**JAVA并发编程**

**一、线程安全性：**

* 要编写线程安全的代码，其核心就是要对状态访问操作进行管理，特别是共享的(Shared)和可变的(Mutable)状态的访问；
* “共享”：多个线程可访问同一变量；
* “可变”：变量值在声明周期内可变化;
* java的同步机制：独占锁synchronized, volatile, 显示锁Lock及原子变量;
* 编写并发程序的原则：1.代码正确运行;2.提高代码速度(需求所需时);

**什么是线程安全性：**

* 其核心概念就是正确性：即某个类的行为与其规范完全一致：
* 线程安全性：当多个线程访问某个类时，这个类始终都能表现出正确的行为，这个类就是线程安全的；
* 无状态对象一定是线程安全的，如：

/\*\*

\* 该类无任何属性域，且不包含任何其他类中域的引用

\* 即计算中的状态都为临时状态，保存在线程栈中

\* 由JMM知线程栈线程私有，仅由当前线程可访问

\* 因此线程安全

\*/

@ThreadSafe

public class StatelessFactorizer implements Servlet {

@Override

public void service(ServletRequest req, ServletResponse repo) {

Map<String, Object> params = extractFromRequest(req);

Map<String, Object> res = doBussiness(params);

reponseTo(res);

}

...

}

**原子性：**

看一个非线程安全的版本：

/\*\*

\* 竞态条件(由于不恰当的执行时序而出现不正确的结果)导致非线程安全

\*/

@NotThreadSafe

public class UnsafeCountingFactorizer implements Servlet {

private long count = 0;

@Override

public void service(ServletRequest req, ServletResponse repo) {

count++; //该操作并非原子(读取--自增--写入), 导致非线程安全

Map<String, Object> params = extractFromRequest(req);

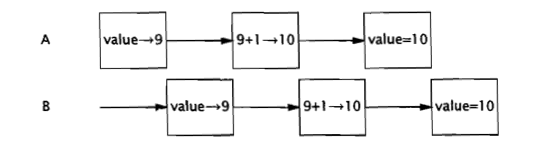
Map<String, Object> res = doBussiness(params);

reponseTo(res);

}

}

例如这种情况，预计值为11，结果为10：



* 用例：延迟初始化的竞态条件：

看个实例：

/\*\*

\* 延迟初始化的竞态条件

\* 由于多个线程可能同时执行到instance == null,

\* 或者由于实例初始化过程比较耗费时间,

\* 这样有可能所谓的"单例"不再单例

\*/

@NotThreadSafe

public class LazyInitRace {

private ExpensiveObject instance;

private LazyInitRace(){}

public ExpensiveObject getInstance() {

if (instance == null) {

instance = new ExpensiveObject();

}

return instance;

}

}

**复合操作:**

* 原子操作：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(要么全部执行，或全部不执行);
* 复合操作：包含一组以原子方式执行的操作，如(检查再初始化, 读取－修改－写入);
* 可通过java提供的原子变量实现原子操作，修改上面的计数器：

/\*\*

\* 可通过java提供的原子变量(Atomic\*)

\* 来解决竞态条件引起的非线程安全

\*/

@ThreadSafe

public class CountingFactorizer implements Servlet {

private AtomicLong count = new AtomicLong(0);

@Override

public void service(ServletRequest req, ServletResponse repo) {

count.incrementAndGet(); //原子增加,该方法最后会调用Unsafe.compareAndSwapLong本地方法

Map<String, Object> params = extractFromRequest(req);

Map<String, Object> res = doBussiness(params);

reponseTo(res);

}

       ...

}

* 当无状态的类中，添加一个状态时，若该状态由线程安全的对象管理，那么这个无状态的类也是线程安全的，如上面这段代码，但是当添加多个状态时就不一定了。

**加锁机制：**

* 对于多个由线程安全的对象管理起来的状态组合起来，且几个状态有关联关系时，这时就可能不线程安全了，如：

/\*\*

\* 多个关联线程安全的状态对象导致线程安全

\* 这里我们对lastNumber进行缓存，若请求的值与其相同，直接返回，反之计算

\*/

@NotThreadSafe

public class UnsafeCachingFactorizer implements Servlet {

private final AtomicReference<BigInteger> lastNumber =

new AtomicReference<>();

private final AtomicReference<BigInteger[]> lastFactors =

new AtomicReference<>();

@Override

public void service(ServletRequest req, ServletResponse repo) {

BigInteger i = extractFromRequest(req);

//A线程发现不等，重新计算;(A未计算完)

//B线程进入也发现不等，也重新计算(但其实有可能经过A计算后是相等的)

//从而达不到缓存效果

if (i.equals(lastNumber.get())){

reponseTo(i, lastFactors);

} else{

BigInteger[] factors = factor(i);

lastNumber.set(i);

lastFactors.set(factors);

reponseTo(i, factors);

}

}

...

}

* 所以要保持状态的一致性，就需要再单个原子操作中更新所有相关的状态变量。

**内置锁：**

* java提供的内置锁机制实现: synchronized;
* 通过synchronized我们可以简单使上面的代码线程安全，但这样性能极低，每次只允许一个请求得到响应，凡人难以接受：

/\*\*

\* 通过synchronized实现线程安全，但性能低下

\*/

@ThreadSafe

public class CachingFactorizer implements Servlet {

private BigInteger lastNumber = new BigInteger("");

private BigInteger[] lastFactors = new BigInteger[]{};

@Override

public synchronized void service(ServletRequest req, ServletResponse repo) {

BigInteger i = extractFromRequest(req);

if (i.equals(lastNumber)){

reponseTo(i,lastFactors);

} else{

BigInteger[] factors = factor(i);

lastNumber = i;

lastFactors = factors;

reponseTo(i, factors);

}

}

}

**重入：**

* 重入：一个已经持有某对象锁的线程再次请求该该对象锁时，这个请求会成功(synchronized允许重入)；
* “重入”：意味着获取锁的粒度是“线程”而不是“调用”；

**用锁来保护状态：**

* 对于可能被多个线程同时访问的可变状态变量，在访问时需要持有同一个锁，在这种情况下，我们称状态变量是由这个锁保护的；
* 每个共享的和可变的变量都应该只由一个锁来保护，从而使维护人员知道是哪一个锁

**活跃性与性能：**

* 为了提升上面Servlet处理请求吞吐量，明显不能对整个方法synchronized, 下面对这个实现进行分段synchronized, 但仍必须保证多个状态变化是原子操作;

/\*\*

\* 通过分段synchronized提升性能

\*/

@ThreadSafe

public class CachedFactorizer implements Servlet {

private BigInteger lastNumber = new BigInteger("");

private BigInteger[] lastFactors = new BigInteger[] {};

private long hits;

private long cacheHits;

@Override

public void service(ServletRequest req, ServletResponse repo) {

BigInteger i = extractFromRequest(req);

BigInteger[] factors = null;

synchronized (this) {

++hits;

if (i.equals(lastNumber)) {

++cacheHits;

factors = lastFactors;

}

}

if (factors == null) {

factors = factor(i);

synchronized (this) {

lastNumber = i;

lastFactors = factors;

}

}

reponseTo(i, lastFactors);

}

       ...

}

* 通常，在简单性与性能之间存在着相互制约的因素。当实现某个同步策略时，一定盲目地为了性能而牺牲简单性(这可能破坏安全性)；
* 当执行时间较长的计算或者可能无法快速完成的操作时(若网络I/O,控制台I/O), 一定不要持有锁。

# 对象的共享：

* 要编写并发程序，关键在于：在访问共享的可变状态时需要进行正确的管理

## 可见性：

/\*\*

\* 可见性问题导致，程序运行结果不正确

\* 有可能由于编译器，处理器及运行时做一些重排序

\*/

public class Novisibility {

private static boolean ready;

private static int number;

private static class ReaderThread extends Thread{

@Override

public void run() {

while (!ready){

Thread.yield(); //主动让出cpu, 进入就绪队列

}

System.out.println(number);

}

}

public static void main(String[] args) {

new ReaderThread().start();

number = 42;

ready = true;

}

}

* 失效数据：就如同上面的代码，没有同步的情况下可能产生错误的结果。

又如：

/\*\*

\* get操作可能与最近set值不一致，产生数据失效

\*/

@NotThreadSafe

public class MutableInteger {

private int value;

public int getValue() {

return value;

}

public void setValue(int value) {

this.value = value;

}

}

可做如下修改：

/\*\*

\* 将get, set同步化，可防止数据失效

\*/

@ThreadSafe

public class MutableInteger {

private int value;

public synchronized int getValue() {

return value;

}

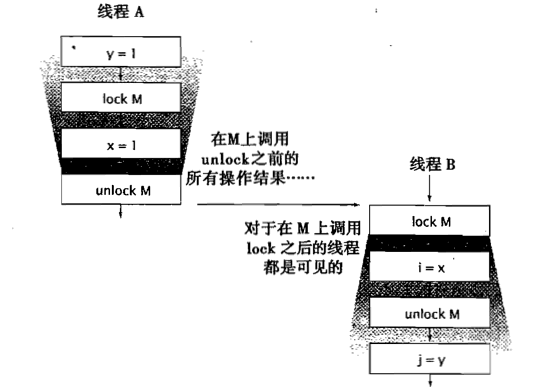
public synchronized void setValue(int value) {

this.value = value;

}

}

* **非原子的64位操作**：对于非volatile类型的long和double, jvm会将64位的读写操作分解为2个32位操作(java虚拟机栈中的操作数栈每个slot为32位)，对于这种变量在多线程读写操作下，有可能读取到某个值的高32位或某个值的低32位，建议在多线程环境下，对共享可变的long或double变量进行volatile声明。
* 加锁与可见性：加锁的含义不仅局限于互斥行为，还包括内存可见性。为了确保所有线程能看到共享可变变量的最新值，所有执行读写操作的线程必须在同一个锁上同步。



* Volatile变量：之前转载过一篇相关文章(<http://my.oschina.net/indestiny/blog/208541>), 当变量声明为volatile后，那么编译器或运行时会主要这个变量是共享的，不会将该变量上的操作与其他内存操作一起重排序。volatile变量不会被缓存在寄存器或其他处理器不可见的地方，因此读取volatile变量总会返回最新的值。

      比较典型的用法：

private volatile boolean asleep;

...

while (!asleep){

    // do sth.

}

* 加锁机制既可以确保可见性又可以确保原子性，而volatile变量只可确保可见性，所以说volatile是一种轻量级同步机制；
* 满足以下所有条件时，可用volatile:

       1. 对变量的写入操作不依赖变量的当前值，或者能确保只有一个线程更新变量的值；

       2. 该变量不会与其他状态变量一起纳入不变性条件；

       3. 访问该变量不需要加锁。

## 发布与逸出：

* 发布对象：使对象能够在当前作用域之外被使用, 如：

//对共有静态变量的发布，集合内部的变量也会被发布

public static Set<Object> publishedObject;

public void init(){

    publishedObject = new HashSet<>();

}

//通过公有方法发布

private Object publishedObject;

public Object get(){

return publishedObject;

}

/\*\*

\* 通过发布类的内部实例

\* this引用被逸出

\*/

public class ThisEscape {

public ThisEscape(EventSource source){

source.registerListener(new EventListener() {

@Override

public void onEvent(Event e) {

// ThisEscape.this实例逸出，但此时该实例并没有构造完成

}

});

}

}

### 安全的对象构造过程：

* 不要在构造过程中使this引用逸出，如上面的ThisEscape；

      可通过工厂方法避免this逸出：

/\*\*

\* 通过工厂方法防止this逸出

\*/

public class SafeListener {

private final EventListener listener;

private SafeListener(){

listener = new EventListener() {

@Override

public void onEvent(Event e) {

//do sth.

}

};

}

public static SafeListener newInstance(EventSource source){

SafeListener safe = new SafeListener();

source.registerListener(safe.listener);

return safe;

}

}

## 线程封闭：

* 线程封闭：在单线程内访问数据，不需要同步，这是实现线程安全最简单的方式。
* Ad-hoc线程封闭：维护线程封闭性的职责完全由程序实现来承担，它很脆弱，因为没有一种语言特性能将对象封闭到目标线程上。
* 栈封闭：是线程封闭的一种特例，在栈封闭中，只能通过局部变量才能访问对象。比Ad-hoc线程封闭更易维护，强壮。
* ThreadLocal类：一种维持线程封闭更规范的方法，它会为每个使用ThreadLocal变量的线程存放一份独立的副本，因此对于该变量线程之间不会相互干扰，你可以想ThreadLocal<T>想成Map<Thread, T>, 当线程终止时 ，这些值会作为垃圾回收。比如，多线程环境下，我们可以这样获取数据库连接：

private static ThreadLocal<Connection> connectionHolder =

new ThreadLocal<Connection>(){

@Override

protected Connection initialValue() {

try {

return DriverManager.getConnection("DB\_URL");

} catch (SQLException e) {

e.printStackTrace();

}

return null;

}

};

public static Connection getConnection(){

//不同线程每次得到的Connection, 都是独立的备份

return connectionHolder.get();

}

## 不变性：

* 不可变对象一定是线程安全的；
* 对象不可变应满足：

      1.对象创建后其状态不能修改；

      2.对象的所有域都是final类型；

      3.对象是正确创建的(在对象创建期间，this未逸出)。

* Final域：final类型的域不能修改(但其指向的引用对象可修改)，重新改造之前文章的CachedFactorizer:

/\*\*

\* 不可变类：

\* 所有域都是final

\*/

public class OneValueCache {

private final BigInteger lastNumber;

private final BigInteger[] lastFactors;

public OneValueCache(BigInteger lastNumber, BigInteger[] lastFactors) {

this.lastNumber = lastNumber;

this.lastFactors = lastFactors;

}

public BigInteger[] getFactors(BigInteger i){

if (lastNumber == null || ! lastNumber.equals(i)){

return null;

} else{

return Arrays.copyOf(lastFactors, lastFactors.length);

}

}

}

/\*\*

\* 使用Volatile类型发布不可变对象

\*/

@ThreadSafe

public class VolatileCachedFactorizer implements Servlet {

private volatile OneValueCache cache = new OneValueCache(null, null); //volatile保证每次写后最新值对其他线程可见

@Override

public void service(ServletRequest req, ServletResponse repo) {

BigInteger i = extractFromRequest(req);

BigInteger[] factors = cache.getFactors(i);

if (factors == null) {

factors = factor(i);

cache = new OneValueCache(i, factors);

}

reponseTo(i, factors);

}

}

## 安全发布：

* 不正确的发布：正确的对象被破坏：

/\*\*

\* 多线程访问下，有可能出错，问题不在Holder本身，而在于未正确地发布，可将n声明为final，避免不正确发布

\*/

public class Holder {

private int n;

public Holder(int n){

this.n = n;

}

public void assertSanity(){

if (n != n){

throw new AssertionError("");

