# 第一章

1. 线程会共享进程范围内的资源，例如内存句柄和文件句柄，但每个线程都有各自的程序计数器（Program Counter）、栈以及局部变量。
2. 通过使用线程，可以将复杂并异步的工作进一步分解为一组简单并且同步的工作流，每个工作流在一个单独的线程中运行，并在特定的同步位置进行交互。现有的一些框架可以实现上述目标，例如Servlet和RMI（Remote Mothod Invocation，远程方法调用）。
3. 安全性的含义是“永远不发生糟糕的事情”，而活跃性则关注另一个目标，即“某件正确的事情最终会发生”。当某个操作无法继续执行下去时，就会发生活跃性问题。在串行程序中，活跃性问题的形式之一就是无意中造成的无限循环。
4. 与活跃性问题密切相关的是性能问题。活跃性意味着某件正确的事情最终会发送，但却不够好，因为我们希望正确的事情尽快发送。

# 第二章

1. 要编写线程安全的代码，其核心在于要对状态访问操作进行管理，特别是对共享的（Shared）和可变的（Mutable）状态的访问。
2. 如果当多个线程访问同一个可变的状态变量时没有使用合适的同步，那么程序就会出现错误，有3中方式可以修复这个问题：
3. 不在线程之间共享该变量。
4. 将状态变量修改为不可变的变量。
5. 在访问状态变量时使用同步。
6. 在编写并发代码时，应始终遵循一个原则：首先使代码正确运行，然后才提高代码的运行速度。即便如此，最好也是当性能测试结果和应用需求告诉你必须提高性能，以及测量结果表明这种优化在实际环境中确实能带来性能提升时，才进行优化。
7. 在线程安全性的含义中，最核心的概念就是正确性，而正确性的含义是：某个类的行为与其规范完全一致。在良好的规范中通常会定义各种不变性条件（Invariant）来约束对象的状态，以及定义各种后验条件（Postcondition）来描述对象操作的结果。
8. 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，那么就称这个类是线程安全的。
9. 无状态对象一定是线程安全的。无状态：既不包含任何域，也不包含任何对其他类中域的引用。
10. 竞态条件：在并发编程中，这种由于不恰当的执行时序而出现不正确的结果是一种非常重要的情况。
11. 假定有两个操作A和B，如果从执行A的线程来看，当另一个线程执行B时，要么将B全部执行完，要么完全不执行B，那么A和B对彼此来说都是原子的。原子操作是指，对于访问同一个状态的所有操作（包括该操作本身）来说，这个操作是一个以原子方式执行的操作。
12. 复合操作：包含了一组必须以原子方式执行的操作以确保线程安全性。
13. 在实际情况中，应尽可能使用现有的线程安全对象（例如AtomicLong）来管理类的状态。与非线程安全的对象相比，判断线程安全对象的可能状态及其状态转换情况要更为容易，从而也更容易验证和维护线程安全性。
14. 每个Java对象都可以用做一个实现同步的锁，这些锁被称为内置锁（Intrinsic Lock）或监视器锁（Monitor Lock）。
15. Java的内置锁相当于一种互斥体（或互斥锁），这意味着最多只有一个线程能持有这个锁。
16. 内置锁是可以重入的，因此如果某个线程试图获得一个已经有它自己持有的锁，那么这个请求就会成功。
17. 对于可能被多个线程同时访问的可变状态变量，在访问它时都需要持有同一个锁，我们称状态变量是由这个锁保护的。
18. 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。一种常见的加锁约定是，将所有的可变状态都封装在对象内部，并通过对象的内置锁对所有访问可变状态的代码路径进行同步，使得在该对象上不会发生并发访问。
19. 在获取与释放锁等操作上都需要一定的开销，因此如果将同步代码块分解的过细，那么通常并不好，尽管这样不会破坏原子性。
20. 当执行时间较长的计算或者可能无法快速完成的操作时，一定不要持有锁。

