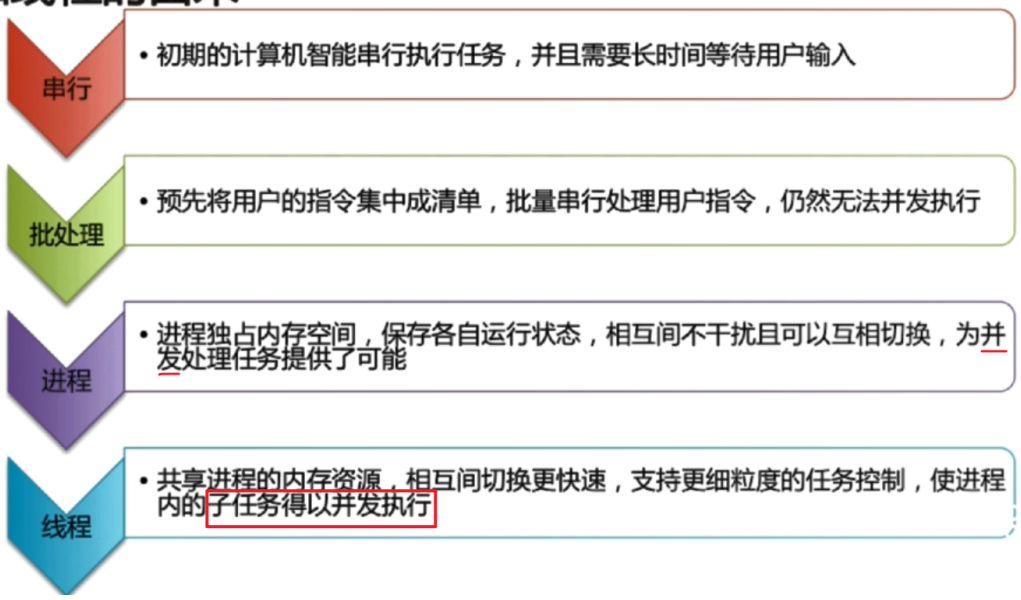
JAVA并发基础

> 更多关于进程线程在操作系统部分。

进程和线程的由来



线程与进程的关系、区别、优缺点

什么是进程？什么是线程？

进程是资源分配的最小单位，线程是CPU调度的最小单位。

进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。

在 windows 中通过查看任务管理器的方式，我们就可以清楚看到 window 当前运行的进程（.exe 文件的运行）。

线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。

进程与线程的区别

拥有资源、是否独立、切换开销、通信。再结合Java说一下

进程是资源分配的最小单位，线程是CPU调度的最小单位。

- **拥有资源：进程是资源分配的最小单位；线程属于某个进程，共享其资源。**
  - 进程拥有完整的虚拟内存地址空间，不同进程有不同虚拟地址空间，同一进程的不同线程共享同一地址空间。
- **是否独立：进程可以看作独立应用（独立的调度、管理、资源分配），线程不能看成独立应用，必须依存于某个应用程序。线程是进程划分成的更小的运行单位。线程执行开销小，但不利于资源的管理和保护**
  - **进程有独立的地址空间，相互不影响，线程只是进程的不同执行路径。**

- 一个进程崩溃后，在保护模式下不会对其他进程产生影响；而一个线程崩溃，所在的进程都会崩溃。因此多进程的程序要比多线程的程序健壮。

- **切换开销：进程切换比线程切换开销大。**
  - 由于**创建或撤销**进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等，所付出的开销远大于创建或撤销线程时的开销。类似地，在进行进程切换时，涉及当前执行进程 CPU 环境的保存及新调度进程 CPU 环境的设置，而线程切换时只需保存和设置少量寄存器内容，开销很小。
  - 在同一进程中，线程的切换不会引起进程切换，从一个进程中的线程切换到另一个进程中的线程时，会引起进程切换。
- **通信：**线程间可以通过直接读写同一进程中的数据进行通信，但是进程通信需要借助IPC。

组成不同：进程=程序+数据+PCB。所有与进程相关的资源，都被记录在PCB中（描述信息，控制信息，资源信息--记录程序段和数据集，CPU现场）。线程只由堆栈寄存器、程序计数器和TCB组成。

Java进程和线程的关系

- Java作为与平台无关的语言，对操作系统提供的功能进行封装，包括进程和线程。
- 在Java中，当我们启动main函数时其实就是启动了一个JVM的进程。程序会自动创建主线程，主线程可以创建子线程，原则上最后完成执行，需要执行各种关闭动作。
- **每个Java进程对应唯一一个JVM实例**，多个线程共享JVM中的**堆**和**方法区**(JDK1.8之后的元空间)资源。但每个线程有自己的**程序计数器**、**虚拟机栈**和**本地方法栈**，所以系统在**产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多**，也正因为如此，线程也被称为轻量级进程。

```
1 public static void main(String[] args) {
2  System.out.println(Thread.currentThread().getName());
3 }
4
5 >>> main
```

Java线程是KLT（内核线程），线程的上下文切换涉及到用户态和内核态的切换，相比用户频繁创建线程，还是使用线程池管理线程更好。

Java程序天生就是多线程程序，JVM实例在创建时，同时会创建很多其他线程。最简单的想法：一定会有负责GC的垃圾收集器线程。我们可以通过 JMX 来看一下一个普通的 Java 程序有哪些线程，代码如下。

```
public class MultiThread {
 public static void main(String[] args) {
 // 获取 Java 线程管理 MXBean
 ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
 // 不需要获取同步的 monitor 和 synchronizer 信息，仅获取线程和线程堆栈信息
 ThreadInfo[] threadInfos = threadMXBean.dumpAllThreads(false, false);
 // 遍历线程信息，仅打印线程 ID 和线程名称信息
 for (ThreadInfo threadInfo : threadInfos) {
 System.out.println("[" + threadInfo.getThreadId() + "] " + threadInfo.getThreadName());
 }
 }
}
```

上述程序输出如下（输出内容可能不同，不用太纠结下面每个线程的作用，只用知道 main 线程执行 main 方法即可）：

```
[5] Attach Listener //添加事件
[4] Signal Dispatcher // 分发处理给 JVM 信号的线程
[3] Finalizer //调用对象finalize方法的线程
[2] Reference Handler //用于处理引用对象本身（软引用、弱引用、虚引用）的垃圾回收问题
[1] main //main 线程,程序入口
```

从上面的输出内容可以看出：一个 Java 程序的运行是 main 线程和多个其他线程同时运行。

Java线程

JVM中创建线程有2种方式

1. new java.lang.Thread().start()

2. 使用JNI将一个native thread attach到JVM中

针对 new java.lang.Thread().start()这种方式，只有调用start()方法的时候，才会真正的在JVM中去创建线程，主要的生命周期步骤有：

1. 创建对应的JavaThread的instance

2. 创建对应的OSThread的instance

3. 创建实际的底层操作系统的native thread

4. 准备相应的JVM状态，比如ThreadLocal存储空间分配等

5. 底层的native thread开始运行，调用java.lang.Thread生成的Object的run()方法

6. 当java.lang.Thread生成的Object的run()方法执行完毕返回后,或者抛出异常终止后，终止native thread

7. 释放JVM相关的thread的资源，清除对应的JavaThread和OSThread

针对JNI将一个native thread attach到JVM中，主要的步骤有：

1. 通过JNI call AttachCurrentThread申请连接到执行的JVM实例

2. JVM创建相应的JavaThread和OSThread对象

3. 创建相应的java.lang.Thread的对象

4. 一旦java.lang.Thread的Object创建之后，JNI就可以调用Java代码了

5. 当通过JNI call DetachCurrentThread之后，JNI就从JVM实例中断开连接

6. JVM清除相应的JavaThread, OSThread, java.lang.Thread对象

为什么程序计数器、虚拟机栈和本地方法栈是线程私有的呢？为什么堆和方法区是线程共享的呢?

复习JVM内存区域这些部分的作用。

程序计数器私有主要是为了**线程切换后能恢复到正确的执行位置。**

为了保证线程中的局部变量不被别的线程访问到，虚拟机栈和本地方法栈是线程私有的。

堆和方法区是所有线程共享的资源，其中堆是进程中最大的一块内存，主要用于存放新创建的对象 (几乎所有对象都在这里分配内存)，方法区主要用于存放已被加载的**类信息(字段、方法)、常量、静态变量、即时编译器编译后的代码**等数据。

什么是上下文切换?

多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。

概括来说就是：当前任务在执行完 CPU 时间片切换到另一个任务之前会先保存自己的状态（就绪状态），以便下次再切换回这个任务时，可以再加载这个任务的状态。**任务从保存到**

再加载的过程就是一次上下文切换。

上下文切换通常是计算密集型的。也就是说，它需要相当可观的处理器时间，在每秒几十上百次的切换中，每次切换都需要纳秒量级的时间。所以，上下文切换对系统来说意味着消耗大量的 CPU 时间，事实上，可能是操作系统中时间消耗最大的操作。

Linux 相比与其他操作系统（包括其他类 Unix 系统）有很多的优点，其中有一项就是，其上下文切换和模式切换的时间消耗非常少。

为什么要使用多线程呢?

同时进行，又要共享某些变量的操作，需要用多进程或多线程。多线程开销要比多进程小的多。

先从总体上来说：

- 提高程序的执行效率、提高程序运行速度。
- 从计算机底层来说： 线程可以比作是轻量级的进程，是程序执行的最小单位，线程间的**切换和调度的成本**远远小于进程。另外，多核 CPU 时代意味着多个线程可以同时运行，这减少了线程上下文切换的开销。
- 从当代互联网发展趋势来说： 现在的系统动不动就要求百万级甚至千万级的并发量，而多线程并发编程正是开发高并发系统的基础，利用好多线程机制可以大大提高系统整体的**并发能力以及性能。**

再深入到计算机底层来探讨：

- 单核时代： 在单核时代多线程主要是为了**提高 CPU 和 IO 设备的综合利用率。**举个例子：当只有一个线程的时候会导致CPU 计算时，IO 设备空闲；进行 IO 操作时，CPU空闲。我们可以简单地说这两者的利用率目前都是 50%左右。但是当有两个线程的时候就不一样了，当一个线程执行 CPU 计算时，另外一个线程可以进行 IO 操作，这样两个的利用率就可以在理想情况下达到 100%了。
- 多核时代: 多核时代多线程主要是为了**提高CPU利用率。**举个例子：假如我们要计算一个复杂的任务，我们只用一个线程的话，CPU只会一个CPU核心被利用到，而创建多个线程就可以让多个 CPU 核心被利用到，这样就提高了 CPU 的利用率。

线程的通信与同步

线程的通信是指线程之间以何种机制来交换信息。在编程中，线程之间的通信机制有两种，共享内存和消息传递。

在**共享内存**的并发模型里，线程之间共享程序的公共状态，线程之间**通过写-读内存中的公共状态来隐式进行通信**，典型的共享内存通信方式就是通过共享对象进行通信。**JMM内存模型**

在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信，在java中典型的消息传递方式就是wait()和notify()。

同步是指程序用于控制不同线程之间操作发生相对顺序的机制。

使用多线程可能带来什么问题?

并发编程的目的就是为了能提高程序的执行效率、提高程序运行速度，但是并发编程并不总是能提高程序运行速度的，而且并发编程可能会遇到很多问题，比如：**内存泄漏、死锁、线程不安全等等。**

Java使用线程有哪几种方式?

