# java

## 1.java引用类型

强引用：指创建一个对象并给一个引用变量

Object object = new Object();

强引用有引用变量指向时永远不会被垃圾回收，JVM宁愿抛出OutOfMemory错误也不会回收这种对象

软引用：如果一个对象具有软引用，内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用，通过SoftReference类来实现

弱引用：是用来描述非必需对象的，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，通过weakReference类来实现

虚引用：虚引用和前面的软引用、弱引用不同，它并不影响对象的生命周期。在java中用java.lang.ref.PhantomReference类表示。如果一个对象与虚引用关联，则跟没有引用与之关联一样，在任何时候都可能被垃圾回收器回收。要注意的是，虚引用必须和引用队列关联使用，当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会把这个虚引用加入到与之 关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动

## 2.java线程池

### 2.1java中的ThreadPoolExecuter类

该类是线程池中最核心的一个类，含有四个构造函数，

public class ThreadPoolExecutor extends AbstractExecutorService {

    public ThreadPoolExecutor(**int** **corePoolSize**,**int** **maximumPoolSize**,**long** **keepAliveTime**,**TimeUnit** **unit**,**BlockingQueue**<Runnable> **workQueue**);

    public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit, BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory);

    public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit, BlockingQueue<Runnable> workQueue,RejectedExecutionHandler handler);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit, BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler);}

该类继承了AbstractExecutorService类，提供四个构造器；

corepoolSeze是指核心池大小,默认情况下，线程池中没有任何线程，而是等待有任务到来才去创建线程执行任务，除非调用prestartAllCoreThreads()或者prestartCoreThread()这两个方法，是在任务到来之前就创建corePoolSize个线程(预创建线程)；默认情况下创建了线程池后线程数量为0，当有任务来之后，就会创建一个线程去执行任务；当线程池中数目达到corePoolSize后，就会把到达的任务放进缓存队列中

maximumPoolSize是指最大线程数，表示线程池中最大能创建多少个线程

keepAliveTime:表示线程没有执行任务是最多保持多久会终止，当线程池线程数大于corePoolSize时触发

unit:时参数keepAliveTime的单位

workQueue：一个阻塞队列，用来存储等待执行的任务

threadFactory：线程工厂

handler：表示当拒绝处理任务时的策略，有四个取值

【1】.ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

【2】ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

【3】ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

【4】ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

ThreadPoolExcutor继承AbstractExecutorService抽象类, AbstractExecutorService实现ExecutorService接口，ExecutorService继承Executor接口

在**ThreadPoolExecutor**类中有几个非常重要的方法：

execute()实际上是Excutor中申明的方法，在ThreadPoolExecutor得到了具体实现，通过这个方法可以向线程池提交一个任务，交由线程池去实现，物返回结果

execute()

submit()

shutdown()

shutdownNow()

submit()也是用来向线程池提交任务，有返回结果

shutdown()和shutdowNow()是用来关闭线程池的，其中

前者将将线程池设置为SHUTDOWN状态，未执行完的任务依然会执行下去，后者线程池处于STOP状态，正在执行的任务会被停止

### 2.2深入线程池实现原理

#### 2.2.1线程池状态

在ThreadPoolExecutor中定义了一个volatile变量，另外定义了几个static final变量表示线程池的各个状态

runState表示当前线程池的状态，是

volatile int runState;

static final int RUNNING = 0;

static final int SHUTDOWN = 1;

static final int STOP = 2;

static final int TERMINATED = 3;

一个volatile变量用来保证线程之间的

可见性

几个final变量是runState可能取得值

当线程处于SHUTDOWN或者STOP状态，并

且所有工作线程已经被销毁，任务缓存队列已经清空货执行结束后，线程池被设置为TERMINATED状态

private final BlockingQueue<Runnable> workQueue; //任务缓存队列，用来存放等待执行的任务

private final ReentrantLock mainLock = new ReentrantLock();//线程池的主要状态锁，对线程池状态（比如线程池大小、runState等）的改变都要使用这个锁

private final HashSet<Worker> workers = new HashSet<Worker>();//用来存放工作集

private volatile long  keepAliveTime;    //线程存货时间

private volatile boolean allowCoreThreadTimeOut;   //是否允许为核心线程设置存活时间

private volatile int   corePoolSize;     //核心池的大小（即线程池中的线程数目大于这个参数时，提交的任务会被放进任务缓存队列）

private volatile int   maximumPoolSize;   //线程池最大能容忍的线程数

private volatile int   poolSize;       //线程池中当前的线程数

private volatile RejectedExecutionHandler handler; //任务拒绝策略

private volatile ThreadFactory threadFactory;   //线程工厂，用来创建线程

private int largestPoolSize;   //用来记录线程池中曾经出现过的最大线程数

private long completedTaskCount;   //用来记录已经执行完毕的任务个数

#### 2.2.2excute实现代码

Execute源代码

public void execute(Runnable command) {

    if (command == null)

        throw new NullPointerException();

    if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {

        if (runState == RUNNING && workQueue.offer(command)) {

            if (runState != RUNNING || poolSize == 0)

                ensureQueuedTaskHandled(command);

