## 3-1.Java的Char是两个字节，如何存UTF-8字符

### 是否熟悉Java Char和字符串

Char是2个字节，存的方式是UTF-16

Unicode通用字符集占2个字节

Unicode扩展字符集用一对Char，占4个字节

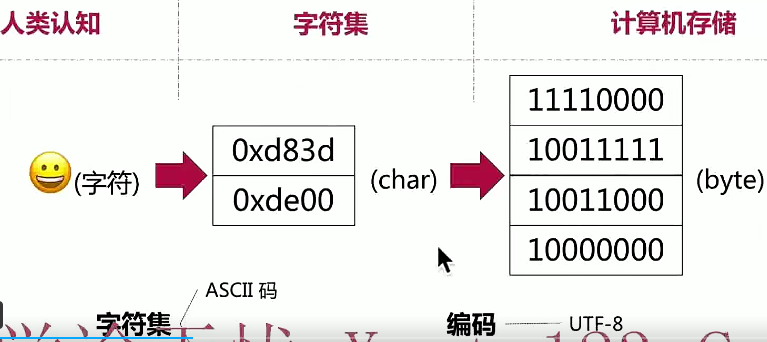
String的length不是字符数，是Char[]的个数

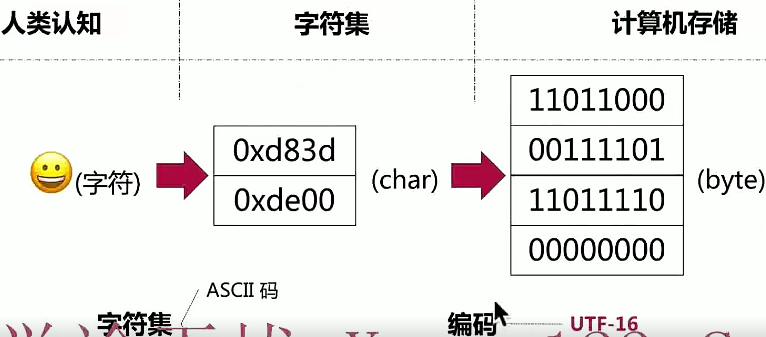
UTF-8是最小单位是1个字节，可变的1-3

UTF-16最小单位是2个字节

Java9 做了优化，如果String里面全部是拉丁字母（占1个字节），则改为byte存储。优化空间

### 是否了解字符的映射和存储细节





可以看见表情emoji是用2个char来表示。

1个char对应2个字节。所以这里用了4个字节来存储！

UTF-8和UTF-16只是编解码的方式不一样，不影响存储空间！！！

### 是否横向对比其他语言

## 3-2.Java String可以有多少长

### 是否对字符串编码有深入了解

string.length()取的是Char[]数组长度

string.getByte().length取得是字节长度

String内部是char[]数组

UTF-8是可变长度的字符集，长度在1-3个

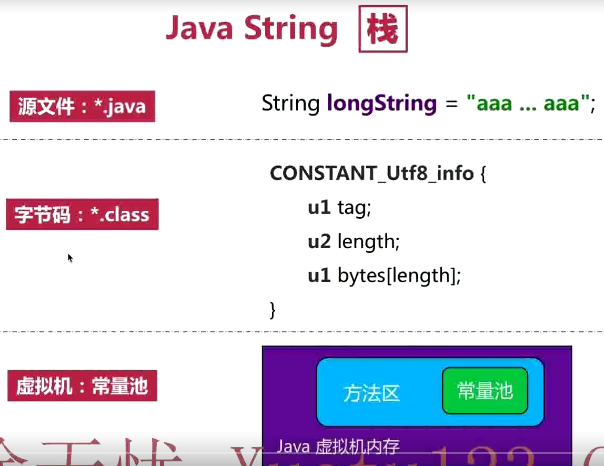
### 是否对字符串在内存当中的存储形式有深入了解

String中char数组

有些imoji表情采用2个Char来存储！

### 是否对java虚拟机字节码有足够了解

栈：在字节码中是按照UTF-8来存的，可以看见最多65535个字节



### 是否对java虚拟机指令有一定的认识

Java虚拟机指令newarray的限制

Java虚拟机堆内存大小的限制

堆：理论最大是Integer.MAX\_VALUE value[]

