The State of Options Merging 25-4-16

Several suspicious issues have emerged over time with the way we operate the options merging and expansion process in the framework, which we now collect here, as things are coming to a head prior to the significant framework rewrite hopefully expected later in the year. It seems that various of the goals and design ideals of the options merging system are conflicting, and we will need to make some hard decisions and significant changes in order to establish a coherent model.

The JIRA FLUID-5800 contains a lot of discussion about the options merging algorithm itself, and records some vacillations over whether we felt the need to move over to a more well-attested grade linearization algorithm such as C3 (described in the classic Dylan paper). As it seems to be turning out, whilst our linearization algorithm, which could be described as a "depth-first painter's algorithm", can lead to some peculiar results, it might well be defensible in the light of our overall approach to graphs of authors described in our Onward paper as well as not being directly implicated in our most serious problems. The JIRA most clearly giving the flavour of our most serious problems is FLUID-5668 "corruption when material written for members requires merging". An issue very similar to this came to light in work for the Onward paper itself.

Problems exposed by mergePolicies

The problem is most clearly exposed when trying to work with "mergePolicies" which are now by far the most immature and undeveloped part of the options merging pipeline, essentially untouched since the days of the ancient framework of 2009. They don't even feature support for global name resolution, implying that any grade definition using them can't even be represented in a JSON store. They haven't been updated primarily since it hasn't been very clear what to do with them, since almost any use of them seems to lead to difficulties. The framework makes use of them in many cases, for example where `listeners`, `modelListeners`, `modelRelay`, `model`, etc. require something other than the simple-minded "painter's algorithm" combination of options by merging. This in itself is suspicious since we shouldn't have the framework require to make use of facilities which are not easily usable by the general user. For example, we consider that the ContextAwareness facilities are more or less a success since they can be implemented simply using standard primitives of the framework - in "usermode" as we say in the Onward paper. If the user doesn't like them, equivalent or more powerful facilities can be easily cooked up using new syntax and the same facilities. The same isn't true of the more fundamental framework features listed above.

The most direct sign of the rot was the strange `fluid.mergingArray` "exotic type" implemented as a stopgap solution for FLUID-5668. This type also ended up pressed into service in the Onward paper when we needed a member which accumulated a list of matrices to be multiplied later, rather than each overwriting each other. This is necessary because of the failure of the mergePolicy "accumulate" algorithm to be what one could call "properly monadic" - it is unable to take account of where it is in its workflow - that is, whether it has been presented with an argument which has been processed once, and hence, in some view, is "inside a container", or whether it has been presented with a fresh piece of material from the "outside world". Another huge workflow problem with mergePolicies stems from the rather odd irregularity of our workflow with respect to merging and expansion. This, like mergePolicies, has been somewhat "emergent" and currently takes the following form:

1. An initial phase of "pure merging" where grade definitions are composited (after the linearization process we spoke of above) into a "merged grade definition" - nothing very unusual occurs during this phase and it was always envisaged that its output might be cached, as itself, pure JSON, although this is now seeming a little unrealistic (especially since we are still not at the stage where all grade definitions themselves can be expressed as pure JSON - most importantly, because of mergePolicies themselves)
2. A mixed phase of "combined merging and expansion" - this forms the core of the "globally ginger process" that was brought in by the FLUID-4330 and essentially marked the birth of the "modern framework" (filed in 2011 and delivered in 2013)
    a. Material is assembled into "mergeBlocks" (the first of which corresponding to defaults, others to other sources of options such as subcomponents, the user's direct arguments, and options distributions) each of which has a "source" consisting of unexpanded material and a "target" consisting of expanded material. Each of these blocks is elaborated in a data-driven way as finally-resolved values for options are demanded across the instantiating component tree
    b. The total list of "target" blocks is merged together into the final component's options, again, piecemeal in a data-driven way

The fundamental problem with the design of mergePolicies is that they are completely blind as to whether they are operating as part of the harmless phase 1 (all they were ever really designed for) or the complex data-driven phase 2b. This means that they may encounter material which has either been expanded or not expanded. This implies that they must make NO ATTEMPT WHATSOEVER to inspect their arguments given their unknown workflow. We make an exception for this structure-agnosticism in the mergePolicies operating core framework primitives such as listeners, modelListeners etc. in which we insist on a fully knowable top-level key-value structure that the framework's expansion process will not attempt to cross (this insistence is responsible for essentially intractible "bugs" such as FLUID-5208). This is also what makes it essentially impossible for the "average user" to ever effectively craft a new mergePolicy - since they effectively have to operate the expansion part of the workflow (2a.) themselves - this can be seen in all of the core primitives listed, e.g. `fluid.mergeModelListeners`, `fluid.mergeListeners` which both punch through to a custom and rather complex `fluid.resolveListenerRecord`, `fluid.establishModelRelay` and the like. This is somewhat forgiveable when the core workflow does require so much intricate resolution and even reinterpretation of the meaning of configuration, but unacceptable when the user is trying to do something as simple as accumulating a list of matrices as in onward.imageRenderer.

