数据库基础与安全

数据库作为信息与技术架构核心的一部分，它的安全性与效率至关重要。

数据库简介

- 定义：**数据库是用于存储、管理和处理数据的系统。它允许存储大量信息，支持数据的有效组织、存储、访问和更新。**

数据库的作用

1 **数据存储**

数据库提供了持久的数据存储机制，适用于文本、数字、图片等多种数据格式的存储。

2 **数据管理**

强大的数据管理功能，包括数据的增加、删除、修改和查询。

3 **数据处理**

支撑各种数据操作，以满足特定的业务需求和流程处理。

4 **数据查询**

通过像SQL这样的强大查询语言，用户可以快速高效地检索所需数据。

数据库的应用场景

- 商业数据存储
- 网站数据管理
- 移动应用数据处理
- ...

相关概念辨析

数据库管理系统（DBMS）

关系型数据库

SQL

数据库管理系统（DBMS）

数据库管理系统（DBMS）是**管理、控制和访问数据库的工具或软件系统**，它提供了一组功能来实现对数据库中的数据进行高效、安全的存储、检索、修改和删除。

- **组成**：DBMS 包含一系列软件组件，例如查询引擎、存储引擎、日志管理、事务管理等。
- **作用**：DBMS 负责数据的存储、更新、查询、并发控制、数据安全、数据备份和恢复等任务。它提供了一组接口，使得用户或应用程序可以通过**查询语言**（如 SQL）来操作数据库。

数据库（Database）

数据库是**实际存储数据的集合**，可以理解为一种**容器**，用于存放结构化的信息。数据库包括表、记录、列等各种数据对象，通过这些对象来组织和存储信息。

- **组成**：数据库中的数据通常是按某种结构存储的，例如在关系型数据库中，数据按表（表由行和列组成）的形式存储。
- **作用**：数据库本身不具备管理、控制数据的功能，它只是数据的存储空间，数据存储的格式和内容由具体的需求决定。

总结

- **数据库**是存储数据的空间，是数据的“容器”。
- **DBMS** 是用于操作和管理数据库的软件系统，它提供了创建、查询、修改、删除数据等一系列管理功能。

关系型数据库

关系型数据库（Relational Database, RDBMS） 是一种基于**关系模型**的数据管理系统，它将数据存储在**表格**中，每个表由行和列组成，表与表之间通过关系连接。关系型数据库通过结构化查询语言（SQL）进行操作，是目前最常用的数据库类型之一，适用于事务型应用和结构化数据。

关系型数据库的基本概念

1. **表（Table）**：数据存储的基本单元，由**行（row）**和**列（column）**组成。每个表代表一个数据实体，如“用户”、“订单”、“产品”等。
2. **行（Row）**：表中的一行表示一个具体的记录或实体实例。例如，用户表的一行表示一个用户。
3. **列（Column）**：表中的列表示数据的属性或字段，每个列有一个数据类型（如整数、字符串、日期等）。例如，用户表的列可以包括“用户 ID”、“姓名”、“邮箱”等。
4. **主键（Primary Key）**：表中唯一标识一行的列，不能重复且不能为空。例如，用户表中的“用户 ID”可以作为主键。
5. **外键（Foreign Key）**：一种用来**建立表之间关系**的列，它引用了另一个表的主键。通过外键，关系型数据库可以将不同表的数据关联起来，实现**数据的完整性**和**一致性**。
6. **关系（Relation）**：表与表之间通过主键和外键建立联系，例如“用户表”与“订单表”可以通过用户 ID 建立关系。

关系型数据库的特点

1. **数据的一致性**：通过**主键**和**外键**约束，确保数据的唯一性和参照完整性，避免冗余和不一致。
2. **使用结构化查询语言（SQL）**：关系型数据库支持 SQL，提供了数据的插入、更新、删除、查询等操作，同时支持事务管理、权限控制等功能。
3. **事务支持和 ACID 特性**：关系型数据库支持事务管理，满足 **ACID** 特性（原子性、一致性、隔离性、持久性），确保数据的可靠性，特别适合银行、金融等高一致性需求的应用场景。
4. **结构化数据管理**：数据是高度结构化的，关系型数据库对数据有固定的模式（schema），如字段名称、数据类型等。每条记录遵循表的结构，这样的数据存储方式对数据关系和结构有很高的要求。

ACID

ACID 是数据库事务的四个基本特性，用于确保数据在并发操作和系统故障时的正确性和一致性。ACID 是 **原子性（Atomicity）**、**一致性（Consistency）**、**隔离性（Isolation）** 和 **持久性（Durability）** 的缩写。

原子性（Atomicity）

原子性确保事务中的所有操作要么全部执行，要么全部不执行，不允许部分完成。例如，银行转账操作必须同时扣除一个账户的余额并增加另一个账户的余额，不能只执行其中一个操作。

