如何定制一个基于REST Service的ODBC驱动程序

# 概要

REST Service能够帮助开发者以简单统一的接口向终端用户提供服务。然而数据分析的应用场景中，一些成熟的数据分析工具（例如Tableau, Excel等）要求用户提供ODBC数据源，在这种情况下，REST Service并不能满足用户所有对数据的使用需求。本文从实现的角度详细介绍了如何在现有REST Service的基础上，完成一个定制ODBC驱动程序的开发。文章侧重介绍了ODBC驱动程序的实现原理，结合代码详细说明了ODBC与REST Service之间的数据交互，并在文章末尾介绍了ODBC客户端程序调用ODBC API的原理，以及实际开发中调试环境的搭建。

# 可能受益的读者

目前主流的数据分析工具，例如Tableau，Microstrategy，excel都只能够ODBC Driver，来访问底层的数据源。也就是说，在开发数据库或者数据仓库的过程中，即使我们已经实现了符合SQL规范的数据访问接口，哪怕提供了自己的JDBC驱动程序，仍然无法保证数据用户能够有效地使用我们的数据。为此，我们需要额外地为数据源定制一个ODBC Driver。

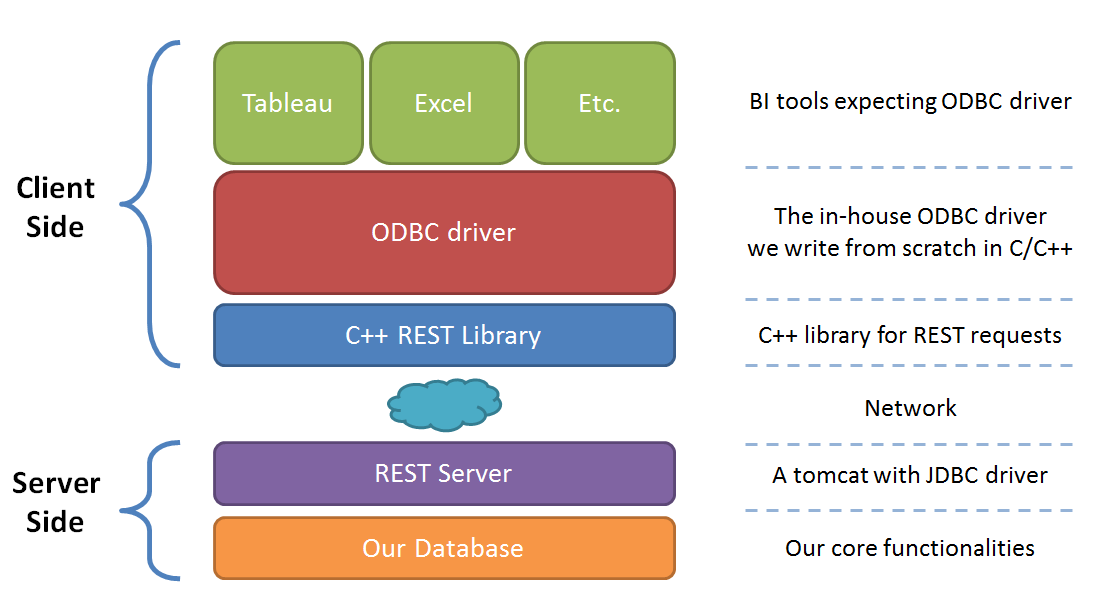
如果你的数据源恰好是类似MongoDB，Hbase这样的常见数据库产品，你或许可以考虑直接从购买一些商业产品，例如Simba ODBC Driver来一劳永逸地解决你的需求，但是将意味着不小的开支。更难办的情况是你的数据源并不是那么主流，还没有任何可以直接购买的驱动程序可以适用于它，那么定制一个自己的ODBC Driver可能是你最好的选择。即使你是一个对ODBC Driver一无所知的开发者，本文也将给你带来或多或少的帮助。

# 我们的处境

简单地说，我们团队用java开发了一个特别的SQL引擎。在项目初期我们只有JDBC驱动程序，还有一个用于服务于网页客户端的REST Server，但是我们没有ODBC 驱动，因此大多数的客户并不能真正地使用我们的产品完成他们地工作。

为了解决这个问题，我们设计了如下图的解决方案：我们使用REST Server统一地接受来自所有客户端的请求，包括网页客户端和使用ODBC Driver的客户端。REST Server中使用JDBC驱动来访问我们的数据库。当然如果你的客户端就是一个java程序，你完全可以直接通过JDBC来访问我们的数据库，从而节省这些步骤带来的开销。这张图片中并未展示这种情况。

在客户端，我们深度定制了一个专有的ODBC Driver，它向上层的应用程序提供了标准的ODBC API，封装所有实现的逻辑。在底层实现上，它调用C++的REST库，将应用程序发送过来的SQL查询请求封装成REST请求，发送给我们的REST Server，并在得到结果后，再以符合ODBC规范的方式，返回给上层的应用程序。



# 从Hello World开始

对于从来没有接触过ODBC的开发者来说，了解一个ODBC客户端的行为有助于理解定制一个ODBC驱动需要实现哪些具体的API。下图中展示了一个简单的ODBC客户端程序的实现，每一行代码都配有详细的注释解释它的行为，通读代码，不难拥有一个直观的理解。为了简化代码，我们省略了所有错误检查的代码。所有的SQLXXX格式的函数，都是ODBC定义的标准API。

