PyTorch JIT Source Code Read Note (Updated at Feb 2020)

zasdfgbnm.github.io/2020/02/07/PyTorch-JIT-Source-Code-Read-Note-Updated-202002/

February 7, 2020

This is my note for reading PyTorch's JIT source. We begin by looking at `torch.jit.script` to find the frontend that compiles the Python code into PyTorch's tree views, and the backend that compiles tree views to graph. We also read the structure of the internal representation of PyTorch's graph. Finally we go to graph executor to look at how the computation graph is further compiled into instructions and how the action of these instructions are defined and executed.

PyTorch is under very active development. So the PyTorch's source code at the time the reader reading this article won't be the same as when I wrote this article. To get the same source code as in this article, the readers could run the following command:

```
git checkout
c6fa6d82aebc1dcd7561ea29ec5d41c5a211bae1
```

Starting point: torch.jit.script

In PyTorch, a Python function can be just-in-time compiled by doing something like:

```
@torch.jit.s
cript
def f(x):
    return x
+ x
```

the `torch.jit.script` is a decorator of your function `f`. If you are unfamiliar with Python's decorator, please refer to this article.

We will start by looking at `torch.jit.script`. To read `torch.jit.script`, we begin by looking at `torch/jit/__init__.py`. To quickly locate `script`, search `def script` in your editor, and you will immediately find it:

```
def script(obj, optimize=True, _frames_up=0):
    ....
    if inspect.isclass(obj):
      ...
    else:
      _check_directly_compile_overloaded(obj)
      maybe_already_compiled_fn = _try_get_jit_cached_function(obj)
      if maybe_already_compiled_fn:
          return maybe_already_compiled_fn
      ast = get_jit_def(obj)
      if _rcb is None:
          _rcb = _jit_internal.createResolutionCallbackFromClosure(obj)
      fn = torch._C._jit_script_compile(qualified_name, ast, _rcb,
get_default_args(obj))
      # Forward docstrings
      fn.__doc__ = obj.__doc__
      _set_jit_function_cache(obj, fn)
      return fn
```

Here we want to quickly get a big picture of `torch.jit`, so we will only look at how a function is compiled. We will ignore the compilation of a module or class, etc.

The core of the above code is this four lines

```
ast = get_jit_def(obj)
if _rcb is None:
    _rcb = _jit_internal.createResolutionCallbackFromClosure(obj)
fn = torch._C._jit_script_compile(qualified_name, ast, _rcb,
get_default_args(obj))
```

Just by reading the English, we can tell that what it does is roughly get the abstract syntax tree(AST) and then call `torch._C._jit_script_compile` to compile it to PyTorch's internal representation.

From the beginning of `__init__.py`, we know that `get_jit_def` is imported from `torch.jit.frontend`. From the name of this function and its owning module, we can tell that this is the frontend of PyTorch's JIT compiler that compiles the source code of the scripted function into AST.

Then `createResolutionCallbackFromClosure` is called. This function is from `_jit_internal.py`, by looking at its definition and the codes around, we can tell that it roughly gets the symbols available to the function so that they could be accessed through

C++ when executing the graph. We will not go into details about how this works.

The next line uses `torch._C._jit_script_compile` to compiles the AST obtained in the previous step into computation graph. The `torch._C` tells us that `_jit_script_compile` is implemented in C++.

The Python frontend

A good starting point of the frontend is the `get_jit_def` we just saw. This function is defined at `torch/jit/frontend.py`. The code is:

```
def get_jit_def(fn, self_name=None):
    sourcelines, file_lineno, filename = get_source_lines_and_file(fn)
    source = ''.join(sourcelines)
    dedent_src = dedent(source)
    py_ast = ast.parse(dedent_src)
    if len(py_ast.body) != 1 or not isinstance(py_ast.body[0], ast.FunctionDef):
        raise RuntimeError("Expected a single top-level function")
    leading_whitespace_len = len(source.split('\n', 1)[0]) -
len(dedent_src.split('\n', 1)[0])
    type_line = torch.jit.annotations.get_type_line(source)
    ctx = SourceContext(source, filename, file_lineno, leading_whitespace_len,
_uses_true_division(fn))
    return build_def(ctx, py_ast.body[0], type_line, self_name)
```

The first 7 lines of function body just use the standard tools provided by Python, `dedent`, `inspect`, and `ast`, to construct the Python AST, do some check to make sure the thing being compiled is “a single top-level function”, as well as getting the position, line numbers, etc. so that useful information could be printted to the user for debugging when there is an error.

The following line `type_line = torch.jit.annotations.get_type_line(source)` is interesting. After looking at `torch/jit/annotations.py`, we can see that PyTorch’s JIT allows the user to specify the type of arguments and return value by writing something like `# type: (Tensor, torch.Tensor) -> Tuple[Tensor, Tensor]`.

In the next line `ctx = SourceContext(.....)`, the `_uses_true_division` is defined in the same file to handle the different behavior of `/` in Python2 with or without `from __future__ import division` (see PEP 238 for the difference). The `SourceContext` is also defined in the same file. It is a subclass of `SourceRangeFactory` with additional field to store if the division is true division. The `SourceRangeFactory` is imported by `from torch._C._jit_tree_views import *`. After reading its definition at `torch/csrc/jit/script/python_tree_views.cpp`, we can see that this is basically a class designed to store the range of source code, e.g. where in the source code a token is located.

The core is the `build_def` in the last line, so we move on:

```
def build_def(ctx, py_def, type_line, self_name=None):
    body = py_def.body
    r = ctx.make_range(py_def.lineno + len(py_def.decorator_list),
                       py_def.col_offset,
                       py_def.col_offset + len("def"))
    param_list = build_param_list(ctx, py_def.args, self_name)
    return_type = None
    if getattr(py_def, 'returns', None) is not None:
        return_type = build_expr(ctx, py_def.returns)
    decl = Decl(r, param_list, return_type)
    is_method = self_name is not None
    if type_line is not None:
        type_comment_decl = torch._C.parse_type_comment(type_line)
        decl = torch._C.merge_type_from_type_comment(decl, type_comment_decl,
is_method)
    return Def(Ident(r, py_def.name),
               decl,
               build_stmts(ctx, body))
```

Reading through this, we can see that what basically this does is to convert the Python's AST into the internal representation. Names like `Decl`, `Def`, `Ident` are all imported by `from torch._C._jit_tree_views import *`. In the last line, we can see that the function body is constructed by `build_stmts`, so let's go further to read `build_stmts`:

```
def build_stmts(ctx, stmts):
    stmts = [build_stmt(ctx, s) for s in
stmts]
    return list(filter(None, stmts))
```

This is a very simple function: call `build_stmt` for each item and filter out those not needed. But what is `build_stmt`? It is defined as: `build_stmt = StmtBuilder()`. The definition of `StmtBuilder` looks like:

```
class StmtBuilder(Builder):
    # ...
    @staticmethod
    def build_Expr(ctx, stmt):
        value = stmt.value
        if value.__class__.__name__ == 'Str':
            # If a statement is a string literal
expression,
            # then it is a docstring. Just ignore it.
            return None
        else:
            return ExprStmt(build_expr(ctx, value))
    # ...
    @staticmethod
    def build_Assign(ctx, expr):
        # ...
    # ...
    @staticmethod
    def build_AnnAssign(ctx, stmt):
        #...
    # ......
```

We can see that, this is a class with many static methods that define what to do for different types of Python AST. I will not go deep into how each type is handled. Since at this point, the readers should be able to catch all the details on how each type of nodes in Python AST are dealt with by themselves. So We will stop our frontend reading right here.