## 3-3.Java的匿名内部类有哪些限制？

### 匿名内部类的概念和用法

匿名内部类名字：外部类$1（顺序）

非静态匿名内部类的构造方法中，会传入外部类！所以会持有外部类的引用

静态内部类，不持有引用

### 语言规范以及语言横向对比

### 内存泄漏切入点

## 3-4.怎么理解Java的方法分派

确定调用谁的，哪个方法

### 多态，虚方法表的认识

当子类重写父类方法的时候，Java里面默认都是虚方法

C++需要加Virtual关键字

### 对编译时和运行时的理解和认识

编译后，字节码会根据左侧类型，匹配方法签名，确定方法走向

运行时，根据实际类型，来返回相应的值

### 对Java语言规范和运行机制的深入认识

### 横向对比各类语言的能力

Groovy是DSL动态语言，所以无论编译还是运行，都是走实际方法

C++和Java一样，方法走向，都是编译走左侧

C++的运行结果分各种有趣的情况，需要明确写出virtual方法

C++假如创建对象是

1. 用构造方法直接创建（创建在栈，子类对象会被裁剪！！）

对象只要被裁剪了。不管方法怎么接收，都是只调用父类方法。因为子类已经被裁剪了

1. 用new方法创建（创建在堆，需要delete手动释放）

栈：SuperClass superClassPtr = SubClass();

堆：SuperClass \*superClassPtr = new SubClass();

method(SuperClass superClass){}

method(SuperClass\* superClass){}

method(SuperClass& superClass){}

C++对于堆上的对象。如果方法接收是

1. 值：method(\*superClassPtr)：传参时，子类会拷贝一份，然后被裁掉，所以是父类的值
2. 指针：method(superClassPtr)：虚函数，就是实际的子类值
3. 引用：method(\*superClassPtr)：引用，不拷贝，直接指向子类，所以就是实际的子类值

## 3-5.Java泛型实现机制是怎么的？

### 对Java泛型的实现机制的认知和理解

泛型编译时被擦除被成Object，不兼容基本类型

泛型类型签名信息在特定场景下反射获得

泛型擦除特点：

1. 内存负担小(泛型擦除后，都是长一样)
2. 兼容性好(为了兼容1.5以前)
3. 基本类型无法作为泛型实参（本质上被替换成Object，基本类型不是Object）
4. 无法方法重载(泛型擦除后，都是长一样)
5. 不能new T()，不能new T[0]，不能T.class，不能 list instanceof List<String>

Android对于key是int，使用spareArray

### 对泛型编程是否有深入的对比研究

Java因为考虑到兼容。所以只能使用泛型擦除

kotlin通过反射注解metadata来获取方法签名

-keep class kotlin.Metadata{\*;}

### 对常见框架原理是否有过深入剖析

Gson中，使用TypeToken来获取泛型里面的类型！

通过反射拿方法签名上的泛型类型

可以通过Class.getGenericSuperclass拿到ParameterizedType

再getActualTypeArguments()拿到泛型类型

getMethod().getGenericReturnType 拿到ParameterizedType

使用了签名要注意混淆

-keepattributes Signature

## 3-6.Activity中的onActivityResult使用起来很麻烦，为什么不设计成回调？

### 是否熟悉onActivityResult的用法？

ActivityA.startActivityForResult() 启动 ActivityB

ActivityB.setResult() 并执行finish()

ActivityA会触发onActivityResult

### 是否思考过用回调替代onActivityResult

大多数可以使用回调替代，但是当内存不足的时候，ActivityA会被回收！则无法使用回调

### 是否意识到回调存在问题

1. 主要是当内存不足的时候，ActivityA会被回收！所以会回调给一个无效的ActivityA！
2. 回调本职是匿名内部类。如果在回调内部，使用了mTextView.setText()类似方法。实际上会持有外部类ActivityA的引用！而销毁ActivityA的时候，ActivityA是不希望被引用

