Life is short, you need Spark!

从零开始

不需要任何基础，带领您无痛入门 Spark

云计算分布式大数据 Spark 实战高手之路

王家林著

Spark 亚太研究院系列丛书

伴随着大数据相关技术和产业的逐步成熟，继 Hadoop 之后，Spark 技术以其无可比拟的优势，发展迅速，将成为替代 Hadoop 的下一代云计算、大数据核心技术。

本书特点

- 云计算分布式大数据 Spark 实战高手之路三部曲之第一部
- 网络发布版为图文并茂方式，边学习，边演练
- 不需要任何前置知识，从零开始，循序渐进

本书作者

Spark 亚太研究院院长和首席专家，中国目前唯一的移动互联网和云计算大数据集大成者。在 Spark、Hadoop、Android 等方面有丰富的源码、实务和性能优化经验。彻底研究了 Spark 从 0.5.0 到 0.9.1 共 13 个版本的 Spark 源码，并已完成 2014 年 5 月 31 日发布的 Spark1.0 源码研究。

Hadoop 源码级专家，曾负责某知名公司的类 Hadoop 框架开发工作，专注于 Hadoop 一站式解决方案的提供，同时也是云计算分布式大数据处理的最早实践者之一。

Android 架构师、高级工程师、咨询顾问、培训专家。

通晓 Spark、Hadoop、Android、HTML5，迷恋英语播音和健美。

“真相会使你获得自由。”

—— 耶稣《圣经》约翰 8:32KJV

“所有人类的不幸都来源于不能直面事实。”

—— 释迦摩尼

“道法自然”

—— 老子《道德经》第 25 章

《云计算分布式大数据 Spark 实战高手之路》

系列丛书三部曲

《云计算分布式大数据 **Spark** 实战高手之路---从零开始》：

不需要任何基础，带领您无痛入门 Spark 并能够轻松处理 Spark 工程师的日常编程工作，内容包括 Spark 集群的构建、Spark 架构设计、RDD、Shark/SparkSQL、机器学习、图计算、实时流处理、Spark on Yarn、JobServer、Spark 测试、Spark 优化等。

《云计算分布式大数据 **Spark** 实战高手之路---高手崛起》：

大话 Spark 源码，全世界最有情趣的源码解析，过程中伴随诸多实验，解析 Spark 1.0 的任何一句源码！更重要的是，思考源码背后的问题场景和解决问题的设计哲学和实现招式。

