几个torque语句分析

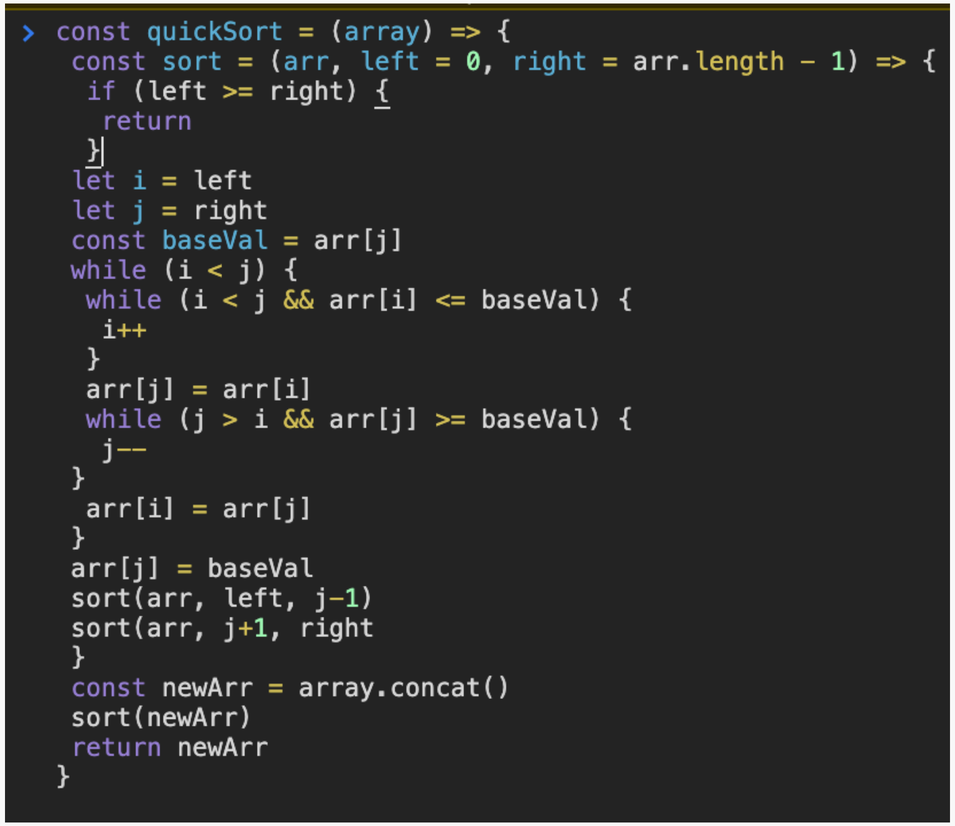
软件所智能软件研发中心 实习生 杨文章

2020/06/03

目 录

0. ADT

抽象数据类型（abstract data type,ADT）只是一个数学模型以及定义在模型上的一组操作。通常是对数据的抽象，定义了数据的取值范围以及对数据操作的集合。



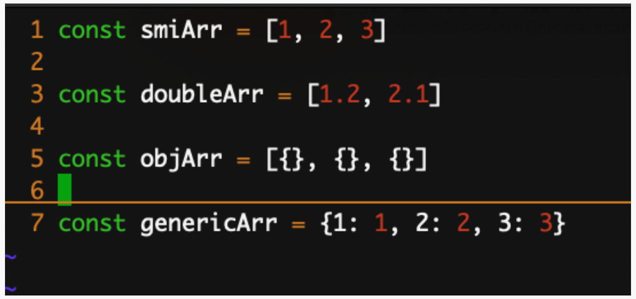
```
> const quickSort = (array) => {
   const sort = (arr, left = 0, right = arr.length - 1) => {
    if (left >= right) {
     return
    }
   let i = left
   let j = right
   const baseVal = arr[j]
   while (i < j) {
    while (i < j && arr[i] <= baseVal) {
     i++
    }
    arr[j] = arr[i]
    while (j > i && arr[j] >= baseVal) {
     j--
   }
    arr[i] = arr[j]
   }
   arr[j] = baseVal
   sort(arr, left, j-1)
   sort(arr, j+1, right
   }
   const newArr = array.concat()
   sort(newArr)
   return newArr
  }
```

