SYBASE®
An SAP Company

Adaptive Server Enterprise 中的 Java

Adaptive Server® Enterprise

15.7

文档 ID：DC38106-01-1570-01
最后修订日期：2011 年 9 月

Sybase, Inc., One Sybase Drive, Dublin, CA 94568。

目录

第 1 章

数据库中的 Java 简介

本章简要介绍了 Adaptive Server® Enterprise 中的 Java。

数据库中的 Java 的优点

Adaptive Server 为 Java 提供了运行时环境，这意味着可以在服务器上执行 Java 代码。在数据库服务器上为 Java 建立一个运行时环境，这将为管理和存储数据与逻辑提供强有力的新途径。

- 可以将 Java 编程语言作为 Transact-SQL 整体的一部分使用。
- 可以在应用程序的不同层 （客户端、中间层或服务器）中重复使用 Java 代码，并在您认为最有意义的任何地方使用它们。
- 对于在数据库中创建逻辑而言， Adaptive Server 中的 Java 是一种比存储过程更强大的语言。
- Java 类已成为丰富的用户定义的数据类型。
- Java 类的方法提供了可从 SQL 访问的新函数。
- 在数据库中使用 Java 不会危害数据库的完整性、安全性和健壮性。使用 Java 不会改变已有 SQL 语句的功能，也不会影响其它与 Java 无关的数据库功能。

数据库中 Java 的功能

Adaptive Server 中的 Java 允许您：

- 在数据库中调用 Java 方法
- 将 Java 类存储为数据类型
- 在数据库中存储和查询 XML

在数据库中调用 Java 方法

可在 Adaptive Server 中安装 Java 类，然后用两种方法调用这些类的静态方法：

- 可直接从 SQL 中调用 Java 方法。
- 可以在 SQL 名称中包装这些方法，并将它们当作标准的 Transact-SQL 存储过程来调用。

直接从 SQL 中调用 Java 方法

面向对象语言的方法与过程语言的函数相对应。可以通过引用的方法调用存储在数据库中的方法，调用这些方法时，需要对实例方法的实例和静态 （类）方法的实例或类进行名称限定。可直接调用方法，例如在 Transact-SQL select 列表和 where 子句中调用。

可以将向调用者返回值的静态方法用作用户定义的函数 (UDF)。

用下列方法使用 Java 方法时，有一定的限制：

- 如果 Java 方法通过 JDBC 访问数据库，则只有 Java 方法可以使用结果集中的值，客户端应用程序无法使用这些值。
- 不支持输出参数。方法可以处理通过 JDBC 连接接收的数据，但只能向其调用者返回一个返回值，该值是在其定义中声明的。

将 Java 方法作为 SQLJ 存储过程和函数来调用

在 SQL 包装中可包括 Java 静态方法，并且它们的使用方法与 Transact-SQL 存储过程或内置函数完全相同。它的功能包括：

- 允许 Java 方法将输出参数和结果集返回到调用环境。
- 允许利用传统的 SQL 语法、元数据及权限功能。
- 允许跨数据库调用 SQLJ 函数。
- 允许在服务器、客户端和任何符合 SQLJ 的第三方数据库上将现有 Java 方法用作 SQLJ 的过程和函数。
- 遵循标准规范的第 1 部分。请参见第 6 页的 “标准”。

将 Java 类存储为数据库类型

利用数据库中的 Java，可以在 SQL 系统中安装纯 Java 类，然后像平常那样将这些类用作 SQL 数据库中的数据类型。此功能采用了一种便于理解的模型和一种便于移植和获取的语言，为 SQL 添加了一个完全面向对象的数据类型扩展机制。使用此功能创建和存储的对象可以传送到任何一个启用 Java 的环境中，无论该环境是另外一个 SQL 系统还是一个独立的 Java 环境。

这种在数据库中使用 Java 类的能力有两个不同但互为补充的作用：

- 为 SQL 提供了一个类型扩展机制，可以将这种机制用于在 SQL 中创建和处理的数据。
- 为 Java 提供了一种持久的数据功能，用于在 SQL 中存储在 Java 中创建和处理（主要方面）的数据。 Adaptive Server 中的 Java 与传统的 SQL 功能相比有一个显著的优点：您无需将 Java 对象映射到标量 SQL 数据类型，或将 Java 对象存储为无类型的二进制字符串。

在数据库中存储和查询 XML

与超文本标记语言 (HTML) 相似，可扩展标记语言 (XML) 允许您定义自己的特定于应用程序的标记，因而它特别适用于数据交换。

《Adaptive Server Enterprise 中的 XML 服务》介绍 Sybase® 本机 XML 处理器和 Sybase 基于 Java 的 XML 支持以及数据库中的 XML，并提供有关构成 XML 服务的查询和映射函数的文档资料。

Java 组件

Adaptive Server 允许插入现成的商业 Java 运行时环境 (JRE) 和 Java 虚拟机 (JVM) 组件。在为 Adaptive Server 配置 Java 后，您可以包括任何支持 Java 6 或更高版本的标准 JVM。通过使用这种基础结构，您可以运行使用 Adaptive Server 15.0.3 之前版本中的 Java 解决方案配置的 Java 应用程序以及使用 Adaptive Server 15.0.3 和更高版本创建的应用程序。

Adaptive Server 的 Java 接口包含商业 JVM 和支持该 JVM 的 Sybase 组件：

- 可插入组件适配器/JVM (PCA/JVM)
- 可插入组件接口 (PCI) 和 PCI 桥，它们是 Adaptive Server 的内部组件

请参见第 2 章 “管理 Java 环境”。

Adaptive Server 15.0.3 和更高版本中的功能变化

在 Adaptive Server 15.0.3 版中， Sybase 引入了对商业 JVM 的支持，如 Sun Java 2 Platform, Standard Edition (J2SE)。 Adaptive Server 15.0.2 和更低版本提供了一个内部 JVM。

Adaptive Server PCA/JVM 确保使用 15.0.3 之前版本创建的 Java 应用程序可以与使用 Adaptive Server 15.0.3 和更高版本创建的 Java 应用程序一起无缝运行。

除了本节中介绍的变化以外，另请参见以下内容：

- 第 57 页的 “Adaptive Server 15.0.3 和更高版本中的静态变量更改”
- 第 57 页的 “Cluster Edition 中的静态变量更改”

类分发变化

Adaptive Server 15.0.2 和更低版本附带提供的 Java 运行时类是 Java 1.2 版的一个有限子集。 Adaptive Server 不再提供运行时类。相反， JVM 使用作为商业 JRE 一部分提供的运行时类。

通常，可以认为较高版本中的 Java 类与早期版本向后兼容。不过，早期版本中标有 “不建议使用”的某些方法或类可能不再与较高版本兼容。请确保应用程序所使用的任何不建议使用的类或方法在较高 Java 版本中仍受支持并保持不变。

Adaptive Server 15.0.2 和更低版本在 *$SYBASE/$SYBASE_ASE/lib* 目录中包含 *runtime.zip* 文件。该文件包含特定于 Adaptive Server 的类、驱动程序支持所需的 JDBC 类以及标准 Java 类的子集。

Adaptive Server 15.0.3 将 *runtime.zip* 文件替换为 *sybasert.jar*（其中包含 PCA/JVM 所需的 Sybase Java 类），并使用 *rt.jar* 提供标准 Java 类集。*sybasert.jar* 位于 *$SYBASE/ASE-15_0/lib/pca* 中，*rt.jar* 位于 Java 分发的 *$SYBASE/shared/<jre_directory>/lib* 中，其中 *jre_directory* 是特定于所用平台的名称。

PCA/JVM 在无人参与模式下运行

Adaptive Server 15.0.2 和更低版本提供的 Java 分发中排除了需要用户交互的类和方法。由于 PCA/JVM 使用标准类分发，因此现在可以使用这些类。为防止用户调用需要用户交互的方法，PCA/JVM 始终在无人参与模式下运行。

内存管理变化

Adaptive Server 15.0.2 和更低版本使用的内存管理系统包含三个不同的堆：全局固定堆、共享类堆和进程对象堆。 Adaptive Server 15.0.3 和更高版本使用单个 PCI 内存池。 Adaptive Server 15.0.3 忽略 15.0.2 和更低版本中的任何现有配置值。您必须使用 pci memory size 配置参数指定 PCI 子系统的总内存。请参见第 2 章 “管理 Java 环境”。

如果要从 Adaptive Server 15.0.2 和更低版本进行转换，您可能需要更改 PCI 内存池的缺省大小。商业 JVM 使用的类生命周期和碎片收集算法与 Sybase 内部 JVM 有很大差别。在适当地配置 PCI 内存池的大小后，您应该看不到行为上有任何差异。

类装载程序行为变化

在 Adaptive Server 15.0.3 和更高版本中，类装载程序行为符合在装载期间检验类的 JVM 规范。

在 Adaptive Server 15.0.2 和更低版本中，将检查对所装载的类中的其它类的引用，但不会完全解析这些引用。例如，如果类 A 引用方法中的类 B，则类装载程序不检查类 B 实际是否可用。因此，类可以成功装载，而无需满足它的所有依赖性。只有在遇到需要未满足的依赖性的方法时，才会引发异常。

在装载初始类时，所有商业 JVM 实现的类装载程序将执行完全类检验。因此，不会装载具有未满足的依赖性的类，这会引发未处理的 Java 异常，并且 Java 堆栈跟踪将该错误作为 “java.lang.NoClassDefFoundError” 列出。

这意味着，在极少数情况下，在 Adaptive Server 15.0.2 和更低版本中成功装载的类可能在 Adaptive Server 15.0.3 和更高版本中不会装载，除非提供了一整套用户类和 Java 提供的类以满足所有依赖性。

标准

ANSI SQL 标准指定用于在 SQL 中使用 Java 功能的 SQL 扩展。 Java-SQL 规范位于 SQL 标准“第 13 部分：使用 Java™ 编程语言 (SQL/JRT) 的 SQL 例程和类型”中。此标准在非正式场合称为 “SQLJ”。

Sybase 支持 Java 例程的 SQLJ 规范并提供 Java 类型的等效功能。此外，Sybase 还对标准进行了扩展。例如， Adaptive Server 允许您直接在 SQL 中引用 Java 方法和类。

数据库中的 Java：问题和解答

尽管本书是面向那些熟悉 Java 的读者，但是仍有许多有关数据库中 Java 的知识有待我们去学习。Sybase 不仅用 Java 来扩展数据库的功能，而且用数据库来扩展 Java 的功能。

无论是经验丰富的 Java 用户还是初学者，都应该阅读这一节。这一节采用问答的形式帮助用户了解 Adaptive Server 中的 Java 的基础知识。

主要功能有哪些？

后面几节详细说明了所有这些内容。通过使用 Adaptive Server 中的 Java，您可以：

- 使用任何支持 Java 6 或更高版本的商业 JVM 运行 Java。
- 直接从 SQL 语句中调用 Java 函数 （方法）。
- 用 SQL 别名包装 Java 方法并将它们当作标准的 SQL 存储过程和内置函数来调用。
- 使用内部 JDBC 驱动程序访问 SQL 数据。
- 将 Java 类用作 SQL 数据类型。
- 在表中保存 Java 类的实例。
- 通过 Adaptive Server 数据库中存储的原始数据生成 XML 格式的文档，也可以反过来，即，将 XML 文档以及从中提取的数据存储到 Adaptive Server 数据库中。
- 调试在数据库中运行的 Java 类。

如何在数据库中存储 Java 指令？

Java 是一种面向对象的语言，其指令采用类的形式。您可以在数据库外部编写 Java 指令并将其编译为已编译的类（字节码），这些类是保存 Java 指令的二进制文件。

然后，可以将已编译的类安装到数据库中，并在数据库服务器上执行这些类。

Adaptive Server 为 Java 类提供了运行时环境。您需要使用 Java 开发环境（如 Sybase PowerJ™ 或 Sun Microsystems Java 开发工具包 (JDK)）编写和编译 Java。

如何在数据库中执行 Java？

当 Adaptive Server 在执行的 SQL 语句中遇到 Java 语句时，服务器将调用 JVM 以执行该语句。如果 JVM 已运行，则会将 Java 调用转发到 JVM；如果这是第一个 Java 请求，则会自动启动 JVM。 JVM 查找并装载 Java 语句指定的类，然后执行字节码。

支持哪些 Java 虚拟机 (JVM)？

Adaptive Server Java 框架设计用于任何支持 Java 6 或更高版本的标准 JVM。Adaptive Server 15.0.3 版经认证可用于 *$SYBASE/shared* 目录中包含的 Java 6 版。使用早期版本的 Java 编译的类可以继续在较高版本的 Java 中正确运行。

什么是无人参与模式？

Adaptive Server 中的 Java 在无人参与模式下运行，这意味着不使用显示设备、键盘和鼠标。虽然标准 Java 分发中的所有类均可供用户使用，但不支持某些需要用户输入或输出设备的方法。

什么是 JDBC？

JDBC 是在 Java 中执行 SQL 的行业标准 API。

Adaptive Server 提供本机 JDBC 驱动程序。根据设计，该驱动程序在服务器上执行时可实现最佳性能，因为它不需要通过网络来通信。该驱动程序允许数据库中安装的 Java 类使用执行 SQL 语句的 JDBC 类。

如何将 Java 与 SQL 一起使用？

数据库中的 Java 设计原则是，为现有 SQL 功能提供自然的开放式扩展。

- *从 SQL 中调用 Java 操作* — Sybase 扩展了 SQL 表达式的范围以包括 Java 对象的字段和方法，以便在 SQL 语句中包含 Java 操作。
- *作为 SQLJ 存储过程和函数的 Java 方法* — 可以为 Java 静态方法创建一个 SQLJ 别名，以便将它们作为标准 SQL 存储过程和用户定义的函数 (UDF) 进行调用。
- *Java 类变为用户定义的数据类型* — 可以通过用于传统 SQL 数据类型的相同 SQL 语句来存储 Java 类实例。

您可以使用作为 Java API 一部分的类和由 Java 开发人员创建和编译的类。

什么是 Java API？

Java 应用程序编程接口 (API) 是一组由 Sun Microsystems 定义的基本功能。Java 开发人员可以使用和扩展该接口。这是使用 Java 完成各种任务的关键所在。

Java API 在它自己的范围内提供了众多功能，它是为单个用户应用程序创建的所有用户定义类的基础。

Java API 支持哪些 Java 类？

Adaptive Server 在数据库中支持所有标准 Java 类。由于数据库中的 Java 在无人参与模式下运行（请参见第 7 页的“什么是无人参与模式？”），某些需要用户输入或输出设备的方法将引发 Java 异常。

是否可以在数据库中安装用户定义的类？

您可以将自己的 Java 类安装到数据库中，例如，用户创建的 Employee 类或开发人员设计、编写并使用 Java 编译器编译的 Inventory 类。

用户定义的 Java 类可以同时包含数据以及对数据执行的方法。在数据库中安装这些类后， Adaptive Server 允许在数据库的任何部分和操作中使用这些类并执行它们的功能 （以类或实例方法的形式）。

是否可以使用 Java 访问数据？

JDBC 接口是用于访问数据库系统的行业标准。JDBC 类可用于连接到数据库，使用 SQL 语句请求数据并返回可在客户端应用程序中处理的结果。

Adaptive Server 提供了一个内部 JDBC 驱动程序，它允许数据库中安装的 Java 类使用执行 SQL 语句的 JDBC 类。

是否可以在客户端和服务器上使用相同的类？

您可以创建在企业应用程序的不同层中使用的 Java 类。可以将同一 Java 类集成到客户端应用程序、中间层或数据库中。

请注意，在不同层中使用的类或在同一层中不同时间使用的类将保持兼容或有意使其不兼容，以使行为在整个应用程序中保持一致。有关详细信息，请参见有关 java.io.Serializable 类中的 serialVersionUID 的 Java 文档。

如何在 SQL 中使用 Java 类

使用用户定义的 Java 类分为三个步骤：

1　编写或获取一组要用作 SQL 数据类型或静态方法的 SQL 别名的 Java 类。

2　在 Adaptive Server 数据库中安装这些类。

注释　Java 分发中包含的类始终可用，不需要在使用之前在数据库中安装这些类。

3 在 SQL 代码中使用这些类执行以下操作：

- 直接将静态方法作为 UDF 进行调用。
- 将 Java 类声明为 SQL 列、变量和参数的数据类型。在本手册中，将它们称为 Java-SQL 列、变量和参数。
- 引用 Java-SQL 列、变量或参数的字段或方法。
- 在 SQL 别名中包装静态方法并将其用作存储过程或函数。

在何处查找数据库中的 Java 的相关信息？

有很多关于 Java 和数据库中的 Java 的书籍。 Sun 网站上提供了最新的 Java 语言规范。

不能使用数据库中的 Java 执行的操作

Adaptive Server 是 Java 类的运行时环境，而并非是 Java 开发环境。

不能在数据库中执行以下操作：

- 编辑类的源文件 （*.*java* 文件）。
- 编译 Java 类的源文件 （*.*java* 文件）。
- 执行不支持的 Java API，例如，小程序和可见类。
- 使用 Java 本机接口 (JNI)。
- 将 Java 对象作为发送到远程过程调用或从中接收的参数。它们不能正确进行转换。

Sybase 建议不要在 Java-SQL 函数、 SQLJ 函数或 SQLJ 存储过程引用的方法中使用非最终静态变量。由于静态变量范围取决于实现情况，因此，为这些变量返回的值可能不可靠。

第 2 章

管理 Java 环境

您可以将现成的标准 Java JRE 和 JVM 组件 （如 J2SE）插入到 Adaptive Server 中。本章介绍了支持 Java 的 Sybase 组件以及如何更改缺省配置值。

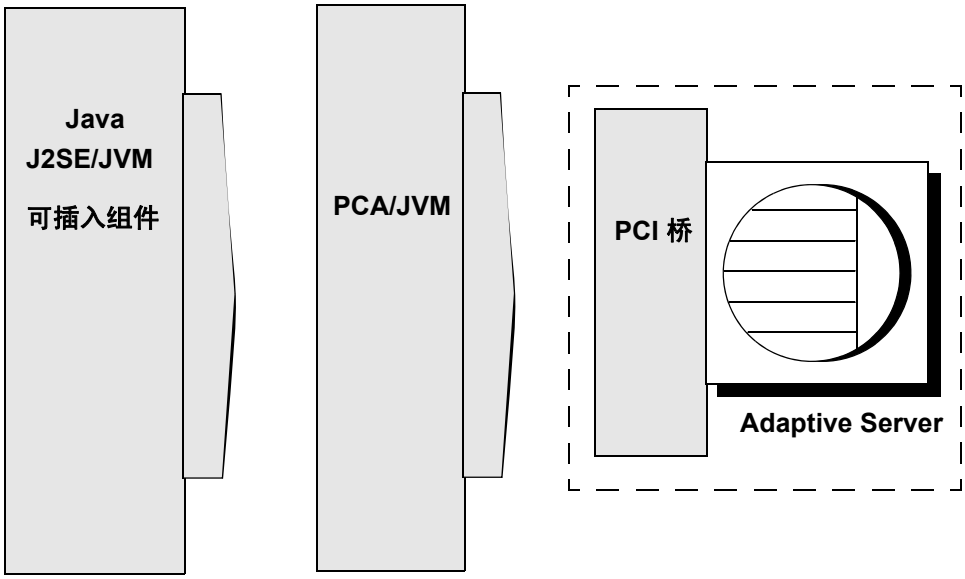
Adaptive Server Java 框架设计用于任何支持 Java 6 或更高版本的标准 JVM。ASE 15.5 经认证可用于 *$SYBASE/shared* 目录中包含的 Java 6 版。使用早期版本的 Java 编译的类可以继续在较高版本的 Java 中正确运行。

JVM 独立于 Adaptive Server。您可以更改或升级 Java 应用程序以利用发布的新 Java 功能。

Java 环境组件

图 2-1 显示了构成 Adaptive Server Java 环境的组件。

图 2-1：Java 组件



JVM 可插入组件

JVM 插件是一个动态装载的模块，它是在平台上独立于 Adaptive Server 进行工程、支持和安装的。对 Adaptive Server 而言，该插件是 “黑箱” 应用程序，而不是 Sybase 支持的技术：JVM 插件发出 Java 结果集，PCI 桥对其进行转换，然后将转换的结果集发送到 Adaptive Server。

由于 JVM 插件是由 PCA/JVM 控制的，因此，它间接连接到 Adaptive Server。您可以独立于 Adaptive Server 安装、升级和启动 JVM 插件。

通常，Java 分发包含一个或多个 JVM 实现。这样，用户就可以选择最符合各个应用程序的性能要求的 VM。

- 客户端应用程序 — 在通常用于客户端应用程序的平台上，JRE 包含调优的 VM 以缩短启动时间并减少使用的内存量。
- 服务器应用程序 — 在所有平台上， JRE 包含的 JVM 版本旨在最大限度提高应用程序执行速度。

有很多种 Java 分发，不过，Java 技术的这些功能对客户端和服务器 VM 版本是相同的：

- Adaptive 编译器 — Java 插件使用标准解释器启动应用程序，但在运行代码时对其进行分析以检测性能瓶颈或 “热点”。
- 快速内存分配和碎片收集 — Java 技术为对象提供了快速内存分配功能，并提供了一组快速、有效且非常先进的碎片收集器供您选择。
- 线程同步 — Java 编程语言允许使用多个并发路径来执行程序（称为 “线程”）。Java 技术包含一项线程处理功能，可轻松扩展该功能以用于大型共享内存多处理器服务器。

注释　应小心使用生成子线程的方法。在 Java 方法中启动的 java.lang.Thread 对象是在运行时调度的，而不是由 Adaptive Server 调度程序调度的。如果这些线程是处理器密集型线程，或者生成了大量线程，服务器性能可能会由于争用处理器时间而下降。

虽然 PCA/JVM 插件可以使用客户端或服务器 JVM，但 Sybase 建议您缺省使用服务器版本以最大限度提高 Java 方法的性能；安装过程将使用服务器版本。

有关适用于您的企业的相应客户端版本的信息，请参见客户端版本文档。

可插入组件适配器/JVM (PCA/JVM)

PCA/JVM 用作一个代理以管理 Adaptive Server 和 JVM 之间的服务请求。PCA/JVM 在两个方向转发和控制请求：从 Adaptive Server 到 JVM 以及从 JVM 到 Adaptive Server。

可插入组件接口 (PCI) 和 PCI 桥

PCI 是 Adaptive Server 的内部通用接口；缺省情况下，在安装或升级 Adaptive Server 时将安装该接口。PCI 桥（PCI 的内部组件）在 Adaptive Server 和 JVM 插件之间执行实际操作。

PCI 桥提供了以下功能：

- 本机线程 （进程）管理
- 内存管理
- 同步 （锁定、条件和事件）管理

- 数据访问服务支持
- 配置管理
- 通过自动装载插件按需分配功能
- 信号和异常处理
- 平台运行时支持
- 动态设备功能
- 将错误消息传送到 Adaptive Server 错误日志

大多数情况下，使用缺省 PCI 桥配置就足够了。如有必要，您应该咨询 Sybase 技术支持人员并使用 sp_pciconfig 系统存储过程修改 PCI 配置。sp_pciconfig 包含一些参数，可用于列出、报告、启用或禁用 sybpcidb 中的指令和参数。请参见第 18 页的 “在运行的服务器中更改配置值”。

PCI 内存池

PCI 内存池是在 PCI 桥初始化时一次性全部分配的；以后，其大小不会增加。它是由 Adaptive Server 控制的，其它内存池受到的限制也同样适用，例如，单个分配不能超过 1MB。PCI 内存池的缺省大小为 32,768 KB。

在为服务器配置 Java 时，可以使用 enable pci 配置参数启用 PCI 内存池。请参见针对所用平台的安装指南。

更改 PCI 内存池的大小

PCI 内存池的缺省大小适用于大多数非集群安装。要增加内存池的大小，请重置 pci memory size 配置参数。

例如，要将 pci memory size 设置为 13800 页 （每页为 2KB），请输入：

```
sp_configure "pci memory size", 13800
```

pci memory size 是一个动态配置参数；无需重新启动 Adaptive Server 即可使更改生效。

如果 Adaptive Server 没有足够的可用内存分配给内存池，则会忽略该配置更改并且不会启动 PCI 桥。

有关 pci memory size 的详细信息，请参见 《系统管理指南，卷 1》。

多引擎 Adaptive Server 中使用的 Java VM 内存

在多引擎环境中，多个 Adaptive Server 任务可能会并行使用 Java VM。结果， Java VM 在多引擎环境中需要的内存比单引擎环境多。因此，您可能需要根据所运行的应用程序类型以及并行执行 Java 的用户数，增加 PCI 内存池的大小。

可以让 Adaptive Server 计算堆大小，您也可以自己使用 sp_jreconfig 设置 PCA_JVM_JAVA_OPTIONS 指令的 -Xmx 和 -Xms 参数以配置堆大小。

要让 Adaptive Server 配置堆大小，计算的堆大小必须大于 4MB，并且不能设置 -Xmx 和 -Xms 参数（Adaptive Server 使用 sybpcidb 中存储的值）。

当 Adaptive Server 配置堆大小时：

- 设置 PCA_JVM_JAVA_OPTIONS 指令的 -Xmx 参数，以使 Java 堆大小为 PCI 内存池大小的 65%。
- 将 -Xms 参数设置为与 -Xmx 相同的值。
- 为新生成的堆配置 20% 的 Java 堆大小，也称为 Eden 区。

sybpcidb 数据库

sybpcidb 数据库存储 PCI 桥和 PCA/JVM 插件的配置信息。在为服务器配置 Java 时，可以创建 sybpcidb，安装其中的表并创建其系统存储过程。请参见针对所用平台的安装指南。

sybpcidb 系统存储过程包括：

- sp_pciconfig — 配置 PCI 桥属性。
- sp_jreconfig — 配置 PCA/JVM 插件属性。

sybpcidb 表

sybpcidb 数据库中包含以下表。

用户表	内容
pci_directives	PCI 桥的指令配置信息。
pci_arguments	PCI 桥的参数配置信息。
pci_slotinfo	每个槽的信息，其中包括相关指令和参数的表名。
pci_slot_syscalls	PCI 桥使用的运行时分配模式的运行时系统调用配置信息。
pca_jre_directives	特定于 PCA/JVM 插件的指令信息。
pca_jre_arguments	特定于 PCA/JVM 插件的参数信息。

有关 sybpcidb 的详细信息，请参见 《参考手册：表》。有关 sp_pciconfig 和 sp_jreconfig 的详细信息，请参见 《参考手册：过程》。

如何在 *sybpcidb* 中组织配置值

PCI 桥和 PCA/JVM 的配置值存储在 sybpcidb 中，并按照指令和参数层次进行划分。每个指令包含一个或多个参数；每个参数保存一个配置值。参数类型如下：

- “开关”参数 — 描述只能启用或禁用的属性。开关参数不包含任何数据。（PCI 桥和 PCA/JVM）
- “数字”参数 — 包含数字属性值。（PCI 桥和 PCA/JVM）
- “字符串”参数 — 包含字符串属性值。（仅限 PCA/JVM）
- “数组”参数 — 包含一个或多个字符串属性值的集合。（仅限 PCA/JVM）

您可以启用或禁用每个指令及其每个参数。指令状态将覆盖其参数状态。例如，假设指令包含三个参数：“arg1”（已启用）、“arg2”（已禁用）和 “arg3”（已禁用）。

- 如果启用了指令，则每个参数将保持其基本状态。也就是说，启用“arg1”并禁用 “arg2”和 “arg3”。
- 如果禁用了指令，指令的禁用状态将覆盖参数的基本状态，从而将“arg1”、“arg2”和 “arg3”全部禁用。
- 不过，如果重新启用了指令，每个参数将恢复其基本状态：启用“arg1”并禁用 “arg2”和 “arg3”。通过采用这种设置，您可以通过一个命令禁用所有参数，或将所有参数恢复为原始状态。

何时更改配置值

对于大多数安装，使用服务器以及 PCI 桥和 PCA/JVM 的缺省配置选项就足够了。虽然您可以自己安全地更改和管理几个配置选项，但对于大多数配置选项，应在 Sybase 技术支持人员的指导下进行更改。

您可以设置配置选项：

- 在服务器级
- 针对 PCI 桥
- 针对 PCA/JVM

服务器级选项

可以使用 sp_configure 更改和管理以下服务器级配置参数：

- enable pci — 启用 PCI 桥。
- enable java — 在数据库中启用 Java。
- pci memory size — 设置 PCI 内存池的最大大小。