（a.继承 Thread 类;b.实现 Runnable 接口;c.实现 Callable 接口d. 使用 Executor 框架;）

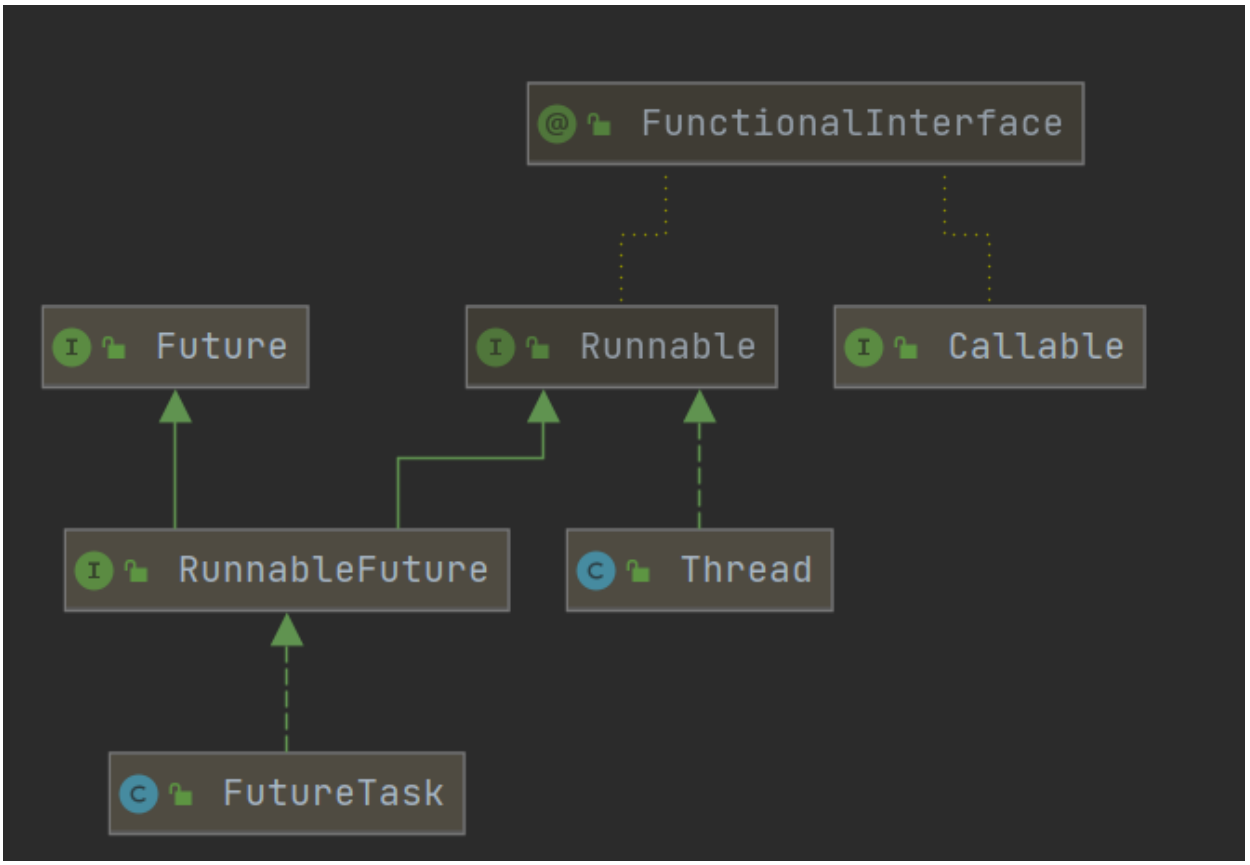
只推荐用Executor。前三种最好是Runnable

创建线程需要什么步骤

有三种使用线程的方法：

- 继承 Thread 类。
- 实现 Runnable 接口；
- 实现 Callable 接口；

因Java类的单一继承原则，为了提升系统的可扩展性，往往使业务类实现Runnable接口，将业务逻辑封装在run方法中，便于后续给普通类附上多线程的特性。



1. 继承 Thread 类

Thread 类实现了Runable接口，需要实现run()方法。

线程的执行代码写在run()方法中；把需要执行的任务放到run()中。不然是空的，线程什么都不会执行。

start()方法会在内部自动调用实例的run()方法。当调用 start() 方法启动一个线程时，虚拟机会将该线程放入就绪队列中等待被调度，当一个线程被调度时会执行该线程的 run() 方法。

```
public class MyThread extends Thread {
 @Override
 public void run() {
 System.out.println("start new thread!");
 }

 public static void main(String[] args) {
 Thread t = new MyThread();
 t.start(); // 启动新线程
  //也可以直接写
  Thread t = new Thread(){
  public void run(){
  ...
  }

```

```
16    }
17 }
```

Thread中常用的一些方法：

```
1 public static native Thread currentThread();
2 public final String getName();
```

一个线程调用两次start()会怎样?

Java的线程是不允许启动两次的，第二次调用必然会抛出**IllegalThreadStateException**，这是一种运行时异常，多次调用start被认为是编程错误。

start()和run()的区别，直接调用run会怎么样?

调用start()方法会创建一个新的子线程并启动，并且会自动调用run()，让被启动的线程执行run()中的任务代码。

run()方法只是Thread的一个普通方法调用。run()就和普通的成员方法一样，可以被重复调用。单独调用run()的话，会在当前线程中执行run()，而并不会启动新线程。

从源码来看：start()调用的native方法里调用了run方法

Thread中的start方法调用JVM包里的JVM_StartThread的方法，创建一个线程，其中thread_entry会call虚拟机，并且传入run method，执行run方法。

如何查看源码：Java并发拓展.note

2. 实现Runnable接口

> In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

实现Runnable接口实际上只包含任务，并没有创建线程。使用的是静态代理模式，Thread类作为代理商，实现Runnable接口，为实现Runnable接口的对象的run方法提供多线程的并发执行的功能。

Thread的构造方法：

Thread(Runnable target)

Thread(Runnable target, String name)

@FunctionalInterface

```
public interface Runnable {
 public abstract void run();
}
```

单方法接口可以用Java8引入的lambda语法进一步简写为：

```
public class Main {
 public static void main(String[] args) {
 Thread t = new Thread(() -> {
 System.out.println("start new thread!");
 });
 t.start(); // 启动新线程
 }
}
```

run()方法是没有参数的，如何给run()方法传参?

1. 通过constructor给**成员变量**赋值，在run()中使用这个成员变量。获取线程的返回值也可以用成员变量法。具体见如何处理线程返回值方法1

2. 或者用set给成员变量赋值。

3. 回调 还没找到合适的例子

通过实现Runnable接口的线程类，是互相共享资源的。

如下例模拟实现了抢票活动：

```
public class MyRunnable implements Runnable {
 private int num = 10;

 @Override
 public void run() {
 while (true) {
 if (num < 0) {
 break;//跳出循环
 }
  System.out.println(Thread.currentThread().getName() + "抢到了" + num--);
  }
  }
```

```
13
14  public static void main(String[] args) {
15  //真实角色
16  MyRunnable qiangpiao = new MyRunnable();
17  //代理
18  Thread t1 = new Thread(qiangpiao, "路人甲");
19  Thread t2 = new Thread(qiangpiao, "黄牛已");
20  Thread t3 = new Thread(qiangpiao, "攻城狮");
21  //启动线程
22  t1.start();
23  t2.start();
24  t3.start();
25  }
26 }
```

输出：

```
1  攻城狮抢到了8
2  攻城狮抢到了7
3  攻城狮抢到了6
4  攻城狮抢到了5
5  攻城狮抢到了4
6  攻城狮抢到了3
7  攻城狮抢到了2
8  攻城狮抢到了1
9  攻城狮抢到了0
10  路人甲抢到了10
11  黄牛已抢到了9
```

Thread和Runnable的关系、对比

Thread是实现了Runnable的类，使得run()支持多线程。

实现Runnable接口实际上只包含任务，并没有创建线程。使用的是静态代理模式，Thread类作为代理商，实现Runnable接口，为实现Runnable接口的对象的run方法提供多线程的并发执行的功能。

实现Runnable接口 VS 继承 Thread

1. 共享资源。可以由多个线程共享同一个资源，**全局变量**。如上述12306购票
2. 接口可以避免java类的单继承的局限性。
3. 在一个地方做开发，在多个地方去使用，代码和线程独立（局部变量）。
4. 线程池只能放入实现Runable或Callable接口的线程，不能直接放入继承Thread的类。

这是一种简单的静态代理模式，详细见设计模式。

静态代理有如下要素：

1.目标角色（真实角色）。MyRunnable

2.代理角色。　Thread

3.目标角色和代理角色实现同一接口。

4.代理角色持有目标角色的引用。new thread(myRunnable)

3. 实现 Callable 接口

与 Runnable 相比，Callable 可以有返回值，返回值通过 FutureTask 进行封装。

//Callable同样是任务，与Runnable接口的区别在于它接收泛型，同时它执行任务后带有返回内容

Runnable和callable比较：

相同点：

- 两者都是接口；
- 两者都可用来编写多线程程序；
- 两者都需要调用 Thread.start()启动线程；

不同点：

- 实现 Callable 接口的线程能返回执行结果；而实现 Runnable 接口的线程不能返回结果；
- Callable 接口的 call()方法允许抛出异常；而 Runnable 接口的 run()方法的不允许抛异常；
- 实现 Callable 接口的线程可以调用 Future.cancel 取消执行，而实现 Runnable 接口的线程不能

注意点：

Callable 接口支持返回执行结果，此时需要调用 FutureTask.get()方法实现，此方法会阻塞主线程直到获取‘将来’结果；当不调用此方法时，主线程不会阻塞！

实现 Runnable 和 Callable 接口的类只能当做一个可以在线程中运行的任务，不是真正意义上的线程，因此最后还需要通过 Thread 来调用。可以理解为任务是通过线程驱动从而执行的。

如何处理线程的返回值

——多个线程结果互相影响的场景：线程B需要用到线程A的运算结果。线程B要怎么确保线程A已经结束运算，返回结果了呢?

1、主线程循环等待法 （优点：实现起来简单，缺点：需要等待的变量一多的话，代码就变的非常臃肿。而且不能精准控制时间）

如下面的例子，主线程要等待run方法结束，把返回值写到成员变量value中。

```
1 public class CycleWait implements Runnable{
2  private String value;
```

```
3  public void run() {
4  try {
5  Thread.currentThread().sleep(5000);
6  } catch (InterruptedException e) {
7  e.printStackTrace();
8  }
9  value = "we have data now";
10  }
11
12  public static void main(String[] args) throws InterruptedException {
13  CycleWait cw = new CycleWait();
14  Thread t = new Thread(cw);
15  t.start();
16  while (cw.value == null){
17  Thread.currentThread().sleep(100);
18  }
19  System.out.println("value : " + cw.value);
20  }
21 }
22
23 >>>value : we have data now
```

2、使用thread的join方法，在主线程中写t.join()，主线程会等待t线程执行结束后才继续执行。这时候已经把set了value

```
1 public static void main(String[] args) throws InterruptedException {
2  CycleWait cw = new CycleWait();
3  Thread t = new Thread(cw);
4  t.start();
5  t.join();
6  System.out.println("value : " + cw.value);
7  }
```

3、通过Callable接口实现：通过FutureTask或线程池获取

```
1 public class MyCallable implements Callable<String> {
2  @Override
3  public String call() throws Exception{
4  String value="test";
5  System.out.println("Ready to work");
6  Thread.currentThread().sleep(5000);
7  System.out.println("task done");
8  return value;
9  }
```

```
10 }
```

FutureTask实现方式：

FutureTask可用于包装Callable或Runnable对象。 因为**FutureTask实现了Runnable接口**，一个FutureTask可以提交到一个Executor执行。

public boolean isDone() 判断线程任务方法是否执行完成

public V get() 阻塞当前调用它的线程，直到callable方法执行完成。

public V get(long timeout, TimeUnit unit) 超过timeout，会抛出TimeoutException

```
1  public class FutureTaskDemo {
2   public static void main(String[] args) throws ExecutionException, Interr
uptedException {
3   FutureTask<String> task = new FutureTask<String>(new MyCallable());
4   new Thread(task).start();
5   if(!task.isDone()){
6   System.out.println("task has not finished, please wait!");
7   }
8   //task.get()会阻塞主线程，直到task执行完成
9   System.out.println("task return: " + task.get());
10   }
11  }
12
13  >>>
14  task has not finished, please wait!
15  Ready to work
16  //五秒之后输出
17  task done
18  task return: test
```

