高并发场景应对缓存击穿，哪种方式更适合你?

1.什么是缓存击穿

缓存击穿，是指一个key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开了一个洞。

2.缓存击穿和缓存雪崩、缓存穿透有什么区别?

缓存穿透，是指查询一个数据库一定不存在的数据。正常的使用缓存流程大致是，数据查询先进行缓存查询，如果key不存在或者key已经过期，再对数据库进行查询，并把查询到的对象，放进缓存。如果数据库查询对象为空，则不放进缓存。

缓存雪崩，是指在某一个时间段，缓存集中过期失效。

3.常用解决方案

3.1 热点数据永不过期

热点数据永不过期是一种比较简单粗暴的方法，同时采用定时任务定期去刷新同步数据即可。

3.2 请求DB互斥锁

该方案是在第一个请求去查询数据库的时候加一个互斥锁，这样其他的请求就会被阻塞住，直接锁被释放。第一个请求处理完成后也会刷新数据到缓存中，后面请求的线程进来发现缓存已经有数据了，就会直接走缓存，从而保护数据库。

该方案由于会阻塞其他的请求线程，此时系统吞吐量会下降。

3.3 更细粒度的互斥锁-singleflight

singleflight采用go语言开发的，该方案在也是采用同3.2中的互斥锁，只是锁的力度更细了点，其设计思路就是将一组相同的请求合并成一个请求，使用map进行存储，这样只会有一个请求到到达后端数据库，同时使用golang中sync.Waitgroup包进行同步，对所有的请求返回相同的结果。

3.3.1 如何使用?

