Table of Contents

Spark运维实战

本书参考Spark官方文档和源码，通过本书你将精通Spark的安装、配置、监控和调优。

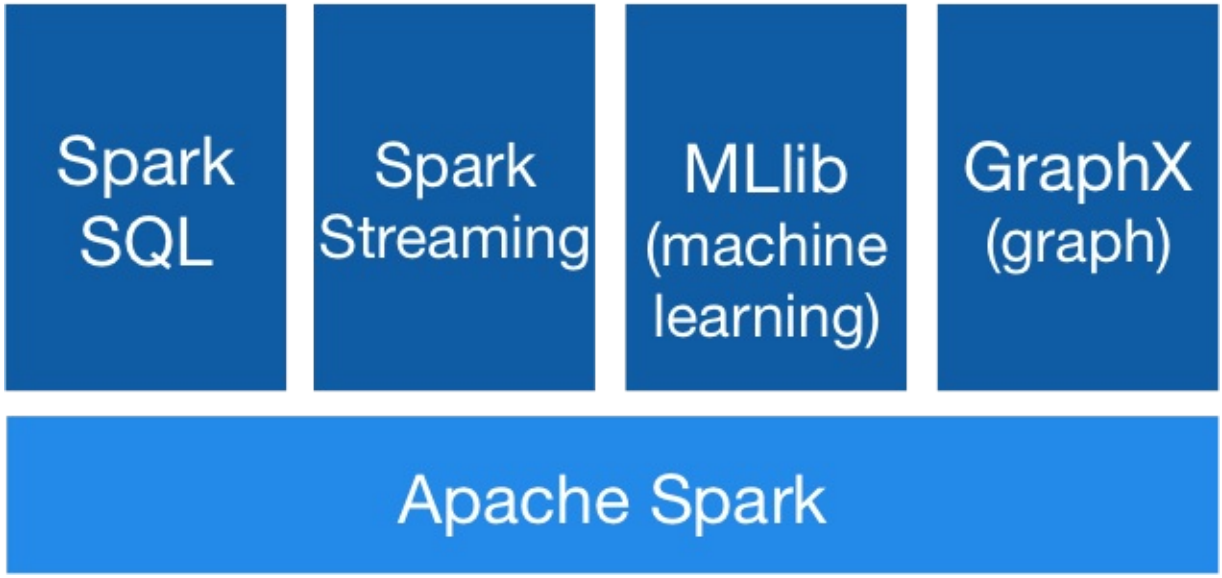
Apache Spark

Spark是伯克利APMLab实验室精心打造的，力图在算法（Algorithms）、机器（Machines）、人（People）之间通过大规模集成，来展现大数据应用的一个平台，其核心引擎就是Spark，其计算基础是弹性分布式数据集，也就是RDD。通过Spark，AMPLab运用大数据、云计算、通信等各种资源，以及各种灵活的技术方案，对海量不透明的数据进行甄别并转化为有用的信息，以供人们更好的理解世界。Spark已经涉及到机器学习、数据挖掘、数据库、信息检索、自然语言处理和语音识别等多个领域。

Sparp ecological environment

随着spark的日趋完善，Spark以其优异的性能正逐渐成为下一个业界和学术界的开源大数据处理平台。随着Spark1.1.0的发布和Spark生态圈的不断扩大，可以预见在今后的一段时间内，Spark将越来越火热。

Spark生态圈以Spark为核心引擎，以HDFS、S3、Techyon为持久层读写原生数据，以Mesos、YARN和自身携带的Standalone作为资源管理器调度job，来完成spark应用程序的计算；而这些spark应用程序可以来源于不同的组件，如Spark的批处理应用、SparkStreaming的实时处理应用、Spark SQL的即席查询、BlinkDB的权衡查询、MLlib或MLbase的机器学习、GraphX的图处理等等。更多的新信息请参看伯克利APMLab实验室的项目进展https://amplab.cs.berkeley.edu/projects/或者 Spark峰会信息http://spark-summit.org/。



Spark

Spark是一个快速的通用大规模数据处理系统，和Hadoop MapRedeuce相比：

更好的容错性和内存计算 高速，在内存中运算100倍速度于MapReduce 易用，相同的应用程序代码量要比MapReduce少2-5倍 提供了丰富的API 支持互动和迭代程序

Spark大数据平台之所以能日渐红火，得益于Spark内核架构的优秀：

- 提供了支持DAG图的分布式并行计算框架，减少多次计算之间中间结果IO开销
- 提供Cache机制来支持多次迭代计算或者数据共享，减少IO开销 *
- RDD之间维护了血统关系，一旦RDD fail掉了，能通过父RDD自动重建，保证了容错性
- ，RDD Partition可以就近读取分布式文件系统中的数据块到各个节点内存中进行计算
- 使用多线程池模型来减少task启动开稍
- shuffle过程中避免不必要的sort操作
- 采用容错的、高可伸缩性的akka作为通讯框架
- 。。。

SparkStreaming

SparkStreaming是一个对实时数据流进行高通量、容错处理的流式处理系统，可以对多种数据源（如Kdfka、Flume、Twitter、Zero和TCP 套接字）进行类似map、reduce、join、window等复杂操作，并将结果保存到外部文件系统、数据库

