垃圾回收（计算机科学） Go（编程语言）

Go 的垃圾回收机制在实践中有哪些需要注意的地方？

5 条评论 分享

4 个回答 按票数排序

达达，服务端程序员，主页： **1234n.com**

230

kurt lau、alexwei、Paul Meng 等人赞同

不想看长篇大论的，这里先给个结论，go的gc还不完善但也不算不靠谱，关键看怎么用，尽量不要创建大量对象，也尽量不要频繁创建对象，这个道理其实在所有带gc的编程语言也都通用。

想知道如何提前预防和解决问题的，请耐心看下去。

先介绍下我的情况，我们团队的项目《仙侠道》在7月15号第一次接受玩家测试，这个项目的服务端完全用Go语言开发的，游戏数据都放在内存中由go 管理。

在上线测试后我对程序做了很多调优工作，最初是稳定性优先，所以先解决的是内存泄漏问题，主要靠memprof来定位问题，接着是进一步提高性能，主要靠cpuprof和自己做的一些统计信息来定位问题。

调优性能的过程中我从cpuprof的结果发现发现gc的scanblock调用占用的cpu竟然有40%多，于是我开始搞各种对象重用和尽量避免不必要的对象创建，效果显著，CPU占用降到了10%多。

但我还是挺不甘心的，想继续优化看看。网上找资料时看到GOGCTRACE这个环境变量可以开启gc调试信息的打印，于是我就在内网测试服开启了，每当go执行gc时就会打印一行信息，内容是gc执行时间和回收前后的对象数量变化。

关注 445 人关注该问题

相关问题

计算机语言C语言目前用在什么地方？ 9 个回答

为什么计算机语言中的变量名都不能以数字开头呢？ 20 个回答

如何自创一门计算机语言？ 20 个回答

Excel 是用哪种计算机语言编写的？ 2 个回答

美国大学的计算机专业的入门编程语言是什么，也是c语言？ 7 个回答

我惊奇的发现一次gc要20多毫秒，我们服务器请求处理时间平均才33微秒，差了一个量级别呢。

于是我开始关心起gc执行时间这个数值，它到底是一个恒定值呢？还是更数据多少有关呢？

我带着疑问在外网玩家测试的服务器也开启了gc追踪，结果更让我冒冷汗了，gc执行时间竟然达到300多毫秒。go的gc是固定每两分钟执行一次，每次执行都是暂停整个程序的，300多毫秒应该足以导致可感受到的响应延迟。

所以缩短gc执行时间就变得非常必要。从哪里入手呢？首先，可以推断gc执行时间跟数据量是相关的，内网数据少外网数据多。其次，gc追踪信息把对象数量当成重点数据来输出，估计扫描是按对象扫描的，所以对象多扫描时间长，对象少扫描时间短。

于是我便开始着手降低对象数量，一开始我尝试用cgo来解决问题，由c申请和释放内存，这部分c创建的对象就不会被gc扫描了。

但是实践下来发现cgo会导致原有的内存数据操作出些诡异问题，例如一个对象明明初始化了，但还是读到非预期的数据。另外还会引起go运行时报申请内存死锁的错误，我反复读了go申请内存的代码，跟我直接用c的malloc完全都没关联，实在是很诡异。

我只好暂时放弃cgo的方案，另外想了个法子。一个玩家有很多数据，如果把非活跃玩家的数据序列化成一个字节数组，就等于把多个对象压缩成了一个，这样就可以大量减少对象数量。

