Jiaxing Zhang

Java Concurrency Utilities

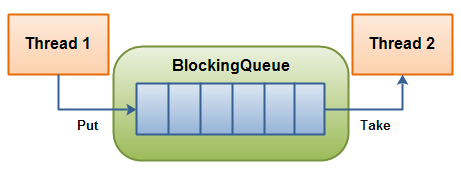
Java Concurrency Utilities

# BlockingQueue（阻塞队列）

java.util.concurrent 包里的 BlockingQueue 接口表示一个线程安放入和提取实例的队列。

## BlockingQueue Usage

BlockingQueue 通常用于一个线程生产对象，而另外一个线程消费这些对象的场景。下图是对这个原理的阐述：



1 A BlockingQueue with one thread putting into it, and another thread taking from it.

**一个线程往里边放，另外一个线程从里边取的一个 BlockingQueue。**

一个线程将会持续生产新对象并将其插入到队列之中，直到队列达到它所能容纳的临界点。也就是说，它是有限的。如果该阻塞队列到达了其临界点，负责生产的线程将会在往里边插入新对象时发生阻塞。它会一直处于阻塞之中，直到负责消费的线程从队列中拿走一个对象。

负责消费的线程将会一直从该阻塞队列中拿出对象。如果消费线程尝试去从一个空的队列中提取对象的话，这个消费线程将会处于阻塞之中，直到一个生产线程把一个对象丢进队列。

## BlockingQueue Methods

BlockingQueue 具有 4 组不同的方法用于插入、移除以及对队列中的元素进行检查。如果请求的操作不能得到立即执行的话，每个方法的表现也不同。这些方法如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| **Insert** | add(o) | offer(o) | put(o) | offer(o, timeout, timeunit) |
| **Remove** | remove(o) | poll() | take() | poll(timeout, timeunit) |
| **Examine** | element() | peek() |  |  |

四组不同的行为方式解释：

* **抛异常**：如果试图的操作无法立即执行，抛一个异常。
* **特定值**：如果试图的操作无法立即执行，返回一个特定的值(常常是 true / false)。
* **阻塞**：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行。
* **超时**：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行，但等待时间不会超过给定值。返回一个特定值以告知该操作是否成功(典型的是 true / false)。

**无法向一个 BlockingQueue 中插入 null。如果你试图插入 null，BlockingQueue 将会抛出一个 NullPointerException。**

可以访问到 BlockingQueue 中的所有元素，而不仅仅是开始和结束的元素。比如说，你将一个对象放入队列之中以等待处理，但你的应用想要将其取消掉。那么你可以调用诸如 remove(o) 方法来将队列之中的特定对象进行移除。***但是这么干效率并不高***(译者注：基于队列的数据结构，获取除开始或结束位置的其他对象的效率不会太高)，因此你尽量不要用这一类的方法，除非你确实不得不那么做。

## BlockingQueue Implementations

BlockingQueue 是个接口，你需要使用它的实现之一来使用 BlockingQueue。java.util.concurrent 具有以下 BlockingQueue 接口的实现(Java 6)：

* **ArrayBlockingQueue**
* **DelayQueue**
* **LinkedBlockingQueue**
* **PriorityBlockingQueue**
* **SynchronousQueue**

## Java BlockingQueue Example

这里是一个 Java 中使用 BlockingQueue 的示例。本示例使用的是 BlockingQueue 接口的 ArrayBlockingQueue 实现。

首先，BlockingQueueExample 类分别在两个独立的线程中启动了一个 Producer 和 一个 Consumer。Producer 向一个共享的 BlockingQueue 中注入字符串，而 Consumer 则会从中把它们拿出来。

public class BlockingQueueExample {

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

BlockingQueue queue = new ArrayBlockingQueue(1024);

Producer producer = new Producer(queue);

Consumer consumer = new Consumer(queue);

new Thread(producer).start();

new Thread(consumer).start();

Thread.sleep(4000);

}

}

以下是 Producer 类。注意它在每次 put() 调用时是如何休眠一秒钟的。这将导致 Consumer 在等待队列中对象的时候发生阻塞。

public class Producer implements Runnable{

protected BlockingQueue queue = null;

public Producer(BlockingQueue queue) {

this.queue = queue;

}

public void run() {

try {

queue.put("1");

Thread.sleep(1000);

queue.put("2");

Thread.sleep(1000);

queue.put("3");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

以下是 Consumer 类。它只是把对象从队列中抽取出来，然后将它们打印到 System.out。

public class Consumer implements Runnable{

protected BlockingQueue queue = null;

public Consumer(BlockingQueue queue) {

this.queue = queue;

}

public void run() {

try {

System.out.println(queue.take());

System.out.println(queue.take());

System.out.println(queue.take());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

# ArrayBlockingQueue

ArrayBlockingQueue 类实现了 BlockingQueue 接口。

ArrayBlockingQueue 是一个***有界的***阻塞队列，其内部实现是将对象放到一个数组里。有界也就意味着，它**不能够存储无限多数量的元素**。它有一个同一时间能够存储元素数量的上限。你可以在对其初始化的时候设定这个上限，但之后就无法对这个上限进行修改了(译者注：因为它是基于数组实现的，也就具有数组的特性：一旦初始化，大小就无法修改)。

ArrayBlockingQueue 内部以 **FIFO**(先进先出)的顺序对元素进行存储。队列中的头元素在所有元素之中是放入时间最久的那个，而尾元素则是最短的那个。

以下是在使用 ArrayBlockingQueue 的时候对其初始化的一个示例：

BlockingQueue queue = new ArrayBlockingQueue(1024);

queue.put("1");

Object object = queue.take();

以下是使用了 Java 泛型的一个 BlockingQueue 示例。注意其中是如何对 String 元素放入和提取的：

BlockingQueue<String> queue = new ArrayBlockingQueue<String>(1024);

queue.put("1");

String string = queue.take();

# DelayQueue（延迟队列）

DelayQueue 实现了 BlockingQueue 接口。

DelayQueue 将会在每个元素的 getDelay() 方法返回的值的时间段之后才释放掉该元素。如果返回的是 0 或者负值，延迟将被认为过期，该元素将会在 DelayQueue 的下一次 take 被调用的时候被释放掉。

以下是使用 DelayQueue 的例子：

public class DelayQueueExample {

public static void main(String[] args) {

DelayQueue queue = new DelayQueue();

Delayed element1 = new DelayedElement();

queue.put(element1);

Delayed element2 = queue.take();

}

}

# LinkedBlockingQueue（链表阻塞队列）

LinkedBlockingQueue 类实现了 BlockingQueue 接口。

LinkedBlockingQueue 内部以一个链式结构(链接节点)对其元素进行存储。如果需要的话，这一链式结构可以选择一个上限。如果没有定义上限，将使用 Integer.MAX\_VALUE 作为上限。

LinkedBlockingQueue 内部以 **FIFO(先进先出, PUT 1 2 3; TAKE 1 2 3)**的顺序对元素进行存储。队列中的头元素在所有元素之中是放入时间最久的那个，而尾元素则是最短的那个。

以下是 LinkedBlockingQueue 的初始化和使用示例代码：

BlockingQueue<String> unbounded = new LinkedBlockingQueue<String>();

BlockingQueue<String> bounded = new LinkedBlockingQueue<String>(1024);

bounded.put("Value");