或应用到实时仪表盘。

SparkStreaming流式处理系统特点有：

- 将流式计算分解成一系列短小的批处理作业
- 将失败或者执行较慢的任务在其它节点上并行执行
- 较强的容错能力(基于RDD继承关系Lineage)
- 使用和RDD一样的语义

Spark SQL

Spark SQL是一个即席查询系统，可以通过SQL表达式、HiveQL或者Scala DSL在Spark上执行查询。

Spark SQL的特点:

- 引入了新的RDD类型SchemaRDD，可以象传统数据库定义表一样来定义SchemaRDD，SchemaRDD由定义了列数据类型的行对象构成。
- SchemaRDD可以从RDD转换过来，也可以从Parquet文件读入，也可以使用HiveQL从Hive中获取。
- 在应用程序中可以混合使用不同来源的数据，如可以将来自HiveQL的数据和来自SQL的数据进行join操作。
- 内嵌catalyst优化器对用户查询语句进行自动优化

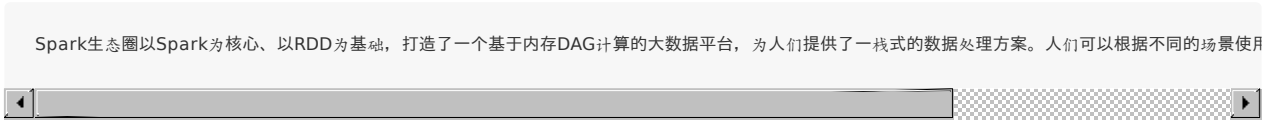
MLlib

MLlib是Spark实现一些常见的机器学习算法和实用程序，包括分类，回归，聚类，协同过滤，降维，以及底层

GraphX

GraphX是基于Spark的图处理和图并行计算API。GraphX定义了一个新的概念：弹性分布式属性图，一个每个顶点和边都带有属性的定向多重图；并引入了三种核心RDD：Vertices、Edges、Triplets；还开放了一组基本操作（如subgraph, joinVertices, and mapReduceTriplets），并且在不断的扩展图形算法和图形构建工具来简化图分析工作。

生态圈的应用



主要应用场景：

用户画像的建立　　用户异常行为的发现　　社交网络关系洞察 用户定向商品、活动推荐

spark 运维相关

安装配置、监控等，请求参考《Spark 运维实战》

graphite

yum install -y bitmap bitmap-fonts-compat Django django-tagging fontconfig cairo python-devel python-memcached python-twisted pycairo mod_python python-ldap python-simplejson memcached python-zope-interface mod_wsgi python-sqlite2

Spark Base

Spark 开发环境

Spark本身是由scala语言开发的，提供了三种语言接口：Scala、Java、Python。根据自己的喜好可以使用相应语言的开发工具。

本书使用scala语言做为开发Spark应用的语言，采用Eclipse为主要的开发工具。

主要介绍了两个流行的开发工具：Eclipse、Intellij IDEA。

JDK安装配置

下载

官方网址：http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html

选择好操作系统版本，32位操作系统选择带i586的安装文件；64位操作系统选择带x64的安装文件。

Linux操作系统推荐下载tar.gz格式的安装文件，Window当然也只有exe格式的文件。

Linux下安装

解压

```
tar -zxvf jdk-7u9-linux-i586.tar.gz -C /opt/
ln - /opt/jdk1.7.0_09 /opt/jdk
```

设置环境变量

用vi编辑配置文件：vi /etc/profile

```
export JAVA_HOME=/opt/jdk
export CLASSPATH=$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
export PATH=$JAVA_HOME/bin:$PATH
```

保存退出按Esc然后输入:wq

使配置生效:

```
source /etc/profile
```

Windows下安装

选择好操作系统版本是32还是64，

解压

双击进行安装一路下一步，便可安装成功。

设置环境变量

测试是否成功

命令行输入：

```
java -version
```

如果出现下面提示说明成功：

```
java version "1.7.0_05"
Java(TM) SE Runtime Environment (build 1.7.0_05-b06)
Java HotSpot(TM) Client VM (build 23.1-b03, mixed mode, sharing)
```

Scala安装配置

安装

下载

到http://scala-lang.org 下载scala程序

wget http://scala-lang.org/files/archive/scala-2.10.4.tgz

解压

```
tar -zxvf scala-2.10.4.tgz -C /opt
ln -s /opt/scala-2.10.4 /opt/scala
```

配置环境变量

vi /etc/profile

```
#SCALA setting
export SCALA_HOME=/opt/scala
export PATH=$SCALA_HOME/bin:$PATH
```

使用Scala解释器

Read-Eval-Print-Loop， REPL， 读取－计算－打印－循环

在控制台输入scala进入scala解释器，

可以使用tab进行提示；

使用上、下箭头查看历史命令，然后使用左右箭头和DEL进行修改；

REPL中以行为单位，如果想写多行代码可以输入:paste，然后粘贴代码，最后按Ctrl + D执行。

使用sbt创建scala项目

SBT安装使用

SBT支持Windows和Linux、Mac等操作系统

下载

官方网站：http://www.scala-sbt.org/

wget https://dl.bintray.com/sbt/native-packages/sbt/0.13.6/sbt-0.13.6.tgz

安装

解压下载包，把sbt/bin放到环境变量PATH下便可。

通常先添加环境变量SBT_HOME,再把SBT_HOME/bin添加到PATH中

windows：

假设把sbt-0.13.6.tgz解压到D:\software\中，设置如下环境变量：

```
SBT_HOME=D:\software\sbt
PATH=%SBT_HOME%\bin;JAVA_HOME\bin
```

