第一章 走入并行

必须知道的几个概念

同步和异步

同步:串行执行，必须得等前面的执行完以后后面的任务才能执行。
异步:调用方法后立即返回，然后调用者可以立即执行后面的任务，而等被调用的方法完成之后会通知调用者。

并发与并行

并发:短时间内还是串行。
并行:真正意义上的同时执行。

临界区

多个线程共享的公共资源或者是共享数据。每一次只能有一个线程占用。

阻塞与非阻塞

阻塞:如果其他线程占用了临界区资源，那么当前线程会在临界区挂起,这种情况称为阻塞。

死锁、饥饿、活锁

死锁

死锁:如果多个线程循环等待,那么就会出现死锁。
死锁产生的三个条件:
1.互斥条件：同一资源同一时刻只能被一个线程占有。
2.请求与保持条件：拿到了资源以后,如果因为请求资源而阻塞,那么就会持有资源不释放。
3.不可剥夺条件:资源在被一个线程拿走之后，其他线程就不能剥夺这个线程的资源。
4.循环等待条件:线程之间会等待其他线程的资源从而形成一个等待循环。
破坏死锁:
1.破坏请求与保持条件:一次性申请所有资源。
2.破坏不可剥夺条件:占用部分资源的线程去申请其他资源，如果申请不到的话那么就释放掉持有的资

源。
3.破坏循环等待条件:按序申请资源,资源反序释放。这样就能保证一个线程从头到尾获得资源。

饥饿

某一线程或者多个线程以为请求不到资源导致一直无法运行(猜测是以时间来判断???),比如因为线程优先级过低，一直有高优先级线程执行。

活锁

活锁则是因为过于“谦让“而导致线程无法拿到全部资源。想象的场景是:两个人不停为对方让路。

并发级别

阻塞

采用的是悲观策略。

无饥饿

尽量采用公平的线程队列，好让所有线程都能够有机会执行。

无障碍

首先是一种乐观的策略,所有线程都可以进入临界区,当多个线程共同修改数据而导致数据破坏时会采用**回滚**策略(侧面也可以看出这种级别认为不会有过多的线程同时修改资源导致并发问题)。
可行的无障碍实现是依赖一个"一致性标记"来实现。线程在操作之前,先读取并保存这个标记,在操作完成以后,再次读取，检查这个标记是否被修改过。

无锁

无锁是无障碍的。
无锁会有多个线程在临界区内修改资源,但是总有一个会成功???

无等待

无等待要求线程在有限步内完成操作,这样不会引起饥饿问题(饥饿就是因为一直不执行)。
典型的无等待结构RCU (Read-Copy-Update)。它的基本思想是,对数据的读不加修改。因此,所有的线程都是无等待的,它们既不会被锁定等待也不会引起任何冲突。

并行有关的两个定律

Amdahal定律

Gustafson定律

回到Java:JMM

JMM:Java内存模型(和JVM关注的点不一样)。由于Java并发的复杂性,所以导致了线程不安全的问题,如何保证下面这些特性也是我们关注的重点:

原子性(Atomicity)

指的是**一个操作还没完成前,其他线程无法干扰**。如果Java虚拟机是32位的话,那么long类型的变量赋值就有可能发生问题(因为long类型是64位,我猜测需要两个线程来完成赋值操作???),所以就会导致值和预期不一致的情况。

可见性(Visibility)

如果在一个线程中修改了某个变量的值,其他共享的线程是否知道。一般来说,其他线程观察当前线程(是否观察到、何时观察到)是无法保证的。

有序性(Ordering)

按照书上的说法来说,干扰有序性的主要因素就是指令重排。指令重排出现的问题就是减少操作系统**流水线**机制的中断从而提升性能。指令重排也得满足**语义串行性**,但是没有义务保证多线程间的语义也一致(书上的例子是线程1给a赋值,线程2对a的值做修改,但是线程2有可能先执行)。

哪些指令不能重排

程序顺序原则

一个线程内保证语义的串行性

volatile规则

volatile变量的写,先发生于读,这保证了volatile变量的可见性

锁规则

解锁(unlock)必然发生在随后的加锁前。

传递性

a=b; b=c;
A必然要先于C

线程的start()先于它的每一个动作

也就是说start方法会先执行。

线程的所有操作先于线程的终结(Thread.join())

线程的中断(interrupt())先于被中断线程的代码

对象的构造函数执行、结束于finalize()方法

finalize()方法在JVM中看到过,可达性分析时会有一次标记,实例在finalize()方法中可以拯救自己一次(和可达链搭上关系)。

第二章 Java并行程序基础

有关线程必知的事

进程与线程

进程资源分配和调度的**基本单位**。
线程是程序的**最小执行单位**。
线程的状态6种，记住并且能说出具体干了什么。

线程的6种状态

初始线程：线程的基本操作

新建

中止

中断

等待(wait)与唤醒(通知 notify)

