Go 进阶训练营答疑
第3课
Go 语言实践 - concurrency

大明

目录

- 并发代码演示
- 知识梳理
- 面试题

代码演示——奇奇怪怪的并发代码

- sync.XXX 在方法之间传递
- sync.Once 用于构造单例和实例初始化
- double-check
- LoadStore 与资源
- 单向幂等并发
- 锁保护资源

代码演示——sync.XXX 传递

```go
// NeedOnce 用于测试 sync.Once 在方法之间传递
func NeedOnce(once sync.Once) {
    once.Do(func() {
        fmt.Println( a...: "calling NeedOnce")
    })
}

// NeedOncePrt 用于测试 sync.Once 的指针在方法之间传递
func NeedOncePrt(once *sync.Once) {
    once.Do(func() {
        fmt.Println( a...: "calling NeedOncePtr")
    })
}
```

```go
func TestNeedOnce(t *testing.T) {
    var once sync.Once
    // 我们调用两次，希望只输出一次
    // 然而结果是输出了两次
    NeedOnce(once)
    NeedOnce(once)

    // 在使用指针的时候，只输出了一次
    NeedOncePrt(&once)
    NeedOncePrt(&once)
}
```

不要在参数里面传递sync.XXX

代码演示——sync.Once 写单例

```go
// 可以定义一个接口
type Single interface {
    Single()
}

type singleton struct {

}

func (s *singleton) Single() {
    fmt.Println(a...: "I am single")
}

var instance *singleton
var instanceOnce sync.Once
// GetSingleInstance 返回接口
func GetSingleInstance() Single {
    instanceOnce.Do(func() {
        instance = &singleton{}
    })
    return instance
}
```

代码演示——double check

```go
// LoadOrStore loaded 代表是返回老的对象，还是返回了新的对象
func (s *SafeMap) LoadOrStore(key string, newVale interface{}) (val interface{}, loaded bool) {
    s.mutex.RLock()
    val, ok := s.m[key]
    s.mutex.RUnlock()
    if ok {
        return val, loaded: true
    }
    s.mutex.Lock()
    defer s.mutex.Unlock()
    s.m[key] = newVale
    return newVale, loaded: false
}
```

代码演示——double check

```go
// LoadOrStore loaded 代表是返回老的对象，还是返回了新的对象
func (s *SafeMap) LoadOrStore(key string, newVale interface{}) (val interface{}, loaded bool) {
    s.mutex.RLock()
    val, ok := s.m[key]
    s.mutex.RUnlock()
    if ok {
        return val, loaded: true
    }
    s.mutex.Lock()
    defer s.mutex.Unlock()
    s.m[key] = newVale
    return newVale, loaded: false
}
```

在判断 ok 的时候，就有别的 goroutine 插进去了同样 key 的值

代码演示——double check

```go
// LoadOrStore loaded 代表是返回老的对象，还是返回了新的对象
func (s *SafeMap) LoadOrStore(key string, newVale interface{}) (val interface{}, loaded bool) {
    s.mutex.RLock()
    val, ok := s.m[key]
    s.mutex.RUnlock()
    if ok {
        return val, loaded: true
    }
    s.mutex.Lock()
    defer s.mutex.Unlock()
    val, ok = s.m[key]
    if ok {   第二次检查
        return val, loaded: true
    }
    s.m[key] = newVale
    return newVale, loaded: false
}
```

代码演示—— check - and - do something 的两种写法

```go
func (s *SafeMap)CheckAndDoSomething() {
    s.mutex.Lock()
    // check and do something
    s.mutex.Unlock()
}
```

```go
func (s *SafeMap) CheckAndDoSomething1() {
    s.mutex.RLock()
    // check 第一次检查
    s.mutex.RUnlock()

    s.mutex.Lock()
    // check and doSomething
    defer s.mutex.Unlock()
}
```