}

}

}

### 不可变对象与初始化安全性：

* 任何线程都可以在不需要额外同步地情况下安全地访问不可变对象，即使在发布这些对象时没有使用同步。

### 安全发布地常用模式：

* 一个正确构造的对象可以通过以下方式来安全地发布：

      1.在静态初始化函数中初始化一个对象引用;

      2.将对象的引用保存到volatile类型地域或AtomicReference对象中；

      3.将对象的引用保存到某个正确构造对象地final类型域中；

      4.将对象的引用保存到一个由锁保护的域中。

### 事实不可变对象：

* 如果对象从技术上来看是可变的，但其状态在发布后不会再改变，称这种对象为"事实不可变对象"。
* 在没有额外的同步的情况下，任何线程都可以安全地使用被安全发布的事实不可变对象。

### 可变对象：

* 可变对象：对象构造后，其状态可以发生改变；
* 对象的发布需求取决于它的可变性：

       1.不可变对象可以通过任意机制来发布；

       2.事实不可变对象必须通过安全方式来发布；

       3.可变对象必须通过安全方式来发布，并且必须是线程安全的或由某个锁保护起来。

### 安全地共享对象：

* 在并发程序中使用和共享对象时，可以使用一些使用的策略，包括：

      1.线程封闭。线程封闭的对象只能由一个线程拥有，对象被封闭在该线程中，只允许这个线程修改；

      2.只读共享。在没有同步的情况下，共享的只读对象可以由多个线程并发访问，但任何线程不能修改它。共享的只读对象包括不可变对象和事实不可变对象。

      3.线程安全共享。线程安全的对象在其内部实现同步，多个线程可以通过公有接口对其访问而不需进一步同步；

      4.保护对象。被保护对象只能通过持有特定锁来访问。保护对象包括封装在其他线程安全对象中的对象，以及已发布的并且由某个特定锁保护的对象。

# 三、对象的组合：

* 如何将现有的线程安全组件，组合成我们想要的更大规模的程序。

## 设计线程安全的类：

* 设计线程安全类的三个要素：

       1.找出构成对象状态的所有变量；

       2.找出约束状态变量的不变性条件；

       3.建立对象状态的并发访问管理策略。

### 收集同步需求：

* 如果不了解对象的不变性条件与后验条件，那就不能确保线程安全性。要满足在状态变量的有效值或状态转换上的各种约束条件，就需要借助原子性和封装性。

### 依赖状态的操作：

* 如果在某个操作中包含有基于状态的先验条件，那么这个操作就称为依赖状态的操作。如在操作前，判断当前状态是否可以进行当前操作。

### 状态的所有权：

* 所有权与封装性总是相互关联的：对象封装它拥有的状态，即对它封装的状态拥有所有权，当然所有权可以通过传递对象，变成共享所有权。

## 实例封闭：

* 将数据封装在对象内部，可以将数据的访问限制在对象的方法上，从而更容易确保线程在访问数据时总能持有正确的锁。

/\*\*

\* 这里将mySet实例封闭在PersonSet中，

\* 尽管HashSet是非线程安全类，

\* 由于mySet是私有且不会逸出的,

\* 我们通过公共接口提供给外部访问，但加上了PersonSet内置锁保护synchronized,

\* 因而PersonSet是一个线程安全的类

\*/

@ThreadSafe

public class PersonSet {

private final Set<Person> mySet = new HashSet<>();

public synchronized void addPerson(Person p){

mySet.add(p);

}

public synchronized boolean containsPerson(Person p){

return mySet.contains(p);

}

}

* 封闭机制更易于构造线程安全的类，因为当封闭类的状态时，在分析类的线程安全性时就无须检查整个程序。

### Java监视器模式：

* Java监视器模式的对象会把对象的所有可变状态都封装起来，并由对象自己的内置锁来保护。如Vector, Hashtable等类；
* 我们也可以通过私有锁来代替内置锁：

public class PrivateLock {

private final Object lock = new Object();

public void methodOne(){

synchronized(lock){

// do sth.

}

}

}

## 线程安全性的委托：

* 多个线程安全的类组合成的类，不一定就是线程安全的。

/\*\*

\* 委托的PersonSet

\* 将内部操作委托给线程安全的类SynchronizedSet

\* 从而自身也是线程安全的

\*/

@ThreadSafe

public class DelegatingPersonSet {

private final Set<Person> mySet =

Collections.synchronizedSet(new HashSet<Person>());

public void addPerson(Person p){

mySet.add(p);

}

public boolean containsPerson(Person p){

return mySet.contains(p);

}

}

### 独立的状态变量：

* 我们还可以将线程安全性委托给多个状态变量，只要这些状态变量彼此独立(不相关)：

/\*\*

\* 将线程安全性委托给多个彼此独立的状态变量

\* VisualComponent使用CopyOnWriteArrayList(线程安全)来保存监听器列表

\* keyListeners, mouseListeners彼此独立

\* 因此VisualComponent线程安全

\*/

public class VisualComponent {

private final List<KeyListener> keyListeners =

new CopyOnWriteArrayList<>();

private final List<MouseListener> mouseListeners =

new CopyOnWriteArrayList<>();

public void addKeyListener(KeyListener keyListener){

keyListeners.add(keyListener);

}

public void removeKeyListener(KeyListener keyListener){

keyListeners.remove(keyListener);

}

public void addMouseListener(MouseListener mouseListener){

mouseListeners.add(mouseListener);

}

public void removeMouseListener(MouseListener mouseListener){

mouseListeners.remove(mouseListener);

}

}

### 当委托失效时：

* 当类内部多个状态变量，他们之间存在不变性条件，即使这些状态变量各自是线程安全的，那么该类不一定就线程安全：??????

/\*\*

\* NumberRange不足以保护它的不变性条件

\* 并发环境下不安全

\*/

@NotThreadSafe

public class NumberRange {

//不变性条件: lower <= upper

private final AtomicInteger lower = new AtomicInteger();

private final AtomicInteger upper = new AtomicInteger();

public void setLower(int i){

if (i > upper.get()){ //不安全的检查

throw new IllegalArgumentException("lower can't > upper");

}

lower.set(i);

}

public void setUpper(int i){

if (i < lower.get()){ //不安全的检查

throw new IllegalArgumentException("lower can't > upper");

}

upper.set(i);

}

}

* 如果一个类是由多个独立且线程安全的状态变量组成，并且在所有的操作中都不包含无效状态转换，那么可以将线程安全性委托给底层的状态变量。

### 发布底层的状态变量：

* 如果一个状态变量是线程安全的，并且没有任何不变性条件来约束它的值，在变量的操作上也不存在任何不允许的状态转换，那么就可以安全地发布这个变量，例如发布上面VisualComponent的keyListeners, mouseListeners。

## 在现有的线程安全类中添加功能：

* 通过扩展类，来添加功能。

/\*\*

\* 通过扩展实现非重复Vector

\*/

public class NoRepeatVector<E> extends Vector<E> {

public synchronized boolean putIfAbsent(E e){

boolean exist = contains(e);

if (!exist)

add(e);

return exist;

}

}

### 客户端加锁机制：

* 客户端加锁：对于使用某个对象X的客户端代码，使用X本身用于保护其状态的锁来保护这段客户端代码。

/\*\*

\* 这段客户端代码看似线程安全，

\* 但其实并不安全，因为锁住的对象不正确，

\* 这里仅是锁住ListHelper对象，但list对象并没有被锁住，

\* 其他客户端仍可在不安全情况下对list进行操作

\*/

@NotThreadSafe

public class ListHelper<E> {

public List<E> list =

Collections.synchronizedList(new ArrayList<E>());

public synchronized boolean putIfAbsent(E x){

boolean absent = !list.contains(x);

if (absent)

list.add(x);

return absent;

}

}

所以上面的代码，我们应该对list加锁，而不是ListHelper对象：

@ThreadSafe

public class SafeListHelper<E> {

public List<E> list =

Collections.synchronizedList(new ArrayList<E>());

public boolean putIfAbsent(E x){

synchronized (list) {

boolean absent = !list.contains(x);

if (absent)

list.add(x);

return absent;

}

}

}

### 组合：

* 当为现有的类添加一个原子操作时，有一种更好的方法：组合(Composition)。

/\*\*

\* 通过组合实现"若没有则添加" 下午4:48:42

\*/

@ThreadSafe

public class improvedList<T> implements List<T> {

private final List<T> list;

public improvedList(List<T> list) {

this.list = list;

}

public synchronized boolean putIfAbsent(T t){

boolean absent = !list.contains(t);

if (absent)

list.add(t);

return absent;

}

@Override

public synchronized int size() {

return list.size();

}

       ...

}

## 将同步策略文档化：

* 在文档中说明客户代码需要了解的线程安全性保证，以及代码维护人员需要了解的同步策略。

# 四、基础构建模块：

* 委托是创建线程安全类的一个最有效的策略：只需让现有的线程安全类管理所有的状态即可。

## 同步容器类：

* 同步容器类包括：Vector, Hashtable及Collections.synchronizedXxx()方法产生的实例。
* 同步容器类是线程安全的，但在某些情况下需要客户端加锁保护来实现一些复合操作。
* 常见复合操作：迭代，跳转，条件运算，如"若没有则添加"。

如下面的复合操作就有可能不安全：

/\*\*

\* getLast, rmLast没有同步，可能导致lastIndex错乱

\*/

@NotThreadSafe

public class UnsafeVector<E> {

private final Vector<E> v = new Vector<>();

public E getLast(){

int lastIndex = v.size()-1;

return v.get(lastIndex);

}

public E rmLast(){

int lastIndex = v.size()-1;

return v.remove(lastIndex);

}

}

* 由于同步容器类要遵守同步策略，即支持客户端加锁，上面代码可以通过客户端加锁实现线程安全：

/\*\*

\* 通过客户端加锁实现线程安全

\*/

@ThreadSafe

public class SafeVector<E> {

private final Vector<E> v = new Vector<>();

public E getLast(){

synchronized (v) {

int lastIndex = v.size()-1;

return v.get(lastIndex);

}

}

public E rmLast(){

synchronized(v){

int lastIndex = v.size()-1;

return v.remove(lastIndex);

}

}

}

### 迭代器与ConcurrentModificationException:

* 容器在迭代过程中被修改时 ，就会抛出一个ConcurrentModificationException异常。

/\*\*

\* 下面将会抛出:ConcurrentModificationException

\* 可通过在迭代前锁住vector, 但这样会损失并发性能

\*/

@NotThreadSafe

public class ModificationExceptionVector {

public static void main(String[] args) {

Vector<Person> vector = new Vector<>();

for (int i=0; i<10; i++){

vector.add(new Person(i, "person" + i));

}

new Thread(new IterateThread(vector)).start();

new Thread(new RemoveThread(vector)).start();

}

private static class RemoveThread implements Runnable{

private Vector<Person> v;

private Random ran = new Random();

public RemoveThread(Vector<Person> v) {

this.v = v;

}

@Override

public void run() {

try {

// do 100 times' remove

for (int i=0 ;i<5; i++){

v.remove(ran.nextInt(v.size()));

Thread.sleep(500);

}

} catch (InterruptedException e) {

}

}

}

private static class IterateThread implements Runnable{

private Vector<Person> v;

public IterateThread(Vector<Person> v) {

this.v = v;

}

@Override

public void run() {

try {

Iterator<Person> it = v.iterator();

while (it.hasNext()){

System.out.println(it.next());

Thread.sleep(500);

}

} catch (InterruptedException e) {

}

}

}

}

### 隐藏迭代器：

* 正如封装对象的状态有助于维持不变性条件一样，封装对象的同步机制同样有助于确保实施同步策略。
* 一些隐藏的迭代操作：hashCode, equals, containsAll, removeAll, retainAll等。

## 并发容器：

* 通过并发容器来代替同步容器，可以极大地提高伸缩性并降低风险。

### ConrrentHashMap:

之前有一篇文章介绍过ConcurrentHashMap: <http://my.oschina.net/indestiny/blog/209458>

* ConcurrentHashMap使用一种粒度更细的加锁机制来实现大程度的共享，这种机制称为分段锁(Lock Striping);
* ConcurrentHashMap的迭代器不会抛出ConcurrentModificationException,因此不需要在迭代过程中加锁，因为其返回的迭代器具有弱一致性，而非"及时失败"。
* ConcurrentHashMap对一些操作进行了弱化，如size(计算的是近似值，而不是精确值), isEmpty等。

### 额外的原子Map操作：

* ConcurrentMap声明了一些原子操作接口：

public interface ConcurrentMap<K, V> extends Map<K, V> {

V putIfAbsent(K key, V value);

boolean remove(Object key, Object value);

boolean replace(K key, V oldValue, V newValue);

V replace(K key, V value);

}

### CopyOnWriteArrayList:

* CopyOnWriteArrayList比同步List具有更高的并发性能，而且在迭代时不需要加锁或复制。
* 其安全性在于：只要发布一个事实不可变的对象，那么在访问该对象时就不需要进一步同步；在每次修改都会创建一个新的容器副本，从而实现可变性。
* 仅当迭代操作远远多于修改操作时，才应该使用"写入时复制"容器。比如事件通知系统，对监听器列表中的每个监听器进行通知。

## 阻塞队列和生产者--消费者模式：

* 在构建高可靠的应用程序时，有界队列是一种强大的资源管理工具；它们能够意志或防止产生过多的工作项，使应用程序在负荷过载的情况下变得更加健壮。
* BlockingQueue实现：LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue,SynchronousQueue;

### 串行线程封闭：

* 对于可变对象，生产者--消费者这种设计与阻塞队列一起，促进了串行线程封闭，从而将对象所有权从生产者交付给消费者。

### 双端队列与工作密取：

* java6提供了双端队列：ArrayDeque, LinkedBlockingDeque；
* 双端队列适用于另一种模式：工作密取，每个消费者有各自的双端队列，这种模式非常适合既是消费者又是生产者问题。
* 当消费者自己的双端队列为空时，它会从其他消费者队列末尾中密取任务。

## 阻塞方法与中断方法：

* 阻塞的原因：等待I/O操作结束，等待获得一个锁，等待从Thread.sleep方法中醒来，或是等待另一个线程的计算结果等。
* 传递InterreuptedException: 抛出异常给方法调用者，或捕获异常，做一些清理工作再抛出抛出异常。
* 恢复中断：有时不能抛出InterruptedException, 比如在Runnable中，则可以恢复中断。

/\*\*

\* 恢复中断状态以避免屏蔽中断

\*/

public class TaskRunnable implements Runnable {

private final BlockingQueue<Task> queue;

public TaskRunnable(BlockingQueue<Task> queue) {

this.queue = queue;

}

@Override

public void run() {

try {

doTask(queue.take());

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

       ...

