



JAVA 面试指南

【第 III 版】

摘要

[与大家相伴已经是第三个年头了，很高兴和大家一起学习，本版本主要分享互联网常见面试题，为大家在面试时提供一些助力]

清风_gary_明月

[关键字 源码 原理 分布式 J.U.C]

说明：本文档主要针对面试复习使用，以便快速复习知识点，不是技术文档也非技术博客，不会把每个知识点都很详细的介绍，这里主要介绍一些面试的核心点；所以大家可以把它作为面试时复习工具，若日常学习时，可以把它作为提高的目录，更深入的研究学习才能最终提高。下面集中回答一下，群中同学问的较多的问题？

1、面试指南适合哪些人？

其实这个问题不难回答，如果你瞅一眼目录都会了，就不需要再看了。

2、指南为什么会收费？

Java 技术本身开源的，但为了写好这些东西是要花费很多精力收集整理的（有时候一个标题都要一两天），喝杯咖啡总是要的吧，貌似最近都买起最便宜的咖啡，下次再问我，我就要点[卡布奇诺] 😞 了。

3、面试指南还会不会更新？

只是可能会停一段时间，没有说完全不更新了，毕竟笔者也是在职工作的，有时候比较忙，大家多谅解哈！大家可以群中多讨论，以后的会慢慢更新；希望大家多多提一些意见，谢谢！

4、面试指南还有哪些知识未涉及？

现在 **CoreJava** 和 **Web** 主流框架这块都差不多 OK 了，还差的是数据结构、数据算法、**Java** 服务端并框架这块需要补充；由于笔者精力有限，有好多目录都整理好了，都没有时间去完成，只能以后慢慢更新。

5、赠送大家一个问题？

本版本为第三个版本，每个版本可能更新东西不是太多，但都有特色，所以大家结合前两个版本和一个数据版本一起看。

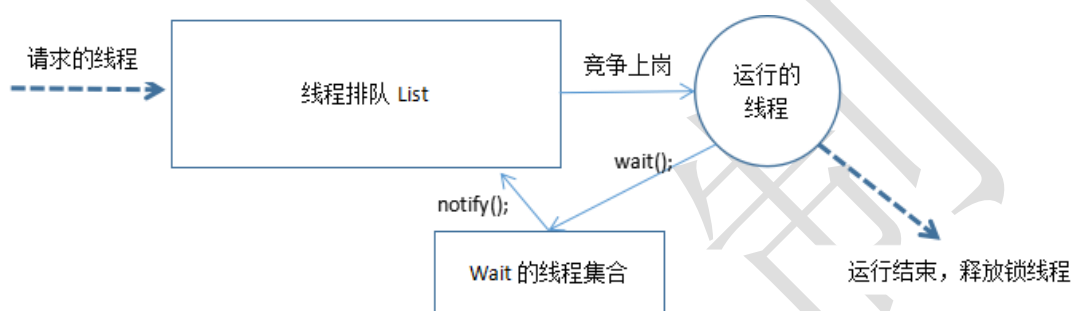
目录

第一部分 CoreJava 核心.....	3
1.1 Synchronized 实现原理	3
1.2 Synchronized 对象锁和类锁的区别	3
1.3 foreach 实现原理.....	4
1.4 Lock 锁的实现.....	6
1.5 Volatile 的内存可见性.....	6
1.6 常用的安全集合类	7
1.7 Switch 能否用 string 做参数	8
1.8 CountdownLatch 实现原理	9
1.9 ThreadLocal 可能造成内存溢出.....	9
1.10 线程池的实现原理	10
1.11 类的加载机制	11
1.12 Redis 与 Memcached 区别	12
第二部分 VIEW 原题	12
2.1 Oracle 删除无序数据	12
2.2 智力题分裂的小球.....	12
2.3 JDK 有哪些线程池工具类	13
2.4 写一个死锁 DEMO	13

第一部分 CoreJava 核心

1.1 Synchronized 实现原理

解析：在第一个版本中已经提到过这个关键字，`synchronized` 在互联网面试中问到概率很高，这里在第一个版本基础上再深入一下分析，对于已经说明过的，比如 `synchronized` 是 JVM 底层实现，在抛出异常时会自动释放锁，而 `Lock` 是需要手动释放锁，否则锁一直占有，诸如此类。不再多作说明（参考版本一和版本二）。



JDK1.5 之前的版本，`Synchronized` 一直是重量级锁，它的底层实现（可以参考上图），当多个线程执行到 `Synchronized` 修饰代码时，其中一个线程获取锁，其它线程在 JVM 维护的一个虚拟列表中等待，当锁被释放时会唤醒等待中的线程，这些等待中的线程是一种竞争上岗，非公平锁的实现。JDK1.6 对 `Synchronized` 进行了优化，引入了轻量级锁，偏向锁，自旋锁的实现。

