链滴

SDB：纯 golang 开发、数据结构丰富、持久化的 NoSQL 数据库

作者：yemingfeng

原文链接：https://ld246.com/article/1636375589459

来源网站：链滴

SDB：纯 golang 开发、数据结构丰富、持久化的 NoSQL 据库

为什么需要 SDB?

试想以下业务场景：

- 计数服务：对内容的点赞、播放等数据进行统计
- 评论服务：发布评论后，查看某个内容的评论列表
- 推荐服务：每个用户有一个包含内容和权重的推荐列表

以上几个业务场景，都可以通过 MySQL + Redis 的方式实现。 这里的问题是：MySQL 更多的是充持久化的能力，Redis 充当的是在线服务的读写能力。

那么只使用 Redis 行不行？ 答案是否定的，因为 Redis 无法保证数据不丢失。

那有没有一种存储能够支持高级的数据结构，并能够将数据进行持久化的呢?

答案是：非常少的。有些数据库要么是支持的数据结构不够丰富，要么是接入成本太高，要么是不可。

为了解决上述问题，SDB 产生了。**SDB 提供了非常丰富的数据结构和持久化能力。**

SDB 简单介绍

- 纯 golang 开发，核心代码不超过 1k，代码易读
- 数据结构丰富
  - string
  - list
  - set
  - sorted set
  - bloom filter
  - hyper log log
  - pub sub
  - geo hash(规划中)
  - 倒排索引(规划中)
  - 向量检索(规划中)
- 持久化
  - 使用 pebble 作为存储引擎

快速使用

服务端使用

```
sh ./scripts/quick_start.sh
```

客户端使用

```
package main

import (
    "fmt"
    pb "github.com/yemingfeng/sdb/server/proto"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "log"
)

func main() {
    conn, err := grpc.Dial(":9000", grpc.WithInsecure())
    if err != nil {
        fmt.Printf("faild to connect: %v", err)
    }
    defer conn.Close()

    // 连接服务器
    c := pb.NewSDBClient(conn)
    setResponse, err := c.Set(context.Background(),
        &pb.SetRequest{Key: "hello", Val: "world"})
    log.Printf("setResponse: %v, err: %v", setResponse, err)
    getResponse, err := c.Get(context.Background(),
        &pb.GetRequest{Key: "hello"})
    log.Printf("getResponse: %v, err: %v", getResponse, err)
}
```

更多客户端例子

- string 操作
- list 操作
- set 操作
- sorted set 操作
- bloom filter 操作
- pub sub 操作

SDB 背后的思考

SDB 存储引擎选型

SDB 项目最核心的问题是数据存储方案的问题。

首先，我们不可能手写一个存储引擎。这个工作量太大，而且不可靠。我们得在开源项目中找到适合

DB 定位的存储方案。

SDB 需要能够提供高性能读写能力的存储引擎。 单机存储引擎方案常用的有：B+ 树、LSM 树、B 等。

还有一个前置背景，golang 在云原生的表现非常不错，而且性能堪比 C 语言，开发效率也高，所以 DB 首选使用纯 golang 进行开发。

那么现在的问题变成了：找到一款纯 golang 版本开发的存储引擎，这是比较有难度的。收集了一系资料后，找到了以下开源方案：

- LSM 树
  - go-leveldb ： 是一个 unstable 的项目，无法使用
  - syndtr-goleveldb ： 未在生产环境中使用过，不敢保证稳定性
  - badger ： 性能低于 leveldb
- B+ 树
  - boltdb-bolt ： 是废弃的项目，无法使用
  - etcd-bolt ： 主要是用于分布式环境下的数据同步，无法应对数据读写

以上存储引擎都或多或少存在问题。在不断的寻找中，找到了 pebble 存储引擎。pebble 是基于 golang-leveldb 项目实现了 RocksDB 的 KV 存储引擎，采用了 LSM
树的设计，提供了高性能读写能力。并且在 cockroachdb 数据库使用，有不错的稳定性。

最终 SDB 选择了 pebble 作为存储引擎。

SDB 数据结构设计

SDB 已经通过 pebble 解决了存储引擎的问题。 但如何在 KV 的存储引擎上增加数据结构的逻辑呢?

