目录

[一、计算机网络](#_Toc1185070750_WPSOffice_Level1) [3](#_Toc1185070750_WPSOffice_Level1)

[1、 HTTP协议](#_Toc1934327915_WPSOffice_Level2) [3](#_Toc1934327915_WPSOffice_Level2)

[1.1 URI和URL](#_Toc1934327915_WPSOffice_Level3) [3](#_Toc1934327915_WPSOffice_Level3)

[1.2 Cookie](#_Toc1641819119_WPSOffice_Level3) [3](#_Toc1641819119_WPSOffice_Level3)

[1.3 报文和实体](#_Toc1036552730_WPSOffice_Level3) [3](#_Toc1036552730_WPSOffice_Level3)

[1.4 HTTP状态码](#_Toc954388646_WPSOffice_Level3) [4](#_Toc954388646_WPSOffice_Level3)

[1.5 代理、网关、隧道](#_Toc854613879_WPSOffice_Level3) [5](#_Toc854613879_WPSOffice_Level3)

[1.6 HTTPS](#_Toc1124833217_WPSOffice_Level3) [5](#_Toc1124833217_WPSOffice_Level3)

[1.7 WebSocket](#_Toc773333578_WPSOffice_Level3) [7](#_Toc773333578_WPSOffice_Level3)

[1.8 防止SQL注入攻击](#_Toc846413802_WPSOffice_Level3) [8](#_Toc846413802_WPSOffice_Level3)

[二、 MySQL数据库](#_Toc1934327915_WPSOffice_Level1) [10](#_Toc1934327915_WPSOffice_Level1)

[1、 数据库和数据库实例](#_Toc1641819119_WPSOffice_Level2) [10](#_Toc1641819119_WPSOffice_Level2)

[2、 存储引擎简单对比](#_Toc1036552730_WPSOffice_Level2) [10](#_Toc1036552730_WPSOffice_Level2)

[3、InnoDB存储引擎](#_Toc954388646_WPSOffice_Level2) [11](#_Toc954388646_WPSOffice_Level2)

[3.1 InnoDB体系架构](#_Toc745092486_WPSOffice_Level3) [11](#_Toc745092486_WPSOffice_Level3)

[3.2 Checkpoint技术](#_Toc792266545_WPSOffice_Level3) [13](#_Toc792266545_WPSOffice_Level3)

[4、 索引与算法](#_Toc854613879_WPSOffice_Level2) [14](#_Toc854613879_WPSOffice_Level2)

[4.1 InnoDB支持的索引](#_Toc1225210415_WPSOffice_Level3) [14](#_Toc1225210415_WPSOffice_Level3)

[4.2 索引组织表](#_Toc2038237469_WPSOffice_Level3) [14](#_Toc2038237469_WPSOffice_Level3)

[4.3 B+树索引](#_Toc2145488186_WPSOffice_Level3) [15](#_Toc2145488186_WPSOffice_Level3)

[4.4 全文检索](#_Toc822025325_WPSOffice_Level3) [18](#_Toc822025325_WPSOffice_Level3)

[5、 锁](#_Toc1124833217_WPSOffice_Level2) [20](#_Toc1124833217_WPSOffice_Level2)

**一、计算机网络**

1. **HTTP协议**

**1.1 URI和URL**

URI：统一资源定位符，Uniform Resource Identifier。

Uniform：规定统一的格式处理不同类型的资源

Resource：可标识的任何资源

Identifier：用于标识资源

URI就是由某个协议方案表示的资源的定位标识符，用字符串标识互联网资源。而URL表示资源的地点。URL是URI的子集。

**1.2 Cookie**

HTTP协议是一种不保存状态的协议，即无状态协议。HTTP协议自身不对请求和响应之间的通信状态进行保存（对发送过的请求或者响应都不做持久化处理），这也就意味着无法根据之前的状态进行本次请求的处理（无法进行用户识别和状态管理），因此引入了Cookie技术。

由于HTTP协议是无状态的，所以HTTP协议本身非常简单，可以使得服务更快的处理请求以及响应，确保协议的可伸缩性。

**1.3 报文和实体**

报文：HTTP通信的基本单位，8位组字节流，由报文首部和报文主体组成。我理解的是其作为HTTP协议中的一种概念，或者一个承载数据的容器，是属于协议范畴的。

实体：作为请求或者响应的有效载荷数据，其内容有实体首部和实体主体组成。我理解的是其作为真实的实际上的被传输的数据，是属于实际应用范畴的。

报文的主体是用于传输请求或响应的实体主体的，通常报文主体就等同于实体主体。

**1.4 HTTP状态码**

1.4.1 1×× 接收的请求正在处理

101 Switching Protocols，切换协议，可用于实现WebSocket通信

1.4.2 2×× 请求成功处理

200 OK：请求已经被正常处理了

204 No Content：请求处理成功，但返回的响应报文不包含实体主体，也不允许包含实体主体。

206 Partial Content：范围请求，响应报文中包含了由请求首部Content-Range指定范围的实体内容

1.4.3 3×× 重定向

301 Moved Permanently：永久性重定向

302 Found：临时性重定向

303 See Other：与302类似，但明确表示使用GET方法进行重定向

304 Not Modified：客户端发送了附带条件的请求（If-Modified-Since等），服务器资源并未改变，可直接使用客户端缓存

307 Temporary Redirect：类似302，虽然标准规定了302禁止在重定向时将方法从POST改成GET，但实际上并没有严格遵守，而返回307则会遵照标准，不会从POST改成GET。

1.4.4 4×× 客户端错误

400 Bad Request：请求报文存在语法错误

401 Unauthorized：请求未携带认证信息

403 Forbidden：禁止访问

404 Not Found：未找到资源

405 Method Not Allowed

1.4.5 5×× 服务器错误

500 Internal Server Error：服务器处理请求时出错

503 Service Unavailable：服务器超载或维护中

1.5 代理、网关、隧道

代理：转发而不会修改URI，进行缓存以及访问控制，缓存服务器是代理服务器的一种

网关：类似代理，但网关能使通信线路上的服务器提供非HTTP协议的服务。网关和服务器之间可以使用HTTP协议之外的协议，或者说利用网关可以由HTTP请求转化为其他协议通信。

隧道：通过隧道与远程服务器建立安全的通信线路。

**1.6 HTTPS**

1.6.1 HTTPS = HTTP + 加密 + 认证 + 完整性验证

加密：防窃听；认证：防伪装；完整性校验：防篡改

所谓HTTPS协议，就是在HTTP协议与TCP协议之间增加了一层SSL协议，采用对称密码体系和公钥密码体系组合的混合加密方式，在交换密钥时使用公钥进行加密传输，之后的HTTP通信使用对称密钥进行加密。

1.6.2 建立SSL连接：

1. 第一阶段SSL握手：

客户端发送：Client Hello，开始SSL通信，报文中包含客户端支持的SSL的版本以及加密组件

服务器发送：Server Hello，表示可进行SSL通信，报文中同样包含支持的SSL版本以及从接收到的客户端的加密组件中筛选出的加密组件

服务器发送：Certificate，报文中包含公钥

服务器发送：Server Hello Done，第一阶段SSL握手结束

1. 第二阶段SSL握手

客户端发送：Client Key Exchange，报文中包含已经使用公钥加密的Pre-master secret随机密码串

客户端发送：Change Cipher Spec，提示服务器此报文之后的通信会采用Pre-master secret密钥加密（其实是使用通过Pre-master secret生成的master secret来进行加密）

客户端发送：Finished，报文中包含连接至今全部报文的整体校验值。第二阶段SSL握手是否成功，要看服务器是否可以正确解密该报文

1. 第三阶段SSL握手

服务器发送：Change Cipher Spec

服务器发送：Finished

1. 进行HTTP通信

服务器和客户端的Finished报文交换完成之后，SSL连接就算建立完成，可以开始应用层协议的通信。应用发送数据时会附加一种MAC (Message Authentication Code) 报文摘要，用以保护报文完整性

