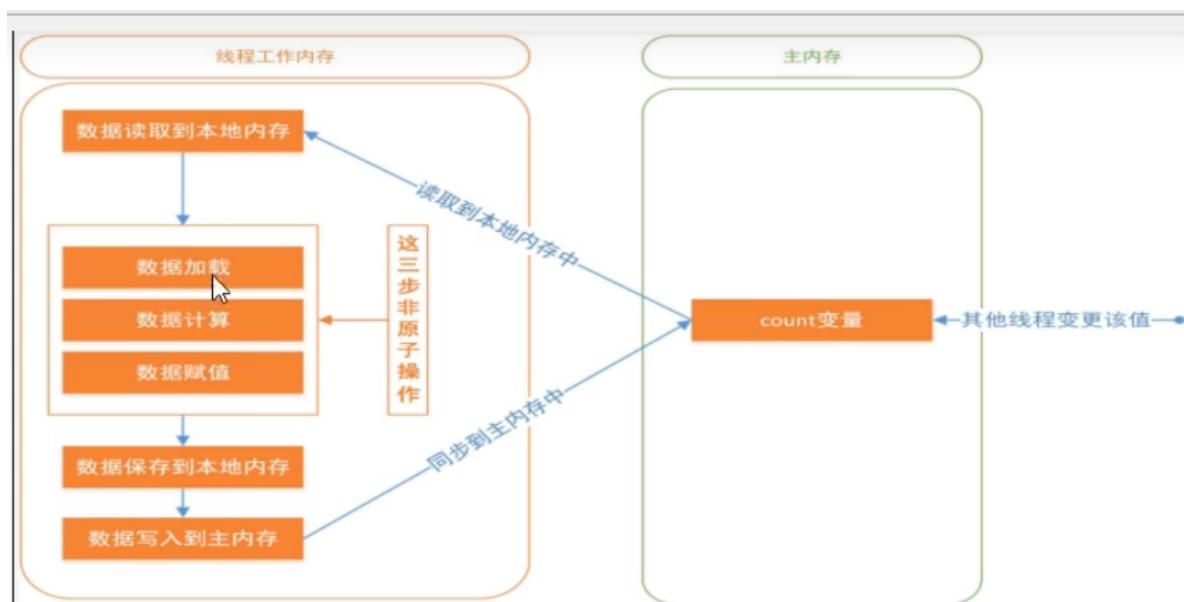
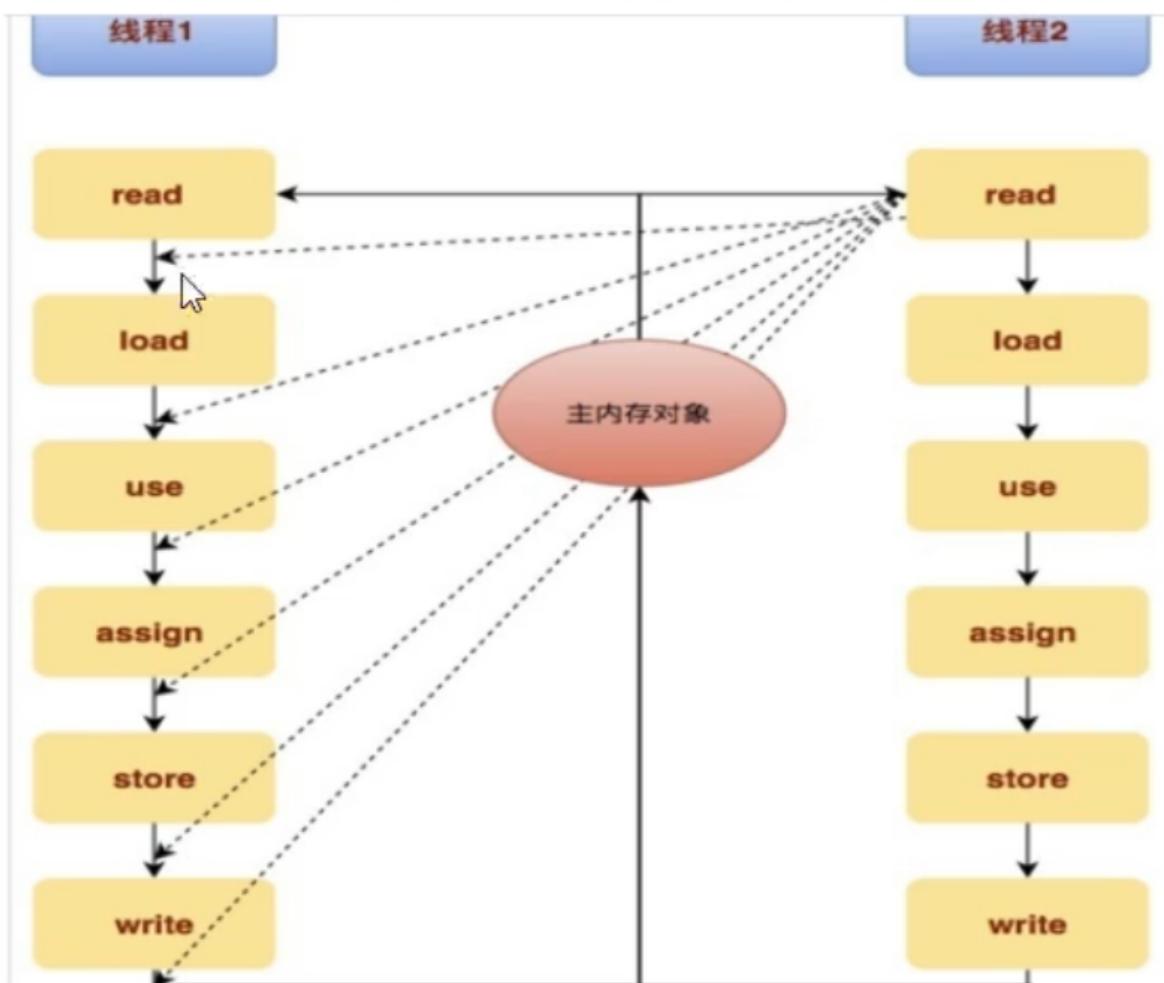
由于上述6条只能保证单条指令的原子性, 针对多条指令的组合性原子保证, 没有大面积加锁, 所以, JVM提供了另外两个原子指令:

lock: 作用于主内存, 将一个变量标记为一个线程独占的状态, 只是写时候加锁, 就只是锁了写变量的过程。

unlock: 作用于主内存, 把一个处于锁定状态的变量释放, 然后才能被其他线程占用

7.4.2 没有原子性

当线程1对主内存对象发起read操作到write操作第一套流程的时间里, 线程2随时都有可能对这个**主内存对象**发起第二套操作



对于volatile变量具备可见性，JVM只是保证从主内存加载到线程工作内存的值是最新的，**也仅是数据加载时最新的**。但是多线程环境下，“数据计算”和“数据赋值”操作可能多次出现，若数据在加载之后，若主内存**volatile**修饰变量发生修改之后，线程工作内存中的操作将会作废去读主内存最新值，操作出现写丢失问题。即**各线程私有内存和主内存公共内存中变量不同步**，进而导致数据不一致。由此可见volatile解决的是变量读时的可见性问题，**但无法保证原子性**，对于多线程修改主内存共享变量的场景必须使用加锁同步。

从i++角度看

```

public volatile int n;
public void add() {
    n++;
}

```

n++被拆分成了3个指令：
执行getfield拿到原始n;
执行iadd进行加1操作;
执行putfield写把累加后的值写回

```

0: aload_0
1: dup
2: getfield    #2                  // Field n:I
5: iconst_1
6: iadd
7: putfield   #2                  // Field n:I
10: return
LineNumberTable:
line 15: 0
line 16: 10
LocalVariableTable:

```

原子性指的是一个操作是不可中断的，即使是在多线程环境下，一个操作一旦开始就不会被其他线程影响。

```

public void add(){
    i++; //不具备原子性，该操作是先读取值，然后写回一个新值，相当于原来的值加上1，分3步完成
}

```

如果第二个线程在第一个线程读取旧值和写回新值期间读取i的域值，那么第二个线程就会与第一个线程一起看到同一个值，并执行相同值的加1操作，这也就造成了线程安全失败，因此对于add方法必须使用synchronized修饰以便保证线程安全

volatile变量不适合参与到依赖当前值的运算

volatile变量不适合参与到依赖当前值的运算，如果i=i+1,i++之类的

那么一开可见性的特点volatile可以用在哪些地方呢？通常volatile用来保存某个状态的boolean值或int值。

《深入理解Java虚拟机》提到：

由于volatile变量只能保证可见性，在不符合以下两条规则的运算场景中，我们仍然要通过加锁（使用synchronized、java.util.concurrent中的锁或原子类）来保证原子性：

·运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值。

·变量不需要与其他的状态变量共同参与不变约束。

对于volatile变量，JVM只是保证从主内存加载到线程工作内存的值是最新的，也只是数据加载时最新的。

如果第二个线程在第一个线程读取旧值和写回新值期间读取i的域值，也就造成了线程安全问题。

7.4.3 指令禁重排

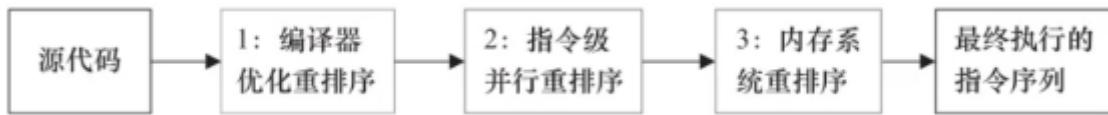
重排序是指编译器和处理器为了优化程序性能而对指令序列进行重新排序的一种手段，有时候会改变程序语句的先后顺序

不存在数据依赖关系，可以重排序；

存在数据依赖关系，禁止重排序

但重排后的指令绝对不能改变原有的串行语义！**这点在并发设计中必须要重点考虑！**

重排序的分类和执行流程



- 编译器优化的重排序:** 编译器在不改变单线程串行语义的前提下, 可以重新调整指令的执行顺序
- 指令级并行的重排序:** 处理器使用指令级并行技术来让多条指令重叠执行, 若不存在数据依赖性, 处理器可以改变语句对应机器指令的执行顺序
- 内存系统的重排序:** 由于处理器使用缓存和读/写缓冲区, 这使得加载和存储操作看上去可能是乱序执行

数据依赖性: 若两个操作访问同一变量, 且这两个操作中有一个为写操作, 此时两操作间就存在数据依赖性。

若存在数据依赖关系, 禁止重排序==> 重排序发生, 会导致程序运行结果不同。

编译器和处理器在重排序时, 会遵守数据依赖性, 不会改变存在依赖关系的两个操作的执行, 但不同处理器和不同线程之间的数据性不会被编译器和处理器考虑, 其只会作用于单处理器和单线程环境, **下面三种情况, 只要重排序两个操作的执行顺序, 程序的执行结果就会被改变。**

名称	代码示例	说明
写后读	a=1; b=a;	写一个变量之后, 再读这个变量
写后写	a=1; a=2;	写一个变量之后, 再写这个变量
读后写	a=b; b=1;	读一个变量之后, 再写这个变量

禁止指令重排:



7.5 如何正确使用volatile

- 对于单一赋值的变量
 - volatile int a = 10
 - volatile boolean flag = false
- 状态标志, 判断业务是否结束
- 开销较低的读, 写锁策略
- DCL双端锁的发布

7.6 总结

可见性

写操作的话，这个变量的最新值会立即刷新回到主内存中

读操作的话，总是能够读取到这个变量的最新值，也就是这个变量最后被修改的值

当某个线程收到通知，去读取volatile修饰的变量的值的时候，**线程私有工作内存的数据失效**，需要重新回到主内存中去读取最新的数据

没有原子性

禁重排

内存屏障：一种屏障指令，它使得CPU或编译器对屏障指令的前和后所发出的内存操作执行一个排序的约束。也叫内存栅栏或栅栏指令

加了volatile关键字为什么就会有内存屏障

字节码层面

```
7 public class VolatileSeeDemo
8 {
9     //static          boolean flag = true;      //不加volatile, 没有可见性
10    volatile boolean flag = true;           //加了volatile, 保证可见性
11 }
```

Local	Local (1)
#16 = NameAndType	#7:#8 // "<init>":()V
#17 = NameAndType	#5:#6 // flag:Z
#18 = Utf8	com/atguigu/juc/jmm/VolatileSeeDemo
#19 = Utf8	java/lang/Object
{	
volatile boolean flag;	
descriptor: Z	
flags: ACC_VOLATILE	
public com.atguigu.juc.jmm.VolatileSeeDemo();	
descriptor: ()V	
flags: ACC_PUBLIC	
Code:	

它影响的是 Class 内的 Field 的 flags :

添加了一个 ACC_VOLATILE

JVM在把字节码生成机器码的时候，发现操作是volatile变量的话，会按照JMM的规范，在相应的位置插入内存屏障

三句话总结

volatile写之前的操作，都禁止重排序到volatile之后

volatile读之后的操作，都禁止重排序到volatile之前

volatile写之后volatile读，禁止重排序

8. CAS

compare and swap的缩写，中文翻译成**比较并交换**，实现并发算法时常用到的一种技术

它包含三个操作数——内存位置、预期原值及更新值

执行CAS操作的时候，将内存位置的值与预期原值比较

如果相匹配，那么处理器会自动将该位置值更新为新值

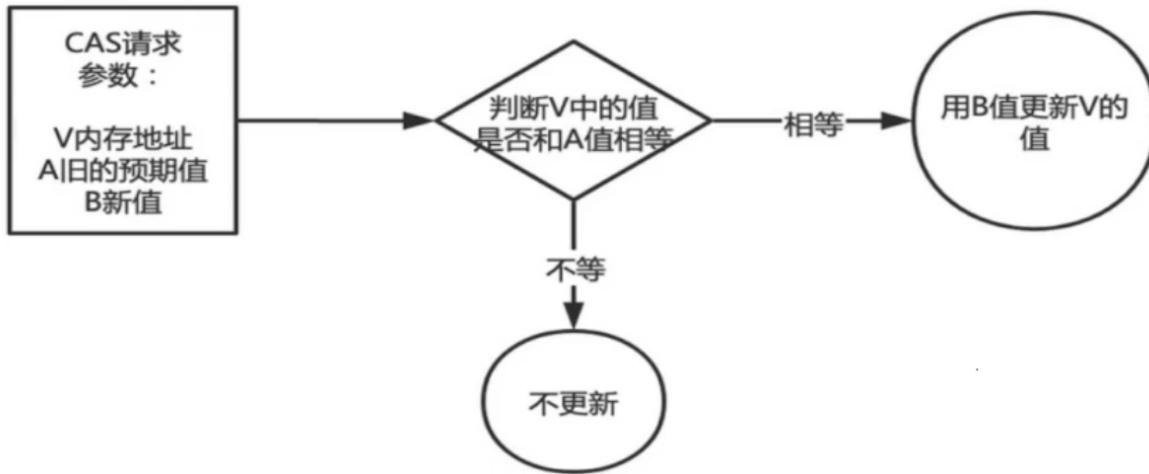
如果不匹配，处理器不做任何操作，多个线程同时执行CAS操作只有一个会成功

8.1 原理

CAS有3个操作数，位置内存值V，旧的预期值A，要修改的更新值B

当且仅当旧的预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做或重来

当它重来重试的这种行为成为----自旋



硬件保证

CAS是JDK提供的非阻塞原子性操作，它通过硬件保证了比较-更新的原子性。

它是非阻塞的且自身具有原子性，也就是说这玩意效率更高且通过硬件保证，说明这玩意更可靠

CAS是一条CPU的原子指令 (**cmpxchg指令**)，不会造成所谓的数据不一致问题，Unsafe提供的CAS方法(如compareAndSwapXXX) 底层实现即为CPU指令cmpxchg。

执行cmpxchg指令的时候，会判断当前系统是否为多核系统，如果是就给总线加锁，只有一个线程会对总线加锁成功，加锁成功之后会执行cas操作，也就是说CAS的原子性实际上是CPU实现独占的，比起用synchronized重量级锁，这里的排他时间要短很多，所以在多线程情况下性能会比较好。

CAS源码

```
public final boolean compareAndSet(int expect, int update) {  
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);  
}
```

```
public final native boolean compareAndSwapObject(Object var1, long var2, Object  
var4, Object var5);
```

```
public final native boolean compareAndSwapInt(Object var1, long var2, int var4,  
int var5);
```

```
public final native boolean compareAndSwapLong(Object var1, long var2, long  
var4, long var6);
```

- var1：表示要操作的对象
- var2：表示要操作对象中属性地址的偏移量

- var4: 表示需要修改数据的期望的值
- var5/var6: 表示需要修改为的新值

8.2 底层原理和对Unsafe类的理解

Unsafe

是CAS的核心类，由于Java方法无法直接访问底层系统，需要通过本地(native) 方法来访问，Unsafe相当于一个后门，基于该类可以直接操作特定内存的数据。Unsafe类存在于sun.misc包中，其内部方法操作可以像C的指针一样直接操作内存，因为Java中CAS操作的执行依赖于Unsafe类的方法。

valueOffset

表示该变量值在内存中的偏移地址，因为Unsafe就是根据内存偏移地址获取数据的

value

```
private volatile int value;
```

用volatile修饰，保证了多线程之间的内存可见性

为什么atomicInteger.getAndIncrement();可以实现多线程的++

CAS的全称为Compare-And-Swap，**它是一条CPU并发原语**。

它的功能是判断内存某个位置的值是否为预期值，如果是则更改为新的值，这个过程是原子的

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销执行效率大为提升

CAS并发原语体现在JAVA语言中就是sun.misc.Unsafe类中的各个方法。调用UnSafe类中的CAS方法，JVM会帮我们实现CAS汇编指令。这是一种完全依赖于硬件的功能，通过它实现了原子操作。再次强调，由于CAS是一种系统原语，原语属于操作系统用语范畴，是由若干条指令组成的，用于完成某个功能的一个过程，并且原语的执行必须是连续的，在执行过程中不允许被中断，也就是说**CAS是一条CPU的原子指令，不会造成所谓的数据不一致问题**。

8.3 源码分析

```
new AtomicInteger().getAndIncrement();
```

Unsafe.java

```
public final int getAndAddInt(Object o, long offset, int delta) {
    int v;
    do {
        v = getIntVolatile(o, offset);
    } while (!compareAndSwapInt(o, offset, v, v + delta));
    return v;
}
```

假设线程A和线程B两个线程同时执行getAndAddInt操作(分别跑在不同CPU上)：

1. AtomicInteger里面的value原始值为3，即主内存中AtomicInteger的value对3，根据JMM模型，线程A和线程B各自持有一份值为3的value的副本分别到各自的工作内存。
2. 线程A通过getIntVolatile(var1,var2)拿到value值3，**这时线程A被挂起**。
3. 线程B也通过detIntVlatile(var1,var2)方法获取到value值3，此时刚好线程B**没有被挂起并执行** compareAndSwapInt方法比较内存值也为3，成功修改内存值为4，线程B打完收工，一切OK。

4. 这时线程A恢复，执行compareAndSwapInt方法比较，发现自己手里的值数字3和主内存的值数字4不一致，说明该值已经被其它线程抢先一步修改过了，那A线程本次修改失败，**只能重新读取重新来一遍了。**
5. 线程A重新获取value值，因为变量value被volatile修饰，所以其它线程对它的修改，线程A总是能够看到，线程A继续执行compareAndSwapInt进行比较替换，直到成功。

