

## 第二十一章：数据库的使用

网上论坛、播客抓取器 (podcatchers) 甚至备份程序通常都会使用数据库进行持久化储存。基于 SQL 的数据库非常常见：这种数据库具有速度快、伸缩性好、可以通过网络进行操作等优点，它们通常会负责处理加锁和事务，有些数据库甚至还提供了故障恢复 (failover) 功能以提高应用程序的冗余性 (redundancy)。市面上的数据库有很多不同的种类：既有 Oracle 这样大型的商业数据库，也有 PostgreSQL、MySQL 这样的开源引擎，甚至还有 Sqlite 这样的可嵌入引擎。

因为数据库是如此的重要，所以 Haskell 也必须对数据库进行支持。本章将介绍其中一个与数据库进行互动的 Haskell 框架，并使用这个框架去构建一个播客下载器 (podcast downloader)，本书的 23 章还会对这个博客下载器做进一步的扩展。

### HDBC 简介

数据库引擎位于数据库栈 (stack) 的最底层，引擎负责将数据实际地储存到硬盘里面，常见的数据库引擎有 PostgreSQL、MySQL 和 Oracle。

大多数现代化的数据库引擎都支持 SQL，也即是结构化查询语言 (Structured Query Language)，并将这种语言用作读取和写入关系式数据库的标准方式。不过本书并不会提供 SQL 或者关系式数据库管理方面的教程[49]。

[49] O'Reilly 出版的《Learning SQL and SQL in a Nutshell》对于没有 SQL 经验的读者来说可能会有所帮助。

在拥有了支持 SQL 的数据库引擎之后，用户还需要寻找一种方法与引擎进行通信。虽然每个数据库都有自己的独有协议，但是因为各个数据库处理的 SQL 几乎都是相同的，所以通过为不同的协议提供不同的驱动，以此来创建一个通用的接口是完全可以做到的。

Haskell 有几种不同的数据库框架可用，其中某些框架在其他框架的基础上提供了更高层次的抽象，而本章将对 HDBC —— 也即是 Haskell DataBase Connectivity 系统进行介绍。通过 HDBC，用户可以在只需进行少量修改甚至无需进行修改的情况下，访问储存在任意 SQL 数据库里面的数据[50]。即使你并不需要更换底层的数据引擎，由多个驱动构成的 HDBC 系统也使得你在单个接口上面有更多选择可用。

[50] 假设你只能使用标准的 SQL。

HSQL 是 Haskell 的另一个数据库抽象库，它与 HDBC 具有相似的构想。除此之外，Haskell 还有一个名为 HaskellDB 的高层次框架，这个框架可以运行在 HDBC 或是 HSQL 之上，它被设计于用来为程序员隔离处理 SQL 时的相关细节。因为 HaskellDB 的设计无法处理一些非常常见的数据库访问模式，所以它并未被广泛引用。最后，Takusen 是一个使用左折叠 (left fold) 方式从数据库里面读取数据的框架。

### 安装 HDBC 和驱动

为了使用 HDBC 去连给定的数据库，用户至少需要用到两个包：一个包是 HDBC 的通用接口，而另一个包则是针对给定数据库的驱动。HDBC 包和所有其他驱动都可以通过 **Hackage** [51] 获得，本章将使用 1.1.3 版本的 HDBC 作为示例。

[51] 想要了解更多关于安装 Haskell 软件的相关信息，请阅读本书的《安装 Haskell 软件》一节。

除了 HDBC 包之外，用户还需要准备数据库后端和数据库驱动。本章会使用 Sqlite 3 作为数据库后端，这个数据库是一个嵌入式数据库，因此它不需要独立的服务器，并且也非常容易设置。很多操作系统本身就内置了 Sqlite 3，如果你的系统里面没有提供这一数据库，那么你可以到 <http://www.sqlite.org/> 里面进行下载。HDBC 的主页上面列出了指向已有 HDBC 后端驱动的连接，针对 Sqlite 3 的驱动也可以通过 Hackage 下载到。

