# 项目：Go-Checker - 并发网站状态检查工具

这是一个从零开始的、循序渐进的实践方案，旨在帮助你通过实战掌握 Go 语言的并发编程、标准库使用以及项目测试。

### **阶段一：万事开头难，先来个单线程版 (Baseline)**

**学习目标：**

* 掌握 flag 包，实现命令行参数解析。
* 掌握 os 和 bufio 包，实现文件读取。
* 掌握 net/http 包，发起网络请求。
* 掌握 time 包，进行基本的时间计算。
* 熟悉 Go 的基本结构：struct 定义、函数编写、main 函数逻辑。

**实践步骤：**

1. **项目初始化**：
   * 创建一个项目目录，如 go-checker。
   * 在目录下创建 main.go 和一个名为 urls.txt 的文本文件。
   * 在 urls.txt 中放入一些网址，每行一个，例如：  
     [https://www.google.com](https://www.google.com)  
     [https://www.baidu.com](https://www.baidu.com)  
     [https://github.com](https://github.com)  
     [https://invalid-url-that-does-not-exist.com](https://invalid-url-that-does-not-exist.com)
2. **编写 main.go 的基础结构**：
   * **定义结果结构体**：我们需要一个结构体来存放每个URL的检查结果。  
     package main  
       
     import (  
      "fmt"  
      "time"  
     )  
       
     type CheckResult struct {  
      URL string  
      StatusCode int  
      Latency time.Duration  
      Error error  
     }
   * **编写核心检查函数**：这个函数负责检查单个URL。  
     import "net/http"  
       
     func checkURL(url string) CheckResult {  
      client := http.Client{  
      Timeout: 5 \* time.Second, // 设置一个5秒的超时，非常重要！  
      }  
      start := time.Now()  
      resp, err := client.Get(url)  
      latency := time.Since(start)  
       
      if err != nil {  
      return CheckResult{URL: url, Error: err}  
      }  
      defer resp.Body.Close()  
       
      return CheckResult{URL: url, StatusCode: resp.StatusCode, Latency: latency}  
     }
   * **编写主函数逻辑（单线程版）**：读取文件，遍历URL列表，逐个调用 checkURL 并打印结果。  
     import (  
      "bufio"  
      "flag"  
      "fmt"  
      "log"  
      "os"  
     )  
       
     func main() {  
      // 1. 使用 flag 包接收命令行传入的文件名  
      filePath := flag.String("file", "urls.txt", "包含URL列表的文件路径")  
      flag.Parse()  
       
      // 2. 读取并解析文件  
      file, err := os.Open(\*filePath)  
      if err != nil {  
      log.Fatalf("无法打开文件: %v", err)  
      }  
      defer file.Close()  
       
      scanner := bufio.NewScanner(file)  
      var urls []string  
      for scanner.Scan() {  
      urls = append(urls, scanner.Text())  
      }  
      if err := scanner.Err(); err != nil {  
      log.Fatalf("读取文件时发生错误: %v", err)  
      }  
       
      // 3. 顺序执行检查  
      fmt.Println("开始检查...")  
      for \_, url := range urls {  
      result := checkURL(url)  
      if result.Error != nil {  
      fmt.Printf("URL: %s, 状态: 失败, 错误: %v\n", result.URL, result.Error)  
      } else {  
      fmt.Printf("URL: %s, 状态码: %d, 延迟: %v\n", result.URL, result.StatusCode, result.Latency)  
      }  
      }  
      fmt.Println("检查完成！")  
     }

运行与验证：

在终端运行 go run main.go 或 go run main.go -file=urls.txt。你会看到程序按顺序输出每个网站的检查结果。记录一下总耗时，这将是我们后续优化的参照基准。

### **阶段二：并发初体验 - Goroutines 与 WaitGroup**

**学习目标：**

* 理解 goroutine 的概念和 go 关键字的使用。
* 理解为什么需要等待 goroutine 执行完成。
* 掌握 sync.WaitGroup 的使用方法 (Add, Done, Wait)。

**实践步骤：**

1. 改造 main 函数以支持并发：  
   我们将使用 WaitGroup 来确保 main 函数会等待所有的检查工作都完成之后再退出。  
   import "sync"  
     
   func main() {  
    // ... 文件读取部分代码不变 ...  
     
