第5章

大数据分布式数据库系统HBase

HBase是一个具有高性能、高可靠性、面向列、可伸缩的分布式存储系统，利用HBase技术可在普通计算机上搭建大规模结构化的存储集群。HBase的目标是存储并处理大型数据，具体来说是仅需使用普通的硬件配置，就能够处理由成千上万的行和列所组成的大型数据。

与MapReduce的离线批处理计算框架不同，HBase是一个可以随机访问的存储和检索数据平台，弥补了HDFS不能随机访问数据的缺陷，适合实时性要求不高的业务场景。HBase中的数据以Byte[]数组的方式存储，它不区分数据类型，支持结构化、半结构化、非结构化数据，允许动态、灵活的数据模型。

本章简要介绍了HBase的特性及其与传统关系数据库的对比，分别介绍了HBase应用场景、数据模型、HBase工作原理、Shell操作命令，并介绍了常用的HBase编程接口以及编程实例。

5.1 HBase概述

5.1.1 HBase简介

HBase是一个分布式的、面向列的开源数据库，该技术来源于Google的论文《BigTable：一个结构化数据的分布式存储系统》。HBase的目标是处理非常庞大的表，可以通过水平扩展的方式，利用计算机集群处理由超过十亿行数据和数百万列元素组成的数据表。

HBase利用Hadoop MapReduce来处理HBase中的海量数据，实现高性能计算；利用ZooKeeper作为协同服务，实现稳定服务和失败恢复；使用HDFS作为高可靠的底层存储，利用廉价集群提供海量数据存储能力。此外，为了方便在HBase上进行数据处理，

Sqoop 为 HBase 提供了高效、便捷的 RDBMS 数据导入功能，Pig 和 Hive 为 HBase 提供了高层语言支持。

5.1.2 HBase 特性

(1) 规模大。一个表可以有上亿行、上百万列。

(2) 面向列。面向列表(族)的存储和权限控制，列(族)独立检索。

(3) 稀疏。对于为空(NULL)的列，并不占用存储空间，因此表可以设计得非常稀疏。

(4) 无模式。每一行都有一个可以排序的主键和任意多的列，列可以根据需要动态增加，同一张表中不同的行可以有截然不同的列。

(5) 数据多版本。在创建表时可以自定义多个 VERSION(版本)。可以将版本数理解为开辟的存储空间数目，当写入数据的次数大于版本信息时，存储的信息将会被清理，即 HBase 只保存数据最新的 N 个版本，当版本数过多时老版本会被删除。默认情况下初始版本为 1，即只提供一个版本空间存储数据。

(6) 数据类型单一。HBase 中的数据都是以 Byte[]数组的方式存储。

5.1.3 HBase 与传统关系数据库对比

关系数据库管理系统(Relational DateBase Management System，RDBMS)从 20 世纪 70 年代发展到今天，是一种非常成熟稳定的数据库管理系统，具备的功能包括面向磁盘的存储和索引结构、多线程访问、基于锁的同步访问机制、基于日志记录的回复机制和事务机制等。

但是随着 Web 2.0 应用的不断发展，传统的关系数据库已经无法满足 Web 2.0 的需求，无论在数据高并发方面，还是在高可扩展性和高可用性方面，传统的关系数据库都显得有些力不从心。关系数据库的关键特性——完善的事务机制和高效的查询机制，在 Web 2.0 时代也成为“鸡肋”。随着 HBase 在内的非关系型数据库的出现，有效弥补了传统关系数据库的缺陷，使 Web 2.0 应用中得到了广泛使用。HBase 与 RDMBS 的对比如表 5.1 所示。

表 5.1 HBase 与 RDBMS 的对比

对比项	HBase	RDBMS
数据类型	用户把不同格式的结构化、非结构化数据存储为 Byte[]数组，需要自己编写程序把字符串解析成不同数据类型	关系数据库采用关系模型，具有丰富的数据类型和存储方式
数据操作	HBase 操作只有简单的插入、查询、删除、清空等，无法实现像关联数据库中的表与表之间连接的操作	关系数据库中包含了丰富的操作，如插入、删除、更新、查询等，其中会涉及复杂的多表连接，需要借助多个表之间的主外键关联来实现
数据存储	HBase 是基于列存储的，每个列族都由几个文件保存，不同列族的文件是分离的	关系数据库是基于行模式存储的，元组或行会被连续地存储在磁盘页中

续表

对比项	HBase	RDBMS
索引	HBase只有一个索引——行键	关系数据库通常可以针对不同列构建复杂的多个索引，以提高数据访问性能
数据更新	HBase在更新时，当更新次数不超过版本号时，并不会删除数据旧的版本，而是生成一个新的本，旧的版本仍然保留	在关系数据库中，更新操作会用最新的当前值去替换记录中原来的旧值，旧值被覆盖后就不存在了
扩展性	可以实现灵活的水平扩展	关系数据库很难实现横向扩展，纵向扩展的空间也比较有限

5.1.4　HBase应用场景

HBase使用场景。

(1) 确信有足够大的数据。HBase适合分布在有上亿或上千亿行的数据。如果只有上百或上千万行，则用传统的RDBMS可能是更好的选择。因为所有数据可以在一两个节点保存，集群其他节点可能闲置。

(2) 确信可以不依赖所有RDBMS的额外特性(例如，列数据类型、第二索引、事务、高级查询语言等)。

(3) 确信有足够的硬件环境。因为HDFS在小于5个数据节点时，基本上体现不出它的优势。

HBase的一个典型应用是WebTable，一个以网页URL为主键的表，其中包含爬取的页面和页面的属性(如语言和MIME类型)。WebTable非常大，行数可以达十亿级别。在WebTable上连续运行用于批处理分析和解析的MapReduce作业，能够获取相关的统计信息，增加验证的MIME类型列以及人工搜索引擎进行索引的解析后的文本内容。同时，表格还会被以不同运行速度的“爬取器”(crawler)随机访问并随机更新其中的行；在用户单击访问网站的缓存页面时，需要实时地将这些被随机访问的页面提供给他们。

5.2　HBase数据模型

HBase是一个类似于BigTable的分布式数据库，它是一个稀疏的、长期存储的(存在HDFS上)、多维度的、排序的映射表。这张表的索引是行键、列(列族：列限定符)和时间戳。HBase的数据都是字符串，没有类型。

可以将一个表想象成一个大的映射关系，通过行键＋列(列族：列限定符)＋时间戳，就可以定位特定数据。由于HBase是稀疏存储数据的，所以某些列可以是空白的。

5.2.1　HBase数据模型术语

1. 表

HBase采用表(table)来组织数据，表由行和列组成，列划分为若干个列族。

2. 行

每个 HBase 表都由若干行(row)组成,每个行由行键(row key)来标识。访问表中的行有 3 种方式:全表扫描、通过单个行键访问、通过一个行键的区间来访问。行键可以是任意字符串,其最大长度为 64KB,行键的长度一般在 10～100 字节之间,在 HBase 内部行键被保存的方式多为字节数组。

3. 列族

一个 HBase 表被分组成许多"列族"(column family)的集合,它是基本的访问控制单元。列族需要在表创建时进行定义。当列族被创建之后,数据可以被存放到列族中的某个列下面。列名都以列族作为前缀,例如,score: math 和 score: English 这两个列都属于 score 这个列族。

