了解Go中的真实世界并发错误

# 摘要

腾飞图\*

宾夕法尼亚州立大学

[tutengfei.kevin@bupt.edu.cn](mailto:tutengfei.kevin@bupt.edu.cn)

临海歌

宾夕法尼亚州立大学

[songlh@ist.psu.edu](mailto:songlh@ist.psu.edu)

刘晓宇普渡大学liu1962@purdue.edu

张怡英普渡大学yiying@purdue.edu

***关键词围棋；并发***错误；Bug研究

Go是一种静态类型的编程语言，旨在提供一种简单、高效、安全的方法来构建多线程软件。自2009年创立以来，Go已在生产和开源软件中获得了大量采用。Go提倡使用mes-sage传递作为线程间通信的手段，并提供了几种新的并发机制和库来简化多线程编程。了解这些新建议的含义以及消息传递和共享内存同步在程序错误或错误方面的比较非常重要。不幸的是，据我们所知，目前还没有关于围棋一致性错误的研究。

在本文中，我们对真实Go程序中的并发错误进行了首次系统研究。我们研究了六种流行的Go软件，包括Docker、Kubernetes和gRPC。我们总共分析了171个并发错误，其中一半以上是由非传统的Go特定问题引起的。除了这些bug的根本原因之外，我们还研究了它们的修复方法，进行了实验以重现它们，并使用两个公开可用的Go bug检测器对它们进行了评估。总体而言，我们的研究提供了对围棋并发模型的更好理解，并可以指导未来的研究人员和从业者编写更好、更可靠的围棋软件，以及开发围棋调试和诊断工具。

## *CCS概念•计算方法→ 当*前编程语言；•软件及其工程→ 软件测试和调试。

\*这项工作是在Tu Tengfei Tu担任宾夕法尼亚州立大学访问学生时完成的。

允许制作本作品的全部或部分的数字或硬拷贝供个人或课堂使用，但前提是复制品的制作或分发不是为了盈利或商业利益，并且复制品在第一页带有本通知和完整引用。必须尊重ACM以外的其他人拥有的本作品组件的版权。允许使用信用进行抽象。以其他方式复制、重新发布、发布到服务器或重新分发到列表，需要事先获得特定许可和/或支付费用。从请求权限permissions@acm.org.

*ASPLOS’19，2019年4月13日至17日，美国佛罗里达州普罗维登斯*

©2019计算机械协会。ACM ISBN ISBN 978-1-4503-6240-5/04$15

<https://doi.org/10.1145/3297858.3304069>

**ACM参考格式：**

涂腾飞、刘晓宇、宋林海和张怡英。2019.围棋中的真实世界并发漏洞。《2019年编程语言和操作系统体系结构支持论文集》（ASPLOS'19）。ACM，纽约，纽约，美国，14页。https://doi.org/10*.1145/3297858.3304069*

# 1 简介

Go[2[0]](#_bookmark48)是一种静态类型语言，最初由谷歌于2009年开发。在过去的几年中，它很快获得了吸引力，现在在实际生产中被许多类型的软件所采用。这些Go应用范围从库[19]和高级软件[26]到云基础设施软件，如容器系统[13，36]和关键值数据库[10，15]。

Go的一个主要设计目标是改进传统的多线程编程语言，使并发编程更容易，更不容易出错。为此，Go将其多线程设计集中在两个原则上：

1） 使线程（称为goroutines）轻量*级且易于创建；2）使*用显式消息传递（称为通道）跨线程通信。凭借这些设计原则，Go不仅提出了一组新的原语和新的库，还提出了现有语义的新实现。了解Go的新并发原语*和机制如何影响*并发错误是至关重要的，这是传统多线程程序设计语言中最难调试和研究最广泛的错误类型[40、43、45、57、61]。不幸的是，以前没有研究Go并发错误的工作。因此，到目前为止，尚不清楚这些并发机制是否真的使Go更容易编程，更不容易出错-

rency比传统语言更容易出错。

在本文中，我们使用六个开源的生产级Go应用程序对Go并发bug进行了第一次实证研究：Docker[13]和Kubernetes[36]，两个数据中心容器系统，etcd[15]，一个分布式键值存储系统，gRPC[19]，一个RPC库，以及Cock-roachDB[10]和Bolt[DB](#_bookmark41)[6]，两个数据库系统。

总共，我们研究了这些应用程序中的171个并发错误。我们分析了它们的根本原因，进行了复制它们的实验，并检查了它们的修复情况

补丁。最后，我们用两个现有的Go con-currency bug检测器（唯一公开的）测试了它们。

我们的研究集中于并发编程中一个长期存在的基本问题：消息传递[27，37]和共享内存之间，这些线程间通信机制中哪一种更不容易出错[2，11，48]。Go是研究这个问题的完美语言，因为它为共享内存和消息传递提供了框架[。然](#_bookmark55)而[，它](#_bookmark63)鼓励在共享内存上使用通道，并认为显式消息传递不易出错[1，2，21]。

为了理解围棋比赛bug以及消息传递和共享内存之间的比较，我们建议按照两个正交维度对比赛bug进行分类：bug的原因及其行为。沿着原因维度，我们将错误分为滥用共享内存导致的错误和滥用消息传递导致的错误。沿着第二个维度，我们将bug分为涉及（任意数量）无法继续的goroutine的bug（我们称之为阻塞bug）和不涉及任何阻塞的bug（非阻塞bug）。

令人惊讶的是，我们的研究表明，与共享内存一样，通过消息传递很容易产生货币错误，有时甚至更容易。例如，大约58%的阻塞错误是由消息传递引起的。除了违反Go的通道使用规则（例如，在没有人向其发送数据或关闭的通道上等待）之外，许多并发错误是由消息传递和其他新语义以及Go中的新库的混合使用导致的，这些错误很容易被忽略，但很难检测到。

为了演示消息传递中的错误，我们在图1中使用了Kubernetes中的一个阻塞错误。finishRequest函数在第4行使用匿名函数创建子goroutine来处理请求，这是Go服务器[程序中](#_bookmark1)的常见做法。子gorroutine执行fn（），并通过第6行的通道ch将结果发送回父gorroutine。子级将在第6行阻塞，直到父级从第9行的ch中提取结果。同时，父级将在select时阻塞，直到子级将结果发送给ch（第9行）或发生超时（第11行）。如果超时提前发生，或者如果Go运行时（非确定性）在两个情况都有效的情况下选择了第11行的情况，则父级将从第12行的requestReq（）返回，其他任何人都不能再从ch中提取结果，导致子级永远被阻止。修复方法是将ch从非缓冲通道更改为缓冲通道，这样子gorroutine即使在父级退出时也可以始终发送结果。

