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

V8中的inline cache实现

陆亚涵 PLCT实验室

2021/10/21

```
function f(o) {
  return o.x
}

f({ x: 1 })
f({ x: 2 })
f({ x: 5 })
f({ x: 4, y: 1 })
f({ x: 5, z: 1 })
f({ x: 6, a: 1 })
f({ x: 7, b: 1 })
```

```
function f(o) {
  return o.x
}
%PrepareFunctionForOptimization(f);
%OptimizeFunctionOnNextCall(f);
f({ x: 1 })
f({ x: 2 })
f({ x: 5 })
f({ x: 4, y: 1 })
f({ x: 5, z: 1 })
f({ x: 6, a: 1 })
f({ x: 7, b: 1 })
```

```
[Feedback slot 0/2 in 0x0028962a26d9 <SharedFunctionInfo f> updated to MONOMORPHIC - Monomorphic]
[Feedback slot 0/2 in 0x0028962a26d9 <SharedFunctionInfo f> updated to POLYMORPHIC - Polymorphic]
[Feedback slot 0/2 in 0x0028962a26d9 <SharedFunctionInfo f> updated to POLYMORPHIC - Polymorphic]
[Feedback slot 0/2 in 0x0028962a26d9 <SharedFunctionInfo f> updated to POLYMORPHIC - Polymorphic]
[Feedback slot 0/2 in 0x0028962a26d9 <SharedFunctionInfo f> updated to MEGAMORPHIC - Megamorphic]
```

```
function f(o) {
  return o.x
}

f({ x: 1 })
f({ x: 2 })
f({ x: 5 })
f({ x: 4, y: 1 })
f({ x: 5, z: 1 })
f({ x: 6, a: 1 })
f({ x: 7, b: 1 })
```

```
[generated bytecode for function: f (0x00ef666e2679 <SharedFunctionInfo f>)]
Bytecode length: 5
Parameter count 2
Register count 0
Frame size 0
OSR nesting level: 0
Bytecode Age: 0
        0xef666e2afe @    0 : 2d 03 00 00      LdaNamedProperty a0, [0], [0]
        0xef666e2b02 @    4 : a9               Return
Constant pool (size = 1)
0xef666e2ab1: [FixedArray] in OldSpace
 - map: 0x00f622f012c1 <Map>
 - length: 1
           0: 0x00ef666e2571 <String[1]: #x>
Handler Table (size = 0)
```

```
function f(o) {
  return o.x
}

f({ x: 1 })
f({ x: 2 })
f({ x: 3 })// o.x cache is still monomorphic here
f({ x: 4, y: 1 }) // polymorphic, degree 2
f({ x: 5, z: 1 }) // polymorphic, degree 3
f({ x: 6, a: 1 }) // polymorphic, degree 4
f({ x: 7, b: 1 }) // megamorphic
```

- Monomorphic cache says “I’ve **only** seen type A”;
- Polymorphic cache of degree N says “I’ve **only** seen A_1, …, A_N”;
- Megamorphic cache says “I’ve seen a lot of things.”;

```
// LdaNamedProperty <object> <
> <slot>
//
// Calls the LoadIC at FeedBackVector slot <slot> for <object> and the name at
// constant pool entry <name_index>.
IGNITION_HANDLER(LdaNamedProperty, InterpreterAssembler) {
  TNode<HeapObject> feedback_vector = LoadFeedbackVector();

  // Load receiver.
  TNode<Object> recv = LoadRegisterAtOperandIndex(0);

  // Load the name and context lazily.
  LazyNode<TaggedIndex> lazy_slot = [=] {
    return BytecodeOperandIdxTaggedIndex(2);
  };
  LazyNode<Name> lazy_name = [=] {
    return CAST(LoadConstantPoolEntryAtOperandIndex(1));
  };
  LazyNode<Context> lazy_context = [=] { return GetContext(); };

  Label done(this);
  TVARIABLE(Object, var_result);
  ExitPoint exit_point(this, &done, &var_result);

  AccessorAssembler::LazyLoadICParameters params(lazy_context, recv, lazy_name,
                                                 lazy_slot, feedback_vector);
  AccessorAssembler accessor_asm(state());
  accessor_asm.LoadIC_BytecodeHandler(&params, &exit_point);

  BIND(&done);
  {
    SetAccumulator(var_result.value());
    Dispatch();
  }
}
```

```
void BaselineCompiler::VisitLdaNamedProperty() {
  CallBuiltin<Builtin::kLoadICBaseline>(

RegisterOperand(0),   // object

Constant<Name>(1),    // name

IndexAsTagged(2));    // slot
}
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 1：判断对象的MAP是否过期