注释　在使用 PCA/JVM 之前，必须启用 Java 和 PCI 桥。

请参见针对所用平台的安装指南和 《系统管理指南，卷 1》。

PCI 桥的配置选项

不要更改 PCI 桥的任何配置选项（指令或参数），除非 Sybase 技术支持人员要求这样做。

PCA/JVM 的配置选项

您可以安全地更改 PCA/JVM 的以下参数：

- pca_jvm_module_path — 只有在使用的 JRE 不是安装提供的 JRE 时，才能更改该属性。如果是这种情况，请将该属性指向 PCA/JVM 使用的 JRE。
- pca_jvm_work_dir — 根据需要，在可使用特定权限掩码配置的每个工作目录（可信目录）的该参数数组中添加一个条目。请参见第 8 章“使用 Java 访问文件和网络”。
- pca_jvm_netio — 启用此参数以启用网络 I/O。禁用此参数可禁用网络 I/O。
- pca_jvm_dbg_agent_port — 启用此参数，并将其数值设置为 JVM 用于调试代理的端口号。 Java 调试程序必须监听相同的端口。
- pca_jvm_java_dbg_agent_suspend — 启用此参数以在挂起状态下启动调试代理。启用此参数是非常有用的，因为在将 Java 调试程序连接到运行的进程后，这可为您留出一些时间在调试程序中设置断点和其它选项。请参见 《参考手册：命令》。

注释 应小心使用 pca_jvm_java_dbg_agent_suspend。启用 pca_jvm_java_dbg_agent_suspend 将导致 JVM 挂起；所有 Adaptive Server Java 任务将一直等到连接调试程序并通知 JVM 继续为止。Sybase 建议您启动 JVM 并运行简单的 Java 命令，以便连接调试程序而不是启用 pca_jvm_java_dbg_agent_suspend。使用 Java 命令可启动 JVM，并且可以在执行要调试的类之前连接调试程序。

应在 Sybase 技术支持人员的指导下更改 PCA/JVM 的任何其它指令或参数。

在运行的服务器中更改配置值

如果要更改缺省配置值，您应该咨询 Sybase 技术支持人员并使用 sp_jreconfig 和 sp_pciconfig 系统存储过程。请参见第 16 页的 “何时更改配置值”。本节介绍了如何将更改的配置值装载到运行的服务器的内存中。

在 Adaptive Server 启动时，如果为服务器配置了 Java，它将自动装载 JVM。不过，直到收到第一个 Java 请求后，才会初始化 JVM。这取决于使用 Java 的频率。在初始化之前更改配置值相对简单一些。在初始化之后，配置信息已读取到内存数据结构中，此时更改配置值会比较困难。

可通过以下方法更新配置信息：

- 重新启动 Adaptive Server
- 在初始化 JVM 之前 （仅限 PCA/JVM 插件）
- 在初始化 JVM 之后 （仅限 PCA/JVM 插件）

重新启动 Adaptive Server 以更改配置值

这是更改配置信息的最简单方法，并且始终可以使用该方法。

注释 如果使用 sp_pciconfig 更改 PCI 桥的配置值，则必须使用该方法。

1 使用 sp_jreconfig 或 sp_pciconfig 更改配置值。

2 重新启动 Adaptive Server。

在初始化 JVM 之前更改配置值

在运行 Adaptive Server 但未初始化 JVM 时，可以使用该方法更改 PCA/JVM 插件的配置值。

1　使用 sp_jreconfig 更改配置值。

2　将配置参数装载到内存中。请输入：

```
sp_jreconfig "reload_config"
```

无需重新启动 Adaptive Server 即可使新配置值生效。

注释　只有在*尚未*初始化 JVM 时，使用 sp_jreconfig "reload_config" 进行的更改才会生效。使用 sp_jreconfig 仅修改 sybpcidb 中的表值，而不会影响在启动 Adaptive Server 时装载到内存中的当前内存数据结构。

您可以安全地尝试该方法，即使您不确定是否初始化了 JVM。如果已初始化 JVM， reload_config 命令将失败并显示一条错误消息。这不会产生任何不利后果。

在初始化 JVM 之后更改配置值

如果 Adaptive Server 正在运行并且您已初始化 JVM，则配置参数位于内存中，您可以更改 PCA/JVM 插件的配置参数。

为更改 JVM 配置值而执行的步骤取决于是将 Adaptive Server 配置为线程化模式还是进程模式。

- 线程化模式 —

  a　使用 sp_jreconfig 更改配置值。

  b　重新启动 Adaptive Server。

  Adaptive Server 重新启动后， JVM 将处于未初始化状态，直至收到第一个 Java 请求为止。

- 进程模式 —

  a　使用 sp_jreconfig 更改配置值。

  b　将运行 JVM 的引擎脱机 （此示例中为引擎 3）：

  ```
  sp_engine "offline", 3
  ```

  c　将运行 JVM 的引擎恢复联机：

  ```
  sp_engine "online", 3
  ```

Adaptive Server 在此过程期间继续运行，但将运行 JVM 的引擎联机后才能使用 Java。将引擎联机后， JVM 将重新处于未初始化状态，直至收到第一个 Java 请求为止。

将缺省配置值恢复为 *sybpcidb*

在初始化 JVM 之后，将缺省配置值恢复为 sybpcidb 配置值的步骤取决于是否可以重新启动 Adaptive Server，以及是使用单引擎还是多引擎 Adaptive Server。

如果使用的是单引擎 Adaptive Server：

1 重新安装 *installpcidb* 安装脚本以将 sybpcidb 配置表值重置为出厂缺省值。例如：

```
isql -Usa -Psa_password -Sserver_name
-i $SYBASE_ASE/scripts/installpcidb
```

2 重新启动 Adaptive Server。在 JVM 初始化以响应第一个 Java 请求时，缺省配置值才会生效。

如果使用的是多引擎 Adaptive Server：

1 重新安装 *installpcidb* 以将 sybpcidb 配置表值重置为出厂缺省值。例如：

```
isql -Usa -Psa_password -Sserver_name
-i $SYBASE_ASE/scripts/installpcidb
```

2 将运行 JVM 的引擎脱机。例如：

```
sp_engine "offline", 3
```

在此示例中， JVM 在引擎 “3” 中运行。

3 将运行 JVM 的引擎恢复联机。例如：

```
sp_engine "online", 3
```

无需重新启动 Adaptive Server 即可使新配置值生效。

使用监控表显示有关 PCI 桥的信息

可以使用以下监控表显示有关 PCI 桥的信息：

- monPCIBridge — 显示有关 PCI 桥的一般信息。

  例如：

```
select * from monPCIBridge
```

```
Status  ConfiguredSlots  ActiveSlots  ConfiguredPCIMemoryKB  UsedPCIMemoryKB
------------------------------------------------------------------------------
ACTIVE                1            1                  65668             1613
```

- monPCISlots — 显示有关绑定到每个槽的插件的信息。例如：

```
select * from monPCISlots
```

```
Slot     Status     Modulename    Engine
-----------------------------------------
   1     IN USE         PCA/JVM          0
```

- monPCIEngine — 显示 PCI 桥及其插件的引擎信息。例如：

```
select * from monPCIEngine
```

```
Engine  Status  PLBStatus  NumberofActiveThreads PLBRequest  PLBWakeupRequests
------------------------------------------------------------------------------
     0 PCA ACTIVE  ACTIVE                     10          4                  4
     1 PCA ACTIVE  ACTIVE                      4          0                  0
```

有关详细信息，请参见 《参考手册：表》。

第 3 章

在数据库中准备和维护 Java

本章介绍 Java 的运行时环境，并说明如何在服务器上启用 Java 以及如何在数据库中安装和维护 Java 类。

Java 运行环境

Adaptive Server 的 Java 运行时环境需要使用第三方 JVM（即 Sybase PCI，作为数据库服务器的一部分提供）和 Sybase 运行时 Java 类（即 Java API）。如果在客户端运行 Java 应用程序，则可能还需要在客户端安装 Sybase JDBC 驱动程序 jConnect。

数据库中的 Java 类

可以使用 Java 类的任何以下来源：

- 位于 *rt.jar* 中的标准 Java 分发以及安装在 Java 安装目录下的“ext”目录中的类。

  注释 ext 目录内容可能随 Java 供应商不同而发生变化。有关这些类的详细信息，请参见供应商文档。

- 用户定义的类。

Sybase 运行时 Java 类

Sybase 运行时 Java 类是安装到支持 Java 的数据库中的底层类。在安装 Adaptive Server 时，将会自动下载这些类，以后可以从 *$SYBASE /$SYBASE_ASE/lib/sybasert.jar* (UNIX) 或 *%SYBASE%\%SYBASE_ASE%\lib\sybasert.jar* (Windows) 中使用这些类。在 JVM 启动时， Adaptive Server 将设置 CLASSPATH 环境。

注释 如果 CLASSPATH 是在操作系统环境中设置的，在内部 JVM 启动时， Adaptive Server 将*忽略* 该值。

用户定义的 Java 类

可以使用 installjava 实用程序将用户定义的类安装到数据库中。安装后，即可从数据库中的其它类和 SQL 中以用户定义的数据类型形式使用它们。

JDBC 驱动程序

Adaptive Server 附带的 Sybase 本机 JDBC 驱动程序支持 JDBC 1.1 和 1.2 版，并且与 JDBC 2.0 版的一些类和方法兼容。有关支持和不支持的类和方法的完整列表，请参见第 9 章 “其它主题”。

如果系统要求在客户端使用 JDBC 驱动程序，则必须使用支持 JDBC 2.0 版的 jConnect 6.x 或更高版本。

JVM

Adaptive Server Java 框架设计用于任何支持 Java 6 或更高版本的标准 JVM。 Adaptive Server 15.0.3 版经认证可用于 *$SYBASE/shared* 目录中包含的 Java 6 版。使用早期版本的 Java 编译的类可以继续在较高版本的 Java 中正确运行。

启用 Java

注释　在启用 PCI 和 Java 之前，请按照针对所用平台的安装指南所述配置 sybpcidb。

要在服务器及其数据库中启用 Java，请在 isql 中输入以下命令：

```
sp_configure "enable pci", 1
sp_configure "enable java", 1
```

然后关闭并重新启动服务器。 Adaptive Server 15.0.3 和更高版本要求将启用 PCI 作为启用 Java 的先决条件。

缺省情况下， Adaptive Server 不启用 Java。只有在服务器中启用了 Java 时，才能安装 Java 类或执行任何 Java 操作。

在数据库中安装 Java 类

要通过客户端操作系统文件安装 Java 类，请从命令行中使用 installjava (UNIX) 或 instjava (Windows) 实用程序。

有关这些实用程序的详细信息，请参见 Adaptive Server Enterprise Utilities Guide （《Adaptive Server Enterprise 实用程序指南》）。这两个实用程序执行相同的任务，为简单起见，本文档使用 UNIX 示例。

使用 *installjava*

installjava 将解压缩的 JAR 文件复制到 Adaptive Server 系统中，并使 JAR 中包含的 Java 类可以在当前数据库中使用。语法为：

installjava
-f *file_name*
[-new | -update]
[-j *jar_name*]
[-S *server_name*]
[-U *user_name*]
[-P *password*]
[-D *database_name*]
[-I *interfaces_file*]
[-a *display_charset*]
[-J *client_charset*]

[-z *language*]
[-t *timeout*]

例如，要安装 *addr.jar* 文件中的类，请输入：

```
installjava -f "/home/usera/jars/addr.jar"
```

–f 参数指定包含 JAR 的操作系统文件。必须使用 JAR 的完整路径名。

本节介绍了保留的 JAR 文件 （使用 -j）以及如何更新安装的 JAR 和类（使用 new 和 update）。有关这些选项和可用于 installjava 的其它选项的详细信息，请参见 《实用程序指南》。

注释 安装 JAR 文件时，应用程序服务器会将文件复制到一个临时表中，然后从该处安装 JAR 文件。如果安装较大的 JAR 文件，则可能需要使用 alter database 命令扩展 tempdb 大小。

安装解压缩的 JAR

installjava 和 instjava 工具需要使用解压缩的 jar 文件。

要在数据库中安装 Java 类，请解压缩并保存 JAR 文件中的类或包。要创建包含 Java 类的解压缩 JAR 文件，请使用 Java jar cf0 （“零”）命令。

在以下 UNIX 示例中，jar 命令创建一个解压缩的 JAR 文件，其中包含 jcsPackage 目录中的所有 *.class* 文件：

```
jar cf0 jcsPackage.jar jcsPackage/*.class
```

保留 JAR 文件

在数据库中安装 JAR 时，服务器将分解 JAR，然后提取并单独存储其中的类。除非指定具有 -j 参数的 installjava，否则不会将 JAR 存储在数据库中。

使用 -j 可确定 Adaptive Server 系统是保留 installjava 中指定的 JAR，还是仅使用 JAR 提取要安装的类。

- 如果指定 -j 参数， Adaptive Server 将按常规方式安装 JAR 中包含的类，然后保留 JAR 及其与已安装类的关联。
- 如果未指定 -j 参数，Adaptive Server 将不保留类与 JAR 的任何关联。这是缺省选项。

Sybase 建议指定一个 JAR 名称，以便能更好地管理安装的类。如果保留 JAR 文件：

- 可以使用 remove java 语句同时删除 JAR 及其关联的所有类。否则，必须一次一个地删除类或类的包。
- 可以使用 extractjava 将 JAR 下载到操作系统文件中。请参见第 29 页的 “下载安装的类和 JAR”。

更新安装的类

installjava 的 new 和 update 子句指示，您是否希望将当前安装的类替换为新类。

- 如果指定 new，则不能安装与现有类名称相同的类。
- 如果指定 update，则可以安装与现有类名称相同的类，并且新安装的类将替换现有类。

警告！ 如果通过重新安装类的修订版本的方法来改变已用作列数据类型的类，则需要确保修改后的类可以读取和使用将它用作数据类型的表中的现有对象 （行）。否则，不重新安装最初的类，就可能无法访问现有对象。

使用新类替换已安装的类还取决于是正在安装的类还是已安装的类与 JAR 关联。因此：

- 如果更新 JAR，则现有 JAR 中的所有类都将被删除，而代之以新的 JAR 中的类。
- 类只能与单个 JAR 关联。如果某 JAR 中已安装了同一名称的类，并且该类已与另一个 JAR 关联，则不能安装该类。同样，如果已安装某个类并且该类与 JAR 关联，则不能安装与 JAR 不关联的类。

  但是，当已安装的类与 JAR 不关联时，可以在保留的 JAR 中安装与该类同名的类。在这种情况下，将删除与 JAR 不关联的类，并将同名的新类与新的 JAR 关联。

如果要在新 JAR 中重新组织已安装的类，比较容易的方法可能是先解除受影响的类与其 JAR 的关联。有关详细信息，请参见第 30 页的 “保留类”。

引用其它 Java-SQL 类

已安装的类可以引用同一 JAR 文件中的其它类和同一数据库中以前安装的类，但不能引用其它数据库中的类。

如果 JAR 文件中的类引用未定义的类，则会发生错误：

- 如果在 SQL 中直接引用未定义的类，则会导致 “未定义类”语法错误。
- 如果在调用的 Java 方法中引用未定义的类，则会抛出可能在调用的 Java 方法中捕获的 Java 异常，或者导致第 39 页的 “Java-SQL 方法中的异常”中所述的一般 SQL 异常。

只要不动态引用或调用不受支持的类和方法，类定义就可包含这些引用。同样地，只要不对类进行实例化或引用，则已安装的类就可以包含在不同数据库中安装的用户定义类的引用。

查看有关安装的类和 JAR 的信息

要查看有关数据库中安装的类和 JAR 的信息，请使用 sp_helpjava。语法为：

```
sp_helpjava [‘class’ [, name [, 'detail' | , 'depends' ]] |
    ‘jar’ [, name [, 'depends' ] ]]
```

例如，要查看有关 Address 类的详细信息（包括版本号），请登录到 isql 并输入：

```
sp_helpjava ’class’, Address, detail
```

有关详细信息，请参见 《参考手册》中的 “sp_helpjava”。

下载安装的类和 JAR

可以下载已安装在一个数据库中的 Java 类的副本，以便在其它数据库或应用程序中使用。

使用 extractjava 系统实用程序可将 JAR 文件及其类下载到客户端操作系统文件中。例如，要将 *addr.jar* 下载到 *~/home/usera/jars/addrcopy.jar* 中，请输入：

```
extractjava -j 'addr.jar' -f
    '~/home/usera/jars/addrcopy.jar'
```

有关详细信息，请参见 《实用程序指南》手册。

删除类和 JAR

可以使用 Transact-SQL remove java 语句从数据库中卸载一个或多个 Java-SQL 类。 remove java 可以指定一个或多个 Java 类名、 Java 包名或保留的 JAR 名称。例如，要卸载 utilityClasses 包，请从 isql 中输入：

```
remove java package "utilityClasses"
```

注释　Adaptive Server 不允许删除用作列和参数的数据类型的类，或由 SQLJ 函数或存储过程引用的类。无法检查其它类的使用情况，可以在存储过程中仍引用这些类的同时将其删除。请确保没有删除用作变量或 UDF 返回类型的子类或类。

remove java package 可删除指定包及其所有子包中的所有类。

有关 remove java 的详细信息，请参见 《参考手册》。

保留类

您可以从数据库中删除 JAR 文件，但将其中的类保留为不再与 JAR 关联的类。例如，如果要重新组织几个保留的 JAR 的内容，可以将 remove java 与 retain classes 选项一起使用。

例如，从 isql 中输入：

```
remove java jar 'utilityClasses' retain classes
```

在解除类与其 JAR 的关联后，可以使用 installjava 和 new 关键字将其与新 JAR 相关联。

第 4 章

在 SQL 中使用 Java 类

本章介绍如何在 Adaptive Server 环境下使用 Java 类。前面的几节提供必要的入门信息；后面的几节会提供一些高级内容。

在本文档中，数据类型是 Java-SQL 类的 SQL 列和变量分别称作 Java-SQL 列和 Java-SQL 变量，或者统称为 Java-SQL 数据项。

一般概念

本节提供了 Java 和 Java-SQL 标识符的常规信息。

使用 Java 需要考虑的事项

在 Adaptive Server 数据库中使用 Java 之前，需要考虑以下的常规问题。

- Java 类包含：
  - 已声明了 Java 数据类型的字段。
  - 其参数和结果具有已声明的 Java 数据类型的方法。
  - 有对应的 SQL 数据类型的 Java 数据类型在第 146 页的 “Java 和 SQL 之间的数据类型映射”中定义。
- Java 类可以包含 private、 protected 或 public 类、字段和方法。

  public 类、字段和方法可以在 SQL 中进行引用。 private 或 protected 类、字段和方法不能在 SQL 中进行引用，但可以在 Java 中进行引用并遵循一般 Java 规则。
- Java 类、字段和方法均具有各种不同的语法属性：
  - 类 — 字段数及其名称
  - 字段 — 其数据类型
  - 方法 — 参数的数目及其数据类型以及结果的数据类型

  SQL 系统使用 Java Reflection API 确定 Java-SQL 类本身中的这些语法属性。

Java-SQL 名称

Java-SQL 类名称（标识符）的长度被限制在 255 个字节以内。Java-SQL 字段和方法的名称可为任意长度，但是如果要在 Transact-SQL 中使用它们，则必须在 255 个字节以内。在 Transact-SQL 语句中使用的所有 Java-SQL 名称都必须符合 Transact-SQL 标识符规则。

长度为 30 个字节或更长的类、字段和方法的名称必须用引号引起来。

名称的第一个字符必须是字母 （大写或小写）或下划线 (_) 符号。后续字符可包括字母、数字、美元 ($) 符号或下划线 (_) 符号。

无论 SQL 系统是否区分大小写， Java-SQL 名称始终区分大小写。

有关标识符的详细信息，请参见第 148 页的 “Java-SQL 标识符”。

使用 Java 类作为数据类型

在安装一组 Java 类之后，可在 SQL 中将它们作为数据类型引用。要用作列数据类型，必须将 Java-SQL 类定义为 public，并且必须实现 java.io.Serializable 或 java.io.Externalizable。

可以将 Java-SQL 类指定为：

- SQL 列的数据类型
- Transact-SQL 存储过程的 Transact-SQL 变量和参数的数据类型
- SQL 列的缺省值

创建表时，可将 Java-SQL 类指定为 SQL 列的数据类型：

```
create table emps (
       name varchar(30),
       home_addr Address,
       mailing Address2Line null )
```

name 列是普通的 SQL 字符串， home_addr 和 mailing_addr 列可以包含 Java 对象，而 Address 和 Address2Line 是安装在数据库中的 Java-SQL 类。

可以将 Java-SQL 类指定为 Transact-SQL 变量的数据类型：

```
declare @A Address
declare @A2 Address2Line
```

也可以为 Java-SQL 列指定缺省值，但要符合常规约束，即指定的缺省值必须是常量表达式。该表达式通常是使用 new 运算符和常量参数的构造方法调用，例如：

```
create table emps (
       name varchar(30),
       home_addr Address default new Address
           ('Not known', ''),
       mailing_addr Address2Line
)
```

创建和更改包含 Java-SQL 列的表

当创建或更改具有 Java-SQL 列的表时，可以将任意已安装的 Java 类指定为列数据类型。您也可以指定如何在列中存储信息。所选择的存储选项将影响 Adaptive Server 引用和更新这些列中的字段的速度。

行的列值通常存储 “在行内”，也就是说，在分配给表的数据页上连续存储。不过，也可以使用与 text 和 image 数据项相同的存储方式，将 Java-SQL 列存储在单独的 “行外”位置。 Java-SQL 列的缺省值存储在行外。

如果 Java-SQL 列存储在行内：

- 处理存储在行内的对象比处理存储在行外的对象速度快。
- 根据数据库服务器的页大小和其它变量，存储在行内的对象最多可占据大约 16K 字节的空间。这包括它的整个序列，而不仅仅是其字段中的值。运行时表示形式超过 16K 字节限制的 Java 对象将会产生异常并中止命令。

如果 Java-SQL 列存储在行外，则该列必须符合适用于 text 和 image 列的限制：

- 存储在行外的对象的处理速度比存储在行内的对象慢。
- 存储在行外的对象可以是任意大小，但必须符合 text 和 image 列的一般限制。
- 不能在检查约束中引用行外列。

  同样，在检查约束中不要引用包含行外列的表。创建或更改表时，Adaptive Server 允许包括检查约束，但会在编译期发出警告信息，而在运行时忽略约束。
- 不能在使用 select distinct 的选择查询的列列表中包含行外列。
- 不能在比较运算符、判定或 group by 子句中指定行外列。

具有 in row/off row 选项的 create table 命令的部分语法为：

```
create table...column_name datatype
    [default {constant_expression | user | null}]
  {[{identity | null | not null}]
    [off row | [ in row [ ( size_in_bytes ) ] ] ]...
```

size_in_bytes 指定行内列的最大大小。最大值可达 16K 字节。缺省值为 255 个字节。

在 create table 中输入的最大行内列大小必须包括列的整个序列，而不仅仅是其字段中的值，还应包括最小开销值。

要确定包括开销和序列值在内的相应列大小，请使用 datalength 系统函数。 datalength 可用于确定要在列中存储的表示对象的实际大小。

例如：

```
select datalength (new class_name(...))
```

其中， *class_name* 是安装的 Java-SQL 类。

alter table 命令的部分语法为：

```
alter table...{add column_name datatype
    [default {constant_expression | user | null}]
    {identity | null} [ off row | [ in row ] ]...
```

注释　在此版本的 Adaptive Server 中，不能使用 alter column 更改行内列的列大小。

更改已分区的表

如果已将包含 Java 列的表分区，则在未删除分区前将无法更改表。若要更改表模式：

1　移除分区。

2　使用 alter table 命令。

3　对表进行重新分区。

选择、插入、更新和删除 Java 对象

在指定 Java-SQL 列之后，赋值给这些数据项的值必须是 Java 实例。这些实例最初是通过使用 new 运算符调用 Java 构造方法来生成的。您可以为列和变量生成 Java 实例。

构造方法是伪实例方法，它们可创建实例。构造方法与类具有相同的名称，但没有声明的数据类型。如果在类定义中未包括构造方法，缺省方法由 Java 基于类对象提供。您可以为每个类提供参数数量和类型各不相同的多个构造方法。当调用构造方法时，将使用具有正确的参数数量和类型的构造方法。

在以下示例中，为列和变量均生成了 Java 实例：

```
declare @A Address, @AA Address, @A2 Address2Line,
       @AA2 Address2Line

 select @A = new Address( )
 select @AA = new Address('123 Main Street', '99123')
 select @A2 = new Address2Line( )
 select @AA2 = new Address2Line('987 Front Street',
       'Unit 2', '99543')

 insert into emps values('John Doe', new Address( ),
       new Address2Line( ))
 insert into emps values('Bob Smith',
       new Address('432 ElmStreet', ë99654í),
       new Address2Line('PO Box 99', 'attn: Bob Smith', '99678') )
```

赋给 Java-SQL 列和变量的值可以接着赋给其它 Java-SQL 列和变量。例如：

```
declare @A Address, @AA Address, @A2 Address2Line,
       @AA2 Address2Line

 select @A = home_addr, @A2 = mailing_addr from emps
       where name = 'John Doe'
 insert into emps values ('George Baker', @A, @A2)

 select @AA2 = @A2
 update emps
       set home_addr = new Address('456 Shoreline Drive', '99321'),
         mailing_addr =  @AA2
         where name = 'Bob Smith'
```

您也可以从一个表向另一个表复制 Java-SQL 列的值。例如：

```
create table trainees (
       name char(30),
       home_addr Address,
       mailing_addr Address2Line null
 )
insert into trainees
select * from emps
       where name in ('Don Green', 'Bob Smith',
       'George Baker')
```

您可以使用标准 SQL 限定来引用和更新 Java-SQL 列的字段和 Java-SQL 变量。为避免与 SQL 使用点来限定名称的方法混淆，当在 SQL 中引用 Java 字段和方法名称时，可用双尖括号 (>>) 来限定它们。

```
declare @name varchar(100), @street varchar(100),
       @streetLine2 varchar(100), @zip char(10), @A Address

 select @A = new Address()
 select @A>>street = '789 Oak Lane'
 select @street = @A>>street

 select @street = home_add>>street, @zip = home_add>>zip from emps
       where name = 'Bob Smith'
 select @name = name from emps
       where home_addr>>street= '456 Shoreline Drive'

 update emps
       set home_addr>>street = '457 Shoreline Drive',
          home_addr>>zip = '99323'
       where home_addr>>street = '456 Shoreline Drive'
```

在 SQL 中调用 Java 方法

通过对实例方法使用实例名称限定，对静态方法使用实例或类名称限定，可以在 SQL 中通过引用 Java 方法来调用它们。

实例方法通常会与其类的某个实例中封装的数据有密切的联系。静态（类）方法影响整个类，而不仅仅是该类的一个特定实例。静态方法经常应用于各种类的对象和值。

静态方法一经安装，即可使用。使用静态方法作为函数的类必须为 public，但它不必是可序列化的。

在 Adaptive Server 中使用 Java 的一个主要优点是，可以将静态方法作为用户定义的函数 (UDF) 向调用者返回一个值。

可以在存储过程、触发器、 where 子句或任何可使用内置 SQL 函数的地方将 Java 静态方法作为 UDF。

在 SQL 中作为 UDF 直接调用的 Java 方法具有以下限制：

- 如果 Java 方法通过 JDBC 访问数据库，则只有 Java 方法可以使用结果集中的值，客户端应用程序无法使用这些值。
- 不支持输出参数。方法可以处理通过 JDBC 连接接收的数据，但只能向其调用者返回一个返回值，该值是在其定义中声明的。
- 只有在将类实例作为列值使用时，才能支持跨数据库的静态方法调用。

执行任何 UDF 的权限被隐式授予 public。如果 UDF 通过 JDBC 执行 SQL 查询，系统将检查 UDF 调用者的数据访问权限。因此，如果用户 A 调用 UDF 以访问表 t1，则用户 A 必须具有 t1 的 select 权限，否则查询将会失败。有关 Java 方法调用的安全模式的详细论述，请参见第 87 页的“安全性和权限”。

要使用 Java 静态方法返回结果集和输出参数，必须将这些方法装入 SQL 包装中，然后将其作为 SQLJ 存储过程或函数进行调用。有关在 Adaptive Server 中调用 Java 方法的各种方式的比较，请参见第 88 页的 “在 Adaptive Server 中调用 Java 方法”。

