由此开始答题

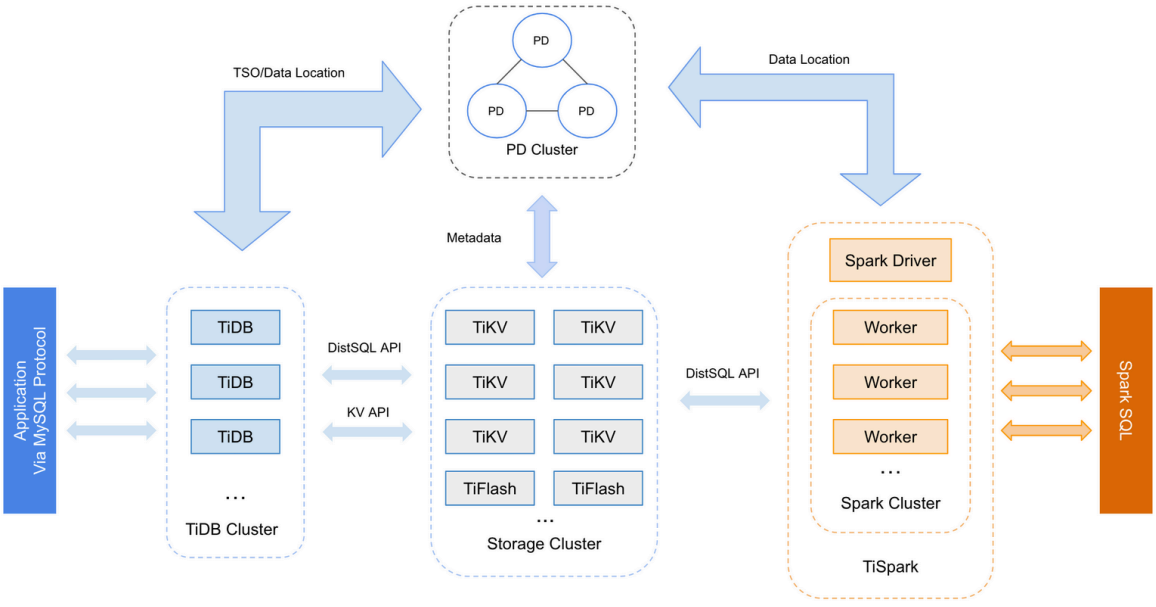
学号： 71205902002　　姓名：苏胤榕

一、答案：

TiDB整体架构

- 纯分布式架构，拥有良好的扩展性，支持弹性的扩缩容
- 支持SQL,对外暴露MySQL的网络协议，并兼容大多数MySQL的语法，在大多数场景下可以直接替换MySQL
- 默认支持高可用，在少数副本失效的情况下，数据库本身能够自动进行数据修复和故障转移，对业务透明
- 支持ACID事务，对于一些有强一致需求的场景友好，例如：银行转账
- 具有丰富的工具链生态，覆盖数据迁移，同步、备份等多种场景

在内核设计上，TIDB分布式数据库将整体架构拆分成了多个模块，各模块之间互相通信，组成完整的TiDB系统。对应的架构图如下：



- TiDB Server: SQL层，对外暴露MySQL协议的连接endpoint, 负责接收客户端的连接，执行SQL解析和优化，最终生成分布式执行计划。TiDB层本身是无装填的，实践中可以启动多个TiDB实例，通过负载均衡组件（如LVS, HAProxy或F5)对外提供统一的接入地址，客户端的连接可以均匀地分摊在多个TiDB实例上以达到负载均衡的效果。TiDB Server本身并不存储数据，只是解析SQL，将实际的数据读取请求转发给底层的存储节点TiKV(或TiFlash).

- PD Server: 整个TiDB集群的元信息管理模块，负责存储每个TiKV实时的数据分布情况和集群的整体拓扑结构，提供TiDB Dashboard管控界面，并为分布式事务分配事务ID。 PD不经存储元信息，同时还会根据TiKV节点实时上报的数据分布状态，下发数据调度命令给具体的TiKV节点，可以说是整个集群的“大脑”。此外，PD本身也是由至少三个节点构成，拥有高可用的能力，建议部署奇数个PD节点。
- 存储节点
    - TiKV Server: 负责存储数据，从外部看TiKV是一个分布式的提供事务的Key-Value存储引擎。存储数据的基本单位是Region, 每个TiKV节点会负责多个Region, TiKV的API在KV键值层面提供对分布式事务的原生支持，默认提供了SI(Snapshot Isolation)的隔离级别，这也是TiDB在SQL层面支持分布式事务的核心，TiDB的SQL层做完SQL解析后，会将SQL的执行计划转换为对TiKV API的实际调用。所以都存储在TiKV中。另外，TiKV中的数据都会自动维护多个副本（默认为三个副本）天然支持高可用和自动故障转移。
    - TiFlash: TiFlash是一类特殊的存储节点。和普通TiKV节点不一样的是，在TiFlash内部，数据时以列式的形式进行存储，主要功能是为分析型的场景加速。

TiDB 数据模型

引言

数据库、操作系统和编译器并称为三大系统，可以说是整个计算机软件的基石。其中数据库更靠近应用层，是很多业务的支撑。这一领域经过了几十年的发展，不断的有新的进展。

很多人用过数据库，但是很少有人实现过一个数据库，特别是实现一个分布式数据库。了解数据库的实现原理和细节，一方面可以提高个人技术，对构建其他系统有帮助，另一方面也有利于用好数据库。

研究一门技术最好的方法是研究其中一个开源项目，数据库也不例外。单机数据库领域有很多很好的开源项目，其中 MySQL 和 PostgreSQL 是其中知名度最高的两个，不少同学都看过这两个项目的代码。但是分布式数据库方面，好的开源项目并不多。 TiDB 目前获得了广泛的关注，特别是一些技术爱好者，希望能够参与这个项目。由于分布式数据库自身的复杂性，很多人并不能很好的理解整个项目，所以我希望能写一些文章，自顶向下，由浅入深，讲述 TiDB 的一些技术原理，包括用户可见的技术以及大量隐藏在 SQL 界面后用户不可见的技术点。

保存数据

数据库最根本的功能是能把数据存下来，所以我们从这里开始。

保存数据的方法很多，最简单的方法是直接在内存中建一个数据结构，保存用户发来的数据。比如用一个数组，每当收到一条数据就向数组中追加一条记录。这个方案十分简单，能满足最基本，并且性能肯定会很好，但是除此之外却是漏洞百出，其中最大的问题是数据完全在内存中，一旦停机或者是服务重启，数据就会永久丢失。