### 是否能给出匿名内部类对外部引用的解决方案

通过替换匿名内部类的外部引用实现回调！（本质就是更新回调方法内部的引用）

1. 可以思考利用Fragment中的onActivityResult来代替Activity中的onActivityResult。
2. 在Activity恢复的时候，利用fragment中的mWho来恢复唯一的fragment
3. 如果用view的id来恢复，需要注意id需全局唯一

## 4-1.如何停止一个线程？

### 对线程的用法

线程只能等方法全部执行完！

虽然线程不能简单终止，但是内部的任务可以终止

主要采用boolean标志位，和Interrupt

### 对线程stop的方法

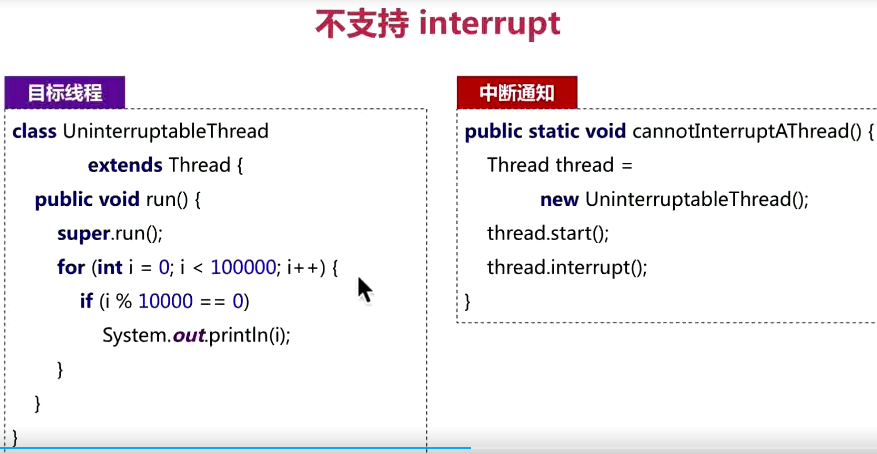
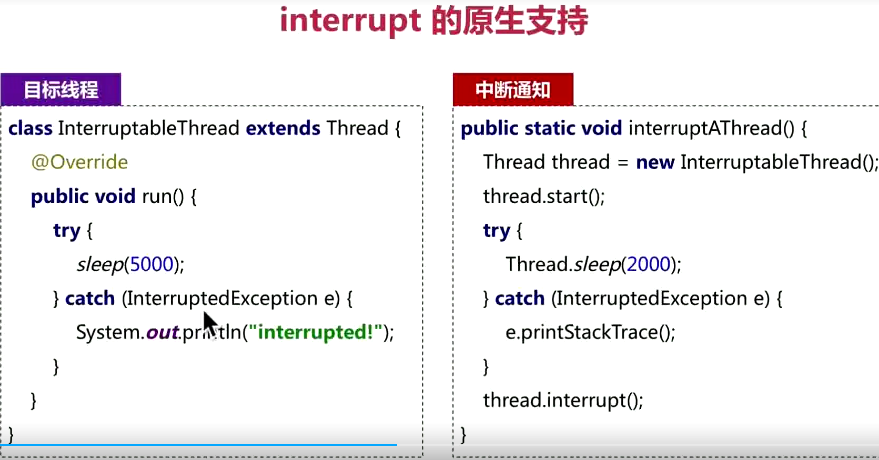
stop方法被废弃掉

