# 背景

mysql是关系数据库,但是说白了是一个软件。在我们复杂的软件世界中，大概分为那么两种软件，一种是CPU密集型，一种是I/O密集型。有这个思想很重要，比如面试官经常问你：

1、redis为什么是单线程？

2、大数据word count用mapduce好吗？

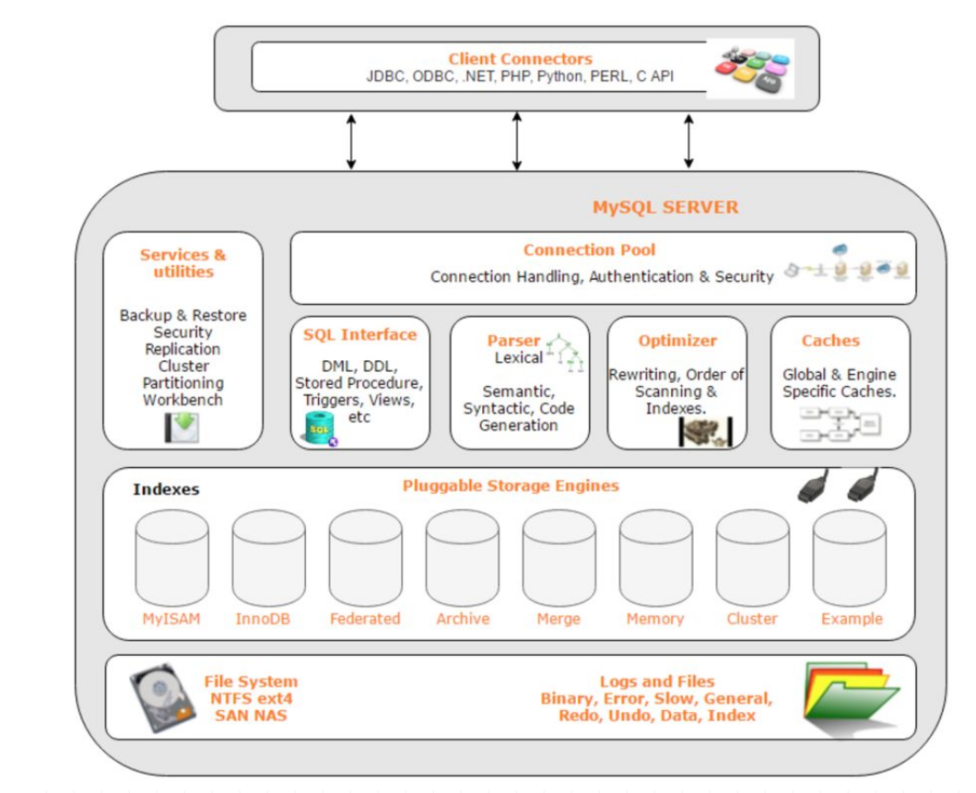
这类问题，首先你就是要区分是CPU密集型还是I/O密集型。而作为一个数据库，显然是一个I/O密集型的软件。如何设计一个I/O密集型软件呢?

1. 你需要接收来自客户端的连接吧?
2. 是不是可以把一些经常访问的数据缓存起来?
3. SQL语句如何被我们要设计的系统识别呢?
4. 数据以何种方式在我们的软件中进行组织?

…

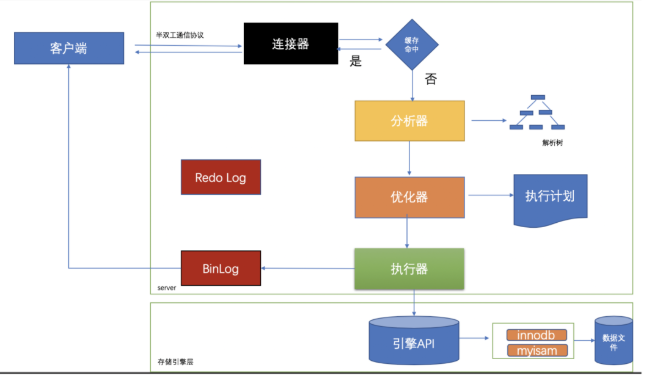
等等诸多问题。比如说数据结构redis中如何组织，答案是用了各种巧妙的数据结构，包括：双端链表、调表等。为什么redis单线程，因为它的瓶颈在I/O又不在计算，所以单线程，这个对应它的连接设计。而mysql如何设计呢?

注：区分该软件是CPU还是IO密集型才是关键，Redis采用多线程也是可以的。



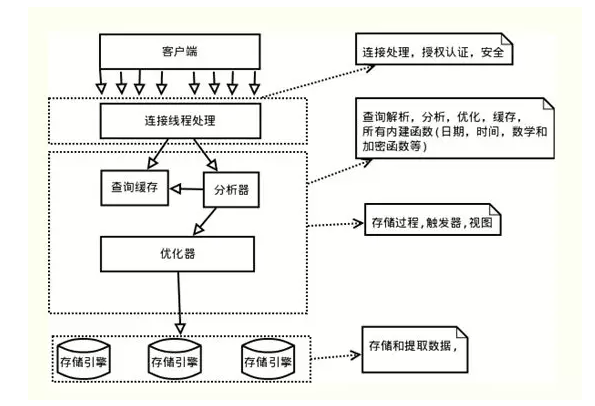
# Mysql基本架构

简单来说 MySQL主要分为Server层和存储引擎层：



1. **MySQL客户端和服务器通讯**

客户端按照MySQL通信协议将SQL发送到服务端，SQL到达服务端后，服务端会单起一个线程执行SQL。MySQL客户端和服务器之间的通讯协议是“半双工”的。  
 **2、查询状态**  
 对于MySQL连接，任何时刻都有一个状态，该状态表示了MySQL当前正在做什么。使用show full processlist命令查看当前状态。在一个查询生命周期中，状态会变化很多次，下面是这些状态的解释：  
 1）sleep：线程正在等待客户端发送新的请求；  
 2）query：线程正在执行查询或者正在将结果发送给客户端；  
 3）locked：在MySQL服务器层，该线程正在等待表锁。在存储引擎级别实现的锁，例如InnoDB的行锁，并不会体现在线程状态中。对于MyISAM来说这是一个比较典型的状态；  
 4）analyzing and statistics：线程正在收集存储引擎的统计信息，并生成查询的执行计划；  
 5）copying to tmp table：线程在执行查询，并且将其结果集复制到一个临时表中，这种状态一般要么是做group by操作，要么是文件排序操作，或者union操作。如果这个状态后面还有on disk标记，那表示MySQL正在将一个内存临时表放到磁盘上；  
 6）sorting result：线程正在对结果集进行排序；  
 7）sending data：线程可能在多个状态间传送数据，或者在生成结果集，或者在想客户端返回数据。  
 **3、查询缓存**  
 MySQL的缓存主要的作用是为了提升查询的效率，缓存以key和value的哈希表形式存储，key是具体的sql语句，value是结果的集合。如果无法命中缓存,就继续走到分析器的的一步,如果命中缓存就直接返回给客户端 。  
 如果使用查询缓存，在进行读写操作时会带来额外的资源消耗，如果在一个写多读少的环境中，缓存会频繁的新增和失效。MySQL8.0版本开始取消查询缓存。  
 **4、查询优化处理** 查询的生命周期的下一步是将一个SQL转换成一个执行计划，MySQL在依照这个执行计划和存储引擎进行交互。这包含多个子阶段：解析SQL、预处理、优化SQL执行计划。这个过程中任何错误都可能终止查询。  
 1）语法解析器和预处理：首先MySQL通过关键字将SQL语句进行解析，并生成一颗对应的“解析树”。MySQL解析器将使用mysql语法规则验证和解析查询；预处理器则根据一些MySQL规则进一步检查解析数是否合法。  
 2）查询优化器：当语法树被认为是合法的了，并且由优化器将其转化成执行计划。一条查询可以有很多种执行方式，最后都返回相同的结果。优化器的作用就是找到这其中最好的执行计划。  
 3）执行计划：MySQL不会生成查询字节码来执行查询，MySQL生成查询的一棵指令树，然后通过存储引擎执行完成这棵指令树并返回结果。最终的执行计划包含了重构查询的全部信息。  
 **5、查询执行引擎**  
 在解析和优化阶段，MySQL将生成查询对应的执行计划，MySQL根据优化器生成的执行计划，调用存储引擎的API来执行查询。  
 **6、返回结果给客户端** 了解select \* from tb\_article where author\_id=20 and title='b';性能和优化策略，一般采用explain命令进行分析。