        }

        else if (!addIfUnderMaximumPoolSize(command))

            reject(command); // is shutdown or saturated

    }

}

addIfUnderCorePoolSize()

private boolean addIfUnderCorePoolSize(Runnable firstTask) {

    Thread t = null;

    final ReentrantLock mainLock = this.mainLock;

    mainLock.lock();

    try {

        if (poolSize < corePoolSize && runState == RUNNING)

          t = addThread(firstTask);//创建线程去执行firstTask任务

        } finally {

        mainLock.unlock();

    }

    if (t == null)

        return false;

    t.start();

    return true;

}

addIfUnderMaximumPoolSize()

private boolean addIfUnderMaximumPoolSize(Runnable firstTask) {

    Thread t = null;

    final ReentrantLock mainLock = this.mainLock;

    mainLock.lock();

    try {

        if (poolSize < maximumPoolSize && runState == RUNNING)

            t = addThread(firstTask);

    } finally {

        mainLock.unlock();

    }

    if (t == null)

        return false;

    t.start();

    return true;

}

}

AddThread()

private Thread addThread(Runnable firstTask) {

    Worker w = new Worker(firstTask);

    Thread t = threadFactory.newThread(w);  //创建一个线程，执行任务

    if (t != null) {

        w.thread = t;       //将创建的线程的引用赋值为w的成员变量

        workers.add(w);

        int nt = ++poolSize;     //当前线程数加1

        if (nt > largestPoolSize)

            largestPoolSize = nt;

    }

    return t;

}

1）首先，要清楚corePoolSize和maximumPoolSize的含义；

2）其次，要知道Worker是用来起到什么作用的；

3）要知道任务提交给线程池之后的处理策略，这里总结一下主要有4点：

如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；

如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；

如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；

如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

## 2.3java并发编程之辅助类

* CountDownLatch

利用该类可以实现类似计数器的功能，其只提供了一个构造器，主要方法包括await()和countDone(),即等计数器为0放进行await()后面的操作

不可被重用

* CyclicBarrier

当前所有线程都被释放后，所有线程就继续后续操作；其含有两个构造函数

**public** CyclicBarrier(**int** parties) {

**this**(parties, **null**);

}

**public** CyclicBarrier(**int** parties, Runnable barrierAction) {

**if** (parties <= 0) **throw** **new** IllegalArgumentException();

**this**.parties = parties;

**this**.count = parties;

**this**.barrierCommand = barrierAction;

}

parties是指制定多少个线程进入等待，barrierAction是指当这些线程都达到Barrier状态时会执行的内容

其最重要的方法就是await()和await(long timeout,TimeUnit unit)

* Semaphore

可以控制同时访问的线程个数，通过acquire()获取一个许可，

## 3.java并发编程之ConCurrentHashMap

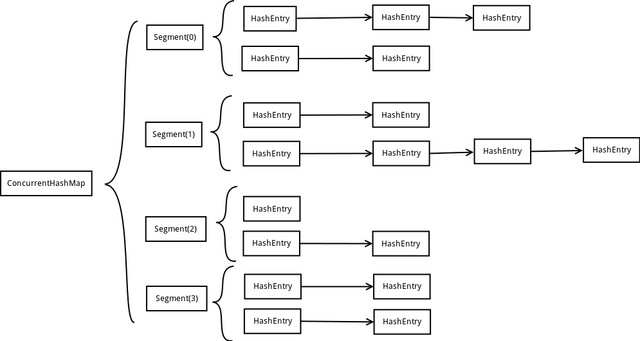
Jdk5中添加了新的concurrent包，相对容器而言，并发容器通过一些机制改近了并发性能；因为同步容器将所有对容器状态的访问都串行化了，这样保证了线程的安全，单严重降低了并发性；当多个线程竞争容器时，吞吐量严重降低。和Vector和HashTable相比，concurrent主要解决了两个问题：