来自于Object的方法

挂起(suspend)与继续执行(resume)

等待线程结束(join)与谦让(yield)

volatile与JMM(Java内存模型)

分门别类的管理(线程组)

驻守后台的线程(守护线程)

先干重要的事(线程优先级)

线程安全的概念与synchronized

方法加锁

实例信息加锁

静态代码块加锁

程序中的幽灵(隐蔽的错误)

第三章 JDK下的并发包

同步控制

synchronized的功能扩展ReentrantLock(可重入锁)

中断响应

这里需要着重强调的是**lockInterruptibly()**方法,**获取当前锁定,除非当前线程为interrupted**。
如果锁没有被另一个线程占用并且立即返回,则将锁定计数设置为1。
如果当前线程已经保存此锁,则保持计数将递增1,该方法立即返回。
如果该锁被另一个线程保存，则当前线程将被禁用,并且处于休眠状态之一,直到发生两件事情之一:
1.锁是由当前线程获得的->则锁定保持计数器设置为1。
2.其他线程当前线程interrupts

锁申请等待限时

公平锁

ReentrantLock常用API

lock()

获得锁，如果锁已经被占用,则等待

lockInterruptibly()

获得锁,但优先响应中断

tryLock()

尝试获得锁,如果成功,返回true,失败返回false。该方法不等待,立即返回。

tryLock(long time,TimeUnit unit)

在给定时间内尝试获得锁

unlock()

释放锁

重入锁的搭档:Condition条件

常用API

await()

使当前线程等待,同时释放当前锁,当其他线程中使用signal()或者signalAll()方法时,线程会重新获得锁并继续执行。线程中断时跳出等待。与Object.wait()方法类似

awaitUniterruptibly()

与await()基本类似，但是不会响应中断

singal()

用于唤醒一个在等待中的线程。与Object.notify()方法类似

ArrayBlockingQueue的put方法

允许多个线程同时访问:信号量(Semaphore)

常用API

构造方法

public Semaphore(int permits)

public Semaphore(int permits,boolean fair)

构造信号量对象时,必须要指定信号量的**准入数**,即同时能申请多少个许可。如果每个只申请一个许可,相当于有多少个线程可以访问某一个资源。
逻辑方法

public void acquire()

尝试获取一个准入的许可,若无法获得

public void acquireUninterruptibly()

与acquire方法相似，不响应中断。

public boolean tryAcquire()

尝试获得一个许可,如果成功返回true，失败立即返回false。

public boolean tryAcquire(long timeout,TimeUnit unit)

public void release()

线程访问结束后，释放一个许可。

ReadWriteLock读写锁

获取读锁
private static Lock readLock=readWriteLock.readLock();
获取写锁
private static Lock writeLock=readWriteLock.writeLock();

倒计时器CountDownLatch

顾名思义，计时器。书上举的例子的场景是火箭倒计时发射，火箭发射前要保证所有检查线程都结束。

常用API

countDown()

CountDownLatch(int count)

循环栅栏:CyclicBarrier

阻止线程继续执行，要求线程在栅栏处等待。书上的场景是10个士兵一组去完成任务
但是书上的例子对于await()方法没有解释

常用API

await()

等待所有parties已经在这个障碍上调用了await

线程阻塞工具类LockSupport

可以在线程内任意位置让线程阻塞。
和Thread.suspend()相比,它弥补了由于resume()在之前发生导致线程无法继续执行的情况(第二章讨论的指令顺序的问题)
和Object.wait()相比,它不需要先获得某个对象的锁,也不会抛出Interruption的异常。

常用API

park()

阻塞当前线程

parkNanos()

parkUntil()

限时等待

线程复用:线程池

为什么使用线程池?