注：其中JAVA_HOME\bin为原来配置

Linux：

假设把sbt-0.13.6.tgz解压到/opt/中:

```
tar -zxvf sbt-0.13.6.tgz -C /opt
```

设置如下环境变量 vi /etc/profile：

```
export SBT_HOME=/opt/sbt
export PATH=$SBT_HOME%/in;$PATH
```

配置插件

找到文件夹~/.sbt/0.13/plugins下面的plugins.sbt配置文件，如果没有相应文件夹和文件则创建。添加插件：

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "2.5.0")

addSbtPlugin("com.github.mpeltonen" % "sbt-idea" % "1.6.0")
```

注：之间必须有空行，SBT语法要求。第一行用来生成eclipse项目配置文件，第三行用来生成Intellij IDEA的项目配置文件。

配置代理仓库

创建scala项目

项目目录结构：

```
├── src
│   ├── main
│   │   ├── java
│   │   ├── resources
│   │   └── scala
├── build.sbt
├── project
│   ├── build.properties
│   ├── plugins.sbt
```

SBT使用的目录结构和MAVEN类似，在src/main/scala下编写scala代码，在src/marin/resources下编写配置文件。

build.sbt

设置项目名称、版本、依赖，内容如下：

```
name := "spark"

version := "1.0"

scalaVersion := "2.10.4"

libraryDependencies += "org.apache.spark" % "spark-core_2.10" % "1.1.0"
```

注：空行不能省略

project/build.properties

设置SBT的版本：

```
sbt.version=0.13.6
```

project/plugins.sbt

设置插件：

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "2.5.0")
```

如果在SBT安装时已经配置了sbteclipse插件，此处可以不写sbteclipse-plugin插件。

生成IDE配置

如果你使用的是Eclipse：

```
sbt eclipse
```

如果你使用的是Intellij IDEA项目

```
sbt idea
```

使用maven创建Java项目

Maven安装配置

创建Java项目

在命令行执行：

```
mvn archetype:create -DgroupId=com.hansight.logger  -DartifactId=logger-spark
```

其中，“com.hansight.logger”是公司产品名，“logger-spark”是项目名。当然，也可以在Eclipse里面创建项目。

在创建完成修改pom.xml添加Spark依赖：

使用Eclipse开发Spark应用

下载

```
官方网址：www.eclipse.org
```

如果使用Scala语言开发Spark应用，最好下载kepler也就是4.3，因为此时最新的Eclipse Scala插件不支持Eclipse的4.3之后的版本。

安装

安装JDK

参考JDK安装配置

安装Scala

参考Scala安装配置

安装Eclipse

解压后直接可用

安装Eclipse的Scala插件

在Eclipse中的Help菜单下，点击Install New Softwar，在弹出的窗口中的Work with输入框里输入下面的链接：

```
http://download.scala-ide.org/sdk/helium/e38/scala210/stable/site
```

选中Scala IDE for Eclipse，然后点击“Next”进行安装

注意：Spark1.1.0使用的Scala是基于2.10版本的。

创建项目

scala语言

参考使用sbt创建scala项目

导入Scala项目

打开Eclipse，点击菜单File，然后选择Import。

在弹出的Import窗口选择General/Existing Projects into Workspace。

选择“Next”进入下个窗口，点击Select root directory右边的Browse选择上面生成的Scala项目。

然后选择“finish”导入完成。

注：以上也是导入普通eclipse项目的方法，sbt eclipse把项目转换成了普通的scala项目。

基于Maven创建java项目

参考使用maven创建Java项目

导入Maven项目

打开Eclipse，点击菜单File，然后选择Import。

在弹出的Import窗口选择Maven/Existing Maven Projects。

选择“Next”进入下个窗口，点击Select root directory右边的Browse选择上面生成的Scala项目。

然后选择“finish”导入完成。

使用Intellij IDEA开发Spark应用

下载

官方网址：www.jetbrains.com/idea/

安装

安装JDK

参考JDK安装配置

安装Scala

参考Scala安装配置

安装Intellij IDEA

一路下一步便可

安装Eclipse的Scala插件

在Eclipse中的Help菜单下，点击Install New Softwar，在弹出的窗口中的Work with输入框里输入下面的链接：

http://download.scala-ide.org/sdk/helium/e38/scala210/stable/site

选中Scala IDE for Eclipse，然后点击“Next”进行安装

注意：Spark1.1.0使用的Scala是基于2.10版本的。

创建项目

基于SBT创建scala项目

参考使用sbt创建scala项目

导入Scala项目

基于Maven创建java项目

参考使用maven创建Java项目

导入Maven项目

Spark Operation

Spark Context

SparkContext实例与一个Spark集群连接，并提供与Spark交互的接口，提交作业。

通过SparkContext实例可以创建RDD，广播变量、计数器。

使用SparkConf

用来配置Spark相关属性

```
val conf = new SparkConf().setAppName("LogParser")
```

注：

- setAppName 设置Spark应用的名称，在查看时显示。
- setMaster 设置Spark集群的Master地址，通常不设置在进行运行时由spark-submit工具进行设置。本地调试时可以设置为“local[2]”，其中“[2]”为设置的线程个数。

