@来源 春节期间，我肝了这份高频面试题解 |《面霸系列》

@来源 这30个我精选的含答案的面试题，硬不硬你说吧

1.你觉得 Java 好在哪儿?

这种笼统的问题如果对某些知识点没有深入、系统地认识绝对会蒙！

所以为什么经常碰到面试官问你一些空、大的问题？其实就是考察你是否有形成体系的理解。

回到问题本身。我觉得可以从跨平台、垃圾回收、生态三个方面来阐述。

首先 Java 是跨平台的，不同平台执行的机器码是不一样的，而 Java 因为加了一层中间层 JVM ，所以可以做到一次编写多平台运行，即 「Write once,Run anywhere」。

编译执行过程是先把 Java 源代码编译成字节码，字节码再由 JVM 解释或 JIT 编译执行，而因为 JIT 编译时需要预热的，所以还提供了 AOT（Ahead-of-Time Compilation），可以直接把字节码转成机器码，来让程序重启之后能迅速拉满战斗力。

（解释执行比编译执行效率差，你想想每次给你英语让你翻译阅读，还是直接给你看中文，哪个快？）

Java 还提供垃圾自动回收功能，虽说手动管理内存意味着自由、精细化地掌控，当时很容易出错。

在内存较充裕的当下，将内存的管理交给 GC 来做，减轻了程序员编程的负担，提升了开发效率，更加划算！

然后现在 Java 生态圈太全了，丰富的第三方类库、网上全面的资料、企业级框架、各种中间件等等，总之你要的都有。

基本上这样答差不多了，之后等着面试官延伸。

当然这种开放性问题没有固定答案，我的回答仅供参考。

2.如果让你设计一个 HashMap 如何设计?

这个问题我觉得可以从 HashMap 的一些关键点入手，例如 hash函数、如何处理冲突、如何扩容。

可以先说下你对 HashMap 的理解。

比如：HashMap 无非就是一个存储 <key,value> 格式的集合，用于通过 key 就能快速查找到 value。

基本原理就是将 key 经过 hash 函数进行散列得到散列值，然后通过散列值对数组取模找到对应的 index 。

所以 hash 函数很关键，不仅运算要快，还需要分布均匀，减少 hash 碰撞。

而因为输入值是无限的，而数组的大小是有限的所以肯定会有碰撞，因此可以采用拉链法来处理冲突。

为了避免恶意的 hash 攻击，当拉链超过一定长度之后可以转为红黑树结构。

当然超过一定的结点还是需要扩容的，不然碰撞就太严重了。

而普通的扩容会导致某次 put 延时较大，特别是 HashMap 存储的数据比较多的时候，所以可以考虑和 redis 那样搞两个 table 延迟移动，一次可以只移动一部分。

不过这样内存比较吃紧，所以也是看场景来 trade off 了。

不过最好使用之前预估准数据大小，避免频繁的扩容。

基本上这样答下来差不多了，HashMap 几个关键要素都包含了，接下来就看面试官怎么问了。

可能会延伸到线程安全之类的问题，反正就照着 currentHashMap 的设计答。

3.并发类库提供的线程池实现有哪些?

虽说阿里巴巴Java 开发手册禁止使用这些实现来创建线程池，但是这问题我被问过好几次，也是热点。

问着问着就会延伸到线程池是怎么设计的。

> 我先来说下线程池的内部逻辑，这样才能理解这几个实现。

首先线程池有几个关键的配置：核心线程数、最大线程数、空闲存活时间、工作队列、拒绝策略。

1. 默认情况下线程不会预创建，所以是来任务之后才会创建线程（设prestartAllCoreThreads可以预创建核心线程）。
2. 当核心线程满了之后不会新建线程，而是把任务堆积到工作队列中。
3. 如果工作队列放不下了，然后才会新增线程，直至达到最大线程数。
4. 如果工作队列满了，然后也已经达到最大线程数了，这时候来任务会执行拒绝策略。
5. 如果线程空闲时间超过空闲存活时间，并且线程线程数是大于核心线程数的则会销毁线程，直到线程数等于核心线程数（设置allowCoreThreadTimeOut 可以回收核心线程）。

我们再回到面试题来，这个实现指的就是 Executors 的 5 个静态工厂方法：

- newFixedThreadPool
- newWorkStealingPool
- newSingleThreadExecutor
- newCachedThreadPool
- newScheduledThreadPool

newFixedThreadPool

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

这个线程池实现特点是核心线程数和最大线程数是一致的，然后 keepAliveTime 的时间是 0 ，队列是无界队列。

按照这几个设定可以得知它任务线程数是固定，如其名 Fixed。

然后可能出现 OOM 的现象，因为队列是无界的，所以任务可能挤爆内存。

它的特性就是**我就固定出这么多线程，多余的任务就排队，就算队伍排爆了我也不管。**

因此不建议用这个方式来创建线程池。

newWorkStealingPool

```
public static ExecutorService newWorkStealingPool() {
    return new ForkJoinPool
        (Runtime.getRuntime().availableProcessors(),
         ForkJoinPool.defaultForkJoinWorkerThreadFactory,
         handler: null, asyncMode: true);
}
```

这个是1.8才有的，从代码可以看到返回的就是 ForkJoinPool，我们1.8用的并行流就是这个线程池。

比如`users.parallelStream().filter(...).sum();`用的就是 ForkJoinPool 。

从图中可以看到线程数会参照当前服务器可用的处理核心数，我记得并行数是核心数-1。

这个线程池的特性从名字就可以看出 Stealing，**会窃取任务。**

每个线程都有自己的**双端队列**，当自己队列的任务处理完毕之后，会去别的线程的任务队列尾部拿任务来执行，加快任务的执行速率。

至于 ForkJoin 的话，就是分而治之，把大任务分解成一个个小任务，然后分配执行之后再总和结果，再详细就自行查阅资料啦~

newSingleThreadExecutor

```
@NotNull
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor( corePoolSize: 1, maximumPoolSize: 1,
                                 keepAliveTime: 0L, TimeUnit.MILLISECONDS,
                                 new LinkedBlockingQueue<Runnable>()));
}
```

这个线程池很有个性，一个线程池就一个线程，一个人一座城，配备的也是无界队列。

它的特性就是**能保证任务是按顺序执行的。**

newCachedThreadPool

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor( corePoolSize: 0, Integer.MAX_VALUE,
                                   keepAliveTime: 60L, TimeUnit.SECONDS,
                                   new SynchronousQueue<Runnable>());
}
```

这个线程池是急性子，核心线程数是 0 ，最大线程数看作无限，然后**任务队列是没有存储空间的**，简单理解成来个任务就必须找个线程接着，不然就阻塞了。

cached 意思就是会缓存之前执行过的线程，缓存时间是 60 秒，这个时候如果有任务进来就可以用之前的线程来执行。

所以它适合用在**短时间内有大量短任务的场景。如果暂无可用线程，那么来个任务就会新启一个线程去执行这个任务，快速响应任务。**

但是如果任务的时间很长，那存在的线程就很多，上下文切换就很频繁，切换的消耗就很明显，并且存在太多线程在内存中，也有 OOM 的风险。

newScheduledThreadPool

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}
```

```
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE, keepAliveTime: 0, NANOSECONDS,
          new DelayedWorkQueue());
}
```

其实就是定时执行任务，重点就是那个延时队列。

关于 Java 的几个定时任务调度相关的：Timer、DelayQueue 和 ScheduledThreadPool，我之前文章都分析过了，还介绍了时间轮在netty和kafka中的应用，有兴趣的可以看看。

4.如果让你设计一个线程池如何设计?

这种设计类问题还是一样，先说下理解，表明你是知道这个东西的用处和原理的，然后开始 BB。基本上就是按照现有的设计来说，再添加一些个人见解。

线程池讲白了就是存储线程的一个容器，池内保存之前建立过的线程来重复执行任务，减少创建和销毁线程的开销，提高任务的响应速度，并便于线程的管理。

我个人觉得如果要设计一个线程池的话得考虑池内工作线程的管理、任务编排执行、线程池超负荷处理方案、监控。

初始化线程数、核心线程数、最大线程池都暴露出来可配置，包括超过核心线程数的线程空闲消亡配置。

任务的存储结构可配置，可以是无界队列也可以是有界队列，也可以根据配置分多个队列来分配不同优先级的任务，也可以采用 stealing 的机制来提高线程的利用率。

再提供配置来表明此线程池是 IO 密集还是 CPU 密集型来改变任务的执行策略。

超负荷的方案可以有多种，包括丢弃任务、拒绝任务并抛出异常、丢弃最旧的任务或自定义等等。

线程池埋好点暴露出用于监控的接口，如已处理任务数、待处理任务数、正在运行的线程数、拒绝的任务数等等信息。

我觉得基本上这样答就差不多了，等着面试官的追问就好。

注意不需要跟面试官解释什么叫核心线程数之类的，都懂的没必要。

当然这种开放型问题还是仁者见仁智者见智，我这个不是标准答案，仅供参考。

5.GC 如何调优?

GC 调优这种问题肯定是具体场景具体分析，但是在面试中就不要讲太细，大方向说清楚就行，不需要涉及具体的垃圾收集器比如 CMS 调什么参数，G1 调什么参数之类的。

GC 调优的核心思路就是尽可能的使对象在年轻代被回收，减少对象进入老年代。

具体调优还是得看场景根据 GC 日志具体分析，常见的需要关注的指标是 Young GC 和 Full GC 触发频率、原因、晋升的速率 、老年代内存占用量等等。

比如发现频繁会产生 Full GC，分析日志之后发现没有内存泄漏，只是 Young GC 之后会有大量的对象进入老年代，然后最终触发 Ful GC。所以就能得知是 Survivor 空间设置太小，导致对象过早进入老年代，因此调大 Survivor 。

或者是晋升年龄设置的太小，也有可能分析日志之后发现是内存泄漏、或者有第三方类库调用了System.gc等等。

反正具体场景具体分析，核心思想就是尽量在新生代把对象给回收了。

基本上这样答就行了，然后就等着面试官延伸了。

6.动态代理是什么?

动态代理就是一个代理机制，动态是相对于静态来说的。

代理可以看作是调用目标的一个包装，通常用来在调用真实的目标之前进行一些逻辑处理，消除一些重复的代码。

静态代理指的是我们预先编码好一个代理类，而动态代理指的是运行时生成代理类。

动态更加方便，可以指定一系列目标来动态生成代理类(AOP)，而不像静态代理需要为每个目标类写对应的代理类。

代理也是一种解耦，目标类和调用者之间的解耦，因为多了代理类这一层。

常见的动态代理有 JDK 动态代理 和 CGLIB。

7.JDK 动态代理与 CGLIB 区别?

JDK 动态代理是基于接口的，所以**要求代理类一定是有定义接口的。**

CGLIB 基于ASM字节码生成工具，它是通过继承的方式来实现代理类，所以**要注意 final 方法。**

之间的性能随着 JDK 版本的不同而不同，以下内容取自：haiq的博客

> - jdk6 下，在运行次数较少的情况下，jdk动态代理与 cglib 差距不明显，甚至更快一些；而当调用次数增加之后， cglib 表现稍微更快一些
> - jdk7 下，情况发生了逆转！在运行次数较少（1,000,000）的情况下，jdk动态代理比 cglib 快了差不多30%；而当调用次数增加之后(50,000,000)， 动态代理比 cglib 快了接近1倍
> - jdk8 表现和 jdk7 基本一致

基本上这样答差不多了，我们再看看 JDK 动态代理实现原理：

1. 首先通过实现 InvocationHandler 接口得到一个切面类。
2. 然后利用 Proxy 根据目标类的类加载器、接口和切面类得到一个代理类。
3. 代理类的逻辑就是把所有接口方法的调用转发到切面类的 invoke() 方法上，然后根据反射调用目标类的方法。

```
public class YesInvoker implements InvocationHandler {

    private Object target; //要代理的目标对象

    public Object newProxyInstance(Object target) {
        this.target = target; //传入目标对象
        //返回代理对象
        return Proxy.newProxyInstance(target.getClass().getClassLoader(), target.getClass().getInterfaces(), h: this);
    }


    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("公众号：yes的练级攻略");
        return method.invoke(this.target, args);
    }

    public static void main(String[] args) throws ServiceException {
        //得到代理类，可以看到只有运行的之后才会生成这个代理类。
        IAccountService accountService = (IAccountService) new YesInvoker().newProxyInstance(new AccountService());
        /**
         * 这个方法实际上转发到 YesInvoker#invoke 上，然后执行代理逻辑，最后通过反射调用目标对象方法
         */
        accountService.getUserInfo(new UserKey());
    }
```

再深一点点就是代理类会现在静态块中通过反射把所有方法都拿到存在静态变量中，我之前反编译看过代理类，我忙写了一下，大致长这样：

```
private static Method m;

static {
    try {
        m = Class.forName("com.yes.AccountService")
                .getMethod( name: "getUserInfo", new Class[] { Class.forName("com.yes.UserKey") });
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}

public final UserInfo getUserInfo(UserKey key) {
    // 会有个构造函数把 InvocationHandler 传进来，也就是这个 h
    super.h.invoke(this,m,new Object[] { key }));
}
```

这一套下来 JDK 动态代理原理应该就很清晰了。

再来看下 CGLIB，其实和 JDK 动态代理的实现逻辑是一致，只是实现方式不同。

```
        Enhancer en = new Enhancer();
        //2.设置父类，也就是代理目标类，上面提到了它是通过生成子类的方式
        en.setSuperclass(target.getClass());
        //3.设置回调函数，这个this其实就是代理逻辑实现类，也就是切面，可以理解为JDK 动态代理的handler
        en.setCallback(this);
        //4.创建代理对象，也就是目标类的子类了。
        return en.create();
```

然后它是通过字节码生成技术而不是反射来实现调用的逻辑，具体就不再深入了。

8.注解是什么原理?

