Avoiding leaks in deoptimization literals

Attention - this doc is public and shared with the world!

Contact: seth.brenith@microsoft.com
Status: Inception | **Draft** | Accepted | Done
Bug: v8:4578
Implementation: [heap] Fix leaks due to deoptimization literals (I02e86683)

LGTMs needed

Name	Write (not) LGTM in this row
mlippautz@	lgtm
mvstanton@	lgtm
<add yourself>	

What's the problem?

Let's start with an example from the bug. Consider the lifetime of the `CustomObject` instantiated in the third line of this program. It is a necessary part of the execution context for the function variable `fn`, but once `fn` is cleared on the last line, the `CustomObject` should become unrooted and therefore eligible for garbage collection. However, we see that the `CustomObject` instance is kept alive via the deoptimization literals for function `g`.

```
class CustomObject {}
function makeFn() {
  var co = new CustomObject();
  co.num = 0;
  return () => { return ++co.num; };
}
var fn = makeFn();
function g(f) { f(); }
%PrepareFunctionForOptimization(g);
%PrepareFunctionForOptimization(fn);
g(fn);
%OptimizeFunctionOnNextCall(g);
g(fn);
```

```
fn = null;
```

How things work now

Deoptimization literals

Any TurboFan-compiled JS function can have a variety of different execution paths that lead to deoptimization. For each of these paths, there is information in the DeoptimizationData describing how to restore an interpreter stack frame which can continue execution where the optimized code left off. The interpreter stack frame might require values that were stored in registers or on the stack at the time of deoptimization, and it also might require constant values. Those constant values are the deoptimization literals. They are stored in a plain FixedArray owned by the DeoptimizationData, which is in turn owned by the Code.

Embedded object pointers

TurboFan generates code that includes object pointers directly in the instruction stream. This is useful, for example, when checking whether a JSFunction instance matches one that was inlined. However, the link from the Code to that embedded JSFunction instance is considered weak in most cases, to ensure that the optimized code doesn't keep the embedded object alive when it's no longer reachable in any user-visible way. If any weak embedded object doesn't survive garbage collection, then the Code pointing to it is marked for deoptimization and all of its embedded object pointers are cleared. The generated code checks whether it has been marked for deoptimization both at function entry and after returning from any other function that could have caused garbage collection. This mechanism is referred to as "lazy deoptimization".

"In most cases"?

If a Code object is currently executing as the top frame on the stack, and it is not at a point where it's just about to check for deoptimization, then clearing all of its embedded object pointers would be very bad. So in that case, all embedded object pointers are treated as strong.

What other embedded object types are weak?

These ones:

```
bool Code::IsWeakObjectInOptimizedCode(HeapObject object) {
  Map map = object.map(kAcquireLoad);
  InstanceType instance_type = map.instance_type();
  if (InstanceTypeChecker::IsMap(instance_type)) {
    return Map::cast(object).CanTransition();
  }
```

```
  return InstanceTypeChecker::IsPropertyCell(instance_type) ||
         InstanceTypeChecker::IsJSReceiver(instance_type) ||
         InstanceTypeChecker::IsContext(instance_type);
}
```

What's the problem? (part 2)

V8 has very careful and precise handling of embedded object pointers to avoid leaks, but a lot of the same objects end up in the deoptimization literals, where they are held strongly. In the example we started with, the optimized function `g` holds all of the following via deoptimization literals:

- SharedFunctionInfo and BytecodeArray for the outer function `g`
- SharedFunctionInfo and BytecodeArray for the inlined function `fn`
- The JSFunction and Context instances for the inlined function `fn`
- The CustomObject instance from the inlined function's Context
- The JSGlobalProxy
- The OptimizedOut oddball value

If we assume that the Code is not currently executing, then I believe the only literals that *must* be held strongly are the two BytecodeArrays (to prevent flushing) and the SharedFunctionInfo for the outer function (which I imagine is probably used during the function-entry deoptimization path). The inlined JSFunction, its Context, and the JSObject value contained by that Context should all be weak. Based on this example, it seems that Code::IsWeakObjectInOptimizedCode would be a perfectly fine selection mechanism for how to treat deoptimization literals.

Proposal

(If you'd rather just see the code, here it is.)

The following content is kept so the comments and discussion are visible, but based on discussion, we can use a simpler implementation. Thanks everyone!

~~Without changing how deoptimization literals are arranged in memory, we can define custom visiting semantics that avoid the leaks presented above. We make use of the fact that the FixedArrays containing deoptimization literals are never shared between multiple Code instances (unless they are the canonical empty array). Otherwise there would be a risk of failing to deoptimize all of the relevant Codes.~~

~~We start by defining a new type, which looks like a FixedArray but with custom-weak visiting behavior:~~

```
@export
class DeoptimizationLiteralArray extends HeapObject {
  const length: Smi;
  weak objects[length]: Object;
}
```

Marking behavior

If the marking visitor encounters a DeoptimizationLiteralArray (henceforth DLA) directly that is still white, then it treats all of the objects as strong. This is because a detached DLA has no way to tell what Code it should mark for deoptimization. Usually DLAs are only reachable via their corresponding Code, but sometimes they might be held via Handles.

When marking a Code object, the marking visitor checks whether a DLA is attached. If so, and if that DLA is still white, then it marks the DLA black and iterates the literals. Any literals that should be treated as weak (according to Code::IsWeakObject) and aren't yet marked are put in a list for processing later, much like the behavior for any other weak reference. Each entry in that list contains pointers to the Code, the DLA, and the slot within the DLA. It seems a little bit redundant to have both the DLA and a slot within it, but we must be able to find the page header for the DLA and I don't see any clear guarantee that it couldn't be in large-object space.

When marking the stack roots, the MarkCompactCollector iterates the deoptimization literals of each running Code and marks them all as roots too. It would be possible to iterate only the deoptimization literals required for the current function position in each Code, but doing so would add a lot of complexity for questionable benefit.

ClearNonLiveReferences behavior

When processing weak references, the MarkCompactCollector must iterate the new list of weak references from DLAs. If any item in the list points to an object which is still unmarked, then the corresponding Code is marked for deoptimization and its embedded object pointers are cleared, just like the behavior for weak embedded object pointers. Also, the slot in the DLA is overwritten with undefined.