存储过程功能开发文档

修订历史

版本	修订日期	修订描述	作者	备注
Cedar 0.1	2016-01-03	存储过程功能开发文档	祝君	
Cedar 0.2	2016-06-15	存储过程功能开发文档	朱涛 王冬慧	

1 总体设计

1.1 综述

Cedar是由华东师范大学数据科学与工程研究院基于OceanBase 0.4.2 研发的可扩展的关系数据库，实现了巨大数据量上的跨行跨表事务。Cedar 0.1版本中的存储过程是一组为了完成特定功能的SQL 语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。Cedar 0.2版本中将存储过程定义为事务，存储过程的执行代表一个事务的执行，满足事务的ACID特性。本版本还添加了存储过程缓存管理模块和事务编译优化模块对事务执行进行优化，提升事务执行的效率。

1.2 名词解释

- 主控服务器（RootServer, RS）：Cedar集群的主控节点，负责管理集群中的所有服务器，以及维护tablet信息。
- 更新服务器（UpdateServer, UPS）：负责存储Cedar系统的增量数据，并提供事务支持。
- 基准数据服务器（ChunkServer, CS）：负责存储Cedar系统的基线数据。
- 合并服务器（MergeServer, MS）：负责接收并解析客户端的SQL请求，经过词法分析、语法分析、查询优化等一系列操作后发送到CS和UPS，合并基线数据和增量数据。
- 存储过程（Stored Procedure）：一组为了完成特定功能的SQL语句集，存储在数据库中，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。
- 数据库事务(Database Transaction)，是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。

1.3 功能

存储过程是数据库系统的一个功能拓展，将一组带有特定功能的SQL语句以及控制流程语句存储在服务器端，用户通过存储过程的名称以及相应参数进行调用。存储过程支持变量，顺序，条件和循环等控制结构，存储过程可以定义参数这使得存储过程具有很强的模块性同时具有批处理性。虽然存储过程没有返回值，但是存储过程可以通过定义输出参数来实现返回结果的功能。这些特性都使得存储过程成为数据库系统必不可少的功能。由于存储过程的源码是存储在服务器的，因此客户端可以通过存储过程名以及参数调用，不需要把这些SQL语句集合发送到服务器，这样的话可以减少网络中传输的数据量，降低网络负荷。

我们将事务封装在存储过程中，这样做的优势在于事务的执行不会受到外界的干扰。Cedar 0.2版本在事务执行前对事务进行了编译优化，分析指令之间的依赖关系，调整指令的执行顺序，将同种类型指令进行成组执行，使得事务执行能够获得更高的效率。此外，在Cedar 0.2版本中还添加了存储过程缓存管理，通过缓存存储过程的物理计划，使得每次执行存储过程时减少编译次数，提升整体执行效率。

1.4 性能指标

对性能无影响。

2 模块设计

2.1 存储过程缓存管理

2.1.1 概述

在系统中对存储过程进行缓存，能够减少从CS上读取存储过程主体代码次数和存储过程主体代码编译成物理计划的次数，使得在调用存储过程时加快执行速度。因此，如何对存储过程进行缓存管理是极其重要的一个部分。

2.1.2 缓存管理模块设计

存储过程的缓存管理根据功能区分主要分为以下几个模块：存储过程的创建；存储过程的删除；存储过程物理计划缓存的创建；存储过程物理计划缓存的删除。为了管理物理计划缓存，创建了两个类（如下所示）用于管理系统的存储过程物理计划。