数据类型是数组，操作集合是 get，set

1. 背景概述

中国科学院软件研究所 Institute of Software Chinese Academy of Sciences | 智能软件研究中心 Intelligent Software Research Center

JS的数组很灵活，比如



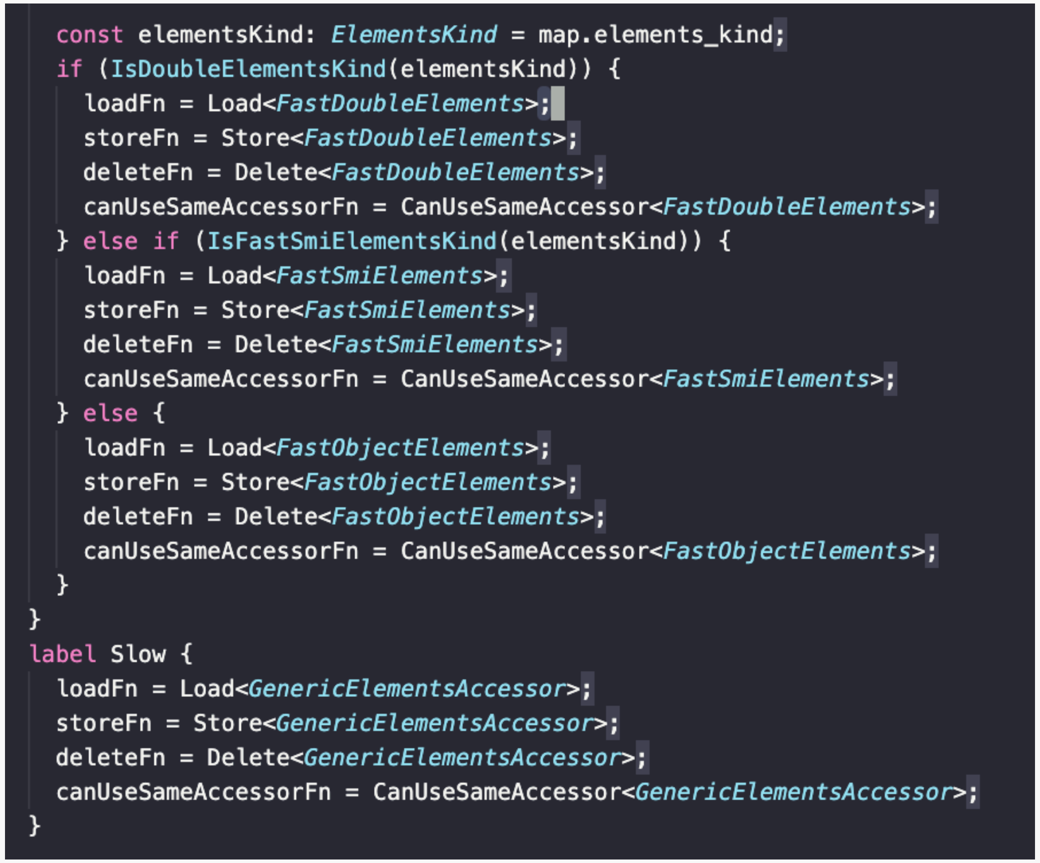
```
const smiArr = [1, 2, 3]

const doubleArr = [1.2, 2.1]

const objArr = [{}, {}, {}]

const genericArr = {1: 1, 2: 2, 3: 3}
```