为了解决数据丢失问题，我们可以把数据放在非易失存储介质（比如硬盘）中。改进的方案是在磁盘上创建一个文件，收到一条数据，就在文件中 Append 一行。OK，我们现在有了一个能持久化存储数据的方案。但是还不够好，假设这块磁盘出现了坏道呢？我们可以做 RAID （Redundant Array of Independent Disks），提供单机冗余存储。如果整台机器都挂了呢？比如出现了火灾，RAID 也保不住这些数据。我们还可以将存储改用网络存储，或者是通过硬件或者软件进行存储复制。到这里似乎我们已经解决了数据安全问题，可以松一口气了。But，做复制过程中是否能保证副本之间的一致性？也就是在保证数据不丢的前提下，还要保证数据不错。保证数据不丢不错只是一项最基本的要求，还有更多令人头疼的问题等待解决：

- 能否支持跨数据中心的容灾？
- 写入速度是否够快？
- 数据保存下来后，是否方便读取？
- 保存的数据如何修改？如何支持并发的修改？
- 如何原子地修改多条记录？

这些问题每一项都非常难，但是要做一个优秀的数据存储系统，必须要解决上述的每一个难题。 为了解决数据存储问题，我们开发了 TiKV 这个项目。接下来我向大家介绍一下 TiKV 的一些设计思想和基本概念。

Key-Value

作为保存数据的系统，首先要决定的是数据的存储模型，也就是数据以什么样的形式保存下来。TiKV 的选择是 Key-Value 模型，并且提供有序遍历方法。简单来讲，可以将 TiKV 看做一个巨大的 Map，其中 Key 和 Value 都是原始的 Byte 数组，在这个 Map 中，Key 按照 Byte 数组总的原始二进制比特位比较顺序排列。 大家这里需要对 TiKV 记住两点：

1. 这是一个巨大的 Map，也就是存储的是 Key-Value pair
2. 这个 Map 中的 Key-Value pair 按照 Key 的二进制顺序有序，也就是我们可以 Seek 到某一个 Key 的位置，然后不断的调用 Next 方法以递增的顺序获取比这个 Key 大的 Key-Value

讲了这么多，有人可能会问了，这里讲的存储模型和 SQL 中表是什么关系？在这里有一件重要的事情要说四遍：

这里的存储模型和 SQL 中的 Table 无关！ 这里的存储模型和 SQL 中的 Table 无关！ 这里的存储模型和 SQL 中的 Table 无关！ 这里的存储模型和 SQL 中的 Table 无关！

现在让我们忘记 SQL 中的任何概念，专注于讨论如何实现 TiKV 这样一个高性能高可靠性的巨大的（分布式的） Map。

RocksDB

任何持久化的存储引擎，数据终归要保存在磁盘上，TiKV 也不例外。但是 TiKV 没有选择直接向磁盘上写数据，而是把数据保存在 RocksDB 中，具体的数据落地由 RocksDB 负责。这个选择的原因是开发一个单机存储引擎工作量很大，特别是要做一个高性能的单机引擎，需要做各种细致的优化，而 RocksDB 是一个非常优秀的开源的单机存储引擎，可以满足我们对单机引擎的各种要求，而且还有 Facebook 的团队在做持续的优化，这样我们只投入很少的精力，就能享受到一个十分强大且在不断进步的单机引擎。当然，我们也为 RocksDB 贡献了一些代码，希望这个项目能越做越好。这里可以简单的认为 RocksDB 是一个单机的 Key-Value Map。

Raft

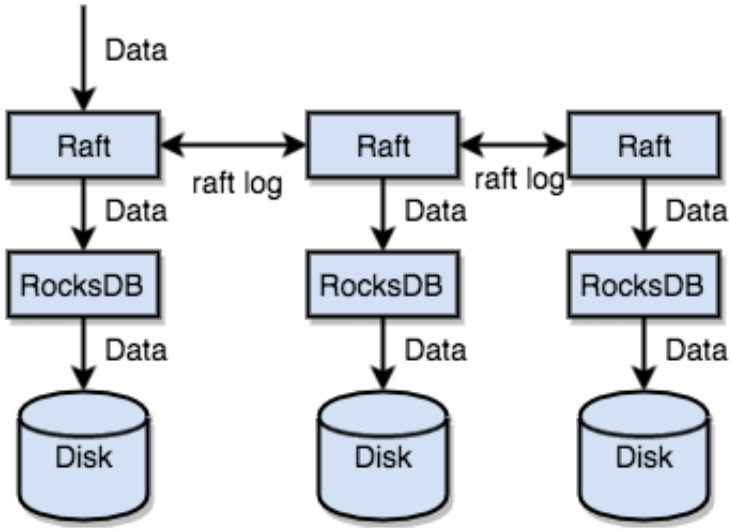
好了，万里长征第一步已经迈出去了，我们已经为数据找到一个高效可靠的本地存储方案。俗话说，万事开头难，然后中间难，最后结尾难。接下来我们面临一件更难的事情：如何保证单机失效的情况下，数据不丢失，不出错？简单来说，我们需要想办法把数据复制到多台机器上，这样一台机器挂了，我们还有其他的机器上的副本；复杂来说，我们还需要这个复制方案是可靠、高效并且能处理副本失效的情况。听上去比较难，但是好在我们有 Raft 协议。Raft 是一个一致性算法，它和 Paxos 等价，但是更加易于理解。Raft 的论文，感兴趣的可以看一下。本文只会对 Raft 做一个简要的介绍，细节问题可以参考论文。另外提一点，Raft 论文只是一个基本方案，严格按照论文实现，性能会很差，我们对 Raft 协议的实现做了大量的优化，具体的优化细节可参考我司首席架构师 tangliu 同学的《TiKV 源码解析系列 - Raft 的优化》这篇文章。

Raft 是一个一致性协议，提供几个重要的功能：

1. Leader 选举

2. 成员变更
3. 日志复制

TiKV 利用 Raft 来做数据复制，每个数据变更都会落地为一条 Raft 日志，通过 Raft 的日志复制功能，将数据安全可靠地同步到 Group 的多数节点中。



