带你一次搞明白 Java 多线程（XI）

7 保障线程安全的设计技术

从面向对象设计的角度出发介绍几种保障线程安全的设计技术,这些技术可以使得我们在不必借助锁的情况下保障线程安全,避免锁可能导致的问题及开销.

7.1 Java 运行时存储空间

Java 运行时(Java runtime)空间可以分为栈区,堆区与方法区(非堆空间).

栈空间(Stack Space)为线程的执行准备一段固定大小的存储空间,每个线程都有独立的线程栈空间,创建线程时就为线程分配栈空间.在线程栈中每调用一个方法就给方法分配一个栈帧,栈帧用于存储方法的局部变量,返回值等私有数据，即局部变量存储在栈空间中，基本类型变量也是存储在栈空间中，引用类型变量值也是存储在栈空间中,引用 的对象存储在堆中. 由于线程栈是相互独立的,一个线程不能访问另外一个线程的栈空间,因此线程对局部变量以及只能通过当前线程的局部变量才能访问的对象进行的操作具有固定的线程安全性.

堆空间(Heap Space)用于存储对象,是在 JVM 启动时分配的一段可

以动态扩容的内存空间. 创建对象时,在堆空间中给对象分配存储空间,实例变量就是存储在堆空间中的, 堆空间是多个线程之间可以共享的空间,因此实例变量可以被多个线程共享. 多个线程同时操作实例变量可能存在线程安全问题

非堆空间(Non-Heap Space)用于存储常量,类的元数据等, 非堆空间也是在 JVM 启动时分配的一段可以动态扩容的存储空间.类的元数据包括静态变量,类有哪些方法及这些方法的元数据(方法名,参数,返回值等). 非堆空间也是多个 线程可以共享的, 因此访问非堆空间中的静态变量也可能存在线程安全问题

堆空间也非堆空间是线程可以共享的空间,即实例变量与静态变量是线程可以共享的,可能存在线程安全问题. 栈空间是线程私有的存储空间,局部变量存储在栈空间中,局部变量具有固定的线程安全性

7.2 无状态对象

对象就是数据及对数据操作的封装, 对象所包含的数据称为对象的状态(State), 实例变量与静态变量称为状态变量.

如果一个类的同一个实例被多个线程共享并不会使这些线程存储共享的状态,那么该类的实例就称为无状态对象(Stateless Object). 反之如果一个类的实例被多个线程共享会使这些线程存在共享状态,那

么 该类的实例称为有状态对象. 实际上无状态对象就是不包含任何实例变量也不包含任何静态变量的对象.

线程安全问题的前提是多个线程存在共享的数据,实现线程安全的一种办法就是避免在多个线程之间共享数据,使用无状态对象就是这种方法

7.3 不可变对象

不可变对象是指一经创建它的状态就保持不变的对象,不可变对象具有固有的线程安全性. 当不可变对象现实实体的状态发生变化时,系统会创建一个新的不可变对象,就如 String 字符串对象. 一个不可变对象需要满足以下条件:

1) 类本身使用 final 修饰,防止通过创建子类来改变它的定义

2) 所有的字段都是 final 修饰的,final 字段在创建对象时必须显示初始化,不能被修改

3) 如果字段引用了其他状态可变的对象(集合,数组),则这些字段必须是 private 私有的

不可变对象主要的应用场景:

1) 被建模对象的状态变化不频繁

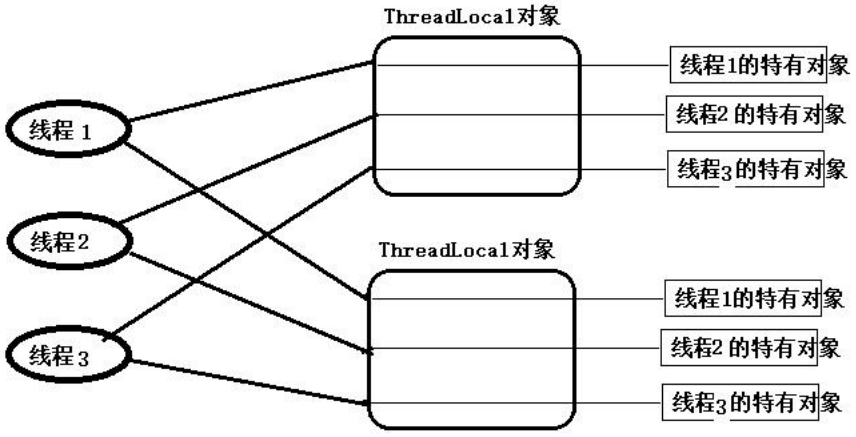
2) 同时对一组相关数据进行写操作,可以应用不可变对象,既可以保障原子性也可以避免锁的使用

3) 使用不可变对象作为安全可靠的 Map 键, HashMap 键值对的存储位置与键的 hashCode()有关,如果键的内部状态发生了变化会导致键的哈希码不同,可能会影响键值对的存储位置. 如果 HashMap 的键是一个不可变对象,则 hashCode()方法的返回值恒定,存储位置是固定的.

