JAVA

[IO相关 6](#_Toc62919988)

[Reader/XXPutStream的区别： 7](#_Toc62919989)

[Lang 8](#_Toc62919990)

[反射相关： 8](#_Toc62919991)

[Reference 8](#_Toc62919992)

[基本数据类型包装类： 9](#_Toc62919993)

[RunTime相关 9](#_Toc62919994)

[ThreadGroup： 10](#_Toc62919995)

[ClassLoader解析 10](#_Toc62919996)

[Object解析 12](#_Toc62919997)

[Object常用工具方法：Objects 12](#_Toc62919998)

[网络相关： 13](#_Toc62919999)

[Nio： 13](#_Toc62920000)

[Concurrent:线程同步 14](#_Toc62920001)

[Atomic，原子操作： 14](#_Toc62920002)

[\*\*\*AQS锁：（共享锁，独占锁） 15](#_Toc62920003)

[AQS核心变量：state 15](#_Toc62920004)

[Condition类： 15](#_Toc62920005)

[集合： 18](#_Toc62920006)

[ArrayBlockingQueue原理： 18](#_Toc62920007)

[ConcurrentHashMap原理 19](#_Toc62920008)

[HashMap 19](#_Toc62920009)

[ConcurrentLinkedDeque原理 19](#_Toc62920010)

[ConcurrentSkipListMap原理 20](#_Toc62920011)

[SkipList跳表原理 20](#_Toc62920012)

[CopyOnWriteArrayListCopyOnWriteArraySet原理 24](#_Toc62920013)

[PriorityQueue原理 26](#_Toc62920014)

[二叉堆实现原理： 26](#_Toc62920015)

[Exchanger<V>实现原理： 28](#_Toc62920016)

[SynchronousQueue实现原理： 29](#_Toc62920017)

[ForkJoinPool（ForkJoinTask、ForkJoinWorkerThread）RecursiveAction、RecursiveTask原理 30](#_Toc62920018)

[Executors线程池 34](#_Toc62920019)

[ExecutorCompletionService原理： 34](#_Toc62920020)

[AbstractExecutorService原理： 35](#_Toc62920021)

[FutureTask<V> 结果任务 35](#_Toc62920022)

[ThreadLocalRandom 35](#_Toc62920023)

[\*\*\*ThreadPoolExecutor分析： 36](#_Toc62920024)

[Jar/zip目录解析： 38](#_Toc62920025)

[Jar文件读写： 38](#_Toc62920026)

[Zip文件读写： 38](#_Toc62920027)

[Java集合： 39](#_Toc62920028)

[ArrayDeque实现原理：（队列） 39](#_Toc62920029)

[ArrayList实现原理： 39](#_Toc62920030)

[Arrays 工具方法： 40](#_Toc62920031)

[BitSet：位图原理（适用场景：整数，无重复；） 41](#_Toc62920032)

[Calendar，Date ，GregorianCalendar（带时区）日期的使用 42](#_Toc62920033)

[Collections： 42](#_Toc62920034)

[EnumMap：枚举Map 43](#_Toc62920035)

[HashMap：原理 43](#_Toc62920036)

[HashTable：同步的hashMap 45](#_Toc62920037)

[IdentityHashMap： 45](#_Toc62920038)

[LinkedHashMap： 46](#_Toc62920039)

[LinkedHashSet 继承HashSet，没有其他修改点，和HashSet一致 47](#_Toc62920040)

[LinkedList：双向链表 47](#_Toc62920041)

[Vector、Stack： 48](#_Toc62920042)

[TreeMap、TreeSet红黑树原理：（有序，与跳表原理相比） 48](#_Toc62920043)

[Properties：读取配置文件默认属性 48](#_Toc62920044)

[Timer、TimerTask定时器原理 49](#_Toc62920045)

[javax.activation 51](#_Toc62920046)

[org.json ：Android Json相关 52](#_Toc62920047)

[Android XML解析：（dom,sax,pull） 54](#_Toc62920048)

[\*\*Android 自定义XML属性（xml文件预加载对象） 57](#_Toc62920049)

[Base64：基础编码解码方式 58](#_Toc62920050)

[javax 58](#_Toc62920051)

[Android 59](#_Toc62920052)

[AccessibilityService：辅助功能详解 59](#_Toc62920053)

[使用方式 59](#_Toc62920054)

[源码流程解析 61](#_Toc62920055)

[原理：观察者模式 63](#_Toc62920056)

[Accounts目录 64](#_Toc62920057)

[android.animation动画 65](#_Toc62920058)

[动画类 65](#_Toc62920059)

[layout布局动画(ViewGroup动画) 65](#_Toc62920060)

[动画加载方式 65](#_Toc62920061)

[AnimatorInflater.loadAnimator 65](#_Toc62920062)

[2. 获取定义属性 67](#_Toc62920063)

[3. 获取属性值 67](#_Toc62920064)

[4. 生成根标签对象并赋值 67](#_Toc62920065)

[Android.app 68](#_Toc62920066)

[JobService 68](#_Toc62920067)

[源码解析 69](#_Toc62920068)

[Android 系统Service加载与调用 70](#_Toc62920069)

[初始化方式 70](#_Toc62920070)

[ServiceManager解析 71](#_Toc62920071)

[App层获取系统服务Context.getSystemService原理 71](#_Toc62920072)

[Service的AIDL原理 73](#_Toc62920073)

[其他远程通信 75](#_Toc62920074)

[Android.app.Activity 77](#_Toc62920075)

[Context上下文对象 77](#_Toc62920076)

[ContextWrapper 77](#_Toc62920077)

[ContextThemeWrapper 78](#_Toc62920078)

[Activity拆解 78](#_Toc62920079)

JAVA类

# IO相关

|  |  |  |
| --- | --- | --- |
| InputStream | OutputStream |  |
| FilterInputStream | FilterOutputStream | InputStream/ OutputStream包装类 |
| BufferedInputStream | BufferedOutputStream | 使用byte[]作为缓存 |
| ByteArrayInputStream | ByteArrayOutputStream | 读取预置byte[]数组并返回给新byte[]，或者读取新char  写入预置byte[]大小，不够进行double扩容（使用ByteArrayOutputStream缺陷就是缓存会一直扩容，造成OOM） |
| FileInputStream | FileOutputStream | 文件读写 |
| DataInputStream | DataOutputStream | 对InputStream outPutStream包装，用于基本数据类型（String）读写，记录写入长度，最大Ingeter.MaxValue，超过将不准确 |
| ObjectInputStream | ObjectOutputStream | NA |
| PipedInputStream | PipedOutputStream | 连接2个stream，一个读，一个写，等待读写操作 |
|  | PrintStream | 向指定outputStream输出内容（基本数据类型/一般类型） |
| PushbackInputStream |  | 使用Byte[]缓存执行回退读取，其实就是pos-- = inputchar |
| SequenceInputStream |  | 使用一个有序数组排入inputStream，并依次读取 |

|  |  |  |
| --- | --- | --- |
| Reader | Writer |  |
| BufferedReader | BufferedWriter | 使用char[]作为缓存 |
| CharArrayReader | CharArrayWriter | 读取预置char[]并返回给新char []，或者读取新char |
| InputStreamReader | OutputStreamWriter | 使用StreamEncoder/ StreamDecoder编码读写 |
| FileReader | FileWriter | InputStreamReader/ OutputStreamWriter传入FileInputStream |
| LineNumberReader |  | 同BufferedReader，按行读取，可记录行数 |
| PipedReader | PipedWriter | 连接2个stream，一个读，一个写，等待读写操作，原理， Reader在没有数据会等待Writer写入，数据满也等待读取，Writer阻塞写入，之后writer可以写入 |
|  | PrintWriter | 向指定Writer输出内容（基本数据类型/一般类型） |
| PushbackReader |  | 使用char[]缓存执行回退读取，其实就是pos-- = inputchar |
| StringReader | StringWriter | 读取string内容，写入StringBuffer |

## Reader/XXPutStream的区别：

1. Reader与Stream功能类似，但是Reader是线程安全的，Stream不是线程安全的

File、RandomAccessFile

FileDescriptor

FileFilter、FilenameFilter

FilterReader、FilterWriter

StreamTokenizer（可用于统计/分割字符内容）

技术分析：

InputStream/outputStream 类：将内容（基本数据类型/String类型）进行byte[]转换输入输出（使用ByteArrayStream/FileStream作为目的地）

Buffer缓存技术，回退技术

使用数组缓存，回退读取

队列读取技术

使用有序数组

读写等待技术

Piped读写，一个读一个写，使用固定大小缓存，阻塞空读，满写

# Lang

## 反射相关：

JAVA 6 使用的Reflect API，在JAVA7 中是由MethodHandle更快的反射（简单易用）

MethodHandles + MethodType

Reflect API不再介绍

## Reference

SoftReference

软引用：内存不足的时候可以被释放，会加入一个ReferenceQueue中

WeakReference

弱引用：管理区域不再使用该变量内存的时候释放，会加入一个ReferenceQueue中

PhantomReference

虚引用：主要用于跟踪对象的垃圾回收机制，因为对象finalize方法是不安全而且是低效率的

子类：Cleaner 可用于追踪对象的死亡，执行一个runnable方法（隐藏类）

ReferenceQueue

对象回收队列

java finalize方法

1. java 中的finalize方法是由垃圾收集器调用的。如果一个程序调用exit方法退出应用，垃圾收集器不会执行。
2. 如果一个对象覆盖了finalize方法，那么这个对象将会放置在一个叫做F-Queue的队列之中。并在稍后由一个虚拟机自动建立的、低优先级的Finalizer线程去执行它。
3. 虚拟机会触发这个方法，但并不承诺会等待它运行结束，这样做的原因是，如果一个对象在finalier方法中执行缓慢，或者发生了死循环，导致整个内存系统崩溃。
4. finalize方法会逃脱死亡命运的最后一次机会。
5. finalize方法只被执行一次，如果finalize执行过，则不会执行第二次。

## 基本数据类型包装类：

Boolean：内部持有一个boolean类型value

Byte：内部持有一个byte类型value（1个字节）

使用ByteCache Byte [256]缓存，范围值Byte [-128 , 127]，用于快速查找返回值，valueOf方法的使用

Short：内部持有一个short类型value（2个字节）

使用ShortCache Short [256]缓存，范围值Short [-128 , 127]，用于快速查找返回值，valueOf方法的使用，比较同integer

Character：内部持有一个char类型value

使用CharacterCache缓存，范围Character [0,128]，128数组的对象，将int转为char

Double：内部持有一个double类型value

没有cache，占用8个字节

Enum：枚举类型，内部string name int value

Float：内部持有一个float类型value

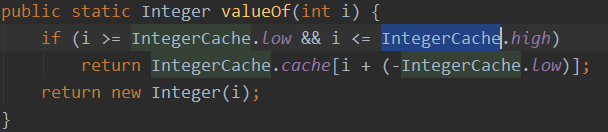
没有cache，占用4个字节

Integer：内部持有一个int类型value

使用IntegerCache缓存，默认大小Integer[-128~127]，同Byte缓存，但是可以启动VM虚拟机修改配置文件大小上限值，不超过int最大值

这里在定义Integer对象的时候，如果值是-128~127，那么会使用缓存对象，在比较对象的时候是相等的，如果不在这个区间之内，将使用新对象包装

自动拆箱使用的是valueOf方法：



Long：内部持有一个long类型value

使用Long Cache缓存，大小Long [-128~127]，对象比较同Integer

## RunTime相关

Runtime.getRuntime()

addShutdownHook(Thread ); 添加程序关闭的时候额外启动的线程

int availableProcessors()：CPU核心数

long freeMemory()：查询memory

long totalMemory()：查询memory

5. String,buffer,builder

对象，buffer使用synchrnized线程安全，builder线程不安全

Buffer与builder都是使用char[]存储内容，扩容策略，system.arrayCopy， size = current + input.length 长度

Buffer内部有一个string串用于调用toString初始化赋值

6. Thread,ThreadGroup,ThreadLocal解析

Thread线程：

Join：阻塞所在线程，等调用它的线程执行完毕，再向下执行

Sleep：阻塞所在线程，不让出CPU

Object.Wait：阻塞所在线程，让出CPU，等待唤醒

yield()：使当前线程从执行状态（运行状态）变为可执行态（就绪状态）。cpu会从众多的可执行态里选择，也就是说，当前也就是刚刚的那个线程还是有可能会被再次执行到的，并不是说一定会执行其他线程而该线程在下一次中不会执行到了（CPU让步）

## ThreadGroup：

线程组，表示一组线程的集合，可以对一批线程和线程组进行管理。可以把线程归属到某一个线程组中，线程组中可以有线程对象，也可以有线程组，组中还可以有线程，这样的组织结构有点类似于树的形式

1. 安全

同一个线程组的线程是可以相互修改对方的数据的。但如果在不同的线程组中，那么就不能“跨线程组”修改数据，可以从一定程度上保证数据安全。

2. 批量管理

可以批量管理线程或线程组对象，有效地对线程或线程组对象进行组织或控制。

默认线程组：取SecurityManager..getThreadGroup()其次parentThread.getThreadGroup()

ThreadLocal：

每个Thread中存在一个map用于存取线程临时值，不同线程有不同的数据

## ClassLoader解析

Java：使用findclass查找的是单个class文件，加载成byte[]通过defineClass获取class

动态加载执行反射

Android：在loadClass通过DexFile加载class

this.dexFile = DexFile.loadDex(this.apkDexPath, this.dexCachePath, 0); 加载apk,zip,jar里面的文件，生成DexFile结构体

通过dexFile.loadClass(name.replace('.', '/'), this); 加载之后的dexfile代理加载class类

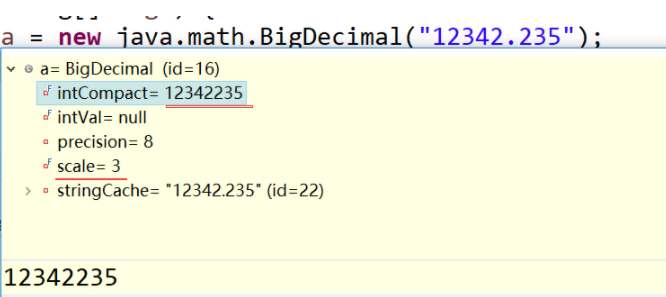
Dex列表类加载策略：热修复

Math：

BigDecimal精度计算

在需要确定精度的时候，小数点之后操作，float/double存在误差，需要将小数转换

数据结构：



IntCompact表示无小数点数值

stringCache将输入内容作为string串保存。

intVal 没有小数点的情况下有值，表示int类型

prcsision 表示输入内容占用字符个数

scale 表示小数点之后有多少位

算术运算：

先转换成long计算结果，intCompact整数计算，然后构造函数传入标志与整型数值，scale，percision

BigInteger 二进制补码操作，使用int[] msg表示正数源码字节数组

## Object解析

getClass()：获取Class<?> 类型

hashCode()：获取当前实例的hashCode码

equals()：当前对象是否相同，对比地址，需要重写

public boolean equals(Object obj) {  
 return (this == obj);  
}

clone()：对象克隆（必须实现cloneable接口）

protected Object clone() throws CloneNotSupportedException {  
 if (!(this instanceof Cloneable)) {  
 throw new CloneNotSupportedException("Class " + getClass().getName() +  
 " doesn't implement Cloneable");  
 }  
  
 return internalClone();  
}

wait,notify：等待唤醒，与synchronized连用，线程等待，释放CPU

finalize()：系统执行GC会导致object可能回调该方法，但是该方法不一定会被执行，参考虚引用Reference

### Object常用工具方法：Objects

public static boolean equals(Object a, Object b) {  
 return (a == b) || (a != null && a.equals(b));  
}