创建SparkContext

```
val sc = new SparkContext(conf)
```

addFile(path:String)与clearFiles

addFile方法会添加一个本地文件，然后可以通过SparkFiles.get()来获取文件的URI。

addFile把添加的本地文件传送给所有的Worker，这样能够保证在每个Worker上正确访问到文件。另外，Worker会把文件放在临时目录下。因此，比较适合用于文件比较小，计算比较复杂的场景。如果文件比较大，网络传送的消耗时间也会增长。

path：可以是local、hdfs（任何hadoop支持的文件系统）、HTTP、HTTPS、FTP等。local方式时，在windows下使用绝对路径时需要加个“/”，如“d:/spam.data”得写成“/d:/spam.data”或“file:///d:/spam.data”。

通过SparkFiles.get(path:String)获取添加的文件路径。

```
通过clearFiles来删除addFile添加的文件。
```

样例：

```
var path = "/D:/spam.data"
sc.addFile(path)
val rdd = sc.textFile(SparkFiles.get(path));
sc.clearFiles;
```

addJar(path:String)与clearJars

addJar(path:String)添加在这个SparkContext实例运行的作业所依赖的jar。

clearJars删除addJar添加的jar。

RDD

RDD（Resilient Distributed Datasets，弹性分布式数据集），能够数据容错和被并行处理。

创建方式

有两种：并行化已经存在的容器、指向外部文件系统的数据集。

并行化容器

使用SparkContexxt的parallelize方法把Scala容器转为RDD。例如：

```
val rdd = sc.parallelize(List(1, 2, 3), 3)
```

第一个参数为：scala.Seq类型

第二个参数，可选，设置RDD的分区partition数量。如果没有使用，Spark会根据集群的大小自动设置相对合理的分区数量。过大或过小的分区数，会造成性能

延迟执行

parallelize是延迟执行的，也就是说在调用parallelize后，并且在第一次对parallelize创建的RDD调用action操作之前，

外部数据集

Spark支持的外部数据集包括：本地文件系统、HDFS、HBase、Hadoop InputFormat。

本地文件系统

HDFS

HBase

#

Transformation

Action

Key-Value Pairs RDD

Scala

base grammer

1、常量和变量

val声明出来的为常量，不能再次赋值；可以省略类型，scala会自动推导。

var声明出来的为变量，可以再次赋值；可以省略类型，scala会自动推导。

var a = "xxx"; 等同于 var a : String = "xxx";

a = "String 2";

val b = 2;

b = 3; // 会出错

声明多个变量：

val a, b = "xxx"; // a, b的值都是 "xxx"

2、scala的常用数据类型：

Byte：内置转换为scala.runtime.RichByte

Char：内置转换为scala.runtime.RichChar

Short：内置转换为scala.runtime.RichShort

Int：内置转换为scala.runtime.RichInt

Long：内置转换为scala.runtime.RichLong

Float：内置转换为scala.runtime.RichFloat

Double：内置转换为scala.runtime.RichDouble

Boolean：内置转换为scala.runtime.RichBoolean

这些类型都是类，Scala并不区分基本类型与引用类型。

与java基本类型相对应，但拥有更多的方法，因为会转换为相应的RichXxxx，在转换后的类里添加scala语言层面的方法。

例如：

1.toLong，其中1会被转换为RichInt类型的实例。

1.to(5)会产生Range(1, 2, 3, 4, 5)

任意有穷数字：

BigInt：

BigDecimal：

字符串：

String：对应于scala.collection.immutable.StringOps

3、类型转换

使用方法，没有强制转换一说。

如：var a : Int = 1

var b : Long = a.toLong

4、操作符重载

val value = 8 + 5

其中，+ 这些操作是方法。

完整的是val value = (8).+(5)；

(8)转换成Int对象，调用Int对象的+方法（方法名为+）。

没有++，--操作符，因为相应类型是不可变的。

5、Scala方法调用的简略写法

方法的简略写法：

a 方法 b

完整写

a.方法(b)

例：1.to(2)，可简略写为1 to 2

apply方法：

a(b)

完整为

a.apply(b)

例： "xxx"(2) 实际转换为 "xxx".apply(2)

常用来构造对象

方法作为参数：

匿名方法：

x, y => x + y 或

+

x, y 表示为参数；=>指向方法体；x + y计算和并返回。

+ 中，第一个“”表示第一个参数，第二个“”表示第二个参数。

例："sU".count(_.isUpper)

```
"sU".count(x => x.isUpper)
```

6、import

import math._

作用是导入scala.math包下所有类和对象。以scala.开头的包可以省略。

7、单例对象object

object定义的对象为单例，可以直接使用 对象名.方法(参数) 进行调用， 类似java中的static方法。

伴生对象：拥有与class类同名的object定义的对象。

8、Scala doc

http://scala-lang.org/api

数学函数位于scala.math包中，而不是位于某个类中；

使用数值类型，使用RichInt、RichDouble等；

使用字符串，使用StringOps；

控制结构与函数

在java语言中，表达式表示值，语句表示执行动作。 例如：表达式1+1表示值；thread.start()表示执行动作。

所有语法结构都有值，那怕是不存在用Unit类型.