String value = bounded.take();

# PriorityBlockingQueue（基于优先级的阻塞队列）

PriorityBlockingQueue 类实现了 BlockingQueue 接口。

PriorityBlockingQueue 是一个**无界的并发队列**。它使用了和类 java.util.PriorityQueue 一样的排序规则。你无法向这个队列中插入 null 值。

所有插入到 PriorityBlockingQueue 的元素必须实现 java.lang.Comparable 接口。因此**该队列中元素的排序就取决于你自己的 Comparable 实现**。

注意 PriorityBlockingQueue 对于具有相等优先级(compare() == 0)的元素并不强制任何特定行为。

同时注意，如果你从一个 PriorityBlockingQueue 获得一个 Iterator 的话，该 Iterator 并不能保证它对元素的遍历是以优先级为序的。

以下是使用 PriorityBlockingQueue 的示例：

BlockingQueue queue = new PriorityBlockingQueue();

//String implements java.lang.Comparable

queue.put("Value");

String value = queue.take();

# SynchronousQueue（同步队列）

SynchronousQueue 类实现了 BlockingQueue 接口。

SynchronousQueue 是一个特殊的队列，它的内部同时只能够容纳单个元素。如果该队列已有一元素的话，试图向队列中插入一个新元素的线程将会阻塞，直到另一个线程将该元素从队列中抽走。同样，如果该队列为空，试图向队列中抽取一个元素的线程将会阻塞，直到另一个线程向队列中插入了一条新的元素。

据此，把这个类称作一个队列显然是夸大其词了。它更多像是一个汇合点。

# BlockingDeque（阻塞双端队列）

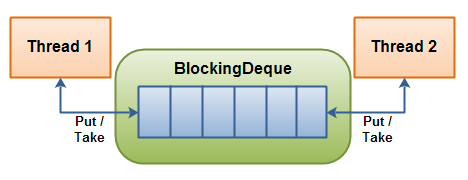
java.util.concurrent 包里的 BlockingDeque 接口表示一个线程安放入和提取实例的双端队列。本小节我将给你演示如何使用 BlockingDeque。

BlockingDeque 类是一个双端队列，在不能够插入元素时，它将阻塞住试图插入元素的线程；在不能够抽取元素时，它将阻塞住试图抽取的线程。

deque(双端队列) 是 "Double Ended Queue" 的缩写。因此，双端队列是一个你可以从任意一端插入或者抽取元素的队列。

## BlockingDeque Usage

在线程既是一个队列的生产者又是这个队列的消费者的时候可以使用到 BlockingDeque。如果生产者线程需要在队列的两端都可以插入数据，消费者线程需要在队列的两端都可以移除数据，这个时候也可以使用 BlockingDeque。BlockingDeque 图解：



1A BlockingDeque - threads can put and take from both ends of the deque.

一个 BlockingDeque - 线程在双端队列的两端都可以插入和提取元素。

一个线程生产元素，并把它们插入到队列的任意一端。如果双端队列已满，插入线程将被阻塞，直到一个移除线程从该队列中移出了一个元素。如果双端队列为空，移除线程将被阻塞，直到一个插入线程向该队列插入了一个新元素。

## BlockingDeque methods

BlockingDeque 具有 4 组不同的方法用于插入、移除以及对双端队列中的元素进行检查。如果请求的操作不能得到立即执行的话，每个方法的表现也不同。这些方法如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws** **Exception** | **Special Value** | **Blocks** | **Times** **Out** |
| **Insert** | addFirst(o) | offerFirst(o) | putFirst(o) | offerFirst(o, timeout, timeunit) |
| **Remove** | removeFirst(o) | pollFirst(o) | takeFirst(o) | pollFirst(timeout, timeunit) |
| **Examine** | getFirst(o) | peekFirst(o) |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws** **Exception** | **Special** **Value** | **Blocks** | **Times** **Out** |
| **Insert** | addLast(o) | offerLast(o) | putLast(o) | offerLast(o, timeout, timeunit) |
| **Remove** | removeLast(o) | pollLast(o) | takeLast(o) | pollLast(timeout, timeunit) |
| **Examine** | getLast(o) | peekLast(o) |  |  |

四组不同的行为方式解释：

1. **抛异常**：如果试图的操作无法立即执行，抛一个异常。
2. **特定值**：如果试图的操作无法立即执行，返回一个特定的值(常常是 true / false)。
3. **阻塞**：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行。
4. **超时**：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行，但等待时间不会超过给定值。返回一个特定值以告知该操作是否成功(典型的是 true / false)。

## BlockingDeque Extends BlockingQueue

BlockingDeque 接口继承自 BlockingQueue 接口。这就意味着你可以像使用一个 BlockingQueue 那样使用 BlockingDeque。如果你这么干的话，各种插入方法将会把新元素添加到双端队列的尾端，而移除方法将会把双端队列的首端的元素移除。正如 BlockingQueue 接口的插入和移除方法一样。

以下是 BlockingDeque 对 BlockingQueue 接口的方法的具体内部实现：

|  |  |
| --- | --- |
| BlockingQueue | BlockingDeque |
| add() | addLast() |
| offer() x 2 | offerLast() x 2 |
| put() | putLast() |
|  |  |
| remove() | removeFirst() |
| poll() x 2 | pollFirst() |
| take() | takeFirst() |
|  |  |
| element() | getFirst() |
| peek() | peekFirst() |

## BlockingDeque Implementations

既然 BlockingDeque 是一个接口，那么你想要使用它的话就得使用它的众多的实现类的其中一个。java.util.concurrent 包提供了以下 BlockingDeque 接口的实现类：

* LinkedBlockingDeque

## BlockingDeque Code Example

Here is a small code example of how to use the BlockingDeque methods:

BlockingDeque<String> deque = new LinkedBlockingDeque<String>();

deque.addFirst("1");

deque.addLast("2");

String two = deque.takeLast();

String one = deque.takeFirst();

# LinkedBlockingDeque（链表阻塞双端队列）

LinkedBlockingDeque 类实现了 BlockingDeque 接口。

deque(双端队列) 是 "Double Ended Queue" 的缩写。因此，双端队列是一个你可以从任意一端插入或者抽取元素的队列。(译者注：唐僧啊，受不了。)

LinkedBlockingDeque 是一个双端队列，在它为空的时候，一个试图从中抽取数据的线程将会阻塞，无论该线程是试图从哪一端抽取数据。

以下是 LinkedBlockingDeque 实例化以及使用的示例：

BlockingDeque<String> deque = new LinkedBlockingDeque<String>();

deque.addFirst("1");

deque.addLast("2");

String two = deque.takeLast();

String one = deque.takeFirst();

# ExecutorService（执行器服务）

java.util.concurrent.ExecutorService 接口表示一个异步执行机制，使我们能够在后台执行任务。因此一个 **ExecutorService 很类似于一个线程池**。实际上，存在于 java.util.concurrent 包里的 ExecutorService 实现就是一个线程池实现。

## ExecutorService Example

以下是一个简单的 ExecutorService 例子：

ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

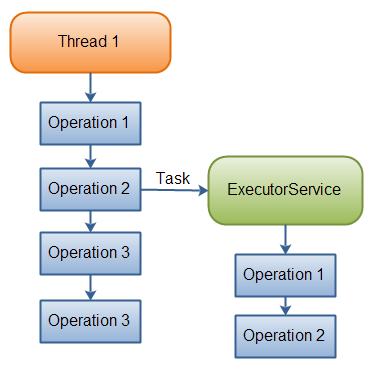
}

});

executorService.shutdown();

## Task Delegation

下图说明了一个线程是如何将一个任务委托给一个 ExecutorService 去异步执行的：



2A thread delegating a task to an ExecutorService for asynchronous execution.