常见提问：一般线程安全如何实现？`synchronized` 与 `lock` 有什么区别？你对 `synchronized` 底层实现有何了解。

参考答案：

`synchronized` 由于是底层 JVM 实现的互斥，因此效率会高一些，JVM 底层会通过监控器把多个线程维护在一个虚拟队列中，从而保持一个线程执行，线程一旦执行结束会唤醒虚拟队列中所有的等待线程，所以它是一种非公平竞争。JDK6 对 `synchronized` 进行了优化，引入了轻量级锁、偏向锁、自旋锁等概念，比如自旋锁，等待线程不需要在线程池中等待，而是通过空循环等待，当之前线程锁释放后，可以立刻获得锁，从而提供了性能。

备注：JDK6 以上锁的状态锁一共有四种状态：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态。感兴趣的可以自行查一下这几种锁的定义。

1.2 Synchronized 对象锁和类锁的区别

解析：只有很清楚时，此题才能从容作答，半迷糊状态很容易被绕晕，所以建议此题大家务必掌握，这个是你掌握多线程的基础。

下图是我简单写了一个小代码片段，面试的过程中，可能面试官可能随便写一个这样的代码，没有下面的 `main` 方法，然后问你，当多个线程执行这块代码时，程序是如何执行的？

回答这样的问题，只要抓住对象锁和类锁（抽象的，可以理解 `Class` 对象锁）两个概念，对象锁作用域在对象范围内（简单说就是一个 `new` 之间），多个对象之间没有关联关系，而

类锁是多个对象共用一个锁的监控器(执行下面注释 4 的代码便知);而且锁不具有继承性。

```
public class SynchronizedTest {
    public synchronized void testA(){} //1、对象锁
    public void testB(){} // 2、对象锁
        synchronized(this){}
    }
    public static synchronized void testC(){} //3、类锁
    public void testD() throws Exception{} //4、类锁
        synchronized(SynchronizedTest.class){
            for(int i=0;i<10;i++){
                System.out.println(Thread.currentThread().getName()+"->class lock:"+i);
                Thread.sleep(1000);
            }
        }
    }
    public static void main(String[] args) {
        ExecutorService dataService = Executors.newFixedThreadPool(3) ; // 构造三个线程
        for(int i=0;i<3;i++){
            dataService.execute(new Thread(){
                public void run() {
                    try {
                        new SynchronizedTest().testD();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            });
        }
    }
}
```

常见提问：1、直接问 `synchronized` 对象锁和类锁区别？（易）2、面试官随手在纸上写一些代码，主要是在静态方法，静态方法块，对象方法，或者直接用 `synchronized(this)` 上，问你他们的区别，或者执行结果之类的？（难）

参考答案：

一、`Synchronized` 对象锁：对象锁是当前对象中，若有多个 `synchronized` 使用的是一个锁的监控器，多个对象之间互不影响，它的作用域仅仅是当前对象。

常用方式：在普通方法或者语句块上加 `synchronized`，或者使用 `synchronized(this)`。

特点：1、当多个线程执行一个对象中的 `synchronized` 方法时，只有一个线程获得锁，其它线程会被阻塞。

2、当多个线程执行一个对象中的 `synchronized` 方法时，其它线程可以执行当前对象的非 `synchronized` 方法。

3、每个对象都有一个锁的监控器，创建多个对象时，对象之间的锁互不影响。

4、锁不具备传递性，也不具备继承性，父子之间的锁是独立的。

二、`Synchronized` 类锁：所谓类锁只是一个抽象概念，并没有实际类锁定义；类加载到常量池中也是以 `Class` 对象存在，所谓类锁可以理解成一个全局的对象锁。

常用方式：在静态方法或者静态语句块上加锁，或者用 `synchronized(类.class)`。

特点：类锁在所有对象中起作用，多个对象中用到的是一个锁的监控器。

1.3 foreach 实现原理

解析：这个问题很多同学可能关注过 `foreach` 与 `for` 的效率问题，由于没有源码可查，

所以关于它的实现原理，很多第一次被问到的同学很容易懵逼，这类问题就是考察大家的知识面，如果有能力的同学可以通过用 `javap` 编译一下 DEMO 的字节码看一下，如果仅仅为了面试那就好记下面的结论。