实现方法：数据库通过日志（log）或回滚机制来实现原子性。日志记录事务的每一个操作，若事务失败，数据库会使用日志回滚到事务开始前的状态。

一致性（Consistency）

一致性保证数据库在事务完成前后都处于一种合法的状态，即所有约束（如主键、外键、唯一性约束等）在事务执行前后都有效。例如，一个银行账户的余额不能变成负数。

实现方法：数据库在每次事务执行时，会自动检查并保持所有约束的完整性，确保数据不会违反这些规则。若违反约束，事务会被回滚。

隔离性（Isolation）

隔离性确保多个事务同时执行时不会互相干扰。一个事务的中间状态对其他事务是不可见的，直到该事务提交。隔离性防止了并发事务间的数据冲突。

- **隔离级别**：
  - **未提交读（Read Uncommitted）**：事务可以看到其他未提交事务的数据，容易引发脏读。
  - **已提交读（Read Committed）**：事务只能看到已提交的数据，避免了脏读。
  - **可重复读（Repeatable Read）**：保证在同一事务中多次读取的数据一致，防止不可重复读。
  - **序列化（Serializable）**：最高隔离级别，所有事务串行执行，避免并发问题，但性能较低。
- **实现方法**：数据库通常通过**锁机制**或**多版本并发控制（MVCC）**来实现隔离性。锁定数据可以防止其他事务在未提交时访问或修改数据，而 MVCC 通过保存数据的多个版本提供更高效的并发支持。

持久性（Durability）

持久性保证事务一旦提交，数据就会**永久保存在数据库中**，即使系统故障或重启，已提交的数据也不会丢失。

- **实现方法**：数据库使用**写前日志（WAL）和数据持久化机制**来确保持久性。在事务提交前，数据库会将所有修改写入日志并保存到磁盘，确保数据的永久性

总结

- **原子性**：事务的操作要么全部完成，要么全部不执行。
- **一致性**：事务的执行不能破坏数据的完整性约束。
- **隔离性**：并发事务间互不干扰，避免数据冲突。
- **持久性**：事务提交后的数据永久保存在数据库中。

ACID 特性确保了数据库系统的可靠性和一致性，使得数据库在并发和故障情况下仍能保持数据的正确性。

经典关系型数据库管理系统-MySQL

MySQL 是一种**关系型数据库管理系统（RDBMS）**，以**结构化查询语言（SQL）**为基础操作语言，用于存储、查询和管理数据。它被广泛应用于 Web 开发、数据存储和分析等场景。