**一个线程将一个任务委派给一个 ExecutorService 去异步执行。**

一旦该线程将任务委派给 ExecutorService，该线程将继续它自己的执行，独立于该任务的执行。

## ExecutorService Implementations

既然 ExecutorService 是个接口，如果你想用它的话就得去使用它的实现类之一。java.util.concurrent 包提供了 ExecutorService 接口的以下实现类：

* **ThreadPoolExecutor**
* **ScheduledThreadPoolExecutor**

## Creating an ExecutorService

ExecutorService 的创建依赖于你使用的具体实现。但是你也可以使用 Executors 工厂类来创建 ExecutorService 实例。以下是几个创建 ExecutorService 实例的例子：

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

## ExecutorService Usage

有几种不同的方式来将任务委托给 ExecutorService 去执行：

* execute(Runnable)
* submit(Runnable)
* submit(Callable)
* invokeAny(...)
* invokeAll(...)

### execute(Runnable)

execute(Runnable) 方法要求一个 java.lang.Runnable 对象，然后对它进行异步执行。以下是使用 ExecutorService 执行一个 Runnable 的示例：

ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

没有办法得知被执行的 Runnable 的执行结果。如果有需要的话你得使用一个 Callable(以下将做介绍)。

### submit(Runnable)

submit(Runnable) 方法也要求一个 Runnable 实现类，但它返回一个 Future 对象。这个 Future 对象可以用来检查 Runnable 是否已经执行完毕。

以下是 ExecutorService submit() 示例：

Future future = executorService.submit(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

future.get(); //returns null if the task has finished correctly.

### submit(Callable)

submit(Callable) 方法类似于 submit(Runnable) 方法，除了它所要求的参数类型之外。Callable 实例除了它的 call() 方法能够返回一个结果之外和一个 Runnable 很相像。Runnable.run() 不能够返回一个结果。

Callable 的结果可以通过 submit(Callable) 方法返回的 Future 对象进行获取。以下是一个 ExecutorService Callable 示例：

Future future = executorService.submit(new Callable(){

public Object call() throws Exception {

System.out.println("Asynchronous Callable");

return "Callable Result";

}

});

System.out.println("future.get() = " + future.get());

以上代码输出：

Asynchronous Callable

future.get() = Callable Result

### invokeAny()

invokeAny() 方法要求一系列的 Callable 或者其子接口的实例对象。调用这个方法并不会返回一个 Future，但它返回其中一个 Callable 对象的结果。无法保证返回的是哪个 Callable 的结果 - 只能表明其中一个已执行结束。

如果其中一个任务执行结束(或者抛了一个异常)，其他 Callable 将被取消。

以下是示例代码：

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

String result = executorService.invokeAny(callables);

System.out.println("result = " + result);

executorService.shutdown();

上述代码将会打印出给定 Callable 集合中的一个的执行结果。我自己试着执行了它几次，结果始终在变。有时是 "Task 1"，有时是 "Task 2" 等等。

### invokeAll()

invokeAll() 方法将调用你在集合中传给 ExecutorService 的所有 Callable 对象。invokeAll() 返回一系列的 Future 对象，通过它们你可以获取每个 Callable 的执行结果。

记住，一个任务可能会由于一个异常而结束，因此它可能没有 "成功"。无法通过一个 Future 对象来告知我们是两种结束中的哪一种。

以下是一个代码示例：

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){

System.out.println("future.get = " + future.get());

}

executorService.shutdown();

## ExecutorService Shutdown

使用完 ExecutorService 之后你应该将其关闭，以使其中的线程不再运行。

比如，如果你的应用是通过一个 main() 方法启动的，之后 main 方法退出了你的应用，如果你的应用有一个活动的 ExexutorService 它将还会保持运行。ExecutorService 里的活动线程阻止了 JVM 的关闭。

要终止 ExecutorService 里的线程你需要调用 ExecutorService 的 shutdown() 方法。ExecutorService 并不会立即关闭，但它将不再接受新的任务，而且一旦所有线程都完成了当前任务的时候，ExecutorService 将会关闭。在 shutdown() 被调用之前所有提交给 ExecutorService 的任务都被执行。

如果你想要立即关闭 ExecutorService，你可以调用 shutdownNow() 方法。这样会立即尝试停止所有执行中的任务，并忽略掉那些已提交但尚未开始处理的任务。无法担保执行任务的正确执行。可能它们被停止了，也可能已经执行结束。

# ThreadPoolExecutor

java.util.concurrent.ThreadPoolExecutor 是 ExecutorService 接口的一个实现。ThreadPoolExecutor 使用其内部池中的线程执行给定任务(Callable 或者 Runnable)。

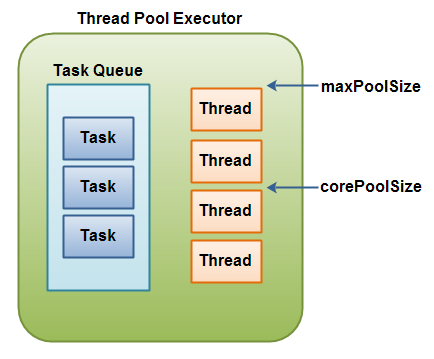
ThreadPoolExecutor 包含的线程池能够包含不同数量的线程。池中线程的数量由以下变量决定：

* corePoolSize
* maximumPoolSize

当一个任务委托给线程池时，如果池中线程数量低于 corePoolSize，一个新的线程将被创建，即使池中可能尚有空闲线程。

如果内部任务队列已满，而且有至少 corePoolSize 正在运行，但是运行线程的数量低于 maximumPoolSize，一个新的线程将被创建去执行该任务。

ThreadPoolExecutor 图解：



3A ThreadPoolExecutor

## Creating a ThreadPoolExecutor

ThreadPoolExecutor 有若干个可用构造子。比如：

int corePoolSize = 5;

int maxPoolSize = 10;

long keepAliveTime = 5000;

ExecutorService threadPoolExecutor =

new ThreadPoolExecutor(

corePoolSize,

maxPoolSize,

keepAliveTime,

TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>()

);

但是，除非你确实需要显式为 ThreadPoolExecutor 定义所有参数，使用 java.util.concurrent.Executors 类中的工厂方法之一会更加方便，正如 ExecutorService 小节所述。

# ScheduledExecutorService

java.util.concurrent.ScheduledExecutorService 是一个 ExecutorService， 它能够将任务延后执行，或者间隔固定时间多次执行。 任务由一个工作者线程异步执行，而不是由提交任务给 ScheduledExecutorService 的那个线程执行。

## ScheduledExecutorService Example

以下是一个简单的 ScheduledExecutorService 示例：

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =

scheduledExecutorService.schedule(new Callable() {

public Object call() throws Exception {

System.out.println("Executed!");

return "Called!";