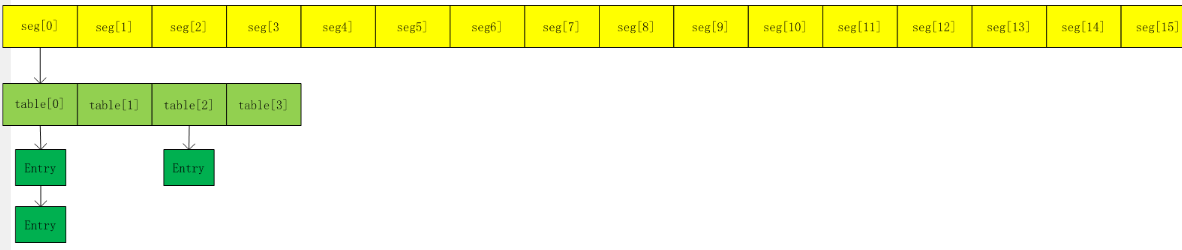
1.更具具体场景进行设计，尽量避免synchronized,提供并发性

2.定义了一些并发安全的复合操作，保证开发环境下的迭代操作不会出错

### 3.1concurrentHashMap内部结构

ConcurrentHashMap为了提高本身的并发能力，在内部采用了一个叫做Segment的结构，一个Segment其实就是一个类HashTable的结构，Segment内部维护了一个链表数组





ConcurrentHashMap定位一个元素的过程需要进行两次Hash操作，第一次Hash定位到Segment,第二次定位到元素所在的链表头部，弊端hash过程加长；好处是读写的时候可以只对元素所在的Segment进行加锁即可，不会影响到其他Segment,在最理想的情况下，ConcurrentHashMap可以最高同时支持Segment数量大小的写操作，大大提高并发能力

### 3.2.Segment数据结构

#### 3.2.1数据结构

statistatic final class Segment<K,V> extends ReentrantLock implements Serializable {

    transient volatile int count;

    transient int modCount;

    transient int threshold;

    transient volatile HashEntry<K,V>[] table;

    final float loadFactor;

}

c final class Segment<K,V> extends ReentrantLock implements Serializable {

    transient volatile int count;

    transient int modCount;

    transient int threshold;

    transient volatile HashEntry<K,V>[] table;

    final float loadFactor;

}

count:Segment中元素的数量

modCount:对table的大小造成影响的操作的数量（比如put或者remove操作）

threshold:阈值，Segment里面元素的数量超过这个值依旧就会对Segment进行扩容

table:链表数组，数组中每一个元素代表了一个链表的头部

loadFactor：负载因子，用于确定threshold

#### 3.2.2HashEntry数据结构

static final class HashEntry<K,V> {

    final K key;

    final int hash;

    volatile V value;

    final HashEntry<K,V> next;

}

### 3.3ConcurrentHashMap的初始化

public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel) {

    if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)

        throw new IllegalArgumentException();

    if (concurrencyLevel > MAX\_SEGMENTS)

        concurrencyLevel = MAX\_SEGMENTS;

    // Find power-of-two sizes best matching arguments

    int sshift = 0;

    int ssize = 1;

    while (ssize < concurrencyLevel) {

        ++sshift;

        ssize <<= 1;// ssize = ssize << 1 , ssize = ssize \* 2

    }

    segmentShift = 32 - sshift;

    segmentMask = ssize - 1;

    this.segments = Segment.newArray(ssize);

    if (initialCapacity > MAXIMUM\_CAPACITY)

        initialCapacity = MAXIMUM\_CAPACITY;

    int c = initialCapacity / ssize;

    if (c \* ssize < initialCapacity)

        ++c;

    int cap = 1;

    while (cap < c)

        cap <<= 1;

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

        this.segments[i] = new Segment<K,V>(cap, loadFactor);

}

初始化一共有三个参数，initialCapacity表示出事的容量,loadFactor表示负载参数，concurrentLevel代表ConcurrentHashMap内部的Segment数量，concurrentLevel已经指定不可改变；后续concurrentHashMap的元素数量导致其需要扩容，concurrentHashMap不会增加Segment的数量，只会增加Segment中链表数组的容量大小（不需要对整个concurrentHashMap做rehash,而只会对Segment里面的元素做一次rehash）

假设构造函数确定了Segment的数量是2的n次方，俺么segmentShift就等于32减去n,而segmentMask就等于2的n次方减一

### 3.4get操作

public V get(Object key) {

    int hash = hash(key.hashCode());

    return segmentFor(hash).get(key, hash);

}

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

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

}

可以看出，get操作并未加锁

SegmentFor这个函数几乎对ConcurrentHashMap的所有操作都会用到这个函数，根据传入的hash值向右无符号右移segmrntShift位，然后和segmentMask进行与操作；假设Segment的数量是2的n次方，根据元素的hash值得高n位就可以确定元素到底在哪一个Segment中

在确定了在哪一个Segment中后，调用对应的Segment的get方法：

V get(Object key, int hash) {

    if (count != 0) { // read-volatile

        HashEntry<K,V> e = getFirst(hash);

        while (e != null) {

            if (e.hash == hash && key.equals(e.key)) {

                V v = e.value;

                if (v != null)

                    return v;

                return readValueUnderLock(e); // recheck

            }

            e = e.next;

        }

    }

    return null;

}

count表示Segment元素的数量，如下定义

transient volatile int count;

