asyncmachine-go manual

- version `v0.10.3`
- Legend
- Machine and States
  - Defining States
  - Asynchronous States
  - Machine Init
  - Clock and Context
  - Active States
  - Inspecting States
  - Auto States
  - Multi States
  - Categories of States
- Changing State
  - State Mutations
  - Mutation Arguments
  - Transition Lifecycle
  - Transition Handlers
  - Self Handlers
  - Defining Handlers
  - Event Struct
  - Calculating Target States
  - Negotiation Handlers
  - Final Handlers
  - Global Handlers
- Advanced Topics
  - Relations
    * `Add` relation
    * `Remove` relation
    * `Require` relation
    * `After` relation
  - Waiting
  - Error Handling
  - Catching Panics
    * Panic in a negotiation handler
    * Panic in a final handler
    * Panic anywhere else
  - Queue and History
  - Logging
    * Customizing Logging
  - Debugging
    * Steps To Debug
    * Enabling Telemetry
    * Breakpoints
  - Typesafe States

Legend

Examples here use a string representations of state machines in the format of `(ActiveState:\d) [InactiveState:\d]` , eg (Foo:1) [Bar:0 Baz:0]. Variables with state machines are called `mach` and pkg/machine is aliased as `am`.

Machine and States

Defining States

States are defined using `am.Struct`, a string-keyed map of `am.State` struct, which consists of **properties and relations**. List of **state names** have a readability shorthand of `am.S` and lists can be combined using `am.SMerge`.

```
am.Struct{
    "StateName": {

        // properties
        Auto:    true,
        Multi:   true,

        // relations
        Require: am.S{"AnotherState1"},
        Add:     am.S{"AnotherState2"},
        Remove:  am.S{"AnotherState3", "AnotherState4"},
        After:   am.S{"AnotherState2"},
    }
}
```

State names have a predefined **naming convention** which is `CamelCase`.

Example - synchronous state

```
Ready: {},
```

Asynchronous States

If a state represents a change from `A` to `B`, then it's considered as an **asynchronous state**. Async states can be represented **by 2 to 4 states**, depending on how granular information we need from them. More than 4 states representing a single abstraction in time is called a Flow.

Example - asynchronous state (double)

```
DownloadingFile: {
    Remove: groupFileDownloaded,
},
FileDownloaded: {
    Remove: groupFileDownloaded,
},
```

Example - asynchronous boolean state (triple)

```
Connected: {
    Remove:  groupConnected,
},
Connecting: {
    Remove:  groupConnected,
},
Disconnecting: {
    Remove: groupConnected,
},
```

Example - full asynchronous boolean state (quadruple)

```
Connected: {
    Remove:  groupConnected,
},
Connecting: {
    Remove:  groupConnected,
},
Disconnecting: {
    Remove: groupConnected,
},
Disconnected: {
    Auto: true,
    Remove: groupConnected,
},
```

Machine Init

There are two ways to initialize a machine - using `am.New` or `am.NewCommon` . The former one always returns an instance of `Machine`, but it's limited to only initializing the states structure and basic customizations via `Opts`. The latter

one is more feature-rich and provides states verification, handler binding, and debugging. It may also return an error.

```go
import am "github.com/pancsta/asyncmachine-go/pkg/machine"

// ...

ctx := context.Background()
states := am.Struct{"Foo":{}, "Bar":{}}
mach := am.New(ctx, states, &am.Opts{
    ID: "foo1",
    LogLevel: am.LogChanges,
})
```

Each machine has an ID (via `Opts.ID` or a random one) and the build-in Exception state.

Clock and Context

Every state has a tick value, which increments ("ticks") every time a state gets activated or de-activated. **Odd ticks mean active, while even ticks mean inactive**. A list (slice) of state ticks forms a **machine clock**. The sum of all the state ticks represents the current **machine time**.

Machine clock is a logical clock, which purpose is to distinguish different instances of the same state. It's most commonly used in the form of `context.Context` via `Machine.NewStateCtx(state string)`, but it also provides methods on its own data type `am.Time`. An instance of state context gets canceled once the state becomes inactive.

Other related methods and functions:

- `Machine.Clock(state string) uint64`
- `Machine.Time(states S) Time`
- `Machine.TimeSum(states S) Time`
- `Machine.IsTime(time Time) bool`
- `Machine.IsClock(clock Clock) bool`
- `IsTimeAfter(t1 Time, t2 Time)`
- `IsActiveTick(tick uint64)`
- `Time.Is(stateIdxs []int)`
- `Time.Is1(stateIdx int)`

Example - clocks

```go
// () [Foo:0 Bar:0 Baz:0 Exception:0]
mach.Clock("Foo") // ->0

mach.Add1("Foo", nil)
mach.Add1("Foo", nil)
```

```go
// (Foo:1) [Bar:0 Baz:0 Exception:0]
mach.Clock("Foo") // ->1

mach.Remove1("Foo", nil)
mach.Add1("Foo", nil)
// (Foo:3) [Bar:0 Baz:0 Exception:0]
mach.Clock("Foo") // ->3
```

Example - state context

```go
func (h *Handlers) DownloadingFileState(e *am.Event) {
    // open until the state remains active
    ctx := e.Machine.NewStateCtx("DownloadingFile")
    // fork to unblock
    go func() {
        // check if still valid
        if ctx.Err() != nil {
            return // expired
        }
    }()
}
```

Side effects:

- expired state context is not an error

Active States

Each state can be **active** or **inactive**, determined by its state clock. You can check the current state at any time, without a long delay, which makes it a dependable source of decisions.