MySQL 的特点

开源免费：MySQL 采用开放源代码许可证，可以免费使用、修改和分发，并提供付费商业版本以获得更多支持。

跨平台：支持多种操作系统，包括 Windows、Linux、macOS 等，可以方便地在不同平台之间迁移。

性能高效：经过优化，提供快速读写操作，适用于中小型应用和 Web 服务。

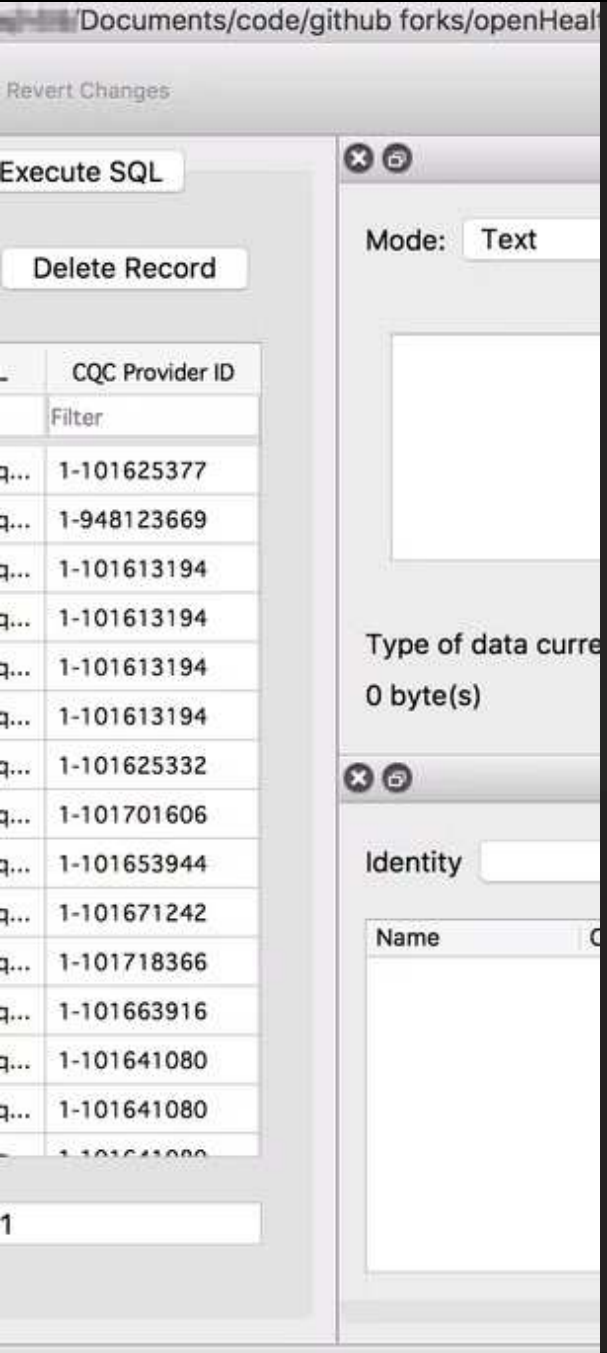
支持多用户并发访问：强大的并发性，可以处理多个用户的同时访问和操作。

数据安全性：提供访问控制、密码加密、数据备份等安全机制，确保数据的完整性和安全性。

丰富的存储引擎：支持多种存储引擎，如 InnoDB 和 MyISAM，可以根据需求选择合适的引擎来优化性能。

MySQL的缺陷

1. 功能限制：相较于其他高级数据库（如 PostgreSQL、Oracle），MySQL 在某些高级功能（如触发器、存储过程、复杂查询）上较弱。
2. 事务处理不如其他数据库：尽管 MySQL 支持事务处理（主要依赖 InnoDB 引擎），但在某些场景下，事务的隔离性和一致性可能不如其他数据库（如 PostgreSQL）。
3. 大数据量性能下降：当数据量非常大或查询变得复杂时，MySQL 的性能可能会受到影响，并需要专门的优化。
4. 有限的 ACID 支持：MySQL 对 ACID 特性（原子性、一致性、隔离性、持久性）的支持依赖于所选存储引擎，而部分引擎可能不完全满足 ACID 要求。
5. 扩展性较弱：MySQL 对于集群和分布式架构的支持有限，需要借助第三方工具或插件来实现分布式数据存储和高可用性。



SQLite

1 嵌入式数据库

SQLite 是一种轻量级、**嵌入式**的关系型数据库管理系统（RDBMS），不需要独立的服务器进程，数据全部存储在一个单一的数据库文件中。

它被广泛用于移动应用、嵌入式系统、桌面软件等场景

SQLite的主要特点

无需服务器

不需要配置和维护独立的数据库服务器。

跨平台

支持多种操作系统，如Windows、Linux、Mac OS等。

小巧灵活

占用资源少，适用于资源受限的环境，便于部署和维护。

易于操作

通过标准的SQL语言进行操作，学习成本低。

对比

MySQL 是功能更强大、适合高并发和大规模应用的服务器型数据库。

SQLite 是简单、轻量的嵌入式数据库，适合本地小规模数据存储和快速开发。

SQLite的基本架构

核心库

包含SQL解析器、查询优化器、存储引擎，负责处理SQL命令和管理数据结构。

存储引擎

B-trees结构，支持快速数据查找和操作，并支持选择不同类型的存储引擎如磁盘或内存。

文件系统接口

使SQLite能够与不同平台的文件系统进行读写交互，以保证跨平台兼容性。

外部接口

提供C语言API，让应用程序可以执行SQL查询、事务管理、数据读写等操作。

SQL语法入门教程

1 **数据库**

数据库是数据存储与管理系统，它由一系列表（Table）构成。

2 **表 (Table)**

数据表是数据库存储数据的基本单位，由行和列构成，类似于电子表格。

3 **列 (Column)**

列代表数据的属性，如“姓名”、“年龄”。

4 **行 (Row)**

每行代表一个数据记录，是各个列的具体值。

SQL语法特征

声明式语言

- SQL 是一种**声明式语言**，即用户只需要描述想要的结果，而无需指定具体的执行步骤。SQL 的查询会由数据库管理系统根据查询优化器生成的执行计划自动执行。
- **示例**：`SELECT name FROM employees WHERE age > 30;` – 这句 SQL 语句只是描述了需要查询“年龄大于 30 的员工的姓名”，不需要用户指定如何获取这些数据。

SQL语法特征

面向集合的操作

- SQL 是**面向集合**的语言，能对成组的数据（即集合）进行操作。查询结果通常是一个记录集合，而不是逐行处理数据。
- **示例**：`SELECT * FROM employees;` – 这条语句返回的是所有员工记录的集合，不是单个记录。

基础SQL语法

1. **SELECT语句**

   用于从表中检索数据。。

2. **WHERE子句**

   用于过滤记录。

3. **INSERT INTO语句**

   用于向表中插入新数据。

```
HAVING count(*) > 1
-- order of the results
ORDER BY col2
```

Useful keywords for **SELECTS**:

DISTINCT - return unique results
BETWEEN a **AND** b - limit the range, the values can be numbers, text, or dates
LIKE - pattern search within the column text
IN (a, b, c) - check if the value is contained among given.

