# 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. 局部内部类不能访问所在外部方法中的局部变量，除非变量是final的(这里指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**。

* 局部内部类的对象需要访问所在方法中的局部变量，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（指变量的引用，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，照样可以通过拷贝的引用访问保存在堆上的变量（持有其引用，指向堆中创建的对象）
* 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，其仍然持有堆中对象的引用（因而无法被GC回收）。**

# 2. 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)**));

# 3. 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。

# 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。**

# Java >>、<<、<<<、>>>

>> n （右移 除2的n次方）

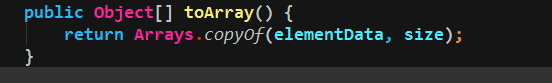
<< n （左移 乘2的n次方）

>>> n 无符号右移

<<< n 无符号左移

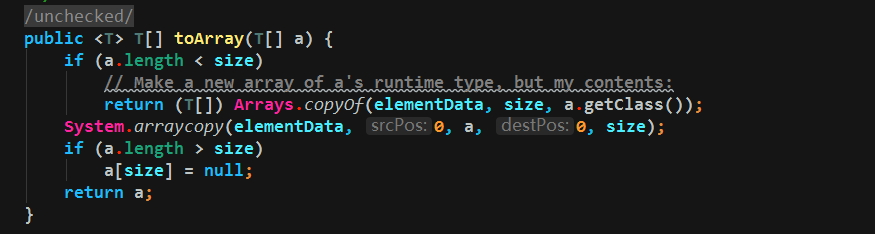
# Java Collection接口的toArry方法注意

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，所以无法向下转型为指定的数组。



即用toArray方法将集合转为数组时返回的是Object[]类对象（数组），**使用时不能对整个数组对象执行强转，只能对数组的单个元素执行强转**。

1. 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。



若传入数组无法容纳目标数组，a.getClass()会获得传入数组的类类型（T[]），并创建新的数组。否则拷贝目标数组到传入数组，如果传入数组容量大于目标数组则在传入数组末尾添加null以标记结束。

# 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

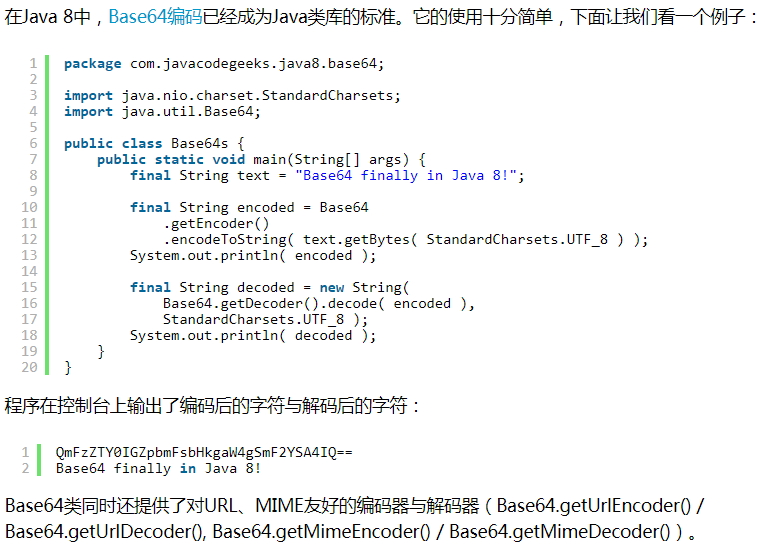
三者只持有ISO-8601格式具有时区信息的日期与时间。前两者都可以从Clock中获得。

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

## 13 Base64加入java.util包



# java值传递

Java使用**对象的引用**完成各种操作，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;

}

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

Object temp = a;

a = b;

b = temp;

}

public void swap(Object[] array,int a,int b) { // 正确的交换方法，交换的是对象的内部值

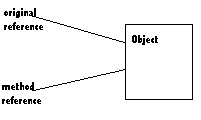
Object temp = array[a];

array[a] = array[b];

array[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。

如：交换数组中元素的位置



# jvm类加载机制

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

# 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,这个参数是状态码，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;

## 类的加载,连接和初始化加载

1. 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**

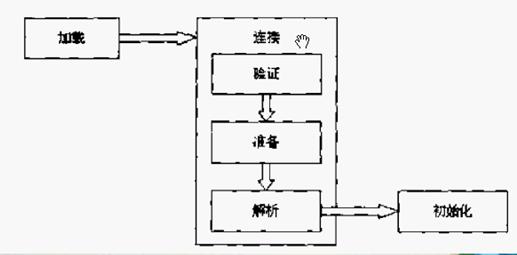
**验证：**确保被加载的类的正确性,使用javac编译工具生成的字节码文件能通过验证,如果不是由javac编译生成的字节码文件,或者生成的字节码文件不符合JVM虚拟机对字节码文件的要求,就会出现验证通不过的情况;比如说随便拿一个文件,将后缀名直接修改为.class,这样的字节码文件肯定不合法;

**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;）



## 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;

1. 主动使用的6种方式:
2. 创建类的实例;
3. 访问某个类或接口的静态变量,或者对静态变量赋值;
4. 调用类的静态方法;
5. 反射:Class.forName(“类名”);
6. 初始化一个类的子类,看做是对父类的主动使用;
7. 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;

## 类的加载

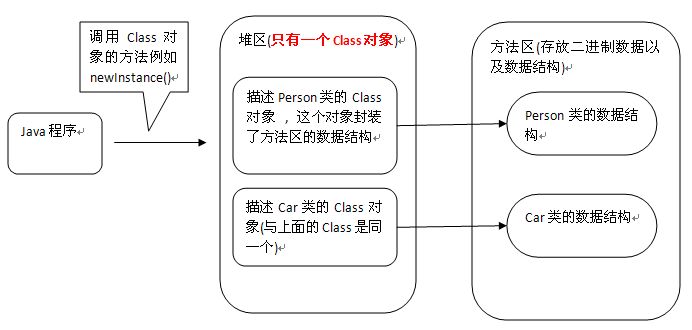
1. 概念

**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**,只有一个。这个类对象是由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);

1. 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;

## 类加载器

1. **Java虚拟机自带的类加载器：**根类加载器(Bootstrap)；
2. 是C++写的,程序员无法在java代码中获取这个类,如果使用getClassLoader()方法获取，获取到的是一个null；（String类是由根类加载器进行加载的）
3. **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。
4. 类加载器并不一定要在某个类被“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;

1. 类的链接

类被加载之后,就进入链接阶段;链接阶段会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;

1. 类的验证：

类文件的结构检查

语义检查

字节码验证

二进制兼容性的验证

1. 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

1. 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

# Thread的yield、sleep、join、interrupt、start

<http://blog.csdn.net/u014290221/article/details/51436710>

## join - 等待线程结束

线程A调用其它线程B的join方法，则线程A执行到该语句时将暂停，等到B线程执行完毕时调用线程才继续往下执行。

Thread th = new Thread(() -> {  
 Thread thread = Thread.*currentThread*();  
 *o*.accept(thread.getName() + " start");  
 try {  
 Thread.*sleep*(3000);  
 *o*.accept(thread.getName() + " finished");  
  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
}, "th#1");  
  
th.start();  
*o*.accept(Thread.*currentThread*().getName() + " start join");  
th.join();  
*o*.accept(Thread.*currentThread*().getName() + " finished join");

输出：

main start join

th#1 start

th#1 finished

main finished join

## sleep wait – 线程睡眠/等待

Thread.sleep：

sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，**将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态**）

Object.wait：

wait()是Object类的方法，调用对象的wait()方法导致**当前线程放弃对象的锁**（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lockpool），如果线程重新获得对象的锁就可以进入就绪状态。

## yield – 线程让步

yield方法和sleep方法有点相似，也是Thread.yield静态方法。yield方法会强制线程从**运行状态转入就绪状态**。

它也可以让当前正在执行的线程暂停，但它不会阻塞该线程，它只是将该线程转入到就绪状态。即让当前线程暂停一下，让系统的线程调度器重新调度一次，完全可能的情况是：当某个线程调用了yield()方法暂停之后，线程调度器又将其调度出来重新执行。**（当线程调用了yield方法后，只有优先级与当前线程相同或者比当前线程高的处于就绪状态的线程才会获得执行机会（抢占cpu执行权））。**

## Interrupt – 中断线程

interrupt()的作用是中断线程

本线程中断自己是被允许的；其它线程调用本线程的interrupt()方法时，会通过checkAccess()检查权限。这有可能抛出SecurityException异常。

如果本线程是处于阻塞状态*（调用线程的wait(), wait(long)或wait(long, int)会让它进入等待(阻塞)状态，或者调用线程的join(), join(long), join(long, int), sleep(long), sleep(long, int)也会让它进入阻塞状态），*此时调用了它的interrupt()方法，那么**它的“中断状态”会被清除并且线程会收到一个InterruptedException异常**。

例如，线程通过wait()进入阻塞状态，此时调用interrupt()中断该线程；调用interrupt()会将线程的中断标记置为“true”，线程会收到一个InterruptedException异常，同时“中断标记”会立即被清除为“false”。（收到InterruptedException异常后，线程就会恢复到运行状态）

中断一个“已终止的线程”不会产生任何操作。

## start – 使线程进入就绪状态

start方法用来启动线程，使其进入就绪状态，接受CPU调度，最终运行。

区别于直接调用run方法，run方法称为线程体，如果直接调用run方法，那么run方法将被当成普通方法执行，不会开启新的线程执行。

# Java线程

## 线程的优先级

线程的优先级用数字表示，范围为1~10，默认为5.

每个线程的默认优先级与创建它的父线程相同。数字越大，优先级越高。

Thread类有三个常量：

MAX\_PRIORITY (10)

MIN\_PRIORITY (1)

NORM\_PRIORITY (5)

Thread thread = new Thread();  
thread.setPriority(Thread.*MAX\_PRIORITY*);

## 线程的终止

Thread.stop方法已经废弃，终止线程的根本方式是**run方法返回**。

Runable的run方法并未抛出异常，所以终止线程的方法就是**run方法返回。**

1. 终止处于“阻塞状态”的线程

通常，我们通过“中断”方式终止处于“阻塞状态”的线程（即调用线程的interrupt方法）。线程由于被调用了sleep(), wait(), join()等方法而进入阻塞状态（这些方法都会抛出InterruptException）；若此时调用线程的interrupt()将线程的中断标记设为true。则线程将产生一个InterruptedException异常（阻塞方法将收到此异常），同时中断标记会被清除（置为false）。

那么只需在收到InterruptException异常时终止run方法执行即可。（run方法返回线程就会结束）。

1. 终止处于“运行状态”的线程

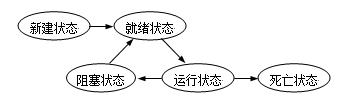
通常通过使用“中断标记”，或添加“额外的结束**标记**”的方式判断线程是否需要结束。

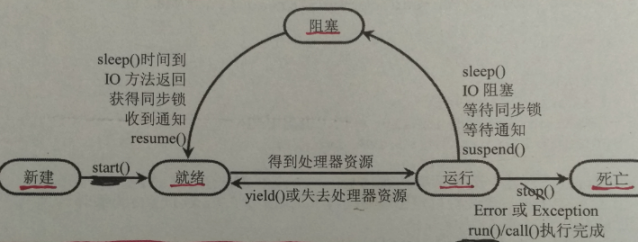
如：调用isInterrupt()判断中断标记是否为true，在需要结束的地方调用线程的interrupt方法。

Thread thread = new Thread(() -> {  
 Thread t = Thread.*currentThread*();  
 while (!t.isInterrupted()) {  
 *o*.accept("a");  
 }  
});  
  
thread.start();  
Thread.*sleep*(3000);  
thread.interrupt();

## 线程的五种状态

线程从创建运行到结束总是处于下面五个状态之一：新建状态、就绪状态、运行状态、阻塞状态及死亡状态。





### 新建状态(New)

当用new操作符创建一个线程时， 例如new Thread(r)，线程还没有开始运行（未调用其start方法），此时线程处在新建状态。 当一个线程处于新生状态时，程序还没有开始运行线程中的代码。

### 就绪状态(Runnable)

一个新创建的线程并不自动开始运行，要执行线程，必须调用线程的start()方法。当线程对象调用start()方法即启动了线程，start()方法创建线程运行的系统资源，并调度线程运行run()方法。当start()方法返回后，线程就处于就绪状态。

处于就绪状态的线程并不一定立即运行run()方法，线程还必须同其他线程竞争CPU时间，只**有获得CPU时间才可以运行线程**。因为在单CPU的计算机系统中，不可能同时运行多个线程，一个时刻仅有一个线程处于运行状态。因此此时可能有多个线程处于就绪状态。对多个处于就绪状态的线程是由Java运行时系统的**线程调度程序**(thread scheduler)来调度的。

### 运行状态(Running)

当线程获得CPU时间后，它才进入运行状态，真正开始执行run()方法

### 阻塞状态(Blocked)

线程运行过程中，可能由于各种原因进入阻塞状态:

1. 线程通过调用**sleep方法**进入睡眠状态；
2. 线程调用一个在**I/O上被阻塞**的操作，即该操作在输入输出操作完成之前不会返回到它的调用者；
3. 线程试图得到一个锁，而该锁正被其他线程持有（**死锁**）。
4. 线程在**等待**某个触发条件；

......

