SparkConf构造较为简单，使用**ConcurrentHashMap**保存各种Spark的配置属性，其中配置属性都是以“spark.“开头的字符串。在代码后半段，可以看到配置项的参数等，在中间部分，是一些开放的参数设置函数，例如set(k,v)，和setMaster()等方法。

val executorOptsKey = "spark.executor.extraJavaOptions"

val executorClasspathKey = "spark.executor.extraClassPath"

val driverOptsKey = "spark.driver.extraJavaOptions"

val driverClassPathKey = "spark.driver.extraClassPath"

val driverLibraryPathKey = "spark.driver.extraLibraryPath"

val sparkExecutorInstances = "spark.exe"

c**lass SparkConf(loadDefaults: Boolean) extends Cloneable with Logging with Serializable {**

**import SparkConf.\_**

/\*\* Create a SparkConf that loads defaults from system properties and the classpath \*/

**def this() = this(true)**

**private val settings = new ConcurrentHashMap[String, String]()**

**@transient private lazy val reader: ConfigReader = {**

**val \_reader = new ConfigReader(new SparkConfigProvider(settings))**

**\_reader.bindEnv(new ConfigProvider {**

**override def get(key: String): Option[String] = Option(getenv(key))**

**})**

**\_reader**

**}**

if (loadDefaults) {

loadFromSystemProperties(false)

}

private[spark] def loadFromSystemProperties(silent: Boolean): SparkConf = {

// Load any spark.\* system properties

for ((key, value) <- Utils.getSystemProperties if key.startsWith("spark.")) {

set(key, value, silent)

}

this

}

/\*\* Set a configuration variable. \*/

def set(key: String, value: String): SparkConf = {

set(key, value, false)

}

private[spark] def set(key: String, value: String, silent: Boolean): SparkConf = {

if (key == null) {

throw new NullPointerException("null key")

}

if (value == null) {

throw new NullPointerException("null value for " + key)

}

if (!silent) {

logDeprecationWarning(key)

}

settings.put(key, value)

this

}

private[spark] def set[T](entry: ConfigEntry[T], value: T): SparkConf = {

set(entry.key, entry.stringConverter(value))

this

}

private[spark] def set[T](entry: OptionalConfigEntry[T], value: T): SparkConf = {

set(entry.key, entry.rawStringConverter(value))

this

}

val executorOptsKey = "spark.executor.extraJavaOptions"

val executorClasspathKey = "spark.executor.extraClassPath"

val driverOptsKey = "spark.driver.extraJavaOptions"

val driverClassPathKey = "spark.driver.extraClassPath"

val driverLibraryPathKey = "spark.driver.extraLibraryPath"

val sparkExecutorInstances = "spark.executor.instances"

// Used by Yarn in 1.1 and before

sys.props.get("spark.driver.libraryPath").foreach { value =>

val warning =

s"""

|spark.driver.libraryPath was detected (set to '$value').

|This is deprecated in Spark 1.2+.

|

|Please instead use: $driverLibraryPathKey

""".stripMargin

logWarning(warning)

}

// Validate spark.executor.extraJavaOptions

getOption(executorOptsKey).foreach { javaOpts =>

if (javaOpts.contains("-Dspark")) {

val msg = s"$executorOptsKey is not allowed to set Spark options (was '$javaOpts'). " +

"Set them directly on a SparkConf or in a properties file when using ./bin/spark-submit."

throw new Exception(msg)

}

if (javaOpts.contains("-Xmx")) {

val msg = s"$executorOptsKey is not allowed to specify max heap memory settings " +

s"(was '$javaOpts'). Use spark.executor.memory instead."

throw new Exception(msg)

}

}

// Validate memory fractions

val deprecatedMemoryKeys = Seq(

"spark.storage.memoryFraction",

"spark.shuffle.memoryFraction",

"spark.shuffle.safetyFraction",

"spark.storage.unrollFraction",

"spark.storage.safetyFraction")

val memoryKeys = Seq(

"spark.memory.fraction",

"spark.memory.storageFraction") ++

deprecatedMemoryKeys

for (key <- memoryKeys) {

val value = getDouble(key, 0.5)

if (value > 1 || value < 0) {

throw new IllegalArgumentException(s"$key should be between 0 and 1 (was '$value').")

}

}

// Warn against deprecated memory fractions (unless legacy memory management mode is enabled)

val legacyMemoryManagementKey = "spark.memory.useLegacyMode"

val legacyMemoryManagement = getBoolean(legacyMemoryManagementKey, false)

if (!legacyMemoryManagement) {

val keyset = deprecatedMemoryKeys.toSet

val detected = settings.keys().asScala.filter(keyset.contains)

if (detected.nonEmpty) {

logWarning("Detected deprecated memory fraction settings: " +

detected.mkString("[", ", ", "]") + ". As of Spark 1.6, execution and storage " +

"memory management are unified. All memory fractions used in the old model are " +

"now deprecated and no longer read. If you wish to use the old memory management, " +

s"you may explicitly enable `$legacyMemoryManagementKey` (not recommended).")

}

}

if (contains("spark.master") && get("spark.master").startsWith("yarn-")) {

val warning = s"spark.master ${get("spark.master")} is deprecated in Spark 2.0+, please " +

"instead use \"yarn\" with specified deploy mode."

get("spark.master") match {

case "yarn-cluster" =>

logWarning(warning)

set("spark.master", "yarn")

set("spark.submit.deployMode", "cluster")

case "yarn-client" =>

logWarning(warning)

set("spark.master", "yarn")

set("spark.submit.deployMode", "client")

case \_ => // Any other unexpected master will be checked when creating scheduler backend.

}

}

if (contains("spark.submit.deployMode")) {

get("spark.submit.deployMode") match {

case "cluster" | "client" =>

case e => throw new SparkException("spark.submit.deployMode can only be \"cluster\" or " +

"\"client\".")

}

}

val encryptionEnabled = get(NETWORK\_ENCRYPTION\_ENABLED) || get(SASL\_ENCRYPTION\_ENABLED)

require(!encryptionEnabled || get(NETWORK\_AUTH\_ENABLED),

s"${NETWORK\_AUTH\_ENABLED.key} must be enabled when enabling encryption.")

}

/\*\*

\* Return a string listing all keys and values, one per line. This is useful to print the

\* configuration out for debugging.

\*/

def toDebugString: String = {

getAll.sorted.map{case (k, v) => k + "=" + v}.mkString("\n")

}

}