dataflow/scopes

Created	@September 13, 2021 1:32 AM
Tags	

This create defines the hierarchical organization of timely dataflow graphs.

`ScopeParent`

Anything (any type) that can serve as a `ScopeParent` should be able to serve as a worker.

```
pub trait ScopeParent: AsWorker+Clone {
    /// The timestamp associated with data in this scope.
    type Timestamp : Timestamp;
}

// Worker of course should implement ScopeParent trait
impl<A: Allocate> ScopeParent for crate::worker::Worker<A> {
    type Timestamp = ();
}
```

The `Scope` trait declares some set of operations that a scope should implement.

`Child`

A `Child` serves as a `Scope`. It wraps a subgraph builder which can build the operators in this `Child` scope (not any more nested scopes it creates), and connect them via dataflow edges. It also stores a clone of the parent scope.

```
pub struct Child<'a, G, T>
where
    G: ScopeParent,
    T: Timestamp+Refines<G::Timestamp>
{
    /// The subgraph under assembly.
    pub subgraph: &'a RefCell<SubgraphBuilder<G::Timestamp, T>>,
    /// A copy of the child's parent scope.
    pub parent:   G,
    /// The log writer for this scope.
    pub logging:  Option<Logger>,
    /// The progress log writer for this scope.
    pub progress_logging:  Option<ProgressLogger>,
}
```

`Child` also implements `AsWorker` trait, as required by `Scope, ScopeParent` traits. It just use the parent scope's index and peers information, and use the parent scope's methods to allocate communication channels.

```
impl<'a, G, T> AsWorker for Child<'a, G, T>
where
    G: ScopeParent,
    T: Timestamp+Refines<G::Timestamp>
{
    fn config(&self) -> &Config { self.parent.config() }
    fn index(&self) -> usize { self.parent.index() }
    fn peers(&self) -> usize { self.parent.peers() }
    fn allocate<D: Data>(&mut self, identifier: usize, address: &[usize]) -> (Vec<Box<dyn Push<Message<D>>>>, Box<dyn Pull<Message<D>>>) {
        self.parent.allocate(identifier, address)
    }
    fn pipeline<D: 'static>(&mut self, identifier: usize, address: &[usize]) -> (ThreadPusher<Message<D>>, ThreadPuller<Message<D>>) {
        self.parent.pipeline(identifier, address)
    }
    fn new_identifier(&mut self) -> usize {
        self.parent.new_identifier()
    }
    fn log_register(&self) -> ::std::cell::RefMut<crate::logging_core::Registry<crate::logging::WorkerIdentifier>> {
        self.parent.log_register()
```

```
    }
}
```

`Child` can also create nested `Child` scopes via `scoped()` method (required by `Scope` trait). It creates a new instance of `Child` with a clone of the calling `Child` (parent), and a reference to a newly created `SubgraphBuilder`. A closure `func` can then takes the mutable reference to the created `Child`. `func` is in charge of calling the methods of `Child` to add operators and connect edges to this nested child scope.

When then call the `SubgraphBuilder`'s `build()` method to create the subgraph. Since subgraph is just an operator to the parent scope, we add this "operator" to the parent scope (parent / calling `Child`).

```
fn scoped<T2, R, F>(&mut self, name: &str, func: F) -> R
    where
        T2: Timestamp+Refines<T>,
        F: FnOnce(&mut Child<Self, T2>) -> R,
    {
        let index = self.subgraph.borrow_mut().allocate_child_id();
        let path = self.subgraph.borrow().path.clone();

        // note that when we call SubgraphBuilder::new_from()
        // the index of this sub-scope (an operator from the parent scope's view) will be pushed to the end of index
        // via path.push(index);
        let subscope = RefCell::new(SubgraphBuilder::new_from(index, path, self.logging().clone(), self.progress_logging.clone(), name));
        let result = {
            let mut builder = Child {
                subgraph: &subscope,
                parent: self.clone(),
                logging: self.logging.clone(),
                progress_logging: self.progress_logging.clone(),
            };
            // func is provided with a mutable reference to the empty Child we just create
            // it then calls the child's methods to add operators and connect them to form a dataflow subgraph
            // we build the subgraph according to func
            func(&mut builder)
        };
        // build the subgraph, subgraph also implements Operator trait

        let subscope = subscope.into_inner().build(self);

        self.add_operator_with_index(Box::new(subscope), index);

        result
    }
```