线程池实现方式

submit方法

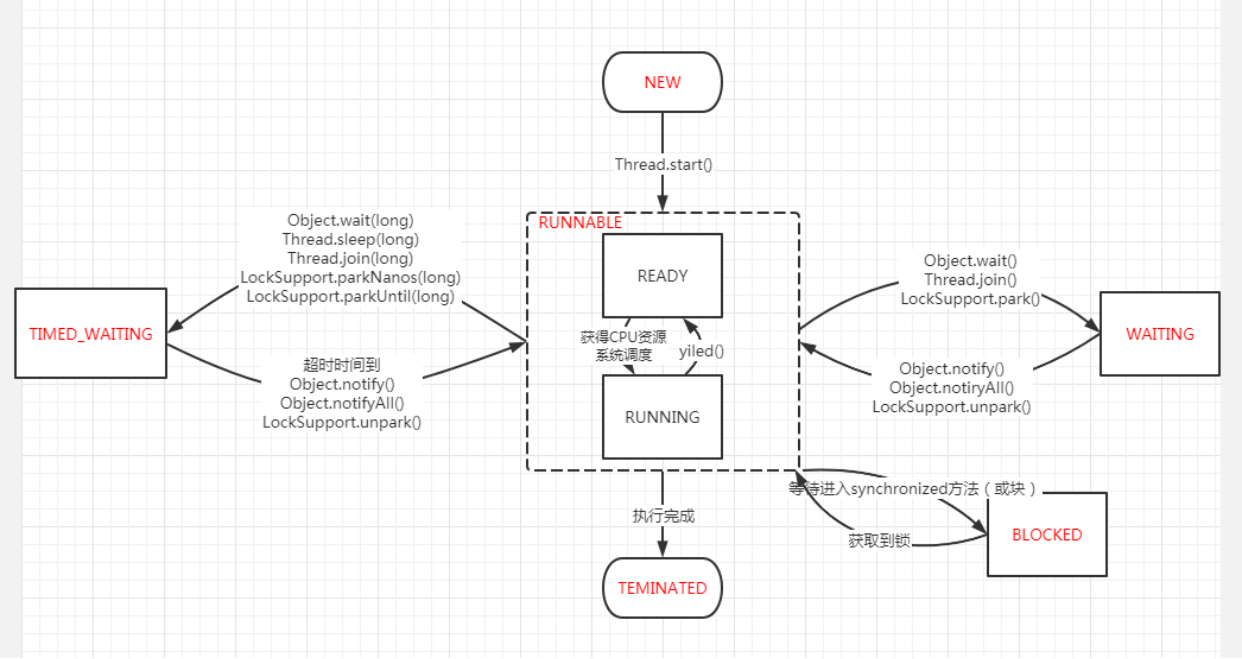
```
1  public class ThreadPoolDemo {
2   public static void main(String[] args) {
3   ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();
4   Future<String> future = newCachedThreadPool.submit(new MyCallable());
5   if(!future.isDone()){
6   System.out.println("task has not finished, please wait!");
7   }
8   try {
```

```
System.out.println(future.get());
 } catch (InterruptedException e) {
 e.printStackTrace();
 } catch (ExecutionException e) {
 e.printStackTrace();
 } finally {
 //不要忘记关闭线程池
 newCachedThreadPool.shutdown();
 }
 }
}

```

说说线程的生命周期和状态

一个线程只能处于一种状态，并且这里的线程状态特指 Java 虚拟机的线程状态，不能反映线程在特定操作系统下的状态。



新建（ NEW）创建后尚未启动。

可运行（RUNABLE）在操作系统层面Running和Ready状态（在线程池中等待被分到时间片而调度）

无限期等待（WAITING）不会被分配CPU执行时间，等待其它线程显式地唤醒。

限期等待（TIMED_WAITING）无需等待其它线程显式地唤醒，在一定时间之后会被系统自动唤醒。

阻塞（BLOCKED）等待获取排他锁

请求获取 monitor lock 从而进入 synchronized 函数或者代码块，但是其它线程已经占用了该 monitor lock，所以出于阻塞状态。要结束该状态进入从而 RUNABLE需要其他线程释放monitor lock。

死亡（TERMINATED）可以是线程结束任务之后自己结束，或者产生了异常而结束。

如果任务结束以后，又调用start()，会java.lang.IllegalThreadStateException。

```
t.start(); t.join(); t.start();
>>> Exception in ... java.lang.IllegalThreadStateException。
```

阻塞和等待的区别在于，阻塞是被动的，它是在等待获取 monitor lock。而等待是主动的，通过调用 Object.wait() 等方法进入。

睡眠和挂起是用来描述行为，而阻塞和等待用来描述状态。

调用 Thread.sleep() 方法使线程进入限期等待状态时，常常用“使一个线程睡眠”进行描述。调用 Object.wait() 方法使线程进入限期等待或者无限期等待时，常常用“挂起一个线程”进行描述。

基础线程机制

wait() 和 sleep() 的区别

本质区别

Thread.sleep只会让出CPU，不会释放锁。Object.wait不仅会让出CPU，还会释放已经占有的同步锁的资源。

基本区别

- wait() 是 Object 的方法，而 sleep() 是 Thread 的静态方法；
- sleep()可以在任何地方使用，wait()方法只能在synchronized方法或者synchronized块中使用。
- wait() 通常被用于线程间交互/通信，sleep() 通常被用于暂停执行。

具体代码链接

Thread.sleep(millisec) 方法会**休眠当前正在执行的线程**，millisec 单位为毫秒。

sleep() 可能会抛出 InterruptedException，因为异常不能跨线程传播回 main() 中，因此必须在本地进行处理。线程中抛出的其它异常也同样需要在本地进行处理。//自己的异常自己处理

```
public void run() {
```

```
2   try {
3   Thread.sleep(3000);
4   } catch (InterruptedException e) {
5   e.printStackTrace();
6   }
7  }
```

notify和notifyAll的区别

notify并不释放锁，只是告诉调用过wait方法的线程可以去参与获得锁的竞争了，但不是马上得到锁，因为锁还在别人手里，别人还没释放。如果notify方法后面的代码还有很多，需要这些代码执行完后才会释放锁 。

notify原理

java虚拟机中运行的每个对象都有两个池，锁池EntryList和等待池WaitSet，这两个池与Object的wait,notify,notifyAll以及synchronized相关。

锁池：假设线程A已经拥有了**某个对象**的锁，而其他线程A和B想要进入这个对象的某个synchronized方法，**B和C线程会被阻塞，进入锁池**中线程A释放锁。entryList

等待池：假设线程A调用了某个对象的wait()方法，线程A就会释放该对象的锁，同时进入该对象的等待池，进入等待池中的线程不会去竞争该对象的锁。waitSet()

当调用notify或notifyAll时，会把等待池中的线程重新加入到锁池中，竞争锁。

notifyAll会让所有处于等待池的线程全部进入锁池去竞争获取锁的机会。

notify只会随机选取一个处于等待池的线程进入锁池去竞争获取锁的机会。

示例代码笔记链接 一般标志位变量都会使用volatile修饰

yeild()

> A hint to the scheduler that the current thread is willing to yield its current use of a processor. The scheduler is free to ignore this hint.

会给线程调度器一个当前线程愿意让出CPU使用的**暗示**，但是线程调度器可能会忽略这个暗示。**对锁没有任何影响。**

它可能对调试或测试有用，可能有助于再现由于竞争条件而产生的bug。在设计并发控制结构，如JUC.lock中有用。

中断

[Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?]

调用thread.interrupt()，通知线程该中断了。

1. 如果线程处于**被阻塞状态**，线程会立即退出被阻塞状态，并抛出一个**InterruptedException异常。**

wait(), wait(long), or wait(long, int) methods of the Object class

or of the join(), join(long), join(long, int), sleep(long), or sleep(long, int), methods of this class

2. 如果线程处于正常活动状态，会将线程的中断标志设置为true，可以用**isInterrupted()**判断。被设置中断标志的线程会继续正常运行，不受影响。 一般和isInterrupted()结合使用，用在while循环的判断语句中。

```
try {
  //在正常运行任务时，经常检查本线程的中断标志位，如果被设置了中断标志就自行停止线程
  while (!Thread.currentThread().isInterrupted()) {
  Thread.sleep(100); // 休眠100ms
  i++;
  System.out.println(Thread.currentThread().getName() + " (" + Thread.currentThread().getState() + ") loop " + i);
  }
 } catch (InterruptedException e) {
  //在调用阻塞方法时正确处理InterruptedException异常。（例如，catch异常后就结束线程。）
   System.out.println(Thread.currentThread().getName() + " (" + Thread.currentThread().getState() + ") catch InterruptedException.");
 }
```

[示例代码笔记链接]

Daemon守护线程

守护线程是程序运行时在后台提供服务的线程，不属于程序中不可或缺的部分。

当所有非守护线程结束时，程序也就终止，同时会杀死所有守护线程。main() 属于非守护线程。

在线程启动之前使用 setDaemon() 方法可以将一个线程设置为守护线程。必须在线程start之前设置

用isDaemon()检查是否是守护线程。

```
public static void main(String[] args) {
  Thread thread = new Thread(new MyRunnable());
  thread.setDaemon(true);
```

```
    thread.start();
    System.out.println(thread.isDaemon());
}
```

应用举例：每隔多长时间去做一些说明事情，主线程结束后自动结束。

线程安全：加锁方案

synchronized和reentrantLock是加锁方案，互斥同步，会阻塞其他没有获得锁的线程。也是一种悲观锁，无论共享数据是否真的会出现竞争，它都要进行加锁、用户态核心态转换、维护锁计数器和检查是否有被阻塞的线程需要唤醒等操作。

synchronized

说说你对synchronized的了解

synchronized关键字解决的是多个线程之间访问资源的同步性，**synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。**

在 Java 早期版本中，synchronized属于重量级锁，效率低下，因为**监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的**，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从**用户态转换到内核态**，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的 synchronized 效率低。庆幸的是在 Java 6 之后 Java 官方对从 **JVM 层面**对synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6对锁的实现引入了大量的优化，如**自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁**等技术来减少锁操作的开销。

synchronized的使用方法

1. **同步代码块(synchronized(this|object))**
2. **同步非静态方法**
3. **同步类(synchronized(类.class)**
4. **同步静态方法**

分为两类：对象锁和类锁。本质上都是对象锁，类锁也是特殊的对象锁，锁class对象。同步类作用在class对象上，同步对象作用在对象上，这是两种不同的锁，两者之间**互相不影响 。要先获取到这个互斥的锁，才能进入被锁修饰的代码块。**

还可以分成同步代码块和同步方法。2*2

注意：构造方法不能使用 synchronized 关键字修饰。

构造方法本身就属于线程安全的，不存在同步的构造方法一说。

1. 同步代码块

```
public void func() {
 synchronized (this|object) {
 // ...
 }
}
```

作用于当前对象实例加锁，进入同步代码前要获得当前对象的锁。

对于以下代码，使用 ExecutorService 执行了两个线程，由于调用的是同一个对象的同步代码块，因此这两个线程会进行同步，当一个线程进入同步语句块时，另一个线程就必须等待。

```
public class SynchronizedExample {
 public void func1() {
 synchronized (this) {
 for (int i = 0; i < 10; i++) {
 System.out.print(i + " ");
 }
 }
 }
}
 public static void main(String[] args) {
  SynchronizedExample e1 = new SynchronizedExample();
  ExecutorService executorService = Executors.newCachedThreadPool();
  executorService.execute(() -> e1.func1());
  executorService.execute(() -> e1.func1());
 }

>>> 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
```

