#paper-reading

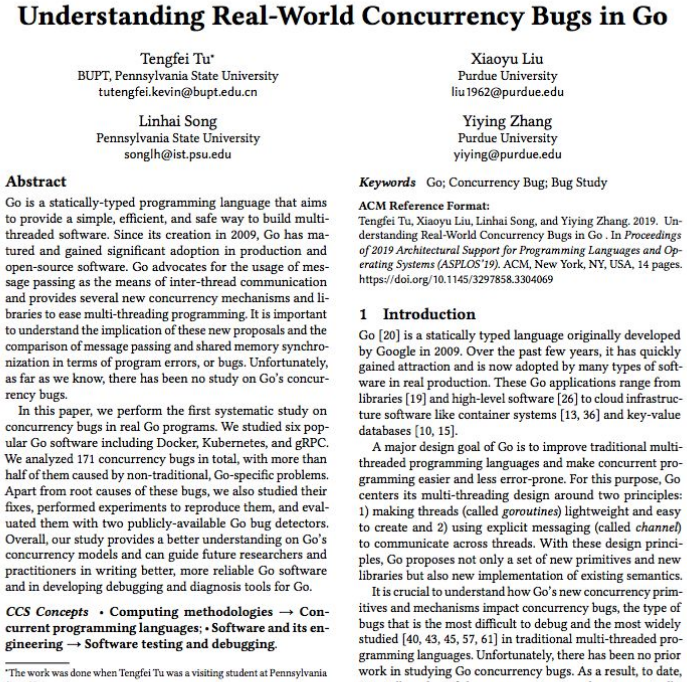
Real-world Go Concurrency Bugs

Go 夜读 SIG 小组 | 欧长坤

第 59 期
Sept. 12, 2019

主要内容

- 论文的研究背景和方法
- Go 并发 Bug 的分类以及部分主要结论
  - 阻塞式 Bug
  - 非阻塞式 Bug
- Go 运行时死锁、数据竞争检测器对 Bug 的检测能力及其算法原理
- 论文的结论、争议与我们的反思



Understanding Real-World Concurrency Bugs in Go

Tengfei Tu*
BUPT, Pennsylvania State University
tutengfei.kevin@bupt.edu.cn

Xiaoyu Liu
Purdue University
liu1962@purdue.edu

Linhai Song
Pennsylvania State University
songlh@ist.psu.edu

Yiying Zhang
Purdue University
yiying@purdue.edu

Abstract

Go is a statically-typed programming language that aims to provide a simple, efficient, and safe way to build multi-threaded software. Since its creation in 2009, Go has matured and gained significant adoption in production and open-source software. Go advocates for the usage of message passing as the means of inter-thread communication and provides several new concurrency mechanisms and libraries to ease multi-threading programming. It is important to understand the implication of these new proposals and the comparison of message passing and shared memory synchronization in terms of program errors, or bugs. Unfortunately, as far as we know, there has been no study on Go's concurrency bugs.

In this paper, we perform the first systematic study on concurrency bugs in real Go programs. We studied six popular Go software including Docker, Kubernetes, and gRPC. We analyzed 171 concurrency bugs in total, with more than half of them caused by non-traditional, Go-specific problems. Apart from root causes of these bugs, we also studied their fixes, performed experiments to reproduce them, and evaluated them with two publicly-available Go bug detectors. Overall, our study provides a better understanding on Go's concurrency models and can guide future researchers and practitioners in writing better, more reliable Go software and in developing debugging and diagnosis tools for Go.

CCS Concepts • **Computing methodologies → Concurrent programming languages;** • **Software and its engineering → Software testing and debugging.**

Keywords Go; Concurrency Bug; Bug Study

ACM Reference Format:
Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go . In *Proceedings of 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS'19).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304069

1 Introduction

Go [20] is a statically typed language originally developed by Google in 2009. Over the past few years, it has quickly gained attraction and is now adopted by many types of software in real production. These Go applications range from libraries [19] and high-level software [26] to cloud infrastructure software like container systems [13, 36] and key-value databases [10, 15].

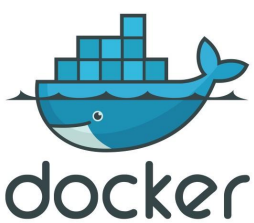
A major design goal of Go is to improve traditional multi-threaded programming languages and make concurrent programming easier and less error-prone. For this purpose, Go centers its multi-threading design around two principles: 1) making threads (called *goroutines*) lightweight and easy to create and 2) using explicit messaging (called *channel*) to communicate across threads. With these design principles, Go proposes not only a set of new primitives and new libraries but also new implementation of existing semantics.

