jdk源码&多线程&高并发-【阶段1、深入多线程】

讲师简介

smart哥，互联网悍将，历经从传统软件公司到大型互联网公司的洗礼，入行即在中兴通讯等大型通信公司担任项目leader，后随着互联网的崛起，先后在美团支付等大型互联网公司担任架构师，公派旅美期间曾与并发包大神Doug Lea探讨java多线程等最底层的核心技术。对互联网架构底层技术有相当的研究和独特的见解，在多个领域有着丰富的实战经验。

码炫课堂技术交流群

群名称:码炫课堂java架构群1
群　号:963060292

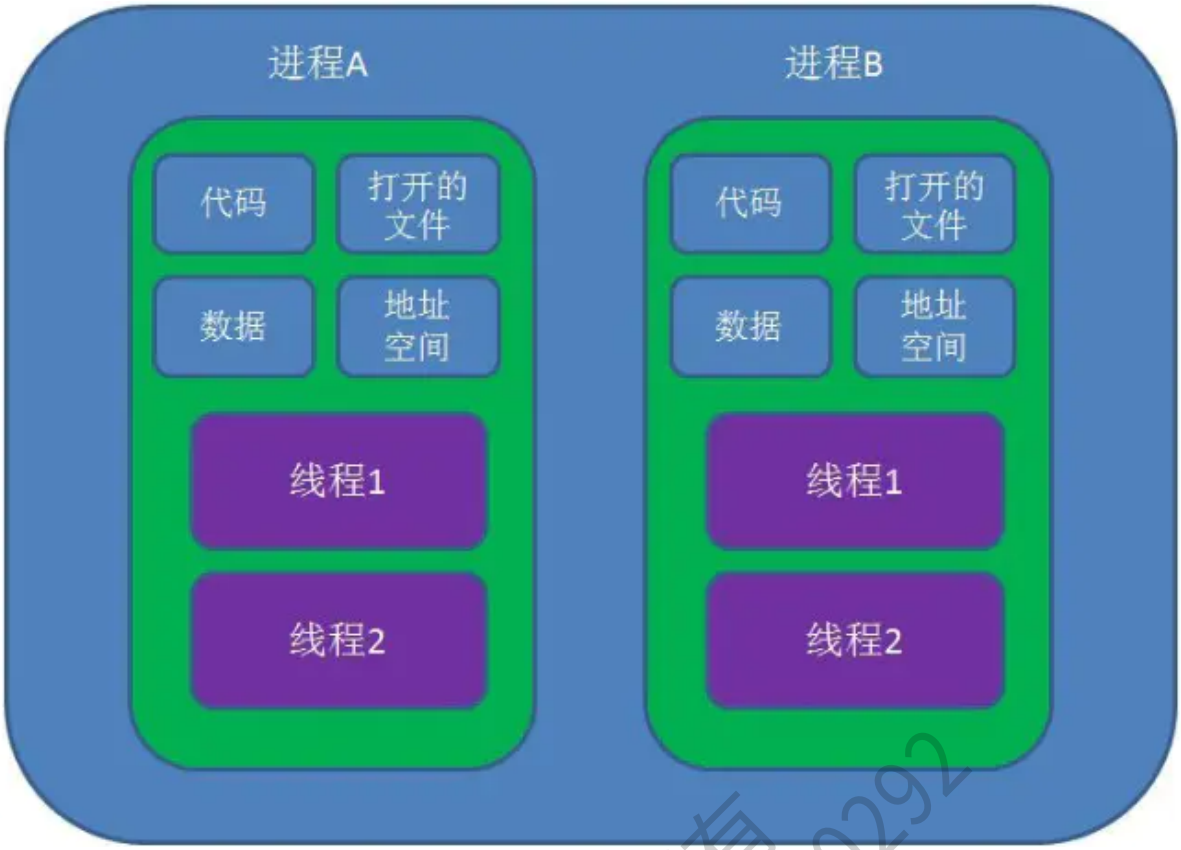
一、java多线程实现及API详解

1、进程 线程 协程的概念和区别

1）什么是进程和线程？

进程是应用程序的启动实例，进程拥有代码和打开的文件资源、数据资源、独立的内存空间。

线程从属于进程，是程序的实际执行者，一个进程至少包含一个主线程，也可以有更多的子线程，线程拥有自己的栈空间。

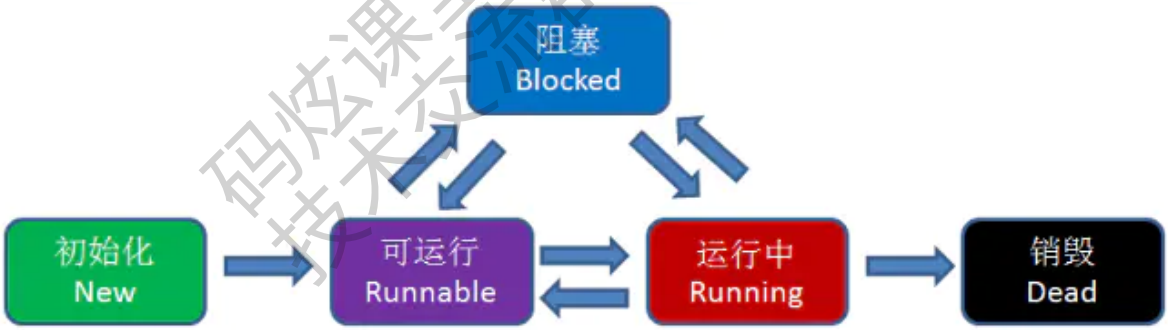


操作系统中的进程和线程

对操作系统而言，**线程是最小的执行单元，进程是最小的资源管理单元。**无论是进程还是线程，都是由操作系统所管理的。

线程的状态

线程具有五种状态：初始化、可运行、运行中、阻塞、销毁



线程状态的转化关系

new Thread（ ）.start（ ）　　run(){ thread.sleep(3000);}

进程与线程的区别

1. 进程是CPU资源分配的基本单位，线程是独立运行和独立调度的基本单位（CPU上真正运行的是线程）。
2. 进程拥有自己的资源空间，一个进程包含若干个线程，线程与CPU资源分配无关，多个线程共享同一进程内的资源。
3. 线程的调度与切换比进程快很多。

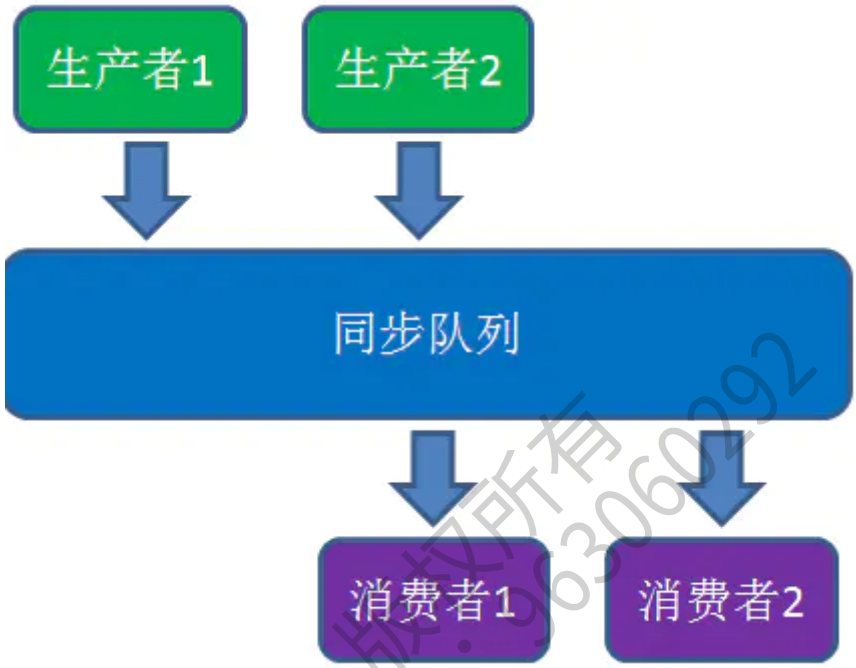
经验：

CPU密集型代码(各种循环处理、计算等等)：使用多进程。很少跟硬盘打交道，io少，阻塞少了

IO密集型代码(文件处理、网络爬虫等)：使用多线程。跟硬盘，网络打交道比较多，硬盘io，网络io比较多

2）线程之间是如何进行协作的呢？

最经典的例子是生产者/消费者模式，即若干个生产者线程向队列中生产数据，若干个消费者线程从队列中消费数据。



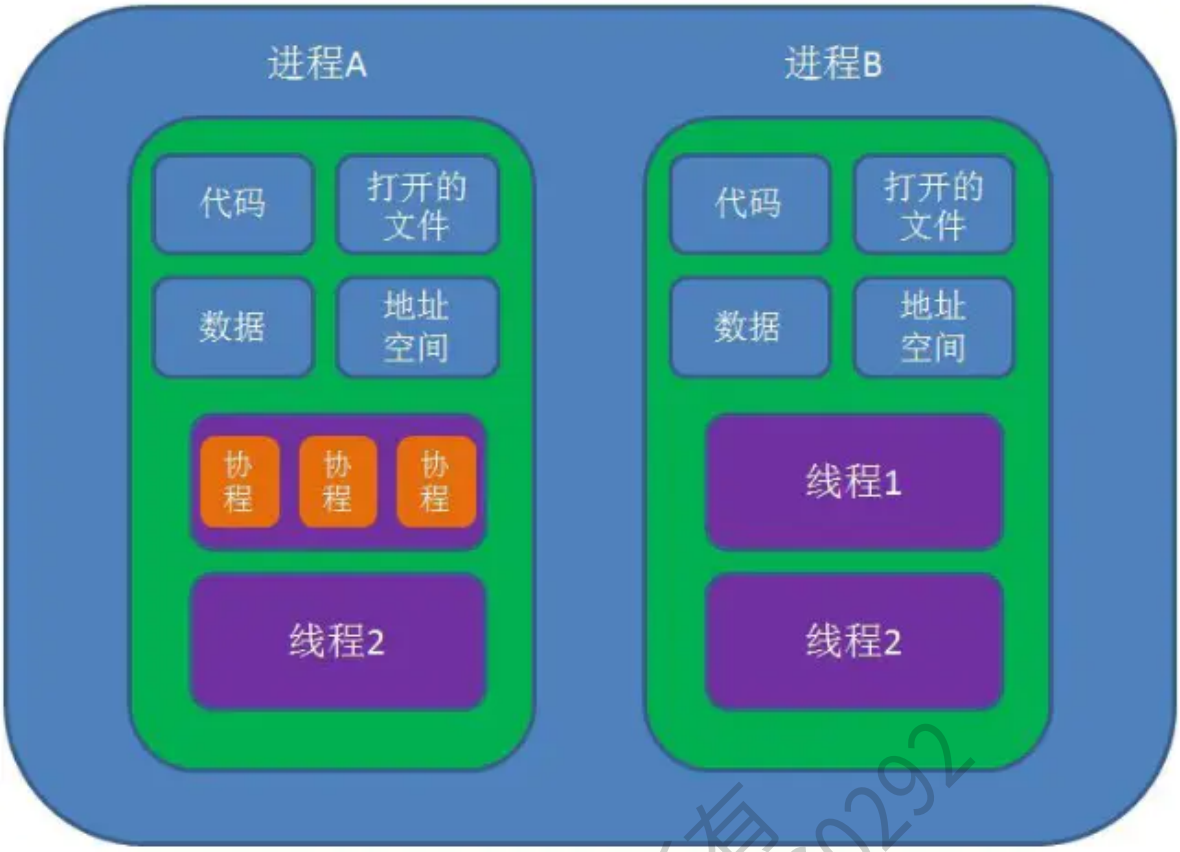
生产者/消费者模式

生产者/消费者模式的性能问题是什么？

- 涉及到同步锁
- 涉及到线程阻塞状态和可运行状态之间的切换
- 设置到线程上下文的切换

3）什么是协程？

协程（Coroutines）是一种比线程更加轻量级的存在，正如一个进程可以拥有多个线程一样，一个线程可以拥有多个协程。



操作系统中的协程

协程不是被操作系统内核所管理的，而是完全由程序所控制，也就是在用户态执行。这样带来的好处是性能大幅度的提升，因为不会像线程切换那样消耗资源。

协程不是进程也不是线程，而是一个特殊的函数，这个函数可以在某个地方挂起，并且可以重新在挂起处外继续运行。所以说，协程与进程、线程相比并不是一个维度的概念。

一个进程可以包含多个线程，一个线程也可以包含多个协程。简单来说，一个线程内可以由多个这样的特殊函数在运行，但是有一点必须明确的是，一个线程的多个协程的运行是串行的。如果是多核CPU，多个进程或一个进程内的多个线程是可以并行运行的，但是一个线程内协程却绝对是串行的，无论CPU有多少个核。毕竟协程虽然是一个特殊的函数，但仍然是一个函数。一个线程内可以运行多个函数，但这些函数都是串行运行的。当一个协程运行时，其它协程必须挂起。

4）进程、线程、协程的对比

- 协程既不是进程也不是线程，协程仅仅是一个特殊的函数，协程与进程和线程不是一个维度的。
- 一个进程可以包含多个线程，一个线程可以包含多个协程。
- 一个线程内的多个协程虽然可以切换，但是多个协程是串行执行的，只能在一个线程内运行，没法利用CPU多核能力。
- 协程与进程一样，切换是存在上下文切换问题的。

上下文切换（画图）

- 进程的切换者是操作系统，切换时机是根据操作系统自己的切换策略，用户是无感知的。进程的切换内容包括页全局目录、内核栈、硬件上下文，切换内容保存在内存中。进程切换过程是由“用户态到内核态到用户态”的方式，切换效率低。
- 线程的切换者是操作系统，切换时机是根据操作系统自己的切换策略，用户无感知。线程的切换内容包括内核栈和硬件上下文。线程切换内容保存在内核栈中。线程切换过程是由“用户态到内核态到用户态”， 切换效率中等。
- 协程的切换者是用户（编程者或应用程序），切换时机是用户自己的程序所决定的。协程的切换内容是硬件上下文，切换内存保存在用户自己的变量（用户栈或堆）中。协程的切换过程只有用户态，即没有陷入内核态，因此切换效率高。