```
GotoIf(IsUndefined(p->vector()), &no_feedback);

TNode<Map> lookup_start_object_map =
    LoadReceiverMap(p->receiver_and_lookup_start_object());
GotoIf(IsDeprecatedMap(lookup_start_object_map), &miss);
```

```
BIND(&miss);
{
  Comment("LoadIC_BytecodeHandler_miss");

  exit_point->ReturnCallRuntime(Runtime::kLoadIC_Miss, p->context(),
                                p->receiver(), p->name(), p->slot(),
                                p->vector());
}
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 2: MAP过期,调用LoadIC_Miss

```
RUNTIME_FUNCTION(Runtime_LoadIC_Miss) {
  HandleScope scope(isolate);
  DCHECK_EQ(4, args.length());
  // Runtime functions don't follow the IC's calling convention.
  Handle<Object> receiver = args.at(0);
  Handle<Name> key = args.at<Name>(1);
  Handle<TaggedIndex> slot = args.at<TaggedIndex>(2);
  Handle<FeedbackVector> vector = args.at<FeedbackVector>(3);
  FeedbackSlot vector_slot = FeedbackVector::ToSlot(slot->value());
  FeedbackSlotKind kind = vector->GetKind(vector_slot);
  if (IsLoadICKind(kind)) {
    LoadIC ic(isolate, vector, vector_slot, kind);
    ic.UpdateState(receiver, key);
    RETURN_RESULT_OR_FAILURE(isolate, ic.Load(receiver, key));
  } else if (IsLoadGlobalICKind(kind)) {
……
}
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 2: MAP过期,调用LoadIC_Miss

```
RUNTIME_FUNCTION(Runtime_LoadIC_Miss) {
  HandleScope scope(isolate);
  DCHECK_EQ(4, args.length());
  // Runtime functions don't follow the IC's calling convention.
  Handle<Object> receiver = args.at(0);
  Handle<Name> key = args.at<Name>(1);
  Handle<TaggedIndex> slot = args.at<TaggedIndex>(2);
  Handle<FeedbackVector> vector = args.at<FeedbackVector>(3);
  FeedbackSlot vector_slot = FeedbackVector::ToSlot(slot->value());
  FeedbackSlotKind kind = vector->GetKind(vector_slot);
  if (IsLoadICKind(kind)) {
    LoadIC ic(isolate, vector, vector_slot, kind);
    ic.UpdateState(receiver, key);
    RETURN_RESULT_OR_FAILURE(isolate, ic.Load(receiver, key));
  } else if (IsLoadGlobalICKind(kind)) {
……
}
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 2: MAP过期,调用LoadIC_Miss