我按这个思路用快速改了一版代码，放到外网实际测试，对象数量从几百万降至几十万，gc扫描时间降至二十几微秒。

效果不错，但是要用玩家数据时要反序列化，这个消耗太大，还需要再想办法。

于是我索性把内存数据都改为结构体和切片存放，之前用的是对象和单向链表，所以一条数据就会有一个对象对应，改为结构体和结构体切片，就等于把多个对象数据缩减下来。

结果如预期的一样，内存多消耗了一些，但是对象数量少了一个量级。

其实项目之初我就担心过这样的情况，那时候到处问人，对象多了会不会增加gc负担，导致gc时间过长，结果没得到答案。

现在我填过这个坑了，可以确定的说，会。大家就不要再往这个坑跳了。

如果go的gc聪明一点，把老对象和新对象区别处理，至少在我这个应用场景可以减少不必要的扫描，如果gc可以异步进行不暂停程序，我才不在乎那几百毫秒的执行时间呢。

但是也不能完全怪go不完善，如果一开始我早点知道用GOGCTRACE来观测，就可以比较早点发现问题从而比较根本的解决问题。但是既然用了，项目也上了，没办法大改，只能见招拆招了。

总结以下几点给打算用go开发项目或已经在用go开发项目的朋友：
1、尽早的用memprof、cpuprof、GCTRACE来观察程序。
2、关注请求处理时间，特别是开发新功能的时候，有助于发现设计上的问题。
3、尽量避免频繁创建对象(&abc{}、new(abc{})、make())，在频繁调用的地方可以做对象重用。
4、尽量不要用go管理大量对象，内存数据库可以完全用c实现好通过cgo来调用。

手机回复打字好累，先写到这里，后面再来补充案例的数据。

数据补充：

图1，7月22日的一次cpuprof观测，采样3000多次调用，数据显示scanblock吃了43.3%的cpu。

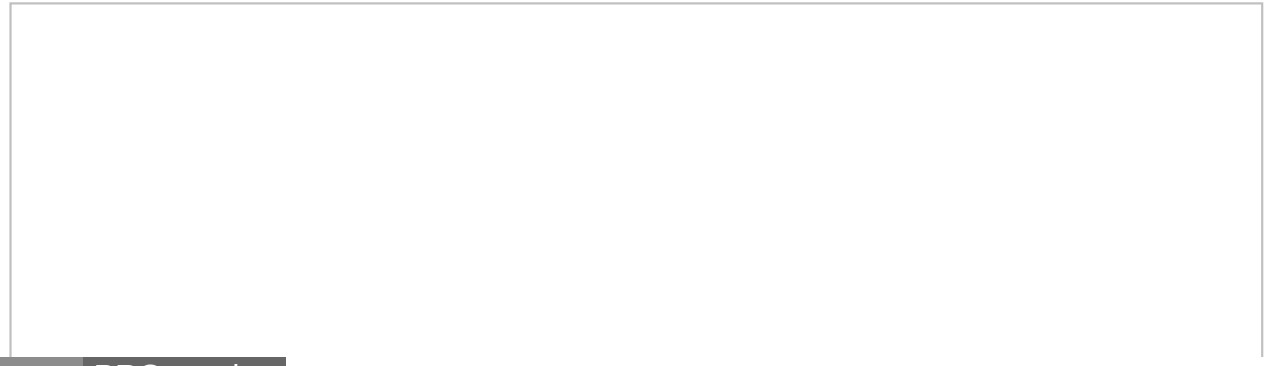

图2，7月23日，对修改后的程序做cpuprof，采样1万多次调用，数据显示cpu占用降至9.8%

数据1，外网服务器的第一次gc trace结果，数据显示gc执行时间有400多ms，回收后对象数量1659922个：

```
gc13(1): 308+92+1 ms , 156 -> 107 MB 3339834 -> 1659922 (12850245-11190323) objects
```

数据2，程序做了优化后的外网服务器gc trace结果，数据显示gc执行时间30多ms，回收后对象数量126097个：

```
gc14(6): 16+15+1 ms, 75 -> 37 MB 1409074 -> 126097 (10335326-10209229) objects, 45
```