方法示例

示例类 Address 和 Address2Line 有名为 toString( ) 的实例方法，而示例类 Misc 有名为 stripLeadingBlanks( )、getNumber( )和 getStreet( ) 的静态方法。您可以在值表达式中以调用函数的方式来调用值方法。

```
declare @name varchar(100)
declare @street varchar(100)
declare @streetnum int
declare @A2 Address2Line

select @name = Misc.stripLeadingBlanks(name),
          @street = Misc.stripLeadingBlanks(home_addr>>street),
          @streetnum = Misc.getNumber(home_addr>>street),
          @A2 = mailing_addr
       from emps
       where home_addr>>toString( ) like '%Shoreline%'
```

有关无类型方法 （没有返回值的方法）的信息，请参见第 50 页的 “类型和无类型方法”。

Java-SQL 方法中的异常

如果 Java-SQL 方法调用在完成时有未处理的异常，将引发 SQL 异常，并显示以下错误消息：

```
Unhandled Java method exception
```

异常消息文本包含引发异常的 Java 类的名称，并在其后显示发生 Java 异常时提供的字符串 （如果存在）。

表示 Java 实例

诸如 isql 等非 Java 客户端无法从服务器接收序列化的 Java 对象。必须等 Adaptive Server 将该对象转换为可查看的表示后，才能查看及使用该对象。

若要使用实际字符串值， Adaptive Server 必须调用可将对象转换为 char 或 varchar 值的方法。 Address 类中的 toString() 方法就是这种方法的示例。必须创建符合自身需要的 toString() 方法，以便可使用该对象的可查看表示。

注释　Java API 中的 toString() 方法不会将对象转变为可查看的表示形式。所创建的 toString() 方法将覆盖 Java API 中的 toString() 方法。

使用 toString( )方法时， Adaptive Server 将对返回的字节数施加限制。Adaptive Server 将把对象的可打印表示截断为全局变量 @@stringsize 的值。 @@stringsize 缺省值是 50；可以使用 set stringsize 命令更改该值。例如：

```
set stringsize 300
```

计算机上的显示软件可以进一步截断数据项，使它们在屏幕上显示时不需要换行。

如果在每个类中包含 toString( ) 或类似方法，则可以使用以下两种方式之一返回对象的 toString( ) 方法的值：

- 可以在 Java-SQL 列中选择一个特定字段，这会自动调用 toString( )：

  ```
  select home__addr>>street from emps
  ```

- 可以选择列和 toString( ) 方法，这会在一个字符串中列出该列中的所有字段值：

  ```
  select home_addr>>toString() from emps
  ```

Java-SQL 数据项的赋值属性

赋值给 Java-SQL 数据项的值最终是通过 Java VM 中的 Java-SQL 方法所构造的值派生的。但是， Java-SQL 变量、参数和结果的逻辑表示与 Java-SQL 列的逻辑表示不同。

- Java-SQL *列* 是持久的 Java 序列化流，它们存储在表的包含行中。它们是包含 Java 实例表示的存储值。
- Java-SQL *变量*、*参数* 和 *函数结果* 是瞬时的。实际上，它们不包含 Java-SQL 实例，而是包含对 Java VM 中所含的 Java 实例的引用。

这些表示上的不同导致了赋值属性的不同，如下例所示。

- 具有 new 运算符的 Address 构造方法在 Java VM 中进行求值。它构造一个 Address 实例并返回对该实例的引用。该引用将赋值为 Java-SQL 变量 *@A* 的值：

```
declare @A Address, @AA Address, @A2 Address2Line,
       @AA2 Address2Line
select @A = new Address('432 Post Lane', '99444')
```

- *@A* 变量包含对 Java VM 中的 Java 实例的引用。该引用将复制到 *@AA* 变量中。 *@A* 和 *@AA* 变量此时引用相同的实例。

```
select @AA=@A
```

- 该赋值将修改 *@A* 所引用的 Address 的 zip 字段，它就是 *@AA* 引用的相同 Address 实例。因此， *@A.zip* 和 *@AA.zip* 的值此时都是“99222”。

```
select @A>>zip='99222'
```

- 具有 new 运算符的 Address 构造方法构造一个 Address 实例并返回对该实例的引用。不过，由于目标是 Java-SQL 列，因此， SQL 系统将序列化该引用所表示的 Address 实例，并将序列化的值复制到 emps 表的新行中。

```
insert into emps
       values ('Don Green', new Address('234 Stone
       Road', '99777'), new Address2Line( ) )
```

  Address2Line 构造方法的运行方式与 Address 方法相同，但它返回的是缺省实例，而不是具有指定参数值的实例。它执行的操作与 Address 实例相同。SQL 系统将序列化缺省 Address2Line 实例，并将序列化的值存储到 emps 表的新行中。

- insert 语句没有为 mailing_addr 列指定任何值，因此，该列将设置为 null，这与任何其它未在 insert 中指定值的列的方式相同。该空值是完全在 SQL 中生成的，并且 mailing_addr 列初始化根本不涉及 Java VM。

```
insert into emps (name, home_addr)  values ('Frank Lee', @A)
```

insert 语句指定将从 Java-SQL 变量 *@A* 中获取 home_addr 列的值。该变量包含对 Java VM 中的 Address 实例的引用。由于目标是 Java-SQL 列，因此，SQL 系统将序列化 *@A* 所表示的 Address 实例，并将序列化的值复制到 emps 表的新行中。

- 以下语句为 “Bob Brown” 插入一个新的 emps 行。将从 SQL 变量 *@A* 中获取 home_addr 列的值。它也是 *@A* 所引用的 Java 实例的序列。

```
insert into emps (name, home_addr)  values ('Bob Brown', @A)
```

- 此 update 语句将 “Frank Lee” 行中的 home_addr 列的 zip 字段设置为 “99777”。这对 “Bob Brown” 行中的 zip 字段没有影响，它仍然是 “99444”。

```
update emps
       set home_add>>zip = '99777'
       where name = 'Frank Lee'
```

- Java-SQL 列 home_addr 包含 Address 实例值的序列化表示形式。 SQL 系统调用 Java VM，将该表示非序列化为 Java VM 中的 Java 实例，并返回对非序列化的新副本的引用。该引用被赋值给 *@AA*。 *@AA* 引用的非序列化 Address 实例与列值和 *@A* 所引用的实例完全无关。

```
select @AA = home_addr from emps where name = 'Frank Lee'
```

- 该赋值将修改 *@A* 引用的 Address 实例的 zip 字段。此实例是 “Frank Lee” 行中的 home_addr 列的副本，但与该列值无关。因此，该赋值不会修改 “Frank Lee” 行中的 home_addr 列的 zip 字段。

```
select @A>>zip = '95678'
```

Java 与 SQL 字段之间的数据类型映射

当您在 Java VM 与 Adaptive Server 之间传送数据（无论沿哪个方向）时，必须注意数据项的数据类型在每个系统中都是不同的。 Adaptive Server 根据第 146 页的“Java 和 SQL 之间的数据类型映射”中的对应关系表，自动在 SQL 项与 Java 项之间进行映射。

因此，SQL 类型 char 将转换为 Java 类型 String，SQL 类型 binary 将转换为 Java 类型 byte[ ]，等等。

- 对于从 SQL 到 Java 的数据类型对应，任意长度的 char、 varchar 和 varbinary 类型都相应地对应为 Java String 或 byte[ ] 数据类型。
- 对于从 Java 到 SQL 的数据类型对应：
  - Java 数据类型 String 和 byte[ ] 与 SQL 数据类型 varchar 和 varbinary 对应，其中 16K 字节的最大长度值是由 Adaptive Server 定义的。
  - Java BigDecimal 数据类型对应于 SQL numeric(precision,scale)，其中的 precision 和 scale 由用户定义。

在 emps 表中， Address 和 Address2Line 类及 street、zip 和 line2 字段的最大值为 255 字节 （缺省值）。这些类的 Java 数据类型为 java.String，而在 SQL 中它们被视为 varchar(255)。

只有在 SQL 环境中使用数据类型为 Java 对象的表达式时，才会将该表达式转换为相应的 SQL 数据类型。例如，如果职员 “Smith”的 home_addr>>street 字段包含 260 个字符，并以 “6789 Main Street ...”开头：

```
select Misc.getStreet(home_addr>>street) from emps where name='Smith'
```

select 列表中的表达式将 260 个字符的 home_addr>>street 值传递给 getStreet( ) 方法 （不将其截断为 255 个字符）。然后， getStreet( ) 方法返回一个包含 255 个字符并以“Main Street...”开头的字符串。这个包含 255 个字符的字符串现在是 SQL select 列表的元素，因此，它将转换为 SQL 数据类型，并截断为 255 个字符 （如果需要）。

数据和标识符的字符集

Java 源代码和 Java String 数据的字符集都是 Unicode。 Java-SQL 类的字段可包含 Unicode 数据。

注释　在可见类的完全限定名或在可见成员的名称中使用的 Java 标识符只能使用拉丁字符和阿拉伯数字。

Java-SQL 数据中的子类型

类的子类型使您可使用 Java 的子类别替代和方法替代等特性。从类到它的一个父类的转换是一种扩大转换；从类到它的一个子类的转换是一种缩小转换。

- 扩大转换在一般的赋值和比较运算中隐式执行。这种转换总是能够成功，原因是每个子类的实例也是其父类的实例。
- 缩小转换必须使用显式的 convert 表达式来指定。仅当父类实例是子类的实例或是子类的子类时，缩小转换才会成功。否则，将发生一个异常。

扩大转换

您不需要用 convert 函数来指定扩大转换。例如，因为 Address2Line 类是 Address 类的子类，所以可将 Address2Line 值赋值给 Address 数据项。在 emps 表中， home_addr 列的数据类型为 Address，而 mailing_addr 列的数据类型为 Address2Line：

```
update emps
    set home_addr = mailing_addr
    where home_addr is null
```

对于实现 where 子句的行， home_addr 列包含 Address2Line，即使 home_addr 的声明类型为 Address。

这种赋值隐式地将类的实例视为该类的父类的实例。子类的运行时实例可保持其子类的数据类型和关联的数据。

缩小转换

必须使用 convert 函数将类的实例转换为该类的子类的实例。例如：

```
update emps
       set mailing_addr = convert(Address2Line, home_addr)
       where mailing_addr is null
```

如果将 update 语句中的缩小转换应用于任何包含非 Address2Line 的 Address 实例的 home_addr 列，则会导致异常。可通过在 where 子句中包含一个条件来避免产生这种异常：

```
update emps
       set mailing_addr = convert(Address2Line, home_addr)
       where mailing_addr is null
       and home_addr>>getClass( )>>toString( ) = 'Address2Line'
```

“home_addr>>getClass( )>>toString( )”表达式调用 Java Object 类的 getClass( ) 和 toString( ) 方法。Object 类隐式为所有类的父类，因此，为其定义的方法适用于所有类。

也可以使用 case 表达式：

```
update emps
       set mailing_addr =
          case
              when home_addr>>getClass( )>>toString( )
                ='Address2Line'
              then convert(Address2Line, home_addr)
              else null
          end
       where mailing_addr is null
```

运行时与编译期数据类型

无论扩大转换或缩小转换，都不会修改实际的实例值或其运行时数据类型；它们只是简单地指定用于编译期类型的类。这样，在将 mailing_addr 列中的 Address2Line 值存储到 home_address 列时，这些值仍具有 Address2Line 的运行时类型。

例如， Address 类和 Address2Line 子类均具有 toString( ) 方法，该方法返回 String 形式的完整地址数据。

```
select name, home_addr>>toString( )  from emps
       where home_addr>>toString( ) not like '%Line2=[ ]'
```

对于 emps 中的每一行， home_addr 列的声明类型均为 Address，但 home_addr 值的运行时类型是 Address 或 Address2Line，具体取决于上一条 update 语句的结果。对于 home_addr 列的运行时值是 Address 的行，将调用 Address 类的 toString( ) 方法；而对于 home_addr 列的运行时值是 Address2Line 的行，将调用 Address2Line 子类的 toString( ) 方法。

有关用于扩大和缩小转换的空值的说明，请参见第 48 页的 “使用 SQL convert 函数时的空值”。

对 Java-SQL 数据中的空值的处理

本节讨论如何在 Java-SQL 数据项中使用空值。

对空值实例字段与方法的引用

如果在字段引用中指定的实例值是空值，那么字段引用也是空值。同样，如果在实例方法调用中指定的实例值是空值，那么调用的结果也是空值。

对引用空值实例的字段或方法的作用， Java 有不同的规则。在 Java 中，如果尝试引用空值实例的字段，将会引发异常。

例如，假设 emps 表包含以下几行：

```
insert into emps (name, home_addr)
    values ("Al Adams",
    new Address("123 Main", "95321"))

insert into emps (name, home_addr)
    values ("Bob Baker",
    new Address("456 Side", "95123"))

 insert into emps (name, home_addr)
    values ("Carl Carter", null)
```

请考虑使用以下 select 语句：

```
select name, home_addr>>zip  from emps
where home_addr>>zip in ('95123', '95125', '95128')
```

如果对 “home_addr>>zip”的引用使用 Java 规则，则这些引用将导致“home_addr”列为空值的 “Carl Carter”行产生异常。要避免产生这种异常，您需要编写如下所示的 select 语句：

```
select name,
    case when home_addr is not null then home_addr>>zip
    else null end

from emps
    where case when home_addr is not null
    then home_addr>>zip
else
    null end
in ('95123', '95125', '95128')
```

SQL 约定也因此可用于对空值实例的字段和方法的引用：如果实例是空值，则任何字段或方法引用都为空值。此 SQL 规则的作用是使上面的 case 语句变为隐式的。

不过，这种用于具有空实例的字段引用的 SQL 规则仅适用于源（右侧）上下文中的字段引用，而不适用于作为赋值或 set 子句目标 （左侧）的字段引用。例如：

```
update emps
    set home_addr>>zip D '99123'
    where name D 'Charles Green'
```

显然，该 where 子句适用于 “Charles Green”行，因此， update 语句将尝试执行 set 子句。因为空值实例不具有可赋值的字段，所以不能赋值给空值实例的字段，这种情况下会引发一个异常。对空值实例的字段的引用是有效的，并会返回右侧上下文中的空值，从而在左侧上下文中引起异常。

同样的考虑和规则也适用于空值实例的方法调用。例如，如果修改上一示例而调用 home_addr 列的 toString( ) 方法：

```
select name, home_addr>>toString( )from emps
    where home_addr>>toString( ) D
    'StreetD234 Stone Road ZIPD 99777'
```

如果在实例方法调用中指定的实例的值是空值，那么该调用的结果也将是空值。因此， select 语句在此处是有效的，尽管它会在 Java 中引发一个异常。

空值作为 Java-SQL 方法的参数

将空值作为参数传递的结果与参数为空值的方法的操作无关，但依赖于返回数据类型传递空值的能力。

您不能将空值当作参数传递给 Java 标量类型方法；Java 标量类型始终不能是空值。但是， Java 对象类型可以接受空值。

对于以下 Java-SQL 类：

```
public class General implements java.io.Serializable {
       public static int identity1(int I) {return I;}
       public static java.lang.Integer identity2
         (java.lang.Integer I) {return I;}
       public static Address identity3 (Address A) {return A;}
   }
```

注意这些调用：

```
declare @I int
declare @A Address;

select @I = General.identity1(@I)
select @I = General.identity2(new java.lang.Integer(@I))
select @A = General.identity3(@A)
```

@I 和 *@A* 变量的值均为空值，因为还没有为它们赋值。

- identity1( ) 方法的调用会引发一个异常。identity1( ) 中的参数 *@I* 的数据类型是 Java int 类型，此类型是一个标量，没有空状态。尝试将空值传递给 identity1( ) 会引发一个异常。
- identity2( ) 方法的调用会成功。identity2( ) 参数的数据类型是 Java 类 java.lang.Integer，而 new 表达式创建了 java.lang.Integer 的实例，该实例被设置为变量 *@I* 的值。
- 可以成功完成对 identity3( ) 方法的调用。

对 identity1( ) 的成功调用将不会返回空值结果，因为返回类型没有空值状态。空值不能直接传递，因为没有参数类型信息，方法解析将失败。

identity2( ) 和 identity3( ) 的成功调用可以返回空结果。

使用 SQL *convert* 函数时的空值

可以使用 convert 函数将某个类的 Java 对象转换为该类的父类或子类的 Java 对象。

正如第 43 页的“Java-SQL 数据中的子类型”所述，emps 表的 home_addr 列可以包含 Address 和 Address2Line 类的值。在下面的示例中：

```
select name, home_addr>>street, convert(Address2Line, home_addr)>>line2,
       home_addr>>zip from emps
```

“convert(Address2Line, home_addr)”表达式包含数据类型 (Address2Line) 和表达式 (home_addr)。在编译期，表达式 (home_addr) 必须是类 (Address2Line) 的子类型或父类型。在运行时，此 convert 调用的操作取决于表达式值的运行时类型是类、子类还是父类：

- 如果表达式 (home_addr) 的运行时值是指定的类 (Address2Line) 或它的一个子类，则会返回该表达式的值，并且表达式值具有指定的数据类型 (Address2Line)。
- 如果表达式 (home_addr) 的运行时值是指定类 (Address) 的父类，则会返回空值。

Adaptive Server 为结果中的每一行计算 select 语句的值。对于每一行：

- 如果 home_addr 列的值是 Address2Line，convert 将返回该值，并且字段引用将提取 line2 字段。如果 convert 返回空值，则字段引用本身也为空值。
- 如果 convert 返回空值，则字段引用本身的求值结果为空值。

因此， select 的结果将为其 home_addr 列是 Address2Line 的行显示 line2 值，而为其 home_addr 列是 Address 的行显示空值。正如第 45 页的“对 Java-SQL 数据中的空值的处理”所述，对于 home_addr 列为空值的那些行， select 也会显示空 line2 值。

Java-SQL string 数据

在 Java-SQL 列中，类型为 String 的字段将存储为 Unicode。

如果将 Java-SQL String 字段赋值给类型为 char、varchar、nchar、nvarchar 或 text 的 SQL 数据项，Unicode 数据将转换为 SQL 系统的字符集。转换错误由 set char_convert 选项指定。

当类型为 char、 varchar、 nchar 或 text 的 SQL 数据项被赋值给存储为 Unicode 的 Java-SQL String 字段时，字符数据将转换为 Unicode。在这样的数据中，未定义的代码点会导致转换错误。

零长度字符串

在 Transact-SQL 中，零长度字符串被视为空值，空字符串 ( ) 被视为一个空格。

为了与 Transact-SQL 保持一致，如果将长度为零的 Java-SQL String 值赋值给类型为 char、varchar、nchar、nvarchar 或 text 的 SQL 数据项，Java-SQL String 值将替换为一个空格。

例如：

```
1> declare @s varchar(20)
2> select @s = new  java.lang.String()
3> select @s, char_length(@s)
4> go
```

(1 row affected)

```
----------------  ----------------
                               1
```

否则，在 SQL 中，零长度值将被视为 SQL 空值；如果将其赋值给 Java-SQL String，则 Java-SQL String 为 Java 空值。

类型和无类型方法

Java 方法 （实例和静态）是类型方法或无类型方法。通常，类型方法返回带有结果类型的值；无类型方法执行某些操作，但不返回任何值。

例如，在 Address 类中：

- toString( ) 方法是类型为 String 的*类型方法*。
- removeLeadingBlanks( ) 方法是一个*无类型方法*。
- Address 构造方法是类型为 Address 类的*类型方法*。

在调用构造方法时，应将类型方法作为函数调用并使用 new 关键字：

```
insert into emps
       values ('Don Green', new Address('234 Stone Road', '99777'),
          new Address2Line( ) )

select name, home_addr>>toString( ) from emps
       where home_addr>>toString( )  like '%Baker%'
```

Address 类的 removeLeadingBlanks( ) 方法是无类型实例方法，它修改给定实例的 street 和 zip 字段。可以为 emps 表中的每一行的 home_addr 列调用 removeLeadingBlanks( )。例如：

```
update emps
    set home_addr =
       home_addr>>removeLeadingBlanks( )
```

removeLeadingBlanks( ) 从 home_addr 列的 street 和 zip 字段中删除前导空格。 Transact-SQL update 语句没有为此类操作提供框架或语法，它只替换列值。

Java 无类型实例方法

要在 SQL 系统中使用 Java 无类型实例方法的 “原位更新”操作，Adaptive Server 中的 Java 将按如下方式处理 Java 无类型实例方法调用：

类 C 的实例 CI 的无类型实例方法 M( ) 应写为 “CI.M(...)”：

- 在 SQL 中，此调用被视为类型方法调用。结果类型隐式为类 C，结果值是对 CI 的引用。在无类型实例方法调用的操作之后，该引用标识实例 CI 的副本。
- 在 Java 中，此调用是一个无类型方法调用，它执行动作并且不返回任何值。

例如，可以为 emps 表中所选行的 home_addr 列调用 removeLeadingBlanks( ) 方法，如下所示：

```
update emps
       set home_addr = home_addr>>removeLeadingBlanks( )
       where home_addr>>removeLeadingBlanks( )>>street like "123%"
```

1　在 where 子句中，“home_addr>>removeLeadingBlanks( )”为 emps 表中的某一行的 home_addr 列调用 removeLeadingBlanks( ) 方法。removeLeadingBlanks( ) 从该列的副本中删除 street 和 zip 字段的前导空格。然后， SQL 系统返回对 home_addr 列的已修改副本的引用。后续的字段引用：

```
home_addr>>removeLeadingBlanks( )>>street
```

返回删除了前导空格的 street 字段。 where 子句中对 home_addr 的引用针对的是该列的一个副本。对 where 子句的求值并*不*修改 home_addr 列。

2　update 语句为 emps 中的每一行执行 set 子句 （当 where 子句为 true 时）。

3　在 set 子句的右侧，对 “home_addr>>removeLeadingBlanks( )”调用的执行方式与 where 子句的执行方式相同：removeLeadingBlank( ) 从该副本的 street 和 zip 字段中删除前导空白。然后， SQL 系统返回对 home_addr 列的已修改副本的引用。

4　由 set 子句右侧的结果表示的 Address 实例被序列化，并被复制到 set 子句左侧指定的列中：set 子句右侧表达式的结果是 home_addr 列的副本，该列已从 street 和 zip 字段中删除了前导空白。然后，将修改的副本作为 home_addr 列的新值重新赋值给该列。

set 子句左右两侧的表达式是独立的，这在 update 语句很正常。

以下 update 语句显示对 set 子句右侧的 mailing_addr 列的一个无类型实例方法的调用，该列被赋值给左侧的 home_address 列。

```
update emps
       set home_addr = mailing_addr>>removeLeadingBlanks( )
       where ...
```

在该 set 子句中， mailing_addr 列的无类型方法 removeLeadingBlanks( ) 生成对 mailing_addr 列中的 Address2Line 实例的已修改副本的引用。然后，由该引用表示的实例被序列化并被赋值给 home_addr 列。此操作会更新 home_addr 列，而不影响 mailing_addr 列。

Java 无类型静态方法

无法使用简单的 SQL execute 命令调用无类型静态方法，而必须将无类型静态方法的调用放在 select 语句中。

例如，假设 Java 类 C 具有无类型静态方法 M(...)，并且 M( ) 执行一个要在 SQL 中调用的操作。例如， M( ) 可以使用 JDBC 调用执行一系列没有返回值的 SQL 语句 （如 create 或 drop），这适用于无类型方法。

您必须在 select 命令中调用无类型静态方法，例如：

```
select C.M(...)
```

为允许使用 select 调用无类型静态方法， SQL 将无类型静态方法视为返回 int 数据类型的空值。

等同性和排序运算

在数据库中使用 Java 时，可使用等同性和排序运算符。您不能：

- 在排序运算中引用 Java-SQL 数据项。
- 如果 Java-SQL 数据项存储在行外列中，则可在等同性运算中引用它们。
- 使用 order by 子句，该子句要求您确定排序顺序。
- 使用 “>”、“<”、“<=”或 “>=”运算符进行直接比较。

行内列中允许使用以下等同性运算：

- 使用 distinct 关键字，这是按照行的等同性定义的（包括 Java-SQL 列）。
- 使用 “=”和 “!=”运算符进行直接比较。
- 使用 union 运算符 （不是 union all），该运算符删除了重复项并要求使用与 distinct 子句相同的比较类型。
- 使用 group by 子句将行划分为多个组，每组中作为分组依据的列具有相等的值。

求值顺序和 Java 方法调用

Adaptive Server 没有定义比较运算和其它运算中操作数的求值顺序。不过， Adaptive Server 将对每一个查询求值，并根据执行的最快速度来选择求值顺序。

本节说明在将列或变量和参数当作参数传递时，不同的求值顺序对结果的影响。本节中的示例使用以下 Java-SQL 类：

```
public class Utility implements java.io.Serializable {
          public static int F (Address A) {
             if (A.zip.length( ) > 5) return 0;
             else {A.zip = A.zip + "-1234"; return 1;}
          }
          public static int G (Address A) {
             if (A.zip.length( ) > 5) return 0;
             else {A.zip = A.zip + "-1234"; return 1;}
          }
}
```

列

通常情况下，应避免在同一 SQL 语句中调用同一 Java-SQL 对象上的多个方法。如果它们当中至少有一个修改了对象，则求值顺序将会影响到结果。

例如，在以下示例中：

```
select *  from emp E
       where Utility.F(E.home_addr) > Utility.F(E.home_addr)
```

where 子句在两个不同的方法调用中传递相同的 home_addr 列。请考虑针对某一行对 where 子句进行求值，该行的 home_addr 列包含 5 个字符的邮政编码，例如 “95123”。

Adaptive Server 可以先对比较运算的左侧或右侧进行求值。第一个求值结束后，再继续进行第二个求值。因为这种方法的执行速度更快，所以 Adaptive Server 可以使第二个调用看到第一个调用所作出的参数修改。

在本示例中， Adaptive Server 选择的第一个调用返回 1，第二个调用返回 0。如果先对左操作数求值，则比较结果是 1>0， where 子句为 true；如果先对右操作数求值，则比较结果是 0>1， where 子句为 false。

变量和参数

同样，在将变量或参数当作参数传递时，求值顺序也会影响结果。

请注意以下语句：

```
declare @A Address
declare @Order  varchar(20)

select @A = new Address('95444', '123 Port Avenue')
select case when Utility.F(@A)>Utility.G(@A)
          then ‘Left’ else ‘Right’ end
select @Order = case when utility.F(@A) > utility.G(@A)
          then 'Left'  else 'Right' end
```

新 Address 具有一个 5 个字符的邮政编码字段。对 case 表达式进行求值时，根据比较运算的左右操作数的求值顺序不同，比较结果是 1>0 或 0>1，并相应地将 *@Order* 变量设置为 “Left”或 “Right”。

就列参数而言，表达式值取决于求值顺序。根据比较运算的左右操作数的求值顺序不同， *@A* 引用的 Address 实例的 zip 字段的求值结果为 “95444-4321”或 “95444-1234”。

表达式中的确定性 Java 函数

如果使用一组相同的输入值对确定性表达式和函数进行求值，它们始终返回相同的结果。 Adaptive Server 中的所有 Java 函数都是确定性的。因此，如果涉及 Java 函数的表达式中的参数和输入值不变，则 Adaptive Server 将整个表达式视为确定性的。

当 Adaptive Server 遇到表达式中的 Java 函数时， Adaptive Server 将立即计算该表达式，以便只执行一次计算，而不是对每一行重复进行计算。这会提高性能，但可能会出现无法预料的结果。

请考虑以下示例：

```
1> create table CaseTest
2> (TestValue varchar(50))
3> go
1> insert into CaseTest values(í07í)
2> go
(1 row affected)

1> declare @IntArray sybase.cpp.value.client/common.IntArray
2> select @IntArray = new sybase.cpp.value.client.common.IntArray()
3> SELECT CASE
4> WHEN CT.TestValue = ’07’
```

```
5> THEN @IntArray >> setInt(new java.lang.Integer(10))
6> ELSE @IntArray >> setInt(new java.lang.Integer(11))
7> END
8> FROM CaseTest CT
9> select @IntArray >> getInt(0) as GetObjAfter0
10> select @IntArray >> getInt(1) as GetObjAfter1
11> select @IntArray >> getArraySize() as NumObjectsOnArray
12> go

-------------------------------------------
sybase.cpp.value.client.common.IntArray@22cc0f30

(1 row affected)
GetObjAfter0
-------------------------------------------
11

(1 row affected)
NumObjectsOnArray
---------------
2

(1 row affected)
```

您可能希望 case 语句的一个分支的计算结果为 true，因而仅在整数数组中插入一个值 (10)，但由于 `setInt(new java.lang.Integer(10))` 和 `setInt(new java.lang.Integer(11))` 表达式是确定性的，因此，Adaptive Server 将 “预先计算”结果并在数组中填充两个值。

