<https://cs.android.com/>

**方法重写原则：**

1.被重写的方法为void方法时，重写的方法必须也是void方法

2.被重写的方法返回值类型为基本数据类型时，重写的方法返回值与被重写的方法相同

3被重写的方法返回值类型为引用类型时，重写的方法返回值类型必须为被重写方法返回值类型或其子类

4.重写的方法不能声明抛出比被重写方法范围更大的异常类型

5.重写的方法不能使用比被重写的方法更严格的访问权限。

**方法调用绑定（摘自Java编程思想）：**

1. 将一个方法调用同一个方法主体关联起来被称作绑定

2. 静态绑定：程序执行前进行绑定（方法执行以声明类型为准，static方法除外）

3. 动态绑定：运行时根据对象实际类型进行绑定

4. 编译器只知道对象的声明类型，而不知道对象的实际类型，方法调用机制能够找到正确的方法体，并加以调用

5. static 、final 、private方法为静态绑定，其他均为动态绑定

**Java运行时内存分布**

**・程序计数器**（Program Counter Register）：线程私有，是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。 在Java虚拟机的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，它是程序控制流的指示器，分支、循环、跳转、异常处  
理、线程恢复等基础功能都需要依赖这个计数器来完成

如果线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是本地（Native） 方法， 这个计数器值则应为空（Undefined）。此内存区域是唯一一个在《Java虚拟机规范》中没有规定任何OutOfMemoryError情况的区域

**・Java虚拟机栈：**线程私有，生命周期与线程相同。虚拟机栈描述的是Java方法执行的线程内存模型：每个方法被执行的时候， Java虚拟机都会同步创建一个栈帧[1]（Stack Frame） 用于存储局部变量表、 操作数栈、 动态连接、 方法出口等信息。 每一个方法被调用直至执行完毕的过程， 就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程

**局部变量表**：是一组变量值的存储空间，用于存放方法参数和方法内部定义的局部变量。 在Java程序被编译为Class文件时，就在方法的Code属性的max\_locals数据项中确定了该方法所需分配的局部变量表的最大容量。

局部变量表的容量以变量槽（Variable Slot）为最小单位，64位长度的long和double类型的数据占

用两个变量槽， 其余的数据类型只占用一个。

局部变量表存放了编译期可知的各种Java虚拟机基本数据类型（boolean、 byte、 char、 short、 int、

float、 long、 double） 、 对象引用（reference类型， 它并不等同于对象本身， 可能是一个指向

对象起始地址的引用指针， 也可能是指向一个代表对象的句柄或者其他与此对象相关的位置）和

returnAddress类型（指向了一条字节码指令的地址）

**操作数栈（Operand Stack**）：也常被称为操作栈， 它是一个后入先出（Last In First Out， LIFO）  
栈。 同局部变量表一样， 操作数栈的最大深度也在编译的时候被写入到Code属性的max\_stacks数据项之中。 操作数栈的每一个元素都可以是包括long和double在内的任意Java数据类型。 32位数据类型所占的栈容量为1， 64位数据类型所占的栈容量为2。

在方法的执行过程中， 会有各种字节码指令往操作数栈中写入和提取内容， 也就是出栈和入栈操作。 譬如在做算术运算的时候是通过将运算涉及的操作数栈压入栈顶后调用运算指令来进行的，又譬如在调用其他方法的时候是通过操作数栈来进行方法参数的传递。例如整数加法的字节码指令iadd， 这条指令在运行的时候要求操作数栈中最接近栈顶的两个元素已经存入了两个int型的数值， 当执行这个指令时， 会把这两个int值出栈并相加， 然后将相加的结果重新入栈

**动态连接：（处理方法覆盖，以便正确调用方法）**每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用， 持有这个引用是为了支持方法调用过程中的动态连接（Dynamic Linking）。Class文件的常量池中存有大量的符号引用， 字节码中的方法调用指令就以常量池里指向方法的符号引用作为参数。这些符号引用一部分会在类加载阶段或者第一次使用的时候就被转化为直接引用， 这种转化被称为静态解析。另外一部分将在每一次运行期间都转化为直接引用，这部分就称为动态连接。