1. 断开SSL连接

客户端发送：Alert warning, close\_notify

**1.7 WebSocket**

1.7.1 HTTP的瓶颈

（1）一条连接上只能发送一个请求？

（2）请求只能从客户端开始，客户端不能接受除响应以外的指令

（3）请求/响应熟不未经压缩就发送

（4）每次都要发送相同的冗长的首部

（5）虽可以任意选择数据压缩格式，但并非强制压缩

1.7.2 实现WebSocket通信

WebSocket即web浏览器与服务器之间进行全双工通信的标准。WebSocket是建立在HTTP基础上的协议，连接的发起者仍是客户端，但是之后可以确立WebSocket通信连接，一旦建立，不论服务器还是客户端都使用该协议进行通信，可向对方发送任意格式的数据。

WebSocket的主要特点就在：（1）推送功能，支持服务器向客户端推送数据；（2）减少通信量，只要建立起WebSocket连接，就可以一直保持连接状态，首部信息也很少。

WebSocket握手：

1. 请求，需要使用HTTP的Upgrade首部：

Upgrade：websocket

Connection: Upgrade

Sec-WebSocket-Key: ×××

Sec-WebSocket-Version: 13

Sec-WebSocket-Key字段为握手过程必不可少的键值

1. 响应，对上述请求返回状态码101 Switching Protocols：

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: ×××

Sec-WebSocket-Accept是由握手请求中的Sec-WebSocket-Key生成的

（3）WebSocket通信，成功握手建立WebSocket连接之后，通信时不再使用HTTP数据帧，而是使用WebSocket独立的数据帧

**1.8 防止SQL注入攻击**

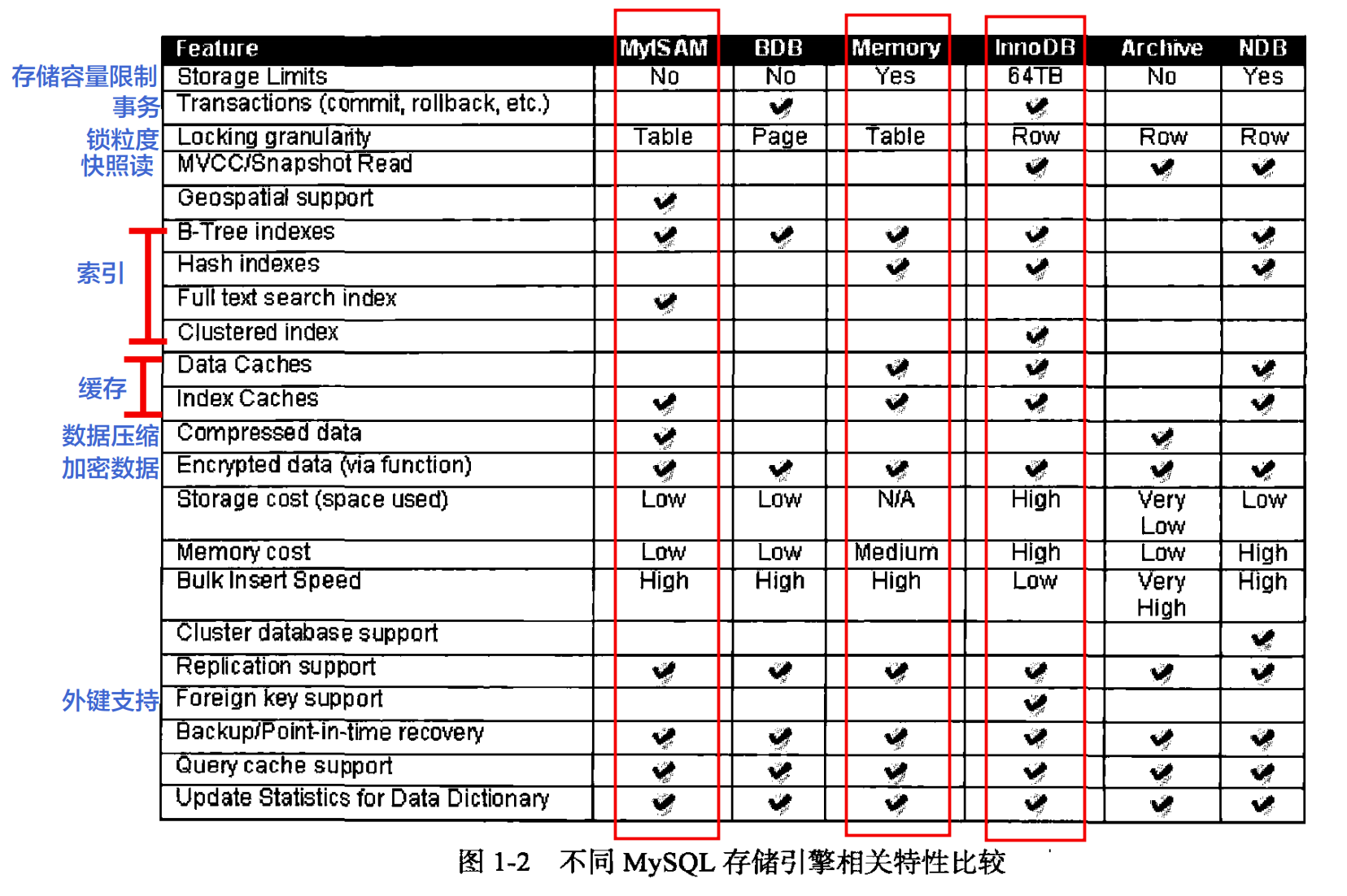
1. 永远不要相信用户的输入，对输入参数进行严格校验，需要符合固定的格式和类型
2. 过滤特殊的符号，对一些特殊符号进行过滤或者转义
3. 不要拼接SQL语句，而是使用参数化的SQL语句，或者说预编译的SQL语句，使用？来代替变量，预编译的SQL语句其语义和结构不会改变
4. 应用的异常信息应该给出尽可能少的提示，最好使用自定义的错误信息对原始错误信息进行包装
5. 控制数据库的访问权限，只提供有限的访问权限
6. 不要将机密数据或者敏感数据直接存放数据库，而是进行加密或者hash计算之后存入数据库
7. **MySQL数据库**
8. **数据库和数据库实例**

