**Java面试题集锦**

##### Java基础

面向对象：

**1).内部类**

内部类分为四种：成员内部类、局部内部类、匿名内部类、静态内部类;

**成员内部类:**

i)成员内部类是依附其外部类而存在的，如果要产生一个成员内部类，比如有一个其外部类的实例

ii)成员内部类中不可以定义静态方法.

iii)成员内部类可以声明为private的，声明为private的成员内部类对外不可见，外部不能调用私有成员内部类的public方法

iv)成员内部类可以声明为public的，声明为public的成员内部类对外可见，外部也可以调用共有成员内部类的public方法.

v)成员内部类可以访问其外部类的私有属性，如果成员内部类的属性和其外部类的属性重名，则以成员内部类的属性值为准.

**局部内部类:** 定义在一个方法或者特定作用域里面的类

public static void main(String[] args)

{

    final int i = 0;

    class A

    {

        public void print()

            {

            System.out.println("AAA, i = " + i);

        }

    }

    A a = new A();

    a.print();

}

局部内部类没有访问修饰符，另外局部内部类要访问外部的变量或者对象，该变量或对象的引用必须是用final修饰的

**匿名内部类**: 匿名内部类是唯一没有构造器的类，其使用范围很有限，一般都用于继承抽象类或实现接口（注意只能继承抽象类，不能继承普通类），匿名内部类Java自动为之起名为XXX$1.classs。另外，和局部内部类一样，td必须是用final修饰的.

**静态内部类:** 用static修饰的内部类就是静态内部类, 静态内部类并不持有其外部类的引用.

1. 静态内部类中可以有静态方法，也可以有非静态方法
2. **静态内部类只能访问其外部类的静态成员与静态方法**
3. 和普通的类一样，要访问静态内部类的静态方法，可以直接”.”出来不需要一个类实例；要访问静态内部类的非静态方法，必须拿到一个静态内部类的实例对象

④注意实例化成员内部类和实例化静态内部类这两种不同的内部类时写法上的差别

（1）成员内部类：外部类.内部类 XXX = 外部类.new 内部类();

（2）静态内部类：外部类.内部类 XXX = new 外部类.内部类();

**2).继承，多态**

i).接口是完全抽象的设计，不能实例化。使A用new方式创建的借口类型，实际上是创建了一个匿名类，该匿名类实现了接口类型

ii). 如果两个接口声明了相同的变量x，则当某接口同时继承这两个接口，或者某类同时实现这两个接口时，通过简单名称访问会产生编译错误

iii).  如果两个接口中声明了相同名称的方法m，并且两个方法没有构成重载，则当某接口能够同时继承这两个接口，或者某类能够同时继承这两个接口时，必须存在一种方法签名，使得该签名同时为两个m方法签名的子签名，并且在方法的返回类型上，必须存在一种类型，使得该类型同时为两个m方法返回类型的可替换类型.

iv).为什么使用继承:使用继承可以有效的实现代码服用,表面重复代码的出现;当两个类具有相同的特征(属性)和行为(方法)时, 可以将相同的部分抽取出来放到一个类中作为父类.

v)为什么使用多态:可以增强程序的可扩展性及可维护性,使得代码更加简洁,不但能减少代码的工作量,而且也能大大提高程序的可维护性以及可扩展性.

**3).静态类，非静态类，静态变量，非静态变量，final**

i)静态类与非静态类:Java中可以在类中定义静态内部类和非静态内部类, 主要区别为: 静态内部类不需要有指向外部类的引用,非静态内部类需要持有对外部类的引用; 非静态内部类内部能够访问到外部类的静态和非静态的成员, 静态内部类不能访问外部类的非静态成员,只能访问外部类的静态成员; 一个非静态内部类不能脱离外部类实体被创建, 一个非静态内部类可以访问外部类的数据和方法, 因为它本身就在外部类里面.

ii)静态变量用static修饰, 在JVM中静态变量的加载顺序在对象之前,因此静态变量不依附于对象的存在,可以在不实例化类的情况下直接使用静态变量,因此静态变量又叫类变量; 一个类不管创建多少个对象, 静态变量在内存中只有一个, 在虚拟机加载类的过程中为静态变量分配内存, 放在方法区, 属于静态变量, 被所有实例共享, 当类被卸载时,静态变量被销毁,并释放内存空间,static变量的生命周期取决于类的生命周期.

iii).final: 声明成员变量,方法,类以及本地变量,一旦声明为final,则不能改变,否则编译器报错; 声明方法: 当方法声明为final时则代表这个方法不能被子类重写; final类: 不能被继承的类,例如String,Interger类都是final的; static final的作用: static表明只有一个, final表明是一个常量, static final表明变量一旦给值,就不可修改,并且可以通过类名访问.

iv).使用final的好处: 提高了性能, JVM和Java应用都会缓存final变量; final变量可以安全的在多线程环境下进行共享,而不需要额外的同步开销; JVM会对final关键字修饰的变量和方法进行优化.

v).final使用注意事项:final成员变量必须要初始化或者在构造函数中初始化; 不能对final变量再次赋值;匿名内部类中变量都必须是final的; final类不可能是abstract的; final方法在编译阶段绑定,称为静态绑定.

**4).异常与错误的处理, throws, throw, 异常处理顺序，finally的用法**

**异常分类:**

i)编译时异常:程序正确, 但是因为外在的环境条件引发,例如:用户错误及IO问题,程序试图打开一个不存在的远程socket端口,这并非程序本身的逻辑错误,而很可能是远程主机名字写错, 所以必须要处理, 编译期异常必须要处理, 如果不捕获,则程序将不能被编译;

ii)运行时异常:程序存在bug,如数组越界,除数为0等,这类异常需要修改程序来避免,这类错误必须处理.

iii).error:可能源于bug,但一般更可能源于环境问题,如内存好近,错误在程序中无需处理.

**异常处理:**

i)try: 试试它所包含的代码段中是否会发生异常; jdk7中引入try-with-resources:

try(resources){

xxxxx

}catch(Exception e){

xxxx

}

称为ARM块(AutomaticResources Management), 如果resources是数据流,则在执行完毕后会自动关闭.其中数据流资源必须实现AutoCloseable接口,并且重写close方法,当资源不在使用时会自动调用close方法.

Try-with-resources的本质: 编译器会自动生成finally块, 在finally块中调用资源重写的close方法.最上层抛出的是最后一个抛出的异常. 通过反编译代码可以看到异常处理时加入了var.addSuppressed(var1), 这是由于jdk7之前的出现异常抑制导致, jdk7之后再Throwable类中新增了addSuppressed方法,支持将一个异常附加到另一个异常上,从而避免异常屏蔽(异常被屏蔽,并没有被丢弃,通过异常的getSuppressed方法可以企图被抑制的异常),被屏蔽的异常会在Suppressed的提示后输出;在使用try-with-resource的工程中,一定要了解资源的close方法的内部逻辑实现,否则还是可能会导致资源泄漏问题, 例如在Java的BIO中采用了大量的装饰器模式,当调用装饰器的close方法时,本质上是调用了装饰器内部包裹的流的close方法,例如:

try(FileInputStream fin = new FileInputStream(new File(“input.txt”));

GZIPOutputStream out = new GZIPOutputStream(new FileOutputStream(new File(“out.txt”)));){

byte[] byte = new byte[4096];

}catch(IOExcption e){

e.printStackTrace();

}

由于GZIPOutputStream是FileOutputStream的装饰器, 这段代码的本质会在嗲用的后面加上finally代码块,并且在里面调用FileOutputStream的close方法和GZIPOutputStream的close方法,事实上GZIPOutputStream的变量out代表的是被装饰FileOutputStream,在调用out的close的方法之前, GZIPOutputStream会调用finish关闭方法,然而该操作还是会继续往FileOutputStream中写入数据,如果此时出现异常,则out.close()方法会被略过,然而这个才是最底层的资源关闭方法,为了避免这种情况发生,一般会在try()中单独声明底层的的资源,保证对应的close方法一定能够被调用, 如下面的更正代码所示:

try(FileInputStream fin = new FileInputStream(new File(“input.txt”));

FileOutputStream fout = new FileOutputStream(new File(“out.txt”));

GZIPOutputStream out = new GZIPOutputStream(fout);){

byte[] byte = new byte[4096];

}catch(IOExcption e){

e.printStackTrace();

}

try-with-resource时,如果对外部资源的处理和对外部的资源的关闭均遭遇了异常,”关闭异常”将被抑制,”处理异常”将被抛出,但”关闭异常”并没有丢失,而是存放在异常处理列表中.

**异常处理顺序:**

throw: 语句抛出一个异常, 语法: throw 异常对象;

throws: 方法可能抛出的异常,用在声明方法时表示该方法可能抛出的异常 将抛出的异常交给上层调用它的方法程序处理.

throw与throws的区别: throw执行在方法体内,throws用在方法声明后面,表示再抛出异常,由方法的调用者来处理; throw是具体的向外抛出异常,抛出的是一个异常实例,throws声明是哪种异常,使它的调用者捕获这个异常; throws是表示出现异常的一种可能性,throw则是抛出了异常,执行throw则一定是抛出了某种异常对象.

**try, catch, finally的执行顺序:**

i).不管try语句块正常结束还是异常结束,finally语句块都是会执行的,如果try语句正常执行,则执行结束后进入finally块中执行; 如果try中异常进入catch后,catch执行结束后再执行finally块; 如果try中有控制语句(return,break,continue,throw),此时finally的执行顺序就要取决于具体情况:

try{

System.out.println(“try block”);

return;

}finally{

System.out.println(“finally block”);

}

执行结果为:

try block

finally block

结果表明finally语句块在try语句块中的return语句之前执行

try{

System.out.println(“try block”);

int i =1;

i = i/0;

return 1;

}catch(Exception e){

System.out.println(“catch block”);

return 2;

}

finally{

System.out.println(“finally block”);

}

执行结果为:

try block

catch block

finally block

return 2(return 执行catch中的)

结果表明finally语句块在try和catch语句块的控制语句之前执行,其中return和throw会把控制权交给调用者,而break和continue的控制权是在当前方法内转移.当finally中对try或者catch语句中的return返回值做了改动会不会影响return的结果?例如:

int i = 1;

try{

System.out.println(“try block”);

return i;

}finally{

System.out.println(“finally block”);

i++;

}

结果返回的是1; 实际上Java虚拟机会把finally语句块作为subroutine直接插入到try或者catch语句块的控制语句之前执行, 但是在执行finally语句块之前,try或者catch语句块会保留其返回值到本地变量表(local variable table)中,等到finally语句块执行完毕后,再回复保留的返回值到操作数栈中,然后通过return或者throw语句将其返回给方法的调用者, 这种机制只使用与return和throw,对于break和continue不适用;

int i = 1;

try{

i = 4;

}finally{

i++;

return i;

}

结果为i=5;

int i = 1;

try{

i = 4;

}finally{

i++;

}

return i;

结果为i=5; 如果finally中有return语句会把try或者catch中的return语句覆盖掉.

try或者catch中如果return的是一个引用而不是基本数据类型,那么在finally中改变这个引用就会影响到return的返回值:

Test test = new Test();

test.setA(10); test.setB(20);

try{

System.out.println("try");

return test;

}catch (Exception e){

System.out.println("catch");

}finally {

System.out.println("finally");

test.setA(100);

test.setB(200);

}

return test;

最终返回的结果A为100, B为200. 这是因为引用类型用的是同一个地址空间,当改变了引用时就会导致return后的引用类型发生变化.

**5).常用类**

System类:提供标准输入,标准输出和错误输出流; 对外部定义的属性和环境变量的访问;加载文件的和库的方法;快捷复制数组.

Runtime类:每个Java应用程序都有一个Runtime实例,使应用程序能够与其运行环境相连接.不能new,可以通过getRuntime方法获取当前运行时.应用程序不能自己创建Runtime实例.

Date和SimpleDateFormat类: Data类，位于java.util 包下,不同于Java.sql.Date; Data类中有好多方法被废弃了。SimpleDateFormat是java.text包下的.

Calendar类: Calendar类本身是一个抽象的基类。与其他语言环境敏感类一样，Calendar提供了一个类方法 getInstance,以获得此类型的一个通用的对象。Calendar的getInstance方法返回一个Calendar对象，其日历字段已由当前日期和时间初始化。

Math类: Math类包含用于执行基本数学运算的方法，比如指数、对数、平方根、三角函数等.

Random类: public class Random extands Object implements Serializable。Random类的实例用于生成随机数流.

**6).正则表达式：Pattern, Matcher**

Pattern类:编译正则表达式后创建一个匹配模式;

Matcher类:使用Pattern实例提供的模式信息对正则表达式进行匹配.

**7).反射, 通过反射获取和设置对象的私有字段的值,单例模式避免反射攻击**

**为什么要使用反射:**首先理解什么是静态编译和动态编译,静态编译是指在编译期确定类型,绑定对象;动态编译指在运行时确定类型,绑定对象,动态编译能最大限度的发挥Java的灵活性,体现了多态的应用,降低类之间的耦合性.

**反射机制的作用(运行时):**

i)在运行时判断任意一个对象所属类;

ii).在运行时构造任意一个类的对象;

iii).在运行时判断任意一个类所具有的成员变量和方法(通过反射甚至可以调用private方法);

iv).在运行时调用任意一个对象的方法;

通过反射,可以在运行时获得程序或者程序集中每一个类型的成员和成员信息,程序中一般的对象的类型都是在编译期就确定下来的,而Java的反射机制可以动态的创建对象并调用其属性,这样的对象类型在编译期是未知的,**反射的核心是JVM在运行时动态加载类或调用方法/访问属性,而不需要事先知道巡行对象是谁.**

**反射的主要用途:**

i).开发各种通用框架,例如spring,通过XML文件配置javabean,action类等,为了保证框架的通用性,根据配置文件加载不同的对象或类,调用不同的方法,这个时候就要用到反射.例如在struts.xml中配置action时,就会通过class配置信息动态的创建Action实例,然后通过反射调用对应的execute方法.

**反射的使用:**

**i)Class对象:**任何一个类都是Class类(类类型)的实例对象,在Java中,每个class都有一个相应的Class对象,也就是说,当编写一个类(.java文件)时,编译完成后,就会产生一个Class对象,表示这个类的类型信息,运行期间,如果要产生某个对象,JVM就会检查该类型的Class对象是否已被加载,如果没有被加载,JVM会根据类的名称找到.class文件并加载它,一旦某个类型的Class对象被加载到内存,就可以用它来产生该类型的所有独享. 获取Class实例对象的三种方式:

使用Class类的forName静态方法:

public static Class<?> forName(String className)

Class.forName(driver);// 在JDBC开发中常用此方法加载数据库驱动

直接获取某一个对象的Class:

Class<?> klass = int.class;

Class<?> classInt = Integer.TYPE;

调用某个对象的getClass()方法:

StringBuilder str = new StringBuilder(“123”);

Class<?> klass = str.getClass();

**ii).判断是否为某个类的实例:**用instanceof关键字判断是否为某个类的实例,通过Class对象的isInstance()方法来判断是否为某个类的实例:

public native boolean isInstance(Object obj);

**iii).创建实例:**通过反射来生成对象:

使用Class对象的newInstance()方法来创建Class对象对应的实例

Class<?> c = String.class;

Object str = c.newInstance();

先通过Class对象获取指定的Constructor对象,再调用Constructor对象的newInstance()方法来创建实例,这种方法可以用指定的构造器构造类的实例:

//获取String所对应的Class对象

Class<?> c = String.class;

//获取String类带一个String参数的构造器

Constructor constructor = c.getConstructor(String.class);

//根据构造器创建实例

Object obj = constructor.newInstance("23333");

System.out.println(obj);

**iv).获取类中的方法:** 获取某个Class对象的方法集合

getDeclareMethods()返回类或接口声明的所有方法,包括public,protected,default,private方法,但是不包括继承的方法:

public Method[] getDeclaredMethods() throws SecurityException

getMethods()方法返回某个类的所有公共方法,包括其继承类的公共方法:  
public Method[] getMethods() throws SecurityException

getMethod方法返回一个特定的方法，其中第一个参数为方法名称，后面的参数为方法的参数对应Class的对象.

public Method getMethod(String name, Class<?>... parameterTypes)

通过getMethods()获取的方法可以获取到父类的方法,比如java.lang.Object下定义的各个方法

例子:

Class<?> c = methodClass.class;

Object object = c.newInstance();

Method[] methods = c.getMethods();

Method[] declaredMethods = c.getDeclaredMethods();

//获取methodClass类的add方法

Method method = c.getMethod("add", int.class, int.class);

//getMethods()方法获取的所有方法

System.out.println("getMethods获取的方法：");

for(Method m:methods)

System.out.println(m);

//getDeclaredMethods()方法获取的所有方法

System.out.println("getDeclaredMethods获取的方法：");

for(Method m:declaredMethods)

System.out.println(m);

}

class methodClass {

public final int fuck = 3;

public int add(int a,int b) {

return a+b;

}

public int sub(int a,int b) {

return a+b;

}

}

**v).获取构造器信息:**获取构造器的用法主要是通过Class类的getConstructor方法得到Constructor类的一个实例,而Constructor类有一个newInstance方法可以创建一个对象实例,此方法可以根据传入的参数来调用对应的Constructor创建实例对象.

public T newInstance(Object ... initargs)

**vi).获取类的成员变量信息:**

*getFiled*: 访问公有的成员变量  
*getDeclaredField*：所有已声明的成员变量。但不能得到其父类的成员变量

vii).调用方法: 从类中获取了一个方法后，就可以用invoke()方法来调用这个方法.

Class<?> klass = methodClass.class;

//创建methodClass的实例

Object obj = klass.newInstance();

//获取methodClass类的add方法

Method method = klass.getMethod("add",int.class,int.class);

//调用method对应的方法 => add(1,4)

Object result = method.invoke(obj,1,4);

System.out.println(result);

class methodClass {

public final int fuck = 3;

public int add(int a,int b) {

return a+b;

}

public int sub(int a,int b) {

return a+b;

}

}

**viii).利用反射创建数组:**

Class<?> cls = Class.forName("java.lang.String");

Object array = Array.newInstance(cls,25);// 为java.lang.reflect.Array

//往数组里添加内容

Array.set(array,0,"hello");

Array.set(array,1,"Java");

Array.set(array,2,"fuck");

Array.set(array,3,"Scala");

Array.set(array,4,"Clojure");

//获取某一项的内容

System.out.println(Array.get(array,3));

**ix).通过反射理解泛型的本质:**

ArrayList list1 = new ArrayList();

ArrayList<String> list2 = new ArrayList<String>();

list2.add("Hello");

// list2.add(20); //报错

Class c1 = list1.getClass();

Class c2 = list2.getClass();

System.out.println(c1 == c2);

/\*\*

\* 反射操作都是编译之后的操作

\* c1==c2结果返回true，说明编译之后集合的泛型是去泛型化的

\* java中集合的泛型是为了防止错误输入的，只在编译阶段有效，绕过编译就无效了

\* 验证：通过方法的反射来绕过编译

\*/

try {

Method m = c2.getMethod("add", Object.class);

m.invoke(list2,20);

System.out.println(list2);

} catch (Exception e) {

e.printStackTrace();

}

**x).访问私有字段和私有方法:**

为了访问私有字段,需要调用Class.getDeclaredField(String name)或者Class.getDeclareFields()方法,而Class.getField(String name)和Class.getFields()仅仅返回共有的字段.需要设置field.setAccessible(true). 同样为了访问私有方法需要调用Class.getDeclareMethod(String name, Class[] parameterTypes)和Class.getDeclaredMethods().

**反射的缺点:**反射会额外消耗系统资源,因此若果不需要动态的创建一个对象,那么久不需要用反射;反射调用方法时可以忽略权限检查,因此可能会破坏封装性而导致安全问题;编译器没法对反射相关的代码进行优化,所以导致反射的性能较慢.

**如何避免单例模式的反射攻击:**

Lazy initialization holder class方式实现: 满足双检锁但是没有显示的同步

public class Singleton{

private static Boolean flag = false;

//私有构造函数,保证外界无法直接实例化

private Singleton(){}

private static class SingletonHolder{

private static final Singleton INSTANCE = new Singleton();

}

//提供外部访问

public static Singleton getInstance(){

return SingletonHolder.INSTANCE;

}

}

通过反射获取构造函数,然后调用setAccessible(true)就可以调用私有的构造函数:

Class<?> class = Singleton.class;

Constructor<?> c = class.getDeclaredConstructor(null);

c.setAccessible(true);

Singleton s1 = (Singleton)c.newInstacne();

Singleton s2 = (Singleton)c.newInstance();

通过对构造函数的修改可以避免这种攻击:

private Singleton(){

synchronized(Singleton.class){

if(flag==false){

flag = true;

}else{

throw new RuntimeException(“单例模式被破坏”);

}

}

}

**或者使用枚举实现单例模式可以防止反射攻击:原理如下**

public enum Singleton {

INSTANCE {

@Override

protected void read() {

System.out.println("read");

}

@Override

protected void write() {

System.out.println("write");

}

};

protected abstract void read();

protected abstract void write();

}

**反编译后代码:**

public abstract class Singleton extends Enum{

private Singleton(String s, int i){

super(s, i);

}

protected abstract void read();

protected abstract void write();

public static Singleton[] values(){

Singleton asingleton[];

int i;

Singleton asingleton1[];

System.arraycopy(asingleton = ENUM$VALUES, 0, asingleton1 = new Singleton[i = asingleton.length], 0, i);

return asingleton1;

}

public static Singleton valueOf(String s){

return (Singleton)Enum.valueOf(singleton/Singleton, s);

}

Singleton(String s, int i, Singleton singleton){

this(s, i);

}

public static final Singleton INSTANCE;

private static final Singleton ENUM$VALUES[];

static {

INSTANCE = new Singleton("INSTANCE", 0) {

protected void read() {

System.out.println("read");

}

protected void write(){

System.out.println("write");

}

};

ENUM$VALUES = (new Singleton[] {

INSTANCE

});

}

}

从反编译代码可以看到: 类的修饰**abstract**，所以没法实例化，反射也无能为力; 关于线程安全的保证，其实是通过类加载机制来保证的，INSTANCE的实例化时机，是在static块中，JVM加载类的过程显然是线程安全的。对于static修饰的代码块, 是不能进行序列化的,因此枚举实现单例模式是最好的选中, 但是缺点是占用空间较大.

**8).JDK6和JDK7中substring的区别以及原理**

substring(int beginIndex, int endIndex)方法在jdk 6和jdk 7中的实现是不同的。substring(int beginIndex, int endIndex)方法截取字符串并返回其[beginIndex,endIndex-1]范围内的内容.

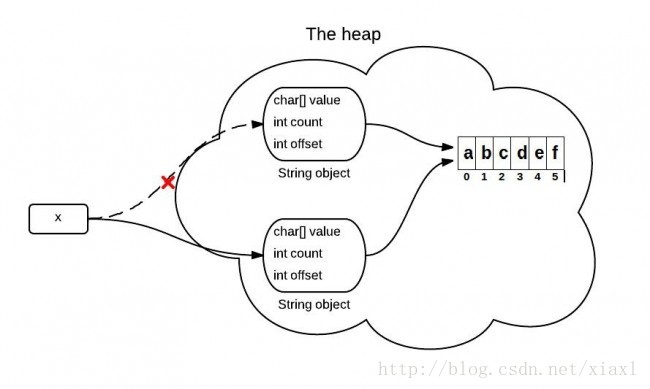
String x = "abcdef";

x = x.substring(1,3);

System.out.println(x);

输出: bc

JDK6中的substring: String是通过字符数组实现的。在jdk 6 中，String类包含三个成员变量：char value[]， int offset，int count。他们分别用来存储真正的字符数组，数组的第一个位置索引以及字符串中包含的字符个数. 当调用substring方法的时候，会创建一个新的string对象，但是这个string的值仍然指向堆中的同一个字符数组。这两个对象中只有count和offset 的值是不同的。



//JDK 6

String(int offset, int count, char value[]) {

this.value = value;//每次都会创建指向原string的字符串

this.offset = offset;

this.count = count;

}

public String substring(int beginIndex, int endIndex) {

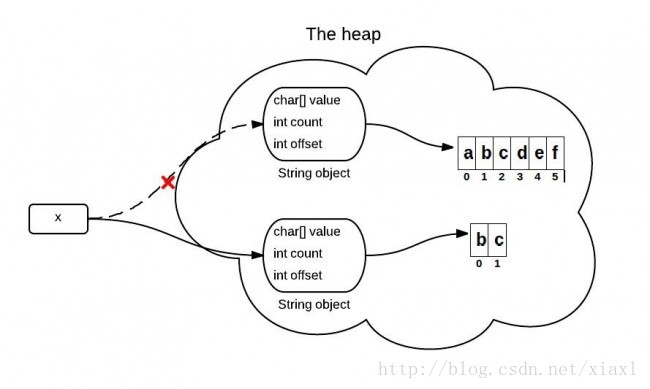
//check boundary

return new String(offset + beginIndex, endIndex - beginIndex, value);

}

JDK6中的substring导致的问题: 如果你有一个很长很长的字符串，但是当你使用substring进行切割的时候你只需要很短的一段。这可能导致性能问题，因为你需要的只是一小段字符序列，但是你却引用了整个字符串（因为这个非常长的字符数组一直在被引用，所以无法被回收，就可能导致内存泄露）。在JDK 6中，一般用以下方式来解决该问题，原理其实就是生成一个新的字符串并引用它: x = x.substring(x, y) + "".

JDK7中的substring: 在jdk 7 中，substring方法会在堆内存中创建一个新的数组.



//JDK 7

public String(char value[], int offset, int count) {

//check boundary

this.value = Arrays.copyOfRange(value, offset, offset + count);

}

public String substring(int beginIndex, int endIndex) {

//check boundary

int subLen = endIndex - beginIndex;

return new String(value, beginIndex, subLen);

}

**9).String.valueOf与Integer.toString的原理与区别**

toString源码: 此方法返回对象本身，在java.lang.Object类中也有toString()方法，所以Java对象都可以调用此方法，但使用的时候**必须保证要转换的对象不为null**，否则将抛出NullPointerException异常.

public String toString() {

return this;

}

valueOf源码: 不用担心Object为null，但使用的时候也要小心，**当Object为null时，它的返回值是“null”，而不是null**，是有区别的.

public static String valueOf(Object obj) {

return (obj == null) ? "null" : obj.toString();

}

**10).关键字:transient, instanceof, volatile, synchronized, final, static, const**

**transient:** 标记为transient的变量，在对象存储时，这些变量状态不会被持久化。当对象序列化的保存在存储器上时，不希望有些字段数据被保存，为了保证安全性，可以把这些字段声明为transient。

**strictfp**:strictfp的意思是FP-strict，也就是说精确浮点的意思。在Java虚拟机进行浮点运算时，如果没有指定strictfp关键字时，Java的编译器以及运行环境在对浮点运算的表达式是采取一种近似于我行我素的行为来完成这些操作，以致于得到的结果往往无法令你满意。而一旦使用了strictfp来声明一个类、接口或者方法时，那么所声明的范围内Java的编译器以及运行环境会完全依照浮点规范IEEE-754来执行。因此如果你想让你的浮点运算更加精确，而且不会因为不同的硬件平台所执行的结果不一致的话，那就请用关键字strictfp。

**volatile:**修饰变量,在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。而且，当成员变量发生变化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。

**instanceof:**用来在运行时指出对象是否是特定类的一个实例.

**synchronized:**修饰方法时不能被继承,父类中的方法用synchronized修饰,则子类重写该方法时不会默认是synchronized;接口中你的方法不能使用synchronized关键字;构造方法不能使用synchronized关键字,但可以使用synchronized代码块来进行同步.

**修饰代码块:** 被修饰的代码块称为同步语句块，其作用的范围是大括号{}括起来的代码，作用的对象是调用这个代码块的对象

**修饰方法:** 被修饰的方法称为同步方法，其作用的范围是整个方法，作用的对象是调用这个方法的对象.

**修饰静态方法:** 其作用的范围是整个静态方法，作用的对象是这个类的所有对象.

**修饰一个类:** 其作用的范围是synchronized后面括号括起来的部分，作用主的对象是这个类的所有对象.

A. 无论synchronized关键字加在方法上还是对象上，如果它作用的对象是非静态的，则它取得的锁是对象；如果synchronized作用的对象是一个静态方法或一个类，则它取得的锁是对类，该类所有的对象同一把锁。   
B. 每个对象只有一个锁（lock）与之相关联，谁拿到这个锁谁就可以运行它所控制的那段代码。   
C. 实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

**const**是Java预留关键字，用于后期扩展用，用法跟final相似，不常用

**11).序列化底层原理, 序列化与单例模式,如何防止序列化破坏单例模式**

对象序列化机制（object serialization）是Java语言内建的一种对象持久化方式，通过对象序列化，可以把对象的状态保存为字节数组，并且可以在有需要的时候将这个字节数组通过反序列化的方式再转换成对象。对象序列化可以很容易的在JVM中的活动对象和字节数组（流）之间进行转换。对象的序列化与反序列化被广泛应用到RMI(远程方法调用)及网络传输中.

**序列化API:**

java.io.Serializable

java.io.Externalizable

ObjectOutput

ObjectInput

ObjectOutputStream

ObjectInputStream

**实现序列化的Serializable和Externalizable的区别:** Externalizable继承了Serializable，该接口中定义了两个抽象方法：writeExternal()与readExternal()。当使用Externalizable接口来进行序列化与反序列化的时候需要开发人员重写writeExternal()与readExternal()方法。在使用Externalizable进行序列化的时候，在读取对象时，会调用被序列化类的无参构造器去创建一个新的对象，然后再将被保存对象的字段的值分别填充到新对象中。所以，实现Externalizable接口的类必须要提供一个public的无参的构造器。如果实现了Externalizable接口的类中没有无参数的构造函数，在运行时会抛出异常：java.io.InvalidClassException。一般使用ObjectOutputStream的writeObject方法把一个对象进行持久化。再使用ObjectInputStream的readObject从持久化存储中把对象读取出来。

**Serializable和Externalizable的比较:**

Serializable提供内建支持,易于实现,但是占用空间大,额外的开销导致速度慢;

Externalizable开销较少,速度较快,但是虚拟机不提供所有帮助,需要开发人员自己实现.

**序列化ID作用:** 序列化ID起着关键的作用，它决定着是否能够成功反序列化,简单来说，java的序列化机制是通过在运行时判断类的serialVersionUID来验证版本一致性的。在进行反序列化时，JVM会把传来的字节流中的serialVersionUID与本地实体类中的serialVersionUID进行比较，如果相同则认为是一致的，便可以进行反序列化，否则就会报序列化版本不一致的异常。为了解决这种由于序列化ID不一致导致的问题, 在本地类中添加一个“serialVersionUID”变量，值保持不变，便可以进行序列化和反序列化。

**序列化注意:**

i).一个类实现了java.io.Serializable接口，那么它就可以被序列化.

ii).通过ObjectOutputStream和 ObjectInputStream对对象进行序列化及反序列化.

iii).虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，一个非常重要的一点是两个类的序列化 ID 是否一致（就是 private static final long serialVersionUID）.

iv).序列化并不保存static静态变量,因为static成员属于类级别的,而不是对象级别,序列化是保存类的对象实例信息的, 如果在一台机器上序列化是static变量能够成功序列化是因为在一台机器上JVM加载该static变量只有一次,所以它的值始终不变,当放到另一个JVM上就会得不到之前的static的值.(static final能被序列化,因为static final修饰的是常量)

v).要想将父类对象也序列化，就需要让父类也实现Serializable  接口.

vi)transient 关键字的作用是控制变量的序列化，在变量声明前加上该关键字，可以阻止该变量被序列化到文件中(可以通过自定义序列化进行实现)，在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null.

vii).服务器端给客户端发送序列化对象数据，对象中有一些数据是敏感的，比如密码字符串等，希望对该密码字段在序列化时，进行加密，而客户端如果拥有解密的密钥，只有在客户端进行反序列化时，才可以对密码进行读取，这样可以一定程度保证序列化对象的数据安全.

**自定义序列化和反序列化:**

如果被序列化的类中定义了writeObject 和 readObject 方法，虚拟机会试图调用对象类里的 writeObject 和 readObject 方法，进行用户自定义的序列化和反序列化, 如果没有这样的方法，则默认调用是 ObjectOutputStream 的 defaultWriteObject 方法以及 ObjectInputStream 的 defaultReadObject 方法. 用户自定义的 writeObject 和 readObject 方法可以允许用户控制序列化的过程，比如可以在序列化的过程中动态改变序列化的数值.

**以ArrayList为例说明是如何自定义序列化的:**

ArrayList中定义了raedObject和writeObject方法，这两个方法中定义了elementData的序列化及反序列化策略; ArrayList实际上是动态数组，每次在放满以后自动增长设定的长度值，如果数组自动增长长度设为100，而实际只放了一个元素，那就会序列化99个null元素。为了保证在序列化的时候不会将这么多null同时进行序列化，ArrayList把元素数组elementData设置为transient; writeObject方法把 elementData数组中的元素遍历的保存到输出流（ObjectOutputStream）中。readObject方法从输入流（ObjectInputStream）中读出对象并保存赋值到 elementData数组中, 从而避免elementData数组中过多的无用的null被序列化.

**自定义序列化中writeObject和readObject是如何被调用的:**

对象的序列化过程通过ObjectOutputStream和ObjectInputputStream来实现的, ObjectOutputStream的writeObject的调用栈：

writeObject ---> writeObject0 --->writeOrdinaryObject--->writeSerialData--->invokeWriteObject, 其中writeObjectMethod.invoke();是关键，通过反射的方式调用writeObjectMethod方法.

**Serializable明明就是一个空的接口，它是怎么保证只有实现了该接口的方法才能进行序列化与反序列化的:**

在类的序列化过程中，会使用instanceof关键字判断一个类是否继承了Serializable类，如果没有，则直接抛出NotSerializableException异常.

**序列化对单例的破坏:**

public class Singleton implements Serializable{

//私有构造函数,保证外界无法直接实例化

private Singleton(){}

private static class SingletonHolder{

private static final Singleton INSTANCE = new Singleton();

}

//提供外部访问

public static Singleton getInstance(){

return SingletonHolder.INSTANCE;

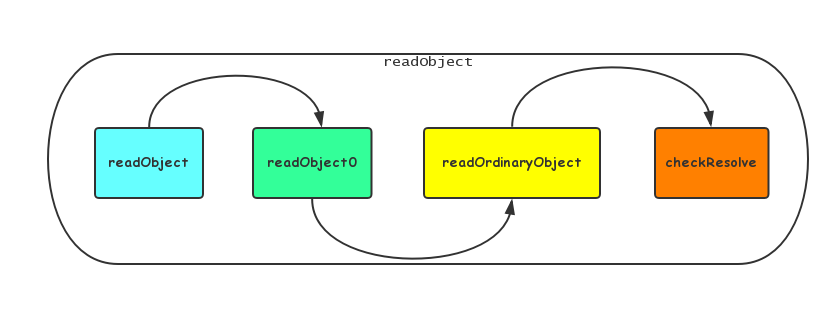
}

}

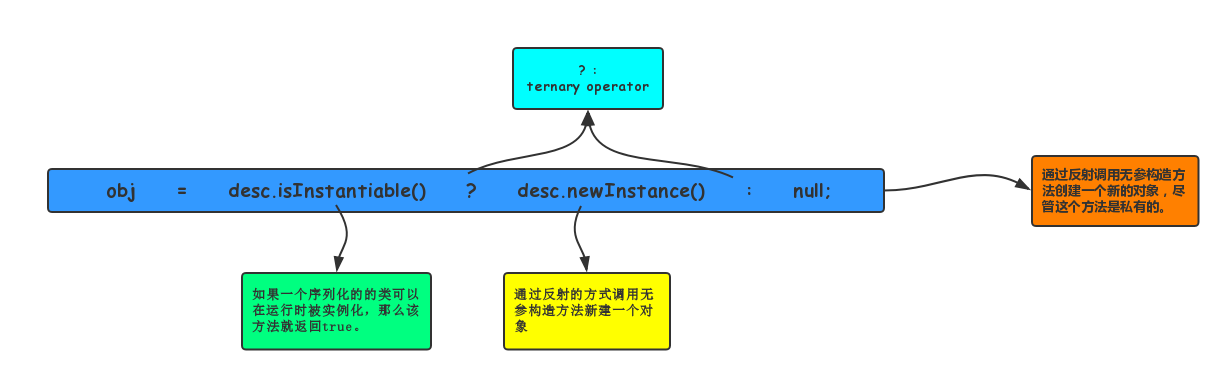
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("seriliable"));  
        oos.writeObject(SingletonDemo.getInstance());  
        File file = new File("seriliable");  
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file));  
        SingletonDemo singletonDemo = (SingletonDemo)ois.readObject();  
        System.out.println(singletonDemo == SingletonDemo.getInstance());

结果输出为false, 证明破坏了单例模式;

为什么序列化会破坏单例模式: 对象的序列化过程通过ObjectOutputStream和ObjectInputputStream来实现的，分析一下ObjectInputputStream 的readObject 方法执行情况到底是怎样的, ObjectInputputStream 的readObject 方法的调用栈如下图:



readOrdinaryObject源码中ObjectInputStream的readObject返回的对象为:



isInstantiable：如果一个serializable/externalizable的类可以在运行时被实例化，那么该方法就返回true;desc.newInstance：该方法通过反射的方式调用无参构造方法新建一个对象; 序列化会通过反射调用无参数的构造方法创建一个新的对象. readOrdinaryObject源码中会通过调用invokeReadResolve方法,然后通过反射调用要被反序列化中的readResolve方法; readOrdinaryObject的源码中hasReadResolveMethod判断如果实现了serializable 或者 externalizable接口的类中包含readResolve则返回true,即没有破坏单例模式. 为了防止序列化破坏单例模式解决办法如下:

public class Singleton implements Serializable{

//私有构造函数,保证外界无法直接实例化

private Singleton(){}

private static class SingletonHolder{

private static final Singleton INSTANCE = new Singleton();

}

//提供外部访问

public static Singleton getInstance(){

return SingletonHolder.INSTANCE;

}

private Object readResolve(){ //加上该方法防止序列化破坏单例模式

return SingletonHolder.INSTANCE;

}

}

**12).时区, 时令, 关于时间的API**

**Date:**

类Date表示特定的瞬间，精确到毫秒。获得一个表示当前时间的Date对象有两种方式：

**Java代码**

1. 1. Date date = **new** Date();
2. 2. Date date = Calendar.getInstance().getTime();

**Date**对象本身所存储的毫秒数可以通过date.getTime()方法得到；该函数返回自1970年1月1日 00:00:00 GMT以来此对象表示的毫秒数。**它与时区和地域没有关系(其实可以认为是GMT时间)，**而且还会告诉我们这个时区是否使用夏令时。有个这个信息，我们就能够继续将时区对象和日期格式化器结合在一起在其它的时区和其它的语言显示时间了.

**Calendar:**

**Calendar**的getInstance()方法有参数为TimeZone和Locale的重载，可以使用指定时区和语言环境获得一个日历。无参则使用默认时区和语言环境获得日历.

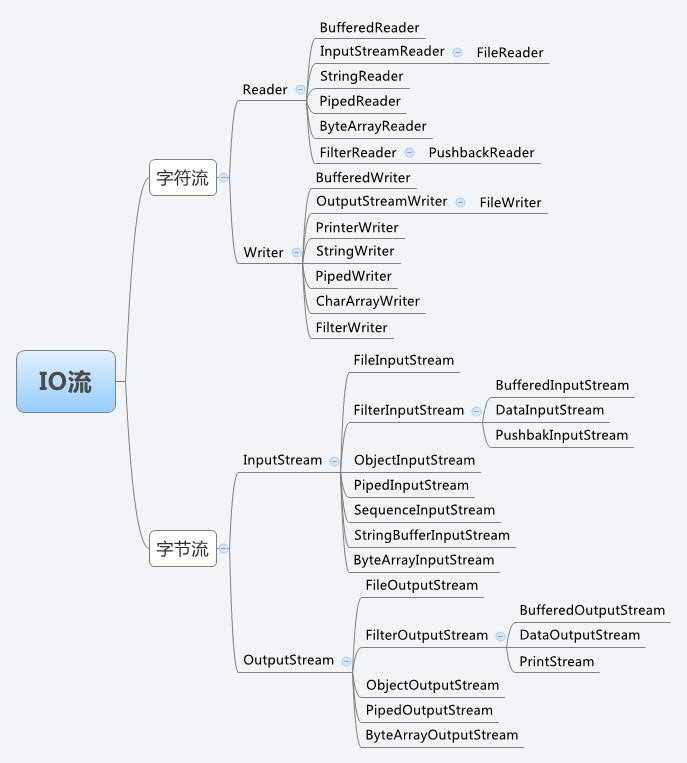
**TimeZone:**

      TimeZone对象给我们的是原始的偏移量，也就是与GMT相差的微秒数，即TimeZone表示时区偏移量，本质上以毫秒数保存与GMT的差值。

       获取TimeZone可以通过时区ID，如"America/New\_York"，也可以通过GMT+/-hh:mm来设定。例如北京时间可以表示为GMT+8:00。TimeZone.getRawOffset()方法可以用来得到当前时区的标准时间到GMT的偏移量。

**影响TimeZone的因素:** 操作系统的时区设置; 数据传输时时区设置.

**13).IO类**



**流的作用:** 流是一组有顺序的，有起点和终点的字节集合，是对数据传输的总称或抽象。即数据在两设备间的传输称为流，**流的本质是数据传输，根据数据传输特性将流抽象为各种类，方便更直观的进行数据操作.**

**IO流的分类:**根据处理数据的类型不同分为字节流和字符流;根据数据流向的不同分为输入流和输出流.

**字节流与字符流的区别:** 读写单位不同：字节流以字节（8bit）为单位，字符流以字符为单位，根据码表映射字符，一次可能读多个字节; 处理对象不同：字节流能处理所有类型的数据（如图片、avi等），而字符流只能处理字符类型的数据. 只要是处理纯文本数据，就优先考虑使用字符流。 除此之外都使用字节流.

**输入输出字节流:**

**输入字节流**: InputStream 是所有的输入字节流的父类，它是一个抽象类; ByteArrayInputStream、StringBufferInputStream、FileInputStream 是三种基本的介质流，它们分别从Byte 数组、StringBuffer、和本地文件中读取数据。PipedInputStream 是从与其它线程共用的管道中读取数据; ObjectInputStream 和所有FilterInputStream 的子类都是装饰流（装饰器模式的主角）.

**输出字节流:** OutputStream 是所有的输出字节流的父类，它是一个抽象类; ByteArrayOutputStream、FileOutputStream 是两种基本的介质流，它们分别向Byte 数组、和本地文件中写入数据。PipedOutputStream 是向与其它线程共用的管道中写入数据; ObjectOutputStream 和所有FilterOutputStream 的子类都是装饰流.

**输入输出字符流:**

**输入字符流:** Reader 是所有的输入字符流的父类，它是一个抽象类; CharReader、StringReader 是两种基本的介质流，它们分别将Char 数组、String中读取数据。PipedReader 是从与其它线程共用的管道中读取数据; BufferedReader 是一个装饰器，它和其子类负责装饰其它Reader 对象; FilterReader 是所有自定义具体装饰流的父类，其子类PushbackReader 对Reader 对象进行装饰，会增加一个行号; InputStreamReader 是一个连接字节流和字符流的桥梁，它将字节流转变为字符流。FileReader 可以说是一个达到此功能、常用的工具类，在其源代码中明显使用了将FileInputStream 转变为Reader 的方法。我们可以从这个类中得到一定的技巧。Reader 中各个类的用途和使用方法基本和InputStream 中的类使用一致.

**输出字符流:**Writer 是所有的输出字符流的父类，它是一个抽象类; CharArrayWriter、StringWriter 是两种基本的介质流，它们分别向Char 数组、String 中写入数据。PipedWriter 是向与其它线程共用的管道中写入数据; BufferedWriter 是一个装饰器为Writer 提供缓冲功能; OutputStreamWriter 是OutputStream 到Writer 转换的桥梁，它的子类FileWriter 其实就是一个实现此功能的具体类.

**字节流与字符流的转换(转换流):**

**转换流:**是字符流和字节流之间的桥梁;对读取到的字节数据经过指定的编码转换成字符;对读取到的字符数据经过指定编码转成字节.

**何时使用转换流:**当有字节与字符的转换操作时;流操作的数据需要解码和编码时.

**具体实现:** InputStreamReader字节流到字符流的桥梁;OutputStreamWriter字符流到字节流的桥梁.

**File类**: File类是对文件系统中文件以及文件夹进行封装的对象，可以通过对象的思想来操作文件和文件夹。 File类保存文件或目录的各种元数据信息，包括文件名、文件长度、最后修改时间、是否可读、获取当前文件的路径名，判断指定文件是否存在、获得当前目录中的文件列表，创建、删除文件和目录等方法.

**RandomAccessFile类:** 该对象并不是流体系中的一员，其封装了字节流，同时还封装了一个缓冲区（字符数组），通过内部的指针来操作字符数组中的数据; 该对象只能操作文件，所以构造函数接收两种类型的参数：a.字符串文件路径；b.File对象; 该对象既可以对文件进行读操作，也能进行写操作，在进行对象实例化时可指定操作模式(r,rw).

**14).socket通信**

**Socket底层原理:**

由于 TCP 提供了一种可信赖的字节流服务，任何写入 Socket 和 OutputStream 的数据副本都必须保留，直到连接的另一端将这些数据成功接收。向输出流写数据并不意味着数据实际上已经被发送——它们只是被复制到了本地缓冲区，就算在 Socket 的 OutputStream 上进行 flush()操作，也不能保证数据能够立即发送到信道。此外，字节流服务的自身属性决定了其无法保留输入流中消息的边界信息。

**数据传输的底层实现**

在使用 TCP 套接字时，需要记住的最重要的一点是：不能假设在连接的一端将数据写入输出流和在另一端从输入流读出数据之间有任何的一致性。尤其是在发送端由单个输出流的 write()方法传输的数据，可能会通过另一端的多个输入流的 read()方法获取，而一个 read()方法可能会返回多个 write()方法传输的数据。

一般来讲，我们可以认为 TCP 连接上发送的所有字节序列在某一瞬间被分成了 3 个 FIFO 队列：

* SendQ：在发送端底层实现中缓存的字节，这些字节已经写入输出流，但还没在接收端成功接收。它占用大约 37KB 内存。
* RecvQ：在接收端底层实现中缓存的字节，这些字节等待分配到接收程序——即从输入流中读取。它占用大约 25KB 内存。
* Delivered：接收者从输入流已经读取到的字节。

当我们调用 OutputStream 的 write()方法时，将向 SendQ 追加字节。TCP 协议负责将字节按顺序从 SendQ 移动到 RecvQ。这里有重要的一点需要明确：这个转移过程无法由用户程序控制或直接观察到，并且在块中发生，这些块的大小在一定程度上独立于传递给 write()方法的缓冲区大小。

接收程序从 Socket 的 InputStream 读取数据时，字节就从 RecvQ 移动到 Delivered 中，而转移的块的大小依赖于 RecvQ 中的数据量和传递给 read()方法的缓冲区的大小.

TCP 协议提供面向连接的服务，通过它建立的是可靠地连接。Java 为 TCP 协议提供了两个类：Socke 类和 ServerSocket 类。一个 Socket 实例代表了 TCP 连接的一个客户端，而一个 ServerSocket 实例代表了 TCP 连接的一个服务器端，一般在 TCP Socket 编程中，客户端有多个，而服务器端只有一个，客户端 TCP 向服务器端 TCP 发送连接请求，服务器端的 ServerSocket 实例则监听来自客户端的 TCP 连接请求，并为每个请求创建新的 Socket 实例，由于服务端在调用 accept（）等待客户端的连接请求时会阻塞，直到收到客户端发送的连接请求才会继续往下执行代码，因此要为每个 Socket 连接开启一个线程。服务器端要同时处理 ServerSocket 实例和 Socket 实例，而客户端只需要使用 Socket 实例。另外，每个 Socket 实例会关联一个 InputStream 和 OutputStream 对象，我们通过将字节写入套接字的 OutputStream 来发送数据，并通过从 InputStream 来接收数据.

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintStream;

import java.net.Socket;

import java.net.SocketTimeoutException;

public class Client1 {

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

//客户端请求与本机在20006端口建立TCP连接

Socket client = new Socket("127.0.0.1", 20006);

client.setSoTimeout(10000);

//获取键盘输入

BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

//获取Socket的输出流，用来发送数据到服务端

PrintStream out = new PrintStream(client.getOutputStream());

//获取Socket的输入流，用来接收从服务端发送过来的数据

BufferedReader buf = new BufferedReader(new InputStreamReader(client.getInputStream()));

boolean flag = true;

while(flag){

System.out.print("输入信息：");

String str = input.readLine();

//发送数据到服务端

out.println(str);

if("bye".equals(str)){

flag = false;

}else{

try{

//从服务器端接收数据有个时间限制（系统自设，也可以自己设置），超过了这个时间，便会抛出该异常

String echo = buf.readLine();

System.out.println(echo);

}catch(SocketTimeoutException e){

System.out.println("Time out, No response");

}

}

}

input.close();

if(client != null){

//如果构造函数建立起了连接，则关闭套接字，如果没有建立起连接，自然不用关闭

client.close(); //只关闭socket，其关联的输入输出流也会被关闭

}

}

}

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.PrintStream;

import java.net.Socket;

/\*\*

\* 该类为多线程类，用于服务端

\*/

public class ServerThread implements Runnable {

private Socket client = null;

public ServerThread(Socket client){

this.client = client;

}

@Override

public void run() {

try{

//获取Socket的输出流，用来向客户端发送数据

PrintStream out = new PrintStream(client.getOutputStream());

//获取Socket的输入流，用来接收从客户端发送过来的数据

BufferedReader buf = new BufferedReader(new InputStreamReader(client.getInputStream()));

boolean flag =true;

while(flag){

//接收从客户端发送过来的数据

String str = buf.readLine();

if(str == null || "".equals(str)){

flag = false;

}else{

if("bye".equals(str)){

flag = false;

}else{

//将接收到的字符串前面加上echo，发送到对应的客户端

out.println("echo:" + str);

}

}

}

out.close();

client.close();

}catch(Exception e){

e.printStackTrace();

}

}

}

import java.net.ServerSocket;

import java.net.Socket;

public class Server1 {

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

//服务端在20006端口监听客户端请求的TCP连接

ServerSocket server = new ServerSocket(20006);

Socket client = null;

boolean f = true;

while(f){

//等待客户端的连接，如果没有获取连接

client = server.accept();

System.out.println("与客户端连接成功！");

//为每个客户端连接开启一个线程

new Thread(new ServerThread(client)).start();

}

server.close();

}

}

**15).深拷贝与浅拷贝**

在 Java 中，除了**基本数据类型**（元类型）之外，还存在 **类的实例对象** 这个引用数据类型。而一般使用 『 **=** 』号做赋值操作的时候。对于基本数据类型，实际上是拷贝的它的值，但是对于对象而言，其实赋值的只是这个对象的引用，将原对象的引用传递过去，他们实际上还是指向的同一个对象; 而浅拷贝和深拷贝就是在这个基础之上做的区分，如果在拷贝这个对象的时候，只对基本数据类型进行了拷贝，而对引用数据类型只是进行了引用的传递，而没有真实的创建一个新的对象，则认为是浅拷贝。反之，在对引用数据类型进行拷贝的时候，创建了一个新的对象，并且复制其内的成员变量，则认为是深拷贝; 所谓的浅拷贝和深拷贝，只是在拷贝对象的时候，对 **类的实例对象** 这种引用数据类型的不同操作而已.

**浅拷贝：**对基本数据类型进行值传递，对引用数据类型进行引用传递般的拷贝，此为浅拷贝;

**深拷贝：**对基本数据类型进行值传递，对引用数据类型，创建一个新的对象，并复制其内容，此为深拷贝.

**如何实现深拷贝:**

i)序列化（serialization）这个对象，再反序列化回来，就可以得到这个新的对象，序列化的规则需要我们自己来写;

ii)继续利用 clone() 方法，既然 clone() 方法，是我们来重写的，实际上我们可以对其内的引用类型的变量，再进行一次 clone():

例如父类FatherClass中有一个引用对象SonClass, 在对FatherClass进行clone时, 如果只在FatherClass中重写clone方法,那么对FatherClass进行的拷贝属于浅拷贝:

public class FatherClass implements Cloneable {

public String name;

public int age;

public SonClass sonClass;

public Object clone(){

try{

//浅拷贝

//return super.clone();

//结合对SonClass中clone方法的重写实现深拷贝

FatherClass fatherClass = (FatherClass)super.clone();

fatherClass.sonClass = (SonClass)this.sonClass.clone();

return fatherClass;

}catch(CloneNotSupportedException e){}

return null;

}

}

此时如果在SonClass中继续重写clone, 则能完成对FatherClass的深拷贝:

public class SonClass implements Cloneable{

public int age;

public String name;

public Object clone(){

try{

return super.clone();

}catch(CloneNotSupportedException e){}

return null;

}

}

浅拷贝和深拷贝只是相对的，如果一个对象内部只有基本数据类型，那用 clone() 方法获取到的就是这个对象的深拷贝，而如果其内部还有引用数据类型，那用 clone() 方法就是一次浅拷贝的操作.

**16).static关键字的作用,为什么方法覆盖不能覆盖一个static修饰的方法**

**static的作用:** 方便在没有创建对象的情况下来进行调用（方法/变量）. 被static关键字修饰的方法或者变量不需要依赖于对象来进行访问，只要类被加载了，就可以通过类名去进行访问, static可以用来修饰类的成员方法、类的成员变量，另外可以编写static代码块来优化程序性能.

**static方法:** static方法一般称作静态方法，由于静态方法不依赖于任何对象就可以进行访问，因此对于静态方法来说，是没有this的，因为它不依附于任何对象，既然都没有对象，就谈不上this了。并且由于这个特性，在静态方法中不能访问类的非静态成员变量和非静态成员方法，因为非静态成员方法/变量都是必须依赖具体的对象才能够被调用, 但是要注意的是，虽然在静态方法中不能访问非静态成员方法和非静态成员变量，但是在非静态成员方法中是可以访问静态成员方法/变量的.

**实例构造方法是不是静态方法?**

从Java语言规范对“方法”的定义来说，构造器根本不是“方法”；其次，实例构造器有一个隐式参数，“this”，在实例构造器中可以访问“this”，可以通过“this”访问到正在初始化的对象实例的所有实例成员; 实例构造器无法被隐藏或覆写，不参与多态，因而可以做静态绑定。从这个意义上可以认为实例构造器是“静态”的，但这种用法与Java语言定义的“静态方法”是两码事; Java语言中，实例构造器只能在new表达式（或别的构造器）中被调用，不能通过方法调用表达式来调用。new表达式作为一个整体保证了对象的创建与初始化是打包在一起进行的，不能分开进行；但实例构造器只负责对象初始化的部分，“创建对象”的部分是由new表达式本身保证的; Java的实例构造器只负责初始化，不负责创建对象；Java虚拟机的字节码指令的设计也反映了这一点，有一个new指令专门用于创建对象实例，而调用实例构造器则使用invokespecial指令, “this”是作为实例构造器的第一个实际参数传入的; 静态方法中不能使用this，而构造器中可以使用this关键字,this是指调用当前方法的对象，而静态方法不属于任何对象.