From Python AST to PyTorch IR: part 1

Now let's move on to read `_jit_script_compile`. To find where it is located, simply run the command `grep _jit_script_compile -r .`. We will find something like:

```
./torch/csrc/jit/script/init.cpp:
"_jit_script_compile",
```

So, `torch/csrc/jit/script/init.cpp` would be a good start point. The complete definition of `_jit_script_compile` is:

```
m.def(
    "_jit_script_compile",
    [](const std::string& qualname,
       const Def& def,
       ResolutionCallback rcb,
       const FunctionDefaults& defaults) {
      C10_LOG_API_USAGE_ONCE("torch.script.compile");
      const auto name = c10::QualifiedName(qualname);
      TORCH_INTERNAL_ASSERT(name.name() == def.name().name());
      return script_compile_function(name, def, defaults,
std::move(rcb));
    });
```

So, let's move on to `script_compile_function`. This is a function defined in the same file

```
static StrongFunctionPtr script_compile_function(
    const c10::QualifiedName& name,
    const Def& def,
    const FunctionDefaults& defaults,
    ResolutionCallback rcb) {
  auto cu = get_python_cu();
  auto defined_functions = cu->define(
      QualifiedName(name.prefix()),
      {def},
      {pythonResolver(std::move(rcb))},
      nullptr,
      true);
  TORCH_INTERNAL_ASSERT(defined_functions.size() == 1);
  auto& defined = defined_functions[0];
  defined->setSchema(getSchemaWithNameAndDefaults(
      def.range(), defined->getSchema(), def.name().name(),
defaults));
  StrongFunctionPtr ret(std::move(cu), defined);
  didFinishEmitFunction(ret);
  return ret;
}
```

Which part of this function does the acutal job of compiling tree views to PyTorch IR? It could be `cu->define`, or constructor of `StrongFunctionPtr` or `didFinishEmitFunction`. It is not clear right now, but seems mostly likely to be in `cu->define`, but let's skim through these functions:

By greping `get_python_cu`, we can find this line

```
inline std::shared_ptr<script::CompilationUnit>
get_python_cu() {
```

which tells us that the `cu` in the code is a `CompilationUnit`. By greping `CompilationUnit::define`, we find its definition in `torch/csrc/jit/script/compiler.cpp`:

```
std::unique_ptr<Function> CompilationUnit::define(
    const c10::optional<QualifiedName>& prefix,
    const Def& def,
    const ResolverPtr& resolver,
    const Self* self,
    const std::unordered_map<std::string, Function*>&
function_table,
    bool shouldMangle) const {
  // .......
  auto creator = [def, _resolver, self](Function& method) {
    // .......
    to_ir(def, _resolver, self, method);
  };
  // .......
  auto fn = torch::make_unique<Function>(
      std::move(name), std::make_shared<Graph>(), creator);
  if (self) {
    // Register this as a method on `self`'s type
    self->getClassType()->addMethod(fn.get());
  }
  return fn;
}
```

Less important parts of the code is omitted. From above, we can find that the core of compiling an AST into a compute graph is done at `to_ir`. It is defined in the same file. Skimming through `to_ir` we find that it is a struct of ~3000 lines of code, with member functions that handles different cases of Python AST. Without knowing PyTorch's IR, it's not easy to understand what `to_ir` does. So let's pause a little bit to take a look at PyTorch IR and come back later.

The PyTorch IR

PyTorch now has a very detailed design doc at `torch/csrc/jit/docs/OVERVIEW.md`. We should read through this document before coming back.

From Python AST to PyTorch IR: part 2

Let's continue the reading by looking at the definition of `to_ir`:

```
to_ir(
    const Def& def,
    ResolverPtr resolver_,
    const Self* self,
    Function& method) // method being constructed
    : method(method),
      graph(method.graph()),
      resolver(std::move(resolver_)),
      typeParser_(resolver),
      environment_stack(nullptr) {
  AT_ASSERT(resolver);
  pushFrame(graph->block(), /*starts_def=*/true);

  // Type annotations exclude explicitly typing the "self" parameter,
so in
  // the case that this is a method with self we expect one fewer
parameter
  // annotation than the number of parameters this Def takes.
  if (self && def.decl().params().size() == 0) {
    throw ErrorReport(def.decl().params().range())
        << "methods must have a self argument";
  }
  method.setSchema(emitDef(def, self, graph->block()));

  // NB ORDERING: SSA conversion has to occur before
  // lifting of closures and forks, this way closures are converted
  // to SSA while part of their original graph, and closures are ready
to
  // be inlined into forked closures
  ConvertToSSA(graph);
  // convert loops with an iter and body condition specified to
  // python-recognize while loops. we do this so they can be exported,
  // and run the pass early to avoid jitter. Like conversion to SSA,
  // it only needs to run once.
  CanonicalizeModifiedLoops(graph);

  runCleanupPasses(graph);
}
```

we can clearly see that we first convert from tree view to PyTorch IR in `emitDef`, the convert the PyTorch IR to SSA in `ConvertToSSA`, and then do some special handling of loops and then cleanup. Looking around, it is also not hard to find that `ConvertToSSA` is defined in `torch/csrc/jit/script/convert_to_ssa.cpp`. Roughly the flow chart is:

```
Python AST --[torch.jit.frontend]--> Tree View --[compiler.cpp]--> PyTorch IR --
[convert_to_ssa.cpp]--> SSA ----> canonicalized and cleanuped SSA
```

Now let's continue looking at `emitDef` for the `Tree View --> PyTorch IR` conversion:

```
FunctionSchema emitDef(const Def& def, const Self* self, Block* block)
{
  auto schema = typeParser_.parseSchemaFromDef(def, bool(self));
  // TODO need guards on init returning none
  if (schema.returns().size() == 1) {
    def_stack_.back().declared_return_type_ =
schema.returns().at(0).type();
  }
  std::vector<Argument> arguments =
      emitFormalArguments(def, self, schema, block);

  // body
  auto stmts_list = def.statements();
  emitStatements(stmts_list.begin(), stmts_list.end());
  handleMaybeNoReturn(def, block);
  std::vector<Argument> returns = {emitOutput(def.range(), schema,
block)};
  return {def.name().name(), "", std::move(arguments),
std::move(returns)};
}
```

Basically what it does is:

1. parse schema and arguments
2. compile function body by `emitStatements`
3. handle return

We will not go into the details about schema and how the return is handled. We only continue looking at `emitStatements`:

```
void emitStatements(
    List<Stmt>::const_iterator begin,
    List<Stmt>::const_iterator end) {
  for (; begin != end; ++begin) {
    auto stmt = *begin;
    ErrorReport::CallStack::update_pending_range(stmt.range());
    switch (stmt.kind()) {
      case TK_IF:
        emitIf(If(stmt));
        break;
      case TK_WHILE:
        emitWhile(While(stmt));
        break;
      case TK_FOR:
        emitFor(For(stmt));
        break;
      case TK_ASSIGN:
        emitAssignment(Assign(stmt));
        break;
      case TK_AUG_ASSIGN:
        emitAugAssignment(AugAssign(stmt));
        break;
      case TK_EXPR_STMT: {
        auto expr = ExprStmt(stmt).expr();
        emitSugaredExpr(expr, 0);
      } break;
      // .......
      default:
        throw ErrorReport(stmt)
            << "Unrecognized statement kind " <<
kindToString(stmt.kind());
    }
  }
}
```

we can see that the handle of each kind of statement is dispatched by `stmt.kind()`. For each specialized emit, we would expect it take the tree view as input, recursively traverse the tree view to emit the code. Let's first take a look at a simple specialization of this kind of emit:

```
// Currently we do not support assigning exceptions to
variables,
// a = Exception("hi")
// raise a
//
// We ignore the expression following raise
void emitRaise(const SourceRange& loc) {
  const std::string exception = "Exception";
  auto string_input = insertConstant(*graph, exception,
loc);
  graph->insert(prim::RaiseException, {string_input}, {},
loc);
  exit_blocks.insert(environment_stack->block());
}
```

we see that it just insert a `prim::RaiseException` into the graph.