我们将这段程序分成了五块区域，分别标记为A~E。A区域和B区域依次初始化了三个与ODBC相关的句柄，分别是：

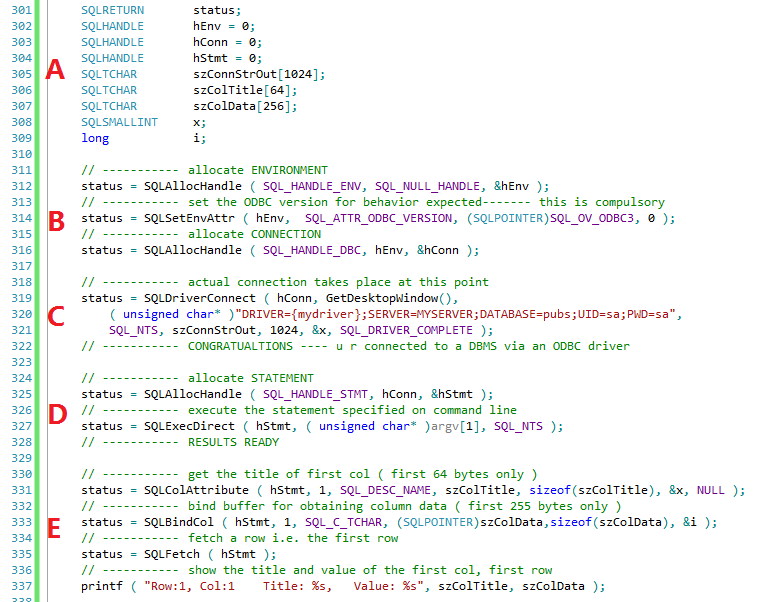
**Environment handle (hEnv):** 包含一个或者多个Connection handle。同时，一些全局的信息也包含在内，例如客户端所需要的ODBC版本，以及环境级别的诊断信息。

**Connection handle (hConn)：**代表了一个对DBMS/数据源的连接，包含了连接级别的信息，例如连接的超时时间，隔离级别，以及连接级别的诊断信息。

**Statement handle (hStmt)：**可以将它看做是某个具体的查询请求，例如 SELECT \* FROM employee。

值得一提的是ODBC规范只定义了数据源以何种方式暴露数据访问的接口，但是并没有规定如何实现，这也包括三类句柄的具体实现。事实上，在代码中这三类句柄都通过SQLHANDLE类型来传递，而SQLHANDLE本质上是一个void \*类型，指向我们自定义的相应的结构体。

ODBC为应用程序提供了一系列的C语言风格的API来支持访问查询。不同于面向对象语言的驱动程序，使用ODBC驱动程序的应用程序需要为将要返回的数据提前准备好内存区域，从这个角度说，ODBC的任务是正确地将用户需要的数据，搬运到用户指定的内存区域之中（可能带有一些数据转化，例如如果应用程序需要支持Unicode，那么ODBC Driver可能需要将char类型的源数据转化为wchar类型）。下图的A区域中初始化了一系列的句柄和变量，其中第305~307行就在程序的栈上开辟了这样一些用作缓存的内存区域。事实上，在E区域，我们传入了变量x和i的引用，因此我们可以把第308~309行的两个数值变量也看作是这样存储返回结果的内存区域。



在区域C中，我们调用SQLDriverConnect函数，同时传入hConn句柄和连接数据源所需要的用户名，密码，驱动名称等信息。我们在ODBC Driver的实现中，完成对hConn的一系列赋值操作（其初始化操作已经在B区域中完成），使得hConn成为一个可用的连接句柄。当SQLDriverConnect的返回值等于SQL\_SUCCESS的时候，一个DBMS/数据源连接就正式被建立好了。

在区域D中，客户端程序首先对Statement句柄hStmt进行了初始化，然后直接使用SQLExecDirect API进行查询。该API的第二个参数接收的字符串，即为查询请求的SQL。

在最后的E区域中，客户端获取查询结果。在这段程序中，客户端首先在第331行利用SQLColAttribute接口提取了第一列的列名属性，其中第二个参数指定返回结果的第一类，第三个常量参数SQL\_DESC\_NAME指定所需要的属性标志（列名）。接下来，客户端利用SQLBindCol接口，告知ODBC Driver它希望将第一列的返回结果填入到szColData所指向这段内存中，并且用常量参数SQL\_C\_TCHAR告知ODBC客户端希望看到的返回类型是char数据类型，这个过程被称作绑定（Bind）。一切就绪之后，客户端调用SQLFetch接口获取结果第一行的第一列，由于没有绑定其他的返回列，因此SQLFetch实际上只会返回第一列的内容。