如果读者打算使用 HDBC 去处理其他数据库，那么可以在 <http://software.complete.org/hdbc/wiki/KnownDrivers> 查看 HDBC 已有的驱动：上面展示的 ODBC 绑定 (binding) 基本上可以让你在任何平台 (Windows、POSIX 等等) 上面连接任何数据库；针对 PostgreSQL 的绑定也是存在的；而 MySQL 同样可以通过 ODBC 绑定进行支持，具体的信息可以在 **HDBC-ODBC API 文档** 里面找到。

### 连接数据库

连接至数据库需要用到数据库后端驱动提供的连接函数。每个数据库都有自己独特的连接方法。用户通常只会在初始化连接的时候直接调用从后端驱动模块载入的函数。

 v: latest ▾

数据库连接函数会返回一个数据库连接，不同驱动的数据库连接类型可能并不相同，但它们总是 `IConnection` 类型类的一个实例，并且所有数据库操作函数都能够与这种类型的实例进行协作。

在完成了与数据库的通信指挥，用户只要调用 `disconnect` 函数就可以断开与数据库的连接。以下代码展示了怎样去连接一个 `Sqlite` 数据库：

```
ghci> :module Database.HDBC Database.HDBC.Sqlite3

ghci> conn <- connectSqlite3 "test1.db"
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.1 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
Loading package HDBC-1.1.5 ... linking ... done.
Loading package HDBC-sqlite3-1.1.4.0 ... linking ... done.

ghci> :type conn
conn :: Connection

ghci> disconnect conn
```

## 事务

大部分现代化 SQL 数据库都具有事务的概念。事务可以确保一项修改的所有组成部分都会被实现，又或者全部都不实现。更进一步来说，事务可以避免访问相同数据库的多个进程看见正在进行的修改动作所产生的不完整数据。

大多数数据库都要求用户通过显式的提交操作来将所有修改储存到硬盘上面，又或者在“自动提交”模式下运行：这种模式在每一条语句的后面都会进行一次隐式的提交。“自动提交”模式可能会给不熟悉事务数据库的程序员带来一些方便，但它对于那些真正想要执行多条语句事务的人来说却是一个阻碍。

HDBC 有意地不对自动提交模式进行支持。当用户在修改数据库的数据之后，它必须显式地将修改提交到硬盘上面。有两种方法可以在 HDBC 里面做到这件事：第一种方法就是在准备好将数据写入到硬盘的时候，调用 `commit` 函数；而另一种方法则是将修改数据的代码包裹到 `withTransaction` 函数里面。`withTransaction` 会在被包裹的函数成功执行之后自动执行提交操作。

在将数据写入到数据库里面的时候，可能会出现一些问题。也许是因为数据库出错了，又或者数据库发现正在提交的数据出现了问题。在这种情况下，用户可以“回滚”事务进行的修改：回滚动作会撤销最近一次提交或是最近一次回滚之后发生的所有修改。在 HDBC 里面，你可以通过 `rollback` 函数来进行回滚。如果你使用 `withTransaction` 函数来包裹事务，那么函数将在事务发生异常时自动进行回滚。

要记住，回滚操作只会撤销掉最近一次 `commit` 函数、`rollback` 函数或者 `withTransaction` 函数引发的修改。数据库并不会像版本控制系统那样记录全部历史信息。本章稍后将展示一些 `commit` 函数的使用示例。

## 简单的查询示例

最简单的 SQL 查询语句都是一些不返回任何数据的语句，这些查询可以用于创建表、插入数据、删除数据、又或者设置数据库的参数。

`run` 函数是向数据库发送查询的最基本的函数，这个函数接受三个参数，它们分别是一个 `IConnection` 实例、一个表示查询的 `String` 以及一个由列表组成的参数。以下代码展示了如何使用这个函数去将一些数据储存到数据库里面。

```
ghci> :module Database.HDBC Database.HDBC.Sqlite3

ghci> conn <- connectSqlite3 "test1.db"
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.1 ... linking ... done.
Loading package bytestring-0.9.0.1 ... linking ... done.
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package mtl-1.1.0.0 ... linking ... done.
Loading package HDBC-1.1.5 ... linking ... done.
Loading package HDBC-sqlite3-1.1.4.0 ... linking ... done.

ghci> run conn "CREATE TABLE test (id INTEGER NOT NULL, desc VARCHAR(80))" []
0
```

 v: latest ▾

```
ghci> run conn "INSERT INTO test (id) VALUES (0)" []
1

ghci> commit conn

ghci> disconnect conn
```