这个bug说明了在围棋中使用新功能的复杂性，以及编写正确的围棋程序的难度。程序员必须对使用匿名函数创建goroutine有一个清晰的理解，Go提出的一个功能是简化goroutine的创建，缓冲通道与非缓冲通道的使用，使用select等待多个通道操作的非确定性，

1. **请求（超**时时间.持续时间）r ob{
2. - ch:=**制造（否**）

3+ch :=制造**（cha**n  **ob，**1）

1. go**函数（）**{
2. 结果：=fn（）
3. ch<-结果//块

7 } ()

1. **选择{**
2. **案例结果**=<-ch:
3. **返回结果**
4. **case**<-time.After（超时）：
5. **返回零**

13 }

14 }

## 图1。由通道引起的阻塞错误。

以及特殊的图书馆时间。尽管这些功能中的每一个都是为了简化多线程编程而设计的，但实际上，很难用它们编写正确的Go程序。总体而言，我们的研究揭示了Go并发编程的新实践和新问题，并为消息传递与共享内存访问的争论提供了答案。我们的发现提高了对Go并发的理解，并可以为未来的工具设计提供有价值的指导。

本文做出了以下关键贡献。

我们对六个真实世界的生产级围棋应用程序进行了围棋一致性错误的第一次实证研究。

•

我们对Go货币错误的原因、修复和检测进行了九次高级关键观察。它们可用于Go程序员的参考。我们进一步对我们的研究结果的含义进行了八个深入的了解，以指导未来围棋开发、测试和错误检测的研究。

•

我们提出了新的方法，从错误原因和行为两个维度对并发错误进行分类。这种分类方法帮助我们更好地比较不同的并发机制以及错误原因和修复的对应关系。我们相信其他bug研究也可以使用类似的分类方法。

•

我们的所有研究结果和研究提交日志都可以在https://github.com/system-pclub/go-concu*rrency-bugs.*

# 背景和应用

Go是一种静态类型的编程语言，从一开始就为并发编程而设计[60]。所有主要的围棋修订都包括对其货币包的改进[23]。本节简要介绍了Go的并发机制，包括其线程模型、线程间通信方法和线程同步机制。我[们还](#_bookmark87)介绍了我们为本研究选择的六个Go应用程序。

## Goroutine公司

Go使用一个叫做goroutine的概念*作为其并发单元。G*oroutine是轻量级的用户级线程

运行库以M到N的方式管理并映射到内核级线程。只需在函数调用之前添加关键字go即可创建goroutine。

为了使goroutine易于创建，Go还支持使用匿名函数（一个没有标识符或“名称”的函数定义）创建新的goroutine。匿名函数可以访问在匿名函数之前声明的所有局部变量，并*且可能在父goroutine和使用匿*名函数创建的子goroutine之间共享，从而导致数据竞争（第6节）。

## 与共享内存同步

Go支持跨goroutine的传统共享内存访问。它支持各种传统的同步原语，如锁定/解锁（Mutex）、读/写锁定（RWMutex），条件变量（Cond）和原子读/写（atomic）。Go对RWMutex的实现不同于C中的pthread\_rwlock\_t。Go中的写锁请求比读锁请求具有更高的权限。

作为Go引入的一个新原语，Once旨在确保函数只执行一次。它有一个Do方法，函数f作为参数。当多次调用Once.Do（f）时，仅在第一次执行f。Once被广泛用于确保共享变量只能由多个goroutine初始化一次。

与C中的pthread\_join类似，Go使用WaitGroup来调用多个goroutine，以在等待goroutine之前完成它们的共享变量进程。通过调用Add将Goroutine添加到WaitGroup。WaitGroup中的goroutine使用Done来通知其完成，而goroutine调用Wait来等待WaitGroup内所有goroutine的完成通知。误用WaitGroup会导致阻塞错误（第5节）和非阻塞错误（6节）。

## 与消息传递同步

通道（chan）是Go引入的一种新的并发原语，用于跨goroutine发送数据和状态，并构建更复杂的功能[3，50]。Go支持两种类型的通道：缓冲通道和非缓冲通道。向非缓冲信道发送数据（或从非缓冲信道接收数据）将阻止一个goroutine，直到另一个gorroutine[从](#_bookmark31)该信[道接](#_bookmark77)收数据（或向其发送数据）。当缓冲区已满时，发送到缓冲通道将仅阻塞。在使用通道时有几个基本规则，违反这些规则会导致并发错误。例如，通道只能在初始化后使用，向nil通道发送数据（或从nil通道接收数据）将永远阻止goroutine。向关闭的通道发送数据或关闭已关闭的通道可能会触发运行时死机。

select语句允许goroutine等待多路通道操作。一个select将阻塞，直到它的一个案例能够取得进展，或者当它能够执行一个默认分支时。当选择中有多个案例有效时，请转到

选择一个要执行。将随机选择一个执行。这种随机性会导致并发错误，这将在第6节中讨论。

Go引入了几个新的语义来简化多个goroutine之间的交互。例如，为了通过生成一组协同工作的goroutine来帮助服务用户请求的编程模型，Go引入了上下文来跨goroutine承载请求特定的数据或元数据。发生未知错误。另一个示例是，管道设计用于在读取器和写入器之间传输数据。上下文和管道都是传递消息的新形式，误用它们会产生新类型的并发错误（第5节）。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **应用** | **星星** | **委员会** | **贡献者** | **位置** | **开发历史记录** |
| Docker公司 | 48975 | 35149 | 1767 | 78.6万 | 4.2年 |
| Kubernets公司 | 36581 | 65684 | 1679 | 2297公里 | 3.9年 |
| 蚀刻 | 18417 | 14101 | 436 | 441千 | 4.9年 |
| 蟑螂DB | 13461 | 29485 | 197 | 5.2万 | 4.2年 |
| gRPC\* | 5594 | 2528 | 148 | 53千 | 3.3年 |
| 螺栓DB | 8530 | 816 | 98 | 9千 | 4.4年 |