`Foreach` 主要用在数组和集合中，下面我把集合的字节码给大家看一下，至于数组大家可以自己动手，可以通过 `javap` 命令也可以安装 `eclipse` 插件。

```

8 public static void main(String[] args) {
9     List<String> list = new ArrayList<String>();
10    list.add("a");
11    list.add("b");
12    list.add("c");
13    Iterator<String> it = list.iterator();
14    while(it.hasNext()){
15        System.out.println(it.next());
16    }
            
```

com/dongych/thread/ForeachTest

LINENUMBER 15 L7

FRAME APPEND [Ljava/util/List java/util/Iterator]

GETSTATIC java/lang/System.out : Ljava/io/PrintStream;

ALOAD 2

INVOKEINTERFACE java/util/Iterator.next ()Ljava/lang/Object;

CHECKCAST java/lang/String

INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/String;)V

L6

LINENUMBER 14 L6

FRAME SAME

ALOAD 2

INVOKEINTERFACE java/util/Iterator.hasNext ()Z

IFNE L7

L8

LINENUMBER 21 L8

RETURN

L9

LOCALVARIABLE args [Ljava/lang/String; L0 L9 0

```

8 public static void main(String[] args) {
9     List<String> list = new ArrayList<String>();
10    list.add("a");
11    list.add("b");
12    list.add("c");
13    for(String s:list){
14        System.out.println(s);
15    }
            
```

com/dongych/thread/ForeachTest

LINENUMBER 13 L4

ALOAD 1

INVOKEINTERFACE java/util/List.iterator ()Ljava/util/Iterator;

ASTORE 2

GOTO L5

L6

FRAME FULL [[Ljava/lang/String; java/util/List T java/util/Iterator] []

ALOAD 3

INVOKEINTERFACE java/util/Iterator.next ()Ljava/lang/Object;

CHECKCAST java/lang/String

ASTORE 2

L7

LINENUMBER 14 L7

GETSTATIC java/lang/System.out : Ljava/io/PrintStream;

ALOAD 2

INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/String;)V

L5

LINENUMBER 13 L5

左边是通过迭代器遍历，右边是通过 `foreach` 遍历；`foreach` 遍历的字节码可以看出，底层实现也是用迭代器来实现的；所以我们可以肯定 `foreach` 只是 JVM 编译器的一个语法糖而已。下面的 `switch` 还会看到。

参考答案：

for-each 主要用于集合和数组元素的遍历，对于它的原理是通过编译器一种隐藏方式实现的，也就是在源码中不能直接看到它的实现代码，基于理论分析可以分为数组实现和集合实现两种。

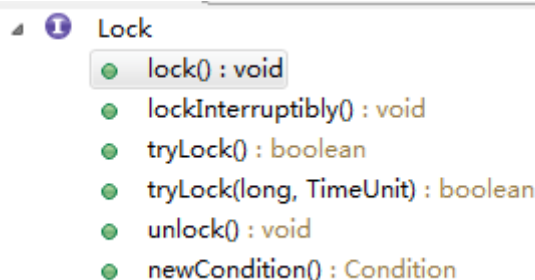
数组 for-each 实现：通过 `javap` 命令可以看到，它的实际执行就是循环遍历数组。（如果不确定，可以通过 `for` 循环的方式对照一下）。

集合 for-each 实现：必须实现 `Iterable` 接口，这个接口中只定义了一个 `Iterator<T> iterator()` 方法，同样通过 `javap` 命令输出可以看出，对于集合容器是输出是通过迭代来实现的。

备注：上面的字节码是通过 eclipse 插件导出的。

1.4 Lock 锁的实现

解析：Java 应用中为了保证线程安全通常是通过加锁，Jdk1.5 之前通常是通过 `synchronized` 来锁定需要同步的代码，而且在 `jdk1.5` 之前是一个重量级锁。为了对 Java 之前版本的兼容性，Jdk1.6 后除了对 `synchronized` 进行了优化，同时引入了 `Lock` 这种轻量锁。



```
Lock
  lock() : void
  lockInterruptibly() : void
  tryLock() : boolean
  tryLock(long, TimeUnit) : boolean
  unlock() : void
  newCondition() : Condition
```

参考答案：

`Lock` 是一种轻量级锁，它相比 `synchronized` 定义更细致的锁操作，它定义了 `lock`、`lockInterruptibly`、`tryLock`、`tryLock(long,timeUnit)` 四个获取锁的操作，提供更轻量的、`tryLock(long,timeUnit)` 定时锁，`lockInterruptibly()` 阻断锁，在调用 `interrupt` 方法时，可以阻断当前线程的等待，当使用锁时必须调用 `unlock()` 方法进行释放。它有一个直接实现类 `ReentrantLock`（可重入锁），和一个间接实现类 `ReentrantReadWriteLock`，`ReentrantReadWriteLock` 中的内部类读锁和写锁是实现了 `Lock` 的。这里说一下 `ReentrantLock` 的实现原理，它有一个内部类 `sync`，这个类实现了 `AQS` 接口 `acquire`、`tryAcquire` 获取锁，`release`、`tryRelease` 释放锁；而 `AQS` 内部则是通过维护一个线程队列和一个 `int` 型 `volatile` 原子变量 `state` 来实现线程对锁的获取和释放。