《云计算分布式大数据 **Spark** 实战高手之路---高手之巅》：

通过当今主流的 Spark 商业使用方法和最成功的 Hadoop 大型案例让您直达高手之巅，从此一览众山小。

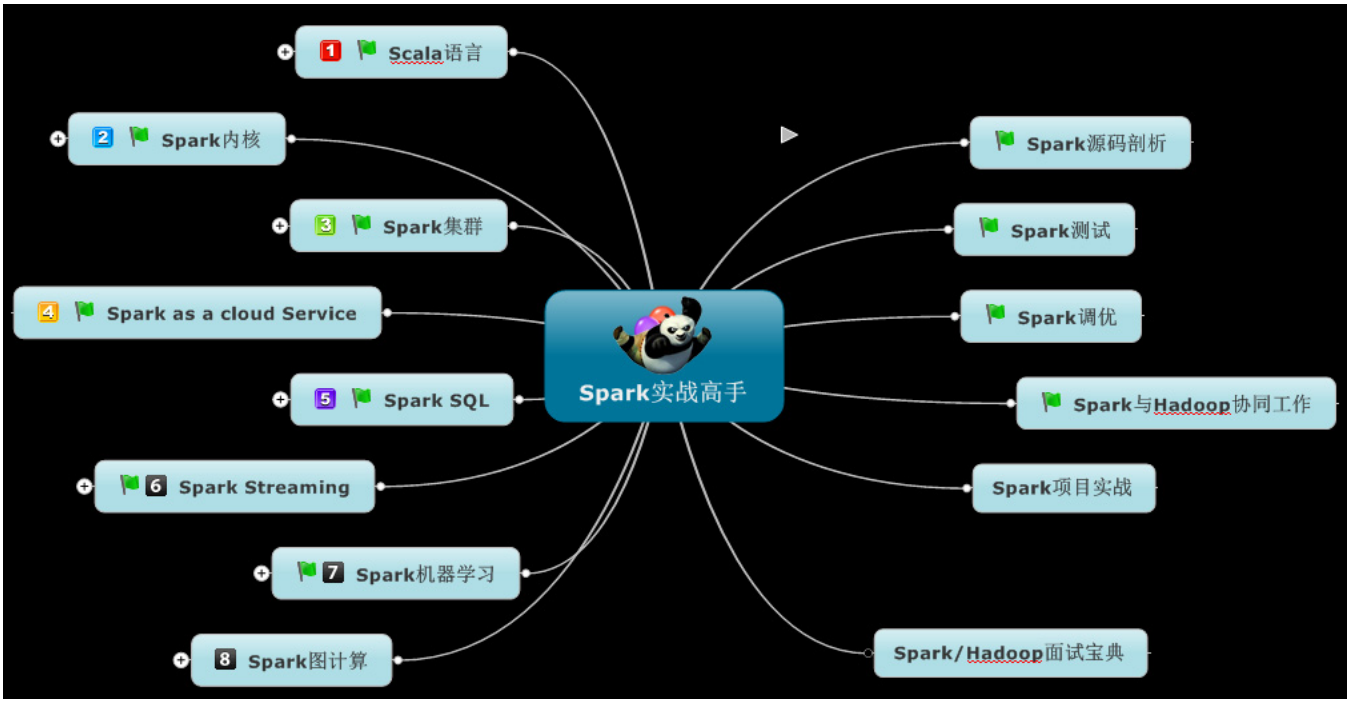
www.sparkinchina.com
TEL: 4006-998-758

《前言》

Spark采用一个统一的技术堆栈解决了云计算大数据的如流处理、图技术、机器学习、NoSQL查询等方面的所有核心问题，具有完善的生态系统，这直接奠定了其一统云计算大数据领域的霸主地位；

要想成为Spark高手，需要经历六大阶段

Spark 实战高手之核心技能点



第一阶段：熟练的掌握Scala语言

1. Spark 框架是采用 Scala 语言编写的，精致而优雅。要想成为 Spark 高手，你就必须阅读 Spark 的源代码，就必须掌握 Scala,；
2. 虽然说现在的 Spark 可以采用多语言 Java、Python 等进行应用程序开发，但是最快速的和支持最好的开发 API 依然并将永远是 Scala 方式的 API，所以你必须掌握 Scala 来编写复杂的和高性能的 Spark 分布式程序；
3. 尤其要熟练掌握 Scala 的 trait、apply、函数式编程、泛型、逆变与协变等；

推荐课程：" 精通Spark的开发语言：Scala最佳实践"

第二阶段：精通Spark平台本身提供给开发者API

1. 掌握Spark中面向RDD的开发模式 掌握各种transformation和action函数的使用；
2. 掌握 Spark 中的宽依赖和窄依赖以及 lineage 机制；
3. 掌握 RDD 的计算流程，例如 Stage 的划分、Spark 应用程序提交给集群的基本过程和 Worker 节点基础的工作原理等

推荐课程： "18 小时内掌握Spark：把云计算大数据速度提高 100 倍以上!"

第三阶段：深入Spark内核

此阶段主要是通过 Spark 框架的源码研读来深入 Spark 内核部分：

1. 通过源码掌握 Spark 的任务提交过程；
2. 通过源码掌握 Spark 集群的任务调度；
3. 尤其要精通 DAGScheduler、TaskScheduler 和 Worker 节点内部的工作的每一步的细节；

推荐课程： “Spark 1.0.0 企业级开发动手：实战世界上第一个Spark 1.0.0 课程，涵盖Spark 1.0.0 所有的企业级开发技术”