When designing a compiler, expression is always an important sublanguage to pay attention to. Let's take a look at how PyTorch JIT handles expression here. The place to look at is `emitSugaredExpr`. Before we start, it worth mentioning that, Python is a dynamically typed language, while PyTorch IR is statically typed. So we would expect `emitSugaredExpr` to have a type system to infer types. The `emitSugaredExpr` is:

```
// any expression that can produce a SugaredValue is handled here
// expressions that only return a single Value* are handled in
emitSimpleExpr
// type_hint is set if there is a type that this value is expected to be
// e.g. a : List[int] = []
// or a = torch.jit.annotate(List[int], [])
// the caller is responsible for checking that the result matches
type_hint
// emitSugaredExpr is free to ignore it.
std::shared_ptr<SugaredValue> emitSugaredExpr(
    const Expr& tree,
    size_t n_binders,
    const TypePtr& type_hint = nullptr) {
  switch (tree.kind()) {
    case TK_VAR:
      return environment_stack->getSugaredVar(Var(tree).name());
    case '.': {
      auto select = Select(tree);
      auto sv = emitSugaredExpr(select.value(), 1);
      return sv->attr(select.range(), method, select.selector().name());
    }
    case TK_APPLY: {
      auto apply = Apply(tree);
      return emitApplyExpr(apply, n_binders);
    } break;
    default:
      return std::make_shared<SimpleValue>(emitSimpleExpr(tree,
type_hint));
  }
}
```

Here we will only go deep into simple expressions. An example of simple expression is `a + b`, and we will look at how this specific example is handled. Now let's look at `emitSimpleExpr`, it is:

```
Value* emitSimpleExpr(
    const TreeRef& tree,
    const TypePtr& type_hint = nullptr) {
  switch (tree->kind()) {
    // .....
    case TK_IN:
    case TK_POW:
    case TK_NE:
    case TK_EQ:
    case '<':
    case '>':
    case TK_LE:
    case TK_GE:
    case '*':
    case '/':
    case '+':
    case '-':
    case '%':
    case '&':
    case '|':
    case '^': {
      const auto& inputs = tree->trees();
      auto kind = getNodeKind(tree->kind(), inputs.size());
      auto overload = getOperatorOverload(tree->kind(), inputs.size());
      auto named_values = getNamedValues(inputs,
/*maybe_unpack=*/false);

      if (tree->kind() == TK_IN) {
        // For `in` the arguments are in reverse order (the object being
        // checked is second)
        std::iter_swap(named_values.begin() + 0, named_values.begin() +
1);
      }

      return asSimple(
          makeMagic(
              overload, std::make_shared<BuiltinFunction>(kind,
at::nullopt))
              ->call(tree->range(), method, named_values, {}, 0));
    }
    // ....
    default:
      throw ErrorReport(tree) << "Cannot emit expr for: " << tree;
  }
}
```

We can see that, the actual work is done inside `BuiltinFunction::call`. This function is defined in `sugared_value.cpp` as:

```
std::shared_ptr<SugaredValue> BuiltinFunction::call(
    const SourceRange& loc,
    Function& m,
    at::ArrayRef<NamedValue> inputs,
    at::ArrayRef<NamedValue> attributes,
    size_t n_binders) {
  return std::make_shared<SimpleValue>(
      emitBuiltinCall(loc, *m.graph(), symbol, inputs, attributes,
self));
}
```

where `emitBuiltinCall` is defined in `schema_matching.cpp` as:

```
// Search for operators matching the provided symbol name and input types.
// If one is found, emit a node to the graph for that operator.
Value* emitBuiltinCall(
    const SourceRange& loc,
    Graph& graph,
    Symbol name,
    at::ArrayRef<NamedValue> inputs,
    at::ArrayRef<NamedValue> attributes,
    const c10::optional<NamedValue>& self) {
  const auto& variants = getAllOperatorsFor(name);
  const auto& builtin_functions = getAllBuiltinFunctionsFor(name);

  std::stringstream failure_messages;
  std::vector<const FunctionSchema*> schemas;
  for (const std::shared_ptr<Operator>& op : variants) {
    schemas.push_back(&op->schema());
  }
  for (const auto method : builtin_functions) {
    method->ensure_defined();
    schemas.push_back(&method->getSchema());
  }

  // no operators found with the same name, print out similarly named operators
  if (schemas.size() == 0) {
    const auto close_symbols = findSimilarOperators(name);
    auto error = ErrorReport(loc);
    const auto& user_function_name = name.toQualString();
    error << "Unknown builtin op: " << user_function_name << ".\n";
    if (close_symbols.size() == 0) {
      error
          << "Could not find any similar ops to " << user_function_name
          << ". This op may not exist or may not be currently supported in
TorchScript.\n";
    } else {
      error << "Here are some suggestions: \n";
      for (const auto& sym : close_symbols) {
        error << "\t" << sym.toQualString() << "\n";
      }
      error << "\nThe original call is";
    }
    throw error;
  }

  auto matched = matchSchemas(schemas, loc, graph, inputs, attributes, self);

  if (matched.first < variants.size()) {
    return emitBuiltinNode(matched.second, loc, graph, name);
  } else {
    Function* fn = builtin_functions[matched.first - variants.size()];
    // we inline builtin calls because they are normally very small
    // wrappers and are not useful for keeping around to debug
    return insertGraph(graph, *fn->graph(), matched.second.inputs).at(0);
  }
}
```

we can see that it look up the database of schemas and find the matching one, then use `emitBuiltinNode` to emit PyTorch IR. `emitBuiltinNode` is implemented as:

```
// Given a successful match between operator schema and symbol, emit a
node
// with the appropriate inputs and outputs.
static Value* emitBuiltinNode(
    const MatchedSchema& matched_schema,
    const SourceRange& loc,
    Graph& graph,
    Symbol name) {
  auto n = graph.insertNode(graph.create(name, matched_schema.inputs,
0))
               ->setSourceRange(loc);

  for (auto& ret : matched_schema.return_types) {
    n->addOutput()->setType(ret);
  }

  // assert that we did indeed create an op that has implementation
  // otherwise schema and dispatch are not in sync
  getOperation(n);

  return packOutputs(graph, n->outputs(),
matched_schema.return_field_names);
}
```

we can see that the type is infered by the schema in the following line

```
for (auto& ret :
matched_schema.return_types) {
  n->addOutput()->setType(ret);
}
```

Reaching this point, we have already get the big picture of the `Tree View --> PyTorch IR` : dispatch by node kind of tree view, and use schemas to do type inference.

After `to_ir` called, we should have a graph of PyTorch IR which is staticly typed. As described before, this graph will be converted to SSA and loops will be canonicalized. The conversion to SSA is a pretty standard task of compilers in general. Readers could refer to Wikipedia for how this could be done and read `convert_to_ssa.cpp` in PyTorch for detail. We will not go into details about loop canonicalization either.

The Graph Executor

Now we have seen how the compilation is done and what does PyTorch JIT's IR looks like, the thing left is how the IR are executed. As we have already seen in From Python AST to PyTorch IR: part 1, `script_compile_function` returns a pointer to a class `Function`. By looking at the implementation of `Function` at `torch/csrc/jit/function.{h, cpp}` we can easily see how a graph is executed:

In `functions.h`:

```
#pragma once
#include
<torch/csrc/jit/graph_executor.h>
```

```
std::shared_ptr<Graph> optimized_graph() const {
  std::lock_guard<std::recursive_mutex>
lock(compile_mutex);
  if (optimized_graph_) {
    return *optimized_graph_;
  }
  optimized_graph_ = graph_->copy();
  preoptimizeGraph(*optimized_graph_);
  return *optimized_graph_;
}
```

```
GraphExecutor& get_executor() {
  ensure_defined();
  std::lock_guard<std::recursive_mutex>
lock(compile_mutex);
  if (executor_) {
    return executor_;
  }
  check_single_output();
  executor_ = GraphExecutor(optimized_graph());
  return executor_;
}
```

and in `functions.cpp`:

```
// functions.cpp
#include
<torch/csrc/jit/passes/inliner.h>
```

```
void Function::run(Stack& stack) {
  get_executor().run(stack);
}

void Function::run(Stack&& stack) {
  run(stack);
}

IValue Function::operator()(
    std::vector<IValue> stack,
    const Kwargs& kwargs) {
  getSchema().checkAndNormalizeInputs(stack,
kwargs);
  run(stack);
  return stack.front();
}
```

```
void preoptimizeGraph(std::shared_ptr<Graph>& graph) {
  // TODO: Invoke cleanup passes before and after inlining to reduce
amount of
  // code we're copying.
  Inline(*graph);
}
```

We can see that, when we want to execute a graph, PyTorch first inline the graph using the inliner defined in `torch/csrc/jit/passes/inliner.h`, then create a `GraphExecutor` for the inlined graph. We will not go into details on how the graph is inlined. We will move to `GraphExecutor` instead.

