Ruby - Feature #19062

Introduce `Fiber#locals` for shared inheritable state.

10/16/2022 10:31 AM - ioquatix (Samuel Williams)

Status:	Closed
Priority:	Normal
Assignee:	ioquatix (Samuel Williams)
Target version:	

Description

After exploring https://bugs.ruby-lang.org/issues/19058, I felt uncomfortable about the performance of copying lots of inheritable attributes. Please review that issue for the background and summary of the problem.

Proposal

Introduce Fiber#locals which is a hash table of local attributes which are inherited by child fibers.

```
Fiber.current.locals[:x] = 10

Fiber.new do
  pp Fiber.current.locals[:x] # => 10
end
```

It's possible to reset Fiber.current.locals, e.g.

```
def accept_connection(peer)
  Fiber.new(locals: nil) do # This causes a new hash table to be allocated.
    # Generate a new request id for all fibers nested in this one:
    Fiber[:request_id] = SecureRandom.hex(32)
    @app.call(env)
  end.resume
end
```

A high level overview of the proposed changes:

```
class Fiber
  def initialize(..., locals: Fiber.current.locals)
    @locals = locals || Hash.new
  end

  attr_accessor :locals

  def self.[] key
    self.current.locals[key]
  end

  def self.[]= key, value
    self.current.locals[key] = value
  end
end
```

See the pull request https://github.com/ruby/ruby/pull/6566 for the full proposed implementation.

Expected Usage

Currently, a lot of libraries use Thread.current[:x] which is unexpectedly "fiber local". A common bug shows up when lazy enumerators are used, because it may create an internal fiber. Because locals are inherited, code which uses Fiber[:x] will not suffer from this problem.

Any program that uses true thread locals for per-request state, can adopt the proposed Fiber#locals and get similar behaviour, without breaking on per-fiber servers like Falcon, because Falcon can "reset" Fiber.current.locals for each request fiber, while servers like Puma won't have to do that and will retain thread-local behaviour.

Libraries like ActiveRecord can adopt Fiber#locals to avoid the need for users to opt into different "IsolatedExecutionState" models, since it can be transparently handled by the web server (see https://github.com/rails/rails/pull/43596 for more details).

We hope by introducing Fiber#locals, we can avoid all the confusion and bugs of the past designs.

Related issues:

Related to Ruby - Feature #19058: Introduce `Fiber.inheritable` attributes/va...	**Closed**
Related to Ruby - Feature #19078: Introduce `Fiber#storage` for inheritable f...	**Closed**

History

#1 - 10/16/2022 10:35 AM - ioquatix (Samuel Williams)

- Description updated

#2 - 10/16/2022 10:40 AM - ioquatix (Samuel Williams)

- Description updated

#3 - 10/16/2022 10:40 AM - ioquatix (Samuel Williams)

- Description updated

#4 - 10/16/2022 10:42 AM - ioquatix (Samuel Williams)

- Description updated

#5 - 10/16/2022 10:47 AM - ioquatix (Samuel Williams)

- Description updated

#6 - 10/16/2022 10:08 PM - ioquatix (Samuel Williams)

As an example of a specific use case, the https://github.com/BMorearty/request_store-fibers gem implements a similar per-fiber state (although it's more complex than the proposed interface).

#7 - 10/17/2022 07:01 AM - nobu (Nobuyoshi Nakada)

Seems pretty different from the thread local storage.
Intentional?

#8 - 10/17/2022 07:40 AM - byroot (Jean Boussier)

So that is indeed a need that arise when you start using fibers or even threads. Existing Ruby code rely a lot on Fiber / Thread global state.

The problem however is the semantic. If all the values in Fiber.locals are immutable, it's quite painless, but if they are mutable what do you do?

Do you share that mutable state? Do you copy or deep copy that state?

Then what if I want to spawn a new Fiber with a clean state? All this to say, I'm not sure whether this is something the programming language should offer. It sounds a bit like a library / framework construct to me.

Do we have any precedent of a similar feature in another language? I suppose UNIX environment variables work a bit like this, but they are limited to Hash[String, String] which simplify the mutability concerns.

#9 - 10/17/2022 07:50 AM - ioquatix (Samuel Williams)

> The problem however is the semantic. If all the values in Fiber.locals are immutable, it's quite painless, but if they are mutable what do you do?

Can you give an example of the problem caused by mutability?