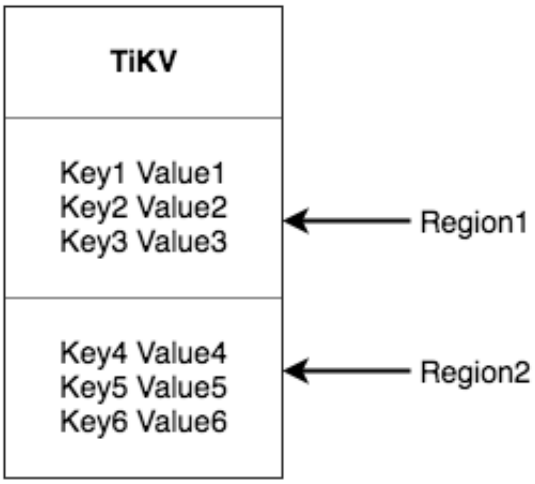
到这里我们总结一下，通过单机的 RocksDB，我们可以将数据快速地存储在磁盘上；通过 Raft，我们可以将数据复制到多台机器上，以防单机失效。数据的写入是通过 Raft 这一层的接口写入，而不是直接写 RocksDB。通过实现 Raft，我们拥有了一个分布式的 KV，现在再也不用担心某台机器挂掉了。

Region

讲到这里，我们可以提到一个 **非常重要的概念**：**Region**。这个概念是理解后续一系列机制的基础，请仔细阅读这一节。

前面提到，我们将 TiKV 看做一个巨大的有序的 KV Map，那么为了实现存储的水平扩展，我们需要将数据分散在多台机器上。这里提到的数据分散在多台机器上和 Raft 的数据复制不是一个概念，在这一节我们先忘记 Raft，假设所有的数据都只有一个副本，这样更容易理解。

对于一个 KV 系统，将数据分散在多台机器上有两种比较典型的方案：一种是按照 Key 做 Hash，根据 Hash 值选择对应的存储节点；另一种是分 Range，某一段连续的 Key 都保存在一个存储节点上。TiKV 选择了第二种方式，将整个 Key-Value 空间分成很多段，每一段是一系列连续的 Key，我们将每一段叫做一个 **Region**，并且我们会尽量保持每个 Region 中保存的数据不超过一定的大小(这个大小可以配置，目前默认是 96mb)。每一个 Region 都可以用 StartKey 到 EndKey 这样一个左闭右开区间来描述。



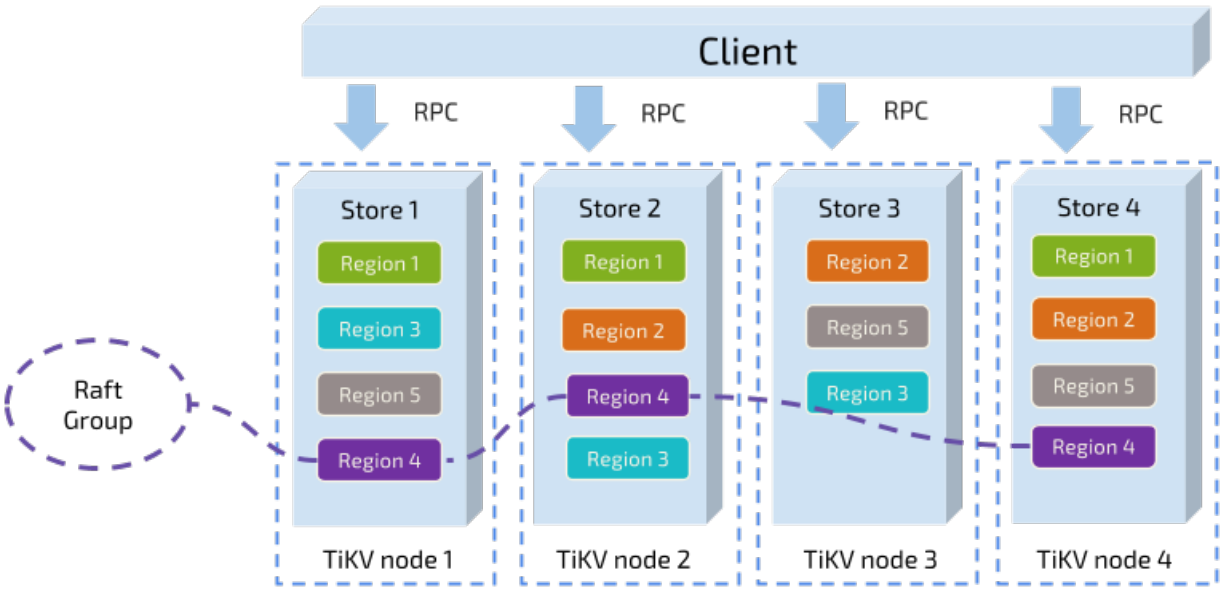
注意，这里的 Region 还是和 SQL 中的表没什么关系！ 请各位继续忘记 SQL，只谈 KV。 将数据划分成 Region 后，我们将会做 **两件重要的事情**：

- 以 Region 为单位，将数据分散在集群中所有的节点上，并且尽量保证每个节点上服务的 Region 数量差不多
- 以 Region 为单位做 Raft 的复制和成员管理

这两点非常重要，我们一点一点来说。

先看第一点，数据按照 Key 切分成很多 Region，每个 Region 的数据只会保存在一个节点上面。我们的系统会有一个组件来负责将 Region 尽可能均匀的散布在集群中所有的节点上，这样一方面实现了存储容量的水平扩展（增加新的结点后，会自动将其他节点上的 Region 调度过来），另一方面也实现了负载均衡（不会出现某个节点有很多数据，其他节点上没什么数据的情况）。同时为了保证上层客户端能够访问所需要的数据，我们的系统中也会有一个组件记录 Region 在节点上面的分布情况，也就是通过任意一个 Key 就能查询到这个 Key 在哪个 Region 中，以及这个 Region 目前在哪个节点上。至于是哪个组件负责这两项工作，会在后续介绍。

对于第二点，TiKV 是以 Region 为单位做数据的复制，也就是一个 Region 的数据会保存多个副本，我们将每一个副本叫做一个 Replica。Replica 之间是通过 Raft 来保持数据的一致（终于提到了 Raft），一个 Region 的多个 Replica 会保存在不同的节点上，构成一个 Raft Group。其中一个 Replica 会作为这个 Group 的 Leader，其他的 Replica 作为 Follower。所有的读和写都是通过 Leader 进行，再由 Leader 复制给 Follower。 大家理解了 Region 之后，应该可以理解下面这张图：



我们以 Region 为单位做数据的分散和复制，就有了一个分布式的具备一定容灾能力的 KeyValue 系统，不用再担心数据存不下，或者是磁盘故障丢失数据的问题。这已经很 Cool，但是还不够完美，我们需要更多的功能。

