Zeppelin源码分析

Zeppelin涉及的框架层面的几个模块，如下的依赖关系：



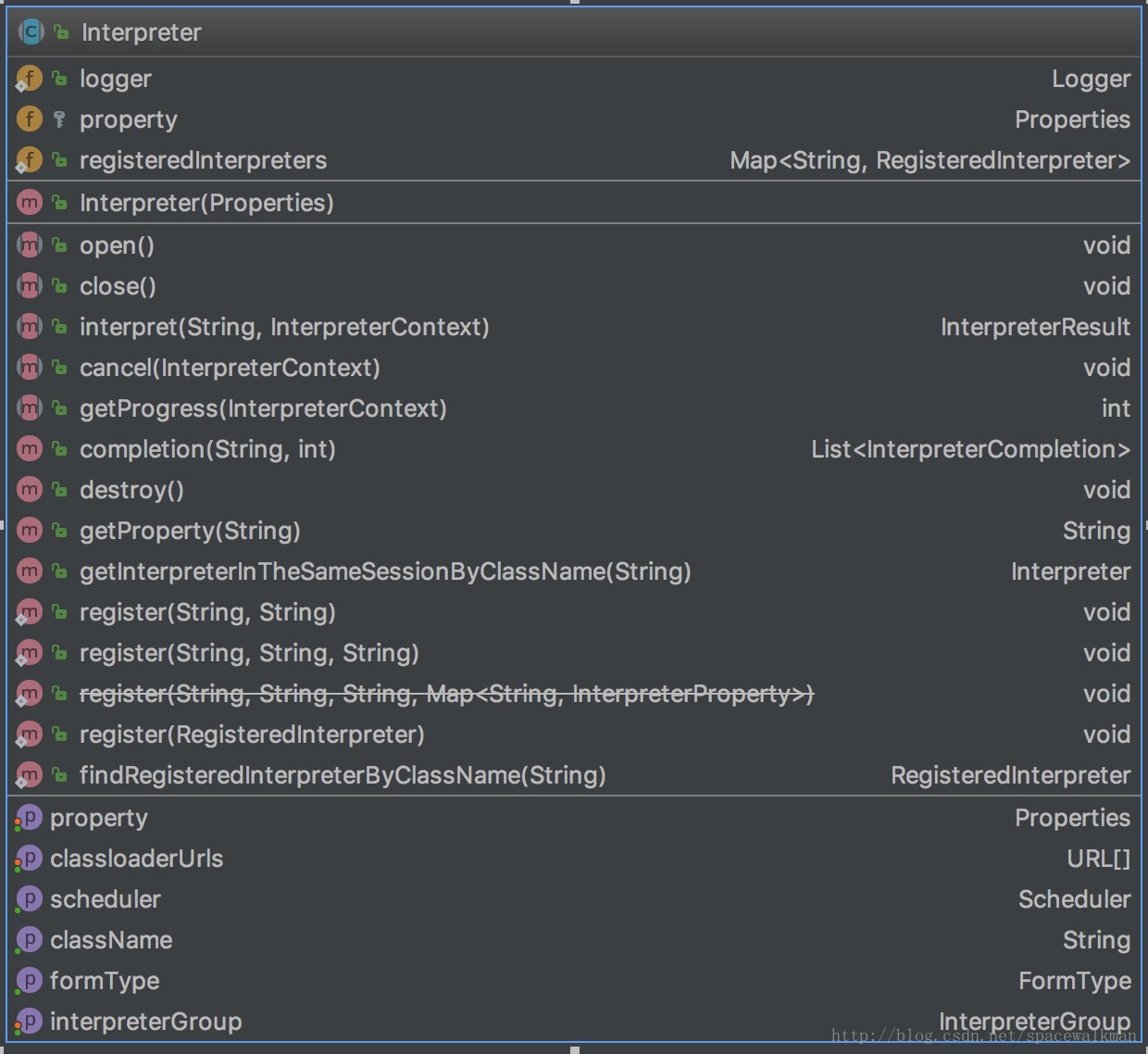
在Zeppelin中的核心类都来自这3个Module，核心类图如下：



下面从Interpreter,Interpreter的执行,noteBook持久化及Notebook的执行几个方面进行介绍

# 1.interprter

Interpreter是Zeppelin的核心，其提供多语言的repl解释器，如上图的右上角标识区域。Interpreter是一个抽象类，解释执行多种语言的代码，各个解释器继承Interpreter类，接口如下所示：



Interpreter主要规定了各语言repl解释器需要遵循的规范，包括：

1）repl解释器的生命周期管理，如open,close,destroy等，规定了产生和销毁repl解释器

2）解释执行代码的接口，interpreter，这是解释器的核心所在

3）执行代码过中交互控制和易用性增强，如cancel,getProgress,completion，分别终止代码的执行，获取执行进度以及代码自动完成

4）解释器的配置接口，如setProperty,setClassLoaderURL等

5）性能优化接口，如getScheduler,getInterpreterGroup等

6）解释器注册接口，如一系列重载的register接口。解释器的注册有两种方式：将interpreter-setting.json打包到jar文件中；放置到/interpreter/{interpreter}/interpreter-setting.json文件中。

以下体现了Zeppelin的repl解释器进程的执行，解释器进程受主进程ZeppelinServer的控制。

## 1.1 RemoteInterpreterServer Thrift协议分析

Apache Thrift是跨语言的RPC通信框架，提供相应的DSL和支持多种语言的代码生成工具。Zeppelin使用Thrift定义了主进程ZeppelinServer与需要采用独立的JVM进程运行的Repl解释器之间的通信协议，之所以采用单独JVM进程，主要考虑：

1. Zeppelin旨在提供一个开放的框架，支持多种语言和产品，由于各种语言和产品都是独立演进的，所以依赖不相同，甚至相互冲突
2. 在大数据分析领域，是否具有横向扩展能力是production-ready的一项重要衡量指标，将repl进程和主进程分开，会有利于扩展性

因此，Zeppelin采用独立JVM的方式启动repl进程后，使用Thrift协议定义主进程和RemoteInterpreterService进程之间的通信协议，具体如下：

*service RemoteInterpreterService {*

*void createInterpreter(1: string intpGroupId, 2: string noteId, 3: string className, 4: map<string, string> properties);*

*void open(1: string noteId, 2: string className);*

*void close(1: string noteId, 2: string className);*

*RemoteInterpreterResult interpret(1: string noteId, 2: string className, 3: string st, 4: RemoteInterpreterContext interpreterContext);*

*void cancel(1: string noteId, 2: string className, 3: RemoteInterpreterContext interpreterContext);*

*i32 getProgress(1: string noteId, 2: string className, 3: RemoteInterpreterContext interpreterContext);*

*string getFormType(1: string noteId, 2: string className);*

*list<InterpreterCompletion> completion(1: string noteId, 2: string className, 3: string buf, 4: i32 cursor);*

*void shutdown();*

*string getStatus(1: string noteId, 2:string jobId);*

*RemoteInterpreterEvent getEvent();*

*// as a response, ZeppelinServer send list of resources to Interpreter process*

*void resourcePoolResponseGetAll(1: list<string> resources);*

*// as a response, ZeppelinServer send serialized value of resource*

*void resourceResponseGet(1: string resourceId, 2: binary object);*

*// get all resources in the interpreter process*

*list<string> resourcePoolGetAll(); // get value of resource*

*binary resourceGet(1: string noteId, 2: string paragraphId, 3: string resourceName);*

*// remove resource*

*bool resourceRemove(1: string noteId, 2: string paragraphId, 3:string resourceName);*

*void angularObjectUpdate(1: string name, 2: string noteId, 3: string paragraphId, 4: string*

*object);*

*void angularObjectAdd(1: string name, 2: string noteId, 3: string paragraphId, 4: string object);*

*void angularObjectRemove(1: string name, 2: string noteId, 3: string paragraphId);*