# 第三章

1. Synchronized关键字不止能用于实现原子性或者确定“临界区（Critical Section）”，还有另一个重要的方面：内存可见性（Memory Visibility）。
2. 仅当volatile变量能简化代码的实现以及对同步策略的验证时，才应该使用它们。如果在验证正确性时需要对可见性进行复杂的判断，那么就不要使用volatile变量。Volatile变量的正确使用方式包括：确保它们自身状态的可见性，确保它们所引用对象的状态的可见性，以及标识一些重要的程序生命周期事件的发生。
3. 加锁机制既可以确保可见性又可以确保原子性，而volatile变量只能确保可见性。
4. 当且仅当满足以下所有条件时，才应该使用volatile变量：
5. 对变量的写入操作不依赖变量的当前值，或者你能确保只有单个线程更新变量的值。
6. 该变量不会与其他状态变量一起纳入不变性条件中。
7. 在访问变量时不需要加锁。
8. 发布（publish）一个对象的意思是指，是对象能够在当前作用域之外的代码中使用。
9. 逸出（Escape）：当某个不应该发布的对象被发布时。
10. 线程封闭：如果仅在单线程内访问数据，就不需要同步。它是实现线程安全性的最简单方式之一。
11. Ad-hoc线程封闭：指维护线程封闭性的职责完全由程序实现来承担。非常脆弱。程序中尽量少用。
12. 栈封闭：它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。局部变量的固有属性之一就是封闭在执行线程中。它们位于执行线程的栈中，其他线程无法访问这个栈。
13. ThreadLocal类：维护线程封闭性的一种更规范的方法是使用ThreadLocal类，这个类能使线程中的某个值与保存值的对象关联起来。ThreadLocal提供了get与set等访问接口或方法，这些方法为每个使用该变量的线程都存有一份独立的副本，因此get总是返回由当前执行线程在调用set时设置的最新值。
14. ThreadLocal对象通常用于防止对可变的单实例变量或全局变量进行共享。从概念上看，你可以将ThreadLocal<T>视为包含了Map<Thread,T>对象，其中保存了特定于线程的值，但ThreadLocal的实现并非如此。
15. ThreadLocal变量类似于全局变量，它能降低代码的可重用性，并在类之间引入隐含的耦合性，因此在使用时要格外小心。
16. 如果某个对象在被创建之后其状态就不能被修改，那么这个对象就称为不可变对象。线程安全性是不可变对象的固有属性之一。
17. 当满足以下条件时，对象才是不可变的：
18. 对象创建以后其状态就不能修改。
19. 对象的所有域都是final类型
20. 对象是正确创建的（在对象的创建期间，this引用没有逸出）
21. 关键字final可以视为C++中const机制的一种受限版本，用于构造不可变对象。final类型的域是不能修改的（但如果final域所引用的对象是可变的，那么这些被引用的对象是可以修改的）。
22. 任何线程都可以在不需要额外同步的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。
23. 要安全地发布一个对象，对象的引用以及对象的状态必须同时对其他线程可见。一个正确构造的对象可以通过以下方式来安全发布：
24. 在静态初始化函数中初始化一个对象引用。
25. 将对象的引用保存到volatile类型的域或者AtomicReference对象中。
26. 将对象的引用保存到某个正确构造对象的final类型域中。
27. 将对象的引用保存到一个由锁保护的域中。
28. 但线程安全库中的容器类提供了以下的安全发布保证：
29. 通过将一个键或者值放入HashTable、synchronizedMap或者ConcurrentMap中，可以安全将它发布给任何从这些容器中访问它的线程（无论是直接访问还是通过迭代器访问）；
30. 通过将某个元素放入Vector、CopyOnWriteArrayList、CopyOnWriteArraySet、synchronizedList或synchronizedSet中，可以将该元素安全地发布到任何从这些容器中访问该元素的线程。
31. 通过将某个元素放入BlockingQueue或者ConcurrentLinkedQueue中，可以将该元素安全地发布到任何从这些队列中访问该元素的线程。
32. 类库中的其他数据传递机制（例如Future和Exchanger）同样能实现安全发布。
33. 如果对象从技术上来说是可变的，但是其状态在发布后不会再改变，那么把这种对象称为“事实不可变对象”，同样是线程安全的。例如一个同步的Map中保存了Data的一个实例化对象，但这个值不会再改变，而Map又是同步的，所以这个Data实例化对象就是事实不可变对象。