4. 列限定符

列限定符(column qualifier)是列族中数据的索引,列族里的数据通过列限定符(或列)来定位。例如,person: student,student 就可以看做是对于列族 person 的一个列限定符,里面放的就是关于学生的数据。列限定符没有数据类型,一般被视为字节数组 Byte[]。在 HBase 中,为了方便操作,使用冒号(:)来分隔列族和列族修饰符,写在 HBase 源码中不能够修改。

5. 单元格

在 HBase 表中,通过行、列族和列限定符和时间戳或版本来确定一个"单元格"(cell)。一个单元格中可以保存同一份数据的多个版本。单元格的内容是不可分割的字节数组。单元格中存储的数据没有数据类型,被视为字节数组 Byte[]。每个版本对应一个不同的时间戳。

6. 时间戳

每个单元格都保存着同一份数据的多个版本,这些版本采用时间戳(timestamp)进行索引。每次对一单元格执行操作时(新建、删除、修改)时,HBase 都会隐式地自动生成并存储一个时间戳。在每个存储单元中,不同版本的数据按照时间戳倒序排序,即最新的版本数排在最前面。时间戳一般是 64 位整型,可以由用户赋值,也可以由 HBase 在数据库写入时自动赋值。

5.2.2 HBase 数据逻辑模型

下面以一个实例来阐释 HBase 的逻辑视图模型。表 5.2 是一张用来存储学生信息的 HBase 表,学号作为行键唯一标识每个学生。表中存有如下两组数据:学生李明,学号 1001,英语成绩 100 分,数学成绩 80 分;学生张三,学号 1002,英语成绩 100 分,数学成绩 90 分。表中的数据通过一个行键、一个列族和列限定符进行索引和查询定位。即通过

{row key,column family,timestamp}可以唯一地确定一个存储值,即一个键值对:

```
{row key, column family, timestamp}→ value
```

表 5.2　HBase 数据逻辑视图——学生表

行键	列族"Sname"	列族"course"		时间戳	value
		math	English		
1001		course: math		t3	80
			course: English	t2	100
	Sname			t1	LiMing
1002		course: math		t6	90
			course: English	t5	100
	Sname			t4	ZhangSan

5.2.3　HBase 数据物理模型

在逻辑视图中,HBase 中的每个表是由许多行组成的,但是在物理存储层面,它采用了基于列的存储方式,而不是像传统关系数据库那样采用基于行的存储方式,这也是HBase 和传统关系数据库的重要区别。表 5.2 中的逻辑视图在物理存储的时候,会存储成如表 5.3 所示的形式。按照 Sname 和 course 这两个列族分别存放,属于同一个列族的数据会保存在一起,空的列不会被存储成 null,而是不会被存储,但是当被请求时会返回 null 值。

表 5.3　HBase 数据物理视图——学生表

列族	行键	列限定符	时间戳	value
Sname	1001	Sname	t1	LiMing
	1002	Sname	t4	ZhangSan
Coures	1002	math	t6	90
		English	t5	100
	1001	math	t3	80
		English	t2	100

5.3　HBase 工作原理

5.3.1　HBase 体系结构

面对海量数据,HBase 可采用 Master/Slave 的方式进行分布式部署,一般采用HDFS 作为底层数据存储。HBase 的表包含的行的数量通常很大,无法存储在一台机器上,根据需要按行键的值对表中的行进行分区,每个行区间构成一个分区,并成为Region,它包含了位于这个区间的所有数据。HBase 体系结构如图 5.1 所示。

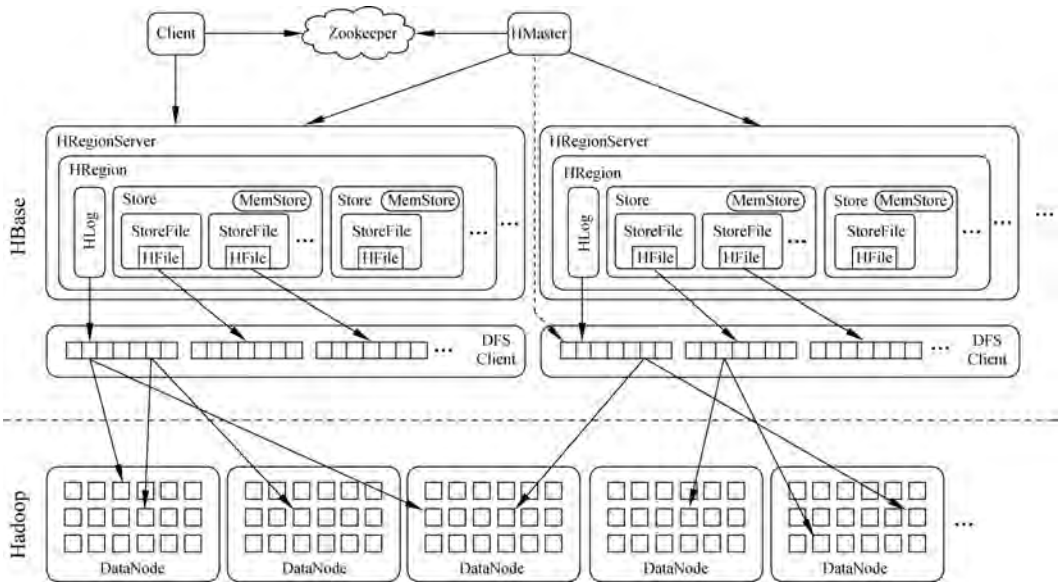

图 5.1 HBase 体系结构

5.3.2 HBase 工作组件

1. Client(客户端)

客户端包含访问 HBase 的接口,可做一些本地缓存,如借助 ZooKeeper 服务器从主服务器 HBase Master 获取 Region 位置信息,并从 Region(HRegion)服务器上读取数据。

2. Master(HMaster)

管理运行不同的 Region 服务器,也为客户端操作 HBase 的所有元数据提供接口,同时负责 RegionServer 的故障处理和 Region 的切分。在 HBase 集群中可以有多个 Master,实现集群的高可用,同时 Master 也负责对表的操作。

Master 功能。

(1) 管理用户对表的增加、删除、修改、查询等操作。

(2) 实现不同 Region 服务器之间的负载均衡。

(3) 在 Region 切分或合并后,负责重新调整 Region 的分布。

(4) 对发生故障失效的 Region 服务器上的 Region 进行迁移。

3. Region 和 Region 服务器

Region 是 HBase 中分布式存储和负载均衡的最小单元,即不同的 Region 可以分别在不同的 Region 服务器上,但同一个 Region 是不会拆分到多个 Region 服务器上的。Region 服务器的功能是管理 Region,负责 Region 的切分和合并。同时,Region 服务器也是 HBase 中最核心的模块,负责维护分配给自己的 Region,并响应用户的读写请求。

HBase一般采用HDFS作为底层存储文件系统，因此Region服务器需要向HDFS文件系统中读写数据。采用HDFS作为底层存储，可以为HBase提供可靠、稳定的数据存储，HBase自身并不具备数据复制和维护数据副本的功能，而HDFS可以为HBase提供这些支持。Region按大小分割，每个表一般只有一个Region，随着数据不断插入表，Region不断增大，当Region的某个列族达到一个阈值（默认256MB）时就会分成两个新的Region。