比较2个对象是否相同，equals比较

public static boolean deepEquals(Object a, Object b) {  
 if (a == b)  
 return true;  
 else if (a == null || b == null)  
 return false;  
 else  
 return Arrays.deepEquals0(a, b);  
}

比较2个对象（数组）地址是否相同，包括数组深度比较，使用==比较

public static int hash(Object... values) {  
 return Arrays.*hashCode*(values);  
}

返回数组的hashCode

public static <T> int compare(T a, T b, Comparator<? super T> c) {  
 return (a == b) ? 0 : c.compare(a, b);  
}

比较对象，返回int

## 网络相关：

自定义

UDP：DatagramSocket

TCP：Socket/ServerSocket

连接

URL：HttpURLConnection

使用的是HttpClient，继承NetWorkClient，来源sun.net包

实现原理：Socket通信

URL地址可以支持jar特定文件，File文件，或者Http地址

## Nio：

待完善

Text：

文本相关，格式化相关

Time：

时间相关

Util：

工具类

# Concurrent:线程同步

## Atomic，原子操作：

AtomicBoolean,Integer, Long，Reference<V>：

使用unSafe#objectFieldOffset方法原子性操作value值

IntegerArray,LongArray,ReferenceArray<E>：使用unSafe# arrayIndexScale方法原子性操作array数组值（其实就是操作数组里面元素，使用cas操作）

AtomicIntegerFieldUpdater，AtomicLongFieldUpdater，ReferenceFieldUpdater<T,V>对象域值操作，内部引用value值，使用unSafe#objectFieldOffset方法原子性操作value值

Long,Double Adder：使用自旋锁原理，执行加减操作

Locks，锁操作：

核心类：AbstractQueuedSynchronizer，（AbstractQueuedLongSynchronizer，同AQS，只不过内部state类型改为long类型，用于64位状态设置）

## \*\*\*AQS锁：（共享锁，独占锁）

原理解析：

通过双向链表的形式链式执行线程等待与唤醒

独占锁通过：acquire方法调用，子类实现tryAcquire方法，只有一个线程可以独占，其他线程阻塞，唤醒机制是唤醒链表下一个

共享锁通过acquireShared方法调用，子类实现tryAcquireShared方法，多个线程都可以执行，其他线程阻塞，唤醒机制是唤醒后继节点，存在多余无用的唤醒，继续锁定

### AQS核心变量：state

用于标志状态，判断是否可以锁，是否可以释放锁，如果释放锁成功，等待节点被尝试唤醒，唤醒之后即在被添加等待的node节点再次判断获取锁（tryAcquire/tryAcquireShared），如果成功，线程继续执行

常用方法：

setExclusiveOwnerThread getExclusiveOwnerThread 设置独占线程

内部类：

### Condition类：

与AQS连用，用于AQS独占锁之后传递给其他线程，并让AQS阻塞，condition.await等待。

public final void await() throws InterruptedException {  
 if (Thread.*interrupted*())  
 throw new InterruptedException();  
 Node node = addConditionWaiter();  
 int savedState = fullyRelease(node);  
 int interruptMode = 0;  
 while (!isOnSyncQueue(node)) {  
 LockSupport.*park*(this);  
 if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)  
 break;  
 }  
 if (acquireQueued(node, savedState) && interruptMode != *THROW\_IE*)  
 interruptMode = *REINTERRUPT*;  
 if (node.nextWaiter != null) // clean up if cancelled  
 unlinkCancelledWaiters();  
 if (interruptMode != 0)  
 reportInterruptAfterWait(interruptMode);  
}

fullyRelease(node)：在获取锁的线程该线程释放锁定，进入while等待，把该锁交给其他线程，其他线程可以执行lock获取独占锁，并执行signal解锁

!isOnSyncQueue ：等待直到外部线程调用signal调用，当前线程尝试acquireQueued调用tryAcquired尝试获取锁（独占锁），从条件队列进入正常的阻塞队列

对于condition解锁代码：（条件队列）

public final void signal() {  
 if (!isHeldExclusively())  
 throw new IllegalMonitorStateException();  
 Node first = firstWaiter;  
 if (first != null)  
 doSignal(first);  
}

当前独占才能执行解锁

也就是执行signal/signalAll解锁的时候必须要先获取锁，其他等待线程尝试acquireQueued，竞争线程

线程等待实现方式：LockSupport.park/unPark

ReentrantReadWriteLock：

内部拥有Sync实现ReadLock与WriteLock，在读锁的时候执行sync.acquireShared(1);共享锁，并增加state值，执行sync.releaseShared(1)解锁减少该state值

执行写锁的时候（独占），如果不存在读锁，那么可以写，否则等待

缺点：悲观锁，读多写少的情况下，写被一直阻塞

针对该问题，引入StampedLock锁，是一种乐观锁，该锁原理是在获取锁的时候记录一个值，并且在获取写锁的时候判断该记录值，如果没有变化则可以写，否则有变化则需要重新获取并执行写

共享锁与独占锁：

共享锁返回int值用于判断是否>0，如果>0那么会继续唤醒后面一个共享节点，直到该节点等待解锁传递

独占锁只会等待，等待独占释放锁才会唤醒后继一个独占节点

CountDownLatch原理，继承AQS实现共享锁，释放锁执行自选保证state正确，在state == 0释放完成（不受线程限制，多个等待可以被同时唤醒，当state == 0）

protected int tryAcquireShared(int acquires) {  
 return (getState() == 0) ? 1 : -1;  
}

多个线程获取释放之后，如果state = 0等待线程都会被唤醒

Semaphore原理（有序唤醒，持有信号量，如果不足，阻塞后续thread）

信号量锁，继承AQS实现共享锁，释放锁执行自选保证state正确，当state（信号量值不足时），需要acquireshared线程被等待，当其他线程释放信号量一定值的时候，信号量被增加，满足需要小于state信号量值得线程被唤醒，拥有公平/非公平锁hasQueuedPredecessors()判断

protected int tryAcquireShared(int acquires) {  
 for (;;) {  
 if (hasQueuedPredecessors())  
 return -1;  
 int available = getState();  
 int remaining = available - acquires;  
 if (remaining < 0 ||  
 compareAndSetState(available, remaining))  
 return remaining;  
 }  
}

（如果公平锁，那么不足信号的时候将等待）

CyclicBarrier原理，适用ReEnterLock#Condition实现等待，当执行等待次数达到设定值的时候执行一个runnable方法

# 集合：

## ArrayBlockingQueue原理：

使用ReEnterLock#Condition生成notFull,notEmpty用于在添加/移除数据的时候signal通知，await等待（定长，超加等待，空取等待）

public void put(E e) throws InterruptedException {  
 *checkNotNull*(e);  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 while (count == items.length)  
 notFull.await();  
 enqueue(e);  
 } finally {  
 lock.unlock();  
 }  
}

public E take() throws InterruptedException {  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {  
 while (count == 0)  
 notEmpty.await();  
 return dequeue();  
 } finally {  
 lock.unlock();  
 }  
}

## ConcurrentHashMap原理

适用synchronized 锁hash数组中查找到的node头节点（线程同步），然后node头节点查找并修改节点中的数据，CAS操作节点值，for自旋锁操作

## HashMap

见后文分析：hashMap原理

hash数组（记录hash值，根据hashcode查找O1）+链表（Node节点key=查找值，value为输入值，next为链表下个节点）

删除操作：synchronized 锁node头节点（线程同步），node.next = next.next移除

如果链表深度大于8，那么Node转换为红黑树TreeNode

## ConcurrentLinkedDeque原理

使用链表结构（next,pre），Node节点使用for自旋+CAS操作保证线程同步，局部变量head,tail记录头尾

ConcurrentLinkedQueue原理，使用链表结构（next）, Node节点使用for自旋+CAS操作保证线程同步，局部变量head,tail记录头尾

## ConcurrentSkipListMap原理

## SkipList跳表原理

通过使用SkipList跳表来实现快速定位，操作效率同红黑树TreeMap，但是ConcurrentSkipListMap适用于多个线程大量数据的情况下：

举例：在4线程1.6万数据的条件下，ConcurrentHashMap 存取速度是ConcurrentSkipListMap 的4倍左右

使用建议：

非多线程的情况下，应当尽量使用TreeMap。此外对于并发性相对较低的并行程序可以使用Collections.synchronizedSortedMap将TreeMap进行包装，也可以提供较好的效率。对于高并发程序，应当使用ConcurrentSkipListMap，能够提供更高的并发度

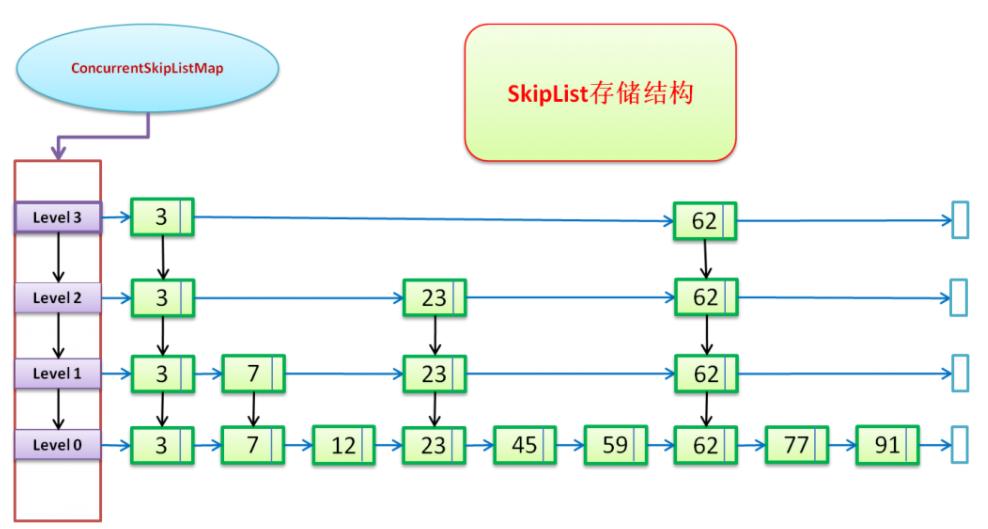
跳表：

针对链表接口一般无法随机访问，以及查找log(n)，在普通链表上添加层次接口，提高定位查找速度

对比红黑树：

跳表和红黑树的性能相当，最主要的优势就是当调整(插入或删除)时，红黑树需要使用旋转来维护平衡性，这个操作需要动多个节点，在并发时候很难控制。而跳表插入或删除时只需定位后插入，插入时只需添加插入的那个节点及其多个层的复制，以及定位和插入的原子性维护。所以它更加可以利用CAS操作来进行无锁编程。

原理图：



Level0 ~ Leveln 随机规律生成，用于添加表层次，方便查找

Level0 最底层包含所有元素，每一层的下面所有层包含此层所有的元素

绿色node节点数据结构：

static class Index<K,V> {  
 final Node<K,V> node;  
 final Index<K,V> down;  
 volatile Index<K,V> right;

}

拥有下一层指向down节点，拥有当前层下个节点right指向，以及node真实数据

static final class Node<K,V> {  
 final K key;  
 volatile Object value;  
 volatile Node<K,V> next;

}

内部数据结构：

*/\*\*  
 \* The topmost head index of the skiplist.  
 \*/*private transient volatile HeadIndex<K,V> head;

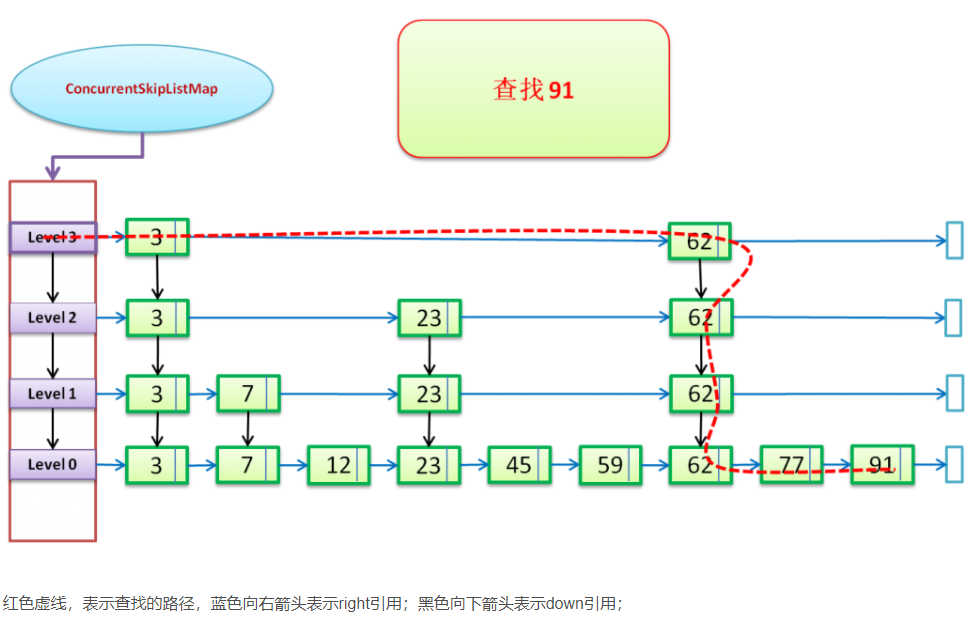
head表示当前最高层级的指针，图上目前是level3

headIndex结构如下：

*/\*\*  
 \* Nodes heading each level keep track of their level.  
 \*/*static final class HeadIndex<K,V> extends Index<K,V> {  
 final int level;  
 HeadIndex(Node<K,V> node, Index<K,V> down, Index<K,V> right, int level) {  
 super(node, down, right);  
 this.level = level;  
 }  
}

