## SpringMVC中来一个请求具体的过程。

## 红黑树什么时候使用，以及它的好处？

## Java 8中stream迭代的优势和区别？

Java 8函数式编程。

回调函数，函数式编程，面向对象之间区别。

ThreadLocal为什么保证线程私有？

框架封装jdbc受检异常的考虑和原因？

处理器指令优化有些什么考虑？

考禁止重排序

同步等于可见性吗？

Lock内部实现。

答：通过AbstractQueuedSynchronizer队列同步器实现。谈了谈队列同步器中的一些重要方法，以及一般自己怎么实现一个锁。

当获取第一个获取锁之后，条件不满足需要释放锁应当怎么做？

While+wait 或者中断

既然线程调用中断方法不会停止程序，那么有什么用？

我们知道一些阻塞线程的方法会抛出InterruptedException表示线程中断发生，在这种情况下就可以使用线程中断来终止线程的运行，很多任务执行的服务程序的逻辑和上面的例子很类似，都可以使用这种方法来终止线程的运行。

如果不了解Java的中断机制，这样的一种解释极容易造成误解，认为调用了线程的interrupt方法就一定会中断线程。   
其实，Java的中断是一种协作机制。也就是说调用线程对象的interrupt方法并不一定就中断了正在运行的线程，它只是要求线程自己在合适的时机中断自己。每个线程都有一个boolean的中断状态（不一定就是对象的属性，事实上，该状态也确实不是Thread的字段），interrupt方法仅仅只是将该状态置为true 。一般说来，如果一个方法声明抛出InterruptedException，表示该方法是可中断的（没有在方法中处理中断却也声明抛出InterruptedException的除外），也就是说可中断方法会对interrupt调用做出响应（例如sleep响应interrupt的操作包括清除中断状态，抛出InterruptedException），如果interrupt调用是在可中断方法之前调用，可中断方法一定会处理中断，像上面的例子，interrupt方法极可能在run未进入sleep的时候就调用了，但sleep检测到中断，就会处理该中断。如果在可中断方法正在执行中的时候调用interrupt，会怎么样呢？这就要看可中断方法处理中断的时机了，只要可中断方法能检测到中断状态为true，就应该处理中断。

简单的来说，Jvm内存模型规定unlock操作前，需要把改变的副本同步回主存，在lock操作之后会清空此变量在工作内存中原先的副本，需要再次从主内存read-load新的值，ReentrantLock做为官方的锁实现，完全遵守此原则。

在了解其详细的实现原理前，可先简单了解下happens before原则中的有一条：

——如果线程1写入了volatile变量v,接着线程2读取了v，那么，线程1 写入v及之前的写操作都对线程2可见。也就是说，如果你感知到了volatile变量v的变化，那么在v之前的所有写操作你都可以感知的到。

ReentrantLock的操作都是委托给其内部抽象类Sync的实例来实现的,但他们实现的lock/unlock原理是通过改变父类中AbstractQueuedSynchronizer中的private volatile int state变量的值来表示加锁、重入、解锁状态的。

当state=0的时候，表示没有线程持有锁，如果此时有线程A调用reentrantLock.lock(),会调用Unsafe类通过Compare And Swap的方式将state加1设置为1，此时当前线程A持有锁，其他线程调用reentrantLock.lock()，发现state>0(由于ReetrantLock可重入，故A可以多次调用reentrantLock.lock()，所以会出现state>1的情况),会阻塞等待A线程释放锁，当A调用reentrantLock.unlock()时，state会减去1，调用后如果state=0,那么线程A就释放了锁，此时其他线程发现发现无任何线程持有锁（重点：即感知到了state=0），那么就会尝试获取锁，即把state设置为1，这样根据之前的happens-before原则，volatile就保证了可见性。

当重写equal方法时候一定要记得重写hashcode方法

能解释下为什么吗？

equals方法（改写后），两个不同的实例有可能在逻辑上是相等的。。。

## 为什么匿名内部类和局部内部类只能访问final变量

## 在java提高篇-----详解内部类中对匿名内部类做了一个简单的介绍，但是内部类还存在很多其他细节问题，所以就衍生出这篇博客。在这篇博客中你可以了解到匿名内部类的使用、匿名内部类要注意的事项、如何初始化匿名内部类、匿名内部类使用的形参为何要为final。

## 一、使用匿名内部类内部类

## 匿名内部类由于没有名字，所以它的创建方式有点儿奇怪。创建格式如下：

## new 父类构造器（参数列表）|实现接口（）

## {

## //匿名内部类的类体部分

## }

## 在这里我们看到使用匿名内部类我们必须要继承一个父类或者实现一个接口，当然也仅能只继承一个父类或者实现一个接口。同时它也是没有class关键字，这是因为匿名内部类是直接使用new来生成一个对象的引用。当然这个引用是隐式的。

## public abstract class Bird {

## private String name;

## public String getName() {

## return name;

## }

## public void setName(String name) {

## this.name = name;

## }

## public abstract int fly();

## }

## public class Test {

## public void test(Bird bird){

## System.out.println(bird.getName() + "能够飞 " + bird.fly() + "米");

## }

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

## Test test = new Test();

## test.test(new Bird() {

## public int fly() {

## return 10000;

## }

## public String getName() {

## return "大雁";

## }

## });

## }

## }

## ------------------

## Output：

## 大雁能够飞 10000米

## 在Test类中，test()方法接受一个Bird类型的参数，同时我们知道一个抽象类是没有办法直接new的，我们必须要先有实现类才能new出来它的实现类实例。所以在mian方法中直接使用匿名内部类来创建一个Bird实例。

## 由于匿名内部类不能是抽象类，所以它必须要实现它的抽象父类或者接口里面所有的抽象方法。

## 对于这段匿名内部类代码其实是可以拆分为如下形式：

## public class WildGoose extends Bird{

## public int fly() {

## return 10000;

## }

## public String getName() {

## return "大雁";

## }

## }

## WildGoose wildGoose = new WildGoose();

## test.test(wildGoose);

## 在这里系统会创建一个继承自Bird类的匿名类的对象，该对象转型为对Bird类型的引用。

## 对于匿名内部类的使用它是存在一个缺陷的，就是它仅能被使用一次，创建匿名内部类时它会立即创建一个该类的实例，该类的定义会立即消失，所以匿名内部类是不能够被重复使用。对于上面的实例，如果我们需要对test()方法里面内部类进行多次使用，建议重新定义类，而不是使用匿名内部类。

## 二、注意事项

## 在使用匿名内部类的过程中，我们需要注意如下几点：

## 1、使用匿名内部类时，我们必须是继承一个类或者实现一个接口，但是两者不可兼得，同时也只能继承一个类或者实现一个接口。

## 2、匿名内部类中是不能定义构造函数的。

## 3、匿名内部类中不能存在任何的静态成员变量和静态方法。

## 4、匿名内部类为局部内部类，所以局部内部类的所有限制同样对匿名内部类生效。

## 5、匿名内部类不能是抽象的，它必须要实现继承的类或者实现的接口的所有抽象方法。

## 使用的形参为何要为final

## 我们给匿名内部类传递参数的时候，若该形参在内部类中需要被使用，那么该形参必须要为final。也就是说：当所在的方法的形参需要被内部类里面使用时，该形参必须为final。

## 为什么必须要为final呢？

## 首先我们知道在内部类编译成功后，它会产生一个class文件，该class文件与外部类并不是同一class文件，仅仅只保留对外部类的引用。当外部类传入的参数需要被内部类调用时，从java程序的角度来看是直接被调用：

## public class OuterClass {

## public void display(final String name,String age){

## class InnerClass{

## void display(){

## System.out.println(name);

## }

## }

## }

## }

## 从上面代码中看好像name参数应该是被内部类直接调用？其实不然，在java编译之后实际的操作如下：

## public class OuterClass$InnerClass {

## public InnerClass(String name,String age){

## this.InnerClass$name = name;

## this.InnerClass$age = age;

## }

## public void display(){

## System.out.println(this.InnerClass$name + "----" + this.InnerClass$age );

## }

## }

## 所以从上面代码来看，内部类并不是直接调用方法传递的参数，而是利用自身的构造器对传入的参数进行备份，自己内部方法调用的实际上时自己的属性而不是外部方法传递进来的参数。

## 直到这里还没有解释为什么是final？在内部类中的属性和外部方法的参数两者从外表上看是同一个东西，但实际上却不是，所以他们两者是可以任意变化的，也就是说在内部类中我对属性的改变并不会影响到外部的形参，而然这从程序员的角度来看这是不可行的，毕竟站在程序的角度来看这两个根本就是同一个，如果内部类改变了，而外部方法的形参却没有改变这是难以理解和不可接受的，所以为了保持参数的一致性，就规定使用final来避免形参的不改变。

## 简单理解就是，拷贝引用，为了避免引用值发生改变，例如被外部类的方法修改等，而导致内部类得到的值不一致，于是用final来让该引用不可改变。

## 故如果定义了一个匿名内部类，并且希望它使用一个其外部定义的参数，那么编译器会要求该参数引用是final的。

## 四、匿名内部类初始化

## 我们一般都是利用构造器来完成某个实例的初始化工作的，但是匿名内部类是没有构造器的！那怎么来初始化匿名内部类呢？使用构造代码块！利用构造代码块能够达到为匿名内部类创建一个构造器的效果。

## public class OutClass {

## public InnerClass getInnerClass(final int age,final String name){

## return new InnerClass() {

## int age\_ ;

## String name\_;

## //构造代码块完成初始化工作

## {

## if(0 < age && age < 200){

## age\_ = age;

## name\_ = name;

## }

## }

## public String getName() {

## return name\_;

## }

## 

## public int getAge() {

## return age\_;

## }

## };

## }

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

## OutClass out = new OutClass();

## InnerClass inner\_1 = out.getInnerClass(201, "chenssy");

## System.out.println(inner\_1.getName());

## InnerClass inner\_2 = out.getInnerClass(23, "chenssy");

## System.out.println(inner\_2.getName());

## }

## }

## JVM中的Class只有满足以下三个条件，才能被GC回收，也就是该Class被卸载（unload）：

## - 该类所有的实例都已经被GC，也就是JVM中不存在该Class的任何实例。

## - 加载该类的ClassLoader已经被GC。

## - 该类的Java.lang.Class 对象没有在任何地方被引用，如不能在任何地方通过反射访问该类的方法

 首先看一下，关于java虚拟机规范中时如何阐述类型卸载(unloading)的：

A class or interface may be unloaded if and only if its class loader is unreachable. The bootstrap class loader is always reachable; as a result， system classes may never be unloaded.  
    **A class or interface may be unloaded if and only if its class loader is unreachable. The bootstrap class loader is always reachable; as a result， system classes may never be unloaded.**    Java虚拟机规范中关于类型卸载的内容就这么简单两句话，大致意思就是：只有当加载该类型的类加载器实例(非类加载器类型)为unreachable状态时，当前被加载的类型才被卸载.启动类加载器实例永远为reachable状态，由启动类加载器加载的类型可能永远不会被卸载.  
  
    我们再看一下Java语言规范提供的关于类型卸载的更详细的信息(部分摘录)：  
    //摘自JLS 12.7 Unloading of Classes and Interfaces  
    1、An implementation of the Java programming language may unload classes.  
    2、Class unloading is an optimization that helps reduce memory use. Obviously，the semantics of a program should not depend  on whether and how a system chooses to implement an optimization such as class unloading.  
    3、Consequently，whether a class or interface has been unloaded or not should be transparent to a program  
  
    通过以上我们可以得出结论： 类型卸载(unloading)仅仅是作为一种减少内存使用的性能优化措施存在的，具体和虚拟机实现有关，对开发者来说是透明的.  
  
    纵观java语言规范及其相关的API规范，找不到显示类型卸载(unloading)的接口， 换句话说：   
    1、一个已经加载的类型被卸载的几率很小至少被卸载的时间是不确定的  
    2、一个被特定类加载器实例加载的类型运行时可以认为是无法被更新的

****【类型卸载总结】**通过以上的相关测试(虽然测试的场景较为简单)我们可以大致这样概括：**    1、由启动类加载器加载的类型在整个运行期间是不可能被卸载的(jvm和jls规范).  
    2、被系统类加载器和标准扩展类加载器加载的类型在运行期间不太可能被卸载，因为系统类加载器实例或者标准扩展类加载器的实例基本上在整个运行期间总能直接或者间接的访问的到，其达到unreachable的可能性极小.(当然，在虚拟机快退出的时候可以，因为不管ClassLoader实例或者Class(java.lang.Class)实例也都是在堆中存在，同样遵循垃圾收集的规则).  
    3、被开发者自定义的类加载器实例加载的类型只有在很简单的上下文环境中才能被卸载，而且一般还要借助于强制调用虚拟机的垃圾收集功能才可以做到.可以预想，稍微复杂点的应用场景中(尤其很多时候，用户在开发自定义类加载器实例的时候采用缓存的策略以提高系统性能)，被加载的类型在运行期间也是几乎不太可能被卸载的(至少卸载的时间是不确定的).  
  
      综合以上三点，我们可以默认前面的结论1， 一个已经加载的类型被卸载的几率很小至少被卸载的时间是不确定的.同时，我们可以看的出来，开发者在开发代码时候，不应该对虚拟机的类型卸载做任何假设的前提下来实现系统中的特定功能.

## 1.Java集合框架是什么？说出一些集合框架的优点？

每种编程语言中都有集合，最初的Java版本包含几种集合类：Vector、Stack、HashTable和Array。随着集合的广泛使用，Java1.2提出了囊括所有集合接口、实现和算法的集合框架。在保证线程安全的情况下使用泛型和并发集合类，Java已经经历了很久。它还包括在Java并发包中，阻塞接口以及它们的实现。集合框架的部分优点如下：

（1）使用核心集合类降低开发成本，而非实现我们自己的集合类。

（2）随着使用经过严格测试的集合框架类，代码质量会得到提高。

（3）通过使用JDK附带的集合类，可以降低代码维护成本。

（4）复用性和可操作性。

## 2.集合框架中的泛型有什么优点？

Java1.5引入了泛型，所有的集合接口和实现都大量地使用它。泛型允许我们为集合提供一个可以容纳的对象类型，因此，如果你添加其它类型的任何元素，它会在编译时报错。这避免了在运行时出现ClassCastException，因为你将会在编译时得到报错信息。泛型也使得代码整洁，我们不需要使用显式转换和instanceOf操作符。它也给运行时带来好处，因为不会产生类型检查的字节码指令。

## 3.Java集合框架的基础接口有哪些？

Collection为集合层级的根接口。**一个集合代表一组对象，这些对象即为它的元素。**Java平台不提供这个接口任何直接的实现。

Set是一个不能包含重复元素的集合。这个接口对数学集合抽象进行建模，被用来代表集合，就如一副牌。

List是一个有序集合，可以包含重复元素。你可以通过它的索引来访问任何元素。List更像长度动态变换的数组。

Map是一个将key映射到value的对象.一个Map不能包含重复的key：每个key最多只能映射一个value。

一些其它的接口有Queue、Dequeue、SortedSet、SortedMap和ListIterator。

## 4.为何Collection不从Cloneable和Serializable接口继承？

**Collection接口指定一组对象，对象即为它的元素。如何维护这些元素由Collection的具体实现决定。**例如，一些如List的Collection实现允许重复的元素，而其它的如Set就不允许。很多Collection实现有一个公有的clone方法。然而，把它放到集合的所有实现中也是没有意义的。**这是因为Collection是一个抽象表现。重要的是实现。**

**当与具体实现打交道的时候，克隆或序列化的语义和含义才发挥作用**。所以，具**体实现应该决定如何对它进行克隆或序列化，或它是否可以被克隆或序列化。**

在所有的实现中授权克隆和序列化，最终导致更少的灵活性和更多的限制。特定的实现应该决定它是否可以被克隆和序列化。

## 5.为何Map接口不继承Collection接口？

尽管Map接口和它的实现也是集合框架的一部分，**但Map不是集合，集合也不是Map。**因此，Map继承Collection毫无意义，反之亦然。

如果Map继承Collection接口，那么元素去哪儿？Map包含key-value对，它提供抽取key或value列表集合的方法，但是**它不适合“一组对象”规范。**

## 6.Iterator是什么？

**Iterator接口提供遍历任何Collection的接口。**我们可以从一个Collection中使用迭代器方法来获取迭代器实例。迭代器取代了Java集合框架中的Enumeration。迭代器允许调用者在迭代过程中移除元素。

## 7.Enumeration和Iterator接口的区别？

**Enumeration的速度是Iterator的两倍，也使用更少的内存。**Enumeration是非常基础的，也满足了基础的需要。但是，与Enumeration相比，Iterator更加安全，因为当一个集合正在被遍历的时候，它会阻止其它线程去修改集合。

迭代器取代了Java集合框架中的Enumeration。迭代器允许调用者从集合中移除元素，而Enumeration不能做到。为了使它的功能更加清晰，迭代器方法名已经经过改善。

## 8.为何没有像Iterator.add()这样的方法，向集合中添加元素？

语义不明，已知的是，Iterator的协议不能确保迭代的次序。然而要注意，ListIterator没有提供一个add操作，它要确保迭代的顺序。

## 9.为何迭代器没有一个方法可以直接获取下一个元素，而不需要移动游标？

它可以在当前Iterator的顶层实现，但是它用得很少，如果将它加到接口中，每个继承都要去实现它，这没有意义。

## 10.Iterater和ListIterator之间有什么区别？

（1）我们可以**使用Iterator来遍历Set和List集合，而ListIterator只能遍历List。**

（2）**Iterator只可以向前遍历，而ListIterator可以双向遍历。**

（3）**ListIterator从Iterator接口继承，然后添加了一些额外的功能，比如添加一个元素、替换一个元素、获取前面或后面元素的索引位置。**

## 11.遍历一个List有哪些不同的方式？

List<String> strList = new ArrayList<>();

//使用for-each循环

for(String obj : strList)

{

System.out.println(obj);

}

//using iterator

Iterator<String> it = strList.iterator();

while(it.hasNext())

{

String obj = it.next();

System.out.println(obj);

}

使用迭代器更加线程安全，因为它可以确保，在当前遍历的集合元素被更改的时候，它会抛出ConcurrentModificationException。

## 12.通过迭代器fail-fast属性，你明白了什么？

每次我们尝试获取下一个元素的时候，**Iterator fail-fast属性检查当前集合结构里的任何改动。如果发现任何改动，它抛出ConcurrentModificationException**。Collection中所有Iterator的实现都是按fail-fast来设计的（ConcurrentHashMap和CopyOnWriteArrayList这类并发集合类除外）。

## 13.fail-fast与fail-safe有什么区别？

**Iterator的fail-fast属性与当前的集合共同起作用，因此它不会受到集合中任何改动的影响**。**Java.util包中的所有集合类都被设计为fail-fast**的，而**java.util.concurrent中的集合类都为fail-safe**的。Fail-fast迭代器抛出**ConcurrentModificationException**，而fail-safe迭代器从不抛出ConcurrentModificationException。

## 14.在迭代一个集合的时候，如何避免ConcurrentModificationException？

在遍历一个集合的时候，我们可以使用**并发集合类来避免ConcurrentModificationException**，比如使用CopyOnWriteArrayList，而不是ArrayList。

## 15.为何Iterator接口没有具体的实现？

**Iterator接口定义了遍历集合的方法，但它的实现则是集合实现类的责任**。每个能够返回用于遍历的Iterator的集合类都有它自己的Iterator实现内部类。

**这就允许集合类去选择迭代器是fail-fast还是fail-safe的**。比如，ArrayList迭代器是fail-fast的，而CopyOnWriteArrayList迭代器是fail-safe的。

## 16.UnsupportedOperationException是什么？

UnsupportedOperationException是用于表明**操作不支持的异常**。在JDK类中已被大量运用，在集合框架**java.util.Collections.UnmodifiableCollection将会在所有add和remove操作中抛出这个异常**。

## 17.在Java中，HashMap是如何工作的？

HashMap在Map.Entry静态内部类实现中存储key-value对。HashMap使用哈希算法，在put和get方法中，它使用hashCode()和equals()方法。当我们通过传递key-value对调用put方法的时候，HashMap使用Key hashCode()和哈希算法来找出存储key-value对的索引。Entry存储在LinkedList中，所以如果存在entry，它使用equals()方法来检查传递的key是否已经存在，如果存在，它会覆盖value，如果不存在，它会创建一个新的entry然后保存。当我们通过传递key调用get方法时，它再次使用hashCode()来找到数组中的索引，然后使用equals()方法找出正确的Entry，然后返回它的值。下面的图片解释了详细内容。

其它关于HashMap比较重要的问题是容量、负荷系数和阀值调整。HashMap默认的初始容量是16，负荷系数是0.75。阀值是为负荷系数乘以容量，无论何时我们尝试添加一个entry，如果map的大小比阀值大的时候，HashMap会对map的内容进行重新哈希，且使用更大的容量。容量总是2的幂，所以如果你知道你需要存储大量的key-value对，比如缓存从数据库里面拉取的数据，使用正确的容量和负荷系数对HashMap进行初始化是个不错的做法。

## 18.hashCode()和equals()方法有何重要性？

HashMap使用Key对象的hashCode()和equals()方法去决定key-value对的索引。当我们试着从HashMap中获取值的时候，这些方法也会被用到。如果这些方法没有被正确地实现，在这种情况下，两个不同Key也许会产生相同的hashCode()和equals()输出，HashMap将会认为它们是相同的，然后覆盖它们，而非把它们存储到不同的地方。同样的，所有不允许存储重复数据的集合类都使用hashCode()和equals()去查找重复，所以正确实现它们非常重要。equals()和hashCode()的实现应该遵循以下规则：

（1）如果o1.equals(o2)，那么o1.hashCode() == o2.hashCode()总是为true的。

（2）如果o1.hashCode() == o2.hashCode()，并不意味着o1.equals(o2)会为true。

## 19.我们能否使用任何类作为Map的key？

我们可以使用任何类作为Map的key，然而在使用它们之前，需要考虑以下几点：

（1）如果类重写了equals()方法，它也应该重写hashCode()方法。

（2）类的所有实例需要遵循与equals()和hashCode()相关的规则。请参考之前提到的这些规则。

（3）如果一个类没有使用equals()，你不应该在hashCode()中使用它。

（4）用户自定义key类的最佳实践是使之为不可变的，这样，hashCode()值可以被缓存起来，拥有更好的性能。不可变的类也可以确保hashCode()和equals()在未来不会改变，这样就会解决与可变相关的问题了。

比如，我有一个类MyKey，在HashMap中使用它。

//传递给MyKey的name参数被用于equals()和hashCode()中 MyKey key = new MyKey('Pankaj'); //assume hashCode=1234 myHashMap.put(key, 'Value'); // 以下的代码会改变key的hashCode()和equals()值 key.setName('Amit'); //assume new hashCode=7890 //下面会返回null，因为HashMap会尝试查找存储同样索引的key，而key已被改变了，匹配失败，返回null myHashMap.get(new MyKey('Pankaj'));

那就是为何String和Integer被作为HashMap的key大量使用。

## 20.Map接口提供了哪些不同的集合视图？

Map接口提供三个集合视图：

（1）Set keyset()：**返回map中包含的所有key的一个Set视图**。集合是受map支持的，map的变化会在集合中反映出来，反之亦然。当一个迭代器正在遍历一个集合时，若map被修改了（除迭代器自身的移除操作以外），迭代器的结果会变为未定义。**集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除，从map中移除对应的映射。它不支持add和addAll操作。**

（2）Collection values()：**返回一个map中包含的所有value的一个Collection视图**。这个collection受map支持的，map的变化会在collection中反映出来，反之亦然。当一个迭代器正在遍历一个collection时，若map被修改了（除迭代器自身的移除操作以外），迭代器的结果会变为未定义。集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除，从map中移除对应的映射。它不支持add和addAll操作。

（3）Set<Map.Entry<K,V>> entrySet()：**返回一个map钟包含的所有映射的一个集合视图**。这个集合受map支持的，map的变化会在collection中反映出来，反之亦然。当一个迭代器正在遍历一个集合时，若map被修改了（除迭代器自身的移除操作，以及对迭代器返回的entry进行setValue外），迭代器的结果会变为未定义。**集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除，从map中移除对应的映射。**它不支持add和addAll操作。

## 21.HashMap和HashTable有何不同？

（1）HashMap允许key和value为null，而HashTable不允许。

（2）HashTable是同步的，而HashMap不是。所以HashMap适合单线程环境，HashTable适合多线程环境。

（3）**在Java1.4中引入了LinkedHashMap，HashMap的一个子类，假如你想要遍历顺序，你很容易从HashMap转向LinkedHashMap**，但是**HashTable不是这样的，它的顺序是不可预知的**。

（4）HashMap提供对key的Set进行遍历，因此它是fail-fast的，但HashTable提供对key的Enumeration进行遍历，它不支持fail-fast。

（5）HashTable被认为是个遗留的类，如果你**寻求在迭代的时候修改Map，你应该使用CocurrentHashMap。**

## 22.如何决定选用HashMap还是TreeMap？

对于在**Map中插入、删除和定位元素这类操作，HashMap是最好的选择**。然而，**假如你需要对一个有序的key集合进行遍历，TreeMap是更好的选择**。基于你的collection的大小，也许向HashMap中添加元素会更快，将map换为TreeMap进行有序key的遍历。

## 23.ArrayList和Vector有何异同点？

ArrayList和Vector在很多时候都很类似。

（1）**两者都是基于索引的，内部由一个数组支持**。

（2）**两者维护插入的顺序，我们可以根据插入顺序来获取元素**。

（3）**ArrayList和Vector的迭代器实现都是fail-fast的**。

（4）Arra**yList和Vector两者允许null值**，也可以使用索引值对元素进行随机访问。

以下是ArrayList和Vector的不同点。

（1）**Vector是同步的，而ArrayList不是**。然而，如果你**寻求在迭代的时候对列表进行改变，你应该使用CopyOnWriteArrayList。**

（2）**ArrayList比Vector快，它因为有同步，不会过载**。

（3）ArrayList更加通用，因为我们可以使用Collections工具类轻易地获取同步列表和只读列表。

## 24.Array和ArrayList有何区别？什么时候更适合用Array？

Array可以容纳基本类型和对象，而ArrayList只能容纳对象。

Array是指定大小的，而ArrayList大小是不固定的。

Array没有提供ArrayList那么多功能，比如addAll、removeAll和iterator等。尽管ArrayList明显是更好的选择，但也有些时候Array比较好用。

（1）如**果列表的大小已经指定，大部分情况下是存储和遍历它们**。

（2）对于**遍历基本数据类型，尽管Collections使用自动装箱来减轻编码任务，在指定大小的基本类型的列表上工作也会变得很慢。**

（3）如果你要**使用多维数组，使用[][]比List<List<>>更容易**。

## 25.ArrayList和LinkedList有何区别？

**ArrayList和LinkedList两者都实现了List接口**，但是它们之间有些不同。

（1）ArrayList是由Array所支持的基于一个索引的数据结构，所以它提供对元素的随机访问，复杂度为O(1)，但LinkedList存储一系列的节点数据，每个节点都与前一个和下一个节点相连接。所以，尽管有使用索引获取元素的方法，内部实现是从起始点开始遍历，遍历到索引的节点然后返回元素，时间复杂度为O(n)，比ArrayList要慢。

（2）与ArrayList相比，在LinkedList中插入、添加和删除一个元素会更快，因为在一个元素被插入到中间的时候，不会涉及改变数组的大小，或更新索引。

（3）LinkedList比ArrayList消耗更多的内存，因为LinkedList中的每个节点存储了前后节点的引用。

## 26.哪些集合类提供对元素的随机访问？

**ArrayList、HashMap、TreeMap和HashTable**类提供对元素的随机访问。

## 27.EnumSet是什么？

java.util.**EnumSet是使用枚举类型的集合实现**。当集合创建时，枚举集合中的所有元素必须来自单个指定的枚举类型，可以是显示的或隐示的。**EnumSet是不同步的，不允许值为null的元素**。它也提供了一些有用的方法，比如copyOf(Collection c)、of(E first,E…rest)和complementOf(EnumSet s)。

## 28.哪些集合类是线程安全的？

**Vector、HashTable、Properties和Stack是同步类，所以它们是线程安全的**，可以在多线程环境下使用。Java1.5并发API包括一些集合类，允许迭代时修改，因为它们都工作在集合的克隆上，所以它们在多线程环境中是安全的。

## 29.并发集合类是什么？

Java1.5并发包（**java.util.concurrent**）包含线程安全集合类，允许在迭代时修改集合。迭代器被设计为fail-safe的，不会抛出ConcurrentModificationException。一部分类为：CopyOnWriteArrayList、 ConcurrentHashMap、CopyOnWriteArraySet。

## 30.BlockingQueue是什么？

Java.util.concurrent.**BlockingQueue是一个队列，在进行检索或移除一个元素的时候，它会等待队列变为非空；当在添加一个元素时，它会等待队列中的可用空间**。BlockingQueue接口是Java集合框架的一部分，主要用于**实现生产者-消费者模式**。我们不需要担心等待生产者有可用的空间，或消费者有可用的对象，因为它都在BlockingQueue的实现类中被处理了。Java提供了集中BlockingQueue的实现，比如**ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue,、SynchronousQueue**等。

## 31.队列和栈是什么，列出它们的区别？

**栈和队列两者都被用来预存储数据**。java.util.**Queue是一个接口**，它的实现类在Java并发包中。**队列允许先进先出（FIFO）检索元素（LinkedList），但并非总是这样**。**Deque接口允许从两端检索元素**。

栈与队列很相似，但它允许对元素进行**后进先出（LIFO）**进行检索。

**Stack是一个扩展自Vector的类，而Queue是一个接口**。

## 32.Collections类是什么？

Java.util.**Collections是一个工具类仅包含静态方法，它们操作或返回集合**。它包含操作集合的多态算法，返回一个由指定集合支持的新集合和其它一些内容。这个类包含集合框架算法的方法，比如**折半搜索**、**排序**、**混编和逆序**等。

## 33.Comparable和Comparator接口是什么？

如果我们**想使用Array或Collection的排序方法时**，**需要在自定义类里实现Java提供Comparable接口**。**Comparable接口有compareTo(T OBJ)方法**，它被排序方法所使用。我们应该重写这个方法，如果“this”对象比传递的对象参数更小、相等或更大时，它返回一个负整数、0或正整数。但是，在大多数实际情况下，我们想根据不同参数进行排序。比如，作为一个CEO，我想对雇员基于薪资进行排序，一个HR想基于年龄对他们进行排序。这就是我们需要使用Comparator接口的情景，因为Comparable.compareTo(Object o)方法实现只能基于一个字段进行排序，我们不能根据对象排序的需要选择字段。**Comparator接口的compare(Object o1, Object o2)方法的实现需要传递两个对象参数**，若第一个参数比第二个小，返回负整数；若第一个等于第二个，返回0；若第一个比第二个大，返回正整数。

## 34.Comparable和Comparator接口有何区别？

Comparable和Comparator接口**被用来对对象集合或者数组进行排序**。**Comparable接口被用来提供对象的自然排序，我们可以使用它来提供基于单个逻辑的排序。**

**Comparator接口被用来提供不同的排序算法，我们可以选择需要使用的Comparator来对给定的对象集合进行排序。**

## 35.我们如何对一组对象进行排序？

如果我们需要对一个对象数组进行排序，我们可以使用Arrays.sort()方法。如果我们需要排序一个对象集合，我们可以使用Collections.sort()方法。两个类都有用于自然排序（使用Comparable）或基于标准的排序（使用Comparator）的重载方法sort()。**Collections内部使用数组排序方法**，所有它们两者都有相同的性能，**只是Collections需要花时间将列表转换为数组**。

## 36.当一个集合被作为参数传递给一个函数时，如何才可以确保函数不能修改它？

在作为参数传递之前，我们可以**使用Collections.unmodifiableCollection(Collection c)方法创建一个只读集合，这将确保改变集合的任何操作都会抛出UnsupportedOperationException。**