需求：100w的线程和100w的协程，同时做运算，处理200w次运算处理

java-agent

5）线程vs协程性能对比

java线程案例

```
package com.mx.lang.Object;

public class JavaThread {


    /**
     * 10w个线程，每个线程处理2百万次运算
     * @param argus
     * @throws InterruptedException
     */
    public static void main(String[] argus) throws InterruptedException {
        long begin = System.currentTimeMillis();
        int threadLength = 100000;//10w
        Thread[] threads = new Thread[threadLength];
        for (int i = 0; i < threadLength; i++) {
            threads[i] = new Thread(() -> {
                calc();
            });
        }


        for (int i = 0; i < threadLength; i++) {
            threads[i].start();
        }
        for (int i = 0; i < threadLength; i++) {
            threads[i].join();
        }
        System.out.println(System.currentTimeMillis() - begin);
    }


    //200w次计算
    static void calc() {
        int result = 0;
        for (int i = 0; i < 10000; i++) {
            for (int j = 0; j < 200; j++) {
                result += i;
            }
        }
    }
}
```

java协程案例

```
package com.mx.lang.Object;

import co.paralleluniverse.fibers.Fiber;
```

```
import java.util.concurrent.ExecutionException;

public class JavaFiber {
    /**
     * 10w个协程，每个协程处理2百万次运算
     * @param argus
     * @throws InterruptedException
     */
    public static void main(String[] argus) throws ExecutionException,
InterruptedException {
        long begin = System.currentTimeMillis();
        int fiberLength = 100000;//10w
        Fiber<Void>[] fibers = new Fiber[fiberLength];
        for (int i = 0; i < fiberLength; i++) {
            fibers[i] = new Fiber(() -> {
                calc();
            });
        }


        for (int i = 0; i < fiberLength; i++) {
            fibers[i].start();
        }
        for (int i = 0; i < fiberLength; i++) {
            fibers[i].join();
        }
        System.out.println(System.currentTimeMillis() - begin);
    }

    //200w次计算
    static void calc() {
        int result = 0;
        for (int i = 0; i < 10000; i++) {
            for (int j = 0; j < 200; j++) {
                result += i;
            }
        }
    }
}
```


查看jvm此时有多少线程？

10w协程分配给3个线程

结论：25w个协程共用一个线程（一个线程中跑多个协程，协程不需要调度，对内核透明），4个线程一个100w个协程。

上下文切换是有成本的。因此，思考一个问题：多线程一定快吗？

代码演示：

```
package com.mx.lang.Object;

import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;
```

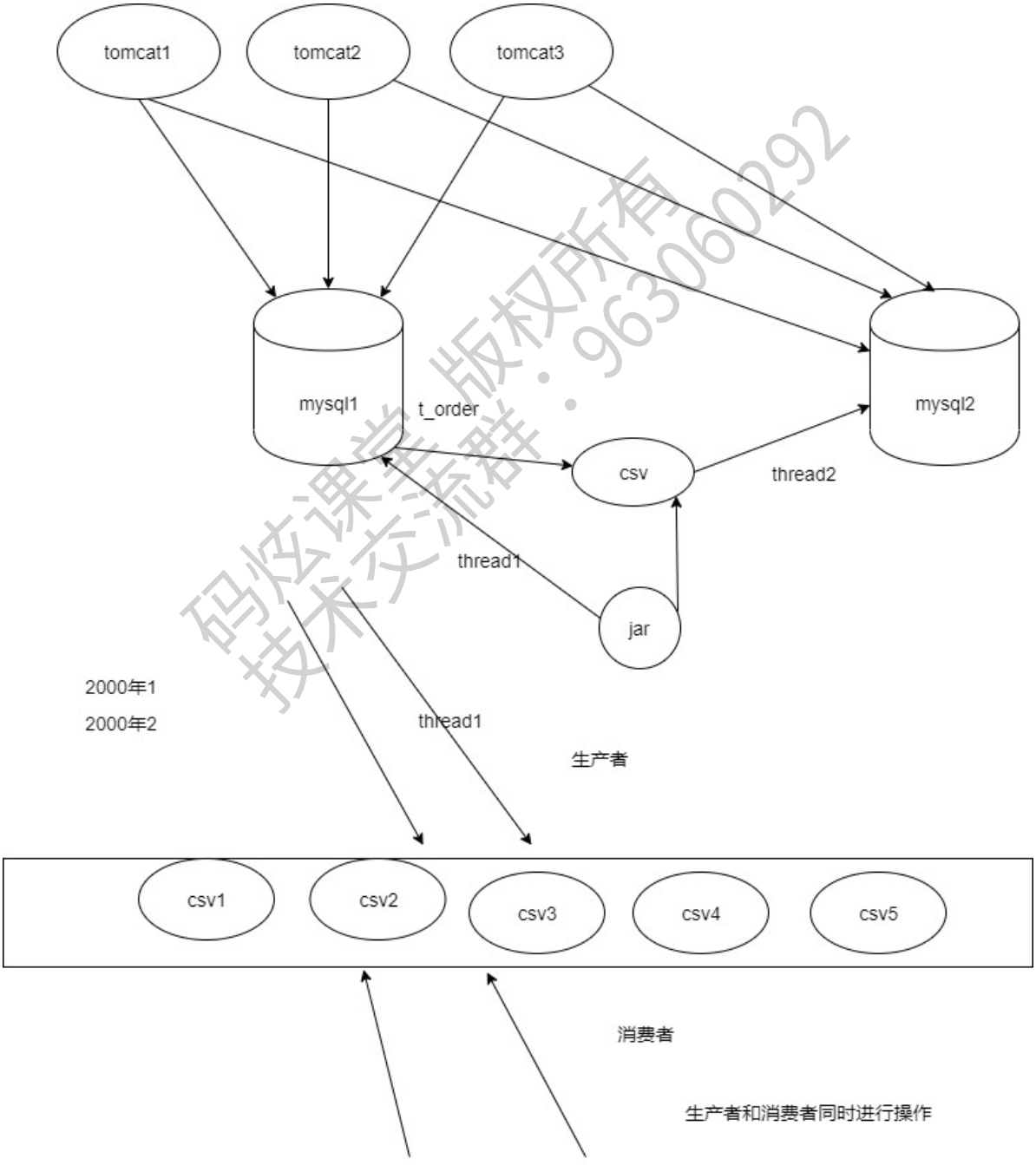
```java
/**
 * 验证多线程是否就一定快
 */
public class ConcurrencyTest {
    private static final long count = 10000000001;

    public static void main(String[] args) throws InterruptedException {
        concurrency();
        serial();
    }

    private static void concurrency() throws InterruptedException {
        long start = System.currentTimeMillis();
//        Thread thread = new Thread(new Runnable() {
//            @Override
//            public void run() {
//                int a = 0;
//                for (long i = 0; i < count; i++) {
//                    a += 5;
//                }
//            }
//        });

        FutureTask task=new FutureTask<Integer>(new Callable<Integer>(){
            @Override
            public Integer call() throws Exception {
                int a = 0;
                for (long i = 0; i < count; i++) {
                    a += 5;
                }
                return a;
            }
        });

        Thread thread2 = new Thread(task);
        thread2.start();
        int b = 0;
        for (long i = 0; i < count; i++) {
            b--;
        }
        //两种方式计算才准：1、用join 2、task.get() 区别是计算总时间放的位置
//        thread2.join();
//        long time = System.currentTimeMillis() - start;
        try {
            System.out.println("b=" + b + ",a=" + task.get() );
            long time = System.currentTimeMillis() - start;
            System.out.println("concurrency :" + time + "ms");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void serial() {
        long start = System.currentTimeMillis();
        int a = 0;
        for (long i = 0; i < count; i++) {
            a += 5;
```

```
        }
        int b = 0;
        for (long i = 0; i < count; i++) {
            b--;
        }
        long time = System.currentTimeMillis() - start;
        System.out.println("serial:" + time + "ms,b=" + b + ",a=" + a);
    }
}
```

2、案例：数据迁移

需求：多线程实现表数据迁移（画图说明）

图：



1）jvm进程被拉起后里面有哪些线程？分别是有什么作用？

```
 /**  * 获取所有线程  */Map<Thread, StackTraceElement[]> allStackTraces = Thread.getAllStackTraces();Set<Thread> threads = allStackTraces.keySet();for (Thread thread : threads) {    System.out.println("线程名：" + thread.getName());}
```

结果如下：

```
线程名：Attach Listener
线程名：Finalizer
线程名：Reference Handler
线程名：Signal Dispatcher
线程名：Monitor Ctrl-Break
线程名：main
```

说明：Monitor Ctrl-Break，官方解释：

> ```
> Monitoring Thread Activity With Thread Dumps Thread dumps,
> or "thread stack traces," reveal information about
> an application's activity that can help you diagnose problems
> and better optimize application and JVM performance;
> for example, thread dumps can show the occurrence of "deadlock" conditions,
> which can seriously impact application performance.
> You can create a thread dump by invoking a control break
> (usually by pressing Ctrl-Break or Ctrl-\ or SIGQUIT on linux).
> This section provides information on working with thread dumps.
> It includes information on these subjects:
> 1.Lock Information in Thread Dumps 2.Detecting Deadlocks
> ```


2）JVM源码跟踪debug分析main方法一般流程

面试题：main方法只能是main线程调用吗？main方法只能由JVM调用吗？

代码演示：

```
public class TestMain {
    public static void main(String[] args) {
        DataTransport transport = new DataTransport();
        transport.main(new String[]{"maxuan"});
    }
}
```

结论：应该是考线程的知识，main线程其实是jvm给你启动的，因为main方法是入口，所以得有个原始线程去调他，此时其实是jvm给你创建的main线程。其实main方法没那么神秘，任何线程都可以去调他，只不过其他线程的入口是哪里呢？其实他也有自己的main线程入口。

思考问题：

JVM是如何调用main方法，是如何调用main方法的呢？

java当中的线程和操作系统的线程是什么关系？
猜想：（c++）java thread —-对应-—> **OS thread** ---原生线程（**hotspot**）ibm j9，ali 华为

GDB

JavaCalls::call_special—》java侧的main线程

演示代码：

```
package com.mx.lang.Thread;

/**************************************
 *
 * jdk源码&多线程&高并发-【阶段1、深入多线程】
 * 主讲：smart哥
 *
 **************************************/
public class TestMain {
    public static void main(String[] args) {
//        DataTransport transport = new DataTransport();
//        transport.main(new String[]{"maxuan"});
//        Thread
        System.out.println(">>>>>>in main:"+Thread.currentThread().getName());
        TestMain testMain=new TestMain();
        Thread.currentThread().setName("main-name");
        System.out.println(">>>>>>>after set name in
main:"+Thread.currentThread().getName());
        testMain.startThread();
    }

    private void startThread() {
        Thread t= new Thread(new Runnable() {
            @Override
            public void run() {
                int i=0;
                while(i<3){
                    System.out.println(">>>>>>in
thread:"+Thread.currentThread().getName());
                    i++;
                }
                i++;
                Thread.currentThread().setName("aaaa");
                System.out.println(">>>>>>in
thread,name:"+Thread.currentThread().getName());
            }});
        t.setName("bbbb");
        t.start();
    }
}
```

具体演示见视频讲解

JVM源码的角度分析main方法一般流程：

1、加载虚拟机（检查库文件libjvm.so中是否方法都已经准备好）：LoadJavaVM(jvmpath, &ifn)。

2、ok，准备好了，创建新线程：**pthread_create**，以下都是在这个线程中做的，本质是调的JavaMain方法

1）--》创建虚拟机：InitializeJVM(&vm, &env, &ifn)

---》 ifn->CreateJavaVM(pvm, (void **)penv, &args);（jni.cpp的JNI_CreateJavaVM方法）

----》创建虚拟机的过程中创建main线程：JavaThread* main_thread = new JavaThread();(注意和普通线 程的区别)。

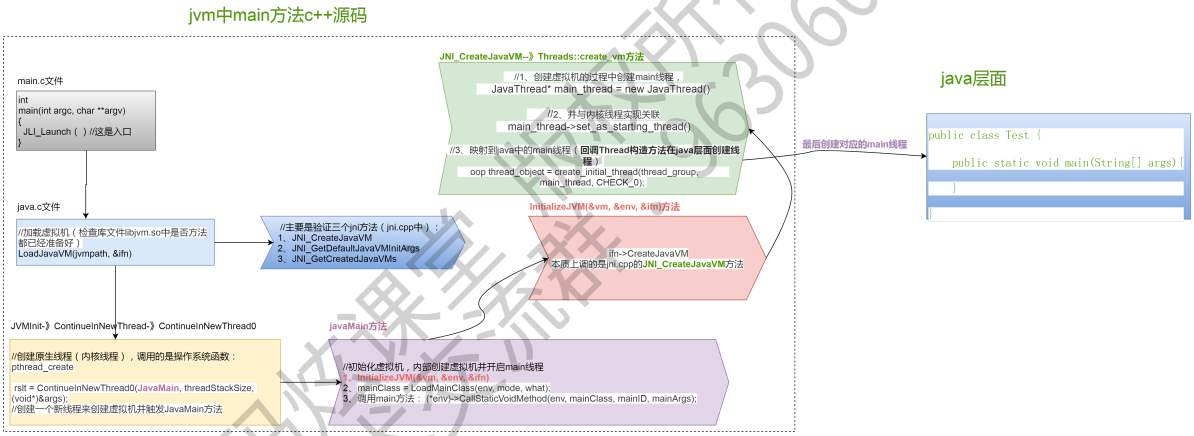
main_thread->set_as_starting_thread()（真正和内核线程关联的是这句代码）

JavaCalls::call_special—》映射java侧的main线程

2）--》mainClass = LoadMainClass(env, mode, what);

