

Kotlin Compiler Reading Notes

Yuxiang Jiang *

March 21, 2020

Contents

1	Methodology	2
2	Tl;dr: Pipeline	3
2.1	Source \rightarrow KtElement	3
2.2	KtElement \rightarrow DeclarationDescriptor	3
2.2.1	LazyTopDownAnalyzer	3
2.3	DeclarationDescriptor \rightarrow FIR	4
2.4	FIR \rightarrow IR	4
2.5	Example stacktrace for running a Kotlin script	4
3	KtElement	5
4	DeclarationDescriptor	5
4.1	Survey of Class Hierarchy	5
4.1.1	CallableD: VisD & NonRootD & Subst	5
4.1.2	MemberD: VisD & NonRootD	5
4.1.3	CallableMemberD: CallableD & MemberD	5
4.1.4	ValueD: CallableD	5
4.1.5	VarD: ValueD	5
4.1.6	ParamD: ValueD	6
4.1.7	VarDWithAccessors	6
4.1.8	FunD: CallableMemberD	6
4.1.9	FunDImpl: NonRootDImpl & FunD	6
4.1.10	ConD: FunD	6
4.1.11	ClassConD: ConD	6
4.1.12	ClassConDImpl: FunDImpl & ClassConD	6
4.2	DFactory	7
5	FIR	7

*yxjiang@linkedin.com

6	IR	7
6.1	compiler.ir.tree	7
7	Type System	8
7.1	Types	8
7.2	core.type-system	8
7.3	TypeSystemTypeFactoryContext	8
7.4	TypeCheckerProviderContext	8
7.5	TypeSystemCommonSuperTypesContext	8
7.6	TypeSystemInferenceExtensionContext	8
7.6.1	Questions	9
7.7	TypeSystemContext	9
7.8	Type System for Fir	9
7.9	Type System for Ir	10
7.10	compiler.resolution/.inference	10
7.11	compiler.frontend/.types	10
7.12	types.expressions	10
8	Resolution	10
8.1	Scopes	10
8.2	Tower	10
9	Analysis	11
9.1	compiler.frontend.LazyTopDownAnalyzer	11
9.1.1	TopDownAnalysisContext	11
9.2	BindingTrace	11
9.3	BindingContext	11
9.4	Smartcasting	11
10	Codegen	11
10.1	ExpressionCodeGen	11
10.2	StackValue	11
10.3	ClassBodyCodeGen	11
10.4	FunctionCodeGen	12
10.5	JS Codegen	12
11	Unsolved Questions	12
	Nomenclature	13

1 Methodology

It's often daunting to read through huge and complex codebase like the Kotlin compiler! Fortunately we have great tools at hand to deal with such complexity. In particular, IntelliJ Idea provides many priceless code navigation tools:

- Type Hierarchy (under Navigate)
- Structure (under View - Tool Windows)
- Find Usage
- Breakpoints

With these we'll be able to understand the compiler architecture and internals bit by bit...

2 Tl;dr: Pipeline

- Kotlin source
- KtElement (Psi)
- core.descriptor (ClassicTypeSystemContext)
- FirElement (nodes) / FirBasedSymbol (infotable?) (ConeTypeContext)
- IrElement (nodes) / IrSymbol (infotable?) (IrTypeSystemContext)

2.1 Source → KtElement

- See class KtVisitor for an overview of many Kotlin PsiElements
- compiler.psi defines KtElement and Stubs, also provides parser
 - Stubs represents the interface parts of a kt compilation unit (.h, .mli, .hi etc).
 - There's also a LighterASTNode (KotlinLightParser) – what is it? (Flyweight pattern is like hash consing)
 - Some KtElements are related to types: e.g. KtTypeReference (some of them are not KtElements but stubs!)
- compiler.psi.KtPsiFactory is the entrypoint for creating KtFile (a PsiElement) from source text.

