**J2SE基础**

**1. 九种数据类型大小，以及他们的封装类**

char 2Bite Character /u0000(null)

byte 1B Byte 0(bity)

short 2B Short 0(short)

int 4B Integer 0

long 8B Long 0L

boolean 1B Boolean false

float 4B Float 0F

double 8B Double 0D

void Void

注：基本数据类型占用的空间是不变的，这种不变性也是java具有可移植性的原因之一。它是放在栈中直接存储值，声明时系统会自动给它分配存储空间，所有的数值类型都有正负号。而引用类型声明时只是分配了引用空间（栈内存）

**2.Switch能否用String做参数**

在JDK1.7版本以前不能用。低版本仅支持byte、char、short、int和Enum类型。

在JDK1.7以后增加对Sring的支持。

**3.equal 与 == 的区别**

“==”： 基本类型，比较变量对应的值是否相等

对象/引用，比较引用变量的内存地址是否相等（栈内存上的引用，堆内存上的地址）。

“equals”: 基本类型，没有该方法。

对象/引用，对于String类型（或其他的包装类Long、Integer..）的引用对象，是比较内容是否相等（String内部重写了equals()），其他对象则是比较内存地址是否相等。（需要重写equals()方法）

**4.object有什么共有方法**

equals()

hashcode()

toString()

finalize() 垃圾回收

wait()

notify()

getClass()

**5.Java四大引用**

强引用：我们正常New出一个对象，这个对象就是强引用对象。强引用对象是我们开发中最常见最普遍的一个引用。拥有强引用的对象垃圾回收器都不会去回收。

软引用：用SoftReference来创建一个对象。这个对象就是软引用对象。软引用对象一般不会被垃圾回收器回收。直到内存空间不足的时候才会被回收。软引用一般可以用来实现对象缓存机制。

注：软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

弱引用：用WeakReference来创建一个对象。这个对象就是弱引用对象。弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

虚引用：与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。虚引用来跟踪对象的垃圾回收过程。

**6.Hashcode的作用**

Hashcode()用来返回对象的哈希码值。Hashcode用来**提高数据结构中数据的查询速度**。（时间复杂度从On –> O1

如果两对象的equals() 方法相等，则在两个对象中的每个对象上调用 hashCode()都必须生成相同的整数结果。

如果equals方法被重写，则要求hashcode()方法也要重写。

**7. ArrayList、LinkedList、Vector的区别**

（1）ArrayList：底层基于数组的数据结构存储数据，当数据超出数组范围时，其大小将会动态的进行增长（**50%**增长）。没有使用线程安全，性能会比较高。

（2）Vector：底层基于数组的数据结构存储数据，当数据超出数组范围时，其大小将会动态的进行增长（**100%**增长）。Vestor底层进行了同步保证线性安全，性能相对ArrayList会低些。

（3）LinkedList：底层基于双向链表的数据结构存储数据，因此没有太多的数据超出的顾虑。

注：一般来说：

1. 对于数据的查询和更新，使用ArrayList或Vector优势表较大，对于给定索引下标，ArrayList和Vector查找数据的时间复杂度为O1，而Linked需要移动链表指针，时间复杂度为On。
2. 同理，对于数据的增删操作，使用LinkedList的优势更大，因为对给定位置的数据增删LinkedList只需要前后对象指针，时间复杂度为O1，而ArrayList和Vector在增删指定位置数据后，还要对后面数据进行移动，时间复杂度为On。
3. Vector是线程安全的，底层进行了线程同步处理。ArrayList则是非线程安全的，没有进行线程同步处理，效率会相对高些。

**8.String、StringBuffer、StringBuilder的区别**

String：字符串常量，一旦初始化就不能再修改他的值。以后对String类型的数据修改都是重新创建一个新对象。

StringBuffer/StringBuilder：字符串缓冲区，是一个可变的对象，与String区别在于对字符串修改是不用重新创建一个新对象。对于需要经常修改的字符串推荐使用StringBuffer或StringBuilder来存储数据，提高系统性能。

StringBuffer和StringBuilder的区别在于，StringBuffer底层实现了同步，是线程安全的。而StringBuilder底层没有实现同步，是非线程安全的，因此性能也会相对高些。