}

## 同步工具类：

* 任何一个对象都可以是同步工具类，java平台提供的一些同步工具类有：Semaphore(信号量), Barrier(栅栏), Latch(闭锁)；

### 闭锁：

* 闭锁可以用来确保某些活动直到其他活动都完成后才继续执行；

一个计算多个线程启动到结束耗时的例子：

/\*\*

\* 在计时测试中使用CountDownLatch来启动和停止线程

\*/

public class TestHarness {

public long timeTasks(int nThreads, final Runnable task) throws InterruptedException{

final CountDownLatch startGate = new CountDownLatch(1); //所有线程同时开始执行task的阀门

final CountDownLatch endGate = new CountDownLatch(nThreads); //所有线程结束的阀门

for (int i=0; i<nThreads; i++){

Thread t = new Thread(){

@Override

public void run() {

try {

startGate.await(); //等待startGate值减为0

try {

task.run();

} finally{

endGate.countDown(); //一个线程运行结束，值减1

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

};

t.start();

}

long start = System.nanoTime();

startGate.countDown(); //所有线程开始执行task

endGate.await(); //等待所有线程执行结束

long end = System.nanoTime();

return end - start;

}

}

### FutureTask:

* FutureTask也可用做闭锁，表示一种抽象的可生成结果的计算。

/\*\*

\* 使用FutureTask来提前加载稍后需要的数据

\*/

public class Preloader {

private final FutureTask<ProductInfo> future = new FutureTask<>(

new Callable<ProductInfo>() {

@Override

public ProductInfo call() throws Exception {

return loadProductInfo();

}

});

private final Thread thread = new Thread(future);

public void start() {

thread.start();

}

private ProductInfo loadProductInfo() {

// TODO Auto-generated method stub

return null;

}

public ProductInfo get() throws InterruptedException {

try {

return future.get();

} catch (ExecutionException e) {

// exception handle

return null;

}

}

}

### 信号量：

* 计数信号量用来控制同时访问某个特定资源的操作数量，或者同时执行某个制定操作的数量，也可以用来实现某种资源池，或者对容器施加边界。

/\*\*

\* 使用Semaphore为容器设置边界

\*/

public class BoundedHashSet<T> {

private final Set<T> set;

private final Semaphore sem;

public BoundedHashSet(int bound){

this.set = Collections.synchronizedSet(new HashSet<T>());

sem = new Semaphore(bound); //非公平

}

public boolean add(T t) throws InterruptedException{

sem.acquire(); //请求semaphore, permits-1或阻塞到permits > 0

boolean wasAdded = false;

try {

wasAdded = set.add(t);

return wasAdded;

} finally{

if (!wasAdded) //未添加成功则释放semaphore

sem.release();

}

}

public boolean remove(T t){

boolean wasRemoved = set.remove(t);

if (wasRemoved) //删除成功permits+1;

sem.release();

return wasRemoved;

}

}

### 栅栏：

* 栅栏(Barrier)类似于闭锁，它能阻塞一组线程直到某个事件发生。栅栏与闭锁的关键区别在于，所有线程必须同时到达栅栏位置，才能继续执行。闭锁用于等待事件(CutDownLatch值减为0)，栅栏用于等待其他线程。

/\*\*

\* CyclicBarrier测试

\*/

public class CyclicBarrierTest {

public static void main(String[] args) {

int threadCount = 3;

CyclicBarrier barrier =

new CyclicBarrier(threadCount, new Runnable() {

@Override

public void run() { //最后一个线程到达栅栏时触发

System.out.println("all have finished.");

}

});

for (int i=0 ;i<threadCount; i++){

new Thread(new WorkThread(barrier)).start();

}

}

private static class WorkThread implements Runnable{

private CyclicBarrier barrier;

public WorkThread(CyclicBarrier barrier) {

this.barrier = barrier;

}

@Override

public void run() {

System.out.println(

Thread.currentThread().getId() + " Working...");

try {

barrier.await(); //当前线程阻塞直到最后一个线程到达

System.out.println(Thread.currentThread().getId() + " awaiting finished.");

} catch (InterruptedException | BrokenBarrierException e) {

e.printStackTrace();

}

}

}

}

* 除Barrier栅栏外，还有Exchanger栅栏，它是一种两方栅栏, 可以实现两个线程之间交换数据。

/\*\*

\* 通过Exchanger交换2个线程数据

\*/

public class ExchangerTest {

public static void main(String[] args) {

Exchanger<String> exchanger = new Exchanger<>();

ExchangerRunnable exchangerRunnable1 =

new ExchangerRunnable(exchanger, "A");

ExchangerRunnable exchangerRunnable2 =

new ExchangerRunnable(exchanger, "B");

new Thread(exchangerRunnable1).start();

new Thread(exchangerRunnable2).start();

}

private static class ExchangerRunnable implements Runnable{

private Exchanger<String> exchanger;

private String data;

public ExchangerRunnable(Exchanger<String> exchanger, String data){

this.exchanger = exchanger;

this.data = data;

}

@Override

public void run() {

try {

String beforeData = this.data;

//阻塞直到另一个线程调用exchanger.exchange(), 交换数据

this.data = this.exchanger.exchange(this.data);

System.out.println(

Thread.currentThread().getName() +

" exchanged " + beforeData + " for " + this.data

);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

## 构建高效且可伸缩的结果缓存：

* 一个简单安全，性能低下的缓存设计：

/\*\*

\* 计算缓存器

\* 内部使用HashMap实现计算结果的缓存

\* 通过外部接口同步操作实现线程安全

\* 但有可能由于计算时间过长导致性能低下

\*/

public class Memoizer1<A, V> implements Computable<A, V> {

private final Map<A, V> cache = new HashMap<A, V>();

private final Computable<A, V> c;

public Memoizer1(Computable<A, V> c) {

this.c = c;

}

@Override

public synchronized V compute(A key) throws InterruptedException {

V result = cache.get(key);

if (result == null){

result = c.compute(key); //计算

cache.put(key, result);

}

return result;

}

}

* 通过并发容器ConcurrentHashMap代替HashMap，提升并发性能：

/\*\*

\* 计算缓存器

\* 通过ConcurrentHashMap代替HashMap, 提升并发性能

\* 但这样有可能多个线程同时调用compute方法，

\* 由于计算过程中还没有结果，有可能导致多个线程计算同样的值

\*/

public class Memoizer2<A, V> implements Computable<A, V> {

private final Map<A, V> cache = new ConcurrentHashMap<A, V>();

private final Computable<A, V> c;

public Memoizer2(Computable<A, V> c) {

this.c = c;

}

@Override

public V compute(A key) throws InterruptedException {

V result = cache.get(key);

if (result == null){

result = c.compute(key); //计算

cache.put(key, result);

}

return result;

}

}

* 通过FutureTask来弥补重复结果计算问题：

/\*\*

\* 计算缓存器

\* 通过FutureTask代替map中的Value

\* 这样可以在计算结果计算完成，就立即返回，

\* 但仍然有可能重复计算，因为存在非原子的复合操作"若没有则添加": if (f == null){...}

\*/

public class Memoizer3<A, V> implements Computable<A, V> {

private final Map<A, Future<V>> cache = new ConcurrentHashMap<A, Future<V>>();

private final Computable<A, V> c;

public Memoizer3(Computable<A, V> c) {

this.c = c;

}

@Override

public V compute(final A key) throws InterruptedException {

Future<V> f = cache.get(key);

if (f == null){

Callable<V> computeTask = new Callable<V>() {

@Override

public V call() throws Exception {

return c.compute(key);

}

};

FutureTask<V> ft = new FutureTask<>(computeTask);

f = ft;

cache.put(key, ft);

ft.run(); //执行计算

}

try {

return f.get(); //获取计算结果

} catch (ExecutionException e) {

//do exception handle

}

return null;

}

}

* 通过对CocurrentHashMap.putIfAbsent()对上面的问题进行修复：

/\*\*

\* 计算缓存器

\* 通过ConcurrentHashMap.putIfAbsent避免重复任务

\*/

public class Memoizer<A, V> implements Computable<A, V> {

private final ConcurrentHashMap<A, Future<V>> cache = new ConcurrentHashMap<A, Future<V>>();

private final Computable<A, V> c;

public Memoizer(Computable<A, V> c) {

this.c = c;

}

@Override

public V compute(final A key) throws InterruptedException {

while(true){

Future<V> f = cache.get(key);

if (f == null){

Callable<V> computeTask = new Callable<V>() {

@Override

public V call() throws Exception {

return c.compute(key);

}

};

FutureTask<V> ft = new FutureTask<>(computeTask);

f = cache.putIfAbsent(key, ft); //该方法不会对相同key的值进行覆盖，这样避免了相同key的任务被计算

if (f == null) ft.run(); //执行计算

}

try {

return f.get(); //获取计算结果

} catch (CancellationException e){

cache.remove(key); //计算取消则移除对应的计算任务key

} catch (ExecutionException e) {

//do exception handle

}

}

}

}

一，二，三，四就讲述了java并发编程的基础知识。

# 并发技巧清单：

* 可变状态是至关重要的。

       所有并发访问都可以归结为如何协调对并发状态的访问，可变状态越少，越容易确保线程安全性。 

* 尽量将域声明为final类型，除非需要它们是可变的。
* 不可变对象一定是线程安全的。

       不可变对象能极大地降低并发编程的复杂性。它们更为简单且安全，可以任意共享而无须使用加锁或保护性复制等机制。

* 封装有助于管理复杂性。

      将数据封装在对象中，更易于维护不变性；将同步机制封装在对象中，更易于遵循同步策略。

* 用锁保护每个可变变量。
* 当保护同一个不变性条件中的所有变量时，要使用同一个锁。
* 在执行复合操作期间，要持有锁。
* 如果从多个线程中访问同一个可变变量时没有同步机制，那么程序会可能出问题。
* 不要自行推断不需要使用同步。
* 在设计过程中考虑线程安全，不要在上线出问题后再做。
* 将同步策略文档化。

# 五、任务执行

* 大多数并发应用程序都是围绕"任务执行"来构造的：任务通常是一些抽象的且离散的工作单元。

## 在线程中执行任务：

* 理想情况下，各个任务之间是相互独立的：任务并不依赖其他任务的状态，结果或边界效应。

### 串行地执行任务：

/\*\*

\* 串行处理请求:

\* 简单正确，但性能低下

\*/

public class SingleThreadWebServer {

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

ServerSocket server = new ServerSocket(80);

boolean listening = true;

while (listening){

Socket connection = server.accept(); //阻塞等待客户端连接请求

handlerRequest(connection);

}

server.close();

}

...

}

### 显示地为任务创建线程：

/\*\*

\* 为每一个用户请求创建一个线程为其服务

\*/

public class ThreadPerTaskWebServer {

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

ServerSocket server = new ServerSocket(80);

boolean listening = true;

while (listening){

final Socket connection = server.accept(); //阻塞等待客户端连接请求

Runnable task = new Runnable() {

@Override

public void run() {

handlerRequest(connection);

}

};

new Thread(task).start();

}

server.close();

}

...

}

上面的实现至少能给我们一些暗示：

* 任务处理过程从主线程分离出来，以提高主线程响应其他请求的能力。
* 任务可以并行处理，提高吞吐量。
* 任务处理代码必须线程安全。

### 但这样无限制创建会带来不足：

* 线程生命周期的开销非常高。线程的创建和销毁都是有代价的，不同平台开销也不同。
* 资源消耗。太多线程会消耗系统资源，如空闲线程的内存占用，大量线程竞争CPU时产生其他性能开销等。
* 稳定性。可创建线程数会受到限制，如jvm启动参数(如-Xss等)，Thread构造函数请求的栈大小，以及底层操作系统对线程的限制(32位机器上，主要限制因素为线程栈的寻址空间)等，破坏这些限制，很可能抛出OutOfMemoryError异常。

## Executor框架：

* 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。
* java类库中，任务执行的抽象不是Thread, 而是Executor。

/\*\*

\* 基于线程池的Web服务器

\*/

public class ThreadPerTaskWebServer {

private static final int NTHREADS = 100;

/\*\*

\* 创建固定线程数量的线程池

\*/

private static final Executor exec =

Executors.newFixedThreadPool(NTHREADS);

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

ServerSocket server = new ServerSocket(80);

boolean listening = true;

while (listening){

final Socket connection = server.accept(); //阻塞等待客户端连接请求

Runnable task = new Runnable() {

@Override

public void run() {

handlerRequest(connection);

}

};

exec.execute(task);

}

server.close();

}

...

}

### 执行策略：

执行策略需要考虑的有：

* 在什么线程中执行任务。
* 任务按照什么顺序执行(FIFO, LIFO, 优先级)。
* 有多少个任务能并发执行。
* 在队列中有多少个任务在等待执行。
* 由于过载，系统应如何拒绝任务？如果通知任务被拒绝？
* 执行任务前后，应该做什么？

### 线程池：

* 线程池：管理一组同构工作线程的资源池。

Executors提供了几种创建线程池的方法：

//创建固定长度的线程池,每当提交一个任务时就创建一个线程，直到达到线程池的最大数量，若有线程发生异常，则会重新创建

public static ExecutorService newFixedThreadPool(int nThreads) {...}

//创建单个线程来执行任务，若该线程发生异常，会创建一个新的线程。该池可按顺序执行队列中的任务(如FIFO,LIFO,优先级等)

public static ExecutorService newSingleThreadExecutor() {...}

//该线程池无长度限制，在线程过多时会回收，过少时会创建

public static ExecutorService newCachedThreadPool() {...}

//创建一个固定长度的线程池，并以延迟或定时的方式执行任务

public static ScheduledExecutorService newScheduledThreadPool(...}

### Executor的生命周期：

* 我们可以通过ExecutorService来对线程池进行生命周期的管理：

public interface ExecutorService extends Executor {

void shutdown();//平缓关闭，不接受新任务，待提交的任务执行完毕后，再关闭

List<Runnable> shutdownNow();//粗暴关闭，尝试取消所有执行中的任务，不再启动队列中尚未开始执行的任务

boolean isShutdown(); //是否已关闭

boolean isTerminated(); //是否已终止

boolean awaitTermination(long timeout, TimeUnit unit)//等待ExecutorService到达终止状态

throws InterruptedException;

...