8.4 汇编层面

Unsafe类中的compareAndSwapInt，是一个本地方法，该方法的实现位于unsafe.cpp中

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe, jobject obj, jlong offset, jint e, jint x))
UnsafeWrapper("Unsafe_CompareAndSwapInt");
oop p = JNIHandles::resolve(obj);
//先想办法拿到变量value在内存中的地址，根据偏移量valueOffset，计算 value 的地址
jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
//调用 Atomic 中的函数 cmpxchg来进行比较交换，其中参数x是要交换的值 e是要比较的值
//cas成功，返回期望值e，等于e，此方法返回true
//cas失败，返回内存中的value值，不等于e，此方法返回false
return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END
//JDK提供的CAS机制，在汇编层级会禁止变量两侧的指令优化，然后使用cmpxchg指令比较并更新变量值(原子性)
(Atomic::cmpxchg(x, addr, e)) == e;
```

// 调用 Atomic 中的函数 cmpxchg来进行比较交换，其中参数x是即将更新的值，参数e是原内存的值
 return (jint)(Atomic::cmpxchg(x, addr, e)) == e;

```
unsigned Atomic::cmpxchg(unsigned int exchange_value, volatile unsigned int* dest, unsigned int compare_value) {
    assert(sizeof(unsigned int) == sizeof(jint), "more work to do");
    /*
     * 根据操作系统类型调用不同平台下的重载函数，这个在预编译期间编译器会决定调用哪个平台下的重载函数*
     */
    return (unsigned int)Atomic::cmpxchg((jint)exchange_value, (volatile jint*)dest, (jint)compare_value);
}
```

```
inline jint Atomic::cmpxchg (jint exchange_value, volatile jint* dest, jint compare_value) {
//判断是否是多核CPU
//三个move指令表示的是将后面的值移动到前面的寄存器上
    int mp = os::is_MP();
    __asm {
        mov edx, dest
        mov ecx, exchange_value
        mov eax, compare_value
        //CPU原语级别，CPU触发
        LOCK_IF_MP(mp)
        //比较并交换指令
        //cmpxchg: 即“比较并交换”指令
        //dword: 全称是 double word 表示两个字，一共四个字节
        //ptr: 全称是 pointer，与前面的 dword 连起来使用，表明访问的内存单元是一个双字单元
        //将 eax 寄存器中的值 (compare_value) 与 [edx] 双字内存单元中的值进行对比,
        //如果相同，则将 ecx 寄存器中的值 (exchange_value) 存入 [edx] 内存单元中
        cmpxchg dword ptr [edx], ecx
    }
}
```

8.5 小总结

你只需要记住：CAS是靠硬件实现的从而在硬件层面提升效率，最底层还是交给硬件来保证原子性和可见性

实现方式是基于硬件平台的汇编指令，在intel的CPU中(X86机器上)，使用的是汇编指令cmpxchg指令。

核心思想就是：比较要更新变量的值V和预期值E (compare)，相等才会将V的值设为新值N (swap)如果不相等自旋再来。

8.6 原子引用

```
class User {  
  
    String userName;  
    int age;  
  
    public User(String userName, int age) {  
        this.userName = userName;  
        this.age = age;  
    }  
  
    @Override  
    public String toString() {  
        return "User{" +  
            "userName='" + userName + '\'' +  
            ", age=" + age +  
            '}';  
    }  
}  
public class AtomicReferenceDemo {  
  
    public static void main(String[] args) {  
        AtomicReference<User> atomicReference = new AtomicReference<>();  
        User z3 = new User("z3", 22);  
        User li4 = new User("li", 23);  
  
        atomicReference.set(z3);  
  
        System.out.println(atomicReference.compareAndSet(z3, li4) + "\t" +  
            atomicReference.get().toString());  
        System.out.println(atomicReference.compareAndSet(z3, li4) + "\t" +  
            atomicReference.get().toString());  
    }  
}
```

8.7 CAS与自旋锁，借鉴CAS思想

自旋锁 (spinlock)

CAS 是实现自旋锁的基础，CAS 利用 CPU 指令保证了操作的原子性，以达到锁的效果，至于自旋呢，看字面意思也很明白，自己旋转。是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，当线程发现锁被占用时，会不断循环判断锁的状态，直到获取。这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU

CAS 是实现自旋锁的基础，自旋翻译成人话就是循环，一般是用一个无限循环实现。这样一来，一个无限循环中，执行一个CAS 操作
当操作成功返回 true 时，循环结束：
当返回 false 时，接着执行循环，继续尝试 CAS 操作，直到返回 true。

```

/**
 * @Author cxl
 * @Date 24/5/2023 15:51
 * @ClassReference: com.cxl.juc.cas.SpinLockDemo
 * @Description:
 * 题目:实现一个自旋锁, 复习CAS思想
 * 自旋锁好处: 循环比较获取没有类似wait的阻塞。
 *
 * 通过CAS操作完成自旋锁, A线程先进来调用myLock方法自己持有锁5秒钟, B随后进来后发现
 * 当前有线程持有锁, 所以只能通过自旋等待, 直到A 释放锁后B 随后抢到。
 */
public class SpinLockDemo {

    AtomicReference<Thread> atomicReference = new AtomicReference<>();

    public void lock() {
        Thread thread = Thread.currentThread();
        System.out.println(Thread.currentThread().getName() + "\t ----come in");
        while (!atomicReference.compareAndSet(null, thread)) {

        }
    }

    public void unlock() {
        Thread thread = Thread.currentThread();
        atomicReference.compareAndSet(thread, null);
        System.out.println(Thread.currentThread().getName() + "\t ----task
over");
    }

    public static void main(String[] args) {
        SpinLockDemo spinLockDemo = new SpinLockDemo();

        new Thread(() -> {
            spinLockDemo.lock();
            try { TimeUnit.SECONDS.sleep(5); } catch (InterruptedException e){
e.printStackTrace(); }
            spinLockDemo.unlock();
        }, "A").start();

        // 线程A先于B启动
        try { TimeUnit.MILLISECONDS.sleep(500); } catch (InterruptedException e){
e.printStackTrace(); }

        new Thread(() -> {
            spinLockDemo.lock();
            spinLockDemo.unlock();
        }, "B").start();
    }
}

```

8.8 CAS缺点

循环时间长开销大

我们可以看到getAndAddInt方法执行时, 有个do while

```

public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}

```

如果CAS失败，会一直进行尝试。如果CAS长时间一直不成功，可能会给CPU带来很大的开销。

ABA问题

CAS会导致“ABA问题”

CAS算法实现一个重要前提需要取出内存中某时刻的数据并在当下时刻比较并替换，那么在这个时间差类会导致数据的变化

比如说一个线程1从内存位置V中取出A，这时候另一个线程2也从内存中取出A，并且线程2进行了一些操作将值变成了B，然后线程2又将V位置的数据变成A，这时候线程1进行CAS操作发现内存中仍然是A，预期OK，然后线程1操作成功。

尽管线程1的CAS操作成功，但是不代表这个过程就是没有问题的。

8.9 ABA问题的解决

AtomicStampedReference的使用

```

class Book {
    private int id;
    private String bookName;

    public Book(int id, String bookName) {
        this.id = id;
        this.bookName = bookName;
    }
}

public class AtomicStampedDemo {

    public static void main(String[] args) {
        Book javaBook = new Book(1, "javaBook");

        AtomicStampedReference<Book> stampedReference = new
        AtomicStampedReference<>(javaBook, 1);

        System.out.println(stampedReference.getReference() + "\t" +
                           stampedReference.getStamp());

        Book mysqlBook = new Book(2, "mysqlBook");

        boolean b;
        b = stampedReference.compareAndSet(javaBook, mysqlBook,
                                           stampedReference.getStamp(), stampedReference.getStamp() + 1);
    }
}

```

```

        System.out.println(b + "\t" + stampedReference.getReference() + "\t" +
stampedReference.getStamp());

        b = stampedReference.compareAndSet(mysqlBook, javaBook,
stampedReference.getStamp(), stampedReference.getStamp() + 1);
        System.out.println(b + "\t" + stampedReference.getReference() + "\t" +
stampedReference.getStamp());
    }
}

```

ABADemo

```

public class ABADemo {

    static AtomicInteger atomicInteger = new AtomicInteger(100);

    static AtomicStampedReference<Integer> stampedReference = new
AtomicStampedReference<>(100 ,1);

    public static void main(String[] args) {

        new Thread(() -> {
            int stamp = stampedReference.getStamp();
            System.out.println(Thread.currentThread().getName() + "\t 首次版本
号: " + stamp);

            try { TimeUnit.MILLISECONDS.sleep(500); }catch (InterruptedException
e){ e.printStackTrace(); }

            stampedReference.compareAndSet(100, 101,
stampedReference.getStamp(), stampedReference.getStamp() + 1);
            System.out.println(Thread.currentThread().getName() + "\t 2次流水号: "
+ stamp);

            stampedReference.compareAndSet(101, 100,
stampedReference.getStamp(), stampedReference.getStamp() + 1);
            System.out.println(Thread.currentThread().getName() + "\t 3次流水号: "
+ stamp);
        }, "t3").start();

        new Thread(() -> {
            int stamp = stampedReference.getStamp();
            System.out.println(Thread.currentThread().getName() + "\t 首次版本
号: " + stamp);

            // 等待t3线程发生ABA问题
            try { TimeUnit.SECONDS.sleep(1); }catch (InterruptedException e){
e.printStackTrace(); }

            boolean b = stampedReference.compareAndSet(100, 202, stamp, stamp +
1);

            System.out.println(b + "\t" + stampedReference.getReference() + "\t"
+ stampedReference.getStamp());
        }, "t4").start();
    }
}

```

```

private static void abaHappen() {
    new Thread(() -> {
        atomicInteger.compareAndSet(100, 101);
        try { TimeUnit.MILLISECONDS.sleep(10); }catch (InterruptedException e){ e.printStackTrace(); }
        atomicInteger.compareAndSet(101, 100);
    }, "t1").start();

    new Thread(() -> {
        try { TimeUnit.MILLISECONDS.sleep(200); }catch (InterruptedException e){ e.printStackTrace(); }
        System.out.println(atomicInteger.compareAndSet(100, 2022) + "\t" +
        atomicInteger.get());
    }, "t1").start();
}
}

```

9. 原子操作类

9.1 基本类型原子类

- AtomicInteger
- AtomicBoolean
- AtomicLong

```

class MyNumber {

    AtomicInteger atomicInteger = new AtomicInteger();

    public void addPlusPlus() {
        atomicInteger.getAndIncrement();
    }
}

public class AtomicIntegerDemo {

    public static final int SIZE = 50;

    public static void main(String[] args) throws InterruptedException {

        MyNumber myNumber = new MyNumber();

        CountDownLatch countDownLatch = new CountDownLatch(SIZE);

        for (int i = 0; i < SIZE; i++) {
            new Thread(() -> {
                try {
                    for (int j = 0; j < 1000; j++) {
                        myNumber.addPlusPlus();
                    }
                } finally {
                    countDownLatch.countDown();
                }
            })
        }
    }
}

```

```

        }
        , string.valueOf(i)).start();
    }

    countDownLatch.await();
    System.out.println(Thread.currentThread().getName() + "\t result:" +
myNumber	atomicInteger.get());
}
}

```

9.2 数组类型原子类

- AtomicIntegerArray
- AtomicLongArray
- AtomicReferenceArray

9.3 引用类型原子类

- AtomicReference
- AtomicStampedReference
- AtomicMarkableReference

```

/**
 * @Author cxl
 * @Date 24/5/2023 19:11
 * @ClassReference: com.cxl.juc.原子操作类.AtomicMarkableReferenceDemo
 * @Description:
 *
 * AtomicStampedReference,version号 , +1
 * AtomicMarkableReference, 一次, false, true
 *
 * 输出结果:
 * t1  默认标识false
 * t2  默认标识false
 * t2  t2线程CASresult:false
 * t2  true
 * t2  1000
 */
public class AtomicMarkableReferenceDemo {

    static AtomicMarkableReference markableReference = new
AtomicMarkableReference(100, false);

    public static void main(String[] args) {

        new Thread(() -> {
            boolean marked = markableReference.isMarked();
            System.out.println(Thread.currentThread().getName() + "\t 默认标识" +
marked);
            // 暂停1秒钟线程, 等待后面的t2线程和t1拿到一样的标识
            try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e){
e.printStackTrace(); }
            markableReference.compareAndSet(100, 1000, marked, true);
        }, "t1").start();
    }
}

```

```

        new Thread() -> {
            boolean marked = markableReference.isMarked();
            System.out.println(Thread.currentThread().getName() + "\t 默认标识" +
marked);
            // 暂停1秒钟线程
            try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e){
e.printStackTrace(); }
            boolean b = markableReference.compareAndSet(100, 1000, marked,
!marked);
            System.out.println(Thread.currentThread().getName() + "\t t2线程
CASresult:" + b);
            System.out.println(Thread.currentThread().getName() + "\t" +
markableReference.isMarked());
            System.out.println(Thread.currentThread().getName() + "\t" +
markableReference.getReference());
        }, "t2").start();
    }
}

```

9.4 对象的属性修改原子类

- AtomicIntegerFieldUpdater: 原子更新对象中int类型字段的值
- AtomicLongFieldUpdater: 原子更新对象中Long类型字段的值
- AtomicReferenceFieldUpdater: 原子更新引用类型字段的值

使用目的：以一种线程安全的方式操作非线程安全对象内的某些字段

使用要求：

- 更新的对象属性必须使用 public volatile 修饰符
- 因为对象的属性修改类型原子类都是抽象类，所以每次使用都必须使用静态方法newUpdater()创建一个更新器，并且需要设置想要更新的类和属性

面试官问你：在哪用了volatile

AtomicRferenceFieldUpdater

AtomicIntegerFieldUpdaterDemo

```

/**
 * @Author cxl
 * @Date 25/5/2023 09:01
 * @ClassReference: com.cxl.juc.原子操作类.AtomicIntegerFieldUpdaterDemo
 * @Description:
 * 需求:
 * 10个线程
 * 每个线程转账1000
 * 不使用synchronized, 尝试使用AtomicIntegerFieldUpdater来实现
 */
class BankAccount {

    String bankName = "CCB";

    // 更新对象的属性必须使用 public volatile 修饰符
    public volatile int money = 0;

    public void add() {

```

```

        money++;
    }

    // 因为对象的属性修改类型原子类都是抽象类，所以每次使用都必须
    // 使用静态方法newUpdater()创建一个更新器，并且需要设置想要更新的类和属性
    AtomicIntegerFieldUpdater fieldUpdater =
        AtomicIntegerFieldUpdater.newUpdater(BankAccount.class, "money");

    // 不加synchronized，保证高性能原子性
    public void transMoney(BankAccount bankAccount) {
        fieldUpdater.getAndIncrement(bankAccount);
    }
}

public class AtomicIntegerFieldUpdaterDemo {

    public static void main(String[] args) throws InterruptedException {
        BankAccount bankAccount = new BankAccount();
        CountDownLatch countDownLatch = new CountDownLatch(10);

        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                try {
                    for (int j = 0; j < 1000; j++) {
                        bankAccount.add();
                        bankAccount.transMoney(bankAccount);
                    }
                } finally {
                    countDownLatch.countDown();
                }
            }, String.valueOf(i)).start();
        }

        countDownLatch.await();
        System.out.println(Thread.currentThread().getName() + "\t result:" +
bankAccount.money);
    }
}

```

AtomicReferenceFieldUpdaterDemo

```

/**
 * @Author cxl
 * @Date 25/5/2023 09:25
 * @ClassReference: com.cxl.juc.原子操作类.AtomicReferenceFieldUpdater
 * @Description:
 * 需求:
 * 多线程并发调用一个类的初始化方法，如果未被初始化过，将执行初始化工作
 * 需求只能被初始化一次，只有一个线程操作成功
 */
class MyVar {

    public volatile Boolean isInit = Boolean.FALSE;

    AtomicReferenceFieldUpdater<MyVar, Boolean> referenceFieldUpdater =
        AtomicReferenceFieldUpdater.newUpdater(MyVar.class, Boolean.class,
"isInit");
}

```

```

public void init(MyVar myVar) {
    if (referenceFieldUpdater.compareAndSet(myVar, Boolean.FALSE,
Boolean.TRUE)) {
        System.out.println(Thread.currentThread().getName() + "\t -----start
init, need 3 seconds");
        // 暂停3秒钟线程
        try { TimeUnit.SECONDS.sleep(3); } catch (InterruptedException e){
e.printStackTrace(); }
        System.out.println(Thread.currentThread().getName() + "\t -----over
init");
    } else {
        System.out.println(Thread.currentThread().getName() + "\t -----已经有
线程在进行初始化工作");
    }
}
}

public class AtomicReferenceFieldUpdaterDemo {

public static void main(String[] args) {

MyVar myVar = new MyVar();

for (int i = 0; i < 5; i++) {
    new Thread(() -> {
        myVar.init(myVar);
    }, i+"").start();
}
}
}

```

9.5 原子操作增强类原理深度解析

- DoubleAccumulator
- DoubleAdder
- LongAccumulator
- LongAdder

Java 开发手册

17. 【参考】`volatile` 解决多线程内存不可见问题。对于一写多读，是可以解决变量同步问题，但是如果多写，同样无法解决线程安全问题。

说明：如果是 `count++` 操作，使用如下类实现：`AtomicInteger count = new AtomicInteger();`

`count.addAndGet(1);` 如果是 JDK8，推荐使用 `LongAdder` 对象，比 `AtomicLong` 性能更好（减少乐观锁的重试次数）。

- `LongAdder` 只能用来计算加法，且从零开始计算
- `LongAccumulator` 提供了自定义的函数操作

LongAdderAPIDemo

```

public class LongAdderAPIDemo {

public static void main(String[] args) {
    LongAdder longAdder = new LongAdder();
}
}

```

```

        longAdder.increment();
        longAdder.increment();
        longAdder.increment();

        System.out.println(longAdder.sum());

    //      LongAccumulator longAccumulator = new LongAccumulator((x, y) -> x+y,
    0);
        LongAccumulator longAccumulator = new LongAccumulator(new
LongBinaryOperator() {
            @Override
            public long applyAsLong(long left, long right) {
                return left + right;
            }
        }, 0);

        longAccumulator.accumulate(1); // 1
        longAccumulator.accumulate(3); // 4

        System.out.println(longAccumulator.get());
    }
}

```

热点商品点赞计算器

热点商品点赞计算器，点赞数加加统计，不要求实时精确

```

/**
 * @Author cx1
 * @Date 25/5/2023 15:09
 * @ClassReference: com.cx1.juc.原子操作类.AccumulatorCompareDemo
 * @Description:
 *
 * 需求：50个线程，每个线程100w次，总点赞数出来
 * ----constTime:2213毫秒      clickBySynchronized:50000000
 * ----constTime:450毫秒       clickByAtomicLong:50000000
 * ----constTime:95毫秒        clickByLongAdder:50000000
 * ----constTime:87毫秒        clickByLongAccumulator:50000000
 */
class ClickNumber {

    int number = 0;

    public synchronized void clickBySynchronized() {
        number++;
    }

    AtomicLong atomicLong = new AtomicLong(0);

    public void clickByAtomicLong() {
        atomicLong.getAndIncrement();
    }

    LongAdder longAdder = new LongAdder();

    public void clickByLongAdder() {
        longAdder.increment();
    }
}

```

```
}

LongAccumulator longAccumulator = new LongAccumulator((x, y) -> x+y, 0);

public void clickByLongAccumulator() {
    longAccumulator.accumulate(1);
}
}

public class AccumulatorCompareDemo {

    public static final int _1W = 10000;

    public static final int threadNumber = 50;

    public static void main(String[] args) throws InterruptedException {
        ClickNumber clickNumber = new ClickNumber();
        long startTime;
        long endTime;