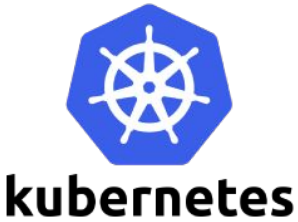
It is crucial to understand how Go's new concurrency primitives and mechanisms impact concurrency bugs, the type of bugs that is the most difficult to debug and the most widely studied [40, 43, 45, 57, 61] in traditional multi-threaded programming languages. Unfortunately, there has been no prior work in studying Go concurrency bugs. As a result, to date,

*The work was done when Tengfei Tu was a visiting student at Pennsylvania

Go 研究背景与方法 (1)

- Go 支持消息通信（channel）与共享内存（sync*）两种形式的同步原语，并提倡用户多使用 channel。
- 研究问题：channel 这一同步机制是否真的让**使用 Go 进行编程更加容易且犯更少**错误？
- "首个" 针对复杂**大型**软件的调研：





Go 研究背景与方法 (2)

- 产生 Bug 的两个维度：**原因**和**行为**
  - **原因**：错误的共享内存、错误的消息通信
  - **行为**：阻塞式（任意多个 goroutine 不能继续执行）非阻塞式（没有发生阻塞的 goroutine）
- 经验**方法**：总结了六个真实的生产环境软件在开发过程中产生的总计 171 个 bug

创建 goroutine 的频率 vs 创建 thread 的频率 (1)

Application	Normal F.	Anonymous F.	Total	Per KLOC
Docker	33	112	145	0.18
Kubernetes	301	223	531	0.23
etcd	86	211	297	0.67
CockroachDB	27	125	152	0.29
gRPC-Go	14	30	44	0.83
BoltDB	2	0	2	0.22
gRPC-C	5	-	5	0.03

创建 goroutine 的频率 vs 创建 thread 的频率 (2)

Workload	Goroutine/Thread		Ave. Execution Time	
	Client	Server	client-Go	server-Go
g_sync_ping_pong	7.33	2.67	63.65%	76.97%
sync_ping_pong	7.33	4	63.23%	76.57%
qps_unconstrained	501.46	6.36	91.05%	92.73%

- gRPC-C 中所有线程的执行时间占应用执行时间的 100%

观察1: 无论从代码上还是运行时上看，goroutine 比 thread 使用得更加频繁。

同步原语的使用频率 (1)

Application	Shared Memory					Message Passing		Total
	Mutex	atomic	Once	WaitGroup	Cond	chan	Misc.	
Docker	62.62%	1.06%	4.75%	1.7%	0.99%	27.87%	0.99%	1410
Kubernetes	70.34%	1.21%	6.13%	2.68%	0.96%	18.48%	0.20%	3951
etcd	45.01%	0.63%	7.18%	3.95%	0.24%	42.99%	0	2075
CockroachDB	55.90%	0.49%	3.76%	8.57%	1.48%	28.23%	1.57%	3245
gRPC-Go	61.20%	1.15%	4.20%	7.00%	1.65%	23.03%	1.78%	786
BoltDB	70.21%	2.13%	0	0	0	23.40%	4.26%	47

同步原语的使用频率 (2)

- 2015.02 ~ 2018.05
- 使用频率比较稳定

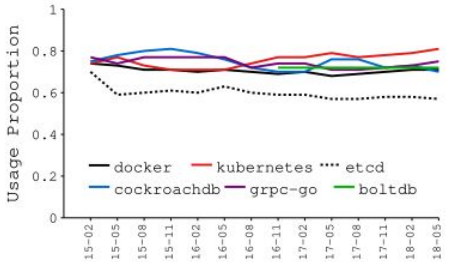


Figure 2. Usages of Shared-Memory Primitives over Time. *For each application, we calculate the proportion of shared-memory primitives over all primitives.*

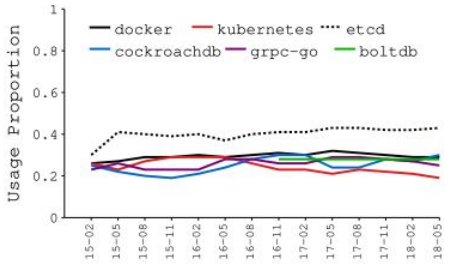


Figure 3. Usages of Message-Passing Primitives over Time. *For each application, we calculate the proportion of message-passing primitives over all primitives.*

