Clickhouse源码导读

Clickhouse源码导读

ClickHouse 是一个由俄罗斯搜索巨头Yandex开源的分布式列存储OLAP数据库。最突出的特点有特点就是一个快字。为了搞懂Clickhouse为什么快，我粗略的看了看Clickhouse的源码，总结一份导读指南，方便他人探索。

基本流程

先从github下载源码看看，本文内容基于 `v18.14.17-stable` 版本。Clickhouse整个项目的结构还是很清晰的，入口的 main函数在 `dbms/programs/main.cpp`。主程序会根据指令分发到 `dbms/programs` 目录下的程序中处理。我们主要关注 `clickhouse server`，所以直接来到 `dbms/programs/server/Server.cpp`，一路走下来解析参数配置，初始化server，再启动服务监听端口。

clickhouse 使用的是 `poco` 这个网络库来处理网络请求，每个client连接的处理逻辑在 `dbms/programs/server/TCPHandler.cpp`的 `TCPHandler::runImpl()`方法里面。除去握手，初始化上下文，异常处理和数据统计的代码，主要的业务可以抽象成:

```
// dbms/programs/server/TCPHandler.cpp

TCPHandler.runImpl()
{
    ...
    while(1) {
        receivePacket()

        /// Processing Query
        state.io = executeQuery(state.query, query_context, false, state.stage);

        if (state.io.out)
            state.need_receive_data_for_insert = true;

        if (state.need_receive_data_for_insert)
            processInsertQuery(global_settings);
        else
            processOrdinaryQuery();

    ...
}
```

client发送的sql在 `executeQuery` 函数处理，`processInsertQuery` 和 `processOrdinaryQuery` 负责将结果返回给client。

`executeQuery` 函数的实现在`dbms/src/Interpreters/executeQuery.cpp`, 主要逻辑简化如下：

```
dbms/src/Interpreters/executeQuery.cpp

executeQueryImpl(...)
{
        ...
        ast = parseQuery(parser, begin, end, "", max_query_size);
        ...

        auto interpreter = InterpreterFactory::get(ast, context, stage);
    res = interpreter->execute();

    ...
}
```

类比 mysql 的处理流程，先解析sql语句生成抽象语法树(AST)，`InterpreterFactory`工厂类根据AST生成 执行器`Interpreter`类实例来执行。

`interpreter->execute()` 返回到结果 `res` 是一个 `BlockIO`, `BlockIO` 其实就是一个 `BlockInputStream`和`BlockOutputStream`的一个封装。这里就引出了 Clickhouse 里面的一些重要概念。

Block和Block Stream

Clickhouse是面向OLAP的列存储数据库系统，数据的存储和读写都是批量处理的。根据文档, 一个`Block`代表着一批的数据，内部是用列来划分的，也就是一个`(IColumn, IDataType, column name)`三元组的集合。Clickhouse 的数据处理都是以Block为单位的，而Clickhouse的高性能也得益于能够使用向量化技术一次批量的处理一个Block里同类型的数据。

而 `Block Stream`就是一个个 `Block` 组成的数据流。`Block Stream`分为两种，负责数据写入的实现 `IBlockOutputStream`接口，通过`write`方法写入一个`Block`。负责数据读取的实现 `IBlockInputStream`接口，通过`read`方法读取一个`Block`。

```
// dbms/src/DataStreams/IBlockOutputStream.h
class IBlockOutputStream : private boost::noncopyable
{
public:
        ...
```

```
    /** Write block.
      */
    virtual void write(const Block & block) = 0;
    ...
}

// dbms/src/DataStreams/IBlockInputStream.h
class IBlockInputStream : private boost::noncopyable
{
public:
    IBlockInputStream() {}
        ...

    /** Read next block.
      * If there are no more blocks, return an empty block (for which operator `bool` returns false).
      * NOTE: Only one thread can read from one instance of IBlockInputStream simultaneously.
      * This also applies for readPrefix, readSuffix.
      */
    virtual Block read() = 0;
    ...
}
```