        CountDownLatch countDownLatch1 = new CountDownLatch(threadNumber);
        CountDownLatch countDownLatch2 = new CountDownLatch(threadNumber);
        CountDownLatch countDownLatch3 = new CountDownLatch(threadNumber);
        CountDownLatch countDownLatch4 = new CountDownLatch(threadNumber);

        startTime = System.currentTimeMillis();
        for (int i = 0; i < threadNumber; i++) {
            new Thread(() -> {
                try {
                    for (int j = 0; j < 100 * _1W; j++) {
                        clickNumber.clickBySynchronized();
                    }
                } finally {
                    countDownLatch1.countDown();
                }
            }, String.valueOf(i)).start();
        }
        countDownLatch1.await();
        endTime = System.currentTimeMillis();
        System.out.println("----constTime:" + (endTime - startTime) + "毫秒 \t"
clickBySynchronized:" + clickNumber.number);

        startTime = System.currentTimeMillis();
        for (int i = 0; i < threadNumber; i++) {
            new Thread(() -> {
                try {
                    for (int j = 0; j < 100 * _1W; j++) {
                        clickNumber.clickByAtomicLong();
                    }
                } finally {
                    countDownLatch2.countDown();
                }
            }, String.valueOf(i)).start();
        }
        countDownLatch2.await();
        endTime = System.currentTimeMillis();
        System.out.println("----constTime:" + (endTime - startTime) + "毫秒 \t"
clickByAtomicLong:" + clickNumber.atomicLong.get());
    }
}
```

```

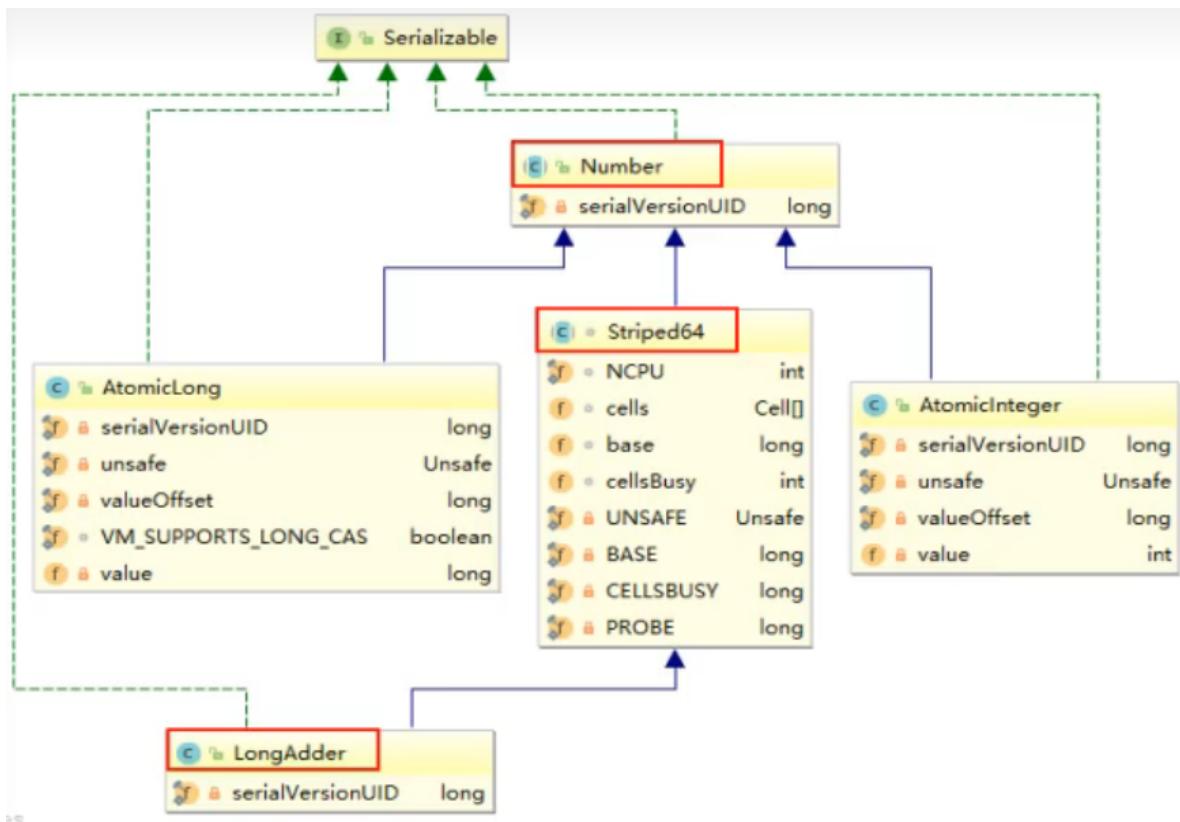
startTime = System.currentTimeMillis();
for (int i = 0; i < threadNumber; i++) {
    new Thread(() -> {
        try {
            for (int j = 0; j < 100 * _1w; j++) {
                clickNumber.clickByLongAdder();
            }
        } finally {
            countDownLatch3.countDown();
        }
    }, String.valueOf(i)).start();
}
countDownLatch3.await();
endTime = System.currentTimeMillis();
System.out.println("----constTime:" + (endTime - startTime) + "毫秒 \t"
clickByLongAdder:" + clickNumber.longAdder.sum());

startTime = System.currentTimeMillis();
for (int i = 0; i < threadNumber; i++) {
    new Thread(() -> {
        try {
            for (int j = 0; j < 100 * _1w; j++) {
                clickNumber.clickByLongAccumulator();
            }
        } finally {
            countDownLatch4.countDown();
        }
    }, String.valueOf(i)).start();
}
countDownLatch4.await();
endTime = System.currentTimeMillis();
System.out.println("----constTime:" + (endTime - startTime) + "毫秒 \t"
clickByLongAccumulator:" + clickNumber.longAccumulator.get());
}
}

```

源码、原理分析

----constTime:2213毫秒 clickBySynchronized:50000000
 ----constTime:450毫秒 clickByAtomicLong:50000000
 ----constTime:95毫秒 clickByLongAdder:50000000
 ----constTime:87毫秒 clickByLongAccumulator:50000000



LongAdder是Striped64的子类

Striped64有个内部类Cell

LongAdder为什么快

LongAdder的基本思路就是**分散热点**，将value值分散到一个Cell数组中，不同线程会命中到数组的不同槽中，各个线程只对自己槽中的那个值进行CAS操作，这样热点就被分散了，冲突的概率就小很多。如果要获取真正的long值，只要将各个槽中的变量值累加返回。

sum0会将所有Cell数组中的value和base累加作为返回值，核心的思想就是将之前AtomicLong一个value的更新压力分散到多个value中去，**从而降级更新热点**。

内部有一个base变量，一个Cell[]数组

base变量：低并发，直接累加到该变量上

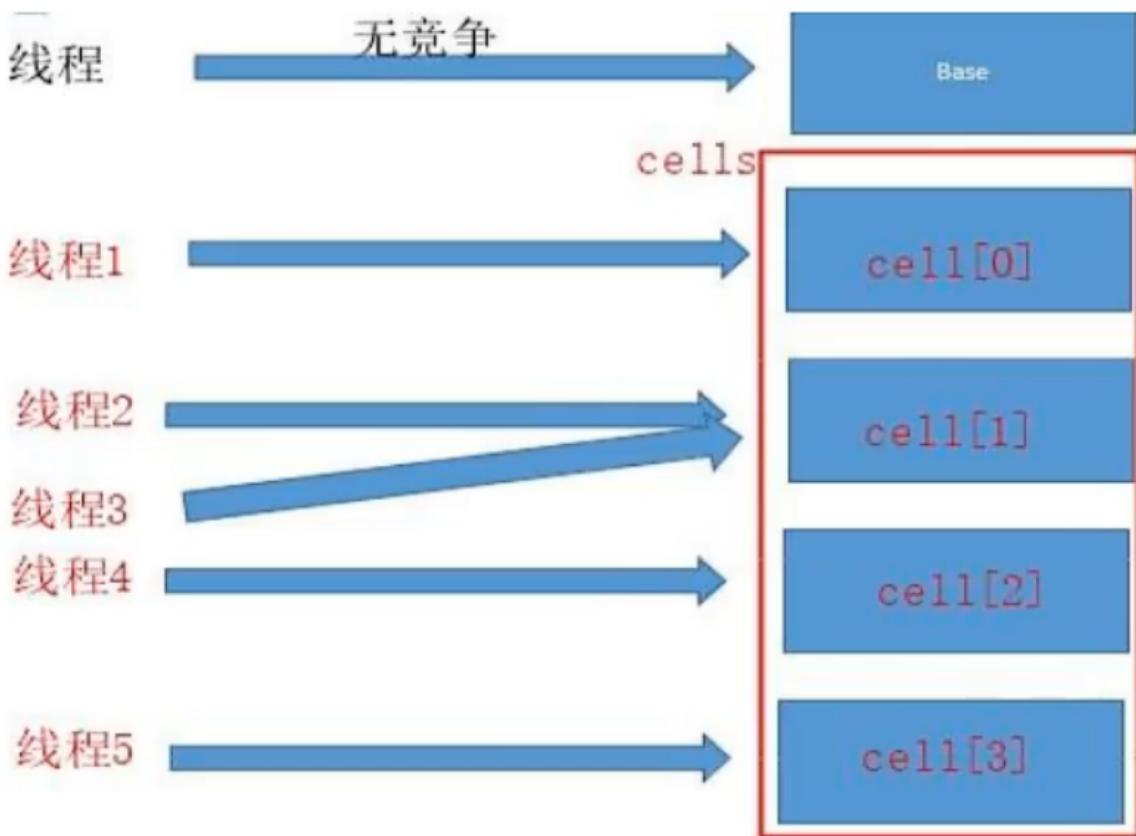
Cell[]数组：高并发，累加进各个线程自己的槽Cell[i]中

数学公式表达：

$$Value = Base + \sum_{i=0}^n Cell[i]$$

public void add(long x)

LongAdder在无竞争的情况下，跟AtomicLong一样，对同一个**base**进行操作，当出现竞争关系时则是采用**化整为零分散热点的做法**，用空回换时间用一个数组cells，将一个value拆分进这个数组cells。多个线程需要同时对value进行操作时候，可以对线程d进行hash得到hash值，再根据hash值映射到这个数组cells的某个下标，再对该下标所对应的值进行自增操作。当所有线程操作完毕，将数组cells的所有值和base都加起来作为最终结果。



```
public void increment() {
    add(1L);
}
```

```
public void add(long x) {
    Cell[] as; long b, v; int m; Cell a;
    if ((as = cells) != null || !casBase(b = base, b + x)) {
        boolean uncontended = true;
        if (as == null || (m = as.length - 1) < 0 ||
            (a = as[getProbe() & m]) == null ||
            !(uncontended = a.cas(v = a.value, v + x)))
            longAccumulate(x, null, uncontended);
    }
}
```

- as表示cells引用
 - b表示获取的base值
 - v表示期望值
 - m表示cells数组的长度
 - a表示当前线程命中的cell单元格
-
- 最初无竞争时只更新base
 - 如果更新base失败后，首次新建一个Cell[]数组
 - 当多个线程竞争同一个Cell比较激烈时，可能就要对Cell[]扩容

小总结

1. 如果Cells表为空，尝试用CAS更新base字段，成功则退出

- 如果Cells表为空， CAS更新base字段失败，出现竞争， uncontended为true，调用longAccumulate
 - 如果Cells表非空，但当前线程映射的槽为空， uncontended为true，调用longAccumulate
 - 如果Cells表非空，且当前线程映射的槽非空， CAS更新Cell的值，成功则返回，否则，uncontended设为false，调用longAccumulate

```
final void longAccumulate(long x, LongBinaryOperator fn, boolean  
wasUncontended)
```

```

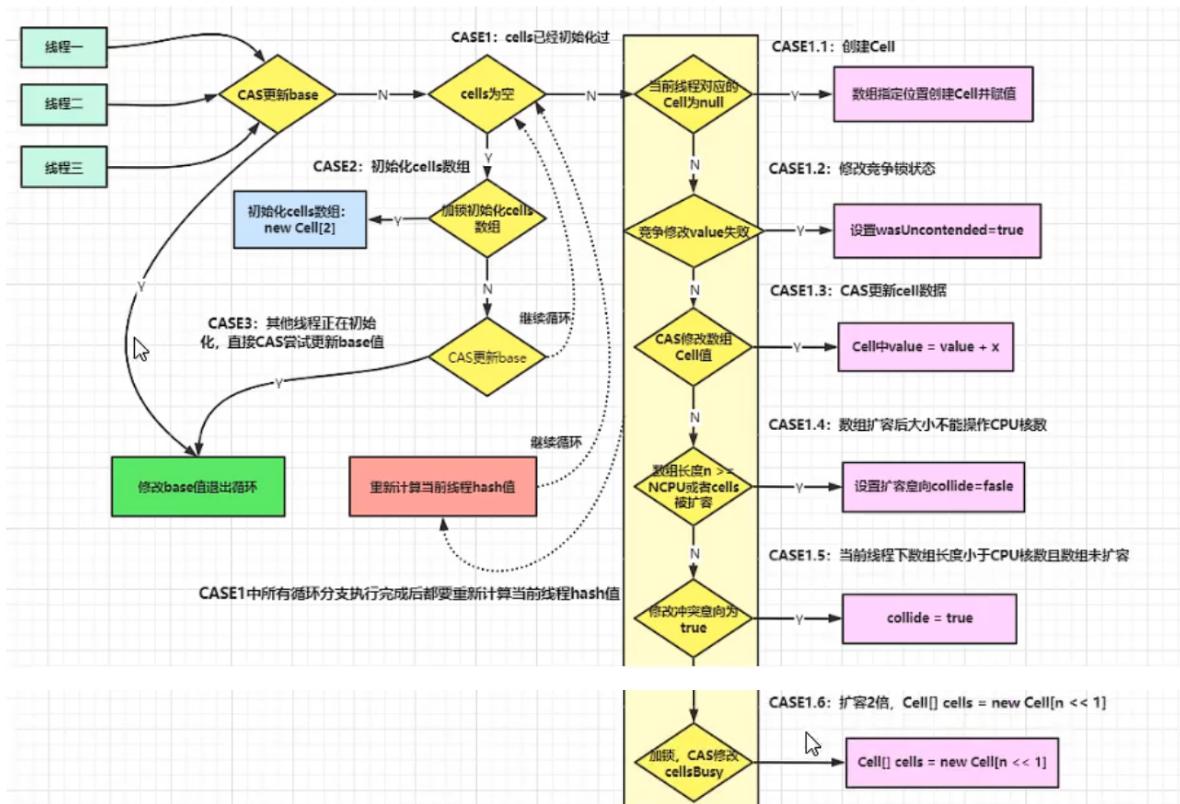
        for (int i = 0; i < n; ++i)
            rs[i] = as[i];
        cells = rs;
    }
} finally {
    cellsBusy = 0;
}
collide = false;
continue; // Retry with expanded table
}
h = advanceProbe(h);
}
else if (cellsBusy == 0 && cells == as && casCellsBusy()) {
    boolean init = false;
    try { // Initialize table
        if (cells == as) {
            Cell[] rs = new Cell[2];
            rs[h & 1] = new Cell(x);
            cells = rs;
            init = true;
        }
    } finally {
        cellsBusy = 0;
    }
    if (init)
        break;
}
else if (casBase(v = base, ((fn == null) ? v + x :
                           fn.applyAsLong(v, x))))
    break; // Fall back on using base
}
}

```

- long x 需要增加的值，一般默认都是1
- LongBinaryOperator fn 默认传递的是null
- wasUncontended竞争标识，如果是false则代表有竞争，只有cells初始化之后，并且当前线程 CAS竞争修改失败，才会使false

Striped64中一些变量或者方法的定义：

- base: 类似于AtomicLong中全局的value值。在没有竞争情况下数据直接累加到base上，或者cells扩容时，也需要将数据写入到base上
- collide: 表示扩容意向，false一定不会扩容，true可能会扩容。
- cellsBusy: 初始化cells或者扩容cells需要获取锁，0:表示无锁状态 1:表示其他线程已经持有了锁
 - casCellsBusy(): 通过CAS操作修改cellsBusy的值，CAS成功代表获取锁，返回true
- NCPU: 当前计算机CPU数量，Cell数组扩容时会使用到
- getProbe(): 获取当前线程的hash值
- advanceProbe(): 重置当前线程的hash值



public long sum()

```

public long sum() {
    Cell[] as = cells; Cell a;
    long sum = base;
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;
        }
    }
    return sum;
}

```

为什么在并发情况下这个值不准确

sum执行时，并没有限制对base和cells的更新(一句要命的话)。所以LongAdder不是强一致性的，它是最终一致性的。

首先，最终返回的sum局部变量，初始被复制为base，而最终返回时，很可能base已经被更新了，而此时局部变量sum不会更新，造成不一致。

其次，这里对cell的读取也无法保证是最后一次写入的值。所以，sum方法在没有并发的情况下，可以获得正确的结果。

总结

- AtomicLong
 - 线程安全，可允许一些性能损耗，要求高精度时可使用
 - 保证精度，性能代价
 - AtomicLong是多个线程针对单个热点值value进行原子操作
- LongAdder
 - 当需要在高并发下有较好的性能表现，且对值的精确度要求不高时，可以使用

- 保证性能，精度代价
- LongAdder是每个线程拥有自己的槽，各个线程一般只对自己槽中的那个值进行CAS操作

10. ThreadLocal

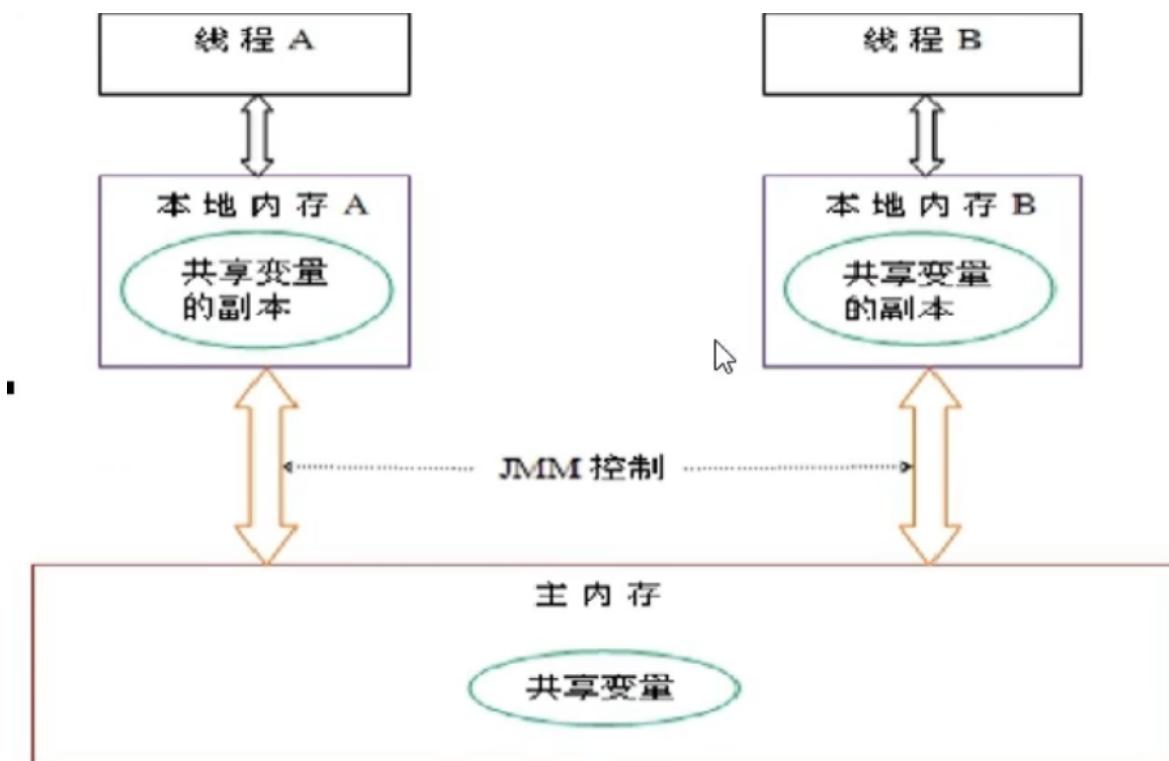
10.1 ThreadLocal简介

定义

ThreadLocal提供线程局部变量。这些变量与正常的变量不同，因为每一个线程在访问ThreadLocal实例的时候(通过其get或set方法)都有自己的、**独立初始化的变量副本**。ThreadLocal实例通常是类中的私有静态字段，使用它的目的是希望将状态(例如，用ID或事务ID)与线程关联起来。

实现**每一个线程都有自己专属的本地变量副本**(自己用自己的变量不麻烦别人，不和其他人共享，人人有份，人各一份)。

主要解决了让每个线程绑定自己的值，通过使用get和set方法，获取默认值或将其值更改为当前线程所存的副本的值**从而避免了线程安全问题**。比如我们之前讲解的8锁案例，资源类是使用同一部手机，多个线程抢夺同一部手机使用，假如人手一份是不是天下太平??



案例

案例1：