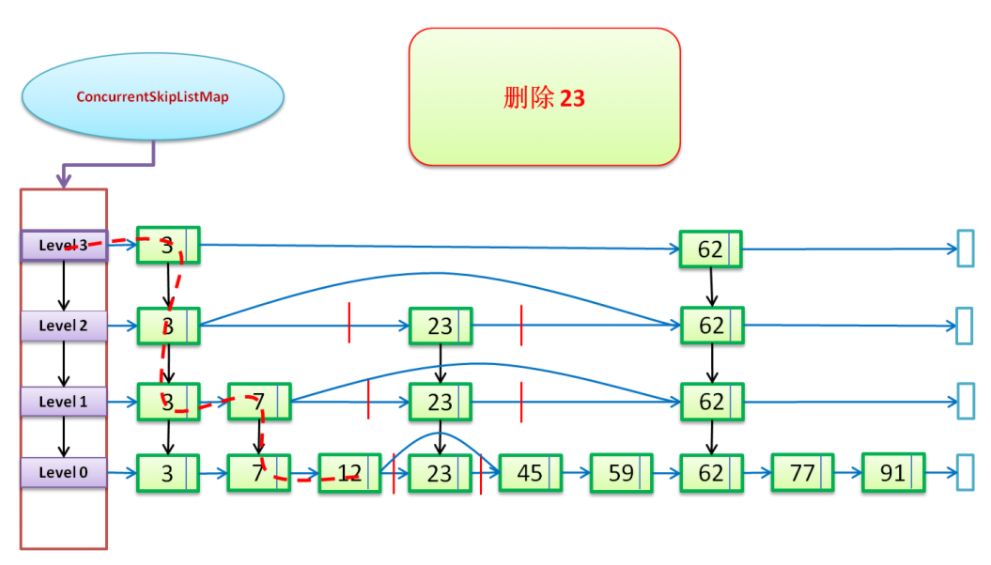
用于追踪记录当前等级，同时也是一个index节点，拥有down与right节点

查找



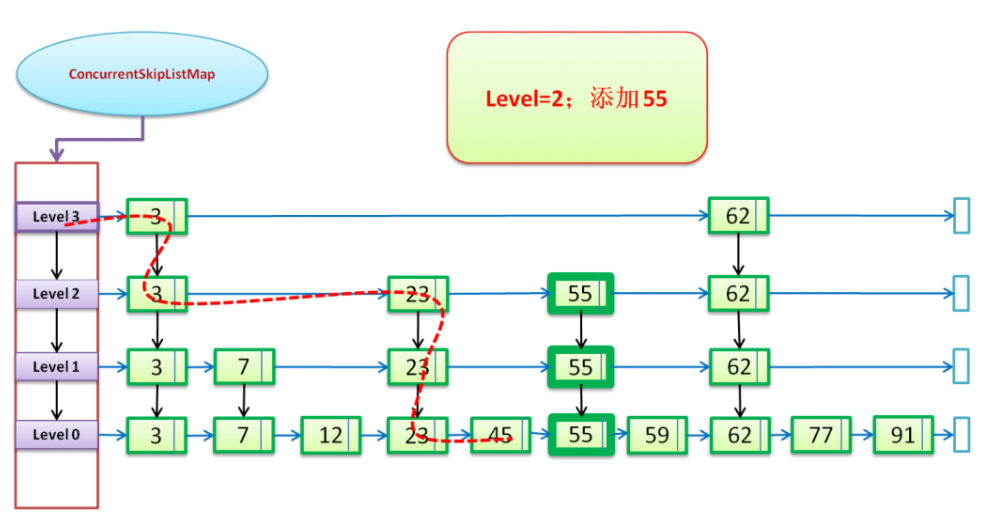
依次向下层查找

删除：



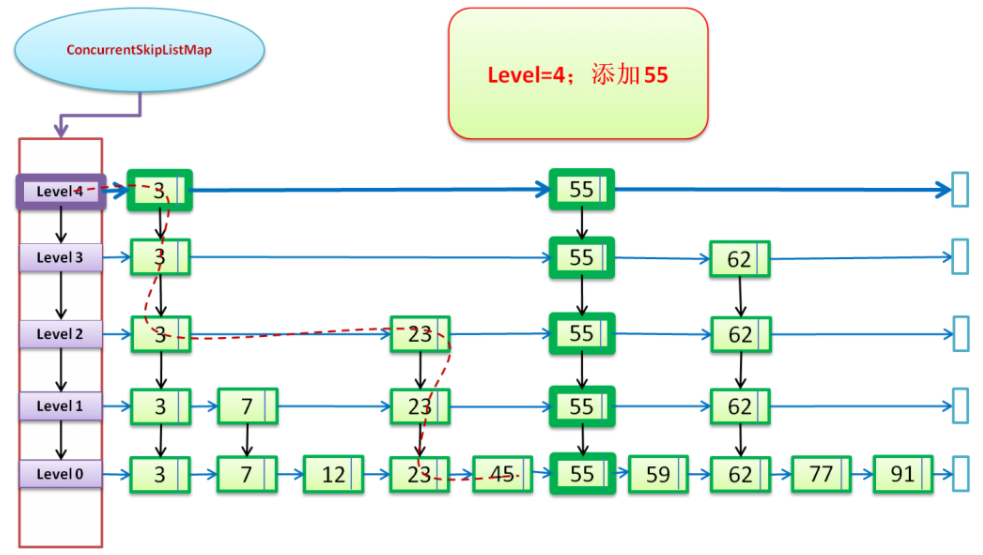
删除只需要找到满足的最高level层，删除链表节点，并向下查找层删除对应节点

插入：



存在level层的情况使用查找插入

如果不存在level层[新生成level4层了，插入元素55]



插入最高一层数据，同时CAS修改head指向level4

优缺点：

因为多了level层次，存在用空间换时间，默认是按照Key值升序的。Skip list让已排序的数据分布在多层链表中，以0-1随机数决定一个数据的向上攀升与否，通过“空间来换取时间”的一个算法，在每个节点中增加了向前的指针，在插入、删除、查找时可以忽略一些不可能涉及到的结点，从而提高了效率。

适用大数据多线程的情况，对比ConcurrentHashMap，有序的key，查找快，但是空间占用大，ConcurrentHashMap适用少线程，少量数据，适用spareArray可以适用单线程，大量数据

TreeMap适用单线程，大量数据，TreeMap是有序的key，查找方便，链表结构。spareArray是无序的key，查找方便，使用数组结构

## CopyOnWriteArrayListCopyOnWriteArraySet原理

使用ReentrantLock写操作（system.arrayCopy，写入完成替换数组引用），读操作是不加锁的，所以该CopyOnWriteArrayList适用于读多写少的场景

public E remove(int index) {  
 final ReentrantLock lock = this.lock;  
 lock.lock();  
 try {  
 Object[] elements = getArray();  
 int len = elements.length;  
 E oldValue = get(elements, index);  
 int numMoved = len - index - 1;  
 if (numMoved == 0)  
 setArray(Arrays.*copyOf*(elements, len - 1));  
 else {  
 Object[] newElements = new Object[len - 1];  
 System.*arraycopy*(elements, 0, newElements, 0, index);  
 System.*arraycopy*(elements, index + 1, newElements, index,  
 numMoved);  
 setArray(newElements);  
 }  
 return oldValue;  
 } finally {  
 lock.unlock();  
 }  
}

通过拷贝array，将index那个移除掉，然后通过setArray修改array变量的指向，如果在read读取的时候，读取的是array的index（非最新内容）

public E get(int index) {  
 return get(getArray(), index);  
}

DelayQueue使用ReEnterLock包装PriorityQueue操作，及对PriorityQueue同步

原理：泛型E实现Delayed接口用于获取等待时间，每次添加数据，加入PriorityQueue并比较delayed接口返回值并插入PriorityQueue二叉堆排序数组，这样在获取元素take方法时候比较head节点getDelayed是否满足<0，并弹出，否则ReEnterLock. Condition等待delayed时间，自旋阻塞获取

PriorityBlockingQueue：对PriorityQueue线程同步，空取阻塞，使用ReentrantLock线程同步操作

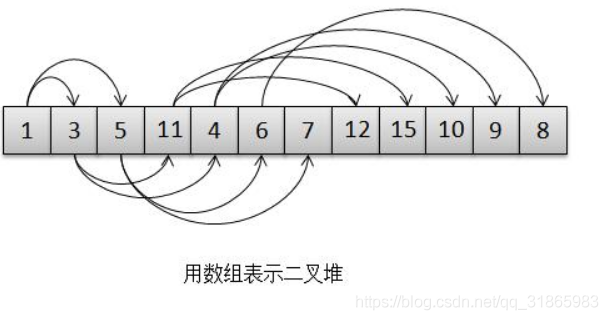
## PriorityQueue原理

有序的List，拥有compare接口）核心方法：siftDown、siftUp原理

二叉堆实现：完全二叉树，使用数组表示

PriorityQueue对元素采用的是堆排序，头是按指定排序方式的最小元素。堆排序只能保证根是最大（最小），整个堆并不是有序的

## 二叉堆实现原理：



父节点和子节点的编号是有联系的，更确切的说父子节点的编号之间有如下关系：

leftNo = parentNo\*2+1

rightNo = parentNo\*2+2

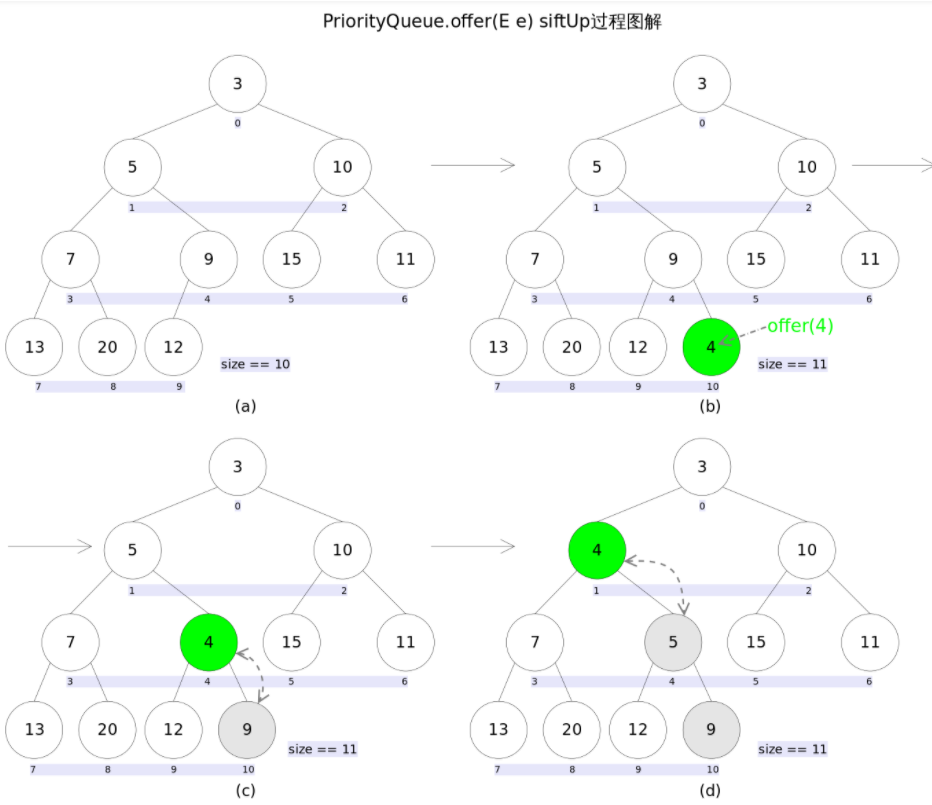
parentNo = (nodeNo-1)/2

计算子节点大小，当前index \* 2 +- 1表示子节点

数组只有第一个是整个数组里面最小的，最后一个元素是数组的最后一个index+1

添加：

1. Array[]最后一个数组index+1赋值



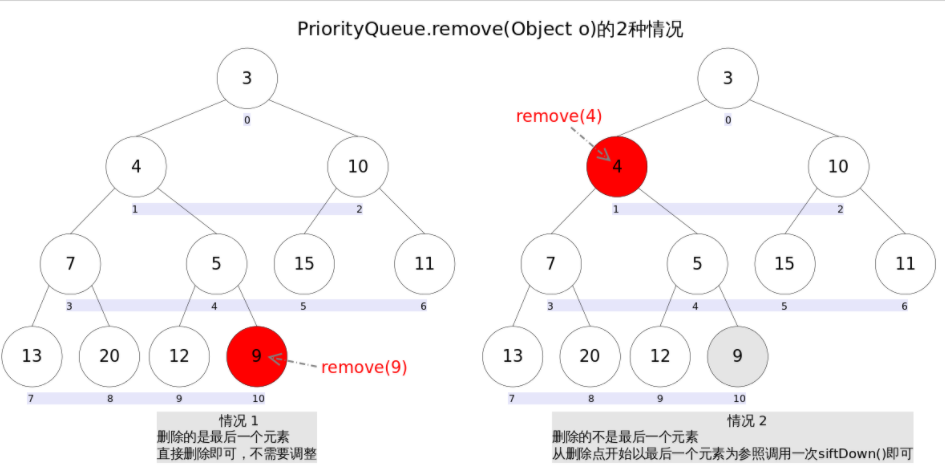
2. 与父节点比较，如果满足比较大小，交换位置，再次与父节点比较，直到不满足，则当前位置是满足子节点都比此节点（大/小）

查找：

第一个（element()和peek()）：

不删除第一个元素，返回index = 0 位置，此数组是数组最大/小的内容

删除：



将最后一个元素与当前删除的位置替换内容，然后与子节点替换位置，如果满足大/小，继续向下交换（只交换内容，交换结束删除数组最后一个index=null值的空间节点）

优缺点：

通过小顶堆实现，内部是无序的，只有第一个元素确定是最小/最大的，子节点的数组都比父节点大/小，但是同层级之间没有可比性，不能与TreeMap对比

## Exchanger<V>实现原理：

用于线程安全交换2个同类型变量，等待同时完成，内部Node维护保存thread（park）与内容object，在2个数据都满足了交换的时候，通过unsafe执行CAS操作

LinkedBlockingDeque、LinkedBlockingQueue使用ReentrantLock锁定操作，并且可以预置列表长度，默认Integer最大值，如果达到长度/空取等待（双向/单链表）

同LinkedBlockingQueue类型，take阻塞，直到有put进入，xfer阻塞，直到有take取用，内部没有数组保存，通过Node链表实现，入队操作拥有同步/异步模式，异步模式将形成链节点加入链表tail，同步模式将查询是否存在消费者，否则阻塞等待消费者产生（put如果队列满，等待take）

LinkedTransferQueue：（是一种生产者-消费者的应用场景，适用于生产者必须等待消费者进行接收数据的阻塞队列）（put可以异步等待消费，无限制）

## SynchronousQueue实现原理：

通过Node结构体生成链表结构，线程添加可用于等待获取，本身没有容量限制（put等待消费，在生成者传入的时候，等待消费者获取）。

SynchronousQueue：线程A使用put将数据添加到队列，如果没有其他线程使用take去获取数据，那么线程A阻塞，直到数据被其他线程获取，同理 如果线程B从队列中获取数据为空，被阻塞，等待线程添加数据。

LinkedTransferQueue：LinkedTransferQueue使用put,tryTransfer和transfer可添加多条数据, LinkedTransferQueue具有SynchronousQueue的功能，但是LinkedTransferQueue的生产者不会阻塞。tryTransfer和transfer与put不同的是，tryTransfer和transfer可检测是否有线程在等待获取数据，检测到则直接唤醒等待线程将数据给这个线程而不用放入队列。

队列模式：

LinkedBlockingQueue容量限制：等待存取[AQS锁定]