Methods to check the active states:

- `Machine.Is(states)`
- `Machine.Is1(state)`
- `Machine.Not(states)`
- `Machine.Not1(state)`
- `Machine.Any(states1, states2...)`
- `Machine.Any1(state1, state2...)`

Methods to inspect / dump the currently active states:

- `Machine.String()`
- `Machine.StringAll()`
- `Machine.Inspect(states)`

`Is` checks if all the passed states are active.

```go
// () [Foo:0 Bar:0 Baz:0 Exception:0]
```

```
mach.Add1("Foo", nil)
// (Foo:1) [Bar:0 Baz:0 Exception:0]

mach.Is1("Foo") // true
mach.Is(am.S{"Foo", "Bar"}) // false
```

Not checks if none of the passed states is active.

```
// () [A:0 B:0 C:0 D:0 Exception:0]

mach.Add(am.S{"A", "B"}, nil)
// (A:1 B:1) [C:0 D:0 Exception:0]

// not(A) and not(C)
mach.Not(am.S{"A", "C"}) // false
// not(C) and not(D)
mach.Not(am.S{"C", "D"}) // true
```

Any is group call to Is, returns true if any of the params return true from Is.

```
// () [Foo:0 Bar:0 Baz:0 Exception:0]

mach.Add1("Foo", nil)
// (Foo:1) [Bar:0 Baz:0 Exception:0]

// is(Foo, Bar) or is(Bar)
mach.Any(am.S{"Foo", "Bar"}, am.S{"Bar"}) // false
// is(Foo) or is(Bar)
mach.Any(am.S{"Foo"}, am.S{"Bar"}) // true
```

Inspecting States

Being able to inspect your machine at any given step is VERY important. These are the basic method which don't require any additional debugging tools.

Example - inspecting active states and their clocks

```
mach.StringAll() // ->() [Foo:0 Bar:0 Baz:0 Exception:0]
mach.String() // ->()

mach.Add1("Foo")

mach.StringAll() // ->(Foo:1) [Bar:0 Baz:0 Exception:0]
mach.String() // ->(Foo:1)
```

Example - inspecting relations

```
// From examples/temporal-fileprocessing/fileprocessing.go
mach.Inspect()
// Exception:
```

6

```
//   State:   false 0
//
// DownloadingFile:
//   State:   false 1
//   Remove:  FileDownloaded
//
// FileDownloaded:
//   State:   true 1
//   Remove:  DownloadingFile
//
// ProcessingFile:
//   State:   false 1
//   Auto:    true
//   Require: FileDownloaded
//   Remove:  FileProcessed
//
// FileProcessed:
//   State:   true 1
//   Remove:  ProcessingFile
//
// UploadingFile:
//   State:   false 1
//   Auto:    true
//   Require: FileProcessed
//   Remove:  FileUploaded
//
// FileUploaded:
//   State:   true 1
//   Remove:  UploadingFile
```

Auto States

Automatic states (`Auto` property) are one of the most important concepts of
asyncmachine-go. After every transition with a clock change (tick), `Auto`
states will try to active themselves via an auto mutation.

- `Auto` states can be set partially (within the same mutation)
- auto mutation is **prepended** to the queue
- `Remove` relation of `Auto` states isn't enforced within the auto mutation

Example - log for FileProcessed causes an `Auto` state UploadingFile to activate

```
// [state] +FileProcessed -ProcessingFile
// [external] cleanup /tmp/temporal_sample1133869176
// [state:auto] +UploadingFile
```

Multi States

Multi-state (`Multi` property) describes a state which can be activated many times, without being de-activated in the meantime. It always triggers `Enter` and `State` transition handlers, plus the clock is always incremented. It's useful for describing many instances of the same event (e.g. network input) without having to define more than one transition handler. `Exception` is a good example of a `Multi` state (many errors can happen, and we want to know all of them). The downside is that `Multi` states don't have state contexts.

Side effects:

- spawning goroutines in a Multi state handler may lead to an overflow, use `errgroup`

Categories of States

States usually belong to one of these categories:

1. Input states (e.g. RPC msgs)
2. Read-only states (e.g. external state / UI state / summaries)
3. Action states (e.g. Start, ShowModal, public API methods)
4. Background tasks (e.g. Processing)
5. Joining states (e.g. ProcessingDone)

Action states often de-activate themselves after they are done, as a part of their final handler. *Joining states* are used for relations with other states, as relations to an inactive state are not possible.

Example - self removal

```go
func (h *Handlers) ClickState(e *am.Event) {
    // add removal to the queue
    e.Machine.Remove1("Click")
}
```

Example - clock-based self removal

```go
func (h *Handlers) ClickState(e *am.Event) {
    mach := e.Machine
    tick := mach.Tick("Click")

    go func() {
        // ... blocking calls

        // last one deactivates
        if tick == mach.Tick("Click") {
            mach.Remove1("Click", nil)
        }
```

```
    }()
}
```

Changing State

Mutations

Mutation is a request to change the currently active states of a machine. Each mutation has a list of states knows as **called states**, which are different from **target states** (calculated during a transition).

Mutations are queued, thus they are never nested - one can happen only after the previous one has been processed. Mutation methods return a `Result`, which can be:

- `Executed`
- `Canceled`
- `Queued`

You can check if the machine is busy executing a transition by calling `Machine.DuringTransition()` (see queue) or wait until it's done with `<-Machine.WhenQueueEnds()`.

There are 3 types of mutations:

- add
- remove
- set

`Machine.Add(states, args)` is the most common method, as it preserves the currently active states. Each activation increases the states' clock to an odd number.

Example - Add mutation

```
// () [Foo:0 Bar:0 Baz:0 Exception:0]

mach.Add(am.S{"Foo"}, nil)
// (Foo:1) [Bar:0 Baz:0 Exception:0]

mach.Add(am.S{"Bar"}, nil)
// (Foo:1 Bar:1) [Baz:0 Exception:0]

mach.Add1("Bar", nil)
// (Foo:1 Bar:1) [Baz:0 Exception:0]
```

`Machine.Remove(states, args)` deactivates only the specified states. Each de-activation increases the states' clock to an even number.

Example - Remove mutation

```
// () [Foo:0 Bar:0 Baz:0 Exception:0]

mach.Add(am.S{"Foo", "Bar"}, nil)
// (Foo:1 Bar:1) [Baz:0 Exception:0]

mach.Remove(am.S{"Foo"}, nil)
// (Bar:1) [Foo:2 Baz:0 Exception:0]

mach.Remove1("Bar", nil)
// [Foo:2 Bar:2 Baz:0 Exception:0]
```

Machine.Set(states, args) de-activates all but the passed states and activates the remaining ones.