Data Modification

```
-- update specific data with the WHERE clause
UPDATE table1 SET col1 = 1 WHERE col2 = 2
-- insert values manually
INSERT INTO table1 (ID, FIRST_NAME, LAST_NAME)
```

LEFT OUTER JOIN *- all rows from table A, even if they do not exist in table B*

INNER JOIN *- fetch the results that exist in both tables*

RIGHT OUTER JOIN *- all rows from table B, even if they do not exist in table A*

Updates on JOINed Queries

You can use **JOIN**s in your **UPDATE**s

```
UPDATE t1 SET a = 1
FROM table1 t1 JOIN table2 t2 ON t1.id = t2.t1_id
WHERE t1.col1 = 0 AND t2.col2 IS NULL;
```

NB! Use database specific syntax, it might be faster!

Semi JOINs

You can use subqueries instead of **JOIN**s:

Useful Utility Functions

```
-- convert strings to dates:
TO_DATE (Oracle, PostgreSQL), STR_TO_DATE (MySQL)
-- return the first non-NULL argument:
COALESCE (col1, col2, "default value")
-- return current time:
CURRENT_TIMESTAMP
-- compute set operations on two result sets
SELECT col1, col2 FROM table1
UNION / EXCEPT / INTERSECT
SELECT col3, col4 FROM table2;
```

SQL基础知识

UPDATE语句

用于更新表中的数据。

DELETE语句

用于删除表中的记录。

CREATE TABLE语句

用于创建新表。

ALTER TABLE语句

用于修改已存在的表。

CREATE TABLE语句

- 用途：用于创建新表。
- 语法： CREATE TABLE 表名 (列名1 数据类型, 列名2 数据类型, ...);
- CREATE TABLE 用户 (id INT AUTO_INCREMENT PRIMARY KEY, 姓名 VARCHAR(100), 年龄 INT);
- 这个例子创建了一个名为“用户”的新表，包含“id”、“姓名”和“年龄”三个列。

SELECT语句

- 用途：用于从表中检索数据。
- 语法： SELECT 列名1, 列名2, ... FROM 表名;
- SELECT 姓名, 年龄 FROM 用户;
- 这个例子从“用户”表中选取了“姓名”和“年龄”两列的数据。

WHERE子句

- 用途：用于过滤满足特定条件的记录。
- 语法： SELECT 列名1, 列名2, ... FROM 表名 WHERE 条件;
- SELECT 姓名, 年龄 FROM 用户 WHERE 年龄 > 18;
- 这个例子选取了“用户”表中年龄大于18岁的记录。

INSERT INTO语句

- 用途：用于向表中插入新数据。
- 语法： INSERT INTO 表名 (列名1, 列名2, ...) VALUES (值1, 值2, ...);
- INSERT INTO 用户 (姓名, 年龄) VALUES ('李四', 25); 这个例子在“用户”表中插入了一个新的记录。

UPDATE语句

- 用途：用于更新表中的数据。
- 语法： UPDATE 表名 SET 列名1 = 值1, 列名2 = 值2, ... WHERE 条件;
- UPDATE 用户 SET 年龄 = 26 WHERE 姓名 = '李四';
- 这个例子更新了“用户”表中姓名为“李四”的记录的年龄。

DELETE语句

- 用途：用于删除表中的记录。
- 语法： DELETE FROM 表名 WHERE 条件;
- DELETE FROM 用户 WHERE 姓名 = '李四';
- 这个例子删除了“用户”表中姓名为“李四”的记录。

ALTER TABLE语句（添加、删除、修改列）

- 用途：用于修改已存在的表。
- 语法： ALTER TABLE 表名 ADD 列名 数据类型;

ALTER TABLE 表名 DROP COLUMN 列名;

ALTER TABLE 表名 MODIFY COLUMN 列名 数据类型;