这里大家可以自己查一下 `ReentrantLock` 和 `ReentrantReadWriteLock` 锁的使用场景。

说明：这里的 `newCondition()` 方法暂时不说，大家可以结合 `AQS` 原理自己看一下，这块比较复杂，上面这些足够用。下面关于 `J.U.C` 包下的工具类，都是基于 `AQS` 和 `CAS` 实现的，每个同步类实现都是相有相似内心，如果你想成为更细入的学习请参考 `AQS` 源码，阅读它。

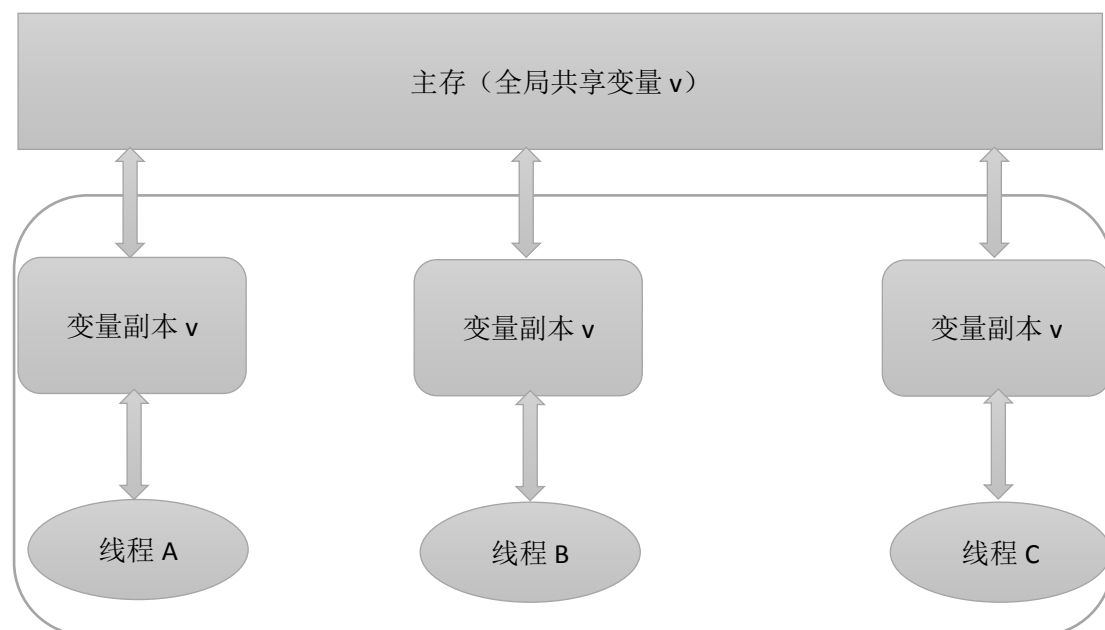
>推荐网上学习博客：<http://www.blogjava.net>

1.5 Volatile 的内存可见性

解析：这个关键字在互联网开发面试中，问到的越来越多，一般会 and `synchronized` 作一下比较，`synchronized` 肯定是可以保证内存可见性和原子性的，而且是重量级保证，在上面已经分析过了；`volatile` 是一弱锁的实现，可以更轻量不阻塞的保证变量的可见性，但不能保证对象的原子性，比如对 `count++` 操作，虽然把 `count` 设置成了 `volatile` 在多个线程下依然不能保证其原子性。

共享变量的安全问题，如下图所示，多个线程在读一个全局的共享变量时，是会把把量从主存中先加载到各自的工作内存中，然后在工作内存中进行操作。若没有特殊处理，线程之间是变量的修改是不可见的。当用 `volatile` 修饰变量时，当一个线程在执行变量的操作前会先和主存中的变量同步，同时修改后会同步刷新到主存中，这个过程是阻塞的。

Volatile 可见性的保障，一是通过编译优化，时时与主存同步刷新，另外一种禁止编译器的指令重排序。（关于指令重排，大家可以自行查找资料，内容比较深入 JVM）



