## 一 Java 局部内部类使用局部final变量

1. 内部类是外部类的一个成员，就像外部类的成员方法一样，所以内部类有权限访问外部类的所有成员，包括private的。
2. 内部类不能访问外部类方法中的局部变量，除非变量是final的(一般发生在方法中定义的内部类)。这是因为局部变量的生命周期原因。
3. 所谓“局部内部类”就是在对象的方法成员内部定义的类。而方法中的类，访问同一个方法中的局部变量，却必须要加上一个final。

原因是编译程序实现上的困难：

* 内部类对象的生命周期会超过局部变量的生命期。局部变量的生命期：当该方法被调用时，该方法中的局部变量在栈中被创建，当方法调用结束时，退栈，这些局部变量全部死亡。而内部类对象生命期，与其它类一样，当创建一个局部内部类对象后，只有当没有其它人再引用它时，它才能死亡。所以完全可能一个方法已调用结束（局部变量已死亡），但该局部类的对象仍然活着。即：局部类的对象生命期会超过局部变量。
* 局部内部类的对象访问同一个方法中的局部变量，那么这就要求只要局部内部类对象还活着，那么栈中的那些它要访问的局部变量就不能“死亡”（否则：它都死了，还访问个什么呢？）。这就是说：**局部变量的生命期至少等于或大于局部内部类对象的生命期。**
* 解决方法：局部内部类的对象可以访问同一个方法中被定义为final的局部变量。定义为final后，编译程序的实现方法：将所有的局部内部类对象要访问的final型局部变量，都**拷贝**成为该内部类对象中的一个数据成员。这样，即使栈中局部变量（含final）已死亡，但由于它是final,其值永不变，因而局部内部类对象在变量死亡后，照样可以访问final型局部变量
* 归纳总结：局部内部类对象中包含有要访问的final型局部变量的一个**拷贝**，成为它的数据成员。因此，正是在这个意义上，final型局部变量的生命期，超过其方法的一次调用。**严格来说，方法调用结束，所有的局部变量（含final）全死亡了。但：局部内部类对象中有final型局部变量的拷贝。**

## 二 java 24 \* 60 \* 60 \* 1000（一天）

Date d=new Date();

SimpleDateFormat df=new SimpleDateFormat("yyyy-MM-dd");

System.out.println("today:"+df.format(d));

System.out.println("the day **befor yestoday**:" + df.format(new Date(**d.getTime() - (long)2 \* 24 \* 60 \* 60 \* 1000)**));

System.out.println("the day **after tomorrow**:" + df.format(new Date(**d.getTime() + (long)3 \* 24 \* 60 \* 60 \* 1000)**));

## 三 Java中子类和父类的初始化顺序-静态优先，父类优先

类的初始化顺序是：

1、初始化父类中的静态成员变量和静态代码块。

2、初始化子类中的静态成员变量和静态代码块。

3、初始化父类中的普通成员变量和代码块，在执行父类中的构造方法。

4、初始化子类中的普通成员变量和代码块，在执行子类中的构造方法。

## 四 Java transient关键字

java语言的关键字，变量修饰符，如果用transient声明一个实例变量，当对象存储时，它的值不需要维持。换句话来说就是，用transient关键字标记的成员变量不参与序列化过程。

## 五 Java >> n & << n

>> n （右移 除2的n次方）

<< n （左移 乘2的n次方）

## 六 Java Collection接口的toArry方法注意

1. public Object[] toArray()方法：方法中会重新构造一个Object[]数组，然后将集合中每个元素赋给数组，由于new的是Object数组，所以无法向上转型回之前的数组。
2. public <T> T[] toArray(T[] a)方法：返回传入的数组a指定的类型，a的length小于集合size时会利用反射重新构建一个指定类型数组，并把集合中所有元素赋给数组。