对于以下代码，两个线程调用了不同对象的同步代码块，因此这两个线程就不需要同步。从输出结果可以看出，两个线程交叉执行。

```
public static void main(String[] args) {
 SynchronizedExample e1 = new SynchronizedExample();
 SynchronizedExample e2 = new SynchronizedExample();
 ExecutorService executorService = Executors.newCachedThreadPool();
 executorService.execute(() -> e1.func1());
 executorService.execute(() -> e2.func1());
}
>>> 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
```

2. 同步非静态方法

```
public synchronized void func () {
 // ...
}
```

它和同步代码块一样，作用于同一个对象。

3. 同步类

```
public void func() {
 synchronized (SynchronizedExample.class) {
 // ...
 }
}
```

作用于整个类，也就是说两个线程调用同一个类的不同对象上的这种同步语句，也会进行同步。

```
public class SynchronizedExample {

 public void func2() {
 synchronized (SynchronizedExample.class) {
 for (int i = 0; i < 10; i++) {
 System.out.print(i + " ");
 }
 }
 }
 }
public static void main(String[] args) {
 SynchronizedExample e1 = new SynchronizedExample();
 SynchronizedExample e2 = new SynchronizedExample();
 ExecutorService executorService = Executors.newCachedThreadPool();
 executorService.execute(() -> e1.func2());
 executorService.execute(() -> e2.func2());
}
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
```

4. 同步静态方法

```
public synchronized static void fun() {
 // ...
}
```

作用于整个类。

synchronized实现原理

synchronized关键字底层原理属于JVM层面，在java虚拟机中的同步是基于**进入和退出管程（Monitor）对象实现的。**同步有显式同步(有明确的**monitorenter和moniterexit**)和隐式同步(**ACC_SYNCHRONIZED**)。

1. synchronized同步语句块

```
1 public void syncBlock() {
2  synchronized (this) {
3  System.out.println("synchronized code block");
4  }
5 }
```

javap -verbose src/javabasic/concurrency/SynchronizedDemo，反编译的汇编指令：

```
public void method();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=1
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter
       4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       7: ldc           #3                  // String Method 1 start
       9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      12: aload_1
      13: monitorexit
      14: goto          22
      17: astore_2
      18: aload_1
      19: monitorexit
      20: aload_2
      21: athrow
      22: return
    Exception table:
       from    to  target type
           4    14    17   any
          17    20    17   any
    LineNumberTable:
      line 5: 0
      line 6: 4
      line 7: 12
      line 8: 22
    StackMapTable: number_of_entries = 2
      frame_type = 255 /* full_frame */
        offset_delta = 17
        locals = [ class test/SynchronizedDemo, class java/lang/Object ]
        stack = [ class java/lang/Throwable ]
      frame_type = 250 /* chop */
        offset_delta = 4
}
SourceFile: "SynchronizedDemo.java"
```

从上面我们可以看出：

synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。

当执行 monitorenter 指令时，线程试图获取锁也就是获取对象监视器monitor的持有权。

在 Java 虚拟机(HotSpot)中，Monitor 是基于 C++实现的，由ObjectMonitor实现的。每个对象中都内置了一个 ObjectMonitor对象。

另外，**wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法**，否则会抛出java.lang.IllegalMonitorStateException的异常。

在执行monitorenter时，会尝试获取对象的锁，如果锁的计数器为 0 则表示锁可以被获取，获取后将锁计数器设为 1 也就是加 1。

在执行 monitorexit 指令后，将锁计数器设为 0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

第二个monitor_exit，保证程序出现异常时，monitor也可以被释放。

2. synchronized修饰方法

```
1 public synchronized void syncMethod() {
2  System.out.println("synchronized method");
3 }
```

```
{
  public test.SynchronizedDemo2();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 3: 0

  public synchronized void method();
    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 synchronized 鏂规硶
         5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
         8: return
      LineNumberTable:
        line 5: 0
        line 6: 8
}
SourceFile: "SynchronizedDemo2.java"
```

synchronized修饰的方法使用 ACC_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法。JVM 通过该 ACC_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

对象头：2021-JVM.note

Monitor：也称为管程，监视器锁。

https://www.jianshu.com/p/32e1361817f0

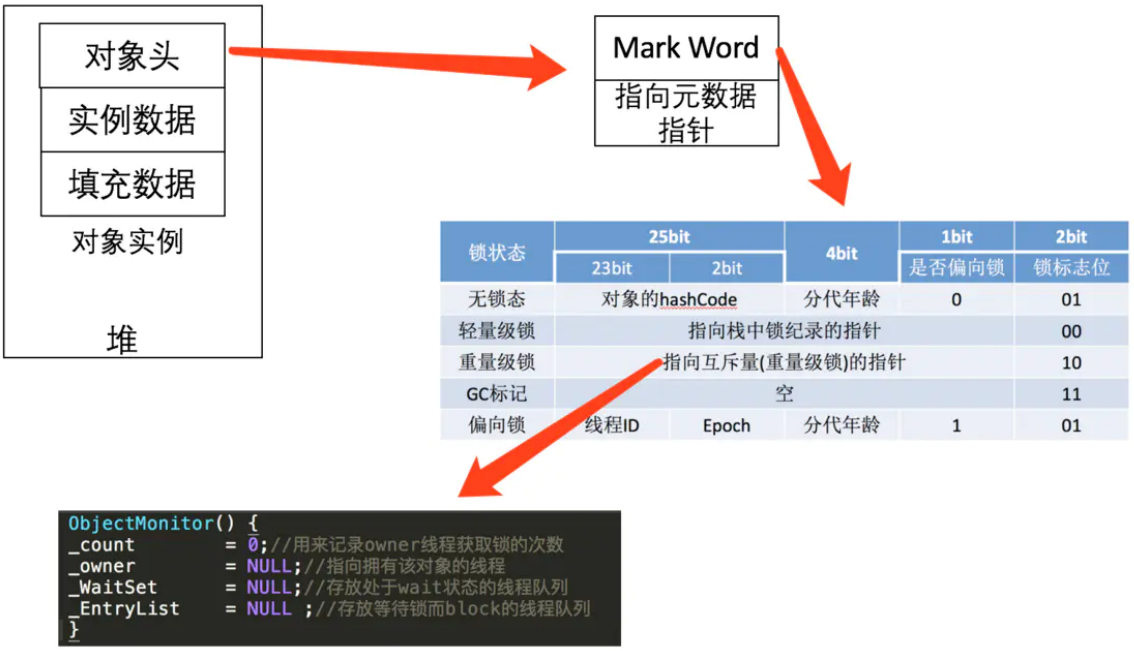
https://www.jianshu.com/p/e624460c645c

操作系统原生提供了信号量（Semaphore）和互斥量（Mutex），开发者用它们也能实现与管程相同的功能，但是信号量和互斥量都是低级原语，使用它们时必须手动编写wait和

signal逻辑，要特别小心。一旦wait/signal逻辑出错，分分钟造成死锁。管程可以对开发者屏蔽掉这些细节，在语言内部实现，更加简单易用。
由上面的叙述可知，管程并不像它的名字所说的一样是个简单的程序，而是由以下3个元素组成：

- **临界区**；
- **条件变量**，用来维护因不满足条件而阻塞的线程队列。注意，条件由开发者在业务代码中定义，条件变量只起指示作用，亦即条件本身并不包含在条件变量内；
- **Monitor对象**，维护管程的入口、临界区互斥量（即锁）、临界区和条件变量，以及条件变量上的阻塞和唤醒操作。

每个Java对象天生自带了一把看不见的锁。可以把它描述为一个对象，每个对象都存在一个monitor与之关联。对象头中指向重量级锁即指向monitor的起始地址。对象与monitor关系有多种实现方式：monitor可以与线程一起创建销毁，或者线程试图获取对象锁时自动生成。当monitor被某个线程持有后，它便处于锁定状态。



标志位为10，即重量级锁定时，Mark Word会保存指向重量级锁的指针。在HotSpot代码中，是指向ObjectMonitor类型的指针。ObjectMonitor的构造方法如下所示：
HotSpot monitor实现源码

```
// initialize the monitor, exception the semaphore, all other fields
 // are simple integers or pointers
 ObjectMonitor() {
 _header = NULL;
 _count = 0;
 _waiters = 0,
```

```
_recursions = 0;
_object = NULL;
_owner = NULL; //指向持有objectMonitor对象的线程。
 _WaitSet = NULL; //等待池
 _WaitSetLock = 0 ;
 _Responsible = NULL ;
 _succ = NULL ;
 _cxq = NULL ;
 FreeNext = NULL ;
 _EntryList = NULL ; //锁池
 _SpinFreq = 0 ;
 _SpinClock = 0 ;
 OwnerIsThread = 0 ;
 _previous_owner_tid = 0;
 }
```

_owner：持有该ObjectMonitor的线程的指针；

_count：线程获取管程锁的次数；

_waiters：处于等待状态的线程数；

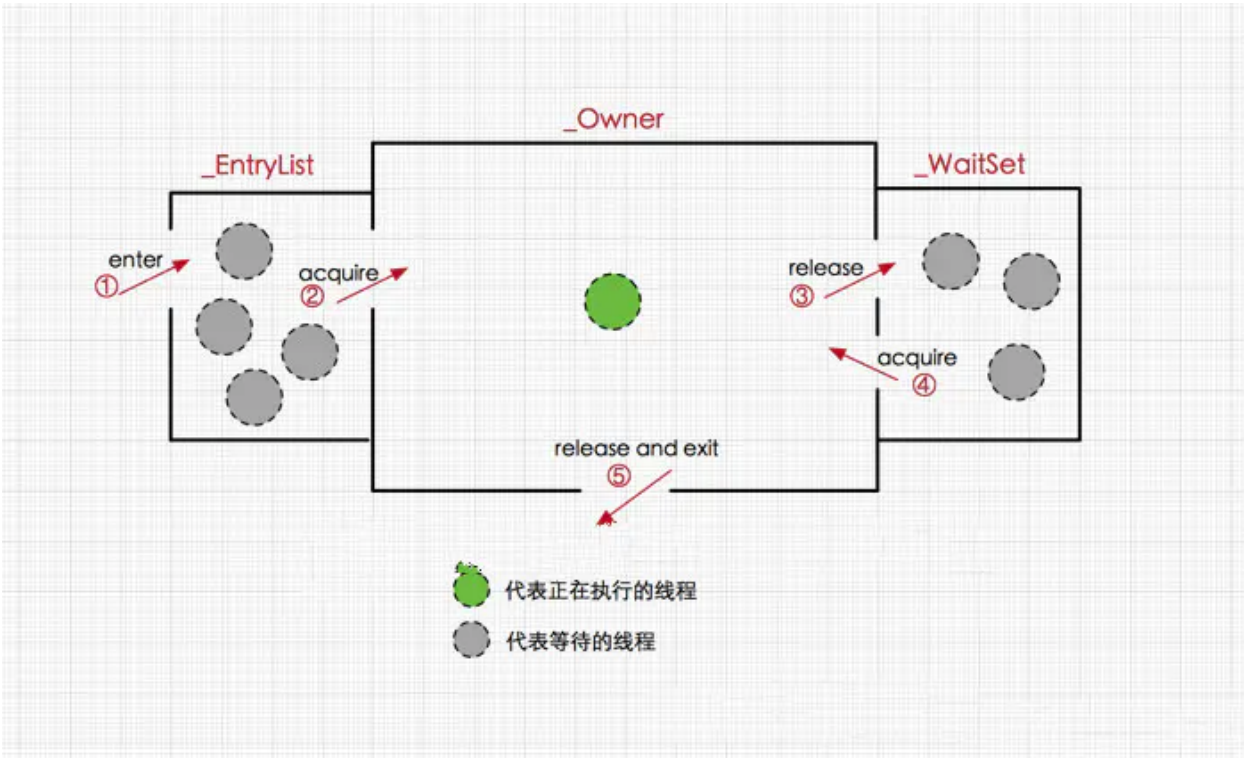
_recursions：管程锁的重入次数；

_EntryList：管程的入口线程队列；

_WaitSet：处于等待状态的线程队列。

当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(**monitor对象的指针存在于每个Java对象的对象头中，获取对象的锁，就是被这个对象的objectMoniter标记指针**)的持有权。当计数器为0则可以成功获取，获取后将管程锁计数器设为1也就是加1。如果当前线程已经拥有objectref的monitor的特有权，那么它可以重入这个monitor，重入时的计算器值也会被执行。相应的在执行 monitorexit 指令后，将锁计数器设为-1，锁计数为0时，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