# 第四章

1. 在设计线程安全类的过程中，需要包含以下三个基本要素：找出够成对象状态的所有变量，找出约束状态变量的不变性条件和建立对象状态的并发访问管理策略。
2. 包含多个变量的不变性条件将带来原子性需求：这些相关的变量必须在单个原子操作中进行读取或更新。
3. 如果不了解对象的不变性条件和后验条件，那么就不能保证线程安全性，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。
4. 类的不变性条件与后验条件约束了在对象上有哪些状态和状态转换是有效的。在某些对象的方法中还包含一些基于状态的先验条件。
5. 要想实现某个等待先验条件为真时才执行的操作，一种更简单的方法是通过现有的类库（例如阻塞队列BlockingQueue或信号量Semaphore来实现依赖状态的行为）。
6. 封装简化了线程安全类的实现过程，它提供了一种实例封闭机制（Instance Confinement），通常也简称为“封闭”。
7. 通过将封闭机制与合适的加锁策略结合起来，可以确保以线程安全的方式来使用非线程安全的对象。
8. 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。
9. 钝化操作（Passivation）：指的是将状态保存到持久性存储。
10. 一些基本的容器类并非是线程安全的，例如ArrayList和HashMap，但类库提供了包装器工厂方法（例如Collections.synchronizedList及其类似方法），使得这些非线程安全的类可以在多线程环境中安全的使用。这些工厂方法通过“装饰器（Decorator）”模式将容器类封装在一个同步的包装器对象中，而包装器能将接口中的每个方法都实现为同步方法，并将调用请求转发到底层的容器对象上。只要包装器对象拥有对底层容器对象的唯一引用（即把底层容器对象封闭在包装器中），那么它就是线程安全的。在这些方法的Javadoc指出，对底层容器对象的所有访问必须通过包装器来进行。
11. 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，在分析累的线程安全性时就无须检查整个程序。
12. 遵循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。
13. 在许多类中都使用了Java监视器模式，例如Vector和Hashtable。Java监视器模式的主要优势就在于它的简单性。
14. 使用私有的锁对象而不是对象的内置锁（或任何其他可通过公有方式访问的锁），有许多有点。私有的锁对象可将锁封装起来，使客户代码无法得到锁，但客户代码可以通过公有方法来访问锁，以便参与到它的同步策略中。
15. 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，那么就可以安全的发布这个变量。
16. 要添加一个新的原子操作，最安全的方式是修改原始的类，但这通常无法做到。另一个方法是扩展这个类，即继承这个类，并添加一个新方法，但是并非所有类都像Vector那样将状态向子类公开，因此这个方法也不合适。第三种方式是扩展类的功能，但并不是扩展类本身，而是将扩展代码放入一个辅助类中，即封装类。一种更好的方法是组合（Composition）。例如以下程序中ImprovedList通过将List对象的操作委托给底层的List实例来实现List的操作，同时还添加了一个原子的putIfAbsent方法。

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 x){

boolean contains = list.contains(x);

if(!contains){

list.add(x);

}

return !contains;

}

}

1. 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。这是尤其重要的！！！！！！

# 第五章

1. 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。
2. 同步容器类包括Vector和Hashtable，二者是早期JDK的一部分，此外还包括在JDK1.2中添加的一些功能相似的类，这些同步的封装器类是由Collections.synchronizedXxx等工厂方法创建的。这些类实现线程安全的方式是：将它们的状态封装起来，并对每个公有方法都进行同步，使得每次只有一个线程能访问容器的状态。
3. 尽管同步容器类是线程安全的，但在某些情况下需要额外的客户端加锁来保护复合操作：迭代（反复访问元素，知道遍历完容器中所有元素）、跳转（根据指定顺序找到当前元素的下一个元素）以及条件运算。