> Then what if I want to spawn a new Fiber with a clean state?

The simplest way: Fiber.new(locals: {}) but we could make an explicit interface for this.

> All this to say, I'm not sure whether this is something the programming language should offer. It sounds a bit like a library / framework construct to me.

It's absolutely something that should be provided by the language, otherwise it gets reinvented over and over, and it's incompatible with each other.

> Do we have any precedent of a similar feature in another language?

Yes, I already mentioned it, a similar concept is used by Kotlin and there is a similar proposal from Java. Please review the linked issue if you have time which goes into more detail.

#10 - 10/17/2022 08:23 AM - byroot (Jean Boussier)

> Can you give an example of the problem caused by mutability?

Sure, you can look at how ActiveSupport::IsolatedExecutionState is used today. For instance ActiveSupport::CurrentAttributes (but applies to most mutable values in there).

If you simply inherit this hash, a child fiber changing Something.current would change it in its parent, sibling and child fibers too, which is super unlikely to be what you want.

Of course the fix is to always copy that registry hash before mutating it, but that's the common question of wether we should expose common traps like this or not.

It's interesting to note that the JEP 429 you mention in the other ticket very explicitly only allow immutable objects.

As always, just because you can shoot yourself in the foot with it doesn't mean it shouldn't be provided, but I think it's interesting to point it out.

> Please review the linked issue

Apologies, I initially missed it.

Again, I think this is definitely a common need, just like POSIX environment variables. And if this was to go through I'd like the same thing for thread locals.

I just think we need to be careful with the semantic. I wonder if applying something akin to Ractor.make_shareable on assignation could make sense (just a quick thought).

#11 - 10/17/2022 08:27 AM - ioquatix (Samuel Williams)

> If you simply inherit this hash, a child fiber changing Something.current would change it in its parent, sibling and child fibers too, which is super unlikely to be what you want.

I think of that more of a design choice rather than an outright issue.

I tried to think in a more abstract sense, and you have:

```
Server Thread
|
|- Request Fiber   \
| |- Fiber         | All share same mutable locals
| \- Fiber         /
|
\- Request Fiber   \
  |- Fiber         | All share same mutable locals
  \- Fiber         /
```

> If you simply inherit this hash, a child fiber changing Something.current would change it in its parent, sibling and child fibers too, which is super unlikely to be what you want.

I'm not sure this is strictly true. In any case, with mutable locals, it's your choice to write:

```
Fiber.current.locals = Fiber.current.locals.dup
Fiber[:x] = 10
```

I would say, in some cases, it's an advantage to have shared mutable state.

> And if this was to go through I'd like the same thing for thread locals.

Can you explain in more detail what you mean?

#12 - 10/17/2022 08:42 AM - byroot (Jean Boussier)

> I think of that more of a design choice rather than an outright issue.

Of course. But it is generally considered good design to help avoid shared mutable state. But again, I can live with the current proposal.

> Can you explain in more detail what you mean?

Well same than for your Fiber example, but with threads. You may want to use threads to process a unit of work concurrently:

```
Server Process
|
|- Request Thread 1
| |- Sub Thread 1 (fetch and compute user info)
| \- Sub Thread 2 (fetch and compute product info)
|
\- Request Thread 2
  |- Sub Thread 1 (fetch and compute user info)
  \- Sub Thread 2 (fetch and compute product info)
```

When you do this today, you end up with the similar challenge you describe for fibers, as the newly spawned thread starts with an empty local store, and may lose some of the unit of work context that was stored there (typically ActiveSupport::CurrentAttributes).

> Fiber#locals

This makes me think that locals may not be the best name, since (to me at least) "locals" suggest that it isn't shared with anyone else. I think naming it inheritable-something instead would make sense. As a library or application author, I could then pick and chose between local and inherited storage depending on my use case.

#13 - 10/17/2022 09:18 AM - ioquatix (Samuel Williams)

> Well same than for your Fiber example, but with threads.

I don't think we should introduce Thread#locals because then we end up with two interfaces for execution context state. Fiber locals are introducing a limited form of "dynamically scoped" variables (or extent local as the JEP calls them). The entire point of this proposal is to avoid having multiple distinct interfaces for this state - so that we don't end up with code which is compatible with threads but not fibers or vice versa.