Problems exposed by "source" type options distributions

Another framework primitives that exposes problems cause by the irregular "sandwich" workflow (simple merging, followed by ginger combined expansion then merging again described in 1, 2a and 2b in the previous section) is options distributions of the "source" type. These are naturally "cruising for a bruising" since their purpose is to broadcast (presumably already fully expanded and merged) options material in its final form from one location in the tree to another. In practice they do not work this way, but operate directly at the "mergeBlocks" level described in the previous section. On encountering a "source" distribution the framework, rather than attempting to distribute the finally merged result, instead distributes as best it can the *raw materials leading to the result* - that is, it filters each of the mergeBlocks at their current position for "source" entries matching the distribution and constructs one, filtered, incarnation of that block at the distribution target site with a filtered "source" record. In theory, this approach, being slightly more principled, has the potential for better results than distributing fully processed options (for a start, being capable of dealing with "site of expansion" issues such as FLUID-5258) , but in practice the implementation is a little half-baked. Our solution to FLUID-5887 whilst resolving the headline site of the issue in Builder.js, ended up exposing an obnoxious problem in InlineEdits.js, where there is an ancient style of distribution (of the kind we used to have in the Uploader) that attempts a "total source distribution":

```
    fluid.defaults("fluid.inlineEditsComponent", {
        gradeNames: ["fluid.viewComponent"],
        distributeOptions: {
            source: "{that}.options",
            // This seems to imply that we've got no option but to start supporting "provenance" in options and
defaults - highly expensive.
            exclusions: ["members.inlineEdits", "members.modelRelay", "members.applier", "members.model",
"selectors.editables", "events"],
            removeSource: true,
            target: "{that > fluid.inlineEdit}.options"
        },
```

This, by distributing ALL sources of options, including those derived from `fluid.modelComponent` itself, caused a double-initialisation via the `members: modelRelay` definition in core `fluid.modelComponent` -

```
        members: {
            model: "@expand:fluid.initRelayModel({that}, {that}.modelRelay)",
            applier: "@expand:fluid.makeHolderChangeApplier({that}, {that}.options.changeApplierOptions)",
            modelRelay: "@expand:fluid.establishModelRelay({that}, {that}.options.model, {that}.options.
modelListeners, {that}.options.modelRelay, {that}.applier)"
        },
```

Whilst we were able to fix up the core grade linearization algorithm to ensure that a given grade is contributed at most once to a particular hierarchy, the options distribution process effaces the information as to where particular material has come from, since by this point in the workflow it has ended up in a general mergeBlock bucket of type "defaults" - note that having multiple blocks of type "defaults" was one of the architectural surprises which "mid-level mature options distributions" produced. This particular problem looks like it might be solved by "provenance" - see below, in "ideas" - the general mergePolicy problem, however, looks like it can't be.

Some ideas

One idea that has been kicking around for a while is the use of "provenance" in options. This will actually be highly useful bordering on essential once we have visual authoring tools - Clemens asked in our of our chats "how well the user has to understand the merging model" and I replied that different sections of options which have come from different source "could be painted in different colours", etc. Certainly, the most useful framework diagnostic implies that each option in a finally merged component could be tracked back to the original site of its authorship (with extra marks, for showing how it got there). This idea has been kicked around for some time, for example with a short mention in FLUID-4930, our post-rewrite "panic JIRA" about the poorly-characterised failure entitled "retrunking" that will require one of the main clarifications of "expansion policy intention" underlying the upcoming FLUID-4982. If we had "per-options provenance" on every piece of options, this would allow us to resolve issues like the "source distribution" one above, since we could insist that a distribution "only distributes options with provenance that does not arise from one of the grades already present at the target".

However, this implies to start with, not only hugely increased storage and computation costs for the options merging process, but also a (previously considered) fatal compromise on the status of "options" and "defaults" as being pure JSON. However, if we require "provenance" to even interact with mergePolicies, this seems painfully unavoidable. The provenance must somehow **accompany** the data, which implies it is packaged somehow. This also presumably allows some resolution of our "monadicity" issue where it is unclear what stage of processing the options have reached. However, the "immutable revolution" already represents some kind of compromise since even the relatively harmless-seeming act of "freezing" all defaults (and eventually, all options) with Object.freeze represents a (smallish) violation of the expectation that "options material represents pure JSON".