第四阶级:掌握基于Spark上的核心框架的使用

Spark 作为云计算大数据时代的集大成者，在实时流处理、图技术、机器学习、NoSQL 查询等方面具有显著的优势，我们使用 Spark 的时候大部分时间都是在使用其上的框架例如 Shark、Spark Streaming 等：

1. Spark Streaming 是非常出色的实时流处理框架 ,要掌握其 DStream、transformation 和 checkpoint 等；
2. Spark 的离线统计分析功能，Spark 1.0.0 版本在 Shark 的基础上推出了 Spark SQL，离线统计分析的功能的效率有显著的提升，需要重点掌握；
3. 对于 Spark 的机器学习和 GraphX 等要掌握其原理和用法；

推荐课程： “Spark企业级开发最佳实践”

第五阶级:做商业级别的Spark项目

通过一个完整的具有代表性的 Spark 项目来贯穿 Spark 的方方面面 ,包括项目的架构设计、用到的技术的剖析、开发实现、运维等，完整掌握其中的每一个阶段和细节，这样就可以让您以后可以从容面对绝大多数 Spark 项目。

推荐课程： “Spark架构案例鉴赏：Conviva、Yahoo！、优酷土豆、网易、腾讯、淘宝等公司的实际Spark案例”

第六阶级：提供Spark解决方案

1. 彻底掌握 Spark 框架源码的每一个细节；
2. 根据不同的业务场景的需要提供 Spark 在不同场景的下的解决方案；
3. 根据实际需要，在 Spark 框架基础上进行二次开发，打造自己的 Spark 框架；

推荐课程： “精通Spark：Spark内核剖析、源码解读、性能优化和商业案例实战”

《第四章：Spark 内核揭秘》

掌握 Spark 内核是精通 Spark 的关键，也是驾驭 Spark 的精髓所在。

基于 Spark 内核，Spark 构建起了一体化多元化的大数据处理流水线，在一个技术堆栈中即可以同时完成批处理、实时流处理、交互式查询、机器学习、图计算以及这些子框架之间数据和 RDD 算子的无缝共享与互操作。

可以说，Spark 内核是每个想彻底掌握 Spark 的人员的必修课，通过对内核的探索，我们对整个 Spark 的运行机制会了如指掌，这对 Spark 的大规模应用、性能优化、系统自定义开发 Spark 系统都是至关重要的。

本章首先带你初探 Spark 内核，接着通过源码游离 Spark 内核中，然后通过源码细致解析 Spark 作业的全生命周期，最后分享 Spark 性能优化，内容组织循序渐进，希望助力诸位 Spark 爱好者能够掌握 Spark 内核。

Spark 内核揭秘共分四个部分：

- 第一部分：Spark 内核初探
- 第二部分：Spark 内核核心源码解析
- 第三部分：Job 全生命周期源码解读
- 第四部分：Spark 性能优化

本讲是 **Spark** 内核揭秘的第二部分：**Spark** 内核核心源码解析，具体内容如下所示：

- 1，SparkContext 核心源码解析初体验；
- 2，TaskScheduler 的启动源码解析初体验；
- 3，DAGScheduler 源码解读初体验；
- 4，Spark 的 Web 监控页面；

不需任何前置知识，从零开始，循序渐进，成为 Spark 高手！

目录

一、SparkContext 核心源码解析初体验

SparkContext 在获得了一系列的初始化信息后开始创建并启动 TaskScheduler 实例：

```
// Create and start the scheduler
private[spark] var taskScheduler = SparkContext.createTaskScheduler(this, master)
private val heartbeatReceiver = env.actorSystem.actorOf(
  Props(new HeartbeatReceiver(taskScheduler)), "HeartbeatReceiver")
@volatile private[spark] var dagScheduler: DAGScheduler = _
try {
  dagScheduler = new DAGScheduler(this)
} catch {
  case e: Exception => throw
    new SparkException("DAGScheduler cannot be initialized due to %s".format(e.getMessage))
}

// start TaskScheduler after taskScheduler sets DAGScheduler reference in DAGScheduler's
// constructor
taskScheduler.start()
```