**static变量:** static变量也称作静态变量，静态变量和非静态变量的区别是：静态变量被所有的对象所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始化。而非静态变量是对象所拥有的，在创建对象的时候被初始化，存在多个副本，各个对象拥有的副本互不影响; static成员变量的初始化顺序按照定义的顺序进行初始化.

**static代码块:** static关键字还有一个比较关键的作用就是 用来形成静态代码块以优化程序性能。static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次.

**static关键字的误区:** 静态成员变量虽然独立于对象，但是不代表不可以通过对象去访问，所有的静态方法和静态变量都可以通过对象访问（只要访问权限足够）, 可以通过this访问static变量, static是不允许用来修饰局部变量.

**17).Java中用到了哪些设计模式?动态代理和静态代理原理以及代码实现**

**代理:** 为某个对象提供一个代理，以控制对这个对象的访问。 代理类和委托类有共同的父类或父接口，这样在任何使用委托类对象的地方都可以用代理对象替代。代理类负责请求的预处理、过滤、将请求分派给委托类处理、以及委托类执行完请求后的后续处理

**i)静态代理:** 在程序运行前就已经存在代理类的字节码文件，代理类和委托类的关系在运行前就确定了.

**代理接口:**

1. /\*\*
2. \* 代理接口。处理给定名字的任务。
3. \*/
4. **public** **interface** Subject {
5. /\*\*
6. \* 执行给定名字的任务。
7. \* @param taskName 任务名
8. \*/
9. **public** **void** dealTask(String taskName);
10. }

**委托类,具体处理业务:**

1. /\*\*
2. \* 真正执行任务的类，实现了代理接口。
3. \*/
4. **public** **class** RealSubject **implements** Subject {
6. /\*\*
7. \* 执行给定名字的任务。这里打印出任务名，并休眠500ms模拟任务执行了很长时间
8. \* @param taskName
9. \*/
10. @Override
11. **public** **void** dealTask(String taskName) {
12. System.out.println("正在执行任务："+taskName);
13. **try** {
14. Thread.sleep(500);
15. } **catch** (InterruptedException e) {
16. e.printStackTrace();
17. }
18. }
19. }

**静态代理类:**

1. /\*\*
2. \*　代理类，实现了代理接口。
3. \*/
4. **public** **class** ProxySubject **implements** Subject {
5. //代理类持有一个委托类的对象引用
6. **private** Subject delegate;
8. **public** ProxySubject(Subject delegate) {
9. **this**.delegate = delegate;
10. }
12. /\*\*
13. \* 将请求分派给委托类执行，记录任务执行前后的时间，时间差即为任务的处理时间
14. \*
15. \* @param taskName
16. \*/
17. @Override
18. **public** **void** dealTask(String taskName) {
19. **long** stime = System.currentTimeMillis();
20. //将请求分派给委托类处理
21. delegate.dealTask(taskName);
22. **long** ftime = System.currentTimeMillis();
23. System.out.println("执行任务耗时"+(ftime - stime)+"毫秒");
25. }
26. }

**客户类:**

1. **public** **class** Client1 {
3. **public** **static** **void** main(String[] args) {
4. Subject proxy = SubjectStaticFactory.getInstance();
5. proxy.dealTask("DBQueryTask");
6. }
8. }

**静态代理的优点:** 业务类只需要关注业务逻辑本身，保证了业务类的重用性。这是代理的共有优点.

**静态代理的缺点:** 代理对象的一个接口只服务于一种类型的对象，如果要代理的方法很多，势必要为每一种方法都进行代理，静态代理在程序规模稍大时就无法胜任了; 如果接口增加一个方法，除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。增加了代码维护的复杂度.

**ii).动态代理**: 动态代理类的源码是在程序运行期间由JVM根据反射等机制动态的生成，所以不存在代理类的字节码文件。代理类和委托类的关系是在程序运行时确定。

**使用到的类:**

java.lang.reflect.Proxy,这是 Java 动态代理机制生成的所有动态代理类的父类，它提供了一组静态方法来为一组接口动态地生成代理类及其对象

1. // 方法 1: 该方法用于获取指定代理对象所关联的调用处理器
2. **static** InvocationHandler getInvocationHandler(Object proxy)
3. // 方法 2：该方法用于获取关联于指定类装载器和一组接口的动态代理类的类对象
4. **static** Class getProxyClass(ClassLoader loader, Class[] interfaces)
5. // 方法 3：该方法用于判断指定类对象是否是一个动态代理类
6. **static** **boolean** isProxyClass(Class cl)
7. // 方法 4：该方法用于为指定类装载器、一组接口及调用处理器生成动态代理类实例
8. **static** Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h)

java.lang.reflect.InvocationHandler, 这是调用处理器接口，它自定义了一个 invoke 方法，用于集中处理在动态代理类对象上的方法调用，通常在该方法中实现对委托类的代理访问。每次生成动态代理类对象时都要指定一个对应的调用处理器对象.

1. // 该方法负责集中处理动态代理类上的所有方法调用。第一个参数既是代理类实例，第二个参数是被调用的方法对象
2. // 第三个方法是调用参数。调用处理器根据这三个参数进行预处理或分派到委托类实例上反射执行
3. Object invoke(Object proxy, Method method, Object[] args)

java.lang.ClassLoader, 这是类装载器类，负责将类的字节码装载到 Java 虚拟机（JVM）中并为其定义类对象，然后该类才能被使用。Proxy 静态方法生成动态代理类同样需要通过类装载器来进行装载才能使用，它与普通类的唯一区别就是其字节码是由 JVM 在运行时动态生成的而非预存在于任何一个 .class 文件中.

**动态代理的步骤:**

a. 实现InvocationHandler接口创建自己的调用处理器  
b. 给Proxy类提供ClassLoader和代理接口类型数组创建动态代理类  
c. 以调用处理器类型为参数，利用反射机制得到动态代理类的构造函数  
d. 以调用处理器对象为参数，利用动态代理类的构造函数创建动态代理类对象

1. // InvocationHandlerImpl 实现了 InvocationHandler 接口，并能实现方法调用从代理类到委托类的分派转发
2. // 其内部通常包含指向委托类实例的引用，用于真正执行分派转发过来的方法调用
3. InvocationHandler handler = **new** InvocationHandlerImpl(..);
4. // 通过 Proxy 为包括 Interface 接口在内的一组接口动态创建代理类的类对象
5. Class clazz = Proxy.getProxyClass(classLoader, **new** Class[] { Interface.**class**, ... });
6. // 通过反射从生成的类对象获得构造函数对象
7. Constructor constructor = clazz.getConstructor(**new** Class[] { InvocationHandler.**class** });
8. // 通过构造函数对象创建动态代理类实例
9. Interface Proxy = (Interface)constructor.newInstance(**new** Object[] { handler });

**简化后的动态代理步骤:**

1. // InvocationHandlerImpl 实现了 InvocationHandler 接口，并能实现方法调用从代理类到委托类的分派转发
2. InvocationHandler handler = **new** InvocationHandlerImpl(..);
3. // 通过 Proxy 直接创建动态代理类实例
4. Interface proxy = (Interface)Proxy.newProxyInstance( classLoader, **new** Class[] { Interface.**class** },  handler );

**动态代理实例:**

**a.创建自定义的调用处理器:**

1. /\*\*
2. \* 动态代理类对应的调用处理程序类
3. \*/
4. **public** **class** SubjectInvocationHandler **implements** InvocationHandler {
5. //代理类持有一个委托类的对象引用
6. **private** Object delegate;
7. **public** SubjectInvocationHandler(Object delegate) {
8. **this**.delegate = delegate;
9. }
10. @Override
11. **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {
12. **long** stime = System.currentTimeMillis();
13. //利用反射机制将请求分派给委托类处理。Method的invoke返回Object对象作为方法执行结果。
14. //因为示例程序没有返回值，所以这里忽略了返回值处理
15. method.invoke(delegate, args);
16. **long** ftime = System.currentTimeMillis();
17. System.out.println("执行任务耗时"+(ftime - stime)+"毫秒");
18. **return** **null**;
19. }
20. }

**b.生成动态代理对象的工厂,工厂方法列举出如何生成动态代理类对象的步骤:**

1. /\*\*
2. \* 生成动态代理对象的工厂.
3. \*/
4. **public** **class** DynProxyFactory {
5. //客户类调用此工厂方法获得代理对象。
6. //对客户类来说，其并不知道返回的是代理类对象还是委托类对象。
7. **public** **static** Subject getInstance(){
8. Subject delegate = **new** RealSubject();
9. InvocationHandler handler = **new** SubjectInvocationHandler(delegate);
10. Subject proxy = **null**;
11. proxy = (Subject)Proxy.newProxyInstance(
12. delegate.getClass().getClassLoader(),
13. delegate.getClass().getInterfaces(),
14. handler);
15. **return** proxy;
16. }
17. }

**c.动态代理客户类:**

1. **public** **class** Client {
2. **public** **static** **void** main(String[] args) {
3. Subject proxy = DynProxyFactory.getInstance();
4. proxy.dealTask("DBQueryTask");
5. }
6. }

**动态代理的优点:** 动态代理与静态代理相比较，最大的好处是接口中声明的所有方法都被转移到调用处理器一个集中的方法中处理（InvocationHandler.invoke）。这样，在接口方法数量比较多的时候，我们可以进行灵活处理，而不需要像静态代理那样每一个方法进行中转。

**动态代理的缺点:** 无法摆脱仅支持 interface 代理的桎梏, 有一个共同的父类叫 Proxy。Java 的继承机制注定了这些动态代理类们无法实现对 class 的动态代理，原因是多继承在 Java 中本质上就行不通.

**18).Java中的权限问题, 接口里面包含哪些东西?接口里方法的默认权限是什么?**

**public：**权限最大，不受类、包等的限制，都可以访问。

**protected**：次于public，限制之处在于如果不再同一个包中，只有和它存在继承关系的子类才可以访问它。

**default：**进一步受限，必须是同一个包才能访问。

**private：**这个就将权限限制在了类中，只有同一个类中的成员才能访问

接口中可以有方法,成员变量,成员变量默认修饰符是static final的, 方法默认是public abstract的, 在jdk8中接口中有default方法,可以进行实现.

**19).Java中类型转换的问题, ++操作问题, char类型与short,byte之间的转换注意点**

i)+0与-0在浮点类型变量存储中，符号位是不同的。当-0和+0参与浮点类型的相关运算（例如相除与求余运算）时，可以产生不同的结果;

ii).浮点的相除与求余运算不同与整型的相除与求余运算，当除数为0时，浮点运算不会产生ArithmeticException异常

iii).  i++和++i其实都是先＋1，再赋值。++i，没什么好说的；i++,以j=i++;为例在底层的实现是：temp = i;i = i + 1; j = temp; 所以，i=15;i=i++;这个表达式的结果是15.（因为加一之后又执行了一次赋值，从16变回15）

iv). float类型可以保留7～8个有效数字，而double类型可以保留15～16个有效数字，因而当int类型或long类型数值多于double或float地有效数字时，该值的一些最低有效位就会丢失，从而造成精度丢失，这时，就会采用IEEE754最近舍入模式，提取与该整型值最接近的浮点值。尽管整型向浮点型的转换属于扩展转换，但当数值很大或很小（绝对值很大）时，就会产生一定的精度丢失

v).  0.1+0.2不等于0.3.System.out.println((double)0.1+(double)0.2);这条语句的输出结果是0.30000000000000004。因为计算机使用二进制来存储数据，而很多小数都不能够准确地使用二进制来表示（事实上，大多数地小数都是近似的），就像使用十进制小数不能准确地表示1/3这样地分数一样。大多数地浮点型，在计算机中只是近似地存储其值，而不像整型那样准确地存储。又例，这是一个死循环：for(float f = 10.1f;f != 11;f+=0.1f){}

vi). 注意char类型，这是一个无符号类型。因此，char与short或char与byte之间的转换必须显示地使用类型转换。 从byte到char的转换为扩展收缩转换，该转换比较特殊，即先将byte扩展转换到int，然后再收缩到char; 在整型数据间的扩展转换中，如果操作数是char类型（无符号类型），则进行无符号扩展，扩展位为0.如果操作数是byte，short或int（有符号类型），则进行有符号扩展，扩展位为该变量的符号位。

vii). 当short，byte，char参加运算时，结果为int型，而非与较高的类型相同。如果变量是byte，short，byte类型，当对其赋予编译时期的常量，而该常量又没有超过变量的取值范围时，编译器就可以进行隐式的收缩转换。这种隐式的收缩转换是安全的，因为该收缩转换只适用于变量的赋值，而不适用于方法调用语句，即不适用于方法调用时的参数传递。（详见java中默认类型转换的小问题）.

**20).装箱与拆箱**

当==或!=运算符的两个操作数的类型一个是基本数据类型，另一个是包装类引用类型时，将引用类型拆箱转换为基本数据类型，然后比较两个基本数据类型的值是否相等.

装箱过程是通过调用包装器的valueOf方法实现的，而拆箱过程是通过调用包装器的 xxxValue方法实现的。（xxx代表对应的基本数据类型）.

**21).语句执行顺序**

class Parent{

static String name = "hello";

{

System.out.println("3 parent block");

}

static {

System.out.println("1 parent static block");

}

public Parent(){

System.out.println("4 parent constructor");

}

}

class Child extends Parent{

static String childName = "hello";

{

System.out.println("5 child block");

}

static {

System.out.println("2 child static block");

}

public Child(){

System.out.println("6 child constructor");

}

}

public class StaticIniBlockOrderTest {

public static void main(String[] args) {

new Child();//语句(\*)

}

}

结果:

1 parent static block  
2 child static block  
3 parent block  
4 parent constructor  
5 child block  
6 child constructor

i）执行父类静态的内容，父类静态的内容执行完毕后，接着去执行子类的静态的内容；

   ii）当子类的静态内容执行完毕之后，再去看父类有没有非静态代码块，如果有就执行父类的非静态代码块，父类的非静态代码块执行完毕，接着执行父类的构造方法；

  iii）父类的构造方法执行完毕之后，它接着去看子类有没有非静态代码块，如果有就执行子类的非静态代码块。子类的非静态代码块执行完毕再去执行子类的构造方法。

总之一句话，静态代码块内容先执行，接着执行父类非静态代码块和构造方法，然后执行子类非静态代码块和构造方法。  
而且子类的构造方法，不管这个构造方法带不带参数，默认的它都会先去寻找父类的不带参数的构造方法。如果父类没有不带参数的构造方法，那么子类必须用supper关键子来调用父类带参数的构造方法，否则编译不能通过.

**22).引用的种类:强引用(StrongReference), 软引用(SoftReference), 弱引用(WeakReference), 虚引用(PhantomReference)**

**强引用(StrongReference):** 强引用在程序代码中普遍存在, 例如: Object obj = new Object(); String str = “hello”; 等都是强引用, 只要某个对象有强引用与之关联, JVM必定不会回收该对象,即使内存不足,抛出OutOfMemory的错误也不会回收这种对象. 如果想要中断强引用和某个对象之间的关系,可以显示的将引用赋值为null, 从而使得JVM在合适的时候对该对象进行回收.

**软引用(SoftReference):** 软引用是用来描述一些有用但并不必须的对象, 在Java中用Java.lang.ref.SoftReference来表示,对于软引用关联的对象,只有在内存不足时JVM才会回收该对象,因此可以很好的解决OOM问题, 这一特性适合实现缓存,比如网页缓存,图片缓存等.

**弱引用(WeakReference):**用来描述非必须对象,当JVM进行垃圾回收时,无论内存是否充足,都会回收被弱引用关联的对象, 在java.lang.ref.WeakReference类来表示.

**虚引用(PhantomReference):** 虚引用并不影响对象的生命周期,在java.lang.ref.PhantomReference类汇总, 如果一个对象和虚引用关联,则跟没有引用与之关联一样,在任何时候都可以被回收.

**软引用于弱引用的对比**:对于强引用,平时用到最多, 对于软引用和弱引用, 都是用来描述非必须对象的, 但是被软引用关联的对象只有在内存不足时才会被回收,而被弱引用关联的对象在JVM进行垃圾回收时会被回收, 针对这种特性, 软引用适合用来进行缓存, 当内存不够使就让JVM进行回收,弱引用能用来在回调函数中防止内存泄漏, 因为回调函数往往是匿名内部类,隐式保存有对外部类的引用,所以如果回调函数是在另一个线程被调用,这时如果要回收外部类,那么就会导致内存泄漏,因为匿名内部类保存有对外部类的强引用.(ThreadLocal中的ThreadLocalMap的Entry就是一种弱引用的体现,防止内存泄漏)

Java8新特性：

**1).Lambda表达式,为什么要引入Lambda表达式?主要解决Java之前的哪些问题?**

在Java平台上引入lambda表达式的一个最主要原因是方便多线程的处理集合。目前，对于list和set等集合类典型的操作方式是使用迭代器。如果需要并行的处理集合，需要编写客户端代码来处理，而不是集合本身处理; 引入闭包的目的就是为了给集合提供函数式的方法，方便处理。这样所带来的好处是,集合能够自己组织他们的元素，而不需要客户端代码来处理。但是，为了使得客户端代码能够使用这一改进，必须给集合添加函数式方法。当前通用的做法是使用匿名类。但是声明匿名类带来了过多的标准化代码。

Lambda表达式的结构:

* 一个 Lambda 表达式可以有零个或多个参数
* 参数的类型既可以明确声明，也可以根据上下文来推断。例如：(int a)与(a)效果相同
* 所有参数需包含在圆括号内，参数之间用逗号相隔。例如：(a, b) 或 (int a, int b) 或 (String a, int b, float c)
* 空圆括号代表参数集为空。例如：() -> 42
* 当只有一个参数，且其类型可推导时，圆括号（）可省略。例如：a -> return a\*a
* Lambda 表达式的主体可包含零条或多条语句
* 如果 Lambda 表达式的主体只有一条语句，花括号{}可省略。匿名函数的返回类型与该主体表达式一致
* 如果 Lambda 表达式的主体包含一条以上语句，则表达式必须包含在花括号{}中（形成代码块）。匿名函数的返回类型与代码块的返回类型一致，若没有返回则为空

例子:

(int a, int b) -> { return a + b; }

() -> System.out.println("Hello World");

(String s) -> { System.out.println(s); }

() -> 42

() -> { return 3.1415 };

**Lambda表达式的使用:**

a).创建线程

// Java 8之前：

new Thread(new Runnable() {

    @Override

    public void run() {

    System.out.println("Before Java8, too much code for too little to do");

    }

}).start();

//Java 8方式：

new Thread( () -> System.out.println("In Java8, Lambda expression rocks !!") ).start();

b).列表迭代:

// Java 8之前：

List features = Arrays.asList("Lambdas", "Default Method", "Stream API", "Date and Time API");

for (String feature : features) {

    System.out.println(feature);

}

// Java 8之后：

List features = Arrays.asList("Lambdas", "Default Method", "Stream API", "Date and Time API");

features.forEach(n -> System.out.println(n));

// 使用Java 8的方法引用更方便，方法引用由::双冒号操作符标示，

// 看起来像C++的作用域解析运算符

features.forEach(System.out::println);

**Lambda表达式与匿名内部类的区别:**

1). 对于匿名类，关键词 this 解读为匿名类，而对于 Lambda 表达式，关键词 this 解读为写就 Lambda 的外部类;

b) 匿名内部类可以为抽象类甚至普通类创建实例；但lambda表达式只能为函数式接口创建实例;

c) 匿名内部类实现的抽象方法可以允许调用接口中定义默认方法。但lambda表达式的代码块不允许调用接口中定义默认方法;

d).编译的角度的区别: java编译器会将lambda表达式编译成类的私有方法，使用了java7的invokedynamic字节码指令动态绑定方法（也就是一个类可以拥有不仅仅是自己的方法还可以动态绑定其他类的方法）。编译器编译匿名内部类只是将其编译成**外部类&numble**的新类.

**2).函数式编程, 函数式接口**

在 Java 中，Marker（标记）类型的接口是一种没有方法或属性声明的接口，简单地说，marker 接口是空接口。相似地，函数式接口是只包含一个抽象方法声明的接口。每个 Lambda 表达式都能隐式地赋值给函数式接口，例如，我们可以通过 Lambda 表达式创建 Runnable 接口的引用。

new Thread(

() -> System.out.println("hello world")

).start();

在上面的代码中，编译器会自动推断：根据线程类的构造函数签名 public Thread(Runnable r) { }，将该 Lambda 表达式赋给 Runnable 接口。

[@FunctionalInterface](http://download.java.net/jdk8/docs/api/java/lang/FunctionalInterface.html) 是 Java 8 新加入的一种接口，用于指明该接口类型声明是根据 Java 语言规范定义的函数式接口。Java 8 还声明了一些 Lambda 表达式可以使用的函数式接口，当你注释的接口不是有效的函数式接口时，可以使用 @FunctionalInterface 解决编译层面的错误。以下是一种自定义的函数式接口：

@FunctionalInterface

public interface WorkerInterface {

public void doSomeWork();

}

根据定义，函数式接口只能有一个抽象方法，如果你尝试添加第二个抽象方法，将抛出编译时错误。

a)函数式接口里是可以包含默认方法，因为默认方法不是抽象方法，其有一个默认实现，所以是符合函数式接口的定义的;

b)函数式接口里是可以包含静态方法，因为静态方法不能是抽象方法，是一个已经实现了的方法，所以是符合函数式接口的定义的;

c)函数式接口里是可以包含Object里的public方法，这些方法对于函数式接口来说，不被当成是抽象方法（虽然它们是抽象方法）；因为任何一个函数式接口的实现，默认都继承了Object类，包含了来自java.lang.Object里对这些抽象方法的实现. Java 8 中将并行进行了优化，我们可以很容易的对数据进行并 行操作。Stream API 可以声明性地通过 parallel() 与 sequential() 在并行流与顺序流之间进行切换。

**3).使用流处理数据，并行流的使用**

**并行流 :**就是把一个内容分成多个数据块，并用不同的线程分别处理每个数据块的流. 并发流使用的默认线程数等于你机器的处理器核心数。

fork/join框架是jdk1.7引入的，java8的stream多线程并行流正是以这个框架为基础的; fork/join框架的目的是以递归方式将可以并行的任务拆分成更小的任务，然后将每个子任务的结果合并起来生成整体结果。它是ExecutorService接口的一个实现，它把子任务分配线程池（ForkJoinPool）中的工作线程。要把任务提交到这个线程池，必须创建RecursiveTask<R>的一个子类，如果任务不返回结果则是RecursiveAction的子类。

将一个顺序执行的流转变成一个并发的流只要调用 parallel()方法

**public static long**parallelSum(**long**n){  
    **return**Stream.iterate(1L, i -> i +1).limit(n).parallel().reduce(0L,Long::sum);

}

**将一个并发流转成顺序的流只要调用sequential()方法**

**stream.parallel() .filter(...) .sequential() .map(...) .parallel() .reduce();**

**4).默认方法的使用, 接口的改变**

public interface A {

    default void foo(){

       System.out.println("Calling A.foo()");

    }

}

public class Clazz implements A {

    public static void main(String[] args){

       Clazz clazz = new Clazz();

       clazz.foo();//调用A.foo()

    }

}

Java8中的接口与抽象类的区别:

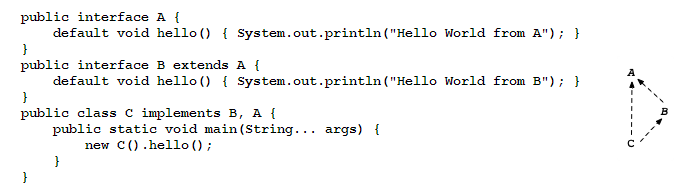
|  |  |
| --- | --- |
| 相同点 | 不同点 |
| 1.都是抽象类型；  2.都可以有实现方法（以前接口不行）；  3.都可以不需要实现类或者继承者去实现所有方法，（以前不行，现在接口中默认方法不需要实现者实现） | 1.抽象类不可以多重继承，接口可以（无论是多重类型继承还是多重行为继承）；  2.抽象类和接口所反映出的设计理念不同。其实抽象类表示的是"is-a"关系，接口表示的是"like-a"关系；  3.接口中定义的变量默认是public static final 型，且必须给其初值，所以实现类中不能改变其值；抽象类中的变量默认是 friendly 型，其值可以在子类中重新定义，也可以重新赋值。 |

多重继承的冲突说明

由于同一个方法可以从不同接口引入，自然而然的会有冲突的现象，默认方法判断冲突的规则如下:

a).一个声明在类里面的方法优先于任何默认方法（classes always win）;

b).否则，则会优先选取最具体的实现，比如下面的例子 B重写了A的hello方法.



**5).Optional类的使用**

Java 8引入了一个新类叫做java.util.Optional<T>,这个类的设计的灵感来源于Haskell语言和Scala语言。这个类可以包含了一个任意值，像下面图和代码表示的那样。你可以把Optional看做是一个有可能包含了值的值，如果Optional不包含值那么它就是空的.  
Optional类并不是要取代null引用。相反地，是为了让设计的API更容易被理解，当你看到一个函数的签名时，你就可以判断要传递给这个函数的值是不是有可能不存在。这就促使你要打开Optional类来处理缺失值的状况了。当我们非常非常的明确将要传给 Optional.of(obj) 的 obj 参数不可能为 null 时, 比如它是一个刚 new 出来的对象(Optional.of(new User(...))), 或者是一个非 null 常量时;  当想为 obj 断言不为 null 时, 即我们想在万一 obj 为 null 立即报告 NullPointException 异常, 立即修改, 而不是隐藏空指针异常时, 我们就应该果断的用 Optional.of(obj) 来构造 Optional 实例, 而不让任何不可预计的 null 值有可乘之机隐身于 Optional 中.  
a)Optional.of()或者Optional.ofNullable()：创建Optional对象，差别在于of不允许参数是null，而ofNullable则无限制;

b)Optional.empty()：所有null包装成的Optional对象;

c)isPresent()：判断值是否存在;

d)ifPresent(Consumer consumer)：如果option对象保存的值不是null，则调用consumer对象，否则不调用;

e)orElse(value)：如果optional对象保存的值不是null，则返回原来的值，否则返回value;

f)orElseGet(Supplier supplier)：功能与orElse一样，只不过orElseGet参数是一个对象;

g)orElseThrow()：值不存在则抛出异常，存在则什么不做;

h)filter(Predicate)：判断Optional对象中保存的值是否满足Predicate，并返回新的Optional;

i)map(Function)：对Optional中保存的值进行函数运算，并返回新的Optional(可以是任何类型);

j)flatMap()：功能与map()相似, flatMap方法与map方法类似，区别在于mapping函数的返回值不同。map方法的mapping函数返回值可以是任何类型T，而flatMap方法的mapping函数必须是Optional.

**6).CompletableFuture**

**Future的缺点:** i) Future虽然可以实现获取异步执行结果的需求，但是它没有提供通知的机制，我们无法得知Future什么时候完成;ii). 要么使用阻塞，在future.get()的地方等待future返回的结果，这时又变成同步操作。要么使用isDone()轮询地判断Future是否完成，这样会耗费CPU的资源.

**Java 8新增的CompletableFuture类**: CompletableFuture能够将回调放到与任务不同的线程中执行，也能将回调作为继续执行的同步函数，在与任务相同的线程中执行。它避免了传统回调最大的问题，那就是能够将控制流分离到不同的事件处理器中。CompletableFuture弥补了Future模式的缺点。在异步的任务完成后，需要用其结果继续操作时，无需等待。可以直接通过thenAccept、thenApply、thenCompose等方式将前面异步处理的结果交给另外一个异步事件处理线程来处理。CompletableFuture背后依靠的是fork/join框架来启动新的线程实现异步与并发。当然，也能通过指定线程池来做这些事情.

**与Future的区别:**

future.get()在等待执行结果时，程序会一直block，如果此时调用complete(T t)会立即执行; 但是complete(T t)只能调用一次，后续的重复调用会失效。

**CompletableFuture中的方法:**

i)runAsync 和 supplyAsync 方法的区别是runAsync返回的CompletableFuture是没有返回值的, 而supplyAsync返回的CompletableFuture是有返回值的;

ii). thenCompose可以用于组合多个CompletableFuture，将前一个结果作为下一个计算的参数，它们之间存在着先后顺序.

**7). 日期类的改变**

**Java 8 的新特性之一就是新的日期与时间库,旧库的缺点:**

i)旧库中的一些类，如 java.util.Date 与 SimpleDateFormatter，不是线程安全的，在多线程环境下需要开发者进行并发处理;

ii). 存在设计缺陷，如 java.util.Date 的年份从 1900 开始，月份从 0 开始，日期从 1 开始，这并不符合直觉感受.

java 8 中引入了新的 java.time 包，以解决这些问题。相比旧的 API，新的 **API 具有如下特性:**

i).不可变。新的 API 中基本日期、时间对象的值都是不可变的，所有修改类的操作都会返回一个新的对象实例，也正因为如此，新的日期类是线程安全的;

ii). 领域驱动设计。旧的 API 中，如 java.util.Date 实际是一个 UNIX 时间戳的包装，当调用 toString() 打印时又会显示时区，容易造成混淆。新的 API 设计避免了这种现象，每个类专注于自己的领域，所有的命名与概念都清晰易懂;

iii). 不同的历法，新的 API 提供了除了 ISO-8601 标准以外的历法，以满足世界上某些区域的特定需求.

**java.time 由五个包构成:**

java.time：包含了基本的时间、日期类;

java.time.chrono：提供了对不同历法的支持;

java.time.format：提供了对日期格式化与解析的支持;

java.time.temporal：提供了扩展特性;

java.time.zone：提供了时区的支持.

Java9新特性：

**Java 平台模块系统:** Java 9 模块的重要特征是在其工件（artifact）的根目录中包含了一个描述模块的 module-info.class 文 件。 工件的格式可以是传统的 JAR 文件或是 Java 9 新增的 JMOD 文件。这个文件由根目录中的源代码文件 module-info.java 编译而来。该模块声明文件可以描述模块的不同特征.

**JShell:** jshell 是 Java 9 新增的一个实用工具。jshell 为 Java 增加了类似 NodeJS 和 Python 中的读取-求值-打印循环（ Read-Evaluation-Print Loop ） 。 在 jshell 中 可以直接 输入表达式并查看其执行结果。当需要测试一个方法的运行效果，或是快速的对表达式进行求值时，jshell 都非常实用。只需要通过 jshell 命令启动 jshell，然后直接输入表达式即可。每个表达式的结果会被自动保存下来 ，以数字编号作为引用，类似 $1 和$2 这样的名称 。可以在后续的表达式中引用之前语句的运行结果。 在 jshell 中 ，除了表达式之外，还可以创建 Java 类和方法。jshell 也有基本的代码完成功能。

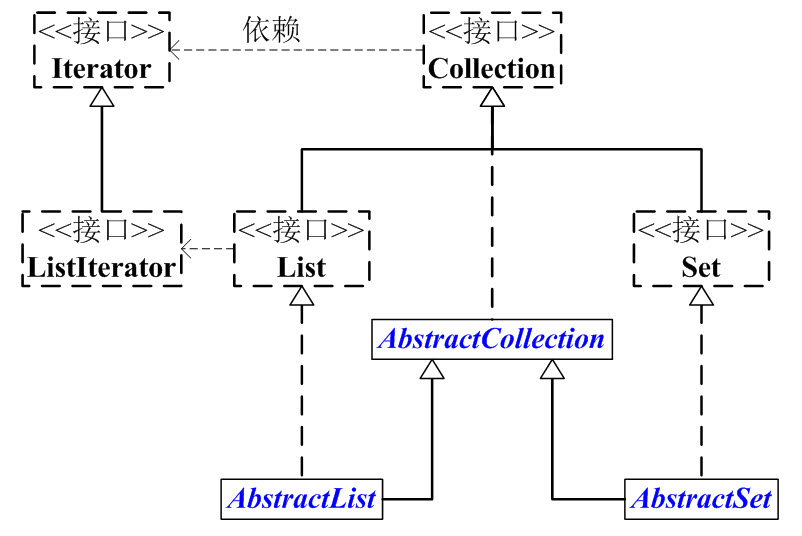
**集合,Stream,Optional:** 在集合上，Java 9 增加 了 List.of()、Set.of()、Map.of() 和 M ap.ofEntries()等工厂方法来创建不可变集合 ; Stream 中增加了新的方法 ofNullable、dropWhile、takeWhile 和 iterate; Optiona l 类中新增了 ifPresentOrElse、or 和 stream 等方法.

**进程的API:** Java 9 增加了 ProcessHandle 接口，可以对原生进程进行管理，尤其适合于管理长时间运行的进程。在使用 P rocessBuilder 来启动一个进程之后，可以通过 Process.toHandle()方法来得到一个 ProcessHandl e 对象的实例。通过 ProcessHandle 可以获取到由 ProcessHandle.Info 表示的进程的基本信息，如命令行参数、可执行文件路径和启动时间等。ProcessHandle 的 onExit()方法返回一个 CompletableFuture<ProcessHandle>对象，可以在进程结束时执行自定义的动作.

集合：

**1).Collection: ArrayList, LinkedList, Vector的区别及用法，Array和ArrayList的区别, 栈和队列,双端队列(Deque:ArrayDeque,LinkedList实现), Set: TreeSet, HashSet**

Collection是一个接口，它主要的两个分支是：**List** 和 **Set:** List和Set都是接口，它们继承于Collection。**List是有序的队列，List中可以有重复的元素**；而**Set是数学概念中的集合，Set中没有重复元素.**



**Collection:** public interface Collection<E> extends Iterable<E> {}**它是一个接口，是高度抽象出来的集合，它包含了集合的基本操作：添加、删除、清空、遍历(读取)、是否为空、获取大小、是否保护某元素等等.**

**AbstractCollection**: 是一个抽象类，它实现了Collection中除iterator()和size()之外的函数, 从而方便其它类实现Collection，比如ArrayList、LinkedList等，它们这些类想要实现Collection接口，通过继承AbstractCollection就已经实现了大部分的接口了.

**List:** List是继承于Collection接口，它自然就包含了Collection中的全部函数接口；由于List是有序队列，它也额外的有自己的API接口。主要有“添加、删除、获取、修改指定位置的元素”、“获取List中的子队列”等.

**AbstractList:**是一个继承于AbstractCollection，并且实现List接口的抽象类。它实现了List中除size()、get(int location)之外的函数, 它实现了List接口中的大部分函数。从而方便其它类继承List.

**ArrayList:** 是一个**数组队列**，相当于 **动态数组**。与Java中的数组相比，它的容量能动态增长。它继承于AbstractList，实现了List, RandomAccess, Cloneable, java.io.Serializable这些接口;

**ArrayList源码分析:**

i).ArrayList数据结构: elementData 是"Object[]类型的数组"，它保存了添加到ArrayList中的元素。实际上，elementData是个动态数组，我们能通过构造函数 ArrayList(int initialCapacity)来执行它的初始容量为initialCapacity；如果通过不含参数的构造函数ArrayList()来创建ArrayList，则elementData的容量默认是10。elementData数组的大小会根据ArrayList容量的增长而动态的增长,具体的增长方式为:

// 确定ArrarList的容量。

// 若ArrayList的容量不足以容纳当前的全部元素，设置 新的容量=“(原始容量x3)/2 + 1”

public void ensureCapacity(int minCapacity) {

// 将“修改统计数”+1

modCount++;

int oldCapacity = elementData.length;

// 若当前容量不足以容纳当前的元素个数，设置 新的容量=“(原始容量x3)/2 + 1”

if (minCapacity > oldCapacity) {

Object oldData[] = elementData;

int newCapacity = (oldCapacity \* 3)/2 + 1;

if (newCapacity < minCapacity)

newCapacity = minCapacity;

elementData = Arrays.copyOf(elementData, newCapacity);

}

}

ii). 当ArrayList容量不足以容纳全部元素时，ArrayList会重新设置容量：**新的容量=“(原始容量x3)/2 + 1”;**

**iii).** ArrayList的克隆函数，即是将全部元素克隆到一个数组中

iv).  ArrayList实现java.io.Serializable的方式。当写入到输出流时，先写入“容量”，再依次写入“每一个元素”；当读出输入流时，先读取“容量”，再依次读取“每一个元素”([ArrayList序列化的原理见集合类的序列化分析中](#_21).集合类与序列化)).

v).遍历方式: 通过Iterator变量; 通过随机访问遍历(下标随机访问遍历); for循环遍历. 其中使用随机访问(通过索引序号访问)效率最高,使用迭代期Iterator效率最低.

**LinkedList:** LinkedList 是一个继承于AbstractSequentialList的双向链表。它也可以被当作堆栈、队列或双端队列进行操作;  实现 List 接口，能对它进行队列操作;  实现 Deque 接口，即能将LinkedList当作双端队列使用; 实现了Cloneable接口，即覆盖了函数clone()，能克隆; 实现java.io.Serializable接口，这意味着LinkedList支持序列化，能通过序列化去传输;是非同步的.

**AbstractSequentialList:** AbstractSequentialList 实现了get(int index)、set(int index, E element)、add(int index, E element) 和 remove(int index)这些函数。**这些接口都是随机访问List的**，LinkedList是双向链表；既然它继承于AbstractSequentialList，就相当于已经实现了“get(int index)这些接口”; 若需要通过AbstractSequentialList自己实现一个列表，只需要扩展此类，并提供 listIterator() 和 size() 方法的实现即可。若要实现不可修改的列表，则需要实现列表迭代器的 hasNext、next、hasPrevious、previous 和 index 方法即可.

**LinkedList源码分析:**

i). LinkedList继承于AbstractSequentialList，并且实现了Dequeue接口, public class LinkedList<E>

extends AbstractSequentialList<E>

implements List<E>, Deque<E>, Cloneable, java.io.Serializable {}

LinkedList包含两个重要的成员：header 和 size, header是双向链表的表头，它是双向链表节点所对应的类Entry(java8中是Node,内部类,是双向链表对应的结点数据结构)的实例。Entry中包含成员变量： previous, next, element; 其中，previous是该节点的上一个节点，next是该节点的下一个节点，element是该节点所包含的值; size是双向链表中节点的个数.

ii). LinkedList实际上是通过双向链表去实现的。既然是双向链表，那么它的**顺序访问会非常高效，而随机访问效率比较低;** 实现了List接口{也就是说，它实现了get(int location)、remove(int location)等“根据***索引值***来获取、删除节点的函数”}。LinkedList是如何实现List的这些接口的，如何将“**双向链表和索引值联系起来的**”？ 是通过一个**计数索引值**来实现的。例如，当我们调用get(int index)时，首先会比较“index”和“双向链表长度的1/2”；若前者小，则从链表头开始往后查找，直到index位置；否则，从链表末尾开始先前查找，直到index位置,如下代码(JDK1.8)。 这也是“双向链表和索引值联系起来”的方法

public E get(int index) {  
        checkElementIndex(index);  
        return node(index).item;  
    }

Node<E> node(int index) {  
        // assert isElementIndex(index);  
        if (index < (size >> 1)) {  
            Node<E> x = first;  
            for (int i = 0; i < index; i++)  
                x = x.next;  
            return x;  
        } else {  
            Node<E> x = last;  
            for (int i = size - 1; i > index; i--)  
                x = x.prev;  
            return x;  
        }  
    }

iii)**.** 从LinkedList的实现方式中可以发现，它不存在LinkedList容量不足的问题; LinkedList的克隆函数，即是将全部元素克隆到一个新的LinkedList对象中; LinkedList实现java.io.Serializable。当写入到输出流时，先写入“容量”，再依次写入“每一个节点保护的值”；当读出输入流时，先读取“容量”，再依次读取“每一个元素”.

iv). 由于LinkedList实现了Deque，而Deque接口定义了在双端队列两端访问元素的方法。提供插入、移除和检查元素的方法。每种方法都存在两种形式：一种形式在操作失败时抛出异常，另一种形式返回一个特殊值（null 或 false，具体取决于操作）.

v). LinkedList可以作为**FIFO**(先进先出)的队列; LinkedList可以作为**LIFO**(后进先出)的栈.

vi).LinkedList的遍历方式: 第一种，通过**迭代器**遍历,即通过Iterator去遍历; 通过**快速随机**访问遍历LinkedList; 通过Foreach遍历; 通过**pollFirst()或pollLast()**来遍历LinkedList; 通过**removeFirst()或removeLast()**来遍历LinkedList; 遍历LinkedList时，使用removeFist()或removeLast()效率最高。但用它们遍历时，会删除原始数据；若单纯只读取，而不删除，应该使用Foreach遍历for (Integer integ:list); 尽量不用随机方式遍历.

**Vector:**  是**矢量队列**，它是JDK1.0版本添加的类。继承于AbstractList，实现了List, RandomAccess, Cloneable, Serializable这些接口;  继承了AbstractList，实现了List；所以，**它是一个队列，支持相关的添加、删除、修改、遍历等功能;** 实现了RandmoAccess接口，即**提供了随机访问功能**。RandmoAccess是java中用来被List实现，为List提供快速访问功能的; 实现了Cloneable接口，即实现clone()函数。它能被克隆; 和ArrayList不同，**Vector中的操作是线程安全的.**

**Vector源码分析:**

i)elementData 是"Object[]类型的数组"，它保存了添加到Vector中的元素。elementData是个动态数组，如果初始化Vector时，没指定动态数组的大小，则使用默认大小10。随着Vector中元素的增加，Vector的容量也会动态增长，capacityIncrement是与容量增长相关的增长系数;

ii). elementCount 是动态数组的实际大小, capacityIncrement 是动态数组的增长系数。如果在创建Vector时，指定了capacityIncrement的大小；则，每次当Vector中动态数组容量增加时, 增加的大小都是capacityIncrement, **若容量增加系数 >0，则将容量的值增加“容量增加系数”；否则，将容量大小增加一倍.**

// 确认“Vector容量”的帮助函数

private void ensureCapacityHelper(int minCapacity) {

int oldCapacity = elementData.length;

// 当Vector的容量不足以容纳当前的全部元素，增加容量大小。

// 若 容量增量系数>0(即capacityIncrement>0)，则将容量增大当capacityIncrement

// 否则，将容量增大一倍。

if (minCapacity > oldCapacity) {

Object[] oldData = elementData;

int newCapacity = (capacityIncrement > 0) ?

(oldCapacity + capacityIncrement) : (oldCapacity \* 2);

if (newCapacity < minCapacity) {

newCapacity = minCapacity;

}

elementData = Arrays.copyOf(elementData, newCapacity);

}

}

iii).vector的遍历方式: 通过**迭代器**遍历。即通过Iterator去遍历; **随机访问**，通过索引值去遍历;Foreach循环遍历; **Enumeration遍历;** 遍历Vector，使用索引的随机访问方式最快，使用迭代器最慢.

iv). **Vector是基于Synchronized实现的线程安全的ArrayList,但在插入元素时扩容的机制和ArrayList有区别,并可通过传入capacityIncrement来控制容量的扩充.**

**ArrayList, LinkedList, Vector的区别:**

i). ArrayList 本质上是一个可改变大小的数组.当元素加入时,其大小将会动态地增长(**(原始容量x3)/2 + 1**).内部的元素可以直接通过get与set方法进行访问.元素顺序存储 ,随机访问很快，删除非头尾元素慢，新增元素慢而且费资源 ,较适用于无频繁增删的情况 ,比数组效率低，如果不是需要可变数组，可考虑使用数组 ,非线程安全;

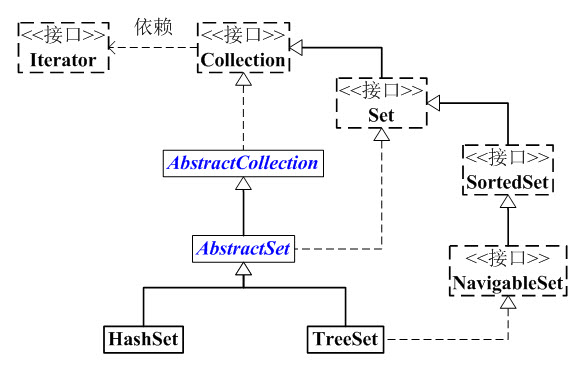
ii). LinkedList 是一个双链表,在添加和删除元素时具有比ArrayList更好的性能.但在get与set方面弱于ArrayList. 适用于 ：没有大规模的随机读取，有大量的增加/删除操作.随机访问很慢，增删操作很快，不耗费多余资源 ,允许null元素,非线程安全;

iii). Vector （类似于ArrayList）但其是同步的，开销就比ArrayList要大。如果你的程序本身是线程安全的，那么使用ArrayList是更好的选择。 Vector和ArrayList在更多元素添加进来时会请求更大的空间。Vector每次请求其大小的双倍空间(如果capacityIncrement>0,则在原来的基础上增加capacityIncrement)，而ArrayList每次对size增长**(原始容量x3)/2 + 1**.

iv).序列化的不同, ArrayList和LinkedList序列化都是自定义实现, 从而对null不进行序列化,而Vector没有限制.

Stack: **先进后出**(FILO, First In Last Out); Stack是继承于[Vector](http://www.cnblogs.com/skywang12345/p/3308833.html)(矢量队列)的，由于Vector是通过数组实现的，这就意味着，**Stack也是通过数组实现的**，**而非链表.** 执行**push**时(即，**将元素推入栈中**)，是通过将元素追加的数组的末尾中执行**peek**时(即，**取出栈顶元素，不执行删除**)，是返回数组末尾的元素, 执行**pop**时(即，**取出栈顶元素，并将该元素从栈中删除**)，是取出数组末尾的元素，然后将该元素从数组中删除,  Stack继承于Vector，意味着Vector拥有的属性和功能，Stack都拥有; Stack的pop,peek,search是线程安全的.

**Set:** Set是一个继承于Collection的接口，即Set也是集合中的一种。Set是没有重复元素的集合.



AbstractSet:是一个继承于AbstractCollection，并且实现Set接口的抽象类。由于Set接口和Collection接口中的API完全一样，Set也就没有自己单独的API。和AbstractCollection一样，它实现了List中除iterator()和size()之外的函数. HastSet 和 TreeSet 是Set的两个实现类; HashSet依赖于HashMap，它实际上是通过HashMap实现的。HashSet中的元素是无序的; TreeSet依赖于TreeMap，它实际上是通过TreeMap实现的。TreeSet中的元素是有序的.

**HashSet:** 是一个**没有重复元素的集合,** 由HashMap实现的，**不保证元素的顺序**，而且**HashSet允许使用 null 元素,** HashSet是**非同步的**。如果多个线程同时访问一个哈希 set，而其中至少一个线程修改了该 set，那么它必须保持外部同步。这通常是通过对自然封装该 set 的对象执行同步操作来完成的。如果不存在这样的对象，则应该使用 Collections.synchronizedSet 方法来“包装” set。最好在创建时完成这一操作，以防止对该 set 进行意外的不同步访问. HashSet通过iterator()返回的**迭代器是fail-fast的.**

**Set如何防重:因为HashSet基于HashMap实现, HashSet中的元素仅仅是HashMap的键key, 值都是相同的Object对象,而HashMap在插入时如果key相同,value不同则会更新value,value相同则不更新,且key始终保持不变,所以HashSet中插入相同元素时会调用HashMap中的put方法,此时返回null.TreeSet基于TreeMap实现,同样key相同是只是更新value而不会影响到key的值.**

**HashSet源码分析:**

i). HashSet继承于AbstractSet，并且实现了Set接口; HashSet的本质是一个"没有重复元素"的集合，它是通过[HashMap](http://www.cnblogs.com/skywang12345/p/3310835.html)实现的。HashSet中含有一个"HashMap类型的成员变量"map，HashSet的操作函数，实际上都是通过map实现的, 因为HashMap中的key不允许重复,所以HashSet中存的元素不能重复, HashSet只存了HashMap的key.

// HashSet是通过map(HashMap对象)保存内容的

private transient HashMap<E,Object> map;

// PRESENT是向map中插入key-value对应的value

// 因为HashSet中只需要用到key，而HashMap是key-value键值对；

// 所以，向map中添加键值对时，键值对的值固定是PRESENT

private static final Object PRESENT = new Object();

// 默认构造函数

public HashSet() {

// 调用HashMap的默认构造函数，创建map

map = new HashMap<E,Object>();

}

// 带集合的构造函数

public HashSet(Collection<? extends E> c) {

// 创建map。

// 为什么要调用Math.max((int) (c.size()/.75f) + 1, 16)，从 (c.size()/.75f) + 1 和 16 中选择一个比较大的树呢？

// 首先，说明(c.size()/.75f) + 1

// 因为从HashMap的效率(时间成本和空间成本)考虑，HashMap的加载因子是0.75。

// 当HashMap的“阈值”(阈值=HashMap总的大小\*加载因子) < “HashMap实际大小”时，

// 就需要将HashMap的容量翻倍。

// 所以，(c.size()/.75f) + 1 计算出来的正好是总的空间大小。

// 接下来，说明为什么是 16 。

// HashMap的总的大小，必须是2的指数倍。若创建HashMap时，指定的大小不是2的指数倍；

// HashMap的构造函数中也会重新计算，找出比“指定大小”大的最小的2的指数倍的数。

// 所以，这里指定为16是从性能考虑。避免重复计算。

map = new HashMap<E,Object>(Math.max((int) (c.size()/.75f) + 1, 16));

// 将集合(c)中的全部元素添加到HashSet中

addAll(c);

}

ii).HashSet实现了Cloneable接口, Serializable接口,但是序列化是自定义实现的, 其中字段HashMap是禁止序列化的, 实现自定义序列化函数的readObject, writeObject, 对HashSet的总容量, 加载因子, 实际容量, 所有元素进行序列化和反序列化.

iii).遍历方式: **通过Iterator遍历HashSet;** **根据toArray()获取HashSet的元素集合对应的数组.**

**iv).HashSet基于HashMap实现, 无容量限制; HashSet是非线程安全的**

**TreeSet:** TreeSet 是一个有序的集合，继承于AbstractSet,它的作用是提供有序的Set集合。它继承于AbstractSet抽象类，实现了NavigableSet<E>(提供了针对给定搜索目标返回最接近匹配项的导航方法), Cloneable, java.io.Serializable接口; 实现了NavigableSet接口，意味着它支持一系列的导航方法。比如查找与指定目标最匹配项; 实现了Cloneable接口，意味着它能被克隆; 实现了java.io.Serializable接口，意味着它支持序列化.

**TreeSet源码分析:**

i). TreeSet是基于TreeMap实现的(原因见问题3)。TreeSet中的元素支持2种排序方式：自然排序 或者 根据创建TreeSet 时提供的 Comparator 进行排序。这取决于使用的构造方法.

ii). TreeSet为基本操作（add、remove 和 contains）提供受保证的 log(n) 时间开销; TreeSet是非同步的。 它的iterator 方法返回的迭代器是fail-fast的.

// NavigableMap对象

private transient NavigableMap<E,Object> m;

// TreeSet是通过TreeMap实现的，

// PRESENT是键-值对中的值。

private static final Object PRESENT = new Object();

// 不带参数的构造函数。创建一个空的TreeMap

public TreeSet() {

this(new TreeMap<E,Object>());

}

// 将TreeMap赋值给 "NavigableMap对象m"

TreeSet(NavigableMap<E,Object> m) {

this.m = m;

}

// 带比较器的构造函数。

public TreeSet(Comparator<? super E> comparator) {

this(new TreeMap<E,Object>(comparator));

}

// 创建TreeSet，并将集合c中的全部元素都添加到TreeSet中

public TreeSet(Collection<? extends E> c) {

this();

// 将集合c中的元素全部添加到TreeSet中

addAll(c);

}

// 创建TreeSet，并将s中的全部元素都添加到TreeSet中

public TreeSet(SortedSet<E> s) {

this(s.comparator());

addAll(s);

}

// 返回TreeSet的顺序排列的迭代器。

// 因为TreeSet时TreeMap实现的，所以这里实际上时返回TreeMap的“键集”对应的迭代器

public Iterator<E> iterator() {

return m.navigableKeySet().iterator();

}

iii). TreeSet实际上是TreeMap实现的,采用NavigabeSet操作,但是调用的方法实现是TreeMap中的(因为NavigableSet是一个接口,而TreeMap实现了该接口), 只存储key的一个集合; 构造TreeSet时,若使用不带参数的构造函数，则TreeSet的使用自然比较器；若用户需要使用自定义的比较器，则需要使用带比较器的参数; TreeSet是非线程安全的; TreeSet实现java.io.Serializable的方式,当写入到输出流时，依次写入“比较器、容量、全部元素”；当读出输入流时，再依次读取(自定义学序列化实现方式)。

iv).TreeSet的迭代方式: **Iterator顺序遍历;转换成数组后用 for-each遍历HashSet.**

**2).HashMap,HashTable的区别，HashMap与TreeMap的区别**

**Map架构:**

i)Map 是**映射接口**，Map中存储的内容是**键值对**(key-value);

ii)AbstractMap 是**实现Map接口的抽象类，它实现了Map中的大部分API,** 其它Map的实现类可以通过继承AbstractMap来减少重复编码;

iii).SortedMap 是继承于Map接口。SortedMap中的内容是**排序的键值对**，排序的方法是通过比较器(Comparator);

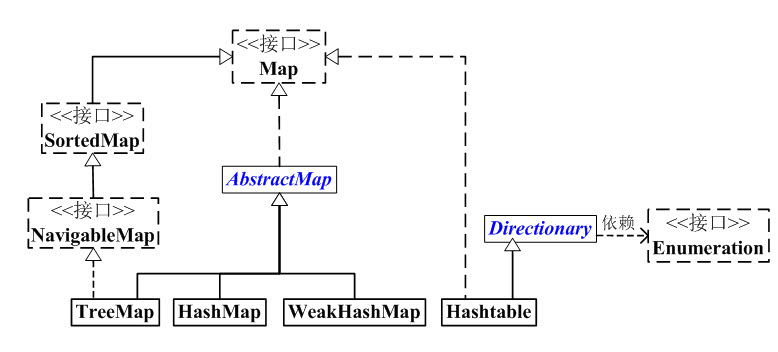
iv).NavigableMap 是继承于SortedMap的接口。相比于SortedMap，NavigableMap有一系列的导航方法；如"获取大于等于某对象的键值对"、“获取小于等于某对象的键值对”等等;

v).TreeMap 继承于AbstractMap，且实现了NavigableMap接口；因此，TreeMap中的内容是“**有序的键值对**”;

vi).HashMap 继承于AbstractMap，但没实现NavigableMap接口；因此，HashMap的内容是“**键值对，但不保证次序**”;

vii).Hashtable 虽然不是继承于AbstractMap，但它继承于Dictionary(Dictionary也是键值对的接口)，而且也实现Map接口；因此，Hashtable的内容也是“**键值对，也不保证次序**”。但和HashMap相比，Hashtable是线程安全的，而且它支持通过Enumeration去遍历;

viii).WeakHashMap 继承于AbstractMap。它和HashMap的键类型不同，**WeakHashMap的键是“弱键”.**



**Map的特点:**

i). Map 是一个键值对(key-value)映射接口。**Map映射中不能包含重复的键；每个键最多只能映射到一个值;**

**ii).** Map 接口提供三种collection 视图，允许以**键集**、**值集**或**键-值映**射关系集的形式查看某个映射的内容( entrySet()用于返回**键-值集**的**Set集合,** keySet()用于返回**键集**的**Set集合,** values()用户返回**值集**的**Collection集合,** 因为Map中不能包含重复的键；每个键最多只能映射到一个值。所以，**键-值集、键集都是Set，值集时Collection**);

iii). Map 映射顺序。有些实现类，可以明确保证其顺序，如 TreeMap；另一些映射实现则不保证顺序，如 HashMap 类;

iv). Map 的实现类应该提供2个“标准的”构造方法：**第一个，void（无参数）构造方法，用于创建空映射**；**第二个，带有单个 Map 类型参数的构造方法，用于创建一个与其参数具有相同键-值映射关系的新映射。**实际上，后一个构造方法允许用户复制任意映射，生成所需类的一个等价映射。尽管无法强制执行此建议（因为接口不能包含构造方法），但是 JDK 中所有通用的映射实现都遵从它;

v). Entry<K,V>是Map中内部的一个接口，Map.Entry是**键值对**，Map通过 entrySet() 获取Map.Entry的键值对集合，从而通过该集合实现对键值对的操作.