MVCC

很多数据库都会实现多版本控制（MVCC），TiKV 也不例外。设想这样的场景，两个 Client 同时去修改一个 Key 的 Value，如果没有 MVCC，就需要对数据上锁，在分布式场景下，可能会带来性能以及死锁问题。 TiKV 的 MVCC 实现是通过在 Key 后面添加 Version 来实现，简单来说，没有 MVCC 之前，可以把 TiKV 看做这样的：

```
Key1 -> Value
Key2 -> Value
……
KeyN -> Value
```

有了 MVCC 之后，TiKV 的 Key 排列是这样的：

```
Key1-Version3 -> Value
Key1-Version2 -> Value
Key1-Version1 -> Value
……
Key2-Version4 -> Value
Key2-Version3 -> Value
Key2-Version2 -> Value
Key2-Version1 -> Value
……
KeyN-Version2 -> Value
KeyN-Version1 -> Value
……
```

注意，对于同一个 Key 的多个版本，我们把版本号较大的放在前面，版本号小的放在后面（回忆一下 Key-Value 一节我们介绍过的 Key 是有序的排列），这样当用户通过一个 Key + Version 来获取 Value 的时候，可以将 Key 和 Version 构造出 MVCC 的 Key，也就是 Key-Version。然后可以直接 Seek(Key-Version)，定位到第一个大于等于这个 Key-Version 的位置。

事务

TiKV 的事务采用的是 Percolator 模型，并且做了大量的优化。事务的细节这里不详述，大家可以参考论文以及我们的其他文章。这里只提一点，TiKV 的事务采用乐观锁，事务的执行过程中，不会检测写写冲突，只有在提交过程中，才会做冲突检测，冲突的双方中比较早完成提交的会写入成功，另一方会尝试重新执行整个事务。当业务的写入冲突不严重的情况下，这种模型性能会很好，比如随机更新表中某一行的数据，并且表很大。但是如果业务的写入冲突严重，性能就会很差，举一个极端的例子，就是计数器，多个客户端同时修改少量行，导致冲突严重的，造成大量的无效重试。

其他

到这里，我们已经了解了 TiKV 的基本概念和一些细节，理解了这个分布式带事务的 KV 引擎的分层结构以及如何实现多副本容错。下一节会介绍如何在 KV 的存储模型之上，构建 SQL 层。

TiDB 事务处理

TiDB 的事务模型参考了 Percolator 论文，Percolator 事务模型详见上一篇文章-Google Percolator 的事务模型

本文将详细介绍事务在 tidb 中实现，主要内容包括

- 基本概念
- tidb 中一致性事务的实现
- tikv 中事务相关的接口逻辑
- tidb 事务如何做到 ACID

TIKV

tidb 为 tikv 的客户端，对外提供兼容 mysql 协议的分布式关系型数据库服务。

pd 提供两大功能

- 提供包括物理时间的全局唯一递增时间戳 tso
- 管理 raft-kv 集群

tikv 对外提供分布式 kv 存储引擎，同时实现了 mvcc 相关的接口 - - 方便客户端实现 ACID 事务。

Columns in TIKV

tikv 底层用 raft+rocksdb 组成的 raft-kv 作为存储引擎，具体落到 rocksdb 上的 column 有四个，除了一个用于维护 raft 集群的元数据外，其它 3 个皆为了保证事务的 mvcc, 分别为 lock, write, default，详情如下：

Lock

事务产生的锁，未提交的事务会写本项，会包含primary lock的位置。其映射关系为

```
${key}=>${start_ts,primary_key,..etc}
```

Write

已提交的数据信息，存储数据所对应的时间戳。其映射关系为

```
${key}_${commit_ts}=>${start_ts}
```

Default(data)

具体存储数据集，映射关系为

```
${key}_${start_ts} => ${value}
```

Primary

TiDB 对于每个事务，会从涉及到改动的所有 Key 中选中一个作为当前事务的 Primary Key。 在最终提交时，以 Primary 提交是否成功作为整个事务是否执行成功的标识，从而保证了分布式事务的原子性。

有了 Primary key 后，简单地说事务两阶段提交过程如下：

1. 从当前事务涉及改动的 keys 选中一个作为 primary key, 剩余的则为 secondary keys
2. 并行 prewrite 所有 keys。 这个过程中，所有 key 会在系统中留下一个指向 primary key 的锁。
3. 第二阶段提交时，首先 commit primary key ,若此步成功，则说明当前事务提交成功。
4. 异步并行 commit secondary keys

一个读取过程如下：

1. 读取 key 时，若发现没有冲突的锁，则返回对应值，结束。
2. 若发现了锁，且当前锁对应的 key 为 primary： 若锁尚未超时，等待。若锁已超时，Rollback 它并获取上一版本信息返回，结束。
3. 若发现了锁，且当前锁对应的 key 为 secondary, 则根据其锁里指定的 primary

找到 primary所在信息，根据 primary 的状态决定当前事务是否提交成功，返回对应具体值。

TIDB 事务处理流程

注意：所有涉及重新获取 tso 重启事务的两阶段提交的地方，会先检查当前事务是否可以满足重试条件：只有单条语句组成的事务才可以重新获取tso作为start_ts。

1. client 向 tidb 发起开启事务 begin
2. tidb 向 pd 获取 tso 作为当前事务的 start_ts
3. client 向 tidb 执行以下请求：
    - 读操作，从 tikv 读取版本 start_ts 对应具体数据.
    - 写操作，写入 memory 中。
4. client 向 tidb 发起 commit 提交事务请求
5. tidb 开始两阶段提交。
6. tidb 按照 region 对需要写的数据进行分组。
7. tidb开始 prewrite操作：向所有涉及改动的 region 并发执行 prewrite 请求。若其中某个prewrite 失败，根据错误类型决定处理方式：
    - KeyIsLock：尝试 Resolve Lock 后，若成功，则重试当前 region 的 prewrite[步骤7]。否则，重新获取 tso 作为 start_ts 启动 2pc 提交（步骤5）。
    - WriteConfict 有其它事务在写当前 key, 重新获取 tso 作为 start_ts 启动 2pc 提交（步骤5）。
    - 其它错误，向 client 返回失败。
8. tidb 向 pd 获取 tso 作为当前事务的 commit_ts。
9. tidb开始 commit:tidb 向 primary 所在 region 发起 commit

    。若 commit primary 失败，则先执行 rollback keys,然后根据错误判断是否重试:
    - LockNotExist 重新获取 tso 作为 start_ts 启动 2pc 提交（步骤5）。
    - 其它错误，向 client 返回失败。
10. tidb 向 tikv 异步并发向剩余 region 发起 commit。
11. tidb 向 client 返回事务提交成功信息。