Example - Set mutation

```
// () [Foo:0 Bar:0 Baz:0 Exception:0]

mach.Add1("Foo", nil)
// (Foo:1) [Bar:0 Baz:0 Exception:0]

mach.Set(am.S{"Bar"}, nil)
// (Bar:1) [Foo:2 Baz:0 Exception:0]
```

Side effects:

- mutations panic for unknown states

Arguments

Each mutation has an optional map of arguments of type `am.A`, passed to handlers via the `am.Event` struct. Technically it's simply a `map[string]any` for simplicity, but in real code one should be using typesafe arguments.

Example - passing arguments to handlers

```
args := am.A{"val": "key"}
mach.Add1("Foo", args)

// ...

// wait for a mutation like the one above
<-mach.WhenArgs("Foo", am.A{"val": "key"}, nil)
```

Transition Lifecycle

Transition is created from a mutation and tries to execute it, which can result in the changing of machine's active states. Each transition has several steps and (optionally) calls several handlers (for each of the bindings). Transitions are atomic and optionally panic to the Exception state.

Once a transition begins to execute, it goes through the following steps:

1. Calculating Target States - collecting target states based on relations, currently active states and called states. Transition can already be `Canceled` at this point.
2. Negotiation handlers - methods called for each state about-to-be activated or deactivated. Each of these handlers can return `false`, which will cause the mutation to be `Canceled` and `Transition.Accepted` to be `false`.
3. Apply the **target states** to the machine - from this point `Is` (and other checking methods) will reflect the target states.
4. Final handlers - methods called for each state about-to-be activated or deactivated, as well as self handlers of currently active ones. Transition cannot be canceled at this point.

Transition Handlers

Asyncmachine implements **AOP** (Aspect Oriented Programming) through a handler naming convention. Use `LogEverything` to see a full list of each transition's handlers.

State handler is a struct method with a predefined suffix or prefix, which receives an Event struct. There are negotiation handlers (returning a `bool`) and final handlers (with no return). Order of the handlers depends on currently active states and relations of active and target states. Handlers executed each in a separate goroutine, with a timeout of `Machine.HandlerTimeout`.

Example - handlers for the state Foo

```go
// can Foo activate?
func (h *Handlers) FooEnter(e *am.Event) bool {}
// with Foo active, can Bar activate?
func (h *Handlers) FooBar(e *am.Event) {}
// Foo activates
func (h *Handlers) FooState(e *am.Event) {}
// can Foo de-activate?
func (h *Handlers) FooExit(e *am.Event) bool {}
// Foo de-activates
func (h *Handlers) FooEnd(e *am.Event) {}
```

List of handlers during a transition from `Foo` to `Bar`, in the order of execution:

- `FooExit` - negotiation handler
- `BarEnter` - negotiation handler
- `FooBar` - negotiation handler
- `FooEnd` - final handler
- `BarState` - final handler

All handlers execute in a series, one by one, thus they don't need to mutually exclude each other for accessing resources. This reduces the number of locks

needed. No blocking is allowed in the body of a handler, unless it's in a goroutine. Additionally, each handler has a limited time to complete (**100ms** with the default handler timeout), which can be set via `am.Opts`.

Self Handlers

Self handler is a negotiation handler for states which were active **before and after** a transition (all no-change active states). The name is a doubled name of the state (eg `FooFoo`).

List of handlers during a transition from `Foo` to `Foo Bar`, in the order of execution:

- `BarEnter` - negotiation handler
- `FooFoo` - negotiation handler and **self handler**
- `BarState` - final handler

Self handlers provide a simple alternative to `Multi` states, while fully maintaining state clocks.

Defining Handlers

Handlers are defined as struct methods. Each machine can have many handler structs bound to itself using `Machine.BindHandlers`, although at least one of the structs should embed the provided `am.ExceptionHandler` (or provide its own). Any existing struct can be used for handlers, as long as there's no name conflict.

Example - define `FooState` and `FooEnter`

```go
type Handlers struct {
    // default handler for the build in Exception state
    *am.ExceptionHandler
}

func (h *Handlers) FooState(e *am.Event) {
    // final activation handler for Foo
}

func (h *Handlers) FooEnter(e *am.Event) bool {
    // negotiation activation handler for Foo
    return true // accept this transition by Foo
}

func main() {
    // ...
    err := mach.BindHandlers(&Handlers{})
}
```

Log output:

```
[add] Foo
[handler] FooEnter
[state] +Foo
[handler] FooState
```

Event Struct

Every handler receives a pointer to an `Event` struct, with `Name`, `Machine` and `Args`.

```go
// definition
type Event struct {
    Name    string
    Machine *Machine
    Args    A
}
// send args
mach.Add(am.S{"Foo"}, A{"test": 123})
// ...
// receive args
func (h *Handlers) FooState(e *am.Event) {
  test := e.Args["test"].(string)
}
```

Calculating Target States

Called states combined with currently active states and a relations resolver result in **target states** of a transition. This phase is **cancelable** - if **any** of the called states gets rejected, the **transition is canceled**. This isn't true for Auto states, which can be partially rejected.