代码演示—— LoadAndStore 释放资源

```go
func TestSafeMap_LoadOrStore(t *testing.T) {
    m := &SafeMap{
        m: map[string]interface{}{},
    }

    for i := 0; i < 10; i++ {
        go func() {
            con := &connection{}
            nc, _ := m.LoadOrStore( key: "hello", con)
            _ = nc.(*connection).send()
        }()
    }
}
```

后边的覆盖前面的

代码演示—— LoadAndStore 释放资源

```go
func TestSafeMap_LoadOrStore(t *testing.T) {
    m := &SafeMap{
        m: map[string]interface{}{},
    }

    for i := 0; i < 10; i++ {
        go func() {
            con := &connection{}
            nc, loaded := m.LoadOrStore( key: "hello", con)
            if loaded {
                _ = con.Close()
            }
            _ = nc.(*connection).send()
        }()
    }
}
```

其实我这里竞争失败了，别的 goroutine 已经弄好了

代码演示—— LoadAndStore 延迟创建

```go
type valProvider func() interface{}

func (s *SafeMap) LoadOrStoreHeavy(key string, p valProvider) (val interface{}, loaded bool) {
    s.mutex.RLock()
    val, ok := s.m[key]
    s.mutex.RUnlock()
    if ok {
        return val, loaded: true
    }
    s.mutex.Lock()
    defer s.mutex.Unlock()
    val, ok = s.m[key]
    if ok {
        return val, loaded: true
    }
    newVale := p()
    s.m[key] = newVale
    return newVale, loaded: false
}
```

创建某个东西非常消耗资源的时候，用这个

代码演示—— LoadAndStore 延迟创建

```go
func TestSafeMap_LoadOrStoreHeavy(t *testing.T) {
    m := &SafeMap{
        m: map[string]interface{}{},
    }

    for i := 0; i < 10; i++ {
        go func() {
            nc, _ := m.LoadOrStoreHeavy( key: "hello", func() interface{} {
                return &connection{}
            })
            _ = nc.(*connection).send()
        }()
    }
}
```

代码演示—— Limiter

```go
type Limiter struct {
    // 当前处理请求的上限
    limit int
    // 处理请求逻辑
    handler func(req interface{}) interface{}
}

// Reject bool 返回值表示究竟有没有执行
func (l Limiter) Reject(req interface{}) (interface{}, bool) {

}
```

代码演示—— Limiter v1版本

```go
// v1
type Limiter struct {
    // 当前处理请求的上限
    limit int
    // 处理请求逻辑
    handler func(req interface{}) interface{}

    mutex sync.Mutex
}

// Reject bool 返回值表示究竟有没有执行
func (l *Limiter) Reject(req interface{}) (interface{}, bool) {
    l.mutex.Lock()
    defer l.mutex.Unlock()
    res := l.handler(req)
    return res, true
}
```

串行，一次一个

代码演示

```go
type Limiter struct {
    // 当前处理请求的上限
    limit int
    // 处理请求逻辑
    handler func(req interface{}) interface{}

    mutex sync.Mutex
    cnt int
}

// Reject bool 返回值表示究竟有没有执行
func (l *Limiter) Reject(req interface{}) (interface{}, bool) {
    l.mutex.Lock()
    if l.cnt < l.limit {
        l.cnt ++
        l.mutex.Unlock()   释放锁，其它goroutine 可以得到处理
        res := l.handler(req)
        l.mutex.Lock()
        defer l.mutex.Unlock()
        l.cnt --     处理完，返回回去之前，要再减回去
        return res, true
    }
    l.mutex.Unlock()
    return nil, false
}
```

代码演示—— Limiter v3版本

```go
// Reject bool 返回值表示究竟有没有执行
func (l *Limiter) Reject(req interface{}) (interface{}, bool) {
    l.mutex.RLock()
    if l.cnt > l.limit {
        l.mutex.RUnlock()
        return nil, false
    }
    l.mutex.Lock()
    if l.cnt > l.limit {
        l.mutex.Unlock()
        return nil, false
    }

    l.cnt ++
    l.mutex.Unlock()
    res := l.handler(req)
    l.mutex.Lock()
    defer l.mutex.Unlock()
    l.cnt --
    return res, true
}
```