可通过添加对列的引用，使 Adaptive Server 不知道每次执行表达式时是否都生成相同的结果，从而将表达式变为非确定性表达式。例如，对示例中的 Transact-SQL 语句进行以下更改：

```
1> declare @IntArray Sybase.cpp.value.client.common.IntArray
2> select @IntArray = new sybase.cpp.value.client.common.IntArray()
3> SELECT CASE
4> WHEN CT.TestValue = '07'
5> THEN @IntArray >> setInt(new java.lang.Integer(10 +
convert(int,CT.TestValue) - convert(int,CT.TestValue)))
6> ELSE @IntArray >> setInt(new java.lang.Integer(11 +
convert(int,CT.TestValue) - convert(int,CT.TestValue)))
7> END
8> FROM CaseTest CT
9> select @IntArray >> getInt(0) as GetObjAfter0
10> select @IntArray >> getInt(1) as GetObjAfter1
11> select @IntArray >> getArraySize() as NumObjectsOnArray
12> go
```

通过在 case 语句的 THEN 和 ELSE 部分中包含列引用，优化程序不再将语句视为常量，并且不会预先计算 Java insert 语句。

Java-SQL 类中的静态变量

声明为静态的 Java 变量与 Java 类关联，而不是与类的每个实例关联。变量只为整个类分配一次。

例如，可以在 Address 类中包含一个静态变量以指定建议的 Street 字段长度限制：

```
public class Address implements java.io.Serializable {

    public static int recommendedLimit;
    public String street;
    public String zip;
// ...
}
```

可将静态变量指定为 final，表明它不可更新：

```
    public static final int recommendedLimit;
```

否则，就可更新该变量。

可通过用类的实例限定静态变量的方式引用 SQL 中的 Java 类的静态变量。例如：

```
declare @a Address
select @a>>recommendedLimit
```

如果没有类实例，则可以使用以下方法：

```
select convert(Address, null)>>recommendedLimit
```

“(convert(null, Address))”表达式将一个空值转换为 Address 类型；即，它生成一个空 Address 实例，可随后使用静态变量名称限定该实例。无法通过使用类名限定静态变量，在 SQL 中引用 Java 类的静态变量。例如，以下两个语句都是错误的：

```
select Address.recommendedLimit

select Address>>recommendedLimit
```

只能在当前会话中访问赋值给非最终静态变量的值。

Adaptive Server 15.0.3 和更高版本中的静态变量更改

在 Adaptive Server 15.0.2 和更低版本中，将为每个任务分配其自己的内部 JVM。每个 JVM 与一组唯一的类装载程序相关联。因此，类变量仅供一个 Adaptive Server 任务使用。

Adaptive Server 15.0.3 和更高版本引入了 PCA/JVM， Adaptive Server 将在相同 JVM 中使用单独的 JVM 线程处理每个 Adaptive Server 任务。所有用户类都是由仅与执行特定 Java 方法的特定 Adaptive Server 任务相关联的类装载程序装载的。由于没有在 Adaptive Server 任务之间共享与用户类关联的类装载程序，而不会将用户类视为相同的类。因此，用户类中的类变量在 Adaptive Server 任务之间不可见。

不过，系统类装载程序装载的类中的类变量在所有 Adaptive Server 任务之间可见，因为所有用户类装载程序均将系统类装载程序作为父装载程序。这适用于所有标准 JVM。在多个任务中使用这些类中的类变量时，并不会降低功能或安全性。

Cluster Edition 中的静态变量更改

在 Cluster Edition 中，Adaptive Server 处理用户类和系统类装载程序装载的类中的类变量，如第 57 页的“Adaptive Server 15.0.3 和更高版本中的静态变量更改”中所述：不过，每个节点运行单独的、不相关的 PCA/JVM 实例。如果在一个节点上设置类变量，并不会在集群中的所有其它节点上自动更改该值。由于 Adaptive Server 任务可以在多个节点上运行，如果用户类依赖于类变量，则必须在所有节点上显式地设置相同的类变量。

多个数据库中的 Java 类

可以在同一个 Adaptive Server 系统上的不同数据库中存储名称相同的 Java 类。本节说明了如何使用这些类。

范围

安装 Java 类或类集时，这些类将安装在当前数据库中。在转储或装载数据库时，即使 Adaptive Server 系统上的其它数据库中存在名称相同的类，也始终会包括当前在该数据库中安装的 Java-SQL 类。

您可以在不同的数据库中安装同名的 Java 类。这些同名的类可以是：

- 安装在不同数据库中的相同类。
- 相互兼容的不同类。这样，由任何一个类生成的序列化值可以被另一个类接受。
- “向上”兼容的不同类。即，其中的一个类生成的序列化值可以被另一个类接受，反之则不然。
- 相互不兼容的不同类。例如，一个名为 Sheet 的类用于纸张供应，而其它名为 Sheet 的类用于亚麻制品供应。

跨数据库引用

可从一个数据库引用另一个数据库的表的列中存储的对象。

例如，假定为以下配置：

- Address 类安装在 db1 和 db2 中。
- 在 db1（所有者 Smith）和 db2（所有者 Jones）中均创建了 emps 表。

在这些示例中，当前数据库是 db1。您可以跨数据库调用连接或方法。例如：

- 跨数据库的 join 可能如下所示：

```
declare @count  int
select @count(*)
    from db2.Jones.emps, db1.Smith.emps
    where db2.Jones.emps.home_addr>>zip =
        db1.Smith.emps.home_addr>>zip
```

- 跨数据库的方法调用可能如下所示：

```
select db2.Jones.emps.home_addr>>toString( )
       from db2.Jones.emps
       where db2.Jones.emps.name = 'John Stone'
```

在这些示例中，不传送实例值。db2 中包含的实例的字段和方法仅由 db1 中的例程引用。这样，对于跨数据库的连接和方法调用：

- db1 不需要包含 Address 类。
- 如果 db1 确实包含 Address 类，则它可以具有与 db2 中的 Address 类完全不同的属性。

类间传送

可以将一个数据库中类的实例赋值给另一个数据库中同名类的实例。在源数据库中由该类创建的实例被传送到声明类型为当前（目标）数据库中的相应类的列或变量中。

可以从一个数据库中的表向另一个数据库中的表插入或更新。例如：

```
insert into db1.Smith.emps select * from
       db2.Jones.emps

update db1.Smith.emps
       set home_addr = (select db2.Jones.emps.home_addr
          from db2.Jones.emps
       where db2.Jones.emps.name =
          db1.Smith.emps.name)
```

可由一个数据库中的变量向另一个数据库插入或更新。（以下代码片段包含在 db2 的存储过程中）。例如：

```
declare @home_addr Address
select @home_addr = new Address('94608', '222 Baker
       Street')
insert into db1.Janes.emps(name, home_addr)
       values ('Jone Stone', @home_addr)
```

在这些示例中，实例值在数据库之间传送。您可以：

- 在两个本地的数据库之间传送实例。
- 在本地数据库和远程数据库之间传送实例。
- 在 SQL 客户端与 Adaptive Server 之间传送实例。
- 用 install 和 update 语句，或者 remove 和 update 语句来替换类。

在类间传送中，Java 序列从源类向目标类传送。如果源数据库中的类与目标数据库中的类不兼容，则引发 Java 异常 InvalidClassException。

传递类间参数

可以在不同数据库中的同名类之间传递参数。当传递类间参数时：

- Java-SQL 列与包含该列的数据库中指定的 Java 类的版本关联。
- Java-SQL 变量（在 Transact-SQL 中）与当前数据库中指定的 Java 类的版本关联。
- 类 C 的 Java-SQL 中间结果与返回结果的 Java 方法所在的同一数据库中的类 C 版本相关联。
- 当一个 Java 实例值 *JI* 被赋值给目标变量或列，或者传递给 Java 方法时， *JI* 从其关联的类转换为与接收的目标或方法关联的类。

临时数据库和工作数据库

对 Java 类或数据库的所有规则也适用于临时数据库和模型数据库：

- 临时表的 Java-SQL 列包含 Java 实例的字节字符串序列。
- Java-SQL 列与临时数据库中指定的类的版本关联。

您可以在临时数据库中安装 Java 类，但仅当临时数据库持久时，它们才会持久。

提供 Java 类在临时数据库中引用的最简单方式是在模型数据库中安装 Java 类。它们将出现在从该模型中派生的任何临时数据库中。

Java 类

本节显示了一些简单的 Java 类，本章使用这些类说明 Adaptive Server 中的 Java。

以下是 Address 类：

```
//
// Copyright (c) 2005
// Sybase, Inc
// Dublin, CA 94568
// All Rights Reserved
//
/**
* A simple class for address data, to illustrate using a Java class
```

```
* as a SQL datatype.
*/
public class  Address implements java.io.Serializable  {

/**
* The street data for the address.
* @serial A simple String value.
*/
       public String street;

/**
* The zipcode data for the address.
* @serial A simple String value.
*/
       String zip;

/** A default constructor.
*/
       public Address ( ) {
          street = "Unknown";
          zip = "None";
       }
/**
* A constructor with parameters
* @param S       a string with the street information
* @param Z       a string with the zipcode information
*/
       public Address (String S, String Z) {
          street = S;
          zip = Z;
       }
/**
* A method to return a display of the address data.
* @returns a string with a display version of the address data.
*/
       public String toString( ) {
          return  "Street= " + street +  "  ZIP= " + zip;
       }
/**
* A void method to remove leading blanks.
* This method uses the static method
* <code>Misc.stripLeadingBlanks</code>.
*/
       public void removeLeadingBlanks( ) {
       street = Misc.stripLeadingBlanks(street);
       zip = Misc.stripLeadingBlanks(street);
```

```
        }
}
```

以下是 Address2Line 类，它是 Address 类的一个子类：

```
//
// Copyright (c) 2005
// Sybase, Inc
// Dublin, CA 94568
// All Rights Reserved
//
/**
* A subclass of the Address class that adds a seond line of address data,
* <p>This is a simple subclass to illustrate using a Java subclass
* as a SQL datatype.
*/
public class Address2Line extends Address implements java.io.Serializable  {

/**
* The second line of street data for the address.
* @serial a simple String value
*/
        String line2;
/**
* A default constructor
*/
        public Address2Line ( ) {
            street = "Unknown";
            line2 = " ";
            zip = "None";
        }
/**
* A constructor with parameters.
* @param S a string with the street information
* @param L2 a string with the second line of address data
* @param Z a string with the zipcode information
*/
public Address2Line (String S, String L2, String Z) {
        street = S;
        line2 = L2;
        zip = Z;
}

/**
* A method to return a display of the address data
* @returns a string with a display version of the address data
*/
```

```
public String toString( ) {
       return "Street= " + street + " Line2= " + line2 +  " ZIP= " + zip;
}

/**
* A void method to remove leading blanks.
* This method uses the static method
* <code>Misc.stripLeadingBlanks</code>.
*/

       public void removeLeadingBlanks( ) {
          line2  = Misc.stripLeadingBlanks(line2);
          super.removeLeadingBlanks( );
       }
}
```

Misc 类包含多组杂类例程：

```
//
// Copyright (c) 2005
// Sybase, Inc
// Dublin, CA 94568
// All Rights Reserved
//
/**
* A non-instantiable class with miscellaneous static methods
* that illustrate the use of Java methods in SQL.
*/

public class Misc{

/**
* The Misc class contains only static methods and cannot be instantiated.
*/

private Misc( ) { }

/**
* Removes leading blanks from a String
*/
       public static String stripLeadingBlanks(String s) {
          if (s == null) return null;
          for (int scan=0; scan<s.length( ); scan++)
             if (!java.lang.Character.isWhitespace(s.charAt(scan) ))
                break;
             } else if (scan == s.length( )){
                return  "";
```

```
            } else return s.substring(scan);
            }
         }
      }
      return  "";
}
/**
* Extracts the street number from an address line.
* e.g., Misc.getNumber("   123 Main Street") == 123
*       Misc.getNumber("   Main Street")  == 0
*       Misc.getNumber("") == 0
*       Misc.getNumber("   123   ") ==  123
*       Misc.getNumber("   Main 123 ")  == 0
* @param s a string assumed to have address data
* @return a string with the extracted street number
*/

      public static int getNumber (String s) {
         String stripped = stripLeadingBlanks(s);
         if (s==null) return -1;
         for(int right=0; right < stripped.length( ); right++){
            if (!java.lang.Character.isDigit(stripped.charAt(right))) {
               break;
            } else if (right==0){
               return 0;
            } else {
               return java.lang.Integer.parseInt
                  (stripped.substring(0, right), 10);
            }
            }
            return -1;
         }

/**
* Extract the "street" from an address line.
* e.g.,  Misc.getStreet("   123 Main Street") ==  "Main Street"
*      Misc.getStreet("   Main Street")  ==   "Main Street"
*      Misc.getStreet("") ==  ""
*      Misc.getStreet("   123   ") ==  ""
*      Misc.getStreet("   Main 123 ")  == "Main 123"
* @param s a string assumed to have address data
* @return a string with the extracted street name
*/
      public static String getStreet(String s) {
         int left;
         if (s==null) return null;
```

```
            for (left=0; left<s.length( ); left++){
                if(java.lang.Character.isLetter(s.charAt(left))) {
                    break;
                } else if (left == s.length( )) {
                    return "";
                } else {
                    return s.substring(left);
                }
            }
            return "";
        }
}
```

第 5 章

使用 JDBC 访问数据

本章说明如何用 Java 数据库连接 (JDBC) 来访问数据。

概述

JDBC 为 Java 应用程序提供 SQL 接口。如果要从 Java 中访问关系型数据，必须使用 JDBC 调用。

可以通过以下两种方式之一使用带有 Adaptive Server SQL 接口的 JDBC：

- *客户端上的 JDBC* — Java 客户端应用程序可以使用 Sybase jConnect JDBC 驱动程序对 Adaptive Server 进行 JDBC 调用。
- *服务器上的 JDBC* — 数据库中安装的 Java 类可以使用 Adaptive Server 本机 JDBC 驱动程序对数据库进行 JDBC 调用。

这两种情况下，使用 JDBC 调用执行 SQL 操作在本质上是相同的。

本章提供了说明如何使用 JDBC 执行 SQL 操作的示例类和方法。这些类和方法并非要用来作为模板，而是作为一般的指南。

JDBC 概念和术语

JDBC 是一个 Java API，也是控制 Java 应用程序开发的基本函数的 Java 类库的一个标准部分。 JDBC 提供的 SQL 功能与 ODBC 和动态 SQL 所提供的相似。

下面的一系列事件是 JDBC 应用程序的典型用法：

1 创建 *Connection* 对象 — 调用 DriverManager 类的静态方法 getConnection( ) 以创建 *Connection* 对象。这会建立一个数据库连接。

2 生成 *Statement* 对象 — 使用 *Connection* 对象生成 *Statement* 对象。

3 将 SQL 语句传递给 *Statement* 对象 — 如果该语句是一个查询，此操作将返回一个 *ResultSet* 对象。

ResultSet 对象包含从 SQL 语句返回的数据，但每次提供一行数据（与游标的工作方式相似）。

4 循环访问结果集中的行 — 调用 *ResultSet* 对象的 next( ) 方法以执行下列操作：

- 将当前行 （结果集中通过 *ResultSet* 对象公开的行）向前移动一行。
- 返回一个 Boolean 值 (true/false) 以指示是否有可前进到的行。

5 对于每一行，检索 *ResultSet* 对象中的列的值 — 使用 getInt( )、getString( ) 或类似方法确定列名或位置。

客户端与服务器端 JDBC 之间的差异

客户端与数据库服务器上的 JDBC 之间的差异在于如何与数据库环境建立连接。

当使用客户端或服务器端 JDBC 时，可以调用 Drivermanager.getConnection() 方法来建立一个到服务器的连接。

- 对于客户端 JDBC，可以使用 Sybase jConnect JDBC 驱动程序，并且用服务器的标识调用 Drivermanager.getConnection() 方法。这样就建立了一个到指定服务器的连接。

- 对于服务器端 JDBC，可以使用 Adaptive Server 本机 JDBC 驱动程序，并使用下列值之一调用 Drivermanager.getConnection() 方法：
  - jdbc:default:connection
  - jdbc:sybase:ase
  - jdbc:default
  - 空字符串

  这将建立一个到当前服务器的连接。只有对 getConnection() 方法的第一次调用创建到当前服务器的新连接。后续的调用将返回该连接的包装，并且所有连接属性均未改变。

您可以通过使用条件语句设置 URL 来编写可在客户端和服务器上运行的 JDBC 类。

权限

- *Java 执行权限* — 与数据库中的所有 Java 类一样，任何用户都可以访问包含 JDBC 语句的类。授予权限以执行过程的 grant execute 语句在 Java 方法中没有等效的语句，不需要使用类所有者名称限定类名。
- *SQL 执行权限* — 在执行 Java 类时，将使用执行它们的连接所具有的权限。这种行为不同于存储过程，存储过程在执行时具有数据库所有者的权限。

使用 JDBC 访问数据

本节介绍如何使用 JDBC 来执行 SQL 应用程序的典型操作。这些示例是从 JDBCExamples 类中提取的，第 78 页的 “JDBCExamples 类”中介绍了该类。

JDBCExamples 说明了用户接口的基础知识，并介绍了 SQL 操作的内部编码方法。

JDBCExamples 类的概述

要在计算机上执行这些示例，请在服务器上安装 Address 类，并将其包括在 jConnect 客户端的 Java CLASSPATH 中。

可以从 jConnect 客户端或 Adaptive Server 调用 JDBCExamples 的方法。

注释 必须从 jConnect 客户端创建或删除存储过程。 Adaptive Server 内部驱动程序不支持 create procedure 和 drop procedure 语句。

JDBCExamples 静态方法执行以下 SQL 操作：

- 创建和删除示例表 xmp：

```
create table xmp (id int, name varchar(50), home Address)
```

- 创建和删除示例存储过程 inoutproc：

```
create procedure inoutproc @id int, @newname varchar(50),
       @newhome Address, @oldname varchar(50) output, @oldhome
       Address output as

select @oldname = name, @oldhome = home from xmp
       where id=@id
update xmp set name=@newname, home = @newhome
       where id=@id
```

- 在 xmp 表中插入一行。
- 在 xmp 表中选择一行。
- 更新 xmp 表中的一行。
- 调用存储过程 inoutproc，它具有数据类型为 java.lang.String 和 Address 的输入参数和输出参数。

JDBCExamples 仅在 xmp 表和 inoutproc 过程中运行。

main( ) 和 *serverMain( )* 方法

JDBCExamples 有两个主要方法：

- main( ) — 从 jConnect 客户端的命令行中调用。
- serverMain( ) — 执行与 main( ) 相同的操作，但在 Adaptive Server 中调用。

JDBCExamples 类的所有操作都是通过调用这些方法之一来调用，并使用一个参数来指示要执行的操作。

使用 *main( )*

可以从如下所示的 jConnect 命令行中调用 main( ) 方法：

```
java JDBCExamples
       "server-name:port-number?user=user-name&password=password" action
```

可以使用 dsedit 工具，从 interfaces 文件中确定 *server-name* 和 *port-number*。*user-name* 和 *password* 是您的用户名和口令。如果忽略 &password=*password*，则缺省为空口令。这里有两个示例：

```
"antibes:4000?user=smith&password=1x2x3"
"antibes:4000?user=sa"
```

确保参数在引号中。

action 参数可以是 create table、create procedure、insert、select、update 或 call。它是区分大小写的。

可以从 jConnect 命令行中调用 JDBCExamples 以创建 xmp 表和 inoutproc 存储过程，如下所示：

```
java JDBCExamples "antibes:4000?user=sa" CreateTable
java JDBCExamples "antibes:4000?user=sa" CreateProc
```

可以为 insert、select、update 和 call 操作调用 JDBCExamples，如下所示：

```
java JDBCExamples "antibes:4000?user=sa" insert
java JDBCExamples "antibes:4000?user=sa" update
java JDBCExamples "antibes:4000?user=sa" call
java JDBCExamples "antibes:4000?user=sa" select
```

这些调用将显示 “操作已执行” 消息。

要删除 xmp 表和 inoutproc 存储过程，请输入：

```
java JDBCExamples "antibes:4000?user=sa" droptable
java JDBCExamples "antibes:4000?user=sa" dropproc
```

使用 *serverMain( )*

注释 由于服务器端 JDBC 驱动程序不支持 create procedure 或 drop procedure，因此，在执行这些示例之前，请通过客户端的 main( ) 方法调用来创建 xmp 表和示例存储过程 inoutproc。请参阅第 70 页的“JDBCExamples 类的概述”。

在创建 xmp 和 inoutproc 后，可以调用 serverMain( ) 方法，如下所示：

```
select JDBCExamples.serverMain('insert')
go
select JDBCExamples.serverMain('select')
go
select JDBCExamples.serverMain('update')
go
select JDBCExamples.serverMain('call')
go
```

注释 serverMain( ) 的服务器端调用不要求 *server-name:port-number* 参数；Adaptive Server 只是连接到自身。

获取一个 JDBC 连接：*Connecter( )* 方法

main( ) 和 serverMain( ) 都调用 connecter( ) 方法，此方法返回一个 JDBC *Connection* 对象。 *Connection* 对象是所有后续 SQL 操作的基础。

main( ) 和 serverMain( ) 都通过为服务器端或客户端环境指定 JDBC 驱动程序的参数来调用 connecter( )。之后，返回的 *Connection* 对象作为参数传递给 JDBCExamples 类的其它方法。通过隔离 connecter( ) 方法中的连接操作， JDBCExamples 的其它方法将独立于其服务器端或客户端环境。

将操作传递给其它方法：*doAction( )* 方法

doAction( ) 方法基于 *action* 参数传递对其它方法的调用。

doAction( ) 具有 *Connection* 参数，它只将此参数传递给目标方法。它也有一个 *locale* 参数，用以指示调用是在服务器端还是在客户端。如果 create procedure 或 drop procedure 在服务器端环境调用，*Connection* 将引发一个异常。

执行必要的 SQL 操作：*doSQL( )* 方法

doSQL( ) 方法执行不要求输入参数或输出参数的 SQL 操作，例如 create table、 create procedure、 drop table 和 drop procedure。

doSQL( ) 有两个参数：*Connection* 对象和它要执行的 SQL 语句。doSQL( ) 创建一个 JDBC *Statement* 对象，并且用它来执行指定的 SQL 语句。

执行 *update* 语句：*updater( )* 方法

updater( ) 方法执行一个 Transact-SQL update 语句。 update 操作为：

```
String sql = "update xmp set name = ?, home = ? where id = ?";
```

它利用给定的 *id* 值更新所有行的 name 和 home 列。

name 和 home 列的 update 值以及 id 值都是由参数标记 (?) 指定的。在准备语句后，执行之前， updater( ) 将为这些参数标记提供值。这些值是由带有以下参数的 JDBC setString( )、 setObject( ) 和 setInt( ) 方法指定的：

- 将被替代的顺序参数标记
- 将被替代的值

例如：

```
pstmt.setString(1, name);
pstmt.setObject(2, home);
pstmt.setInt(3, id);
```

完成这些替代后， updater( ) 执行 update 语句。

为了简化 updater( )，示例中被替换的值是固定的。正常情况下，应用程序将计算替换值或者将它们作为参数来获得。

执行 *select* 语句：*selecter( )* 方法

selecter( ) 方法执行一个 Transact-SQL select 语句：

```
String sql = "select name, home from xmp where id=?";
```

where 子句使用参数标记 (?) 标记要选择的行。在准备 SQL 语句后， selecter( ) 使用 JDBC setInt( ) 方法为参数标记提供值：

```
PreparedStatement pstmt =
    con.prepareStatement(sql);
pstmt.setInt(1, id);
```

然后， selecter( ) 执行 select 语句：

```
ResultSet rs = pstmt.executeQuery();
```

注释 对于不返回结果的 SQL 语句，请使用 doSQL( ) 和 updater( )。它们使用 executeUpdate( ) 方法执行 SQL 语句。

对于返回结果的 SQL 语句，使用 executeQuery( ) 方法，该方法返回一个 JDBC *ResultSet* 对象。

ResultSet 对象与 SQL 游标类似。开始，它定位在结果的第一行前。每一次调用 next( ) 方法都将 *ResultSet* 对象前进到下一行，直到最后一行。

selecter( ) 要求 *ResultSet* 对象仅包含一行。selecter( ) 调用下一个方法，并检查 *ResultSet* 没有行或有多于一行的情况。

```
if (rs.next()) {
   name = rs.getString(1);
   home = (Address)rs.getObject(2);
   if (rs.next()) {
      throw new Exception("Error:  Select returned multiple rows");
   } else { // No action
   }
} else { throw new Exception("Error:  Select returned no rows");
}
```

在上面的代码中， getString( ) 和 getObject( ) 方法调用检索结果集的第一行中的两列。“(Address)rs.getObject(2)”表达式将第二列作为 Java 对象进行检索，然后将该对象强制指定为 Address 类。如果返回的对象不是 Address，那么将引发一个异常。

selecter( ) 检索单个行，并检查没有行或有多个行的情况。处理多行 *ResultSet* 的应用程序简单地在 next( ) 方法调用中循环，并将每一行作为单行处理。

以批处理模式执行

如果要执行一批 SQL 语句，请确保使用 execute( ) 方法。如果对批处理模式使用 executeQuery( )：

- 如果批处理操作不返回结果集 （不包含 select 语句），批处理执行不会出错。
- 如果批处理操作返回一个结果集，则返回结果语句之后的所有语句都被忽略。如果调用 getXXX( ) 来获得一个输出参数，则将执行剩余语句并关闭当前结果集。
- 如果批处理操作返回多个结果集，将引发一个异常并中止该操作。

使用 execute( ) 可确保在所有情况下完成批处理的执行。

调用 SQL 存储过程：*caller( )* 方法

caller( ) 方法调用 inoutproc 存储过程：

```
create proc inoutproc @id int, @newname varchar(50), @newhome Address,
          @oldname varchar(50) output,  @oldhome Address output as

       select @oldname = name, @oldhome = home from xmp where id=@id
       update xmp set name=@newname, home = @newhome where id=@id
```

此过程有三个输入参数 （*@id*、 *@newname* 和 *@newhome*）和两个输出参数（*@oldname* 和 *@oldhome*）。caller( ) 使用 *@id* 的 ID 值将 xmp 表行的 name 和 home 列设置为 *@newname* 和 *@newhome* 值，并在输出参数 *@oldname* 和 *@oldhome* 中返回这些列以前的值。

inoutproc 过程说明如何在一个 JDBC 调用中提供输入和输出参数。

caller( ) 执行以下调用语句来准备调用语句：

```
CallableStatement cs = con.prepareCall("{call inoutproc (?, ?, ?, ?, ?)}");
```

调用的所有参数都被指定为参数标记 (?)。

caller( ) 使用 JDBC setInt( )、setString( ) 和 setObject( ) 方法提供输入参数的值， doSQL( )、 updatAction( ) 和 selecter( ) 方法中使用了这些方法：

```
cs.setInt(1, id);
cs.setString(2, newName);
cs.setObject(3, newHome);
```

这些 set 方法不适合输出参数。在执行调用语句之前，caller( ) 使用 JDBC registerOutParameter( ) 方法指定输出参数的预期数据类型：

```
cs.registerOutParameter(4, java.sql.Types.VARCHAR);
cs.registerOutParameter(5, java.sql.Types.JAVA_OBJECT);
```

然后 caller( ) 执行调用语句并且使用与 selecter( ) 方法所用的相同 getString( ) 和 getObject( ) 方法来获取输出值：

```
int res = cs.executeUpdate();
String oldName = cs.getString(4);
Address oldHome = (Address)cs.getObject(5);
```

本机 JDBC 驱动程序中的错误处理

Sybase 支持并实现 java.sql.SQLException 和 java.sql.SQLWarning 类中的所有方法。SQLException 提供有关数据库访问错误的信息。SQLWarning 扩展了 SQLException 并提供有关数据库访问警告的信息。

Adaptive Server 所引发的错误按其严重性编号。编号越小表示严重性越低，编号越大则表示严重性越高。将按照严重性对错误进行分组：

- 警告 （EX_INFO：严重级 10) — 转换为 SQLWarnings。
- 异常 （严重级 11 到 18）— 转换为 SQLException。
- 致命错误 （严重级 19 到 24）— 转换为致命的 SQLException。

可通过 JDBC、Adaptive Server 或本机 JDBC 驱动程序引发 SQLException。引发 SqlException 将中止导致该错误的 JDBC 查询。后续系统行为将随捕获到错误的位置不同而发生变化：