## 客户端连接层

客户端连接层：处理客户端连接、授权认证等。

## Server层

Server层：主要包括连接器、查询缓存、分析器、优化器、执行器等，所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图，函数等，还有一个通用的日志模块binglog日志模块。

**具体如下：**

连接器：身份认证和权限相关(登录MySQL的时候)。

查询缓存：执行查询语句的时候，会先查询缓存（MySQL 8.0 版本后移除，因为这个功能不太实用）。

分析器：没有命中缓存的话，SQL语句就会经过分析器，分析器就是要先看你的SQL语句要干嘛，再检查你的SQL语句语法是否正确。

优化器：按照 MySQL 认为最优的方案去执行。

执行器：执行语句，然后从存储引擎返回数据。

## 存储引擎

存储引擎：主要负责数据的存储和读取，采用可以替换的插件式架构，支持 InnoDB、MyISAM、Memory等多个存储引擎，其中InnoDB引擎有自有的日志模块 redolog 模块。现在最常用的存储引擎是InnoDB，它从MySQL 5.5.5版本开始就被当做默认存储引擎了。

# Server组件

## 连接器

连接器主要负责将mysql客户端和服务端建立连接，连接成功后，会获取当前连接用户的权限。这里获取到的权限对整个连接都有效，一旦连接成功后，如果使用管理员账号对该用户更改权限，当前连接中的拥有的权限保持不变，只有等到下次重新连接才会更新权限。

客户端按照MySQL通信协议将SQL发送到服务端，SQL到达服务端后，服务端会单起一个线程执行SQL。MySQL客户端和服务器之间的通讯协议是**“半双工”**的。

注：

**一般用程序连接数据库时,用长连接,避免由于短连接对于频繁创建、销毁带来的性能损耗。但是长连接过多又会导致mysql消耗内存很快**。

**查询状态**

对于MySQL连接，任何时刻都有一个状态，该状态表示了MySQL当前正在做什么。使用show full processlist命令查看当前状态。在一个查询生命周期中，状态会变化很多次，下面是这些状态的解释：

1、sleep：线程正在等待客户端发送新的请求；

2、query：线程正在执行查询或者正在将结果发送给客户端；

3、locked：在MySQL服务器层，该线程正在等待表锁。在存储引擎级别实现的锁，例如InnoDB的行锁，并不会体现在线程状态中。对于MyISAM来说这是一个比较典型的状态；

4、analyzing and statistics：线程正在收集存储引擎的统计信息，并生成查询的执行计划；

5、copying to tmp table：线程在执行查询，并且将其结果集复制到一个临时表中，这种状态一般要么是做group by操作，要么是文件排序操作，或者union操作。如果这个状态后面还有on disk标记，那表示MySQL正在将一个内存临时表放到磁盘上；

6、sorting result：线程正在对结果集进行排序；

7、sending data：线程可能在多个状态间传送数据，或者在生成结果集，或者在想客户端返回数据。

## 查询缓存

连接成功后，即开始要正式执行select语句了，但是在执行查询之前，mysql 会去看下有没有该条语句的缓存内容，如果有缓存直接从缓存中读取并返回数据，不再执行后面的步骤了，结束查询操作。

如果没有缓存则继续往后执行，并将执行结果和语句保存在缓存中。

注意：在mysql8后已经没有查询缓存这个功能了，因为**这个缓存非常容易被清空掉，命中率比较低**。只要对表有一个更新，这个表上的所有缓存就会被清空，因此你刚缓存下来的内容，还没来得及用就被另一个更新给清空了。

MySQL的缓存主要的作用是为了提升查询的效率，缓存以key和value的哈希表形式存储，key是具体的sql语句，value是结果的集合。如果无法命中缓存,就继续走到分析器的的一步,如果命中缓存就直接返回给客户端 。

如果使用查询缓存，在进行读写操作时会带来额外的资源消耗，如果在一个写多读少的环境中，缓存会频繁的新增和失效。MySQL8.0版本开始取消查询缓存。

## 分析器

既然没有查到缓存，就需要开始执行sql语句了，在执行之前肯定需要先对sql语句进行解析。分析器主要对sql语句进行语法和语义分析，检查单词是否拼写错误，还有检查要查询的表或字段是否存在。

1、词法分析，一条SQL语句有多个字符串组成，首先要提取关键字，比如 select，提出查询的表，提出字段名，提出查询条件等等。做完这些操作后，就会进入第二步。

2、语法分析，主要就是判断你输入的sql是否正确，是否符合MySQL的语法。

完成这2步之后，MySQL就准备开始执行了，但是如何执行，怎么执行是最好的结果呢？这个时候就需要优化器上场了。

如果分析器检测出有错误就会返回类似 "You have an error in your sql" 这样的错误信息，并结束查询操作。

## 优化器

通过分析器之后，mysql就算是理解了你要执行的操作了。通常对于同一个 sql语句，mysql内部可能存在多种执行方案，比如存在多个索引时，该选择哪个索引，多个表关联查询时，怎么确认各个表的连接顺序。

这些方案的执行结果都一样，但是执行效率不一样，所以mysql在执行之前需要尝试找出一个最优的方案来，这就是优化器的主要工作，但是mysql也会有选择错误方案的时候。

**查询优化处理**

查询的生命周期的下一步是将一个SQL转换成一个执行计划，MySQL在依照这个执行计划和存储引擎进行交互。这包含多个子阶段：解析SQL、预处理、优化SQL执行计划。这个过程中任何错误都可能终止查询。

1、语法解析器和预处理：首先MySQL通过关键字将SQL语句进行解析，并生成一颗对应的“解析树”。MySQL解析器将使用mysql语法规则验证和解析查询；预处理器则根据一些MySQL规则进一步检查解析数是否合法。

2、查询优化器：当语法树被认为是合法的了，并且由优化器将其转化成执行计划。一条查询可以有很多种执行方式，最后都返回相同的结果。优化器的作用就是找到这其中最好的执行计划。

3、执行计划：MySQL不会生成查询字节码来执行查询，MySQL生成查询的一棵指令树，然后通过存储引擎执行完成这棵指令树并返回结果。最终的执行计划包含了重构查询的全部信息。