## 七 编程习惯

### 异常处理

<http://www.importnew.com/26858.html>

不要随便加null判断（并不“健壮”，反而掩盖错误）：如果写了，就必须测试为空和不为空的两种情况

不捕获异常，将异常交给调用者处理：比起从日志中寻找错误，应在编码阶段处理，尽早让错误抛出。

### 接口定义

<https://zhuanlan.zhihu.com/p/28708259>

### Controller规范

## 八 Java 8

<http://www.importnew.com/16436.html>

<http://www.importnew.com/11908.html>

1. Lambda表达式（闭包）；
2. 函数式编程支持： java.util.function.\* 下的众多接口；
3. 流API；
4. 接口的默认方法和静态方法；
5. 新的Date和Time API。

### 1 Optional 接口

<http://www.importnew.com/6675.html>

对对象的null值检测进行包装，如果值存在则isPresent返回true，否则false。

Optional.of：创建一个Optional，不能为null，赋值抛出NullPointerException。

Optional.ofNullable：可以创建一个null的Optional。

isPresent：值存在返回true，否则false。

ifPresent：值存在就调用Consumer对象的accept方法，否则无任何动作。

get：返回值，空Optional抛NoSuchElementException异常。

orElse：值存在返回，否则返回指定值。

orElseGet：值存在返回，否则调用Supplier的get返回。

orElseThrow：存在返回，否则抛出指定异常。

map、flatMap、filter

### 2 Lambda

Lambda表达式允许将函数作为一个方法参数传递，或许将代码看成数据。

用 () -> {} 代替整个匿名类。

new Thread( **() -> {**int i = 9 ; System.out.println(“somting”+i) ; }).start();

() 中参数可省略类型，没有参数时 () 仍然要保留。

{} 只有一条语句时 {} 可以省略，语句尾的 ; 也可以省略。

Arrays.asList(“a”, “b”, “c”, “d”).forEach(va -> System.out.println(va));

**FunctionalInterface：函数式接口，如果一个接口只有一个方法，那么编译器会将该接口视为函数式接口，如果在接口上加注解FunctionalInterface注解，那么接口就会被强制要求符合函数式接口的规范（只有一个抽象方法，默认，静态方法不包括在内）。**

### 3 Predicate接口

适用于过滤。

该接口有一个test方法，返回boolean，表示检查结果。提供了类似于逻辑操作符AND和OR的方法and()、or()、xor()。可以将多个条件进行组合。

Predicate<String> testStart = (str) -> str.startsWith("a");  
Predicate<String> testContain = (str) -> str.contains("b");  
boolean result = testStart.and(testContain).test("acvvb"); // true

### 4 Map的Reduce

map方法（Function接口）遍历并将每一个数乘以2，reduce（BiFunction接口）方法将结果汇总于sum。

List<Integer> list = Arrays.*asList*(1, 2, 3, 4, 5, 6);  
Optional<Integer> reduce = list.stream().map(va -> va >> 1).reduce((sum, cur) -> sum += cur);  
*o*.accept(reduce.orElse(-1));

### 5 Function<T,R>接口

用于产生对象，函数式接口，接受一个参数，构造并返回目标对象。

R apply(T t);

### 6 BiFunction<T,R,U>接口

用于产生对象，接受两个参数，生成一个结果。

R apply(T t,U u);

### 7过滤集合元素

Filter方法（Predicate接口）遍历元素并检查，返回true保留，collect方法（Collector接口）将结果组合为新的集合。

List<Integer> list = Arrays.*asList*(1, 2, 3, 4, 5, 6);  
List<Integer> collect = list.stream().filter(va -> va % 2 == 0).collect(Collectors.*toList*());  
*o*.accept(collect);

### 8 Supplier接口

用来产生对象

T get();

### 9 计算集合元素的最大，最小，总和以及平均值

IntStream、LongStream和DoubleStream中有个summaryStatics方法，可以返回IntSummaryStatics、LongSummaryStatics或DoubleSummaryStatics，描述流中的各种摘要数据。

Stream的mapToInt/Long/Double方法可将指定对象作为参数，产生对应的基本类型，mapXXX中接受的接口为：ToInt/Long/DoubleFunction，方法为(以int为例)：int applyToInt(T value);

List<Integer> list = Arrays.*asList*(1, 2, 3, 4, 5, 6);  
IntSummaryStatistics statistics = list.stream().mapToInt(i -> i).summaryStatistics();  
statistics.getMax();  
statistics.getMin();  
statistics.getAverage();  
statistics.getSum();  
statistics.getCount();