- *如果在 Java 中捕获到错误* — 将由 “try” 块和后续 “catch” 块处理错误。

  Adaptive Server 提供了一些特定于 JDBC 驱动程序的扩展 SQLException 错误消息。它们都是 EX_USER （严重级 16）并且都能被捕获。没有特定于驱动程序的 SQLWarning 消息。

- *如果错误不是在 Java 中捕获的* — Java VM 将控制权返回给 Adaptive Server， Adaptive Server 将捕获错误并抛出未处理的 SQLException 错误。

  raiserror 命令常用在存储过程中，用来抛出错误并打印用户定义的错误消息。当通过 JDBC 来执行调用 raiserror 命令的存储过程时，错误将作为严重级为 EX_USER 的内部错误来处理，并将引发一个非致命的 SQLException。

  注释 不能使用 raiserror 命令来访问扩展错误数据；with errordata 子句不适用于 SQLException。

如果某个错误导致事务中止，则后果由调用 Java 方法的事务环境所决定：

- *如果事务包含多条语句* — 事务将中止并将控制权返回给服务器，服务器将回退整个事务。在从服务器返回控制权之前，JDBC 驱动程序将停止处理查询。
- *如果事务包含一条语句* — 事务将中止，回退它所包含的 SQL 语句，并且 JDBC 驱动程序继续处理查询。

下列情况说明了不同的结果。考虑包含这些语句的 Java 方法 jdbcTests.Errorexample()：

```
stmt.executeUpdate("delete from parts where partno = 0"); Q2
stmt.executeQuery("select 1/0"); Q3
stmt.executeUpdate("delete from parts where partno = 10"); Q4
```

包含多条语句的事务将包括下列 SQL 命令：

```
begin transaction
delete from parts where partno = 8 Q1
select JDBCTests.Errorexample()
```

在这种情况下，中止的事务将产生下列操作：

- 在 Q3 中引发一个除零异常。
- Q1 和 Q2 中的更改将被回退。
- 整个事务中止。

包含单条语句的事务包括下列 SQL 命令：

```
set chained off
delete from parts where partno = 8 Q1
select JDBCTests.Errorexample()
```

在这种情况下：

- 在 Q3 中引发一个除零异常。
- 不会回退 Q1 和 Q2 中的更改。
- 该异常是在 JDBCTests.Errorexample 的 “catch” 和 “try” 块中捕获的。
- 不会执行在 Q4 中指定的删除，因为它是在与 Q3 相同的 “try” 和 “catch” 块中处理的。
- 可以执行当前 “try” 和 “catch” 块外部的 JDBC 查询。

JDBCExamples 类

```
// An example class illustrating the use of JDBC facilities
// with the Java in Adaptive Server feature.
//
// The methods of this class perform a range of SQL operations.
// These methods can be invoked either from a Java client,
// using the main method, or from the SQL server, using
// the serverMain method.
//
import java.sql.*;              // JDBC
public class JDBCExamples {
{
```

main() 方法

```
// The main method, to be called from a client-side command line
//
    public static void main(String args[]) {
        if (args.length!=2) {
            System.out.println("\n Usage:    "
                 + "java ExternalConnect server-name:port-number
                 action ");
            System.out.println("  The action is connect, createtable,
              " + "createproc, drop, "
              + "insert, select, update, or call \n" );
            return;
        }
        try{
           String server = args[0];
           String action = args[1].toLowerCase();
           Connection con = connecter(server);
           String workString = doAction( action, con, client);
           System.out.println("\n" + workString + "\n");
        }  catch (Exception e) {
           System.out.println("\n Exception: ");
           e.printStackTrace();
        }
    }
```

serverMain( ) 方法

```
// A JDBCExamples method equivalent to 'main',
// to be called from SQL or Java in the server

    public static String serverMain(String action) {
        try {
           Connection con = connecter("default");
           String workString = doAction(action, con, server);
           return workString;
        }  catch ( Exception e ) {
             if (e.getMessage().equals(null)) {
               return "Exc: " + e.toString();
             } else {
               return "Exc - " + e.getMessage();
             }
        }
    }
```

connecter( ) 方法

```
// A JDBCExamples method to get a connection.
// It can be called from the server with argument 'default',
// or from a client, with an argument that is the server name.

public static  Connection connecter(String server)
          throws Exception, SQLException, ClassNotFoundException  {

       String forName="";
       String url="";

       if (server=="default") {  // server connection to current server
          forName = "sybase.asejdbc.ASEDriver";
          url = "jdbc:default:connection";
       } else if (server!="default") {  //client connection to server
          forName= "com.sybase.jdbc.SybDriver";
          url = "jdbc:sybase:Tds:"+ server;
       }

       String user = "sa";
       String password = "";

       // Load the driver
       Class.forName(forName);
       // Get a connection
```

```
        Connection con = DriverManager.getConnection(url,
           user, password);
        return con;
    }
```

doAction( ) 方法

```
// A JDBCExamples method to route to the 'action' to be performed

    public static String doAction(String action, Connection con,
              String locale)
          throws Exception {

        String createProcScript =
             " create proc inoutproc @id int, @newname varchar(50),
             @newhome Address, "
           + "   @oldname varchar(50) output,  @oldhome Address
             output as "
           + " select @oldname = name, @oldhome = home from xmp
             where id=@id "
           + " update xmp set name=@newname, home = @newhome
             where id=@id ";
        String createTableScript =
          " create  table xmp (id int, name varchar(50),
             home Address)" ;

        String dropTableScript = "drop   table xmp  ";
        String dropProcScript = "drop  proc inoutproc ";
          String insertScript = "insert  into xmp "
          + "values (1, 'Joe Smith', new Address('987 Shore',
          '12345'))";

        String workString = "Action (" + action + ) ;
        if (action.equals("connect")) {
             workString += "performed";
        } else if (action.equals("createtable")) {
             workString += doSQL(con, createTableScript );
        } else if (action.equals("createproc")) {
             if (locale.equals(server))  {
                 throw new exception (CreateProc cannot be performed
                 in the server);
             }  else {
                 workString += doSQL(con, createProcScript );
             }
        } else if (action.equals("droptable")) {
```

```
            workString += doSQL(con, dropTableScript );
        } else if (action.equals("dropproc")) {
            if (locale.equals(server))  {
                throw new exception (CreateProc cannot be performed
                in the server);
            }  else {
                workString += doSQL(con, dropProcScript );
            }
        } else if (action.equals("insert")) {
            workString += doSQL(con, insertScript );
        } else if (action.equals("update")) {
            workString += updater(con);
        } else if (action.equals("select")) {
            workString += selecter(con);
        } else if (action.equals("call")) {
            workString += caller(con);
        } else { return "Invalid action:  " + action ;
        }
        return workString;
    }
```

doSQL( ) 方法

```
// A JDBCExamples method to execute an SQL statement.

    public static String doSQL (Connection con, String action)
           throws Exception {

        Statement stmt = con.createStatement();
        int res = stmt.executeUpdate(action);
        return "performed";
 }
```

updater( ) 方法

```
// A method that updates a certain row of the 'xmp' table.
// This method illustrates prepared statements and parameter markers.

    public static String updater(Connection con)
           throws Exception {

        String sql = "update xmp set name = ?, home = ? where id = ?";
        int id=1;
```

```
        Address home = new Address("123 Main", "98765");
        String name = "Sam Brown";
        PreparedStatement pstmt = con.prepareStatement(sql);
        pstmt.setString(1, name);
        pstmt.setObject(2, home);
        pstmt.setInt(3, id);
        int res = pstmt.executeUpdate();
        return "performed";
    }
```

selecter( ) 方法

```
//  A JDBCExamples method to retrieve a certain row
//  of the 'xmp' table.
//  This method illustrates prepared statements, parameter markers,
//  and result sets.

    public static String selecter(Connection con)
           throws Exception {

        String sql = "select name, home from xmp where id=?";
        int id=1;
        Address home = null;
        String name = "";
        String street = "";
        String zip = "";
        PreparedStatement pstmt = con.prepareStatement(sql);
        pstmt.setInt(1, id);
        ResultSet rs = pstmt.executeQuery();
        if (rs.next()) {
           name = rs.getString(1);
           home = (Address)rs.getObject(2);
           if (rs.next()) {
              throw new Exception("Error:  Select returned
                  multiple rows");
           } else { // No action
           }
        } else { throw new Exception("Error:  Select returned no rows");
        }
        return "- Row with id=1:  name("+ name + )
                + " street(" + home.street + ) zip("+ home.zip + );
```

caller( ) 方法

```
// A JDBCExamples method to call a stored procedure,
// passing input and output parameters of datatype String
 // and Address.
 // This method illustrates callable statements, parameter markers,
 // and result sets.

    public static String caller(Connection con)
           throws Exception {
        CallableStatement cs  = con.prepareCall("{call inoutproc
          (?, ?, ?, ?, ?)}");
        int id = 1;
        String newName = "Frank Farr";
        Address newHome = new Address("123 Farr Lane", "87654");
        cs.setInt(1, id);
        cs.setString(2, newName);
        cs.setObject(3, newHome);
        cs.registerOutParameter(4, java.sql.Types.VARCHAR);
        cs.registerOutParameter(5, java.sql.Types.JAVA_OBJECT);
        int res = cs.executeUpdate();
        String oldName = cs.getString(4);
        Address oldHome = (Address)cs.getObject(5);
        return "- Old values of row with id=1: name("+oldName+ )
          street(" + oldHome.street + ")  zip("+ oldHome.zip + );
    }
}
```

第 6 章

SQLJ 函数和存储过程

本章介绍如何将 Java 方法包装到 SQL 名称中，并将其用作 Adaptive Server 的函数和存储过程。

概述

在 SQL 包装中可包括 Java 静态方法，并且它们的使用方法与 Transact-SQL 存储过程或内置函数完全相同。它的功能包括：

- 允许 Java 方法将输出参数和结果集返回到调用环境。
- 符合 ANSI SQLJ 标准规范的第 1 部分。
- 允许利用传统的 SQL 语法、元数据及权限功能。
- 允许在服务器、客户端和任何符合 SQLJ 的第三方数据库上将现有 Java 方法用作 SQLJ 的过程和函数。

❖ **创建 SQLJ 存储过程或函数**

执行以下步骤创建并执行 SQLJ 存储过程或函数。

1 创建和编译 Java 方法。使用 installjava 实用程序在数据库中安装方法类。

有关在 Adaptive Server 中创建、编译和安装 Java 方法的信息，请参阅第 3 章 “在数据库中准备和维护 Java”。

2 使用 SQLJ create procedure 或 create function 语句定义方法的 SQL 名称。

3 执行过程或函数。本章中的示例使用了 JDBC 方法调用或 isql。也可使用 Embedded SQL 或 ODBC 执行该方法。

符合 SQLJ 第 1 部分规范

Adaptive Server SQLJ 存储过程和函数符合 SQLJ 标准规范的第 1 部分，从而能够在 SQL 中使用 Java。有关 SQLJ 标准的说明，请参见第 6 页的“标准”。

Adaptive Server 支持 SQLJ 第 1 部分规范中介绍的大多数功能；但也存在一些差异。第 110 页的表 6-3 列出了不支持的功能；第 110 页的表 6-4 列出了部分支持的功能。第 111 页的表 6-5 列出了 Sybase 定义的功能（指标准未定义但留待实现的功能）。

在 Sybase 专有实现不同于 SQLJ 规范的情况下， Sybase 支持 SQLJ 标准。例如，非 Java Sybase SQL 存储过程支持两种参数模式：in 和 inout。 SQLJ 标准支持三种参数模式：in、 out 和 inout。创建 SQLJ 存储过程的 Sybase 语法支持全部三种参数模式。

一般问题

本节介绍适用于 SQLJ 函数和存储过程的常见问题及约束。

安全性和权限

Sybase 为 SQLJ 存储过程和 SQLJ 函数提供不同的安全模式。

SQLJ 函数和用户定义的函数 (UDF)（请参见第 37 页的 “在 SQL 中调用 Java 方法”）使用相同的安全模式。将执行任何 UDF 或 SQLJ 函数的权限隐式授予 public。如果函数通过 JDBC 执行 SQL 查询，则将对照函数的调用者来检查访问数据的权限。因此，如果用户 A 调用访问表 t1 的函数，则他必须具有对 t1 的 select 权限，否则查询将失败。

SQLJ 存储过程使用与 Transact-SQL 存储过程相同的安全模式。必须显式授予用户执行 SQLJ 或 Transact-SQL 存储过程的权限。如果 SQLJ 过程通过 JDBC 执行 SQL 查询，则支持隐式授予权限。此安全模式允许存储过程的所有者（如果该所有者拥有该过程引用的所有 SQL 对象）将该过程的执行权限授予另一用户。具有执行权限的用户可以执行存储过程中的所有 SQL 查询，即使用户没有这些对象的访问权限。

通常，在配置并运行 JVM 后，任何用户都可以从可运行 Java 类的数据库中访问这些类。不过，以下操作会受到限制：

- 线程操作，但创建和连接所需的操作除外
- 影响服务器的系统操作，如 exit( ) 和 abort( )
- 对类装载程序层次的更改
- 覆盖安装的安全管理器

有关存储过程安全性的详细说明，请参见 《安全性管理指南》。

SQLJ 示例

本章中使用的示例假设名为 sales_emps 的 SQL 表包含以下列：

- name — 职员的姓名
- id — 职员的标识号
- state — 职员所在的州
- sales — 职员的销售额
- jobcode — 职员的工作代码

表定义为：

```
create table sales_emps
    (name varchar(50), id char(5),
    state char(20), sales decimal (6,2),
    jobcode integer null)
```

示例类为 SQLJExamples，方法为：

- region() — 将美国的州代码映射到区域编号。该方法不使用 SQL。
- correctStates() — 执行 SQL update 命令以更正 state 代码拼写。原有拼写和新拼写由输入参数指定。
- bestTwoEmps() — 按 sales 记录确定业绩最好的两名职员，并将这些值作为输出参数返回。
- SQLJExamplesorderedEmps() — 创建一个 SQL 结果集（包含按 sales 列中的值排序的选定职员行），并将结果集返回到客户端。
- job() — 返回一个与整数工作代码值对应的字符串值。

有关每种方法的内容，请参见第 112 页的 “SQLJExamples 类”。

在 Adaptive Server 中调用 Java 方法

在 Adaptive Server 中可按两种不同方式调用 Java 方法：

- 在 SQL 中直接调用 Java 方法。第 4 章 “在 SQL 中使用 Java 类”提供了以这种方式调用方法的说明。
- 使用为方法名称提供 Transact-SQL 别名的 SQLJ 存储过程和函数间接调用 Java 方法。本章介绍了如何以这种方式调用 Java 方法。

无论选择哪种方式，必须首先创建 Java 方法，并使用 installjava 实用程序在 Adaptive Server 数据库中安装它们。有关详细信息，请参见第 3 章 “在数据库中准备和维护 Java”。

使用 Java 名称直接调用 Java 方法

通过使用完全限定的 Java 名称引用 Java 方法，可在 SQL 中调用它们。可引用实例方法的实例和静态方法的实例或类。

可使用静态方法作为将值返回给调用环境的用户定义函数 (UDF)。在存储过程、触发器、 where 子句、 select 语句或在可以使用内置 SQL 函数的任何位置，都可使用 Java 静态方法作为 UDF。

使用其名称调用 Java 方法时，不能使用将输出参数或结果集返回到调用环境的方法。 Java 方法可处理从 JDBC 连接接收的数据，但该方法只能将在其定义中声明的单一返回值返回给调用环境。

不能使用跨数据库的 UDF 函数调用。

有关以这种方式使用 Java 方法的信息，请参见第 4 章 “在 SQL 中使用 Java 类”。

使用 SQLJ 间接调用 Java 方法

可将 Java 方法作为 SQLJ 函数或存储过程来调用。通过将 Java 方法包装到 SQL 包装中，可充分利用以下功能：

- 可使用 SQLJ 存储过程，将结果集和输出参数返回到调用环境。
- 可利用 SQL 元数据功能。例如，可查看数据库中所有存储过程或函数的列表。
- SQLJ 提供方法的 SQL 名称，它使您可以利用标准的 SQL 权限来保护方法调用。
- Sybase SQLJ 符合公认的 SQLJ 第 1 部分标准，该标准允许在非 Sybase 环境中使用 Sybase SQLJ 过程和函数。
- 可跨数据库调用 SQLJ 函数和 SQLJ 存储过程。
- 因为 Adaptive Server 会在创建 SQLJ 例程时检查数据类型映射，所以执行这些例程时无需考虑数据类型映射。

在 SQLJ 例程中必须引用静态方法；而不能引用实例方法。

本章介绍如何使用 Java 方法作为 SQLJ 存储过程和函数。

使用 Sybase Central 管理 SQLJ 函数和过程

通过在命令行中使用 isql 或通过 Sybase Central 的 Adaptive Server 插件，可以管理 SQLJ 函数和过程。利用 Adaptive Server 插件可以：

- 创建 SQLJ 函数或过程
- 执行 SQLJ 函数或过程
- 查看和修改 SQLJ 函数或过程的属性
- 删除 SQLJ 函数或过程
- 查看 SQLJ 函数或过程的依赖性
- 创建 SQLJ 过程的权限

下列过程描述如何创建和查看 SQLJ 例程的属性。在例程的属性表中，可以查看依赖性以及创建和查看权限。

❖ **创建 SQLJ 函数/过程**

首先，创建并编译 Java 方法，使用 installjava 在数据库中安装方法类，然后执行如下步骤：

1 启动 Adaptive Server 插件并连接到 Adaptive Server。

2 双击要创建例程的数据库。

3 打开 “SQLJ 过程/SQLJ 函数” (SQLJ Procedures/SQLJ Functions) 文件夹。

4 双击 “添加新的 Java 存储过程/函数” (Add new Java Stored Procedure/Function) 图标。

5 使用 “添加新的 Java 存储过程/函数” (Add new Java Stored Procedure/Function) 向导创建 SQLJ 过程或函数。

向导使用完毕后， Adaptive Server 插件将在编辑屏幕中显示所创建的 SQLJ 例程，您可在该屏幕中修改和执行该例程。

❖ **查看 SQLJ 函数或过程的属性**

1 启动 Adaptive Server 插件并连接到 Adaptive Server。

2 双击存储例程的数据库。

3 打开 “SQLJ 过程/SQLJ 函数” (SQLJ Procedures/SQLJ Functions) 文件夹。

4 突出显示函数或过程图标。

5 选择 “文件” (File) | “属性” (Properties)。

SQLJ 用户定义函数

create function 命令指定 Java 方法的 SQLJ 函数名和签名。可使用 SQLJ 函数读取和修改 SQL，并返回由引用的方法描述的值。

create function 的 SQLJ 语法为：

```
create function [owner].sql_function_name
       ([sql_parameter_name sql_datatype
          [( length)| (precision[, scale])]
       [, sql_parameter_name sql_datatype
          [( length )  | ( precision[, scale]) ]]
       ...])
   returns sql_datatype
       [( length)| (precision[, scale])]
   [modifies sql data]
   [returns null on null input |
       called on null input]
   [deterministic | not deterministic]
   [exportable]
   language java
   parameter style java
   external name 'java_method_name
       [([java_datatype[ {, java_datatype }
       ...])]'
```

创建 SQLJ 函数时：

- **SQL 函数签名**是每个函数参数的 SQL 数据类型 *sql_datatype*。
- 为了符合 ANSI 标准，参数名前不要包括符号 @。

  Sybase 会在内部添加 @ 符号以支持参数名绑定。使用 sp_help 输出有关 SQLJ 存储过程的信息时您将看到 @ 符号。
- 在创建 SQLJ 函数时，必须包括将 *sql_parameter_name* 和 *sql_datatype* 信息括起来的小括号 （即使不包含这些信息）。

  例如：

  ```
  create function sqlj_fc()
      language java
      parameter style java
  external name 'SQLJExamples.method'
  ```
- modifies sql data 子句指定，该方法将调用 SQL 操作并读取和修改 SQL 数据。该值为缺省值。您不需要包括它，除非要在语法上符合 SQLJ 第 1 部分标准。

- returns null on null input 和 called on null input 指定 Adaptive Server 如何处理函数调用的空参数。returns null on null input 指定如果任何参数的值在运行时为空，则将函数返回值设置为空并且不调用函数体。called on null input 为缺省值。它指定被调用的函数而不考虑空参数值。

  第 95 页的“处理函数调用中的空值”中详细说明了函数调用和空参数值。

- 可包括 deterministic 或 not deterministic 关键字，但 Adaptive Server 不使用它们。包括它们是为了在语法上符合 SQLJ 第 1 部分标准。
- 子句 exportable 关键字指定函数将使用 Sybase OmniConnect™ 功能在远程服务器上运行。函数及其所基于的方法都必须安装在远程服务器上。
- 子句 language java 和 parameter style java 指明引用方法是用 Java 编写的并且参数是 Java 参数。创建 SQLJ 函数时，*必须* 包括这些短语。
- external name 子句指定不在 SQL 中编写例程并标识 Java 方法、类和包名称 （如果有的话）。
- Java 方法签名指定每个方法参数的 Java 数据类型 *java_datatype*。Java 方法签名是可选的。如果未指定， Adaptive Server 将从 SQL 函数签名推断 Java 方法签名。

  Sybase 建议包括方法签名（像本练习处理所有数据类型转换一样）。请参见第 106 页的 “映射 Java 和 SQL 数据类型”。

- 使用 create function 可为同一 Java 方法定义不同的 SQL 名称，然后以同样的方法使用它们。

编写 Java 方法

创建 SQLJ 函数之前，必须编写它所引用的 Java 方法，编译该方法类，并在数据库中安装它。

在本示例中， SQLJExamples.region() 将州代码映射到区域编号，并将该编号返回给用户。

```
public static int region(String s)
       throws SQLException {
    s = s.trim();
    if (s.equals "MN") || s.equals("VT") ||
        s.equals("NH") ) return 1;
    if (s.equals("FL") || s.equals("GA") ||
        s.equals("AL") ) return 2;
    if (s.equals("CA") || s.equals("AZ") ||
        s.equals("NV") ) return 3;
    else throw new SQLException
        ("Invalid state code", "X2001");

}
```

创建 SQLJ 函数

编写并安装完方法后，即可创建 SQLJ 函数。例如：

```
create function region_of(state char(20))
       returns integer
language java parameter style java
external name
       'SQLJExamples.region(java.lang.String)'
```

SQLJ create function 语句指定输入参数(state char(20))和 integer 返回值。 SQL 函数签名为 char(20)。 Java 方法签名为 java.lang.String。

调用函数

SQLJ 函数如同内置函数一样可以直接调用。例如：

```
select name, region_of(state) as region
   from sales_emps
where region_of(state)=3
```

注释　在 Adaptive Server 中函数的搜索顺序为：

1　内置函数

2　SQLJ 函数

3　直接调用的 Java-SQL 函数

处理空参数值

Java 类数据类型和 Java 原始数据类型使用不同方式处理空参数值。

- **Java 对象数据类型**（如 java.lang.Integer、 java.lang.String、java.lang.byte[] 和 java.*sql*.Timestamp）属于类，可以保存实际值和空引用值。
- **Java 原始数据类型**（如 boolean、 byte、 short 和 int）没有空值表示形式。它们只能保存非空值。

如果对 Java 方法的调用导致将一个 SQL 空值作为参数传递给数据类型为 Java 类的 Java 参数，那么该 SQL 空值将作为 Java 空引用值传递。但是，如果将一个 SQL 空值作为参数传递给 Java 原始数据类型的 Java 参数，将引发一个异常，这是因为 Java 原始数据类型没有空值表示形式。

通常，在编写 Java 方法时会指定属于类的 Java 参数数据类型。在这种情况下，处理空值不会引发异常。如果选择编写使用 Java 参数（这些参数不能处理空值）的 Java 函数，则可以：

- 创建 SQLJ 函数时，包括 returns null on null input 子句，或者
- 使用 case 或其它条件表达式调用 SQLJ 函数以测试空值并仅对非空值调用 SQLJ 函数。

创建 SQLJ 函数或调用它时，可处理预期的空值。下面的部分将说明这两种情形并引用此方法：

```
public static String job(int jc)
       throws SQLException {
if (jc==1) return "Admin";
else if (jc==2) return "sales";
else if (jc==3) return "clerk";
else return "unknown jobcode";
}
```

创建函数时处理空值

如果预计到有空值，可在创建函数时包括 returns null on null input 子句。例如：

```
create function job_of(jc integer)
    returns varchar(20)
returns null on null input
language java parameter style java
external name 'SQLJExamples.job(int)'
```

然后，可按如下方式调用 job_of：

```
select name, job_of(jobcode)
    from sales_emp
where job_of(jobcode) <> "Admin"
```

在 SQL 系统对 jobcode 列为空的某行 sales_emps 的调用 job_of(jobcode) 求值时，调用的值将设置为空，而不实际调用 Java 方法 SQLJExamples.job。对于 jobcode 列有非空值的行，调用正常执行。

因此，当使用 returns null on null input 子句创建的 SQLJ 函数遇到空参数时，函数调用结果将设置为空，而不调用该函数。

注释 如果创建 SQLJ 函数时包括 returns null on null input 子句，则 returns null on null input 子句将应用于*所有*函数参数，包括可为空的参数。

如果包括 called on null input 子句 （缺省情况下），不可为空的参数的空参数将产生异常。

处理函数调用中的空值

可使用条件函数调用来处理不可为空的参数的空值。下列示例使用 case 表达式：

```
select name,
    case when jobcode is not null
       then job_of(jobcode)
       else null end
from sales_emps where
    case when jobcode is not null
       then job_of(jobcode)
       else null end <> "Admin"
```

在此示例中，假设函数 job_of 是使用缺省子句 called on null input 创建的。

删除 SQLJ 函数名

使用 drop function 命令，可删除 Java 方法的 SQLJ 函数名。例如，输入：

```
drop function region_of
```

这会删除 region_of 函数名及其对 SQLJExamples.region 方法的引用。 drop function 不会影响引用的 Java 方法或类。

有关完整的语法和用法信息，请参见 《参考手册：构件块》。

SQLJ 存储过程

使用 Java-SQL 功能，可在数据库中安装 Java 类，然后从客户端或 SQL 系统内部调用这些方法。也可以使用另一种方式调用 Java 静态（类）方法，即，以 SQLJ 存储过程的形式。

SQLJ 存储过程：

- 可将结果集和/或输出参数返回到客户端
- 它在执行时的行为与 Transact-SQL 存储过程完全一样
- 可以使用 ODBC、 isql 或 JDBC 从客户端调用

- 可从服务器内的其它存储过程或本机 Adaptive Server JDBC 调用

最终用户不必知道被调用的过程是 SQLJ 存储过程还是 Transact-SQL 存储过程。它们的调用方式均相同。

create procedure 的 SQLJ 语法为：

```
create procedure [owner.]sql_procedure_name
    ([[ in | out | inout ] sql_parameter_name
       sql_datatype [( length) |
       (precision[, scale])]
    [, [ in | out | inout ] sql_parameter_name
       sql_datatype [( length) |
       (precision[, scale]) ]]
    ...])
    [modifies sql data]
    [dynamic result sets integer]
    [deterministic | not deterministic]
    language java
    parameter style java
    external name 'java_method_name
       [([java_datatype[, java_datatype
    ...]])]'
```

注释 为了符合 ANSI 标准，SQLJ create procedure 命令语法不同于创建 Sybase Transact-SQL 存储过程时使用的语法。

有关此命令中各关键字和选项的详细说明，请参见《参考手册：命令》。

创建 SQLJ 存储过程时：

- **SQL 过程签名**是每个过程参数的 SQL 数据类型 *sql_datatype*。
- 创建 SQLJ 存储过程时，参数名前不要包括符号 @。这样做是为了符合 ANSI 标准。

  Sybase 会在内部添加 @ 符号以支持参数名绑定。使用 sp_help 输出有关 SQLJ 存储过程的信息时您将看到 @ 符号。

- 在创建 SQLJ 存储过程时，必须包括将 *sql_parameter_name* 和 *sql_datatype* 信息括起来的小括号 （即使不包含这些信息）。

  例如：

  ```
  create procedure sqlj_sproc ()
     language java
     parameter style java
  external name "SQLJExamples.method1"
  ```

- 您可以包括 modifies *sql* data 关键字以表示该方法将调用 SQL 操作并读取和修改 SQL 数据。该值为缺省值。
- 如果要将结果集返回到调用环境，必须包括 dynamic result sets 的 *integer* 选项。使用 *integer* 变量可指定期望结果集的最大数量。
- 可包括关键字 deterministic 或 not deterministic，以符合 SQLJ 标准。但 Adaptive Server 不使用此选项。
- 必须包括 language java parameter 和 style java 关键字，它们可告诉 Adaptive Server 外部例程是用 Java 编写的，传递到外部例程的参数运行时约定是 Java 约定。
- external name 子句表示外部例程是用 Java 编写的，并标识 Java 方法、类和包名称 （如果有的话）。
- Java 方法签名指定每个方法参数的 Java 数据类型 *java_datatype*。Java 方法签名是可选的。如果未指定， Adaptive Server 将从 SQL 过程签名中推断一个。