}

* ExecutorService生命周期状态：运行，关闭，已终止。

/\*\*

\* 对线程池进行生命周期管理

\*/

public class LifecycleWebServer {

private static final int NTHREADS = 100;

private final ServerSocket server;

public LifecycleWebServer() throws IOException{

server = new ServerSocket(80);

}

/\*\*

\* 创建固定线程数量的线程池

\*/

private static final ExecutorService exec =

Executors.newFixedThreadPool(NTHREADS);

public void start() throws IOException{

while (!exec.isShutdown()){

try {

final Socket connection = server.accept(); //阻塞等待客户端连接请求

Runnable task = new Runnable() {

@Override

public void run() {

handlerRequest(connection);

}

};

exec.execute(task);

} catch (RejectedExecutionException e) {

if (!exec.isShutdown()){

//task submission is rejected

}

}

}

}

public void stap() throws IOException{

exec.shutdown(); //平缓关闭线程池

server.close();

}

private static void handlerRequest(Socket connection) {

// handle request

}

}

### 延迟任务与周期任务：

* 建议通过ScheduledThreadPoolExecutor来代替Timer,TimerTask。
* Timer基于绝对时间，ScheduledThreadPoolExecutor基于相对时间。
* Timer执行所有定时任务只能创建一个线程，若某个任务执行时间过长，容易破坏其他TimerTask的定时精确性。
* 已经调度但未执行的TimerTask将不会再执行，新的任务也不会被调度，出现"线程泄漏"，如：

/\*\*

\* 错误的Timer行为，Timer是脆弱的

\*/

public class OutOfTime {

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

Timer timer = new Timer();

timer.schedule(new ThrowTask(), 1); //第一个任务抛出异常

Thread.sleep(1000);

timer.schedule(new ThrowTask(), 1); //第二个任务将不能再执行, 并抛出异常Timer already cancelled.

Thread.sleep(5000);

System.out.println("end.");

}

static class ThrowTask extends TimerTask{

@Override

public void run() {

throw new RuntimeException("test timer's error behaviour");

}

}

}

### 串行的页面渲染器：

/\*\*

\* 串行地渲染页面元素, 性能很低下

\* 下载图片过程中有可能IO时间长阻塞，

\* CPU没能有效利用

\*/

public class SingleThreadRenderer {

void rendererPage(CharSequence source){

renderText(source);

List<ImageData> imageDatas = new ArrayList<>();

//解析文本中的图片连接

for (ImageInfo imageInfo : scanForImage(source)){

imageDatas.add(imageInfo.downloadImageData()); //下载图片

}

//渲染图片

for (ImageData data : imageDatas){

renderImage(data);

}

}

...

}

### 携带结果的任务Callable与Future:

* Executor执行任务的4个生命周期：创建，提交，开始，完成。
* Executor框架中，可以取消已提交但未开始执行的任务，对于已经开始执行的任务，只能当他们能响应中断时，才能取消，取消已经完成的任务不会有影响。

/\*\*

\* 使用Future等待图像下载

\* 将渲染过程分为:

\* IO密集型(下载图像)

\* CPU密集型(渲染页面)

\* 但这里仍然必须图片下载完成了才能看到页面，只是缩短了总时间

\*/

public class FutureRenderer {

private final ExecutorService exec = Executors.newFixedThreadPool(10);

void rendererPage(CharSequence source){

final List<ImageInfo> imageInfos = scanForImage(source); //抽出图片链接信息

Callable<List<ImageData>> task =

new Callable<List<ImageData>>() {

@Override

public List<ImageData> call() throws Exception {

List<ImageData> result =

new ArrayList<>();

for (ImageInfo imageInfo : imageInfos){

result.add(imageInfo.downloadImageData()); //下载图片

}

return result;

}

};

Future<List<ImageData>> future = exec.submit(task); //提交下载图片的任务

renderText(source); //渲染文本

try {

List<ImageData> imageDatas = future.get();//阻塞获取下载的图片

for (ImageData data : imageDatas){ //渲染图片

renderImage(data);

}

} catch (InterruptedException e) {

//重新设置线程的中断状态

Thread.currentThread().interrupt();

future.cancel(true);

} catch (ExecutionException e) {

// handle exception

}

}

...

}

### 异构任务并行化中存在的局限：

* 当异构任务之间的执行效率悬殊很大时，对于整体的性能提升来看并不是很有效。比如要是上面下载图片过程时间比渲染文本时间长很多，那么整体并发提升并不是很明显。

### CompletionService: Executor与BlockingQueue:

/\*\*

\* 使用CompletionService, 使页面元素在下载完成后立即显示出来

\* 类似Mobile中的新闻加载，图片时被异步加载的

\*/

public class Renderer {

private final ExecutorService executor;

public Renderer(ExecutorService executor){

this.executor = executor;

}

void rendererPage(CharSequence source){

final List<ImageInfo> imageInfos = scanForImage(source); //抽出图片链接信息

CompletionService<ImageData> completionService

= new ExecutorCompletionService<>(this.executor);

for (final ImageInfo imageInfo : imageInfos){

//提交下载图片的任务, 每下载一个图片就是一个任务，达到下载图片并行性

completionService.submit(new Callable<ImageData>() { //内部会将执行完后封装的Future对象放到一个BlockingQueue中

@Override

public ImageData call() throws Exception {

return imageInfo.downloadImageData();

}

});

}

renderText(source); //渲染文本

try {

for (int i=0, n=imageInfos.size(); i<n; i++){

Future<ImageData> f = completionService.take();

ImageData imageData = f.get();

renderImage(imageData); //渲染图片

}

} catch (InterruptedException e) {

//重新设置线程的中断状态

Thread.currentThread().interrupt();

} catch (ExecutionException e) {

// handle exception e.getCause()

}

}

...

}

### 为任务设置超时：

任务超时设置可通过Future的get超时版本：

V get(long timeout, TimeUnit unit)

throws InterruptedException, ExecutionException, TimeoutException;

你可以捕获其TimeoutException来做相应处理即可。

### 提交多个任务：

可以通过ExecutorService提交一组任务：

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

long timeout, TimeUnit unit) throws InterruptedException;

# 六、取消与关闭：

* 如何正确，安全地取消或关闭任务。

## 任务取消：

* 若外部代码能在某个操作正常完成之前将其置入“完成”状态，则还操作是可取消的。
* 取消操作的原因：

       1. 用户请求取消。

       2. 有时间限制的操作，如超时设定。

       3. 应用程序事件。

       4. 错误。

       5. 关闭。

如下面这种取消操作实现：

/\*\*

\* 一个可取消的素数生成器

\* 使用volatile类型的域保存取消状态

\* 通过循环来检测任务是否取消

\*/

@ThreadSafe

public class PrimeGenerator implements Runnable {

private final List<BigInteger> primes = new ArrayList<>();

private volatile boolean canceled;

@Override

public void run() {

BigInteger p = BigInteger.ONE;

while (!canceled){

p = p.nextProbablePrime();

synchronized (this) { //同步添加素数

primes.add(p);

}

}

}

/\*\*

\* 取消生成素数

\*/

public void cancel(){

canceled = true;

}

/\*\*

\* 同步获取素数

\* @return 已经生成的素数

\*/

public synchronized List<BigInteger> get(){

return new ArrayList<>(primes);

}

}

其测试用例:

public class PrimeGeneratorTest {

public static void main(String[] args) {

PrimeGenerator pg = new PrimeGenerator();

new Thread(pg).start();

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

} finally{

pg.cancel(); //始终取消

}

System.out.println("all primes: " + pg.get());

}

}

### 中断：

* 调用interrupt并不意味者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。会在下一个取消点中断自己，如wait, sleep,join等。
* 通常，中断是实现取消的最合理方式。

下面通过中断实现取消功能：

/\*\*

\* 通过中断来实现取消

\* 不采用boolean变量，

\* 防止在queue.put()时由于阻塞，不能检查到boolean变量而无法取消

\* 但使用interrupt就可以,

\* 即使queue.put()阻塞, 也会检查到interrupt信号，从而抛出IntteruptedException

\* 从而达到取消的目的

\*/

public class PrimeProducer extends Thread {

private final BlockingQueue<BigInteger> queue;

public PrimeProducer(BlockingQueue<BigInteger> queue){

this.queue = queue;

}

@Override

public void run() {

try {

BigInteger p = BigInteger.ONE;

while (!Thread.currentThread().isInterrupted()){

queue.put(p = p.nextProbablePrime());

}

} catch (InterruptedException e) {

// thread exit

}

}

/\*\*

\* 取消

\*/

public void cancel(){

interrupt(); //中断当前线程

}

}

### 中断策略：

* 由于每个线程拥有各自的中断策略，因此除非你知道中断对该线程的含义，否则就不应该中断这个线程。

### 响应中断：

* 处理InterruptedException的实用策略：

      1. 传递异常。

      2. 恢复中断状态，从而事调用栈的上层代码能够对其进行处理。

* 只有实现了线程中断策略的代码才可以屏蔽中断请求，在常规的任务和库代码中都不应该屏蔽中断请求。

### 通过Future实现取消：

public void timedRun(Runnable r, long timeout, TimeUnit unit)

throws InterruptedException {

Future<?> task = taskExec.submit(r);

try {

task.get(timeout, unit);

} catch (ExecutionException e) {

//任务执行中抛出异常

} catch (TimeoutException e) {

//任务超时处理

} finally{

//任务执行完毕,没有影响; 任务执行中会中断任务

if (task != null) task.cancel(true);

}

}

* 当Future.get抛出InterruptedException或TimeoutException时 ，如果你知道不再需要结果，那么就可以调用Future.cancel来取消任务。

### 处理不可中断的阻塞：

* 造成线程阻塞的原因：

       1. java.io包中的同步Socket I/O。如套接字中进行读写操作read, write方法。

       2. java.io包中的同步I/O。如当中断或关闭正在InterruptibleChannel上等待的线程时，会对应抛出ClosedByInterruptException或AsynchronousCloseException。

       3. Selector的异步I/O。如果一个线程在调用Selector.select时阻塞了，那么调用close, wakeup会使线程抛出ClosedSelectorException。

       4. 获取某个锁。当一个线程等待某个锁而阻塞时，不会响应中断。但Lock类的lockInterruptibly允许在等待锁时响应中断。

/\*\*

\* 通过改写interrupt方法将非标准的取消操作封装在Thread中

\*/

public class ReaderThread extends Thread {

private final Socket socket;

private final InputStream in;

private int bufferSize;

public ReaderThread(Socket socket, InputStream in) {

this(socket, in, 1024);

}

public ReaderThread(Socket socket, InputStream in, int bufferSize) {

this.socket = socket;

this.in = in;

this.bufferSize = bufferSize;

}

@Override

public void interrupt() {

try {

socket.close(); //中断前关闭socket

} catch (IOException e) {

} finally{

super.interrupt();

}

}

@Override

public void run() {

try {

byte[] buf = new byte[bufferSize];

while (true) {

int count = in.read(buf);

if (count < 0) {

break;

} else if (count > 0) {

processBuffer(buf, count);

}

}

} catch (IOException e) {

// 线程中断处理

}

}

...

}

### 采用newTaskFor来封装非标准的取消：

/\*\*

\* 可取消的任务接口

\*/

public interface CancellableTask<T> extends Callable<T> {

void cancel();

RunnableFuture<T> newTask();

}

/\*\*

\* 使用了Socket的任务

\* 在取消时需要关闭Socket

\*/

public abstract class SocketUsingTask<T> implements CancellableTask<T> {

private Socket socket;

public void setSocket(Socket socket) {

this.socket = socket;

}

@Override

public T call() throws Exception {

//do working

       ...

}

@Override

public synchronized void cancel() {

try {

if (socket != null){

socket.close();

}

} catch (IOException ignored) {

}

}

@Override

public RunnableFuture<T> newTask() {

return new FutureTask<T>(this){

@Override

public boolean cancel(boolean mayInterruptIfRunning) {

try {

SocketUsingTask.this.cancel();

} catch (Exception ignored) {

}

return super.cancel(mayInterruptIfRunning);

}

};

}

}

/\*\*

\* 通过newTaskFor将非标准的取消操作封装在任务中

\*/

public class CancellingExecutor extends ThreadPoolExecutor {

public CancellingExecutor(int corePoolSize, int maximumPoolSize,

long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {

super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);

}

@Override

protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {

if (callable instanceof CancellableTask){ //若是我们定制的可取消任务

return ((CancellableTask<T>)callable).newTask();

}

return super.newTaskFor(callable);

}

}

## 停止基于线程的服务:

* 正确的封装原则：除非拥有某个线程，否则不能对该线程进行操控。如中断线程，修改线程优先级等。
* 对于持有线程的服务，只要服务的存在时间大于创建线程的方法的存在时间，那么就应该提供生命周期的方法。如ExecutorService提供的shutdown(), shutdownNow()。

/\*\*

\* 不支持关闭的生产者-消费者日志服务

\*/

public class LogWriter {

private final BlockingQueue<String> queue;

private final LoggerThread logger;

public LogWriter(Writer writer){

this.queue = new LinkedBlockingDeque<String>();

this.logger = new LoggerThread(writer);

}

public void start(){

logger.start();

}

public void log(String msg) throws InterruptedException{

queue.put(msg);

}

private class LoggerThread extends Thread{

private final Writer writer;

public LoggerThread(Writer writer) {

this.writer = writer;

}

@Override

public void run() {

try {

while(true){

writer.write(queue.take());

}

} catch (IOException e) {

// io exception handle

} catch (InterruptedException e) {

// interrupt exceptino handle

} finally{

try {

writer.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

}

* 向LogWriter添加可靠的取消操作

/\*\*

\* 为LoggerWriter添加可靠的取消操作

\*/

public class LogService {

private final BlockingQueue<String> queue;

private final LoggerThread logger;

private final PrintWriter writer;

private boolean isShutdown; //用于终止生产者

private int reservations; //队列中的消息数

public LogService(PrintWriter writer){

this.queue = new LinkedBlockingDeque<String>();

this.logger = new LoggerThread();

this.writer = writer;

}

/\*\*

\* 产生日志

\* @param msg 日志内容

\* @throws InterruptedException

\*/

public void log(String msg) throws InterruptedException{

synchronized (this) {

if (isShutdown){

throw new IllegalStateException("can't log, service has stopped.");

}

++reservations;

}

queue.put(msg);

}

/\*\*

\* 启动日志香妃

\*/

public void start(){

logger.start();

}

/\*\*

\* 停止日志服务

\*/

public void stop(){

synchronized(this){

isShutdown = true;

}

logger.interrupt(); //中断日志线程

}

/\*\*

\* 消费日志线程

\*/

private class LoggerThread extends Thread{

@Override

public void run() {

try {

while(true){

try {

synchronized (LogService.this) {

if (isShutdown)

break;

}

String msg = queue.take();

synchronized (LogService.this) {

--reservations;

}

writer.println(msg);

} catch (InterruptedException e) {

// retry

}

}

}finally{

writer.close();

}

}

}

}

### 关闭ExecutorService:

* ExecutorService提供两种关闭服务的方法：

      1. shutdown: 安全关闭。不再接受新任务提交，待所有队列中的任务执行完成再关闭。

      2. shutdownNow: 强行关闭。不再接受新任务提交，停止正在执行的任务，并返回未开始执行的任务列表。

/\*\*

\* 封装ExecutorService实现日志服务

\*/

public class LogService2 {

private final ExecutorService exec = Executors.newSingleThreadExecutor();

private final PrintWriter writer;

public LogService2(PrintWriter writer){

this.writer = writer;

}

/\*\*

\* 产生日志

\* @param msg 日志内容

\* @throws InterruptedException

\*/

public void log(String msg) throws InterruptedException{

exec.execute(new WriteTask(msg));

}

/\*\*

\* 停止日志服务

\* @throws InterruptedException

\*/

public void stop(long timeout, TimeUnit unit) throws InterruptedException{

try {

exec.shutdown(); //平缓关闭服务

//关闭服务后, 阻塞到所有任务被执行完毕或者超时发生，或当前线程被中断

exec.awaitTermination(timeout, unit);

} finally{

writer.close();

}

}

...

