Day01

**大数据概述**

大数据高并发基础 -高并发 分布式

nio,concurrent,序列号反序列化,RPC,Zookeeper

大数据离线分析 -积攒了大量的数据后 一次性处理 得到结果

Hadoop -HDFS MapReduce Yarn

Hive Hbase Flume Sqoop

大数据实时分析 -海量数据 实时处理 数据不停的过来 实时处理后 立即展示变化

Kafka (-消息队列工具)

Storm

大数据内存计算 -主要指以Spark为核心的 基于内存的海量数据处理技术

Spark Core

Spark Sql

Spark Streaming

SparkMllib

Spark Graphx

(Spark AI未来指向)

爬虫&数据可视化

python语言

Python爬虫开发

可视化 -Echars Davav HeatMap...

机器学习 & 算法

基于R语言

常用算法 -数学



**一.NIO概述**

BIO - JDK1.0 - 同步阻塞式IO 面向流 操作字节或字符 单向传输数据

在执行ACCEPT CONNECT READ WRITE 时都会产生阻塞,在平常开发当中并不是问题 甚至因为这样的模型直观而简单 用的很多,但是在高并发的场景下 这样的阻塞式IO可能会造成问题,在服务器开发中 需要在服务器端通过少量线程处理多个客户端请求 这就要求在少量的线程应该可以灵活的切换处理不同客户端 但传统的BIO阻塞式的工作方式 一旦阻塞了线程 线程就被挂起 无法继续执行 无法实现这样的功能

InputStream OutputStream Reader Writer

NIO - JDK4.0 - 同步非阻塞式IO 面向通道 操作缓冲区 双向传输数据

和传统的BIO比起来最主要的特点是 在执行ACCEPT CONNECT READ WRITE 操作时是非阻塞的,非常便于实现 在服务器开发中 用少量的线程来处理多个客户端请求 由于以上四种操作都是非阻塞的 可以随时让线程切换所处理的客户端 从而可以实现高并发服务器的开发

Buffer Channel Selector

阻塞、非阻塞：

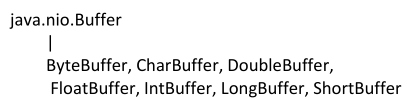
从线程的角度考虑 ，线程挂起 不在抢夺CPU 则称为线程被阻塞

同步、异步：

从并发参与者角度考虑，多个参与者是否需要互相等待协调，如果任务的执行需要双方互相等待、互相协调，则为同步，否则为异步.

**二.Buffer详解**

1.缓冲区,本质上就是一段连续的内存空间,用来临时存放大量指定类型的数据



2.Buffer中的重要概念,position,limit,capacity

3.创建Buffer:

创建出来的Buffer默认 capacity等于容量 position为0 limit等于capacity

没有构造方法 不能直接new

static ByteBuffer allocate(int capacity)

static ByteBuffer wrap(byte[] array)

4.写入数据到buffer --存到缓冲区里都是字节数据

abstract ByteBuffer put(byte b...)

5.从buffer中获取数据--可以向缓冲区中读入不同类型的数据

abstract byte... get()

6.反转缓冲区--本质上就是将 limit设置为当前position的值,再将position设置为0的过程

Buffer flip()

7.判断边界

int remaining() 返回 limit - position 获取读写时是距离边界的距离

boolean hasRemaining() 返回 limit-position>0的值,判断度写时是否到达了边界

8.重绕缓冲区-将position置为0 ,可以重新进行读写操作

Buffer rewind()

9.设置/重置标记 --设置一个临时的指针mark指向当前position的值 之后 在任何时候 可以调用reset()方法 使position重新指向mark指定的值 从而 恢复到上一次mark位进行读写操作

Buffer mark() 在此缓冲区的位置设置标记

Buffer reset() 将此缓冲区的位置重置为以前标记的位置

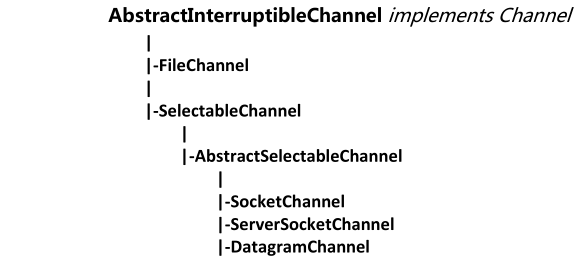
10.清空缓冲区--它并不会真的取清除缓冲区中的数据,而只是修改响应buffer的参数.抛弃mark 将limit设置为capacity 将position设置为0,效果上等价于清空数据.