参考答案：**volatile** 通常是通过一种弱锁的实现变量内存的可见性，它主要通过两种方式来保证内存的可见性；第一，当多个线程把主存中的数据加载到各自的工作空间作变更操作时，它能保证共享变量的副本时时与主存同步更新（**happens-before** 原则，简单总结 **volatile** 变量的修改对其它线程可见）。第二，它通过在编译器加载的执行到该关键字的位置前后插入一条 **StoreStore** 和 **StoreLoad** 屏障来防止编译器指令重排序。

追问一：可以创建 **volatile** 数组吗？

追问二：**volatile** 能否让一个非原子性变成原子性操作？

这两个的答案都是可以，这里不多作解释说明，动手查一下，收货会满满的哦。

1.6 常用的安全集合类

解析：这个题目主要想问一下通常多线程下怎么保证集合对象的安全使用，通过用到的 **set, list** 一些接口的实现集合类，大多数是非线程安全的，如果保证在多线程中安全中，可以通过 **Collections.synchronizedList(集合对象)** 来保证线程安全，当然也有线程安全的实现，像 **Vector**。在 **JDK5.0** 版本后引入 **J.U.C** 线程安全框架，里面有很多大师写好的线程安全的集合框架（建议大家打开看下），比如 **CopyOnWriteArrayList**、**CopyOnWriteArraySet** 等，当你回答完这些，马上就会被问到它们的实现原理，这些 **J.U.C** 中的类的实现原理基本核心都是一致的，都是继承了 **AQS**（这个大家一定要看，一通全通，这里不深入）。

参考答案：常用的安全集合类如 **Vector**，它是通过 **synchronized** 来对每个方法进行加锁，从而保持强一致性。其它常用的 **arrayList, hashset** 等都是通过 **Collections** 提供的同步方法 **collections.synchronizedlist(set)...** 来保持同步的。**JDK5** 引入了 **J.U.C** 并发包，在引入的并发包中提供了很多线程安全操作的集合，如 **CopyOnWriteArrayList**（用于读多写少的环境）、**CopyOnWriteArraySet** 等。

备注：其它这里问到最后，还是想考一下安全集合类的原理和使用场景，原理就不多说了。

了，本文档前前后后会提到多次，建议先看一下 AQS 的源码就一切 OK 了。

1.7 Switch 能否用 string 做参数

解析：在 JDK1.6 之前的版本中，大家都知道有四种常的基本类型 int,short,byte,char，而在 JDK1.7 版中引入了 String 类型。之所以 String 能作为 switch 的入参，那是因为 String 的特殊性能，不可变性。

如下图所示，右边为反编译后的代码，在反编译的代码中，switch 中的字符串变成了字符串的 hashCode，也就是一个 int 型整数。所以用 String 作为 switch 入以，可以看作是一个编译器的障眼法，或者说语法糖。

```
public class SwitchTest {  
    public static void main(String[] args) {  
        String str = "123";  
        switch (str) {  
            case "1":  
                break;  
            case "2":  
                break;  
            default:  
                System.out.println("OK");  
        }  
    }  
}  
  
public class SwitchTest  
{  
    public static void main(String[] args)  
    {  
        String str1;  
        String str = "123";  
        switch ((str1 = str).hashCode()) { ca  
            if (!(str1.equals("1")));  
            break;  
            case 50:  
                if (!(str1.equals("2")));  
            }  
            System.out.println("OK");  
        }  
    }  
}
```

原代码

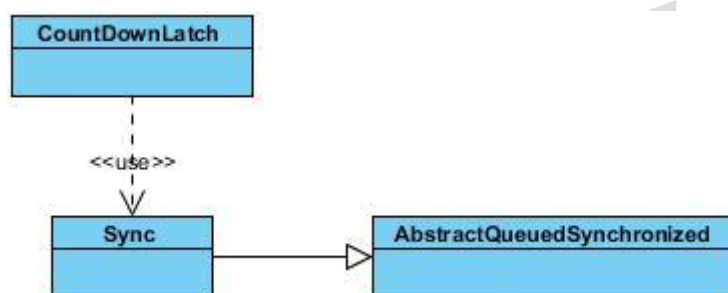
反编译代码

参考答案：switch 在 jdk1.7 版本以后，开始支持对 String 入参的支持，它是一种编译器的语法糖实现，实际底层传入的是 String 对象的 hashCode，一个 int 型整数。

1.8 CountdownLatch 实现原理

解析：这是互联网开发中经常被问到的一个 J.U.C 并发包下的并发辅助类。使用方法：创建对象时通过构造方法设置初始值，调用 CountdownLatch 对象的 await() 方法则处于等待状态，调用 countDown() 方法就将计数器减 1，当计数到达 0 时，唤醒所有等待者或单个等待者开始执行。使用示例，大家打开 JDK 中 CountdownLatch 源码，在源码上面的注释中有一个自带的 DEMO，写的非常清晰（看不懂注释也没有关系，COPY 下来先运行一下）。