可以看到count是volatile，保证了写操作对后续的读操作都是可见的，这样后面的get的后续操作就可以拿到完整的元素内容

在第三行，确定了链表的头部，同样，这里也是用未操作来确定链表的头部，hash值和hashTable的长度减一做与操作，最后的结果就是hash值得低n位，其中n是HashTable的长度以2位底的结果

HashEntry<K,V> getFirst(int hash) {

    HashEntry<K,V>[] tab = table;

    return tab[hash & (tab.length - 1)];

}

### 3.5put操作

//ConcurrentHashMap#put

public V put(K key, V value) {

　　Segment<K,V> s;

　　if (value == null)

　　　　throw new NullPointerException();

　　int hash = hash(key);//根据散列函数，计算出key值的散列值

　　int j = (hash >>> segmentShift) & segmentMask;//这个操作就是定位Segment的数组下标，jdk1.7之前是segmentFor返回Segment，1.7之后直接就取消了这个方法，直接计算数组下标，然后通过偏移量底层操作获取Segment

　　if ((s = (Segment<K,V>)UNSAFE.getObject // nonvolatile; recheck

　　　　　　(segments, (j << SSHIFT) + SBASE)) == null) // in ensureSegment

　　　　s = ensureSegment(j);//通过偏移量定位不到就调用ensureSegment方法定位Segment

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

}

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

    // 在往该 segment 写入前，需要先获取该 segment 的独占锁

    //    先看主流程，后面还会具体介绍这部分内容

    HashEntry<K,V> node = tryLock() ? null :

        scanAndLockForPut(key, hash, value);

    V oldValue;

    try {

        // 这个是 segment 内部的数组

        HashEntry<K,V>[] tab = table;

        // 再利用 hash 值，求应该放置的数组下标

        int index = (tab.length - 1) & hash;

        // first 是数组该位置处的链表的表头

        HashEntry<K,V> first = entryAt(tab, index);

// 下面这串 for 循环虽然很长，不过也很好理解，想想该位置没有任何元素和已经存在一个链表这两种情况

        for (HashEntry<K,V> e = first;;) {

            if (e != null) {

                K k;

                if ((k = e.key) == key ||

                    (e.hash == hash && key.equals(k))) {

                    oldValue = e.value;

                    if (!onlyIfAbsent) {

                        // 覆盖旧值

                        e.value = value;

                        ++modCount;

                    }

                    break;

                }

                // 继续顺着链表走

                e = e.next;

            }

            else {

       // node 到底是不是 null，这个要看获取锁的过程，不过和这里都没有关系。

       // 如果不为 null，那就直接将它设置为链表表头；如果是null，初始化并设置为链表表头。

                if (node != null)

                    node.setNext(first);

                else

                    node = new HashEntry<K,V>(hash, key, value, first);

                int c = count + 1;

                // 如果超过了该 segment 的阈值，这个 segment 需要扩容

                if (c > threshold && tab.length < MAXIMUM\_CAPACITY)

                    rehash(node); // 扩容后面也会具体分析

                else

                    // 没有达到阈值，将 node 放到数组 tab 的 index 位置，

                    // 其实就是将新的节点设置成原链表的表头

                    setEntryAt(tab, index, node);

                ++modCount;

                count = c;

                oldValue = null;

                break;

            }

        }

    } finally {

        // 解锁

        unlock();

    }

    return oldValue;

}

获取写入锁：scanAndLockForPut

private HashEntry<K,V> scanAndLockForPut(K key, int hash, V value) {

    HashEntry<K,V> first = entryForHash(this, hash);

    HashEntry<K,V> e = first;

    HashEntry<K,V> node = null;

    int retries = -1; // negative while locating node

    // 循环获取锁

    while (!tryLock()) {

        HashEntry<K,V> f; // to recheck first below

        if (retries < 0) {

            if (e == null) {

                if (node == null) // speculatively create node

                    // 进到这里说明数组该位置的链表是空的，没有任何元素

                    // 当然，进到这里的另一个原因是 tryLock() 失败，所以该槽存在并发，不一定是该位置

                    node = new HashEntry<K,V>(hash, key, value, null);

                retries = 0;

            }

            else if (key.equals(e.key))

                retries = 0;

            else

                // 顺着链表往下走

                e = e.next;

        }

        // 重试次数如果超过 MAX\_SCAN\_RETRIES（单核1多核64），那么不抢了，进入到阻塞队列等待锁

        //    lock() 是阻塞方法，直到获取锁后返回

        else if (++retries > MAX\_SCAN\_RETRIES) {

            lock();

            break;

        }

        else if ((retries & 1) == 0 &&

                 // 这个时候是有大问题了，那就是有新的元素进到了链表，成为了新的表头

                 //     所以这边的策略是，相当于重新走一遍这个 scanAndLockForPut 方法

                 (f = entryForHash(this, hash)) != first) {

            e = first = f; // re-traverse if entry changed

            retries = -1;