进入 createTaskScheduler 方法：

```
/** Creates a task scheduler based on a given master URL. Extracted for testing. */
private def createTaskScheduler(sc: SparkContext, master: String): TaskScheduler = {
  // Regular expression used for local[N] and local[*] master formats
  val LOCAL_N_REGEX = """local\[([0-9]+|\*)\]""".r
  // Regular expression for local[N, maxRetries], used in tests with failing tasks
  val LOCAL_N_FAILURES_REGEX = """local\[([0-9]+|\*)\s*,\s*([0-9]+)\]""".r
  // Regular expression for simulating a Spark cluster of [N, cores, memory] locally
  val LOCAL_CLUSTER_REGEX = """local-cluster\[\s*([0-9]+)\s*,\s*([0-9]+)\s*,\s*([0-9]+)\s*]""".r
  // Regular expression for connecting to Spark deploy clusters
  val SPARK_REGEX = """spark://(.*)""".r
  // Regular expression for connection to Mesos cluster by mesos:// or zk:// url
  val MESOS_REGEX = """(mesos|zk)://.*""".r
  // Regular expression for connection to Simr cluster
  val SIMR_REGEX = """simr://(.*)""".r

  // When running locally, don't try to re-execute tasks on failure.
  val MAX_LOCAL_TASK_FAILURES = 1

  master match {
    case "local" =>
```

我们看一下其 Standalone 的方式：

SparkContext.scala | BlockManager.scala | TaskSet.scala | TaskS

```
case SPARK_REGEX(sparkUrl) =>
  val scheduler = new TaskSchedulerImpl(sc)
  val masterUrls = sparkUrl.split(",").map("spark://" + _)
  val backend = new SparkDeploySchedulerBackend(scheduler, sc, masterUrls)
  scheduler.initialize(backend)
  scheduler
```

在上述代码中首先实例化一个 TaskSchedulerImpl：

```
/**
 * Schedules tasks for multiple types of clusters by acting through a SchedulerBackend.
 * It can also work with a local setup by using a LocalBackend and setting isLocal to true.
 * It handles common logic, like determining a scheduling order across jobs, waking up to launch
 * speculative tasks, etc.
 *
 * Clients should first call initialize() and start(), then submit task sets through the
 * runTasks method.
 *
 * THREADING: SchedulerBackends and task-submitting clients can call this class from multiple
 * threads, so it needs locks in public API methods to maintain its state. In addition, some
 * SchedulerBackends synchronize on themselves when they want to send events here, and then
 * acquire a lock on us, so we need to make sure that we don't try to lock the backend while
 * we are holding a lock on ourselves.
 */
private[spark] class TaskSchedulerImpl(
    val sc: SparkContext,
    val maxTaskFailures: Int,
    isLocal: Boolean = false)
  extends TaskScheduler with Logging
{
  def this(sc: SparkContext) = this(sc, sc.conf.getInt("spark.task.maxFailures", 4))
```

然后构建出了 masterUrls；

接着创建出了非常关键的 backend：

```
val backend = new SparkDeploySchedulerBackend(scheduler, sc, masterUrls)
```

我们进入其实现看一下：

```
package org.apache.spark.scheduler.cluster

import ...

private[spark] class SparkDeploySchedulerBackend(
    scheduler: TaskSchedulerImpl,
    sc: SparkContext,
    masters: Array[String])
  extends CoarseGrainedSchedulerBackend(scheduler, sc.env.actorSystem)
  with AppClientListener
  with Logging {

  var client: AppClient = null
  var stopping = false
  var shutdownCallback : (SparkDeploySchedulerBackend) => Unit = _

  val maxCores = conf.getOption("spark.cores.max").map(_.toInt)
  val totalExpectedCores = maxCores.getOrElse(0)
```