如果经常命中这个分支，那么性能要好一点

代码演示—— Limiter v4版本

```go
type Limiter struct {
    // 当前处理请求的上限
    limit int32
    // 处理请求逻辑
    handler func(req interface{}) interface{}

    cnt int32
}

// Reject bool 返回值表示究竟有没有执行
func (l *Limiter) Reject(req interface{}) (interface{}, bool) {
    cnt := atomic.LoadInt32(&l.cnt)
    if cnt >= l.limit {
        return nil, false
    }

    atomic.AddInt32(&l.cnt, delta: 1)
    defer atomic.AddInt32(&l.cnt, delta: -1)
    res := l.handler(req)
    return res, true
}
```

代码演示—— Limiter v4版本

```go
// Reject bool 返回值表示究竟有没有执行
func (l *Limiter) Reject(req interface{}) (interface{}, bool) {
    cnt := atomic.LoadInt32(&l.cnt)
    if cnt >= l.limit {
        return nil, false
    }

    问题出在这里
    atomic.AddInt32(&l.cnt, delta: 1)
    defer atomic.AddInt32(&l.cnt, delta: -1)
    res := l.handler(req)
    return res, true
}
```

代码演示—— Limiter v4版本

```go
// Reject bool 返回值表示究竟有没有执行
func (l *Limiter) Reject(req interface{}) (interface{}, bool) {
    // 假如此时 limit = 10. cnt = 9
    // 来了两个 goroutine
    cnt := atomic.LoadInt32(&l.cnt)

    // 毫无疑问，两个goroutine 都能通过这个检测
    if cnt >= l.limit {
        return nil, false
    }

    // 两个 goroutine 都加1，很好，limit = 11了，超过了
    atomic.AddInt32(&l.cnt, delta: 1)
    defer atomic.AddInt32(&l.cnt, delta: -1)
    res := l.handler(req)
    return res, true
}
```

代码演示—— Limiter v4版本

```go
// Reject bool 返回值表示究竟有没有执行
func (l *Limiter) Reject(req interface{}) (interface{}, bool) {
    cnt := atomic.LoadInt32(&l.cnt)
    if cnt >= l.limit {
        return nil, false
    }

    cnt = atomic.AddInt32(&l.cnt, delta: 1)
    defer atomic.AddInt32(&l.cnt, delta: -1)
    if cnt >= l.limit {
        return nil, false
    }
    res := l.handler(req)
    return res, true
}
```

代码演示—— Limiter v4版本

```go
// Reject bool 返回值表示究竟有没有执行
func (l *Limiter) Reject(req interface{}) (interface{}, bool) {

    // 进来，二话不说，我就直接先给你加了，就是说，我分配了一个位置给你
    cnt := atomic.AddInt32(&l.cnt, delta: 1)
    defer atomic.AddInt32(&l.cnt, delta: -1)
    // 我发现我超出了上限
    if cnt >= l.limit {
        return nil, false
    }
    res := l.handler(req)
    return res, true
}
```

代码演示—— 单向幂等修改

```go
type filter struct {
    // 处理请求逻辑
    handler func(req interface{}) interface{}
    // 0 代表不拒绝，不为0 代表拒绝
    reject int32
}

func (f *filter) Handle(req interface{}) (interface{}, bool) {

}

func (f *filter) RejectNewRequest() {
}
```

我们在关闭服务器的时候，让filter 拒绝所有的新请求

代码演示—— 单向幂等修改 标准写法

```go
type filter struct {
    // 处理请求逻辑
    handler func(req interface{}) interface{}
    // 0 代表不拒绝，不为0 代表拒绝
    reject int32
}

func (f *filter) Handle(req interface{}) (interface{}, bool) {
    if atomic.LoadInt32(&f.reject) > 0 {
        return nil, false
    }
    return f.handler(req), true
}

func (f *filter) RejectNewRequest() {
    atomic.StoreInt32(&f.reject, val: 1)
}
```