在一个更加现实的客户端代码中，可能会首先调用SQLNumResultCols接口得知返回结果总共有多少列。对于每一个返回的列，客户端使用比SQLColAttribute接口更加便捷的SQLDescribeCol接口，一次性获取该列的所有基本信息，包括列名，类型，长度等信息。根据返回的列信息，客户端有针对性地调整SQLBindCol的参数，以便正确地接受相应的返回结果。一切就绪之后，客户端调用SQLFetch接口，得到需要的查询结果。由于每次调用SQLFetch返回结果集中的一行，客户端程序需要重复调用SQLFetch，直到SQLFetch不再返回SQL\_SUCCESS，而是返回SQL\_NO\_DATA，表示已经不再有更多的行可以返回。客户端可以根据需求的不同，考虑就究竟是复用同一块内存空间来接受结果中不同的行（取一行，使用一行），还是在一开始就申请能够容纳所有行的大块内存，每次绑定传入不同的内存位置（取完所有行后再一起使用结果数据）。

# 开始定制ODBC Driver

通过上一节对ODBC Driver所需要提供的接口有一定了解后，如果我们需要从无到有地写出一个完整的ODBC Driver，我们需要完成两项工作：

1. 实现客户端所需要的所有API，MSDN给出了ODBC规范中每一个API的详细定义（<http://msdn.microsoft.com/en-us/library/ms714562(v=vs.85).aspx>， 庆幸的是我们没必要实现每一个接口，只需要根据客户端的行为找到最小的必要集），在Windows中，我们将所有的API的实现打包成为一个可执行模块，通常是一个dll文件。
2. 让程序客户端程序能够正确地找到我们的Driver，简而言之，我们需要正确地将ODBC Driver安装到客户端程序运行的机器上。

由于在实现复杂度上，第二步明显低于第一步，另外对第二步的介绍也有助于我们能够对ODBC Driver有整体性的理解。因此虽然第二步事实上依赖于第一步的完成，我们仍然优先介绍第二步的实现。在这里我们可以假设我们已经实现了所有必须的API，这些API的实现都被包装在一个名为driver.dll的文件中。

## 第二步：安装ODBC Driver

### 理解ODBC架构

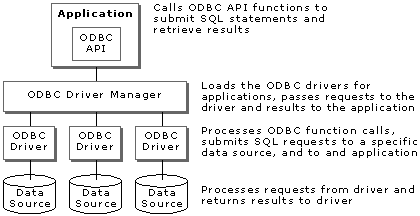
在ODBC架构中( <http://msdn.microsoft.com/en-us/library/aa266933(v=vs.60).aspx> )，有四个关键的模块，分别是：

**API**：通过调用ODBC的接口来连接数据源，发送和接受数据，以及关闭连接。这里的API仅仅是接口，并没有实现，具体的实现需要在Driver模块中完成。

**Driver Manager：** 向应用程序提供诸如可用的数据源的信息，按需动态加载驱动程序，提供参数检查等。

**Driver：**处理ODBC的函数方法，管理应用程序和特定的DBMS／数据源之间的所有交互。如果有必要，Driver还会将标准SQL格式的请求语句转为目标数据源的原生SQL格式。

**Data Source：** 由数据及其数据库引擎组成。



其中API和Driver Manager已经一般是操作系统自带的。在Window上，我们可以通过安装MDAC (Microsoft Data Access Components, <http://www.microsoft.com/en-us/download/details.aspx?id=5793>)，来获得所有所需的头文件已经相关的工具资源。在Unix环境中，也有类似的UnixODBC。在下文中我们仅考虑Windows下的ODBC开发。在开发ODBC Driver的时候，底层的数据源一般也已经就绪。因此我们仅需要将ODBC Driver注册到Driver Manager之中。