观察2: 尽管共享内存原语仍然频繁使用，Go 程序员仍然会使用一定数量的消息传递原语。

研究方法

分类结果

Application	Behavior		Cause	
	blocking	non-blocking	shared memory	message passing
Docker	21	23	28	16
Kubernetes	17	17	20	14
etcd	21	16	18	19
CockroachDB	12	16	23	5
gRPC	11	12	12	11
BoltDB	3	2	4	1
Total	85	86	105	66

Table 5. Taxonomy. *This table shows how our studied bugs distribute across different categories and applications.*

Blocking Bugs v.s. Non-Blocking Bugs

Application	Shared Memory			Message Passing		
	Mutex	RWMutex	Wait	Chan	Chan w/	Lib
Docker	9	0	3	5	2	2
Kubernetes	6	2	0	3	6	0
etcd	5	0	0	10	5	1
CockroachDB	4	3	0	5	0	0
gRPC	2	0	0	6	2	1
BoltDB	2	0	0	0	1	0
Total	28	5	3	29	16	4

Table 6. Blocking Bug Causes. *Wait includes both the Wait function in* `Cond` *and in* `WaitGroup`*. Chan indicates channel operations and Chan w/ means channel operations with other operations. Lib stands for Go libraries related to message passing.*

- 42% blocking bug 由共享内存导致
- 58% blocking bug 由消息通信导致

Application	Shared Memory				Message Passing	
	traditional	anon.	waitgroup	lib	chan	lib
Docker	9	6	0	1	6	1
Kubernetes	8	3	1	0	5	0
etcd	9	0	2	2	3	0
CockroachDB	10	1	3	2	0	0
gRPC	8	1	0	1	2	0
BoltDB	2	0	0	0	0	0
Total	46	11	6	6	16	1

Table 9. Root causes of non-blocking bugs. *traditional: traditional non-blocking bugs; anonymous function: non-blocking bugs caused by anonymous function; waitgroup: misusing WaitGroup; lib: Go library; chan: misusing channel.*

- 80% non-blocking bug 由共享内存导致
- 20% non-blocking bug 由消息通信导致

观察3: 与普遍认为消息通信产生更少 bug 的观点不同，实际中它导致比共享内存更多的 Bug。

Selected Bugs

Bug 1: Blocking - Shared Memory Docker#25384

解释：group.Wait() 会发生永久阻塞，因为 group.Done() 只会被调用一次

```
  var group sync.WaitGroup
  group.Add(len(pm.plugins))
  for _, p := range pm.plugins {
      go func(p *plugin) {
          defer group.Done()
      }
-     group.Wait()
  }
+ group.Wait()
```

观察4: 部分共享内存导致的阻塞 bug 的成因在于 Go 使用了新的实现语义

Bug 2: Blocking - Message Passing Kubernetes#5316

解释：如果发生 timeout，则向 ch 发送数据的 goroutine 会由于没有接收方而产生 goroutine 泄漏。

```
func finishReq(timeout time.Duration) ob {
-       ch := make(chan ob)
+       ch := make(chan ob, 1)
        go func() {
            result := fn()
            ch <- result // block
        }()
        select {
        case result := <- ch:
                return result
        case <- time.After(timeout):
                return nil
        }
}
```

Bug 3: Blocking - Message Passing (unknown source)

解释：当 timeout > 0 时候，原先创建的 context 会被覆盖并无法再被关闭，产生 goroutine 泄漏。

```
- hctx, hcancel := context.WithCancel(ctx)
+ var hctx, context.Context
+ var hcancel context.CancelFunc
  if timeout > 0 {
      hctx, hcancel = context.WithTimeout(ctx, timeout)
+ } else {
+     hctx, hcancel = context.WithCancel(ctx)
  }
```

Bug 4: Blocking - Message Passing + Shared Memory (unknown source)

解释：当 g1 向 channel 发送数据时，由于某种原因需要持有 Mutex 锁，但 g2 会因此阻塞在 Mutex 锁上，从而无法接收到数据。

```
func g1() {
    m.Lock()
-   ch <- request // blocks
+   select {
+   case ch <- request:
+   default:
+   }
    m.Unlock()
}

func g2() {
    for {
        m.Lock() // blocks
        m.Unlock()
        request <- ch
    }
}
```