示例1，数据结构的重构过程：

最初的数据结构类似这样

```
// 玩家数据表的集合
type tables struct {
        tableA *tableA
        tableB *tableB
        tableC *tableC
        // ..... 此处省略一大堆表
}

// 每个玩家只会有一条tableA记录
type tableA struct {
        fieldA int
        fieldB string
}

// 每个玩家有多条tableB记录
```

```
type tableB struct {
        xxoo int
        ooxx int
        next *tableB  // 指向下一条记录
}

// 每个玩家只有一条tableC记录
type tableC struct {
        id int
        value int64
}
```

最初的设计会导致每个玩家有一个tables对象，每个tables对象里面有一堆类似tableA和tableC这样的一对一的数据，也有一堆类似tableB这样的一对多的数据。

假设有1万个玩家，每个玩家都有一条tableA和一条tableC的数据，又各有10条tableB的数据，那么将总的产生1w (tables) + 1w (tableA) + 1w (tableC) + 10w (tableB)的对象。

而实际项目中，表数量会有大几十，一对多和一对一的表参半，对象数量随玩家数量的增长倍数显而易见。

为什么一开始这样设计？

1、因为有的表可能没有记录，用对象的形式可以用 == nil 来判断是否有记录
2、一对多的表可以动态增加和删除记录，所以设计成链表
3、省内存，没数据就是没数据，有数据才有对象

改造后的设计：

```
// 玩家数据表的集合
type tables struct {
        tableA tableA
        tableB []tableB
```

```
        tableC tableC
        // ..... 此处省略一大堆表
}

// 每个玩家只会有一条tableA记录
type tableA struct {
        _is_nil bool
        fieldA int
        fieldB string
}

// 每个玩家有多条tableB记录
type tableB struct {
        _is_nil bool
        xxoo int
        ooxx int
}

// 每个玩家只有一条tableC记录
type tableC struct {
        _is_nil bool
        id int
        value int64
}
```

一对一表用结构体，一对多表用slice，每个表都加一个_is_nil的字段，用来表示当前的数据是否是有用的数据。

这样修改的结果就是，一万个玩家，产生的对象总量是 1w (tables) + 1w ([]tablesB)，跟之前的设计差别很明显。

但是slice不会收缩，而结构体则是一开始就占了内存，所以修改后会导致内存消耗增大。

参考链接：

go的gc代码，scanblock等函数都在里面：
http://golang.org/src/pkg/runtime/mgc0.c

go的runtime包文档有对GOGCTRACE等关键的几个环境变量做说明：
golang.org/pkg/runtime/

如何使用cpuprof和memprof，请看《Profiling Go Programs》：
blog.golang.org/profili ...

我做的一些小试验代码，优化都是基于这些试验的数据的，可以参考下：
github.com/realint/labs ...

2013-09-08 29 条评论

9

天才

知乎用户、陈阳、Robin Yao 等人赞同

@达达 写得不错，实践经验，总结得很好。
尽量不要创建大量对象，也尽量不要频繁创建对象
我也说两点吧：
1.每次垃圾回收耗时。
2.垃圾回收频率。

显示全部

2013-09-09 5 条评论

0

祝川 ，游戏设计师

首先在go里面临时对象都应该创建在一个有终止节点的goruntine中，如果是缓存在服务器中的玩家数据结构肯定不需要回收
我在开发测试中数千个玩家做各种事情，还有怪物等等，我的方法是将所有对象全部打散和碎片化，每条消息每一条AI都是一个goruntine，这样临时对象在goruntine结束的时候就会被销毁掉，或者把这些对象放置在一个for中，在goruntine切换节点的时候也可以利用GC将其直接回收
还有在struct中我倾向于于用标准写法，保存指针并不会带来明显的GC压力（关键看你的GC节点在哪

2013-09-09 添加评论

0

陈文平 ，后台开发,熟悉**python, delphi,**对网络应用...

我觉得这里描述有点不太合适，go的struct就是对象吧，从“tableC* tableC”,改成“tableC tableC”，对象数是没有变的，只是由于直接引用，对象一同分配内存，gc不需要关心下面的对象释放问题；这里说的对象总量，应该是gc关心的独立对象

2013-09-09 2 条评论