所谓阻塞状态是正在运行的线程**没有运行结束，暂时让出CPU，这时其他处于就绪状态的线程就可以获得CPU时间，进入运行状态。**

### 死亡状态(Dead)

有两个原因会导致线程死亡：

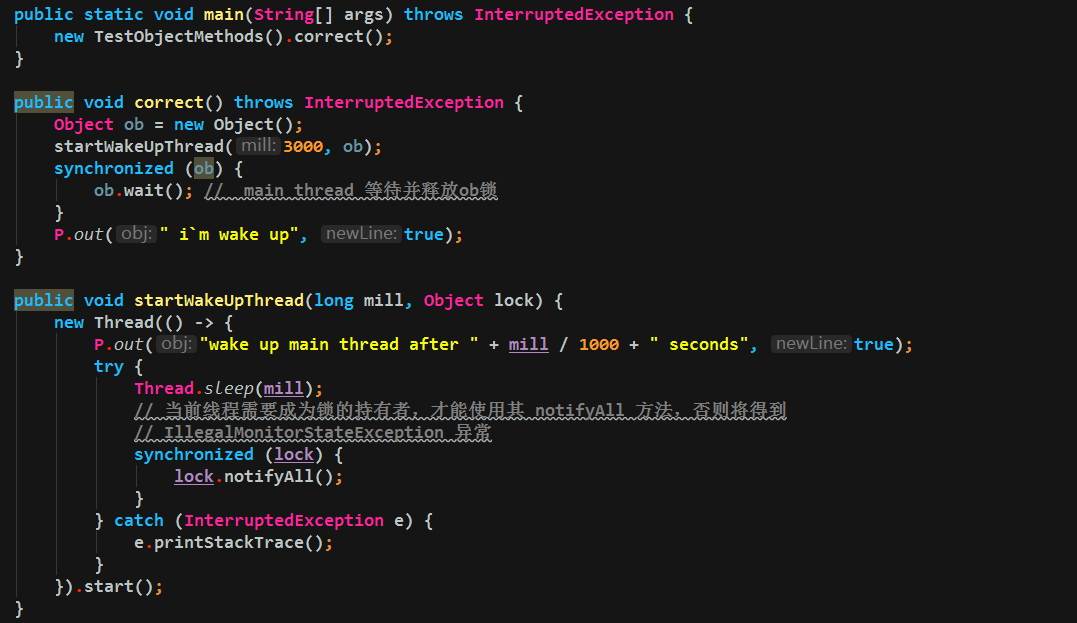
1. run方法正常退出而**自然死亡**
2. 一个**未捕获的异常**终止了run方法而使线程猝死。

为了确定线程在当前是否存活着（就是要么是可运行的，要么是被阻塞了），需要使用**isAlive**方法。如果是可运行或被阻塞，这个方法返回true； 如果线程仍旧是new状态且不是可运行的， 或者线程死亡了，则返回false.

## Object的wait，notify，notifyAll方法

wait()，notify()和notifyAll()方法只能在同步方法或同步代码块里调用。

使用示例：



输出：

06/12/2017 16:27:13-176 wake up main thread after 3 seconds

06/12/2017 16:27:16-179 i`m wake up

### Thread.sleep和wait区别

Thread.sleep：

sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，**将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态，已获得锁，等待CPU调度**）

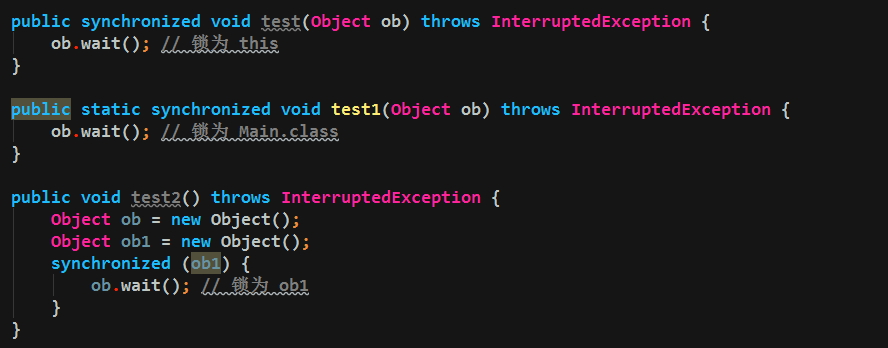
Object.wait：

wait()是Object类的方法，调用对象的wait()方法导致**当前线程放弃对象的锁**（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lockpool），如果线程重新获得对象的锁就可以进入就绪状态。

## IllegalMonitorStateException

如果在非同步控制方法里调用这些方法，程序能通过编译，但运行的时候，将得到IllegalMonitorStateException异常，并伴随着一些含糊的消息，比如"**当前线程不是拥有者**"。消息的意思是，**调用wait()，notify()和notifyAll()的线程在调用这些方法前必须"拥有"对象的锁。**当前的线程不是此对象锁的所有者，却调用该对象的notify()，notify()，wait()方法时抛出该异常。

如下几种情况都会得到该异常：



# 守护（后台）线程

<http://blog.csdn.net/shimiso/article/details/8964414>

在java中有两类线程：用户线程（User Thread），守护线程（Daemon Thread）

任何一个守护线程都是整个JVM中所有非守护线程的保姆。

只要当前JVM实例中尚存在**任何一个非守护线程**没有结束，守护线程就全部工作；只有当最后一个非守护线程结束时，守护线程随着JVM一同结束工作。

Daemon的作用是为其他线程的运行提供便利服务，守护线程最典型的应用就是 GC (垃圾回收器)，它就是一个很称职的守护者。

User和Daemon两者几乎没有区别，唯一的不同之处就在于虚拟机的离开：如果 User Thread已经全部退出运行了，只剩下Daemon Thread存在了，虚拟机也就退出了。 因为没有了被守护者，Daemon也就没有工作可做了，也就没有继续运行程序的必要了。

Thread daemonThread = new Thread();  
Thread thread = new Thread();  
  
// 设定 daemonThread 为 守护线程，default false(非守护线程)  
daemonThread.setDaemon(true);  
  
// 验证当前线程是否为守护线程，返回 true 则为守护线程  
*o*.accept(daemonThread.isDaemon()); // true  
*o*.accept(thread.isDaemon()); // false

注意点：

1. 手动创建守护线程，setDaemon(true)需在start之前设置（否则会抛IllegalThreadStateException）。
2. 在Daemon中启动的线程默认也是daemon的

JRE判断程序是否执行结束的标准是所有的前台线程执行完毕了。

补充说明：

1. **定义**：守护线程--也称“服务线程”，在没有用户线程可服务时会自动离开。
2. **优先级**：守护线程的优先级比较低，用于为系统中的其它对象和线程提供服务。
3. **设置**：通过setDaemon(true)来设置线程为“守护线程”；将一个用户线程设置为守护线程的方式是**在线程start之前 用线程对象的setDaemon方法，并置为true**。
4. **example**: 垃圾回收线程就是一个经典的守护线程，当我们的程序中不再有任何运行的Thread,程序就不会再产生垃圾，垃圾回收器也就无事可做，所以当垃圾回收线程是JVM上仅剩的线程时，垃圾回收线程会自动离开。它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源。
5. **生命周期**：守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。也就是说守护线程不依赖于终端，但是依赖于系统，与系统“同生共死”。

那Java的守护线程是什么样子的呢。当JVM中所有的线程都是守护线程的时候，JVM就可以退出了；如果还有一个或以上的非守护线程则JVM不会退出。

实际应用：消息推送线程。

# java枚举类型

<http://www.cnblogs.com/hyl8218/p/5088287.html>

# 正则表达式

正则表达式定义了字符串的模式。正则表达式可以用来搜索、编辑或处理文本。正则表达式并不仅限于某一种语言，但是在每种语言中有细微的差别。

## Pattern

正则表达式的已编译版本，通过传递正则表达式给静态方法compile来创建Pattern对象。

## Matcher

用来匹配输入字符串和创建的 pattern 对象的正则引擎对象，我们用patten对象的matcher方法，使用输入字符串作为参数来获得一个Matcher对象。然后使用matches方法，通过返回的布尔值判断输入字符串是否与正则匹配。

## PatternSyntaxException

如果正则表达式语法不正确将抛出PatternSyntaxException异常。

## 注意

由正则定义的pattern是从左至右应用的，一旦一个原字符在一次匹配中使用过了，将不会再次使用。例如，正则“121”只会匹配两次字符串“31212142121″，就像这样“\_121\_\_\_\_121″。

## Java正则表达式元字符



## 正则表达式通用匹配符号



## 正则表达式量词



## 正则表达式capturing group

Capturing group是用来对付作为一个整体出现的多个字符。你可以通过使用()来建立一个group。输入字符串中和capturing group相匹配的部分将保存在内存里，并且可以通过使用Backreference调用。

可以使用matcher.groupCount方法来获得一个正则pattern中capturing groups的数目。例如((a)(bc))包含3个capturing groups; ((a)(bc)), (a) 和 (bc)。

System.out.println(Pattern.matches("(\\w\\d)\\1", "a2a2")); //true

System.out.println(Pattern.matches("(\\w\\d)\\1", "a2b2")); //false

System.out.println(Pattern.matches("(AB)(B\\d)\\2\\1", "ABB2B2AB")); //true

System.out.println(Pattern.matches("(AB)(B\\d)\\2\\1", "ABB2B3AB")); //false

在第一个例子里，运行的时候第一个capturing group是(\w\d)，在和输入字符串“a2a2″匹配的时候获取“a2″并保存到内存里。因此\1是”a2”的引用，并且返回true。基于相同的原因，第二行代码打印false。

## 常用方法

Matcher： find(), group(), start() end() replaceAll() replaceFirst()

Pattern：compile() split()

1. 正则表达式总是和字符串有关， Java 1.4对String类进行了扩展，提供了一个matches方法来匹配pattern。
2. 可以创建一个带有标志的Pattern对象。例如Pattern.CASE\_INSENSITIVE可以进行大小写不敏感的匹配。Pattern类同样提供了和String类相似的split(String) 方法
3. Pattern类toString()方法返回被编译成这个pattern的正则表达式字符串。如：

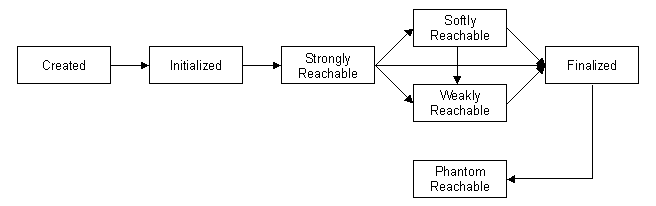
Pattern pattern = Pattern.compile("(AB)(B\\d)[\\2\\1](file:///\\2\\1)");

Pattern.toString()：(AB)(B\d)\2\1

1. Matcher类有start()和end()索引方法，他们可以显示从输入字符串中匹配到的准确位置。
2. Matcher类同样提供了字符串操作方法replaceAll(String replacement)和replaceFirst(String replacement)。

# java四种引用类型

<http://blog.csdn.net/panyongcsd/article/details/46605613>



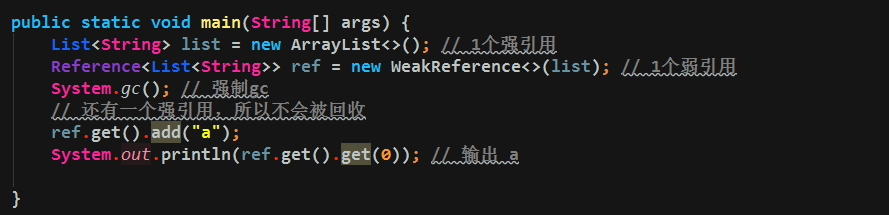
## 强引用

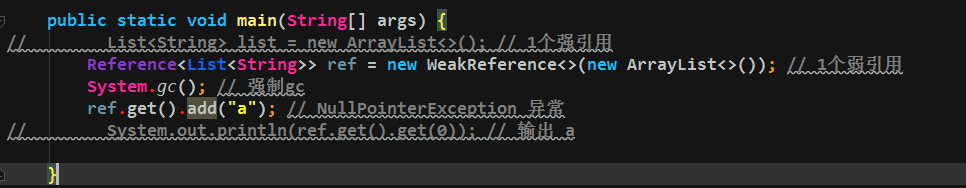
用new关键字创建的引用就属于强引用，如果一个对象属于强引用，**垃圾回收器就绝不会回收它，**当内存不足时，JVM宁愿抛出OutOfMemoryError，也不会回收它们。更不会在GC的时候回收具有强引用的对象。

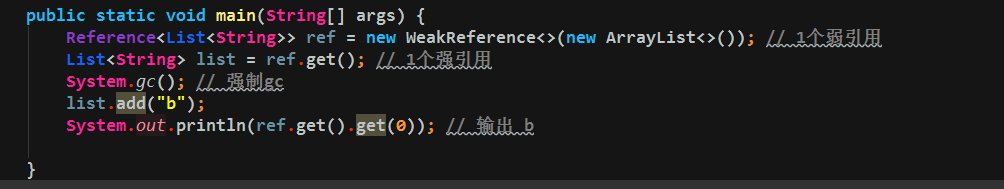
## 软引用 SoftReference

**如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。**只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。 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，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。

垃圾回收器会尽可能长时间的保留该对象，直到要抛出OutOfMemeoryError以前才会进行回收。







## 弱引用 WeakReference

弱引用与软引用的区别在于：只具有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。**不过，由于垃圾回收器是一个优先级很低的线程（守护线程）， 因此不一定会很快发现那些只具有弱引用的对象。 弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

## 虚引用 PhantomReference

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。**

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。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。程序如果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。

往往我们不会再去访问这个对象，仅仅是当做一个该对象被回收的信号使用，垃圾回收器可以随时回收它。

# 线程池

## 好处

1. 重用线程中的线程，减少线程创建、销毁的开销。
2. 控制线程的最大并发数，避免大量线程互相抢夺系统资源而阻塞。
3. 能够对线程进行简单管理，提供定时执行及间隔循环执行的功能。

## 线程池中线程执行规则

1. 若线程池中线程数小余核心线程数，会直接启动核心线程执行任务。
2. 若线程数已达到或超过核心线程数，那么任务会被**插入到任务队列中排队**。
3. 如果（2）无法插入任务（这往往是因为**任务队列已满**），那么此时若线程数未达到maximumPoolSize，那么会启动一个非核心线程执行任务。
4. 如果（3）中线程数到达maximumPoolSize，那么就会拒绝执行任务，ThreadPoolExecutor会调用RejectExceptionHandler通知调用者。

## 关键参数

继承结构：

ThreadPoolExecutor implements Executor

### corePoolSize

线程池的核心线程数，默认情况下，核心线程会在线程池中一直存活，即使处于闲置状态；若将allowCoreThreadTimeOut置为true（调用allowCoreThreadTimeout(boolean)），那么核心线程在等待新任务时会有超时策略，该事件由keepAliveTime决定，等待时间超过该时间时核心线程就会终止。