LinkedTransferQueue：生产者消费者，可以生产者可以等待消费者，也可以不等待，消费者（选择性等待）[有序take，内部使用CAS操作，没有AQS实现]

SynchronousQueue：生产者必须等待消费者（put等待，offer可以不等待）[有序take，内部使用CAS操作，没有AQS实现]

## ForkJoinPool（ForkJoinTask、ForkJoinWorkerThread）RecursiveAction、RecursiveTask原理

对于ThreadPoolExector线程池，使用blockingQueue只能顺序执行入队任务，当任务过大的时候，存在后续任务等待时间过久，导致等待太久，针对blockingQueue的缺点，引入了ForkJoinPool

可以将1个过大的任务（执行时间过长的任务），拆分为很多小任务执行并汇总

举例：

计算1~100W数字之和

1. 单个runnable执行for循环计算 耗时很久

采用ForkJoinPool，将1~100W采取分治法拆区间执行并将结果汇总，比如：1~50W + 50W~100W 之和，然后针对 50W区间再次拆分计算和，有点递归的意思

适用场景：

任务拆分汇总，单个任务可以被拆分并且不存在关联关系

采取算法：

work-stealing（工作窃取算法）

ForkJoinPool提供了一个更有效的利用线程的机制，当ThreadPoolExecutor还在用单个队列存放任务时，ForkJoinPool已经分配了与线程数相等的队列（WorkQueue），当有任务加入线程池时，会被平均分配到对应的队列上，各线程进行正常工作，当有线程提前完成时，会从队列的末端“窃取”其他线程未执行完的任务，当任务量特别大时，CPU多的计算机会表现出更好的性能

ForkJoinPool主要用来使用分治法(Divide-and-Conquer Algorithm)来解决问题。典型的应用比如快速排序算法。

区别：ThreadPoolExecutor中的Thread无法选择优先执行子任务

使用方式：（举例）

public class CountTaskTmp extends RecursiveTask<Integer> {  
 private static final int *THRESHOLD* = 2;  
 private int start;  
 private int end;  
 public CountTaskTmp(int start, int end) {  
 this.start = start;  
 this.end = end;  
 }  
 //实现compute 方法来实现任务切分和计算  
 protected Integer compute() {  
 int sum = 0;  
 boolean canCompute = (end - start) <= *THRESHOLD*;  
 if (canCompute) {  
 for (int i = start; i <= end; i++)  
 sum += i;  
 } else {  
 //如果任务大于阀值，就分裂成两个子任务计算  
 int mid = (start + end) / 2;  
 CountTaskTmp leftTask = new CountTaskTmp(start, mid);  
 CountTaskTmp rightTask = new CountTaskTmp(mid + 1, end);  
 //执行子任务  
 leftTask.fork();  
 rightTask.fork();  
 //等待子任务执行完，并得到结果  
 int leftResult = (int) leftTask.join();  
 int rightResult = (int) rightTask.join();  
  
 sum = leftResult + rightResult;  
 }  
 return sum;  
 }  
 public static void main(String[] args) {  
 //使用ForkJoinPool来执行任务  
 ForkJoinPool forkJoinPool = new ForkJoinPool();  
 //生成一个计算资格，负责计算1+2+3+4  
 CountTaskTmp task = new CountTaskTmp(1, 4);  
 Integer r = forkJoinPool.invoke(task);  
 System.*out*.println(r);  
 }  
}

核心类：

线程类： ForkJoinWorkerThread 数据结构

final ForkJoinPool pool; // the pool this thread works in  
final ForkJoinPool.WorkQueue workQueue; // work-stealing mechanics

run执行

try {  
 onStart();  
 pool.runWorker(workQueue);  
} catch (Throwable ex) {  
 exception = ex;  
}

可以看出，每个线程拥有一个WorkQueue并且执行该WorkQueue

线程池类：ForkJoinPool数据结构

volatile WorkQueue[] workQueues; // main registry

final int config; // parallelism, mode

volatile long ctl; // main pool control

内部拥有WorkQueue数组，与ForkJoinWorkerThread一一对应（一个线程对应一个WorkQueue），线程数量由config变量判断，初始化设置的parallelism参数与mode[同步/异步]位操作结果

线程的工作队列类：WorkQueue数据结构

ForkJoinTask<?>[] array; // the elements (initially unallocated)  
final ForkJoinPool pool; // the containing pool (may be null)  
final ForkJoinWorkerThread owner; // owning thread or null if shared  
volatile Thread parker; // == owner during call to park; else null  
volatile ForkJoinTask<?> currentJoin; // task being joined in awaitJoin  
volatile ForkJoinTask<?> currentSteal; // mainly used by helpStealer

内部拥有一个数组array表示队列中的任务数量

持有pool和所在线程owner引用

currentJoin与currentSteal表示当前引用任务

实现步骤：

1. submit（forkJoinTask）提交一个任务

1.1 如果线程不够parallelism，生成新线程（构造函数生成workQueue）

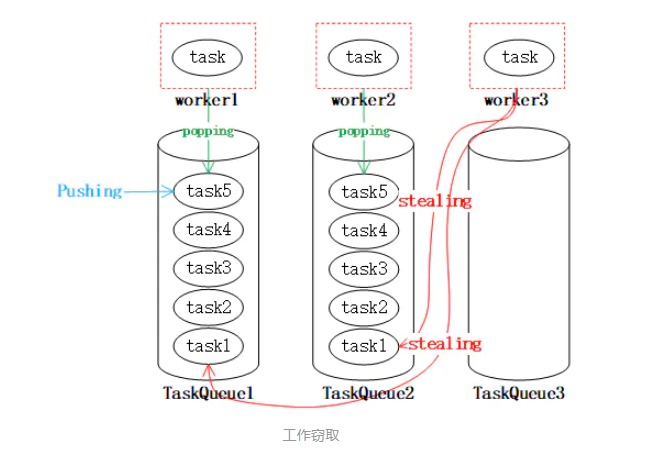
2.2 足够，将平均分给线程中的array（取较少的线程中的array并添加）

2. 新线程执行run方法

final void runWorker(WorkQueue w) {  
 w.growArray(); // allocate queue  
 int seed = w.hint; // initially holds randomization hint  
 int r = (seed == 0) ? 1 : seed; // avoid 0 for xorShift  
 for (ForkJoinTask<?> t;;) {  
 if ((t = scan(w, r)) != null)  
 w.runTask(t);  
 else if (!awaitWork(w, r))  
 break;  
 r ^= r << 13; r ^= r >>> 17; r ^= r << 5; // xorshift  
 }  
}

for循环等待获取任务，核心方法scan获取任务，真正任务[forkjoinTask]执行run

此时线程循环取自身workqueue里面array数组里面的forkjoinTask，如果该array数组已经空了，没有任务，scan将窃取其他线程的forkjoinTask执行



[先进后出LIFO，栈实现]

窃取算法：

从内部workQueue[]数组里面随机找到一个workQueue，并从workQueue队列array中获取一个forkjoinTask返回[要求该窃取的任务状态没有被执行]，取决于ForkJoinPool构造函数boolean asyncMode

asyncMode：[true：工作线程在处理本地任务时也使用 FIFO 顺序，这种模式下的 ForkJoinPool 更接近于是一个消息队列，而不是用来处理递归式的任务，默认false]

for循环等待获取任务，直到线程池关闭或者等待超时[!awaitWork(w, r)]，执行break，ForkJoinWorkerThread线程运行结束

1. ForkJoinPool 的每个工作线程都维护着一个工作队列（WorkQueue），这是一个双端队列（Deque），里面存放的对象是任务（ForkJoinTask）。
2. 每个工作线程在运行中产生新的任务（通常是因为调用了 fork()）时，会放入工作队列的队尾，并且工作线程在处理自己的工作队列时，使用的是 LIFO 方式，也就是说每次从队尾取出任务来执行。
3. 每个工作线程在处理自己的工作队列同时，会尝试窃取一个任务（或是来自于刚刚提交到 pool 的任务，或是来自于其他工作线程的工作队列），窃取的任务位于其他线程的工作队列的队首，也就是说工作线程在窃取其他工作线程的任务时，使用的是 FIFO 方式。[工作线程队列方式变更]
4. 在遇到 join() 时，如果需要 join 的任务尚未完成，则会先处理其他任务，并等待其完成。
5. 在既没有自己的任务，也没有可以窃取的任务时，进入休眠。

## Executors线程池

### ExecutorCompletionService原理：

（父类AbstractExecutorService）

内部使用一个BlockingQueue以及Exector，在提交任务之后将任务包装成内部Future,done返回之后将该future加入BlockingQueue，后续可以从内部的BlockingQueue中take取出结果

### AbstractExecutorService原理：

将runnable/callable包装成Future

invokeAny方法：随机启动传入的集合任务（不确定性）

invokeAll方法：启动所有任务，并等待所有任务执行完成返回（耗时）

for (Callable<T> t : tasks) {  
 RunnableFuture<T> f = newTaskFor(t);  
 futures.add(f);  
 execute(f);  
}  
for (int i = 0, size = futures.size(); i < size; i++) {  
 Future<T> f = futures.get(i);  
 if (!f.isDone()) {  
 try {

// 等待每个任务返回  
 f.get();  
 } catch (CancellationException ignore) {  
 } catch (ExecutionException ignore) {  
 }  
 }  
}  
done = true;  
return futures;

### FutureTask<V> 结果任务

原理：使用unSafe修改state状态值，并且在get方法判断是否state已经完成，如果没有，使用LockSupport.park阻塞等待（自旋），等待unpark退出自旋，state进入完成状态（Normal）

### ThreadLocalRandom

使用UnSafe操作修改Thread下面随机数threadLocalRandomSeed

体现ThreadLocal即在每个Thread下面添加了成员变量threadLocalRandomSeed，类型long，不管溢出问题

### \*\*\*ThreadPoolExecutor分析：

corePoolSize：常驻线程个数

maximumPoolSize：最大线程池大小，最大线程个数

workQueue：阻塞队列，用于存取

keepAliveTime：超过常驻线程之后启动的线程IDLE活跃时间，超时关闭

threadFactory：生产线程工厂

rejectedExecutionHandler：任务异常被拒绝执行（比如线程池被关闭了），后续需要执行的异常handler类

workerList：记录Worker工作线程集合

execute分析：

int c = ctl.get();  
if (*workerCountOf*(c) < corePoolSize) {  
 if (addWorker(command, true))  
 return;  
 c = ctl.get();  
}  
if (*isRunning*(c) && workQueue.offer(command)) {  
 int recheck = ctl.get();  
 if (! *isRunning*(recheck) && remove(command))  
 reject(command);  
 else if (*workerCountOf*(recheck) == 0)  
 addWorker(null, false);  
}  
else if (!addWorker(command, false))  
 reject(command);

首次启动（未超过核心线程）：

Worker生成线程（threadFactory.newThread）

调用runWorker启动worker线程

Loop死循环getTask()[BlockingQueue.take()阻塞]并执行该task，如果task不为null执行，否则结束该循环：

while (task != null || (task = getTask()) != null) {  
 // runnable执行  
}

超时机制，即移除worker[退出while循环]

private Runnable getTask() {  
 boolean timedOut = false; // Did the last poll() time out?  
 // Are workers subject to culling?  
 boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;  
 try {  
 Runnable r = timed ?  
 workQueue.poll(keepAliveTime, TimeUnit.*NANOSECONDS*) :  
 workQueue.take();  
 if (r != null)  
 return r;  
 timedOut = true;  
 } catch (InterruptedException retry) {  
 timedOut = false;  
 }  
 }  
}

通过使用超时机制（从BlockingQueue中取出数据超时），返回null，那么worker退出while循环[allowCoreThreadTimeOut可以退出核心线程或者当前worker超过corePoolSize]

allowCoreThreadTimeOut || wc > corePoolSize;

2.如果超过核心线程数，将任务加入blockingQueue成功

校验worker，如果coreworker == 0，那么启动非核心worker

3.如果加入blockingQueue失败，即队列满

启动非核心worker[addWorker(command, false)]

int wc = *workerCountOf*(c);  
if (wc >= *CAPACITY* ||  
 wc >= (core ? corePoolSize : maximumPoolSize))  
 return false;

如果非核心线程，并且当前worker工作线程已经超过max最大了，那么加入失败，执行拒绝策略：

final void reject(Runnable command) {  
 handler.rejectedExecution(command, this);  
}

拒绝策略：

1.尝试加入核心线程，失败加入blockingQueue，失败加入max工作线程，如果当前worker个数超出，那么执行拒绝策略[加入队列失败，worker超过max限制]

2.每个worker存在取超时机制，如果当前worker个数超过核心格式，worker竞争获取runnable失败时候退出[allowCoreThreadTimeOut=false]，退出的worker不会保证顺序，任意worker都有可能退出，只要保证core数量的worker即可

# Jar/zip目录解析：

## Jar文件读写：

JarInputStream/JarOutputStream/JarFile/JarEntry

## Zip文件读写：

ZipInputStream/ZipOutputStream/ZipFile/ZipEntry

使用方式：

Jar继承Zip读写，都是创建对应Jar/Zip File执行读写，在读写的时候可以获取对应Entry表示内部目录，并执行一定操作

使用方式与读写不再介绍

# Java集合：

AbstractCollection，AbstractList，AbstractQueue，AbstractSequentialList，AbstractSet，AbstractMap 抽象集合类，子类实现继承自定义实现

## ArrayDeque实现原理：（队列）

实现Deque接口，循环队列

充分利用空间，使用head,tail记录当前位置，当head==tail的时候列表表示空，head与tail没有前后顺序

内部使用数组+head,tail记录当前位置维护循环队列实现

## ArrayList实现原理：

private static final int *DEFAULT\_CAPACITY* = 10;

transient Object[] elementData;

数据元素：elementData（初始长度10）

添加：

public boolean add(E e) {  
 ensureCapacityInternal(size + 1); // Increments modCount!!  
 elementData[size++] = e;  
 return true;  
}

扩容策略：

arraySize +1

System.arrayCopy数组

删除数据：

public E remove(int index) {  
 if (index >= size)  
 throw new IndexOutOfBoundsException(outOfBoundsMsg(index));  
  
 modCount++;  
 E oldValue = (E) elementData[index];  
  
 int numMoved = size - index - 1;  
 if (numMoved > 0)  
 System.*arraycopy*(elementData, index+1, elementData, index,  
 numMoved);  
 elementData[--size] = null; // clear to let GC do its work  
  
 return oldValue;  
}

一次后数组整体前移

## Arrays 工具方法：

BinarySearch()：二分查找法搜索数组中元素，否则返回需要插入的位置的取反值，使用之前必须数组是有序的

1. 可以判断是否>0存在元素

2. 如果<0 ，使用~index取反操作可以确定插入的位置

3. 搜索算法：binarySearch0

？？？

copyOf(),copyOfRange() ：拷贝数组

fill()：填充数组

sort()：数组排序

排序算法：DualPivotQuicksort.sort

？？？

toString()：返回string串

## BitSet：位图原理（适用场景：整数，无重复；）

使用位图来维护一个整型的是否出现

原理分析：

*/\*\*  
 \* The internal field corresponding to the serialField "bits".  
 \*/*private long[] words;