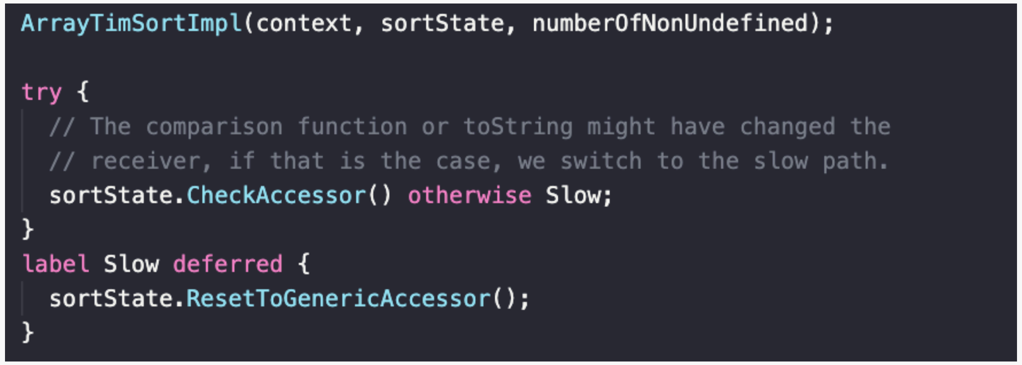
在v8内部为了加快处理速度，对其做了区分
下面是array-sort.tq中的代码片段



```
  const elementsKind: ElementsKind = map.elements_kind;
  if (IsDoubleElementsKind(elementsKind)) {
    loadFn = Load<FastDoubleElements>;
    storeFn = Store<FastDoubleElements>;
    deleteFn = Delete<FastDoubleElements>;
    canUseSameAccessorFn = CanUseSameAccessor<FastDoubleElements>;
  } else if (IsFastSmiElementsKind(elementsKind)) {
    loadFn = Load<FastSmiElements>;
    storeFn = Store<FastSmiElements>;
    deleteFn = Delete<FastSmiElements>;
    canUseSameAccessorFn = CanUseSameAccessor<FastSmiElements>;
  } else {
    loadFn = Load<FastObjectElements>;
    storeFn = Store<FastObjectElements>;
    deleteFn = Delete<FastObjectElements>;
    canUseSameAccessorFn = CanUseSameAccessor<FastObjectElements>;
  }
}
label Slow {
  loadFn = Load<GenericElementsAccessor>;
  storeFn = Store<GenericElementsAccessor>;
  deleteFn = Delete<GenericElementsAccessor>;
  canUseSameAccessorFn = CanUseSameAccessor<GenericElementsAccessor>;
}
```

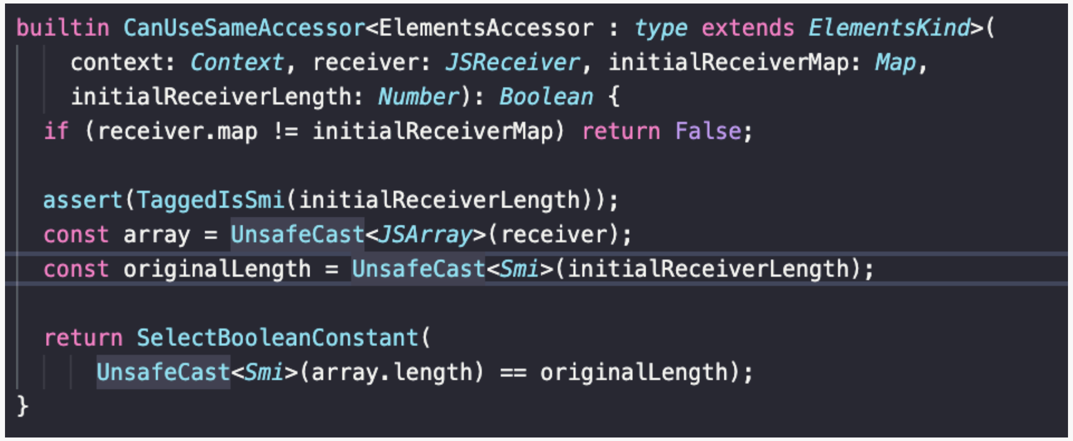
2. 语句详解

在调用ArrayTimSortImpl之后，会去做一个accessor访问检查，查看数组的元素类型是否发生改变



```
ArrayTimSortImpl(context, sortState, numberOfNonUndefined);

try {
  // The comparison function or toString might have changed the
  // receiver, if that is the case, we switch to the slow path.
  sortState.CheckAccessor() otherwise Slow;
}
label Slow deferred {
  sortState.ResetToGenericAccessor();
}
```