```
/**
 * @Author cxl
 * @Date 29/5/2023 16:28
 * @ClassReference: com.cxl.juc.threadlocal.ThreadLocalDemo
 * @Description: 需求1：5个销售卖房子，集团高层只关心销售总量的准确统计数
 */
class House {
```

```

int saleCount = 0;

public synchronized void saleHouse() {
    ++saleCount;
}

/*ThreadLocal<Integer> saleVolume = new ThreadLocal<Integer>() {

    @Override
    protected Integer initialValue() {
        return super.initialValue();
    }
};*/
}

ThreadLocal<Integer> saleVolume = ThreadLocal.withInitial(() -> 0);

public void saleVolumeByThreadLocal() {
    saleVolume.set(1 + saleVolume.get());
}
}

public class ThreadLocalDemo {

    public static void main(String[] args) {
        House house = new House();

        for (int i = 0; i < 5; i++) {
            new Thread(() -> {
                int size = new Random().nextInt(5)+1;
//                System.out.println(size);

                try {
                    for (int j = 0; j < size; j++) {
                        house.saleHouse();
                        house.saleVolumeByThreadLocal();
                    }
                    System.out.println(Thread.currentThread().getName() + "\t 号
销售卖出: " + house.saleVolume.get());
                } finally {
                    house.saleVolume.remove();
                }
            }, String.valueOf(i)).start();
        }

        try { TimeUnit.MILLISECONDS.sleep(300); }catch (InterruptedException e){
e.printStackTrace(); }

        System.out.println(Thread.currentThread().getName() + "\t 共计卖出多少套: "
+ house.saleCount);
    }
}

```

案例2:

```

class MyData {

    ThreadLocal<Integer> threadLocalField = ThreadLocal.withInitial(() -> 0);
}

```

```

public void add() {
    threadLocalField.set(1 + threadLocalField.get());
}

public class ThreadLocalDemo2 {

    public static void main(String[] args) {
        MyData myData = new MyData();
        ExecutorService threadPool = Executors.newFixedThreadPool(3);

        try {
            for (int i = 0; i < 10; i++) {
                threadPool.submit(() -> {
                    try {
                        Integer beforeInt = myData.threadLocalField.get();
                        myData.add();
                        Integer afterInt = myData.threadLocalField.get();
                        System.out.println(Thread.currentThread().getName() +
                                "\tbeforeInt: " + beforeInt + "\tafterInt: " + afterInt);
                    } finally {
                        myData.threadLocalField.remove();
                    }
                });
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            threadPool.shutdown();
        }
    }
}

```

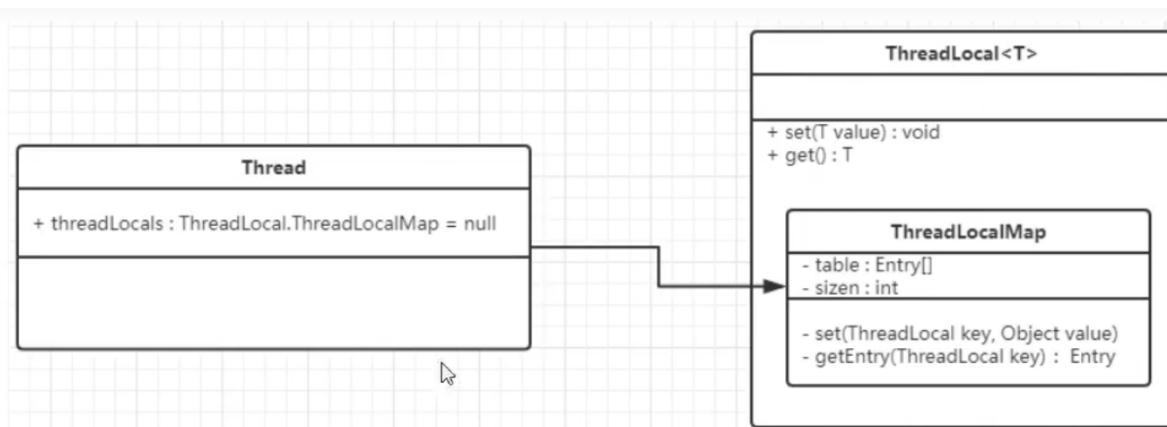
总结

因为每个 Thread 内有自己的**实例副本**且该副本只由当前线程自己使用

既然其它 Thread 不可访问，那就不存在多线程间共享的问题

统一设置初始值，但是每个线程对这个值的修改都是各自线程互相独立的

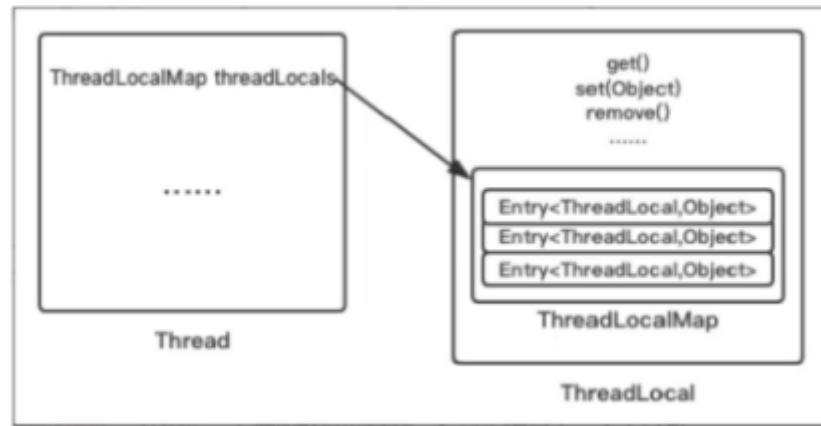
10.2 ThreadLocal源码分析



`threadLocalMap`实际上就是一个以`threadLocal`实例为key，任意对象为value的`Entry`对象。

当我们为threadLocal变量赋值，实际上就是以当前threadLocal实例为key，值为value的Entry往这个threadLocalMap中存放

ThreadLocaMap从字面上就可以看出这是一个保存ThreadLoca对象的m(其实是以ThreadLocal为Key，不过是经过了两层包装的ThreadLocal对象



JVM内部维护了一个线程版的Map<ThreadLocal.Value>(通过ThreadLoca对象的set方法, 结果把ThreadLocal对象自己当做key放进了ThreadLoalMap中), 每个线程要用到这个T的时候, 用当前的线程去Map里面获取, 通过这样让每个线程都拥有了自己独立的变量, 人手一份, 竞争条件被彻底消除, 在并发模式下是绝对安全的变量。

10.3 ThreadLocal内存泄漏问题

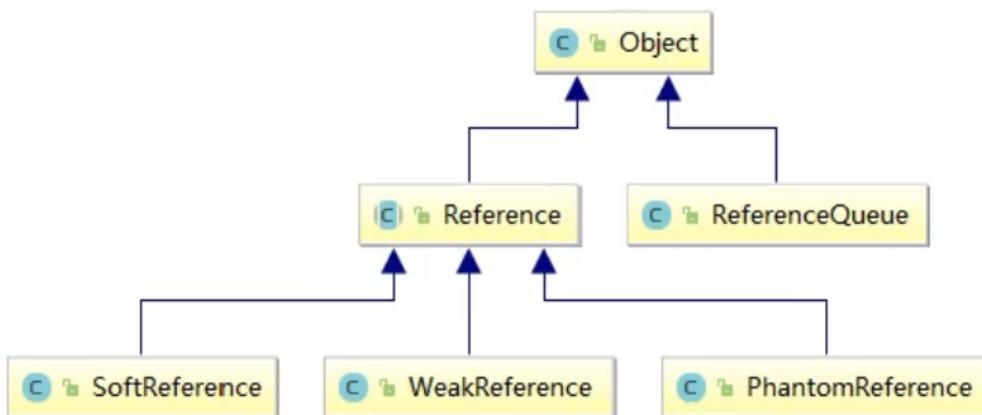
不再会被使用的对象或者变量占用的内存不能被回收，就是内存泄露

ThreadLocalMap与WeakReference

ThreadLocalMap从字面上就可以看出这是一个保存ThreadLocal对象的map(以ThreadLocal为Key), 不过是经过了两层包装的ThreadLocal对象:

- 第一层包装是使用 WeakReference<ThreadLoca?>> 将ThreadLocal对象变成一个弱引用的对象;
- 第二层包装是定义了一个专门的类 Entry 来扩展 WeakReference<ThreadLocal<?>>

强引用、软引用、弱引用、虚引用



Java 技术允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。

强引用

当内存不足，JVM开始垃圾回收，对于强引用的对象，就算是出现了OOM也不会对该对象进行回收，死都不收。

强引用是我们最常见的普通对象引用，只要还有强引用指向一个对象，就能表明对象还“活着”，垃圾收集器不会碰这种对象。

在Java中最常见的就是强引用，把一个对象赋给一个引用变量，这个引用变量就是一个强引用。

当一个对象被强引用变量引用时，它处于可达状态，它是不可能被垃圾回收机制回收的，

即使该对象以后永远都不会被用到，JVM也不会回收。因此强引用是造成Java内存泄漏的主要原因之一。

对于一个普通的对象，如果没有其他的引用关系，只要超过了引用的作用域或者显式地将相应(强)引用赋值为null。

般认为就是可以被垃圾收集的了(当然具体回收时机还是要看垃圾收集策略)。

```
/**  
 * 输出结果：  
 * gc before:com.cxl.juc.threadlocal.MyObject@1b6d3586  
 * -----invoke finalize method!  
 * gc after:null  
 */  
private static void strongReference() {  
    MyObject myObject = new MyObject();  
    System.out.println("gc before:" + myObject);  
  
    myObject = null;  
    System.gc(); // 人工开启GC  
  
    // 暂停1秒钟线程  
    try { TimeUnit.MILLISECONDS.sleep(500); } catch (InterruptedException e){  
        e.printStackTrace();  
    }  
  
    System.out.println("gc after:" + myObject);  
}
```

软引用

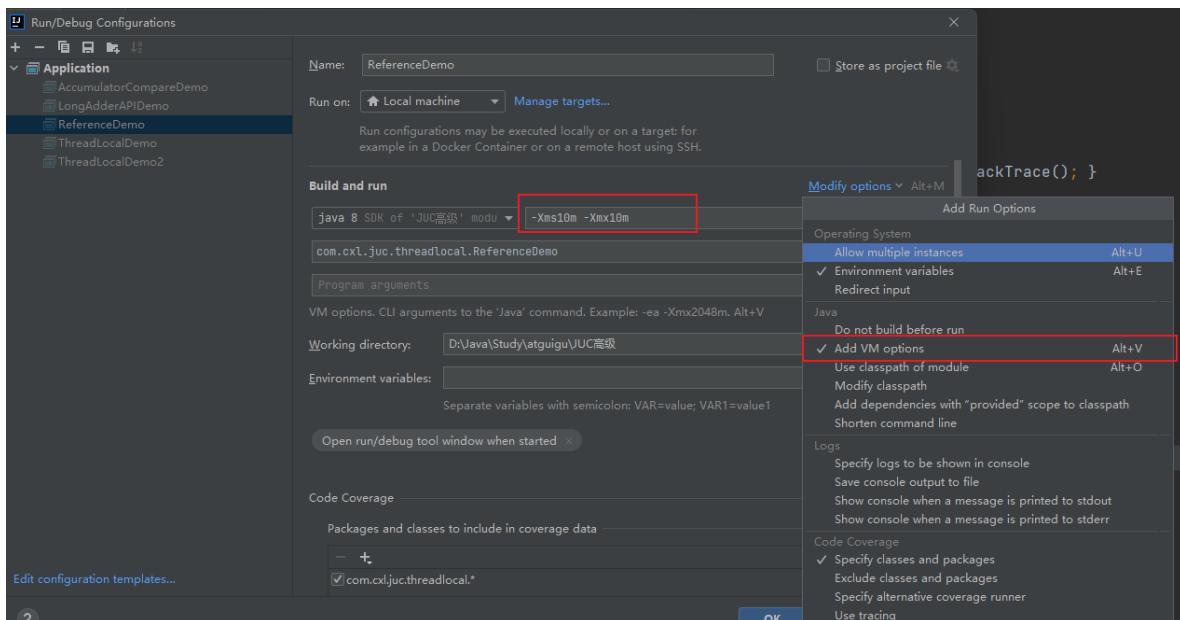
软引用是一种相对强引用弱化了一些的引用，需要用java.lang.ref.SoftReference类来实现，可以让对象豁免一些垃圾收集。

对于只有软引用的对象来说

- 当系统内存充足时它不会被回收
- 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，比如高速缓存就有用到软引用，内存够用的时候就保留，不够用就回收

进行内存配置



```
/**
 * 输出结果:
 * -----gc after 内存够用: com.cxl.juc.threadlocal.MyObject@1b6d3586
 * -----gc after内存不够用: null
 * -----invoke finalize method!
 * Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
 *     at com.cxl.juc.threadlocal.ReferenceDemo.main(ReferenceDemo.java:36)
 */
private static void softReference() {
    SoftReference<MyObject> softReference = new SoftReference<>(new MyObject());
    // System.out.println("-----softReference" + softReference.get());
    System.gc();
    try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e){
    e.printStackTrace(); }
    System.out.println("-----gc after 内存够用: " + softReference.get());

    try {
        byte[] bytes = new byte[20 * 1024 * 1024]; // 20MB对象
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        System.out.println("-----gc after内存不够用: " + softReference.get());
    }
}
```

弱引用

弱引用需要用java.lang.ref.WeakReference类来实现，它比软引用的生存期更短

对于只有弱引用的对象来说，只要垃圾回收机制一运行，不管JVM的内存空间是否足够，都会回收该对象占用的内存。

```
/**
 * 输出结果:
 * -----gc before 内存够用: com.cxl.juc.threadlocal.MyObject@1b6d3586
 * -----invoke finalize method!
 * -----gc after内存够用: null
 */
private static void weakReference() {
```

```

WeakReference<MyObject> weakReference = new WeakReference<>(new MyObject());
System.out.println("----gc before 内存够用: " + weakReference.get());

System.gc();
// 暂停1秒钟线程
try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e){
e.printStackTrace();
}

System.out.println("----gc after内存够用: " + weakReference.get());
}

```

软引用和弱引用使用场景

假如有一个应用需要读取大量的本地图片

- 如果每次读取图片都从硬盘读取则会严重影响性能
- 如果一次性全部加载到内存中又可能造成内存溢出

此时使用软引用可以解决这个问题。

设计思路是: 用一个HashMap来保存图片的路径和相应图片对象关联的软引用之间的映射关系, 在内存不足时, JVM会自动回收这些缓存图片对象所占用的空间, 从而有效地避免了OOM的问题。

```

Map<String, SoftReference<Bitmap>> imageCache = new HashMap<String,
SoftReference<Bitmap>>();

```

虚引用

1. 虚引用必须和引用队列(ReferenceQueue)联合使用

- 虚引用需要java.lang.ref.PhantomReference类来实现, 顾名思义, **就是形同虚设**, 与其他几种引用都不同, 虚引用并不会决定对象的生命周期。如果一个对象仅持有引用, 那么它就和没有任何引用一样, 在任何时候都可能被垃圾回收器回收, 它不能单独使用也不能通过它访问对象, **虚引用必须和引用队列(ReferenceQueue)联合使用**。

2. PhantomReference的get方法总是返回null

- 虚引用的主要作用是跟踪对象被垃圾回收的状态。仅仅是提供了一种确保对象被 finalize以后, 做某些事情的通知机制。
PhantomReference的get方法总是返回null, 因此无法访问对应的引用对象。

3. 处理监控通知使用

- 换句话说, 设置虚引用关联对象的唯一目的, 就是在这个对象被收集器回收的时候收到一个系统通知或者后续添加进一步的处理, 用来实现比finalize机制更灵活的回收操作

与上方一致, 配置内存大小为10m