观察5: 所有由消息传递导致的阻塞 bug 都与 Go 的新的消息传递语义有关

Go 的死锁检测器的原理与局限

- **论文中的 lift 指标**
  - **lift(cause,fix) = P(cause, fix)/(P(cause)P(fix))**
  - **Mutex lift value = 1.52**
- 原理：#m - #midle - #miclelocked - #msys > 0
  - 只检查是否所有的工作线程都已休眠
- 缺点：
  - 当存在可运行的 goroutine 时，系统监控死锁检测不会认为发生阻塞
  - 运行时只检查 goroutine 是否在 Go 并发原语中被阻止，不考虑等待其他系统资源的 goroutine。

	Add_s	$Move_s$	$Change_s$	$Remove_s$	Misc.
Shared Memory					
Mutex	9	7	2	8	2
Wait	0	1	0	1	1
RWMutex	0	2	0	3	0
Message Passing					
Chan	15	1	5	4	4
Chan w/	6	3	2	4	1
Messaging Lib	1	0	0	1	2
Total	31	14	9	21	10

Table 7. Fix strategies for blocking bugs. *The subscript s stands for synchronization.*

Root Cause	# of Used Bugs	# of Detected Bugs
Mutex	7	1
Chan	8	0
Chan w/	4	1
Messaging Libraries	2	0
Total	21	2

Table 8. Benchmarks and evaluation results of the deadlock detector.

观察6: 大多数阻塞 bug可以由简单的方法进行修复，且大部分修复都与 导致 bug 的原因有关

Bug 5: Non-Blocking - Shared Memory (Docker#22985, CockroachDB#6111)

解释：向通道内发送指针，导致的共享内存。

```
func (m *mockcwlogsclient) PutLogEvents(input *PutLogEventsInput) (*PutLogEventsOutput, error) {
-   m.putLogEventsArgument <- input
+   events := make([]*InputLogEvent, len(input.LogEvents))
+   copy(events, input.LogEvents)
+   m.putLogEventsArgument <- &PutLogEventsInput{
+       LogEvents:     events,
+       SequenceToken: input.SequenceToken,
+       LogGroupName:  input.LogGroupName,
+       LogStreamName: input.LogStreamName,
+   }
    output := <-m.putLogEventsResult
    return output.successResult, output.errorResult
}
```

Bug 6: Non-Blocking - Shared Memory (unknown source)

解释：wg.Add(1) 可能在 wg.Wait() 之后被执行

```
func (p *peer) send() {
    p.mu.Lock()
    defer p.mu.Unlock()
    switch p.status {
    case idle:
+       p.wg.Add(1)
        go func() {
-           p.wg.Add(1)
            ...
            p.wg.Done()
        }()
    }
}

func (p *peer) stop() {
    p.mu.Lock()
    p.status = stopped
    p.mu.Unlock()
    p.wg.Wait()
}
```

Bug 7: Non-Blocking - Shared Memory (unknown source)

解释：i 在父 goroutine 中写，在子 goroutine 中读，并发导致数据竞争。

```
for i := 17; i <= 21; i++ { // write
-    go func()
+    go func(i int) {
         apiVersion := fmt.Sprintf("v1.%d", i) // read
         ...
-    }()
+    }(i)
}
```

观察7: ⅓ 的非阻塞共享内存 bug 是由于新的并发语义导致的

Bug 8: Non-Blocking - Message Passing (Docker#24007)

解释：可能存在多个 goroutine 进入 default 分支从而导致 channel 被多次关闭，进而引发运行时错误。

```
- select {
- case <- c.closed:
- default:
+     once.Do(func() {
          close(c.closed)
+     })
- }
```

Bug 9: Non-Blocking - Message Passing (unknown source)

解释：如果 stopCh 和 ticker 同时到达，ticker 可能会被随机选中，从而导致 f 被多执行一次。

```
    ticker := time.NewTicker()
    for {
+       select {
+       case <- stopCh:
+           return
+       default:
+       }
        f()
        select {
        case <- stopCh:
            return
        case <- ticker:
        }
    }
```

Bug 10: Non-Blocking - Message Passing (Docker#21692)

