DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.

Threading and Code Execution in Rails

After reading this guide, you will know:

- What code Rails will automatically execute concurrently
- How to integrate manual concurrency with Rails internals
- How to wrap all application code
- How to affect application reloading

Automatic Concurrency

Rails automatically allows various operations to be performed at the same time.

When using a threaded web server, such as the default Puma, multiple HTTP requests will be served simultaneously, with each request provided its own controller instance.

Threaded Active Job adapters, including the built-in Async, will likewise execute several jobs at the same time. Action Cable channels are managed this way too.

These mechanisms all involve multiple threads, each managing work for a unique instance of some object (controller, job, channel), while sharing the global process space (such as classes and their configurations, and global variables). As long as your code doesn't modify any of those shared things, it can mostly ignore that other threads exist.

The rest of this guide describes the mechanisms Rails uses to make it "mostly ignorable", and how extensions and applications with special needs can use them.

Executor

The Rails Executor separates application code from framework code: any time the framework invokes code you've written in your application, it will be wrapped by the Executor.

The Executor consists of two callbacks: `to_run` and `to_complete`. The Run callback is called before the application code, and the Complete callback is called after.

Default callbacks

In a default Rails application, the Executor callbacks are used to:

- track which threads are in safe positions for autoloading and reloading
- enable and disable the Active Record query cache

- return acquired Active Record connections to the pool
- constrain internal cache lifetimes

Prior to Rails 5.0, some of these were handled by separate Rack middleware classes (such as `ActiveRecord::ConnectionAdapters::ConnectionManagement`), or directly wrapping code with methods like `ActiveRecord::Base.connection_pool.with_connection`. The Executor replaces these with a single more abstract interface.

Wrapping application code

If you're writing a library or component that will invoke application code, you should wrap it with a call to the executor:

```ruby
Rails.application.executor.wrap do
  # call application code here
end
```

TIP: If you repeatedly invoke application code from a long-running process, you may want to wrap using the Reloader instead.

Each thread should be wrapped before it runs application code, so if your application manually delegates work to other threads, such as via `Thread.new` or Concurrent Ruby features that use thread pools, you should immediately wrap the block:

```ruby
Thread.new do
  Rails.application.executor.wrap do
    # your code here
  end
end
```

NOTE: Concurrent Ruby uses a `ThreadPoolExecutor`, which it sometimes configures with an `executor` option. Despite the name, it is unrelated.

The Executor is safely re-entrant; if it is already active on the current thread, `wrap` is a no-op.

If it's impractical to wrap the application code in a block (for example, the Rack API makes this problematic), you can also use the `run!` / `complete!` pair:

```ruby
Thread.new do
  execution_context = Rails.application.executor.run!
  # your code here
ensure
  execution_context.complete! if execution_context
end
```

Concurrency

The Executor will put the current thread into `running` mode in the Load Interlock. This operation will block temporarily if another thread is currently either autoloading a constant or unloading/reloading the application.

Reloader

Like the Executor, the Reloader also wraps application code. If the Executor is not already active on the current thread, the Reloader will invoke it for you, so you only need to call one. This also guarantees that everything the Reloader does, including all its callback invocations, occurs wrapped inside the Executor.

```
Rails.application.reloader.wrap do
  # call application code here
end
```

The Reloader is only suitable where a long-running framework-level process repeatedly calls into application code, such as for a web server or job queue. Rails automatically wraps web requests and Active Job workers, so you'll rarely need to invoke the Reloader for yourself. Always consider whether the Executor is a better fit for your use case.

Callbacks

Before entering the wrapped block, the Reloader will check whether the running application needs to be reloaded – for example, because a model's source file has been modified. If it determines a reload is required, it will wait until it's safe, and then do so, before continuing. When the application is configured to always reload regardless of whether any changes are detected, the reload is instead performed at the end of the block.

The Reloader also provides `to_run` and `to_complete` callbacks; they are invoked at the same points as those of the Executor, but only when the current execution has initiated an application reload. When no reload is deemed necessary, the Reloader will invoke the wrapped block with no other callbacks.

Class Unload

The most significant part of the reloading process is the Class Unload, where all autoloaded classes are removed, ready to be loaded again. This will occur immediately before either the Run or Complete callback, depending on the `reload_classes_only_on_change` setting.

Often, additional reloading actions need to be performed either just before or just after the Class Unload, so the Reloader also provides `before_class_unload` and `after_class_unload` callbacks.

Concurrency

Only long-running "top level" processes should invoke the Reloader, because if it determines a reload is needed, it will block until all other threads have completed any Executor invocations.

If this were to occur in a "child" thread, with a waiting parent inside the Executor, it would cause an unavoidable deadlock: the reload must occur before the child thread is executed, but it cannot be safely performed while the parent thread is mid-execution. Child threads should use the Executor instead.

