# DocumentDB 服务器端编程：存储过程、数据库触发器和 UDF

了解 DocumentDB 的语言如何集成、JavaScript 的事务执行如何使开发人员以 JavaScript 本机编写**存储过程**、**触发器**和**用户定义的函数 (UDF)**。这让你能够编写可以在数据库存储分区上直接传送和执行的数据库程序应用程序逻辑。

我们建议通过观看下面的视频入门，该视频中 Andrew Liu 简单介绍了 DocumentDB 服务器端数据库编程模型。

[AZURE.VIDEO azure-demo-a-quick-intro-to-azure-documentdbs-server-side-javascript]

然后，返回到本文，你将在其中了解以下问题的答案：

* 如何使用 JavaScript 编写存储过程、触发器或 UDF？
* DocumentDB 如何保证 ACID？
* 事务在 DocumentDB 中如何工作？
* 什么是预触发器和后触发器，以及如何编写它们？
* 如何通过使用 HTTP 以 RESTful 方式注册并执行存储过程、触发器或 UDF？
* 可用什么 DocumentDB SDK 来创建和执行存储过程、触发器和 UDF？

## 存储过程和 UDF 编程简介

此种“将 JavaScript 作为新式 T-SQL”的方法让应用程序开发人员摆脱了类型系统不匹配和对象关系映射技术的复杂性。它还具有许多可利用以构建丰富应用程序的内在优势：

* **过程逻辑：**JavaScript 作为一种高级别的编程语言，提供了表达业务逻辑的丰富且熟悉的界面。你可以执行与数据更接近的操作的复杂序列。
* **原子事务：**DocumentDB 保证在单个存储过程或触发器内部执行的数据库操作是原子事务。这使得应用程序能在单个批处理中合并相关操作，因此要么它们全部成功，要么全部不成功。
* **性能：**本质上将 JSON 映射到 Javascript 语言类型系统且它还是 DocumentDB 中存储的基本单位，这一事实允许大量的优化，如缓冲池中 JSON 文档的延迟具体化和使它们按需对执行代码可用。还有更多与传送业务逻辑到数据库相关的性能优点：
  + 批处理 – 开发人员可以分组操作（如插入）并批量提交它们。用于创建单独事务的网络流量延迟成本和存储开销显著降低。
  + 预编译 – DocumentDB 预编译存储过程、触发器和用户定义的函数 (UDF) 以避免每次调用产生的 JavaScript 编译成本。对于过程逻辑生成字节代码的开销被摊销为最小值。
  + 序列化 – 很多操作需要可能涉及执行一个或多个次要存储操作的副作用（“触发器”）。除了原子性之外，当移动到服务器时，它的性能也更高。
* **封装：**可以使用存储过程在一个位置对业务逻辑进行分组。这样做有两个优点：
  + 它会在原始数据之上添加抽象层，这使得数据架构师能够从数据独立发展他们的应用程序。当数据无架构时，如果他们必须直接处理数据，则由于可能需要兼并到应用程序中的脆性假设，使得这样做尤其有益。
  + 这种抽象使企业通过从脚本简化访问来保证他们的数据安全。

数据库触发器、存储过程和自定义查询运算符的创建和执行通过许多平台（包括 .NET、Node.js 和 JavaScript）中的 [REST API](https://msdn.microsoft.com/library/azure/dn781481.aspx)、[DocumentDB Studio](https://github.com/mingaliu/DocumentDBStudio/releases) 和[客户端 SDK](/documentation/articles/documentdb-sdk-dotnet) 得到支持。

**本教程使用** [**具有 Q Promises 的 Node.js SDK**](http://azure.github.io/azure-documentdb-node-q/) 来阐明存储过程、触发器和 UDF 的语法和用法。

## 存储过程

### 示例：编写简单的存储过程

让我们从一个返回“Hello World”响应的简单存储过程开始。

var helloWorldStoredProc = {  
 id: "helloWorld",  
 body: function () {  
 var context = getContext();  
 var response = context.getResponse();  
  
 response.setBody("Hello, World");  
 }  
}

存储过程是按集合注册的，且可以在该集合中存在的任何文档和附件中运作。以下代码段显示如何使用一个集合注册 helloWorld 存储过程。

// register the stored procedure  
var createdStoredProcedure;  
client.createStoredProcedureAsync('dbs/testdb/colls/testColl', helloWorldStoredProc)  
 .then(function (response) {  
 createdStoredProcedure = response.resource;  
 console.log("Successfully created stored procedure");  
 }, function (error) {  
 console.log("Error", error);  
 });