### 注册ODBC Driver

Driver Manager通过注册表得知所有可用的ODBC Driver的列表，以及它们各自的详细信息。具体位置在（假设目标机器安装了64位Windows）：

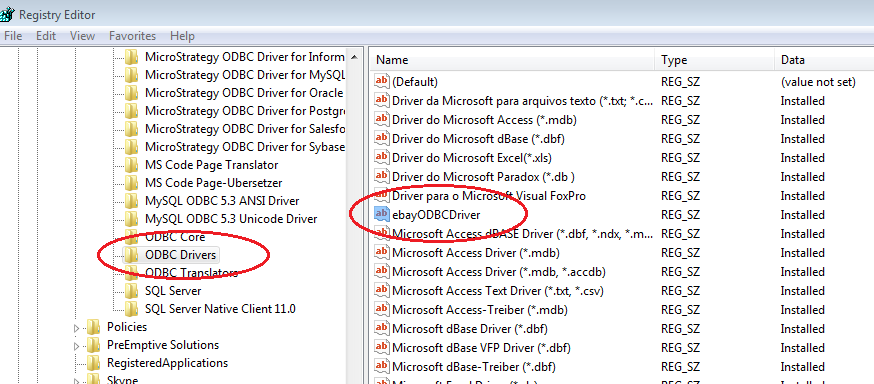
**32位 驱动：**

*HKEY\_LOCAL\_MACHINE\SOFTWARE\Wow6432Node\ODBC\ODBCINST.INI\ODBC Drivers*

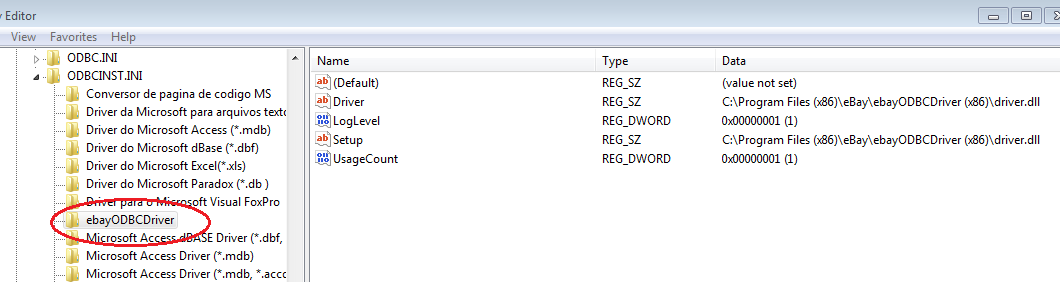
**64位 驱动：**

*HKEY\_LOCAL\_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\ODBC Drivers*

以32位Windows为例，我们打开注册表中的 ODBC Drivers键，可以看到系统中所有安装的32位ODBC的驱动程序都在其中，我们将自己的ODBC起名为ebayODBCDriver，并在ODBC Driver中加入相应的一行：

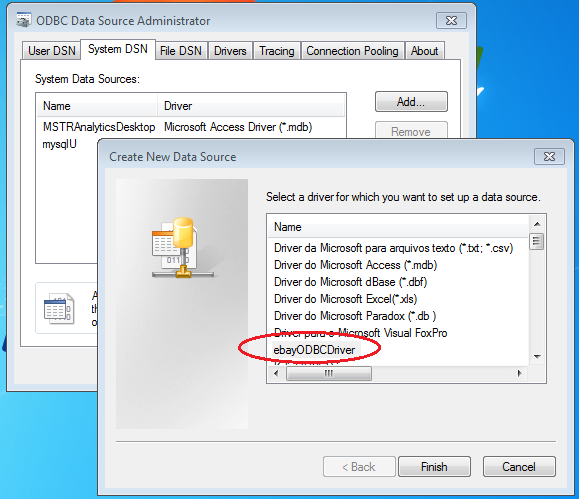


在得知ODBC Driver的名字之后，Driver Manager会在ODBC Driver的父亲节点上，也就是ODBCINST.INI中寻找相应的ODBC Driver的详细信息。必备的信息包括Driver属性和Setup属性，分别告诉ODBC Manager在哪里寻找Driver和Setup的可执行程序。其中Driver对应于我们将要实现的ODBC Driver，而Setup程序中包括了一些设置DSN的时候用到的API，根据通常的惯例，这部分的API也和ODBC Driver的API一同编译在同一个dll文件中，因此我们看到在ebayODBCDriver下，Driver和Setup指向同一个dll文件。在此也可以定义一些其他的属性，但这些都是可选的。



一切就绪，我们就能够在Control Panel--Administrator Tools--Data Sources(ODBC) 中为我们的ODBD Driver创建DSN了。对于32位的ODBC Driver而言，我们需要使用C:\Windows\SysWOW64\odbcad32.exe这个32位版本的Data Sources(ODBC)。值得一提的是在使用Data Sources(ODBC)创建DSN的过程中，我们使用到了上文提及的Setup程序中的接口，尤其是ConfigDSN接口。( <http://msdn.microsoft.com/en-us/library/ms709275(v=vs.85).aspx> )

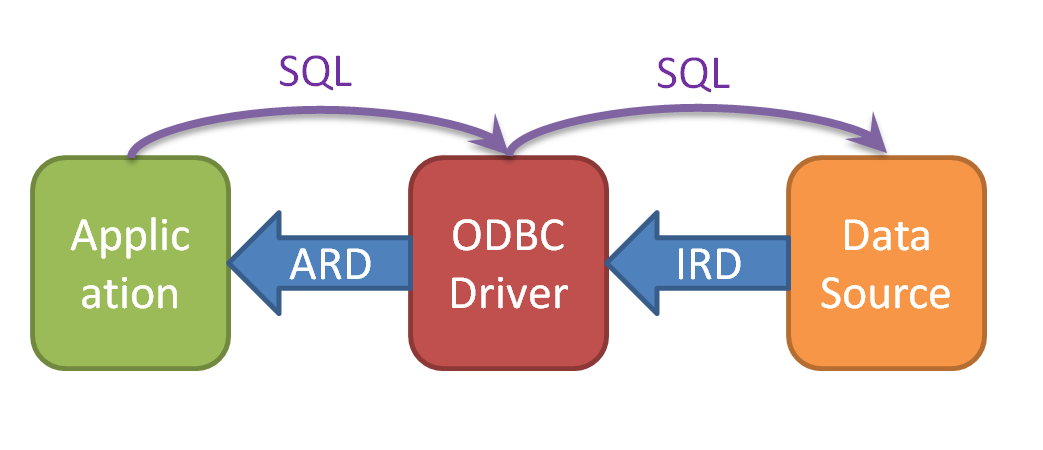
为了简化这些安装步骤，我们可以以Windows Installer的形式，包装所有这些注册ODBC Driver的逻辑，让使用者可以简单地通过安装一个exe，完成ODBC Driver的安装和注册。