if/else语句

有值，为跟在if或else之后的表达式的值。

例：val s = if(x > 0) 1 else 0;

等同于var s:Int = 0; if (x > 0) s = 1 else s = 0;

val s2 = if (x > 0) "string" else 1;

s2的类型为Any

val s3 = if (x > 0) 1;

s3的类型也是Any，但x <= 0时s3的值为Unit， Unit类表示“无有用值”，写做()

没有switch语句

语句终止

一行一条语句，行尾不需要分号

一行多条语句，需要使用分号把它们隔开

较长语句分两行时，在第一行尾部不要用语句结束的符号结束，通常使用一个操作符。如：

val s = "sssssssssssssssssssssssssss" + "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb" + //告诉解析器这里不是语句的结尾

"ccc" + "dd"

块表达式和赋值

{}块包含一系列表达式，其结果是块中最后一个表达式的值。

例：val s = {1+2; 2+3} //s的值为5

注意：赋值语句的返回值是Unit类型，写作()

例：val u = {b = 3}

x = y = 1，结果为x的值Unit，y的值1。因为y = 1语句的值为Unit；

输入与输出

print()

println() 打印完后追加一个换行符

printf()具有c风格， %s , %d, %f,

输入：

readLine("tips")：从控制台读取一行，"tips"为输出提示

readDouble, readFloat, readByte, readChar readShort, readInt, readLong, readBoolean

循环

while循环

```
while (n > 0){
    // TODO
}
```

for循环

```
for (i <- 1 to n) {
    // TODO
}
```

for的参数表达式说明：让变量i遍历<-右边表达式的所有值。

通常在遍历时使用util，until返回一个不包括上限的区间。例：0 until 2，表示Range(0, 1)

注：scala没有类似java的for(;;)型for语句

for高级特性：

循环体带yield，即for推导：

```
val z = for (i <- 0 until 3) yield i;
```

z的值为Vector(0, 1, 2)

多个生成器、守卫、变量定义：

```
for (i <- 0 until 5; j <- 3 until ;) { println(i, j)} //多个生成器
for (i <- 0 until 5; from = 4 - i; j <- from until 5) {} //变量定义
for (i <- 0 until 5; from = 4 - i; j <- from until 5 if i != j) {} //每个生成器可以带一个守卫，注意if前没有分号
```

没有break和continue

需要时怎么办？

1、使用变量控制

2、外层嵌套函数，使用return进行break;

3、使用异常机制，抛出异常与捕获异常控制。

scala提供的：

```
import scala.util.control.Breaks.break
import scala.util.control.Breaks.breakable

breakable {
  for (i <- 0 until 2) {
    println(i)
    if (i == 1) break;
  }
}
```

函数

定义函数只需要给出函数名、参数和函数体

def 函数名[(参数名:参数类型，…)][: 返回类型]= {

    // function body

}

注：[]表示可选，只要函数不是递归的就不需要指定函数的返回类型；scala会自动推断函数的返回类型。

函数体最后一个表达式的值就是函数的返回值，当然也可以用return显式返回。

如果函数没有参数，括号可以省略，但调用时也不能带括号。

默认参数和带名参数

```
def test(str: String, left: String = "[", right: String = "]") = left + str + right
test("hello") // 返回[hello]
test("hello", right = ">>")
test("hello", left = "<", right = ">")
test("hello", right = ">")
```

变长参数：

```
def sum(args: Int*) = {
 var result: Int = 0
 for (arg <- args) result += arg
 result
}
```

可以使用合意多的参数来调用该函数

sum(1, 2, 3)

如果有了一个值的序列，不能直接调用，得转化

    val s = sum(1 to 10: *) // *将1 to 10当作参数序列处理。

    val s = sum(1 to 10) // 会报错

此语法的用处是递归

```
def recursiveSum(args: Int*): Int = {
 if (args.length == 0) 0
 else args.head + recursiveSum(args.tail: _*)
}
```

args.head表示args的首个元素，而tail表示所有其它的元素。

过程：

没有返回值的函数，函数体前不带“=”。

lazy:

lazy val words = Source.fromFile("readme2.txt").mkString

函数第一次调用时加载

也有额外开销，每次访问懒值，都会调用一个方法以线程安全的方式检查值是否已经初始化

异常

方法不用带类似throw IOException的签名

try{

} catch {

case *: IOException => println("xx") //* 不需要使用异常对象，可以使用来代替异常名。

case e: Exception => e.printStatckTrace()

} finally {

}

array

定长数组：

val nums = new Array[Int](10)

变长数组：

val b = new ArrayBuffer[Int] 或 ArrayBuffer[Int]()

b.+=1

b.+= (2, 3, 4)

b.++= Array(5, 6, 7, 8)

b.trimEnd(3)

b.insert(2, 3)

b.insert(2, 3, 4, 5)

b.remove(2)

b.remove(2, 4)

b.toArray

遍历

```
for (i <- 0 until b.length)
    println(b(i))
for (i <- (0 until b.lenght).reverse)
    println(b(i))
for (element <- b)
    println(element)
```

