dataflow/operators/generic: infrastructure

Created	@September 12, 2021 7:24 PM
Tags	

Capability

`Capability`

An output port of an operator has to hold capabilities in order to send messages. An output port can send messages with timestamp `t`, only if the output port holds a capability with timestamp `t' <= t`.

```
pub trait CapabilityTrait<T: Timestamp> {
    /// The timestamp associated with the capability.
    fn time(&self) -> &T;
    fn valid_for_output(&self, query_buffer: &Rc<RefCell<ChangeBatch<T>>>) -> bool;
}

pub struct Capability<T: Timestamp> {
    time: T,
    // internal is the system-wide view of all capabilities (of different timestamps and their counts) at this location (operator's output
    internal: Rc<RefCell<ChangeBatch<T>>>,
}
```

Here, `internal` is just a shared `ChangeBatch` passed to the capability, the dataflow system uses the `ChangeBatch` to record all outstanding capabilities of different timestamps correspond to this output port. **Note that capabilities are bounded to a location (an operator's output port). Without location, the capability is meaningless.**

`valid_for_output()` checks whether `query_buffer` refers to the same underlaying `ChangeBatch`. If so, it indicates `query_buffer` corresponds to the same output port (location) this capability bounded to. So the implementation for `valid_for_output` is simple:

```
impl<T: Timestamp> CapabilityTrait<T> for Capability<T> {
    fn time(&self) -> &T { &self.time }
    fn valid_for_output(&self, query_buffer: &Rc<RefCell<ChangeBatch<T>>>) -> bool {
        // if the query_buffer refers to the same underlaying ChangeBatch
        // it means the current Capacity is bounded to the same location (output port) as query_buffer points to
        // we would have the ability to output messages with timestamp T
        Rc::ptr_eq(&self.internal, query_buffer)
    }
}
```

A capability can generate new capability with timestamp greater or equal than its (Since an operator (output port) can output any messages with timestamp greater than what it currently holds):

```
pub fn delayed(&self, new_time: &T) -> Capability<T> {
        if !self.time.less_equal(new_time) {
            panic!("Attempted to delay {:?} to {:?}, which is not `less_equal` the capability's time.", self, new_time);
        }
        mint(new_time.clone(), self.internal.clone())
    }
```

`Capability` will inform the system when it is dropped.

```
impl<T: Timestamp> Drop for Capability<T> {
    #[inline]
    fn drop(&mut self) {
        self.internal.borrow_mut().update(self.time.clone(), -1);
    }
}
```

`CapabilityRef`

`CapabilityRef` does not actually own capabilities, but it instead refers to multiple output ports of an operator. From `CapabilityRef`, we can create new (delayed) `Capability` for any of the operator's output port. This is also why we need a vector for `internal`.

```
pub struct CapabilityRef<'cap, T: Timestamp+'cap> {
    time: &'cap T,
    // we have a vector here, since CapabilityRef can refer to multiple output ports of the same operator
    internal: Rc<RefCell<Vec<Rc<RefCell<ChangeBatch<T>>>>>>,
}
```

`ActivateCapability`

This is a wrapper for `Capability`. It activates an address when it is dropped.

```
pub struct ActivateCapability<T: Timestamp> {
    // it holds a Capability
    pub(crate) capability: Capability<T>,
    pub(crate) address: Rc<Vec<usize>>,
    pub(crate) activations: Rc<RefCell<Activations>>,
}

impl<T: Timestamp> Drop for ActivateCapability<T> {
    // activate the address when dropped
    fn drop(&mut self) {
        self.activations.borrow_mut().activate(&self.address[..]);
    }
}
```

`CapabilitySet`

```
#[derive(Clone, Debug)]
pub struct CapabilitySet<T: Timestamp> {
    elements: Vec<Capability<T>>,
}
```

It maintains a "frontier" of capabilities, which holds a set of "minimal" incomparable capabilities. Just look at the `insert` method:

```
pub fn insert(&mut self, capability: Capability<T>) {
        // we maintain a "frontier" of capabilities
        // CapabilitySet holds a set of "minimal" incomparable capabilities
        if !self.elements.iter().any(|c| c.less_equal(&capability)) {
            self.elements.retain(|c| !capability.less_equal(c));
            self.elements.push(capability);
        }
    }
```