其中，ObNameCodeMap主要用于存储全局的存储过程名与存储过程主体的hashmap。
ObProcCache主要用于存储存储过程名与物理计划缓存版本的hashmap。
ObProcedureManager作为MS端向RS端获取存储过程全量信息和编译MS端存储过程为物理计划缓存到session中的管理者。

```
class ObNameCodeMap
{
  public:
    typedef hash::ObHashMap<ObString, int64_t> ObProcHashCache;
  private:
      hash::ObHashMap<ObString, ObString> name_code_map_;
      ObProcHashCache name_hash_map_;
      common::ObStringBuf arena_;
      bool is_ready_;
};
```

```
class ObProcedureManager : public tbsys::CDefaultRunnable
{
  public:
    typedef hash::ObHashMap<ObString, sql::ObSQLResultSet *> ObProc
Cache;
  private:
    mutable tbsys::CThreadMutex lock_;
    ModuleArena arena_;
    ObStringBuf proc_name_buf_;
    mergeserver::ObMergeServerService * mergeserver_service_;
    common::DefaultBlockAllocator block_allocator_;
    ObSQLSessionInfo session_;
    ObProcCache name_cache_map_;
    ObNameCodeMap name_code_map_;
    bool has_init_;
};
```

存储过程的保存

当客户端发起创建一个存储过程的请求时，该存储过程需要始终保存在系统中。这里我们采用的基本方法是在创建阶段，将存储过程的源码保存到系统表中。Cedar 0.1实现的方法是在生成create procedure物理计划时，在物理计划树上挂一个insert_op_，从而显式构建一个insert语句，实现将存储过程保存到系统表__all_procedure中去。在实现存储过程物理计划缓存的过程中，我们将insert过程放在RS上进行。具体的流程如下：

1. MS生成create procedure语句的物理执行计划后，打开procedure_create 物理操作符，将source code和procedure name序列化为一个packet发送给RS；
2. RS在接收到MS发送的create procedure包后，反序列化后构造出增量mutator，发送给UPS执行；
3. UPS接收到mutator后，更新系统表__all_procedure，将新建的存储过程插入到表中去；
4. RS在收到UPS更新增量成功的response后，将source code和procedure name插入到本地的ObNameCodeMap中；
5. RS将source code和procedure name序列化为一个packet发送给每个MS，MS接收到

信息后将其保存在本地ObNameCodeMap中。

RS端的ObNameCodeMap在RS宕机后就丢失了，因此在系统重启时，RS需要从磁盘读取__all_procedure系统表，将其中的存储过程信息全部添加到ObHashCodeMap中去。

存储过程的删除

当客户端显式的删除存储过程时，需要从系统表中删除对应过程的记录项。同存储过程的保存类似，之前实现的方法是在生成drop procedure物理计划时，在物理计划树上挂一个delete_op_，构建出delete语句，将其从系统表中删除。我们重写了drop procedure过程，将delete_op_删除，procedure的删除过程同样放在RS上操作。具体流程类似create procedure。

存储过程物理计划缓存的创建

每个 客户端连接MS的时候会产生一个session，在session中调用一个存储过程时，首先根据存储过程名判断session本地有没有缓存物理计划（存储过程创建时，RS会通知所有的MS，MS会缓存存储过程源码），若没有，去 MS ObNameCodeMap缓存中读取源码，然后重新编译，编译成功后将物理计划缓存在session中以备重复调用。缓存管理的优势在于，一方面是避免每次调用的时候都要从RS端获取源码，造成RS的压力，以及增大延迟；另外一方面是降低每次调用时MS编译存储过程的开销。

缓存需要在以下情况创建：1、创建存储过程时；2、MS上线时。

1. 创建存储过程时。MS将source code和procedure name发送给RS，RS完成存储过程的保存操作后，将其插入到本地的ObNameCodeMap中去，通知其他MS更新缓存。
2. MS上线分为两种，一种是初次启动，一种是重启。但是无论哪种，都需要从RS端主动获取存储过程全量信息，将procedure name 和procedure source code存储在本地ObNameCodeMap。由于MS上线时，schema等信息初始化的缓慢，在MS主动向RS获取存储过程全量信息的过程中往往会出现失败。因此，我们将ObProcedureManager写成一个线程的方式。与schema生成并行执行，当MS上的schema初始化完成的时候，再向RS获取信息。

存储过程物理计划缓存的删除

当存储过程删除后，需要将各个server上的ObNameCodeMap缓存信息修改，并且将session中的物理计划缓存清空。在drop procedure时，MS将procedure name发送给RS，RS反序列化后构造出增量mutator，发送给UPS执行删除操作。接着RS通知全局MS将保存在本地的ObNameCodeMap中对应procedure name的记录项删除。最后将session中对应存储过程的物理计划删除。该存储过程随之失效。