- ALTER TABLE 用户 ADD 邮箱 VARCHAR(255);
- 这个例子在“用户”表中添加了一个名为“邮箱”的新列。

特殊符号

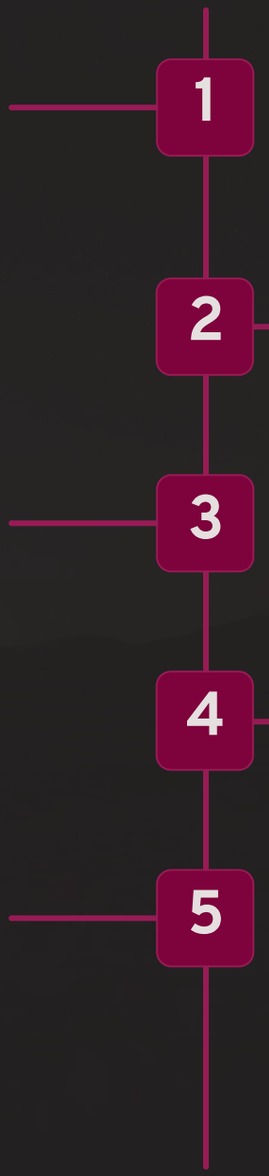
- *（星号）：在SQL查询语句中，*是一个通配符，用于表示选择表中的所有列。
  - `SELECT * FROM table_name;`
  - `*`意味着从指定的table_name表中选取所有的字段数据。
- %（百分号）：在SQL LIKE操作符中，%是通配符，代表零个或多个任意字符。
  - `SELECT column FROM table WHERE column LIKE '%pattern%';`
  - 会查找column列中包含"pattern"任何位置的行

特殊符号

- _（下划线）：与%类似，但在LIKE操作符中，_代表单个任意字符。
  - `SELECT column FROM table WHERE column LIKE 'prefix___';`
  - 将查找以"prefix"开头且后面跟三个任意字符的行。
- \\（反斜杠）：在SQL字符串和正则表达式中，反斜杠 \ 用来转义特殊字符。
  - 如果要匹配一个实际的反斜杠字符，需要写两个反斜杠 `\\`
- ^（插入符号）：在正则表达式中，^代表字符串的开始位置。
  - `SELECT column FROM table WHERE column REGEXP '^pattern';`
  - 会查找以"pattern"开头的字符串。
- **$**（美元符号）：在正则表达式中，$代表字符串的结束位置。
  - `SELECT column FROM table WHERE column REGEXP 'pattern$';`
  - 会查找以"pattern"结尾的字符串。

- **连接运算符**：
  - *= 和 =* 曾经在某些旧版数据库系统中用于表示外连接操作，但不是SQL标准的一部分。在现代SQL中，使用 LEFT JOIN, RIGHT JOIN 或 FULL OUTER JOIN 来进行外连接操作。
  - `SELECT * FROM table1 LEFT JOIN table2 ON table1.id *= table2.id;`
  - 这样的写法在非标准SQL中可能表示左连接，即返回`table1`的所有行以及与`table2`匹配的行，如果`table2`没有匹配项，则结果中的`table2`列将填充NULL。
- **其他符号和运算符**：
  - <, >, <=, >=, <>, !=, AND, OR, NOT, IN, BETWEEN等是比较和逻辑运算符，用于在WHERE子句和其他条件表达式中构建复杂的过滤条件。

SQL实操演示



安装SQLite

首先确保安装了SQLite，这是进入实操准备的第一步。

创建数据库

通过命令sqlite3 my_database.db创建一个新库或打开现有数据库。

创建表

使用CREATE TABLE语句在数据库中创建表。

插入数据

使用INSERT INTO语句进行数据插入操作。

查询数据

利用SELECT语句查询表中的数据。

SQL注入

SQL注入是一种常见的网络安全攻击手段，它利用了Web应用程序对用户输入数据处理不严的漏洞。当应用程序在构建动态SQL查询时直接拼接用户提供的数据，恶意用户可以通过提交精心构造的输入来改变原SQL语句的逻辑，从而获取、修改或删除数据库中的信息。

SQL注入

1

直接注入

攻击者通过输入字段直接插入恶意SQL，如输入 admin' OR '1'='1，可能导致身份验证失效。

```
SELECT * FROM users WHERE username = 'admin' OR '1'='1' AND password = '...'
```

2

注释注入

攻击者在输入中添加SQL注释符号绕过原有SQL语句的剩余部分，如输入 admin'--。

```
SELECT * FROM users WHERE username = 'admin'--' OR '1'='1' AND password = '...'
```

3

联合查询注入

攻击者使用UNION结合多个SELECT语句，获取不应访问的数据。

```
1 UNION SELECT username, password FROM users
```