*void angularRegistryPush(1: string registry);*

*}*

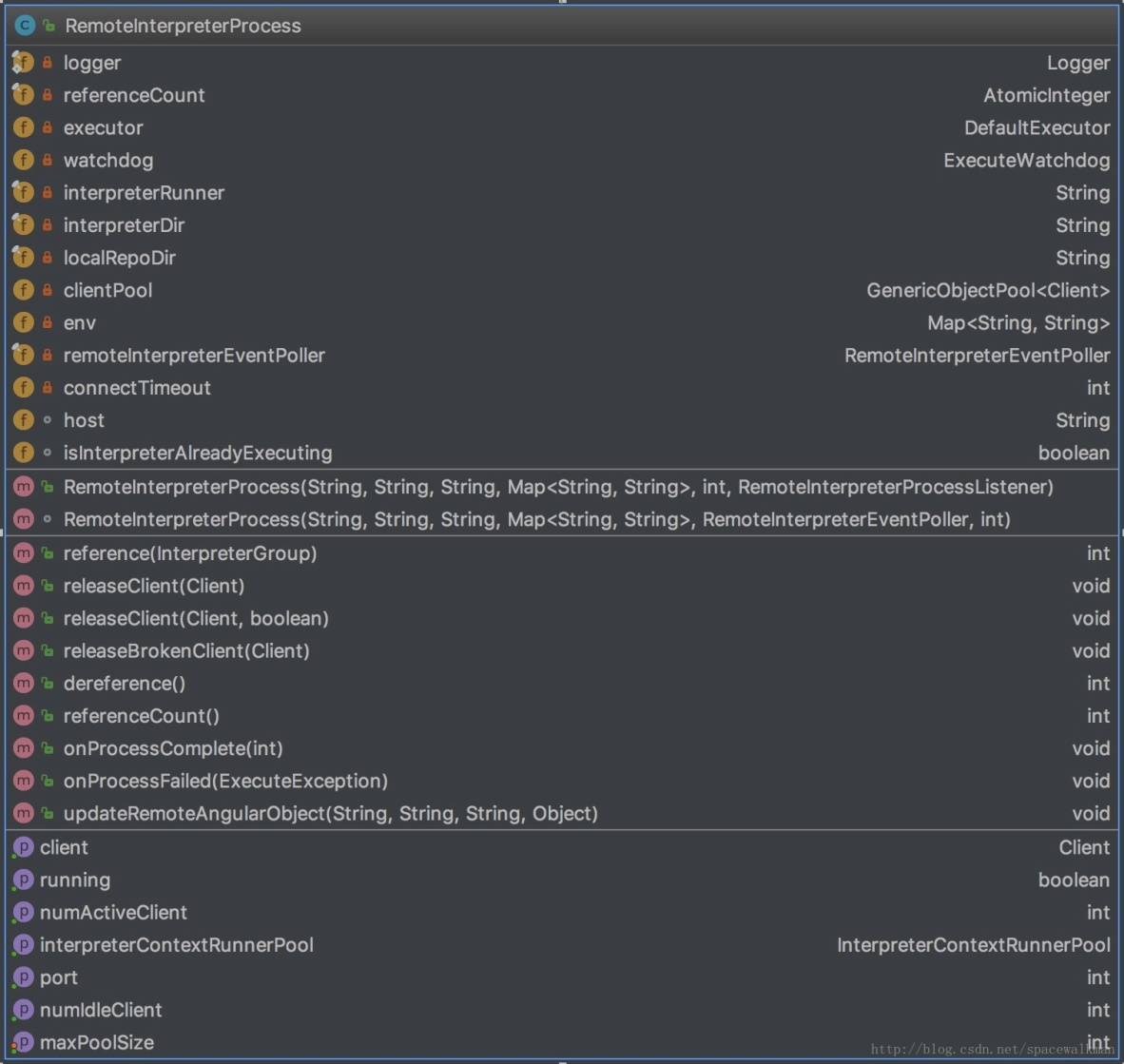
与前面的Interpreter类的定义进行对比发现，RemoteInterpreterService Thrift接口与Interpreter抽象类定义的接口大部分相同，不同之处在于：

1. RemoteInterpreterService接口的实现类由于运行在不同的JVM中，每个接口方法中额外传递环境信息，如noteId和ClassName等
2. RemoteInterpreterService接口中多出两种类型的接口，一种是为了完成ZeppelinServer进程和RemoteInterpreter进程之间的resource协商（neigotiation），如resourcexxx接口，另外是为了完成两种之间的augular object的前后台双向绑定

具体文件为：RemoteInterpreterServer.thrift

## 1.2 RemoteInterpreterProcess

RemoteInterpreterProcess是采用独立JVM启动repl进程的具体执行类，采用Apache Commons Exec框架来根据Zeppelin进程的的指示启动独立进程：



具体逻辑如下：

*port = RemoteInterpreterUtils.findRandomAvailablePortOnAllLocalInterfaces();*

*CommandLine cmdLine = CommandLine.parse(interpreterRunner);*

*cmdLine.addArgument("-d", false);*

*cmdLine.addArgument(interpreterDir, false);*

*cmdLine.addArgument("-p", false);*

*cmdLine.addArgument(Integer.toString(port), false);*

*cmdLine.addArgument("-l", false);*

*cmdLine.addArgument(localRepoDir, false);*

*executor = new DefaultExecutor();*

*watchdog = new ExecuteWatchdog(ExecuteWatchdog.INFINITE\_TIMEOUT);*

*executor.setWatchdog(watchdog);*

*running = true;*

*try {*

*Map procEnv = EnvironmentUtils.getProcEnvironment();*

*procEnv.putAll(env);*

*logger.info("Run interpreter process {}", cmdLine);*

*executor.execute(cmdLine, procEnv, this);*

*} catch (IOException e) {*

*//省略...*

*}*

1. 该进程端口是Zeppelin自动寻找OS中当前可用端口
2. RemoteInterpreterProcess并非在构造函数中，启动JVM，而是在被引用reference方法中调用之后，才启动
3. 具体的InterpreterRunner脚本为bin/interperter.sh

Interpreter.sh的重点部分如下：

*ZEPPELIN\_SERVER=org.apache.zeppelin.interpreter.remote.RemoteInterpreterServer*

*SPARK\_APP\_JAR="$(ls ${ZEPPELIN\_HOME}/interpreter/spark/zeppelin-spark\*.jar)"*

*if [[ -n "${SPARK\_SUBMIT}" ]]; then*

*${SPARK\_SUBMIT} --class ${ZEPPELIN\_SERVER} --driver-class-path "${ZEPPELIN\_INTP\_CLASSPATH\_OVERRIDES}:${CLASSPATH}" --driver-java-options "${JAVA\_INTP\_OPTS}" ${SPARK\_SUBMIT\_OPTIONS} ${SPARK\_APP\_JAR} ${PORT} &*

*else*

*${ZEPPELIN\_RUNNER} ${JAVA\_INTP\_OPTS} ${ZEPPELIN\_INTP\_MEM} -cp ${ZEPPELIN\_INTP\_CLASSPATH\_OVERRIDES}:${CLASSPATH} ${ZEPPELIN\_SERVER} ${PORT} &*

*fi*

在启动的单独JVM进程，执行RemoteInterpreterServer，并向其main方法中，传递必要的参数。

## 1.3 RemoteInterpreter

其本质是远程Interpreter的本地 代理，是Proxy模式的典型应用，运行在Zeppelin主进程中，通过Thrift服务的Client来控制远程Interpreter的执行，其初始化代码，如下：

