THREAD SAFETY IN RUBY AND RUBY ON RAILS

An overview of the concepts and best practices for ensuring thread-safe code in Ruby and Ruby on Rails applications.

BY KONSTANTIN GREDESKOUL | FEBRUARY 2025

INTRODUCTION TO THREAD SAFETY

WHAT IS THREAD SAFETY?

Thread safety refers to the ability of a computer program or application to execute correctly and predictably when accessed by multiple threads or fibers at the same time.

IMPORTANCE IN RUBY AND RUBY ON RAILS

In Ruby and Ruby on Rails applications, thread safety is crucial to ensure proper utilization of available CPU cores, given that Ruby's GIL prevents more than one operation on a given CPU core at any given time.

RACE CONDITIONS

Shared resources, such as variables or objects, can lead to race conditions when accessed by multiple threads, resulting in data inconsistency or corruption. For instance, the infamous ||= operator is not thread-safe.

SYNCHRONIZATION TECHNIQUES

Ruby offers a class Mutex to provide synchronization and to help manage access to shared resources, ensuring thread safety.

CONCURRENCY PATTERNS IN RUBY

Ruby developers can leverage concurrency patterns, like the Reactor Pattern and the Actor Model, to design thread-safe applications that handle concurrent requests and tasks effectively.

THREAD-SAFE DATA STRUCTURES

There is only one provided thread-safe data structure in Ruby: the class **Queue**, and its subclass **SizedQueue**. The latter one will block on **push()** if the size is at the limit.

IMPORTANT CONCURRENCY NOTES

GIL & MRI RUBY

In MRI there is a global lock preventing true parallel Ruby code execution. However, context switching (especially when using sleep, I/O, or Thread.pass) can still cause unexpected scheduling that leads to race conditions in non-atomic read–modify–write sequences.

JRUBY OR RUBINIUS

When running on JRuby or Rubinius, there is no GIL and so real parallelism across multiple cores can occur, making race conditions even more likely to appear.

TEST CONCURRENT CODE

Any test that relies on race conditions can be somewhat nondeterministic. You might see tests pass or fail sporadically. In real-world scenarios, you'll want to employ robust concurrency testing techniques (stress tests, repeated runs, etc.) and properly use synchronization constructs like Mutex, Queue, or thread-safe data structures.

KEY POINTS

— Ruby MRI uses a Global Interpreter Lock (GIL)
— Only one thread can execute Ruby code at a time
— GIL doesn't protect against all thread safety issues
— I/O operations can run in parallel
— C extensions can bypass the GIL

RUBY SINGLETON CLASSES

There are two meanings of a "singleton class" in Ruby:

1. It's a class that includes Singleton module

2. It's unique features of Ruby language that allows adding behavior to a single instance of a class (an object), the entire class, or a module.

To access object's singleton class, open the object with **instance_eval** {} and then access its singleton class with **class << self; end**

This is particularly useful when you need to manage global state or provide a centralized access point for shared resources, such as sub-classes of a given superclass.

However, it's important to consider the thread safety implications of using singleton classes, as they can introduce potential issues if not implemented correctly.

CLASS / MODULE INSTANCE VARIABLES

```ruby
# frozen_string_literal: true

module ThreadUnsafe
  # @description A simple counter that can be incremented by multiple threads.
  # @note This class is not thread-safe, unless increment(safely: true) is used.
  module Counter
    class << self
      attr_reader :count

      def reset
        with_mutex { @count = 0 }
      end

      # This increment method is deliberately not using any synchronization,
      # and uses a read-modify-write approach that can be interrupted by other threads.
      def increment(safely: false)
        if safely
          with_mutex { unsafe_increment }
        else
          unsafe_increment
        end
      end

      private

      def with_mutex(&)
        mutex.synchronize(&)
      end

      def unsafe_increment
        current = count || 0
        # Force a tiny pause to encourage context switches
        # and make race conditions more likely to appear.
        sleep(0.000001) if rand(100) < 50
        @count = current + 1
      end

      def mutex
        @mutex ||= Mutex.new
      end
    end
  end
end
```

CLASS-LEVEL INSTANCE VARIABLES

WHAT ARE CLASS-LEVEL INSTANCE VARIABLES?

Class-level instance variables are variables that are shared across all instances of a class. They are defined at the class level and can be accessed and modified by any instance of the class.

THREAD SAFETY CONSIDERATIONS

When working with class-level instance variables, developers must be mindful of thread safety. If multiple threads access and modify the same class-level instance variable concurrently, it can lead to race conditions and unexpected behavior.