## 第一步：实现ODBC API

### Descriptors

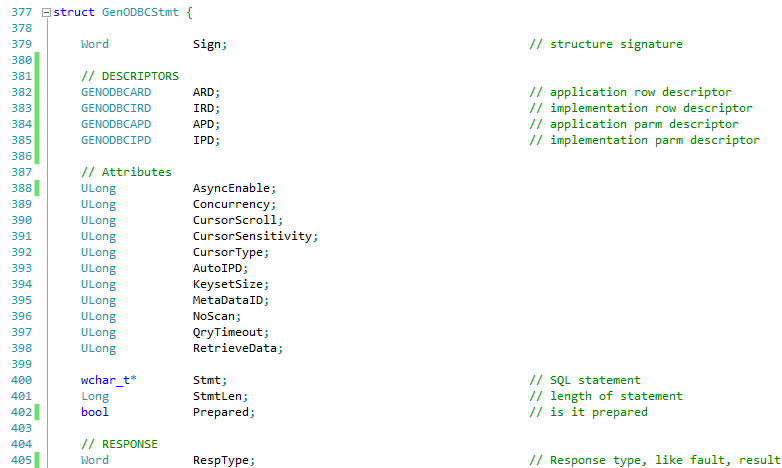
在MSDN对ODBC架构的阐述中，ODBC Driver模块的核心职能在于管理应用程序和特定的DBMS／数据源之间的所有交互。交互的载体在于数据，而有数据就意味着需要内存空间对其进行存储。前文已经提到，ODBC Driver本身定位于一个数据的搬运工，它将应用程序的请求转交给数据源，并且将数据源返回的数据结果逐行搬运给应用程序。在这个过程中ODBC Driver需要至少两块内存区域，或者简称buffer：一块用来缓存从数据源返回的结果，另外一块用来缓存移交给应用程序的结果。这两块buffer不仅包含数据本身，还包括对数据的描述。例如在返回给应用程序的数据中，ODBC Driver不仅需要维护列数据本身，还维护了该列数据的类型，长度等信息，在ODBC Driver和应用程序之间，这些数据和信息统称为Application Row Buffer Descriptor(ARD)。相应地，数据源交给ODBC Driver的也不仅仅是数据本身，还包括对每一个返回的列的元信息描述，这部分信息和数据统称为Implementation Row Buffer Descriptor(IRD)。



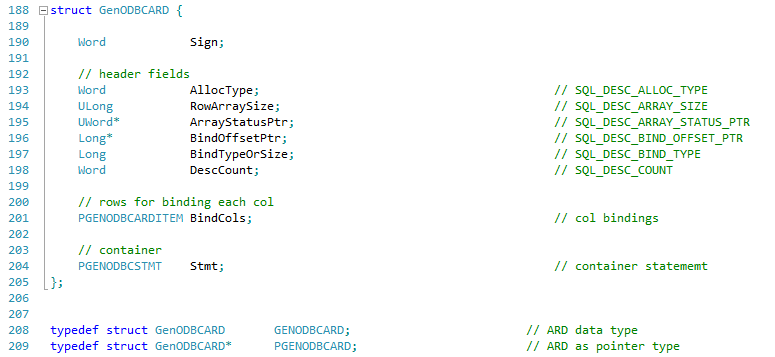
事实上ARD中保存数据本身的内存区域，是由应用程序在调用SQLBindCol的时候传入的，ARD并不负责这段内存的申请和释放，而存放其他信息所需要的内存则由ODBC Driver负责维护。当应用程序调用例如SQLNumResultCols，SQLColAttribute，SQLDescribeCol等接口的时候，ODBC Driver找到ARD中相应的内容，返回给调用者；当应用程序调用SQLBindCol和SQLFetch的时候，ODBC Driver通过ARD得知该返回数据应该被存放的位置（指针），从IRD中读取最新的一行数据，施加一些必要的据类型转化，将其搬运到指定位置。

与ARD，IRD对应的，ODBC标准还提供了另外两种buffer， 分别是Application parameter descriptor (APD) 和Implementation parameter descriptor (IPD)，用来处理动态查询中的参数，本文中对这两类buffer不做详细介绍。这四类buffer构成了ODBC世界中的四个最主要的Descriptor。更多信息，读者可以参考(<http://msdn.microsoft.com/en-us/library/ms716262(v=vs.85).aspx> )。

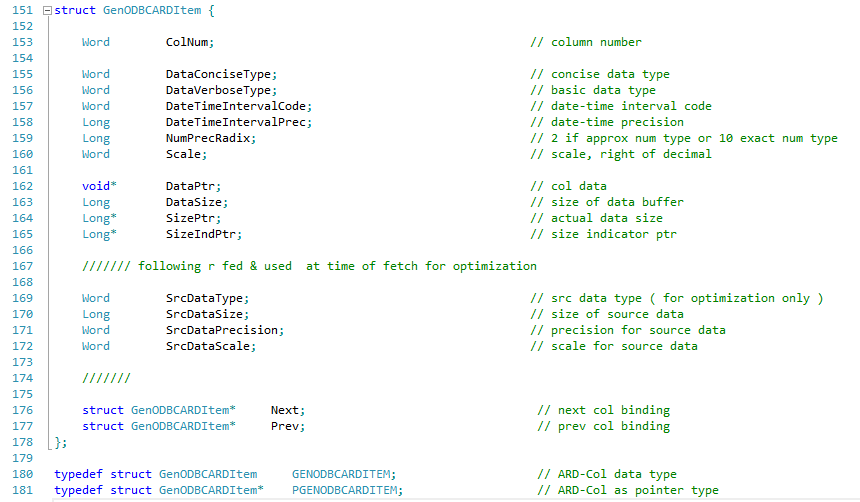
在具体的实现中，ARD和IRD被定义为特殊的结构体(struct)，存放与代表Statement的结构体GenODBCStmt(Generic ODBC Statement)之中。下图展示了我们一个GenODBCStmt结构体的部分成员：首先是标识其类型的标签（区分与代表Environment和代表Connection的结构体），然后是前文提到的四种不同用途的descriptor，接着是Statement级别的一些属性信息，SQL语句等等。