2.2 KtElement → DeclarationDescriptor

DeclarationDescriptor seems to be a high-level IR with type analyzed. It contains both the expr tree and the types.

2.2.1 LazyTopDownAnalyzer

compiler.frontend.LazyTopDownAnalyzer seems to be the psi analyzer. (Is there another analyzer that's not lazy?)

analyzeDeclarations returns an AnalysisResult which contains a ModuleDescriptor and a BindingContext.

2.3 DeclarationDescriptor → FIR

FIR (compiler.fir) seems to be an intermediate yet still high-level IR.

- See generated class FirVisitor for an overview of many of the Fir exprs
- compiler.frontend (not sure which step it is in, but it at least does symbol resolution, type checking, (Psi → CFG?)
 - See key classes: AnalysisResult, BindingContext, BindingTrace (records the collected binding / type substs?)
- compiler.resolution: tower/ReceiverValue etc
- compiler.fir: cones (types and symbols used in Fir?), fir2ir (lowering to Ir), psi2fir (lowering Psi to Fir), resolve, jvm, tree (Fir definitions and impl for psi2fir)
 - cones: StandardClassIds contains a bunch of core Kt (read: not JVM) type Ids. They have a JVM-like ffname.
SyntheticCallableId contains when/try/nullcheck synthetic call exprs
 - tree.gen contains all Fir expressions (see tree.tree-generator's readme for how they are generated), as well as extra info (FirTypeRef). And even on Fir level, the generic types are not yet erased (FirTypeProjectionWithVariance)

2.4 FIR → IR

IR (compiler.ir) seems to be an lower-level IR. This is an experimental IR that's intended to be used across all Kotlin backends. Also see this writeup.

A sample IR lowering pass can be found here. Looks that the compiler API provides a convenient IrBuilder, and a visitor style IrTransformer.

2.5 Example stacktrace for running a Kotlin script

E.g. kotlin -e <expr>

- CLIDriver
 - compiler.cli / K2JVMCompiler
 - plugins.scripting-compiler
 - compiler.cli / TopDownAnalyzerFacadeForJVM (analyzeFilesWithJavaIntegration)
- Analyzer
 - compiler.frontend / LazyTopDownAnalyzer.analyzeDeclarations
 - (HUGE) go through all stmts
 - BodyResolver.resolveBodies
 - DeclarationChecker.process

- (HUGE) go through files, annotations, class's modifiers, idents, header (super+generic bounds); function, property, destructionDecl, typealias's modifiers and idents.

3 KtElement

KtFile and KtScript for toplevel decls, KtClass for `class Foo`, KtNamedFunction for `fun foo()`, KtProperty for `val foo`.

4 DeclarationDescriptor

See `core.descriptors.DeclarationDescriptorVisitor` for an overview.

`DeclarationDescriptorVisitor` sounds like an type-instantiated/abstracted wrapper of an element. Also has a bunch of annotations (`AnnotationDescriptor`) and a name. And is also a tree node (has parent: `getContainingDecl`)

4.1 Survey of Class Hierarchy

Looks that descriptors are something that's used throughout the whole compilation pipeline. They are more often used in frontend, but even in backend I can see some usage of them.

4.1.1 CallableD: VisD & NonRootD & Subst

Receiver types (dispatch / extension), arg types, return types, type params;
 Parameter names, names may be unstable/synthesized (e.g. from JVM object code)
 Parameter values (See 4.1.6)
 Cross ref to overridden methods
`UserDataKey<A>`: stores typed user data

4.1.2 MemberD: VisD & NonRootD

Has member modifiers: `expect` / `actual` / `external`. And modality: `final` / `sealed` / `open` / `abstract`.

4.1.3 CallableMemberD: CallableD & MemberD

Kind: `decl` / `delegation` / `fakeOverride` / `synthesized` (what's the last two?)

4.1.4 ValueD: CallableD

Has a `KotlinType`.

4.1.5 VarD: ValueD

Has `isVar` (wat), `isLateinit`, `isConst`, and an optional `compileTimeInitializer`.

