Add block or filtered forms of Kernel#caller to allow early bail-out

There are many libraries that use caller or caller_locations to gather stack information for logging or instrumentation. These methods generate an array of informational stack frames based on the current call stack.

Both methods accept parameters for level (skip some number of Ruby frames) and length (only return this many frames). However many use cases are unable to provide one or both of these.

Instrumentation uses, for example, may need to skip an unknown number of frames at the top of the trace, such as to dig out of rspec plumbing or active_record internals and report the first line of user code. In such cases, the typical pattern is to simply request all frames and then filter out the one that is desired.

This leads to a great deal of wasted work gathering those frames and constructing objects to carry them to the user. On optimizing runtimes like JRuby and TruffleRuby, it can have a tremendous impact on performance, since each frame has a much higher cost than on CRuby.

I propose that we need a new form of caller that takes a block for processing each element.

```ruby
def find_matching_frame(regex)
  caller do |frame|
    return frame if frame.file =~ regex
  end
end
```

An alternative API would be to allow passing a query object as a keyword argument, avoiding the block dispatch by performing the match internally:

```ruby
def find_matching_frame(regex)
  caller(file: regex)
end
```

This API would provide a middle ground between explicitly specifying a maximum number of stack frames and asking for all frames. Most common, hot-path uses of caller could be replaced by these forms, reducing overhead on all Ruby implementations and drastically reducing it where stack traces are expensive.
inside a callback (e.g., Java 9's StackWalker, TruffleRuntime#iterateFrames), and after every block call we can easily find out where we are on the 
stack and keep iterating (what’s typically not allowed is get some stack entry and return it without materializing it, not an issue here).

I think if caller/caller_locations is given a block, then it should return nil like e.g. Array#zip.
That way we avoid the cost of appending to an array if it’s not used, and it is possible to escape analyze Thread::Backtrace::Location or String objects 
passed to the block (otherwise they would always escape to that array).
This is important for the performance of this new feature.

#3 - 11/09/2021 12:36 PM - Eregon (Benoit Daloze)
The non-local return like return frame is likely to be rather costly in this case unfortunately.
Stack walking typically can’t be JITed since it accesses a lot of VM stuff, and so the exception/jump for the return (or break) is quite unoptimized (if a 
JIT sees both from where and to where it jumps it can optimize well, but not possible here).

I think caller { ... } should return the first truthy value (not nil not false) to avoid that cost:

```ruby
def find_matching_frame(regex)
  caller do |frame|
    frame if frame.file =~ regex
  end
end
```
or more explicitly

```ruby
def find_matching_frame(regex)
  caller do |frame|
    if frame.file =~ regex
      frame
    else
      nil # keep searching
    end
  end
end
```

If we iterated the whole stack and none of the block calls returned a truthy value, then nil would be returned.

People can still use an explicit return/break if they want but we can document that’s typically less efficient for this method.

#4 - 11/09/2021 01:28 PM - zverok (Victor Shepelev)
Can't the specialized Enumerator be used for that? The downside is it probably would be inconsistent with the existing protocol of caller, so maybe 
the new method would be necessary, but, roughly speaking:

```ruby
lazy_caller.find { _1.file.match?(regex) }
# or
caller(lazy: true).find { _1.file.match?(regex) }
```

...would be consistent with what we want (?) users to expect of Ruby, instead of "just this method's custom protocol".

#5 - 11/09/2021 04:32 PM - Eregon (Benoit Daloze)

zverok (Victor Shepelev) No, that can't work because then in your example the actual place where the stack is queried is e.g. inside #find which e.g. 
adds an extra entry on the stack.
Also in general we can't use Enumerable/Enumerator here, the passed caller information is only meaningful at the place caller was called.

We can have arbitrary calls from the block of caller, that's fine, but we can't ever return "something" that exposes the stack where caller was called, 
because that something would then need to eagerly capture the stack and miss the whole point of this feature.
Or that "something" would iterate the stack at the place it's actually used (not where lazy_caller was called), and that would just be confusing.