Framework Behavior

The Rails framework components use these tools to manage their own concurrency needs too.

`ActionDispatch::Executor` and `ActionDispatch::Reloader` are Rack middlewares that wrap requests with a supplied Executor or Reloader, respectively. They are automatically included in the default application stack. The Reloader will ensure any arriving HTTP request is served with a freshly-loaded copy of the application if any code changes have occurred.

Active Job also wraps its job executions with the Reloader, loading the latest code to execute each job as it comes off the queue.

Action Cable uses the Executor instead: because a Cable connection is linked to a specific instance of a class, it's not possible to reload for every arriving WebSocket message. Only the message handler is wrapped, though; a long-running Cable connection does not prevent a reload that's triggered by a new incoming request or job. Instead, Action Cable uses the Reloader's `before_class_unload` callback to disconnect all its connections. When the client automatically reconnects, it will be speaking to the new version of the code.

The above are the entry points to the framework, so they are responsible for ensuring their respective threads are protected, and deciding whether a reload is necessary. Other components only need to use the Executor when they spawn additional threads.

Configuration

The Reloader only checks for file changes when `cache_classes` is false and `reload_classes_only_on_change` is true (which is the default in the `development` environment).

When `cache_classes` is true (in `production`, by default), the Reloader is only a pass-through to the Executor.

The Executor always has important work to do, like database connection management. When `cache_classes` and `eager_load` are both true (`production`), no autoloading or class reloading will occur, so it does not need the Load Interlock. If either of those are false (`development`), then the Executor will use the Load Interlock to ensure constants are only loaded when it is safe.

Load Interlock

The Load Interlock allows autoloading and reloading to be enabled in a multi-threaded runtime environment.

When one thread is performing an autoload by evaluating the class definition from the appropriate file, it is important no other thread encounters a reference to the partially-defined constant.

Similarly, it is only safe to perform an unload/reload when no application code is in mid-execution: after the reload, the `User` constant, for example, may point to a different class. Without this rule, a poorly-timed reload would mean `User.new.class == User`, or even `User == User`, could be false.

Both of these constraints are addressed by the Load Interlock. It keeps track of which threads are currently running application code, loading a class, or unloading autoloaded constants.

Only one thread may load or unload at a time, and to do either, it must wait until no other threads are running application code. If a thread is waiting to perform a load, it doesn't prevent other threads from loading (in fact, they'll cooperate, and each perform their queued load in turn, before all resuming running together).

`permit_concurrent_loads`

The Executor automatically acquires a `running` lock for the duration of its block, and autoload knows when to upgrade to a `load` lock, and switch back to `running` again afterwards.

Other blocking operations performed inside the Executor block (which includes all application code), however, can needlessly retain the `running` lock. If another thread encounters a constant it must autoload, this can cause a deadlock.

For example, assuming `User` is not yet loaded, the following will deadlock:

```ruby
Rails.application.executor.wrap do
  th = Thread.new do
    Rails.application.executor.wrap do
      User # inner thread waits here; it cannot load
           # User while another thread is running
    end
  end

  th.join # outer thread waits here, holding 'running' lock
end
```

To prevent this deadlock, the outer thread can `permit_concurrent_loads`. By calling this method, the thread guarantees it will not dereference any possibly-autoloaded constant inside the supplied block. The safest way to meet that promise is to put it as close as possible to the blocking call:

```
Rails.application.executor.wrap do
  th = Thread.new do
    Rails.application.executor.wrap do
      User # inner thread can acquire the 'load' lock,
           # load User, and continue
    end
  end

  ActiveSupport::Dependencies.interlock.permit_concurrent_loads do
    th.join # outer thread waits here, but has no lock
  end
end
```

Another example, using Concurrent Ruby:

```
Rails.application.executor.wrap do
  futures = 3.times.collect do |i|
    Concurrent::Promises.future do
      Rails.application.executor.wrap do
        # do work here
      end
    end
  end

  values = ActiveSupport::Dependencies.interlock.permit_concurrent_loads do
    futures.collect(&:value)
  end
end
```

ActionDispatch::DebugLocks

If your application is deadlocking and you think the Load Interlock may be involved, you can temporarily add the ActionDispatch::DebugLocks middleware to `config/application.rb`:

```
config.middleware.insert_before Rack::Sendfile,
                                  ActionDispatch::DebugLocks
```

If you then restart the application and re-trigger the deadlock condition, `/rails/locks` will show a summary of all threads currently known to the interlock, which lock level they are holding or awaiting, and their current backtrace.

Generally a deadlock will be caused by the interlock conflicting with some other external lock or blocking I/O call. Once you find it, you can wrap it with `permit_concurrent_loads`.