SparkDeploySchedulerBackend 核心是为了启动 CoarseGrainedExecutorBackend：

```
// The endpoint for executors to talk to us
val driverUrl = "akka.tcp://spark@%s:%s/user/%s".format(
  conf.get("spark.driver.host"), conf.get("spark.driver.port"),
  CoarseGrainedSchedulerBackend.ACTOR_NAME)
val args = Seq(driverUrl, "{{EXECUTOR_ID}}", "{{HOSTNAME}}", "{{CORES}}", "{{WORKER_URL}}")
val extraJavaOpts = sc.conf.getOption("spark.executor.extraJavaOptions")
  .map(Utils.splitCommandString).getOrElse(Seq.empty)
val classPathEntries = sc.conf.getOption("spark.executor.extraClassPath").toSeq.flatMap { cp =>
  cp.split(java.io.File.pathSeparator)
}
val libraryPathEntries =
  sc.conf.getOption("spark.executor.extraLibraryPath").toSeq.flatMap { cp =>
    cp.split(java.io.File.pathSeparator)
  }

// Start executors with a few necessary configs for registering with the scheduler
val sparkJavaOpts = Utils.sparkJavaOpts(conf, SparkConf.isExecutorStartupConf)
val javaOpts = sparkJavaOpts ++ extraJavaOpts
val command = Command("org.apache.spark.executor.CoarseGrainedExecutorBackend",
  args, sc.executorEnvs, classPathEntries, libraryPathEntries, javaOpts)
val appDesc = new ApplicationDescription(sc.appName, maxCores, sc.executorMemory, command,
  sc.ui.appUIAddress, sc.eventLogger.map(_.logDir))
```

此处使用了 Akka 技术进行不同机器之间的通信，CoarseGrainedExecutorBackend 是具体在 Worker 上执行具体的任务的进程的代表，所以我们的 backend 实例就是用来提交任务给 Executor 的：

```
package org.apache.spark.executor

import ...

private[spark] class CoarseGrainedExecutorBackend(
    driverUrl: String,
    executorId: String,
    hostPort: String,
    cores: Int,
    sparkProperties: Seq[(String, String)])
  extends Actor with ActorLogReceive with ExecutorBackend with Logging {

  Utils.checkHostPort(hostPort, "Expected hostport")

  var executor: Executor = null
  var driver: ActorSelection = null

  override def preStart() {
```

其实 CoarseGrainedExecutorBackend 是 Executor 的代理人，能够完成很多任务，例如启动一个任务：

```
override def receiveWithLogging = {
  case RegisteredExecutor =>
    logInfo("Successfully registered with driver")
    // Make this host instead of hostPort ?
    executor = new Executor(executorId, Utils.parseHostPort(hostPort)._1, sparkProperties,
      false)

  case RegisterExecutorFailed(message) =>
    logError("Slave registration failed: " + message)
    System.exit(1)

  case LaunchTask(data) =>
    if (executor == null) {
      logError("Received LaunchTask command but executor was null")
      System.exit(1)
    } else {
      val ser = SparkEnv.get.closureSerializer.newInstance()
      val taskDesc = ser.deserialize[TaskDescription](data.value)
      logInfo("Got assigned task " + taskDesc.taskId)
      executor.launchTask(this, taskDesc.taskId, taskDesc.name, taskDesc.serializedTask)
    }
```

回到 Standalone 的方式的代码处：

SparkContext.scala | BlockManager.scala | TaskSet.scala | TaskS

```
      case SPARK_REGEX(sparkUrl) =>
        val scheduler = new TaskSchedulerImpl(sc)
        val masterUrls = sparkUrl.split(",").map("spark://" + _)
        val backend = new SparkDeploySchedulerBackend(scheduler, sc, masterUrls)
        scheduler.initialize(backend)
        scheduler
```

接着代码是把 backend 传给了 initialize 方法中：

```
def initialize(backend: SchedulerBackend) {
  this.backend = backend
  // temporarily set rootPool name to empty
  rootPool = new Pool("", schedulingMode, 0, 0)
  schedulableBuilder = {
    schedulingMode match {
      case SchedulingMode.FIFO =>
        new FIFOSchedulableBuilder(rootPool)
      case SchedulingMode.FAIR =>
        new FairSchedulableBuilder(rootPool, conf)
    }
  }
  schedulableBuilder.buildPools()
}
```