4. Store（HStore）

Region虽然是分布式存储的最小单元，但并不是存储的最小单元。Region由一个或者多个Store组成，每个Store保存一个列族；每个Strore又由一个MemStore和0至多个StoreFile组成，StoreFile包含HFile，HFile是HBase中KeyValue数据的存储格式，HFile是Hadoop的二进制格式文件，实际上StoreFile就是对HFile做了轻量级包装，即StoreFile底层就是HFile；MemStore存储在内存中，StoreFile存储在HDFS上（数据写入先写MemStore，当MemStore超过阈值（默认64MB），则会刷入以StoreFile磁盘）。HBase系统为每个Region服务器配置了一个HLog文件，它是一种预写式日志（Write Ahead Log）。

简单概括Region服务器是HBase的核心模块，而Store则是Region服务器的核心。每个Store对应了表中的一个列族的存储。每个Store包含了一个MemStore缓存和若干个StoreFile文件。

5. HLog

每当我们写一个请求，数据会先写入MemStore中，当MemStore超过了当前阈值才会被放入StoreFile中，但是在这期间一旦主机停电，那么之前MemStore缓存中的数据就会全部丢失。因此，HBase使用HLog来应对这种突发情况。

HBase要求用户更新数据必须先被记入日志后才能写入MemStore缓存，并且直到MemStore缓存内容对应的日志已经被写入磁盘后，该缓存内容才会被刷新写入磁盘。

6. ZooKeeper服务器

ZooKeeper服务器通常由多台机器构成集群来提供稳定、可靠的协同服务。在HBase服务器集群中，包含了一个Master和多个Region服务器，Master是这个HBase集群的“总管”，它必须知道Region服务器的状态。使用ZooKeeper就可以轻松做到这一点，每个Region服务器都需要到ZooKeeper中进行注册，ZooKeeper会实时监控每个Region服务器的状态并通知给Master，当某个Region服务器发生故障时，ZooKeeper会将故障通知Master。

在HBase中还可以同时启动多个Master，这时ZooKeeper不仅能够帮助维护当前的集群中的机器的服务状态，而且能够帮助选出一个Master作为“总管”，让这个“总管”来管理集群，并保证在任何时刻总有唯一一个Master在运行，避免Master的“单点失效”问题。

在 ZooKeeper 文件中还存储了-ROOT-表(不能被分割,用于实现 Region 定位)的地址和 Master 的地址。HBase 通过“三级寻址”方式找到需要的数据,客户端利用 ZooKeeper 服务器上的 ZooKeeper 文件来访问 HBase 数据。-ROOT-表记录了.META.表(切分成多个 Region)的 Region 位置信息,.META.表记录了用户数据表的 Region 位置信息,HBase 数据访问的三层结构如图 5.2 所示。

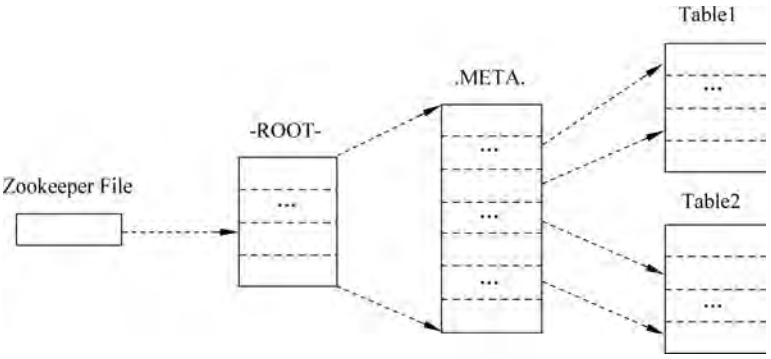

图 5.2 HBase 数据访问的三层结构

5.4 HBase 操作命令

HBase 提供了常用的 Shell 命令,方便用户对表进行操作。首先,我们需要启动 HDFS 和 HBase 进程;然后,在终端输入“hbase shell”命令即可进入 Shell 环境。

5.4.1 HBase 表操作

1. create(创建表)

(1) 创建表 student_1,列族为 Sname,列族版本号为 5(相当于可以存储最近的 5 个版本的记录),命令如下。

```
hbase > create 'student_1', {NAME => 'Sname', VERSIONS => 5}
```

(2) 创建表 student_2, 3 个列族分别为 Sname、gender、score,命令如下。

```
hbase > create 'student_2',{NAME => 'Sname'}, {NAME => 'gender}, {NAME => 'score'}
```

或者使用如下等价的命令(推荐)。

```
hbase > create 'student_2', 'Sname', 'gender, 'score'
```

(3) 创建表 student_3,将表依据分割算法 HexStringSplit 分布在 5 个 Region 里,命令如下。

```
Hbase > create 'student_3', 'Sname',{NUMREGIONS => 5, SPLITALGO => 'HexStringSplit'}
```

(4) 创建表 student_4,指定切分点,命令如下。

```
hbase > create 'student_4', 'Sname', {SPLITALGO => ['10','20', '30', '40']}
```

2. alter(修改列族模式)

(1) 向表 student_1 中添加列族 gender,命令如下。

```
hbase > alter 'student_1', NAME => 'gender'
```

(2) 删除表 student_1 中的列族 gender,命令如下。

```
Hbase > alter 'student_1', NAME => 'gender', METHOD => 'delete'
```

(3) 设定表 student_1 中列族 Sname 最大值为 128MB,命令如下。

```
hbase > alter 'student_1', METHOD => 'table_att', MAX_FILESIZE => '134217728'
```

以上命令中,"134217728"表示字节数,128MB 等于 134217728 字节。

3. list(列出 HBase 中所有的表信息)

```
hbase > list
```

4. count(统计表中的行数)

可以使用如下命令统计表 student_1 的行数。

```
hbase > count 'student_1'
```

5. describe(显示表的相关信息)

可以使用如下命令显示表 student_1 的信息。

```
hbase > describe 'student_1'
```

6. enable/disable(使表有效或无效)

可以使用如下命令使表 student 1 无效。

```
hbase > disable 'student_1'
```

可以使用如下命令使表 student 1 有效。

```
hbase > enable 'student_1'
```

7. drop(删除表)

删除某个表之前,必须先使该表无效。例如,删除表 student_1,命令如下。

```
hbase > disable 'student_1'
hbase > drop 'student_1'
```

8. exists(判断表是否存储)

判断表 student_1 是否存在,命令如下。

```
hbase > exists 'student_1'
```

9. exit(退出表)

退出 HBase Shell。

```
hbase > exit
```

5.4.2 HBase 数据操作

1. put(向表、行、列指定的单元格添加数据)

向表 student_2 中行键为 1001 号的学生和列 score: math 所对应的单元格中增加数据 80,时间戳设为 1234(若不指定会默认生成系统当前时间与 1970 年 1 月 1 日零点的毫秒值之差作为时间戳)命令如下。

```
hbase > put 'student_2', 1001, 'score:math', 80,1234
```

2. get(通过指定表名、行键、列、时间戳、时间范围和版本号来获得相应单元格的值)

(1) 获得表 student_2、行键 1001、列 score: math、时间范围为[ts1,ts2]版本号为 4 的数据,命令如下。

```
hbase > get 'student_2','1001',{COLUMN =>'score:math', TIMERANGE =>[ts1, ts2], VERSIONS =>
4}
```