  Sybase 建议包括方法签名（像本练习处理所有数据类型转换一样）。有关详细信息，请参见第 106 页的 “映射 Java 和 SQL 数据类型”。
- 使用 create procedure 可为同一 Java 方法定义不同的 SQL 名称，然后以同样的方法使用它们。

修改 SQL 数据

可使用 SQLJ 存储过程修改数据库中的信息。 SQLJ 过程引用的方法必须为下列之一：

- 一个无类型方法，或
- 具有 int 返回类型的方法 （包括 int 返回类型是 SQLJ 标准的 Sybase 扩展）。

编写 Java 方法

SQLJExamples.correctStates() 方法执行 SQL update 语句以更正州代码拼写。输入参数指定了新旧拼写。 correctStates() 是无类型方法；不向调用者返回值。

```
public static void correctStates(String oldSpelling,
String newSpelling) throws SQLException {

    Connection conn = null;
    PreparedStatement pstmt = null;
    try {
        Class.forName("sybase.asejdbc.ASEDriver");
        conn = DriverManager.getConnection
            ("jdbc:default:connection");
    }
    catch (Exception e) {
        System.err.println(e.getMessage() +
            ":error in connection");
    }
    try {
        pstmt = conn.prepareStatement
            ("UPDATE sales_emps SET state = ?
            WHERE state = ?");
        pstmt.set.String(1, newSpelling);
        pstmt.set.String(2, oldSpelling);
        pstmt.executeUpdate();
    }
    catch (SQLException e) {
        System.err.println("SQLException: "+
        e.getErrorCode() + e.getMessage());
    }
    return;
}
```

创建存储过程

在使用 SQL 名称调用 Java 方法之前，必须使用 SQLJ create procedure 命令为其创建 SQL 名称。 modifies sql data 子句是可选的。

```
create procedure correct_states(old char(20),
       not_old char(20))
    modifies sql data
    language java parameter style java
    external name
        'SQLJExamples.correctStates
        (java.lang.String, java.lang.String)'
```

correct_states 过程的 SQL 过程签名为 char(20)、 char(20)。 Java 方法签名为 java.lang.String, java.lang.String。

调用存储过程

执行 SQLJ 过程与执行 Transact-SQL 过程完全相同。在此示例中将从 isql 执行过程：

```
execute correct_states 'GEO', 'GA'
```

使用输入和输出参数

Java 方法不支持输出参数。但在 SQL 中包装 Java 方法时，可利用 Sybase SQLJ 功能，它允许对 SQLJ 存储过程使用输入、输出以及输入/输出参数。

创建 SQLJ 过程时，将每个参数的模式分别标识为 in、 out 或 inout。

- 对于输入参数，请使用 in 关键字限定参数。 in 为缺省值；如果未输入参数模式， Adaptive Server 将使用输入参数。
- 对于输出参数，使用 out 关键字。
- 对于可将值从引用的 Java 方法中传入和传出的参数，使用 inout 关键字。

注释　仅使用 in 和 out 关键字创建 Transact-SQL 存储过程。out 关键字与 SQLJ inout 关键字相对应。有关详细信息，请参见 《参考手册：命令》 中的 create procedure 参考页。

若要创建定义输出参数的 SQLJ 存储过程，必须：

- 在创建 SQLJ 存储过程时，使用 out 或 inout 选项定义输出参数。
- 在 Java 方法中将这些参数声明为 Java 数组。 SQLJ 将数组作为方法的输出参数值的容器。

  例如，如果希望 Integer 参数向调用者返回值，则必须在方法中将参数类型指定为 Integer[ ]（Integer 数组）。

  out 或 inout 参数的数组对象是由系统隐式创建的。它具有单个元素。在调用 Java 方法之前，输入值（如果有的话）放在数组的第一个元素（仅有的）中。当 Java 方法返回时，第一个元素被删除并分配给输出变量。通常，此元素将由调用方法分配新值。

以下示例使用 Java 方法 bestTwoEmps() 和引用该方法的存储过程 best2 说明输出参数的使用。

编写 Java 方法

SQLJExamples.bestTwoEmps() 方法返回具有最高销售业绩记录的两名职员的姓名、ID、所在区域和销售额。前八个参数是需要包含数组的输出参数。第九个参数是输入参数，不需要数组。

```
public static void bestTwoEmps(String[] n1,
   String[] id1, int[] r1,
   BigDecimal[] s1, String[] n2,
   String[] id2, int[] r2, BigDecimal[] s2,
   int regionParm) throws SQLException {

n1[0] = "****";
id1[0] = "";
r1[0] = 0;
s1[0] = new BigDecimal(0):
n2[0] = "****",
id2[0] = "";
r2[0] = 0;
s2[0] = new BigDecimal(0);

try {
   Connection conn = DriverManager.getConnection
      ("jdbc:default:connection");
   java.sql.PreparedStatement stmt =
      conn.prepareStatement("SELECT name, id,"
      + "region_of(state) as region, sales FROM"
      + "sales_emps WHERE"
      + "region_of(state)>? AND"
      + "sales IS NOT NULL ORDER BY sales DESC");
   stmt.setInteger(1, regionParm);
   ResultSet r = stmt.executeQuery();

   if(r.next()) {
      n1[0] = r.getString("name");
      id1[0] = r.getString("id");
      r1[0] = r.getInt("region");
      s1[0] = r.getBigDecimaL("sales");
   }
   else return;

   if(r.next()) {
      n2[0] = r.getString("name");
      id2[0] = r.getString("id");
      r2[0] = r.getInt("region");
      s2[0] = r.getBigDecimal("sales");
   }
   else return;
}
```

```
    catch (SQLException e) {
       System.err.println("SQLException: "+
          e.getErrorCode() + e.getMessage());
       }
    }
```

创建 SQLJ 过程

为 bestTwoEmps 方法创建 SQL 名称。前八个参数是输出参数；第九个是输入参数。

```
create procedure best2
    (out n1 varchar(50), out id1 varchar(5),
    out s1 decimal(6,2), out r1 integer,
    out n2 varchar(50), out id2 varchar(50),
    out r2 integer, out s2 decimal(6,2),
    in region integer)
    language java
    parameter style java
    external name
       'SQLJExamples.bestTwoEmps (java.lang.String,
       java.lang.String, int, java.math.BigDecimal,
       java.lang.String, java.lang.String, int,
       java.math.BigDecimal, int)'
```

best2 的 SQL 过程签名为：varchar(20)、varchar(5)、decimal (6,2) 等。Java 方法签名为 String、 int、 BigDecimal 等等。

调用过程

在数据库中安装该方法且创建了引用该方法的 SQLJ 过程后，即可调用 SQLJ 过程。

在运行时， SQL 系统：

1　在调用 SQLJ 过程时，为 out 和 inout 参数创建必需的数组。

2　从 SQLJ 过程中返回时，将参数数组的内容复制到 out 和 inout 目标变量中。

以下示例从 isql 调用 best2 过程。 region 输入参数的值指定区域编号。

```
declare @n1 varchar(50), @id1 varchar(5),
@s1 decimal (6,2), @r1 integer, @n2 varchar(50),
@id2 varchar(50), @r2 integer, @s2 decimal(6,2),
@region integer
select @region = 3
execute best2 @n1 out, @id1 out, @s1 out, @r1 out,
@n2 out, @id2 out, @r2 out, @s2 out, @region
```

注释 Adaptive Server 调用 SQLJ 存储过程与其调用 Transact-SQL 存储过程完全相同。因此，使用 isql 或任何其它非 Java 客户端时，必须在参数名前加 @ 符号。

返回结果集

SQL 结果集是传递给调用环境的一系列 SQL 行。

Transact-SQL 存储过程返回一个或多个结果集时，这些结果集是过程调用的隐式输出。即，它们不声明为显式参数或返回值。

Java 方法可以返回 Java 结果集对象，但它们作为显式声明的方法值执行此操作。

若要从 Java 方法返回 SQL 风格的结果集，必须先将 Java 方法包装到 SQLJ 存储过程中。作为 SQLJ 存储过程调用方法时，结果集（由 Java 方法作为 Java 结果集对象返回）将由服务器转换为 SQL 结果集。

当编写作为 SQLJ 过程来调用并返回 SQL 风格的结果集的 Java 方法时，必须为方法可返回的每个结果集的方法指定一个额外的参数。每个此类参数都是 Java ResultSet 类的单个元素数组。

本节将介绍编写方法、创建 SQLJ 存储过程和调用方法的基本过程。有关返回结果集的详细信息，请参见第 107 页的“显式或隐式指定 Java 方法签名”。

编写 Java 方法

以下方法 SQLJExamples.orderedEmps 可调用 SQL、包括 ResultSet 参数以及使用 JDBC 调用以确保安全连接和打开语句。

```
public static void orderedEmps
       (int regionParm, ResultSet[] rs) throws
       SQLException {

   Connection conn = null;
   PreparedStatement pstmt = null;

   try {
      Class.forName
         ("sybase.asejdbc.ASEDriver");
      Connection conn =
         DriverManager.getConnection
         ("jdbc:default:connection");
   }
```

```
        catch (Exception e) {
            System.err.println(e.getMessage()
                + ":error in connection");
        }

        try {
            java.sql.PreparedStatement
                stmt = conn.prepareStatement
                ("SELECT name, region_of(state)"
                "as region, sales FROM sales_emps"
                "WHERE region_of(state) > ? AND"
                "sales IS NOT NULL"
                "ORDER BY sales DESC");
            stmt.setInt(1, regionParm);
            rs[0] = stmt.executeQuery();
            return;
        }
        catch (SQLException e)
            System.err.println("SQLException:"
            + e.getErrorCode() + e.getMessage());
        }
        return;
    }
```

orderedEmps 返回单个结果集。也可编写返回多个结果集的方法。对于每个返回的结果集，必须：

- 在方法签名中包括单独的 ResultSet 数组参数。
- 创建每个结果集的 Statement 对象。
- 将所有结果集都分配到其 ResultSet 数组的第一个元素。

Adaptive Server 始终返回每个 Statement 对象的当前打开的 ResultSet 对象。创建返回结果集的 Java 方法时：

- 创建要返回到客户端的每个结果集的 Statement 对象。
- 不要显式关闭 ResultSet 和 Statement 对象。 Adaptive Server 会自动将其关闭。

  注释　Adaptive Server 将确保受到影响的结果集已被处理并返回到客户端后， ResultSet 和 Statement 才被碎片收集关闭。

- 如果结果集的某些行是通过调用 Java next() 方法读取的，则仅有剩余的结果集行返回给客户端。

创建 SQLJ 存储过程

创建返回结果集的 SQLJ 存储过程时，必须指定最多可返回的结果集数。在本示例中， ranked_emps 过程返回单个结果集。

```
create procedure ranked_emps(region integer)
    dynamic result sets 1
    language java parameter style java
    external name 'SQLJExamples.orderedEmps(int,
       ResultSet[]'
```

如果 ranked_emps 产生的结果集比 create procedure 指定的多，将显示警告消息并且该过程只返回指定数量的结果集。按照规定， ranked_emps SQLJ 存储过程仅与一个 Java 方法匹配。

注释 当推断某一包括结果集的方法签名时，某些限制适用于方法重载。有关详细信息，请参见第 106 页的 “映射 Java 和 SQL 数据类型”。

调用过程

在数据库中安装方法的类并创建引用该方法的 SQLJ 存储过程后，即可调用该过程。可使用处理 SQL 结果集的任何机制编写调用。

例如，若要使用 JDBC 调用 ranked_emps 过程，请输入以下内容：

```
java.sql.CallableStatement stmt =
    conn.prepareCall("{call ranked_emps(?)}");
stmt.setInt(1,3);
ResultSet rs = stmt.executeQuery();
while (rs.next()) {
    String name = rs.getString(1);
    int.region = rs.getInt(2);
    BigDecimal sales = rs.get.BigDecimal(3);
    System.out.print("Name = " + name);
    System.out.print("Region = "+ region);
    System.out.print("Sales = "+ sales);
    System.out.printIn():
}
```

ranked_emps 过程仅提供在 create procedure 语句中声明的参数。 SQL 系统提供 ResultSet 参数的空数组并调用 Java 方法，该方法将输出结果集分配给数组参数。Java 方法完成后，SQL 系统将输出数组元素中的结果集作为 SQL 结果集返回。

注释 仅当使用外部 JDBC 驱动程序 （如 jConnect）时，才可从临时表中返回结果集。对于此任务，不能使用 Adaptive Server 本机 JDBC 驱动程序。

删除 SQLJ 存储过程名

使用 drop procedure 命令可删除 Java 方法的 SQLJ 存储过程名。例如，输入：

```
drop procedure correct_states
```

这会删除 correct_states 过程名及其对 SQLJExamples.correctStates 方法的引用。 drop procedure 不会影响该过程引用的 Java 类和方法。

查看有关 SQLJ 函数和过程的信息

多个系统存储过程可提供有关 SQLJ 例程的信息：

- sp_depends 列出由 SQLJ 例程引用的数据库对象和引用 SQLJ 例程的数据库对象。
- sp_help 列出 SQLJ 例程的每个参数名、类型、长度、精度、标度、参数顺序、参数模式及返回类型。
- sp_helpjava 列出数据库中安装的 Java 类和 JAR 的相关信息。depends 参数列出在 SQLJ create function 或 SQLJ create procedure 语句的 external name 子句中命名的指定类的依赖性。
- sp_helprotect 报告 SQLJ 存储过程和 SQLJ 函数的权限。

有关这些系统过程的完整语法和用法信息，请参见《参考手册：过程》。

高级主题

下列主题向高级用户提供了有关 SQLJ 主题的详细说明。

映射 Java 和 SQL 数据类型

创建引用 Java 方法的存储过程或函数时，如果值从 SQL 环境转换到 Java 环境，然后进行反向转换，则输入和输出参数或结果集的数据类型不能互相冲突。这种映射的产生规则与 JDBC 标准实现一致。如下面和第 106 页的表 6-1中所示。

每个 SQL 参数及其相应的 Java 参数都必须是可映射的。SQL 和 Java 数据类型在下列情况下是可映射的：

- 如果按表 6-1 中指定的那样，SQL 数据类型和原始 Java 数据类型是*完全可映射的*。
- 如果按表 6-1 中指定的那样，SQL 数据类型和非原始 Java 数据类型是*对象可映射的*。
- 如果 SQL 抽象数据类型 (ADT) 和非原始 Java 数据类型是相同的类或接口，则它们是 *ADT 可映射的*。
- 如果 Java 数据类型是数组，并且 SQL 数据类型对 Java 数据类型是完全可映射的、对象可映射的或 ADT 可映射的，则 SQL 数据类型和 Java 数据类型是*输出可映射的*。例如， character 和 String[ ] 是输出可映射的。
- 如果 Java 数据类型是面向结果集的类 (java.*sql*.ResultSet) 的数组，则它是*结果集可映射的*。

通常，如果 Java 方法的每个参数都可以映射到 SQL 并且其结果集参数是结果集可映射的，同时返回类型是可映射的 （函数）或是 void 或 int （过程），则 Java 方法可映射到 SQL。

支持 SQLJ 存储过程的 int 返回类型是 SQLJ 第 1 部分标准的 Sybase 扩展。

表 6-1：完全可映射的和对象可映射的 SQL 和 Java 数据类型

	对应 Java 数据类型	
SQL 数据类型	**完全可映射的**	**对象可映射的**
char/unichar		java.lang.String
nchar		java.lang.String
varchar/univarchar		java.lang.String
nvarchar		java.lang.String
text		java.lang.String
numeric		java.math.BigDecimal
decimal		java.math.BigDecimal
money		java.math.BigDecimal
smallmoney		java.math.BigDecimal
bit	boolean	Boolean

SQL 数据类型	对应 Java 数据类型	
	完全可映射的	对象可映射的
tinyint	byte	Integer
smallint	short	Integer
integer	int	Integer
bigint	long	java.math.BigInteger
unsigned smallint	int	Integer
unsigned int	long	Integer
unsigned bigint		java.math.BigInteger
real	float	Float
float	double	Double
double precision	double	Double
binary		byte[]
varbinary		byte[]
datetime		java.sql.Timestamp
smalldatetime		java.sql.Timestamp
date		java.sql.Date
time		java.sql.Time

显式或隐式指定 Java 方法签名

创建 SQLJ 函数或存储过程时，通常指定 Java 方法签名。也可以让 Adaptive Server 根据本节前面部分和表 6-1 中介绍的标准 JDBC 数据类型的对应规则，从例程的 SQL 签名中推断 Java 方法签名。

Sybase 建议为确保所有数据类型转换都按指定的来处理，应像本练习一样包括 Java 方法签名。

可允许 Adaptive Server 为下列数据类型推断方法签名：

- 完全可映射的
- ADT 可映射的
- 输出可映射的
- 结果集可映射的

例如，如果希望 Adaptive Server 推断 correct_states 的方法签名，则 create procedure 语句为：

```
create procedure correct_states(old char(20),
       not_old char(20))
   modifies sql data
   language java parameter style java
   external name 'SQLJExamples.correctStates'
```

Adaptive Server 可推断 java.lang.String 和 java.lang.String 的 Java 方法签名。如果显式添加 Java 方法签名，则 create procedure 语句如下所示：

```
create procedure correct_states(old char(20),
       not_old char(20))
    modifies sql data
    language java parameter style java
    external name 'SQLJExamples.correctStates
       (java.lang.String, java.lang.String)'
```

*必须*为对象可映射的数据类型显式指定 Java 方法签名。否则，Adaptive Server 将推断原始的、完全可映射的数据类型。

例如，SQLJExamples.job 方法包含类型为 int 的参数。（请参见第 93 页的“处理空参数值”。）创建引用该方法的函数时，Adaptive Server 将推断 int 的 Java 签名，您不需要指定它。

但是，需要假定 SQLJExamples.job 的参数是 Java Integer，它是对象可映射类型。例如：

```
public class SQLJExamples {
    public static String job(Integer jc)
           throws SQLException ...
```

然后，在创建引用 Java 方法的函数时，必须指定 Java 方法签名：

```
create function job_of(jc integer)
...
external name
    'SQLJExamples.job(java.lang.Integer)'
```

返回结果集和方法重载

创建返回结果集的 SQLJ 存储过程时，指定最多可返回的结果集数。

如果指定 Java 方法签名，Adaptive Server 将查找与方法名称和签名匹配的单个方法。例如：

```
create procedure ranked_emps(region integer)
    dynamic result sets 1
    language java parameter style java
    external name 'SQLJExamples.orderedEmps
       (int, java.sql.ResultSet[])'
```

在这种情况下，Adaptive Server 使用常规 Java 重载约定来解析参数类型。

但要假定未指定 Java 方法签名：

```
create procedure ranked_emps(region integer)
    dynamic result sets 1
    language java parameter style java
    external name 'SQLJExamples.orderedEmps'
```

如果存在两个方法，一个带有 int、 RS[ ] 签名，另一个带有 int、 RS[ ]、 RS[ ] 签名，则 Application Server 将无法区分这两个方法，因而该过程将失败。如果返回结果集时允许 Adaptive Server 推断 Java 方法签名，则确保*仅有一个方法*满足推断条件。

注释　指定的动态结果集数量仅影响可返回的结果的最大数量，而不会影响方法重载。

确保签名有效

如果安装的类已修改，则当调用引用该类的 SQLJ 过程或函数时，Adaptive Server 将进行检查以确保方法签名有效。如果修改后的方法的签名仍有效，则 SQLJ 例程的执行将成功。

使用命令 main 方法

在 Java 客户端，通常通过类的命令 main 方法运行 Java 虚拟机 (VM) 来开始 Java 应用程序。例如，JDBCExamples 类就包含有 main 方法。通过以下方式从命令行执行该类时，将执行命令 main 方法：

```
java JDBCExamples
```

注释　在 SQLJ create function 语句中，不能引用 Java main 方法。

如果在 SQLJ create procedure 语句中引用 Java main 方法，则命令 main 方法必须具有 Java 方法签名 String[]，如下所示：

```
public static void main(java.lang.String[]) {
...
}
```

如果在 create procedure 语句中指定 Java 方法签名，则它必须被指定为 `(java.lang.String[])`。如果未指定 Java 方法签名，则假定它为 `(java.lang.String[])`。

如果 SQL 过程签名包含参数，则这些参数必须为 char、unichar、varchar 或 univarchar。在运行时，它们作为 java.lang.String 的 Java 数组传递。

提供给 SQLJ 过程的每个参数必须为 char、 unichar、 varchar、 univarchar 或文字字符串，原因是它将作为 java.lang.String 数组的一个元素传递给 main 方法。创建 main 过程时，不能使用 dynamic result sets 子句。

SQLJ 和 Sybase 实现：比较

本节介绍 SQLJ 第 1 部分标准规范与 SQLJ 存储过程和函数的 Sybase 专有实现之间的区别。

表 6-2 介绍了 Adaptive Server 对 SQLJ 实现的改进。

表 6-2：Sybase 改进

类别	SQLJ 标准	Sybase 实现
create procedure 命令	仅支持不返回值的 Java 方法。方法的返回类型必须为无类型。	支持允许返回整型值的 Java 方法。在 create procedure 中引用的方法可有整型或无类型返回类型。
create procedure 和 create function 命令	在 create procedure 或 create function 参数列表中仅支持 SQL 数据类型。	支持将 SQL 数据类型和非原始 Java 数据类型作为抽象数据类型 (ADT)。
SQLJ 函数和 SQLJ 过程调用	不支持隐式将 SQL 转换为 SQLJ 数据类型。	支持隐式将 SQL 转换为 SQLJ 数据类型。
SQLJ 函数	不允许 SQLJ 函数在远程服务器上运行。	允许 SQLJ 函数利用 Sybase OmniConnect 功能在远程服务器上运行。
drop procedure 和 drop function 命令	需要完整命令名：drop procedure 或 drop function。	支持完整函数名和简化名称：drop proc 和 drop func。

表 6-3 介绍了 Sybase 实现中未包含的 SQLJ 标准功能。

表 6-3：不支持的 SQLJ 功能

SQLJ 类别	SQLJ 标准	Sybase 实现
create function 命令	允许用户为多个 SQLJ 函数指定相同的 SQL 名称。	对于所有存储过程和函数都需要唯一名称。
实用程序	支持 sqlj.install_jar、sqlj.replace_jar、sqlj.remove_jar 和类似的实用程序安装、替换和删除 JAR 文件。	支持 installjava 实用程序和 remove java Transact-SQL 命令执行类似的功能。

表 6-4 介绍 Sybase 实现部分支持的 SQLJ 标准功能。

表 6-4：部分支持的 SQLJ 功能

SQLJ 类别	SQLJ 标准	Sybase 实现
create procedure 和 create function 命令	允许用户在同一数据库中安装具有相同名称的不同类，但前提是它们存在于不同的 JAR 文件中。	在相同数据库中需要唯一类名称。
create procedure 和 create function 命令	支持关键字 no *sql*、 contains *sql*、reads *sql* data 和 modifies *sql* data 来指定 Java 方法可执行的 SQL 操作。	仅支持 modifies *sql* data。

SQLJ 类别	SQLJ 标准	Sybase 实现
create procedure 命令	支持 java.*sql*.ResultSet 和 SQL/OLB 迭代程序声明。	仅支持 java.*sql*.ResultSet。
drop procedure 和 drop function 命令	支持关键字 restrict，它要求用户在删除过程或函数之前，先删除调用该过程或函数的所有 SQL 对象 （表、视图和例程）。	不支持 restrict 关键字和功能。

表 6-5 介绍了 SQLJ 实现在 Sybase 实现中定义的功能。

表 6-5：由实现定义的 SQLJ 功能

SQLJ 类别	SQLJ 标准	Sybase 实现
create procedure 和 create function 命令	支持 deterministic |not deterministic 关键字，它指定过程或函数是否始终为 out 和 inout 参数和函数结果返回相同的值。	仅支持 deterministic | not deterministic 的语法，而不支持功能。
create procedure 和 create function 命令	在执行 create 命令时或调用过程或函数时，可进行 SQL 签名与 Java 方法签名之间的映射检验。实现将定义何时执行检验。	如果引用的类已更改，则在执行 create 命令时进行所有检验，这可加快执行速度。
create procedure 和 create function 命令	在配置描述符文件中或在 SQL DDL 语句中，可指定 create procedure 或 create function 命令。实现将定义以哪些方式支持命令。	支持在配置描述符外部用作 SQL DDL 语句的 create procedure 和 create function。
调用 SQLJ 例程	在 Java 方法执行 SQL 语句时，任何异常条件都将作为 Exception.*sql*Exception 子类的 Java 异常在该 Java 方法中引发。异常条件的影响由实现定义。	遵循 Adaptive Server JDBC 的规则。
调用 SQLJ 例程	实现将定义用 SQL 名称调用的 Java 方法使用创建过程或函数的用户的特权，还是过程或函数的调用者的特权来执行。	SQLJ 过程和函数将分别继承 SQL 存储过程和 Java-SQL 函数的安全特性。
drop procedure 和 drop function 命令	在配置描述符文件或 SQL DDL 语句中，可指定 drop procedure 或 drop function 命令。实现将定义以哪些方式支持命令。	支持在配置描述符外部用作 SQL DDL 语句的 create procedure 和 create function。

SQLJExamples 类

本节显示了用于说明 SQLJ 存储过程和函数的 SQLJExamples 类。

```
import java.lang.*;
import java.sql.*;
import java.math.*;

static String _url = "jdbc:default:connection";

public class SQLExamples {

    public static int region(String s)
           throws SQLException {
       s = s.trim();
       if (s.equals("MN") || s.equals("VT") ||
          s.equals("NH") ) return 1;
       if (s.equals("FL") || s.equals("GA") ||
          s.equals("AL") ) return 2;
       if (s.equals("CA") || s.equals("AZ") ||
          s.equals("NV") ) return 3;
       else throw new SQLException
          ("Invalid state code", "X2001");

    }
    public static void correctStates
       (String oldSpelling, String newSpelling)
       throws SQLException {

       Connection conn = null;
       PreparedStatement pstmt = null;
       try {
          Class.forName
             ("sybase.asejdbc.ASEDriver");
          conn = DriverManager.getConnection(_url);
       }
       catch (Exception e) {
          System.err.println(e.getMessage() +
             ":error in connection");
       }
       try {
          pstmt = conn.prepareStatement
             ("UPDATE sales_emps SET state = ?
             WHERE state = ?");
          pstmt.setString(1, newSpelling);
          pstmt.setString(2, oldSpelling);
          pstmt.executeUpdate();
       }
```

```
    catch (SQLException e) {
        System.err.println("SQLException: "+
        e.getErrorCode() + e.getMessage());
    }

}
public static String job(int jc)
        throws SQLException {
    if (jc==1) return "Admin";
    else if (jc==2) return "Sales";
    else if (jc==3) return "Clerk";
    else return "unknown jobcode";
}
public static String job(int jc)
        throws SQLException {
if (jc==1) return "Admin";
else if (jc==2) return "Sales";
else if (jc==3) return "Clerk";
else return "unknown jobcode";
}

public static void bestTwoEmps(String[] n1,
    String[] id1, int[] r1,
    BigDecimal[] s1, String[] n2,
    String[] id2, int[] r2, BigDecimal[] s2,
    int regionParm) throws SQLException {

n1[0] = "****";
id1[0] = "";
r1[0] = 0;
s1[0] = new BigDecimal(0):
n2[0] = "****";
id2[0] = "";
r2[0] = 0;
s2[0] = new BigDecimal(0);

try {
    Connection conn = DriverManager.getConnection
        ("jdbc:default:connection");
    java.sql.PreparedStatement stmt =
        conn.prepareStatement("SELECT name, id,"
        + "region_of(state) as region, sales FROM"
        + "sales_emps WHERE"
        + "region_of(state)>? AND"
        + "sales IS NOT NULL ORDER BY sales DESC");
    stmt.setInteger(1, regionParm);
    ResultSet r = stmt.executeQuery();
```

```
    if(r.next()) {
        n1[0] = r.getString("name");
        id1[0] = r.getString("id");
        r1[0] = r.getInt("region");
        s1[0] = r.getBigDecimal("sales");
    }
    else return;

    if(r.next()) {
        n2[0] = r.getString("name");
        id2[0] = r.getString("id");
        r2[0] = r.getInt("region");
        s2[0] = r.getBigDecimal("sales");
    }
    else return;
}
catch (SQLException e) {
    System.err.println("SQLException: "+
        e.getErrorCode() + e.getMessage());
    }
}

public static void orderedEmps
       (int regionParm, ResultSet[] rs) throws
       SQLException {

    Connection conn = null;
    PreparedStatement pstmt = null;

    try {
        Class.forName
            ("sybase.asejdbc.ASEDriver");
        Connection conn =
            DriverManager.getConnection
            ("jdbc:default:connection");
    }
    catch (Exception e) {
        System.err.println(e.getMessage()
            + ":error in connection");
    }

    try {
        java.sql.PreparedStatement
            stmt = conn.prepareStatement
            ("SELECT name, region_of(state)"
```

```
                "as region, sales FROM sales_emps"
                "WHERE region_of(state) > ? AND"
                "sales IS NOT NULL"
                "ORDER BY sales DESC");
            stmt.setInt(1, regionParm);
            rs[0] = stmt.executeQuery();
            return;
        }
        catch (SQLException e) {
            System.err.println("SQLException:"
            + e.getErrorCode() + e.getMessage());
        }
        return;
    }  return;
    }
}
```