If you want to propagate state between threads, it can be done using the current proposal and is fairly straight forward. When constructing the thread, we can copy from the current fiber locals, and then this copied (or assigned) into the root fiber locals. For threads, the shared mutable state is more of a problem, since without locking it would be unsafe to mutate. Using dup when copying into the thread would be one way, but it doesn't prevent nested objects from being thread unsafe. For very basic things, like request_id or connection pools (which in theory can be thread safe), it should be totally fine.

I'll update the PR to show how this can work.

> This makes me think that locals may not be the best name

This terminology is well established, e.g. thread-locals, finer-locals - extent locals (JEP). I don't have a strong opinion but I'm also not sure there is a better name. We already have convenient support for non-inherited per-fiber variables:

```
class Fiber
  attr_accessor :my_per_thread_variable
end
```

#14 - 10/17/2022 09:30 AM - ioquatix (Samuel Williams)

Just for my own note:

```
struct rb_execution_context_struct {
    /* storage (ec (fiber) local) */
    struct rb_id_table *local_storage;
}
```

Fiber locals already exist on rb_execution_context_struct.

#15 - 10/17/2022 09:44 AM - ioquatix (Samuel Williams)

Okay, with this small change, we can propagate locals across threads:
https://github.com/ruby/ruby/pull/6566/commits/ac09cff4c36ca9535b0439e3d599ac7ef9899c97

Small test program:

```
Fiber[:x] = 10

thread = Thread.new do
  pp Fiber.current.locals # {:x=>10}
end

thread.join
```

My PR uses dup but I'm not sure if we should use that, it's a performance bottleneck and it assumes people can't use this feature safely (which is already tricky). @Eregon (Benoit Daloze) what do you think?

#16 - 10/17/2022 10:29 AM - Eregon (Benoit Daloze)

For sure we need the .dup on the internal hash used for storage when inheriting it to a new Fiber.
It would be a semantic bug if adding a new local in a child fiber affects a parent fiber as @byroot (Jean Boussier) said.
Or we'd need some kind of persistent map to avoid the actual copy, but semantically it would behave the same as if copied.

The question is should we deep copy / dup the values too?
(for the keys probably we can require Symbol/String and freeze if a String like regular Hash do).

I'm not sure what's the advantage of this proposal vs #19058.
In both cases it's O(locals) per new fiber if using a hash.
The persistent map would make that O(1), but at the cost of slower lookup and updates.

> we can propagate locals across threads

Isn't that always a bad idea? Sharing e.g. a DB connection between threads would be unsafe.
If we allow inheriting across threads then at least it should be very explicit.

#17 - 10/17/2022 10:48 AM - ioquatix (Samuel Williams)

> It would be a semantic bug if adding a new local in a child fiber affects a parent fiber as @byroot (Jean Boussier) (Jean Boussier) said.

I don't agree this is a semantic bug. Ruby doesn't have any existing data structures which behave like this, and Ruby is fundamentally a mutable language.

> The question is should we deep copy / dup the values too?

No, I think this is a bad idea, not only is it expensive, it's not a well defined operation in Ruby, and prevents lots of useful sharing, like connection pools.

> I'm not sure what's the advantage of this proposal vs #19058. In both cases it's O(locals) per new fiber if using a hash.

No, it's not. This is O(1) per fiber creation, and O(hash dup) per thread creation. So it's more efficient.

There are no thread safety concerns when inheriting the locals by a fiber, so there is no need to pay any cost there. Yes, semantically it's different, but threads and fibers are semantically very different (in terms of what operations are safe when interacting with other threads/fibers/etc), so I'm not sure it's a problem, and I can't see any use case hindered by the proposed model, and I basically only advantages over what people are doing currently.

> Isn't that always a bad idea? Sharing e.g. a DB connection between threads would be unsafe.

No, it's not bad to share a DB connection between threads, if the DB connection itself is thread safe, it's totally fine. It's also more likely you'd be sharing a connection pool (which in most Ruby projects are unfortunately just globals).

> If we allow inheriting across threads then at least it should be very explicit.

I don't agree with this, the entire point of this model is to have implicit sharing of important things, e.g. request_id, connection_pool, trace_id, etc. Being explicit is more likely to ensure that these fields are not propagated correctly.

#18 - 10/17/2022 10:55 AM - byroot (Jean Boussier)

> No, it's not bad to share a DB connection between threads, if the DB connection itself is thread safe

"Thread safe" can mean a lot of things. Generally speaking, the vast majority of protocols don't support concurrent access, so most database client protect the connection with a mutex or similar. If you use concurrency primitives like fibers or threads you likely don't want to have to wait for other threads to be done with the connection.