来源：https://github.com/go-demo/singleflight-demo/blob/master/main.go

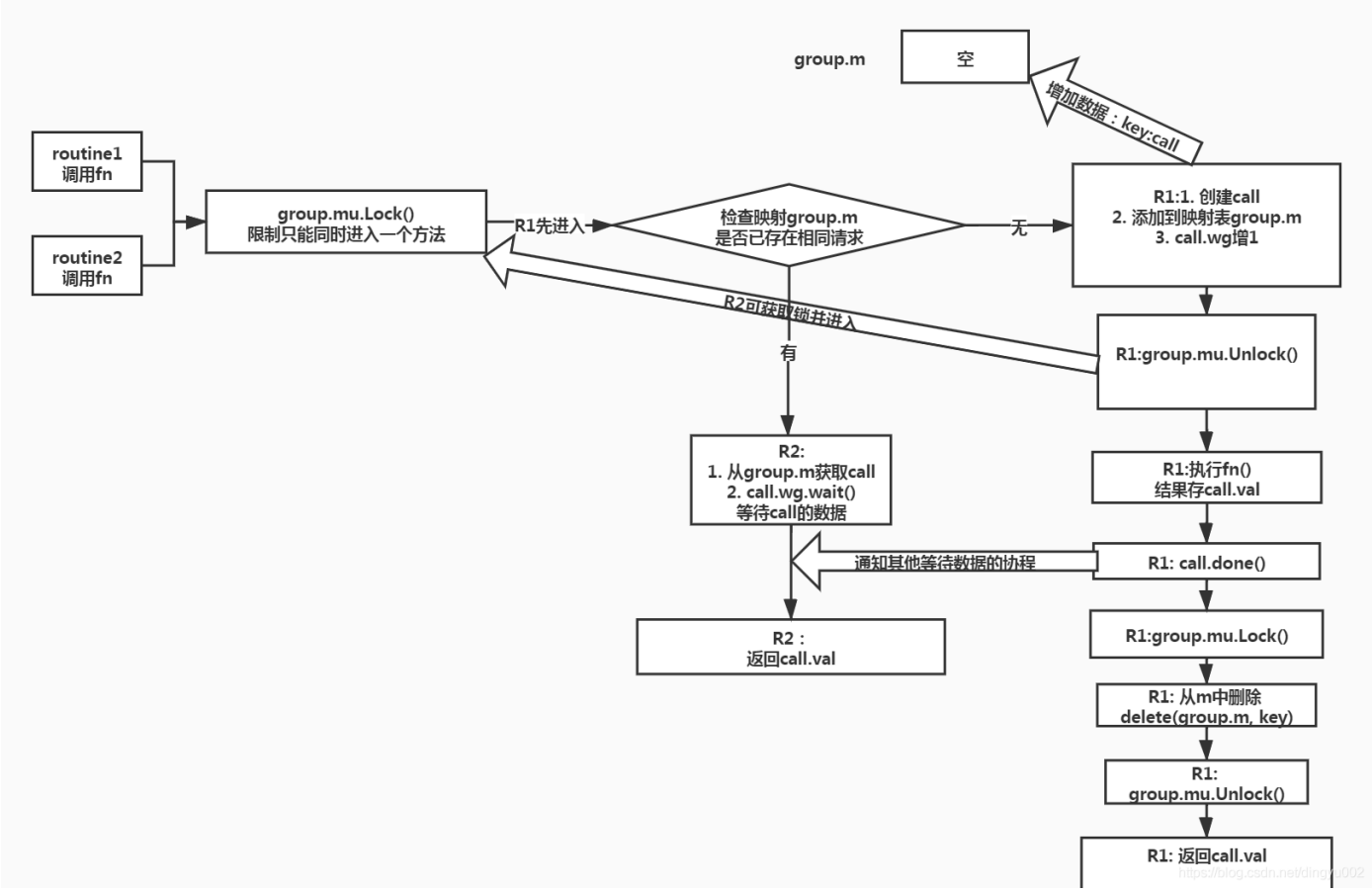
```
package main

import (
  "errors"
  "log"
  "sync"

  "golang.org/x/sync/singleflight"
```

```
)

var errorNotExist = errors.New("not exist")
var g singleflight.Group

func main() {
  var wg sync.WaitGroup
  wg.Add(10)

  //模拟10个并发
  for i := 0; i < 10; i++ {
    go func() {
      defer wg.Done()
      data, err := getData("key")
      if err != nil {
        log.Print(err)
        return
      }
      log.Println(data)
    }()
  }
  wg.Wait()
}

//获取数据
func getData(key string) (string, error) {
  data, err := getDataFromCache(key)
  if err == errorNotExist {
    //模拟从db中获取数据
    v, err, _ := g.Do(key, func() (interface{}, error) {
      return getDataFromDB(key)
      //set cache
    })
    if err != nil {
      log.Println(err)
      return "", err
    }

    //TOOD: set cache
    data = v.(string)
  } else if err != nil {
    return "", err
  }
  return data, nil
}

//模拟从cache中获取值，cache中无该值
func getDataFromCache(key string) (string, error) {
  return "", errorNotExist
```

```
}

//模拟从数据库中获取值
func getDataFromDB(key string) (string, error) {
  log.Printf("get %s from database", key)
  return "data", nil
}
```

3.3.2 singleflight主要逻辑流程

来源：



3.3.3 singleflight核心源码解析

1.主要数据结构

```
// call is an in-flight or completed singleflight.Do call
type call struct {
  wg sync.WaitGroup
  // 函数执行返回值，在wg.Done之前会写入，在wg.Done之后时只读的
  val interface{}
  // 存储返回的错误信息
  err error

  // 标记forget方法是否被调用
```

```
  forgotten bool

  // 统计调用次数
  dups  int
  // 返回执行结果的channel
  chans []chan<- Result
}

// Group represents a class of work and forms a namespace in
// which units of work can be executed with duplicate suppression.
type Group struct {
  mu sync.Mutex       // 互斥锁
  m  map[string]*call // map存储，保存请求key和调用的映射，使用懒加载，非创建时进行初始化
}

// Result holds the results of Do, so they can be passed
// on a channel.
type Result struct {
  // 存储返回值
  Val    interface{}
  // 存储返回的错误信息
  Err    error
  // 标识结果是否是共享结果
  Shared bool
}
```