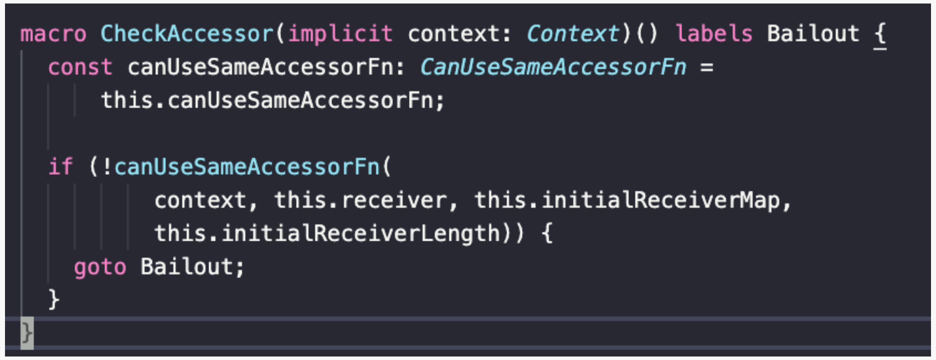


```
builtin CanUseSameAccessor<ElementsAccessor : type extends ElementsKind>(
    context: Context, receiver: JSReceiver, initialReceiverMap: Map,
    initialReceiverLength: Number): Boolean {
  if (receiver.map != initialReceiverMap) return False;

  assert(TaggedIsSmi(initialReceiverLength));
  const array = UnsafeCast<JSArray>(receiver);
  const originalLength = UnsafeCast<Smi>(initialReceiverLength);

  return SelectBooleanConstant(
      UnsafeCast<Smi>(array.length) == originalLength);
}
```

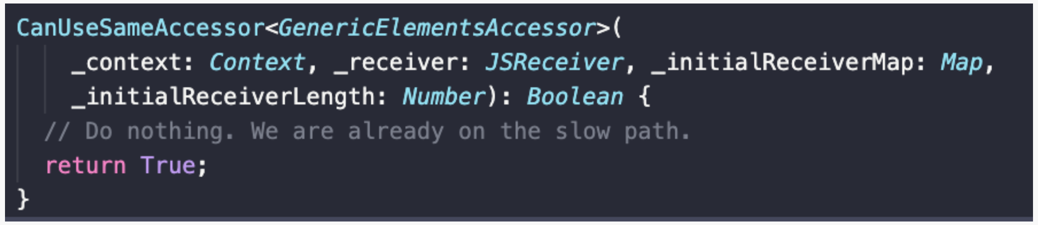
最终会调用之前设置好的canUseSameAccessorFn



```
macro CheckAccessor(implicit context: Context)() labels Bailout {
  const canUseSameAccessorFn: CanUseSameAccessorFn =
      this.canUseSameAccessorFn;

  if (!canUseSameAccessorFn(
          context, this.receiver, this.initialReceiverMap,
          this.initialReceiverLength)) {
    goto Bailout;
  }
}
```

在v8里面，最顶层的类型就是generic，类型将不会发生改变，所以直接返回了true

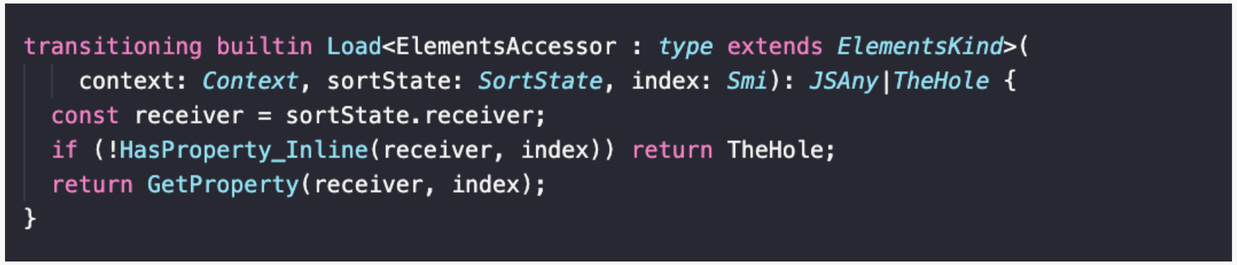


```
CanUseSameAccessor<GenericElementsAccessor>(
    _context: Context, _receiver: JSReceiver, _initialReceiverMap: Map,
    _initialReceiverLength: Number): Boolean {
  // Do nothing. We are already on the slow path.
  return True;
}
```