4.1.6 ParamD: ValueD

Represents a parameter that can be supplied to a callableD.

ReceiverParamD: ParamD Has a ReceiverValue.

ValueParamD: ParamD & VarD Has a index, hasDefaultValue, varargElementType, isCrossinline / Noinline (why on param?)

4.1.7 VarDWithAccessors

Has optional getter / setter typed VarAccessorD.

4.1.8 FunD: CallableMemberD

initialSignatureD: the initial D before renaming (didn't find SimpleFunctionD.rename)

hiddenToOvercomeSignatureClash: hack to handle corner case signature clash (said see nio.CharBuffer); also hiddenEverywhereBesideSupercalls: see 4.1.9.

Function modifiers: infix/inline/operator/suspend/tailrec

4.1.9 FunDImpl: NonRootDImpl & FunD

Base impl for function modifiers. Setters set the local modifier (mostly happen during conversion from KtElement), while some getters (infix, operator) respect super class methods.

Base impl for substitution (doSubstitute), and substituted value param. Worth reading.

Base impl for initialize.

Only here documents hiddenToOvercomeSignatureClash and hiddenEverywhereBesideSupercalls: former makes the function completely hidden (even in super-call), latter permits super-call and propagates to overridden methods

4.1.10 ConD: FunD

containingD: ClassifierDWithTypeParams (what is this?)

constructedClass: ClassD

4.1.11 ClassConD: ConD

Just a bunch of return type specializations

4.1.12 ClassConDImpl: FunDImpl & ClassConD

Default (<init>) or synthesized.

Has a way to calculate dispatchReceiverParam. If inner, init's receiver is outer class instance (Whereas Java's outer class instance is passed as a param. Though in compiled code there's no difference, just at descriptor level they are different.); else null.

CommonizedClassConD: ClassConDImpl ClassConDImpl with source and originalD stripped.

DefaultClassConD: ClassConDImpl Read as (default constructor) of given class, not default implementation of (any class constructor). So this is just the no-param constructor of a class. Its visibility depends on the classD's visibility.

DeserClassConD: ClassConDImpl & DeserCallableMemberD ProtoBuf based deserialized ClassConD. Holds a bunch of TypeTable, NameResolver, ContainerSource etc to help with further deserialization (so this is also in some sense lazy).

JClassConD: ClassConDImpl & JCallableMemberD A Java imported class's constructor. Since this is from JVM object code, it provides property impl of hasStableParamNames / hasSynthesizedParamNames.

enhance() implementation makes a copy with enhanced receiver param and value params.

SamAdapterClass: ClassConDImpl & SamAdapterD<JavaClassConD> A synthetic ClassConD that wraps another ClassConD.

4.2 DFactory

In core.descriptors / resolve.

5 FIR

compiler.fir.resolve: ResolutionStage sounds like something pipeline-related

6 IR

IR (compiler.ir) seems to be an lower-level IR. This is an experimental IR that's intended to be used across all Kotlin backends. Also see this writeup.

A sample IR lowering pass can be found here. Looks that the compiler API provides a convenient IrBuilder, and a visitor style IrTransformer.

6.1 compiler.ir.tree

IrSymbol definitions. See IrSymbolVisitor for an overview. Looks that they have descriptors attached.

7 Type System

7.1 Types

Kotlin compiler uses Marker Interface pattern extensively in type definitions. See `TypeSystemContext.kt` (`KotlinTypeMarker`, `TypeArgumentMarket` etc).

`KotlinType` is the base class for all types in Kotlin. It has a `tycon`, list of `tyargs`, nullability (so nullability is built-in to any type – can this be problematic?). It also has a `MemberScope` (see 8) and a `refine(KotlinTypeRefiner)` method.

7.2 `core.type-system`

type system core (equality, bounds checking etc). However this is more of an interface module – the actual impls are in `core.descriptors`, `fir` and `ir` modules.