在上述代码中显示处理调度模式例如 FIFO 和 Fair 的模式。
在代码块的最后返回实例化后的 scheduer：

```
case SPARK_REGEX(sparkUrl) =>
  val scheduler = new TaskSchedulerImpl(sc)
  val masterUrls = sparkUrl.split(",").map("spark://" + _)
  val backend = new SparkDeploySchedulerBackend(scheduler, sc, masterUrls)
  scheduler.initialize(backend)
  scheduler
```

二、TaskSceduler 启动源码解析初体验

TaskScheduler 实例对象启动源代码如下所示：

```
// start TaskScheduler after taskScheduler sets DAGScheduler reference in DAGScheduler's
// constructor
taskScheduler.start()
```

进入 start 方法：

```
/**
 * Low-level task scheduler interface, currently implemented exclusively by TaskSchedulerImpl.
 * This interface allows plugging in different task schedulers. Each TaskScheduler schedulers tasks
 * for a single SparkContext. These schedulers get sets of tasks submitted to them from the
 * DAGScheduler for each stage, and are responsible for sending the tasks to the cluster, running
 * them, retrying if there are failures, and mitigating stragglers. They return events to the
 * DAGScheduler.
 */
private[spark] trait TaskScheduler {

  def rootPool: Pool

  def schedulingMode: SchedulingMode

  def start(): Unit
```

找到 TaskSchedulerImpl 实现类中的 start 方法实现：

```
override def start() {
  backend.start()

  if (!isLocal && conf.getBoolean("spark.speculation", false)) {
    logInfo("Starting speculative execution thread")
    import sc.env.actorSystem.dispatcher
    sc.env.actorSystem.scheduler.schedule(SPECULATION_INTERVAL milliseconds,
          SPECULATION_INTERVAL milliseconds) {
      Utils.tryOrExit { checkSpeculatableTasks() }
    }
  }
}
```

其中 TaskSchedulerImpl 实例对象的主要目的在于启动 backend ,backend 的类型如下所示：

```
var backend: SchedulerBackend = null
```

SchedulerBackend 的具体实现如下所示：

```
/**
 * A scheduler backend that waits for coarse grained executors to connect to it through Akka.
 * This backend holds onto each executor for the duration of the Spark job rather than relinquishing
 * executors whenever a task is done and asking the scheduler to launch a new executor for
 * each new task. Executors may be launched in a variety of ways, such as Mesos tasks for the
 * coarse-grained Mesos mode or standalone processes for Spark's standalone deploy mode
 * (spark.deploy.*).
 */
private[spark]
class CoarseGrainedSchedulerBackend(scheduler: TaskSchedulerImpl, actorSystem: ActorSystem)
  extends SchedulerBackend with Logging
{
```

三、DAGScheduler 源码解读初体验

当构建完 TaskScheduler 之后，我们需要构建 DAGScheduler 这个核心对象：

```
@volatile private[spark] var dagScheduler: DAGScheduler = _
try {
  dagScheduler = new DAGScheduler(this)
} catch {
  case e: Exception => throw
    new SparkException("DAGScheduler cannot be initialized due to %s".format(e.getMessage))
}
```

进入其构造器代码：

```
def this(sc: SparkContext) = this(sc, sc.taskScheduler)
```

```
def this(sc: SparkContext, taskScheduler: TaskScheduler) = {
  this(
    sc,
    taskScheduler,
    sc.listenerBus,
    sc.env.mapOutputTracker.asInstanceOf[MapOutputTrackerMaster],
    sc.env.blockManager.master,
    sc.env)
}
```

```
private[spark]
class DAGScheduler(
    private[scheduler] val sc: SparkContext,
    private[scheduler] val taskScheduler: TaskScheduler,
    listenerBus: LiveListenerBus,
    mapOutputTracker: MapOutputTrackerMaster,
    blockManagerMaster: BlockManagerMaster,
    env: SparkEnv,
    clock: Clock = SystemClock)
  extends Logging {
```

