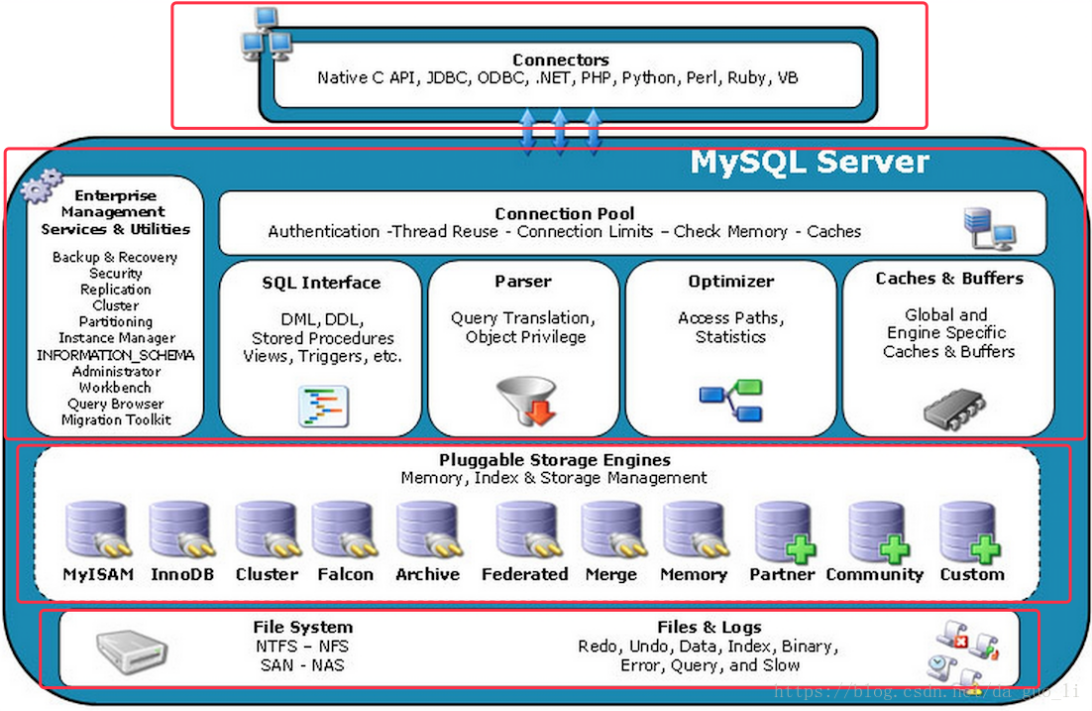
### MySQL架构



这四层自顶向下分别是网络连接层，服务层（核心层），存储引擎层，系统文件层。

#### 网络接入层

**（安全的接受客户端的连接实现通信）**

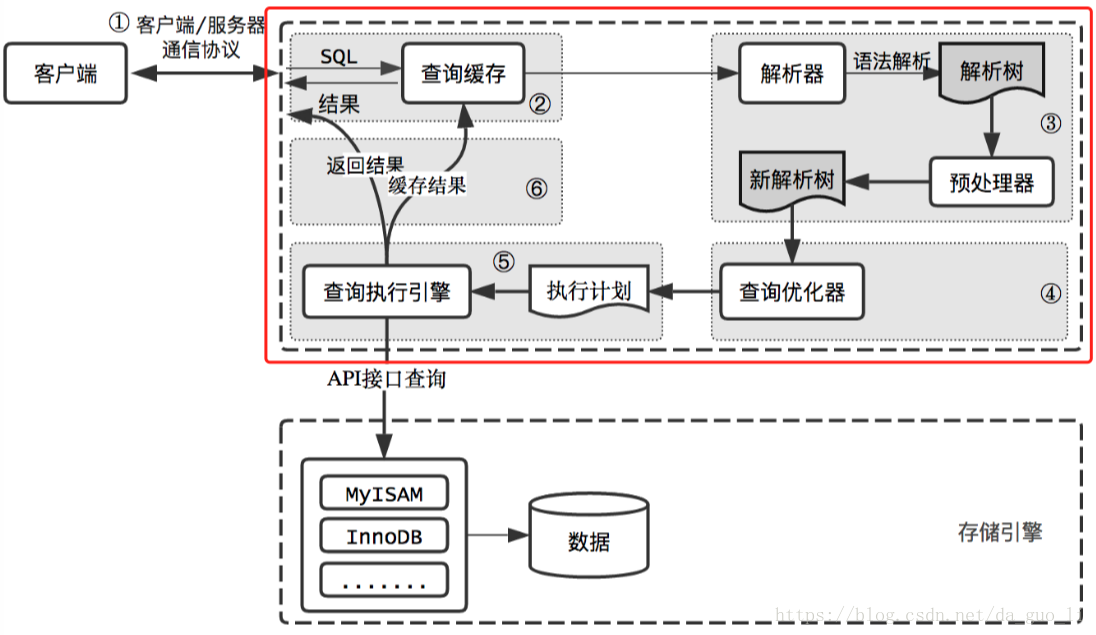
作用：

 主要负责连接管理、授权认证、安全等等。**每个客户端连接都对应着服务器上的一个线程**。**服务器上维护了一个线程池，避免为每个连接都创建销毁一个线程（避免浪费资源）**。当客户端连接到MySQL服务器时，服务器对其进行认证。可以通过用户名与密码认证，也可以通过SSL证书进行认证。登录认证后，服务器还会验证客户端是否有执行某个查询的操作权限。这一层并不是MySQL所特有的技术。

#### 服务层

作用：

      第二层服务层是MySQL的核心，MySQL的核心服务层都在这一层，查询解析，SQL执行计划分析，SQL执行计划优化，查询缓存。以及跨存储引擎的功能都在这一层实现：存储过程，触发器，视图等。通过下图来观察服务层的内部结构：



图中红色框中标出来的就是MySQL服务层内部执行的过程

下面来简单分析SQL语句在服务层中具体的流程：

查询缓存

    在解析查询之前，服务器**首先**会检查查询缓存，如果能找到对应的查询，服务器不必进行查询解析、优化和执行的过程，直接返回缓存中的结果集。

解析器与预处理器（**解析器是判断命令的语法 预处理是判断命令的合理性**）