至于它的实现原理，上面也有提到，基本上所有强大的并发工具类，都有一颗相似的心，基本上都有一个 Sync 类实现了 AQS 实现的。[图片源于网络]



参考答案： CountdownLatch 是 J.U.C 并发包下的一个并发工具辅助类，它内部有一个静态的 Static 的 Sync 类实现 AQS 类，在 AQS 中维护着一个双向队列和 volatile 变量的 state 值。CountdownLatch 实例的值会传递给 state，当 CountdownLatch 对象的 await 方法调用时，调用线程将被阻塞并放到 AQS 同步队列中；当 CountdownLatch 对象中的 countDown 方法被用时，AQS 中的 state 将减一，当 state=0 时会唤醒同步队列中头节点中的等待线程，然后依次链式唤醒其它同步队列中的等待线程。

说明：J.U.C 下的并发工具大多数实现都很像，把 AQS 弄懂，一切面试都 OK 了。

1.9 ThreadLocal 可能造成内存溢出

解析：这个问题可以有两种问法，一个是问 ThreadLocal 实现原理，一个就是像我们题目所问到的 ThreadLocal 内存溢出问题，如果仅仅问内存溢出的问题，你想回答的清楚也需要把 ThreadLocal 原理解释一下。

ThreadLocal 是 Thread 一个局部变量，不是为了解决多线程访问共享变量，而是为每个线程创建一个单独的变量副本，保持对象的方法和避免参数传递的复杂性，这个可以根据 Thread 的源码看到。至于为什么会内存溢出，可以分析一下它的核心代码（答案中分析）。

```

static class Entry extends WeakReference<ThreadLocal> {
    /** The value associated with this ThreadLocal. */
    Object value;

    Entry(ThreadLocal k, Object v) {
        super(k);
        value = v;
    }
}
    
```

参考答案：ThreadLocal 是用来存放 Thread 的变量副本，变量副本存储是在 ThreadLocal

中声明了一个静态的 `ThreadLocalMap` 类, `ThreadLocalMap` 类中定义了一个静态的 `Entry` 类对象, 是用来存储对象的数据结构对象, 可以把它理解成一个以 `key,value` 键字对存储的 `Map`; 从源码中可以看到, 它的 `Key` 值是一个弱引用类型的 `ThreadLocal` 引用对象; 所以当 `ThreadLocal` 被回收时, 即 `Key` 值被回收了; 但 `ThreadLocalMap` 本身是被强引用的, `Entry` 中的 `Value` 对象一直存在, 而且不会被立即回收; 但由于 `ThreadLocal` 是依附于线程, 如果线程销毁, 其所附的对象都会被回收, 而不会内存溢出。

如果当使用线程池的时候, 线程只是被回收到线程池中, 而不会被销毁, 而且 `Entry` 中 `value` 对象过大时, 而且又不能用时释放是可能会造成内存溢出。

如何避免内存溢出呢? 最好的办法就是在线池中慎用 `threadLocal`, 当然也是最糟糕的建议; 好的东西自然提倡使用, 只要在线程结束的地方手动关闭一下。

1.10 线程池的实现原理

解析: JDK 提供了很多已经实现好的线程池, 在实际开发中可以直接使用 (下面会提到), JDK 中已经实现的线程池工具类, 底层实现都很相似, 都是调用了 `ThreadPoolExecutor` 线程池类, 它的核心构造如下图, 只要理解了构造中的参数, 线程池的实现原理就容易明白。

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

参考答案: 要说清楚线程池的工作原理, 就要先了解一下线程池的一些参数设置; `corePoolSize`:核心线程数, 就是线程池在空闲的时候, 也会一直保持的线程数。

`maximumPoolSize`:最大线程数, 但任务队列满的时候, 线程数大于 `corePoolSize` 而小于 `maximumPoolSize`, 会创建线程, 直到线程数等于 `maximumPoolSize`。

`keepAliveTime`:闲置时, 线程存活的时间。

`workQueue`:任务队列。

`work`:工作线程。

`workers`:工作线程集合。

`RejectedExecutionHandler`: 运行策略 (默认是抛异常)。

线程池的原理:1、在线池启动时, 线程中并没有线程, 当进行调用 `execute` 方法时, 会进行线程的初始化。2、会首先判断当前线程数是不是小于 `corePoolSize`, 如果是小于, 则会直接创建线程, 并且执行任务。3、如果当前执行线程数等 `corePoolSize`, 此时新增加的任务会被放到工作队列 `workQueue` 中去。4、当 `workQueue` 队列任务满时, 会判断当前线程是否小于 `maximumPoolSize`, 如果小于就会继续增加线程到 `maximumPoolSize` 数。

