**原文：**

**Thread Safety**

Perhaps surprisingly, concurrent programming isn’t so much about threads or locks, any more than civil engineering is about rivets and I-beams. Of course, building bridges that don’t fall down requires the correct use of a lot of rivets and I-beams, just as building concurrent programs require the correct use of threads and locks. But these are just mechanisms—means to an end. Writing thread-safe code is, at its core, about managing access to state, and in particular to shared, mutable state.

Informally, an object’s state is its data, stored in state variables such as instance or static fields. An object’s state may include fields from other, dependent objects; a HashMap’s state is partially stored in the HashMap object itself, but also in many Map.Entry objects. An object’s state encompasses any data that can affect its externally visible behavior.

By shared, we mean that a variable could be accessed by multiple threads; by mutable, we mean that its value could change during its lifetime. We may talk about thread safety as if it were about code, but what we are really trying to do is protect data from uncontrolled concurrent access.

Whether an object needs to be thread-safe depends on whether it will be accessed from multiple threads. This is a property of how the object is used in a program, not what it does. Making an object thread-safe requires using synchronization to coordinate access to its mutable state; failing to do so could result in data corruption and other undesirable consequences.

Whenever more than one thread accesses a given state variable, and one of them might write to it, they all must coordinate their access to it using synchronization. The primary mechanism for synchronization in Java is the synchronized keyword, which provides exclusive locking, but the term “synchronization” also includes the use of volatile variables, explicit locks, and atomic variables.

You should avoid the temptation to think that there are “special” situations in which this rule does not apply. A program that omits needed synchronization might appear to work, passing its tests and performing well for years, but it is still broken and may fail at any moment.

If multiple threads access the same mutable state variable without appropriate synchronization, your program is broken. There are three ways to fix it: Don’t share the state variable across threads; Make the state variable immutable; Use synchronization whenever accessing the state variable

If you haven’t considered concurrent access in your class design, some of these approaches can require significant design modifications, so fixing the problem might not be as trivial as this advice makes it sound. It is far easier to design a class to be thread-safe than to retrofit it for thread safety later.

In a large program, identifying whether multiple threads might access a given variable can be complicated. Fortunately, the same object-oriented techniques that help you write well-organized, maintainable classes—such as encapsulation and data hiding—can also help you create thread-safe classes. The less code that has access to a particular variable, the easier it is to ensure that all of it uses the proper synchronization, and the easier it is to reason about the conditions under which a given variable might be accessed. The Java language doesn’t force you to encapsulate state—it is perfectly allowable to store state in public fields (even public static fields) or publish a reference to an otherwise internal object—but the better encapsulated your program state, the easier it is to make your program thread-safe and to help maintainers keep it that way.

When designing thread-safe classes, good object-oriented techniques— encapsulation, immutability, and clear specification of invariants—are your best friends.

There will be times when good object-oriented design techniques are at odds with real-world requirements; it may be necessary in these cases to compromise the rules of good design for the sake of performance or for the sake of backward compatibility with legacy code. Sometimes abstraction and encapsulation are at odds with performance—although not nearly as often as many developers believe—but it is always a good practice first to make your code right, and then make it fast. Even then, pursue optimization only if your performance measurements and requirements tell you that you must, and if those same measurements tell you that your optimizations actually made a difference under realistic conditions.1

If you decide that you simply must break encapsulation, all is not lost. It is still possible to make your program thread-safe, it is just a lot harder. Moreover, the thread safety of your program will be more fragile, increasing not only development cost and risk but maintenance cost and risk as well. Chapter 4 characterizes the conditions under which it is safe to relax encapsulation of state variables.

We’ve used the terms “thread-safe class” and “thread-safe program” nearly interchangeably thus far. Is a thread-safe program one that is constructed entirely of thread-safe classes? Not necessarily—a program that consists entirely of thread-safe classes may not be thread-safe, and a thread-safe program may contain classes that are not thread-safe. The issues surrounding the composition of thread-safe classes are also taken up in Chapter 4. In any case, the concept of a thread-safe class makes sense only if the class encapsulates its own state. Thread safety may be a term that is applied to code, but it is about state, and it can only be applied to the entire body of code that encapsulates its state, which may be an object or an entire program.

Defining thread safety is surprisingly tricky. The more formal attempts are so complicated as to offer little practical guidance or intuitive understanding, and the rest are informal descriptions that can seem downright circular. A quick Google search turns up numerous “definitions” like these: can be called from multiple program threads without unwanted interactions between the threads. may be called by more than one thread at a time without requiring any other action on the caller’s part.

Given definitions like these, it’s no wonder we find thread safety confusing! They sound suspiciously like “a class is thread-safe if it can be used safely from multiple threads.” You can’t really argue with such a statement, but it doesn’t offer much practical help either. How do we tell a thread-safe class from an unsafe one? What do we even mean by “safe”?