每一个请求都检验

主函数收到关闭信号，调用这个方法

代码演示—— 单向幂等修改 略奇诡写法

```go
type filter struct {
    // 处理请求逻辑
    handler func(req interface{}) interface{}
    reject int
}

func (f *filter) Handle(req interface{}) (interface{}, bool) {
    if f.reject > 0 {
        return nil, false
    }
    return f.handler(req), true
}

func (f *filter) RejectNewRequest() {
    f.reject = 1
}
```

只要能接受可见性的短暂延迟，就没啥问题

代码演示—— 锁保护资源

```
// PublicResource 你永远不知道你的用户拿了它会干啥
// 他即便不用 PublicResourceLock 你也毫无办法
var PublicResource interface{}
var PublicResourceLock sync.Mutex
```

代码演示—— 锁保护资源

```
// safeResource 很棒，所有的期望对资源的操作都只能通过定义在上 safeResource 上的方法来进行
type safeResource struct {
    resource interface{}
    lock sync.Mutex
}

func (s *safeResource) DoSomethingToResource()  {
    s.lock.Lock()
    defer s.lock.Unlock()
}
```

代码演示—— 锁保护资源

```go
// safeResource 很棒，所有的期望对资源的操作都只能通过定义在上 safeResource 上的方法来进行
type safeResource struct {
    resource interface{}
    lock sync.Mutex
}

func (s *safeResource) DoSomethingToResource()  {
    s.lock.Lock()
    defer s.lock.Unlock()
}
```

代码演示—— 锁保护资源

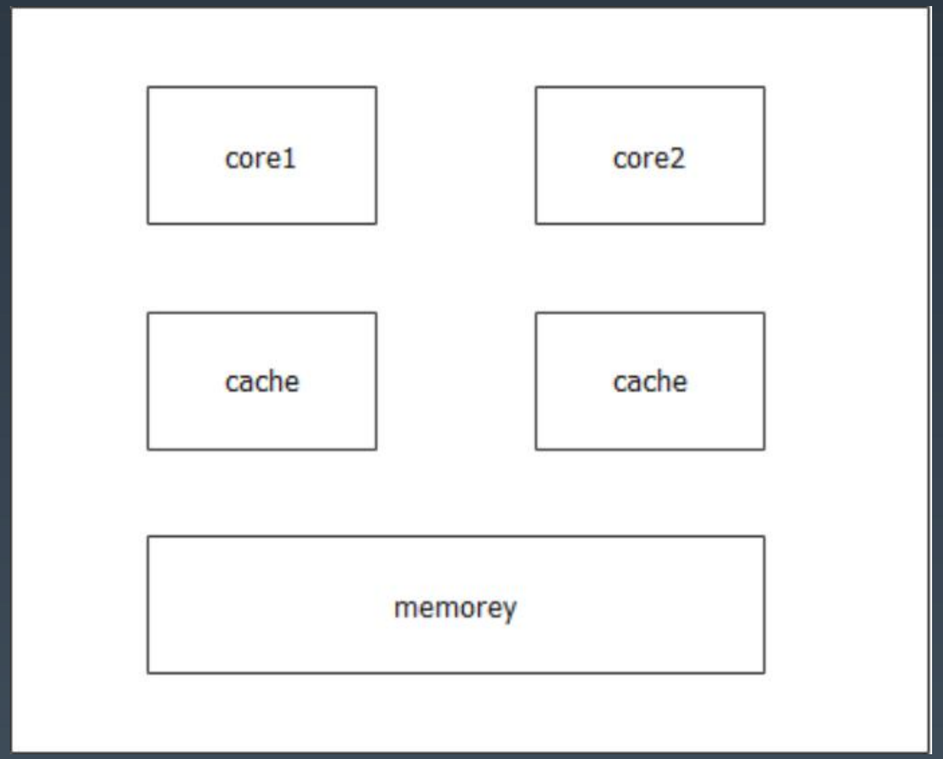
```go
// Registry 没有用锁，并不安全
type Registry struct {
    resources map[string]interface{}
}

func (r *Registry) Register(name string, resource interface{})  {
    r.resources[name] = resource
}

func (r *Registry) Get(name string) (interface{}, error) {

}
```