注解其实就是一个标记，可以标记在类上、方法上、属性上等，标记自身也可以设置一些值。

有了标记之后，我们就可以**在解析的时候**得到这个标记，然后做一些特别的处理，这就是注解的用处。

比如我们可以定义一些切面，在执行一些方法的时候看下方法上是否有某个注解标记，如果是的话可以执行一些特殊逻辑(RUNTIME类型的注解)。

注解生命周期有三大类，分别是：

- RetentionPolicy.SOURCE：给编译器用的，不会写入 class 文件
- RetentionPolicy.CLASS：会写入 class 文件，在类加载阶段丢弃，也就是运行的时候就没这个信息了
- RetentionPolicy.RUNTIME：会写入 class 文件，永久保存，可以通过反射获取注解信息

所以我上文写的是解析的时候，没写具体是解析啥，因为不同的生命周期的解析动作是不同的。

像常见的：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

就是给编译器用的，编译器编译的时候检查没问题就over了，class文件里面不会有 Override 这个标记。

再比如 Spring 常见的 Autowired ，就是 RUNTIME 的，所以**在运行的时候可以通过反射得到注解的信息**，还能拿到标记的值 required 。

```
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.PARAMETER,
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Autowired {
    boolean required() default true;
}
```

所以注解就是一个标记，可以给编译器用、也能运行时候用。

9.反射用过吗?

如果你用过那就不用我多说啥了，场景说一下，然后等着面试官继续挖。

如果没用过那就说生产上没用过，不过私下研究过反射的原理。

反射其实就是Java提供的能在运行期可以得到对象信息的能力，包括属性、方法、注解等，也可以调用其方法。

一般的编码不会用到反射，在框架上用的较多，因为很多场景需要很灵活，所以不确定目标对象的类型，届时只能通过反射动态获取对象信息。

PS：对反射不了解的，可以网上查查，这里不深入了。

10.能说下类加载过程吗?

类加载顾名思义就是把类加载到 JVM 中，而输入一段二进制流到内存，之后经过一番解析、处理转化成可用的 class 类，这就是类加载要做的事情。

二进制流可以来源于 class 文件，或者通过字节码工具生成的字节码或者来自于网络都行，只要符合格式的二进制流，JVM 来者不拒。

类加载流程分为加载、连接、初始化三个阶段，连接还能拆分为：验证、准备、解析三个阶段。

所以总的来看可以分为 5 个阶段：

- 加载：将二进制流搞到内存中来，生成一个 Class 类。
- 验证：主要是验证加载进来的二进制流是否符合一定格式，是否规范，是否符合当前 JVM 版本等等之类的验证。
- 准备：为静态变量(类变量)赋初始值，也即为它们在方法区划分内存空间。这里注意是**静态变量**，并且是初始值，比如 int 的初始值是 0。
- 解析：将常量池的符号引用转化成直接引用。符号引用可以理解为只是个替代的标签，比如你此时要做一个计划，暂时还没有人选，你设定了个 A 去做这个事。然后等计划真的要落地的时候肯定要找到确定的人选，到时候就是小明去做一件事。

  解析就是把 A(符号引用) 替换成小明(直接引用)。符号引用就是一个字面量，没有什么实质性的意义，只是一个代表。直接引用指的是一个真实引用，在内存中可以通过这个引用查找到目标。
- 初始化：这时候就执行一些静态代码块，为静态变量赋值，这里的赋值才是代码里面的赋值，准备阶段只是设置初始值占个坑。

这个问题我觉得回答可以比我写的更粗，几个阶段一说，大致做的说一说就 ok 了。

想要知道更详细的流程可以看下《深入理解虚拟机Java》虚拟机的类加载章节。

11.双亲委派知道不？来说说看？

类加载机制一问基本上就会接着问双亲委派。

双亲委派的意思是：

如果一个类加载器需要加载类，那么首先它会把这个类加载请求委派给父类加载器去完成，如果父类还有父类则接着委托，每一层都是如此。

一直递归到顶层，当父加载器无法完成这个请求时，子类才会尝试去加载。

这里的双亲其实就指的是父类，没有mother。

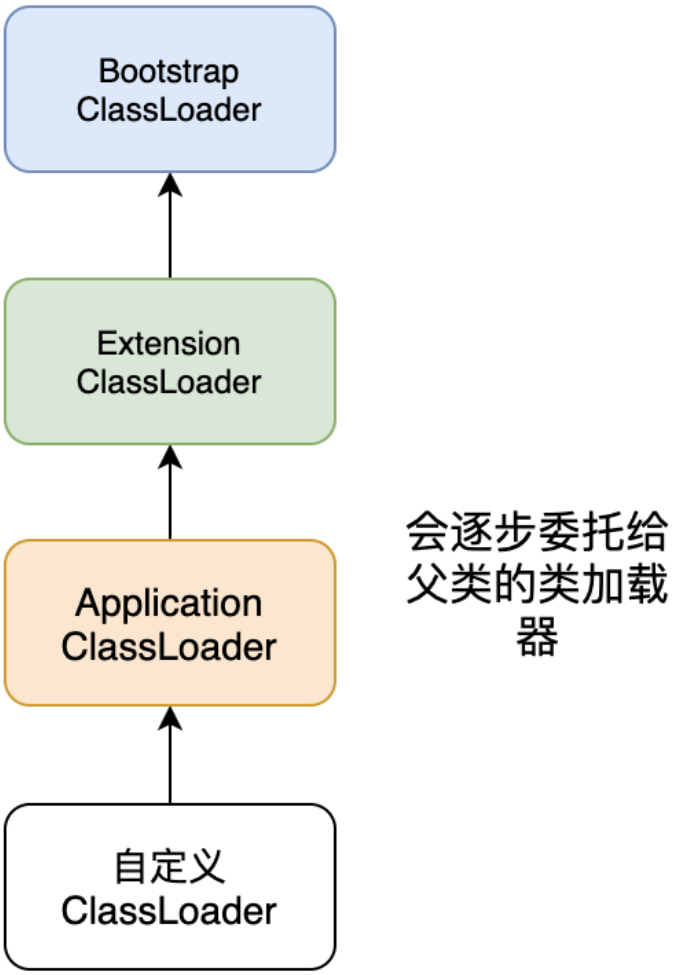
父类也不是我们平日所说的那种继承关系，只是调用逻辑是这样。

关于双亲委派我之前写过文章，我把一些比较重要的内容拷过来：

Java 自身提供了 3 种类加载器：

1. 启动类加载器(Bootstrap ClassLoader)，它是属于虚拟机自身的一部分，用 C++ 实现的，主要负责加载`<JAVA_HOME>\lib`目录中或被-Xbootclasspath指定的路径中的并且文件名是被虚拟机识别的文件。它是所有类加载器的爸爸。
2. 扩展类加载器(Extension ClassLoader),它是Java实现的，独立于虚拟机，主要负责加载`<JAVA_HOME>\lib\ext`目录中或被java.ext.dirs系统变量所指定的路径的类库。
3. 应用程序类加载器(Application ClassLoader),它是Java实现的，独立于虚拟机。主要负责加载用户类路径(classPath)上的类库，如果我们没有实现自定义的类加载器那这玩意就是我们程序中的默认加载器。

所以一般情况类加载会从应用程序类加载器委托给扩展类再委托给启动类，启动类找不到然后扩展类找，扩展类加载器找不到再应用程序类加载器找。



双亲委派模型不是一种强制性约束，也就是你不这么做也不会报错怎样的，**它是一种JAVA设计者推荐使用类加载器的方式。**

为什么要双亲委派?

它使得类有了层次的划分。就拿 java.lang.Object 来说，加载它经过一层层委托最终是由Bootstrap ClassLoader来加载的，也就是最终都是由Bootstrap ClassLoader去找<JAVA_HOME>\lib中rt.jar里面的java.lang.Object加载到JVM中。

这样如果有不法分子自己造了个java.lang.Object,里面嵌了不好的代码，如果我们是按照双亲委派模型来实现的话，最终加载到JVM中的只会是我们rt.jar里面的东西，也就是这些核心的基础类代码得到了保护。

因为这个机制使得系统中只会出现一个java.lang.Object。不会乱套了。你想想如果我们JVM里面有两个Object，那岂不是天下大乱了。

那你知道有违反双亲委派的例子吗?

典型的例子就是：JDBC。

JDBC 的接口是类库定义的，但实现是在各大数据库厂商提供的 jar 包中，那通过启动类加载器是找不到这个实现类的，所以就需要应用程序加载器去完成这个任务，这就违反了自下而上的委托机制了。

具体做法是搞了个**线程上下文类加载器**，通过 setContextClassLoader() 默认设置了应用程序类加载器，然后通过 Thread.current.currentThread().getContextClassLoader() 获得类加载器来加载。

12.JDK 和 JRE 的区别?

JRE(Java Runtime Environment)指的是 Java 运行环境，包含了 JVM 和 Java 类库等。

JDK(Java Development Kit) 可以视为 JRE 的超集，还提供了一些工具比如各种诊断工具：jstack，jmap，jstat 等。

13.用过哪些 JDK 提供的工具?

这个就考察你平日里面有没有通过一些工具进行问题的分析、排查。

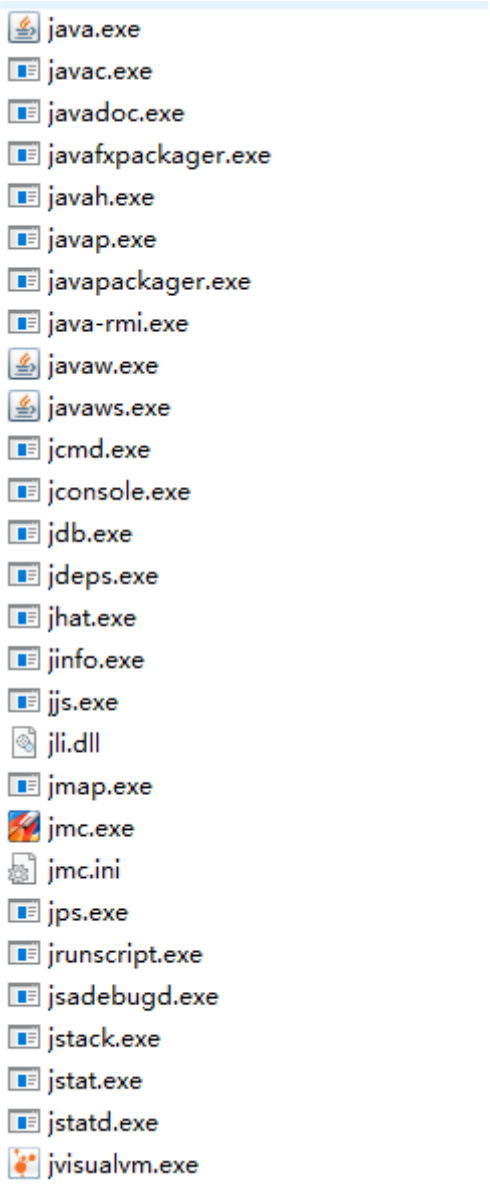
如果你用过肯定很好说，比如之前排查内存异常的时候用 jmap dump下来内存文件用 MAT 进行分析之类的。

如果没用过的话可以试试，自己找场景试验一下。

我列几个之前写过文章的工具，建议自己用用，还是很简单的。

- jps：虚拟机进程状况工具
- jstat：虚拟机统计信息监视工具
- jmap：Java内存映像工具
- jhat：虚拟机堆转储快照分析工具
- jstack：Java堆栈跟踪工具
- jinfo：Java配置信息工具
- VisualVM：图形化工具，可以得到虚拟机运行时的一些信息：内存分析、CPU 分析等等，在 jdk9 开始不再默认打包进 jdk 中。

工具其实还有很多，看看下面这个截图。



更详细的可以去《深入理解虚拟机Java》第四章查看。

总之就是自己找机会用用，没机会就自己给自己创造机会，防范于未然。

14.接口和抽象类有什么区别?

接口：只能包含抽象方法，不能包含成员变量，当 has a 的情况下用接口。

接口是对行为的抽象，类似于条约。在 Java 中接口可以多实现，从 has a 角度来说接口先行，也就是先约定接口，再实现。

抽象类: 可以包含成员变量和一般方法和抽象方法，当 is a 并且主要用于代码复用的场景下使用抽象类继承的方式，子类必须实现抽象类中的抽象方法。

在 Java 中只支持单继承。从 is a 角度来看一般都是先写，然后发现代码能复用，然后抽象一个抽象类。

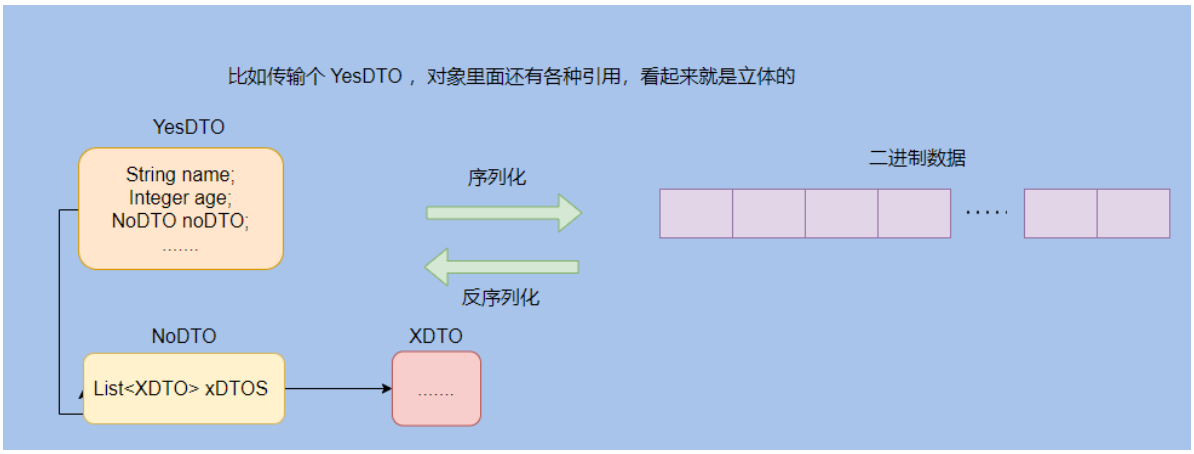
15.什么是序列化？什么是反序列化?