(2) 获得表 student_2、行键 1001、列 socre: math 和 score: English 上的数据,命令如下。

```
hbase > get 'student_2', '1001', 'score:math', 'score:English'
```

3. scan(查询整个表的相关信息)

可以通过 TIMERANGE、FILTER、LIMIT、STARTROW、STOPROW、TIMESTAMP、MAXLENGTH、COLUMNS、CACHE 来限定所需要浏览的数据。

(1) 浏览表".META."、列 info: regioninfo 上的数据,命令如下。

```
hbase > scan '.META.', {COLUMNS => 'info:regioninfo'}
```

(2) 浏览表 student_1、列 score: English、时间范围为[1303668804,1303668904]的

数据，命令如下。

```
hbase > scan 'student_1', {COLUMNS => 'score:English', timerange => [1303668804, 1303668904]}
```

4. delete(删除指定单元格的数据)

删除表 student_1、行 1001、列族 Sname、时间戳为 ts1 上的数据，命令如下。

```
hbase > delete 'student_1', '1001', 'Sname', ts1
```

5.5　HBase 编程接口

5.5.1　HBase 常用 Java API

与 HBase 数据存储管理相关的 Java API 主要包括 Admin、HBaseConfiguration、HTableDescriptor、HColumnDescriptor、Put、Get、ResultScanner、Result、Scan。以下讲解这些类的功能与常用方法。

1. org.apache.hadoop.hbase.client.Admin

public interface Admin 是一个接口，必须通过调用 Connection.getAdmin()方法，返回一个实例化对象。该接口用于管理 HBase 数据库的表信息，包括创建表、删除表、列出表项、使表有效或无效、增加或删除表的列族成员、检查 HBase 的运行状态等。如表 5.4 所示。

表 5.4　Admin 接口的主要方法

方　　法	功　　能
void addColumn(TableName tableName,ColumnFamily-Descriptor columnFamily)	向一个已存在的表中添加列
void closeRegion(String regionname,String serverName)	关闭 Resign
void createTable(TableDescriptor desc)	创建表
void disableTable(TableName tableName)	使表无效
void deleteTable(TableName tableName)	删除表
Void enableTable(TableName tableName)	使表有效
Boolean tableExists(TableName tableName)	检查表是否存在
HTableDescriptor[] listTables()	列出所有表
Void void abort(String why, Throwable e)	终止服务器或客户端
Boolean balancer()	负载均衡

2. org.apache.hadoop.hbase.HBaseConfiguration

该类用于管理 HBase 的配置信息。如表 5.5 所示。

表 5.5 HBaseConfiguration 类的主要方法

方 法	功 能
Configuration create()	使用默认的 HBase 配置文件创建 Configuration
Configuration addHbaseResources(Configuration conf)	向当前 Configuration 增加 conf 中的配置信息
Void merge (Configuration destConf, Configuration srcConf)	合并两个 Configuration

3. org.apache.hadoop.hbase.client.Table

public interface Table 接口,必须调用 Connection.getTable()返回一个实例化对象。该接口用于与 HBase 进行通信。多线程情况下,使用 HTablePool 较好。如表 5.6 所示。

表 5.6 Table 接口的主要方法

方 法	功 能
Void close()	释放所有资源
Void delete(Delete delete)	删除指定的单元格或行
Boolean exists(Get get)	检查 Get 对象指定的列是否存在
Result get(Get get)	从指定行的单元格中取得相应的值
Void put(Put put)	向表中增加值
ResultScanner getScanner(byte[] family) ResultScanner getScanner(byte[] family, byte[] qualifier) ResultScanner getScanner(Scan scan)	获得 ResultScanner 实例
HTableDescriptor getTableDescriptor()	获得当前表格的 HTableDescriptor 对象
TableName getName()	获取当前表名

4. org.apache.hadoop.hbase.HTableDescriptor

HTableDescriptor 包含了 HBase 中表格的详细信息,如表中的列族、该表的类型(-ROOT-,.META.)、该表是否只读、MemStore 的最大空间、Region 什么时候应该切分等。如表 5.7 所示。

表 5.7 HTableDescriptor 类的主要方法

方 法	功 能
HTableDescriptor addFamily(HColumnDescriptor family)	增加列族
Collection < HColumnDescriptor > getFamilies()	返回所有列族的名称
TableName getTableName()	返回表名实例
Byte[] getValue(Bytes key)	获得属性值
HTableDescriptor removeFamily(byte[] column)	删除列族
HTableDescriptor setValue(byte[] key, byte[] value)	设置属性的值

5. org.apache.hadoop.hbase.HColumnDescriptor

HColumnDescriptor 包含了列族的详细信息，如列族的版本号、压缩设置等。如表 5.8 所示。

表 5.8 HColumnDescriptor 类的主要方法

方 法	功 能
Byte[] getName()	获得列族名称
Byte[] getValue(byte[] key)	获得某列单元格的值
HColumnDescriptor setValue(byte[] key, byte[] value)	设置某列单元格的值

6. org.apache.hadoop.hbase.client.Put

用于对单元格执行增加数据操作。如表 5.9 所示。

表 5.9 Put 类的主要方法

方 法	功 能
Put addColumn(byte[] family, byte[] qualifier, byte[] value)	将指定的列族、列、值增加到 Put 对象中
List < Cell > get(byte[] family, byte[] qualifier)	获取列族和列中的所有单元格
Boolean has(byte[] family, byte[] qualifier)	列族和列是否存在
Boolean has(byte[] family, byte[] qualifier, byte[] value)	检查列族和列中是否存在 value

7. org.apache.hadoop.hbase.client.Get

用来获取单行的信息。如表 5.10 所示。

表 5.10 Get 类的主要方法

方 法	功 能
Get addColumn(byte[] family, byte[] qualifier)	根据列族和列获取对应的列
Get setFilter(Filter filter)	通过设置过滤器获取具体的列

8. org.apache.hadoop.hbase.client.Result

用于存放 Get 或 Scan 操作后的查询结果，并以< key，value >的格式存储在 map 结构中。如表 5.11 所示。

表 5.11 Result 类的主要方法

方 法	功 能
Boolean containsColumn (byte [ ] family, byte [ ] qualifier)	检查是否包含列族和列限定符指定的列
List < Cell > getColumnCells (byte[ ] family, byte[ ] qualifier)	获得列族和列限定符指定的列中的所有单元格

续表

方　　法	功　　能
NavigableMap < byte[ ], byte[ ]> getFamilyMap(byte[ ] family)	根据列族获得包含列和值的所有行的键值对
Byte[ ] getValue(byte[ ] family, byte[ ] qualifier)	获得列族和列指定的单元格的最新值