注册存储过程后，我们可以针对集合进行执行，并读取返回到客户端的结果。

// execute the stored procedure  
client.executeStoredProcedureAsync('dbs/testdb/colls/testColl/sprocs/helloWorld')  
 .then(function (response) {  
 console.log(response.result); // "Hello, World"  
 }, function (err) {  
 console.log("Error", error);  
 });

上下文对象提供对所有可在 DocumentDB 存储上执行的操作的访问，以及对请求和响应对象的访问。在本例中，我们使用响应对象来设置发送回客户端的响应的主体。有关更多详细信息，请参阅 [DocumentDB JavaScript 服务器 SDK 文档](http://azure.github.io/azure-documentdb-js-server/)。

让我们扩展此示例，并将更多数据库相关的功能添加到存储过程中。存储过程可以创建、更新、读取、查询和删除集合内部的文档和附件。

### 示例：编写创建文档的存储过程。

下一个代码段将演示如何使用上下文对象与 DocumentDB 资源进行交互。

var createDocumentStoredProc = {  
 id: "createMyDocument",  
 body: function createMyDocument(documentToCreate) {  
 var context = getContext();  
 var collection = context.getCollection();  
  
 var accepted = collection.createDocument(collection.getSelfLink(),  
 documentToCreate,  
 function (err, documentCreated) {  
 if (err) throw new Error('Error' + err.message);  
 context.getResponse().setBody(documentCreated.id)  
 });  
 if (!accepted) return;  
 }  
}

此存储过程将 documentToCreate 作为输入，它是要在当前集合中创建的文档的主体。所有此类操作均是异步操作且依赖 JavaScript 函数回调。回调函数具有两个参数，一个用于错误对象（假如操作失败），另一个用于已创建的对象。在回调内部，用户可以处理异常或引发错误。如果未提供回调而又存在错误，则 DocumentDB 运行时将引发错误。

在上面的示例中，如果操作失败，回调将引发错误。否则，它会将已创建文档的 ID 设置为对客户端的响应的主体。此为该存储过程如何使用输入参数进行执行。

// register the stored procedure  
client.createStoredProcedureAsync('dbs/testdb/colls/testColl', createDocumentStoredProc)  
 .then(function (response) {  
 var createdStoredProcedure = response.resource;  
  
 // run stored procedure to create a document  
 var docToCreate = {  
 id: "DocFromSproc",  
 book: "The Hitchhiker’s Guide to the Galaxy",  
 author: "Douglas Adams"  
 };  
  
 return client.executeStoredProcedureAsync('dbs/testdb/colls/testColl/sprocs/createMyDocument',  
 docToCreate);  
 }, function (error) {  
 console.log("Error", error);  
 })  
.then(function (response) {  
 console.log(response); // "DocFromSproc"  
}, function (error) {  
 console.log("Error", error);  
});

请注意，可以修改该存储过程以将文档主体的数组作为输入并在同一存储过程执行中创建它们全部，而不用执行多个网络请求以单独创建它们中的每一个。这可以用来执行 DocumentDB 的有效批量导入程序（之后会在本教程中讨论）。

所述的示例演示了如何使用存储过程。稍后我们将在教程中介绍触发器和用户定义的函数 (UDF)。

## 数据库程序事务

典型数据库中的事务可以定义为一系列作为单个逻辑单元工作执行的操作。每个事务提供 **ACID 保证**。ACID 是已知的代表四个属性（原子性、一致性、隔离和持续性）的缩写词。

简单地说，原子性保证所有在一个事物内部执行的工作被视为单个单元，其中要么全部工作都提交，要么都不提交。一致性确保数据始终在各个事务间处于良好内部状态。隔离保证没有两个互相干扰的事务存在 – 一般来说，大多数商业系统都提供多个可以基于应用程序需求而使用的隔离级别。持续性确保数据库中提交的任何更改将始终存在。

在 DocumentDB 中，JavaScript 被托管在与数据库相同的内存空间中。因此，在存储过程和触发器内提出的请求将在相同范围的数据库会话中执行。这让 DocumentDB 能够保证所有属于单个存储过程/触发器的操作的 ACID。考虑以下存储过程定义：

// JavaScript source code  
var exchangeItemsSproc = {  
 name: "exchangeItems",  
 body: function (playerId1, playerId2) {  
 var context = getContext();  
 var collection = context.getCollection();  
 var response = context.getResponse();  
  
 var player1Document, player2Document;  
  