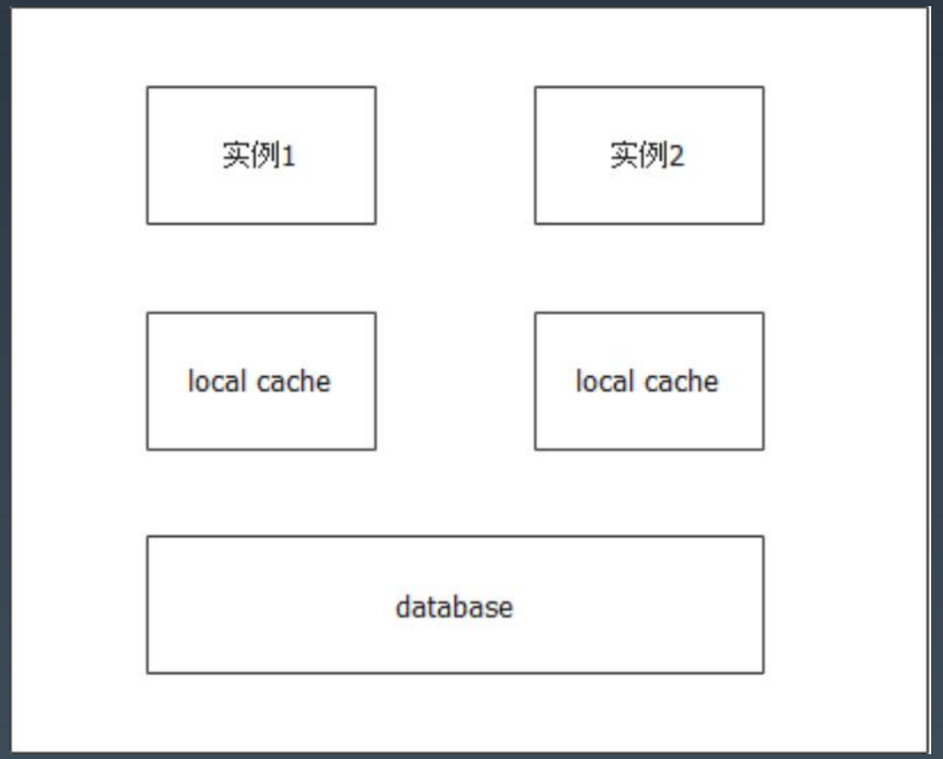
代码演示—— 锁保护资源

```go
// Registry 没有用锁，并不安全
type Registry struct {
    resources map[string]interface{}
}

func (r *Registry) Register(name string, resource interface{})  {
    r.resources[name] = resource
}

func (r *Registry) Get(name string) (interface{}, error) {

}
```

```go
func TestRegistry_Get(t *testing.T) {


    registry := &Registry{
        resources: map[string]interface{}{},
    }

    // 要求在应用启动前全部注册好
    registry.Register( name: "a",  resource: "a-r")
    registry.Register( name: "b",  resource: "b-r")
    registry.Register( name: "c",  resource: "c-r")
    registry.Register( name: "d",  resource: "d-r")
    registry.Register( name: "e",  resource: "e-r")
    runApp()
}

func runApp()  {
    // 这里对资源的并发操作
}
```