解释：当 dur 大于零或者 ctx.Done() 时会正常返回。但当 dur 小于等于零时，timer.C 会立刻受到通知，从而导致过早返回。

```
- timer := time.NewTimer(0)
+ var timeout <- chan time.Time
  if dur > 0 {
-     timer = time.NewTimer(dur)
+     timeout = time.NewTimer(dur).C
  }
  select {
- case <- timer.C:
+ case <- timeout:
  case <- ctx.Done():
      return nil
  }
```

观察8: 由消息传递导致的非阻塞 bug 相对于共享内存导致的 bug 更少

非阻塞 bug 的修复

	Timing		Instruction Bypass	Data Private	Misc.
	Add_s	$Move_s$			
Shared Memory					
traditional	27	4	5	10	0
waitgroup	3	2	1	0	0
anonymous	5	2	0	4	0
lib	1	2	1	0	2
Message Passing					
chan	6	7	3	0	0
lib	0	0	0	0	1
Total	42	17	10	14	3

Table 10. Fix strategies for non-blocking bugs. *The subscript s stands for synchronization.*

	Mutex	Channel	Atomic	WaitGroup	Cond	Misc.	None
Shared Memory							
traditional	24	3	6	0	0	0	13
waitgroup	2	0	0	4	3	0	0
anonymous	3	2	3	0	0	0	3
lib	0	2	1	1	0	1	2
Message Passing							
chan	3	11	0	2	1	2	1
lib	0	1	0	0	0	0	0
Total	32	19	10	7	4	3	19

Table 11. Synchronization primitives in patches of non-blocking bugs.

观察9: 传统的共享内存同步技术仍然是修复 Go 中非阻塞 bug 的主要修复手段

Go 的数据竞争检测器的原理

- 数据竞争检测可以通过代码静态分析或运行时动态检测实施
- 静态分析由于不能检测分配在堆数据等原因，未被广泛使用
- 常见的两个动态分析算法：
  - Happens-before 算法 （Go 采用的数据竞争检测方法）
  - Lockset 算法

Root Cause	# of Used Bugs	# of Detected Bugs
Traditional Bugs	13	7
Anonymous Function	4	3
Lib	2	0
Misusing Channel	1	0
Total	20	10

Table 12. Benchmarks and evaluation results of the data race detector. *We consider a bug detected within 100 runs as a detected bug.*

Happens-before 算法

- 本质：向量时钟算法
- 缺点
  - 必须在每个共享变量的位置记录每个线程的信息
  - 依赖于调度器，可能遗漏错误，例如

```
  Goroutine 1
b += 1
mu.Lock()
a += 1
mu.Unlock()
```

happens-before

```
  Goroutine 2
mu.Lock()
a += 1
mu.Unlock()
b += 1
```

b 可能发生竞争，但也可能不会被检测到。

结论

- 观察1 => 结论1: 频繁使用 goroutine 和新的并发原语，**Go 程序可能产生更多的并发 Bug**。
- 结论2:
  - 观察2,3,4,5 => 消息传递可能导致比共享内存更多的阻塞 Bug
  - 观察7 => Go 引入的新的编程模型和库可能就是导致更多并发 bug的原因
  - 观察8 => 当正确使用时，消息传递比共享内存而言更少出现非阻塞 bug
  - 观察9 => Go 程序更喜欢使用消息传递作为特殊情况下的一个修复非阻塞 bug 的手段，因为他们认为消息传递是一种更加安全的线程间通信手段
- 观察6 => 结论3: Go的阻塞 Bug 的成因和修复的简洁性表明，开发自动 Bug 修复工具很有希望
- 观察6 => 结论4: 简单的运行时死锁检查器不足以检查 Go 的阻塞 bug
- 观察9 => 结论5: 简单的数据竞争检测器不能有效的检测所有 Go 中的非阻塞 bug。

这篇文章的巨大贡献：

总结了非常多数量且有用的真实 Bug，用以警示我们未来使用这些同步原语时可能会犯的错误。

争议

- **没有真正的对照组**：达成同样目的的项目，在使用不同语言提供的同步机制时（例如 C++ 同样也可以使用 MPI 来实现消息通信），是否一方比另一方所犯错误更多？
- **没有切实意义的比较**：goroutine 本身作为一种 thread 的 multiplex，自然会更频繁的使用
- **并不正交的两个维度**："消息传递产生的阻塞 bug 更多，共享内存产生的非阻塞 bug 更多" 这个结论是不证自明的，从消息传递的语义上而言，更容易产生阻塞（unbuffered channel、消息队列满）；从共享内存的语义上而言，更容易产生非阻塞（竞争变量未被正确保护）
- **不够合理的研究方法**：
  - 运行时死锁检查器并未真正设计为能够检查死锁，统计性的评估检查器检查 bug 的能力是没有意义的；
  - 数据竞争检查器是检查数据竞争的，自然不能检查非数据竞争产生的非阻塞 bug；happens-before 算法本身就有漏报的缺陷；这种统计性的评估仍然是没有意义的
- ...