Buffer clear()

**三.Channel详解**

1.通道概述:

NIO中的基本概念,类似于BIO中的流,不同的是,操作的是缓冲区,且可以双向传输数据



2.ServerSocketChannel:代表tcp通信中的服务器端

构造方法被保护起来 无法直接使用

static ServerSocketChannel open()--得到ServerSocketChannel对象

abstract ServerSocket socket()--获取ServerSocketChannel底层对应的真正套接字对象。

abstract ServerSocket bind(InetSocketAddress isa)-将通道绑定到某一地址端口---jdk1.7开始 有这个方法

SelectableChannel configureBlocking(boolean block)-调整此通道的阻塞模式。默认true 阻塞

abstract SocketChannel accept()-接受到此通道套接字的连接

**在非阻塞模式下,ACCEPT操作没有阻塞,无论是否收到一个连接,都直接执行下去,此时即使ACCEPT方法执行成功,也无法确认连接完成.此时应该自己通过代码来控制实现连接,或者,通过选择器来实线选择操作.**

SocketChannel sc = null;

while(sc == null){

sc = ssc.accept(); }

close()-关闭此通道

3.SocketChannel

代表tcp通信中的客户端

构造方法被保护起来 无法直接使用

open()-打开套接字通道。

configureBlocking(boolean block)-调整此通道的阻塞模式。

**abstract boolean connect(SocketAddress remote)-连接此通道的套接字。**

abstract boolean finishConnect()-完成套接字通道的连接过程

**在非阻塞模式下,CONNECT操作没有阻塞,无论是否完成一个连接,都直接执行下去,此时即使CONNECT方法执行成功,也无法确认连接完成.此时应该自己通过代码来控制实现连接,或者,通过选择器来实现选择操作.**

**boolean isConn = sc.connect(new InetSocketAddress("127.0.0.1", 44444));**

**if(!isConn){**

**while(!sc.finishConnect()){ } }**

**read(ByteBuffer dst)-将字节序列从此通道中读入给定的缓冲区。**

**\*\*在非阻塞模式下,read方法执行过后,并不能保证真的读到了数据,或读全了数据,此时只能自己写代码来控制读取的过程.**

**ByteBuffer buf = ByteBuffer.allocate(5);**

**while(buf.hasRemaining()){**

**sc.read(buf); }**

**write(ByteBuffer src)-将字节序列从给定的缓冲区中写入此通道。**

close()-关闭此通道。

**四.Selector详解**

1.Selector概述

Selector选择器是NIO中实现少量线程服务多个客户端连接的最关键的组件.一方面允许多个客户端连接注册到选择器中关注对应的事件.另一方面提供了"选择"操作 来在之前注册的操作中选择已经就绪的操作 交给线程来执行.一头连接了多个客户端连接 另一头连接了少量的线程,进行协调 从而实现了少量线程 来处理多个客户端连接的功能.

从功能上来说:

功能1:注册功能 允许客户端连接 注册到选择器上 关注对应事件

功能2:选择功能 可以在线程中 选择已经就绪的注册事件 来进行处理

2.继承结构 java.nio.channels.Selector

3.构造方法被保护起来,无法直接new.