1. 创建和小会线程都会花费时间

2. 线程本身也会占用内存空间

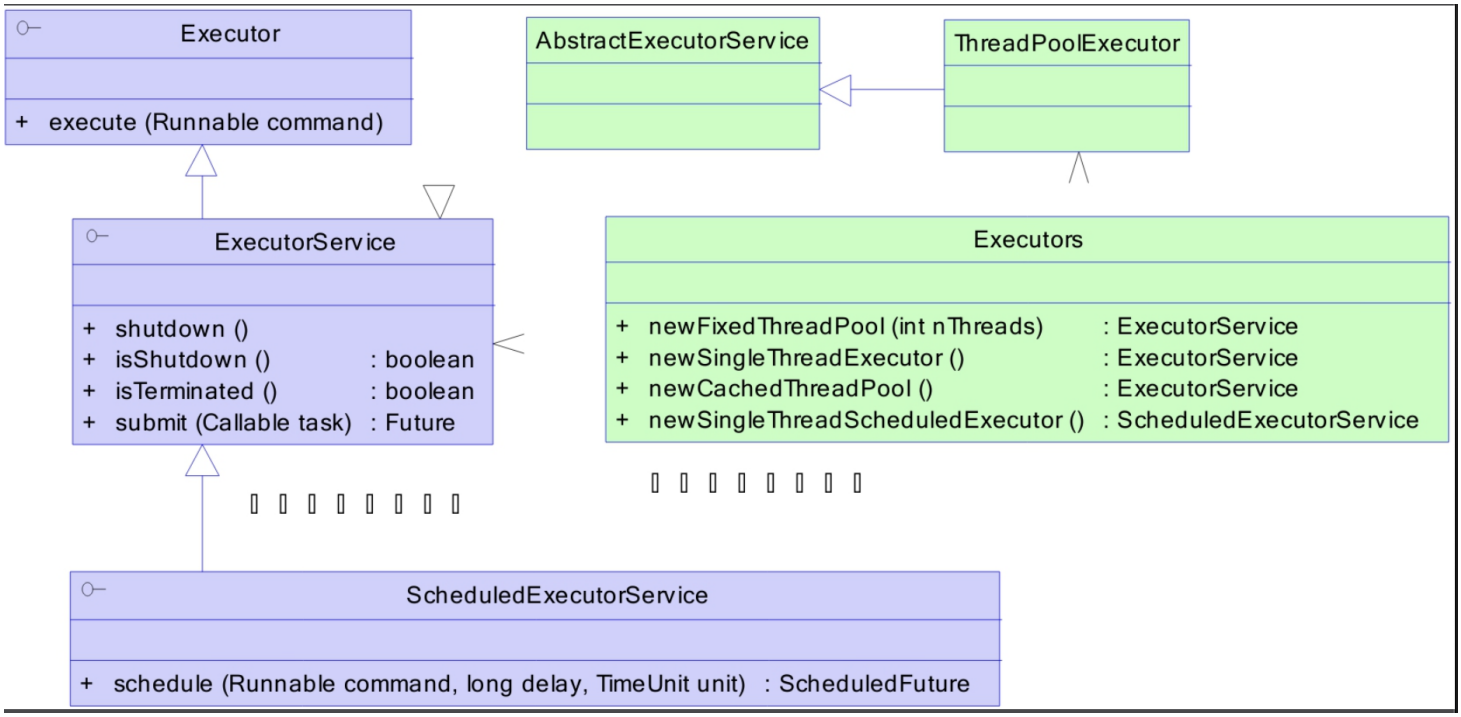
所以,线程的使用必须掌握一个度,在有限的范围内,增加线程的数量可以明显提高系统的吞吐量

什么是线程池

和数据库连接池类似,维护一些数据库连接。

不要重复发明轮子:JDK对线程池的支持

jdk提供了Executor框架，其本质就是一个线程池。



上面的都是juc包中的核心类

ThreadPoolExecutor表示一个**线程池**。

Executors类则扮演着**线程池工厂**的的角色,通过Executors可以取得一个拥有特定功能的线程池。

常用API

public static ExecutorService newFixedThreadPool()

该方法返回一个固定线程数量的线程池, 有新的任务提交时，如果池中有空闲线程则立即执行。

public static ExecutorService newSingleThreadExecutor()

返回一个只有一个线程的线程池。多个任务提交的话，会有一个等待队列。

public static ExecutorService newCachedThreadPool()

返回一个可根据实际情况调整线程数量的线程池。如果空闲线程可以复用，则优先使用。如果没有线程可以复用，则创建新的线程。

public static ScheduledExecutorService newSingleThreadScheduledExecutorS()

返回一个ScheduleExecutorService对象，线程池大小为1，这个对象可以周期性执行某个任务。

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

也返回一个ScheduleExecutorService对象，但是可以指定线程数量。

核心线程池的内部实现

三种模式的线程池内部都实现了ThreadPoolExecutor

```
public static ExecutorService newFixedThreadPool(int nThreads){
    return new ThreadPoolExecutor(nThreads,nThreads,0L,TimeUnit.MILLISECONDS,new LinkedB
}
```

ThreadPoolExecutor最重要的构造函数

```
public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUn
```