## 执行器

在解析和优化阶段，MySQL将生成查询对应的执行计划，MySQL根据优化器生成的执行计划，调用存储引擎的API来执行查询。

经过优化器选定了一个方案后，执行器就按照选定的方案执行 sql 语句。前面我们有讲过，在连接器中会读取当前用户的权限，连接器中只是获取权限而已，并没有对权限进行判断和校验。

所以在执行器中，在执行语句之前会判断权限，如果没有对应的权限则会直接返回并提示没有相关权限。

这里你可能会问，为什么不在连接器中就直接判断权限呢，这里我觉得可能是因为mysql要查询的表并不一定仅限于sql语句中字面上的那些表，有的时候可能需要经过分析器和优化器之后才能确定到底要怎么执行，所以权限校验放在执行器中是有道理的。

注意如果是在前面的查询缓存中查到缓存之后，也会在返回结果前做权限校验的。

权限校验通过之后，就继续打开表，调用存储引擎提供的接口去查询并返回结果集数据。

到这里，一条查询sql语句就执行结束了。

# 词法解析

## Lex

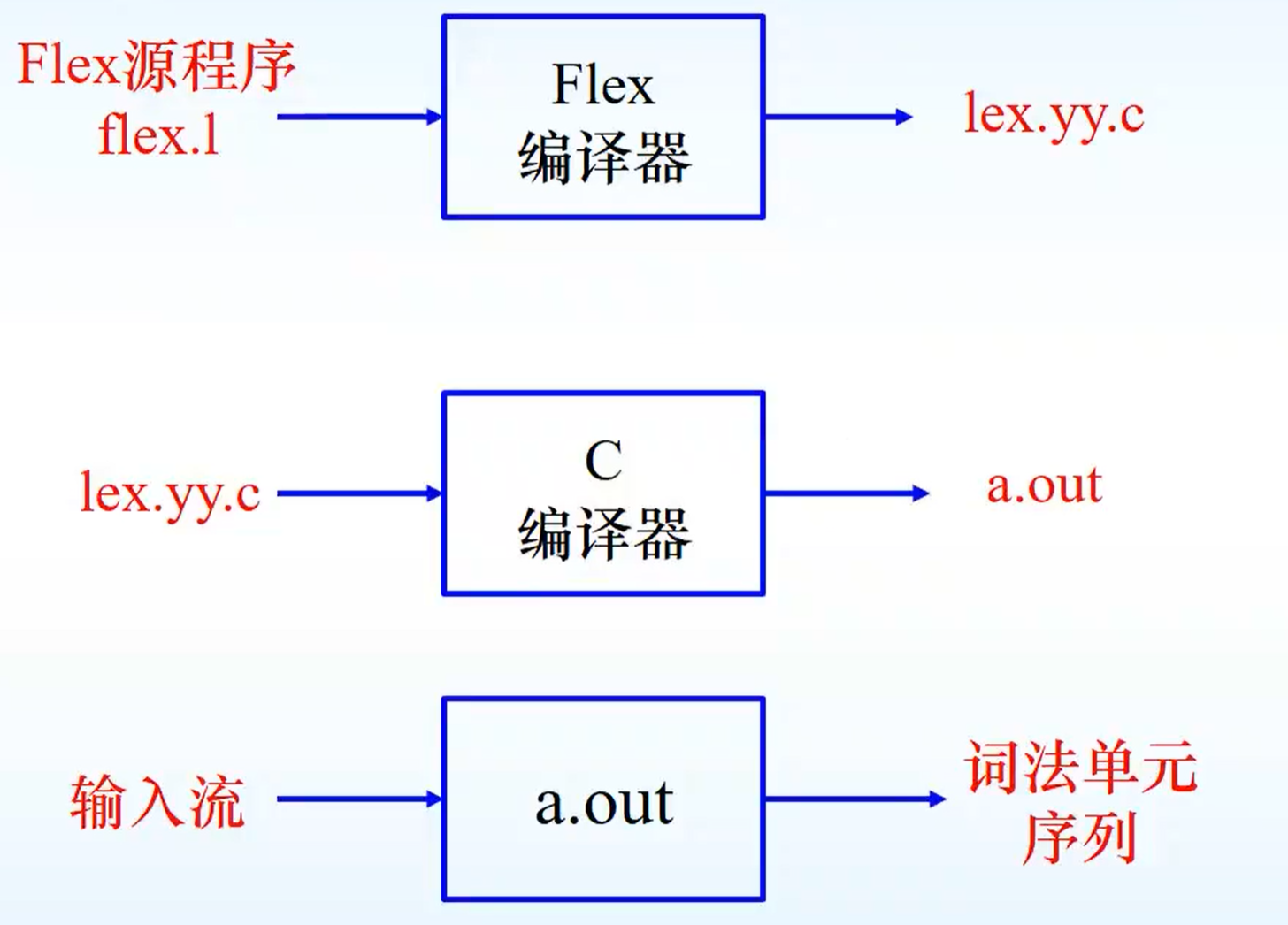
## flex

### 概述

Flex是一个用于生成词法分析器的工具，支持通过正则表达式来对词法单元的模式进行匹配。

Flex本身也称为Flex编译器，它将输入的模式转换成一个状态转换图，并生成相应的代码（核心是yylex()函数），存放到文件lex.yy.c中。

Flex的输入（也称为Flex源程序）是后缀为.l的文件，该文件按照Flex语言进行编写，用于描述将要生成的词法分析器。



.l文件经过Flex编译器，默认会生成名为lex.yy.c的C文件，可以使用-o来重新指定生成的文件名。

同样地，C编译器编译lex.yy.c，生成可执行文件a.out（默认名称，也可以使用-o自定义输出名称）。

**Flex定义的一些常用变量：**

yytext：词法分析程序当前识别到的词素，与转换规则部分中的某个模式相匹配。

yylength：词法分析程序当前识别到的词素的长度。

yylval：是在bison中定义的联合类型变量（union），因为Flex生成的词法分析程序yylex()需要向bison生成的语法分析器返回识别到的词法单元，所以需要使用yylval来保存词法单元的属性值。

注：在Flex中定义yylval是非常必要的，因为Flex只是负责词法解析，即将输入解析为一个个token，但是token之间的关联性需要借助bison的规则去建立，这里就需要将Flex处理后的值传递给bison，yylval就是这个作用。

### 源程序结构

Flex源文件.l的结构如下：

#### 声明部分

声明部分包含：

1、%option yylineno noyywrap

选项设置，使用%option进行指定，例如yylineno会告诉Flex生成一个名为yylineno的整型变量来保存当前行号，noyywrap告诉Flex不要使用yywra()函数。

1. 注释

%{

/\*\*注释/

#include “”

%}

%{和%}之间的内容会原样复制到lex.yy.c文件中。

1. EXP([Ee][-+]?[0-9]+)

名称声明，可以简化转换规则部分的模式定义，格式为：

名称 定义

定义使用正则表达式

#### 转换规则

Flex的规则没有bison的那么强大，一般都是Flex做词法解析，然后bison负责借助自己的规则定义，实现具体动作。

#### 辅助函数

### 正则表达式

任何flex词法分析器的模式都使用了强大的正则表达式语言。正则表达式使用元语言（metalanguage）来描述你想匹配的模式。Flex的正则表达式语言本质上是扩展的POSIX正则表达式。元语言使用标准的文本字符，一部分代表它们自身而另外一部分则代表模式。所有下文没有列出的字符，包括字母和数字，将匹配自身。