}

### "毒丸"对象：

* 毒丸：指一个放在队列上的对象，当得到这个对象时，就立即停止。

通过毒丸对象来关闭服务：

/\*\*

\* 索引服务

\* 通过一个毒丸对象来关闭服务

\*/

public class IndexingService {

private static final File POISON = new File(""); //毒丸对象

private final IndexerThread consumer = new IndexerThread(); //消费者

private final CrawlerThread producer = new CrawlerThread(); //生产者

private final BlockingQueue<File> queue = new LinkedBlockingDeque<File>();

private final File root;

public IndexingService(File root) {

this.root = root;

}

/\*\*

\* 启动索引服务

\*/

public void start(){

producer.start();

consumer.start();

}

public void stop(){

producer.interrupt(); //中断爬虫线程

}

public void awaitTermination() throws InterruptedException{

consumer.join(); //等待消费者线程结束

}

/\*\*

\* 爬虫线程

\*/

private class CrawlerThread extends Thread{

@Override

public void run() {

try {

crawl(root);

} catch (InterruptedException e) {

// handle the exception

}

try {

while(true){

queue.put(POISON);

break;

}

} catch (InterruptedException e) {

// retry

}

}

private void crawl(File root) throws InterruptedException{

// crawl from web

}

}

/\*\*

\* 建立索引的线程

\*/

private class IndexerThread extends Thread{

@Override

public void run() {

try {

while (true){

File file = queue.take();

if (file == POISON){ //若是毒丸对象

break;

} else{

indexFile(file); //建立索引文件

}

}

} catch (InterruptedException e) {

// handle exception

}

}

private void indexFile(File file) {

}

}

}

### shutdownNow的局限性：

* 在关闭服务过程中，我们无法通过常规方法来得知哪些任务已经开始但未结束。

/\*\*

\* 在ExecutorService中跟踪在关闭之后被取消的任务

\*/

public class TrackingExecutor extends AbstractExecutorService {

private final ExecutorService exec;

private final Set<Runnable> tasksCancelledAtShutdown =

Collections.synchronizedSet(new HashSet<Runnable>());

public TrackingExecutor(ExecutorService exec) {

this.exec = exec;

}

/\*\*

\* 获取关闭后取消的任务

\*/

public List<Runnable> getCancelledTasks(){

if (!exec.isTerminated()){

throw new IllegalStateException("service doesn't stop");

}

return new ArrayList<>(tasksCancelledAtShutdown);

}

@Override

public void execute(final Runnable command) {

exec.execute(new Runnable() {

@Override

public void run() {

try {

command.run();

} finally{ //有可能出现误报: 任务执行完毕了, 线程池

if (isShutdown() && //若Executor已经关闭了

Thread.currentThread().isInterrupted()){ //且当前线程被中断了

tasksCancelledAtShutdown.add(command);

}

}

}

});

}

}

### 处理非正常的线程终止：

* 当一个线程由于未捕获异常而退出时, jvm会把这个事件报告给应用程序提供的UncaughtExceptionHandler异常处理器。若没有提供任何异常处理器，则默认行为是将栈追踪信息输出到System.err。

/\*\*

\* 将异常写入日志的UncaughtExceptionHandler

\*/

public class UEHLogger implements UncaughtExceptionHandler {

@Override

public void uncaughtException(Thread t, Throwable e) {

Logger logger = Logger.getAnonymousLogger();

logger.log(Level.SEVERE, "the thread with exceptoin: "+t.getName(), e);

}

}

* 在运行时间较长的应用程序中，通常会为所有线程的未捕获异常制定同一个异常处理器，并且该处理器至少会将异常信息记录到日志中。

## JVM关闭：

### 关闭钩子：

* 关闭钩子：通过Runtime.addShutdownHook注册的但尚未开始的线程。
* jvm不保证关闭钩子的调用顺序。
* 当强制关闭jvm时，不会运行关闭钩子。
* 最后对所有服务使用同一个关闭钩子，防止多个钩子之间的出现共享资源竞争。

### 守护线程：

* 守护线程(Daemon Thread)：执行一些辅助工作，不会阻碍JVM的关闭。
* 线程分为普通线程和守护线程。jvm启动时创建的所有线程中，除了主线程，其他线程都是守护线程(例如垃圾回收器以及其他执行辅助工作的线程)。
* 新创建的线程，默认会继承创建它的线程的守护状态，所以默认时，主线程创建的所有线程都是普通线程。
* 普通线程与守护线程之间的差异：当一个线程退出时，jvm会检查其他正在运行的线程，如果这些线程是守护线程，那么jvm会正常退出操作。当jvm停止时，所有仍然存在的守护线程都会被抛弃－不执行finally块，不执行回卷栈，而直接退出。所有尽可能少用守护线程，特别是包含一些I/O操作的任务。
* 守护线程通常不能用来替代应用程序管理程序中各个服务的生命周期。

终结器：

* 避免使用终结器。

# 七、线程池的使用：

* 线程池分配，调优，使用注意事项。

## 在任务与执行策略之间的隐性耦合：

* 有些任务需要明确指定执行策略：

       1. 依赖性任务。提交的任务需要依赖其他任务，此时需要小心维护这些执行策略以避免产生活跃性问题。

       2. 使用线程封闭机制的任务。任务要求其执行所在的Executor是线程安全的。

       3. 对响应时间敏感的任务。

       4. 使用ThreadLocal的任务。ThreadLocal使每个线程都可以拥有某个变量的一个私有"版本"。

* 只有当任务都是同类型的且相互独立时，线程池的性能才能达到最佳。

### 线程饥饿死锁：

* 在线程池中，如果所有正在执行任务的线程都由于等待其他仍处于工作队列中的任务而阻塞，这种现象称为线程饥饿死锁。

/\*\*

\* 在单线程Executor中任务发生死锁

\*/

public class ThreadDeadLock {

ExecutorService exec = Executors.newSingleThreadExecutor();

public class RenderPageTask implements Callable<String>{

@Override

public String call() throws Exception {

Future<String> header, footer; //页眉, 页脚

header = exec.submit(new LoadFileTask("header.html"));

footer = exec.submit(new LoadFileTask("footer.html"));

String body = renderBody();

//有可能发生死锁---任务等待子任务完成

return header.get() + body + footer.get();

}

...

}

}

* 每当提交了一个有依赖性的Executor任务时，要清楚地知道可能会出现线程"饥饿"死锁，因此需要在代码或配置Executor地配置文件中记录线程池地大小限制或配置限制。

### 运行时间较长的任务：

* 避免等待运行时间较长的任务而阻塞过长时间，可以使用阻塞方法的超时版本，如Thread.join, BlockingQueue.put,CutDownLatch.await, Selector.select等。

## 设置线程池的大小：

* 线程池的理想大小取决于被提交任务的类型及所部署系统的特性。
* 可根据计算任务类型进行线程池大小：如CPU密集型则可采用Runtime.avaliableProcesses()+1个线程；对于I/O密集型，由于阻塞操作多，可使用更多的线程，如2倍cpu核数。

## 配置ThreadPoolExecutor:

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler) {

### 线程的创建与销毁：

* CorePoolSize: 线程池基本大小，即线程池的目标大小，即在没有任务执行时线程池的大小，并且只有在工作队列满了的情况下才会创建超出这个数量的线程。
* MaxmumPooSize: 线程池最大大小表示可同时活动的线程数量的上限。若某个线程的空闲时间超过了keepAliveTime, 则被标记为可回收的，并且当前线程池的大小大于基本大小，这个线程将被终止。
* newFixedThreadPool: CorePoolSize = MaxmumPoolSize。
* newCachedThreadPool: CorePoolSize=0, MaxmumPoolSize=Integer.MAX\_VALUE，比较适合执行短时间任务。
* newSingleThreadPool: CorePoolSize=MaxmumPoolSize=1，其不可被重配置。
* newScheduledThreadPool: 只能设置CorePoolSize。内部实现不同于其他的ThreadPoolExecutor, 而是SchedduleThreadPoolExecutor。可执行定时任务或者隔时任务。

### 管理队列任务：

* 对于Executor, newCachedThreadPool工厂方法时一种很好的默认选择，它能提供比固定大小的线程池更好的排队性能。当需要限制当前任务的数量以满足资源管理需求时，那么可以选择固定大小的线程池，就像接受网络客户请求的服务应用程序中，如果不进行限制，那么很容易发生过载问题。
* 只有当任务相互独立时，为线程池或工作队列设置界限才是合理的。如果任务之间存在依赖性，那么有界的线程池或队列就可能导致线程"饥饿"死锁问题。此时应该使用无界的线程池，如newCachedThreadPool。

### 饱和策略：

* 当有界队列被填满后，饱和策略开始发挥作用。
* jdk提供的几种饱和策略：

      1. AbortPolicy(中止策略)，默认的饱和策略。会抛出RejectedExecutionException异常。

      2. DiscardPolicy(抛弃策略): 会抛弃该任务。

      3. DiscardOldestPolicy：会抛弃下一个将被执行的任务，然后尝试重新提交新的任务。最好不和优先级队列一起使用，因为它会抛弃优先级最高的任务。

      4. CallerRunsPolicy(调用者运行策略)：将任务回退给调用者。它不会在线程池的某个线程中执行新提交的任务，而是在一个调用execute的线程中执行该任务。

/\*\*

\* 创建一个固定大小的线程池,

\* 并采用有界队列与"调用者运行"饱和策略

\*/

public void intThreadPool() {

ThreadPoolExecutor executor =

new ThreadPoolExecutor(N\_THREADS, N\_THREADS,

0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>(CAPACITY));

executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());

}

/\*\*

\* 使用Semaphore来控制任务的提交速率

\*/

public class BoundedExecutor {

private final Executor exec;

private final Semaphore semaphore;

public BoundedExecutor(Executor exec, int bound) {

this.exec = exec;

this.semaphore = new Semaphore(bound);

}

public void submitTask(final Runnable command){

try {

semaphore.acquire(); //提交任务前请求信号量

exec.execute(new Runnable() {

@Override

public void run() {

try{

command.run();

} finally{

semaphore.release(); //执行完释放信号

}

}

});

} catch (InterruptedException e) {

// handle exception

}

}

}

### 线程工厂：

* 我们可以通过定制线程工厂，从而定制线程池中创建的线程，这样可以实现些扩展功能，如调试信息，设置UncaughtExceptionHandler等。

/\*\*

\* 自定义的线程工厂

\*/

public class MyThreadFactory implements ThreadFactory {

private final String poolName;

public MyThreadFactory(String poolName) {

super();

this.poolName = poolName;

}

@Override

public Thread newThread(Runnable r) {

return new MyAppThread(r);

}

}

public class MyAppThread extends Thread {

public static final String DEFAULT\_NAME="MyAppThread";

private static volatile boolean debugLifecycle = false;

private static final AtomicInteger created = new AtomicInteger();

private static final AtomicInteger alive = new AtomicInteger();

private static final Logger log = Logger.getAnonymousLogger();

public MyAppThread(Runnable r) {

this(r, DEFAULT\_NAME);

}

public MyAppThread(Runnable r, String name) {

super(r, name+ "-" + created.incrementAndGet());

setUncaughtExceptionHandler( //设置未捕获的异常发生时的处理器

new Thread.UncaughtExceptionHandler() {

@Override

public void uncaughtException(Thread t, Throwable e) {

log.log(Level.SEVERE, "UNCAUGHT in thread " + t.getName(), e);

}

});

}

@Override

public void run() {

boolean debug = debugLifecycle;

if (debug)

log.log(Level.FINE, "running thread " + getName());

try {

alive.incrementAndGet();

super.run();

} finally {

alive.decrementAndGet();

if (debug)

log.log(Level.FINE, "existing thread " + getName());

}

}

}

* 当应用需要利用安全策略来控制某些特殊代码库的访问权，可以利用PrivilegedThreadFactory来定制自己的线程工厂，以免出现安全性异常。

### 在调用构造函数后再定制ThreadPoolExecutor:

* 可以在创建线程池后，再通过Setter方法设置其基本属性。

## 扩展ThreadPoolExecutor:

/\*\*

\* 增加日志和记时等功能的线程池

\*/

public class TimingThreadPoolExecutor extends ThreadPoolExecutor {

private final ThreadLocal<Long> startTime = new ThreadLocal<Long>();//任务执行开始时间

private final Logger log = Logger.getAnonymousLogger();

private final AtomicLong numTasks = new AtomicLong(); //统计任务数

private final AtomicLong totalTime = new AtomicLong(); //线程池运行总时间

public TimingThreadPoolExecutor(int corePoolSize, int maximumPoolSize,

long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {

super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);

}

@Override

protected void beforeExecute(Thread t, Runnable r) {

super.beforeExecute(t, r);

log.fine(String.format("Thread %s: start %s", t, r));

startTime.set(System.nanoTime());

}

@Override

protected void afterExecute(Runnable r, Throwable t) {

try{

long endTime = System.nanoTime();

long taskTime = endTime - startTime.get();

numTasks.incrementAndGet();

totalTime.addAndGet(taskTime);

log.fine(String.format("Thread %s: end %s, time=%dns", t, r, taskTime));

} finally{

super.afterExecute(r, t);

}

}

@Override

protected void terminated() {

try{

//任务执行平均时间

log.info(String.format("Terminated: average time=%dns", totalTime.get() / numTasks.get()));

}finally{

super.terminated();

}

}

}

# 八、避免活跃性危险：

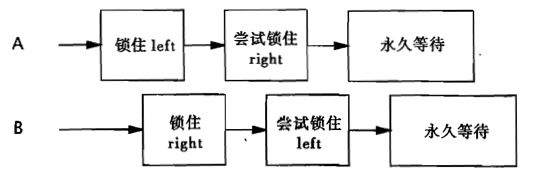
* 本部分讨论活跃性故障的原因，及如何避免它们。

## 死锁：

* 典型的哲学家进餐问题。

### 锁顺序死锁：

如上面哲学家进餐有可能发生下面的情况：



* 上面发生死锁的根本原因在于两个线程以不同的顺序来获取相同的锁。
* 如果所有线程都以固定的顺序获取锁，那么程序就不会出现锁顺序死锁问题。

/\*\*

\* 容易因为获取锁的顺序导致死锁

\*/

public class LeftRightDeadLock {

private final Object left = new Object();

private final Object right = new Object();

public void leftRight(){

synchronized(left){

synchronized(right){

// to do sth.