### maximumPoolSize

线程池能容纳的最大线程数，当活动线程数达到这个数值后，后续的线程将会阻塞。

### keepAliveTime

非核心线程闲置时的超时时长，超过该时间，非核心线程就会被回收，当置allowCoreThreadTimeOut为true时，该时间将会作用与核心线程。

### unit（TimeUnit）

keepALiveTime的时间单位，枚举：TimeUnit.HOURS、TimeUnit.NANOSECONDS …

### workQueue（BlockingQueue）

线程池中的任务队列，向Executor中提交的Runnable对象保存于此。

### ThreadFactory

线程创建工程，为线程池提供统一的线程创建方法。newThread(Runnable r)

### RejectExceptionHandler

当线程池无法执行任务（这可能是任务队列已满且达到了maximumPoolSize或线程执行出错）

## 四种常见线程池-阻塞队列

这四个类都可通过Executors类创建。

### FixedThreadPool

Executors.newFixedThreadPool(13)

只有核心线程，不会被回收，任务队列无限大。

能够快速的响应外界请求。

new ThreadPoolExecutor(13,13,0L,TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>());

**LinkedBlockingQueue**：阻塞的线程安全队列 – FIFO - 生产者/消费者

可以指定容量，也可以不指定，不指定的话，默认最大是Integer.MAX\_VALUE，其中主要用到**put**和**take**方法，put方法在队列满的时候会阻塞直到有队列成员被消费，take方法在队列空的时候会阻塞，直到有队列成员被放进来。

### CacheThreadPool

Executors.newCachedThreadPool()

只有非核心线程，线程数量无限大，60s超时时长，任务队列“不能”装任务（一put就立即take）

任何任务都会立即执行，适合大量耗时较少的任务。

new ThreadPoolExecutor(0,Integer.MAX\_VALUE,60,TimeUnit.SECONDS,new SynchronousQueue<Runnable>())

**SynchronousQueue**：

此队列设计的理念类似于"**单工模式**",对于每个put/offer操作,必须等待一个take/poll操作。因为这种策略,最终导致队列中并没有一个真正的元素。

### ScheduledThreadPool

Executors.newScheduledThreadPool(13)

核心线程数固定，非核心线程数无限制，非核心线程闲置时会立即被回收。

用于执行定时任务和具有固定周期的任务。

new ThreadPoolExecutor(13,Integer.MAX\_VALUE,0L,TimeUnit. NANOSECONDS,new DeloyedWorkQueue())

**DeloyedWorkQueue**：<http://ideasforjava.iteye.com/blog/657384>

其中的对象只能在其到期时才能从队列中取走。这种队列是有序的，即队头对象的延迟到期时间最长。注意：不能将null元素放置到这种队列中。

### SingleThreadPool

Executors.newSingleThreadExecutor()

maximumPoolSize为1，即为唯一的一个核心线程，该线程不会被回收。

确保所有的任务都在同一个线程中顺序执行，任务间无需处理线程同步。

new ThreadPoolExecutor(1,1,0L,TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>())

## put/take（阻塞）、offer/poll（false/null）、add/remove(exception)

1. 向队列中**添加**元素的方法有：put，add，offer

**put**: 若向队尾添加元素的时候发现队列已经满了会发生**阻塞**一直等待空间，以加入元素。

**add**:若超出了度列的长度会直接**抛出异常**，否则直接将元素插入队尾。

**offer**:若队列已满，会返回false。否则直接将元素插入队尾。

1. 从队列中**取出**（同时移除）的方法有：poll，remove，take

**take**:若队列为空，发生**阻塞**，等待直到有元素。

**remove**:若队列为空，**抛出NoSuchElementException异常**。

**poll**: 若队列为空，立即返回null。

# 创建线程的三种方式

## 1 继承Thread类创建线程

extends Thread并覆写run方法。

## 2 实现Runnable接口

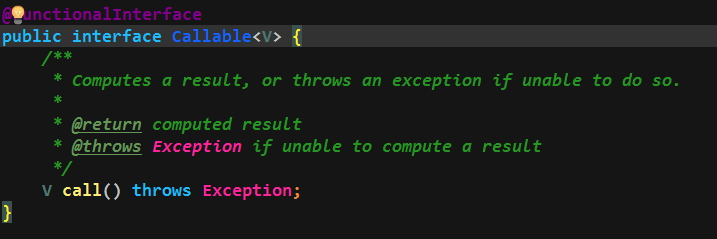
implements Runnable 并覆写run方法即可。

## 3 实现Callable接口通过FutureTask包装器来创建Thread线程

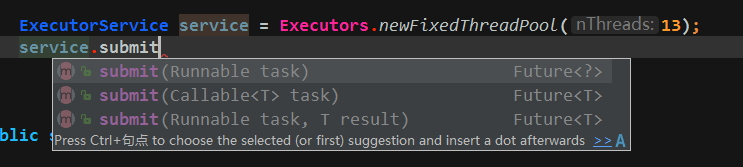
上面的两种方法有一个缺点：线程运行是与其创建者独立的，线程没有返回结果（类似于为了“副作用”而调用的方法）。

从JAVA SE 5.0开始引入了Callable和Future，通过它们构建的线程，在任务执行完成后就可以获取执行结果。

### Callbale接口

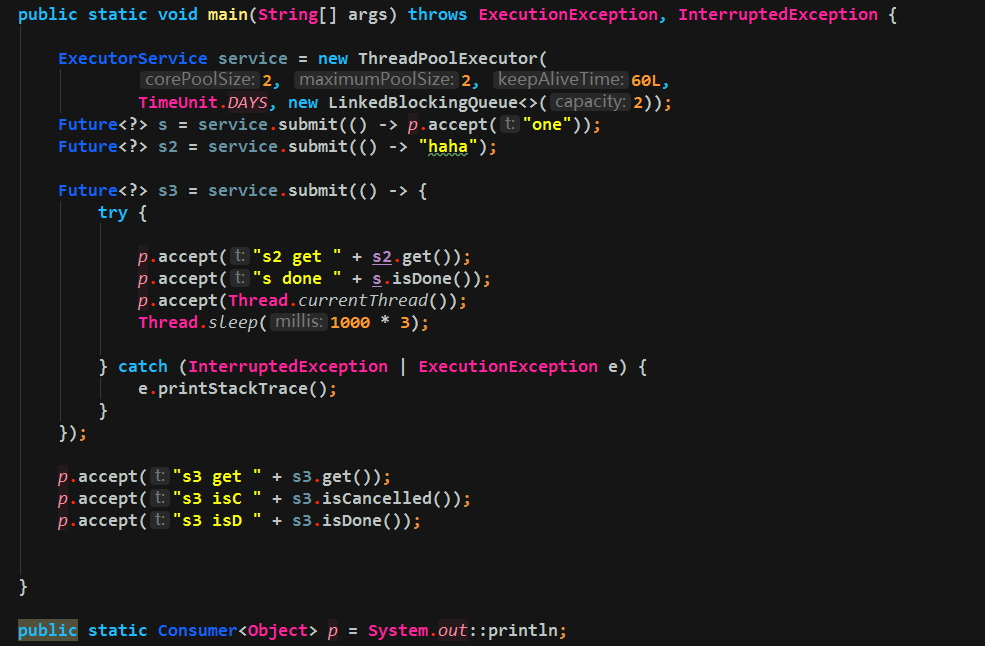


无论是Runnable接口的实现类还是Callable接口的实现类，都可以被ThreadPoolExecutor执行，ThreadPoolExecutor实现了ExcutorService接口，而因此Callable需要和Executor框架中的ExcutorService结合使用，我们先看看ExecutorService提供的方法：



submit方法将任务提交到线程池，通过返回的Future对象可获得异步执行结果或查看当前任务的执行状态。

### Future<T>接口



输出：

s2 get haha

s done false

Thread[pool-1-thread-2,5,main]

One

//---------------阻塞（get方法没有结果可返回，因而会**阻塞线程**）

s3 get null

s3 isC false

s3 isD true

1. **V get()** ：获取异步执行的结果，如果没有结果可用，此方法会**阻塞直到异步计算完成**。
2. **V get(Long timeout , TimeUnit unit)** ：获取异步执行结果，如果没有结果可用，此方法会阻塞，但是会有**时间限制**，如果阻塞时间超过设定的timeout时间，该方法将抛出异常。
3. **boolean isDone()** ：如果任务执行结束，无论是正常结束或是中途取消还是发生异常，都返回true。
4. **boolean isCanceller()** ：如果任务**完成前被取消**，则返回true。
5. **boolean cancel(boolean mayInterruptRunning)** ：

1如果任务**还没开始**，执行cancel(...)方法将返回false；

2如果任务**已经启动**，执行cancel(**true**)方法将以中断执行此任务线程的方式来试图停止任务，如果停止成功，返回true；

3当任务**已经启动**，执行cancel(**false**)方法将不会对正在执行的任务线程产生影响(让线程正常执行到完成)，此时返回false；

4当任务**已经完成**，执行cancel(...)方法将返回false。

由以上分析可知Future提供了3种功能

（1）能够中断执行中的任务

（2）判断任务是否执行完成

（3）获取任务执行完成后的结果。

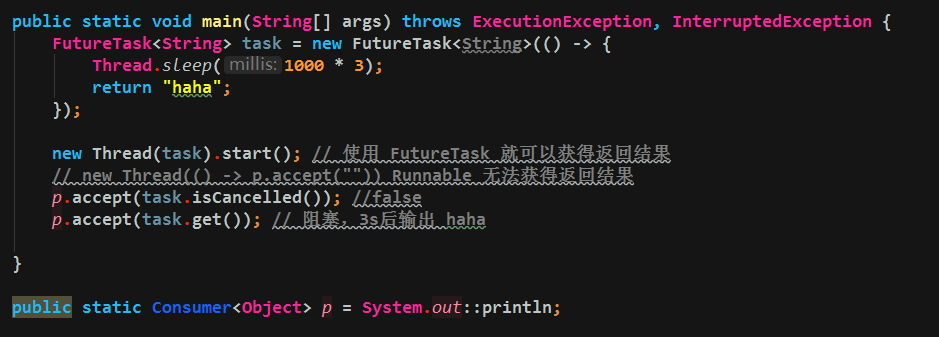
但是我们必须明白Future只是一个接口，我们无法直接创建对象，因此就需要其实现类FutureTask。

### FutureTask<V>

public class **FutureTask**<V> implements **RunnableFuture**<V> {

public interface **RunnableFuture**<V> extends **Runnable**, **Future**<V> {

FutureTask除了实现了Future接口外还实现了Runnable接口，因此FutureTask也可以直接提交给Executor执行。当然也可以作为参数创建Thread并运行线程（**Thread.start()**，如果直接调用FutureTask的run方法，实际上是直接调用Runnable的run，只是普通方法调用，并未开启线程）。



## 联合使用ExecutorService、Callable、Future



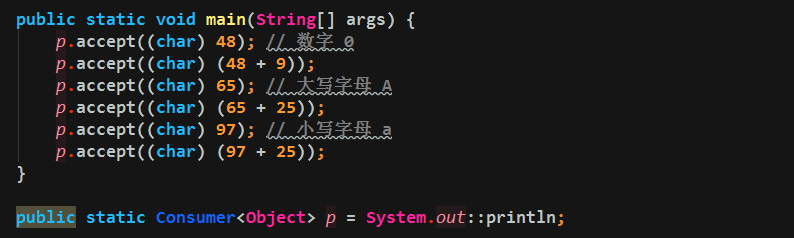
# 常用ASCI

<http://ascii.911cha.com/>

数字：0-9 **48~57**

大写字母：A-Z **65~90**

小写字母：a-z **97~122**



**输出：**

**0**

**9**

**A**

**Z**

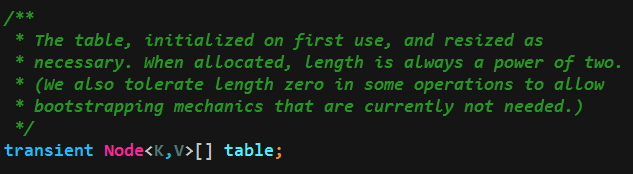
**a**

**z**

# HashMap

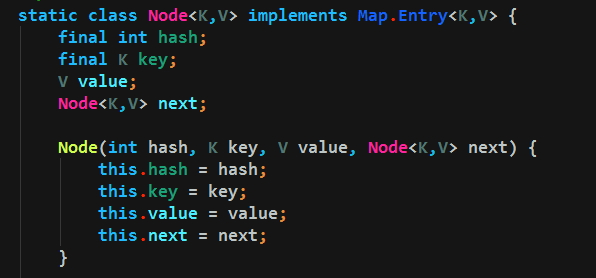
<http://mp.weixin.qq.com/s?__biz=MzAxOTc0NzExNg==&mid=2665514069&idx=1&sn=2996d864bbe596d0af763fba3d244fa7&chksm=80d67c16b7a1f500ec6a191eb4a0beac0e95dbd5a7bf8ee01f5ed2cb17960b9ab32c0b965949&mpshare=1&scene=23&srcid=1201V2GnmuBvSQ8zSCHJ20jV#rd>

HashMap是一个用于存储Key-Value键值对的集合，每一个键值对也叫做Entry。这些个键值对（Entry）分散存储在一个数组当中，这个数组就是HashMap的主干。