但当一个线程再次请求自己持有对象锁的临界资源时，属于重入。synchronized是可重入的。



monitorenter的逻辑在InterpreterRuntime::monitorenter()方法中，其源码如下:

```
IRT_ENTRY_NO_ASYNC(void, InterpreterRuntime::monitorenter(JavaThread* thr
ead, BasicObjectLock* elem))
#ifdef ASSERT
 thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
 if (PrintBiasedLockingStatistics) {
   Atomic::inc(BiasedLocking::slow_path_entry_count_addr());
}
 Handle h_obj(thread, elem->obj());
 assert(Universe::heap()->is_in_reserved_or_null(h_obj()),
        "must be NULL or an object");
 if (UseBiasedLocking) {
   // Retry fast entry if bias is revoked to avoid unnecessary inflation
   ObjectSynchronizer::fast_enter(h_obj, elem->lock(), true, CHECK);
} else {
   ObjectSynchronizer::slow_enter(h_obj, elem->lock(), CHECK);
}
 assert(Universe::heap()->is_in_reserved_or_null(elem->obj()),
        "must be NULL or an object");
#ifdef ASSERT
 thread->last_frame().interpreter_frame_verify_monitor(elem);
#endif
IRT_END
```

该方法会根据是否启用偏向锁（UseBiasedLocking）来决定是使用偏向锁（调用ObjectSynchronizer::fast_enter()方法）还是轻量级锁（调用ObjectSynchronizer::slow_enter()方法）。如果不能获取到锁，就会按偏向锁→轻量级锁→重量级锁的顺序膨胀，而重量级锁就是与ObjectMonitor（即管程）相关的锁。

JDK1.6之后对synchronized的优化

https://www.cnblogs.com/wuqinglong/p/9945618.html

synchronized在早期属于重量级锁，线程之间的切换需要从用户状态切换到内核状态，开销大。优化的思路：尽量减少重量级锁的使用。

JDK1.6 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐膨胀升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

关于这几种优化的详细信息可以查看下面这篇文章：Java6 及以上版本对 synchronized 的优化

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, 和执行非同步代码方法的性能相差无几.	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的同步场景
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	线程交替执行的场景
重量级锁	线程竞争不适用自旋, 不会消耗CPU	线程堵塞, 响应时间缓慢	追求吞吐量, 同步快执行时间速度较长

自旋锁与自适应锁

互斥同步对性能最大的影响就是阻塞的实现，因为**挂起线程/恢复线程的操作都需要转入内核态中完成。**

在许多应用中，**共享数据的锁定状态只会持续很短的一段时间，所以仅仅为了这一点时间去挂起线程/恢复线程是得不偿失的。** 所以，虚拟机的开发团队就这样去考虑：“我们能不能让后面来的请求获取锁的线程等待一会而不被挂起呢？看看持有锁的线程是否很快就会释放锁”。**为了让一个线程等待，我们只需要让线程执行一个忙循环（自旋），这项技术就叫做自旋。**

自旋锁在 JDK1.4 之前其实就已经引入了，不过是默认关闭的，需要通过`--XX:+UseSpinning`参数来开启。JDK1.6及1.6之后，就改为默认开启的了。需要注意的是：

自旋等待不能完全替代阻塞，因为它还是要占用处理器时间。**若锁被其他线程长时间占用，会带来更多的CPU开销。**因此自旋等待的时间必须要有限度。如果自旋超过了限定次数任然没有获得锁，就应该挂起线程。**自旋次数的默认值是10次，用户可以修改**`--XX:PreBlockSpin`**来更改。**

另外，在 JDK1.6 中引入了**自适应自旋锁。**自适应的自旋锁带来的改进就是：自旋的时间不在固定了，而是和**前一次在同一个锁上的自旋时间以及锁的拥有者的状态**来决定。如果在同一个锁上自旋刚获取成功，并且持有锁的线程正在运行中，JVM会认为该锁自旋获取到锁的可能性很大，会自动增加等待时间。相反如果某个锁自旋很少获取到锁，在以后获取锁时可能省略掉自旋过程，避免浪费CPU资源。锁的预测会越来越精准，JVM会越来越“聪明”。

锁消除

锁消除是指**对于被检测出不可能存在竞争的共享数据的锁进行消除。**

锁消除主要是通过**逃逸分析**来支持，如果堆上的共享数据不可能逃逸出去被其它线程访问到，那么就可以把它们当成私有数据对待，也就可以将它们的锁进行消除。**哪些数据是私有的，不能被其他线程访问到？**

```
1 public String concatString(String s1, String s2) {
2  StringBuffer sb = new StringBuffer();
3  sb.append(s1);
4  sb.append(s2);
5  return sb.toString();
6 }
```

StringBuffer的append操作是synchronized的，由于这里StringBuffer的对象属于本地变量，属于不可能共享的资源，JVM会自动消除内部的锁。

锁粗化

如果一系列的连续操作都对**同一个对象反复加锁和解锁**，频繁的加锁操作就会导致性能损耗。

上面的示例代码中连续的 append() 方法就属于这类情况。如果虚拟机探测到由这样的一串零碎的操作都对同一个对象加锁，将会把加锁的范围扩展（粗化）到整个操作序列的外部。比如while循环中的append操作，只需要在while循环外部加锁一次就可以了。

偏向锁

减少同一线程获取锁的代价。大多数情况下，锁不存在多线程竞争，总是由同一线程多次获得。

引入偏向锁的目的和引入轻量级锁的目的很像，他们都是为了**没有多线程竞争的前提**下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。但是不同是：**轻量级锁在无竞争的情况下使用 CAS 操作去代替使用互斥量。而偏向锁在无竞争的情况下会把整个同步都消除掉。**

偏向锁的“偏”就是偏心的偏，它的意思是会偏向于第一个获得它的线程，如果在接下来的执行中，该锁没有被其他线程获取，那么持有偏向锁的线程就不需要进行同步，甚至连CAS操作也不再需要。

当锁对象第一次被线程获得的时候，进入偏向状态，标记为 1 01。同时使用**CAS**操作将**线程ID记录到 对象头的Mark Word中**，如果 CAS 操作成功，这个线程以后每次进入这个锁相关的同步块就**不需要再进行任何同步操作**，只需要检查Mark Word是偏向锁状态以及当前线程ID等于Mark Word中的Thread ID，这样就省去了大量有关锁申请的操作。

对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失。

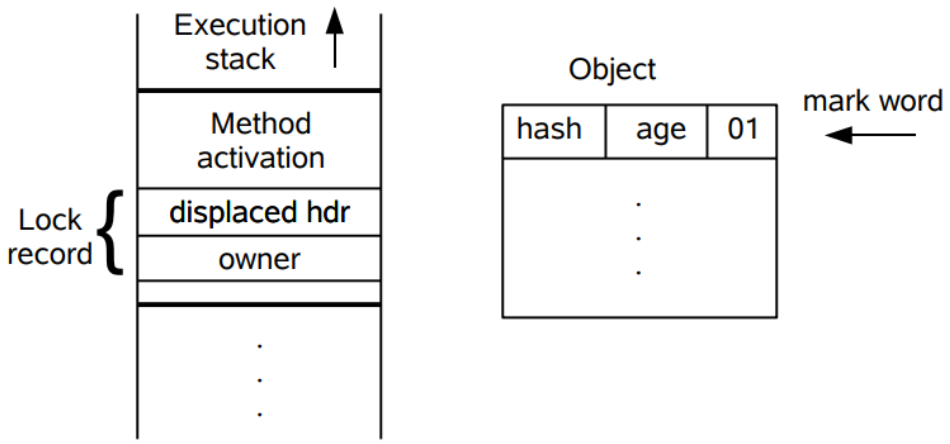
当有另外一个线程去尝试获取这个锁对象时，偏向状态就宣告结束，此时撤销偏向（Revoke Bias）后恢复到未锁定状态或者轻量级锁状态。
需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先**升级为轻量级锁。**

偏向锁状态转移原理: https://juejin.im/post/5c17964df265da6157056588
关于偏向锁的原理可以查看《深入理解Java虚拟机：JVM高级特性与最佳实践》第二版的13章第三节锁优化。

轻量级锁
轻量级锁的加锁和解锁都依赖CAS操作。
对于绝大部分锁，在整个同步周期内都是不存在竞争的。**如果没有竞争，轻量级锁使用 CAS 操作避免了使用互斥操作的开销。**
适用场景：线程交替执行同步块。
如果存在同一时间访问同一锁的情况，除了互斥量开销外，还会额外发生CAS操作，因此在有锁竞争的情况下，轻量级锁比传统的重量级锁更慢。如果锁竞争激烈，那么轻量级将很快膨胀为重量级锁。

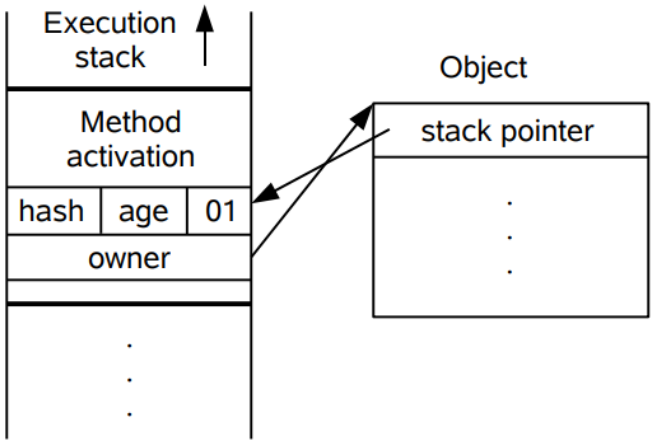
轻量级锁加锁

线程在执行同步块之前，如果锁对象标记为 0 01，说明锁对象的锁未锁定（unlocked）状态。JVM会先在**当前线程的栈帧中创建用户存储锁记录(Lock record)**的空间，并将对象头中的MarkWord复制到锁记录中。



然后线程尝试使用**CAS将对象头中的MarkWord的ptr_to_lock_record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，表示其它线程竞争锁，当前线程便尝试使用自旋来获取锁，之后再来的线程，发现是轻量级锁，就开始进行自旋。如果有两条以上的线程争用同一个锁，**那轻量级锁就不再有效，要膨胀为重量级锁，Mark Word中存储指向重量级锁（互斥量）的指针**，后面等待锁的线程要进入阻塞状态。

MarkWord中 ***ptr_to_lock_record***：指向栈中锁记录的指针。




```
|-------------------------------------------------------|----------------|
| Mark Word (32 bits) | State |
|-------------------------------------------------------|----------------|
| identity_hashcode:25 | age:4 | biased_lock:1 | lock:2 | Normal |
|-------------------------------------------------------|----------------|
```

```
| thread:23 | epoch:2 | age:4 | biased_lock:1 | lock:2 | Biased |
|-----------------------------------------------------------|----------------|
| ptr_to_lock_record:30 | lock:2 | Lightweight Locked |
|-----------------------------------------------------------|----------------|
| ptr_to_heavyweight_monitor:30 | lock:2 | Heavyweight Locked |
|-----------------------------------------------------------|----------------|
| | lock:2 | Marked for GC |
|-----------------------------------------------------------|----------------|
```