}

}

}

public void rightLeft(){

synchronized(right){

synchronized(left){

// to do sth.

}

}

}

}

### 动态的锁顺序死锁：

* 典型的就是银行转账问题

public void transferMoney(Account fromAccount, Account toAccount, int money){

synchronized (fromAccount) {

synchronized(toAccount){

if (fromAccount.getBalance() > money){

//余额不足

} else{

fromAccount.debit(money);

toAccount.credit(money);

}

}

}

}

当我们以下面这种方式调用，就有可能出现死锁：

transferMoney(a1, a2, money);

transferMoney(a2, a1, money);

要解决这种问题，就得使内部以相同的顺序加锁，无论外部怎么调用。

/\*\*

\* 用于当输入参数的hash值一样时使用

\*/

private static final Object tieLock = new Object();

public static void transferMoney(final Account fromAccount,

final Account toAccount, final int money){

class Helper {

public void transfer(){

if (fromAccount.getBalance() < money){

//余额不足

} else{

fromAccount.debit(money);

toAccount.credit(money);

}

}

}

int fromHash = System.identityHashCode(fromAccount);

int toHash = System.identityHashCode(toAccount);

//无论客户端怎么传入参数，我们都以先锁定hash值小的，再锁定hash大的

//也可以利用业务中排序关系，如Account的编号等来比较

if (fromHash < toHash){

synchronized (fromAccount){

synchronized (toAccount) {

new Helper().transfer();

}

}

} else if (fromHash > toHash){

synchronized (toAccount){

synchronized (fromAccount) {

new Helper().transfer();

}

}

} else { //hash值相等, 情况很小

synchronized (tieLock) {

synchronized (fromAccount) {

synchronized (toAccount) {

new Helper().transfer();

}

}

}

}

}

在协作对象之间发生的死锁：

* 即在不同方法中相互持有等待得锁。

class Taxi {

private Point location;

private Point destination;

private final Dispatcher dispatcher;

public Taxi(Dispatcher dispatcher) {

this.dispatcher = dispatcher;

}

public synchronized Point getLocation(){

return location;

}

public synchronized void setLocation(Point location){

this.location = location;

if (location.equals(destination)){

dispatcher.notifyAvaliable(this);

}

}

}

class Dispatcher {

private final Set<Taxi> taxis;

private final Set<Taxi> avaliableTaxis;

public Dispatcher(){

taxis = new HashSet<>();

avaliableTaxis = new HashSet<>();

}

public synchronized void notifyAvaliable(Taxi taxi) {

avaliableTaxis.add(taxi);

}

public synchronized Image getImage(){

Image image = new Image();

for (Taxi t :taxis){

image.drawMarker(t.getLocation());

}

return image;

}

}

上面的 setLocation和 getImage就有可能发生死锁现象：setLocation获取到Taxi对象锁后，在dispacher.notifiyAvaliable()时需要dispatcher锁，而getImage获取到dispacher锁后，t.getLocation要求Taxi锁。

* 如果在持有锁时调用某个外部方法，那么将出现活跃性问题，在这个外部方法中可能会获取其他锁(这可能会产生死锁)，或者阻塞时间过长，导致其他线程无法及时获得当前被持有的锁。

### 开放调用：

* 如果在调用某个方法时不需要持有锁，那么这种调用就被称为开放调用(Open Call)。

/\*\*

\* 通过公开调用来避免在相互协作的对象之间产生死锁

\*/

public class OpenCall {

class Taxi {

private Point location;

private Point destination;

private final Dispatcher dispatcher;

public Taxi(Dispatcher dispatcher) {

this.dispatcher = dispatcher;

}

public synchronized Point getLocation(){

return location;

}

public void setLocation(Point location){

boolean reachedDestination;

synchronized(this){

this.location = location;

reachedDestination = location.equals(destination);

}

if (reachedDestination){

dispatcher.notifyAvaliable(this); //这里持有dispatcher锁，但已释放taxi锁

}

}

}

class Dispatcher {

private final Set<Taxi> taxis;

private final Set<Taxi> avaliableTaxis;

public Dispatcher(){

taxis = new HashSet<>();

avaliableTaxis = new HashSet<>();

}

public synchronized void notifyAvaliable(Taxi taxi) {

avaliableTaxis.add(taxi);

}

public Image getImage(){

Set<Taxi> copy;

synchronized (this) {

copy = new HashSet<Taxi>(taxis);

}

Image image = new Image();

for (Taxi t :copy){

image.drawMarker(t.getLocation());//调用外部方法前已释放锁

}

return image;

}

}

}

* 在程序中尽量使用开放调用。与那些在持有锁调用外部方法的程序时，更易于对依赖于开放调用的程序进行死锁分析。

### 资源死锁：

* 多个资源池(如数据库连接池)，一个线程需要连接2个数据库连接，如线程A持有Pool1的连接，等待Pool2的连接；线程B持有Pool2的连接，等待Pool1的连接。
* 线程饥饿死锁，如一个任务中提交另一个任务，并一直等待被提交任务完成。
* 有界线程池/资源池与相互依赖的任务不能一起使用。

## 死锁的避免与诊断：

### 支持定时的锁：

* 限时等待。如Lock中的tryLock, 给定一个超时时限，若等待超过该时间，则会给出错误信息，避免永久等待。

### 通过线程转储信息来分析死锁：

* 可先通过jstack <pid>获取线程栈信息。
* 再通过分析工具Thread Dump Analyzer: <https://java.net/projects/tda/downloads> 进行死锁分析。

## 其他活跃性危险：

### 饥饿：

* 由于线程无法访问它所需的资源而不能继续执行时，就发生了"饥饿"。
* 要避免使用线程优先级，这会增加平台依赖性，并可能导致活跃性问题，在大多数并发应用程序中，都可以使用默认的线程优先级。

### 糟糕的响应性：

* 例如GUI程序中使用了后台线程。该后台任务若为CPU密集型，将可能影响程序响应性。
* 不良的锁管理也可能导致糟糕的响应性。

### 活锁：

* 活锁是另一种形式的活跃性问题，该问题尽管不会阻塞线程，但也不能继续执行，因为线程将不断重复执行相同的操作，而且总会失败。
* 当多个相互协作的线程都对彼此进行响应从而修改各自的状态，并使得任何一个线程都无法继续执行时，就发生了活锁。
* 在并发应用中，通过等待随机长度的时间和回退可以有效的避免活锁的发生。

# 九、性能与可伸缩性：

## 对性能的思考：

* 提升性能意味着用更少的资源做更多的事情。
* 资源：CPU, 内存，I/O带宽，网络带宽，数据库请求，磁盘空间等。

### 性能与可伸缩性：

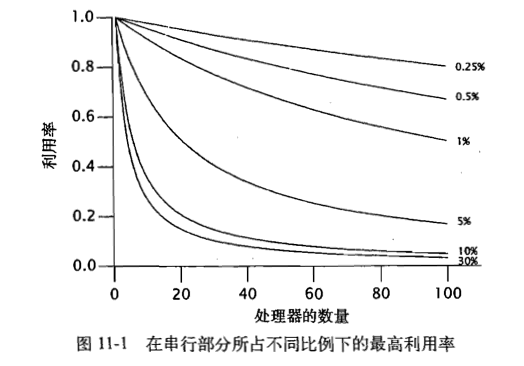
* 应用程序性能的衡量指标：服务时间，延迟时间，吞吐率，效率，可伸缩性及容量等。
* 可伸缩性指：增加资源时，程序的吞吐量或者处理能力相应地增加。

### 评估各种性能权衡因素：

* 避免不成熟地优化，首先使程序正确，然后再提高运行速度--如果它还运行得不够快。
* 以测试为基准，不要猜测。

## Amdahl定律：

* 在增加计算资源的情况下，程序在理论上能够实现最高加速比，这个值取决于程序中可并行组件与串行组件的比重。
* 可参考：<http://zh.wikipedia.org/wiki/Amdahl%E5%AE%9A%E7%90%86>。
* 见图：



* 对任务队列的串行访问

/\*\*

\* 对任务队列的串行访问

\*/

public class WorkThread extends Thread {

private final BlockingQueue<Runnable> queue;

public WorkThread(BlockingQueue<Runnable> queue){

this.queue = queue;

}

public void run(){

while (true){

try {

Runnable task = queue.take(); //此处为程序的串行部分

task.run();

} catch (InterruptedException e) {

break;

}

}

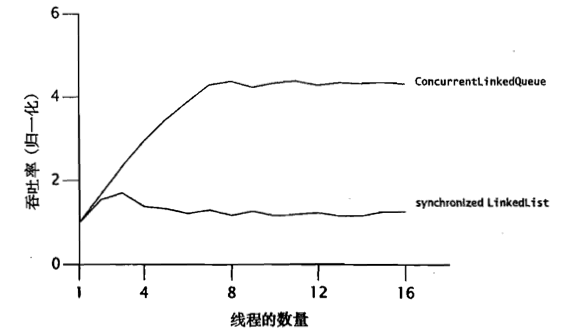
}

}

* 在所有并发程序中都包含一些串行部分，如果你认为你的程序中不存在串行部分，那么可以再仔细检查一遍。

### 在各种框架中隐藏的串行部分：

* 不同同步队列吞吐量差异



### Amdahl定律的应用：

* 降低锁粒度的技术：锁分解(1个锁分解为2个锁)，锁分段(1个锁分解为多个锁)。

## 线程引入的开销：

* 对于为了提升性能而引入的线程来说，并行带来的性能提升必须超过并发导致的开销。

### 上下文切换：

* 切换上下文需要一定的开销，而在线程调度过程中需要访问操作系统和JVM共享的数据结构。
* 上下文切换带来的开销因平台而异，一般情况就是几微秒。

### 内存同步：

* 同步操作的性能开销包括多个方面，在synchronized和volatile提供的可见性保证中可能会使用一些特殊命令，即内存栅栏。
* 在内存栅栏中，大多数操作都是不能被重排序的。
* JVM可以通过优化来去掉一些不会发生竞争的锁，如：

//object只能由当前线程所访问，所以会去掉锁

synchronized(new Object()){

// do sth.

}

//局部变量v不会逃逸, 因此线程私有，优化会取消加锁

public String getStoogeNames(){

Vector<String> v = new Vector<>();

v.add("Hello");

v.add("World");

return v.toString();

}

### 阻塞：

* 阻塞的线程将包含两次额外的上下文切换：

       1. 阻塞时，cpu时间片未用完前被交换出去。

       2. 请求的锁或资源可用时，再次被切换回来。

## 减少锁的竞争：

* 在并发程序中，对可伸缩性的最主要威胁就是独占方式的资源锁。
* 有3中方式可以降低锁的竞争程度：

       1. 减少锁的持有时间。

       2. 降低锁的请求频率。

       3. 使用带有协调机制的独占锁。

### 减少锁的范围(“快进快出”)：

* 较少锁的持有时间，如将锁无关的代码移出同步块。

/\*\*

\* 不必要的长时间持有锁

\*/

public class AttrbuteStore {

private final Map<String, String> attributes

= new HashMap<String, String>();

/\*\*

\* synchronized锁住当前对象

\*/

public synchronized boolean userLocationMatcher(String name, String regexp){

String key = "users." + name + ".location";

String location = attributes.get(key);

if (location == null)

return false;

else

return Pattern.matches(regexp, location);

}

}

可修改上面的方法：

public boolean userLocationMatcher(String name, String regexp){

String key = "users." + name + ".location";

String location = null;

synchronized(this){

location = attributes.get(key); //仅锁住共享对象

}

if (location == null)

return false;

else

return Pattern.matches(regexp, location);

}

更好的方式是将attributes用并发容器来实现，如ConcurrentHashMap等。

### 减小锁的粒度：

* 可通过锁分解和锁分段技术来实现。
* 锁分解实例：

/\*\*

\* 多个状态由一个锁来保护

\*/

public class ServerStatus {

public final Set<String> users;

private final Set<String> queries;

...

public synchronized void addUser(String u){

users.add(u);

}

public synchronized void addQuery(String q){

queries.add(q);

}

}

将锁进行分解：

/\*\*

\* 多个状态由多个锁来保护

\*/

public class BetterServerStatus {

public final Set<String> users;

private final Set<String> queries;

...

public void addUser(String u){

synchronized(users){

users.add(u);

}

}

public void addQuery(String q){

synchronized(queries){

queries.add(q);

}

}

}

### 锁分段：

* 将所分解技术进一步扩展为对一组独立对象上的锁进行分解，这种情况被称为锁分段。
* 实例如ConrrentHashMap的实现：[http://my.oschina.net/indestiny/blog/209458](http://my.oschina.net/indestiny/blog/209458" \t "_blank)

### 避免热点域：

* 热点域：比如ConcurrentHashMap.size()求元素个数时，是通过枚举每个segment.size累加的，如果你说想单独用一个size来保存元素个数，这样size(), isEmpty()这些方法就很简单了，但同样来一个问题，size的修改会很频繁，切须进行锁保护，反而又降低性能，这时的size 就是一个热点域。

### 一些替代独占锁的方法：

* 第三种降低竞争锁的影响就是放弃使用独占锁。如并发容器，读写锁，不可变对象，原子变量。

# 十一、显示锁：

## Lock与ReentrantLock:

* Lock接口

void lock(); //获取锁

void lockInterruptibly() throws InterruptedException; //获取锁，且当前线程可被中断

boolean tryLock(); //尝试获取锁，true获取到锁, false未获取到锁

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

void unlock(); //释放锁

Condition newCondition(); //在当前锁上创建一个等待条件

* Lock的标准用法

Lock lock = new ReentrantLock();

lock.lock();

try{

// to do sth.