That is why I don't think a single store cuts it. I believe having an "inheritance mechanism" makes perfect sense, but also having a store that isn't inherited does too.

#19 - 10/17/2022 11:12 AM - ioquatix (Samuel Williams)

ActiveRecord connections are thread safe, and it makes total sense to me that an enumerator might like to run in the same transaction as its surrounding context. In fact I've read many times people complaining about this exact problem.

> I believe having an "inheritance mechanism" makes perfect sense

Yes, it can do, but it's also costly.

> but also having a store that isn't inherited does too.

In practice, as shown with lazy enumerators that internally create fibers, I would argue that it doesn't actually make a huge amount of sense in comparison to dynamically scoped variables (which is roughly what this proposal implements at the fiber level, rather than stack frame level which would be prohibitively complex).

#20 - 10/17/2022 11:48 AM - byroot (Jean Boussier)

> ActiveRecord connections are thread safe

Again, "thread safe" is a very loaded term. It just means that things won't be utterly broken if you use that API in a concurrent scenario, it doesn't mean that you want to share that connection object between multiple threads / fibers.

> Yes, it can do, but it's also costly.

There must be a misunderstanding, because I'm not proposing anything different from you here. All I'm saying is that your proposed API should be in addition to the existing fiber locals, not replace them.

#21 - 10/17/2022 11:49 AM - Eregon (Benoit Daloze)

ioquatix (Samuel Williams) wrote in #note-17:

> > It would be a semantic bug if adding a new local in a child fiber affects a parent fiber as @byroot (Jean Boussier) (Jean Boussier) said.
>
> I don't agree this is a semantic bug. Ruby doesn't have any existing data structures which behave like this, and Ruby is fundamentally a mutable language.

I would think all other languages with similar functionality don't have new locals in the parent/child leaking to the other one. Am I wrong?
Also that updating a local affects both and not only one of them (I think should be one).

> No, I think this is a bad idea, not only is it expensive, it's not a well defined operation in Ruby, and prevents lots of useful sharing, like connection pools.

Right, probably not necessary for these locals to copy the values.

> > I'm not sure what's the advantage of this proposal vs #19058. In both cases it's O(locals) per new fiber if using a hash.
>
> No, it's not. This is O(1) per fiber creation, and O(hash dup) per thread creation. So it's more efficient.

Only if sharing the Hash semantics which seems very counter-intuitive.

> There are no thread safety concerns when inheriting the locals by a fiber

Only if it's a Fiber of the same Thread.
And also it's invalid e.g. if sharing a DB connection (using some fd) and the Fiber scheduler, e.g. it can mismatch requests/responses.
Making the DB connections thread/concurrency-safe seems difficult (most don't), and would lose the point of it being fiber/thread-local and not needing such synchronization.

> Isn't that always a bad idea? Sharing e.g. a DB connection between threads would be unsafe.

No, it's not bad to share a DB connection between threads, if the DB connection itself is thread safe, it's totally fine. It's also more likely you'd be sharing a connection pool (which in most Ruby projects are unfortunately just globals).

What's a concrete scenario where the connection pool should be inherited like that vs global and it would be necessary?

> If we allow inheriting across threads then at least it should be very explicit.

I don't agree with this, the entire point of this model is to have implicit sharing of important things, e.g. request_id, connection_pool, trace_id, etc. Being explicit is more likely to ensure that these fields are not propagated correctly.

It seems extremely unsafe to inherit by default across threads to me.

I'd think 99% of what you want to solve here is inheriting between fibers and notably the Enumerator fibers, right?

#22 - 10/17/2022 07:42 PM - ioquatix (Samuel Williams)

The only thing I really care about is an efficient way to implicitly share specific state within an execution context.

```
let(x = 10) do
  Thread.new do
    get(x) # => 10
  end

  Enumerator.new do
    get(x) # => 10
  end
end
```

Limiting variable binding to some kind of let block is a bit limiting and prevents lots of use cases which are common in Ruby. I'm okay with Fibers (read execution contexts) being the implicit scope because it's pretty much natural for most practical use cases.

```
Fiber/Thread.new do # An execution context
  set(x)
  Thread.new do
    get(x) # => 10
  end.join

  set(x, 20)
  Enumerator.new do
    get(x, 20) # => 20
  end
end
```

That means all the ECs share a single mutable set of locals. It's basically the same as a local variable bound except rather than lexical scope, it's dynamic based on the ECs.