**方法返回地址:**用于方法返回时恢复主调方调用方法时的最初位置

Java虚拟机的解释执行引擎被称为“基于栈的执行引擎”， 里面的“栈”就是操作数栈

在《Java虚拟机规范》中，对虚拟机栈内存区域规定了两类异常状况：如果线程请求的栈深度大于虚  
拟机所允许的深度， 将抛出StackOverflowError异常； 如果Java虚拟机栈容量可以动态扩展，当栈扩  
展时无法申请到足够的内存会抛出OutOfMemoryError异常

**・本地方法栈**（Native Method Stacks）：与虚拟机栈所发挥的作用是非常相似的，其区别只是虚拟机  
栈为虚拟机执行Java方法（也就是字节码）服务， 而本地方法栈则是为虚拟机使用到的本地（Native）  
方法服务

**・**J**ava堆：**虚拟机所管理的内存中最大的一块。《Java虚拟机规范》 中对Java堆的描述是： “所有  
的对象实例以及数组都应当在堆上分配”

**・方法区**（Method Area）：与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载  
的类型信息、常量、静态变量、即时编译器编译后的代码缓存等数据

**运行时常量池**（Runtime Constant Pool） 是方法区的一部分。 Class文件中除了有类的版本、 字  
段、 方法、 接口等描述信息外， 还有一项信息是常量池表（Constant Pool Table） ， 用于存放编译期生成的各种字面量与符号引用， 这部分内容将在类加载后存放到方法区的运行时常量池中

运行时常量池相对于Class文件常量池的另外一个重要特征是具备动态性， Java语言并不要求常量  
一定只有编译期才能产生， 也就是说， 并非预置入Class文件中常量池的内容才能进入方法区运行时常量池， 运行期间也可以将新的常量放入池中， 这种特性被开发人员利用得比较多的便是String类的intern()方法

**垃圾回收**

垃圾回收需要要解决三个问题:哪些内存需要回收？什么时候回收？如何回收？

**哪些内存需要回收：判定方法**

**・引用计数算法：**在对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加一；当引用失效时，计数器值就减一；任何时刻计数器为零的对象就是不可能再被使用的，可被回收

**优点：**原理简单，判定效率高

**缺陷：**相互引用的对象导致计数器值不为0，从而无法回收，需要配合额外的机制来避免

**・可达性分析算法：**通过一系列称为“GC Roots”的根对象作为起始节点集，从这些节点开始，根据引用关系向下搜索，搜索过程所走过的路径称为“引用链”（Reference Chain），如果某个对象到GC Roots间没有任何引用链相连，或者用图论的话来说就是从GC Roots到这个对象不可达时，则证明此对象是不可能再被使用的

**Java内存管理子系统采用可达性分析算法判定对象是否可以回收**

在Java技术体系里面， 固定可作为GC Roots的对象包括以下几种：  
·在虚拟机栈（栈帧中的本地变量表）中引用的对象，譬如各个线程被调用的方法堆栈中使用到的  
参数、局部变量、临时变量等。  
·在方法区中类静态属性引用的对象， 譬如Java类的引用类型静态变量。  
·在方法区中常量引用的对象， 譬如字符串常量池（String Table） 里的引用。  
·在本地方法栈中JNI（即通常所说的Native方法） 引用的对象。  
·Java虚拟机内部的引用， 如基本数据类型对应的Class对象， 一些常驻的异常对象（比如  
NullPointExcepiton、 OutOfMemoryError） 等， 还有系统类加载器。  
·所有被同步锁（synchronized关键字） 持有的对象。  
·反映Java虚拟机内部情况的JMXBean、 JVMTI中注册的回调、 本地代码缓存等。

除了这些固定的GC Roots集合以外， 根据用户所选用的垃圾收集器以及当前回收的内存区域不  
同， 还可以有其他对象“临时性”地加入， 共同构成完整GC Roots集合

\*真正宣告一个对象死亡， 至少要经历两次标记过程： 如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链， 那它将会被第一次标记， 随后进行一次筛选， 筛选的条件是此对象是否有必要执行finalize()方法。 假如对象没有覆盖finalize()方法， 或者finalize()方法已经被虚拟机调用过，那么虚拟机将这两种情况都视为“没有必要执行”,将被回收。finalize()方法中只要重新与引用链上的任何一个对象建立关联，即可逃脱一次垃圾回收过程，仅且一次