Input and Output Handles

Input Handle

```
pub struct InputHandle<T: Timestamp, D, P: Pull<Bundle<T, D>>> {
    pull_counter: PullCounter<T, D, P>,
    // internal should be the internals (capabilities information storage) for every output port of the operator
    internal: Rc<RefCell<Vec<Rc<RefCell<ChangeBatch<T>>>>>>,
    logging: Option<Logger>,
}
```

Input handles takes a `PullCounter` to receive message (data records). `internal` refers to the system-wide view of this operator's outstanding capabilities (for every output port of the operator).

`next()` method reads (pulls) messages from `pull_counter`, and then generator a `CapabilityRef` to this operator (without creating capabilities) with the timestamp corresponds to the received message.

```
pub fn next(&mut self) -> Option<(CapabilityRef<T>, RefOrMut<Vec<D>>)> {
        let internal = &self.internal;
        // pull the data from the puller
        // Bundle<T, D>
        self.pull_counter.next().map(|bundle| {
            match bundle.as_ref_or_mut() {
                // call mint_capability_ref() to create a capability reference of output ports with capability corresponds to the input mes
                // this function does not create new capability
                RefOrMut::Ref(bundle) => {
                    (mint_capability_ref(&bundle.time, internal.clone()), RefOrMut::Ref(&bundle.data))
                },
                RefOrMut::Mut(bundle) => {
                    (mint_capability_ref(&bundle.time, internal.clone()), RefOrMut::Mut(&mut bundle.data))
                },
            }
        })
    }
```

We have another `FrontieredInputHandle`, which just wraps an input handle and holds a reference to the input port's frontier.

```
pub struct FrontieredInputHandle<'a, T: Timestamp, D: 'a, P: Pull<Bundle<T, D>>+'a> {
    /// The underlying input handle.
    pub handle: &'a mut InputHandle<T, D, P>,
    /// The frontier as reported by timely progress tracking.
    // just stores a reference to the frontier
    // the frontier is provided by timely dataflow's progress tracking module
    pub frontier: &'a MutableAntichain<T>,
}
```

`InputHandle` should be bounded to **an input port**.

Output Handle

```
pub struct OutputHandle<'a, T: Timestamp, D: 'a, P: Push<Bundle<T, D>>+'a> {
    // holds only reference to the (buffered) pusher and the capability internal (storage) of the output port
    push_buffer: &'a mut Buffer<T, D, PushCounter<T, D, P>>,
    internal_buffer: &'a Rc<RefCell<ChangeBatch<T>>>,
}

impl<'a, T: Timestamp, D, P: Push<Bundle<T, D>>> OutputHandle<'a, T, D, P> {
    /// Obtains a session that can send data at the timestamp associated with capability `cap`.
    pub fn session<'b, C: CapabilityTrait<T>>(&'b mut self, cap: &'b C) -> Session<'b, T, D, PushCounter<T, D, P>> where 'a: 'b {
        // open a session if a valid capability is provided
        assert!(cap.valid_for_output(&self.internal_buffer), "Attempted to open output session with invalid capability");
        self.push_buffer.session(cap.time())
    }
}
```

Through the output handle (which is bounded to **an output port**), we can allocate a session that send data with a given timestamp, only if the provided capability is valid.

Notifications

Notificator tracks requests for notification and delivers available notifications.

`FrontierNotificator`

```
pub struct FrontierNotificator<T: Timestamp> {
    pending: Vec<(Capability<T>, u64)>,
    available: ::std::collections::BinaryHeap<OrderReversed<T>>,
}
```

`FrontierNotificator` tracks the pending notifications, that is, the timestamps that we need to deliver notifications. `avaiable` stores the timestamps that has no frontier elements preventing the notification delivery, we can already deliver the notification with the timestamp.