```

/**
 * 输出结果:
 * null      list add ok
 * -----invoke finalize method!
 * null      list add ok
 * Exception in thread "t1" java.lang.OutOfMemoryError: Java heap space
 *      at
com.cxl.juc.threadlocal.ReferenceDemo.lambda$main$0(ReferenceDemo.java:39)

```

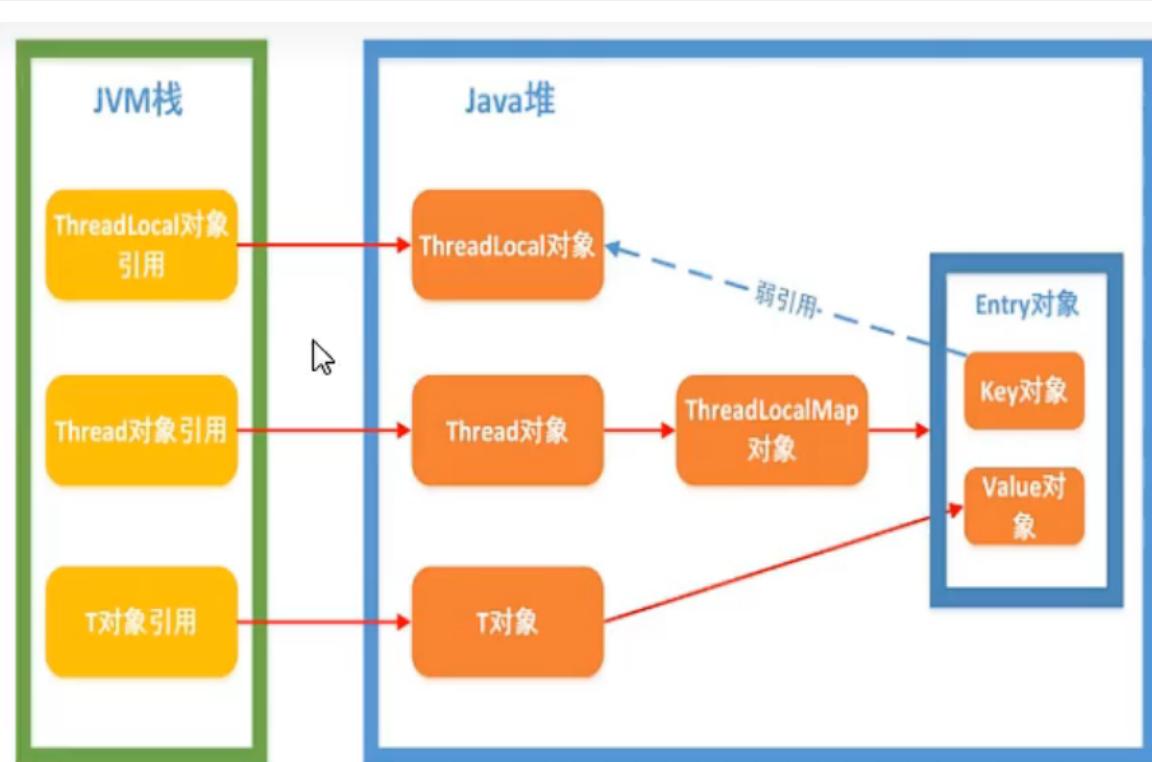
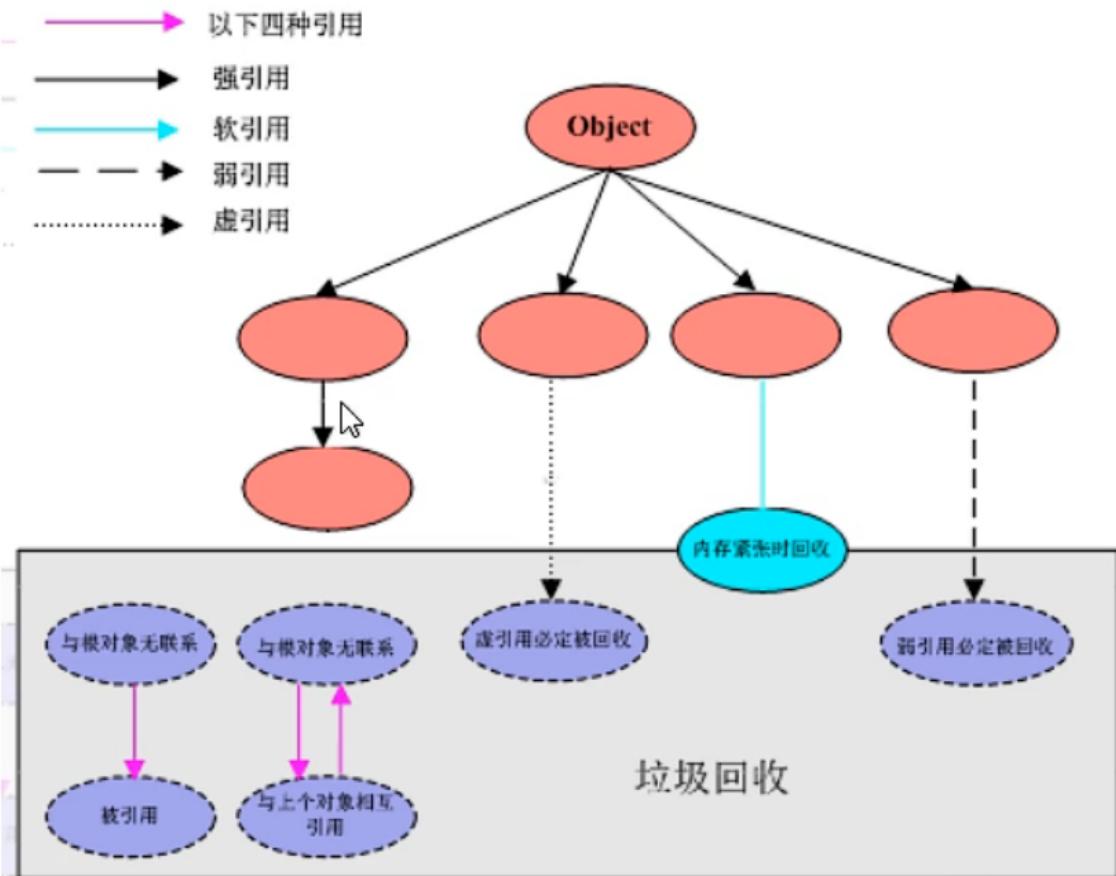
```
*      at
com.cxl.juc.threadlocal.ReferenceDemo$$Lambda$1/1324119927.run(Unknown Source)
*      at java.lang.Thread.run(Thread.java:748)
* ----有虚对象回收加入了队列
*
* Process finished with exit code 0
*/
private static void phantomReference() {
    MyObject myObject = new MyObject();
    ReferenceQueue<MyObject> referenceQueue = new ReferenceQueue<>();
    PhantomReference<MyObject> phantomReference = new PhantomReference<>
(myObject, referenceQueue);
    //          System.out.println(phantomReference.get());

    List<byte[]> list = new ArrayList<>();

    new Thread(() -> {
        while (true) {
            list.add(new byte[1 * 1024 * 1024]);
            try { TimeUnit.MILLISECONDS.sleep(500); }catch (InterruptedException e){ e.printStackTrace(); }
            System.out.println(phantomReference.get() + "\t list add ok");
        }
    }, "t1").start();

    new Thread(() -> {
        while (true) {
            Reference< extends MyObject> reference = referenceQueue.poll();
            if (reference != null) {
                System.out.println("----有虚对象回收加入了队列");
                break;
            }
        }
    }, "t1").start();
}
```

GCRoots和四大引用小总结



ThreadLoca是一个壳子，真正的存储结构是ThreadLocal里有ThreadLocalMap这么个内部类，每个Thread对象维护着一个ThreadLocalMap的引用，ThreadLocalMap是ThreadLocal的内部类，用Entry来进行存储。

1. 调用ThreadLocal的set()方法时，实际上就是往ThreadLocalMap设置值，key是ThreadLocal对象，值Value是传递进来的对象
2. 调用ThreadLocal的get()方法时，实际上就是往ThreadLocalMap获取值，key是ThreadLocal对象

ThreadLocal本身并不存储值(ThreadLoca是一个壳子), 它只是自己作为一个key来让线程从ThreadLocalMap获取value正因为这个原理, 所以ThreadLocal能够实现“数据隔离”, 获取当前线程的局部变量值, 不受其他线程影响

为什么使用弱引用

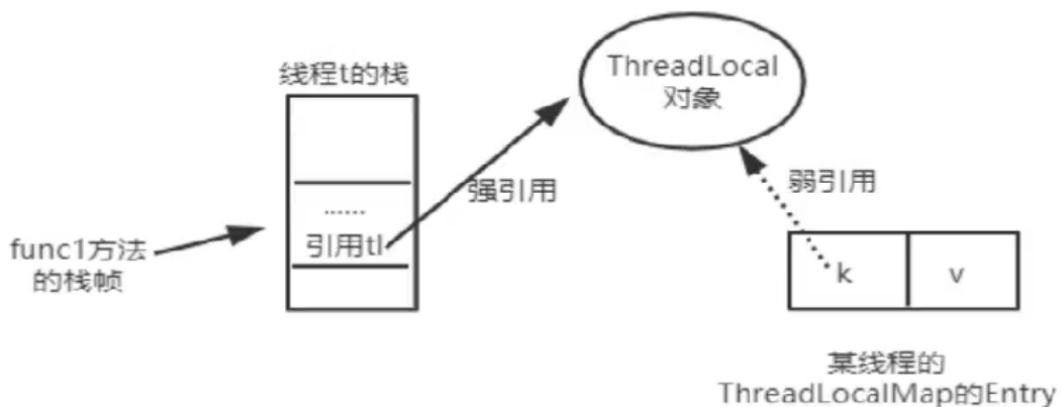
```
public void function01() {  
    ThreadLocal<String> t1 = new ThreadLocal<>();  
    t1.set("123123123");  
    t1.get();  
}
```

当function01方法执行完毕后, 栈销毁强引用 t1 也就没有了。但此时线程的ThreadLocalMap里某个entry的key引用还指向这个对象

若这个key引用是强引用, 就会导致key指向的ThreadLoca对象及v指向的对象不能被gc回收, 造成内存泄漏;

若这个key引用是弱引用就大概率会减少内存泄漏的问题

就可以使ThreadLocal对象在方法执行完毕后顺利被回收且**Entry的key引用指向为null**。



当key为null时

- 当我们为threadLocal变量赋值, 实际上就是当前的Entrv/threadLocal实例为key, 值为value)往这个threadLocalMap中存放。Entry中的key是弱引用, 当threadLocal外部强引用被置为null(tl=null), 那么系统 GC 的时候, 根据可达性分析, 这个threadLocal实例就没有任何一条路能够引用到它, 这个ThreadLocal势必会被回收。这样以来, ThreadLocalMap中就会出现key为null的Entry, 就没有办法访问这些key为nu的Entry的value.如果当前线程再迟迟不结束的话, 这些key为null的Entryvalue就会一直存在一条强引用: Thread Ref-> Thread -> ThreaLocalMap -> Entryvalue永远无法回收, 造成内存泄漏。
- 当然, 如果当前thread运行结束, threadLocal, threadLocalMap, Entry没有引用链可达, 在垃圾回收的时候都会被系统进行回收。
- 但在实际使用中我们有时候会用线程池去维护我们的线程, 比如在 Executors.newFixedThreadPool()时创建线程的时候, 为了复用线程是不会结束的, 所以 threadLocal内存泄漏就值得我们小心

ThreadLocalMap使用ThreadLocal的弱引用作为key, 如果一个ThreadLocal没有外部强引用引用他, 那么系统gc的时候, 这ThreadLocal势必会被回收, 这样一来, ThreadLocalMap中就会出现**key为null的Entry**, 就没有办法访问这些key为null的Entry的value, 如果当前线程再迟迟不结束的话(比如正好用在线程池), 这些key为null的Entry的value就会一直存在一条强引用链。

虽然弱引用，保证了key指向的ThreadLocal对象能被及时回收，但是v指向的value对象是需要ThreadLocalMap调用get、set时发现key为null时才会去回收整个entry、value，因此弱引用不能100%保证内存不泄露。我们要在不使用某个ThreadLocal对象后，手动调用remove方法来删除它，尤其是在在线程池中，不仅仅是内存泄露的问题，因为线程池中的线程是重复使用的，意味着这个线程的ThreadLocalMap对象也是重复使用的，如果我们不手动调用remove方法，那么后面的线程就有可能获取到上个线程遗留下来的value值，造成bug。

从前面的set, getEntryremove方法看出，在threadLocal的生命周期里，针对threadLocal存在的内存泄漏的问题，都会通过expungeStaleEntry, cleanSomeSlots, replaceStaleEntry这三个方法清理掉key为null的脏entry。

10.4 总结

使用总结：

- ThreadLocal.withInitial(() -> 初始化值);
- 建议把ThreadLocal修饰为static
- 用完记得手动remove

功能总结：

- ThreadLocal 并不解决线程间共享数据的问题
- ThreadLocal 适用于变量在线程间隔离且在方法间共享的场景
- ThreadLocal 通过隐式的在不同线程内创建独立实例副本避免了实例线程安全的问题
- 每个线程持有一个只属于自己的专属Map并维护了ThreadLocal对象与具体实例的映射该Map由于只被持有它的线程访问，故不存在线程安全以及锁的问题
- ThreadLocalMap的Entry对ThreadLocal的引用为弱引用，避免了ThreadLocal对象无法被回收的问题
- 都会通过expungeStaleEntry, cleanSomeSlots, replaceStaleEntry这三个方法回收键为 null 的Entry对象的值(即为具体实例，以及 Entry 对象本身从而防止内存泄漏属于安全加固的方法)

11. Java对象内存布局和对象头

在HotSpot虚拟机里，对象在堆内存中的存储布局可以划分为三个部分：对象头(Header)、实例数据(Instance Data) 和对齐填充(Padding)。



对象内部结构分为：对象头、实例数据、对齐填充（保证8个字节的倍数）

11.1 对象头

- 对象标记Mark Word
- 类元信息klassOop（又叫类型指针），类元信息存储的是指向该对象类元数据（klass）的首地址

对象标记



HotSpot虚拟机对象头Mark Word

存储内容	标志位	状态
对象哈希码、对象分代年龄	01	未锁定
指向锁记录的指针	00	轻量级锁定
指向重量级锁的指针	10	膨胀(重量级锁定)
空，不需要记录信息	11	GC 标记
偏向线程ID、偏向时间截、对象分代年龄	01	可偏向

在64位系统中，Mark Word占了8个字节，类型指针占了8个字节，一共是16个字节

表2-5 Mark Word的存储结构

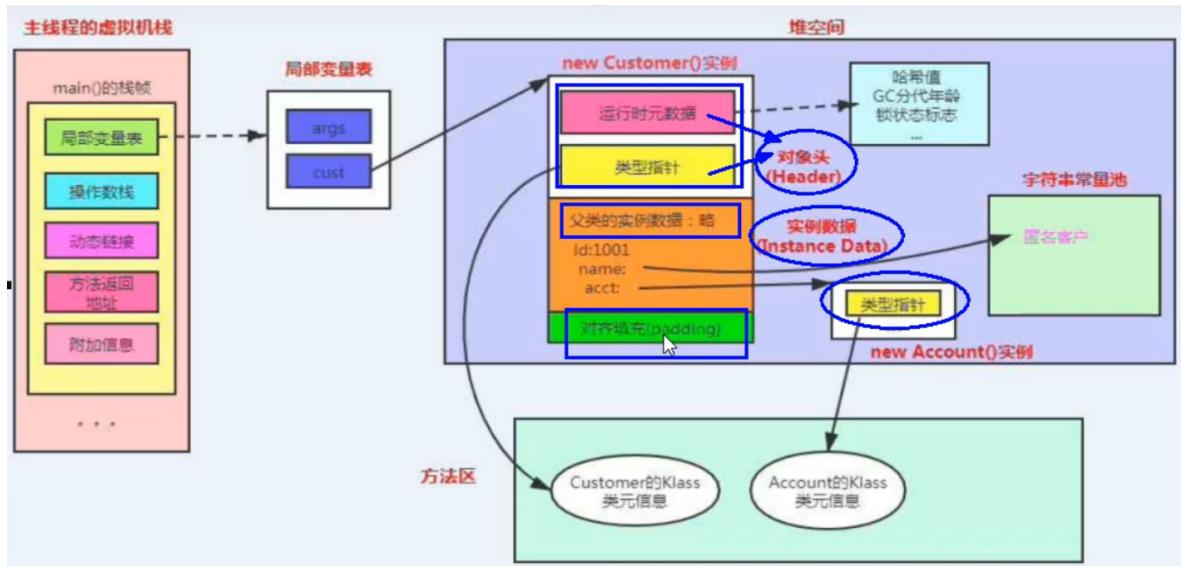
锁状态	25bit	31bit	1bit	4bit	1bit	2bit
			cms_free	分代年龄	偏向锁	锁标志位
无锁	unused	hashCode			0	01
偏向锁	ThreadID(54bit)	Epoch(2bit)			1	01

默认存储对象的HashCode、分代年龄和锁标志位等信息。

这些信息都是与对象自身定义无关的数据卷，所以MarkWord被设计成一个非固定的数据结构以便在极小的控件内存存储尽量多的数据。

它会根据对象的状态复用自己的存储空间，也就是说运行期间MarkWord里存储的数据会随着锁标志位的变化而变化

类元信息



对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例

11.2 实例数据

```
// 只有一个对象的实例对象，16字节（忽略压缩指针的影响）+4字节（int）+1字节（boolean）=21字节
--> 对齐填充，24字节
class Custom {
    int id;
    boolean flag = false;
}
```

存放类的属性（Field）数据信息，包括父类的属性信息

11.3 对齐填充

虚拟机要求对象起始地址必须是8字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐这部分内存按8字节补充对齐

11.4 MarkWord源码解析

```
64 bits:
-----
unused:25 hash:31 -->| unused:1   age:4    biased_lock:1 lock:2 (normal object)
JavaThread*:54 epoch:2 unused:1   age:4    biased_lock:1 lock:2 (biased object)
PromotedObject*:61 ----->| promo_bits:3 ----->| (CMS promoted object)
size:64 ----->| (CMS free block)

unused:25 hash:31 -->| cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && normal object)
JavaThread*:54 epoch:2 cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && biased object)
narrowOp:32 unused:24 cms_free:1 unused:4 promo_bits:3 ----->| (COOPs && CMS promoted object)
unused:21 size:35 -->| cms_free:1 unused:7 ----->| (COOPs && CMS free block)
```

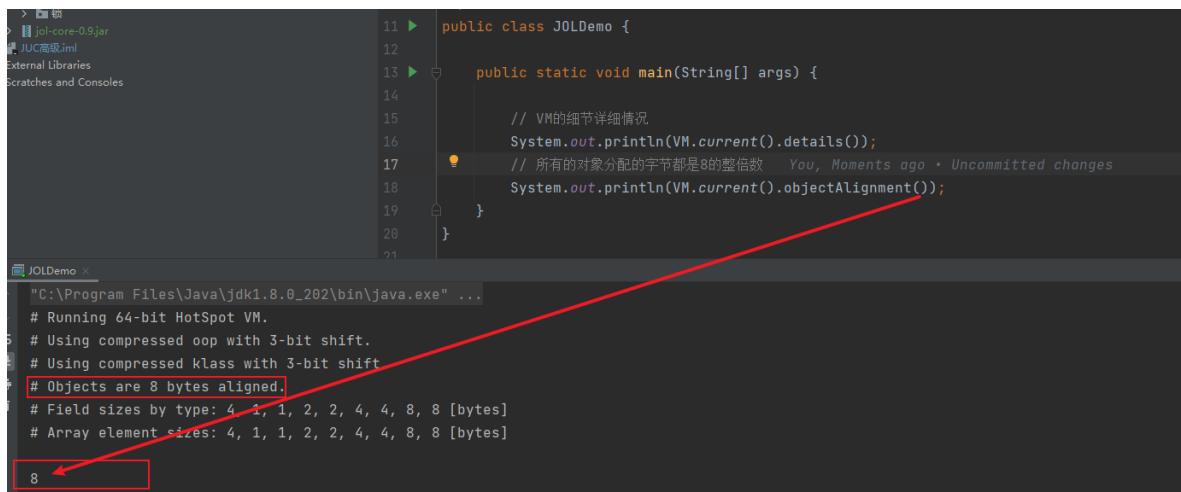
hash: 保存对象的哈希码
age: 保存对象的分代年龄
biased_lock: 偏向锁标识位
lock: 锁状态标识位
JavaThread*: 保存持有偏向锁的线程ID
epoch: 保存偏向时间戳

11.5 JOL证明

JOL: 是一个工具，分析对象在JVM的大小和分布

JOL (Java Object Layout) is the tiny toolbox to analyze object layout schemes in JVMs. These tools are using Unsafe, JVMTI, and Serviceability Agent (SA) heavily to decode the *actual* object layout, footprint, and references. This makes JOL much more accurate than other tools relying on heap dumps, specification assumptions, etc.

去maven仓库下载 jol-core 0.9version

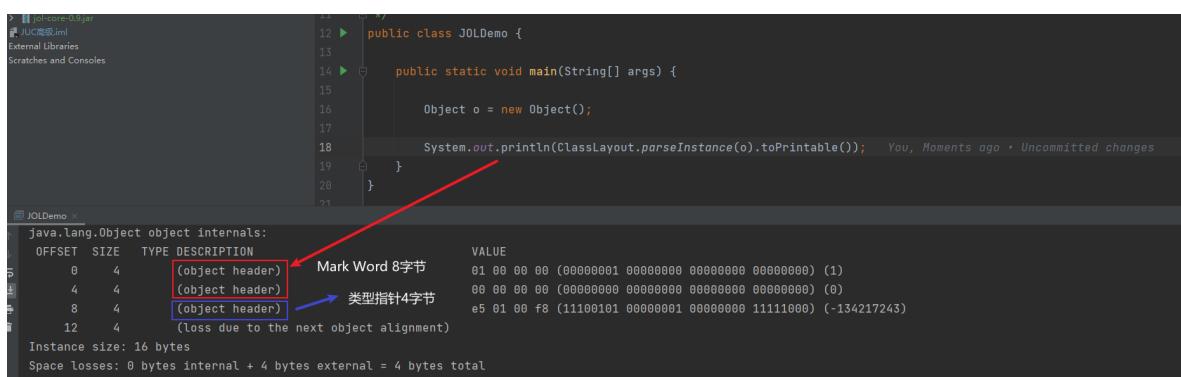