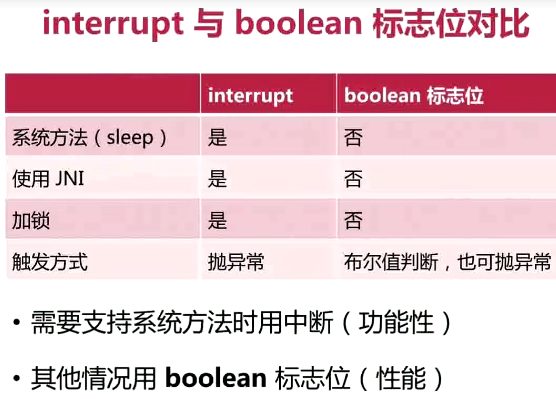
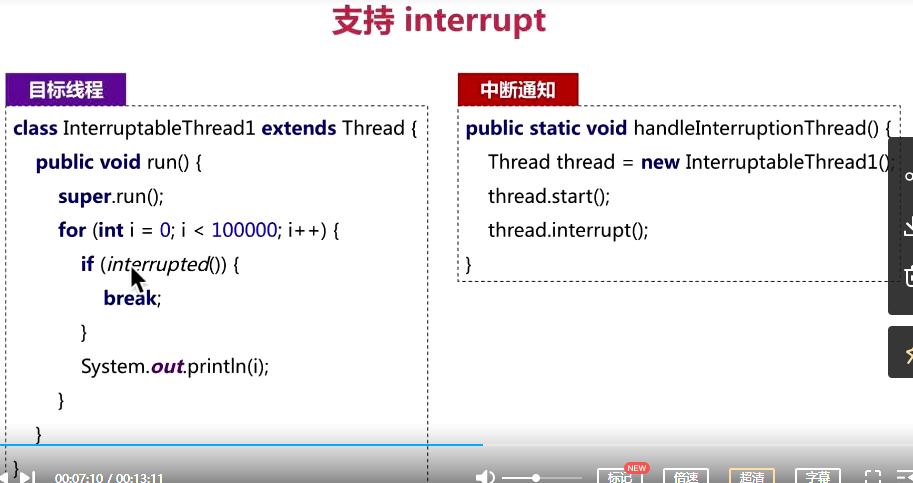
### 对线程stop过程存在问题

不能简单停止一个线程，涉及资源清理

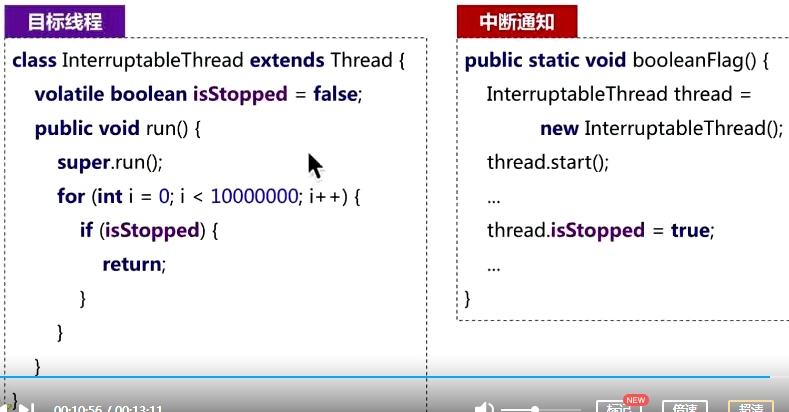
1. 因为在共享内存中，Thread-1持有锁。Thread-3等待。Thread-1关闭，不释放锁，Thread-3会一直等待。
2. 如果Thread-1立即释放锁。虽然Thread-3可以获得锁继续操作，但是如果操作区域是和Thread-1共享内容，也会出现内存异常