Provided with a slice of frontiers (one for each input port of the operator), we check the frontiers to see whether there is pending notifications that can be delivered, that is, no frontier element with timestamp ≤ the timestamp of the notification.

```
/// Enables pending notifications not in advance of any element of `frontiers`.
    pub fn make_available<'a>(&mut self, frontiers: &'a [&'a MutableAntichain<T>]) {

        // By invariant, nothing in self.available is greater_equal anything in self.pending.
        // It should be safe to append any ordered subset of self.pending to self.available,
        // in that the sequence of capabilities in self.available will remain non-decreasing.

        // move the capacity (or just say timestamp) that we can notify (no frontier elements stopping us)
        // from pending to available
        if !self.pending.is_empty() {
            // sort & merge the capabilities (timestamps) to notify
            self.pending.sort_by(|x,y| x.0.time().cmp(y.0.time()));
            for i in 0 .. self.pending.len() - 1 {
                if self.pending[i].0.time() == self.pending[i+1].0.time() {
                    self.pending[i+1].1 += self.pending[i].1;
                    self.pending[i].1 = 0;
                }
            }
            // the count does not matter actually, as long as it is >= 1, we notify
            self.pending.retain(|x| x.1 > 0);

            for i in 0 .. self.pending.len() {
                // multiple frontiers corresponds to all the input ports of the operator (to be notified)?
                // TODO for code reading: confirm this statement
                if frontiers.iter().all(|f| !f.less_equal(&self.pending[i].0)) {
                    // TODO : This clones a capability, whereas we could move it instead.
                    self.available.push(OrderReversed::new(self.pending[i].0.clone()));
                    self.pending[i].1 = 0;
                }
            }
            self.pending.retain(|x| x.1 > 0);
        }
    }
```

The counts in the `pending` after sorting and merging are not actually important. As long as the count ≥ 1, it means that we need to deliver a notification for this timestamp. For the timestamp with a count > 1 we still only need to deliver a single notification for this timestamp.

`next()` method returns the next available timestamp to deliver notification (encapsulated in a `Capability`). It also removes repeated timestamps in `self.available` to avoid repeated notifications.

```
pub fn next<'a>(&mut self, frontiers: &'a [&'a MutableAntichain<T>]) -> Option<Capability<T>> {
        if self.available.is_empty() {
            self.make_available(frontiers);
        }
        // we have applied OrderReversed in self.available to impose an increasing time order.
        self.available.pop().map(|front| {
            // remove repeated timestamps to avoid repeated notifications
            while self.available.peek() == Some(&front) { self.available.pop(); }
            front.element
        })
    }
```

Stream

Stream serves as a handle bounded to an output port of an operator.

```
pub struct Stream<S: Scope, D> {
    /// The progress identifier of the stream's data source.
    name: Source,
    /// The `Scope` containing the stream.
    scope: S,
    /// Maintains a list of Push<Bundle<T, D>> interested in the stream's output.
    ports: TeeHelper<S::Timestamp, D>,
}
```

It maintains a list of pushers, which are the pushers (of the input ports the output port connects to) we need to push the output of the stream (output port) to.

OperatorCore

OperatorCore is the "actual" operator, it implements `Operator` trait, wraps a logic that takes a reference to the OperatorCore's `SharedProgress`. When the operator is scheduled, the logic is executed. A bool value is returned, indicating whether the operator is finished.

```
// OperatorCore packs the logic of
// which describe the actual behavior of the operator
// the logic can access the SharedProgress of the operator
// inspect the inputs frontier, output port capabilities
// consumed messages, produced messages
struct OperatorCore<T, L>
where
    T: Timestamp,
    L: FnMut(&mut SharedProgress<T>)->bool+'static,
{
    shape: OperatorShape,
    address: Vec<usize>,
    // logic takes the reference of the SharedProgress
    // it also possess the ability to change the SharedProgress
    // returns a bool, indicate whether the operation is completed
    logic: L,
    shared_progress: Rc<RefCell<SharedProgress<T>>>,
    activations: Rc<RefCell<Activations>>,
    // the internal summaries between inputs ports and output ports are stored here
    summary: Vec<Vec<Antichain<T::Summary>>>,
}
```