内部使用long类型数组，长度是64 \* N，一个long = 64位，可用于记录0~63数字，该long的位操作，如设置3 出现则

原内容一个Long = 0000000000…0000;

设置之后 Long = 0000000000…1000;

当数组空间不足时候扩容：

public void set(int bitIndex) {  
 if (bitIndex < 0)  
 throw new IndexOutOfBoundsException("bitIndex < 0: " + bitIndex);  
  
 int wordIndex = *wordIndex*(bitIndex);  
 expandTo(wordIndex);  
  
 words[wordIndex] |= (1L << bitIndex); // Restores invariants  
  
 checkInvariants();  
}

使用位操作可以紧凑的记录数字是否与否，相比Long/Int类型的集合Set节省内存空间

举例：计算1000万个数字是否出现过，以及未出现的数字

使用HashSet可能需要1000万\*整数类型 +1000万hash 数组 的内存空间

使用普通数组需要1000万\*整数类型 的内存空间

使用BitSet只需要 1000万 / 64 的内存空间

## Calendar，Date ，GregorianCalendar（带时区）日期的使用

## Collections：

binarySearch：二分查找法，使用之前必须数组是有序的

reverse：数组反转

shuffle：数组打乱（重新洗牌）

fill（填充），copy（拷贝），swap（交换）

min，max：依次查找最小/最大内容

replaceAll：替换元素

unmodifiable/List/Set/Map：生成不可修改内容

synchronizedCollection：集合线程同步

asLifoQueue：Deque队列转常规Queue队列

## EnumMap：枚举Map

原理：

*/\*\*  
 \* All of the values comprising K. (Cached for performance.)  
 \*/*private transient K[] keyUniverse;  
  
*/\*\*  
 \* Array representation of this map. The ith element is the value  
 \* to which universe[i] is currently mapped, or null if it isn't  
 \* mapped to anything, or NULL if it's mapped to null.  
 \*/*private transient Object[] vals;

K数组存在Enum.value 下标内容

Value 存入对应key的index下标存入object内容

## HashMap：原理

*/\*\*  
 \* The default initial capacity - MUST be a power of two.  
 \*/*static final int *DEFAULT\_INITIAL\_CAPACITY* = 1 << 4; // aka 16  
  
*/\*\*  
 \* The maximum capacity, used if a higher value is implicitly specified  
 \* by either of the constructors with arguments.  
 \* MUST be a power of two <= 1<<30.  
 \*/*static final int *MAXIMUM\_CAPACITY* = 1 << 30;  
  
*/\*\*  
 \* The load factor used when none specified in constructor.  
 \*/*static final float *DEFAULT\_LOAD\_FACTOR* = 0.75f;

*/\*\*  
 \* The table, initialized on first use, and resized as  
 \* necessary. When allocated, length is always a power of two.  
 \* (We also tolerate length zero in some operations to allow  
 \* bootstrapping mechanics that are currently not needed.)  
 \*/*transient Node<K,V>[] table;

1. 默认初始化容量16

2. 最大列表长度2^30（计算hash值）

3. factor，列表增长因子，当数组有效数据容量达到75%，数组扩容

4. 数组节点（计算hash值）

5. Node是每个数组内数组的head链表

Node数据结构：

static class Node<K,V> implements Map.Entry<K,V> {  
 final int hash;  
 final K key;  
 V value;  
 Node<K,V> next;

}

Hash值，key ，value，下一个节点next

添加：

计算hash数组下标

static final int hash(Object key) {  
 int h;  
 return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}

p = tab[i = (n - 1) & hash]

如果链表深度大于8，那么Node转换为红黑树TreeNode

当元素数量size（已经包含的元素的个数） > threshold的时候，threshold = 容量\*负载因子，数组内容扩充1倍，并且rehash

Rehash过程，重新计算每个元素的hash值并生成Node重新赋值

为什么Hash数组长度一定要保证是2的次幂？

比如16的二进制表示为 10000，那么length-1就是15，二进制为01111，同理扩容后的数组长度为32，二进制表示为100000，length-1为31，二进制表示为011111。

解释： 这样会保证低位全为1，而扩容后只有一位差异，也就是多出了最左位的1，这样在通过 h&(length-1)的时候，只要h对应的最左边的那一个差异位为0，就能保证得到的新的数组索引和老数组索引一致(大大减少了之前已经散列良好的老数组的数据位置重新调换)

经过rehash之后，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置，我们在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了

**WeakHashMap：**

*/\*\*  
 \* The entries in this hash table extend WeakReference, using its main ref  
 \* field as the key.  
 \*/*private static class Entry<K,V> extends WeakReference<Object> implements Map.Entry<K,V> {  
 V value;  
 final int hash;  
 Entry<K,V> next;

Node节点的Key是弱引用的HashMap

每次调用getTable的时候清除一下弱引用，并且重新刷新一下链表，移除该entry节点链

## HashTable：同步的hashMap

与hashMap相比：不能存入key/value = null的数据

使用synchronized线程安全，性能不如concurrentHashMap

HashTable扩容策略/因子同hashMap，但HashTable快被淘汰了，不建议使用

HashTable是继承的Dictionary字典，2者都实现的Map接口

## IdentityHashMap：

对比hashMap，唯一区别就是比较的是地址，使用 == 比较key

而hashMap使用equals与hashCode比较key

## LinkedHashMap：

public class LinkedHashMap<K,V>  
 extends HashMap<K,V>  
 implements Map<K,V>

继承HashMap

有序的hashMap。Key存入的顺序

实现原理：

1. Node是双向链表

static class LinkedHashMapEntry<K,V> extends HashMap.Node<K,V> {  
 LinkedHashMapEntry<K,V> before, after;  
 LinkedHashMapEntry(int hash, K key, V value, Node<K,V> next) {  
 super(hash, key, value, next);  
 }  
}

Node<K,V> newNode(int hash, K key, V value, Node<K,V> e) {  
 LinkedHashMapEntry<K,V> p =  
 new LinkedHashMapEntry<K,V>(hash, key, value, e);  
 linkNodeLast(p);  
 return p;  
}

2. 覆盖HashMap添加删除回调方法

afterNodeAccess、afterNodeInsertion、afterNodeRemoval

3. 引用头节点与尾节点

*/\*\*  
 \* The head (eldest) of the doubly linked list.  
 \*/*transient LinkedHashMapEntry<K,V> head;  
  
*/\*\*  
 \* The tail (youngest) of the doubly linked list.  
 \*/*transient LinkedHashMapEntry<K,V> tail;

## LinkedHashSet 继承HashSet，没有其他修改点，和HashSet一致

## LinkedList：双向链表

*/\*\*  
 \* Pointer to first node.  
 \* Invariant: (first == null && last == null) ||  
 \* (first.prev == null && first.item != null)  
 \*/*transient Node<E> first;  
  
*/\*\*  
 \* Pointer to last node.  
 \* Invariant: (first == null && last == null) ||  
 \* (last.next == null && last.item != null)  
 \*/*transient Node<E> last;

private static class Node<E> {  
 E item;  
 Node<E> next;  
 Node<E> prev;  
  
 Node(Node<E> prev, E element, Node<E> next) {  
 this.item = element;  
 this.next = next;  
 this.prev = prev;  
 }  
}

与ArrayDeque区别：

ArrayDeque使用的是队列，LIFO,FIFO，内部使用array数组实现，查找快速（head,tail是int类型的index）

LinkedList使用的是双向链表，查找慢，删除插入快速（head,tail是Node类型）

## Vector、Stack：

栈的一种，方式为FIFO与LIFO，使用array数组存储数据，扩容策略：默认

int newCapacity = oldCapacity + ((capacityIncrement > 0) ?  
 capacityIncrement : oldCapacity);

使用synchronized，线程同步

## TreeMap、TreeSet红黑树原理：（有序，与跳表原理相比）

待分析

# Properties：读取配置文件默认属性

列如 1.txt：

Name=xiaoming

Age=15

实现原理：

public synchronized void load(Reader reader) throws IOException {  
 load0(new LineReader(reader));  
}

private void load0 (LineReader lr) throws IOException {  
 char[] convtBuf = new char[1024];  
 while ((limit = lr.readLine()) >= 0) {  
 String key = loadConvert(lr.lineBuf, 0, keyLen, convtBuf);  
 String value = loadConvert(lr.lineBuf, valueStart, limit - valueStart, convtBuf);  
 put(key, value);  
 }  
}

线程同步的，读取Reader对象，内容必须是单行的，使用XX = XX形式

# Timer、TimerTask定时器原理

Timer数据结构：

*/\*\*  
 \* The timer task queue. This data structure is shared with the timer  
 \* thread. The timer produces tasks, via its various schedule calls,  
 \* and the timer thread consumes, executing timer tasks as appropriate,  
 \* and removing them from the queue when they're obsolete.  
 \*/*private final TaskQueue queue = new TaskQueue();  
  
*/\*\*  
 \* The timer thread.  
 \*/*private final TimerThread thread = new TimerThread(queue);

TaskQueue：存储TimerTask任务

原理：

通过启动线程，while循环获取queue里面数据，如果队列为null，queue等待（wait），每次加入数据的时候判断是否是第一个数据，如果是，那么唤醒等待

while (true) {  
 try {  
 TimerTask task;  
 boolean taskFired;  
 synchronized(queue) {  
 // Wait for queue to become non-empty  
 while (queue.isEmpty() && newTasksMayBeScheduled)  
 queue.wait();  
 if (queue.isEmpty())  
 break; // Queue is empty and will forever remain; die  
  
 // Queue nonempty; look at first evt and do the right thing  
 long currentTime, executionTime;  
 task = queue.getMin();  
 synchronized(task.lock) {  
 if (task.state == TimerTask.*CANCELLED*) {  
 queue.removeMin();  
 continue; // No action required, poll queue again  
 }  
 currentTime = System.*currentTimeMillis*();  
 executionTime = task.nextExecutionTime;  
 if (taskFired = (executionTime<=currentTime)) {  
 if (task.period == 0) { // Non-repeating, remove  
 queue.removeMin();  
 task.state = TimerTask.*EXECUTED*;  
 } else { // Repeating task, reschedule  
 queue.rescheduleMin(  
 task.period<0 ? currentTime - task.period  
 : executionTime + task.period);  
 }  
 }  
 }  
 if (!taskFired) // Task hasn't yet fired; wait  
 queue.wait(executionTime - currentTime);  
 }  
 if (taskFired) // Task fired; run it, holding no locks  
 task.run();  
 } catch(InterruptedException e) {  
 }  
}

添加数据：

synchronized(queue) {  
 if (!thread.newTasksMayBeScheduled)  
 throw new IllegalStateException("Timer already cancelled.");  
  
 synchronized(task.lock) {  
 if (task.state != TimerTask.*VIRGIN*)  
 throw new IllegalStateException(  
 "Task already scheduled or cancelled");  
 task.nextExecutionTime = time;  
 task.period = period;  
 task.state = TimerTask.*SCHEDULED*;  
 }  
  
 queue.add(task);  
 if (queue.getMin() == task)  
 queue.notify();  
}

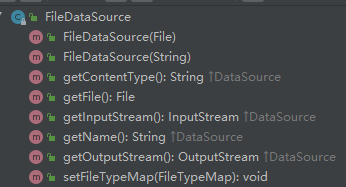
通过synchronized实现线程同步，内部等待唤醒机制queue.wait（long timeOut）

# javax.activation

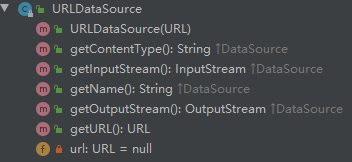
DataSource 数据源解析（文件/URL）

实现类：

FileDataSource 文件源，常用方法：



URLDataSource 统一资源定位符，常用方法：



DataHandler ： DataSource资源处理器

public DataHandler(DataSource ds) {  
 // save a reference to the incoming DS  
 dataSource = ds;  
 oldFactory = *factory*; // keep track of the factory  
}

包装DataSource并提供额外方法

# org.json ：Android Json相关

JSONArray、JSONObject不再介绍

JSONTokener：json串解析器

*/\*\*  
 \** ***@param*** *in JSON encoded string. Null is not permitted and will yield a  
 \* tokener that throws {****@code*** *NullPointerExceptions} when methods are  
 \* called.  
 \*/*public JSONTokener(String in) {  
 // consume an optional byte order mark (BOM) if it exists  
 if (in != null && in.startsWith("\ufeff")) {  
 in = in.substring(1);  
 }  
 this.in = in;  
}

核心解析方法：

public Object nextValue() throws JSONException {  
 int c = nextCleanInternal();  
 switch (c) {  
 case -1:  
 throw syntaxError("End of input");  
  
 case '{':  
 return readObject();  
  
 case '[':  
 return readArray();  
  
 case '\'':  
 case '"':  
 return nextString((char) c);  
  
 default:  
 pos--;  
 return readLiteral();  
 }  
}

JSONStringer：Json串生成器（通过StringBuilder构建string）

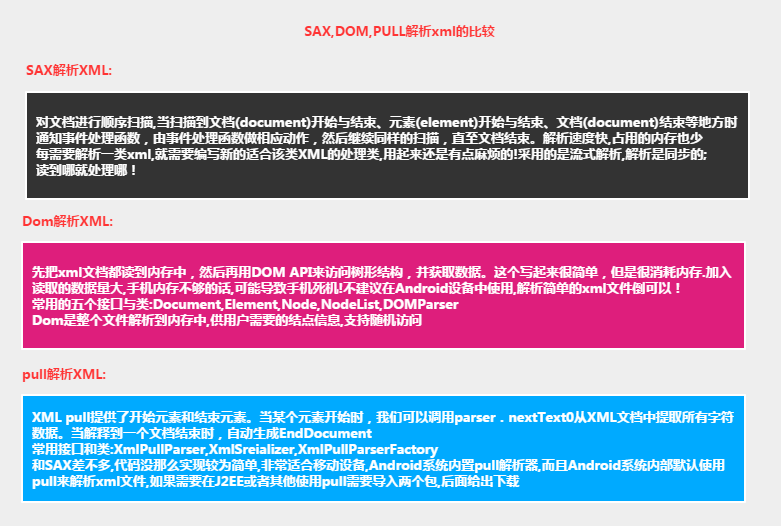
JsonObject生成json串：

void writeTo(JSONStringer stringer) throws JSONException {  
 stringer.object();  
 for (Map.Entry<String, Object> entry : nameValuePairs.entrySet()) {  
 stringer.key(entry.getKey()).value(entry.getValue());  
 }  
 stringer.endObject();  
}

jsonArray生成json串

void writeTo(JSONStringer stringer) throws JSONException {  
 stringer.array();  
 for (Object value : values) {  
 stringer.value(value);  
 }  
 stringer.endArray();  
}