## 37.我们如何从给定集合那里创建一个synchronized的集合？

我们可以使用Collections.synchronizedCollection(Collection c)根据指定集合来获取一个synchronized（线程安全的）集合。

## 38.集合框架里实现的通用算法有哪些？

Java集合框架提供常用的算法实现，比如**排序和搜索**。**Collections类包含这些方法实现。大部分算法是操作List的，但一部分对所有类型的集合都是可用的**。部分算法有排序、搜索、**混编、最大最小值**。

## 39.大写的O是什么？举几个例子？

大写的O描述的是，就**数据结构中的一系列元素而言**，**一个算法的性能**。Collection类就是实际的数据结构，我们通常基于时间、内存和性能，使用大写的O来选择集合实现。比如：例子1：ArrayList的get(index i)是一个常量时间操作，它不依赖list中元素的数量。所以它的性能是O(1)。例子2：一个对于数组或列表的线性搜索的性能是O(n)，因为我们需要遍历所有的元素来查找需要的元素。

## 40.与Java集合框架相关的有哪些最好的实践？

（1）**根据需要选择正确的集合类型**。比如，**如果指定了大小，我们会选用Array**而非ArrayList。如果我们想**根据插入顺序遍历一个Map，我们需要使用TreeMap**。如果我们**不想重复，我们应该使用Set**。

（2）**一些集合类允许指定初始容量，所以如果我们能够估计到存储元素的数量，我们可以使用它**，就避免了重新哈希或大小调整。

（3）**基于接口编程，而非基于实现编程**，它允许我们后来轻易地改变实现。

（4）**总是使用类型安全的泛型**，避免在运行时出现ClassCastException。

（5）**使用JDK提供的不可变类作为Map的key，可以避免自己实现hashCode()和equals()**。

（6）尽可能使用**Collections工具类，或者获取只读、同步或空的集合**，而非编写自己

1. Java中的原始数据类型都有哪些，它们的大小及对应的封装类是什么？

（1）boolean

   boolean数据类型非true即false。这个数据类型表示1 bit的信息，但是它的大小并没有精确定义。

   《Java虚拟机规范》中如是说：“虽然定义了boolean这种数据类型，但是只对它提供了非常有限的支持。在Java虚拟机中没有任何供boolean值专用的字节码指令，Java语言表达式所操作的boolean值，在编译之后都使用Java虚拟机中的int数据类型来代替，而boolean数组将会被编码成Java虚拟机的byte数组，每个元素boolean元素占8位”。这样我们可以得出boolean类型单独使用是4个字节，在数组中又是1个字节。那虚拟机为什么要用int来代替boolean呢？为什么不用byte或short，这样不是更节省内存空间吗？实际上，使用int的原因是，对于当下32位的CPU来说，一次进行32位的数据交换更加高效。

    综上，我们可以知道：官方文档对boolean类型没有给出精确的定义，《Java虚拟机规范》给出了“单独时使用4个字节，boolean数组时1个字节”的定义，具体还要看虚拟机实现是否按照规范来，所以1个字节、4个字节都是有可能的。这其实是一种时空权衡。

    boolean类型的封装类是Boolean。

（2）byte——1 byte——Byte

（3）short——2 bytes——Short

（4）int——4 bytes——Integer

（5）long——8 bytes——Long

（6）float——4 bytes——Float

（7）double——8 bytes——Double

（8）char——2 bytes——Character

2. 谈一谈”==“与”equals()"的区别。

    《Think in Java》中说：“关系操作符生成的是一个boolean结果，它们计算的是操作数的值之间的关系”。

    "=="判断的是两个对象的内存地址是否一样，适用于原始数据类型和枚举类型（它们的变量存储的是值本身，而引用类型变量存储的是引用）；equals是Object类的方法，Object对它的实现是比较内存地址，我们可以重写这个方法来自定义“相等”这个概念。比如类库中的String、Date等类就对这个方法进行了重写。

    综上，**对于枚举类型和原始数据类型的相等性比较，应该使用"=="；对于引用类型的相等性比较，应该使用equals方法。**

3. Java中的四种引用及其应用场景是什么？

**强引用:** 通常我们使用new操作符创建一个对象时所返回的引用即为强引用

**软引用:** 若一个对象只能通过软引用到达，那么这个对象在内存不足时会被回收，可用于图片缓存中，内存不足时系统会自动回收不再使用的Bitmap

**弱引用**: 若一个对象只能通过弱引用到达，那么它就会被回收（即使内存充足），同样可用于图片缓存中，这时候只要Bitmap不再使用就会被回收

**虚引用**: 虚引用是Java中最“弱”的引用，通过它甚至无法获取被引用的对象，**它存在的唯一作用就是当它指向的对象回收时，它本身会被加入到引用队列中，这样我们可以知道它指向的对象何时被销毁**。

4. object中定义了哪些方法？

    clone(), equals(), hashCode(), toString(), notify(), notifyAll(), wait(), finalize(), getClass()

5. hashCode的作用是什么？

    请参见散列表的基本原理与实现

6. ArrayList, LinkedList, Vector的区别是什么？

ArrayList: 内部采用数组存储元素，支持高效随机访问，支持动态调整大小，默认大小是10

LinkedList: 内部采用双向链表来存储元素，支持快速插入/删除元素，但不支持高效地随机访问

Vector: 可以看作线程安全版的ArrayList

7. String, StringBuilder, StringBuffer的区别是什么？

String: 不可变的字符序列，若要向其中添加新字符需要创建一个新的String对象

StringBuilder: 可变字符序列，支持向其中添加新字符（无需创建新对象）

StringBuffer: 可以看作线程安全版的StringBuilder，只是StringBuffer 中的方法大都采用了 synchronized 关键字进行修饰，因此是线程安全的，而 StringBuilder 没有这个修饰，可以被认为是线程不安全的。

jdk的实现中StringBuffer与StringBuilder都继承自AbstractStringBuilder，对于多线程的安全与非安全看到StringBuffer中方法前面的一堆synchronized就大概了解了。

这里随便讲讲AbstractStringBuilder的实现原理：我们知道使用StringBuffer等无非就是为了提高java中字符串连接的效率，因为直接使用+进行字符串连接的话，jvm会创建多个String对象，因此造成一定的开销。AbstractStringBuilder中采用一个char数组来保存需要append的字符串，char数组有一个初始大小，当append的字符串长度超过当前char数组容量时，则对char数组进行动态扩展，也即重新申请一段更大的内存空间，然后将当前char数组拷贝到新的位置，因为重新分配内存并拷贝的开销比较大，所以每次重新申请内存空间都是采用申请大于当前需要的内存空间的方式，这里是2倍。

8. Map, Set, List, Queue、Stack的特点及用法。

Map<K, V>:  Java中存储键值对的数据类型都实现了这个接口，表示“映射表”。支持的两个核心操作是get(Object key)以及put(K key, V value)，分别用来获取键对应的值以及向映射表中插入键值对。

Set<E>: 实现了这个接口的集合类型中不允许存在重复的元素，代表数学意义上的“集合”。它所支持的核心操作有add(E e),remove(Object o), contains(Object o)，分别用于添加元素，删除元素以及判断给定元素是否存在于集中。

List<E>: Java中集合框架中的列表类型都实现了这个接口，表示一种有序序列。支持get(int index), add(E e)等操作。

Queue<E>: Java集合框架中的队列接口，代表了“先进先出”队列。支持add(E element), remove()，offer(E element), poll()等操作。

Stack<E>: Java集合框架中表示堆栈的数据类型，堆栈是一种“后进先出”的数据结构。支持push(E item), pop()等操作。

    更详细的说明请参考官方文档，对相关数据结构不太熟悉的同学可以参考《算法导论》或其他相关书籍。

9. HashMap和HashTable的区别

HashTable是线程安全的，而HashMap不是

HashMap中允许存在null键和null值，而HashTable中不允许

    更加详细的分析请参考深入解析HashMap、HashTable

10. HashMap的实现原理

简单的说，HashMap的底层实现是“基于拉链法的散列表”。详细分析请参考深入解析HashMap、HashTable

Jdk1.8是数组+链表+红黑树

11. ConcurrentHashMap的实现原理

    ConcurrentHashMap是支持并发读写的HashMap，它的特点是读取数据时无需加锁，写数据时可以保证加锁粒度尽可能的小。由于其内部采用“分段存储”，只需对要进行写操作的数据所在的“段”进行加锁。关于ConcurrentHashMap底层实现的详细分析请参考Java并发编程：并发容器之ConcurrentHashMap

### 12. TreeMap, LinkedHashMap, HashMap的区别是什么？

**HashMap的底层实现是散列表，因此它内部存储的元素是无序的**；

**TreeMap的底层实现是红黑树，所以它内部的元素的有序的**。**排序的依据是自然序或者是创建TreeMap时所提供的比较器（Comparator）对象。**

**LinkedHashMap能够记住插入元素的顺序**。对于LinkedHashMap而言，它继承与HashMap、**底层使用哈希表与双向链表来保存所有元素。**

更加详细的说明请参考HashMap，LinkedMap，TreeMap的区别

13. Collection与Collections的区别是什么？

    Collection<E>是Java集合框架中的基本接口；Collections是Java集合框架提供的一个工具类，其中**包含了大量用于操作或返回集合的静态方法。**

    对Java集合框架还不太熟悉的小伙伴请参考Java核心技术点之集合框架

### 14. 对于“try-catch-finally”，若try语句块中包含“return”语句，finally语句块会执行吗？

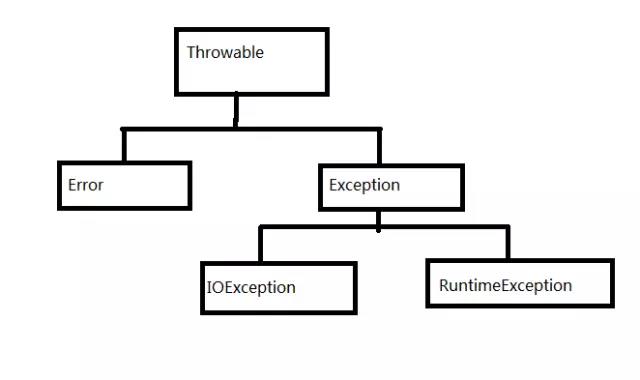
答案是会执行。只有两种情况finally块中的语句不会被执行：

**调用了System.exit()方法；**

**JVM“崩溃”了。**

### 15. Java中的异常层次结构

    Java中的异常层次结构如下图所示：



    我们可以看到**Throwable类是异常层级中的基类**。**Error类表示内部错误**，这类错误使我们无法控制的；Exception表示异常，**RuntimeException及其子类属于未检查异常**，这类异常包括**ArrayIndexOutOfBoundsException**、**NullPointerException**等，我们应该通过条件判断等方式语句避免未检查异常的发生。**IOException及其子类属于受检查异常，编译器会检查我们是否为所有可能抛出的受检查异常提供了异常处理器，若没有则会报错**。对于免检查异常，我们无需捕获（当然Java也允许我们捕获，但我们应该做的事避免未检查异常的发生）。

### 

## **ClassNotFoundException和NoClassDefFoundError的区别**

正如它们的名字所说明的：NoClassDefFoundError是一个错误(Error)，而ClassNOtFoundException是一个异常,受检查异常，在Java中错误和异常是有区别的，我们可以从异常中恢复程序但却不应该尝试从错误中恢复程序。

###### ClassNotFoundException

ClassNotFoundException的产生原因：就是找不到指定的class。 ClassNotFoundException发生在****装入阶段****。 当应用程序试图通过类的字符串名称，使用常规的三种方法装入类，但却找不到指定名称的类定义时就抛出该异常

常见的场景就是：

　　1 调用class的forName方法时，找不到指定的类

　　2 ClassLoader 中的 findSystemClass() 方法时，找不到指定的类

　　3 ClassLoader 中的 loadClass() 方法时，找不到指定的类

要解决这个问题很容易，唯一需要做的就是要确保所需的类连同它依赖的包存在于类路径中 .当Class.forName被调用的时候，类加载器会查找类路径中的类，如果找到了那么这个类就会被成功加载，如果没找到，那么就会抛出ClassNotFountException，除了Class.forName，ClassLoader.loadClass、ClassLOader.findSystemClass在动态加载类到内存中的时候也可能会抛出这个异常。

另外还有一个导致ClassNotFoundException的原因就是：当一个类已经被某个类加载器加载到内存中了，此时另一个类加载器又尝试着动态地从同一个包中加载这个类.

###### NoClassDefFoundError

这个就比较奇葩了，查找其他的资料是说，通过了编译，但是使用的时候，比如new的时候会出错。 当目前执行的类已经编译，但是找不到它的定义时，也就是说你如果编译了一个类B，在类A中调用，编译完成以后，你又删除掉B，运行A的时候那么就会出现这个错误

****加载时从外存储器找不到需要的class就出现ClassNotFoundException**** ****连接时从内存找不到需要的class就出现NoClassDefFoundError****

常见如下的场景：

　　1 类依赖的class或者jar不存在

　　2 类文件存在，但是存在不同的域中

　　3 大小写问题，javac编译的时候是无视大小的，很有可能你编译出来的class文件就与想要的不一样！

### 16. Java面向对象的三个特征与含义

     三大特征：**封装、继承、多态**。详细介绍请戳Java面向对象三大特性

### 17. Override, Overload的含义与区别

**Override表示“重写”**，是子类对父类中同一方法的重新定义

**Overload表示“重载”**，也就是定义一个与已定义方法名称相同但签名不同的新方法

### 18. 接口与抽象类的区别

    接口是一种约定，实现接口的类要遵循这个约定；抽象类本质上是一个类，使用抽象类的代价要比接口大。接口与抽象类的对比如下：

**抽象类中可以包含属性，方法（包含抽象方法与有着具体实现的方法），常量；接口只能包含常量和方法声明。Java 8中允许接口方法中有方法体。**

**抽象类中的方法和成员变量可以定义可见性**（比如public、private等）；而接口中的方法只能为public（缺省为public）。

**一个子类只能有一个父类（具体类或抽象类）**；而一个接口可以继承一个多个接口，一个类也可以实现多个接口。

**子类中实现父类中的抽象方法时，可见性可以大于等于父类中的；而接口实现类中的接口 方法的可见性只能与接口中相同（public）。**

### 19. 静态内部类与非静态内部类的区别

    静态内部类不会持有外围类的引用，而非静态内部类会隐式持有外围类的一个引用。

    欲进一步了解内部类，请戳Java核心技术点之内部类

### 20. Java中多态的实现原理

   所谓多态，指的就是父类引用指向子类对象，调用方法时会调用子类的实现而不是父类的实现。多态的实现的关键在于“动态绑定”。详细介绍请戳Java动态绑定的内部实现机制

### 21. 简述Java中创建新线程的两种方法

继承Thread类（假设子类为MyThread），并重写run()方法，然后new一个MyThread对象并对其调用start()即可启动新线程。

实现Runnable接口（假设实现类为MyRunnable），而后将MyRunnable对象作为参数传入Thread构造器，在得到的Thread对象上调用start()方法即可。

### 22. 简述Java中进行线程同步的方法

volatile: **Java**Memory Model保证了对同一个volatile变量的写happens before对它的读；

synchronized: 可以来对一个代码块或是对一个方法上锁，被“锁住”的地方称为临界区，进入临界区的线程会获取对象的monitor，这样其他尝试进入临界区的线程会因无法获取monitor而被阻塞。由于等待另一个线程释放monitor而被阻塞的线程无法被中断。

ReentrantLock:  尝试获取锁的线程可以被中断并可以设置超时参数。

    更加详细的介绍请戳Java核心技术点之多线程

### 23. 简述Java中具有哪几种粒度的锁

    Java中可以对类、对象、方法或是代码块上锁。更加详细的介绍请戳Java核心技术点之多线程

### 24. 给出“生产者-消费者”问题的一种解决方案

    使用阻塞队列：

IMG_257

public class BlockingQueueTest {

private int size = 20;

private ArrayBlockingQueue<Integer> blockingQueue = new ArrayBlockingQueue<>(size);

   public static void main(String[] args)  {

       BlockingQueueTest test = new BlockingQueueTest();

       Producer producer = test.new Producer();

       Consumer consumer = test.new Consumer();

       producer.start();

       consumer.start();

   }

   class Consumer extends Thread{

       @Override

public void run() {

while(true){

try {

//从阻塞队列中取出一个元素

queue.take();

System.out.println("队列剩余" + queue.size() + "个元素");

               } catch (InterruptedException e) {

               }

           }

       }

   }

   class Producer extends Thread{

       @Override        public void run() {

while (true) {

try {

//向阻塞队列中插入一个元素

queue.put(1);

                   System.out.println("队列剩余空间：" + (size - queue.size()));

               } catch (InterruptedException e) {

               }

           }

       }

   }

}

IMG_258

25. ThreadLocal的设计理念与作用

**ThreadLocal的作用是提供线程内的局部变量，在多线程环境下访问时能保证各个线程内的ThreadLocal变量各自独立。**也就是说，**每个线程的ThreadLocal变量是自己专用的，其他线程是访问不到的。**ThreadLocal最常用于以下这个场景：多线程环境下存在对非线程安全对象的并发访问，而且该对象不需要在线程间共享，但是我们不想加锁，ThreadLocal是如何做到为每一个线程维护变量的副本的呢？其实实现的思路很简单：**在ThreadLocal类中有一个Map，用于存储每一个线程的变量副本，Map中元素的键为线程对象，而值对应线程的变量副本。**

这时候可以使用ThreadLocal来使得每个线程都持有一个该对象的副本。

我们再看一下ThreadLocal类的源码：

这个是设值的方法

**[java]**

1. public void set(T value) {
2. Thread t = Thread.currentThread();
3. ThreadLocalMap map = getMap(t);
4. if (map != null)
5. map.set(this, value);
6. else
7. createMap(t, value);
8. }

public void set(T value) {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null)

map.set(this, value);

else

createMap(t, value);

}

 其中的getMap（）方法如下：

**[java]**

1. ThreadLocalMap getMap(Thread t) {
2. return t.threadLocals;
3. }

ThreadLocalMap getMap(Thread t) {

return t.threadLocals;

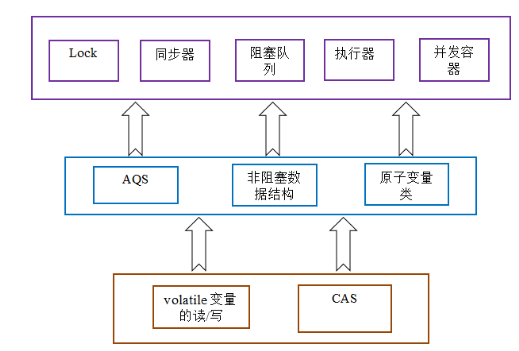
}

可以看到ThreadLocal操作值的时候是取得当前线程的ThreadLocalMap对象，然后把值设置到了这个对象中，这样对于不同的线程得到的就是不同的ThreadLocalMap，那么向其中保存值或者修改值都只是会影响到当前线程，这样就保证了线程安全。

    关于ThreadLocal的实现原理分析请戳深入剖析ThreadLocal

### 26. concurrent包的整体架构

896380f896380f

896380f

### 27. ArrayBlockingQueue, CountDownLatch类的作用

**CountDownLatch:  允许线程集等待直到计数器为0。适用场景: 当一个或多个线程需要等待指定数目的事件发生后再继续执行。**

**ArrayBlockingQueue:  一个基于数组实现的阻塞队列，它在构造时需要指定容量**。当试图向满队列中添加元素或者从空队列中移除元素时，当前线程会被阻塞。通过阻塞队列，我们可以按以下模式来工作：工作者线程可以周期性的将中间结果放入阻塞队列中，其它线程可以取出中间结果并进行进一步操作。若工作者线程的执行比较慢（还没来得及向队列中插入元素），其他从队列中取元素的线程会等待它（试图从空队列中取元素从而阻塞）；若工作者线程执行较快（试图向满队列中插入元素），则它会等待其它线程取出元素再继续执行。

### 28. wait()，sleep() 的区别

**wait():  Object类中定义的实例方法**。在指定对象上调用wait方法会让当前**线程进入等待状态**（前提是当前线程持有该对象的monitor），此时当前线程会释放相应对象的monitor，这样一来其它线程便有机会获取这个对象的monitor了。当其它线程获取了这个对象的monitor并进行了所需操作时，便可以调用notify方法唤醒之前进入等待状态的线程。

**sleep():  Thread类中的静态方法，作用是让当前线程进入休眠状态**，以便让其他线程有机会执行。进入休眠状态的线程**不会释放它所持有的锁。**

### 29. 线程池的用法与优势

优势:  **实现对线程的复用，避免了反复创建及销毁线程的开销**；使用线程池统一管理线程可以减少并发线程的数目，而线程数过多往往会在线程上下文切换上以及线程同步上浪费过多时间。

用法:  **我们可以调用ThreadPoolExecutor的某个构造方法来自己创建一个线程池。但通常情况下我们可以使用Executors类提供给我们的静态工厂方法来更方便的创建一个线程池对象**。创建了线程池对象后，我们就可以**调用submit方法提交任务到线程池中去执行了**；线程池使用完毕后我们要记得**调用shutdown方法来关闭它**。

   关于线程池的详细介绍以及实现原理分析请戳深入理解Java之线程池

### 30. for-each与常规for循环的效率对比

   关于这个问题我们直接看《Effective Java》给我们做的解答：

**for-each能够让代码更加清晰，并且减少了出错的机会**。下面的惯用代码适用于集合与数组类型：

for (Element e : elements) {

   doSomething(e);  
}

**使用for-each循环与常规的for循环相比，并不存在性能损失**，即使对数组进行迭代也是如此。实际上，**在有些场合下它还能带来微小的性能提升，因为它只计算一次数组索引的上限。**

### 31. 简述Java IO与NIO的区别

**Java IO是面向流的**，这意味着我们需要每次从流中读取一个或多个字节，直到读取完所有字节；**NIO是面向缓冲的，也就是说会把数据读取到一个缓冲区中，然后对缓冲区中的数据进行相应处理**。

**Java IO是阻塞IO，而NIO是非阻塞IO。**

**Java NIO中存在一个称为选择器（selector）的东西，它允许你把多个通道（channel）注册到一个选择器上，然后使用一个线程来监视这些通道**：若这些通道里有某个准备好可以开始进行读或写操作了，则开始对相应的通道进行读写。**而在等待某通道变为可读/写期间，请求对通道进行读写操作的线程可以去干别的事情。**

    更进一步的说明请戳Java NIO与IO

### 

### 32. 反射的作用与原理

    反射的作用概括地说是运行时获取类的各种定义信息，比如定义了哪些属性与方法。**原理是通过类的class对象来获取它的各种信息。**

    详细介绍请参见Java核心技术点之反射

### 33. Java中的泛型机制

    关于泛型机制的详细介绍请直接戳Java核心技术点之泛型

 Java中的泛型基本上都是在编译器这个层次来实现的。在生成的Java字节码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，编译器在编译的时候去掉。这个过程就称为类型擦除。**1、先检查，在编译，以及检查编译的对象和引用传递的问题1、先检查，在编译，以及检查编译的对象和引用传递的问题**

既然说类型变量会在编译的时候擦除掉，那为什么我们往ArrayList<String> arrayList=new ArrayList<String>();所创建的数组列表arrayList中，不能使用add方法添加整形呢？不是说泛型变量Integer会在编译时候擦除变为原始类型Object吗，为什么不能存别的类型呢？既然类型擦除了，如何保证我们只能使用泛型变量限定的类型呢？

java是如何解决这个问题的呢？**java编译器是通过先检查代码中泛型的类型，然后再进行类型擦除，再进行编译的**。

List< Object > - List can contain Object or it’s subtype

List< ? extends Number > - List can contain Number or its subtypes.

List< ? super Number > - List can contain Number or its supertypes.

在Java中不允许创建泛型数组

我们无法对泛型代码直接使用instanceof关键字，因为Java编译器在生成代码的时候会擦除所有相关泛型的类型信息，正如我们上面验证过的JVM在运行时期无法识别出ArrayList<Integer>和ArrayList<String>的之间的区别

**Java中List<Object>和原始类型List之间的区别?**

原始类型和带参数类型<Object>之间的主要区别是，在编译时编译器不会对原始类型进行类型安全检查，却会对带参数的类型进行检 查，通过使用Object作为类型，可以告知编译器该方法可以接受任何类型的对象，比如String或Integer。这道题的考察点在于对泛型中原始类 型的正确理解。它们之间的第二点区别是，你可以把任何带参数的类型传递给原始类型List，但却不能把List<String>传递给接受 List<Object>的方法，因为会产生编译错误。

**Java中List<?>和List<Object>之间的区别是什么?**

这道题跟上一道题看起来很像，实质上却完全不同。List<?> 是一个未知类型的List，而List<Object> 其实是任意类型的List。你可以把List<String>, List<Integer>赋值给List<?>，却不能把List<String>赋值给 List<Object>。

List<?> listOfAnyType;

List<Object> listOfObject = new ArrayList<Object>();

List<String> listOfString = new ArrayList<String>();

List<Integer> listOfInteger = new ArrayList<Integer>();

listOfAnyType = listOfString; //legal

listOfAnyType = listOfInteger; //legal

listOfObjectType = (List<Object>) listOfString; //compiler error – in-convertible types

**List<String>和原始类型List之间的区别.**

该题类似于“原始类型和带参数类型之间有什么区别”。带参数类型是类型安全的，而且其类型安全是由编译器保证的，但原始类型List却不是类型安全 的。你不能把String之外的任何其它类型的Object存入String类型的List中，而你可以把任何类型的对象存入原始List中。使用泛型的 带参数类型你不需要进行类型转换，但是对于原始类型，你则需要进行显式的类型转换。

List listOfRawTypes = new ArrayList();

listOfRawTypes.add(“abc”);

listOfRawTypes.add(123); //编译器允许这样 – 运行时却会出现异常

String item = (String) listOfRawTypes.get(0); //需要显式的类型转换

item = (String) listOfRawTypes.get(1); //抛ClassCastException，因为Integer不能被转换为String

List<String> listOfString = new ArrayList();

listOfString.add(“abcd”);

listOfString.add(1234); //编译错误，比在运行时抛异常要好

item = listOfString.get(0); //不需要显式的类型转换 – 编译器自动转换

Java 泛型是由类型擦除来实现的？有以下两个原因：

1).避免 JVM 的大换血。如果 JVM 将泛型类型延续到运行期，那么到运行期时 JVM 就需要进行大量的重构工作了，提高了运行期的效率。

2). 版本兼容。 在编译期擦除可以更好地支持原生类型（Raw Type）。

**1. [Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/andychiu/article/details/_blank)中的泛型是什么 ? 使用泛型的好处是什么?**

这是在各种Java泛型[面试](http://yjbys.com/mianshi/" \t "http://blog.csdn.net/andychiu/article/details/_blank)中，一开场你就会被问到的问题中的一个，主要集中在初级和中级面试中。那些拥有Java1.4或更早版本的开发背景的人 都知道，在集合中存储对象并在使用前进行类型转换是多么的不方便。泛型防止了那种情况的发生。它提供了编译期的类型安全，确保你只能把正确类型的对象放入 集合中，避免了在运行时出现ClassCastException。

**2. Java的泛型是如何工作的 ? 什么是类型擦除 ?**

这是一道更好的泛型面试题。泛型是通过类型擦除来实现的，编译器在编译时擦除了所有类型相关的信息，所以在运行时不存在任何类型相关的信息。例如 List<String>在运行时仅用一个List来表示。这样做的目的，是确保能和**[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/andychiu/article/details/_blank)** 5之前的版本开发二进制类库进行兼容。你无法在运行时访问到类型参数，因为编译器已经把泛型类型转换成了原始类型。根据你对这个泛型问题的回答情况，你会 得到一些后续提问，比如为什么泛型是由类型擦除来实现的或者给你展示一些会导致编译器出错的错误泛型代码。

**3. 什么是泛型中的限定通配符和非限定通配符 ?**

这是另一个非常流行的Java泛型面试题。限定通配符对类型进行了限制。有两种限定通配符，一种是<? extends T>它通过确保类型必须是T的子类来设定类型的上界，另一种是<? super T>它通过确保类型必须是T的父类来设定类型的下界。泛型类型必须用限定内的类型来进行初始化，否则会导致编译错误。另一方面<?>表示了非限定通配符，因为<?>可以用任意类型来替代。更多信息请参阅我的文章泛型中限定通配符和非限定通配符之间的区别。

**4. List<? extends T>和List <? super T>之间有什么区别 ?**

这和上一个面试题有联系，有时面试官会用这个问题来评估你对泛型的理解，而不是直接问你什么是限定通配符和非限定通配符。这两个List的声明都是 限定通配符的例子，List<? extends T>可以接受任何继承自T的类型的List，而List<? super T>可以接受任何T的父类构成的List。例如List<? extends Number>可以接受List<Integer>或List<Float>。在本段出现的连接中可以找到更多信息。

**5. 如何编写一个泛型方法，让它能接受泛型参数并返回泛型类型?**

编写泛型方法并不困难，你需要用泛型类型来替代原始类型，比如使用T, E or K,V等被广泛认可的类型占位符。泛型方法的例子请参阅Java集合类框架。最简单的情况下，一个泛型方法可能会像这样:

public V put(K key, V value) {

return cache.put(key, value);

}

**6. Java中如何使用泛型编写带有参数的类?**

这是上一道面试题的延伸。面试官可能会要求你用泛型编写一个类型安全的类，而不是编写一个泛型方法。关键仍然是使用泛型类型来代替原始类型，而且要使用JDK中采用的标准占位符。

**7. 编写一段泛型程序来实现LRU缓存?**

对于喜欢Java编程的人来说这相当于是一次练习。给你个提示，LinkedHashMap可以用来实现固定大小的LRU缓存，当LRU缓存已经满 了的时候，它会把最老的键值对移出缓存。LinkedHashMap提供了一个称为removeEldestEntry()的方法，该方法会被put() 和putAll()调用来删除最老的键值对。当然，如果你已经编写了一个可运行的JUnit**[测试](http://lib.csdn.net/base/softwaretest" \o "软件测试知识库" \t "http://blog.csdn.net/andychiu/article/details/_blank)**，你也可以随意编写你自己的实现代码。

**8. 你可以把List<String>传递给一个接受List<Object>参数的方法吗？**

对任何一个不太熟悉泛型的人来说，这个Java泛型题目看起来令人疑惑，因为乍看起来String是一种Object，所以 List<String>应当可以用在需要List<Object>的地方，但是事实并非如此。真这样做的话会导致编译错误。如 果你再深一步考虑，你会发现Java这样做是有意义的，因为List<Object>可以存储任何类型的对象包括String, Integer等等，而List<String>却只能用来存储Strings。

List<Object> objectList;

List<String> stringList;

objectList = stringList; //compilation error incompatible types

**9. Array中可以用泛型吗?**

这可能是Java泛型面试题中最简单的一个了，当然前提是你要知道Array事实上并不支持泛型，这也是为什么Joshua Bloch在Effective Java一书中建议使用List来代替Array，因为List可以提供编译期的类型安全保证，而Array却不能。

**10. 如何阻止Java中的类型未检查的警告?**

如果你把泛型和原始类型混合起来使用，例如下列代码，**[java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/andychiu/article/details/_blank)** 5的javac编译器会产生类型未检查的警告，例如

List<String> rawList = new ArrayList()

注意: Hello.java使用了未检查或称为不安全的操作;

这种警告可以使用@SuppressWarnings(“unchecked”)注解来屏蔽。

### 34. Java 7与Java 8的新特性

    这里有两篇总结的非常好的：Java 7的新特性 Java 8的新特性

### 35. 常见设计模式

   所谓“设计模式”，不过是面向对象编程中一些常用的软件设计手法，并且经过实践的检验，这些设计手法在各自的场景下能解决一些需求，因此它们就成为了如今广为流传的”设计模式“。也就是说，正式因为在某些场景下产生了一些棘手的问题，才催生了相应的设计模式。明确了这一点，我们在学习某种设计模式时要充分理解它产生的背景以及它所解决的主要矛盾是什么。

   常用的设计模式可以分为以下三大类：

创建型模式:  包括工厂模式（又可进一步分为简单工厂模式、工厂方法模式、抽象工厂模式）、建造者模式、单例模式。

结构型模式:  包括适配器模式、桥接模式、装饰模式、外观模式、享元模式、代理模式。

行为型模式:  包括命令模式、中介者模式、观察者模式、状态模式、策略模式。

    关于每个模式具体的介绍请参考图说设计模式

### 36. JNI的基本用法

    关于JNI，这里有篇好文：Android中的JNI

### 37. 动态代理的定义、应用场景及原理

    关于动态代理，请直接参见Java核心技术点之动态代理

### 

### 38. 注解的基本概念与使用

注解可以看作是“增强版的注释”，它可以向编译器、虚拟机说明一些事情。

**注解是描述Java代码的代码，它能够被编译器解析**，注解处理工具在运行时也能够解析注解。注解本身是“被动”的信息，只有主动解析它才有意义。

除了向编译器/虚拟机传递信息，我们也可以使用注解来生成一些“模板化”的代码。

39. 我们为编写的类重写hashCode方法时，可能会看到如下所示的代码，其实我们**不太理解为什么要使用这样的乘法运算来产生哈希码（散列码），而且为什么这个数是个素数，为什么通常选择31这个数？**前两个问题的答案你可以自己百度一下，**选择31是因为可以用移位和减法运算来代替乘法，从而得到更好的性能。**说到这里你可能已经想到了：**31 \* num 等价于(num << 5) – num**，左移5位相当于乘以2的5次方再减去自身就相当于乘以31，**现在的VM都能自动完成这个优化**。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | public class PhoneNumber {      private int areaCode;      private String prefix;      private String lineNumber;        @Override      public int hashCode() {          final int prime = 31;          int result = 1;          result = prime \* result + areaCode;          result = prime \* result                  + ((lineNumber == null) ? 0 : lineNumber.hashCode());          result = prime \* result + ((prefix == null) ? 0 : prefix.hashCode());          return result;      }        @Override      public boolean equals(Object obj) {          if (this == obj)              return true;          if (obj == null)              return false;          if (getClass() != obj.getClass())              return false;          PhoneNumber other = (PhoneNumber) obj;          if (areaCode != other.areaCode)              return false;          if (lineNumber == null) {              if (other.lineNumber != null)                  return false;          } else if (!lineNumber.equals(other.lineNumber))              return false;          if (prefix == null) {              if (other.prefix != null)                  return false;          } else if (!prefix.equals(other.prefix))              return false;          return true;      }    } |
|  |  |

1. 基数要用质数  
   **质数的特性（只有1和自己是因子）能够使得它和其他数相乘后得到的结果比其他方式更容易产成唯一性，也就是hash code值的冲突概率最小。**2.选择31是观测分布结果后的一个选择，不清楚原因，但的确有利。

我们会注意那个狗血的31这个系数为什么总是在里面乘来乘去？为什么不适用32或者其他数字？

大家都知道，计算机的乘法涉及到移位计算。当一个数乘以2时，就直接拿该数左移一位即可！选择31原因是因为31是一个素数！

所谓素数：

**质数又称素数。指在一个大于1的自然数中，除了1和此整数自身外，没法被其他自然数整除的数。**

在存储数据计算hash地址的时候，我们希望尽量减少有同样的hash地址，所谓“冲突”。如果使用相同hash地址的数据过多，那么这些数据所组成的hash链就更长，从而降低了查询效率！所以在选择系数的时候要选择尽量长(31 = 11111[2])的系数并且让乘法尽量不要溢出(如果选择大于11111的数，很容易溢出)的系数，因为如果计算出来的hash地址越大，所谓的“冲突”就越少，查找起来效率也会提高。

31可以 由i\*31== (i<<5)-1来表示，现在很多虚拟机里面都有做相关优化，使用31的原因可能是为了更好的分配hash地址，并且31只占用5bits！

在java乘法中如果数字相乘过大会导致溢出的问题，从而导致数据的丢失.