**什么时候回收：GC触发时机**

**如何回收：回收方案**

**分代收集理论：**主流虚拟机垃圾回收器都遵循该理论设计。分代收集名为理论，实质是一套符合大多数程序运行实际情况的经验法则， 它建立在两个分代假说之上：

1）弱分代假说（Weak Generational Hypothesis）：绝大多数对象都是朝生夕灭的。  
2）强分代假说（Strong Generational Hypothesis）：熬过越多次垃圾收集过程的对象就越难以消亡

根据上述假说收集器应该将Java堆划分出不同的区域， 然后将回收对象依据其年龄（年龄即对象熬过垃圾收集过程的次数） 分配到不同的区域之中存储。 显而易见， 如果一个区域中大多数对象都是朝生夕灭， 难以熬过垃圾收集过程的话， 那么把它们集中放在一起， 每次回收时只关注如何保留少量存活而不是去标记那些大量将要被回收的对象， 就能以较低代价回收到大量的空间； 如果剩下的都是难以消亡的对象， 那把它们集中放在一块，虚拟机便可以使用较低的频率来回收这个区域， 这就同时兼顾了垃圾收集的时间开销和内存的空间有效利用

**泛型**

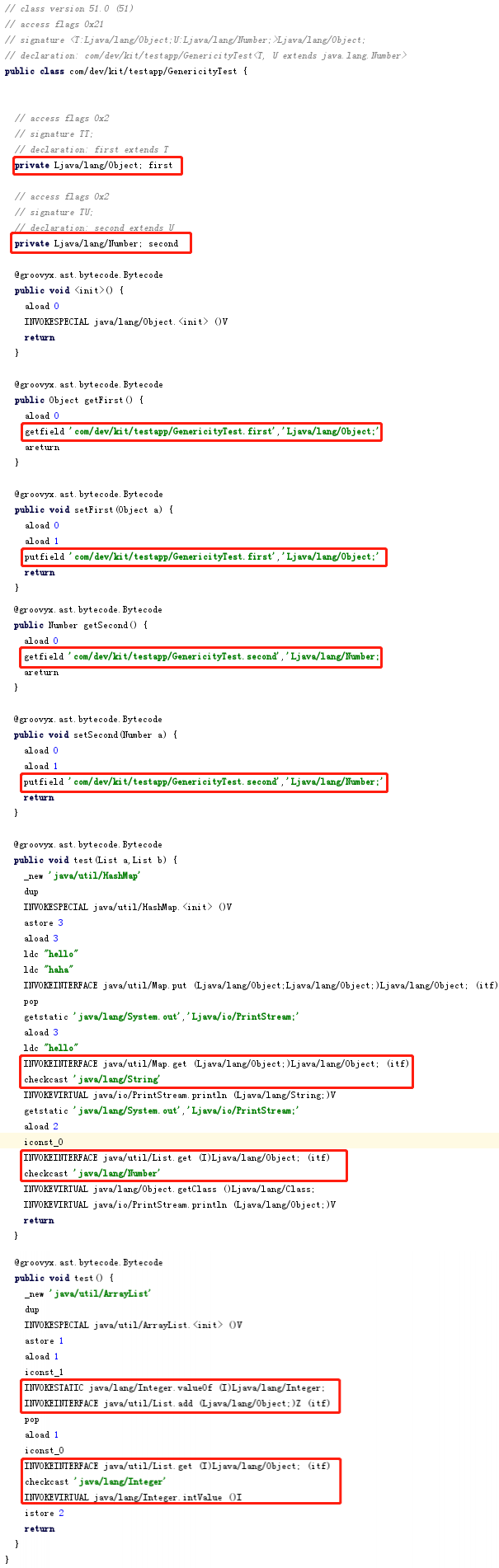
**泛型的本质是参数化类型（Parameterized Type） 或者参数化多态（Parametric Polymorphism） 的  
应用， 即可以将操作的数据类型指定为方法签名中的一种特殊参数， 这种参数类型能够用在类、 接口  
和方法的创建中， 分别构成泛型类、 泛型接口和泛型方法。**

Java选择的泛型实现方式叫作“类型擦除式泛型”，为了探究java泛型实现机制，先看一下泛型类经过编译之后的字节码是什么样的

**原始代码：**

****

**字节码：**



通过字节码可以看出，类型参数被擦除并被替换成了限定类型（无限定的变量用替换为Object），涉及到域的访问和修改的地方会自动插入强制类型转换指令。此外，原始类型（Primitive Types）数据还会发生装箱与拆箱操作。