TiKV 事务处理细节

Prewrite

Prewrite是事务两阶段提交的第一步，其从pd获取代表当前物理时间的全局唯一时间戳作为当前事务的 start_ts，尝试对所有被写的元素加锁(为应对客户端故障，tidb 为所有需要写的key选出一个作为primary,其余的作为secondary)，将实际数据存入 rocksdb。其中每个key的处理过程如下，中间出现失败，则整个prewrite失败：

1. 检查 write-write 冲突：从 rocksdb 的write 列中获取当前 key 的最新数据，若其 commit_ts 大于等于start_ts,说明存在更新版本的已提交事务，向 tidb 返回 WriteConflict 错误，结束。
2. 检查 key 是否已被锁上，如果 key 的锁已存在，收集 KeyIsLock 的错误，处理下一个 key
3. 往内存中的 lock 列写入 lock(start_ts,key) 为当前key加锁,若当前key被选为 primary, 则标记为 primary,若为secondary,则标明指向primary的信息。
4. 若当前 value 较小，则与 lock 存在一起，否则，内存中存入 default(start_ts,key,value)。

处理完所有数据后，若存在 KeyIsLock 错误，则向 tidb 返回所有 KeyIsLocked 信息。
否则，提交数据到 raft-kv 持久化，当前 prewrite 成功。

Commit

1. tidb 向 tikv 发起第二阶段提交 commit(keys,commit_ts,start_ts)
2. 对于每个 key 依次检查其是否合法，并在内存中更新（步骤3-4）
3. 检查 key 的锁，若锁存在且为当前 start_ts 对应的锁，则在内存中添加 write(key,commit_ts,start_ts),删除 lock(key,start_ts)，继续执行下一个 key(跳至步骤3)。否则，执行步骤 4
4. 获取当前 key 的 start_ts 对应的数据 write(key,start_ts,commit_ts)，若存在，说明已被提交过，继续执行下一个 key(跳至步骤4)。否则，返回未找到锁错误到 tidb，结束。
5. 到底层 raft-kv 中更新 2-4 步骤产生的所有数据 - - 这边保证了原子性。
6. tikv 向 tidb 返回 commit 成功。

Rollback

当事务在两阶段提交过程中失败时， tidb 会向当前事务涉及到的所有 tikv 发起回滚操作。

1. `tidb ` 向 tikv 发起 rollback(keys,start_ts), 回滚当前 region 中 start_ts 所在的 key 列表
2. 对于每个 key, tikv 依次检查其合法性，并进行回滚(依次对每个 key 执行 3-4)
3. 检查当前 key 的锁，情况如下：
   * 若当前 (start_ts,key) 所对应的锁存在，则在内存中删除该锁,继续回滚下一个 key(跳至步骤2)
   * 若当前事务所对应的锁不存在，则进入步骤 4 检查提交情况
4. tikv 从 raft-kv 中获取到当前 (key,start_ts) 对应的提交纪录：

- 若 commit(key,commit_ts,start_ts) 存在，且状态为PUT／DELETE，则返回 tidb 告知事务已被提交，回滚失败，结束。
- 若 commit(key,commit_ts,start_ts) 存在，且状态为 Rollback, 说明当前 key 已被 rollback 过，继续回滚下一个 key(跳至步骤2)
- 若 (key,start_ts) 对应的提交纪录不存在，说明当前 key 尚未被 prewrite 过，为预防 prewrite 在之后过来，在这里留下 (key,start_ts,rollback), 继续回滚下一个 key(跳至步骤2)

5. 将步骤 2-4 中更新的内容持久化到 raft-kv
6. tikv 向 tidb 返回回滚成功。

Resolve Lock

tidb 在执行 prewrite, get 过程中，若遇到锁，在锁超时的情况下，会向 tikv 发起清锁操作。

1. tidb 向 tikv 发起 Resolve(start_ts,commit_ts)
2. tikv 依次找出所有 lock.ts==start_ts 的锁并执行对应的清锁操作（循环执行步骤3-5）。
3. tikv 向 raft-kv 获取下一堆lock.ts==start_ts 的锁。
4. 若没有剩余锁，退出循环，跳至步骤6。
5. 对本批次获取到的锁，根据情况进行清锁操作
   * 若 commit_ts 为空，则执行回滚
   * 若 commit_ts 有值，则执行提交。
6. 将2-5 产生的更新持久化到 raft-kv
7. tikv 向 tidb 返回清锁成功。

Get

1. tidb 向 tikv 发起 get(key,start_ts) 操作
2. tikv 检查当前 key 的锁状态：若锁存在且 lock.ts<start_ts, 即 start_ts 这一刻该值是被锁住的，则返回 tidb 被锁住错误，结束。
3. 尝试获取 start_ts 之前的最近的有效提交，初始化 版本号version 为 start_ts－1
4. 从 write 中获取 commit_ts<=version 的最大 commit_ts 对应纪录：
   * 若 write_type=PUT,即有效提交，返回 tidb 当前版本对应值。
   * 若 write 不存在或者 write_type=DEL, 即没有出现过该值，或该值最近已被删除，返回tidb空
   * 若 write_type=rollback,commit_ts，则 version=commit_ts-1, 继续查找下一个最近版本（跳至步骤3）

GC

1. tidb 向 tikv 发起 GC操作，要求清理 safe-point 版本之前的所有无意义版本
2. tikv 分批进行 GC

3. tikv 向 raft-kv 从 write 列中获取一批提交纪录，若上一批已处理过，则从上一批最后一个开始扫。若未获取到数据，则本次 GC 完成，返回 tidb 成功。
4. tikv对本批次的所有 key 挨个进行GC:
    - 清理所有小于 safe-point 的 Rollback 和 Lock
    - 若小于 safe-point 的除了Rollback 和 Lock外第一个为 delete, 清理所有safe-point 之前的数据。
    - 若小于 safe-point 的除了Rollback 和 Lock外第一个为 PUT, 保留该提交，清理所有该提交之前的数据。
5. tikv 向 raft-kv 更新步骤 4 的改动。进入下一批清理 （步骤3）.

ACID in TIDB

原子性

```
两阶段提交时，通过 `primary key` 保证原子性。 `primary key` commit 成功与否决定事务成功与否。
```

一致性

隔离性

通过两阶段提交，保证隔离级别为 RR

持久性

tikv 保证持久化。

Why 2 PC?

1 PC 存在的问题

1 pc 无法保证隔离性为 RR.