}

},

5,

TimeUnit.SECONDS);

首先一个内置 5 个线程的 ScheduledExecutorService 被创建。之后一个 Callable 接口的匿名类示例被创建然后传递给 schedule() 方法。后边的俩参数定义了 Callable 将在 5 秒钟之后被执行。

## ScheduledExecutorService Implementations

既然 ScheduledExecutorService 是一个接口，你要用它的话就得使用 java.util.concurrent 包里对它的某个实现类。ScheduledExecutorService 具有以下实现类：

* ScheduledThreadPoolExecutor

## Creating a ScheduledExecutorService

如何创建一个 ScheduledExecutorService 取决于你采用的它的实现类。但是你也可以使用 Executors 工厂类来创建一个 ScheduledExecutorService 实例。比如：

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

## ScheduledExecutorService Usage

一旦你创建了一个 ScheduledExecutorService，你可以通过调用它的以下方法：

* schedule (Callable task, long delay, TimeUnit timeunit)
* schedule (Runnable task, long delay, TimeUnit timeunit)
* scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)
* scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

### schedule (Callable task, long delay, TimeUnit timeunit)

这个方法计划指定的 Callable 在给定的延迟之后执行。

这个方法返回一个 ScheduledFuture，通过它你可以在它被执行之前对它进行取消，或者在它执行之后获取结果。

以下是一个示例：

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =

scheduledExecutorService.schedule(new Callable() {

public Object call() throws Exception {

System.out.println("Executed!");

return "Called!";

}

},

5,

TimeUnit.SECONDS);

System.out.println("result = " + scheduledFuture.get());

scheduledExecutorService.shutdown();

This example outputs:

Executed!

result = Called!

### schedule (Runnable task, long delay, TimeUnit timeunit)

除了 Runnable 无法返回一个结果之外，这一方法工作起来就像以一个 Callable 作为一个参数的那个版本的方法一样，因此 ScheduledFuture.get() 在任务执行结束之后返回 null。

### scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)

这一方法规划一个任务将被定期执行。该任务将会在首个 initialDelay 之后得到执行，然后每个 period 时间之后重复执行。

如果给定任务的执行抛出了异常，该任务将不再执行。如果没有任何异常的话，这个任务将会持续循环执行到 ScheduledExecutorService 被关闭。

如果一个任务占用了比计划的时间间隔更长的时候，下一次执行将在当前执行结束执行才开始。计划任务在同一时间不会有多个线程同时执行。

### scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

除了 period 有不同的解释之外这个方法和 scheduleAtFixedRate() 非常像。

scheduleAtFixedRate() 方法中，period 被解释为前一个执行的开始和下一个执行的开始之间的间隔时间。

而在本方法中，period 则被解释为前一个执行的结束和下一个执行的结束之间的间隔。因此这个延迟是执行结束之间的间隔，而不是执行开始之间的间隔。

## ScheduledExecutorService Shutdown

正如 ExecutorService，在你使用结束之后你需要把 ScheduledExecutorService 关闭掉。否则他将导致 JVM 继续运行，即使所有其他线程已经全被关闭。

你可以使用从 ExecutorService 接口继承来的 shutdown() 或 shutdownNow() 方法将 ScheduledExecutorService 关闭。参见 ExecutorService 关闭部分以获取更多信息。

# Copy On Write Collection

## 什么是CopyOnWrite容器

CopyOnWrite容器即**写时复制**的容器。通俗的理解是当我们**往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器**。这样做的好处是我们可以对CopyOnWrite容器**进行并发的读，而不需要加锁**，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

### CopyOnWriteArrayList的实现原理

添加需要加锁

public boolean add(E e) {

final ReentrantLock lock = this.lock;

lock.lock();

try {

Object[] elements = getArray();

int len = elements.length;

Object[] newElements = Arrays.copyOf(elements, len + 1);

newElements[len] = e;

setArray(newElements);

return true;

} finally {

lock.unlock();

}

}

读取不需要加锁

private E get(Object[] a, int index) {

return (E) a[index];

}

## CopyOnWrite的应用场景

CopyOnWrite并发容器用于**读多写少的并发场景**。比如白名单，黑名单，商品类目的访问和更新场景，假如我们有一个搜索网站，用户在这个网站的搜索框中，输入关键字搜索内容，但是某些关键字不允许被搜索。这些不能被搜索的关键字会被放在一个黑名单当中，黑名单每天晚上更新一次。当用户搜索时，会检查当前关键字在不在黑名单当中，如果在，则提示不能搜索。

## CopyOnWrite使用注意

1. **减少扩容开销**。根据实际需要，初始化CopyOnWriteMap的大小，避免写时CopyOnWriteMap扩容的开销。

2. **使用批量添加**。因为每次添加，容器每次都会进行复制，所以减少添加次数，可以减少容器的复制次数。如使用上面代码里的addBlackList方法。

## CopyOnWrite的缺点

* **内存占用问题**。因为CopyOnWrite的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象（注意:在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。如果这些对象占用的内存比较大，比如说200M左右，那么再写入100M数据进去，内存就会占用300M，那么这个时候很有可能造成频繁的Yong GC和Full GC。之前我们系统中使用了一个服务由于每晚使用CopyOnWrite机制更新大对象，造成了每晚15秒的Full GC，应用响应时间也随之变长。针对内存占用问题，可以通过压缩容器中的元素的方法来减少大对象的内存消耗，比如，如果元素全是10进制的数字，可以考虑把它压缩成36进制或64进制。或者不使用CopyOnWrite容器，而使用其他的并发容器，如ConcurrentHashMap。
* **数据一致性问题**。CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。

# ConcurrentHashMap

* **线程不安全的HashMap:** 因为多线程环境下，使用Hashmap进行put操作会引起死循环，导致CPU利用率接近100%，所以在并发情况下不能使用HashMap。
* **效率低下的HashTable容器:** HashTable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法时，其他线程访问HashTable的同步方法时，可能会进入阻塞或轮询状态。如线程1使用put进行添加元素，线程2不但不能使用put方法添加元素，并且也不能使用get方法来获取元素，所以竞争越激烈效率越低。
* **ConcurrentHashMap的锁分段技术**: 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。

我们通过ConcurrentHashMap的类图来分析ConcurrentHashMap的结构。

# 并发相关知识

## 竞态条件 & 临界区

当两个线程竞争同一资源时，如果对资源的访问顺序敏感，就称存在**竞态条件**。导致竞态条件发生的代码区称作**临界区**。

## 线程安全与共享资源

### 局部变量（绝对线程安全）

局部变量存储在线程自己的栈中。也就是说，**局部变量永远也不会被多个线程共享**。所以，**基础类型的局部变量是线程安全的**。下面是基础类型的局部变量的一个例子：

public void someMethod(){

long threadSafeInt = 0;

threadSafeInt++;

}

### 局部的对象引用（没有传递给其他线程即为线程安全）

对象的局部引用和基础类型的局部变量不太一样。尽管引用本身没有被共享，但引用所指的对象并没有存储在线程的栈内。所有的对象都存在共享堆中。如果在某个方法中创建的对象不会逃逸出（译者注：**即该对象不会被其它方法获得，也不会被非局部变量引用到**）该方法，那么它就是线程安全的。实际上，哪怕将这个对象作为参数传给其它方法，只要别的线程获取不到这个对象，那它仍是线程安全的。

public void someMethod(){

LocalObject localObject = new LocalObject();

localObject.callMethod();

method2(localObject);