#6 - 11/18/2021 06:57 AM - matz (Yukihiro Matsumoto)
The behavior of the caller method with a block is not obvious. I understand the needs, should we add a method with a new name?

Matz.

#7 - 11/18/2021 09:47 AM - Eregon (Benoit Daloze)

It's an API to find one of the callers, or to find some information (e.g., file, line, method name) from ones of the callers.

So maybe find_caller { ... } and find_caller_locations { ... }?
An advantage of those names is they are clear that the block would return on the first truthy value (similar to Enumerable#find), which matches the 
optimal semantics I described above.

I think using the existing method names (caller/caller_locations) and enhancing their behavior would be fine too, and has the advantage to not 
introduce additional methods (in Kernel).
It's an API to find one of the callers, or to find some information (e.g., file, line, method name) from ones of the callers.

I disagree. The usage is not limited to find only one caller. You can use it to find multiple callers, such as the first two callers meeting a criteria. I think limiting it to a single caller would make a common case easier, but would make other cases more difficult.

So maybe find_caller {...} and find_caller_locations {...}? An advantage of those names is they are clear that the block would return on the first truthy value (similar to Enumerable#find), which matches the optimal semantics I described above.

As I think it's best to not restrict to only returning a single frame, I propose the simpler each_caller/each_caller_location. As Eregon (Benoit Daloze) mentioned, we can have these return nil to avoid the issue of creating an array up front. That would simplify the implementation.

If I understand correctly, one of the reasons Eregon (Benoit Daloze) is proposing find_caller over each_caller is optimization. If that is correct, I think we should only go with find_caller if we have benchmarked it and it has a significant performance advantage over each_caller. Without benchmarks, I'm not convinced that avoiding the return/break will make a significant performance difference, at least on CRuby.

I disagree. The usage is not limited to find only one caller. You can use it to find multiple callers, such as the first two callers meeting a criteria. I think limiting it to a single caller would make a common case easier, but would make other cases more difficult.

It's not really limiting, the block can do anything, including getting two callers and then returning the result, or the user can even use break/return if they find that easier. But it makes it possible for common cases to do something really efficient.

If I understand correctly, one of the reasons Eregon (Benoit Daloze) is proposing find_caller over each_caller is optimization. If that is correct, I think we should only go with find_caller if we have benchmarked it and it has a significant performance advantage over each_caller.

That's correct.

Without benchmarks, I'm not convinced that avoiding the return/break will make a significant performance difference, at least on CRuby.

We should benchmark it. It might not make a difference on current CRuby, but I suspect it does on Ruby implementations with a JIT or with frames which can actually skip being allocated, or for VMs which can't use longjmp() and instead must use some kind of exceptions for non-local returns (the case on JVM).

find_caller is going to find a single entry from call frames. Is it sufficient for OP's demand? Besides that, is it worth adding a global Kernel method?

I doubt both. I am not against adding the feature, but first, we have to make the following clear:

- iteration or search
- place and name of the method

Matz.

find_caller is going to find a single entry from call frames. Is it sufficient for OP's demand?

We should benchmark it. Since the entire point of this feature is to stop iterating once we found what we wanted, imho it makes sense to do that when the block returns true. It's simpler than using break or return, and more consistent with Enumerable patterns in general.
Hopefully headius (Charles Nutter) can answer that. I assume that finding the first matching entry would be the most common use of this method. For cases where you do want to iterate past the first matching entry, you could still get normal iteration in find_caller by having the block return nil/false, and use break to return a different value.

I'm fine with either find_caller or each_caller. Each is implementable in terms of the other. Which we choose is not of great importance to me, but I think it would be useful to offer some way to support this.

Besides that, is it worth adding a global Kernel method?

Maybe not. nobu (Nobuyoshi Nakada) mentioned Thread::Backtrace.find_caller, which is a possibility. Alternatively, it could be a singleton method on Kernel instead of an instance method (Kernel.find_caller without Kernel#find_caller), or a singleton method on Thread.