**8.Map、Set、List、Queue、Stack的特点和用法**

Collection三个子接口：List、Set、Queue

List三个主要实现类：ArrayList、LinkedList、Vector

Set两个主要实现类：HashSet、TreeSet

Map又有四个实现类：HashMap、Hashtable、TreeMap、LinkedHashMap

Stack是基于Vector的实现。

以上都是存储数据的容器，

**对于Map：**基于键值映射，Key、Value一一映射，容器内不能包含重复的Key。

TreeMap，数据有序，键不可以为null值，内部实现使用红黑树实现的；

HashMap，数据无序，线程不安全，**键、值 都可以为null。**

HashTable，是线程安全的，不能存储null值

**对于List：**有序的可以索引到元素的容器，并且里面的元素可以重复。

ArrayList 是线程不安全的， Vector 是线程安全的，这两个类底层都是由数组实现的。

LinkedList是非线程安全的，底层是由双向链表实现的。

查询、更新 使用 ArrayList / Vector，插入、删除 使用LinkedList

**对于Set：**不包含重复元素的集合，**不允许重复项**，Set中最多包含一个null元素，支持Iterator/for-earch遍历。

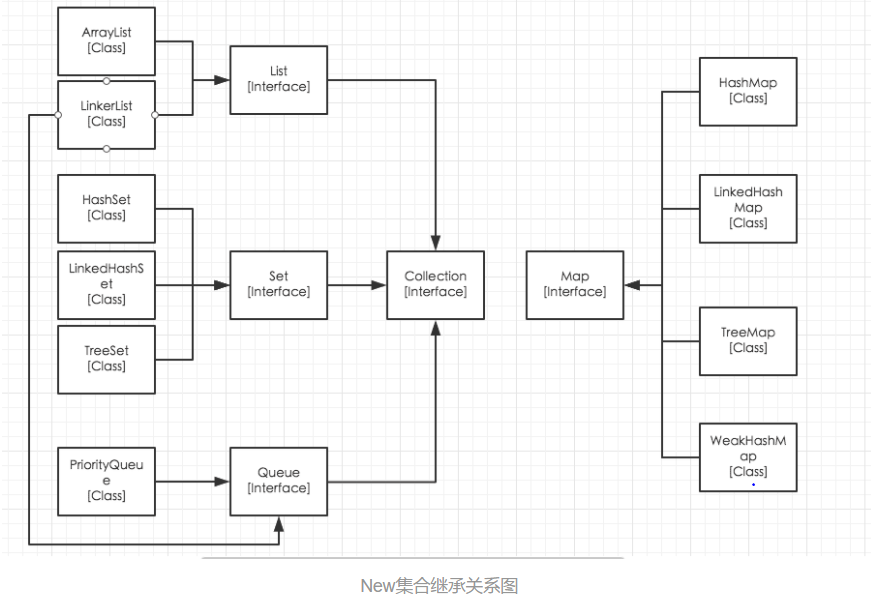
HashSet，无序的、无重复的数据集合。基于HashMap。（允许有一个Null值）

TreeSet，有序的，无重复的数据集合。元素需要实现Comparable接口。基于TreeMap（不允许有Null值）

LinkedHashSet，在HashSet基础上满足插入或访问顺序的结构，链表的实现避免了插入到hashset中的元素的顺序的紊乱，也不会给TreeSet招致增加实现Competor的代价，**允许一个为null**。