2.1.3 思考

为什么不直接在MS上存储已经编译好的存储过程，调用时直接使用物理计划？

一方面，物理计划的复制几乎与物理计划的编译消耗的时间差不多，这样做的缓存几乎不能提交时间效率。另一方面，物理计划占用的空间要比存储过程源码占用的空间要大。

2.2 编译优化

在存储过程中，每个指令存在三种执行模式：1. local processing; 2. cs-rpc; 3. ups-rpc。优化目标是调整不同类型指令的执行次序，将 ups-rpc 合并在一起执行。每个指令需要告诉优化器它们的执行类型。通过3个bit位(CallType)来表示一个指令中包含的操作类型（对于包含嵌套结构的指令，内部会同时包含不同类型的指令）。

在CallType中，对应bit位代表的含义如下：

- 1号bit位代表是否存在 local processing
- 2号bit位代表是否存在 cs-rpc
- 3号bit位代表是否存在 ups-rpc

每个指令提供以下接口告诉优化器它的执行类型：

```
CallType get_call_type() const
```

2.2.1 操作对象识别

指令次序的调整不能影响原来存储过程的语义。判断指令之间的次序是否可以调整，关键是判断不同指令对相同的对象（变量，或者数据项）的读写次序是否敏感。因此，首先要判断一个指令读写的对象集合。

变量的识别

变量的识别在生成物理计划时完成。由于所有的变量都保存在表达式中。我们通过分析表达式的逻辑计划找出所有的变量名。对每个存储过程需要支持的语法类型，它们在从逻辑计划生成物理计划、处理表达式时，需要同时分析它们使用的变量名，并保存到一个变量集合中。对于**Update，Insert，Select...For Update** 这类操作要特殊处理。这些操作实际上会产生两个子操作，一个用于从CS端获得基线数据，一个用于向UPS推送增量修改。这两个操作的变量识别要独立处理。

对每个指令最终对外提供两个接口，用于在优化阶段分析指令之间的依赖关系：

```
virtual void get_read_variable_set(SpVariableSet &read_set);
virtual void get_write_variable_set(SpVariableSet &write_set);
```

数据项的识别

数据项是一个SQL语句可能访问的数据库中的内容。在当前的架构下，一个完整的数据库包含以下部分：

- CS 上的数据纪录：包含了所有的基线数据，属于只读数据，任何SQL对其的访问顺序可以任意调整。
- UPS 上的数据纪录：包含了所有的增量数据，属于读写数据，需要保证不同SQL对其

访问的正确数据。

因此，对数据库纪录的识别主要是针对UPS上可能访问的数据纪录。事实上，在存储过程执行前，我们并不知道实际执行参数的取值。因此，无法从很小的粒度上去判别操作可能访问的数据。我们只能知道一个SQL会访问哪张数据表。因此，数据项的识别是确定SQL需要访问的表。该阶段同样是在生成物理计划过程中完成的，通过分析SQL的逻辑计划确定需要访问的表。

2.2.2 依赖关系识别

分析了每个指令访问的变量集合以及表集合后，我们可以判定指令之间是否存在访问依赖。依赖类型主要为以下四种（假设指令a 先于指令b 执行）：

- True Dependence: 指令b读取了指令a的写入
- Anti Dependence: 指令a读取了指令b将要写入的对象
- Output Dependence: 指令a与指令b均修改同一个对象。
- Control Dependence: 指令a的执行决定了指令b是否执行。

可通过函数check_dep()判断指令间依赖关系逻辑：

```
int SpInst::check_dep(SpInst &inst_in, SpInst &inst_out);
```

其中 Control Dependence 存在于分支控制结构中。其他的几种依赖类型通过分析每个指令的读写集合是否相交进行判定。此外，还有一种特殊的 True Dependence 需要引入。对应于同一个SQL产生的基线查询和增量修改指令间存在 True Dependence。这是因为，增量修改指令需要读取基线查询指令产生的基线数据。