**表1。所选应用程序的信息。GitHub上的明星数、提交数、贡献者数、源代码行总数以***及GitHub的开发历史。\*：在Go中编写的gRPC版本。*

## Go应用程序

近年来，围棋的普及率和采用率迅速提高。2017年，Go在GitHub上排名第9[18]。截至本文撰写之时，有187K个GitHub存储库是用Go编写的。

在这项研究中，我们选择了六个用Go编写的具有代表性的真实世界软件，包括两个容器系统（Docker和Kubernetes）、一个键值存储系统（etcd）、两个数据库（CockrachDB和BoltDB）和一个RPC库（gRPC-go1）（表1）。这些应用程序是在数据中心环境中得到广泛使用的开源项目。例如，Docker和Kubernetes是在GitHub上使用Go编写的最受欢迎的应用程序的前2名，分别为48.9K和36.5K星（etcd排名第10，其余排名前100）。我们选择的应用程序都有至少[三年的开](#_bookmark4)发历史，目前主要由开发人员开发。我们选择的所有应用程序都是中型到大型的，代码行从9000到200多万。在这六个应用程序中，Kubernetes和gRPC是谷歌最初开发的项目。

# Go并发使用模式

在研究Go并发bug之前，首先了解真实世界中的Go并发程序是什么样子很重要。本节介绍了我们在选定的六个应用程序中对goroutine用法和Go并发原语用法的静态和动态分析结果。

1除非另有规定，否则我们将使用gRPC来表示下一篇文章中编写的gRPC版本。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **应用** | **正常F。** | **匿名F。** | **全部的** | **KLOC提供** |
| Docker公司 | 33 | 112 | 145 | 0.18 |
| Kubernets公司 | 301 | 233 | 534 | 0.23 |
| 蚀刻 | 86 | 211 | 297 | 0.67 |
| 蟑螂DB | 27 | 125 | 152 | 0.29 |
| gRPC启动 | 14 | 30 | 44 | 0.83 |
| 螺栓DB | 2 | 0 | 2 | 0.22 |
| gRPC-C | 5 | - | 5 | 0.03 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **应用** | **共享内存** | | | | | **消息** | | **全部的** |
| **Mutex公司** | **原子的** | **一旦** | **等待组** | **Cond** | **陈** | **其他。** |
| Docker公司 | 62.62% | 1.06% | 4.75% | 1.70% | 0.99% | 27.87% | 0.99% | 1410 |
| Kubernets公司 | 70.34% | 1.21% | 6.13% | 2.68% | 0.96% | 18.48% | 0.20% | 3951 |
| 蚀刻 | 45.01% | 0.63% | 7.18% | 3.95% | 0.24% | 42.99% | 0 | 2075 |
| 蟑螂DB | 55.90% | 0.49% | 3.76% | 8.57% | 1.48% | 28.23% | 1.57% | 3245 |
| gRPC启动 | 61.20% | 1.15% | 4.20% | 7.00% | 1.65% | 23.03% | 1.78% | 786 |
| 螺栓DB | 70.21% | 2.13% | 0 | 0 | 0 | 23.40% | 4.26% | 47 |

**表2。goroutine/线程创建站点的数量。使用普通函数和匿名函数的goroutine/***线程创建站点的数量，创建站点的总数，以及每千行代码的创建站点。*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **工作量** | **Goroutines/线程** | | **平均执行时间** | |
| **客户** | **服务器** | **客户端Go** | **服务器Go** |
| g\_sync\_ping\_pong | 7.33 | 2.67 | 63.65% | 76.97% |
| 同步\_ping\_pong | 7.33 | 4 | 63.23% | 76.57% |
| qps无约束 | 201.46 | 6.36 | 91.05% | 92.73% |

**表3。执行RPC基准测试时的动态信息。goroutine数除以线程数的比率，以及由整个应用程序的执行时间归一化的平均g***oroutine执行时间。*

## Goroutine用法

为了理解围棋的竞争，我们应该首先了解围棋程序在现实世界中是如何使用的。Go的设计理念之一是使goroutines轻便易用。因此，我们问“真正的Go程序员是否倾向于用许多goroutine（静态）编写代码？”和“真正的围棋应用程序是否在运行时（动态）创建了许多goroutines？”

为了回答第一个问题，我们收集了goroutine创建站点的数量（即创建goroutine的源代码行）。表2总结了结果。总体而言，这六个应用程序使用了大量的goroutine。每千条源线的平均创建站点范围为

0.18至0.83。我们进一步将创建站点分为使用普通函数创建goroutine的站点和使用匿名函数的站点。除了Kubernetes和BoltDB之外，所有应用程序都使用更多的匿名函数。

为了理解Go和传统语言之间的区别，我们还分析了gRPC的另一种实现，即gRPC-C，它是在C/C++中实现的。gRPC-C包含140K行代码，也由Google的gRPC团队维护。与gRPC-Go相比，gRPC-C的线程创建数量惊人地少（每个KLOC只有五个创建站点和0.03个站点）。

我们进一步研究了goroutine的运行时创建。我们运行了gRPCGo和gRPC-C来处理三个性能基准，这些基准旨在比较用不同编程语言编写的多个gRPC版本的性能[22]。这些基准测试使用不同的消息格式、不同数量的连接以及同步和异步RPC请求来配置gRPC。由于gRPC-C比gRPC Go更快[22]，我们运行gRPC-C和gRPC Go来处理相同数量的RPC请求，而不是相同数量的总时间。

**表4。并发原语用法。Mutex列包括Mutex和RWMutex。**

表3显示了[运](#_bookmark7)行这三个工作负载时，gRPCGo中创建的goroutine数与gRPC-C中创建的线程数之比。在客户端和服务器端的不同工作负载上创建了更多的goroutine。表3还展示了我们对goroutine运行时持续时间的研究结果，并将其与gRPC-C的线程运行时持续进行了比较。由于gRPCGo和gRPC-C的总执行时间不同，比较绝对goroutine/线程持续时间没有意义，因此我们报告并比较相对于gRPCGo与gRPC-C[总](#_bookmark7)运行时间的goroutine/thread持续时间。具体来说，我们计算所有goroutine/线程的平均执行时间，并使用程序的总执行时间将其标准化。我们发现gRPC-C中的所有线程都从整个程序的开始到结束执行（即100%），因此表3中只包含了gRPC-Go的结果。对于所有工作负载，goroutine的正常执行时间都比线程短。观察1：Goroutine比C更短，但创建频率更高（静态和运行时）。