.：匹配除了换行符“\n”以外的任何单个字符

\*：匹配前面表达式的0或多个拷贝

[]：匹配括号中任意字符的字符类，如果第一个字符是“^”，则匹配除括号中的字符以外的任意字符。“-”表示一个字符范围，例如“[0-9]”和“[0123456789]”的含义相同。除了以“\”开始的转义序列，元字符在方括号内没有特殊含义。

^：作为正则表达式的行首匹配行的开头，也用于方括号中的否定。

$：作为正则表达式的行尾匹配行的结尾。

\：用于转义元字符，也作为常用的C转义序列的一部分，例如“\n”表示换行，“\*”表示非元字符的星号。

+：匹配前面的正则表达式1或多次出现。

?：匹配前面的正则表达式0或多次出现。

|：匹配前面的正则表达式或随后的正则表达式。

“...”：引号中的每个字符解释为字面意义，除C转义序列外元字符会失去其特殊含义。

()：将一系列正则表达式组成一个新的正则表达式，例如(01)，表示字符序列01。

{}：当括号中包含一个或两个数字时，指示前面的模式允许被匹配多少次，例如A{1,3}表示匹配字母A1-3次。

### 二义性

大多数flex程序都具有二义性，相同的输入可能被多种不同的模式匹配。Flex通过两个简单的规则来解决它：

1. 词法分析器匹配输入时匹配**尽可能长**的字符串
2. 如果可以匹配多个模式，则匹配在程序中**更早出现**的模式

例如：

“+” {return ADD;}

“=” {return ASSIGN;}

“+=” {return ASSIGNAND;}

“else” {return KEYWORDELSE;}

[a-zA-Z\_][a-zA-Z0-9]\* {return IDENTFIER;}

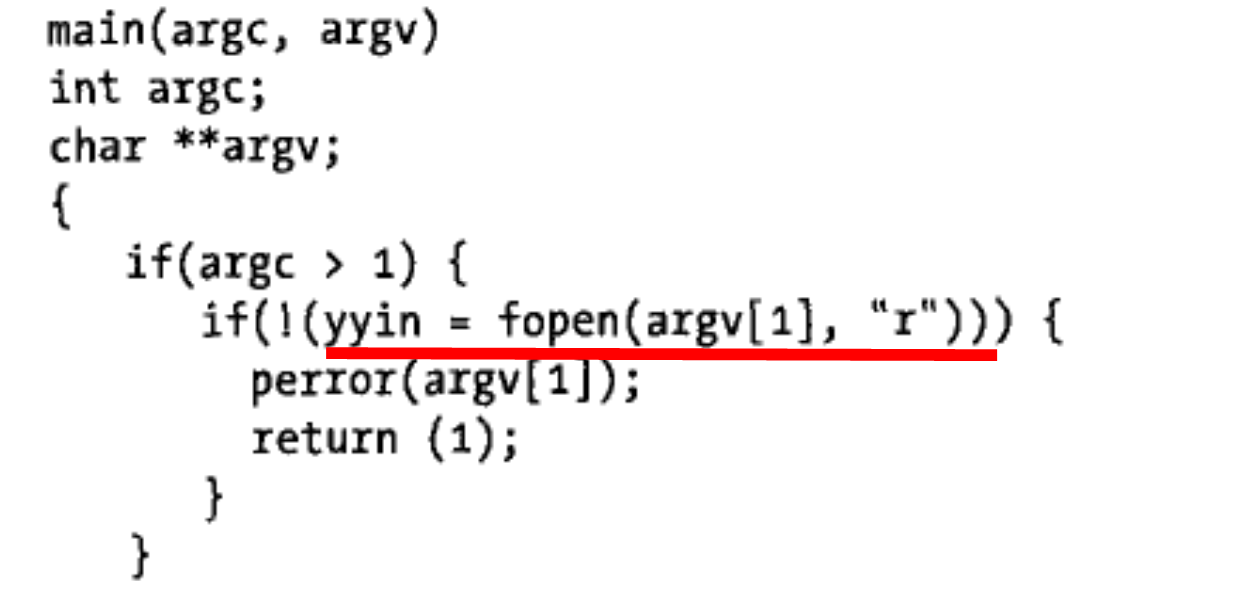
对于前三个模式来说，字符串“+=”被匹配为一个记号（匹配最长的字符串）。

对于后两个模式来说，“else”会被匹配程一个关键字而不是标识符（两个都匹配，取最早出现的）。

### 文件I/O操作

除非你另行指定，否则flex词法分析器总是读取标准输入。实际上，大多数词法分析器都是从文件读取输入。

除非你另做安排，否则词法分析器总是通过名为yyin的文件句柄读取输入。



如果用户在命令行中给出了文件名，主例程会打开这个文件并把相应的文件句柄赋给yyin；否则，yyin将保持未赋值的状态，这种情况下，yylex会自动把stdin赋给它。

#### Flex词法分析器输入

就一个包含词法分析器的程序而言，词法分析器的性能通常决定了整个程序的性能。

大多数情况下，flex词法分析器从文件或者标准输入（比如用户终端）中读取输入。从文件读取和从终端读取存在着一个微小但是重要的差异——预读机制。

Flex词法分析器使用称为YY\_BUFFER\_STATE的数据结构来处理输入，该结构定义了一个单一的输入源。它包含一个字符串缓冲区以及一些变量和标记。通常它会有一个指向所读取文件的FILE\*，但是我们也可以创建一个与文件无关的YY\_BUFFER\_STATE来分析已经在内存中的字符串。

最后，处于最大灵活性的考虑，可以重新定义flex用于读取输入到当前缓冲区的宏：

#define YY\_INPUT(buf,result,max\_size)…

由于重新定义YY\_INPUT的能力先于附加的YY\_BUFFER\_STATE，所以过去大多数人使用前者来实现，现在而言他们用后者来实现会更好。目前，自定义YY\_INPUT主要用在事件驱动系统里，这种系统的输入来自于其他一些内容，而这些内容无法预先载入字符串缓冲区且无法支持标准输入输出。