Default capabilities will be set:

```
fn get_internal_summary(&mut self) -> (Vec<Vec<Antichain<T::Summary>>>, Rc<RefCell<SharedProgress<T>>>) {

        // Request the operator to be scheduled at least once.
        self.activations.borrow_mut().activate(&self.address[..]);

        // by default, we reserve a capability for each output port at `Default::default()`.
        // for each output port, initially gives it the capability to output messages with any timestamp
        // e.g., capability with T=0
        // and for each worker we give an capability
        self.shared_progress
            .borrow_mut()
            .internals
            .iter_mut()
            .for_each(|output| output.update(T::minimum(), self.shape.peers as i64));

        (self.summary.clone(), self.shared_progress.clone())
    }
```

RawOperatorBuilder

OperatorBuilder defines and connects input and output ports of the operators.

`new_input_connection()`

It takes a stream and a ParallelizationContract. The pact is used to allocate a channel between the stream output and this input node. The stream takes the pusher to push its outputs. The puller (to receive data from the output port) is returned.

```
pub fn new_input_connection<D: Data, P>(&mut self, stream: &Stream<G, D>, pact: P, connection: Vec<Antichain<<G::Timestamp as Timestamp>::S
    where
        P: ParallelizationContract<G::Timestamp, D> {
```

```
        let channel_id = self.scope.new_identifier();
        let logging = self.scope.logging();
        // construct a channel between the output port (where it connected to) and this input port.
        let (sender, receiver) = pact.connect(&mut self.scope, channel_id, &self.address[..], logging);
        let target = Target::new(self.index, self.shape.inputs);
        // connect the stream
        // i.e., now the stream will push the data it received (pushed to the Tee)
        // to the sender we just allocated
        // so that receiver can receive data
        stream.connect_to(target, sender, channel_id);

        self.shape.inputs += 1;
        assert_eq!(self.shape.outputs, connection.len());
        // push the summaries from this input port to all existing output ports
        self.summary.push(connection);

        receiver
    }
```

`new_output_connection()`

It returns a pusher and a stream. Produced data should be pushed to the pusher, a `Tee`, which then pushes the records to all the input ports the output port connects to.

```
pub fn new_output_connection<D: Data>(&mut self, connection: Vec<Antichain<<G::Timestamp as Timestamp>::Summary>>) -> (Tee<G::Timestamp, D>

        // we create a Tee to enable simultaneously push to multiple downstream input ports
        let (targets, registrar) = Tee::<G::Timestamp,D>::new();
        let source = Source::new(self.index, self.shape.outputs);
        // we can connect the input ports to the stream
        // the input port will create a pusher and puller
        // the input port then uses the puller to receive data
        // the pusher is then added to the Tee to let the output stream pushes the outputs to
        let stream = Stream::new(source, registrar, self.scope.clone());

        self.shape.outputs += 1;
        assert_eq!(self.shape.inputs, connection.len());
        // add summaries from existing input ports to this new output port
        for (summary, entry) in self.summary.iter_mut().zip(connection.into_iter()) {
            summary.push(entry);
        }

        // return the pusher and the stream
        // we must push the data to output into the pusher
        (targets, stream)
    }
```

OperatorBuilder

This struct packs the RawOperatorBuilder with progress information (`frontier, consumed, internal,produced`)

`new_input_connection()`

It creates the frontier / consumed storage for the input, `consumed` is filled by the `PullCounter`. It also returns a `InputHandle`

```
pub fn new_input_connection<D: Data, P>(&mut self, stream: &Stream<G, D>, pact: P, connection: Vec<Antichain<<G::Timestamp as Timestamp>::S
        where
            P: ParallelizationContract<G::Timestamp, D> {

        let puller = self.builder.new_input_connection(stream, pact, connection);

        let input = PullCounter::new(puller);
        self.frontier.push(MutableAntichain::new());
        self.consumed.push(input.consumed().clone());

        new_input_handle(input, self.internal.clone(), self.logging.clone())
    }
```