Transition exposes the currently called, target and previous states using:

- `e.Transition.StatesBefore`
- `e.Transition.TargetStates`
- `e.Transition.ClockBefore()`
- `e.Transition.ClockAfter()`
- `e.Transition.Mutation.CalledStates`

```go
// machine
mach := am.New(ctx, am.Struct{
    "Foo": {
        Add: am.S{"Bar"},
    },
    "Bar": {}
}, nil)
```

```go
// ...

// handlers
func (h *Handlers) FooEnter(e *am.Event) bool {
    e.Transition.StatesBefore // ()
    e.Transition.TargetStates // (Foo Bar)
    e.Machine.Transition.CalledStates() // (Foo)

    e.Machine.Is(am.S{"Foo", "Bar"}) // false
    return true
}
func (h *Handlers) FooState(e *am.Event) {
    e.Transition.StatesBefore // ()
    e.Transition.TargetStates // (Foo Bar)
    e.Machine.Transition.CalledStates() // (Foo)

    e.Machine.Is(am.S{"Foo", "Bar"}) // true
}

// ...

// usage
mach.Add1("Foo", nil)

[add] Foo
[implied] Bar
[handler] FooEnter
FooEnter
| From: []
| To: [Foo Bar]
| Called: [Foo]
() [Bar:0 Foo:0 Exception:0]
[state] +Foo +Bar
[handler] FooState
FooState
| From: []
| To: [Foo Bar]
| Called: [Foo]
(Bar:1 Foo:1) [Exception:0]
end
(Bar:1 Foo:1) [Exception:0]
```

Negotiation Handlers

```go
// can Foo activate?
func (h *Handlers) FooEnter(e *am.Event) bool {}
```

14

```go
// can Foo de-activate?
func (h *Handlers) FooExit(e *am.Event) bool {}
// with Bar active, can Foo activate?
func (h *Handlers) BarFoo(e *am.Event) bool {}
```

Negotiation handlers `Enter` and `Exit` are called for every state which is going to be activated or de-activated. State-state handler (eg `FooBar`) are `Foo` is active and `Bar` wants to activate. They are allowed to cancel a transition by optionally returning `false`. **Negotiation handlers** are limited to read-only operations, or at least to side effects free ones. Their purpose is to make sure that final transition handlers are good to go.

```go
// negotiation handler
func (h *Handlers) ProcessingFileEnter(e *am.Event) bool {
    // read-only ops
    // decide if moving fwd is ok
    // no blocking
    // lock-free critical section
    return true
}
```

Example - rejected negotiation

```go
// machine
mach := am.New(ctx, am.Struct{
    "Foo": {
        Add: am.S{"Bar"},
    },
    "Bar": {},
}, nil)

// ...

// handlers
func (h *Handlers) FooEnter(e *am.Event) bool {
    return false
}

// ...

// usage
mach.Add1("Foo", nil) // ->am.Canceled
// () [Bar:0 Foo:0 Exception:0]
```

```
[add] Foo
[implied] Bar
[handler] FooEnter
[cancel:ad0d8] (Foo Bar) by FooEnter
```

15

```
() [Bar:0 Foo:0 Exception:0]
```

Final Handlers

```go
func (h *Handlers) FooState(e *am.Event) {}
func (h *Handlers) FooEnd(e *am.Event) {}
```

Final handlers `State` and `End` are where the main handler logic resides. After the transition gets accepted by relations and negotiation handlers, final handlers will allocate and dispose resources, call APIs, and perform other blocking actions with side effects. Just like negotiation handlers, they are called for every state which is going to be activated or de-activated. Additionally, the Self handlers are called for states which remained active.

Like any handler, final handlers cannot block the mutation. That's why they need to start a goroutine and continue their execution within it, while asserting the state context is still valid.

```go
func (h *Handlers) ProcessingFileState(e *am.Event) {
    // read & write ops
    // no blocking
    // lock-free critical section
    mach := e.Machine
    // tick-based context
    stateCtx := mach.NewStateCtx("ProcessingFile")
    go func() {
        // block in the background, locks needed
        if stateCtx.Err() != nil {
            return // expired
        }
        // blocking call
        err := processFile(h.Filename, stateCtx)
        if err != nil {
            mach.AddErr(err, nil)
            return
        }
        // re-check the tick ctx after a blocking call
        if stateCtx.Err() != nil {
            return // expired
        }
        // move to the next state in the flow
        mach.Add1("FileProcessed", nil)
    }()
}
```

Global Handlers

`AnyEnter` is the first negotiation handler and always gets executed.

```go
func (d *Debugger) AnyEnter(e *am.Event) bool {
    tx := e.Transition()

    // ...

    return true
}
```

`AnyState` is the last final handler and always gets executed for accepted transitions.

```go
func (d *Debugger) AnyState(e *am.Event) {
    tx := e.Transition()

    // redraw on auto states
    if tx.IsAuto() && tx.Accepted {
        d.updateTxBars()
        d.draw()
    }
}
```

Side effects:

- using a global handler make the "Empty" filter useless in am-dbg, as every transition always triggers a handler.

Advanced Topics

Relations

Mutations are the heartbeat of asyncmachine, while relations define the rules of the flow. Each state can have 4 types of **relations**. Each relation accepts a list of state names. Relations guarantee consistency among active states.

Relations form a multigraph (with identity edges) of state nodes, but are not Turing complete, as they guarantee termination. Only auto states trigger a single, automatic mutation attempt.

Add relation The `Add` relation tries to activate listed states, whenever the owner state gets activated.

Their activation is optional, meaning if any of those won't get accepted, the transition will still be `Executed`.

```go
// machine
mach := am.New(ctx, am.Struct{
    "Foo": {
        Add: am.S{"Bar"},
    },
```

17

```
    "Bar": {},
}, nil)

// usage
mach.Add1("Foo", nil) // ->Executed
// (Foo:1 Bar:1) [Exception:0]

[add] Foo
[implied] Bar
[state] +Foo +Bar
(Foo:1 Bar:1) [Exception:0]
```

Remove relation   The `Remove` relation prevents from activating, or deactivates listed states.

If some of the called states `Remove` other called states, or some of the active states `Remove` some of the called states, the transition will be `Canceled`.