不同的Stream可以组合起来完成数据的转化。比如最初的 `IBlockInputStream`外层套一个 `FilterBlockInputStream`过滤掉不符合条件的数据，再接一个`AggregatingBlockInputStream`将原始数据聚合给下一个 `IBlockInputStream`。其实`Block Stream`类似`TiDB`里面的算子, 或者类比`Python`的迭代器，最外层不断调用 `read`/`write`方法驱动整个计算的进行。

下面我们追踪最简单的数据写入Insert过程和查询Select过程讲讲相关的代码。

写入

让我们回到`InterpreterFactory`, Insert语句对应`InterpreterInsertQuery`这个执行器。

```
// dbms/src/Interpreters/InterpreterFactory.cpp

InterpreterFactory::get()
{
        if (typeid_cast<ASTSelectQuery *>(query.get()))
    {
        /// This is internal part of ASTSelectWithUnionQuery.
        /// Even if there is SELECT without union, it is represented by ASTSelectWithUnionQuery with single ASTSelectQuery as a child.
        return std::make_unique<InterpreterSelectQuery>(query, context, Names{}, stage);
    }
    else if (typeid_cast<ASTSelectWithUnionQuery *>(query.get()))
    {
        ProfileEvents::increment(ProfileEvents::SelectQuery);
        return std::make_unique<InterpreterSelectWithUnionQuery>(query, context, Names{}, stage);
    }
    else if (typeid_cast<ASTInsertQuery *>(query.get()))
    {
        ProfileEvents::increment(ProfileEvents::InsertQuery);
        /// readonly is checked inside InterpreterInsertQuery
        bool allow_materialized = static_cast<bool>(context.getSettingsRef().insert_allow_materialized_columns);
        return std::make_unique<InterpreterInsertQuery>(query, context, allow_materialized);
    }

    ....... // 分发
}

// dbms/src/Interpreters/InterpreterInsertQuery.cpp

StoragePtr InterpreterInsertQuery::getTable(const ASTInsertQuery & query)
{
    if (query.table_function)
    {
        auto table_function = typeid_cast<const ASTFunction *>(query.table_function.get());
        const auto & factory = TableFunctionFactory::instance();
        return factory.get(table_function->name, context)->execute(query.table_function, context);
    }

    /// Into what table to write.
    return context.getTable(query.database, query.table);
}

BlockIO InterpreterInsertQuery::execute()
{
        ...
        StoragePtr table = getTable(query);
        ...
        out = std::make_shared<PushingToViewsBlockOutputStream>(query.database, query.table, table, context, query_ptr, query.no_destination);
        ...
}
```

`PushingToViewsBlockOutputStream`的会先写入更低层的`BlockOutputStream`, 然后查看一下写入的数据源是否有 `MaterialView`,若有，调用`process`方法用`MaterializingBlockInputStream`往相关的`MaterialView`写入数据。而`PushingToViewsBlockOutputStream`更低层的`BlockOutputStream`是 `getTable`方法获取的`IStorage`对象提供的。

```
// dbms/src/Storages/IStorage.h

class IStorage : public std::enable_shared_from_this<IStorage>, private boost::noncopyable, public ITableDeclaration
{
        ....
        virtual BlockOutputStreamPtr write(
        const ASTPtr & /*query*/,
        const Settings & /*settings*/)
    {
        throw Exception("Method write is not supported by storage " + getName(), ErrorCodes::NOT_IMPLEMENTED);
    }
    ....
    virtual BlockInputStreams read(
        const Names & /*column_names*/,
        const SelectQueryInfo & /*query_info*/,
        const Context & /*context*/,
        QueryProcessingStage::Enum /*processed_stage*/,
        size_t /*max_block_size*/,
        unsigned /*num_streams*/)
    {
        throw Exception("Method read is not supported by storage " + getName(), ErrorCodes::NOT_IMPLEMENTED);
    }
    ....

}
```