*//省略掉了出错处理等其他内容*

*public synchronized void init() {*

*RemoteInterpreterProcess interpreterProcess = getInterpreterProcess();*

*final InterpreterGroup interpreterGroup = getInterpreterGroup();*

*interpreterProcess.reference(interpreterGroup);*

*synchronized (interpreterProcess) {*

*Client client = interpreterProcess.getClient();*

*client.createInterpreter(groupId, noteId, getClassName(), (Map) property);*

*}*

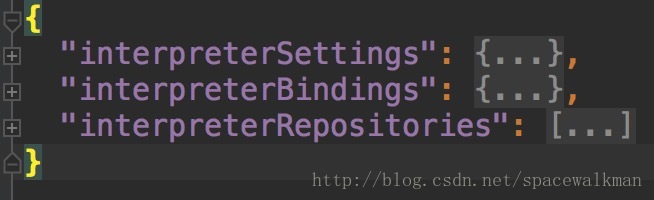
*}*

## 1.4 InterpreterInfoSaving

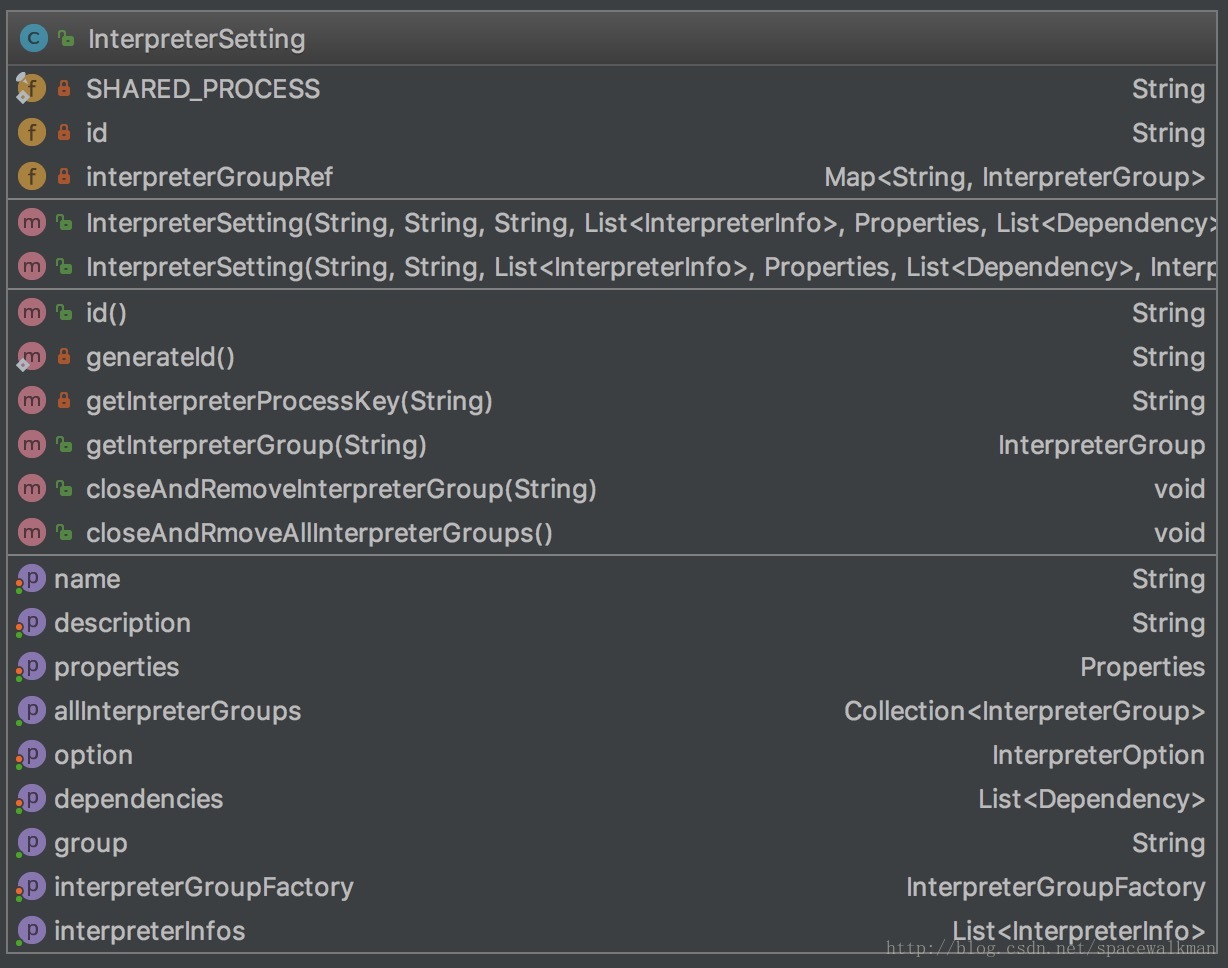
InterpreterInfoSaving是一个Convenient类，由其他类组成完成，设计目的是为了方便调用，定义的目的是将原来分散存储在各个Interpreter子文件夹中的InterpreterSetting、InterpreterBinding和InterpreterRepositories集中到一处，方便查找和持久化。

Interpreter的注册方式是采用Json配置来保存，Zeppelin启动时加载，该文件为conf/interpreter-setting.json，文件内容示例如下所示：

InterpreterInfoSaving类唯一实例持久化如下图：



1. InterpreterSettings



InterpreterSettings是Interpreter创建的Contract，其职责如下：

维护了interpreter相关元信息，这些信息对于正确运行interpreter进程至关重要，如外界环境交互，第三方依赖加载等

维护了note和interpreterGroup之间的关系，如下所示：

*interpreterSettings": {*

*"2CSR9KAQW": {*

*"id": "2CSR9KAQW",*

*"name": "bigquery",*

*"group": "bigquery",*

*"properties": {*

*"zeppelin.bigquery.max\_no\_of\_rows": "100000",*

*"zeppelin.bigquery.project\_id": " ",*

*"zeppelin.bigquery.wait\_time": "5000"*

*},*

*"status": "READY",*

*"interpreterGroup": [*

*{*

*"name": "sql",*

*"class": "org.apache.zeppelin.bigquery.BigQueryInterpreter",*

*"defaultInterpreter": false,*

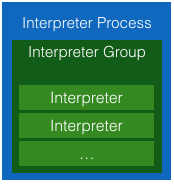
*"editor": {*

*"language": "sql",*

*"editOnDblClick": false*

*} }*

对于每个Interpreter进程，可以有一个或者多个interpreterGroup。三种关系如下：



InterpreterSettings实例被InterpreterFactory对象初始化并填充

1. Interpreter for note

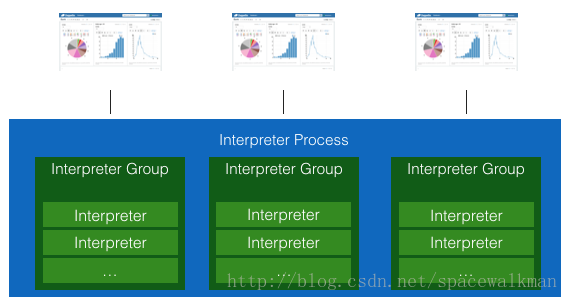
在创建多个Note时，每个Note会有不同的Interpreter进程还是共享同一个Interpreter进程，Zeppelin UI有三种模式：shared,scoped及isolated

* Shared

一种解释器只有一个Interpreter进程，并且该进程中只有一个InterpreterGroup，所有的Interpreter实例都属于该InterpreterGroup。

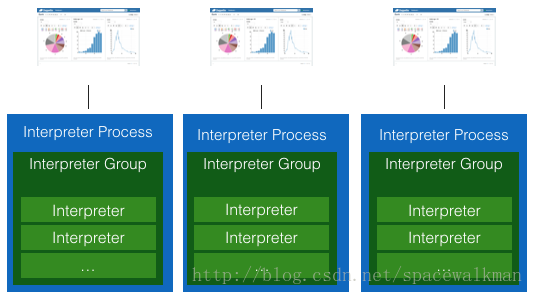
* Scoped

一种repl解释器只有一个Interpreter进程，但是与Shared模式不同，会创建过个InterpreterGroup，每个note关联一个InterpreterGroup。这样每个note相当于有了自己的session，session与session相互隔离，但是仍然由于这些InterpreterGroup仍然在同一个进程中，仍然可以在它们之间共享变量。



* Isolated模式

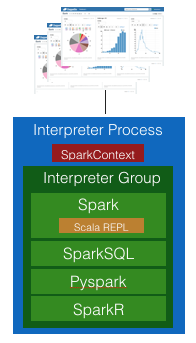
独占式，每个Note创建一个独立的Interpreter进程，该进程中创建一个InterpreterGroup实例，为该note的服务的Interpreter实例从属于该InterpreterGroup。



1. SparkInterperter的三种模式

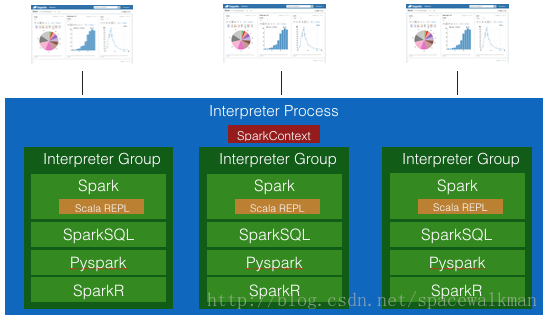
* SparkInterpreter Shared

所有的note共享同一个SparkContext和[scala](http://lib.csdn.net/base/scala) REPL实例，因此，如果其中一个note定义了变量a，另外一个note可以访问并且修改该变量a。



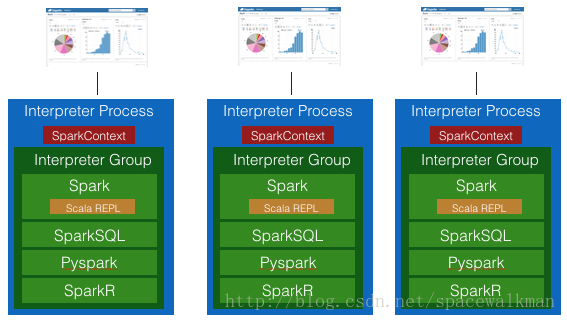
* SparkInterpreter Scoped

所有的note共享同一个SparkContext，所有的spark job都是通过同一个SparkContext提交的，但是不同的Scala repl解释器，由于不同享Scala repl，故不存在一个note访问并修改了另一个note定义的变量的问题。