## 3.2并发原始用途

在基本了解了goroutine在真实世界围棋程序中的用法之后，我们接下来将研究goroutine如何在这些程序中进行通信和同步。具体来说，我们计算了六个应用程序中不同类型并发原语的使用情况。表4显示了总的（原语使用的绝对数量）以及每种类型的原语占总原语的比例。共享内存同步操作比消息传递更常用，Mutex是所有应用程序中使用最广泛的原语。对于消息传递原语，chan是使用频率最高的一种，从18.48%到42.99%不等。

我们进一步比较了gRPC-C和gRPC-Go中并发原语的用法。gRPC-C只使用锁，它在746个地方使用（每个KLOC使用5.3个原语）。gRPC Go在786个位置使用八种不同类型的原语（每个KLOC使用14.8个原语）。显然，与gRPC-C相比，gRPCGo使用的并发原语数量更大，种类也更多。

接下来，我们将研究并发原语的用法如何随时间变化。图2和图3显示了2015年2月至2018年5月六个应用程序中共享内存和消息传递原语的使用情况。总的[来说](#_bookmark9)，随着时[间的](#_bookmark9)推移，这些用法趋于稳定，这也意味着我们的研究结果将对未来的围棋程序员有价值。

1

docker kubernetes etcd蟑螂grpc−gobo ltdb

使用比例

0.8

0.6

0.4

0.2

0

15-20

15-50

15-08

15-11

16-20

16-05

16-08

16-11

17-20

17~50

17-08

17-11

18-20

18-50

## 图2:共享内存的使用

1

0.8

docker kubernetes etcd蟑螂grpc−gobo ltdb

使用比例

0.6

0.4

0.2

0

15-20

15-50

15-08

15-11

16-20

16-05

16-08

16-11

17-20

17~50

17-08

17-11

18-20

18-50

## Figure 3. Usages of Message-

1

0.8

研究的Bug百分比

0.6

0.4

0.2

0

共享存储器

消息传递

0 100 200 300 400 500 600 700

Bug生存时间（天）

**基本体随时间变化。对于每个应用程序，我们计***算共享内存基元占所有基元的比例。*

**随时间传递基本体。对于每个应用程序，我们计算消息传递原语与***所有原语的比例。*

**图4。Bug生命周期。所有共享内存错误和所有消息***传递错误的生存时间的CDF。*

**观察2：虽然传统的共享内存***线程通信和同步仍然被大量使用，但Go程序员也使用了大量的消息传递原语。*

**含义1：随着goroutin***e和新类型的竞赛原语的大量使用，围棋程序可能会引入更多的竞赛bug。*

# Bug研究方法

本节讨论如何在本研究中收集、分类和再现并发错误。

**正在收集并发错误。为了收集并发错误，我们首先通过搜索六个应**用程序的提交日志中与并发相关的关键字来过滤它们的GitHub提交历史，包括“race”、“deadlock”、“synchroniza-tion”、“concurrency”、“lock”、“mutex”、“atomic”、“competite”、“context”、“once”和“goroutine leak”。这些关键词中的一些在以前的工作中用于收集其他语言中的并发错误[40，42，45]。其中一些与Go引入的新并发原语或库相关，例如“once”和“context”。其中一个“goroutine泄漏”与围棋中的一个特殊问题有关。总共，我们找到了3211个符合搜索条件的不同提交。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **应用** | **行为** | | **原因** | |
| **舞台调度** | **非阻塞** | **共享存储器** | **消息传递** |
| Docker公司 | 21 | 23 | 28 | 16 |
| Kubernets公司 | 17 | 17 | 20 | 14 |
| 蚀刻 | 21 | 16 | 18 | 19 |
| 蟑螂DB | 12 | 16 | 23 | 5 |
| gRPC | 11 | 12 | 12 | 11 |
| 螺栓DB | 3 | 2 | 4 | 1 |
| **全部的** | 85 | 86 | 105 | 66 |

**表5。分类学此表显示了我们研究的***bug是如何在不同类别和应用程序中分布的。*

然后，我们随机抽样过滤后的提交，确定修复并发错误的提交，并手动研究它们。许多与错误相关的提交日志也提到了相应的错误报告，我们还研究了这些报告以进行错误分析。我们总共研究了171个并发错误。Bug分类。我们提出了一种根据两个正交维度对Go并发bug进行分类的新方法。

第一个维度是基于bug的行为。如果一个或多个gorouti*ne在执行过程中*被无意地卡住而无法前进，我们称这种并发问题为阻塞错误。相反，如果所有的goroutine都可以完成任务，但它们的行为不被期望，我们称它们为非阻塞的。以前的大多数并发bug研究[24，43，45]将bug分为死锁bug和非死锁bug*，其中死锁包括跨多个线程循*环等待的情况。我们对阻塞的定义比死锁更广泛，包括没有循环等待但一个（或多个）goroutine等待其他goroutine无法提供的资源的情况。正如我们将在第5节中所展*示的，相当多的Go并发b*ug都属于这种类型。我们认为，随着像Go这样的新语言的新编程习惯和语义，我们应[该更](#_bookmark52)加[关注](#_bookmark70)这[些非](#_bookmark72)死锁阻塞错误，并扩展传统的并发错误分类机制。第二个维度是并发错误的原因。当多个线程尝试通信时，并发错误就会发生，并且在通信过程中会发生错误。因此，我们的想法是通过不同的goroutine如何通信来分类并发错误的原因：通过访问共享内存或传递消息。这种分类可以帮助程序员和研究人员选择更好的方式来执行线程间通信，并检测和避免

在执行这种通信时可能发生错误。

根据我们的分类方法，共有85个阻塞错误和86个非阻塞错误，共有105个错误是由错误的共享内存保护引起的，66个错误是由于错误的消息传递引起的。表5显示了每个应用程序中错误类别的详细细分。