7.3 `TypeSystemTypeFactoryContext`

Contains a bunch of common type factories:

- `flexibleType` has lower/upper bounds
- `simpleType` has `tycon`, `tyargs`, `nullablep`
- `tyarg` has `ty` and `variance`
- `star` has `tyarg` (why?)
- there's also an error type used in diagnosis

7.4 `TypeCheckerProviderContext`

- modular axioms (`errorType` unifiable with all types etc)
- what is a stub type? (Probably `PsiStub` related?)

7.5 `TypeSystemCommonSuperTypesContext`

Used to check if two type has common super types, and lowest-common ancestor utils.
`typeDepth` is a safe overestimation of the depth (from 'Any').
Seems to also be used in `Fir`.

7.6 `TypeSystemInferenceExtensionContext`

Inference related.

7.6.1 Questions

- What is a `isCapturedTypeConstructor`?
- What is a `singleBestRepresentative`?
- What is a `noInferAnnotation`?
- What is `maybeTypeVariable`?
- What is a `defaultType`?
- Read impl of `isUnit` vs `isUnitTypeConstructor`
- Read impl of `createCapturedType`
- Read impl of `createStubType`
- Read impl `createEmptySubstitutor`, `typeSubstitutorByTypeConstructor`, `safeSubstitute`

7.7 TypeSystemContext

- `fastCorrespondingSupertypes` has no actual impl? (No, it's just that IntelliJ's search functionality fail to find overridden extension methods)
- `isCommonFinalClassConstructor` is implemented in three (Psi, Fir, Ir) stage's `TypeSystemContext`:
 - `ClassicTypeSystemContext`: get `ClassDescriptor` from `TypeConstructor`'s `declarationDescriptor`, then check it's final but not (enum or annotation). So the method really checks that the tycon is "final" but is not a uncommon (enum/annotation) class.
 - `ConeTypeContext`: Does almost the same thing, but also return true if is anonymous object (final by design). Works on `FirBasedSymbol` (some sort of class infotable?). Check that this is a `FirRegularClassSymbol`, whose `FirRegularClass` is final but not uncommon.
 - `IrTypeSystemContext`: Check this is a `IrClassSymbol` whose owner is final and not uncommon.
So basically `ClassDescriptor`, `FirRegularClass` and `IrClassSymbol.owner` are the same thing across three stages.
Sounds that reading the common implemented methods of these three `TySysCtx` impl classes would be super helpful to understand the stages.

7.8 Type System for Fir

Read `ConeTypeContext`

7.9 Type System for Ir

Read IrTypeSystemContext

7.10 compiler.resolution/.inference

Type inference? constraint system, subst, fresh tycon, tyvar etc

7.11 compiler.frontend/.types

TypeIntersector (unify), DeferredType (I guess this is for when inference can't proceed at some first, and will retry when it has more information. Not really fully bidirectional (H-M style) type inference, but an approximation)

7.12 types.expressions

Contains a bunch of KtElement visitors that does type recon/checking:

- ExpressionTypingVisitorDispatcher
- ControlStructureTypingVisitor
- FunctionsTypingVisitor
- BasicExpressionTypingVisitor (constants etc)
 - This actually does a bit of parsing/validation work... e.g. understore on int literals.
 - Also uses ConstantExpressionEvaluator to check for possible compile time constants (this indeed sounds like something a parser would do).
Folds boolean && and ||
Look up simple unary and binary func in OperationsMapGenerated

8 Resolution

Package: compiler.resolution

8.1 Scopes

core.descriptor / ResolutionScope: contains information about what identifier it contributes to a given lookup location. Identifiers have separate namespaces: variable, function, and classifier (type).

compiler.resolution / LexicalScope is a ResolutionScope that has a parent, a ownerD, a LexicalScopeKind (what kind of syntactical structure created this scope?)

8.2 Tower

i.e. ImplicitScopeTower. Some sort of multi-level scopes? Can't understand this part.