我们以ARD为例详细分析，ARD的具体实现不受ODBC规范的约束，可以自由实现。在我们的实现中，我们用结构体GENODBCARD代表一个Statement的所对应的ARD。每个ARD包括了一些所有返回列共享的信息，又包含了不同返回列的不同的详细信息，用更细力度的结构体GenODBCARDItem来代表。



观察GenODBCARDItem中的成员变量，很容易和ODBC API产生一一对应的关系。例如这里的DataConciseType对应与SQLColAttribute返回的列类型信息，而DataPtr成员则对应SQLBindCol所传入的内存空间的指针。总而言之，大多数ODBC API的实现，本质上就是对ARD和IRD的不同成员变量的访问和修改。



### 利用REST API访问数据源

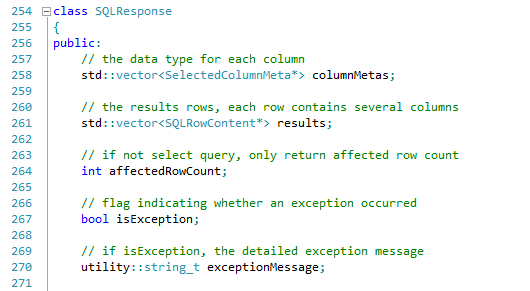
ARD负责ODBC Driver与应用程序之间的交互，其初始化在应用程序调用SQLBindCol的过程中完成。而IRD负责ODBC Driver与数据源之间的交互，其初始化需要在和数据源进行数据交换的的过程中完成。

我们首先定义REST请求的接口：

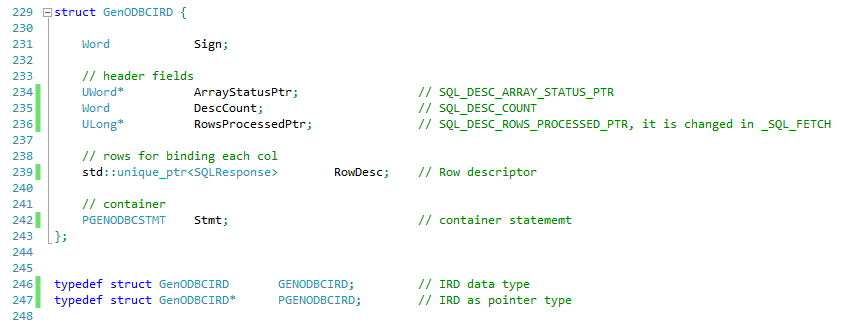
std::unique\_ptr<SQLResponse> restQuery(

wchar\_t\* rawSql, char\* serverAddr, char\* username, char\* passwd);

SQLResponse类封装一个SQL请求所有的返回内容，对于每个SQL查询，REST Server 返回一个SQLResponse的实例。该实例中的columnMetas成员包含了每一个返回列的信息，而results成员则以字符串形式保存了返回结果的每一行。



ODBC Driver将返回的SQLResponse实例交给该Statement的IRD，这样IRD在事实上拥有了该SQL查询所有的返回结果。当诸如SQLFetch的API被调用的时候，ODBC Driver只需要找到IRD中的的SQLResponse实例，对其中的信息进行解析，即可返回调用者需要的信息。



### 其他实现

#### SQLTables，SQLColumns的实现

应用程序调用这两个API来获得当前连接所能查询的所有表和列。由于这部分信息是整个数据库连接过程中公用的，我们在应用程序SQLDriverConnect，建立连接句柄的时候，就向REST Server发送获得所有表和列元信息的请求。类似于SQLResponse，我们用MetadataResponse封装这些信息，并将该返回的实例交给代表连接句柄的GenODBCConn结构体负责维护。当SQLTable，SQLColumns的请求到来时，ODBC Driver从GenODBCConn的MetadataResponse中抽取需要的信息返回给调用者。

#### SQLDriverConnect的实现

检查传入的连接字符串，如果其中明确地给出了连接数据库所需的地址，用户名，密码等信息，则直接用REST请求的方式确认REST Server存活，并且获得表和列得元信息MetadataResponse。如果上述信息不完整，则向应用程程序弹出对话框补全这些信息。当然，调用者也可以直接在连接字符串中指定已经配置好的DSN，直接完成连接。