# Android XML解析：（dom,sax,pull）



1. Dom解析：

//①获得DOM解析器的工厂示例:  
DocumentBuilderFactory dbFactory = DocumentBuilderFactory.*newInstance*();  
//②从Dom工厂中获得dom解析器  
DocumentBuilder dbBuilder = dbFactory.newDocumentBuilder();  
//③把要解析的xml文件读入Dom解析器  
Document doc = dbBuilder.parse(context.getAssets().open("person2.xml"));

通过pare一次性将xml解析进入Document对象中，形成树形结构数据，取用方便，但是极大占用内存，不推荐

2. Sax解析

自定义类实现继承DefaultHandler，覆盖对应方法

public class SaxHelper extends DefaultHandler2 {  
 private String tagName = null;//当前解析的元素标签  
  
 @Override  
 public void startDocument() throws SAXException {  
 super.startDocument();  
 }  
  
 @Override  
 public void endDocument() throws SAXException {  
 super.endDocument();  
 }  
  
 @Override  
 public void startElement(String uri, String localName, String qName, Attributes attributes) throws SAXException {  
 super.startElement(uri, localName, qName, attributes);  
 this.tagName = localName;  
 }  
  
 @Override  
 public void endElement(String uri, String localName, String qName) throws SAXException {  
 super.endElement(uri, localName, qName);  
 }  
  
 @Override  
 public void characters(char[] ch, int start, int length) throws SAXException {  
 //【接收标签中字符数据时，调用此方法】  
 //【ch存放标签中的内容，start是起始位置，length是内容长度】  
 super.characters(ch, start, length);  
 if(tagName != null) {  
 String data = new String(ch, start, length);  
 if (tagName.equals("name")) {  
 this.currentEntity.setName(data);  
 }  
 }  
 }  
}

创建解析方式：

public static List<Person> readXmlBySAX(InputStream inputStream) {  
 try {  
 // 【创建解析器】  
 SAXParserFactory spf = SAXParserFactory.*newInstance*();  
 SAXParser saxParser = spf.newSAXParser();  
 SaxHelper handler = new SaxHelper();  
 saxParser.parse(inputStream, handler);  
 inputStream.close();  
 return handler.getResult();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 return null;  
}

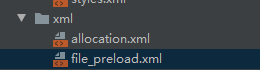
Pull解析方式：

public static List<Person> readXmlByPull(InputStream inputStream) {  
 // 【创建解析器】  
 XmlPullParser xmlPullParser = Xml.*newPullParser*();  
 xmlPullParser.setInput(inputStream,"utf-8");  
 int eventType = xmlPullParser.getEventType();  
 while (eventType != XmlPullParser.*END\_DOCUMENT*) {  
 switch (eventType) {  
 case XmlPullParser.*START\_DOCUMENT*:  
 break;  
 case XmlPullParser.*START\_TAG*:  
 String name = xmlPullParser.getName();  
 if("name".equals(name)) {  
 // 获取tag属性值  
 String value = xmlPullParser.nextText();  
 }  
 break;  
 case XmlPullParser.*END\_TAG*:  
 break;  
 }  
 eventType = xmlPullParser.next();  
 }  
}

Sax与Pull方式都是事件驱动的解析方式，流处理，减少内存占用。Android采用pull方式解析XML文件

# \*\*Android 自定义XML属性（xml文件预加载对象）

1. 定义xml文件，文件夹xml



2. 内容如下

<?xml version="1.0" encoding="utf-8"?>  
<xml xmlns:android="http://schemas.android.com/apk/res/android">  
 <tag name="key1">value1</tag>  
</xml>

3. 获取XML内容的pullParser

XmlResourceParser parser = getResources().getXml(R.xml.*file\_preload*);

4. 使用xml解析：

XMLEntity entity = null;  
while ((type = parser.next()) != XmlPullParser.*END\_DOCUMENT*) {  
 switch (type) {  
 case XmlPullParser.*START\_DOCUMENT*:  
 break;  
 case XmlPullParser.*START\_TAG*:  
 String name = parser.getName();  
 if ("tag".equals(name)) {  
 entity = getEntity(parser);  
 }  
 break;  
  
 }  
}

private XMLEntity getEntity(XmlResourceParser parser) throws IOException, XmlPullParserException {  
 XMLEntity entity = new XMLEntity();  
 entity.key = parser.getAttributeValue(0);  
 entity.value = parser.nextText();  
 return entity;  
}

详见android.animation动画AnimatorInflater

# Base64：基础编码解码方式

将byte[]转换成编码之后的byte[]

输出被映射到一组字符A-Za-z0-9+/，编码不添加任何行标，输出的解码仅支持A-Za-z0-9+/。

一共有64个字符，占用6个字节，一个byte是8个字节，3个byte = 4个确定位字符

将传入的byte[]，每次取3个字节，转换成4个字符，最后不足的使用’ =’字符补齐

输出内容是原先长度的4/3

自定义编码解码方式：

定义 0~9 A~F 共16字符。长度4位

1个byte = 8 位 = 2个字节

这样可以取一个byte高8位低8位输出2个字符

输出内容是原先长度的2倍

# javax

SSL加密传输相关：

SSLSocket、SSLServerSocket待解析

XML相关：

javax.xml 待解析

# Android

# AccessibilityService：辅助功能详解

前沿：

辅助服务的设计初衷提供给无法和界面进行交互的残疾用户。来协助帮助他们进行一些用户操作，比如点击，返回，长按，获取屏幕信息等能力。后来被开发者另辟蹊径，用于一些插件开发，做一些监听第三方应用的插件。

生命周期：

服务的启动只能用户在设备设置中明确启动服务来触发。当系统绑定到服务后，它会调用AccessibilityService#onServiceConnected方法。当用户在设置设置中关闭时，辅助服务功能将停止，或者调用AccessibilityService#disableSelf方法

## 使用方式

1. 自定义继承

* disableSelf，禁用当前服务，也就是说关闭当服务
* dispatchGesture(GestureDeion gesture, AccessibilityService.GestureResultCallback callback, Handler handler)，将手势发送到触摸屏
* findFocus(int focus)：寻找到具有指定焦点类型的视图。
* getRootInActiveWindow：获取当前活动窗口中的根节点。
* getServiceInfo：获取当前服务的配置信息
* getSystemService(String name)：获取系统服务
* getWindows:获取屏幕上的窗口。
* performGlobalAction(int action)：执行全局操作：比如返回，回到主页，打开最近等操作
* onAccessibilityEvent 发生用户界面事件回调此事件
* onInterrupt 中断可访问性反馈

不太常用的：

* getAccessibilityButtonController：返回系统导航区域内辅助功能按钮的控制器。
* getFingerprintGestureController：获取指纹手势的控制器。
* getMagnificationController:返回放大器控制器，可用于查询和修改显示放大率的状态。
* getSoftKeyboardController:返回软键盘控制器，可用于查询和修改软键盘显示模式。

2. 注册

<service  
 android:name="com.example.accesservice.MyApplicationAccessService"  
 android:label="辅助服务测试"  
 android:permission="android.permission.BIND\_ACCESSIBILITY\_SERVICE">  
 <intent-filter>  
 <action android:name="android.accessibilityservice.AccessibilityService"/>  
 </intent-filter>  
</service>

3. 编写XML配置文件

3.1 静态配置<meta-data/>

<accessibility-service xmlns:android="http://schemas.android.com/apk/res/android"  
 android:accessibilityEventTypes="typeAllMask"  
 android:deion="demo"  
 android:accessibilityFeedbackType="feedbackSpoken"  
 android:canRetrieveWindowContent="true"  
 android:notificationTimeout="1000"  
 />

<meta-data  
 android:name="android.accessibilityservice"  
 android:resource="@xml/allocation"/>

3.2 动态配置onServiceConnected

@Override  
protected void onServiceConnected() {  
 super.onServiceConnected();  
 AccessibilityServiceInfo info = new AccessibilityServiceInfo();  
 info.eventTypes = AccessibilityEvent.*TYPE\_VIEW\_CLICKED* | AccessibilityEvent.*TYPE\_VIEW\_FOCUSED*;  
 info.packageNames = new String[] {"com.example.android.myFirstApp"};  
 info.feedbackType = AccessibilityServiceInfo.*FEEDBACK\_SPOKEN*;  
 info.notificationTimeout = 100;  
 this.setServiceInfo(info);  
}

4. 处理事件：

第一步，寻找该控件。第二部，模拟人的操作对各个控件进行操作(点击，长按，输入，读取)

4.1 寻找控件首先要找的它对象窗口内容对应的树，即AccessibilityWindowInfo(代表)和AccessibilityNodeInfo(代表具体的View)。注意，这个功能需要在xml资源配置中声明SERVICEMETADATA。

**AccessibilityWindowInfo**：AccessibilityWindowInfo表示可访问窗口的状态快照。

**AccessibilityNodeInfo**：AccessibilityNodeInfo表示窗口内容的节点以及可以进行的操作。

4.2 处理事件--操作控件

AccessibilityAction表示通过addAction，可以给AccessibilityNodeInfo添加对于的动作。每个操作都有一个唯一的ID，这是必需的和可选的数据

4.3 手势控制：

在Android7.0之后，AccessibilityService又增加了一个新的方法dispatchGesture。可以将手势发送到触摸屏上。但是要使用这个功能必须在配置文件中声明，canPerformGestures = “true”

## 源码流程解析

1. AccessibilityService继承Service，onbind绑定IAccessibilityServiceClient.sub，通过AIDL实现binder远程调用

public final IBinder onBind(Intent intent) {  
 return new IAccessibilityServiceClientWrapper(this, getMainLooper(), new Callbacks() {  
 @Override  
 public void onServiceConnected() {  
 AccessibilityService.this.dispatchServiceConnected();  
 }  
 @Override  
 public void onInterrupt() {  
 AccessibilityService.this.onInterrupt();  
 }  
 @Override  
 public void onAccessibilityEvent(AccessibilityEvent event) {  
 AccessibilityService.this.onAccessibilityEvent(event);  
 }  
 @Override  
 public void init(int connectionId, IBinder windowToken) {  
 mConnectionId = connectionId;  
 mWindowToken = windowToken;  
 // The client may have already obtained the window manager, so  
 // update the default token on whatever manager we gave them.  
 final WindowManagerImpl wm = (WindowManagerImpl) getSystemService(*WINDOW\_SERVICE*);  
 wm.setDefaultToken(windowToken);  
 }  
  
}

public static class IAccessibilityServiceClientWrapper extends IAccessibilityServiceClient.Stub

下面分析事件如何分发给该服务的：

View.performClick事件：

public boolean performClick() {  
。。。。。。  
 sendAccessibilityEvent(AccessibilityEvent.*TYPE\_VIEW\_CLICKED*);  
。。。。。。。  
}

sendAccessibilityEvent ->sendAccessibilityEventInternal-> sendAccessibilityEventUncheckedInternal

最终调用ViewParent. requestSendAccessibilityEvent(ViewParent = ViewRootImpl)

@Override  
public boolean requestSendAccessibilityEvent(View child, AccessibilityEvent event) {  
 mAccessibilityManager.sendAccessibilityEvent(event);  
 return true;  
}

最后通过mAccessibilityManager分发的事件

mAccessibilityManager = AccessibilityManagerService # sendAccessibilityEvent

@Override  
public void sendAccessibilityEvent(AccessibilityEvent event, int userId) {  
 synchronized (mLock) {  
 notifyAccessibilityServicesDelayedLocked(event, false);  
 notifyAccessibilityServicesDelayedLocked(event, true);  
 mUiAutomationManager.sendAccessibilityEventLocked(event);  
 }  
 }  
 if (*OWN\_PROCESS\_ID* != Binder.*getCallingPid*()) {  
 event.recycle();  
 }  
}

在方法中，最后会调用notifyAccessibilityServicesDelayedLocked()方法，然后将event进行回收

private void notifyAccessibilityServicesDelayedLocked(AccessibilityEvent event,  
 boolean isDefault) {  
 try {  
 UserState state = getCurrentUserStateLocked();  
 for (int i = 0, count = state.mBoundServices.size(); i < count; i++) {  
 AccessibilityServiceConnection service = state.mBoundServices.get(i);  
  
 if (service.mIsDefault == isDefault) {  
 service.notifyAccessibilityEvent(event);  
 }  
 }  
 } catch (IndexOutOfBoundsException oobe) {  
 // An out of bounds exception can happen if services are going away  
 // as the for loop is running. If that happens, just bail because  
 // there are no more services to notify.  
 }  
}

通过绑定的跨组件服务，进行分发该事件

启用/关闭服务：

if (userState.mComponentNameToServiceMap.get(serviceName) == null) {  
 enableAccessibilityServiceLocked(serviceName, mCurrentUserId);  
} else {  
 disableAccessibilityServiceLocked(serviceName, mCurrentUserId);  
}

启用自定义的Service连接（updateServicesLocked(UserState userState)）

if (service == null) {  
 service = new AccessibilityServiceConnection(userState, mContext, componentName,  
 installedService, *sIdCounter*++, mMainHandler, mLock, mSecurityPolicy,  
 this, mWindowManagerService, mGlobalActionPerformer);  
} else if (userState.mBoundServices.contains(service)) {  
 continue;  
}  
service.bindLocked();

## 原理：观察者模式

通过启用AccessManagerService，将Service绑定到AccessibilityManagerService中，然后在ViewRootImpl应用中分发事件，AccessibilityManagerService保存的ServiceConnect执行回调事件（当前应用->系统服务->远程服务）

远程服务如何获取应用信息，与应用交互：

ViewRootImpl#ensureConnection

向AccessManagerService中添加ViewRootImpl#AccessibilityInteractionConnection，远程服务通过该连接可以查询ViewRootImpl的信息（远程服务->系统服务->当前应用）

mAccessibilityManager.addAccessibilityInteractionConnection(mWindow,  
 mContext.getPackageName(),  
 new AccessibilityInteractionConnection(ViewRootImpl.this));

交互连接存储到userState中（AccessManagerService#addAccessibilityInteractionConnection）

至此分析完毕

移除交互连接：

void dispatchDetachedFromWindow() {  
mAccessibilityInteractionConnectionManager.ensureNoConnection();

}

# Accounts目录

用户账户相关包，用于存储用户账号信息的，使用SQL保存信息，并通过SystemServer调用

调用核心类：

AccountManager

实现类：

AccountManagerService

操作数据库：com.android.server.accounts. AccountsDb

# android.animation动画

核心抽象类：Animator

## 动画类

ObjectAnimator、AnimatorSet、LayoutTransition

## layout布局动画(ViewGroup动画)

LayoutTransition

启用动画：

android:animateLayoutChanges="true"

子View添加动画：

mTransition.setAnimator(LayoutTransition.APPEARING, animIn);

子View移除动画

mTransition.setAnimator(LayoutTransition.CHANGE\_DISAPPEARING,  
changeOut);