2.2.3 建立指令依赖图

我们在ob_procedure_optimizer.h中采用类ObProcDepGraph建立了一个存储过程中，不同指令之间的依赖图。对存在依赖的指令之间，我们建立一条有向边（指向的方向在不同的优化策略下有所不同，在调整指令执行次序的优化中，我们实际上建立的是一张反向图），并且记录依赖的类型。

```
class ObProcDepGraph
{
    struct SpNode
    {
        int64_t id_;
        int dep_type_;
        SpNode *next_;
    };
    ObSEArray<SpNode, MAX_GRAPH_SIZE> graph_;
}
```

目前依赖图的建立取决于存储过程中最上层的指令，并且所有的嵌套结构都作为一个单独的指令处理。对于最上层的指令，程序分析指令之间的依赖关系，并在依赖图中插入相应的依赖边。因此，依赖图中显示指令之间的数据依赖，而不能显示控制依赖。

2.2.4 执行次序调整

借助于建立反向图来进行次序调整（如果指令a先于指令b执行并且它们之间存在依赖，那么建立一条从b指向a的边，通过这样的方式即可构建反向图），指令执行次序的调整流程如下（可参考ObProcDepGraph::reorder_for_group()）：

> Step 1: 首先，将非cs-rpc类的根节点压入执行栈中，并删除相关的边。持续该步骤，直到剩余的节点都是cs-rpc类的根节点。
> Step 2: 然后，判断非ups-rpc类的根节点，如果该节点指向了某个ups-rpc类的节点，那么就将该节点压入执行栈中，并删除相关的边。持续该步骤，直到找不到这节点。
> Step 3: 重复执行Step 1 和 Step 2。直到两个步骤都无法从图中删除节点。
> Step 4: 将剩余节点按照拓扑顺序压入执行栈中。
> Step 5: 将执行栈中的节点依次出栈，形成最终的执行次序。

在正确调整执行次序后，需要将连续的ups-rpc合并。合并方法如下（可参考函数ObProcedureOptimizer::group()）：

> Step 1: 按照执行次序，找到第一个ups-rpc，然后继续探查直到遇到一个cs-rpc，记录最后一个ups-rpc
> Step 2: 如果第一个ups-rpc和最后一个ups-rpc相同的，那么不进行成组，若该指令是一个循环指令，那单独成组该指令；如果不相同，那么将这段指令成组在一起。
> Step 3: 分配一个 SpGroupInst，将需要成组的指令按次序插入到 SpGroupInst中。

2.2.5 复杂结构处理

Loop、If以及Case-When等结构在调整次序的过程中将作为一个整体进行调整。这大大限制了指令次序调整的性能优化空间。考虑将这些嵌套结构拆分成多个子结构突破这一性能限制。

Split算法

考虑一个嵌套结构中的指令序列，如果所有的cs-rpc均是用于基线数据读取，可以尝试将整个序列分隔成两个部分，第一个部分进行所有的cs-rpc操作，产生基线数据；第二个部分不包含任何cs-rpc操作，可以被成组执行。这些cs-rpc操作可以在嵌套结构外执行（因为读取基线数据不会改变其它指令的读取内容，不会改变事务的语义）,但需要保证读取的基线数据的准确性，即保证 cs-rpc 读取的变量取值是正确的。需要满足“与cs-rpc存在 true-dependence的指令应该被正确的执行”的要求。

Split具体流程如下：

> 1、 迭代标记所有与cs-rpc存在true dependence的指令。
> 2、 依次扫描 嵌套结构中的指令，如果所有被标记的指令都不需要访问ups（通过cs-rpc间接访问ups的也认为是访问ups），那么认为该嵌套结构是可以切分的；如果不是，那么放弃split。
> 3、分配一个预执行指令序列
> 4、依次扫描 嵌套结构中的指令，如果指令是local processing，那么拷贝一份到预执行指令序列中；如果指令是 cs-rpc，那么从原始的指令序列中删除，并添加到预执行执行序列中。

