从执行 SQL 的角度来看，分库分表可以看作是一种路由机制，把 SQL 语句路由到我们期望的数据库或数据表中并获取数据，分片算法可以理解成一种路由规则。

咱们先捋一下它们之间的关系，分片策略只是抽象出的概念，它是由分片算法和分片健组合而成，分片算法做具体的数据分片逻辑。

分库、分表的分片策略配置是相对独立的，可以各自使用不同的策略与算法，每种策略中可以是多个分片算法的组合，每个分片算法可以对多个分片健做逻辑判断。

注意：sharding-jdbc 并没有直接提供分片算法的实现，需要开发者根据业务自行实现。

广播表

广播表：存在于所有的分片数据源中的表，表结构和表中的数据在每个数据库中均完全一致。一般是为字典表或者配置表 ，某个表一旦被配置为广播表，只要修改某个数据库的广播表，所有数据源中广播表的数据都会跟着同步。

绑定表

绑定表：那些分片规则一致的主表和子表。比如： 订单表和 订单服务项目表，都是按 字段分片，因此两张表互为绑定表关系。

绑定表关系

注意：在关联查询时 它作为整个联合查询的主表。所有相关的路由计算都只使用主表的策略， 表的分片相关的计算也会使用 的条件，所以要保证绑定表之间的分片键要完全相同。

计算机生成了可选文字:
ShardingDataSource 
DataSource 
ShardIngConnection 
Connection 
ShardingResultSet 
ResultSet 

计算机生成了可选文字:
Result-I 
SQL*M 
查 询 优 化 
SQLEü 
SQLäS 
SQL 执 行 
结 果 归 并 
Result-2 
Result-3 

计算机生成了可选文字:
接 着 语 法 解 折 会 将 拆 分 后 的 SQL 转 换 为 抽 象 语 法 树 ， 通 过 对 抽 象 语 法 树 遍 历 ， 提 炼 出 分 片 
所 需 的 上 下 文 ， 上 下 文 包 含 查 询 字 段 信 息 0 、 表 信 息 0 、 查 洵 条 件 0 、 排 序 信 息 
（ ） 、 分 组 信 息 （ ） 以 及 分 页 信 息 （ ） 等 ， 并 标 记 出 SQL 中 有 可 能 需 要 改 写 的 位 置 。 

SQL 路由

SQL 路由通过解析分片上下文，匹配到用户配置的分片策略，并生成路由路径。简单点理解就是可以根据我们配置的分片策略计算出 SQL该在哪个库的哪个表中执行，而SQL路由又根据有无分片健区分出 和 。

SQL 改写

将基于逻辑表开发的SQL改写成可以在真实数据库中可以正确执行的语句。比如查询 订单表，我们实际开发中 SQL是按逻辑表 写的。

但分库分表以后真实数据库中 表就不存在了，而是被拆分成多个子表 分散在不同的数据库内，还按原SQL执行显然是行不通的，这时需要将分表配置中的逻辑表名称改写为路由之后所获取的真实表名称。

SQL执⾏

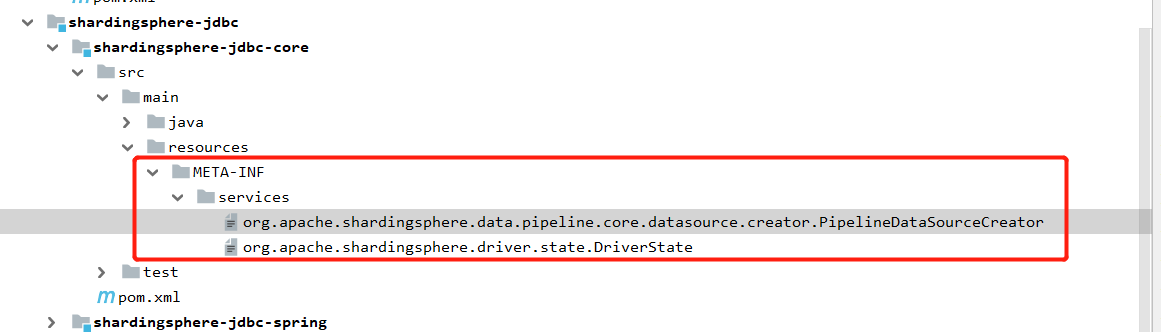
将路由和改写后的真实 SQL 安全且高效发送到底层数据源执行。但这个过程并不是简单的将 SQL 通过JDBC 直接发送至数据源执行，而是平衡数据源连接创建以及内存占用所产生的消耗，它会自动化的平衡资源控制与执行效率。

结果归并

将从各个数据节点获取的多数据结果集，合并成一个大的结果集并正确的返回至请求客户端，称为结果归并。而我们SQL中的排序、分组、分页和聚合等语法，均是在归并后的结果集上进行操作的。