我们进一步分析了我们研究的bug的生存时间，即从将bug代码添加（提交）到软件到在软件中修复（提交bug修复补丁）的时间。如图4所示，我们研究的大多数bug（共享内存和消息传递）都有很长的生存时间。我们还发现，报告这些错误的时间接近修复它们的时间。这些结果表明，我们研究的大多数bug都不容易

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **应用** | **共享内存** | | | **消息传递** | | |
| **Mutex公司** | **RWMutex公司** | **等待** | **陈** | **陈w/** | **伦敦银行同业拆借利率** |
| Docker公司 | 9 | 0 | 3 | 5 | 2 | 2 |
| Kubernets公司 | 6 | 2 | 0 | 3 | 6 | 0 |
| 蚀刻 | 5 | 0 | 0 | 10 | 5 | 1 |
| 蟑螂DB | 4 | 3 | 0 | 5 | 0 | 0 |
| gRPC | 2 | 0 | 0 | 6 | 2 | 1 |
| 螺栓DB | 2 | 0 | 0 | 0 | 1 | 0 |
| **全部的** | 28 | 5 | 3 | 29 | 16 | 4 |

**表6。阻塞错误原因。Wait包括Cond和WaitGroup***中的Wait函数。Chan表示频道操作，Chan表示其他操作的频道操作。L*ib代表*与消息传递*相关的Go库。

被触发或检测到，但一旦被触发，它们很快就被修复了。因此，我们认为这些bug是非常重要的，值得仔细研究。

**播放并发错误。为了评估内置的死锁和数据竞争检测技术，我们复制**了21个阻塞错误和20个非阻塞错误。为了重现bug，我们将应用程序回滚到bug版本，构建bug版本，并使用bug报告中描述的bug触发输入运行构建的程序。我们利用错误报告中提到的症状来决定是否成功复制了错误。由于其非确定性，并发错误很难再现。有时，我们需要多次运行一个有问题的程序，或者手动为一个有错误的程序添加睡眠。对于未再现的bug，这要么是因为我们没有找到一些依赖库，要么是因为没有观察到所描述的症状。

**对有效性的威胁。对我们研究有效性的威胁可能**来自多个方面。我们选择了六个具有代表性的围棋应用程序。在Go中实现了许多其他应用程序，它们可能没有相同的货币问题。我们只研究了已修复的并发错误。可能有其他并发bug很少被复制，开发人员也从未修复过。对于一些固定的并发错误，提供的信息太少，难以理解。我们的研究中没有包含这些错误。尽管有这些限制，我们还是尽了最大努力收集真实世界中的Go并发错误，并进行了全面而公正的研究。我们相信，我们的研究结果足够普遍，足以激励和指导未来对抗Go并发bug的研究。

# 阻止Bugs

本节介绍了我们对阻塞错误的研究结果，包括其根本原因、修复方法以及内置运行时Go死锁检测器在检测阻塞情况方面的有效性。

## 阻塞Bug的根本原因

当一个或多个gorroutine执行等待资源和这些资源的操作时，阻塞错误就会出现

永远不可用。要检测并避免阻塞错误，了解其根本原因很重要。我们通过检查哪个操作阻止了一个goroutine，以及为什么其他goroutine没有阻止该操作，来研究阻塞bug的根本原因。使用错误分类的第二个维度，我们将阻塞错误分为旨在保护共享内存访问的阻塞操作和消息传递操作导致的阻塞错误。表6总结了所有阻塞错误的根本原因。

总的来说，我们发现大约42%的阻塞错误是由保护共享内存的错误引起的，58%是由消息传递错误引起的。考虑到共享内存原语的使用频率比消息传递原语更高（第3.2节），消息传递操作更可能导致阻塞错误。

**观察3：与普遍认为消息传递***不那么容易出错的观点相反，我们研究的Go应用程序中更多的阻塞错误是由错误的消息传递引起的，而不是由错误的共享内存保护引起的。*

## （mis）共享内存的保护

众所周知，共享内存访问很难正确编程，一直是死锁研究的主要焦点之一[35，51，54]。它们继续导致围棋中的阻塞bug，既有传统模式，也有新的围棋特定原因。

*Mutex*  28阻塞错误是由误用锁（Mutex）引起的，包括双重锁定、在冲突订单中获取锁以及忘记解锁。这一类别中的所有bug都是传统的bug，我们认为传统的死锁检测算法应该能够通过静态程序分析来检测这些bug。

*RWMutex*如第2.2节所述，Go的写锁定请求比读锁定请求具[有更高的权](#_bookmark3)限。当一个goroutine（th-a）通过读锁定两次获取一个RWMutex，并且这两个读锁定操作被另一个gorroutine（th-B）的写锁定操作交织时，这种独特的锁实现可能会导致阻塞错误。当th-A的第一次读锁定操作成功时，它将阻止th-B的写锁定操作，因为写锁定是互斥的。然而，th-B的写锁定操作也会*阻止th-A的*第二次读锁定操作，因为写锁定请求在Go的实现中具有更高的权限。A和B都无法继续。

五个阻塞错误是由这个原因引起的。注意，由于pthread\_rwlock\_t在默认设置下优先处理读锁请求，相同的交错锁定模式不会导致C中pthread\_rwlock\_t的阻塞错误。RWMutex阻塞错误类型意味着，即使Go使用与传统语言相同的并发语义，由于Go对语义的新实现，仍然可能存在新类型的错误。

1. **var**组同步.WaitGroup
2. group.Add（**len**（pm.plugins））
3. **对于\_**，p:=范**围pm.p**lugins{
4. go  **fun**c（p\*插件）{
5. **defer**-group.Done（）

6 }

7-gr oup.Wait（）

8 }

9+group.Wait（）

## 图5。WaitGroup导致的阻塞错误。

*等待三个*阻塞错误是由于等待操作无法继续。与Mutex和RWMutex相关的bug不同，它们不涉及循环等待。当Cond用于保护共享内存访*问并且一个*gor*outine调*用Cond.Wait（）时，会出现其中两个错误，但之后没有其他goroutine会调用Cond.Signal（）（或Cond.Broadcast（））。