输入管理的三个层次是：

1. 设置yyin来读取所需文件
2. 创建并使用YY\_BUFFER\_STATE输入缓冲区
3. 重定义YY\_INPUT

#### Flex词法分析器输出

# 语法解析

## YACC

## Bison

### 概述

Bison是yacc的现代版本，由自由软件基金会（Free Software Foundation）的GNU项目帮助发布。

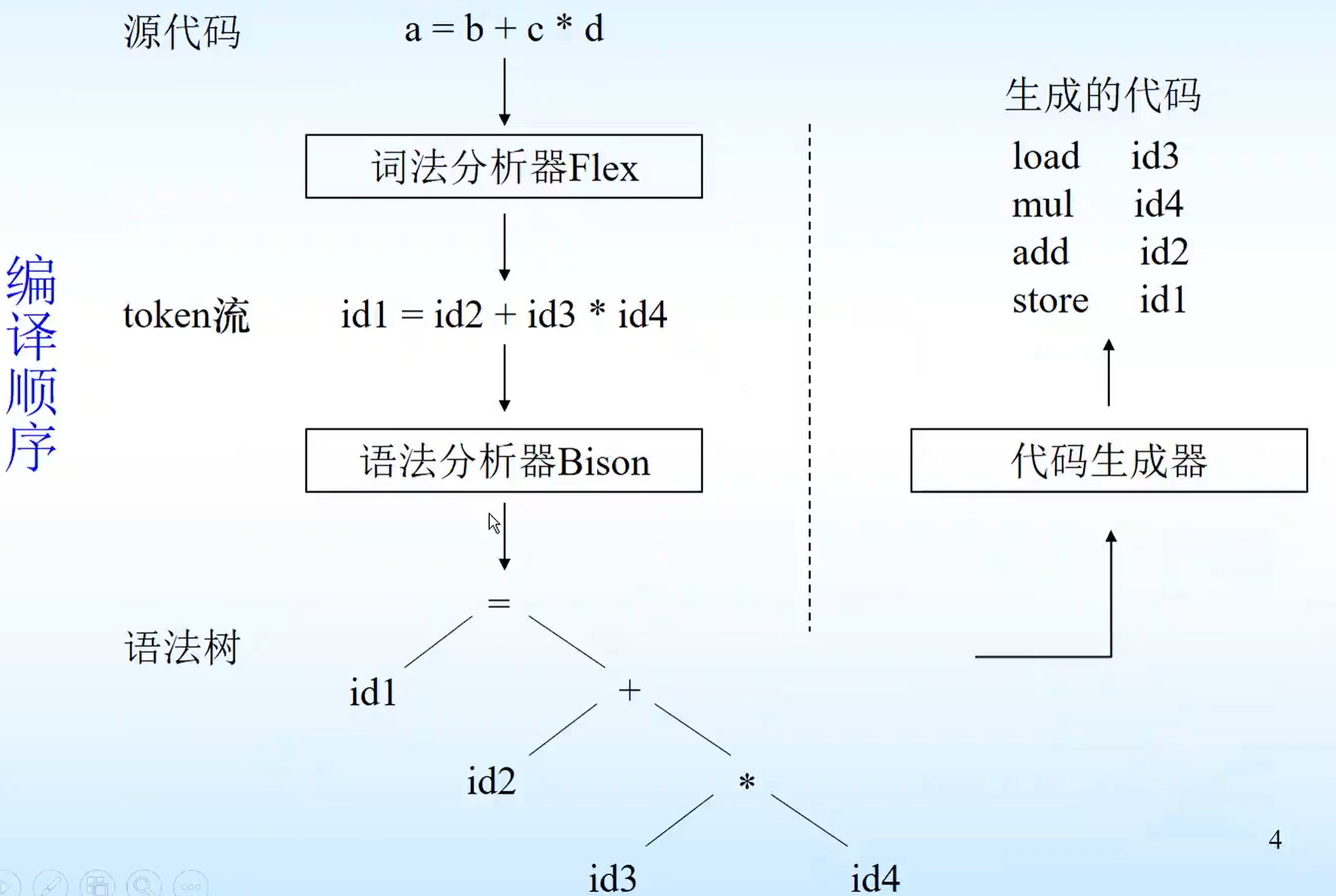
一般Flex和Bison是协同工作的：

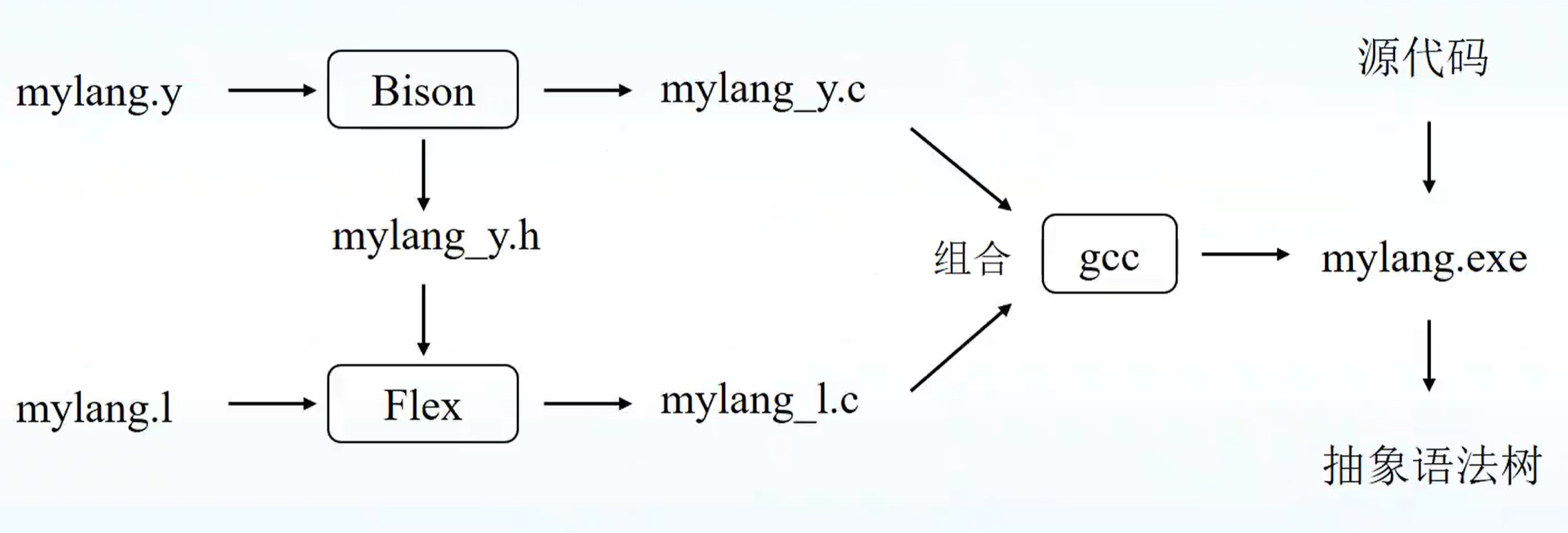
Flex：词法分析器

将输入分割成一个个有意义的词块，称为记号（token）（这个其实就对应Bison中定义的类型%token，识别的就是Flex解析后的结果token）。

Bison：语法分析器

根据给定的语法规则将Flex生成的tokens转换为抽象语法树（AST）。





### 源程序结构

Bison的输入文件\*.y结构如下：

#### 定义段

以C语法写的一些定义和声明，例如：文件包含，宏定义，全局变量定义，函数声明等。该部分以符号%{%}包裹。

##### %token %left

##### %right

##### %nonassoc

对词法的终结符合非终结符做一些相关声明，这些声明主要包括：%token，%left，%right，%nonassoc，%union，%type，%start。

%token定义词法中使用的终结符。

%left，%right，%nonassoc也是定义词法中使用的终结符，定义形式与%token类似，但是它们定义的终结符具有某种优先级和结合性。

%left表示左结合（例如“+,-,\*,/”）  
 %right表示右结合（例如“=”，x=y=z🡪x=(y=z)）

%nonassoc表示不可结合（例如“-”负号）

优先级关系则是以它们定义出现的顺序决定的，先定义的优先级低，最后定义的优先级最高（即Flex的最长匹配原则类似），同时定义的优先级相同。

例如：%left A B %nonassoc C %right D

结合性关系为：A和B左结合，C不可结合，D右结合

优先级关系为：A=B<C<D

##### %union

##### %type

一旦联合类型被定义，便需要告诉bison每种符号使用的值类型，这通过放置在尖括号（<>）中的联合类型的相应成员名字来确定。

终结符：

%token <num> TOKEN1

%token <id> TOKEN2 TOKEN3

非终结符：

%type <id> sym1 sym2

%type <num> sym3

