文 golang 版本的 ring buffer （变长，持久化）

golang **taowen** 3月13日发布

最终的实现代码： https://github.com/esdb/drbuffer
本文是整个 kafka agent 实现过程中的第一步：https://segmentfault.com/a/1190000004567774

内存结构

每个写入的packet格式如下

```
---
packet_size(uint16)
---
packet_body([]byte)
---
```

通过存储packet的长度实现变长数据的存储。目标是把这样的内存结构存储到一个ring buffer里。整体的ring buffer的结构如下

```
--- <-- nextReadFrom
packet 1
---
packet 2
--- <-- nextWriteFrom
...
```

除了每个packet是一个变长的结构体，貌似并没有什么特殊的。存储两个offset，一个指向读的位置，一个指向写入的位置。

持久化的需求其实并不困难，只需要把内存区域从一个文件mmap而来就可以。只要操作系统不挂（突然断电）数据的一致性和持久化都可以保证。对于日志，监控，离线分析等场景这样的持久化保证是足够的。利用这样的一个持久化的ring buffer可以隔离快速的写入方和不那么靠谱的后端。让业务进程对后端有一定的容错性。但是对于和金钱相关的业务事件，单机和单磁盘的可用性是不够的，那就需要额外添加网络复制来保证了。

golang使用mmap非常方便，直接就是[]byte，甚至可以用type alias转成别的类型来用。而且mmap是堆

外内存不归golang的gc管理，不会增加gc的负担。不得不说相比jvm，golang对于unsafe的操作花样（作死的方式）更多，堆内和堆外内存的使用体验也非常一致。

难点一：绕回

所谓ring buffer就是一个环，写入到了尾部之后就要绕回到头部。对于成员定长的队列，这个并不困难。写到最后一个slot之后就绕回到0就行了。但是对于成员变长的队列来说，绕回就变得非常困难。考虑一下几种情况

```
---
1
---
0
---
A
---
```

上面是3个byte，对应了一个完整的packet，内容是"A"。

```
---
1
---
0
---
A
---
free slot 1
---
```

如果只有一个空位，那么下一个packet的头部2个byte就会被分成两部分。

```
---
1
---
0
---
A
---
free slot 1
---
free slot 2
---
```

如果有两个空位，那么下一个packet的头部2个byte是可以写下了，但是body又得绕回到头部再写了。所以可见因为成员是变长的，所以无法通过预先规划使得尾部总是一个byte不多，一个byte不少。解决办法就是让这个绕回的点变成活动的。绕回的时候留下一个wrapAt的offset，让读的时候可以跟随着wrapAt的

指示跟着绕回。

```
---
1
---
0
---
A
--- <-- wrapAt
free slot 1
---
```

难点二：写入速度过快，覆盖了未读的区域

```
--- <-- nextWriteFrom
packet 1
--- <-- nextReadFrom
packet 2
---
packet 3
---
```

如果像上面这样的情况，nextWriteFrom接着写入就可能会覆盖nextReadFrom指向的区域。如果不移动nextReadFrom，那么下次读取的时候就会指向到一个错误的区域，比如位已经错位了的包，从而读取错误。问题的关键在于，怎么样判断nextReadFrom会被覆盖？咋一看这是一个很简单的问题

```
[writeFrom, writeTo)
```

如果readFrom落在这个区间里就可以认为会被影响到了。因为writeTo指向的byte其实不会被写入，所以很自然认为readFrom不会受影响。但是一旦我们允许了这样的行为就会出现非常复杂的情况，给定

```
nextReadFrom == nextWriteFom
```

这个情况下是有更多的字节可读，还是已经到队尾了？如果是认为到队尾了，那么意味着上面的情况我们没有移动readFrom指针导致一大片本来可以继续读的内存被跳过了。如果认为没有到队尾，那么队尾在哪里？队尾那就必须是前面的wrapAt的位置。那么就必须维护这样的一个范围 `[nextReadFrom, wrapAt)` 是包含有效的数据的。这是可行的，但是会非常麻烦。最简单的实现方式是要求 `[writeFrom, writeTo]` 整个闭区间的范围内都不包含读的指针，这样通过简单判断读写指针是不是相等就可以知道是不是无数据可读了。

难点三：怎么移读指针？

一旦写入速度过快要移读指针了，但是移读指针也是一个大难题。

```
--- <-- nextReadFrom
1
---
0
---
A
---
1
---
0
---
B
---
```

如果nextReadFrom如上所示，这个时候要求nextReadFrom往后移动4格，nextReadFrom不能是这样

```
---
1
---
0
---
A
---
1
--- <-- nextReadFrom
0
---
B
---
```

因为这样指向的是一个packet的中间是无效的。所以nextReadFrom往后退也要是整packet往后退。这里我们可以使用一个最简单的实现，一旦nextReadFrom被征用了，直接把nextReadFrom置为0，因为0总是一个合法的指向位置。而且0一次性清空的内存是足够的。缺点就是一次性置为0会丢掉大量未读的数据。

难点四：重读的需求

传统的ring buffer，读取和移动读取指针是一个操作。一旦读了，这篇区域就不再被保存了。但是经常我们需要读数据出来，做一些处理，当处理确定成功了再去移位置。否则下次重读（重启之后恢复了内存状态）的时候还是从上一次的地方重复读取。如何廉价地支持这样的可靠读取的需求？

简单的实现是保存两个读指针，一个是lastReadTo（已经提交了的），一个是nextReadFrom（未提交确认的）

```
--- <-- lastReadTo
1
---
0
---
A
--- <-- nextReadFrom
1
---
0
---
B
---
```

每次读取一次，就把上次读取到的位置（nextReadFrom）存入lastReadTo。这样做，有一个潜在的问题是写入可能会覆盖两个读指针。仔细思考一下，只需要考虑覆盖lastReadTo的问题，因为总是先覆盖到lastReadTo，再覆盖到nextReadFrom的。所以做覆盖的检查的时候以lastReadTo为准，一旦发生覆盖，则把lastReadTo和nextReadFrom都置为0，从头开始，丢掉未读的部分。

3月13日发布 更多

1 推荐 收藏

你可能感兴趣的文章

Go - Channel 原理 20 收藏，4.2k 浏览

笨办法学C 练习44：环形缓冲区 1 收藏，250 浏览

go 模拟 python 的 generator 612 浏览

讨论区

使用评论询问更多信息或提出修改意见，请不要在评论里回答问题

提交评论

评论支持部分 Markdown 语法： **bold** _italic_ [link](http://example.com) > 引用 `code` - 列表 。
同时，被你 @ 的用户也会收到通知

本文隶属于专栏

taowen

I write code

taowen
作者

关注专栏

相关收藏夹 换一组

Golang 问题收集与解决…
12 个条目 | 1 人关注

go语言
5 个条目 | 6 人关注

Go&Python
7 个条目 | 0 人关注

分享扩散：