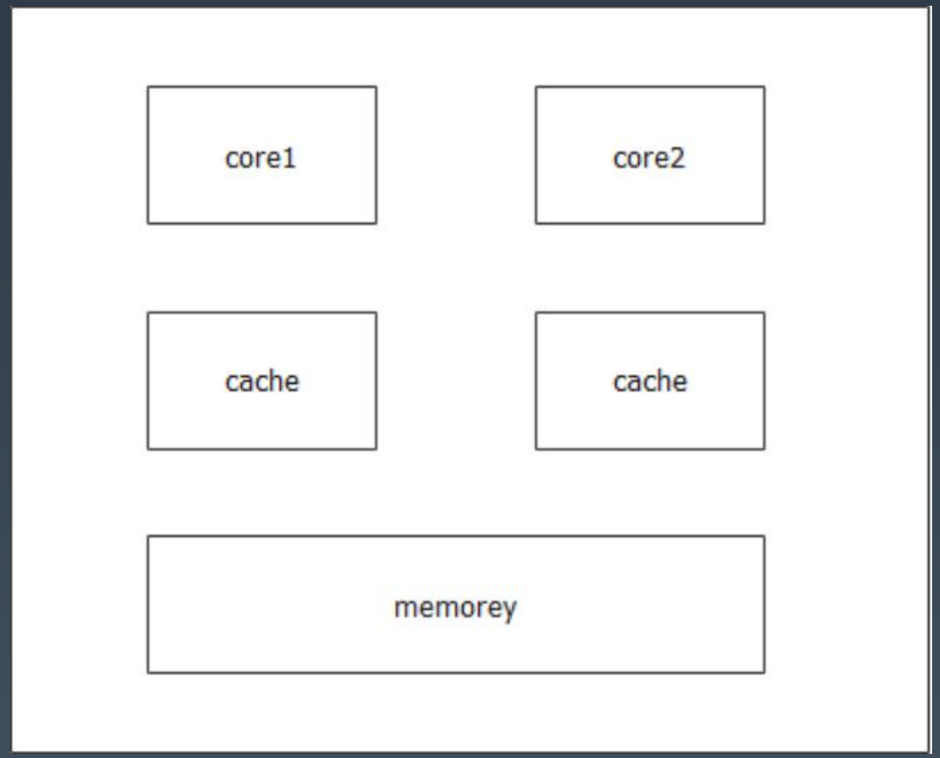
知识梳理—— 可见性

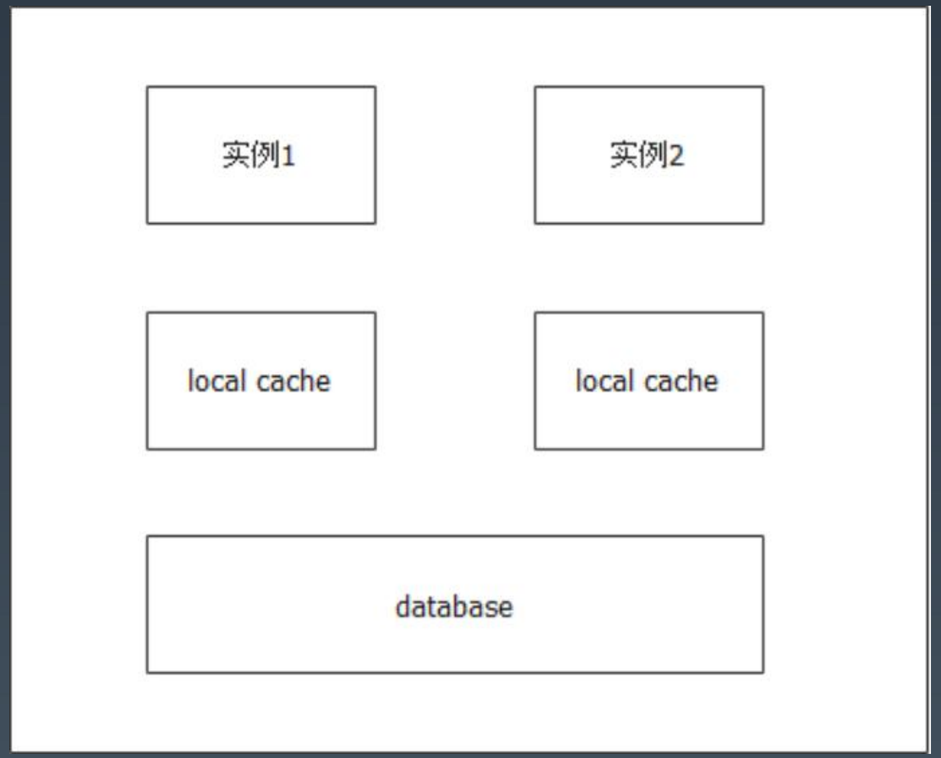




不能说是毫无关系，只能说是一模一样了

知识梳理—— 可见性





1. 如果只是修改本地缓存，别的实例肯定感知不到
2. 如果修改了本地缓存，还写回去了数据库，别的实例也不一定能看到，因为它们自身缓存还没过期
3. 写回去数据库之后，还要通知别的实例让缓存过期，重新加载