在连接到数据库之后，程序首先创建了一个名为 `test` 的表，接着向表里面插入了一个行。最后，程序将修改提交到数据库，并断开与数据库的连接。记住，如果程序不调用 `commit` 函数，那么修改将不会被写入到数据库里面。

`run` 函数返回因为查询语句而被修改的行数量。在上面展示的代码里面，第一个查询只是创建一个表，它并没有修改任何行；而第二个查询则向表里面插入了一个行，因此 `run` 函数返回了数字 `1`。

## SqlValue

在继续讨论后续内容之前，我们需要先了解一种由 HDBC 引入的数据类型：`SqlValue`。因为 Haskell 和 SQL 都是强类型系统，所以 HDBC 会尝试尽可能地保留类型信息。与此同时，Haskell 和 SQL 类型并不是——对应的。更进一步来说，日期和字符串里面的特殊字符这样的东西，在每个数据库里面的表示方法都是不相同的。

`SqlValue` 类型具有 `SqlString`、`SqlBool`、`SqlNull`、`SqlInteger` 等多个构造器，用户可以通过使用这些构造器，在传给数据库的参数列表里面表示各式各样不同类型的数据，并且仍然能够将这些数据储存到一个列表里面。除此之外，`SqlValue` 还提供了 `toSql` 和 `fromSql` 这样的常用函数。如果你非常关心数据的精确表示的话，那么你还是可以在有需要的时候，手动地构造 `SqlValue` 数据。

## 查询参数

HDBC 和其他数据库一样，都支持可替换的查询参数。使用可替换参数主要有几个好处：它可以预防 SQL 注射攻击、避免因为输入里面包含特殊字符而导致的问题、提升重复执行相似查询时的性能、并通过查询语句实现简单且可移植的数据插入操作。

假设我们想要将上千个行插入到新的表 `test` 里面，那么我们可能会执行像 `INSERT INTO test VALUES (0, 'zero')` 和 `INSERT INTO test VALUES (1, 'one')` 这样的查询上千次，这使得数据库必须独立地分析每条 SQL 语句。但如果我们将被插入的两个值替换为占位符，那么服务器只需要对 SQL 查询进行一次分析，然后就可以通过重复地执行这个查询来处理不同的数据了。

使用可替换参数的第二个原因和特殊字符有关。因为 SQL 使用单引号表示域 (field) 的末尾，所以如果我们想要插入字符串 `"I don't like 1"`，那么大多数 SQL 数据库都会要求我们把这个字符串写成 `I don't like 1`，并且不同的特殊字符（比如反斜杠符号）在不同的数据库里面也会需要不同的转移规则。但是只要使用 HDBC，它就会帮你自动完成所有转义动作，以下展示的代码就是一个例子：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> run conn "INSERT INTO test VALUES (?, ?)" [toSql 0, toSql "zero"]
1