数据库：物理文件的集合，依照某种数据模型组织起来的数据集合。

数据库实例：数据库实例是程序，是数据库管理软件，对数据库数据的任何操作都是通过数据库实例来完成的。数据库实例才是真正用于操作数据库文件的。

1. **存储引擎简单对比**

存储引擎是基于表的，而不是数据库。

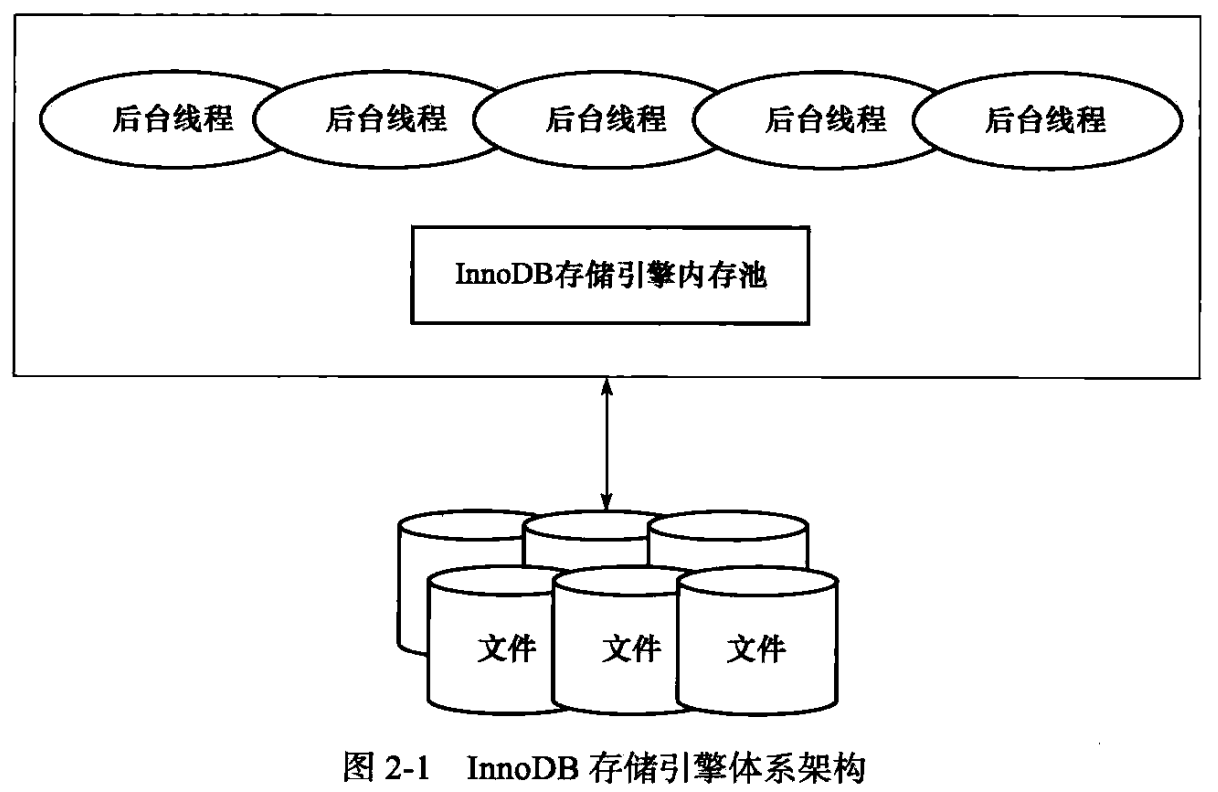


* InnoDB：支持事务，面向在线事务处理（OLTP）应用，行锁设计，非锁定读，聚集索引，支持外键。InnoDB从1.2.x开始支持全文检索，并支持MyISAM的全部功能。
* MyISAM：不支持事务，面向在线分析处理（OLAP），表锁设计，支持全文索引，只缓存索引而不缓存数据，支持压缩数据。
* Memory：表中数据保存在内存中，表锁设计，默认使用哈希索引，使用Memory存储引擎表来存放查询的中间结果集。

**3、InnoDB存储引擎**

**3.1 InnoDB体系架构**

3.1.1 InnoDB存储引擎组成：



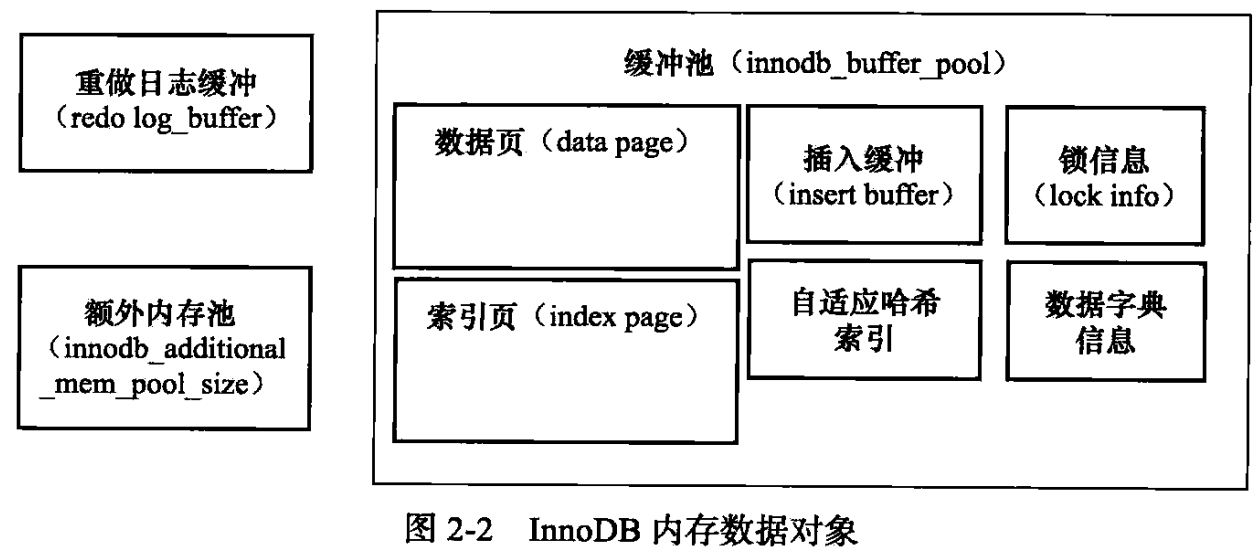
* 后台线程：刷新内存池中的数据，将已修改的数据刷新回磁盘文件
* 内存池：多个内存块组成的内存池
  + 维护多个进程/线程访问的内部数据结构
  + 缓存磁盘上的数据，缓存对磁盘文件的修改
  + 重做日志（redo log）缓存

3.1.2 后台线程

* Master Thread：将缓冲池中的数据异步刷新到磁盘，保证数据的一致性

3.1.3 内存

1. 缓冲池



* 读取页：将从磁盘中读取的页放在缓冲池中，称为将页FIX在缓冲池中。下一次读取相同的页，若该页在缓冲池中，称该页在缓冲池中被命中。
* 修改页：首先修改缓冲池中的页，然后以一定频率刷新到磁盘上（Master Thread）。
* 缓存的数据页：索引页，数据页，undo页，插入缓冲，自适应哈希索引，锁信息，数据字典信息

1. LRU List、Free List和Flush List
2. LRU List

InnoDB存储引擎使用LRU算法对缓冲池中的页进行管理，但引入了midpoint：新读取到的页并不直接放到LRU List的首部，而是放到LRU List的midpoint位置，称为midpoint插入策略。默认midpoint在LRU List长度的5/8处，midpoint之前的5/8称为old列表，midpoint之后的3/8称为new列表。

为什么要引入midpoint？当进行索引或数据的扫描操作时，需要临时访问表中的许多页，如果将这些大量的临时非活跃的页插入LRU列表的首部，会导致很多热点数据页被从LRU列表中移除，降低了缓冲池中的命中率，导致缓冲池LRU列表被**污染**。

另外，InnoDB还引入了innodb\_old\_blocks\_time参数，表示页读取到midpoint后需要等待多久才会被加入到LRU列表的new端。很容易理解，新读取的页首先放到midpoint位置，在规定的时间内如果仍然在LRU列表而没有被挤出，说明该页相对来说属于热点数据而被频繁读取。

1. Free List

LRU列表用来管理已经读取的页，当数据库刚启动时，LRU列表为空，此时页都存放在Free列表中。

1. 脏页

当LRU列表中的页被修改之后，该页就称为脏页，即缓冲池中的页和磁盘中的页数据不一致。数据库会根据checkpoint机制将脏页刷新回磁盘，而Flush List中的页就是脏页列表。要注意，脏页既存在于LRU列表，也存在Flush List中。LRU列表用于管理缓冲池中页的可用性，而Flush列表用于管理刷新回磁盘的页，并不冲突。

**3.2 Checkpoint技术**

Write Ahead Log策略：当事务提交时，先写重做日志，再修改页，当发生宕机时，可通过重做日志来完成数据恢复。

Checkpoint所做的事总之就是将缓冲池中的脏页刷新回磁盘，但是每次刷新多少页、每次从哪里读取脏页以及何时触发Checkpoint都非常复杂。

Sharp Checkpoint：当数据库关闭时将所有脏页刷新回磁盘。

Fuzzy Checkpoint：只刷新一部分脏页。

1. **索引与算法**

**4.1 InnoDB支持的索引**

* B+树索引
* 全文索引
* 哈希索引

InnoDB支持的哈希索引是自适应的，InnoDB会根据表的使用情况自动为表生成哈希索引，不能人为干预是否在一张表中生成哈希索引。InnoDB会自动根据访问的频率和模式来自动为热点页建立哈希索引，生成的哈希索引是通过缓冲池的B+树构造而来，不需要对整张表构建哈希索引。自适应哈希索引要求对某个页的连续访问模式是一样的，即查询的条件必须一模一样。不过，哈希索引只支持等值查询，范围查找是不能使用哈希索引的。

**4.2 索引组织表**

在InnoDB中，表都是根据主键顺序存放的，这种存储方式的表称为索引组织表。在InnoDB存储引擎表中，每张表都有个主键（\_rowid可以显示表的主键值）。如果创建表时没有显式定义主键，InnoDB会按如下方式选择或创建主键：

* 如果表中有非空的唯一索引（Unique Key Not Null）则以该列为主键。当表中有多个非空唯一索引时，将选择建表时第一个定义的非空唯一索引为主键（是非空唯一索引的定义顺序，而不是建表时列的顺序）
* 如果没有非空唯一索引，则自动创建一个6字节的指针。