SQL注入攻击类型

1

子查询注入

攻击者嵌入一个或多个额外的SELECT子查询到原SQL语句内部，以实现对数据库未经授权的读取或其他操作。

2

布尔盲注

攻击者发送构造好的SQL查询，通过观察应用程序的行为或响应内容，逐步推断数据库中的敏感信息。

3

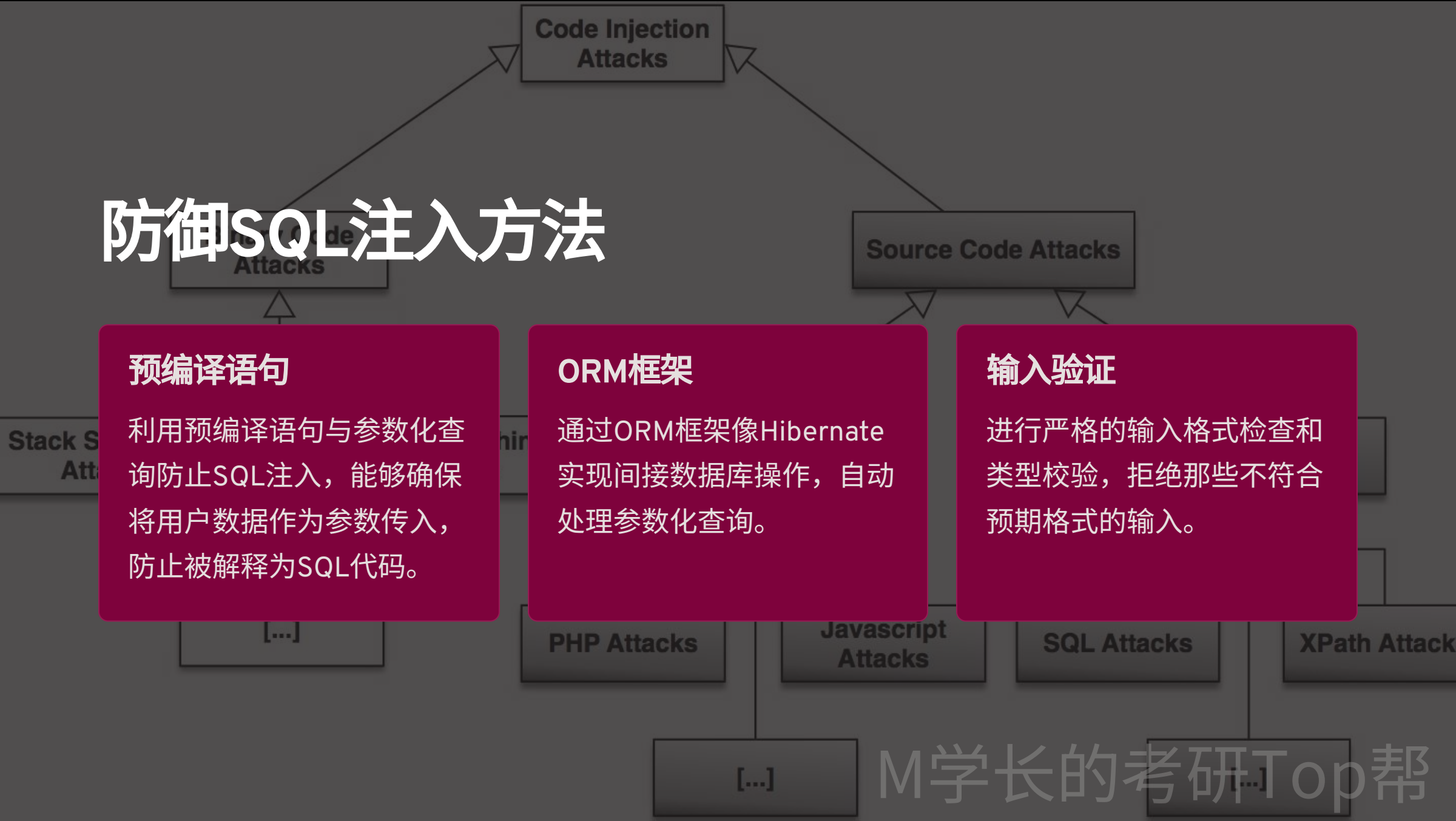
时间延迟注入

攻击者构造SQL查询，利用服务器响应的时间差异来间接推断数据库内的数据。

4

二次注入

攻击者将含有恶意SQL指令的内容注入到数据库存储的部分，然后在应用程序进行查询时触发这些恶意代码。



防御SQL注入方法

预编译语句

利用预编译语句与参数化查询防止SQL注入，能够确保将用户数据作为参数传入，防止被解释为SQL代码。

ORM框架

通过ORM框架像Hibernate实现间接数据库操作，自动处理参数化查询。

输入验证

进行严格的输入格式检查和类型校验，拒绝那些不符合预期格式的输入。

其他方法

1 **最小权限原则**

应用程序连接数据库所使用的账户应当仅拥有完成当前任务所需的最小子集权限，比如只读访问特定表，而非整个数据库的所有权限，更不能使用超级管理员账号。

2 **禁止直接拼接SQL**

绝对避免将用户输入直接插入到SQL字符串中形成新的查询，因为这种做法极易受到SQL注入攻击。

3 **使用安全函数**

对于不支持预编译语句或参数化查询的老旧数据库接口，可使用数据库自身提供的安全函数对特殊字符进行转义以防止SQL注入。

其他的攻击手段

DoS/DDoS攻击

会导致服务不可用，影响用户体验和业务运营。

攻击者通过发送大量无意义请求，耗尽服务器处理能力或网络带宽，使合法用户请求无法得到响应。

防御手段包括防火墙过滤、流量清洗和负载均衡。

跨站脚本攻击（Cross-Site Scripting, XSS）

XSS攻击通过注入恶意脚本到网页中，并利用受害者的浏览器执行这些脚本。分为持久型XSS、反射型XSS和DOM Based XSS三类。

可能导致用户会话劫持、敏感信息窃取等严重后果。

防御手段：过滤用户输入 输出内容转义

. 中间人攻击（Man-in-the-Middle Attack, MITM）

- MITM攻击者在通信双方之间拦截、篡改或监听数据，常见于不安全的Wi-Fi环境或SSL/TLS协议降级攻击中。
- 预防措施包括使用HTTPS加密通信、启用HSTS策略以及采用可信证书。

