

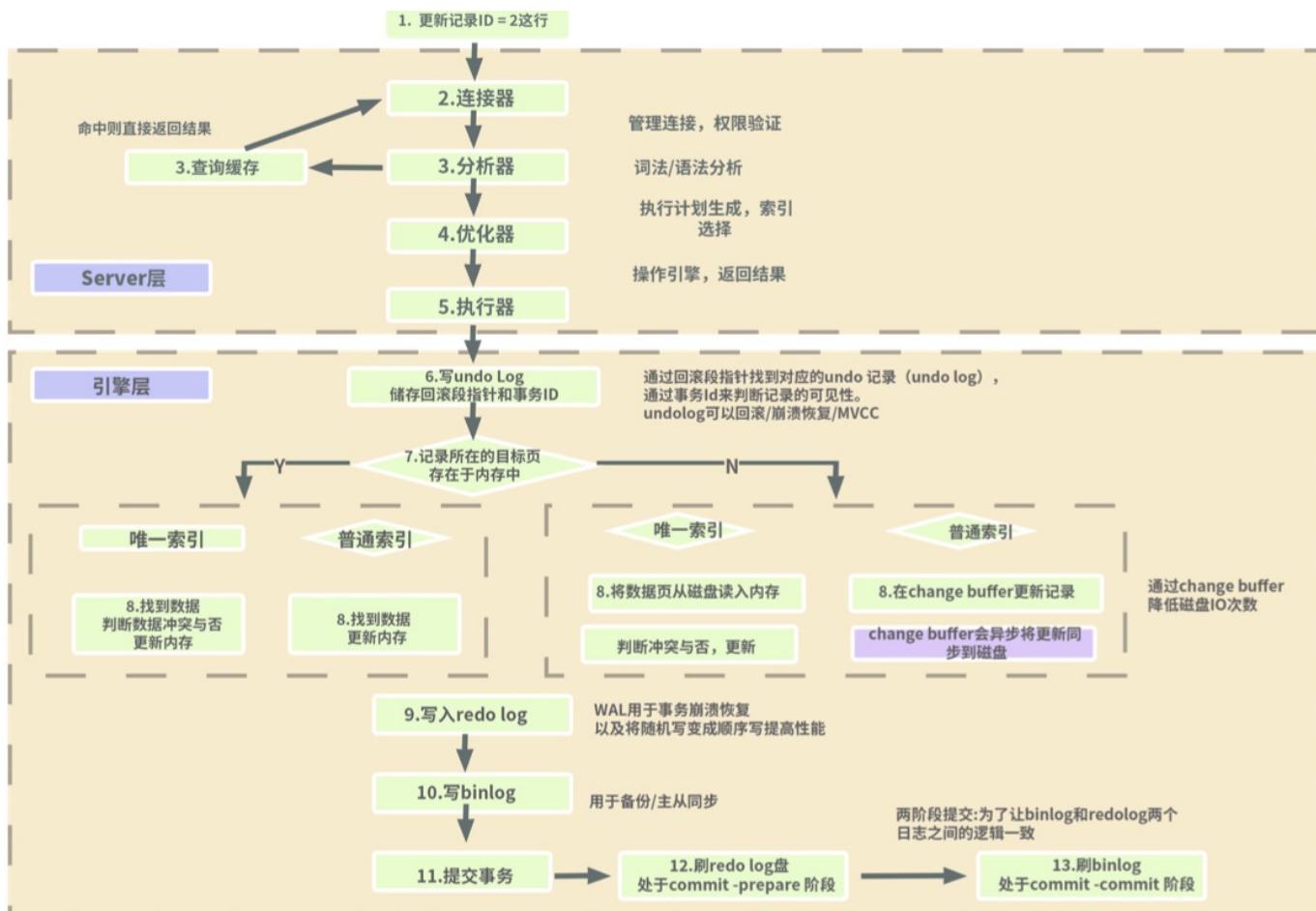
MySQL

query_cache_size 5.7,5.6 have, 8.0 don't have this.

sourt_buffer_size default 1M

max_allowed_packet default: 32M

innodb_buffer_pool_size = 128M



sudo chmod -R 777 根路径

r=4, w=2, x=1

mysiyam 有一个记录总数的表，对总数返回比innodb快
sql 执行顺序

from -> on -> join -> where -> group by -> having -> select -> order by -> limit
mysql 单表数据不超过2000万， 因为3层索引结构是性能最佳
一个数据块大约16K， 一条数据1K 8 (主键bitint)+6 = 14 一层： 16*1024/14 = 1170 两层：
1170*16 三层1170 * 1170 * 16
innoDB B+树实现聚集索引