#### SQLGetInfo的实现

当应用程序无法通过ODBC Driver的名称确认ODBC Driver的来源时，它将调用一系列的SQLGetInfo接口来获得该ODBC Driver的一些特性，例如版本，支持的函数，支持的数据类型等。Tableau等BI工具会根据这些返回结果来选择不同的行为，例如Tableau在生成SQL查询的时候无法确定表名等标志符该用单引号还是双引号来包围，它就会调用SQLGetInfo来获得SQL\_IDENTIFIER\_QUOTE\_CHAR 属性来确定。ODBC Driver实现的过程中需要通过对应用程序行为的分析，确定SQLGetInfo该返回的结果。

#### Unicode的支持

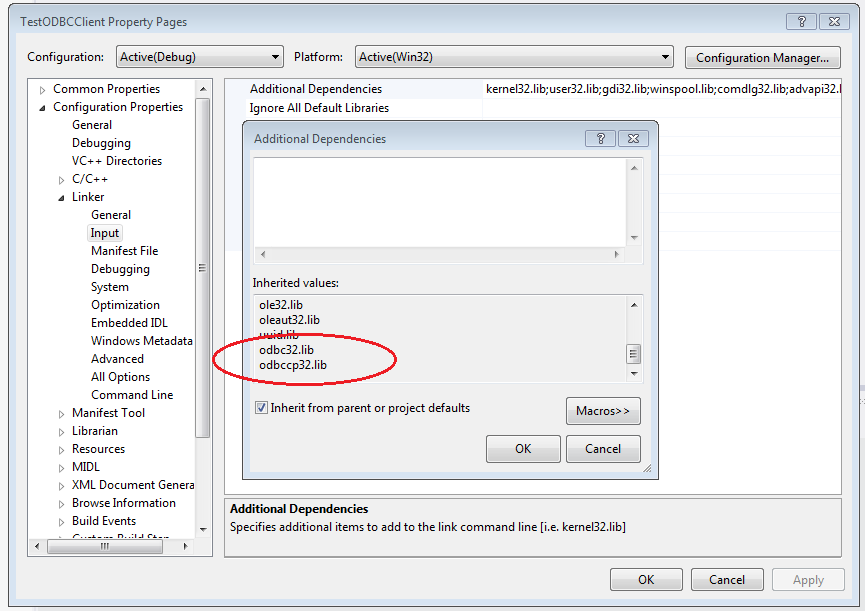
从ODBC 3.5开始，ODBC同时支持UNICODE和ANSI编码的API。ODBC使用后缀W来代表支持UNICODE的接口，例如SQLDriverConnect和SQLDriverConnectW。详情可以参考(<http://msdn.microsoft.com/en-us/library/ms716246(v=vs.85).aspx> )。

# 客户端原理及诊断

## 客户端原理

在Visual Studio中创建一个标准的ODBC客户端程序需要：

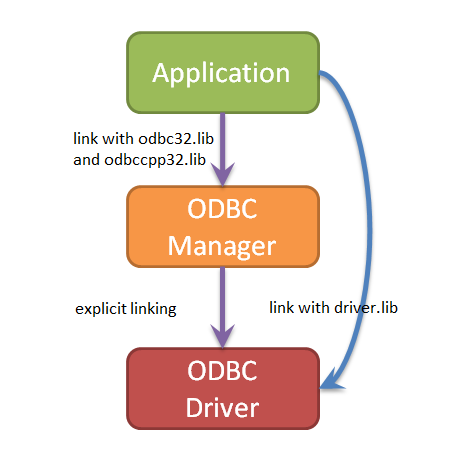
1. 引入<sqlext.h>头文件，对该文件的引用会间接引用<sql.h>, <sqltypes.h>, <sqlucode.h>等头文件，在这些头文件中包含了ODBC标准中的各种常量和方法的声明，例如SQLDriverConnect方法和SQL\_SUCCESS, SQL\_ERROR常量等。
2. 在项目的Linker中，加入对odbc32.lib和odbccpp32.lib的依赖。在Visual Studio 2012中，新创建的C++项目默认就对这两个静态链接库依赖，如下图所示：



这两部所需的头文件资源和静态链接库资源都由ODBC Manager（或者说Windows操作系统）提供。在这两步完成之后，我们就可以像在文章开始的客户端示例程序中那样，任意地调用ODBC的API。客户端程序能够成功地访问对应的ODBC驱动程序中的相应的API，归功于ODBC Manager在中间的帮助。我们可以注意到在compile和link期间，客户端程序并不与我们的ODBC驱动程序产生任何依赖，相反，客户端程序只依赖于ODBC Manager提供的头文件和静态链接库。其中的调用原理如下图紫色箭头所示：

ODBC Manager在odbc32.lib和odbccpp32.lib中给出了所有ODBC API的“实现”，这样客户端才能在不依赖于任何具体ODBC Driver的前提下完成link。然而这种“实现”事实上只是一种单纯的转发，ODBC Manger根据客户端程序中指定的驱动程序名称，将客户端的ODBC API请求转发到相应的ODBC驱动程序中。