*注意：预执行指令序列需要在一个新的上下文中执行，所有指令执行完后，仅保留读取到的基线数据，变量修改一律不保存，因为这些产生变量修改的指令会在原始的嵌套结构中被正确执行。当前的变量结果仅用于生成基线数据。

If Split

具体执行流程参考如下(可参考函数ObProcedureOptimizer::ctrl_split())：

> Step 1: 尝试对 then 结构进行split。
> Step 2: 尝试对 else 结构进行 split。
> Step 3: 如果then 与 else 均可以split；那么if结构被分解为，then的预执行序列，else的预执行序列，以及无cs-rpc的if结构。

*注：针对Case-when等结构的split方法是相同的。但当前只实现了针对 if 结构的split流程。

Loop Split

当我们尝试切割loop的时候，首先需要判断loop结构是不是一个简单循环，即：迭代的次数不会受到循环体的影响（**实际上该约束可以进一步放宽，目前为了保证准确性，我们采用了严格要求**）。

简单循环的判别方式如下：

> 循环体中没有指令修改计数变量。

对循环体的split算法与普通的split算法有所不同。考虑a，b两个指令，其中a读取基线数据，并且在指令序列中a在b的前面；虽然在同一个次迭代中，a不会读取到b的写入内容；但是，在相邻的两次迭代中，第二次迭代中的a有可能读取到第一次迭代b产生的结果。因此，我们需要在两次迭代的指令序列上判断循环体是否可以切割，我们需要标记出 后一次迭代中cs-rpc依赖的所有其他指令，当被标记的指令中不包含ups-rpc，那么该循环是可以切割的。

*注：
（1）以上split过程需要在代码中添加；
（2）While和Loop事实上也可以进行split，但需要对应的简单循环判别算法。

2.3 成组执行

成组执行是指向UPS一次发送多个增量修改操作，并且增量修改操作之间存在逻辑关系，一个增量操作的输入可能是另外一个增量操作的输出。

在原始的ups-rpc中，增量操作的输入包括两类：1、基线数据；2、调用变量。在将增量修改计划发送给ups前，ms会在物理计划中填入真实的变量取值和基线数据。然而，成组执行无法采用这样的设计。这包括以下两个原因：

1. 成组执行中，部分变量的取值需要在线计算；在序列化物理计划前，我们无法确知真实的变量取值。
2. 在循环结果中，同一个增量修改计划会被反复调用，每次调用需要的变量和基线数据都不同；因此，也不能将基线数据和变量取值直接保存在物理计划中。

因此，成组执行的实现主要包括以下几方面：

- 物理计划的参数化序列方式；
- 基线数据的管理
- 成组指令的调用协议

2.3.1 物理计划参数化

实现物理计划的参数化主要涉及到update，replace，insert操作的物理算子序列化方式的修改。主要包括：

```
ObMemSSTableScan //保存基线数据
ObExprValues  //保存多个表达式
ObIncScan / ObUpsIncScan  //保存查询的主键信息
ObPostfixExpression  //后缀表达式
```

普通执行模式下，算子序列化时会读取变量的值，并序列化变量的值；但在成组执行模式下，我们将序列化变量名。而且，普通执行模式下，ObMemSSTableScan会从ObValues中读取基线数据，并序列化基线数据；但在成组模式下，我们只序列化一个代表基线数据的id。基线数据的序列化通过基线数据管理器负责。在UPS端，ObMemSSTableScan根据 id 和循环迭代次数从基线数据管理器中读取所需的数据。

判断当前是否在成组执行模式下的方式如下：

在ObPhysicalPlan 中增加一个bool字段区别成组执行模式和普通执行模式。每个算子都保存了 my_phy_plan_ 指针，可以通过该指针判断是否是成组执行模式，以决定序列化的方式。

