Scala Type-safe API Design Doc

Introduction

This projects is Mid-sized with great impact to the current Scala API designs. It will include two major tasks:

- Design, implement and integrate the new Scala API
- Implement and Integrate the Scala docs generator to current pipeline

Design Proposal sent to the Dev-list

PRs:
https://github.com/apache/incubator-mxnet/pull/10660
https://github.com/apache/incubator-mxnet/pull/10787
https://github.com/apache/incubator-mxnet/pull/10991
https://github.com/apache/incubator-mxnet/pull/11063

The way to call the api will be Symbol.api and NDArray.api.

New Scala API

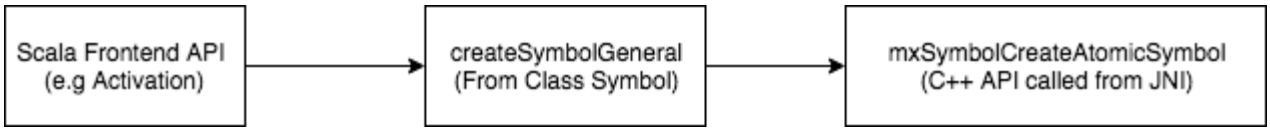
The new Scala API design is shown below:

```
def Activation(data : Option[Symbol], act_type : String, name : Option[String], attr : Option[Map[String,
String]]) : Symbol= {

    val map : mutable.Map[String, Any] = mutable.Map()
    if (!data.isEmpty) map("data") = data.get
    map("act_type") = act_type
    val currName = name.getOrElse("")
    val currAttr = attr.getOrElse(Map())
    createSymbolGeneral("Activation", currName, currAttr, Seq(), map.toMap)

}
```

This is a very simple example of how we can use Scala to use MXNet. The thing that behind this can be explained from this flowchart. This is how an Scala API called and executed (Python API have a similar approach).



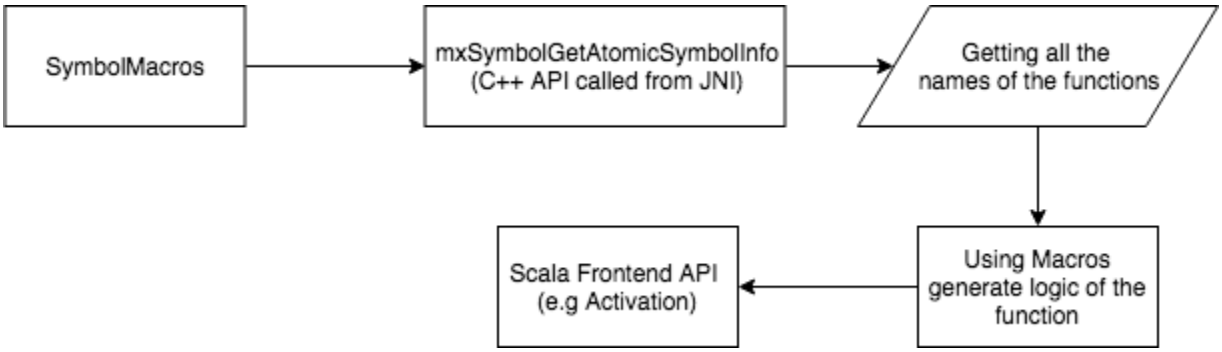
Current implementation of this API is very limited, we need to put all parameters as key-value args in a map and send to the backend to process. User will have no ideas what parameters are required for each of the function. The implementation of the old API is as shown below:

```
def Activation(name : String = null, attr : Map[String, String] = null)
 (args : Symbol*)(kwargs : Map[String, Any] = null) : Symbol = {
    createSymbolGeneral("Activation",name,attr,args,kwargs)
}
```

We are proposing a new design of the API, brings different arguments to different functions as well as a full implementation on the backend side in order to execute them.

The old Scala API Macros design

Before we introduce how we implement this, let's start from how we generated the code before 1.3:



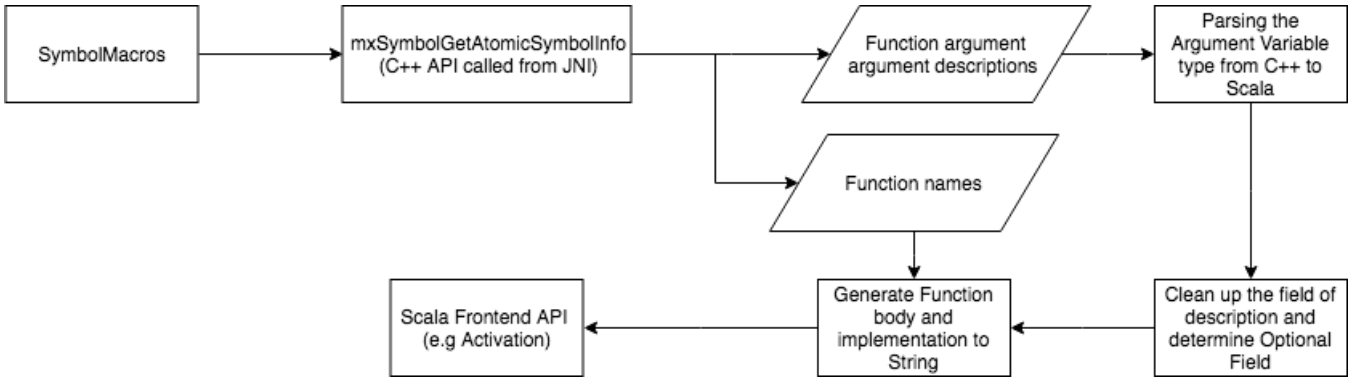
We use Scala Macros that generate the code in here. the “mxSymbolGetAtomicSymbolInfo” function would bring us all of the information of functions that used in Symbol and NDArray. The old Scala API only takes the name and create a universal argument field for all of the APIs. This is limited by the poor Code generation support from old Scala version. We need to write a lot of the Parser code in order to translate a function in the code generator.

New Scala API macros

In order to solve this problem, we have studied deeply into the current Scala version and found a great methods that could simplify the process and help us to build a better function namely quasiquote. It is a super powerful tool which can automatically parse string into the function structure as shown above. The implementation of the Activation function following universal argument field can be simplified as shown below:

```
val funcName = symbolfunction.name
val tName = TermName(funcName)
     q"""
         def $tName(name : String = null, attr : Map[String, String] = null)
         (args : org.apache.mxnet.Symbol*)(kwargs : Map[String, Any] = null)
          : org.apache.mxnet.Symbol = {
           createSymbolGeneral($funcName,name,attr,args,kwargs)
           }
      """
```

It brings developer lesser code and much clearer view of what code will be generated in this case. Inspired from this method, we are creating a brand new Pipeline in Macros to generate code:



In brief, we need to extract the argument description into an individual section, translate it into a Scala type and set it to required/optional field. We also need to create a template for the new APIs, how it constructed. After we finish the previous steps, we can make use of the previous information to generate the implementation for new API.

The hardest component in this pipeline is how to use limited Scala docs for code generation. We will discuss this in details in the following sections.

API Generation from step to step

Raw data from C++

The raw data includes the following fields:

- name: The function name
- desc: The description of the function
- argNames: Argument names

- argTypes: the type for each argument
- argDescs: The Argument description

As discussed previously, we need to use the API that provided from C++ to generated corresponding Scala APIs. The C++ side only provide a type description rather than an actual type in order to make sure the other language developer can correctly translate that into the correct type. The argType will display in the following formats:

`<C++ Type>, <Required/Optional>, <Default=>`

C++ to Scala type converter

```
  // Convert C++ Types to Scala Types
  def typeConversion(in : String, argType : String = "") : String = {
    in match {
      case "Shape(tuple)" | "ShapeorNone" => "org.apache.mxnet.Shape"
      case "Symbol" | "NDArray" | "NDArray-or-Symbol" => "org.apache.mxnet.Symbol"
      case "Symbol[]" | "NDArray[]" | "NDArray-or-Symbol[]" | "SymbolorSymbol[]"
      => "Array[org.apache.mxnet.Symbol]"
      case "float" | "real_t" | "floatorNone" => "org.apache.mxnet.Base.MXFloat"
      case "int" | "intorNone" | "int(non-negative)" => "Int"
      case "long" | "long(non-negative)" => "Long"
      case "double" | "doubleorNone" => "Double"
      case "string" => "String"
      case "boolean" => "Boolean"
      case "tupleof<float>" | "tupleof<double>" | "ptr" | "" => "Any"
      case default => throw new IllegalArgumentException(
        s"Invalid type for args: $default, $argType")
    }
  }
```