9 Analysis

Package: `compiler.frontend`

9.1 `compiler.frontend.LazyTopDownAnalyzer`

9.1.1 `TopDownAnalysisContext`

Stores the toplevel declarations (in typed maps) found during analysis.

9.2 `BindingTrace`

Has a `BindingContext`. Is writable. Can record/inquiry `KotlinType` for a `KtElement`.
Impls: `BindingTraceContext` and `ObservingBindingTrace`

9.3 `BindingContext`

Sounds like a read-only counterpart to the `BindingTrace`.

9.4 `Smartcasting`

`compiler.frontend.smartcasts.DataFlowInfo`: bunch of maps to stores the data flow analysis result useful for smart casts.

`DataFlowValue`: one instance of a value in a dataflow

`DataFlowValue.Kind`: classify exprs into smart cast enabled, possible, or disabled ones. Quite intuitive.

`IdentifierInfo`: represents both qualifier and ident name. what is this for?

10 Codegen

Mostly in package `compiler.backend`.

10.1 `ExpressionCodeGen`

Generates JVM bytecode directly from `KtElement`. This is a `KtVisitor<StackValue2>`.
Has an `InstructionAdapter` (JVM bytecode emitter).

10.2 `StackValue`

Sounds like a helper class to represent operands on the JVM stack. Has a couple subclasses.

10.3 `ClassBodyCodeGen`

Generate class body from a `KtPureClassOrObject` with a `ClassDescriptor`. Also does bridge generation.

10.4 FunctionCodeGen

Generate class body from a `KtPureClassOrObject` with a `ClassDescriptor`. Also does bridge generation.

10.5 JS Codegen

e.g. `js.js.translator / PropertyTranslator`. Looks that it directly translates descriptor + Psi (`KtElement`) into JS statement / expressions.

Also see `FunctionBodyTranslator` – takes a `FunD` and a `KtElement` (`KtDeclWithBody`).

11 Unsolved Questions

- What is a `LazyTypeParameterDescriptor`?
- What is a `core.descriptors.MemberScope`?
- What's the diff between `TypeCheckerContext` and `TypeSystemContext`?

Nomenclature

Bridge	Kotlin's way (i.e. doesn't require Java 8) of representing default implementation for interfaces in generate code. See <code>compiler.backend-common / impl.kt</code> 's code doc.
Commonization	Seems to be a klib process to strip source and original information from descriptors. For example see <code>CommonizedClassConD</code> . Maybe this is to reduce storage size for these Ds in klibs?
Con	Constructor. Can be either a type constructor, or a value / class constructor like <code><init></code>
D	DeclarationDescriptor or just Descriptor. See the DeclarationDescriptor section
Deser	Deserialization. See <code>core.deserialization</code> – The compiler is able to serialize / deserialize descriptors. And it seems to be done lazily – Many deserialized descriptors still hold <code>TypeTable</code> / <code>NameResolver</code> / <code>VersionRequirementTable</code> to help with further deserialization.
FIR	Intermediate level IR, below <code>KtElement</code> and above IR. See 5.
Fun	Function
Impl	Implementation
IR	Can either be a general term (Intermediate Representation), or specifically Kotlin compiler's new experimental lowlevel IR (see 6).
J	Java
Param	Parameter
PSI	JetBrain's universal parse tree API. See its doc here.
Stub	In the context of PSI, this is the interface part of a PSI tree. It's initially calculated from PSI trees and then cached for efficient retrieval. See JetBrain's doc.
Subst	Substitution, usually in the context of type checking / reconstruction / unification algorithms. See Wikipedia: Unification and nLab: Substitution.
TyCon	Type constructor. Basically a type-level function that takes zero or more types, and returns another type. E.g. <code>List</code> is a tycon: $\forall a, \text{List}<a>$
Var	Variable
Vis	Visibility, as in <code>public</code> / <code>private</code> / <code>internal</code> etc.