LoadIC::Load

```
bool use_ic = (state() != NO_FEEDBACK) && FLAG_use_ic && update_feedback;
 if (receiver.is_null()) {
   receiver = object;
 }
 if (IsAnyHas() ? !object->IsJSReceiver()
                : object->IsNullOrUndefined(isolate())) {……}
 // If we encounter an object with a deprecated map, we want to update the
 // feedback vector with the migrated map.
 // Mark ourselves as RECOMPUTE_HANDLER so that we don't turn megamorphic due
 // to seeing the same map and handler.
 if (MigrateDeprecated(isolate(), object)) {
   UpdateState(object, name);
 }
 JSObject::MakePrototypesFast(object, kStartAtReceiver, isolate());
 update_lookup_start_object_map(object);
 PropertyKey key(isolate(), name);
 LookupIterator it = LookupIterator(isolate(), receiver, key, object);
 // Named lookup in the object.
 LookupForRead(&it, IsAnyHas());
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 2: MAP过期,调用LoadIC_Miss

LoadIC::Load

```
if (name->IsPrivate()) {...}
if (it.IsFound() || !ShouldThrowReferenceError()) {
  // Update inline cache and stub cache.
  if (use_ic) {   UpdateCaches(&it);}  else if (state() == NO_FEEDBACK) {
    // Tracing IC stats
    IsLoadGlobalIC() ? TraceIC("LoadGlobalIC", name)
                     : TraceIC("LoadIC", name);  }
  if (IsAnyHas()) {
    // Named lookup in the object.
    Maybe<bool> maybe = JSReceiver::HasProperty(&it);
    if (maybe.IsNothing()) return MaybeHandle<Object>();
    return maybe.FromJust() ? ReadOnlyRoots(isolate()).true_value_handle()
                            : ReadOnlyRoots(isolate()).false_value_handle(); }
  // Get the property.
  Handle<Object> result;
  ASSIGN_RETURN_ON_EXCEPTION(
      isolate(), result, Object::GetProperty(&it, IsLoadGlobalIC()), Object);
  if (it.IsFound()) {  return result;  } else if (!ShouldThrowReferenceError()) { …. } }
 return ReferenceError(name);
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 2: MAP过期,调用LoadIC_Miss

```
void LoadIC::UpdateCaches(LookupIterator* lookup) {
  Handle<Object> handler;
  if (lookup->state() == LookupIterator::ACCESS_CHECK) { ….
  } else {
    ……..
    handler = ComputeHandler(lookup);
  }
  // Can't use {lookup->name()} because the LookupIterator might be in
  // "elements" mode for keys that are strings representing integers above
  // JSArray::kMaxIndex.
  SetCache(lookup->GetName(), handler);
  TraceIC("LoadIC", lookup->GetName());
}
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 2: MAP过期,调用LoadIC_Miss

```
void IC::SetCache() {
  DCHECK(IsHandler(*handler));
  // Currently only load and store ICs support non-code handlers.
  DCHECK(IsAnyLoad() || IsAnyStore() || IsAnyHas());
  switch (state()) {
    case NO_FEEDBACK: UNREACHABLE();
    case UNINITIALIZED: UpdateMonomorphicIC(handler, name); break;
    case RECOMPUTE_HANDLER:
    case MONOMORPHIC:
      if (IsGlobalIC()) { UpdateMonomorphicIC(handler, name); break;}
      V8_FALLTHROUGH;
    case POLYMORPHIC:
      if (UpdatePolymorphicIC(name, handler)) break;
      if (UpdateMegaDOMIC(handler, name)) break;
      if (!is_keyed() || state() == RECOMPUTE_HANDLER)  {CopyICToMegamorphicCache(name);}
      V8_FALLTHROUGH;
    case MEGADOM: ConfigureVectorState(MEGAMORPHIC, name); V8_FALLTHROUGH;
    case MEGAMORPHIC: UpdateMegamorphicCache(lookup_start_object_map(), name, handler);
       vector_set_ = true;
      break;
    case GENERIC: UNREACHABLE(); }
}
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 2: MAP过期,调用LoadIC_Miss

```
void IC::UpdateMonomorphicIC(const MaybeObjectHandle& handler,
                             Handle<Name> name) {
  DCHECK(IsHandler(*handler));
  ConfigureVectorState(name, lookup_start_object_map(), handler);
}

void IC::ConfigureVectorState(Handle<Name> name, Handle<Map> map,
                              const MaybeObjectHandle& handler) {
  if (IsGlobalIC()) {
    nexus()->ConfigureHandlerMode(handler);
  } else {
    // Non-keyed ICs don't track the name explicitly.
    if (!is_keyed()) name = Handle<Name>::null();
    nexus()->ConfigureMonomorphic(name, map, handler);
  }
  OnFeedbackChanged(IsLoadGlobalIC() ? "LoadGlobal" : "Monomorphic"); }
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 2: MAP过期,调用LoadIC_Miss