来看一下参数
corePoolSize:指定了线程池中的线程数量
maximumPoolSize:指定了线程池中的最大线程数量
keepAliveTime:当线程池数量超过corePoolSize时多余的线程存活的时间
unit:keepAliveTime的单位
workQueue:任务队列,被提交但尚未被执行的任务
threadFactory:线程工厂,用于创建线程,一般用默认的即可
handler:拒绝策略。当任务太多来来不及处理,如何拒绝任务
需要着重关注的是workQueue和handler参数
workQueue是被提交但未执行的**任务队列**,它是一个BlockingQueue接口的对象,仅用于存放Runnable对象。
根据队列的狗功能分类,在ThreadPoolExecutor的构造函数中可以使用以下几种BlockingQueue。

1. 直接提交的队列
   该功能由SynchronousQueue对象提供。任务不会真实的保存,总是将新任务提交给线程执行(后面书上的描述???)
2. 有界的任务队列ArrauBlockingQueue(int capacity)
   由ArrayBlockingQueue实现。如果线程池的实际线程数小于corePoolSize,则会优先创建新的线程。

如果大于corePoolSize,则会将新任务加入等待队列。若等待队列已满,无法加入。

3. 无界的任务队列
   无界的任务队列:无界任务队列可以通过LinkedBlockingQueue类实现,与有界的任务队列相比,除非系统资源耗尽,不存在入队失败的情况。
4. 优先任务队列
   带着优先级的无界队列,通过PriorityBlockingQueue实现。

超过负载之后的拒绝策略

ThreadPoolExecutor的最后一个参数指定了拒绝策略。当线程池中的线程用完了,无法为新任务服务,同时,等待队列中已经排满了,无法容纳新的任务。
JDK内置了四种拒绝策略

1. AbortPolicy策略:该策略会直接抛出异常,阻止系统正常工作。
2. CallerRunsPolicy策略:只要线程池未关闭,改策略直接在调用者线程中，运行当前被丢弃任务。
3. DiscardOledestPolicy策略:该策略将丢弃最老的一个请求,也是就是即将被执行的任务。
4. DiscardPolicy:默默丢弃无法处理的任务。
   以上策略均实现了RejectExectionHandler接口，也可以自己扩展接口。

自定义线程的创建:ThreadFactory

扩展线程池

ThreadPoolExecutor也是一个可以扩展到线程池。它提供了三个接口对线程池进行控制:
beforeExecute()
afterExecute()
terminate()
ThreadPoolExecutor.Worker是ThreadPoolExecutor的**内部类**,它是一个**实现了Runnable接口的类**。
ThreadPoolExecutor线程中的工作线程也是Worker实例。通过beforeExecute()和afterExecute()可以帮助我们输出一些调试的信息。
按照书上的说法对于应用程序的调试和诊断是非常有帮助的。

优化线程池线程数量

线程池中的堆栈

对于我们查看多线程程序的运行情况的了解非常有帮助。
使用execute()或者Future类型,如:
Future re=pools.submit(new DivTask(100,i));

re.get();
更进一步,我们可以找到任务是在哪里提交的,这就需要我们来动手扩展线程池
其实和上面的Runnable扩展一样，本质上是输出堆栈信息。

分而治之:Fork/Join框架

将大任务分割成若干个小任务。

常用API

public ForkJoinTask submit(ForJoinTask task)

可以向ForkJoinPool线程池提交一个ForkJoinTask任务.

JDK的并发容器

JDK提供大量好用的容器类

并发集合简介

1. ConcurrentHashMap:这是一个高效的并发HashMap。**线程安全的HashMap**。
2. CopyOnWriteArrayList:适合**读多写少**的场合。
3. ConcurrentLinkedQueue:高效的并发队列,使用链表实现。可以看做是一个**线程安全的Linkedlist**。
4. BlockingQueue：这是一个接口,JDK内部通过链表和数组实现。表示阻塞队列,适合作为**数据共享的通道**。
5. ConcurrentSkipListMap:跳表的实现。Map,使用跳表的数据结构进行快速查找。

Map的安全集合ConcurrentHashMap

List的线程安全Vector

高效的读写队列:ConcurrentLinkedQueue

不变模式下的CopyOnWriteArrayList

数据共享通道BlockingQueue

随机数据结构:跳表(SkipList)

第四章 关于锁和锁的优化

对于多线程来说，系统除了处理功能需求以外，还需要维护和多线程相关的信息，例如，线程本身的**元数据**(???并不是很懂),线程的调度和线程上下问的切换。
这一章主要围绕如何优化锁来展开，采用的手段有：避免死锁、减小锁的粒度和锁分离等

