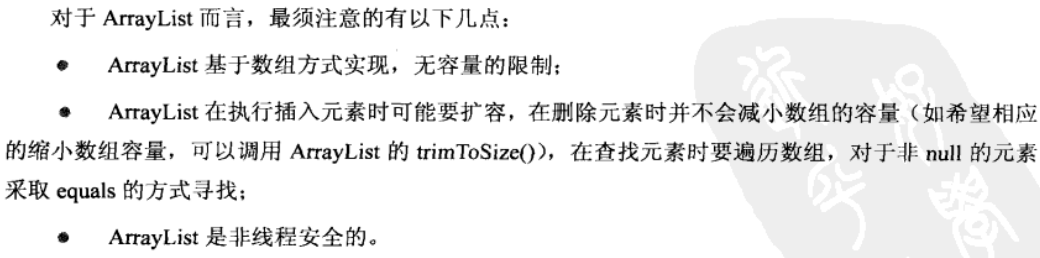
# 1 基础知识

对于构建分布式应用而言，使用好JDK重的集合、并发、网络（BIO、NIO及AIO）以及序列化/反序列化非常重要。

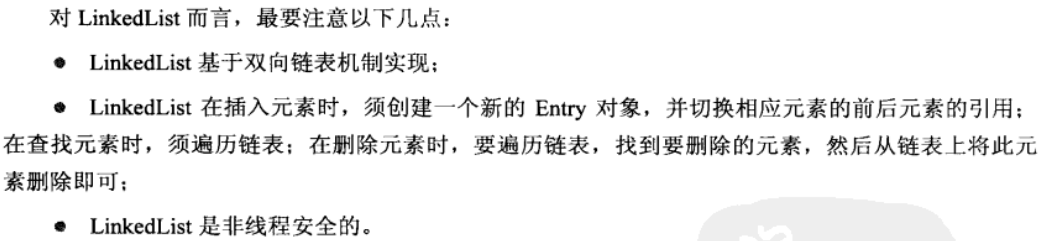
## 1.1 集合包(package)

常用的有Collection和Map两个接口的实现类型，Collection用于存放多个单对象，Map用于存放Key-Value形式的键值对。

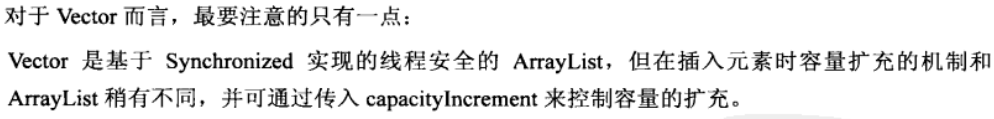
ArrayList：



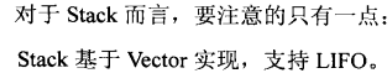
LinkedList：



Vector



Stack



# 2 并发编程

## 2.1 基础概念

要编写线程安全的代码，核心在于对状态访问操作进行管理（特别是共享的和可变的状态访问）。共享意味着可以由多个线程同时访问，可变则意味着变量的值在其生命周期内可发生改变。

一个对象是否需要线程安全，取决于它是否被多个线程同时访问。指的是程序中访问对象的方式，要使得对象线程安全，则需采用同步机制来协同对象的可变状态的访问。

Java的同步机制包括关键字synchronized、volatile类型变量、显式锁以及原子变量。

*当设计线程安全的类时，良好的面向对象技术、不可修改性以及清晰的不变性规范都能起到一定的帮助作用。*

*无状态的对象一定是线程安全的（如工具类）。，*

**什么是线程安全性**

当多个线程访问某一个类时，不管运行时采用了何种调度方式或者多个线程如何交替执行，并且在主调代码不需要任何额外的同步或协同，这个类都能表现出正确的行为，那么这个类就是线程安全的。

### 2.1.1原子性

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。原子操作是指，对于访问同一个状态的所有操作（包括该操作本身）来说，这个操作是一个以原子方式执行的操作。