### interrupt中断的用法



### 解释使用boolean标志位的好处



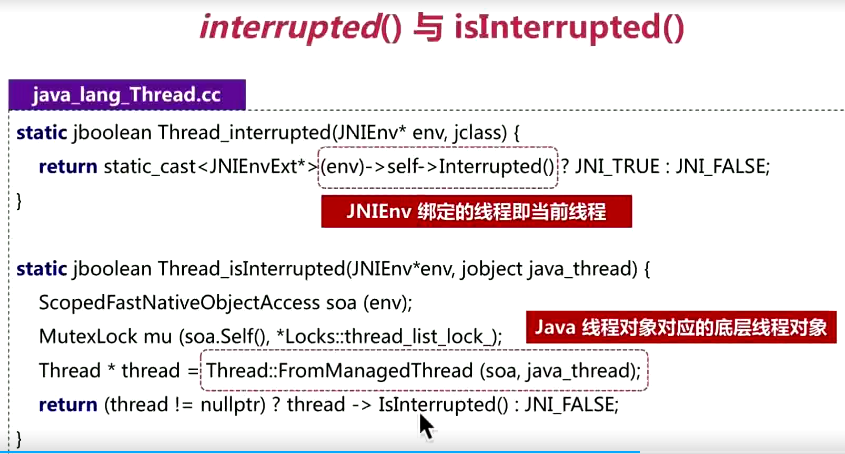
### 是否知道interrupt底层细节

interrupted()是静态方法，获取当前线程的中断状态，并清空

中断前，始终是false，中断并清空后，首次返回调用true，重复调用返回false

isInterrupted()是非静态方法，获取当前线程的中断状态，不清空

中断前，始终是false，中断并清空前，一直是true，清空后，返回false



### 线程安全

## 4-2.如何写出线程安全的程序

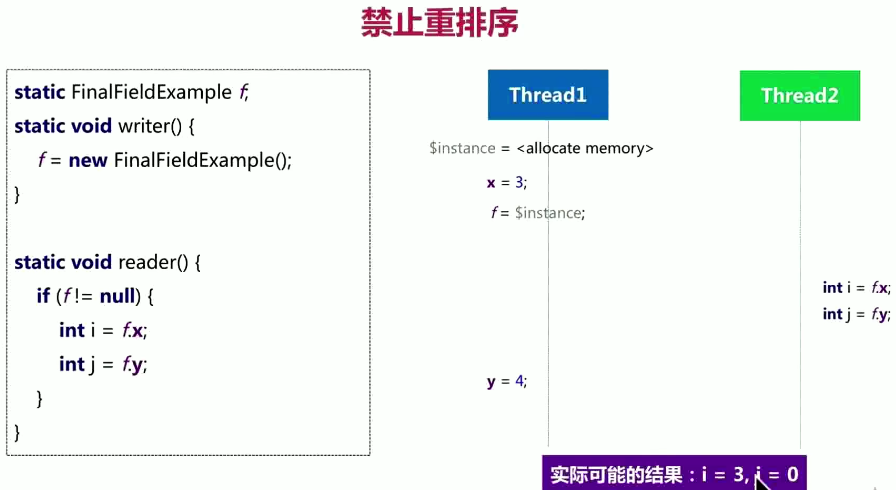
### 是否对线程安全有初步了解

1. 不共享资源
2. 共享不可变资源
3. 共享可变资源（这里才需要做特殊处理）

### 是否知道线程安全产生的原因

可变资源内存线程间共享

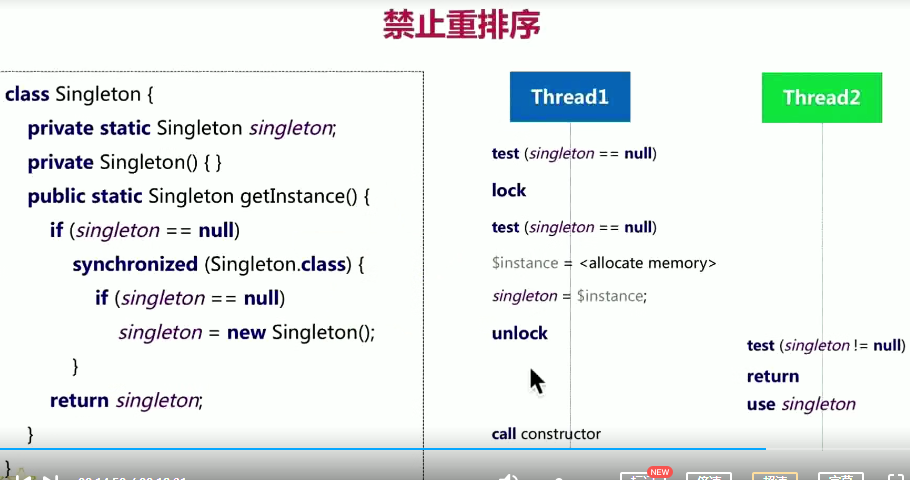
### 是否知道final，volatile关键字的作用



final在线程安全的作用，禁用重排序！！