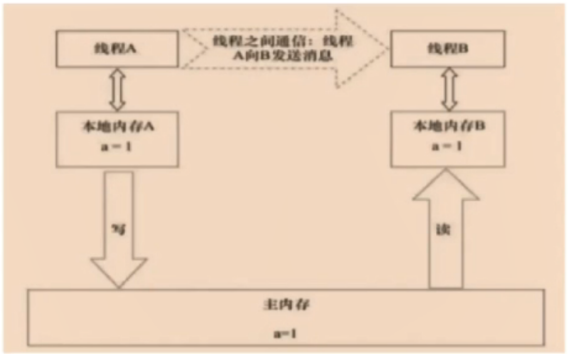
解锁过程

轻量级锁解锁时，**会使用原子的CAS操作将当前线程的锁记录替换回到对象头。**如果替换成功，整个同步过程就完成了。如果替换失败，说明有其他线程尝试过获取该锁（此锁已经膨胀），那就要在释放锁的同时，唤醒被挂起的线程。

锁的内存语义

当线程释放锁时，Java内存模型会把该线程对应的本地内存中的共享变量刷新到主内存中；

而当线程获取锁时，Java内存模型会把该线程对应的本地内存置为无效，从而使得被监视器保护的临界区代码必须从主内存中读取共享变量。



ReentrantLock

基于JDK中的JUC.AQS实现，位于java.util.concurrent.locks包，相比于synchronized修饰对象作为锁，直接使用ReentrantLock对象，增加了各种锁的管理机制。

实现原理

继承AQS的内部类实现公平锁、非公平锁机制

```
public class ReentrantLock implements Lock, java.io.Serializable {
 private static final long serialVersionUID = 7373984872572414699L;
```

```
    private final Sync sync;
    abstract static class Sync extends AbstractQueuedSynchronizer {
    ...
    }
    static final class NonfairSync extends Sync {}
    static final class FairSync extends Sync {}
}
```

使用举例：

- **默认为非公平锁，公平锁：ReentrantLock lock = new ReentrantLock(true);**
- **必须unlock()释放，一般在finally中。**

```
public class ReentrantLockDemo implements Runnable {
    //默认是非公平锁。
// ReentrantLock lock = new ReentrantLock();
    //参数true为公平锁。 使用公平锁，两个线程会交替获得到锁。
    ReentrantLock lock = new ReentrantLock(true);

    @Override
    public void run() {
    while (true) {
     lock.lock();
     try {
     System.out.println(Thread.currentThread().getName() + " get lock");
     Thread.sleep(1000);
     } catch (InterruptedException e) {
     e.printStackTrace();
     } finally {
     lock.unlock(); // 确保释放锁，避免发生死锁。
     }
     }
     }

     public static void main(String[] args) {
     ReentrantLockDemo reentrantLockDemo = new ReentrantLockDemo();
     Thread thread1 = new Thread(reentrantLockDemo);
     Thread thread2 = new Thread(reentrantLockDemo);
     thread1.start();
     thread2.start();
```

```
28     }
29 }
```

公平锁会顺序获取到reentrantLock，非公平锁随机获取到reentrantLock。

谈谈 synchronized 和 ReentrantLock 的相同和不同

两者都是可重入锁

“可重入锁” 指的是自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增 1，所以要等到锁的计数器下降为 0 时才能释放锁。

synchronized是关键字，ReentrantLock是类。

ReentrantLock更灵活，可以被继承，有方法和各种类变量。

synchronized 依赖于 JVM 而 ReentrantLock 依赖于 API

synchronized 操纵对象头中的Mark Word，Lock调用Unsafe()中的park()方法。

synchronized 是依赖于 JVM 实现的，前面我们也讲到了 虚拟机团队在 JDK1.6 为 synchronized 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。ReentrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock() 方法配合 try/finally 语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。

synchronized会自动释放锁(a.线程执行完同步代码会释放锁；b.线程执行过程中发生异常会释放锁)，Lock需在finally中手工释放锁（unlock()方法释放锁），否则容易造成线程死锁；

相比synchronized，ReentrantLock增加了一些高级功能。

等待可中断：ReentrantLock提供了一种能够中断等待锁的线程的机制，通过 lock.lockInterruptibly() 来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。

可实现公平锁: ReentrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。ReentrantLock默认情况是非公平的，可以通过 ReentrantLock类的ReentrantLock(boolean fair)构造方法来制定是否是公平的。

更精细化的控制：

判断是否有线程拥有锁：lock.getOwner()
判断是否有线程正在等待：lock.hasQueuedThreads(); lock.hasQueuedThread(Thread thread)
可实现选择性通知（锁可以绑定多个条件）：synchronized关键字与wait()和notify()/notifyAll()方法相结合可以实现等待/通知机制。ReentrantLock类当然也可以实现，借助于JUC.locks.Condition接口与newCondition()方法。

性能已不是选择标准。提倡除非需要使用 ReentrantLock 的高级功能，优先考虑使用synchronized关键字来进行同步，优化后的synchronized和ReenTrantLock一样，在很多地方都是用到了CAS操作。synchronized 是 JVM 实现的一种锁机制，JVM 原生地支持它，而 ReentrantLock 不是所有的 JDK 版本都支持。并且使用 synchronized 不用担心没有释放锁而导致死锁问题，因为JVM 会确保锁的释放。

Condition是 JDK1.5 之后才有的，它具有很好的灵活性，比如可以实现多路通知功能。也就是在一个Lock对象中可以创建多个Condition实例（即对象监视器），线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在调度线程上更加灵活。 在使用notify()/notifyAll()方法进行通知时，被通知的线程是由 JVM 选择的，synchronized关键字就相当于整个 Lock 对象中只有一个Condition实例，所有的线程都注册在它一个身上。如果执行notifyAll()方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而Condition实例的**signalAll()**方法 只会唤醒注册在该Condition实例中的所有等待线程。
await(); signal(); signalAll();

Condition最典型的用法在ArrayBlockingQueue中：take的时候要保证notEmpty，put的时候要保证notFull

```
public ArrayBlockingQueue(int capacity, boolean fair) {
 if (capacity <= 0)
 throw new IllegalArgumentException();
 this.items = new Object[capacity];
 lock = new ReentrantLock(fair);
 notEmpty = lock.newCondition();
 notFull = lock.newCondition();
}

public E take() throws InterruptedException {
 final ReentrantLock lock = this.lock;
```

```
 lock.lockInterruptibly();
 try {
 //当队列为空时，试图take的线程要先等待有新的消息被加入到队列
 while (count == 0)
 notEmpty.await();
 return dequeue();
 } finally {
 lock.unlock();
 }
}


/**
 * Inserts the specified element at the tail of this queue, waiting
 * for space to become available if the queue is full.
 */
public void put(E e) throws InterruptedException {
 Objects.requireNonNull(e);
 final ReentrantLock lock = this.lock;
 lock.lockInterruptibly();
 try {
 while (count == items.length)
 notFull.await();
 enqueue(e);
 } finally {
 lock.unlock();
 }
}

/**
 * Inserts element at current put position, advances, and signals.
 * Call only when holding lock.
 */
private void enqueue(E e) {
 // assert lock.isHeldByCurrentThread();
 // assert lock.getHoldCount() == 1;
 // assert items[putIndex] == null;
 final Object[] items = this.items;
 items[putIndex] = e;
 if (++putIndex == items.length) putIndex = 0;
```

```
52    count++;
53    notEmpty.signal();
54  }
```

JMM内存模型

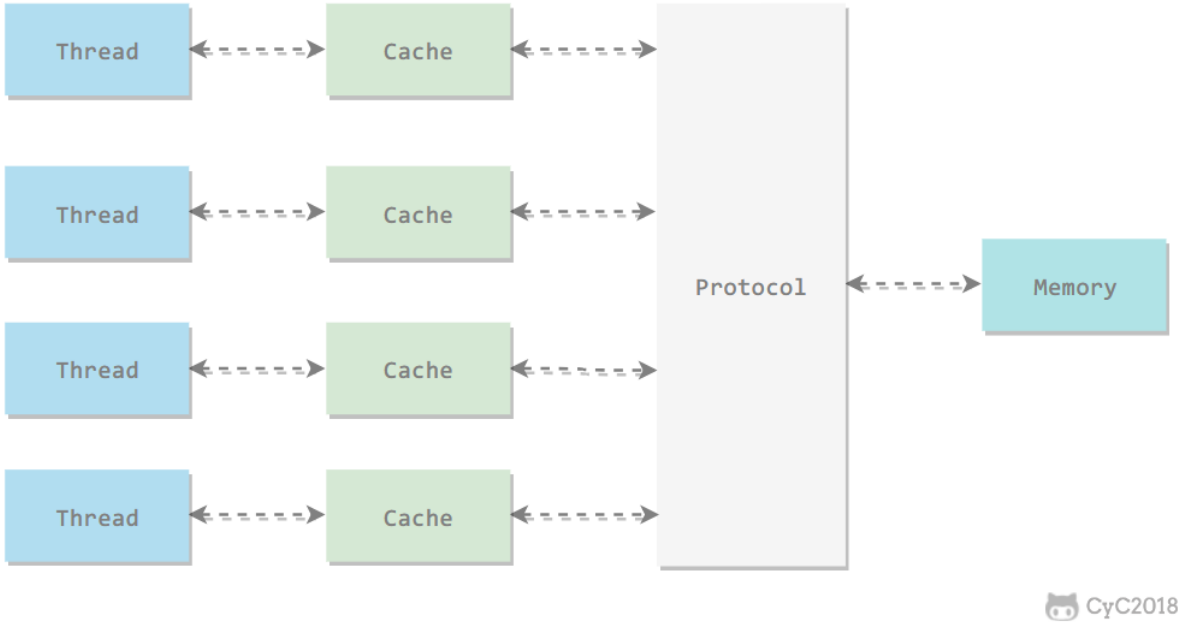
https://zhuanlan.zhihu.com/p/29881777

JMM是一种抽象的概念，并不真实存在，描述的是一组规则和规范，通过这组规范定义了Java虚拟机与计算机内存是如何协同工作的：规定了一个线程如何和何时可以看到由其他线程修改过后的共享变量的值，以及在必须时如何同步的访问共享变量。

抽象理解：主内存就是硬件的内存。为了获取更好的运行速度，虚拟机及硬件系统会让工作内存优先存储于寄存器和高速缓存中，拷贝一份主内存的数据，操作完成之后，传回主内存。