* SparkInterpreter Isolated

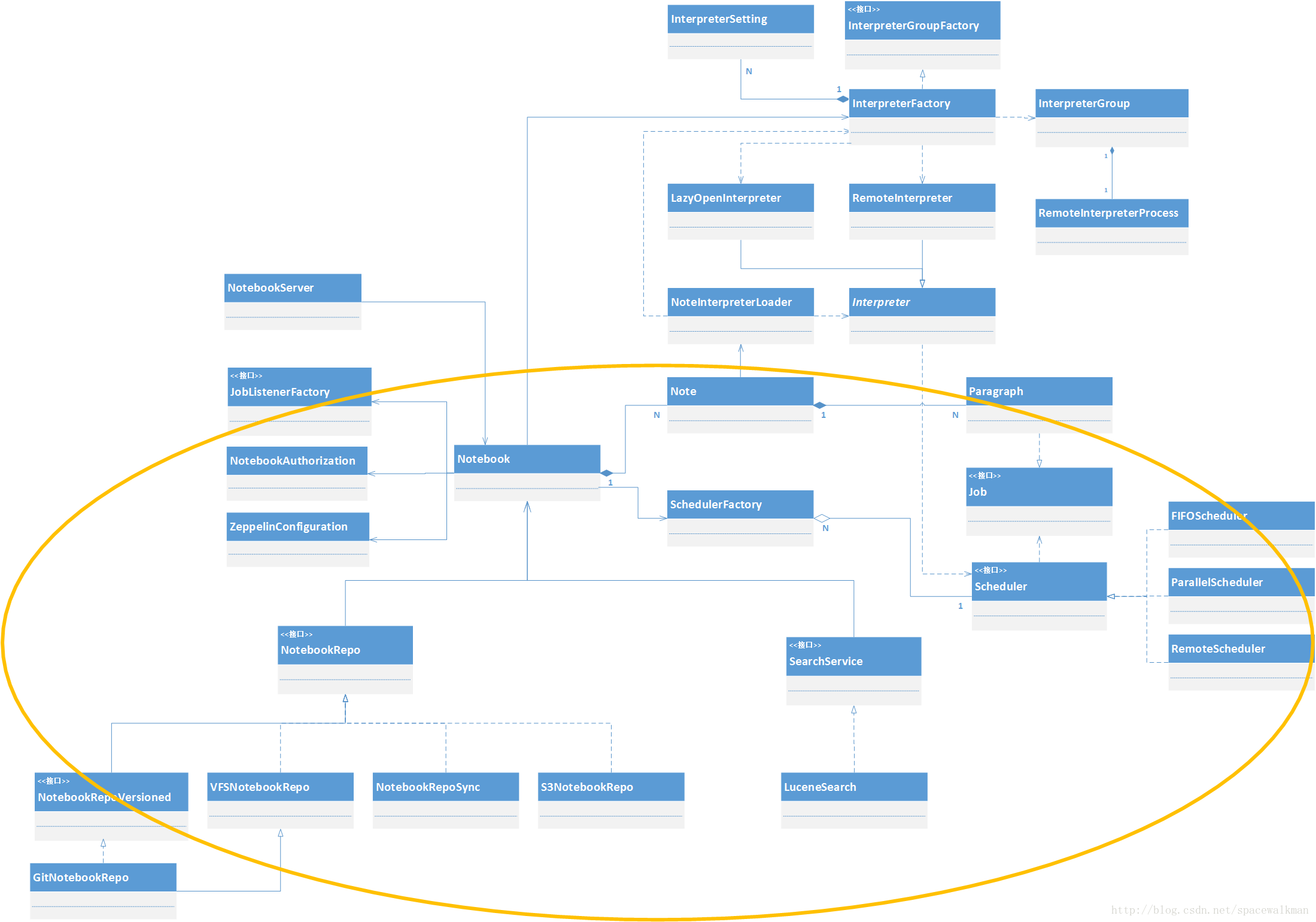
独占式，每个note都有自己的SparkContext和Scala repl，不共享。



其实这三种模式，底层都是通过InterpreterOption类来控制的

# 2.Note

Note是Zeppelin的另一个核心，涉及的类如下所示



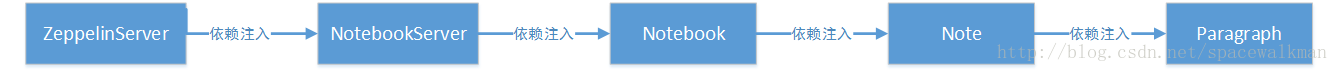
Note是单个记事本的内存对象，是Zeppelin管理的最小单位，无论是做权限控制、共享还是持久化，都是以Note为粒度。从类关系上看，Note是由一些列的有序Paragrapsh组成，大部分职责与管理Paragraph有关：

* Paragraph的CRUD、相对顺序控制
* 与处理前后端数据双向推送的AngularObject的管理
* 整体和单个Paragraph 执行，以及执行过程的基于Observer模式的执行过程Hook
* Note基本的样式外观控制

为了分离关注点，其他功能：

* Note相关的Interpreter加载和初始化
* 持久化与反持久化，包括延迟持久化
* 权限控制

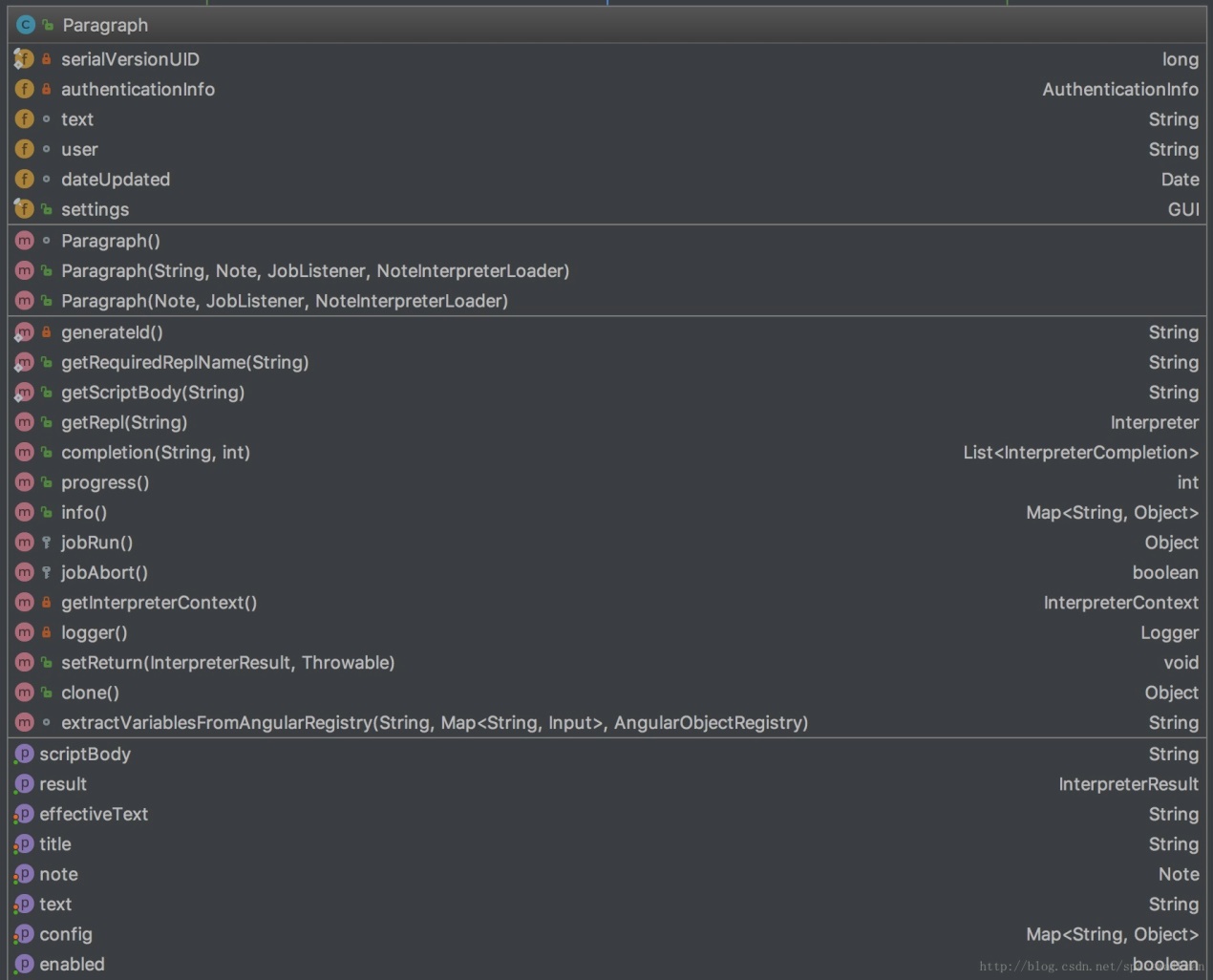
都交给Note依赖的服务来解决，这些服务在运行时，会由组合Note的Notebook类来负责注入，注入顺序为：



## 2.1 Paragrah

Paragraph代表一段代码以及支持其执行所需要的环境信息，是代码执行的最小单位。Paragraph的职责如下：

* + 获取代码文本，并解析分离类似%[Spark](http://lib.csdn.net/base/spark)的interpreter声明段和可执行代码段。
  + 代码执行，以及执行过程控制（进度和终止）
  + 代码执行结果获取
  + 代码中变量查找以及替换