而31则是素数（质数）而且不是很长的数字，最终它被选择为相乘的系数的原因不过与此！

[http://computinglife.wordpress.com/2008/11/20/why-do-hash-functions-use-prime-numbers/](http://computinglife.wordpress.com/2008/11/20/why-do-hash-functions-use-prime-numbers/" \t "http://blog.csdn.net/steveguoshao/article/details/_blank)

另外，String.hashCode内部会缓存第一次计算的值，因为这是一个final（不可变）类，也就是String对象的内容是不会变的。这能够在多次put到HashMap的场合提高性能，不过似乎用处不多。

1.hashcode是用来查找的，如果你学过数据结构就应该知道，在查找和排序这一章有

例如内存中有这样的位置

0  1  2  3  4  5  6  7

而我有个类，这个类有个字段叫ID,我要把这个类存放在以上8个位置之一，如果不用hashcode而任意存放，那么当查找时就需要到这八个位置里挨个去找，或者用二分法一类的算法。

但如果用hashcode那就会使效率提高很多。  我们这个类中有个字段叫ID,那么我们就定义我们的hashcode为ID％8，然后把我们的类存放在取得得余数那个位置。比如我们的ID为9，9除8的余数为1，那么我们就把该类存在1这个位置，如果ID是13，求得的余数是5，那么我们就把该类放在5这个位置。这样，以后在查找该类时就可以通过ID除 8求余数直接找到存放的位置了。

1. 但是如果两个类有相同的hashcode怎么办那（我们假设上面的类的ID不是唯一的），例如9除以8和17除以8的余数都是1，那么这是不是合法的，回答是：可以这样。那么如何判断呢？在这个时候就需要定义 equals了。  也就是说，我们先通过 hashcode来判断两个类是否存放某个桶里，但这个桶里可能有很多类，那么我们就需要再通过 equals 来在这个桶里找到我们要的类。  那么。重写了equals()，为什么还要重写hashCode()呢？  想想，你要在一个桶里找东西，你必须先要找到这个桶啊，你不通过重写hashcode()来找到桶，光重写equals()有什么用啊
2. （Effective Java）首先equals方法必须满足自反性（x.equals(x)必须返回true）、对称性（x.equals(y)返回true时，y.equals(x)也必须返回true）、传递性（x.equals(y)和y.equals(z)都返回true时，x.equals(z)也必须返回true）和一致性（当x和y引用的对象信息没有被修改时，多次调用x.equals(y)应该得到同样的返回值），而且对于任何非null值的引用x，x.equals(null)必须返回false。实现高质量的equals方法的诀窍包括：1. 使用==操作符检查”参数是否为这个对象的引用”；2. 使用instanceof操作符检查”参数是否为正确的类型”；3. 对于类中的关键属性，检查参数传入对象的属性是否与之相匹配；4. 编写完equals方法后，问自己它是否满足对称性、传递性、一致性；5. 重写equals时总是要重写hashCode；6. 不要将equals方法参数中的Object对象替换为其他的类型，在重写时不要忘掉@Override注解。

[面向对象设计之------Is-A(继承关系)、Has-A(合成关系，组合关系)和Use-A(依赖关系)](http://www.cnblogs.com/feichengwulai/articles/3632537.html)

1，**Is-A继承关系**：“**表示类与类之间的继承关系、接口与接口之间的继承的关系以及类对接口实现的关系”。如：**

---对象的继承关系是在编译时就定好了，所以无法在运行时改变从父类继承的实现。子类的实现与它的父类有非常紧密的依赖关系，以至于父类实现中的任何变化必然会导致子类发生变化。当你需要复用子类时，如果继承下来的实现不适合解决新的问题，则父类必须重写或被其他更适合的类替换。**这种依赖关系限制了灵活性并最终限制了复用性。**

2，**Has-A合成关系：“是关联关系的一种**，**是整体和部分（通常为一个私有的变量）之间的关系**，**并且代表的整体对象负责构建和销毁代表部分对象，代表部分的对象不能共享**”。如：

    public class Heart

    {

       ……

    }

    public class Man

    {

        private Heart heart = new Heart();

       ……

    }

Man类和Heart类便是Has-A关系，人死的时候心也就死了（当然有心脏移植，不想抬杠，哈哈），人和心是一种比较强的拥有关系，并且心是不能共享的。

3，**Use-A依赖关系：“是类与类（通常为函数的参数）之间的连接，依赖总是单向的”**。如

     public abstract class Course

    {

        ……

    }

    public class Student

    {

        public void Learn(Course course)

        {

           ……

        }

   }

Student类和Course的关系就是Use-A关系，Student类总是单向指向Course，学生知道自己学的是什么课程，而课程根本不关心它被哪个学生学习。

继承是一种强耦合的结构，父类变，子类就必须要变

---is-a，一个圆形就是一个几何形状。圆形可以继承几何形状。几何形状是圆形的抽象。

**什么情况下用+运算符进行字符串连接比调用StringBuffer/StringBuilder对象的append方法连接字符串性能更好？**

答：如果使用少量的字符串操作，使用 (+运算符)连接字符串；  
如果频繁的对大量字符串进行操作，则使用  
1：全局变量或者需要多线程支持则使用StringBuffer；  
2：局部变量或者单线程不涉及[线程安全](https://www.baidu.com/s?wd=%E7%BA%BF%E7%A8%8B%E5%AE%89%E5%85%A8&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3my7WuHDvrHTsnWP-P1D30ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1c4n1cLPWfLnjbLn1bknWfd" \t "https://zhidao.baidu.com/question/_blank)则使有StringBuilder。

如果在编写代码的过程中大量使用+进行字符串还是会对性能造成比较大的影响，但是使用的个数在1000以下还是可以接受的，大于10000的话，执行时间将可能超过1s，会对性能产生较大影响。如果有大量需要进行字符串拼接的操作，最好还是使用StringBuffer或StringBuilder进行。

**41、日期和时间：**

**- 如何取得年月日、小时分钟秒？**

**- 如何取得从1970年1月1日0时0分0秒到现在的毫秒数？**

**- 如何取得某月的最后一天？**

**- 如何格式化日期？**

答：

问题1：创建**java.util.Calendar 实例**，调用其get()方法传入不同的参数即可获得参数所对应的值。**Java 8中可以使用java.time.LocalDateTimel来获取**，代码如下所示。

public class DateTimeTest {

    public static void main(String[] args) {

        Calendar cal = Calendar.getInstance();

        System.out.println(cal.get(Calendar.YEAR));

        System.out.println(cal.get(Calendar.MONTH));    // 0 - 11

        System.out.println(cal.get(Calendar.DATE));

        System.out.println(cal.get(Calendar.HOUR\_OF\_DAY));

        System.out.println(cal.get(Calendar.MINUTE));

        System.out.println(cal.get(Calendar.SECOND));

        // Java 8

        LocalDateTime dt = LocalDateTime.now();

        System.out.println(dt.getYear());

        System.out.println(dt.getMonthValue());     // 1 - 12

        System.out.println(dt.getDayOfMonth());

        System.out.println(dt.getHour());

        System.out.println(dt.getMinute());

        System.out.println(dt.getSecond());

    }

}

问题2：以下方法均可获得该毫秒数。

Calendar.getInstance().getTimeInMillis();

System.currentTimeMillis();

Clock.systemDefaultZone().millis(); // Java 8

问题3：代码如下所示。

Calendar time = Calendar.getInstance();

time.getActualMaximum(Calendar.DAY\_OF\_MONTH);

问题4：利用**java.text.DataFormat 的子类（如SimpleDateFormat类**）中的format(Date)方法可将日期格式化。Java 8中可以用**java.time.format.DateTimeFormatter**来格式化时间日期，代码如下所示。

import java.text.SimpleDateFormat;

import java.time.LocalDate;

import java.time.format.DateTimeFormatter;

import java.util.Date;

class DateFormatTest {

    public static void main(String[] args) {

        SimpleDateFormat oldFormatter = new SimpleDateFormat("yyyy/MM/dd");

        Date date1 = new Date();

        System.out.println(oldFormatter.format(date1));

        // Java 8

        DateTimeFormatter newFormatter = **DateTimeFormatter.ofPattern**("yyyy/MM/dd");

        LocalDate date2 = LocalDate.now();

        System.out.println(date2.format(newFormatter));

    }

}

补充：Java的时间日期API一直以来都是被诟病的东西，为了解决这一问题，Java 8中引入了新的时间日期API，其中包括**LocalDate、LocalTime、LocalDateTime、Clock、Instant，Duration,DateTimeFormatter**等类，这些的类的设计都使用了不变模式，因此是线程安全的设计。

42、打印昨天的当前时刻。

答：

import java.util.Calendar;

class YesterdayCurrent {

    public static void main(String[] args){

        Calendar cal = Calendar.getInstance();

        cal.add(Calendar.DATE, -1);

        System.out.println(cal.getTime());

    }

}

在Java 8中，可以用下面的代码实现相同的功能。

import java.time.LocalDateTime;

class YesterdayCurrent {

    public static void main(String[] args) {

        LocalDateTime today = LocalDateTime.now();

        LocalDateTime yesterday = today.minusDays(1);

        System.out.println(yesterday);

    }

}

**44、什么时候用断言（assert）？**

答：**断言在软件开发中是一种常用的调试方式，很多开发语言中都支持这种机制**。一般来说，断言用于保证程序最基本、关键的正确性。**断言检查通常在开发和测试时开启。为了保证程序的执行效率，在软件发布后断言检查通常是关闭的。**断言是一个包含布尔表达式的语句，在执行这个语句时假定该表达式为true；如果表达式的值为false，那么系统会报告一个AssertionError。断言的使用如下面的代码所示：

assert(a > 0); // throws an AssertionError if a <= 0

断言可以有两种形式：

assert Expression1;

assert Expression1 : Expression2 ;

Expression1 应该总是产生一个布尔值。

Expression2 可以是得出一个值的任意表达式；这个值用于生成显示更多调试信息的字符串消息。

**要在运行时启用断言，可以在启动JVM时使用-enableassertions或者-ea标记**。**要在运行时选择禁用断言，可以在启动JVM时使用-da或者-disableassertions标记。**要在系统类中启用或禁用断言，可使用-esa或-dsa标记。还可以在包的基础上启用或者禁用断言。

注意：断言不应该以任何方式改变程序的状态。简单的说，如果希望在不满足某些条件时阻止代码的执行，就可以考虑用断言来阻止它。

**48、运行时异常与受检异常有何异同？**

答：**异常表示程序运行过程中可能出现的非正常状态**，**运行时异常表示虚拟机的通常操作中可能遇到的异常，是一种常见运行错误，只要程序设计得没有问题通常就不会发生**。**受检异常跟程序运行的上下文环境有关**，即使程序设计无误，仍然可能因使用的问题而引发。**Java编译器要求方法必须声明抛出可能发生的受检异常，但是并不要求必须声明抛出未被捕获的运行时异常。**异常和继承一样，是面向对象程序设计中经常被滥用的东西，在Effective Java中对异常的使用给出了以下指导原则：

- 不要将异常处理用于正常的控制流（设计良好的API不应该强迫它的调用者为了正常的控制流而使用异常）

- 对可以恢复的情况使用受检异常，对编程错误使用运行时异常

- **避免不必要的使用受检异常**（可以通过一些状态检测手段来避免异常的发生）

- **优先使用标准的异常**

- 每个方法抛出的异常都要有文档

- 保持异常的原子性

- **不要在catch中忽略掉捕获到的异常**

**49、列出一些你常见的运行时异常？**

答：

- **ArithmeticException**（算术异常）

- **ClassCastException** （类转换异常）

- **IllegalArgumentException** （非法参数异常）

- **IndexOutOfBoundsException** （下标越界异常）

- **NullPointerException** （空指针异常）

- SecurityException （安全异常）

**66、线程的基本状态以及状态之间的关系？**

答：



说明：其中Running表示运行状态，Runnable表示就绪状态（万事俱备，只欠CPU），Blocked表示阻塞状态，阻塞状态又有多种情况，可能是因为调用wait()方法进入等待池，也可能是执行同步方法或同步代码块进入等锁池，或者是调用了sleep()方法或join()方法等待休眠或其他线程结束，或是因为发生了I/O中断。

**67、简述synchronized 和java.util.concurrent.locks.Lock的异同？**

答：Lock是Java 5以后引入的新的API，和关键字synchronized相比主要相同点：Lock 能完成synchronized所实现的所有功能；主要不同点：Lock有比synchronized更精确的线程语义和更好的性能，而且不强制性的要求一定要获得锁。synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且最好在finally 块中释放（这是释放外部资源的最好的地方）。

**68、Java中如何实现序列化，有什么意义？**

答：**序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化**。**可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决对象流读写操作时可能引发的问题（如果不进行序列化可能会存在数据乱序的问题）。**

**要实现序列化，需要让一个类实现Serializable接口，该接口是一个标识性接口，标注该类对象是可被序列化的**，然后使用一个输出流来构造一个对象输出流并通过writeObject(Object)方法就可以将实现对象写出（即保存其状态）；如果需要反序列化则可以用一个输入流建立对象输入流，然后通过readObject方法从流中读取对象。**序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆**（可以参考第29题）。

**69、Java中有几种类型的流？**

答：**字节流和字符流**。**字节流继承于InputStream、OutputStream**，**字符流继承于Reader、Writer**。在java.io 包中还有许多其他的流，主要是为了提高性能和使用方便。关于Java的I/O需要注意的有两点：一是两种对称性（输入和输出的对称性，字节和字符的对称性）；二是两种设计模式（适配器模式和装饰器模式）。另外Java中的流不同于C#的是它只有一个维度一个方向。

面试题 - 编程实现文件拷贝。（这个题目在笔试的时候经常出现，下面的代码给出了两种实现方案）

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import java.nio.ByteBuffer;

import java.nio.channels.FileChannel;

public final class MyUtil {

    private MyUtil() {

        throw new AssertionError();

    }

    public static void fileCopy(String source, String target) throws IOException {

        try (InputStream in = new FileInputStream(source)) {

            try (OutputStream out = new FileOutputStream(target)) {

                byte[] buffer = new byte[4096];

                int bytesToRead;

                while((bytesToRead = in.read(buffer)) != -1) {

                    out.write(buffer, 0, bytesToRead);

                }

            }

        }

    }

    public static void fileCopyNIO(String source, String target) throws IOException {

        try (FileInputStream in = new FileInputStream(source)) {

            try (FileOutputStream out = new FileOutputStream(target)) {

                FileChannel inChannel = in.getChannel();

                FileChannel outChannel = out.getChannel();

                ByteBuffer buffer = ByteBuffer.allocate(4096);

                while(inChannel.read(buffer) != -1) {

                    buffer.flip();

                    outChannel.write(buffer);

                    buffer.clear();

                }

            }

        }

    }

}

注意：上面用到Java 7的TWR，使用TWR后可以不用在finally中释放外部资源 ，从而让代码更加优雅。

**70、写一个方法，输入一个文件名和一个字符串，统计这个字符串在这个文件中出现的次数**。

答：代码如下：

import java.io.BufferedReader;

import java.io.FileReader;

public final class MyUtil {

    // 工具类中的方法都是静态方式访问的因此将构造器私有不允许创建对象(绝对好习惯)

    private MyUtil() {

        throw new AssertionError();

    }

    /\*\*

     \* 统计给定文件中给定字符串的出现次数

     \*

     \* @param filename  文件名

     \* @param word 字符串

     \* @return 字符串在文件中出现的次数

     \*/

    public static int countWordInFile(String filename, String word) {

        int counter = 0;

        try (FileReader fr = new FileReader(filename)) {

            try (BufferedReader br = new BufferedReader(fr)) {

                String line = null;

                while ((line = br.readLine()) != null) {

                    int index = -1;

                    while (line.length() >= word.length() && (index = line.indexOf(word)) >= 0) {

                        counter++;

                        line = line.substring(index + word.length());

                    }

                }

            }

        } catch (Exception ex) {

            ex.printStackTrace();

        }

        return counter;

    }

}

**71、如何用Java代码列出一个目录下所有的文件？**

答：

如果只要求列出当前文件夹下的文件，代码如下所示：

import java.io.File;

class Test12 {

    public static void main(String[] args) {

        File f = new File("/Users/Hao/Downloads");

        for(File temp : f.listFiles()) {

            if(temp.isFile()) {

                System.out.println(temp.getName());

            }

        }

    }

}

如果需要对文件夹继续展开，代码如下所示：

import java.io.File;

class Test12 {

    public static void main(String[] args) {

        showDirectory(new File("/Users/Hao/Downloads"));

    }

    public static void showDirectory(File f) {

        \_walkDirectory(f, 0);

    }

    private static void \_walkDirectory(File f, int level) {

        if(f.isDirectory()) {

            for(File temp : f.listFiles()) {

                \_walkDirectory(temp, level + 1);

            }

        }

        else {

            for(int i = 0; i < level - 1; i++) {

                System.out.print("\t");

            }

            System.out.println(f.getName());

        }

    }

}

在Java 7中可以使用NIO.2的API来做同样的事情，代码如下所示：

class ShowFileTest {

    public static void main(String[] args) throws IOException {

        Path initPath = Paths.get("/Users/Hao/Downloads");

        Files.walkFileTree(initPath, new SimpleFileVisitor<Path>() {

            @Override

            public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)

                    throws IOException {

                System.out.println(file.getFileName().toString());

                return FileVisitResult.CONTINUE;

            }

        });

    }

}

**73、XML文档定义有几种形式？它们之间有何本质区别？解析XML文档有哪几种方式？**

答：**XML文档定义分为DTD和Schema两种形式**，**二者都是对XML语法的约束，其本质区别在于Schema本身也是一个XML文件，可以被XML解析器解析，而且可以为XML承载的数据定义类型，约束能力较之DTD更强大**。对XML的解析主要有**DOM（文档对象模型，Document Object Model）、SAX（Simple API for XML）和StAX（Java 6中引入的新的解析XML的方式，Streaming API for XML）**，其中DOM处理大型文件时其性能下降的非常厉害，这个问题是由DOM树结构占用的内存较多造成的，而且DOM解析方式必须在解析文件之前把整个文档装入内存，适合对XML的随机访问（典型的用空间换取时间的策略）；**SAX是事件驱动型的XML解析方式，它顺序读取XML文件，不需要一次全部装载整个文件**。当遇到像文件开头，文档结束，或者**标签开头与标签结束时，它会触发一个事件，用户通过事件回调代码来处理XML文件，适合对XML的顺序访问**；顾名思义，**StAX把重点放在流上，实际上StAX与其他解析方式的本质区别就在于应用程序能够把XML作为一个事件流来处理。**将XML作为一组事件来处理的想法并不新颖（SAX就是这样做的），但不**同之处在于StAX允许应用程序代码把这些事件逐个拉出来，而不用提供在解析器方便时从解析器中接收事件的处理程序**。**一种利用拉模式解析 (pull-parsing)XML** 文档的 API.**StAX 通过提供一种基于事件迭代器(Iterator) 的 API 让程序员去控制 xml 文档解析过程**,程序遍历这个事件迭代器去处理每一 个解析事件，解析事件可以看做是程序拉出来的，也就是程序促使解析器产生一 个解析事件然后处理该事件，之后又促使解析器产生下一个解析事件，如此循环 直到碰到文档结束符；**SAX 也是基于事件处理 xml 文档，但却是用推模式解析， 解析器解析完整个 xml 文档后， 才产生解析事件， 然后推给程序去处理这些事件**； DOM 采用的方式是将整个 xml 文档映射到一颗内存树，这样就可以很容易地得到 父节点和子结点以及兄弟节点的数据，但如果文档很大，将会严重影响性能。

1）**DOM4J 性能最好**，连 Sun 的 JAXM 也在用 DOM4J.目前许多开源项目中大量采用 DOM4J，例如**大名鼎鼎的 hibernate 也用 DOM4J 来读取 XML 配置文件。如果不考 虑可移植性，那就采用 DOM4J**.

2）JDOM 和 DOM 在性能测试时表现不佳，在测试 10M 文档时内存溢出。在小文档 情况下还值得考虑使用 DOM 和 JDOM.虽然 JDOM 的开发者已经说明 他们期望在正 式发行版前专注性能问题，但是从性能观点来看，它确实没有值得推荐之处。另 外，DOM 仍是一个非常好的选择。DOM 实现广泛应用于多种编程语 言。它还是许 多其它与 XML 相关的标准的基础，因为它正式获得 W3C 推荐（与基于非标准的 Java 模型相对），所以在某些类型的项目中可能也需要它（如在 JavaScript 中使用 DOM）。

3）**SAX 表现较好**，这要依赖于它特定的解析方式－事件驱动。一个 SAX 检测即 将到来的 XML 流，但并没有载入到内存（当然当 XML 流被读入时，会有部分文档 暂时隐藏在内存中）。

**XML解析：   
先比较几种XML文件的解析方式：**   
1. DOM（Document Object Model)

DOM是用与平台和语言无关的方式表示XML文档的官方W3C标准。DOM是以层次结构组织的节点或信息片断的集合。这个层次结构允许开发人员在树中寻找特定信息。分析该结构通常需要加载整个文档和构造层次结构，然后才能做任何工作。由于它是基于信息层次的，因而DOM被认为是基于树或基于对象的。

【优点】

①允许应用程序对数据和结构做出更改。

②访问是双向的，可以在任何时候在树中上下导航，获取和操作任意部分的数据。

【缺点】

①通常需要加载整个XML文档来构造层次结构，消耗资源大。

SAX（Simple API for XML)

SAX处理的优点非常类似于流媒体的优点。分析能够立即开始，而不是等待所有的数据被处理。而且，由于应用程序只是在读取数据时检查数据，因此不需要将数据存储在内存中。这对于大型文档来说是个巨大的优点。事实上，应用程序甚至不必解析整个文档；它可以在某个条件得到满足时停止解析。一般来说，SAX还比它的替代者DOM快许多。

选择DOM还是选择SAX？ 对于需要自己编写代码来处理XML文档的开发人员来说， 选择DOM还是SAX解析模型是一个非常重要的设计决策。 DOM采用建立树形结构的方式访问XML文档，而SAX采用的是事件模型。

DOM解析器把XML文档转化为一个包含其内容的树，并可以对树进行遍历。用DOM解析模型的优点是编程容易，开发人员只需要调用建树的指令，然后利用navigation APIs访问所需的树节点来完成任务。可以很容易的添加和修改树中的元素。然而由于使用DOM解析器的时候需要处理整个XML文档，所以对性能和内存的要求比较高，尤其是遇到很大的XML文件的时候。**由于它的遍历能力，DOM解析器常用于XML文档需要频繁的改变的服务中。**

SAX解析器采用了基于事件的模型，它在解析XML文档的时候可以触发一系列的事件，当发现给定的tag的时候，它可以激活一个回调方法，告诉该方法制定的标签已经找到。SAX对内存的要求通常会比较低，因为它让开发人员自己来决定所要处理的tag.特别是当开发人员只需要处理文档中所包含的部分数据时，SAX这种扩展能力得到了更好的体现。但用SAX解析器的时候编码工作会比较困难，而且很难同时访问同一个文档中的多处不同数据。

【优势】

①不需要等待所有数据都被处理，分析就能立即开始。

②只在读取数据时检查数据，不需要保存在内存中。

③可以在某个条件得到满足时停止解析，不必解析整个文档。

④效率和性能较高，能解析大于系统内存的文档。

【缺点】

①需要应用程序自己负责TAG的处理逻辑（例如维护父/子关系等），文档越复杂程序就越复杂。

②单向导航，无法定位文档层次，很难同时访问同一文档的不同部分数据，不支持XPath。

JDOM(**[Java](http://lib.csdn.net/base/javase" \o "Java SE知识库" \t "https://juejin.im/entry/_blank)**-based Document Object Model)

JDOM的目的是成为Java特定文档模型，它简化与XML的交互并且比使用DOM实现更快。由于是第一个Java特定模型，JDOM一直得到大力推广和促进。正在考虑通过“Java规范请求JSR-102”将它最终用作“Java标准扩展”。从2000年初就已经开始了JDOM开发。

JDOM与DOM主要有两方面不同。首先，JDOM仅使用具体类而不使用接口。这在某些方面简化了API，但是也限制了灵活性。第二，API大量使用了Collections类，简化了那些已经熟悉这些类的Java开发者的使用。

JDOM文档声明其目的是“使用20%（或更少）的精力解决80%（或更多）Java/XML问题”（根据学习曲线假定为20%）。JDOM对于大多数Java/XML应用程序来说当然是有用的，并且大多数开发者发现API比DOM容易理解得多。JDOM还包括对程序行为的相当广泛检查以防止用户做任何在XML中无意义的事。然而，它仍需要您充分理解XML以便做一些超出基本的工作（或者甚至理解某些情况下的错误）。这也许是比学习DOM或JDOM接口都更有意义的工作。

JDOM自身不包含解析器。它通常使用SAX2解析器来解析和验证输入XML文档（尽管它还可以将以前构造的DOM表示作为输入）。它包含一些转换器以将JDOM表示输出成SAX2事件流、DOM模型或XML文本文档。JDOM是在Apache许可证变体下发布的开放源码。

【优点】

①使用具体类而不是接口，简化了DOM的API。

②大量使用了Java集合类，方便了Java开发人员。

【缺点】

①没有较好的灵活性。

②性能较差。

DOM4J(Document Object Model for Java)

虽然DOM4J代表了完全独立的开发结果，但最初，它是JDOM的一种智能分支。它合并了许多超出基本XML文档表示的功能，包括集成的XPath支持、XML Schema支持以及用于大文档或流化文档的基于事件的处理。它还提供了构建文档表示的选项，它通过DOM4J API和标准DOM接口具有并行访问功能。从2000下半年开始，它就一直处于开发之中。

为支持所有这些功能，DOM4J使用接口和抽象基本类方法。DOM4J大量使用了API中的Collections类，但是在许多情况下，它还提供一些替代方法以允许更好的性能或更直接的编码方法。直接好处是，虽然DOM4J付出了更复杂的API的代价，但是它提供了比JDOM大得多的灵活性。

在添加灵活性、XPath集成和对大文档处理的目标时，DOM4J的目标与JDOM是一样的：针对Java开发者的易用性和直观操作。它还致力于成为比JDOM更完整的解决方案，实现在本质上处理所有Java/XML问题的目标。在完成该目标时，它比JDOM更少强调防止不正确的应用程序行为。

DOM4J是一个非常非常优秀的**[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "https://juejin.im/entry/_blank)** XML API，具有性能优异、功能强大和极端易用使用的特点，同时它也是一个开放源代码的软件。如今你可以看到越来越多的Java软件都在使用DOM4J来读写XML，特别值得一提的是连Sun的JAXM也在用DOM4J.