There are two levels of mutability:

1. The ability to bind key-values.
2. The ability for values themselves to mutate.

I would argue that since we can't practically prevent (2), trying to prevent (1) is pointless. However, feel free to convince me otherwise.

I'd be okay with a copy-on-write scheme where the first update would pay the cost of the internal copy, since we expect writes to be much less frequent than reads and clones, both of which need to be O(1) where possible. But let me add, because of the above, it's trivially possible to bypass such a copy operation, i.e. Fiber[:x] = [] will create a shared reference and no amount of duping the locals will solve that problem. So why even bother?

Whatever model we come up with, it makes sense that threads and fibers are handled consistently, i.e. I'm not sure that Thread.new needs to dup the locals. It's not thread unsafe to update the locals in Ruby because of the GVL. It's just poorly synchronised. If users choose to write that kind of code, they'd need to provide their own locking, which I think is acceptable too.

When you say something like "It seems extremely unsafe to inherit by default across threads to me" I would personally like to see the code example

where it's unsafe, otherwise it's hard for me to understand exactly what the problem is and/or how we could address it.

#23 - 10/17/2022 07:57 PM - ioquatix (Samuel Williams)

Here is a short example of what I want:

```
Fiber[:count] = 0

enumerator = Enumerator.new do |y|
  10.times do |i|
    Fiber[:count] += 1
    y << i
  end
end

enumerator.next
p Fiber[:count] # 1

enumerator.to_a
p Fiber[:count] # 11
```

If you implicitly dup, there will be odd side effects in this example.

#24 - 10/17/2022 08:05 PM - byroot (Jean Boussier)

> Here is a short example of what I want:

That example makes very little sense to me:

- How is this a "local" if it's accessible by other fibers?
- I could understand inheriting (a copy of) the locals of the parent fibers, but not sharing these "locals".
- Even if the "locals hash" is copied when a new fiber is created, you can still have the behavior your want by storing a mutable object.

```
Counter = Struct.new(:count)

Fiber[:counter] = Count.new(0)

enumerator = Enumerator.new do |y|
 10.times do |i|
  Fiber[:counter].count += 1
  y << i
 end
end

enumerator.next
p Fiber[:counter].count # 1

enumerator.to_a
p Fiber[:counter].count # 11
```

I think what you currently propose as the same type of semantic issue than Ruby's class variables, where the variable is shared with all the descendants.

#25 - 10/17/2022 09:58 PM - ioquatix (Samuel Williams)

> Even if the "locals hash" is copied when a new fiber is created, you can still have the behavior your want by storing a mutable object.

And that's exactly my point, so why bother trying to prevent mutability, especially if there is a performance cost. I'm not against it, I'm just not convinced the benefit outweighs the cost.

> That example makes very little sense to me.

Okay, so in your opinion, this is more logical?

```
Fiber[:count] = 0

enumerator = Enumerator.new do |y|
  10.times do |i|
```

```
    Fiber[:count] += 1
    y << i
  end
end

enumerator.next
p Fiber[:count] # 0

enumerator.to_a
p Fiber[:count] # 10
```

#26 - 10/18/2022 08:14 AM - byroot (Jean Boussier)

> this is more logical?

What is logical to me is to behave like POSIX environment variables. When you spawn a new fiber/thread it is initialized with a *shallow copy* of it's parent "locals".

What makes sense to me is:

```
Fiber[:count] = 1
Fiber.new do
  p Fiber[:count] # => 1
  Fiber[:count] += 1
  p Fiber[:count] # => 2
end.resume
p Fiber[:count] # => 1
```

It's easily understood as (pseudo code):

```
class Fiber
  def initialize(locals: Fiber.current.locals.dup)
    # ...
  end
end
```

#27 - 10/18/2022 10:15 AM - ioquatix (Samuel Williams)

Thanks for your examples. I agree there are different ways to solve this problem.

The key model I'm trying to introduce is a shared state "per request". It's actually similar to https://github.com/steveklabnik/request_store in a lot of ways.

I'm concerned about the memory and performance cost of calling dup per fiber.

I'm also concerned about hidden fibers (and threads) causing the visible behaviour to change unexpectedly.