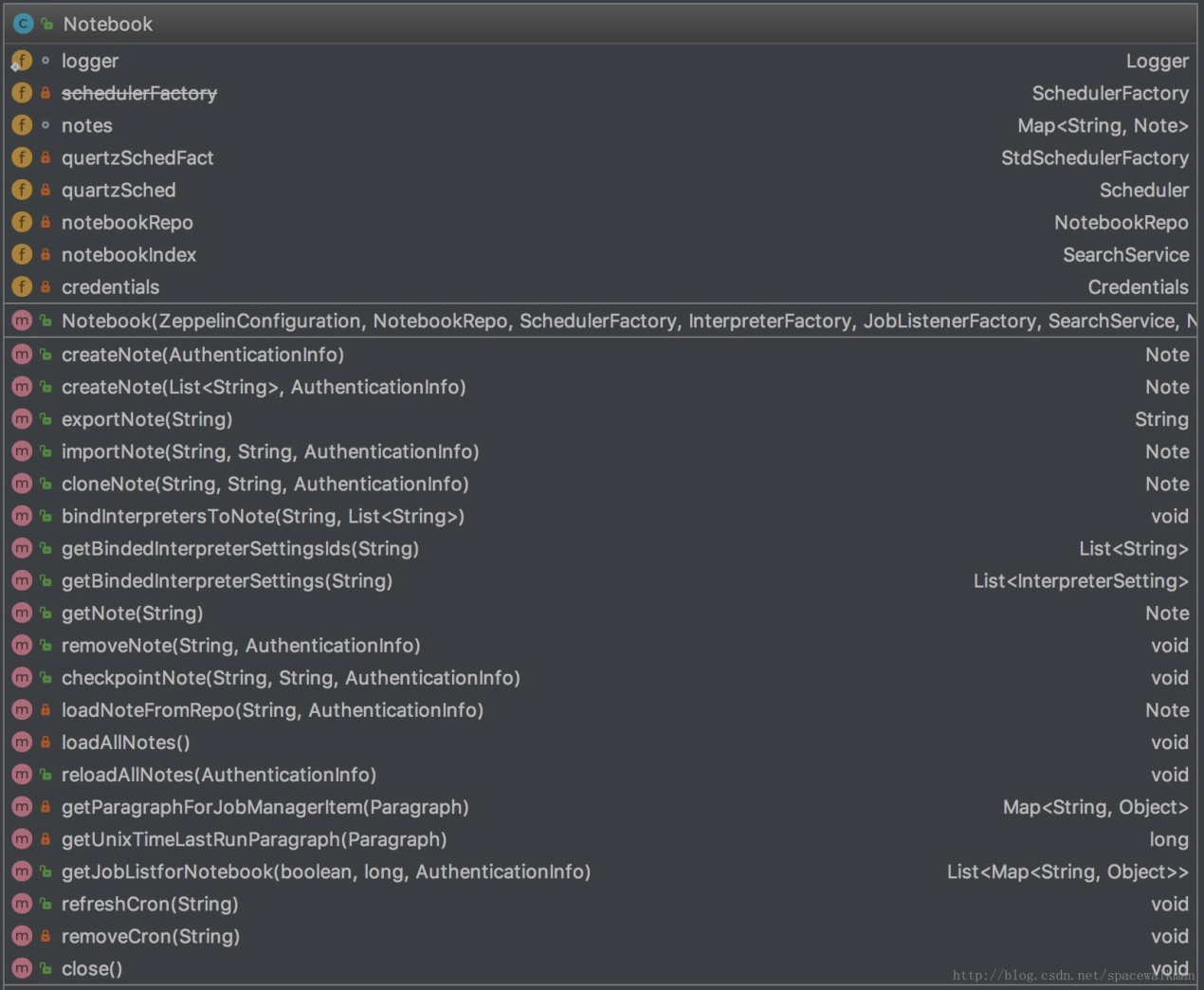
## 2.2 Notebook

Notebook实际上是Note的Manager，职责如下：

* + Note的CRUD，克隆、导入/导出
  + Note和相关Interpreter配置时和运行时映射关系维护
  + Note cron式调度执行控制

其他所有Note公共的服务，都交给ZeppelinServer类来注入，具体的有：

| 服务 | 说明 |
| --- | --- |
| NotebookRepo | Note的持久化服务 |
| SearchService | Note的全文检索服务 |
| NotebookAuthorization | Note的Authorization服务 |
| Credentials | 数据源相关的“用户/密码”服务 |



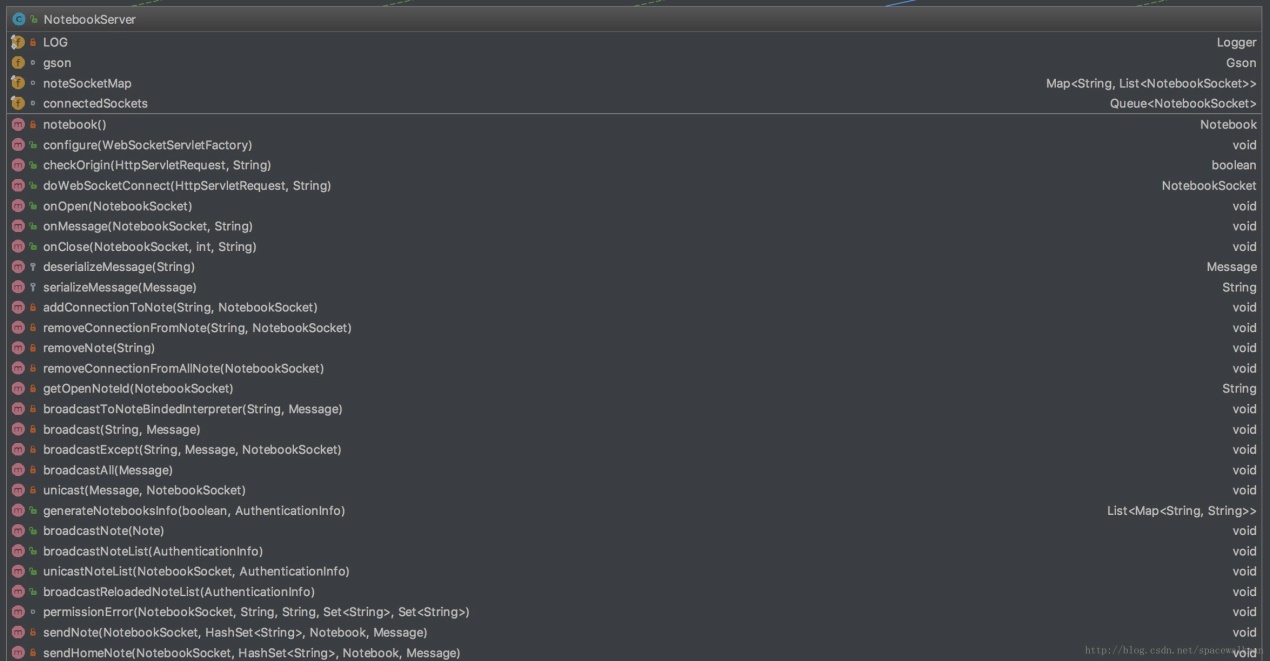
## 2.3 NotebookServer

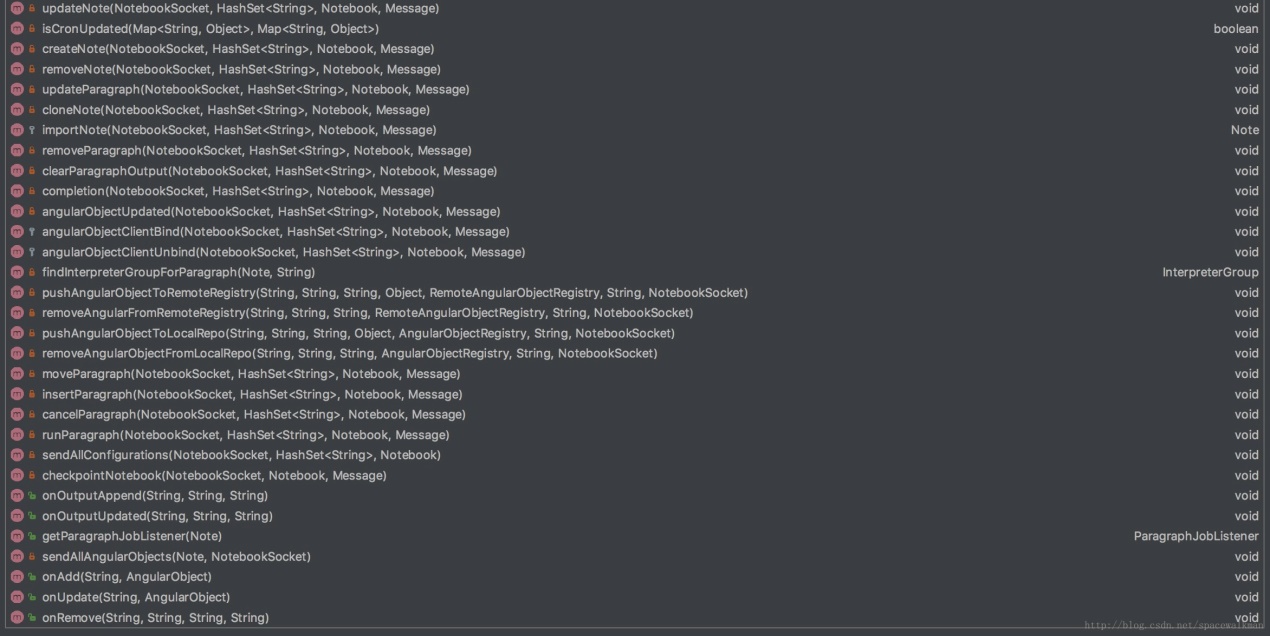
NotebookServer的主要功能是将Notebook,Note,Paragraph,Interpreter等类封装的能力，通过WebSocket形式对web客户端提供出去，具体功能包括：

* + 维护WebSocket连接与Note之间映射关系
  + 处理客户端和服务器之间的双向通信（通过WebSocket，具体的通信协议见：Message类），包括消息的序列化/反序列化，消息解析和服务端处理、处理结果的向客户端广播/单播发送等。
  + Note的CRUD操作以及Paragraph的CRUD操作、执行、导入、导出时的权限控制
  + 前后端AngularObject的双向bind处理
  + WebSocket客户端合法性校验(checkOrigin)