    var wg sync.WaitGroup // 声明一个 WaitGroup  
     
    fmt.Println("开始并发检查...")  
    for \_, url := range urls {  
    wg.Add(1) // 在启动一个 goroutine 前，计数器+1  
    go func(u string) {  
    defer wg.Done() // goroutine 结束时，计数器-1  
    result := checkURL(u)  
    if result.Error != nil {  
    fmt.Printf("URL: %s, 状态: 失败, 错误: %v\n", result.URL, result.Error)  
    } else {  
    fmt.Printf("URL: %s, 状态码: %d, 延迟: %v\n", result.URL, result.StatusCode, result.Latency)  
    }  
    }(url) // 注意这里必须把 url 作为参数传入匿名函数，避免闭包问题  
    }  
     
    wg.Wait() // 阻塞，直到所有 goroutine 都调用了 Done()  
    fmt.Println("检查完成！")  
   }

运行与验证：

再次运行 go run main.go。你会发现输出结果的顺序是混乱的，因为它们是并发执行的。最重要的是，总耗时会大大缩短，大约等于耗时最长的那个请求的时间。你已经成功实现了并发！

思考题：

如果直接在 go func() { ... }(url) 的匿名函数里使用 url 变量而不是传入参数 u，会发生什么？（提示：goroutine的调度与for循环变量的生命周期）

### **阶段三：Go的哲学 - 使用 Channel 安全地通信**

**学习目标：**

* 理解 channel 的作用：在 goroutine 之间传递数据。
* 掌握 channel 的创建 (make)、发送 (<-)、接收 (<-) 和关闭 (close)。
* 学会使用 range 优雅地遍历 channel。

问题背景：

在阶段二中，我们直接在 goroutine 里打印结果。但通常我们需要将结果收集起来，进行统一处理（比如排序、计算统计数据等）。如果多个 goroutine 同时去写一个共享的切片（slice），就会产生数据竞争（race condition）。channel 是解决这个问题的最佳方案。

**实践步骤：**

1. **再次改造 main 函数，引入 channel**：  
   func main() {  
    // ... 文件读取部分代码不变 ...  
     
    var wg sync.WaitGroup  
    // 创建一个带缓冲的 channel，容量等于URL的数量  
    resultsChannel := make(chan CheckResult, len(urls))  
     
    fmt.Println("开始并发检查 (使用Channel)...")  
    for \_, url := range urls {  
    wg.Add(1)  
    go func(u string) {  
    defer wg.Done()  
    // 将检查结果发送到 channel  
    resultsChannel <- checkURL(u)  
    }(url)  
    }  
     
    // 启动一个新的 goroutine，它专门负责等待所有检查完成后关闭 channel  
    // 这样做可以防止主流程阻塞在 wg.Wait()，从而无法接收 channel 的数据  
    go func() {  
    wg.Wait()  
    close(resultsChannel)  
    }()  
     
    // 使用 for range 遍历 channel，直到 channel 被关闭  
    // 这个循环会在这里阻塞，直到有数据可读或 channel 关闭  
    for result := range resultsChannel {  
    if result.Error != nil {  
    fmt.Printf("URL: %s, 状态: 失败, 错误: %v\n", result.URL, result.Error)  
    } else {  
    fmt.Printf("URL: %s, 状态码: %d, 延迟: %v\n", result.URL, result.StatusCode, result.Latency)  
    }  
    }  
     
    fmt.Println("检查完成！")  
   }

运行与验证：

效果和阶段二类似，但现在的代码结构更清晰、更安全、更符合Go的风格。我们把“做事”（checkURL）和“收集结果”这两个关注点分离开来了。

### **阶段四：精细化控制 - Worker Pool（工作池模式）**

**学习目标：**

* 理解无限制创建 goroutine 的潜在风险（资源耗尽）。
* 学会用 “工作池” 模式来控制并发数量。
* 进一步熟练 channel 的应用（一个任务 channel，一个结果 channel）。

问题背景：

如果 urls.txt 里有一万个网址，阶段三的代码会瞬间创建一万个 goroutine。这可能会耗尽你电脑的文件描述符、内存或CPU资源。我们需要一种方法来限制同时运行的 goroutine 数量。

**实践步骤：**

1. **定义 worker 函数**：  
   // worker 函数从 jobs channel 接收任务，并将结果发送到 results channel  
   func worker(id int, jobs <-chan string, results chan<- CheckResult) {  
    for url := range jobs {  
    fmt.Printf("Worker %d 开始处理 %s\n", id, url)  
    result := checkURL(url)  
    results <- result  
    }  
   }
2. **重构 main 函数，实现工作池逻辑**：  
   func main() {  
    // ... flag 解析代码 ...  
     