7.4 线程特有对象

我们可以选择不共享非线程安全的对象,对于非线程安全的对象,每个线程都创建一个该对象的实例,各个线程线程访问各自创建的实例,一个线程不能访问另外一个线程创建的实例. 这种各个线程创建各自的实例,一个实例只能被一个线程访问的对象就称为线程特有对象. 线程特有对象既保障了对非线程安全对象的访问的线程安全,又避免了锁的开销.线程特有对象也具有固有的线程安全性.

ThreadLocal<T>类相当于线程访问其特有对象的代理,即各个线程通过 ThreadLocal 对象可以创建并访问各自的线程特有对象,泛型 T 指定了线程特有对象的类型. 一个线程可以使用不同的 ThreadLocal 实例来创建并访问不同的线程特有对象



ThreadLocal 实例为每个访问它的线程都关联了一个该线程特有的对象, ThreadLocal 实例都有当前线程与特有实例之间的一个关联.

7.5 装饰器模式

装饰器模式可以用来实现线程安全,基本思想是为非线程安全的对象创建一个相应的线程安全的外包装对象,客户端代码不直接访问非线程安全的对象而是访问它的外包装对象. 外包装对象与非线程安全的对象具有相同的接口,即外包装对象的使用方式与非线程安全对象的使用方式相同,而外包装对象内部通常会借助锁,以线程安全的方式调用相应的非线程安全对象的方法.

在 java.util.Collections 工具类中提供了一组 synchronizedXXX(xxx) 可以把不是线程安全的 xxx 集合转换为线程安全的集合,它就是采用了这种装饰器模式. 这个方法返回值就是指定集合的外包装对象.这类集合又称为同步集合.

使用装饰器模式的一个好处就是实现关注点分离,在这种设计中,实现同一组功能的对象的两个版本:非线程安全的对象与线程安全的对象. 对于非线程安全的在设计时只关注要实现的功能,对于线程安全的版本只关注线程安全性

8 锁的优化及注意事项

8.1 有助于提高锁性能的几点建议

1 减少锁持有时间

对于使用锁进行并发控制的应用程序来说,如果单个线程特有锁的时间过长,会导致锁的竞争更加激烈,会影响系统的性能.在程序中需要尽可能减少线程对锁的持有时间,如下面代码:

```
public synchronized void    syncMethod(){
    othercode1();
    mutexMethod();
    othercode();
}
```

在 syncMethod 同步方法中,假设只有 mutexMethod()方法是需要同步的, othercode1()方法与 othercode2()方法不需要进行同步. 如果

othercode1 与 othercode2 这两个方法需要花费较长的 CPU 时间,在并发量较大的情况下,这种同步方案会导致等待线程的大量增加. 一个较好的优化方案是,只在必要时进行同步,可以减少锁的持有时间,提高系统的吞吐量,如把上面的代码改为:

```
public    void    syncMethod(){
    othercode1();
    synchronized (this) {
        mutexMethod();
    }
    othercode();
}
```

只对 mutexMethod()方法进行同步,这种减少锁持有时间有助于降低锁冲突的可能性,提升系统的并发能力.

2 减小锁的粒度

一个锁保护的共享数据的数量大小称为锁的粒度. 如果一个锁保护的共享数据的数量大就称该锁的粒度粗,否则称该锁的粒度细.锁的粒度过粗会导致线程在申请锁时需要进行不必要的等待.减少锁粒度是一种削弱多线程锁竞争的一种手段,可以提高系统的并发性

在JDK7前,java.util.concurrent.ConcurrentHashMap类采用分段锁协

议,可以提高程序的并发性

3 使用读写分离锁代替独占锁

使用 ReadWriteLock 读写分离锁可以提高系统性能，使用读写分离锁也是减小锁粒度的一种特殊情况．第二条建议是能分割数据结构实现减小锁的粒度,那么读写锁是对系统功能点的分割.