2. 语句详解

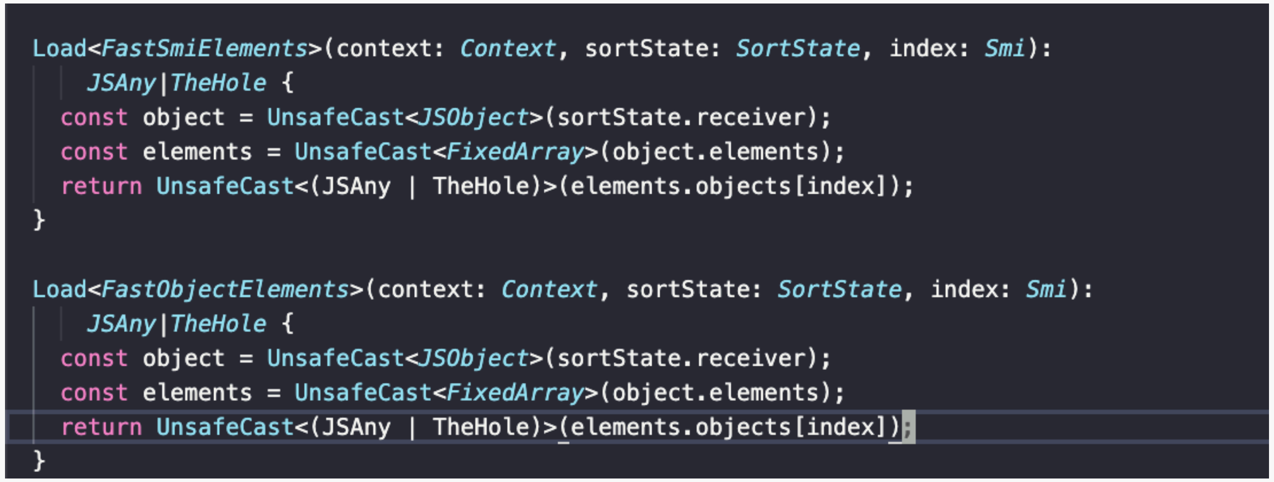
ISCAS 中国科学院软件研究所 Institute of Software Chinese Academy of Sciences | 智能软件研究中心 Intelligent Software Research Center

Load是对数组元素get操作的封装，对应不同类型的方法



```
transitioning builtin Load<ElementsAccessor : type extends ElementsKind>(
    context: Context, sortState: SortState, index: Smi): JSAny|TheHole {
  const receiver = sortState.receiver;
  if (!HasProperty_Inline(receiver, index)) return TheHole;
  return GetProperty(receiver, index);
}
```

可以看到Load FastSmiElements和FastObjectelements的代码是一样的

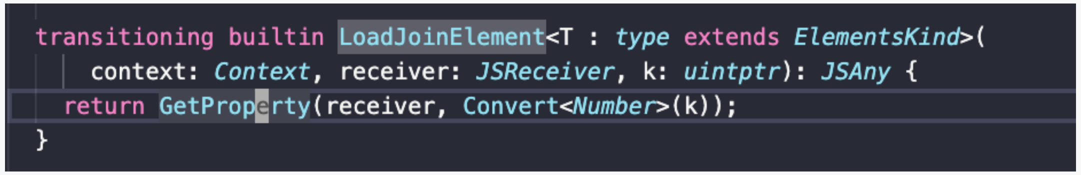


```
Load<FastSmiElements>(context: Context, sortState: SortState, index: Smi):
    JSAny|TheHole {
  const object = UnsafeCast<JSObject>(sortState.receiver);
  const elements = UnsafeCast<FixedArray>(object.elements);
  return UnsafeCast<(JSAny | TheHole)>(elements.objects[index]);
}

Load<FastObjectElements>(context: Context, sortState: SortState, index: Smi):
    JSAny|TheHole {
  const object = UnsafeCast<JSObject>(sortState.receiver);
  const elements = UnsafeCast<FixedArray>(object.elements);
  return UnsafeCast<(JSAny | TheHole)>(elements.objects[index]);
}
```