提升锁性能的建议

减少锁的持有时间

书上举的例子是100个人排队填写表格，如果每个人都拿着笔才开始思考要填什么，那么话费时间肯定会过长。
对应到代码上应该进行如下优化(以jdk并发包中的正则表达式类Pattern为例)：

```
public Matcher matcher(CharSequence input){
    if(!complied){
        synchornized(this){
            if(!complied){ //个人猜测为了一种可能为了线程安全，还有一种可能为了代码的健壮性
                complied();
            }
        }
    }
    Mathcer m=new Matcher(this,input);
    return m;
}
```

减少锁的粒度

这里以ConcurrentHashMap为例(还是1.7的版本,不过不影响学习其锁优化的过程)，其中最为重要的就是put方法和get方法，因为增删的时候最需要保证并发的安全性。
1.7采取的策略是在类内部又有16个HashMap,每个称之为段,这样一个实例的不同段就可被不同的线程持有。
下面这段是put方法(其中有一部分暂时还看不懂)

```
public V put(K key,V value){
   Segment<K,V> s;
   //先处理空指针
   if(value==null){
       throw new NullPointExeception();
   }
   //对key进行hash处理
   int hash=hsah(key);
   //获得对应的段的序号
   int j=(hash>>>segmentShift)&segmentMask;
   if((s=(Segment<K,V>)UNSAFE.getObject(segments,(j<<SSHIFT)+SBASE))==null){
       //上面的if进行了一次非可变性的检验，确保在确认的段里(过于拗口)
       s=ensureSegment(j);
   }
   return s.put(key,hash,value,false);
}
```

但是另一个问题随之而来，就是在获取全局锁的时候消耗资源会更大。
ConcurrentHashMap中的size()方法就是获取全部的有效表项之和,为了达成这一目的，就需要去获取全部的锁,那么在计算时就需要对每个段进行加锁。
但是如果每次计算size的都需要加锁肯定消耗过大,所以首先用采用无锁的策略,只有失败之后才会尝试加锁。
综上:
只有size()这种全局方法调用不频繁时,加锁才能提供并发吞吐量。

读写分离锁来替换独占锁

这个之前有学习过，先掌握ReadWriterLock的API即可。

锁分离

在读写锁的思想上进一步延伸，就是锁分离。
根据应用程序的功能,采用类似锁分离的思想,将独占锁替换为可分离的锁。
书上的例子是BlockingLinkQueue这个数据结构，众所周知，所有的数据结构最基本的数据操作就是创建销毁和增删改查。从这个角度出发，将take()和put()用两个锁来分离。

```
private final ReentrantLock takeLock=new ReentrantLock(); //被take,poll操作线程持有的锁
private final Conditional notEmpty=takeLock.newCondition();
private final ReentrantLock putLock=new ReentrantLock(); //被put,offer操作线程持有的锁。
private final Conditional notFull=putLock.newCondition();
```

take和put之间并不存在锁竞争的关系。
下面来看一下take方法的源码

锁粗化

如果代码中有多个代码块加锁,那么JVM虚拟机会尝试将能够合并的锁进行合并。

Java对锁优化的努力

锁偏向

实例对象已经被当前线程lock了,那么实例对象的头部信息里会有一个标志来进行锁偏向的记录,就可以减少加锁的资源开销。

轻量级锁

如果线程获取偏向锁失败,虚拟机并不会立即挂起线程,它还会使用一种被称为轻量级锁的优化手段。这个操作将对象的头部作为指针指向持有锁的线程堆栈的内部,来判断一个对象是否持有对象锁。如果轻量级锁加锁失败,则表示其他线程争夺到了锁,那么当前线程的锁请求就膨胀为重量级锁。

自旋锁

重量级锁前的最后尝试,虚拟机会让当前线程几个空循环,如果还不能获取锁,才会真实地在操作系统层面挂起。

锁的消除

Java虚拟机在JIT编译时,通过对运行上下文的扫描,去除不可能存在共享资源竞争的锁。
书上的例子很简单

```
public String[] createStrings(){
    Vector<String> v=new Vector<String>();
    for(int i=0;i<100;i++){
        v.add(Integer.toString(i));
    }
    return v.toArray(new String[]{});
}
```

函数有自己方法栈,也就是说v是局部变量,属于线程私有的数据,因此**不可能被其他线程访问**,内部所有加锁同步都是没有必要的。如果虚拟机检测到这种情况,就会将这些无用的锁操作去除。
锁消除涉及的一项关键技术为**逃逸分析**,所谓逃逸分析就是观察某个变量是否会逃出某个作用域。