### 10 Lambda VS 匿名内部类

this关键字：匿名类的this指向匿名类，而Lambda的this指向外部类。

编译方式：Lambda被编译为外部类的私有成员方法，而匿名类被编译为单独的类。

同：两种方式在引用外部的局部变量时局部变量要被final修饰。

### 11 方法引用

List<Car> cars = new ArrayList<>();

1. 方法构造器（无参可访问的构造器）引用：Class::new ，可以便捷的创建一个对象，通常配合Supplier接口使用。
2. 引用静态方法（一参可访问）：Class::static\_method，通常配合Consumer接口使用。
3. 引用特定类的方法（无参可访问）：Class::method
4. 引用特定对象的方法：instance::method

### 12 新的Date/Time API

在java.time.\*包下：

新的API涵盖了所有处理日期、时间、时区、过程与时钟的操作。

1. Clock类：

UTC（Universal Time Coordinated）为世界协调时间，与格林尼治时间一样，与英国伦敦时间也一样。北京是东八区，领先于UTC8个小时。

Clock clock = Clock.*systemUTC*();  
 Instant instant = clock.instant();  
 Instant plus = instant.plus(8, ChronoUnit.*HOURS*);// 加 8 小时  
  
 Instant minus = instant  
// .plus(24 \* 60 \* 60 \* 1000, ChronoUnit.MILLIS) // 加一天  
 .minus(24 \* 60 \* 60 \* 1000, ChronoUnit.*MILLIS*);// 减一天  
 *o*.accept(instant); // UTC 时间：2017-10-24T07:11:12.576Z  
 *o*.accept(plus);// 北京时间：2017-10-24T15:11:12.576Z  
 *o*.accept(minus);// UTC 时间减一天：2017-10-23T07:11:12.576Z

1. LocalDate、LocalTime和LocalDateTime

三者只持有ISO-8601格式具有时区信息的日期与时间。前两者都可以从Clock中获得。

Clock clock = Clock.*systemUTC*();  
LocalDate date = LocalDate.*now*();  
LocalTime time = LocalTime.*now*(clock);  
LocalDateTime dateTime = LocalDateTime.*now*();  
*o*.accept(date); //2017-10-24  
*o*.accept(time); // 07:22:07.781 系统时间：15:22:37.050  
*o*.accept(time.getNano()); // 319000000  
*o*.accept(dateTime); // 2017-10-24T15:23:54.109

1. Duration，在秒与纳秒级别上的一段时间。简化了计算两个时间差的不同。

Duration duration = Duration.*between*(  
 LocalDateTime.*of*(2014, Month.*JULY*, 13, 21, 0, 0), // 2014-07-13 21:00:00  
 LocalDateTime.*of*(2014, Month.*JULY*, 14, 22, 13, 16)); // 2014-07-14 22:13:16  
*o*.accept(duration.toDays()); // 1  
*o*.accept(duration.toHours()); // 25  
*o*.accept(duration.toMinutes()); // 1513  
*o*.accept(duration.toNanos()); // 90796000000000

## 九 java值传递

Java使用对象的引用进行计算，但是java在向方法传递对象时传递的不是引用，是值。

常见的swap函数：

public void swap(int a,int b) { // 错误的交换方法

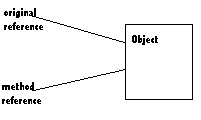
int temp = a;

a = b;

b = temp;

}

但swap方法返回时，a，b并没有交换。Java通过传值来传递引用：这意味着，**传向函数的引用实际上是原始引用的副本**。进行交换操作时，交换的是副本引用，真正的引用并未交换。（我们应该交换原始的引用，而不是它们的副本）。



当函数被调用后，一个对象至少存在两个引用。

## 十jvm类加载机制

<http://shulianghan.iteye.com/blog/1699341>

1. 类加载器深入剖析Java虚拟机与程序的生命周期:

当我们执行一个java程序的时候,会启动一个JVM进程,当程序执行完之后,JVM进程就消亡了;

在如下情况下JVM将结束生命周期:

1. **System.exit(int)方法:**

当执行这个方法的时候,虚拟机会退出;这个方法传入一个整形参数,这个参数是状态吗:如果这个整形是*0的话,就是正常退出,如果不是0的话,就是异常退出;*

1. **程序执行过程中,遇到了异常或错误,而异常终止:**

如果我们的程序中出现了异常,而不去处理,会将异常一直抛给main函数,main函数会将异常抛给JVM,JVM如果处理不了异常,JVM就会异常退出;

1. **由于操作系统出现错误导致JVM进程终止:**

JVM所依赖的平台出现错误,导致JVM终止;

1. 类的加载,连接和初始化加载:
2. **加载：**

JVM查找并加载类的二进制数据,将class字节码文件加载到内存中;

1. **连接： 验证 – 准备 – 解析**

确保被加载的类的正确性,使用javac编译工具生成的字节码文件能通过验证,如果不是由javac编译生成的字节码文件,或者生成的字节码文件不符合JVM虚拟机对字节码文件的要求,就会出现验证通不过的情况;比如说随便拿一个文件,将后缀名直接修改为.class,这样的字节码文件肯定不合法;

**准备:**

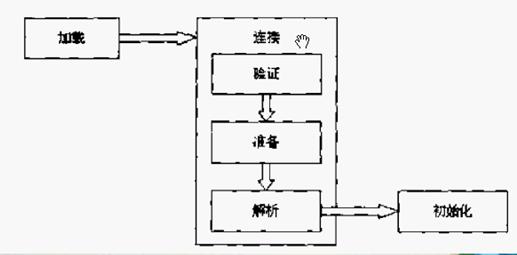
为类的静态变量分配内存,并将其初始化为默认值;

**解析:**

把类中的符号引用转为直接引用;

1. **初始化**:

为类的静态变量赋予正确的初始值(正确的值指的是用户赋的值);（好像这个与连接阶段的准备有些重复,在连接的准备阶段只是赋予初始变量,如果用户给这个变量赋了初始值,那么这个变量在连接的准备阶段仍然会赋予初始值;在这个阶段,才会真正的将初始值赋给静态变量;）



1. Java程序对类的使用方式有**主动使用**和**被动使用**;所有的JVM实现,必须在每个类或者接口,被java程序“首次主动使用”时才初始化他们;
2. 主动使用的6中方式:
3. 创建类的实例;
4. 访问某个类或接口的静态变量,或者对该静态变量赋值;
5. 调用类的静态方法;
6. 反射:Class.forName(“类名”);
7. 初始化一个类的子类,看做对父类的主动使用;
8. java虚拟机启动的时候,被标明启动类的类,即包含main方法的类,程序的入口;

除了上面6种主动使用之外,其它的情况均为被动使用,其它情况都不会执行第三步初始化;

1. 类的加载
2. 概念

* **类的加载:**

将类的.class文件中的二进制数据读入到内存中,将其放在运行时数据区的方法区内,然后再在堆区创建一个java.lang.Class对象,用来封装类在方法区内的数据结构;

* **反射:**

反射就是根据堆区的字节码文件,获取方法区的数据结构;

* **解析:**

Class对象是由JVM自己创建的,所有的对象都是经过Class对象创建,这个Class对象是反射的入口,通过Class对象,可以关联到目标class字节码文件的内部结构;