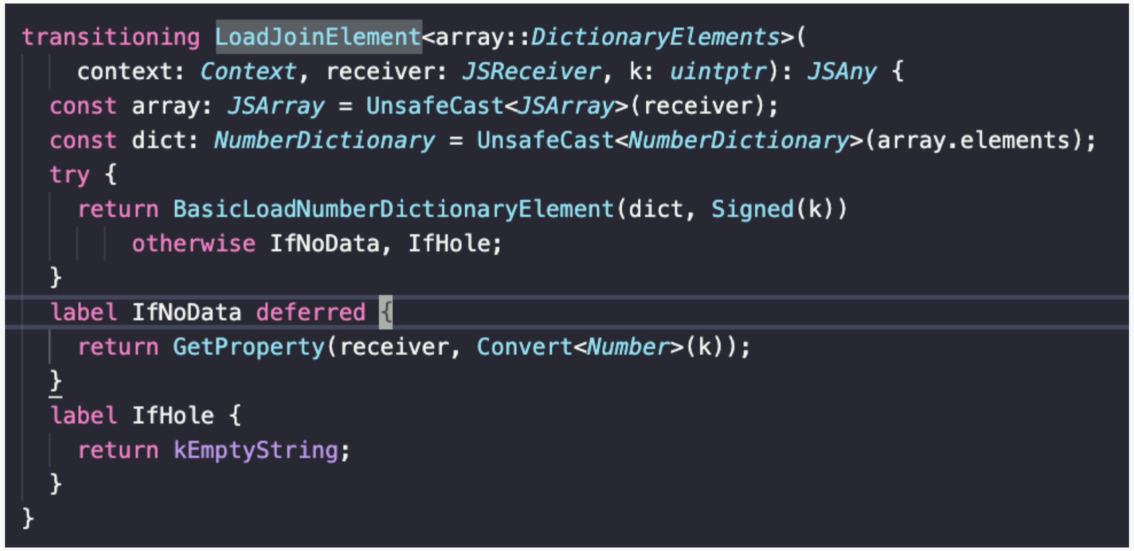
2. 语句详解

LoadJoinElement是在array-join.tq中定义的，也是封装了对数组元素的get操作，对应于不同的类型。



```
transitioning builtin LoadJoinElement<T : type extends ElementsKind>(
    context: Context, receiver: JSReceiver, k: uintptr): JSAny {
  return GetProperty(receiver, Convert<Number>(k));
}
```

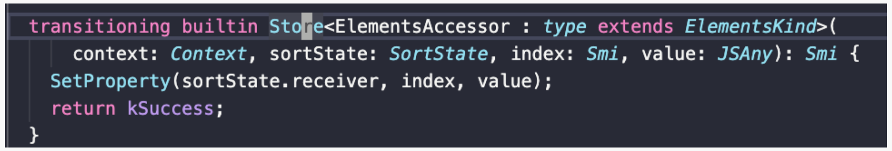
也会有特化之后的代码，array-join中需要对DictionaryElements类型的数组封装出特定的load方法，这也展示了为什么 Load 这么常规的方法不是统一封装，而是不同的tq文件根据需要定义自己的load。



```
transitioning LoadJoinElement<array::DictionaryElements>(
    context: Context, receiver: JSReceiver, k: uintptr): JSAny {
  const array: JSArray = UnsafeCast<JSArray>(receiver);
  const dict: NumberDictionary = UnsafeCast<NumberDictionary>(array.elements);
  try {
    return BasicLoadNumberDictionaryElement(dict, Signed(k))
        otherwise IfNoData, IfHole;
  }
  label IfNoData deferred {
    return GetProperty(receiver, Convert<Number>(k));
  }
  label IfHole {
    return kEmptyString;
  }
}
```