        }

    }

    return node;

}

## 4.java锁

### 4.1synchronized缺陷

Synchronized是java的一个关键字，设计java语言内置的特性，当一个线程获取了对应的锁，其他线程便只能一直等待虎丘锁的线程释放锁，其只会有两种情况释放锁：

1）获取锁所得线程执行完代码

2）线程发生异常，有jvm释放

弊端：

1.如果获取锁的线程由于要等待IO或者其他原因被阻塞，但是又没有释放锁，其他线程只能是处于无期限的等待状态

2.如只读时允许多个线程同时读等无法实现

3.锁的机制无法人工干预

其弊端通过lock可以实现

|  |  |
| --- | --- |
| Synchroized | lock |
| 不需要手动释放 | 需要手动释放，且位于finally里面 |
| 是java的一个关键字 | 是java的一个类 |
| 可重入、不可中断、非公平 | 可重入、可判断、公平 |
| 少量同步 | 大量同步 |

## 4.2java.util.concurrent.locks包下常用的类

### 4.2.1Lock

Lock是一个接口，含有4个获取锁的方法

public interface Lock {

    void lock();

    void lockInterruptibly() throws InterruptedException;

    boolean tryLock();

    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;

    void unlock();

    Condition newCondition();

}

lock必须在try{}catch{}块中进行，并且释放锁的操作是在finally块中进行的，以保证锁一定被释放，防止死锁的发生

tryLock()有返回值true OR false在拿不到锁的情况下不会一直等

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，这个线程能够响应中断，即中断线程的等待状态。也就是说，当两个线程同时通过lock. lockInterruptibly()方法向获取某个锁时，假设A取得了锁，而B线程只能等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程；由于lockInterruptibly的申明中抛出了异常，所以lock. lockInterruptibly()必须在try{}块中或者在lockInterruptibly()的方法外生命抛出InterruptedException

注：当一个线程获取锁后是不能通过interrupt()方法终端的，该方法只针对于正在等待的线程

### 4.2.2ReentrantLock(可重入锁)

关于重入锁：

class MyClass {

    public synchronized void method1() {

        method2();

    }

    public synchronized void method2() {

    }

}

如method1中调用另外一个synchronized方法method2,此时线程不需要重新去申请锁，而是可以直接执行方法method2

### 4.2.3****ReadWriteLock****

ReadWriteLock也是一个接口，含有两个方法

public interface ReadWriteLock {

    /\*\*

     \* Returns the lock used for reading.

     \*

     \* @return the lock used for reading.

     \*/

    Lock readLock();

    /\*\*

     \* Returns the lock used for writing.

     \*

     \* @return the lock used for writing.

     \*/

Lock writeLock();

}

一个用来获取读锁，一个用来获取写锁，也就是说将文件的读写操作分开，分成2个锁来分配给线程，从而多个线程可以同时进行读操作，**ReentrantReadWriteLock**是ReadWriteLock的实现类；主要也是那两个方法的实现；如果有一个线程已经占用了读锁，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；如果一个线程已经占用了写锁，此时其他线程如果需要申请读锁或者写锁，则申请的线程会一直等待释放写锁。

## 5.hash表

Hash函数：采用函数映射的思想将记录的存储位置与记录的关键字关联起来，从而能够快速进行定位查找；可以直接根据关键字和映射关系计算出该记录在表中得存储位置，这种映射关系被称为hash函数，而通过Hash函数和关键字计算出来的存储位置称为hash地址

### 5.1构造hash函数的方法

1）直接定址法：取关键字获关键字的某个现行函数为Hash地址

2)平均取中法

3）折叠法

4）除留取余法

#### 5.1.2确定hash表大小

Hash表大小的确定非常关键，如果Hash表的空间远远大于最后实际存储的巨鹿个数，则造成了很大的空间浪费，如果取小了的话，容易赵成冲突；在实际情况中需要更具最终记录存储个数和关键字的分布特点来确定Hash表大小