序列化其实就是将对象转化成可传输的字节序列格式，以便于存储和传输。

因为对象在 JVM 中可以认为是“立体”的，会有各种引用，比如在内存地址Ox1234 引用了某某对象，那此时这个对象要传输到网络的另一端时候就需要把这些引用“压扁”。

因为网络的另一端的内存地址 Ox1234 可以没有某某对象，所以传输的对象需要包含这些信息，然后接收端将这些扁平的信息再反序列化得到对象。

所以**反序列化就是将字节序列格式转换成对象的过程。**



我再扩展一下 Java 序列化。

> 首先说一下 Serializable，这个接口没有什么实际的含义，就是起标记作用。

来看下源码就很清楚了，除了 String、数组和枚举之外，如果实现了这个接口就走 `writeOrdinaryObject`，否则就序列化就抛错。

```
// remaining cases
if (obj instanceof String) {
    writeString((String) obj, unshared);
} else if (cl.isArray()) {
    writeArray(obj, desc, unshared);
} else if (obj instanceof Enum) {
    writeEnum((Enum<?>) obj, desc, unshared);
} else if (obj instanceof Serializable) {
    writeOrdinaryObject(obj, desc, unshared);
} else {
    if (extendedDebugInfo) {
        throw new NotSerializableException(
            cl.getName() + "\n" + debugInfoStack.toString());
    } else {
        throw new NotSerializableException(cl.getName());
    }
}
```

serialVersionUID 又有什么用?

`private static final long serialVersionUID = 1L;`

想必经常会看到这样的代码，这个 ID 其实就是用来验证序列化的对象和反序列化对应的对象ID 是否一致。

所以这个 ID 的数字其实不重要，无论是 1L 还是 idea 自动生成的，只要序列化时候对象的 serialVersionUID 和反序列化时候对象的 serialVersionUID 一致的话就行。

如果没有显示指定 serialVersionUID ，则编译器会根据类的相关信息自动生成一个，可以认为是一个指纹。

所以如果你没有定义一个 serialVersionUID 然后序列化一个对象之后，在反序列化之前把对象的类的结构改了，比如增加了一个成员变量，则此时的反序列化会失败。

因为类的结构变了，生成的指纹就变了，所以 serialVersionUID 就不一致了。

所以 serialVersionUID 就是起验证作用。

Java 序列化不包含静态变量

简单地说就是序列化之后存储的内容不包含静态变量的值，看下下面的代码就很清晰了。

```
public class Test implements Serializable {

    private static final long serialVersionUID = 1L;

    public static int yes = 1;

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

            ObjectOutputStream out = new ObjectOutputStream(
                    new FileOutputStream( name: "公众号yes的练级攻略"));
            out.writeObject(new Test());
            out.close();

            //序列化后修改为值为 2
            Test.yes = 2;

            ObjectInputStream oin = new ObjectInputStream(new FileInputStream(
                    name: "公众号yes的练级攻略"));
            Test t = (Test) oin.readObject();
            oin.close();

            //此时的值是 2，而不是1，因为 yes 是静态变量
            System.out.println(t.yes);
```

```
Test
/Library/Java/JavaVirtualMachines/jdk1.8.0_281.jdk/Contents/Home/bin/java ...
2

Process finished with exit code 0
```

16.什么是不可变类?

不可变类指的是无法修改对象的值，比如 String 就是典型的不可变类，当你创建一个 String 对象之后，这个对象就无法被修改。

因为无法被修改，所以像执行 `s += "a"`; 这样的方法，其实返回的是一个新建的 String 对象，老的 s 指向的对象不会发生变化，只是 s 的引用指向了新的对象而已。

所以才会有不要在字符串拼接频繁的场景不要使用 + 来拼接，因为这样会频繁的创建对象。

不可变类的好处就是**安全**，因为知晓这个对象不可能会被修改，因此可以放心大胆的用，在多线程环境下也是线程安全的。

如何实现一个不可变类?

这个问题我被面试官问过，其实就参考 String 的设计就行。

String 类用 final 修饰，表示无法被继承。

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];
```

String 本质是一个 char 数组，然后用 final 修饰，不过 final 限制不了数组内部的数据，所以这还不够。

所以 value 是用 private 修饰的，并且没有暴露出 set 方法，这样外部其实就接触不到 value 所以无法修改。

当然还是有修改的需求，比如 replace 方法，所以这时候就需要返回一个新对象来作为结果。

```
public String replace(char oldChar, char newChar) {
    if (oldChar != newChar) {
        int len = value.length;
        int i = -1;
        char[] val = value; /* avoid getfield opcode */

        while (++i < len) {
            if (val[i] == oldChar) {
                break;
            }
        }
        if (i < len) {
            char buf[] = new char[len];
            for (int j = 0; j < i; j++) {
                buf[j] = val[j];
            }
            while (i < len) {
                char c = val[i];
                buf[i] = (c == oldChar) ? newChar : c;
                i++;
            }
            return new String(buf, share: true);
        }
    }
    return this;
}
```

总结一下就是私有化变量，然后不要暴露 set 方法，即使有修改的需求也是返回一个新对象。

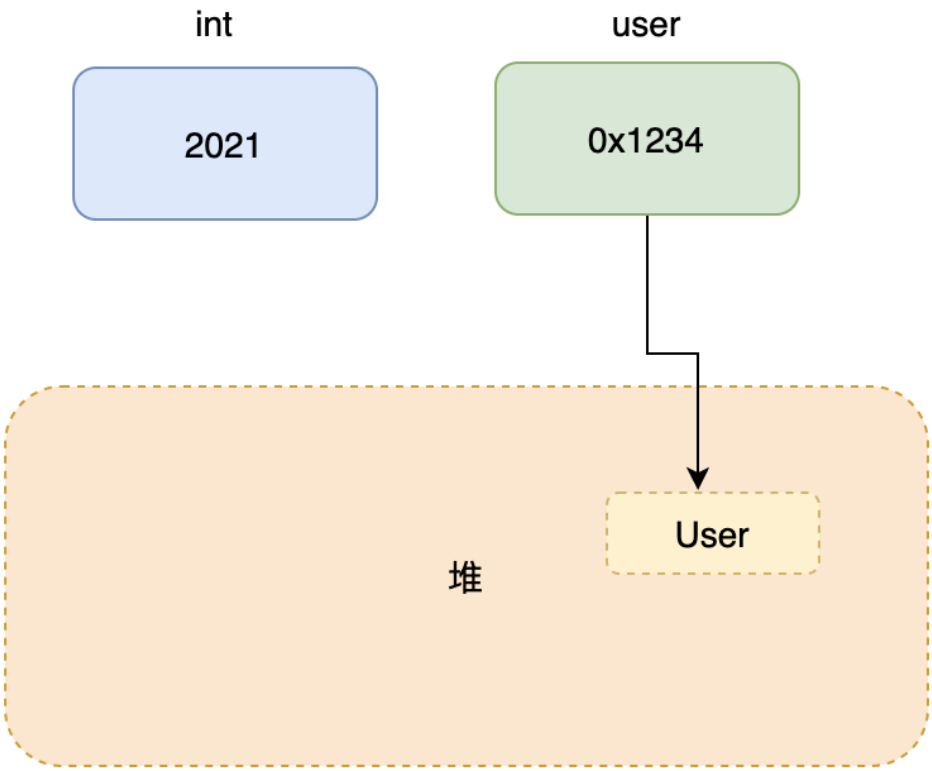
17.Java 按值传递还是按引用传递?

Java 只有按值传递，不论是基本类型还是引用类型。

基本类型是值传递很好理解，引用类型有些同学可能有点理解不了，特别是初学者。

JVM 内存有划分为栈和堆，局部变量和方法参数是在栈上分配的，基本类型和引用类型都占 4 个字节，当然 long 和 double 占 8 个字节。

而对象所占的空间是在堆中开辟的，引用类型的变量存储对象在堆中地址来访问对象，所以传递的时候可以理解为把变量存储的地址给传递过去，因此引用类型也是值传递。



18.泛型有什么用？泛型擦除是什么?

泛型可以把类型当作参数一样传递，使得像一些集合类可以明确存储的对象类型，不用显示地强制转化（在没泛型之前只能是Object，然后强转）。

并且在编译期能识别类型，类型错误则会提醒，增加程序的健壮性和可读性。

泛型擦除指的指参数类型其实在编译之后就被抹去了，也就是生成的 class 文件是没有泛型信息的，所以称之为擦除。

不过这个擦除有个细节，我们来看下代码就很清晰了，代码如下：

```
public class Yes<T> {
    private Yes<String> yess;
    private T y;
}
```

然后我们再来看看编译后的 class 文件。

```
Constant pool:
   #1 = Methodref          #3.#18         // java/lang/Object."<init>":()V
   #2 = Class              #19            // Yes
   #3 = Class              #20            // java/lang/Object
   #4 = Utf8               yess
   #5 = Utf8               LYes;
   #6 = Utf8               Signature
   #7 = Utf8               LYes<Ljava/lang/String;>;
   #8 = Utf8               y
   #9 = Utf8               Ljava/lang/Object;
  #10 = Utf8               TT;
  #11 = Utf8               <init>
```

可以看到 yess 是有类型信息的，所以在代码里写死的泛型类型是不会被擦除的！

这也解释了为什么根据反射是可以拿到泛型信息的，因为这种写死的就没有被擦除！

至于泛型擦除是为了向后兼容，因为在 JDK 5 之前是没有泛型的，所以要保证 JDK 5 之前编译的代码可以在之后的版本上跑，而类型擦除就是能达到这一目标的一个实现手段。

其实 Java 也可以搞别的手段来实现泛型兼容，只是擦除比较容易实现。

19.说说强、软、弱、虚引用?

Java 根据其生命周期的长短将**引用类型**又分为强引用、软引用、弱引用、幻象引用。

- 强引用：就是我们平时 new 一个对象的引用。当 JVM 的内存空间不足时，宁愿抛出 OutOfMemoryError 使得程序异常终止，也不愿意回收具有强引用的存活着的对象。
- 软引用：生命周期比强引用短，当 JVM 认为内存空间不足时，会试图回收软引用指向的对象，也就是说在 JVM 抛出 OutOfMemoryError 之前，会去清理软引用对象，适合用在内存敏感的场景。
- 弱引用：比软引用还短，在 GC 的时候，不管内存空间足不足都会回收这个对象，ThreadLocal中的 key 就用到了弱引用，适合用在内存敏感的场景。
- 虚引用：也称幻象引用，之所以这样叫是因为虚引用的 get 永远都是 null ，称为get 了个寂寞，所以叫虚。

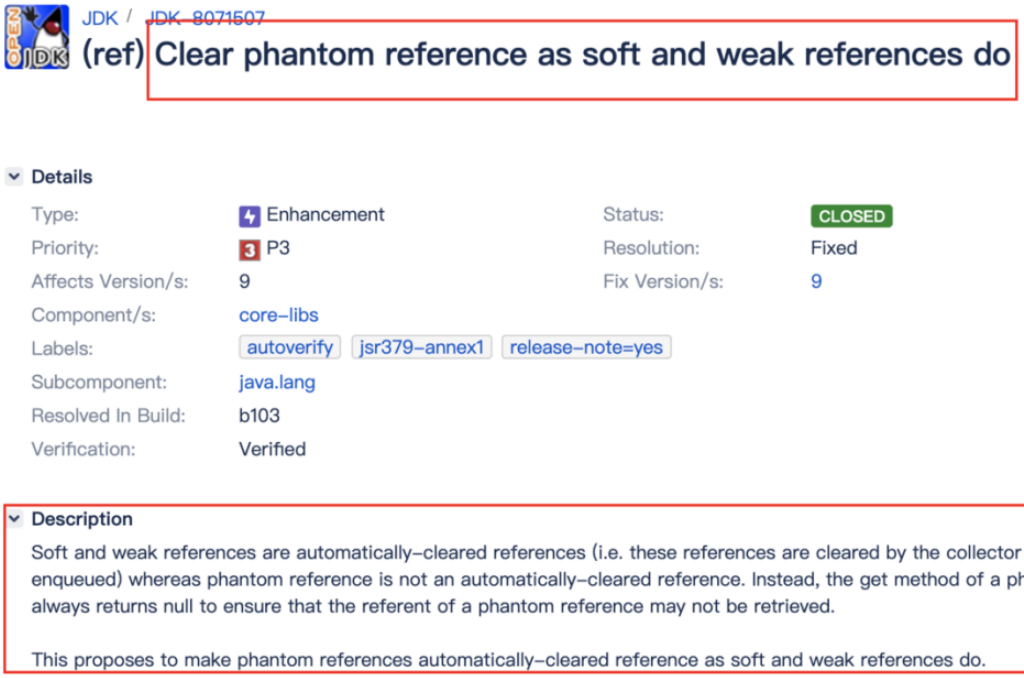
虚引用的唯一作用就是配合引用队列来监控引用的对象是否被加入到引用队列中，也就是可以准确的让我们知晓对象何时被回收。

还有一点有关虚引用的需要提一下，之前看文章都说虚引用对 gc 回收不会有任何的影响，但是看 1.8 doc 上面说

```
 * <p> Unlike soft and weak references, phantom references are not
 * automatically cleared by the garbage collector as they are enqueued.  An
 * object that is reachable via phantom references will remain so until all
 * such references are cleared or themselves become unreachable.
 *
 * @author   Mark Reinhold
 * @since    1.2
 */

public class PhantomReference<T> extends Reference<T> {
```

> 简单翻译下就是：与软引用和弱引用不同，虚引用在排队时不会被垃圾回收器自动清除。通过**虚引用可访问的对象将保持这种状态，直到所有这些引用被清除或者它们本身变得不可访问。**

简单的说就是被虚引用引用的对象不能被 gc，然而在 JDK9 又有个变更记录：



链接：https://bugs.openjdk.java.net/browse/JDK-8071507

按照这上面说的 JDK9 之前虚引用的对象是在虚引用自身被销毁之前是无法被 gc 的，而 JDK9 之后改了。

我没下 JDK9，不过我有 JDK11，所以看了下 11 doc 的确实改了。