```

11  public class JOLDemo {
12
13  public static void main(String[] args) {
14
15      // VM的细节详细情况
16      System.out.println(VM.current().details());
17      // 所有的对象分配的字节都是8的倍数 You, Moments ago + Uncommitted changes
18      System.out.println(VM.current().objectAlignment());
19  }
20
21
  
```

"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...
Running 64-bit HotSpot VM.
Using compressed oop with 3-bit shift.
Using compressed klass with 3-bit shift
Objects are 8 bytes aligned.
Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

8



OFFSET	SIZE	TYPE	DESCRIPTION	VALUE
0	4	(object header)		01 00 00 00 (00000001 00000000 00000000 00000000) (1)
4	4	(object header)		00 00 00 00 (00000000 00000000 00000000 00000000) (0)
8	4	(object header)		e5 01 00 f8 (11100101 00000001 00000000 11110000) (-134217243)
12	4		(loss due to the next object alignment)	

Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

- **OFFSET:** 偏移量，也就是到这个字段位置所占用的byte数
- **SIZE:** 后面类型的字节大小
- **TYPE:** 是Class中定义的类型
- **DESCRIPTION:** DESCRIPTION是类型的描述

- VALUE: VALUE是TYPE在内存中的值

```

> JOLDemo.x
> jdk-core-0.9.jar
JUC高级.aiml
Internal Libraries
Watchers and Consoles

Customer c1 = new Customer();
System.out.println(ClassLayout.parseInstance(c1).toPrintable());

class Customer {
    // 第一种情况, 只有对象头, 没有任何其它实例数据
    int id;
    boolean flag = false;
}

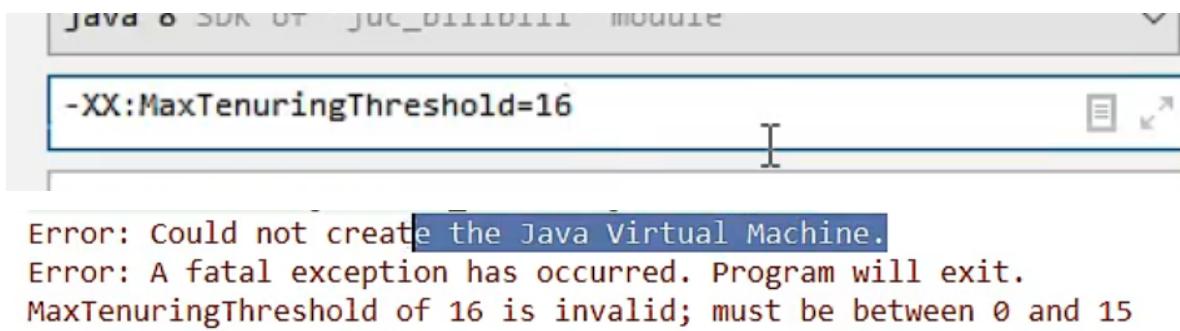
C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...
com.cxl.juc.objecthead.Customer object internals:
OFFSET  SIZE   TYPE DESCRIPTION          VALUE
     0     4   (object header)  01 00 00 00 (00000001 00000000 00000000 00000000) (1)
     4     4   (object header)  00 00 00 00 (00000000 00000000 00000000 00000000) (0)
     8     4   (object header)  43 c1 00 f8 (01000011 11000001 00000000 11111000) (-134168253)
    12     4   int Customer.id      0
    16     1   boolean Customer.flag  false
    17     7   (loss due to the next object alignment)

Instance size: 24 bytes
Space losses: 0 bytes internal + 7 bytes external = 7 bytes total

```

11.6 GC年龄

GC年龄采用4位bit存储，最大为15例如MaxTenuringThreshold参数默认值就是15



11.7 压缩指针

1. 默认配置，启动了压缩指针，-XX:+UseCompressedClassPointers
 $12+4(\text{对齐填充}) = \text{一个对象}16\text{个字节}$
2. 手动配置，关闭了压缩指针 -XX:-UseCompressedClassPointers
 $8+8=16\text{个字节}$

12. synchronized与锁升级

锁状态	25位	31位	1位	4bit	1bit 偏向锁位	2bit 锁标志位
无锁态 (new)	unused	hashCode (如果有调用)	unused	分代年龄	0	0 1
锁状态	54位	2位	1位	4bit	1bit 偏向锁位	2bit 锁标志位
偏向锁	当前线程指针 <code>JavaThread*</code>	Epoch	unused	分代年龄	1	0 1
锁状态	62位				2bit 锁标志位	
轻量级锁 自旋锁 无锁	指向线程栈中Lock Record的指针				0	0
重量级锁	指向互斥量 (重量级锁) 的指针				1	0
GC标记信息	CMS过程用到的标记信息				1	1

synchronized锁由对象头中的MarkWord根据锁标志位的不同而被复用及锁升级策略

锁升级过程

- 无锁
- 偏向锁
- 轻量级锁
- 重量级锁

```

285     ObjectMonitor* monitor() const {
286         assert(has_monitor(), "check");
287         // Use xor instead of &~ to provide one extra tag-bit check.
288         return (ObjectMonitor*) (value() ^ monitor_value);
289     }

```

Monitor可以理解为一种同步工具，也可理解为一种同步机制，常常被描述为一个Java对象。Java对象是天生的Monitor，每一个Java对象都有成为Monitor的潜质，因为在Java的设计中，每一个Java对象自打娘胎里出来就带了一把看不见的锁，它叫做内部锁或者Monitor锁。

```

public class MyObject
{
    public static void main(String[] args)
    {
        Object objectLock = new Object();
        new Thread(() -> {
            synchronized (objectLock) {
                // your code is here.....
            }
        }, name: "t1").start();
    }
}

```

Monitor的本质是依赖于底层操作系统的MutexLock实现，操作系统实现线程之间的切换需要从用户态到内核态的转换，成本非常高。

Mutex Lock

Monitor是在ivm底层实现的，底层代码是C++。本质是依赖于底层操作系统的Mutex Lock实现，操作系统实现线程之间的切换需要从用户态到内核态的转换，状态转换需要耗费很多的处理器时间成本非常高。**所以synchronized是Java语言中的一个重量级操作。**

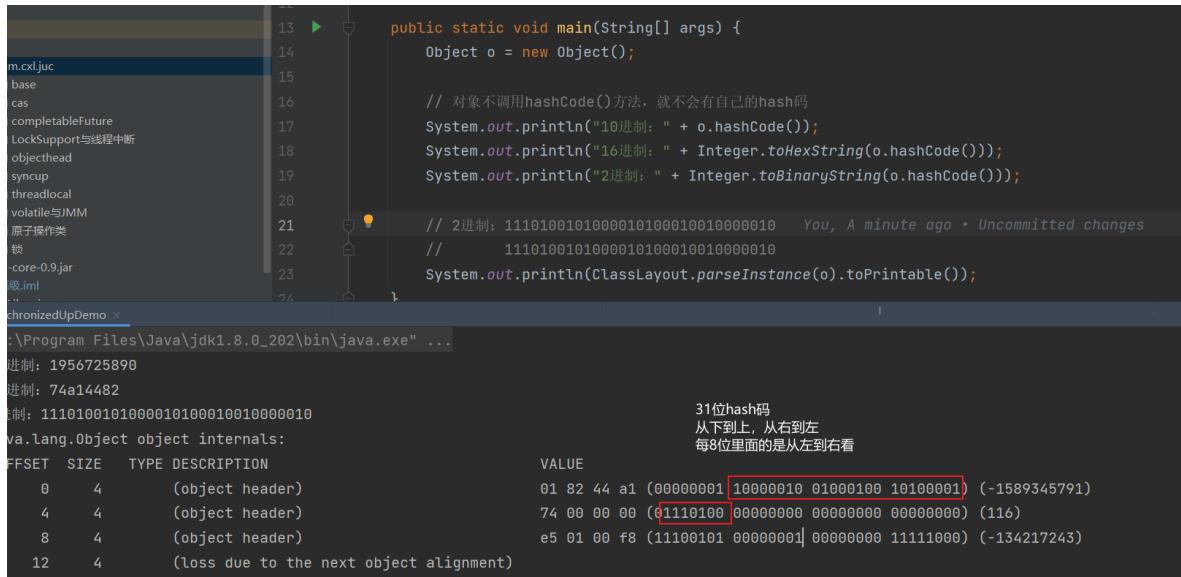
12.1 synchronized锁种类及升级步骤

锁指向

- 偏向锁: MarkWord存储的是偏向的线程ID;
- 轻量锁: MarkWord存储的是指向线程栈中Lock Record的指针;
- 重量锁:MarkWord存储的是指向堆中的monitor对象的指针;

无锁

无锁:初始状态，一个对象被实例化后，如果还没有被任何线程竞争锁，那么它就为无锁状态(001)



```
public static void main(String[] args) {
    Object o = new Object();

    // 对象不调用hashCode()方法，就不会有自己的hashCode
    System.out.println("10进制: " + o.hashCode());
    System.out.println("16进制: " + Integer.toHexString(o.hashCode()));
    System.out.println("2进制: " + Integer.toBinaryString(o.hashCode()));

    // 2进制: 11101001010000101000100010000010 You, A minute ago * Uncommitted changes
    // 11101001010000101000100010000010
    System.out.println(ClassLayout.parseInstance(o).toPrintable());
}

进制: 1956725890
进制: 74a14482
转制: 11101001010000101000100010000010
va.Lang.Object object internals:
FFSET SIZE TYPE DESCRIPTION VALUE
0 4 (object header) 01 82 44 a1 (00000001 10000010 01000100 10100001) (-1589345791)
4 4 (object header) 74 00 00 00 (11101001 00000000 00000000 00000000) (116)
8 4 (object header) e5 01 00 f8 (11100101 00000001 00000000 11110000) (-134217243)
12 4 (loss due to the next object alignment)
```

12.2 偏向锁

偏向锁：单线程竞争

当线程A第一次竞争到锁时，通过操作修改Mark Word中的偏向线程ID、偏向模式。

如果不存在其他线程竞争，那么持有偏向锁的线程将永远不需要进行同步。

当一段同步代码一直被同一个线程多次访问，由于只有一个线程那么该线程在后续访问时便会自动获得锁

Hotspot 的作者经过研究发现，大多数情况下：

多线程的情况下，锁不仅不存在多线程竞争，还存在锁由同一个线程多次获得的情况

偏向锁就是在这种情况下出现的，它的出现是为了解决**只有在一个线程执行同步时提高性能**

备注：偏向锁会偏向于第一个访问锁的线程，如果在接下来的运行过程中，该锁没有被其他的线程访问，则持有偏向锁的线程将永远不需要触发同步。也即偏向锁在资源没有竞争情况下消除了同步语句，懒的连CAS操作都不做了，直接提高程序性能

说明

在实际应用运行过程中发现，“锁总是同一个线程持有，很少发生竞争”，也就是说**锁总是被第一个占用他的线程拥有，这个线程就是锁的偏向线程。**

那么只需要在锁第一次被拥有的时候，记录下偏向线程ID。这样偏向线程就一直持有锁(后续这个线程进入和退出这段加了同步锁的代码块时，**不需要再次加锁和释放锁**。而是直接会去检查锁的MarkWord里面是不是放的自己的线程ID)。

如果相等，表示偏向锁是偏向于当前线程的，就不需要再尝试获得锁了，直到竞争发生才释放锁。以后每次同步，检查锁的偏向线程ID与当前线程ID是否一致，如果一致直接进入同步。无需每次加锁解锁都去CAS更新对象头。**如果自始至终使用锁的线程只有一个**，很显偏向锁几乎没有额外开销，性能极高。

如果不等，表示发生了竞争，锁已经不是总是偏向于同一个线程了，这个时候会尝试使用CAS来替换MarkWord里面的线程ID为新线程的ID，

竞争成功，表示之前的线程不存在了，MarkWord里面的线程ID为新线程的ID，锁不会升级，仍然为偏向锁。

竞争失败，这时候可能需要升级变为轻量级锁，才能保证线程间公平竞争锁。

注意，偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，线程是不会主动释放偏向锁的。

一个synchronized方法被一个线程抢到了锁时，那这个方法所在的对象就会在其所在的Mark Word中将偏向锁修改状态位，同时还会有占用前54位来存储线程指针作为标识。若该线程再次访问同一个synchronized方法时，该线程只需去对象头的Mark Word中去判断一下是否有偏向锁指向本身的ID，无需再进入Monitor去竞争对象了。

举例说明

偏向锁的操作不用直接插到操作系统，不涉及**用户到内核转换**，**不必要直接升级为最高级**，我们以一个account对象的“对象头”为例，



假如有一个线程执行到synchronized代码块的时候，JVM使用CAS操作把线程指ID记录到Mark Word当中，并修改偏向标志，标志当前线程就获得该锁。锁对象变成偏向锁(通过CAS修改对象头里的锁标志位)，字面意思是“偏向于第一个获得它的线程”的锁。执行完同步代码块后，线程并不会主动释放偏向锁。



这时线程获得了锁，可以执行同步代码块。当该线程第二次到达同步代码块时会判断此时持有锁的线程是否还是自己(持有锁的线程ID也在对象头里)，JVM通过account对象的Mark Word判断：当前线程ID还在，说明还持有这个对象的锁，就可以继续进入临界区工作。**由于之前没有释放锁这里也就不需要重新加锁**。**如果自始至终使用锁的线程只有一个**，很明显偏向锁几乎没有额外开销，性能极高。结论：JVM不

用和操作系统协商设置Mutex(争取内核)，它只需要记录下线程ID就标示自己获得了当前锁，不用操作系统接入。

上述就是偏向锁:在没有其他线程竞争的时候，一直偏向偏心当前线程，当前线程可以一直执行。

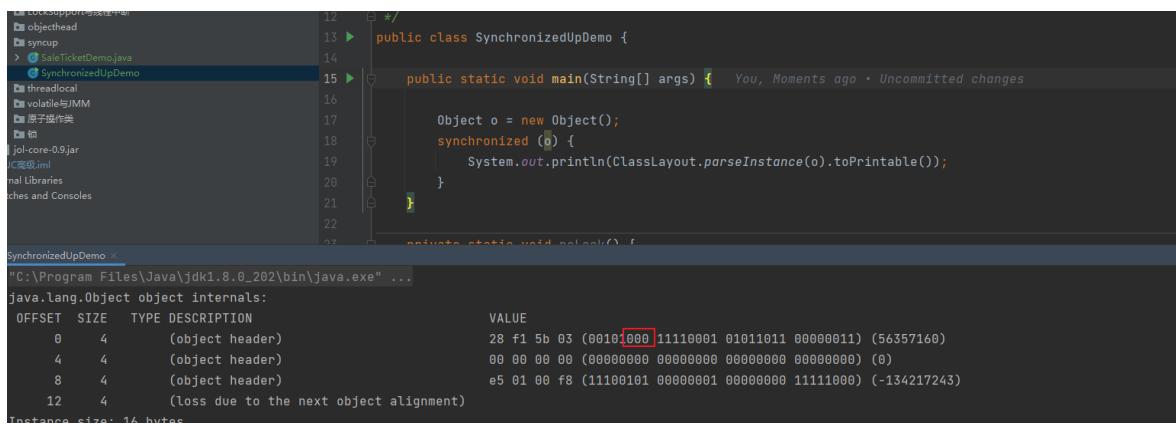
偏向锁JVM命令

```
java -XX:+PrintFlagsInitial | grep BiasedLock*
```

```
C:\Users\13228>java -XX:+PrintFlagsInitial | grep BiasedLock*
    intx BiasedLockingBulkRebiasThreshold      = 20
    intx BiasedLockingBulkRevokeThreshold       = 40
    intx BiasedLockingDecayTime                = 25000
    intx BiasedLockingStartupDelay             = 4000
    bool TraceBiasedLocking                  = false
    bool UseBiasedLocking                    = true
{product}
{product}
{product}
{product}
{product}
{product}
{product}
```

C:\Users\13228>

参数启动偏向锁



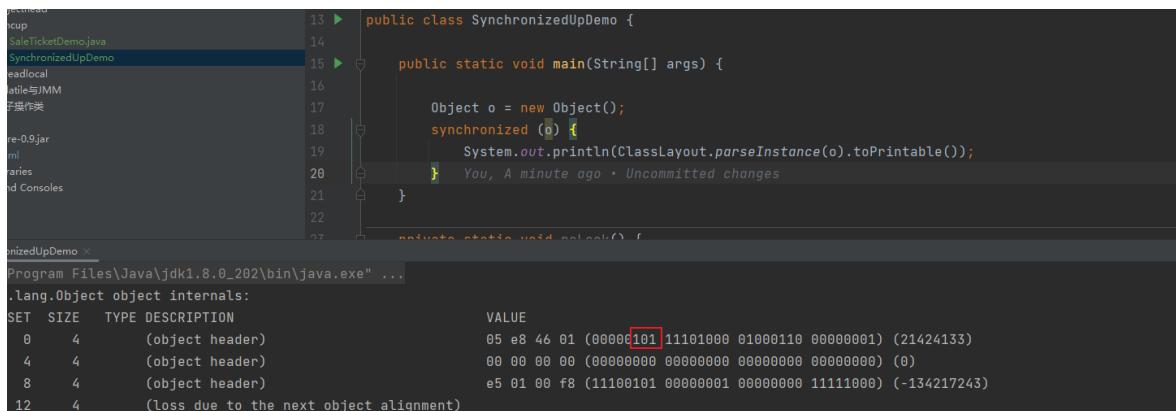
```
/*
 * SynchronizedUpDemo
 */
public class SynchronizedUpDemo {
    public static void main(String[] args) {
        Object o = new Object();
        synchronized (o) {
            System.out.println(ClassLayout.parseInstance(o).toPrintable());
        }
    }
}
```

Object internals:

OFFSET	SIZE	TYPE	DESCRIPTION	VALUE
0	4	(object header)		28 f1 5b 03 (0010:000 11110001 01011011 00000011) (56357160)
4	4	(object header)		00 00 00 00 (00000000 00000000 00000000 00000000) (0)
8	4	(object header)		e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
12	4		(loss due to the next object alignment)	

000 代表的是轻量级锁

因为偏向锁是需要4秒的启动时间的，所以设置虚拟机参数 -XX:BiasedLockingStartupDelay=0



```
/*
 * SynchronizedUpDemo
 */
public class SynchronizedUpDemo {
    public static void main(String[] args) {
        Object o = new Object();
        synchronized (o) {
            System.out.println(ClassLayout.parseInstance(o).toPrintable());
        }
    }
}
```

Object internals:

OFFSET	SIZE	TYPE	DESCRIPTION	VALUE
0	4	(object header)		05 e8 46 01 (0000:101 11101000 01000110 00000001) (21424133)
4	4	(object header)		00 00 00 00 (00000000 00000000 00000000 00000000) (0)
8	4	(object header)		e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
12	4		(loss due to the next object alignment)	

暂停启动偏向锁

第一种情况