 // query for players  
 var filterQuery = 'SELECT \* FROM Players p where p.id = "' + playerId1 + '"';  
 var accept = collection.queryDocuments(collection.getSelfLink(), filterQuery, {},  
 function (err, documents, responseOptions) {  
 if (err) throw new Error("Error" + err.message);  
  
 if (documents.length != 1) throw "Unable to find both names";  
 player1Document = documents[0];  
  
 var filterQuery2 = 'SELECT \* FROM Players p where p.id = "' + playerId2 + '"';  
 var accept2 = collection.queryDocuments(collection.getSelfLink(), filterQuery2, {},  
 function (err2, documents2, responseOptions2) {  
 if (err2) throw new Error("Error" + err2.message);  
 if (documents2.length != 1) throw "Unable to find both names";  
 player2Document = documents2[0];  
 swapItems(player1Document, player2Document);  
 return;  
 });  
 if (!accept2) throw "Unable to read player details, abort ";  
 });  
  
 if (!accept) throw "Unable to read player details, abort ";  
  
 // swap the two players’ items  
 function swapItems(player1, player2) {  
 var player1ItemSave = player1.item;  
 player1.item = player2.item;  
 player2.item = player1ItemSave;  
  
 var accept = collection.replaceDocument(player1.\_self, player1,  
 function (err, docReplaced) {  
 if (err) throw "Unable to update player 1, abort ";  
  
 var accept2 = collection.replaceDocument(player2.\_self, player2,  
 function (err2, docReplaced2) {  
 if (err) throw "Unable to update player 2, abort"  
 });  
  
 if (!accept2) throw "Unable to update player 2, abort";  
 });  
  
 if (!accept) throw "Unable to update player 1, abort";  
 }  
 }  
}  
  
// register the stored procedure in Node.js client  
client.createStoredProcedureAsync(collection.\_self, exchangeItemsSproc)  
 .then(function (response) {  
 var createdStoredProcedure = response.resource;  
 }  
);

此存储过程使用游戏应用内的事务在单个操作中的两个玩家之间交易项。该存储过程尝试读取两个分别与作为参数传递的玩家 ID 对应的文档。如果两个玩家文档都被找到，那么存储过程将通过交换它们的项来更新文档。如果在此过程中遇到了任何错误，它将引发隐式终止事务的 JavaScript 异常。

如果存储过程针对其注册的集合是单区集合，那么该事务的范围为该集合内的所有文档。如果集合已分区，那么存储过程将在单个分区键的事务范围中执行。每个存储过程执行必须包含对应于事务在其下运行的范围的分区键值。有关更多详细信息，请参阅 [DocumentDB 分区](/documentation/articles/documentdb-partition-data)。

### 提交和回滚

事务将在本机深入集成到 DocumentDB 的 JavaScript 编程模型中。在 JavaScript 函数内，所有操作都在单个事务下自动包装。如果 JavaScript 在没有任何异常的情况下完成，将提交针对数据库的操作。实际上，关系数据库中的“BEGIN TRANSACTION”和“COMMIT TRANSACTION”语句在 DocumentDB 中是隐式的。

如果存在任何传播自脚本的异常，DocumentDB 的 JavaScript 运行时将回滚整个事务。正如之前的示例中所示，引发异常实际上等同于 DocumentDB 中的“ROLLBACK TRANSACTION”。

### 数据一致性

存储过程和触发器始终在 DocumentDB 集合的主要副本上执行。这确保了从存储过程内部的读取提供强一致性。使用用户定义的函数的查询可以在主要或任何次要副本上执行，但通过选择合适的副本我们可以确保满足所要求的一致性级别。

## 绑定的执行

所有 DocumentDB 操作必须在服务器指定的请求超时持续时间内完成。此约束也适用于 JavaScript 函数（存储过程、触发器和用户定义的函数）。如果某个操作未在时间限制内完成，则将回滚事务。JavaScript 函数必须在时间限制内完成，或实施一个基于延续的模型以批处理/继续执行过程。

为了简化存储过程和触发器的开发以处理时间限制，集合对象内的所有函数（文档和附件的创建、读取、替换和删除）返回表示该操作是否完成的布尔值。如果该值为 false，则表示时间限制将要过期且该过程必须完成执行。如果存储过程及时完成且没有任何更多请求在排队的话，将保证完成排在第一个拒绝存储操作之前的操作。

JavaScript 函数也被绑定在资源消耗量上。DocumentDB 基于预配的数据库帐户大小按集合保留吞吐量。吞吐量按照规范化单位的 CPU、内存和 IO 消耗量（称为请求单位或 RU）来表示。JavaScript 函数可能在短时间内耗尽大量的 RU，如果达到了集合的限制，则可能受到速率限制。资源密集型存储过程还可能被隔离以确保原始数据库操作的可用性。

### 示例：将数据批量导入到数据库程序中

下面是一个编写以批量导入文档到集合的存储过程的示例。请注意存储过程通过检查来自 createDocument 的布尔返回值，然后使用插入在每次存储过程调用中的文档的计数以在批处理之间跟踪和恢复进度，从而处理绑定执行的方式。

function bulkImport(docs) {  
 var collection = getContext().getCollection();  
 var collectionLink = collection.getSelfLink();  
  