【优点】

①大量使用了Java集合类，方便Java开发人员，同时提供一些提高性能的替代方法。

②支持XPath。

③有很好的性能。

【缺点】

①大量使用了接口，API较为复杂。

二、比较

1. DOM4J性能最好，连Sun的JAXM也在用DOM4J。目前许多开源项目中大量采用DOM4J，例如大名鼎鼎的Hibernate也用DOM4J来读取XML配置文件。如果不考虑可移植性，那就采用DOM4J.

2. JDOM和DOM在性能测试时表现不佳，在测试10M文档时内存溢出，但可移植。在小文档情况下还值得考虑使用DOM和JDOM.虽然JDOM的开发者已经说明他们期望在正式发行版前专注性能问题，但是从性能观点来看，它确实没有值得推荐之处。另外，DOM仍是一个非常好的选择。DOM实现广泛应用于多种编程语言。它还是许多其它与XML相关的标准的基础，因为它正式获得W3C推荐（与基于非标准的Java模型相对），所以在某些类型的项目中可能也需要它（如在JavaScript中使用DOM）。

3. SAX表现较好，这要依赖于它特定的解析方式－事件驱动。一个SAX检测即将到来的XML流，但并没有载入到内存（当然当XML流被读入时，会有部分文档暂时隐藏在内存中）。

我的看法：如果XML文档较大且不考虑移植性问题建议采用DOM4J；如果XML文档较小则建议采用JDOM；如果需要及时处理而不需要保存数据则考虑SAX。但无论如何，还是那句话：适合自己的才是最好的，如果时间允许，建议大家讲这四种方法都尝试一遍然后选择一种适合自己的即可。

**74、你在项目中哪些地方用到了XML？**

答：XML的主要作用有两个方面：**数据交换和信息配置。**在做数据交换时，XML将数据用标签组装成起来，然后压缩打包加密后通过网络传送给接收者，接收解密与解压缩后再从XML文件中还原相关信息进行处理，XML曾经是异构系统间交换数据的事实标准，但此项功能几乎已经被JSON（JavaScript Object Notation）取而代之。当然，目前很多软件仍然使用XML来存储配置信息，我们在很多项目中通常也会将作为配置信息的硬代码写在XML文件中，Java的很多框架也是这么做的，而且这些框架都选择了dom4j作为处理XML的工具，因为Sun公司的官方API实在不怎么好用。

补充：现在有很多时髦的软件（如Sublime）已经开始将配置文件书写成JSON格式，我们已经强烈的感受到XML的另一项功能也将逐渐被业界抛弃。

**75、阐述JDBC操作数据库的步骤**。

答：下面的代码以连接本机的Oracle数据库为例，演示JDBC操作数据库的步骤。

加载驱动。

Class.forName("com.mysql.jdbc.Driver");

创建连接。

Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/testdb?User=root&password=123");

创建语句。

PreparedStatement ps = con.prepareStatement("select \* from emp where sal between ? and ?");

ps.setInt(1, 1000);

ps.setInt(2, 3000);

执行语句。

ResultSet rs = ps.executeQuery();

处理结果。

while(rs.next()) {

    System.out.println(rs.getInt("empno") + " - " + rs.getString("ename"));

}

关闭资源。

finally {

    if(con != null) {

        try {

            con.close();

        } catch (SQLException e) {

            e.printStackTrace();

        }

    }

}

提示：关闭外部资源的顺序应该和打开的顺序相反，也就是说先关闭ResultSet、再关闭Statement、在关闭Connection。上面的代码只关闭了Connection（连接），虽然通常情况下在关闭连接时，连接上创建的语句和打开的游标也会关闭，但不能保证总是如此，因此应该按照刚才说的顺序分别关闭。此外，第一步加载驱动在JDBC 4.0中是可以省略的（自动从类路径中加载驱动），但是我们建议保留。

**76、Statement和PreparedStatement有什么区别？哪个性能更好？**

答：与Statement相比，①PreparedStatement接口代表**预编译的语句**，它主要的优势在于可以**减少SQL的编译错误并增加SQL的安全性**（减少SQL注入攻击的可能性）；②PreparedStatement中的SQL语句是可以带参数的，避免了**用字符串连接拼接SQL语句的麻烦和不安全**；③当**批量处理SQL或频繁执行相同的查询时，PreparedStatement有明显的性能上的优势**，**由于数据库可以将编译优化后的SQL语句缓存起来**，下次执行相同结构的语句时就会很快（不用再次编译和生成执行计划）。

补充：为了提供对存储过程的调用，JDBC API中还提供了CallableStatement接口。存储过程（Stored Procedure）是数据库中一组为了完成特定功能的SQL语句的集合，经编译后存储在数据库中，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。虽然调用存储过程会在网络开销、安全性、性能上获得很多好处，但是存在如果底层数据库发生迁移时就会有很多麻烦，因为每种数据库的存储过程在书写上存在不少的差别。

**77、使用JDBC操作数据库时，如何提升读取数据的性能？如何提升更新数据的性能？**

答：要提升读取数据的性能，**可以指定通过结果集（ResultSet）对象的setFetchSize()方法指定每次抓取的记录数（典型的空间换时间策略）**；要提**升更新数据的性能可以使用PreparedStatement语句构建批处理，将若干SQL语句置于一个批处理中执行**。

**78、在进行数据库编程时，连接池有什么作用？**

答：由于创建连接和释放连接都有很大的开销（尤其是数据库服务器不在本地时，每次建立连接都需要进行TCP的三次握手，释放连接需要进行TCP四次挥手，造成的开销是不可忽视的），为了提升系统访问数据库的性能，可以事先创建若干连接置于连接池中，需要时直接从连接池获取，使用结束时归还连接池而不必关闭连接，从而避免频繁创建和释放连接所造成的开销，这是典型的用空间换取时间的策略（浪费了空间存储连接，但节省了创建和释放连接的时间）。池化技术在Java开发中是很常见的，在使用线程时创建线程池的道理与此相同。基于Java的开源数据库连接池主要有：C3P0、Proxool、DBCP、BoneCP、Druid等。

补充：在计算机系统中时间和空间是不可调和的矛盾，理解这一点对设计满足性能要求的算法是至关重要的。大型网站性能优化的一个关键就是使用缓存，而缓存跟上面讲的连接池道理非常类似，也是使用空间换时间的策略。可以将热点数据置于缓存中，当用户查询这些数据时可以直接从缓存中得到，这无论如何也快过去数据库中查询。当然，缓存的置换策略等也会对系统性能产生重要影响，对于这个问题的讨论已经超出了这里要阐述的范围。

**79、什么是DAO模式？**

答：**DAO（Data Access Object）顾名思义是一个为数据库或其他持久化机制提供了抽象接口的对象，在不暴露底层持久化方案实现细节的前提下提供了各种数据访问操作。**在实际的开发中，应该将所有对数据源的访问操作进行抽象化后封装在一个公共API中。用程序设计语言来说，就是建立一个接口，接口中定义了此应用程序中将会用到的所有事务方法。在这个应用程序中，当需要和数据源进行交互的时候则使用这个接口，并且编写一个单独的类来实现这个接口，在逻辑上该类对应一个特定的数据存储。DAO模式实际上包含了两个模式，**一是Data Accessor（数据访问器），二是Data Object（数据对象）**，**前者要解决如何访问数据的问题，而后者要解决的是如何用对象封装数据。**

**80、事务的ACID是指什么？**

答：

- 原子性(Atomic)：事务中各项操作，要么全做要么全不做，任何一项操作的失败都会导致整个事务的失败；

- 一致性(Consistent)：事务结束后系统状态是一致的；

- 隔离性(Isolated)：并发执行的事务彼此无法看到对方的中间状态；

- 持久性(Durable)：事务完成后所做的改动都会被持久化，即使发生灾难性的失败。通过日志和同步备份可以在故障发生后重建数据。

补充：关于事务，在面试中被问到的概率是很高的，可以问的问题也是很多的。首先需要知道的是，只有存在并发数据访问时才需要事务。当多个事务访问同一数据时，可能会存在5类问题，包括3类数据读取问题（脏读、不可重复读和幻读）和2类数据更新问题（第1类丢失更新和第2类丢失 更新）。

脏读（Dirty Read）：A事务读取B事务尚未提交的数据并在此基础上操作，而B事务执行回滚，那么A读取到的数据就是脏数据。

| **时间** | **转账事务A** | **取款事务B** |
| --- | --- | --- |
| T1 |  | 开始事务 |
| T2 | 开始事务 |  |
| T3 |  | 查询账户余额为1000元 |
| T4 |  | 取出500元余额修改为500元 |
| T5 | 查询账户余额为500元（脏读） |  |
| T6 |  | 撤销事务余额恢复为1000元 |
| T7 | 汇入100元把余额修改为600元 |  |
| T8 | 提交事务 |  |

不可重复读（Unrepeatable Read）：事务A重新读取前面读取过的数据，发现该数据已经被另一个已提交的事务B修改过了。

| **时间** | **转账事务A** | **取款事务B** |
| --- | --- | --- |
| T1 |  | 开始事务 |
| T2 | 开始事务 |  |
| T3 |  | 查询账户余额为1000元 |
| T4 | 查询账户余额为1000元 |  |
| T5 |  | 取出100元修改余额为900元 |
| T6 |  | 提交事务 |
| T7 | 查询账户余额为900元（不可重复读） |  |

幻读（Phantom Read）：事务A重新执行一个查询，返回一系列符合查询条件的行，发现其中插入了被事务B提交的行。

| **时间** | **统计金额事务A** | **转账事务B** |
| --- | --- | --- |
| T1 |  | 开始事务 |
| T2 | 开始事务 |  |
| T3 | 统计总存款为10000元 |  |
| T4 |  | 新增一个存款账户存入100元 |
| T5 |  | 提交事务 |
| T6 | 再次统计总存款为10100元（幻读） |  |

第1类丢失更新：事务A撤销时，把已经提交的事务B的更新数据覆盖了。

| **时间** | **取款事务A** | **转账事务B** |
| --- | --- | --- |
| T1 | 开始事务 |  |
| T2 |  | 开始事务 |
| T3 | 查询账户余额为1000元 |  |
| T4 |  | 查询账户余额为1000元 |
| T5 |  | 汇入100元修改余额为1100元 |
| T6 |  | 提交事务 |
| T7 | 取出100元将余额修改为900元 |  |
| T8 | 撤销事务 |  |
| T9 | 余额恢复为1000元（丢失更新） |  |

第2类丢失更新：事务A覆盖事务B已经提交的数据，造成事务B所做的操作丢失。

| **时间** | **转账事务A** | **取款事务B** |
| --- | --- | --- |
| T1 |  | 开始事务 |
| T2 | 开始事务 |  |
| T3 |  | 查询账户余额为1000元 |
| T4 | 查询账户余额为1000元 |  |
| T5 |  | 取出100元将余额修改为900元 |
| T6 |  | 提交事务 |
| T7 | 汇入100元将余额修改为1100元 |  |
| T8 | 提交事务 |  |
| T9 | 查询账户余额为1100元（丢失更新） |  |

数据并发访问所产生的问题，在有些场景下可能是允许的，但是有些场景下可能就是致命的，数据库通常会通过锁机制来解决数据并发访问问题，按锁定对象不同可以分为表级锁和行级锁；按并发事务锁定关系可以分为共享锁和独占锁，具体的内容大家可以自行查阅资料进行了解。  
直接使用锁是非常麻烦的，为此数据库为用户提供了自动锁机制，只要用户指定会话的事务隔离级别，数据库就会通过分析SQL语句然后为事务访问的资源加上合适的锁，此外，数据库还会维护这些锁通过各种手段提高系统的性能，这些对用户来说都是透明的（就是说你不用理解，事实上我确实也不知道）。ANSI/ISO SQL 92标准定义了4个等级的事务隔离级别，如下表所示：

| **隔离级别** | **脏读** | **不可重复读** | **幻读** | **第一类丢失更新** | **第二类丢失更新** |
| --- | --- | --- | --- | --- | --- |
| READ UNCOMMITED | 允许 | 允许 | 允许 | 不允许 | 允许 |
| READ COMMITTED | 不允许 | 允许 | 允许 | 不允许 | 允许 |
| REPEATABLE READ | 不允许 | 不允许 | 允许 | 不允许 | 不允许 |
| SERIALIZABLE | 不允许 | 不允许 | 不允许 | 不允许 | 不允许 |

需要说明的是，事务隔离级别和数据访问的并发性是对立的，事务隔离级别越高并发性就越差。所以要根据具体的应用来确定合适的事务隔离级别，这个地方没有万能的原则。

**81、JDBC中如何进行事务处理？**

答：Connection提供了事务处理的方法，通过调用setAutoCommit(false)可以设置手动提交事务；当事务完成后用commit()显式提交事务；如果在事务处理过程中发生异常则通过rollback()进行事务回滚。除此之外，从JDBC 3.0中还引入了Savepoint（保存点）的概念，允许通过代码设置保存点并让事务回滚到指定的保存点。



**82、JDBC能否处理Blob和Clob？**

答： Blob是指二进制大对象（Binary Large Object），而Clob是指字符大对象（Character Large Objec），因此其中Blob是为存储大的二进制数据而设计的，而Clob是为存储大的文本数据而设计的。JDBC的PreparedStatement和ResultSet都提供了相应的方法来支持Blob和Clob操作。下面的代码展示了如何使用JDBC操作LOB：

下面以MySQL数据库为例，创建一个张有三个字段的用户表，包括编号（id）、姓名（name）和照片（photo），建表语句如下：

create table tb\_user

(

id int primary key auto\_increment,

name varchar(20) unique not null,

photo longblob

);

下面的Java代码向数据库中插入一条记录：

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStream;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.SQLException;

class JdbcLobTest {

    public static void main(String[] args) {

        Connection con = null;

        try {

            // 1. 加载驱动（Java6以上版本可以省略）

            Class.forName("com.mysql.jdbc.Driver");

            // 2. 建立连接

            con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "123456");

            // 3. 创建语句对象

            PreparedStatement ps = con.prepareStatement("insert into tb\_user values (default, ?, ?)");

            ps.setString(1, "骆昊");              // 将SQL语句中第一个占位符换成字符串

            try (InputStream in = new FileInputStream("test.jpg")) {    // Java 7的TWR

                ps.setBinaryStream(2, in);      // 将SQL语句中第二个占位符换成二进制流

                // 4. 发出SQL语句获得受影响行数

                System.out.println(ps.executeUpdate() == 1 ? "插入成功" : "插入失败");

            } catch(IOException e) {

                System.out.println("读取照片失败!");

            }

        } catch (ClassNotFoundException | SQLException e) {     // Java 7的多异常捕获

            e.printStackTrace();

        } finally { // 释放外部资源的代码都应当放在finally中保证其能够得到执行

            try {

                if(con != null && !con.isClosed()) {

                    con.close();    // 5. 释放数据库连接

                    con = null;     // 指示垃圾回收器可以回收该对象

                }

            } catch (SQLException e) {

                e.printStackTrace();

            }

        }

    }

}

**83、简述正则表达式及其用途。**

答：在编写处理字符串的程序时，经常会有查找符合某些复杂规则的字符串的需要。**正则表达式就是用于描述这些规则的工具。换句话说，正则表达式就是记录文本规则的代码。**

说明：计算机诞生初期处理的信息几乎都是数值，但是时过境迁，今天我们使用计算机处理的信息更多的时候不是数值而是字符串，**正则表达式就是在进行字符串匹配和处理的时候最为强大的工具**，绝大多数语言都提供了对正则表达式的支持。

**84、Java中是如何支持正则表达式操作的？**

答：Java中的String类提供了支持正则表达式操作的方法，包括：matches()、replaceAll()、replaceFirst()、split()。此外，**Java中可以用Pattern类表示正则表达式对象，它提供了丰富的API进行各种正则表达式操作**，请参考下面面试题的代码。

面试题： - 如果要从字符串中截取第一个英文左括号之前的字符串，例如：北京市(朝阳区)(西城区)(海淀区)，截取结果为：北京市，那么正则表达式怎么写？

import java.util.regex.Matcher;

import java.util.regex.Pattern;

class RegExpTest {

    public static void main(String[] args) {

        String str = "北京市(朝阳区)(西城区)(海淀区)";

        Pattern p = Pattern.compile(".\*?(?=\\()");

        Matcher m = p.matcher(str);

        if(m.find()) {

            System.out.println(m.group());

        }

    }

}

**85、获得一个类的类对象有哪些方式？**

答：

- 方法1：类型.class，例如：String.class

- 方法2：对象.getClass()，例如：”hello”.getClass()

- 方法3：Class.forName()，例如：Class.forName(“java.lang.String”)

**86、如何通过反射创建对象？**

答：

- 方法1：通过类对象调用newInstance()方法，例如：String.class.newInstance()

- 方法2：通过类对象的getConstructor()或getDeclaredConstructor()方法获得构造器（Constructor）对象并调用其newInstance()方法创建对象，例如：String.class.getConstructor(String.class).newInstance(“Hello”);

**87、如何通过反射获取和设置对象私有字段的值？**

答：可以通过类对象的getDeclaredField()方法字段（Field）对象，然后再通过字段对象的setAccessible(true)将其设置为可以访问，接下来就可以通过get/set方法来获取/设置字段的值了。下面的代码实现了一个反射的工具类，其中的两个静态方法分别用于获取和设置私有字段的值，字段可以是基本类型也可以是对象类型且支持多级对象操作，例如ReflectionUtil.get(dog, “owner.car.engine.id”);可以获得dog对象的主人的汽车的引擎的ID号。

import java.lang.reflect.Constructor;

import java.lang.reflect.Field;

import java.lang.reflect.Modifier;

import java.util.ArrayList;

import java.util.List;

/\*\*

 \* 反射工具类

 \* @author 骆昊

 \*

 \*/

public class ReflectionUtil {

    private ReflectionUtil() {

        throw new AssertionError();

    }

    /\*\*

     \* 通过反射取对象指定字段(属性)的值

     \* @param target 目标对象

     \* @param fieldName 字段的名字

     \* @throws 如果取不到对象指定字段的值则抛出异常

     \* @return 字段的值

     \*/

    public static Object getValue(Object target, String fieldName) {

        Class<?> clazz = target.getClass();

        String[] fs = fieldName.split("\\.");

        try {

            for(int i = 0; i < fs.length - 1; i++) {

                Field f = clazz.getDeclaredField(fs[i]);

                f.setAccessible(true);

                target = f.get(target);

                clazz = target.getClass();

            }

            Field f = clazz.getDeclaredField(fs[fs.length - 1]);

            f.setAccessible(true);

            return f.get(target);

        }

        catch (Exception e) {

            throw new RuntimeException(e);

        }

    }

    /\*\*

     \* 通过反射给对象的指定字段赋值

     \* @param target 目标对象

     \* @param fieldName 字段的名称

     \* @param value 值

     \*/

    public static void setValue(Object target, String fieldName, Object value) {

        Class<?> clazz = target.getClass();

        String[] fs = fieldName.split("\\.");

        try {

            for(int i = 0; i < fs.length - 1; i++) {

                Field f = clazz.getDeclaredField(fs[i]);

                f.setAccessible(true);

                Object val = f.get(target);

                if(val == null) {

                    Constructor<?> c = f.getType().getDeclaredConstructor();

                    c.setAccessible(true);

                    val = c.newInstance();

                    f.set(target, val);

                }

                target = val;

                clazz = target.getClass();

            }

            Field f = clazz.getDeclaredField(fs[fs.length - 1]);

            f.setAccessible(true);

            f.set(target, value);

        }

        catch (Exception e) {

            throw new RuntimeException(e);

        }

    }

}

**88、如何通过反射调用对象的方法？**

答：请看下面的代码：

import java.lang.reflect.Method;

class MethodInvokeTest {

    public static void main(String[] args) throws Exception {

        String str = "hello";

        Method m = str.getClass().getMethod("toUpperCase");

        System.out.println(m.invoke(str));  // HELLO

    }

}

1. **简述一下面向对象的”六原则一法则”。**

**单一职责原则，开闭原则，依赖倒转原则，里氏替换原则，接口隔离原则，合成聚合复用原则，迪米特法则。**

答：

- **单一职责原则**：一个类只做它该做的事情。（单一职责原则想表达的就是”高内聚”，写代码最终极的原则只有六个字”高内聚、低耦合”，就如同葵花宝典或辟邪剑谱的中心思想就八个字”欲练此功必先自宫”，所谓的高内聚就是一个代码模块只完成一项功能，在面向对象中，如果只让一个类完成它该做的事，而不涉及与它无关的领域就是践行了高内聚的原则，这个类就只有单一职责。我们都知道一句话叫”因为专注，所以专业”，一个对象如果承担太多的职责，那么注定它什么都做不好。这个世界上任何好的东西都有两个特征，一个是功能单一，好的相机绝对不是电视购物里面卖的那种一个机器有一百多种功能的，它基本上只能照相；另一个是模块化，好的自行车是组装车，从减震叉、刹车到变速器，所有的部件都是可以拆卸和重新组装的，好的乒乓球拍也不是成品拍，一定是底板和胶皮可以拆分和自行组装的，一个好的软件系统，它里面的每个功能模块也应该是可以轻易的拿到其他系统中使用的，这样才能实现软件复用的目标。）

- **开闭原则**：软件实体应当对扩展开放，对修改关闭。（在理想的状态下，当我们需要为一个软件系统增加新功能时，只需要从原来的系统派生出一些新类就可以，不需要修改原来的任何一行代码。要做到开闭有两个要点：①抽象是关键，一个系统中如果没有抽象类或接口系统就没有扩展点；②封装可变性，将系统中的各种可变因素封装到一个继承结构中，如果多个可变因素混杂在一起，系统将变得复杂而换乱，如果不清楚如何封装可变性，可以参考《设计模式精解》一书中对桥梁模式的讲解的章节。）

- **依赖倒转原则**：面向接口编程。（该原则说得直白和具体一些就是声明方法的参数类型、方法的返回类型、变量的引用类型时，尽可能使用抽象类型而不用具体类型，因为抽象类型可以被它的任何一个子类型所替代，请参考下面的里氏替换原则。）

**里氏替换原则**：任何时候都可以用子类型替换掉父类型。（关于里氏替换原则的描述，Barbara Liskov女士的描述比这个要复杂得多，但简单的说就是能用父类型的地方就一定能使用子类型。里氏替换原则可以检查继承关系是否合理，如果一个继承关系违背了里氏替换原则，那么这个继承关系一定是错误的，需要对代码进行重构。例如让猫继承狗，或者狗继承猫，又或者让正方形继承长方形都是错误的继承关系，因为你很容易找到违反里氏替换原则的场景。需要注意的是：子类一定是增加父类的能力而不是减少父类的能力，因为子类比父类的能力更多，把能力多的对象当成能力少的对象来用当然没有任何问题。）

- **接口隔离原则**：接口要小而专，绝不能大而全。（臃肿的接口是对接口的污染，既然接口表示能力，那么一个接口只应该描述一种能力，接口也应该是高度内聚的。例如，琴棋书画就应该分别设计为四个接口，而不应设计成一个接口中的四个方法，因为如果设计成一个接口中的四个方法，那么这个接口很难用，毕竟琴棋书画四样都精通的人还是少数，而如果设计成四个接口，会几项就实现几个接口，这样的话每个接口被复用的可能性是很高的。Java中的接口代表能力、代表约定、代表角色，能否正确的使用接口一定是编程水平高低的重要标识。）

- **合成聚合复用原则**：优先使用聚合或合成关系复用代码。（通过继承来复用代码是面向对象程序设计中被滥用得最多的东西，因为所有的教科书都无一例外的对继承进行了鼓吹从而误导了初学者，**类与类之间简单的说有三种关系，Is-A关系、Has-A关系、Use-A关系，分别代表继承、关联和依赖。其中，关联关系根据其关联的强度又可以进一步划分为关联、聚合和合成，但说白了都是Has-A关系，合成聚合复用原则想表达的是优先考虑Has-A关系而不是Is-A关系复用代码**，原因嘛可以自己从百度上找到一万个理由，需要说明的是，即使在Java的API中也有不少滥用继承的例子，例如Properties类继承了Hashtable类，Stack类继承了Vector类，这些继承明显就是错误的，更好的做法是在Properties类中放置一个Hashtable类型的成员并且将其键和值都设置为字符串来存储数据，而Stack类的设计也应该是在Stack类中放一个Vector对象来存储数据。记住：任何时候都不要继承工具类，工具是可以拥有并可以使用的，而不是拿来继承的。）

- **迪米特法则**：迪米特法则又叫最少知识原则，一个对象应当对其他对象有尽可能少的了解。（迪米特法则简单的说就是如何做到”低耦合”，**门面模式和调停者模式就是对迪米特法则的践行。**对于门面模式可以举一个简单的例子，你去一家公司洽谈业务，你不需要了解这个公司内部是如何运作的，你甚至可以对这个公司一无所知，去的时候只需要找到公司入口处的前台美女，告诉她们你要做什么，她们会找到合适的人跟你接洽，前台的美女就是公司这个系统的门面。再复杂的系统都可以为用户提供一个简单的门面，Java Web开发中作为前端控制器的Servlet或Filter不就是一个门面吗，浏览器对服务器的运作方式一无所知，但是通过前端控制器就能够根据你的请求得到相应的服务。调停者模式也可以举一个简单的例子来说明，例如一台计算机，CPU、内存、硬盘、显卡、声卡各种设备需要相互配合才能很好的工作，但是如果这些东西都直接连接到一起，计算机的布线将异常复杂，在这种情况下，主板作为一个调停者的身份出现，它将各个设备连接在一起而不需要每个设备之间直接交换数据，这样就减小了系统的耦合度和复杂度，如下图所示。迪米特法则用通俗的话来将就是不要和陌生人打交道，如果真的需要，找一个自己的朋友，让他替你和陌生人打交道。）





90、简述一下你了解的设计模式。

答：所谓设计模式，就是一套被反复使用的代码设计经验的总结（情境中一个问题经过证实的一个解决方案）。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。设计模式使人们可以更加简单方便的复用成功的设计和体系结构。将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。

在GoF的《Design Patterns: Elements of Reusable Object-Oriented Software》中给出了三类（创建型[对类的实例化过程的抽象化]、结构型[描述如何将类或对象结合在一起形成更大的结构]、行为型[对在不同的对象之间划分责任和算法的抽象化]）共23种设计模式，包括：Abstract Factory（抽象工厂模式），Builder（建造者模式），Factory Method（工厂方法模式），Prototype（原始模型模式），Singleton（单例模式）；Facade（门面模式），Adapter（适配器模式），Bridge（桥梁模式），Composite（合成模式），Decorator（装饰模式），Flyweight（享元模式），Proxy（代理模式）；Command（命令模式），Interpreter（解释器模式），Visitor（访问者模式），Iterator（迭代子模式），Mediator（调停者模式），Memento（备忘录模式），Observer（观察者模式），State（状态模式），Strategy（策略模式），Template Method（模板方法模式）， Chain Of Responsibility（责任链模式）。

面试被问到关于设计模式的知识时，可以拣最常用的作答，例如：

- 工厂模式：工厂类可以根据条件生成不同的子类实例，这些子类有一个公共的抽象父类并且实现了相同的方法，但是这些方法针对不同的数据进行了不同的操作（多态方法）。当得到子类的实例后，开发人员可以调用基类中的方法而不必考虑到底返回的是哪一个子类的实例。

- 代理模式：给一个对象提供一个代理对象，并由代理对象控制原对象的引用。实际开发中，按照使用目的的不同，代理可以分为：远程代理、虚拟代理、保护代理、Cache代理、防火墙代理、同步化代理、智能引用代理。

- 适配器模式：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起使用的类能够一起工作。

- 模板方法模式：提供一个抽象类，将部分逻辑以具体方法或构造器的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法（多态实现），从而实现不同的业务逻辑。

除此之外，还可以讲讲上面提到的门面模式、桥梁模式、单例模式、装潢模式（Collections工具类和I/O系统中都使用装潢模式）等，反正基本原则就是拣自己最熟悉的、用得最多的作答，以免言多必失。

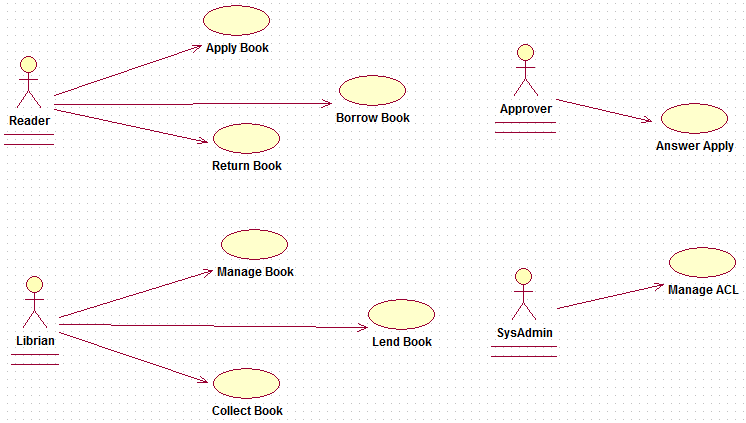
**92、什么是UML？**

答：UML是统一建模语言（Unified Modeling Language）的缩写，它发表于1997年，综合了当时已经存在的面向对象的建模语言、方法和过程，**是一个支持模型化和软件系统开发的图形化语言，为软件开发的所有阶段提供模型化和可视化支持。使用UML可以帮助沟通与交流，辅助应用设计和文档的生成，还能够阐释系统的结构和行为。**

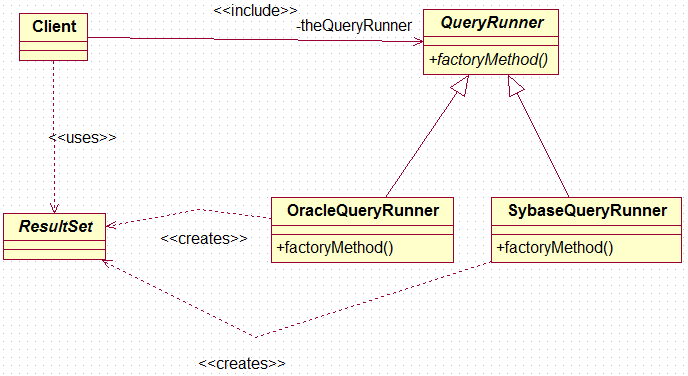
**93、UML中有哪些常用的图？**

答：UML定义了多种图形化的符号来描述软件系统部分或全部的静态结构和动态结构，包括：用**例图（use case diagram）、类图（class diagram）、时序图（sequence diagram）**、协作图（collaboration diagram）、状态图（statechart diagram）、活动图（activity diagram）、构件图（component diagram）、部署图（deployment diagram）等。在这些图形化符号中，有三种图最为重要，分别是：**用例图（用来捕获需求，描述系统的功能，通过该图可以迅速的了解系统的功能模块及其关系）、类图（描述类以及类与类之间的关系，通过该图可以快速了解系统）、时序图（描述执行特定任务时对象之间的交互关系以及执行顺序，通过该图可以了解对象能接收的消息也就是说对象能够向外界提供的服务）**。

用例图：

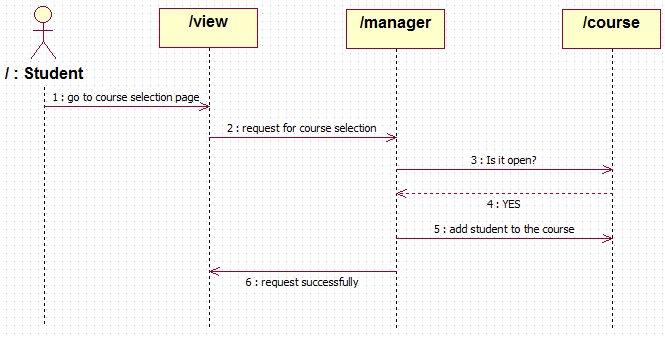


Djfa



上（类图）

下（时序图）



**94、用Java写一个冒泡排序。**

答：冒泡排序几乎是个程序员都写得出来，但是面试的时候如何写一个逼格高的冒泡排序却不是每个人都能做到，下面提供一个参考代码：

|  |
| --- |
| import java.util.Comparator;    /\*\*   \* 排序器接口(策略模式: 将算法封装到具有共同接口的独立的类中使得它们可以相互替换)   \* @author骆昊   \*   \*/  public interface Sorter {       /\*\*      \* 排序      \* @param list 待排序的数组      \*/     public <T extends Comparable<T>> void sort(T[] list);       /\*\*      \* 排序      \* @param list 待排序的数组      \* @param comp 比较两个对象的比较器      \*/     public <T> void sort(T[] list, Comparator<T> comp);  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | import java.util.Comparator;    /\*\*   \* 冒泡排序   \*   \* @author骆昊   \*   \*/  public class BubbleSorter implements Sorter {        @Override      public <T extends Comparable<T>> void sort(T[] list) {          boolean swapped = true;          for (int i = 1, len = list.length; i < len && swapped; ++i) {              swapped = false;              for (int j = 0; j < len - i; ++j) {                  if (list[j].compareTo(list[j + 1]) > 0) {                      T temp = list[j];                      list[j] = list[j + 1];                      list[j + 1] = temp;                      swapped = true;                  }              }          }      }        @Override      public <T> void sort(T[] list, Comparator<T> comp) {          boolean swapped = true;          for (int i = 1, len = list.length; i < len && swapped; ++i) {              swapped = false;              for (int j = 0; j < len - i; ++j) {                  if (comp.compare(list[j], list[j + 1]) > 0) {                      T temp = list[j];                      list[j] = list[j + 1];                      list[j + 1] = temp;                      swapped = true;                  }              }          }      }  } |

**95、用Java写一个折半查找。**

答：折半查找，也称二分查找、二分搜索，是一种在有序数组中查找某一特定元素的搜索算法。搜素过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜素过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组已经为空，则表示找不到指定的元素。这种搜索算法每一次比较都使搜索范围缩小一半，其时间复杂度是O(logN)。

import java.util.Comparator;

public class MyUtil {

   public static <T extends Comparable<T>> int binarySearch(T[] x, T key) {

      return binarySearch(x, 0, x.length- 1, key);

   }

   // 使用循环实现的二分查找

   public static <T> int binarySearch(T[] x, T key, Comparator<T> comp) {

      int low = 0;

      int high = x.length - 1;

      while (low <= high) {

          int mid = (low + high) >>> 1;

          int cmp = comp.compare(x[mid], key);

          if (cmp < 0) {

            low= mid + 1;

          }

          else if (cmp > 0) {

            high= mid - 1;

          }

          else {

            return mid;

          }

      }

      return -1;

   }

   // 使用递归实现的二分查找

   private static<T extends Comparable<T>> int binarySearch(T[] x, int low, int high, T key) {

      if(low <= high) {

        int mid = low + ((high -low) >> 1);

        if(key.compareTo(x[mid])== 0) {

           return mid;

        }

        else if(key.compareTo(x[mid])< 0) {

           return binarySearch(x,low, mid - 1, key);

        }

        else {

           return binarySearch(x,mid + 1, high, key);

        }

      }

      return -1;

   }

}

说明：上面的代码中给出了折半查找的两个版本，一个用递归实现，一个用循环实现。需要注意的是计算**中间位置时不应该使用(high+ low) / 2的方式，因为加法运算可能导致整数越界**，这里应该使用以下三种方式之一：low + (high – low) / 2或low + (high – low) >> 1或(low + high) >>> 1（>>>是逻辑右移，是不带符号位的右移）

### **2.Java集合框架树**



Collection接口：它是Java集合框架的一个根接口，也是List、Set和Queue接口的父接口。同时它定义了可用于操作List、Set和Queue的方法—增删改查。

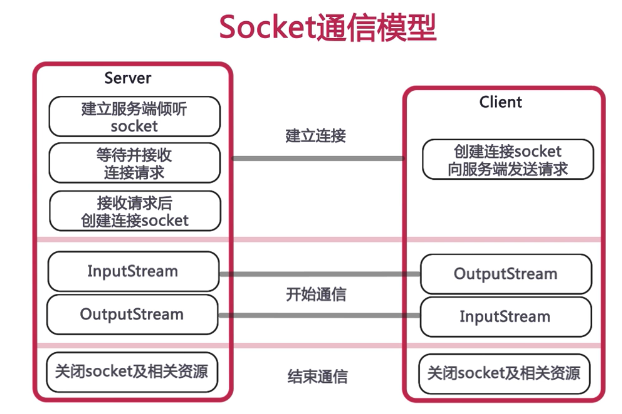
Map接口：它提供了一种映射关系，其中的元素是以键值对（key-value）的形式存储的。Map接口有一个重要的实现类HashMap。  
①在Map接口中的键值对是以Entry类型的对象实例形式存在。  
②在Map接口中键值（Key值）是不可以重复的，value值却可以重复，也就是存在多对一的关系。  
③在Map接口中提供了分别返回Key值的集合、value值的集合以及Entry集合的方法。  
④Map支持泛型，同时Key值和value值都是可以为null的。  
⑤在HashMap中的Entry对象是无序排序的，这点特性和List接口是相反的。  
⑥在HashMap中有且只能有一个Key值为null的映射。（注：key值是不能重复的）

List接口：List接口有一个重要的实现类ArrayList（数组序列）。  
①List是元素有序并且可以重复的集合。  
②List可以精确的控制每个元素的插入位置，或删除某个元素的位置。

Comparator接口：临时的比较规则。如果某一个类要实现这个接口，那必然要实现它的Compare()方法。

Comparable接口：默认的比较规则。当实现了这个接口，则表示这个类的实例可以比较大小，可以进行自然排序。如果某一个类要实现这个接口，那必然要实现它的CompareTo()方法。

**4.Java的Socket通信（多线程）**



思路：

①首先创建服务器端Socket，指定并侦听某一个端口，然后循环监听开始等待客户端的连接….