```

15  public static void main(String[] args) {
16
17     try { TimeUnit.SECONDS.sleep( timeout: 5); }catch (InterruptedException e){ e.printStackTrace(); }
18
19     Object o = new Object();
20     System.out.println(ClassLayout.parseInstance(o).toPrintable());
21     System.out.println("=====");
22
23 } You, Today * Uncommitted changes
24
25 private static void noLock() {
26     Object o = new Object();

```

02\bin\java.exe" ...
::
VALUE
len) 05 00 00 00 (00000101 00000000 00000000 00000000) (5)
len) 00 00 00 00 (00000000 00000000 00000000 00000000) (0)
len) e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)

先暂停线程5秒，之后获得的也是偏向锁

但此时，对象头中没有保持偏向锁的线程ID

第二种情况

偏向锁带着线程ID的情况

```

15  public static void main(String[] args) {
16
17     try { TimeUnit.SECONDS.sleep( timeout: 5); }catch (InterruptedException e){ e.printStackTrace(); }
18
19     Object o = new Object();
20     System.out.println(ClassLayout.parseInstance(o).toPrintable());
21     System.out.println("=====");
22     new Thread(() -> {
23         synchronized (o) {
24             System.out.println(ClassLayout.parseInstance(o).toPrintable());
25         }
26     }, name: "t1").start(); You, Moments ago * Uncommitted changes
27
28

```

external = 4 bytes total

VALUE
05 e0 bf 21 (00000101 11100000 10111111 00100001) (566222853)
00 00 00 00 (00000000 00000000 00000000 00000000) (0)
e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)

偏向锁的撤销

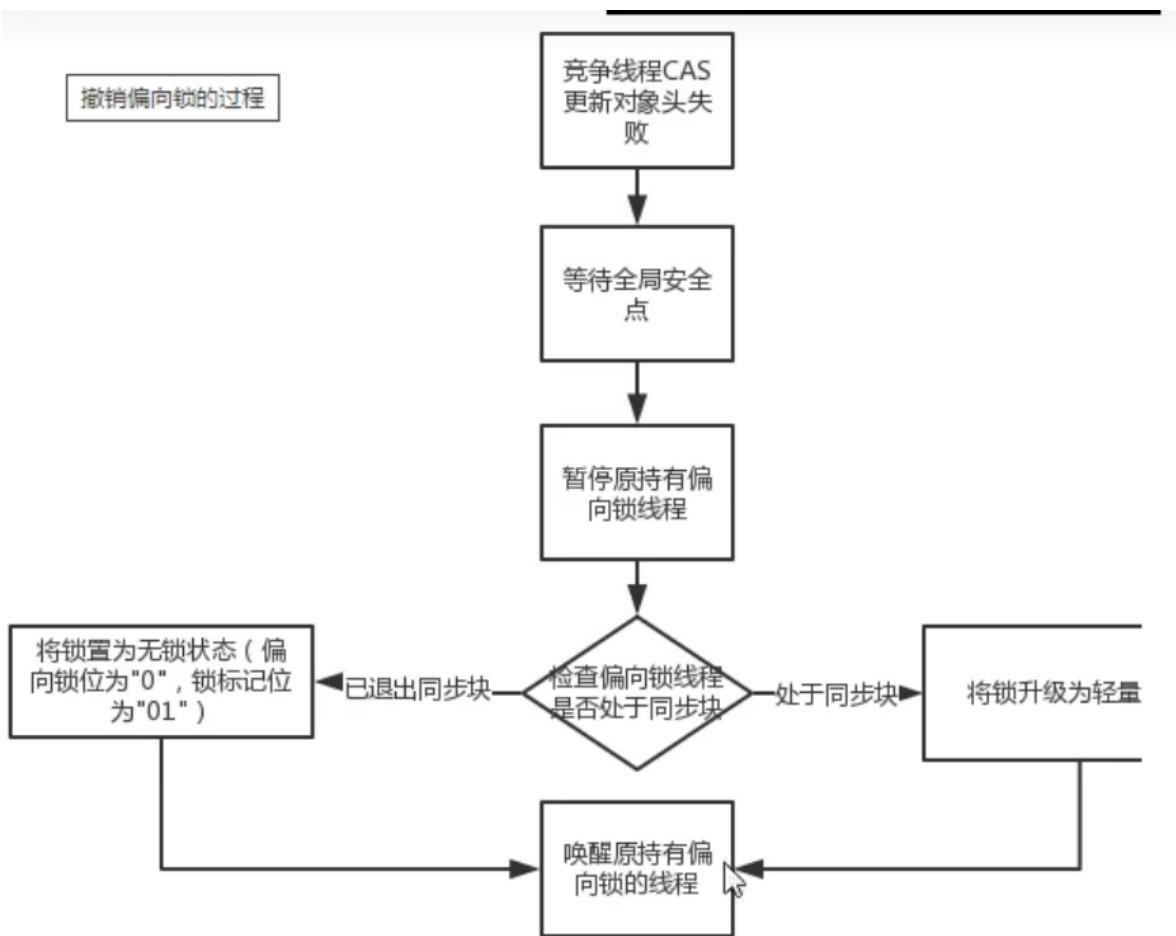
当有另外线程逐步来竞争锁的时候，就不能再使用偏向锁了，要升级为轻量级锁

竞争线程尝试CAS更新对象头失败，会等待到全局安全点(此时不会执行任何代码)撤销偏向锁。

偏向锁使用一种等到竞争出现才释放锁的机制，只有当其他线程竞争锁时，持有偏向锁的原来线程才会被撤销。

撤销需要等待全局安全(该时间点上没有字节码正在执行)，同时检查持有偏向锁的线程是否还在执行：

1. 第一个线程正在执行synchronized方法(处于同步块)，它还没有执行完，其它线程来抢夺，该偏向锁会被取消掉并出现锁升级。此时轻量级锁由原持有偏向锁的线程持有，继续执行其同步代码，而正在竞争的线程会进入自旋等待获得该轻量级锁。
2. 第一个线程执行完成synchronized方法(退出同步块)，则将对象头设置成无锁状态并撤销偏向锁，重新偏向。



12.3 轻量级锁

轻量级锁：多线程竞争，但是任意时刻最多只有一个线程竞争，即不存在锁竞争太过激烈的情况，也就没有线程阻塞

主要作用：

- 有线程来参与锁的竞争，但是获取锁的冲突时间极短
- 本质就是自旋锁CAS

轻量级锁的获取

轻量级锁是为了在线程近乎交替执行同步块时提高性能。

主要目的：在没有多线程竞争的前提下，通过CAS减少重量级锁使用操作系统互斥量产生的性能消耗，说白了先自旋，不行才升级阻塞。

升级时机：当关闭偏向锁功能或多线程竞争偏向锁会导致偏向锁升级为轻量级锁

假如线程A已经拿到锁，这时线程B又来抢该对象的锁，由于该对象的锁已经被线程A拿到，当前该锁已是偏向锁了。

而线程B在争抢时发现对象头Mark Wrd中的线程ID不是线程B自己的线程ID(而是线程A)，那线程B就会进行CAS操作希望能获得锁。

此时线程B操作中有两种情况：

- **如果锁获取成功**，直接替换Mark Wrd中的线程ID为B自己的ID(A → B)。重新偏向于其他线程(即将偏向锁交给其他线程，相当于当前线程“被”释放了锁)，该锁会保持偏向锁状态，A线程Over，B线程上位；

锁状态	54位	2位	1位	4bit	1bit 偏向锁位	2bit 锁标志位
偏向锁	当前线程指针 JavaThread*	Epoch	unused	分代年龄	1	0 1

- 如果锁获取失败，则偏向锁升级为轻量级锁(设置偏向锁标识为0并设置锁标志位为00)，此时轻量级锁由原持有偏向锁的线程持有，继续执行其同步代码，而正在竞争的线程B会进入自旋等待获得该轻量级锁。

锁状态	62位	2bit 锁标志位
轻量级锁 自旋锁 无锁	指向线程栈中Lock Record的指针	0 0

轻量锁级的加锁

JVM会为每令线程在当前线程的栈顺中创建用于存储锁记录的空间，官方成为Displaced Mak Word。若一个线程获得锁时发现是轻量级锁，会把锁的MarkWord复制到自己的Displaced Mak Word里面。然后线程尝试用CAS将锁的MarkWord替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，表示Mark Wrd已经被替换了其他线程的锁记录，说明在与其它线程竞争锁，当前线程就尝试使用自旋来获取锁。

自旋CAS:不断尝试去获取锁, 能不升级就不往上捅, 尽量不要阻塞

轻量级锁的释放

在释放锁时，当前线程会使用 CAS 操作将 Displaced Mark Word 的内容复制回锁的 Mark Word 里面。如果没有发生竞争，那么这个复制的操作会成功。如果有其他线程因为自旋多次导致轻量级锁升级成了重量级锁，那么 CAS 操作会失败，此时会释放锁并唤醒被阻塞的线程。

代码演示

关闭偏向锁，就可以直接进入轻量级锁：`-XX:-UseBiasedLocking`

The screenshot shows the JD-GUI interface with the following details:

- Left Panel (Project Tree):** Shows files like LockSupport与线程中断, objecthead, syncup, SaleTicketDemo.java, SynchronizedUpDemo, threadlocal, volatile与JMM, 原子操作器, core-0.9.jar, 二级xml, Libraries, and e and Consoles.
- Code Editor (Main View):** Displays the `SaleTicketDemo.java` file with the following code:

```
public static void main(String[] args) {
    Object o = new Object();
    new Thread(() -> {
        synchronized (o) {
            System.out.println(ClassLayout.parseInstance(o).toPrintable());
        }
    }, "t1").start();
}
```
- Code Editor (Bottom):** Shows the `biasedLock()` method definition.
- Bottom Panel (Registers):** Shows the following register values:

FFSET	SIZE	TYPE	DESCRIPTION	VALUE
0	4	(object header)		50 f2 f3 21 (01010000 11110010 11110011 00100001) (569635408)
4	4	(object header)		00 00 00 00 (00000000 00000000 00000000 00000000) (0)
8	4	(object header)		e5 01 00 f8 (1100101 00000001 00000000 1111000) (-134217243)

自适应自旋锁的原理

自适应自旋锁的大致原理

线程如果自旋成功了，那下次自旋的最大次数会增加，因为JVM认为既然上次成功了，那么这一次也很大概率会成功。

反之

如果很少会自旋成功，那么下次会减少自旋的次数甚至不自旋，避免CPU空转。

轻量级锁与偏向锁的区别和不同

- 争夺轻量级锁失败时，自旋尝试抢占锁
 - 轻量级锁每次退出同步块都需要释放锁，而偏向锁是在竞争发生时才释放锁

12.4 重量级锁

有大量线程参与锁的竞争，冲突性很高

Java中synchronized的重量级锁，是基于进入和退出Monitor对象实现的。在编译时会将同步块的开始位置插入monitor enter指令，在结束位置插入monitor exit指令。

当线程执行到monitor enter指令时，会尝试获取对象所对应的Monitor所有权，如果获取到了，即获取到了锁，会在Monitor的owner中存放当前线程的id，这样它将处于锁定状态，除非退出同步块，否则其他线程无法获取到这个Monitor。

12.5 总结

锁升级发生后， hashCode去哪了

锁升级为轻量级或重量级锁后，Mark Word中保存的分别是线程栈帧里的锁记录指针和重量级锁指针，已经没有位置再保存哈希码和GC年龄了，那么这些信息被移动到哪里去了呢？

用书中的一段话来描述 锁和hashCode 之前的关系

在Java语言里面一个对象如果计算过哈希码，就应该一直保持该值不变（强烈推荐但不强制，因为用户可以重载hashCode()方法按自己的意愿返回哈希码），否则很多依赖对象哈希码的API都可能存在出错风险。而作为绝大多数对象哈希码来源的Object::hashCode()方法，返回的是对象的一致性哈希码（Identity Hash Code），这个值是能强制保证不变的，它通过在对象头中存储计算结果来保证第一次计算之后，再次调用该方法取到的哈希码值永远不会再发生改变。因此，当一个对象已经计算过一致性哈希码后，它就再也无法进入偏向锁状态了；而当一个对象当前正处于偏向锁状态，又收到需要计算其一致性哈希码请求^[1]时，它的偏向状态会被立即撤销，并且锁会膨胀为重量级锁。在重量级锁的实现中，对象头指向了重量级锁的位置，代表重量级锁的ObjectMonitor类里有字段可以记录非加锁状态（标志位为“01”）下的Mark Word，其中自然可以存储原来的哈希码。

在无锁状态下，Mark Word中可以存储对象的identity hash code值。当对象的hashCode()方法第一次被调用时，JM会生成对应的identity hash code值并将该值存储到Mark Word中。

对于偏向锁，在线程获取偏向锁时，会用Thread ID和epoch值覆盖identity hash code所在的位置。如果一个对象的hashCode()方法已经被调用过一次之后，这个对象不能被设置偏向锁。因为如果可以的话，那Mark Word中的identity hash code必然会被偏向线程Id给覆盖，这就会造成同一个对象前后两次调用hashCode()方法得到的结果不一致。

升级为轻量级锁时，JVM会在当前线程的栈帧中创建一个锁记录(Lock Record)空间，用于存储锁对象的Mark Word拷贝，该拷贝中可以包含identity hash code，所以轻量级锁可以和identity hash code共存，哈希码和GC年龄自然保存在此，释放锁后会将这些信息写回到对象头。

升级为重量级锁后，Mark Word保存的重量级锁指针，代表重量级锁的ObjectMonitor类里有字段记录非加锁状态下的Mark Word，锁释放后也会将信息写回到对象头。

Code1

```

14
15 public static void main(String[] args) {
16
17     try { TimeUnit.SECONDS.sleep( timeout: 5); } catch (InterruptedException e){ e.printStackTrace(); }
18     Object o = new Object();
19     System.out.println("本应是偏向锁");
20     System.out.println(ClassLayout.parseInstance(o).toPrintable());
21
22     // 次有重写，一致性哈希，重写后无效，当一个对象已经计算过identity hash code，它就无法进入偏向锁状态
23     o.hashCode();
24
25     synchronized (o) {
26         System.out.println("本应是偏向锁，但是由于计算过一致性哈希，会直接升级为轻量级锁"); You, Moments ago + Uncommitted
27         System.out.println(ClassLayout.parseInstance(o).toPrintable());
28     }
29
30

```

堆内存分析结果：

```

av.lang.Object object internals:
OFFSET  SIZE  TYPE DESCRIPTION          VALUE
    0   4   (object header)      28 f5 80 02 (00101000 11110101 10000000 00000010) (42005800)
    4   4   (object header)      00 00 00 00 (00000000 00000000 00000000 00000000) (0)
    8   4   (object header)      e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
12   4   (loss due to the next object alignment)

instance size: 16 bytes

```

Code2

```

> objecthead
> syncup
> SynchronizedUpDemo
> threadlocal
> volatile与MM
> 原子操作类
> 锁
> jdk-core-0.8.jar
> JUC高级.ihl
External Libraries
Scratches and Consoles

```

```

29
30     try { TimeUnit.SECONDS.sleep( timeout: 5); } catch (InterruptedException e){ e.printStackTrace(); }
31     Object o = new Object();
32
33     synchronized (o) {
34         o.hashCode();
35         System.out.println("偏向锁过程中遇到一致性哈希计算请求，立马撤销偏向模式，膨胀为重量级锁");
36         System.out.println(ClassLayout.parseInstance(o).toPrintable());
37     } You, Moments ago + Uncommitted changes

```

堆内存分析结果：

```

java.lang.Object object internals:
OFFSET  SIZE  TYPE DESCRIPTION          VALUE
    0   4   (object header)      8a 25 7b 1c (10001010 00100101 01111011 00011100) (477832586)
    4   4   (object header)      00 00 00 00 (00000000 00000000 00000000 00000000) (0)
    8   4   (object header)      e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243)
12   4   (loss due to the next object alignment)

```

各种锁优缺点、synchronized锁升级和实现原理

锁	优点	缺点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到锁竞争的线程，使用自旋会消耗CPU	追求响应时间 同步块执行速度非常快
重量级锁	线程竞争不使用自旋，不会消耗CPU	线程阻塞，响应时间缓慢	追求吞吐量 同步块执行速度较长

synchronized锁升级过程总结:一句话，就是先自旋，不行再阻塞。

实际上是把之前的悲观锁(重量级锁)变成在一定条件下使用偏向锁以及使用轻量级(自旋锁CAS)的形式
synchronized在修饰方法和代码块在字节码上实现方式有很大差异，但是内部实现还是基于对象头的MarkWord来实现的。

JDK1.6之前synchronized使用的是重量级锁，JDK1.6之后进行了优化，拥有了无锁->偏向锁->轻量级锁->重量级锁的升级过程，而不是无论什么情况都使用重量级锁。

12.6 JIT编译器对锁的优化

JIT: Just In Time Compiler, 编译器

锁消除

```
/*
 * @Author cxl
 * @Date 1/6/2023 21:16
 * @ClassReference: com.cxl.juc.syncup.LockClearUPDemo
 * @Description: 锁消除
 * 从JIT角度看相当于无视它, synchronized(o)不存在了
 * 这个锁对象并没有被共用扩散到其它线程使用,
 * 极端的说就是根本没有加这个锁对象的底层机器码, 消除了锁的使用
 *
 * 输出结果:
 * -----hello LockClearUPDemo 127935703 157801043
 * -----hello LockClearUPDemo 1333410369 157801043
 * -----hello LockClearUPDemo 39096934 157801043
 * -----hello LockClearUPDemo 1835179342 157801043
 * -----hello LockClearUPDemo 593800070 157801043
 * -----hello LockClearUPDemo 1857211517 157801043
 * -----hello LockClearUPDemo 847910085 157801043
 * -----hello LockClearUPDemo 1878133438 157801043
 * -----hello LockClearUPDemo 543589342 157801043
 * -----hello LockClearUPDemo 1923132015 157801043
 */
public class LockClearUPDemo {

    static Object objectLock = new Object();

    public void m1() {
        /*synchronized (objectLock) {
            System.out.println("-----hello LockClearUPDemo");
        }*/
    }

    // 锁消除问题, JIT编译器会无视它, synchronized(o), 每次new出来的, 不存在了, 非正常的
    Object o = new Object();
    synchronized (o) {
        System.out.println("-----hello LockClearUPDemo\t" + o.hashCode() +
"\t" + objectLock.hashCode());
    }
}

public static void main(String[] args) {
    LockClearUPDemo lockClearUPDemo = new LockClearUPDemo();

    for (int i = 0; i < 10; i++) {
        new Thread(() -> {
            lockClearUPDemo.m1();
        }, ""+i).start();
    }
}
```

锁粗化