`new_output_connection()`

It creates the `produced` progress `ChangeBatch` for the output port. It returns a `OutputWrapper`.

```
pub fn new_output_connection<D: Data>(&mut self, connection: Vec<Antichain<<G::Timestamp as Timestamp>::Summary>>) -> (OutputWrapper<G::Tim

        let (tee, stream) = self.builder.new_output_connection(connection);

        let internal = Rc::new(RefCell::new(ChangeBatch::new()));
        self.internal.borrow_mut().push(internal.clone());

        let mut buffer = PushBuffer::new(PushCounter::new(tee));
        self.produced.push(buffer.inner().produced().clone());

        (OutputWrapper::new(buffer, internal), stream)
    }
```

`build_reschedule()`

It calls the RawOperatorBuilder's build method, but wraps the logic to drain the frontier updates from the `SharedProgress` to provide it to the logic, pushes `internal`, `produceds`, `consumed` to the `SharedProgress`.

```
pub fn build_reschedule<B, L>(self, constructor: B)
    where
        B: FnOnce(Vec<Capability<G::Timestamp>>) -> L,
        L: FnMut(&[MutableAntichain<G::Timestamp>])->bool+'static
    {
        // create capabilities, discard references to their creation.
        let mut capabilities = Vec::with_capacity(self.internal.borrow().len());
        for output_index in 0  .. self.internal.borrow().len() {
            let borrow = &self.internal.borrow()[output_index];
            capabilities.push(mint_capability(G::Timestamp::minimum(), borrow.clone()));
            // Discard evidence of creation, as we are assumed to start with one.
            borrow.borrow_mut().clear();
        }

        let mut logic = constructor(capabilities);

        // self_frontier, self_consumed, self_internal, self_produced
        // TAKES OWNERSHIP of frontier, consumed, internal, produced
        let mut self_frontier = self.frontier;
        let self_consumed = self.consumed;
        let self_internal = self.internal;
        let self_produced = self.produced;

        // raw_logic wraps the logic with progress information
        let raw_logic =
        move |progress: &mut SharedProgress<G::Timestamp>| {
            // NOW THE CLOSURE TAKES OWNERSHIP of frontier, consumed, internal, produced

            // the frontier change is reported by the parent scope
            // drain frontier changes from SharedProgress -> self.frontier
            for index in 0 .. progress.frontiers.len() {
                self_frontier[index].update_iter(progress.frontiers[index].drain());
            }

            // invoke supplied logic
            // the logic is provided with the updated frontier
            let result = logic(&self_frontier[..]);

            // move batches of consumed changes.
            // self_consumed -> SharedProgress
            for index in 0 .. progress.consumeds.len() {
                self_consumed[index].borrow_mut().drain_into(&mut progress.consumeds[index]);
            }

            // move batches of internal changes.
            // self_internal -> SharedProgress
            let self_internal_borrow = self_internal.borrow_mut();
            for index in 0 .. self_internal_borrow.len() {
                let mut borrow = self_internal_borrow[index].borrow_mut();
                progress.internals[index].extend(borrow.drain());
            }

            // move batches of produced changes.
            // self_produced -> SharedProgress
            for index in 0 .. progress.produceds.len() {
                self_produced[index].borrow_mut().drain_into(&mut progress.produceds[index]);
            }

            result
        };

        self.builder.build(raw_logic);
    }
```

Operator

`dataflow/operators/generic.rs` defines some common routines to build streaming & blocking operators.

An operator `x` which outputs records of type `D1`, can use methods like `unary_frontier` to create a new operator `y` . This operator has **a single input and a single output port**. x's output port would provide a `Stream`, this stream enables us to connect the output port to `y`'s input port. The operator `y` take `x`'s data, applies some logic, and then output records with data type `D2`, `x->y->out`. A `Stream` of this output is then returned. We can continue use this stream to build further operators to move data & compute.