②创建客户端socket，指定服务器地址和端口，然后获取输出流，向服务器端发送请求，并关闭socket输出流。

③服务端接收到客户端的请求后，创建新线程并启动。

④创建线程处理类，执行线程操作，获取输入流，服务端读取客户端用户详情，关闭资源。

⑤执行线程操作，获取输出流，响应客户端请求，客户端接受到服务端的响应，关闭资源。

简单点讲，就相当于我跟你说话（客户端→服务端），你收到我说的话（服务端→线程处理类），大脑进行思考后（线程处理类），做出回答我的话（线程处理类→客户端）。

**Map接口，HashMap和HashTable的相同点和不同点分别是什么？**

Hashtable和HashMap的区别：

1.Hashtable是Dictionary的子类，HashMap是Map接口的一个实现类；

2.Hashtable中的方法是同步的，而HashMap中的方法在缺省情况下是非同步的。即是说，在多线程应用程序中，不用专门的操作就安全地可以使用Hashtable了；而对于HashMap，则需要额外的同步机制。但HashMap的同步问题可通过Collections的一个静态方法得到解决：

Map Collections.synchronizedMap(Map m)

这个方法返回一个同步的Map，这个Map封装了底层的HashMap的所有方法，使得底层的HashMap即使是在多线程的环境中也是安全的。

3.在HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。当get()方法返回null值时，即可以表示HashMap中没有该键，也可以表示该键所对应的值为null。因此，在HashMap中不能由get()方法来判断HashMap中是否存在某个键，而应该用containsKey()方法来判断。

**46）Java 中的 HashSet，内部是如何工作的？**

HashSet 的内部采用 HashMap来实现。由于 Map 需要 key 和 value，所以所有 key 的都有一个默认 value。类似于 HashMap，HashSet 不允许重复的 key，只允许有一个null key，意思就是 HashSet 中只允许存储一个 null 对象。 存储的不同的key,所有key对应的value都是统一一个默认值

**47）写一段代码在遍历 ArrayList 时移除一个元素？**

该问题的关键在于面试者使用的是 ArrayList 的 remove() 还是 Iterator 的 remove()方法。这有一段示例代码，是使用正确的方式来实现在遍历的过程中移除元素，而不会出现 ConcurrentModificationException 异常的示例代码。

**并发，是在同一个cpu上同时（不是真正的同时，而是看来是同时，因为cpu要在多个程序间切换）运行多个程序。**

**并行，是每个cpu运行一个程序。**

**并发的实质是一个物理CPU(也可以多个物理CPU) 在若干道程序之间多路复用，并发性是对有限物理资源强制行使多用户共享以提高效率。**

**并行性指两个或两个以上事件或活动在同一时刻发生。在多道程序环境下，并行性使多个程序同一时刻可在不同CPU上同时执行。**

**Java 最佳实践的面试问题**

**55）Java 中，编写多线程程序的时候你会遵循哪些最佳实践？**

a）**给线程命名**，这样可以帮助调试。

b）**最小化同步的范围**，而不是将整个方法同步，只对关键部分做同步。

c）如果可以，更**偏向于使用 volatile 而不是 synchronized**。

d）**使用更高层次的并发工具**，而不是使用 wait() 和 notify() 来实现线程间通信，如 BlockingQueue，CountDownLatch 及 Semeaphore。

e）**优先使用并发集合**，而不是对集合进行同步。并发集合提供更好的可扩展性。

56）说出几点 Java 中使用 Collection 的最佳实践？

a）**使用正确的集合类**，例如，如果不需要同步列表，使用 ArrayList 而不是 Vector。

b）**优先使用并发集合，而不是对集合进行同步**。并发集合提供更好的可扩展性。

c）**使用接口代表和访问集合**，如使用List存储 ArrayList，使用 Map 存储 HashMap 等等。

d）**使用迭代器来循环集合**。

e）**使用集合的时候使用泛型。**

57）说出在 Java 中使用线程的最佳实践？

a）**对线程命名**

b）**将线程和任务分离**，使用线程池执行器来执行 Runnable 或 Callable。

c）**使用线程池**

58）说出 IO 的最佳实践？

a）使**用有缓冲区的 IO 类**，而不要单独读取字节或字符。

b）使用 NIO 和 NIO2

c）在 finally 块中关闭流，或者使用 try-with-resource 语句。

d）使用内存映射文件获取更快的 IO。

59）列出应该遵循的 JDBC 最佳实践？

a）**使用批量的操作来插入和更新数据**

b）**使用 PreparedStatement 来避免 SQL 异常，并提高性能。**

c）**使用数据库连接池**

d）**通过列名来获取结果集，不要使用列的下标来获取**。

60）说出几条 Java 中方法重载的最佳实践？

a）**不要重载这样的方法**：一个方法接收 int 参数，而另个方法接收 Integer 参数。

b）**不要重载参数数量一致，而只是参数顺序不同的方法。**

c）**如果重载的方法参数个数多于 5 个，采用可变参数。**

Date、Time 及 Calendar 的面试题

61）在多线程环境下，SimpleDateFormat 是线程安全的吗？

不是，非常不幸，DateFormat 的所有实现，包括 SimpleDateFormat 都不是线程安全的，因此你不应该在多线程序中使用，除非是在对外线程安全的环境中使用，如 将 SimpleDateFormat 限制在 ThreadLocal 中。如果你不这么做，在解析或者格式化日期的时候，可能会获取到一个不正确的结果。因此，从日期、时间处理的所有实践来说，我强力推荐 joda-time 库。

62）Java 中如何格式化一个日期？如格式化为 ddMMyyyy 的形式？

Java 中，可以使用 SimpleDateFormat 类或者 joda-time 库来格式日期。DateFormat 类允许你使用多种流行的格式来格式化日期。参见答案中的示例代码，代码中演示了将日期格式化成不同的格式，如 dd-MM-yyyy 或 ddMMyyyy。

关于 OOP 和设计模式的面试题

63）接口是什么？为什么要使用接口而不是直接使用具体类？

接口用于定义 API。它定义了类必须得遵循的规则。同时，它提供了一种抽象，因为客户端只使用接口，这样可以有多重实现，如 List 接口，你可以使用可随机访问的 ArrayList，也可以使用方便插入和删除的 LinkedList。接口中不允许写代码，以此来保证抽象，但是 Java 8 中你可以在接口声明静态的默认方法，这种方法是具体的。

64）Java 中，抽象类与接口之间有什么不同？

Java 中，抽象类和接口有很多不同之处，但是最重要的一个是 Java 中限制一个类只能继承一个类，但是可以实现多个接口。抽象类可以很好的定义一个家族类的默认行为，而接口能更好的定义类型，有助于后面实现多态机制。关于这个问题的讨论请查看答案。

65）除了单例模式，你在生产环境中还用过什么设计模式？

这需要根据你的经验来回答。一般情况下，你可以说依赖注入，工厂模式，装饰模式或者观察者模式，随意选择你使用过的一种即可。不过你要准备回答接下的基于你选择的模式的问题。

66）适配器模式是什么？什么时候使用？

适配器模式提供对接口的转换。如果你的客户端使用某些接口，但是你有另外一些接口，你就可以写一个适配去来连接这些接口。

67）构造器注入和 setter 依赖注入，那种方式更好？

每种方式都有它的缺点和优点。构造器注入保证所有的注入都被初始化，但是 setter 注入提供更好的灵活性来设置可选依赖。如果使用 XML 来描述依赖，Setter 注入的可读写会更强。经验法则是强制依赖使用构造器注入，可选依赖使用 setter 注入。

68）依赖注入和工厂模式之间有什么不同？

虽然两种模式都是将对象的创建从应用的逻辑中分离，但是依赖注入比工厂模式更清晰。通过依赖注入，你的类就是 POJO，它只知道依赖而不关心它们怎么获取。使用工厂模式，你的类需要通过工厂来获取依赖。因此，使用 DI 会比使用工厂模式更容易测试。

69）适配器模式和装饰器模式有什么区别？

虽然适配器模式和装饰器模式的结构类似，但是每种模式的出现意图不同。适配器模式被用于桥接两个接口，而装饰模式的目的是在不修改类的情况下给类增加新的功能。

70）适配器模式和代理模式之前有什么不同？

这个问题与前面的类似，适配器模式和代理模式的区别在于他们的意图不同。由于适配器模式和代理模式都是封装真正执行动作的类，因此结构是一致的，但是适配器模式用于接口之间的转换，而代理模式则是增加一个额外的中间层，以便支持分配、控制或智能访问。

71）什么是模板方法模式？

模板方法提供算法的框架，你可以自己去配置或定义步骤。例如，你可以将排序算法看做是一个模板。它定义了排序的步骤，但是具体的比较，可以使用 Comparable 或者其语言中类似东西，具体策略由你去配置。列出算法概要的方法就是众所周知的模板方法。

72）什么时候使用访问者模式？

访问者模式用于解决在类的继承层次上增加操作，但是不直接与之关联。这种模式采用双派发的形式来增加中间层。

73）什么时候使用组合模式？

组合模式使用树结构来展示部分与整体继承关系。它允许客户端采用统一的形式来对待单个对象和对象容器。当你想要展示对象这种部分与整体的继承关系时采用组合模式。

74）继承和组合之间有什么不同？

虽然两种都可以实现代码复用，但是组合比继承共灵活，因为组合允许你在运行时选择不同的实现。用组合实现的代码也比继承测试起来更加简单。

75）描述 Java 中的重载和重写？

重载和重写都允许你用相同的名称来实现不同的功能，但是重载是编译时活动，而重写是运行时活动。你可以在同一个类中重载方法，但是只能在子类中重写方法。重写必须要有继承。

76）Java 中，嵌套公共静态类与顶级类有什么不同？

类的内部可以有多个嵌套公共静态类，但是一个 Java 源文件只能有一个顶级公共类，并且顶级公共类的名称与源文件名称必须一致。

77) OOP 中的 组合、聚合和关联有什么区别？

如果两个对象彼此有关系，就说他们是彼此相关联的。**组合和聚合是面向对象中的两种形式的关联。**组合是一种比聚合更强力的关联。组合中，一个对象是另一个的拥有者，而聚合则是指一个对象使用另一个对象。如果对象 A 是由对象 B 组合的，则 A 不存在的话，B一定不存在，但是如果 A 对象聚合了一个对象 B，则即使 A 不存在了，B 也可以单独存在。

78）给我一个符合开闭原则的设计模式的例子？

开闭原则要求你的代码对扩展开放，对修改关闭。这个意思就是说，如果你想增加一个新的功能，你可以很容易的在不改变已测试过的代码的前提下增加新的代码。有好几个设计模式是基于开闭原则的，如策略模式，如果你需要一个新的策略，只需要实现接口，增加配置，不需要改变核心逻辑。一个正在工作的例子是 Collections.sort() 方法，这就是基于策略模式，遵循开闭原则的，你不需为新的对象修改 sort() 方法，你需要做的仅仅是实现你自己的 Comparator 接口。

79）什么时候使用享元模式？

享元模式通过共享对象来避免创建太多的对象。为了使用享元模式，你需要确保你的对象是不可变的，这样你才能安全的共享。JDK 中 String 池、Integer 池以及 Long 池都是很好的使用了享元模式的例子。

12）Java 中应该使用什么数据类型来代表价格？

如果不是特别关心内存和性能的话，使用BigDecimal，否则使用预定义精度的 double 类型。

13）怎么将 byte 转换为 String？

可以使用 String 接收 byte[] 参数的构造器来进行转换，需要注意的点是要使用的正确的编码，否则会使用平台默认编码，这个编码可能跟原来的编码相同，也可能不同。

14）我们能将 int 强制转换为 byte 类型的变量吗？如果该值大于 byte 类型的范围，将会出现什么现象？

是的，我们可以做强制转换，但是 Java 中 int 是 32 位的，而 byte 是 8 位的，所以，如果强制转化是，int 类型的高 24 位将会被丢弃，byte 类型的范围是从 -128 到 127。

15）Java 中 ++ 操作符是线程安全的吗？

不是线程安全的操作。它涉及到多个指令，读取变量值，增加，然后存储回内存，这个过程可能会出现多个线程交差。

16）a = a + b 与 a += b 的区别？

+= 隐式的将加操作的结果类型强制转换为持有结果的类型。如果两这个整型相加，如 byte、short 或者 int，首先会将它们提升到 int 类型，然后在执行加法操作。如果加法操作的结果比 a 的最大值要大，则 a+b 会出现编译错误，但是 a += b 没问题，如下：

byte a = 127;

byte b = 127;

b = a + b; // error : cannot convert from int to byte

b += a; // ok

注：其实无论 a+b 的值为多少，编译器都会报错，因为 a+b 操作会将 a、b 提升为 int 类型，所以将 int 类型赋值给 byte 就会编译出错

17）我能在不进行强制转换的情况下将一个 double 值赋值给 long 类型的变量吗？

不行，你不能在没有强制类型转换的前提下将一个 double 值赋值给 long 类型的变量，因为 double 类型的范围比 long 类型更广，所以必须要进行强制转换。

18）3\*0.1 == 0.3 将会返回什么？true 还是 false？

false，因为有些浮点数不能完全精确的表示出来。

19）int 和 Integer 哪个会占用更多的内存？

Integer 对象会占用更多的内存。Integer 是一个对象，需要存储对象的元数据。但是 int 是一个原始类型的数据，所以占用的空间更少。

20）为什么 Java 中的 String 是不可变的（Immutable）？

Java 中的 String 不可变是因为 Java 的设计者认为字符串使用非常频繁，将字符串设置为不可变可以允许多个客户端之间共享相同的字符串。

21）Java 中的构造器链是什么？

当你从一个构造器中调用另一个构造器，就是Java 中的构造器链。这种情况只在重载了类的构造器的时候才会出现。

30）JRE、JDK、JVM 及 JIT 之间有什么不同？

JRE 代表 Java 运行时（Java run-time），是运行 Java 引用所必须的。JDK 代表 Java 开发工具（Java development kit），是 Java 程序的开发工具，如 Java 编译器，它也包含 JRE。JVM 代表 Java 虚拟机（Java virtual machine），它的责任是运行 Java 应用。JIT 代表即时编译（Just In Time compilation），当代码执行的次数超过一定的阈值时，会将 Java 字节码转换为本地代码，如，主要的热点代码会被准换为本地代码，这样有利大幅度提高 Java 应用的性能。

Java 基本概念面试题

35）“a==b”和”a.equals(b)”有什么区别？

如果 a 和 b 都是对象，则 a==b 是比较两个对象的引用，只有当 a 和 b 指向的是堆中的同一个对象才会返回 true，而 a.equals(b) 是进行逻辑比较，所以通常需要重写该方法来提供逻辑一致性的比较。例如，String 类重写 equals() 方法，所以可以用于两个不同对象，但是包含的字母相同的比较。

36）a.hashCode() 有什么用？与 a.equals(b) 有什么关系？

hashCode() 方法是相应对象整型的 hash 值。它常用于基于 hash 的集合类，如 Hashtable、HashMap、LinkedHashMap等等。它与 equals() 方法关系特别紧密。根据 Java 规范，两个使用 equal() 方法来判断相等的对象，必须具有相同的 hash code。

37）final、finalize 和 finally 的不同之处？

final 是一个修饰符，可以修饰变量、方法和类。如果 final 修饰变量，意味着该变量的值在初始化后不能被改变。finalize 方法是在对象被回收之前调用的方法，给对象自己最后一个复活的机会，但是什么时候调用 finalize 没有保证。finally 是一个关键字，与 try 和 catch 一起用于异常的处理。finally 块一定会被执行，无论在 try 块中是否有发生异常。

38）Java 中的编译期常量是什么？使用它又什么风险？

公共静态不可变（public static final ）变量也就是我们所说的编译期常量，这里的 public 可选的。实际上这些变量在编译时会被替换掉，因为编译器知道这些变量的值，并且知道这些变量在运行时不能改变。这种方式存在的一个问题是你使用了一个内部的或第三方库中的公有编译时常量，但是这个值后面被其他人改变了，但是你的客户端仍然在使用老的值，甚至你已经部署了一个新的jar。为了避免这种情况，当你在更新依赖 JAR 文件时，确保重新编译你的程序。

**什么是可变参数？**

可变参数允许调用参数数量不同的方法。请看下面例子中的求和方法。此方法可以调用1个int参数，或2个int参数，或多个int参数。

//int(type) followed ... (three dot's) is syntax of a variable argument.

public int sum(int... numbers) {

//inside the method a variable argument is similar to an array.

//number can be treated as if it is declared as int[] numbers;

int sum = 0;

for (int number: numbers) {

sum += number;

}

return sum;

}

public static void main(String[] args) {

VariableArgumentExamples example = new VariableArgumentExamples();

//3 Arguments

System.out.println(example.sum(1, 4, 5));//10

//4 Arguments

System.out.println(example.sum(1, 4, 5, 20));//30

//0 Arguments

System.out.println(example.sum());//0

}

**断言的用途？**

断言是在Java 1.4中引入的。它能让你验证假设。如果断言失败（即返回false），就会抛出AssertionError（如果启用断言）。基本断言如下所示。

private int computerSimpleInterest(int principal,float interest,int years){

assert(principal>0);

return 100;}

**什么时候使用断言？**

断言不应该用于验证输入数据到一个public方法或命令行参数。IllegalArgumentException会是一个更好的选择。在public方法中，只用断言来检查它们根本不应该发生的情况。

**什么是垃圾回收？**

垃圾回收是Java中自动内存管理的另一种叫法。垃圾回收的目的是为程序保持尽可能多的可用堆（heap）。 JVM会删除堆上不再需要从堆引用的对象。

**用一个例子解释垃圾回收？**

比方说，下面这个方法就会从函数调用。

void method(){

Calendar calendar = new GregorianCalendar(2000,10,30);

System.out.println(calendar);}

通过函数第一行代码中参考变量calendar，在堆上创建了GregorianCalendar类的一个对象。

函数结束执行后，引用变量calendar不再有效。因此，在方法中没有创建引用到对象。

JVM认识到这一点，会从堆中删除对象。这就是所谓的垃圾回收。

**什么时候运行垃圾回收？**

垃圾回收在JVM突发奇想和心血来潮时运行（没有那么糟糕）。运行垃圾收集的可能情况是：

堆可用内存不足  
CPU空闲

**垃圾回收的最佳做法？**

用编程的方式，我们可以要求（记住这只是一个请求——不是一个命令）JVM通过调用System.gc()方法来运行垃圾回收。

当内存已满，且堆上没有对象可用于垃圾回收时，JVM可能会抛出OutOfMemoryException。

对象在被垃圾回收从堆上删除之前，会运行finalize()方法。我们建议不要用finalize()方法写任何代码。

**什么是初始化数据块？**

初始化数据块——当创建对象或加载类时运行的代码。

有两种类型的初始化数据块：

静态初始化器：加载类时运行的的代码

实例初始化器：创建新对象时运行的代码

**什么是静态初始化器？**

请看下面的例子：static{ 和 }之间的代码被称为静态初始化器。它只有在第一次加载类时运行。只有静态变量才可以在静态初始化器中进行访问。虽然创建了三个实例，但静态初始化器只运行一次。

public class InitializerExamples {

static int count;

int i;

static{

//This is a static initializers. Run only when Class is first loaded.

//Only static variables can be accessed

System.out.println("Static Initializer");

//i = 6;//COMPILER ERROR

System.out.println("Count when Static Initializer is run is " + count);

}

public static void main(String[] args) {

InitializerExamples example = new InitializerExamples();

InitializerExamples example2 = new InitializerExamples();

InitializerExamples example3 = new InitializerExamples();

}}

示例输出

Static Initializer

Count when Static Initializer is run is 0.

**什么是实例初始化块？**

让我们来看一个例子：每次创建类的实例时，实例初始化器中的代码都会运行。

public class InitializerExamples {

static int count;

int i;

{

//This is an instance initializers. Run every time an object is created.

//static and instance variables can be accessed

System.out.println("Instance Initializer");

i = 6;

count = count + 1;

System.out.println("Count when Instance Initializer is run is " + count);

}

public static void main(String[] args) {

InitializerExamples example = new InitializerExamples();

InitializerExamples example1 = new InitializerExamples();

InitializerExamples example2 = new InitializerExamples();

}}

示例输出

Instance Initializer

Count when Instance Initializer is run is 1

Instance Initializer

Count when Instance Initializer is run is 2

Instance Initializer

Count when Instance Initializer is run is 3

**什么是正则表达式？**

正则表达式能让解析、扫描和分割字符串变得非常容易。Java中常用的正则表达式——Pattern，Matcher和Scanner类。

**什么是令牌化？**

令牌化是指在分隔符的基础上将一个字符串分割为若干个子字符串。例如，分隔符；分割字符串ac;bd;def;e为四个子字符串ac，bd，def和 e。

分隔符自身也可以是一个常见正则表达式。

String.split(regex)函数将regex作为参数。

**给出令牌化的例子？**

private static void tokenize(String string,String regex) {

String[] tokens = string.split(regex);

System.out.println(Arrays.toString(tokens));}

tokenize("ac;bd;def;e",";");//[ac, bd, def, e]

**如何使用扫描器类（Scanner Class）令牌化？**

private static void tokenizeUsingScanner(String string,String regex) {

**Scanner scanner = new Scanner(string);**

**scanner.useDelimiter(regex);**

List<String> matches = new ArrayList<String>();

while(scanner.hasNext()){

matches.add(scanner.next());

}

System.out.println(matches);}

tokenizeUsingScanner("ac;bd;def;e",";");//[ac, bd, def, e]

**如何添加小时(hour)到一个日期对象（Date Objects）？**

现在，让我们如何看看添加小时到一个date对象。所有在date上的日期操作都需要通过添加毫秒到date才能完成。例如，如果我们想增加6个小时，那么我们需要将6小时换算成毫秒。6小时= 6 60 60 \* 1000毫秒。请看以下的例子。

Date date = new Date();

//Increase time by 6 hrs

date.setTime(date.getTime() + 6 \* 60 \* 60 \* 1000);System.out.println(date);

//Decrease time by 6 hrs

date = new Date();

date.setTime(date.getTime() - 6 \* 60 \* 60 \* 1000);System.out.println(date);

**如何格式化日期对象？**

格式化日期需要使用DateFormat类完成。让我们看几个例子。

//Formatting DatesSystem.out.println(DateFormat.getInstance().format(

date));//10/16/12 5:18 AM

带有区域设置的格式化日期如下所示：

System.out.println(DateFormat.getDateInstance(

DateFormat.FULL, new Locale("it", "IT"))

.format(date));//marted“ 16 ottobre 2012

System.out.println(DateFormat.getDateInstance(

DateFormat.FULL, Locale.ITALIAN)

.format(date));//marted“ 16 ottobre 2012

//This uses default locale USSystem.out.println(DateFormat.getDateInstance(

DateFormat.FULL).format(date));//Tuesday, October 16, 2012

System.out.println(DateFormat.getDateInstance()

.format(date));//Oct 16, 2012System.out.println(DateFormat.getDateInstance(

DateFormat.SHORT).format(date));//10/16/12System.out.println(DateFormat.getDateInstance(

DateFormat.MEDIUM).format(date));//Oct 16, 2012

System.out.println(DateFormat.getDateInstance(

DateFormat.LONG).format(date));//October 16, 2012

**Java中日历类（Calendar Class）的用途？**

Calendar类（Youtube视频链接 - [https://www.youtube.com/watch?v=hvnlYbt1ve0](https://www.youtube.com/watch?v=hvnlYbt1ve0" \t "http://www.imooc.com/article/_blank)） 在Java中用于处理日期。Calendar类提供了增加和减少天数、月数和年数的简便方法。它还提供了很多与日期有关的细节（这一年的哪一天？哪一周？等等）

**如何在Java中获取日历类（Calendar Class）的实例？**

Calendar类不能通过使用new Calendar创建。得到Calendar类实例的最好办法是在Calendar中使用getInstance() static方法。

//Calendar calendar = new Calendar(); //COMPILER ERROR

**Calendar calendar = Calendar.getInstance();**

**解释一些日历类（Calendar Class）中的重要方法？**

在Calendar对象上设置日（day），月（month）或年（year）不难。对Day，Month或Year调用恰当Constant的set方法。下一个参数就是值。

calendar.set(Calendar.DATE, 24);

calendar.set(Calendar.MONTH, 8);//8 - September

calendar.set(Calendar.YEAR, 2010);

calendar get方法

要获取一个特定日期的信息——2010年9月24日。我们可以使用calendar get方法。已被传递的参数表示我们希望从calendar中获得的值—— 天或月或年或……你可以从calendar获取的值举例如下：

System.out.println(calendar.get(Calendar.YEAR));//2010

System.out.println(calendar.get(Calendar.MONTH));//8

System.out.println(calendar.get(Calendar.DATE));//24

System.out.println(calendar.get(Calendar.WEEK\_OF\_MONTH));//4

System.out.println(calendar.get(Calendar.WEEK\_OF\_YEAR));//39

System.out.println(calendar.get(Calendar.DAY\_OF\_YEAR));//267

System.out.println(calendar.getFirstDayOfWeek());//1 -> Calendar.SUNDAY

**数字格式化类（Number Format Class）的用途？**

数字格式用于格式化数字到不同的区域和不同格式中。

使用默认语言环境的数字格式

System.out.println(NumberFormat.getInstance().format(321.24f));//321.24

使用区域设置的数字格式

使用荷兰语言环境格式化数字：

System.out.println(NumberFormat.getInstance(new Locale("nl")).format(4032.3f));//4.032,3

使用德国语言环境格式化数字：

System.out.println(NumberFormat.getInstance(Locale.GERMANY).format(4032.3f));//4.032,3

使用默认语言环境格式化货币

System.out.println(NumberFormat.getCurrencyInstance().format(40324.31f));//$40,324.31

使用区域设置格式化货币

使用荷兰语言环境格式化货币：

System.out.println(NumberFormat.getCurrencyInstance(new Locale("nl")).format(40324.31f));//? 40.324,31

"原子操作(atomic operation)是不需要synchronized"，这是Java多线程编程的老生常谈了。所谓原子操作是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch （切[1] 换到另一个线程）。

**foreach循环的原理:**

**在编译的时候编译器会自动将对for这个关键字的使用转化为对目标的迭代器的使用**

**进而，我们再得出两个结论：**

**1、ArrayList之所以能使用foreach循环遍历，是因为ArrayList所有的List都是Collection的子接口，而Collection是Iterable的子接口，ArrayList的父类AbstractList正确地实现了Iterable接口的iterator方法。之前我自己写的ArrayList用foreach循环直接报空指针异常是因为我自己写的ArrayList并没有实现Iterable接口**

**2、任何一个集合，无论是JDK提供的还是自己写的，只要想使用foreach循环遍历，就必须正确地实现Iterable接口**

**实际上，这种做法就是23中设计模式中的迭代器模式。**

**对于数组的foreach循环转换为对于这个数组每一个的循环引用。**

**这里我们只要知道下面的事实就好了：**

**For-each语法内部，对collection是用nested iteratoration来实现的，对数组是用下标遍历来实现。**

**Java 5 及以上的编译器隐藏了基于iteration和下标遍历的内部实现。（注意，这里说的是“Java编译器”或Java语言对其实现做了隐藏，而不是某段Java代码对其实现做了隐藏，也就是说，我们在任何一段JDK的Java代码中都找不到这里被隐藏的实现。这里的实现，隐藏在了Java 编译器中，我们可能只能像这篇帖子中说的那样，查看一段For-each的Java代码编译成的字节码，从中揣测它到底是怎么实现的了）**

结论：

效率：System.arraycopy > clone > Arrays.copyOf > for循环

理由：

A：for循环，效率最低，随便写个程序验证一下，效率慢的不是一点.....我测试的时候比clone和System.arraycopy差了100多倍

B：System.arraycopy：原型是

public static native void arraycopy(Object src, int srcPos , Object dest, int destPos, int length);