**AbstractMap:** public abstract class AbstractMap<K,V> implements Map<K,V> {};AbstractMap类提供 Map 接口的骨干实现，以最大限度地减少实现此接口所需的工作; 要实现不可修改的映射，编程人员只需扩展此类并提供 entrySet 方法的实现即可，该方法将返回映射的映射关系 set 视图。通常，返回的 set 将依次在 AbstractSet 上实现。此 set 不支持 add() 或 remove() 方法，其迭代器也不支持 remove() 方法; 要实现可修改的映射，编程人员必须另外重写此类的 put 方法（否则将抛出 UnsupportedOperationException），entrySet().iterator() 返回的迭代器也必须另外实现其 remove 方法.

**SortedMap:** 是一个继承于Map接口的接口。它是一个有序的SortedMap键值映射; 排序方式有两种：**自然排序** 或者 **用户指定比较器**。 插入有序 SortedMap 的所有元素都必须实现 Comparable 接口（或者被指定的比较器所接受）.

**NavigableMap**: public interface NavigableMap<K,V> extends SortedMap<K,V> { }

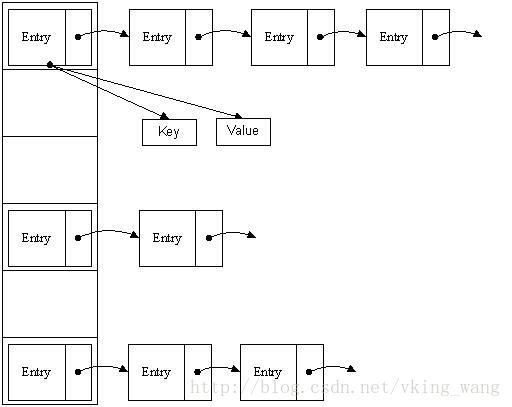
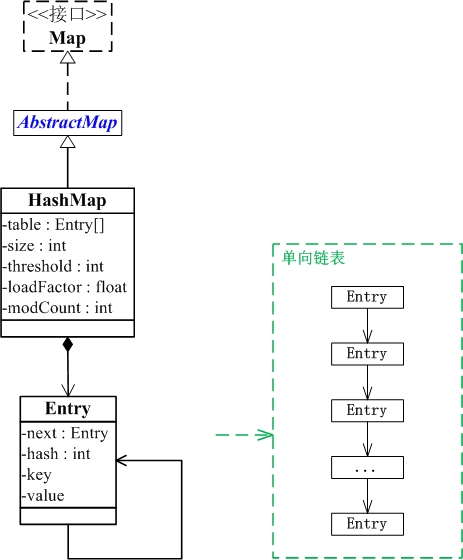
NavigableMap是继承于SortedMap的接口。它是一个可导航的键-值对集合，具有了为给定搜索目标报告最接近匹配项的导航方法; NavigableMap分别提供了获取“键”、“键-值对”、“键集”、“键-值对集”的相关方法.

i). **提供操作键-值对的方法:**  lowerEntry、floorEntry、ceilingEntry 和 higherEntry 方法，它们分别返回与小于、小于等于、大于等于、大于给定键的键关联的 Map.Entry 对象; firstEntry、pollFirstEntry、lastEntry 和 pollLastEntry 方法，它们返回和/或移除最小和最大的映射关系（如果存在），否则返回 null;

ii). **提供操作键的方法:**  lowerKey、floorKey、ceilingKey 和 higherKey 方法，它们分别返回与小于、小于等于、大于等于、大于给定键的键;  
iii). **获取键集:** navigableKeySet、descendingKeySet分别获取正序/反序的键集.

**HashMap:** 继承于AbstractMap，实现了Map、Cloneable、java.io.Serializable接口; HashMap 的实现不是同步的，这意味着它不是线程安全的。它的key、value都可以为null。此外，HashMap中的映射不是有序的; HashMap 的实例有两个参数影响其性能：“**初始容量**” 和 “**加载因子**”。容量 是哈希表中桶的数量，初始容量 只是哈希表在创建时的容量。加载因子 是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 rehash 操作（即重建内部数据结构），从而哈希表将具有大约两倍的桶数; 通常，**默认加载因子是 0.75**, 这是在时间和空间成本上寻求一种折衷。加载因子过高虽然减少了空间开销，但同时也增加了查询成本（在大多数 HashMap 类的操作中，包括 get 和 put 操作，都反映了这一点）。在设置初始容量时应该考虑到映射中所需的条目数及其加载因子，以便最大限度地减少 rehash 操作次数。如果初始容量大于最大条目数除以加载因子，则不会发生 rehash 操作.

**JDK1.7之前HashMap源码分析:**



**size:** HashMap中存放KV的数量;

**capacity:** 容量, HashMap中桶的数量,默认值为16,容量都是2的幂;

**loadFactor:** 装载因子,用来衡量HashMap满的程度,默认值为0.75, 实时装载因子=size/capacity;

**threshold:** 表示当HashMap的size大于threshold时会执行resize操作, threshold = capacity\*loadFactor.

i). HashMap继承于AbstractMap类，实现了Map接口。Map是"key-value键值对"接口，AbstractMap实现了"键值对"的通用函数接口;

ii). table是一个Entry[]数组类型(JDK1.8中是Node)，而Entry实际上就是一个单向链表。哈希表的"key-value键值对"都是存储在Entry数组中, size是HashMap的大小，它是HashMap保存的键值对的数量, threshold是HashMap的阈值，用于判断是否需要调整HashMap的容量。threshold的值="容量\*加载因子"，当HashMap中存储数据的数量达到threshold时，就需要将HashMap的容量加倍; loadFactor就是加载因子; modCount是用来实现fail-fast机制的.

**属性:**

// 默认的初始容量是16，必须是2的幂。

static final int DEFAULT\_INITIAL\_CAPACITY = 16;

// 最大容量（必须是2的幂且小于2的30次方，传入容量过大将被这个值替换）

static final int MAXIMUM\_CAPACITY = 1 << 30;

// 默认加载因子

static final float DEFAULT\_LOAD\_FACTOR = 0.75f;

// 存储数据的Entry数组，长度是2的幂。

// HashMap是采用拉链法实现的，每一个Entry本质上是一个单向链表

transient Entry[] table;

// HashMap的大小，它是HashMap保存的键值对的数量

transient int size;

// HashMap的阈值，用于判断是否需要调整HashMap的容量（threshold = 容量\*加载因子）

int threshold;

// 加载因子实际大小

final float loadFactor;

// HashMap被改变的次数

transient volatile int modCount;

**链式存储对应的链表结构:**

// Entry是单向链表。

// 它是 “HashMap链式存储法”对应的链表。

// 它实现了Map.Entry 接口，即实现getKey(), getValue(), setValue(V value), //equals(Object o), hashCode()这些函数

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

// 指向下一个节点

Entry<K,V> next;

final int hash;

// 构造函数。

// 输入参数包括"哈希值(h)", "键(k)", "值(v)", "下一节点(n)"

Entry(int h, K k, V v, Entry<K,V> n) {

value = v;

next = n;

key = k;

hash = h;

}

….Entry接口中的方法实现….

i)哈希表是由**数组+链表**组成的，一个长度为16的数组中，每个元素存储的是一个链表的头结点。那么这些元素是按照什么样的规则存储到数组中呢。一般情况是通过hash(key)%len获得，也就是元素的key的哈希值对数组长度取模得到.

ii). HashMap里面实现一个静态内部类Entry，而Entry 实现了Map.Entry 接口，即实现getKey(), getValue(), setValue(V value), equals(Object o), hashCode()这些函数。这些都是基本的读取/修改key、value值的函数. Entry重要的属性有 key , value, next，从属性key,value我们就能很明显的看出来Entry就是HashMap键值对实现的一个基础bean, HashMap的基础就是一个线性数组，这个数组就是Entry[]，Map里面的内容都保存在Entry[]里面.

**HashMap的构造函数:**

public HashMap():默认构造函数,初始化大小capacity为16,加载因子loadFactor为0.75;

public HashMap(int initialCapacity, float loadFactor) :指定容量和加载因子,HashMap的最大容量是1<<30.

public HashMap(int initialCapacity):指定容量

public HashMap(Map<? extends K, ? extends V> m):包含子Map的构造函数

**HashMap的put函数:** Entry(Node)类里面有一个next属性，作用是指向下一个Entry。打个比方， 第一个键值对A进来，通过计算其key的hash得到的index=0(indexFor)，记做:Entry[0] = A。一会后又进来一个键值对B，通过计算其index也等于0，现在怎么办？HashMap会这样做:*B.next = A*,Entry[0] = B,如果又进来C,index也等于0,那么*C.next = B*,Entry[0] = C；这样我们发现index=0的地方其实存取了A,B,C三个键值对,他们通过next这个属性链接在一起。所以疑问不用担心。**也就是说数组中存储的是最后插入的元素。**

**public** V put(K key, V value) {

**if** (key == **null**)

**return** putForNullKey(value); //null总是放在数组的第一个链表中

**int** hash = *hash*(key.hashCode());

**int** i = *indexFor*(hash, table.length);

        //遍历链表

**for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {

            Object k;

            //如果key在链表中已存在，则替换为新value

**if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

                V oldValue = e.value;

                e.value = value;

                e.recordAccess(**this**);

**return** oldValue;

            }

        }

        modCount++;

        addEntry(hash, key, value, i);

**return** **null**;

}

**void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {

    Entry<K,V> e = table[bucketIndex];

    table[bucketIndex] = **new** Entry<K,V>(hash, key, value, e); //参数e, 是Entry.next

    //如果size超过threshold，则扩充table大小。再散列

**if** (size++ >= threshold)

            resize(2 \* table.length);

}

在调用put方法时,会先调用key的hashCode()方法,返回key的hash值,计算Index后用户找到哈希桶(bucket)的位置来存储Entry对象, 如果两个对象的key的hash值相同,那么对于的bucket位置也相同,但equals()不同,添加元素时会发生hash冲突,HashMap采用链表来解决冲突. 在put时,先判断数组中是否存在完全相同的key对象,如果这个key对象在table数组中已经存在,就用新的value替代老的value,如果不存在,就创建一个新的Entry对象添加到table[i]处. 如果table[i]已经存在其他元素,那么新Entry对象将会存储在bucket链表的表头,通过next指向原有的Entry对象,形成链表结构.

**HashMap的get函数:**

**public** V get(Object key) {

**if** (key == **null**)

**return** getForNullKey();

**int** hash = *hash*(key.hashCode());

        //先定位到数组元素，再遍历该元素处的链表

**for** (Entry<K,V> e = table[*indexFor*(hash, table.length)];

             e != **null**;

             e = e.next) {

            Object k;

**if** (e.hash == hash && ((k = e.key) == key || key.equals(k)))

**return** e.value;

        }

**return** **null**;

}

如果两个不同的key的hashCode相同,那么这两个值存储在同一个bucket中,要获取value,首先根据key的hashCode找到bucket的位置,因为HashMap在链表中存储的是Entry(JDK1.8为Node)键值对,所以找到bucket位置之后,会调用key的equals()方法,按顺序变量链表的每个Entry,知道找到Entry为止.

**HashMap的indexFor方法:**确定数组的index(属于哪一个桶), 在进行HashMap的get和put时需要计算key属于哪个table, 当确定属于哪个table时, 然后在通过key去遍历当前table为头结点的链表,直到找到key对应的值.

**HashMap的遍历方式:**

i). **遍历HashMap的键值对**

// 假设map是HashMap对象

// map中的key是String类型，value是Integer类型

Integer integ = null;

Iterator iter = map.entrySet().iterator();

while(iter.hasNext()) {

Map.Entry entry = (Map.Entry)iter.next();

// 获取key

key = (String)entry.getKey();

// 获取value

integ = (Integer)entry.getValue();

}

ii). **遍历HashMap的键**

// 假设map是HashMap对象

// map中的key是String类型，value是Integer类型

String key = null;

Integer integ = null;

Iterator iter = map.keySet().iterator();

while (iter.hasNext()) {

// 获取key

key = (String)iter.next();

// 根据key，获取value

integ = (Integer)map.get(key);

}

iii). **遍历HashMap的值**

// 假设map是HashMap对象

// map中的key是String类型，value是Integer类型

Integer value = null;

Collection c = map.values();

Iterator iter= c.iterator();

while (iter.hasNext()) {

value = (Integer)iter.next();

}

**HashMap(1.7)中的hash原理:**

第一步是通过key.hashCode()得到hashCode值,第二步高位参与与运算key.hashCode^(key.hashCode>>>16),第三步取模h & (length-1), length是bucket的个数,即table的长度; 最终通过indexFor返回得到该key对应的table的位置,即属于哪个桶.

**HashMap(1.7)的桶的大小为什么是2的n次方:**

// 找出“大于initialCapacity”的最小的2的幂

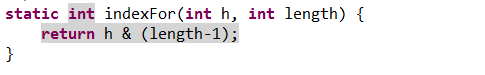
int capacity = 1;

while (capacity < initialCapacity)

capacity <<= 1;

在HashMap的构造函数中,如果不指定大小,则容量为默认16,如果自定义大小,那么会把容量指定为大于自定义大小的最小的2次幂, 例如自定义initialCapacity为5, 那么执行上述while循环后, capacity为8, 即大于5的最小2的幂为8.

桶的大小(容量)为2的n次幂的原因在于用(数组长度-1)作为一个”低位掩码”, 进行与操作的结果就是散列值的高位全部归零,只保留低位值用来做数组的下标访问. 例如: 容量为16, (16-1)二进制为:00000000 00000000 00001111, 与key的hashCode后再进行扰动(调用hash)的二进制数:10100101 11000100 00100101进行与运算得到结果为: 00000000 00000000 00000101 ; 高位全部归零,只保留低位, 这样能够很好的避免冲突;



但是如果这个操作直接让(16-1)与key.hashCode()做与运算,如果只取最后低几位的散列,冲突仍然会严重,为了解决这个问题, 在key.hashCode()的基础上进行散列扰动,即hash(key.hashCode()), 扰动处理加大哈希码低位的随机性,使得分布更均匀,从而提高对应数组存储下标位置的随机性和均匀性, 最终减少hash冲突. 其中hash(K key)在JDK1.7中的定义如下:

static int hash(int h) {

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

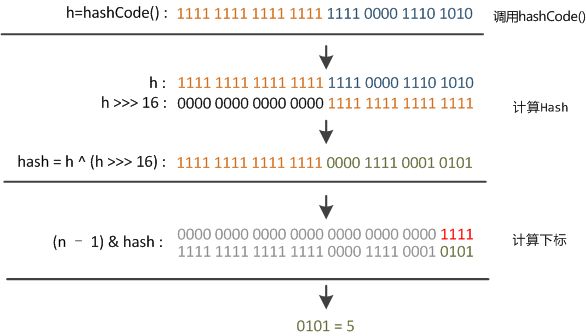
在JDK1.8中定义如下:

static int hash(Object key) {

int h;

return (key == null) ? 0 : (h=key.hashCode())^(h>>>16));

}



hashCode右移16位,正好是32位的一半,自己的高半区和低半去做异或运算, 可以混合原始哈希码的高位和低位,以此来加大低位的随机性,而且混合后的低位掺杂了高位的部分特征,这样高位信息也能变相被保存下来.

Java8中的扰动和Java7中的区别, Java8认为扰动做一次就可以,多了的话边际效用不大, 所以在Java8中可以看到扰动只做了一次异或运算.

**HashMap(1.7)的扩容机制:**

**resize函数:**void resize(int newCapacity) {   //传入新的容量

    Entry[] oldTable = table;    //引用扩容前的Entry数组

    int oldCapacity = oldTable.length;

    if (oldCapacity == MAXIMUM\_CAPACITY) {  //扩容前的数组大小如果已经达到最大(2^30)了

        threshold = Integer.MAX\_VALUE; //修改阈值为int的最大值(2^31-1)，这样以后就不会扩容了

        return;

    }

    Entry[] newTable = new Entry[newCapacity];  //初始化一个新的Entry数组

    transfer(newTable);                         //！！将数据转移到新的Entry数组里

    table = newTable;                           //HashMap的table属性引用新的Entry数组

    threshold = (int) (newCapacity \* loadFactor);//修改阈值

}

**transfer函数:** void transfer(Entry[] newTable) {

    Entry[] src = table;                   //src引用了旧的Entry数组

    int newCapacity = newTable.length;

    for (int j = 0; j < src.length; j++) { //遍历旧的Entry数组

        Entry<K, V> e = src[j];             //取得旧Entry数组的每个元素

        if (e != null) {

            src[j] = null;//释放旧Entry数组的对象引用（for循环后，旧的Entry数组不再引用任何对象）

            do {

                Entry<K, V> next = e.next;

                int i = indexFor(e.hash, newCapacity); //！！重新计算每个元素在数组中的位置

                e.next = newTable[i]; //标记[1]

                newTable[i] = e;      //将元素放在数组上

                e = next;             //访问下一个Entry链上的元素

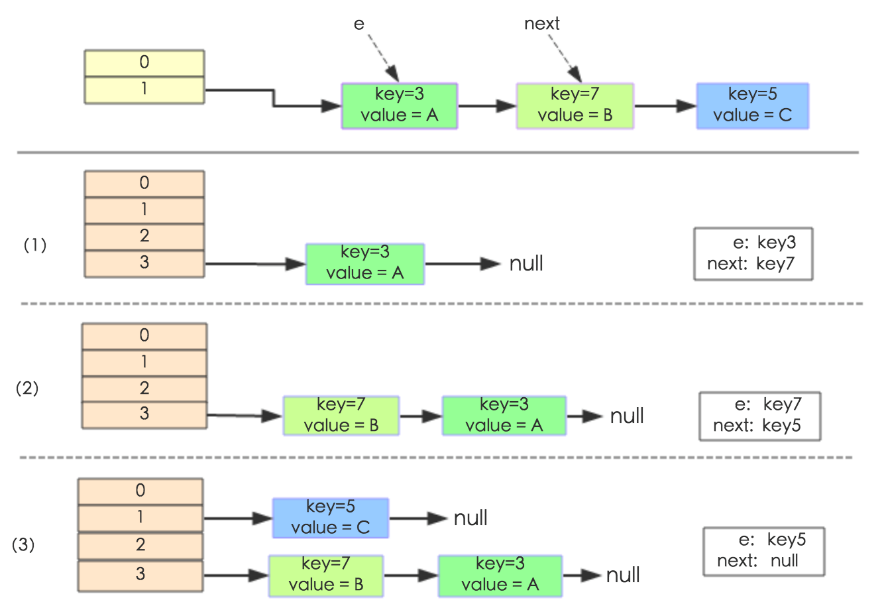
            } while (e != null);

        }

    }

}

**例如:** 一个HashMap的size为2, key=3,7,5; 则进行put5,3,7时, 假设解决冲突的方式为mod2, 那么就会如下图所示: 5,3,7均会映射到桶1中,然后按照后插入的元素插入到头结点(头插法)一次插入, 此时超过size=2, 所以需要进行扩容.



**扩容的经过:** 在插入put操作时,会调用addEntry,判断如果size>=threshold(其中threshold=capacity\*loadFactor, size为所存储的时间元素个数)时,会进行扩容,创建一个新的table数组,传入新的数组大小为2\*table.length,扩大到原来的2倍, 并更新threshold, 所以此时容量变为4, 然后在扩容的时候对所有的Node进行rehash,重新分布,如上图.

**JDK1.7之前与JDK1.8中HashMap的区别:** JDK1.7主要基于数组+链表实现, JDK1.8基于数组+链表+红黑树

i). hash(key)的区别(即 计算存放在数组 table 中的位置（即数组下标、索引）的过程的区别): 计算传入数据的哈希码（哈希值、Hash值）; 该函数在JDK 1.7 和 1.8 中的实现不同，但原理一样 = 扰动函数 = 使得根据key生成的哈希码（hash值）分布更加均匀、更具备随机性，避免出现hash值冲突（即指不同key但生成同1个hash值）, 具体扰动处理见上面分析过程.

JDK 1.7 做了9次扰动处理 = 4次位运算 + 5次异或运算:

// JDK 1.7实现：将 键key 转换成 哈希码（hash值）操作 = 使用hashCode() + 4次位运算 + 5次异或运算（9次扰动）

static final int hash(int h) {

h ^= k.hashCode();

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

JDK 1.8 简化了扰动函数 = 只做了2次扰动 = 1次位运算 + 1次异或运算  
// JDK 1.8实现：将 键key 转换成 哈希码（hash值）操作 = 使用hashCode() + 1次位运算 + 1次异或运算（2次扰动）

// 1. 取hashCode值： h = key.hashCode()

// 2. 高位参与低位的运算：h ^ (h >>> 16)

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

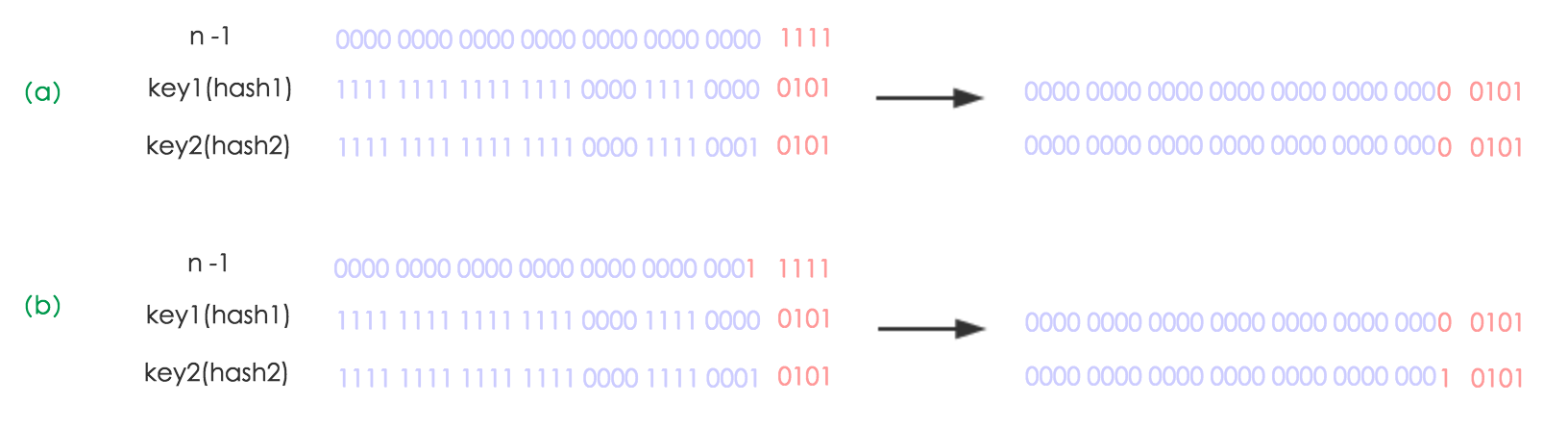
// a. 当key = null时，hash值 = 0，所以HashMap的key 可为null

// 注：对比HashTable，HashTable对key直接hashCode（），若key为null时，会抛出异常，所以HashTable的key不可为null

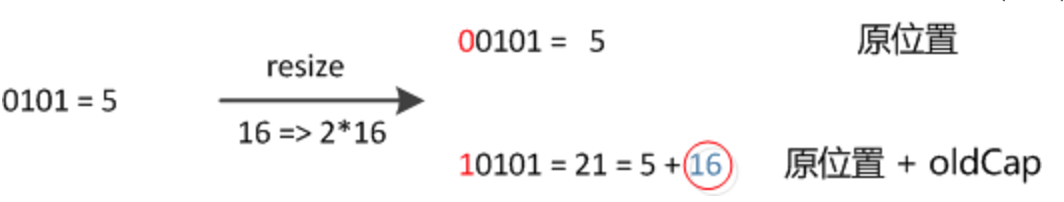
// b. 当key ≠ null时，则通过先计算出 key的 hashCode()（记为h），然后 对哈希码进行 扰动处理： 按位 异或（^） 哈希码自身右移16位后的二进制

}

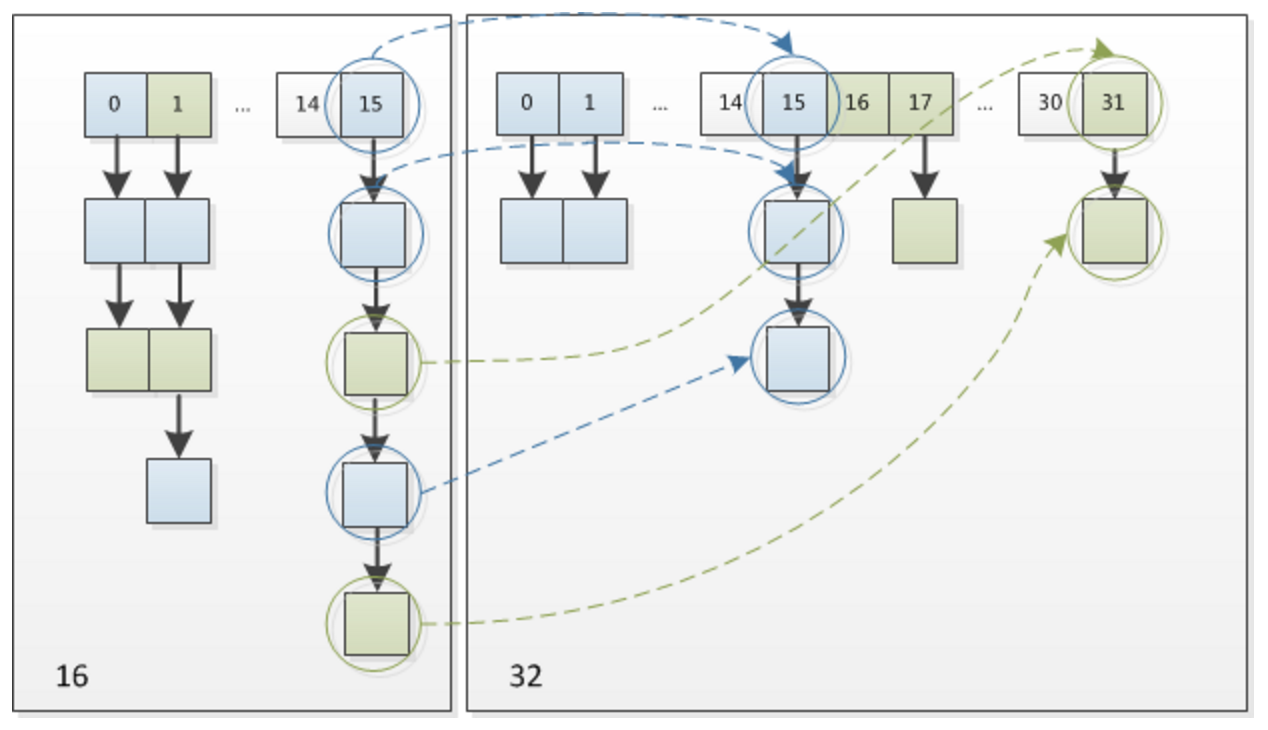
ii).put的区别: JDK1.8由于数据结构中加入了红黑树，所以在存放数据到哈希表中时，需进行多次数据结构的判断：**数组、红黑树、链表.** 在JDK1.7中插入时,直接在链表中插入; 在JDK1.8中出了用来判断是否冲突与JDK1.7一致外, 还需要判断插入的是否为红黑树节点, 如果为红黑树, 那么调用红黑树的插入; 如果不是红黑树,则直接调用链表的方法插入,当链表的长度大于8时,则把链表转换为红黑树; 同时如果是链表插入,如果有新的冲突进来需要插入当前数元素的对应链表中, JDK1.7是采用头插法, JDK1.8采用尾插法.  
iii).扩容机制:JDK1.7的扩容机制见上面的分析, 在1.8中做了优化,因为每次扩充时都是按照2的次幂扩充,所以扩充后元素的位置要么在原位置, 要么在原位置再移动2次幂的位置,如下图:图(a)表示扩容前的key1和key2确定索引(哪一个bucket)的位置, 图(b)表示扩容后的key1和key2确定索引的位置.



元素在重新计算hash后,因为n变为原来的2倍,那么(n-1)的”掩码”范围就会在高位多一位1, 因此新的index就会发生如下变化:



所以在扩容HashMap的时候不需要与1.7一样完全重新计算hash, 只需要判断原来的hash值新增的那个bit位是1还是0, 如果是0索引位置就没变,如果是1那么新的索引位置就会变为(原索引+oldCap(原容量)), 下图为由16扩容为32的过程:



这个设计确实非常的巧妙，既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的bucket了。这一块就是JDK1.8新增的优化点。有一点注意区别，JDK1.7中rehash的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置.

iv). 线程安全性: HashMap是非线程安全的.

v).性能对比: 因为1.8中在链表的长度大于8的时候会转换为红黑树,所以会将时间复杂度从O(n)降低到O(lgn), 随着size 的增大,1.7花费的时间是增长的趋势, 而1.8是降低趋势.

HashMap的序列化: 自定义实现, 重写writeObject, readObject方法,对HashMap的buckets,size,进行序列化.在对Node节点进行序列化时, 需要去除null的Node然后将Node的key,value分别序列化.

**HashMap为什么是非线程安全的:**

i).向HashMap中插入数据时调用addEntry方法, 假如A线程和B线程同时进入addEntry，然后计算出了相同的哈希值对应了相同的数组位置，因为此时该位置还没数据，然后对同一个数组位置调用createEntry，两个线程会同时得到现在的头结点，然后A写入新的头结点之后，B也写入新的头结点，那B的写入操作就会覆盖A的写入操作造成A的写入操作丢失。

ii).HashMap扩容的时候: 当多个线程同时进来，检测到总数量超过门限值的时候就会同时调用resize操作，各自生成新的数组并rehash后赋给该map底层的数组table，结果最终只有最后一个线程生成的新数组被赋给table变量，其他线程的均会丢失。而且当某些线程已经完成赋值而其他线程刚开始的时候，就会用已经被赋值的table作为原始数组，这样也会有问题。所以在扩容操作的时候也有可能会引起一些并发的问题.

iii).删除数据的时候: 可能会出现两种线程安全问题，第一种是一个线程判断得到了指定的数组位置i并进入了循环，此时，另一个线程也在同样的位置已经删掉了i位置的那个数据了，然后第一个线程那边就没了。 另一种情况，当多个线程同时操作同一个数组位置的时候，也都会先取得现在状态下该位置存储的头结点，然后各自去进行计算操作，之后再把结果写会到该数组位置去，其实写回的时候可能其他的线程已经就把这个位置给修改过了，就会覆盖其他线程的修改。

iv).多线程中HashMap如何造成死循环(链表出现环):JDK1.7中

void transfer(Entry[] newTable) {

Entry[] src = table;

int newCapacity = newTable.length;

for (int j = 0; j < src.length; j++) {

Entry<K,V> e = src[j];

if (e != null) {

src[j] = null;

do {

Entry<K,V> next = e.next;

int i = indexFor(e.hash, newCapacity);

e.next = newTable[i];

newTable[i] = e;

e = next;

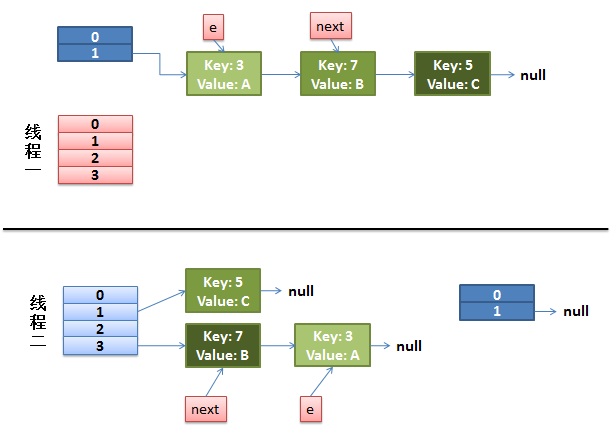
} while (e != null);

}

}

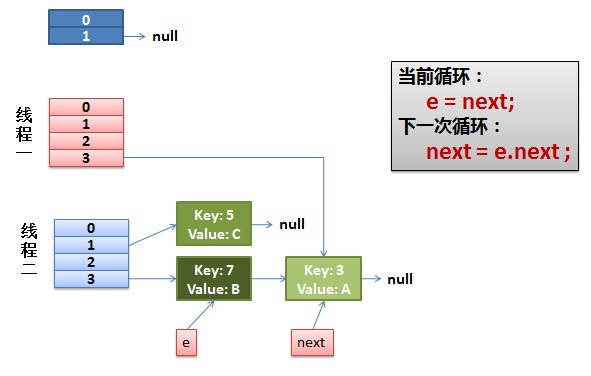
}

初始如下图的第一行所示,假设此时线程2执行了rehash操作:

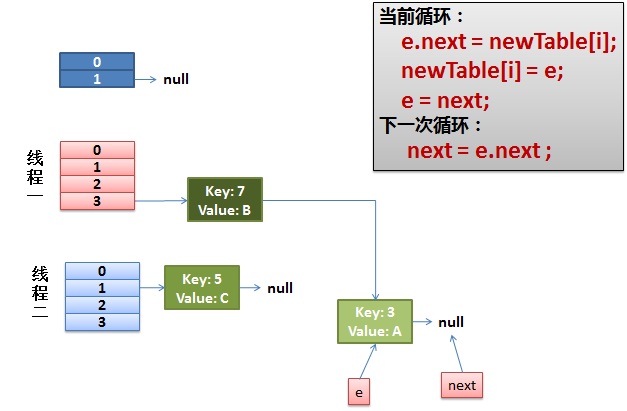


**因为Thread1的 e 指向了key(3)，而next指向了key(7)(初始状态相同,Thread2做了rehash操作)，其在线程二rehash后，指向了线程二重组后的链表;**

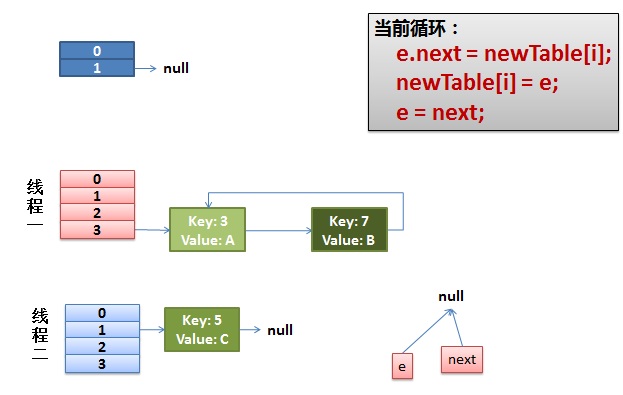
然后接着执行线程1, **先是执行 newTalbe[i] = e; 然后是e = next，导致了e指向了key(7); 而下一次循环的next = e.next导致了next指向了key(3);**



线程一接着工作。**把key(7)摘下来，放到newTable[i]的第一个，然后把e和next往下移;**



环状的出现: **e.next = newTable[i] 导致  key(3).next 指向了 key(7), 此时的key(7).next 已经指向了key(3)， 环形链表就这样出现了.**



**HashTable:**

i). Hashtable继承于Dictionary类，实现了Map接口Cloneable和java.io.Serializable。Map是"key-value键值对"接口，Dictionary是声明了操作"键值对"函数接口的抽象类;

ii). Hashtable是通过"拉链法"实现的哈希表。它包括几个重要的成员变量：table, count, threshold, loadFactor, modCount, table是一个Entry[]数组类型，而Entry实际上就是一个单向链表。哈希表的"key-value键值对"都是存储在Entry数组中的, count是Hashtable的大小，它是Hashtable保存的键值对的数量, threshold是Hashtable的阈值，用于判断是否需要调整Hashtable的容量。threshold的值="容量\*加载因子", loadFactor就是加载因子, modCount是用来实现fail-fast机制的.

**HashTable源码分析:**

i).HashTable的源码与HashMap1.7相似,初始大小为11, 加载因子为0.75,只是多数方法加了synchronized同步.

ii).put方法逻辑与HashMap1.7类似:先获取synchronized锁,put方法不允许null(不同于HashMap),然后计算key的hash值和index(计算index时直接用key的hashCode与length进行mod运算),然后遍历对应的链表,如果发现有相同的值,则更新,否则插入, 在插入时需要扩容, newCapacity = oldCapacity<<1 + 1(原来的容量的2倍加1),扩容后用新数组承接旧数组.

iii).get方法与HashMap的get类似,在计算key的hash值时有区别, int hash = key.hashCode();int index = (hash & 0x7FFFFFFF) % tab.length;

iv).自定义序列化:同步方法,把容量table.length, 总数count不为空的键值对序列化.  
**HashMap,HashTable,TreeMap的区别(主要针对1.7中的区别, 1.8中HashMap已经大改):**

**i).计算hash的方式不同:** HashMap中计算key的hash方式如下:

4次位运算 + 5次异或运算（9次扰动）

static final int hash(int h) {

h ^= k.hashCode();

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

JDK 1.8 简化了扰动函数 = 只做了2次扰动 = 1次位运算 + 1次异或运算  
// JDK 1.8实现：将 键key 转换成 哈希码（hash值）操作 = 使用hashCode() + 1次位运算 + 1次异或运算（2次扰动）

// 1. 取hashCode值： h = key.hashCode()

// 2. 高位参与低位的运算：h ^ (h >>> 16)

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

// a. 当key = null时，hash值 = 0，所以HashMap的key 可为null

// 注：对比HashTable，HashTable对key直接hashCode（），若key为null时，会抛出异常，所以HashTable的key不可为null

// b. 当key ≠ null时，则通过先计算出 key的 hashCode()（记为h），然后 对哈希码进行 扰动处理： 按位 异或（^） 哈希码自身右移16位后的二进制

}

而HashTable中的hash方式如下:

int hash = key.hashCode();int index = (hash & 0x7FFFFFFF) % tab.length;

在计算hash时直接采用key 的hashCode的方式;

**ii).初始化的不同:**HashMap的初始容量为16, 加载因子为0.75, HashTable的初始容量为11,加载因子为0.75;

**iii).扩容机制的不同:**HashMap的扩容是newCap=2\*oldCap;而HashTable是newCap=oldCap\*2+1的方式;

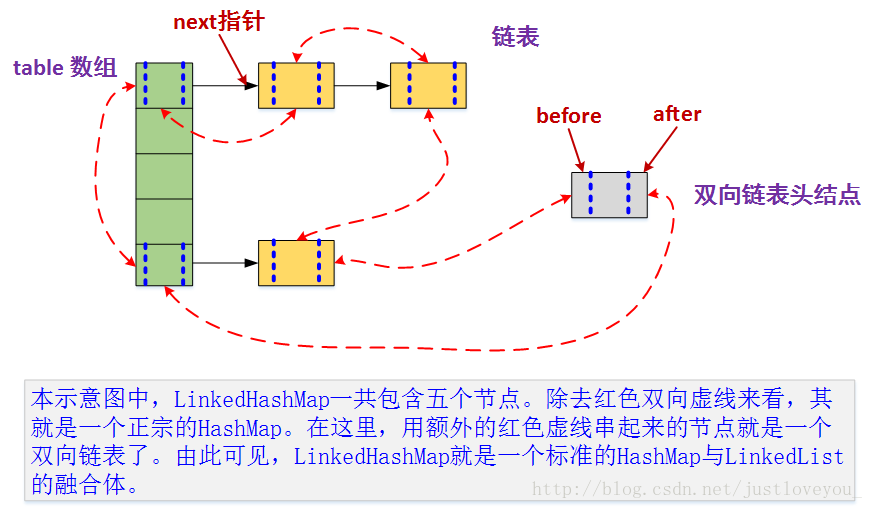
iv).key为null的不同:HashMap可以使用null作为key, 但不建议这么做, HashTable不允许key为null;

v).线程安全性的区别:HashTable通过使用synchronized保证线程安全, 而HashMap无法保证线程安全, 因此相对而言HashMap性能会高一些，我们平时使用时若无特殊需求建议使用HashMap，在多线程环境下若使用HashMap需要使用Collections.synchronizedMap()方法来获取一个线程安全的集合（Collections.synchronizedMap()实现原理是Collections定义了一个SynchronizedMap的内部类，这个类实现了Map接口，在调用方法时使用synchronized来保证线程同步,当然了实际上操作的还是我们传入的HashMap实例，简单的说就是Collections.synchronizedMap()方法帮我们在操作HashMap时自动添加了synchronized来实现线程同步，类似的其它Collections.synchronizedXX方法也是类似原理）.

vi).TreeMap是基于红黑树实现,有序的(根据key排序)Map,默认是升序,可以自定义排序规则.

**LinkedHashMap:** LinkedHashMap增加了时间和空间上的开销，但是它通过维护一个额外的双向链表保证了迭代顺序。特别地，该迭代顺序可以是插入顺序，也可以是访问顺序。因此，根据链表中元素的顺序可以将LinkedHashMap分为：**保持插入顺序的LinkedHashMap** 和 **保持访问顺序LinkedHashMap**，其中LinkedHashMap的默认实现是按插入顺序排序的.

**LinkedHashMap源码分析:**



i). HashMap和双向链表合二为一即是LinkedHashMap, 所谓LinkedHashMap，主要部分与HashMap相同，因此更准确地说，它是一个将所有Entry节点链入一个双向链表双向链表的HashMap, 在LinkedHashMapMap中，所有put进来的Entry都保存在哈希表中(与HashMap相同)，但由于它又额外定义了一个以head为头结点的双向链表(如上图所示)，因此对于每次put进来Entry，除了将其保存到哈希表中对应的位置上之外，还会将其插入到双向链表的尾部.

ii).上图中, HashMap和双向链表的密切配合和分工合作造就了LinkedHashMap, 需要注意的是，next用于维护HashMap各个桶中的Entry链，before、after用于维护LinkedHashMap的双向链表，虽然它们的作用对象都是Entry，但是各自分离，是两码事儿.

iii). public class LinkedHashMap<K,V>

extends HashMap<K,V>

implements Map<K,V> {

}

与HashMap相比,增加了两个成员变量用于保存迭代顺序, 分别是**双向链表头结点**header 和 **标志位accessOrder** (值为true时，表示按照访问顺序迭代；值为false时，表示按照插入顺序迭代).

iv).定义双向链表的Entry: LinkedHashMap采用的hash算法和HashMap相同，但是它重新定义了Entry。LinkedHashMap中的Entry增加了两个指针 **before** 和 **after**，它们分别用于维护双向链接列表。特别需要注意的是，**next用于维护HashMap各个桶中Entry的连接顺序，before、after用于维护Entry插入的先后顺序的，**源代码如下：

private static class Entry<K,V> extends HashMap.Entry<K,V> {

// These fields comprise the doubly linked list used for iteration.

Entry<K,V> before, after;

Entry(int hash, K key, V value, HashMap.Entry<K,V> next) {

super(hash, key, value, next);

}

}

v).LinkedHashMap与LRU(Least recently used):LinkedHashMap有一个属性是accessOrder, LinkedHashMap的所有构造函数, 除了三个参数的LinkedHashMap(int initialCapacity, float loadFactor, Boolean accessOrder)以外的其余全部初始默认accessOrder为false, 当accessOrder为false时, 没有开启LRU的,每次插入元素都会插入到末尾,而且对于LinkedHashMap的其他操作并不会影响其中元素的顺序, 当accessOrder为true时, 插入元素仍会插入到双链表的末尾,但是其他操作会影响元素在链表中的顺序.例如:

例1:

public class LinkedHashMapTest {

public static void main(String[] args) {

LinkedHashMap<Integer, Integer> map = new LinkedHashMap<Integer, Integer>();//默认accessOrder为false

for (int i = 0; i < 10; i++) {//按顺序放入1~9

map.put(i, i);

}

System.out.println("原数据："+map.toString());

map.get(3);

System.out.println("查询存在的某一个："+map.toString());

map.put(4, 4);

System.out.println("插入已存在的某一个："+map.toString()); //直接调用已存在的toString方法，不然自己需要用迭代器实现

map.put(10, 10);

System.out.println("插入一个原本没存在的："+map.toString());

}

//输出结果

// 原数据：{0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=7, 8=8, 9=9}

// 查询存在的某一个：{0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=7, 8=8, 9=9}

// 插入已存在的某一个：{0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=7, 8=8, 9=9}

// 插入一个原本没存在的：{0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=7, 8=8, 9=9, 10=10}

}

例2:

public class LinkedHashMapTest {

public static void main(String[] args) {

LinkedHashMap<Integer, Integer> map = new LinkedHashMap<Integer, Integer>(20, 0.75f, true);

for (int i = 0; i < 10; i++) {//按顺序放入1~9

map.put(i, i);

}

System.out.println("原数据："+map.toString());

map.get(3);

System.out.println("查询存在的某一个："+map.toString());

map.put(4, 4);

System.out.println("插入已存在的某一个："+map.toString()); //直接调用已存在的toString方法，不然自己需要用迭代器实现

map.put(10, 10);

System.out.println("插入一个原本没存在的："+map.toString());

}

//输出结果

// 原数据：{0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=7, 8=8, 9=9}

// 查询存在的某一个：{0=0, 1=1, 2=2, 4=4, 5=5, 6=6, 7=7, 8=8, 9=9, 3=3} //被访问（get）的3放到了最后面

// 插入已存在的某一个：{0=0, 1=1, 2=2, 5=5, 6=6, 7=7, 8=8, 9=9, 3=3, 4=4}//被访问（put）的4放到了最后面

// 插入一个原本没存在的：{0=0, 1=1, 2=2, 5=5, 6=6, 7=7, 8=8, 9=9, 3=3, 4=4, 10=10}//新增一个放到最后面

}

可以看到当启用LRU时,最近访问的会移动到双链表的尾部,调用了LinkedHashMap中的afterNodeAccess, afterNodeAccess方法就是如何支持LRU规则的，如果在accessOrder为true的时候，**节点调用这个函数，就会把该节点放置到最后面**。put，get等都会调用这个函数来调整顺序; LinkedHashMap中的get方法调用了该方法, 但是LinkedHashMap中没有put方法,直接调用的是HashMap中的put方法,但是在HashMap中的put方法中是有afterNodeAccess方法的,只是在HashMap中没有实现,而LinkedHashMap又继承自HashMap,所以在LinkedHashMap中对该方法进行实现, 所以LinkedHashMap中没有put方法,调用HashMap中的put方法时仍然会调用LinkedHashMap中的afterNodeAccess方法实现LRU.

vi).LinkedHashMap有序性的维护:

private abstract class LinkedHashIterator<T> implements Iterator<T>; private class KeyIterator extends LinkedHashIterator<K>;(key迭代器)

private class ValueIterator extends LinkedHashIterator<V>;(value迭代器)

private class EntryIterator extends LinkedHashIterator<Map.Entry<K,V>>; LinkedHashMap 增加了双向链表头结点header 和 标志位accessOrder两个属性用于保证迭代顺序。但是要想真正实现其有序性, 但是为了真正保证输出的有序性,**LinkedHashMap重写了HashMap 的迭代器，它使用其维护的双向链表进行迭代输出.**

**WeakedHashMap:** WeakHashMap 继承于AbstractMap，实现了Map接口, 和[HashMap](http://www.cnblogs.com/skywang12345/p/3310835.html)一样，WeakHashMap 也是一个**散列表**，它存储的内容也是**键值对(key-value)映射**，而且**键和值都可以是null**。不过WeakHashMap的**键是“弱键”**。在 WeakHashMap 中，当某个键不再正常使用时，会被从WeakHashMap中被自动移除。更精确地说，对于一个给定的键，其映射的存在并不阻止垃圾回收器对该键的丢弃，这就使该键成为可终止的，被终止，然后被回收。某个键被终止时，它对应的键值对也就从映射中有效地移除了.

**如何实现”弱键”:** **通过WeakReference和ReferenceQueue实现的,** WeakHashMap的key是“弱键”，即是WeakReference类型的；ReferenceQueue是一个队列，它会保存被GC回收的“弱键”。实现步骤是：

i). 新建WeakHashMap，将“**键值对**”添加到WeakHashMap中,  实际上，WeakHashMap是通过数组table保存Entry(键值对)；每一个Entry实际上是一个单向链表，即Entry是键值对链表;

ii).当**某“弱键”不再被其它对象引用**，并**被GC回收**时。在GC回收该“弱键”时，**这个“弱键”也同时会被添加到ReferenceQueue(queue)队列**中;

iii).当下一次我们需要操作WeakHashMap时，会先同步table和queue。table中保存了全部的键值对，而queue中保存被GC回收的键值对；同步它们，就是**删除table中被GC回收的键值对.**

**3).TreeMap,TreeSet的区别以及底层实现**

**TreeMap:**

i). TreeMap 是一个**有序的key-value集合**，它是通过[红黑树](http://www.cnblogs.com/skywang12345/p/3245399.html)实现的;

ii). TreeMap **继承于AbstractMap**，实现NavigableMap接口(继承SortedMap接口),所以它是一个Map，即一个key-value集合;

iii). TreeMap 实现了NavigableMap接口，意味着它**支持一系列的导航方法。**比如返回有序的key集合; TreeMap 实现了Cloneable接口，意味着**它能被克隆;** TreeMap 实现了java.io.Serializable接口，意味着**它支持序列化;**

iv). TreeMap基于**红黑树（Red-Black tree）实现**。该映射根据**其键的自然顺序进行排序**，或者根据**创建映射时提供的 Comparator 进行排序**，具体取决于使用的构造方法; 它包含几个重要的成员变量： root, size, comparator。root 是红黑数的根节点。它是Entry类型，Entry是红黑数的节点，它包含了红黑数的6个基本组成成分：key(键)、value(值)、left(左孩子)、right(右孩子)、parent(父节点)、color(颜色)。Entry节点根据key进行排序，Entry节点包含的内容为value。 红黑数排序时，根据Entry中的key进行排序；Entry中的key比较大小是根据比较器comparator来进行判断的。size是红黑数中节点的个数。

v). TreeMap的基本操作 containsKey、get、put 和 remove 的时间复杂度是 log(n) 。TreeMap是**非同步**的。 它的iterator 方法返回的**迭代器是fail-fastl**的.

vi).序列化自定义实现,对size和所有的元素进行序列化;

vii).TreeMap的遍历:

**遍历TreeMap的键值对:第一步**根据entrySet()获取TreeMap的“键值对”的Set集合, 通过Iterator迭代器遍历“第一步”得到的集合;

Iterator iter = map.entrySet().iterator();

while(iter.hasNext()) {

Map.Entry entry = (Map.Entry)iter.next();

// 获取key

key = (String)entry.getKey();

// 获取value

integ = (Integer)entry.getValue();

}

**遍历TreeMap的键:第一步**根据keySet()获取TreeMap的“键”的Set集合, 通过Iterator迭代器遍历“第一步”得到的集合:

Iterator iter = map.keySet().iterator();

while (iter.hasNext()) {

// 获取key

key = (String)iter.next();

// 根据key，获取value

integ = (Integer)map.get(key);

}

**遍历TreeMap的值:第一步**根据values()获取TreeMap的“值”的集合, 通过Iterator迭代器遍历“第一步”得到的集合:

Collection c = map.values();

Iterator iter= c.iterator();

while (iter.hasNext()) {

value = (Integer)iter.next();

}

**TreeSet(同问题1):**

i). TreeSet是基于TreeMap实现的。TreeSet中的元素支持2种排序方式：自然排序 或者 根据创建TreeSet 时提供的 Comparator 进行排序。这取决于使用的构造方法.

ii). TreeSet为基本操作（add、remove 和 contains）提供受保证的 log(n) 时间开销; TreeSet是非同步的。 它的iterator 方法返回的迭代器是fail-fast的.

iii). TreeSet实际上是TreeMap实现的; 构造TreeSet时,若使用不带参数的构造函数，则TreeSet的使用自然比较器；若用户需要使用自定义的比较器，则需要使用带比较器的参数; TreeSet是非线程安全的; TreeSet实现java.io.Serializable的方式,当写入到输出流时，依次写入“比较器、容量、全部元素”；当读出输入流时，再依次读取(自定义学序列化实现方式)。

iv).TreeSet的迭代方式: **Iterator顺序遍历;转换成数组后用 for-each遍历HashSet.**

**为什么说TreeSet是基于TreeMap实现的:**

在TreeSet的构造函数中: 可以看到调用的是TreeMap的构造函数,只是将key-value中的value固定. 一方面以空间换时间,因为TreeMap基于红黑树实现,所以可以保证TreeSet的基本操作,add,remove,contains的logn的时间,空间复杂度为O(n).

// 不带参数的构造函数。创建一个空的TreeMap

public TreeSet() {

this(new TreeMap<E,Object>());

}

// 将TreeMap赋值给 "NavigableMap对象m"

TreeSet(NavigableMap<E,Object> m) {

this.m = m;

}

// 带比较器的构造函数。

public TreeSet(Comparator<? super E> comparator) {

this(new TreeMap<E,Object>(comparator));

}

// 创建TreeSet，并将集合c中的全部元素都添加到TreeSet中

public TreeSet(Collection<? extends E> c) {

this();

// 将集合c中的元素全部添加到TreeSet中

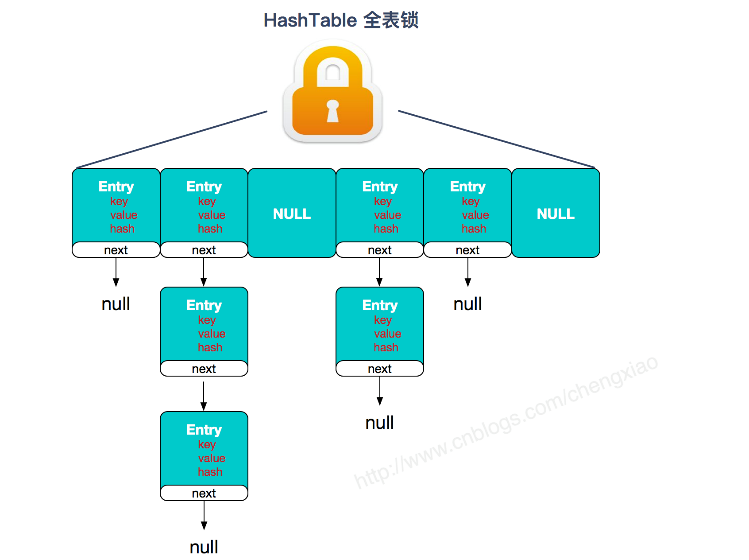
addAll(c);

}

**4).HashMap和ConcurrentHashMap的区别以及底层实现**

**HashMap** ： HashMap是**线程不安全**的，在并发环境下，可能会形成**环状链表**（扩容时可能造成，具体原因见问题2中的HashMap的分析），导致get操作时，cpu空转，所以，在并发环境中使用HashMap是非常危险的.