Example of an accepted transition involving a `Remove` relation:

```
// machine
mach := am.New(ctx, am.Struct{
    "Foo": {
        Remove: am.S{"Bar"},
    },
    "Bar": {},
}, nil)

// usage
mach.Add1("Foo", nil) // ->Executed
mach.Add1("Bar", nil) // ->Executed
println(m.StringAll()) // (Foo:1) [Bar:0 Exception:0]

[add] Foo
[state] +Foo
[add] Bar
[cancel:reject] Bar
(Foo:1) [Bar:0 Exception:0]
```

Example of a canceled transition involving a `Remove` relation - some of the called states `Remove` other Called States.

```
// machine
mach := am.New(ctx, am.Struct{
    "Foo": {},
    "Bar": {
        Remove: am.S{"Foo"},
    },
}, nil)
```

```go
// usage
m.Add(am.S{"Foo", "Bar"}, nil) // ->Canceled
m.Not1("Bar") // true
// () [Foo:0 Bar:0 Exception:0]
```

```
[add] Foo Bar
[cancel:reject] Foo
() [Exception:0 Foo:0 Bar:0]
```

Example of a canceled transition involving a `Remove` relation - some of the active states `Remove` some of the called states.

```go
// machine
mach := am.New(ctx, am.Struct{
    "Foo": {},
    "Bar": {
        Remove: am.S{"Foo"},
    },
}, nil)
```

```go
// usage
mach.Add1("Bar", nil) // ->Executed
mach.Add1("Foo", nil) // ->Canceled
m.StringAll() // (Foo:1) [Bar:0 Exception:0]
```

```
[add] Bar
[state] +Bar
[add] Foo
[cancel:reject] Foo
(Bar:1) [Foo:0 Exception:0]
```

Require relation   The `Require` relation describes the states required for this one to be activated.

Example of an accepted transition involving a `Require` relation:

```go
// machine
mach := am.New(ctx, am.Struct{
    "Foo": {},
    "Bar": {
        Require: am.S{"Foo"},
    }
}, nil)
```

```go
// usage
mach.Add1("Foo", nil) // ->Executed
mach.Add1("Bar", nil) // ->Executed
```

```
// (Foo:1 Bar:1) [Exception:0]
```

```
[add] Foo
[state] +Foo
[add] Bar
[state] +Bar
(Foo:1 Bar:1) [Exception:0]
```

Example of a canceled transition involving a `Require` relation:

```
// machine
mach := am.New(ctx, am.Struct{
    "Foo": {},
    "Bar": {
        Require: am.S{"Foo"},
    },
}, nil)
```

```
// usage
mach.Add1("Bar", nil) // ->Canceled
// () [Foo:0 Bar:0 Exception:0]
```

```
[add] Bar
[reject] Bar(-Foo)
[cancel:reject] Bar
() [Foo:0 Bar:0 Exception:0]
```

After relation   The `After` relation decides about the order of execution of transition handlers. Handlers from the defined state will be executed **after** handlers from listed states.

```
// machine
mach := am.New(ctx, am.Struct{
    "Foo": {
        After: am.S{"Bar"},
    },
    "Bar": {
        Require: am.S{"Foo"},
    },
}, nil)
```

```
// ...
```

```
// handlers
func (h *Handlers) FooState(e *am.Event) {
    println("Foo")
}
func (h *Handlers) BarState(e *am.Event) {
```

```
    println("Bar")
}

// ...

// usage
m.Add(am.S{"Foo", "Bar"}, nil) // ->Executed

[add] Foo Bar
[state] +Bar +Foo
[handler] BarState
Bar
[handler] FooState
Foo
```

Waiting

Developer can subscribe to almost any state permutation using "when" methods.

```
// wait until FileDownloaded becomes active
<-mach.When1("FileDownloaded", nil)

// wait until FileDownloaded becomes inactive
<-mach.WhenNot1("DownloadingFile", args, nil)

// wait for EventConnected to be activated with an arg ID=123
<-mach.WhenArgs("EventConnected", am.A{"ID": 123}, nil)

// wait for Foo to have a tick >= 6 and Bar tick >= 10
<-mach.WhenTime(am.S{"Foo", "Bar"}, am.T{6, 10}, nil)

// wait for DownloadingFile to have a tick increased by 2 since now
<-mach.WhenTicks("DownloadingFile", 2, nil)
```

Almost all "when" methods return a shared channel which closes when an event happens (or the optionally passed context is canceled). They are used to wait until a certain moment, when we know the execution can proceed. Using "when" methods creates new channels and should be used with caution, possibly making use of the early disposal context. In the future, these channels will be reused and should scale way better.

"When" methods are:

- `Machine.When(states, ctx)`
- `Machine.WhenNot(states, ctx)`
- `Machine.When1(state, ctx)`
- `Machine.WhenNot1(state, ctx)`
- `Machine.WhenArgs(state, args, ctx)`

21

- `Machine.WhenTime(states, ctx)`
- `Machine.WhenTicks(state, ctx)`
- `Machine.WhenTicksEq(state, ctx)`
- `Machine.WhenQueueEnds(state, ctx)`
- `Machine.WhenErr(state, ctx)`

Example - waiting for states `Foo` and `Bar` to being active at the same time:

```go
// machine
mach := am.New(ctx, am.Struct{
    "Foo": {
        Add: am.S{"Bar"},
    },
    "Bar": {},
})

// ...

// usage
select {
    case <-mach.When(am.S{"Foo", "Bar"}, nil):
        println("Foo Bar")
    }
}

// ...

// state change
mach.Add1("Foo", nil)
mach.Add1("Bar", nil)
// (Foo:1 Bar:1) [Exception:0]
```

```
[add] Foo
[implied] Bar
[state] +Foo +Bar
Foo Bar
(Bar:1 Foo:1) [Exception:0]
```

Side effects:

- disposing the passed context will close a wait channel
- disposing a machine will close all wait channels

Error Handling

Considering that everything meaningful can be a state, so can errors. Every machine has a predefined `Exception` state (which is a `Multi` state), and an optional `ExceptionHandler`, which can be embedded into handler structs.