4.open()-打开一个选择器

5.register(Selector sel, int ops)-向给定的选择器注册此通道，返回一个选择键。

static int OP\_ACCEPT--用于套接字接受操作的操作集位。

static int OP\_CONNECT--用于套接字连接操作的操作集位。

static int OP\_READ--用于读取操作的操作集位。

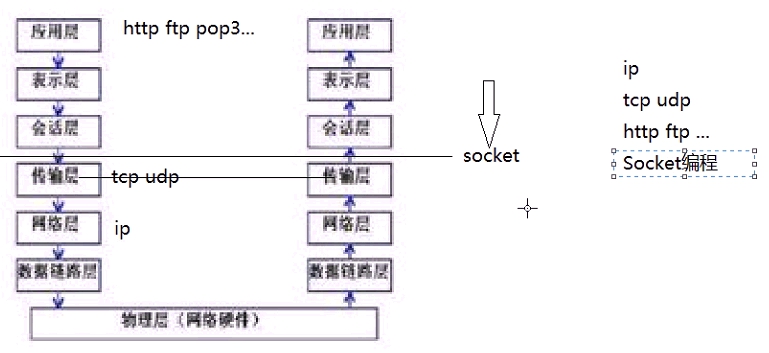
static int OP\_WRITE--用于写入操作的操作集位。

6.abstract in select()-选择一组键，其相应的通道已为 I/O 操作准备就绪

7.abstract Set<SelectionKey> selectedKeys()-返回此选择器的已选择键集

**五.粘包问题**

socket编程其实是基于传输层的协议在开发网络程序的过程.传输层本身就没有针对会话进行管理的功能.所以在进行socket编程时会遇到和会话相关的各种问题.最典型的就是粘包问题.所谓的粘包问题 就是当 通过socket发送多段数据时 底层的tcp协议 会自动根据需要 将数据 拆分或合并 组成数据包后发送给接受者 ,接受者收到数据后 无法直接通过tcp协议本身判断数据的边界,这个问题就称之为粘包问题.粘包问题本质上是因为tcp协议是传输层的协议 本身没有对会话控制提供相应的能力 我们基于socket开发网络程序时相当于在自己实现 会话层 表示层 和应用层的功能 所以 需要自己来相办法解决粘包问题.



**1.粘包问题的解决方案**

a.只发送固定长度的数据

通信的双发约定每次发送数据的长度,每次只发送固定长度的数据,接收数据方 每次都按照固定长度获取数据

缺点:不够灵活,只适合每次传输的数据都有固定长度的场景

b.约定分隔符

通信双方约定一个特殊的分隔符用来表示数据的边界,接收方收到数据时,不停读取,以分隔符为标志,区分数据的边界

缺点:如果数据本身就包含分隔符字符,则需要对数据进行预处理将数据本身包含的分隔符进行转义,相对来说比较麻烦

c.使用协议

所谓的协议 其实就是通信双方遵循的通信方式的约定,,协议分为 公有协议 和 私有协议

所谓的公有协议 通常是由 国际标准化组织定义 全世界的计算机都遵循的协议 例如 HTTP协议 FTP协议,POP3协议 SMTP协议…

所谓的私有协议 通常是由 公司 组织 个人自定义的协议 只在小范围内使用 解决小范围内通信时特定的需求

1.开源的NIO结构的服务器框架

MINA Netty

2.IO方式的总结

阻塞/非阻塞:讨论的是线程的角度,当执行某些操作不能立即完成时,线程是否被挂起,失去cpu争夺权 无法继续执行 直到阻塞结束 或被 唤醒

同步/异步:讨论的是参与通信双方的工作机制,是否需要互相等待对方的执行.

同步指的是 通信过程中 一方在处理通信 另一方 要等待对方执行 不能去做其他无关的事.

异步指的是 通信过程中 一方在处理通信 另一方 可以不用等待对方 而可以去做其他无关的事

,直到对方处理通信完成 再在适合的时候继续处理通信过程

BIO jdk1.0 同步阻塞式IO 面向流 操作字节或字符 单向传输数据

NIO jdk4.0 同步非阻塞式IO 面向通道 操作缓冲区 双向传输数据

AIO jdk7.0 异步非阻塞式IO 大量使用回调函数 异步处理通信过程

掌握NIO相关概念和原理

Day02

**一.Courrent概述**

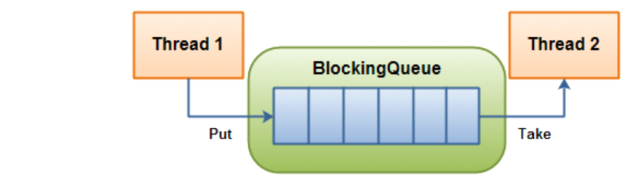
1.Courrent概述-Concurrent包是jdk5中开始提供的一套并发编程包,其中包含了大量和多线程开发相关的工具类,大大的简化了java的多线程开发,在高并发 分布式场景下应用广泛.

2.Courrent包

**1.BlockingQueue-阻塞式队列概述**

a.本身是一种队列数据结构,和其他队列比起来,多了阻塞机制,从而可以在多个线程之间进行存取队列的操作,而不会有线程并发安全问题.所以称之为阻塞式队列.