`IStorage` 是Clickhouse存储引擎的接口，我们直接看最关键的 `MergeTree`引擎的实现

```
// dbms/src/Storages/StorageMergeTree.cpp

BlockOutputStreamPtr StorageMergeTree::write(const ASTPtr & /*query*/, const Settings & /*settings*/)
{
    return std::make_shared<MergeTreeBlockOutputStream>(*this);
}
dbms/src/Storages/MergeTree/MergeTreeBlockOutputStream.cpp

void MergeTreeBlockOutputStream::write(const Block & block)
{
    storage.data.delayInsertOrThrowIfNeeded();

    auto part_blocks = storage.writer.splitBlockIntoParts(block);
    for (auto & current_block : part_blocks)
    {
        Stopwatch watch;

        MergeTreeData::MutableDataPartPtr part = storage.writer.writeTempPart(current_block);
        storage.data.renameTempPartAndAdd(part, &storage.increment);

        PartLog::addNewPart(storage.context, part, watch.elapsed());

        /// Initiate async merge - it will be done if it's good time for merge and if there are space in 'background_pool'.
        storage.background_task_handle->wake();
    }
}
```

追踪到最底层的 `MergeTreeBlockOutputStream` 我们会发现最终数据由`MergeTreeDataWriter`(`dbms/src/Storages/MergeTree/MergeTreeDataWriter.h`)写入，而`MergeTreeDataWriter`是`MergeTreeData`(`dbms/src/Storages/MergeTree/MergeTreeData.h`)的封装，`MergeTree`的数据都由`MergeTreeData`对象管理。存储的格式可以看看这篇文章，后面可能会另写文再说说。

`MergeTreeBlockOutputStream`一次写入一个`Block`,然后会唤醒后台任务将一个个小的`Block`合并。这应该就是`MergeTree`命名的由来了。由此我们可知，Clickhouse应尽可能的批量写入数据而不是一条一条的写。

最后再回来往上走，看看是在哪里调用最外层的 `write`方法写入的。

```
void TCPHandler::processInsertQuery(const Settings & global_settings)
{
    /** Made above the rest of the lines, so that in case of `writePrefix` function throws an exception,
      *  client receive exception before sending data.
      */
    state.io.out->writePrefix();

    /// Send block to the client - table structure.
    Block block = state.io.out->getHeader();
    sendData(block);

    readData(global_settings);    <--- here
    state.io.out->writeSuffix();
    state.io.onFinish();
}


void TCPHandler::readData(const Settings & global_settings)
{
        ...
        receiveData()
        ...
}

bool TCPHandler::receiveData()
{
        ...
```

```
    /// Read one block from the network and write it down
    Block block = state.block_in->read();

    ....
        if (block)
            state.io.out->write(block);
        return true;
    ....
}
```

读取

读取最外层`BlockStream`的地方就在`processOrdinaryQuery`。

```
// dbms/programs/server/TCPHandler.cpp

void TCPHandler::processOrdinaryQuery()

{
        ....
        AsynchronousBlockInputStream async_in(state.io.in);
        ...
        block = async_in.read();
        ...
        sendData(block);

}
```

在前面的`InterpreterFactory::get`方法可以看到Select语句会在初始化`InterpreterSelectQuery`, 于是我们来到`InterpreterSelectQuery.cpp`

```
dbms/src/Interpreters/InterpreterSelectQuery.cpp

void InterpreterSelectQuery::executeImpl(....)
{
        auto optimize_prewhere = [&](auto & merge_tree)
        {
            SelectQueryInfo query_info;
            query_info.query = query_ptr;
            query_info.sets = query_analyzer->getPreparedSets();

            /// Try transferring some condition from WHERE to PREWHERE if enabled and viable
            if (settings.optimize_move_to_prewhere && query.where_expression && !query.prewhere_expression && !query.final())
                MergeTreeWhereOptimizer{query_info, context, merge_tree.getData(), query_analyzer->getRequiredSourceColumns(), log};
        };

        AnalysisResult expressions;

        expressions = analyzeExpressions(from_stage, false);

        /** Read the data from Storage. from_stage - to what stage the request was completed in Storage. */
    executeFetchColumns(from_stage, pipeline, expressions.prewhere_info, expressions.columns_to_remove_after_prewhere);
    ....
    if (expressions.has_where)
        executeWhere(pipeline, expressions.before_where, expressions.remove_where_filter);

    if (expressions.need_aggregate)
        executeAggregation(pipeline, expressions.before_aggregation, aggregate_overflow_row, aggregate_final);
    else
    {
        executeExpression(pipeline, expressions.before_order_and_select);
        executeDistinct(pipeline, true, expressions.selected_columns);
    }

    if (!expressions.second_stage && !expressions.need_aggregate && !expressions.has_having)
    {
        if (expressions.has_order_by)
            executeOrder(pipeline);

        if (expressions.has_order_by && query.limit_length)
            executeDistinct(pipeline, false, expressions.selected_columns);

        if (query.limit_length)
            executePreLimit(pipeline);
     .....
}
```