```
 * <p> In order to ensure that a reclaimable object remains so, the referent of
 * a phantom reference may not be retrieved: The {@code get} method of a
 * phantom reference always returns {@code null}.
 *
 * @author   Mark Reinhold
 * @since    1.2
 */

public class PhantomReference<T> extends Reference<T> {
```

看起来是把那段删了。所以 JDK9 之前虚引用对引用对象的GC是有影响的，9及之后的版本没影响。

20.Integer 缓存池知道吗?

因为根据实践发现大部分的数据操作都集中在值比较小的范围，因此 Integer 搞了个缓存池，默认范围是 -128 到 127，可以根据通过设置`JVM-XX:AutoBoxCacheMax=<size>`来修改缓存的最大值，最小值改不了。

实现的原理是int 在自动装箱的时候会调用Integer.valueOf，进而用到了 IntegerCache。

```
@HotSpotIntrinsicCandidate
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

没什么花头，就是判断下值是否在范围之内，如果是的话去 IntegerCache 中取。

IntegerCache 在静态块中会初始化好缓存值。

```
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[];

    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue =
            VM.getSavedProperty( key: "java.lang.Integer.IntegerCache.high");
        if (integerCacheHighPropValue != null) {
            try {
                int i = parseInt(integerCacheHighPropValue);
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
            } catch( NumberFormatException nfe) {
                // If the property cannot be parsed into an int, ignore it.
            }
        }
        high = h;

        cache = new Integer[(high - low) + 1];
        int j = low;
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);

        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }

    private IntegerCache() {}
}
```

遍历创建对象

所以这里还有个面试题，就是啥 Integer 127 之内的相等，而超过 127 的就不等了，因为 127 之内的就是同一个对象，所以当然相等。

不仅 Integer 有，Long 也是有的，不过范围是写死的 -128 到 127。

```
@HotSpotIntrinsicCandidate
public static Long valueOf(long l) {
    final int offset = 128;
    if (l >= -128 && l <= 127) { // will cache
        return LongCache.cache[(int)l + offset];
    }
    return new Long(l);
}
```

对了 Float 和 Double 是没有滴，毕竟是小数，能存的数太多了。

21.Exception 和 Error 的区别知道吗?

Exception 是程序正常运行过程中可以预料到的意外情况，应该被开发者捕获并且进行相应的处理。

Error 是指在正常情况下不太可能出现的情况，绝大部分的 Error 都会导致程序处于不正常、不可恢复的状态，也就是挂了。

所以不便也不需被开发者捕获，因为这个情况下你捕获了也无济于事。

Exception和Error都是继承了Throwable类，在Java代码中只有继承了Throwable类的实例才可以被throw或者被catch。

随便我再提一提异常处理的注意点，之前写过文章总结过：

1.尽量不要捕获类似Exception这样通用的异常，而应该捕获特定的异常。

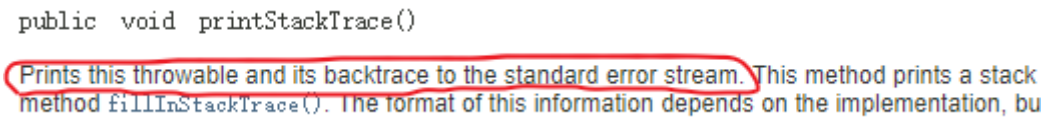
软件工程是一门协作的艺术，在日常的开发中我们有义务使自己的代码能更直观、清晰的表达出我们想要表达的信息。

但是如果你什么异常都用了Exception，那别的开发同事就不能一眼得知这段代码实际想要捕获的异常，并且这样的代码也会捕获到可能你希望它抛出而不希望捕获的异常。

2.不要"吞"了异常

如果我们捕获了异常，不把异常抛出，或者没有写到日志里，那会出现什么情况？线上除了 bug 莫名其妙的没有任何的信息，你都不知道哪里出错以及出错的原因。

这可能会让一个简单的bug变得难以诊断，而且有些同学比较喜欢用 catch 之后用e.printStackTrace()，在我们产品中通常不推荐用这种方法，一般情况下这样是没有问题的但是这个方法输出的是个标准错误流。



比如是在分布式系统中，发生异常但是找不到stacktrace。

所以最好是输入到日志里，我们产品可以自定义一定的格式，将详细的信息输入到日志系统中，适合清晰高效的排查错误。

3.不要延迟处理异常

比如你有个方法，参数是个 name，函数内部调了别的好几个方法，其实你的name传的是 null 值，但是你没有在进入这个方法或者这个方法一开始就处理这个情况，而是在你调了别的好几个方法然后爆出这个空指针。

这样的话明明你的出错堆栈信息只需要抛出一点点信息就能定位到这个错误所在的地方，进过了好多方法之后可能就是一坨堆栈信息。

4.只在需要try-catch的地方try-catch，try-catch的范围能小则小

只要必要的代码段使用try-catch，不要不分青红皂白try住一坨代码，因为try-catch中的代码会影响JVM对代码的优化，例如重排序。

5.不要通过异常来控制程序流程

一些可以用if/else的条件语句来判断例如null值等，就不要用异常，异常肯定是比一些条件语句低效的，有 CPU 分支预测的优化等。

而且每实例化一个Exception都会对栈进行快照，相对而言这是一个比较重的操作，如果数量过多开销就不能被忽略了。

6.不要在finally代码块中处理返回值或者直接return

在finally中return或者处理返回值会让发生很诡异的事情，比如覆盖了 try 中的return，或者屏蔽的异常。

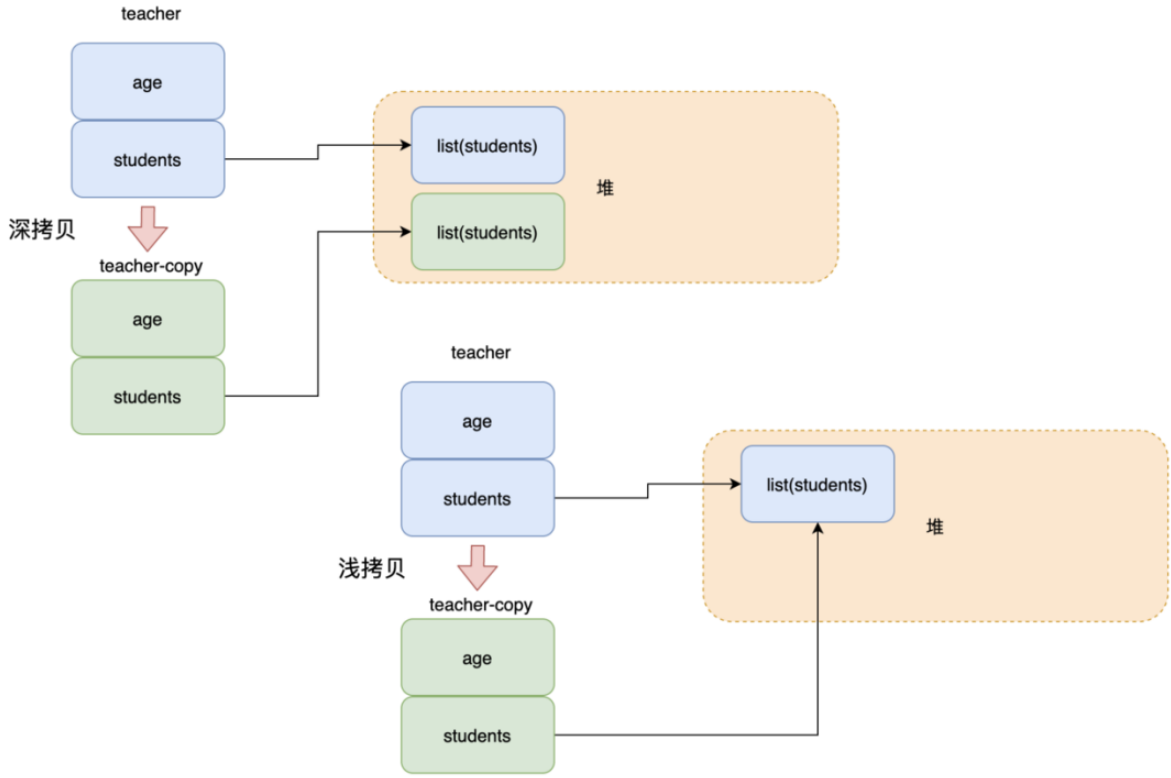
22.深拷贝和浅拷贝?

深拷贝：完全拷贝一个对象，包括基本类型和引用类型，堆内的引用对象也会复制一份。

浅拷贝：仅拷贝基本类型和引用，堆内的引用对象和被拷贝的对象共享。

所以假如拷贝的对象成员间有一个 list，深拷贝之后堆内有 2 个 list，之间不会影响，而浅拷贝的话堆内还是只有一个 list。

比如现在有个 teacher 对象，然后成员里面有一个 student 列表。



因此深拷贝是安全的，浅拷贝的话如果有引用对象则原先和拷贝对象修改引用对象的值会相互影响。

23.面向对象编程和面向过程编程的区别?

面向对象编程(Object Oriented Programming，OOP)是一种编程范式或者说编程风格。

把类或对象作为基本单元来组织代码，并且运用提炼出的：封装、继承和多态来作为代码设计指导。

面向过程编程是以过程作为基本单元来组织代码的，过程其实就是动作，对应到代码中来就是函数，面向过程中函数和数据是分离的，数据其实就是成员变量。

而面向对象编程的类中数据和动作是在一起的，这也是两者的一个显著的区别。

如果对这两个概念还是比较模糊的话，可以看我写的这篇文章，4800多字来讲面向对象和面向过程，看完之后肯定懂！面向对象和面向过程解析

24.重载与重写的区别?

重载：指的是方法名相同，参数类型或者顺序或个数不同，**这里要注意和返回值没有关系**，方法的签名是名字和参数列表，不包括返回值。

重写：指的是子类重写父类的方法，方法名和参数列表都相同，也就是方法签名是一致的。重写的子类逻辑抛出的异常和父类一样或者是其父类异常的子类，并且方法的访问权限不得低于父类。

简单的理解为儿子不要超过爸爸，要尊老爱幼。

25.什么是内部类，有什么用?

内部类顾名思义就是定义在一个类的内部的类，按位置分：在成员变量的位置定义，则是成员内部类，在方法内定义，则是局部内部类。

如果用 static 修饰则为静态内部类，还有匿名内部类。

一般而言只会用成员内部类、静态内部类和匿名内部类。

成员内部类可以使用外部类的所有成员变量以及方法，包括 private 的。

静态内部类只能使用外部类的静态成员变量以及方法。

匿名类常用来作为回调，使用的时候再实现具体逻辑来执行回调。

实际上内部类是一个编译层面的概念，像一个语法糖一样，经过编译器之后其实内部类会提升为外部顶级类，和外部类没有任何区别，所以**在 JVM 中是没有内部类的概念的。**

一般情况下非静态内部类用在内部类和其他类无任何关联，专属于这个外部类使用，并且也便于调用外部类的成员变量和方法，比较方便。

静态外部类其实就等于一个顶级类，可以独立于外部类使用，所以更多的只是表明类结构和命名空间。

26.说说 Java 的集合类吧?

这种问题一般大致提一下，然后等着面试官深挖。

常用的集合有 List、Set、Map、Queue 等。

List 常见实现类有 ArrayList 和 LinkedList。

- ArrayList 基于动态数组实现，支持下标随机访问，对删除不友好。
- LinkedList 基于双向链表实现，不支持随机访问，只能顺序遍历，但是支持O(1)插入和删除元素。

Set 常见实现类有：HashSet、TreeSet、LinkedHashSet。

- HashSet 其实就是 HashMap 包了层马甲，支持 O(1)查询，无序。
- TreeSet 基于红黑树实现，支持范围查询，不过基于红黑树的查找时间复杂度是O(lgn)，有序。
- LinkedHashSet，比 HashSet 多了个双向链表，通过链表保证有序。

Map 常见实现类有：HashMap、TreeMap、LinkedHashMap

- HashMap：基于哈希表实现，支持 O(1) 查询，无序。
- TreeMap：基于红黑树实现，O(lgn)查询，有序。
- LinkedHashMap：同样也是多了双向链表，支持有序，可以很好的支持 lru 的实现。

设置有序，并且重写LinkedHashMap中的 removeEldestEntry 方法，即可实现 lru。

> 这里有一点要提一下，如果你对某个东西比较熟悉就要在合适的地方抛出来。比如通过 LinkedHashMap 你还能延伸到 lru ，这表明你对 LinkedHashMap 有研究并且也知晓 lru，面试官自己可能都不清楚，会觉得你有点东西。

> 而且面试官基本会追问 lru 然后接着延伸，比如延伸到改进的 lru ， mysql 缓存中的 lru 等等，这就是通过你的引导把问题领域迁移到你自身熟悉的地方，这岂不美哉？如果你不熟悉，那少 bb。

Queue 常见的实现类有：LinkedList、PriorityQueue。

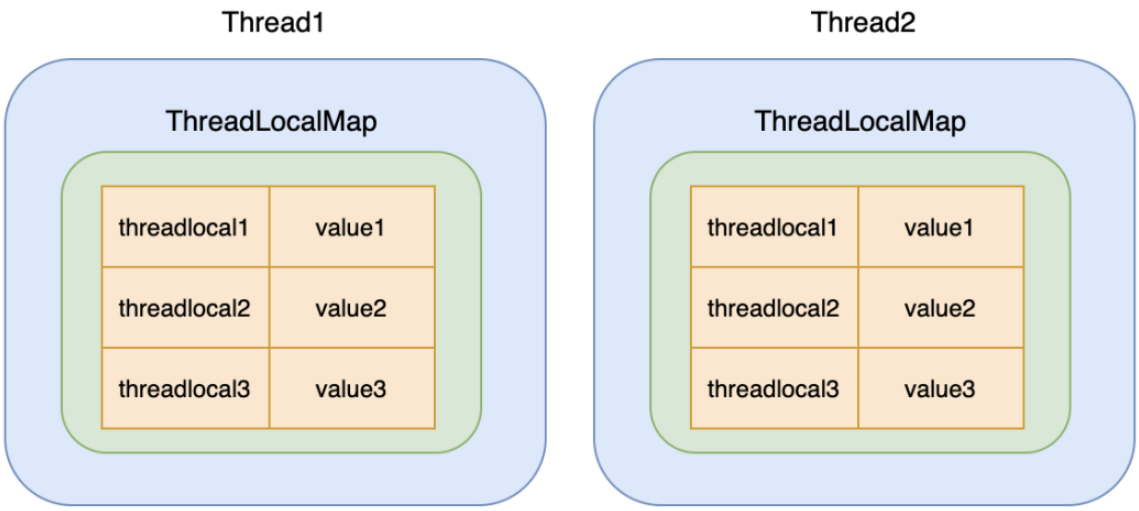
PriorityQueue：优先队列，是基于堆实现的，底层其实就是数组。

基本上回答不了这么全，稍微讲几个可能就被打断，然后深挖了，届时只能见招拆招。

27.说说 ThreadLocal ?