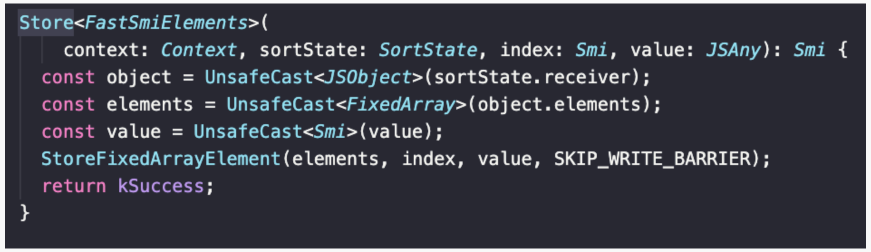
2. 语句详解

Store也是一样的，分多个不同的类型



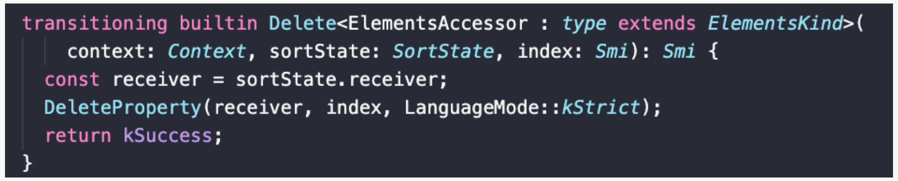
```
transitioning builtin Store<ElementsAccessor : type extends ElementsKind>(
    context: Context, sortState: SortState, index: Smi, value: JSAny): Smi {
  SetProperty(sortState.receiver, index, value);
  return kSuccess;
}
```

特化的Store类型



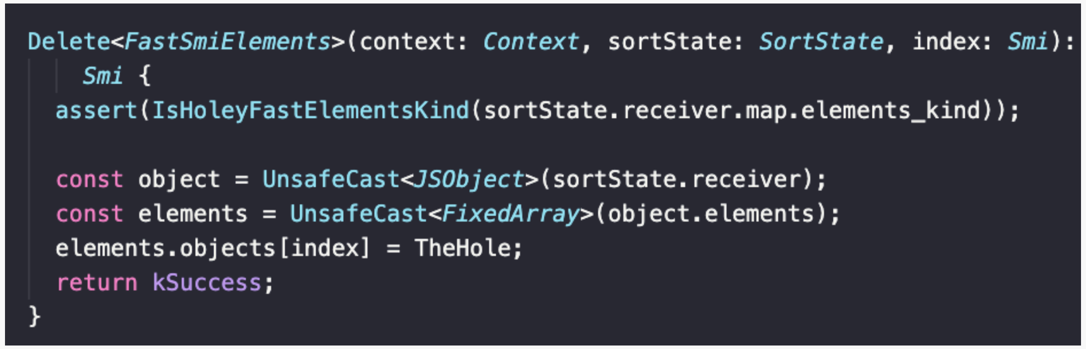
```
Store<FastSmiElements>(
    context: Context, sortState: SortState, index: Smi, value: JSAny): Smi {
  const object = UnsafeCast<JSObject>(sortState.receiver);
  const elements = UnsafeCast<FixedArray>(object.elements);
  const value = UnsafeCast<Smi>(value);
  StoreFixedArrayElement(elements, index, value, SKIP_WRITE_BARRIER);
  return kSuccess;
}
```

同理，delete也是一样



```
transitioning builtin Delete<ElementsAccessor : type extends ElementsKind>(
    context: Context, sortState: SortState, index: Smi): Smi {
  const receiver = sortState.receiver;
  DeleteProperty(receiver, index, LanguageMode::kStrict);
  return kSuccess;
}
```

特化的delete类型，操作是给对应的位置赋值为TheHole值



```
Delete<FastSmiElements>(context: Context, sortState: SortState, index: Smi):
    Smi {
  assert(IsHoleyFastElementsKind(sortState.receiver.map.elements_kind));

  const object = UnsafeCast<JSObject>(sortState.receiver);
  const elements = UnsafeCast<FixedArray>(object.elements);
  elements.objects[index] = TheHole;
  return kSuccess;
}
```

谢 谢

欢迎交流合作

2020/6/10

参考资料