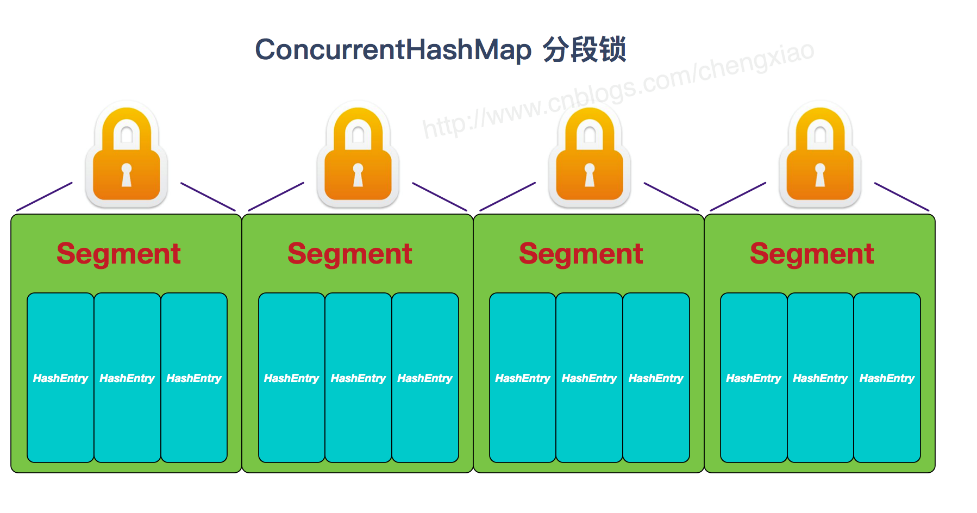
**HashTable** ： HashTable和HashMap的实现原理几乎一样，差别无非是**HashTable不允许key和value为null;HashTable是线程安全的。**但是HashTable线程安全的策略实现代价却太大了，简单粗暴，get/put所有相关操作都是synchronized的，这相当于给整个哈希表加了一把**大锁**，多线程访问时候，只要有一个线程访问或操作该对象，那其他线程只能阻塞，相当于将所有的操作**串行化**，在竞争激烈的并发场景中性能就会非常差.



HashTable性能差主要是由于所有操作需要竞争同一把锁，而如果容器中有多把锁，每一把锁锁一段数据，这样在多线程访问时不同段的数据时，就不会存在锁竞争了，这样便可以有效地提高并发效率。这就是ConcurrentHashMap所采用的"**分段锁**"思想.

**ConcurrentHashMap(JDK1.7):**

HashTable性能差主要是由于所有操作需要竞争同一把锁，而如果容器中有多把锁，每一把锁锁一段数据，这样在多线程访问时不同段的数据时，就不会存在锁竞争了，这样便可以有效地提高并发效率。这就是ConcurrentHashMap所采用的"**分段锁**"思想: 一个ConcurrentHashMap实例中包含由若干个Segment实例组成的数组，而一个Segment实例又包含由若干个桶，每个桶中都包含一条由若干个 HashEntry 对象链接起来的链表。特别地，ConcurrentHashMap 在默认并发级别下会创建16个Segment对象的数组，如果键能均匀散列，每个 Segment 大约守护整个散列表中桶总数的 1/16.



public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>

implements ConcurrentMap<K, V>, Serializable {

final int segmentMask; // 用于定位段，大小等于segments数组的大小减 1，是不可变的

final int segmentShift; // 用于定位段，大小等于32(hash值的位数)减去对segments的大小取以2为底的对数值，是不可变的

final Segment<K,V>[] segments; // ConcurrentHashMap的底层结构是一个Segment数组

}

**ConcurrentHashMap的构造方法:** public ConcurrentHashMap(int initialCapacity,

float loadFactor, int concurrencyLevel)

初始化方法有三个参数，如果用户不指定则会使用默认值，initialCapacity为16(为每一个Segment中table数组的大小)，loadFactor为0.75（负载因子，扩容时需要参考），concurrentLevel(并发级别)为16; **Segment数组的大小ssize是由concurrentLevel来决定的，但是却不一定等于concurrentLevel，ssize一定是大于或等于concurrentLevel的最小的2的次幂。比如：默认情况下concurrentLevel是16，则ssize为16；若concurrentLevel为14，ssize为16；若concurrentLevel为17，则ssize为32(原因与HashMap中的capacity是2的次幂的原因相同).**

ConcurrentHashMap的初始化是会通过位与运算来初始化Segment的大小, 用ssize来表示:

//MAX\_SEGMENTS 为1<<16=65536，也就是最大并发数为65536

if (concurrencyLevel > MAX\_SEGMENTS)

concurrencyLevel = MAX\_SEGMENTS;

//2的sshif次方等于ssize，例:ssize=16,sshift=4;ssize=32,sshif=5

int sshift = 0;

//ssize 为segments数组长度，根据concurrentLevel计算得出

int ssize = 1;

while (ssize < concurrencyLevel) {

++sshift;

ssize <<= 1;

}

concurrencyLevel最大只能用16位的二进制来表示，即65536，换句话说，Segment的大小最多65536个，没有指定concurrencyLevel元素初始化，Segment的大小ssize默认为16.

每一个Segment元素下的HashEntry的初始化也是按照位于运算来计算，用cap来表示，如下所示:

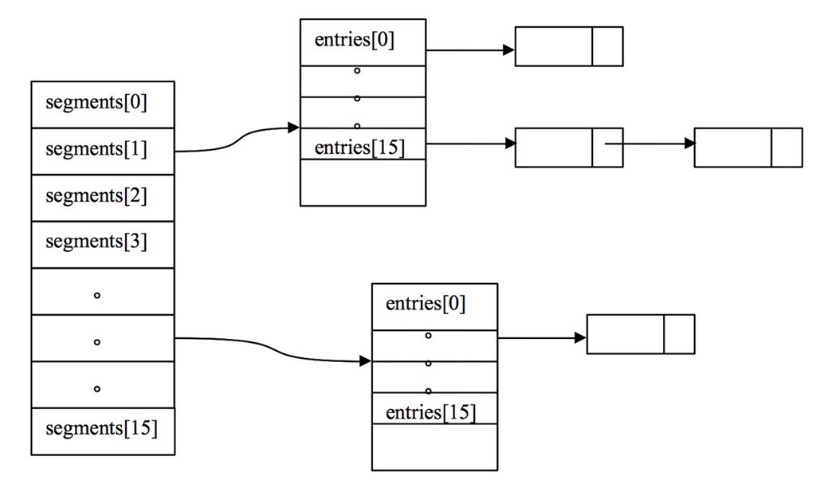
int cap = 1;

while (cap < c)

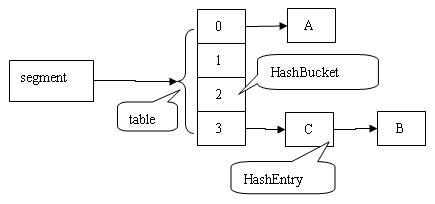
    cap <<= 1;

HashEntry大小的计算也是2的N次方（cap <<=1）， cap的初始值为1，所以HashEntry最小的容量为2.

在JDK1.7版本中，ConcurrentHashMap的数据结构是由一个Segment数组和多个HashEntry(1.8中HashEntry为Node)组成; 把区间按照并发级别(concurrentLevel)，分成了若干个segment。默认情况下内部按并发级别为16来创建。对于每个segment的容量，默认情况也是16。当然并发级别(concurrentLevel)和每个段(segment)的初始容量都是可以通过构造函数设定的:



**Segment的定义**:static final class Segment<K,V> extends ReentrantLock implements Serializable; Segment继承了ReentrantLock，表明每个segment都可以当做一个锁. 这样对每个segment中的数据需要同步操作的话都是使用每个segment容器对象自身的锁来实现, 只有对全局需要改变时锁定的是所有的segment. Segment中的成员变量count, count是volatile的, 每个 Segment 对象中包含一个计数器，而不是在 ConcurrentHashMap 中使用全局的计数器，是对 ConcurrentHashMap 并发性的考虑：**因为这样当需要更新计数器时，不用锁定整个ConcurrentHashMap.ConcurrentHashMap允许多个修改(写)操作并发进行，其关键在于使用了锁分段技术，它使用了不同的锁来控制对哈希表的不同部分进行的修改(写)，而 ConcurrentHashMap 内部使用段(Segment)来表示这些不同的部分。**实际上，每个段实质上就是一个小的哈希表，每个段都有自己的锁(Segment 类继承了 ReentrantLock 类)。这样，只要多个修改(写)操作发生在不同的段上，它们就可以并发进行。下图是依次插入 ABC 三个 HashEntry 节点后，Segment 的结构示意图：



**HashEntry:** 在HashEntry类中，key，hash和next域都被声明为final的，value域被volatile所修饰，因此HashEntry对象几乎是不可变的，这是ConcurrentHashmap读操作并不需要加锁的一个重要原因。next域被声明为final本身就意味着我们不能从hash链的中间或尾部添加或删除节点，因为这需要修改next引用值，因此所有的节点的修改只能从头部开始。对于put操作，可以一律添加到Hash链的头部。但是对于remove操作，可能需要从中间删除一个节点，这就需要将要删除节点的前面所有节点整个复制(重新new)一遍，最后一个节点指向要删除结点的下一个结点. 由于value域被volatile修饰，所以其可以确保被读线程读到最新的值，这是ConcurrentHashmap读操作并不需要加锁的另一个重要原因。实际上，ConcurrentHashMap完全允许多个读操作并发进行，读操作并不需要加锁.

static final class HashEntry<K,V> {

final K key; // 声明 key 为 final 的

final int hash; // 声明 hash 值为 final 的

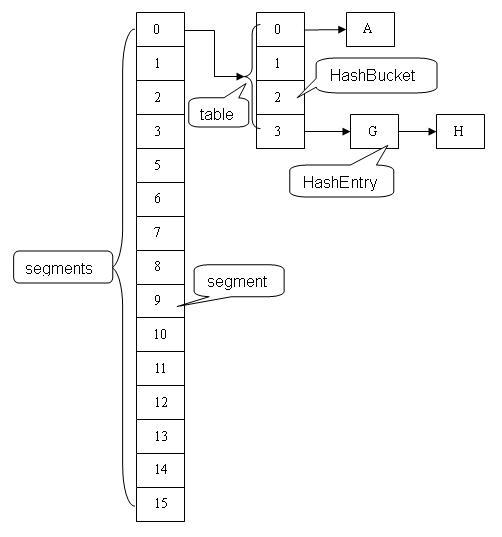
volatile V value; // 声明 value 被volatile所修饰

final HashEntry<K,V> next; // 声明 next 为 final 的

}

**Segment中采用分离锁技术**: *分拆锁(lock spliting)就是若原先的程序中多处逻辑都采用同一个锁，但各个逻辑之间又相互独立，就可以拆(Spliting)为使用多个锁，每个锁守护不同的逻辑。 分拆锁有时候可以被扩展，分成可大可小加锁块的集合，并且它们归属于相互独立的对象，这样的情况就是****分离锁****(lock striping)。*

**ConcurrentHashMap使用分段锁Segment来保护不同段的数据:通过使用段(Segment)将ConcurrentHashMap划分为不同的部分，ConcurrentHashMap就可以使用不同的锁来控制对哈希表的不同部分的修改，从而允许多个修改操作并发进行, 这正是ConcurrentHashMap锁分段技术的核心内涵:**



在插入和获取元素的时候，必须先通过哈希算法定位到Segment:

**get操作:** ConcurrentHashMap的get操作是直接委托给Segment的get方法(第一次需要经过一次hash定位到Segment的位置，然后再hash定位到指定的HashEntry，遍历该HashEntry下的链表进行对比，成功就返回，不成功就返回null), get操作不需要锁, 除非读到的值是空的才会加锁重读， HashTable容器的get方法是需要加锁的, ConcurrentHashMap的get操作是它的get方法里将要使用的共享变量都定义成volatile.

V get(Object key, int hash) {

if (count != 0) { // read-volatile，首先读 count 变量

HashEntry<K,V> e = getFirst(hash); // 获取桶中链表头结点

while (e != null) {

if (e.hash == hash && key.equals(e.key)) { // 查找链中是否存在指定Key的键值对

V v = e.value;

if (v != null) // 如果读到value域不为 null，直接返回

return v;

// 如果读到value域为null，说明发生了重排序，加锁后重新读取

return readValueUnderLock(e); // recheck

}

e = e.next;

}

}

return null; // 如果不存在，直接返回null

}

**get操作不加锁的原理:** HashEntry中的key、hash和next指针都是final的。这意味着，我们不能把节点添加到链表的中间和尾部，也不能在链表的中间和尾部删除节点。这个特性可以保证：在访问某个节点时，这个节点之后的链接不会被改变，这个特性可以大大降低处理链表时的复杂性。与此同时，由于HashEntry类的value字段被声明是Volatile的，因此Java的内存模型就可以保证：某个写线程对value字段的写入马上就可以被后续的某个读线程看到。此外，由于在ConcurrentHashMap中不允许用null作为键和值，所以当读线程读到某个HashEntry的value为null时，便知道产生了冲突 —— 发生了重排序现象，此时便会加锁重新读入这个value值。这些特性互相配合，使得读线程即使在不加锁状态下，也能正确访问 ConcurrentHashMap。ConcurrentHashMap就是一个Segment数组，而每个Segment都有一个volatile变量count去统计Segment中的HashEntry的个数。并且，在ConcurrentHashMap中，所有不加锁读方法在进入读方法时，首先都会去读这个count变量。比如我们在上一节提到的get方法.

**put操作:** **ConcurrentHashMap不同于HashMap，它既不允许key值为null，也不允许value值为null。**put操作的代码如下:

public V put(K key, V value) {

if (value == null)

throw new NullPointerException();

int hash = hash(key.hashCode());

return segmentFor(hash).put(key, hash, value, false);

}

此外，还可以看到，实际上对ConcurrentHashMap的put操作被ConcurrentHashMap委托给特定的段来实现。也就是说，当向ConcurrentHashMap中put一个Key/Value对时，首先会获得Key的哈希值并对其再次哈希，然后根据最终的hash值定位到这条记录所应该插入的段，定位段的segmentFor()方法源码如下:

final Segment<K,V> segmentFor(int hash) {

return segments[(hash >>> segmentShift) & segmentMask];

}

segmentFor()方法根据传入的hash值向右无符号右移segmentShift位，然后和segmentMask进行与操作就可以定位到特定的段。**根据key的hash值的高n位就可以确定元素到底在哪一个Segment中**(具体不做分析);紧接着，调用这个段的put()方法来将目标Key/Value对插到段中，段的put()方法的源码如下所示:

V put(K key, int hash, V value, boolean onlyIfAbsent) {

lock(); // 上锁

try {

int c = count;

if (c++ > threshold) // ensure capacity

rehash();

HashEntry<K,V>[] tab = table; // table是Volatile的

int index = hash & (tab.length - 1); // 定位到段中特定的桶

HashEntry<K,V> first = tab[index]; // first指向桶中链表的表头

HashEntry<K,V> e = first;

// 检查该桶中是否存在相同key的结点

while (e != null && (e.hash != hash || !key.equals(e.key)))

e = e.next;

V oldValue;

if (e != null) { // 该桶中存在相同key的结点

oldValue = e.value;

if (!onlyIfAbsent)

e.value = value; // 更新value值

}else { // 该桶中不存在相同key的结点

oldValue = null;

++modCount; // 结构性修改，modCount加1

tab[index] = new HashEntry<K,V>(key, hash, first, value); // 创建HashEntry并将其链到表头

count = c; //write-volatile，count值的更新一定要放在最后一步(volatile变量)

}

return oldValue; // 返回旧值(该桶中不存在相同key的结点，则返回null)

} finally {

unlock(); // 在finally子句中解锁

}

}

从源码可以看出, ConcurrentHashMap对Segment的put操作是加锁完成的; Segment是ReentrantLock的子类，因此Segment本身就是一种可重入的Lock，所以我们可以直接调用其继承而来的lock()方法和unlock()方法对代码进行上锁/解锁, 这里的加锁操作是针对某个具体的Segment，锁定的也是该Segment而不是整个ConcurrentHashMap, 因为插入键/值对操作只是在这个Segment包含的某个桶中完成，不需要锁定整个ConcurrentHashMap。因此，其他写线程对另外15个Segment的加锁并不会因为当前线程对这个Segment的加锁而阻塞。故而 **相比较于 HashTable 和由同步包装器包装的HashMap每次只能有一个线程执行读或写操作，ConcurrentHashMap 在并发访问性能上有了质的提高。在理想状态下，ConcurrentHashMap 可以支持 16 个线程执行并发写操作（如果并发级别设置为 16），及任意数量线程的读操作.** 在将Key/Value对插入到Segment之前，首先会检查本次插入会不会导致Segment中元素的数量超过阈值threshold，如果会，那么就先对Segment进行扩容和重哈希操作，然后再进行插入.

**rehash操作:**通过上述的put方法可以看到,当插入元素导致当前Segment中的数量超过threshold时,会导致扩容和rehash操作, **特别需要注意的是，ConcurrentHashMap的重哈希实际上是对ConcurrentHashMap的某个段的重哈希，因此ConcurrentHashMap的每个段所包含的桶位自然也就不尽相同. 由于扩容是按照2的幂次方进行的，所以扩展前在同一个桶中的元素，现在要么还是在原来的序号的桶里，或者就是原来的序号再加上一个2的幂次方，就这两种选择.**

**size操作:** size操作属于跨段操作, 相似的操作还有containsValues操作, 以size操作为例, 计算ConcurrentHashMap的元素大小时，因为是并发操作的，就是在计算size的时候，他还在并发的插入数据，可能会导致计算出来的size和实际的size有相差（在return size的时候，插入了多个数据）; ConcurrentHashMap会在不上锁的前提逐个Segment计算3次size，如果某相邻两次计算获取的所有Segment的更新次数（每个Segment都与HashMap一样通过modCount跟踪自己的修改次数，Segment每修改一次其modCount加一）相等，说明这两次计算过程中无更新操作，则这两次计算出的总size相等，可直接作为最终结果返回。如果这三次计算过程中数据有更新，则对所有Segment加锁重新计算size.

try {

    for (;;) {

        if (retries++ == RETRIES\_BEFORE\_LOCK) {

            for (int j = 0; j < segments.length; ++j) ensureSegment(j).lock(); // force creation

        }

        sum = 0L;

        size = 0;

        overflow = false;

        for (int j = 0; j < segments.length; ++j) {

            Segment<K,V> seg = segmentAt(segments, j);

            if (seg != null) { sum += seg.modCount; int c = seg.count; if (c < 0 || (size += c) < 0)

               overflow = true;

            } }

        if (sum == last) break;

        last = sum; } }

finally {

    if (retries > RETRIES\_BEFORE\_LOCK) {

        for (int j = 0; j < segments.length; ++j)

            segmentAt(segments, j).unlock();

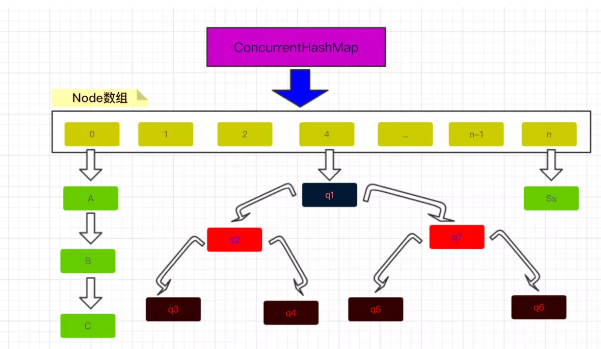
    }

}

**扩容操作:**在put操作进入Segment时会判断当前Segment(每一个Segment是一个HashEntry数组+链表构成,每一个Segment相当于一个单独的HashMap)的容量是否超过阈值,如果超过阈值则进行扩容, 具体扩容过程与JDK1.7中的HashMap的扩容机制相似, Segment的扩容判断比HashMap更恰当，因为HashMap是在插入元素后判断元素是否已经到达容量的，如果到达了就进行扩容，但是很有可能扩容之后没有新元素插入，这时HashMap就进行了一次无效的扩容, 扩容的时候首先会创建一个两倍于原容量的数组，然后将原数组里的元素进行再hash后插入到新的数组里。为了高效ConcurrentHashMap不会对整个容器进行扩容，而只对某个segment进行扩容

**ConcurrentHashMap在JDK1.8中的实现:**

在1.8中ConcurrentHashMap与1.8中的HashMap相似, ConcurrentHashMap使用了一个table来存储Node，ConcurrentHashMap同样使用记录的key的hashCode来寻找记录的存储index，而处理哈希冲突的方式与HashMap也是类似的，冲突的记录将被存储在同一个位置上，形成一条链表，当链表的长度大于8的时候会将链表转化为一棵红黑树，从而将查找的复杂度从O(N)降到了O(lgN):



**哈希桶的初始化:** 初始化table的工作将发生在进行put操作时，如果发现table还没有被初始化，那么就会调用方法initTable来进行table的初始化:

private transient volatile int sizeCtl;

private final Node<K,V>[] initTable() {

Node<K,V>[] tab; int sc;

while ((tab = table) == null || tab.length == 0) {

if ((sc = sizeCtl) < 0)

Thread.yield(); // lost initialization race; just spin

else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {

try {

if ((tab = table) == null || tab.length == 0) {

int n = (sc > 0) ? sc : DEFAULT\_CAPACITY;

@SuppressWarnings("unchecked")

Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];

table = tab = nt;

sc = n - (n >>> 2);

}

} finally {

sizeCtl = sc;

}

break;

}

}

return tab;

}

sizeCtl是一个用于同步多个线程的共享变量，如果它的当前值为负数，则说明table(不是Segment)正在被某个线程初始化或者扩容，所以，如果某个线程想要初始化table或者对table扩容，需要去竞争sizeCtl这个共享变量，获得变量的线程才有许可去进行接下来的操作，没能获得的线程将会一直自旋来尝试获得这个共享变量，所以获得sizeCtl这个变量的线程在完成工作之后需要设置回来，使得其他的线程可以走出自旋进行接下来的操作。而在initTable方法中我们可以看到，当线程发现sizeCtl小于0的时候，他就会让出CPU时间，而稍后再进行尝试，当发现sizeCtl不再小于0的时候，就会通过调用方法compareAndSwapInt来将sizeCtl共享变量变为-1，以告诉其他试图获得sizeCtl变量的线程，目前正在由本线程在享用该变量，在我完成我的任务之前你可以先休息一会，等会再来试，我完成工作之后会释放掉的。而其他的线程在发现sizeCtl小于0的时候就会理解这种交流，他们会让出cpu时间，等待下次调度再来尝试获取sizeCtl来进行自己的工作。在完成初始化table的任务之后，线程需要将sizeCtl设置成可以使得其他线程获得变量的状态，这其中还有一个地方需要注意，就是在某个线程通过U.compareAndSwapInt方法设置了sizeCtl之前和之后进行了两次check，来检测table是否被初始化过了，这种检测是必须的，因为在并发环境下，可能前一个线程正在初始化table但是还没有成功初始化，也就是table依然还为null，而有一个线程发现table为null他就会进行竞争sizeCtl以进行table初始化，但是当前线程在完成初始化之后，那个试图初始化table的线程获得了sizeCtl，但是此时table已经被初始化了，所以，如果没有再次判断的话，可能会将之后进行put操作的线程的更新覆盖掉，这是极为不安全的行为.

**get方法:** 在ConcurrentHashMap中查询一条记录首先需要知道这条记录存储的table的位置（可以称为卡槽，每个卡槽中都会有一个链表或者一棵红黑树），该位置上可能为null，如果为null，说明想要查询的记录还不存在于ConcurrentHashMap中，否则，就在该位置上的链表或者红黑树中查找记录.

public V get(Object key) {

Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;

int h = spread(key.hashCode());

if ((tab = table) != null && (n = tab.length) > 0 &&

(e = tabAt(tab, (n - 1) & h)) != null) {

if ((eh = e.hash) == h) {

if ((ek = e.key) == key || (ek != null && key.equals(ek)))

return e.val;

}

else if (eh < 0)

return (p = e.find(h, key)) != null ? p.val : null;

while ((e = e.next) != null) {

if (e.hash == h &&

((ek = e.key) == key || (ek != null && key.equals(ek))))

return e.val;

}

}

return null;

}

首先，计算出记录的key的hashCode，然后通过使用(hashCode & (length - 1))的计算方法来获得该记录在table中的index，然后判断该位置上是否为null，如果为null，则返回null，否则，如果该位置上的第一个元素（链表头节点或者红黑树的根节点）与我们先要查找的记录匹配，则直接返回这个节点的值，否则，如果该节点的hashCode小于0，则说明该位置上是一颗红黑树(在红黑树TreeNode类中定义了TREEBIN=-2表示是根节点), 如果是红黑树，则通过调用Node的find方法来查找到节点，而这个Node的find方法在子类中被重写，所以会直接调用子类的find方法来进行查找。还有一种情况是table的index位置上为一条链表，那么就通过链表的查找方法来进行记录查找; 同样发现get方法并没有使用任何与锁等效的手段来保证线程安全, 因为ConcurrentHashMap中table数组的类Node中value和next指针是volatile修饰的,从而能够保证内存可见性.

static class Node<K,V> implements Map.Entry<K,V> {

final int hash;

final K key;

volatile V val;

volatile Node<K,V> next;

....

}

**put方法:** 在进行put操作的时候，可能table数组还没有初始化的情况，或者发现table中容纳的记录数量超过了阈值的情况，前者需要进行table的初始化，而后者需要对table进行扩容操作, put一个记录需要的过程: 第一，我们需要计算这个记录的key的hashCode，并且根据hashCode来计算它在table数组中应该存储的index，然后将他存放到对应位置里面的链表或者红黑树中去，并且在某些情况下要进行链表转换红黑树的操作，以及table扩容操作,更细size操作等.

final V putVal(K key, V value, boolean onlyIfAbsent) {

if (key == null || value == null) throw new NullPointerException();

int hash = spread(key.hashCode());

int binCount = 0;

for (Node<K,V>[] tab = table;;) {

Node<K,V> f; int n, i, fh;

if (tab == null || (n = tab.length) == 0)

tab = initTable();

else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {

if (casTabAt(tab, i, null,

new Node<K,V>(hash, key, value, null)))

break; // no lock when adding to empty bin

}

else if ((fh = f.hash) == MOVED)

tab = helpTransfer(tab, f);

else {

V oldVal = null;

synchronized (f) {

if (tabAt(tab, i) == f) {

if (fh >= 0) {

binCount = 1;

for (Node<K,V> e = f;; ++binCount) {

K ek;

if (e.hash == hash &&

((ek = e.key) == key ||

(ek != null && key.equals(ek)))) {

oldVal = e.val;

if (!onlyIfAbsent)

e.val = value;

break;

}

Node<K,V> pred = e;

if ((e = e.next) == null) {

pred.next = new Node<K,V>(hash, key,

value, null);

break;

}

}

}

else if (f instanceof TreeBin) {

Node<K,V> p;

binCount = 2;

if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,

value)) != null) {

oldVal = p.val;

if (!onlyIfAbsent)

p.val = value;

}

}

}

}

if (binCount != 0) {

if (binCount >= TREEIFY\_THRESHOLD)

treeifyBin(tab, i);

if (oldVal != null)

return oldVal;

break;

}

}

}

addCount(1L, binCount);

return null;

}

首先，计算记录的key的hashCode，然后计算table的index位置，然后获取该index的值，如果该位置还为null，说明该位置上还没有记录，则通过调用casTabAt(支持CAS操作)方法来讲该新的记录插入到table的index位置上去，否则，通过synchronized关键字对table的index位置加锁，需要注意的是，当前线程只会锁住table的index位置，其他位置上没有锁住，所以此时其他线程可以安全的获得其他的table位置来进行操作。这也就提高了ConcurrentHashMap的并发度(在1.7之前是通过锁住某一个段Segment是达到目的)。然后判断table的index位置上的第一个节点的hashCode值，这个节点要么是链表的头节点，要么是红黑树的根节点，如果hashCode值小于0，那么就是一颗红黑树，如果不小于0，那么就还是一条链表，如果是一条链表，那么就寻找是否已经有记录的key和当前想要插入的记录是一致的，如果一致，那么这次put的效果就是replace，否则，将该记录添加到链表中去。如果是一颗红黑树，那么就通过调用putTreeVal方法来进行插入操作。在插入操作完成之后，需要判断本次操作是否是更新操作，如果是更新操作，则不会造成size的变化，否则，如果本次put操作时一次添加操作，那么就需要进行更新size的操作,同时, table的扩容操作也会在更新size的时候发生，如果在更新size之后发现table中的记录数量达到了阈值，就需要进行扩容操作.

需要注意的是HashMap中是可以允许key和value为null的,但是ConcurrentHashMap中是不允许key和value为null,否则报出NPE(空指针异常), 在ConcurrentHashMap中可以通过使用get操作来测试是否具有某个记录，因为只要get方法返回null，就说明table中必然不存在一个记录和当前查询的匹配，而在HashMap中，get操作返回null有可能是我们查询的记录的value就是null，所以不能使用get方法来测试某个记录是否存在于table中.

**size操作:** 在完成一次put操作之后，需要更新table中的记录数量，并且在更新之后如果发现超出了阈值，那么就需要进行table扩容操作,通过addCount实现. ConcurrentHashMap维护baseCount(transient volatile long baseCount)来表示当前的记录数量，这在后面获取记录数量的size方法中会用到，而在put操作和remove操作的时候回通过调用方法addCount来更新它. size操作只要通过baseCount和counterCells数组来得到,通过累计两者的数量即可获得当前ConcurrentHashMap中的记录总量:

public int size() {

long n = sumCount();

return ((n < 0L) ? 0 :

(n > (long)Integer.MAX\_VALUE) ? Integer.MAX\_VALUE :

(int)n);

}

final long sumCount() {

CounterCell[] as = counterCells; CounterCell a;

long sum = baseCount;

if (as != null) {

for (int i = 0; i < as.length; ++i) {

if ((a = as[i]) != null)

sum += a.value;

}

}

return sum;

}

**扩容操作:** jdk8中，采用多线程扩容。整个扩容过程，通过CAS设置sizeCtl，transferIndex等变量协调多个线程进行**并发扩容.**

**nextTable成员变量:**扩容期间，将table数组中的元素 迁移到 nextTable:

/\*\*

扩容时，将table中的元素迁移至nextTable . 扩容时非空

\*/

private transient volatile Node<K,V>[] nextTable;

**sizeCtl属性:**多线程之间，以volatile的方式读取sizeCtl属性，来判断ConcurrentHashMap当前所处的状态。通过cas设置sizeCtl属性，告知其他线程ConcurrentHashMap的状态变更:

未初始化：sizeCtl=0：表示没有指定初始容量;sizeCtl>0：表示初始容量。

初始化中：sizeCtl=-1,标记作用，告知其他线程，正在初始化

正常状态：sizeCtl=0.75\*n ,扩容阈值

扩容中:sizeCtl < 0 : 表示有其他线程正在执行扩容;sizeCtl = (resizeStamp(n) << RESIZE\_STAMP\_SHIFT) + 2 :表示此时只有一个线程在执行扩容

**transferIndex属性**:扩容索引，表示已经分配给扩容线程的table数组索引位置,主要用来协调多个线程，并发安全地获取迁移任务（hash桶）.

private transient volatile int transferIndex;

/\*\*

扩容线程每次最少要迁移16个hash桶

\*/

private static final int MIN\_TRANSFER\_STRIDE = 16;

可以将待迁移的table数组，看成一个任务队列，transferIndex看成任务队列的头指针。而扩容线程，就是这个队列的消费者。扩容线程通过CAS设置transferIndex索引的过程，就是消费者从任务队列中获取任务的过程。为了性能考虑，当然不会每次只获取一个任务（hash桶），因此ConcurrentHashMap规定，每次至少要获取16个迁移任务（迁移16个hash桶，MIN\_TRANSFER\_STRIDE = 16）.

**cas设置transferIndex的源码如下:**

private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {

//计算每次迁移的node个数

if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN\_TRANSFER\_STRIDE)

stride = MIN\_TRANSFER\_STRIDE; // 确保每次迁移的node个数不少于16个

for (int i = 0, bound = 0;;) {

...

//cas无锁算法设置 transferIndex = transferIndex - stride

if (U.compareAndSwapInt

(this, TRANSFERINDEX, nextIndex,

nextBound = (nextIndex > stride ?

nextIndex - stride : 0))) {

...

}

...//省略迁移逻辑

}

}

**ForwardingNode节点:** 标记作用，表示其他线程正在扩容，并且此节点已经扩容完毕. 关联了nextTable,扩容期间可以通过find方法，访问已经迁移到了nextTable中的数据.

static final class ForwardingNode<K,V> extends Node<K,V> {

final Node<K,V>[] nextTable;

ForwardingNode(Node<K,V>[] tab) {

//hash值为MOVED（-1）的节点就是ForwardingNode

super(MOVED, null, null, null);

this.nextTable = tab;

}

//通过此方法，访问被迁移到nextTable中的数据

Node<K,V> find(int h, Object k) {

...

}

}

**扩容的时间点:**

i).当容量超过阈值时扩容:

final V putVal(K key, V value, boolean onlyIfAbsent) {

...

addCount(1L, binCount);

...

}

private final void addCount(long x, int check) {

...

if (check >= 0) {

Node<K,V>[] tab, nt; int n, sc;

//s>=sizeCtl 即容量达到扩容阈值，需要扩容

while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&

(n = tab.length) < MAXIMUM\_CAPACITY) {

//调用transfer()扩容

...

}

}

}

ii).当链表中元素个数超过8个,数组的大小还未超过64,此时进行数组的扩容,如果超过则将链表转换为红黑树;

iii).如果当前线程发现有其他线程正在扩容时,可以协助其他线程扩容:

final V putVal(K key, V value, boolean onlyIfAbsent) {

...

//f.hash == MOVED 表示为：ForwardingNode，说明其他线程正在扩容

else if ((fh = f.hash) == MOVED)

tab = helpTransfer(tab, f);

...

}

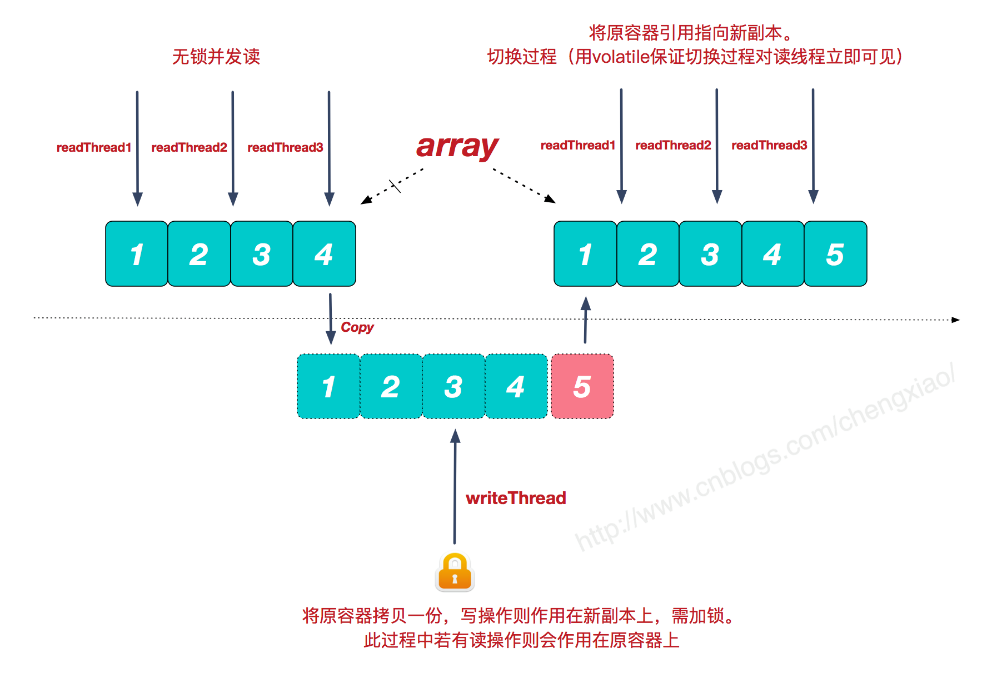
**5).ArrayList与CopyOnWriteArrayList的区别以及底层实现**

**CopyOnWriteArrayList:** Java并发包中提供的一个并发容器，它是个**线程安全且读操作无锁的ArrayList**，写操作则通过创建底层数组的新副本来实现，是一种**读写分离**的并发策略，我们也可以称这种容器为"写时复制器"，Java并发包中类似的容器还有CopyOnWriteSet.

**写时复制:** Copy-On-Write简称COW，是一种用于程序设计中的优化策略。其基本思路是，从一开始大家都在共享同一个内容，当某个人想要修改这个内容的时候，才会真正把内容Copy出去形成一个新的内容然后再改，这是一种延时懒惰策略。从JDK1.5开始Java并发包里提供了两个使用CopyOnWrite机制实现的并发容器,它们是CopyOnWriteArrayList和CopyOnWriteArraySet

**CopyOnWriteArrayList源码分析:**

集合框架中的ArrayList是非线程安全的，Vector虽是线程安全的，但由于简单粗暴的锁同步机制，性能较差。而CopyOnWriteArrayList则提供了另一种不同的并发处理策略, 多数情况下系统应该的是**读多写少**的并发场景, CopyOnWriteArrayList容器允许并发读，读操作是无锁的，性能较高。至于写操作，比如向容器中添加一个元素，**则首先将当前容器复制一份，然后在新副本上执行写操作，结束之后再将原容器的引用指向新容器。**



CopyOnWriteArrayList实现List, RandomAccess, Cloneable, Serializable接口,多用在读多写少的场景下,在进行读操作时,由于CopyOnWriteArrayList中的数组对象Object[]是volatile修饰的,保证了内存可见性,所以读操作时可见的,读操作不需要加锁.对于写操作add,需要加锁ReentrantLock lock = this.lock.

public boolean add(E e) {

//ReentrantLock加锁，保证线程安全

final ReentrantLock lock = this.lock;

lock.lock();

try {

Object[] elements = getArray();

int len = elements.length;

//拷贝原容器，长度为原容器长度加一

Object[] newElements = Arrays.copyOf(elements, len + 1);

//在新副本上执行添加操作

newElements[len] = e;

//将原容器引用指向新副本

setArray(newElements);

return true;

} finally {

//解锁

lock.unlock();

}

}

添加的逻辑，先将原容器copy一份，然后在新副本上执行写操作，之后再切换引用。此过程是要加锁的.

删除操作remove同add操作的原理, 需要拷贝一个新的副本, 删除后再更新原数组.

**使用场景:** CopyOnWrite并发容器用于读多写少的并发场景.

**CopyOnWriteArrayList的优点:** 读操作性能很高，因为无需任何同步措施，比较适用于**读多写少**的并发场景。Java的list在遍历时，若中途有别的线程对list容器进行修改，则会抛出**ConcurrentModificationException**异常。而CopyOnWriteArrayList由于其"读写分离"的思想，遍历和修改操作分别作用在不同的list容器，所以在使用迭代器进行遍历时候，也就不会抛出ConcurrentModificationException异常了.

**CopyOnWriteArrayList的缺点:** **一是内存占用问题**，每次执行写操作都要将原容器拷贝一份，数据量大时，对内存压力较大，可能会引起频繁GC；**二是无法保证实时性**，Vector对于读写操作均加锁同步，可以保证读和写的强一致性。而CopyOnWriteArrayList由于其实现策略的原因，写和读分别作用在新老不同容器上，在写操作执行过程中，读不会阻塞但读取到的却是老容器的数据(CopyOnWriteArrayList与ConcurrentHashMap均是弱一致性的,为了均衡效率考虑,如果满足强一致性就需要额外加锁,效率就会降低).

**6).EnumSet**

**EnumSet:** 是Set接口的一个实现(继承AbstractSet)，它只能用来存储Enum常量或其子类，不能存储其它类型。EnumSet是设计模式中工厂方法创建实例的一个很好例子,EnumSet被声明为abstract class类型，EnumSet有两种实现方式，RegularEnumSet和JumboEnumSet，但是这两种实现方式是包私有的，不能在包外访问，因此必须使用工厂方法来创建并返回EnumSet实例，不能通过构造函数来创建。EnumSet中提供了多种创建EnumSet实例的静态工厂方法，例如of方法（进行了函数重载），copyOf方法，noneOf方法等.

**EnumSet的实现:** EnumSet是一个抽象类，有两个具体实现：java.util.RegularEnumSet和java.util.JumboEnumSet。二者的主要区别在于RegularEnumSet使用long来表示元素的数量，而JumboEnumSet使用long[]来表示元素的数量。这二者中表示元素数量使用的是位域结构，即通过long的二进制位数来表示元素数量，例如：RegularEnumSet使用的是long表示元素数量，long数值是通过64位二进制表示的，因此其只能包含的元素最大数量为64，如果元素数目大于64，采用的是JumboEnumSet表示，JumboEnumSet的long[]中有一个long数值，就能表示64个元素，两个long数值就能表示128个元素，以此类推.

**EnumSet的使用场景:** 需要对枚举类型进行特定的分组时，你可以使用EnumSet。例如，一周中有七天，你想将周末单独分出来的时候（enum int pattern）:

private enum DayOfWeek{

MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

}

private EnumSet weekend = EnumSet.of(SATURDAY,SUNDAY);

EnumSet的注意点:

i). 一个EnumSet中只能包含一种枚举类型;

ii).EnumSet中不能放入null元素，放入会抛出空指针异常;

iii).EnumSet是线程非安全的;

iv).EnumSet的Iterator是自动防故障和弱一致的，不会抛出并发修改异常，即在迭代过程的中的修改结果不一定会在迭代过程中显示;

v).EnumSet是高性能的Java集合。由于是基于数组的访问，因此add，contains，next方法的时间复杂度为O(1), 存储枚举常量时使用EnumSet而不要用HashSet.

**7).线程安全的集合类**

[见多线程中的问题6](#_6).多线程集合类：)

**8).Iterator与Enumeration比较**

在Java集合中，通常都通过 “Iterator(迭代器)” 或 “Enumeration(枚举类)” 去遍历集合。

**Enumeration:**

public interface Enumeration<E> {

boolean hasMoreElements();

E nextElement();

}

**Iterator:**

public interface Iterator<E> {

boolean hasNext();

E next();

default void remove(){throw new UnsupportedOperationException(“remove”)}

default void forEachRemaining(Consumer<? Super E> action){

Objects.requireNonNull(action);

while(hashNext()){action.accept(next());}

}

}

**二者之间的区别:**

i).Iterator添加了可选的remove()方法（“可选”意味着一个实现Iterator接口的类可以选择不实现remove()方法）以及forEachRemaining方法(对余下的所有元素进行指定的操作).

ii).Iterator支持”fast-fail”机制,如果在遍历的过程中底下的容器发生了结构变化（例如add或者remove了元素），则它会抛出ConcurrentModificationException, 而Enumeration没有这种检查机制.

iii).Iterator可以配合Iterable接口用于for-each循环中.

iv). Enumeration 是JDK 1.0添加的接口。使用到它的函数包括Vector、Hashtable等类，这些类都是JDK 1.0中加入的，Enumeration存在的目的就是为它们提供遍历接口。Enumeration本身并没有支持同步，而在Vector、Hashtable实现Enumeration时，添加了同步, 而Iterator 是JDK 1.2才添加的接口，它也是为了HashMap、ArrayList等集合提供遍历接口。Iterator是支持fail-fast机制的：当多个线程对同一个集合的内容进行操作时，就可能会产生fail-fast事件.

**9).fast-fail机制**

“快速失败”也就是fail-fast，它是Java集合的一种错误检测机制。当多个线程对集合进行结构上的改变的操作时，有可能会产生fail-fast机制。记住是有可能，而不是一定。例如：假设存在两个线程（线程1、线程2），线程1通过Iterator在遍历集合A中的元素，在某个时候线程2修改了集合A的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出 ConcurrentModificationException 异常，从而产生fail-fast机制.

**Fail-fast产生的原因:** 程序在对 collection 进行迭代时，某个线程对该 collection 在结构上对其做了修改，这时迭代器就会抛出 ConcurrentModificationException 异常信息，从而产生 fail-fast. 迭代器在调用next()、remove()方法时都是调用checkForComodification()方法，该方法主要就是检测modCount == expectedModCount ? 若不等则抛出ConcurrentModificationException 异常，从而产生fail-fast机制. 集合类中无论add、remove、clear方法只要是涉及了改变集合元素的个数的方法都会导致modCount的改变。所以由于expectedModCount 得值与modCount的改变不同步，导致两者之间不等从而产生fail-fast机制.

**例如:** 有两个线程（线程A，线程B），其中线程A负责遍历list、线程B修改list。线程A在遍历list过程的某个时候（此时expectedModCount = modCount=N），同时线程B增加一个元素，这是modCount的值发生改变（modCount + 1 = N + 1）。线程A继续遍历执行next方法时，通告checkForComodification方法发现expectedModCount  = N  ，而modCount = N + 1，两者不等，这时就抛出ConcurrentModificationException 异常，从而产生fail-fast机制.

**Fail-fast的解决办法:**

i). 在遍历过程中所有涉及到改变modCount值得地方全部加上synchronized或者直接使用Collections.synchronizedList，这样就可以解决。但是不推荐，因为增删造成的同步锁可能会阻塞遍历操作;

ii). fail-fast机制，是一种错误检测机制。**它只能被用来检测错误，因为JDK并不保证fail-fast机制一定会发生。**若在多线程环境下使用fail-fast机制的集合，建议使用“java.util.concurrent包下的类”去取代“java.util包下的类”.

**10).hashCode()与equals()**

i).相等（相同）的对象必须具有相等的哈希码（或者散列码）: 假如两个Java对象A和B，A和B相等（eqauls结果为true），但A和B的哈希码不同，则A和B存入HashMap时的哈希码计算得到的HashMap内部数组位置索引可能不同，那么A和B很有可能允许同时存入HashMap，显然相等/相同的元素是不允许同时存入HashMap，HashMap不允许存放重复元素;

ii). 两个对象的hashCode相同，它们并不一定相同: 不同对象的hashCode可能相同；假如两个Java对象A和B，A和B不相等（eqauls结果为false），但A和B的哈希码相等，将A和B都存入HashMap时会发生哈希冲突，也就是A和B存放在HashMap内部数组的位置索引相同这时HashMap会在该位置建立一个链接表，将A和B串起来放在该位置，显然，该情况不违反HashMap的使用原则，是允许的;

iii). equals()相等的两个对象，hashcode()一定相等；equals()不相等的两个对象，却并不能证明他们的hashcode()不相等.

**11).Iterater和ListIterator的区别**

Iterator的定义如下: public interface Iterator<E> {}

abstract boolean hasNext()

abstract E next()

abstract void remove()

**Iterator:**是一个接口，它是集合的迭代器。集合可以通过Iterator去遍历集合中的元素。Iterator提供的API接口，包括：是否存在下一个元素、获取下一个元素、删除当前元素; Iterator遍历Collection时，是fail-fast机制的。即，当某一个线程A通过iterator去遍历某集合的过程中，若该集合的内容被其他线程所改变了；那么线程A访问集合时，就会抛出ConcurrentModificationException异常，产生fail-fast事件.

ListIterator的定义如下: public interface ListIterator<E> extends Iterator<E> {}

// 继承于Iterator的接口

abstract boolean hasNext()

abstract E next()

abstract void remove()

// 新增API接口

abstract void add(E object)

abstract boolean hasPrevious()

abstract int nextIndex()

abstract E previous()

abstract int previousIndex()

abstract void set(E object)

**ListIterator:**是一个继承于Iterator的接口，它是队列迭代器。专门用于遍历List，能提供向前/向后遍历。相比于Iterator，它新增了添加、是否存在上一个元素、获取上一个元素等等API接口

**12). Collections.sort和Arrays.sort详解**

集合如何实现排序:

i).实现Comparable接口,例如要对List<A> list进行排序, 那么A要实现Comparable接口,且需要在A中重写compareTo方法, 然后调用Collections.sort(list)进行排序;

ii).使用Comparator比较器接口, List<A> list, Collections.sort(list, new Comparator(){重写compare方法});

**Arrays.sort:** 集合类的排序底层都是通过调用Arrays.sort实现的, JDK1.7之前用归并排序, 1.7改为TimSort.sort; 同时给用户提供了可修改的排序规则(设置java.util.Arrays.useLegacyMergeSort), 用户通过修改系统的属性改为归并排序(为了兼容就版本).

**TimSort:** Timsort是结合了归并排序（merge sort）和插入排序（insertion sort）而得出的排序算法，它在现实中有很好的效率.

例如：array[] = [24,63,70,55,41,92,81,80]，排序步骤如下：

i). 拆分分区：[24,63], [70,55], [41,92], [81,80]   
ii). 重排分区：[24,63], [55,70], [41,92], [80,81]   
iii). 合并分区：[24,63,55,70], [41,92,80,81]   
iv). 重排分区：[24,55,63,70], [41,80,81,92]   
v). 合并分区：[24,55,63,70,41,80,81,92]   
vi). 重排：[24,41,55,63,70,80,81,92]

Java对于Timsort的实现与上述原理有区别。Java首先会根据数组长度，采用Binarysort（折半插入排序法）对长度小于32（MIN\_MERGE）直接进行排序返回结果；However，对于长度大于等于32的数组，先分区，再对单个分区进行采用Binarysort排序，最后合并分区并排序.

**折半插入排序算法:** 二分法插入排序Binarysort要求首先找出数组（此数组即分区）中从0位开始连续升序区块，及区块下一位元素pivot；例如下面的例子:

i).[1,3,5,7,9,4,8] 的起始连续升序区块是 [1,3,5,7,9]，区块长度为5，即runLen；pivot是4(升区块的下一个元素)；

 ii).通过二分法比较pivot与区分元素的升降序关系，计算pivot在区块中的位置；并将pivot插入到该位置，组成新的区块；4在区块中[1,3,5,7,9]，先比较5 > 4，是降序；再比较3 < 4，是升序，确认位置，通过native方法System.arraycopy()插入到区块中, 插入后为[1,3,4,5,7,9];

iii). 再计算新区块与新pivot(下一个元素为8)的位置关系，直到完成排序.

**分区方法:** 假定数据长度为n，   
If n < MIN\_MERGE(32)，返回n   
Else if n 是2的倍数，返回MIN\_MERGE(32) / 2 = 16   
Else 返回整数k，k取值范围MIN\_MERGE(32) / 2 = 16 <= k <= MIN\_MERGE(32);

例：   
Array Length = 15 ; minRun = 15   
Array Length = 50 ; minRun = 25   
Array Length = 500 ; minRun = 32

iv).将上述得到的每个分区入栈(ComparableTimSort),当栈顶的分区长度不满足下列两个条件中的任意一个时,采用归并排序将其中长度最短的两个分区合并成一个分区,然后在用折半插入排序进行排序, 最终分区个数为1则排序完成:

① runLen[n-2] > runLen[n-1] + runLen[n]

② runLen[n-1] > runLen[n]

**13).Comparable和Comparator的区别**

i).包不同: Comparable在java.lang下，Comparator在java.util下;

ii).包含方法不同: Comparable只包含一个方法: int compareTo(T object); 而Comparator包含两个方法: int compare(T object1, T object2)和boolean equals(T obj);

iii). 都可以作用于集合中元素相互之间的比较，但是Comparable使用在集合元素对象内部(比如要比较List<A> list, 那么在类A中实现Comparable,然后重写compareTo方法)，Comparator使用在元素对象外部,就Collections.sort()而言，当对类T实现了Comparable接口的时候，sort()方法通过类T中定义的compareTo方法对元素进行比较。如果类T没有实现Comparable方法，可以在调用sort()方法之前生成一个比较器：Comparator，将比较器传递给sort，sort()方法会按照Comparator中定义的compare方法对元素进行比较(List<B> listB; Collections.sort(listB, new Comparator(){实现compare方法和equals方法})).