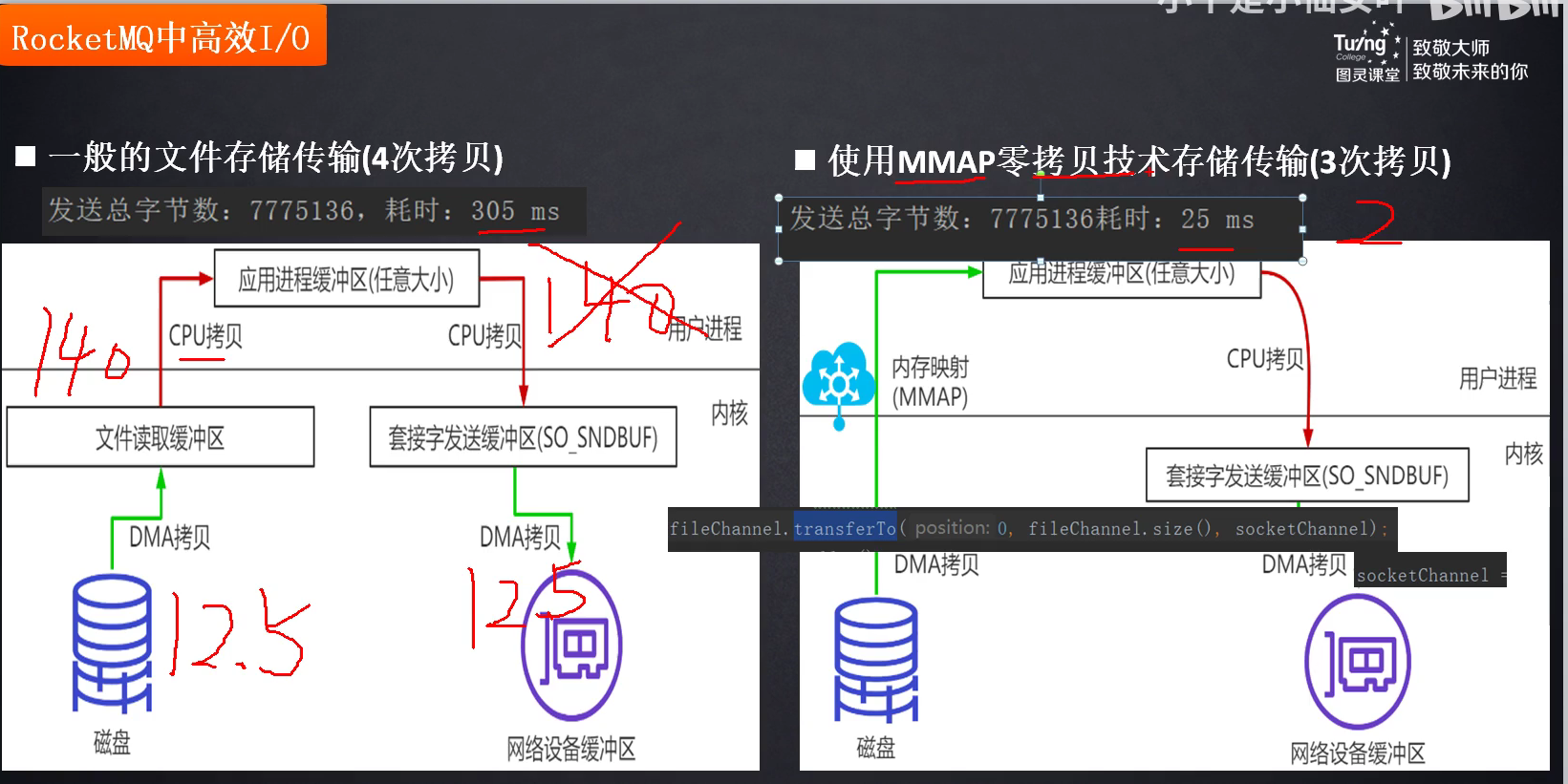
代理其实就是伪装成一个MYSQL

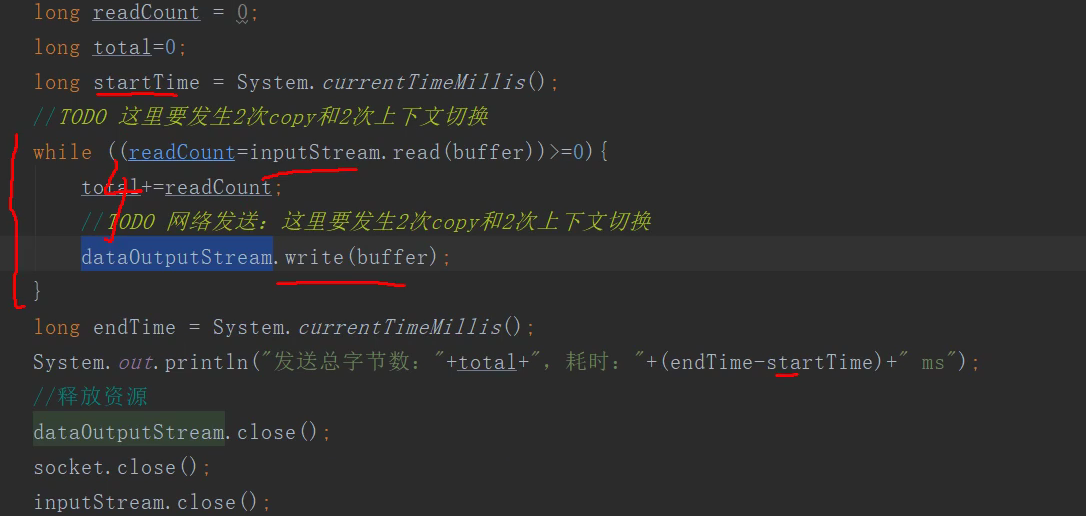
注册 == 加载

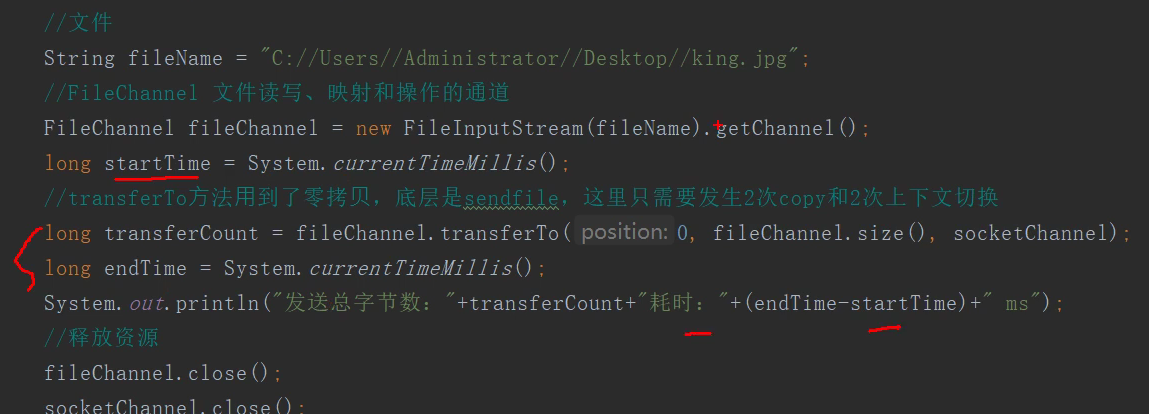
 SPI机制,java.util.serviceLoader

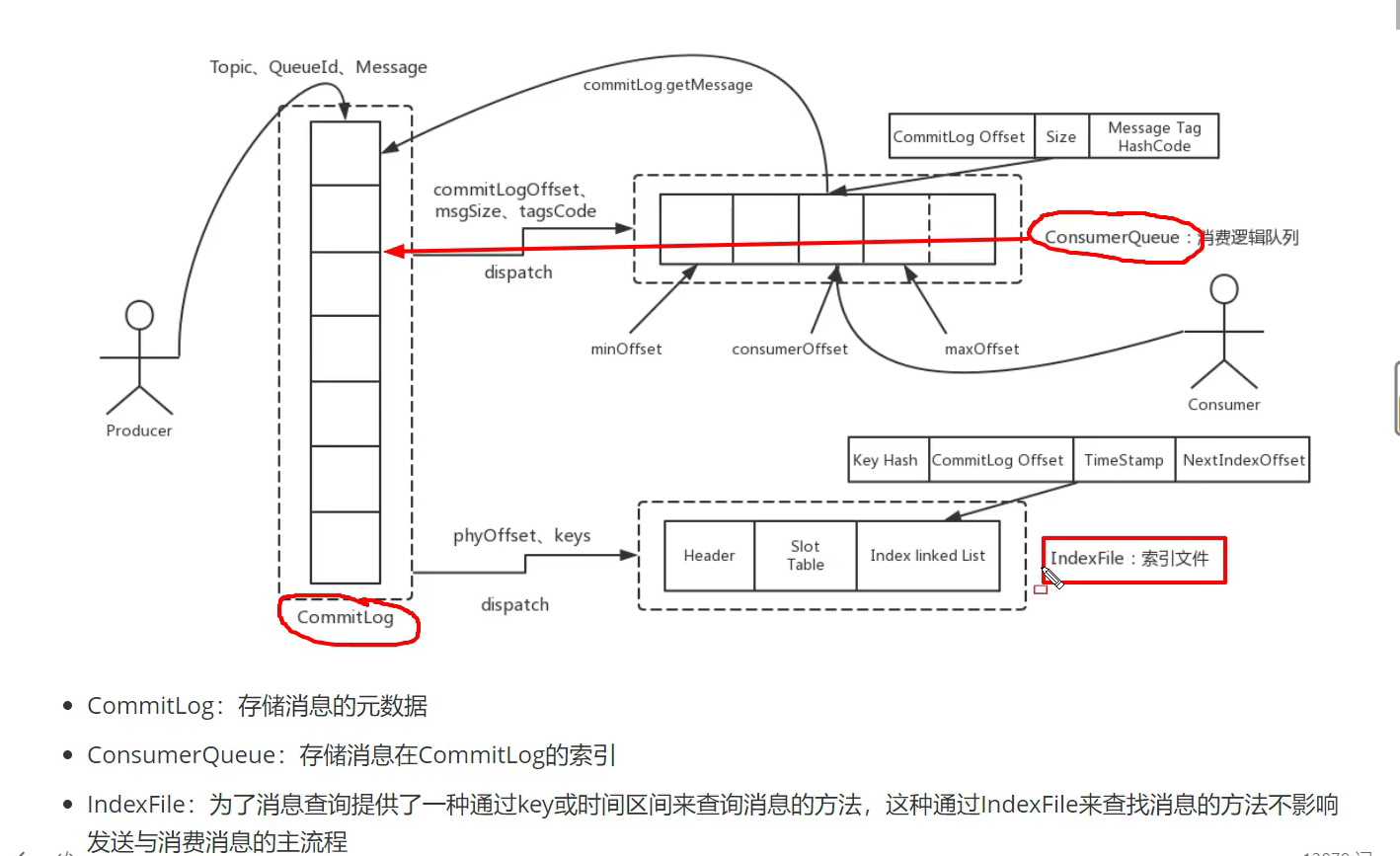
RocketMQ