可以简单的理解为,阻塞式队列是专门设计用来在多个线程间通过队列共享数据.

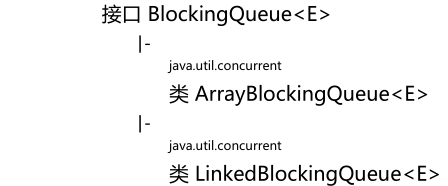


在阻塞式队列中,如果队列满了,仍然有线程向其中写入数据,则这次写入操作会被阻塞住,直到有另外的线程从队列中消费了数据,队列有了空间,阻塞才会被放开,写入操作才可以执行.

同样,如果队列是空的,仍然有线程从队列中获取数据,则读取操作将会被阻塞住,直到有另外的线程向队列中写入了数据,队列不再为空,阻塞才会被放开,读取操作才可以执行

可以发现这个过程中,通过阻塞机制,协调了多个线程在一个队列上的读写操作,所以称之为阻塞式队列.阻塞式队列可以非常方便的实现 多线程开发中 经典的 生产者-消费者模式

b..继承模式



ArrayBlockingQUeue 底层是数组 需要在创建之初就指定好大小

LinkedBlockingQUeue 底层是链表 可以在创建时指定大小 也可以不指定

c..重要方法:



**2.ConcurrentMap-并发Map**

a.是一个线程安全的Map,可以防止多线程并发安全问题.

**ConcurrentMap 性能非常好的原因**

a锁更加精细:HashTable加锁是锁在整个HashTable上,一个线程操作时其他线程无法操作,性能比较低.ConcurrentMap加锁是 将锁加载数据分段(桶)上 只锁正在操作的部分数据 效率高.

b引入了读写锁机制:在多线程并发操作的过程中,多个并发的读其实不需要隔离,但只要有任意一个写操作,就必须隔离.HashTable没有考虑一点,无论什么类型的操作,直接在整个HashTable上加锁.ConcurrentMap则区分了读写操作,读的时候加读锁,写的时候加写锁,读锁和读锁可以共存,写锁和任意锁都不能共存,从而实现了在多个读的过程中不会隔离 提高了效率..

b.继承结构



**3.CountDownLatch-闭锁--Cook**

是Concurrent包提供的一种新的并发构造,可以协调线程的执行过程,实现协调某个线程阻塞直到其他若干线程执行达到一定条件才放开阻塞继续执行的效果.

CountDownLatch(int count) 构造一个用给定计数初始化的 CountDownLatch

await() 使当前线程在锁存器倒计数至零之前一直等待，除非线程被中断。

countDown() 递减锁存器的计数，如果计数到达零，则释放所有等待的线程。

**4.CyclicBarrier-栅栏--Horse**

Concurrent包中提供的一种并发构造,可以实现多个并发的线程在执行过程中,在某一个节点进行阻塞等待,直到所有的线程都到达了指定位置后,一起放开阻塞继续运行的效果.

构造方法 CyclicBarrier(int parties) 创建一个新的 CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，但它不会在启动 barrier 时执行预定义的操作。

await() 在所有参与者都已经在此 barrier 上调用 await 方法之前，将一直等待。

**5.Exchanger-交换机--交换暗号**

可以实现两个线程交换对象的效果 先到达的线程会产生阻塞 等待后续到来的线程,直到两个线程都到达交换机后 互换对象 各自继续执行

构造方法 Exchanger() 创建一个新的 Exchanger。

exchange(V x)-等待另一个线程到达此交换点（除非当前线程被中断），然后将给定的对象传送给该线程，并接收该线程的对象。

**6.Semaphore-信号量--保护人民大会堂**

Concurrent包中提供的一个并发构造,可以在创建信号量时指定信号量的初始数量,后续可以调用acquire()来获取信号量 通过release()释放信号量,如果某一个时刻,信号量被取完,再调用acquire()方法时,该方法将会产生阻塞,直到有其他线程release()信号量回来.