以上算子在序列化时，会序列化bool字段，代表当前的执行模式。在反序列化时，根据bool字段，决定反序列化的策略。

2.3.2 基线数据管理

基线数据的生成，使用，发送由 ObProcedureStaticDataMgr 管理。本节主要介绍基线数据是如何生成、存储以及被使用的。

保存

所有的基线数据都按照（id，hkey，data）三元组的方式存储。

- **id**是在编译节点生成的，指代某个基线读取指令的唯一标识符。该指令生成的所有基线数据共享同一个id。
- **hkey**是用来区别同一条指令在不同循环迭代下产生的基线数据。hkey根据循环迭代的次数计算生成。对于嵌套循环，会对所有迭代的计算产生一个hkey。
- **data**是保存了基线读取指令某次执行生成的基线数据，采用ObRowStore的方式存储。

生成

实际上，并不是所有的基线查询指令产生的基线数据都是由管理器管理的。基线数据管理器只管理用于成组执行的数据。根据基线数据使用方式的不同，基线查询指令可以分为以下两种类型：

- 用于普通执行：基线数据对应的增量操作指令采用普通执行方式。
- 用于成组执行：基线数据对应的增量操作指令采用成组执行方式。

在编译优化完成后，根据增量操作指令的类型，我们可以确定对应的基线操作指令的类型。不同类型的基线操作，它们的执行方式存在区别。普通基线操作指令执行后，对应的基线读取算子不需要关闭，该算子的结果将会被后续的增量操作算子读取，并正确关闭。而成组基线操作指令执行后，对应的基线读取算子中的数据需要保存到基线数据管理器中，并立刻关闭。

使用

在成组执行过程中，ObMemSSTableScan中保存了基线数据的id。该算子根据id向 ObUpsProcedure 获取对应的基线数据，ObUpsProcedure会进一步根据此时的指令所处的迭代状态确定hkey，根据（id，hkey）从基线数据管理器中读取所需的数据。

2.3.3 成组执行

MS端的流程如下：

1. 设置ObPhysicalPlan的执行状态为 group execution，代表所有算子将采用成组执行策略。
2. 序列化指令集
3. 序列化需要的变量集合
4. 序列化需要的基线数据
5. 发起远程调用，并等待结果
6. 更新变量结果
7. 设置ObPhysicalPlan的指令状态为 normal execution。

UPS端的流程如下：

1. 反序列化指令集
2. 反序列化变量集合
3. 反序列化基线数据
4. 执行所有的指令
5. 向MS返回执行后的变量结果。

变量返回

UPS端在完成成组执行后，需要将被修改的变量取值返回给MS。变量是采用关系的形式保存并返回的。在返回结果集中，每一行的Schema如下：(变量名，变量取值)。MS端遍历结果集，并更新变量的取值。

3 模块接口

3.1 对外接口

缓存模块

缓存模块主要涉及到以下两个类：ObNameCodeMap和 ObProcedureManager。

```
//插入一条存储过程源码
ObNameCodeMap::put_source_code(const ObString &proc_name, const ObS
tring &sour_code);

//删除一条存储过程源码
ObNameCodeMap::del_source_code(const ObString &proc_name);

//根据过程名，获得存储过程源码。范围值为： const ObString *
ObNameCodeMap::get_source_code(const ObString &proc_name);
```

在RS上，ObNameCodeMap 持久化到磁盘中，以保证系统故障后，可以从磁盘恢复建立的存储过程。当客户端创建或者删除存储过程时，RS会将对ObNameCodeMap的修改广播给所有的MS。ObProcedureManager在Mergeserver上管理所有的存储过程缓存，包括源码的缓存和物理计划的生成。

```
//在MS上保存一份存储过程源码
ObProcedureManager::create_procedure(...);

//在MS上删除一份存储过程源码
ObProcedureManager::delete_procedure(...);

//使用context生成一份proc_name的存储过程物理计划。
ObProcedureManager::get_procedure_lazy(const ObString &proc_name,
                                       ObSqlContext &context,
                                       uint64_t& stmt_id);
```