9. org.apache.hadoop.hbase.client.ResultScanner

客户端获取值的接口，如表 5.12 所示。

表 5.12　ResultScanner 类的主要方法

方　　法	功　　能
Void close()	关闭 scanner 并释放资源
Result next()	获得下一个 Result 实例

10. org.apache.hadoop.hbase.client.Scan

可以利用 Scan 设定需要查找的数据，如设定版本号、起始行号、终止行号、列族、列名、返回值数量上限等。如表 5.13 所示。

表 5.13　Scan 类的主要方法

方　　法	功　　能
Scan addFamily(byte[ ] family)	设定列族
public Scan addColumn (byte[ ] family, byte[ ] qualifier)	设定列族和列
public Scan setMaxVersions() public Scan setMaxVersions(int maxVersions)	设定版本的最大个数
public Scan setTimeRange (long minStamp,long maxStamp)	设定最大最小时间戳范围
public Scan setFilter(Filter filter)	设定 Fileter 过滤
public Scan setStartRow(byte[ ] startRow)	设定开始的行
public Scan setStopRow(byte[ ] stopRow)	设定结束的行(不包含)
public Scan setBatch(int batch)	设定最多返回的单元格数目

5.5.2　HBase API 编程实例

5.5.1 节学习了 HBase 数据库的 Java API 操作，这一小节采用在 Linux 操作系统下，使用 IDEA 进行程序开发。在进行 HBase 编程之前需要启动 Hadoop 和 HBase。这里使用的 Hadoop 版本为 2.7.3，HBase 版本为 2.2.0。

1. 在 IDEA 中创建项目

IDEA 启动后，呈现如图 5.3 所示的启动界面，选择 Create New Project(创建一个新项目)，弹出如图 5.4 所示界面，选中左边栏 Java，单击 Next 按钮，弹出如图 5.5 所示的界

面，输入创建项目名为 hbaseDemo，项目路径可以自由定义，本书将项目路径设置为～/IdeaProjects/hbaseDemo，单击 Finish 按钮，IDEA 启动后界面如图 5.6 所示。