```
void FeedbackNexus::ConfigureMonomorphic(Handle<Name> name,
                                         Handle<Map> receiver_map,
                                         const MaybeObjectHandle& handler) {
  DCHECK(handler.is_null() || IC::IsHandler(*handler));
  if (kind() == FeedbackSlotKind::kStoreDataPropertyInLiteral) {
    SetFeedback(HeapObjectReference::Weak(*receiver_map), UPDATE_WRITE_BARRIER,
                *name);
  } else {
    if (name.is_null()) {
      SetFeedback(HeapObjectReference::Weak(*receiver_map),
                  UPDATE_WRITE_BARRIER, *handler);
    } else {
      Handle<WeakFixedArray> array = CreateArrayOfSize(2);
      array->Set(0, HeapObjectReference::Weak(*receiver_map));
      array->Set(1, *handler);
      SetFeedback(*name, UPDATE_WRITE_BARRIER, *array);
    }
  }
}
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 2: MAP过期,调用LoadIC_Miss

```
bool IC::UpdatePolymorphicIC(Handle<Name> name, const MaybeObjectHandle& handler) {
  Handle<Map> map = lookup_start_object_map();
  std::vector<MapAndHandler> maps_and_handlers;
  maps_and_handlers.reserve(FLAG_max_valid_polymorphic_map_count);
  int deprecated_maps = 0;
  int handler_to_overwrite = -1;
 { DisallowGarbageCollection no_gc;  int i = 0;  for (FeedbackIterator it(nexus()); !it.done(); it.Advance()) { ……}
  int number_of_maps = static_cast<int>(maps_and_handlers.size());
  int number_of_valid_maps =  number_of_maps - deprecated_maps - (handler_to_overwrite != -1);
  if (number_of_valid_maps >= FLAG_max_valid_polymorphic_map_count) return false;
  if (number_of_maps == 0 && state() != MONOMORPHIC && state() != POLYMORPHIC) { return false;}
  number_of_valid_maps++;
  if (number_of_valid_maps == 1) {
    ConfigureVectorState(name, lookup_start_object_map(), handler);
  } else {
    if (is_keyed() && nexus()->GetName() != *name) return false;
    if (handler_to_overwrite >= 0) { ….} else {
      maps_and_handlers.push_back(MapAndHandler(map, handler));
    }
    ConfigureVectorState(name, maps_and_handlers);
  } return true;
}
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 2: MAP过期,调用LoadIC_Miss