#### 5.1.2解决hash冲突

1）开放地址法

当一个关键字和另一个关键字发生冲突时，使用某种探测技术在hash表中星城一个探测序列，然后沿着这个探测序列一次查找下去，当碰到一个空的单元时，则插入其中

2）链地址法

采用数组和链表相结合的办法，将hash地址相同的记录存储在同一张线性表中，而每张表的表头的序号即为得到的Hash地址

冲突时无法避免的，需要根据实际情况选取解决冲突的办法

## 6.java NIO

### 6.1java i/o模型

#### 6.1.1同步和异步

同步：如果有多个任务或者事件发生，这些时间可以并发的执行；一个事件或者任务的执行会导致整个流程的暂时等待，这些事件没有办法并行执行

异步：多个任务或者时间发生时，这些事件可以同时执行

#### 6.1.2阻塞和非阻塞

阻塞：当某个事件或者任务正在执行的过程中，结果返回前线程会被挂起，一直在哪里等待直至满足条件

非阻塞：及时条件未得到满足，不会一直处于等待状态

#### 6.1.3阻塞IO和非阻塞IO

通常来说，IO操作包括对硬盘的读写、对sockert的读写以及外设的读写；

当一个用户发起一个IO请求操作时，内核回去查看要去读取的数据是否就绪，对于阻塞IO来说，如果数据没有就绪，则会一直等待在那里；对于非阻塞IO而言，如果数据未就绪，则返回一个标志新年西告知用户线程当前要的数据未就绪，当数据就绪后，将数据拷贝到用户线程，完成一个完整的IO读请求操作；其包含了两个步骤：

1.查看数据是否就绪

2.进行数据拷贝

Java传统的IO都是阻塞IO，如socket读取数据

#### 6.1.4什么是同步IO、什么是异步IO

同步IO和异步是针对用户线程和内核的交互来说的

同步IO，当用户发出IO请求之后，如果数据未就绪，需要通过用户线程或者内核不断去轮询数据是否就绪，直至数据就绪完成拷贝

异步IO，只有IO请求操作的发出时有用户线程来进行的，IO操作的两个阶段都是有内核自动完成，然后发送通知告知用户线程IO操作已经完成。也就是说在异步IO中不会造成用户线程的阻塞

其关键之处在于数据拷贝阶段是由线程完成还是内核完成；因此异步必须有操作系统支持

注意6.1.1、6.1.2、6.1.3、6.1.4概念上的区别

### 6.2javaNIO

#### 6.2.1NIO的几个基本概念

Channel（通道）、Buffer(缓冲区)、Selector(选择器)

public class Test {

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

        File file = new File("data.txt");

        InputStream inputStream = new FileInputStream(file);

        byte[] bytes = new byte[1024];

        inputStream.read(bytes);

        inputStream.close();

    }

}

1.chanel

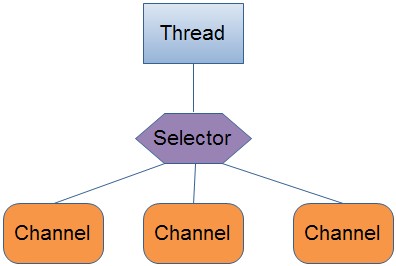
如InputStream/OutputStream实际上就是读取文件提供的一个通道，一次可以将NIO中的Channel同传统IO中的Stream来类比，但注意的是传统IO Stream是单向的，Channel是双向的

2.Buffer

是NIO中非常重要的东西，NIO中的所有数据读和写都离不开Buffer，如上代码读取数据时放在byte数组中，而在NIO中，读写的数据只能放在Buffer中

3.Selector

是NIO中最关键的一个部分，Selector的作用就是用来轮询每个注册的Channel，一旦发现Channel有注册的事情发生，变获取时间然后进行处理



#### 6.2.2Channel

常用的几种通道：

FileChannel -- 文件

SocketChanel -- TCP

ServerSocketChannel -- 能够监听客户端发起的TCP链接

DatagramChannel -- UDP协议

Example:

public class Test {

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

        File file = new File("data.txt");

        FileOutputStream outputStream = new FileOutputStream(file);

        FileChannel channel = outputStream.getChannel();

        ByteBuffer buffer = ByteBuffer.allocate(1024);

        String string = "java nio";

        buffer.put(string.getBytes());

        buffer.flip();     //此处必须要调用buffer的flip方法

        channel.write(buffer);

        channel.close();

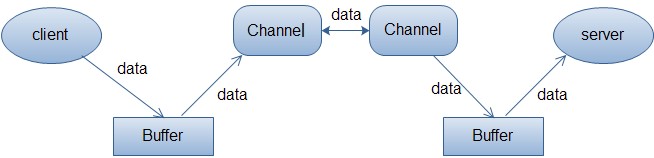
        outputStream.close();