知识梳理—— 加锁过程

https://segmentfault.com/a/1190000039855697

面试题

- 进程、线程和协程的不同？
- 为什么要引入协程？
- goroutine 泄露的典型场景
- 怎么避免`goroutine`泄露
- mutex 加锁过程

面试题—— 进程、线程和协程的不同

进程、线程和协程的不同?

分析：这种很明显的是一种逐步演化的路径。也就是`进程-线程-协程`总体而言可以看做是一种演化路线。

审视这个演化，就会发现它是朝着更轻量的方向演进的。于是结合需求和计算机的发展，就会发现：业务越来越复杂，计算机越来越强大，但是我们需要的确是越来越细粒度的资源分配。

进程演进到线程，共享了内存，但是线程可以被CPU单独调度；线程到协程，内存使用量更少了，多个协程绑定到一个线程，相当于大家平分了这个线程的 CPU。

以上这一段吹牛，如果面试记得，可以跟面试官聊。不记得就算了。

答案：（首先是标准答案）
1. 进程是资源分配的最小单位，而线程是 CPU 调度的单位，一个进程可以有多个线程。因为同一个进程内的线程共享了堆内存，所以在经常会引起并发编程问题；
2. 协程比线程更轻量级。线程的创建和销毁、调度还需要陷入到内核中，而协程可以认为完全是依赖于用户空间创建、销毁和调度的。同时协程相比线程，占据的资源更加小。

（其次，开始引申）目前来说，很多语言都开始尝试支持协程，主要是因为现在的很多业务都是短平快，或者是 IO 密集的，相比之下，线程也过于重了。

最有名的就是`goroutine`（实际上也不是最有名吧，不过都面试 go 了，就说最有名了），此外还有`kotlin`协程，`python`的协程。

面试题—— 为什么引入协程

为什么要引入协程?

分析：`goroutine` 的引入，本质上还是为了规避一个问题：我又想有并发，但是我又不想陷入到内核里面去。于是就有了这个 `goroutine` 的东西。

该面试题的核心，就是协程“轻量”。这种轻量体现在两方面：

1. 所需要的资源更少
2. 创建销毁和调度更轻量，并不需要陷入内核

其实个人看法是大规模应用协程随着需求和计算机性能发展而来的大趋势。需求上，要求我们有更高的并发。而大多数并发执行的任务都是短平快，单独一个线程划不来，即便是使用线程池，也会带来频繁切换上下文的问题；而计算机性能提高，使得我们可以将整个计算机资源划分为更细粒度进行分配，两者叠加就是协程的出现。这一段算是个人体会，面试慎用。

答案：很简单，因为我们需要一个更轻量的东西来取代线程。（开始聊自己理解的起源）当前绝大多数系统处理的任务都是非常短平快的，或者是 IO 密集这种频繁触发上下文切换的任务。这导致我们如果使用线程，就不得不面临线程频繁切换，陷入内核的问题—这是一个极大的开销。