Advised error handling strategy (used by **/pkg/node**):

- create more detailed error states like `ErrNetwork`
  - with a `Require` relation to the `Exception` state
  - but without being a `Multi` state, so it has state context
- create regular sentinel errors, like `ErrRpc`
- create separate mutation function foc each sentinel error, like
  - `func AddErrWorker(event *am.Event, mach *am.Machine, err error, args am.A) error`
- one error state can be responsible for many sentinel errors

Error handling methods:

- `Machine.AddErr(error, Args)`
- `Machine.AddErrState(string, error, Args)`
- `Machine.WhenErr(ctx)`
- `Machine.Err()`
- `Machine.IsErr()`

Example - detailed error states

```go
var States = am.Struct{

    ErrWorker:  {Require: am.S{am.Exception}},
    ErrPool:    {Require: am.S{am.Exception}},

    // ...
}
```

Example - timeout flow with error handling

```go
select {
case <-time.After(10 * time.Second):
    // timeout
case <-mach.WhenErr(nil):
    // error or machine disposed
    fmt.Printf("err: %s\n", mach.Err())
case <-mach.When1("Bar", nil):
    // state Bar active
}

[add] Foo
[state] +Foo
[add] Exception
[state] +Exception
err: fake err
(Foo:1 Exception:1) [Bar:0]
```

Example - AddErrRpc wraps an error in the ErrRpc sentinel and adds to a machine as ErrNetwork

```go
func AddErrRpc(mach *am.Machine, err error, args am.A) {
    err = fmt.Errorf("%w: %w", ErrRpc, err)
    mach.AddErrState(states.BasicStates.ErrNetwork, err, args)
}
```

Side effects:

- it's not possible to use `Machine.AddErr*` methods inside `Exception*` handlers

Catching Panics

Panics are automatically caught and transformed into the `Exception` state in case they take place in the main body of any handler method. This can be disabled using `Machine.PanicToException` or `Opts.DontPanicToException`. Same goes for `Machine.LogStackTrace` and `DontLogStackTrace`, which decides about printing stack traces to the log sink.

In case of a panic inside a transition handler, the recovery flow depends on the type of the erroneous handler.

Panic in a negotiation handler

1. Cancels the whole transition.
2. Active states of the machine stay untouched.
3. Add mutation for the `Exception` state is prepended to the queue.

Panic in a final handler

1. Transition has been accepted and target states has been set as active states.
2. Not all the final handlers have been executed, so the states from non-executed handlers are removed from active states.
3. Add mutation for the `Exception` state is prepended to the queue and the integrity should be restored manually (e.g. relations, resources involved).

```go
// TestPartialFinalPanic
type TestPartialFinalPanicHandlers struct {
    *ExceptionHandler
}

func (h *TestPartialFinalPanicHandlers) BState(_ *Event) {
    panic("BState panic")
}

func TestPartialFinalPanic(t *testing.T) {
    // init
    mach := NewNoRels(t, nil)
    // () [A:0 B:0 C:0 D:0]
```

```
    // logger
    log := ""
    captureLog(t, m, &log)

    // bind handlers
    err := m.BindHandlers(&TestPartialFinalPanicHandlers{})
    assert.NoError(t, err)

    // test
    m.Add(S{"A", "B", "C"}, nil)

    // assert
    assertStates(t, m, S{"A", "Exception"})
}
```

Panic anywhere else   For places like goroutines and functions called from
the outside (e.g. request handlers), there are dedicated methods to catch panics.
They support `Exception` as well as arbitrary error states.

- `Machine.PanicToErr(args)`
- `Machine.PanicToErrState(string, args)`

```
var mach *am.Machine
func getHandler(w http.ResponseWriter, r *http.Request) {
    defer mach.PanicToErr(nil)
    // ...
}
```

Queue and History

The purpose of **asyncmachine-go** is to synchronize actions, which results in
only one handler being executed at the same time. Every mutation happening
inside the handler, will be queued and the mutation call will return `Queued`.

Queue itself can be accessed via `Machine.Queue()` and checked using
`Machine.IsQueued()`. After the execution, queue creates history, which can be
captured using a dedicated package `pkg/history`. Both sources can help to
make informed decisions based on scheduled and past actions.

- `Machine.Queue()`
- `Machine.IsQueued(mutationType, states, withoutArgsOnly,`
  `statesStrictEqual, startIndex)`
- `Machine.Transition()`
- `History.ActivatedRecently(state, duration)`
- `Machine.WhenQueueEnds()`
- `Machine.WillBe()`
- `Machine.WillBeRemoved()`

Example - handles adds a mutation to the queue, and checks it

```go
// machine
mach := am.New(ctx, am.Struct{
    "Foo": {},
    "Bar": {}
}, nil)

// ...

// handlers
func (h *Handlers) FooState(e *am.Event) {
    e.Machine.Add1("Bar", nil) // -> Queued
    e.Machine.Is1("Bar", nil) // -> false
    e.Machine.WillBe1("Bar", nil) // -> true
}

// ...

// usage
mach.Add1("Foo", nil) // -> Executed
```

```
[add] Foo
[state] +Foo
[handler] FooState
[queue:add] Bar
[postpone] queue running (1 item)
[add] Bar
[state] +Bar
```

Side effects:

- all state additions without arguments, for non-multi states, are reduced into 1 to avoid polluting the queue.

Logging

Besides inspecting methods, **asyncmachine-go** offers a very verbose logging system with 4 levels of granularity:

- `LogNothing` (default)
- `LogChanges` state changes and important messages
- `LogOps` detailed relations resolution, called handlers, queued and rejected mutations
- `LogDecisions` more verbose variant of Ops, explaining the reasoning behind
- `LogEverything` useful only for deep debugging

Example of all log levels for the same code snippet:

26

```go
// machine
mach := am.New(ctx, am.Struct{
    "Foo": {},
    "Bar": {
        Auto: true,
    },
    // disable ID logging
}, &am.Opts{DontLogID: true})
m.SetLogLevel(am.LogOps)

// ...

// handlers
func (h *Handlers) FooState(e *am.Event) {
    // empty
}
func (h *Handlers) BarEnter(e *am.Event) bool {
    return false
}

// ...

// usage
mach.Add1("Foo", nil) // Executed
```

- log level LogChanges

```
[state] +Foo
```

- log level LogOps

```
[add] Foo
[state] +Foo
[handler] FooState
[auto] Bar
[handler] BarEnter
[cancel:4a0bc] (Bar Foo) by BarEnter
```