Now, at least at runtime, we can store the provenance map inside the component's shadow, making it accessible with an API. But during options merging, it seems that we have to make some attempt to supply this via direct arguments. This seems to imply some form of "mergPolicy hoisting". The core algorithm inside "onward.arrayConcatPolicy" has already become strongly obscured. This implies that we want to "hoist" the core algorithm (that of concatenating arrays or some other thing) into a form where it only gets to manipulate options abstractly, which are then hidden behind a kind of armoury of "exotic types" like fluid.mergingArray of the kind which we traditionally considered very distasteful. This armoury would let us discover i) the location of the material, ii) its state of expansion, iii) its source, e.g. in grade terms - but to the core algorithm would then represent a kind of opaque, immutable token. Pretty unpleasant, but this seems hard to avoid.

The algorithm itself

Whilst there may be ways around some of these problems, the discussion so far doesn't shed much light on what we could do about the workflow ("sandwich") itself. Cindy has noted in the channel that it can seem a little odd, to someone who has to deal with it quite closely - in particular, the fact that defaults merging represents "merging before expansion" whilst the merging applied to options distribution represents "merging after expansion". A big part of the reason we did defaults merging the way we did was in order to allow for the possibility of caching large parts of the workflow, and preferably in a form which was JSON-amenable. This was primarily done in the bad old CSpace days of 2009, where we were faced with a very large page comprising perhaps a thousand Infusion components which needed to render relatively quickly on the still rather rickety JS VMs of the day. This also drove the "accumulative linearisation" approach which we abandoned the form of in the FLUID-5887 fix (although we noted that we are still actually following exactly the same algorithm, if a little more inefficiently, as a result of its "painter's-like" nature). In the FLUID-4982 world, we plan for "huge chains of linked immutable prototypes" as our optimisation for grade merging and loading - which hardly seem amenable to effective JSON representation. If we can make this work, it seem that completely abolishing phase 1 of the merging and expansion process will be feasible in a performant system. This will also return it more towards accepted computer science norms, where the necessity for "expansion before merging" (that is, evaluation of function arguments before returning results) is taken as axiomatic. This might also simplify the requirements on the "mergePolicy hoisting system" mentioned in the previous section.

We need massive improvements in performance during component instantiation, which has not been heavily optimised since the CSpace days. Our optimisation work on invokers and listeners for FLUID-5249 and FLUID-5796 showed the effectiveness of the "monomorphisation" approach recommended by Egorov and others for improving the performance of code governed by the "polymorphic inline caches" invented by Ungar et al. for accelerating highly dynamic languages. We need to do something similar for the much more complex expansion and merging process. It seems we should be able to reuse some of the same ideas in a more elaborate form - for example, the "shape cache" for the V8 runtime is a simple flat model of property occurrence on a single object. The merging process features "occlusion" of earlier expanded objects by later ones, but if we can be sure that successive resolved objects are the same "shape" in the deep sense - that is, as entire JSON objects, we can be sure that the resulting merged object fetches material from its source in the same pattern. This leads to the idea of a form of **symbolic execution** of the merge pathway. Having acquired the **shape** of fetched items, one imagines that one would replace them by "self-skeletons" - objects which have their own path references replacing all leaves, and executing the merge operation with those instead. Or instead, to perform the merge with "tokens" wrapping both trunk and leaf objects which will allow provenance information to be tracked. This "symbolic merge output" would be "keyed" in storage by not only the list of grades involved, but also by the *shapes* of every object fetched via expanders or references. Options distributions and user records would be assigned "nonce grade names" in order to participate in this tracking. After that, merging simply has the form of i) fetching this record, ii) scanning through the monomorphised list of unoccluded expansion material, fetching the root references, iii) writing the unoccluded parts of these objects into a single "stencil object" which contains all variable material for the finally merged options in a single layer. All other material would be sourced from the static "liana chains of immutable prototypes" which we use to represent the defaults chain.

This leaves a couple of questions - firstly, what becomes of the "partial evaluation" model? The result is that "a lot more work" gets done during the minimal phase of the algorithm in that at the very moment we determine the final grade lists around the skeleton, we also get all information resulting from the immutable prototype chains for defaults for free. This is fine - we then just need to schedule the work of fetching the unoccluded expansion material in the familiar ginger way, once we have been signalled to start the process.

Secondly, what of mergePolicies? If we require fully "provenanced" output, does this imply that we need to prohibit all mergePolicies which do anything other than shift and select object references (the "path-based model transform dialect without transforms" that we started with in 2011)? Perhaps not. We could just distinguish between two classes of mergePolicies - i) the default, which selects only unoccluded material, and ii) anything else, which might expect to read ALL material. In the latter case, we automatically assemble each participating non-empty element into an array, WHILST tracking its provenance, and then only operate the mergePolicy at the very final step. This also has the virtue of preserving our original "reduce-based" mergePolicies and placing them on a sane footing - in fact they will only ever reduce arrays, and only ever see fully expanded material, and only ever operate directly back-to-back rather than in a scattered fashion throughout the expansion process.

This all seems generally workable, subject to getting some detailed understanding of how the "stencilling" process that we use to symbolically compute occluding contours during the merge process will actually work.