首先 pebble 提供了以下的接口能力：

- set
- get
- del
- batch
- iterator

接下来，我以支持 List 数据结构为例子，剖析下 SDB 是如何通过 pebble 存储引擎支持 List 的。

List 数据结构提供了以下接口：LPush、LPop、LExist、LRange、LCount。

如果一个 List 的 key 为：[hello]，该 List 的列表元素有：[aaa, ccc, bbb]，那么该 List 的每个元素在 pebble 的存储为：

```
pebble key -> pebble value
l/hello/{unique_ordering_key1} -> aaa
l/hello/{unique_ordering_key2} -> ccc
```

l/hello/{unique_ordering_key3} -> bbb

List 元素的 pebble key 生成策略：

- 数据结构前缀：List 都以 **l** 字符为前缀，Set 是以**s** 为前缀...
- List key 部分：List 的 key 为 hello
- unique_ordering_key：生成方式是通过雪花算法实现的，雪花算法保证局部自增
- pebble value 部分：List 元素真正的内容，如 aaa、ccc、bbb

为什么这么就能保证 List 的插入顺序呢?

这是因为 pebble 是 LSM 的实现，内部使用 key 的字典序排序。为了保证插入顺序，SDB 在 pebble key 中增加了 unique_ordering_key
作为排序的依据，从而保证了插入顺序。

有了 pebble key 的生成策略，一切都变得简单起来了。我们看看 LPush、LPop、LRange 的核心逻辑：

LPush

```
func LPush(key string, values []string) (bool, error) {
        // 批量写入器
    batchAction := store.NewBatchAction()
    for _, value := range values {
        // 为每个元素生成 unique_ordering_key
        batchAction.Set(generateListKey(key, util.GetOrderingKey()), value)
    }
    // 写入
    if err := batchAction.Commit(); err != nil {
        return false, err
    }
    return true, nil
}
```

LPop

在写入到 pebble 的时候，key 的生成是通过 unique_ordering_key 的方案。 无法直接在 pebble 找到 List 的元素在 pebble
key。在删除一个元素的时候，需要遍历 List 的所有元素，找到 value = 待删除的元素，然后进行删除。核心逻辑如下：

```
func LPop(key string, values []string) (bool, error) {
    it := store.NewBiIterator(
        &store.IteratorOption{Start: generateListPrefixKey(key)})
    defer it.Close()

    batchAction := store.NewBatchAction()
    for it.Next() {
        for _, value := range values {
            if value == it.Value() {
                batchAction.Del(it.Key())
```

```
            }
        }
    }
    if err := batchAction.Commit(); err != nil {
        return false, err
    }
    return true, nil
```

LRange

和删除逻辑类似，通过 iterator

接口进行遍历。 这里对反向迭代做了额外的支持

允许 Offset 传入 -1，代表从后进行迭代。

```
func LRange(key string, offset int32, limit int32) ([]string, error) {
    it := store.NewBiIterator(
        &store.IteratorOption{Start: generateListPrefixKey(key), Offset: offset})
    defer it.Close()

    index := int32(0)
    res := make([]string, limit)
    for it.Next() && index < limit {
        res[index] = it.Value()
        index++
    }
    return res[0:index], nil
}
```

以上就实现了对 List 的数据结构的支持。

其他的数据结构大体逻辑类似，其中 sorted_set

更加复杂些。可以自行查看。

SDB 通讯协议方案

解决完了存储和数据结构的问题后，SDB 面临了【最后一公里】的问题是通讯协议的选择。

SDB 的定位是支持多语言的，所以需要选择支持多语言的通讯框架。

grpc 是一个非常不错的选择，只需要使用 SDB proto 文件，就能通过 protoc 命令行工具自动生成种语言的客户端，解决了需要开发不同客户端的问题。

SDB 集群方案

SDB 的集群方案其实是在规划中的，之前也考虑了 TiKV 集群方案和 Redis 集群方案。

但目前 SDB 把注意力放在持久化、数据结构上。增加更多的数据结构，并将易用性做到极致。**之后实现集群方案。**