ThreadLocal 本质是通过本地化资源来避免共享，也就是每个线程都有自己的本地私有化变量，这样每个线程访问自己属性即可，避免了多线程竞争导致的锁等消耗。

具体关系如下图所示：



ThreadLocalMap 是采用线性探测的方式来解决 hash 冲突，所以要注意 ThreadLocal 的数量，因为这种冲突解决方式比较低效。

ThreadLocal 在 Entry 中作为 key 是弱引用，所以当外部对 ThreadLocal 的强引用消失之后，只剩下弱引用的 ThreadLocal 会被 GC 清除，这时候 Entry 中的 value 还在，但是已经访问不到了，所以称之为内存泄漏。

不过当调用 get 和 set 方法时，如果直接 hash 没中，开始线性探测，那么碰到 key 为 null 的节点才会清理掉。

当然更好的方式是显示的在用完之后调用 remove，这样就能及时清理。

既然弱引用会导致内存泄漏，那为什么还要弱引用?

首先如果 key 不用弱引用，那么当外部对 ThreadLocal 的强引用消失之后，由于 ThreadLocalMap 是这个线程的成员之一，所以这个线程还在，那么 ThreadLocalMap 就在，而 ThreadLocalMap 在，那么 Entry 肯定在，而 Entry 在那么强引用的 key 和 value 就肯定在。

所以如果 key 不用弱引用，那么 key 都无法被 GC 。

所以 key 用弱引用那么至少 key 这点内存是可以被省掉的，并且线性探测还能清一些 Entry。

其实发生内存泄漏的根本不在于 key 是弱引用，因为他们都属于一个线程的属性，所以线程活着它们就不能被 GC，这一条引用链是无法更改的。

然后现在都是用线程池，所以线程有可能长时间存活，因此就会逐渐堆积，导致内存满了。

所以这点需要明确。

与之相关的还有个 InheritableThreadLocal

这玩意可以理解为就是可以把父线程的 threadlocal 传递给子线程，所以如果要这样传递就用 InheritableThreadLocal ，不要用 threadlocal。

原理其实很简单，在 Thread 中已经包含了这个成员：

```
/* ThreadLocal values pertaining to this thread. This map is maintained
 * by the ThreadLocal class. */
ThreadLocal.ThreadLocalMap threadLocals = null;

/*
 * InheritableThreadLocal values pertaining to this thread. This map is
 * maintained by the InheritableThreadLocal class.
 */
ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
```

在父线程创建子线程的时候，子线程的构造函数可以得到父线程，然后判断下父线程的 InheritableThreadLocal 是否有值，如果有的话就拷过来。

```
private Thread(ThreadGroup g, Runnable target, String name,
               long stackSize, AccessControlContext acc,
               boolean inheritThreadLocals) {
    if (name == null) {
        throw new NullPointerException("name cannot be null");
    }

    this.name = name;

    Thread parent = currentThread();   // 获取父线程
    SecurityManager security = System.getSecurityManager();
    if (g == null) {
.java
    this.target = target;
    setPriority(priority);
    if (inheritThreadLocals && parent.inheritableThreadLocals != null)
        this.inheritableThreadLocals =
            ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
    /* Stash the specified stack size in case the VM cares */
    this.stackSize = stackSize;

    /* Set thread ID */
    this.tid = nextThreadID();
}
```

如果父线程不等于null，
并且允许inheritThreadLocal，
就把值拷过来

这里要注意，**只会在线程创建的时会拷贝 InheritableThreadLocal 的值**，之后父线程如何更改，子线程都不会首其影响。

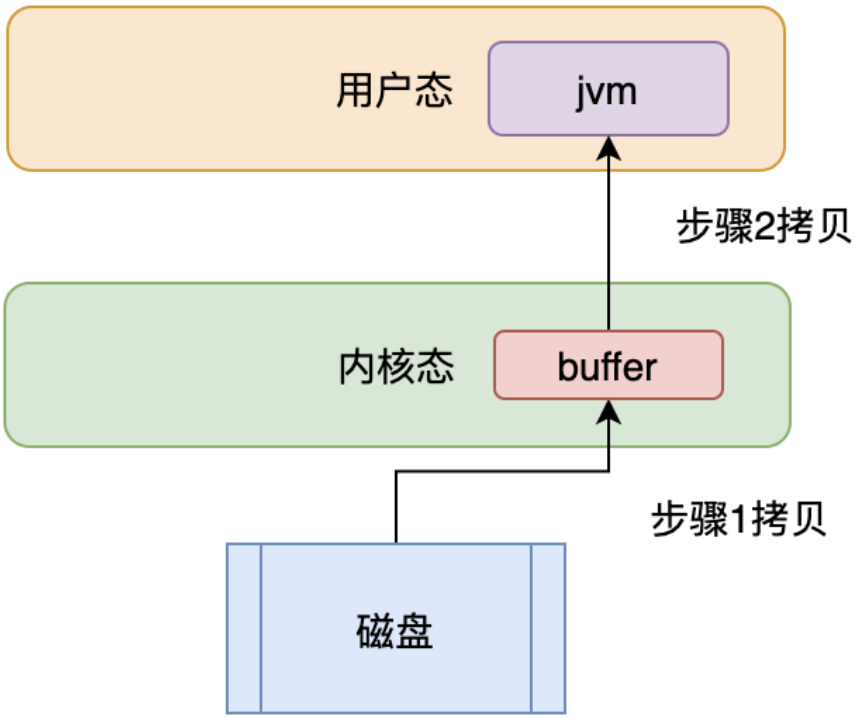
至此有关 ThreadLocal 的知识点就差不多了。

28.同步、异步、阻塞、非阻塞 IO 的区别?

我们的程序和硬件之间隔了个操作系统，而为了安全考虑，Linux 系统分了：用户态和内核态。

在这个前提下，我们再明确程序要从磁盘（网卡）读数据的两个步骤：

1. 数据从存储设备拷贝到内核缓存
2. 数据再从内核缓存拷贝到用户空间



好了，现在咱们可以看看这几个概念了。

- 同步I/O：指的是线程需要等待 2 执行完毕。
- 异步I/O：指的是线程不需要等待 2 执行。
- 阻塞I/O：指的是步骤 1 会阻塞，即线程需要阻塞等待 1 执行完毕。
- 非阻塞I/O：指的是步骤 1 不会被阻塞，不需要阻塞等待。
- 所以平时还会有同步阻塞I/O，或者啥同步非阻塞I/O，就是步骤 1 和 2 的组合罢了。

我们再来理解一下，毕竟咱们这面霸系列不是背诵，是理解。

同步和异步指的是：是否需要等待方法的调用执行完毕。

这两个概念主要注重的是调用方式，同步和异步调用编码方式是不同的，同步其实就是一条道写下来，异步则是需要回调、事件等方式来实现后面的逻辑。

阻塞和非阻塞：**一般用在底层系统调用身上，阻塞指的是线程未满足条件会被阻塞，进入 sleep 状态**，即时间片还未到就让出 CPU，非阻塞则是计算未满足条件也直接返回。

所以阻塞是真的被阻塞住了，是在等待数据，是需要让出时间片的。

而同步的线程其实还是有时间片的，所以同步一般有个超时时间，计算超时之后就会返回继续执行后面的代码。

29.BIO、NIO、AIO?

BIO 指的是同步阻塞I/O，相信看了 28 题之后对这个同步阻塞很清晰了，就是等着。

在这种模型下只能是来一个连接用一个线程，连接多并发大的话服务器顶不住这么多线程的。

NIO 指的是同步非阻塞I/O，我们熟知的 IO 多路复用就是NIO，适合用在连接多、每次传输较为短的场景。

AIO 指的是异步I/O，调用了之后就不管了，数据来了自动会执行回调方法。

异步可以有效的减少线程的等待，减少了用户线程拷贝数据的那段等待，效率更高。

30.JDK8 有哪些新特性?

JDK8 较为重要和平日里经常被问的特性如下：

- 用元空间替代了永久代。
- 引入了 Lambda 表达式。
- 引入了日期类、接口默认方法、静态方法。
- 新增 Stream 流式接口

然后相信你们对 HashMap 和 ConcurrentHashMap 有一定的准备，所以抛出来

- 修改了 HashMap 和 ConcurrentHashMap 的实现（等着八股文之问）
- 新增了 CompletableFuture 、StampedLock 等并发实现类。

像一些中间件异步化代码都用了 CompletableFuture 来实现，所以还是得做一些了解的，如果不熟悉这条就不用提了。

31.你都用过哪些 Java 并发工具类?

Semaphore、CyclicBarrier、CountDownLatch 三连。

当然 JUC 下面还有挺多，反正列几个说说就行，面试的时候切忌不要一股脑儿的把知道的都扔出来，这叫**留白**。

Semaphore

这玩意叫信号量，广泛应用于各种操作系统中，相对于平日只允许一个线程访问临界区的 lock 和 synchronized 来说，**信号量允许多线程同时访问一个临界区**。

原理就简单的理解为初始化一个数，如果来了一个线程则把数减一，如果减一之后数的值小于 0 则阻塞当前线程，移入一个阻塞队列中，否则允许执行。

当一个线程执行完毕之后将数加一，并唤醒阻塞队列中的一个等待线程。

实际是内部有个继承自 AQS 的 Sync 类，通过依托 AQS 的封装来实现功能。

主要用于流量的控制，比如停车场只允许停一定数量的车位。

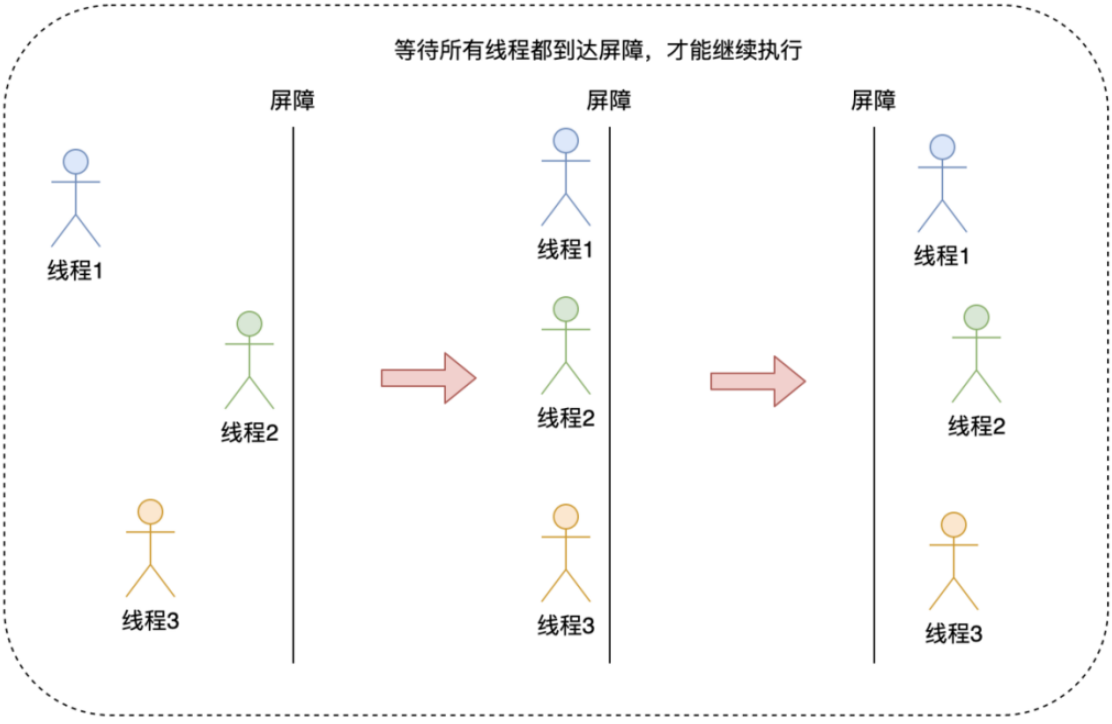
简单示例如下：

```
    int count;
    final Semaphore semaphore   = new Semaphore(1); // 初始化信号量
    // 用信号量保证互斥
    void addOne() {
      try {
          semaphore.acquire();   //对应down，计数减一
          count+=1;
        } catch (InterruptedException e) {
          e.printStackTrace();
        } finally {
          semaphore.release();  //对应up，计数加一
        }
    }
```