- log level LogDecisions

```
[add] Foo
[state] +Foo
[handler] FooState
[auto] Bar
[add:auto] Bar
[handler] BarEnter
[cancel:2daed] (Bar Foo) by BarEnter
```

- log level LogEveryting

```
[start] handleEmitterLoop Handlers
[add] Foo
[emit:Handlers:d7a58] AnyFoo
[emit:Handlers:d32cd] FooEnter
[state] +Foo
[emit:Handlers:aa38c] FooState
[handler] FooState
[auto] Bar
[add:auto] Bar
[emit:Handlers:f353d] AnyBar
[emit:Handlers:82e34] BarEnter
[handler] BarEnter
[cancel:82e34] (Bar Foo) by BarEnter
```

Logging-related methods:

- `Machine.Log(msg string, args ...any)`
- `Machine.SetLoggerSimple(sprintf, level LogLevel)`
- `Machine.SetLoggerEmpty(level LogLevel)`
- `Machine.SetLogLevel(level LogLevel)`
- `Machine.GetLogLevel() LogLevel`
- `Machine.SetLogger(fn Logger)`
- `Machine.GetLogger(fn Logger): *Logger`
- `Machine.SetLogArgs(mapper LogArgsMapper)`
- `Machine.GetLogArgs() LogArgsMapper`

Customizing Logging   Example - binding to a test logger

```go
// test log with the minimal log level
mach.SetLoggerSimple(t.Logf, am.LogChanges)
```

Example - logging mutation arguments

```go
// include some args in the log and traces
mach.SetLogArgs(am.NewArgsMapper([]string{"id", "name"}, 20))
```

Example - custom logger

```go
// max out the log level
mach.SetLogLevel(am.LogEverything)
// level based dispatcher
mach.SetLogger(func(level LogLevel, msg string, args ...any) {
    if level > am.LogChanges {
        customLogDetails(msg, args...)
        return
    }
    customLog(msg, args...)

})
```

Debugging

asyncmachine-go comes with a `TUI debugger (am-dbg)`, which makes it very easy to hook into any state machine on a transition's step-level, and retain state machine's log. It also combines very well with the Golang debugger when stepping through code.

Environment variables used for debugging can be found in config/env/README.md.

Steps To Debug

1. Install `go install github.com/pancsta/asyncmachine-go/tools/am-dbg@latest`
2. Run `am-dbg`
3. Enable telemetry
4. Run your code with `env AM_DEBUG=1` to increase timeouts and enable stack traces

Enabling Telemetry   Telemetry for **am-dbg** can be enabled manually using /pkg/telemetry, or with a helper from **/pkg/helpers**.

- `MachDebug(*am.Machine, string, am.LogLevel, bool)`
- `MachDebugt(*am.Machine, bool)`

Example - enable telemetry manually

```
import "github.com/pancsta/asyncmachine-go/pkg/telemetry"
// ...
err := telemetry.TransitionsToDBG(mach, "")
```

Example - enable telemetry using helpers

```
// AM_DBG_ADDR=localhost:6831
// AM_LOG=2

import amhelp "github.com/pancsta/asyncmachine-go/pkg/helpers"

// debug
amhelp.MachDebugEnv(mach)
```

Breakpoints   // TODO // AddBreakpoint adds a breakpoint for an outcome of mutation (added and // removed states). Once such mutation happens, a log message will be printed // out. You can set an IDE's breakpoint on this line and see the mutation's sync // stack trace. When Machine.LogStackTrace is set, the stack trace will be // printed out as well. Many breakpoints can be added, but none removed.

```
worker.AddBreakpoint(am.S{"Healthcheck"}, nil)
```

- Machine.AddBreakpoint(added S, removed S)
- Log: `[breakpoint] Machine.breakpoint`

Typesafe States

While it's perfectly possible to operate on pure string names for state names (e.g. for prototyping), it's not type safe, leads to errors, doesn't support godoc, nor looking for references in IDEs. `/tools/cmd/am-gen` will generate a conventional type-safe states file, along with inheriting from predefined state machines. It also aliases common functions to manipulates state lists, relations, and structure. After the initial bootstrapping, the file should be edited manually.

Example - using am-gen to bootstrap a states file

```go
package states

import (
    am "github.com/pancsta/asyncmachine-go/pkg/machine"
    ss "github.com/pancsta/asyncmachine-go/pkg/states"
)

// MyMachStatesDef contains all the states of the MyMach state machine.
type MyMachStatesDef struct {
    *am.StatesBase

    // State1 is the first state
    State1 string
    // State2 is the second state
    State2 string

}

// MyMachGroupsDef contains all the state groups MyMach state machine.
type MyMachGroupsDef struct {
}

// MyMachStruct represents all relations and properties of MyMachStates.
var MyMachStruct = am.Struct{

    ssM.State1: {},
    ssM.State2: {},
}

// EXPORTS AND GROUPS

var (
    ssM = am.NewStates(MyMachStatesDef{})
    sgM = am.NewStateGroups(MyMachGroupsDef{})

    // MyMachStates contains all the states for the MyMach machine.
```

```
    MyMachStates = ssM
    // MyMachGroups contains all the state groups for the MyMach machine.
    MyMachGroups = sgM
)
```

States are commonly aliased as `ss` (first-last rune), as they are constantly being referenced, while imports of external states files have the package name added, eg `ssrpc`. It's also crucial to "verify" states, as map keys have a random order.

Example - importing a states file

```
import (
    am "github.com/pancsta/asyncmachine-go/pkg/machine"

    "github.com/owner/repo/states"
)

var ss := states.MyMachStates

// ...

mach := am.New(ctx, states.MyMachStruct, nil)
err := mach.VerifyStates(ss.Names())
mach.Add1(ss.State1, nil)
```

Typesafe Arguments

The default format of arguments in asyncmachine is `map[string]any`, which is very KISS, but not very practical long term. Because of that, it's advised to structure arguments into a single struct and Parse-Pass helpers, ideally with a package namespace.