It also prevents shared optimisations which is something I want to use for my connection pool code.

```
def connection_pool
  Fiber[:connection_pool] ||= ConnectionPool.new
end
```

Scenario 1:

```
connection_pool.query(...)

Fiber.new do
  # Reuse connection pool:
  connection_pool.query(...)
end.resume
```

Scenario 2:

```
Fiber.new do
  connection_pool.query(...)
end.resume

connection_pool.query # oh no, it will create a 2nd connection pool.
```

It also stands to reason, that even if we followed your suggestion, simply having Fiber.curent.locals[:shared] = {} at the very root reintroduces the same behaviour as I'm proposing, so it's unavoidable.

I feel that the current PR is pragmatic, small, and efficient, and solves a fairly fundamental problem.

I understand the semantic value in always calling dup, but as of right now, I'm not convinced that semantic model outweighs the practical issues listed above. I can be convinced of course! But especially the performance cost concerns me a lot, because Async encourages lots of short lived fibers.

#28 - 10/18/2022 10:37 AM - byroot (Jean Boussier)

> I'm concerned about the memory and performance cost of calling dup per fiber.

It's extremely negligible.

> I'm also concerned about hidden fibers (and threads) causing the visible behaviour to change unexpectedly.

I really don't see how.

> It also prevents shared optimisations which is something I want to use for my connection pool code.

No, again, new fibers are initialized with a dup of the locals hash, so they have access to the same ConnectionPool instance, which can be mutable if you desire so.

> It also stands to reason, that even if we followed your suggestion, simply having Fiber.curent.locals[:shared] = {} at the very root reintroduces the same behaviour as I'm proposing

Yes and it's fine, only the top level store is shallow-copied, it's ok if you wish to use mutable values in there.

Now imagine something like this:

```
module MyLibrary
  extend self

  def with_log_level(level)
   old_level = Fiber[:log_level]
   Fiber[:log_level] = level
   yield
  ensure
   Fiber[:log_level] = old_level
  end
end
```

- Should a newly spawn fiber inherit whatever log_level it's parent set? Yes definitely.
- Should a child fiber be able to flip the log level in its parent and sibblings? Definitely not.

That is for this kind of use cases I think dup is necessary.

If you don't dup, there this is absolutely not "locals" it's just some fairly contrived global variable.

> But especially the performance cost concerns me a lot

Hash#dup is something Ruby code does constantly. That isn't remotely a performance concern.

#29 - 10/18/2022 11:05 AM - Eregon (Benoit Daloze)

> It's not thread unsafe to update the locals in Ruby because of the GVL.

We should not design general semantics based on a CRuby-only implementation detail.
It is thread unsafe and could crash other Ruby implementations for instance.

I agree with what @byroot (Jean Boussier) says.
If it's some sort of "local/x-local" then I expect save/restore like with_log_level to work correctly, to have mutations isolated per Fiber, etc.

> When you say something like "It seems extremely unsafe to inherit by default across threads to me" I would personally like to see the code example where it's unsafe, otherwise it's hard for me to understand exactly what the problem is and/or how we could address it.

Example:

```
Fiber.current.locals[:db_connection] = some_IO
Thread.new { Fiber.current.locals[:db_connection].query(...) }
Thread.new { Fiber.current.locals[:db_connection].query(...) }
```

This badly breaks the application because the fd is incorrectly shared between threads.
If the name is "something local" I'd think people expect the value to actually be at least Fiber-local.

I guess the name is the fundamental problem here. I believe what you propose is not related to what most people understand by "Fiber locals".

I think any such sharing should be explicit, e.g.:

```
# or maybe with_inherited_state
Fiber.with_shared_state({ connection_pool: pool }, inherited_by: Fiber / [Fiber, Thread]) {
  Thread.new {
    Fiber.current.shared_state(:connection_pool) # => pool
  }
  Fiber.new {
    Fiber.current.shared_state(:connection_pool) # => pool
  }.resume
}
```

BTW Java has:

- https://docs.oracle.com/javase/7/docs/api/java/lang/InheritableThreadLocal.html, i.e., a way to mark which Fiber-local should be inherited
- https://download.java.net/java/early_access/loom/docs/api/java.base/java/lang/Thread.Builder.html#allowSetThreadLocals(boolean) which says Sets whether the thread is allowed to set values for its copy of thread-local variables.. So the intuitive thing of having a copy per thread.
- https://cr.openjdk.java.net/~rpressler/loom/loom/sol1_part2.html talks about scoped variables, which seems closer to what you want

#30 - 10/18/2022 11:06 AM - byroot (Jean Boussier)

- Tracker changed from Bug to Feature

- Backport deleted (2.7: UNKNOWN, 3.0: UNKNOWN, 3.1: UNKNOWN)

#31 - 10/18/2022 11:20 AM - ioquatix (Samuel Williams)

- Tracker changed from Feature to Bug

- Backport set to 2.7: UNKNOWN, 3.0: UNKNOWN, 3.1: UNKNOWN

> It's extremely negligible.