讨论 Q&A

【Go 夜读问卷调查】
#59 Real-world Go Concurrency Bugs

Q：可以再详细说一下 lift 指标吗？

A：可以从两种不同的角度来思考这个指标。

1. 借鉴 person 相关性系数（余弦相似性）|X·Y|/(|X|*|Y|)
2. 借鉴 Bayes 公式 P(B|A) = P(AB)/P(A)

- Lift(cause, fix) = 导致阻塞的 cause 且使用了 fix 进行修复的概率 除以 cause 的概率乘以 fix 的概率 = P(cause, fix) / (P(cause)P(fix)) = P(cause|fix)/P(cause)
- 接近 1 时，说明 fix 导致 cause 的概率接近 cause 自己的概率，即 P(cause|fix) 约等于 P(cause) 于是 fix 和 cause 独立
- 大于 1 时，说明 fix 导致 cause 的概率比 cause 自己的概率大，即 P(cause|fix) > P(cause) => P(fix | cause) > P(fix)，即 cause 下 fix 的概率比 fix 本身的概率大，正相关
- 小于 1 时，同理，负相关

Q：可以贴一下提到的两篇相关文献吗？

A：论文引用了两篇很硬核的形式化验证的论文：

- Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off go: Liveness and safety for channel-based programming. In Proceedings ofthe 44th ACMSIGPLANSymposium on Principles of Programming Languages (POPL '17), Paris, France, January 2017.
- Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in go using be- havioural types. In IEEE/ACM40th International Conference on Software Engineering (ICSE '18), Gothenburg, Sweden, June 2018.

讨论 Q&A

【Go 夜读问卷调查】
#59 Real-world Go Concurrency Bugs

Q：作者还分享了其他语言的关于并发 Bug 的论文，比如 Rust。

A：地址在这里，但是思路完全一致，可以直接扫一眼结论。

Q：能否将 CSP 和 Actor 模型进行一下简单比较？

A：CSP 和 Actor 的本质区别在于如何对消息传递进行建模。Actor 模型强调每个通信双方都有一个“信箱”，传递消息的发送方对将这个消息发给谁是很明确的，这种情况下 Actor 的信箱必须能够容纳不同类型的消息，而且理论上这个信箱的大小必须无穷大，很像你想要送一件礼物给别人，你会直接把礼物递给这个人，如果这个人不在，你就扔到他家的信箱里 CSP 需要有一个信道，因此对发送方而言，其实它并不知道这个消息的接收方是谁，更像是你朝大海扔了一个漂流瓶，大海这个信道根据洋流将这个漂流瓶传递给了其他正在观察监听大海的人。

Q：读论文的目标是什么？

A：我读论文主要有两个目标：1. 了解论文的研究方法，因为研究方法可能可以用在我未来的研究中 2. 了解论文的整体思路，因为论文很多，思路远比它们的结果对我未来自己的研究更重要。

Q：去哪儿找这类论文？

A：我们这次讨论的论文是我偶然在 Go 语言 GitHub 仓库的 Wiki 上看到的；一般情况下我会订阅 ArXiv，然后定期浏览新发出来的文章。

进一步阅读的参考文献

1. 论文相关的一些链接

- 论文 Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19). ACM, New York, NY, USA, 865-878. https://songlh.github.io/paper/go-study.pdf
- PPT https://slideplayer.com/slide/17049966/
- 仓库 https://github.com/system-pclub/go-concurrency-bugs
- Hacker News 的讨论：https://news.ycombinator.com/item?id=19280927
- Go channels are bad and you should feel bad：https://www.jtolio.com/2016/03/go-channels-are-bad-and-you-should-feel-bad/
- 与论文作者的一次面谈：https://www.jexia.com/en/blog/golang-error-proneness-message-passing/

2. Go 夜读 第 56 期：channel & select 源码分析 https://github.com/developer-learning/reading-go/issues/450