数组转换：

val a = Array(1, 2, 3, 4)

val b = for (e <- a) yield e * 2

val b = a.map(_ * 2)

val c = for (e <- a if e % 2 == 0) yield 2*e

val c = a.filter(*%2==0).map(2**)

常用方法：

sum：必须是数值型类型

min

max

sorted：

    scala.util.Sorting.quickSort(a)

mkString

toString

多维数组：

创建维度不同的二维数组：

var arr = new ArrayArray[Int]; arr(0) = Array(1, 3) arr(2) = new ArrayInt

创建维度相同的数组：

var matrix = Array.ofDimInt //二行，五列

matrix(0)(1) = 2 // 访问元素使用两个圆括号

与Java互操作：

Map和Tuple

Map

构造Map

不可变：

val map = Map("sa" -> 1, "s" -> 2) map("sa") = 3 // error

val emptyMap = new scala.collection.immutable.HashMap[String, Int]

可变：

val map2 = scala.collection.mutable.Map("sa" -> 2) map2("sa") = 3

val emptyMap = new scala.collection.mutable.HashMap[String, Int]

注：->用来创建元组， "sa" -> 1即("sa", 1)

初始化完全可以 val map = Map(("sa", 1), ("s", 2))

获取Map中的值：

如果map中不包含请求中使用的key值，则抛异常。NoSuchElementException

```
map("sa") // 类似于java中的map.get("sa")
```

要检查map中是否包含某个key，使用contains方法。

```
val sa = if (map2.contains("sa3")) map2("sa3") else 0;
```

快捷的方式：

```
val sa2 = map.getOrElse("sa2", 0)
```

一次得到是否包含key，并获取值：

```
val sa3 = map.get("sa3"); // Option类型，        println(sa3.isEmpty)
```

更新Map中的值：

添加或更新：

```
map("sa") = 3
```

添加或更新多个：

```
map += ("aa" -> 4, "bb" -> 5)
```

移除某个key和对应的值：

```
map -= "aa"
```

不可变的map也可以使用+和-操作，但是会生成新的map

```
var map = Map("aa" -> 1)
```

map = map + ("bb" -> 2)

map += ("cc" -> 2)

map -= "aa"

迭代**map**：

```
for ((k, v) <- map) {
    // TODO
}
```

所有key：

map.keySet

所有值：

map.values

反转：

map2 = for((k, v) <- map) yield (v, k)

已排序**Map**：

按key排序：

SortedMap

按添加顺序：

LinkedHashMap

Map与**Java**互操作：

Java Properties转为scala.collection.Map：

import scala.collection.JavaConversions.propertiesAsScalaMap val prop: scala.collection.Map[String, String] = System.getProperties();

Java Map转为scala.collection.mutable.Map[String, Int]：

import scala.collection.JavaConversions.mapAsScalaMap val map: scala.collection.mutable.Map[String, Int] = new TreeMap[String, Int]

Scala Map转为Java Map:

import scala.collection.JavaConversions.mapAsJavaMap import java.awt.font.TextAttribute._ var fs = Map(FAMILY -> "Serif", SIZE -> 12) var fonts = new Font(fs)

元组**Tuple**：

不同类型值的集合

val tp = (1, "ss", 2.0)

访问数值：

tp._1

tp._2

tp._3

下标从1开始

通过模式匹配获取元组：

val (first, second, third) = set

可以用于函数返回多个值的时候

拉链操作

```
val arrkey = Array(1, 3, 5)
val arrValue = Array("a", "b", "c")
val tupleArr = arrkey.zip(arrValue) // tupleArr为Array((1,a), (3,b), (5,c))
val map = tupleArr.toMap
```

class

简单类和无参方法：

```
1 class Counter {
2   private var value = 0; //必须初始化字段
3   def increment() = value += 1 //方法默认是公有的
4   def current = value
5 }
```

使用：

```
1    val counter1 = new Counter // 或 new Counter()
2    // 类定义方法是带了()，调用时可带，也可不带
3    counter1.increment
4    counter1.increment()
5    println(counter1.current)
6    println(counter1.current()) // error :
```

类定义方法时没带()，调用时也不能带

推荐的风格是：改值的方法带()；取值的方法不带()，定义时不带便可强制取值风格。

Setter和Getter

```
1 class Person {
2   var age = 0;
3 }
```

对于属性age隐式的生成了age()和age_=(age: Int)方法。

如果属性带private，则生成的方法也带private；如果属性不带private，则生成的方法为public。

如果属性为val，则只生成get方法。

如果禁用生成隐式的方法，使用private[this]声明。这样只能访问实例自身的属性，不能访问其它实例的属性，即对象私有成员。

注：不能实现只有Setter，没有Getter的属性。想实现，只能用其它名称了。

显式定义age()和age_=(age: Int)方法：

```
1 class Person {
2   private var _age = 0; // 变成私有并改名
3
4   def age = this._age;
5   def age_=(_age: Int) {
6     this._age = _age;
7   }
8 }
```

生成的java代码实际上生成了四个方法：显式定义的两个，改名后变量隐式生成的（private）

为什么要改名？不改名无法区分方法名和属性名，而且出现方法定义两次错误

Java Bean规范

生成类似Java Bean的setAge和getAge方法，只需要添加一个注解：@BeanProperty

```
1 class Person {
2   @BeanProperty
3   var age = 0;
4 }
```