**对于Queue：**遵从先进先出原则。-- 常见实现：Deque

**对于Stack：**遵从后进先出原则。Stack是Vector的实现，常用方法push和pop操作，以及取堆栈顶点的peek()



**10. HashMap和HashTable的区别**

HashMap和Hashtable都是Map接口的实现。

区别在于HashMap是非线程安全的，涉及线程同步问题需要自行增加同步处理。而Hashtable底层实现了线程同步，性能效率上相对HashMap会低一些。

HashMap的Key允许为Null（1个），而Hashtable的Key不允许为Null。

hashMap.entrySet().iteraor(); 键值集合

hashMap.keySet().iterator(); 键集合

**11.HashMap和ConcurrentHashMap的区别，HashMap的底层源码**

---HashMap和ConcurrentHashMap(分段数组+链表)底层都是由数组+链表的离散数据结构构成。

HashMap是非线程安全的，底层没有进行线程同步处理。

ConcurrentHashMap是线程安全的，底层实现线程同步。

ConcurrentHashMap(高版本JDK)的同步不像HashTable那样（全表锁）的方式实现，实现同步的细节更复杂（分段锁），性能更高。

在没有线程并发问题情况下使用HashMap会好些，否则在多线程并发环境下使用ConcurrentHashMap会更好。

扩容：当Map中元素总数超过Entry数组的75%，触发扩容操作newSize = oldSize \* 2

ArrayList则是初始大小10， 不够触发扩容：newSize = oldSize + oldSize/2 + 1；

**12.TreeMap、HashMap、LinkedHashMap的区别**

**--**都是Map接口的实现类。

**--**TreeMap底层是通过红黑树（二叉树）实现，数据是有序的。

**--**HashMap是最常见也是最常用的Map容器，底层是通过hashcode来存储数据。只能有一个键为Null。本身是非线程安全的，需要同步可使用Collections.synchronizedMap(Map m);

**--**LinkedHashMap底层通过链表和哈希表的数据结构实现。因此对数据的增删操作性能效率更高。而查询或更新数据使用HashMap优势更大。

**13.Clooection包结构，Collections的区别**

Collection是集合的上级接口，也就是List、Set和Queue的父接口。

Collections是针对集合的辅助类，能够加强基本集合的功能，比如对集合实现同步（ synchronsizedList(List l) ），或者对集合进行排序（ sort() ）;

**14.try、catch、finally的区别，try里面return finally还会执行吗？**

Try、Catch块：一般代码有可能出现Expection异常情况是需要执行trycatch操作。

而finally中代码能够在代码执行后期或出现异常之后做一些收尾工作，比如关闭流资源，关闭数据库连接等等。。

Finally中的代码一般都会执行到，无论Try块中是否出现Expction异常，除非手动强制关闭Jvm虚拟机才不会被执行。

**15.Excption与Error包结构，其中OOM有遇到过哪些情况。SOF有遇到过没？**

一般Java的错误异常结构分为3种：

1 错误Error

代码出现重大错误，直接导致程序终止。比如内存溢出、栈溢出等。

2 运行时异常RuntimeExcption

这类异常发生原因一般是程序的BUG，无法通过trycatch处理，典型例子：NullPointerException 、ClassCastException 、ArrayIndexOutOfBoundsException 。。

3 受检异常/编译时异常 Checked Excption

受检异常能够通过trycatch处理，或者throw抛给上级。典型例子：各种IOExcption、EOFException(end of file)、FileNotFoundExcption。。

其中Error错误和RuntimeExcption运行时异常都不能被编译器检测。遇到就需要通过检查代码或Debug程序。

**OOM:** 内存溢出，当内存占有量超过虚拟机分配的最大值时就会产生OOM。

一般产生原因：加载对象多大、资源过多，来不及加载、各种内存泄露也有可能引发OOM。

一般解决办法：对象用软引用来封装、对图片进行边界压缩、缓存处理。修改虚拟机的堆内存分配大小。

**SOF:** 栈溢出，当应用程序递归太深而发生堆栈溢出时，抛出该SOF异常。

一般产生原因：程序内**递归调用**、大量使用循环或循环嵌套。。

一般解决办法：避免程序的递归调用，或大量循环。

**16.Java面向对象的三特征和含义**

**继承、封装、多态**

继承：继承的概念是相对与子类和父类而言的，子类继承父类所有的属性和方法，同时也有自己的独立行为也可以复写父类方法。

封装：对类内部实现细节进行抽取、封装，隐藏内部实现、对外暴露最简单的接口。

多态：接口的多种不同的实现方式即为多态。相同的行为，不同的实现。代码中就是同一个接口使用不同的实例而执行不同操作。

**17.Override和OverLoad的区别**

Override：重写方法，一般对于子类和父类而言的概念。子类方法用Override声明就代表该方法是重写父类的方法。

OverLoad：重载方法，一般对于同一个类而言，表示同一个类中能够有多个名字相同的方法，但参数类型和参数个数不同。

Override是子父类间多态的表现，OverLoad是一个类上的多态表现。

**18.Interface和Abstract的区别**

Interface声明的是接口，abstract声明的是抽象类或抽象方法。

Interface接口上的方法和变量都是共有的，用于让实现类实现。

abstract抽象类中则是可以声明私有方法或私有变量。子类实现只需实现abstract声明的抽象方法。和选择性的重写方法。

一般来说开发中最顶级的是接口，然后抽象类实现接口，最后写具体的实现类。

**19.Static Class和Not Static Class的区别**

一般来说Static指的是静态内部类，而Not Static Class则是指非静态内部类。

对于静态内部类：

1. 不需要有指向外部类的引用。
2. 只能访问外部类的静态成员，不能够访问外部类非静态成员。

对于非静态内部类：

1. 会持有外部类的引用（导致内存泄露的常见原因）
2. 能够访问外部类的静态和非静态成员。
3. 非静态内部类的创建不能脱离外部类实例。

**20.Java中多态的实现**

多态的含义简单来说是相同的方法拥有不同的实现。

一般来说多态的实现分为两种：

子类父类而言：子类重写父类方法，常见的是用父类引用指向子类对象。在运行时动态确认子类具体实现。

对单独的一个类而言：通过方法重载来实现类内多态。

**21.实现多线程的两种方式：Thread、Runable**

（1）继承Thread然后去重写他的run()方法。创建该Thread类调用Start方法；

缺点：Java只能单继承，一旦集成了Thread类就无法基层其他类。

（2）实现Runnable接口，实现run()方法。然后通过new Thread(Runnable run)把runnable对象传入进去，调用start()方法即可实现多线程。（推荐）

public class ThreadTest extends Thread{

继承Thread，重写run（）方法  
 public static class NewThread extends Thread {  
 @Override  
 public void run() {  
 System.***out***.println("This is Thread Running~");  
 }  
 }  
  