3）--》调用main方法：(*env)->CallStaticVoidMethod(env, mainClass, mainID, mainArgs);

流程图：



原图：http://huatu.xuanheng05.cn/lct/#R4a768a9e0da6b8dafa58ad76b8680bd2

结论：java Thread：os thread=1：1

io多路复用：epoll ，poll，nio native

3）线程改造数据迁移案例

面试题：java中实现实现线程有几种方式？

1、继承Thread类

2、实现Runnable接口

3、实现Callable接口（拿回返回值，FutuTask）

4、线程池

方式一：继承Thread类的方式

1. 创建一个继承于Thread类的子类
2. 重写Thread类中的run()：将此线程要执行的操作声明在run()
3. 创建Thread的子类的对象
4. 调用此对象的start():①启动线程 ②调用当前线程的run()方法

方式二：实现Runnable接口的方式

1. 创建一个实现Runnable接口的类
2. 实现Runnable接口中的抽象方法：run():将创建的线程要执行的操作声明在此方法中
3. 创建Runnable接口实现类的对象
4. 将此对象作为参数传递到Thread类的构造器中，创建Thread类的对象
5. 调用Thread类中的start():① 启动线程 ② 调用线程的run() --->调用Runnable接口实现类的run()

以下两种方式是jdk1.5新增的！

方式三：实现Callable接口

说明：

1. 与使用Runnable相比， Callable功能更强大些
2. 实现的call()方法相比run()方法，可以返回值
3. 方法可以抛出异常
4. 支持泛型的返回值
5. 需要借助FutureTask类，比如获取返回结果

- Future接口可以对具体Runnable、Callable任务的执行结果进行取消、查询是否完成、获取结果等。
- FutureTask是Futrue接口的唯一的实现类
- FutureTask 同时实现了Runnable, Future接口。它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值

方式3和方式2的区别就是有些场景需要有返回值，此时方式3派上用场

方式四：使用线程池

说明：

- 提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用。类似生活中的公共交通工具。

好处：

1. 提高响应速度（减少了创建新线程的时间）
2. 降低资源消耗（重复利用线程池中线程，不需要每次都创建）
3. 便于线程管理

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.FutureTask;
import java.util.concurrent.ThreadPoolExecutor;
```

```java
//方式一
class ThreadTest extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName() + ":" + i);
        }
    }
}

// 方式二
class RunnableTest implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName() + ":" + i);
        }
    }
}

// 方式三
class CallableTest implements Callable<Integer> {

    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName() + ":" + i);
            sum += i;
        }
        return sum;
    }

}

// 方式四
class ThreadPool implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName() + ":" + i);
        }
    }

}

public class Test {
    public static void main(String[] args) {
        // 继承Thread
        ThreadTest thread = new ThreadTest();
        thread.setName("方式一");
        thread.start();

        // 实现Runnable
        RunnableTest runnableTest = new RunnableTest();
        Thread thread2 = new Thread(runnableTest, "方式二");
```

```
        thread2.start();

        // 实现Callable<> 有返回值
        CallableTest callableTest = new CallableTest();
        FutureTask<Integer> futureTask = new FutureTask<>(callableTest);
        new Thread(futureTask, "方式三").start();
        // 返回值
        try {
            Integer integer = futureTask.get();
            System.out.println("返回值（sum）：" + integer);
        } catch (Exception e) {
            e.printStackTrace();
        }

        // 线程池
        ExecutorService pool = Executors.newFixedThreadPool(10);

        ThreadPoolExecutor executor = (ThreadPoolExecutor) pool;
        /*
         * 可以做一些操作：
         * corePoolSize：核心池的大小
         * maximumPoolSize：最大线程数
         * keepAliveTime：线程没任务时最多保持多长时间后会终止
         */
        executor.setCorePoolSize(5);

        // 开启线程
        executor.execute(new ThreadPool());
        executor.execute(new ThreadPool());
        executor.execute(new ThreadPool());
        executor.execute(new ThreadPool());

    }

}
```

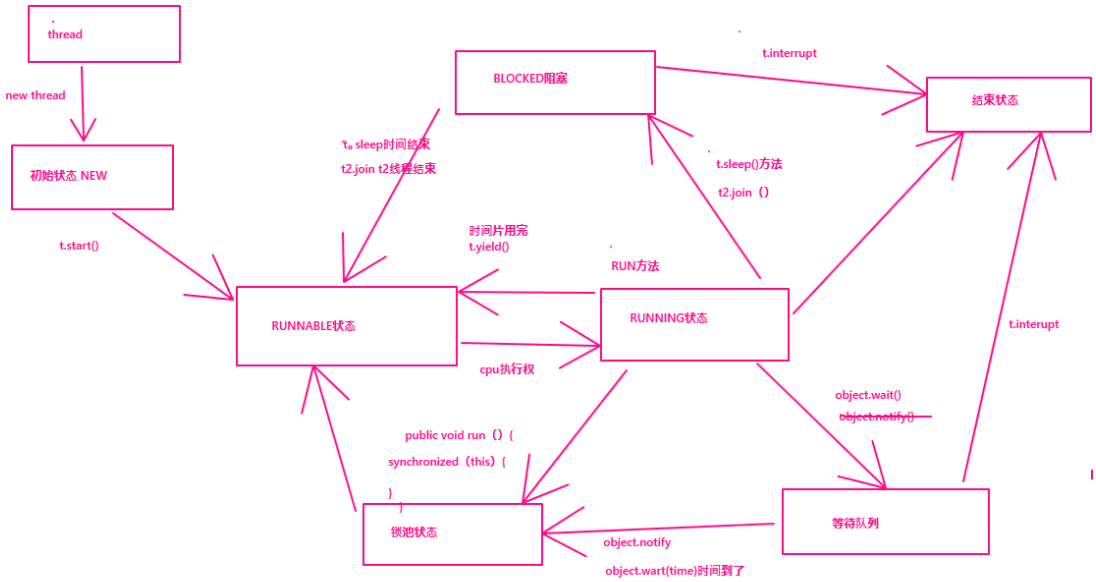

4）消费者线程安全改造分析

1）用synchronized和原子类的区别

2）synchronized放置位置的不同会引起错误的执行结果

详见视频

3）线程之间的状态及转换



5）线程状态转换验证

```
package com.mx.lang.Thread.threadstate;

/**************************************
 *
 * jdk源码&多线程&高并发-【阶段1、深入多线程】
 * 主讲：smart哥
 *
 **************************************/
public class MyLock {
    static Object obj= new Object();
}
```

```
package com.mx.lang.Thread.threadstate;

/**************************************
 *
 * jdk源码&多线程&高并发-【阶段1、深入多线程】
 * 主讲：smart哥
 *
 **************************************/
public class MyThread extends Thread{

    /**
     * NEW  创建线程，未启动
     * RUNNABLE
     * BLOCKED---------------------
     * WAITING---------------------object.wait
     * TIMED_WAITING
     * TERMINATED
     */
    public MyThread() {
        System.out.println("this thread state:"+this.getState());//NEW
```

```
    }

    @Override
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("this thread in run:"+this.getState());//RUNNABLE
    }
}
```

```
package com.mx.lang.Thread.threadstate;

/**************************************
 *
 * jdk源码&多线程&高并发-【阶段1、深入多线程】
 * 主讲：smart哥
 *
 **************************************/
public class MyThreadForBlock1 extends Thread {

    /**
     * NEW  创建线程，未启动
     * RUNNABLE
     * BLOCKED----------------------
     * WAITING----------------------object.wait
     * TIMED_WAITING
     * TERMINATED
     */
//    Object obj = new Object();

    @Override
    public void run() {
        //进入run和抢到锁直接有时间差
        synchronized (MyLock.obj) {
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
package com.mx.lang.Thread.threadstate;

/**************************************
 *
 * jdk源码&多线程&高并发-【阶段1、深入多线程】
 * 主讲：smart哥
 *
```

```
 ****************************************/
public class MyThreadForBlock2 extends Thread {

    /**
     * NEW  创建线程，未启动
     * RUNNABLE
     * BLOCKED----------------------
     * WAITING----------------------object.wait
     * TIMED_WAITING
     * TERMINATED
     */
//    Object obj = new Object();

    @Override
    public void run() {
        synchronized (MyLock.obj) {
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
package com.mx.lang.Thread.threadstate;

/****************************************
 *
 * jdk源码&多线程&高并发-【阶段1、深入多线程】
 * 主讲：smart哥
 *
 ****************************************/
public class MyThreadForWait extends Thread {

    /**
     * NEW  创建线程，未启动
     * RUNNABLE
     * BLOCKED----------------------
     * WAITING----------------------object.wait
     * TIMED_WAITING
     * TERMINATED
     */
    Object obj = new Object();

    @Override
    public void run() {
        synchronized (obj){
            try {
                obj.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
```

```
    }
}
```

```
package com.mx.lang.Thread.threadstate;

/**************************************
 *
 * jdk源码&多线程&高并发-【阶段1、深入多线程】
 * 主讲：smart哥
 *
 **************************************/
public class StateTest {
    public static void main(String[] args) {
//        MyThread myThread = new MyThread();
//        System.out.println("in main the thread state1
is:"+myThread.getState());
//        try {
//            Thread.sleep(1000);
//        } catch (InterruptedException e) {
//            e.printStackTrace();
//        }
//        myThread.start();
//        try {
//            TimeUnit.SECONDS.sleep(1);
//            System.out.println("in main the thread state2
is:"+myThread.getState());
//        } catch (InterruptedException e) {
//            e.printStackTrace();
//        }
//        System.out.println("in main the thread state3
is:"+myThread.getState());

        //测试waitting状态
//        MyThreadForWait thread = new MyThreadForWait();
//        thread.start();
//        System.out.println("in main before sleep the thread state
is:"+thread.getState());
//
//        try {
//            Thread.sleep(1000);
//        } catch (InterruptedException e) {
//            e.printStackTrace();
//        }
//        System.out.println("in main after sleep the thread state
is:"+thread.getState());