The `GraphExecutor` is defined in `torch/csrc/jit/graph_executor.{h, cpp}`.

The constructor and `run` tells us that `GraphExecutor` is just a wrapper of `GraphExecutorImplBase`:

```
GraphExecutor::GraphExecutor(std::shared_ptr<Graph> graph)
    : pImpl(
          getExecutorMode() ? dynamic_cast<GraphExecutorImplBase*>(
                                  new
ProfilingGraphExecutorImpl(graph))
                            : dynamic_cast<GraphExecutorImplBase*>(
                                  new GraphExecutorImpl(graph))) {}

void GraphExecutor::run(Stack& inputs) {
  return pImpl->run(inputs);
}
```

We can also tell that there are actually two implementations of graph executor: `GraphExecutorImpl` and `ProfilingGraphExecutorImpl`. Both are subclasses of `GraphExecutorImplBase`. `GraphExecutorImpl` is implemented in `graph_executor.cpp`, while `ProfilingGraphExecutorImpl` is implemented in `profiling_graph_executor_impl.cpp`. We will look at both implementations.

Before looking at any implementation, let's first look at the base class, which is also implemented in `graph_executor.cpp`:

```
void GraphExecutorImplBase::run(Stack& stack) {
  TORCH_CHECK(
      stack.size() >= num_inputs,
      "expected ",
      num_inputs,
      " inputs, but got only ",
      stack.size());

  C10_LOG_API_USAGE_ONCE("torch.graph_executor.run");
  logging::getLogger()->addStatValue(
      logging::runtime_counters::GRAPH_EXECUTOR_INVOCATIONS,
1.0);

  ExecutionPlan plan =
      getPlanFor(stack,
GraphExecutor::getDefaultNumBailOuts());
  InterpreterState(plan.code).run(stack);
  last_executed_optimized_graph = plan.graph;
}
```

we can see that it first get an `ExecutionPlan`, create a state machine `InterpreterState`, and run the state machine on the stack.

GraphExecutorImpl

Now let's move on to `GraphExecutorImpl`:

```
ExecutionPlan getPlanFor(Stack& stack, size_t
remaining_bailout_depth)
    override {
  return getGraphExecutorOptimize() ? getOrCompile(stack)
                                    : getOrCompileFallback();
}
```

We can see that the second argument `remaining_bailout_depth` is completely ignored, which indicate that `GraphExecutorImpl` does not have a bailout mechanism.

We also see that the graph is compiled at the first time it runs to get an execution plan. Compilation of graph to execution plan is done by `getOrCompile` or `getOrCompileFallback` depending on if optimization is enabled. These two methods are copied below:

```
const ExecutionPlan& getOrCompileFallback() {
  std::lock_guard<std::mutex> lock(compile_mutex);
  if (!fallback) {
    auto graph_ = graph->copy();
    runRequiredPasses(graph_);
    fallback = ExecutionPlan(graph_);
  }
  return fallback;
}

const ExecutionPlan& getOrCompile(const Stack& stack) {
  // outside lock guard, to minimize the time holding the lock on the
fast
  // path ArgumentSpec even computes its hashCode here.
  ArgumentSpec spec =
      arg_spec_creator_.create(autograd::GradMode::is_enabled(),
stack);
  {
    std::lock_guard<std::mutex> lock(compile_mutex);
    auto it = plan_cache.find(spec);
    if (it != plan_cache.end()) {
      logging::getLogger()->addStatValue(
          logging::runtime_counters::EXECUTION_PLAN_CACHE_HIT, 1.0);
      return it->second;
    }
    auto plan = compileSpec(spec);
    auto r = plan_cache.emplace(std::move(spec), std::move(plan));
    logging::getLogger()->addStatValue(
        logging::runtime_counters::EXECUTION_PLAN_CACHE_MISS, 1.0);
    return r.first->second;
  }
}
```

These code explain itself well: if optimization is turned off, then we only run required passes and cache the result. Otherwise, depending on the characteristic of inputs (`ArgumentSpec`), we run full optimization and cache the generated plan for each different `ArgumentSpec` . The plan is created by the constructor of `ExecutionPlan` .

It worth a look at what passes are called:

```
ExecutionPlan compileSpec(const ArgumentSpec& spec) {
  auto opt_graph = graph->copy();
  SOURCE_DUMP("Optimizing the following function:", opt_graph);
  arg_spec_creator_.specializeTypes(*opt_graph, spec);

  // Phase 0. Inline functions, then clean up any artifacts that the
inliner
  //          left in that may inhibit optimization
  Inline(*opt_graph);
  LowerGradOf(*opt_graph);
  specializeAutogradZero(*opt_graph);
  LowerSimpleTuples(opt_graph);
  ConstantPooling(opt_graph);

  // Phase 1. Specialize to input definedness (this is very important for
  //          gradient graphs), and run required passes to bring the
graph
  //          to an executable form.
  runRequiredPasses(opt_graph);

  // Phase 2. Propagate detailed information about the spec through the
  //          graph (enabled more specializations in later passes).
  //          Shape propagation sometimes depends on certain arguments
being
  //          constants, and constant propagation doesn't need shape
  //          information anyway, so it's better to run it first.
  ConstantPropagation(opt_graph);
  PropagateInputShapes(opt_graph);
  PropagateRequiresGrad(opt_graph);

  // Phase 3. Run differentiable optimizations (i.e. simple graph
rewrites
  //          that we can still execute using autograd).
  runOptimization(opt_graph);

  // Phase 4. If this graph will be differentiated, we need to slice out
the
  //          symbolically differentiable subgraphs for further
optimizations.
  // Phase 5. Apply non-differentiable optimizations to the graphs we've
found
  //          (or the whole graph if we know we won't need its
derivative).
  if (needsGradient(opt_graph)) {
    auto diff_nodes = CreateAutodiffSubgraphs(
        opt_graph,
        autodiff_subgraph_inlining ? autodiffSubgraphNodeThreshold : 1);
    for (Node* dnode : diff_nodes) {
      auto diff_graph = std::move(dnode->g(attr::Subgraph));
      Gradient gradient = differentiate(diff_graph);
      // Run post differentiation optimizations, Autodiff will replace
some
      // parts of graph with new graph, these new graphs usually consists
of
      // control flows and miss shape information on nodes, so we run
shape
      // prop and differentiable optimizations to ensure the graph is
      // optimized
      PropagateInputShapes(gradient.f);
      runOptimization(gradient.f);
      // run non diff optimization on the forward graph
```

```
      runNondiffOptimization(gradient.f);
      packGradient(gradient, dnode);
    }
    InlineAutodiffSubgraphs(
        opt_graph,
        autodiff_subgraph_inlining ? autodiffSubgraphInlineThreshold :
1);
  } else {
    runNondiffOptimization(opt_graph);
  }
  // Make sure there are no leftovers from any passes.
  EliminateDeadCode(opt_graph);
  return ExecutionPlan(opt_graph);
}
```

```
void runNondiffOptimization(std::shared_ptr<Graph>& graph) {
  // decomposition pass, decompose certain ops that will be used in the
  // following passes (like batchmm and jit fusion)
  if (!getProfilingMode()) {
    DecomposeOps(graph);
  }

  // TupleConstruct / TupleUnpack pairs can still be present at this
point
  // and must be removed for fusion.
  LowerSimpleTuples(graph);

  // Rewrite subgraphs with many MMs into expressions that batch them.
  BatchMM(graph);

  // Fuse the dequant - op - quant patterns into quantized ops
  QuantFusion(graph);

  FuseGraph(graph);

  // Run custom passes that different backends can register.
  // This is done last to give internal optimization passes priority.
  for (const auto& pass : getCustomPasses()) {
    pass(graph);
  }
}

void runOptimization(std::shared_ptr<Graph>& graph) {
  // Basic graph preprocessing to eliminate noise.
  EliminateDeadCode(graph);
  EliminateCommonSubexpression(graph);

  PeepholeOptimize(graph);
  ConstantPropagation(graph);
  ConstantPooling(graph);

  // Unroll small loops, and eliminate expressions that are the same at
every
  // iteration.
  UnrollLoops(graph);
  EliminateCommonSubexpression(graph);

  CheckInplace(graph);
}
```