CyclicBarrier

从名字分析，这是一个可循环的屏障。

屏障的意思是：让一组线程都运行到同一个屏障点之后，线程会阻塞等待所有线程都达到这个屏障点，然后所有线程才得以继续执行。



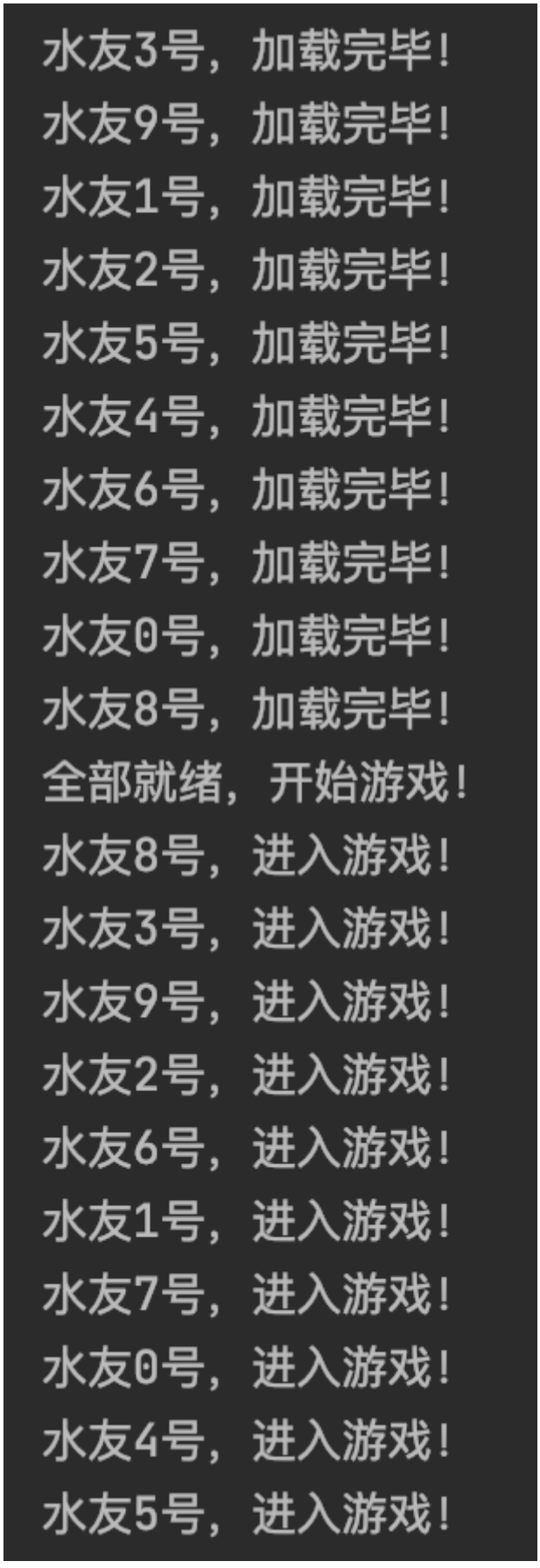
来看一下用法：

```java
static CyclicBarrier cyclicBarrier;

public static void main(String[]args){
    /**
     * 公众号：yes的练级攻略 ，准备的线上LOL水友赛。
     */
    cyclicBarrier = new CyclicBarrier( parties: 10, () -> System.out.println("全部就绪，开始游戏！"));

    for(int i = 0; i < 10; i++){
        new Thread(() -> {
            try {
                /**
                 * 模拟公众号：yes的练级攻略，水友们电脑加载游戏的时间
                 */
                Thread.sleep((long) (Math.random() * 3000));
                System.out.println(Thread.currentThread().getName() + "，加载完毕！");
                cyclicBarrier.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "，进入游戏！");
        }, name: "水友" + i + "号").start();
    }
}
```

这个是 barrierCommand

这是屏障

结果如下：



它实际上是基于 ReentrantLock 和 Condition 的封装来实现这一功能的。

原理我先口述一下，因为面试官很有可能会问原理。

首先设置了达到屏障的线程数量，当线程调用 await 的时候计数器会减一，如果计数器减一不等于 0 的时候，线程会调用 condition.await 进行阻塞等待。

如果计数器减一的值等于0，说明最后一个线程也到达了屏障，于是如果有 barrierCommand 就执行 barrierCommand ，然后调用 condition.signalAll 唤醒之前等待的线程，并且重置计数器，然后开启下一代。

源码我就不贴了，建议自己看下，不难的，算上一大推注释都不到 500 行，核心方法就 60 几行。

至于循环的话，来看一下这个代码就理解了：

```
private void nextGeneration() {
    // signal completion of last generation
    trip.signalAll();
    // set up next generation
    count = parties; 重置计数
    generation = new Generation();开始新的一代
}
```

当规定数量的线程到达屏障之后会把计数重置回去，并且开启了下一代，所以 CyclicBarrier 是可以循环使用的。

CountDownLatch

这个锁其实和 CyclicBarrier 有点类似，都是等待一个节点的到达，但是还是不太一样的。

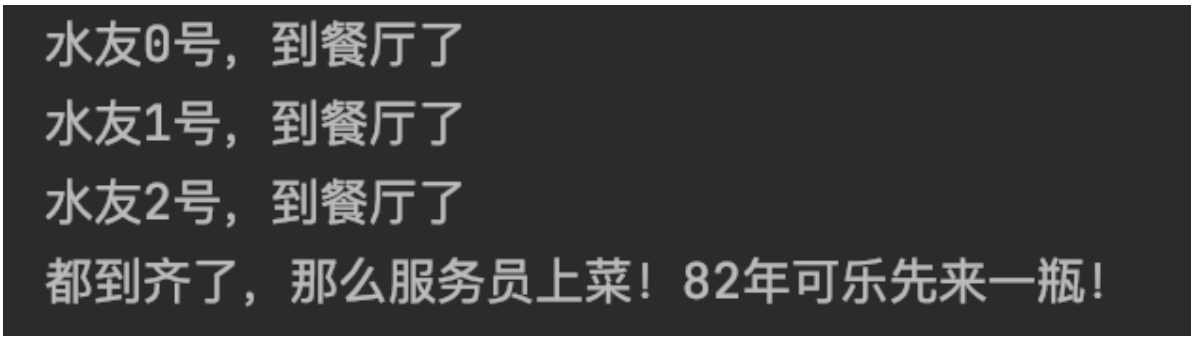
CyclicBarrier 是各个线程等待阻塞所有线程都达到一个节点之后，所有线程继续执行。

CountDownLatch 是一个线程阻塞着等待其他线程到达一个节点之后才能继续执行，**这个过程中其他线程是不会阻塞的。**

示例如下：

```java
public static void main(String[]args) throws InterruptedException {
    /**
     * 公众号：yes的练级攻略 ，请三个人吃饭。
     */
    countDownLatch = new CountDownLatch(3);

    for(int i = 0; i < 3; i++){
        new Thread(() -> {
            try {
                /**
                 * 模拟公众号：yes的练级攻略，水友们来餐厅路上的时间
                 */
                Thread.sleep((long) (Math.random() * 3000));
                System.out.println(Thread.currentThread().getName() + ", 到餐厅了");
                countDownLatch.countDown();   计数器减1
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, name: "水友" + i + "号").start();
    }

    countDownLatch.await();    阻塞等待

    System.out.println("都到齐了，那么服务员上菜！82年可乐先来一瓶！");
}
```

结果如下：



实现原理：内部又一个继承自 AQS 的 Sync 类，核心其实就是围绕一个整数 state。

初始化 state 的值，当调用一次 countDown 会把 state 的值减一，当 state 的值减到 0 的时候就会唤醒之前调用 await 等待的线程。

主要是依靠 AQS 封装的好，所以代码很少，原理也很清晰简单。

StampedLock

既然30题提到了这个，之前也专门写过文章分析，那刚好拿来讲讲。

可以认为是读写锁的“改进”版本。读写锁读写是互斥的，而 StampedLock 搞了个悲观读和乐观读，悲观读和写是互斥的，乐观读则不会。

搞个官方示例看下就很清晰了：

```java
class Point {
    private double x, y;
    private final StampedLock sl = new StampedLock();

```

```
    void move(double deltaX, double deltaY) { // an exclusively locked
method
        long stamp = sl.writeLock();  //获取写锁
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            sl.unlockWrite(stamp); //释放写锁
        }
    }

    double distanceFromOrigin() { // A read-only method
        long stamp = sl.tryOptimisticRead(); //乐观读
        double currentX = x, currentY = y;
        if (!sl.validate(stamp)) { //判断共享变量是否已经被其他线程写过
            stamp = sl.readLock();  //如果被写过则升级为悲观读锁
            try {
                currentX = x;
                currentY = y;
            } finally {
                sl.unlockRead(stamp); //释放悲观读锁
            }
        }
        return Math.sqrt(currentX * currentX + currentY * currentY);
    }

    void moveIfAtOrigin(double newX, double newY) { // upgrade
        // Could instead start with optimistic, not read mode
        long stamp = sl.readLock(); //获取读锁
        try {
            while (x == 0.0 && y == 0.0) {
                long ws = sl.tryConvertToWriteLock(stamp);  //升级为写锁
                if (ws != 0L) {
                    stamp = ws;
                    x = newX;
                    y = newY;
                    break;
                }
                else {
                    sl.unlockRead(stamp);
                    stamp = sl.writeLock();
                }
            }
        } finally {
            sl.unlock(stamp);
        }
    }
}
```

乐观锁就是获取判断一下，如果被修改了那么就升级为悲观锁。

但是 Semaphore 是不可重入锁，而且也不支持 condition 。并且如果线程使用writeLock() 或者 readLock() 获得锁之后，线程还没执行完就被 interrupt() 的话，会导致CPU飙升，需要用 `readLockInterruptibly` 或者 `writeLockInterruptibly`。

32.Java 中的阻塞队列用过哪些?

阻塞队列主要用来阻塞队列的插入和获取操作，当队列满了的时候阻塞队列的插入操作，直到队列有空位。当队列为空的时候阻塞队列的获取操作，直到队列有值。

常用在实现生产者和消费者场景，在笔试题中比较常见。

常见的有 ArrayBlockingQueue 和 LinkedBlockingQueue，分别是基于数组和链表的有界阻塞队列。

两者原理都是基于 ReentrantLock 和 Condition 。

都是有界阻塞队列两者有什么区别?

ArrayBlockingQueue 基于数组，内部实现只用了一把锁，可以指定公平或者非公平锁。

LinkedBlockingQueue 基于链表，内部实现用了两把锁，take 一把、put 一把，所以入队和出队这两个操作是可以并行的，从这里看并发度应该比 ArrayBlockingQueue 高。

还有 PriorityBlockingQueue 和 DelayQueue，分别是支持优先级的无界阻塞队列和支持延时获取的无界阻塞队列，如果你看过 DelayQueue 实现就会发现内部用的是 PriorityQueue。

```
public class DelayQueue<E extends Delayed> extends AbstractQueue<E>
    implements BlockingQueue<E> {

    private final transient ReentrantLock lock = new ReentrantLock();
    private final PriorityQueue<E> q = new PriorityQueue<E>();
```

还有 SynchronousQueue、 LinkedBlockingDeque 和 LinkedTransferQueue。

SynchronousQueue 前面线程池分析有提到过，它是不占空间的，入队比如等待一个出队，也就是生产者必须等待消费者拿货，无法把先把货存在队列。

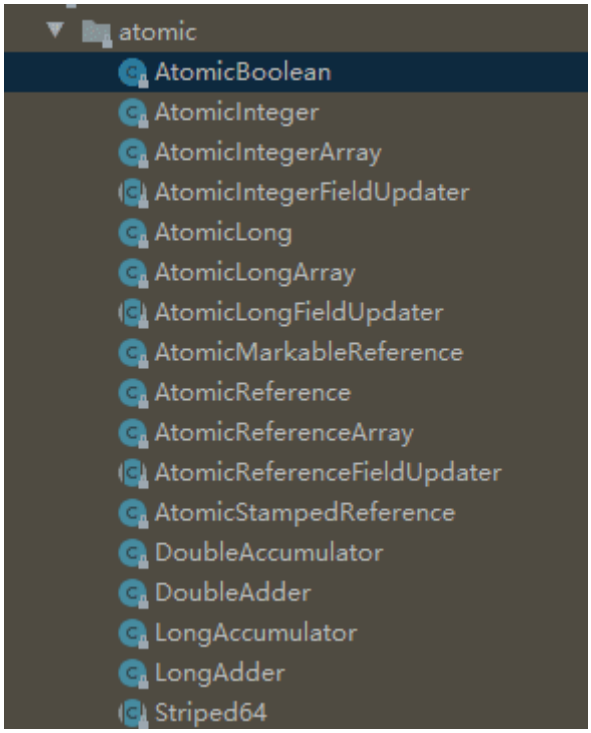
LinkedBlockingDeque 是双端阻塞无界队列，就是队列的头尾都能操作，头尾都能插入和移除。

LinkedTransferQueue，相对于其他阻塞队列从名字来看它有 Transfer 功能，其实也不是什么神奇功能，一般阻塞队列都是将元素入队，然后消费者从队列中获取元素。

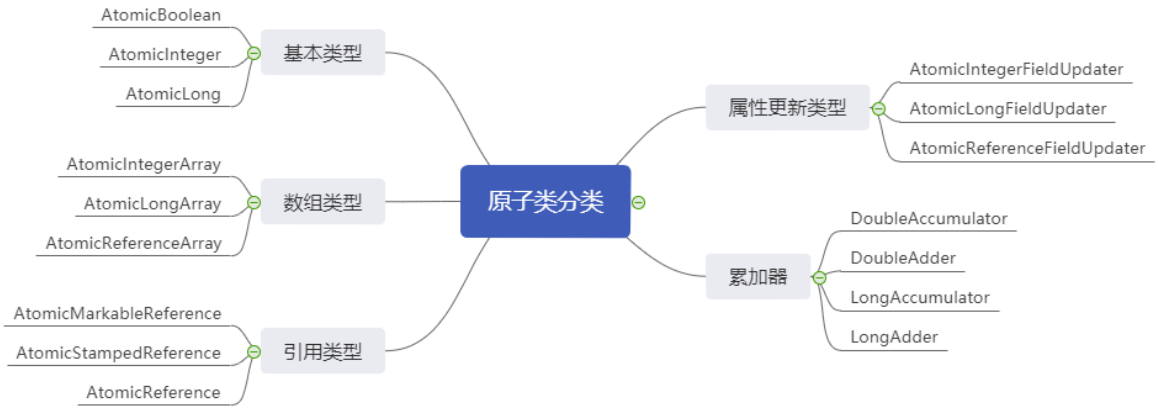
而 LinkedTransferQueue 的 transfer 是元素入队的时候看看是否已经有消费者在等了，如果有在等了直接给消费者即可，所以就是这里少了一层，没有锁操作。

33.用过Java 中哪些原子类?