信号量的主要用途:a..保护一个重要(代码)部分防止一次超过 N 个线程进入。 b.在两个线程之间发送信号。

构造方法摘要

Semaphore(int permits) 创建具有给定的许可数和非公平的公平设置的 Semaphore。

Semaphore(int permits, boolean fair) 创建具有给定的许可数和给定的公平设置的 Semaphore。

acquire()-从此信号量获取一个许可，在提供一个许可前一直将线程阻塞，否则线程被中断。

release()-释放一个许可，将其返回给信号量

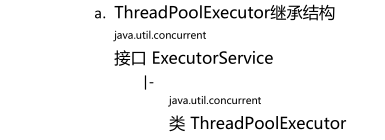
**7.ExecutorService-执行器服务**

ExecutorService是Concurrent包下提供的一个接口,主要用来实现线程池

所谓的池就是用来重用对象的一个集合,可以减少对象的创建和销毁,提高效率.

而线程本身就是一个重量级的对象,线程的创建和销毁都是非常耗费资源和时间,所以如果需要频繁使用大量线程,不建议每次都创建线程销毁线程,而是应该利用线程池的机制,实现线程对象的共享,提升程序的效率

ExecutorService用法 - 通过ThreadPoolExecutor实现类创建线程池



b.ThreadPoolExecutor的重要API

通过构造方法创建线程池ThreadPoolExecutor。

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

corePoolSize 核心池大小

maximumPoolSize 最大池大小

keepAliveTime 空闲的多余线程保持时间

unit 时间单位

workQueue 阻塞式队列

handler 拒绝服务助手

**\*线程池的工作方式**

1):在线程池刚创建出来时,线程池中没有任何线程,当有任务提交过来时,如果线程池中管理的线程的数量小于corePoolSize,则无论是否有闲置的线程都会创建新的线程来使用.而当线程池中管理的线程的数量达到了corePoolSize,再有新任务过来时,会复用闲置的线程.

2)当所有的核心池大小中的线程都在忙碌,则再有任务提交,会存入workQueue中,进行排队,当核心池大小中的线程闲置后,会自动从workQueue获取任务执行

3).而当所有的核心池大小中的线程都在忙碌,workQueue也满了,则会再去创建新的临时线程来处理提交的任务,但是,无论如何,总的线程数量,不允许超过maximumPoolSize

4)而当所有的核心池大小中的线程都在忙碌,workQueue也满了,也创建了达到了

maximumPoolSize的临时线程,再有任务提交,此时会交予RJHandler来拒绝该任务

1. 当任务高峰过去,workQueue中的任务也都执行完成,线程也依次闲置了下来,则在此时,会将闲置时间超过keepAliveTime(单位为unit)时长的线程关闭掉,但是关闭时会至少保证线程池中管理的线程的数量 不少于corePoolSize个

**ExecutorService用法 - 通过Executors工具类的静态方法创建线程池**

a.创建一个擅长处理 大量短任务的线程池

static ExecutorService newCachedThreadPool()

创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们

b.可以实现使用单一线程处理任务,多个任务在无界的阻塞式队列中排队等待处理

static ExecutorService newSingleThreadExecutor()

创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程

c.可以实现使用指定数量的线程处理任务,多个任务在无界的阻塞式队列中排队等待处理

static ExecutorService newFixedThreadPool(int nThreads)

创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。

**向线程池中提交任务**

a.execute(Runnable) 最普通的提交任务的方法,直接传入一个Runnable接口的实现类对象,即可要求线程池取执行这个任务,这种方式提交的任务无法监控线程的执行 也无法在线程内向调用者返回返回值.

void execute(Runnable command) 在未来某个时间执行给定的命令。

b.submit(Runnable)

此方法也可以通过传入一个Runnable接口实现类对象的方式来向线程池提交任务,不同之处在于此方法带有一个Future类型的返回值,可以通过Future对象的get()方法来检测线程是否执行结束,如果线程未执行结束get()方法将会阻塞.

Future<?> submit(Runnable task) 提交一个 Runnable 任务用于执行，并返回一个表示该任务的 Future。

c.submit(Callable)

和上面submit(Runnable)方法非常类似,只不过这个方法传入的是Callable接口的实现类,Callable接口功能和Runnable接口基本一致,唯一的不同在于,内部的方法叫call,且可以返回返回值,这个返回值可以通过Future对象通过get()方法得到

<T> Future<T>submit(Callable<T> task)

提交一个返回值的任务用于执行，返回一个表示任务的未决结果的Future

d.invokeAny(...)可以接收若干Callable组成的集合,此方法将会自动从中选择任意一个执行

<T> T invokeAny(Collection<? extends Callable<T>> tasks) 执行给定的任务，如果某个任务已成功完成（也就是未抛出异常），则返回其结果。

e.invokeAll(...) 可以接收若干Callable组成的集合,此方法将会执行所有的Callable将结果组成集合返回

<T>List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)