**总结：**

Java选择的 “类型擦除式泛型”，没有创造新的泛型类，直接把已有的类型泛型化，让所有需要泛型化的已有类型， 譬如ArrayList， 原地泛型化后变成了ArrayList<T>， 而且保证以前直接用ArrayList的代码在泛型新版本里必须还能继续用这同一个容器， 这就必须让所有泛型化的实例类型， 譬如ArrayList<Integer>、 ArrayList<String>这些全部自动成为ArrayList的子类型才能可以， 否则类型转换就是不安全的。由此就引出了擦除类型参数后“裸类型”（Raw Type）的概念， 裸类型应被视为所有该类型泛型化实例的共同父类型（Super Type），只有这样，像如下代码清单中的赋值才是被系统允许的从子类到父类的安全转型。

ArrayList<Integer> ilist = new ArrayList<Integer>();  
ArrayList<String> slist = new ArrayList<String>();  
ArrayList list; // 裸类型  
list = ilist;  
list = slist;

对于裸类型的实现，java并没有选择在运行时由虚拟机自动地、真实地构造出ArrayList<Integer>这样的类型， 并且自动实现从ArrayList<Integer>派生自ArrayList的继承关系来满足裸类型的定义。而是把ArrayList<Integer>还原回ArrayList， 只在元素访问、 修改时自动插入一些强制类型转换和检查指令。

类型擦除导致无法直接获取泛型类中的参数化类型信息，需要额外引入新的属性(Signature、 LocalVariableTypeTable等)用于解决参数化类型的识别问题， Signature是其中最重要的一项属性， 它的作用就是存储一个方法在字节码层面的特征签名。

从Signature属性的出现我们还可以得出结论所谓的类型擦除， 仅仅是对方法的Code属性中的字节码进行擦除， 实际上元数据中还是保留了泛型信息， 这也是我们在编码时能通过反射手段取得参数化类型的根本依据

java擦除式泛型的优缺点：

优点：几乎只需要在Javac编译器上做出改进即可， 不需要改动字节码、 不需要改动Java虚拟机， 也保证了以前没有使用泛型的库可以直接运行在Java 5.0之上

缺点：使用不方便，效率低。尤其类似于ArrayList<Integer>这样的类型，面临自动拆箱与装箱的操作，效率更低

**Java线程及线程安全**

内核线程（Kernel-Level Thread， KLT） 就是直接由操作系统内核（Kernel， 下称内核） 支持的线程， 这种线程由内核来完成线程切换， 内核通过操纵调度器（Scheduler） 对线程进行调度， 并负责将线程的任务映射到各个处理器上。广义上来讲， 一个线程只要不是内核线程， 都可以认为是用户线程

主流Java虚拟机实现中， 每个Java线程都是映射到操作系统的原生内核线程之上的， 如果要阻塞或唤醒一条线程， 则需要操作系统来帮忙完成， 这就不可避免地陷入用户态到核心态的转换中， 进行这种状态转换需要耗费很多的处理器时间

Java线程可以通过yield()方法可以主动让出执行时间但无法主动获取执行时间。Java线程有10个优先级，可以通过设置优先级来建议操作系统为java线程多分配或者少分配执行时间，但最终还是由操作系统来决定。此外操作系统原生线程优先级的数量很可能与java线程优先级数量不一致，可能出现几个java线程优先级对应到同一个操作系统优先级的情况。

**锁优化**

通过锁(synchronized和Lock)实现的同步为互斥同步，互斥同步对性能最大的影响是阻塞的实现， 挂起线程和恢复线程的操作都需要转入内核态中完成，开销较大，甚至超过代码执行本身所需时间。为了避免频繁的线程切换开销，HotSpot虚拟机开发团队做了一些优化

**• 自旋等待**：对于需要请求锁的线程，让其执行一个忙循环（自旋，循环次数有限制）。自旋期间内其他线程可能已经释放了锁，自旋线程便可能获取到锁而无需阻塞

**• 锁消除**：指虚拟机即时编译器在运行时， 对一些代码要求同步， 但是对被检测到不可能存在共享  
数据竞争的锁进行消除，从而消除不必要的阻塞

**• 锁粗化**：如果虚拟机探测到有一连串的操作都对同一个对象加锁， 将会把加锁同步的范围扩展（粗化） 到整个操作序列的外部。（如果一系列的连续操作都对同一个对象反复加锁和解锁， 甚至加锁操作是出现在循环体之中的， 那即使没有线程竞争， 频繁地进行互斥同步操作也会导致不必要的性能损耗，例如连续的进行StringBuffer append操作）