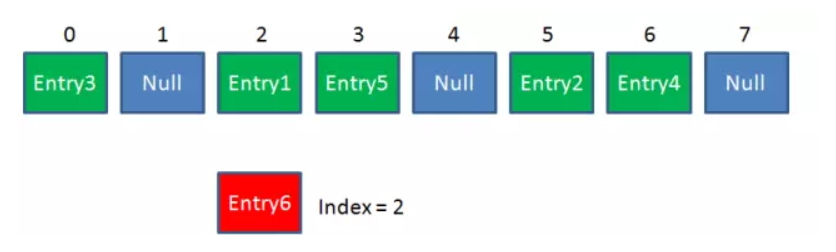
第一次使用的时候初始化，必要时进行扩充。

数组元素同时为一个链表结构（链表的头结点）：



## Hash冲突（put方法）

因为HashMap的长度在一段时间内是有限的，当插入的Entry越来越多时，再完美的Hash函数也难免会出现index冲突的情况。

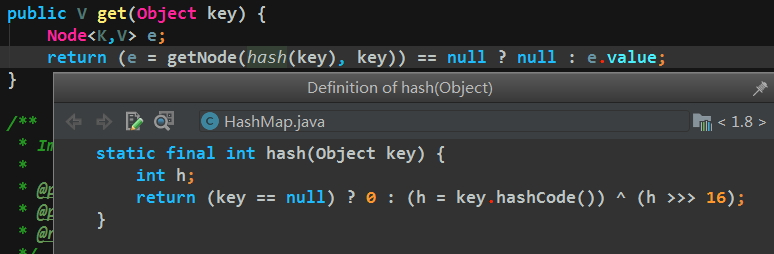


此时使用链表解决冲突：新来的Entry节点插入链表时，使用的是“**头插法**”。

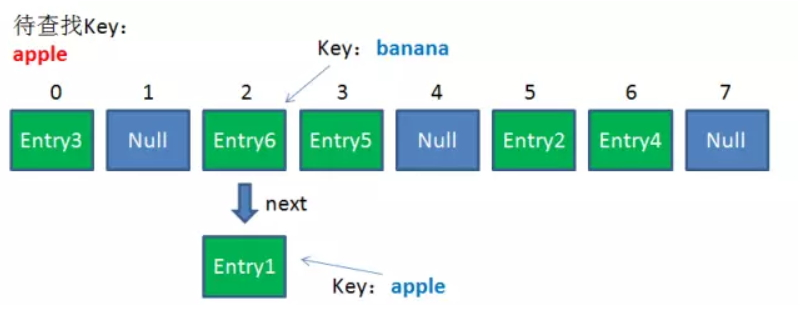


## Get方法

首先会把输入的Key做一次Hash映射，得到对应的index：



由于Hash冲突，同一个位置有可能匹配到多个Entry，这时候就需要顺着对应链表的头节点，一个一个向下来查找。假设我们要查找的Key是“apple”：



之所以把Entry6放在头节点，是因为HashMap的发明者认为，后插入的Entry被查找的可能性更大。

## table初始长度16，hash函数，长度减一&散列值求得下标

**默认长度是16**，并且每次自动扩展或手动初始化时长度必须是2的幂。

之所以选择16，是为了服务Key到index的hash算法：

如下为Java 8 源码，与java7不同



实现一个尽量均匀分布，且使冲突尽量少。

<https://www.zhihu.com/question/20733617/answer/111577937>

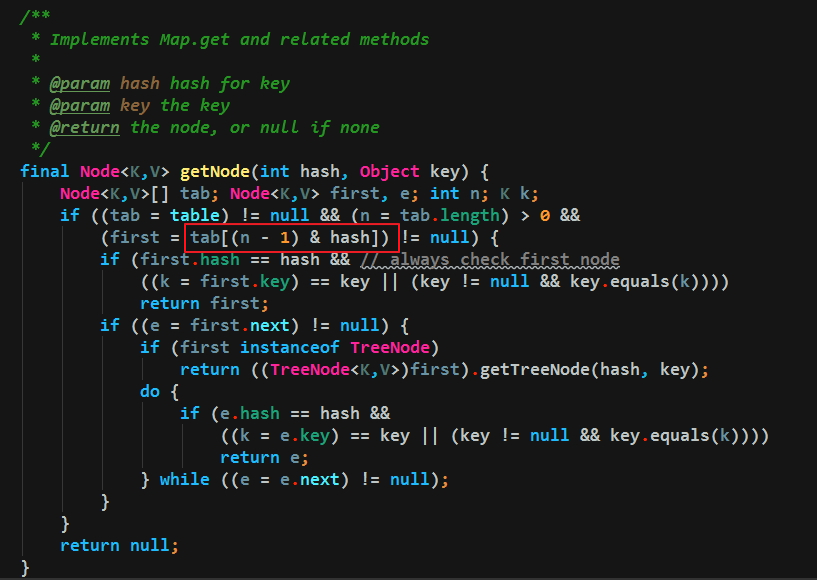
**散列值计算：**

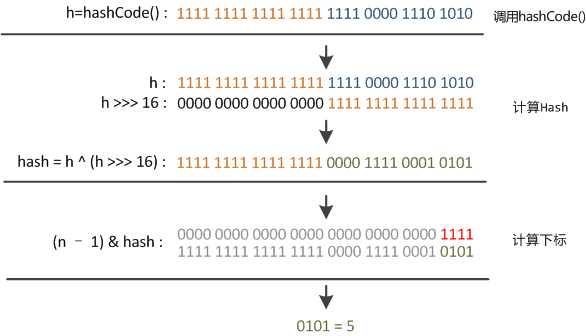
“扰动函数”：**hash = (h = key.hashCode()) ^ (h >>> 16)**

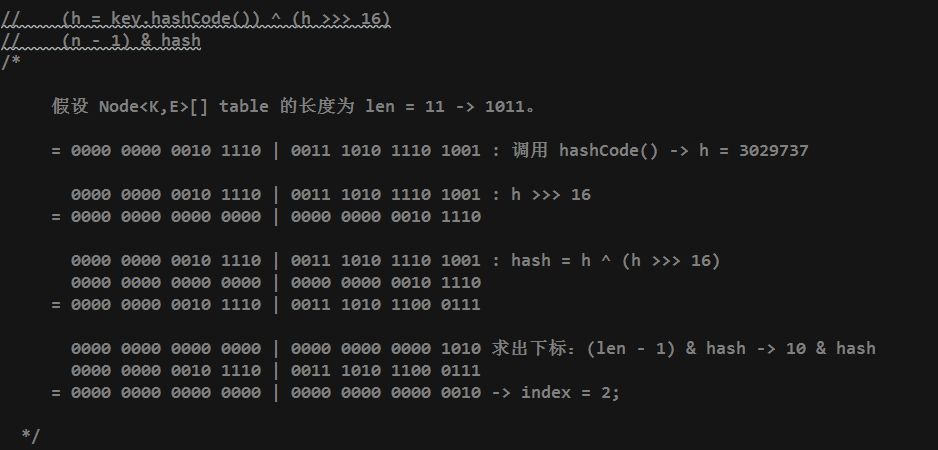
**下标计算：**

**hash &(length - 1)**

散列值取模







# 为什么 Java 中要使用 Checked Exceptions

<http://www.iteye.com/topic/2038>

# Stackoverflowerror和outofmemoryerror

<http://blog.csdn.net/chenchaofuck1/article/details/51144223>

## stackOverflower

java程序启动一个新的线程时，JVM会为其分配一个栈，java栈以帧为单位保持线程运行状态，当线程调用一个方法是，jvm压入一个新的栈帧到这个线程的栈中，只要这个方法还没返回，这个栈帧就存在。 如果方法的嵌套调用层次太多(如递归调用),随着java栈中的帧的增多，最终导致这个线程的栈中的所有栈帧的大小的总和大于-Xss设置的值，而产生StackOverflowError溢出异常。

## Outofmemory

JVM抛出OutOfMemoryError前，会尝试进行一次Full GC，如果GC后可用内存还是不足，才会抛出OutOfMemoryError。因此，这时程序猿必然无法主动处理这一问题，只能等程序崩溃后再去查证原因。

1. 栈内存溢出

java程序启动一个新线程时，没有足够的空间为该线程分配java栈，一个线程java栈的大小由-Xss设置决定；JVM则抛出OutOfMemoryError异常。

1. 堆内存溢出

java堆用于存放对象的实例，当需要为对象的实例分配内存时，而堆的占用已经达到了设置的最大值(通过-Xmx)设置最大值，则抛出OutOfMemoryError异常。

1. 方法区内存溢出

方法区用于存放java类的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。在类加载器加载class文件到内存中的时候，JVM会提取其中的类信息，并将这些类信息放到方法区中。

当需要存储这些类信息，而方法区的内存占用又已经达到最大值（通过-XX:MaxPermSize）；将会抛出OutOfMemoryError异常对于这种情况的测试，基本的思路是运行时产生大量的类去填满方法区，直到溢出。这里需要借助CGLib直接操作字节码运行时，生成了大量的动态类。

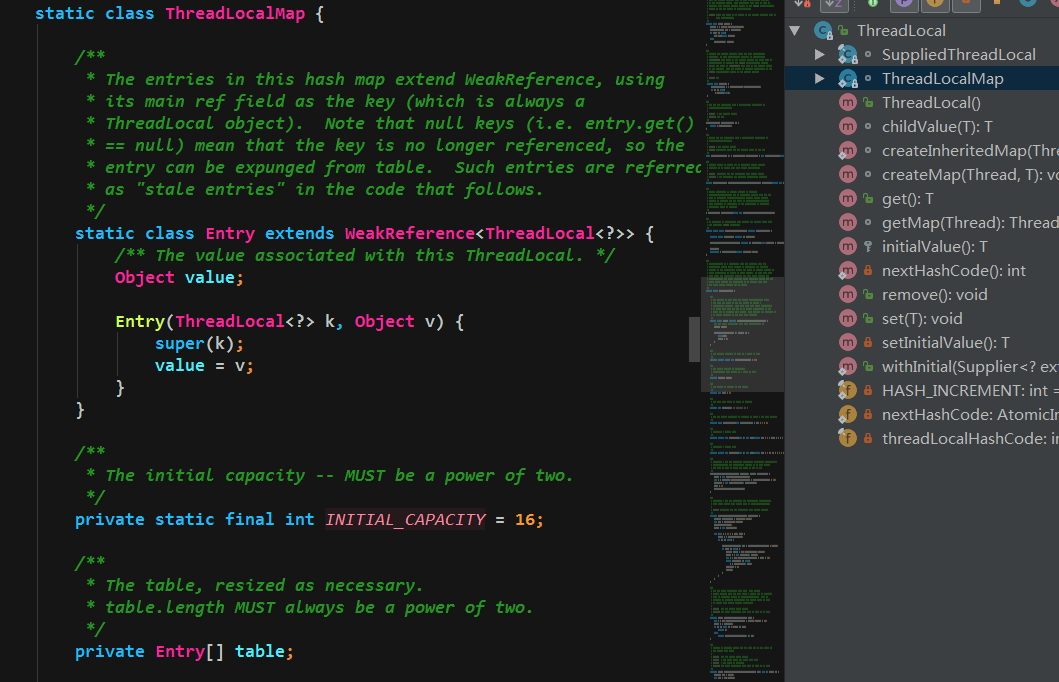
# ThreadLocal

<http://mp.weixin.qq.com/s?__biz=MjM5NzMyMjAwMA==&mid=2651479529&idx=2&sn=956b7870b4b4ddda694ddb9dd45e4115&chksm=bd2531968a52b880c62882188504e2f69fbc93765a6359988de2861774a6fe6df9b0ee941ec9&mpshare=1&scene=23&srcid=1201lPTSxvajWkRcQCP5vKmp#rd>

ThreadLocal是一个为线程提供**线程局部变量**的工具类。它的思想也十分简单，就是为线程提供一个线程私有的变量副本，这样多个线程都可以随意更改自己线程局部的变量，不会影响到其他线程。不过需要注意的是，ThreadLocal提供的只是一个**浅拷贝**，如果变量是一个引用类型，那么就要考虑它内部的状态是否会被改变，想要解决这个问题可以通过重写ThreadLocal的**initialValue**()函数来自己实现深拷贝，建议在使用ThreadLocal时一开始就重写该函数。

ThreadLocal与像synchronized这样的锁机制是不同的。首先，它们的应用场景与实现思路就不一样，**锁更强调的是如何同步多个线程去正确地共享一个变量**，ThreadLocal则是为了解决**同一个变量如何不被多个线程共享**。从性能开销的角度上来讲，如果锁机制是用时间换空间的话，那么ThreadLocal就是用空间换时间。

ThreadLocal中含有一个叫做ThreadLocalMap的内部类，该类为一个采用线性探测法实现的HashMap。它的**key为ThreadLocal对象**而且还使用了WeakReference，ThreadLocalMap正是用来存储变量副本的。



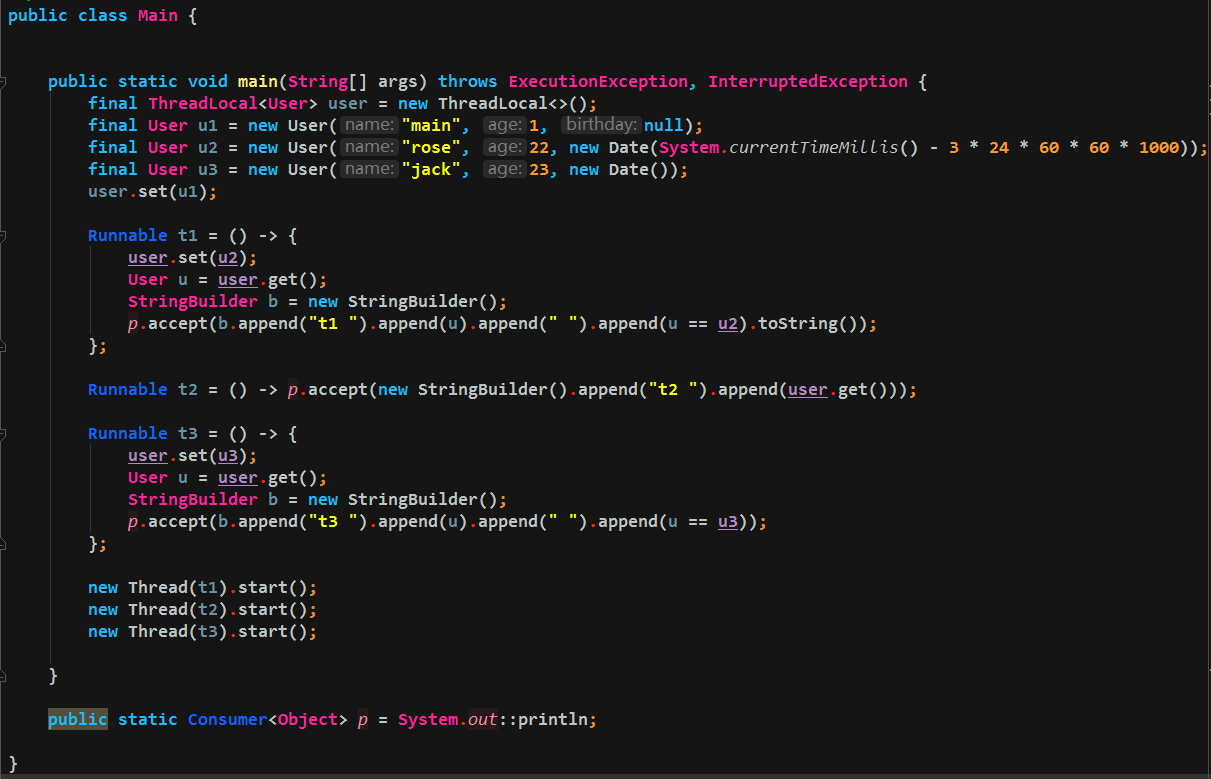
## 用法

输出：

t2 null

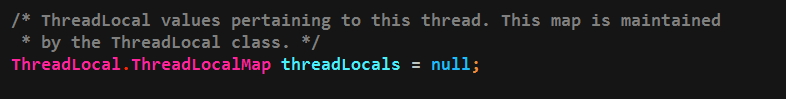
t1 User(name=rose, age=22, birthday=Fri Dec 01 11:22:40 CST 2017) true