```
void IC::ConfigureVectorState(
    Handle<Name> name, std::vector<MapAndHandler> const& maps_and_handlers) {
  DCHECK(!IsGlobalIC());
  // Non-keyed ICs don't track the name explicitly.
  if (!is_keyed()) name = Handle<Name>::null();
  nexus()->ConfigurePolymorphic(name, maps_and_handlers);
  OnFeedbackChanged("Polymorphic");
}
void FeedbackNexus::ConfigurePolymorphic(
    Handle<Name> name, std::vector<MapAndHandler> const& maps_and_handlers) {
  int receiver_count = static_cast<int>(maps_and_handlers.size());
  DCHECK_GT(receiver_count, 1);
  Handle<WeakFixedArray> array = CreateArrayOfSize(receiver_count * 2);
  for (int current = 0; current < receiver_count; ++current) { ….}
  if (name.is_null()) {
    SetFeedback(*array, UPDATE_WRITE_BARRIER, UninitializedSentinel(),
                SKIP_WRITE_BARRIER);
  } else {
    SetFeedback(*name, UPDATE_WRITE_BARRIER, *array);
  }
}
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 2: MAP过期,调用LoadIC_Miss

```
void IC::ConfigureVectorState(
    Handle<Name> name, std::vector<MapAndHandler> const& maps_and_handlers) {
  DCHECK(!IsGlobalIC());
  // Non-keyed ICs don't track the name explicitly.
  if (!is_keyed()) name = Handle<Name>::null();
  nexus()->ConfigurePolymorphic(name, maps_and_handlers);
  OnFeedbackChanged("Polymorphic");
}
void FeedbackNexus::ConfigurePolymorphic(
    Handle<Name> name, std::vector<MapAndHandler> const& maps_and_handlers) {
  int receiver_count = static_cast<int>(maps_and_handlers.size());
  DCHECK_GT(receiver_count, 1);
  Handle<WeakFixedArray> array = CreateArrayOfSize(receiver_count * 2);
  for (int current = 0; current < receiver_count; ++current) { ….}
  if (name.is_null()) {
    SetFeedback(*array, UPDATE_WRITE_BARRIER, UninitializedSentinel(),
                SKIP_WRITE_BARRIER);
  } else {
    SetFeedback(*name, UPDATE_WRITE_BARRIER, *array);
  }
}
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 3: TryMonomorphicCase

```
TVARIABLE(MaybeObject, var_handler);
  Label try_polymorphic(this), if_handler(this, &var_handler);

TNode<MaybeObject> feedback = TryMonomorphicCase(
      p->slot(), CAST(p->vector()), lookup_start_object_map, &if_handler,
      &var_handler, &try_polymorphic);
```

TryMonomorphicCase

```
TNode<IntPtrT> offset = ElementOffsetFromIndex(slot, HOLEY_ELEMENTS);
TNode<MaybeObject> feedback = ReinterpretCast<MaybeObject>(
    Load(MachineType::AnyTagged(), vector,
        IntPtrAdd(offset, IntPtrConstant(header_size))));

// Try to quickly handle the monomorphic case without knowing for sure
// if we have a weak reference in feedback.
GotoIfNot(IsWeakReferenceTo(feedback, lookup_start_object_map), if_miss);
TNode<MaybeObject> handler = UncheckedCast<MaybeObject>(
    Load(MachineType::AnyTagged(), vector,
        IntPtrAdd(offset, IntPtrConstant(header_size + kTaggedSize))));

*var_handler = handler;
Goto(if_handler);
return feedback;
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 4: HandleLoadICHandlerCase

```
BIND(&if_handler);
HandleLoadICHandlerCase(p, CAST(var_handler.value()), &miss, exit_point);
```

HandleLoadICHandlerCase

```
  Label if_smi_handler(this, {&var_holder, &var_smi_handler});
  Label try_proto_handler(this, Label::kDeferred),
      call_handler(this, Label::kDeferred);
  Branch(TaggedIsSmi(handler), &if_smi_handler, &try_proto_handler);
  BIND(&try_proto_handler);
{   GotoIf(IsCodeT(CAST(handler)), &call_handler);
    HandleLoadICProtoHandler(p, CAST(handler), &var_holder, &var_smi_handler,&if_smi_handler, miss, exit_point, ic_mode, access_mode);}
  // |handler| is a Smi, encoding what to do. See SmiHandler methods
  // for the encoding format.
  BIND(&if_smi_handler);
  { HandleLoadICSmiHandlerCase(p, var_holder.value(), CAST(var_smi_handler.value()), handler, miss, exit_point, ic_mode, on_nonexistent,
support_elements, access_mode);}
  BIND(&call_handler);
  {// TODO(v8:11880): avoid roundtrips between cdc and code.
    TNode<Code> code_handler = FromCodeT(CAST(handler));
    exit_point->ReturnCallStub(LoadWithVectorDescriptor{}, code_handler,
                               p->context(), p->lookup_start_object(),
                               p->name(), p->slot(), p->vector());
  }
}
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 4: HandleLoadICHandlerCase

HandleLoadICSmiHandlerCase