#13 - 12/11/2021 09:57 AM - headius (Charles Nutter)

Sorry I missed all discussions, this has been a pretty busy autumn.

find_caller is going to find a single entry from call frames. Is it sufficient for OP's demand?

find_caller is one use case, and probably the most common use case. I model my proposal on the JDK 9+ "StackWalker" which simply takes a closure and passes you stack trace elements one at a time. As long as the walk is active you can consume as many or as few frames as you want.

However Java does not have a concept of returning from a closure. The closure always returns to its receiving method, so you have to decide up front what the receiving method will return. In this case, returning a single "found" stack frame would seem to be the most natural option. Supporting the return of both an array of elements and a found search in the same interface feels too confusing (and as Eregon (Benoit Daloze) pointed out in the PR, returning the array is wasteful if you only want to find one element).

I would like to see this feature happen in some way, but most of the use cases I considered will want to process and keep the top X frames, where X is variable at runtime depending on what was called (example: tests and specs that trim our own internals and shorten the call stack; they do not know how many frames to skip so they request all of them just to ultimately keep N < all).

As it stands today, the only way to trim stack frames down to some search-based depth is to get them all and crop it after the fact.

Actually, a thought occurs reading through other comments...

From zverok (Victor Shepelev):

Can't the specialized Enumerator be used for that?

From Eregon (Benoit Daloze):

Also in general we can't use Enumerable/Enumerator here...

This isn't quite correct. We can return a stack trace generator that acts like an Enumerable.

Enumerable is by default internal iteration, which is exactly what we want in this case: we can only iterate downstream from where we have requested a stack trace. The stack frames would be materialized lazily and passed via each to whatever Enumerable methods you used to filter it. If you have a straightforward case, it should still only cost as many frames as you actually want to see.

```ruby
Thread.each_backtrace.drop_while { |frame| test_library_frame?(frame) } .keep_until { |frame| test_library_frame?(frame) }
```

This would perform the hypothetical test suite scrubbing out its own internal frames but keeping those relevant to the test itself. Once the keep_until failed, no more frames would be materialized and the slice of them would be returned.

Anything that turns it into an Array would force it to materialize all unfiltered frames, but filtered frames that don't need to be traversed won't be materialized. In most implementations it would also have to materialize once each_backtrace returns, since we'll start popping important frames at that point.

Of course this code also works with the current Array return value, but we have to populate the whole Array. Better would be a class that keeps its contents opaque until forced to materialize them.

Something like

```ruby
class BacktraceWalker
  include Enumerable
  def initialize(live_trace)
    @live_trace = live_trace
  end

  def each(&block)
    return @live_trace.capture_trace(&block) # but internal so doesn't have to filter itself
  end
end
```

03/11/2022
rescue StopIteration
  return $!.result
end
end

---

#14 - 12/11/2021 06:53 PM - jeremyevans0 (Jeremy Evans)

headius (Charles Nutter) wrote in #note-13:

   I would like to see this feature happen in some way, but most of the use cases I considered will want to process and keep the top X frames, where X is variable at runtime depending on what was called (example: tests and specs that trim our their own internals and shorten the call stack; they do not know how many frames to skip so they request all of them just to ultimately keep N < all).

This makes it sound like you favor each_caller (iteration) over find_caller (search). Can you please confirm which you prefer? I think the expectation is that find_caller would return first string/location where block returns truthy, and each_caller would return nil.

As it stands today, the only way to trim stack frames down to some search-based depth is to get them all and crop it after the fact.

Yes. That's why I think this feature would be useful. We just need to agree on the method name, location, and semantics.

This isn't quite correct. We can return a stack trace generator that acts like an Enumerable.