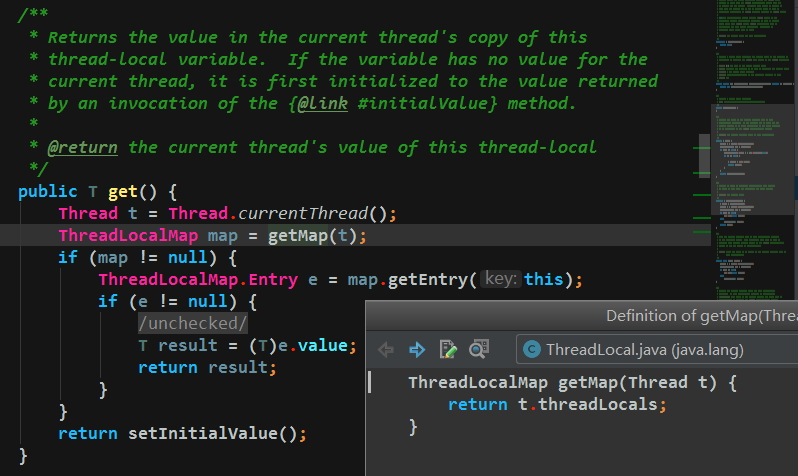
t3 User(name=jack, age=23, birthday=Mon Dec 04 11:22:40 CST 2017) true



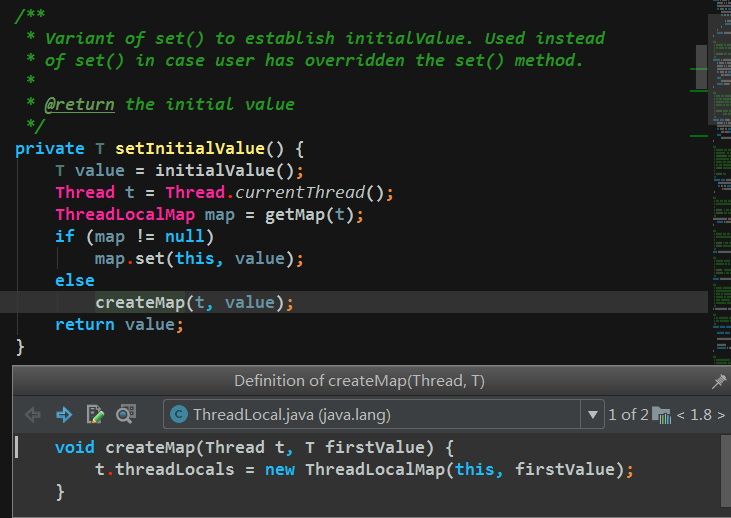
## 原理

要获得当前**线程私有的变量**副本需要调用get()函数。首先，它会调用getMap()函数去获得当前**线程的ThreadLocalMap**，这个函数需要接收当前线程的实例作为参数。如果得到的ThreadLocalMap为null，那么就去调用setInitialValue()函数来进行初始化，如果不为null，就通过map来获得变量副本并返回。

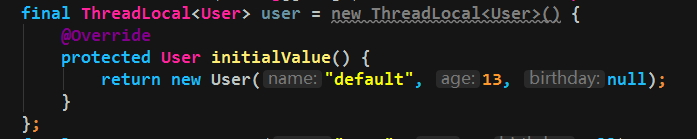




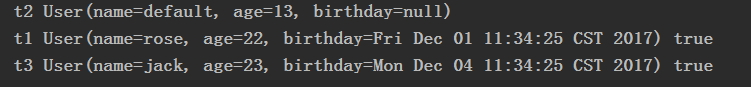
setInitialValue()函数会去先调用**initialValue**()函数来生成初始值，该函数**默认返回null**，我们可以通过重写这个函数来返回我们想要在ThreadLocal中维护的变量。之后，去调用getMap()函数获得ThreadLocalMap，如果该map已经存在，那么就用新获得value去覆盖旧值，否则就调用createMap()函数来创建新的map。



## 覆写initialValue方法

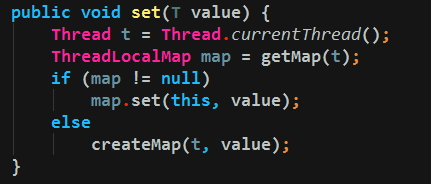
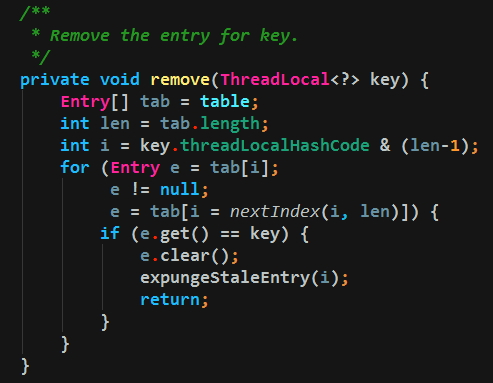


仍然使用上面的例子，输出为：



## Set和remove方法

ThreadLocal的set()与remove()函数要比get()的实现还要简单，都只是通过getMap()来获得ThreadLocalMap然后对其进行操作。

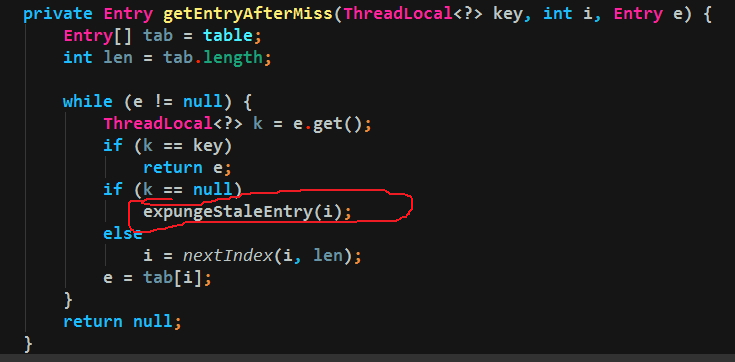


## ThreadLocalMap

有一种普遍的方法是通过一个全局的线程安全的Map来存储各个线程的变量副本，但是这种做法已经完全违背了ThreadLocal的本意，设计ThreadLocal的初衷就是为了避免多个线程去并发访问同一个对象，尽管它是线程安全的。而在每个Thread中存放与它关联的ThreadLocalMap是完全符合ThreadLocal的思想的，当想要对线程局部变量进行操作时，只需要把Thread作为key来获得Thread中的ThreadLocalMap即可。**这种设计相比采用一个全局Map的方法会多占用很多内存空间，但也因此不需要额外的采取锁等线程同步方法而节省了时间上的消耗。（空间换时间）**

## ThreadLocal中的内存泄漏及解决

如果**ThreadLocal被设置为null后**，而且没有任何强引用指向它，根据垃圾回收的可达性分析算法，ThreadLocal将会被回收。这样一来，ThreadLocalMap中就会含有key为null的Entry，而且ThreadLocalMap是在Thread中的，只要线程迟迟不结束，这些**无法访问到的value会形成内存泄漏**。为了解决这个问题，ThreadLocalMap中的getEntry()、set()和remove()函数都会清理key为null的Entry。



**强引用key：**ThreadLocal被设置为null，由于ThreadLocalMap持有ThreadLocal的强引用（有一个强引用），如果不手动删除，那么ThreadLocal将不会回收，产生内存泄漏。

**弱引用key：**ThreadLocal被设置为null，由于ThreadLocalMap持有ThreadLocal（只剩弱引用）的弱引用，即便不手动删除，ThreadLocal仍会被回收，ThreadLocalMap在之后调用set()、getEntry()和remove()函数时会清除所有key为null的Entry。

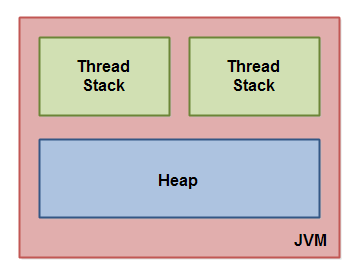
但要注意的是，ThreadLocalMap仅仅含有这些被动措施来补救内存泄漏问题。如果你在之后没有调用ThreadLocalMap的set()、getEntry()和remove()函数的话，那么仍然会存在内存泄漏问题。

在使用线程池的情况下，如果不及时进行清理，内存泄漏问题事小，甚至还会产生程序逻辑上的问题。所以，为了安全地使用ThreadLocal，必须要像每次使用完锁就解锁一样，在每次使用完ThreadLocal后都要调用remove()来清理无用的Entry。

# Java内存模型

## Java内存模型内部原理

<http://www.importnew.com/19612.html>

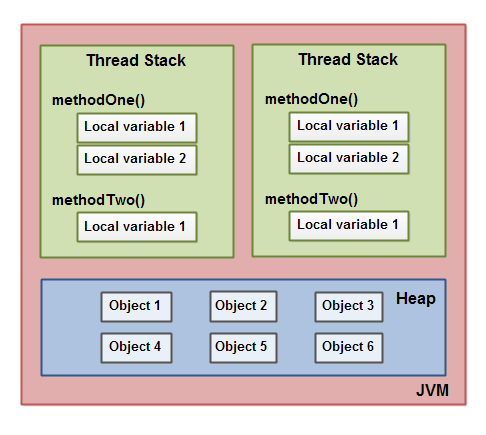


每一个运行在Java虚拟机里的**线程都拥有自己的线程栈**。

这个线程栈包含了这个线程调用的方法当前执行点相关的信息。一个线程仅能访问自己的线程栈。**一个线程创建的本地变量（局部变量）对其它线程不可见，仅自己可见。**即使两个线程执行同样的代码，这两个线程任然在在自己的线程栈中的代码来创建本地变量。因此，每个线程拥有每个本地变量的独有版本。

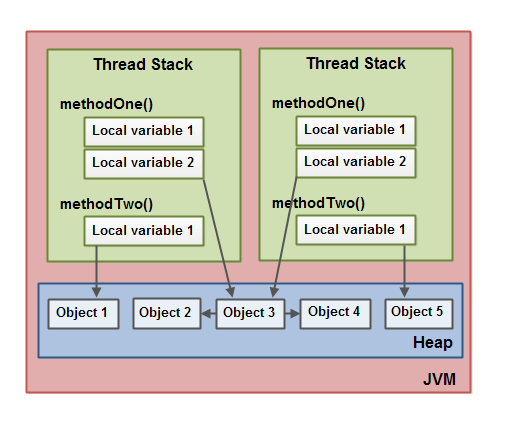
**所有原始类型的本地变量都存放在线程栈上**，因此对其它线程不可见。一个线程可能向另一个线程传递一个原始类型变量的拷贝，但是它不能共享这个原始类型变量自身。

堆上包含在Java程序中创建的**所有对象**，无论是哪一个对象创建的。这包括原始类型的包装类。如果一个对象被创建然后赋值给一个局部变量，或者用来作为另一个对象的成员变量，这个对象任然是存放在堆上。



一个本地变量也可能是指向一个对象的一个**引用**。在这种情况下，**引用（这个本地变量）存放在线程栈上，但是对象本身存放在堆上。**

**静态成员变量跟随着类定义一起也存放在堆上**



两个线程拥有一些本地变量。其中一个本地变量（Local Variable 2）执行堆上的一个共享对象（Object 3）。**这两个线程分别拥有同一个对象的不同引用**。这些引用都是本地变量，因此存放在各自线程的线程栈上。这两个不同的引用**指向堆上同一个对象**。

注意，这个共享对象（Object 3）持有Object2和Object4一个引用作为其成员变量（如图中Object3指向Object2和Object4的箭头）。通过在Object3中这些成员变量引用，这两个线程就可以访问Object2和Object4。

这张图也展示了**指向堆上两个不同对象的一个本地变量**。在这种情况下，指向两个不同对象的引用不是同一个对象。理论上，两个线程都可以访问Object1和Object5，如果两个线程都拥有两个对象的引用。但是在上图中，每一个线程仅有一个引用指向两个对象其中之一。

## 硬件内存架构



**一个现代计算机通常由两个或者多个CPU。其中一些CPU还有多核。**从这一点可以看出，在一个有两个或者多个CPU的现代计算机上同时运行多个线程是可能的。每个CPU在某一时刻运行一个线程是没有问题的。这意味着，如果你的Java程序是多线程的，在你的Java程序中每个CPU上一个线程可能同时（**并发**）执行。