##### %start

%start声明起始规则，也就是语法分析器首先开始分析的规则，默认是第一个规则。

大多数情况下，最清楚的表达语法的方式是自上而下，起始位置放在第一个，这样%start就不需要了。

起始符号必须具备一个空规则，旨在让开始输入的记号能够从起始符号开始匹配。

#### 语法规则段

#### 辅助函数段

辅助函数段用C语言语法来写，一般指在语法规则段汇总用到或在语法分析器的其他部分用到的函数。

一般来说，除语法规则段用到的函数外，辅助函数还包括例程：yylex()，yyerror()，main()。

main()是主程序，主要作用是调用语法分析驱动程序yyparse()函数。

**yylex()是词法分析程序，返回记号。yyparse()将会调用yylex()获取记号**。

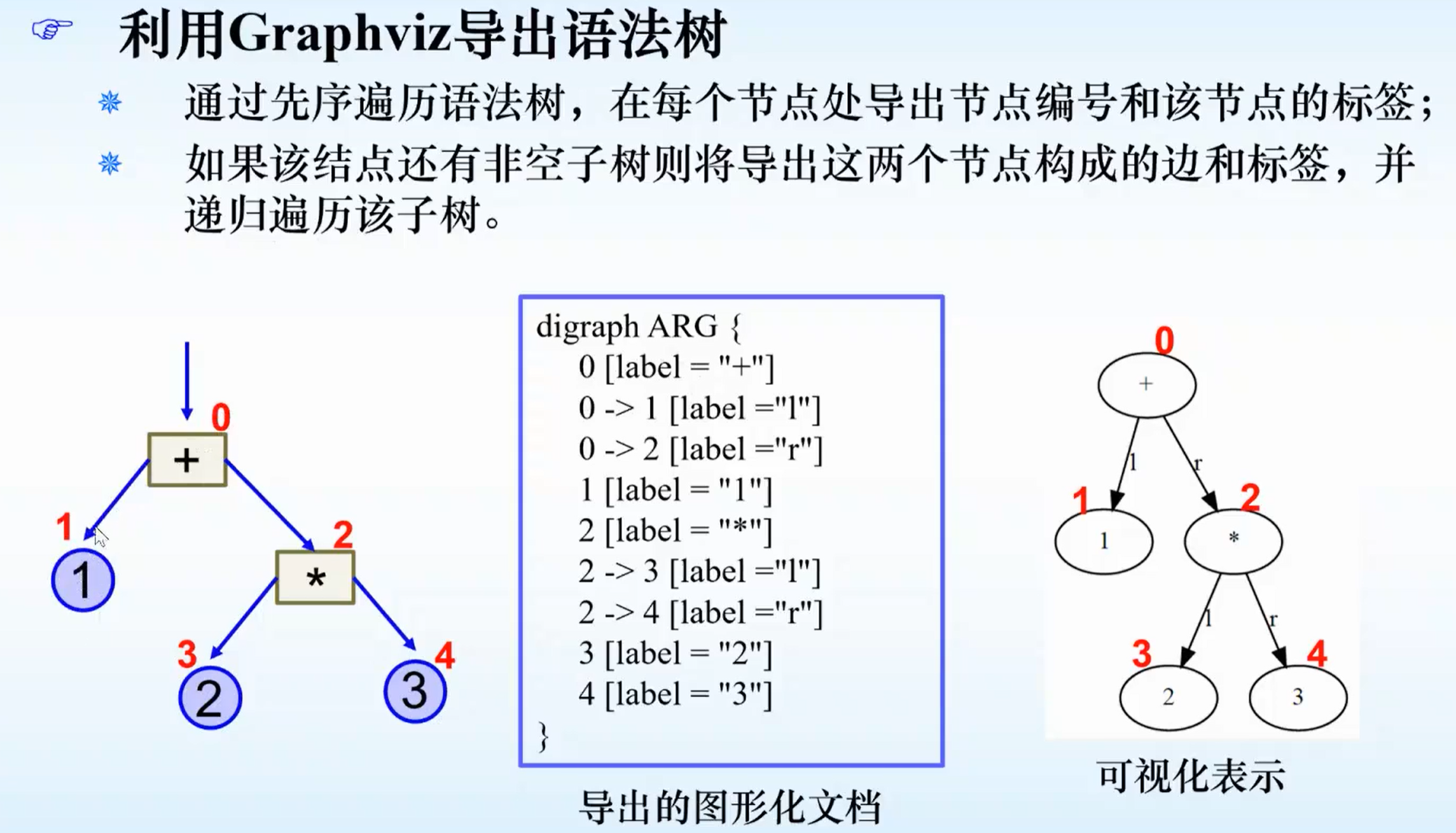
yyerror()函数输出错误信息。

### 语法树

#### 语法树遍历

#### Graphviz可视化

是开源图形可视化软件，可以将结构信息表示为抽象图和网络图。在网络、生物信息学、软件工程、数据库设计等领域的可视化方面有着重要的应用。



# YACC

## 概述

YACC全称（Yet Another Compiler Compiler），是一个经典的生成语法分析器的工具。是Unix/Linux上一个用来生成编译器的编译器（编译器代码生成器）。YACC生成的编译器主要是用C语言写成的语法解析器（Parser），需要与词法解析器Lex一起使用，再把两部分产生出来的C程序一并编译。

YACC的基本内容可以参考bison，yacc是bison的前身，bison是它的现代版本，yacc一般与lex一起使用，而bison一般与flex一起使用。

## 语法结构

一般情况下，YACC主要有以下三方面内容组成：定义部分、规则部分和用户子例程。

...定义部分...

%%

...规则部分...

%%

...用户子程序...

这三个部分通过两个百分号组成的行来分割，前两个部分是必须的，而最后一个部分及其前面的两个百分号都是可以省略的。

### 定义部分

定义部分可以包含文字块，它们是被原样拷贝到生成的C文件的开始部分的C代码，**文字块存在于%{和%}之间**。

声明包括%union、%start、%token、%type、%left、%right和%nonassoc。

在yacc中还可以用c语言的注释语法/\*...\*/。

#### 文字块

数据库中yacc文字块包含了define、include以及一些函数，文字块的所有内容在%{和%}之间。

#### %union

%union声明标识出了符号值可能拥有的所有C类型，格式如下：

%union{

...域声明...