I'm not sure this is true in CRuby, since we need to walk the VM frame stack at the point of call. I don't think CRuby can support something like the following, unless we make fairly drastic changes:

def foo
  each_caller # returns Enumerator
end

def bar
  foo
end

bar.each{|frame| # iterates

This is because when each is called, we've already lost the first few frames of the stack (the calls to caller, foo, and bar).

headius (Charles Nutter) in order to move forward on this, we need to answer Matz's two questions:

matz (Yukihiro Matsumoto) wrote in #note-10:

   ● iteration or search
   ● place and name of the method

It sounds like you want iteration, and you are recommending with Thread.each_backtrace yielding Thread::Backtrace::Location objects. Is that correct? While your Enumerable example would not work in CRuby, you may be able to do:

    array = []
    Thread.each_backtrace do |frame|
      if test_library_frame?(frame)
        break unless array.empty?
      else
        array << frame
      end
    end

#15 - 12/15/2021 02:56 PM - headius (Charles Nutter)

@jeremyevans0:

It sounds like you want iteration, and you are recommending with Thread.each_backtrace yielding Thread::Backtrace::Location objects. Is that correct?

Yes to iteration. Capturing a single frame is only one use case (and maybe not the most common). We should make it possible to capture multiple.

Thread.each_backtrace yielding Location objects would cover the use cases I am interested in, yes.

While your Enumerable example would not work in CRuby, you may be able to do:
Yes I botched that part of my example. The Enumerable would have to be yielded to a block and used within that block. My example done right would be more like:

```ruby
Kernel.caller_locations do |enum|
    enum.drop_while {|frame| test_library_frame?(frame)}
    .keep_until {|frame| test_library_frame?(frame)}
    .to_a
end
```

If the enum is used outside of this context, it would immediately error or StopIteration as it can no longer materialize lost frames. Some combination of Lazy and eager Enumerator logic could be used here to materialize required frames at the point that caller_locations returns.

Your example, with a simple each_backtrace does cover all the use cases I'm interested in, however. The only thing we lose over my version is the ability to apply other lazy Enumerator filtering in a more fluent style of code. My version would apply more naturally to the existing API, also, and would work the same for caller, backtrace and *_locations APIs.

---

#16 - 12/15/2021 04:28 PM - Dan0042 (Daniel DeLorme)

headius (Charles Nutter) wrote in #note-15:

```ruby
Kernel.caller_locations do |enum|
    enum.drop_while {|frame| test_library_frame?(frame)}
    .keep_until {|frame| test_library_frame?(frame)}
    .to_a
end
```

That looks somewhat complex to implement, but it's such an elegant API, and doesn't even require a new method.

+1

#17 - 01/03/2022 06:16 PM - jeremyevans0 (Jeremy Evans)

headius (Charles Nutter) wrote in #note-15:

```
Yes I botched that part of my example. The Enumerable would have to be yielded to a block and used within that block. My example done right would be more like:

Kernel.caller_locations do |enum|
    enum.drop_while {|frame| test_library_frame?(frame)}
    .keep_until {|frame| test_library_frame?(frame)}
    .to_a
end
```

If the enum is used outside of this context, it would immediately error or StopIteration as it can no longer materialize lost frames. Some combination of Lazy and eager Enumerator logic could be used here to materialize required frames at the point that caller_locations returns.

I think this approach of yielding an enumerable would be much more complex to implement, and it would be challenging to detect misuse and raise. Considering this feature is only needed rarely, I think the cost of the additional complexity is higher than the benefit of the nicer API.

Your example, with a simple each_backtrace does cover all the use cases I'm interested in, however. The only thing we lose over my version is the ability to apply other lazy Enumerator filtering in a more fluent style of code. My version would apply more naturally to the existing API, also, and would work the same for caller, backtrace and *_locations APIs.

Since a simpler each_backtrace that yields caller locations will cover all use cases you are interested in, I think it makes sense to use that approach.

I'll add this ticket to the next developer meeting, so we can hopefully get approval to move forward.

#18 - 01/04/2022 10:59 AM - Eregon (Benoit Daloze)

I also think the approach of yielding an enumerable while smart looks really complicated to implement properly, and I could imagine it might cause issues in some VMs.