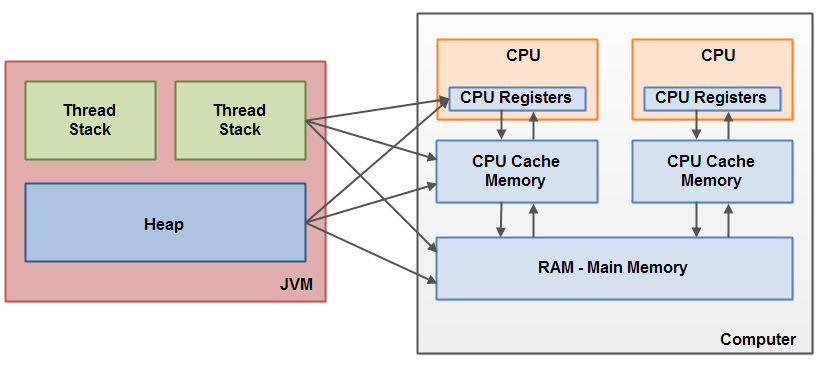
**每个CPU都包含一系列的寄存器**，它们是CPU内内存的基础。**CPU在寄存器上执行操作的速度远大于在主存上执行的速度。**这是因为CPU访问寄存器的速度远大于主存。

**每个CPU可能还有一个CPU缓存层。**实际上，绝大多数的现代CPU都有一定大小的缓存层。

通常情况下，当一个CPU需要读取主存时，它会**将主存的部分读到CPU缓存中**。它甚至可能将缓存中的部分内容读到它的内部寄存器中，然后在寄存器中执行操作。当CPU需要将结果写回到主存中去时，它会将内部寄存器的值刷新到缓存中，然后在**某个时间**点将值刷新回主存。

## Java内存模型和硬件内存架构之间的桥接

Java内存模型与硬件内存架构之间存在差异。硬件内存架构没有区分线程栈和堆。对于硬件，所有的线程栈和堆都分布在主存中。部分线程栈和堆可能有时候会出现在CPU缓存中和CPU内部的寄存器中。



**共享对象可见性**

如果两个或者更多的线程在没有正确的使用volatile声明或者同步的情况下共享一个对象，一个线程更新这个共享对象可能对其它线程来说是不可见的。

共享对象被初始化在主存中。跑在CPU上的一个线程将这个共享对象读到**CPU缓存**中。然后修改了这个对象。只要**CPU缓存没有被刷新回主存**，对象修改后的版本对跑在其它CPU上的线程都是**不可见**的。这种方式可能导致每个线程拥有这个共享对象的私有拷贝，每个拷贝停留在不同的CPU缓存中。

## 可见性和原子性

<http://blog.csdn.net/guyuealian/article/details/52525724>

1. **可见性**：一个线程对共享变量的修改，能够及时的被其他线程看到
2. **原子性**：即不可再分，不能再细分为多步操作。

比如赋值或者return。比如"a = 1;"和 "return a;"这样的操作都具有原子性。

类似"a += b"这样的操作不具有原子性，在某些JVM中"a += b"可能要经过这样三个步骤：

1. 取出a和b
2. 计算a+b
3. 将计算结果写入内存

如：

Singleton模式的DCL失效问题：new不是一个原子操作，指令重排序的问题。

<http://www.jianshu.com/p/83f7958b0944>

instance = new Singleton(); //问题的根源产生

JVM中，上面的代码对应三个操作：

memory = allocate(); //#1为对象分配内存空间

init(memory); //#2初始化成员

instance = memory; //#3设置instance,将其指向刚分配的内存空间。

重排序后可能出现的情况：

memory = allocate(); //#1为对象分配内存空间

instance = memory; //#2**设置instance,将其指向刚分配的内存空间。**

init(memory); //#3初始化成员

**实例并未初始化，但引用却赋给了对象**。

修复：将唯一实例使用volatile关键字修饰。

## volatile –保证：可见性，一定的“有序性”，

**保证可见性（跳过CPU缓存），可以通过volatile关键字来保证一定的“有序性”，（允许指令重排序），但不保证操作的原子性，**

volatile关键字可以保证**直接从主存中读取一个变量，如果这个变量被修改后，总是会被立刻写回到主存中去。**

Volatile实现内存可见性是通过store和load指令完成的；也就是对volatile变量执行写操作时，会在写操作后加入一条store指令，即**强迫线程将最新的值刷新到主内存中**；而在读操作时，会加入一条load指令，即**强迫从主内存中读入变量的值**。但volatile不保证volatile变量的原子性。

如：i++；

i++不是原子操作，因为其可以分为：

1 从主存读取i的值

2 CPU执行计算，将i的值+1

3 将结果刷新回主存

对于i++;操作，线程1和线程2都执行一次，最后输出Num的值可能是：1或者2

## Synchronized（Lock）-保证 ：可见性，有序性，原子性

**保证可见性和原子性**

一个同步块可以保证在**同一时刻仅有一个线程可以进入代码的临界区**。**同步块还可以保证代码块中所有被访问的变量将会从主存中读入，当线程退出同步代码块时，所有被更新的变量都会被刷新回主存中去，不管这个变量是否被声明为volatile。**

加锁（synchronized 同步）的功能不仅仅局限于互斥行为，同时还存在另外一个重要的方面：内存可见性。

在Java内存模型中，synchronized规定，线程在加锁时:

清空工作内存 →拷贝主存变量副本到工作内存（CPU缓存，CPU寄存器）→ 执行代码 →将更改后的共享变量的值刷新到主存 → 释放互斥锁。

## Synchronized和Volatile的比较

1. Synchronized保证内存可见性，操作的原子性和原子性
2. Volatile只能保证内存可见性和一定的“有序性”
3. Volatile不需要加锁，比Synchronized更**轻量级**，并不会阻塞线程（volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。）
4. volatile标记的变量不会被编译器优化,而synchronized标记的变量可以被编译器优化（如编译器重排序的优化）.
5. volatile是变量修饰符，仅能用于变量，而synchronized是一个方法或块的修饰符。

# Java内存管理

<https://www.cnblogs.com/gw811/archive/2012/10/18/2730117.html>

## 运行时数据区域

Java虚拟机在执行Java程序的过程中会把它所管理的内存划分为若干个不同的数据区域。这些区域都有各自的用途，以及创建和销毁的时间，**有的区域随着虚拟机进程的启动而存在，有些区域则是依赖用户线程的启动和结束而建立和销毁。**



### 1 程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是**当前线程所执行的字节码的行号指示器**。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），**字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。分支、循环、跳转、异常处理、线程恢复等**基础功能都需要依赖这个计数器来完成。

每条线程都需要有一个独立的程序计数器，各条线程之间的计数器互不影响，独立存储。

此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

### 2 栈（Java虚拟机栈）

与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stacks）也是**线程私有**的，它的**生命周期与线程相同**。虚拟机栈描述的是**Java方法**执行的内存模型：

**每个方法被执行的同时会创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。**每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

会抛出StackOverflowError和OutOfMemoryError异常

### 3 本地方法栈

本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而**本地方法栈则是为虚拟机使用到的Native方法服务**。虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。

会抛出StackOverflowError和OutOfMemoryError异常。

### 4 堆

对于大多数应用来说，Java堆（Java Heap）是Java虚拟机所管理的**内存中最大的一块**。Java堆是**被所有线程共享的一块内存区域**，**在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。**这一点在Java虚拟机规范中的描述是：**所有的对象实例以及数组都要在堆上分配**，但是随着JIT编译器的发展与逃逸分析技术的逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也渐渐变得不是那么“绝对”了。

**Java堆是垃圾收集器管理的主要区域**，因此很多时候也被称做“GC堆”（Garbage Collected Heap）。如果从内存回收的角度看，由于现在收集器基本都是采用的分代收集算法，所以Java堆中还可以细分为：新生代和老年代；再细致一点的有Eden空间、From Survivor空间、To Survivor空间等。

会出OutOfMemoryError异常。

### 5 方法区

方法区（Method Area）与Java堆一样，是**各个线程共享的内存区域**，它**用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据**。虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做Non-Heap（非堆），目的应该是与Java堆区分开来。

**这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载**，一般来说这个区域的回收“成绩”比较难以令人满意，尤其是类型的卸载，条件相当苛刻。

会抛出OutOfMemoryError异常。

#### 运行时常量池

运行时常量池（Runtime Constant Pool）是**方法区的一部分**。**Class文件**中除了有类的版本、字段、方法、接口等描述等信息外，还有一项信息是**常量池**（Constant Pool Table），用于存放**编译期生成的各种字面量和符号引用**，这部分内容将在类加载后存放到方法区的**运行时常量池**中。**具备动态性（如String类的intern()方法）。**

会受到方法区内存的限制，当常量池无法再申请到内存时会抛出OutOfMemoryError异常。

## 对象访问

在Java语言中，对象访问是如何进行的？

对象访问在Java语言中无处不在，是最普通的程序行为，但即使是最简单的访问，也会却涉及**Java栈、Java堆、方法区**这三个最重要内存区域。

如：

Object obj = new Object();

假设这句代码出现在方法体中

1. 那“Object obj”这部分的语义将会反映到**Java栈的本地变量表**中，作为一个**reference类型**数据出现。
2. 而“new Object()”这部分的语义将会反映到**Java堆**中，形成一块存储了Object类型所有**实例数据值**的结构化内存，这块内存的长度是不固定的。
3. 在Java堆中还必须包含能查找到此**对象类型数据**（如对象类型、父类、实现的接口、方法等）的地址信息，这些**类型数据则存储在方法区**中。

## 常量池分类

<https://www.cnblogs.com/holos/p/6603379.html>

### 全局字符串池

string pool也有叫做string literal pool

全局字符串池里的内容是在**类加载完成，经过验证，准备阶段之后**在**堆**中生成字符串对象实例，然后将该字符串对象实例的引用值存到string pool中（记住：**string pool中存的是引用值而不是具体的实例对象，具体的实例对象是在堆中开辟的一块空间存放**。），只有一份， 被所有类，所有线程共享。

### Class文件常量池

**.java文件被编译为.class文件时产生。存在于.class文件中。**

class文件中除了包含类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池(constant pool table)，用于**存放编译器生成的各种字面量和符号引用**。

字面量就是我们所说的常量概念，如文本字符串、被声明为final的常量值等。

符号引用是一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可

一般包括下面三种：

1. 类和接口的全限定名
2. 字段的名称和描述符
3. 方法的名称和描述符

### 运行时常量池

runtime constant pool

当java文件被编译成class文件之后，也就是会生成我上面所说的class常量池，那么运行时常量池又是什么时候产生的呢？

jvm在执行某个类的时候，必须经过**加载、连接、初始化**，而连接又包括**验证、准备、解析**三个阶段。而当类加载到内存中后，jvm就会将class常量池中的内容存放到运行时常量池中，由此可知，运行时常量池也是每个类都有一个。

class常量池中存的是字面量和符号引用，也就是说他们存的并不是对象的实例，而是对象的符号引用值。而经过解析之后，也就是把符号引用替换为直接引用，解析的过程会去查询全局字符串池，也就是我们上面所说的StringTable，以保证运行时常量池所引用的字符串与全局字符串池中所引用的是一致的。

### 总结

1. 全局字符串池在每个JVM中只有一份，存放的是字符串常量的**引用值**。
2. class常量池是在编译的时候每个class都有的，在编译阶段，存放的是常量的**符号引用**。
3. 运行时常量池是在类加载完成之后，将每个class常量池中的符号引用值转存到运行时常量池中，也就是说，每个class都有一个运行时常量池，类在解析之后，将符号引用替换成直接引用，与全局常量池中的引用值保持一致。

## 8种基本类型的包装类与常量池

<http://blog.csdn.net/chen1280436393/article/details/51768761>

Java中基本类型的包装类大部分实现了常量池技术，这些类是：Byte Short Integer Long Character Boolean，另外**两个浮点类型的包装类则没有实现**。另外**Byte、Short、Integer、Long、Character这5种包装类也只有在对应的值小于127时才能使用常量池。**String也实现了常量池技术。

JVM 会自动维护八种基本类型的常量池，int常量池中初始化有 **-128~127** 的数值（只维护-128~127的数值，范围外的不放在常量池中），所以当Integer=127时，在自动装箱过程中是取常量池中的数值，当Integer=128时，不在范围内，所以会new一个对象，而不是从常量池中返回。

# Java的几种加密方式

<http://www.jb51.net/article/73292.htm>

## BASE

严格地说，属于**编码格式**，而非加密算法，Base编码可用于**在HTTP环境下传递较长的标识信息**。例如，在Hibernate中，就采用了Base来将一个较长的唯一标识符（一般为bit的UUID）编码为一个字符串，用作表单和HTTP URL中的参数。在其他应用程序中，也常常需要把**二进制数据编码为适合放在URL（包括隐藏表单域）中的形式**。此时，采用Base编码具有不可读性，即所编码的数据不会被人用肉眼所直接看到。

**BASE的加密解密是双向的，可以求反解。**BASE64Encoder和BASE64Decoder是非官方JDK实现类。虽然可以在JDK里能找到并使用，但是在API里查不到。JRE中sun和com.sun开头包的类都是未被文档化的，他们属于java, javax 类库的基础，其中的实现大多数与底层平台有关，一般来说是不推荐使用的。

BASE 严格地说，属于编码格式，而非加密算法，主要就是BASE64Encoder、BASE64Decoder两个类，我们只需要知道使用对应的方法即可。

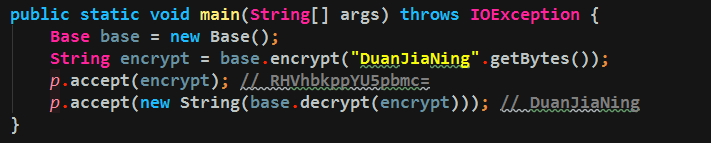
sun.misc.BASE64Decoder;  
sun.misc.BASE64Encoder;