 // The count of imported docs, also used as current doc index.  
 var count = 0;  
  
 // Validate input.  
 if (!docs) throw new Error("The array is undefined or null.");  
  
 var docsLength = docs.length;  
 if (docsLength == 0) {  
 getContext().getResponse().setBody(0);  
 }  
  
 // Call the create API to create a document.  
 tryCreate(docs[count], callback);  
  
 // Note that there are 2 exit conditions:  
 // 1) The createDocument request was not accepted.   
 // In this case the callback will not be called, we just call setBody and we are done.  
 // 2) The callback was called docs.length times.  
 // In this case all documents were created and we don’t need to call tryCreate anymore. Just call setBody and we are done.  
 function tryCreate(doc, callback) {  
 var isAccepted = collection.createDocument(collectionLink, doc, callback);  
  
 // If the request was accepted, callback will be called.  
 // Otherwise report current count back to the client,   
 // which will call the script again with remaining set of docs.  
 if (!isAccepted) getContext().getResponse().setBody(count);  
 }  
  
 // This is called when collection.createDocument is done in order to process the result.  
 function callback(err, doc, options) {  
 if (err) throw err;  
  
 // One more document has been inserted, increment the count.  
 count++;  
  
 if (count >= docsLength) {  
 // If we created all documents, we are done. Just set the response.  
 getContext().getResponse().setBody(count);  
 } else {  
 // Create next document.  
 tryCreate(docs[count], callback);  
 }  
 }  
}

## 数据库触发器

### 数据库预触发器

DocumentDB 提供通过文档中的操作执行或触发的触发器。例如，当创建文档时你可以指定预触发器 – 此预触发器将在文档创建之前运行。下面就是如何使用预触发器来验证正在创建的文档的属性的示例：

var validateDocumentContentsTrigger = {  
 name: "validateDocumentContents",  
 body: function validate() {  
 var context = getContext();  
 var request = context.getRequest();  
  
 // document to be created in the current operation  
 var documentToCreate = request.getBody();  
  
 // validate properties  
 if (!("timestamp" in documentToCreate)) {  
 var ts = new Date();  
 documentToCreate["my timestamp"] = ts.getTime();  
 }  
  
 // update the document that will be created  
 request.setBody(documentToCreate);  
 },  
 triggerType: TriggerType.Pre,  
 triggerOperation: TriggerOperation.Create  
}

该触发器对应的 Node.js 客户端注册代码：

// register pre-trigger  
client.createTriggerAsync(collection.self, validateDocumentContentsTrigger)  
 .then(function (response) {  
 console.log("Created", response.resource);  
 var docToCreate = {  
 id: "DocWithTrigger",  
 event: "Error",  
 source: "Network outage"  
 };  
  
 // run trigger while creating above document   
 var options = { preTriggerInclude: "validateDocumentContents" };  
  
 return client.createDocumentAsync(collection.self,  
 docToCreate, options);  
 }, function (error) {  
 console.log("Error", error);  
 })  
.then(function (response) {  
 console.log(response.resource); // document with timestamp property added  
}, function (error) {  
 console.log("Error", error);  
});

预触发器不能有任何输入参数。可以使用请求对象操纵与操作相关联的请求消息。此处，预触发器随文档的创建而运行，且请求消息正文包含要以 JSON 格式创建的文档。

当注册触发器后，用户可以指定随触发器一起运行的操作。此触发器使用的是 TriggerOperation.Create 创建的，这意味着下列操作不被允许。

var options = { preTriggerInclude: "validateDocumentContents" };  
  
client.replaceDocumentAsync(docToReplace.self,  
 newDocBody, options)  
.then(function (response) {  
 console.log(response.resource);  
}, function (error) {  
 console.log("Error", error);  
});  
  
// Fails, can’t use a create trigger in a replace operation

### 数据库后触发器

后触发器，跟预触发器一样，与文档上的操作相关联且不接受任何输入参数。它们在操作完成**后**运行，且具有对发送到客户端的响应消息的访问权限。

下面的示例显示正在运作的后触发器：

var updateMetadataTrigger = {  
 name: "updateMetadata",  
 body: function updateMetadata() {  
 var context = getContext();  
 var collection = context.getCollection();  
 var response = context.getResponse();  
  
 // document that was created  
 var createdDocument = response.getBody();  
  