    // 新增一个 flag 用于控制并发的 worker 数量  
    concurrency := flag.Int("c", 10, "并发的 worker 数量")  
    flag.Parse() // 注意 flag.Parse() 只调用一次  
     
    // ... 文件读取代码 ...  
     
    // 创建任务 channel 和结果 channel  
    jobs := make(chan string, len(urls))  
    results := make(chan CheckResult, len(urls))  
     
    // 启动指定数量的 worker  
    for w := 1; w <= \*concurrency; w++ {  
    go worker(w, jobs, results)  
    }  
     
    // 将所有 URL 发送到任务 channel  
    for \_, url := range urls {  
    jobs <- url  
    }  
    close(jobs) // 发送完所有任务后，关闭 jobs channel  
     
    // 收集所有结果  
    for a := 1; a <= len(urls); a++ {  
    result := <-results  
    if result.Error != nil {  
    fmt.Printf("URL: %s, 状态: 失败, 错误: %v\n", result.URL, result.Error)  
    } else {  
    fmt.Printf("URL: %s, 状态码: %d, 延迟: %v\n", result.URL, result.StatusCode, result.Latency)  
    }  
    }  
    fmt.Println("检查完成！")  
   }

运行与验证：

现在，你可以通过命令行参数来控制并发数了！尝试用不同的并发数运行：

* go run main.go -c=5
* go run main.go -c=20  
  观察总耗时和输出，体会并发数对性能的影响。你已经实现了一个健壮的并发模型。

### **阶段五：锦上添花 - 格式化输出与统计**

**学习目标：**

* 使用 text/tabwriter 包美化终端输出。
* 进行基本的数据处理和统计。

**实践步骤：**

1. **在 main 函数收集结果后，进行处理和打印**：  
   import (  
    "os"  
    "text/tabwriter"  
   )  
     
   func main() {  
    // ... worker 池逻辑 ...  
     
    var allResults []CheckResult  
    // 收集所有结果  
    for a := 1; a <= len(urls); a++ {  
    result := <-results  
    allResults = append(allResults, result)  
    }  
     
    // 使用 tabwriter 格式化输出  
    w := tabwriter.NewWriter(os.Stdout, 0, 0, 3, ' ', tabwriter.AlignRight|tabwriter.Debug)  
    fmt.Fprintln(w, "URL\tStatusCode\tLatency\tError\t")  
    fmt.Fprintln(w, "---\t----------\t-------\t-----\t")  
     
    var successCount, failCount int  
    var totalLatency time.Duration  
     
    for \_, res := range allResults {  
    if res.Error != nil {  
    fmt.Fprintf(w, "%s\tN/A\tN/A\t%v\t\n", res.URL, res.Error)  
    failCount++  
    } else {  
    fmt.Fprintf(w, "%s\t%d\t%v\t%s\t\n", res.URL, res.StatusCode, res.Latency, "N/A")  
    successCount++  
    totalLatency += res.Latency  
    }  
    }  
    w.Flush() // 不要忘记 Flush  
     
    // 打印统计信息  
    fmt.Println("\n--- 统计信息 ---")  
    fmt.Printf("总计URL数量: %d\n", len(allResults))  
    fmt.Printf("成功数量: %d\n", successCount)  
    fmt.Printf("失败数量: %d\n", failCount)  
    if successCount > 0 {  
    fmt.Printf("平均延迟: %v\n", totalLatency/time.Duration(successCount))  
    }  
    fmt.Println("-----------------")  
   }

运行与验证：

现在的输出应该是一个对齐的、非常清晰的表格，并在最后附带了总结信息。你的工具已经非常专业了！

### **阶段六：灵魂升华 - 单元测试与性能基准**

**学习目标：**

* 掌握 testing 包，编写单元测试。
* 使用 net/http/httptest 创建模拟服务器，解耦测试依赖。
* 理解并使用 go test -race 来检测数据竞争。
* 编写基准测试（Benchmark），用数据证明并发的威力。