I will not go deep into most of these passes in this note, interested readers can read them at `torch/csrc/jit/passes/`.

ProfilingGraphExecutorImpl

Now let's take a look at the profiling graph executor. We also start from `getPlanFor`:

```
ExecutionPlan ProfilingGraphExecutorImpl::getPlanFor(
    Stack& stack,
    size_t remaining_bailout_depth) {
  std::lock_guard<std::mutex> lock(compile_mutex);
  GRAPH_DEBUG("Running ProfilingGraphExecutorImpl ", this);

  if (optimized_plan_) {
    return *optimized_plan_;
  }

  // simple executor
  if (remaining_bailout_depth == 0) {
    auto copy = graph->copy();
    runProfilingInsensitiveOptimizations(copy);
    GRAPH_DUMP("Optimized SimpleExecutor Graph : ", copy);
    optimized_plan_ = ExecutionPlan(copy);
    return *optimized_plan_;
  }

  // if a profiling graph hasn't been created yet
  if (!pr_) {
    auto copy = graph->copy();
    runProfilingInsensitiveOptimizations(copy);
    pr_ = ProfilingRecord::instrumentGraph(copy);
    auto pr_copy = pr_->graph()->copy();
    GRAPH_DUMP("Profiled Graph: ", pr_copy);
    profiling_plan_ = ExecutionPlan(pr_copy);
    // fall-through
  }

  // profile until a graph is ready
  if (!pr_->ready()) {
    return *profiling_plan_;
  }

  auto copy = pr_->graph()->copy();
  runProfilingOptimizations(copy);
  // cache
  optimized_plan_ = ExecutionPlan(copy,
remaining_bailout_depth);
  return *optimized_plan_;
}
```

We can see that there is a “bailout” mechanism with limited depth, if the depth is reached, then the executor just do “profiling insentive optimizations”. But what is “bailout”, what is “profiling”, and what is “profiling insentive optimizations”? It is still not clear at this point. We can also see that `remaining_bailout_depth` is passed to the constructor of `ExecutionPlan`, so the bailout mechanism must be a collaboration of interpreter and graph executor. The optimizations are:

```
void ProfilingGraphExecutorImpl::runProfilingOptimizations(
    std::shared_ptr<Graph>& copy) {
  if (!getGraphExecutorOptimize()) {
    LowerGradOf(*copy);
    runRequiredPasses(copy);
    return;
  }

  InsertGuards(copy);
  LowerGradOf(*copy);
  EliminateRedundantGuards(copy);
  InsertBailOuts(copy);
  GRAPH_DUMP("After InsertBailOuts: ", copy);
  specializeAutogradZero(*copy);

  runRequiredPasses(copy);
  ConstantPropagation(copy);
  runOptimization(copy);

  if (needsGradientInProfilingMode(copy->block())) {
    auto diff_nodes = CreateAutodiffSubgraphs(
        copy,
        getAutodiffSubgraphInlining() ? autodiffSubgraphNodeThreshold :
1);
    for (Node* dnode : diff_nodes) {
      auto diff_graph = std::move(dnode->g(attr::Subgraph));
      Gradient gradient = differentiate(diff_graph);
      runOptimization(gradient.f);
      // run non diff optimization on the forward graph
      runNondiffOptimization(gradient.f);
      packGradient(gradient, dnode);
    }
    InlineAutodiffSubgraphs(
        copy,
        getAutodiffSubgraphInlining() ? autodiffSubgraphInlineThreshold
: 1);

  } else {
    runNondiffOptimization(copy);
  }
  EliminateDeadCode(copy);
  GRAPH_DUMP("Optimized Graph : ", copy);
}

void ProfilingGraphExecutorImpl::runProfilingInsensitiveOptimizations(
    std::shared_ptr<Graph>& copy) {
  LowerGradOf(*copy);
  GRAPH_DUMP("runProfilingInsensitiveOptimizations", copy);
  // clear any residual undefinedness
  // as double backward graph inputs'
  // may carry over undefinedness
  // from profiled backward graphs
  ClearUndefinedness(copy);
  runRequiredPasses(copy);
  if (!getGraphExecutorOptimize()) {
    return;
  }

  DecomposeOps(copy);
  ConstantPropagation(copy);
  EliminateDeadCode(copy);
```

```
  EliminateCommonSubexpression(copy);
  ConstantPooling(copy);
  PeepholeOptimize(copy);
  EliminateDeadCode(copy);
  CheckInplace(copy);
}
```

We will postpone the reading of profiling and bailout after we have read how the interpreter works.

PyTorch IR –> Interpreter Instructions

Now it's time to look at `ExecutionPlan` defined at `graph_executor.h`:

```
struct ExecutionPlan {
  ExecutionPlan() = default;
  ExecutionPlan(
      std::shared_ptr<Graph> graph,
      size_t remaining_bailout_depth = 0)
      : code(graph, remaining_bailout_depth),
graph(std::move(graph)) {}

  operator bool() const {
    return static_cast<bool>(graph);
  }

  Code code;
  std::shared_ptr<Graph> graph;
};
```

It just convert the graph into an object of `Code`, and the running is done by `InterpreterState`.

`Code` and `InterpreterState` are defined in `torch/csrc/jit/interpreter.{h,cpp}`. These two classes are just a wrapper of its implementations:

```
Code::Code(const std::shared_ptr<Graph>& graph, size_t
remaining_bailout_depth)
    : pImpl(new CodeImpl(graph, remaining_bailout_depth)) {}
```

```
InterpreterState::InterpreterState(const Code& code)
    : pImpl(c10::make_intrusive<InterpreterStateImpl>
(code)) {}
```

`CodeImpl` is a long struct, but quite logical. A selected list of fields it has is listed below:

```
PreprocessGraph preprocess;
std::vector<Instruction>
instructions;
```

Its constructor is:

```
CodeImpl(const std::shared_ptr<Graph>& graph, size_t
remaining_bailout_depth)
    : preprocess_(*graph),
      current_node_(preprocess_.graph->return_node()),
      remaining_bailout_depth_(remaining_bailout_depth) {
  graph_ = preprocess_.graph;
  n_outputs = graph_->outputs().size();
  if (n_outputs == 1) {
    return_type_ = graph->outputs().at(0)->type();
  } else {
    return_type_ = TupleType::create(
        fmap(graph->outputs(), [](const Value* v) { return v->type();
}));
  }
  n_inputs = graph_->inputs().size();
  // std::cout << *graph_ << "\n";
  emitCodeForBlock(graph_->block());
  insertInstruction(RET);
  // we deferred the emission of bailout blocks so they appear at the
end
  // emit them now and patch up the jumps
  insertBailoutBlocks();
}
```

Clearly we can see what it does is:

1. preprocess the graph, and then
2. emit instructions for interpreter.
3. insert bailout blocks

The preprocessing of graph is very well explained in the beginning of file:

```
// Before we translate to intepreter instructions, we do
// some preprocessing of the graph to turn it into a form that is closer
// to what the instructions will look like.
// In particular we:
// *  Computes whether a input to a node is the last use, so we can issue
MOVE
//    rather than LOAD instructions.
// *  Drop nodes are inserted for any node that is unused to create a
dummy use
//    that will cause the interpreter to free the node.
//    A drop node just pops its input off the stack to  ensure the
interpreter
//    releases references to nodes that are never used. Drop nodes are
also
//    inserted when the last use of a node is in some conditionally run
control
//    flow (e.g. one side of an If) and the interpreter must free the node
only
//    after the control flow has reconverged
// Outputs are:
// * graph - the post processed copy of g
// * move_flags[n] - a list of booleans, one for each input,
//   indicating whether this is the last use of the value. The interpreter
//   should generate a move rather than a copy in this case.
```

The `emitCodeForBlock` emits instructions:

```
void emitCodeForBlock(Block* block) {
  emitNodeAtBlockLevel(block-
>param_node());
  for (auto node : block->nodes()) {
    emitNodeAtBlockLevel(node);
  }
  emitNodeAtBlockLevel(block-
>return_node());
}
```

Since the nodes are topologically sorted, we just need to iterate the linked list and generate code for each node.

The `emitNodeAtBlockLevel` is:

```
void emitNodeAtBlockLevel(Node* node) {
  WithCurrentNode guard(&current_node_,
node);
  switch (node->kind()) {
    case prim::Constant:
      emitConstant(node);
      break;
    case prim::Return:
      emitLoadInputs(node->inputs());
      break;
    default:
      if
(!preprocess_.can_emit_inline[node]) {
        emitNode(node);
        emitStoreOutputs(node);
      }
      break;
  }
}
```

where `emitNode` dispatches according to node kind:

```
void emitNode(Node* node) {
  WithCurrentNode guard(&current_node_, node);
  switch (node->kind()) {
    default:
      emitOperator(node);
      break;
    case prim::Drop:
      emitDrop(node->inputs());
      break;
    case prim::Constant:
      emitConstant(node);
      break;
    case prim::If:
      emitIf(node);
      break;
    case prim::Loop:
      emitLoop(node);
      break;
    case aten::wait:
      emitWait(node);
      break;
    case prim::Param:
      break;
    case prim::CallFunction:
      emitCall(
          node->inputs().at(0)->type()->expect<FunctionType>()-
>function(),
          node->inputs().slice(1));
      break;
    case prim::CallMethod:
      if (auto class_type = node->inputs().at(0)->type()->cast<ClassType>
()) {
        emitCall(class_type->getMethod(node->s(attr::name)), node-
>inputs());
      } else {
        emitInterfaceCall(node->s(attr::name), node->inputs());
      }
      break;
    case prim::BailOut:
      emitBailOut(node);
      break;
    case prim::GetAttr:
      emitGetAttr(node);
      break;
    case prim::SetAttr:
      emitSetAttr(node);
      break;
  }
}
```

Let's further take a look at `emitOperator`, `emitIf` and `emitBailOut` as example.

```
void emitOperator(Node* node) {
  emitLoadInputs(node->inputs());
  insertInstruction(OP,
operator_table_.size());

operator_table_.emplace_back(getOperation(node
));
}
```

```
void emitIf(Node* node) {
  emitLoadInputs(node->inputs());
  size_t start_if = instructions_.size();
  insertInstruction(JF, 0); // dummy offset to be filled in
  emitCodeForBlock(node->blocks().at(0));
  insertInstruction(JMP, 0); // dummy offset
  size_t start_else = instructions_.size();
  instructions_[start_if].X = start_else - start_if;
  emitCodeForBlock(node->blocks().at(1));
  instructions_[start_else - 1].X = instructions_.size() - (start_else
- 1);
}
```

these two are pretty standard compiler implementations.

Now let's take a look at how bailout works:

```
void emitBailOut(Node* node) {
  auto jf_index = emitGuard(node);
  auto unoptimized_graph = node->inputs().at(0)->node()-
>g(attr::Subgraph);
  // note, guaded input is already loaded onto the stack
  // for GUARD instruction
  emitLoadInputs(node->inputs().slice(2));
  insertInstruction(TAIL_CALL, function_table_.size());
  TORCH_INTERNAL_ASSERT(node->kind() == prim::BailOut);
  auto bailout_index = node->i(attr::index);
  TORCH_INTERNAL_ASSERT(bailout_index >= 0);

  auto build_bailout_graph = [bailout_index,
                              unoptimized_graph](Function &func) {

    BuildBailOutGraphFrom(bailout_index, unoptimized_graph,
func.graph());
  };

  auto empty_graph = std::make_shared<Graph>();
  auto func = torch::make_unique<Function>(
      "bailout", empty_graph, build_bailout_graph);
  function_table_.emplace_back(func.get());
  bailout_functions_.emplace_back(std::move(func));
  createBailoutBlock(jf_index);
}
```

The `emitBailOut` seems to save an unoptimized graph and make a function out of it. We will not go into details about this right now, because this requires a big picture of how the profiling is done in the executor. For now we just want a brief organization of the code generally. But we will do a thorough read of this in a separate section. We will move to the virtual machine right now.

The Virtual Machine

`InterpreterStateImpl` is the virtual machine that executes instructions. The related functions are located here:

```
void run(Stack& stack) {
  if (runImpl(stack)) {
    future_->wait();

    auto num_outputs = frames.front().function-
>n_outputs;
    if (num_outputs == 1) {
      push(stack, future_->value());
    } else {
      auto tuple = future_->value().toTuple();
      for (const IValue& value : tuple->elements()) {
        push(stack, value);
      }
    }
  }
}
```

which invoke `runImpl` to run asynchronously and wait for the async run to finish(therefore sync in effect). The `runImpl` looks like:

```
bool runImpl(Stack& stack) {
  // if we have never run before, then we might have to return the
  // stack when we suspend, record where it starts so we return the
right
  // stack
  if (stack_start_ == -1) {
    TORCH_INTERNAL_ASSERT(stack.size() >= frames.back().function-
>n_inputs);
    stack_start_ = stack.size() - frames.back().function->n_inputs;
  } else {
    // during restarts, all of the stack is always our own, so we leave
    // nothing
    stack_start_ = 0;
  }

  ActiveFrame af(frames.back());
  try {
    while (true) {
//         std::cout << "RUNNING ";
//         frames.back().function->dump(std::cout, af.pc);
      Instruction inst = af.instructions[af.pc];
      switch (inst.op) {
        case OP:
          af.operators[inst.X](stack);
          ++af.pc;
          break;
        case OPN:
          AT_ERROR("OPN is currently supported in mobile mode only.");
          break;
        case LOAD:
          stack.emplace_back(reg(inst.X));
          ++af.pc;
          break;
        case MOVE:
          stack.emplace_back(std::move(reg(inst.X)));
          ++af.pc;
          break;
        case STORE: ...
        case STOREN: ...
        case DROP: ...
        case DROPR: ...
        case LOADC: ...
        case GET_ATTR: ...
        case SET_ATTR: ...
        case JF: ...
        case JMP: ...
        case LOOP: ...
        case CALL:
        case INTERFACE_CALL: ...
        case RET: ...
        case WAIT: ...
        case FAIL_GUARD: ...
        case GUARD: ...
        case TAIL_CALL: ...
      }
    }
  } catch (std::exception& e) {
    frames.back().pc = af.pc;
    bool is_jit_exception = dynamic_cast<JITException*>(&e);
    handleError(ExceptionMessage(e), is_jit_exception);
    return false;
```

```
    }
}
```

There is nothing special, just mimicking the behavior of processors.