C：Arrays.copyOf底层调用了上面的System.arraycopy效率比上面两个低。

D：clone()的完整定义：protected native Object clone() throws CloneNotSupportedException;

高票答案说的clone()返回的是Object类型，其实是错误的，只有Object[]数组的clone()方法才返回Object类型，子类重写了父类的方法。

其实，一般情况下，前三个的效率差别几乎忽略不计，但是从Arrays.copyOf底层调用的是System.arraycopy，自然Arrays.copyOf效率自然要低一些。

而clone()和System.arraycopy只是从实验的结果来看是System.arraycopy的效率高。

**请描述一下jvm加载class文件的原理机制？？**答：JVM中类的装载是由类加载器（ClassLoader）和它的子类来实现的，Java中的类加载器是一个重要的Java运行时系统组件，它负责在运行时查找和装入类文件中的类。

JVM中类加载的机制——双亲委派模型。这个模型要求除了Bootstrap ClassLoader外，其余的类加载器都要有自己的父加载器。**子加载器通过组合来复用父加载器的代码，而不是使用继承。**在某个类加载器加载class文件时，它首先委托父加载器去加载这个类，依次传递到顶层类加载器(Bootstrap)。如果顶层加载不了（它的搜索范围中找不到此类），子加载器才会尝试加载这个类。

**双亲委派模型最大的好处就是让Java类同其类加载器一起具备了一种带优先级的层次关系。**这句话可能不好理解，我们举个例子。比如我们要加载顶层的Java类——java.lang.Object类，无论我们用哪个类加载器去加载Object类，这个加载请求最终都会委托给Bootstrap ClassLoader，这样就保证了所有加载器加载的Object类都是同一个类。如果没有双亲委派模型，那就乱了套了，完全可以搞出Root::Object和L1::Object这样两个不同的Object类。  
由于Java的跨平台性，经过编译的Java源程序并不是一个可执行程序，而是一个或多个类文件。当Java程序需要使用某个类时，JVM会确保这个类已经被加载、连接（验证、准备和解析）和初始化。类的加载是指把类的.class文件中的数据读入到内存中，通常是创建一个字节数组读入.class文件，然后产生与所加载类对应的Class对象。加载完成后，Class对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。最后JVM对类进行初始化，包括：1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；2)如果类中存在初始化语句，就依次执行这些初始化语句。  
类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader的子类）。从Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。**PDM更好的保证了Java平台的安全性**，在该机制中，JVM自带的Bootstrap是根加载器，其他的加载器都有且仅有一个父加载器。类的加载首先请求父加载器加载，父加载器无能为力时才由其子类加载器自行加载。JVM不会向Java程序提供对Bootstrap的引用。下面是关于几个类加载器的说明：

Bootstrap：一般用本地代码实现，负责加载JVM基础核心类库（rt.jar）jre/lib/rt.jar；

Extension：从java.ext.dirs系统属性所指定的目录中加载类库，它的父加载器是Bootstrap；jre/lib/ext/

System：又叫应用类加载器，其父类是Extension。它是应用最广泛的类加载器。它从环境变量classpath或者系统属性java.class.path所指定的目录中记载类，是用户自定义加载器的默认父加载器。

**委托机制的意义 — 防止内存中出现多份同样的字节码**

比如两个类A和类B都要加载System类：

如果不用委托而是自己加载自己的，那么类A就会加载一份System字节码，然后类B又会加载一份System字节码，这样内存中就出现了两份System字节码。

如果使用委托机制，会递归的向父类查找，也就是首选用Bootstrap尝试加载，如果找不到再向下。这里的System就能在Bootstrap中找到然后加载，如果此时类B也要加载System，也从Bootstrap开始，此时Bootstrap发现已经加载过了System那么直接返回内存中的System即可而不需要重新加载，这样内存中就只有一份System的字节码了。

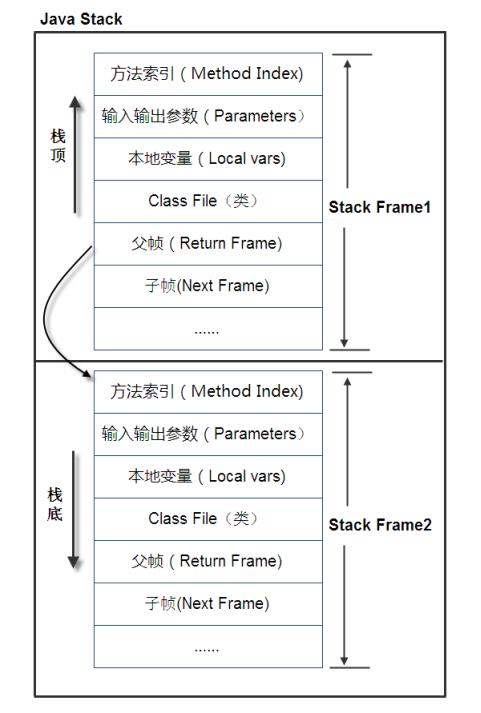
**能不能自己写个类叫java.lang.System？**

答案：**通常不可以**，但可以采取另类方法达到这个需求。

解释：**为了不让我们写System类，类加载采用委托机制**，这样可以保证爸爸们优先，爸爸们能找到的类，儿子就没有机会加载。而System类是Bootstrap加载器加载的，就算自己重写，也总是使用Java系统提供的System，自己写的System类根本没有机会得到加载。

但是，我们可以**自己定义一个类加载器来达到这个目的类，为了避免双亲委托机制**，这个类加载器也必须是特殊的。由于系统自带的三个类加载器都加载特定目录下的类，如果我们自己的类加载器放在一个特殊的目录，那么系统的加载器就无法加载，也就是最终还是由我们自己的加载器加载。

**（个人理解不一定准确）java堆（JavaHeap）**1.用来存放对象的，几乎所有对象都放在这里，被线程共享的，或者说是被栈共享的2.堆又可以分为新生代和老年代，实际还有一个区域叫永久代，但是jdk1.7已经去永久代了，所以可以当作没有，永久代是当jvm启动时就存放的JDK自身的类和接口数据，关闭则释放。新生代可以分为Eden区和两个幸存区，这么设计是为了更好地利用内存。之前的设计是只分为两部分一样一半 后来发现这样只利用到了一半的内存 才改为按比例分成三个区的，使用的是复制回收算法，两个幸存区是较小的区域。逻辑是每次使用Eden区和其中一个幸存区，回收时将其还存活着的对象一次性的复制到另一个幸存区中，最后清理掉刚才使用的Eden和其中一个幸存区。美团的面试官也问了这个问题，他也说了他的理解，我感觉可能是不准确的。新建对象就在Eden区，Eden就是伊甸，顾名思义。但是并不是对象最活跃的区域，对象最活跃的区域是老年代，因为经过各种垃圾回收之后对象都跑到这里来了。3.内存溢出内存溢出其实没什么好讲的，满了就会溢出。怎么才能满呢，不断创建对象，那问题又来了，创建多了被回收怎么办，好办，将新建的对象存到list里去，就不会回收了，为什么呢，因为jvm判定一个对象的死活就是根据对象是不是被引用。此外堆跟随jvm的，有jvm就有堆。堆也是垃圾回收的主要区域，又叫GC堆，垃圾堆，玩笑。jvm栈1.要说栈是用来存什么的，其实我感觉不严谨，栈是运行时创建的，是跟随线程的，它不是用来存什么的，那它用来干什么的，它是用来存栈帧的，没有图不太好说呢，等下我去截个图。



图来了我就不用多说了。每个栈帧其实可以理解为一个方法，我是这么理解的，之间的关系就是调用。2.栈的好处就是不需要垃圾回收，随着线程结束内存就释放。3.但是并不是说就不会内存溢出，那么栈的内存溢出是怎么产生的呢，肯定也是满了，这个满了怎么理解呢，一是要申请的不够了，二是jvm内存太小，这是个有趣的问题。但是产生的错误却是不一样的，如果创建一个void方法调用自身，错误是stackoverflowError，如果不断创建线程则会outOfMemoryError。这里就有一个比较高级的问题了，对于第二种多线程内存溢出该怎么解决呢，深入理解jvm一书中给出的解决方案是这样的，通过减小最大堆和栈容量来换取更多的线程。

**方法区和运行时常量池**1.方法区是堆的一个逻辑区域，但是又叫非堆。运行时常量池又是方法区的一部分，真正的一部分。方法区并不是存方法的，存方法的应该是栈或者栈帧。方法区存的是类信息、常量、静态变量等，也是被线程共享的区域。运行时常量池存放的是编译期生产的各种字面量和符号引用。2.这块内存区域的回收没啥好说的，因为我也不太清楚，我只知道HotSpot的设计团队选择把GC分代扩展至方法区了，或者是使用永久代实现方法区。3.内存是肯定会溢出的，不断创建类会导致方法区内存溢出，而不断将常量放入常量池（String.intern()），常量池也会内存溢出。内存是jvm的重点区域，也是jvm优化的重点区域，既然这么重要一下子也不可能写完了，慢慢学习补充吧。

30、GC是什么？为什么要有GC？  
答：GC是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java提供的GC功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java语言没有提供释放已分配内存的显示操作方法。Java程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：System.gc() 或Runtime.getRuntime().gc() ，但JVM可以屏蔽掉显式的垃圾回收调用。  
垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。在Java诞生初期，垃圾回收是Java最大的亮点之一，因为服务器端的编程需要有效的防止内存泄露问题，然而时过境迁，如今Java的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得iOS的系统比Android系统有更好的用户体验，其中一个深层次的原因就在于Android系统中垃圾回收的不可预知性。

**补充：**垃圾回收机制有很多种，包括：分代复制垃圾回收、标记垃圾回收、增量垃圾回收等方式。标准的Java进程既有栈又有堆。栈保存了原始型局部变量，堆保存了要创建的对象。Java平台对堆内存回收和再利用的基本算法被称为标记和清除，但是Java对其进行了改进，采用“分代式垃圾收集”。这种方法会跟Java对象的生命周期将堆内存划分为不同的区域，在垃圾收集过程中，可能会将对象移动到不同区域：  
- 伊甸园（Eden）：这是对象最初诞生的区域，并且对大多数对象来说，这里是它们唯一存在过的区域。  
- 幸存者乐园（Survivor）：从伊甸园幸存下来的对象会被挪到这里。  
- 终身颐养园（Tenured）：这是足够老的幸存对象的归宿。年轻代收集（Minor-GC）过程是不会触及这个地方的。当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC），这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。

与垃圾回收相关的JVM参数：

* -Xms / -Xmx — 堆的初始大小（最小尺寸） / 堆的最大大小

**记住：Java会尽量的维持在最小堆运行，即使设置的最大值很大，只有当GC之后也无法满足最小堆，才会去扩容。**

* -Xmn — 堆中新生代的大小
* -XX:-DisableExplicitGC — 让System.gc()不产生任何作用
* -XX:+printGC参数来使能JVM的GC日志打印
* -XX:+PrintGCDetails — 打印GC的细节
* -XX:+PrintGCDateStamps — 打印GC操作的时间戳
* -XX:NewSize / XX:MaxNewSize — 设置新生代大小/新生代最大大小
* -XX:NewRatio — 可以设置老生代和新生代的比例
* -XX:PrintTenuringDistribution — 设置每次新生代GC后输出幸存者乐园中对象年龄的分布
* -XX:InitialTenuringThreshold/ -XX:MaxTenuringThreshold：设置老年代阀值的初始值和最大值
* -XX:TargetSurvivorRatio：设置幸存区的目标使用率
* 设置新生代大小，-Xmn参数，设置的是绝对值，30m就是30m，10m就是10m。还有一个参数 -XX:NewRatio，看名字就知道是按照比例来设置，意思是设置新生代（eden+2\*s）和老年代（不包含永久区）的比值，比如-XX:NewRatio4 表示 新生代:老年代=1:4。

设置两个Survivor区（s0，s1或者from和to）和eden的比例 -XX:SurvivorRatio，比如-XX:SurvivorRatio8表示两个Survivor : eden=2:8，即一个Survivor占年轻代的1/10。

-Xloggc:log/gc.log，指定GC log的位置，把GC日志输出到工作空间的log文件夹下的gc.log文件中，能更加方便的帮助开发人员分析问题。

打印最详细的GC堆的日志： -XX:+PrintHeapAtGC　　意思是每次记录GC日志，前后都要打印Java堆的详细信息。

监控Java类的加载情况： -XX:+TraceClassLoading 监控系统中每一个类的加载，每一行代表一个类，主要用于跟踪调试程序。

监控类的使用情况：-XX:+PrintClassHistogram

OOM时导出堆到文件进行内存分析和问题排查

　　-XX:+HeapDumpOnOutOfMemoryError，-XX:+HeapDumpPath 导出OOM的路径，比如：

-Xmx20m -Xms5m -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/a.dump。

**堆的分配参数总结**

1. 根据实际事情调整新生代和幸存代的大小

2. 官方推荐新生代占堆的3/8

3. 幸存代占新生代的1/10

4. 在OOM时，记得Dump出堆，确保可以排查现场问题

**永久区分配参数**

-XX:PermSize ， -XX:MaxPermSize，设置永久区的初始空间和最大空间，他们表示，一个系统可以容纳多少个类型。类似-Xms和-Xmx。当使用一些框架时，会产生大量的类，这样的类越来越多，会可能挤爆永久区，导致OOM。也就是说如果堆没有用完（实际堆的空间占用很少），也抛出了OOM错误，很有可能是永久区导致的OOM问题。

**栈大小分配 -Xss**

通常只有几百K，一般很少调大，它的大小决定了函数调用的深度，之前也说了每个线程都有独立的栈空间，保存了局部变量、参数等。如果想跑更多的线程，需要把栈用-Xss尽量调小，而不是变大！因为线程越多，每个线程都要分配内存空间，这样每个栈的空间越大，占据的内存越多……但是也得预防很深的函数调用可能导致栈内存溢出问题，比如不合适的递归调用。

**Java为什么不用他呢，因为引用计数法有很多缺点**：

1. 性能，每次引用和去引用都要加减

2. 循环引用问题

现代Java的垃圾回收使用的基本的算法思想是标记-清除算法：

**标记-清除算法是现代垃圾回收算法的思想基础**。将垃圾回收分为两个阶段：标记阶段和清除阶段。一种可行的实现是，在标记阶段，首先通过根节点，标记所有从根节点开始的可达对象（从GC ROOT开始标记引用链——又叫可达性算法）。因此，未被标记的对象就是未被引用的垃圾对象。然后，在清除阶段，清除所有未被标记的对象。这样就不怕循环问题了。

PS：Java中可以作为GC ROOT的对象有：

1. 方法区中类静态变量引用的对象

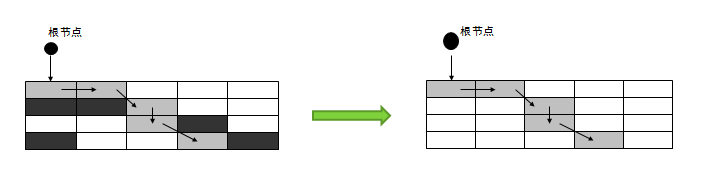
2. 方法区中常量引用的对象

3. 本地方法栈（JNI，即一般说的Native方法）引用的对象

4. 虚拟机栈（栈帧中的本地变量表）引用的对象

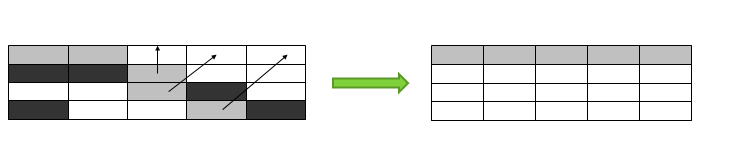
即作为GC Roots的结点主要在全局性的引用（例如常量和类静态属性）与执行上下文（例如栈帧中的本地变量表）中，虚拟机栈、本地方法栈这都是局部变量，某个方法执行完，某些局部使用的对象可以被回收。

如图，从根节点能到达的都是不能回收的，是被引用的。标记下。而这种算法的缺点就是容易出现内存碎片。利用率不高。



**要知道，现代的Java虚拟机都是使用的分代回收的设计**，比如在标记-清除算法的基础上做了一些优化的——标记-压缩算法，适合用于存活对象较多的场合，如老年代。

和标记-清除算法一样，标记-压缩算法也首先需要从根节点开始，对所有可达对象做一次标记。但之后，它并不简单的清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。有效解决内存碎片问题。



**还有一个算法，针对新生代的回收，叫复制算法**

**和标记-清除算法相比，复制算法是一种相对高效的回收方法，但是 不适用于存活对象较多的场合如老年代，使用在新生代**， 原理是 将原有的内存空间分为两块，两块空间完全相同，每次只用一块，在垃圾回收时，将正在使用的内存中的存活对象复制到未使用的内存块中，之后，清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收。同样也没有内存 碎片产生。   **复制算法的缺点是内存的浪费，因为每次只是使用了一半的空间**， 而大多数存活对象都在老年代，故复制算法不用在老年代，老年代是Java堆的空间的担保地区。复制算法主要用在新生代。在垃圾回收的时候，**大对象直接从新生代进入了老年代存放，大对象一般不使用复制算法，因为一是太大，复制效率低，二是过多的大对象，会使得小对象复制的时候无地方存放**。**还有被长期引用的对象也放在了老年代**。 Java的垃圾回收机制使用的是分代的思想。 依据对象的存活周期进行分类，短命对象归为新生代，长命对象归为老年代。根据不同代的特点，选取合适的收集算法。少量对象存活（新生代，朝生夕死的特性），适合复制算法，大量对象存活（老年代，生命周期很长，甚至和应用程序存放时间一样），适合标记清理或者标记压缩算法。 以上一定注意：Java没有采用引用计数算法！   经过上述总结，想到所有的算法，需要能够识别一个垃圾对象，那么怎么才能识别呢？ 因此需要给出一个可触及性的定义：

**可触及的**–从GC ROOT这个根节点对象，沿着引用的链条，可以触及到这个对象，该对象就叫可触及的，也就是之前说的可达性算法的思想。

**可复活的**–一旦所有引用被释放，就是可复活状态，因为在finalize()中可能复活该对象（finalize方法只会调用一次）。

**不可触及的**–在finalize()后，可能会进入不可触及状态，不可触及的对象不可能复活，就可以回收了。

**引出一个方法的理解：finalize方法**

GC准备释放内存的时候，会先调用finalize()。而调用了这个方法不代表对象一定会被回收。因为GC和finalize() 都是靠不住的，只要JVM还没有快到耗尽内存的地步，它是不会浪费时间进行垃圾回收的。

**finalize()在什么时候被调用?**

　有三种情况

1. 所有对象被Garbage Collection时自动调用,比如运行System.gc()的时候。

2. 程序退出时为每个对象调用一次finalize方法。

3. 显式的调用finalize方法。

finalize 是Object的 protected 方法，子类可以覆盖该方法以实现资源清理工作，**GC在回收对象之前调用该方法。**finalize与C++中的析构函数不是对应的。C++中的析构函数调用的时机是确定的（对象离开作用域或delete掉），但Java中的finalize的调用具有不确定性。

**不建议用finalize方法完成“非内存资源”的清理工作**，**因为Java语言规范并不保证finalize方法会被及时地执行、而且根本不会保证它们会被执行，而且 finalize 方法可能会带来性能问题。**因为JVM通常在单独的低优先级线程中完成finalize的执行，finalize方法中，可将待回收对象赋值给GC Roots可达的对象引用，从而达到对象再生的目的。**finalize方法至多由GC执行一次**(用户当然可以手动调用对象的finalize方法，但并不影响GC对finalize的行为)

**但建议用于：**

1. 清理本地对象(通过JNI创建的对象)；

2. 作为确保某些非内存资源(如Socket、文件等)释放的一个补充：在finalize方法中显式调用其他资源释放方法。

**说到这里，不得不提下Java的四种引用类型:**

前面说了，GC是分代的，GC的的回收条件取决于识别该对象是不是垃圾。而识别垃圾对象又取决于指向该对象的引用类型。**Java中有四种引用类型，强，软，弱，虚。 如果一个对象只有弱引用指向它，GC会立即回收该对象，这是一种急切回收方式。相对的，如果有软引用指向这些对象，则只有在JVM需要内存时才回收这些对象。**弱引用和软引用的特殊行为使得它们在某些情况下非常有用。 例如：**软引用可以很好的用来实现缓存**，当JVM需要内存时，垃圾回收器就会回收这些只有被软引用指向的对象。而**弱引用非常适合存储元数据**，例如：存储ClassLoader引用。如果没有类被加载，那么也没有指向ClassLoader的引用。一旦上一次的强引用被去除，只有弱引用的ClassLoader就会被回收。

**1. 强引用：**类似我们常见的，比如 A a = new A（）；a就叫强引用。任何被强引用指向的对象都不能GC，这些对象都是在程序中需要的。

**2. 软引用：**使用java.lang.ref.SoftReference类来表示，软引用可以很好的用来实现缓存，当JVM需要内存时，垃圾回收器就会回收这些只有被软引用指向的对象。如下：

Counter prime = new Counter();

SoftReference soft = new SoftReference(prime) ; //soft reference

prime = null;

　　强引用置空之后，代码的第二行为对象Counter创建了一个软引用，该引用同样不能阻止垃圾回收器回收对象，但是可以延迟回收，**软引用更适用于缓存机制，而弱引用更适用于存贮元数据。**

**3. 弱引用：**使用java.lang.ref.WeakReference 类来表示，**弱引用非常适合存储元数据**，例如：存储ClassLoader引用。如果没有类被加载，那么也没有指向ClassLoader的引用。一旦上一次的强引用被去除，只有弱引用的ClassLoader就会被回收。也就是说**如果一个对象只有弱引用指向它，GC会立即回收该对象，这是一种急切回收方式**。如：

Counter counter = new Counter(); // strong reference

WeakReference<Counter> weakCounter = new WeakReference<Counter>(counter); //weak reference

counter = null;

　　只要给强引用对象counter赋null,该对象就可以被垃圾回收器回收。因为该对象不再含有其他强引用，即使指向该对象的弱引用weakCounter也无法阻止垃圾回收器对该对象的回收。相反的，如果该对象含有软引用，Counter对象不会立即被回收，除非JVM需要内存。

　　另一个使用弱引用的例子是WeakHashMap，它是除HashMap和TreeMap之外，Map接口的另一种实现。**WeakHashMap有一个特点：map中的键值(keys)都被封装成弱引用，也就是说一旦强引用被删除，WeakHashMap内部的弱引用就无法阻止该对象被垃圾回收器回收。**

1. **虚引用：**没什么实际用处，就是一个标志，当GC的时候好知道。拥有虚引用的对象可以在任何时候GC。

**虚引用主要用来跟踪对象被垃圾回收器回收的活动**。虚引用与软引用和弱引用的一个区别在于：**虚引用必须和引用队列 （ReferenceQueue）联合使用**。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

ReferenceQueue queue = new ReferenceQueue ();

PhantomReference pr = new PhantomReference (object, queue);

程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

　　除了了解弱引用、软引用、虚引用和WeakHashMap，还需要了解ReferenceQueue。在创建任何弱引用、软引用和虚引用的过程中，可以通过如下代码提供引用队列ReferenceQueue：

ReferenceQueue refQueue = new ReferenceQueue();

DigitalCounter digit = new DigitalCounter();

PhantomReference<DigitalCounter> phantom = new PhantomReference<DigitalCounter>(digit, refQueue);

引用实例被添加在引用队列中，可以在任何时候通过查询引用队列回收对象。

**引要方法**。弱引用有个构造函数取引用队列作为参数。如果用关联的引用队列创建弱引用，在弱引用对象成为 GC 候选对象时，这个引用对象就在引用清除后加入到引用队列中

我们希望当一个对象被gc掉的时候通知用户线程，进行额外的处理时，就需要使用引用队列了。ReferenceQueue即这样的一个对象，当一个obj被gc掉之后，其相应的包装类**用队列是垃圾收集器向应用程序返回关于对象生命周期的信息的主**，即ref对象会被放入queue中。我们可以从queue中获取到相应的对象信息，同时进行额外的处理。比如反向操作，数据清理等。

消除陈旧对象时，提到了**weakHashMap**，看了下还是适用的，即在我们使用短时间内就过期的缓存时最好使用weakHashMap，**它包含了一个自动调用的方法expungeStaleEntries，这样就会在值被引用后直接执行这个隐含的方法，将不用的键清除掉。**

注意：**WeakHashMap并不是你啥也不干他就能自动释放内部不用的对象的，而是在你访问它的内容的时候释放内部不用的对象。**这两句话看似区别不大，但是有时候一个小小的区别就会要了命的。就是说你只put 了压根没有get过，这个值是永远都存在的。

我们也可以看下这个移除键值的实现

1. **private** **void** expungeStaleEntries() {
2. Entry<K,V> e;
3. **while** ( (e = (Entry<K,V>) queue.poll()) != **null**) {
4. **int** h = e.hash;
5. **int** i = indexFor(h, table.length);
7. Entry<K,V> prev = table[i];
8. Entry<K,V> p = prev;
9. **while** (p != **null**) {
10. Entry<K,V> next = p.next;
11. **if** (p == e) {
12. **if** (prev == e)
13. table[i] = next;
14. **else**
15. prev.next = next;
16. e.next = **null**;  // Help GC
17. e.value = **null**; //  "   "
18. size--;
19. **break**;
20. }
21. prev = p;
22. p = next;
23. }
24. }
25. }

就是使用了链表，找到这个hash值  将这个hash值移除 size减少

**现在我对一个对象的生命周期进行描述：**

新建Java对象A首先处于可达的，未执行finalize方法的状态，随着程序的运行，一些引用关系会消失，或者变迁，当对A使用可达性算法判断，对象A变成了 GC Roots 不可达时，A从可达状态变迁到不可达状态，但是JVM不会就就这样把它清理了，而是在第一次GC的时候，对它首先进行一个标记（标记清除算法），之后最少还要再进行一次筛选，而对其筛选的的条件就是看该对象是否覆盖了Object的finalize方法，或者看这个对象是否执行过一次finalize方法。如果没有执行，也没有覆盖，就满足筛选条件，JVM将其放入F-Queue队列，由JVM的一个低优先级的线程执行该队列中对象的finalize方法。此时执行finalize方法优先级是很低的，且不会保证等待finalize方法执行完毕才进行第二次回收（怕发生无限等待的情景，JVM崩溃），之后不久GC对队列里的对象进行二轮回收，去判断该对象是否可达，若不可达，才进行回收，否则，对象“复活”（执行finalize的过程中，应用程序是可以让对象再次被引用，复活的）。而在可达性判断的时候，还要兼顾四种引用类型，根据不同的引用类型特点去判断是否是回收的对象。

package wys.demo1;

public class Demo1 {

public static Demo1 obj;

@Override

protected void finalize() throws Throwable {

super.finalize();

System.out.println("CanReliveObj finalize called");

obj = this;// 把obj复活了！！！

}

@Override

public String toString(){

return "I am CanReliveObj";

}

public static void main(String[] args) throws InterruptedException{

obj = new Demo1();// 强引用

obj = null; //不会被立即回收，是可复活的对象

System.gc();// 主动建议JVM做一次GC，GC之前会调用finalize方法，而我在里面把obj复活了！！！

Thread.sleep(1000);

if(obj == null){

System.out.println("obj 是 null");

}else{

System.out.println("obj 可用");

}

System.out.println("第二次gc");

obj = null; //不可复活

System.gc();

Thread.sleep(1000);

if(obj == null){

System.out.println("obj 是 null");

}else{

System.out.println("obj 可用");

}

}

}

结果：

CanReliveObj finalize called

obj 可用

第二次gc

obj 是 null

　　说明JVM不管程序员手动调用finalize，JVM它就是执行一次finalize方法。执行finalize方法完毕后，GC会再次进行二轮回收，去判断该对象是否可达，若不可达，才进行回收。

　　建议：避免使用finalize方法！

　　太复杂了，还是让系统照管比较好。可以定义其它的方法来释放非内存资源。建议使用try-catch-finally来替代它执行清理操作。

　　如果手动调用了finalize，很容易出错。且它执行的优先级低，何时被调用，不确定——也就是何时发生GC不确定，因为只有当内存告急时，GC才工作，即使GC工作，finalize方法也不一定得到执行，这是由于程序中的其他线程的优先级远远高于执行finalize（）的线程优先级。 因此当finalize还没有被执行时，系统的其他资源，比如文件句柄、数据库连接池等已经消耗殆尽，造成系统崩溃。且垃圾回收和finalize方法的执行本身就是对系统资源的消耗，有可能造成程序的暂时停止，因此在程序中尽量避免使用finalize方法。

　　上面提到了GC或者执行finalize可能造成程序暂停，这引出一个概念：**Stop-The-World现象。**

　　这是Java中一种全局暂停的现象，全局停顿，所有Java代码停止，类似JVM挂起的状态……但是native代码可以执行，但不能和JVM交互。**这多半由于GC引起，其他的引起原因比如：**

**1. Dump线程**

**2. JVM的死锁检查**

**3. 堆的Dump。**

**这三者出现概率很低，多半是程序员手动引起的，而GC是JVM自动引起的。**

**GC时为什么会有全局停顿？**

　　类比在聚会时打扫房间，聚会时很乱，又有新的垃圾产生，房间永远打扫不干净，只有让大家停止活动了，才能将房间在某一个状态下打扫干净。回程序中就是只有程序暂停了，才能全面，完整，正确的清理一次垃圾对象，否则前脚清理了，后脚还有新的，永远清理不完，对判断垃圾对象也是一个判断上干扰的问题，也永远干净不了。

**Stop-The-World现象危害**

**长时间服务停止，没有响应，一般新生代的GC停顿时间很短，零点几秒。而老年代比较时间长，几秒甚至几十分钟……一般堆内存越大，GC时间越长，也就是Stop-The-World越久。所以，JVM的内存不是越大越好，要根据实际情况设置。**

　　遇到HA系统，可能引起主备切换，严重危害生产环境。比如一个系统，一个主机服务器，一个备机服务器，不会同时启动，我们会只使用一个，比如主机暂时因为GC没有响应，如果时间太长，我们会使用备机，一旦主机恢复了，主机也启动了，此时备机主机都启动了，很可能导致服务器数据不一致……

-Xmx 和 –Xms 应该保持一个什么关系，可以让系统的性能尽可能的好呢？是不是虚拟机内存越大越好？

答：占坑，后续的GC机制来补充回答这个问题。首先并不是虚拟机内存越大就越好，大概原因是因为：内存越大，JVM 进行 Full GC 所需的时间越久，由于 Full GC 时 stop whole world 特性，如果是用于响应HTTP 请求的服务器，这个时候就表现为停止响应，对于需要低延迟的应用来说，这是不可接受的。对于需要高吞吐量的应用来说，可以不在乎这种停顿，比如一些后台的应用之类的，那么内存可以适当调大一些。需要根据具体情况权衡。

Java堆整体分两代，新生代和老年代，顾名思义，前者存放新生对象，大部分都是朝生夕死！进行GC的次数不多，后者存放的是时间比较久的对象，也就是多次GC还没死的对象。对象创建的时候，大部分都是放入新生代的eden区，除非是很大的对象，可能会直接存放到老年代，还有之前说的栈上分配（逃逸分析）。

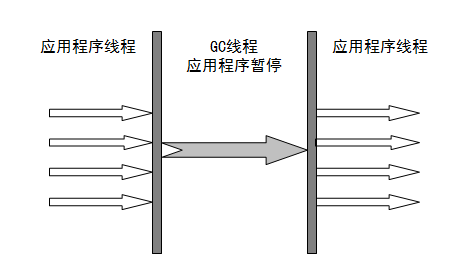
Java 7 堆结构分为了eden伊甸园，两个幸存代survivor，前三者也叫年轻代，其次是老年代old和永久代permanent。一个Java对象被创建，先是存在于eden，如果存活时间超过了两个幸存代就转移到老年代保存，而永久带保存了对象的方法，变量等元数据，如果永久带没地方了就会发生内存泄漏异常错误OutOfMemeoryError：PermGen。

Java 8的堆内存结构有变化，移除了永久带，也就是不再有OutOfMemeoryError：PermGen错误了。新加了元数据区，和对应的参数-XX:MaxMetaspaceSize。