优化模块

该函数的输入为编译后的存储过程物理计划，该函数会分析存储过程中，每个操作之间的依赖关系，对操作的执行次序进行合理的调整。函数返回后，proc中保存了优化后的执行计划。

```
ObProceduerOptimizer::optimize(ObProcedure &proc);
```

物理计划模块

物理计划的表示分为三层： SpProcedure, ObProcedure, ObUpsProcedure。

- SpProcedure中保存了存储过程的基本信息，分配的指令操作，原始的执行次序，维护的变量集合和基线数据集合，以及使用存储过程必要的对象。
- ObProcedure继承了SpProcedure，它是存储过程在Mergeserver端的表示方式，其中包含了在Mergeserver端执行所需的一些对象，例如对其他节点进行rpc访问的句柄等。
- ObUpsProcedure继承了SpProcedure,它是存储过程在Updateserver端的表示方式，其中包含了在Updateserver端执行所需的一些对象。

```
//在生成物理计划时，生成一条存储过程指令，用来封装一些操作的逻辑。生成的指令将会添加到 mul_inst 结构中。
ObProcedure::create_inst(SpMultiInsts *mul_inst);

// 该函数决定存储过程中rpc的执行模式，成组或者非成组。
ObProcedure::deter_exec_mode();
```

执行模块

```
class SpInstExecStrategy
{
  public:
    //执行一条指令
    virtual int execute_inst(SpInst *inst) = 0; //to provide the si
mple routine
  private:
    //执行一条赋值操作
    virtual int execute_expr(SpExprInst *inst) = 0;
    //执行一条读取基线操作
    virtual int execute_rd_base(SpRdBaseInst *inst) = 0;
    //执行一条修改增量操作
    virtual int execute_wr_delta(SpRwDeltaInst *inst) = 0;
    //执行一条读取增量操作
    virtual int execute_rd_delta(SpRwDeltaIntoVarInst *inst) = 0;
    //执行一条读写全量操作
    virtual int execute_rw_all(SpRwCompInst *inst) = 0;
    //执行一条成组操作
    virtual int execute_group(SpGroupInsts *inst) = 0;
    ...
};
```

3.2 内部接口

优化模块

对loop结构进行切分，将一个loop切分成若干个小的loop。

```
ObProcedure::loop_split(SpLoop *loop_inst, SpInstList &inst_list);
```

对if结构进行切分，将一些基线数据操作从if结构中分析。参见上文对if的处理方式。

```
ObProcedure::ctrl_split(SpIfCtrlInsts *if_inst, SpInstList &inst_li
st);
```

expand_list是原始的操作执行次序，下述函数会将连续的ups访问合并，并加入到proc中。

```
ObProcedure::group(ObProcedure &proc, SpInstList &expand_list);
```

物理计划模块

- 分配一个基线数据的存储对象。该基线数据通过(sdata_id, hkey)唯一标示。
- sdata_id 是对应基线操作的唯一标示符。
- hkey 是对应循环的迭代次数的hash值。它用于区分同一条基线操作，在循环迭代的不同次数下产生的基线数据。

```
SpProcedure::store_static_data(int64_t sdata_id, int64_t hkey, ObRo
wStore *&p_row_store);
```

指令执行过程中，用于获取基线数据。

```
SpProcedure::get_static_data_by_id(int64_t sdata_id, ObRowStore
*&p_row_store);
```

4 使用限制条件和注意事项

- 在增加一条新的指令时，需要至少在2处进行修改。首先，在ob_sp_procedure.h中要增加指令的声明和定义。其次，需要在ob_procedure.h中增加在MS上执行指令的策略。如果指令仅在ms上执行，务必设置指令的类型为s_rpc。这样可以防止优化器将操作成组到ups的rpc中。（如果只在MS上执行，可以不定义序列化和反序列化函数）。
- 如果，新增的指令可以被成组优化到ups上执行。那么首先需要正确设置指令的类型。然后，需要在ob_ups_procedure.h中增加在UPS上的执行策略。此外，需要正确的定义指令的序列化和反序列化操作。
- 增加一条新的指令时，务必确保正确地返回指令的读写变量集合。这是优化器正确工作的基本前提。