    }

}

#### 6.2.3Buffer

Buffer是一个顶层的父类，实际上是一个容器，是一个连续的数组。Channel提供从文件、网络读取数据的渠道，但是读取或写入的数据都必须经有Buffer



常用的子类

ByteBuffer

IntBuffer

CharBuffer

LongBuffer

DoubleBuffer

FloatBuffer

ShortBuffer

如果是文件的读写，其上的几种均会用到；但是对于网络读写来说，用的最多的是ByteBuffer

# 7.java基础

## 7.1java深拷贝和浅拷贝

实现基本数据类型的拷贝：Java实现一个基本数据类型的拷贝只需要赋值即可

实现一个对象的拷贝：1.被复制对象实现Cloneable接口

2.覆盖clone方法并将权限改为public

浅拷贝：指只实现本类的clone,对于注入类没有实现

深拷贝：所有类都实现clone方法，并且在被拷贝的clone方法里实现注入类的拷贝调用

## 7.2final关键字

* final修饰的类不可被继承
* final修饰的变量不可被改变
* final修饰的对象不可指向其他值

## 7.3内部类

### 7.3.1成员内部类

* 是最普通的内部类，定义域=于一个类的内部，内部类可以访问外部内所有成员（包括private和静态成员）
* 当内部类成员与外部类相同时，改成员默认为内部类，若要访问外部类的成员，需要显式的使用**外部类.this.成员**进行访问
* 成员内部类是依附外部类存在的，如果需要创建成员内部类对象，需要通过外部类如**Outter.Inner inner = outter.new Inner();**

进行创建

* 如果需要继承内部类，则必须再构造函数里显示的调用外部类的构造器

### 7.3.2局部内部类

局部内部类是定义在一个方法或者一个作用域里面的类，和成员内部类的区别在于访问仅限于方法或者作用域内

### 7.3.3匿名内部类

* 匿名内部类不能用static修饰和访问修饰
* 匿名内部类没有构造器，适用于接口i回调

### 7.3.4静态内部类

是static修饰的成员内部类，其不需要依附于外部类，也不能使用外部类的非static成员

### 7.4装箱与拆箱

### 7.4.1装箱与拆箱的概念

拆箱:自动将包装器类型转化为基本类型

装箱：自动将基本类型转换为包装类型

### 7.4.2应用

INT:-128-127

Boolean

## 7.5java的异常处理和设计

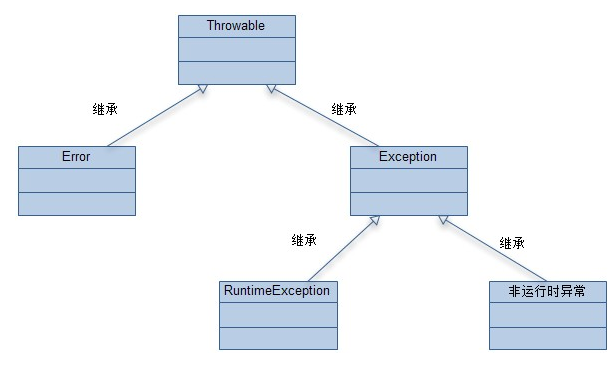
### 7.5.1异常

在java中异常被当作对象来处理，根类是java.lang.Rhrowable类，在java中异常分为两大类，Exception和Error

Error是无法处理的异常，如OutOfMemoryError,一般发生这种异常，JVM会选择终止程序，开发时无需关心这种异常

Exception是常见的异常，如NullPointerException、IndexOutOfBoundsException等，这些异常时我们可以处理的异常；Exception类包括检测到的异常和无法检测的异常（运行时异常）；

检查的异常如IOException、SQLException,对于非运行时异常如果不进行捕获或者抛出，编译都不会通过



典型的运行时异常如：NullPointerException、IndexOutOfBoundsException、IllegalArgumentException等

### 7.5.2try，catch,finally.throws.throw五个关键字

try,catch,finally：……………可有多个catch

throws出现在方法的申明中

throw出现在方法体中

原则：异常处理精良放在高层进行，在finally中释放资源，避免多次在日志中记录同一个异常等

# 8.java垃圾回收机制

## 8.1确定对象是“垃圾

* 引用技术发法

在java中通过引用和对象进行关联，如果要操作对象必须通过应用来操作；很显然一个简单的方法就是通过引用计数来判断一个对象是否可以被回收，不是一般性，如果一个对象没有任何引用与之关联，则说明该对象不大可能被使用到，即会被回收；这种方式实现简单而且效率高；但是无法解决循环应用的问题，如

public class Main {

    public static void main(String[] args) {

        MyObject object1 = new MyObject();

        MyObject object2 = new MyObject();

        object1.object = object2;

        object2.object = object1;

        object1 = null;

        object2 = null;

    }

}

class MyObject{

    public Object object = null;

}

* 可达性分析法

通过一系列的GC Roots的对象作为起始点，从这些根节点向下搜索，搜索走过的路劲称为引用链，当一个对象到GC Root没有任何引用链时，证明此对象时不可用的

## 8.2典型的垃圾回收算法

* Mark-Sweep算法

这是最基础的垃圾回收算法，标记阶段的任务时标记出需要被回收的对象，清楚阶段就是回收被标记的对象所占用的空间，容易产生内存碎片

* Copying算法

将可用内存按容量划分为大小相等的两块，每次只使用其中一块，当这一块的内存用完了，就将还或者的对象复制到另一块上面，然后再把已使用的内存空间一次清理掉，这样一来就不会出现内存碎片；但对内存空间的使用做出了高昂的代价