ghci> commit conn

ghci> disconnect conn
```

在这个示例里面，`INSERT` 查询包含的问号是一个占位符，而跟在占位符后面的就是要传递给占位符的各个参数。因为 `run` 函数的第三个参数接受的是 `SqlValue` 组成的列表，所以我们使用了 `toSql` 去将列表中的值转换为 `SqlValue`。HDBC 会根据目前使用的数据库，自动地将 `String "zero"` 转换为正确的表示方式。

在插入大量数据的时候，可替换参数实际上并不会带来任何性能上的提升。因此，我们需要对创建 SQL 查询的过程做进一步的控制，具体的方法在接下来的一节里面就会进行讨论。

### Note

#### 使用可替换参数

当服务器期望在查询语句的指定部分看见一个值的时候，用户才能使用可替换参数：比如在执行 `SELECT` 语句的 `WHERE` 子句时就可以使用可替换参数；又或者在执行 `INSERT` 语句的时候就可以把要插入的值设置为可替换参数；但执行 `run "SELECT * from ?" [toSql "tablename"]` 是无法运行的。这是因为表的名字并非一个值，所以大多数数据库都不允许这种语法。因为在实际中很少人会使用这种方式去替换一个不是值的事物，所以这并不会带来什么大的问题。

 v: latest ▼

## 预备语句

HDBC 定义了一个 `prepare` 函数，它可以预先准备好一个 SQL 查询，但是并不将查询语句跟具体的参数绑定。`prepare` 函数返回一个 `Statement` 值来表示已编译的查询。

在拥有了 `Statement` 值之后，用户就可以对它调用一次或多次 `execute` 函数。在对一个会返回数据的查询执行 `execute` 函数之后，用户可以使用任意的获取函数去取得查询所得的数据。诸如 `run` 和 `quickQuery` 这样的函数都会在内部使用查询语句和 `execute` 函数；为了让用户可以更快捷妥当地执行常见的任务，像是 `run` 和 `quickQuery` 这样的函数都会在内部使用 `Statement` 值和 `execute` 函数。当用户需要对查询的具体执行过程有更多的控制时，就可以考虑使用 `Statement` 而不是 `run` 函数。

以下代码展示了如何通过 `Statement` 值，在只使用一条查询的情况下插入多个值：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> stmt <- prepare conn "INSERT INTO test VALUES (?, ?)"

ghci> execute stmt [toSql 1, toSql "one"]
1

ghci> execute stmt [toSql 2, toSql "two"]
1

ghci> execute stmt [toSql 3, toSql "three"]
1

ghci> execute stmt [toSql 4, SqlNull]
1

ghci> commit conn

ghci> disconnect conn
```

在这段代码里面，我们创建了一个预备语句并使用 `stmt` 函数去调用它。我们一共执行了那个语句四次，每次都向它传递了不同的参数，这些参数会被用于替换原有查询字符串中的问号。在代码的最后，我们提交了修改并断开数据库。

为了方便地重复执行同一个预备语句，HDBC 还提供了 `executeMany` 函数，这个函数接受一个由多个数据行组成的列表作为参数，而列表中的数据行就是需要调用预备语句的数据行。正如以下代码所示：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> stmt <- prepare conn "INSERT INTO test VALUES (?, ?)"

ghci> executeMany stmt [[toSql 5, toSql "five's nice"], [toSql 6, SqlNull]]

ghci> commit conn

ghci> disconnect conn
```

### Note

#### 更高效的查询执行方法

在服务器上面，大多数数据库都会对 `executeMany` 函数进行优化，使得查询字符串只会被编译一次而不是多次。**[52]**在一次插入大量数据的时候，这种优化可以带来极为有效的性能提升。有些数据库还可以将这种优化应用到执行查询语句上面，并非所有数据库都能做到这一点。

**[52]** 对于不支持这一优化的数据库，HDBC 会通过模拟这一行为来为用户提供一致的 API，以便执行重复的查询。

## 读取结果

本章在前面已经介绍过如何通过查询语句，将数据插入到数据库；在接下来的内容中，我们将学习从数据库里面获取数据的方法。`quickQuery` 函数的类型和 `run` 函数非常相似，只不过 `quickQuery` 函数返回的是一个由查询结果组成的列表而不是被执行的查询语句。`quickQuery` 函数通常与 `SELECT` 语句一起使用，正如以下代码所示：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> quickQuery' conn "SELECT * from test where id < 2" []
[[SqlString "0", SqlNull], [SqlString "0", SqlString "zero"], [SqlString "1", SqlString "one"]]

ghci> disconnect conn
```