## 动画加载方式

### AnimatorInflater.loadAnimator

private static Animator createAnimatorFromXml(Resources res, Theme theme, XmlPullParser parser,  
 AttributeSet attrs, AnimatorSet parent, int sequenceOrdering, float pixelSize)  
 throws XmlPullParserException, IOException {  
 Animator anim = null;  
 while (((type = parser.next()) != XmlPullParser.*END\_TAG* || parser.getDepth() > depth)  
 && type != XmlPullParser.*END\_DOCUMENT*) {  
 if (type != XmlPullParser.*START\_TAG*) {  
 continue;  
 }  
 if (name.equals("objectAnimator")) {  
 anim = *loadObjectAnimator*(res, theme, attrs, pixelSize);  
 } else if (name.equals("animator")) {  
 anim = *loadAnimator*(res, theme, attrs, null, pixelSize);  
 } else if (name.equals("set")) {  
 anim = new AnimatorSet();  
 *createAnimatorFromXml*(res, theme, parser, attrs, (AnimatorSet) anim, ordering,  
 pixelSize);   
 } else if (name.equals("propertyValuesHolder")) {  
 PropertyValuesHolder[] values = *loadValues*(res, theme, parser,  
 Xml.*asAttributeSet*(parser));  
 if (values != null && anim != null && (anim instanceof ValueAnimator)) {  
 ((ValueAnimator) anim).setValues(values);  
 }  
 gotValues = true;  
 } else {  
 throw new RuntimeException("Unknown animator name: " + parser.getName());  
 }  
 }  
 return anim;  
}

通过XmlResourceParser(继承XmlPullParser)解析XML文件得到属性文件，根据不同根标签值 生成不同对象并赋值

获取方式：getResources().get资源文件名

比如：

public XmlResourceParser getAnimation(@AnimatorRes @AnimRes int id) throws NotFoundException {  
 return loadXmlResourceParser(id, "anim");  
}

public XmlResourceParser getLayout(@LayoutRes int id) throws NotFoundException {  
 return loadXmlResourceParser(id, "layout");  
}

public XmlResourceParser getXml(@XmlRes int id) throws NotFoundException {  
 return loadXmlResourceParser(id, "xml");  
}

返回的XmlResourceParser（XmlBlock.Parser内部类实现）可用于解析数据，而XmlResourceParser可以经过转换成AttributeSet

AttributeSet attrs = Xml.asAttributeSet(parser)

### 2. 获取定义属性

TypedArray arrayAnimator = null;  
if (theme != null) {  
 arrayAnimator = theme.obtainStyledAttributes(attrs, R.styleable.Animator, 0, 0);  
} else {  
 arrayAnimator = res.obtainAttributes(attrs, R.styleable.Animator);  
}

### 3. 获取属性值

long duration = arrayAnimator.getInt(R.styleable.Animator\_duration, 300);  
long startDelay = arrayAnimator.getInt(R.styleable.Animator\_startOffset, 0);  
int valueType = arrayAnimator.getInt(R.styleable.Animator\_valueType, *VALUE\_TYPE\_UNDEFINED*);

### 4. 生成根标签对象并赋值

#### \*\*\*\*自定义XML获取属性并转换为attributeSet

1. 通过XMLPullParse解析，见AndroidXML解析

2. 通过attributeSet解析

2.1 生成attribute：

XmlResourceParser parser = getResources().getXml(R.xml.fileName);  
AttributeSet attributeSet = Xml.*asAttributeSet*(parser);

2.2 获取android属性：

带主题的属性：（默认取activity样式中的属性）

TypedArray typedArray = getTheme().obtainStyledAttributes(attrs, R.styleable.Animator, 0, 0);

如果不带主题，则使用带资源的属性：

TypedArray typedArray = getResources().obtainAttributes(attrs, R.styleable.Animator);

(AttributeSet 对象中的数据是从 XML 布局文件中读取出来的，因此此时如果我们不在布局文件中为 attrs 相关属性设置值，那么就会报错，所以建议使用带Theme主题的obtainStyledAttributes方法)

obtainStyledAttributes (AttributeSet set, int[] attrs, int defStyleAttr, int defStyleRes)方法解析：

attribute 最终由下面四个因素决定：

1. 在 AttributeSet 中定义的属性（Any attribute values in the given AttributeSet）；

2. AttributeSet 指定的样式资源文件（The style resource specified in the AttributeSet (named “style”)）；

3. 由 defStyleAttr 和 defStyleRes 指定的样式资源文件（The default style specified by defStyleAttr and defStyleRes）；

4. 主题中的默认值（The base values in this theme）

插值器TypeEvaluator

Animator.setEvaluator(evaluator);

常见插值器不再介绍，int/float/rgb etc.

# Android.app

## JobService

1.使用方法，自定义继承

public class MyJobService extends JobService {  
 @Override  
 public boolean onStartJob(JobParameters params) {  
 return false;  
 }  
 @Override  
 public boolean onStopJob(JobParameters params) {  
 return false;  
 }  
}

2. xml配置

<service android:name="com.example.job.MyJobService"  
 android:permission="android.permission.BIND\_JOB\_SERVICE"  
 />

3. 回调处理

onStartJob，任务被启动时调用，onStopJob，任务被关闭时候调用，参数入参

4. jobFinished任务执行完成，通知系统回调任务完成

### 源码解析

|  |  |  |
| --- | --- | --- |
| Client | 过程 | System |
| JobService-> | AIDL -> | Service.sub文件交给system ↓ |
| 申明自定义JobService（仅仅系统使用，外部使用存在安全问题）定义XML与权限 |  |  |
| 定义JobInfo,主动调用SystemService#Schedule-> | 携带JobInfo中component定义申明JobServcie -> | System使用JobInfo并安排周期任务 ↓ |
| Client自定义JobService触发onJobStart开始执行任务↓ | <- 绑定client service，并调用onJobStart【并传入JobParam参数，其中携带的JobCallback是回调，定义为IJobCallback.Stub】 | <- 周期任务触发，system绑定JobInfo中component中的Service，通过Service.sub文件回调onJobStart |
| 执行任务中 ↓ |  |  |
| 执行完成拿到入参 jobParam下面的 JobCallback通知系统回调任务完成 -> | JobCallback#onJobFinish -> | 系统处理收尾工作 |

注意：

为什么系统使用JobCallback让client回调，普通回调是否可行？

不可行，使用JobCallback是使用的IJobCallback.Stub，跨进程传递参数的时候使用的是parcelable，传递IJobCallback.Stub回调是binder对象，写入方法如下：

JobParameters(Parcel in)

callback = in.readStrongBinder();

writeToParcel(Parcel dest)

dest.writeStrongBinder(callback);

## Android 系统Service加载与调用

### 初始化方式

举例

public class JobSchedulerService extends com.android.server.SystemService{

@Override  
public void onStart() {  
 publishBinderService(Context.*JOB\_SCHEDULER\_SERVICE*, mJobSchedulerStub);  
}

protected final void publishBinderService(String name, IBinder service,  
 boolean allowIsolated, int dumpPriority) {  
 ServiceManager.addService(name, service, allowIsolated, dumpPriority);  
}

*/\*\*  
 \* Binder stub trampoline implementation  
 \*/*final class JobSchedulerStub extends IJobScheduler.Stub {}

}

初始化的Service在onStart下面向ServiceManager添加服务

### ServiceManager解析

ServiceManager由ServiceManagerNative实现，向native层添加/获取Service的binder[这里写入内核空间，多进程共享访问服务binder]将Context.XXX\_SERVICE常量String作为key，Ibinder作为value[由Service中XXX.Sub实例化]

1、提供addService()方法，向native层注册服务

2、提供getService()方法，从native层获取服务

3、维护了缓存Map

### App层获取系统服务Context.getSystemService原理

1. 远程服务：Ibinder模式

实现类：ContextImpl#getSystemService

@Override  
public String getSystemServiceName(Class<?> serviceClass) {  
 return SystemServiceRegistry.*getSystemServiceName*(serviceClass);  
}

最终通过SystemServiceRegistry获取：

*/\*\*  
 \* Gets the name of the system-level service that is represented by the specified class.  
 \*/*public static String getSystemServiceName(Class<?> serviceClass) {  
 return *SYSTEM\_SERVICE\_NAMES*.get(serviceClass);  
}

静态map存储内部service：

private static final Map<Class<?>, String> *SYSTEM\_SERVICE\_NAMES* =  
 new ArrayMap<Class<?>, String>();

static {  
 //CHECKSTYLE:OFF IndentationCheck  
 *registerService*(Context.*ACCESSIBILITY\_SERVICE*, AccessibilityManager.class,  
 new CachedServiceFetcher<AccessibilityManager>() {  
 @Override  
 public AccessibilityManager createService(ContextImpl ctx) {  
 return AccessibilityManager.getInstance(ctx);  
 }});  
  
 *registerService*(Context.*CAPTIONING\_SERVICE*, CaptioningManager.class,  
 new CachedServiceFetcher<CaptioningManager>() {  
 @Override  
 public CaptioningManager createService(ContextImpl ctx) {  
 return new CaptioningManager(ctx);  
 }});  
  
 *registerService*(Context.*ACCOUNT\_SERVICE*, AccountManager.class,  
 new CachedServiceFetcher<AccountManager>() {  
 @Override  
 public AccountManager createService(ContextImpl ctx) throws ServiceNotFoundException {  
 IBinder b = ServiceManager.getServiceOrThrow(Context.*ACCOUNT\_SERVICE*);  
 IAccountManager service = IAccountManager.Stub.asInterface(b);  
 return new AccountManager(ctx, service);  
 }});

。。。。。

}

*/\*\*  
 \* Statically registers a system service with the context.  
 \* This method must be called during static initialization only.  
 \*/*private static <T> void registerService(String serviceName, Class<T> serviceClass,  
 ServiceFetcher<T> serviceFetcher) {  
 *SYSTEM\_SERVICE\_NAMES*.put(serviceClass, serviceName);  
 *SYSTEM\_SERVICE\_FETCHERS*.put(serviceName, serviceFetcher);  
}

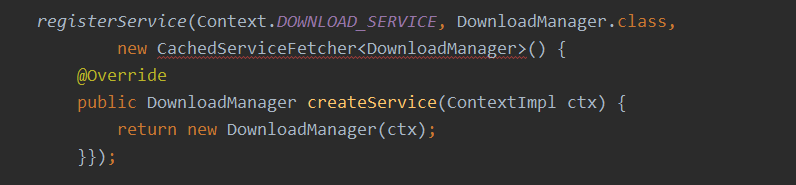
通过static静态初始化map集合，静态方法registerService写入服务，真正获取服务核心方法：

*IBinder b = ServiceManager.getServiceOrThrow(Context.VR\_SERVICE);*

*XXXX.Stub.asInterface(b)*

得到ServiceManager下的binder对象，使用不同Manager获取该binder引用，客户端调用方法，经过Manager包装，最终都是IPC 远程调用实现调用远程系统服务

2. 本地服务：静态注册模式



比如本地服务：DownloadManager

Activity/context调用getSystemService获取该manager实例，内部实现不通过Ibinder远程调用

### Service的AIDL原理

那么，客户端自定义Service的AIDL方式又是怎样实现的？

1. 服务端编写AIDL文件，build生成interface.sub文件

2. 客户端远程调用bindService，拿到客户端匹配的interface.sub文件，在系统绑定服务端Service的时候，系统会将内核空间android.os.Ibinder[binder驱动]返回客户端回调onServiceConnect获取的Ibinder需要通过interface.sub包装转换，此时调用asInterface方法，将通过interface.sub.proxy包装IBinder

3. 方法调用：

客户端调用使用binder

mRemote.transact(Stub.*TRANSACTION\_enableUserInfo*, \_data, \_reply, 0);

服务端通过Ibinder的interface.sub文件下方法操作onTransact解析参数并执行方法

@Override  
public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags)  
 throws android.os.RemoteException {  
 java.lang.String descriptor = *DESCRIPTOR*;  
 switch (code) {  
 case *INTERFACE\_TRANSACTION*: {  
 reply.writeString(descriptor);  
 return true;  
 }  
 case *TRANSACTION\_enableUserInfo*: {  
 data.enforceInterface(descriptor);  
 boolean \_arg0;  
 \_arg0 = (0 != data.readInt());  
 this.enableUserInfo(\_arg0);  
 reply.writeNoException();  
 return true;  
 }  
 default: {  
 return super.onTransact(code, data, reply, flags);  
 }  
 }  
}

4. 如果本地service定义使用本地AIDL，那么不存在远程调用

*/\*\*  
 \* Cast an IBinder object into an com.huawei.hms.adstest.ExSplashService interface,  
 \* generating a proxy if needed.  
 \*/*public static com.huawei.hms.adstest.ExSplashService asInterface(android.os.IBinder obj) {  
 if ((obj == null)) {  
 return null;  
 }  
 android.os.IInterface iin = obj.queryLocalInterface(*DESCRIPTOR*);  
 if (((iin != null) && (iin instanceof com.huawei.hms.adstest.ExSplashService))) {  
 return ((com.huawei.hms.adstest.ExSplashService) iin);  
 }  
 return new com.huawei.hms.adstest.ExSplashService.Stub.Proxy(obj);  
}

Proxy才是远程调用，本地定义server与本地使用调用的aidl下面asInterface只会返回当前aidl接口对象，即本地调用本地实现，不会走内核binder的transtract与ontranst

总结：

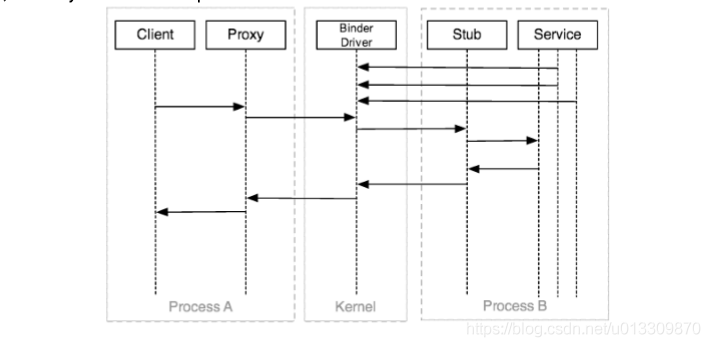
客户端使用的interface.sub通过asInterface生成proxy代理远程remote IBinder

客户端调用方法，mRemote.transact通过系统Binder驱动调用，系统内核空间传递参数给远程服务端，远程服务端通过ontransact解析调用方法

Binder通信的数据都是通过parcelable传递

自定义AIDL相比ServiceManager

自定义Service不会在Onstart中调用ServiceManager#addService添加Ibinder到系统内核数组下面长期持有，但是该Service在内核空间也会占用与客户端一对一管道



Proxy = remote Ibinder代理，调用方式：mRemote.transact

Stub为AIDL接口生成的继承Binder，解析方式ontransact调用方法

### 其他远程通信

Messenger：

通过IMessenger.Stub实现，

优点是不需要自定义aidl文件，传递消息使用Message，处理消息通过Handler

缺点是Handler处理消息，必须顺序执行，不能多线程

如何处理回调

Message.replyTo对象，指向客户端新产生的Messenger（Handler）对象

如下：

public Messenger(Handler target) {  
 mTarget = target.getIMessenger();  
}

@UnsupportedAppUsage  
final IMessenger getIMessenger() {  
 synchronized (mQueue) {  
 if (mMessenger != null) {  
 return mMessenger;  
 }  
 mMessenger = new MessengerImpl();  
 return mMessenger;  
 }  
}

private final class MessengerImpl extends IMessenger.Stub {  
 public void send(Message msg) {  
 msg.sendingUid = Binder.*getCallingUid*();  
 Handler.this.sendMessage(msg);  
 }  
}