5、如果当前线程数等于 `maximumPoolSize`，而且任务队列也满时，将会调用一些策略处理，比如抛异常 `RejectExecutionException`。

当工作队列中的任务结束后，而当前线程数量大于 `corePoolSize` 数时，若闲置时间超过 `keepAliveTime` 时间，线程将会关闭至 `corePoolSize`。

1.11 类的加载机制

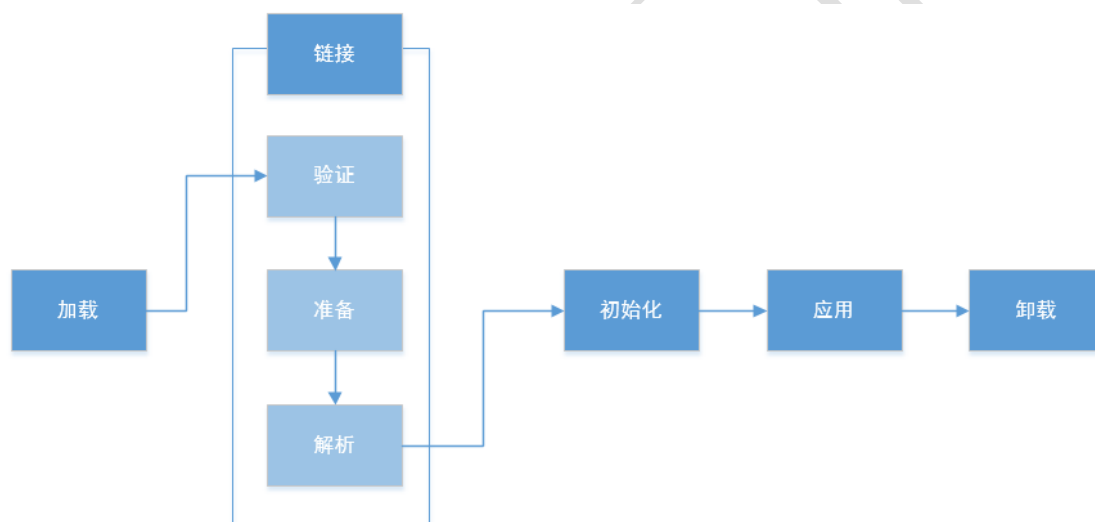
解析：这是一个最近面试中问到比较多的问题，回到不好直接可以 PASS 了，类的加载机制只是 JVM 原理的一个部分，所以求职者必须很熟练的才可以。而且这个看似高深的问题其实很简单，它是死的东西，只要理解掌握就 OK 了；可以把下面的图用你自己的话在面试中表达一遍就 OK 了。

类的加载步骤：**加载---链接（验证、准备、解析）---初始化---使用---卸载。**

加载：可以介绍一下类的加载器，加载类时的委托机制。

链接：验证、准备、解析等方面说，可以说一下准备过程中的初始化与后面初始化区别。

初始化：描述一下静态变量与成员变量初始化的不同。



使用：卸载：可以不作为重点，已经超出加载的过程，属于类的生命周期内容了。

参考答案：类的加载机制可以分为加载-链接-初始化三个阶段，链接又可以分为验证、准备、解析三个过程。

加载：通过类的加载器查找并加载二进制字节流的过程，在堆内存中的方法区生成一个代表这个类的 `java.lang.Class` 对象，作为这个类的数据请求入口。（这里可以把上面类加载器加载文件的过程描述一下（参考版本一，不作重复））。

验证：主要是对一些词法、语法进行规范性校验，避免对 JVM 本身安全造成危害；比如对文件格式，字节码验证，无数据验证等。但验证阶段是非必须的，可以通过参数设置来进行关闭，以提高加载的时效。

准备：对类变量分配内存，并且对类变量预初始化，初始化成数据类型的原始值，比如 `static int a=11`，会被初始化成 `a=0`；如果是 `static double a =11`，则会被初始化成 `a=0.0`；而成员变量只会成实例化后的堆中初始化。

解析：把常量池中的符号引用转换为直接引用的过程。

初始化：对类的静态变量和静态块中的变量进行初始化。（上面的准备阶段可以作为预初始化，初始到变量类型的原值，但如果被 `final` 修饰会进行真正初始化）
上面加载、链接、初始化的各个阶段并不是彼此独立，而是交叉进行，这点很重要。