At the heart of any reasonable definition of thread safety is the concept of correctness. If our definition of thread safety is fuzzy, it is because we lack a clear definition of correctness.

Correctness means that a class conforms to its specification. A good specification defines invariants constraining an object’s state and postconditions describing the effects of its operations. Since we often don’t write adequate specifications for our classes, how can we possibly know they are correct? We can’t, but that doesn’t stop us from using them anyway once we’ve convinced ourselves that “the code works”. This “code confidence” is about as close as many of us get to correctness, so let’s just assume that single-threaded correctness is something that “we know it when we see it”. Having optimistically defined “correctness” as something that can be recognized, we can now define thread safety in a somewhat less circular way: a class is thread-safe when it continues to behave correctly when accessed from multiple threads.

A class is thread-safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

Since any single-threaded program is also a valid multithreaded program, it cannot be thread-safe if it is not even correct in a single-threaded environment.2 If an object is correctly implemented, no sequence of operations—calls to public methods and reads or writes of public fields—should be able to violate any of its invariants or postconditions. No set of operations performed sequentially or concurrently on instances of a thread-safe class can cause an instance to be in an invalid state.

Thread-safe classes encapsulate any needed synchronization so that clients need not provide their own**.**

In Chapter 1, we listed a number of frameworks that create threads and call your components from those threads, leaving you with the responsibility of making your components thread-safe. Very often, thread-safety requirements stem not from a decision to use threads directly but from a decision to use a facility like the Servlets framework. We’re going to develop a simple example—a servlet-based factorization service—and slowly extend it to add features while preserving its thread safety.

Listing 2.1 shows our simple factorization servlet. It unpacks the number to be factored from the servlet request, factors it, and packages the results into the servlet response.

StatelessFactorizer is, like most servlets, stateless: it has no fields and references no fields from other classes. The transient state for a particular computation exists solely in local variables that are stored on the thread’s stack and are accessible only to the executing thread. One thread accessing a StatelessFactorizer cannot influence the result of another thread accessing the same StatelessFactorizer; because the two threads do not share state, it is as if they were accessing different instances. Since the actions of a thread accessing a stateless object cannot affect the correctness of operations in other threads, stateless objects are threadsafe.

Stateless objects are always thread-safe.

The fact that most servlets can be implemented with no state greatly reduces the burden of making servlets thread-safe. It is only when servlets want to remember things from one request to another that the thread safety requirement becomes an issue.

**Atomicity**

What happens when we add one element of state to what was a stateless object? Suppose we want to add a “hit counter” that measures the number of requests processed. The obvious approach is to add a long field to the servlet and increment it on each request, as shown in UnsafeCountingFactorizer in Listing 2.2.

Unfortunately, UnsafeCountingFactorizer is not thread-safe, even though it would work just fine in a single-threaded environment. Just like UnsafeSequence on page 6, it is susceptible to lost updates. While the increment operation, ++count, may look like a single action because of its compact syntax, it is not atomic, which means that it does not execute as a single, indivisible operation. Instead, it is a shorthand for a sequence of three discrete operations: fetch the current value, add one to it, and write the new value back. This is an example of a read-modify-write operation, in which the resulting state is derived from the previous state.

Figure 1.1 on page 6 shows what can happen if two threads try to increment a counter simultaneously without synchronization. If the counter is initially 9, with some unlucky timing each thread could read the value, see that it is 9, add one to it, and each set the counter to 10. This is clearly not what is supposed to happen; an increment got lost along the way, and the hit counter is now permanently off by one.

You might think that having a slightly inaccurate count of hits in a web-based service is an acceptable loss of accuracy, and sometimes it is. But if the counter is being used to generate sequences or unique object identifiers, returning the same value from multiple invocations could cause serious data integrity problems.3 The possibility of incorrect results in the presence of unlucky timing is so important in concurrent programming that it has a name: a race condition.

UnsafeCountingFactorizer has several race conditions that make its results unreliable. A race condition occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime; in other words, when getting the right answer relies on lucky timing.4 The most common type of race condition is check-then-act, where a potentially stale observation is used to make a decision on what to do next.

We often encounter race conditions in real life. Let’s say you planned to meet a friend at noon at the Starbucks on University Avenue. But when you get there, you realize there are two Starbucks on University Avenue, and you’re not sure which one you agreed to meet at. At 12:10, you don’t see your friend at Starbucks A, so you walk over to Starbucks B to see if he’s there, but he isn’t there either. There are a few possibilities: your friend is late and not at either Starbucks; your friend arrived at Starbucks A after you left; or your friend was at Starbucks B, but went to look for you, and is now enroute to Starbucks A. Let’s assume the worst and say it was the last possibility. Now it’s 12:15, you’ve both been to both Starbucks, and you’re both wondering if you’ve been stood up. What do you do now? Go back to the other Starbucks? How many times are you going to go back and forth? Unless you have agreed on a protocol, you could both spend the day walking up and down University Avenue, frustrated and undercaffeinated.