因此我们需要一个比线程更加轻量的东西。这个东西要具备两个特征：

1. 占有的资源小—我们都是小任务，不需要那么多资源；
2. 创建销毁和调度消耗少—小任务，还时常阻塞，所以调度一定要快要轻；

结合在一起就是`goroutine`了。

面试题—— goroutine 泄露的典型场景

```
5
6  ### `goroutine`泄露的典型场景
7
8  分析：这个问题答案来自煎鱼大佬的文章[跟读者聊 Goroutine 泄露的 N 种方法，
   真刺激！](https://blog.csdn.net/EDDYCJY/article/details/115535237)
9
0  PS：煎鱼大佬的文章都很浅显易懂，即便是难题也能说得很容易理解，大家可以多读
   读，他有一个公众号《脑子进煎鱼了》，可以关注。
1
2  记住，至少背下来里面的一个例子，防着面试官让你手写一个`goroutine`泄露的例
   子。
3
4  然后面试官让你看一段代码，如果有锁，就要怀疑死锁；如果有`channel`就要怀疑
   `goroutine`泄露；
5
6  其实`channel`的代码坑极多，在`channel`里面进一步讨论。
7
8  答：有：
9  - `channel`发送不接收
0  - `channel`接收不发送
1  - `nil channel`
2  - 慢等待
3  - 互斥锁忘记解锁
4  - 同步锁使用不当
5
6  （同样聊排查作为亮点）排查主要用`runtime.NumGoroutine`或者`pprof`工具。
   `pprof`会返回所有带有堆栈跟踪的`goroutine`列表。
7
```

https://blog.csdn.net/EDDYCJY/article/details/115535237

面试题—— 怎么避免`goroutine`泄露

怎么避免`goroutine`泄露?

分析：如果你不知道`goroutine`什么时候会结束，就不要使用`goroutine`。这是核心原则。讲完这个原则之后，可以讲一些如何做到“知道goroutine”何时结束。

大体上就是两个方向：
1. 超时控制
2. 信号通知。这一步基本上就是利用`channel`

这两个方向，基本上都离不开要使用`select`来配合。

然后刷两点可以回答”如何发现`goroutine`泄露“。

答案：避免`goroutine`泄露的核心原则是"Never start a goroutine without knowning when it will stop"（用英文会显得你比较专业，记不住可以替换为对应的中文）。

归根结底，就是要有办法控制住结束掉自己开启的`goroutine`。大体上有两类做法：
1. 超时控制，主要利用`context.Timeout`的特性；
2. 主动发信号给`goroutine`关闭。一般是要利用到`channel`的特性；

两种做法基本都要配合`select`特性来。要么是业务正常结束，退出`goroutine`，要么是超时，或者收到关闭信号，异常退出。

（开始讨论如何发现`goroutine`泄露）如果`goroutine`都不是自己开启的，那肯定是没得办法了。只能通过`runtime.NumGoroutine()`方法监控`goroutine`的数量来判断有没有泄露。如果`goroutine`一直在上涨，而且数量也很多，说明泄露很严重。而如果只是轻微泄露，比如说一万个`goroutine`里面泄露了十个，是很难看出来的。

（后面可以进一步引申，跳到**`goroutine`泄露的典型场景**）
（这个问题也可以针对`goroutine`泄露的典型场景来回答，比如说小心使用`channel`，正确使用`mutex`，防止业务一直阻塞等，不过略等于啥也没说）

面试题—— mutex 加锁

`mutex`是如何加锁的

分析：也是考察`mutex`的实现原理。基本上就是围绕`自旋-FIFO`来说的。简单理解就是，`mutex`先尝试自旋，自旋不行就所有`goroutine`步入`FIFO`，先到先得。

加锁的这个步骤，讲得非常详细。

答案：`mutex`加锁大概分成两种模式：
1. 在正常模式下，`goroutine`通过自旋来获得锁；
2. 但是如果存在一个`goroutine`等待锁超过了`1ms`，那么`mutex`就会进入饥饿模式，在饥饿模式下，会遵循`FIFO`原则，将锁交给下一个`goroutine`。也就是从等待队列里面唤醒第一个等待者；

（讨论一下公平性的问题）所以从严格意义上来说，它并不是公平锁，因为在正常状态下，一个新的请求锁的`goroutine`和等待的`goroutine`一起竞争锁。而严格意义的公平应该是永远遵循 `FIFO`。

类似问题
- 什么是饥饿状态
- 是不是公平锁

代码仓库

https://github.com/flycash/geekbang-go-camp