1.12 Redis 与 Memcached 区别

解析：这是互联网公司中，最常用 NoSQL 数据库中的两种，一般面试中会被问到用过哪些 NoSQL 数据库有哪些，说一下它们的特或者原理。

参考答案：Redis 和 Memcached 都是比较流行的 NoSQL 数据库，都是基于内存 key-value 存储的数据库。

Redis 主要特点为：1、除了支持简单的 k/v 类型数据，还提供 list、set、hash 等丰富的数据结构类型存储。2、Redis 支持数据库的 master-slave 模式的数据备份。3、Redis 可以将内存中的数据序列化到硬盘存储，服务启动时再从硬盘中加载到内存。4、Redis 可以主从同步，进行一主多从或者一主一从的同步。5、Redis 可以作为消息队列，时时统计计算的数据集，也适合作多数据类型的缓存等。

Memcached 主要特点：1、协议简单，支持对象类型相对单一，基于 libevent 的事件处理，采用 LRU 算法。2、Memcached 本身没有数据冗余机制，也不太有必要，分布式实现主要依赖客户端实现，利用 magent 做一主多从备份。3、Memcached 适合于缓存网页查询数据，缓存 SQL 查询语句，用户临时性数据，会话相关的信息。

说明，这里只是做一下简单比较，如果想深入理解，可以从内存管理机制和 IO 模型了解。

第二部分 VIEW 原题

2.1 Oracle 删除无序数据

题目：下面有一张 tt 表，里面有两个字段 name,password,存了三行都是 1，写条 SQL 语句删除第 2 行。

	NAME	PASSWORD
1	1	1
2	1	1
3	1	1

参考答案：`delete from tt where rowid = (select max(rowid) from tt t where rownum <= 2)`

2.2 智力题分裂的小球

题目：有一个瓶子，里面有一个小球，这个小球每一分钟分裂成两个小球（类似细胞分

裂)，二分钟分裂四个，依次分裂下去，30 分钟可以把瓶子装满；如果同时放两个小球，多长时间可以把瓶子装满。

参考答案：

$$\begin{aligned} &\text{设瓶子可满 } N. \text{ 时间为} \\ &2^{30} = N \\ &2 \times 2^x = N \end{aligned} \Rightarrow x+1=30 \Rightarrow x=29.$$

2.3 JDK 有哪些线程池工具类

解析：这个是 So Easy 的题，之前有同学问到，而我出去面试过程中的也被问到，我当时还确认问了一下，所以还是写出来一下（如果你线程池的知识并不深厚，千万别确认人家是不是在问线程池的原理，线程池的原理上面也会提到）。对于这种污辱智商的面试题，如果不会，就去查一下 JDK 的 API 就 OK。

参考答案：

JDK 的并发包 `java.util.concurrent` 中，提供了一个 `Executors` 类，该类中定义了很多常用的线程池，比如：

`newFixedThreadPool`：声明一个指定线程数量的线程池。

`newSingleThreadExecutor`：每次声明一个单独的线程执行。

`newCachedThreadPool`：创建一个可以缓冲的线程池，理论上最大可以支持 `int` 类开的最大值，但工作中的线程闲置一段时间后会自动终止。

`newScheduleThreadPool`：建一个可以安排在给定延迟后运行命令或定期执行的线程池。

2.4 写一个死锁 DEMO

解析：死锁一般会在笔试中出现，偶尔会在面试中问到，此题也不难，只要实际动手写过就很容易掌握；造成死锁的原因（源于网络），1、互斥使用，即当资源被一个线程使用(占有)时，别的线程不能使用 2、不可抢占，资源请求者不能强制从资源占有者手中夺取资源，资源只能由资源占有者主动释放。3、请求和保持，即当资源请求者在请求其他的资源的同时保持对原有资源的占有。4、循环等待，即存在一个等待队列：T1 和 T2 互相等释放，这样就形成了一个等待环。

参考答案:

```
public class DeadLockTest {
    public static void main(String[] s) {
        Object o1 = new Object();
        Object o2 = new Object();
        new Thread(new DeadLocakThread(o1, o2), "T1").start();
        new Thread(new DeadLocakThread(o2, o1), "T2").start();
    }
}

class DeadLocakThread implements Runnable {
    private Object o1;
    private Object o2;

    public DeadLocakThread(Object o1, Object o2) {
        this.o1 = o1;
        this.o2 = o2;
    }

    @Override
    public void run() {
        synchronized (o1) { // 获得了o1锁
            try {
                System.out.println(Thread.currentThread().getName() + ": 获得了锁一");
                Thread.sleep(1000);
            } catch (Exception e) {
                e.printStackTrace();
            }
            synchronized (o2) { // 获取了o2锁
                try {
                    System.out.println(Thread.currentThread().getName()
                        + ": 获得了锁二");
                    Thread.sleep(1000);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

禁止复制