第三个bug，Docker#25384，发生在使用WaitGroup类型的共享变量时，如图5所示。第7行的Wait（）只能在第5行的Done（）被调用len（pm.plugins）次时[解除阻](#_bookmark13)止，因为

len（pm.plugins）用作第行调用Add（）的参数

-hctx，hcancel:=上下文。WithCancel（ctx）

2+**var**  hctx context.context

3+**var**  hcancel context.CancelFunc

1. **如果**超时>0{
2. hctx，hcancel=context.WithTimeout（ctx，超时）

6+}**其他{**

7+hc tx，hcancel=context.WithCancel（ctx）

8 }

## 图6。上下文导致的阻塞错误。

对象hcancel在第1行创建。同时创建一个新的goroutine，消息可以通过hcancel的channel字段发送到新的gorroutine。如果第4行的超时值大于0，则在第5行创建另一个上下文对象，并且hcancel指向新对象。之后，就无法向或关闭附加到旧对象的goroutine发送消息。补丁是为了避免在超时大于0时创建额外的上下文对象。

* 1. **平方函数**
  2. m、 锁定（）

2.然而，Wait（）在循环中被调用，因此它在以后的迭代中阻止了第4行的goroutine创建，并阻止了在每个创建的gorou-tine中调用Done（）。修复此错误的方法是将Wait（）的调用从循环中移出。

* 1. - ch<-请求//块

4+选择 **{**

5+ca **se c**h<-请求

6+默认 **值：**

7 + }

8 m.解锁（）

9 }

1. **函数go**routine2（）{
2. **用于{**
3. m、 Lock（ ）//块
4. m、 解锁（）
5. 请求<-ch

6 }

7 }

尽管条件变量和线程组等待都是传统的并发技术，但我们怀疑Go的新编程模型是程序员产生这些并发错误的原因之一。例如，与pthread\_join不同，它是一个显式等待（命名）线程完成的函数调用，WaitGroup是一个可以在goroutine之间共享的变量，它的Wait函数隐式等待Done函数。

**观察4：大多数由共享内存同步***引起的阻塞错误的原因和修复程序与传统语言相同。然而，其中一些与传统语言不同，要么是因为Go对现有原语的新Implementation或其新的编程语义。*

## 信息传递的误用

我们现在讨论由消息传递错误引起的阻塞错误，这与我们所研究的应用程序中阻塞错误的主要类型相反。

*使用通道在go*rroutine之间传递消息时出错导致29个阻塞错误。许多与信道相关的阻塞错误是由于缺少发送到信道（或从信道接收）或关闭信道导致的，这将导致阻塞等待从信道接收（或发送到信道）的goroutine。图1就是一个这样的例子。当结合Go特殊库的使用时，通道创建和goroutine阻塞可能隐藏在库调用中。如图6所示

* + - 1. 程序1（b）程序2

## 图7。错误使用带有锁的通道导致的阻塞错误。

*通道和其他阻塞原语对于16个阻塞错误，一个goroutine在通道*操作时被阻塞，另一个gorroutine在锁定或等待时被阻塞。例如，如图7所示，goroutine1在向通道ch发送请求时被阻止，而goroutine2在m.Lock（）时被阻止

修复方法是为goroutine1添加一个带有默认分支的select，以使ch不再阻塞。

*消息传递库Go提供了几个传递数据或消*息的库，如Pipe。如果不正确使用，这些特殊的库调用也会导致阻塞错误。例如，与通道类似，如果管道未关闭，则当goroutine试图向未关闭的管道发送数据或从中提取数据时，它可能会被阻止。有4个收集到的阻塞错误是由特殊的Go消息传递库调用引起的。

**观察5：由消息传递引起的所***有阻塞错误都与Go的新消息传递语义（如Channel）有关。它们可能很难检测，特别是当消息传递操作与其他同步机制一起使用时。*

**含义2：与通常的看法相反，***消息传递比共享内存会导致更多的阻塞错误。我们呼吁注意消息传递编程中的潜在危险，并提出了这一领域的错误检测研究问题。*

|  |  |  |
| --- | --- | --- |
| **根本原因** | **#已用Bug数** | **#检测到的错误** |
| Mutex公司 | 7 | 1 |
| 陈 | 8 | 0 |
| 陈w/ | 4 | 1 |
| 消息传递库 | 2 | 0 |
| **全部的** | 21 | 2 |

**表7。修复阻止错误的策略。下标s**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **添加** | | **移动** | **变化** | **删除** | **其他。** |
| **共享内存**  Mutex等待RWMutex | 9  0  0 | 7  1  2 | 2  0  0 | 8  1  3 | 2  1  0 |
| **消息传递**  陈陈w/  消息库 | 15  6  1 | 1  3  0 | 5  2  0 | 4  4  1 | 4  1  2 |
| **全部的** | 31 | 14 | 9 | 21 | 10 |

*表示同步。*

## 修复阻塞错误

在了解了围棋中阻塞bug的原因之后，我们现在分析了围棋程序如何在现实世界中修复这些bug。

消除挂起的goroutine的阻塞原因将解除阻塞，这是修复阻塞错误的一般方法。为了实现这一目标，Go开发人员经常调整同步操作，包括添加丢失的同步操作、移动或更改错误/误用的同步操作，以及删除多余的同步操作。表7总结了这些修复。

由于错误地保护共享内存访问而导致的大多数阻塞错误都是通过类似于传统死锁修复的方法修复的。例如，在33个Mutex或RWMutex相关的bug中，有8个通过添加缺失的解锁被修复；9个通过将锁定或解锁操作移动到适当位置而固定；并且11通过移除额外的锁定操作而被固定。

通过在与阻塞程序不同的goroutine上向通道（以及两次向管道）添加丢失的消息或关闭操作，修复了由错误消息传递导致的11个阻塞错误。通过添加带有默认选项的select（如图7）或在不同频道上操作的case，修复了8个阻塞错误。与通道相关的阻塞错误的另一个常见修复方法是用缓冲通道替换未缓冲的通道（例如，图1）。其他与通道相关的阻塞错误可以通过策略解决，例如将通道操作移出关键部分，并用共享变量替换通道。

为了理解阻塞bug的原因与其修复之间的关系，我们应用了一种称为提升的统计度量，之前对真实世界bug进行了实证研究[29，41]。升力计算为升力（A，B）=P（AB），其中A表示根本原因类别，B表*示修复策*略类别，P（AB）表示阻塞由A引起并由B修复的概率。当升力值等于1时，A根本原因与B修复策略无关。当升力[值大](#_bookmark57)于1[时，](#_bookmark68)A和B*正相关，这*意味着如果阻塞由A引起，则更可能由B固定。当升力值小于1*时*，*B和*A负相关。在所有*具*有超过10个阻塞错误的错误类别中（我们忽略了具有少于*1*0个错误的类别，因为它们的统计意义很小），Mutex是与修*复*类*型相*关最强的类别，它与提升值为1.52的Moves相关。相关性