正如之前展示过的一样，`quickQuery'` 函数能够接受可替换参数。上面的代码没有使用任何可替换参数，所以在调用 `quickQuery'` 的时候，我们没有在函数调用的末尾给定任何的可替换值。`quickQuery'` 返回一个由行组成的列表，其中每个行都会被表示为 `[SqlValue]`，而行里面的值会根据数据库返回时的顺序进行排列。在有需要的时候，用户可以使用 `fromSql` 可以将这些值转换为普通的 Haskell 类型。

因为 `quickQuery'` 的输出有一些难读，我们可以对上面的示例进行一些扩展，将它的结果格式化得更美观一些。以下代码展示了对结果进行格式化的具体方法：

```
-- file: ch21/query.hs
import Database.HDBC.Sqlite3 (connectSqlite3)
import Database.HDBC

{- | 定义一个函数，它接受一个表示要获取的最大 id 值作为参数。
   函数会从 test 数据库里面获取所有匹配的行，并以一种美观的方式将它们打印到屏幕上面。 -}
query :: Int -> IO ()
query maxId =
  do -- 连接数据库
    conn <- connectSqlite3 "test1.db"

    -- 执行查询并将结果储存在 r 里面
    r <- quickQuery' conn
      "SELECT id, desc from test where id <= ? ORDER BY id, desc"
      [toSql maxId]

    -- 将每个行转换为 String
    let stringRows = map convRow r

    -- 打印行
    mapM_ putStrLn stringRows

    -- 断开与服务端之间的连接
    disconnect conn

  where convRow :: [SqlValue] -> String
        convRow [sqlId, sqlDesc] =
          show intid ++ ": " ++ desc
          where intid = (fromSql sqlId)::Integer
                desc = case fromSql sqlDesc of
                  Just x -> x
                  Nothing -> "NULL"
        convRow x = fail $ "Unexpected result: " ++ show x
```

这个程序所做的工作和本书之前展示过的 `ghci` 示例差不多，唯一的区别就是新添加了一个 `convRow` 函数。这个函数接受来自数据库行的数据，并将它转换为一个易于打印的 `String` 值。

注意，这个程序会直接通过 `fromSql` 取出 `intid` 值，但是在处理 `fromSql sqlDesc` 的时候却使用了 `Maybe String`。不知道你是否还记得，我们在定义表的时候，曾经将表的第一列设置为不准包含 `NULL` 值，但是第二列却没有进行这样的设置。所以，程序不需要担心第一列是否会包含 `NULL` 值，只要对第二行进行处理就可以了。虽然我们也可以使用 `fromSql` 去将第二行的值直接转换为 `String`，但是这样一来，程序只要遇到 `NULL` 值就会出现异常。因此，我们需要把 SQL 的 `NULL` 转换为字符串 `"NULL"`。虽然这个值在打印的时候可能会与字符串 `'NULL'` 出现混淆，但对于这个例子来说，这样的问题还是可以接受的。让我们尝试在 `ghci` 里面调用这个函数：

```
ghci> :load query.hs
[1 of 1] Compiling Main             ( query.hs, interpreted )
Ok, modules loaded: Main.

ghci> query 2
0: NULL
0: zero
1: one
2: two
```

 v: latest ▼



## 使用语句进行数据读取操作

正如前面的《预备语句》一节所说，用户可以使用预备语句进行读取操作，并且在一些环境下，使用不同的方法从这些语句里面读出数据将是一件非常有用的事情。像 `run`、`quickQuery` 这样的常用函数实际上都是使用语句去完成任务的。

为了创建一个执行读取操作的预备语句，用户只需要像之前执行写入操作那样使用 `prepare` 函数来创建预备语句，然后使用 `execute` 去执行那个预备语句就可以了。在语句被执行之后，用户就可以使用各种不同的函数去读取语句中的数据。`fetchAllRows` 函数和 `quickQuery` 函数一样，都返回 `[[SqlValue]]` 类型的值。除此之外，还有一个名为 `sFetchAllRows` 的函数，它在返回每个列的数据之前，会先将它们转换为 `Maybe String`。最后，`fetchAllRowsAL` 函数对于每个列返回一个 `(String, SqlValue)` 二元组，其中 `String` 类型的值是数据库返回的列名。本章接下来的《数据库元数据》一节还会介绍其他获取列名的方法。