网络攻击的许多面貌

目录遍历（Directory Traversal）

攻击者利用Web应用程序漏洞查看并下载服务器上未经授权访问的文件

CSRF攻击

跨站请求伪造攻击串改用户请求，执行非授权操作。

文件包含漏洞

由于文件路径参数控制不佳，攻击者能够执行恶意代码。

1

零日漏洞利用（Zero-day Exploits）

针对软件或系统中存在的未知或未修复的安全漏洞发起攻击，开发者尚未发布补丁。

2

权限提升（Privilege Escalation）

攻击者通过发现并利用程序或系统漏洞从低权限账户获取更高权限。

3

弱口令破解与默认配置滥用

对服务器使用默认用户名/密码或易猜解的口令进行暴力破解，或者利用未更改的默认配置项进行攻击。

4

缓冲区溢出（Buffer Overflow）

当程序向内存缓冲区写入超过其预设大小的数据时，可能会覆盖相邻内存区域，从而可能执行任意代码。

5

点击劫持（Clickjacking）

一种视觉欺骗技术，让受害者在不知情的情况下点击一个隐藏在透明界面下的按钮或链接。

SQL注入攻击

1. 数据删除攻击： 在上述情况下，如果攻击者输入'; DROP TABLE users; --，则SQL语句变成：
   `SELECT * FROM users WHERE username = ''; DROP TABLE users; --' AND password = '...';`
2. 这会导致users表被删除。

防注入代码

```
// 准备SQL语句，预编译以防止SQL注入攻击
if (sqlite3_prepare_v2(db, sql.c_str(), -1, &stmt, nullptr) != SQLITE_OK) {
// 如果准备SQL语句失败，记录日志并返回false
LOG_INFO("Failed to prepare registration SQL for user: "username);return false;
}

// 绑定SQL语句中的参数，防止SQL注入
sqlite3_bind_text(stmt, 1, username.c_str(), -1, SQLITE_STATIC);
sqlite3_bind_text(stmt, 2, password.c_str(), -1, SQLITE_STATIC);
```

预编译SQL语句与参数绑定

预编译SQL语句（sqlite3_prepare_v2）

预编译SQL语句提高执行效率和增强安全性。

- 使用sqlite3_prepare_v2函数将SQL语句编译成预编译语句对象。
- 预编译过程类似于编译器处理源代码。
- 预编译固定SQL语句结构，防止SQL注入攻击。
- 预编译语句不受用户输入特殊字符影响，避免执行意外命令。

绑定参数（sqlite3_bind_text）

参数绑定是预编译语句的一个重要环节。

将值绑定到参数占位符上：

- 在预编译的SQL语句中，通常使用问号?作为参数的占位符。
- sqlite3_bind_text函数用于将实际的数据值绑定到这些占位符上。
- 绑定操作意味着指定的数据直接替换对应的占位符，但不会改变SQL语句的结构。

确保输入值的安全处理：

- 通过参数绑定，即使用户输入包含潜在的危险字符，这些输入也只被视为字符串数据。
- 数据库引擎在执行SQL语句时，会把这些绑定的值视为普通数据，而不是SQL指令。

总结：为什么预编译能防止 SQL 注入

- 通过预编译，SQL 语句结构在执行前已经确定，参数内容不能影响查询的逻辑；
- 参数值与 SQL 语句分开处理，DBMS 将参数视为纯数据，不会将它们解释为 SQL 代码；
- **绑定参数**的方式确保了即便输入恶意代码，也只是作为字符串数据处理，不会干扰 SQL 逻辑。

代码讲解

互联网开发面试常问问题

对比一下Redis和SQLite

常考：什么是Redis

什么是Redis

Redis（Remote Dictionary Server） 是一种开源的**内存数据库**，主要用于**高速缓存、消息队列、会话存储和实时数据处理**等场景。Redis 是基于 **键值对（Key-Value）** 形式存储数据，并且支持丰富的数据结构，如字符串、哈希、列表、集合、有序集合等，能够高效地执行各种操作