} finally{

lock.unlock(); //须要在finally中释放锁

}

### 轮询锁与定时锁：

* 轮询锁和定时锁可由tryLock来实现。
* 轮询锁,定时锁可以避免死锁的发生。
* 由tryLock实现轮询锁

public boolean transferMoney(Account fromAcct,

Account toAcct,

int amount,

long timeout,

TimeUnit unit){

long fixedDelay = getFixedDelayComponentNanos(timeout, unit);

long randMod = getRandomDelayModulusNanos(timeout, unit);

long stopTime = System.nanoTime() + unit.toNanos(timeout);

while (true){

if (fromAcct.lock.tryLock()){ //若获取到源账户锁

try{

if (toAcct.lock.tryLock()){ //若获取到目的账户锁

try{

if (fromAcct.getBalance() < amount){

throw new RuntimeException("money.not.enough");

} else{

fromAcct.debit(amount);

toAcct.credit(amount);

return true;

}

} finally{

toAcct.lock.unlock();

}

}

} finally{

fromAcct.lock.unlock();

}

}

if (System.nanoTime() < stopTime){

return false;

}

try {

Thread.sleep(fixedDelay + rand.nextLong()%randMod);

} catch (InterruptedException e) {

}

}

}

你也可以采用定时实现：

lock.tryLock(timeout, unit);

#### 可中断的锁获取操作：

lock.lockInterruptibly();

try{

// maybe throws InterruptedException

doSomething();

} finally{

lock.unlock();

}

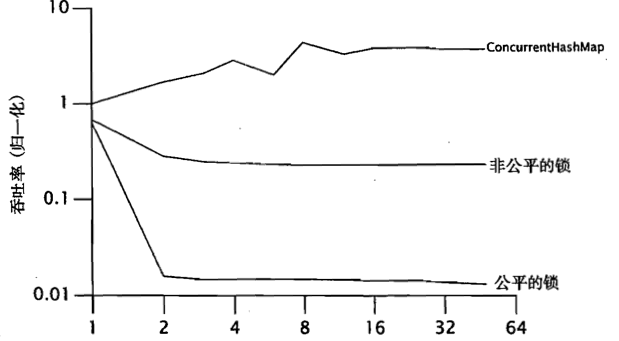
### 非块结构的加锁：

* 如ConcurrentHashMap中的分段锁实现。

HashTable容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

## 公平性：

* ReentrantLock可初始化为公平或非公平的锁。
* 大多数情况下非公平锁的性能高于公平锁的性能。
* 基于公平锁，非公平锁及ConcurrentHashMap对HashMap进行吞吐量测试。



* 当持有锁的时间相对较长，或者请求锁的平均时间间隔较长，那么应该使用公平锁。

## 在Synchronized和ReentrantLock之间作出选择：

* 在一些内置锁无法满足需求的情况下，ReentrantLock可以作为一种高级工具。当需要一些高级功能时才应该使用ReentrantLock,这些功能包括：可定时，可轮询，可中断，公平队列，及非块结构的锁。否则还是应该优先使用synchronized.

## 读--写锁：

//读写锁允许同时多个线程读, 或最多一个线程写

public interface ReadWriteLock {

Lock readLock();

Lock writeLock();

}

* 读写锁的可选实现：

       1. 释放优先。当写入锁释放后，应该优先选择读线程，写线程，还是最先发出请求的线程?

       2. 读线程插队。锁由读线程持有，写线程再等待，再来一个读线程，是继续让读线程访问，还是让写线程访问.

       3. 重入性。读取锁和写入锁是否可重入?

       4. 降级。将写入锁降级为读取锁。

       5. 升级。将读取锁升级为写入锁。

* 当锁的持有时间较长并且大部分操作都不会修改被守护的资源时，可用读写锁提高并发性。

/\*\*

\* 读写锁来包装Map

\*/

public class ReadWriteMap<K, V> {

private final Map<K, V> map;

private final ReadWriteLock lock = new ReentrantReadWriteLock();

private final Lock r = lock.readLock();

private final Lock w = lock.writeLock();

public ReadWriteMap(Map<K, V> map){

this.map = map;

}

// 其他写操作...

public V put(K key, V value){

w.lock(); //请求写锁

try{

return map.put(key, value);

} finally{

w.unlock(); //勿忘

}

}

// 其他读操作...

public V get(K key){

r.lock(); //请求读锁

try{

return map.get(key);

} finally{

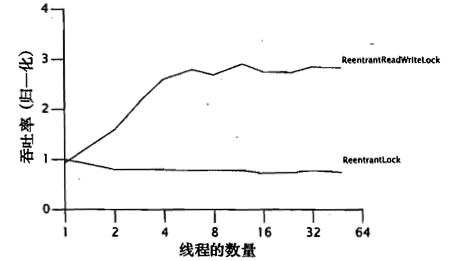
r.unlock(); //勿忘

}

}

}

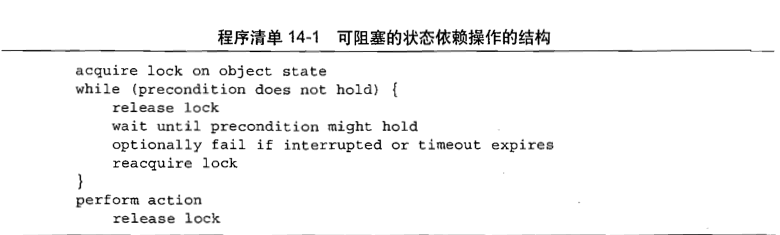
* 对ArrayList使用ReentrantLock和ReadWriteLock测试吞吐量。



# 十二、构建自定义的同步工具：

## 状态依赖性的管理：

* 依赖状态的操作可以一直阻塞直到可以继续执行，这比使他们先失败再实现起来更为方便且更不易出错。



* 有界缓存的几种实现

/\*\*

\* 有界缓存实现的基类

\*/

public abstract class BaseBoundedBuffer<V> {

private final V[] buf;

private int tail;

private int head;

private int count;

protected BaseBoundedBuffer(int capacity){

this.buf = (V[]) new Object[capacity];

}

protected synchronized final void doPut(V v){

buf[tail] = v;

if (++tail == buf.length){

tail = 0;

}

++count;

}

protected synchronized final V doTake(){

V v = buf[head];

buf[head] = null; //let gc collect

if (++head == buf.length){

head = 0;

}

--count;

return v;

}

public synchronized final boolean isFull(){

return count == buf.length;

}

public synchronized final boolean isEmpty(){

return count == 0;

}

}

* 将前提条件的失败传给调用者

/\*\*

\* 当不满足前提条件时，有界缓存不会执行相应的操作

\*/

public class GrumyBoundedBuffer<V> extends BaseBoundedBuffer<V> {

public GrumyBoundedBuffer(int size){

super(size);

}

public synchronized void put(V v){

if (isFull()){

throw new BufferFullException();

}

doPut(v);

}

public synchronized V take(){

if (isEmpty())

throw new BufferEmptyExeption();

return doTake();

}

}

* 对上面有界缓存类的调用

while (true){

try {

String item = buffer.take();

} catch (BufferEmptyExeption e) { //调用者来处理异常

e.printStackTrace();

}

}

这样将状态依赖性交给调用着来管理，可能导致一些功能不能实现，如维持FIFO顺序。

### 通过轮询与休眠来实现简单的阻塞：

/\*\*

\* 使用简单阻塞实现的有界缓存

\*/

public class SleepyBounedBuffer<V> extends BaseBoundedBuffer<V> {

private static long SLEEP\_TIME;

public SleepyBounedBuffer(int size) {

super(size);

}

public void put(V v) throws InterruptedException{

while (true){

synchronized(this){

if (!isFull()){

doPut(v);

return;

}

}

Thread.sleep(SLEEP\_TIME);

}

}

public V take() throws InterruptedException{

while (true){

synchronized(this){

if (!isEmpty()){

return doTake();

}

}

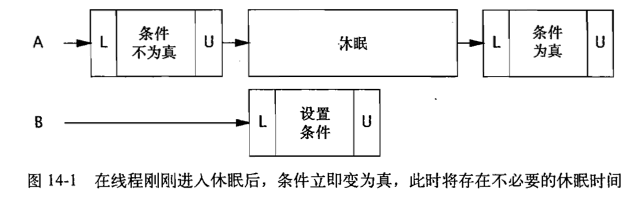
Thread.sleep(SLEEP\_TIME);

}

}

}

上面的休眠时间的设置，需要在 响应性与 CPU使用率之间的权衡。



### 条件队列：

* 条件队列使得一组线程(等待线程集合)以某种特定的方式来等待待定条件为真。

/\*\*

\* 使用条件队列实现的有界缓存

\* 每个对象都可看作一个条件队列

\*/

public class BoundedBuffer<V> extends BaseBoundedBuffer<V> {

public BoundedBuffer(int capacity) {

super(capacity);

}

public synchronized void put(V v) throws InterruptedException{

while (isFull()){

wait();

}

doPut(v);

notifyAll();

}

public synchronized V take() throws InterruptedException{

while (isEmpty()){

wait();

}

V v = doTake();

notifyAll();

return v;

}

}

## 使用条件队列：

* 条件队列使构建高效以及高可响应性的状态依赖变得更容易，但同时也很容易被不正确地使用。

### 条件谓词：

* 要想正确使用条件队列，关键是找出对象在哪个谓词条件上等待。
* 条件谓词是使某个操作成为状态依赖操作的前提条件。在有界缓存中，比如take()方法的条件谓词就是"缓存不为空"。
* 条件谓词是由类中各个状态变量构成的表达式。比如判断"缓存不为空时"，是以状态count==0来进行判断的。
* 将与条件队列相关联的条件谓词以及在这些条件谓词上等待的操作都写入文档。
* 在条件等待中存在一种重要的三元关系：加锁, wait, 条件谓词(正如上面的代码)。
* 每一次wai调用都隐式地与特定的条件谓词相关联，当调用某个条件谓词的wait时，调用者必须已经持有与条件队列相关的锁，并且这个锁必须保护着构成条件谓词的状态变量。

### 过早唤醒：

当使用条件等待时(如Object.wait(), 或Condition.await())：

* 通常都有一个条件谓词--包括一些对象状态的测试，线程在执行前必须首先通过这些测试。
* 在调用wait之前测试条件谓词，并且从wait中返回时再次进行测试。
* 在一个循环中调用wait。
* 确保使用与条件队列相关的锁来保护构成条件谓词的各个状态变量。
* 当调用wait, notify或notifyAll等方法时，一定要持有与条件队列相关的锁。
* 在检查条件谓词之后以及开始执行相应的操作之前，不要释放锁。

### 丢失的信号：

* 丢失的信号：线程必须等待一个已经为真的条件。(即错过被唤醒的机会)。

### 通知：

* 每当在等待一个条件时，一定要确保在条件谓词变为真时通过某种方式发出通知。
* 上面的代码我们使用notifyAll进行唤醒，而不是用notify，是为了防止某种意义上的"信号丢失"，比如notify总是去唤醒一个等待条件为假的线程，而阻止了等待条件为真的线程被执行。
* 在以下情况时，则可以用notify而不是notifyAll:

       1. 所有等待线程的类型都相同，只有一个条件谓词与条件队列相关，并且每个线程从wait返回后执行相同的操作。

       2. 单进单出。在条件变量上的每次通知，最多只能唤醒一个线程来执行。

### 阀门类：

/\*\*

\* 使用wait和notifyAll来实现可重新关闭的阀门

\*/

public class ThreadGate {

private boolean isOpen;

private int generation;

public synchronized void close(){

isOpen = false;

}

public synchronized void open(){

++ generation;

* isOpen = true;

notifyAll();

}

public synchronized void await() throws InterruptedException{

int arrivalGeneration = generation;

while (!isOpen && arrivalGeneration == generation){

wait();

}

}

}

### 子类的安全问题：

* 如果在实施子类化时违背了条件通知或单次通知的某个需求，那么在子类中可以增加合适的通知机制来代表基类。

### 封装条件队列：

* 将条件队列对象封装起来，与线程安全类的最常见设计模式并不一致，在这种模式中建议使用对象自身既是锁，又是条件队列。

### 入口协议与出口协议：

* 入口出口协议用来描述wait和notify方法的正确使用。
* 入口协议就是该操作的条件谓词，出口协议则包括，检查被该操作修改的所有状态变量，并确认它们是否使某个其他的条件谓词变为真，如果是则通知相关的条件队列。

## 显示的Condition对象：

public interface Condition {

void await() throws InterruptedException;

void awaitUninterruptibly();

long awaitNanos(long nanosTimeout) throws InterruptedException;

long awaitNanos(long nanosTimeout) throws InterruptedException;

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

boolean awaitUntil(Date deadline) throws InterruptedException;

void signal();

}

* 基于Condition实现的有界缓冲队列

/\*\*

\* 使用显示条件变量的有界缓存

\*/

public class ConditionBoundedBuffer<T> {

private static final int BUFFER\_SIZE = 10;

protected final Lock lock = new ReentrantLock();

private final Condition notFull = lock.newCondition();

private final Condition notEmpty = lock.newCondition();

private final T[] items = (T[])new Object[BUFFER\_SIZE];

private int tail, head, count;

public void put(T x) throws InterruptedException{

lock.lock();

try {

while (count == items.length){

notFull.await(); //队列已满等待。

}

items[tail] = x;

if (++ tail == items.length){

tail = 0;

}

++ count;

notEmpty.signal(); //队列非空，唤醒其他等待条件

} finally{

lock.unlock();

}

}

public T take() throws InterruptedException{

lock.lock();

try {

while (count == 0){

notEmpty.await();

}

T x = items[head];

items[head] = null; // let gc collect

--count;

notFull.signal();

return x;

} finally{

lock.unlock();

}

}

}

## Synchronizer剖析：

/\*\*

\* 使用Lock来实现信号量

\* 不是java.util.concurrent.Semaphore的真实实现

\*/

public class SemaphoreOnLock {

private final Lock lock = new ReentrantLock();

private final Condition permitsAvailable = lock.newCondition(); //等待可用条件

private int permits; //允许请求的线程数

public SemaphoreOnLock(int initialPermits) {

lock.lock();

try {

permits = initialPermits;

} finally{

lock.unlock();

}

}

public void acruire() throws InterruptedException{

lock.lock();

try {

while (permits <= 0){

permitsAvailable.await();

}

-- permits;

} finally {

lock.unlock();

}

}

public void release(){

lock.lock();

try {

++permits;

permitsAvailable.signal(); //已可用信号通知

} finally{

lock.unlock();

}

}

}

* ReentrantLock与Semaphore都使用了同一个基类AbstractQueueSynchronizer(AQS)。

## AbstractQueuedSynchronizer类：

### 一个简单的闭锁：

/\*\*

\* 使用AbstractQueuedSynchronizer实现的二元闭锁

\*/

public class OneShotLatch {

private Sync sync = new Sync();

public void signal(){

sync.releaseShared(0);

}

public void await() throws InterruptedException{

sync.acquireSharedInterruptibly(0);

}

private class Sync extends AbstractQueuedSynchronizer{

private static final long serialVersionUID = -4086289179080702705L;

protected int tryAcquireShared(int ingored){

// 如果闭锁是开的(state == 1), 那么这个操作符成功，否则将失败

return (getState() == 1) ? 1 : -1;

}

protected boolean tryReleaseShared(int ignored){

setState(1); // 打开闭锁

return true; // 其他的线程可以获取该闭锁

}

}

}

## java.util.concurrent同步器类中的CAS:

protected boolean tryAcquire(int ignored){

final Thread current = Thread.currentThread();

int c = getState();

if (c == 0){ //若为被锁住

if (compareAndSetState(0, 1)){ //设置锁住

owner = current;

return true;

}

} else if(current == owner){ //锁重入

setState(c + 1);

return true;

}

return false;

}

### Semaphore与CountDownLatch:

//Semaphore的tryAcquireShared, truReleaseShared

protected int tryAcquireShared(int acquires) {

for (;;) {

int available = getState();

int remaining = available - acquires;

if (remaining < 0 ||

compareAndSetState(available, remaining))

return remaining;