 // query for metadata document  
 var filterQuery = 'SELECT \* FROM root r WHERE r.id = "\_metadata"';  
 var accept = collection.queryDocuments(collection.getSelfLink(), filterQuery,  
 updateMetadataCallback);  
 if(!accept) throw "Unable to update metadata, abort";  
   
 function updateMetadataCallback(err, documents, responseOptions) {  
 if(err) throw new Error("Error" + err.message);  
 if(documents.length != 1) throw 'Unable to find metadata document';  
   
 var metadataDocument = documents[0];  
   
 // update metadata  
 metadataDocument.createdDocuments += 1;  
 metadataDocument.createdNames += " " + createdDocument.id;  
 var accept = collection.replaceDocument(metadataDocument.\_self,  
 metadataDocument, function(err, docReplaced) {  
 if(err) throw "Unable to update metadata, abort";  
 });  
 if(!accept) throw "Unable to update metadata, abort";  
 return;   
 }   
 },  
 triggerType: TriggerType.Post,  
 triggerOperation: TriggerOperation.All  
}

可以按照下面示例中所示方法注册触发器。

// register post-trigger  
client.createTriggerAsync('dbs/testdb/colls/testColl', updateMetadataTrigger)  
 .then(function(createdTrigger) {   
 var docToCreate = {   
 name: "artist\_profile\_1023",  
 artist: "The Band",  
 albums: ["Hellujah", "Rotators", "Spinning Top"]  
 };  
  
 // run trigger while creating above document   
 var options = { postTriggerInclude: "updateMetadata" };  
   
 return client.createDocumentAsync(collection.self,  
 docToCreate, options);  
 }, function(error) {  
 console.log("Error" , error);  
 })  
.then(function(response) {  
 console.log(response.resource);   
}, function(error) {  
 console.log("Error" , error);  
});

此触发器查询元数据文档并在其中更新新建文档的详细信息。

务必要注意的一点是 DocumentDB 中触发器的**事务**执行。此后触发器作为与原始文档的创建相同的事务的一部分运行。因此，如果我们从后触发器引发异常（假设我们无法更新元数据文档），那么整个事务都将失败并回滚。不会创建文档，而将返回异常。

## 用户定义的函数

将用户定义的函数 (UDF) 用来扩展 DocumentDB SQL 查询语言语法和实现自定义业务逻辑。它们只能从查询内部调用。它们不具有对上下文对象的访问权限且旨在被用作仅计算的 JavaScript。因此，UDF 可以在 DocumentDB 服务的次要副本上运行。

以下示例创建 UDF 来计算基于各种收入档次的税率的所得税，然后在查询内部使用它查找所有支付税款超过 $20,000 的人。

var taxUdf = {  
 name: "tax",  
 body: function tax(income) {  
  
 if(income == undefined)   
 throw 'no input';  
  
 if (income < 1000)   
 return income \* 0.1;  
 else if (income < 10000)   
 return income \* 0.2;  
 else  
 return income \* 0.4;  
 }  
}

UDF 随后可以用在诸如下面示例的查询中：

// register UDF  
client.createUserDefinedFunctionAsync('dbs/testdb/colls/testColl', taxUdf)  
 .then(function(response) {   
 console.log("Created", response.resource);  
  
 var query = 'SELECT \* FROM TaxPayers t WHERE udf.tax(t.income) > 20000';   
 return client.queryDocuments('dbs/testdb/colls/testColl',  
 query).toArrayAsync();  
 }, function(error) {  
 console.log("Error" , error);  
 })  
.then(function(response) {  
 var documents = response.feed;  
 console.log(response.resource);   
}, function(error) {  
 console.log("Error" , error);  
});

## JavaScript 语言集成的查询 API

除了使用 DocumentDB 的 SQL 语法发起查询外，服务器端 SDK 还允许你在没有任何 SQL 知识的情况下使用流畅的 JavaScript 接口来执行优化的查询。JavaScript 查询 API 允许你使用与 ECMAScript5 的数组内置项类似的语法和如 lodash 等热门的 JavaScript 库，通过将谓词函数传递到可链的函数调用中以编程方式生成查询。使用 DocumentDB 的索引进行有效执行的 JavaScript 运行时将对查询进行分析。

[AZURE.NOTE] \_\_（双下划线）是 getContext().getCollection() 的别名。换言之，你可以使用 \_\_ 或 getContext().getCollection() 来访问 JavaScript 查询 API。

支持的函数包括：

chain() … .value([callback] [, options])

发起一个必须用 value() 终止的连锁调用。

filter(predicateFunction [, options] [, callback])

使用返回 true/false 的谓词函数对输入进行筛选，以便将输入文档筛选出或筛选到结果集。这与 SQL 中的 WHERE 子句行为相似。

map(transformationFunction [, options] [, callback])