上面的例子除了隐式生成了age()和age_=(age:Int)方法外，另外生成Java Bean风格的getAge()和setAge(age:Int)方法

主构造方法：

和类定义交织在一起：

- 参数被编译成字体；
- 主构造方法会执行类定义中的所有语句；
- 构造方法中参数不带val也不带var，只有被一个以上方法调用时才会提升为字段。类似于private[this] val效果

```
1 class Person(@BeanProperty var age: Int) {
2 }
```

会生成一个带age:Int参数的构造函数；

- age()和age_=(age:Int)方法
- getAge()和setAge(age:Int)方法

一个类如果没有显式定义主构造方法，会隐式生成一个无参不做任何操作的主构造方法。

辅助构造方法：

辅助构造方法的名称为this;

辅助构造方法每一行必须是调用构造方法（主构造方法或其它辅助构造方法）

```
def this(age:Int) {
    this()
    _age = age
}
```

嵌套类：

内部类与外部类实例相关联

```
1 class Network {
2   class Member(val name: String) {
3     val contacts = new ArrayBuffer[Member]
4   }
5   private val members = new ArrayBuffer[Member]
6   def join(name: String) = {
7     val m = new Member(name)
8     members += m
9     m
10  }
11 }
```

使用：

```
1    val chatter = new Network
2    val myFace = new Network
3    val fred = chatter.join("fred")
4    val wilma = chatter.join("wilma")
5    fred.contacts += wilma
6    val ngy = myFace.join("hongdao");
7    fred.contacts += ngy; // error
```

解析：

val fred和wilma的类型为chatter.Member；ngy的类型为myFace.Memeber；fred.contacts的实际类型为ArrayBuffer[chatter.Member]；　　所以fred.contacts可以添加wilma，不可以添加ngy。

如果不希望这个效果，可以使用两种方式：

方式一：伴生对象

```
1 class Network {
 2   private val members = new ArrayBuffer[Network.Member]
 3   def join(name: String) = {
 4     val m = new Network.Member(name)
 5     members += m
 6     m
 7   }
 8 }
 9 object Network {
10   class Member(val name: String) {
11     val contacts = new ArrayBuffer[Member]
12   }
13 }
```

方式二：类型投影 **Network#Member**

任何Network的Member

```
 1 import scala.collection.mutable.ArrayBuffer
 2
 3 class Network {
 4   class Member(val name: String) {
 5     val contacts = new ArrayBuffer[Network#Member]
 6   }
 7   private val members = new ArrayBuffer[Network#Member]
 8   def join(name: String) = {
 9     val m = new Member(name)
10     members += m
11     m
12   }
13 }
```

内部类访问外部实例：

- 外部类.this
- 外部类实例别名

```
class Network(val name: String) { outter => //外部类实例别名
  class Member(val name: String) {
    val contacts = new ArrayBuffer[Network#Member]
    def out = println(Network.this.name)
    def out2 = println(outter.name)
  }
}
```

对象

单例对象

scala没有静态方法和静态字段。scala使用object实现，object定义了单个实例。

```
1 object Accounts {
2   private var lastNumber = 0;
3   def newUniqueNumber() = { lastNumber += 1; lastNumber }
4 }
```

使用：

```
for (i <- 0 to 10) println(Accounts.newUniqueNumber)
```

对象的构造方法在该对象被第一次使用时调用。

伴生对象

在Java中，常遇到类即有静态属性和方法，又有非静态属性和方法。

Scala中，可以通过类与类同名的“伴生”对象达到相同目的。

类和伴生对象可以互相访问私有属性，它们必须存在同一源文件中。

apply方法

定义和使用object的applay方法，进行创建伴生类实例。

好处：少写了new，在很多场景下用起来方便，看起来清晰。

应用程序对象

每个scala程序必须从一个对象的main方法开始，这个方法的类型为Arra[String] => Unit

扩展App特质，代码放入构造方法体中，

scala -Dscala.time Hello Fred

枚举

scala本身没有枚举类型

标准库里提供Enumeration助手类，可用于产生出枚举。

```
1 object EventType extends Enumeration {
2   val DDOS, DI = Value
3 }
```

枚举类型是EventType.Value

给类型起个别名：只有引入的时候有意义

```
1 object EventType extends Enumeration {
2   type EventType = Value
3   val DDOS, DI = Value
4 }
```

package 和 import

包

作用：管理大型程序中的名称

- 源文件目录和包之间没有直接的关联关系；
- 包定义可以包含在多个scala文件中；
- 一个文件中可以定义多个包；

```
1 package com {
2   package hansight {
3     package scala {
4       object EventType extends Enumeration {
5         type EventType = Value
6         val DDOS, DI = Value
7       }
8     }
9   }
10 }
```

简单的包定义：

作用域规则：

相对包名：

可能导入混乱，

使用绝对包名：var arr = *root*.scala.collection.mutable.ArrayBuffer[EventType]

```
1 package com {
2   package hansight {
3     package scala {
4       object EventType extends Enumeration {
5         type EventType = Value
6         val DDOS, DI = Value
7       }
8     }
9   }
10 }
```