Also if I understand correctly, that would mean enum.each would start the stack walking and so the line at which enum.foo is called would matter, and there would need to be some magical filtering to remove caller_locations and the block from the stacktrace and everything until enum.each (there could be extra calls/locks in between), or otherwise those would show up as extra entries which would be highly inconvenient.

And I think .lazy would likely not work as then it'd iterate the backtrace of the lazy Fiber and not the original one (not sure about this part).

+1 for Thread.each_caller_location or Kernel.each_caller_location yielding Thread::Backtrace::Location objects as @jeremyevans0 proposed.

(I give up on find_caller*, while more optimize-able it likely won't matter too much in practice in comparison to the costs of stack walking, and I agree the each_* variant is more intuitive in Ruby)

#19 - 01/11/2022 04:47 PM - Dan0042 (Daniel DeLorme)
I was thinking what should be the return value of each_caller() with no break. Building and returning an array of frames seems wasteful since in most cases it would never be used (due to break), so nil seems a better choice. But then it occurred to me we could take inspiration from #18136 and have an API like callers_upto( cond ) to return all frames instantiated up to and including the condition. So the "search" case becomes callers_upto( cond ).last.

That would mean always keeping an Array while iterating, so that's inefficient for all cases (except the special case of wanting all frames up to a given frame, but that's not hard to write manually either).

This API is never for daily use, it is no need to make it so convenient like a generator. So, it would return nil if a block is given, and raise LocalJumpError or something if a block is not given. Still, we can create an enumerator by Thread.to_enum(:each_caller_location), but the behavior will be not guaranteed, I guess.

I think a prototype would make it easier to talk about.

I did a quick implementation for JRuby in https://github.com/jruby/jruby/pull/7014. It passes all tests except the to_enum form, which might be tricky to support. When the Enumerator is actually run, we are at least two frames deeper (to_a and each) which throws off the "top" frame in the resulting iteration.

I added a comment to jeremyevans0 (Jeremy Evans)'s PR for discussion.

I did a quick implementation for JRuby in https://github.com/jruby/jruby/pull/7014. It passes all tests except the to_enum form, which might be tricky to support. When the Enumerator is actually run, we are at least two frames deeper (to_a and each) which throws off the "top" frame in the resulting iteration.

I added the to_enum tests to make sure that using it doesn't crash the interpreter, not because it should ever be used. So I don't think it's an issue. I'm fine modifying the tests so they pass on both CRuby's and JRuby's implementation.

I think a prototype would make it easier to talk about.
Is there a way to prohibit the to_enum form? I don't think it will be useful to support since it may vary greatly across implementations (depending on what the stack looks like and what gets included).

I don't know of a way to do this. I don't think the method being called can tell whether it is being run via an enumerator or not. In CRuby, you could probably override to_enum, check if the method being called is a C method using the same C function, and fail in that case, but that's trivial to work around by rebinding the Kernel to_enum method.

It does raise a question for me, though... is your patch eagerly capturing the stack? I have been trying to figure out how the stack trace is identical in both the block form and the to_enum form and it seems like the trace would have to have been captured at the same level, rather than at the point the enum starts to run.

It isn't identical. From the tests:

```ruby
cllr = caller_locations(1, 2); ary = Thread.to_enum(:each_caller_location).to_a[2..3]
assert_equal(cllr.map(&:to_s), ary.map(&:to_s))
```

The [2..3] shows there are extra entries in the to_enum case.

#27 - 02/17/2022 07:56 AM - matz (Yukihiro Matsumoto)
I accept each_caller_location.

Matz.

#28 - 02/17/2022 04:54 PM - jeremyevans (Jeremy Evans)
- Status changed from Open to Closed

Applied in changeset git|4c366ec9775eb6acb3fcb3b88038d051512c75a2

Add Thread.each_caller_location

This method takes a block and yields Thread::Backtrace::Location objects to the block. It does not take arguments, and always starts at the default frame that caller_locations would start at.

Implements [Feature #16663]