在Java中，对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。

x = 10; //语句1

y = x; //语句2

x++; //语句3

x = x + 1; //语句4

只有语句1是原子性操作，其他三个语句都不是原子性操作。读取和写入共享变量都需要使用同步。

Java提供了一种内置的锁机制来支持原子性：同步代码块（Synchronized Block）。包括两部分：一是作为锁的对象引用，另外一个作为锁保护的代码块。以关键字synchronized来修饰的方法即是横跨整个方法的代码块，其中该同步代码块的锁就是方法调用所在的对象。每个Java对象都可以用作一个实现同步的锁，这些锁被称为内置锁。线程进入同步代码块之前会自动获得锁。

synchronized (lock) {

//访问或修改由锁保护的共享状态

}

同步代码块和同步方法可以确保多个操作以原子的方式执行操作。

共享和发布对象，从而使它们能够安全地由多个线程同时访问。

### 2.1.2 可见性

单线程环境下如果某个变量写入值，没有其它写入操作的情况下读取这个变量，总能得到相同的值。然而多线程环境下，当读和写操作在不同线程中执行时，情况并非如此。通常，我们无法确保读操作的线程能够适时地看到其他线程写入的值。为了确保多个线程之间对内存写入操作的可见性，必须使用同步机制。

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。

初始化final字段确保可见性

对于可见性，Java提供了volatile关键字来保证可见性。另外，通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。

**非原子的64位操作**

非volatile的64位数值变量（long和double），Java内存模型要求，变量的读取操作和写入操作都必须为原子操作，但对于非volatile类型的long和double变量，JVM允许将64位的读或写操作分解为两个32位的操作。故在多线程环境下使用共享且可变的long和double变量也是线程不安全的，除非使用关键字volatile来声明它们，或者加锁保护。

**加锁**

加锁的含义不仅仅局限于互斥行为，还包括内存的可见性。为了确保所有线程都能看到共享变量的最新值，所有执行读操作或者写操作的线程都必须在同一个锁上同步。

### 2.1.3 有序性

### 2.1.4 volatile关键字

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

* 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
* 禁止进行指令重排序。

volatile变量，一种稍弱的同步机制，用来确保将变量的更新操作通知其他线程。它不会被缓存在寄存器或者其他处理器不可见的地方，因此读取volatile变量时总会返回最新的值。

在访问volatile变量时不会执行锁操作，不会使线程阻塞，它是一种比关键字synchronized更轻量级的同步机制。不建议过度依赖volatile变量提供可见性，通常比锁的代码更脆弱。

*volatile变量正确使用方式包括：确保它们自身状态的可见性、确保它们所引用对象的状态可见性、标识一些重要的程序生命周期事件的发生。*

加锁机制既可以确保可见性又可以确保原子性，而volatile只能确保可见性。

**使用volatile关键字的场景**

synchronized关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized，但是要注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性。通常来说，使用volatile必须具备以下2个条件：

1）对变量的写操作不依赖于当前值

2）该变量没有包含在具有其他变量的不变式中

　　实际上，这些条件表明，可以被写入 volatile 变量的这些有效值独立于任何程序的状态，包括变量的当前状态。

### 2.1.5 线程封闭

一种避免使用同步的方式就是不共享数据，仅仅单线程内访问数据就不需要同步。这种技术称为线程封闭（Thread Confinement）。它是实现线程安全的最简单方式之一。

**AD-Hoc线程封闭**

指维护线程封闭性的职责完全由程序实现来承担。AD-Hoc线程封闭是非常脆弱的，在程序中尽量少用它。

**栈封闭**

栈封闭是线程封闭的一种特例，只能通过局部变量才能访问对象。局部变量固有属性之一就是封闭在执行线程之中，其他线程无法访问这个栈。

**ThreadLocal**

维持线程封闭性的一种更规范的方法使用ThreadLocal，ThreadLocal提供了get()与set()方法，这些方法为每个使用它的线程维护一份单独的拷贝。所以get()总是返回当前执行线程通过set()设置的最新值。