应用给定一个转换函数的投影，该函数将每个输入项映射到 JavaScript 对象或值。这与 SQL 中的 SELECT 子句行为相似。

pluck([propertyName] [, options] [, callback])

此为从每个输入项提取单个属性的值的映射的快捷方式。

flatten([isShallow] [, options] [, callback])

将每个输入项合且并平展到单个数组。这与 LINQ 中的 SelectMany 行为相似。

sortBy([predicate] [, options] [, callback])

通过使用给定的谓词在输入文档流中按升序方式对文档进行排序来生成新的一组文档。这与 SQL 中的 ORDER BY 子句行为相似。

sortByDescending([predicate] [, options] [, callback])

通过使用给定的谓词在输入文档流中按降序方式对文档进行排序来生成新的一组文档。这与 SQL 中的 ORDER BY x DESC 子句行为相似。

当在其中包含谓词和/或选择器函数时，以下 JavaScript 构造将自动优化以在 DocumentDB 索引上直接运行：

* 简单的运算符：= + - \* / % | ^ & == != === !=== < > <= >= || && << >> >>>! ~
* 文本（包括对象文本）：{}
* var, return

以下 JavaScript 构造不会对 DocumentDB 索引进行优化：

* 控制流（如 if、for、while）
* 函数调用

有关详细信息，请查看我们的[服务器端 JSDocs](http://azure.github.io/azure-documentdb-js-server/)。

### 示例：使用 JavaScript 查询 API 编写存储过程

下面的代码示例是一个有关可如何在存储过程的上下文中使用 JavaScript 查询 API 的示例。存储过程使用 \_\_.filter() 方法插入一个由输入参数给定的文档并更新元数据文档，其中 minSize、maxSize 和 totalSize 以输入文档的大小属性为基础。

/\*\*  
 \* Insert actual doc and update metadata doc: minSize, maxSize, totalSize based on doc.size.  
 \*/  
function insertDocumentAndUpdateMetadata(doc) {  
 // HTTP error codes sent to our callback funciton by DocDB server.  
 var ErrorCode = {  
 RETRY\_WITH: 449,  
 }  
  
 var isAccepted = \_\_.createDocument(\_\_.getSelfLink(), doc, {}, function(err, doc, options) {  
 if (err) throw err;  
  
 // Check the doc (ignore docs with invalid/zero size and metaDoc itself) and call updateMetadata.  
 if (!doc.isMetadata && doc.size > 0) {  
 // Get the meta document. We keep it in the same collection. it's the only doc that has .isMetadata = true.  
 var result = \_\_.filter(function(x) {  
 return x.isMetadata === true  
 }, function(err, feed, options) {  
 if (err) throw err;  
  
 // We assume that metadata doc was pre-created and must exist when this script is called.  
 if (!feed || !feed.length) throw new Error("Failed to find the metadata document.");  
  
 // The metadata document.  
 var metaDoc = feed[0];  
  
 // Update metaDoc.minSize:  
 // for 1st document use doc.Size, for all the rest see if it's less than last min.  
 if (metaDoc.minSize == 0) metaDoc.minSize = doc.size;  
 else metaDoc.minSize = Math.min(metaDoc.minSize, doc.size);  
  
 // Update metaDoc.maxSize.  
 metaDoc.maxSize = Math.max(metaDoc.maxSize, doc.size);  
  
 // Update metaDoc.totalSize.  
 metaDoc.totalSize += doc.size;  
  
 // Update/replace the metadata document in the store.  
 var isAccepted = \_\_.replaceDocument(metaDoc.\_self, metaDoc, function(err) {  
 if (err) throw err;  
 // Note: in case concurrent updates causes conflict with ErrorCode.RETRY\_WITH, we can't read the meta again   
 // and update again because due to Snapshot isolation we will read same exact version (we are in same transaction).  
 // We have to take care of that on the client side.  
 });  
 if (!isAccepted) throw new Error("replaceDocument(metaDoc) returned false.");  
 });  
 if (!result.isAccepted) throw new Error("filter for metaDoc returned false.");  
 }  
 });  
 if (!isAccepted) throw new Error("createDocument(actual doc) returned false.");  
}

## SQL 到 Javascript 备忘单

下表表示各种不同的 SQL 查询和对应的 JavaScript 查询。

对于 SQL 查询，文档属性密钥（例如 doc.id）需区分大小写。

SQL

JavaScript 查询 API

详细信息

结果为所有文档（使用延续令牌分页）保持原样。

从所有文档投影 ID、消息（别名为 msg）和操作。

查询具有此谓词的文档：id = “X998\_Y998”。

查询具有 Tags 属性且 Tags 为一个包含值 123 的数组的文档。

查询具有谓词 id = “X998\_Y998” 的文档，然后投影 ID 和消息（别名为 msg）。

筛选具有数组属性 Tags 的文档，按 \_ts 时间戳系统属性对结果文档进行排序，然后投影并平展 Tags 数组。

## 运行时支持

[DocumentDB JavaScript 服务器端 SDK](http://azure.github.io/azure-documentdb-js-server/) 为大多数由 [ECMA-262](http://www.ecma-international.org/publications/standards/Ecma-262.htm) 规范化的主要 JavaScript 语言功能提供支持。