如何恰当选择引擎： 不需要强事物： MyISAM , 需要强事物： Innodb, 大量重复就用压缩： toku
库表命名： 模块前缀命名
数据类型： 明确， 尽量小
text, blob,clob： 造成空间浪费， 或者增多数据库操作次数。
数据精度问题： int(3), int(8) 这两者只有显示的问题。

唯一约束与索引的关系： 唯一约束自动就会建索引。
尽量少用存储过程， 函数
分布式尽量不使用自增主键
在线修改表结构会锁表 (DDL操作)
导入数据可以先删除索引， 然后导入再重建索引

Mysql 事物 ACID和锁

1. 原子性 Atomicity
2. 一致性 Consistency
3. 隔离性 Isolation
4. 持久性 Durability

mysql 锁

1. 表级锁： 意向锁 , 共享锁 S, 排他锁 X
 - a. 共享意向锁 IS
 - b. 排他意向锁 IX
 - c. insert 意向锁
2. 其他：
 - a. 自增锁 由于自增都是内存里操作所以要加锁
 - b. LOCK TABLES/DDL select * from table in share model //添加共享锁； select * from table for update // 添加排他锁
3. 行锁：

- a. 记录锁 (Record) select * from table where xxx for update // 添加行锁
- b. 间隙锁 (Gap)
- c. 临建锁 (Next-Key)
- d. 谓词锁 (Predicat)

事物

读未提交：不能保证数据一致性， 读脏数据

读已提交

可重复读：多版本并发控制， 以及间隔锁， 保证没幻读

可串行化：最严格的隔离级别， 资源消耗最大

undo log: 撤销日志

1. 用于事物回滚， 一致性读， 崩溃恢复
2. 保证事物的原子性
3. 记录事物回滚是所需的撤销操作
4. undo tablespaces 8.0 ,

redo log: 重做日志

1. 确保事物的持久性， 防止事物提交后数据未刷新到磁盘就掉电或崩溃
2. 事物执行过程中写入redo log ,记录事物对数据页做里哪些修改
3. 提升性能

mvcc 多版本并发控制

1. 隐藏列 事物ID， 指示最后插入或更新该行的事务。 小于我自己的事务ID的都可见
2. 回滚指针， 指向回滚段中写入的undo log。
3. row id: 聚簇索引

SQL 优化

1. 小于5的倒序， 大于5的顺序： select * from student order by if(id < 5, -id, id);
2. 没特别因素就选择InnoDB 引擎。 归档库就选择TokuDB引擎
3. mysql 有隐式转换， 与javascript类似， 隐式转换不走索引， 影响性能。
4. slowlog
5. 索引类型： hash, B+ 树。
6. B+树的所有数据都在叶子节点，并且数据块之间有双向指针， 不需要回溯父节点如果找到了其中一块
7. 自增减少页分裂问题
8. 主键索引是直接跟数据关联的
9. 非主键的索引是二级索引， 跟数据没多大关系。 非主键索引最终还是通过回表走主键索引来找数据
10. 组合索引， 索引冗余。

修改表结构

1. 索引重建
2. 锁表
3. 抢占资源
4. 主从延时

写入优化

1. 大批量写入的优化
 - a. PreparedStatement 减少SQL解析
 - b. Multiple values / add batch减少交互
 - c. load data 直接导入
 - d. 先去掉索引和约束，然后导入后重建索引和约束
2. 数据更新
 - a. 注意GAP lock，最好用ID精确到具体的行。
 - b. 模糊查询，like 默认是前缀匹配，否则不走索引（%a% 这种不走索引）
 - c. 连接查询：避免笛卡尔积，
 - d. 索引失效：NULL, not, not in , or, 函数, like '%%'. 减少使用or, 如果使用or可以优化成使用union

实现主键ID

1. 自增ID，分库分表不建议使用，容易冲突
2. sequence : oracle & db2, 是单独的表
3. mysql实现sequence，需要单独设置一个表，sequence name, init name, 步长。坏处不连续。容易被估算出数据规模，可能泄漏商业机密
4. uuid
5. 时间戳加随机数
6. 雪花算法：snowflake，机器+时间戳+自增