### 2.1.6 ThreadLocal类

**ThreadLocal是什么**

ThreadLocal，顾名思义，它不是一个线程，而是线程的一个本地化对象。当工作于多线程中的对象使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程分配一个独立的变量副本。所以每一个线程都可以独立地改变自己的副本，而不会影响其他线程所对应的副本。从线程的角度看，这个变量就像是线程的本地变量，这也是类名中“Local”所要表达的意思。

ThreadLocal为了防止对可变的单例变量或全局变量进行共享。ThreadLocal实例通常是私有静态变量，并通常和线程的状态有关。

ThreadLocal和多线程并发没有什么关系。ThreadLocal模式是为了解决单线程内的跨类跨方法调用的。

ThreadLocal是线程范围内的资源共享，并不能在多线程共享资源的情况下使用，因为它会为每个线程创建独立的资源，且不可与其他线程共享

### 2.1.7 不变性

不可变的对象一定是线程安全的。不可变的对象很简单，它们只有一种状态，并且状态由构造函数来控制。

在Java语言规范和Java内存模型中都没有给出不可变性的正式定义，不可变性不等于将对象中的所有域都声明为final类型，即使对象中所有域都是final类型的，这个对象仍然是可变的，因为final类型的域中可以保存对可变对象的引用。

满足以下条件时，对象才是不可变的：

* 对象创建以后其状态就不能修改
* 对象所有的域都是final类型的
* 对象是正确创建的（在对象创建期间，this引用没有溢出）

## 2.2 基础构建模块

### 2.1.1 同步容器类

同步容器类包括：Vector、HashTable。实现线程安全的方式：将它们的状态封装起来，并对每个公有的方法都进行同步，使得每一次只有一个线程能访问容器的状态。

同步容器将所有容器状态的访问都串行化，以实现它们的线程安全。这种方法严重降低并发性，当多个线程竞争容器的锁时，吞吐量将严重降低。

### 2.1.2 并发容器类

Java 5.0提供了多种并发容器类来改进同步容器的性能。并发容器是针对多个线程并发访问设计的。通过并发容器来代替同步容器，可极大提高伸缩性并降低风险。

ConcurrentHashMap代替HashTable、HashMap(非同步)，CopyOnWriteArrayList代替List。Java 5.0新增两个容器类：Queue和BlockingQueue。Queue用来临时保存一组等待处理的元素。

**ConcurrentHashMap**

ConcurrentHashMap是线程安全的HashMap实现，它并不是将每个方法都在一个锁上同步并使得每一次只能有一个线程访问容器。而是使用一种粒度更细的加锁机制来实现更大程度的共享，这种机制成为分段锁。可多线程进行并发读和写操作。在并发场景中选用ConcurrentHashMap。

额外的原子Map操作

ConcurrentHashMap不能被加锁来执行独占空间，因此无法使用客户端加锁来创建新的原子操作。

**CopyOnWriteArrayList**

CopyOnWriteArrayList是线程安全、并且在读操作时是无锁的ArrayList。通过使用ReentrantLock来保证线程安全，并不是在需同步的方法中加入synchronized关键字。在并发场景中，CopyOnWriteArrayList性能比ArrayList好。

**CopyOnWriteArraySet**

CopyOnWriteArraySet基于CopyOnWriteArrayList的实现，唯一不同是在add时调用的是CopyOnWriteArrayList的addIfAbsent方法。如数组中已存在当前对象，则直接返回；没有则放入数组的尾部，并返回。

CopyOnWriteArraySet在add时每次都要进行数组的遍历，其性能会略低于CopyOnWriteArrayList。

**ArrayBlockingQueue**

ArrayBlockingQueue是一个基于数组、先进先出、线程安全的集合类，可实现指定时间的阻塞读写，并且容量是可限制的。

**AtomicInteger**

AtomicInteger是一个支持原子操作的Integer类，

## 2.3 异步任务执行框架（Executor）

任务是一组逻辑工作单元，而线程则是使任务异步执行的机制。

*执行任务的策略(过时的)*

* *所有的任务放在单个线程串行执行，糟糕的响应性和吞吐量*
* *将每个任务放在各自的线程中执行，资源有限*

### 2.3.1 Executor

Executor接口为异步任务执行框架提供了基础，该框架支持多种不同类型的任务执行策略。Excecutor提供标准的方法将任务的提交过程与执行过程解耦开来，并用Runnable表示任务。Exceutor的实现还提供了对生命周期的支持，以及统计信息收集、应用程序管理机制和性能监视等。