final 可以确保变量在构造方法之前赋值！对于有些CPU架构，会报非final赋值放到构造方法之后！

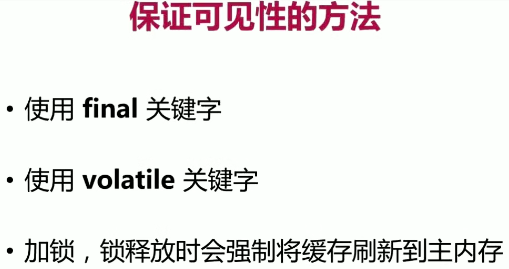
### 是否知道1.5之前Java DCL为什么有缺陷



单例中，写双重检测，一定要写volatile，也是禁用重排序问题，而造成构造方法异常！

### 是否清楚知道如何编写线程安全的程序

#### 可见性：

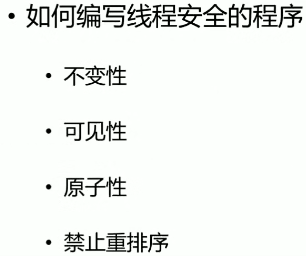


加锁后，在锁释放才能强制将缓存刷新到主内存

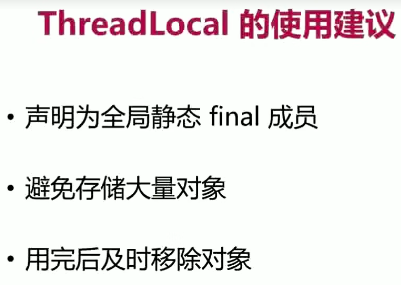
#### 原子性：



#### 总结：



### 是否对ThreadLocal的使用注意事项有认识



## 4-3.ConcurrentHashMap如何支持并发访问？

### 是否熟练掌握线程安全的概念

CHM是ConcurrentHashMap。

### 是否深入理解CHM的各项并发优化的原理



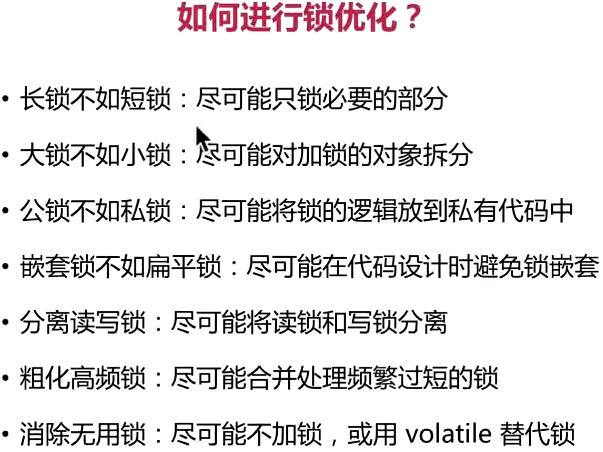
JDK 5：会造成某一个分段内容巨增

JDK 6：专门处理了分段巨增问题，让每一个分段都平均获取