通过 `fetchRow` 函数，用户可以每次只读取一个行上面的数据，这个函数会返回 `IO (Maybe [SqlValue])` 类型的值：当所有行都已经被读取了之后，函数返回 `Nothing`；如果还有尚未读取的行，那么函数返回一个行。

## 惰性读取

前面的《惰性 I/O》一节曾经介绍过如何对文件进行惰性 I/O 操作，同样的方法也可以用于读取数据库中的数据，并且在处理可能会返回大量数据的查询时，这种特性将是非常有用的。通过惰性地读取数据，用户可以继续使用 `fetchAllRows` 这样的方便的函数，不必再在行数据到达时手动地读取数据。通过以谨慎的方式使用数据，用户可以避免将所有结构都缓存到内存里面。

不过要注意的是，针对数据库的惰性读取比针对文件的惰性读取要负责得多。用户在以惰性的方式读取完整文件之后，文件就会被关闭，不会留下什么麻烦的事情。另一方面，当用户以惰性的方式从数据库读取完数据之后，数据库的连接仍然处于打开状态，以使用户继续执行其他操作。有些数据库甚至支持同时发送多个查询，所以 HDBC 是无法在用户完成一次惰性读取之后就关闭连接的。

在使用惰性读取的时候，有一点是非常重要的：在尝试关闭连接或者执行一个新的查询之前，一定要先将整个数据集读取完。我们推荐你使用严格（strict）函数又或者以一行接一行的方式进行处理，从而尽量避免惰性读取带来的复杂的交互行为。

### Tip

如果你是刚开始使用 HDBC，又或者对惰性读取的概念并不熟悉，但是又需要读取大量数据，那么可以考虑通过反复调用 `fetchRow` 来获取数据。这是因为惰性读取虽然是一种非常强大而且有用的工具，但是正确地使用它并不是那么容易的。

要对数据库进行惰性读取，只需要使用不带单引号版本的数据库函数就可以了。比如 `fetchAllRows` 就是 `fetchAllRows'` 的惰性读取版本。惰性函数的类型和对应的严格版本函数的类型一样。以下代码展示了一个惰性读取示例：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> stmt <- prepare conn "SELECT * from test where id < 2"

ghci> execute stmt []
0

ghci> results <- fetchAllRowsAL stmt
[("id",SqlString "0"),("desc",SqlNull)],[("id",SqlString "0"),("desc",SqlString "zero")],[("id",SqlString "1"),("desc",SqlString "one")]

ghci> mapM_ print results
[("id",SqlString "0"),("desc",SqlNull)]
[("id",SqlString "0"),("desc",SqlString "zero")]
[("id",SqlString "1"),("desc",SqlString "one")]

ghci> disconnect conn
```

虽然使用 `fetchAllRowsAL` 函数也可以达到取出所有行的效果，但是如果需要读取的数据集非常大，那么 `fetchAllRowsAL` 函数可能就会消耗非常多的内容。通过以惰性的方式读取数据，我们同样可以读取非常大的数据集，但是只需要使用常数数量的内存。惰性版本的数据库读取函数会把结果放到一个块里面进行求值；而严格版的数据库读取函数则会直接获取所有结果，把它们储存到内存里面，接着打印。

## 数据库元数据

在一些情况下，能够知道一些关于数据库自身的信息是非常有用的。比如说，一个程序可能会想要看看数据库里面目前已有的表，然后自动创建缺失的表或者对数据库的模式（schema）进行更新。而在另外一些情况下，程序可能会需要根据正在使用的数据库后端对自己的行为进行修改。

通过使用 `getTables` 函数，我们可以取得数据库目前已定义的所有列表；而 `describeTable` 函数则可以告诉我们给定表的各个列的定义信息。

调用 `dbServerVer` 和 `proxiedClientName` 可以帮助我们了解正在运行的数据库服务器，而 `dbTransactionSupport` 函数则可以让我们了解到数据库是否支持事务。以下代码展示了这三个函数的调用示例：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> getTables conn
["test"]

ghci> proxiedClientName conn
"sqlite3"

ghci> dbServerVer conn
"3.5.9"

ghci> dbTransactionSupport conn
True

ghci> disconnect conn
```