也就是如果上面的上面的方法返回的是v,那么认为局部变量v逃逸。所以虚拟机中就不能消除v中的操作。

ThreadLocal

简单使用

ThreadLocal的实现原理

对性能有何帮助

无锁

比较交换CAS

原子类(AtomicInteger)

Java中的指针Unsafe类

无锁的对象引用:AtomicReference

带有时间戳的对象引用

无锁数组:AtomicIntegerArray

让普通变量也享受原子操作:AtomicIntegerFieldUpdater

无锁的Vector实现

细看SynchronousQueue的实现

有关死锁

并行模式与算法

单例模式

不变模式(final关键字)

生产者-消费者模式

高性能的无锁的实现

Future模式

异步调用机制,虽然它无法立即返回需要的数据,但是会返回一个契约,将来凭借这个契约去重新获取你需要的信息。

Future模式的主要角色

1. Main 系统启动,调用Client发出请求
2. Client 返回Data对象,立即返回FutureData,并开启ClientThread线程装配RealData
3. Data 返回数据的接口
4. FutureData Future数据,构造很快,但是一个虚拟的数据,需要装配RealData
5. 真实数据,其构造是比较慢的

并行流水线

并行搜索

并行排序

并行算法:矩阵算法

NIO(准备好了再通知我)

书上的例子是从编写一个NIO服务器为例,但是在此之前我们需要对于unix网络编程有一定的了解。
本地进程通信的方式,其中一种就是socket(套接字,也就是将IP与端口号集合起来的抽象概念)。
首先实现了用tcp的Socket编写服务器(这部分可以参考基础部分),然后进阶到NIO编程。
NewIO,通过知乎的文章https://zhuanlan.zhihu.com/p/25004921,
我们可以对于unix编程提供的接口有一定的了解,再去看书上的例子可以更加清晰。

操作系统提供Soket编程的接口

socket()函数

int socket(int domain,int type,int protocol)
socket函数对应于普通文件打开操作,返回一个文件描述字,socket()用于创建一个socket描述符
domain:协议族。AF_INET、AF_INET6,协议族决定了socket的地址类型,例如ANF_INET决定了要用ipv4(32位)与端口号的组合。
type:指定socket类型。
protocol:顾名思义，就是指定协议。

bind()函数

int bind(int sockfd,const struct sockaddr *addr ,socklen_t addrlen)
把一个地址族中的特定地址赋给socket

listen()、connect()函数

listen()服务端在调用socket()、bind()之后调用listen()来监听这个socket
如果客户端调用connect()发出连接请求,服务端就会接收到这个请求

accept()函数

TCP服务器监听到这个请求之后,就会调用accept()函数接收请求,这样连接就建立好了。

read()、write()