*P*（*A*）*P*（*B*）

## 表8。死锁检测器的基准测试和评估结果。

Chan和Ad*ds之间*是第二*高的，具*有升力值

1.42.具有10个以上阻塞错误的所有其他类别的提升值都低于1.16，没有显示出强相关性。

我们还根据补丁中使用的并发原语类型分析了阻塞bug的修复。正如预期的那样，大多数原因与某种类型的基元有关的bug也通过调整该基元来修复。例如，通过调整Mutex原语修复了所有与Mutex相关的错误。

错误原因与修复错误所用的原语和策略之间的高度相关性，加上围棋中同步原语的类型有限，这表明在研究围棋中阻止错误的自动纠正方面取得了丰硕的成果。我们进一步发现，我们研究的阻塞bug的补丁大小很小，平均有6.8行代码。大约90%的研究阻塞错误是通过调整同步原语来修复的。

**观察6：我们研究中的大多数***阻塞性错误（传统共享内存和消息传递）都可以用简单的解决方案修复，许多修复程序与错误原因有关。*

**含义3：围棋阻塞bug的原***因和修复之间的高度相关性及其修复的简单性表明，开发完全自动化或半自动化的工具来修复围棋中的阻塞bug是有希望的。*

## 检测阻塞错误

Go提供了一个内置的死锁检测器，它在goroutine调度程序中实现。在Go运行时，检测器始终处于启用状态，当运行进程中的任何错误都无法取得进展时，它会报告死锁。我们用Go内置的死锁检测器测试了所有复制的阻塞错误，以评估它能找到什么错误。对于每一个测试过的bug，阻塞都可以在每次运行中被确定地触发。因此，对于每个bug，我们在这个实验中只运行了一次。表8总结了我们的测试结果。

内置死锁检测器只能检测两个阻塞错误，BoltDB#392和BoltDB#240，并且在所有其他情况下都会失败（尽管检测器不会报告任何错误位置[38，39]）。内置检测器未能检测到其他阻塞错误的原因有两个。首先，当仍有一些运行的goroutine时，它不会将被监控的系统视为阻塞。其次，它只检查goroutine是否在Go并发原语中被阻止，但不考虑等待其他系统的g[or](#_bookmark64)o[ut](#_bookmark65)ine

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **应用** | **共享内存** | | | | **消息传递** | |
| **传统的** | **匿名者。** | **等待组** | **利布** | **陈** | **利布** |
| Docker公司 | 9 | 6 | 0 | 1 | 6 | 1 |
| Kubernets公司 | 8 | 3 | 1 | 0 | 5 | 0 |
| 蚀刻 | 9 | 0 | 2 | 2 | 3 | 0 |
| 蟑螂DB | 10 | 1 | 3 | 2 | 0 | 0 |
| gRPC | 8 | 1 | 0 | 1 | 2 | 0 |
| 螺栓DB | 2 | 0 | 0 | 0 | 0 | 0 |
| **全部的** | 46 | 11 | 6 | 6 | 16 | 1 |

**表9。非阻塞错误的根本原因。传统：传统的非阻塞bug；匿名函数：匿名函数导致的非阻塞错***误；waitgroup：误用waitgroup；lib:Go库；陈：滥用频道。*

资源。这两个限制很大程度上是由于内置检测器的设计目标是最小化运行时开销。

1 **对于i**:=17；i<=21；i++{//写入

2-go f**unc（**）｛/\*创建新的goroutine\*/

3+go 函数**（i i**nt**）{**

4 apiVersion:=fmt.Sprintf（“v1.%d”，i）//读取

5 ...

6 - }()

7+}（ i）

8 }

## 图8。匿名函数导致的数据竞争。

1. **发送（）**{
2. p、 μLock（）
3. **defer**  p.mu.Unlock（）
4. **开关p.st**atus{
5. **案例空闲**：

6+p. wg.添加（1）

1. go**函数（）**{

当在运行时调度器中实现时，检测器很难在不牺牲性能的情况下有效地识别复杂的阻塞错误。

**含义4：简单运行时死锁检测***器不是ef-*

*有效检测Go阻塞错误。未来的研究应该*

1. - p.wg.Add（1）

9 ...

10 p.wg.Done()

11 }()

12 **机箱停止**：

13 }

14 }

1. **func**（p\*对等）stop（）{
2. p、 μLock（）
3. p、 状态=已停止
4. p、 μ解锁（）
5. P.wg.wait

6 }

*专注于构建新的阻塞错误检测技术，例如，结合静态和动态阻塞模式检测。*

# 非阻塞错误

本节介绍了我们对非阻塞bug的研究。与我们在第5节中所做的类似，我们研究了非阻塞bug的根本原因和修复方法，并评估了围棋的内置种族检测器。

## 非阻塞错误的根本原因

与阻塞错误类似，我们还将收集的非阻塞错误分类为未能保护共享内存导致的错误和消息传递错误（表9）。

## 无法保护共享内存

先前的工作[8、14、16、1[7](#_bookmark36)、4[6、](#_bookmark42)47[、5](#_bookmark44)2、[62](#_bookmark45)–6[4]](#_bookmark73)发现[，不](#_bookmark74)保护[共享](#_bookmark79)内存[访问](#_bookmark89)或[此类](#_bookmark91)保护中的错误是数据竞争和其他非死锁错误的主要原因。类似地，我们发现大约80%的收集到的非阻塞错误是由于未受保护或错误保护的共享内存访问。然而，并不是所有的错误都与传统语言中的非阻塞错误有相同的原因。传统错误我们收集到的非阻塞错误中有一半以上是由传统问题引起的，这些问题也发生在C和Java等经典语言中，例如原子性违反[8、16、46]、顺序违反[17、47、62、64]和数据竞争[14、52、63]。这一结果表明，不同语言的开发人员也会犯同样的错误。它还表明，有希望应用现有的并发错误检测算法来查找Go中的新错误。