图 5.3　IDEA 启动界面

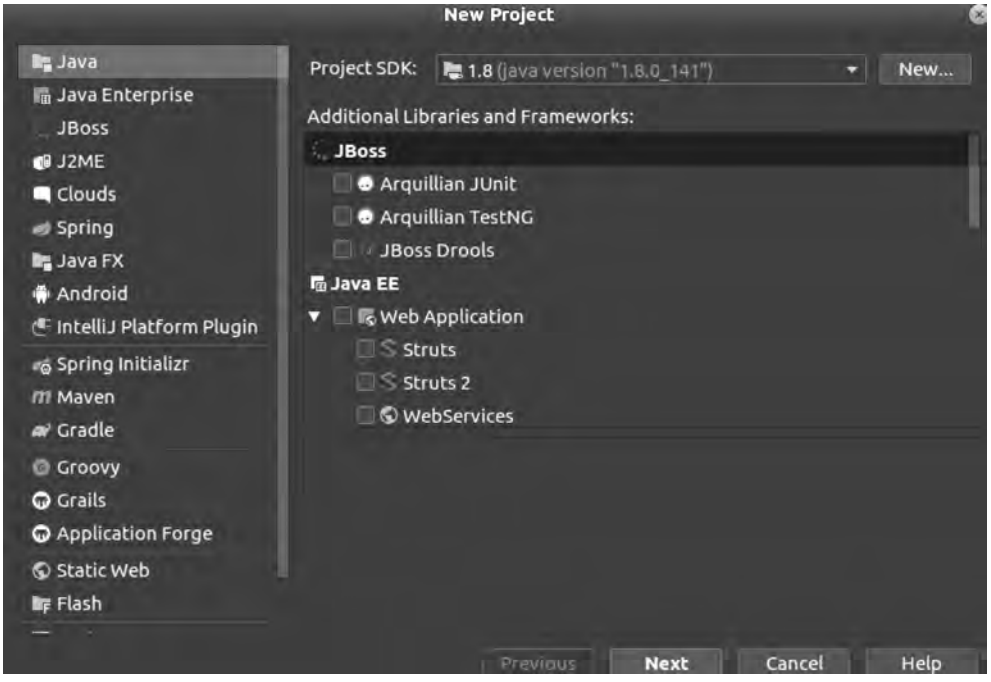

图 5.4　项目选择界面

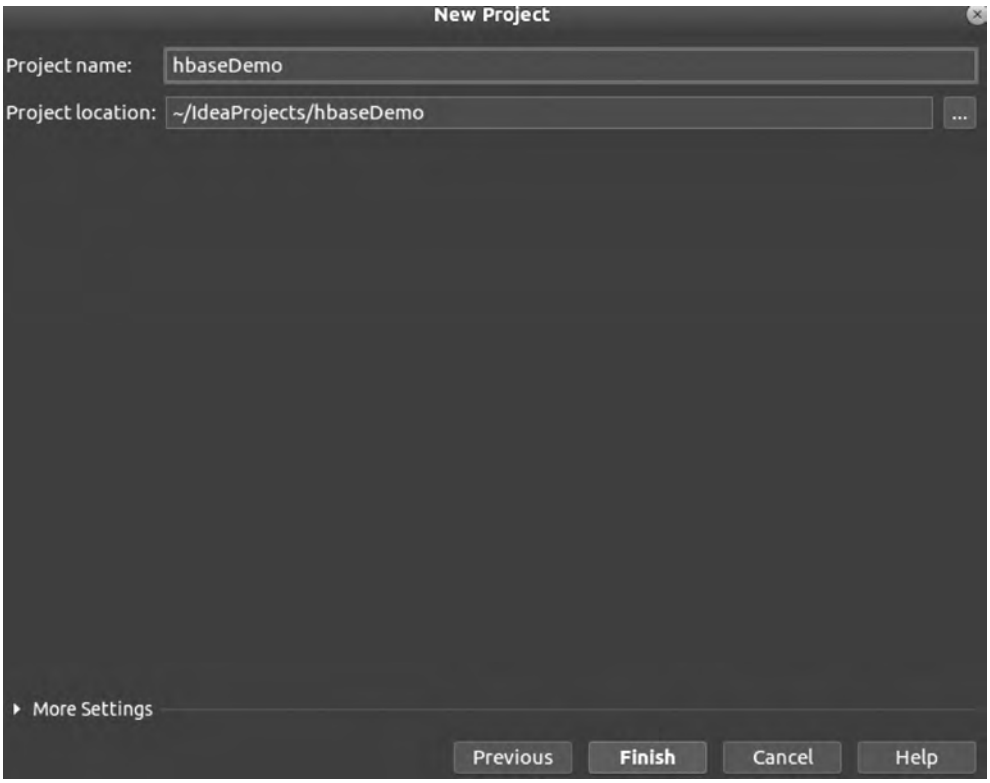

图 5.5　设置项目名称和路径界面

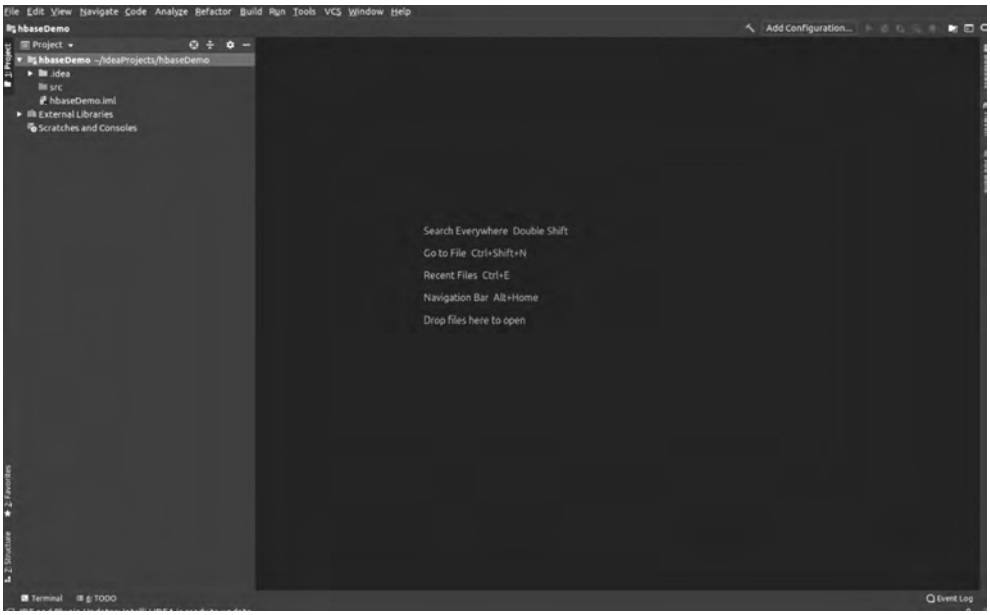

图 5.6　IDEA 启动后的界面

2. 为项目添加所需要的 JAR 包

在这里已经配置好了 JDK1.8 的环境，需要配置 HBase 需要用到的 JAR 包。单击左上角的 File 下拉菜单，选择 Project Structure 命令进入如图 5.7 所示的界面。

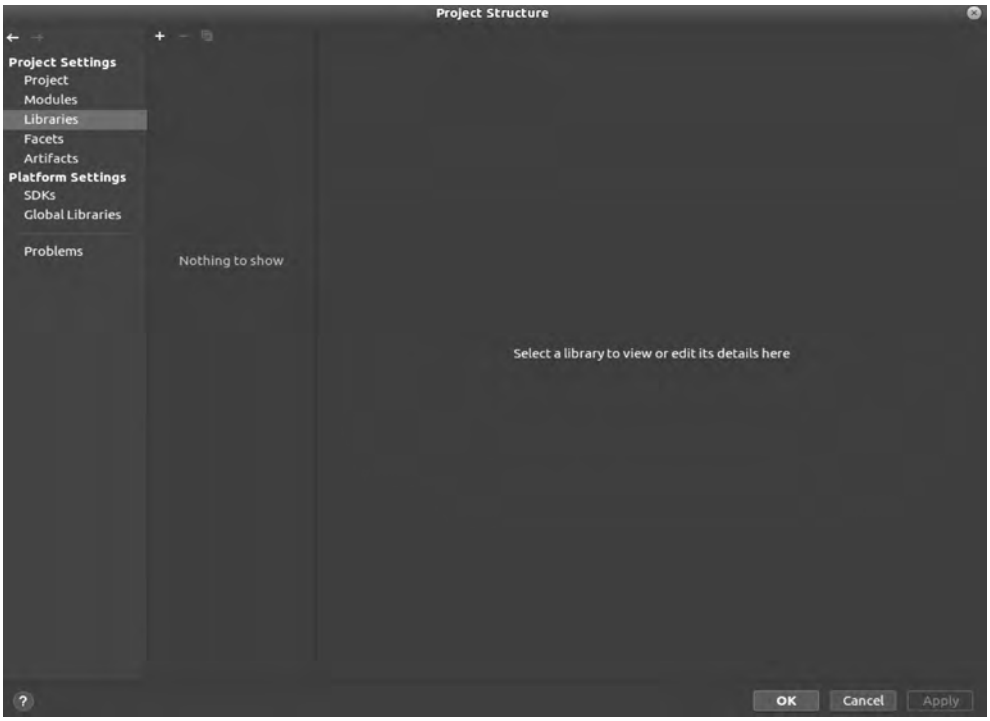

图 5.7　Project Structure 中的 Libraties

单击左上角的“＋”号，选择 Java，出现如图 5.8 所示的界面。为了能够编写一个能够与 HBase 交互的 Java 应用程序，需要在这个界面中加载该 Java 程序所需要用到的 JAR 包，这些 JAR 包中包含了可以访问 HBase 的 Java API。这些 JAR 包都位于 Linux 系统的 HBase 安装目录下，在本书中位于/usr/locak/hbase/lib 目录下。选中所有的.jar 文件（包括文件夹下面的），单击 OK 按钮，如图 5.9 所示。

3. 编写 Java 应用程序

编写一个 Java 应用程序，对 HBase 数据库进行操作。具体实现功能如下。

（1）完成与 HBase 数据库的连接和关闭。

（2）创建表 student_test。

（3）向 student_test 表中插入一条数据：行键 1001，Sname 为 XiaoQiang。

（4）在 student_test 表中删除一条数据。

（5）查询数据。

在 IDEA 工作界面左侧的面板中（如图 5.10 所示），右击 src 文件夹，在弹出的快捷菜单中选择 New→Java Class 命令，弹出如图 5.11 所示的对话框。