        //测试block状态
        MyThreadForBlock1 myThreadForBlock1 = new MyThreadForBlock1();
        MyThreadForBlock2 myThreadForBlock2 = new MyThreadForBlock2();
        myThreadForBlock1.start();
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
        myThreadForBlock2.start();
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("myThreadForBlock1 state
is:"+myThreadForBlock1.getState());
        System.out.println("myThreadForBlock2 state
is:"+myThreadForBlock2.getState());
    }
}
```

6) JVM源码debug分析线程start0方法

面试题：Thread类的start0方法究竟发生了什么？为什么执行start方法之后，系统会自动执行run方法？

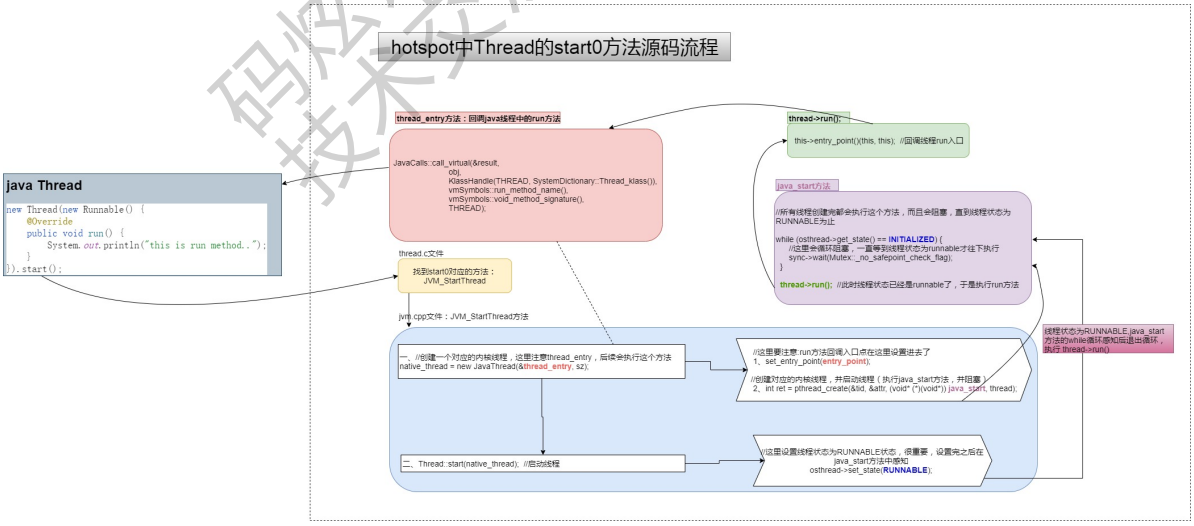
猜想：在jvm层面启动一个原生线程，然后原生线程置为RUNNABLE状态，然后回调java中的run方法

结论：

1、在jvm层面new thread之后（此时状态是initialized状态），然后会一直while死循环判断，判断如果是非initialized状态，才会继续往下执行run方法（等待调用start方法之后，讲状态改为runnable）；

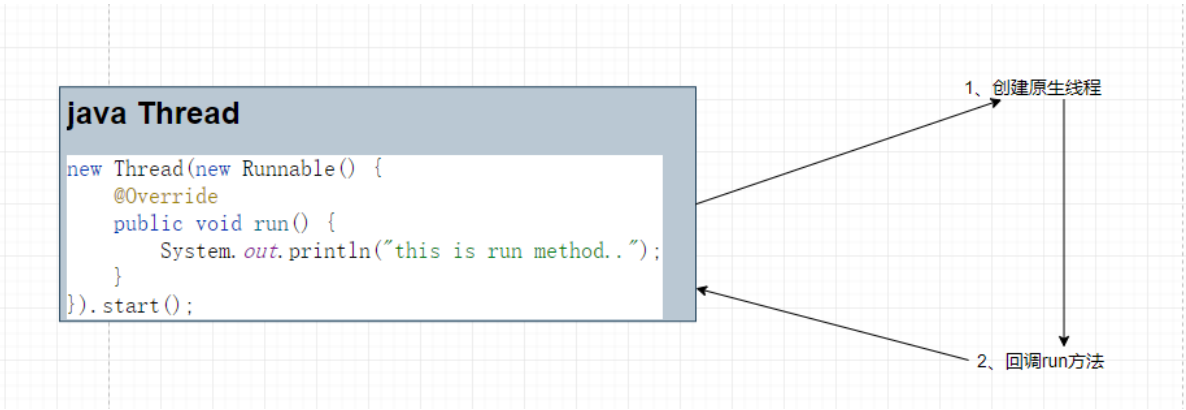
2、执行start方法，修改线程状态为runnable，然后第1步，开始放行，执行run方法（回调java层面的run）

详细分析见视频



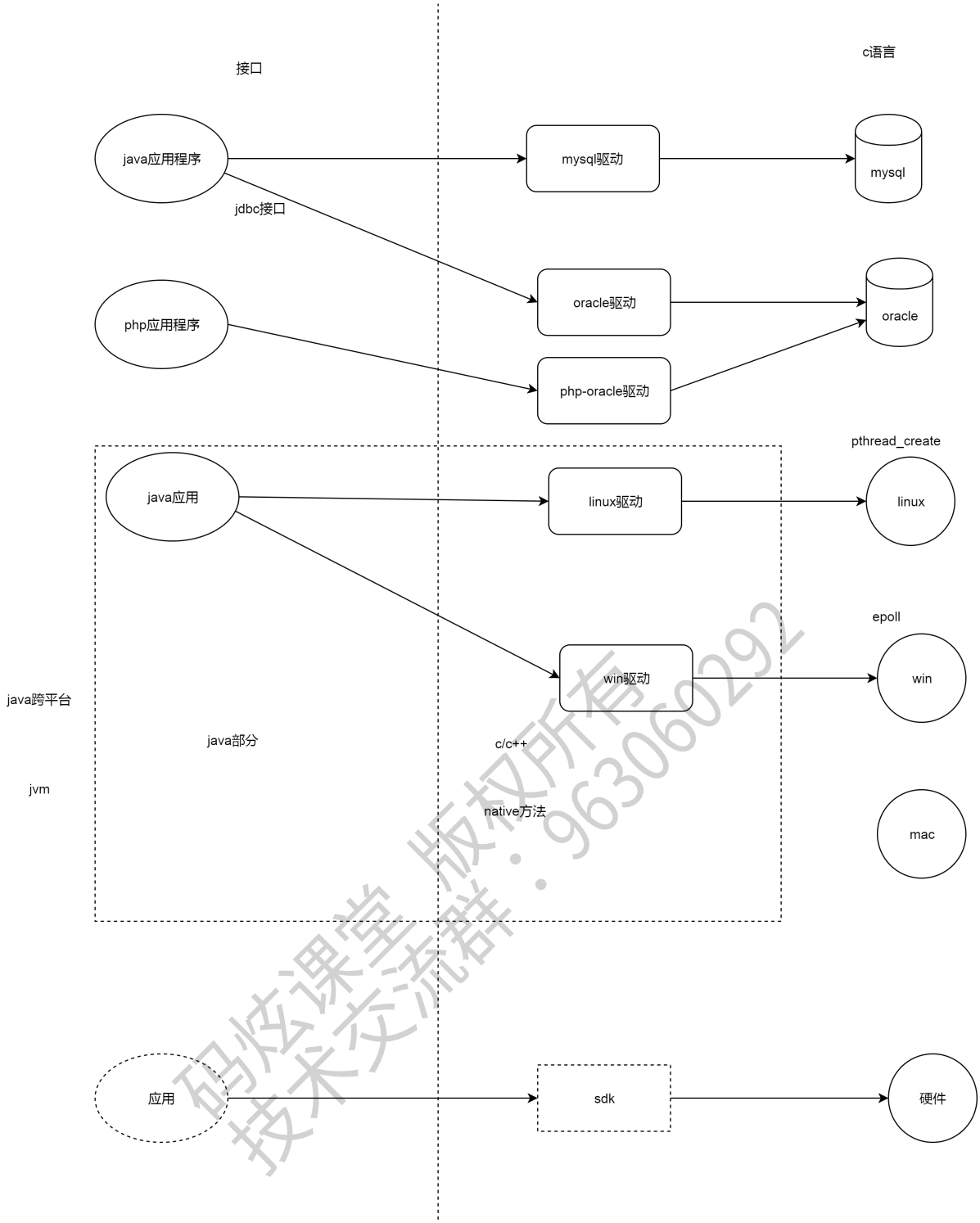
流程图详见：http://huatu.xuanheng05.cn/lct/#R451fb285bcd8ce066b090823fec6df85

7）jvm中手写自定义java 线程对应的原生线程，并实现java侧run方法回调



详细讲解见视频

8）native方法类比总结



具体讲解见视频

3、线程API详解

1）构造方法解析演示

2）线程组

3）如何防止第三方系统在使用线程中搞破坏？

```
Thread(Runnable target, AccessControlContext acc)
```

4）线程和stacksize的关系

栈容量：默认值是多少？？？

栈深度（增大单个栈帧的大小，栈深度变小）

1）试验：默认栈容量下的栈深度和调整局部变量后的栈深度（*JT*/编译器优化了，局部变量不使用的话，不会分配内存）

1）-Xss的影响（分别在main和thread上的区别）

2）-Xss的最大最小值（win上和linux上），为什么是236k，有的博客是228k，源码解析

3）linux上默认栈容量为1M？源码解析

4）比较win和linux上的栈深度

5）线程优先级

1）本质上就是调的操作系统函数：setpriority

setpriority, pthread_create, pthread_join

优先级只是让操作系统和cpu尽量优先分配资源，但并不能用来控制执行顺序。

6）守护线程

1）守护线程（能不能把main线程设置成守护线程？）

2）jvm源码解析守护线程为什么在jvm退出时退出----

wait,notify

3）守护线程场景：Java获取系统内存、CPU、磁盘等信息

4）补充：问题：DestroyJavaVM是守护线程吗？

5）能不能救守护线程一命？

7）hook（钩子）线程

1）为什么要有钩子线程？

2）钩子线程与destroyJavaVM线程的关系？

3）钩子线程在哪个阶段被执行？

4）可以有多个hook线程吗？

5）源码解析

6）使用场景：一个完整的钩子线程的运用案例（演示kill -9）

比如有一个任务，一直在运行，如果说发生异常（main线程，main结束），

1、此时需要上报异常，邮件通知相关人员

2、此时需要释放资源，网络，数据库连接资源等等

方案：加两个钩子

8、线程出现异常的处理

1）try。。catch。。在run方法外能否获取线程异常？

2）如何在外面拿到异常？

3）setDefaultUncaughtExceptionHandler 设置的一定会被调用到么？

4）改变异常处理行为（查看源码）

5）异常处理嵌入点及异常处理的优先级（查看源码分析）

二、线程间通信及并发

方法间通信

```
int method1(){
int i;
.....
return  i;
}
String method2(int a){
}
void method3(){
      int x= method1();
      method2(x);
}
```

问题：线程间通信？？

共享变量 int i;