第 7 章

在数据库中调试 Java

所有 PCA/JVM 均包含对 Java 平台调试程序体系结构 (JPDA) 的内置支持。通过使用 JPDA，您可以调试在 Adaptive Server 上运行的 Java 代码。 JPDA 包括：

- 控制调试的用户界面，即调试程序
- 运行要调试的类的 JVM 以及提供 JVM 访问的调试代理
- 调试代理和调试程序之间的通信通道

JPDA 允许用户从命令行中调试 Java 类 （在调试程序应用程序中启动 JVM）或远程调试 Java 类 （将调试程序连接到运行的 JVM 上的调试代理）。由于用户无法访问服务器中的 JVM 命令行，因此，Adaptive Server 数据库中的 Java 的所有调试是远程完成的。

支持的 Java 调试程序

每个 JDK 均在其开发工具包中提供了一个基本命令行调试程序 “jdb” 实现。也可以使用集成开发环境 (IDE) 进行 Java 开发和调试，例如，Sun Java Studio、 IBM WebSphere Studio、 JBuilder 和 Eclipse。此外，还可以使用独立 JPDA 调试程序，如 JSwat。

如果使用 IDE 或独立调试程序工具，请查阅供应商提供的文档以了解具体的 JDK 要求。

注释 jdb 调试程序未包含在 JRE 分发中。要使用 jdb，您必须安装可访问 jdb 调试程序的 JDK。

设置 Java 调试

无论使用的是 IDE、独立调试程序还是 jdb 调试程序，您必须：

1 配置服务器以支持调试

2 将远程调试程序连接到 JVM 调试代理

配置服务器以支持调试

使用用户提供的端口号或缺省端口号启动 JVM 调试代理。将 sp_jreconfig 与以下配置参数结合使用以启用调试，选择端口号并指定是否立即挂起 JVM：

- pca_jvm_java_dbg_agent_port — 启用或禁用调试，并确定 JVM 中的调试代理所监听的端口号。如果启用该参数，在 JVM 启动时将以某种方式运行调试代理以允许连接远程调试程序。缺省情况下，调试代理监听端口 8000。要启用调试代理并允许使用缺省端口进行调试，请输入：

```
sp_jreconfig "enable", "pca_jvm_java_dbg_agent_port"
```

要使用不同的端口，请在启动 JVM 之前更改端口号。在启动 JVM 并运行调试代理后，调试代理将监听该端口，直至 JVM 关闭。要启用调试并更改调试代理所监听的端口，请输入：

```
sp_jreconfig "update", "pca_jvm_java_dbg_agent_port", new_port_number
```

- pca_jvm_java_dbg_agent_suspend — 控制在启动时运行调试代理时是否挂起 JVM。缺省情况下，将禁用 pca_jvm_java_dbg_agent_suspend。

  在启用 pca_jvm_java_dbg_agent_suspend 后，在连接调试程序并重新启动 JVM 之前，将无法执行任何 Java 方法。通过挂起 JVM，您可以在装载任何类之前检查 JVM 早期初始化。通常，调试用户类时不需要挂起 JVM。

  要启用 pca_jvm_java_dbg_agent_suspend，请输入：

```
sp_jreconfig "enable", "pca_jvm_java_dbg_agent_suspend"
```

注释 应小心使用 pca_jvm_java_dbg_agent_suspend。启用 pca_jvm_java_dbg_agent_suspend 将导致 JVM 挂起；所有 Adaptive Server Java 任务将一直等到连接调试程序并通过它通知 JVM 继续为止。 Sybase 建议您启动 JVM 并运行简单的 Java 命令，以便连接调试程序而不是启用 pca_jvm_java_dbg_agent_suspend。这样，就可以引导 JVM，并在执行要调试的类之前连接调试程序。

设置在 JVM 中启用调试代理的配置值后，下次启动 JVM 时，便可以使用调试代理了。要禁用调试代理，请禁用配置参数并重新启动 JVM（在运行代理的情况下启动 JVM 后，将无法关闭该代理）。

注释　缺省情况下，不要运行调试代理。在调试代理运行时，任何具有主机网络访问权限的调试应用程序都可能会连接到 JVM 并访问对象内部数据。

将远程调试程序连接到 JVM 调试代理

在远程调试程序连接到在 Adaptive Server 中运行的调试代理时，将开始运行调试会话。除了使用 sp_jreconfig 提供的连接信息以外，您还必须输入要调试的类的源文件位置。

如果使用 IDE 或独立调试程序，请查阅供应商文档以了解有关如何将远程调试程序连接到调试代理的说明。

以下示例假设您使用的是 jdb 命令行调试程序。您通过端口 8000 连接到计算机“myhost”上的调试代理，并指定主目录下的 JAR 存档 *mysource.jar* 中的 Java 源文件。

```
jdb -attach myhost:8000 -source .:${HOME}/mysource.jar
```

对于其它调试程序工具，语法会有所不同，但必须始终提供连接信息和源文件位置。

第 8 章

使用 Java 访问文件和网络

本章举例说明了如何使用 Java 访问文件和网络。

Adaptive Server 使用 java.io、 java.net 和 java.nio 包支持文件和网络 I/O 功能。

注释 如果文件和网络 I/O 将较大文本文档流式传入或传出服务器，则可能需要增加 JVM 的可用内存量。如果要处理较大文档，则可能需要增大 pci memory size 配置参数值以满足较大内存要求。请参见第 14 页的 “PCI 内存池”。

使用 *java.io* 访问文件

PCA/JVM 通过 java.io 和 java.nio 包支持直接文件 I/O。通过使用这些包，用户可以从文件系统中读取文件以及将文件写入到文件系统中。

必须明确区分操作系统使用的用户标识和 Adaptive Server 使用的用户标识。

用户标识和权限

在 Adaptive Server 启动时，将使用启动服务器进程的系统用户 ID 执行该进程。例如，如果 Adaptive Server 是由系统用户 ID “sybase”启动的，请输入：

```
% ps -Usybase -o user,pid,command

USER       PID   CMD
sybase   20405   /sybase/ASE-15-0/bin/dataserver ...
```

因此， Adaptive Server 进程和操作系统之间的所有交互与启动 Adaptive Server 的系统用户 ID 关联。

不过，在服务器中，情况会有所不同。当每个用户登录到服务器时，该用户使用 Adaptive Server 服务器中定义的用户 ID 执行上述操作。此用户 ID 与主机上定义的用户 ID 不同，尽管您可能希望用户 ID 在 Adaptive Server 和操作系统上表示同一个人。

在数据库中，用户可以根据为其分配的角色执行不同的操作。登录到 Adaptive Server 的用户可能在主机上没有用户帐户。因此，启动服务器的用户帐户可以作为任意数量的数据库用户的代理。例如，假设 Adaptive Server 用户要读取两个文件 （用户的文件权限被严格限定为只读）。

```
-r-----------1 sybase sybuser    1263 Aug 19 18:54 myfile1.dat
-r-----------1 jdoe   sybuser     952 Aug  7  9:02 myfile2.dat
```

如果用户登录到 Adaptive Server 以运行一个尝试读取这些文件的 Java 方法， Java 文件 I/O 最终将采用主机接口管理的函数：

```
isql -Usa -P...
isql -Ujdoe -P...
isql -Ujanedoe -P...
```

对于每个用户来说，基本 read() 运行时函数的行为是相同的。每个用户可以读取 *myfile1.dat*，该文件归系统用户 ID “sybase”所有，因为将向操作系统指明服务器归该用户所有。不过，任何用户都无法读取 *myfile2.dat*，即使它似乎归某个数据库用户所有，因为所有数据库用户标识压缩为单个操作系统标识 “sybase”，它与进程所有者相关联。因此，将拒绝访问文件。

指定文件 I/O 目录：UNIX 平台

可以使用传统的 UNIX 表示法指定可选的额外路径权限限制。例如，“u+rw”为用户提供读写访问权限，为组提供只读访问权限并拒绝所有其它帐户进行访问。这些限制不影响操作系统权限；对于具有只读操作系统权限的目录，在配置语句中授予读写访问权限的用户并不会获得该目录的写入访问权限。

对于所有写入操作 （包括文件创建），如果未提供掩码，则目录使用缺省掩码 0666。该掩码不能用于只读操作。

如果提供了掩码，则假定缺省掩码全部为零。这可确保指定为 (u+rw) 的掩码生成掩码 0600。

掩码语法

work_dir （可信目录）权限掩码：

- 必须放在紧靠路径后面的位置，它们之间没有空格。
- 可以使用前导字符（u、g、o 和 a）以及后面的 +、–、=、r、w 和 x 定义用户、组、其它和所有掩码。

例如：

- (u=rw,go=r) 等于 0644
- (ugo+r,u+w) 等于 0644
- (ugo+r,u+wx) 等于 0755
- (ugo=rwx,go–wx) 等于 0755

可以使用多种方法定义掩码，但始终从左到右进行求值。例如，假设最初将掩码定义为 0777 (ugo=rwx)。如果后来删除了组和其它帐户的写入 (w) 和执行 (x) 权限，则等效的八进制值变为 0744，掩码变为 (ugo=rwx,go–wx)。

如果未指定掩码（掩码部分是可选的），则目录使用缺省写入掩码 0666。

有效语法值包括：

u ... 用户 （或 owner）。
g ... 组。
o ... 其它帐户 （或 world）。
a ... 所有 （设置 u、 g 和 o）。例如：(a+rw) 为 u、 g 和 o 启用读取和写入。
+ ... 启用位。
– ... 禁用位。

= ... 替换位。例如：(u=rw) 替换用户。
r ... 读取位。
w ... 写入位。
x ... 执行位。

示例

- 要在 pca_jvm_work_dir 数组中添加新的工作目录路径，请输入：

  ```
  sp_jreconfig "add", "work_dir", "/some/path(u+rw)
  ```

  或

  ```
  sp_jreconfig "add", "work_dir", "/some/path(u=rw)
  ```

- 要从 pca_jvm_work_dir 数组中删除现有的工作目录路径，请输入：

  ```
  sp_jreconfig "delete", "work_dir", "/some/path"
  ```

  在删除或更新 work_dir 数组元素或路径条目时，仅路径部分在提供的字符串中是必需的。

- 要在 pca_jvm_work_dir 数组中修改现有的工作目录路径，请输入：

  ```
  sp_jreconfig "update", "work_dir", "/old", "/new"
  ```

- 要更改路径并更新权限，请输入：

```
sp_jreconfig "update", "work_dir", "/some/path(u+rw)", "/some/path(u+w)"
```

- 要在 pca_jvm_work_dir 数组中禁用现有的工作目录路径，请输入：

  ```
  sp_jreconfig "disable", "work_dir", "/some/path"
  ```

  最后一个参数是用于标识单个 work_dir 数组元素的完整或部分字符串值，即使数组中只有一个元素，也必须提供该参数。

- 要在 pca_jvm_work_dir 数组中清除整个工作目录路径集，请输入：

  ```
  sp_jreconfig "array_clear", "work_dir"
  ```

- 要启用整个数组，请输入：

  ```
  sp_jreconfig "array_enable", "work_dir"
  ```

- 要禁用整个数组，请输入：

  ```
  sp_jreconfig "array_disable", "work_dir"
  ```

指定文件 I/O 目录：Windows 平台

可以使用以下表示法指定可选的额外路径权限限制。

掩码语法

在 Windows 环境中，可以将以下语法添加到工作目录定义末尾以定义权限掩码：

- /RW — 定义读取/写入权限
- /RO — 定义只读权限
- /NA — 不定义任何访问权限

示例

- 要将 *D:\my_work_dir* 定义为具有完全访问权限的可信目录，请输入：

```
sp_jreconfig "add", "work_dir", "C:\my_work_dir/RW"
```

- 要将 *D:\my_read_only* 定义为具有只读访问权限的可信目录，请输入：

```
sp_jreconfig "add", "work_dir","D:\my_read_only_dir/RO"
```

- 要将 *E:\general* 定义为具有完全访问权限的可信目录，但禁止访问名为 *TOP_SECRET* 的 *E:\general* 子目录，请输入：

```
sp_jreconfig "add", "work_dir","E:\general/RW;E:\general\TOP_SECRET/NA"
```

请使用分号分隔各个目录条目。

文件 I/O 更改

JVM 中的文件 I/O 主要是通过文件打开操作控制的。在成功打开文件后，通常允许对文件执行其它 I/O 操作。为安全起见，发出的所有文件打开请求必须使用物理文件的绝对路径；不支持软链接。在尝试执行任何文件 I/O 操作之前，相对路径将转换为绝对路径。因此，无法将 *$SYBASE* 目录设置为软链接。这样做可防止 JVM 进行初始化，因为它无法打开 *$SYBASE/shared* 中的文件。

如果文件打开操作不符合特定的规则集，则无法打开文件。文件打开规则基于：

- 文件是否已存在
- 文件是以只读还是读写访问权限打开的
- 要打开的文件的位置

打开现有文件的规则

本节介绍了在 UNIX 和 Windows 平台上打开文件的规则和检查。

注释 如果任何检查失败，则会拒绝文件打开请求并向调用者报告错误。

UNIX 平台

如果与服务器关联的用户 ID 具有访问文件的权限，并且该文件位于 *$SYBASE/shared* 目录中，则可以打开该文件进行只读访问。任何其它 $SYBASE 目录不允许进行读取访问。

注释 任何 $SYBASE 目录绝不允许进行写入访问 （包括文件创建）。

对于打开以进行写入访问的文件，将在批准文件打开请求之前进行额外的检查。 Adaptive Server 检查：

- 发出文件打开请求的用户是文件所有者。
- 硬链接数不超过一个。如果超过一个，请求将会失败。
- 要打开的文件位于有效目录位置。如果文件位于 $SYBASE 目录中，或者没有位于配置的某个工作目录中，请求将会失败。
- 为工作目录配置了允许使用写入访问权限打开文件的访问掩码。缺省掩码为 0666。除非您希望使用非缺省掩码，否则不需要该掩码。

Windows 平台

如果与服务器关联的用户 ID 具有访问文件的权限，则在以下情况下授予访问权限：

- 该文件在 %SYBASE% 目录结构中已存在，允许进行只读访问，并且打开以进行写入的请求收到 ERROR_ACCESS_DENIED 错误，或者
- 该文件存在或正在 Windows %TEMP% 目录中创建该文件，并且允许进行读写访问，或者
- 该文件存在或正在配置的工作目录中创建 （可信目录）该文件，允许进行的访问是为工作目录定义的访问，或者
- 该文件存在或正在可信目录下的任何子目录中创建该文件，允许进行的访问是为父目录定义的访问。
- 如果一个可信目录嵌套在另一个可信目录中，系统将检查目标文件路径中的每个可信父目录的访问权限，并应用具有最大限制的访问权限。因此，可以允许对可信目录树进行读写访问，但指定对它下面的指定目录进行只读访问或不允许进行访问。此行为与 Windows 将 ACL 应用于文件的行为类似。

使用文件打开操作创建文件的规则

打开不存在的文件的请求实质上就是文件创建操作，必须使用与已存在的文件不同的方式处理该请求。适用于打开以进行写入访问的现有文件的相同位置限制也适用于新创建的文件：如果新创建的文件位于 *$SYBASE* 目录结构中，或者不包含在配置的工作目录中，请求将会失败。此外，目录的访问掩码必须允许与服务器进程关联的用户 ID 在目标目录中写入内容。

注释　始终允许在 */tmp* 目录中进行写入访问 （包括文件创建）。

在 UNIX 平台上 — 通过打开请求创建的文件必须指定写入访问权限，并且始终使用文件打开标志 (O-CREAT | O-EXCL | O-RDWR) 和访问掩码 (0600) 打开文件。为安全起见，应始终使用这些文件打开标志和该访问掩码，而不管文件打开请求指定了什么标志和访问掩码。无法使用指定打开文件以进行只读访问的文件打开标志创建文件。要限制文件大小或设置磁盘使用限额，必须在操作系统级别执行此操作。

最终文件检查

在文件打开请求通过了所有文件检查并允许打开文件后，最终检查可确保打开的文件与最初请求的文件相一致。这可防止尝试打开本来不允许打开的文件，以使其逃避检查的企图难以得逞。如果文件打开请求失败，则会在审计跟踪中添加注释，并且调用方法引发 java.lang.IOException。特定于方法的 IOException 处理决定了该异常对用户可见，还是由 Java 代码中的替代机制处理。

使用 *java.net* 访问文件

借助于 Adaptive Server 对 java.net 和 java.nio 的支持，您可以在服务器中创建客户端 Java 网络应用程序。可以创建一个连接到任何服务器的网络 Java 客户端应用程序，这实际上是将 Adaptive Server 作为外部服务器的客户端。

可以使用 java.net 和 java.nio 执行以下操作：

- 从 Internet 上的任意 URL 下载文档。
- 从服务器中发送电子邮件。
- 连接到外部服务器以保存文档和执行文件功能，如保存或编辑文档。
- 使用 XML 访问文档。

注释 应小心使用 java.net：

- 大多数与 java.net 关联的对象不能序列化；无法将它们插入到表中。
- 大多数与 I/O 相关的方法使用缓冲 I/O，不会自动刷新这些方法。必须显式地刷新这些方法，如 PrintWriter。

示例

本节提供了使用 Socket 类和 URL 类的示例。您可以：

- 通过 URL 类使用 XML 查询语言 (XQL) 访问外部文档。
- 使用 MailTo 类以邮件形式发送文档。

使用 Socket 类

与 URL 类相比， Java Socket 类允许进行更复杂的网络传输。通过使用 Socket 类，您可以连接到任何网络主机上的指定端口，并使用 InputStream 和 OutputStream 类读写数据。

使用 URL 类

可以使用 URL 类执行以下操作：

- 发送电子邮件。
- 从 Web 服务器中下载 HTTP 文档。 HTTP 文档可以是静态文件，也可以是 Web 服务器动态创建的文件。
- 使用 XQL 访问外部文档。
- 使用 mailto:URL 类以邮件形式发送文档。

例如，可以使用 URL 类以邮件形式发送文档。客户端必须连接到邮件服务器，以使系统属性引用的计算机（以下示例中为 salsa.sybase.com）可以运行邮件服务器 （如 sendmail）。

对于以下示例，步骤如下：

1　创建一个 URL 对象。

2　设置一个 URLConnection 对象。

3　通过 URL 对象创建一个 OutputStream 对象。

4　编写邮件。例如：

```
import java.io.*;
import java.net.*;
public class MailTo {
    public static void sendIt()
        throws Exception{
        System.getProperty("mail.host", "salsa.sybase.com");
        URL url = new URL("mailto:name@sybase.com");
        URLConnection conn = url.openConnection();
        PrintStream out = new PrintStream(conn.getOutputStream(),true);
        out.println ("From janedoes@sybase.com");
        out.println ("Subject: Works Great!");
        out.println ("Thanks for the example - it works great!");
        out.close();
        System.out.println("Message Sent");
    }
}
```

5 安装 mailto:URL，以便从数据库中发送电子邮件：

```
select MailTo.sendIt()
```

也可以使用 URL 类从 HTTP URL 中下载文档。在启动时，客户端将连接到 Web 服务器。步骤如下：

1 创建一个 URL 对象。

2 通过 URL 对象创建一个 InputStream 对象。

3 使用 InputStream 对象中的 read 读取文档。

以下代码示例将整个文档读取到 Adaptive Server 内存中，然后针对内存中的文档创建一个新 InputStream。

```
import java.io.*;
import java.net.*;
public class URLprovess {
          public static InputStream readURL()
              throws Exception {
              URL u = newURL("http://www.xxxx.con");
              InputStream in = u.openStream");
              //This is the same as creating URLConnection, then calling
              //getInputStream(). In Adaptive Server, you must read the entire
              //document into memory, and then create an InputStream on the
              //in-memory copy.
              int n = 0;
              int off = 0;
              byte b() = new byte(50000);
              for)off = 0; (off<b.length512) &&
                  ((n = in.read(b.off,512)! = 1);off+=n) {}
              System.out.prinln("Number of bytes read :" + off);
              in.close();
              ByteArrayInputStream test = new ByteArrayInputStream(b,-,off);
              return (InputStream) test;
          }
}
```

在创建新的 InputStream 类后，可以安装该类并使用它将文本文件读取到数据库中。以下示例在 mytable 表中插入数据。

```
create table mytable (cl text)
go
insert into mytable values (URLprocess.readURL())
go
Number of bytes read :40867
select datalength(cl) from mytable
go

---------------------
40867
```

第 9 章

其它主题

本章提供有关几个参考主题的信息。

服务器中的 Java 类的 JDK 要求

在服务器中安装和使用的 Java 类必须是通过 PCA/JVM 插入到 Adaptive Server 中的 JVM 版本或更低版本。 PCA/JVM 支持 Java 6 和更高版本。

赋值

本节定义在数据类型为 Java-SQL 类的 SQL 数据项之间赋值的规则。

每次赋值是将*源实例*传送给*目标数据项*：

- 对于指定具有 Java-SQL 列的表的 insert 语句，将 Java-SQL 列当作目标数据项来引用，将插入值当作源实例来引用。
- 对于更新 Java-SQL 列的 update 语句，将 Java-SQL 列当作目标数据项来引用，将更新值当作源实例来引用。
- 对于给变量或参数赋值的 select 或 fetch 语句，将变量或参数当作目标数据项来引用，并将检索到的值当作源实例来引用。

注释 如果源是变量或参数，则它是对 Java VM 中的对象的引用。如果源是一个包含序列的列引用，则列引用规则（请参见第 151 页的“Java-SQL 列引用”）会生成对 Java VM 中的对象的引用。因此，源是对 Java VM 中的对象的引用。

编译期赋值规则

1 将 SC 和 TC 定义为源和目标的编译期类名。将 SC_T 和 TC_T 定义为与目标关联的数据库中名为 SC 和 DT 的类。类似地，将 SC_S 和 TC_S 定义为与源关联的数据库中名为 SC 和 DT 的类。

2 SC_T 必须与 TC_T 或 TC_T 的子类相同。

运行时赋值规则

假定 DT_SC 与 DT_TC 或者它的子类相同。

- 将 RSC 定义为源值的运行时类名。将 RSC_S 定义为与源关联的数据库中名为 RSC 的类。将 RSC_T 定义为安装在与目标关联的数据库中的 RSC_T 类的名称。如果没有 RSC_T 类，则会引发异常。如果 RSC_T 既不与 TC_T 相同，也不是 TC_T 的子类，则会引发异常。
- 如果与源和目标关联的数据库不是同一个数据库，则源对象由其当前类 RSC_S 进行序列化，而该序列将在与目标关联的数据库中由它的关联类 RSC_T 取消序列化。

- 如果目标是 SQL 变量或参数，则通过引用将源复制到目标。
- 如果目标是 Java-SQL 列，则会对源进行序列化，并将该序列深度复制到目标。

允许的转换

可以使用 convert 按以下方式更改表达式数据类型：

- 当 Java 数据类型是 Java 对象类型时，将 Java 类型转换为 SQL 数据类型，如第 146 页的 “Java 和 SQL 之间的数据类型映射”所示。convert 函数的操作是 Java-SQL 映射隐含的映射。
- 将 SQL 数据类型转换为 Java 类型，如第 146 页的 “Java 和 SQL 之间的数据类型映射”所示。convert 函数的操作是 SQL-Java 映射隐含的映射。
- 如果表达式 （源类）的编译期数据类型是目标类的子类或父类，则安装在 SQL 系统中的 Java-SQL 类将转换为安装在 SQL 系统中的其它 Java-SQL 类。否则，将引发一个异常。

转换结果与当前数据库相关联。

有关使用 convert 函数处理 Java 子类型的讨论，请参见 “使用 SQL convert 函数处理 Java 子类型”。

将 Java-SQL 对象传送到客户端

当数据类型为 Java-SQL 对象类型的值从 Adaptive Server 传送到客户端时，对象的数据转换取决于客户端的类型：

- 如果客户端是 isql 客户端，则调用对象的 toString( ) 或类似方法并将结果截断为 varchar （将传送到客户端）。

注释　传送到客户端的字节数取决于全局变量 @@stringsize 的值。缺省值为 50 个字节。有关详细信息，请参见第 39 页的 “表示 Java 实例”。

- 如果客户端是使用 jConnect 4.0 或更高版本的 Java 客户端，则服务器会将对象序列传输至客户端。此序列被 jConnect 无缝地非序列化，以生成对象的副本。
- 如果客户端是 b 客户端：
  - 如果对象是声明为 in row 的列，包含在列中的被序列化的值将作为 varbinary 值传送到客户端 （其长度由列的大小确定）。
  - 否则，对象的序列化值 （对象的 writeObject 方法的结果）将作为图像值传送到客户端。

改善性能的建议

本节提供了在 Adaptive Server 中使用 Java 时改善性能的准则。

最大限度降低从 SQL 到 JVM 的调用数

现成的 JVM 以及 PCA/JVM 得益于大幅提高和优化的 JVM 功能，其速度明显高于 Adaptive Server 15.0.2 和更低版本中的内部 JVM。不过，如果将 SQL 调用传播到 Java，则可能仍会产生瓶颈，该问题在 PCA/JVM 中甚至更加突出。

要利用 PCA/JVM 的速度优势，应最大限度降低从 SQL 到 JVM 的调用数。

请考虑简单的 Address 类：

```
public class Address implements java.io.Serializable {
    private int state;
    private String street;
    private String zip;

    // ...

    public Address()
    {
        // ...
    }

    public Address(String street, String zip, int state)
    {
        this();
        this.setStreet(street);
```

```
        this.setZip(zip);
        this.setState(state);
    }

    // ,,,

    public void setStreet(String street)
    (
        // ..
    }

    public void setZip(String zip)
    {
        // ...
    }
}
```

由于与到 JVM 的调用相关的开销，使用来自 SQL 的三参数构造方法比使用零参数构造方法后跟数据成员的 set 方法快得多。因此，以下语句：

```
1> declare @a Address
2> select @a=new Address("123 Elm Street", "12345", 10)
```

比下面的语句更有效：

```
1> declare @a Adress
2> select @a = new Address()
3> select @a >> setStreet("123 Elm Street")
4> select @a >> setZip("12345")
5> select @a >> setState(10)
```

通过尽可能在 Java 中进行处理，而无需反复经由 SQL-Java 接口，可以降低开销并更充分地利用改善的 JVM 功能。

应小心使用 *java.lang.Thread* 类

PCA/JVM 支持 java.lang.Thread 类，该类可用于创建在 Adaptive Server 中使用多线程方法的类。在 Java 方法中创建的线程与 Adaptive Server 争用 CPU 和其它资源。如果线程数量很大，或者线程是资源密集型线程，则可能会影响整体服务器性能。

确定是否在 PCA/JVM 中运行

通常，在 PCA/JVM 或独立 JVM 中运行类的差别并不大。可以使用布尔逻辑检验类是不是通过 Sybase ContextClassLoader 装载的。例如：

```
boolean running_in_ase = false;

running_in_ase =
this.getClass().getClassLoader().getName().equals
("sybase.aseutils.ContextClassLoader");

if (running_in_ase)
{
    //in ASE
    ...
}
else
{
    //in a standalone JVM
    ...
}
```

避免在多引擎环境中使用 SQL 循环

在多引擎环境中，某些 Java/SQL 命令可能会对性能造成不利影响。在 SQL 循环中多次执行同一 Java 方法时，通常会发生这种情况。为避免出现这种问题，在编写 Java/SQL 命令时，应使方法和循环在 VM 上下文中执行：