### “安全”

JavaScript 存储过程和触发器经过沙盒处理，以使一个脚本的效果不会在未经过数据库级别的快照事务隔离的情况下泄漏到其他脚本。运行时环境是共用的，但是在每次运行后都会清理上下文。因此可以保证它们安全避免互相之间的任何意外副作用。

### 预编译

存储过程、触发器和 UDF 是隐式预编译到字节代码格式的，这是为了避免每次脚本调用时产生的编译成本。这可确保存储过程的调用迅速且痕迹较少。

## 客户端 SDK 支持

除了 [Node.js](/documentation/articles/documentdb-sdk-node) 客户端之外，DocumentDB 还支持 [.NET](/documentation/articles/documentdb-sdk-dotnet)、[Java](/documentation/articles/documentdb-sdk-java)、[JavaScript](http://azure.github.io/azure-documentdb-js/) 和 [Python SDK](/documentation/articles/documentdb-sdk-python)。也可以使用这些 SDK 来创建和执行存储过程、触发器和 UDF。以下示例演示如何使用 .NET 客户端创建和执行存储过程。请注意 .NET 类型是如何以 JSON 传递到存储过程中并从中读回的。

var markAntiquesSproc = new StoredProcedure  
{  
 Id = "ValidateDocumentAge",  
 Body = @"  
 function(docToCreate, antiqueYear) {  
 var collection = getContext().getCollection();   
 var response = getContext().getResponse();   
  
 if(docToCreate.Year != undefined && docToCreate.Year < antiqueYear){  
 docToCreate.antique = true;  
 }  
  
 collection.createDocument(collection.getSelfLink(), docToCreate, {},   
 function(err, docCreated, options) {   
 if(err) throw new Error('Error while creating document: ' + err.message);   
 if(options.maxCollectionSizeInMb == 0) throw 'max collection size not found';   
 response.setBody(docCreated);  
 });  
 }"  
};  
  
// register stored procedure  
StoredProcedure createdStoredProcedure = await client.CreateStoredProcedureAsync(UriFactory.CreateDocumentCollectionUri("db", "coll"), markAntiquesSproc);  
dynamic document = new Document() { Id = "Borges\_112" };  
document.Title = "Aleph";  
document.Year = 1949;  
  
// execute stored procedure  
Document createdDocument = await client.ExecuteStoredProcedureAsync<Document>(UriFactory.CreateStoredProcedureUri("db", "coll", "sproc"), document, 1920);

此示例演示如何使用 [.NET SDK](https://msdn.microsoft.com/library/azure/dn948556.aspx) 创建预触发器并使用启用的触发器创建文档。

Trigger preTrigger = new Trigger()  
{  
 Id = "CapitalizeName",  
 Body = @"function() {  
 var item = getContext().getRequest().getBody();  
 item.id = item.id.toUpperCase();  
 getContext().getRequest().setBody(item);  
 }",  
 TriggerOperation = TriggerOperation.Create,  
 TriggerType = TriggerType.Pre  
};  
  
Document createdItem = await client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri("db", "coll"), new Document { Id = "documentdb" },  
 new RequestOptions  
 {  
 PreTriggerInclude = new List<string> { "CapitalizeName" },  
 });

下面的示例则演示如何创建用户定义的函数 (UDF) 并在 [DocumentDB SQL 查询](/documentation/articles/documentdb-sql-query)中使用它。

UserDefinedFunction function = new UserDefinedFunction()  
{  
 Id = "LOWER",  
 Body = @"function(input)   
 {  
 return input.toLowerCase();  
 }"  
};  
  
foreach (Book book in client.CreateDocumentQuery(UriFactory.CreateDocumentCollectionUri("db", "coll"),  
 "SELECT \* FROM Books b WHERE udf.LOWER(b.Title) = 'war and peace'"))  
{  
 Console.WriteLine("Read {0} from query", book);  
}

## REST API

所有 DocumentDB 操作都能够以 RESTful 方式执行。可以通过在集合下使用 HTTP POST 来注册存储过程、触发器和用户定义的函数。下面为如何注册存储过程的一个示例：