Okay, good to know.

```
require 'benchmark'

locals = {a: 1, b: 2, c: 3}

Benchmark.bm do |b|
  b.report("dup") {1_000_000.times{x = locals.dup}}
  b.report("assign") {1_000_000.times{x = locals}}
end
```

```
              user     system      total        real
dup       0.175779   0.001153   0.176932 (  0.176988)
assign    0.023896   0.000084   0.023980 (  0.023998)
```

So, like, about 10x slower than doing nothing, but it's also a pretty small overhead per fiber, so it might be acceptable to describe it as negligible, unless there are worse cases we should consider.

That being said, I'd prefer to avoid doing things we don't need to do.

> Should a child fiber be able to flip the log level in its parent and sibblings? Definitely not.

If all fibers are part of the same request, I have no problem with that model. This proposal is two parts: a way to inherit state to child execution contexts, and a way to share that state between execution contexts within the same request or operation.

> No, again, new fibers are initialized with a dup of the locals hash, so they have access to the same ConnectionPool instance, which can be mutable if you desire so.

Not if it wasn't created in the parent first. If you have code to lazy initialize the connection pool, and it happens in the child fiber before it happens in

the parent, it won't be shared if the locals are duped.

> I really don't see how.

I gave an example showing exactly this.

```
Fiber[:count] = 0

enumerator = Enumerator.new do |y|
  10.times do |i|
    Fiber[:count] += 1
    y << i
  end
end

enumerator.next
p Fiber[:count] # 1

enumerator.to_a
p Fiber[:count] # 11
```

If the locals are duped, the hidden fiber will dup the locals and the results will be 0 and 10 respectively. To me, this is confusing, since the fiber is an implementation detail. One of the problems I'm trying to solve is behavioural changes due to the internal fiber of enumerator. Can you propose some other way to fix this?

> If you don't dup, there this is absolutely not "locals" it's just some fairly contrived global variable.

I mean, if you want to do contrived, everything is basically just a global variable if you squint hard enough :)

> It is thread unsafe and could crash other Ruby implementations for instance.

It's not thread unsafe to share a hash table between fibers. In my PR, the locals are duped between threads to prevent any thread safety issues.

> This badly breaks the application because the fd is incorrectly shared between threads.

I like this example, probably sharing across thread boundaries is dangerous. The solution you proposed feels far to complex for Ruby.

The model I'm inspired by is "dynamically scoped free variables" such as those from LISP. I think that's a pretty good model, but it's still a little too cumbersome for Ruby.

> I guess the name is the fundamental problem here. I believe what you propose is not related to what most people understand by "Fiber locals".

Yes, it can be. Actually, the entire way thread local and fiber local is named and used in Ruby is a total mess.

> BTW Java has:

https://openjdk.org/jeps/429 is a nice proposal I linked in the other issue.

If everyone feels strongly about dup in every case, I'm okay with it (assuming the performance is acceptable), but it means that Enumerator's internal fiber has user visible side effects, which was something I wanted to avoid as it's caused a lot of pain in the past.

Maybe Fibers can default to with dup but enumerator's hidden fiber can default to without dup to avoid this problem. What do you think?

#32 - 10/18/2022 11:21 AM - ioquatix (Samuel Williams)

- Tracker changed from Bug to Feature

- Backport deleted (2.7: UNKNOWN, 3.0: UNKNOWN, 3.1: UNKNOWN)

#33 - 10/18/2022 11:26 AM - ioquatix (Samuel Williams)

If we were concerned about having a separate but similarly inheritable scope for Threads, maybe we should have this model:

```
Thread.locals -> child threads will inherit
Fiber.locals -> initially blank and only propagated to other fibers, OR
Fiber.locals -> initially inherited from Thread.locals, propagated to other fibers.
```

What do you think? It means things that are safe to use between fibers but not threads, won't get inherited by threads. However, I'm a little concerned this will cause people to adopt thread locals and be incompatible with request per fiber models.