1 使用 Java 编写循环。

2 从 Java 编码的循环中调用方法。

控制在 PCA/JVM 中访问本机方法

Java 语言允许按本机方法通过 Java 本机接口 (JNI) 使用以非 Java 语言实现的功能。使用本机方法的类必须使用 load(String *filename*) 或 loadLibrary(String *libname*) 方法显式地装载本机库，如 java.lang.System 和 java.langRuntime 类中所述。由于这些库不是作为控制的对象存储在数据库中，一些用户可能会认为它们不太安全。

为防止意外访问本机库，PCA/JVM 引入了系统属性 sybase.allow.native.lib 以控制本机库装载。

可以在命令行中设置很多 Java 属性，也可以从应用程序中通过 java.lang.System setProperty(String *key*, String *value*) 方法设置这些属性。不过，安全管理器禁止这样做以防止用户覆盖系统策略。缺省情况下，用户无法装载本机库。如果尝试装载本机库或更改现有属性设置，则会引发 SecurityException 并且装载尝试失败。

例如，如果尝试装载 java.net.ServerSocket 类而未设置 sybase.allow.native.lib 属性，初始化程序将会失败，因为它要求装载 Socket 库。实际 Java 堆栈会有所不同。不过，该堆栈或客户端消息将显示：

```
java.lang.SecurityException: Cannot load native
libraries from within a user Task!
```

这表示无法装载所需的本机库。

要允许装载本机库，请在启动 JVM 之前在 sybpcidb 数据库中设置以下属性：

```
1>  sp_jreconfig "add","pca_jvm_java_option",
"-Dsybase.allow.native.lib=true"

2>  go
```

在将 sybase.allow.native.lib 设置为 true 后，在 JVM 启动时将在命令行中向 JVM 传递一个额外属性。在运行 JVM 时，无法更改该属性。如果不再需要装载库，请使用 sp_jreconfig 删除或禁用 pca_jvm_java_option。

不支持的 Java API 包、类和方法

Adaptive Server 支持 Java API 中的许多（但不是所有）类和方法。另外，Adaptive Server 可能会施加安全性限制及实施限制。例如， Adaptive Server 并不支持 java.lang.Thread 的所有线程操作工具。

警告！ 应小心使用生成子线程的方法。在 Java 方法中启动的 java.lang.Thread 对象是在运行时调度的，而不是由 Adaptive Server 调度程序调度的。如果这些线程是处理器密集型线程，或者生成了大量线程，服务器性能可能会由于用户线程争用处理器时间而下降。

由于 PCA/JVM 使用标准 Java 插件，因此，您可以使用完整的类分发。通常，支持使用方法，除非使用方法妨碍服务器运行或其它 Java 任务。

Adaptive Server 中的 Java 不支持通过 Java 本机接口 (JNI) 调用的本机方法。

本节内容包括：

- 不支持的 Java 方法
- 不支持的 java.sql 方法

受限制的 Java 包、类和方法

- 由于 JVM 在无人参与模式下运行，因此，将禁用需要用户输入或输出的 Java 方法
- 不允许使用可能妨碍服务器运行或其它 JVM 任务的操作
- 不允许使用以下 java.lang.Thread 方法：
  - interrupt()
  - setPriority ()
  - setName()
  - enumerate()
  - setDaemon()
  - checkAccess()
  - getContextClassLoader()
  - setDefaultExceptionHandler()

    - setContextClassLoader()
    - getStackTrace()
    - getAllStarkTraces()
    - setDefaultUncaughtExceptionHandler()
    - stop()
    - destroy()
    - suspend()
    - resume()
    - 允许使用不建议使用的方法，但可能不太安全
        - countStackFrames()
- 不允许使用以下 java.lang.ThreadGroup 方法：
    - getParent()
    - setDaemon()
    - setMaxPriority()
    - checkAccess()
    - enumerate()
    - interrupt()
    - stop()
    - destroy()
    - suspend()
    - resume()
    - 允许使用不建议使用的方法，但可能不太安全
        - allowThreadSuspension()
- 安全问题：
    - 不能覆盖现有的安全管理器或实例化其它类装载程序。
    - 不允许使用 java.lang.System 和 java.lang.Runtime 中的 exit() 方法。

不支持的 *java.sql* 方法和接口

对于 Java 6 类分发， java.sql 包符合 JDBC 4.x 规范。不过，基本 Sybase 实现处于 JDBC 2.0 级别。不支持 JDBC 2.0 规范以后增添的任何 JDBC 方法。此外，也不支持在 JDBC 2.0 中指定的以下方法。

- Connection.commit( )
- Connection.getMetaData( )
- Connection.nativeSQL( )
- Connection.rollback( )
- Connection.setAutoCommit( )
- Connection.setCatalog( )
- Connection.setReadOnly( )
- Connection.setTransactionIsolation( )
- DatabaseMetaData.* — 支持 DatabaseMetaData，但以下方法除外：
  - deletesAreDetected()
  - getUDTs()
  - insertsAreDetected()
  - updatesAreDetected()
  - othersDeletesAreVisible()
  - othersInsertsAreVisible()
  - othersUpdatesAreVisible()
  - ownDeletesAreVisible()
  - ownInsertsAreVisible()
  - ownUpdatesAreVisible()
- PreparedStatement.setAsciiStream( )
- PreparedStatement.setUnicodeStream( )
- PreparedStatement.setBinaryStream( )
- ResultSetMetaData.getCatalogName( )
- ResultSetMetaData.getSchemaName( )
- ResultSetMetaData.getTableName( )

- ResultSetMetaData.isCaseSensitive( )
- ResultSetMetaData.isReadOnly( )
- ResultSetMetaData.isSearchable( )
- ResultSetMetaData.isWritable( )
- Statement.getMaxFieldSize( )
- Statement.setMaxFieldSize( )
- Statement.setCursorName( )
- Statement.setEscapeProcessing( )
- Statement.getQueryTimeout( )
- Statement.setQueryTimeoutt()

从 Java 中调用 SQL

Adaptive Server 提供一个本机 JDBC 驱动程序 java.sql，它执行 JDBC 1.1 和 1.2 规范，并与版本 2.0 兼容。 java.sql 允许在 Adaptive Server 中执行 Java 方法，以执行 SQL 操作。

特殊注意事项

java.sql.DriverManager.getConnection( ) 接受以下 URL：

- 空
- “”（空字符串）
- jdbc:default:connection

从 Java 中调用 SQL 时，存在一些限制：

- 执行更新操作（update、insert 或 delete）的 SQL 查询无法使用 java.sql 功能调用其它也执行更新操作的 SQL 操作。
- SQL 使用 java.sql 功能引发的触发器无法生成结果集。
- java.sql 不能用于执行扩展存储过程或远程存储过程。

Java 方法中的 Transact-SQL 命令

可以在从 SQL 系统内调用的 Java 方法中使用某些 Transact-SQL 命令。表 9-1 列出了 Transact-SQL 命令，并指明是否可以在 Java 方法中使用它们。在 Sybase 《参考手册：命令》中，可以查找到大部分命令的详细信息。

表 9-1：Transact-SQL 命令的支持状态

命令	状态
alter database	不支持。
alter role	不支持。
alter table	支持。
begin ... end	支持。
begin transaction	不支持。
break	支持。
case	支持。
checkpoint	不支持。
commit	不支持。
compute	不支持。
connect - disconnect	不支持。
continue	支持。
create database	不支持。
create default	不支持。
create existing table	不支持。
create function	支持。
create index	不支持。
create procedure	不支持。
create role	不支持。
create rule	不支持。
create schema	不支持。
create table	支持。
create trigger	不支持。
create view	不支持。
cursor	不支持。 仅支持 “服务器游标”，即在存储过程内声明和使用的游标。
dbcc	不支持。

命令	状态
declare	支持。
disk init	不支持。
disk mirror	不支持。
disk refit	不支持。
disk reinit	不支持。
disk remirror	不支持。
disk unmirror	不支持。
drop database	不支持。
drop default	不支持。
drop function	支持。
drop index	不支持。
drop procedure	不支持。
drop role	不支持。
drop rule	不支持。
drop table	支持。
drop trigger	不支持。
drop view	不支持。
dump database	不支持。
dump transaction	不支持。
execute	支持。
goto	支持。
grant	不支持。
group by 和 having 子句	支持。
if...else	支持。
insert table	支持。
kill	不支持。
load database	不支持。
load transaction	不支持。
online database	不支持。
order by Clause	支持。
prepare transaction	不支持。
print	不支持。
raiserror	支持。
readtext	不支持。
return	支持。

命令	状态
revoke	不支持。
rollback trigger	不支持。
rollback	不支持。
save transaction	不支持。
set	有关 set 选项的信息，请参见表 9-2。
setuser	不支持。
shutdown	不支持。
truncate table	支持。
union Operator	支持。
update statistics	不支持。
update	支持。
use	不支持。
waitfor	支持。
where Clause	支持。
while	支持。
writetext	不支持。

表 9-2 列出了 set 命令选项，并指明了是否可以在 Java 方法中使用它们。

表 9-2：set 命令选项的支持状态

set 命令选项	状态
ansinull	支持。
ansi_permissions	支持。
arithabort	支持。
arithignore	支持。
chained	不支持。参见注释 1。
char_convert	不支持。
cis_rpc_handling	不支持
close on endtran	不支持
cursor rows	不支持
datefirst	支持。
dateformat	支持。
fipsflagger	不支持
flushmessage	不支持
forceplan	支持。

set 命令选项	状态
identity_insert	支持。
language	不支持
lock	支持。
nocount	支持。
noexec	不支持
offsets	不支持
or_strategy	支持。
parallel_degree	支持。参见注释 2。
parseonly	不支持
prefetch	支持。
process_limit_action	支持。参见注释 2。
procid	不支持
proxy	不支持
quoted_identifier	支持。
replication	不支持
role	不支持
rowcount	支持。
scan_parallel_degree	支持。参见注释 2。
self_recursion	支持。
session_authorization	不支持
showplan	支持。
sort_resources	不支持
statistics io	不支持
statistics subquerycache	不支持
statistics time	不支持
string_rtruncation	支持。
stringsize	支持。
table count	支持。
textsize	不支持
transaction iso level	不支持。参见注释 1。
transactional_rpc	不支持

set 命令选项	状态
注释 (1) 带有 chained 或 transaction isolation level 选项的 set 命令只有在它们指定的设置已生效的情况下才允许使用。也就是说，只有在这类 set 命令没有影响时才允许使用。这样设计的目的是为了在存储过程中支持公用编码。	
注释 (2) 与并行度有关的 set 命令是允许的，但不会产生任何影响。它支持为其它环境设置并行度的存储过程的使用。	

Java 和 SQL 之间的数据类型映射

Adaptive Server 将 SQL 数据类型映射到 Java 类型 （SQL-Java 数据类型映射），并将 Java 标量类型映射到 SQL 数据类型 （Java-SQL 数据类型映射）。表 9-3 显示了 SQL-Java 数据类型映射。

表 9-3：将 SQL 数据类型映射到 Java 类型

SQL 类型	Java 类型
char	String
varchar	String
nchar	String
nvarchar	String
unichar	String
univarchar	String
unitext	String
text	String
numeric	java.math.BigDecimal
decimal	java.math.BigDecimal
money	java.math.BigDecimal
smallmoney	Java.math.BigDecimal
bit	boolean
tinyint	byte
smallint	short
integer	int
bigint	long

SQL 类型	Java 类型
unsigned smallint	int
unsigned int	long
unsigned bigint	java.math.BigInteger
bigint	java.math.BigInteger
real	float
float	double
double precision	double
binary	byte[ ]
varbinary	byte[ ]
image	java.io.InputStream
datetime	java.sql.Timestamp
smalldatetime	java.sql.Timestamp
bigdatetime	java.sql.Timestamp
bigtime	java.sql.Time
date	java.sql.Date
time	java.sql.Time

注释　unsigned bigint 到 double 的映射是一个近似值；它无法提供精确值。要得到精确值，在将 unsigned bigint 值传递到 Java 方法时，请将该值转换为 string 值。

表 9-4 显示了 Java-SQL 数据类型映射。

表 9-4：将 Java 标量类型映射到 SQL 数据类型

Java 标量类型	SQL 类型
boolean	bit
byte	tinyint
short	smallint
int	integer
long	bigint
float	real
double	double

Java-SQL 标识符

说明

Java-SQL 标识符是可以在 SQL 中引用的 Java 标识符的子集。

语法

java_sql_identifier ::= alphabetic character | underscore (_) symbol
[alphabetic character | arabic numeral | underscore(_) symbol |
dollar ($) symbol]

用法

- 如果 Java-SQL 标识符在引号中，则最长可为 255 字节。否则，它们不能多于 30 个字节。
- 标识符的第一个字符必须为字母 （大写或小写）或下划线 (_) 符号。后续的字符可以包含字母 （大写或小写）、数字、美元 ($) 符号或者下划线 (_) 符号。
- Java-SQL 标识符是区分大小写的。

分隔标识符

- 分隔标识符是双引号中的对象名称。对 Java-SQL 标识符使用分隔标识符使您可以避免 Java-SQL 标识符名称存在的一些限制。

  注释 无论 set quoted_identifier 选项是 on 还是 off，都可以在 Java-SQL 标识符中使用双引号。

- 分隔标识符允许对包、类和方法等使用 SQL 保留字。每次在语句中使用分隔标识符时，必须使其在双引号内。例如：

```
create table t1
       (c1 char(12)
       c2 p1.”select”.p2.”jar”)
```

- 双引号仅将各个 Java-SQL 标识符引起来，而不将完全限定名引起来。

另请参见

有关标识符的其它信息，请参见《参考手册：构件块》中的第 4 章 “表达式、标识符和通配符”。

Java-SQL 类和包名称

说明

要引用 Java-SQL 类或包，请使用以下语法：

语法

java_sql_class_name ::= [*java_sql_package_name.*]*java_sql_identifier*

java_sql_package_name ::=
[*java_sql_package_name.*]*java_sql_identifier*

参数

java_sql_class_name
当前数据库中的 Java-SQL 类的完全限定名。

java_sql_package_name
当前数据库中的 Java-SQL 包的完全限定名。

java_sql_identifier
请参见 Java-SQL 标识符。

用法

对于 Java-SQL 类名：

- 类名引用始终引用当前数据库中的类。
- 如果指定 Java-SQL 类名而没有引用包名，则当前数据库中只能存在一个该名称的 Java-SQL 类，它的包必须是缺省 （匿名）包。
- 如果 SQL 用户定义数据类型和 Java-SQL 类具有相同的标识符序列，Adaptive Server 将使用 SQL 用户定义数据类型名称，而忽略 Java-SQL 类名。

对于 Java-SQL 包名称：

- 如果指定 Java-SQL 子包名，则必须使用包名引用子包名：

  java_sql_package_name.java_sql_subpackage_name

- 将 Java-SQL 包名仅作为用于类名或子包名的限定符来使用，并使用 remove java 命令从数据库中删除包。

Java-SQL 列声明

说明

要在创建或更改表时声明 Java-SQL 列，请使用以下语法：

语法

java_sql_column ::= column_name java_sql_class_name

参数

java_sql_column
指定 Java-SQL 列声明的语法。

column_name
Java-SQL 列的名称。

java_sql_class_name
当前数据库中的 Java-SQL 类的名称。这是列的 “声明的类”。

用法

- 声明的类必须实现 Serializable 或 Externalizable 接口。
- Java-SQL 列始终与当前数据库相关联。
- 不能将 Java-SQL 列指定为：
  - not null
  - unique
  - 主键

另请参见

只有在创建或更改表时，才可以使用 Java-SQL 列声明。请参见 《参考手册：命令》中有关 create table 和 alter table 的信息。

Java-SQL 变量声明

说明

使用 Java-SQL 变量声明来声明用于属于 Java-SQL 类的数据类型的变量和存储过程参数。

语法

java_sql_variable ::= *@variable_name java_sql_class_name*

java_sql_parameter ::= *@parameter_name java_sql_class_name*

参数

java_sql_variable
指定 SQL 存储过程中的 Java-SQL 变量的语法。

java_sql_parameter
指定 SQL 存储过程中的 Java-SQL 参数的语法。

java_sql_class_name
当前数据库中的 Java-SQL 类的名称。

用法　*java_sql_variable* 或 *java_sql_parameter* 始终与包含存储过程的数据库相关联。

另请参见　有关变量声明的详细信息，请参见 《参考手册》。

Java-SQL 列引用

说明　要引用 Java-SQL 列，请使用以下语法：

语法

column_reference ::=
[[[*database_name*.]*owner*.]*table_name*.]*column_name*
| *database_name*..*table_name*.*column_name*

参数

column_reference
对数据类型为 Java-SQL 类的列的引用。

用法

- 如果列的值为空值，那么列引用也为空值。
- 如果列的值为 Java 序列 S，其类名为 CS，则：
  - 如果 CS 类在当前数据库中不存在，或者 CS 不是与序列关联的数据库中的类名，则会引发异常。

    注释　与该序列相关联的数据库通常是包含该列的数据库。不过，工作表和使用 “insert into #tempdb” 创建的临时表中包含的序列与最初存储该序列的数据库相关联。

  - 列引用的值为：

    CSC.readObject(S)

    此处的 CSC 是列引用。如果表达式引发一个未发生的 Java 异常，则会引发异常。

    表达式生成对 Java VM 中的对象的引用，与序列相关联的数据库与此对象相关联。

Java-SQL 成员引用

说明

引用类或类实例的字段或方法。

语法

member_reference ::= *class_member_reference* |
instance_member_reference

class_member_reference ::= *java_sql_class_name.method_name*

instance_member_reference ::= *instance_expression>>member_name*

instance_expression ::= *column_reference* | *variable_name*
| *parameter_name* | *method_call* | *member_reference*

member_name ::= *field_name* | *method_name*

参数

member_reference
描述类或对象的字段或方法的表达式。

class_member_reference
描述 Java-SQL 类的静态方法的表达式。

instance_member_reference
描述 Java-SQL 类实例的静态或动态方法或字段的表达式。

java_sql_class_name
当前数据库中的 Java-SQL 类的完全限定名。

instance_expression
数据类型为 Java-SQL 类的表达式。

member_name
类或类实例的字段或方法的名称。

用法

- 如果成员引用某个类实例的字段，而该实例具有空值，并且 Java-SQL 成员引用是 fetch、 select 或 update 语句的目标，则会引发异常。

  否则， Java-SQL 成员引用将包含空值。

- 双尖括号 (>>) 和点 (.) 限定比诸如加号 (+) 或等号 (=) 之类的其它运算符优先级高，例如：

  ```
  X>>A1>>B1 + X>>A1>>B2
  ```

  在此表达式中，加运算在成员被引用后执行。

- 成员引用指定的字段或方法的关联数据库与它的 Java-SQL 类或 Java-SQL 类实例的数据库相同。

  如果成员引用的 Java 类型是 Java 标量类型之一（如 boolean、 byte 等），可通过将 Java 类型映射到等效的 SQL 类型来获得引用的相应 SQL 数据类型。

  如果成员引用的 Java 类型是对象类型，则 SQL 数据类型是相同的 Java 对象类型或类。

Java-SQL 方法调用

说明

要调用返回单一值的 Java-SQL 方法，请使用以下语法：

语法

method_call ::= *member_reference* ([*parameters*])
　　| new *java_sql_class_name* ([*parameters*])

parameters ::= *parameter* [(, *parameter*)...]

parameter ::= *expression*

参数

method_call
静态方法、实例方法或类构造方法的调用。可以在要求使用方法的数据类型的非常量值的表达式中使用该方法调用。

member_reference
表示方法的成员引用。

参数
要传递给方法的参数列表。如果没有参数，则包括空的小括号。

用法

方法重载

- 当在相同的类或实例中有同名的方法时，将根据 Java 方法重载规则解决此问题。

方法调用的数据类型

- 方法调用的数据类型根据下列情况判断：
  - 如果方法调用指定 new，则它的数据类型是其 Java-SQL 类的数据类型。
  - 如果方法调用指定的成员引用是类型值方法，则方法调用的数据类型是此种类型。

  - 如果方法调用指定的成员引用表示无类型静态方法，则方法调用的数据类型是 SQL integer。
  - 如果方法调用指定的成员引用是类的无类型实例方法，那么方法调用的数据类型是该类的数据类型。
- 当参数是与另一个数据库相关联的 Java-SQL 实例时，要在成员引用中包括该参数，必须确保在这两个数据库中都包括与 Java-SQL 实例相关联的类名。否则，将引发一个异常。

运行时结果

- 方法调用的运行期结果如下所示：
  - 如果方法调用指定的成员引用的运行期值为空值 （即空值实例成员的引用），那么结果为空值。
  - 如果方法调用指定的成员引用是类型值方法，则结果是由方法返回的值。
  - 如果方法调用指定的成员引用是无类型静态方法，则结果是空值。
  - 如果方法调用指定的成员引用是类实例的无类型实例方法，则结果是对该实例的引用。
  - 方法调用及其结果与相同的数据库相关联。
  - Adaptive Server 并不将空值作为参数值传递给 Java 类型是标量的方法。

词汇表

本词汇表介绍本手册中所使用的 Java 和 Java-SQL 术语。有关 Adaptive Server 和 SQL 术语的说明，请参考 《Adaptive Server 词汇表》。

Java 档案 (JAR) 一种与平台无关的格式，用于将类收集到单个文件中。

Java 对象 包含在 Java VM 存储区中的 Java 类的实例。在 SQL 中引用的 Java 实例是 Java 列或 Java 对象的值。

Java 方法签名 Java 方法中每个参数的 Java 数据类型。

Java 开发工具包 (JDK) Sun Microsystems 提供的工具箱，用于在操作系统中编写和测试 Java 程序。

Java 数据库连接 (JDBC) 一种 Java-SQL API，它是用于控制 Java 应用程序开发的 Java 类库的标准部分。 JDBC 提供类似于 ODBC 的功能。

Java 数据类型 Java 类 （可以是用户定义的或来自 JavaSoft API）或 Java 原始数据类型，如 boolean、 byte、 short 和 int。

Java 文件 类型为 “java” 的文件 （例如， *myfile.java*），其中包含 Java 源代码。请参见**类文件**和 **Java 档案 (JAR)**。

Java 虚拟机 (JVM) 在服务器中处理 Java 的 Java 解释器，可以通过 SQL 实现来调用它。

Java-SQL 变量 数据类型为 Java-SQL 类的 SQL 变量。

Java-SQL 类 已经安装在 Adaptive Server 系统中的公共 Java 类，它由一组变量定义和方法组成。

类实例由类的每个字段的实例组成。类实例按照类名严格分类。

子类是声明的用来扩展 （大多数情形）另一个类的类。所扩展的类称为这个子类的直接父类。子类具有其直接和间接父类的所有变量和方法，并且可与它们交替使用。

Java-SQL 列 数据类型为 Java-SQL 类的 SQL 列。

Java-SQL 数据类型映射 Java 与 SQL 数据类型之间的转换。请参见第 146 页的 “Java 和 SQL 之间的数据类型映射”。

public Java 中定义的公共字段和方法。

SQL 过程签名	SQLJ 过程中每个参数的 SQL 数据类型。
SQL 函数签名	SQLJ 函数中每个参数的 SQL 数据类型。
SQL-Java 数据类型映射	Java 与 SQL 数据类型之间的转换。请参见第 146 页的 “Java 和 SQL 之间的数据类型映射”。
Unicode	ISO 10646 定义的 16 位字符集，它支持多种语言。
保留的 JAR	请参见**关联的 JAR**。
变量	在 Java 中，变量局限于类、类实例或方法。声明为 static 的变量是类的局部变量。在类中声明的其它变量是该类实例的局部变量。这些变量称为类的字段。在方法中声明的变量是方法的局部变量。
方法	包含在 Java 类中的指令集合，用于执行某一任务。方法可声明为 static（静态），在这种情况下该方法称为类方法。否则，它是实例方法。通过用类名或类实例名来限定方法名称，可以引用类方法。通过用类实例名来限定方法名称，可以引用实例方法。实例方法的方法本身可以引用该实例的局部变量。
父类	在层次结构中处于一个或多个类之上的类。它向下级类传递特性和行为。它可能无法与其子类交替使用。请参见**子类**、**缩小转换**和**扩大转换**。
赋值	一般术语，指由 select、 fetch、 insert 和 update Transact-SQL 命令指定的数据传送。赋值是将源值设置到目标数据项中。
关联的 JAR	如果在安装 class/JAR 时使用了 installjava 和 -jar 选项，那么 JAR 将保留在数据库中，而类将在数据库中与关联的 JAR 链接。请参见**保留的 JAR**。
过程	SQL 存储过程或具有*无类型*结果类型的 Java 方法。
接口	方法声明的命名集合。如果一个类在一个接口中定义了声明的所有方法，就可以执行这个接口。
静态方法	不需引用对象的方法调用。静态方法影响整个类，而不仅仅是该类的一个实例。也称为类方法。
可插入组件接口 (PCI)	Adaptive Server Java 框架；借助于 PCA/JVM，它允许将通过商业途径获得的 JVM 与 Adaptive Server 一起使用。
可插入组件接口 (PCI) 桥	一个 Adaptive Server 组件，它是 PCI 的一部分，可实现 JVM 插件和 Adaptive Server 之间的交互。
可插入组件适配器/JVM	一个 Sybase 组件，用于管理 Adaptive Server 和 JVM 之间的服务请求。

可见

如果安装在 SQL 系统中的 Java 类被声明为 public，则该 Java 类在 SQL 中可见；如果 Java 实例的字段或方法为 public 并且可映射，则该方法或字段在 SQL 中可见。在 SQL 中可以引用可见的类、字段和方法，但不能引用其它类、字段和方法，包括：声明为 private、protected 或 friendly 的类；声明为 private、protected 或 friendly 的字段或方法；不可映射的字段或方法。

可映射的

当 Java 数据类型属于下列情况时，就是可映射的：

- 在第 146 页的表 9-3 的第一列中列出，或
- 已安装在 Adaptive Server 系统中的公共 Java-SQL 类。

当 SQL 数据类型属于下列情况时，就是可映射数据类型：

- 在第 147 页的表 9-4 的第一列中列出，或
- 已内置或安装在 Adaptive Server 系统中的公共 Java-SQL 类。

如果 Java 方法的所有参数和结果数据类型都是可映射的，则它也是可映射的。

扩大转换

一种 Java 操作，用于将类实例的引用转换为该类的父类实例的引用。此操作用 SQL 的 convert 函数编写。另请参见**缩小转换**。

类

类是 Java 程序的基本元素，其中包含字段声明和方法的集合。类是决定该类中的每个实例的行为和属性的主副本。类定义是活动数据类型的定义，它指定一组合法的值并定义一组处理数值的方法。请参见**类实例**。

类方法

请参见**静态方法**。

类实例

类数据类型的值，它包含类中每个字段的值并接受类的所有方法。

类文件

类型为 “class” 的文件 （例如， *myclass.class*），其中包含 Java 类的已编译字节码。请参见 **Java 文件**和 **Java 档案 (JAR)**。

软件包

包是相关类的集合。一个类指定一个包，或者缺省（匿名）包的一部分。类可以使用 Java import 语句指定其它可从中引用类的包。

声明的类

Java-SQL 数据项的声明数据类型，它是运行时值的数据类型或其父类型。

实例方法

可调用的用来引用类的特定实例的方法。

数据类型映射

Java 与 SQL 数据类型之间的转换。

缩小转换

一种 Java 操作，用于将类实例的引用转换为该类的子类实例的引用。此操作用 SQL 的 convert 函数编写。另请参见**扩大转换**。

同名类

具有相同的完全限定名但安装在不同数据库中的 Java-SQL 类。

外部化

Java 实例的外部化是一种字节流，其中包含足够的信息来重新构造该实例。外部化由可外部化的界面来定义。所有 Java-SQL 类都必须可外部化或可序列化。请参见**序列化**。

序列化

Java 实例的序列化是一种字节流，其中包含足够的信息来识别其类和重新构造该实例。所有 Java-SQL 类都必须可外部化或可序列化。请参见**外部化**。

已安装的类

Java 类和方法由 installjava 实用程序放置在 Adaptive Server 系统中。

子类

在层次结构中处于另一个类之下的类。它继承上级类的特性和行为。子类可与其父类交替使用。子类之上的类是它的直接父类。请参见**父类**、**缩小转换**和**扩大转换**。

字节码

由 Java VM 执行的 Java 源代码的已编译形式。

组织良好的文档

在 XML 中，组织良好的文档必须具有以下特征：所有元素都具有开始和结束标记，特性值放在引号中，所有元素都嵌套正确。

索引

符号

英文

A

B

C

D

F

G

H

J

K

L

M

P

Q

S

T

W

X

Y

Z