2.主要方法

```
// Do executes and returns the results of the given function, making
// sure that only one execution is in-flight for a given key at a
// time. If a duplicate comes in, the duplicate caller waits for the
// original to complete and receives the same results.
// The return value shared indicates whether v was given to multiple callers.
func (g *Group) Do(key string, fn func() (interface{}, error)) (v interface{}, err
error, shared bool) {
  g.mu.Lock()
  if g.m == nil {
    // 懒加载创建map
    g.m = make(map[string]*call)
  }
  // 判断是否有相同请求
  if c, ok := g.m[key]; ok {
    // 请求次数+1
    c.dups++
    // 解锁等到执行结果
    g.mu.Unlock()
    c.wg.Wait()

    if e, ok := c.err.(*panicError); ok {
```

```
      panic(e)
    } else if c.err == errGoexit {
      runtime.Goexit()
    }
    return c.val, c.err, true
  }
  // 请求不存在
  c := new(call)
  // 设置相同请求时，只有一个请求处理，其他请求都等待执行结果
  c.wg.Add(1)
  g.m[key] = c
  g.mu.Unlock()

  g.doCall(c, key, fn)
  return c.val, c.err, c.dups > 0
}

// 异步返回
func (g *Group) DoChan(key string, fn func() (interface{}, error)) <-chan Result {
  ch := make(chan Result, 1)
  g.mu.Lock()
  if g.m == nil {
    g.m = make(map[string]*call)
  }
  if c, ok := g.m[key]; ok {
    c.dups++
    c.chans = append(c.chans, ch)
    g.mu.Unlock()
    return ch
  }
  c := &call{chans: []chan<- Result{ch}}
  c.wg.Add(1)
  g.m[key] = c
  g.mu.Unlock()
  // 执行请求处理
  go g.doCall(c, key, fn)

  return ch
}

// doCall handles the single call for a key.
func (g *Group) doCall(c *call, key string, fn func() (interface{}, error)) {
  // 标识是否正常返回
  normalReturn := false
  // 标识是否发生panic
  recovered := false

  // use double-defer to distinguish panic from runtime.Goexit,
  // more details see https://golang.org/cl/134395
```

```
defer func() {
  // 判断是否runtime导致直接退出了
  if !normalReturn && !recovered {
    c.err = errGoexit
  }

  c.wg.Done()
  g.mu.Lock()
  defer g.mu.Unlock()
  // 防止重复删除key
  if !c.forgotten {
    delete(g.m, key)
  }
  // 检测是否发生了panic错误
  if e, ok := c.err.(*panicError); ok {
    // In order to prevent the waiting channels from being blocked forever,
    // needs to ensure that this panic cannot be recovered.
    if len(c.chans) > 0 {
      go panic(e)
      select {} // 保住这个goroutine，这样可以将panic写入crash dump
    } else {
      panic(e)
    }
  } else if c.err == errGoexit {
    // runtime错误不需要做任何处理，已经退出了
  } else {
    // 正常返回，将结果写入channel中
    for _, ch := range c.chans {
      ch <- Result{c.val, c.err, c.dups > 0}
    }
  }
}()

func() {
  defer func() {
    if !normalReturn {
      // 如果发生了panic，则进行recover，并把错误信息返回上层
      if r := recover(); r != nil {
        c.err = newPanicError(r)
      }
    }
  }()
  // 执行函数
  c.val, c.err = fn()
  // fn没有发生panic
  normalReturn = true
}()
// 判断执行函数是否发生了panic
if !normalReturn {
```

```
    recovered = true
  }
}

// Forget tells the singleflight to forget about a key.  Future calls
// to Do for this key will call the function rather than waiting for
// an earlier call to complete.
func (g *Group) Forget(key string) {
  g.mu.Lock()
  if c, ok := g.m[key]; ok {
    c.forgotten = true
  }
  delete(g.m, key)
  g.mu.Unlock()
}
```

备注说明：这里为啥要区分panic和runtime错误，如果不区分的话，调用出现panic，但是锁没有释放，这样就会导致使用相同key的所有后续调用都出现了死锁

4.参考

[1]https://github.com/golang/sync/blob/master/singleflight/singleflight.go

[2]https://silenceper.com/blog/202003/singleflight/