如果一个线程正对共享容器进行迭代，另一线程也正对这个共享容器进行写操作，就很可能发生运行时异常，抛出ConcurrentModificationException。这是个“及时失败”的预警异常

1. 为保证在复合操作中保证同步容器类的线程安全性，一种方法是在迭代期间加锁保护，另一种方法是“克隆”，并在副本上进行迭代。
2. 并发容器：ConcurrentHashMap替代同步且基于散列的Map。

CopyOnWriteArrayList代替同步的List（遍历操作为主的情况下）。

JDK 5.0新增：Queue和BlockingQueue。

Queue（非阻塞）提供了几种实现：ConcurrentLinkedQueue,PriorityQueue.

BlockingQueue（阻塞），在“生产者-消费者”设计模式中，非常有用。

JDK 6.0引入ConcurrentSkipListMap和ConcurrentSkipListSet，分别作为同步的SortedMap和SortedSet的并发替代品（例如用synchronizedMap包装的TreeMap或TreeSet）。

1. ConcurrentHashMap:加锁策略（分段锁 Lock Striping），更高的并发性和伸缩性，并非将锁加在每个方法上，而是一种粒度更细的加锁机制，从而实现更大程度的共享。在这种机制下，任意数量的读取线程可以并发地访问Map，执行读取操作的线程和执行写入操作的线程可以并发的访问Map，并且一定数量的写入线程可以并发的修改Map。ConcurrentHashMap带来的结果是，在并发访问环境下将实现更高的吞吐量，而在单线程中只损失非常小的性能。
2. 并发容器增强了同步容器：它们提供的迭代器不会抛出ConcurrentModificationException，因此不需要在迭代过程中对容器加锁。ConcurrentHashMap返回的迭代器具有弱一致性（Weakly Consistent），而并非“及时失败”。弱一致性的迭代器可以容忍并发的修改，当创建迭代器时会遍历已有的元素，并可以（但是不保证）在迭代器被构造后将修改操作反映给容器。
3. 与Hashtable和synchronizedMap相比，ConcurrentHashMap有着更多的优势以及更少的劣势，因此在大多数情况下，用ConcurrentHashMap来代替同步Map能进一步提高代码的可伸缩性，只有当应用程序需要加锁Map以进行独占访问时，才应该放弃使用ConcurrentHashMap。
4. 虽然ConcurrentHashMap不能被加锁来执行独占访问，因此无法使用客户端加锁来创建新的原子操作。但ConcurrentMap的接口中已经声明了例如“若没有则添加（putIfAbsent）”、“若相等则移除（remove）”和“若相等则替换（replace）”等原子操作方法。
5. CopyOnWriteArrayList：替代同步List，迭代期间不需要对容器加锁或复制。此对象在每次修改时，都会创建并重新发布一个新的容器副本，从而实现可变性。“写入时复制”容器的迭代器保留一个指向底层基础数组的引用，这个数组当前位于迭代器的起始位置，由于它不会被修改，因此在对其进行同步时只需确保数组内容的可见性。因此多个线程可以同时对这个容器进行迭代，而不会彼此干扰或者与修改容器的线程相互干扰。
6. 因为每当修改此容器都会复制底层数组，需要一定开销，特别是当容器规模较大。所以仅当迭代操作远多于修改操作时，才应该使用“写入时复制”容器。
7. 阻塞队列和生产者-消费者模式：一种最常见的生产者-消费者设计模式就是线程池与工作队列的组合，在Executor任务执行框架中就体现了这种模式。
8. 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：它们能抑制并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。