假设初始状态下，(A,t0)=>v1, 以下操作有序发生时：

1. txn1: 向 pd 获取 tso1 作为当前事务的 start_ts。
2. txn2: 向 pd 获取 tso2 作为当前事务的 start_ts, 此时 tso2>tso1
3. txn2: 向 pd 获取 A, 此时获取到 A 在 tikv 上的小于等于 tso2 的最新版本为 v1.
4. txn1: 向 pd 更新 A为 v2, 此时tikv 上有记录：（A,tso1）=>v2
5. txn2: 向 pd 获取 A, 此时获取到 A 在 tikv 上的小于等于 tso2 的最新版本为 v2.

2 PC 保证隔离级别

两阶段提交简介：
第一阶段 prewrite:

1. 获取一个包含当前物理时间的、全局唯一递增的时间戳 t1 作为当前事务的 start_ts
2. lock(key,start_ts), save data(key,start_ts,data)
   第二阶段 commit 数据：
3. 获取一个包含当前物理时间的、全局唯一递增的时间戳 t2 作为当前事务的 commit_ts
4. commit(key,commit_ts, start_ts)

事务读取数据步骤如下：

1. 获取一个包含当前物理时间的、全局唯一递增的时间戳 t1 作为当前事务的 start_ts.
2. 检查当前查询 key 的锁，若锁存在，且 lock.ts<t1, 说明这一刻 key 正在被写入，需要等待写入事务完成再读取。
3. 到这一步时，说明 要么 锁不存在，要么 lock.ts > t1, 这两种情况都能说明， 下一个该 key 的 commit_ts 一定会大于当前的 t1, 所以可以直接读取当前小于 t1 的最大 commit_ts 对应的数据。

综上，两阶段提交可以保证事务的隔离级别为 RR，示例如下：

假设初始状态下，`(A,t0)=>v1`, 现有事务1－txn1, 事务2-txn2。其中 txn-1 将会修改 A 的值为 v2。假设 txn－1 的 start_ts=s1, commit_ts=c1, 事务 txn-2 的 start_ts=s2, 那么有：

- 若 s1>s2, 那么 txn2 读取的是 s2 之前的数据，无论如何都不会读取到 txn-1 的更新的数据。

- 若 s1 < s2, 则有以下两种情况。
    1. 若 s2 读取 A 时， 若未发现lock(s1)， 如上图中，txn-2 第一次获取数据时，此时 txn-1 尚未提交，由于 commit_ts 只有在 prewrite 完成后才能获取，所以可以保证 c1>s2, 也就是说，s2 读取不到 txn-1 的数据。
    2. 若 s2 读取 A 时， 若发现了 lock(s1), 如上图中，txn-2 第二次获取数据时的场景， 此时无法确定 c1 与 s2 的大小，所以此时 txn-2 会等待 txn-1 commit, 当 txn-1 commit 结束后，txn-2 才会根据 txn-1 的 commit_ts 确定是否获取 txn-1 的更新数据。也就是说，发现有锁时，txn-2 一定会等待直到确定与 txn-1 的 commit_ts 的大小才会决定获取哪份数据。

综上可见， 两阶段提交能很好的保证事务发生的时序，从而保证了事务的隔离级别为 可重复读（RR）

TiDB 查询语言

SQL 基本操作

成功部署 TiDB 集群之后，便可以在 TiDB 中执行 SQL 语句了。因为 TiDB 兼容 MySQL，你可以使用 MySQL 客户端连接 TiDB，并且大多数情况下可以直接执行 MySQL 语句。

SQL 是一门声明性语言，它是数据库用户与数据库交互的方式。它更像是一种自然语言，好像在用英语与数据库进行对话。本文档介绍基本的 SQL 操作。完整的 SQL 语句列表，参见 TiDB SQL 语法详解。

分类

SQL 语言通常按照功能划分成以下的 4 个部分：

- DDL (Data Definition Language)：数据定义语言，用来定义数据库对象，包括库、表、视图和索引等。
- DML (Data Manipulation Language)：数据操作语言，用来操作和业务相关的记录。
- DQL (Data Query Language)：数据查询语言，用来查询经过条件筛选的记录。
- DCL (Data Control Language)：数据控制语言，用来定义访问权限和安全级别。

常用的 DDL 功能是对象（如表、索引等）的创建、属性修改和删除，对应的命令分别是 CREATE、ALTER 和 DROP。

查看、创建和删除数据库

TiDB 语境中的 Database 或者说数据库，可以认为是表和索引等对象的集合。

使用 `SHOW DATABASES` 语句查看系统中数据库列表：

```
SHOW DATABASES;
```

使用名为 `mysql` 的数据库：

```
use mysql;
```

使用 `SHOW TABLES` 语句查看数据库中的所有表。例如：

```
SHOW TABLES FROM mysql;
```

使用 `CREATE DATABASE` 语句创建数据库。语法如下：

```
CREATE DATABASE db_name [options];
```

例如，要创建一个名为 `samp_db` 的数据库，可使用以下语句：

```
CREATE DATABASE IF NOT EXISTS samp_db;
```

添加 `IF NOT EXISTS` 可防止发生错误。

使用 `DROP DATABASE` 语句删除数据库。例如：

```
DROP DATABASE samp_db;
```

创建、查看和删除表

使用 CREATE TABLE 语句创建表。语法如下：

```
CREATE TABLE table_name column_name data_type constraint;
```

例如，要创建一个名为 person 的表，包括编号、名字、生日等字段，可使用以下语句：

```
CREATE TABLE person (
    id INT(11),
    name VARCHAR(255),
    birthday DATE
    );
```

使用 SHOW CREATE 语句查看建表语句，即 DDL。例如：

```
SHOW CREATE table person;
```

使用 DROP TABLE 语句删除表。例如：

```
DROP TABLE person;
```

创建、查看和删除索引

索引通常用于加速索引列上的查询。对于值不唯一的列，可使用 CREATE INDEX 或 ALTER TABLE 语句创建普通索引。例如：

```
CREATE INDEX person_id ON person (id);
```

或者：

```
ALTER TABLE person ADD INDEX person_id (id);
```

对于值唯一的列，可以创建唯一索引。例如：

```
CREATE UNIQUE INDEX person_unique_id ON person (id);
```

或者：

```
ALTER TABLE person ADD UNIQUE person_unique_id (id);
```

使用 SHOW INDEX 语句查看表内所有索引：

```
SHOW INDEX from person;
```

使用 ALTER TABLE 或 DROP INDEX 语句来删除索引。与 CREATE INDEX 语句类似，DROP INDEX 也可以嵌入 ALTER TABLE 语句。例如：

```
DROP INDEX person_id ON person;
ALTER TABLE person DROP INDEX person_unique_id;
```