```
new Thread(()->{
   //业务逻辑
  i;
});
```

```
new Thread(()->{
    //业务逻辑
    i;
});
```

前提：线程之间如何通信及线程间如何同步？

方法之间可以通过传递参数的形式进行通信，那么线程之间怎么传递参数呢？

1、共享内存和消息传递

2、演示各种诡异现象

思考：什么时候线程会重新读取主存共享变量？

1、线程并发安全机制

1）volatile机制

马姓

正确理解volatile的姿势：单cpu架构->cpu多级cache结构 -> **缓存一致性协议（MESI）-> store buffer和invalidate queue引入 ->造成mesi协议不一致了**-> 内存屏障-->volatile-->mesi协议再次一致

cpu芯片，非开源

x86没有invalid queue

只有storebuffer

- **cpu多级cache结构** vs **JMM模型**
  - 图解 cpu多级缓存架构
    高级语言（java）--》字节码（jvm识别）-->汇编指令（硬件cpu）
    mov
    jap
    add 0x12333 i
    RAM (主内存)------》cpu（寄存器）0-1ns
    100-120个cpu周期

    ---

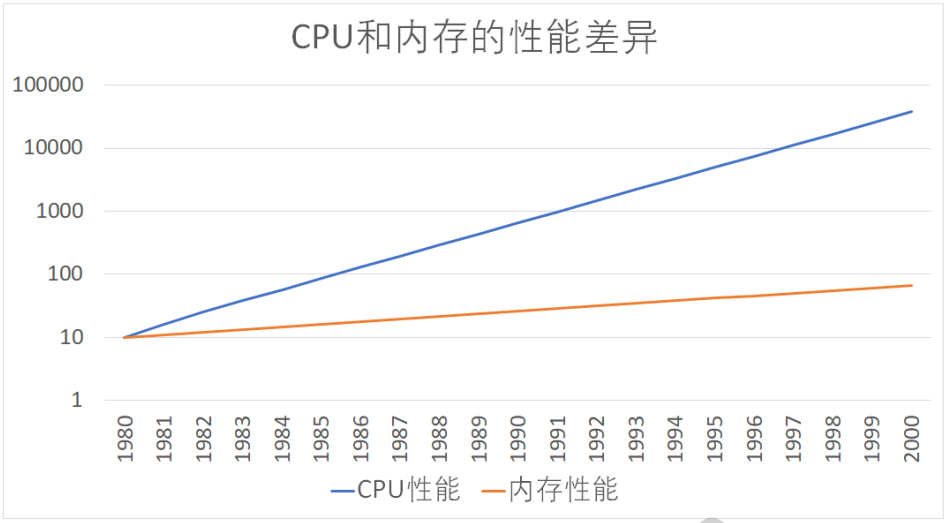
    cpu 24亿/s（寄存器，storebuffer）
    cache（l1，l2，l3）
    RAM 7%,内存访问时间 80-100ns，1s/100ns=1000w
    硬盘 10ms，100/s

    应用，软件领域，来自于硬件
    application

redis

db

- 为什么需要cpu高速缓存？

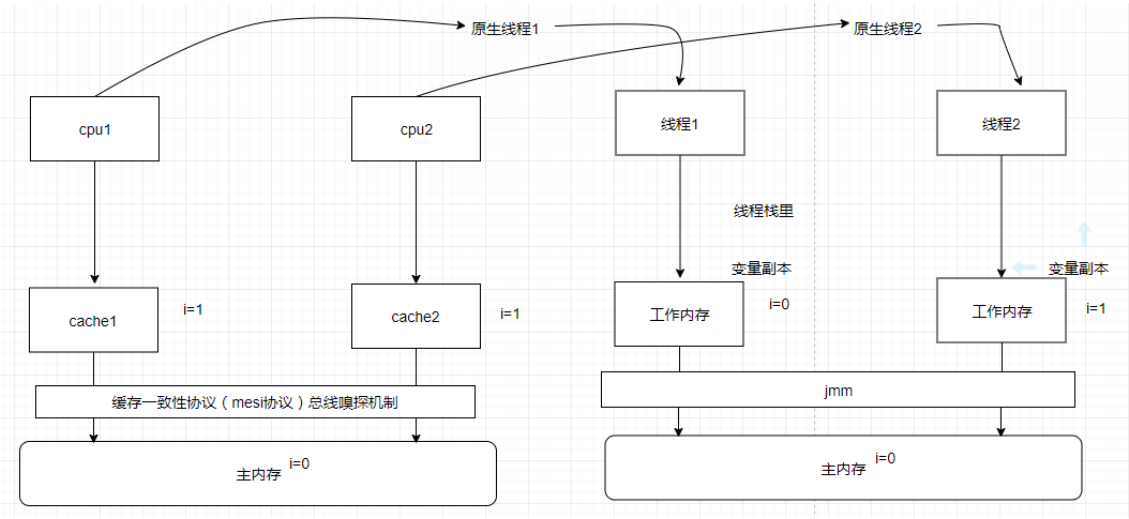


- 为什么需要cpu多级缓存？

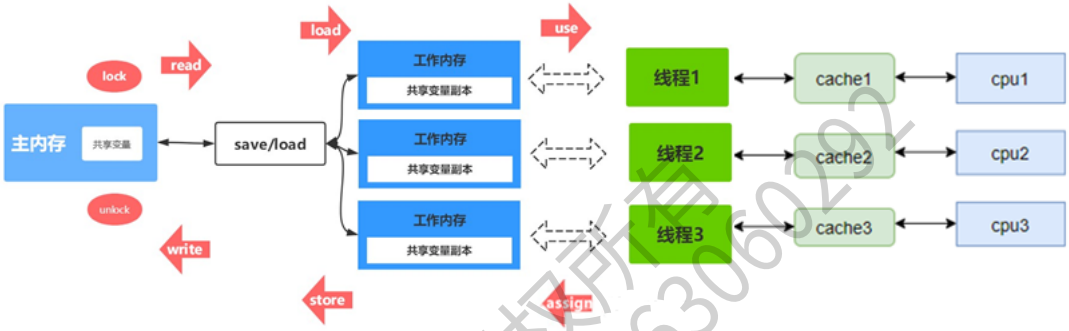
  随着发展，cpu速度和内存速度差距越来越大，所以引入多级缓存

- 图解 jmm模型架构

- Java内存模型与硬件内存架构的关系



- JMM存在的必要性(JMM8种同步操作和cpu的关系)-------重点



关于主内存与工作内存之间的交互协议，即一个变量如何从主内存拷贝到工作内存。如何从工作内存同步到主内存中的实现细节。java内存模型定义了8种操作来完成。这8种操作每一种都是原子操作。8种操作如下：

- lock(锁定)：作用于主内存，它把一个变量标记为一条线程独占状态；
- read(读取)：作用于主内存，它把变量值从主内存传送到线程的工作内存中，以便随后的load动作使用；
- load(载入)：作用于工作内存，它把read操作的值放入工作内存中的变量副本中；
- **use(使用)：作用于工作内存，它把工作内存中的值传递给执行引擎，每当虚拟机遇到一个需要使用这个变量的指令时候，将会执行这个动作；**
- **assign(赋值)：作用于工作内存，它把从执行引擎获取的值赋值给工作内存中的变量，每当虚拟机遇到一个给变量赋值的指令时候，执行该操作；**
- store(存储)：作用于工作内存，它把工作内存中的一个变量传送给主内存中，以备随后的write操作使用；
- write(写入)：作用于主内存，它把store传送值放到主内存中的变量中。
- unlock(解锁)：作用于主内存，它将一个处于锁定状态的变量释放出来，释放后的变量才能够被其他线程锁定；

Java内存模型还规定了执行上述8种基本操作时必须满足如下规则:

（1）不允许read和load、store和write操作之一单独出现（即不允许一个变量从主存读取了但是工作内存不接受，或者从工作内存发起会写了但是主存不接受的情况），以上两个操作必须按顺序执行，但没有保证必须连续执行，也就是说，read与load之间、store与write之间是可插入其他指令的。

（2）不允许一个线程丢弃它的最近的assign操作，即变量在工作内存中改变了之后必须把该变化同步回主内存。

（3）不允许一个线程无原因地（没有发生过任何assign操作）把数据从线程的工作内存同步回主内存中。

（4）一个新的变量只能从主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量，换句话说就是对一个变量实施use和store操作之前，必须先执行过了assign和load操作。

（5）一个变量在同一个时刻只允许一条线程对其执行lock操作，但lock操作可以被同一个条线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁。

（6）如果对一个变量执行lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作初始化变量的值。

（7）如果一个变量实现没有被lock操作锁定，则不允许对它执行unlock操作，也不允许去unlock一个被其他线程锁定的变量。

（8）对一个变量执行unlock操作之前，必须先把此变量同步回主内存（执行store和write操作）。

- **cpu缓存一致性协议（mesi协议详解）**
  - cacheline(cache block)结构与原理
    - cacheline基础

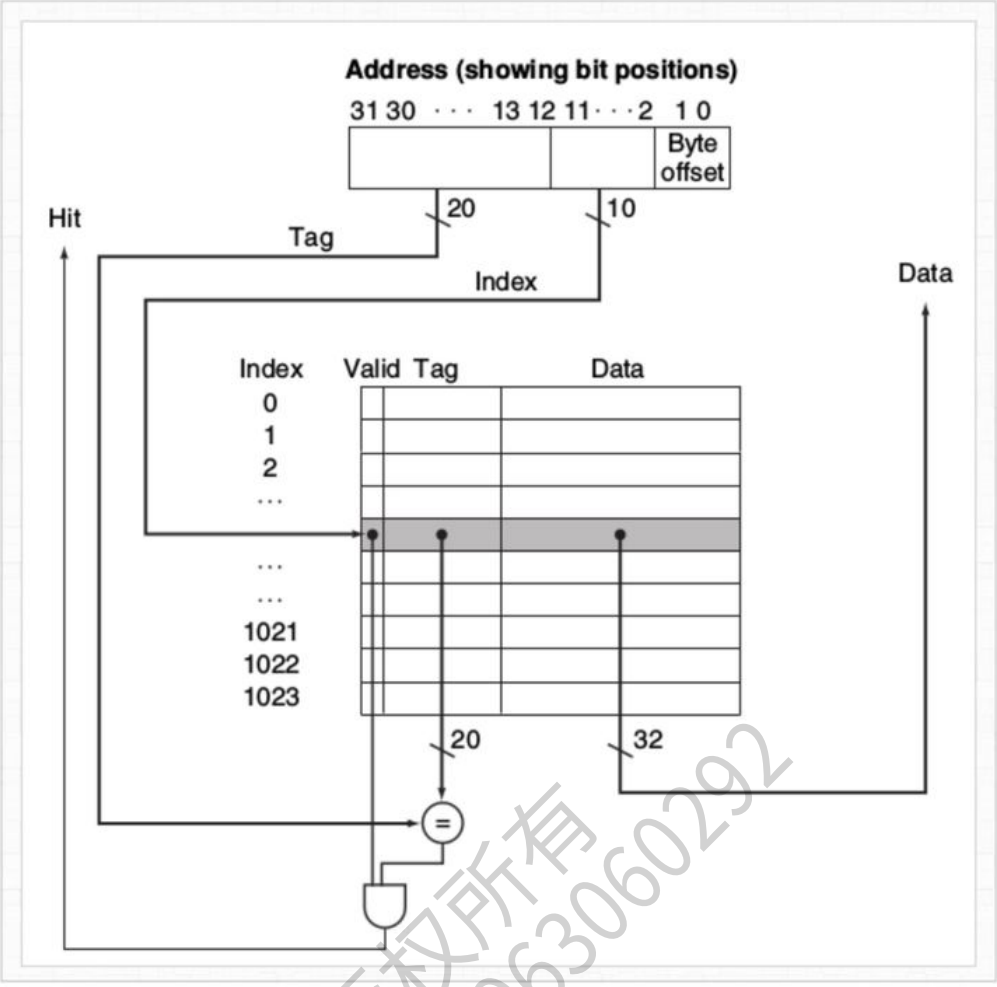
      1）什么是cacheline？

      cache block

      2）图解cacheline结构

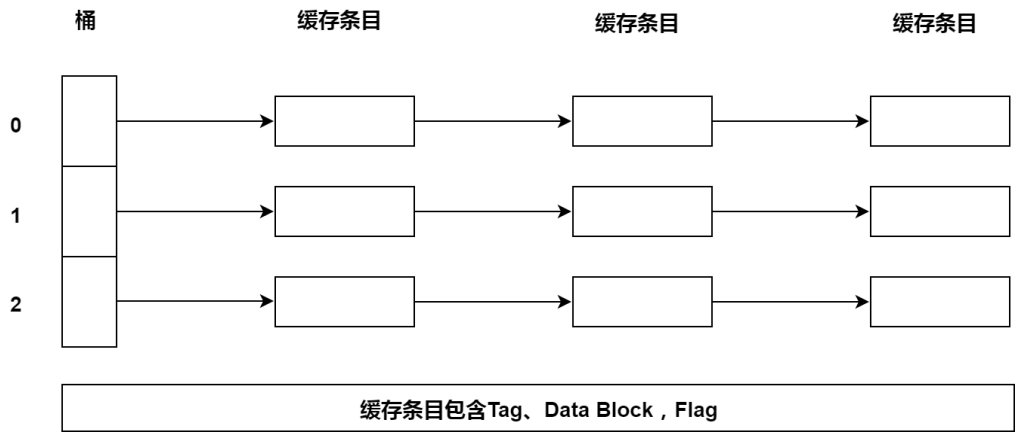
      cache line与内存之间的对应关系

      <1>、Direct Mapped Cache



<2>、Two-way Set Associative Cache

	Way 0	Way 1
0x0	0x12345000	
0x1	0x12345100	
0x2	0x12345200	
0x3	0x12345300	
0x4	0x12345400	
0x5	0x12345500	
0x6	0x12345600	
0x7	0x12345700	
0x8	0x12345800	
0x9	0x12345900	
0xA	0x12345A00	
0xB	0x12345B00	
0xC	0x12345C00	
0xD	0x12345D00	
0xE	0x12345E00	0x43210E00
0xF		



3）cpu与cache， 内存交互的过程

Cache写机制：

缓存命中

write through----------直写，写通

修改cache之后，立马写回主存

write back----------回写

修改完之后，不立即写入主存，而是等到一定的时候，触发一次回写

缓存未命中

write allocate

cache替换策略：

1、LFU(least frequence used 最不经常使用)

2、LRU（least recently used 近期最少使用） 算法复杂，但是命中率高，90%

3、随机替换

4）cacheline的两个局部性约束：时间局部性和空间局部性

- cacheline的伪共享问题及解决方案

如何保证？

1、通过class的层级padding（jdk8之前推荐使用）

2、自动padding

伪共享很容易出现吗？什么情况下出现几率大？

TLAB=thread local allocate buffer

- 实践案例分析

1、ConcurrentHashmap

2、thread

- Java中对Cache line经典设计（Disruptor框架）(放到第4或者5阶段)

更juc包放一起讲

- MESI状态

M:modified

cpu拥有cacheline，并且做了修改，但是修改值还没有刷新到主内存

E: exclusive

cpu拥有cacheline，但是还没有做修改。

M，E 重要特性：在所有cpu的cache中，只有唯一的一个cacheline是M或者E状态

S：shared

所有cpu的cache都对某一个cacheline拥有read，但是不能写

I：无效，失效

当前某个cacheline数据无效，也就相当于里面没有数据

- MESI协议消息

read：就是请求数据，读取一个物理内存地址上的数据，把消息通过总线广播，然后等待回应

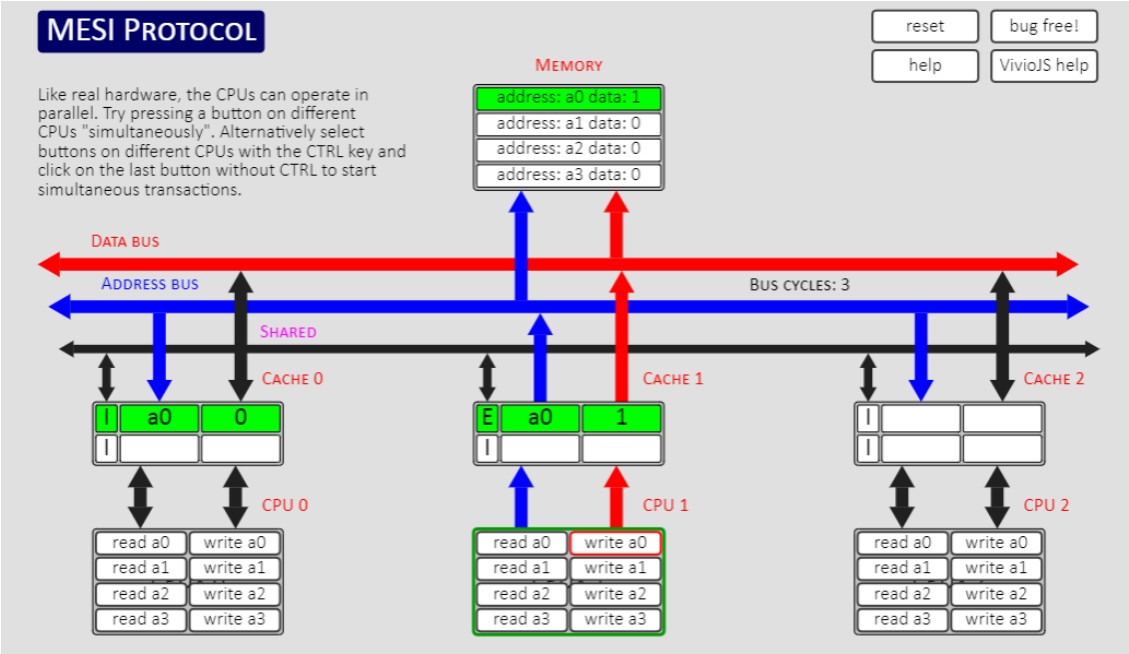
read response：包括read请求的数据，这个数据可能来源于其他cpu，也可能来源于主内存

invalidate：包含一个物理地址，告知其他cpu，把cache中的这个地址所对应的cacheline置为无效

invalidate acknowledge：置无效之后的反馈信息

read invalidate：read+invalidate，这种对应cacheline里没有响应的变量的时候

writeback：回写数据，回写到主内存，m



- MESI状态切换

Table C.1: Cache Coherence Example

			CPU Cache				Memory	
Sequence #	CPU #	Operation	0	1	2	3	0	8
0		Initial State	–/I	–/I	–/I	–/I	V	V
1	0	Load	0/S	–/I	–/I	–/I	V	V
2	3	Load	0/S	–/I	–/I	0/S	V	V
3	0	Invalidation	8/S	–/I	–/I	0/S	V	V
4	2	RMW	8/S	–/I	0/E	–/I	V	V
5	2	Store	8/S	–/I	0/M	–/I	I	V
6	1	Atomic Inc	8/S	0/M	–/I	–/I	I	V
7	1	Writeback	8/S	8/S	–/I	–/I	V	V

- MESI协议案例分析

  mesi协议动态演示

  https://www.scss.tcd.ie/Jeremy.Jones/VivioJS/caches/MESIHelp.htm

  用的MESI的变种

  x86-MESIF协议

  arm-MEOSI协议

- 体验一个美团面试题

  可见性怎么发生？

- **早期mesi协议性能低下，于是引入store buffer**
  - store buffer的引入

    消息队列是一样的

目的：提高效率

可见性的本质：就是队列的异步化

- store forwarding机制

  如果指令之间具有依赖性，所以不会重排

- **Store Buffers和内存屏障机制**

  cpu工程师给了应用工程师解决方案：加内存屏障

  本质：串行化操作

- **手动加入写屏障案例演示**

  见视频讲解

- **store buffer的引入导致不必要的延迟，于是引入invalidate queue**
  - invalidate queue的引入

    为了ack消息立即回复

  - Invalidate Queues带来的问题

    读到脏数据

  - **invalidate queues和内存屏障机制**

    加读屏障

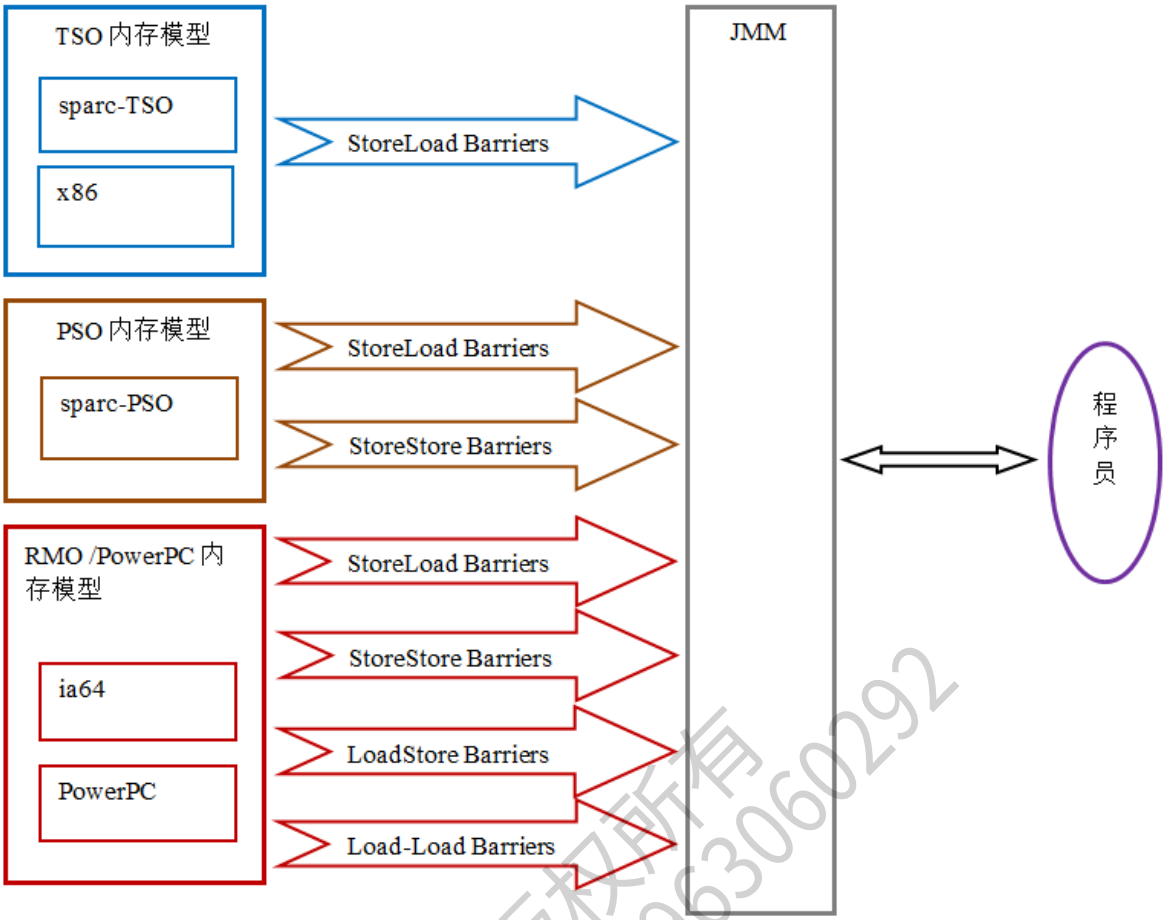
  - **手动加入读屏障案例演示**

    没法演示，为什么？

附：课程讲解画图地址 ：http://huatu.xuanheng05.cn/lct/#R88b3e579acd7885cf5bb040e149112eb

- 内存一致性模型

  因为处理器要遵守as-if-serial语义，处理器不会对存在数据依赖性的两个内存操作做重排序



- 剖析unsafe类中的读，写屏障在hotspot中的源码

storeFence====storestore 在x86上是空操作。第一是写操作，中间是storestore ，第二个也是写操作

loadFence=====loadload 在x86上是空操作。第一是读操作，中间是loadload ，第二个也是读操作

loadstore===== ？？？？numa架构下才会出现，smp架构下一般不会出现这个重排序

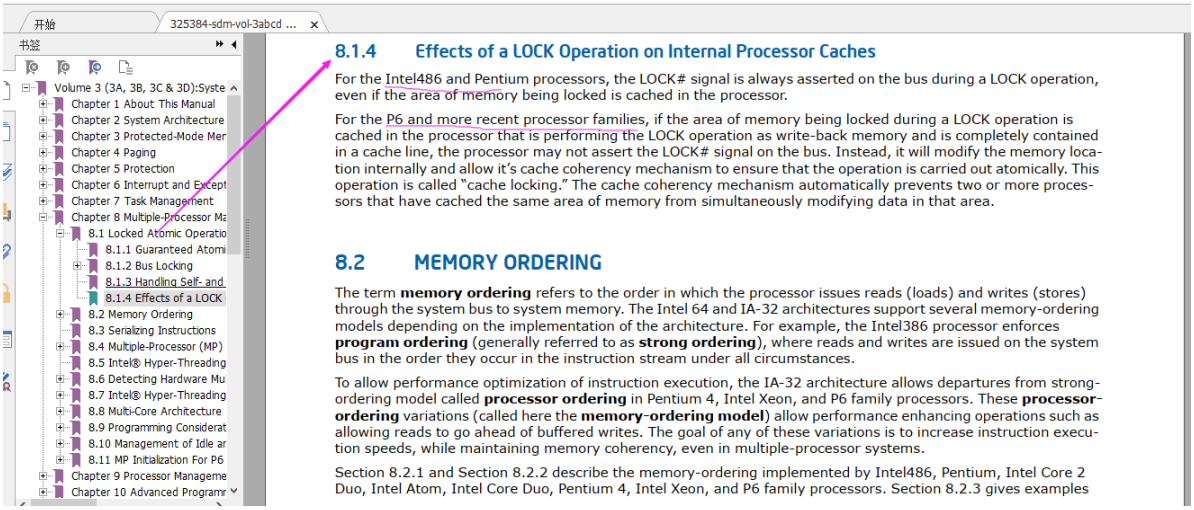
fullFence======storeload 在x86下唯一的一个屏障 ，第一个是写，中间是storeload ，第二个是读屏障

lock addl

- 解析lock指令前缀

https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

intel架构文档第3卷，8.1.4章节



- volatile和lock指令前缀的关系
  - 一个案例：volatile底层究竟发生了什么？

    代码演示

    需要一个工具来查看底层汇编码：HSDIS

    下载地址： http://lafo.ssw.uni-linz.ac.at/hsdis/att/

    放置路径：E:\dev_env\java-1.8.0-openjdk\jre\bin\server

    hotspot是用jit编译，jit分c1编译器和c2编译器

    c1编译器------client模式

    c2编译器------server模式

    加上虚拟机参数：

    -server
    -XX:+UnlockDiagnosticVMOptions
    -XX:+PrintAssembly
    -XX:-Inline

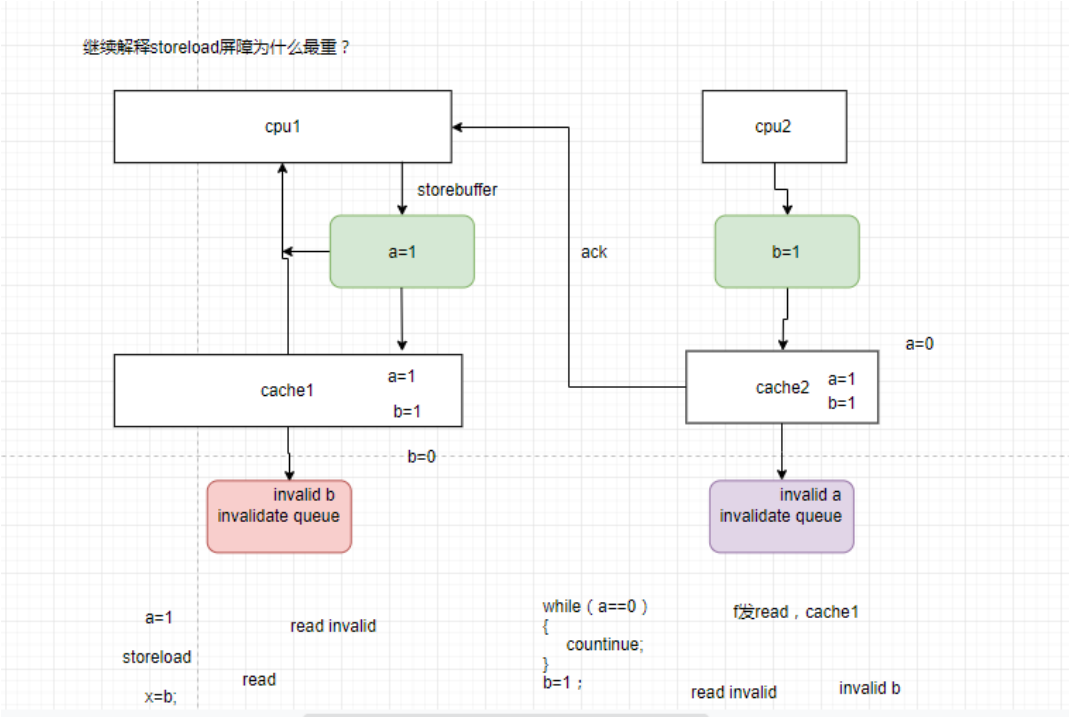
    结论：字段上面如果加上volatile修饰，那么底层汇编码会加上lock前缀指令，从而起到全屏障的作用

  - 再次解释storeload屏障为什么最重？？

    volatile----》底层全屏障（storeload）-----》lock指令-----》

    写读屏障为什么最重？？

    写指令　storeload屏障　读指令



- 深度解析volatile的两大重要特性

  1、可见性

  每次读volatile变量的时候，读到的总是最新的值（即所有cpu，最后一次写入的操作），就是任何一个线程的最新写入。

  2、禁止指令重排

  维护happens-before：

  -对volatile变量的写入不能重排到写入之前的操作之前，从而保证别的线程能够看到最新的写入的值

  -对volatile变量的读操作，不能排到后续的操作之后。

  注意：禁止重排并不是禁止所有的重排，只有volatile写入不能向前，读取不能向后。除了这两种以外，其他重排可以可以允许的。

  volatile int b；

  volatile int a；

  a=1；

  b=1；

  列举：**normal store，normal load，volatile store，volatile load**

  重点关注后面这个指令，volatile store

  normal store，**storestore** ，*volatile store*

  volatile store ，**storestore**，*volatile store*

  normal load，**loadstore**，*volatile store* ----------numa架构

  volatile load，**loadstore**，*volatile store*----------numa架构

第一个指令是volatile写指令， 都可以重排

volatile int a；

int b；

a=1；

x=b；

volatile store，normal load--------可以重排

volatile store，normal store--------可以重排

---

volatile int a；

volatile int b；

int c；

x=a；

y=b；

z=c；

volatile load，**loadload**，volatile load

volatile load，**loadload**，normal load

volatile load，**loadstore**，normal store

---

normal store，volatile load----------可以重排

normal load，volatile load----------可以重排

最特别的一个，x86唯一的一个

volatile store，**storeload**，volatile load

还有4个，都可以重排序

nomal store，normal load

nomal load，normal load

nomal store，normal store

nomal load，normal store

一共16个组合，记忆规则：

针对面试

1）只要看第二个指令，是不是volatile写，如果是volatile写，那么一定需要加xxstore屏障，xx根据第一个指令来，如果是读（不管是normal读还是volatile读）那么xx=load，如果是写（不管是normal写还是volatile写），那么xx=store

2）只要看第一个指令，是不是volatile读，如果是volatile读，那么一定需要加loadxx屏障，xx根据第二个指令来，如果是读（不管是normal读还是volatile读）那么xx=load，如果是写（不管是normal写还是volatile写），那么xx=store

3）还有一个特殊的，volatile写后面接一个volatile读。这就是全屏障，一定会加storeload屏障

4）其余都不需要屏障，可以重排序

- volatile运行时指令重排序

  美团案例

- volatile编译时重排序（java和c++中编译期指令重排序）
  - java中用jitwatch查看JIT编译期指令重排

    需要工具：HSDIS ，JITWatch

    jitwatch：可以同时对比java源码，java字节码，汇编码

    1）jitwatch下载地址 https://github.com/AdoptOpenJDK/jitwatch

    2）下载完解压之后，然后进入core文件夹，执行 mvn clean compile exec:java

  编译完之后 打开 launchUI.bat 文件，如果闪退，则回到上级目录下继续执行 mvn clean compile exec:java

  然后重新打开launchUI.bat 文件。

跑程序的时候需要加上虚拟机参数如下：