可以看到，最底层的`IBlockInputStream`通过`executeFetchColumns`方法从`storage`里面读取出来。

```
void InterpreterSelectQuery::executeFetchColumns(...)
{
        ...
        pipeline.streams = storage->read(required_columns, query_info, context, processing_stage, max_block_size, max_streams);

        if (pipeline.streams.empty())
        {
            pipeline.streams.emplace_back(std::make_shared<NullBlockInputStream>(storage->getSampleBlockForColumns(required_columns)));

            if (query_info.prewhere_info)
                pipeline.streams.back() = std::make_shared<FilterBlockInputStream>(
                        pipeline.streams.back(), prewhere_info->prewhere_actions,
```

```
                    prewhere_info->prewhere_column_name, prewhere_info->remove_prewhere_column
                );
        }
        ....
}
```

跟写入过程类似, StorageMergeTree调用封装了`MergeTreeData`的`MergeTreeDataSelectExecutor`的`read`方法从存储里面获取数据。

```
// dbms/programs/src/Storages/StorageMergeTree.cpp

BlockInputStreams StorageMergeTree::read(...)
{
    return reader.read(column_names, query_info, context, max_block_size, num_streams, 0);
}
```

回到`InterpreterSelectQuery`, `executeFetchColumns`方法取出数据后会调用各种`executeXXX`方法再给套上各种数据处理的`BlockStream`。

```
...
void InterpreterSelectQuery::executeWhere(Pipeline & pipeline, const ExpressionActionsPtr & expression, bool remove_fiter)
{
    pipeline.transform([&](auto & stream)
    {
        stream = std::make_shared<FilterBlockInputStream>(stream, expression, query.where_expression->getColumnName(), remove_fiter);
    });
}

void InterpreterSelectQuery::executeAggregation(Pipeline & pipeline, const ExpressionActionsPtr & expression, bool overflow_row, bool final)
{
        ...
}
```

高性能

Clickhouse文档里面提到了Clickhouse高性能的秘密是vectorized query execution 和 runtime code generation，即向量化SIMD的运用和JIT。这两点是怎么体现的呢?

JIT

其实我们只要在代码里面搜`USE_EMBEDDED_COMPILER`这个编译宏就可以找出所有JIT相关的代码。最主要的地方在`dbms/src/Interpreters/ExpressionJIT.cpp`里面。

若是开启了`USE_EMBEDDED_COMPILER`， `compileFunctions`函数会将复杂的表达式即时编译成机器码执行, Clickhouse会缓存编译结果，由此提高性能。

SIMD

SIMD (Single Instruction Multiple Data) 是一种采用一个控制器来控制多个处理器，同时对一组数据（又称"数据向量"）中的每一个分别执行相同的操作从而实现空间上的并行性的技术。简单来说就是一条指令处理多个数据，由此来提升性能。

SIMD技术需要CPU支持SIMD的指令集，如MMX、SSE、AVX。

Clickhouse使用的是SSE2，我们可以在代码里面搜`__SSE2__`这个编译宏找出所有SIMD相关的代码。Clickhouse在许多地方比如过滤，压缩，字符串处理函数等都有用到`__SSE2__`。比较多的地方还是在过滤，毕竟是最常用的场景。