* Mark-Compackt算法

该算法标记阶段和Mark-Sweep算法一样，但在完成标记之后，不死直接清理可回收对象，而是将存活对象都向一端移动，然后清理调端边界以外的内存

* 分代搜集算法

分代收集是目前大部分JVM的垃圾收集器都采用的算法，其核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域，一般情况下将堆区划分为老年代和新生代，老年代的特点是每次收集垃圾是只有少量对象需要被回收，而新生代的特点是每次垃圾回收是都有大量对象需要被回收，那么就可以根据不同代的特点采取最合适的收集算法

目前大部分垃圾回收器对于新生代都采用Copying算法，因为新生代中每次垃圾回收都要回收大部分对象，也就是说需要复制的操作次数较少；但是实际中是将新生代划分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden空间和其中的一块Survivor空间，当进行回收时，将Eden和Survivor中还存活的对象复制到另一块SurVivor空间中，然后清理掉Eden和刚才使用过的Survivor空间

老年代的特点就是每次回收都只回收少量对象，一般使用Mark-Compact算法

# 9.java注解

**注解**：也称为元数据；为我们在代码中添加信息提供了一种形式化的方法；是我们在在稍后有的某个时刻非常方便的使用这些数据；是java5引入的语言重要变化之一

如：

**package** com.zy.annatate;

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

@Target(ElementType.***METHOD***)

@Retention(RetentionPolicy.***RUNTIME***)

**public** **@interface** Test{}

这种没有元素的注解成为标记注解；@Target用来定义注解将应用于什么地方（如是一个方法或者是一个域），@Retention用来定义注解在哪一个级别可用，在远代码中（SOURCE）、类文件中（CLASS）、或者运行时（RUNTIME）

在注解中，一般都会包含一些元素以及某些值，在分析处理注解时，程序或者工具可以利用这些值，注解元素看起来就像接口的方法，唯一的区别是可以指定默认值

如

**package** com.zy.annatate;

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

@Target(ElementType.***METHOD***)

@Retention(RetentionPolicy.***RUNTIME***)

**public** **@interface** UseCase{

**public** **int** id();

**public** String description() **default** "no description";

}

id和description类似方法定义，编译器会对id进行类型检查，descrip可以设置一个默认值

## 9.1元注解

Java内置三种标准注解@Override、@Deprecated、@SuppressWarning和四种注解元素；元注解专职负责注解其它注解

* @Target 表示注解可以用在什么地方可选的**ElementType**参数包括

1. CONSTRUCTOR:构造器申明
2. FIELD:域声明
3. LOCAL\_VARIABLE:局部变量声明
4. METHOD:方法申明
5. PACKAGE:报申明
6. PARAMETER:参数申明
7. TYPE：类、接口

* @Retention 表示需要在什么级别保存改注解信息,可选的RetentionPolicy参数包括

1. SOURCE：注解江北编译器丢弃
2. CLASS：注解子class文件中可用，但会被VM丢弃
3. RUNTIME:VM将在运行中也保留注解，已就此可以通过反射机制读取注解的信息

* @Documented将此注解包含着在Javadoc中
* @Inherited允许自雷继承父类中的注解

## 9.2编写注解处理器

在使用注解过程中，很重要的一个部分就是创建与使用**注解处理器**；如

**package** com.zy.annatate;

**import** java.lang.reflect.Method;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** UseCaseTracker {

**public** **static** **void** tracker(List<Integer> ids, Class<?> clazz) {

**int** size = ids.size();

Method[] methods = clazz.getDeclaredMethods();

**for**(Method method : methods) {

Usecase uc = method.getDeclaredAnnotation(Usecase.**class**);

**if**(ids.contains(uc.id())) {

System.***out***.println("catch: " +uc.id() + uc.description());

}

}

}

**public** **static** **void** main(String[] args) {

List ids = **new** ArrayList<>();

ids.add(1);

ids.add(2);

ids.add(3);

*tracker*(ids, AnatateTest.**class**);

}

}

# 10.数据结构补强

## 10.1二叉排序树（BST）

* 左子树不为空，左子树的所哟节点的值小于根节点的值
* 右子树不为空，右子树上所有节点值均大于其根节点的值
* 左右子树也分别为二叉排序树

可用普通的递归实现

## 10.2平衡二叉树（AVL）

* 左子树和右子树的深度差不能超过1，且左子树和右子树都是一颗平衡二叉树。
* 平衡因子：节点的做子树深度减去右子树深度，显然-1<=bf<=1

平衡二叉树实在儿茶排序树上引入的，就是为了解决二叉树的不平衡性导致时间复杂度大大下降，那么AVL久保持住了BST的最好时间复杂度O（logn）