```
 -server
-XX:+UnlockDiagnosticVMOptions
-XX:+PrintAssembly
-XX:-Inline
-XX:+LogCompilation
-XX:LogFile=jit.log
```

- c++中的编译期指令重排序演示

  g++ 编译的时候选择o2编译器之后，指令发生重排序

```
int x,y,r;

void method(){
   x=r;
  __asm__ volatile("" ::: "memory");//加入了内存屏障
   y=1;
}
```

- 总结

  1）具体到x86里面，storeload屏障具体是怎么一回事？

  x86里面没有invalidqueue

  2）解释之前的案例

  3）volatile在中间件源码中的运用

  tomcat

  4）早期的mesi协议锁总线，效率低下（强一致性）----》cpu工程师不能忍受，于是加入了storebuffer（好处：提高了访问效率，带来的缺陷是：1、可见性，2、容量问题）相对弱一点----》引入invalid queue(效率再次提升，带来的影响：一致性更加弱了)-----》请软件工程师自己解决（volatile）

  马姓：mesi（storebuffer，invalidqueue）

2）synchronized机制

- 变量的安全性问题探讨
  - 多线程高并发，有共享变量

    1、有**共享变量**，有线程安全性问题（实例变量）

    2、**局部变量**没有线程安全性问题
  - synchronized的 可见性，有序性，原子性

- synchronized字节码解析

  synchronized的底层究竟是什么？

  为什么会有多个monitorexit？

- 对象与锁

  锁的本质就是一个对象，所有线程要争抢这个对象

  问题：锁代码块和锁方法所对应的那把锁是同一个锁吗？？

  证明了this 锁

  证明class锁

- 锁重入机制

  synchronized是可以重入，重入锁

  重入锁一定要注意，在锁嵌套的时候，所有嵌套的方法签名上一定要加上synchronized关键字

  在继承关系下锁重入机制是否可行？？

  答：可行

- synchronized修饰的方法如果发生异常锁怎么办？会释放吗？

2个方法，都是有synchronized修饰的，其中一个发生异常了，jvm会在发生异常之后自动释放锁

- 方法同步 vs 代码块同步

  同步代码的优化方案

- 小总结

  是异步还是同步主要就是看synchronized是不是同一把锁。

  如果是同一把锁，那么就是同步执行

  如果不是同一把锁，那么就是异步执行

- this锁和class锁究竟在什么情况之下使用？

  this锁实际上是当前方法所在的实例----》单例比较多

  class锁实际上是当前方法所在的类----》new N个实例

总结：

1）如果当前class的实例是单例，那么就用this锁

2）如果当前class的实例不确定是否是单例，那么如果方法间需要同步，则只用class锁

前提：多线程执行多个方法，多个方法间需要同步

- 线程的死锁问题

  1）见视频讲解

  2）如何避免死锁？

  - 锁的获取保持有序
  - 设置超时（synchronized没有超时机制，我们自定义lock来设置超时）
  - 做死锁检测（数据结构）

- 锁对象的改变导致异步执行

  见视频讲解

- 锁优化-降低锁粒度

  concurrenthashmap，里面有一个叫做锁分段机制

- 逃逸分析及锁消除

  -server //jit的c2编译器

  -XX:+DoEscapeAnalysis //逃逸分析

  -XX:+EliminateLocks //锁消除

  -XX:+EliminateAllocations //标量替换

  面试题：new创建的对象是否一定分配在堆上面？

  不一定，比如说栈上分配，tlab

new point（int x，int y）；

1）逃逸分析是什么？

堆上面是有gc的，full gc

定义在方法内部的局部变量，

逃逸分析：实际上就是看局部变量有么有逃出方法之外，

int method1（ ）{}

int x；

。。。。

return x；

}

Point method2（ ）{}

Point point=new Point（x，y）；

system.out.print(point.x,point.y)

return point；

}

非栈上分配和栈上分配

非栈上分配

