Machine deletion phase hooks

Summary

Defines a set of annotations that can be applied to a machine which affect the linear progress of a machine's lifecycle. These annotations are optional and may be applied during machine creation, sometime after machine creation by a user, or sometime after machine creation by another controller or application.

Motivation

- Allow custom and 3rd party components to interact easily pause the reconciliation of a machine in order to take some desired action.

Goals

- Define an initial set of hook points.
- Define an initial set and form of related annotations.

Non-Goals/Future Work

- Create an exhaustive list of hooks; we can add more over time.
- Dictate behaviors of controllers that respond to the hooks.
- Implement ordering in the machine-controller.
- Require anyone use these hooks for normal machine operations, these are strictly optional and for custom integrations only.

Proposal

- Utilize annotations to implement lifecycle hooks.
- Each **lifecycle point** can have 0 or more hooks.
- Hooks do not enforce ordering.
- Hooks found during machine reconciliation effectively pause reconciliation until all hooks for that **lifecycle point** are gone.

User Stories

Story 1

(pre-delete) As an operator, I would like to have the ability to perform different actions between the time a machine is marked deleted in the api and the time the machine is deleted from the cloud.

One use-case is when replacing a control plane machine, ensure a new control plane machine has been successfully created and joined to the cluster before removing the instance for the deleted machine. This might be useful in case there are disruptions during replacement and we need the disk of the existing instance to perform some disaster recovery operation

Story 2

(pre-drain) As an operator, I want the ability to utilize my own draining controller instead of the logic built into the machine-controller. This will allow me better flexibility and control over the lifecycle of workloads on each node.

Implementation Details/Notes/Constraints

For each defined lifecycle point, one or more hooks may be applied as an annotation to the machine object. These annotations will pause reconciliation of a machine object until all hooks are resolved for that lifecycle point. The hooks should be managed by an **Owning Controller** or other external application, or manually created and removed by an administrator.

Lifecycle Points

- pre-drain
    - Hooks defined at this point will prevent the machine-controller from draining a node after the machine-object has been marked for deletion until the hooks are removed.
- Pre-delete
    - Hooks defined at this point will prevent the machine-controller from removing the instance in the cloud provider until the hooks are removed.

Annotation Form

<lifecycle-point>.hook.machine.cluster-api.x-k8s.io/<hook-name>: <owner/creator>

lifecycle-point: This is the point in the lifecycle of reconciling a machine the annotation will have effect and pause the machine-controller.

hook-name: Each hook should have a unique and descriptive name that describes in 1-3 words what the intent/reason for the hook is. Each hook name should be unique and managed by a single entity.

owner : (Optional) Some information about who created or is otherwise in charge of managing the annotation. This might be a controller or a username to indicate an administrator applied the hook directly.

Annotation Examples

```
pre-drain.hook.machine.cluster-api.x-k8s.io/migrate-important-app:
my-app-migration-controller

pre-delete.hook.machine.cluster-api.x-k8s.io/preserve-instance:
my-instance-replacement-controller

pre-delete.hook.machine.cluster-api.x-k8s.io/never-delete: cluster-admin
```

Owning Controller Design

Owning Controller is the component that manages a particular lifecycle hook.

Owning Controllers **must:**

- Watch machine objects and determine when an appropriate action must be taken.
- After completing the desired hook action, remove the hook annotation.

Owning Controllers **may:**

- Watch machine objects and add a hook annotation as desired by the operator
- After completing a hook, set an addition annotation that indicates this controller has finished for dependency ordering

Risks and Mitigations

- Annotation keys must conform to length limits: https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/#syntax-and-character-set
- Requires well-behaved controllers and admins to keep things humming smoothly. Would be easy to disrupt machines with poor configuration

Alternatives

Custom Machine Controller

Require advanced users to fork and customize. This can already be done if someone chooses, so not much of a solution.

Finalizers

We define additional finalizers, but this really only implies the deletion lifecycle point. Finalizers also have no great way to enforce ordering, which could be done with other methods.

Status Field

Harder for users to modify or set hooks during machine creation. How would a user remove a hook if a controller that is supposed to remove it is misbehaving? We'd probably need an annotation like 'skip-hook-xyz' or similar and that seems redundant to just using annotations in the first place

Spec Field

We probably don't want other controllers dynamically adding and removing spec fields on an object. Firstly, it's not very declarative to utilize spec fields in that way.

CRDs

Seems like we'd need to sync information to and from a CR. There are different approaches to CRDs (1-to-1 mapping machine to CR, match labels, present/absent vs status fields) that each have their own drawbacks and are more complex to define and configure.

Upgrade Strategy

Nothing defined here should directly impact upgrades other than defining hooks that impact creation/deletion of a machine, generally.

Additional Details

Test Plan [optional]

Note: *Section not required until targeted at a release.*

Consider the following in developing a test plan for this enhancement:

- Will there be e2e and integration tests, in addition to unit tests?
- How will it be tested in isolation vs with other components?

No need to outline all of the test cases, just the general strategy. Anything that would count as tricky in the implementation and anything particularly challenging to test should be called out.

All code is expected to have adequate tests (eventually with coverage expectations). Please adhere to the Kubernetes testing guidelines when drafting this test plan.

Graduation Criteria [optional]

Note: *Section not required until targeted at a release.*

Define graduation milestones.

These may be defined in terms of API maturity, or as something else. Initial proposal should keep this high-level with a focus on what signals will be looked at to determine graduation.

Consider the following in developing the graduation criteria for this enhancement:

- Maturity levels (`alpha`, `beta`, `stable`)
- Deprecation policy

Clearly define what graduation means by either linking to the API doc definition, or by redefining what graduation means.

In general, we try to use the same stages (alpha, beta, GA), regardless how the functionality is accessed.

Version Skew Strategy [optional]

If applicable, how will the component handle version skew with other components? What are the guarantees? Make sure this is in the test plan.

Consider the following in developing a version skew strategy for this enhancement:

- Does this enhancement involve coordinating behavior in the control plane and in the kubelet? How does an n-2 kubelet without this feature available behave when this feature is used?
- Will any other components on the node change? For example, changes to CSI, CRI or CNI may require updating that component before the kubelet.

Implementation History

TODO

Open Questions

Does this mean the annotation is removed?

Maybe we should keep it, but make the value the completion timestamp?

Or have multiple annotations for a single hook-name to record start time, completion time, status (i.e. error/failed)?

Do hooks have timeouts?

What happens if a hook fails?

Do we need fail-open & fail-closed like Kubernetes has for its webhooks?

Also, if annotation is being removed after completion - how can one know what's the right timing/state to re-add it to be triggered in the next time this operation is running?

It needs to re-add it only after all existing hooks were completed, but you can't tell if existing hooks are running, or just waiting for the next run.