可以看出构建 DAGScheduler 实例的时候需要把 TaskScheduler 实例对象作为参数传入。

在 DAGScheduler 实例化最终调用其 primary constructor 的时候会导致以下函数的执行：

```
initializeEventProcessActor()
```

看一下该函数内部所做的事情：

```
private def initializeEventProcessActor() {
  // blocking the thread until supervisor is started, which ensures eventProcessActor is
  // not null before any job is submitted
  implicit val timeout = Timeout(30 seconds)
  val initEventActorReply =
    dagSchedulerActorSupervisor ? Props(new DAGSchedulerEventProcessActor(this))
  eventProcessActor = Await.result(initEventActorReply, timeout.duration).
    asInstanceOf[ActorRef]
}
```

可以看出核心在于实例化 eventProcessActor 对象，eventProcessActor 会负责接收和发送 DAGScheduler 的消息，是 DAGScheduler 的通信载体。

四、Spark 的 Web 监控页面

在 SparkContext 中可以看到如下代码：

```
// Initialize the Spark UI, registering all associated listeners
private[spark] val ui = new SparkUI(this)
ui.bind()
```

首先是创建一个 Spark Application 的 Web 监控实例对象：

```
/**
 * Top level user interface for a Spark application.
 */
private[spark] class SparkUI(
    val sc: SparkContext,
    val conf: SparkConf,
    val securityManager: SecurityManager,
    val listenerBus: SparkListenerBus,
    var appName: String,
    val basePath: String = "")
  extends WebUI(securityManager, SparkUI.getUIPort(conf), conf, basePath, "SparkUI")
  with Logging {
```

然后 bind 方法会绑定一个 web 服务器：

```
/** Bind to the HTTP server behind this web interface. */
def bind() {
  assert(!serverInfo.isDefined, "Attempted to bind %s more than once!".format(className))
  try {
    serverInfo = Some(startJettyServer("0.0.0.0", port, handlers, conf, name))
    logInfo("Started %s at http://%s:%d".format(className, publicHostName, boundPort))
  } catch {
    case e: Exception =>
      logError("Failed to bind %s".format(className), e)
      System.exit(1)
  }
}
```

可以看出我们使用 Jetty 服务器来监控程序的运行和显示 Spark 集群的信息的。

Spark 亚太研究院

Spark 亚太研究院，提供 Spark、Hadoop、Android、Html5、云计算和移动互联网一站式解决方案。以帮助企业规划、部署、开发、培训和使用为核心，并规划和实施人才培训完整路径，提供源码研究和应用技术训练。

近期活动及相关课程

1、决战云计算大数据时代 Spark 亚太研究院 100 期公益大奖堂

每周四晚上 20：00—21:00

课程介绍：http://edu.51cto.com/course/course_id-1659.html#showDesc

报名参与：http://ke.qq.com/cgi-bin/courseDetail?course_id=6167

2、大数据 Spark 实战高手之路—熟练掌握 Scala 语言视频课程

国内第一个 Scala 视频学习课程！

成为 Spark 高手必备技能，必修课程！

现在购买，即可享受套餐优惠！

课程地址： http://edu.51cto.com/pack/view/id-124.html

近期公开课：

《决胜大数据时代：Hadoop、Yarn、Spark 企业级最佳实践》

集大数据领域最核心三大技术：Hadoop 方向 50%：掌握生产环境下、源码级别下的 Hadoop 经验，解决性能、集群难点问题；Yarn 方向 20%：掌握最佳的分布式集群资源管理框架，能够轻松使用 Yarn 管理 Hadoop、Spark 等；Spark 方向 30%：未来统一的大数据框架平台，剖析 Spark 架构、内核等核心技术，对未来转向 SPARK 技术，做好技术储备。课程内容落地性强，即解决当下问题，又有助于驾驭未来。

开课时间：9 月 26—28 日 上海、10 月 26—28 日北京、11 月 1—3 日深圳

咨询电话：4006-998-758

QQ 交流群： 1 群：317540673 （已满）
2 群 297931500

微信公众号：spark-china