`describeResult` 函数返回一组 `[(String, SqlColDesc)]` 类型的二元组，二元组的第一个项是列的名字，第二个项则是与列相关的信息：列的类型、大小以及这个列能够为 `NULL` 等等。完整的描述可以参考 HDBC 的 API 手册。

需要注意一点是，某些数据库并不能提供所有这些元数据。在这种情况下，程序将引发一个异常。比如 `Sqlite3` 就不支持前面提到的 `describeResult` 和 `describeTable`。

## 错误处理

HDBC 在错误出现时会引发异常，异常的类型为 `SqlError`。这些异常会传递来自底层 SQL 引擎的信息，比如数据库的状态、错误信息、数据库的数字错误代号等等。

因为 `ghci` 并不清楚应该如何向用户展示一个 `SqlError`，所以这个异常将导致程序停止，并打印一条没有什么用的信息。就像这样：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> quickQuery' conn "SELECT * from test2" []
*** Exception: (unknown)

ghci> disconnect conn
```

上面的这段代码因为使用了 `SELECT` 去获取一个不存在的表，所以引发了错误，但 `ghci` 返回的错误信息并没有说清楚这一点。通过使用 `handleSqlError` 辅助函数，我们可以捕捉 `SqlError` 并将它重新抛出为 `IOError`。这种格式的错误可以被 `ghci` 打印，但是这种格式会使得用户比较难于通过编程的方式来获取错误信息的指定部分。以下是一个使用 `handleSqlError` 处理异常的例子：

```
ghci> conn <- connectSqlite3 "test1.db"

ghci> handleSqlError $ quickQuery' conn "SELECT * from test2" []
*** Exception: user error (SQL error: SqlError {seState = "", seNativeError = 1, seErrorMsg = "prepare 20: SELECT * from test2: no such table:

ghci> disconnect conn
```

这个新的错误提示具有更多信息，它甚至包含了一条说明 `test2` 表并不存在的消息，这比之前的错误提示有用得多了。作为一种标准实践（standard practice），很多 HDBC 程序员都将 `main = handleSqlError $ do` 放到程序的开头，确保所有未被捕获的 `SqlError` 都会以更有效的方式被打印。

除了 `handleSqlError` 之外，HDBC 还提供了 `catchSql` 和 `handleSql` 这两个函数，它们类似于标准的 `catch` 函数和 `handle` 函数，主要的区别在于 `catchSql` 和 `handleSql` 只会中断 HDBC 错误。想要了解更多关于错误处理的信息，可以参考本书第 19 章《错误处理》一章。

讨论

0条评论

Real World Haskell 中文版


1 登录

推荐

推文

分享

最新发布



开始讨论...

通过以下方式登录

或注册一个 DISQUS 帐号

姓名

来做第一个留言的人吧!

- 在 REAL WORLD HASKELL 中文版 上还有

第八章：高效文件处理、正则表达式、文件名匹配 — Real World Haskell 中文版

1条评论 • 4年前

Yutong Zhang —

第十章：代码案例学习：解析二进制数据格式 — Real World Haskell 中文版

1条评论 • 4年前

Y — 此处少个“+”：instance Show Greymap where show (Greymap w h m \_) = "Greymap " ++ show w ++ "x" ++ show h ++ " " ++ show m

Real World Haskell 中文版

2条评论 • 6年前

yadsun — 校正：ghc是生成快速本底代码的优化编译器。中 本底 - ->本地

Pearls of Functional Algorithm Design

1条评论 • 6年前

Tonghua Su — where is the content?