**• 轻量级锁是**:JDK 6时加入的新型锁机制。

**核心思想**：在无竞争（没有其他线程争抢锁）的情况下使用CAS操作去消除同步使用的互斥量

轻量级锁能提升程序同步性能的依据是“对于绝大部分的锁， 在整个同步周期内都是不存在竞争  
的”这一经验法则。 如果没有竞争， 轻量级锁便通过CAS操作成功避免了使用互斥量的开销； 但如果确实存在锁竞争， 除了互斥量的本身开销外， 还额外发生了CAS操作的开销。 因此在有竞争的情况下，轻量级锁反而会比传统的重量级锁更慢

**实现方式**：对象头有2个比特用于存储锁标志位， 还有1个比特固定为0（这表示未进入偏向模式）。在代码即将进入同步块的时候， 如果此同步对象没有被锁定（锁标志位为“01”状态） ， 虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record） 的空间， 用于存储锁对象目前的Mark Word的拷贝，官方为这份拷贝加了一个Displaced前缀， 即Displaced Mark Word）。然后， 虚拟机将使用CAS操作尝试把对象的Mark Word更新为指向Lock Record的指针。 如果这个更新动作成功了， 即代表该线程拥有了这个对象的锁， 并且对象Mark Word的锁标志位（Mark Word的最后两个比特） 将转变为“00”， 表示此对象处于轻量级锁定状态。如果这个更新操作失败了， 那就意味着至少存在一条线程与当前线程竞争获取该对象的锁。 虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧， 如果是， 说明当前线程已经拥有了这个对象的锁， 那直接进入同步块继续执行就可以了， 否则就说明这个锁对象已经被其他线程抢占了。 如果出现两条以上的线程争用同一个锁的情况， 那轻量级锁就不再有效， 必须要膨胀为重量级锁， 锁标志的状态值变为“10”， 此时Mark Word中存储的就是指向重量级锁（互斥量） 的指针， 后面等待锁的线程也必须进入阻塞状态。解锁过程也同样是通过CAS操作来进行的， 如果对象的Mark Word仍然指向线程的锁记录， 那就用CAS操作把对象当前的Mark Word和线程中复制的DisplacedMark Word替换回来。 假如能够成功替换， 那整个同步过程就顺利完成了； 如果替换失败， 则说明有其他线程尝试过获取该锁， 就要在释放锁的同时， 唤醒被挂起的线程。

**• 偏向锁**：JDK 6中引入的一项锁优化措施

**核心思想**：锁偏向于第一个获得它的线程， 如果在接下来的执行过程中， 该锁一直没有被其他的线程获取， 则持有偏向锁的线程将永远不需要再进行同步，连CAS操作都不去做 目的是消除数据在无竞争情况下的同步原语，进一步提高程序的运行性能。

**实现方式：**当锁对象第一次被线程获取的时候， 虚拟机将会把对象头中的标志  
位设置为“01”、 把偏向模式设置为“1”， 表示进入偏向模式。 同时使用CAS操作把获取到这个锁的线程的ID记录在对象的Mark Word之中。 如果CAS操作成功， 持有偏向锁的线程以后每次进入这个锁相关的同步块时， 虚拟机都可以不再进行任何同步操作（例如加锁、 解锁及对Mark Word的更新操作等）。

一旦出现另外一个线程去尝试获取这个锁的情况， 偏向模式就马上宣告结束。 根据锁对象目前是否处于被锁定的状态决定是否撤销偏向（偏向模式设置为“0”） ， 撤销后标志位恢复到未锁定（标志位为“01”） 或轻量级锁定（标志位为“00”） 的状态， 后续的同步操作就按照上面介绍的轻量级锁那样去执行

{在HotSpot虚拟机里， 对象在堆内存中的存储布局可以划分为三个部分： 对象头（Header） 、 实例数据（Instance Data） 和对齐填充（Padding） 。  
HotSpot虚拟机对象的对象头部分包括两类信息。 第一类是用于存储对象自身的运行时数据， 如哈希码（HashCode） 、 GC分代年龄、 锁状态标志、 线程持有的锁、 偏向线程ID、 偏向时间戳等， 这部分数据的长度在32位和64位的虚拟机（未开启压缩指针） 中分别为32个比特和64个比特， 官方称它为“Mark Word”。}