**4.3 B+树索引**

4.3.1 聚集索引

聚集索引就是按照索引组织表的主键构造一棵B+树，同时叶子节点存放的即是整张表的行记录数据，其叶子节点也称为数据页。所以，索引组织表中的数据也是索引的一部分。由于实际的数据页只能按照一棵B+树进行排序，因此每张表只能拥有一个聚集索引。

聚集索引能够在B+树的叶子节点上直接找到数据，而且由于定义了数据的逻辑结构，聚集索引的范围查找和排序查找速度非常快。

聚集索引的存储并不是物理上连续的，而是逻辑上连续的。逻辑上连续指的是，叶子节点按照主键顺序排列并通过双向链表连接，同时，叶子节点中的记录也是通过双向链表连接的。

4.3.2 辅助索引（非聚集索引）

辅助索引的叶子节点并不包含行记录的数据，除了键值外，还有一个指向索引对应行数据的书签，也就是相应行数据的聚集索引键。

辅助索引并不影响聚集索引的组织，每张表可以有多个辅助索引。当通过辅助索引查找数据时，InnoDB会根据辅助索引找到叶子节点，由此获得主键索引的主键，在通过主键聚集索引找到行记录。

4.3.3 B+树索引的分裂

要注意的是，B+树索引的分裂跟单纯的B+树分裂还是有区别的。B+树索引页的分裂并不总是从页的中间记录开始，否则可能会导致页空间的浪费。若插入的是随机的，则仍然取页的中间记录作为分裂点。但是插入往往是根据自增顺序插入的，如果依然按照中间记录作为分裂点，则分裂后左边的页可能永远不会有记录再插入，造成空间的浪费。

InnoDB可以决定向左还是向右分裂，同时决定分裂点位置。如果欲插入位置右边没有记录，则该待插入记录就是分裂点，即分离出的页只有待插入的记录。否则也会使得分裂出的节点拥有比较少的记录。

4.3.4 Fast Index Creation

MySQL数据库对索引的添加和删除，首先创建一张临时表，临时表的定义即为原表创建或删除索引之后的结构，然后将原表数据导入临时表，删除原表，并将临时表重名为原表名。在这一过程中，MySQL数据库服务不可用。

对于辅助索引的添加，InnoDB支持Fast Index Creation（FIC），会对创建索引的表加上一个S锁，在创建索引的过程中只允许读操作，写操作依然不可用。但是创建过程中，不需要重建表，速度更快。

删除辅助索引，将辅助索引的空间标记为可用（就是清除辅助索引空间的意思？），同时删除MySQL数据库内部视图上对该表的索引定义即可。

FIC方式只限定于辅助索引，主键索引的创建和删除同样需要重建表。

4.3.4 B+树索引的适用场景

字段的取值范围很小，称为低选择性，相反，取值范围很广，属于高选择性。

当访问表中很少一部分，字段属于高选择性时，就非常适合添加B+树索引。

通过SHOW INDEX FROM tb\_name可以看到表中所有索引列的信息，每个索引会有一个Cardinality属性（这个属性是针对索引列的），表示索引中不重复记录数量的预估值。实际应用中，Cardinality/n\_rows\_in\_table应尽可能接近1，那么表示该索引列具有高选择性，索引的创建就非常有价值。

Cardinality表示索引列的选择性。InnoDB对Cardinality的更新时机发生在两个操作上：INSERT和UPDATE，但并非每次操作就更新，而是：

· 表中1/16的数据发生过变化

· 某一行数据频繁地更新超过某个值

时才会更新Cardinality。而对Cardinality的统计是采用随机采样的方式：随机取B+树的8个叶子节点，统计每页不同记录的个数，再预估整个表中的不同记录数。每次对Cardinality值的统计，可能会有不同的结果，也可能都一样（表足够小，叶子节点没超过8个）。

4.3.5 覆盖索引

覆盖索引/索引覆盖，即从**辅助索引**中就可以得到查询的记录，而不需要查询聚集索引中的记录。辅助索引不包含整行记录的信息，远小于聚集索引，可以减少IO操作。

我的理解就是，覆盖索引包括两个方面：where条件中的列必须是辅助索引列以及主键，其次查询的数据也必须是辅助索引列以及主键，也即：查询条件覆盖，查询字段覆盖（指被辅助索引列覆盖）

InnoDB中的辅助索引包含了主键，因此在仅针对主键和辅助索引列的查询时，仅使用一次辅助索引即可完成查询。

另外覆盖索引也可以提高统计查询的速度。由于辅助索引远小于聚集索引，通过辅助索引可以减少IO操作，进而快速的进行统计，而不是通过聚集索引。而且，通常情况下，对于联合索引，针对**非**最左前缀的查询，是不能通过联合索引来查询的。但是如果是统计操作，并且是覆盖索引，优化器依然会选择联合索引。（当最左前缀相同时，中间索引列是排序的）

· 扫描聚集索引：即扫描全表

· 查找聚集索引：根据主键进行查找

当查询的数据包含整行的部分或全部信息时，不能仅通过辅助索引进行索引覆盖时，优化器也可能还是会选择辅助索引，这通常是查询的数据较少的情况下。这时意味着通过辅助索引查找到数据后还需要进行书签查找，即通过主键查找聚集索引。

上面这一段理解一下。如果某个查询条件包含辅助索引列，且需要查询整行信息，此时就不是覆盖索引。但是优化器依然选择了辅助索引，这前提是通过辅助索引查找到的数据量很少。如果数据量很多（>全表的20%），那么优化器会直接扫描聚集索引，也就是进行全表扫描（因为查询条件并不包含主键，如果包含主键的话就直接根据主键查找聚集索引而不是全表扫描，就没这么多事了）。

**4.4 全文检索**

4.4.1 倒排索引（inverted index）

全文检索通常使用倒排索引来实现。倒排索引也是一种索引结构，存放在辅助表（Auxiliary Table）中，存储了单词与单词自身在一个或多个文档中所在位置之间的映射，有两种表现形式：

· inverted file index：{单词，单词所在文档ID}

· full inverted file index：{单词，{单词所在文档ID，在文档中的位置}}

4.4.2 InnoDB全文检索（Full Test Search, FTS）

InnoDB从1.2.x开始支持全文检索，采用full inverted index，即{word, {DocumentId, Position}}，将{DocumentId, Position}视为一个ilist，并且在word字段上设有索引。InnoDB中，存放倒排索引的辅助表共有6张，这种表是持久的表，存放在磁盘上。

InnoDB中还有FTS Index Cache（全文检索索引缓存），是一个红黑树结构，根据（word，ilist）进行排序。当插入新的数据时，InnoDB并不会马上更新辅助表，而是先放在FTS Index Cache中，然后批量更新全文索引。当进行全文检索时，会先将FTS Index Cache中的全文索引合并（同步/刷新）到辅助表，然后再进行查询。

这里所说的“单词所在文档”，或者DocumentId中的Document就是进行全文检索的字段值，即一行数据中的TEXT格式的数据。对于支持全文检索的表都有一个关键的FTS\_DOC\_ID列（类型必须为BIGINT UNSIGNED NOT NULL），用来与word进行映射，InnoDB会自动为FTS\_DOC\_ID添加唯一索引。

当文档中发生删除操作时，并不会删除辅助表中的记录，而只是删除FTS Index Cache中的记录，并将其保存到DELETED Auxiliary table中（只记录FTS Document ID）。