注意：DDL 操作不是事务，在执行 DDL 时，不需要对应 COMMIT 语句。

常用的 DML 功能是对表记录的新增、修改和删除，对应的命令分别是 INSERT、UPDATE 和 DELETE。

记录的增删改

使用 INSERT 语句向表内插入表记录。例如：

```
INSERT INTO person VALUES("1","tom","20170912");
```

使用 INSERT 语句向表内插入包含部分字段数据的表记录。例如：

```
INSERT INTO person(id,name) VALUES("2","bob");
```

使用 UPDATE 语句向表内修改表记录的部分字段数据。例如：

```
UPDATE person SET birthday="20180808" WHERE id=2;
```

使用 DELETE 语句向表内删除部分表记录。例如：

```
DELETE FROM person WHERE id=2;
```

注意：UPDATE 和 DELETE 操作如果不带 WHERE 过滤条件是对全表进行操作。

DQL 数据查询语言是从一个表或多个表中检索出想要的数据行，通常是业务开发的核心内容。

查询数据

使用 SELECT 语句检索表内数据。例如：

```
SELECT * FROM person;
```

在 SELECT 后面加上要查询的列名。例如：

```
SELECT name FROM person;
+------+
| name |
+------+
| tom  |
+------+
```

使用 WHERE 子句，对所有记录进行是否符合条件的筛选后再返回。例如：

```
SELECT * FROM person where id<5;
```

常用的 DCL 功能是创建或删除用户，和对用户权限的管理。

创建、授权和删除用户

使用 CREATE USER 语句创建一个用户 tiuser，密码为 123456：

```
CREATE USER 'tiuser'@'localhost' IDENTIFIED BY '123456';
```

授权用户 tiuser 可检索数据库 samp_db 内的表：

```
GRANT SELECT ON samp_db.* TO 'tiuser'@'localhost';
```

查询用户 tiuser 的权限：

```
SHOW GRANTS for tiuser@localhost;
```

删除用户 tiuser：

```
DROP USER 'tiuser'@'localhost';
```

TiDB 典型应用案例：

猿辅导是国内拥有最多中小学生用户的在线教育机构，旗下有猿题库、小猿搜题、猿辅导三款在线教育 APP，为用户提供在线题库、拍照搜题、名师在线辅导相关的服务。其中，猿辅导 APP 已经有超过 116 万付费用户，提供小学英语、奥数，和初中高中全学科的直播辅导课程，全国任何地区的中小学生，都可以享受在家上北京名师辅导课的服务。

海量的题库、音视频答题资料、用户数据以及日志，对猿辅导后台数据存储和处理能力都提出了严峻的要求。

猿辅导的业务决定了其后台系统具有以下特点：

- 数据体量大，增速快，存储系统需要能够灵活的水平扩展；
- 有复杂查询，BI 方面的需求，可以根据索引，例如城市、渠道等，进行实时统计；
- 数据存储要具备高可用、高可运维性，实现自动故障转移。

在最初方案选型时，猿辅导初期考虑用单机 MySQL。但根据业务发展速度预估，数据存储容量和并发压力很快就会达到单机数据库的处理瓶颈。如果在 MySQL 上加入分库中间件方案，则一定要指定 sharding key，这样是无法支持跨 shard 的分布式事务。同时 proxy 的方案对业务层的侵入性较强，开发人员必须了解数据库的分区规则，无法做到透明。

除此之外，分库分表很难实现跨 shard 的聚合查询，例如全表的关联查询、子查询、分组聚合等业务场景，查询的复杂度需要转嫁给开发者。即使某些中间件能实现简单的 join 支持，但是仍然没有办法保证查询的正确性。另外广播是一个没有办法 Scale 的方案，当集群规模变大，广播的性能开销是很大的。同时，传统 RDBMS 上 DDL 锁表的问题，对于数据量较大的业务来说，锁定的时间会很长，如果使用 gh-ost 这样第三方工具来实现非阻塞 DDL，额外的空间开销会比较大，而且仍然需要人工的介入确保数据的一致性，最后切换的过程系统可能会有抖动。可以说，运维的复杂性是随着机器数量指数级增长，而扩容复杂度则是直接转嫁给了 DBA。

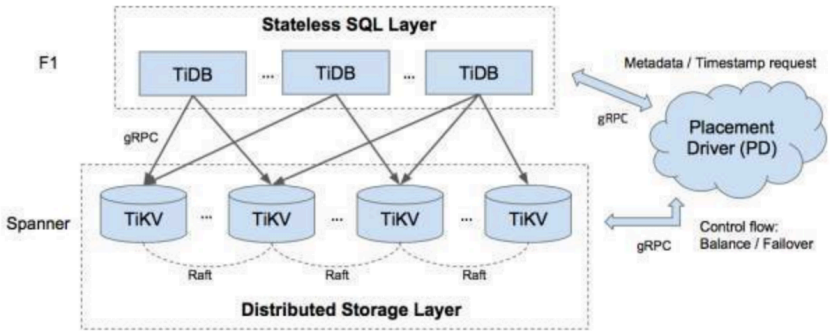
最终，猿辅导的后台开发同学决定寻求一个彻底的分布式存储解决方案。通过对社区方案的调研，猿辅导发现分布式关系型数据库 TiDB 项目。

TiDB 是一款定位于在线事务处理/在线分析处理（HTAP）的融合型数据库产品，具备在线弹性水平扩展、分布式强一致性事务、故障自恢复的高可用、跨数据中心多活等核心特性；对业务没有任何侵入性，能优雅的替换传统的数据库中间件、数据库分库分表等 Sharding 方案，并在此过程中保证了事务的 ACID 特性。同时它也让开发运维人员不用关注数据库 Scale 的细节问题，专注于业务开发，极大的提升研发的生产力。用户可以把 TiDB 当作一个容量无限扩展的单机数据库，复杂的分布式事务和数据复制由底层存储引擎来支持，开发者只需要集中精力在业务逻辑的开发上面。

特性	传统 MySQL 中间件/分库分表	TiDB
强一致的分布式事务	✘	√
水平扩展	✘	√
复杂查询（JOIN / GTOUP BY / ...）	✘	√
无人工介入的高可用	✘	√
业务兼容性	Low	High