    MySQL会解析查询，并创建了一个内部数据结构（解析树）。这个过程解析器主要通过语法规则来验证和解析。比如SQL中是否使用了错误的关键字或者关键字的顺序是否正确等等。预处理会根据MySQL的规则进一步检查解析树是否合法。比如要查询的数据表和数据列是否存在等。

查询优化器（**找出最优查询路径生成执行计划**）

    优化器将其转化成查询计划。多数情况下，一条查询可以有很多种执行方式，最后都返回相应的结果。优化器的作用就是找到这其中最好的执行计划。优化器并不关心使用的什么存储引擎，但是存储引擎对优化查询是有影响的。优化器要求存储引擎提供容量或某个具体操作的开销信息来评估执行时间。

查询引擎（**根据执行计划调用存储引擎的相应api实现查询**）

在完成解析和优化阶段以后，MySQL会生成对应的执行计划，查询执行引擎根据执行计划给出的指令调用存储引擎的接口得出结果。

#### 存储引擎层

作用：

    负责MySQL中数据的存储与提取。 服务器中的查询执行引擎通过API与存储引擎进行通信，通过接口屏蔽了不同存储引擎之间的差异。MySQL采用插件式的存储引擎。MySQL为我们提供了许多存储引擎，每种存储引擎有不同的特点。我们可以根据不同的业务特点，选择最适合的存储引擎。如果对于存储引擎的性能不满意，可以通过修改源码来得到自己想要达到的性能。例如阿里巴巴的X-Engine，为了满足企业的需求facebook与google都对InnoDB存储引擎进行了扩充。