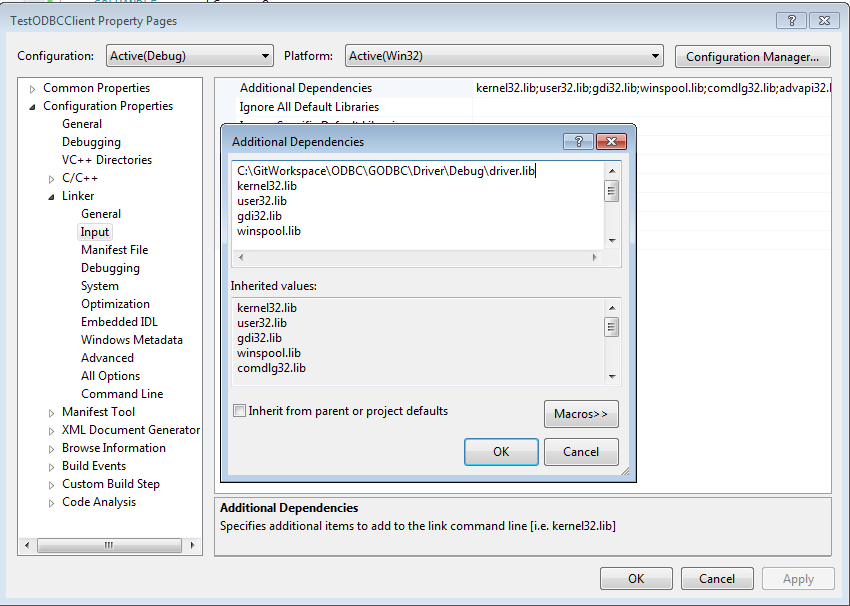
以客户端调用SQLDriverConnect 为例，由于客户端程序和odbc32.lib被link在了一起，因此程序会首先进入odbc32.lib中对SQLDriverConnect的实现之中，该实现通过explicit linking（可以理解为运行时动态的link，参考 http://msdn.microsoft.com/en-us/library/784bt7z7.aspx），可以找到并调用我们真实的ODBC驱动程序的DLL（简称driver.dll）中的SQLDriverConnect方法，于是最终客户端程序调用到了driver.dll中的SQLDriverConnect方法。



ODBC客户端程序，ODBC Manager，ODBC Driver这三者这样的角色分配可以优雅地完成客户端程序和具体驱动程序之间的解耦，保证了我们可以在系统中自由地增加和指定新的ODBC驱动程序。然而这样的设计也给开发和调试带来了麻烦：我们无法简单地在Visual Studio中以Debug模式追溯客户端调用过程中程序运行的每一步，因为客户端并不对我们的驱动程序产生显示的依赖，所以Visual Studio无法为我们找到驱动程序对应的源代码。

## 诊断客户端程序

解决方法是取消Visual Studio中的C++项目对ODBC Manger提供的静态链接库的依赖，转而直接依赖自定制ODBC驱动程序的静态链接库（简称driver.lib，该静态链接库会在我们编译driver.dll的时候作为副产品同时产生）。在Property Pages->Configuration Properties->Linker->Additional Dependencies中，我们首先取消”Inherit from parent or project defaults”选项，然后将”Inherited Values”中除了odbc32.lib和odbccpp32.lib以外的值拷入Additional Dependencies。这样保证了我们不失去其它的项目默认依赖，同时也去除了客户端对ODBC Manager的odbc32.lib和odbccpp32.lib的依赖。最后，我们在此加入driver.lib的路径，使客户端程序显示地依赖我们的ODBC驱动程序的实现。

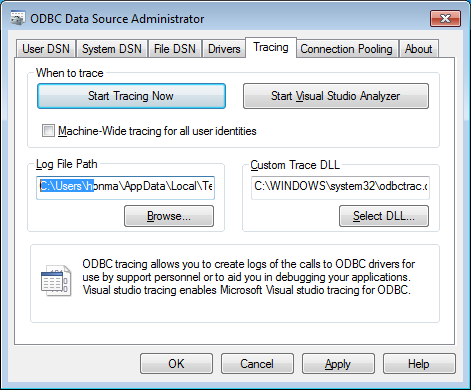


值得一提的是我们仍然需要引用头文件<sqlext.h>，在Visual Studio看来，唯一的不同之处是，头文件中的ODBC API的实现不再在odbc32.lib和odbccpp32.lib之中，而是在我们的驱动程序driver.lib之中了。Visual Studio可以根据这种认识自动定位到驱动程序的源代码，于是我们就可以在Visual Studio的Debug模式中自由地追溯程序间的函数调用了。

## 其他Debug 工具

ODBC Driver的Debug相对繁琐，除了将一些关键步骤打印到日志文件，这里还推荐另外两种方式进行补充：

1. 使用ODBC Data Source Administrator的Tracing功能捕捉ODBC Manager的日志。



1. Include windows.h, 使用OutputDebugString方法输出debug语句，配合dbmon观察程序输出。该方式可以在log日志无法工作的时候作为补充，而且在dbmon没有开启的情况下，不会对系统造成额外的开销。具体参见<http://msdn.microsoft.com/en-us/library/windows/desktop/aa363362(v=vs.85).aspx> 。

# 总结

本文详细描述了如何在已有REST服务的前提下，完成一个以REST API作为后台数据访问方式的ODBC驱动程序。本文工作能够帮助REST服务的提供团队以ODBC的形式包装其服务，使得他们的用户能够在更多的商用平台上消费他们的服务。