TDH_SOCKET 对比分析

	TDH_SOCKET	HandlerSocket	SQL
IO 策略	Dynamic IOStrategy	Same-thread IOStrategy	one-thread-per-connection
优点	worker 线程只处理与 DB 相关的逻辑 最大化 DB 的操作吞吐量	上下文切换真心很少	资源分离 不太会相互干扰
缺点	上下文切换一般,测试时最高在 10w,但可以接受	IO 逻辑对 DB 逻辑影响较大 一旦 io 处理的慢了,会导致 QPS 下降很多(比如说较多的连接数并发请求导致 io 处理变慢也会是整体 QPS 下降)	线程资源浪费严重,太多的线程会导致高并发是上下文切换多,从而影响性能,测试时最高 60w
读策略	1. 无需 SQL 解析 2. 可在 THD 上 cache open 过的 table(可配置关闭) 3. 一次轮询只 lock 一次 table 4. 请求按 table 的 hash 值平均分配到可活动的线程上 5. 如果有 table 被过多访问,那么此 table 的请求可被分配到多个线程上 6. 在带 io 请求少时,可以用较少的线程 work.来减少锁征用导致的上下文切换 7. 有一定量带 io 请求时,能调整 worker 线程数来充分利用 io 资源. 8. 当带 io 请求巨多时,将区分带 io 的请求和不带 io 的请求.分别用两个线程池来执行,防止带 io 的请求堵住不带 io 的请求. 9. 还可以设置流控带 io 的请求数来防止 client 的堵塞情况,能使 DB 继续输出较	1. 无需 SQL 解析 2. 可在 THD 上 cache open 过的 table(不可配置关闭) 3. 一次轮询只 lock 一次 table 4. worker 线程数固定,且无法做请求数的平衡,只能做连接数的平衡	1. 可 global 的做 table 的 cache 2. 每次请求都 lock 一次 table

	高的 QPS 10. 如果返回大批量数据可以以 stream 的方式返回,避免过多占用内存		
优点	多种策略配合能一直输出较高的 QPS	在全缓存的请求下,输出的 QPS 很高	由于是一个连接一个请求,不会出现由带 io 的请求堵塞不带 io 的请求的情况出现
缺点	流控会丢失一部分请求, 预测请求带不带 io 不是非常准确 (如果使用 row cache 就能较精确的预测了)	一旦有带 io 的请求出现,就很容易堵塞住不带 io 的请求出现,所以缓存命中率下降后 QPS 下降的非常厉害 返回大批量数据会占用过多内存	有极限一般 QPS 最高输出到 6-7w
写策略	1. 一次轮询的请求一次 commit 2. 请求按 table 的 hash 值平均分配到写线程上,即同一个表肯定在一个写线程的被执行,不会有死锁的问题,也不会有同一条 row 的锁问题. 3. 支持 Batch 方式的小事务,Batch 内的请求保证同时成功,同时失败 4. 可以在 Client 端通过指定 hash 的方式,来控制请求被指定的线程执行,来避免一些锁或死锁	1. 一次轮询的请求一次 commit 2. 可配置多个写线程,但是有一个 user_lock 保证只有一个写线程在工作,来防止死锁	1. 一次请求一次 commit
优点	同一个表必然在同一个线程上执行,所以行锁的征用被降到最低,加上 group commit 就能提供很搞的 TPS 可以多个写线程并发写,不会有死锁问题 支持 batch 小事务	Group commit 可以减少 fsync 次数,提高写性能	支持大事务

缺点	写线程数有限,在有 io 堵塞请求下,性能会有下降	同一时刻只有一个写线程能被执行,很容易达到瓶颈,一旦有多表插入并发出现,很容易影响整体更新性能 生成 binlog 没有执行时间(bug) 多表 insert 会有自增 id 获取失败的问题(bug)	性能受限于 IOPS
可维护性	1. DDL 操作不会被 hang 住(因为可以主动 close 被 cache 的 table) 2. 支持 KILL <thread id>命令直接终止正在执行的操作 3. 大部分参数都为在线可配置 4. status 数据监控完善	1. DDL 会被 hang 住,因为 handlersocket 只 open table 不管主动 close table,只有在流量较小 time out 时才会 close table	最高
易用性	1. client 端不需要 openIndex,直接请求即可(如: client.query().use("test").select("id", "a", "b", "c").from("t") .where().fields("a").in(["1", "1"], ["2"], ["1"]).get()) 2. server 端只需要一个 port 就能做读写分离 3. 可支持一些 MySQL 的内置函数 1. 现在支持 Update/Insert 的时候可以插入 now() 4. 基于辅助索引的联合主键可以支持简单的 order by 1. 只能支持=的 order by 5. Java 客户端支持 JDBC	1. 读写分离需要在 server 端配置两个 port 是实现 2. client 请求前需要先 openIndex	最高