按照RFC的定义，Base被定义为：Base内容传送编码被设计用来**把任意序列的位字节描述为一种不易被人直接识别的形式。**

**常见于邮件、http加密，截取http信息，你就会发现登录操作的用户名、密码字段是通过BASE加密的。**



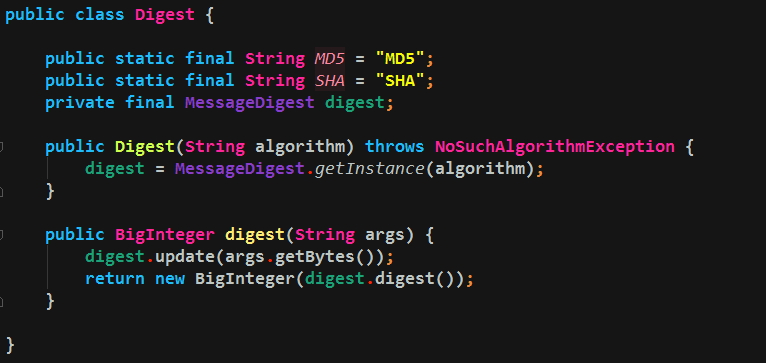
**示例：**



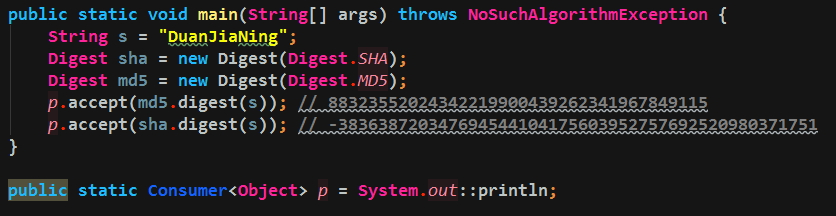
## MessageDigest（Message Digest algorithm）

信息摘要算法

用于确保信息传输完整一致。是计算机广泛使用的杂凑算法之一（又译摘要算法、哈希算法），主流编程语言普遍已有MD实现。



示例：



### MD5

广泛用于加密和解密技术，**常用于文件校验**。**不管文件多大，经过MD后都能生成唯一的MD值。可用来验证文件一致性。**

**特点**

1. **压缩性**：任意长度的数据，算出的MD值长度都是固定的。
2. **容易计算**：从原数据计算出MD值很容易。
3. **抗修改性**：对原数据进行任何改动，哪怕只修改个字节，所得到的MD值都有很大区别。
4. **弱抗碰撞**：已知原数据和其MD值，想找到一个具有相同MD值的数据（即伪造数据）是非常困难的。
5. **强抗碰撞**：想找到两个不同的数据，使它们具有相同的MD值，是非常困难的。

### SHA(Secure Hash Algorithm)

安全散列算法，安全哈希算法主要**适用于数字签名标准里面定义的数字签名算法。**

该算法的思想是**接收一段明文，然后以一种不可逆的方式将它转换成一段（通常更小）密文**，也可以简单的理解为取一串输入码（称为预映射或信息），并把它们转化为长度较短、位数固定的输出序列即散列值（也称为信息摘要或信息认证代码）的过程。散列函数值可以说是对明文的一种“指纹”或是“摘要”所以对散列值的数字签名就可以视为对此明文的数字签名。

### SHA与MD5的比较

因为二者均由MD导出，SHA和MD5彼此很相似。相应的，他们的强度和其他特性也是相似。

有以下几点不同：

1. **对强行攻击的安全性**：最显著和最重要的区别是SHA摘要比MD摘要长。SHA对强行攻击有更大的强度。
2. **对密码分析的安全性**：由于MD的设计，易受密码分析的攻击，SHA显得不易受这样的攻击。
3. **速度**：在相同的硬件上，**SHA的运行速度比MD慢。**

# Java中的锁

## 可重入锁

可重入锁又名递归锁，是**指在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁。可重入锁的一个好处是可一定程度避免死锁。**

对应java.util.concurrent.locks.**ReentrantLocak**

### ReentrantLocak

重入锁（ReentrantLock）是一种**递归无阻塞的同步机制。**

一个可重入的互斥锁（Lock），它具有与使用 synchronized 方法和语句所访问的隐式监视器锁定相同的一些基本行为和语义，但功能更强大。

1. ReentrantLock 将由当前已经成功获得锁，并且还没有释放锁的线程所拥有。
2. 使用lock获得锁，unlock释放锁。
3. 可以使用 isHeldByCurrentThread() 和 getHoldCount() 方法来检查当前线程是否获得锁及调用了几次lock

#### lock

1. 当锁没有被任何线程获得时，调用lock方法的线程将成功获的锁定并返回，将锁的保持计数设置为 1。
2. 如果当前线程已经获得该锁（重复获取锁），则将保持计数加 1，并且该方法立即返回。
3. 如果该锁已经被另一个线程获得，则出于线程调度的目的，将当前线程阻塞，并且在获得锁之前，该线程将一直处于休眠状态。

### Condition

<http://www.importnew.com/9281.html>

任何一个Java对象，都拥有一组监视器方法，主要包括wait()、notify()、notifyAll()方法，这些方法与synchronized关键字配合使用可以实现等待/通知机制。而且前面我们已经使用这种方式实现了生产者-消费者模式。类似地，Condition接口也提供类似的Object的监视器的方法，主要包括await()、signal()、signalAll()方法，这些方法与Lock锁配合使用也可以实现等待/通知机制。

Condition 将 Object 监视器方法（wait、notify 和 notifyAll）分解成截然不同的对象，以便通过将这些对象与任意 Lock 实现组合使用，为每个对象提供多个等待 set （wait-set）。

其中，**Lock 替代了 synchronized 方法和语句的使用，Condition 替代了 Object 监视器方法的使用。**

在Condition中，用await()替换wait()，用signal()替换notify()，用signalAll()替换notifyAll()。传统线程的通信方式，Condition都可以实现，这里注意，Condition是被绑定到Lock上的，要创建一个Lock的Condition必须用newCondition()方法。

这样看来，Condition和传统的线程通信没什么区别，Condition的强大之处在于**它可以为多个线程间建立不同的Condition**。

Condition是一个**多线程间协调通信的工具类**，使得某个（或者某些）线程一起等待某个条件（Condition）,只有该条件具备满足时，这些等待线程才会被唤醒( signal 或者 signalAll方法被带调用)，重新争夺锁。

相比Object实现的监视器方法，Condition接口的监视器方法具有一些Object所没有的特性：

1. Condition接口可以支持**多个等待队列**，在前面已经提到一个Lock实例可以绑定多个Condition，所以自然可以支持多个等待队列了
2. Condition接口支持响应中断，前面已经提到过
3. Condition接口支持当前线程释放锁并进入等待状态到将来的某个时间，也就是支持定时功能

#### 和Object监视器区别

在java中，对于任意一个java对象，它都拥有一组定义在java.lang.Object上监视器方法，包括wait()，wait(long timeout)，notify()，notifyAll()，这些方法配合synchronized关键字一起使用可以实现等待/通知模式。

同样，Condition接口也提供了类似Object监视器的方法，通过与Lock配合来实现等待/通知模式。



Condition支持在await时不响应Thread的interrupt请求（awaitUninterruptibly方法）：



Condition支持的定时唤醒功能：



#### await

在A线程中调用await()方法，A线程将**阻塞并释放锁**，之后A线程进入**该条件**的等待集中，直到有其它线程（B线程）调用signalAll()方法，之后A线程将解除阻塞并重新竞争锁。A线程重新获得锁之后，将从被阻塞的地方继续往下执行程序。

调用await方法后，**当前线程会释放锁并进入Condition变量的等待队列**，而当其他线程调用该条件的signal方法，通知正在Condition变量等待队列的线程从await方法返回，并且在返回前已经获得了锁。

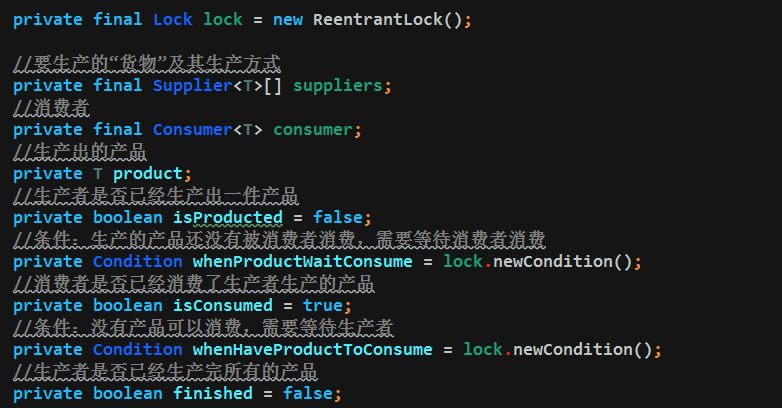
#### signalAll

该方法重新激活因为这个conditon条件而等待的所有线程。当这些线程从等待集中移出时，他们再次成为可运行的，调度器将再次激活它们，同时它们将试图重新进入该对象。一旦锁成为可用的，它们中的**某个**将从await（）调用返回，获得锁并从被阻塞的地方继续执行。

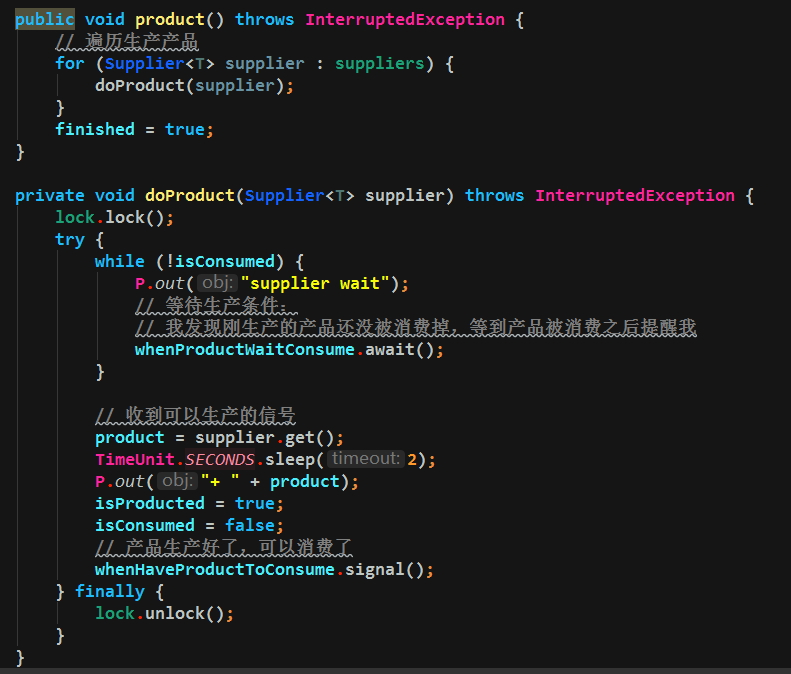
注意：signalAll()方法不会立即激活一个等待的线程。它仅仅解除等待的线程阻塞，以便这些线程可以在当前线程退出同步方法之后，通过竞争实现对对象的访问。

#### Condition实现生产者-消费者

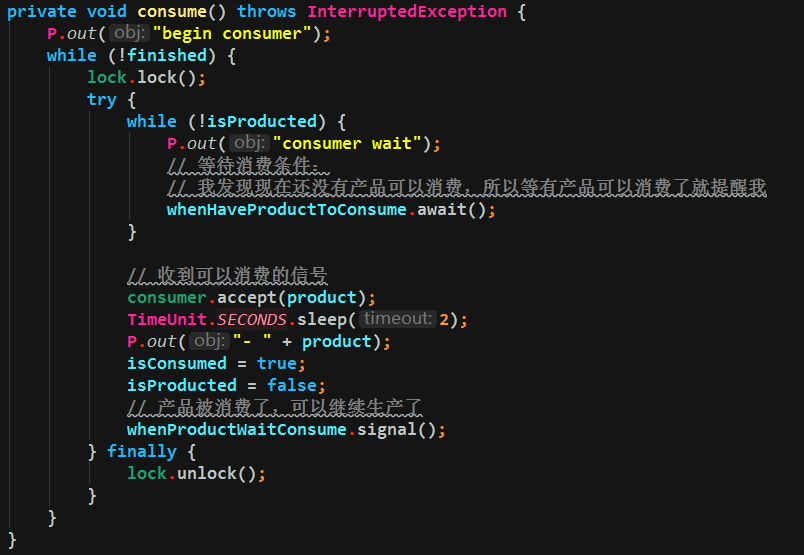
关键变量：



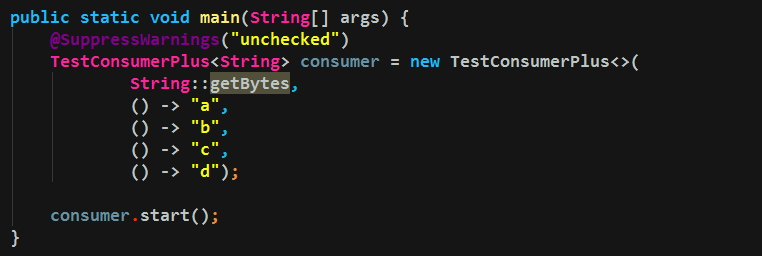
生产方法：

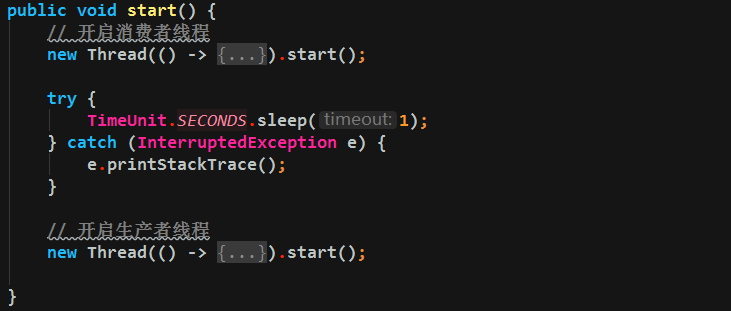


消费方法：

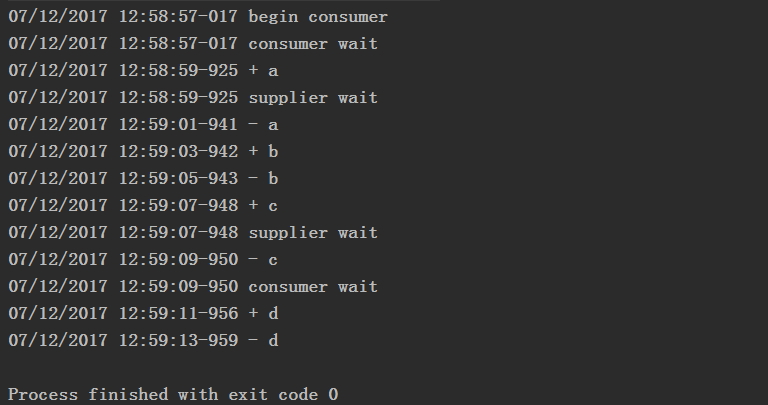


测试方法：





输出：



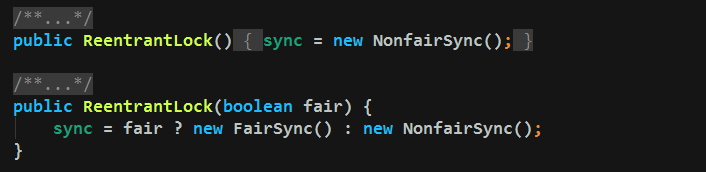
## 分类

### 公平锁/非公平锁

公平锁是指**多个线程按照申请锁的顺序来获取锁**。

非公平锁是指**多个线程获取锁的顺序并不是按照申请锁的顺序**，有可能后申请的线程比先申请的线程优先获取锁。有可能，会造成优先级反转或者饥饿现象。

**ReentrantLocak**通过构造函数指定该锁是否为公平锁，默认是非公平锁。**非公平锁的优点在于吞吐量比公平锁大**。



### 独享锁/共享锁

独享锁是指**该锁一次只能被一个线程所持有。**

共享锁是指**该锁可被多个线程所持有。**

对于Java ReentrantLock而言，其是独享锁。但是对于Lock的另一个实现类**ReadWriteLock**，其**读锁是共享锁，其写锁是独享锁**。读锁的共享锁可保证并发读是非常高效的，读写，写读 ，写写的过程是互斥的。