Constructor

When we build an operator, we must provide a `constructor` that takes a vector of capabilities, one for each output port.

```
constructor: B
B: FnOnce(Vec<Capability<G::Timestamp>>) -> L
```

These are the **initial capabilities** that the operator obtain. By default, at the beginning of the dataflow computation, **each output port would acquire a capability with default timestamp** (t=0), so they can send messages with any timestamp at the beginning. We can use these capabilities (including derived / delayed ones) to create a session to generate messages for the output port. We can also use these capabilities to request notifications of the operator at any timestamp (since the initial capabilities come with t=0). For instance:

```
timely::example(|scope| {
        (0u64..10).to_stream(scope)
            .unary_frontier(Pipeline, "example", |default_cap, _info| {
                let mut cap = Some(default_cap.delayed(&12));
                let mut notificator = FrontierNotificator::new();
                let mut stash = HashMap::new();
                let mut vector = Vec::new();
                move |input, output| {
                    // what happens each time when the unary_frontier operator is executed
                    // cap is moved into the logic closure
                    if let Some(ref c) = cap.take() {
                        output.session(&c).give(12);
                    }
                    // takes input and put them into the hash map
                    while let Some((time, data)) = input.next() {
                        data.swap(&mut vector);
                        stash.entry(time.time().clone())
                            .or_insert(Vec::new())
                            .extend(vector.drain(..));
                    }
                    // also if there are notifications that can be delivered (even if it didn't got any inputs)
                    notificator.for_each(&[input.frontier()], |time, _not| {
                        if let Some(mut vec) = stash.remove(time.time()) {
                            output.session(&time).give_iterator(vec.drain(..));
                        }
                    });
                }
            });
    });
```

Here we used:

```
let mut cap = Some(default_cap.delayed(&12));
output.session(&c).give(12);
output.session(&time).give_iterator(vec.drain(..));
```

These initial capabilities passed to the `constructor` are not managed by each output's capability's `internal`, their evidence of creation is destroyed. Because when we call `get_internal_summary()`, we already allocates a capability with timestamp 0 for each output port for each worker. **These initial capabilities are already acknowledged in SharedProgress. They are just passerby that we use them to acquire delayed capabilities.** We can also try to acquire the onwernship of `capabilities` in inner logic. If they are not acquired, after `constructor` is returned, `capabilities` are dropped. At this point, no computation is performed. The system is still initializing.

```
for output_index in 0  .. self.internal.borrow().len() {
            let borrow = &self.internal.borrow()[output_index];
            capabilities.push(mint_capability(G::Timestamp::minimum(), borrow.clone()));
            // Discard evidence of creation, as we are assumed to start with one.
            // this capability is already acknowledged in SharedProgress
            // when get_internal_summary() is called.
            borrow.borrow_mut().clear();
        }
```

`unary_frontier()`

We can construct an operator `y` that can take inputs from one of the output port of `x`, it can also inspect the frontier at this input port (`y`'s) (via `FrontieredInputHandle`). It can also output data of type `D2` to `y`'s output port.

```
fn unary_frontier<D2, B, L, P>(&self, pact: P, name: &str, constructor: B) -> Stream<G, D2>
    where
        D2: Data,
        B: FnOnce(Capability<G::Timestamp>, OperatorInfo) -> L,
        L: FnMut(&mut FrontieredInputHandle<G::Timestamp, D1, P::Puller>,
                 &mut OutputHandle<G::Timestamp, D2, Tee<G::Timestamp, D2>>)+'static,
        P: ParallelizationContract<G::Timestamp, D1> {

        let mut builder = OperatorBuilder::new(name.to_owned(), self.scope());
        let operator_info = builder.operator_info();

        let mut input = builder.new_input(self, pact);
        let (mut output, stream) = builder.new_output();

        builder.build(move |mut capabilities| {
            // `capabilities` should be a single-element vector.
            // we have only a single capability, since we only have a single output
            let capability = capabilities.pop().unwrap();
            // the capability in the constructor can be used to acquire a session
            // from the OutputHandle
            let mut logic = constructor(capability, operator_info);
            move |frontiers| {
                let mut input_handle = FrontieredInputHandle::new(&mut input, &frontiers[0]);
                let mut output_handle = output.activate();
                logic(&mut input_handle, &mut output_handle);
            }
        });

        stream
    }
```