Executor基于生产者 – 消费者模式，提交任务的操作相当于生产者，执行任务相当于消费者。

**执行策略**

通过将任务的提交和执行解耦开来，就可以为任务指定和修改这行策略。定义任务的“What、Where、How、When”等方面，包括：

* 在什么（What）线程中执行任务？
* 任务按照什么（What）顺序执行（FIFO、LIFO、优先级）？
* 有多少个任务（How Many）任务并发执行？
* 如何通知（How）应用程序有任务被拒绝？
* 如果系统过载需要拒绝任务，应该选择哪个（Which）任务？
* 在执行一个任务之前或之后，应该进行哪些(What)动作？

ExecutorService扩展了Executor接口，添加了用于生命周期管理的方法。ExecutorService的生命周期有种状态：运行、关闭和已终止。

Executor执行的任务有4个生命周期：创建、提交、开始和完成。Future表示一个任务的生命周期，提供了相应的方法判断是否已经完成或取消，获取任务的结果和取消任务等。Future的规范中隐含的意义是，任务的生命周期只能前进，不能后退。

Future中的get()行为取决于任务的状态（尚未开始、正在运行、已完成）。如果已完成那么get()会立即返回或抛出一个Exception，如果任务未完成，那么get将阻塞并直到任务完成。如果抛出了异常，get()会将异常封装成ExecutionException抛出，如果任务取消，get()返回CancellationException

## 2.4 取消和关闭

要使任务和线程能安全、快速、可靠地停止下来，并不是件容易的事。Java没有提供任何机制来安全终止线程。但提供了中断（Interruption）的协作机制，能使一个线程终止另外一个线程的当前工作。

### 2.4.1 取消任务

在Java中没有哪一种用来停止线程的方法是绝对安全的，因此没有哪一种方法优先用来停止任务。只有一些协作的机制，通过协作，使任务和代码遵循一个统一的协议，用来请求取消。通常任务需拥有取消策略，这个策略详细定义了How、When以及What。以下是协作机制：

* 设置某个“已取消请求”标志，任务（另外一个线程）定期查看该标志，如果设置了该标志，任务将提前结束。
* 线程中断也是一种协作机制，线程可以通过中断来通知另外一个线程，告诉它在合适的或可能的情况停止当前的工作，并转入其他的工作。每个线程都有一个boolean类型的中断状态，当中断时状态被置为true。
* 通过Future来实现取消，Future拥有cancel()，返回boolean表示取消操作是否成功。

*调用interrupt()并不意味着立即停止目标线程正在进行的工作，而只是传递了请求中断的消息。它不会真正地中断一个正在运行的线程，而是发出了中断请求，然后由线程自己在下一个合适的时刻中断自己。*

**中断策略**

最合理的中断策略是某中形式的线程级取消操作或服务级取消操作：尽快退出，必要时进行清理。

interrupted()：测试当前线程是否已经中断。线程的中断状态 由该方法清除。

**响应中断**

两种策略可用于处理InterruptedException：

* 传递异常，从而使你的方法也成为可中断的阻塞方法。

传递InterruptedException可添加到throws子句中抛出

* 恢复中断状态，从而调用栈的上层代码能够对其进行处理。

如果无法传递InterruptedException，那么需要保存中断请求。标准的方法就是再次调用interrupt()来恢复中断状态。不能屏蔽InterruptedException，例如在catch块中捕获到异常却不处理，除非在你的代码中实现了线程的中断策略。

**不可中断的阻塞**

有些阻塞的方法或者阻塞机制是不会响应中断的，此时中断请求只能设置线程的中断状态，然而没有任何作用。对于这些由于执行不可中断操作而被阻塞的线程，可以使用类似中断的手段来停止这些线程，但这要求必须知道线程阻塞的原因。以下是不可中断的阻塞：