处理器上的寄存器的读写的速度比内存快几个数量级，为了解决这种速度矛盾，在它们之间加入了高速缓存。

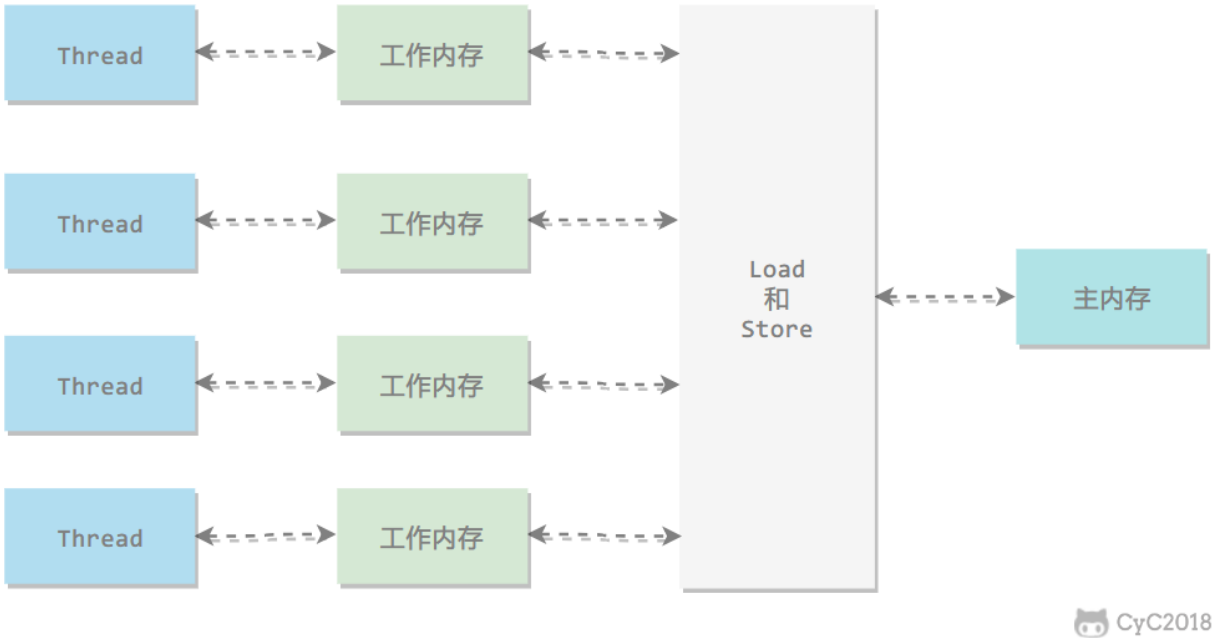
加入高速缓存带来了一个新的问题：**缓存一致性。**如果多个缓存共享同一块主内存区域，那么多个缓存的数据可能会不一致，需要一些协议来解决这个问题。



所有的变量都存储在主内存中，每个线程还有自己的工作内存，工作内存存储在高速缓存或者寄存器中，保存了该线程使用的变量的主内存副本拷贝，操作完成之后刷新回主内存。

线程只能直接操作工作内存中的变量，不同线程之间的变量值传递需要通过主内存来完成。

共享变量包括：成员变量、static变量、类信息、常量。不包括局部变量，哪怕是引用变量



JMM模型下的线程间通信：线程间通信必须要经过主内存。

如下，如果线程A与线程B之间要通信的话，必须要经历下面2个步骤：

1）线程A把本地内存A中更新过的共享变量刷新到主内存中去。

2）线程B到主内存中去读取线程A之前已更新过的共享变量。

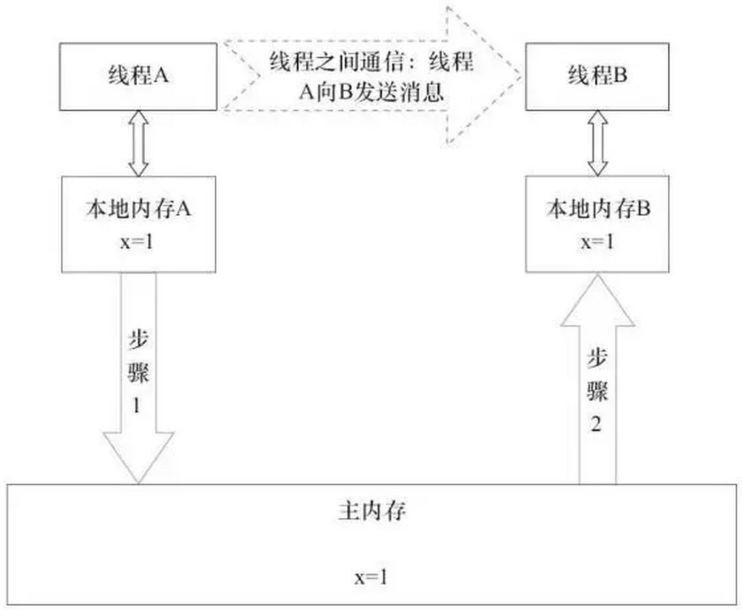


图3-2 线程之间的通信图

Java内存模型是围绕着并发编程中原子性、可见性、有序性这三个特征来建立的。

原子性

如果多个线程对同一个共享数据进行访问而不采取同步操作的话，那么操作的结果是不一致的。

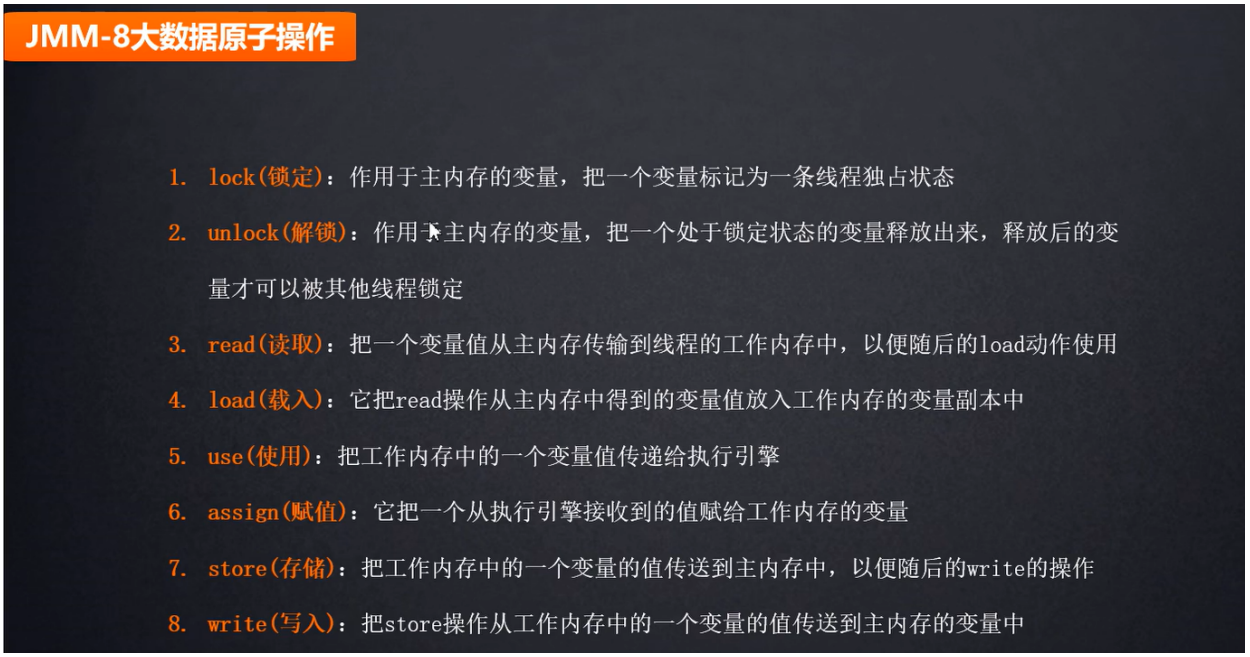
以下代码演示了 **1000 个线程同时对 共享变量 cnt 执行自增操作，操作结束之后它的值有可能小于 1000。**

```
public class ThreadUnsafeExample {
 private int cnt = 0;
 public void add() {
 cnt++;
 }

 public int get() {
 return cnt;
 }
}
public static void main(String[] args) throws InterruptedException {
 final int threadSize = 1000;
 ThreadUnsafeExample example = new ThreadUnsafeExample();
 final CountDownLatch countDownLatch = new CountDownLatch(threadSize);
 ExecutorService executorService = Executors.newCachedThreadPool();
 for (int i = 0; i < threadSize; i++) {
 executorService.execute(() -> {
 example.add();
 countDownLatch.countDown();
 });
 }
 countDownLatch.await();
 executorService.shutdown();
 System.out.println(example.get());
}
>>> 997
```

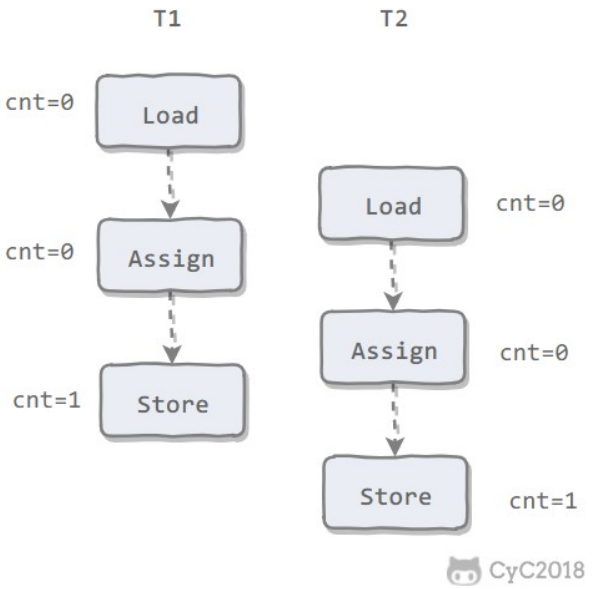
原因：共享变量cnt的自增操作不是原子性的。

Java 内存模型保证了 read、load、use、assign、store、write、lock 和 unlock 操作具有原子性，例如对一个 int 类型的变量执行 assign 赋值操作，这个操作就是原子性的。



为了方便讨论，将内存间的交互操作简化为 3 个：load、assign、store。

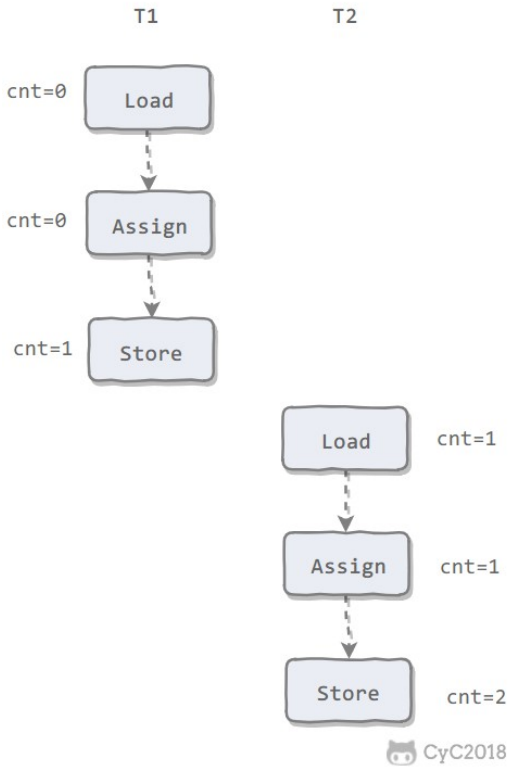
下图演示了两个线程同时对 cnt 进行操作，load、assign、store 这一系列操作整体上看不具备原子性，那么在 T1 修改 cnt 并且还没有将修改后的值写入主内存，T2 依然可以读入旧值。可以看出，这两个线程虽然执行了两次自增运算，但是主内存中 cnt 的值最后为 1 而不是 2。因此对 int 类型读写操作满足原子性只是说明 load、assign、store 这些单个操作具备原子性。



怎样解决原子性的问题呢?