**所有的类对应的Class对象都是唯一的一个**,这个类是由JVM进行创建的,并且只有JVM才会创建Class对象;类加载的最终产品是位于堆区中的Class对象,Class对象封装了类在方法区内的数据结构,并且向Java程序员提供了访问方法区内的数据结构的接口(反射用的接口);(2)加载.class文件的方式从本地系统中直接加载:编译好的.class字节码文件直接从硬盘中加载;通过网络下载.class文件:将class字节码文件放在网络空间中,使用URLClassLoader来加载在网络上的.class字节码文件,使用默认的父亲委托机制加载字节码文件;从zip,jar等压缩文件中加载字节码文件:在开发的时候,导入jar包,就是这种方式;从专有的数据库中提取字节码文件;将java源文件动态编译为字节码文件;(3)类加载器lJava虚拟机自带的类加载器:-根类加载器(Bootstrap):是C++写的,程序员无法再java代码中获取这个类,如果使用getClassLoader()方法获取到的是一个null值;packagejvm;Java代码publicclassClassLoaderTest{publicstaticvoidmain(String[]args)throwsException{//java.lang包下的类使用的是跟类加载器进行加载的Classclazz=Class.forName("java.lang.String");System.out.println(clazz.getClassLoader());//自定义的类使用的是应用类加载器(系统加载器)Classclazz2=Class.forName("jvm.C");System.out.println(clazz2.getClassLoader());}}classC{}执行结果:nullJava代码sun.misc.Launcher$AppClassLoader@1372a1a-扩展类加载器(Extension):Java编写;-系统类加载器(应用加载器)(System):Java编写;用户自定义的类加载器:-自定义的类加载器都是java.lang.ClassLoader子类;-用户可以定制类的加载方式String类是由根类加载器进行加载的,我们可以调用Class对象的关于代理中创建对象的类加载器:创建代理对象的时候,动态创建一个类,然后使用指定的类加载器将这个类加载到内存中,然后用加载到内存中的类生成代理对象;创建代理对象