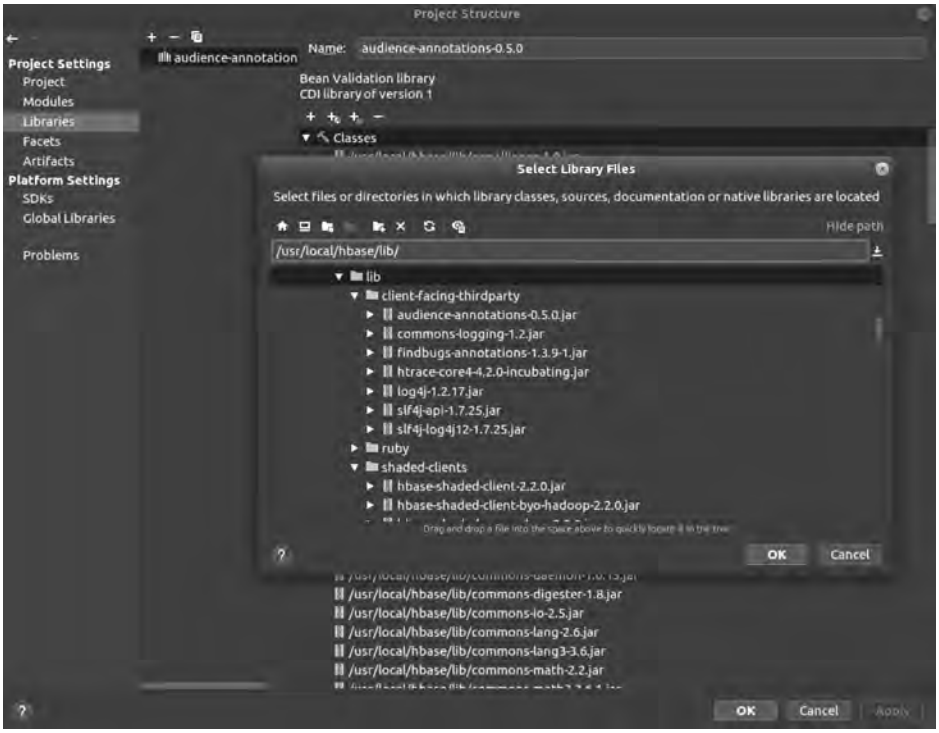

图 5.8　添加相关 jar 包界面

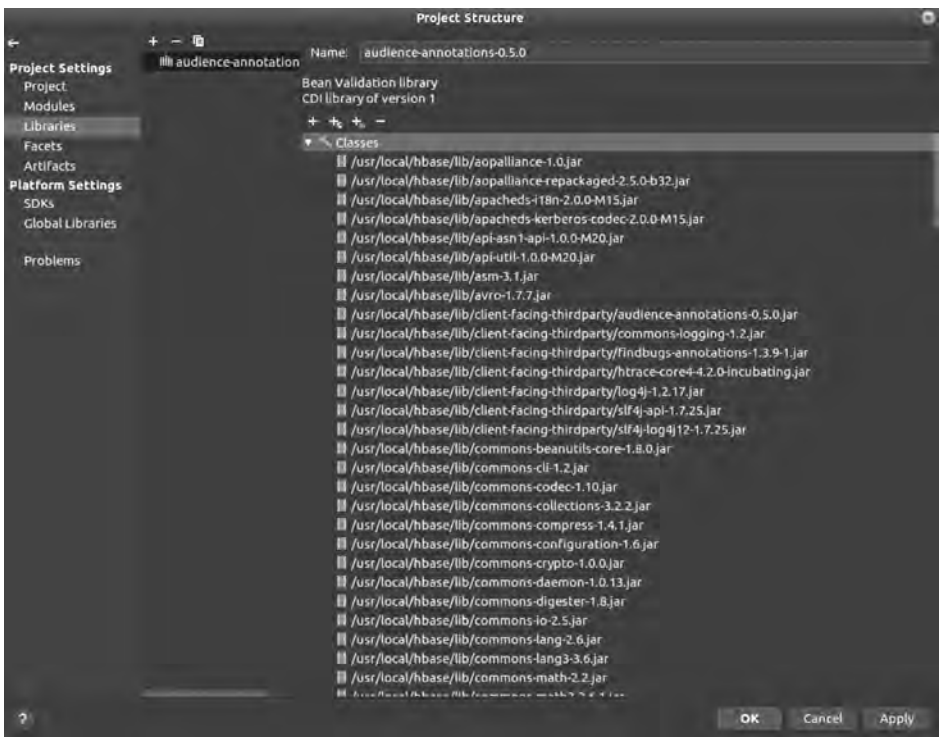

图 5.9　相关 jar 包添加完成

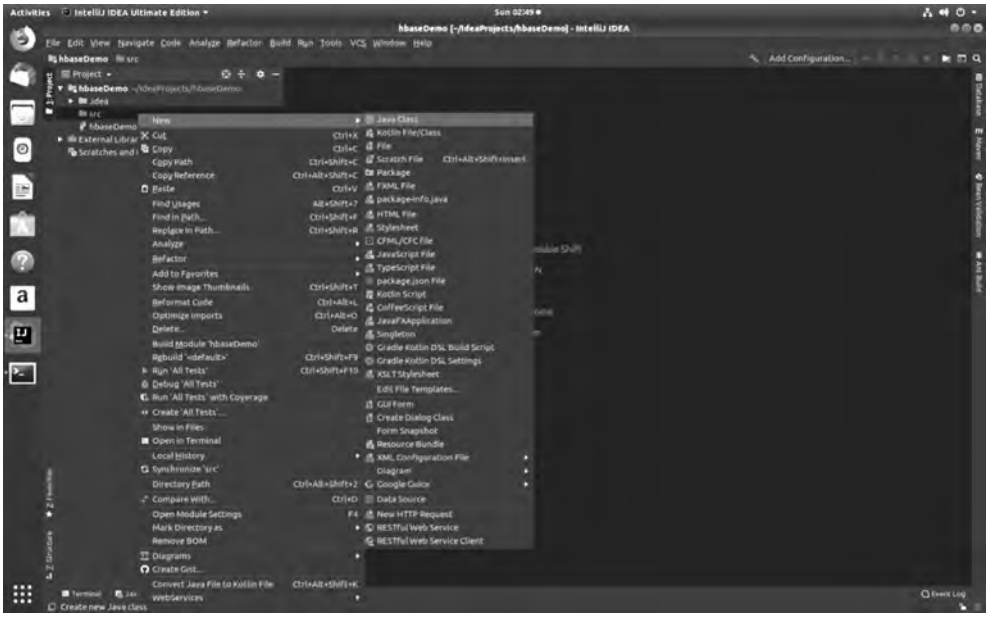

图 5.10　新建 Java Class 文件

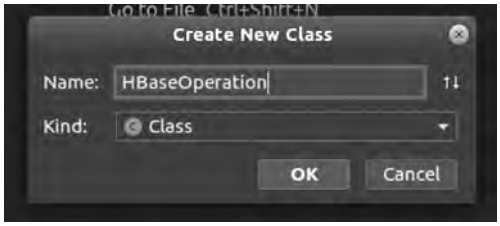

图 5.11　新建 Java Class 文件对话框

在对话框输入类名 HBaseOperation，单击 OK 按钮，弹出如图 5.12 所示的界面。

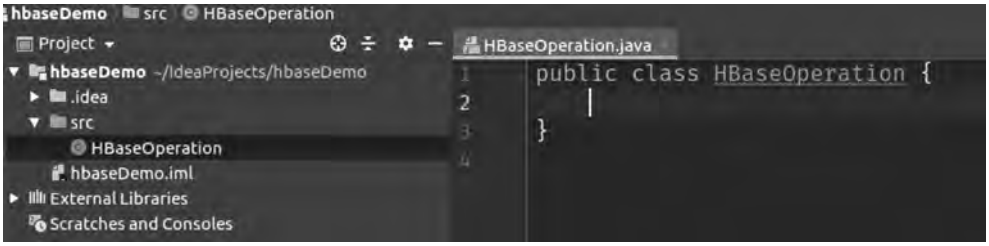

图 5.12　新建一个类文件之后的 IDEA 界面

IDEA 自动创建了一个名为 HBaseOperation.java 的源代码文件，在该文件中可以输入的代码如下。

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.*;
import org.apache.hadoop.hbase.client.*;
```

```
import java.io.IOException;
public class HBaseOperation {
    public static Configuration configuration;
    public static Connection connection;
    public static Admin admin;
    //主函数
    public static void main(String[] args) throws IOException {
        createTable("student_test", new String[]{"Sname"});
        insertRow("student_test", "1001", "Sname", "", "XiaoQiang");
        getData("student_test", "1001", "Sname", "", "XiaoQiang");
    }
    //建立连接
    public static void init(){
//        configuration.addResource("core-site.xml");
//        configuration.addResource("hbase-site.xml");
        configuration = HBaseConfiguration.create();
        configuration.set("hbase.rootdir", "hdfs://localhost:9000/hbase");
        try {
            connection = ConnectionFactory.createConnection(configuration);
            admin = connection.getAdmin();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    //关闭连接
    public static void close(){
        if(admin != null){
            try {
                admin.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if (null != connection){
            try {
                connection.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    //建立表
    @Deprecate
    public static void createTable(String myTableName, String[] colFamily)
            throws IOException{
        init();
        TableName tableName = TableName.valueOf(myTableName);
        if (admin.tableExists(tableName)){
            System.out.println("该表已存在");
```