两种方法

1. AtomicInteger 能保证多个线程修改的原子性。



使用 AtomicInteger 重写之前线程不安全的代码之后得到以下线程安全实现：

```
public class AtomicExample {
 private AtomicInteger cnt = new AtomicInteger();

 public void add() {
 cnt.incrementAndGet();
 }

 public int get() {
 return cnt.get();
  }
}
```

2. 除了使用原子类之外，也可以使用 synchronized 互斥锁来保证操作的原子性。

它对应的内存间交互操作为：lock 和 unlock，在虚拟机实现上对应的字节码指令为 monitorenter 和 monitorexit。

```
public class AtomicSynchronizedExample {
 private int cnt = 0;

 public synchronized void add() {
 cnt++;
 }

```

```
8   public synchronized int get() {
9   return cnt;
10   }
11  }
```

用volatile是不可以的，volatile不保证原子性。

可见性——volatile

可见性指当一个线程修改了共享变量的值，其它线程能够立即得知这个修改。Java 内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值来实现可见性的。

主要有三种实现可见性的方式：

- **volatile**
- **synchronized**，对一个变量执行 unlock 操作之前，必须把变量值同步回主内存。
- **final**，被 final 关键字修饰的字段在构造器中一旦初始化完成，并且没有发生 this 逃逸（其它线程通过 this 引用访问到初始化了一半的对象），那么其它线程就能看见 final 字段的值。

对前面的线程不安全示例中的 cnt 变量使用 volatile 修饰，不能解决线程不安全问题，因为 volatile 并不能保证操作的原子性。

但是对于变量中的boolean类型，修改操作是原子性的，可以用volatile修饰，不需要加synchronized的重锁就可以实现同步。和interrupt()作用相同

```
1  public class VolatileDemo {
2   volatile boolean shutdown;
3   public void close() {
4   this.shutdown = true;
5   }
6   public void dowork() {
7   while (!shutdown) {
8   System.out.println("safe....");
9   }
10   }
11  }
```

在CAS原理 value 是一个volatile变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。

volatile为何立即可见

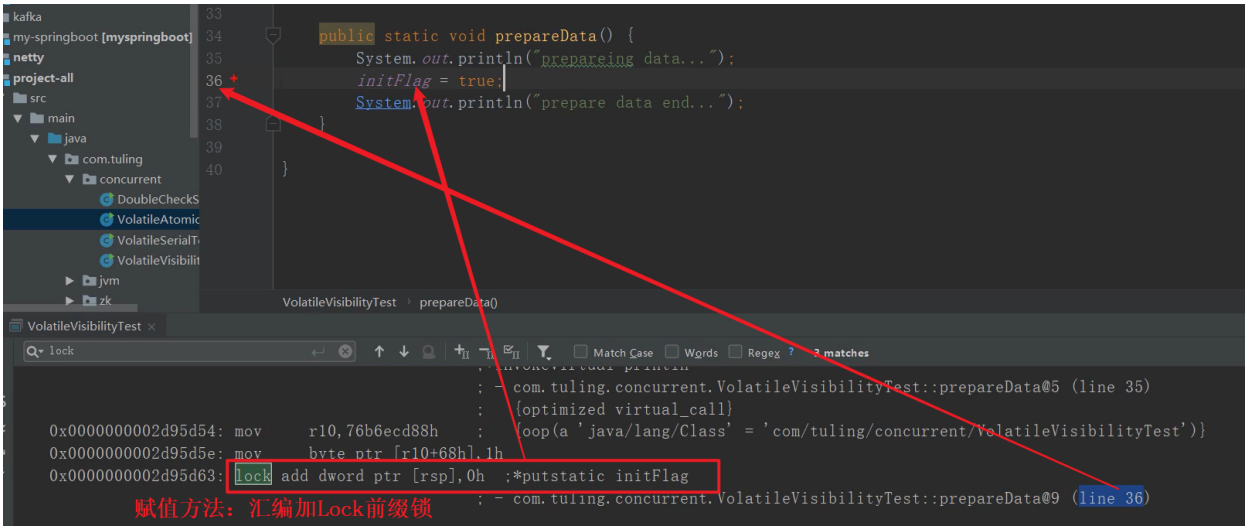
cpu缓存当中数据的四种状态：独占，共享，已修改，已失效。

volitile依赖于底层的汇编指令——lock前缀指令，当前缓存行的数据立即写回内存，Lock会对CPU总线和高速缓存加锁，加锁期间其他线程不能读写数据。相当于对store和write原子操作加锁。

lock前缀指令作用：

会将当前处理器缓存行的指令立即写回系统内存；

这个写回内存的操作会引起其他在CPU里缓存了该内存地址的数据无效（MESI总线一致协议）



会触发mesi缓存一致性协议：工作区（缓存）修改的变量值通过总线传入内存时，其他CPU能**嗅探（总线嗅探机制）**到主存区的值已经改了，会立即失效自己缓存区中的数据。在下一轮指令周期，发现没有值（已失效），会从内存重新load数据。上面的对store和write加锁也保证了重新load到的数据一定是正确的修改后的，没修改完之前不会释放锁，不能load，不会重复读到还没修改前的。

有序性

有序性是指：在本线程内观察，所有操作都是有序的。在一个线程观察另一个线程，所有操作都是无序的，无序是因为发生了指令重排序。在 Java 内存模型中，允许编译器和处理器对指令进行重排序，重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

为什么要指令重排？在不改变程序执行结果的前提下，尽可能的提高并行度。

volatile 关键字通过添加内存屏障的方式来禁止指令重排，即重排序时不能把后面的指令放到内存屏障之前。

更详细的volatile内存屏障

也可以通过 synchronized 来保证有序性，它保证每个时刻只有一个线程执行同步代码，相当于是让线程顺序执行同步代码。

volatile禁止指令重排举例：

单线程时允许这样的指令重排，但多线程不行：

```
uniqueInstance = new Singleton(); 这段代码其实是分为三步执行：
为 uniqueInstance 分配内存空间
初始化 uniqueInstance
将 uniqueInstance 指向分配的内存地址
```

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1->3->2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被初始化。

使用 volatile 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行。

```
//单例的双重检测实现
class Singleton{
    private volatile static Singleton instance = null;

    private Singleton() {

    }

    public static Singleton getInstance() {
         if(instance==null) {
             synchronized (Singleton.class) {
                 if(instance==null)
                     instance = new Singleton();
             }
         }
         return instance;
     }
}
```

说说 synchronized 关键字和 volatile 关键字的区别

synchronized 关键字和 volatile 关键字是两个互补的存在，而不是对立的存在！

- volatile的本质是告诉JVM当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取；synchronized是锁住当前变量，只有当前线程可以访问该变量，其他线程被阻塞直到当前线程完成操作为止。
- volatile 关键字是线程同步的轻量级实现，所以volatile 性能肯定比synchronized关键字要好。但是volatile 关键字只能用于变量而 synchronized 关键字可以修饰方法以及代码块。
- volatile 关键字能保证数据的可见性，但不能保证数据的原子性。synchronized 关键字两者都能保证。
- volatile关键字主要用于解决变量在多个线程之间的可见性，而 synchronized 关键字解决的是多个线程之间访问资源的同步性。

happens-before，先行发生原则

满足happens-before原则时，不会进行指令重排。A的操作结果需要对B可见，则A和B存在happens-before关系。反过来说，A和B存在happens-before关系，A对共享变量的操作对B可见。

单一线程原则：在一个线程内，在程序前面的操作先行发生于后面的操作。

只对单线程有效，保证单线程结果一致

管程锁定规则：一个 unlock 操作先行发生于后面对同一个锁的 lock 操作。

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。

volatile保证了线程的可见性

线程启动规则：Thread 对象的 start() 方法调用先行发生于此线程的每一个动作。

线程加入规则：Thread 对象的结束先行发生于 join() 方法返回。

B.join()，B在结束之前对共享变量的修改对A是可见的

线程中断规则： 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 interrupted() 方法检测到是否有中断发生。

A调用了B的中断方法，调用前A对共享变量的操作对B可见，比如标志位

对象终结规则：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。

传递性：如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那么操作 A 先行发生于操作 C。

线程安全：无锁方案

乐观锁

乐观锁和悲观锁的区别?

悲观锁假定会发生并发冲突，访问的时候都要先获得锁，保证同一个时刻只有线程获得锁，行锁，表锁等，读锁，写锁，synchronized等，都是在做操作之前先上锁。
乐观锁总是假设最好的情况，不会发生并发冲突，是无锁操作。先进行操作，如果没有其它线程争用共享数据，那操作就成功了，**否则采取补偿措施**（不断地重试，直到成功为止）。这种乐观的并发策略的许多实现都不需要将线程阻塞，因此这种同步操作称为非阻塞同步。

CAS适用于多读场景，synchronized多写场景

乐观锁适用于多读的应用类型，这样可以提高吞吐量。对于资源竞争较少（线程冲突较轻）的情况，使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。
对于资源竞争严重（线程冲突严重----多写）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。

这两种锁在Java和MySQL分别是怎么实现的?

Java乐观锁通过CAS实现，悲观锁通过synchronize、ReentrantLock实现。
MySQL乐观锁通过MVCC，也就是版本实现，悲观锁通过select... for update加上排它锁实现。

CAS

即compare and swap（比较与交换），是一种有名的无锁算法，乐观锁的一种实现方式。在Java中java.util.concurrent.atomic包下面的原子变量类和StampedLock的stampedLock.tryOptimisticRead() CAS实现的。

支持原子更新操作，适用于计数器，序列发生器（变量自增工具）等场景。

CAS 操作包含三个操作数 —— **内存位置（V）、预期原值（A）和新值(B)。** 当执行操作时，只有当 V 位置处的值等于 A，才将 V 的值更新为 B。否则，不更新，用户可以重新获取value(volatile修饰的)，再进行重试。

CAS并发原语提现在Java语言中就是sun.miscUnSafe类中的各个方法。调用UnSafe类中的CAS方法,JVM会帮我实现CAS汇编指令。这是一种**完全依赖于硬件功能**，通过它实现了原子操作。由于CAS是一种系统原语，不会被打断。

CAS/乐观锁的缺点

1. ABA 问题

如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 "ABA"问题。

ABA的解决
JDK1.5可以利用AtomicStampedReference类来解决这个问题，
AtomicStampedReference内部不仅维护了对象值，还维护了一个**时间戳**。当AtomicStampedReference对应的数值被修改时，除了更新数据本身外，还必须要更新时间戳，对象值和时间戳都必须满足期望值，写入才会成功
https://blog.csdn.net/lixinkuan328/article/details/94319775

2 竞争激烈场景下，循环时间长，开销大

自旋CAS（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给CPU带来非常大的执行开销。 因此CAS只适合多读场景。

> 如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

3. 只能保证一个共享变量的原子操作

CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5开始，提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作。所以我们可以使用锁或者利用AtomicReference类把多个共享变量合并成一个共享变量来操作。

原子类

J.U.C 包里面的整数原子类的方法大量调用了 Unsafe 类的 CAS 操作。

AtomicInteger

对于之前的共享变量自增的例子，可以采用加锁方案实现线程安全。也可以采用CAS的无锁方案：

```
public class CASCase {
 private AtomicInteger count = new AtomicInteger();

 public void increment() {
 count.incrementAndGet();
 }

 //使用AtomicInteger之后，不需要加锁，也可以实现线程安全。
 public int getCount() {
  return count.get();
  }
}
```

incrementAndGet() 返回值是新值 getAndIncrement()返回值是更新前的值

还可以传入加的值：addAndGet(1) getAndAdd(1)

以下代码是 getAndAddInt() 源码，var1 指示对象内存地址，var2 指示该字段相对对象内存地址的偏移，var4 指示操作需要加的数值，这里为 1。通过 getIntVolatile(var1, var2) 得到旧的预期值，通过调用 compareAndSwapInt() 来进行 CAS 比较，如果该字段内存地址中的值等于 var5，那么就更新内存地址为 var1+var2 的变量为 var5+var4。

可以看到 getAndAddInt() 在一个循环中进行，发生冲突的做法是不断的进行重试。

```
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;
}
```

AQS

```
/** 先入先出的等待队列
 * Head of the wait queue, lazily initialized. Except for
 * initialization, it is modified only via method setHead. Note:
 * If head exists, its waitStatus is guaranteed not to be
```

```
 * CANCELLED.
 */
private transient volatile Node head;

/**
 * Tail of the wait queue, lazily initialized. Modified only via
 * method enq to add new wait node.
 */
private transient volatile Node tail;

/**
 * The synchronization state.
 */
private volatile int state;
```