网络I/O操作

close()函数

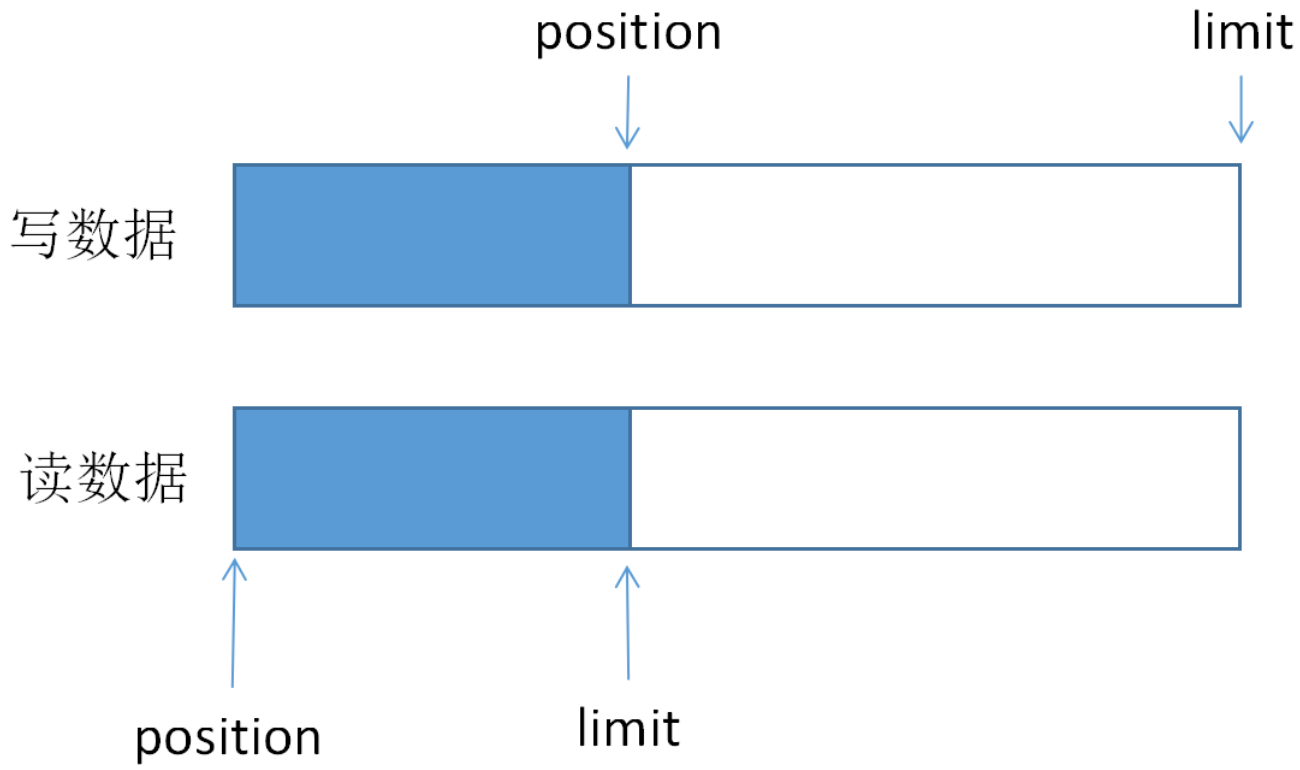
完成了读写操作之后就要关闭相应的socket描述字。

JDK中的核心类

两台计算机连接:

1. 服务器实例化一个**ServerSocket对象***,表示通过服务器上的端口通信。
2. 服务器调用ServerSocket类的**accept()**方法,该方法将一直等待,直到客户端连接到服务器上给定的端口。
3. 服务器正在等待时,一个客户端实例化一个Socket对象,指定服务器名称和端口号来请求连接。
4. Socket类的构造函数将客户端连接到指定的服务器和端口号,如果通信被建立,则在客户端创建一个Socket对象能够与服务器进行通信(和3在同一步骤)。
5. 在服务器,accept()方法返回服务器上的一个新的socket引用,该socket连接到客户端的socket。
   此外还有一个数据结构**ByteBuffer**,本质上就是一个数组。几个重要的参数:
6. 容量(Capacity)缓冲区能够容纳的数据元素的最大数量。容量在缓冲区创建时被设定,并且永远不能改变。
7. 上界(Limit)缓冲区里的数据的总数,代表了当前缓冲区一共有多少数据(这个参数下文会反复提到,filp()函数中会对这个值做修改)。
8. 位置(Position)下一个要被读或写的元素的位置。Position会自动由相应的get()和put()函数更新。
9. 标记(Mark)一个备忘位置。用于记录上一次读写的位置。
   如果在ByteBuffer中放入了数据,然后想从中读取的话,就需要position这个字段放到想读的位置(那为什么不放这个字段呢?)

```
public final Buffer position(int newPosition){
    if(newPosition > limit || (newPosition<0)){
        throw new IllegalArgumentExcetion;
    }
    position=newPosition;
    if(mark > position){
        mark=-1;
    }
    return this;
}
```



其中limit代表可写或者可读的总数。一个新创建的bytebuffer,它可写的总数就是它的capacity。如果写入了一些数据以后,想从头开始读的话,这时候的**limit就是当前ByteBuffer数据的长度**。
为了从写数据变成读数据的情况,还需要修改limit,这就要用到limit方法

```
public final Buffer limit(int newLimit){
    if(newLimit > capacity || (newLimit<0)){
        throw new IllegalArgumentExcetion;
    }
    limit=newLimit;
    if(postion> limit) position=limit;
    if(mark>limit) mark=-1;
    return this;
}
```

然后就是执行方法
byteBuffer.limit(byteBuffer.position());
byteBuffer.position(0);
由于这个方法操作非常频繁,jdk为此封装了一个方法flip()

注意:

1. 客户端的输出流连接到服务器端的输入流，而客户端的输入流连接到服务器端的输出流
2. TCP双向通信,同一时刻可由客户端发送到服务器端,也可由服务器端发送到客户端。