}

public void method2(LocalObject localObject){

localObject.setValue("value");

}

样例中LocalObject对象没有被方法返回，也没有被传递给someMethod()方法外的对象。每个执行someMethod()的线程都会创建自己的LocalObject对象，并赋值给localObject引用。因此，这里的LocalObject是线程安全的。事实上，整个someMethod()都是线程安全的。即使将LocalObject作为参数传给同一个类的其它方法或其它类的方法时，它仍然是线程安全的。当然，如果LocalObject通过某些方法被传给了别的线程，那它就不再是线程安全的了。

### 对象成员（线程不安全）

对象成员存储在堆上。如果两个线程同时更新同一个对象的同一个成员，那这个代码就不是线程安全的。下面是一个样例：

public class NotThreadSafe{

StringBuilder builder = new StringBuilder();

public add(String text){

this.builder.append(text);

}

}

## 线程控制逃逸规则

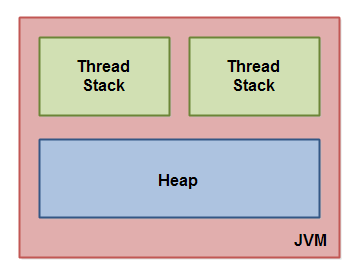
**如果一个资源的创建，使用，销毁都在同一个线程内完成，且永远不会脱离该线程的控制，则该资源的使用就是线程安全的。**

总结：

1. 局部变量中的基本数据类型(8种)永远是线程安全的。
2. 局部变量中的对象类型只要不会被其他线程访问到，也是线程安全的。
3. 一个对象实例被多个线程同时访问时，他的成员变量就可能是线程不安全的。

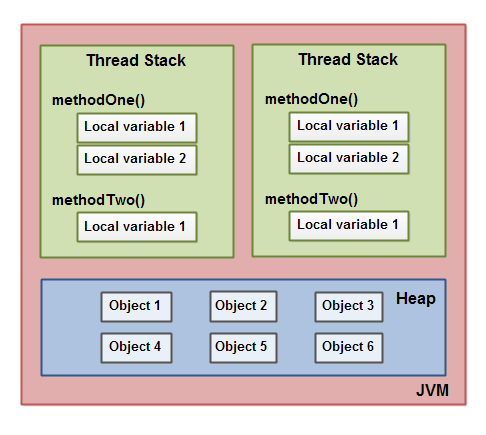
## Java内存模型（JDK1.8）

Java内存模型逻辑视图



* 每一个运行在Java虚拟机里的线程都拥有自己的**线程栈**。这个线程栈包含了这个线程调用的方法当前执行点相关的信息。一个线程仅能访问自己的线程栈。一个线程创建的本地变量对其它线程不可见，仅自己可见。即使两个线程执行同样的代码，这两个线程仍然在在自己的线程栈中的代码来创建本地变量。因此，每个线程拥有每个本地变量的独有版本。
* 所有原始类型的本地变量都存放在线程栈上，因此对其它线程不可见。一个线程可能向另一个线程传递一个原始类型变量的拷贝，但是它不能共享这个原始类型变量自身。
* 堆上包含在Java程序中创建的所有对象，无论是哪一个对象创建的。这包括原始类型的对象版本。如果一个对象被创建然后赋值给一个局部变量，或者用来作为另一个对象的成员变量，这个对象任然是存放在堆上。

下面这张图演示了调用栈和本地变量存放在线程栈上，对象存放在堆上。



* 一个本地变量可能是**原始类型**（8种原始数据类型），在这种情况下，它总是“呆在”线程栈上。
* 一个本地变量也可能是指向一个**对象的一个引用**。在这种情况下，引用（这个本地变量）存放在线程栈上，但是对象本身存放在堆上。
* 一个对象可能包含方法，这些方法可能包含本地变量。这些本地变量仍然存放在线程栈上，即使这些方法所属的对象存放在堆上。
* 一个对象的成员变量可能随着这个对象自身存放在堆上。不管这个成员变量是原始类型还是引用类型。
* 静态成员变量跟随着类定义一起也存放在堆上。
* 存放在堆上的对象可以被所有持有对这个对象引用的线程访问。当一个线程可以访问一个对象时，它也可以访问这个对象的成员变量。如果两个线程同时调用同一个对象上的同一个方法，它们将会都访问这个对象的成员变量，但是每一个线程都拥有这个本地变量的私有拷贝。



两个线程拥有一些列的本地变量。其中一个本地变量（Local Variable 2）执行堆上的一个共享对象（Object 3）。这两个线程分别拥有同一个对象的不同引用。这些引用都是本地变量，因此存放在各自线程的线程栈上。这两个不同的引用指向堆上同一个对象。

注意，这个共享对象（Object 3）持有Object2和Object4一个引用作为其成员变量（如图中Object3指向Object2和Object4的箭头）。通过在Object3中这些成员变量引用，这两个线程就可以访问Object2和Object4。

这张图也展示了指向堆上两个不同对象的一个本地变量。在这种情况下，指向两个不同对象的引用不是同一个对象。理论上，两个线程都可以访问Object1和Object5，如果两个线程都拥有两个对象的引用。但是在上图中，每一个线程仅有一个引用指向两个对象其中之一。

因此，什么类型的Java代码会导致上面的内存图呢？如下所示：

public class MyRunnable implements Runnable() {

public void run() {

methodOne();

}

public void methodOne() {

int localVariable1 = 45;

MySharedObject localVariable2 =

MySharedObject.sharedInstance;

//... do more with local variables.

methodTwo();

}

public void methodTwo() {

Integer localVariable1 = new Integer(99);

//... do more with local variable.

}

}

public class MySharedObject {

//static variable pointing to instance of MySharedObject

public static final MySharedObject sharedInstance =

new MySharedObject();

//member variables pointing to two objects on the heap

public Integer object2 = new Integer(22);

public Integer object4 = new Integer(44);

public long member1 = 12345;

public long member1 = 67890;