`unary_notify()`

```
fn unary_notify<D2: Data,
            L: FnMut(&mut InputHandle<G::Timestamp, D1, P::Puller>,
                     &mut OutputHandle<G::Timestamp, D2, Tee<G::Timestamp, D2>>,
                     &mut Notificator<G::Timestamp>)+'static,
             P: ParallelizationContract<G::Timestamp, D1>>
             (&self, pact: P, name: &str, init: impl IntoIterator<Item=G::Timestamp>, mut logic: L) -> Stream<G, D2> {
        self.unary_frontier(pact, name, move |capability, _info| {
            // wraps a FrontierNotificator
            let mut notificator = FrontierNotificator::new();
            // the operator needs to be notified at these points
            for time in init {
                notificator.notify_at(capability.delayed(&time));
            }
            let logging = self.scope().logging();
            move |input, output| {
                let frontier = &[input.frontier()];
                // directly construct a notificator wraps both the frontier
                // so the dataflow developer does not need to directly access the frontier
                let notificator = &mut Notificator::new(frontier, &mut notificator, &logging);
                // logic can call notificator to check to deliver any notifications that are available
                logic(&mut input.handle, output, notificator);
            }
        })
    }
```

`unary()`

```
fn unary<D2, B, L, P>(&self, pact: P, name: &str, constructor: B) -> Stream<G, D2>
    where
        D2: Data,
        B: FnOnce(Capability<G::Timestamp>, OperatorInfo) -> L,
        L: FnMut(&mut InputHandle<G::Timestamp, D1, P::Puller>,
                 &mut OutputHandle<G::Timestamp, D2, Tee<G::Timestamp, D2>>)+'static,
        P: ParallelizationContract<G::Timestamp, D1> {

        let mut builder = OperatorBuilder::new(name.to_owned(), self.scope());
        let operator_info = builder.operator_info();

        let mut input = builder.new_input(self, pact);
        let (mut output, stream) = builder.new_output();
        builder.set_notify(false);

        builder.build(move |mut capabilities| {
            // `capabilities` should be a single-element vector.
            let capability = capabilities.pop().unwrap();
            let mut logic = constructor(capability, operator_info);
            // `input` and `output` are moved into the closure
            move |_frontiers| {
                // _frontiers are of no use here
                let mut output_handle = output.activate();
                logic(&mut input, &mut output_handle);
            }
        });
        stream
    }
```

`binary_...`

binary version: build an operator that take two inputs and generates a single output.

`source()`

A single output, no input.

```
pub fn source<G: Scope, D, B, L>(scope: &G, name: &str, constructor: B) -> Stream<G, D>
where
    D: Data,
    B: FnOnce(Capability<G::Timestamp>, OperatorInfo) -> L,
    L: FnMut(&mut OutputHandle<G::Timestamp, D, Tee<G::Timestamp, D>>)+'static {

    // a single output, no input
    let mut builder = OperatorBuilder::new(name.to_owned(), scope.clone());
    let operator_info = builder.operator_info();

    let (mut output, stream) = builder.new_output();
    builder.set_notify(false);

    builder.build(move |mut capabilities| {
        // `capabilities` should be a single-element vector.
        let capability = capabilities.pop().unwrap();
        let mut logic = constructor(capability, operator_info);
        move |_frontier| {
            logic(&mut output.activate());
        }
    });
    stream
}
```

`sink()`

A single input, no output

```
    fn sink<L, P>(&self, pact: P, name: &str, mut logic: L)
    where
        L: FnMut(&mut FrontieredInputHandle<G::Timestamp, D1, P::Puller>)+'static,
        P: ParallelizationContract<G::Timestamp, D1> {

        let mut builder = OperatorBuilder::new(name.to_owned(), self.scope());
        let mut input = builder.new_input(self, pact);

        builder.build(|_capabilities| {
            move |frontiers| {
                let mut input_handle = FrontieredInputHandle::new(&mut input, &frontiers[0]);
                logic(&mut input_handle);
            }
        });
    }
```