**14).如何确保集合对象在函数中不被修改**

可以通过调用Collections.unmodifiableCollection(Collection c)方法来创建一个只读的集合，然后再将它传入函数中，这样就会确保任何企图修改集合的操作都会引起 UnsupportedOperationException异常。

**15).集合的排序算法原理**

[见问题12](#_12)._Collections.sort和Arrays.sort详解)

**16).集合类使用的数据结构总结**

见问题1,2,3,4,5

**17).快速失败(fail-fast)和安全失败(fail-safe)的区别**

**快速失败（fail—fast）**

在用迭代器遍历一个集合对象时，如果遍历过程中集合对象中的内容发生了修改（增加、删除、修改），则会抛出ConcurrentModificationException。

**原理**：迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个modCount变量。集合在被遍历期间如果内容发生变化，就会改变modCount的值。每当迭代器使用hashNext()/next()遍历下一个元素之前，都会检测modCount变量是否为expectedmodCount值，是的话就返回遍历；否则抛出异常，终止遍历。

**注意**：这里异常的抛出条件是检测到 modCount！=expectedmodCount这个条件。如果集合发生变化时修改modCount值刚好又设置为了expectedmodCount值，则异常不会抛出。因此，不能依赖于这个异常是否抛出而进行并发操作的编程，这个异常只建议用于检测并发修改的bug。

**场景**：java.util包下的集合类都是快速失败的，不能在多线程下发生并发修改（迭代过程中被修改）。

**安全失败（fail—safe）**

采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。

**原理**：由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发ConcurrentModificationException。

**缺点**：基于拷贝内容的优点是避免了ConcurrentModificationException，但同样地，迭代器并不能访问到修改后的内容，即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。

**场景**：java.util.concurrent包下的容器都是安全失败，可以在多线程下并发使用，并发修改。

**18).优先级队列(Priority Queue)**

**PriorityQueue类**: 在Java1.5中引入并作为 [Java Collections Framework](http://www.journaldev.com/1260/java-collections-framework-tutorial) 的一部分。PriorityQueue是基于优先堆(小顶堆)的一个无界队列，这个优先队列中的元素可以默认自然排序或者通过提供的[Comparator](http://www.journaldev.com/780/java-comparable-and-comparator-example-to-sort-objects)（比较器）在队列实例化的时排序.

i). 优先队列不允许空值，而且不支持non-comparable（不可比较）的对象，比如用户自定义的类。优先队列要求使用[Java Comparable和Comparator接口](http://www.journaldev.com/780/java-comparable-and-comparator-example-to-sort-objects)给对象排序，并且在排序时会按照优先级处理其中的元素;

ii). 优先队列的头是基于自然排序或者[Comparator](http://www.journaldev.com/780/java-comparable-and-comparator-example-to-sort-objects)排序的最小元素。如果有多个对象拥有同样的排序，那么就可能随机地取其中任意一个。当我们获取队列时，返回队列的头对象;

iii). 优先队列的大小是不受限制的，但在创建时可以指定初始大小。当我们向优先队列增加元素的时候，队列大小会自动增加;

iv). PriorityQueue是非线程安全的，所以Java提供了PriorityBlockingQueue（实现[BlockingQueue接口](http://www.journaldev.com/1034/java-blockingqueue-example-implementing-producer-consumer-problem)）用于[Java多线程环境](http://www.journaldev.com/1079/java-thread-tutorial);

Queue<Integer> integerPriorityQueue = new PriorityQueue<>(初始大小);//使用自然排序

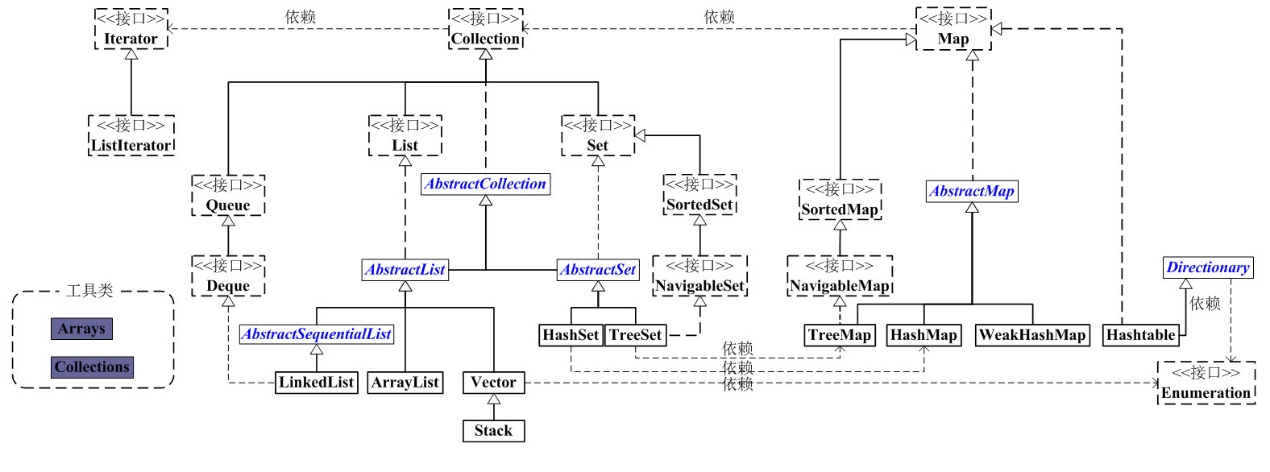
Queue<Customer> customerPriorityQueue = new PriorityQueue<>(初始大小, new Comparator(){});

**19).为什么集合没有继承Cloneable和Serializable接口**

集合接口没有继承Cloneable和Serializable接口,但是在集合的具体实现类中进行了实现,因为每个具体的集合类的实现方式不同,Serializable需要自定义区实现,见问题21.

**20).集合类的体系结构**

Java集合是java提供的工具包，包含了常用的数据结构：集合、链表、队列、栈、数组、映射等。Java集合工具包位置是java.util.\*  
Java集合主要可以划分为4个部分：List列表、Set集合、Map映射、工具类(Iterator迭代器、Enumeration枚举类、Arrays和Collections).



Collection是一个接口，是高度抽象出来的集合，它包含了集合的基本操作和属性, **Collection包含了List和Set两大分支:**

i).  List是一个有序的队列，每一个元素都有它的索引。第一个元素的索引值是0, List的实现类有LinkedList, ArrayList, Vector, Stack。

ii). Set是一个不允许有重复元素的集合, Set的实现类有HastSet和TreeSet。HashSet依赖于HashMap，它实际上是通过HashMap实现的；TreeSet依赖于TreeMap，它实际上是通过TreeMap实现的.

**Map是一个映射接口，即key-value键值对。Map中的每一个元素包含“一个key”和“key对应的value”:**

i). AbstractMap是个抽象类，它实现了Map接口中的大部分API。而HashMap，TreeMap，WeakHashMap都是继承于AbstractMap.

ii). Hashtable虽然继承于Dictionary，但它实现了Map接口.

**Iterator:** 它是遍历集合的工具，即我们通常通过Iterator迭代器来遍历集合。我们说Collection依赖于Iterator，是因为Collection的实现类都要实现iterator()函数，返回一个Iterator对象; ListIterator是专门为遍历List而存在的.

**Enumeration:**它是JDK 1.0引入的抽象类;作用和Iterator一样，也是遍历集合；但是Enumeration的功能要比Iterator少。在上面的框图中，Enumeration只能在Hashtable, Vector, Stack中使用.

**Arrays和Collections:** 它们是操作数组、集合的两个工具类

**21).集合类与序列化**

**序列化的步骤:**

i)写入:首先创建一个OutputStream输出流;然后创建一个ObjectOutputStream输出流，并传入OutputStream输出流对象; 最后调用ObjectOutputStream对象的writeObject()方法将对象状态信息写入OutputStream;

ii).读取: 首先创建一个InputStream输入流; 然后创建一个ObjectInputStream输入流，并传入InputStream输入流对象; 最后调用ObjectInputStream对象的readObject()方法从InputStream中读取对象状态信息;

**ArrayList序列化:** 在ArrayList源码中transient [Object](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8u40-b25/java/lang/Object.java#Object%5B%5D)[] elementData即elementData是禁止序列化的, 但是ArrayList又实现了Serializable接口,从序列化的步骤中可以看出, 要想序列化对象，使用ObjectOutputStream对象输出流的writeObject()方法写入对象状态信息，即可使用readObject()方法读取信息.

在ArrayList的源码中重写了readObject和writeObject方法,其中writeObject方法中虽然elementData被transient修饰，不能被序列化，但是我们可以将它的值取出来，然后将该值写入输出流:

private void writeObject(java.io.ObjectOutputStream s) throws java.io.IOException{  
    // Write out element count, and any hidden stuff  
    int expectedModCount = modCount;  
    s.defaultWriteObject();  
    // Write out size as capacity for behavioural compatibility with clone()  
    s.writeInt(size);  
    // Write out all elements in the proper order.  
    for (int i = 0; i < size; i++)// 循环时是使用i<size而不是 i<elementData.length，说明序列化时，只需实际存储的那些元素，而不是整个数组  
    {  
        s.writeObject(elementData[i]);// 传值时，是将实参elementData[i]赋给s.writeObject()的形参  
    }  
    if (modCount != expectedModCount)  
    {  
        throw new ConcurrentModificationException();  
    }  
}

反序列化同样:

// java.io.Serializable的读取函数：根据写入方式读出

// 先将ArrayList的“容量”读出，然后将“所有的元素值”读出

private void readObject(java.io.ObjectInputStream s)

throws java.io.IOException, ClassNotFoundException {

// Read in size, and any hidden stuff

s.defaultReadObject();

// 从输入流中读取ArrayList的“容量”

int arrayLength = s.readInt();

Object[] a = elementData = new Object[arrayLength];

// 从输入流中将“所有的元素值”读出

for (int i=0; i<size; i++)

a[i] = s.readObject();

}

}

可以发现ArrayList的readObject和writeObject方法都是private的,那么**何时被调用**: ObjectOutputStream会调用这个类的writeObject方法进行序列化，ObjectInputStream会调用相应的readObject方法进行反序列化; 那么ObjectOutputStream又是如何知道一个类是否实现了writeObject方法呢？又是如何自动调用该类的writeObject方法: 通过反射实现, 例如ObjectOutputStream会根据传进来的ArrayList对象得到Class,然后在包装成ObjectStreamClass, 在writeSerialData方法里，会调用ObjectStreamClass的 invokeWriteObject方法.

**既然ArrayList需要被序列化为什么又要加transient关键字:**

ArrayList的自动扩容机制，elementData数组相当于容器，当容器不足时就会再扩充容量，但是容器的容量往往都是大于或者等于ArrayList所存元素的个数, 比如，现在实际有了8个元素，那么elementData数组的容量可能是8x1.5=12, 如果直接序列化elementData数组，那么就会浪费4个元素的空间，特别是当元素个数非常多时，这种浪费是非常不合算的,所以ArrayList的设计者将elementData设计为transient，然后在writeObject方法中手动将其序列化，并且只序列化了实际存储的那些元素，而不是整个数组,在源码中体现为:

for (int i = 0; i < size; i++)// 循环时是使用i<size而不是 i<elementData.length，说明序列化时，只需实际存储的那些元素，而不是整个数组

{  
        s.writeObject(elementData[i]);// 传值时，是将实参elementData[i]赋给s.writeObject()的形参

**LinkedList序列化:** LinkedList中size, Node first, Node last字段均是transient修饰的,但是重写了writeObject和readObject对size进行了序列化,同时对LinkedList中非null的元素进行序列化.

private void writeObject(java.io.ObjectOutputStream s)  
        throws java.io.IOException {  
        // Write out any hidden serialization magic  
        s.defaultWriteObject();  
        // Write out size  
        s.writeInt(size);  
        // Write out all elements in the proper order.  
        for (Node<E> x = first; x != null; x = x.next)  
            s.writeObject(x.item);  
    }  
    private void readObject(java.io.ObjectInputStream s)  
        throws java.io.IOException, ClassNotFoundException {  
        // Read in any hidden serialization magic  
        s.defaultReadObject();  
        // Read in size  
        int size = s.readInt();  
        // Read in all elements in the proper order.  
        for (int i = 0; i < size; i++)  
            linkLast((E)s.readObject());  
    }

**HashSet序列化:** 自定义实现, 对HashSet的总容量, 实际容量,加载因子,所有元素进行序列化.

**TreeSet序列化:** 自定义实现, 将TreeSet的比比较器Comparator,容量,所有元素值进行序列化和反序列化.

**HashMap的序列化:** 自定义实现, 重写writeObject, readObject方法,对HashMap的buckets,size,进行序列化.在对Node节点进行序列化时, 需要去除null的Node然后将Node的key,value分别序列化.

private void writeObject(java.io.ObjectOutputStream s)  
        throws IOException {  
        int buckets = capacity();  
        // Write out the threshold, loadfactor, and any hidden stuff  
        s.defaultWriteObject();  
        s.writeInt(buckets);  
        s.writeInt(size);  
        internalWriteEntries(s);  
}

void internalWriteEntries(java.io.ObjectOutputStream s) throws IOException {  
        Node<K,V>[] tab;  
        if (size > 0 && (tab = table) != null) {  
            for (int i = 0; i < tab.length; ++i) {  
                for (Node<K,V> e = tab[i]; e != null; e = e.next){  
                    s.writeObject(e.key);  
                    s.writeObject(e.value);  
                }  
            }  
        }  
}

**HashTable序列化**: 自定义实现, 对HashTable的table.length, 总数count以及非null的元素序列化.

**ConcurrentHashMap序列化:**自定义实现,对Segment数组的容量,segmentShift,segmentMask,以及ConcurrentHashMap中key,value进行序列化.

字符串：

**1).String,StringBuffer,StringBuilder,StringTokenizer的区别**

i)String为字符串常量,而StringBuffer和StringBuilder均为字符串变量,即String对象一旦创建后就不能修改,但后两者均是可以修改的. String str = “abc”; str = str +”de”; 其实JVM又创建了一个新的对象str,然后在把原来的str的值和”de”加起来再赋值给新的str,所以String对象进行操作的实际上是一个不断创建新的对象并且将旧的对象回收的一个过程,所以速度很慢.

ii)StringBuffer和StringBuilder是直接对变量进行操作修改对象进行更改,而不进行创建和回收操作.

iii)线程安全上, StringBuilder是线程不安全的,而StringBuffer是线程安全的,如果一个StringBuffer对象在字符串缓冲区被多个线程使用时,StringBuffer中很多方法是带有synchronized关键字的,所以可以保证线程安全性,但是StringBuilder的方法不能保证线程安全, 所以如果是多线程操作,一般是由StringBuffer,在单线程中使用StringBuilder效率更高.

iv)使用场景:String适合少了的字符串的操作,StringBuilder使用与单线程下在字符串缓冲区进行大量操作的情况,StringBuffer使用与多线程下在字符串缓冲区进行大量的操作.

v).StringTokenizer: 用来分割字符串的工具类.

**2).String的特征**

i).不变模式:一个字符串对象创建后它的值不能改变, String类用final修饰,String不能被继承,并且它的成员方法都是默认的final的, 即无论sub,concat,replace操作都不是在原来的字符串上进行的,而是重新生成了一个新的字符串对象, String对象一旦被创建后就固定不变了,对String对象的任何改变都不影响到原对象,任何相关的change操作都会生成新的对象;

ii).字符串常量池:每当创建一个字符串常量时,JVM会首先检查字符串常量池,如果该字符串已经存在常量池中,那么就直接返回常量池的引用实例,如果字符串不存在常量池中,就会实例化该字符串并且将其放到常量池中,由于String字符串的不可变性,则在常量池中一定不存在两个相同的字符串.(常量池: 静态常量池是指\*.class文件中的常量池,class文件中的常量池不仅仅包含字符串字面量,还包含类,方法的信息,占用class文件绝大部分空间; 运行时常量池是JVM在完成类装载后,将class文件汇总的常量池载入到内存汇总,并保存在方法区中,常说的常量池就是方法区中的运行常量池).

iii).String初始化后为不可变的,因为String使用private final char value[]来实现字符串的存储,String创建后就不能修改对象中存储的字符串内容.

**3).String s = new String(“abc”)创建了几个对象**

一个或者两个, 如果常量池中原来有”abc”字符串,那么只创建一个”abc”对象,如果常量池中原来没有字符串”abc”,那么会在一个全局共享的字符串常量池中创建一个与字符串”abc”字面量对应的字符串,以及通过new String(String)在堆中创建的对象.

**String s = new String(“abc”);或者String s=null; 涉及用户声明的几个String类型的变量?**

一个, 就是String s. Java中变量就是变量, 引用类型的变量只是对某个对象实例或者null的引用,不是实例本身, 声明变量的个数根创建实例的个数没有必然关系, 例如下面语句:

String s1 = “a”;

String s2 = s1.concat(“”);

String s3 = null;

new String(s1);

这段代码会涉及到三个String的变量, s1, s2, s3; 同时涉及到3个String实例, i)“a”字面常量对应的驻留的字符串常量String实例;

ii)“”对应的String实例(String.concat()当发现传入的参数是空字符串会返回this,并不会创建新的String实例);

iii)通过new String(String)创建的新的String的实例,没有任何变量指向它.

**4).String的intern方法**

运行时常量池相当于Class文件常量池具有动态性，Java语言并不要求常量一定只有编译期间才能产生，运行期间也可以将新的常量放入池中，String类的intern()方法就是这样的.

//String s = new String("1"):生成了两个对象, 一个是堆中的"1"对象,另外一个常量池中的对象;  
//s.intern创建常量池对象,发现常量池中已经有"1"了,就不在创建;  
 //s2发现常量池中有"1",直接指向"1";  
//由于s是堆中对象, s2是常量池中对象,所以为false  
//JDK6:false,因为s为堆中对象,而s2位常量池对象,二者地址不同  
// String s = new String("1");  
// s.intern();  
// String s2 = "1";  
// System.out.println(s == s2);  
// //JDK1.7以上, 常量池移入Java堆中, s3创建对象后, 首先在常量池中创建"1", 并且创建"11"对象,s3指向"11"的引用,此时常量池中没有"11";  
// //s3.intern()后, 由于JDK7以后常量池移入堆中,此时因为堆中存有"11",所以此时常量池"11"直接指向s3的地址引用;  
// //s4指向常量池中的"11",所以此时s3和s4都指向同一个引用.  
 //JDK6:false;因为s3位堆中对象,而s4为常量池中对象, JDK6以前常量池存放在方法区中的永久区(Perm 区)中, 所以二者指向不同的地址  
// String s3 = new String("1") + new String("1");  
// s3.intern();  
// String s4 = "11";  
// System.out.println(s3 == s4);  
//JDK1.7以上,String s = new String("1"):此处创建两个对象, 一个堆里的对象,s指向; 另外一个在常量池中的对象;  
 //s2="1"指向常量池中的"1";  
 //s.intern()把s指向常量池,此时由于new String("1")已经在常量池中创建对象"1",所以此条语句没有实际作用;  
//最终s指向堆中的"1",而s2指向常量池,因此是false  
 String s = new String("1");  
 String s2 = "1";  
 s.intern();  
 System.out.println(s == s2);  
 //String s3 = new String("1") + new String("1");在常量池中创建"1",在堆中创建"11"对象;  
 //String s4 = "11"在常量池中创建;  
 //因为此时常量池中已经有"11"了,因此s3.intern不起作用  
 //s3位堆中对象, s4指向常量池, 所以为false  
 String s3 = new String("1") + new String("1");  
 String s4 = "11";  
 s3.intern();  
 System.out.println(s3 == s4);

**5).字符串比较“==”和equals**

i)基本数据类型，也称原始数据类型byte,short,char,int,long,float,double,boolean 他们之间的比较，应用双等号（==）,比较的是他们的值。

ii)复合数据类型(类) 当他们用（==）进行比较的时候，比较的是他们在内存中的存放地址，所以，除非是同一个new出来的对象，他们的比较后的结果为true，否则比较后结果为false。 JAVA当中所有的类都是继承于Object这个基类的，在Object中的基类中定义了一个equals的方法，这个方法的初始行为是比较对象的内存地 址，但在一些类库当中这个方法被覆盖掉了，如String,Integer,Date在这些类equals有其自身的实现，而不再是比较类在堆内存中的存放地址了。对于复合数据类型之间进行equals比较，在没有覆写equals方法的情况下，他们之间的比较还是基于他们在内存中的存放位置的地址值的，因为Object的equals方法也是用双等号（==）进行比较的，所以比较后的结果跟双等号（==）的结果相同。

如果两个对象x和y满足x.equals(y) == true，它们的哈希码（hash code）应当相同。Java对于eqauls方法和hashCode方法是这样规定的：(1)如果两个对象相同（equals方法返回true），那么它们的hashCode值一定要相同；(2)如果两个对象的hashCode相同，它们并不一定相同。

**6).为什么存储密码要用char[]而不用String**

String在Java中是不可变对象，这就意味着一旦创建了一个字符串，如果另一个进程把尝试内存的数据导出（dump），在GC进行垃圾回收之前该字符串会一直保留在内存中，那么该进程就可以轻易的读取到该字符串。而对于数组，可以在该数组处理它之后再显示地擦掉数组中内容，可以以任何方式把数组内容覆盖掉，即使是GC触发前密码也不会出现在系统的任何地方。引用官方的文档，Java加密体系结构指南说到char[] vs String。在对象java.lang.String中收集和存储密码看似符合逻辑，但是字符串对象是不可变的，没有任何方法可以改变（重写）或清空内容。这一特性使得字符串对象不适合存储安全敏感信息，比如用户密码。你应当使用一个字符数组来代替，以便收集和存储安全敏感的信息。安全编码指南4.0也提到了类似的问题.

**7).String在JDK7以后可以用于switch中**

Java 7 中加入了对String类型的支持。所以支持的有：char、byte、short、int 和 Character、Byte、Short、Integer 和 String

这个新特性是在编译器这个层次上实现的。而在Java虚拟机和字节码这个层次上还是只支持在switch语句中使用与整数类型兼容的类型。这么做的目的就是为了减少这个特性所影响的范围，以降低实现的代价。在编译器层次实现的含义是，虽然开发人员在Java源代码的switch语句中使用了字符串类型，但是在编译的过程中，编译器会根据源代码的含义进行转换，将字符串类型转换成与整数类型兼容的格式。不同的Java编译器可能采用不同的方式来转换，并采用不同的优化策略。比如：如果switch语句中只包含一个case语句，那么就可以简单的将其转换成一个if语句。如果包含一个case和一个default语句，就可以转换成if-else语句。而对于复杂的情况（多个case语句），也可以转换成Java 7 之前的switch语句，只不过使用字符串的哈希值作为switch语句表达式的值。经过转换，Java 虚拟机看到的仍然是与整数类型兼容的类型。这里要注意的是，在case字句中对应的语句块中仍然需要使用String的equals方法来进行字符串比较，这是因为哈希函数在映射的时候可能存在冲突，这样更加保险了

**8).subString()方法的原理**

[见面向对象中的问题8](#_8).JDK6和JDK7中substring的区别以及原理)

多线程：

**1)Thread,Runnable,Callable区别及用法**

Thread:通过继承Thread类来创建线程,重写run()方法;

Runnable:通过实现Runnable接口创建线程,实现run()方法;

Callable: Callable并不像Runnable那样通过Thread的start方法就能启动实现类的run方法，所以它通常利用ExecutorService的**submit**方法去启动call方法自执行任务，而ExecutorService的submit又返回一个**Future**类型的结果，因此Callable通常也与Future一起使用.

**Thread与Runnable的区别:**

在程序开发中只要是多线程，肯定永远以实现Runnable接口为主，因为实现Runnable接口相比继承Thread类有如下优势:

i)可以避免由于Java的单继承特性而带来的局限；

ii)增强程序的健壮性，代码能够被多个线程共享，代码与数据是独立的；

iii)适合多个相同程序代码的线程区处理同一资源的情况。

iv).Thread在new一个线程后, 多个线程独立执行自己的任务, Runnable在创建多个线程时,多个线程完成一个任务,对于一些特殊的任务需要加锁保证可行性.

**Runnable与Callable的区别:**

i)Callable规定的方法是call(),Runnable规定的方法是run()

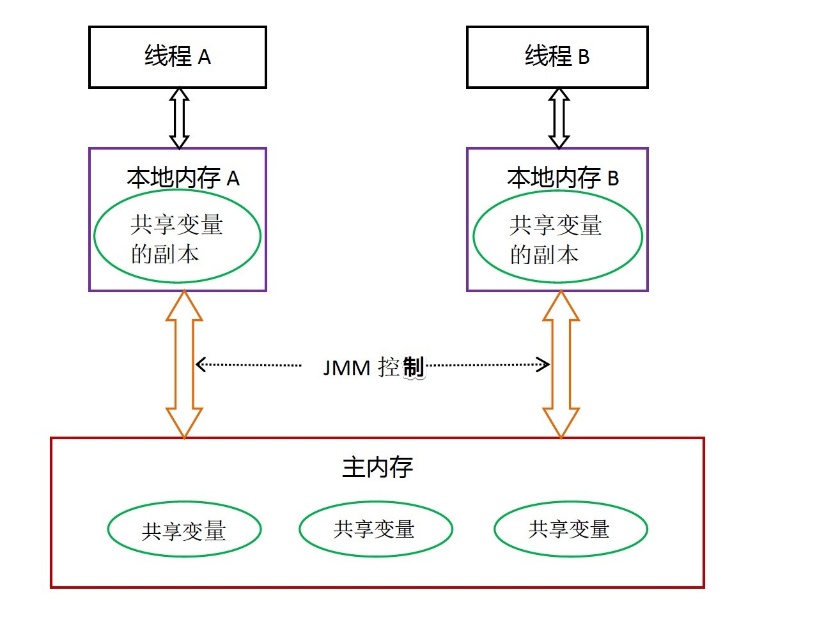
ii)Callable的任务执行后可返回值，而Runnable的任务是不能返回值得

iii)call方法可以抛出异常，run方法不可以

iv)运行Callable任务可以拿到一个Future对象，Future 表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并获取计算的结果。计算完成后只能使用 get 方法来获取结果，如果线程没有执行完，Future.get()方法可能会阻塞当前线程的执行；如果线程出现异常，Future.get()会throws InterruptedException或者ExecutionException；如果线程已经取消，会跑出CancellationException。取消由cancel 方法来执行。isDone确定任务是正常完成还是被取消了。一旦计算完成，就不能再取消计算。如果为了可取消性而使用 Future 但又不提供可用的结果，则可以声明Future<?> 形式类型、并返回 null 作为底层任务的结果.

**2).Java内存模型(内存可见性, 重排序, 顺序一致性, volatile, final)**

**Java内存模型(JMM):**用来屏蔽各种硬件和操作系统的内存访问差异,从而让Java程序在各种平台下都能达到一致的内存访问效果.



如果线程 A 和线程 B 要通信的话，要如下两个步骤: i)线程 A 需要将本地内存 A 中的共享变量副本刷新到主内存去; ii)线程 B 去主内存读取线程 A 之前已更新过的共享变量. (注:共享变量存储在主内存中,共享变量副本存储在工作内存中)

**内存可见性:**线程A对共享变量的修改要想被线程B看见,首先必须把线程A中的本地内存中更新过的变量刷新到主存中; 然后主存将最新的共享变量更新到线程B的本地内存中.

**内存可见的方式:**

**i)synchronized实现可见性:**通过实现互斥锁(同步),需要遵循:线程解锁前必须把共享变量刷新到主存中去,线程加锁时,需要清空工作内存(本地内存中共享变量的副本)中共享变量的值, 需要的时候从主存中去取新的值; 线程解锁前对共享变量的修改在下次加锁时对其他线程可见.

**ii)volatile实现可见性:** volatile能够保证变量的可见性,但不能保证volatile变量符合操作的原子性(例如i=0, 执行i++操作, 线程1和线程2都执行i++操作,如果线程1执行完i+1后把i=2刷新到主存,此时主存中i=2, 但是线程2已结执行完了i+1操作,线程2从主存中获取的i最新值也为2,但是自己已经执行完i+1操作,此时i也为2,此时刷新到主存最终i为2,而不是3, 所以无法保证原子性),主要通过添加内存屏障和禁止重排序(重排序时不能把后面的指令重排序到内存屏障之前的位置)来保证可见性:

对volatile变量执行写操作时,会在写入后加入一条store屏障指令(禁止重排序)

对volatile变量执行读操作时,会在读操作前加入一条load屏障指令(禁止重排序)

即, volatile变量在每次被线程访问时,都强迫从主存中读取该变量的值,当变量发生改变时,又会强迫线程将最新的值刷新到主存,从而是线程每次都能看到volatile变量的最新值. 但是volatile无法保证复杂操作的原子性,例如++i操作.(需要通过synchronized和ReentrantLock来保证).

**iii)synchronized与volatile的比较:** volatile不需要加锁,比synchronized更轻量级,不会阻塞线程,效率更高; 从内存可见性的角度来看, volatile读相当于加锁, 写相当于解锁; synchronized技能保证可见性,又能保证原子性,而volatile只能保证可见性,不能保证原子性.

**重排序:** 如果两个操作访问同一个变量，且这两个操作中有一个为写操作，此时这两个操作之间就存在数据依赖性。数据依赖分下列三种类型. 只要重排序两个操作的执行顺序，程序的执行结果将会被改变.

|  |  |  |
| --- | --- | --- |
| 写后读 | a = 1;b = a; | 写一个变量之后，再读这个位置。 |
| 写后写 | a = 1;a = 2; | 写一个变量之后，再写这个变量。 |
| 读后写 | a = b;b = 1; | 读一个变量之后，再写这个变量 |

编译器和处理器在重排序时，会遵守数据依赖性，编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序; 这里所说的数据依赖性仅针对单个处理器中执行的指令序列和单个线程中执行的操作，不同处理器之间和不同线程之间的数据依赖性不被编译器和处理器考虑.

**as-if-serial语义:** as-if-serial语义的意思指：不管怎么重排序（编译器和处理器为了提高并行度），（单线程）程序的执行结果不能被改变。编译器，runtime 和处理器都必须遵守as-if-serial语义; 为了遵守as-if-serial语义，编译器和处理器不会对存在数据依赖关系的操作做重排序，因为这种重排序会改变执行结果。但是，如果操作之间不存在数据依赖关系，这些操作可能被编译器和处理器重排序。

**重排序对多线程的影响:**

class Demo {  
 int a = 0;  
 boolean flag = false;  
 public void write() {  
 a = 1; //1  
 flag = true; //2  
 }  
 public void read() {  
 if(flag) { //3  
 int i = a \* a; //4  
 }  
 }  
}

操作 1 和 2 没有数据依赖关系，编译器和处理器可以对这两个操作重排序；操作 3 和操作 4 没有数据依赖关系，编译器和处理器也可以对这两个操作重排序;如果操作 1 和操作 2 做了重排序。程序执行时，线程 A 首先写标记变量 flag，随后线程 B 读这个变量。由于条件判断为真，线程 B 将读取变量 a。此时，变量 a 还根本没有被线程 A 写入，在这里多线程程序的语义被重排序破坏了.

**顺序一致性:** 一个线程中的所有操作必须按照程序的顺序来执行; （不管程序是否同步）所有线程都只能看到一个单一的操作执行顺序。在顺序一致性内存模型中，每个操作都必须原子执行且立刻对所有线程可见. 但是在JMM中无法保证顺序一致性. 未同步程序在 JMM 中不但整体的执行顺序是无序的，而且所有线程看到的操作执行顺序也可能不一致。比如，在当前线程把写过的数据缓存在本地内存中，在还没有刷新到主内存之前，这个写操作仅对当前线程可见；从其他线程的角度来观察，会认为这个写操作根本还没有被当前线程执行。只有当前线程把本地内存中写过的数据刷新到主内存之后，这个写操作才能对其他线程可见。在这种情况下，当前线程和其它线程看到的操作执行顺序将不一致

**需要借助同步锁来保证顺序一致性:**

class demo {  
 int a = 0;  
 boolean flag = false;  
 public synchronized void write() { //获取锁  
 a = 1;  
 flag = true;  
 } //释放锁  
 public synchronized void read() { //获取锁  
 if(flag) {  
 int i = a;  
 }  
 } //释放锁  
}

假设 A 线程执行 write() 方法后，B 线程执行 reade() 方法。这是一个正确同步的多线程程序。根据JMM规范，该程序的执行结果将与该程序在顺序一致性模型中的执行结果相同. 在顺序一致性模型中，所有操作完全按程序的顺序执行。而在 JMM 中，临界区内的代码可以重排序（但 JMM 不允许临界区内的代码“逸出”到临界区之外，那样会破坏监视器的语义）。JMM 会在退出临界区和进入临界区这两个关键时间点做一些特别处理，使得线程在这两个时间点具有与顺序一致性模型相同的内存视图。虽然线程 A 在临界区内做了重排序，但由于监视器的互斥执行的特性，这里的线程 B 根本无法“观察”到线程 A 在临界区内的重排序。这种重排序既提高了执行效率，又没有改变程序的执行结果。

**顺序一致性与JMM:** 未同步程序在 JMM 中的执行时，整体上是无序的，其执行结果无法预知。未同步程序在两个模型中的执行特性有下面几个差异:

i)顺序一致性模型保证单线程内的操作会按程序的顺序执行，而 JMM 不保证单线程内的操作会按程序的顺序执行（比如上面正确同步的多线程程序在临界区内的重排序）。

ii)顺序一致性模型保证所有线程只能看到一致的操作执行顺序，而 JMM 不保证所有线程能看到一致的操作执行顺序。

iii)JMM 不保证对 64 位的 long 型和 double 型变量的读/写操作具有原子性，而顺序一致性模型保证对所有的内存读/写操作都具有原子 。

**final域重排序规则:** 在引用变量为任意线程可见之前，该引用变量指向的对象的 final 域已经在构造函数中被正确初始化过了.(这其中要保证final域不能逸出)

**i)final 写**：“构造函数内对一个final域的写入”，与“随后把这个被构造对象的引用赋值给一个引用变量”，这两个操作之间不能重排序.( **编译器会在 final 域的写之后，构造函数 return 之前，插入一个 StoreStore 屏障**)

**2)final 读**：“初次读一个包含final域的对象的引用”，与“随后初次读对象的final域”，这两个操作之间不能重排序.(**编译器会在读 final 域操作的前面插入一个 LoadLoad 屏障**).

**原子性:**由Java内存模型来直接保证的原子性变量操作包括read,load,assign,use,store和write.基本数据类型的访问和读写是具有原子性的.(对于long和double类型变量的特殊规则:允许虚拟机实现选择可以不保证64为数据类型的load,store,read和write这4个操作的原子性,这就是long和double的非原子性协定.)

**3).volatile, ThreadLocal,synchronized,Automic的区别**

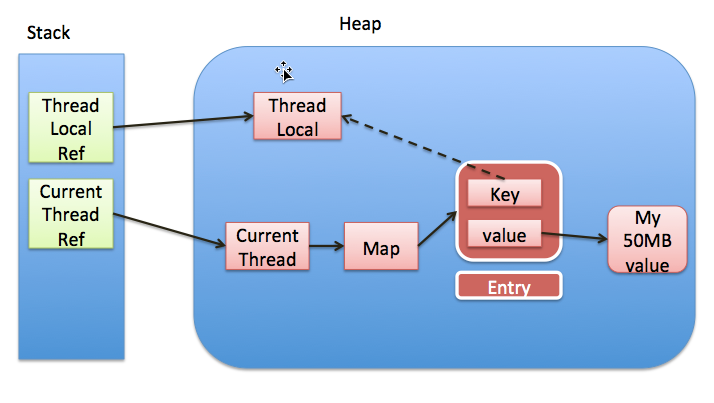
**volatile:** 主要是用来在多线程中同步变量

在一般情况下，为了提升性能，每个线程在运行时都会将主内存中的变量保存一份在自己的工作内存中作为变量副本，但是这样就很容易出现多个线程中保存的副本变量不一致，或与主内存的中的变量值不一致的情况。  
而当一个变量被volatile修饰后，该变量就不能被缓存到线程的工作内存中，它会告诉编译器不要进行任何移出读取和写入操作的优化，换句话说就是不允许有不同于“主”内存区域的变量拷贝，所以当该变量有变化时，所有调用该变量的线程都会获得相同的值，这就确保了该变量在应用中的可见性(当一个任务做出了修改在应用中必须是可见的)，同时性能也相应的降低了(还是比synchronized高)。  
但需要注意volatile只能确保操作的是同一块内存，并不能保证操作的原子性。所以volatile一般用于声明简单类型变量，使得这些变量具有原子性，即一些简单的赋值与返回操作将被确保不中断。但是当该变量的值由自身的上一个决定时，volatile的作用就将失效，这是由volatile关键字的性质所决定的。  
所以在volatile时一定要谨慎，千万不要以为用volatile修饰后该变量的所有操作都是原子操作，不再需要synchronized关键字了。

**为什么使用volatile:** Volatile变量修饰符如果使用恰当的话，它比synchronized的使用和执行成本会更低，因为它不会引起线程上下文的切换和调度.

**ThreadLocal:** 这个类的出现并不是用来解决在多线程并发环境下资源的共享问题的，它和volatile,synchronized,Automic关键字不一样，其它三个关键字都是从线程外来保证变量的一致性，这样使得多个线程访问的变量具有一致性，可以更好的体现出资源的共享. 而ThreadLocal的设计，**并不是解决资源共享的问题**，而是用来提供线程内的局部变量，这样每个线程都自己管理自己的局部变量，别的线程操作的数据不会对我产生影响，互不影响，所以不存在解决资源共享这么一说, 而ThreadLocal则是自己管理自己的，相当于封装在Thread内部了，供线程自己管理.

i).ThreadLocal的用法: 一般使用ThreadLocal，官方建议我们定义为private static ，至于为什么要定义成静态的，这和内存泄露有关, 原因如下:



每个Thread 维护一个 ThreadLocalMap 映射表，这个映射表的 key 是 ThreadLocal 实例本身，value 是真正需要存储的 Object,

例如: 在类A中定义private static ThreadLocal<String> threadLocal = new ThreadLocal<String>(), 则会在每一个线程中创建一个ThreadLocal.ThreadLocalMap的成员变量threadLocals,用例存储实际变量的副本,其中key为threadLocal, 值为当前线程需要存储的value.

初始时,在Thread里面,threadLocals为空,当通过ThreadLocal变量调用get()或者set()方法,就会对Thread类中的threadLocals进行初始化,并且以当前的ThreadLocal变量为键值,以要保存的副本变量为值,存到threadLocals中, 然后在线程Thread里面如果要使用变量副本,就可以通过get方法在threadLocals这个Map里面查找.

**ThreadLocal源码分析:**

**ThreadLocal类提供的几个方法如下:** get()方法是用来获取ThreadLocal在当前线程中保存的变量副本，set()用来设置当前线程中变量的副本，remove()用来移除当前线程中变量的副本，initialValue()是一个protected方法，一般是用来在使用时进行重写的，它是一个延迟加载方法.

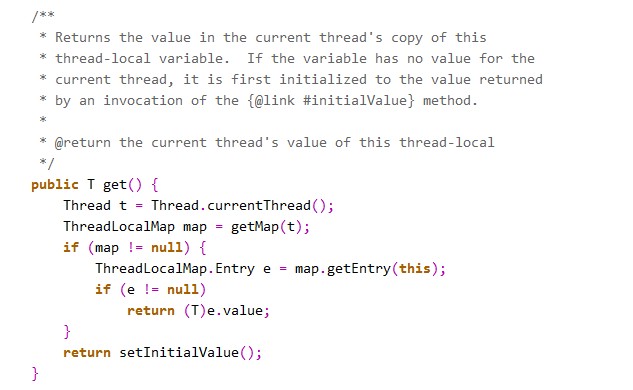
public T get() { }

public void set(T value) { }

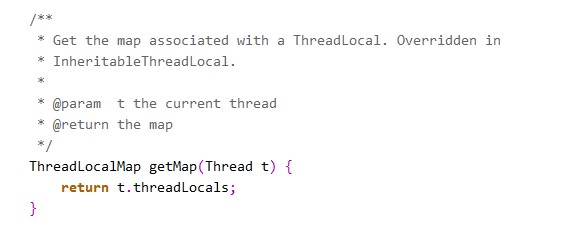
public void remove() { }

protected T initialValue() { }

a)get方法的实现: 第一句是取得当前线程，然后通过getMap(t)方法获取到一个map，map的类型为ThreadLocalMap。然后接着下面获取到<key,value>键值对，注意这里获取键值对传进去的是  this(ThreadLocal对象)，而不是当前线程t; 如果获取成功，则返回value值; 如果map为空，则调用setInitialValue方法返回value.

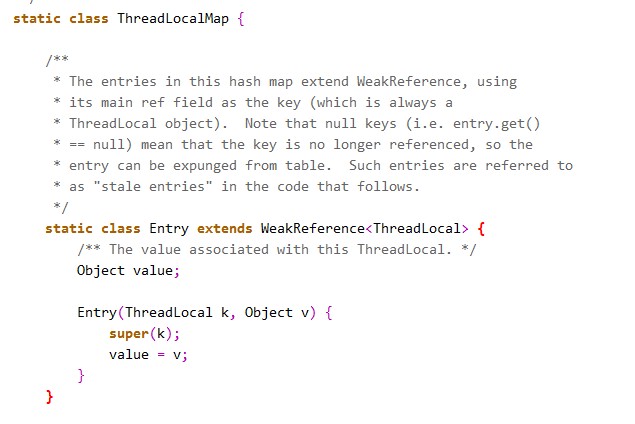


b).getMap: 调用当期线程t，返回当前线程t中的一个成员变量threadLocals, threadLocals是一个ThreadLocalMap对象.ThreadLocal.ThreadLocalMap threadLocals = null;



**c).ThreadLocalMap:**是ThreadLocal类的一个内部类, ThreadLocalMap的Entry继承了WeakReference，并且使用ThreadLocal作为键值.

**ThreadLocal 为每个线程创建变量副本总结:** 首先，在每个线程Thread内部有一个ThreadLocal.ThreadLocalMap类型的成员变量threadLocals，这个threadLocals就是用来存储实际的变量副本的，键值为当前ThreadLocal变量，value为变量副本（即T类型的变量. 初始时，在Thread里面，threadLocals为空，当通过ThreadLocal变量调用get()方法或者set()方法，就会对Thread类中的threadLocals进行初始化，并且以当前ThreadLocal变量为键值，以ThreadLocal要保存的副本变量为value，存到threadLocals。然后在当前线程里面，如果要使用副本变量，就可以通过get方法在threadLocals里面查找.



a). 实际的通过ThreadLocal创建的副本是存储在每个线程自己的threadLocals中的,

b). 为何threadLocals的类型ThreadLocalMap的键值为ThreadLocal对象，因为每个线程中可有多个threadLocal变量,

c). 在进行get之前，必须先set，否则会报空指针异常；如果想在get之前不需要调用set就能正常访问的话，必须重写initialValue()方法。

关于内存泄漏:ThreadLocalMap使用ThreadLocal对象作为弱引用,当垃圾回收的时候,ThreadLocalMap中的key(ThreadLocal)将会被回收,也就是key设置为null的Entry,如果线程迟迟无法结束,就是ThreadLocal对象将一直不会回收,那么会导致内存泄漏. 但是在调用ThreadLocal中你的remove, get,set方法时,会清除为null的弱引用,回收ThreadLocal.

a).JVM利用设置ThreadLocalMap的key为弱应用,避免内存泄漏;

b).JVM利用调用remove,get,set方法的时候回收弱引用;

c).当ThreadLocal存储很多key为null 的Entry的时候,而不在调用remove,get,set方法,那么将导致内存泄漏;

d).当使用static ThreadLocal的时候,延长了ThreadLocal的生命周期,那么可能导致内存泄漏,因为static变量在类未加载的时候,它就以及加载了, 当现场结束的时候,static变量不一定会回收,比起普通成员变量的时候加载,static生命周期加载将更容易导致内存泄漏.

e).ThreadLocal主要用在数据库连接的管理,session的管理

**synchronized:** synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性.

Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：

i)普通同步方法，锁是当前实例对象

ii)静态同步方法，锁是当前类的class对象

iii)同步方法块，锁是括号里面的对象

通过反编译java代码得到:

**同步普通方法:** synchronized方法则会被翻译成普通的方法调用和返回指令如:invokevirtual、areturn指令，在VM字节码层面并没有任何特别的指令来实现被synchronized修饰的方法，而是在Class文件的方法表中将该方法的access\_flags字段中的synchronized标志位置1，表示该方法是同步方法并使用调用该方法的对象或该方法所属的Class在JVM的内部对象表示class做为锁对象。

**同步代码块:** monitorenter指令插入到同步代码块的开始位置，monitorexit指令插入到同步代码块的结束位置，JVM需要保证每一个monitorenter都有一个monitorexit与之相对应。任何对象都有一个monitor与之相关联，当且一个monitor被持有之后，他将处于锁定状态。线程执行到monitorenter指令时，将会尝试获取对象所对应的monitor所有权，即尝试获取对象的锁.

**同步静态方法:**静态方法属于类方法,所以调用的时候需要获取同一个类上的monitor(每个类只对应一个class对象).

**synchronized同步的本质:**

**Java对象头:** synchronized用的锁是存在Java对象头里的, Hotspot虚拟机的对象头主要包括两部分数据：Mark Word（标记字段）、Class Pointer（类型指针）。其中Class Point是是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例，Mark Word用于存储对象自身的运行时数据，它是实现轻量级锁和偏向锁的关键.

Mark Word用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等等。Java对象头一般占有两个机器码（在32位虚拟机中，1个机器码等于4字节，也就是32bit），但是如果对象是数组类型，则需要三个机器码，因为JVM虚拟机可以通过Java对象的元数据信息确定Java对象的大小，但是无法从数组的元数据来确认数组的大小，所以用一块来记录数组长度.

Monitor: 一种同步机制，它通常被描述为一个对象。与一切皆对象一样，所有的Java对象是天生的Monitor，每一个Java对象都有成为Monitor的潜质，因为在Java的设计中 ，每一个Java对象自打娘胎里出来就带了一把看不见的锁，它叫做内部锁或者Monitor锁(监视锁)。Monitor 是线程私有的数据结构，每一个线程都有一个可用monitor record列表，同时还有一个全局的可用列表。每一个被锁住的对象都会和一个monitor关联（对象头的MarkWord中的LockWord指向monitor的起始地址），同时monitor中有一个Owner字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用:



**Owner**：初始时为NULL表示当前没有任何线程拥有该monitor record，当线程成功拥有该锁后保存线程唯一标识，当锁被释放时又设置为NULL；   
**EntryQ**:关联一个系统互斥锁（semaphore），阻塞所有试图锁住monitor record失败的线程。   
**RcThis**:表示blocked或waiting在该monitor record上的所有线程的个数。   
**Nest**:用来实现重入锁的计数。   
**HashCode**:保存从对象头拷贝过来的HashCode值（可能还包含GC age）。   
**Candidate**:用来避免不必要的阻塞或等待线程唤醒，因为每一次只有一个线程能够成功拥有锁，如果每次前一个释放锁的线程唤醒所有正在阻塞或等待的线程，会引起不必要的上下文切换（从阻塞到就绪然后因为竞争锁失败又被阻塞）从而导致性能严重下降。Candidate只有两种可能的值0表示没有需要唤醒的线程1表示要唤醒一个继任线程来竞争锁.

**JDK1.6后对synchronized实现的锁进行了优化,详情见问题24.**

**Automic:** Atomic的核心操作就是CAS（compare and set,利用CMPXCHG指令实现，它是一个原子指令）,该指令有三个操作数，变量的内存值V（value的缩写），变量的当前预期值E（exception的缩写），变量想要更新的值U（update的缩写），当内存值和当前预期值相同时，将变量的更新值覆盖内存值，执行伪代码如下。

if(V == E){

V = U

return true

}else{

return false

}

        现在我们就用CAS操作来解决上述问题。B线程将内存中的变量i读取一个临时变量中（假设此时读取的值为0），然后再将i的值读取到core1的算数运算单元中，接下来进行加1操作**，比较临时变量中的值和i当前的值是否相同，如果相同用运算单元中的结果（即i+1）的值覆盖内存中i的值**（注意这一部分就是CAS操作，它是个原子操作，不能被中断且其它线程中的CAS操作不能同时执行），否则指令执行失败。如果指令失败，说明A线程已经将i的值加1。由此可知如果两个线程一开始读取的i的值为都为0，那么必然只有一个线程的CAS操作能够成功，因为CAS操作不能并发执行。对于CAS操作执行失败的线程，只要循环执行CAS操作，那么一定能够成功。可以看到并没有线程阻塞，这和synchronize的原理有着本质的不同。

**Automic包以及源码分析:**

Atomic包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

        Atomic系列的类中的核心方法都会调用unsafe类中的几个本地方法。需要先知道一个东西就是Unsafe类，全名为：sun.misc.Unsafe，这个类包含了大量的对C代码的操作，包括很多直接内存分配以及原子操作的调用，而它之所以标记为非安全的，是告诉你这个里面大量的方法调用都会存在安全隐患，需要小心使用，否则会导致严重的后果，例如在通过unsafe分配内存的时候，如果自己指定某些区域可能会导致一些类似C++一样的指针越界到其他进程的问题。

Atomic包中的类按照操作的数据类型可以分成4组:

线程安全的基本类型的原子性操作: AtomicBoolean，AtomicInteger，AtomicLong

线程安全的数组类型的原子性操作，它操作的不是整个数组，而是数组中的单个元素: AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray;

基于反射原理对象中的基本类型（长整型、整型和引用类型）进行线程安全的操作: AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdate;

 线程安全的引用类型及防止ABA问题的引用类型的原子操作: AtomicReference ，AtomicMarkableReference，AtomicStampedReference.

以AutomicInteger为例:

a). 有参构造函数

public AtomicInteger(int initialValue) {

value = initialValue;

}

      从构造函数函数可以看出，数值存放在成员变量value中

private volatile int value;

      成员变量value声明为volatile类型，说明了多线程下的可见性，即任何一个线程的修改，在其它线程中都会被立刻看到

b). compareAndSet方法（value的值通过内部this和valueOffset传递）

public final boolean compareAndSet(int expect, int update) {

return unsafe.compareAndSwapInt(this, valueOffset, expect, update);

}//这个方法就是最核心的CAS操作

c). getAndSet方法 , 在该方法中调用了compareAndSet方法

public final int getAndSet(int newValue) {

for (;;) {

int current = get();

if (compareAndSet(current, newValue))

return current;

}

}

      如果在执行if (compareAndSet(current, newValue) 之前其它线程更改了value的值，那么导致   value 的值必定和current的值不同，compareAndSet执行失败，只能重新获取value的值，然后继续比较，直到成功。

d). i++的实现

public final int getAndIncrement() {

for (;;) {

int current = get();

int next = current + 1;

if (compareAndSet(current, next))

return current;

}

}

e). ++i的实现

public final int incrementAndGet() {

for (;;) {

int current = get();

int next = current + 1;

if (compareAndSet(current, next))

return next;

}

}

**4).ExecutorService, Callable, Future, FutureTask, CompletableFuture的区别**

**Future:** Future保存异步计算的结果,可以在我们执行任务时去做其他工作，并提供了以下几个方法  
\* cancel(boolean mayInterruptIfRunning)：试图取消执行的任务，参数为true时直接中断正在执行的任务，否则直到当前任务执行完成，成功取消后返回true，否则返回false  
\* isCancel()：判断任务是否在正常执行完前被取消的，如果是则返回true  
\* isDone()：判断任务是否已完成  
\* get()：等待计算结果的返回，如果计算被取消了则抛出  
\* get(long timeout,TimeUtil unit)：设定计算结果的返回时间，如果在规定时间内没有返回计算结果则抛出TimeOutException  
使用Future的好处：  
i). 获取任务的结果，判断任务是否完成，中断任务  
ii). Future的get方法很好的替代的了Thread.join或Thread.join(long millis)  
iii). Future的get方法可以判断程序代码(任务)的执行是否超时

**FutureTask:** FutureTask实现了RunnableFuture接口，提供了即可以使用Runnable来执行任务，又可以使用Future执行任务并取得结果的构造器，所以可以利用FutureTask去封装Runnable或Callable对象，之后再submit任务。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。 当然也可以调用线程直接执行(FutureTask.run()）。接下来我们根据FutureTask.run()的执行时机来分析其所处的3种状态：  
（i）未启动，FutureTask.run()方法还没有被执行之前，FutureTask处于未启动状态，当创建一个FutureTask，而且没有执行FutureTask.run()方法前，这个FutureTask也处于**未启动状态**。  
（ii）已启动，FutureTask.run()被执行的过程中，FutureTask处于**已启动状态**。  
（iii）已完成，FutureTask.run()方法执行完正常结束，或者被取消或者抛出异常而结束，FutureTask都处于**完成状态.**

**5).CyclicBarrier和CountDownLatch的区别, Semaphore与CountDownLatch的区别.**

**CyclicBarrier:** 回环栅栏，通过它可以实现让一组线程等待至某个状态之后再全部同时执行。叫做回环是因为当所有等待线程都被释放以后，CyclicBarrier可以被重用。我们暂且把这个状态就叫做barrier，当调用await()方法之后，线程就处于barrier了, CyclicBarrier类位于java.util.concurrent包下.

public CyclicBarrier(int parties, Runnable barrierAction) {

}//参数parties指让多少个线程或者任务等待至barrier状态；参数barrierAction为当这些线程都达到barrier状态时会执行的内容

public CyclicBarrier(int parties) {

}//

public int await() throws InterruptedException, BrokenBarrierException { };//用来挂起当前线程，直至所有线程都到达barrier状态再同时执行后续任务

public int await(long timeout, TimeUnit unit)throws InterruptedException,BrokenBarrierException,TimeoutException { };//让这些线程等待至一定的时间，如果还有线程没有到达barrier状态就直接让到达barrier的线程执行后续任务

**CountDownLatch:** CountDownLatch类位于java.util.concurrent包下，利用它可以实现类似计数器的功能。比如有一个任务A，它要等待其他4个任务执行完毕之后才能执行，此时就可以利用CountDownLatch来实现这种功能了:

public CountDownLatch(int count) {  };  //参数count为计数值

public void await() throws InterruptedException { };   //调用await()方法的线程会被挂起，它会等待直到count值为0才继续执行

public boolean await(long timeout, TimeUnit unit) throws InterruptedException { };  //和await()类似，只不过等待一定的时间后count值还没变为0的话就会继续执行

public void countDown() { };  //将count值减1

**Semphore:** 信号量，Semaphore可以控同时访问的线程个数，通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可, Semaphore类位于java.util.concurrent包下.

//构造方法

a)public Semaphore(int permits) {          //参数permits表示许可数目，即同时可以允许多少线程进行访问

    sync = new NonfairSync(permits);

}

b)public Semaphore(int permits, boolean fair) {    //这个多了一个参数fair表示是否是公平的，即等待时间越久的越先获取许可

    sync = (fair)? new FairSync(permits) : new NonfairSync(permits);

}

//阻塞方式的获取和释放许可

a)public void acquire() throws InterruptedException {  }     //获取一个许可

b)public void acquire(int permits) throws InterruptedException { }    //获取permits个许可

c)public void release() { }          //释放一个许可

d)public void release(int permits) { }    //释放permits个许可

acquire()用来获取一个许可，若无许可能够获得，则会一直等待，直到获得许可;

release()用来释放许可。注意，在释放许可之前，必须先获获得许可;

//非阻塞方式的获取和释放许可

a)public boolean tryAcquire() { };    //尝试获取一个许可，若获取成功，则立即返回true，若获取失败，则立即返回false

b)public boolean tryAcquire(long timeout, TimeUnit unit) throws InterruptedException { };  //尝试获取一个许可，若在指定的时间内获取成功，则立即返回true，否则则立即返回false

c)public boolean tryAcquire(int permits) { }; //尝试获取permits个许可，若获取成功，则立即返回true，若获取失败，则立即返回false

d)public boolean tryAcquire(int permits, long timeout, TimeUnit unit) throws InterruptedException { }; //尝试获取permits个许可，若在指定的时间内获取成功，则立即返回true，否则则立即返回false

通过availablePermits()方法得到可用的许可数目.

**CountDownLatch,CyclicBarrier,Semaphore的区别:**

i). CountDownLatch与CyclicBarrier虽然都采用计数方法等待线程，但是CountDownLatch是做减法，到0的时候放马。CyclicBarrier做的是加法;

ii). CountDownLatch等到是0的时候，就放马所有线程，但是它无法重置。CyclicBarrier也是等到线程都到位的时候，释放所有线程，但是CyclicBarrier是可以重复利用的;

iii). CountDownLatch调用countDown()后计数减一，调用await()方法只进行阻塞，对计数没任何影响；CyclicBarrier调用await()方法计数加1，若加1后的值不等于构造方法的值，则线程阻塞.

iv). Semaphore：是用来控制同时访问特定资源的线程数量，它通过协调各个线程，以保证合理的使用公共资源,Semaphore，每次semaphore.acquire()，获取一个资源，每次semaphore.acquire(n)，获取n个资源，当达到semaphore 指定资源数量时就不能再访问线程处于阻塞，必须等其它线程释放资源，semaphore.relase()每次资源一个资源，semaphore.relase(n)每次释放n个资源.

v). CountDownLatch、CyclicBarrier、Semaphore 都有一个int类型参数的构造方法。CountDownLatch、CyclicBarrier这个值作为计数用，达到该次数即释放等待的线程，而Semaphore 中所有acquire获取到的资源达到这个数，会使得其它线程阻塞.