POST https://<url>/sprocs/ HTTP/1.1  
authorization: <<auth>>  
x-ms-date: Thu, 07 Aug 2014 03:43:10 GMT  
  
  
var x = {  
 "name": "createAndAddProperty",  
 "body": function (docToCreate, addedPropertyName, addedPropertyValue) {  
 var collectionManager = getContext().getCollection();  
 collectionManager.createDocument(  
 collectionManager.getSelfLink(),  
 docToCreate,  
 function(err, docCreated) {  
 if(err) throw new Error('Error: ' + err.message);  
 docCreated[addedPropertyName] = addedPropertyValue;  
 getContext().getResponse().setBody(docCreated);  
 });  
 }  
}

通过针对 URI dbs/testdb/colls/testColl/sprocs 执行 POST 请求（其主体包含要创建的存储过程）来注册存储过程。同样，可以通过分别针对 /triggers 和 /udfs 发出 POST 来注册触发器和 UDF。 然后可以通过针对其资源链接发出 POST 请求来执行此存储过程。

POST https://<url>/sprocs/<sproc> HTTP/1.1  
authorization: <<auth>>  
x-ms-date: Thu, 07 Aug 2014 03:43:20 GMT  
  
  
[ { "name": "TestDocument", "book": "Autumn of the Patriarch"}, "Price", 200 ]

此处，存储过程的输入在请求主体中传递。请注意，输入是作为输入参数的 JSON 数组进行传递的。存储过程将第一个输入作为响应主体的文档。我们收到的响应如下：

HTTP/1.1 200 OK  
   
{   
 name: 'TestDocument',  
 book: ‘Autumn of the Patriarch’,  
 id: ‘V7tQANV3rAkDAAAAAAAAAA==‘,  
 ts: 1407830727,  
 self: ‘dbs/V7tQAA==/colls/V7tQANV3rAk=/docs/V7tQANV3rAkDAAAAAAAAAA==/’,  
 etag: ‘6c006596-0000-0000-0000-53e9cac70000’,  
 attachments: ‘attachments/’,  
 Price: 200  
}

与存储过程不一样，不能直接执行触发器。相反，它们将作为文档上的操作的一部分进行执行。我们可以使用 HTTP 标头，指定触发器通过请求进行运行。下面为创建文档的请求。

POST https://<url>/docs/ HTTP/1.1  
authorization: <<auth>>  
x-ms-date: Thu, 07 Aug 2014 03:43:10 GMT  
x-ms-documentdb-pre-trigger-include: validateDocumentContents   
x-ms-documentdb-post-trigger-include: bookCreationPostTrigger  
  
  
{  
 "name": "newDocument",  
 “title”: “The Wizard of Oz”,  
 “author”: “Frank Baum”,  
 “pages”: 92  
}

此处，要通过请求运行的预触发器在 x-ms-documentdb-pre-trigger-include 标头中指定。相应地，任何后触发器将在 x-ms-documentdb-post-trigger-include 标头中给定。请注意，可以针对某个给定的请求指定预触发器和后触发器。

## 代码示例

你可以在我们的 [Github 存储库](https://github.com/Azure/azure-documentdb-js-server/tree/master/samples)上查找更多服务器端代码示例（包括 [bulk-delete](https://github.com/Azure/azure-documentdb-js-server/tree/master/samples/stored-procedures/bulkDelete.js) 和 [update](https://github.com/Azure/azure-documentdb-js-server/tree/master/samples/stored-procedures/update.js)）。

想要共享你令人惊叹的存储过程吗？ 请向我们发送拉取请求！

## 后续步骤

在你创建了一个或多个存储过程、触发器和用户定义的函数之后，可以使用脚本资源管理器在 Azure 门户中加载和查看它们。有关详细信息，请参阅[使用 DocumentDB 脚本资源管理器查看存储过程、触发器和用户定义的函数](/documentation/articles/documentdb-view-scripts)。

还可以查找以下参考和资源，可帮助你了解更多有关 DocumentDB 服务器端编程的信息：

* [Azure DocumentDB SDK](https://msdn.microsoft.com/library/azure/dn781482.aspx)
* [DocumentDB Studio](https://github.com/mingaliu/DocumentDBStudio/releases)
* [JSON](http://www.json.org/)
* [JavaScript ECMA-262](http://www.ecma-international.org/publications/standards/Ecma-262.htm)
* [JavaScript – JSON 类型系统](http://www.json.org/js.html)
* [安全和可移植的数据库扩展性](http://dl.acm.org/citation.cfm?id=276339)
* [面向服务的数据库体系结构](http://dl.acm.org/citation.cfm?id=1066267&coll=Portal&dl=GUIDE)
* [在 Microsoft SQL 服务器中托管 .NET 运行时](http://dl.acm.org/citation.cfm?id=1007669)