**实践步骤：**

1. **创建 main\_test.go 文件**。
2. 编写单元测试（测试 checkURL）：  
   我们需要一个模拟的HTTP服务器，这样测试就不会依赖真实网络。  
   package main  
     
   import (  
    "net/http"  
    "net/http/httptest"  
    "testing"  
    "time"  
   )  
     
   func TestCheckURL(t \*testing.T) {  
    // 创建一个模拟服务器  
    server := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r \*http.Request) {  
    // 模拟一个正常的响应  
    w.WriteHeader(http.StatusOK)  
    }))  
    defer server.Close()  
     
    result := checkURL(server.URL)  
     
    if result.Error != nil {  
    t.Errorf("期望没有错误，但得到了: %v", result.Error)  
    }  
    if result.StatusCode != http.StatusOK {  
    t.Errorf("期望状态码 %d, 但得到了 %d", http.StatusOK, result.StatusCode)  
    }  
    if result.Latency <= 0 {  
    t.Error("期望延迟大于0")  
    }  
     
    // 测试一个无效的 URL  
    invalidResult := checkURL("[http://invalid-url-that-will-fail.com](http://invalid-url-that-will-fail.com)")  
    if invalidResult.Error == nil {  
    t.Error("期望得到一个错误，但没有得到")  
    }  
   }  
     
   运行 go test，你应该能看到测试通过。
3. 运行竞争检测：  
   回到阶段二的代码（使用 WaitGroup 但没有 channel 的版本），如果你当时尝试修改一个共享切片，现在运行 go test -race 就会报告数据竞争。这是一个非常有价值的体验。对于我们当前使用 channel 的代码，运行 go test -race 应该不会报告任何问题。
4. 编写基准测试：  
   对比单线程版本和并发版本的性能。  
   // 在 main\_test.go 中添加  
   func BenchmarkSequentialCheck(b \*testing.B) {  
    urls := []string{  
    "[https://www.google.com](https://www.google.com)",  
    "[https://www.baidu.com](https://www.baidu.com)",  
    "[https://github.com](https://github.com)",  
    }  
    for i := 0; i < b.N; i++ {  
    for \_, url := range urls {  
    checkURL(url)  
    }  
    }  
   }  
     
   func BenchmarkConcurrentCheck(b \*testing.B) {  
    urls := []string{  
    "[https://www.google.com](https://www.google.com)",  
    "[https://www.baidu.com](https://www.baidu.com)",  
    "[https://github.com](https://github.com)",  
    }  
    for i := 0; i < b.N; i++ {  
    var wg sync.WaitGroup  
    for \_, url := range urls {  
    wg.Add(1)  
    go func(u string) {  
    defer wg.Done()  
    checkURL(u)  
    }(url)  
    }  
    wg.Wait()  
    }  
   }  
     
   运行 go test -bench=.。你会得到类似下面的输出，清晰地展示了并发带来的巨大性能提升。  
   BenchmarkSequentialCheck-8 ... 500 ms/op  
   BenchmarkConcurrentCheck-8 ... 150 ms/op

**总结与简历写法：**

恭喜你！完成以上六个阶段后，你不仅拥有了一个可以写在简历上的项目，更重要的是，你亲手实践了Go语言的并发精髓和测试体系。

在简历上，你可以这样描述这个项目：

**Go-Checker 并发网站状态检查工具**

* **项目描述**：为学习和实践Go语言并发模型，独立开发的一款命令行工具。该工具可从文件批量读取URL列表，并利用**Goroutine**并发检查各网站的HTTP状态码和响应延迟，最终生成格式化的报告。
* **技术实现**：
  + 通过**Goroutine**和**sync.WaitGroup**实现了高并发的请求处理，相比单线程执行，性能提升约 **X** 倍（根据你的Benchmark结果填写）。
  + 采用**Channel**作为Goroutine间的安全通信机制，避免了数据竞争，实现了结果的异步收集与汇总。
  + 实现了\*\*Worker Pool（工作池）\*\*模式，有效控制了并发粒度，防止了高负载下系统资源的耗尽。
  + 使用Go标准库（net/http, flag, os, text/tabwriter）构建整个应用，无任何第三方框架依赖。
  + 编写了**单元测试**（利用httptest模拟服务器）和**性能基准测试**，确保了核心功能的稳定性和高性能。

这个项目不大，但五脏俱全，完美地展现了你作为Go开发者的核心素养。祝你面试顺利！