为表创建全文检索索引时，先导入数据，再进行倒排索引（全文索引）的创建，是比较推荐的一种方式。

InnoDB的全文检索有以下限制：

· 每张表只能有一个全文检索的索引；

· 多列组合的全文检索索引列必须使用相同的字符集与排序规则

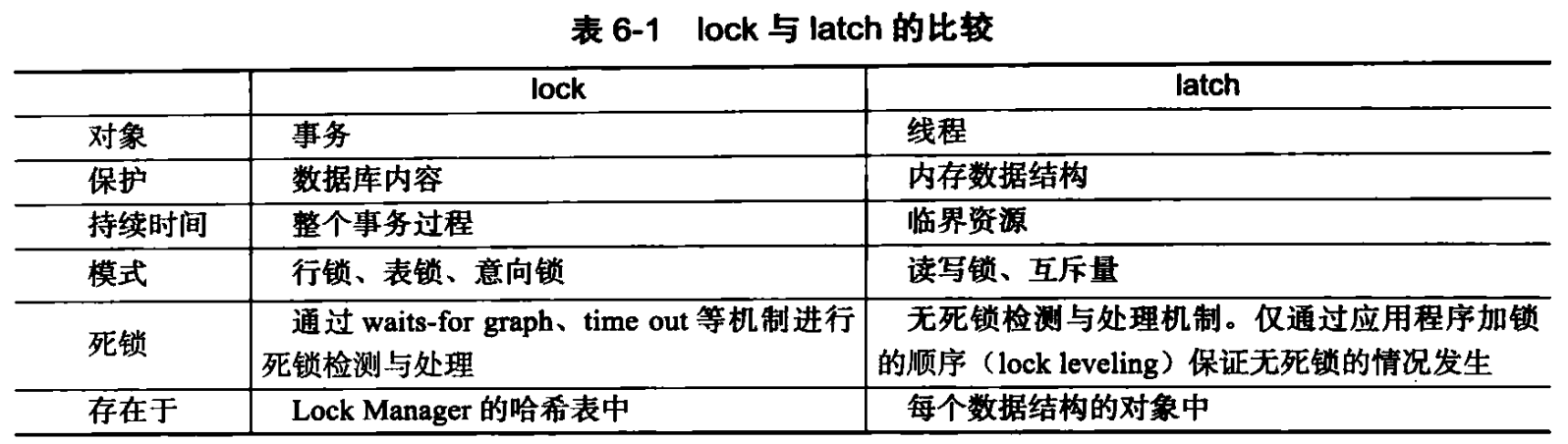
· 不支持没有单词界定符的语言，比如中文日语韩语

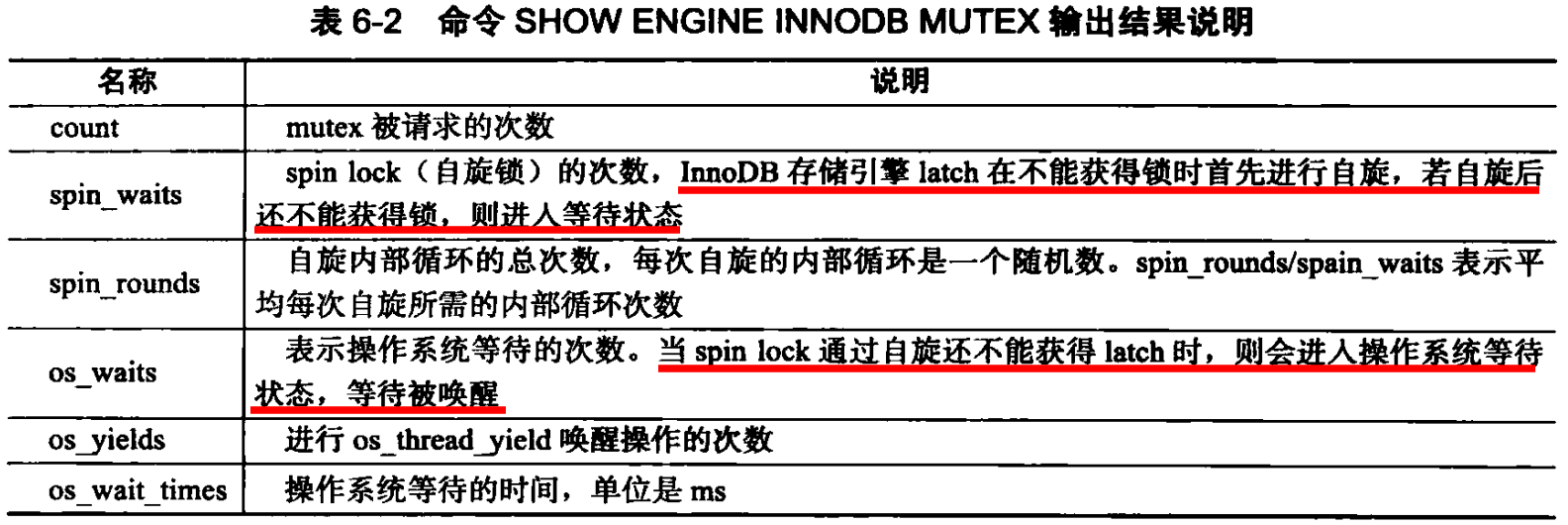
1. **锁**

**5.1 lock和latch**

latch为轻量级的锁，在InnoDB中分为mutex互斥量和rwlock读写锁。其目的是用来保证并发线程操作临界资源（critical area）的正确性，并且通常没有死锁检测。

lock的对象是事务，用来锁定数据库中的对象，如表、页、行。lock的对象仅在事务commit或rollback之后进行释放。并且lock有死锁机制。





**5.2 InnoDB存储引擎中的锁**

5.2.1 锁的类型

* 共享锁（S Lock），允许事务读取一行数据
* 排他锁（X Lock），允许事务删除/更新一行数据

S锁之间可以兼容，而X锁与任何锁都不兼容。S和X锁都是行锁，兼容指的是对同一行记录锁的兼容性。

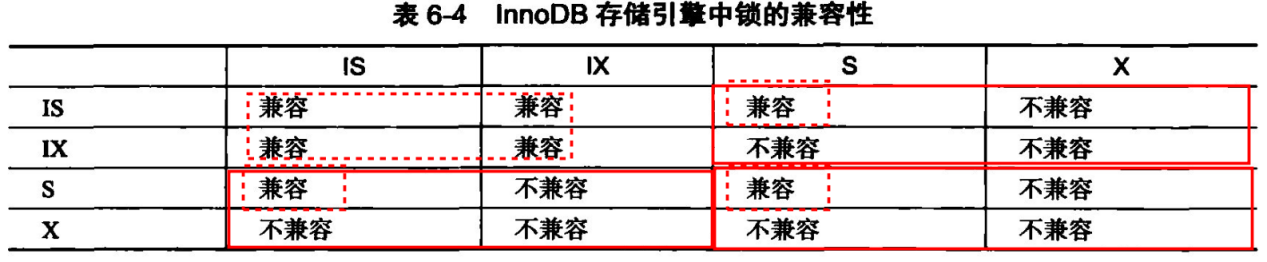
InnoDB支持多粒度的锁，同时支持表级锁和行级锁。那么粗粒度的锁和细粒度的锁之间就会存在冲突。因此InnoDB增加了意向锁，它实际上是一种表锁。InnoDB支持两种意向锁：

* 意向共享锁（IS Lock），表示事务想要或已经获得表的某几行的共享锁
* 意向排它锁（IX Lock），表示事务想要或已经获得表的某一行排他锁

当对最细粒度的对象（行）加锁时，首先需要对粗粒度的对象（表）上锁，即当需要对行加上X锁时，需要先对数据库、表、页加上意向锁IX，然后再对行加上X锁。

理解一下意向锁，它的出现是为了解决表锁和行锁的冲突问题。因为InnoDB同时支持行级锁和表级锁，假设事务A获得行S/X锁，事务B获得表锁（ALTER TABLE/DROP TABLE/LOCK TABLE)，那么事务B理应可以修改所有行，这与事务A的行S/X锁冲突。所以，在加表锁时，不仅应当判断此时表是否有其他表锁，还要判断是否有行级锁，那么需要遍历全部行来判断是否有行级锁。反之，在加行锁时不仅要判断当前行是否有其他行级锁，也需要判断锁定行所在表是否有表级锁。基于此，引入了意向锁。当要加上行级S/X锁时，数据库会**自动**对表加上相应的意向锁IS/IX，表示这张表即将或已经加上了行级锁。而意向锁也是一种表级锁，它会与其他的表锁冲突，因此可以阻塞其他的表锁请求，或者阻塞自己的意向锁请求并因此阻塞相应行级锁的请求，避免粗粒度锁和细粒度锁的冲突。