**CountDownLatch使用场景:** 由于CountDownLatch有个countDown()方法并且countDown()不会引起阻塞，所以CountDownLatch可以应用于主线程等待所有子线程结束后再继续执行的情况。具体使用方式为new一个构造参数为subThread数目的CountDownLatch，启动所有子线程后主线程await(),在每个子线程的最后执行countDown()，这样当所有子线程执行完后计数减为0，主线程释放等待继续执行。比如赛跑，每个运动员看做一个子线程，裁判就是主线程，裁判发令（设置一个值为1的计数器，发令之前所有子线程await等待命令，裁判员发令让计数置为0，所有子线程同时开跑）所有运动员开跑后，需要等待所有人跑完再统计成绩（设置一个值为运动员数目的计数器，所有运动员开跑后裁判await被阻塞，每个运动员跑完的时候countDown()一下，所有运动员跑完计数达到0，裁判释放阻塞开始计分）。

**CyclicBarrier使用场景:** 由于CyclicBarrier计数达到指定后会重新循环使用，所以CyclicBarrier可以用在所有子线程之间互相等待多次的情形。比如在某种需求中，比如一个大型的任务，常常需要分配好多子任务去执行，只有当所有子任务都执行完成时候，才能执行主任务，这时候，就可以选择CyclicBarrier了。 比如团队旅游，一个团队通常分为几组，每组人走的路线可能不同，但都需要到达某一地点等待团队其它成员到达后才能进行下一站。

**Semaphore使用场景:** Semaphore可以用于做流量控制，特别公用资源有限的应用场景，比如数据库连接。假如有一个需求，要读取几万个文件的数据，因为都是IO密集型任务，我们可以启动几十个线程并发的读取，但是如果读到内存后，还需要存储到数据库中，而数据库的连接数只有10个，这时我们必须控制只有十个线程同时获取数据库连接保存数据，否则会报错无法获取数据库连接。这个时候，我们就可以使用Semaphore来做流控.

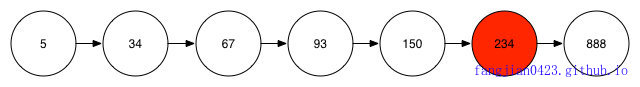
**6).多线程集合类：**

Map:

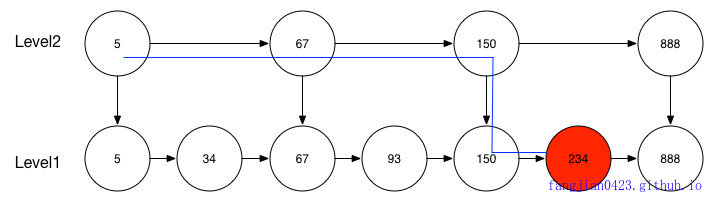
ConcurrentHashMap:[分析见集合模块问题4.](#_4).HashMap和ConcurrentHashMap的区别以及底层)

**ConcurrentSkipListMap**:

在分析ConcurrentSkipListMap之前, 先分析SkipList(跳表): 跳表是一种允许在一个有顺序的序列中进行快速查询的数据结构, 如果在普通的顺序链表中查询一个元素，需要从链表头部开始一个一个节点进行遍历，然后找到节点，如下图所示，要查找234元素的话需要从5元素节点开始一个一个节点进行遍历，这样的效率是非常低的:



跳表可以解决这种查询时间过长的问题：



从图中可以看到跳表有如下性质:

i). 由很多层组成，level越高的层节点越少，最底一层level拥有所有的节点数据;

ii). 每一层的节点数据也都是有顺序的;

iii). 上面层的节点肯定会在下面层中出现;

iv). 每个节点都有两个指针，分别是同一层的下一个节点指针和下一层节点的指针, 使用跳表查询元素的时间复杂度是O(log n)，跟红黑树一样, 但是跳表的存储容量变大了，本来一共只有7个节点的数据，使用跳表之后变成了14个.

跳表是一种使用”空间换时间”的概念用来提高查询效率的链表，开源软件Redis, LevelDB都使用到了跳表,跳表相比B树，红黑树，AVL树时间复杂度一样，但是耗费更多的存储空间，但是跳表的优势就是它相比树，实现简单，不需要考虑树的一些重新调整操作等.

ConcurrentSkipListMap提供了一种线程安全的并发访问的排序映射表(get的线程安全通过volatile保证, put通过CAS操作保证)。内部是SkipList（跳表）结构实现，在理论上能够在O(log(n))时间内完成查找、插入、删除操作:

// 每个节点的封装，跟层数没有关系

static final class Node<K,V> {

final K key; // 节点的关键字

volatile Object value; // 节点的值

volatile Node<K,V> next; // 节点的next节点引用

...

}

// 每一层节点的封装，叫做索引

static class Index<K,V> {

final Node<K,V> node; // 对应的节点

final Index<K,V> down; // 下一层索引

volatile Index<K,V> right; // 同一层的下一个索引

...

}

// 每一层的头索引

static final class HeadIndex<K,V> extends Index<K,V> {

final int level; // Level 级别

HeadIndex(Node<K,V> node, Index<K,V> down, Index<K,V> right, int level) {

super(node, down, right);

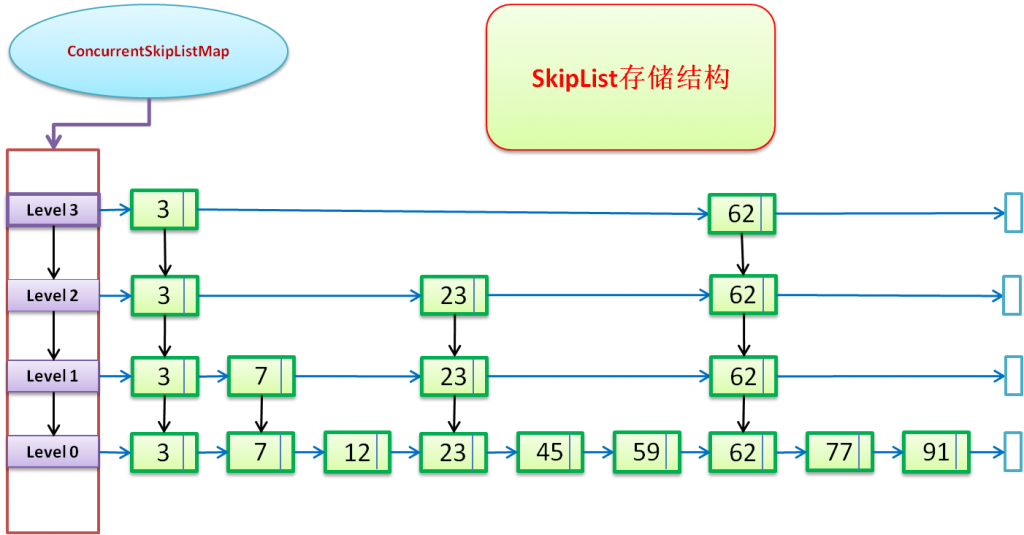
this.level = level;

}

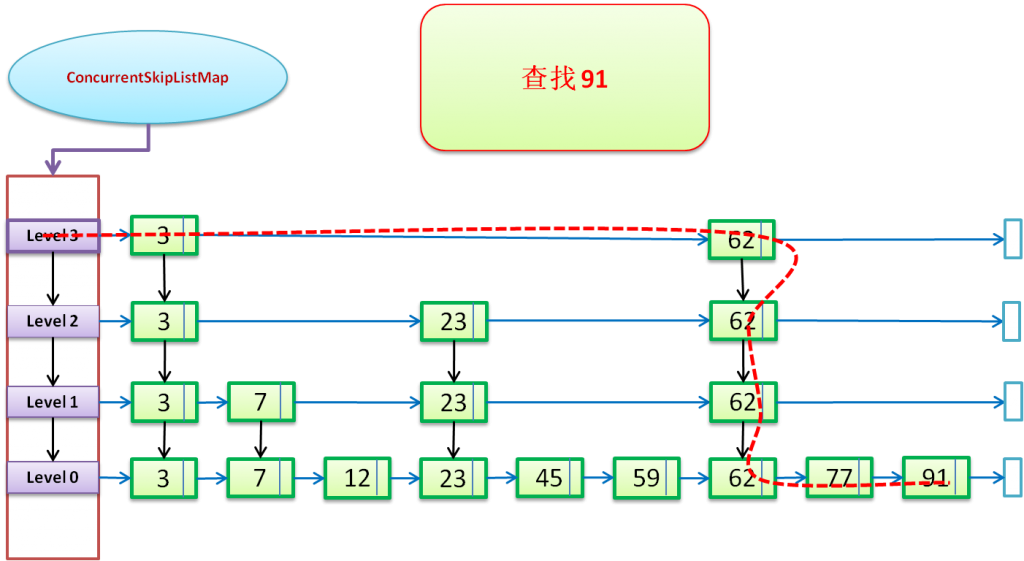
...

}

ConcurrentSkipListMap的存储结构如下图所示:



查找操作:



红色虚线，表示查找的路径，蓝色向右箭头表示right引用；黑色向下箭头表示down引用.

**CurrentSkipListMap的性质:**

i).分析CurrentSkipListMap的源码可以发现, 其get(调用doGet方法),put(调用doPut)以及remove(调用doRemove方法)方法中都没有用到锁操作,其中get由于其中跳表的结点Node类中的value是volatile修饰的,所以可以保证多线程读的可见性,**而put和remove操作是利用CAS的原子性操作,通过死循环判断新的结点与旧的是否发生静态条件,从而保证多线程的安全性**.

private V doPut(K kkey, V value, boolean onlyIfAbsent) {

Comparable<? super K> key = comparable(kkey);

for (;;) {

// 找到key的前继节点

Node<K,V> b = findPredecessor(key);

// 设置n为“key的前继节点的后继节点”，即n应该是“插入节点”的“后继节点”

Node<K,V> n = b.next;

for (;;) {

if (n != null) {

Node<K,V> f = n.next;

// 如果两次获得的b.next不是相同的Node，就跳转到”外层for循环“，重新获得b和n后再遍历。

if (n != b.next)

break;

// v是“n的值”

Object v = n.value;

// 当n的值为null(意味着其它线程删除了n)；此时删除b的下一个节点，然后跳转到”外层for循环“，重新获得b和n后再遍历。

if (v == null) { // n is deleted

n.helpDelete(b, f);

break;

}

// 如果其它线程删除了b；则跳转到”外层for循环“，重新获得b和n后再遍历。

if (v == n || b.value == null) // b is deleted

break;

// 比较key和n.key

int c = key.compareTo(n.key);

if (c > 0) {

b = n;

n = f;

continue;

}

if (c == 0) {

if (onlyIfAbsent || n.casValue(v, value))

return (V)v;

else

break; // restart if lost race to replace value

}

// else c < 0; fall through

}

// 新建节点(对应是“要插入的键值对”)

Node<K,V> z = new Node<K,V>(kkey, value, n);

// 设置“b的后继节点”为z

if (!b.casNext(n, z))

break; // 多线程情况下，break才可能发生(其它线程对b进行了操作)

// 随机获取一个level

// 然后在“第1层”到“第level层”的链表中都插入新建节点

int level = randomLevel();

if (level > 0)

insertIndex(z, level);

return null;

}

}

}

ii).CurrentSkipListMap是TreeMap的多线程版本,支持key的排序(自然顺序或者自定义顺序);

iii).ConcurrentSkipListMap支持更高的并发,存取时间是O(lgn),与线程数无关,即在数据量一定的情况下,并发线程数越多,ConcurrentSkipList越能体现出优势.

**ConcurrentSkipListSet:**

i).基于ConcurrentSkipListMap实现, 取其中的key , ConcurrentSkipSet是TreeSet的并发版本,支持排序;

ii).ConcurrentSkipListSet继承于AbstractSet。因此，它本质上是一个集合;

iii). ConcurrentSkipListSet是通过ConcurrentSkipListMap实现的。它包含一个ConcurrentNavigableMap对象m，而m对象实际上是ConcurrentNavigableMap的实现类ConcurrentSkipListMap的实例。ConcurrentSkipListMap中的元素是key-value键值对；而ConcurrentSkipListSet是集合，它只用到了ConcurrentSkipListMap中的key

CopyOnWriteArrayList:[分析见集合中的问题5](#_5).ArrayList与CopyOnWriteArrayList的区)

CopyOnWriteArraySet:

i). CopyOnWriteArraySet和HashSet虽然都继承于共同的父类AbstractSet；但是，HashSet是通过[“散列表](http://www.cnblogs.com/skywang12345/p/3310835.html)(HashMap)”实现的，而CopyOnWriteArraySet则是通过“[动态数组(CopyOnWriteArrayList)](http://www.cnblogs.com/skywang12345/p/3498483.html)”实现的(构造函数中直接new 一个CopyOnWriteArrayList, 成员变量也一个CopyOnWriteArrayList);

ii) CopyOnWriteArraySet是一个集合,其中的方法多数是直接调用CopyOnWriteArrayList中的方法, 但是CopyOnWriteArrayList是动态数组,可以允许重复元素,但是CopyOnWriteArraySet不允许重复,所以在CopyOnWriteArrayList中额外提供了addIfAbsent()和addAllAbsent()这两个添加元素的API，通过这些API来添加元素时，只有当元素不存在时才执行添加操作(保证set中没有重复元素);

iii). CopyOnWriteArraySet的“线程安全”机制，和[CopyOnWriteArrayList](http://www.cnblogs.com/skywang12345/p/3498483.html)一样，是通过volatile和互斥锁来实现的.

**Queue: ArrayBlockingQueue, LinkedBlockingQeque, LinkedBlockingDeque, ConcurrentLinkedQueue, ConcurrentLinkedDeque, BlockingDeque,DelayQueue,PriorityBlockingQueue**

**ArrayBlockingQueue:**

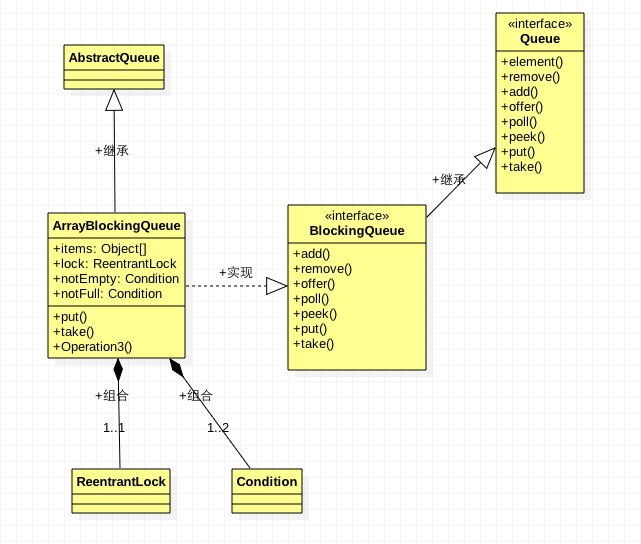
i).  ArrayBlockingQueue：基于数组实现的一个阻塞队列，在创建ArrayBlockingQueue对象时必须制定容量大小。 并且可以指定公平性与非公平性，默认情况下为非公平的，即不保证等待时间最长的队列最优先能够访问队列;

ii). ArrayBlockingQueue内部通过Object[]数组保存数据的，也就是说ArrayBlockingQueue本质上是通过数组实现的。ArrayBlockingQueue的大小，即数组的容量是在创建创建ArrayBlockingQueue时候指定的;

iii). 如下图所示，ArrayBlockingQueue和ReentrantLock是组合关系，ArrayBlockingQueue中包含一个ReentrantLock对象。ReentrantLock是可重入的互斥锁。ArrayBlockingQueue就是根据ReentrantLock互斥锁实现"多线程对共享资源的访问"。ReentrantLock分为公平锁和非公平锁，关于具体使用公平锁还是非公平锁，在创建ArrayBlockingQueue时可以指定；而且，ArrayBlockingQueue默认会使用非公平锁;

iv). ArrayBlockingQueue和Condition是组合关系，ArrayBlockingQueue中包含两个Condition对象(notEmpty和notFull)。使用通知模式实现:所谓通知模式，当生产者往满的队列里面添加元素的时候，会阻塞生产者（调用Condition notFull.await()进行等待）；当消费者消费了一个队列中的元素后，会通知（调用Condition notFull.signal()唤醒生产者)生产者当前队列可用。反之，当消费者消费的时候，发现队列是空的，则消费者会被阻塞（通过Condition的 notEmpty.await()进行等待），当生产者插入了队列中的一个元素后，则会调用notEmpty.signal()唤醒消费者继续消费.

v).ArrayBlockingQueue中的ReentrantLock lock与Condition: **Lock的作用是提供独占锁机制，来保护竞争的资源;而Condition是为了更精细的对锁进行控制，但是依赖于lock，通过某个条件对多线程进行控制.**



**LinkedBlockingQeque:**

LinkedBlockingQueue是基于链表的阻塞队列，同ArrayListBlockingQueue类似，其内部也维持着一个数据缓冲队列（该队列由一个链表构成），当生产者往队列中放入一个数据时，队列会从生产者手中获取数据，并缓存在队列内部，而生产者立即返回；只有当队列缓冲区达到最大值缓存容量时（LinkedBlockingQueue可以通过构造函数指定该值），才会阻塞生产者队列，直到消费者从队列中消费掉一份数据，生产者线程会被唤醒，反之对于消费者这端的处理也基于同样的原理。而LinkedBlockingQueue之所以能够高效的处理并发数据，还因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步，这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能.

**ArrayBlockingQueue与LinkedBlockingQueue的区别:**

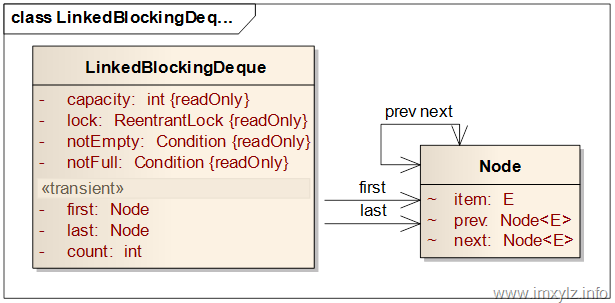
i).队列中锁的实现不同: ArrayBlockingQueue实现的队列中的锁是没有分离的，即生产和消费用的是同一个锁, LinkedBlockingQueue实现的队列中的锁是分离的，即生产用的是putLock，消费是takeLock;

ii). 在生产或消费时操作不同:ArrayBlockingQueue实现的队列中在生产和消费的时候，是直接将枚举对象插入或移除的,  LinkedBlockingQueue实现的队列中在生产和消费的时候，需要把枚举对象转换为Node<E>进行插入或移除，会影响性能;

iii). 队列大小初始化方式不同: ArrayBlockingQueue实现的队列中必须指定队列的大小, LinkedBlockingQueue实现的队列中可以不指定队列的大小，但是默认是Integer.MAX\_VALUE.

**LinkedBlockingDeque:**

双向并发阻塞队列,双向是指可以从队列的头和尾同时操作，并发只是线程安全的实现，阻塞允许在入队出队不满足条件时挂起线程，这里说的队列是指支持FIFO/FILO实现的链表.



i). 要想支持阻塞功能，队列的容量一定是固定的，否则无法在入队的时候挂起线程。也就是capacity是final类型的;

ii). 双向链表，每一个结点就需要前后两个引用，这样才能将所有元素串联起来，支持双向遍历。也即需要prev/next两个引用;

iii). LinkedBlockingDeque多了addFirst，addLast，offerFirst，offerLast，peekFirst，peekLast等方法，以First单词结尾的方法，表示插入，获取（peek）或移除双端队列的第一个元素。以Last单词结尾的方法，表示插入，获取或移除双端队列的最后一个元素;

iv). 双向阻塞队列LinkedBlockingDeque可以运用在“工作窃取”模式中(fork/join框架);

iv). LinkedBlockingDeque实现了BlockingDeque接口，它支持多线程并发。当多线程竞争同一个资源时，某线程获取到该资源之后，其它线程需要阻塞等待; LinkedBlockingDeque是通过双向链表实现的;  first是双向链表的表头, last是双向链表的表尾; count是LinkedBlockingDeque的实际大小，即双向链表中当前节点个数;  capacity是LinkedBlockingDeque的容量，它是在创建LinkedBlockingDeque时指定的;

v). lock是控制对LinkedBlockingDeque的互斥锁，当多个线程竞争同时访问LinkedBlockingDeque时，某线程获取到了互斥锁lock，其它线程则需要阻塞等待，直到该线程释放lock，其它线程才有机会获取lock从而获取cpu执行权;

vi). notEmpty和notFull分别是“非空条件”和“未满条件”。通过它们能够更加细腻进行并发控制.

**BlockingDeque:**继承Queue接口,提供操作阻塞队列的方法(会抛出InterruptedException).

**ConcurrentLinkedQueue:**ConcurrentLinkedQueue是Queue的一个安全实现．Queue中元素按FIFO原则进行排序．采用CAS操作，来保证元素的一致性. Node是ConcurrentLinkedQueue定义的内部对象，其内部定义了item变量用来包裹实际入队元素及next变量用来保存当前节点的下一节点引用。且上述变量都被volatile关键字修饰，这意味着对item变量和next变量的读写都会被立刻刷入主存，可以被其他线程及时看到。Node还定义了其他一系列可以更改item及next变量的方法。这些方法底层都是通过CAS来实现，CAS会使用现代处理器上提供的高效机器级别的原子指令，也就是说这些方法涉及的操作都是原子操作。循环CAS更新volatile变量，这是JDK实现非阻塞类库的主要方式.

**ConcurrentLinkedDeque:**非阻塞，无锁，无界 ，线程安全双端操作的队列(对称双向链接的链表结构),是ConcurrentLinkedQueue的升级版; 一个节点定义成包含前置节点pre、数据项item、后置节点next。当节点item!=null的时候视为节点还存活，当通过CAS设置item为null的时候，逻辑上该节点已经从集合中删除了。任何时候显然第一个节点的前置节点为null，其终结任何链表的从活着的结点开始的前置引用，最后一个节点同样的，其终结任何链表从活着的结点开始后面的引用。第一个节点和最后一个节点并不一定活着，而且总是能够相互到达。一个新元素原子添加是通过CAS设置前置引用或后置引用为null入队列.

**DelayQueue:** 阻塞队列，并且元素是Delay的子类，保证元素在达到一定时间后才可以取得到.

**PriorityBlockingQueue:** 原理和内部结构跟PriorityQueue基本一样，只是多了个线程安全,基于优先级堆的无界的并发安全的优先级队列，队列的元素按照其自然顺序进行排序，或者根据构造队列时提供的 Comparator 进行排序，具体取决于所使用的构造方法. PriorityBlockingQueue是基于优先级，不是FIFO, 可以用来实现优先级的线程池，高优先级的先执行，低优先级的后执行.其中使用ReentrantLock保证线程安全.

**7).Collections.synchronizedXXX与普通集合类之间的区别, Collections.synchronizedXXX与JUC中并发集合的区别**

Collections.synchronizedXXX是对Collections类中设计到的集合类进行synchronized同步的综合, 其中synchronized是用在代码块中而不是同步到方法中.

JUC是JDK1.5提出的java并发集合包, 具体分析见集合类分析和并发集合分析.

**8).竞态条件(先检查后执行Check-Then-Act)和临界区**

当两个线程竞争同一资源时，如果对资源的访问顺序敏感，就称存在竞态条件。导致竞态条件发生的代码区称作临界区。在临界区中使用适当的同步就可以避免竞态条件.

**9).多线程间如何通信？进程间如何通信？死锁，如何避免死锁？如何解决死锁?活锁与死锁的区别？检测线程是否拥有锁的方法(Thread类中的holdsLock()方法)**

Java中线程间的通信:

i). 假设有两个线程，一个是线程 A，另一个是线程 B，两个线程分别依次打印 1-3 三个数字即可

private static void demo2() {

    Thread A = new Thread(new Runnable() {

        @Override

        public void run() {

            printNumber("A");

        }

    });

    Thread B = new Thread(new Runnable() {

        @Override

        public void run() {

            System.out.println("B 开始等待 A");

            try {

                A.join();

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

            printNumber("B");

        }

    });

    B.start();

    A.start();

}

A.join() 方法会让 B 一直等待直到 A 运行完毕

**ii). 如何让 两个线程按照指定方式有序交叉运行**

private static void demo3() {

    Object lock = new Object();

    Thread A = new Thread(new Runnable() {

        @Override

        public void run() {

            synchronized (lock) {

                System.out.println("A 1");

                try {

                    lock.wait();

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

                System.out.println("A 2");

                System.out.println("A 3");

            }

        }

    });

    Thread B = new Thread(new Runnable() {

        @Override

        public void run() {

            synchronized (lock) {

                System.out.println("B 1");

                System.out.println("B 2");

                System.out.println("B 3");

                lock.notify();

            }

        }

    });

    A.start();

    B.start();

}

A 1

A waiting…

B 1

B 2

B 3

A 2

A 3

执行步骤:

a)首先创建一个 A 和 B 共享的对象锁 lock = new Object();

当 A 得到锁后，先打印 1，然后调用 lock.wait() 方法，交出锁的控制权，进入 wait 状态；

b)对 B 而言，由于 A 最开始得到了锁，导致 B 无法执行；直到 A 调用 lock.wait() 释放控制权后， B 才得到了锁；

c)B 在得到锁后打印 1， 2， 3；然后调用 lock.notify() 方法，唤醒正在 wait 的 A;

d)A 被唤醒后，继续打印剩下的 2，3。

**iii). 四个线程 A B C D，其中 D 要等到 A B C 全执行完毕后才执行，而且 A B C 是同步运行的**

A B C 三个线程同时运行，各自独立运行完后通知 D；对 D 而言，只要 A B C 都运行完了，D 再开始运行。针对这种情况，我们可以利用 CountdownLatch 来实现这类通信方式。它的基本用法是:

a)创建一个计数器，设置初始值，CountdownLatch countDownLatch = new CountDownLatch(3)

b)在 等待线程 里调用 countDownLatch.await() 方法，进入等待状态，直到计数值变成 0

c). 在 其他线程 里，调用 countDownLatch.countDown() 方法，该方法会将计数值减小 1

d). 当 其他线程 的 countDown() 方法把计数值变成 0 时，等待线程 里的 countDownLatch.await() 立即退出，继续执行下面的代码

private static void runDAfterABC() {

    int worker = 3;

    CountDownLatch countDownLatch = new CountDownLatch(worker);

    new Thread(new Runnable() {

        @Override

        public void run() {

            System.out.println("D is waiting for other three threads");

            try {

                countDownLatch.await();

                System.out.println("All done, D starts working");

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }).start();

    for (char threadName='A'; threadName <= 'C'; threadName++) {

        final String tN = String.valueOf(threadName);

        new Thread(new Runnable() {

            @Override

            public void run() {

                System.out.println(tN + "is working");

                try {

                    Thread.sleep(100);

                } catch (Exception e) {

                    e.printStackTrace();

                }

                System.out.println(tN + "finished");

                countDownLatch.countDown();

            }

        }).start();

    }

}

输出结果:

D is waiting for other three threads

A is working

B is working

C is working

A finished

C finished

B finished

All done, D starts working

CountDownLatch 就是一个倒计数器，我们把初始计数值设置为3，当 D 运行时，先调用 countDownLatch.await() 检查计数器值是否为 0，若不为 0 则保持等待状态；当A B C 各自运行完后都会利用countDownLatch.countDown()，将倒计数器减 1，当三个都运行完后，计数器被减至 0；此时立即触发 D 的 await() 运行结束，继续向下执行

**iv). 三个运动员各自准备，等到三个人都准备好后，再一起跑**

a). 先创建一个公共 CyclicBarrier 对象，设置 同时等待 的线程数，CyclicBarrier cyclicBarrier = new CyclicBarrier(3);

b). 这些线程同时开始自己做准备，自身准备完毕后，需要等待别人准备完毕，这时调用 cyclicBarrier.await(); 即可开始等待别人；

c). 当指定的 同时等待 的线程数都调用了 cyclicBarrier.await();时，意味着这些线程都准备完毕好，然后这些线程才 同时继续执行

private static void runABCWhenAllReady() {

    int runner = 3;

    CyclicBarrier cyclicBarrier = new CyclicBarrier(runner);

    final Random random = new Random();

    for (char runnerName='A'; runnerName <= 'C'; runnerName++) {

        final String rN = String.valueOf(runnerName);

        new Thread(new Runnable() {

            @Override

            public void run() {

                long prepareTime = random.nextInt(10000) + 100;

                System.out.println(rN + "is preparing for time:" + prepareTime);

                try {

                    Thread.sleep(prepareTime);

                } catch (Exception e) {

                    e.printStackTrace();

                }

                try {

                    System.out.println(rN + "is prepared, waiting for others");

                    cyclicBarrier.await(); // 当前运动员准备完毕，等待别人准备好

                } catch (InterruptedException e) {

                    e.printStackTrace();

                } catch (BrokenBarrierException e) {

                    e.printStackTrace();

                }

                System.out.println(rN + "starts running"); // 所有运动员都准备好了，一起开始跑

            }

        }).start();

    }

}

输出结果:

|  |
| --- |
| A is preparing for time: 4131  B is preparing for time: 6349  C is preparing for time: 8206    A is prepared, waiting for others    B is prepared, waiting for others    C is prepared, waiting for others    C starts running  A starts running  B starts running |

**v).子线程完成某件任务后，把得到的结果回传给主线程**

让子线程去计算从 1 加到 100，并把算出的结果返回到主线程

private static void doTaskWithResultInWorker() {

    Callable<Integer> callable = new Callable<Integer>() {

        @Override

        public Integer call() throws Exception {

            System.out.println("Task starts");

            Thread.sleep(1000);

            int result = 0;

            for (int i=0; i<=100; i++) {

                result += i;

            }

            System.out.println("Task finished and return result");

            return result;

        }

    };

    FutureTask<Integer> futureTask = new FutureTask<>(callable);

    new Thread(futureTask).start();

    try {

        System.out.println("Before futureTask.get()");

        System.out.println("Result:" + futureTask.get());

        System.out.println("After futureTask.get()");

    } catch (InterruptedException e) {

        e.printStackTrace();

    } catch (ExecutionException e) {

        e.printStackTrace();

    }

}

输出结果:

|  |
| --- |
| Before futureTask.get()  Task starts  Task finished and return result  Result: 5050  After futureTask.get() |

主线程调用 futureTask.get() 方法时阻塞主线程；然后 Callable 内部开始执行，并返回运算结果；此时 futureTask.get() 得到结果，主线程恢复运行; 通过 FutureTask 和 Callable 可以直接在主线程获得子线程的运算结果，只不过需要阻塞主线程。当然，如果不希望阻塞主线程，可以考虑利用 ExecutorService，把 FutureTask 放到线程池去管理执行.

**vi).管道:** 创建管道输出流PipedOutputStream pos和管道输入流PipedInputStream pis; 将pos和pis匹配，pos.connect(pis); 将pos赋给信息输入线程，pis赋给信息获取线程，就可以实现线程间的通讯了.

管道的缺点:

a)管道流只能在两个线程之间传递数据线程consumer1和consumer2同时从pis中read数据，当线程producer往管道流中写入一段数据后，每一个时刻只有一个线程能获取到数据，并不是两个线程都能获取到producer发送来的数据，因此一个管道流只能用于两个线程间的通讯。不仅仅是管道流，其他IO方式都是一对一传输。

b）管道流只能实现单向发送，如果要两个线程之间互通讯，则需要两个管道流

**进程间如何通信**

a)管道（Pipe）：管道可用于具有亲缘关系进程间的通信，允许一个进程和另一个与它有共同祖先的进程之间进行通信。  
b）命名管道（named pipe）：命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关 系 进程间的通信。命名管道在文件系统中有对应的文件名。命名管道通过命令mkfifo或系统调用mkfifo来创建。  
c）信号（Signal）：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送 信号给进程本身；linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数）。  
d）消息（Message）队列：消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺  
e）共享内存：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。  
f）内存映射（mapped memory）：内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它。  
g）信号量（semaphore）：主要作为进程间以及同一进程不同线程之间的同步手段。

h）套接字（Socket）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。

**解决死锁:** 在内置锁中,恢复的唯一办法就是重新启动程序,从而防止死锁的唯一方法就是在构造程序时避免出现不一致的锁顺序;可轮训锁与可轮训的锁提供了另一种避免死锁的方法.(具体分析见问题24)

**10).notify,nofityAll的区别，为什么在Object类里而不在Thread类里？**

**锁池:** 设线程A已经拥有了某个对象(注意:不是类)的锁，而其它的线程想要调用这个对象的某个synchronized方法(或者synchronized块)，由于这些线程在进入对象的synchronized方法之前必须先获得该对象的锁的拥有权，但是该对象的锁目前正被线程A拥有，所以这些线程就进入了该对象的锁池中。

**等待池:** 假设一个线程A调用了某个对象的wait()方法，线程A就会释放该对象的锁(因为wait()方法必须出现在synchronized中，这样自然在执行wait()方法之前线程A就已经拥有了该对象的锁)，同时线程A就进入到了该对象的等待池中。如果另外的一个线程调用了相同对象的notifyAll()方法，那么处于该对象的等待池中的线程就会全部进入该对象的锁池中，准备争夺锁的拥有权。如果另外的一个线程调用了相同对象的notify()方法，那么仅仅有一个处于该对象的等待池中的线程(随机)会进入该对象的锁池.

**notify与notifyAll的区别:**

i).如果线程调用了对象的wait()方法,那么线程便会处于该对象的等待池中,等待池中的线程不会去竞争该对象的锁.

ii).当有线程调用了notifyAll()方法(唤醒所有wait线程)或者notify()方法(只随机的唤醒一个wait线程),被唤醒的线程便会进入该对象的锁池中,锁池中的线程会去竞争该对象所,即:调用notify方法后只有一个线程会由等待池进入锁池,而调用notifyAll方法会将该对象的等待池中的所有线程移到锁池中,等待锁竞争.

iii).优先级高的线程竞争到对象的锁的概率大,假若某线程没有竞争到该对象锁,它还会留在锁池中,唯有线程再次调用wait方法,它才会重新回到等待池中,而竞争到对象锁的线程则继续往下执行,直到执行完了synchronized代码块,它会释放掉该对象锁,这时锁池中的线程会继续竞争该对象锁.

所谓唤醒线程,就是将线程由等待池移到锁池,notifyAll调用后,会将全部线程由等待池移到锁池,然后参与锁的竞争,竞争成功则继续执行,如果不成功则留在锁池中等待锁被释放后再次参与竞争,而notify只会唤醒一个线程.

**为什么notify,wait,notifyAll在Object里面而不在Thread里面:** 由于每个对象都拥有monitor（即锁），所以让当前线程等待某个对象的锁，当然应该通过这个对象来操作了。而不是用当前线程来操作，因为当前线程可能会等待多个线程的锁，如果通过线程来操作，就非常复杂了. 如果调用某个对象的wait()方法，当前线程必须拥有这个对象的monitor（即锁），因此调用wait()方法必须在同步块或者同步方法中进行（synchronized块或者synchronized方法）。

调用某个对象的wait()方法，相当于让当前线程交出此对象的monitor，然后进入等待状态，等待后续再次获得此对象的锁（Thread类中的sleep方法使当前线程暂停执行一段时间，从而让其他线程有机会继续执行，但它并不释放对象锁）；notify()方法能够唤醒一个正在等待该对象的monitor的线程，当有多个线程都在等待该对象的monitor的话，则只能唤醒其中一个线程，具体唤醒哪个线程则不得而知。同样地，调用某个对象的notify()方法，当前线程也必须拥有这个对象的monitor，因此调用notify()方法必须在同步块或者同步方法中进行（synchronized块或者synchronized方法）。nofityAll()方法能够唤醒所有正在等待该对象的monitor的线程，这一点与notify()方法是不同的

[为什么**notify,wait,notifyAll在synchronized中执行?**](#_16)._Condition中的await,_signal,signa)

**11).ExecutorService中invokeAll与invokeAny的区别**

**i).invokeAny(...)：**方法接收的是一个Callable的集合，执行这个方法不会返回Future，但是会返回所有Callable任务中其中一个任务的执行结果。这个方法也无法保证返回的是哪个任务的执行结果，反正是其中的某一个,只要有一个任务执行完成(没有抛出异常),线程池会终止其他未完成的任务; 如果提交的任务列表中没有能够正常完成的任务,那么最终会抛出异常; invokeAny和任务提交的顺序无关,只是返回最早执行完的任务的结果。

**ii).invokeAll(...)：**与 invokeAny(...)类似也是接收一个Callable集合，但是invokeAll执行之后会返回一个Future的List，其中对应着每个Callable任务执行后的Future对象;

invokeAll是一个阻塞方法，会等待任务列表中的所有任务都执行完成。不管任务是正常完成，还是异常终止，Future.isDone()始终返回true。通过Future.isCanceled()可以判断任务是否在执行的过程中被取消。通过Future.get()可以获取任务的返回结果，或者是任务在执行中抛出的异常; invokeAll(Collection<? extends Callable<T>> tasks,long timeout, TimeUnit unit), 给定的超时期满，还没有完成的任务会被取消，即Future.isCancelled()返回true；在超时期之前，无论是正常完成还是异常终止的任务，Future.isCancelled()返回false; 一旦ExecutorService.invokeAll()方法产生了异常，线程池中还没有完成的任务会被取消执行.

**12).interrupt,interrupted和isInterrupted的区别**

二者都是Thread中的方法

public static boolean interrupted () {

return currentThread().isInterrupted(true);

}

public boolean isInterrupted () {

return isInterrupted( false);

}

private native boolean isInterrupted( boolean ClearInterrupted);

i)interrupt用于中断线程, 调用该方法的线程的状态将被设置为中断状态(线程的中断只是设置线程中断的状态为,并不会真正中断线程,需要用户自己去监视线程的状态并作出处理); 支持线程中断的方法（也就是线程中断后会抛出InterruptedException的方法，比如这里的sleep，以及Object.wait等方法）就是在监视线程的中断状态，一旦线程的中断状态被置为“中断状态”，就会抛出中断异常.

ii)当中断线程调用静态方法Thread.interrupted来检查中断状态时,中断状态就会被清零, 而非静态状态isInterrupted方法用来查询其他线程的中断状态且不会改变中断状态的标识.

iii)从interrupted和isInterrupted的源码看, 二者都是调用同一个native方法,不同的是interrupted方法会传入true参数用于对中断状态的清零, 而isInterrupted掺入false,中断状态不会清零.

**13).synchronized与ReentrantLock的区别,双检锁**

**synchronized与ReentrantLock的对比:**

**i)使用场景:** ReentrantLock在内存上的语义与synchronize相同, 但是它提供了额外的功能, 可以作为一种高级工具. 当需要一些 **可定时, 可轮询, 可中断的锁获取操作, 或者希望使用公平锁, 或者使用非块结构的编码时** 才应该考虑ReetrantLock ; 在业务并发简单清晰的情况下推荐synchronized, 在业务逻辑并发复杂, 或对使用锁的扩展性要求较高时, 推荐使用ReentrantLock这类锁. 另外今后JVM的优化方向一定是基于底层synchronized的, 性能方面应该选择synchronized.

**ii)公平性:** synchronized不支持公平性。一旦锁被释放了，任何线程都能获得锁，不能指定哪个线程能够优先获取锁；另一方面，通过在创建ReentrantLock实例的时候我们可以指定公平属性。在竞争锁的过程中，通过指定公平性，我们能够为最长等待线程（即等待此锁最长的线程）提供锁.

public ReentrantLock(){

sync = new NonfairSync();

}

public ReentrantLock(boolean fair){

sync = fair ? new FairSync() : new NonfairSync();

}

**iii). tryLock方法**, ReentrantLock提供了tryLock方法，当且仅当锁是可用的并且没有被其他的线程持有时tryLock方法才能获取此锁。这减轻了线程在等待锁时产生的阻塞.

**iv).ReentrantLock**在等待获取锁的过程中提供了中断，即我们能够中断获取锁的过程。在synchronized中，一个线程在等到锁的过程中会一直阻塞，如果陷入了无限等待中，我们是没有办法控制这个过程的。而ReentrantLock提供了一个lockInterruptibly方法，当线程正在等待获取锁的过程中，我们能够用此方法来中断线程。同时我们也可以给tryLock指定一个超时时间，在这个时间范围内如果没有获得锁，则返回false。

public boolean tryLock(long timeout, TimeUnit unit)

throws InterruptedException {

return sync.tryAcquireNanos(1, unit.toNanos(timeout));

}

**v).从锁的角度分析:** synchronized原始采用的是CPU悲观锁机制，即线程获得的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。而在CPU转换线程阻塞时会引起线程上下文切换，当有很多线程竞争锁的时候，会引起CPU频繁的上下文切换导致效率很低; 而Lock用的是乐观锁方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁实现的机制就是CAS操作（Compare and Swap）。进一步研究ReentrantLock的源代码，会发现其中比较重要的获得锁的一个方法是[compareAndSetState](#_31).CAS操作,_ABA问题,_AQS)。这里其实就是调用的CPU提供的特殊指令.

**ReentrantLock的优势:**

i)能够中断获取锁过程   
ii)在等待锁时可以指定超时时间   
iii)能够创建公平锁   
iv)获取所有等待锁的线程   
v)可以灵活的尝试获取锁而不锁定线程

**ReentrantLock的劣势:**

i).ReentrantLock的主要缺点是使用ReentrantLock时必须将ReentrantLock中的方法放到try-finally结构中，这就造成了代码不易读并且隐藏了业务逻辑.

ii) 使用ReentrantLock之后，类的获取以及释放全部由程序员负责，这虽然为我们编写代码提供了更多功能，但是也带来了新的bug，特别是我们忘记在finally块中释放锁.

**ReentrantLock的使用:**

**i).Lock接口(基于乐观锁):** lock()、tryLock()、tryLock(long time, TimeUnit unit)和lockInterruptibly()是用来获取锁的。unLock()方法是用来释放锁的.

public interface Lock {

void lock();

void lockInterruptibly() throws InterruptedException;

boolean tryLock();

boolean tryLock(long time, TimeUnit unit) throws InterruptedException;

void unlock();

Condition newCondition();

}

**tryLock()**方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false，也就说这个方法无论如何都会立即返回。在拿不到锁时不会一直在那等待;

**tryLock(long time,TimeUnitunit)**方法和tryLock()方法是类似的，只不过区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回false。如果如果一开始拿到锁或者在等待期间内拿到了锁，则返回true;

**lockInterruptibly()**方法比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就使说，当两个线程同时通过lock.lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程

**ii).ReentrantLock实现了Lock接口**: ReentrantLock中有3个内部类，分别是Sync、FairSync和NonfairSync; [Sync是一个继承AQS的抽象类](#_31).CAS操作,_ABA问题,_AQS)，使用独占锁，复写了tryRelease方法。tryAcquire方法由它的两个FairSync(公平锁)和NonfairSync(非公平锁)实现; ReentrantLock的lock方法使用sync的lock方法，Sync的lock方法是个抽象方法，由公平锁和非公平锁去实现。[unlock方法直接使用AQS的release方法](#_31).CAS操作,_ABA问题,_AQS)。所以说公平锁和非公平锁的释放锁过程是一样的，不一样的是获取锁过程.  
public ReentrantLock() {

// 默认非公平策略

sync = new NonfairSync();

}

public ReentrantLock(boolean fair) {

sync = fair ? new FairSync() : new NonfairSync();

}

iii). Java.util.concurrent.locks.ReadWriteLock(接口)有一种高级的线程锁机制，它允许多个线程读某个资源，但每次只允许一个线程来写; ReadWriteLock内部实际上包含两个Lock示例。一个保护读操作，另一个保护写操作.  
ReadLock   
如果没有写入线程锁住ReadWriteLock，并且没有线程需要获得写入锁进行写入操作。那么多个线程可以获得锁来进行读操作。

WriteLock   
如果没有线程在写或者读操作，那么一次仅有一个线程可以获得锁以进行写操作。

iv). ReentrantReadWriteLock实现了ReadWriteLock接口; 如果有一个线程已经占用了读锁，则此时其他线程如果要申请写锁，则申请写锁的线程会一直等待释放读锁; 如果有一个线程已经占用了写锁，则此时其他线程如果申请写锁或者读锁，则申请的线程会一直等待释放写锁.

**synchronized与ReentrantLock使用时的选择:**

i）Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；  
ii）synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；  
iii）Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；  
iv）通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。  
v）Lock可以提高多个线程进行读操作的效率。

在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而当竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。

**14).三个线程T1,T2,T3如何确保他们按顺序执行**

方法1: 在t1中调用t2, t2中调用t3;

方法2: 用join方法, 等另一个执行完了在执行自己的线程

**15).yield,sleep,wait区别**

**sleep()**   
　　sleep()方法需要指定等待的时间，它可以让当前正在执行的线程在指定的时间内暂停执行，进入阻塞状态，该方法既可以让其他同优先级或者高优先级的线程得到执行的机会，也可以让低优先级的线程得到执行机会。但是sleep()方法不会释放“锁标志”，也就是说如果有synchronized同步块，其他线程仍然不能访问共享数据。

**wait()**   
　　wait()方法notify()及notifyAll()，这三个方法用于协调多个线程对共享数据的存取，所以必须在synchronized语句块内使用([为什么会在synchronized中执行](#_16)._Condition中的await,_signal,signa))，也就是说，调用wait()，notify()和notifyAll()的任务在调用这些方法前必须拥有对象的锁。注意，它们都是Object类的方法，而不是Thread类的方法。   
　　wait()方法与sleep()方法的不同之处在于，wait()方法会释放对象的“锁标志”。当调用某一对象的wait()方法后，会使当前线程暂停执行，并将当前线程放入对象等待池中，直到调用了notify()方法后，将从对象等待池中移出任意一个线程并放入锁标志等待池中，只有锁标志等待池中的线程可以获取锁标志，它们随时准备争夺锁的拥有权。当调用了某个对象的notifyAll()方法，会将对象等待池中的所有线程都移动到该对象的锁标志等待池。   
　　除了使用notify()和notifyAll()方法，还可以使用带毫秒参数的wait(long timeout)方法，效果是在延迟timeout毫秒后，被暂停的线程将被恢复到锁标志等待池。   
　　此外，wait()，notify()及notifyAll()只能在synchronized语句中使用，但是如果使用的是ReenTrantLock实现同步，该如何达到这三个方法的效果呢？解决方法是使用ReenTrantLock.newCondition()获取一个Condition类对象，然后Condition的await()，signal()以及signalAll()分别对应上面的三个方法.

**yield()**   
　　yield()方法和sleep()方法类似，也不会释放“锁标志”，区别在于，它没有参数，即yield()方法只是使当前线程重新回到可执行状态，所以执行yield()的线程有可能在进入到可执行状态后马上又被执行，另外yield()方法只能使同优先级或者高优先级的线程得到执行机会，这也和sleep()方法不同。

**join()**   
　　join()方法会使当前线程等待调用join()方法的线程结束后才能继续执行.

**16). Condition中的await, signal,signalAll与Object中的wait, notify, notifyAll之间的区别和联系,为什么wait, notify,notifyAll会在synchronized中执行?**

i). wait()、notify()和notifyAll()是基于synchronized, Condition是基于Lock的。

ii). Condition是在java 1.5中才出现的，它用来替代传统的Object的wait()、notify()实现线程间的协作，相比使用Object的wait()、notify()，使用Condition的await()、signal()这种方式实现线程间协作更加安全和高效。阻塞队列实际上是使用了Condition来模拟线程间协作。

iii)Condition是个接口，基本的方法就是await()和signal()方法；

iv)Condition依赖于Lock接口，生成一个Condition的基本代码是lock.newCondition()

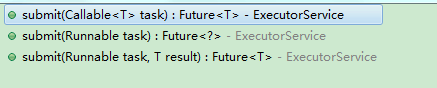
v)调用Condition的await()和signal()方法，都必须在lock保护之内，就是说必须在lock.lock()和lock.unlock之间才可以使用

vi)对应关系:Conditon中的await()对应Object的wait()；Condition中的signal()对应Object的notify()；Condition中的signalAll()对应Object的notifyAll()。

vii).谓词条件的不同:  synchronized 方式对应的 wait, notify 不能有多个谓词条件，Lock 对应的 Condition await, signal 则可以有多个谓词条件; 例如队列已满，所有的生产者现场阻塞，某个时刻消费者消费了一个元素，则需要唤醒某个生产者线程，而通过 Object notify 方式唤醒的线程不能确保一定就是一个生产者线程，因为 notify 是随机唤醒某一个正在该 synchronized 对应的锁上面通过 wait 方式阻塞的线程，如果这时正好还有消费者线程也在阻塞中，则很可能唤醒的是一个消费者线程；notifyAll 更是会唤醒所有在对应锁上通过 wait 方式阻塞的线程，而不管是生产者还是消费者线程。与之不同的 Condition await, signal 方式则可以对应多个谓词条件（notEmpty, notFull），可以很方便的实现让生产者线程和消费者线程分别在不同的谓词条件上进行等待.

**17).submit与execute的区别**

submit:



execute:

https://img-blog.csdn.net/20151202163634814

i). 参数类型: execute只能接受Runnable类型的任务; submit不管是Runnable还是Callable类型的任务都可以接受，但是Runnable返回值均为void，所以使用Future的get()获得的还是null;

ii). 返回值: execute没有返回值; submit有返回值，所以需要返回值的时候必须使用submit;

iii).异常处理: execute中的是Runnable接口的实现，所以只能使用try、catch来捕获CheckedException，通过实现UncaughtExceptionHande接口处理UncheckedException, 即和普通线程的处理方式完全一致; submit中**不管提交的是Runnable还是Callable类型的任务，如果不对返回值Future调用get()方法，都会吃掉异常.**

**为什么wait, notifyAll, notify会在synchronized中执行:**

i).从Java语言层面分析: 如果不在synchronized中调用这三个方法会导致抛出IllegalMonitorStateException异常.

ii).底层源码分析: synchronized代码块通过javap生成的字节码中包含”monitorenter”和”monitorexit”指令,执行monitorenter指令可以获取对象的monitor, 而wait方法的调用时通过native方法的wait(0)调用的, 该方法表明调用wait方法时必须拥有该对象的monitor. 如果A线程和B线程同时执行monitorenter指令,并获取当前对象关联的monitor,只有一个线程可以和monitor建立联系, 假设A执行加锁成功, 那么线程B竞争加锁失败,进入等待队列进行等待,当A遇到wait方法时,会将线程A放入wait set中,等待被唤醒,并放弃当前对象上的所有同步声明, 此时通过notify通知线程B进行加锁操作.HotSpot虚拟机中,monitor采用ObjectMonitor实现.

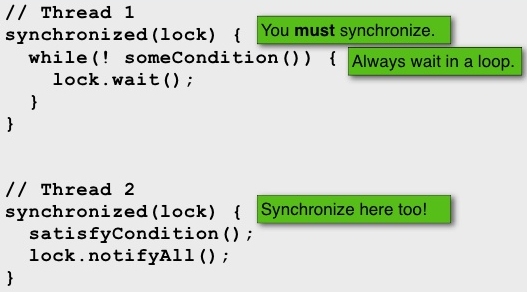
iii).wait和notify之间存在任何的竞态条件: 例如在生产者-消费者模型中:

a).生产者测试条件(缓冲区已满或未满)并确保在缓冲区满时进入等待状态;

b).消费者在消费了一个缓冲区元素后设置等待条件;

c).消费者调用notify方法, 如果此时生产者线程没有处于等待状态则不会收到通知;

d).生产者进入等待状态;

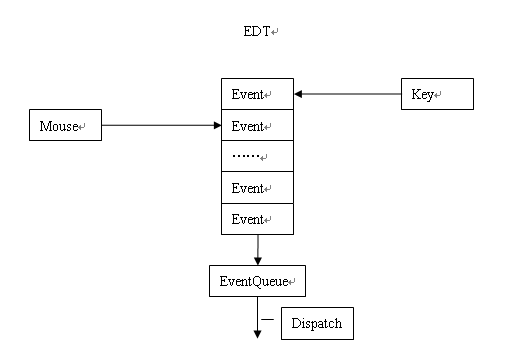


因此由于[竞态条件](#_8).竞态条件(先检查后执行Check-Then-Act)),潜在的失去了一个通知,并且如果使用了缓冲区或者只有一个生产者线程,则该线程将永远等待, 程序将会挂起.为了解决这种竞态条件java中需要通过加锁或者关键字synchronized来解决; 由于Java中的wait()方法在等待之前释放锁，并在从wait()方法返回之前重新获取锁，因此我们必须使用该锁来确保条件检查（缓冲区已满或未满），和条件重置（从缓冲区获取元素）是原子的，这可以通过在Java中使用synchronized方法或块来实现.

**18).invokeAndWait和invokeLater的区别**

首先swing是单线程的, 当swing界面程序启动的时候，会启动3个线程: i)主线程; ii)系统工具包线程:负责捕获操作系统事件，然后将事件转换成swing的事件，然后发送到事件派发线程EDT; iii)事件派发线程(EDT):将事件派发到各个组件，并负责调用绘制方法更新界面.

所有的事件，例如键盘，鼠标事件，都会由工具包线程转换成swing事件，然后放到事件队列EventQueue中，而这个EventQueue的派发机制是由EDT来管理的:



何修改组件状态的方法都应该在EDT中执行, 在EDT中执行长时间的事件，使EDT不能及时响应更新界面的事件，就是所说的界面卡住, 所以必须避免在EDT中执行长时间的操作，而避免的方法就是多线程，启动另外的线程来处理冗长的操作，比如操作数据库，读写文件等，在这过程中可能要更新界面来给用户以提示，比如显示一个进度条，过一段事件更新一下界面，但是在EDT以外的线程中更新界面都是无效的，这在前面已经说过，要更新界面就要将对界面的更新操作放到EDT中，但是事件又是在另外的线程中执行的，要解决这个问题就要使用SwingUtilities提供的一个方法了 invokeLater; invokeLater负责创建一个含有Runnable的特定事件，并让其在EDT中排队等待调用，当被调用时就会运行Runnable中的run方法进行派发.

**invokeLater和invokeAndWait的区别:**

i)与invoikeLater一样，invokeAndWait也把可运行对象排入事件派发线程的队列中，invokeLater在把可运行的对象放入队列后就返回，而invokeAndWait一直等待知道已启动了可运行的run方法才返回。如果一个操作在另外一个操作执行之前必须从一个组件获得信息，则invokeAndWait方法是很有用的.

ii). 可以从事件派发线程中调用invokeLater，却不能从事件派发线程中调用invokeAndWait，从事件派发线程调用invokeAndWait的问题是：invokeAndWait锁定调用它的线程，直到可运行对象从事件派发线程中派发出去并且该可运行的对象的run方法激活，如果从事件派发线程调用invoikeAndWait，则会发生死锁的状况，因为invokeAndWait正在等待事件派发，但是，由于是从事件派发线程中调用invokeAndWait，所以直到invokeAndWait返回后事件才能派发

**19).ReadWriteLock(具体分析见问题13)**

Java.util.concurrent.locks.ReadWriteLock(接口)有一种高级的线程锁机制，它允许多个线程读某个资源，但每次只允许一个线程来写; ReadWriteLock内部实际上包含两个Lock示例。一个保护读操作，另一个保护写操作.  
ReadLock   
如果没有写入线程锁住ReadWriteLock，并且没有线程需要获得写入锁进行写入操作。那么多个线程可以获得锁来进行读操作。

WriteLock   
如果没有线程在写或者读操作，那么一次仅有一个线程可以获得锁以进行写操作。

iv). ReentrantReadWriteLock实现了ReadWriteLock接口; 如果有一个线程已经占用了读锁，则此时其他线程如果要申请写锁，则申请写锁的线程会一直等待释放读锁; 如果有一个线程已经占用了写锁，则此时其他线程如果申请写锁或者读锁，则申请的线程会一直等待释放写锁.