The problem with the “I’ll just nip up the street and see if he’s at the other one” approach is that while you’re walking up the street, your friend might have moved. You look around Starbucks A, observe “he’s not here”, and go looking for him. And you can do the same for Starbucks B, but not at the same time. It takes a few minutes to walk up the street, and during those few minutes, the state of the system may have changed.

The Starbucks example illustrates a race condition because reaching the desired outcome (meeting your friend) depends on the relative timing of events (when each of you arrives at one Starbucks or the other, how long you wait there before switching, etc). The observation that he is not at Starbucks A becomes potentially invalid as soon as you walk out the front door; he could have come in through the back door and you wouldn’t know. It is this invalidation of observations that characterizes most race conditions—using a potentially stale observation to make a decision or perform a computation. This type of race condition is called check-then-act: you observe something to be true (file X doesn’t exist) and then take action based on that observation (create X); but in fact the observation could have become invalid between the time you observed it and the time you acted on it (someone else created X in the meantime), causing a problem (unexpected exception, overwritten data, file corruption).

A common idiom that uses check-then-act is lazy initialization. The goal of lazy initialization is to defer initializing an object until it is actually needed while at the same time ensuring that it is initialized only once. LazyInitRace in Listing 2.3 illustrates the lazy initialization idiom. The getInstance method first checks whether the ExpensiveObject has already been initialized, in which case it returns the existing instance; otherwise it creates a new instance and returns it after retaining a reference to it so that future invocations can avoid the more expensive code path.

LazyInitRace has race conditions that can undermine its correctness. Say that threads A and B execute getInstance at the same time. A sees that instance is null, and instantiates a new ExpensiveObject. B also checks if instance is null. Whether instance is null at this point depends unpredictably on timing, including the vagaries of scheduling and how long A takes to instantiate the ExpensiveObject and set the instance field. If instance is null when B examines it, the two callers to getInstance may receive two different results, even though getInstance is always supposed to return the same instance.

The hit-counting operation in UnsafeCountingFactorizer has another sort of race condition. Read-modify-write operations, like incrementing a counter, define a transformation of an object’s state in terms of its previous state. To increment a counter, you have to know its previous value and make sure no one else changes or uses that value while you are in mid-update.

Like most concurrency errors, race conditions don’t always result in failure: some unlucky timing is also required. But race conditions can cause serious problems. If LazyInitRace is used to instantiate an application-wide registry, having it return different instances from multiple invocations could cause registrations to be lost or multiple activities to have inconsistent views of the set of registered objects. If UnsafeSequence is used to generate entity identifiers in a persistence framework, two distinct objects could end up with the same ID, violating identity integrity constraints.

Both LazyInitRace and UnsafeCountingFactorizer contained a sequence of operations that needed to be atomic, or indivisible, relative to other operations on the same state. To avoid race conditions, there must be a way to prevent other threads from using a variable while we’re in the middle of modifying it, so we can ensure that other threads can observe or modify the state only before we start or after we finish, but not in the middle.

Operations A and B are atomic with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either all of B has executed or none of it has. An atomic operation is one that is atomic with respect to all operations, including itself, that operate on the same state.

If the increment operation in UnsafeSequence were atomic, the race condition illustrated in Figure 1.1 on page 6 could not occur, and each execution of the increment operation would have the desired effect of incrementing the counter by exactly one. To ensure thread safety, check-then-act operations (like lazy initialization) and read-modify-write operations (like increment) must always be atomic. We refer collectively to check-then-act and read-modify-write sequences as compound actions: sequences of operations that must be executed atomically in order to remain thread-safe. In the next section, we’ll consider locking, Java’s builtin mechanism for ensuring atomicity. For now, we’re going to fix the problem another way, by using an existing thread-safe class, as shown in CountingFactorizer in Listing 2.4.

The java.util.concurrent.atomic package contains atomic variable classes for effecting atomic state transitions on numbers and object references. By replacing the long counter with an AtomicLong, we ensure that all actions that access the counter state are atomic.5 Because the state of the servlet is the state of the counter and the counter is thread-safe, our servlet is once again thread-safe.

We were able to add a counter to our factoring servlet and maintain thread safety by using an existing thread-safe class to manage the counter state, AtomicLong. When a single element of state is added to a stateless class, the resulting class will be thread-safe if the state is entirely managed by a thread-safe object. But, as we’ll see in the next section, going from one state variable to more than one is not necessarily as simple as going from zero to one.

Where practical, use existing thread-safe objects, like AtomicLong, to manage your class’s state. It is simpler to reason about the possible states and state transitions for existing thread-safe objects than it is for arbitrary state variables, and this makes it easier to maintain and verify thread safety.