执行给定的任务，当所有任务完成时，返回保持任务状态和结果的Future 列表。

**关闭线程池**

线程池中维护了大量线程,很耗费资源,所以当使用线程池结束时,应该手动关闭线程池,释放资源,关闭线程池的方法,即使调用也不会立即关闭所有线程,而是不再接收新的任务,之前已经提交但尚未完成执行的线程仍然会继续执行,直到所有的任务都执行完,线程池关闭所有线程,退出.

shutdown() shutdownNow()

**Day03**

**8.锁**

java.util.concurrent.locks.Lock 是一个类似于 synchronized 块的线程同步机制.但是Lock比 synchronized 块更加灵活、精细.

继承结构



构造方法



void lock() 获取锁。

void unlock() 试图释放此锁

lock和syncronized对别

a.lock可以配置公平策略,实现线程按照先后顺序获取锁

b.提供了trylock方法 可以试图获取锁 获取到 或 获取不到时 返回不同的返回值 让程序可以灵活处理

c.lock()和unlock()可以在不同的方法中执行,可以实现同一个线程在上一个方法中lock()在后续的其他方法中unlock(),比syncronized灵活的多

读写锁--表现是更加灵活

对于多线程并发安全问题,其实只在设计到并发写的时候才会发生,多个并发的读并不会有线程安全问题,所以在Concurrent包中提供了读写锁的机制,可以实现,分读锁和写锁来进行并发控制.多个读锁可以共存,而写锁和任意锁都不可共存,从而实现多个并发读并行执行提升效率,而任意时刻写都进行隔离,保证安全.这是一种非常高效而精细的锁机制.





ReentrantReadWriteLock.ReadLock readLock() 返回用于读取操作的锁。

ReentrantReadWriteLock.WriteLock writeLock() 返回用于写入操作的锁。

**9.原子性操作**

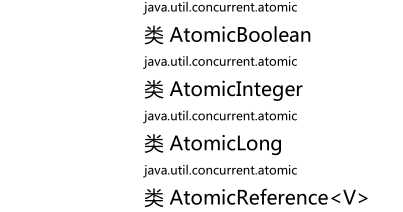
在java中最基本的运算都是非原子性的,在底层存在多个步骤,所以在多线程并发操作的过程中有可能有多线程并发安全问题

**解决方案:**

**1.加synchronized** 使用同步代码块进行同步,可以解决问题,但是如果在多线程并发场景下,最基本的运算都要同步的话,代码会被大量的同步代码块包裹,代码混乱,效率低下

**2.使用Concurrent包中提供的Atomic原子型操作**

**包装类型,**



以AtomicInteger为例:



int addAndGet(int delta) 以原子方式将给定值与当前值相加。

int getAndAdd(int delta) 以原子方式将给定值与当前值相加

int getAndIncrement() 以原子方式将当前值加 1。

int getAndDecrement() 以原子方式将当前值减 1。

void set(int newValue) 设置为给定值。

**序列化/反序列化**

1.概述

序列化/反序列化:java是面向对象的语言,万物皆对象,对象平常是存活在内存中的,动态变化的数据.但是有的时候,我们需要将内存中存活的动态变化的对象数据转化成可以固定下来的字节信息,以便于保存或传输.而序列化技术就可以实现这个效果,而将已经固定下来的对象的字节信息再转换回内存对象的过程就称之为反序列化的过程.

序列化是持久化的前提，持久化是序列化的可能目的之一。

2.序列化与反序列化的应用场景:

a.将对象序列化后[持久化]保存

将内存中的对象信息序列化后保存到外部存储设备中.这个过程也称之为将对象持久化保存了.之后再需要的时候可以将持久化,保存的对象再读取回程序内进行反序列化再恢复为对象,这个过程也称之为反持久化.

b.将对象序列化化后通过网络传输实现[RPC]

将内存中的对象信息序列化后,将字节信息通过网络发送给其他程序,其他程序收到这些数据后,进行反序列化就可以恢复出对象,得到对象中的信息.