**20).多线程中的忙循环**

忙循环就是程序员用循环让一个线程等待，不像传统方法wait(), sleep() 或 yield() 它们都放弃了CPU控制，而忙循环不会放弃CPU，它就是在运行一个空循环。这么做的目的是为了保留CPU缓存，在多核系统中，一个等待线程醒来的时候可能会在另一个内核运行，这样会重建缓存。为了避免重建缓存和减少等待重建的时间就可以使用它了。

**21).同步块抛出异常会发生什么?**

无论同步块是正常还是异常退出的，里面的线程都会释放锁

**22).双检锁(DCL)**

class SomeClass {  
 private Resource resource = null;  
 public Resource getResource() {  
 if (resource == null) {  
 synchronized {  
 if (resource == null)  
 resource = new Resource();  
 }  
 }  
 return resource;  
 }  
}

当一个线程运行到resource = new Resource();时,因为new一个对象需要分配空间,初始化字段,调用构造方法.当为resource分配好空间后,外面的其它线程就可以看到不为null的resource,而这时有可能还没有初始化字段和调用构造方法就被其它线程引用了(早期错误结论).确实,在JAVA2(以jdk1.2开始)以前对于实例字段是直接在主储区读写的.所以当一个线程对resource进行分配空间,初始化和调用构造方法时,可能在其它线程中分配空间动作可见了,而初始化和调用构造方法还没有完成.

但是从JAVA2以后,JMM发生了根本的改变,分配空间,初始化,调用构造方法只会在线程的工作存储区完成,在没有向主存储区复制赋值时,其它线程绝对不可能见到这个过程.而这个字段复制到主存区的过程,更不会有分配空间后没有初始化或没有调用构造方法的可能.在JAVA中,一切都是按引用的值复制的.向主存储区同步其实就是把线程工作存储区的这个已经构造好的对象有压缩堆地址值COPY给主存储区的那个变量.这个过程对于其它线程,要么是resource为null,要么是完整的对象.绝对不会把一个已经分配空间却没有构造好的对象让其它线程可见.

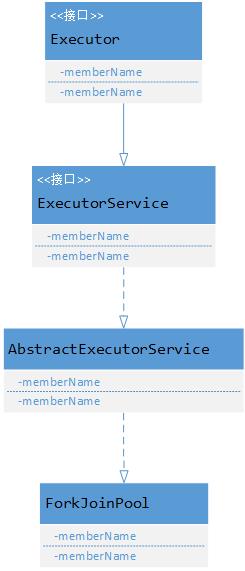
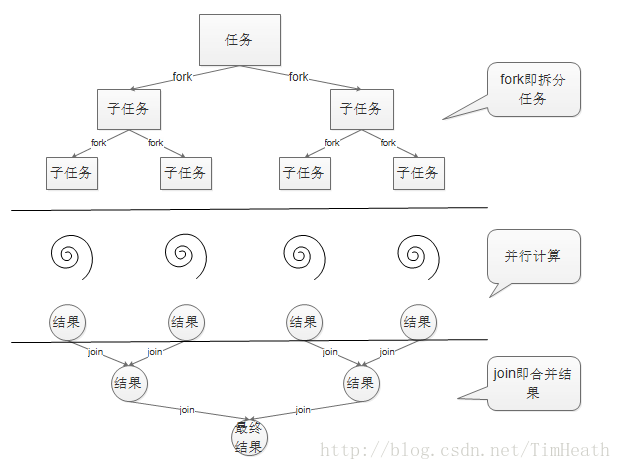
public MyObject{  
 private static MyObect obj;  
 private Date d = new Data();  
 public Data getD(){return this.d;}  
 public static MyObect getInstance(){  
 if(obj == null){  
 synchronized(MyObect .class){  
 if(obj == null)  
 obj = new MyObject();//这里  
 }  
 }  
 return obj;  
 }  
}

 一个线程A运行到"这里"时,对于A的工作区中,肯定已经产生一个MyObect对象,而且这时这个对象已经完成了Data d.现在线程A调用时间到,执行权被切换到另一个线程B来执行,会有什么问题呢? 如果obj不为null,线程B获得了一个obj,但可能obj.getD()却还没有初始化.为什么?既然obj已经可见了(线程A还没有离开同步块),而d却不可见呢?如果d不可见,那么obj也应该为null啊?线程B应该等待A释放同步块啊?事实上,对于"这里"这条语句,线程A还没有离开同步块.因为没有"离开同步块"这个条件,线程a的工作区没有强制与主存储器同步,这时工作区中有两个字段obj,d 到底先把谁同步到主存储区,没有条件限制,虽然在线程A的工作区obj和d都是完整的,但有JSL没有强制不允许先把obj映射到主存储区,如果哪个jvm实现按它的优化方案先把工作存储器中的obj同步到主存储器了,这时正好线程B获取了,而d却没有同步过去,那么线程B就获取了obj的引用却找不能obj.getD();

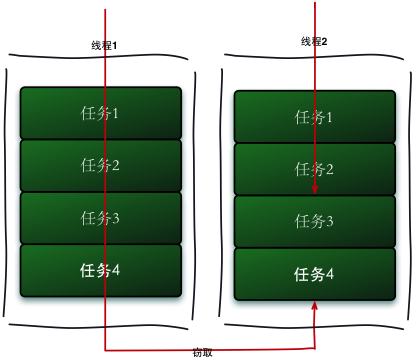
       我们发现,其实对于安全性问题都是基于即时构造对象这样的条件下的.如果把DCL用来控制其它不重复操作,它就不会出现这样的问题,就象上面从JNDI中查找DataSource,因为查找本身是耗时的,所以我用DCL来控制"查找"这个行为而不是控制对象本身.其实还有很多可以正确应用DCL的地方.象JLive,OFBiz这些开源项目中都在大量应用DCL

**23).fork join框架**

Fork/Join框架是Java 7提供了的一个用于并行执行任务的框架， 把大任务分割成若干个小任务，最终把每个小任务结果汇总起来得到大任务结果的框架。是ExecutorService接口的一个实现，可以帮助开发人员充分利用多核处理器的优势，编写出并行执行的程序，提高应用程序的性能；设计的目的是为了处理那些可以被递归拆分的任务. 与其它ExecutorService的实现类相似，会给线程池中的线程分发任务，不同之处在于它使用了工作窃取算法，所谓工作窃取，指的是对那些处理完自身任务的线程，会从其它线程窃取任务执行。fork/join框架的核心是[ForkJoinPool](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html)类，该类继承了AbstractExecutorService类。ForkJoinPool实现了工作窃取算法并且能够执行 [ForkJoinTask](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html)任务。



**工作窃取模式(ForkJoinPool):** ForkJoinTask需要通过ForkJoinPool来执行， 是一种任务调度方法，由多个工作线程组成，每个工作线程用一个双端队列维护一组任务。任务分割出的子任务会添加到当前工作线程所维护的双端队列中(或者叫做 deque, Java 6 在类库中添加了几种 deque实现,包 括 ArrayDeque 和 LinkedBlockingDeque)，进入队列的头部。当一个工作线程的队列里暂时没有任务时，它会随机从其他工作线程的队列的尾部获取一个任务。比起传统的线程池类ThreadPoolExecutor，ForkJoinPool 实现了工作窃取算法，使得空闲线程能够主动分担从别的线程分解出来的子任务，从而让所有的线程都尽可能处于饱满的工作状态，提高执行效率. 如下图所示: 线程1负责处理第一个队列里的任务。但是有的线程会先把自己队列里的任务干完，而其他线程对应的队列里还有任务等待处理。干完活的线程与其等着，不如去帮其他线程干活，于是它就去线程2的队列里窃取一个任务来执行。而在这时它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。



**为什么要用工作窃取(双端队列):** 与标准队列相比，deque 具有两方面的优势：减少争用和窃取。因为只有工作线程会访问自身的 deque 的头部，deque 头部永远不会发生争用；因为只有当一个线程空闲时才会访问 deque 的尾部，所以也很少存在线程的 deque 尾部的争用（在 fork-join 框架中结合 deque 实现会使这些访问模式进一步减少协调成本）。跟传统的基于线程池的方法相比，减少争用会大大降低同步成本。此外，这种方法暗含的后进先出（last-in-first-out，LIFO）任务排队机制意味着最大的任务排在队列的尾部，当另一个线程需要窃取任务时，它将得到一个能够分解成多个小任务的任务，从而避免了在未来窃取任务。因此，工作窃取实现了合理的负载平衡，无需进行协调并且将同步成本降到了最小。

Fork/Join原理:

i).ForkJoinPool由ForkJoinTask数组和ForkJoinWorkerThread数组组成，ForkJoinTask数组负责存放程序提交给ForkJoinPool的任务，而ForkJoinWorkerThread数组负责执行这些任务;

ii). ForkJoinTask的fork方法实现原理。当我们调用ForkJoinTask的fork方法时，程序会调用ForkJoinWorkerThread的pushTask方法异步的执行这个任务，然后立即返回结果。

iii). pushTask方法把当前任务存放在ForkJoinTask 数组queue里。然后再调用ForkJoinPool的signalWork()方法唤醒或创建一个工作线程来执行任务。

iv). ForkJoinTask的join方法实现原理。Join方法的主要作用是阻塞当前线程并等待获取结果, 首先，它调用了doJoin()方法，通过doJoin()方法得到当前任务的状态来判断返回什么结果，任务状态有四种：已完成（NORMAL），被取消（CANCELLED），信号（SIGNAL）和出现异常（EXCEPTIONAL）。在doJoin()方法里，首先通过查看任务的状态，看任务是否已经执行完了，如果执行完了，则直接返回任务状态，如果没有执行完，则从任务数组里取出任务并执行。如果任务顺利执行完成了，则设置任务状态为NORMAL，如果出现异常，则纪录异常，并将任务状态设置为EXCEPTIONAL。

**ForkJoinPool 提供了三类方法来调度子任务:**

a).execute 系列：异步执行指定的任务;

b).invoke 和 invokeAll：执行指定的任务，等待完成，返回结果;

c).submit 系列：异步执行指定的任务并立即返回一个 Future 对象.

**ForkJoinTask**：我们要使用ForkJoin框架，必须首先创建一个ForkJoin任务。它提供在任务中执行fork()和join()操作的机制，通常情况下我们不需要直接继承ForkJoinTask类，而只需要继承它的子类，Fork/Join框架提供了以下两个子类:

RecursiveTask：用于有返回结果的任务;

RecursiveAction：用于没有返回结果的任务;

**Fork/Join的异常处理:** ForkJoinTask在执行的时候可能会抛出异常，但是我们没办法在主线程里直接捕获异常，所以ForkJoinTask提供了isCompletedAbnormally()方法来检查任务是否已经抛出异常或已经被取消了，并且可以通过ForkJoinTask的getException方法获取异常。使用如下代码：

if(task.isCompletedAbnormally())

{

System.out.println(task.getException());

}

**ForkJoinPool与一般ExecutorService实现的差别:**

ForkJoin实现了ExecutorService接口，这个接口就是用来把任务交给线程池中的工作线程去执行。ForkJoin也是一个ExecutorService，但区别在于ForkJoin使用了work-stealing算法，见前面的介绍。普通的线程池是按FIFO的方式执行，而ForkJoin优先执行（由其它任务）后创建子任务。对于大部分会产生子任务的任务模式，ForkJoin的处理实现会很高效。如果设置了异步模式， ForkJoin也可能适合执行事件类型（不需要join）的任务.

**应用场景:** Fork/Join框架适合能够进行拆分再合并的计算密集型（CPU密集型）任务。Fork/Join框架是一个并行框架，因此要求服务器拥有多CPU、多核，用以提高计算能力. 如果是单核、单CPU，不建议使用该框架，会带来额外的性能开销，反而比单线程的执行效率低。当然不是因为并行的任务会进行频繁的线程切换，因为Fork/Join框架在进行线程池初始化的时候默认线程数量为Runtime.getRuntime().availableProcessors()，单CPU单核的情况下只会产生一个线程，并不会造成线程切换，而是会增加Fork/Join框架的一些队列、池化的开销。

**24).锁的使用, 公平锁与非公平锁；死锁，如何避免死锁(Java Concurrency in Practice p178)，轮训锁与定时锁, Java中出现的锁的对比,可重入锁的原理. 锁的内存语义**

锁存在Java对象头里。如果对象是数组类型，则虚拟机用3个Word（字宽）存储对象头，如果对象是非数组类型，则用2字宽存储对象头。在32位虚拟机中，一字宽等于四字节，即32bit.(对象头的详解见问题3)

**公平锁与非公平锁:** 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. 对于Java ReentrantLock而言，通过构造函数指定该锁是否是公平锁，默认是非公平锁。非公平锁的优点在于吞吐量比公平锁大。对于Synchronized而言，也是一种非公平锁。由于其并不像ReentrantLock是通过AQS的来实现线程调度，所以并没有任何办法使其变成公平锁. 公平锁和非公平锁的最主要区别就是获取锁的方式不一样; 公平锁获取锁的时候，首先先读取状态位state，然后再做判断，之后使用cas设置状态位。能获取锁的线程就获取锁，不能获取锁的线程被挂起进入队列。之后再来的线程的等待时间没有已经在队列里的线程等待时间长，所以会一直进入等待队列; 非公平锁获取锁的时候，立马就使用cas判断设置状态位，是一种抢占式的方式。同时非公平锁也没有等待时间长的线程会优先获取锁这个概念。

**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, 且不受影响. ReentrantLock 和synchronized 都是 可重入锁. 可重入锁的最大作用是避免死锁.

**偏向锁:** 偏向锁的目的是在某个线程获得锁之后，消除这个线程锁重入（CAS）的开销，看起来让这个线程得到了偏护. 偏向锁的释放不需要做任何事情，这也就意味着加过偏向锁的MarkValue(对象头中的字段)会一直保留偏向锁的状态，因此即便同一个线程持续不断地加锁解锁，也是没有开销的。  
　　另一方面，偏向锁比轻量锁更容易被终结，轻量锁是在有锁竞争出现时升级为重量锁，而一般偏向锁是在有不同线程申请锁时升级为轻量锁，这也就意味着假如一个对象先被线程1加锁解锁，再被线程2加锁解锁，这过程中没有锁冲突，也一样会发生偏向锁失效，不同的是这回要先退化为无锁的状态，再加轻量锁

**轻量级锁:** 线程在执行同步块之前，JVM会先在当前线程的栈桢中创建用于存储锁记录的空间，并将对象头中的Mark Word复制到锁记录中，官方称为Displaced Mark Word。然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，则自旋获取锁，当自旋获取锁仍然失败时，表示存在其他线程竞争锁(两条或两条以上的线程竞争同一个锁)，则轻量级锁会膨胀成重量级锁.

**重量级锁:** 重量锁在JVM中又叫对象监视器（Monitor），它很像C中的Mutex，除了具备Mutex互斥的功能，它还负责实现了Semaphore的功能，也就是说它至少包含一个竞争锁的队列，和一个信号阻塞队列（wait队列），前者负责做互斥，后一个用于做线程同步  


**自旋锁:** 如果自旋锁被另外一个线程对象持有，那么当前获取锁的线程将陷入while循环等待，直到那个持有自旋锁的线程对象释放它所持有的自旋锁，那么那些想要获取该自旋锁的线程对象 将会有一个获得该自旋锁; 基于他这种原理，等待的时候，并不释放cpu时间片，相比synchronized  wait()操作，减小了释放，重新获取的消耗。 该自旋锁适用于，当前线程竞争不强烈的时候使用.JDK 1.6提出了适应自旋锁,即如果上次自旋锁成功了, 那么下次会允许自旋等待持续更多的次数, 反之，如果对于某个锁，很少有自旋能够成功的，那么在以后要或者这个锁的时候自旋的次数会减少甚至省略掉自旋过程，以免浪费处理器资源.

**锁粗化:** 将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁。

**消除锁:** 为了保证数据的完整性，我们在进行操作时需要对这部分操作进行同步控制，但是在有些情况下，JVM检测到不可能存在共享数据竞争，这是JVM会对这些同步锁进行锁消除。锁消除的依据是逃逸分析的数据支持。如StringBuffer、Vector、HashTable等，这个时候会存在隐形的加锁操作。比如StringBuffer的append()方法，Vector的add()方法, 如果JVM可以明显检测到变量vector没有逃逸出方法之外，所以JVM可以大胆地将vector内部的加锁操作消除.

**锁的内存语义:** 当线程释放锁时，JMM会把该线程对应的本地内存中的共享变量刷新到主内存中; 当线程获取锁时，JMM会把该线程对应的本地内存置为无效。从而使得被监视器保护的临界区代码必须要从主内存中去读取共享变量:

i)线程A释放一个锁，实质上是线程A向接下来将要获取这个锁的某个线程发出了（线程A对共享变量所做修改的）消息。  
ii)线程B获取一个锁，实质上是线程B接收了之前某个线程发出的（在释放这个锁之前对共享变量所做修改的）消息。  
iii)线程A释放锁，随后线程B获取这个锁，这个过程实质上是线程A通过主内存向线程B发送消息  
**死锁:** 死锁是指**两个或两个以上的进程**在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们将一直互相等待而无法推进下去。也就是说，死锁会让你的程序挂起无法完成任务。

**活锁:** **活锁进程的状态可以改变（死锁不能改变），但是和死锁一样无法继续执行。**活锁可以理解为在狭小的山道，两辆车相向而行，为了避让而同时往一个方向转头，结果谁都过不去。

**Java中发生死锁的条件:**

| **死锁条件** | **描述** |
| --- | --- |
| 不剥夺 | 进程已经获得的资源，未使用完之前不能强行剥夺 |
| 请求与保持 | 一个进程因请求资源而阻塞时，对已获得的资源保持不放 |
| 互斥 | 一个资源每次只能被一个进程使用 |
| 循环等待 | 若干进程之间形成一种头尾相接的循环等待资源关系 |

**如何阻止死锁:** 选择阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作来避免死锁.

**消除死锁的方法:**

i). 最简单、最常用的方法就是进行系统的重新启动，不过这种方法代价很大，它意味着在这之前所有的进程已经完成的计算工作都将付之东流，包括参与死锁的那些进程，以及未参与死锁的进程;

ii).撤消进程，剥夺资源。终止参与死锁的进程，收回它们占有的资源，从而解除死锁。这时又分两种情况：一次性撤消参与死锁的全部进程，剥夺全部资源；或者逐步撤消参与死锁的进程，逐步收回死锁进程占有的资源。一般来说，选择逐步撤消的进程时要按照一定的原则进行，目的是撤消那些代价最小的进程，比如按进程的优先级确定进程的代价；考虑进程运行时的代价和与此进程相关的外部作业的代价等因素；

iii).进程回退策略，即让参与死锁的进程回退到没有发生死锁前某一点处，并由此点处继续执行，以求再次执行时不再发生死锁。虽然这是个较理想的办法，但是操作起来系统开销极大，要有堆栈这样的机构记录进程的每一步变化，以便今后的回退，有时这是无法做到的.

**Java诊断死锁的工具以及方法:**

**Thread Dump:** Thread Dump是非常有用的诊断Java应用问题的工具。每一个Java虚拟机都有及时生成所有线程在某一点状态的thread-dump的能力，虽然各个 Java虚拟机打印的thread dump略有不同，但是大多都提供了当前活动线程的快照，及JVM中所有Java线程的堆栈跟踪信息，堆栈信息一般包含完整的类名及所执行的方法，如果可能的话还有源代码的行数.

**Thread Dump的特点:** 能在各种操作系统下使用; 能在各种Java应用服务器下使用; 可以在生产环境下使用而不影响系统的性能; 可以将问题直接定位到应用程序的代码行上.

**Thread Dump能诊断的问题:** 查找内存泄露，常见的是程序里load大量的数据到缓存.

**Thread Dump的使用:** 通过它可以得到线程，线程运行状态、标识、调用的堆栈，堆栈包含完整的类名，所执行的方法，以及源代码所在的行数等等信息:

首先通过jps查看进程的信息, 从而获得java进程的进程号。也可以用top动态观察CPU使用情况分析进程情况。然后通过jdk自带的工具jstack进行分析。jstack <pid> >> filename 可以将pid对应的java进程的线程使用情况写到后面的file中去。

**jstack Dump 日志文件中的线程状态**

dump 文件里，值得关注的线程状态有：

i)死锁，Deadlock（重点关注）

ii)执行中，Runnable

iii)等待资源，Waiting on condition（重点关注）

iv)等待获取监视器，Waiting on monitor entry（重点关注）

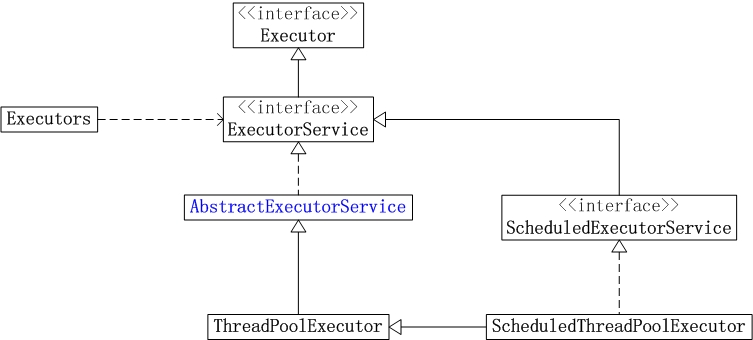
v)暂停，Suspended

vi)对象等待中，Object.wait() 或 TIMED\_WAITING

vii)阻塞，Blocked（重点关注）

viii)停止，Parke

**25).线程池的使用：Executor框架, newFixedThreadPool, new CachedThreadPool, newSingleThreadExecutor, newScheduledThreadPool, ScheduleThreadPoolExecutor, ScheduleExecutorService, CompletionService: Executor与BlockingQueue, newSingleThreadScheduledExecutor**



**Executor：**是Java线程池的超级接口；提供一个execute(Runnable command)方法;我们一般用它的继承接口ExecutorService; 是个静态工厂类。它通过静态工厂方法返回ExecutorService、ScheduledExecutorService、ThreadFactory 和 Callable 等类的对象;

**Executors：**是java.util.concurrent包下的一个类，提供了若干个静态方法，用于生成不同类型的线程池。Executors一共可以创建下面这四类线程池:

i). newFixedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程;

ii). newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待;

iii). newScheduledThreadPool 创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行

iv). newSingleThreadExecutor 创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行;

**ExecutorService：**它是线程池定义的一个接口，继承Executor。有两个实现类，分别为ThreadPoolExecutor,ScheduledThreadPoolExecutor。

**ExecutorService常用的几个方法：**

**i)execute(Runnable):**从父类继承过来的方法, 方法接收一个Runnable实例，并且异步的执行;

**ii)** **submit(Runnable)方法：**返回一个**Future**对象，通过返回的Future对象，我们可以检查提交的任务是否执行完毕;

**iii).submit(Callable)：**与submit(Callable)类似，也会返回一个Future对象，但是除此之外，submit(Callable)接收的是一个Callable的实现，Callable接口中的call()方法有一个返回值，可以返回任务的执行结果，而Runnable接口中的run()方法是void的，没有返回值;([submit与execute的区别见问题17](#_17).submit与execute的区别))

**iv).invokeAny(...)：**方法接收的是一个Callable的集合，执行这个方法不会返回Future，但是会返回所有Callable任务中其中一个任务的执行结果。这个方法也无法保证返回的是哪个任务的执行结果，反正是其中的某一个,只要有一个任务执行完成(没有抛出异常),线程池会终止其他未完成的任务; 如果提交的任务列表中没有能够正常完成的任务,那么最终会抛出异常; invokeAny和任务提交的顺序无关,只是返回最早执行完的任务的结果。

**v).invokeAll(...)：**与 invokeAny(...)类似也是接收一个Callable集合，但是invokeAll执行之后会返回一个Future的List，其中对应着每个Callable任务执行后的Future对象;

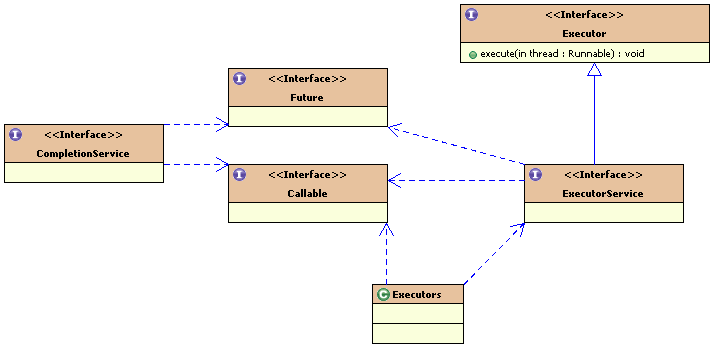
invokeAll是一个阻塞方法，会等待任务列表中的所有任务都执行完成。不管任务是正常完成，还是异常终止，Future.isDone()始终返回true。通过Future.isCanceled()可以判断任务是否在执行的过程中被取消。通过Future.get()可以获取任务的返回结果，或者是任务在执行中抛出的异常; invokeAll(Collection<? extends Callable<T>> tasks,long timeout, TimeUnit unit), 给定的超时期满，还没有完成的任务会被取消，即Future.isCancelled()返回true；在超时期之前，无论是正常完成还是异常终止的任务，Future.isCancelled()返回false; 一旦ExecutorService.invokeAll()方法产生了异常，线程池中还没有完成的任务会被取消执行.

**AbstractExecutorService:** AbstractExecutorService是一个抽象类，它实现了ExecutorService接口; 存在的目的是为ExecutorService中的函数接口提供了默认实现.

**ThreadPoolExecutor:** "线程池",它继承于AbstractExecutorService抽象类;

**ScheduledExecutorService:** ScheduledExecutorService是一个接口，它继承于于ExecutorService。它相当于提供了"延时"和"周期执行"功能的ExecutorService, 提供了相应的函数接口，可以安排任务在给定的延迟后执行，也可以让任务周期的执行;

**ScheduledThreadPoolExecutor:** ScheduledThreadPoolExecutor继承于ThreadPoolExecutor，并且实现了ScheduledExecutorService接口。它相当于提供了"延时"和"周期执行"功能的ScheduledExecutorService.



**CompletionService:** 使用ExecutorService类的时候，我们常维护一个list保存submit的callable task所返回的Future对象。然后在主线程中遍历这个list并调用Future的get()方法取到Task的返回值; 其实除了使用ExecutorService外，还可通过CompletionService包装ExecutorService，然后调用其take()方法去取Future对象.

**CompletionService与ExecutorService的区别:**  ExecutorService中从list中遍历的每个Future对象并不一定处于完成状态，这时调用get()方法就会被阻塞住，如果系统是设计成每个线程完成后就能根据其结果继续做后面的事，这样对于处于list后面的但是先完成的线程就会增加了额外的等待时间;  而CompletionService的实现是维护一个保存Future对象的BlockingQueue。只有当这个Future对象状态是结束的时候，才会加入到这个Queue中，take()方法其实就是**Producer-Consumer**中的Consumer。它会从Queue中取出Future对象，如果Queue是空的，就会阻塞在那里，直到有完成的Future对象加入到Queue中。所以，**先完成的必定先被取出**。这样就减少了不必要的等待时间.

**Callable，Future返回结果:**  Future<V>代表一个异步执行的操作，通过get()方法可以获得操作的结果，如果异步操作还没有完成，则，get()会使当前线程阻塞。FutureTask<V>实现了Future<V>和Runable<V>。Callable代表一个有返回值的操作。

**26).线程的中断, 处理不可中断的阻塞(p121), 处理超时任务,停止线程**

在java库中，许多可阻塞的方法都是通过提前返回或者抛出InterruptedException来响应中断请求的，从而使开发人员更容易构建出能响应取消请求的任务。然而并非所有的可阻塞方法或者阻塞机制都能响应中断；如果一个线程由于执行同步的Socket I/O或者等待获得内置锁而阻塞，那么中断请求只能设置线程的中断状态，除此之外没有其他任何作用.对于那些由于执行不可中断操作而被阻塞的线程,可以使用类似于中断的手段来停止这些线程.

i).Java .io包中的同步Socket I/O, 在服务器应用程序中,最常见的I/O形式就是对套接字进行读取和写入,虽然InputStream和OutputStream中的read和write等方法都不会响应中断,但是通过关闭底层的套接字,可以使得由于执行read或write等方法二被阻塞的线程抛出一个SocketEexeption;

ii)Java.io包中的同步I/O,当中断一个正在InterruptibleChannel上等待的线程,将抛出ClosedByInterruptException并关闭链路;

iii)Selector的异步I/O,如果一个线程在调用Selector.select方法时阻塞了,那么调用close或者wakeup方法会是线程抛出CloseSelectorException并提前返回.

iv).获取某个锁,如果一个线程由于等待某个内置锁而阻塞,那么将无法响应中断,因为线程认为它肯定会获得锁,所以将不会理会中断请求,但是在Lock类中提供了lockInterruptibly方法,该方法允许在等待一个锁的同时仍能响应中断.

**超时任务的处理:**

i)串行超时任务处理: 循环中判断条件为时间

ii).利用wait,notify实现异步超时处理, 让主线程在启动任务线程之后进行带超时参数的wait操作，如果任务线程超时，则wait不再等待，wait返回后主动中断任务线程；如果在超时时间内任务线程执行完毕，则通过notify方法通知主线程，这样主线程的wait方法也可以返回.

iii).利用Future实现超时处理: Future的get(long timeout, TimeUnit unit)将会在指定的时间内去获得执行结果，如果操作还没执行完，就会抛出TimeoutException，在捕获这个超时异常时就可以取消耗时任务的执行(通过Future.cancel(true))

**停止线程:**

i). 使用violate boolean变量来标识线程是否停止

ii). 停止线程时，需要调用停止线程的interrupt()方法，因为线程有可能在wait()或sleep(), 提高停止线程的即时性.

iii). 对于blocking IO的处理，尽量使用InterruptibleChannel来代替blocking IO.

**27).线程池的饱和策略(p144)**

**什么是饱和策略:** Java线程池会将提交的任务先置于工作队列中，在从工作队列中获取(阻塞队列直接由生产者提交给工作线程)。那么工作队列就有两种实现策略：无界队列和有界队列。无界队列不存在饱和的问题，但是其问题是当请求持续高负载的话，任务会无脑的加入工作队列，那么很可能导致内存等资源溢出或者耗尽。而有界队列不会带来高负载导致的内存耗尽的问题，但是有引发工作队列已满情况下，新提交的任务如何管理的难题，这就是线程池工作队列饱和策略要解决的问题.

Abort策略：默认策略，该异常可由调用者捕获, 当工作队列满了的时候，直接抛出了异常，而且JVM一直不退出, 新任务提交时直接抛出未检查的异常RejectedExecutionException,调用者捕获这个异常,然后根据需求写出自己的处理代码;

CallerRuns策略：为调节机制，既不抛弃任务也不抛出异常，而是将某些任务回退到调用者,从而降低新任务的流量。不会在线程池的线程中执行新的任务，而是在调用exector的线程中运行新的任务;

Discard策略：新提交的任务被抛弃;

Discard-Oldest: 抛弃最旧的策略, 会抛弃下一个将被执行的任务,然后尝试重新提交新的任务,如果工作队列是一个优先级队列,那么抛弃最旧的任务会首先抛弃优先级最高的任务,因此最好不要讲抛弃最旧的饱和策略和优先级队列放在一起使用;

**28). Amdahl定律**

Amdahl定律描述的是：在增加计算资源的情况下，程序在理论上能够实现最高加速比，这个值取决于程序中可并行组件与串行组件所占的比重。假定F是必须被串行执行的部分(F为串行占的比例)，那么根据Amdahl定律，在包含N个处理器的机器中，最高的加速比为: S=1/(F+(1-F)/N)

**29).线程引入的开销:上下文切换, 内存同步,阻塞**

**多线程引入的开销:**

**i). 设计更复杂:** 多线程程序在访问共享数据的时候往往需要我们很小心的处理,否则就会出现难以发现的BUG，一般地，多线程程序往往比单线程程序设计会更加复杂（尽管有些单线程处理程序可能比多线程程序要复杂），而且**错误很难重现**（因为线程调度的无序性，某些bug的出现**依赖于某种特定的线程执行时序**）;

**ii).上下文切换的开销:** 线程是由CPU进行调度的，CPU的一个时间片内只执行一个线程上下文内的线程，当CPU由执行线程A切换到执行线程B的过程中会发生一些列的操作，这些操作主要有”**保存线程A的执行现场**“然后”**载入线程B的执行现场**”，这个过程称之为“**上下文切换（context switch）**”,这个**上下文切换过程并不廉价**，如果没有必要，应该尽量减少上下文切换的发生. 引起上下文切换的主要原因:a). 时间片用完，CPU正常调度下一个任务;b).被其他优先级更高的任务抢占;c).执行任务碰到IO阻塞,调度器挂起当前任务,切换到其他任务;d).用户代码主动挂起当前任务,让出CPU时间;e).多任务抢占资源,没有抢到资源被挂起;f).硬件中断.

iii).增加更多的资源消耗: 除了**CPU执行上下文切换的消耗**以外，线程的执行还将有其他一些资源的消耗，比如:**内存同步的开销**（线程需要一些内存在维持线程本地栈,每个线程都有本地独立的栈用以存储线程专用数据），**上下文切换的开销**（前面已经讲过），**线程创建和消亡的开销**，以及**调度的开销**（占用操作系统的一些资源来管理和协调线程）.

**30).减小锁的竞争: 缩小锁的范围, 减小锁的粒度, 锁分段, 避免热点域, 替代独占锁(比如： ReadWriteLock是一种读多写少的加锁机制p198)**

**缩小锁的范围:** 降低发生竞争可能性的一种有效方式就是尽可能缩短锁的持有时间。例如，可以将一些与锁无关的代码移除同步代码块，尤其是那些开销比较大的曹组，以及可能被阻塞的操作，例如I/O操作。

**减小锁的粒度:** 另一种减小锁的持有时间的方式是降低线程请求锁的频率（从而减小发生竞争的可能性）。这可以通过锁分解和锁分段技术来实现，在这些技术中将采用多个相互独立的锁来保护独立的状态变量，从而改变这些变量在之前由单个锁来保护的情况。这些技术能减小锁操作的粒度，并能实现更高的可伸缩性，然而，使用的锁越多，那么发生死锁的风险也越高.

**锁分段:** 在某些情况下，可以将锁分解技术进一步扩展为对一组独立对象上的锁进行分解，这种情况被称为锁分段。例如，在ConcurrentHashMap的实现中使用了一个包含16个锁的数组，每个锁保护所有散列桶的1/16，其中第N个散列桶由第（N mod 16）个锁来保护。假设散列函数具有合理的分布性，并且关键字能够均匀分布，那么这大约能把对于锁的请求减少到原来的1/16，正是这项技术使得ConcurrentHashMap能够支持多达16个并发的写入器。（要使得拥有大量处理器的系统在高访问量的情况下实现更高的并发性，还可以进一步增加锁的数量，但仅当你能证明并发写入线程的竞争足够激烈并需要突破这个限制时，才能将锁分段的数量超过默认的16个。）; 锁分段的一个劣势在于：与采用单个锁来实现独占访问相比，要获取多个锁来实现独占访问将更加困难并且开销更高。通常，在执行一个操作时最多只需获取一个锁，但在某些情况下需要加锁整个容器，例如当ConcurrentHashMap需要扩展映射范围，以及重新计算键值的散列值要分布到更大的桶集合中时，就需要获取分段锁集合中的所有锁.

**避免热点域:** 锁分解和锁分段技术都能提高可伸缩性，因为它们都能使不同的线程在不同的数据（或者同一个数据的不同部分）上操作，而不会相互干扰。如果程序采用锁分段或分解技术，那么一定要表现出在锁上的竞争频率高于在锁保护的数据上发生竞争的频率。如果一个锁保护两个独立变量X和Y，并且线程A想要访问X，而线程B想要访问Y（这类似于在ServerStatus中，一个线程调用addUser，而另一个线程调用addQuery），那么这两个线程不会在任何数据上发生竞争，即使它们会在同一个锁上发生竞争。当每个操作都请求多个变量时，锁的粒度将很难降低。这是在性能与可伸缩性之间相互制衡的另一个方面，一些常见的优化措施，例如将一些反复计算的结果缓存起来，都会引入一些”热点域“，而这些热点域往往会限制可伸缩性.

当实现HashMap时，你需要考虑如何在size方法中计算Map中的元素数量。最简单的方法就是，在每次调用时都统计一次元素的数量。一种常见的优化措施是，在插入和移除元素时更新一个计数器，虽然这在put和remove等方法中略微增加了一些开销，以确保计数器是最新的值，但这把size方法的开销从O(n)降低到O(1)。   
在单线程或者采用完全同步的实现中，使用一个独立的计算器能很好地提高类似size和isEmpty这些方法的执行速度，但却导致更难以提升实现的可伸缩性，因为每个修改map的操作都需要更新这个共享的计数器。即使使用锁分段技术来实现散列链，那么在对计数器的访问进行同步时，也会重新导致在使用独占锁时存在的可伸缩性问题。一个看似性能优化的措施——缓存size操作的结果，已经变成了一个可伸缩性问题。在这种情况下，计数器也被称为热点域，因为每个导致元素数量发生变化的操作都需要访问它。   
为了避免这个问题，ConcurrentHashMap中的size将对每个分段进行枚举并将每个分段中的元素数量相加，而不是维护一个全局计数。为了避免枚举每个元素，ConcurrentHashMap为每个分段都维护一个独立的计数，并通过每个分段的锁来维护这个值。

**代替独占锁:** 第三种降低竞争锁的影响的技术就是放弃使用独占锁，从而有助于使用一种友好并发的方式来管理共享状态。例如，使用并发容器、读-写锁、不可变对象以及原子变量; ReadWriteLock实现了一种在多个读取操作以及单个写入操作情况下的加锁规则：如果多个读取操作都不会修改共享资源，那么这些读取操作可以同时访问该共享资源，但在执行写入操作时必须以独占方式来获取锁。对于读取操作占多数的数据结构，ReadWriteLock能够提供比独占锁更高的并发性。而对于只读的数据结构，其中包含的不变性可以完全不需要加锁操作。

**31).CAS操作, ABA问题, AQS**

**CAS(Compare And Swap)操作的过程**: 涉及到三个参数, 主存中的V值, 被所有线程共享, 线程从主存中读取的V值放到当前线程的临时变量区(栈帧)中, 需要更新(写入主存)的新值B, 在写入之前需要进行CAS操作, 比较当前线程中临时变量A和当前主存中的V是否相等, 如果相等则把新值B写入主存, 如果不相等, 则说明在该线程操作的期间有其他线程对主存中的V值进行了更新, 测试需要重新操作, 即需要重新从主存中读取V值到当前线程的临时变量A, 然后更新为B, 在更新写入主存前仍然进行CAS操作, 如果满足V与A相等, 就更新B, 如果不等, 继续CAS操作.

CAS操作例如: 有线程T1, 线程T2, 主存中的变量V=0

i)线程T1,T2都需要对V进行+1操作, 首先T1,T2获取主存中V的值为0, T1把V赋给T的临时变量区A1 = 0, T2把V的值赋给T2的临时变量A2 = 0;

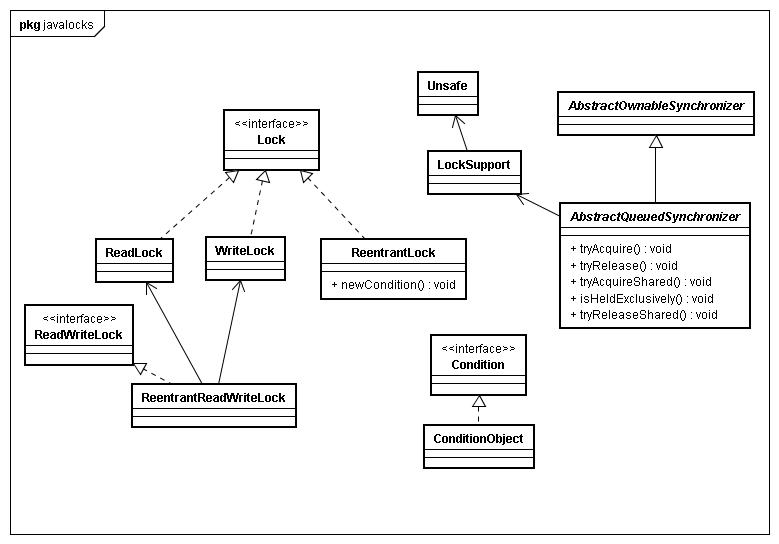
ii).T1对临时变量A1进行+1操作, 变为1, 然后T1需要进行CAS操作, 将A1与主存中的V进行比较, 如果主存中V与T1中的临时变量A1相等, 则直接将T1中的A1更新后的值1直接写入主存, 如果不相等, 则继续上述操作;

iii).线程T2此时临时变量中的A2值为0, 进行加1操作后, 进行CAS操作, 比较A2与主存中的V, V=1, 而A2 = 0, 二者不相等, 所以要继续进行上述操作, 重新从主存中读取V=1赋给A2, 然后进行加1操作,后为2, 然后在写入主存前进行CAS操作, 如果满足A2与主存的V相等, 则直接把更新后的值写入主存为2.

**ABA问题:** 如果一开始线程T2读取到的V值A2,然后进行加1操作后,写入主存前进行CAS操作,发现主存中的V值与A2相等, 此时并不能说明主存中的变量没有被修改过, 有可能是线程T1更新后又重新更新到原始值, 这就是ABA问题.用互斥同步可以解决并发的正确性问题.

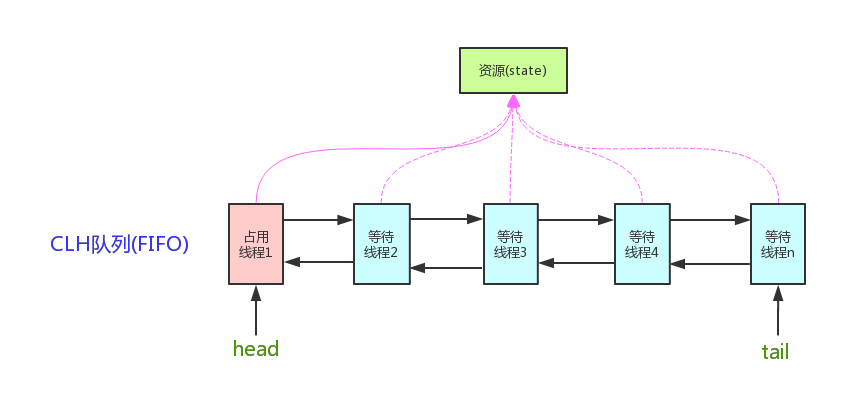
ABA问题的解决办法: 在变量前面追加版本号：每次变量更新就把版本号加1，则A-B-A就变成1A-2B-3A。例如在AtomicMarkableReference，AtomicStampedReference。他们在实现compareAndSet指令的时候除了要比较当对象的前值和预期值以外，还要比较当前（操作的）戳值和预期（操作的）戳值，当全部相同时，compareAndSet方法才能成功。每次更新成功，戳值都会发生变化，戳值的设置是由编程人员自己控制的。

**AQS(AbstractQueuedSynchronizer同步器): 抽象的队列式的同步器，AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch**结构如下图:



AQS考虑如下需求:

独占锁和共享锁两种机制; 线程阻塞后,如果需要取消,需要支持中断.



它维护了一个volatile int state（代表共享资源）和一个FIFO线程等待队列（多线程争用资源被阻塞时会进入此队列）。这里volatile是核心关键词，具体volatile的语义，在此不述。state的访问方式有三种:

* getState()
* setState()
* compareAndSetState()

　　AQS定义两种资源共享方式：Exclusive（独占，只有一个线程能执行，如ReentrantLock）和Share（共享，多个线程可同时执行，如Semaphore/CountDownLatch）。

　　不同的自定义同步器争用共享资源的方式也不同。**自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可**，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

* isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
* tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
* tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
* tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
* tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。

　　以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，**释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念**。但要注意，获取多少次就要释放多么次，这样才能保证state是能回到零态的。

　　再以CountDownLatch以例，任务分为N个子线程去执行，state也初始化为N（注意N要与线程个数一致）。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS减1。等到所有子线程都执行完后(即state=0)，会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后余动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，如ReentrantReadWriteLock

**下面分析独占锁和共享锁对资源的获取与释放:**

**独占锁:**

**acquire源码分析:**

**i)acquire(int):**

public final void acquire(int arg) {

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt();

}

此方法是独占模式下线程获取共享资源的顶层入口。如果获取到资源，线程直接返回，否则进入等待队列，直到获取到资源为止，且整个过程忽略中断的影响。这也正是lock()的语义，当然不仅仅只限于lock()。获取到资源后，线程就可以去执行其临界区代码了,执行流程如下:

a)tryAcquire()尝试直接去获取资源，如果成功则直接返回；

b)addWaiter()将该线程加入等待队列的尾部，并标记为独占模式；

c)acquireQueued()使线程在等待队列中获取资源，一直获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false。

d)如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上。

**ii)tryAcquire(int)**

protected boolean tryAcquire(int arg) {

throw new UnsupportedOperationException();

}

此方法尝试去获取独占资源。如果获取成功，则直接返回true，否则直接返回false; AQS这里只定义了一个接口，具体资源的获取交由自定义同步器去实现了（通过state的get/set/CAS）！！！至于能不能重入，能不能加塞，那就看具体的自定义同步器怎么去设计了.当然，自定义同步器在进行资源访问时要考虑线程安全的影响。

**iii)addWaiter(Node):** 此方法用于将当前线程加入到等待队列的队尾，并返回当前线程所在的结点.

**iv). enq(Node):** 此方法用于将node加入队尾

**v). acquireQueued(Node, int):** 通过tryAcquire()和addWaiter()，该线程获取资源失败，已经被放入等待队列尾部了, **进入等待状态休息，直到其他线程彻底释放资源后唤醒自己，自己再拿到资源，然后就可以去干自己想干的事了**。

**vi). shouldParkAfterFailedAcquire(Node, Node):** 此方法主要用于检查状态，看看自己是否真的可以去休息了（进入waiting状态）.

**vii). parkAndCheckInterrupt():**此方法就是让线程去休息，真正进入等待状态, park()会让当前线程进入waiting状态。在此状态下，有两种途径可以唤醒该线程：1）被unpark()；2）被interrupt().

**Release源码分析:**

public final boolean release(int arg) {

if (tryRelease(arg)) {

Node h = head;//找到头结点

if (h != null && h.waitStatus != 0)

unparkSuccessor(h);//唤醒等待队列里的下一个线程

return true;

}

return false;

}

此方法是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。这也正是unlock()的语义，当然不仅仅只限于unlock().

**共享锁:**

**acquireShared(int)源码分析:**

public final void acquireShared(int arg) {

if (tryAcquireShared(arg) < 0)

doAcquireShared(arg);

}

此方法是共享模式下线程获取共享资源的顶层入口。它会获取指定量的资源，获取成功则直接返回，获取失败则进入等待队列，直到获取到资源为止，整个过程忽略中断. AQS已经把其返回值的语义定义好了：负值代表获取失败；0代表获取成功，但没有剩余资源；正数表示获取成功，还有剩余资源，其他线程还可以去获取.

**releaseShared源码分析:**

public final boolean releaseShared(int arg) {

if (tryReleaseShared(arg)) {//尝试释放资源

doReleaseShared();//唤醒后继结点

return true;

}

return false;

}

此方法是共享模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果成功释放且允许唤醒等待线程，它会唤醒等待队列里的其他线程来获取资源。

**Mutex（互斥锁）:** Mutex是一个不可重入的互斥锁实现。锁资源（AQS里的state）只有两种状态：0表示未锁定，1表示锁定.

**32).线程与进程的区别,并发与并行的区别**

**进程**：是并发执行的程序在执行过程中分配和管理资源的基本单位;

**线程：**一个服务器通常需要接收大量且不确定数量用户的并发请求，为每一个请求都创建一个进程显然是行不通的，——无论是从系统资源开销方面或是响应用户请求的效率方面来看。因此，操作系统中线程的概念便被引进了。线程，是进程的一部分。线程有时又被称为轻权进程或轻量级进程，也是 CPU 调度的一个基本单位。线程之间共用一个进程的内存空存空间;

通常在一个进程中可以包含若干个线程，它们可以利用进程所拥有的资源。（线程之间共用一个进程的内存空存空间）在引入线程的操作系统中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位。由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统内多个程序间并发执行的程度.

**线程与进程的区别：**

a).地址空间和其它资源：进程间相互独立，同一进程的各线程间共享。某进程内的线程在其它进程不可见。  
b).通信：进程间通信IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。  
c).调度和切换：线程上下文切换比进程上下文切换要快得多。  
d).在多线程OS中，进程不是一个可执行的实体。

e).进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些.

f). 线程的改变只代表了 CPU 执行过程的改变，而没有发生进程所拥有的资源变化;

g) . 计算机内的软硬件资源的分配与线程无关，线程只能共享它所属进程的资源;

h). 进程拥有一个完整的虚拟地址空间，不依赖于线程而独立存在；反之，线程是进程的一部分，没有自己的地址空间，与进程内的其他线程一起共享分配给该进程的所有资源.

i). 同一进程中的两段代码不能够同时执行，除非引入线程。 线程是属于进程的，当进程退出时该进程所产生的线程都会被强制退出并清除.

**并发：**在[操作系统](http://baike.baidu.com/view/880.htm)中，是指一个时间段中有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个[处理机](http://baike.baidu.com/view/2107226.htm)上运行，但任一个时刻点上只有一个程序在处理机上运行。

**并行：**在单处理器中多道程序设计系统中，进程被交替执行，表现出一种并发的外部特种；在多处理器系统中，进程不仅可以交替执行，而且可以重叠执行。在多处理器上的程序才可实现并行处理。从而可知，并行是针对多处理器而言的。并行是同时发生的多个并发事件，具有并发的含义，但并发不一定并行，也亦是说并发事件之间不一定要同一时刻发生。

**互斥:** 进程间相互排斥的使用临界资源的现象，就叫互斥.

**同步:** 进程之间的关系不是相互排斥临界资源的关系，而是相互依赖的关系。进一步的说明：就是前一个进程的输出作为后一个进程的输入，当第一个进程没有输出时第二个进程必须等待。具有同步关系的一组并发进程相互发送的信息称为消息或事件.

**异步:** 异步和同步是相对的，同步就是顺序执行，执行完一个再执行下一个，需要等待、协调运行。异步就是彼此独立,在等待某事件的过程中继续做自己的事，不需要等待这一事件完成后再工作。线程就是实现异步的一个方式。异步是让调用方法的主线程不需要同步等待另一线程的完成，从而可以让主线程干其它的事情。

**并行与并发的区别:** 在单CPU系统中，系统调度在某一时刻只能让一个线程运行，虽然这种调试机制有多种形式(大多数是时间片轮巡为主)，但无论如何，要通过不断切换需要运行的线程让其运行的方式就叫并发(concurrent)。而在多CPU系统中，可以让两个以上的线程同时运行，这种可以同时让两个以上线程同时运行的方式叫做并行(parallel)。

i) 并发通常指提高运行在单处理器上的程序的性能；

ii). 并发是有状态的，“具有可论证的确定性，但是实际上具有不可确定性”；

iii). "并发"在微观上不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行，从宏观外来看，好像是这些进程都在执行;

iv). 使用多个线程可以帮助我们在单个处理系统中实现更高的吞吐量，如果一个程序是单线程的，这个处理器在等待一个同步I/O操作完成的时候，他仍然是空闲的。在多线程系统中，当一个线程等待I/O的同时，其他的线程也可以执行.

**异步与多线程的区别:**  异步和多线程并不是一个同等关系,异步是最终目的,多线程只是我们实现异步的一种手段。异步是当一个调用请求发送给被调用者,而调用者不用等待其结果的返回而可以做其它的事情。实现异步可以采用多线程技术或则交给另外的进程来处理.

**异步与同步的区别:** 在io等待的时候，同步不会切走，浪费了时间.

**多线程与异步操作的异同:**

a) 异步的本质: DMA就是直接内存访问的意思，也就是说，拥有DMA功能的硬件在和内存进行数据交换的时候可以不消耗CPU资源。只要CPU在发起数据传输时发送一个指令，硬件就开始自己和内存交换数据，在传输完成之后硬件会触发一个中断来通知操作完成。这些无须消耗CPU时间的I/O操作正是异步操作的硬件基础。

b).线程的本质: 线程不是一个计算机硬件的功能，而是操作系统提供的一种逻辑功能，线程本质上是进程中一段并发运行的代码，所以线程需要操作系统投入CPU资源来运行和调度。

c). 因为异步操作无须额外的线程负担，并且使用回调的方式进行处理，在设计良好的情况下，处理函数可以不必使用共享变量（即使无法完全不用，最起码可以减少共享变量的数量），减少了死锁的可能。当然异步操作也并非完美无暇。编写异步操作的复杂程度较高，程序主要使用回调方式进行处理，与普通人的思维方式有些初入，而且难以调试。

d). 多线程的优点很明显，线程中的处理程序依然是顺序执行，符合普通人的思维习惯，所以编程简单。但是多线程的缺点也同样明显，线程的使用（滥用）会给系统带来上下文切换的额外负担。并且线程间的共享变量可能造成死锁的出现。

**多线程和异步适用的范围:** 当需要执行I/O操作时，使用异步操作比使用线程+同步I/O操作更合适。I/O操作不仅包括了直接的文件、网络的读写，还包括数据库操作、Web Service、HttpRequest以及.Net Remoting等跨进程的调用。而线程的适用范围则是那种需要长时间CPU运算的场合，例如耗时较长的图形处理和算法执行。但是往往由于使用线程编程的简单和符合习惯，所以很多朋友往往会使用线程来执行耗时较长的I/O操作。这样在只有少数几个并发操作的时候还无伤大雅，如果需要处理大量的并发操作时就不合适了。

**33).用户线程与守护线程的区别**

所谓守护线程，是指在程序运行的时候在后台提供一种通用服务的线程，比如垃圾回收线程就是一个很称职的守护者，并且这种线程并不属于程序中不可或缺的部分。因此，当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程。反过来说，只要任何非守护线程还在运行，程序就不会终止。

用户线程和守护线程两者几乎没有区别，唯一的不同之处就在于虚拟机的离开：如果用户线程已经全部退出运行了，只剩下守护线程存在了，虚拟机也就退出了。因为没有了被守护者，守护线程也就没有工作可做了，也就没有继续运行程序的必要了。

如果要把一个线程设置为守护线程，则必须在它启动前，调用setDaemon(boolean on) 方法进行设置.