Example - define pkg-level args and helpers (from `pkg/node`)

```
// A is a struct for node arguments. It's a typesafe alternative to am.A.
type A struct {
    Id string
    PublicAddr string
    LocalAddr string
}

// ParseArgs extracts A from [am.Event.Args]["am_node"].
func ParseArgs(args am.A) *A {
    if r, _ := args["am_node"].(*ARpc); r != nil {
        return amhelp.ArgsToArgs(r, &A{})
    } else if r, ok := args["am_node"].(ARpc); ok {
        return amhelp.ArgsToArgs(&r, &A{})
    }
```

```go
    a, _ := args["am_node"].(*A)
    return a
}

// Pass prepares [am.A] from A to pass to further mutations.
func Pass(args *A) am.A {
    return am.A{"am_node": args}
}
```

Example - a handler parses, uses and passes arguments further.

```go
func (s *Supervisor) ForkingWorkerState(e *am.Event) {
    args := ParseArgs(e.Args)
    b := args.Bootstrap
    argsOut := &A{Bootstrap: b}

    // ...

    // err
    if err != nil {
        AddErrWorker(s.Mach, err, Pass(argsOut))
        return
    }

    // ...

    // next
    s.Mach.Add1(ssS.AwaitingWorker, Pass(argsOut))
}
```

Arguments Subtypes   Sometimes it's necessary to have move then 1 set of arguments per package (eg 1 for RPC and 1 local). Embedding is one option, and using `ArgsToArgs` another one. It will copy overlapping fields between both arguments structs.

```go
// APublic is a subset of A, that exposes only public addresses.
type APublic struct {
    PublicAddr string
}

// PassRpc prepares [am.A] from A, according to APublic.
func PassPublic(args *A) am.A {
    return am.A{"am_node": amhelp.ArgsToArgs(args, &APublic{})}
}

// ...
```

```go
// this will only pass "PublicAddr", with other fields removed
mach.Add1("WorkerAddr", PassPublic(&A{
    LocalAddr:  w.LocalAddr,
    PublicAddr: w.PublicAddr,
    Id:         w.Mach.ID,
}))
```

Arguments Logging   Typesafe arguments can be easily extracted for logging via `Machine.SetLogArgs`. See `amhelp.ArgsToLogMap` from `pkg/helpers`.

```go
type A struct {
    // tag for logging
    Id string `log:"id"`
}
```

// TODO merging log args

```go
func LogArgs(args am.A) map[string]string {
    a1 := amnode.ParseArgs(args)
    a2 := ParseArgs(args)
    if a1 == nil && a2 == nil {
        return nil
    }

    return am.AMerge(amhelp.ArgsToLogMap(a1), amhelp.ArgsToLogMap(a2))
}
```

Tracing and Metrics

Asyncmachine offers several telemetry exporters for logging, tracing, and metrics. Please refer to `pkg/telemetry` for detailed information. There're traceable versions of mutation methods available, which accept `*am.Event` as the first param and decorate telemetry messages with MachineID, TransitionID, and Machine Time as the source. It's currently supported by the am-dbg debugger.

// TOOD list EvAdd, EvAdd1, EvRemove, . . . +example

Optimizing Data Input

It's a good practice to batch frequent operations, so the relation resolution and transition lifecycle doesn't execute for no reason. It's especially true for network packets. Below a simple debounce with a queue.

Example - batch data into a single transition every 1s

```go
var debounce = time.Second

var queue []*Msg
var queueMx sync.Mutex
```

33

```go
var scheduled bool

func Msg(msgTx *Msg) {
    queueMx.Lock()
    defer queueMx.Unlock()

    if !scheduled {
        scheduled = true
        go func() {
            // wait some time
            time.Sleep(debounce)

            queueMx.Lock()
            defer queueMx.Unlock()

            // add in bulk
            mach.Add1("Msgs", am.A{"msgs": queue})
            queue = nil
            scheduled = false
        }()
    }
    // enqueue
    queue = append(queue, msgTx)
}
```

Disposal and GC

// TODO

- wait contexts
- Dispose
- WhenDisposed
- RegisterDisposalHandler

Dynamically Generating States

// TODO

- Machine.SetStrcut

Cheatsheet

- **State**: main entity of the machine, higher-level abstraction of a meaningful workflow step
- **Active states**: states currently activated in the machine, `0-n` where `n == len(states)`
- **Called states**: states passed to a mutation method, explicitly requested

- **Target states**: states after resolving relations, based on previously active states, about to become new active states
- **Mutation**: change to currently active states, created by mutation methods
- **Transition**: container struct for a mutation, handles relations
- **Accepted transition**: transition which mutation has passed negotiation and relations
- **Canceled transition**: transition which mutation has NOT passed negotiation or relations
- **Queued transition**: transition which couldn't execute immediately, as another one was in progress, and was added to the queue instead
- **Transition handlers**: methods defined on a handler struct, which are triggered during a transition
- **Negotiation handlers**: handlers executed as the first ones, used to make a decision if the transition should be accepted
- **Final handlers**: handlers executed as the last ones, used for operations with side effects

Other sources

- examples
- diagrams
- readme
- cookbook

Packages

asyncmachine-go provides additional tooling and extensions in the form of separate packages, available at `/pkg` and `/tools`.

Each of them has a readme with examples:

- `/pkg/helpers` Useful functions when working with async state machines.
- `/pkg/history` History tracking and traversal.
- `/pkg/node` Distributed worker pools with supervisors.
- `/pkg/rpc` Remote state machines, with the same API as local ones.
- `/pkg/states` Reusable state definitions and piping.
- `/pkg/telemetry` Telemetry exporters for metrics, traces, and logs.
- `/pkg/pubsub` Decentralized PubSub based on libp2p gossipsub.
- `/tools/cmd/am-dbg` am-dbg is a multi-client TUI debugger.
- `/tools/cmd/am-gen` am-gen generates states files and Grafana dashboards.
- `/tools/cmd/am-vis` Generates diagrams of interconnected state machines.
- `/tools/cmd/arpc` Network-native REPL and CLI.