9. Java 6增加了两种容器类型，Deque（发音为“deck”）和BlockingDeque，它们分别对Queue和BlockingQueue进行了扩展。Deque是一个双端队列，实现了在队列头和队列尾的高效插入和移除。具体实现包括ArrayDeque和LinkedBlockingDeque。
10. 双端队列适用于一种相关模式：工作密取（Work Stealing）。在工作密取设计中，每个消费者都有各自的双端队列。如果一个消费者完成了自己双端队列中的全部工作，那么它可以从其他消费者的双端队列末尾秘密地获取工作。
11. 中断是一种协作机制。一个线程不能强制其他线程停止正在运行的操作而去执行其他操作。当线程A中断线程B时，A仅仅是要求B在执行到某个可以暂停的地方停止正在执行的操作----前提是线程B愿意停下来。
12. 当在代码中调用了一个将抛出InterruptedException异常的方法时，你自己的方法也就变成了一个阻塞方法，并且必须处理对中断的响应。对于库代码来说，有两种方式：传递InterruptedException，即不捕获这个异常或捕获之后，执行某种简单的清理工作后再次抛出这个异常。二是恢复中断，捕获异常，调用当前线程的interrupt方法恢复中断状态，这样在调用栈上更高层的代码将看到引发了一个中断。
13. 不应该在出现中断异常时，捕获它却不做出任何反应。
14. 阻塞队列（BlockingQueue）可以作为一种作为同步工具类，其他类型的同步工具类还有信号量（Semaphore）、栅栏（Barrier）以及闭锁（Latch）。
15. 闭锁是一种同步工具类，可以延迟线程的进度直到其到达终止状态。闭锁的作用相当于一扇门：在闭锁到达结束状态之前，这扇门一直是关闭的，并且没有任何线程都能通过，当到达结束状态时，这扇门就会打开并允许所有的线程通过。当闭锁到达结束状态后，将不会在改变其状态，因此这扇门将永远保持打开状态。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。具体实现有CountDownLatch。
16. FutureTask也可以用作闭锁。它表示的计算是通过Callable来实现的，相当于一种可生成结果的Runnable，并且其Future.get方法是一个阻塞方法。FutureTask在Executor框架中表示异步任务，此外还可以用来表示一些时间较长的计算，这些计算可以在使用计算结果之前启动。
17. 信号量（Semaphore）：用来控制同时访问某个特定资源的操作数量，或者同时执行某个特定操作的数量。（Counting Semaphore）计数信号量可以用来实现某种资源池，或者对容器施加边界。Semaphore中管理着一组虚拟的许可（permit），许可的初始数量可通过构造函数来指定，在执行操作时可以首先获得许可（只有还剩有许可），使用完之后释放许可。如果没许可，那么acquire将阻塞直到有许可，release方法将返回一个许可给信号量。其实，这种实现并不包含真正的许可对象，许可也没有和线程相关联。
18. 栅栏（Barrier）：闭锁是一次性对象，不能被重置，而栅栏可被重置。它类似于闭锁，但是闭锁用于等待事件，栅栏用于等待线程。它能阻塞一组线程直到某个事件发生，所有线程必须同时到达栅栏位置，才能继续执行。具体实现有CyclicBarrier和Exchanger（两方栅栏）

# 第一部分小结

1. 可变状态是至关重要的。所有的并发问题都可以归纳为如何协调对并发状态的访问。可变状态越少，就越容易确保线程安全性。
2. 尽量将域声明为final类型，除非需要它们是可变的。
3. 不可变对象一定是线程安全的。它们极大降低并发编程的复杂性。更为简单和安全，可以任意共享而无须加锁或保护性复制等机制。
4. 封装有助于管理复杂性。
5. 用锁来保护每个可变变量。
6. 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。
7. 在执行复合操作期间，要持有锁。
8. 如果从多个线程中访问同一个可变变量时没有同步机制，那么程序会出现问题。
9. 不要故作聪明推断出不需要使用同步。
10. 在设计过程中考虑线程安全，或者在文档中明确指出他不是线程安全的。
11. 将同步策略文档化。