JDK 7：之前会一次性生成16个分段。现在是懒加载分段，大量使用volatile

JDK 8：不用分段了。直接加锁到table

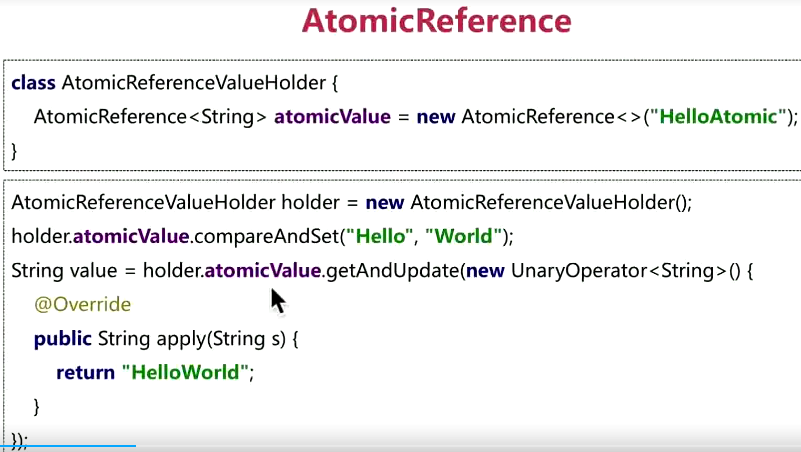
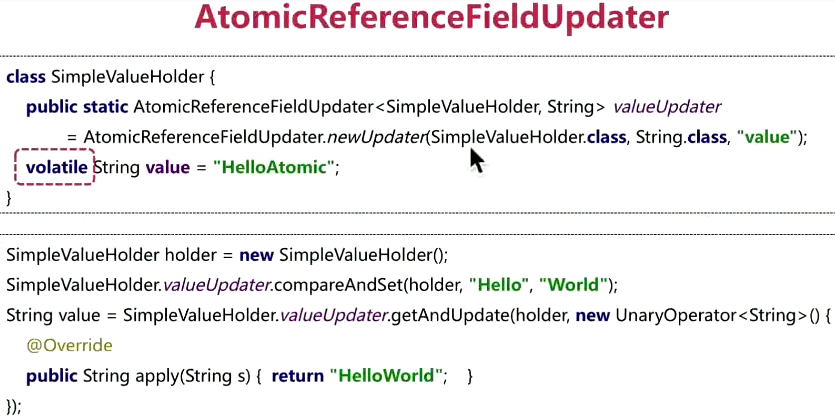
### 是否掌握锁优化的方法



## 4-4.AtomicReference和AtomicReferenceFieldUpdater有何异同？

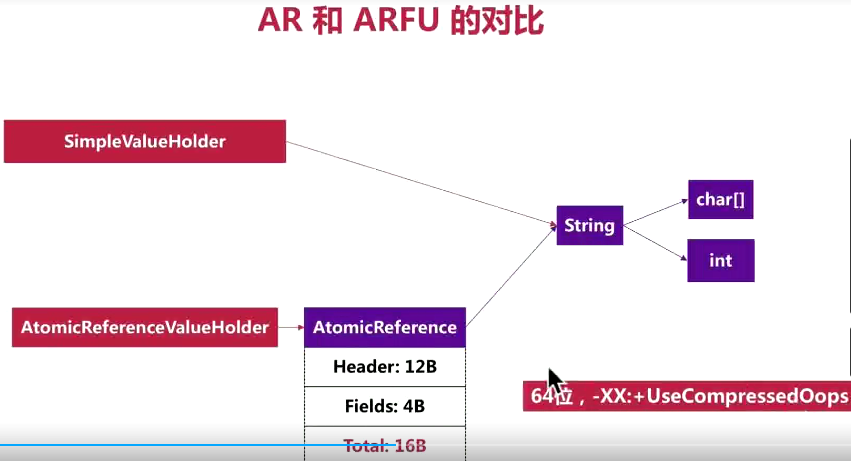
### 是否熟练掌握原子操作的感念

### 是否熟悉AR和ARFU这两个类的用法和原理

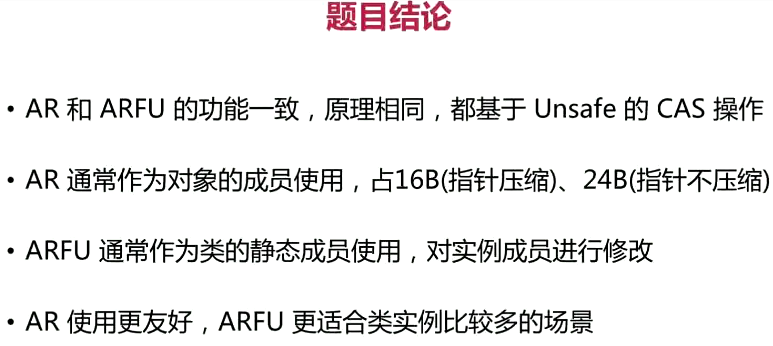
 

ARFU用的比较多。因为采用反射，不生成对象，节省空间

AR是对象，会占用一定的空间



### 是否对Java对象内存占用有认识



## 4-5如果在android中写出优雅的异步代码？

### 是否熟练编写异步和同步代码

### 是否熟悉回调地狱

### 是否熟练使用Rxjava

### 是否对kotlin协程有了解

### 是否具备良好的代码意识和能力