Zeppelin是共享式，Notebook式的大数据分析环境，以repl的方式执行以Paragraph为最小粒度的代码段：

* + 首先repl的方式强调实时反馈执行结果，特别是在大数据环境下，一段代码可能需要执行很长时间，在执行的过程中，zeppelin的用户期望看到执行进度和中间结果，需要在前后端之间建立一个长连接，便于实时传递数据。
  + 另外zeppelin的另一个亮点是其结果可视化能力，需要在前后台传递图片，并且支持较大数据量的传输的能力（相对传统http技术）。
  + 再者，由于是共享式环境，一个Note可能被多个用户同时看到、甚至编辑，需要在各个已经打开了同一个Note的web客户端之间同步Note的代码、执行结果和进度信息。

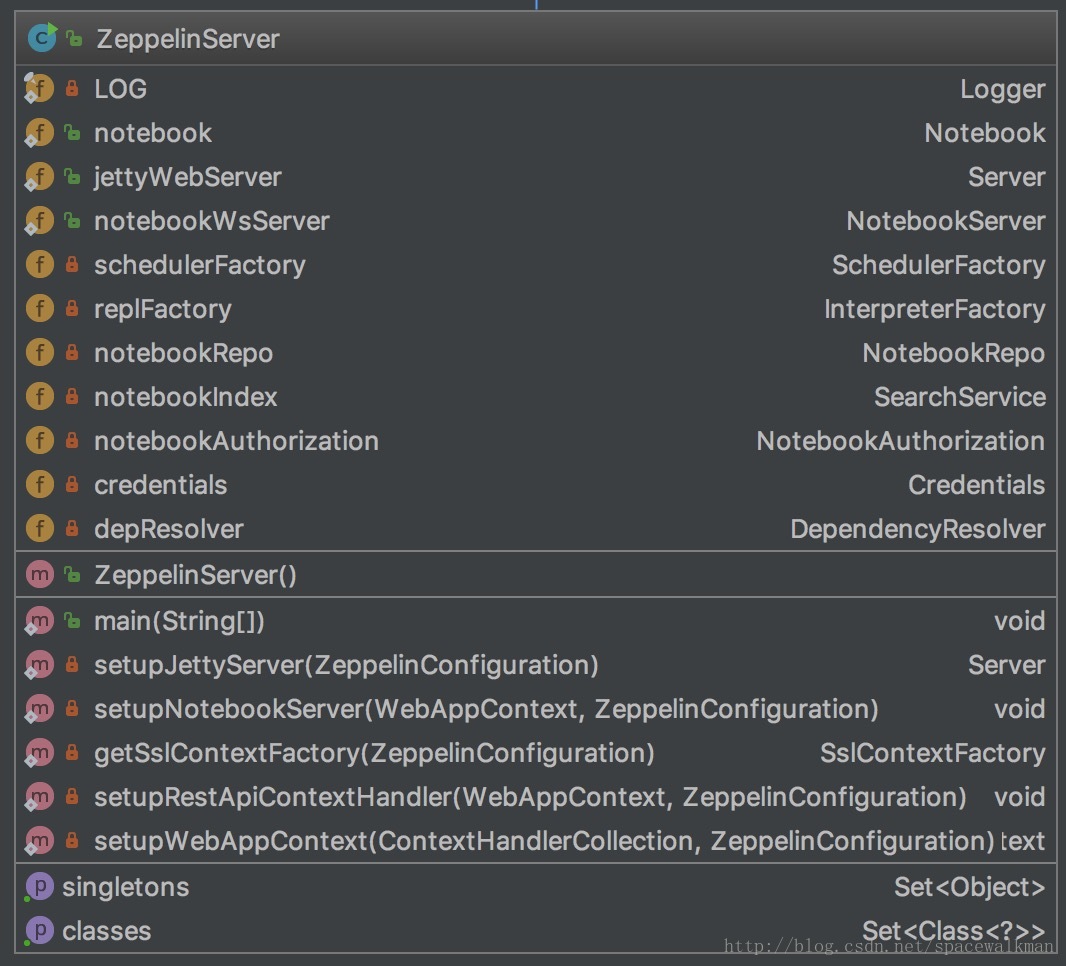




## 2.4 ZeppelinServer

ZeppelinServer是各个组件的组装者，是系统的主入口，功能如下：

* + 内嵌jetty服务器，支持以WebSocket和REST两种方式对外暴露系统功能
  + 创建NotebookServer实例，建立起处理WebSocket Connection和消息处理的服务端
  + 创建Notebook需要的相关依赖，如Note持久化服务(NotebookRepo)、Note的全文索引服务（SearchService），并完成向Note、Paragraph的注入。
  + Note权限配置文件的加载以及初始化
  + InterpreterFactory的初始化
  + 初始化动态依赖加载器(DependencyResolver)

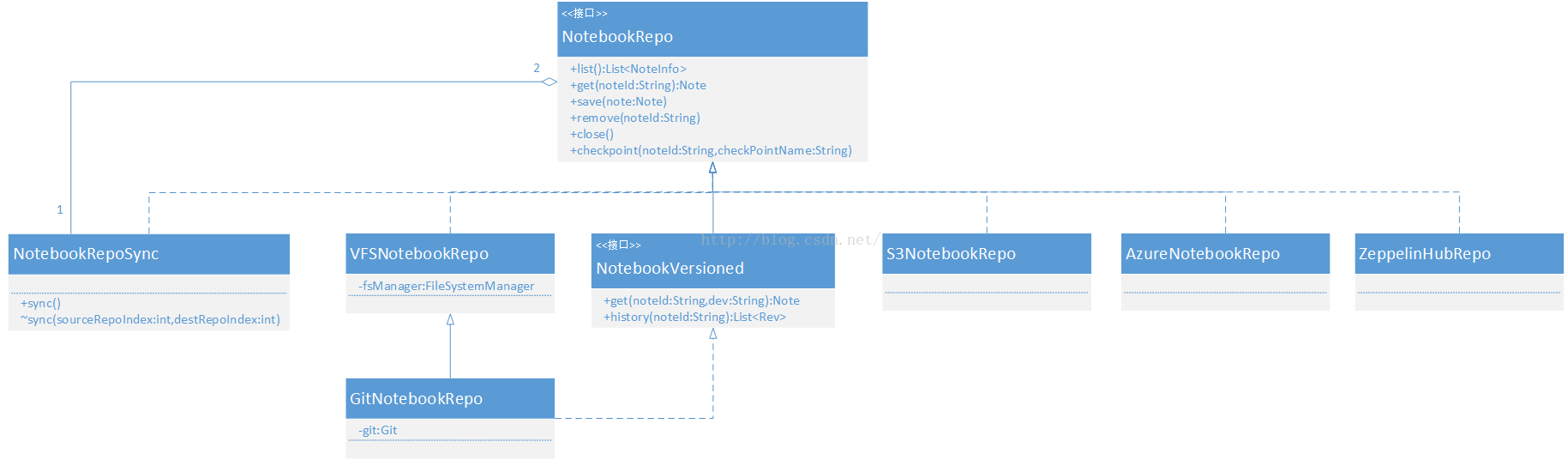


## 2.5 NotebookRepo以及Note持久化子系统

Notebook的持久化子系统主要由NotebookRepo以及其子类组成。各类主要的职责如下：

* NotebookRepo是顶层接口，规定了持久化层基本的CRUD接口。
* NotebookVersioned定义了Note的版本管理接口，目前其实现类只有 GitNotebookRepo。

主要的类如下所示：



NotebookRepo的实现有以下几种：

* GitNotebookRepo是以JGit库实现的基于本地文件系统的、支持以Note为粒度进行checkin和show log的Note仓库。
* VFSNotebookRepo是zeppelin的默认实现类（配置参数zeppelin.notebook.storage控制）。使用apache common-vfs来实现多文件系统支持。
* S3NotebookRepo和AzureNotebookRepo，实现向2大云存储系统的持久化Notebook。
* ZeppelinHubRepo是为了向zeppelinhub持久化Notebook而设计的，zeppelinhub是一个类似于Github的分享网站，区别在于Github是分享git仓库的，zeppelinhub是分享note的。