在多数情况下都允许多个线程同时读,在写的使用采用独占锁,在读多写少的情况下,使用读写锁可以大大提高系统的并发能力.

4 锁分离

将读写锁的思想进一步延伸就是锁分离.读写锁是根据读写操作功能上的不同进行了锁分离.根据应用程序功能的特点,也可以对独占锁进行分离.如 java.util.concurrent.LinkedBlockingQueue 类中 take()与 put()方法分别从队头取数据,把数据添加到队尾．虽然这两个方法都是对队列进行修改操作,由于操作的主体是链表,take()操作的是链表的头部,put()操作的是链表的尾部,两者并不冲突．如果采用独占锁的话,这两个操作不能同时并发,在该类中就采用锁分离,take()取数据时有取锁, put()添加数据时有自己的添加锁,这样 take()与 put()相互独立实现了并发.

5 粗锁化

为了保证多线程间的有效并发,会要求每个线程持有锁的时间尽量短.但是凡事都有一个度,如果对同一个锁不断的进行请求,同步和释放,也会消耗系统资源.如:

```
public void    method1(){
    synchronized( lock ){
        同步代码块 1
    }
    synchronized( lock ){
        同步代码块 2
    }
}
```

JVM 在遇到一连串不断对同一个锁进行请求和释放操作时,会把所有的锁整合成对锁的一次请求,从而减少对锁的请求次数,这个操作叫锁的粗化,如上一段代码会整合为:

```
public void    method1(){
    synchronized( lock ){
        同步代码块 1
        同步代码块 2
    }
}
```

在开发过程中,也应该有意识的在合理的场合进行锁的粗化,尤其在循环体内请求锁时,如:

```
for(int i = 0 ; i< 100; i++){
    synchronized(lock){}
}
```

这种情况下,意味着每次循环都需要申请锁和释放锁,所以一种更合理的做法就是在循环外请求一次锁,如:

```
synchronized( lock ){
    for(int i = 0 ; i< 100; i++){}
}
```

8.2 JVM 对锁的优化

1 锁偏向

锁偏向是一种针对加锁操作的优化,如果一个线程获得了锁,那么锁就进入偏向模式, 当这个线程再次请求锁时,无须再做任何同步操作,这样可以节省有关锁申请的时间,提高了程序的性能.

锁偏向在没有锁竞争的场合可以有较好的优化效果,对于锁竞争比较激烈的场景,效果不佳, 锁竞争激烈的情况下可能是每次都是不同的线程来请求锁,这时偏向模式失效.

2 轻量级锁

如果锁偏向失败,JVM 不会立即挂起线程,还会使用一种称为轻量级锁的优化手段. 会将对象的头部作为指针,指向持有锁的线程堆栈内部, 来判断一个线程是否持有对象锁. 如果线程获得轻量级锁成功,就进入临界区. 如果获得轻量级锁失败,表示其他线程抢到了锁,那么当前线程的锁的请求就膨胀为重量级锁.当前线程就转到阻塞队列中变为阻塞状态.

偏向锁,轻量级锁都是乐观锁,重量级锁是悲观锁

一个对象刚开始实例化时,没有任何线程访问它,它是可偏向的,即它认为只可能有一个线程来访问它,所以当第一个线程来访问它的时候,它会偏向这个线程. 偏向第一个线程,这个线程在修改对象头成为偏向锁时使用 CAS 操作,将对象头中 ThreadId 改成自己的 ID,之后再访问这个对象时,只需要对比 ID 即可. 一旦有第二个线程访问该对象,因为偏向锁不会主动释放,所以第二个线程可以查看对象的偏向状态,当第二个线程访问对象时,表示在这个对象上已经存在竞争了,检查原来持有对象锁的线程是否存活,如果挂了则将对象变为无锁状态,然后重新偏向新的线程; 如果原来的线程依然存活,则马上执行原来线程的栈,检查该对象的使用情况,如果仍然需要偏向锁,则偏向锁升级为轻量级锁.

轻量级锁认为竞争存在,但是竞争的程度很轻,一般两个线程对同一个锁的操作会错开,或者稍微等待一下(自旋)另外一个线程就会释

放锁. 当自旋超过一定次数,或者一个线程持有锁,一个线程在自旋,又来第三个线程访问时，轻量级锁会膨胀为重量级锁，重量级锁除了持有锁的线程外,其他的线程都阻塞.