* Java.io中的同步Socket I/O，read()、write() 等方法都不会响应中断，可以抛出SocketException停止。
* Selector的异步I/O 如果一个线程在调用Selector.select()方法是阻塞了，那么调用close()或wakeup()线程抛出CloseSelectorException提前返回
* 获取某个锁，如果一个线程由于等待某个内置锁而阻塞，那么将无法响应中断，因为线程认为它肯定会获得锁，以不会理会中断请求。但是Lock提供了lockInterruptibly()，该方法运行在等待一个锁的同时仍能响应中断。

### 2.4.2 停止基于线程的服务

应用程序通常会创建拥有多个线程的服务，例如线程池。当应用程序退出时，那么服务所拥有的线程也需要结束，没有优先的方法来退出线程，因此它们自行结束。除非拥有某个线程，否则不能对该线程进行操控。在示例中，线程池是工作线程的所有者，如果要中断线程，应该由线程池来负责。

线程的所有权是不可传递的。当应用程序关闭服务时，服务就可以关闭所有线程了。

### 2.9.3 发布

加锁方式：

* 第一种方法：可以将所有的可变状态都封装在对象内部，并通过对象的内置锁对所有访问可变状态的代码进行同步，使得该对象上不会发生并发访问。例如Vector类。

*该种方法编译器或运行时都不会强制实施这种模式，如果添加新的方法或代码时忘记使用同步，那么加锁协议就很容易被破坏。*

并发性能考量

## 测试并发实现

# 2 分布式示例

## 2.1 分布式服务器架构（J2EE）

这是一个全国性的通信平台,对性能,海量数据,容错性以及扩展性有非常高的要求,所以在系统的架构上就不能简单的采用集中式，需满足以下条件：

1.数据分布式存储

2.请求分布式调度

3.多结点分布式部署

4.双重备份,热切换

系统的核心无非就是网络架构,分布式算子和通信,要求如下:

分布式算子:

1.对于任意输入,输出均匀分布

2.输出结果数可控

通信:

1.高并发量

2.多线程

分布式算子我们选择的是sun公司的hash函数,通信用的则是cindy socket通信.网络架构以及具体的描述会在后面的blog中逐步给出.

### 2.1.1 网络架构

整个系统的架构如下图所示,包括四层,每一层可以由若干结点来对数据和请求分流:

* **接口服务器(Interface Server):**

1) 对外提供访问接口并接受请求,考虑到HTTP的广泛性,一般内置一个http服务器进程

2) 监控各dispatcher server的工作状态

3) 转发请求到其中的一个最优dispatcher中,这里的最优性判断以各dispatcher server的工作状态为依据,当然在这一层上不心请求的具体内容可以简单地采用轮询或随机算法.

* **消息分发服务器(Dispatcher Server):**

1) 接受来自于接口服务器的请求

2) 解析请求,提取特征参数(一般是类似于用户帐号之类的东西,一个帐号下的数据会被分布到同一个结点上),然后对该参数执行hash函数,计算出目标数据所在的App Server,然后将请求转发给该App Server.

3) 事实上,在实际的项目中的处理比上面的介绍要更复杂一些,但伸缩性大大加强了.

* **应用服务器(App Server):**

1) 执行业务逻辑,等同于集中式系统中的应用服务器,已经不存在分布式的特征了.所处理的数据就是自己数据库中的数据,与网络上的其他结点无关.

2) 被划分为多个逻辑组(group),同一个组中的服务器负载均衡

3) 考虑到数据库的双重备份,热切换和负载均衡,才用了多数据库单读多写策略。对于读,监控各数据库工作状态,选择一个最优数据库来提供数据;对于写,同时写所有的数据库,因此必须保证操作的事务性。

* **数据库服务器(DB Server):**

1).提供数据访问,没什么好说的,对于非事务性数据库需要在App Server层提供辅助措施;

* **结点之间的通信**

1).数据(请求,响应,异常)以网络格式异步并发传输