**34).线程调度**

JVM规范中规定每个线程都有优先级，且优先级越高越优先执行，但优先级高并不代表能独自占用执行时间片，可能是优先级高得到越多的执行时间片，反之，优先级低的分到的执行时间少但不会分配不到执行时间。JVM的规范没有严格地给调度策略定义，正是因为面对众多不同调度策略，JVM要封装所有细节提供一个统一的策略不太现实，于是给了一个不严谨但足够统一的定义。Java使用的线程调度是抢占式调度，在JVM中体现为让可运行池中优先级高的线程拥有CPU使用权，如果可运行池中线程优先级一样则随机选择线程，但要注意的是实际上一个绝对时间点只有一个线程在运行（这里是相对于一个CPU来说，如果你的机器是多核的还是可能多个线程同时运行的），直到此线程进入非可运行状态或另一个具有更高优先级的线程进入可运行线程池，才会使之让出CPU的使用权，更高优先级的线程抢占了优先级低的线程的CPU。

Java中线程会按优先级分配CPU时间片运行，那么线程什么时候放弃CPU的使用权？可以归类成三种情况：

1. )当前运行线程主动放弃CPU，JVM暂时放弃CPU操作（基于时间片轮转调度的JVM操作系统不会让线程永久放弃CPU，或者说放弃本次时间片的执行权），例如调用yield()方法。
2. )当前运行线程因为某些原因进入阻塞状态，例如阻塞在I/O上。
3. )当前运行线程结束，即运行完run()方法里面的任务。

Java的线程的调度机制都由JVM实现，假如有若干条线程，你想让某些线程拥有更长的执行时间，或某些线程分配少点执行时间，这时就涉及“线程优先级”，Java把线程优先级分成10个级别，线程被创建时如果没有明确声明则使用默认优先级，JVM将根据每个线程的优先级分配执行时间的概率。有三个常量Thread.MIN\_PRIORITY、Thread.NORM\_PRIORITY、Thread.MAX\_PRIORITY分别表示最小优先级值（1）、默认优先级值（5）、最大优先级值（10）。不过线程优先级并不靠谱,原因是Java的线程时通过映射到系统的原生线程上来实现的,所以线程调度最终还是取决于操作系统(某些系统中优先级会被系统自行改变,例如Windows系统中会有优先级推进器,当系统发现某个线程经常工作时,就可能会越过优先级为它分配执行时间).

由于JVM的实现以宿主操作系统为基础，所以Java优先级值与各种不同操作系统的原生线程优先级必然存在某种映射关系，这样才足以封装所有操作系统的优先级提供统一优先级语义。例如1-10优先级值在linux可能要与0-99优先级值进行映射，而windows系统则有7个优先级要映射。

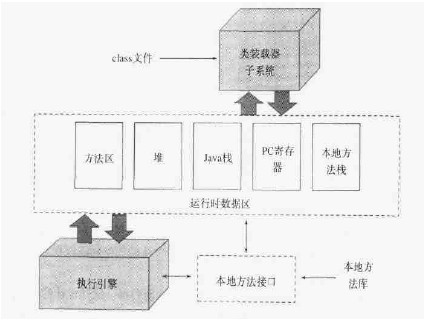
**线程的调度策略决定**

上层多线程运行机制，JVM的线程调度器实现了抢占式调度，每条线程执行的时间由它分配管理，它将按照线程优先级的建议对线程执行的时间进行分配，优先级越高，可能得到CPU的时间则越长。

内存：

**1).堆栈,方法区,直接内存的区别,堆的结构,永久代,老年代,新生代的区别**

**JVM结构图:**



**类加载器（ClassLoader）**:在JVM启动时或者在类运行时将需要的class加载到JVM中;

**字节码执行引擎：**负责执行class文件中包含的字节码指令；

**本地方法接口：主要是调用C或C++实现的本地方法及返回结果;**

**内存区（也叫运行时数据区）：是在JVM运行的时候操作所分配的内存区。运行时内存区主要可以划分为5个区域:**

**i).方法区:** 用于存储类结构信息的地方，包括**常量池、静态变量、构造函数等**。虽然JVM规范把方法区描述为堆的一个逻辑部分， 但它却有个别名non-heap（非堆），所以不要搞混淆了;

**ii).堆区:** 存储**java实例或者对象**的地方,这块是GC的主要区域。方法区和堆是被所有java线程共享的;

**iii)**.**栈(Stack)**：java栈总是和线程关联在一起，每当创建一个线程时，JVM就会**为这个线程创建一个对应的java栈**。在这个java栈中又会包含多个栈帧，**每运行一个方法就创建一个栈帧**，用于存储局部变量表、操作栈、方法返回值等。每一个方法从调用直至执行完成的过程，就对应一个栈帧在java栈中入栈到出栈的过程。

堆的大小由新生代和老年代组成;

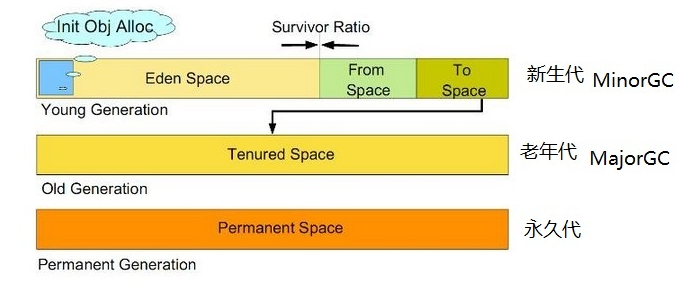
**iv).程序计数器(PC Register)**：用于保存当前**线程执行的内存地址**。由于JVM程序是多线程执行的（线程轮流切换），所以为了保证线程切换回来后，还能恢复到原先状态，就需要一个独立的**计数器，记录之前中断的地方**，程序计数器是线程私有的;

**v).本地方法栈:** 和java栈的作用差不多，只不过是**为JVM使用到的native方法服务的.**

**方法区,堆区,栈区,程序计数器,本地方法栈的区别:**

  java栈、程序计数器、本地方法栈都是线程私有的，线程生就生，线程灭就灭，栈中的栈帧随着方法的结束也会撤销，内存自然就跟着回收了。所以这几个区域的内存分配与回收是自动的。java堆和方法区则不一样，我们只有在程序运行期间才知道会创建哪些对象，所以这部分内存的分配和回收都是动态的，垃圾回收也是针对这一部分。

**新生代,老年代,永久代:**



**新生代**:主要是用来存放新生的对象。一般占据堆的1/3空间。由于频繁创建对象，所以新生代会频繁触发MinorGC进行垃圾回收。

**新生代分为三个区:**

Eden区: Java新对象的出生地（如果新创建的对象占用内存很大，则直接分配到老年代）。当Eden区内存不够的时候就会触发MinorGC，对新生代区进行一次垃圾回收。

FromSurvivor区: 上一次GC的幸存者，作为这一次GC的被扫描者.

ToSurvivor区: 保留了一次MinorGC过程中的幸存者.

**MinorGC的过程**：MinorGC采用复制算法。首先，把Eden和FromSurvivor区域中存活的对象复制到ToSurvivor区域（如果有对象的年龄以及达到了老年的标准，则赋值到老年代区），同时把这些对象的年龄+1（如果ToSurvivor不够位置了就放到老年区; 当对象的年龄达到某个值时(默认是 15 岁，可以通过参数 -XX:MaxTenuringThreshold 来设定)，这些对象就会成为老年代）；然后，清空Eden和FromSurvivor中的对象；最后，ToSurvivor和FromSurvivor互换，原ToSurvivor成为下一次GC时的FromSurvivor区。JVM 每次只会使用 Eden 和其中的一块 Survivor 区域来为对象服务，所以无论什么时候，总是有一块 Survivor 区域是空闲着的,Eden和Survivor的比例为8:1。因此，新生代实际可用的内存空间为 9/10 ( 即90% )的新生代空间.

**老年代:** 主要存放应用程序中生命周期长的内存对象.

老年代的对象比较稳定，所以MajorGC不会频繁执行。在进行MajorGC前一般都先进行了一次MinorGC，使得有新生代的对象晋身入老年代，导致空间不够用时才触发。当无法找到足够大的连续空间分配给新创建的较大对象时也会提前触发一次MajorGC进行垃圾回收腾出空间。

    MajorGC采用标记—清除算法：首先扫描一次所有老年代，标记出存活的对象，然后回收没有标记的对象。MajorGC的耗时比较长，因为要扫描再回收。MajorGC会产生内存碎片，为了减少内存损耗，我们一般需要进行合并或者标记出来方便下次直接分配。

     当老年代也满了装不下的时候，就会抛出OOM（Out of Memory）异常。

**永久代:** 指内存的永久保存区域，主要存放Class(静态文件,Java类,方法等)和Meta（元数据）的信息,Class在被加载的时候被放入永久区域. 它和和存放实例的区域不同,GC不会在主程序运行期对永久区域进行清理。所以这也导致了永久代的区域会随着加载的Class的增多而胀满，最终抛出OOM异常。在Java8中，永久代已经被移除，被一个称为“元数据区”（元空间）的区域所取代, 元空间的本质和永久代类似，都是对JVM规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制。类的元数据放入 native memory, 字符串池和类的静态变量放入java堆中. 这样可以加载多少类的元数据就不再由MaxPermSize控制, 而由系统的实际可用空间来控制.

**永久代中为什么被元空间代替:**

i)为了解决永久代的OOM问题，元数据和class对象存在永久代中，容易出现性能问题和内存溢出。

ii)类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出（因为堆空间有限，此消彼长）。

iii)永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。

iv)Oracle 可能会将HotSpot 与 JRockit 合二为一。

**2).调控线程的堆栈大小**

 查看堆的大小(单位字节):Runtime.getRuntime().maxMemory();

**堆内存配置:**

-Xms 初始堆内存 默认物理内存1/64,也是最小分配堆内存。当空余堆内存小于40%时，会增加到-Xms的最大限制

-Xmx 最大堆内存分配 默认物理内存1/4，当空余堆内存大于70%时，会减小到-Xms的最小限制。

-Xmx128m：设置JVM最大可用内存为128M

-Xms128m：设置JVM最小内存为128m

**非堆内存的配置:**

-XX:PermSize 非堆内存的初始值，默认物理内存的1/64 ，也是最小非堆内存。  
-XX:MaxPermSize 非堆内存最大值，默认物理内存的1/4.

-XX:MaxPermSize=16m：设置持久代大小为16m.

XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代.

-XX:NewSize 年轻代大小

-XX:MaxNewSize 年轻代最大值

-XX:NewRatio 年老代与年轻代比值

内存监控工具:jconsole, jmap

**3).JVM可达性分析的本质**

通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链相连 （用图论的话来说，就是从GC Roots到这个对象不可达）时，则证明此对象是不可用的。如图所示，对象object 5、object 6、object 7虽然互相有关联，但是它们到GC Roots是不可达的，所以它们将会被判定为是可回收的对象。



**4).?实例构造器和类构造器**

**5).内存溢出, 堆溢出, 栈溢出异常, 内存泄漏与监控**

**内存溢出(OOM)的场景:**

当老年代也满了装不下的时候，就会抛出OOM（Out of Memory）异常.

永久代的区域会随着加载的Class的增多而胀满，最终抛出OOM异常.

**内存溢出(OOM)的定位:**

1. **堆内存溢出:** Java创建的对象一般都是分配在堆中，如果是由于过期对象没能回收（内存泄漏）或者对象过多导致放不下（内存溢出），一般报错” OutOfMemoryError: Java heap space”. 解决这部分的异常，重点是通过内存映像分析工具分析堆的转储快照，确定异常是由于内存泄漏还是内存溢出导致的。 如果是内存泄漏导致的，则进一步查看泄漏对象到GCRoots的引用链，观察泄漏对象是通过怎样的路径与GCRoots相关联并导致垃圾回收器无法回收的;如果是内存溢出导致的，则检测堆的大小参数（Xmx、Xms）看看能否再调大，检测是否有某些对象生命周期过长。
2. **方法区溢出:** 方法区主要存放类的信息、静态变量、常量池等，当常量池溢出或者不停地有类动态创建并加载时，方法区也能产生OOM报错信息”OutOfMemoryError:PermGen space”.
3. **栈溢出(虚拟机栈和本地方法栈):** JVM在执行方法时就会创建方法栈，方法的递归、调用等使得其他方法不停地入栈，其他方法执行完毕就会弹出栈帧。当一个方法栈的深度大于JVM所允许的深度时就会报StackOverFlowError；一般，出现StackOverFlowError时就要检查代码是否有无穷递归的情况出现了. 栈空间扩展时没有足够的内存则报OutOfMemory。
4. **本地直接内存溢出:** 直接内存可以通过 -XX：MaxDirectMemorySize指定。如果本地直接内存溢出，我们可以发现堆转储快照中无明显异常指示，并且快照文件很小，而程序中又使用了NIO等技术，则可以检查是否直接内存溢出了.

**监控工具:** jconsole(jvm进程的实时情况,GUI), jmap(内存占用情况查), jstack(进程所包含线程情况查询), jstat(可以实时监测系统资源占用与jvm运行情况)

**6).虚拟机类加载机制**



**虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型**，这就是虚拟机的类加载机制. 上面这七个过程，除了解析这个过程外，其余过程必须按部就班地执行，即顺序是确定的，而解析过程不一定，在某些情况下可以在初始化阶段之后再执行，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）

java虚拟机规范中，并没有规定类加载过程中的第一个阶段（即加载阶段）的执行时机，但是对于初始化阶段，虚拟机规范中严格规定了“有且只有”下面5种情况下必须立即对类进行初始化（而这时，加载、验证、准备自然需要在此之前开始）：

（i）遇到new、getstatic、putstatic、invokestatic这四条指令时，必须触发其初始化。这四条指令最常见的场景是：使用new关键字实例化对象、读取或设置一个类的静态字段（被final修饰、已经在编译期把结果放入常量池的静态字段除外，即常量除外）、调用一个类的静态方法的时候；

（ii）进行反射调用的时候；

（iii）初始化一个类的时候，如果其父类还没有初始化，则需要先触发其父类的初始化；

（iv）当虚拟机启动时，需要先初始化那个包含main方法的要执行的主类；

（v）当使用JDK1.7的动态语言支持时，如果一java.lang.invoke.MethodHandle实例最后的解析结果为REF\_getStatic 、REF\_putStatic、REF\_invokeStatic的方法句柄，句柄对应的类会被初始化.

**7).重写与重载的本质**

**java文件的编译过程中不存在传统编译的连接过程**，一切方法调用在class文件中存放的只是符号引用，而不是方法在实际运行时内存布局中的入口地址(对于重写静态检查,动态调用;重载是静态调用)

重载: 对重载方法的调用主要看静态类型，静态类型是什么类型，就调用什么类型的参数方法.

静态类型与实际类型，方法接收者

1 Human man = new Man();

2 man.foo();

上面这条语句中，man的静态类型为Human，实际类型为Man。所谓方法接收者，就是指将要执行foo()方法的所有者（在多态中，有可能是父类Human的对象，也可能是子类Man的对象）.

重载只会发生在编译期，即编译器时jvm可以通过静态类型确定符号引用所对应的直接引用.

**虚拟机（准确说是编译器）在重载时是通过参数的静态类型而不是实际类型作为判定依据的。**   
对于字面量类型，编译器会自动进行类型转换。转换的顺序为：   
char-int-long-float-double-Character-Serializable-Object   
转换成Character是因为发生了自动装箱，转换成Serializable是因为Character实现了Serializable接口

重写: 对重写方法的调用主要看实际类型。实际类型如果实现了该方法则直接调用该方法，如果没有实现，则在继承关系中从低到高搜索有无实现.

重写发生在运行期，在运行时jvm会先判断对象的动态类型，而后根据对象的动态类型选择对应vtable，从而根据符号引用找到对应的直接引用.

在编译阶段，编译器只知道对象的静态类型，而不知道实际类型，所以在class文件中只能确定要调用父类的方法。但是在执行时却会判断对象的实际类型。如果实际类型实现这个方法，则直接调用，如果没有实现，则按照继承关系从下往上一次检索，只要检索到就调用，如果始终没有检索到，则抛异常

**java语言是一个静态多分派，动态单分派的语言:**

**方法的接收者和方法的参数统称为方法的宗量。** 根据分派基于宗量多少（接收者是一个宗量，参数是一个宗量），可以将分派分为单分派和多分派。单分派是指根据一个宗量就可以知道调用目标（即应该调用哪个方法），多分派需要根据多个宗量才能确定调用目标, 方法重载属于静态多分派, 方法重写属于动态单分派.

**8).语法糖，解语法糖, 泛型，类型擦除**

**语法糖**: 在计算机语言中添加某种语法，这种语法能使程序员更方便的使用语言开发程序，同时增强程序代码的可读性，避免出错的机会.

Java中的语法糖有**泛型、自动装箱、自动拆箱、foreach循环、变长参数、内部类、枚举类、断言（assert）等**.

**解语法糖:** 语法糖的存在主要是方便开发人员使用。但其实，Java 虚拟机并不支持这些语法糖，这些语法糖在编译阶段就会被还原成简单的基础语法结构，这个过程就是解语法糖

①.foreach循环: 说明对于数组，foreach是用普通for循环实现的; 在对有实现Iterable接口的对象采用foreach语法糖的话，编译器会将这个for关键字转化为对目标的迭代器使用;

②.自动装箱拆箱: 在自动装箱的时候，Java虚拟机会自动调用Integer的valueOf方法；在自动拆箱的时候，Java虚拟机会自动调用Integer的intValue方法。这就是自动拆箱和自动装箱的原理

③.泛型与类型擦除: 在程序源码中存在，在编译后的字节码文件中，就已经**被替换为原来的原生类型（Raw Type，也称为裸类型）**了，并且在相应的地方插入了强制转型代码;对于运行期的Java语言来说，ArrayList <Integer>与ArrayList<String>编译出来的代码是一样的，所以说泛型技术实际上是Java语言的一颗语法糖，Java语言中的泛型实现方法称为**类型擦除**，基于这种方法实现的泛型被称为伪泛型.

④.枚举类型: 在JVM字节码文件结构中，并没有“枚举”这个类型。Java的枚举类型，会在编译期被编译成一个普通子类, Java枚举编译后实际上是生成了一个类，该类继承了 java.lang.Enum<E>，并添加了一个返回枚举数组的values()方法和valueOf()方法, 所以Enum类型就是一个语法糖，编译器帮我们做了语法的解析和编译

⑤.内部类和闭包: **因为Java不支持多继承，支持实现多个接口。**但有时候会存在一些使用接口很难解决的问题，这个时候我们可以利用内部类提供的、可以继承多个具体的或者抽象的类的能力来解决这些程序设计问题。可以这样说，接口只是解决了部分问题，而内部类使得多重继承的解决方案变得更加完整.内部类之所以成为语法糖, 是因为它仅仅是一个编译时的概念，outer.java里面定义了一个内部类inner，一旦编译成功，就会生成两个完全不的.class文件了，分别是outer.class和outer$inner.class。**所以内部类的名字完全可以和它的外部类名字相同.**

**⑥.**变长参数: 所谓变长参数，就是方法可以接受长度不定确定的参数。变长参数特性是在JDK1.5中引入的，使用变长参数有两个条件，一是变长的那一部分参数具有相同的类型，二是变长参数必须位于方法参数列表的最后面。变长参数同样是Java中的语法糖，其内部实现是Java数组, 编译为class文件后如下，从中可以很明显的看出变长参数内部是通过数组实现的.

**9).编译期优化, 运行期优化**

**编译期优化**: 虚拟机把对性能的优化集中到了后端的即时编译器（JIT）中，这样可以让那些不是由javac编译器产生的class文件也同样能享受到编译器优化所带来的好处。但是javac做了许多针对编码过程的优化措施来改善程序员的编码风格和提高编码效率。许多新生的java语法特性，都是靠编译器的“语法糖”来实现，而不是依赖虚拟机的底层改进来支持。所以说，java中即时编译器在运行期间的优化过程对于程序的运行来说更重要，而前端编译器在编译期的优化过程对于程序编码来说关系更加密切。

**运行期优化:** 在部分商用虚拟机中，java程序最初是通过解释器进行解释执行的，当虚拟机发现某个方法或代码块运行特别频繁，就会把这些代码认定为“热点代码”，为了提高热点代码的执行效率，在运行时，虚拟机就会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化，完成这个任务的编译器称为即使编译器或JIT编译器。即时编译器并不是虚拟机必须的部分，但是即时编译器编译性能的好坏、代码优化程度的高低确是衡量一款商用虚拟机优秀与否的最关键的指标之一。众多主流的虚拟机都同时包含解释器和JIT编译器，解释器与JIT编译器各有优势：当程序需要迅速启动和执行时，解释器可以首先发挥作用，省去编译的时间，立即执行。当程序运行后，随着事件的推移，JIT编译器逐渐发挥作用，把越来越多的代码编译成本地代码之后，可以获取更高的执行效率

**10).JVM参数**

**11).JVM对CAS的支持**

**12).JVM关闭**

**13).虚拟机性能监控, 调优与故障处理工具(jps, jstack, jmap, jstat, jcomsole, jinfo, jhat,javap,btrace,TProfiler)**

**14).编译与反编译: javac, javap,jad,CRF**

**15).JVM定时内存回收:内存回收gc时都需要暂停当前应用程序**

**16).System.gc()和Runtime.gc()会做什么事情**

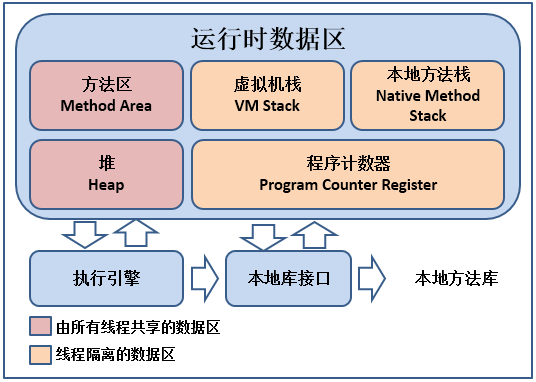
**17).finalize()方法的调用实际,析构函数的目的是什么**

**18).如果对象的引用被置为null,垃圾收集器是否立即释放对象占用的内存**

不会立即释放对象占用的内存。 如果对象的引用被置为null，只是断开了当前线程栈帧中对该对象的引用关系，而垃圾收集器是运行在后台的线程，只有当用户线程运行到安全点(safe point)或者安全区域才会扫描对象引用关系，扫描到对象没有被引用则会标记对象，这时候仍然不会立即释放该对象内存，因为有些对象是可恢复的（在 finalize方法中恢复引用 ）。只有确定了对象无法恢复引用的时候才会清除对象内存。

**19).happens-before**

##### JVM垃圾回收



(Java8把方法区移到堆中，称为元空间)

**哪些内存需要回收？**

从JVM区域结构看，可将这些区域划分为“静态内存”和“动态内存”两类。程序计数器、虚拟机栈、本地方法3个区域是“静态”的，因为这几个区域的内存分配和回收都具备确定性，都随着线程而生，随着线程而灭。但Java堆和方法区不一样，内存分配都存在不确定性，只有在程序处于运行期间才能知道会创建哪些对象，这部分内存和回收都是动态的，垃圾收集器所关注的是这部分内存。

**如何确定对象是否应该被回收？**

1).引用计数法：分配对象时给对象添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是没有再被使用了。客观地说，引用计数法（Reference Counting）的实现简单，判断效率也很高，但是在主流的Java虚拟机里面没有选用引用计数法来管理内存，其中最主要的原因是它很难解决对象之间相互循环引用的问题。

2). 可达性分析

在Java语言中，可作为GC Roots的对象包括下面几种：

* 虚拟机栈（栈帧中的本地变量表）中引用的对象。
* 方法区中类静态属性引用的对象。
* 方法区中常量引用的对象。
* 本地方法栈中JNI（即一般说的Native方法）引用的对象。

无论是引用计数法还是可达性分析算法，都用到了引用的概念，从JDK1.2开始，Java对引用的概念进行了扩充，将引用分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）四种，增强引用的适用性。

* 强引用就是指在程序代码之中普遍存在的，类似“Object obj = new Object()”这类的引用，只有强引用还存在，垃圾收集器永远不会回收掉被引用的对象。
* 软引用也是用来描述一些有用但并非必要的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。
* 弱引用也是用来描述非必要对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。但垃圾收集工作时，无论当前内存是否足够，都会回收掉弱引用关联的对象。
* 虚引用也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。因为finalize函数（后续会讲）被调用的不确定性，所以无法预知对象是否被回收，所在这里虚引用就起作用了。

即使在可达性分析算法中不可达的对象，也并非“非死不可”的，这个时候它们暂时处于缓刑阶段，要真正宣告一个对象死亡，至少要经历两次标记过程。第一次是在可达性分析后发现没有与GC Roots相连接的引用链并同时进行一次筛选，如果筛选出此对象没有必要执行finalize方法（当前对象没有覆盖finalize方法或此方法已经被执行过了，注意：finalize方法在整个对象生命周期只会被调用一次而已）。没有必要执行finalize方法的对象确认已经死亡。如果有必要执行finalize方法的对象则会将此对象放入一个叫做F-Queue的队列之中，并稍后有一个虚拟机自动建立的、低优先级的Finalizer线程去执行它。这也是对象最后的自救时刻，它可在finalize方法把自己（this）赋值给某个类变量或者对象的成员变量来自救不在这次垃圾回收掉。但在“虚引用”中已经提到，这个finalize方法的调用存在不确定性。因为虚拟机不承诺会等待整个F-Queue执行完，因为不排除有部分finalize方法存在死循环等极端情况造成F-Queue永远等待。所以说，finalize()方法是对象逃脱死亡的最后一个机会，但不一定被调用且在整个生命周期只会被调用一次，稍后GC将对F-Queue中的对象进行第二次小规模标记。如果没有成功逃脱，那这个对象真的会被回收了。

**方法区回收(永久代)：**

方法区（或者说HotSpot虚拟机中的永久代）的垃圾收集主要回收两部分内容：**废弃的常量和无用的类**。但永久代的回收效率远没有堆的回收效率高。判断一个常量是否“废弃常量”跟堆中判断对象非常相似，而要判断一个类是否是“无用的类”的条件相对苛刻：

* 该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例。
* 加载该类的ClassLoader已经被回收。
* 该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

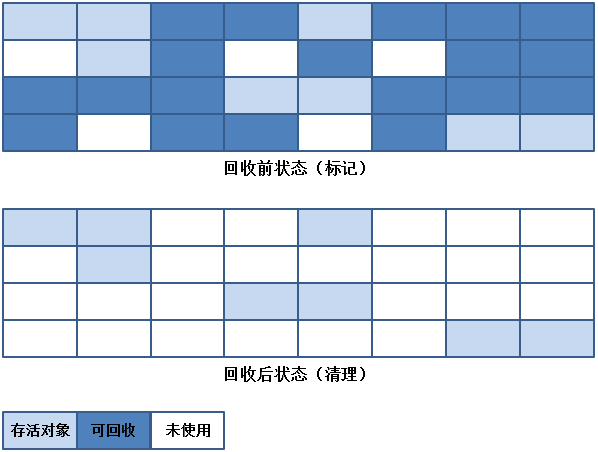
但是需要注意，就算满足以上三种条件也未必一定会回收。可能永久代的效果不理想，**在JDK8中，已经没有永久代的概念了，原先这块区域被放置到本地内存了**。

**回收时间：**

在主流的语言中（如Java）使用的都是可达性分析算法来判断对象的存活。而可达性分析算法是根据GC Roots的向下关联性来寻找没有被关联的对象。如果逐个区域检查是非常耗时的，**目前比较主流的Java虚拟机（例如HotSpot）使用的都是准确式GC**。HotSpot使用一种称为OopMap的数据结构来记录某个时刻栈和寄存器的哪些具体的地址是引用地址，从而可以快速找到GC roots来进行对象的标记操作。这里所说的“某个时刻”就是大名鼎鼎的Stop The World。虚拟机为了分析GC Roots这项工作必须在一个能确保一致性的快照中进行，这里的“一致性”的意思就是指在整个分析期间整个执行系统看起来就像被冻结在某个时间点上——这叫安全点。当然，程序执行时并非在所有地方都能停顿下来开始GC**，只有到达安全点时才能暂停**。安全点选址也有规定的，选定基本上是以程序“是否具有让程序长时间执行的特征”为标准进行选定的。这里的长时间执行的最明显特征是指令列复用，例如方法调用、循环跳转、异常跳转等。虚拟机为了能让所有线程都“跑”到安全点上停顿下来，设计了两个方案：抢先式中断和主动式中断。其中抢先式中断是虚拟机发生GC时，首先把所有线程全部中断，如果发生有线程中断的地方不在安全点上，就恢复线程，让它“跑”到安全点上。这种方式现在比较少用了。而主动式中断是虚拟机需要GC时仅仅简单的设置一个标志，各个线程执行到安全点时主动去轮询这个标志，发现中断标志为真时就自己中断挂起。安全点机制只保证了执行时的程序可以收到“通知”，但对于“不执行”的程序（如等待CPU时间的程序）如何是好？例如线程处于Sleep状态或者Blocked状态，这时线程无法响应JVM的中断强求。对于这种情况，JVM使用了**安全区域**（Safe Region）来解决。安全区域是指一段代码片段之中，引用关系不会发生变化的，在这个区域中任意地方开始GC都是安全的。例如线程执行到安全区域中的代码时，首先标识自己已经进入了安全区域。那么JVM发起GC时，就不用管那些已经被标自己为Safe Region状态的线程了。当这些线程要离开安全区域时，它同时要检查系统是否已经完成了根节点枚举（或者是整个GC过程）才能继续执行，否则必须等待收到可以安全离开安全区域的信号为止。

**如何回收--垃圾回收算法**

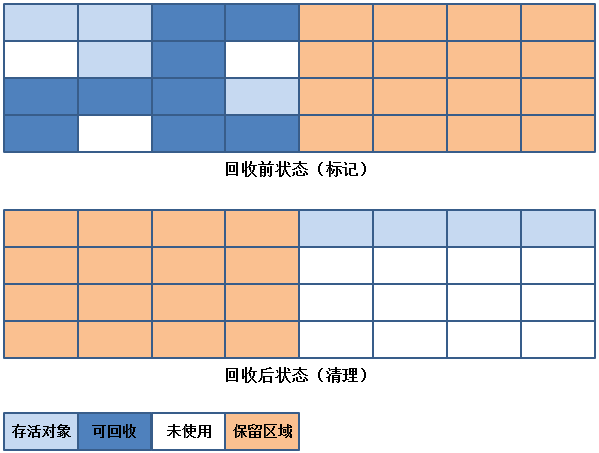
**标记-清除（Mark-Sweep）算法：**这是最基础的算法，就像它名字一样，算法分为“标记”和“清除”两个阶段：首先标记处所有需要回收的对象（如哪些内存需要回收所描述的对象），对标记完成后统一回收所有被标记的对象，如下图所示：



**缺点**：一个是效率问题，标记和清除两个过程的效率都不高；另一个是空间问题，标记清除后悔产生大量的不连续的内存碎片，可能会导致后续无法分配大对象而导致再一次触发垃圾收集动作。

**复制算法：**

为了针对标记-清除算法的不足，复制算法将可用内存容量划分为大小相等的两块，每次只使用一块。当一块的内存用完了，就将还存活的对象复制到另一块上面去。然后把已使用过的内存空间一次清理掉，弥补了标记清除算法中内存布局混乱的缺点,如下图所示：

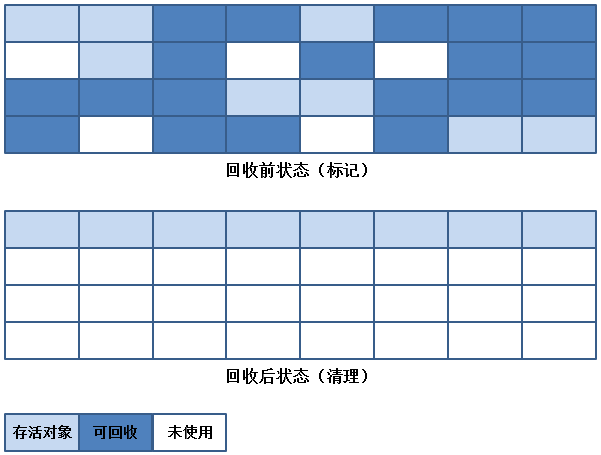


**缺点：**使用内存比原来缩小了一半;如果对象的存活率非常高,会导致大量对象被移动到保留区,移动对象后,相应的对象的内存地址也会发生变化,导致效率降低。

现在的商业虚拟机都采用这种收集算法来**回收新生代**，有企业分析的得出其实并不需求将内存按1:1的比例划分，因为新生代中的对象大部分都是“朝生夕死”的。所以，HotSpot虚拟机默认的Eden(使用空间)和Survivor(保留空间)的大小比例是8:1。一块Eden和两块Survivor，每次使用一块Eden和一块Survivor，也就是说只有10%是浪费的。如果另一块Survivor都无法存放上次垃圾回收的对象时，那这些对象将通过“担保机制”进入老年代了(新生代与老年代:)。

**标记-整理（Mark-Compact）算法：**

复制算法一般是对对象存活率较低的一种回收操作，但对于对象存活率较高的内存区域（老年代）来说，效果就不是那么理想了，标记-整理算法因此诞生了。标记-整理算法和标记-清除算法差不多，都是一开始对回收对象进行标记，但后续不是直接对对象清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存，弥补了标记-清除算法中内存区域分散的缺点以及复制算法中内存减半的缺点,如下图所示：



**缺点:**标记所有存活的对象以及整理所有存活对象的引用地址导致效率不高.

**分代收集算法：**

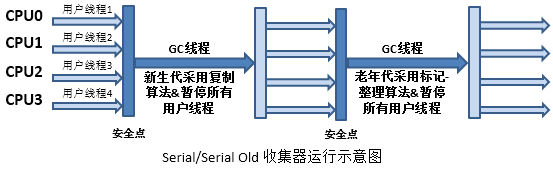
当前商业虚拟机的垃圾收集都采用“分代收集”（Generational Collection）算法，这种算法只是根据对象存活周期的不同将内存划分为几块，一般把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中, 每次垃圾收集时都发现有大批对象死去,只有少量存活,那就选用复制算法,只需要付出少量存活对象的复制成本就可以完成收集, 对于老年代中因为对象存活率较高,没有额外空间进行分配担保,就必须使用"标记-清理"或者"标记-整理"算法进行回收.

**方法的对比:** 效率上复制算法>标记-整理>标记-清除; 内存整齐度上复制算法>标记-整理>标记-清除; 内存利用率上标记整理>标记清除>复制算法.

**垃圾收集器: 垃圾回收的具体实现,需要使用到不同的回收算法**

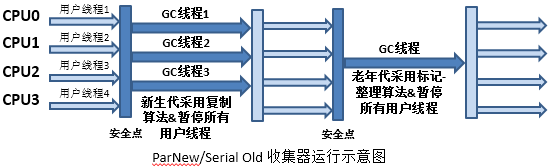
**Serial收器：**

Serial收集器是最基本、发展历史最悠久的收集器。这个收集器是一个单线程的收集器，当它工作时必须暂停其他线程的工作，也就是Stop The World。这显示是它的缺点， 这也是垃圾收集器一直努力的方向。当然，对于相比其它单线程收集器，Serial收集器简单而高效。对于桌面应用来说，分配的管理内存不会太多，停顿时间完全可以控制在几十毫秒最多一百毫秒以内。所以，Serial收集器对于运行在Client模式下的虚拟机来说是一个很好的选择。下图为Serial结合Serial Old收集器（后续介绍）的运行过程：



**ParNew收集器：**

ParNew收集器其实就是Serial收集器的多线程版本（多CPU下使用效果较好），下图为ParNew结合Serial Old收集器（后续介绍）的运行过程：



ParNew收集器对于Serial来说并没有太多的创新之处，但它却是许多运行在Server模式下的虚拟机中首选的新生代收集器，因为除了Serial收集器外，剩下只有它能与CMS收集器（后续介绍）配合工作了。所以，遗憾的是CMS作为老年代的收集器，却无法与JDK1.4中已经存在的新生代收集器Parallel Scavenge配合工作。

**Parallel Scavenge收集器：**

Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器，看上去跟ParNew差不多。但是Parallel Scavenge收集器与其他收集器不同在于CMS等收集器的关注点在于尽可能地缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量（Throughput）。所谓的吞吐量就是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，其中垃圾收集花费1分钟，那吞吐量就是99%。

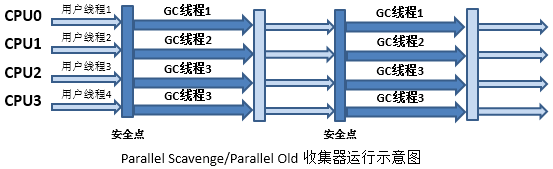
 Parallel Scavenge收集器分别可通过-XX:MaxGCPauseMillis参数控制最大垃圾收集停顿时间以及直接设置吞吐量大小的-XX:GCTimeRatio参数。Parallel Scavenge收集器还有一个-XX:UseAdaptiveSizePolicy开关参数，打开参数后就不需要手动指定新生代的大小、Eden与Survivor区的比例、晋升老年代对象年龄等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整相关参数。这种调节方式叫自适应的调节策略，也是Parallel Scavenge收集器与ParNew收集器的一个重要区别。

**Serial Old收集器：**

Serial Old是一个老年代收集器，它同样是一个单线程收集器，使用的是“标记-整理”算法。这个收集器的主要意义也是在于给Client模式下的虚拟机使用。如果在Server模式下，那么它主要还有两大用途：一种用途是在JDK1.5以及之前的版本中与Parallel Scavenge收集器搭配使用；另一种用途就是作为CMS收集器的后备方案，在并发收集发生ConCurrent Mode Failure时使用。

**Parallel Old收集器：**

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器是在JDK1.6中才开始提供的，在此之前，新生代的Parallel Scavenge收集器一直处于比较尴尬的状态。原因是新生代如果选择了Parallel Scavenge收集器，老年代除了Serial Old收集器别无选择（因为它无法与CMS配合使用）。在都CPU时代，由于Serial Old收集器在服务端性能上的“拖累”，使用了Parallel Scavenge收集器也未必能在整体应用上获得吞吐量最大化的效果。直到Parallel Old收集器的出现后，“吞吐量优先”收集器才有了比较名副其实的应用组合。在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器，如下图所示：

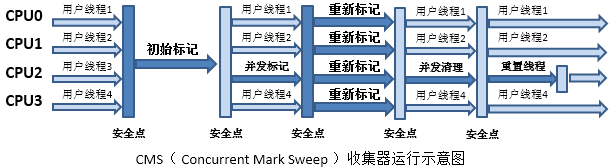


**CMS收集器：**

CMS（Concurrent Mark Sweep）收集器是一种以获得最短回收停顿时间为目标的收集器。目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。从名字上就可以看出，CMS收集器是基于“标记-清除”算法实现的。但它的实际运作过程对于前面几种收集器来说更复杂一些，整个过程分为4个步骤：

* 初始标记（CMS initial mark）
* 并发标记（CMS concurrent mark）
* 重新标记（CMS remark）
* 并发清除（CMS concurrent sweep）

其中，初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快；并发标记阶段就是进行GC Roots Tracing的过程；而重新标记阶段则是为了修正并发标记期间用用户程序继续运作而导致标记产生的那一部分对象的标记记录。这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记时间短。由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的（注意并发与并行的概念），如下图所示



CMS是一款优秀的收集器，但是还远达不到的完美程度，它有以下3个明显缺点：

* CMS收集器对CPU资源非常敏感。因为在并发阶段，它会占用了一部分线程（或者说CPU资源）而导致应用程序变慢，总吞吐量会降低。
* CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。由于CMS并发清理阶段用户线程还在运行着，伴随着程序运行自然就还会有新的垃圾不断产生，这部分垃圾出现在标记过程之后，CMS无法在当次收集中处理它们，只好留待在下一次GC时再清理掉，这一部分垃圾就称为“浮动垃圾”。
* 还有最后一点，CMS是一款基于“标记-清除”算法实现的收集器，这意味着收集结束时会有大量的空间碎片产生。为了解决这个问题，CMS收集器提供了一个-XX:+UseCMSCompactAtFullColletion开关参数（默认是开启的），用于在CMS收集器顶不住要进行Full GC时开启内存碎片合并整理过程，内存整理的过程是无法并发的，空间的碎片问题没有了，但停顿的时间不得不变长了。虚拟机设计者还提供了另外一个参数-XX:CMSFullGCsBeforeCompaction，这个参数是用于设置执行多少次不压缩的Full GC后，跟着来一次带压缩的（默认值为0，表示每次进入Full GC时都进行碎片整理）。

**G1收集器：**

G1（Garbage First）收集器是当今收集器技术发展的最前沿成果之一，从JDK 6u14中开始就有Early Acsess版本的G1收集器供开发人员实验、试用，由此开始G1收集器的 “Experimental” 状态持续了数年时间，直到JDK7u4，Sun公司才认为它达到足够成熟的商用程度，移除了“Experimental”的标识。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可以替换掉JDK1.5中发布的CMS收集器。其与其它收集器相比，G1具备如下特点：

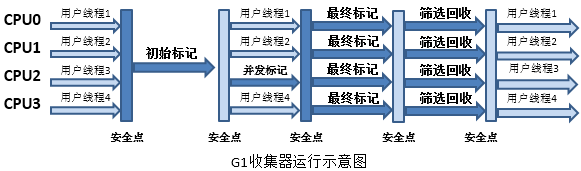
* 并行与并发：和CMS类似。
* 分代收集：分代概念在G1中依然得以保留。虽然G1可以不需要其它收集器配合就能独立管理整个GC堆，但它能够采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次GC的旧对象以获取更好的收集效果。也就是说G1可以自己管理新生代和老年代了。
* 空间整合：由于G1使用了独立区域（Region）概念，G1从整体来看是基于“标记-整理”算法实现收集，从局部（两个Region）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片。
* 可预测的停顿：这是G1相对于CMS的另一大优势，降低停顿时间是G1和CMS共同的关注点，但G1除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用这明确指定一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒。

与其它收集器相比，G1变化较大的是它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留了新生代和老年代的概念，但新生代和老年代不再是物理隔离的了它们都是一部分Region（不需要连续）的集合。同时，为了避免全堆扫描，G1使用了Remembered Set来管理相关的对象引用信息。当进行内存回收时，在GC根节点的枚举范围中加入Remembered Set即可保证不对全堆扫描也不会有遗漏了。

如果不计算维护Remembered Set的操作，G1收集器的运作大致可划分为以下几个步骤：

* 初始标记（Initial Making）
* 并发标记（Concurrent Marking）
* 最终标记（Final Marking）
* 筛选回收（Live Data Counting and Evacuation）

看上去跟CMS收集器的运作过程有几分相似，不过确实也这样。初始阶段仅仅只是标记一下GC Roots能直接关联到的对象，并且修改TAMS（Next Top Mark Start）的值，让下一阶段用户程序并发运行时，能在正确可以用的Region中创建新对象，这个阶段需要停顿线程，但耗时很短。并发标记阶段是从GC Roots开始对堆中对象进行可达性分析，找出存活对象，这一阶段耗时较长但能与用户线程并发运行。而最终标记阶段需要吧Remembered Set Logs的数据合并到Remembered Set中，这阶段需要停顿线程，但可并行执行。最后筛选回收阶段首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划，这一过程同样是需要停顿线程的，但Sun公司透露这个阶段其实也可以做到并发，但考虑到停顿线程将大幅度提高收集效率，所以选择停顿。下图为G1收集器运行示意图：



##### 3.  [BIO，NIO，AIO的区别](https://blog.csdn.net/u013068377/article/details/70312551)

BIO：线程发起IO请求，不管内核是否准备好IO操作，从发起请求起，线程一直阻塞，直到操作完成。

NIO(reactor模型)：线程发起IO请求，立即返回；内核在做好IO操作的准备之后，通过调用注册的回调函数通知线程做IO操作，线程开始阻塞，直到操作完成。

AIO(proactor模型)：线程发起IO请求，立即返回；内存做好IO操作的准备之后，做IO操作，直到操作完成或者失败，通过调用注册的回调函数通知线程做IO操作完成或者失败。如下图：

1)、BIO

在JDK1.4出来之前，我们建立网络连接的时候采用BIO模式，需要先在服务端启动一个ServerSocket，然后在客户端启动Socket来对服务端进行通信，默认情况下服务端需要对每个请求建立一堆线程等待请求，而客户端发送请求后，先咨询服务端是否有线程响应，如果没有则会一直等待或者遭到拒绝请求，如果有的话，客户端会线程会等待请求结束后才继续执行。

2)、NIO

    NIO本身是基于事件驱动思想来完成的，其主要想解决的是BIO的大并发问题： 在使用同步I/O的网络应用中，如果要同时处理多个客户端请求，或是在客户端要同时和多个服务器进行通讯，就必须使用多线程来处理。也就是说，将每一个客户端请求分配给一个线程来单独处理。这样做虽然可以达到我们的要求，但同时又会带来另外一个问题。由于每创建一个线程，就要为这个线程分配一定的内存空间（也叫工作存储器），而且[**操作系统**](http://lib.csdn.net/base/operatingsystem)本身也对线程的总数有一定的限制。如果客户端的请求过多，服务端程序可能会因为不堪重负而拒绝客户端的请求，甚至服务器可能会因此而瘫痪。

  NIO基于Reactor，当socket有流可读或可写入socket时，操作系统会相应的通知应用程序进行处理，应用再将流读取到缓冲区或写入操作系统。  也就是说，这个时候，已经不是一个连接就要对应一个处理线程了，而是有效的请求，对应一个线程，当连接没有数据时，是没有工作线程来处理的。

2)、BIO

   BIO与NIO一个比较重要的不同，是我们使用BIO的时候往往会引入多线程，每个连接一个单独的线程；而NIO则是使用单线程或者只使用少量的多线程，每个连接共用一个线程。相对于BIO的流，NIO抽象出了新的通道（Channel）作为输入输出的通道，并且提供了缓存（Buffer）的支持，在进行读操作时，需要使用Buffer分配空间，然后将数据从Channel中读入Buffer中，对于Channel的写操作，也需要现将数据写入Buffer，然后将Buffer写入Channel中。

      NIO的最重要的地方是当一个连接创建后，不需要对应一个线程，这个连接会被注册到多路复用器上面，所以所有的连接只需要一个线程就可以搞定，当这个线程中的多路复用器进行轮询的时候，发现连接上有请求的话，才开启一个线程进行处理，也就是一个请求一个线程模式。

      在NIO的处理方式中，当一个请求来的话，开启线程进行处理，可能会等待后端应用的资源(JDBC连接等)，其实这个线程就被阻塞了，当并发上来的话，还是会有BIO一样的问题。

　　HTTP/1.1出现后，有了Http长连接，这样除了超时和指明特定关闭的http header外，这个链接是一直打开的状态的，这样在NIO处理中可以进一步的进化，在后端资源中可以实现资源池或者队列，当请求来的话，开启的线程把请求和请求数据传送给后端资源池或者队列里面就返回，并且在全局的地方保持住这个现场(哪个连接的哪个请求等)，这样前面的线程还是可以去接受其他的请求，而后端的应用的处理只需要执行队列里面的就可以了，这样请求处理和后端应用是异步的.当后端处理完，到全局地方得到现场，产生响应，这个就实现了异步处理。

3)、AIO

     与NIO不同，当进行读写操作时，只须直接调用API的read或write方法即可。这两种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流传入read方法的缓冲区，并通知应用程序；对于写操作而言，当操作系统将write方法传递的流写入完毕时，操作系统主动通知应用程序。  即可以理解为，read/write方法都是异步的，完成后会主动调用回调函数。  在JDK1.7中，这部分内容被称作NIO.2，主要在[Java](http://lib.csdn.net/base/javase).nio.channels包下增加了下面四个异步通道：

AsynchronousSocketChannel

AsynchronousServerSocketChannel

AsynchronousFileChannel

AsynchronousDatagramChannel

其中的read/write方法，会返回一个带回调函数的对象，当执行完读取/写入操作后，直接调用回调函数。

BIO是一个连接一个线程。

**NIO是一个请求一个线程：**

Channel和IO中的Stream(流)是差不多一个等级的。只不过Stream是单向的，譬如：InputStream, OutputStream。而Channel是双向的，既可以用来进行读操作，又可以用来进行写操作，NIO中的Channel的主要实现有：FileChannel、DatagramChannel、SocketChannel、ServerSocketChannel；通过看名字就可以猜出个所以然来：分别可以对应文件IO、UDP和TCP（Server和Client）。

NIO中的关键Buffer实现有：ByteBuffer、CharBuffer、DoubleBuffer、 FloatBuffer、IntBuffer、 LongBuffer,、ShortBuffer，分别对应基本数据类型: byte、char、double、 float、int、 long、 short。当然NIO中还有MappedByteBuffer, HeapByteBuffer, DirectByteBuffer等。

Selector 是NIO相对于BIO实现多路复用的基础，Selector 运行单线程处理多个 Channel，如果你的应用打开了多个通道，但每个连接的流量都很低，使用 Selector 就会很方便。例如在一个聊天服务器中。要使用 Selector , 得向 Selector 注册 Channel，然后调用它的 select() 方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新的连接进来、数据接收等。

AIO是一个有效请求一个线程。

先来个例子理解一下概念，以银行取款为例：

* 同步 ： 自己亲自出马持银行卡到银行取钱（使用同步IO时，Java自己处理IO读写）；
* 异步 ： 委托一小弟拿银行卡到银行取钱，然后给你（使用异步IO时，Java将IO读写委托给OS处理，需要将数据缓冲区地址和大小传给OS(银行卡和密码)，OS需要支持异步IO操作API）；
* 阻塞 ： ATM排队取款，你只能等待（使用阻塞IO时，Java调用会一直阻塞到读写完成才返回）；
* 非阻塞 ： 柜台取款，取个号，然后坐在椅子上做其它事，等号广播会通知你办理，没到号你就不能去，你可以不断问大堂经理排到了没有，大堂经理如果说还没到你就不能去（使用非阻塞IO时，如果不能读写Java调用会马上返回，当IO事件分发器会通知可读写时再继续进行读写，不断循环直到读写完成）

Java对BIO、NIO、AIO的支持：

* [Java](http://lib.csdn.net/base/java)BIO ： 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。
* Java NIO ： 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。
* Java AIO(NIO.2) ： 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，

BIO、NIO、AIO适用场景分析:

* BIO方式适用于连接数目比较小且固定的[架构](http://lib.csdn.net/base/architecture)，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。
* NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。
* AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

另外，I/O属于底层操作，需要操作系统支持，并发也需要操作系统的支持，所以性能方面不同操作系统差异会比较明显。

在高性能的I/O设计中，有两个比较著名的模式Reactor和Proactor模式，其中Reactor模式用于同步I/O，而Proactor运用于异步I/O操作。

    在比较这两个模式之前，我们首先的搞明白几个概念，什么是阻塞和非阻塞，什么是同步和异步,同步和异步是针对应用程序和内核的交互而言的，同步指的是用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪，而异步是指用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知。而阻塞和非阻塞是针对于进程在访问数据的时候，根据IO操作的就绪状态来采取的不同方式，说白了是一种读取或者写入操作函数的实现方式，阻塞方式下读取或者写入函数将一直等待，而非阻塞方式下，读取或者写入函数会立即返回一个状态值。

 一般来说I/O模型可以分为：同步阻塞，同步非阻塞，异步阻塞，异步非阻塞IO

同步阻塞IO：在此种方式下，用户进程在发起一个IO操作以后，必须等待IO操作的完成，只有当真正完成了IO操作以后，用户进程才能运行。JAVA传统的IO模型属于此种方式！

同步非阻塞IO:在此种方式下，用户进程发起一个IO操作以后边可返回做其它事情，但是用户进程需要时不时的询问IO操作是否就绪，这就要求用户进程不停的去询问，从而引入不必要的CPU资源浪费。其中目前JAVA的NIO就属于同步非阻塞IO。

异步阻塞IO：此种方式下是指应用发起一个IO操作以后，不等待内核IO操作的完成，等内核完成IO操作以后会通知应用程序，这其实就是同步和异步最关键的区别，同步必须等待或者主动的去询问IO是否完成，那么为什么说是阻塞的呢？因为此时是通过select系统调用来完成的，而select函数本身的实现方式是阻塞的，而采用select函数有个好处就是它可以同时监听多个文件句柄，从而提高系统的并发性！

 异步非阻塞IO:在此种模式下，用户进程只需要发起一个IO操作然后立即返回，等IO操作真正的完成以后，应用程序会得到IO操作完成的通知，此时用户进程只需要对数据进行处理就好了，不需要进行实际的IO读写操作，因为真正的IO读取或者写入操作已经由内核完成了。

##### 4.JVM线程调度方式

JVM规范中规定每个线程都有优先级，且优先级越高越优先执行，但优先级高并不代表能独自占用执行时间片，可能是优先级高得到越多的执行时间片，反之，优先级低的分到的执行时间少但不会分配不到执行时间。JVM的规范没有严格地给调度策略定义，正是因为面对众多不同调度策略，JVM要封装所有细节提供一个统一的策略不太现实，于是给了一个不严谨但足够统一的定义。Java使用的线程调度是抢占式调度，在JVM中体现为让可运行池中优先级高的线程拥有CPU使用权，如果可运行池中线程优先级一样则随机选择线程，但要注意的是实际上一个绝对时间点只有一个线程在运行（这里是相对于一个CPU来说，如果你的机器是多核的还是可能多个线程同时运行的），直到此线程进入非可运行状态或另一个具有更高优先级的线程进入可运行线程池，才会使之让出CPU的使用权，更高优先级的线程抢占了优先级低的线程的CPU。

Java中线程会按优先级分配CPU时间片运行，那么线程什么时候放弃CPU的使用权？可以归类成三种情况：

1.)当前运行线程主动放弃CPU，JVM暂时放弃CPU操作（基于时间片轮转调度的JVM操作系统不会让线程永久放弃CPU，或者说放弃本次时间片的执行权），例如调用yield()方法。

2.)当前运行线程因为某些原因进入阻塞状态，例如阻塞在I/O上。

3.)当前运行线程结束，即运行完run()方法里面的任务。

Java的线程的调度机制都由JVM实现，假如有若干条线程，你想让某些线程拥有更长的执行时间，或某些线程分配少点执行时间，这时就涉及“线程优先级”，Java把线程优先级分成10个级别，线程被创建时如果没有明确声明则使用默认优先级，JVM将根据每个线程的优先级分配执行时间的概率。有三个常量Thread.MIN\_PRIORITY、Thread.NORM\_PRIORITY、Thread.MAX\_PRIORITY分别表示最小优先级值（1）、默认优先级值（5）、最大优先级值（10）。不过线程优先级并不靠谱,原因是Java的线程时通过映射到系统的原生线程上来实现的,所以线程调度最终还是取决于操作系统(某些系统中优先级会被系统自行改变,例如Windows系统中会有优先级推进器,当系统发现某个线程经常工作时,就可能会越过优先级为它分配执行时间).

由于JVM的实现以宿主操作系统为基础，所以Java优先级值与各种不同操作系统的原生线程优先级必然存在某种映射关系，这样才足以封装所有操作系统的优先级提供统一优先级语义。例如1-10优先级值在linux可能要与0-99优先级值进行映射，而windows系统则有7个优先级要映射。

**线程的调度策略决定**

上层多线程运行机制，JVM的线程调度器实现了抢占式调度，每条线程执行的时间由它分配管理，它将按照线程优先级的建议对线程执行的时间进行分配，优先级越高，可能得到CPU的时间则越长。