}

}

protected boolean tryReleaseShared(int releases) {

for (;;) {

int current = getState();

int next = current + releases;

if (next < current) // overflow

throw new Error("Maximum permit count exceeded");

if (compareAndSetState(current, next))

return true;

}

}

### FutureTask:

* 在FutureTask中，AQS同步状态被用来保存任务的状态。

### ReentrantReadWriteLock:

# 十三、原子变量与非阻塞同步机制：

## 锁的优势：

* 当线程在锁上发生竞争时，智能的JVM不一定会挂起线程，而是根据之前获取操作中对锁持有时间来判断此线程是挂起还是自旋。

## 硬件对并发的影响：

* 独占锁是一项悲观技术。
* 现在处理器基本都支持一些读-写-改的指令，如比较并交换(CAS)或关联加载/条件存储(Load-Linked/Store-Conditional)。操作系统和JVM使用这些指令来实现锁和并发数据结构。jdk5之前不能直接使用这些指令。

### 比较并交换：

* CAS包含了3个操作数--需要读写的内存位置V, 进行比较的值A和拟写入的新值B。
* 一个模拟的CAS操作：

/\*\*

\* 模拟CAS

\*/

public class SimulatedCAS {

private int value;

public synchronized int get(){

return value;

}

public synchronized int compareAndSwap(int expectedValue, int newValue){

int oldValue = value;

if (oldValue == expectedValue){

value = newValue;

}

return oldValue;

}

public synchronized boolean compareAndSet(int expectedValue, int newValue){

return (expectedValue == compareAndSwap(expectedValue, newValue));

}

}

### 非阻塞的计数器：

/\*\*

\* 基于CAS实现的非阻塞计数器

\*/

public class CasCounter {

private SimulatedCAS value;

public int getValue(){

return value.get();

}

public int increment(){

int v;

do {

v = value.get();

} while (v != value.compareAndSwap(v, v+1));

return v + 1;

}

}

* 在大多数处理器上，在无竞争的锁获取与释放的"快速代码路径"上的开销，大约CAS开销的两倍。

### JVM对CAS的支持：

* JVM为数字类型和引用类型提供了一些高效的CAS操作，如AtomicXxx。

## 原子变量类：

### 原子变量是一种"更好的volatile":

/\*\*

\* 通过CAS来维持包含多个变量的不变性条件

\*/

public class CasNumberRange {

private static class IntPair{

// 不变性条件: lower <= upper

final int lower;

final int upper;

public IntPair(int lower, int upper) {

this.lower = lower;

this.upper = upper;

}

}

private AtomicReference<IntPair> values = new AtomicReference<>();

public int getLower(){

return values.get().lower;

}

public int getUpper(){

return values.get().upper;

}

public void setLower(int i){

while (true){

IntPair oldv = values.get();

if (i > oldv.upper){

throw new IllegalArgumentException("lower can't > upper");

}

IntPair newv = new IntPair(i, oldv.upper);

if (values.compareAndSet(oldv, newv)){

return;

}

}

}

}

### 性能比较：锁与原子变量

* 基于ReentrantLock的随机数生成器：

/\*\*

\* 基于ReentrantLock实现的随机数生成器

\*/

public class ReentrantLockPreudoRandom extends PseudoRandom{

private final Lock lock = new ReentrantLock(false);

private int seed;

public ReentrantLockPreudoRandom(int seed){

this.seed = seed;

}

public int nextInt(int n){

lock.lock();

try{

int s = seed;

seed = calculateNext(s);

int remainder = s % n;

return remainder > 0 ? remainder : remainder + n;

} finally{

lock.unlock();

}

}

       ...

}

* 基于AtomicInteger的随机数生成器

/\*\*

\* 基于AtomicInteger实现的随机数生成器

\*/

public class AtomicPseudoRandom extends PseudoRandom{

private AtomicInteger seed;

public AtomicPseudoRandom(int seed){

this.seed = new AtomicInteger(seed);

}

public int nextInt(int n){

while (true){

int s = seed.get();

int nextSeed = calculateNext(s);

if (seed.compareAndSet(s, nextSeed)){

int remainder = s % n;

return remainder > 0 ? remainder: remainder + n;

}

}

}

       ...

}

## 非阻塞算法：

* 一个线程的失败或挂起不会导致其他线程也失败或挂起，那么这种算法就是非阻塞算法。

### 非阻塞栈实现：

/\*\*

\* 使用Treiber算法构造的非阻塞栈

\*/

public class ConcurrentStack<E> {

private AtomicReference<Node<E>> top = new AtomicReference<ConcurrentStack.Node<E>>();

public void push(E item){

Node<E> newHead = new Node<E>(item);

Node<E> oldHead;

do{

oldHead = top.get();

newHead.next = oldHead;

} while (!top.compareAndSet(oldHead, newHead));

}

public E pop(){

Node<E> oldHead;

Node<E> newHead;

do {

oldHead = top.get();

if (oldHead == null)

return null;

newHead = oldHead.next;

} while (!top.compareAndSet(oldHead, newHead));

return oldHead.item;

}

private static class Node<E>{

public final E item;

public Node<E> next;

public Node(E item){

this.item = item;

}

}

}

### 非阻塞链表的插入操作实现：

/\*\*

\* 链表中非阻塞算法中的插入排序，来自Michael-Scott

\*/

public class LinkedQueue<E> {

private static class Node<E>{

final E item;

final AtomicReference<Node<E>> next;

public Node(E item, Node<E> next){

this.item = item;

this.next = new AtomicReference<>(next);

}

}

private final Node<E> dummy = new Node<E>(null, null);

private final AtomicReference<Node<E>> head =

new AtomicReference<>(dummy);

private final AtomicReference<Node<E>> tail =

new AtomicReference<>(dummy);

public boolean put(E item){

Node<E> newNode = new Node<E>(item, null);

while (true){

Node<E> curTail = tail.get();

Node<E> tailNext = curTail.next.get();

if (curTail == tail.get()){ //尾部还未修改

if (tailNext != null){

// 队列处于中间状态(即新节点已经接上，尾节点还未更新)，推进尾节点

tail.compareAndSet(curTail, tailNext);

} else{

// 处于稳定状态, 尝试插入新节点

if (curTail.next.compareAndSet(null, newNode)){

// 插入成功后，推进尾节点

tail.compareAndSet(curTail, tailNext);

return true;

}

}

}

}

}

}

### 原子的域更新器：

private static class Node<E>{

private final E item;

private volatile Node<E> next; // volatile变量

public Node(E item){

this.item = item;

}

}

// 基于CAS的更新器

private static AtomicReferenceFieldUpdater<Node, Node> nextUpdater

= AtomicReferenceFieldUpdater.newUpdater(Node.class, Node.class, "next");

### ABA问题：

* ABA问题是一种一场现象：如果在算法中的节点可以被循环使用，那么在使用"比较并交换"时就可能出现这种问题。在某些算法中，V的值由A变为B, 再由B变为A,仍然被认为发生了变化，并需要重新执行算法中的某些步骤。
* 可通过AtomicStampedReference或AtomicMarkableReference变量来避免ABA问题。

# 十四、Java内存模型：

* 了解java内存模型，便于我们更容易理解如安全发布，同步策略的规范及一致性等机制。

## 什么是内存模型，为什么需要它

* JMM规定了JVM必须遵循一组最小保证，这组保证规定了对变量的写入操作在何时将对其他线程可见。

### 平台的内存模型

* JVM通过在适当的位置上插入内存栅栏来屏蔽在JMM与底层平台内存模型之间的差异。

### 重排序

**编译期重排序**

编译期重排序的典型就是通过调整指令顺序，在不改变程序语义的前提下，尽可能减少寄存器的读取、存储次数，充分复用寄存器的存储值。

假设第一条指令计算一个值赋给变量A并存放在寄存器中，第二条指令与A无关但需要占用寄存器（假设它将占用A所在的那个寄存器），第三条指令使用A的值且与第二条指令无关。那么如果按照顺序一致性模型，A在第一条指令执行过后被放入寄存器，在第二条指令执行时A不再存在，第三条指令执行时A重新被读入寄存器，而这个过程中，A的值没有发生变化。通常编译器都会交换第二和第三条指令的位置，这样第一条指令结束时A存在于寄存器中，接下来可以直接从寄存器中读取A的值，降低了重复读取的开销。

/\*\*

\* 如果在程序中没有包含足够的同步, 那么可能产生奇怪的结果

\*/

public class PossibleReordering {

static int x = 0, y = 0;

static int a = 0, b = 0;

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

Thread one = new Thread(new Runnable(){

@Override

public void run() {

a = 1;

x = b;

}

});

Thread other = new Thread(new Runnable(){

@Override

public void run() {

b = 1;

y = a;

}

});

one.start();

other.start();

one.join();

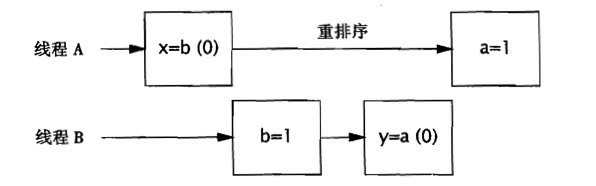
other.join();

System.out.println("x = " + x + ", y = " + y);

}

}

由于重排序导致的不确定性：

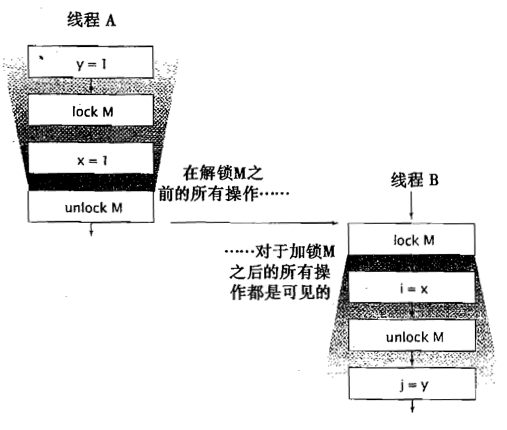


### Java内存模型简介

* Java内存模型是通过各种操作来定义的，包括变量的读/写操作，监视器的加锁和释放操作，以及线程的启动和合并操作。
* JMM为程序中所有的操作定义了一个偏序关系，称为Happens-Before。两个操作缺乏Happens-Before关系，则Jvm会对它们进行任意的重排序。
* Happends-Before的规则包括：

      1. 程序顺序规则。若程序中操作A在操作B之前，则线程中操作A在操作B之前执行。

      2. 监视器锁规则。在同一监视器锁上的解锁操作必须在加锁操作之前执行。如图所示，



      3. volatile变量规则。对volatile变量的写操作必须在读操作之前执行。

      4. 线程启动规则。Thread.start必须在线程中执行的操作之前被调用。

      5. 线程关闭规则。线程中的任何操作必须在其他线程检测到该线程结束之前执行，或者从Thread.join中返回，或调用Threas.isAlive返回false。

      6. 中断规则。当一个线程在另一个线程上调用interrupt时，必须在被中断线程检测到interrupt调用之前执行*。*

7. 终结器规则。对象的构造函数必须在启动该对象的终结器之前执行完成。

      8. 传递性。如果操作A在操作B之前执行，操作B在操作C之前执行，则操作A必须在操作C之前执行。

### 借助同步

* 一个借助同步的例子

public class Sync extends AbstractQueuedSynchronizer {

private static final int RUNNING = 1, RAN = 2, CANCELLED = 4;

private Object result;

private Exception exception;

void innerSet(Object o){

while (true){

int s = getState();

if (ranOrCancelled(s)){

return;

}

if (compareAndSetState(s, RAN)){ //设置状态

break;

}

}

result = o;

releaseShared(0); //共享模式下释放锁

done();

}

Object innerGet() throws InterruptedException, ExecutionException{

acquireSharedInterruptibly(0); //共享模式下请求锁

if (getState() == CANCELLED){

throw new CancellationException();

}

if (exception != null){

throw new ExecutionException(exception);

}

return result;

}

}

类库中提供的其他Happens-Before排序包括：

* 将一个元素放入一个线程安全容器的操作将在另一个线程从该容器中获得这个元素的操作之前执行。
* 在CountDownLatch上的倒数操作将在线程从闭锁上的await方法中返回之前执行。
* 释放Semaphore许可的操作将在从该Semaphore上获得一个许可之前执行。
* Future表示的任务的所有操作将在Future.get中返回之前执行。
* 向Executor提交一个Runnable或Callable将在任务开始执行之前执行。
* 一个线程到达CyclicBarrier或Exchanger的操作将在其他到达该栅栏或交换点的线程释放之前执行。如果CyclicBarrier使用一个栅栏操作，那么到达栅栏的操作将在栅栏操作之前执行，而栅栏操作又会在线程从栅栏中释放之前执行。

## 发布

* 造成不正确发布的真正原因："发布一个共享对象"与"另一个线程访问该对象"之间缺少一种Happens-Before的关系。

### 不安全的发布

/\*\*

\* 不安全的延迟初始化

\*/

public class UnsafeLazyInitialization {

private static Object resource;

public static Object getInstance(){

if (resource == null){

resource = new Object(); //不安全的发布

}

return resource;

}

}

* 除了不可变对象以外，使用被另一个线程初始化的对象通常都是不安全的，除非对象的发布操作是在使用该对象的线程开始使用之前执行。

### 安全的发布

* Happens-Before排序是在内存访问级别上操作的，它是一种"并发级汇编语言"，而安全发布的运行级别更接近程序设计。

### 安全初始化模式

通过同步实现安全发布，带来一定的性能开销

/\*\*

\* 通过同步安全发布

\*/

public class UnsafeLazyInitialization {

private static Object resource;

public synchronized static Object getInstance(){

if (resource == null){

resource = new Object();

}

return resource;

}

}

提前初始化实现安全发布：

/\*\*

\* 提前初始化

\*/

public class EagerInitialization {

private static Object resource = new Object();

public static Object getInstance(){

return resource;

}

}

类占位实现安全发布：

/\*\*

\* 延长初始化占位类模式

\*/

public class ResourceFactory {

private static class ResourceHolder{

public static Object resource = new Object();

}

public static Object getInstance(){

return ResourceHolder.resource;

}

}

### 双重检查加锁

/\*\*

\* 双重检查加锁, 不安全,

\* 线程可能看到无效的值, 可加上volatile修饰

\*/

public class DoubleCheckedLocking {

private static Object resource;

public static Object getInstance(){

if (resource == null){

synchronized (DoubleCheckedLocking.class){

if (resource == null){

resource = new Object();

}

}

}

return resource;

}

}

## 初始化过程中的安全性

/\*\*

\* 不可变对象的初始化安全性

\*/

public class SafeStates {

private final Map<String, String> states; //确保final，以免由于并发而被修改

public SafeStates(){

states = new HashMap<String, String>();

states.put("one", "One");

//...

states.put("two", "Two");

//...

}

public String getAbbreviation(String s){

return states.get(s);

}

}