所以：

Handler也可以使用Binder通信，内部拥有Imessager.sub实现，可以被Service初始化经过远程调用

android.app.usage

NetworkStatsManager：网络状态管理器（提供对网络使用历史和统计信息的访问）

StorageStatsManager：存储状态管理器（访问详细的存储统计信息）

UsageStatsManager：手机使用状态管理器（提供对设备使用历史记录和统计信息的访问）

## Android.app.Activity

### Context上下文对象

定义基本抽象方法

获取attributes方法

obtainAttributes：仅仅获取xml中对应属性，不带theme

obtainStyledAttributes：获取带默认theme下的xml属性，优先取重写属性，否则取默认属性

registerComponentCallbacks ：回调内存状态，如onLowMemory()当后台进程都已经被杀死才会调用，onTrimMemory调用频繁，系统采用LRU逐渐通知activity入参状态，activity根据入参状态及时清理内存，方法本身最终由Application实现，activity可以实现内部类调用自身清理方法

onTrimMemory等级：ComponentCallbacks2#常量

• TRIM\_MEMORY\_COMPLETE：内存不足，并且该进程在后台进程列表最后一个，马上就要被清理

• TRIM\_MEMORY\_MODERATE：内存不足，并且该进程在后台进程列表的中部。

• TRIM\_MEMORY\_BACKGROUND：内存不足，并且该进程是后台进程。

• TRIM\_MEMORY\_UI\_HIDDEN：内存不足，并且该进程的UI已经不可见了。

• TRIM\_MEMORY\_RUNNING\_CRITICAL：内存不足(后台进程不足3个)，并且该进程优先级比较高，需要清理内存

• TRIM\_MEMORY\_RUNNING\_LOW：内存不足(后台进程不足5个)，并且该进程优先级比较高，需要清理内存

• TRIM\_MEMORY\_RUNNING\_MODERATE：内存不足(后台进程超过5个)，并且该进程优先级比较高，需要清理内存

### ContextWrapper

使用attachBaseContext来代理Context方法实现，方法传入baseContext，默认实现为ContextImpl

### ContextThemeWrapper

带资源的Context，内部拥有Theme，核心类Resources产生Theme/获取AssetsManager

### Activity拆解[API 29]

1. registerActivityLifecycleCallbacks, unregisterActivityLifecycleCallbacks

添加Activity生命周期回调，支持

preXXXed[预回调],onXXXed[调用super.XXX回调],onActivityPostXXX[方法调用之后回调] XXX = Create,Start,Resume,Pause,Stop,Destory, SaveInstanceState

2. onCreate(Bundle savedInstanceState,PersistableBundle persistentState)

对比onCreate(Bundle savedInstanceState)

参数2 PersistableBundle方法调用时机：

当申明的Activity使用

android:persistableMode=“persistAcrossReboots|persistRootOnly|persistNever"

persistAcrossReboots：当手机重启/电源强制关机，系统会在下次重新进入该页面的时候调用onCreate(Bundle savedInstanceState,PersistableBundle persistentState)方法，并将PersistableBundle传入

关联方法：onSaveInstanceState(Bundle outState, PersistableBundle outPersistentState)当设备重启会调用onCreate传入

3. void reportFullyDrawn() 测量当前Activity从创建到该方法调用的时间间隔【暂无用，开发者自己计算】

4. 多窗口模式/画中画模式回调

5. getLastNonConfigurationInstance()

onRetainNonConfigurationInstance() 返回object对象

该方法返回数据不会随activity重建而数据丢失，在activity重建的时候保存该反方返回数据在内存中，待activity创建完成调用getLastNonConfigurationInstance()可以获取丢失的数据并重新初始化

onRetainNonConfigurationChildInstances 返回map对象

getLastNonConfigurationChildInstances

\*\*\* LiveData持久化就是使用该方法存储的ViewStore，老版本使用fragment#setRetainInstance方法存储，后来改为该方法存储

6. addContentView与setContentView区别

原理：PhoneWindow#decorView 【FrameLayout】 添加View到其布局下面，set为替换到View下面，ViewId = R.id.content

7. setContentTransitionManager getContentScene()

设置场景转换Manager，要求当前activity必须满足Window#FEATURE\_CONTENT\_TRANSITIONS 特性，否则不会将当前content作为Scene场景

这里涉及Scene场景相关特性【待补充】

8. setFinishOnTouchOutside

当activity被作为窗口弹框样式的时候，点击外部可以关闭该window

9. onKeyDown 调用onBackPressed方法原理【与onKeyUp连用】

public boolean onKeyDown(int keyCode, KeyEvent event) {  
 if (keyCode == KeyEvent.*KEYCODE\_BACK*) {  
 if (getApplicationInfo().targetSdkVersion  
 >= Build.VERSION\_CODES.*ECLAIR*) {  
 event.startTracking();  
 }

}

}

public boolean onKeyUp(int keyCode, KeyEvent event) {  
 if (getApplicationInfo().targetSdkVersion  
 >= Build.VERSION\_CODES.*ECLAIR*) {  
 if (keyCode == KeyEvent.*KEYCODE\_BACK* && event.isTracking()  
 && !event.isCanceled()) {  
 onBackPressed();  
 return true;  
 }  
 }  
 return false;  
}

退出是在onKeyDown+onKeyUp中触发的，表示是一次连续性动作

10. requestPermissions请求权限

public final void requestPermissions(@NonNull String[] permissions, int requestCode) {  
 if (requestCode < 0) {  
 throw new IllegalArgumentException("requestCode should be >= 0");  
 }  
 if (mHasCurrentPermissionsRequest) {  
 Log.*w*(*TAG*, "Can request only one set of permissions at a time");  
 // Dispatch the callback with empty arrays which means a cancellation.  
 onRequestPermissionsResult(requestCode, new String[0], new int[0]);  
 return;  
 }  
 Intent intent = getPackageManager().buildRequestPermissionsIntent(permissions);  
 startActivityForResult(*REQUEST\_PERMISSIONS\_WHO\_PREFIX*, intent, requestCode, null);  
 mHasCurrentPermissionsRequest = true;  
}

请求权限要求：

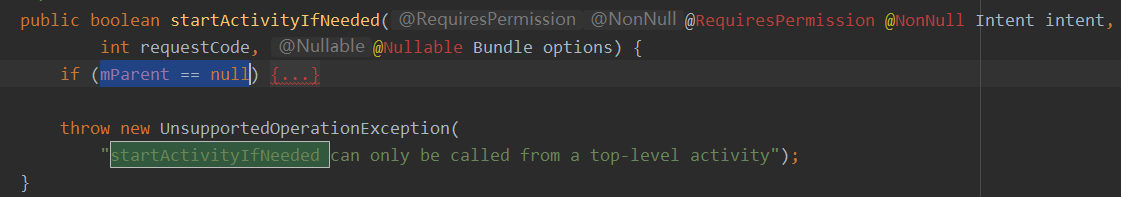
RequestCode > 0

当前如果已经有请求，那么直接返回取消重复请求

11. startActivityIfNeeded

该方法只能在顶层activity调用，如果该方法调用之后，如果存在已经有的栈底activity能够处理onNewIntent，那么该方法返回false，栈底activity不会接收intent，如果不存在已经有的activity栈内处理，该方法如果新开activity，将返回true，并且待新启动的activity处理完成返回时候，回调当前activity#onActivityResult

注意，该方法只能被 singleTask or singleTop 模式的activity使用，否则异常



12. \*\*\*startActivity流程解析：

如果有parent走startActivityFromChild会调用父parent activity【mInstrumentation.execStartActivity】，否则走当前Activity的mInstrumentation.execStartActivity

如果activity当前是可以运行在其他activity之上的，那么parent不为null（比如弹框样式的activity，默认parent为null）

*/\*\* Return the parent activity if this view is an embedded child. \*/*public final Activity getParent() {  
 return mParent;  
}

13. getLayoutInflater() 获取layout加载器

14. overridePendingTransition 页面打开/关闭动画

public void overridePendingTransition(int enterAnim, int exitAnim) {  
 try {  
 ActivityTaskManager.*getService*().overridePendingTransition(  
 mToken, getPackageName(), enterAnim, exitAnim);  
 } catch (RemoteException e) {  
 }  
}

API 29 由 ActivityManagerService变更为ActivityTaskManagerService

15. setResult 相关

setResult(int resultCode, Intent data) 设置结果

16. getCallingPackage() / getCallingActivity()

查询启动该activity的package名，原理通过ActivityTaskManagerService查询

17 isFinishing与 isDestory区别

调用了finish方法，ATMS调用finish返回之后可以查询，isDestory，当MainLooper执行了onDestory消息，activity本身执行performDestroy可以包装已经destory了

18 isChangingConfigurations 判断是否配置改变

该方法一般在onStop中判断，用于保存activity成员变量，方法：onRetainNonConfigurationInstance

19 recreate 重新创建该activity

该方法可能抛出异常，如果mparent非空[即activity嵌套]

20 finish(int finishTask)

关闭activity，入参类型：

*/\*\** ***@hide*** *Task isn't finished when activity is finished \*/*public static final int *DONT\_FINISH\_TASK\_WITH\_ACTIVITY* = 0;  
*/\*\*  
 \** ***@hide*** *Task is finished if the finishing activity is the root of the task. To preserve the  
 \* past behavior the task is also removed from recents.  
 \*/*public static final int *FINISH\_TASK\_WITH\_ROOT\_ACTIVITY* = 1;  
*/\*\*  
 \** ***@hide*** *Task is finished along with the finishing activity, but it is not removed from  
 \* recents.  
 \*/*public static final int *FINISH\_TASK\_WITH\_ACTIVITY* = 2;

结果该activity，如果是root，home可以保留/不保留最近任务

21. finishAffinity()

关闭具有相同affinity的activity，一般用于应用跳转，外部应用连续打开多个activity情况，用户点击返回可以直接返回自己的application

*/\*\*  
 \* Finish this activity as well as all activities immediately below it  
 \* in the current task that have the same affinity. This is typically  
 \* used when an application can be launched on to another task (such as  
 \* from an ACTION\_VIEW of a content type it understands) and the user  
 \* has used the up navigation to switch out of the current task and in  
 \* to its own task. In this case, if the user has navigated down into  
 \* any other activities of the second application, all of those should  
 \* be removed from the original task as part of the task switch.  
 \*  
 \* <p>Note that this finish does <em>not</em> allow you to deliver results  
 \* to the previous activity, and an exception will be thrown if you are trying  
 \* to do so.</p>  
 \*/*public void finishAffinity()

注意：该方法不可以用与返回activityResult，不能有mparent，resultCode，要求resultCode必须是RESULT\_CANCELED

22. finishAfterTransition

执行scene场景回退动画，执行完成之后调用finish关闭页面，如果不存在场景，直接finish

public void finishAfterTransition() {  
 if (!mActivityTransitionState.startExitBackTransition(this)) {  
 finish();  
 }  
}

23. finishActivity(int requestCode)

关闭请求的activity，当请求的activity调用startActivityForResult

24. \*\*\*createPendingResult

创建当前Activity的pendingIntent，用于回调activity的onActivityResult

使用场景：ActivityA跳转B【Activity/Service】跳转C…N，当B,C…N-1都关闭之后，可以在N众使用ActivityA传递的pendingIntent.send方法回调ActivityA#onActivityResult

举例，ActivityA:

PendingIntent pendingResult = createPendingResult(100, new Intent(), 0);  
Intent i = new Intent(this, MyService.class);  
i.putExtra(PENDING\_\_RESULT, pendingResult);  
startService(i);

Service:

PendingIntent pendingIntent = intent.getParcelableExtra(MainActivity.PENDING\_\_RESULT);  
try {  
 pendingIntent.send(MyService.this, 101,  
 new Intent().putExtra("data", "service返回数据"));  
} catch (CanceledException e) {  
 e.printStackTrace();  
}

25. setRequestedOrientation设置屏幕旋转方向

26. getRequestedOrientation() 获取屏幕旋转方向

27. getTaskId() 获取当前activity所在task栈的ID

static int getTaskForActivityLocked(IBinder token, boolean onlyRoot) {  
 final ActivityRecord r = ActivityRecord.*forTokenLocked*(token);  
 if (r == null) {  
 return INVALID\_TASK\_ID;  
 }  
 final TaskRecord task = r.task;  
 final int activityNdx = task.mActivities.indexOf(r);  
 if (activityNdx < 0 || (onlyRoot && activityNdx > task.findEffectiveRootIndex())) {  
 return INVALID\_TASK\_ID;  
 }  
 return task.taskId;  
}

原理：查询远程ATMS中activityRecord的所在任务栈Task id

28. isTaskRoot() 是否是栈底activity

29. moveTaskToBack(boolean nonRoot) 将activity置到后台

参数：nonRoot 是否是root activity，仅支持栈底activity，或者忽略，任意activity都可以

30. getSystemService 获取系统服务

通过ContextImpl调用SystemServiceResgity获取

31. runOnUiThread(Runnable action)

public final void runOnUiThread(Runnable action) {  
 if (Thread.*currentThread*() != mUiThread) {  
 mHandler.post(action);  
 } else {  
 action.run();  
 }  
}

运行在UI线程中，该方法如果在主线程中运行将同步调用，所以一般这边查询View状态得不到，所以要delay，因为是同步调用

32. boolean isImmersive() 是否是沉浸式activity，如果是，那么将不会被通知栏打扰

总结：ActivityTaskManager.getService().XXX【mToken】来调用远程服务

**\*\*\*ActivityStack , TaskRecord，ActivityRecord详解**

mToken详解：

1. 实现类：ActivityRecord#Token

static class Token extends IApplicationToken.Stub

2. 实例化时机：

Instrumentation#execStartActivity-> ATMS 【ActivityTaskManagerService】–> ActivityStarter#startActivity

3. ATMS创建新的ActivityRecord并将远程代理Ibinder交给Activity，用于Activity调用远程ATMS服务调用方法

\*\*\*这里讲到了ActivityRecord创建，那么Activity众ActivityStack与TaskRecord又是什么呢？

一个ActivityRecord对应一个Activity，保存了一个Activity的所有信息;但是一个Activity可能会有多个ActivityRecord,因为Activity可以被多次启动，这个主要取决于其启动模式。

一个TaskRecord由一个或者多个ActivityRecord组成，这就是我们常说的任务栈或者Task栈，具有后进先出的栈特征。

ActivityStack则是用来管理TaskRecord的，包含了一个或者多个TaskRecord，具有后进先出的栈特征。

ActivityStack包含TaskRecord，TaskRecord包含ActivityRecord

33. setEnterSharedElementCallback/ setExitSharedElementCallback 设置场景转换动画[SceneTransitionAnimation] 进入/退出回调监听

## android.app.Dialog

Dialog实现原理篇

Dialog是在Activity之上的一种弹框，类似一种简化的activity