原子类是 JUC 封装的通过无锁的方式实现的一系列线程安全的原子操作类。



上面截图的原子类主要分为五大类，我画个脑图汇总一下：



原子类的核心原理就是基于 CAS（Compare And Swap）。

CAS 简单的理解为：给予一个共享变量的内存地址，然后内存中应该的值(预期值)和新值，然后通过**一条 CPU 指令**来比较此内存地址上的值是否等于预期值，如果是则替换内存地址上的值为新值，如果不是则不予替换且换回。

也就是说硬件层面支持一条指令来实现这么几个操作，一条指令是不会被打断的，所以保证了原子性。

基本类型

可以简单的理解为通过基本类型原子类 AtomicBoolean、AtomicInteger 和 AtomicLong 就可线程安全地、原子地更新这几个基本类型。

数组类型

AtomicIntegerArray、AtomicLongArray 和 AtomicReferenceArray，简单的理解为可以原子化地更新数组内的每个元素，几个的差别无非就是数组里面存储的数据是什么类型。

引用类型

AtomicReference、AtomicStampedReference 和 AtomicMarkableReference，就是对象引用的原子化更新。

差别在于 AtomicStampedReference 和 AtomicMarkableReference 可以避免 CAS 的 ABA 问题。

AtomicStampedReference 是通过版本号 stamp 来避免， AtomicMarkableReference 是通过一个布尔值 mark 来避免。

ABA 问题

因为 CAS 是将期望值和当时内存地址上的值进行对比，假设期望值是 1 ，地址上的值现在是 1，只不过中间被人改成了 2 ，然后又改回了1，所以此时你 CAS 操作去对比是可以替换的，你无法得知中间值是否改过，这种情况就叫 ABA 问题。

而解决 ABA 问题的做法就是用版本号，每次修改版本就+1，这样即使值是一样的但是版本不同，就能得知之前被改过了。

属性更新类型

AtomicIntegerFieldUpdater、AtomicLongFieldUpdater 和 AtomicReferenceFieldUpdater，是通过反射，原子化的更新对象的属性，不过要求属性必须用 volatile 修饰来保证可见性，看下源码，很直观。

```
if (field.getType() != int.class)
    throw new IllegalArgumentException("Must be integer type");

if (!Modifier.isVolatile(modifiers))
    throw new IllegalArgumentException("Must be volatile type");
```

累加器

上述的都是更新数据，而 DoubleAccumulator、DoubleAdder、LongAccumulator 和 LongAdder 主要用来累加数据。

首先 AtomicLong 也能累加，而 LongAdder 是专业累加，也只能累加，并发度更高，它通过分多个 cells 来减少线程的竞争，提高了并发度。

你可以理解为如果拿 AtomicLong 是实现累加就是一本本子，然后 20 个人要让本子上累加计数。

而 LongAdder 分了 10 个本子，20个人可以分别拿这 10 个本子来计数(减少了竞争，提高了并发度)，然后最后的结果再由 10个本子上的数相加即可。

xxxAccumulator 和 xxxAdder 两者的区别?

xxxAccumulator 的功能比 xxxAdder 丰富，可以自定义累加方法，也可以设置初始值，按照注释上的解释 xxxAdder 等价于 new xxxAccumulator((x, y) -> x + y, 0L}。

所以可以说 xxxAdder 是 xxxAccumulator 的一个特例。

34.Synchronized 和 ReentrantLock 区别?

Synchronized 和 ReentrantLock 都是可重入锁，ReentrantLock 需要手动解锁，而 Synchronized 不需要。

ReentrantLock 支持设置超时时间，可以避免死锁，比较灵活，并且支持公平锁，可中断，支持条件判断。

Synchronized 不支持超时，非公平，不可中断，不支持条件。

总的而言，一般情况下用 Synchronized 足矣，比较简单，而 ReentrantLock 比较灵活，支持的功能比较多，所以复杂的情况用 ReentrantLock 。

至于说 Synchronized 性能不如 ReentrantLock 的，那都是 N 多年前的事儿了。

35.Synchronized 原理知道不?

Synchronized 的原理其实就是基于一个锁对象和锁对象相关联的一个 monitor 对象。

在偏向锁和轻量级锁的时候只需要利用 CAS 来操控锁对象头即可完成加解锁动作。

在升级为重量级锁之后还需要利用 monitor 对象，利用 CAS 和 mutex 来作为底层实现。

monitor 对象颞部会有等待队列和条件等待队列，未竞争到锁的线程存储到等待队列中，获得锁的线程调用 wait 后便存放在条件等待队列中，解锁和 notify 都会唤醒相应队列中的等待线程来争抢锁。

然后由于阻塞和唤醒依赖于底层的操作系统实现，系统调用存在用户态与内核态之间的切换，所以有较高的开销，**因此称之为重量级锁。**

所以才会有偏向锁和轻量级锁的优化，并且引入自适应自旋机制，来提高锁的性能。

关于 Synchronized 其实我写过两篇文章，看完这两篇文章你可以跟 Synchronized 说，你看过 JVM 源码，毫不夸张，因为就是从源码级别上分析的。

而且指明了一个几乎网上都错了的观点和一个常见的认知错误。

总而言之，看完之后对 Synchronized 基本上超越很多人了。

Synchronized 深入JVM分析

Synchronized 升级到重量级锁之后就下不来了？你错了！

36.ReentrantLock 的原理?

ReentrantLock 其实就是基于 AQS 实现的一个可重入锁，支持公平和非公平两种方式。

内部实现其实就是依靠一个 state 变量和两个等待队列：同步队列和等待队列。

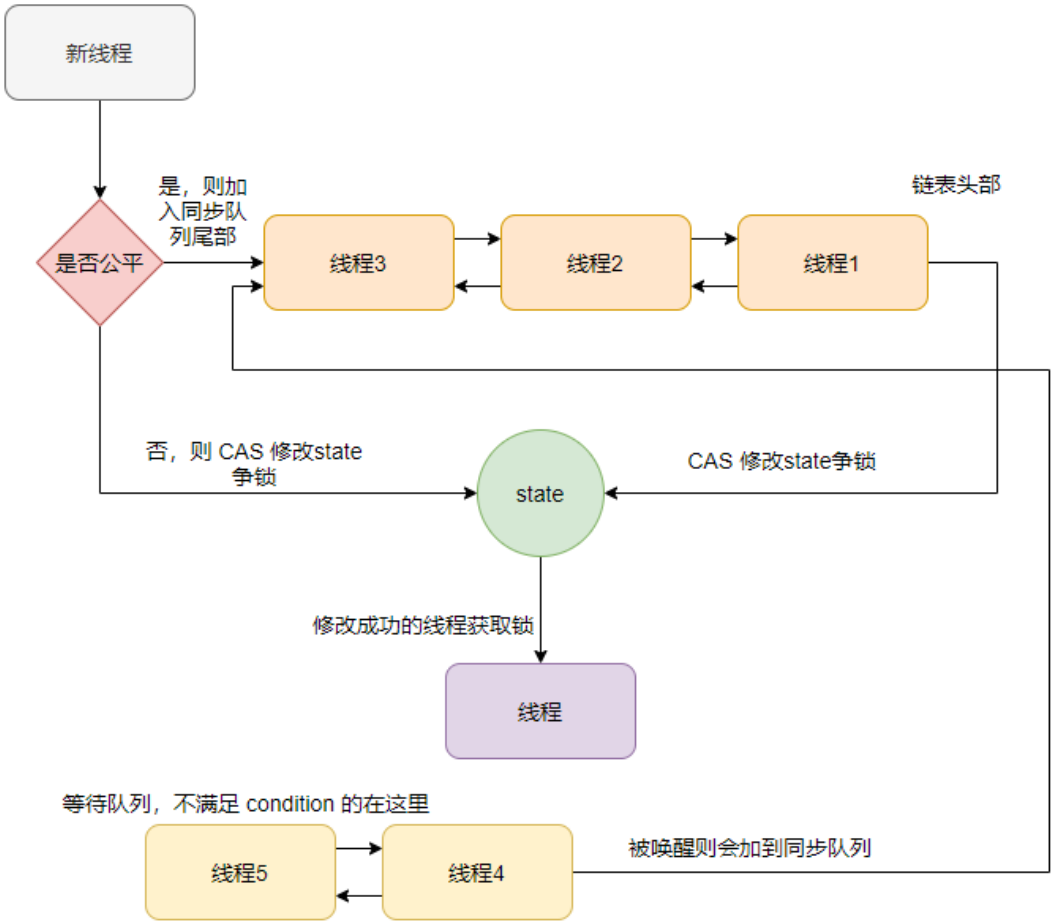
利用 CAS 修改 state 来争抢锁。

争抢不到则入同步队列等待，同步队列是一个双向链表。

条件 condition 不满足时候则入等待队列等待，也是个双向链表。

是否是公平锁的区别在于：线程获取锁时是加入到同步队列尾部还是直接利用 CAS 争抢锁。

就是这么回事儿，理解起来应该不难，操心的我再画个图，嘿嘿。



37.说说 AQS 吧?

AQS 的原理其实就是上面提到的，这里就不再赘述了。

如果面试官问你为什么需要 AQS ，就这样回答。

AQS 将一些操作封装起来，比如入队等基本方法，暴露出方法，便于其他相关 JUC 锁的使用。

比如 CountDownLatch、Semaphore 等等。

就是起到了一个抽象，封装的作用。

38.读写锁知道不?

读写锁在 Java 中一般默认指的是 ReentrantReadWriteLock。

读写锁是有两把锁，分别是读锁和写锁。

除了读读操作不互斥之外，其他都互斥。

所以读很多写比较少的情况，用读写锁比较合适。

还有一点要注意，如果不是这种情况不要用读写锁，**因为读写锁需要额外维护读锁的状态，所以如果读读操作不多还不如一般的锁。**

读写锁也是基于 AQS 实现的，再具体点的实现就是将 state分为了两部分，高16bit用于标识读状态、低16bit标识写状态。

就这样灵巧的通过一个 state 实现了两把锁，嘿嘿。

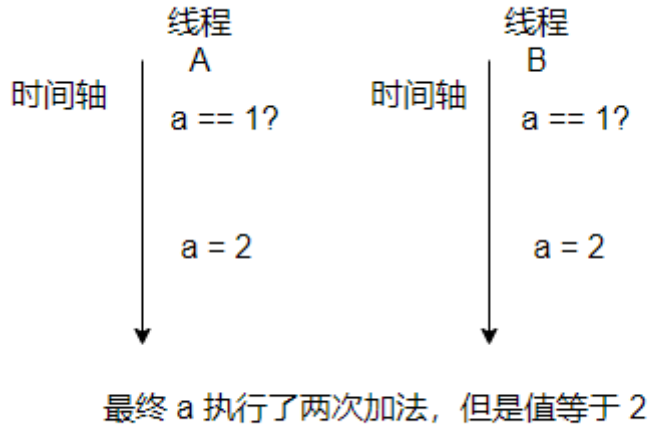
39.CAS 知道不?

CAS 就是 compare and swap，即比较并交换。

举个例子，我们经常有累加需求，比较一个值是否等于 1，如果等于 1 我们将它替换成 2，如果等于 2 替换成 3。

这种比较在多线程的情况下就不安全，比如此时同时有两个线程执行到比较值是否等于 1，然后两个线程发现都等于 1。

然后两个线程都将它变成了 2，这样明明加了两次，值却等于 2。



这种情况其实加锁可以解决，但是加锁是比较消耗资源的。

因此硬件层面就给予支持，将这个比较和交换的动作封装成一个指令，这样就保证了原子性，不会判断值确实等于 1，但是替换的时候值以及不等于 1了。

这指令就是 CAS。

CAS 需要三个操作数，分别是旧的预期值(图中的1)，变量内存地址(图中a的内存地址)，新值(图中的2)。

指令是根据变量地址拿到值，比较是否和预期值相等，如果是的话则替换成新值，如果不是则不替换。

其实 33 题已经提过这个了，包括 ABA 问题，之所以再写一下是可以从我提供的思路跟面试官说。

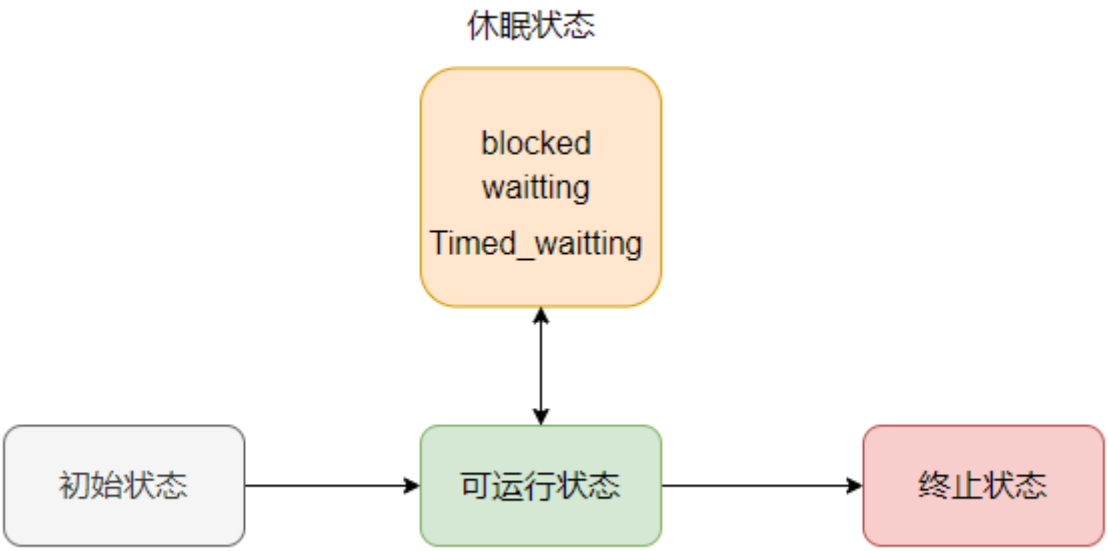
不要一上来就三个操作数。

你把遇到的场景(上面说的累加)，然后多线程不安全，然后用锁不好，然后硬件提供了这个指令。

按这样的思路说出来，如果我是面试官，我一听就会觉得，小伙子可以。

40.说说线程的生命周期?