高效分页

1. 分页插件
2. 重写select count(*) from 主表。通常主表能决定数据的页数
3. 模糊分页

乐观锁悲观锁

1. select * from xx for update; update xxx; commit; – 悲观锁
2. select * from xxx update xxx where value = old value – 乐观锁

从单机到主从复制

master insert binlog

slave pull binlog and transfer to relay log

use sql thread execute relay log

MGR: innodb group replication

binlog 格式:

row : 行模式, 每一个操作很精确, 清楚明白

statement: sql 模式

mixed: 混合以上两者

传统主从复制 – 异步复制

1. master insert binlog
2. slave pull binlog and transfer to relay log
3. use sql thread execute relay log
4. generate binlog in slave and commit transaction.

以上跟primary是无交互

半同步复制:

1. primary generate binlog and publish to slave
2. slave receive binlog and transfer to relay log and one of slave ack to primary
3. primary commit transaction

保证了可靠性。但是超时后会退化到异步复制, 如果slave操作慢了或者宕机, 会退化到异步复制

MySQL group replication

没有主从概念。

增加验证过程避免冲突

然后生成binlog 并提交

读写分离

v2: sharding sphere plugin

中间件: 读写分离规则放在中间件上。 application -> APIs -> DB

mysql 高可用

读写分离提升读的处理能力

故障转移， 提供failover能力 灾难恢复： 热备 (hot-hot) , 冷备 (hot-warm)

两地三中心 (两个不同机房, 多个城市 – 三个不同数据中心)

高可用方案：

1. 手动切换主从
 - a. 数据可能不一致
 - b. 需要人工干预
 - c. 代码和配置的侵入性
2. 用LVS+Keepalived 实现多个节点的探活 + 请求路由
3. MHA 至少需要3台机
4. MGR: 基于组复制, 保证数据一致性。
 - a. 外部需要配置SLA
 - b. 脑裂: 只要大多数节点都宕机, 然后不对外提供服务, 这样防止脑裂
5. MySQL cluster
6. Orchestrator : 中间件, 有UI界面。拖拽改变主从关系。

```
CREATE TABLE test (
id bigint NOT NULL AUTO_INCREMENT,
value varchar(255) DEFAULT NULL,
PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
```

数据库拆分

1. 容量, 性能, 可用性和运维成本
2. 主从复制不能解决性能和容量问题
3. DDL (加索引和列) 操作会锁表
4. 主从结构解决了高可用, 读扩展
5. 分库分表, 分布式数据库解决性能问题
6. 扩展立方体: X轴: 集群; Y轴: 业务拆分; Z轴: 数据分片
7. 数据复制 -> 主从结构, 备份和高 -> X轴拆分
8. 分布式服务化, 微服务 -> 垂直分库分表 -> 业务分类数据 -> Y轴拆分
9. 分布式结构, 任意扩容 -> 水平分库分表 -> 任意数据 -> Z轴拆分

数据库垂直拆分：不同业务数据放不用的库

1. 连接数不够用，数据库连接数是有限的，也是宝贵的资源；数据库升级也有强依赖。
2. 可拆库和拆表
3. 拆库对sql是有影响的，对大的join sql 就采用分成两段sql处理。
4. 拆表：
 - a. 梳理清楚拆分范围和影响范围
 - b. 检查评估和重新影响到的服务
 - c. 准备新的数据库集群复制数据
 - d. 修改系统配置并发布新版本上线
5. 优点：
 - a. 单库变小，便于管理和维护
 - b. 对性能和容量有提升作用
 - c. 改造后，系统和数据复杂度降低
 - d. 可用作为微服务改造的基础
6. 缺点：
 - a. 库变多，管理变复杂
 - b. 对业务系统有较强的侵入性
 - c. 改造过程复杂，容易出故障
 - d. 拆分到一定程度就无法继续拆分

数据库水平拆分：水平拆分就是把某个表的一部分数据拿出去。

分库分表有什么优缺点：

- 1、解决容量问题
- 2、比垂直拆分对系统影响小
- 3、部分提升性能和稳定性

- 1、集群规模大，管理复杂
- 2、复杂 SQL 支持问题（业务侵入性、性能）
- 3、数据迁移问题
- 4、一致性问题

1.