```
// 栈上分配---》锁会自动消除
```

- synchronized源码分析

附：课程讲解画图地址：http://huatu.xuanheng05.cn/lct/#R486b3cb25e10d85e15b67295b5a549f4

jdk1.6之前就是用的monitor（操作系统底层的互斥锁）

串行来访问共享资源

本质：实际上同步互斥访问（多个线程来争取一个**对象**）

new Object()---->对象最终是丢给jvm来管理----》jvm会在对象上加一些管理信息

包装完之后：

1）**对象头（重点）**

2）实例数据

3）填充数据

32位

锁状态	25 bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
GC标记	空				11
重量级锁	指向重量级锁Monitor的指针（依赖Mutex操作系统的互斥）				10
轻量级锁	指向线程栈中锁记录的指针 pointer to Lock Record				00
偏向锁	线程ID	Epoch	对象分代年龄	1	01
无锁	对象的hashCode		对象分代年龄	0	01

64位 Markword

56bit		1bit	4bit	1bit	2bit	锁状态
54bit	2bit		分代年龄	是否偏向	锁标志	
hashcode		无用	分代年龄	0	01	无锁
线程ID	epoch	无用	分代年龄	1	01	偏向锁
指向持有锁线程的 lockRecord 的指针					00	轻量级锁
指向 monitor 的指针					10	重量级锁
空					11	GC标记

- 步骤：先分析jvm源码，把流程整理出来，然后java层面写出来
- 系统上下文切换：用户态和内核态直接的切换，是比较消耗资源的
- 先来实现重量级锁

总结：一旦进入重量级锁，那么线程就会进入队列，一旦进入队列，说明线程已经挂起（进入内核态），　　后续被唤醒，又需要从内核态进入用户态

对象头（锁所在的对象头）---->MarkWord[ptrObjectMonitor]------>ObjectMonitor

- 优化方案??

轻量级锁状态：如果竞争没这么激烈（cas能够拿到锁），是轻微竞争，那么此时就不需要将线程挂起。

LockRecord------》当前获取到锁的线程栈上面的锁记录

轻量级锁加锁过程：

1）首先，在线程栈中创建一个锁记录（LockRecord）

2）拷贝对象头的markword到当前栈帧中的锁记录中

3）cas尝试将markword中的指针指向当前线程栈中的锁记录，所标记也要改成00，表示此时已经是轻量级锁状态

4）如果更新失败，表示此时竞争激烈，1、需要进行锁膨胀操作 2、重入锁

- 膨胀过程（轻量级锁膨胀为重量级锁）：

  见图示讲解

- 能不能再优化了？？

  synchronized，都是只有1个线程在执行（在一定的时间之内没有线程竞争）

- 为什么要有锁的撤销？？

  偏向锁撤销

  轻量级锁撤销

疑问：为什么撤销轻量级锁需要cas来撤销，而且还会撤销失败？

可能存在一种情况如下：

线程t1获取了轻量级锁，markword指向t1所在在栈帧，此时t2也来请求锁，此时拿不到锁，那么就会升 级膨胀为重量级锁，就把markword更新为ObjectMonitor指针。此时t1线程在退出的时候准备将markword还原，那么此时就会失败。t1只能膨胀为重量级锁退出

- 讲了半天您还是没有实现锁啊？那就自定义实现一个真正能跑的锁。

  juc里面有一个aqs的东西（一个队列+state）

  countdownlatch，barrier

- synchronized 无锁 偏向锁 轻量级锁 重量级锁证明

  jol

  12个字节表示对象头，4个字节做对齐

  12*8=96比特位

  96-64=32个比特位

  对象头（markword（64）+klass pointer（32））

  - 证明无锁状态

    hash之前

    00000001 **00000000 00000000 00000000**

    00000000 00000000 00000000 00000000

    01000001 11000001 00000000 11111000------》klass pointer

    hashcode---0x7d e2 6d b8

hash之后

b8 6d e2

00000001 **10111000 01101101 11100010**

7d

01111101 00000000 00000000 00000000

01000001 11000001 00000000 11111000------》klass pointer

- 证明偏向锁状态

关注最后8个比特位

加锁前

00000 001

加锁中

11110 000

> 加锁后（理论上应该已经是偏向）

00000 **001** 001实际上是无锁？？？

> 原因： 虚拟机启动之后4s之后才会开启偏向锁

> 实验：休眠5秒之后

加锁前

00000101

加锁中

00000101

加锁后

00000101

问题：为什么偏向锁要延迟4s？？

因为jvm在启动的时候会需要加载很多资源，在这些对象上加上偏向锁没有意义，减少了大量偏向锁撤销的成本，所以默认就把偏向锁延迟4s。

-XX:+UseBiasedLocking

-XX:BiasedLockingStartupDelay=0

- 证明轻量级锁

把休眠代码去掉即可验证

验证偏向锁升级为重量级锁

- 证明重量级锁（轻量级锁升级为重量级锁的过程）

所有线程跑完之后，重量级锁恢复为无锁时是有延时的。

- 锁是否可以降级？请证明

只能升级不能降级，其实锁是可以降级的

降级条件：

stw的时候时候（安全点的时候），vmthread，轮询所有的ObjectMonitor，将此时没有被使用的重量级锁降级为轻量级锁

- synchronized的非公平锁验证

面试题：synchronized为什么是一把非公平锁

所谓的非公平锁：先来的线程未必先进入同步代码块执行

原理：三大队列

cxq，entrylist，waitset

- 总结

synchronized是基于JVM内置锁实现，通过内部对象Monitor(监视器锁)实现，基于进入与退出Monitor对象实现方法与代码块同步，监视器锁的实现依赖底层操作系统的Mutex lock（互斥锁）实现，它是一个重量级锁性能较低。

当然，JVM内置锁在1.5之后版本做了重大的优化，如锁粗化（Lock Coarsening）、锁消除（Lock Elimination）、轻量级锁（Lightweight Locking）、偏向锁（Biased Locking）、适应性自旋（Adaptive Spinning）等技术来减少锁操作的开销，内置锁的并发性能已经基本与Lock持平。

synchronized关键字被编译成字节码后会被翻译成monitorenter 和 monitorexit 两条指令分别在同步块逻辑代码的起始位置与结束位置。

Monitor 设计缺陷

由于java的线程是映射到操作系统原生线程之上的，所以线程在放入队列里面的时候是需要阻塞的，在WaitSet里面的线程是需要唤醒的，这时阻塞/唤醒就会产生问题，因为阻塞/唤醒是需要调用Linux内核的命令，但是运行的线程是在jvm虚拟机上，这个时候就会存在操作系统用户态和内核态的转换

> 扩展：
>
> Linux操作系统的体系架构分为：内核和用户空间（应用程序的活动空间，例如jvm）

> 内核：本质上可以理解为一种软件，控制计算机的硬件资源（CPU，硬盘，网络等），并提供上层应用程序运行的环境
>
> 用户空间：上层应用程序活动的空间。应用程序的执行必须依托于内核提供的资源，包括CPU资源、存储资源、I/O资源等
>
> 系统调用：为了使上层应用能够访问到这些资源，内核必须为上层应用提供访问的接口
>
> - 用户态 ： 所有进程初始都运行于用户空间，此时即为用户运行状态，简称用户态。例如jvm虚拟机运行在用户态
> - 内核态 ： 用户态的进程通过系统调用执行某些操作时，例如 I/O调用，此时就需要陷入内核中运行，简称内核态。例如线程阻塞/唤醒

因为会有线程的阻塞和唤醒，这个操作是借助操作系统的系统调用来实现的，常见的 Linux 下就是利用 pthread 的 mutex 来实现的。

而涉及到系统调用就会有上下文的切换，即用户态和内核态的切换，开销挺大。

2、线程通信相关方法

1）wait/nofity机制

- 不使用wait/notify的线程通信机制

通常消费者线程在使用while(true)做轮询，直到生产者线程生产了产品，那么消费者才停止轮询，这样就浪费了cpu资源

- 什么是wait/notify机制

  案例演示：

  1）一个生产者线程，一个消费者线程，同时最多只能生产1个产品

  2）多个生产者线程，多个消费者线程，同时可以生产和消费多个产品

  需求：100个生产者，100个消费者，然后同时最多只能生产10个产品

  问题：总线程数减少了，为什么？？

  虚假唤醒

- 问题：**notify方法真的是去唤醒吗？？（unpark）（并没有真正去唤醒，只是去挪动节点）**

  wait和notify并不是线程的方法，而是Object的方法

  问题：为什么wait方法入队阻塞不需要进行cas入队？？

  wait方法底层：

  将当前线程包装成ObjectWait节点加入到一个叫做waitset的环形双向链表中，然后线程执行park挂起

  wait(timeout) timeout时间是干什么用的？？？

  Policy==0 ：放入到entrylist队列的排头位置

  Policy==1 ：放入到entrylist队列的末尾位置

Policy==2 ：判断entrylist是否为空，为空就放入entrylist中，否则放入cxq队列排头位置（默认策略）

Policy==3 ：判断cxq是否为空，如果为空，直接放入头部，否则放入cxq队列末尾位置

```
synchronized(this){
    this.notify();
}//出了这里才去唤醒
```

三大队列：cxq，entrylist，waitset

总结：一个线程不可能同时出现在这三个队列里，同一时刻只能出现在一个队列里

notifyAll其实就是循环notify（本质就是把waitset中的节点一个一个拿出来丢到cxq或者entrylist中）

- QMode和policy抉择综合案例

  代码见演示

  过程：

  a启动b之后wait，b启动c之后，b休眠200ms（此时c已经抵达锁的入口），然后b唤醒a

  此时a，c一起抢锁

  结果：a永远在c前面先抢到锁，为什么？？？

  思考问题：

  线程A在wait的时候做了什么？

  1）包装a线程成ObjectWait对象

  2）ObjectWait对象进入waitset队列（waitset本质上是个环形双向链表）

  问题：wait为什么不用cas抢着进队列？？

  3）当前线程挂起

  线程c启动之后，由于此时线程b持有锁，那些线程c在干什么？

  c此时在竞争锁，如果竞争不到，进入cxq阻塞

  线程b在notify的时候做了什么？

  根据policy的值做出抉择（操作的是线程a）

  Policy==0 ：放入到entrylist队列的排头位置

  Policy==1 ：放入到entrylist队列的末尾位置

  Policy==2 ：判断entrylist是否为空，为空就放入entrylist中，否则放入cxq队列排头位置（默认策略）

  Policy==3 ：判断cxq是否为空，如果为空，直接放入头部，否则放入cxq队列末尾位置

直接判断，由于默认policy=2，所以此时线程**a被放置到entrylist**

c在哪里？**c在cxq的头部**

QMode

线程b释放锁的时候会根据QMode抉择究竟谁先被唤醒？

根据QMode策略唤醒：

QMode=2，取cxq头部节点直接唤醒

QMode=3，如果cxq非空，把cxq队列放置到entrylist的尾部（顺序跟cxq一致）

QMode=4，如果cxq非空，把cxq队列放置到entrylist的头部（顺序跟cxq相反）

QMode=0，啥都不做，继续往下走（QMode默认是0）默认是0

否则Policy为其他值时才会去直接唤醒（所以notify执行完本质上并没有去唤醒任何线程，只是去挪动节点）

结论：Qmode=0的判断逻辑就是先判断entrylist是否为空，如果不为空，则取出第一个唤醒，如果为空再从cxq里面获取第一个唤醒。**所以最终a先被唤醒，则a先抢到锁，然后a执行完之后释放锁，然后c再去抢**

切记：执行notify后并不是立即就唤醒（Policy不是0，1，2，3时才唤醒），而是要等到有线程退出了同步代码块才触发唤醒动作（这也解释了为什么notify必须放在synchronized同步代码块中）

waitset，cxq，entrylist三大队列，非常重要！！！

- WaitSet、EntryList、cxq三大队列揭秘

  疑难问题：

  1）wait后被唤醒重新抢锁是重新从无锁-》偏向锁-》轻量级锁-》重量级锁 这个流程开始，还是直接抢重量级锁？？

  2）cxq移动元素到entrylist和waitset移动元素到entrylist有什么不同点？

  - cxq是批量移动
  - waitset是一个一个移动

2）join方法

首先，全网博文解释join的，很多都是不准确的，即使有稍微深入一点的解释其实也是只有一知半解

研究join的源码

join方法上是有synchronized锁的，join的本质实际上就是wait

- 问题：究竟是谁唤醒调用wait方法的线程？？

  猜测：jvm的c++层面来唤醒

c++里面的 析构函数,跟构造函数相反

void m1（。。。）{

v1