### 2.5.1 Note加载过程

1） VFSNotebookRepo首先查看如下参数指定地址：

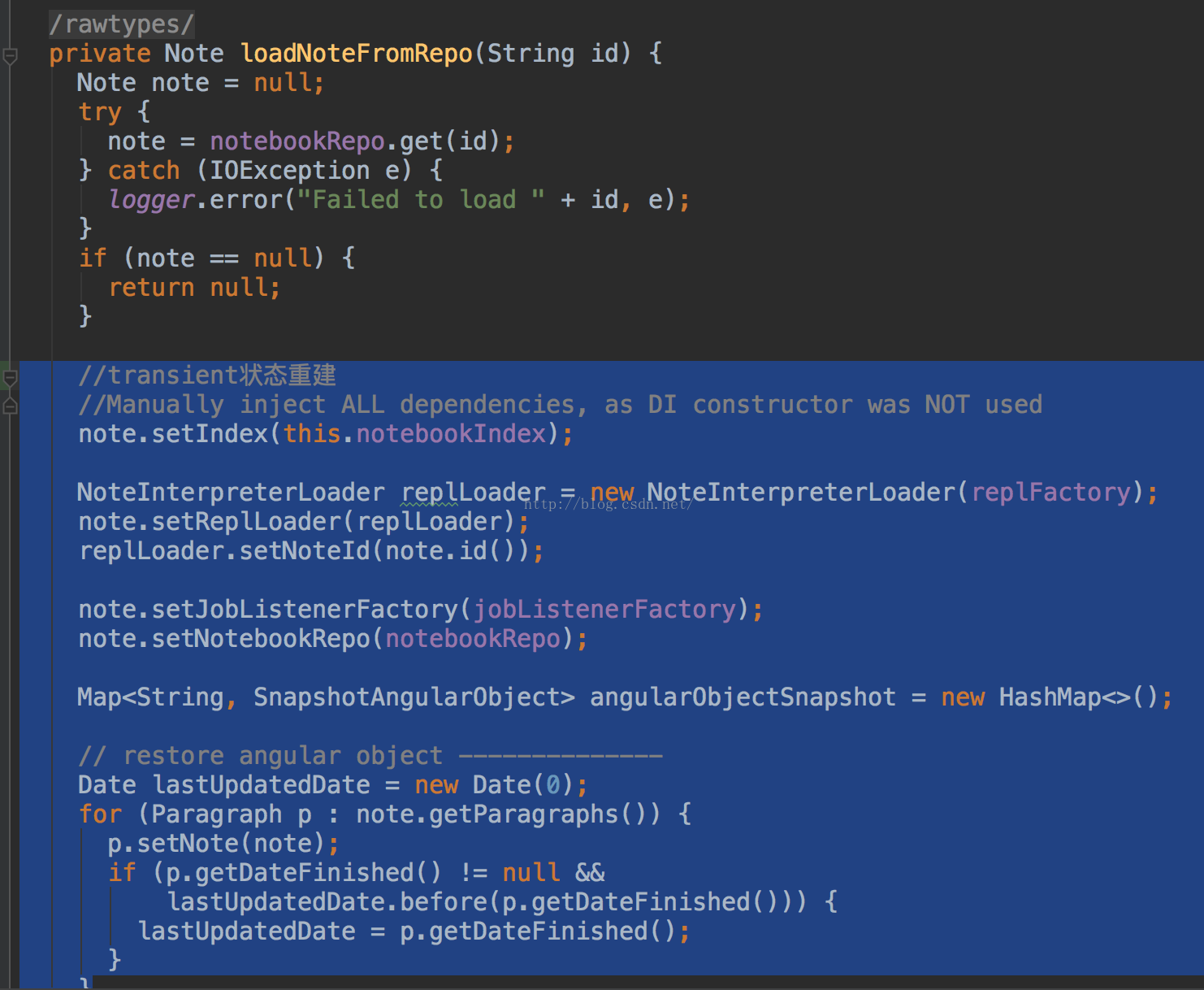
*ZEPPELIN\_NOTEBOOK\_DIR("zeppelin.notebook.dir", "notebook")*

默认相对路径notebook，该路径会被解释成本地文件路径

2）在该路径下每个子目录会被存储了一个名为note.json的文件，目录名字被解释成note的id。

3）使用Gson将该Note.json文件反序列化成Note实例

4）重建Note和相关的transient状态，Notebook.loadNoteFromRepo(String id)方法，如下：



如下是zeppelin自带的notebook中的demo note，经过VFSNotebookRepo加载之后，会产生2个Note实例，id分布为2A94M5J1Z和r

*notebook/*

*├── 2A94M5J1Z*

*│   └── note.json*

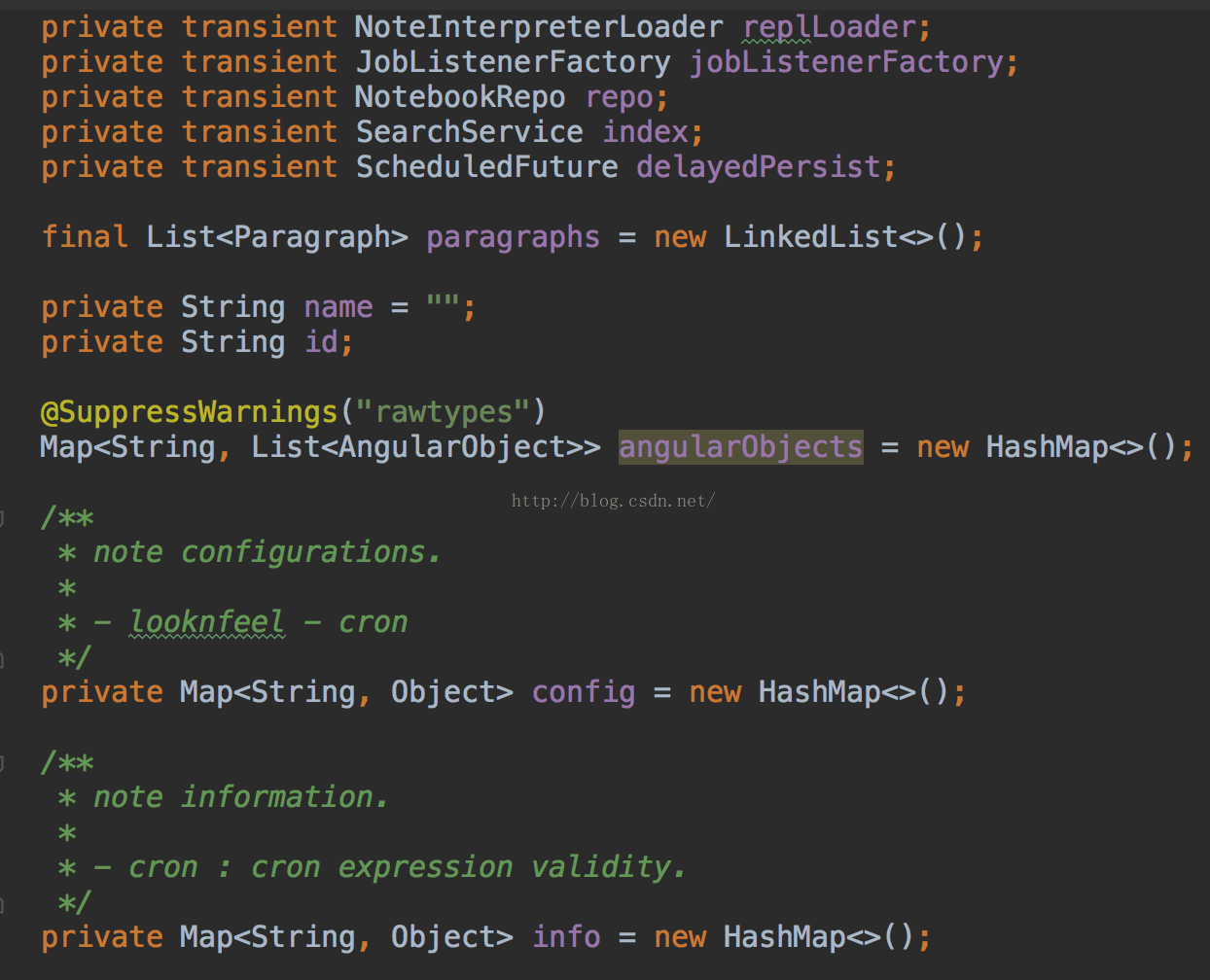
*└── r*

*└── note.json*

Note.json的序列化json后的内容：



通过对比Note字段可以发现，以上json实际上是与Note的非transient字段一一对应。



由于Note实例化成json，完成图的序列化，实际上会将Paragrah,Job,GUI config,ParagraphConfig。

## 2.6 NotebookRepoSync

NotebookRepoSync的初衷是为了让2个NotebookRepo之间进行自动同步修改，实现：在本地repo保存修改的同时，让zeppelin自动将修改同步到远程的repo上。

要启用2个repo之间的同步，做如下修改：

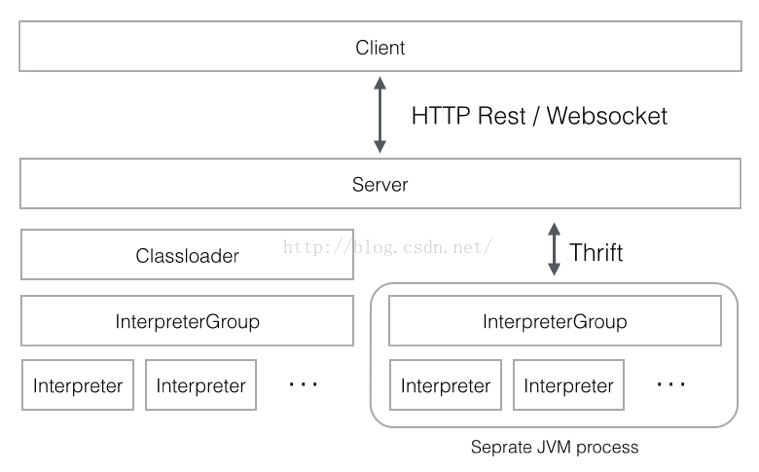
* 在zeppelin-site.xml中修改配置参数zeppelin.notebook.storage，以逗号分隔2个实现类的完整类名
* 注意顺序，一般是将VFSNotebookRepo作为一个，而S3NotebookRepo或者是AzureNotebookRepo等作为第二个。zeppelin目前只支持最大2个Repo（maxRepoNum=2作为编译时常量），不能通过配置修改。