3.常见的序列化反序列化技术

序列化 反序列化 是在分布式开发中 包括 [持久化] [RPC] 操作的基础,是在分布式环境下非常常见的操作 所以序列化反序列时性能的好坏 直接影响分布式程序的性能 所以非常的重要 在分布式程序中 选择一个良好的 序列化反序列化技术 往往对性能有非常直接影响

1. Java自带的

ObjectInputStream ObjectOutputStream Serializable

优点:java原生的序列化反序列化机制,不需要导入任何第三方包就可以使用

缺点:

效率低下 - 浪费时间

产生的结果数据大 - 浪费空间

只能再java中使用 - 无法跨语言

b.其他开源序列化反序列化技术-AVRO

apache开源组织提供的,性能优良,体积较小,开源跨语言,在大数据技术中常见

c.其他开源序列化反序列化技术-GoogleProtobuffer

由Google提供的开源序列化反序列化技术,性能优良 体积较小 可以跨语言,在业界广泛使用

d.其他开源序列化反序列化技术-Thrift--由facebook提供的开源序列化反序列化技术,性能优良 体积较小 可以跨语言,在特定领域用的较多

**Java自带的序列化反序列化机制**

a.想要被java序列化机制操作的对象的类必须实现Serializable接口

b.序列化和反序列化 操作的对象并不是同一个 而是相同类型的不同对象具有相同的信息

c.在java序列化反序列化过程中,实现了Serializable接口的类,需要指定 静态常量serialVersionUID属性,如果不指定虚拟机在编译过程中会自动生成一个,来指定当前类的序列化版本编号,在序列化时,在产生的字节信息中会保存这个编号,而在反序列化时也会去检查这个编号,如果不一致则认为序列化和反序列化时不是同一个类 抛出异常 保证了 序列化反序列化时类型的一致安全

d.在通过网络将序列化的对象字节信息发送给另一台机器进行反序列化时,虽然两端是同样的类但是并不是同一个类,默认生成的SeriaVersionUid很可能不同,会造成反序列化失败,此时需要在两头都手动指定SeriaVersionUid为相同的值,才可以保证正常反序列化

e.可以在实现了Serializable接口的类的属性中,用transient关键字声明指定属性不需要被序列化,这样在序列化该类的对象时,该属性会自动被忽略,这在保护对象中一些隐秘信息时非常的有用

**GoolgeProtoBuffer序列化反序列化**

谷歌提供的第三方序列化框架,以效率高,体积小,支持多种语言而著称,在许多技术汇总都可以见到它的身影.

**使用GPB的步骤:**

a.编写proto文件,利用GPB自定义的语法来声明要进行序列化的类的结构

b.通过GPB提供的编译器将proto文件编译为需要的语言的类文件

c.将生成的类导入项目中,利用GPB提供的API实现序列化反序列化

**标识号:**

消息中的每个属性都应该有一个唯一的标识号,用来在二进制消息中唯一的代表该属性,且一经使用不可再改.[1-15]占1个字节,[16,2047]占两个字节,所以经常用的字段优先使用[1-15]

**字段规则:**

声明字段在类中存在的规则

required：一个格式良好的消息一定要含有1个这种字段。表示该值是必须要设置的

optional：消息格式中该字段可以有0个或1个值（不超过1个）。

repeated：在一个格式良好的消息中，这种字段可以重复任意多次（包括0次）。重复的值的顺序会被保留。表示该值可以重复，相当于java中的List。

**RPC概述:**

RPC即java的远程方法调用,是java在分布式环境下,分布式程序的各个部分之间进行互相调用的机制.

传统的方法调用都是在一台JVM虚拟机中运行,直接在内存中完成方法的调用,但是在分布式的环境下,方法的调用可能需要跨JVM虚拟机 跨进程 跨网络来进行,此时无法直接在内存中完成,需要通过序列化反序列化机制 网络通信机制 动态代理机制 等等的机制来实这种远程的方法的调用,这个过程称之为java的远程方法调用,即RPC.

RPC并不是一项独立的技术,而是基于若干种技术,针对分布式环境下远程的方法调用提出的一种解决方案.

GoogleProtoBuffer中的RPC :略

**RPC技术的重要性**

rpc是通过序列化技术 网络通信技术 来实线的远程方法调用的技术方案,可以非常方便的

实现分布式环境下方法调用,是分布式程序开发的基础,是后续学习的大数据框架底层实现

技术,虽然以后很少回去写RPC的代码,需要大家有所了解.