This conversion tool will catch all possible types that defines in C++ and translate it into something that can be used in Scala. It is not finished fully, as you can see, we still have some types defined as type “Any”.

```
def argumentCleaner(argType : String) : (String, Boolean) = {
  val spaceRemoved = argType.replaceAll("\\s+", "")
  var commaRemoved : Array[String] = new Array[String](0)
  // Deal with the case e.g: stype : {'csr', 'default', 'row_sparse'}  if (spaceRemoved.charAt(0)== '{') {
    val endIdx = spaceRemoved.indexOf('}')
    commaRemoved = spaceRemoved.substring(endIdx + 1).split(",")
    commaRemoved(0) = "string"  } else {
    commaRemoved = spaceRemoved.split(",")
  }
  // Optional Field  if (commaRemoved.length >= 3) {
    // arg: Type, optional, default = Null    require(commaRemoved(1).equals("optional"))
    require(commaRemoved(2).startsWith("default="))
    (typeConversion(commaRemoved(0), argType), true)
  } else if (commaRemoved.length == 2 || commaRemoved.length == 1) {
    val tempType = typeConversion(commaRemoved(0), argType)
    val tempOptional = tempType.equals("org.apache.mxnet.Symbol")
    (tempType, tempOptional)
  } else {
    throw new IllegalArgumentException(
      s"Unrecognized arg field: $argType, ${commaRemoved.length}")
  }

}
```

The following code would do the following translation

`<C++ Type>, <Required/Optional>, <Default=>`

`<Scala Type>, <is Optional>`

SCala marcos code generation

After we finish up the cleaning part, we can make a start building up the Functions in the class. The target for this will be creating a series of functions that lives in an existing object. However, there is no such a tool kit for us to build a Scala API conveniently. The accepted input from Scala will looks like below:

```
DefDef(functionScope, TermName(newName), List(),
        List(
          List(
            ValDef(Modifiers(Flag.PARAM | Flag.DEFAULTPARAM), TermName("name"),
              Ident(TypeName("String")), Literal(Constant(null))),
            ValDef(Modifiers(Flag.PARAM | Flag.DEFAULTPARAM), TermName("attr"),
              AST_TYPE_MAP_STRING_STRING, Literal(Constant(null)))
          ),
          List(
            ValDef(Modifiers(), TermName("args"), AST_TYPE_SYMBOL_VARARG, EmptyTree)
          ),
          List(
            ValDef(Modifiers(Flag.PARAM | Flag.DEFAULTPARAM), TermName("kwargs"),
              AST_TYPE_MAP_STRING_ANY, Literal(Constant(null)))
          )
        ), TypeTree(),
        Apply(
          Ident(TermName("createSymbolGeneral")),
          List(
            Literal(Constant(funcName)),
            Ident(TermName("name")),
            Ident(TermName("attr")),
            Ident(TermName("args")),
            Ident(TermName("kwargs"))
          )
        )
      )
```

Let's walk through this part of the code, “DefDef()” will return a function back as we placed functionScope inside. It contains a “TermName” which contains the name of the function. Then a 2D list contains the structure of ()()(). First List in the List contains arg “name : String = null” and “attr : Map[String, String] = null”. Second one is “arg : Symbol*” and third one is “kwargs : Map[String, Any] = null”. Then, we can define what inside the function, where Apply() means the return line. This function returns a Symbol.

This is what the previous code generated, a function with complete implementation.

However, we decide to change the current implementation to a dynamic way where the args and implementation field will be changed dynamically.

```
def $funcName(args ...) : Symbol/NDArray = {
    dynamic implementation ...
}
```

This cannot be done easily through the structure introduced before. From Scala version 2.10, there is a Quasiquote toolkit provided for developers. It can convert String typed code into the real implementation just as shown above.

```
blackbox.Context.parse(string)
```

Use this single line, we can convert string type definition and implementation into a TreeType, which used for code generation.

Integration to current API

We name this API into a namespace as: Symbol.api.<Function name>. This api contains every functions that generated from new Macros

```
package org.apache.mxnet

@AddSymbolAPIs(false)
/**
* typesafe Symbol API: Symbol.api._
* Main code will be generated during compile time through Macros
*/
object SymbolAPI extends SymbolAPIBase {
}
```

API Docs generation

Due to the limitation of the Macros generation, the API docs would not come as a part of the current code. In order to achieve that, we call the same methods (mxSymbolGetAtomicSymbolInfo) to obtain all the information related to the function as well as the arguments. Currently, the implementation was done by writing files as abstract class and use a plugin to execute in the Macros compile stage. This use the same C++ to Scala functions to generate function body.

```
def $tName(name : String = null, attr : Map[String, String] = null)
         (args : org.apache.mxnet.Symbol*)(kwargs : Map[String, Any] = null)
          : org.apache.mxnet.Symbol = {
           createSymbolGeneral($funcName,name,attr,args,kwargs)
           }
```