如果eden对象在GC时幸存，就会进入幸存区，也就是s0，s1，或者叫from和to，或者叫survivor（两个），大小一样。完全对称，功能也一样。前面说了GC有复制算法，那么就是使用在这里，GC在新生代时，eden区的存活对象被复制到未使用的幸存区，假设是to，而正在使用的是from区的年轻的对象也会一起被复制到了to区，如果to区满了，这些对象也和大对象，老年对象一样直接进入了老年代保存（担保空间）。此时，eden区剩余的对象和from区剩余的对象就是垃圾对象，能直接GC，to区存放的是新生代的此次GC活下来的对象。避免了产生内存碎片。

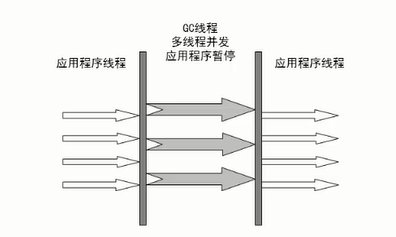
先不说了，先看看JVM的垃圾回收器吧，**先看一种最古老的收集器——串行收集器**

最古老，最稳定，效率高，但是串行的最大问题就是停顿时间很长！**因为串行收集器只使用一个线程去回收，可能会产生较长的停顿现象。**我们可以使用参数-XX:+UseSerialGC，设置新生代、老年代使用串行回收，此时新生代使用复制算法，老年代使用标记-压缩算法（标记-压缩算法首先需要从根节点开始，对所有可达对象做一次标记。但之后，它并不简单的清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。有效解决内存碎片问题）。



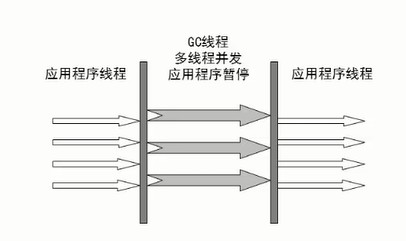
**还有一种收集器叫并行收集器（两种并行收集器）**

1. **一种是ParNew并行收集器**。使用JVM参数设置XX:+UseParNewGC，设置之后，那么新生代就是并行回收，而老年代依然是串行回收，也就是并行回收器不会影响老年代，它是Serial收集器在新生代的并行版本，新生代并行依然使用复制算法，但是是多线程，需要多核支持，我们可以使用JVM参数： XX:ParallelGCThreads 去限制线程的数量。如图：



注意：新生代的多线程回收不一定快！看在多核还是单核，和具体环境。、

1. **还有一种是Parallel收集器**，它类似ParNew，**但是更加关注JVM的吞吐量**！同样是在新生代复制算法，老年代使用标记压缩算法，可以使用JVM参数XX:+UseParallelGC设置使用Parallel并行收集器+ 老年代串行，或者使用XX:+UseParallelOldGC，使用Parallel并行收集器+ 并行老年代。也就是说，Parallel收集器可以同时让新生代和老年代都并行收集。如图：



关于并行收集器还有两个参数设置：

　　-XX:MaxGCPauseMills，代表最大的GC线程占用的停顿时间，单位是毫秒，GC尽力保证回收时间不超过设定值，不是100%的保证。

**-XX:GCTimeRatio，GC使用的cpu时间占总时间的百分比，理解为吞吐量**，0-100的取值范围，垃圾收集时间占总时间的比，默认99，即最大允许1%时间做GC。我们肯定希望停顿时间短，且占用总时间比例少，但是这两个参数是矛盾的。因为停顿时间和吞吐量不可能同时调优。

如果GC很频繁，那么GC的最大停顿时间变短，但吞吐量变小，如果GC次数很少，最大的停顿时间就会变长，但吞吐量增大。

**最后看一个很重要的收集器-CMS（并发标记清除收集器Concurrent Mark Sweep）收集器**

　　顾名思义，它在老年代使用的是标记清除算法，而不是标记压缩算法，也就是说CMS是老年代收集器（新生代使用ParNew：新生代并行回收），所谓并发标记清除就是CMS与用户线程一起执行。标记-清除算法与标记-压缩相比，并发阶段会降低吞吐量，使用参数-XX:+UseConcMarkSweepGC打开。

 　　CMS运行过程比较复杂，着重实现了标记的过程，可分为：

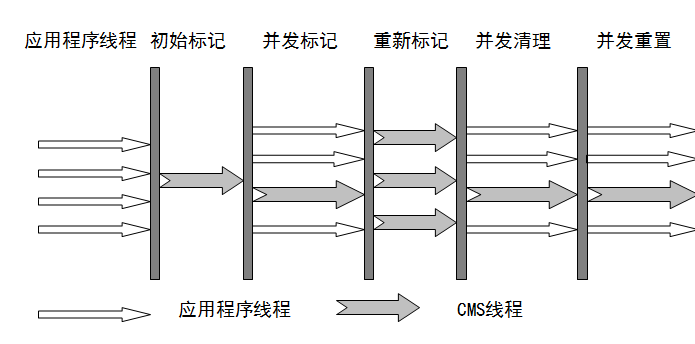
**初始标记**，标记GC ROOT 根可以直接关联到的对象（会产生全局停顿），但是初始标记速度快。

**并发标记**（和用户线程一起），主要的标记过程，标记了系统的全部的对象（不论垃圾不垃圾）。

**重新标记**，由于并发标记时，用户线程依然运行（可能产生新的对象），因此在正式清理前，再做一次修正，会产生全局停顿。

**并发清除**（和用户线程一起），基于标记结果，直接清理对象。这也是为什么使用标记清除算法的原因，因为清理对象的时候用户线程还能执行！标记压缩算法的压缩过程涉及到内存块移动，这样会有冲突。

**并发重置**，为下一次GC做准备工作。



**CMS的特点**

**尽可能降低了JVM的停顿时间，但是会影响系统整体吞吐量和性能**，比如：

1. 在用户线程运行过程中，分一半CPU去做GC，系统性能在GC阶段，反应速度就下降一半。

2. 清理不彻底。因为在清理阶段，用户线程还在运行，会产生新的垃圾，无法清理。

3. 因为和用户线程基本上是一起运行的，故不能在空间快满时再清理。

可以使用-XX:CMSInitiatingOccupancyFraction设置触发CMS GC的阈值，设置空间内存占用到多少时，去触发GC，如果不幸内存预留空间不够，就会引起concurrent mode failure。

可以使用-XX:+ UseCMSCompactAtFullCollection， Full GC后，进行一次整理，而整理过程是独占的，会引起停顿时间变长。

可以使用-XX:+CMSFullGCsBeforeCompaction，设置进行几次Full GC后，进行一次碎片整理。

还可以使用-XX:ParallelCMSThreads，设定CMS的线程数量，一般设置为cpu数量，不用太大。

为减轻GC压力，我们需要注意些什么？

 　　从三个方面考虑：

1. 软件如何设计架构

2. 代码如何写

3. 堆空间如何分配

**一、内存模型**

**Question：在并发编程中，多个线程之间采取什么机制进行通信（信息交换），什么机制进行数据的同步？**

**Answer：在Java语言中，采用的是内存共享模型来实现多线程之间的信息交换和数据同步的。**

线程之间通过共享程序公共的状态，通过读-写内存中公共状态的方式来进行隐式的通信。同步指的是程序在控制多个线程之间执行程序的相对顺序的机制，在共享内存模型中，同步是显式的，程序员必须显式指定某个方法/代码块需要在多线程之间互斥执行。

Java内存模型，往往是指Java程序在运行时内存的模型，而Java代码是运行在Java虚拟机之上的，由Java虚拟机通过解释执行(解释器)或编译执行(即时编译器)来完成，故Java内存模型，也就是指Java虚拟机的运行时内存模型。

作为Java开发人员来说，并不需要像C/C++开发人员，需要时刻注意内存的释放，而是全权交给虚拟机去管理，那么有就必要了解虚拟机的运行时内存是如何构成的。**运行时内存模型，分为线程私有和共享数据区两大类，其中线程私有的数据区包含程序计数器、虚拟机栈、本地方法区，所有线程共享的数据区包含Java堆、方法区，在方法区内有一个常量池。**



jvm\_memory\_1

（1）线程私有区：

程序计数器，记录正在执行的虚拟机字节码的地址；

虚拟机栈：方法执行的内存区，每个方法执行时会在虚拟机栈中创建栈帧；

本地方法栈：虚拟机的Native方法执行的内存区；

（2）线程共享区：

Java堆：对象分配内存的区域；

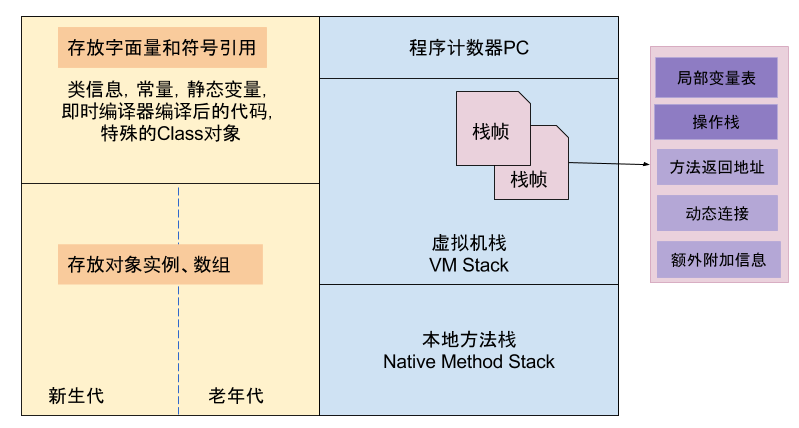
方法区：存放类信息、常量、静态变量、编译器编译后的代码等数据；

常量池：存放编译器生成的各种字面量和符号引用，是方法区的一部分。

对于大多数的程序员来说，Java内存比较流行的说法便是堆和栈，这其实是非常粗略的一种划分，这种划分的”堆”对应内存模型的Java堆，”栈”是指虚拟机栈，然而Java内存模型远比这更复杂，想深入了解Java的内存，还是有必要明白整个内存模型。

二、 详细模型

运行时内存分为五大块区域（常量池属于方法区，算作一块区域），前面简要介绍了每个区域的功能，那接下来再详细说明每个区域的内容，Java内存总体结构图如下：



stack\_heap\_info

2.1 程序计数器PC

程序计数器PC，当前线程所执行的字节码行号指示器。每个线程都有自己计数器，是私有内存空间，该区域是整个内存中较小的一块。

当线程正在执行一个Java方法时，**PC计数器记录的是正在执行的虚拟机字节码的地址**；当线程正在执行的一个Native方法时，PC计数器则为空（Undefined）。

2.2 虚拟机栈

**虚拟机栈，生命周期与线程相同，是Java方法执行的内存模型**。每个方法(不包含native方法)执行的同时都会创建一个栈帧结构，方法执行过程，对应着虚拟机栈的入栈到出栈的过程。

栈帧(Stack Frame)结构

栈帧是用于支持虚拟机进行方法执行的数据结构，是属性运行时数据区的虚拟机站的栈元素。见上图， 栈帧包括：

**局部变量表** (locals大小，编译期确定)，一组变量存储空间， 容量以slot为最小单位。

**操作栈(**stack大小，编译期确定)，操作栈元素的数据类型必须与字节码指令序列严格匹配

**动态连接**， 指向运行时常量池中该栈帧所属方法的引用，为了 动态连接使用。

前面的解析过程其实是静态解析；

对于运行期转化为直接引用，称为动态解析。

**方法返回地址**

正常退出，执行引擎遇到方法返回的字节码，将返回值传递给调用者

异常退出，遇到Exception,并且方法未捕捉异常，那么不会有任何返回值。

**额外附加信息**，虚拟机规范没有明确规定，由具体虚拟机实现。

因此，一个栈帧的大小不会受到

异常(Exception)

Java虚拟机规范规定该区域有两种异常：

**StackOverFlowError：当线程请求栈深度超出虚拟机栈所允许的深度时抛出**

**OutOfMemoryError：当Java虚拟机动态扩展到无法申请足够内存时抛出**

2.3 本地方法栈

本地方法栈则为虚拟机使用到的Native方法提供内存空间，而前面讲的虚拟机栈式为Java方法提供内存空间。有些虚拟机的实现直接把本地方法栈和虚拟机栈合二为一，比如非常典型的Sun HotSpot虚拟机。

异常(Exception)：Java虚拟机规范规定该区域可抛出StackOverFlowError和OutOfMemoryError。

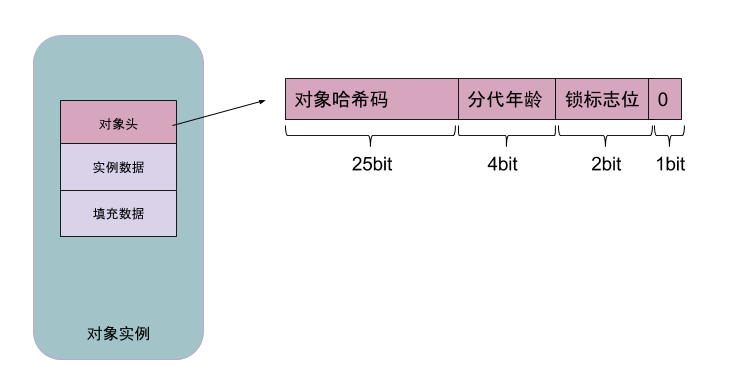
2.4 Java堆

Java堆，是Java虚拟机管理的最大的一块内存，也是GC的主战场，里面存放的是几乎所有的对象实例和数组数据。JIT编译器有栈上分配、标量替换等优化技术的实现导致部分对象实例数据不存在Java堆，而是栈内存。

从内存回收角度，Java堆被分为新生代和老年代；这样划分的好处是为了更快的回收内存；

从内存分配角度，Java堆可以划分出线程私有的分配缓冲区(Thread Local Allocation Buffer,TLAB)；这样划分的好处是为了更快的分配内存；

对象创建的过程是在堆上分配着实例对象，那么对象实例的具体结构如下：



java\_object

对于填充数据不是一定存在的，仅仅是为了字节对齐。HotSpot VM的自动内存管理要求对象起始地址必须是8字节的整数倍。对象头本身是8的倍数，当对象的实例数据不是8的倍数，便需要填充数据来保证8字节的对齐。该功能类似于高速缓存行的对齐。

另外，关于在堆上内存分配是并发进行的，虚拟机采用CAS加失败重试保证原子操作，或者是采用每个线程预先分配TLAB内存.

异常(Exception)：Java虚拟机规范规定该区域可抛出OutOfMemoryError。

2.5 方法区

方法区主要存放的是已被虚拟机加载的类信息、常量、静态变量、编译器编译后的代码等数据。GC在该区域出现的比较少。

异常(Exception)：Java虚拟机规范规定该区域可抛出OutOfMemoryError。

2.6 运行时常量池

运行时常量池也是方法区的一部分，用于存放编译器生成的**各种字面量和符号引用**。运行时常量池除了编译期产生的Class文件的常量池，还可以在运行期间，将新的常量加入常量池，比较常见的是String类的intern()方法。

字面量：与Java语言层面的常量概念相近，包含文本字符串、声明为final的常量值等。

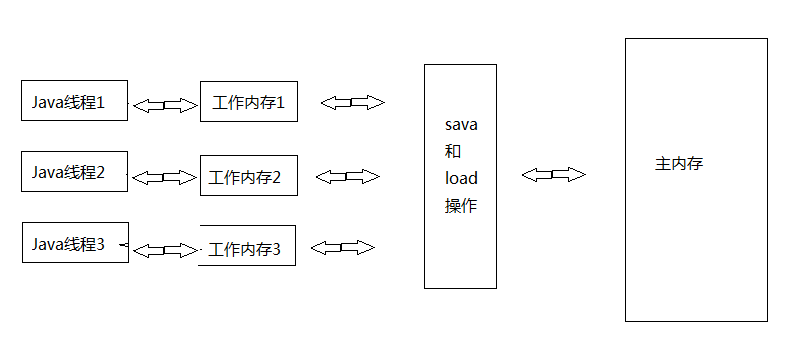
符号引用：编译语言层面的概念，包括以下3类：

类和接口的全限定名

字段的名称和描述符

方法的名称和描述符

但是该区域不会抛出OutOfMemoryError异常。

**JMM（java内存模型）规定了所有的变量都存储在主内存（Main Memory）中。每个线程还有自己的工作内存（Working Memory）**,线程的工作内存中保存了该线程使用到的变量的主内存的副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量（volatile变量仍然有工作内存的拷贝，但是由于它特殊的操作顺序性规定，所以看起来如同直接在主内存中读写访问一般）。不同的线程之间也无法直接访问对方工作内存中的变量，线程之间值的传递都需要通过主内存来完成。****JMM规定了所有的变量都存储在主内存（Main Memory）中。每个线程还有自己的工作内存（Working Memory）,线程的工作内存中保存了该线程使用到的变量的主内存的副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量（volatile变量仍然有工作内存的拷贝，但是由于它特殊的操作顺序性规定，所以看起来如同直接在主内存中读写访问一般）。不同的线程之间也无法直接访问对方工作内存中的变量，线程之间值的传递都需要通过主内存来完成。****  


线程1和线程2要想进行数据的交换一般要经历下面的步骤：  
1.线程1把工作内存1中的更新过的共享变量刷新到主内存中去。  
2.线程2到主内存中去读取线程1刷新过的共享变量，然后copy一份到工作内存2中去。

**内存模型的特性**

**Java内存模型是围绕着并发编程中原子性、可见性、有序性**这三个特征来建立的，那我们依次看一下这三个特征：

**原子性（Atomicity）**

**原子性是指一个操作不能被打断，要么全部执行完毕，要么不执行**。在这点上有点类似于事务操作，要么全部执行成功，要么回退到执行该操作之前的状态。

基本类型数据的访问大都是原子操作，long 和double类型的变量是64位，但是在32位JVM中，32位的JVM会将64位数据的读写操作分为2次32位的读写操作来进行，这就导致了long、double类型的变量在32位虚拟机中是非原子操作，数据有可能会被破坏，也就意味着多个线程在并发访问的时候是线程非安全的。

**可见性**

一个线程对共享变量做了修改之后，其他的线程立即能够看到（感知到）该变量这种修改（变化）。

Java内存模型是通过将在工作内存中的变量修改后的值同步到主内存，在读取变量前从主内存刷新最新值到工作内存中，这种依赖主内存的方式来实现可见性的。

**无论是普通变量还是volatile变量都是如此，区别在于：volatile的特殊规则保证了volatile变量值修改后的新值立刻同步到主内存，每次使用volatile变量前立即从主内存中刷新，因此volatile保证了多线程之间的操作变量的可见性，而普通变量则不能保证这一点。**

除了volatile关键字能实现可见性之外，还有synchronized,Lock，final也是可以的。

使用synchronized关键字，在同步方法/同步块开始时（Monitor Enter）,使用共享变量时会从主内存中刷新变量值到工作内存中（即从主内存中读取最新值到线程私有的工作内存中），在同步方法/同步块结束时(Monitor Exit),会将工作内存中的变量值同步到主内存中去（即将线

程私有的工作内存中的值写入到主内存进行同步）。

**使用Lock接口的最常用的实现ReentrantLock(重入锁)来实现可见性**：当我们在方法的开始位置执行lock.lock()方法，这和synchronized开始位置（Monitor Enter）有相同的语义，即使用共享变量时会从主内存中刷新变量值到工作内存中（即从主内存中读取最新值到线程私有的工作内存中），在方法的最后finally块里执行lock.unlock()方法，和synchronized结束位置（Monitor Exit）有相同的语义,即会将工作内存中的变量值同步到主内存中去（即将线程私有的工作内存中的值写入到主内存进行同步）。

**final关键字的可见性**是指：被final修饰的变量，在构造函数数一旦初始化完成，并且在构造函数中并没有把“this”的引用传递出去（“this”引用逃逸是很危险的，其他的线程很可能通过该引用访问到只“初始化一半”的对象），那么其他线程就可以看到final变量的值。

**有序性**

对于一个线程的代码而言，我们总是以为代码的执行是从前往后的，依次执行的。这么说不能说完全不对，在单线程程序里，确实会这样执行；但是在多线程并发时，程序的执行就有可能出现乱序。用一句话可以总结为：在本线程内观察，操作都是有序的；如果在一个线程中观察另外一个线程，所有的操作都是无序的。前半句是指“线程内表现为串行语义（WithIn Thread As-if-Serial Semantics）”,后半句是指“指令重排”现象和“工作内存和主内存同步延迟”现象。

**Java提供了两个关键字volatile和synchronized来保证多线程之间操作的有序性**,**volatile关键字本身通过加入内存屏障来禁止指令的重排序**，**而synchronized关键字通过一个变量在同一时间只允许有一个线程对其进行加锁的规则来实现**，在单线程程序中，不会发生“指令重排”和“工作内存和主内存同步延迟”现象，只在多线程程序中出现。

**happens-before原则**

**Java内存模型中定义的两项操作之间的次序关系**，如果说操作A先行发生于操作B，操作A产生的影响能被操作B观察到，“影响”包含了修改了内存中共享变量的值、发送了消息、调用了方法等。

下面是Java内存模型下一些”天然的“happens-before关系，这些happens-before关系无须任何同步器协助就已经存在，可以在编码中直接使用。如果两个操作之间的关系不在此列，并且无法从下列规则推导出来的话，它们就没有顺序性保障，虚拟机可以对它们进行随意地重排序。

1. 程序次序规则(Pragram Order Rule)：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、循环结构。

2. 管程锁定规则(Monitor Lock Rule)：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。

3. volatile变量规则(Volatile Variable Rule)：对一个volatile变量的写操作先行发生于后面对这个变量的读取操作，这里的”后面“同样指时间上的先后顺序。

4. 线程启动规则(Thread Start Rule)：Thread对象的start()方法先行发生于此线程的每一个动作。

5. 线程终于规则(Thread Termination Rule)：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()的返回值等作段检测到线程已经终止执行。

6. 线程中断规则(Thread Interruption Rule)：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()方法检测是否有中断发生。

7. 对象终结规则(Finalizer Rule)：一个对象初始化完成(构造方法执行完成)先行发生于它的finalize()方法的开始。

8. 传递性(Transitivity)：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。

一个操作”时间上的先发生“不代表这个操作会是”先行发生“，那如果一个操作”先行发生“是否就能推导出这个操作必定是”时间上的先发生“呢？也是不成立的，一个典型的例子就是指令重排序。所以时间上的先后顺序与happens-before原则之间基本没有什么关系，所以衡量并发安全问题一切必须以happens-before 原则为准。

## 类加载过程

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using)和卸载(Unloading)7个阶段。其中准备、验证、解析3个部分统称为连接（Linking）。如图所示。  
  
加载、验证、准备、初始化和卸载这5个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地开始，而解析阶段则不一定：它在某些情况下可以在初始化阶段之后再开始，这是为了支持Java语言的运行时绑定（也称为动态绑定或晚期绑定）。以下陈述的内容都已HotSpot为基准。

### **加载**

在加载阶段（可以参考java.lang.ClassLoader的loadClass()方法），虚拟机需要完成以下3件事情：

1. 通过一个类的全限定名来获取定义此类的二进制字节流（并没有指明要从一个Class文件中获取，可以从其他渠道，譬如：网络、动态生成、数据库等）；
2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构；
3. 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口；

加载阶段和连接阶段（Linking）的部分内容（如一部分字节码文件格式验证动作）是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始，但这些夹在加载阶段之中进行的动作，仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

### 验证

验证是连接阶段的第一步，这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。  
验证阶段大致会完成4个阶段的检验动作：

1. **文件格式验证**：验证字节流是否符合Class文件格式的规范；例如：是否以魔术0xCAFEBABE开头、主次版本号是否在当前虚拟机的处理范围之内、常量池中的常量是否有不被支持的类型。
2. **元数据验证**：对字节码描述的信息进行**语义分析**（注意：对比javac编译阶段的语义分析），以保证其描述的信息符合Java语言规范的要求；例如：这个类是否有父类，除了java.lang.Object之外。
3. **字节码验证**：通过数据流和控制流分析，**确定程序语义是合法的、符合逻辑的**。
4. **符号引用验证**：确保解析动作能正确执行。

验证阶段是非常重要的，但不是必须的，它对程序运行期没有影响，如果所引用的类经过反复验证，那么可以考虑采用-Xverifynone参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

### **准备**

**准备阶段是正式为类变量分配内存并设置类变量初始值的阶段**，这些变量所使用的内存都将在方法区中进行分配。这时候进行内存分配的仅包括类变量（被static修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在堆中。其次，这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量的定义为：

|  |  |
| --- | --- |
| 1 | public static int value=123; |

那变量value在准备阶段过后的初始值为0而不是123.因为这时候尚未开始执行任何java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器()方法之中，所以把value赋值为123的动作将在初始化阶段才会执行。  
至于“特殊情况”是指：public static final int value=123，即当类字段的字段属性是ConstantValue时，会在准备阶段初始化为指定的值，所以标注为final之后，value的值在准备阶段初始化为123而非0.

### **解析**

**解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程**。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

### **初始化**

类初始化阶段是类加载过程的最后一步，**到了初始化阶段，才真正开始执行类中定义的java程序代码**。在准备极端，变量已经付过一次系统要求的初始值，而在初始化阶段，则根据程序猿通过程序制定的主管计划去初始化类变量和其他资源，或者说：**初始化阶段是执行类构造器<clinit>()方法的过程.  
<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块static{}中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但是不能访问。**如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class Test  {      static      {          i=0;          System.out.println(i);//这句编译器会报错：Cannot reference a field before it is defined（非法向前应用）      }      static int i=1;  } |

<**clinit>()方法与实例构造器<init>()方法不同，它不需要显示地调用父类构造器，虚拟机会保证在子类<init>()方法执行之前，父类的<clinit>()方法方法已经执行完毕**，回到本文开篇的举例代码中，结果会打印输出：SSClass就是这个道理。  
由于父类的<clinit>()方法先执行，也就意味着父类中定义的静态语句块要优先于子类的变量赋值操作。  
<clinit>()方法对于类或者接口来说并不是必需的，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生产<clinit>()方法。  
接口中不能使用静态语句块，但仍然有变量初始化的赋值操作，因此接口与类一样都会生成<clinit>()方法。**但接口与类不同的是，执行接口的<clinit>()方法不需要先执行父接口的<clinit>()方法。只有当父接口中定义的变量使用时，父接口才会初始化。另外，接口的实现类在初始化时也一样不会执行接口的<clinit>()方法。  
虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确的加锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。**如果在一个类的<clinit>()方法中有好事很长的操作，就可能造成多个线程阻塞，在实际应用中这种阻塞往往是隐藏的。（静态内部类实现单例模式）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | package jvm.classload;    public class DealLoopTest  {      static class DeadLoopClass      {          static          {              if(true)              {                  System.out.println(Thread.currentThread()+"init DeadLoopClass");                  while(true)                  {                  }              }          }      }        public static void main(String[] args)      {          Runnable script = new Runnable(){              public void run()              {                  System.out.println(Thread.currentThread()+" start");                  DeadLoopClass dlc = new DeadLoopClass();                  System.out.println(Thread.currentThread()+" run over");              }          };            Thread thread1 = new Thread(script);          Thread thread2 = new Thread(script);          thread1.start();          thread2.start();      }  } |

运行结果：（即一条线程在死循环以模拟长时间操作，另一条线程在阻塞等待）

|  |  |
| --- | --- |
| 1  2  3 | Thread[Thread-0,5,main] start  Thread[Thread-1,5,main] start  Thread[Thread-0,5,main]init DeadLoopClass |

需要注意的是，其他线程虽然会被阻塞，但如果执行<clinit>()方法的那条线程退出<clinit>()方法后，其他线程唤醒之后不会再次进入<clinit>()方法。同一个类加载器下，一个类型只会初始化一次。  
将上面代码中的静态块替换如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | static          {              System.out.println(Thread.currentThread() + "init DeadLoopClass");              try              {                  TimeUnit.SECONDS.sleep(10);              }              catch (InterruptedException e)              {                  e.printStackTrace();              }          } |

运行结果：

|  |  |
| --- | --- |
| 1  2  3  4  5 | Thread[Thread-0,5,main] start  Thread[Thread-1,5,main] start  Thread[Thread-1,5,main]init DeadLoopClass (之后sleep 10s)  Thread[Thread-1,5,main] run over  Thread[Thread-0,5,main] run over |

虚拟机规范严格规定了有且只有5中情况（jdk1.7）必须对类进行“初始化”（而加载、验证、准备自然需要在此之前开始）：

1. 遇到new,getstatic,putstatic,invokestatic这失调字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候、读取或设置一个类的静态字段（被final修饰、已在编译器把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。
2. 使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。
3. 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
4. 当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。
5. 当使用jdk1.7动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF\_getstatic,REF\_putstatic,REF\_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先出触发其初始化。

开篇已经举了一个范例：通过子类引用父类的静态字段，不会导致子类初始化。  
这里再举两个例子。  
1. 通过数组定义来引用类，不会触发此类的初始化：（SuperClass类已在本文开篇定义）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class NotInitialization  {      public static void main(String[] args)      {          SuperClass[] sca = new SuperClass[10];      }  } |

运行结果：（无）  
2. 常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class ConstClass  {      static      {          System.out.println("ConstClass init!");      }      public static  final String HELLOWORLD = "hello world";  }  public class NotInitialization  {      public static void main(String[] args)      {          System.out.println(ConstClass.HELLOWORLD);      }  } |

运行结果：hello world

附：昨天从论坛上看到一个例子，很有意思，如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | package jvm.classload;    public class StaticTest  {      public static void main(String[] args)      {          staticFunction();      }        static StaticTest st = new StaticTest();        static      {          System.out.println("1");      }        {          System.out.println("2");      }        StaticTest()      {          System.out.println("3");          System.out.println("a="+a+",b="+b);      }        public static void staticFunction(){          System.out.println("4");      }        int a=110;      static int b =112;  } |

问题是：请问输出是什么？  
程序跑一下就知道结果，如果想知道为什么，请在下方留言~~

结果：

2

3

A=110,b=0

1

4

类的初始化阶段需要做是执行类构造器（类构造器是编译器收集所有静态语句块和类变量的赋值语句按语句在源码中的顺序合并生成类构造器，对象的构造方法是<init>()，类的构造方法是<clinit>()，可以在堆栈信息中看到），因此先执行第一条静态变量的赋值语句即st = new StaticTest ()，此时会进行对象的初始化，对象的初始化是先初始化成员变量再执行构造方法，因此设置a为110->打印2->执行构造方法(打印3,此时a已经赋值为110，但是b只是设置了默认值0，并未完成赋值动作)，等对象的初始化完成后继续执行之前的类构造器的语句，接下来就不详细说了，按照语句在源码中的顺序执行即可。

**垃圾回收：**

**在JVM分代垃圾回收机制中，将应用程序可用的堆空间分为年轻代（young generation）和年老代（old generation），又将年轻代分为Eden区（Eden Space）、From区和To区，新建对象总是在Eden区中被创建，当Eden区已满，就触发一次Young GC，将还被使用的对象复制到From区，这样整个Eden区都是未被使用的空间，可供继续创建对象，当Eden区再次用完，再触发一次Young GC，将Eden区和From区还在被使用的对象复制到To区，下一次Young GC则是将Eden区和To区还被使用的对象复制到From区。因此经过多次Young GC，某些对象会在From区和To区多次复制，如果超过某个阈值对象还未被释放，则将该对象复制到Old Generation。如果Old Generation空间也已用完，那么就会触发Full GC，即所谓的全量回收，全量回收会对系统性能产生较大影响，因此应根据系统业务特点和对象生命周期，合理设置年轻代和年老代大小，尽量减少Full GC。事实上，某些Web应用在整个运行期间可以做到从不进行Full GC。**

**类加载机制类加载机制**

启动类加载器( Bootstrap ClassLoader)启动类加载器无法被 java 程序员直接引用, 这个类加载器负责把存放在<JAVA\_HOME>\lib目录中的, 或者被-Xbootclasspath参数指定路径中的, 并且是被虚拟机识别的类库加载到虚拟机内存中.