#34 - 10/18/2022 11:44 AM - byroot (Jean Boussier)

Sorry but that's not remotely a good benchmark. We're talking about a potential overhead, so it has to be considered relative to the exiting work.

```
require 'bundler/inline'

gemfile do
  source 'https://rubygems.org'
  gem 'benchmark-ips'
end

locals = { a: 1, b: 2 }
Benchmark.ips do |x|
  x.time = 5
  x.report('baseline') { Fiber.new {} }
  x.report('locals-dup') { locals.dup; Fiber.new {} }
  x.compare!(order: :baseline)
end
```

> If all fibers are part of the same request, I have no problem with that model.

The problem is that with the fiber scheduler, the execution order might be different, so some codepaths will something run with a specific log level and something with another. That's terrible behavior.

> If you have code to lazy initialize the connection pool

Then don't. Eagerly instantiate that pool, set it in the inherited registry and have that object be lazy.

#35 - 10/18/2022 11:55 AM - ioquatix (Samuel Williams)

```
Warming up --------------------------------------
            baseline   287.720k i/100ms
          locals-dup   229.210k i/100ms
Calculating --------------------------------------
            baseline      2.922M (± 0.7%) i/s -     14.674M in   5.022671s
          locals-dup      2.289M (± 0.4%) i/s -     11.460M in   5.007640s

Comparison:
            baseline:  2921640.6 i/s
          locals-dup:  2288640.1 i/s - 1.28x  (± 0.00) slower
```

So in the context of creating a fiber, it's 30% slower. That's quite significant.

> The problem is that with the fiber scheduler, the execution order might be different, so some codepaths will something run with a specific log level and something with another. That's terrible behavior.

Yep, that's a fair point.

> Then don't. Eagerly instantiate that pool, set it in the inherited registry and have that object be lazy.

I fear this will cause people to use thread locals instead.

#36 - 10/18/2022 12:06 PM - byroot (Jean Boussier)

> So in the context of creating a fiber, it's 30% slower. That's quite significant.

It's less than that on my machine, more like 20%, also it would be sligthly faster than that since done from C, so no method lookup etc.

```
Warming up --------------------------------------
            baseline   221.823k i/100ms
          locals-dup   185.551k i/100ms
Calculating --------------------------------------
            baseline      2.223M (±19.4%) i/s -     10.869M in   5.102287s
          locals-dup      1.835M (±19.3%) i/s -      8.906M in   5.058429s

Comparison:
            baseline:  2222864.6 i/s
          locals-dup:  1834532.9 i/s - same-ish: difference falls within error
```

> I fear this will cause people to use thread locals instead.

Well, a connection pool should be global anyway, so again the example is not the best.

The key argument here is about composability. If you don't dup you break composability, the same way Ruby class variables break composability.

#37 - 10/18/2022 12:32 PM - Eregon (Benoit Daloze)

> One of the problems I'm trying to solve is behavioural changes due to the internal fiber of enumerator. Can you propose some other way to fix this?

Maybe Enumerator-created Fiber should always use the Fiber locals of the caller Fiber?

IMHO Enumerator-created Fiber are too slow and should not be used, so it's rarely a problem in practice.
Also I think the Enumerator#{peek,next} API feels unidiomatic and there is rarely if ever a good reason to use it, and IIRC that's the only two methods creating a Fiber in Enumerator.

#38 - 10/18/2022 12:33 PM - Eregon (Benoit Daloze)

- Related to Feature #19058: Introduce `Fiber.inheritable` attributes/variables for dealing with shared state. added

#39 - 10/20/2022 06:48 AM - matz (Yukihiro Matsumoto)

At least it shouldn't be named locals since it's not **local** to a fiber.
Maybe storage?

Matz.

#40 - 10/20/2022 12:09 PM - ioquatix (Samuel Williams)

- Status changed from Open to Closed

After many discussion, I think it's fair to say, dup by default makes the most sense, and with that model, it could be considered more local. So I'll open a new proposal with the refined design.

#41 - 11/08/2022 03:07 PM - Eregon (Benoit Daloze)

- Related to Feature #19078: Introduce `Fiber#storage` for inheritable fiber-scoped variables. added