但是，意向锁之间并不会冲突。因为意向锁是在加上行级锁之前所加的锁，而行级锁之间的兼容问题的前提是，请求锁定的是同一行。所以，假设事务A请求行S锁，那么会首先加上意向IS锁，事务B同时请求X锁，会首先加上意向IX锁，意向锁之间是兼容的，因此IS和IX可以共存。如果事务A锁定的行和事务B锁定的行并不是同一行，那么它们之间就不存在兼容冲突问题。如果是同一行，那么就按照行级锁的兼容规则，阻塞其中一方。



这张图如何理解S/X锁和IS/IX之间的兼容？我理解的是，这里的S/X锁和意向锁IS/IX之间的兼容或不兼容，应该指的是表级S/X锁和意向锁IS/IX之间的兼容问题。因为意向锁本身就是在加上行级S/X锁之前加的表级锁，因此行级S/X锁和意向锁IS/IX肯定是不冲突的。这里与意向锁IS/IX的兼容问题应该指的是表级S/X锁。（但是前面不是说S/X锁是行级锁吗？应该是有表级S/X锁）

5.2.2 一致性非锁定读

一致性非锁定读是通过**版本控制**实现的：当读取的行正在执行DELETE/UPDATE，即被加上了行X锁时，读取操作无需等待行X锁的释放（也就是不加行S锁的读），而是读取行的一个快照数据。

快照数据指的是行之前的版本数据，也就是有关该行的事务提交一次，就会产生一个该行的历史版本。快照是通过undo段来实现的，而undo是用来在事务中回滚的，因此快照数据本身并没有额外的开销。并且，读取快照数据不需要上锁，因为没有事务是需要对历史数据进行修改操作的。

一致性非锁定读是InnoDB的默认读取方式，即读取不会占用和等待表上的锁。但是在不同事务隔离级别下，读取的方式不同，而且并不是在每个事务隔离级别下都是采用非锁定的一致性读（InnoDB默认隔离级别是可重复度）。即使是使用非锁定的一致性读，读取快照的方式也都不一样。

快照数据就是当前行数据之前的历史版本，每行记录可以有多个版本。这种通过行多版本技术来控制并发的手段，称之为多版本并发控制。

READ COMMITTED和REPEATABLE READ下，InnoDB都使用非锁定的一致性读。但是READ COMMITTED总是读取读取被锁定行的最新一份快照数据。而REPEATABLE READ总是读取事务开始时的行数据版本。

5.2.3 一致性锁定读

在默认情况下，事务隔离级别为REPEATABLE READ，InnoDB的SELECT操作都使用一致性的非锁定读。但是在某些情况下，需要显式地对数据库读取操作进行加锁，以保证数据逻辑的一致性。InnoDB对SELECT语句支持两种一致性的锁定读：

* SELECT...FOR UPDATE：对读取行记录加一个X锁。在UPDATE和DELETE操作中就隐含了SELECT操作，因此需要加上一个排它锁
* SELECT...LOCK IN SHARE MODE：对读取的行加一个S锁。外键值的插入或更新，需要先查询父表，此时使用的不是非锁定读，而是这种锁定模式，即对父表中记录添加了S锁。

这两种锁都必须在一个事务内，事务提交后释放锁。

**5.3 锁的算法**

5.3.1 行锁的三种算法

* Record Lock：单个行记录的锁。
* Gap Lock：间隙锁，锁定一个范围，不包含记录本身
* Next-Key Lock：以上两者的组合，锁定一个范围，包括记录本身。

Record Lock总是去锁定索引记录，如果有多个索引，则需要分别进行锁定。如果InnoDB表建立时没有设置索引，会使用隐式的主键来锁定。这是Read Committed事务隔离级别下的锁

Next-Key Lock技术主要用于解决幻行问题， Gap Lock的作用是为了阻止多个事务将记录插入到同一范围内，导致幻行问题的产生。这是Repeatable Read事务隔离级别下的加锁方式

但是，当查询的索引含有**唯一属性**时，即查询的列是**唯一索引**，并且是对该唯一属性的**等值查询**时，会从Next-Key Lock降级为Record Lock，只锁定索引本身，而不包括范围。因为查询的值具有唯一性，其他事务的插入绝对不会影响对该唯一属性的等值查询结果。因此，对聚集索引的锁定，只使用Record Lock，而对辅助索引，会使用Next-Key Lock，并且还会对辅助索引的下一个键值加上Gap Lock。

5.3.2 幻行问题

幻行问题指的是在同一事务下，连续执行两次同样的查找，第二次可能返回之前不存在的行，而这个行是此事务期间由其他事务插入的，这就违反了事务的隔离性，即当前事务能够看到其他事务的结果。

InnoDB默认的事务隔离级别是REPEATABLE READ，采用Next-Key Lock方式来加锁，可以避免出现幻行。而READ COMMITTED隔离级别下仅采用Record Lock。

如果通过索引查询一个值，并对该行加上了S Lock，即使查询的数据不存在，锁定的也是一个范围。

5.3.3 理解间隙锁

首先Gap Lock或Next-Key Lock都是一种行锁策略，针对的都是对行的锁定，因此版本控制的快照读不在讨论范围内。

其次，S Lock和X Lock之间的兼容，针对的是SELECT和UPDATE/DELETE之间的冲突，避免对同一行的查询和 修改/删除 同时进行。

而间隙锁Gap Lock/Next-Key Lock，针对的是SELECT和INSERT之间的冲突，解决的是在事务内多次查询的结果由于其他事务的插入而导致结果前后不一致的问题。

**5.4 锁问题**

5.4.1 不可重复读（幻行）

不可重复读指的是一个事务内多次读取同一数据集合，由于其他事务的提交，而导致前后读取的数据不一样。在MySQL官方文档中，**将不可重复读的问题定义为幻读问题**（Phantom Problem）。

InnoDB存储引擎中，通过Next-Key Lock来避免不可重复读问题。Next-Key Lock不仅锁定索引，还锁住索引覆盖的范围，在这个范围内的插入都是不允许的。

我的理解是，多版本并发控制实际上也可以避免幻行的问题。当事务内的查询是一致性的非锁定读时，那么就默认使用多版本并发控制来读取快照，也可以避免不可重复读的问题。当需要一致性的锁定读时（SELECT...FOR UPDATE/SELECT...LOCK IN SHARE MODE），就需要采用Next-Key Lock算法，不仅施加S Lock/X Lock锁定记录本身，还需要施加间隙锁，锁住一定的范围。

另外，幻行问题，或者说不可重复读问题，跟查询的数据是多行记录还是单行记录无关。因为，如果是查询单行记录，也有可能查询的值不存在，第二次查询时却存在了；或者其他事务插入了同样的相等记录，第二次查询的行数变多了，这都是幻读的问题。只要是涉及一致性锁定读的情况下，事务需要读取实际的行记录，那么就需要使用Next-Key Lock算法来避免幻读的问题。

**5.5 死锁**

5.5.1 阻塞

由于锁的兼容问题，一个事务请求的锁需要等待另一个事务中的锁释放后才能获得，这就是阻塞。阻塞并不是一件坏事，其保证了事务并发且正常运行。

在InnoDB中，innodb\_lock-wait\_timeout参数用来控制阻塞等待的时间，当发生超时时会抛出一个1205异常。innodb\_rollback\_on\_timeout参数用来设定是否在等待超时时会进行中的事务进行回滚。

在默认情况下，InnoDB不会回滚超时引发的异常。实际上，不仅是阻塞超时引发的异常，InnoDB在大部分下都不会对任何异常进行回滚，但是**死锁除外**。当发生异常时，需要明确判断是否需要提交还是回滚，否则十分危险。

5.5.2 死锁

死锁是指，当两个或两个以上的事务，因争夺锁资源而造成的一种互相等待的现象。若无外部介入，事务永远无法推进下去。

解决死锁最简单但不可行的方法就是禁止等待，将任何的阻塞等待都转化为回滚，并且事务重新开始。但是可能导致并发性能的下降，甚至任何一个事务都不能进行。

解决死锁的另一个方法就是超时，当两个事务互相等待时，其中一个事务由于超时而进行回滚，而另一个事务就能继续进行。但是超时机制过于简单，它不能对超时的事务所占权重进行比较。如果事务操作更新了很多行，占用了很多undo log，如果回滚需要的时间会更多。

数据库普遍采用的死锁检测机制是wait-for graph（等待图），这是一种更为主动的死锁检测方式。InnoDB也采用这种方式。Wait-for graph需要数据库保存两种信息：