可以分为初始状态、可运行状态、终止状态和休眠状态四大类。



线程新建的时候就是初始状态，还未start。

可运行状态就是可以运行，可能正在运行，也可能正在等 CPU 时间片。

休眠状态分为三种，一种是等待锁的 blocked 状态，一种是等待条件的 waitting 状态，或者有时间限制的等待 timed_waitting 状态。

- 等待条件的操作有：Object.wait、Thread.join、LockSupport.park()
- 时间等待就是上面设置了timeout参数的方法，例如Object.wait(1000)。

终止状态就是线程结束执行了，可以是结束任务后的自动结束，也可以是产生了异常而结束。

41.什么是 JMM？

JMM 即 Java Memory Model，Java 内存模型。

JMM 其实是一组规则，规定了一个线程的写操作何时会对另一个线程可见(JSR133)。

抽象的来看 JMM 会把内存分为本地内存和主存，每个线程都有自己的私有化的本地内存，然后还有个存储共享数据的主存。

由 JMM 来定义这两个内存之间的交互规则。

这里要注意本地内存只是一种抽象的说法，实际指代：寄存器、CPU 缓存等等。

总之 JMM 屏蔽了各大底层硬件的细节，是抽象出来的一套虚拟机层面的内存规范。

42.说说原子性、可见性、有序性?

原子性

指的是一个操作不会被中断，要么这个操作执行完毕，要么不会执行，不会有执行一半的存在。

可见性

指的是一个线程对某个共享变量进行了修改，则其他线程能立刻获取到最新值。

有序性

指的是编译器或者处理器会将指令进行重排，这种操作会影响多线程的执行顺序导致错误。

43.说说 Java 常见的垃圾收集器?

这题我不太想写答案，内容比较死板，建议还是看《深入理解JVM虚拟机》吧，不过那本书里面没有详细说 ZGC。

而 ZGC 我倒是写了一篇，可以看看呐。

美团面试官问我：ZGC 的 Z 是什么意思?

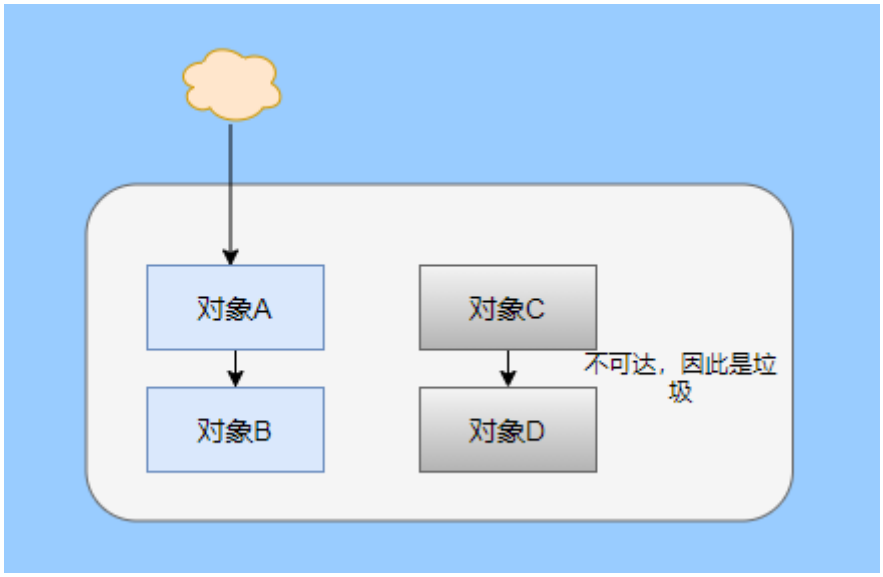
44.垃圾回收，如何判断对象是否是垃圾?

一共有两种方式，分别是引用计数和可达性分析。

引用计数有循环依赖的问题，但是是可以解决的。

可达性分析则是从根引用（GCRoots） 开始进行引用链遍历扫描，如果可达则对象存活，如果不可达则对象已成为垃圾。

所谓的根引用包括全局变量、栈上引用、寄存器上的等。



我之前也写过，详细的看这篇文章吧，看完之后这一块针对面试绝对没问题，而且已经超越了很多面试官。

深度揭秘垃圾回收底层，这次让你彻底弄懂她

45.你知道有哪些垃圾回收算法?

常见的就是：复制、标记-清除、标记整理。

标记-清除

标记-清除算法应该是最符合我们人一开始处理垃圾的思路的算法。

例如我们想清除房间的垃圾，我们肯定是先定位(对应标记)哪些是垃圾，然后把这些垃圾之后扔了(对应清除)，简单粗暴，剩下的不是垃圾的东西我也懒得理，不管了哈哈哈。

但是，这算法有个缺点：

1. 空间碎片问题，这样会使得比较大的对象要申请比较多的连续空间的时候申请不到，明明你空间还很足的。然后导致又一次GC。

复制算法

复制算法一般用于新生代，粗暴的复制算法就是把空间一分为二，然后将一边存活的对象复制到另一边，这样没有空间碎片问题，但是内存利用率太低了，只有 50%，所以 HotSpot 中是把一块空间分为 3 块，一块Eden,两块Survivor。

因为正常情况下新生代的大部分对象都是短命鬼，所以能活下来的不多，所以默认的空间划分比例是 8:1:1。

用法就是每次只使用Eden和一块Survivor,然后把活下来的对象都扔到另一块Survivor。再清理Eden和之前的那块Survivor。然后再把Eden和存放存活对象的那一块Survivor用来迎接新的对象。就等于每次回收了之后都会对调一下两个Survivor。

标记-整理算法

标记-整理算法的思路也是和标记-清除算法一样，先标记那些需要清除的对象，但是后续步骤不一样，它是整理，对就是像上面说的那些清除房间垃圾每次都会整理的人一样那么勤劳。

每次会移动所有存活的对象，且按照内存地址次序依次排列，也就是把活着的对象都像一端移动，然后将末端内存地址以后的内存全部回收。所以用了它也就没有空间碎片的问题了。

来吧，之前写的，看完之后差不多了。

炸了！一口气问了我18个JVM问题！

46.String，Stringbuffer，StringBuilder的区别?

String 是 Java 中基础且重要的类，并且 String 也是 Immutable 类的典型实现，被声明为 final class，除了 hash 这个属性其它属性都声明为 final。

因为它的不可变性，所以例如拼接字符串时候会产生很多无用的中间对象，如果**频繁的进行这样的操作**对性能有所影响。

StringBuffer 就是为了解决大量拼接字符串时产生很多中间对象问题而提供的一个类，提供 append 和 add 方法，可以将字符串添加到已有序列的末尾或指定位置。

它的本质是一个线程安全的可修改的字符序列，把所有修改数据的方法都加上了 synchronized。但是保证了线程安全是需要性能的代价的。

在很多情况下我们的字符串拼接操作不需要线程安全，这时候**StringBuilder**登场了，StringBuilder是JDK1.5发布的，它和StringBuffer 本质上没什么区别，就是**去掉了保证线程安全的那部分，减少了开销。**

StringBuffer 和 StringBuilder 二者都继承了 AbstractStringBuilder ，底层都是利用可修改的 char 数组(JDK 9 以后是 byte 数组)。

所以如果我们有大量的字符串拼接，如果能预知大小的话最好在new StringBuffer 或者StringBuilder 的时候设置好 capacity，避免多次扩容的开销。

扩容要抛弃原有数组，还要进行数组拷贝创建新的数组。

47.happens-before 听过吗?

这个问题估计应该都是来自《深入理解Java虚拟机》这本书的。

happens-before 就是定义的一些规则，在一些特定场景下，一些操作会先行发生于另一些操作。

A先行发生于B，其实含义就是 A 操作得到的结果在 B 操作开始时可以得到，重点不在于 A 执行的时间比 B 早，而是 A 的结果是可以在 B 开始时候被 B 读取的。

这是 JVM 规定的有序性，你也可以认为写 JVM 的程序员需要按照这样的规则来实现 JVM。

如操作符合以下的规则就会按照下面的定义动作先行发生。

- 程序次序规则：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确地说，应该是控制流顺序而不是程序代码顺序，因为要考虑分支、循环等结构。
- 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。这里必须强调的是同一个锁，而“后面”是指时间上的先后顺序。
- 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，这里的“后面”同样是指时间上的先后顺序。
- 传递性规则：如果操作A先行发生于操作B，操作B先行发生于操作C，那就可以得出操作A先行发生于操作C的结论。
- 线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作。
- 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupted()方法检测到是否有中断发生。
- 线程终止规则：线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值等手段检测到线程已经终止执行。
- 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。

48.什么是锁的自适应自旋?

这里指的就是 Syncronized 在身为重量级锁时候的自旋。

具体指的是在重量级锁时，一个线程如果竞争锁失败会进行自旋操作，说白了就是执行一些无意义的执行，空转 CPU 等着锁的释放。

因为一些情况下可能线程刚被阻塞，锁就被释放了，这样开销就比较大，所以自旋在一定程度上是有优化的。

形象一点就像怠速停车和熄火的区别，如果等待时候很长(长时候都拿不到锁)，那肯定熄火划算(阻塞)。

如果一会儿就要出发(拿到锁)，那怠速停车(自旋)比较划算。

不过因为这个自旋次数不好判断，所以**引入自适应自旋**。

说白了就是结合经验值来看，如果上次自旋一会儿就拿到锁，那这次多自旋几次，如果上次自旋很久都拿不到，这次就少自旋。

这就叫锁的自适应自旋。

49.JVM 内存区域划分

Java 虚拟机运行时数据区分为程序计数器、虚拟机栈、本地方法栈、堆、方法区。

程序计数器、虚拟机栈、本地方法栈这 3 个区域是线程私有的，会随线程消亡而自动回收，所以不需要管理。

而堆和方法区是线程共享的，所以垃圾回收器会关注这两个地方。

堆只要存放的就是平时 new 的对象。

方法区存放的就是加载的类型信息、即时编译(JIT)后的代码等。

50.指令重排知道吗?

为了提高程序执行的效率，CPU或者编译器就将执行命令重排序。

原因是因为内存访问的速度比 CPU 运行速度慢很多，因此需要编排一下执行的顺序，防止因为访问内存的比较慢的指令而使得 CPU 闲置着。

CPU 执行有个指令流水线的概念，还有分支预测等，关于这个我之前写过一篇文章，可以看下CPU分支预测

总之为了提高效率就会有指令重排的情况，导致指令乱序执行的情况发生，不过会保证结果肯定是与单线程执行结果一致的，这叫 **as-if-serial**。

不过多线程就无法保证了，在 Java 中的 volatile 关键字可以禁止修饰变量前后的指令重排。

51.final 可以保证可见性吗?

不可以。

你可能看到一些答案说可以保证可见性，**那不是我们常说的可见性。**

一般而言我们指的可见性是一个线程修改了共享变量，另一个线程可以立马得知更改，得到最新修改后的值。

而 final 并不能保证这种情况的发生，volatile 才可以。

而有些答案提到的 final 可以保证可见性，其实指的是 final 修饰的字段在构造方法初始化完成，并且期间没有把 this 传递出去，那么当构造器执行完毕之后，其他线程就能看见 final 字段的值。

如果不用 final 修饰的话，那么有可能在构造函数里面对字段的写操作被排序到外部，这样别的线程就拿不到写操作之后的值。

来看个代码就比较清晰了。

```java
public class YesFinalTest {
   final int a;
   int b;
   static YesFinalTest testObj;

   public void YesFinalTest () { //对字段赋值
       a = 1;
       b = 2;
   }

   public static void newTestObj () {  // 此时线程 A 调用这个方法
       testObj = new YesFinalTest ();
   }

   public static void getTestObj () {  // 此时线程 B 执行这个方法
       YesFinalTest object = obj;
       int a = object.a; //这里读到的肯定是 1
       int b = object.b; //这里读到的可能是 2
   }
}
```

对于 final 域，编译器和处理器要遵守两个重排序规则(参考自infoq程晓明)：

1. 在构造函数内对一个 final 域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。初次读一个包含
2. final 域的对象的引用，与随后初次读这个 final 域，这两个操作之间不能重排序。

所以这才是 final 的可见性，这种可见性和我们在并发中常说的可见性不是一个概念！

所以 final 无法保证可见性！

52.锁如何优化?

要注意锁的粒度，不能粗暴的直接在方法外围定义锁，锁的代码块越小越好，像双检锁就是典型的优化。

不同场景定义不同的锁，不能粗暴的一把锁搞定，例如在读多写少的场景可以使用读写锁、写时复制等。

再具体的可以看看我之前写的 Java “锁” 事 。