}

域声明将被原封不动的拷贝到输出文件中类型为YYSTYPE的c的union声明里，如果不在%union中声明，将会把YYSTYPE定义为int。**可以通过%type来把%union中声明的类型与特定的符号关联起来**。

#### %start

通常起始规则，也就是语法分析其首先开始分析的规则，是第一个出现的规则。如果你希望以其他规则作为起始规则，你可以在声明部分中如下编写：

%start somename

来规则somename作为起始规则。

在数据库的yacc文件中，我们表达语法的方式是自上而下，起始规则放在第一个，这样就不需要%start。

#### %token

Token是记号，或者叫做**终止符**，是**词法分析器传递给语法分析器的符号**。

记号可以是通过%token定义的符号或者是单引号中的各个字符，所有被用来作为记号的符号必须在定义部分显示声明，例如：

#token up down left right

**记号可以通过%left、%right和%nonassoc来声明，它们都具有和%token一样的语法选项，只是它们各自的优先级和结合性不同；%left表示操作符左结合，%right表示操作符右结合，%nonassoc表示操作符无结合**。

%left表示左结合（例如“+,-,\*,/”）  
 %right表示右结合（例如“=”）

%nonassoc表示不可结合（例如“-”负号）

注：在数据库中%token就是mysql语法关键字，比如insert、update、select等等。根据优先级和结合性，可以使用%token，%left，%right，%nonassoc。

#### %type

可以使用%type来声明**非终结符的类型**。该定义具有如下格式：

%type <type> name.name,...

每个<type>的名字必须以%union定义过，而每一个name就是非终结符的名字。

**说明：**

终结字符集：**不能单独出现在推导式左边的符号，不能够再继续推导**。

非终结字符集：不是终结字符集的符号都为非终结字符集，是可以拆分元素。

### 规则部分

规则部分包含**语法规则和语义动作**的C代码。

#### 语法规则

语法规则是由一组规则组成，每一个规则由一个**非终结符开始**，然后是冒号和可能为空的符号、文字符号和动作的列表。**规则需要用分号结束**。例如：

Date：month ‘/’ day ‘/’ year

其中起始符号和冒号被称为规则的左部，即date，而规则剩余的部分是右部，**右部可以为空**。

如果语法中几个规则拥有相同的左部，则可以写成如下的格式：

Declaration：EXTERNAL name

|ARRAY name ‘(’size’)’

**使用竖线是很好的风格，但是在竖线之前的分号必须被省略**。

**对于不定长的项目列表，需要使用递归规则，也就是用自身定义自己**。例如：

Exprlist：expr

| exprlist ‘,’ expt

;

注意：**任意递归规则里面每一个规则都必须至少有一条非递归的分支（即不指向其自身）；否则，将没有任何途径来终止它所匹配的字符串，这是一个错误**。

#### 动作

动作是yacc匹配语法中一条规则时执行的c代码，动作必须是c语言的复合语句，如：

Date：month ‘/’ day ‘/’ year {printf(“data found”);};

**动作可以通过使用一个美元符号加上一个数字来使用规则中语法符号所关联的值，冒号后第一个语法符号的数字是1**，如：

Date：month ‘/’ day ‘/’ year {printf(“date %d-%d-%d found,$1,$3,$5”);};

**名字$$指向左部符号，也就是冒号左边符号的值，对于没有语义动作的规则，使用如下默认动作：{$$=$1;}**

如果在规则中间写上一个嵌套动作，将会创造一个新的规则，这条规则的左部为空，而右部是一个伪造的名字，然后把嵌入动作作为该规则的语义动作，并且使用那个伪造的名字来替换原有的规则中的嵌入动作，所以末尾的语义动作也可以像使用其他符号一样使用它的值，如：

Thing：A {$$=17;} B C {printf(“%d”,$2);}

其会打印出“17”，此时，在尾部的语义动作中，$1表示A，$3表示B，$4表示C，而此时$2表示嵌入动作的值。

当一条规则中间部分出现了语义动作时，而语义动作前面的规则和其他规则一样，这时候语法分析器会自动读后面一个符号，一般情况下此时后面的一个符号是有区别的，如果后面一个符号没有区别就会出现冲突。

#### 特殊字符

下面是数据库yacc中用到的一些特殊字符：

%：两个%%的行用来分割语法的各个部分，所有定义部分的声明都以%%开始，包括%{ %}等。

$：在语义动作中，美元符号引入一个值的引用。例如，$3代表规则右部第三个符号的值，$$代表左部符号。

‘：文字记号用单引号引起。

{}：语义动作中c代码使用花括号括起。

;：规则部分每一个规则都必须使用分号结尾，后面紧跟以竖线开始的另一个规则的规则除外，但其实分号是可以省略的，不过我们强制使用。

|：当两个连续的规则具有相同的左部时，第二个规则可以把左部的符号和冒号替换为竖线。

:：在每一条规则中，冒号出现在规则左部的符号之后。

‘-’expr %prec ‘\*’为使一元’一’的优先级与’\*’相同，我们使用了子句：

%prec’\*’

它说明它所在的语法规则中最右边的运算符或终结符的优先级与%prec后面的符号的优先级相同，注意%prec子句必须出现在某语法规则结尾处分号之前，%prec子句并不改变’-’作为二元运算符时的优先级。

**操作符按照优先级的升序被声明。在同一行的所有操作符具有相同的优先级**。

即，先出现的规则优先被匹配。

例如：

%left ‘+’’-’

%left’\*’’/’

%right POW

优先级最低的操作符是+和-，优先级中等的操作符是\*和/，优先级最高的是POW。

#### YYABORT

语义动作中的特殊语句：YYABORT。

该语句是的语法分析器的分析历程yyparse立即返回一个非零值，表明存在错误。

### 总结

对于一条建表语句：create table if not exists test.t1(a int key, b int,c int,d int, e int)distributed by duplicate(g1,g2,g3);其yacc语法解析总体流程如下：

query->verb\_clause->statement->create->opt\_if\_not\_exists->table\_ident->create2->create1a->create\_filed\_list->opt\_create\_partitioning->opt\_create\_distributed

1. 首先一条SQL语句流进入yacc解析，首先匹配的是query标签
2. 在query标签中会匹配verb\_clause标签，如果sql语句为空，则直接结束
3. 在verb\_clause标签中会去匹配statement标签，另一个标签是begin
4. 在statement中会去匹配所有SQL语句类型关键字的标签，如create、drop、alter、delete、insert、update、select、truncate等，除了begin外，begin是在verb\_clause中匹配的
5. 进入匹配到的关键字的标签，如create，开始匹配create的语法，并进行语法树相应的处理

# 语句分析

## 查询语句

说了以上这么多，那么究竟一条sql语句是如何执行的呢？其实我们的sql可以分为两种，一种是查询，一种是更新（增加，更新，删除）。我们先分析下查询语句，语句如下：

select \* from tb\_student A where A.age='18' and A.name=' 张三 ';

结合上面的说明，我们分析下这个语句的执行流程：

先检查该语句是否有权限，如果没有权限，直接返回错误信息，如果有权限，在MySQL8.0版本以前，会先查询缓存，以这条sql语句为key 在内存中查询是否有结果，如果有直接缓存，如果没有，执行下一步。

通过分析器进行词法分析，提取 sql 语句的关键元素，比如提取上面这个语句是查询 select，提取需要查询的表名为 tb\_student,需要查询所有的列，查询条件是这个表的 id='1'。然后判断这个 sql 语句是否有语法错误，比如关键词是否正确等等，如果检查没问题就执行下一步。