 //实现runnable接口

public static class MyRunable implements Runnable{  
 @Override  
 public void run() {  
 System.***out***.println("This is My Runnable!~");  
 }  
 }  
  
  
 public static void main(String[] args) throws InterruptedException {  
 new Thread(new MyRunable()).start();

new NewThread().start();  
 }  
}

**22.线程同步的方法：synchronized、lock、reentrantLock**

Synchronized是Java的一个关键字，是一种同步锁。他能够修饰代码块、静态或非静态方法

修饰方法时，一般以当前对象作为锁。

修饰静态方法时，一般以类Class对象作为锁。

修饰代码块时，需要自行传入一个对象作为同步锁。

Lock是一个接口，而ReentrantLock则是Lock接口的实现。ReentrantLock拥有Synchronized的功能，都是用于解决多线程环境下并发同步的问题。

**23.锁的等级：方法锁、对象锁、类锁**

方法锁：一般通过synchronized关键字在方法上声明。

public synchronized void method() {

　　System.out.println("我是对象锁也是方法锁");

　　}

对象锁：通过synchronized修饰的方法或synchronized（this）同步的代码块。进入同步区域时，需要获取对象锁。

public synchronized void method1() {

// 同步方法

System.out.println("对象锁：方式一");

}

public void method2() {

// 同步代码块

synchronized (this) {

System.out.println("对象锁：方式二");

}

}

类锁：通过synchronized关键字修饰的静态代码或synchronized（xx.class）同步的代码块。（静态方法或静态变量在内存中都只存在一份，因此同步是共用一把锁）进入同步区域时，需要先获取类锁。

public static synchronized void method1() {

System.out.println("类锁:方式一");

}

public void method2() {

synchronized (Trhead\_.class) {

System.out.println("类锁：方式二");

}

}

**24.手撕生产者、消费者模式**

**25.ThreadLocal的作用和设计理念**

ThreadLocal是一个创建线程私有变量的类，一般情况我们创建变量都能够被所有线程所修改，而使用ThreadLocal创建的变量只能够被当前线程所访问和修改，其他线程无法被访问。

显然，ThreadLocal这个类的目的是用来解决多线程环境下的并发同步问题。(ThreadLocal的本质不是Thread)

**26.ThreadPool的用法和优势**

ThreadPool也就是线程池。

优势：通过固定数量的线程来统一为大量的操作服务，减少高并发环境下大量线程的创建和销毁所消耗的时间和资源。从而执行提高效率。

用法：Executor.newFixedThreadPool(4);

executor.execute(new Runnable());

底层：

executor = new ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime,  
 TimeUnit.*SECONDS*, new LinkedBlockingDeque<Runnable>(),  
 Executors.*defaultThreadFactory*(), new ThreadPoolExecutor.AbortPolicy());

**27.Concurrent包下的其他：ArrayBlockingQueue、CountDownLatch..**

ArrayBlockingQueue: 一个由数组支持的有界阻塞队列。FIFO原理。

CountDownLatch: 是一个倒数计数的锁，当倒数到0时触发事件，也就是开锁。

**28.wait() 和 sleep() 区别**

wait 是object的方法，sleep是Thread的方法。

每当执行到wait或sleep时，进程都会停止执行。

他们的区别在于：

wait：wait睡眠时会释放对象锁，不占用cpu资源。

sleep：sleep睡眠时会一直保持对象锁，会一直占用cpu资源。

1. **final** Object synObj = **new** Object();
2. Thread t1 = **new** Thread(**new** Runnable() {
3. @Override
4. **public** **void** run() {
5. **synchronized**(synObj) {
6. System.out.println("T1获取synObj的对象监视器，开始执行同步块");
7. **try** {
8. TimeUnit.MINUTES.sleep(1);
9. System.out.println("T1在 wait()时挂起了");
10. synObj.wait();
11. System.out.println("T1被T2唤醒后并重新获得synObj的对象监视器，继续执行");
12. }**catch**(InterruptedException e) {
13. e.printStackTrace();
14. }
15. System.out.println("T1获取synObj的对象监视器，结束同步块");
16. }
17. };
18. });
19. t1.start();