SYNCHRONIZATION TECHNIQUES

To ensure thread safety, developers can use synchronization techniques such as Mutex or Queue to control access to class-level instance variables. This ensures that only one thread can access the variable at a time, preventing race conditions.

IMMUTABLE CLASS-LEVEL INSTANCE VARIABLES

One way to avoid thread safety issues is to make class-level instance variables immutable. Immutable variables cannot be modified after they are created, which eliminates the need for synchronization and simplifies the code. In Ruby we can "freeze" the value to make it immutable (however, it's also possible to unfreeze it).

THREAD-LOCAL STORAGE

Another approach is to use thread-local storage, which provides a way to associate data with a specific thread. This allows each thread to have its own copy of the class-level instance variable, avoiding the need for synchronization.

Thread Local storage is accessed via eg **Thread.current[key] = value**

CLASS VARIABLES

```ruby
# frozen_string_literal: true

module ThreadUnsafe
  # @description A class-level variable @@count can be incremented by multiple threads.
  # @note This class is not thread-safe, unless increment(safely: true) is used.
  class ClassCounter
    @@count = 0

    def reset
      with_mutex { @@count = 0 }
    end

    def count
      @@count
    end

    # This increment method is deliberately not using any synchronization,
    # and uses a read-modify-write approach that can be interrupted by other threads.
    def increment(safely: false)
      if safely
        with_mutex { unsafe_increment }
      else
        unsafe_increment
      end
    end

    private

    def with_mutex(&)
      mutex.synchronize(&)
    end

    def unsafe_increment
      current = @@count || 0
      # Force a tiny pause to encourage context switches
      # and make race conditions more likely to appear.
      sleep(0.000001) if rand(100) < 50
      @@count = current + 1
    end

    def mutex
      @mutex ||= Mutex.new
    end
  end
end
```

CLASS VARIABLES

WHAT ARE CLASS VARIABLES?

Class variables are variables that are shared among all instances of a class as well as sub-classes. They are defined at the class level and can be accessed by both the class and its instances.

THREAD SAFETY CHALLENGES

Class variables can pose thread safety challenges, as multiple threads may attempt to access and modify the same class variable simultaneously, leading to race conditions and potential data corruption.

SHARED STATE

When multiple threads access a shared class variable, they are operating on a shared state. Proper synchronization mechanisms are required to ensure thread safety and prevent race conditions.

IMMUTABLE CLASS VARIABLES

Using immutable class variables can help eliminate many thread safety concerns, as immutable objects are inherently thread-safe. However, this approach may not always be feasible, and other synchronization techniques may be necessary.

DON'T USE CLASS VARIABLES

Key reasons to avoid class variables:

They create hidden coupling between classes through inheritance

They're not thread-safe by default

They create global state which makes testing difficult

They can cause race conditions in concurrent environments

They make code harder to reason about and maintain

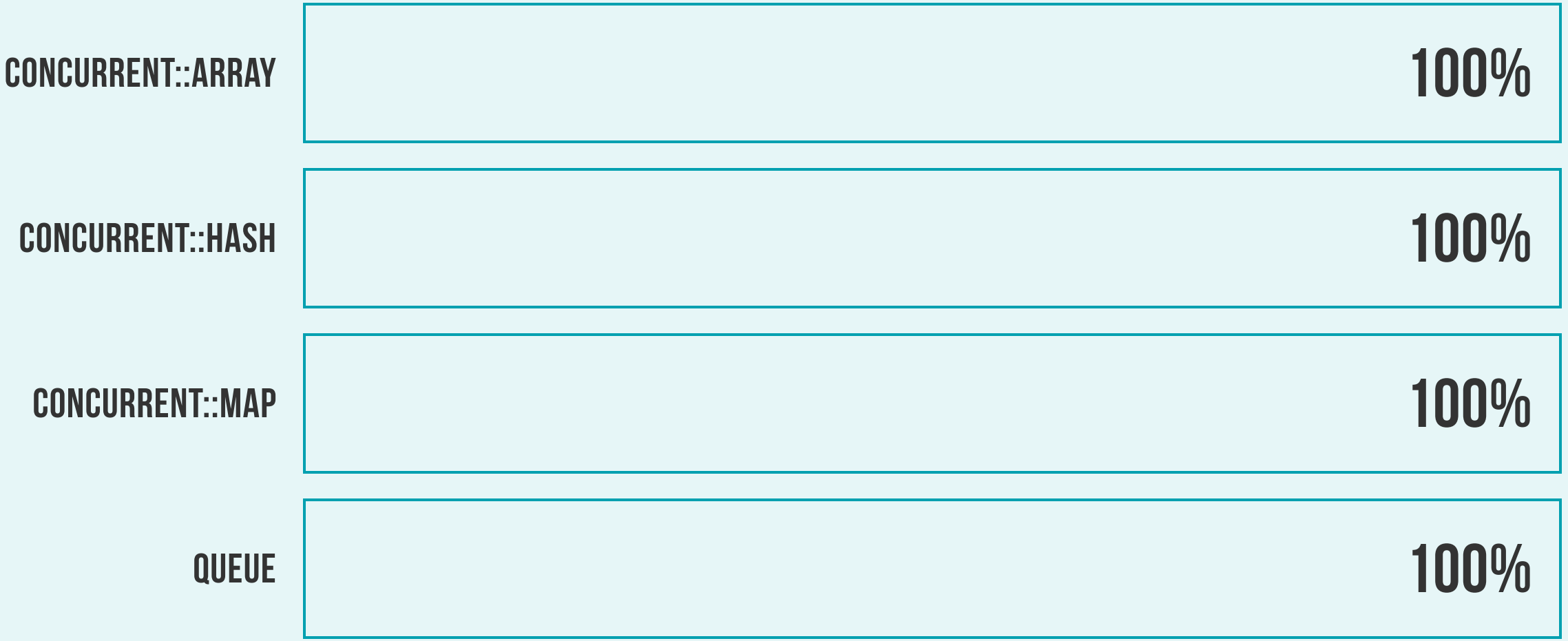
They can cause memory leaks if not properly managed

GLOBAL VARIABLES

- POTENTIAL FOR UNINTENDED MODIFICATIONS
- LACK OF ENCAPSULATION
- DIFFICULTY IN DEBUGGING
- INCREASED LIKELIHOOD OF CONCURRENCY ISSUES

THREAD-SAFE DATA STRUCTURES IN RUBY

Percentage of thread-safe operations supported



PROTECTING DATA FROM PARALLEL WRITE ACCESS

MUTEX LOCKS

Mutex locks (mutual exclusion) allow only one thread to access a critical section of code at a time, preventing race conditions and ensuring data integrity during concurrent write operations.

Eg.
Mutex.new.synchronize { ... }

SEMAPHORES

Ruby doesn't have built-in semaphores, but the gem **concurrent-ruby** gem does.

Concurrent::Semaphore is a signaling mechanisms to control access to a shared resources by multiple threads, allowing for more fine-grained concurrency control compared to mutex locks.

READ-WRITE LOCKS

Read-write locks allow multiple threads to read from a shared resource simultaneously, but restrict write access to only one thread at a time, providing a balance between concurrency and data protection.

For example, built-in Queue class supports this use-case, but it's difficult to enforce single-writer condition.

ATOMIC OPERATIONS

Atomic operations provide hardware-level guarantees for performing indivisible read-modify-write operations, ensuring data consistency without the need for explicit locking mechanisms.

Concurrent Ruby offers Concurrent::Atom class for this purpose and several derivatives, such as **AtomicFixnum** and **AtomicBoolean**.

TRANSACTIONAL MEMORY

Transactional memory is a concurrency control mechanism that allows multiple threads to perform a series of memory accesses in an atomic, isolated, and consistent manner, similar to database transactions.

While Ruby doesn't support STM (software transactional memory) there is a proposal.

BEST PRACTICES AND RECOMMENDATIONS

UTILIZE CONCURRENT RUBY GEMS

Leverage the Concurrent Ruby gem to manage thread-safe data structures and synchronization primitives, ensuring thread safety in your application.

AVOID SHARED MUTABLE STATE

Minimize the use of shared mutable state across threads, as this can lead to race conditions and other concurrency issues. Prefer immutable data structures or use local variables within thread-safe methods.

IMPLEMENT MUTEX LOCKING

Use Mutex locking to protect critical sections of your code that access shared resources. This ensures that only one thread can execute the critical section at a time, preventing race conditions.

UTILIZE REACTIVE PROGRAMMING

Consider using a reactive or actor-driven programming approach, such as the Celluloid gem, to handle concurrency and asynchronous tasks in a more declarative and thread-safe manner.

MONITOR THREAD EXECUTION

Regularly monitor and profile your application to identify potential thread safety issues, such as deadlocks, livelocks, or starvation. Use tools like rbspy or TracePoint to aid in this process.

DEMO TIME

THANK YOU!