## 2.7 SearchService以及Note全文检索子系统

Note的全文检索子系统由SearchService接口和其子类组成，目前只有一个实现类LuceneSearch，采用基于内存的索引，在Notebook每次重新加载所有的Note、或者Note有更新、删除的时候会重新索引该Note，保证全文索引与持久化的Note文件一致性。

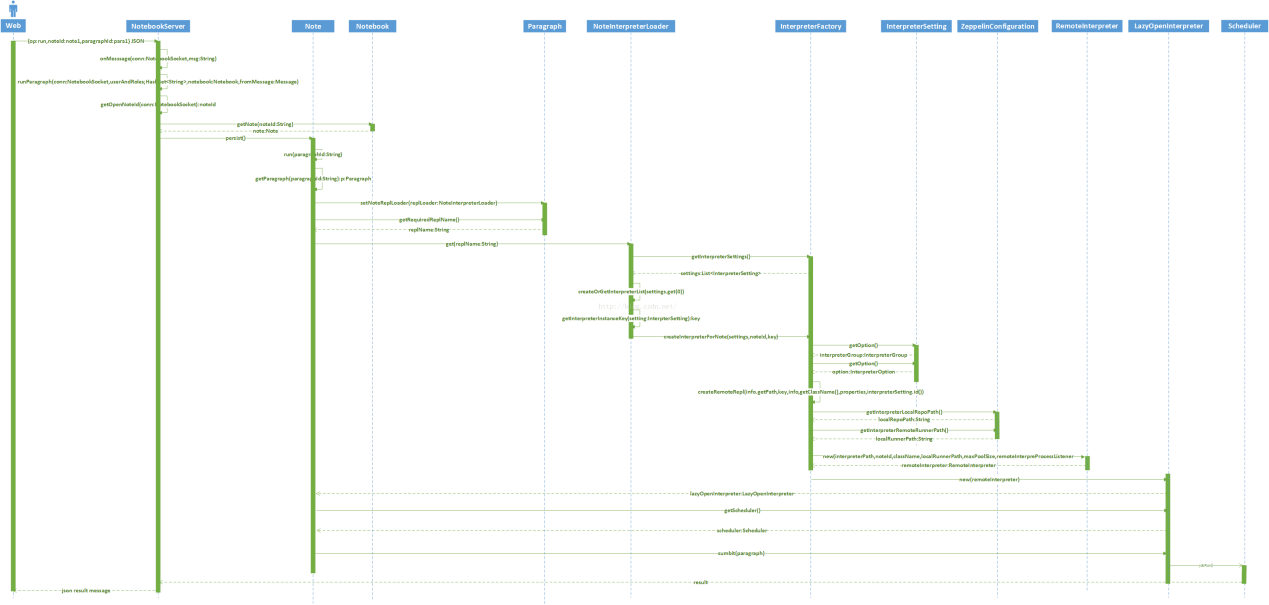
# 3.Note的执行过程

Zeppelin的前后台交互模型如下所示：



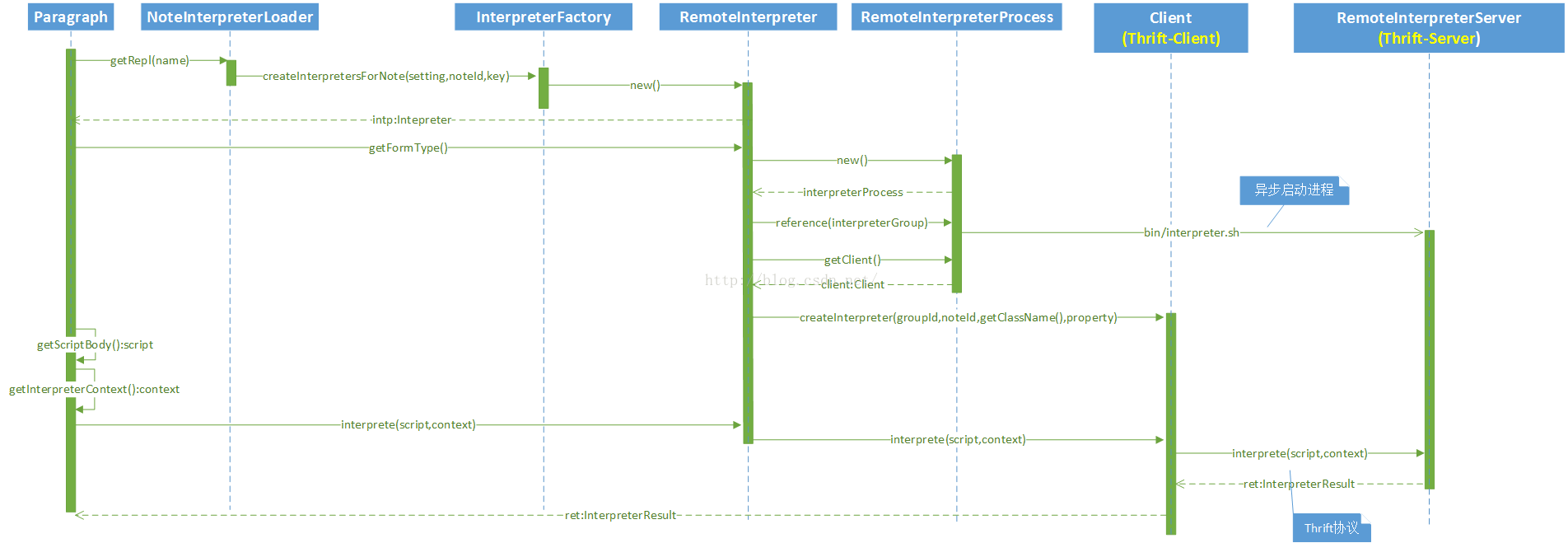
Zeppelin采用单独的JVM来启动interpreter进程，该进程与ZeppelinServer进程之间采用Thrift协议通信，其中RemoteInterpreterProcess是Thrift-Client端，而相应的RemoteInterpreterServer是ThriftServer端。

Paragraph的执行分成从前端UI提交ParagraphJob到其相关Interpreter的Scheduler和Scheduler执行的2个部分，这两个部分是异步执行：



上图是从前台请求执行指定Note到指定的Paragraph开始，到该Paragraph提交到Scheduler之间的时序图。

下面是Scheduler执行该ParagraphJob的时序图：



1. InterpreterFactory目前将所有的InterPreter都实例化成RemoteInterpreter，

参见createInterpreterForNote方法

*for (String intName : keys) {*

*RegisteredInterpreter info = Interpreter.*registeredInterpreters*.get(intName);*

*if (info.getClassName().equals(className)*

*&& info.getGroup().equals(groupName)) {*

*Interpreter intp;*

*if (option.isRemote()) {//根据option配置来创建*

*intp = createRemoteRepl(info.getPath(),*

*key,*

*info.getClassName(),*

*properties,*

*interpreterSetting.id());*

*} else {*

*intp = createRepl(info.getPath(),*

*info.getClassName(),*

*properties);*

*}*

2) RemoteInterpreterProcess在reference相关的InterPreterGroup的时候，会使用apache commo

n-exec框架创建新的进程

*public int reference(InterpreterGroup interpreterGroup) {*

*synchronized (referenceCount) {*

*if (executor == null) {*

*// start server process*

*try {*

*port = RemoteInterpreterUtils.*findRandomAvailablePortOnAllLocalInterfaces*();//随机可用端口*

*} catch (IOException e1) {*

*throw new InterpreterException(e1);*

*}*

*CommandLine cmdLine = CommandLine.*parse*(interpreterRunner);*

*cmdLine.addArgument("-d", false);*

*cmdLine.addArgument(interpreterDir, false);*

*cmdLine.addArgument("-p", false);*

*cmdLine.addArgument(Integer.*toString*(port), false);*

*cmdLine.addArgument("-l", false);*

*cmdLine.addArgument(localRepoDir, false);*

*executor = new DefaultExecutor();*

*watchdog = new ExecuteWatchdog(ExecuteWatchdog.*INFINITE\_TIMEOUT*);*

*executor.setWatchdog(watchdog);*

*running = true;*

*try {*

*Map procEnv = EnvironmentUtils.*getProcEnvironment*();*

*procEnv.putAll(env);*

logger*.info("Run interpreter process {}", cmdLine);*

*executor.execute(cmdLine, procEnv, this);//启动新的进程*

*} catch (IOException e) {*

*running = false;*

*throw new InterpreterException(e);*

*}*

其中interpreterRunner会指向bin/interpreter.sh脚本，该脚本的主要功能根据是否定义了SPARK\_HOME环境变量（定位到[Spark](http://lib.csdn.net/base/spark)-submit脚本），构建classpath，然后以指定的port运行ZEPPELIN\_SERVER指定的主类，该变量被定义为：

*ZEPPELIN\_SERVER=org.apache.zeppelin.interpreter.remote.RemoteInterpreterServer*

1. Remote含义是另外一个进程，与ZeppelinServer所在进程不是同一个，该进程并非启动在另外一个独立的机器上，所有的InterPreter JVM都启动在localhost上。

<http://blog.csdn.net/column/details/15939.html>