特点：

存储引擎是针对于表的而不是针对库的（一个库中不同表可以使用不同的存储引擎），服务器通过API与存储引擎进行通信，用来屏蔽不同存储引擎之间的差异。

下面大致介绍一下MySQL中常见的的存储引擎

InnoDB

    特点：支持事务，适合OLTP应用，假设没有什么特殊的需求，一般都采用InnoDB作为存储引擎。支持行级锁，从MySQL5.5.8开始,InnoDB存储引擎是默认的存储引擎。

MyISAM

    特点

        不支持事务，表锁设计，支持全文索引，主要应用于OLAP应用

    场景

        在排序、分组等操作中，当数量超过一定大小之后，由查询优化器建立的临时表就是MyISAM类型

        报表，数据仓库

Memory

    特点

        数据都存放在内存中，数据库重启或崩溃，表中的数据都将消失，但是标的结构还是会保存下来。默认使用Hash索引。

    场景

        适合存储OLTP应用的临时数据或中间表。

        用于查找或是映射表，例如邮编和地区的对应表。

除此之外还有CSV,Federated、Archive等等。后面会开一篇博客专门讲解MySQL存储引擎。

#### 系统文件层

作用：

        该层主要是将数据库的数据存储在文件系统之上，并完成与存储引擎的交互。

**总结：数据库就是避免了用户直接与底层数据文件操作，而提出的一套规范的访问数据的标准（提供高效多功能的操作，不用你自己去实现相应的功能，使用相应语句调用就行了）。就是利用相应的SQL语言来进行访问操作。因为要向数据库发送信息，所以客户端有连接操作，数据库也有相应的网络层。**

### 索引

**本质：**就像二叉搜索树一样提升查找效率

**查找时间组成：**

1.对内存里面的索引的遍历

2.对磁盘的IO操作，将磁盘数据放入内存

**因为内存访问很快，所以对磁盘IO访问占主要**

**为什么不用二叉树呢？**  
 因为我们要考虑**磁盘IO**的影响，它相对于内存来说是很慢的。数据库索引是存储 在磁盘上的，当数据量大时，就不能把整个索引全部加载到内存了，只能逐一加载每一 个磁盘页（对应索引树的节点）。所以我们要减少IO次数，对于树来说，IO次数就是 树的高度，而“矮胖”就是b树的特征之一，它的每个节点最多包含m个孩子，m称为b 树的阶，m的大小取决于**磁盘页**的大小

**采用的两种数据结构：**

**B树：**

**关键字集合**分布在整颗树中；

任何一个关键字出现且**只出现在一个结点中**；

搜索有可能在非叶子结点结束；

其搜索性能等价于在关**键字全集内做一次二分查找**

**B+树：**

有n棵子树的非叶子结点中含有n个关键字（b树是n-1个），这些关键字不保存数据，**只用来索引**，所有数据都保存在叶子节点（b树是每个关键字都保存数据）

所有的**叶子结点中包含了全部关键字**的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。

所有的非叶子结点可以看成是索引部分，结点中仅含其子树中的最大（或最小）关键字。

通常在b+树上有**两个头指针**，一个指向根结点，一个指向关键字最小的叶子结点。

同一个数字会在不同节点中**重复**出现，根节点的最大元素就是b+树的最大元素。

**B+树为什么优于B树？**

因为B+树非叶子节点只存放了**索引值**，而B树节点还包含了**关键字**对应的**存储地址**，所以一次IO访问就能读入更大的B+树关键字，索引的阶数就更大，访问就更快。

**索引理解：**

**比如一个有一个表，包含了学生的年龄，姓名，等很多信息，现在以年龄建立一张索引表，B+树中的非叶子节点可能就只有一个年龄数据，而B树则还要有该年龄对应的完整数据的地址**