的方法:newProxyInstance(ClassLoaderloader,Class[]Interfaces,InvocationHandlerh)loader是定义的代理类的类加载器,中间的接口数组是代理类的要实现的接口列表,h是指派方法调用的调用处理程序;类加载器并不需要在某个类被“首次主动使用”时再加载它:-预加载机制:JVM规范允许类加载器在预料某个类将要被使用的时就预先加载它;-报错时机:如果在预加载的过程中遇到了字节码文件缺失或者存在错误的情况,类加载器会在程序首次主动使用(上面提到的六种情况)该类的时候报错(LinkageError错误);-不报错时机:如果这个错误的字节码文件所对应的类一直没有被使用,那么类加载器就不会报告错误,即便有错误也不会报错;LinkageError:这个错误是Error的子类,程序不能处理这些错误,这些错误都是由虚拟机来处理,这个错误表示出错的是子类,在一定程序上依赖于另一个类,在编译了前面一个类的时候,与后面所依赖的类出现了不兼容的情况;例如:我们使用了jdk1.6在编译一个程序,但是运行环境是jre1.5的,就会出现LinkageError错误;4.类的连接(1)定义类被加载之后,就进入链接阶段;链接:将已读入内存的二进制数据合并到虚拟机的运行时环境中去;链接顾名思义就是讲类与类之间进行关联,例如我们在类A中调用了类B,在链接过程中,就将A与B进行链接,将面向对象语言转化为面向过程语言;(2)类的验证类文件的结构检查:确保类文件遵从java类文件的固定格式,开始类的描述,声明,方法调用格式等;语义检查:确保类本身符合java语言的语法规定,比如final类型的类没有子类,final类型的方法没有被覆盖,在eclipse中这种错误编译的时候不能通过,但是通过其他的方法可以生成错误的字节码文件,这里就是检测恶意生成的字节码文件;字节码验证:确保字节码流可以被JVM安全的执行,字节码流代表java方法(包括静态方法和实例方法),它是由被称作操作码的单字节指令组成的序列,每一个操作码后面跟着一个或多个操作数,字节码验证步骤会检查每个操作码是否合法,即是否有着合法的操作数;下面是指令码组成的序列,类似于微指令:Jvm编译指令代码代码//CompiledfromByteToCharCp1122.java(version1.5:49.0,superbit)publicclasssun.io.ByteToCharCp1122extendssun.io.ByteToCharSingleByte{//Fielddescriptor#17Lsun/nio/cs/ext/IBM1122;privatestaticfinalsun.nio.cs.ext.IBM1122nioCoder;//Methoddescriptor#18()Ljava/lang/String;//Stack:1,Locals:1publicjava.lang.StringgetCharacterEncoding();0ldc[1]2areturnLinenumbers:[pc:0,line:25]//Methoddescriptor#2()V//Stack:2,Locals:1publicByteToCharCp1122();0aload\_0[this]1invokespecialsun.io.ByteToCharSingleByte()[25]4aload\_0[this]5getstaticsun.io.ByteToCharCp1122.nioCoder:sun.nio.cs.ext.IBM1122[23]8invokevirtualsun.nio.cs.ext.IBM1122.getDecoderSingleByteMappings():java.lang.String[27]11putfieldsun.io.ByteToCharSingleByte.byteToCharTable:java.lang.String[24]14returnLinenumbers:[pc:0,line:28][pc:4,line:29][pc:14,line:30]//Methoddescriptor#2()V//Stack:2,Locals:0static{};0newsun.nio.cs.ext.IBM1122[15]3dup4invokespecialsun.nio.cs.ext.IBM1122()[26]7putstaticsun.io.ByteToCharCp1122.nioCoder:sun.nio.cs.ext.IBM1122[23]10returnLinenumbers:[pc:0,line:22]}l二进制兼容性的验证:确保相互引用的类之间协调一致的;例如在A类的a()方法中调用B类的b()方法,JVM在验证A类的时候,会验证B类的b()方法,加入b()方法不存在,或者版本不兼容(A,B两类使用不同的JDK版本编译),会抛出NoSuchMethodError错误;(3)准备阶段在准备阶段,JVM为类的静态变量分配内存空间,并设置默认的初始值.例如下面的Sample类,在准备阶段,为int类型的静态变量分配4个字节,并赋予初始值0;为long类型的静态变量b,分配8个字节,并赋予初始值0;PS:在java中基本类型变量占用的空间是一定的,java运行在JVM上的,在C中,就要根据平台变化而变化了;publicclassSample{Java代码privatestaticinta=1;privatestaticlongb;static{b=2;}(4)类的解析在解析阶段,JVM会把类的二进制数据中的符号引用替换为直接引用,例如在A类中的a()方法引用B类中的b()方法;在A类的二进制数据中包含了一个对B类的b()方法的符号引用,这个符号引用由b()方法的全名和相关的描述符组成,在Java解析阶段,就会把这个符号引用替换为指针,这个指针就是C语言中的指针了,该指针指向B类的b()方法在方法区中的内存位置,这个指针就是直接引用;5.类的初始化在初始化阶段,Java虚拟机执行类的初始化操作,为类的静态变量赋予初始值,在程序中,静态变量初始化有两种途径:直接在声明处进行初始化,例如下面的Sample中的变量a;在静态代码块中进行初始化,例如下面的Sample中的变量b;Java代码publicclassSample{privatestaticinta=1;privatestaticlongb;static{b=2;}}6.面试题介绍Java代码publicclassPrepareOrInit{publicstaticvoidmain(String[]args){Singletonsingleton=Singleton.getInstance();System.out.println(singleton.count1);System.out.println(singleton.count2);}}classSingleton{privatestaticSingletonsingleton=newSingleton();publicstaticintcount1;publicstaticintcount2=0;privateSingleton(){count1++;count2++;}publicstaticSingletongetInstance(){returnsingleton;}}执行结果:10分析:这段代码与类的链接中的准备阶段和初始化阶段有关系,准备阶段是给静态的字段赋予默认值,初始化阶段给静态变量赋予正确的值,即用户的值;在主函数中,调用了类的静态方法,相当于主动使用,这里调用了类的静态方法;之后进行连接的准备操作,给类中的静态变量赋予初值,singleton值为null,count1与count2值为0;执行初始化操作,给类中的静态变量赋予正确的值,给singleton变量赋予正确的值,调用构造方法,此时count1与count2执行自增操作,两者都变成1,然后执行count1的赋予正确值操作,这里用户没有赋值操作,count2用户进行了赋值为0的操作,0将原来的1覆盖掉了,因此结果为1,0;Java代码publicclassPrepareOrInit{publicstaticvoidmain(String[]args){Singletonsingleton=Singleton.getInstance();System.out.println(singleton.count1);System.out.println(singleton.count2);}}classSingleton{publicstaticintcount1;publicstaticintcount2=0;privatestaticSingletonsingleton=newSingleton();privateSingleton(){count1++;count2++;}publicstaticSingletongetInstance(){returnsingleton;}}执行结果:11在准备阶段count1和count2都赋值为0,然后在初始化阶段,全部赋值为1;