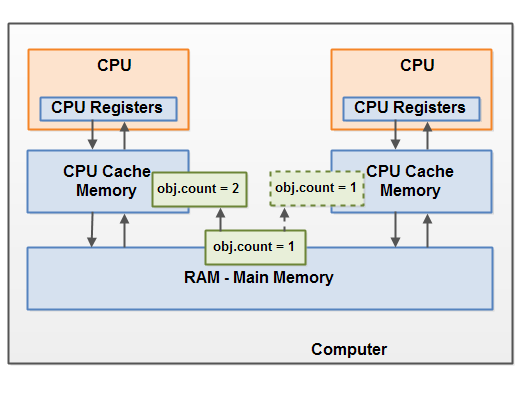
}

* 如果两个线程同时执行run()方法，就会出现上图所示的情景。run()方法调用methodOne()方法，methodOne()调用methodTwo()方法。
* methodOne()声明了一个原始类型的本地变量和一个引用类型的本地变量。
* 每个线程执行methodOne()都会在它们对应的线程栈上创建localVariable1和localVariable2的私有拷贝。localVariable1变量彼此完全独立，仅“生活”在每个线程的线程栈上。一个线程看不到另一个线程对它的localVariable1私有拷贝做出的修改。
* 每个线程执行methodOne()时也将会创建它们各自的localVariable2拷贝。然而，两个localVariable2的不同拷贝都指向堆上的同一个对象。代码中通过一个静态变量设置localVariable2指向一个对象引用。仅存在一个静态变量的一份拷贝，这份拷贝存放在堆上。因此，localVariable2的两份拷贝都指向由MySharedObject指向的静态变量的同一个实例。MySharedObject实例也存放在堆上。它对应于上图中的Object3。
* 注意，MySharedObject类也包含两个成员变量。这些成员变量随着这个对象存放在堆上。这两个成员变量指向另外两个Integer对象。这些Integer对象对应于上图中的Object2和Object4.
* 注意，methodTwo()创建一个名为localVariable的本地变量。这个成员变量是一个指向一个Integer对象的对象引用。这个方法设置localVariable1引用指向一个新的Integer实例。在执行methodTwo方法时，localVariable1引用将会在每个线程中存放一份拷贝。这两个Integer对象实例化将会被存储堆上，但是每次执行这个方法时，这个方法都会创建一个新的Integer对象，两个线程执行这个方法将会创建两个不同的Integer实例。methodTwo方法创建的Integer对象对应于上图中的Object1和Object5。
* 还有一点，MySharedObject类中的两个long类型的成员变量是原始类型的。因为，这些变量是成员变量，所以它们任然随着该对象存放在堆上，仅有本地变量存放在线程栈上。

## 共享对象可见性

* 如果两个或者更多的线程在没有正确的使用volatile声明或者同步的情况下共享一个对象，一个线程更新这个共享对象可能对其它线程来说是不接见的。
* 共享对象被初始化在主存中。跑在CPU上的一个线程将这个共享对象读到CPU缓存中。然后修改了这个对象。只要CPU缓存没有被刷新会主存，对象修改后的版本对跑在其它CPU上的线程都是不可见的。这种方式可能导致每个线程拥有这个共享对象的私有拷贝，每个拷贝停留在不同的CPU缓存中。

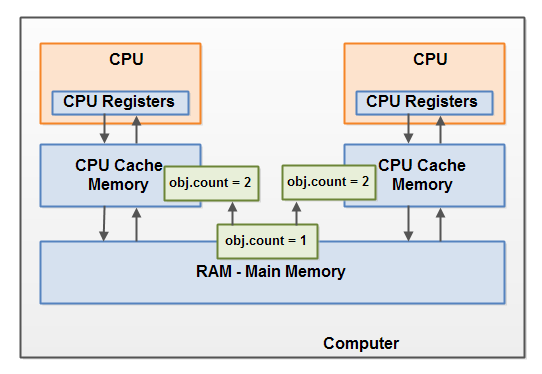
下图示意了这种情形。跑在左边CPU的线程拷贝这个共享对象到它的CPU缓存中，然后将count变量的值修改为2。这个修改对跑在右边CPU上的其它线程是不可见的，因为修改后的count的值还没有被刷新回主存中去。



解决这个问题你可以使用Java中的volatile关键字。volatile关键字可以保证直接从主存中读取一个变量，如果这个变量被修改后，总是会被写回到主存中去。

## Race Conditions

* 如果两个或者更多的线程共享一个对象，多个线程在这个共享对象上更新变量，就有可能发生race conditions。
* 如果线程A读一个共享对象的变量count到它的CPU缓存中。再想象一下，线程B也做了同样的事情，但是往一个不同的CPU缓存中。现在线程A将count加1，线程B也做了同样的事情。现在count已经被增在了两个，每个CPU缓存中一次。
* 如果这些增加操作被顺序的执行，变量count应该被增加两次，然后原值+2被写回到主存中去。
* 然而，两次增加都是在没有适当的同步下并发执行的。无论是线程A还是线程B将count修改后的版本写回到主存中取，修改后的值仅会被原值大1，尽管增加了两次。



解决这个问题可以使用**Java同步块**。一个同步块可以保证在同一时刻仅有一个线程可以进入代码的临界区。同步块还可以保证代码块中所有被访问的变量将会从主存中读入，当线程退出同步代码块时，所有被更新的变量都会被刷新回主存中去，不管这个变量是否被声明为volatile。

## Java同步块

### Java 同步关键字（synchronized）

Java中的同步块用synchronized标记。同步块在Java中是同步在某个对象上。所有同步在一个对象上的同步块在同时只能被一个线程进入并执行操作。所有其他等待进入该同步块的线程将被阻塞，直到执行该同步块中的线程退出。

有四种不同的同步块：

1. 实例方法
2. 静态方法
3. 实例方法中的同步块
4. 静态方法中的同步块

上述同步块都同步在不同对象上。实际需要那种同步块视具体情况而定。

#### 实例方法同步

下面是一个同步的实例方法：

public **synchronized** void add(int value){

this.count += value;

}

注意在方法声明中同步（synchronized ）关键字。这告诉Java该方法是同步的。

**Java实例方法同步是同步在拥有该方法的对象上**。这样，每个实例其方法同步都同步在不同的对象上，即该方法所属的实例。只有一个线程能够在实例方法同步块中运行。如果有多个实例存在，那么一个线程一次可以在一个实例同步块中执行操作。一个实例一个线程。

#### 静态方法同步

静态方法同步和实例方法同步方法一样，也使用synchronized 关键字。Java静态方法同步如下示例：

public **static** **synchronized** void add(int value){

count += value;

}

**静态方法的同步是指同步在该方法所在的类对象上**。因为在Java虚拟机中一个类只能对应一个类对象，所以同时只允许一个线程执行同一个类中的静态同步方法。

对于不同类中的静态同步方法，一个线程可以执行每个类中的静态同步方法而无需等待。不管类中的那个静态同步方法被调用，一个类只能由一个线程同时执行。

#### 实例方法中的同步块

有时你不需要同步整个方法，而是同步方法中的一部分。Java可以对方法的一部分进行同步。

在非同步的Java方法中的同步块的例子如下所示：

public void add(int value){

**synchronized**(this){

this.count += value;

}

}

示例使用Java同步块构造器来标记一块代码是同步的。该代码在执行时和同步方法一样。

注意Java同步块构造器用括号将对象括起来。在上例中，使用了“this”，即为调用add方法的实例本身。**在同步构造器中用括号括起来的对象叫做监视器对象**。上述代码使用监视器对象同步，同步实例方法使用调用方法本身的实例作为监视器对象。

一次只有一个线程能够在同步于同一个监视器对象的Java方法内执行。

下面两个例子都同步他们所调用的实例对象上，因此他们在同步的执行效果上是等效的。

public class MyClass {

public **synchronized** void log1(String msg1, String msg2){

log.writeln(msg1);

log.writeln(msg2);

}

public void log2(String msg1, String msg2){

**synchronized(this)**{

log.writeln(msg1);

log.writeln(msg2);

}

}

}

在上例中，每次只有一个线程能够在两个同步块中任意一个方法内执行（不允许同时访问，因为这两个是同一个监视器对象）。

如果第二个同步块不是同步在this实例对象上，那么两个方法可以被线程同时执行。

#### 静态方法中的同步块

和上面类似，下面是两个静态方法同步的例子。这些方法同步在该方法所属的类对象上。

public class MyClass {

public static **synchronized** void log1(String msg1, String msg2){

log.writeln(msg1);

log.writeln(msg2);