```
/*
 * @Author cxl
 * @Date 1/6/2023 21:21
 * @ClassReference: com.cxl.juc.syncup.LockBigDemo
 * @Description: 锁粗化
 */
public class LockBigDemo {

    static Object objectLock = new Object();

    public static void main(String[] args) {
        new Thread(() -> {
            synchronized (objectLock) {
                System.out.println("1111");
            }

            synchronized (objectLock) {
                System.out.println("2222");
            }

            synchronized (objectLock) {
                System.out.println("3333");
            }

            synchronized (objectLock) {
                System.out.println("4444");
            }
        // 上面四个synchronized会被编译器合并这个下面这个
        synchronized (objectLock) {
            System.out.println("1111");
            System.out.println("2222");
            System.out.println("3333");
            System.out.println("4444");
        }
        }, "t1").start();
    }
}
```

13. AbstractQueuedSynchronizer之AQS

13.1 AQS是什么

抽象的队列同步器

是用来实现锁或者其它同步器组件的公共基础部分的抽象实现，是重量级基础框架及整个JUC体系的基石，主要用于解决锁分配给“谁”的问题

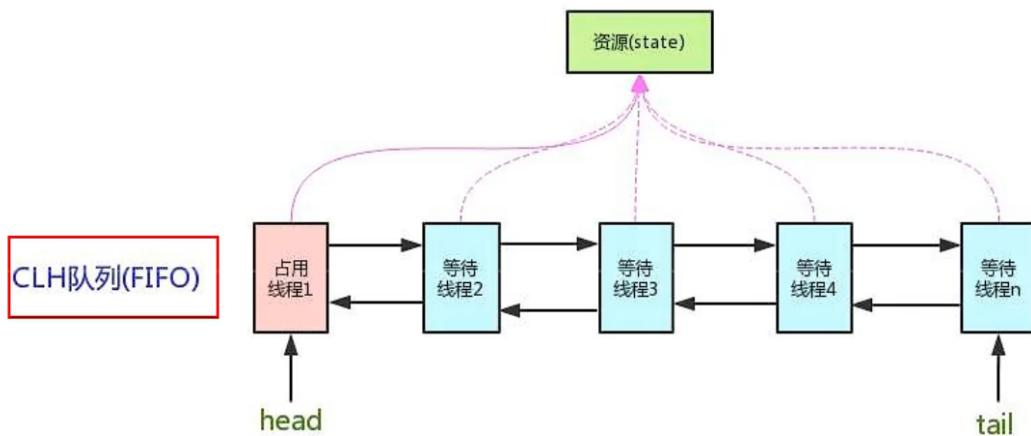
DougLee提出统一规范并简化了锁的实现，将其抽象出来屏蔽了同步状态管理、同步队列的管理和维护、阻塞线程排队和通知、唤醒机制等是一切锁和同步组件实现的-----公共基础部分

整体就是一个抽象的FIFO队列来完成资源获取线程的排队工作，并通过一个int类变量表示持有锁的状态

抢到资源的线程直接使用处理业务，抢不到资源的必然涉及一种排队等候机制。抢占资源失败的线程继续去等待(类似银行业务办理窗口都满了，暂时没有受理窗口的顾客只能去候客区排队等候)，但等候线程仍然保留获取锁的可能且获取锁流程仍在继续(候客区的顾客也在等着叫号，轮到了再去受理窗口办理业务)。

既然说到了排队等候机制，那么就一定会有某种队列形成，这样的队列是什么数据结构呢？

如果共享资源被占用，就需要一定的阻塞等待唤醒机制来保证锁分配。这个机制主要用的是CLH队列的变体实现的，将暂时获取不到锁的线程加入到队列中，这个队列就是AQS同步队列的抽象表现。它将要请求共享资源的线程及自身的等待状态封装成队列的结点对象(**Node**)，通过CAS、自旋以及LockSupport.park()的方式，维护state变量的状态，使并发达到同步的效果。



CLH: Craig、Landin and Hagersten 队列，是一个单向链表，AQS中的队列是CLH变体的虚拟双向队列 FIFO

AQS使用一个volatile的int类型的成员变量来表示同步状态，通过内置的FIFO队列来完成资源获取的排队工作将每条要去抢占资源的线程封装成一个Node节点来实现锁的分配，通过CAS完成对State值的修改。

13.2 AQS内部体系架构

AQS自身

- 同步状态State成员变量 `private volatile int state;`
- CLH队列，双向队列

有阻塞就需要排队，实现排队必然需要队列

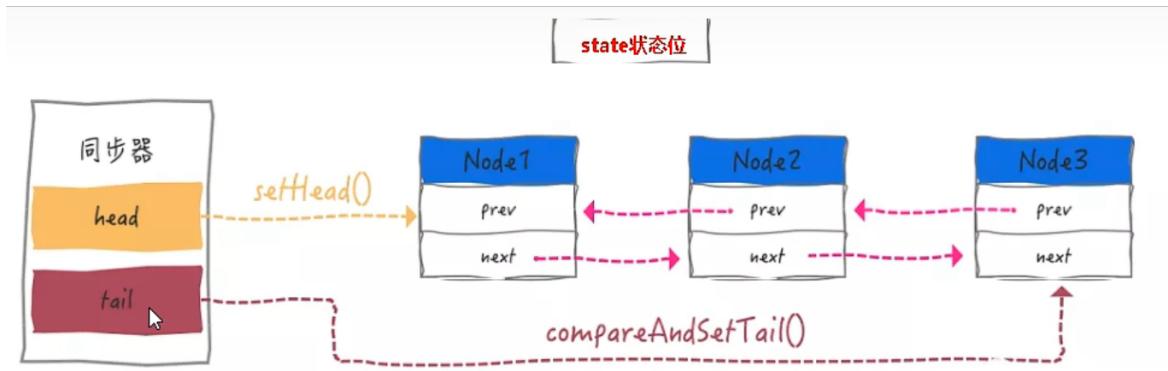
state变量+CLH双端队列

AQS内部类Node

- int变量，Node的等待状态waitState成员变量 `volatile int waitStatus;`

模式	含义	方法和属性值	含义
SHARED	表示线程以共享的模式等待锁	waitStatus	当前节点在队列中的状态
EXCLUSIVE	表示线程正在以独占的方式等待锁	thread	表示处于该节点的线程
Node	SHARED EXCLUSIVE CANCELLED SIGNAL CONDITION PROPAGATE	prev	前驱指针
	waitStatus prev next thread nextWaiter	predecessor	返回前驱节点, 没有的话抛出Inpe
	Node0 Node(Thread, Node) Node(Thread, int)	nextWaiter	指向下一个处于CONDITION状态的节点(
	isShared() predecessor()	next	后继指针

13.3 源码分析



hasQueuedPredecessors() 中判断了是否需要排队，导致公平锁和非公平锁的差异如下：

公平锁: 公平锁讲究先来先到，线程在获取锁时，如果这个锁的等待队列中已经有线程在等待，那么当前线程就会进入等待队列中；

非公平锁: 不管是否有警待队列，如果可以获取锁，则立刻占有锁对象。也就是说队列的第一个排队线程苏醒后，不一定就是排头的这个线程获得锁，它还是需要参加竞争锁(存在线程竞争的情况下)，后来的线程可能不讲武德插队夺锁了。

tryAcquire(arg)

```

class ReentrantLock.NonfairSync {
    final void lock() {
        if (compareAndSetState(0, 1))
            setExclusiveOwnerThread(Thread.currentThread());
        else
            acquire(1);
    }

    final boolean nonfairTryAcquire(int acquires) {
        final Thread current = Thread.currentThread();
        int c = getState();
        if (c == 0) {
            if (compareAndSetState(0, acquires)) {
                setExclusiveOwnerThread(current);
            }
        }
    }
}

```

```

        return true;
    }
}

else if (current == getExclusiveOwnerThread()) {
    int nextc = c + acquires;
    if (nextc < 0) // overflow
        throw new Error("Maximum lock count exceeded");
    setState(nextc);
    return true;
}
return false;
}

class AbstractQueuedSynchronizer {
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
}

```

addWaiter(Node.EXCLUSIVE)

```

private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);
    // Try the fast path of enq; backup to full enq on failure
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    enq(node);
    return node;
}

private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}

```

双向链表不，第一个节点为虚节点(也叫哨兵节点)，其实并不存储任何信息，只是占位真正的第一个有数据的节点，“是从第二个节点开始的。

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
    }final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
} finally {
    if (failed)
        cancelAcquire(node);
}
```

14. ReentrantLock、 ReentrantReadWriteLock、 StampedLock讲解

14.1 读写锁简介

读写锁定义为:一个资源能够被多个读线程访问, 或者被一个写线程访问, 但是不能同时存在读写线程

它只允许读读共存, 而读写和写写依然是互斥的, 大多实际场景是“读/读”线程间并不存在互斥关系.

只有“读/写”线程或“写/写”线程间的操作需要互斥的。因此引入ReentrantReadWriteLock。

一个ReentrantReadWriteLock同时只能存在一个写锁但是可以存在多个读锁, 但不能同时存在写锁和读锁(切菜还是拍蒜选一个)

也即一个资源可以被**多个读操作访问** 或 **一个写操作访问**, 但两者不能同时进行。

只有在读多写少情景之下, 读写锁才具有较高的性能体现。

```
// 资源类, 模拟一个简单的缓存
class MyResource {

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

    Lock lock = new ReentrantLock();

    ReadwriteLock rwLock = new ReentrantReadWriteLock();

    public void write(String key, String value) {
        //      lock.lock();
        rwLock.writeLock().lock();
        try {
            System.out.println(Thread.currentThread().getName() + "\t正在写入");
            map.put(key, value);
            try { TimeUnit.MILLISECONDS.sleep(500); }catch (InterruptedException e){ e.printStackTrace(); }
            System.out.println(Thread.currentThread().getName() + "\t完成写入");
        } finally {
            //      lock.unlock();
            rwLock.writeLock().unlock();
        }
    }

    public void read(String key) {
        //      lock.lock();
        rwLock.readLock().lock();
        try {
            System.out.println(Thread.currentThread().getName() + "\t正在读取");
            String result = map.get(key);
            //try { TimeUnit.MILLISECONDS.sleep(200); }catch (InterruptedException e){ e.printStackTrace(); }
            // 暂停2秒, 演示读锁没有完成之前, 写锁无法获得
            try { TimeUnit.SECONDS.sleep(2); }catch (InterruptedException e){
            e.printStackTrace(); }
            System.out.println(Thread.currentThread().getName() + "\t完成读取" + result);
        } finally {
            //      lock.unlock();
            rwLock.readLock().unlock();
        }
    }
}
```

```

}

public class ReentrantReadWriteLockDemo {

    public static void main(String[] args) {
        MyResource myResource = new MyResource();

        for (int i = 0; i < 10; i++) {
            int finalI = i;
            new Thread(() -> {
                myResource.write(finalI + "", finalI + "");
            }, String.valueOf(i)).start();
        }

        for (int i = 0; i < 10; i++) {
            int finalI = i;
            new Thread(() -> {
                myResource.read(finalI + "");
            }, String.valueOf(i)).start();
        }

        // 暂停1秒钟线程
        try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e){
            e.printStackTrace();
        }

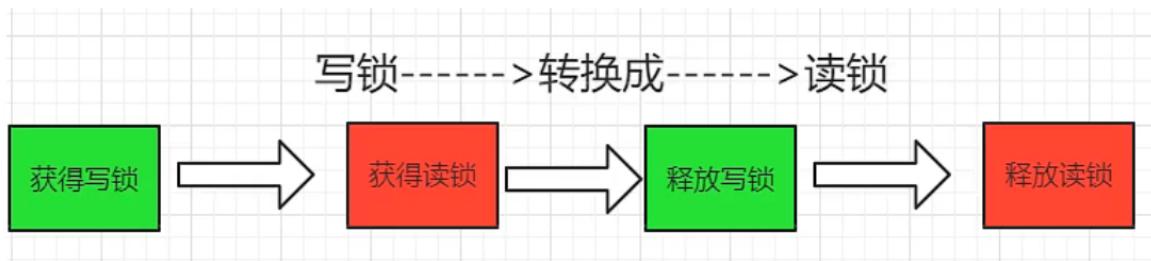
        for (int i = 0; i < 3; i++) {
            int finalI = i;
            new Thread(() -> {
                myResource.write("新写锁=>" + finalI + "", finalI + "");
            }, String.valueOf(i)).start();
        }
    }
}

```

14.2 锁降级

锁降级:遵循获取写锁—再获取读锁—再释放写锁的次序，写锁能够降级成为读锁。

如果一个线程占有了写锁，在不释放写锁的情况下，它还能占有读锁，即写锁降级为读锁。



```

public class LockDownGradingDemo {

    public static void main(String[] args) {

        ReentrantReadWriteLock reentrantReadWriteLock = new
        ReentrantReadWriteLock();

        ReentrantReadWriteLock.WriteLock writeLock =
        reentrantReadWriteLock.writeLock();
    }
}

```

```

        ReentrantReadWriteLock.ReadLock readLock =
reentrantReadWriteLock.readLock();

/*
     // 正常A B两个线程
     // A
writeLock.lock();
System.out.println("---写入");
writeLock.unlock();

     // B
readLock.lock();
System.out.println("---读取"); */

// 本例, only one 同一个线程
writeLock.lock();
System.out.println(" ---写入");

readLock.lock();
System.out.println(" ---读取");

writeLock.unlock();
readLock.unlock();
}
}

```

线程获取读锁是不能直接升级为写入锁的

在ReentrantReadWriteLock中，当读锁被使用时，如果有线程尝试获取写锁，该写线程会被阻塞。所以，需要释放所有读锁，才可获取写锁

写锁和读锁是互斥的(这里的瓦斥是指**线程间的互斥**，当前线程可以获取到写锁又获取到读锁，但是获取到了读锁不能继续获取写锁)，这是因为读写锁要**保持写操作的可见性**。因为，如果允许读锁在被获取的情况下对写锁的获取，那么正在运行的其他读线程无法感知到当前写线程的操作。

如果有线程正在读，写线程需要等待读线程释放锁后才能获取写锁，见前面Case(code演示LockDownGradingDemo)

即ReentrantReadWriteLock**读的过程中不允许写，只有等待线程都释放了读锁，当前线程才能获取写锁也就是写入必须等待，这是一种悲观的读锁，o(T-T)，人家还在读着那，你先别去写，省的数据乱。**

14.3 StampedLock

stampedLock是JDK1.8中新增的一个读写锁也是对JDK1.5中的读写锁ReentrantReadWriteLock的优化。

stamp(戳记，long类型)：代表了锁的状态。当stamp返回零时，表示线程获取锁失败。并且，当释放锁或者转换锁的时候，都要传入最初获取的stamp值

锁饥饿问题

ReentrantReadWriteLock实现了读写分离，但是一旦读操作比较多的时候，想要获取写锁就变得比较困难了，假如当前1000个线程，999个读，1个写，有可能999个读取线程长时间抢到了锁，那1个写线程就悲剧了，**因为当前有可能会一直存在读锁，而无法获得写锁**，根本没机会写

- ReentrantReadWriteLock

允许多个线程同时读。但是只允许一个线程写，在线程获取到写锁的时候，其他写操作和读操作都会处于阻塞状态读锁和写锁也是互斥的，所以在读的时候是不允许写的，读写锁比传统的synchronized速度要快很多，原因就是在于ReentrantReadWriteLock支持读并发，读读可以共享

- StampedLock

ReentrantReadWriteLock的读锁被占用的时候，其他线程尝试获取写锁的时候会被阻塞，但是，StampedLock采取乐观获取锁后，其他线程尝试获取写锁时**不会被阻塞**，这其实是对读锁的优化，在获取乐观读锁后，还需要对结果进行校验。

特点

- 所有**获取锁**的方法，都返回一个邮戳 (Stamp)，Stamp为零表示获取失败，其余都表示成功；
- 所有**释放锁**的方法，都需要一个邮戳 (Stamp)，这个Stamp必须是和成功获取锁时得到的Stamp一致；
- StampedLock**是不可重入的，危险**(如果一个线程已经持有了写锁，再去获取写锁的话就会造成死锁)
- StampedLock有三种访问模式
 - Reading(读模式悲观): 功能和ReentrantReadWriteLock的读锁类似
 - Writing(写模式):功能和ReentrantReadWriteLock的写锁类似
 - Optimistic reading (乐观读模式):无锁机制，类似于数据库中的乐观锁，支持读写并发，**很乐观认为读取时没人修改，假如被修改再实现升级为悲观读模式**

代码案例

```
/***
 * @Author cx1
 * @Date 4/6/2023 20:25
 * @ClassReference: com.cx1.juc.rwlock.StampedLockDemo
 * @Description: StampedLock = ReentrantReadWriteLock + 读的过程中也允许获取写锁介入
 */
public class StampedLockDemo {

    static int number = 37;

    static StampedLock stampedLock = new StampedLock();

    public void write() {
        long stamp = stampedLock.writeLock();
        System.out.println(Thread.currentThread().getName() + "\t写线程准备修改");
        try {
            number = number + 13;
        } finally {
            stampedLock.unlock(stamp);
        }
        System.out.println(Thread.currentThread().getName() + "\t写线程结束修改");
    }

    // 悲观读，读没有完成时候写锁无法获得锁
    public void read() {
        long stamp = stampedLock.readLock();
        System.out.println(Thread.currentThread().getName() + "\t come in
readlock code block, 4seconds continue...");
```

```

        for (int i = 0; i < 4; i++) {
            // 暂停1秒钟线程
            try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e){
e.printStackTrace(); }
            System.out.println(Thread.currentThread().getName() + "\t正在读取
中...");
        }
        try {
            int result = number;
            System.out.println(Thread.currentThread().getName() + "\t获得成员变量
值:" + result);
            System.out.println("写线程没有修改成功，读锁时候写锁无法接入，传统的读写互
斥");
        } finally {
            stampedLock.unlock(stamp);
        }
    }

    public void tryOptimisticRead() {
        long stamp = stampedLock.tryOptimisticRead();
        int result = number;
        // 故意间隔4秒钟，很乐观认为读取中没有其它吸纳成修改过number值，具体靠判断
        System.out.println("4秒前stampedLock.validate方法值(true无修改, false有修
改)" + "\t" + stampedLock.validate(stamp));

        for (int i = 0; i < 4; i++) {
            try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e){
e.printStackTrace(); }
            System.out.println(Thread.currentThread().getName() + "\t正在读取..." +
+ i +
                "秒后stampedLock.validate方法值(true无修改, false有修改)" + "\t"
+ stampedLock.validate(stamp));
        }

        if (!stampedLock.validate(stamp)) {
            System.out.println("有人修改-----有写操作");
            stamp = stampedLock.readLock();
            try {
                System.out.println("从乐观读 升级为 悲观读");
                result = number;
                System.out.println("重新悲观读后result: " + result);
            } finally {
                stampedLock.unlockRead(stamp);
            }
        }

        System.out.println(Thread.currentThread().getName() + "\t finally
value:" + result);
    }

    public static void main(String[] args) {
        StampedLockDemo resource = new StampedLockDemo();

        // 传统版
        /*new Thread() -> {
            resource.read();
        }, "readThread").start();
    }
}

```

```

    // 暂停1秒钟线程
    try { TimeUnit.SECONDS.sleep(1); }catch (InterruptedException e){
e.printStackTrace(); }

    new Thread(() -> {
        System.out.println(Thread.currentThread().getName() + "\t---come
in");
        resource.write();
    }, "writeThread").start();

    try { TimeUnit.SECONDS.sleep(4); }catch (InterruptedException e){
e.printStackTrace(); }

    System.out.println(Thread.currentThread().getName() + "\t number:" + number);*/

    new Thread(() -> {
        resource.tryOptimisticRead();
    }, "readThread").start();

    try { TimeUnit.SECONDS.sleep(6); }catch (InterruptedException e){
e.printStackTrace(); }

    new Thread(() -> {
        System.out.println(Thread.currentThread().getName() + "\t----come
in");
        resource.write();
    }, "writeThread").start();
}
}

```

StampedLock的缺点

- stampedLock 不支持重入，没有Re开头
- StampedLock 的悲观读锁和写锁都不支持条件变量 (Condition)，这个也需要注意。
- 使用stampedLock一定不要调用中断操作，即不要调用interrupt()方法

15. 课程总结与回顾

-
- CompletableFuture
 - “锁”事
 - JMM
 - synchronized及升级优化
 - CAS
 - volatile
 - LockSupport和线程中断
 - AbstactQueuedSynchronizer
 - ThreadLocal
 - 原子增强类Atomic