Redis 的特点

1. **高性能**：
   - Redis 是内存数据库，所有数据都保存在内存中，提供了非常高的读写性能，尤其适合实时数据处理场景。与磁盘存储的数据库相比，Redis 的读写延迟非常低。
2. **丰富的数据结构**：
   - Redis 支持多种数据类型，不仅可以存储简单的字符串，还可以存储哈希、列表、集合、有序集合、位图、HyperLogLog 等数据结构，适合多种应用场景。
3. **持久化机制**：
   - 虽然 Redis 是内存数据库，但支持数据的持久化，提供了 RDB 和 AOF 两种持久化机制，可以定期或实时地将数据保存到磁盘，以确保数据在系统重启后不会丢失。

特点	Redis	SQLite
类型	内存数据库、NoSQL 数据库	关系型数据库（RDBMS）
数据结构	键值对存储，支持多种数据类型（字符串、列表、集合等）	表结构，数据以行和列的关系型结构存储
性能	高性能内存操作，读写速度非常快	磁盘读写，速度相对较慢，但适合小型应用和嵌入式环境
查询语言	无 SQL，使用 Redis 提供的命令集	支持完整的 SQL 查询
事务支持	提供基本的事务支持，不具备严格的隔离级别	支持完整的事务（ACID 特性）
使用场景	高速缓存、会话管理、实时分析、消息队列、排行榜等	移动应用、本地存储、嵌入式设备的小型数据存储

什么是范式?

范式（Normalization） 是数据库设计中的一种规范化理论，用于指导数据库的结构设计。其主要目的是减少或消除数据冗余，避免数据更新、插入和删除时产生异常，从而提高数据库的性能和数据一致性。

范式通过一系列规则来约束数据库表的设计，这些规则被称为 **正常形式（Normal Form）**。每个范式都建立在前一个范式的基础上，逐步提高数据库的规范化程度

常见的数据库范式

数据库设计中常用的范式主要包括：

1. 第一范式（1NF）：确保表中的每个字段都是原子性的，即不可再分的最小数据单位。
2. 第二范式（2NF）：在满足第一范式的基础上，消除非主属性对主键的部分函数依赖，即每个非主属性完全依赖于主键。
3. 第三范式（3NF）：在满足第二范式的基础上，消除非主属性对主键的传递函数依赖，即非主属性不依赖于其他非主属性。

第一范式（1NF）：

定义：确保表中的每个字段都是原子性的，即每个字段只能存储一个值，不能被进一步拆分。

举例： 考虑一个包含学生信息的表：

学生ID	姓名	联系电话
1	张三	123456, 789012
2	李四	345678

问题

问题在于“联系电话”字段存储了多个电话号码，这不符合原子性。**修改**：将每个电话号码拆分成单独的行来满足1NF。

学生ID	姓名	联系电话
1	张三	123456
1	张三	789012
2	李四	345678

第二范式（2NF）：

定义：在满足1NF的基础上，确保每个非主属性都完全依赖于主键，而不是仅依赖于主键的一部分。这意味着表中不能有非主属性对主键的部分函数依赖。

举例：考虑一个课程注册表：

学生ID	课程ID	学生姓名	课程名称
1	101	张三	数学
2	102	李四	英语

在此表中，主键是复合主键（学生ID，课程ID）。但是，“学生姓名”仅依赖于“学生ID”，“课程名称”仅依赖于“课程ID”，这就存在**部分依赖**。**修改**：将表拆分为两个表，一个保存学生信息，一个保存课程信息。

学生表

学生ID	学生姓名
1	张三
2	李四

课程表

课程ID	课程名称
101	数学
102	英语

学生-课程注册表

学生ID	课程ID
1	101
2	102

第三范式（3NF）：

定义：在满足2NF的基础上，非主属性不能传递依赖于主键，即非主属性不能依赖于其他非主属性。

举例： 考虑一个订单表：

订单ID	客户ID	客户姓名	客户地址
1001	1	王五	北京市
1002	2	赵六	上海市

在这个表中，“客户姓名”和“客户地址”都依赖于“客户ID”，而“客户ID”依赖于“订单ID”。因此，这两个字段对“订单ID”是**传递依赖**。**修改**：将客户信息拆分成单独的客户表，以消除传递依赖。

修改表格

订单表

订单ID	客户ID
1001	1
1002	2

客户表:

客户ID	客户姓名	客户地址
1	王五	北京市
2	赵六	上海市

第三范式的重要性

遵循第三范式可以：

- **减少数据冗余**：避免重复存储相同的数据，节省存储空间。

- **消除更新异常**：防止在插入、更新、删除数据时出现不一致或异常。

- **提高数据完整性**：确保数据的准确性和可靠性。

第三范式的局限性

虽然第三范式有助于规范数据库设计，但过度的规范化可能导致：

- **性能下降**：由于表被分解，查询需要连接多个表，增加了查询的复杂度和开销。

- **过度拆分**：对于小型数据库，过度的范式化可能并不实际。

谢谢大家