扩展类加载器(Extension ClassLoader)负责加载在<JAVA\_HOME>\lib\ext目录中的, 或者被java.ext.dirs系统变量所指定的路径中的所有类库。

应用程序类加载器( Application ClassLoader )这个类加载器是ClassLoader 中的 getSystemClassLoader()方法的返回值, 一般称其为系统类加载器, 它负责加载用户类路径( ClassPath )上所指定的类库

从 java 虚拟机的角度而降, 只存在两种不同的类加载器：

一个是启动类加载器( Bootstrap ClassLoader ), 这个类加载使用 C++ 语言实现, 是虚拟机自身的一部分;

另一种是其他所有的类加载器, 他们由 java 语言实现, 独立于虚拟机之外, 并且全部继承自java.lang.ClassLoader

加载类的寻找范围就是 JVM 默认路径加上Classpath, 类具体是使用哪个类加载器不确定。

这两个名词都是并发编程中的概念，在谈论垃圾收集器的上下文语境中，它们可以解释如下。

并行（Parallel）：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态（系统多核支持）。

并发（Concurrent）：指用户线程与垃圾收集线程同时执行（但不一定是并行的，可能会交替执行），用户程序在继续运行，而垃圾收集程序运行于另一个CPU上。****Minor GC 和 Full GC****

新生代GC（Minor GC）：指发生在新生代的垃圾收集动作，因为**[Java](http://lib.csdn.net/base/javase" \o "Java SE知识库" \t "http://blog.csdn.net/u011080472/article/details/_blank)**对象大多都具备朝生夕灭的特性，所以Minor GC非常频繁，一般回收速度也比较快。

老年代GC（Major GC / Full GC）：指发生在老年代的GC，出现了Major GC，经常会伴随至少一次的Minor GC（但非绝对的，在Parallel Scavenge收集器的收集策略里就有直接进行Major GC的策略选择过程）。Major GC的速度一般会比Minor GC慢10倍以上。****吞吐量****

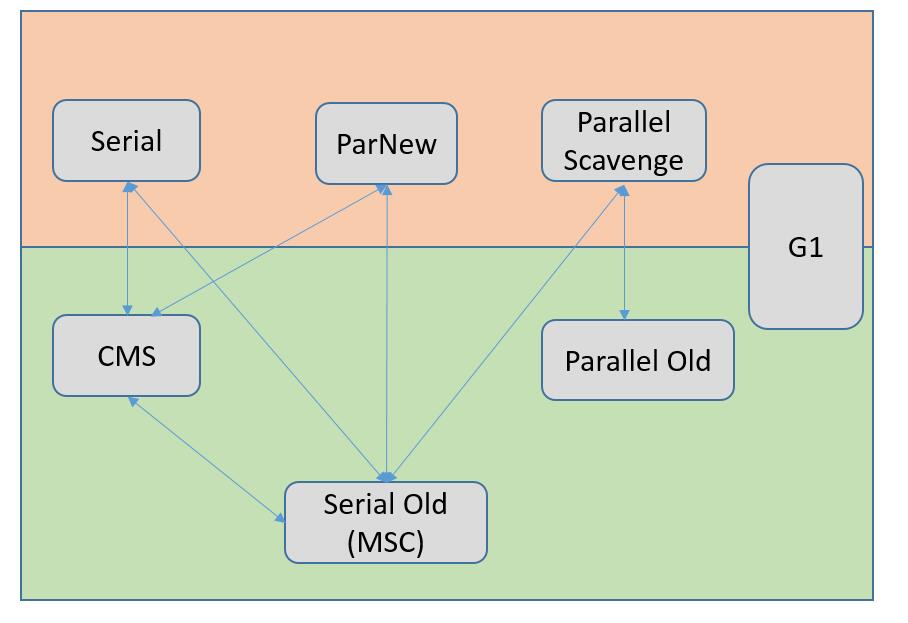
吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即

吞吐量 = 运行用户代码时间 /（运行用户代码时间 + 垃圾收集时间）。

虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。

# ****HotSpot垃圾收集器****

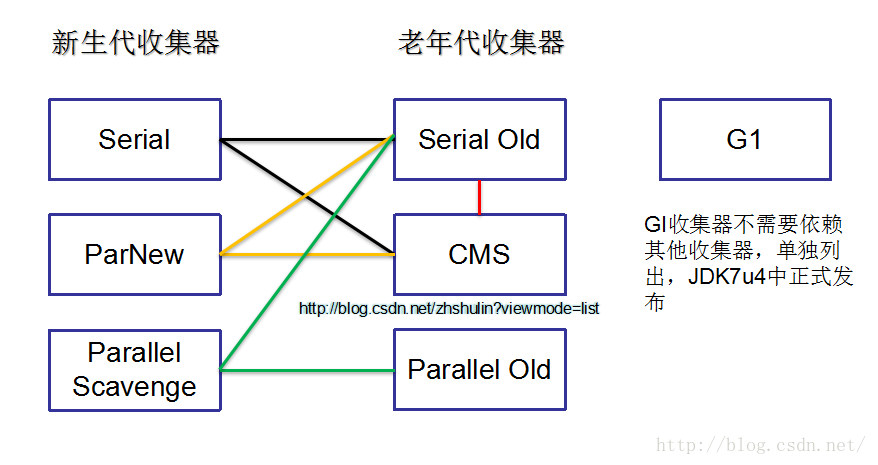
在虚拟机规范中并没有对垃圾回收器如何实现具体介绍，因此每个厂商的垃圾回收器可能会完全不同，但是我们介绍的是基于JDK1.7之后的Hotspot虚拟机（包括前面对Java虚拟机的介绍也是基于jdk1.7版本的）。在Hotspot中，虚拟机的收集器主要有下：



可以看到垃圾收集器是按对象的分代来划分的，可以用双箭头连接的垃圾收集器表示两者可以配合使用。可以看到新生代垃圾收集器有Serial、ParNew、Parallel Scavenge，G1，属于老年代的垃圾收集器有CMS、Serial Old、Parallel Old和G1.其中的G1是一种既可以对新生代对象也可以对老年代对象进行回收的垃圾收集器。然而，在所有的垃圾收集器中，并没有一种普遍使用的垃圾收集器。在不同的场景下，每种垃圾收集器有各自的优势。

通过学习了解到现在商用的JVM中的垃圾收集采用的是分代收集**[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "http://blog.csdn.net/zhshulin/article/details/_blank)**，即针对不同年代采用不同的收集算法。在JVM中，GC主要作用于堆内存中，堆内存又被划分为新生代和老年代，由于新生代对象绝大多数是朝生夕死，而老年代相对存活时间就很长，故而需要使用不同的垃圾收集机制，所以垃圾收集器也就分为新生代收集器和老年代收集器，两者相互组合进行JVM堆内存的空间回收（下图中相连的垃圾收集器表示可以相互组合，注意Serial Old和CMS也可以联合进行老年代的垃圾收集）。JDK6u14中开始**[测试](http://lib.csdn.net/base/softwaretest" \o "软件测试知识库" \t "http://blog.csdn.net/zhshulin/article/details/_blank)**的G1垃圾收集器，正式发布于JDK7u4中，是目前唯一不需要依赖其他垃圾收集器即可完成新生代和老年代内存收集。阅读之前先了解，GC的两个指标：暂停时间-应对与存在大量用户交互的场景；吞吐量-应对后台计算任务。

* 新生代的垃圾收集器有：Serial收集器、ParNew收集器、Parallel Scavenge收集器
* 老年代的垃圾收集器有：Serial Old收集器、Parallel Old收集器、CMS收集器
* G1收集器。[http://f.dataguru.cn/thread-514678-1-1.html](http://f.dataguru.cn/thread-514678-1-1.html" \t "http://blog.csdn.net/zhshulin/article/details/_blank)



笔者使用的是JDK7u51，也就是JDK1.7.0\_51

    下面我将试着通过自己的理解来分析各个垃圾收集器的特点，目前并没有一个适用于任何场景的垃圾收集器，所以选择何种垃圾收集器进行配合是根据具体应用来区别对待的，那么了解各种垃圾收集器的特点以及他们之间是否可以相互配合，就十分重要了。

    垃圾收集器运行过程中必然会发生“Stop the world”，只是时间长短和暂停时间可不可控的区别。

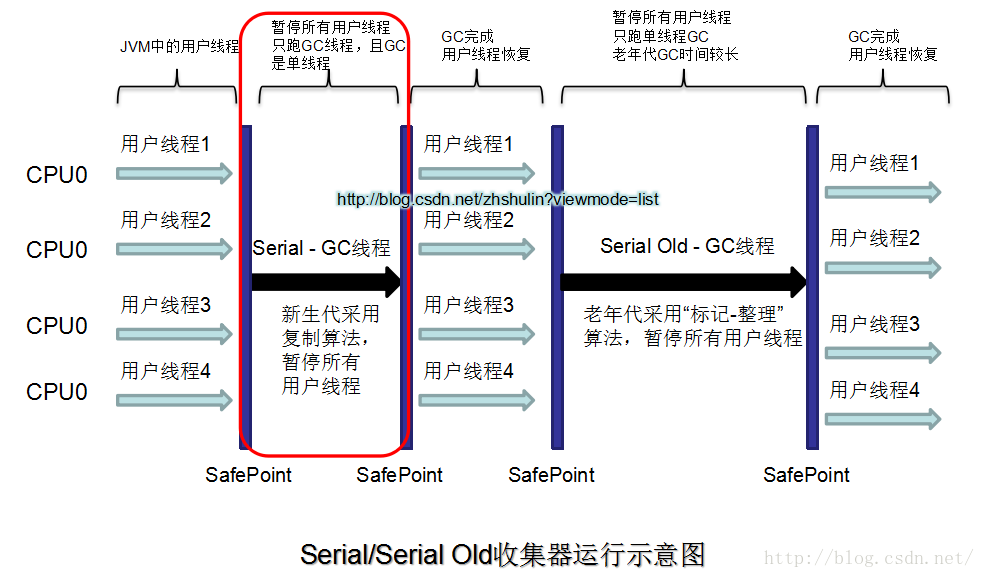
    在进行下面的阅读之前，首先明确在垃圾收集器中，“并发”和“并行”这两个概念的差别：

1. 并行（Parallel）：多个垃圾收集线程并行工作，此时用户线程处于等待状态
2. 并发（Concurrent）：垃圾收集线程和用户线程同时执行（不一定是并行，可能是交替执行），用户程序继续执行，而GC运行在另一个CPU上

* Serial

       Serial垃圾收集器，通过这个单词的意思“连续”，我认为这个应该是指的GC之后内存空间不存在内存碎片的意思，那么必然不会采用“标记-清除算法”来实现，所以这个垃圾收集器在新生代使用的是“复制算法”，而Serial Old作为Serail收集器的老年代版本，使用的就是“标记-整理算法”。

    为什么先说Serial垃圾收集器，是因为这个收集器是最基本、历史最悠久的收集器，在JDK1.3.1之前，是JVM新生代收集的唯一选择。Serial收集器是一个单线程的收集器，这个“单线程”是指JVM在使用它进行GC的时候，必须暂停其他所有的工作线程（sun将这件事情称为“Stop the world”），直到GC完成，这是一件非常可怕的事情。看到这里，你可能想我一定要修改我的JVM的新生代收集器，不用Serial了，但是直至现在，Serial依然是JVM在运行Client模式下默认的新生代 收集器。与其他垃圾收集器的单线程相比，Serial简单而高效。对于用户桌面应用场景来说，分配给JVM的内存一般不会太大，收集十几甚至一两百兆的内存，停顿时间可以控制在几十毫秒，最多一百多毫秒以内，只要不是特别频繁，这些停顿还是可以接受的。所以，对于Client模式下的JVM来说，Serial是个很好的新生代收集器，简单高效。

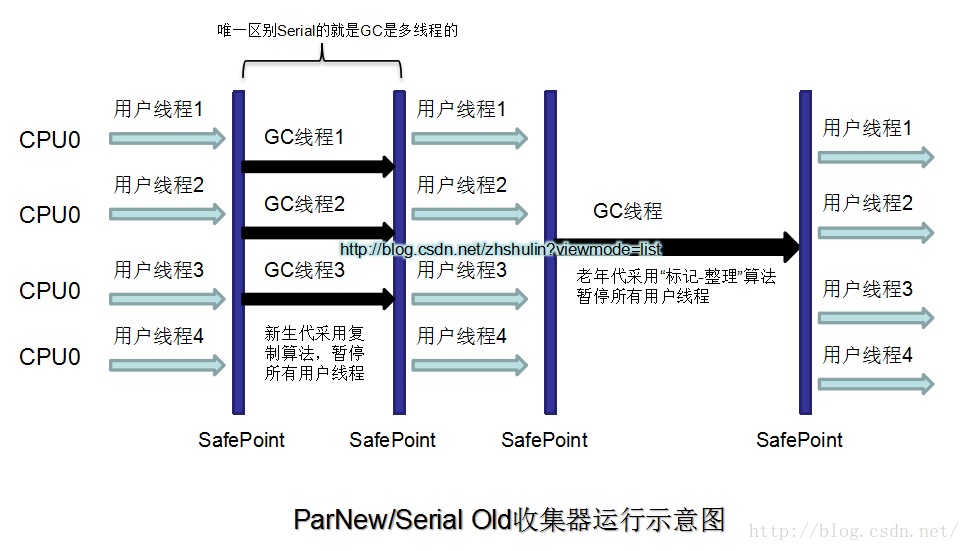


* ParNew-Parallel New

       ParNew收集器也是一个新生代收集器，其实就是Serial收集器的多线程版本，是一个“并行”的垃圾收集器，除了多线程外，其他和Serial差不多。想想也就明白了，当JVM团队开发出来了Serial，可以满足Client模式下的JVM，但是对于Server模式下的JVM来说，运行很长时间，有很多的对象需要收集（可能几十个G），单线程导致的停顿时间太长了（比如每运行1小时需要停顿5分钟），用户无法接受业务线程停顿那么长的时间，我猜测这种情况下那些大牛能想到的最简单的办法就是让Serial变成多线程，这样开多个线程就可以有效的降低停顿时间，故而这个Serial的多线程版本也就诞生了。

    ParNew是许多运行在Server模式下的JVM中首选的垃圾收集器，其中一个重要原因就是除了Serial，它是唯一可以和CMS（Concurrent Mark Sweep）老年代垃圾收集器配合工作。

**ParNew在单CPU环境中收集效果不如Serial收集器，但是随着CPU的增加，它对于GC时系统资源的利用还是很有好处的，默认开启的线程数与CPU的数量一致 。**

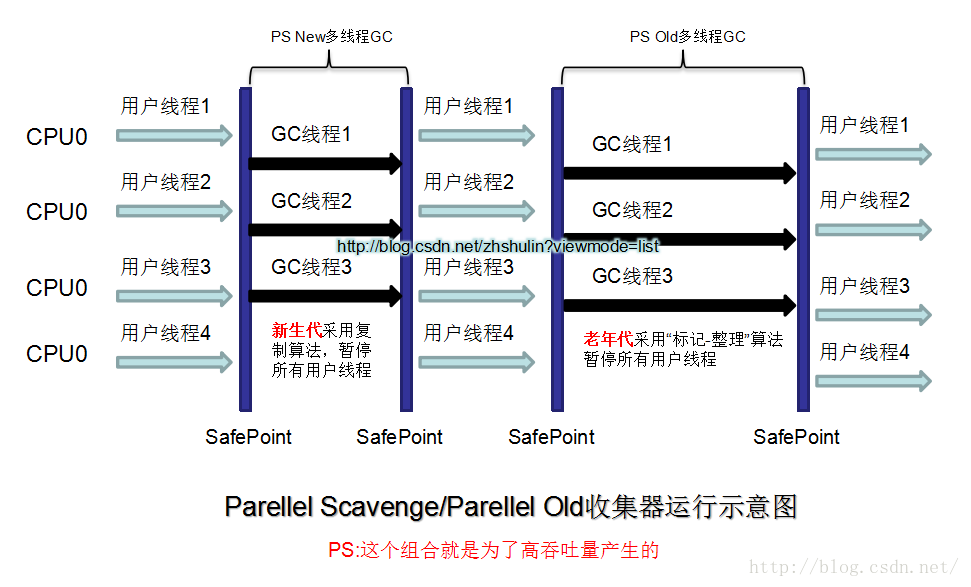


* Parallel Scavenge

    Parallel Scavenge收集器，简称PS收集器，它和ParNew收集器一样是一个多线程的并行新生代垃圾收集器，一样采用“复制算法”（始发于JDK1.4.0）。那为什么还要这个PS收集器呢？现在想一下，ParNew收集器为什么会产生，不就是嫌Serial收集器导致的“Stop the world”的时间太长了嘛，搞个多线程，减少停顿时间。这种的垃圾收集器适合重视服务的响应速度的应用程序（比如购物网站，肯定希望停顿时间越短越好，这样用户体验才好），但是对于一个后台计算任务（比如MapReduce）来说，没有太多的交互任务，那么它所重视的就不是这种响应速度，而是CPU的有效时间利用率（这是我的理解），官方称之为“吞吐量（Throughtput）”。吞吐量就是指CPU用来运行用户代码的时间和CPU的总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+GC消耗的时间）。

    Parallel Scavenge收集器正是基于对“吞吐量”的追求而产生的，它的目标就是达到一个可控的吞吐量。由于与吞吐量关系密切，Parallel Scavenge收集器也被称为“吞吐量优先“收集器。Parallel Scavenge提供了两个参数用来精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX：MaxGCPauseMillis参数（单位：毫秒），以及直接设置吞吐量大小的-XX：GCTimeRatio参数（0-100之间，不包括首尾）。GCTimeRatio参数的计算规则是，比如设成19，那么允许最大时间就占总时间的5%，即1/（1+19），默认值是99，也就是默认允许最大GC时间占比是1%。

    Parallel Scavenge收集器还有一个参数来开启GC的自适应调节策略，只需要将JVM基本内存设置好，并且制定上述两个参数中的一个来作为JVM的优化目标，那么JVM就可以根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大吞吐量，这个参数就是-XX：+UseAdaptiveSizePolicy。**自适应调节策略也是PS收集器 相对于ParNew收集器的一个重要区别。**ParNew收集器需要手工指定新生代大小（-Xmn）、Eden与Survivor的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节参数。



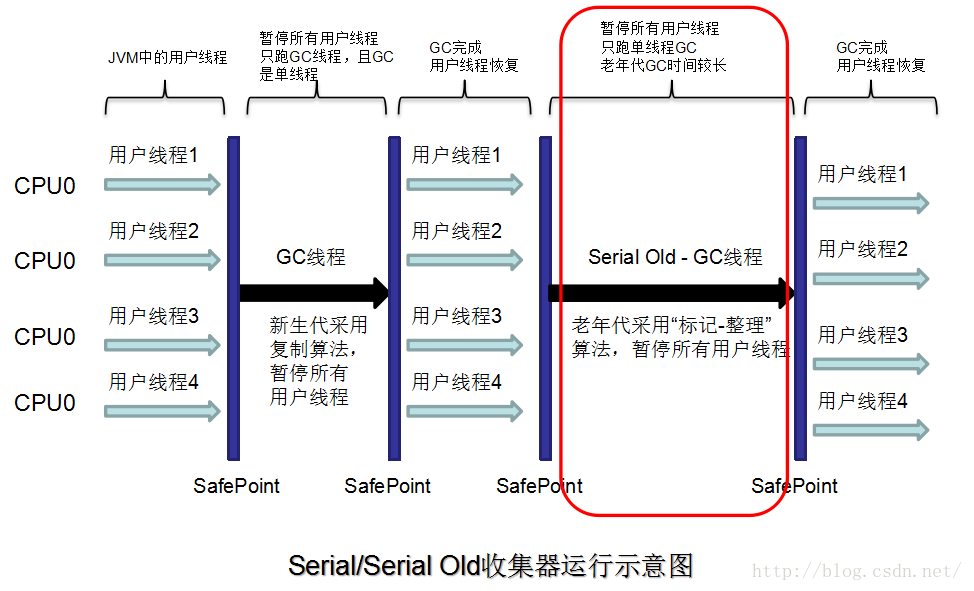
* Serial Old

       Serial收集器的老年代版本，自然也是JVM中最早的老年代垃圾收集器，又称为PS MarkSweep。它与Serial收集器一样是一个单线程收集器，使用”标记-整理算法“。这个收集器的主要一样也是被Client模式下的JVM使用。如果在Server模式下，它主要有个两大用途：

1. 在JDK1.5和之前的版本中与Parallel Scavenge收集器搭配使用，因为此时CMS还没有，CMS正式发布于JDK1.6中
2. 在JDK1.6和以后的版本中，作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure的时候使用。

    现在JDK已经发展到了JDK8了，大量系统使用的稳定版本也是JDK1.6和JDK1.7，所以第一种用途几乎没什么意义了，第二种还在用。

PS：为何在JDK1.4的时候，新生代已经有了Serial、ParNew和Parallel Scavenge这三种收集器，而老年代此时还是只有Serial Old这一种最原始的单线程收集器，个人推测，是由于Young GC发生的频率远远高于Full GC，故而如何有效提高Young GC的收集效率减少停顿时间和增加吞吐量才是JDK1.4及之前的时间里JVM研发团队的首要任务，当对新生代的垃圾收集做到不错的程度的时候（有了并行收集器），工作重心才转移到老年代的垃圾收集上，这就是一个工作的优先级问题，毕竟每个人、每个团队、每个公司都避免不了的只有有限的精力和资源。在JDK1.5的时候推出了新的老年代收集器CMS，JDK1.6的时候推出了Parallel Old。



* CMS

        在面对新生代的时候，垃圾收集器有两种提升的方式，一种是减少用户线程暂停的时间，另一种是提高”吞吐量“。假设现在处于JDK1.4发布到1.5发布之间，你要设计一个老年代的垃圾收集器，当前只有Serial Old一种，那么如果让你来选，你优先提高GC的何种性能呢？前面也分析了”Stop the world“和”吞吐量“性能提升所应对的场景，那么再问你，你觉得优先提高JVM老年代垃圾收集来满足用户对响应速度的需求比较重要，还是优先提高JVM老年代垃圾收集来满足后台的计算任务呢？JDK1.4于2002年2月发布，那是JAVA已经比较火了，从96年开始JAVA就被用来制作网页了，那么显然用户对于响应速度的提升需求比较强烈，那么对于JVM项目组来说在已经有了3种新生代垃圾收集器的前提下，研发一款减少用户线程停顿时间的老年代垃圾收集器是必然的，所以在2004年9月，历时两年半，发布了JDK1.5，有了CMS垃圾收集器（Concurrent Mark Sweep）。（以上均为个人推测，与事实不符概不负责）

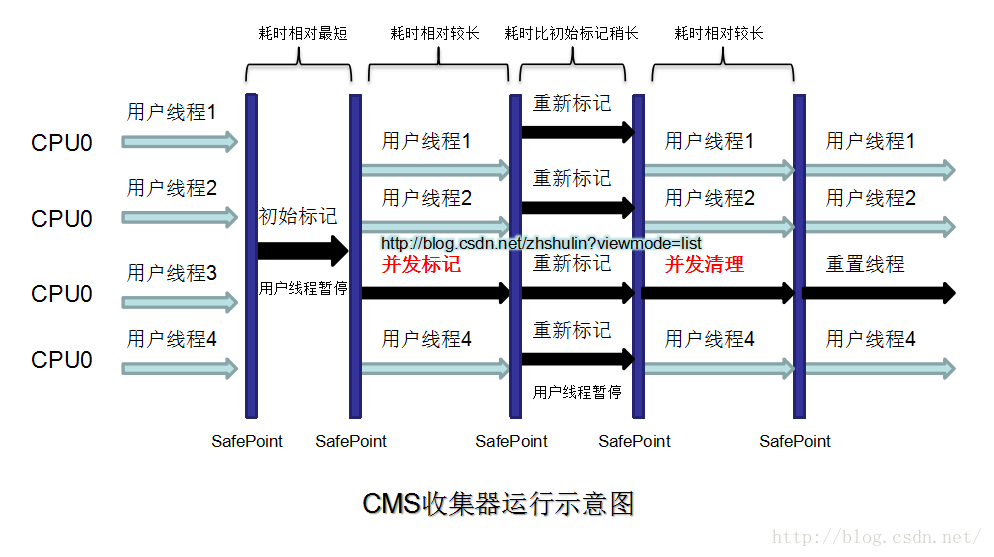
    CMS，Concurrent Mark Sweep，从名字就可以看出来这是一个基于”标记-清除“算法的并发垃圾收集器（注意并发和并行的区别），用于老年代的垃圾收集，它在Sun的一些文档中被称为并发低停顿收集器（Concurrent Low Pause Collector）。CMS收集器是一种以获取最短停顿时间为目标的收集器，优先满足重视服务响应速度的需求。CMS的运行过程相对前面几种收集器来说比较复杂，整个过程分为4步：

1. 初始标记：仅仅标记一下GC Roots能直接关联到的对象，速度很快
2. 并发标记：GC Roots Tracing，梳理引用链
3. 重新标记：修正并发标记过程中，用户线程运行导致标记变动的那一部分对象的标记记录。
4. 并发清除

    其中，初始标记和重新标记都需要”Stop the world“，但是整个过程中耗时最长的并发标记和并发清除阶段都是可以与用户线程一起工作的，所以**总体来说，CMS收集器的内存回收过程是与用户线程一起并发地执行的。**

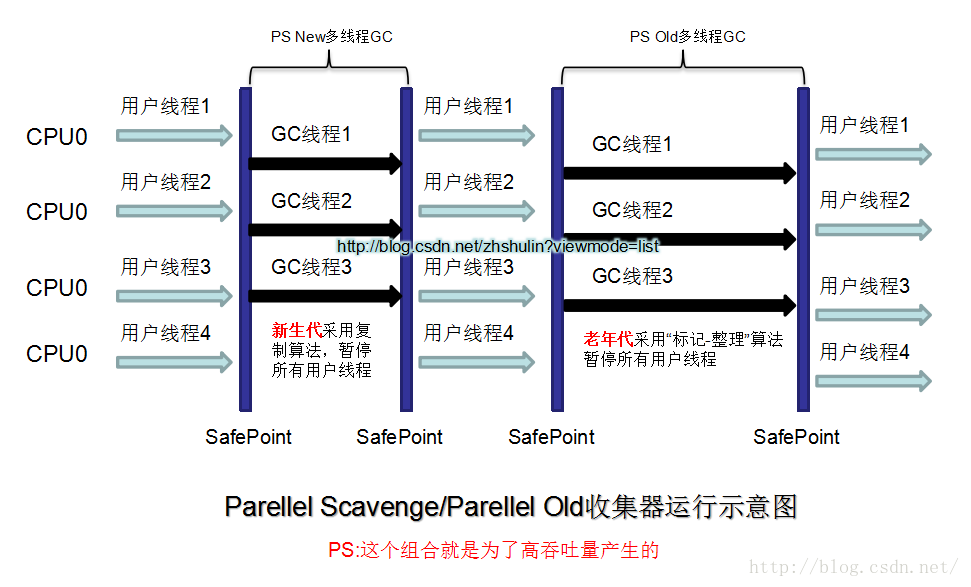
    虽然CMS很优秀，是HotSpot虚拟机中有史以来的第一款真正意义上的并发收集器，但是CMS还远远达不到完美的程度，它有三个显著的缺点：

1. 对CPU资源非常敏感
2. 无法处理浮动垃圾，所谓的浮动垃圾就是CMS并发清除阶段用户线程运行产生的垃圾，这部分垃圾必须等待下一次的垃圾收集来清除。所以CMS执行GC的时候需要预留足够的内存空间（默认32%，可调节）给用户线程使用，如果预留空间无法满足用户线程的内存需求，那么就会发生“Concurrent Mode Failure”失败，然后虚拟机就会启动Serial Old来重新进行老年代的垃圾收集，这样就会导致停顿时间很长了。
3. 会产生空间碎片（”标记清除“算法的特点），CMS在Full GC发生之后附带了一次碎片整理过程，而内存整理是无法并发的，导致停顿时间不得不变长。发生这个问题的时候，可能就会调用Serial Old来处理老年代的垃圾回收了。



* Parallel Old

    Parallel Old是Parallel Scavenge收集器的老年代版本，简称PS Old，使用了多线程和”标记-整理“算法，这个收集器是在JDK1.6中才提供的，在此之前PS new的地位比较尴尬，因为在此之前老年代的垃圾回收只有Serial Old这一种收集器，与Serial Old配合，Parallel Scavenge无法产生理想的回收效果，吞吐量在老年代很大且硬件比较高级的环境中可能还不如使用ParNew与CMS的组合”给力“，而PS Old产生之后，PS New才变得名副其实。  
    也就是说，在JDK1.6及之后，在注重吞吐量和CPU资源敏感的场合，都可以优先考虑PS New和PS Old的组合。



* G1

       在JDK6u14中提供了Early Access版本的G1收集器以供试用，直到JDK7u4的时候才正式发布。**G1是一款面向服务端应用的垃圾收集器，HotSpot开发团队赋予它的使命是未来替换掉CMS收集器，从这点上看，G1也是追求短停顿时间的。**

    G1收集器与上述的6种收集器相比，具有以下的特点：

1. 并行和并发：G1收集器不仅能充分利用多CPU、多核环境下的硬件优势来减少停顿时间，而且仍可以通过并发的方式让用户线程继续执行
2. 分代收集：虽然在使用G1收集器的时候，JAVA堆的内存布局已经不再是物理隔离了，仅仅是逻辑隔离，但是分代的概念得到保留，G1可以独立管理整个GC堆
3. 空间整合：CMS采用了“标记-清除”算法，会产生内存碎片。而**G1整体看来采用的是“标记-整理”算法**，**局部看来采用的是“复制”算法**，故而G1运行期间都不会产生内存碎片，这种特性有利于程序长时间的运行。
4. 可预测停顿：这是G1相比较CMS的一大优势，G1除了和CMS一样追求低停顿外，**还能建立可预测的停顿模型**，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不超过N毫秒，这几乎已经是实时JAVA（RTSJ）垃圾收集器的特征了。

    G1最大的特点在我看来就是G1将**[Java](http://lib.csdn.net/base/javase" \o "Java SE知识库" \t "http://blog.csdn.net/zhshulin/article/details/_blank)**堆划分成多个大小相等的独立区域（Region），虽然保留了新生代和老年代的概念，但是它们已经不再是物理隔离了，而都是一部分Region的集合。G1在后台维护一个 优先列表，这个列表中保存了G1收集到的各个Region里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收需要的时间的经验值），当需要进行垃圾回收的时候，根据用户允许的收集时间，优先回收列表中价值最大的那个Region（这就是Garbage-First，G1名字的由来）。这种使用Region划分空间以及根据优先级的区域回收方式，保证了G1收集器可以在有效的时间内获得尽可能高的收集效率，同时也避免了在整个JAVA堆中进行全区域的垃圾收集。

    G1产生的原因我认为就是HotSpot团队对于低延时和吞吐量两者同时考虑，不断追求一个完美的垃圾收集器的产物，虽然在执行流程上和CMS有差不多，并且在初始标记和最终标记阶段都需要暂停用户线程，但是通过重新定义JAVA堆，引出了Region的概念，成功的让其在性能上能兼顾到低延时和吞吐量，且不需要依赖其他收集器。G1的未来就是优化初始标记和最终标记阶段，如果能解决这两个阶段的用户线程暂停，实现并发，那么就很有可能产生一个近似理想状态的一个垃圾收集器。

    不过用户对于新生事物必须的认同需要一定的时间，并且之前垃圾收集器相互配合也可以满足用户需求，那么G1对于大多数公司来说不是必需品，但是，随着技术的不断成熟，我认为G1很有可能成为Server模式下的HotSpot默认的收集器。

    对于G1收集器，可以参考：[http://f.dataguru.cn/thread-514678-1-1.html](http://f.dataguru.cn/thread-514678-1-1.html" \t "http://blog.csdn.net/zhshulin/article/details/_blank)