```
构造函数（开辟空间）
```

析构函数（v1的资源释放）

}

void ~m1（ ）{

}

- c++演示析构函数用法

见视频

3）sleep方法

面试题：sleep和wait的区别

1）sleep是thread的方法，但是wait是Object的方法

2）sleep不释放锁，wait会释放锁

3）sleep不需要在同步代码块中执行，但是wait需要在同步代码块中执行

jvm层面的源码解析

4）interrupt机制

notify

```
1、检验是不是字面意思的中断，发现并没有被中断，那么究竟干了什么事呢？
```

源码

interrupter实际上干了2件事：

1）设置中断标记位（其实就是设置一个true）

2）唤醒线程

```
sleep中断和wait中断的差别
唤醒机制是否相同？？
不相同

源码分析如下：
1）sleep是for循环中不断检测中断，检测到中断则抛出中断异常
2）wait是被唤醒后判断是否是notify唤醒，如果不是notify唤醒的，那么检测中断标记，如果中断标记为true，说明是中断唤醒的，于是抛出异常，然后java中可以捕获这个中断异常
```

5）ThreadLocal原理

- threadlocal使用场景

  1）共享变量

  2）在某些方法里计算的中间结果需要共享给其他方法

  ```
  String m1（）{
      string ss=m2（）；
      。。。。
     。。。。
    。。。。
      return ss；
  }
  void m3（）{
      String s3=m1（）；
  }
  ```

- threadlocal源码解析

  见图示讲解：

  http://huatu.xuanheng05.cn/lct/#Rcf57fd37da74ee236d28b760727716cc

- threadlocal内存泄漏分析

  key值为什么是弱引用？？

目的是弱引用可以被垃圾回收

如何防止内存泄漏

```
防止内存泄漏，此时一定要在使用完之后remove掉
```


三、线程池

简述：

java中的线程跟原生线程是一一对应的

pthread_create(创建线程)，切换到内核态去创建函数

pthread_exit（切换内核态）

实现创建多个线程，放到池子里待用，如果有业务请求过来了，那么此时从池子里获取一个线程，执行业务。执行完，不销毁，直接还回到池子里去（节省了创建和销毁线程这两个步骤，这两个步骤非常消耗资源）

1）为什么要用线程池？

- 降低了资源消耗（避免了用户态和内核态之间的频繁切换，从而节省资源）
- 提高了响应速度，提高了吞吐量（任务抵达的时候可以立即执行，不需要去创建线程了，业务执行完也不需要去销毁线程）
- 提高了线程的可管理性：可以使用线程池对线程做统一的管理和分配以及监控

2）线程池架构体系及设计思路

- 线程池类比见图解

  http://huatu.xuanheng05.cn/lct/#R38d5524acb7d5b8d168552eac227acc1

  线程池和工厂的类比：

  工厂---线程池

  订单---任务

  正式工人---核心线程

  临时工---普通线程

  总工人数---最大线程数

  仓库---队列

  调度员--getTask（ ）

- 流程图

见视频讲解

3）线程池细节详解

- 构造方法
  - 4个构造方法

    见视频讲解

- 任务队列
  - 线程池推荐3个队列
    - SynchronousQueue（同步移交）

      次队列里面没有容器，一个生产者线程，当他生产产品的时候（put），如果当前没有消费者线程想要消费，此时生产者线程必须阻塞，等待一个消费者线程调用（take），take操作会唤醒生产者线程，同时消费者线程会消费产品，所以这叫做一次配对

      案例：

      原理：

      内部是使用的cas来实现线程的安全访问，不像LinkedBlockingQueue，ArrayBlockingQueue使用aqs，并且有公平和非公平之分（队列和栈）

    - LinkedBlockingQueue

      无界队列（严格意义上讲其实不是无界队列，只是界限太大，Integer.MAX_VALUE）,基于链表结构。如果使用这个无界队列的话，当核心线程都繁忙时，后续的任务就 会加入这个无限队列，此时线程池中的线程数不会超过总线程数。那么这种队列的好处就是提高系统吞吐量，但是代价就是牺牲了内存空间，甚至会引发OOM异常。常见的做法是指定队列容量

        - ArrayBlockingQueue

          有界队列，基于数组实现，在线程池初始化的时候指定了队列容量，缺点就是无法再调整，好处就是可以防止资源耗尽。

- 拒绝策略

  4种拒绝策略（中止、丢弃、抛弃最旧的、调用者运行）

  - AbortPolicy----------中止抛出异常
  - CallerRunsPolicy---调用者运行
  - DiscardPolicy--------直接拒绝，不抛异常
  - DiscardOldestPolicy------丢弃最旧的任务

  案例演示：

  见视频

  - 自定义一个拒绝策略

    只要实现RejectedExecutionHandler接口就ok了

  - 比较下常见的中间件的线程池的拒绝策略

    - dubbo的拒绝策略

      实际上也是继承AbortPolicy

- 线程池状态，线程池初始化、容量调整、关闭

  - 5种状态

    - RUNNING

      当创建线程池后，初始时，线程池就处于RUNNING状态

    - SHUTDOWN

      如果你调用了shutdown方法，线程池处于就SHUTDOWN状态，此时线程池不能够接受新任务，他会等待所有任务执行完毕（包括队列里面的任务）

    - STOP

      如果调用了shutdownNow方法，那么此时线程池处于STOP状态，此时线程池不会接受新任务，并且会中断正在执行的任务，并且会清空队列，并返回剩余的任务

    - TIDYING

      SHUTDOWN或者STOP状态下的线程池全部处理完毕

    - TERMINATED

      在TIDYING状态的时候执行ternimate方法，此时进入TERMINATED状态

  - 线程池初始化

    - 在线程池启动的时候就提前启动一个线程 ：prestartCoreThread()
    - 在线程池启动的时候就提前启动核心线程 ：prestartAllCoreThreads()

  - 容量调整

    - 包括线程容量和队列容量

    问题：如何动态的设置任务队列？？

把线程池做成组件（线程池1（192,168,0,101），线程池（192.168.0.102）.。。。线程池N）

- 其他框架中的线程池容量是如何调整的？

  tomcat里面容量是如何调整的？

  ```
  这里会先把核心线程池扩容到最大线程数，然后再往队列里面塞任务
  而我们这边jdk的策略是达到核心线程数先入队列，队列满了之后才会扩容到最大线程数
  ```

- 线程关闭
  - 面试题：shutdown和shutdownNow的区别

    shutdown表示不会立即终止线程池，而是等到所有的任务队列中的任务都执行完了才终止，但是也不再接受新任务。

    shutdownNow表示此时会立即发起中断，并且会清空任务队列，而且还会返回尚未执行的任务。这里也再接受新任务。

4）使用线程池

2种方式

- 正常的使用方式

  注意点：队列一定要加上容量

- 阿里开发手册不允许

  主要是Excutors框架提供的线程池

  主要原因：最大线程和最大队列任务数没有设置（Integer.MAX_VALUE）,易引发OOM异常。

  见视频讲解

5）手写线程池

- 线程池的工作原理

  1、定义一个队列，用来存放提交的任务

  2、线程也需要一个队列

  3、内部就是线程操作任务

  4、外部线程池提交任务

  5、拒绝策略

  6、关闭线程池（实现shutdown：1、不接收新任务，2、必须把任务给完成，任务出队列，线程池出队列）

  shutdownNow就不实现了。大家自行研究实现

  7、线程池的扩容和缩容（互联网大厂会有自己的线程池服务）

  常识：更少的线程可以处理更多的任务，所以这里设置线程的三个参数：min，core（active），max

  1）、min<任务数<active（core），或者任务数<min。线程数只要min就可以了----------默认

  2）、active<任务数<max,当前线程数<active 。则线程数就要扩展到active

  3）、max<任务数，那此时线程数要扩展到max

  缩容：

max，缩容会缩到active（core）

6）线程池源码解析

向线程池提交任务是用ThreadPoolExecutor的execute()方法，但在其内部，线程任务的处理其实是相当复杂的，涉及到ThreadPoolExecutor、Worker、Thread三个类的5-6个方法：

ThreadPoolExecutor类

- execute()/submit()

  举例：一个学校有1-6年级，然后一个年级有2个班，一个班上有40个人。

  需求：给每个同学分配学号

  年级　　人数=80　　64 32 16 8 4 2 1

  001------1　　000000-111111　　一年级学号： 001 000000 | 001 000001。。。。。。001 101000

  010------2　　000000-111111　　二年级学号： 010 000000 | 010 000001。。。。。。010 101000

  011------3　　000000-111111　　三年级学号： 011 000000 | 011 000001。。。。。。011 101000

  100------4　　000000-111111　　四年级学号： 100 000000 | 100 000001。。。。。。100 101000

  101------5　　000000-111111　　五年级学号： 101 000000 | 101 000001。。。。。。101 101000

  110------6　　000000-111111　　六年级学号： 110 000000 | 110 000001。。。。。。110 101000

- addWorker()

  主要就是创建线程和任务，然后执行

  线程和任务捆绑在一起（Worker来表示线程和任务的结合体）

  做两件事：

  1）线程数+1

  2）worker加进hashset，并启动线程

Worker类

- runWorker()

- getTask()

  从队列里获取一个任务执行

- processWorkerExit()

  从hashset中移除一个线程，然后又增加一个线程

  结论：调用shutdown，并不会减少线程，直到队列为空

其他方法

- shutdown
- shutdownNow

7）线程池扩展点

- 三个扩展点：
  - beforeExecute（ ）
  - afterExecute（ ）
  - terminated（ ）

需求： 在多个任务执行的时候，可以暂停线程池（比如暂停1秒），暂停之后必须可以支持恢复线程池；在每个任务执行之后需要获取执行结果；在最后所有任务执行完成之后，必须通知管理员。

- execute和submit方法的区别

放到第三阶段juc讲解

8）面试题：如何合理地估算线程池大小？

1000个并发请求，10台机器，每台机器4核，请设计线程池（核心线程数，最大线程数，任务队列容量）

- 首选得看当前系统是cpu密集型还是io密集型

cpu密集型：尽量减少上下文切换（书本上的答案线程数就是**N+1**=4+1=5），队列容量100个

IO密集型：核心线程=**2N+1**=2×4+1=9 ，9个最大线程。任务队列100个

针对面试你可以这样讲

实际情况：就是说你设置完核心线程，最大线程，任务队列之后，一样是需要进行系统压测。然后根据系统流量状况做出调整，那么调整的前提是必须对线程池进行监控。

- 总结：

1）cpu密集型

2）IO密集型

3）通过压测得到一个合理的值或者范围

4）支持线程池的动态配置

重点是动态配置：核心线程数，最大线程数，**队列容量**

问题：如何来修改队列容量？？？

支持最大任务是多少？

cpu 最大任务量 qps=100+5

io 最大任务量 qps=100+9

常见中间件比如tomcat里面最大任务：

默认bio通道：最大线程数=200+100=300(最大qps)tps

- 现有的解决方案的痛点。

美团调研后发现业内常见的3种解决方案:

方案	问题
$N_{cpu} = number\ of\ CPUs$ $U_{cpu} = target\ CPU\ utilization, 0 \le U_{cpu} \le 1$ $\frac{W}{C} = ratio\ of\ wait\ time\ to\ compute\ time$ *The optimal pool size for keeping the processors at the desired utilization is :* $N_{threads} = N_{cpu} * U_{cpu} * (1 + \frac{W}{C})$	出自《Java并发编程实践》 该方案偏理论化。首先，线程计算的时间和等待的时间要如何确定呢？这个在实际开发中很难得到确切的值。另外计算出来的线程个数逼近线程实体的个数，Java线程池可以利用线程切换的方式最大程度利用CPU核数，这样计算出来的结果是非常偏离业务场景的。
$coreSize = 2 * N_{cpu}$ $maxSize = 25 * N_{cpu}$	没有考虑应用中往往使用多个线程池的情况，统一的配置明显不符合多样的业务场景。
$coreSize = tps * time$ $maxSize = tps * time * (1.7 - 2)$	这种计算方式，考虑到了业务场景，但是该模型是在假定流量平均分布得出的。业务场景的流量往往是随机的，这样不符合真实情况。

第一种设置方案是《Java并发编程实战》一书中给出的计算方式是这样的：

给定下列定义：

$$N_{cpu}=number\ of\ CPUs$$

$$U_{cpu}=target\ CPU\ utilization，\ 0 \leqslant U_{cpu} \leqslant 1$$

$$\frac{W}{C}=ratio\ of\ wait\ time\ to\ compute\ time$$

要使处理器达到期望的使用率，线程池的最优大小等于：

$$N_{threads}=N_{cpu}*U_{cpu}*\left(1+\frac{W}{C}\right)$$

可以通过 Runtime 来获得 CPU 的数目：

```
int N_CPUS = Runtime.getRuntime().availableProcessors();
```


和实际业务场景有所偏离的。

第二种设置方案是 2×cpus，当作io密集型来使用了，但是项目中一般来说不止一个自定义线程池，比如有数据上报的线程池，还有专门处理查询请求的线程池，所以如果这样做一个简单的线程隔离，显然是不合理的。

第三种设置方案是理想状态，各家系统在不同的时间段，流量是不均匀的，所以这样简单粗暴的计算也是不合理的

- 最优解决方案是什么？

  线程池参数动态化

重点是动态配置：核心线程数，最大线程数，**队列容量**

1）动态配置的理论基础

核心线程，最大线程，队列都是可以动态修改且即时生效

2）动态调整过程中需要注意的地方

经验：核心线程调整的时候连同最大线程数一起调整，尤其是新的核心线程>旧的最大线程的时候，一定要注意，两个参数一起调整，否则只调整核心线程数会无效

无效原因见视频

最重要的队列修改：把LinkedBlockQueue源码复制一份，然后把容量的final修饰去掉，增加get set方法

3）最容易出的面试题

- 线程池被创建之后里面有没有线程？如果没有的话，有么有方法对线程池预热
- 核心线程数会被回收吗？