}

public static void log2(String msg1, String msg2){

**synchronized(MyClass.class)**{

log.writeln(msg1);

log.writeln(msg2);

}

}

}

这两个方法不允许同时被线程访问（因为这两个方法用的是同一个监视器对象）。

如果第二个同步块不是同步在MyClass.class这个对象上。那么这两个方法可以同时被线程访问。

### 同步案例

在下面例子中，启动了两个线程，都调用Counter类同一个实例的add方法。因为同步在该方法所属的实例上，所以同时只能有一个线程访问该方法。

public class Counter{

long count = 0;

public **synchronized** void add(long value){

this.count += value;

}

}

public class CounterThread extends Thread{

protected Counter counter = null;

public CounterThread(Counter counter){

this.counter = counter;

}

public void run() {

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

counter.add(i);

}

}

}

public class Example {

public static void main(String[] args){

Counter counter = new Counter();

Thread threadA = new CounterThread(counter);

Thread threadB = new CounterThread(counter);

threadA.start();

threadB.start();

}

}

如果两个线程引用了两个不同的Counter实例，那么他们可以同时调用add()方法。这些方法调用了不同的对象，因此这些方法也就同步在不同的对象上。这些方法调用将不会被阻塞。如下面这个例子所示：

public class Example {

public static void main(String[] args){

Counter counterA = new Counter();

Counter counterB = new Counter();

Thread threadA = new CounterThread(counterA);

Thread threadB = new CounterThread(counterB);

threadA.start();

threadB.start();

}

}

注意这两个线程，threadA和threadB，不再引用同一个counter实例。CounterA和counterB的add方法同步在他们所属的对象上。调用counterA的add方法将不会阻塞调用counterB的add方法。

### ThreadLocal

Java中的ThreadLocal类可以让你创建的变量只被同一个线程进行读和写操作。因此，尽管有两个线程同时执行一段相同的代码，而且这段代码又有一个指向同一个ThreadLocal变量的引用，但是这两个线程依然不能看到彼此的ThreadLocal变量域。

#### ThreadLocal初始化

由于ThreadLocal对象的set()方法设置的值只对当前线程可见，那有什么方法可以为ThreadLocal对象设置的值对所有线程都可见。可以通过ThreadLocal子类的实现，并覆写initialValue()方法，就可以为ThreadLocal对象指定一个初始化值。在set()方法调用前，当调用get()方法的时候，所有线程都可以看到同一个初始化值。

private ThreadLocal myThreadLocal = new ThreadLocal<String>() {

@Override protected String **initialValue()** {

return "This is the initial value";

}

};

#### InheritableThreadLocal

InheritableThreadLocal类是ThreadLocal的子类。为了解决ThreadLocal实例内部每个线程都只能看到自己的私有值，所以InheritableThreadLocal允许一个线程创建的所有子线程访问其父线程的值。

## Java线程通讯

线程通信的目标是使线程间能够互相发送信号。另一方面，线程通信使线程能够等待其他线程的信号。例如，线程B可以等待线程A的一个信号，这个信号会通知线程B数据已经准备好了。本文将讲解以下几个JAVA线程间通信的主题：

### 通过共享对象通信

线程间发送信号的一个简单方式是在共享对象的变量里设置信号值。线程A在一个同步块里设置boolean型成员变量hasDataToProcess为true，线程B也在同步块里读取hasDataToProcess这个成员变量。这个简单的例子使用了一个持有信号的对象，并提供了set和check方法:

public class MySignal{

protected boolean hasDataToProcess = false;

public synchronized boolean hasDataToProcess(){

return this.hasDataToProcess;

}

public synchronized void setHasDataToProcess(boolean hasData){

this.hasDataToProcess = hasData;

}

}

线程A和B必须获得指向一个MySignal共享实例的引用，以便进行通信。如果它们持有的引用指向不同的MySingal实例，那么彼此将不能检测到对方的信号。需要处理的数据可以存放在一个共享缓存区里，它和MySignal实例是分开存放的。

### 忙等待(Busy Wait)

准备处理数据的线程B正在等待数据变为可用。换句话说，它在等待线程A的一个信号，这个信号使hasDataToProcess()返回true。线程B运行在一个循环里，以等待这个信号：

protected MySignal sharedSignal = ...

...

while(!sharedSignal.hasDataToProcess()){

//do nothing... busy waiting

}

### wait(),notify()和notifyAll()

忙等待没有对运行等待线程的CPU进行有效的利用，除非平均等待时间非常短。否则，让等待线程进入睡眠或者非运行状态更为明智，直到它接收到它等待的信号。

Java有一个内建的等待机制来允许线程在等待信号的时候变为非运行状态。java.lang.Object 类定义了三个方法，wait()、notify()和notifyAll()来实现这个等待机制。

**一个线程一旦调用了任意对象的wait()方法，就会变为非运行状态，直到另一个线程调用了同一个对象的notify()方法。**为了调用wait()或者notify()，线程必须先获得那个对象的锁。也就是说，线程必须在同步块里调用wait()或者notify()。以下是MySingal的修改版本——使用了wait()和notify()的MyWaitNotify：

public class MonitorObject{

}

public class MyWaitNotify{

MonitorObject myMonitorObject = new MonitorObject();

public void doWait(){

synchronized(myMonitorObject){

try{

myMonitorObject.wait();

} catch(InterruptedException e){...}

}

}

public void doNotify(){

synchronized(myMonitorObject){

myMonitorObject.notify();

}

}

}

等待线程将调用doWait()，而唤醒线程将调用doNotify()。当一个线程调用一个对象的notify()方法，正在等待该对象的所有线程中将有一个线程被唤醒并允许执行（校注：这个将被唤醒的线程是随机的，不可以指定唤醒哪个线程）。同时也提供了一个notifyAll()方法来唤醒正在等待一个给定对象的所有线程。

注意：**不管是等待线程还是唤醒线程都在同步块里调用wait()和notify()。这是强制性的！**一个线程如果没有持有对象锁，将不能调用wait()，notify()或者notifyAll()。否则，会抛出IllegalMonitorStateException异常。

（校注：JVM是这么实现的，当你调用wait时候它首先要检查下当前线程是否是锁的拥有者，不是则抛出IllegalMonitorStateExcept，参考JVM源码的 1422行。）

但是，这怎么可能？等待线程在同步块里面执行的时候，不是一直持有监视器对象（myMonitor对象）的锁吗？等待线程不能阻塞唤醒线程进入doNotify()的同步块吗？答案是：的确不能。一旦线程调用了wait()方法，它就释放了所持有的监视器对象上的锁。这将允许其他线程也可以调用wait()或者notify()。

一旦一个线程被唤醒，不能立刻就退出wait()的方法调用，直到调用notify()的线程退出了它自己的同步块。换句话说：被唤醒的线程必须重新获得监视器对象的锁，才可以退出wait()的方法调用，因为wait方法调用运行在同步块里面。如果多个线程被notifyAll()唤醒，那么在同一时刻将只有一个线程可以退出wait()方法，因为每个线程在退出wait()前必须获得监视器对象的锁。