独享锁与共享锁也是通过AQS来实现的，通过实现不同的方法，来实现独享或者共享。

**ReentrantLock**和**Synchronized都是独享锁。**

### 互斥锁/读写锁

上面讲的独享锁/共享锁就是一种广义的说法，互斥锁/读写锁就是具体的实现。

互斥锁在Java中的具体实现就是**ReentrantLock**

读写锁在Java中的具体实现就是**ReadWriteLock**

### 乐观锁/悲观锁

乐观锁与悲观锁不是指具体的什么类型的锁，而是指看待并发同步的角度。

**悲观锁认为对于同一个数据的并发操作，一定是会发生修改的，哪怕没有修改，也会认为修改。**因此对于同一个数据的并发操作，悲观锁采取加锁的形式。悲观的认为，不加锁的并发操作一定会出问题。

**乐观锁则认为对于同一个数据的并发操作，是不会发生修改的。**在更新数据的时候，会采用尝试更新，不断重新的方式更新数据。乐观的认为，不加锁的并发操作是没有事情的。

从上面的描述我们可以看出，**悲观锁适合写操作非常多的场景，乐观锁适合读操作非常多的场景。不加锁会带来大量的性能提升。**

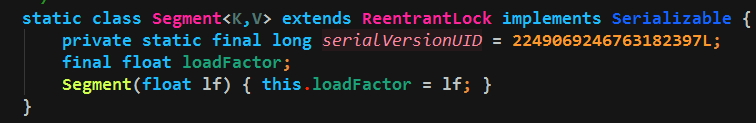
悲观锁在Java中的使用，就是利用各种锁。

乐观锁在Java中的使用，是无锁编程。

### 分段锁

分段锁其实**是一种锁的设计**，并不是具体的一种锁，对于**ConcurrentHashMap而言，其并发的实现就是通过分段锁的形式来实现高效的并发操作。**

我们以ConcurrentHashMap来说一下分段锁的含义以及设计思想，ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap（JDK7与JDK8中HashMap的实现）的结构，即**内部拥有一个Entry数组，数组中的每个元素又是一个链表**；同时又是一个ReentrantLock（Segment继承了ReentrantLock)。



当需要put元素的时候，**并不是对整个hashmap进行加锁，而是先通过hashcode来知道他要放在哪一个分段中，然后对这个分段进行加锁，**所以当多个线程执行put操作的时候，只要不是放在同一个分段中（由其hashcode决定），就能够实现真正的并行插入。

但是，在**统计size**的时候，可就是获取hashmap全局信息的时候，就需要获取所有的分段锁才能统计。

分段锁的设计目的是细化锁的粒度，当操作不需要更新整个数组的时候，就仅仅针对数组中的一项进行加锁操作。

### 偏向锁/轻量级锁/重量级锁

这三种锁是指**锁的状态**，并且是针对Synchronized。在Java 5通过引入锁升级的机制来实现高效Synchronized。这三种锁的状态是通过**对象监视器**在对象头中的字段来表明的。

1. **偏向锁**是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁。降低获取锁的代价。
2. **轻量级锁**是指当锁是偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，提高性能。
3. **重量级锁**是指当锁为轻量级锁的时候，另一个线程虽然是自旋，但自旋不会一直持续下去，当自旋一定次数的时候，还没有获取到锁，就会进入阻塞，该锁膨胀为重量级锁。重量级锁会让其他申请的线程进入阻塞，性能降低。

### 自旋锁

在Java中，自旋锁是指**尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU。**

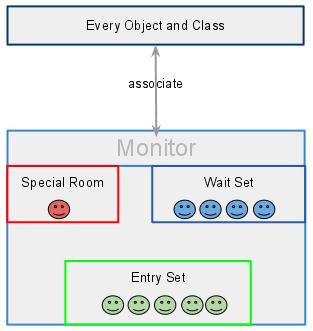
# Java监视器 monitor

<http://ifeve.com/monitors-java-synchronization-mechanism/>

监视器是操作系统实现同步的重要基础概念，同样它也用在JAVA的线程同步中。

## 什么是监视器

监视器可以看做是经过特殊布置的建筑，这个建筑有一个特殊的房间，该房间通常包含一些数据和代码，但是一次只能一个消费者(thread)使用此房间。



当一个消费者(线程)需要使用了这个房间，首先他必须到一个大厅(Entry Set)等待，操作系统调度程序将基于某些标准(e.g. FIFO)从大厅中选择一个消费者(线程)，进入特殊房间，如果这个线程因为某些原因被“挂起”，它将被调度程序安排到“等待房间”，并且一段时间之后会被重新分配到特殊房间。

监视器用来**监视线程进入这个特别房间，他确保同一时间只有一个线程可以访问特殊房间中的数据和代码。**

## JAVA中监视器的实现

在JAVA虚拟机中，每个对象(Object和Class)通过某种逻辑**关联监视器**，为了实现监视器的互斥功能，**每个对象都关联着一个锁**(有时也叫“**互斥量**mutex”)。

如果一个线程拥有了某个对象的锁，其他的线程就无法获得锁，直到这个线程释放了这个锁。

## 与java对应

Java的每一个对象都会关联一个监视器（Monitor），**监视器管理了线程对对象的临界区的访问。**

这里：对象可视为为监视器（建筑），特殊房间为临界区（如Synchronized方法或代码块），等待房间由监视器实现。

Java提供了wait，notify，notifyAll，Thread.sleep，Thread.interupt等方法让开发人员来操纵监视器。

# Java运算符优先级





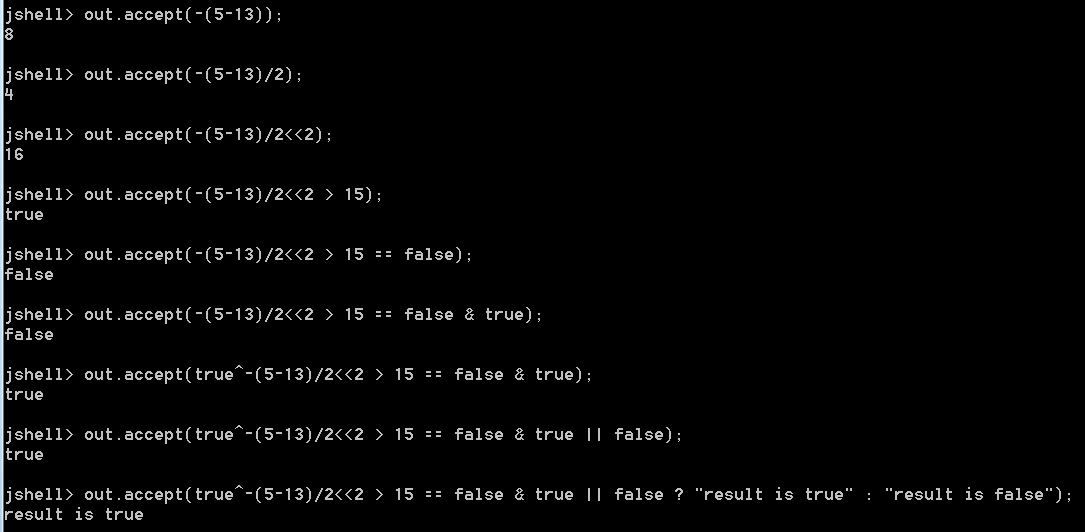
示例：

使用java9的jshell进行测试

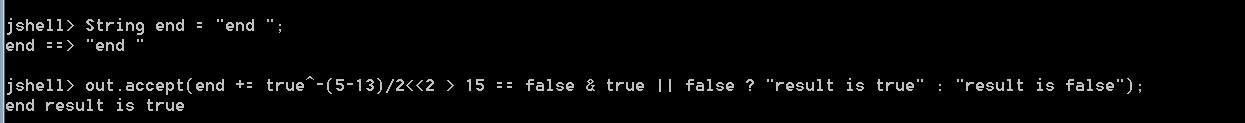
定义输出变量：



按优先级从高到底依次改写运算式：



定义变量end并测试优先级最低的 += 符号：



# Comparable和Comparator

<http://mp.weixin.qq.com/s?__biz=MjM5NzMyMjAwMA==&mid=2651479592&idx=2&sn=4d8fa2af187248963ea00a9e7fcfb404&chksm=bd2532578a52bb41da0bf348c53d8c7b4ac5af6c3dc1611faa6a9c22c6fc38c12615c80cc62b&mpshare=1&scene=23&srcid=1211CDntIMxgR5XsBLIERr0O#rd>

Comparable 是排序接口；若一个类实现了 Comparable 接口，就意味着 “该类支持排序”。而 Comparator 是比较器；我们若需要控制某个类的次序，可以建立一个 “该类的比较器” 来进行排序。

前者应该比较固定，和一个具体类相绑定，而后者比较灵活，它可以被用于各个需要比较功能的类使用。可以说前者属于 “静态绑定”，而后者可以 “动态绑定”。

我们不难发现：Comparable 相当于 “内部比较器”，而 Comparator 相当于 “外部比较器”。

# 并发编程

<http://www.cnblogs.com/dolphin0520/p/3920373.html>

在并发编程中，我们通常会遇到以下三个问题：原子性问题，可见性问题，有序性问题。

## 原子性

即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

只有简单的读取、赋值（而且必须是将数字赋值给某个变量，变量之间的相互赋值不是原子操作）才是原子操作。

Java内存模型只保证了基本读取和赋值是原子性操作，如果要实现更大范围操作的原子性，可以通过synchronized和Lock来实现。由于**synchronized和Lock能够保证任一时刻只有一个线程执行该代码块，那么自然就不存在原子性问题了，从而保证了原子性。**

## 可见性

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

Java提供了**volatile关键字来保证可见性**。

当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。

另外，通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且**在释放锁之前会将对变量的修改刷新到主存**当中。因此可以保证可见性。

## 有序性

即程序执行的顺序按照代码的先后顺序执行。

int i = 0;

boolean flag = false;

i = 1; //语句1

flag = true; //语句2

上面代码定义了一个int型变量，定义了一个boolean类型变量，然后分别对两个变量进行赋值操作。从代码顺序上看，语句1是在语句2前面的，那么JVM在真正执行这段代码的时候会保证语句1一定会在语句2前面执行吗？**不一定**，为什么呢？这里可能会发生指令重排序（Instruction Reorder）。

指令重排序：一般来说，**处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的**。

指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性。也就是说，**要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。**只要有一个没有被保证，就有可能会导致程序运行不正确。

# 深拷贝&浅拷贝

## 浅拷贝

在Java语言中，通过覆盖Object类的clone()方法就是实现浅克隆，在浅克隆中，当对象被复制时**只复制它本身和其成员中为值类型的成员变量**，而引用类型的成员对象并没有复制，也就是说原型对象只是将引用对象的**地址**复制一份给克隆对象，克隆对象和原型对象的引用类型成员变量还是指向相同的内存地址。

注意：能够实现克隆的Java类必须实现一个标识接口Cloneable，表示这个Java类支持被复制。如果一个类没有实现这个接口但是调用了clone()方法，Java编译器将抛出一个CloneNotSupportedException异常。

## 深拷贝

在深克隆中，**无论原型对象的成员变量是值类型还是引用类型，都将复制一份给克隆对象**，简单来说，在深克隆中，除了对象本身被复制外，对象所包含的**所有成员**变量也将复制。

那么如何实现深克隆呢？

1. 在Java语言中，如果需要实现深克隆，可以通过**序列化(**Serialization)方式来实现。序列化就是将对象写到流的过程，写到流中的对象是原有对象的一个拷贝，而原对象仍然存在于内存中。通过序列化实现的拷贝不仅可以复制对象本身，而且可以复制其引用的成员对象，因此通过序列化将对象写到一个流中，再从流里将其读出来，可以实现深克隆。需要注意的是能够实现序列化的对象其类必须实现Serializable接口，否则无法实现序列化操作。
2. 覆写clone方法时通过调用引用类型成员变量的clone方法（前提为成员变量类型实现Cloneable接口并覆写clone方法，但这涉及到一个“有多深”的问题）。