很多东西表面看起来很简洁, 其实内部及其的复杂, 细节及其的多, 只是内部的东西对我们来说是透明的而已;





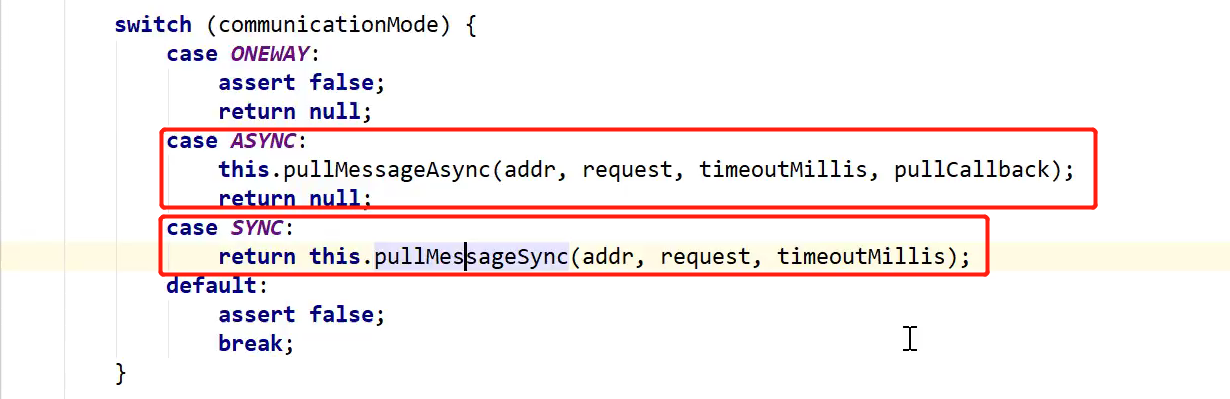




长轮训: 轮训肯定是客户端定时pulll, 长代表的是每次请求等待的时间都比较长;

发送: 发送者到broker

投递: broker到消费者



服务降级(指提供部分服务): 默认实现(返回null, 打日志,插入数据库等等)

|

服务熔断(电路中的保险丝, 完全不提供服务): 暂时不调用这台机器或者是这个服务;

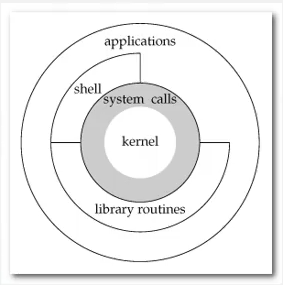
强一致性(如: 账务系统), 追求高效率(如:订单系统)

客户端, 服务端, 内存, 磁盘(本地文件), 日志

系统调用(syscall): 用户程序调用操作系统内核中的特殊函数

Linux: 操作系统内核

Linux操作系统[centos ,redhat]





**Spring**