Profiling and bailout

A good starting point for profiling and bailout is the official design doc `torch/csrc/jit/docs/OVERVIEW.md`:

```
# Profiling Programs

`prim::profile` nodes are inserted on every **use** of a value by
`ProfilingRecord::instrumentBlock`. Every `prim::profile` node runs a lambda that
uses a captured, initial type value and the type of an incoming tensor and merges
the two into a refined `TensorType`

`prim::profile` nodes are replaced with `prim::Guard` nodes by `InsertGuards`.
`prim::Guard` nodes are inserted to guarantee that beyond the guard a guarded
tensor will always be of the profiled shape. This guarantee will enable
optimizations and codegens to generate more efficient code.

JIT attempts to reduce the number of `prim::Guard` nodes as these nodes may
interefere with optimizations.
* First, `GuardElimination::moveGuardsToDefs` tries to move `prim::Guards` to
their definitions, so the guards guarding the same tensor follow the definition
directly or another guard on the same tensor. This step is done in
* This ordering allows us to **coalesce** (done in
`GuardElimination::coalesceGuards`) multiple guards into a single one.
* After guards are  **coaslesced** , `GuardElimination::eliminateGuards` attempts
to eliminate more guards as follows: it inspects each operation and its inputs.
It checks if inputs to the operation are guarded and also if the operation
produces the consistent shapes given the guarded inputs. For example, if two
inputs to `add` are guaranteed to be of shape `(2, 3) `, the output shape will
also always be `(2, 3)` If this property holds, JIT is allowed to remove the
guard guarding operation's output.

Lastly, JIT needs to be handle cases when the assumptions about tensor shapes
fail at runtime. To handle guard failures, JIT needs to be able to run the
original code i.e. the code  that doesn't rely on assumptions about shapes. As
guards can be inserted and moved (by Optimizer) at/to arbitrary points in a
computional graph, JIT needs to be able to resume execution starting from those
arbitrary points onward.

`InsertBailoutNodes` builds deoptimized versions of the original computational
graph, that contain the rest of computations starting from their corresponding
guard failure poins and, also, captures live values needed to execute those
deoptimized graphs. In other words, the pass replaces `prim::Guard` nodes with
`prim::BailOut` nodes which have the`attr::Subgraph` attributes set to the
deoptimized versions of the  remaining computations at their corresponding
`prim::Guard`s.
```

After getting the rough idea, let's look at the code. In `ProfilingGraphExecutorImpl::getPlanFor`, which runs everytime when we execute a graph, we have:

```
// if a profiling graph hasn't been
created yet
if (!pr_) {
  auto copy = graph->copy();

runProfilingInsensitiveOptimizations(copy)
;
  pr_ =
ProfilingRecord::instrumentGraph(copy);
  auto pr_copy = pr_->graph()->copy();
  GRAPH_DUMP("Profiled Graph: ", pr_copy);
  profiling_plan_ =
ExecutionPlan(pr_copy);
  // fall-through
}

// profile until a graph is ready
if (!pr_->ready()) {
  return *profiling_plan_;
}
```

From this, we can see that for the first few runs the graph executor instrument graph with profiling nodes and execute until no more profiling is needed.
Let go deep to see how the graph is instrumented. `ProfilingRecord::instrumentGraph` is defined in `torch/csrc/jit/profiling_record.cpp` :

```
std::unique_ptr<ProfilingRecord> ProfilingRecord::instrumentGraph(
    const std::shared_ptr<Graph>& graph) {
  auto new_g = graph->copy();
  auto pr = std::unique_ptr<ProfilingRecord>(new
ProfilingRecord(new_g));
  auto raw_pr = pr.get();
  unprofileGraphInputs(new_g);
  unprofileBlock(new_g->block());
  pr->instrumentBlock(new_g->block());

  for (auto i : new_g->return_node()->inputs()) {
    if (i->type()->isSubtypeOf(TensorType::get())) {
      pr->insertShapeProfile(new_g->return_node(), i);
    }
  }
  std::function<void(Stack&)> counter = [raw_pr](Stack&) {
    std::lock_guard<std::mutex> lock(raw_pr->mutex_);
    if (raw_pr->profiling_count_ > 0)
    {
        raw_pr->profiling_count_--;
    }
  };

  auto pop = pr->createProfileNode(counter, {});
  new_g->appendNode(pop);
  return pr;
}
```

Combining the knowledge from the docs, we can see that this function calls `instrumentBlock` to instruct the main block of the graph. Let's now move on to `instrumentBlock`:

```
void ProfilingRecord::instrumentBlock(Block *block) {
  for (auto it = block->nodes().begin(); it != block->nodes().end();
++it) {
    auto n = *it;
    for (auto i : n->inputs()) {
      if (!i->type()->isSubtypeOf(TensorType::get()) ||
          i->node()->kind() == prim::profile) {
        continue;
      }

      insertShapeProfile(n, i);
    }

    for (auto b : n->blocks()) {
      instrumentBlock(b);
    }
  }
}
```

we can see that it iterate the whole graph, and for every node that has tensor input, we do `insertShapeProfile`. And we also recursively instrument blocks. The `ProfilingRecord` looks like:

```
void ProfilingRecord::insertShapeProfile(Node *n, Value *i) {

  auto pn = createProfileNode(nullptr, {i});
  auto pno = pn->addOutput();
  bool first = true;
  pno->setType(TensorType::get());
  std::function<void(Stack &)> shape_profiler = [this, pno,
                                                 first](Stack &stack)
mutable {
    IValue t;
    pop(stack, t);
    if (t.isTensor()) {

      if (t.toTensor().defined()) {
        auto pttp = tensorTypeInCurrentExecutionContext(t.toTensor());
        std::lock_guard<std::mutex> lock(this->mutex_);
        if (auto type = pno->type()->cast<TensorType>()) {
          if (!first) {
            pttp = pttp->merge(type);
          }
          pno->setType(pttp);
          first = false;
        }
      } else {
        pno->setType(TensorType::get()->withUndefined());
      }
    }

    // passing t through
    push(stack, t);

  };

  pn->setCallback(shape_profiler);
  pn->insertBefore(n);
  n->replaceInputWith(i, pn->output());
}
```

We can see that it inserts a profile node before, and the call back of the proflie node set the the lambda `shape_profiler`. We will stop going deeper on `ProfilingRecord` at here since we already have a rough idea on what it does.

Now let's move to `InsertGuards` which is called in `ProfilingGraphExecutorImpl::runProfilingOptimizations`, which is called in `ProfilingGraphExecutorImpl::getPlanFor`. The `InsertGuards` is defined at `torch/csrc/jit/passes/insert_guards.cpp`:

```
struct GuardInserter {
  GuardInserter(std::shared_ptr<Graph> graph) :
graph_(std::move(graph)) {}

  void run() {
    insertGuards(graph_->block());
    removeProfilingNodes(graph_->block());
  }

 private:
  void removeProfilingNodes(Block* b) {
    for (auto it = b->nodes().begin(); it != b->nodes().end(); it++) {
      if (it->kind() == prim::profile) {
        it.destroyCurrent();
      } else {
        for (Block* ib : it->blocks()) {
          removeProfilingNodes(ib);
        }
      }
    }
  }

  void insertGuards(Block* b) {
    for (auto it = b->nodes().begin(); it != b->nodes().end(); it++) {
      auto n = *it;
      if (n->kind() == prim::profile && n->outputs().size() == 1) {
        auto pttp = n->output()->type()->cast<TensorType>();
        if (pttp) {
          auto guard = graph_->create(prim::Guard, {n->input()}, 1);
          auto go = guard->output();
          go->setType(pttp);
          guard->insertBefore(n);
          n->output()->replaceAllUsesWith(go);
        } else {
          // we didn't go down this path i.e
          // no profiling information is available
          n->output()->replaceAllUsesWith(n->input());
        }
        it.destroyCurrent();
      } else {
        for (Block* ib : n->blocks()) {
          insertGuards(ib);
        }
      }
    }
  }

  std::shared_ptr<Graph> graph_;
};

void InsertGuards(std::shared_ptr<Graph> graph) {
  GuardInserter gi(std::move(graph));
  gi.run();
}
```

It is just a standard recursive graph traverse that replaces insert the guard before profiling node and then remove all profiling nodes.