```
if (access_mode == LoadAccessMode::kHas) {
   HandleLoadICSmiHandlerHasNamedCase(p, holder, handler_kind, miss, exit_point, ic_mode);
 } else { HandleLoadICSmiHandlerLoadNamedCase(p, holder, handler_kind, handler_word,
&rebox_double, &var_double_value, handler, miss, exit_point, ic_mode, on_nonexistent, support_elements);}
```

HandleLoadICSmiHandlerLoadNamedCase

```
GotoIf(WordEqual(handler_kind, LOAD_KIND(kField)), &field);

GotoIf(WordEqual(handler_kind, LOAD_KIND(kConstantFromPrototype)), &constant);

GotoIf(WordEqual(handler_kind, LOAD_KIND(kNonExistent)), &nonexistent);

GotoIf(WordEqual(handler_kind, LOAD_KIND(kNormal)), &normal);

GotoIf(WordEqual(handler_kind, LOAD_KIND(kAccessor)), &accessor);

GotoIf(WordEqual(handler_kind, LOAD_KIND(kNativeDataProperty)),
      &native_data_property);

GotoIf(WordEqual(handler_kind, LOAD_KIND(kApiGetter)), &api_getter);

GotoIf(WordEqual(handler_kind, LOAD_KIND(kApiGetterHolderIsPrototype)),
      &api_getter);

GotoIf(WordEqual(handler_kind, LOAD_KIND(kGlobal)), &global);
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 4: HandleLoadICHandlerCase

HandleLoadICSmiHandlerCase

```
if (access_mode == LoadAccessMode::kHas) {
    HandleLoadICSmiHandlerHasNamedCase(p, holder, handler_kind, miss, exit_point, ic_mode);
  } else { HandleLoadICSmiHandlerLoadNamedCase(p, holder, handler_kind, handler_word,
&rebox_double, &var_double_value, handler, miss, exit_point, ic_mode, on_nonexistent, support_elements);}
```

HandleLoadICSmiHandlerLoadNamedCase

```
 BIND(&field);
  {
    CSA_DCHECK(this, IsClearWord<LoadHandler::IsWasmStructBits>(handler_word));
    HandleLoadField(CAST(holder), handler_word, var_double_value, rebox_double,
                    miss, exit_point);

}
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 4: HandleLoadICHandlerCase

HandleLoadICSmiHandlerCase

```
if (access_mode == LoadAccessMode::kHas) {
  HandleLoadICSmiHandlerHasNamedCase(p, holder, handler_kind, miss, exit_point, ic_mode);
 } else { HandleLoadICSmiHandlerLoadNamedCase(p, holder, handler_kind, handler_word,
&rebox_double, &var_double_value, handler, miss, exit_point, ic_mode, on_nonexistent, support_elements);}
```

HandleLoadField

```
 Branch(IsSetWord<LoadHandler::IsInobjectBits>(handler_word),
&inobject,&out_of_object);
 BIND(&inobject);
  {Label is_double(this);
   GotoIf(IsSetWord<LoadHandler::IsDoubleBits>(handler_word),
&is_double);
   exit_point->Return(LoadObjectField(holder, offset));
   BIND(&is_double);
   TNode<Object> heap_number = LoadObjectField(holder, offset);
   GotoIf(TaggedIsSmi(heap_number), miss);
   GotoIfNot(IsHeapNumber(CAST(heap_number)), miss);
   *var_double_value = LoadHeapNumberValue(CAST(heap_number));
   Goto(rebox_double);}
```