串联包：

```
1 package com.hansight.scala { // com 和 com.hansight包在这里不可见
2   object EventType extends Enumeration {
3     type EventType = Value
4     val DDOS, DI = Value
5   }
6 }
```

文件顶部标记法：

```
1 package com.hansight // 此包中的内容是可见的
2 package scala
3 object EventType extends Enumeration {
4   type EventType = Value
5   val DDOS, DI = Value
6 }
```

等同于

```
1 package com.hansight { // 此包中的内容是可见的
2   package scala {
3     object EventType extends Enumeration {
4       type EventType = Value
5       val DDOS, DI = Value
6     }
7   }
8 }
```

包对象

用于组织工具函数和常量

需要在父包中定义：

```
1 package com.hansight
2 package object people {
3   val defaultName = "scala examples"
4 }
5 package people {
6   class Person {
7     var name = defaultName
8   }
9 }
```

包可见性：

类成员在包中的可见度

```
 1 package com.hansight
 2 package object people {
 3   val defaultName = "scala examples"
 4 }
 5 package people {
 6   class Person {
 7     var name = defaultName
 8     private[people] def description = "people name:" + name; // people包内可见
 9   }
10 }
```

也可以延伸到上级：

private[hansight] def description = "people name:" + name;

引入import

import语句让你使用更短的名称，而非完整名称。

引入某个包的全部成员：

import com.hansight.utils. // 和Java中的通配符一样，为什么用_？因为可以作为包名

引入类或包的全部成员：

import com.hansihgt.utils.ESUtils._ // 就像Java中的import static，scala推荐使用

引入了包，便可使用较短名称访问子包：

import java.awt._

def handler(env: event.ActionHandler){ //TODO...} //event是java.awt的子包

在任何地方都可以使用import：

类里面

方法里面

引入包中多个成员

import java.awt.{Color, Font}

引入类并重命名

import java.util.{HashMap => JavaHashMap} import scala.collection.mutable._

隐藏某个引入的类

import java.util.{HashMap => , } // 引入java.util下的所有成员，但HashMap不引入

隐式引入：

import java.lang._

import scala._ // 会覆盖java.lang中的引入

import Predef._ // 包含一些工具函数，在scala有包对象之前已经存在的遗留

继承

继承类

scala继承类和java一样用关键字extends，子类可以定义超类没有的字段和方法，或重写超类的方法。

声明类为final它就不能被继承，还可以将方法或字段声明为final，以确保它们不会被重写。 和Java不同，Java中final声明常量，类似Scala中的val。

重写方法

Scala中重写非抽象方法必须使用override修饰符；

Scala中调用超类的方法和Java一样用super；

类型检查和转换

p是String类或String子类（貌似没子类）的实例

if (p.isInstanceOf[String]) {　　val s = p.asInstanceOf[String] }

如果p是null，返回false

如果p不是String，p.asInstanceOf[String]将抛异常。

p是String类的实例

if (p.getClass() == classOf[String]) {　　val s = p.asInstanceOf[String] }

模式匹配：

p match {　　case s: String => // String do　　case _ => //not String do }

受保护的字段和方法

可以将字段或方法声明为protected，这样成员可以被任意子类看到，但不能从其它位置看到。

访问对象仅限当前对象protected[this], 类似于private[this]

超类的构造

只有主构造方法才能调用超类的构造方法，辅构造方法永远不可能调用超类的构造方法；

调用超类的构造方法也和类定义交织在一起：

```
class Employee(val name: String, age: Int) extends Person(age) {

}
```

重写字段

匿名子类：

```
def meet(p: Person { def greeting: String }) = {

}
val alien = new Person(2) {
def greeting = "Greetings!"
}
meet(alien)
```

抽象类：

和Java一样：

使用abstract标记类为抽象类；

不能实例化；

和Java不一样：

抽象方法没有定义方法体，不需要abstract标记；

子类重写超类的抽象方法时，不需要使用override关键字

抽象字段：

指没有初始值的字段

子类重写超类的字段时，不需要使用override关键字

构造顺序和提前定义：

```
class Creature {
  val range = 10
  val env = new Array[Int](range)
}
class Ant extends Creature {
  override val range = 2
}

object TestAnt extends App {
  val ant = new Ant()
  println(ant.env.length) // 结果为0
  println(ant.range) // 结果为2
}
```

因为先构造Creature；构造env时调用range()，而Ant没有实例化返回0；然后构造Ant。

解决方法：

提前定义

```
class Ant extends {
override val range = 2
} with Creature {
}
```

Scala继承层次

io

```
var s = Source.fromFile("test.txt")
```

line

```
var lines = s.getLines
for (line <- lines) {
   println(line)
}
var arr = lines.toArray
val contents = s.mkString
```

char

```
for (c <- s) {
   println(c)
}

val iter = s.buffered
while(iter.hasNext) {
   if (iter.head ...)
      处理 iter.next
   else
      ..
}
```

词法单元和数字

```
val tokens = s.mkString.split("\\s+")
val numbers = for (w <- tokens) yield w.toDouble
或
val numbers = tokens.map(_.toDouble)
```

从控制台读取

默认会自动引入scala.Console对象，因此可以使用它的方法：

- readInt
- readLong
- read…