ServerSocket

服务器端用这个类创建Socket,并监听客户端请求

Socket

客户端用的Socket

InetAddress

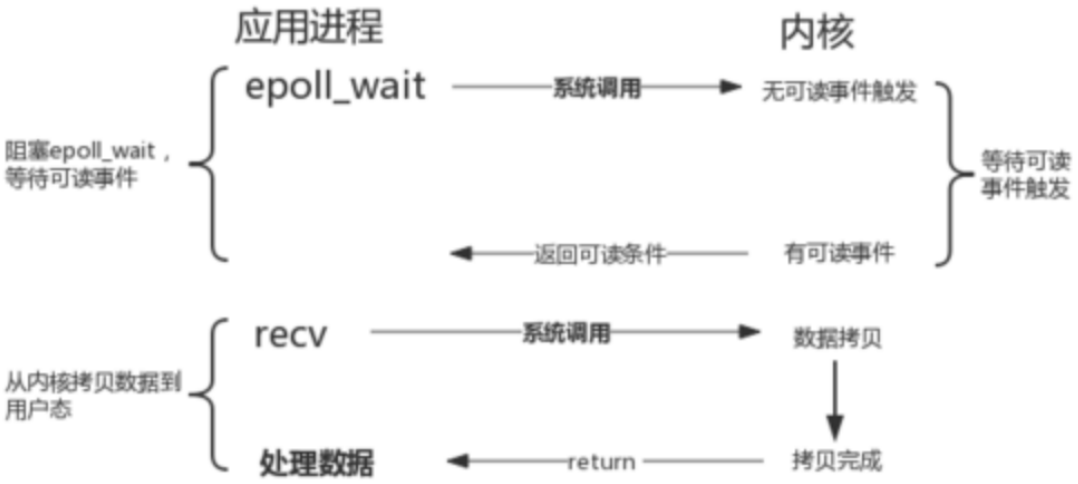
表示IP地址的类

再看NIO服务器

这次我们从IO模型出发,其实Java代码最终也是通过JNI调用系统接口。
知乎上这篇专栏讲得很好,用C语言手写了一个服务端,再通过Java客户端去进行访问。

为什么会有Selector???

看书的时候会有疑问
第一个疑问是书上只说了用Selector来管理Channnel,但是却没有说明为什么使用。
第二个疑问是注册是设置感兴趣的事件。
为了回答上面的问题,还是得从IO模型出发,5种模型有一种模型叫做IO多路复用模型。
先来回答第一个问题,Selector其实就是select/poll/epoll的包装类。
上面的回答狗看了直摇头,这是什么玩意儿...
先回答第二个问题



最常用的I/O**事件通知机制**就是IO复用(I/O multiplexing),**I/O复用接口本身是阻塞的**,在应用程序中通过I/O复用接口**向内核注册fd(描述符)所关注的事件(回头看Java中的API也就能理解了),当事件触发时,通过I/O复用接口的返回值通知到应用程序。I/O复用接口可以同时监听多个I/O事件**提升处理效率。

AIO(读完了再通知我)

在开始学习之前,我们需要了解一下unix编程的5种IO模型
https://zhuanlan.zhihu.com/p/27382996
NIO和AIO分别对应的是非阻塞模型和异步IO模型(其他的可以在unix网络编程中进行了解)
来看一下start()方法

```
//客户端请求的处理和接收
    public void start() {
        System.out.println("server listen on"+PORT);

        //注册事件和事件完成后的处理器
        server.accept(null, new CompletionHandler<AsynchronousSocketChannel, Object>() {
            final ByteBuffer buffer=ByteBuffer.allocate(1024);

            @Override
            public void completed(AsynchronousSocketChannel result, Object attachment) {
                System.out.println(Thread.currentThread().getName());

                Future<Integer> writeResult=null;
                try{
                    buffer.clear();
                    result.read(buffer).get(100, TimeUnit.SECONDS);
                    buffer.flip();
                    writeResult=result.write(buffer);
                } catch (ExecutionException e) {
                    throw new RuntimeException(e);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                } catch (TimeoutException e) {
                    throw new RuntimeException(e);
                }finally {
                    try{
                        server.accept(null,this);
                        writeResult.get();
                        result.close();
                    } catch (ExecutionException e) {
                        throw new RuntimeException(e);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                }
            }

            @Override
            public void failed(Throwable exc, Object attachment) {
                System.out.println("failed:"+exc);
            }
        });
    }
```

在上面的start()方法中最重要的就是accept方法，第一个参数不多做描述
主要是第二个参数其实就是一个Handler(在Java基础中接触过,调用处理器,当某个特定事件发生时会自动被调用),这个接口中有两个方法

void completed(V result,A attachment)
void failed(Throwable exc,A attchment)
这两个方法分别在**异步操作accept()成功时或者失败时被回调**
因此AsynchronousServerSocketChannel.accept()主要做了两件事:

1. 发起accept请求,**告诉系统可以开始监听端口**。
2. 注册CompletionHandler实例,告诉系统一旦有客户端连接,如果成功连接,就去执行CompletionHandler.completed()方法;如果连接失败,就去执行CompletionHandler.failed()方法