* 锁的信息链表
* 事务等待链表

由此可以构造出一张wait-for graph，如果图中存在回路，就代表存在死锁。

wait-for graph是一种较为主动的死锁检测机制，在每个事务请求锁发生等待时都会判断是否存在回路，若存在则有死锁，InnoDB通常会选择回滚undo量最小的事务。

Wait-for graph的死锁检测通常采用深度优先的算法实现，InnoDB 1.2之前采用递归方式，之后采用非递归的方式实现，进一步提高性能。

1. **事务**

**6.1**

1. **Redis**

**1、数据结构与对象**

**1.1 简单动态字符串**

1.1.1 SDS

Redis只在不需要对字符串进行修改的地方使用C语言传统的字符串，比如日志打印，而需要可以被修改的字符串值时，会使用Redis构建的简单动态字符串（Simple Dynamic String, SDS），包括包含字符串的键值对。

sds.h/sdshdr表示一个SDS，结构包含三个字段：

* len：记录buf数组中已使用字节数
* free：记录buf数组中未使用的字节数
* buf[ ]：字节数组

SDS遵循C字符串以空字符结尾的惯例，保存空字符的1字节不计算在len内，且是SDS函数自动添加。这样的好处是可以重用C字符串函数库的一部分函数。

1.1.2 SDS相比C字符串的优点

* 常数时间获取字符串长度

C字符串并不记录自身的长度信息，获取长度需要遍历整个字符串直到字符串结尾的空字符。而SDS在len属性中记录了SDS本身的长度，且设置和更新SDS长度的工作是由SDS的API在执行时自动完成的。

* 杜绝缓冲区溢出

C字符串由于不记录本身的长度，在进行字符串拼接时，如果没有提前分配足够的内存，就会发生溢出，覆盖后面的有用数据。SDS的空间分配策略可以杜绝这种可能性。SDS API在对SDS进行修改时，会先检查SDS的空间是否足够，自动扩展SDS的空间。

* 减少修改字符串带来的内存重分配次数

当进行字符串拼接时，需要提前通过内存重分配来扩展底层数组的大小，否则会产生溢出。当进行字符串截断时，需要通过内存重分配来释放字符串不需要的空间，否则会产生内存泄漏。

内存重分配涉及复杂的算法，并且可能需要执行系统调用。对于追求存取速度，数据频繁改动的Redis，内存重分配会严重降低性能。为了避免这种缺陷，SDS通过未使用空间（free）实现了空间预分配和惰性空间释放两种优化策略。

· 空间预分配：优化字符串的增长操作，如果SDS修改之后长度小于1 MB，那么分配和len同样大小的未使用空间；如果修改之后SDS长度大于等于1 MB，那么分配1 MB的未使用空间。通过空间预分配策略，减少连续执行字符串增长操作需要的内存重分配次数。

· 惰性空间释放：优化字符串的缩短操作，当SDS缩短字符串时，并不立即通过内存重分配来回收字节，而是用free属性将这些字节记录起来，等待将来使用，减少内存重分配，并优化将来可能的字符串增长。

* 二进制安全

C字符串中的字符必须符合某种编码，并且除了字符串末尾，字符串中不能包含空字符，因此C字符串只能保存文本数据。而所有的SDS API会以处理二进制的方式来处理SDS存放在buf字节数组里的数据，数据写入是什么样，读取时也是什么样。因此SDS可以保存任意格式的二进制数据。

* 兼容部分C字符串函数

虽然SDS API都是二进制安全的，但是依然遵循着字符串以空字符结尾的惯例，这是为了让保存文本数据的SDS可以重用一部分C库中定义的函数。

**1.2 链表**

当一个列表键包含数量较多的元素，或者列表中包含的元素都是比较长的字符串，Redis就会使用链表作为列表键的底层实现。

每个链表节点adlist.h/listNode包含pre前驱点指针，next后继点指针以及value节点值。

每个链表adlist.h/list包含head表头节点，tail表尾节点，len节点数量，以及节点值复制函数、释放函数以及对比函数。函数成员是用于实现多态链表所需的特定类型的函数，因此**链表可以用于保存各种不同类型的值**。

链表为双端无环链表，表头的pre和表尾的next都指向null，对链表的访问以null为终点。

**1.3 字典**

当一个哈希键包含的键值对比较多，或者键值对中的元素都是比较长的字符串时，Redis就会使用字典作为哈希键的底层实现。

Redis的字典使用哈希表作为底层实现，一个哈希表包含多个哈希表节点，每个哈希表节点保存一个键值对。

1.3.1 哈希表

dict.h/dictht哈希表包含：table哈希表数组，size哈希表大小，sizemask哈希表大小掩码=size-1，used哈希表已有节点数量

table指向一个数组，每个数组元素都是一个指向dict.h/dictEntry哈希表节点结构的指针，每个dictEntry保存一个键值对。

1.3.2 哈希表节点

dict.h/dictEntry哈希表节点包含：key键，v值，next指向下一个哈希表节点的指针，形成链表。v属性保存键值对中的值，可以是一个指针，也可以是uint64\_t整数，或者int64\_t整数。

1.3.3 字典

dict.h/dict字典包含：type类型特定函数，privdata私有数据，ht[2]哈希表，rehashidx（rehash索引）。

type和pridata属性**针对不同类型的键值对**，type是一个指向dictType结构的指针，dictType保存了一系列操作特定类型键值对的函数，privdata则保存了需要传给类型特定函数的可选参数。

ht[2]包含两个哈希表（ht[0]和ht[1]），ht[1]哈希表是在进行rehash时使用，rehashidx记录rehash的进度，当前没有正在进行rehash则值为-1。

1.3.4 rehash

rehash的时机：当负载因子大于1或者小于0.1时

* 为字典的第二个备用哈希表ht[1]分配空间。如果是扩展操作，ht[1]的大小为第一个大于等于ht[0].used \* 2的2^n；如果是收缩操作，ht[1]的大小为第一个大于等于ht[0].used的2^n。
* 将保存在ht[0]中的所有键值对rehash到ht[1]，ht[0]中的键值对都迁移到ht[1]之后，释放ht[0]，并将ht[1]设置为ht[0]，并创建一个新的空白哈希表（size=0）作为ht[1]。

1.3.5 渐进式rehash

扩展或收缩哈希表时需要将ht[0]的键值对移动到ht[1]中，但并不是一次性、集中式完成，而是分多次、渐进式移动的。

* 为ht[1]分配空间
* 将字典中的rehashidx值设为0，表示rehash开始
* 在rehash期间，每次对字典执行添加、删除、查找或更新操作时，都会顺带将ht[0]中的键值对rehash到ht[1]中，并将rehashidx增一
* 当ht[0]所有键值对都rehash到ht[1]后，将rehashidx置为-1

渐进式rehash将rehash所需的工作分摊到对字典的每个添加、删除、查找和更新操作上。字典的删除、查找和更新操作会在ht[0]和ht[1]两个哈希表上进行，而添加的键值对一律保存到ht[1]中。

**1.4 跳跃表**

当一个有序集合包含的元素数量较多，或者有序集合元素都是比较长的字符串，Redis就会使用跳跃表作为**有序集合**的底层实现。Redis除了有序集合之外，跳跃表只在集群节点中用作内部数据结构，除此之外没有其他用途（那特么面试还尽考这个？）

1.4.1 跳跃表节点

redis.h/zskiplistNode跳跃表节点包含：level[]层，backward后退指针，score分值，obj对象成员。

level[]每个层都包含两个属性：forward前进指针，span跨度。forward指向表尾方向的前进指针，span表示前进指针所指向节点与当前节点之间的距离，指向null的前进指针跨度都为0。每次创建一个新跳跃表节点的时候，都会根据幂次定律随机生成一个介于1~32之间的值作为level数组的大小。