We will skip the part on how the guard is moved for optimization. Let's move on how the guard is converted to bailouts. From the document, we know that is is done at `InsertBailoutNodes`, but after some searching, we can not find `InsertBailoutNodes`. The docs must already been outdated. But after looking around, we see in `ProfilingGraphExecutorImpl::runProfilingOptimizations`, we have

```
InsertGuards(copy);
LowerGradOf(*copy);
EliminateRedundantGuards(copy);
InsertBailOuts(copy);
GRAPH_DUMP("After InsertBailOuts: ",
copy);
```

So it must be `InsertBailOuts` actually doing the job. This function is defined at `torch/csrc/jit/passes/bailout_graph.cpp`:

```
void InsertBailOuts(std::shared_ptr<Graph>
graph) {
  BailOutInserter ibo(std::move(graph));
  ibo.run();
}
```

```
// `BailOutInserter` replaces prim::Guard nodes
with
// prim::BailOut nodes that allow interpreter to
// resume execution of the unoptimized(deoptimized)
// version of an original graph from a particular
point
struct BailOutInserter {
  explicit BailOutInserter(std::shared_ptr<Graph>
graph)
      : graph_(std::move(graph)), bailout_index_(0)
{}

  void run() {
    liveness_sets_ = BuildLivenessSets(graph_);
    insertBailOuts(graph_->block());
    replaceGuardsWithBailouts();
    // embed a full original graph
    addUnoptimizedFuncToBailouts();
  }
......
```

We see a term `liveness`, this is an analysis in compilers, and it is explained in Wikipedia: Live variable analysis. We won't go deep into it, all we need to know is its definition:

```
A variable is live at some point if it holds a value that may be needed in the
future, or equivalently if its value may be read before the next time the
variable is written to.
```

Now, let's look at `insertBailOuts`:

```
// Inserts prim::BailOut nodes for every prim::Guard
// Each BailOut point takes the set of inputs live
// at that particular execution point.
// An input is live if it's used beyond the guard/BailOut
// point to compute graph's outputs
void insertBailOuts(Block* b) {
  for (auto it = b->nodes().begin(); it != b->nodes().end(); ++it) {
    if (it->kind() == prim::Guard) {
      auto bailout_node = b->owningGraph()->create(prim::BailOut);
      bailouts_.push_back(bailout_node);

      const auto& live_inputs = liveness_sets_[*it];

      // guarded inputs come first
      // currently, there's always one guarded input
      bailout_node->addInput(it->input());
      for (auto li : live_inputs) {
        // Guarded inputs have already been added
        // Also, skip some inputs that BailOutGraphBuilder can
        // materialize into bailout graphs directly
        if (!shouldBeCapturedInByBailOut(li->node()) || li == it-
>input()) {
          continue;
        }
        bailout_node->addInput(li);
      }

      bailout_node->output()->setType(it->output()->type());
      bailout_node->i_(attr::index, bailout_index_++);
      // we can't immediately replace nodes since this action will
corrupt
      // the liveness sets of following BailOut nodes if any of their
      // arguments are BailOut nodes themselves
      replacements_.insert({it->output(), bailout_node->output()});

    } else {
      for (auto ib : it->blocks()) {
        insertBailOuts(ib);
      }
    }
  }
}
```

We can see that it recursively traverse the graph, at each `prim::Guard` node, it creates a new node of kind `prim::BailOut` and set its input to live input at current point. The newly created node is not inserted into the graph, but stored at `replacements_` instead.

Then in `replaceGuardsWithBailouts`:

```
// replace each prim::Guard
// with its corresponding prim::BailOut
void replaceGuardsWithBailouts() {
  for (auto e : replacements_) {
    e.first->replaceAllUsesWith(e.second);
    e.second->node()->insertAfter(e.first-
>node());
    e.first->node()->destroy();
  }
}
```

Knowing that how the guards are converted to bailout nodes, the last step is to see how the bailout node is executed. From previous reading, we know that instructions for different kind of nodes are emitted in the constructor of `CodeImpl`, which calls `emitCodeForBlock`, which calls `emitNodeAtBlockLevel`, which calls `emitNode` which dispatch on node kind and calls `emitBailOut` for bailout nodes.

Now let's go back to see `emitBailOut` again:

```
void emitBailOut(Node* node) {
  auto jf_index = emitGuard(node);
  auto unoptimized_graph = node->inputs().at(0)->node()-
>g(attr::Subgraph);
  // note, guaded input is already loaded onto the stack
  // for GUARD instruction
  emitLoadInputs(node->inputs().slice(2));
  insertInstruction(TAIL_CALL, function_table_.size());
  TORCH_INTERNAL_ASSERT(node->kind() == prim::BailOut);
  auto bailout_index = node->i(attr::index);
  TORCH_INTERNAL_ASSERT(bailout_index >= 0);

  auto build_bailout_graph = [bailout_index,
                              unoptimized_graph](Function &func) {

    BuildBailOutGraphFrom(bailout_index, unoptimized_graph,
func.graph());
  };

  auto empty_graph = std::make_shared<Graph>();
  auto func = torch::make_unique<Function>(
      "bailout", empty_graph, build_bailout_graph);
  function_table_.emplace_back(func.get());
  bailout_functions_.emplace_back(std::move(func));
  createBailoutBlock(jf_index);
}
```

We can see that inside `emitBailOut`, it first `emitGuard` then emit a `TAIL_CALL` instruction. The `emitGuard` is:

```
size_t emitGuard(Node* node) {
  // unoptimized graph is at index 0
  // guarded input is at index 1
  // the rest of args follow
  emitLoadInputs(node->inputs().slice(1, 1));
  insertInstruction(GUARD, type_table_.size());

  type_table_.emplace_back(node->outputs().at(0)-
>type());
  insertInstruction(JF, 0 /* to be patched */);

  return instructions_.size() - 1;
}
```

we can see that it emits a `GUARD` instruction and a `JF` instruction. The definition of how each instruction is executed is defined in `InterpreterStateImpl::runImpl`:

```
case JF:
  af.pc += (pop(stack).toBool()) ? 1 : inst.X;
  break;
.......
case GUARD: {
  auto t = stack.back().toTensor();
  const TypePtr& expected = af.types[inst.X];
  bool comp = expected->cast<TensorType>()
                  -
>isCompatibleWithInCurrentExecutionContext(t);
  push(stack, comp);
  ++af.pc;
} break;
case TAIL_CALL: {
  GRAPH_DEBUG("running TAIL_CALL for ", inst.X);
  af.functions[inst.X]->ensure_defined();
  size_t remaining_bailout_depth =
      frames.back().function->remaining_bailout_depth_ > 0
      ? frames.back().function->remaining_bailout_depth_ - 1
      : 0;
  const Code& code = af.functions[inst.X]
                          ->get_executor()
                          .getPlanFor(stack,
remaining_bailout_depth)
                          .code;
  size_t num_inputs = code.num_inputs();
  size_t base_pointer = frames.back().base_pointer;
  TORCH_INTERNAL_ASSERT(stack.size() >= num_inputs);
  size_t inputs_start = stack.size() - num_inputs;
  for (size_t i = 0; i < num_inputs; ++i) {
    stack.at(base_pointer + i) =
        std::move(stack.at(inputs_start + i));
  }
  stack.resize(base_pointer + num_inputs);
  leaveFrame();
  enterFrame(code, base_pointer);
  af = ActiveFrame(frames.back());
} break;
```

We will not worry about how the jump address is computed, but we can see that the big picture is: check if the tensor on the stack is compatible, if not, then call the backup unoptimized graph.