```
    }else{
        HTableDescriptor hTableDescriptor = new HTableDescriptor(tableName);
        for (String str : colFamily) {
            HColumnDescriptor hColumnDescriptor = new HColumnDescriptor(str);
            hTableDescriptor.addFamily(hColumnDescriptor);
        }
        admin.createTable(hTableDescriptor);
    }
    close();
}
//删除表
public static void deleteTable(String tableName) throws IOException{
    init();
    TableName tn = TableName.valueOf(tableName);
    if (admin.tableExists(tn)){
        admin.disableTable(tn);
        admin.deleteTable(tn);
    }
    close();
}
//查看已有表
@Deprecate
public static void listTables() throws IOException{
    init();
    HTableDescriptor[] hTableDescriptors = admin.listTables();
    for (HTableDescriptor hTableDescriptor : hTableDescriptors) {
        System.out.println(hTableDescriptor.getNameAsString());
    }
    close();
}
//插入数据
public static void insertRow(String tableName, String rowKey, String colFamily,
                             String col, String val) throws IOException{
    init();
    Table table = connection.getTable(TableName.valueOf(tableName));
    Put put = new Put(rowKey.getBytes());
    put.addColumn(colFamily.getBytes(), col.getBytes(), val.getBytes());
    table.put(put);
    table.close();
    close();
}
//删除数据
public static void deleteRow(String tableName, String rowKey, String colFamily,
                             String col, String val) throws IOException{
    init();
    Table table = connection.getTable(TableName.valueOf(tableName));
    Delete delete = new Delete(rowKey.getBytes());
    //删除指定列族
    //delete.addFamily(Bytes.toBytes(colFamily));
```

```
        //删除指定列
        //delete.addColumn(Bytes.toBytes(colFamily), Bytes.toBytes(col));
        table.delete(delete);
        table.close();
        close();
    }
    //根据 rowkey 查找数据
    public static void getData(String tableName, String rowKey, String colFamily,
                               String col, String val) throws IOException{
        init();
        Table table = connection.getTable(TableName.valueOf(tableName));
        Get get = new Get(rowKey.getBytes());
        get.addColumn(colFamily.getBytes(), col.getBytes());
        Result result = table.get(get);
        showCell(result);
        table.close();
        close();
    }
    //格式化输出
    public static void showCell(Result result){
        Cell[] cells = result.rawCells();
        for (Cell cell : cells) {
            System.out.println("RowName:" + new String(CellUtil.cloneRow(cell)) + "");
            System.out.println("TimeStamp:" + cell.getTimestamp() + "");
              System.out.println("column Family:" + new String(CellUtil.cloneFamily
(cell)) + "");
            System.out.println("row Name:" + new String(CellUtil.cloneQualifier(cell)) + "");
            System.out.println("value" + new String(CellUtil.cloneValue(cell)) + "");
        }
    }
}
```

4. 编译运行程序

在确保 Hadoop 和 HBase 已经启动的情况下编译运行以上编写的代码。单击 IDEA 工作界面右上方的绿色箭头，即可运行程序，如图 5.13 所示。

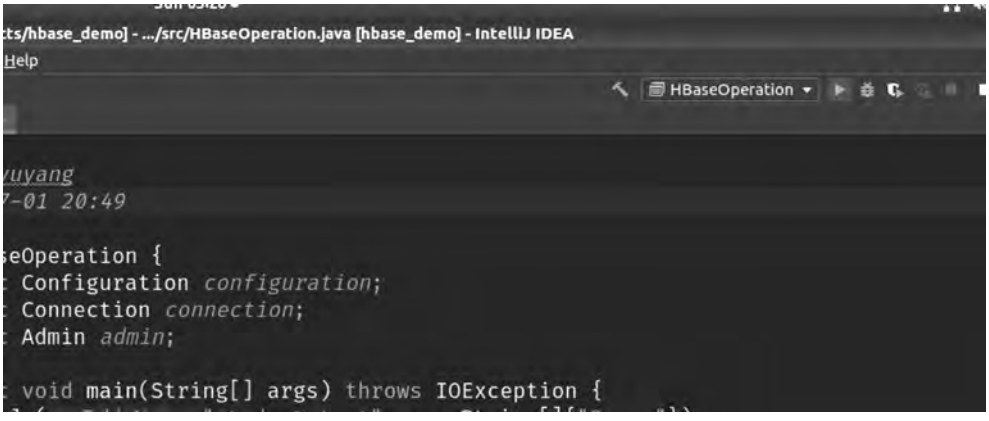

图 5.13　运行程序

程序运行完成之后，在控制台面板中会打印写入 HBase 数据库的学生信息，如图 5.14 所示。

可以在 Linux 的终端中启动 HBase Shell，查看生成的 student_test 表，启动 HBase Shell 的命令如图 5.15 所示。

```
RowName:1001
TimeStamp:1564309564713
column Family:Sname
row Name:
valueXiaoQiang
```

图 5.14　控制台打印信息

```
~$ cd /usr/local/hbase
/usr/local/hbase$ ./bin/hbase shell
```

图 5.15　启动 HBase Shell 的命令

进入 HBase Shell，可以使用 list 命令查看 HBase 数据库中是否含有名称为 student_test 的表。如图 5.16 所示，我们发现 student_test 表已经创建成功。

```
hbase(main):001:0> list
TABLE
stu
stu_test
student
student_2
student_test
5 row(s)
Took 0.3988 seconds
=> ["stu", "stu_test", "student", "student_2", "student_test"]
```

图 5.16　查看 HBase 中的所有表

可以使用 scan 命令查询刚才插入到 student_test 表中的一行数据，如图 5.17 所示。

```
hbase(main):004:0> scan 'student_test'
ROW                   COLUMN+CELL
 1001                 column=Sname:, timestamp=1564309564713, value=XiaoQiang
1 row(s)
Took 0.0215 seconds
```

图 5.17　通过 scan 命令查询插入的数据

如果需要反复调试 HBaseOperation.java 代码，需要删除 HBase 已经创建的 student test，可以使用 Shell 命令删除 student_test 表，如图 5.18 所示。

```
hbase(main):005:0> disable 'student_test'
Took 1.4126 seconds
hbase(main):006:0> drop 'student_test'
Took 0.4764 seconds
hbase(main):007:0>
```

图 5.18　删除 student_test 表

代码中的删除一条数据和关闭 Hadoop 与 HBase 连接的操作请自行测试。

5. 应用程序的部署

可以将以上编写的 Java 应用程序打包后部署到远程的服务器上运行。具体的部署方法请参见本书的第 14 章，在此不再赘述。

习题

1. 简述 HBase 及 HBase 的特性。

2. 对比分析 HBase 与传统关系数据库的异同。

3. 解释下列 HBase 中的模型术语：行键、列族、列限定符、单元格、时间戳。

4. 描述 HBase 的逻辑视图和物理视图的不同。

5. 已知职工表 emp：语文老师，姓名张三，年龄 30，薪金 8000；英语老师，姓名王五，年龄 35，薪金 10 000。请分别做出 HBase 逻辑视图和物理视图。

6. 简述 HBase 各功能组件及其作用。

7. 简述 HBase 的三层架构中各层次的名称和作用。

8. 列举并说明 HBase 的常用命令。