每个节点都只有一个指向前一个节点的后退指针。

分值是一个double类型的浮点数，跳跃表中所有节点都按分值从小到大排列。obj是一个指针，指向一个**SDS字符串对象**。

同一个跳跃表中，每个节点保存的成员对象必须是唯一的，但分值可以相同。

1.4.2 跳跃表

redis.h/zskiplist跳跃表包含：header头结点，tail尾结点，length节点数量，level层数最大节点的层数。

**1.5 整数集合**

当一个集合只包含数值元素，并且元素数量不多时，Redis就会使用整数集合作为**集合键**的底层实现。

整数集合可以保存int16\_t、int32\_t或者int64\_t的整数值，**并且保证集合中不会出现重复元素**。

intset.h/intset整数集合包含：encoding编码方式，length元素数量，contents[]元素数组，数组中按照值大小从小到大有序排列，且不包含任何重复项。contents[]数组的类型取决于encoding的值。

1.5.1 升级

当往整数集合中添加元素时，新元素的类型更长，那么整数集合就需要先进行升级：

· 根据新元素的类型，扩展整数集合底层数组的大小，并为新元素分配空间

· 将现有元素都转换为与新元素相同的类型，并将其放在合适的位置，保证有序性不变

· 将新元素添加到数组中

升级策略可以是底层数组来适应不同类型的新元素，提升灵活性，同时既能让集合同时支持三种不同的类型，又能确保只有在需要的时候才会进行数组升级，节约内存。

整数集合不支持降级哦。

**1.6 压缩列表**

压缩列表是列表键和哈希键的底层实现之一。

当列表键只包含少量元素，并且每个元素为小整数或短字符串，就使用压缩列表作为列表键的底层实现。

当哈希键只包含少量键值对，每个键值对的键和值为小整数或短字符串，就使用压缩列表作为哈希键的底层实现。

1.6.1 压缩列表构成

压缩列表为了节约内存而开发的，由**连续内存块**构成的**顺序型**数据结构。一个压缩列表包含任意多个节点，每个节点可以保存一个字节数组或整数值。

ziplist包含：zlbytes压缩列表字节数，zltail压缩列表表尾节点的偏移量，zllen压缩列表节点数量，entryX列表节点，zlend压缩列表末端标记。

1.6.2 压缩列表节点构成

压缩列表节点包含：previous\_entry\_length前一个节点的字节数，encoding记录content属性保存数据的类型以及长度，content节点值。

根据previous\_entry\_length可以通过指针运算，由当前节点的起始地址来计算出前一个节点的起始地址，结合压缩列表的ztail属性，可以实现从表尾向表头遍历。

**1.7 对象**

1.7.1 总结

Redis数据中每个键值对都是由**对象**组成，其中数据库键总是一个字符串对象，而数据库键的值可以是字符串对象、列表对象、哈希对象、集合对象、有序集合对象中的一种。

**字符串对象**：SDS

**列表对象**：当元素较多/为长字符串，用链表，否则用**压缩列表**

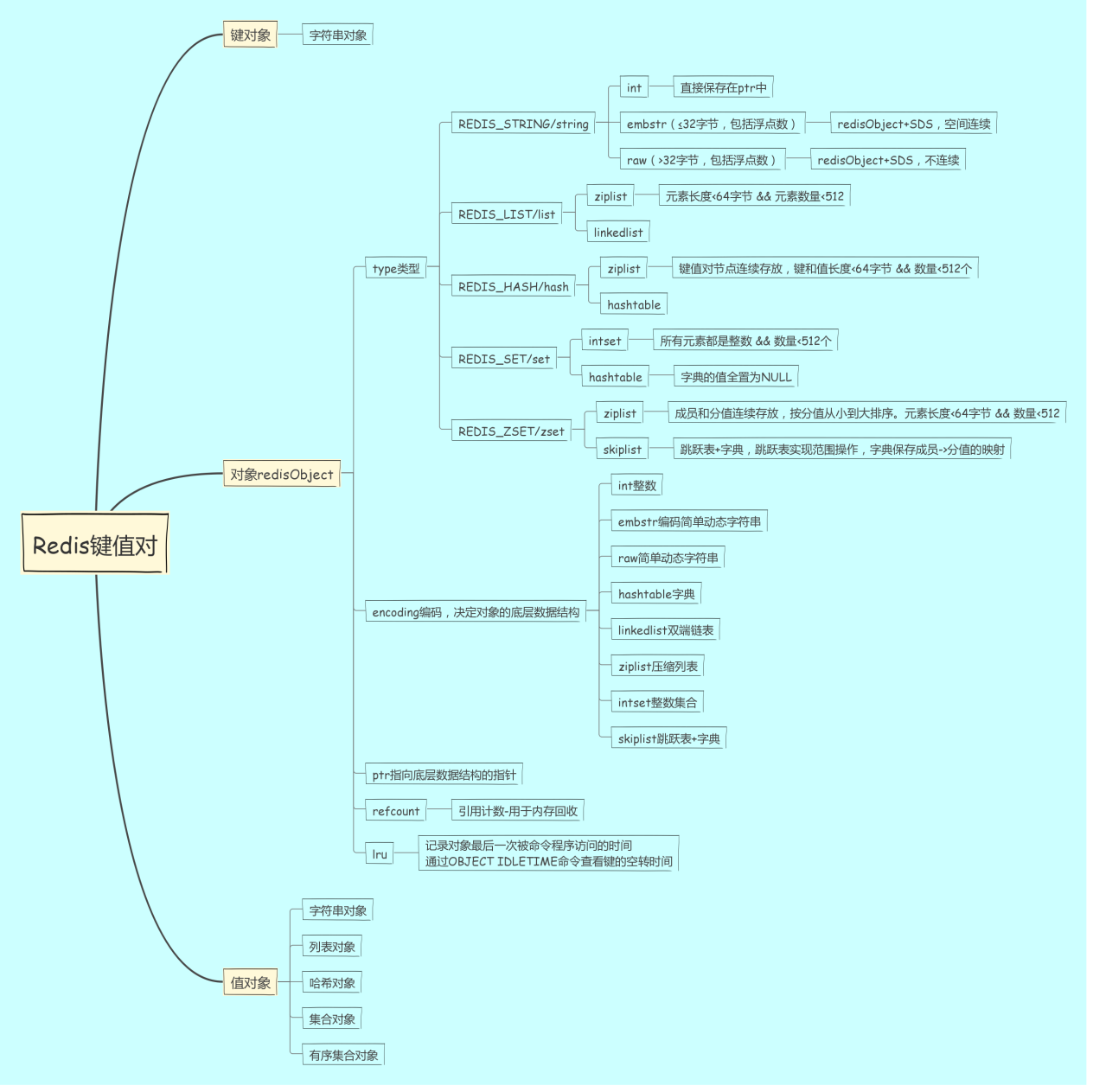
**哈希对象**：当元素较多/为长字符串，用字典，否则用**压缩列表**

**集合对象**：只包含数值&数量不多，用整数集合，否则用字典

**有序集合对象**：当元素较多/为长字符串，用跳表+字典，否则用**压缩列表**

列表、哈希、有序集合对象都是当元素较多或都为长字符串时，就采用其相对应的实现，否则，当元素较少，且都是小整数或者短字符串，就用**压缩列表**。

集合对象本身就可以使用字典来实现，跟Java中是一样的，只有在集合只包含数值且数量较少时，可以采用整数集合来实现。



1.7.2 类型检查和命令多态

Redis用于操作键的命令可以分为两种：可以用于任何类型键的，和只能用于特定类型键的命令。Redis会先检查命令的输入键的类型（type）是否正确，然后再根据值对象的编码（encoding）来选择用于值对象的正确命令。也就是先类型检查，再进行命令多态。

实际上，用于所有类型（type）键的命令也可以称为多态命令，这是一种基于类型的多态。而专用于特定类型（type）键的命令则是一种基于编码（encoding）的多态，可以处理同一类型中的不同编码。

1.7.3 内存回收和对象共享

Redis通过redisObject结构的refcount属性记录对象的引用计数，创建新对象时，refcount初始化为1，当对象的引用计数减为0时，就会对该对象进行内存回收。

refcount也可以用于对象共享，当多个字符串键共享同一个值对象时，需要将指针指向该共享对象，同时将refcount引用计数增一。不仅字符串键可以共享这些字符串，潜逃了字符串对象的对象，如列表对象、哈希对象等都可以共享这些字符串对象。但是，需要注意的是，Redis只共享包含整数值的字符串对象，而不共享包含字符串或者多个值的对象，因为需要先判断要创建的目标对象与共享对象是否完全相同，这需要线性时间甚至更高，而整数值的字符串对象只需要常数时间。