### 丢失的信号（Missed Signals）

notify()和notifyAll()方法不会保存调用它们的方法，因为当这两个方法被调用时，有可能没有线程处于等待状态。通知信号过后便丢弃了。因此，如果一个线程先于被通知线程调用wait()前调用了notify()，等待的线程将错过这个信号。这可能是也可能不是个问题。不过，在某些情况下，这可能使等待线程永远在等待，不再醒来，因为线程错过了唤醒信号。

为了避免丢失信号，必须把它们保存在信号类里。在MyWaitNotify的例子中，通知信号应被存储在MyWaitNotify实例的一个成员变量里。以下是MyWaitNotify的修改版本：

public class MyWaitNotify2{

MonitorObject myMonitorObject = new MonitorObject();

boolean wasSignalled = false;

public void doWait(){

synchronized(myMonitorObject){

if(!wasSignalled){

try{

myMonitorObject.wait();

} catch(InterruptedException e){...}

}

//clear signal and continue running.

wasSignalled = false;

}

}

public void doNotify(){

synchronized(myMonitorObject){

wasSignalled = true;

myMonitorObject.notify();

}

}

}

留意doNotify()方法在调用notify()前把wasSignalled变量设为true。同时，留意doWait()方法在调用wait()前会检查wasSignalled变量。事实上，如果没有信号在前一次doWait()调用和这次doWait()调用之间的时间段里被接收到，它将只调用wait()。

（校注：为了避免信号丢失， 用一个变量来保存是否被通知过。在notify前，设置自己已经被通知过。在wait后，设置自己没有被通知过，需要等待通知。）

### 假唤醒

由于莫名其妙的原因，线程有可能在没有调用过notify()和notifyAll()的情况下醒来。这就是所谓的假唤醒（spurious wakeups）。无端端地醒过来了。

如果在MyWaitNotify2的doWait()方法里发生了假唤醒，等待线程即使没有收到正确的信号，也能够执行后续的操作。这可能导致你的应用程序出现严重问题。

为了防止假唤醒，保存信号的成员变量将在一个while循环里接受检查，而不是在if表达式里。这样的一个while循环叫做自旋锁（校注：这种做法要慎重，目前的JVM实现自旋会消耗CPU，如果长时间不调用doNotify方法，doWait方法会一直自旋，CPU会消耗太大）。被唤醒的线程会自旋直到自旋锁(while循环)里的条件变为false。以下MyWaitNotify2的修改版本展示了这点：

public class MyWaitNotify3{

MonitorObject myMonitorObject = new MonitorObject();

boolean wasSignalled = false;

public void doWait(){

synchronized(myMonitorObject){

while(!wasSignalled){

try{

myMonitorObject.wait();

} catch(InterruptedException e){...}

}

//clear signal and continue running.

wasSignalled = false;

}

}

public void doNotify(){

synchronized(myMonitorObject){

wasSignalled = true;

myMonitorObject.notify();

}

}

}

留意wait()方法是在while循环里，而不在if表达式里。如果等待线程没有收到信号就唤醒，wasSignalled变量将变为false,while循环会再执行一次，促使醒来的线程回到等待状态。

### 多个线程等待相同信号

如果你有多个线程在等待，被notifyAll()唤醒，但只有一个被允许继续执行，使用while循环也是个好方法。每次只有一个线程可以获得监视器对象锁，意味着只有一个线程可以退出wait()调用并清除wasSignalled标志（设为false）。一旦这个线程退出doWait()的同步块，其他线程退出wait()调用，并在while循环里检查wasSignalled变量值。但是，这个标志已经被第一个唤醒的线程清除了，所以其余醒来的线程将回到等待状态，直到下次信号到来。

### 不要在字符串常量或全局对象中调用wait()

（校注：本章说的字符串常量指的是值为常量的变量）

本文早期的一个版本在MyWaitNotify例子里使用字符串常量（””）作为管程对象。以下是那个例子：

public class MyWaitNotify{

String myMonitorObject = "";

boolean wasSignalled = false;

public void doWait(){

synchronized(myMonitorObject){

while(!wasSignalled){

try{

myMonitorObject.wait();

} catch(InterruptedException e){...}

}

//clear signal and continue running.

wasSignalled = false;

}

}

public void doNotify(){

synchronized(myMonitorObject){

wasSignalled = true;

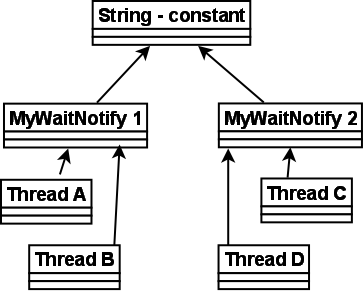
myMonitorObject.notify();

}

}

}

在空字符串作为锁的同步块(或者其他常量字符串)里调用wait()和notify()产生的问题是，JVM/编译器内部会把常量字符串转换成同一个对象。这意味着，即使你有2个不同的MyWaitNotify实例，它们都引用了相同的空字符串实例。同时也意味着存在这样的风险：在第一个MyWaitNotify实例上调用doWait()的线程会被在第二个MyWaitNotify实例上调用doNotify()的线程唤醒。这种情况可以画成以下这张图：



起初这可能不像个大问题。毕竟，如果doNotify()在第二个MyWaitNotify实例上被调用，真正发生的事不外乎线程A和B被错误的唤醒了 。这个被唤醒的线程（A或者B）将在while循环里检查信号值，然后回到等待状态，因为doNotify()并没有在第一个MyWaitNotify实例上调用，而这个正是它要等待的实例。这种情况相当于引发了一次假唤醒。线程A或者B在信号值没有更新的情况下唤醒。但是代码处理了这种情况，所以线程回到了等待状态。记住，即使4个线程在相同的共享字符串实例上调用wait()和notify()，doWait()和doNotify()里的信号还会被2个MyWaitNotify实例分别保存。在MyWaitNotify1上的一次doNotify()调用可能唤醒MyWaitNotify2的线程，但是信号值只会保存在MyWaitNotify1里。

问题在于，由于doNotify()仅调用了notify()而不是notifyAll()，即使有4个线程在相同的字符串（空字符串）实例上等待，只能有一个线程被唤醒。所以，如果线程A或B被发给C或D的信号唤醒，它会检查自己的信号值，看看有没有信号被接收到，然后回到等待状态。而C和D都没被唤醒来检查它们实际上接收到的信号值，这样信号便丢失了。这种情况相当于前面所说的丢失信号的问题。C和D被发送过信号，只是都不能对信号作出回应。

如果doNotify()方法调用notifyAll()，而非notify()，所有等待线程都会被唤醒并依次检查信号值。线程A和B将回到等待状态，但是C或D只有一个线程注意到信号，并退出doWait()方法调用。C或D中的另一个将回到等待状态，因为获得信号的线程在退出doWait()的过程中清除了信号值(置为false)。

看过上面这段后，你可能会设法使用notifyAll()来代替notify()，但是这在性能上是个坏主意。在只有一个线程能对信号进行响应的情况下，没有理由每次都去唤醒所有线程。

所以：在wait()/notify()机制中，不要使用全局对象，字符串常量等。应该使用对应唯一的对象。例如，每一个MyWaitNotify3的实例（前一节的例子）拥有一个属于自己的监视器对象，而不是在空字符串上调用wait()/notify()。