```
Node*
CodeAssembler::LoadFromObject(MachineType type,
TNode<Object> object,
                              TNode<IntPtrT> offset) {
  return raw_assembler()->LoadFromObject(type,
object, offset);}
Node* LoadFromObject(MachineType type, Node*
base, Node* offset) {
   DCHECK_IMPLIES(V8_MAP_PACKING_BOOL &&
IsMapOffsetConstantMinusTag(offset),
                  type == MachineType::MapInHeader());
   ObjectAccess access = {type,
WriteBarrierKind::kNoWriteBarrier};
   Node* load =
AddNode(simplified()->LoadFromObject(access),
base, offset);
   return load; }
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 4: HandleLoadICHandlerCase

HandleLoadICSmiHandlerCase

```
if (access_mode == LoadAccessMode::kHas) {
   HandleLoadICSmiHandlerHasNamedCase(p, holder, handler_kind, miss, exit_point, ic_mode);
  } else { HandleLoadICSmiHandlerLoadNamedCase(p, holder, handler_kind, handler_word,
&rebox_double, &var_double_value, handler, miss, exit_point, ic_mode, on_nonexistent, support_elements);}
```

HandleLoadField

```
 Branch(IsSetWord<LoadHandler::IsInobjectBits>(handler_word),
&inobject,&out_of_object);
 BIND(&inobject);
  {Label is_double(this);
    GotoIf(IsSetWord<LoadHandler::IsDoubleBits>(handler_word),
&is_double);
    exit_point->Return(LoadObjectField(holder, offset));
    BIND(&is_double);
    TNode<Object> heap_number = LoadObjectField(holder, offset);
    GotoIf(TaggedIsSmi(heap_number), miss);
    GotoIfNot(IsHeapNumber(CAST(heap_number)), miss);
    *var_double_value = LoadHeapNumberValue(CAST(heap_number));
    Goto(rebox_double);}
  BIND(&out_of_object);
  { Label is_double(this);
    TNode<HeapObject> properties =
LoadFastProperties(holder);
    TNode<Object> value = LoadObjectField(properties,
offset);

GotoIf(IsSetWord<LoadHandler::IsDoubleBits>(handler
_word), &is_double);
    exit_point->Return(value);
    BIND(&is_double);
    GotoIf(TaggedIsSmi(value), miss);
    GotoIfNot(IsHeapNumber(CAST(value)), miss);
    *var_double_value =
LoadHeapNumberValue(CAST(value));
    Goto(rebox_double);}
```

void AccessorAssembler::LoadIC_BytecodeHandler

Step 4: HandleLoadICHandlerCase

```
BIND(&try_polymorphic);
{
  TNode<HeapObject> strong_feedback =
      GetHeapObjectIfStrong(feedback, &miss);
  GotoIfNot(IsWeakFixedArrayMap(LoadMap(strong_feedback)), &stub_call);
  HandlePolymorphicCase(lookup_start_object_map, CAST(strong_feedback),
                        &if_handler, &var_handler, &miss);}
```

HandlePolymorphicCase

```
TNode<IntPtrT> length = LoadAndUntagWeakFixedArrayLength(feedback);
Label loop(this, &var_index), loop_next(this);
Goto(&loop); BIND(&loop);
  {  TNode<MaybeObject> maybe_cached_map = LoadWeakFixedArrayElement(feedback, var_index.value());
  CSA_DCHECK(this, IsWeakOrCleared(maybe_cached_map));
  GotoIfNot(IsWeakReferenceTo(maybe_cached_map, lookup_start_object_map), &loop_next);
  TNode<MaybeObject> handler = LoadWeakFixedArrayElement(feedback, var_index.value(), kTaggedSize);
  *var_handler = handler;
  Goto(if_handler);
  BIND(&loop_next);
  var_index =  Signed(IntPtrSub(var_index.value(), IntPtrConstant(kEntrySize)));
  Branch(IntPtrGreaterThanOrEqual(var_index.value(), IntPtrConstant(0)),
        &loop, if_miss);
```