22. Thread t2 = **new** Thread(**new** Runnable() {
23. @Override
24. **public** **void** run() {
25. System.out.println("T2启动，但是因为T1占用了synObj的对象监视器，则等待T1执行synObj.wait来释放它");
26. **synchronized**(synObj) {
27. **try** {
28. System.out.println("在T1执行synObj.wait后，T2获取synObj的对象监视器，进入同步块");
29. synObj.notify();
30. System.out.println("T2执行synObj.notify()，T1被唤醒，但T2还在同步块中，没有释放synObj的对象监视器，T1等待synObj的对象监视器");
31. TimeUnit.MINUTES.sleep(1);
32. System.out.println("T2结束同步块，释放synObj的对象监视器，T1获取到synObj的对象监视器，并执行wait后面的操作");
33. }**catch**(InterruptedException e) {
34. e.printStackTrace();
35. }
36. }
37. };
38. });
39. t2.start();

**29.forEach和For循环的效率对比**

ForEach是属于高版本的语法糖，底层实现是通过iterater进行遍历。

对于一般的遍历，普通For循环效率会更高。其次是iterater迭代器，效率最低的是forEach遍历。

For > Iterater > ForEach

**30.Java IO 和 NIO**

Java中的IO一般分为：本地IO 和 网络IO

本地IO就是应用从本地磁盘中读写数据。

网络IO就是通过ip+端口进行的远程数据读写操作。

NIO即 No-Blocking IO：非阻塞式IO，在输入输出操作的同时可以做别的事。

而Java IO是阻塞式IO，读写数据时不能做其他事。

JavaIO默认没有缓冲区的，而NIO有缓冲区。

**31.反射的原理和作用**

原理：我们的Java程序要被执行，就需要将编译完成的Java类加载到虚拟机上运行。这样的程序执行需要我们在**编译期知道哪个类被加载**。 **JAVA语言编译之后会生成一个.class文件，反射就是通过字节码文件找到某一个类、类中的方法以及属性等。**而我们反射则是在编译期并不知道加载类的信息，直到运行期才动态得去获取和加载这个类，运行期才动态加载一个类，获取他的类信息-属性和方法；动态构建类对象，并调用其方法或属性。

作用：反射一般都3种用途：

1. 运行时动态获取对象所属的类。
2. 运行时动态的构造类对象。
3. 运行时动态获取类的属性和方法。（getFields()、getMethods()）

**32.泛型的常用特点，List<String>能否转为List<Object>**

实现类型安全：泛型定义了变量类型的约束，另变量在编译阶段就能够进行类型检查。

消除强制类型转换：使用泛型的话，能够避免强制类型的转换。

List<String>理论上不该转化为List<Object>: 强转-> List<Object> objs = (List<String>)strs

33.解析XML的几种方法原理及特点：DOM、SAX、PULL

DOM: 将整个xml文档树加载进内存。能够读写xml内容。。

SAX: 采用事件驱动模型，不需要将整个xml文档加载进内存，而是在读取文档的过程中就解析它。内存占用小。不能写入xml

PULL: android内置的xml解析器。原理跟SAX解析类似。Android下开发推荐PULL。解析效率更高。

**34.Java和C++对比**

Java跨平台，程序移植性好。

C++程序性能、运行速度、执行效率。

Java更多用来开发网站、手机应用、小型软件居多。C++用来开发驱动、游戏、嵌入式系统等对性能要求比较高的东西。

Java开发效率高、C++执行效率高。

**35.Java1.7 和 Java1.8的新特性。**

**JDK5: 自动拆装箱；可变参数；泛型；**

**JDK6: 插入式注解处理API；**

-- **JDK7**：Switch中可以使用String作为参数。 一个catch内可铺货多个异常，用 “ | ”隔开。

-- **JDK8**：接口中实现具体方法、静态方法。 Lambada函数，简化代码书写。

**JDK9：** 多版本兼容 JAR；私有接口方法；

**JDK10： 并行、完整的 G1 垃圾收集器；**

**36.设计模式：单例、工厂、适配器、责任链、观察者。**

创建：单例、工厂、Builder；

结构：Adapter、Proxy；

行为：策略、状态、Observer；

**37.JNI的使用**

2D/3D绘画渲染、各种驱动、媒体播放器等等高性能的需求一般都要使用JNI调用本地C/C++方法.

Java Native Interface - Java本地调用.

用来调用Native底层本地方法：C/C++函数。Native方法又能反向调用Java代码。