数据迁移：

全量 + 增量

binlog + 全量 + 增量

中间件

分布式事务：

分布式条件下，多个节点操作的整体事务一致性

强一致性：XA分布式事务：两阶段的分布式事务

mysql innodb 引擎的XA事务

Mysql

柔性事务 TCC

1. 准备操作Try
2. 确认操作confirm
3. 取消操作cancel

注意事项

1. 允许空回滚
2. 防悬挂控制
3. 幂等设计：多次执行结果一致

SAGA模式：

1. 没有try阶段，直接提交事务

RPC 原理

RPC分布式服务化

1. 多个相同服务如何管理 --> 注册中心
2. 服务的注册发现机制
3. 负载均衡，路由等集群功能
4. 熔断，限流等治理功能
5. 重试等策略

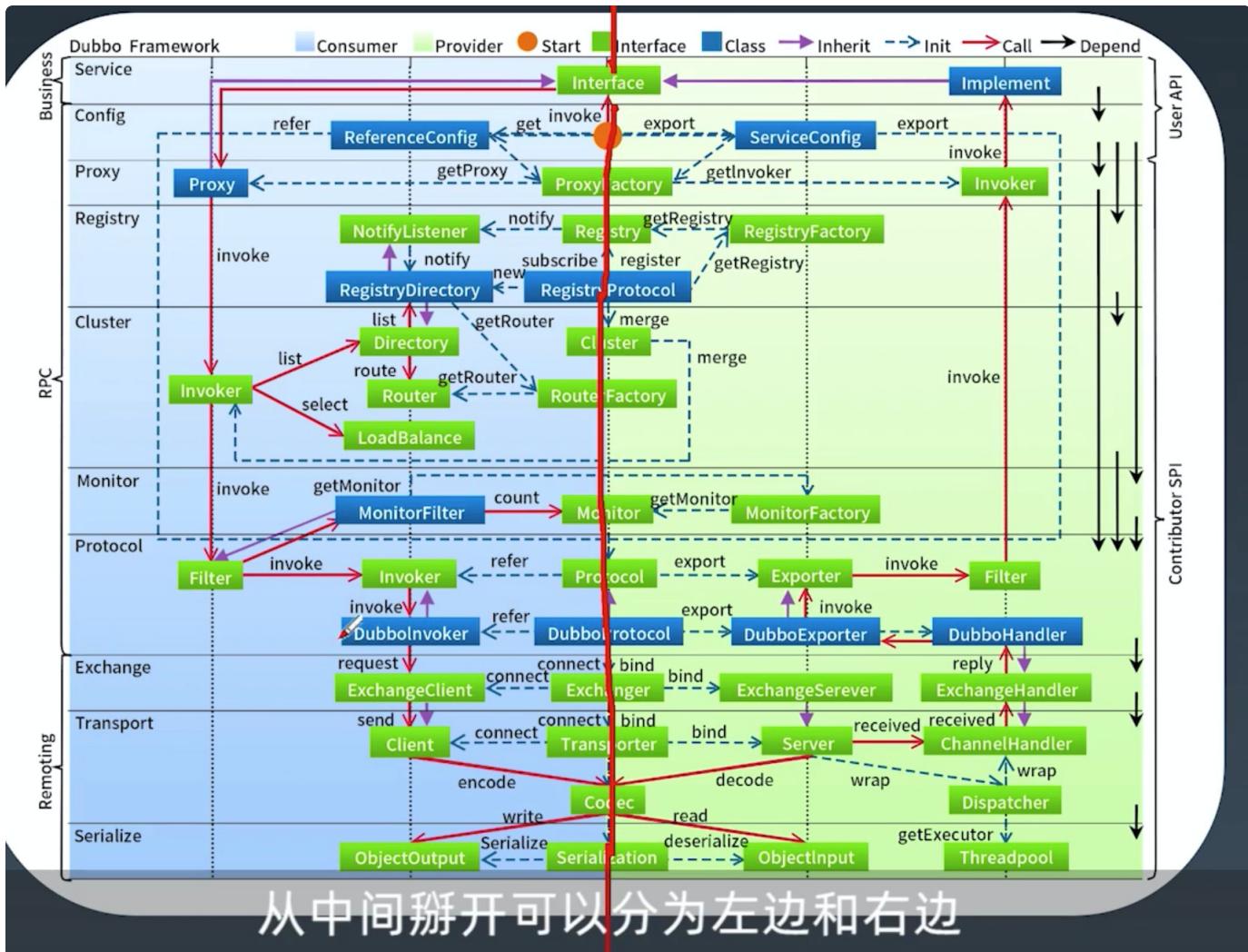
Dubbo 框架

基础功能：RPC调用

多协议

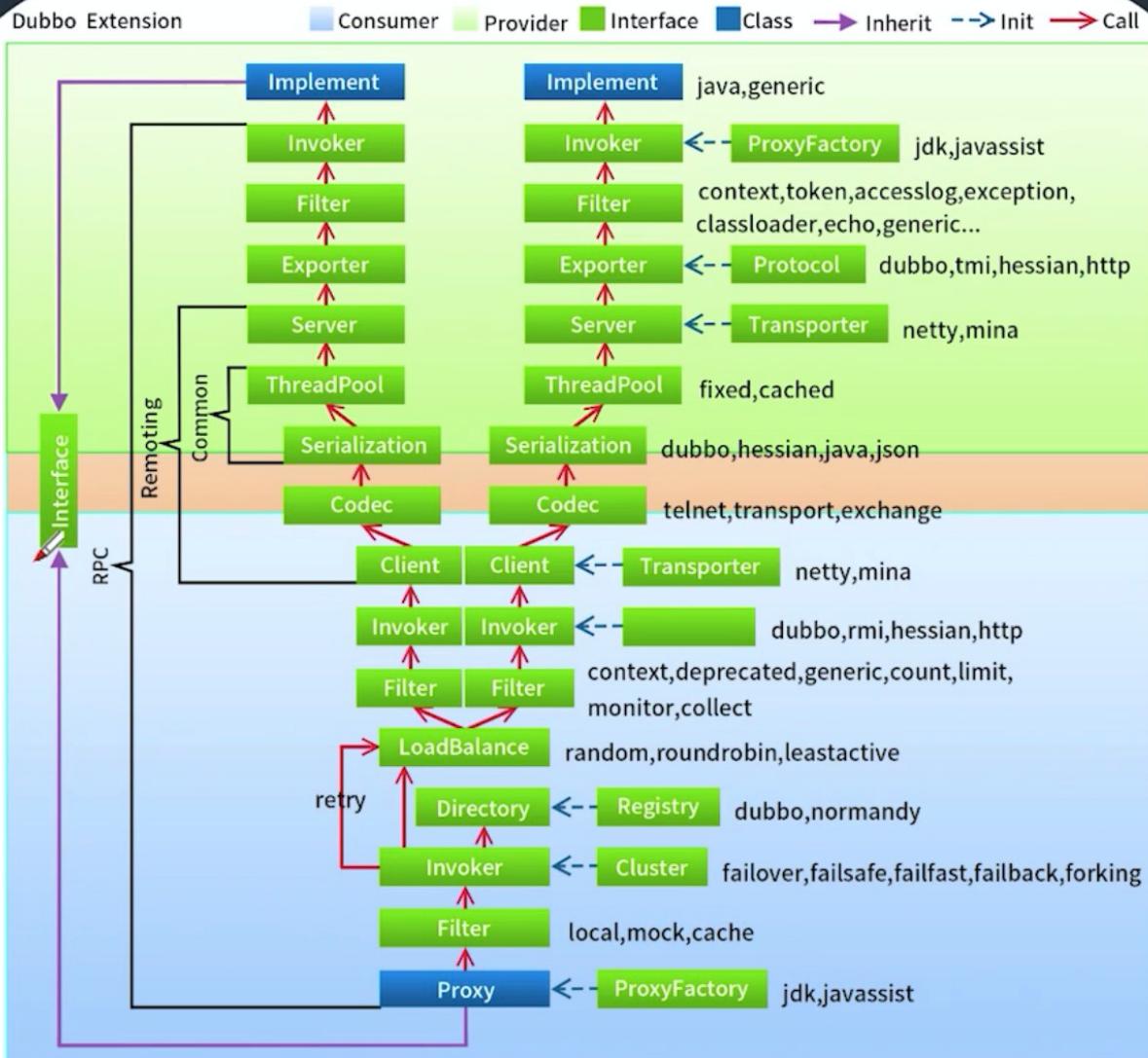
集群，负载均衡

治理，路由



从中间掰开可以分为左边和右边

Dubbo



这张图跟那张图画的方向是反的

解耦：callback, event bus, spi meta-inf/接口权限限定名 实现（jdk自带的service loader）