图为 TiDB 与传统的 MySQL 中间件方案的一些对比

TiDB 集群主要分为三个组件：TiDB Server、TiKV Server、PD Server。



TiDB 整体架构图

TiDB Server 负责处理 SQL 请求，随着业务的增长，可以简单的添加 TiDB Server 节点，提高整体的处理能力，提供更高的吞吐。TiKV 负责存储数据，随着数据量的增长，可以部署更多的 TiKV Server 节点解决数据 Scale 的问题。PD 会在 TiKV 节点之间以 Region 为单位做调度，将部分数据迁移到新加的节点上。所以企业在业务的早期，可以只部署少量的服务实例，随着业务量的增长，按照需求添加 TiKV 或者 TiDB 实例。

在实际上线的部署设置中，猿辅导选择了 2 TiDB + 3 TiKV + 3 PD 的架构，随着业务数据的增加可以弹性扩容，数据条数每天 500w，日常库中数亿条记录，峰值 QPS 1000。

猿辅导的用户端会做一些直播过程的音视频质量的数据收集，比如丢包，延迟，质量打分。然后客户端把这些数据发回服务器，服务器把这些数据存到 TiDB 上。

在**猿辅导研发副总裁郭常圳**看来：“TiDB 是一个很有野心的项目，从无到有的解决了 MySQL 过去遇到的扩展性问题，在很多场合下也有 OLAP 的能力，省去了很多数据仓库搭建成本和学习成本。这在业务层是非常受欢迎的。”对于接下来的计划，猿辅导预计在其他分库分表业务中，通过 syncer 同步，进行合并，然后进行统计分析。

实际上，类似猿辅导这种场景的并不是第一家，在互联网快速发展下，大量的企业面对着业务激增的情况。TiDB 灵活的水平扩展能力，能够满足企业业务快速发展的需要。

参考：

https://docs.pingcap.com/zh/tidb/dev/tidb-architecture

https://docs.pingcap.com/zh/tidb/stable/basic-sql-operations

https://karellincoln.github.io/2018/06/07/TiDB-analysis-application/

https://en.wikipedia.org/wiki/TiDB

https://en.wikipedia.org/wiki/NewSQL

https://andremouche.github.io/tidb/transaction_in_tidb.html

https://docs.pingcap.com/zh/tidb/dev/transaction-overview

https://pingcap.com/blog-cn/tidb-internal-1/

https://pingcap.com/blog-cn/tidb-internal-2/

https://pingcap.com/blog-cn/tidb-internal-3/

https://pingcap.com/blog-cn/how-do-we-build-tidb/

https://pingcap.com/cases-cn/

https://pingcap.com/cases-cn/user-case-yuanfudao/

二、答案：

略

三、答案：

仿射密码

仿射密码加密思路：首先将明文乘以密钥的一部分，然后在加上密钥的剩余部分。

$$\text{假设：}\ x, y, a, b \in Z_{26} \tag{1}$$

$$\text{加密：}\ e_k(x) = y \equiv (a \cdot x\ +\ b\ )\ mod\ 26 \tag{2}$$

$$\text{解密：}\ d_k(y) = x \equiv (a^{-1} \cdot (\ y\ -\ b\ ))\ mod\ 26 \tag{3}$$

$$\text{密钥为：}\ k = (a, b), \text{且满足限制条件}\ gcd(a, 26) = 1 \tag{4}$$

解密可以很容易地从加密函数中推导出来：

$$a \cdot x + b \equiv y\ mod\ 26 \tag{5}$$

$$a \cdot x \equiv (y - b)\ mod\ 26 \tag{6}$$

$$x \equiv (a^{-1} \cdot (y - b))\ mod\ 26 \tag{7}$$

gcd(a, 26) = 1这个限制条件来源于这样一个事实： 解密时需要求密钥参数a的逆元。如果逆元存在，则元素a的模数必须为互素，因此， a必须在如下集合中：

$$a \in \{1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25\} \tag{8}$$

那么，怎么样才能找到 a^{-1}? 只能通过暴力法遍历，对于给定的a, 只需要一次尝试所有可能的值a^{-1}，直到得到

$$a \cdot a^{-1} \equiv 1 \ mod \ 26 \tag{9}$$

因此本题的解密函数为

$$d_k(y) = 9 \cdot (x - 4) \ mod \ 26 \tag{10}$$

加密后的文本： JZCGCGEGCOXLQQVEOXLQUTETTCRQKCXZQD

解密后的文本： THISISASIMPLEEXAMPLEOFAFFINECIPHER

代码: https://github.com/DaviRain-Su/my_rust_road/blob/master/rust_project/Affine-cipher/src/main.rs

参考：

https://www.cnblogs.com/zishu/p/8650214.html

https://learnku.com/docs/cryptography/28-affine-cryptography/8932

https://baike.baidu.com/item/%E4%BB%BF%E5%B0%84%E5%AF%86%E7%A0%81

四、答案：

数组中有一个数字出现的次数超过了数组长度的一半， 也就说它出现的次数比其他所有数字出现的次数还要多，因此，我们可以考虑在遍历数组的时候保存两个值： 一个是数组中的一个数字； 另一个是次数，当我们遍历到下一个数字的时候，如果写一个数字和我们之前保存的数字相同，则次数加1； 如果下一个数字和我们之前保存的数字不同，则次数减1。如果次数为零，那么我们需要保存下一个数字，并把次数设为2.由于我们要找的数字出现的次数比其他所有出现的次数之和还要多，那么要找的数字肯定是最后一次把次数设为1时所对应的数字。

```
int more_than_half_num(int* numbers, int length){
  if(check_invalid_array(numbers, length))
    return 0;

  int result = numbers[0];
  int times = 1;
```

```
  for (int i = 1; i < length; ++i) {
    if (times == 0) {
      result = numbers[i];
      times = 1;
    }else if (numbers[i] == result)
      times ++;
    else
      times --;
  }

  if (!check_more_than_half(numbers, length, result))
    result = 0;
  return result;
}

bool g_input_invalid = false;
bool check_invalid_array(int* numbers, int length) {
  g_input_invalid = false;
  if (numbers == nullptr || length <= 0)
    g_input_invalid = true;
  return g_input_invalid;
}

bool check_more_than_half(int* numbers, int length, int number) {
  int times = 0;
  for (int i = 0; i < length; ++i) {
    if (numbers[i] == number)
      times ++;
  }

  bool is_more_than_half = true;
  if (times * 2 <= length) {
    g_input_invalid = true;
    is_more_than_half = false;
  }
  return is_more_than_half;
}
```

参考：

剑指offer， P206

五、答案：