接下来就是优化器进行确定执行方案，上面的 sql 语句，可以有两种执行方案：

a.先查询学生表中姓名为“张三”的学生，然后判断是否年龄是 18。

b.先找出学生中年龄 18 岁的学生，然后再查询姓名为“张三”的学生。

那么优化器根据自己的优化算法进行选择执行效率最好的一个方案（优化器认为，有时候不一定最好）。那么确认了执行计划后就准备开始执行了。

进行权限校验，如果没有权限就会返回错误信息，如果有权限就会调用数据库引擎接口，返回引擎的执行结果。

## 更新语句

以上就是一条查询sql的执行流程，那么接下来我们看看一条更新语句如何执行的呢？sql语句如下：

update tb\_student A set A.age='19' where A.name=' 张三 ';

我们来给张三修改下年龄，在实际数据库肯定不会设置年龄这个字段的，不然要被技术负责人打的。其实条语句也基本上会沿着上一个查询的流程走，只不过执行更新的时候肯定要记录日志啦，这就会引入日志模块了，MySQL 自带的日志模块式binlog（归档日志），所有的存储引擎都可以使用，我们常用的InnoDB引擎还自带了一个日志模块 redo log（重做日志），我们就以InnoDB 模式下来探讨这个语句的执行流程。流程如下：

•先查询到张三这一条数据，如果有缓存，也是会用到缓存。

•然后拿到查询的语句，把age改为19，然后调用引擎API接口，写入这一行数据，InnoDB引擎把数据保存在内存中，同时记录redo log，此时redo log进入prepare状态，然后告诉执行器，执行完成了，随时可以提交。

•执行器收到通知后记录binlog，然后调用引擎接口，提交redo log为提交状态。

•更新完成。

这里肯定有同学会问，为什么要用两个日志模块，用一个日志模块不行吗?

这是因为最开始MySQL并没与InnoDB引擎( InnoDB 引擎是其他公司以插件形式插入MySQL的) ，MySQL自带的引擎是MyISAM，但是我们知道redo log是InnoDB引擎特有的，其他存储引擎都没有，这就导致会没有crash-safe的能力(crash-safe的能力即使数据库发生异常重启，之前提交的记录都不会丢失)，binlog日志只能用来归档。

并不是说只用一个日志模块不可以，只是InnoDB引擎就是通过redo log来支持事务的。那么，又会有同学问，我用两个日志模块，但是不要这么复杂行不行，为什么redo log要引入prepare预提交状态？这里我们用反证法来说明下为什么要这么做？

•先写redo log直接提交，然后写binlog，假设写完 redo log后，机器挂了，binlog日志没有被写入，那么机器重启后，这台机器会通过redo log恢复数据，但是这个时候bingog并没有记录该数据，后续进行机器备份的时候，就会丢失这一条数据，同时主从同步也会丢失这一条数据。

•先写binlog，然后写redo log，假设写完了binlog，机器异常重启了，由于没有redo log，本机是无法恢复这一条记录的，但是binlog又有记录，那么和上面同样的道理，就会产生数据不一致的情况。

如果采用redo log两阶段提交的方式就不一样了，写完binglog后，然后再提交redo log就会防止出现上述的问题，从而保证了数据的一致性。那么问题来了，有没有一个极端的情况呢？假设redo log处于预提交状态，binglog也已经写完了，这个时候发生了异常重启会怎么样呢？这个就要依赖于MySQL的处理机制了，MySQL的处理过程如下：

•判断redo log是否完整，如果判断是完整的，就立即提交。

•如果redo log只是预提交但不是commit状态，这个时候就会去判断binlog是否完整，如果完整就提交redo log，不完整就回滚事务。

这样就解决了数据一致性的问题。

# 代价模型

总代价模型：COST = CPU Cost + IO Cost

MySQL在cost类型上分为IO、CPU和Memory，MySQL5.7的代价模型还在完善中，Memory的代价虽然已经收集了，但还没有计算在最终的代价中。

MySQL5.7在源码上对cost模型进行了大量重构，代价分为server层和engine层。server层主要是CPU代价，而engine层主要是IO代价。MySQL5.7 引入了两个系统表mysql.server\_cost和mysql.engine\_cost来分别配置这两个层的代价。

以下分析均基于MySQL5.7.10

## server\_cost

1、row\_evaluate\_cost (default 0.2) 计算符合条件的行的代价，行数越多，此项代价越大；

2、memory\_temptable\_create\_cost (default 2.0) 内存临时表的创建代价；

3、memory\_temptable\_row\_cost (default 0.2) 内存临时表的行代价；

4、key\_compare\_cost (default 0.1) 键比较的代价，例如排序；

5、disk\_temptable\_create\_cost (default 40.0) 内部myisam或innodb临时表的创建代价；

6、disk\_temptable\_row\_cost (default 1.0) 内部myisam或innodb临时表的行代价；

可以看出创建临时表的代价是很高的，尤其是内部的myisam或innodb临时表。

## engine\_cost

1、io\_block\_read\_cost (default 1.0) 从磁盘读数据的cost，对innodb来说，表示从磁盘读一个page的cost；

2、memory\_block\_read\_cost (default 1.0）；

从内存读数据的cost，对innodb来说，表示从buffer pool读一个page的cost。

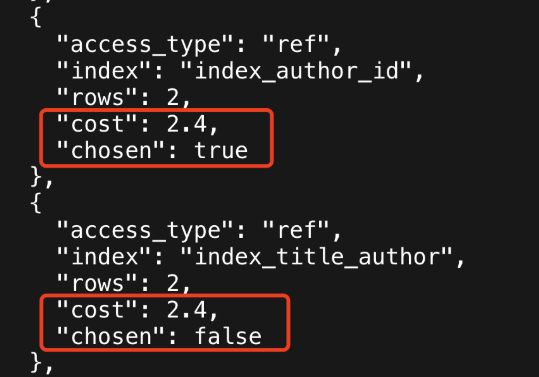
目前io\_block\_read\_cost和memory\_block\_read\_cost默认值均为1，实际生产中建议酌情调大memory\_block\_read\_cost，特别是对普通硬盘的场景。

对表tb\_article创建复合索引index\_title\_author

ALTER TABLE tb\_article ADD KEY index\_title\_author(`title`,`author\_id`);

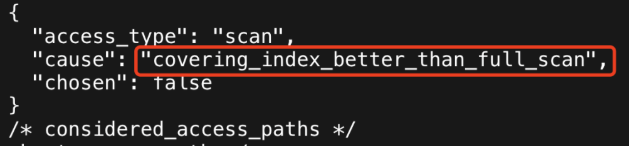
select \* from tb\_article where author\_id=20 and title='b';

index\_author\_id和index\_title\_author的cost相等，MySQL会优先选择叶子块数量较少的索引。



对于SQL语句：select title, author\_id from tb\_article where author\_id=20 and title='b';

MySQL会优先选择走复合索引index\_title\_author，原因是index\_title\_author是索引覆盖扫描，不需要回表，性能较高。



# 总结

MySQL主要分为Server层和引擎层，Server层主要包括连接器、查询缓存、分析器、优化器、执行器，同时还有一个日志模块（binlog），这个日志模块所有执行引擎都可以共用，redolog只有InnoDB有。

引擎层是插件式的，目前主要包括，MyISAM，InnoDB，Memory 等。

SQL等执行过程分为两类，一类对于查询等过程如下：权限校验🡪查询缓存🡪分析器🡪优化器🡪权限校验🡪执行器🡪引擎

对于更新等语句执行流程如下：分析器🡪权限校验🡪执行器🡪引擎🡪redo log prepare🡪binlog🡪redo log commit