有趣的是，我们发现了七个非阻塞错误，其根本原因是传统的，但主要是由于对新的Go功能缺乏清晰的理解。例如

* + - 1. 功能1（b ）功能2

## 图9。误用导致的非阻塞错误

**等待组。**

Docker#22985和CockrocheDB#6111是由共享变量上的数据竞争引起的，该共享变量的引用通过通道跨goroutine传递。

*匿名函数Go设计者使goroutin*e声明类似于常规函数调用（甚至不需要有“函数名”），从而简化goroutine的创建。匿名函数可以访问Go匿名函数之前声明的所有局部变量。不幸的是，当使用匿名函数创建goroutine时，这种编程的简单性可能会增加数据竞争错误的机会，因为开发人员可能没有足够注意保护这样的共享局部变量。

我们发现了11个这种类型的bug，其中9个是由父goroutine和使用匿名函数创建的子goroutine之间的数据竞争引起的。另外两个是由两个子goroutine之间的数据竞赛引起的。Docker的一个示例如图8所示。局部变量i在父goroutine和它在第2行创建的goroutine之间共享。开发人员希望每个子goroutine使用一个不同的i值来初始化第4行的字符串apiVersion。然而，在bug程序中，apiVersion的值是不确定的。例如，如果子goroutine在父goroutine的整个循环完成后开始[，则a](#_bookmark20)piVersion的值都等于“v1.21”。只有当每个子goroutie在创建后立即初始化字符串apiVersion，并且在i被分配给新值之前，bug程序才会产生所需的结果。Docker开发人员通过在每次迭代时复制共享变量i并将复制的值传递给新的goroutine来修复这个bug。

*滥用WaitGroup使用WaitGr*oup时有一个基本规则，即必须在Wait之前调用Add。违反此规则会导致6个非阻塞错误。图9

在etcd中显示了一个这样的错误，其中无法保证在func1第8行的Add发生在func2第5行的Wait之前。修复方法是将Add移动到一个关键部分，这确保Add要么在Wait之前执行，要么不执行。

*特殊库Go提供了许多新的库，其中一些*库使用由多个gor-outine隐式共享的对象。如果没有正确使用，数据竞争可能会发生。例如，上下文对象类型被设计为可由附加到上下文的*多个goroutine*访问。etcd#7816是由多个goroutine访问上下文对象的字符串字段导致的数据竞争错误。

另一个例子是测试包-

签名支持自动测试。测试函数（通过以“Test”开头的函数名来标识）只接受一个Test.T类型的参数，用于传递错误和日志等测试状态。从运行测试函数的goroutine和在测试函数内创建的其他goroutine访问test.T变量会导致三个数据竞争错误。

**观察7：大约三分之二的共享内***存非阻塞性错误是由传统原因引起的。Go的新多线程语义和新图书馆为其他三分之一做出了贡献。*

**含义5：Go为简化多线程编***程而引入的新编程模型和新库本身可能是导致更多竞争bug的原因。*

## 消息传递过程中的错误

消息传递过程中的错误也会导致非阻塞错误，它们占我们收集的非阻塞错误的20%左右。

*滥用信道正如我们在第2节中所讨*论的，使用信道时有几个规则，违反这些规则除了会导致阻[塞错](#_bookmark2)误之外，还会导致非阻塞错误。有16个非阻塞错误是由通道的误用引起的。

1. -**选择{**
2. - **案例<-**c.关闭：
3. - **默认值：**

4+On ce.Do（函数**）{**

5 **关闭（c.**关闭）

6 + })

7 - }

## 图10。关闭频道两次导致的错误。

例如，图10中的Docker#24007是因为违反了通道只能关闭一次的规则。[当多个](#_bookmark23)goroutine执行这段代码时，其中多个gorroutine可以执行默认子句并尝试关闭第5行的通道，从而导致Go中的运行时死机。修复方法是使用Once包强制信道只关闭一次。

当同时使用channel和select时，会发生另一种并发错误。在Go中，当选定对象收到多条消息时，无法保证哪一条会

1. ticker:=time.NewTicker（）
2. **用于{**

3+选择 **{**

4+案例 **<-st**opCh：

5+返回

6+默认 **值：**

7 + }

8 f（）

1. **选择{**
2. **case**<-stopCh：
3. **回来**
4. **大小写<**-代码：

13 }

14 }

## 图11。选择和频道导致的非阻塞错误。

1-计时器：=time.NewTimer（0）

2+**var**超时<-chan t**ime.**time

1. **如果**dur>0{
2. - timer=time.NewTimer（dur）

5+ti meout=time.NewTimer（dur）.C

6 }

1. **选择{**
2. - **case**<-计时器.C:

9+案例 **<-超时**：

1. **case**<-ctx.Done（）：
2. **返回零**

12 }

## 图12。计时器导致的非阻塞错误。

首先处理。select的这种非确定性实现导致了3个错误。图11显示了一个这样的示例。第2行的循环在第8行执行一个重函数f（），每当在第12行（情况2）出现报时。如果从stop[Ch接](#_bookmark24)收到一条消息，并且自动收报机同时发出滴答声，则无法保证select会选择哪一条。如果select选择情况2，f（）将不必要地再执行一次。修复方法是在循环开始时添加另一个select，以处理来自stopCh的未处理信号。

*特殊库Go的一些特殊库以微妙的方式使*用通道，这也会导致非阻塞性错误。图12显示了一个与设计用于测量时间的时间包相关的bug。这里，在第1行创建了超时持续时间为0的计时器。在创建Timer对象时，Go运行时（隐式）启动一个库内部goroutine，它[启动计](#_bookmark25)时器倒计时。计时器在第4行设置超时值dur。这里的开发人员

仅当dur大于0或ctx.Done（）时才从当前函数返回。但是，当dur不大于0时，库内部goroutine将发出信号

计时器创建后，立即启动timer.C通道，导致函数过早返回（第8行）。修复方法是避免在第1行创建计时器。

**观察8：通过消息传递而不是共***享内存访问造成的非阻塞性错误要少得多。渠道规则和与其他渠道使用渠道的复杂性*

|  |  |  |
| --- | --- | --- |
| **根本原因** | **#已用Bug数** | **#检测到的错误** |
| 传统Bugs | 13 | 7 |
| 匿名函数 | 4 | 3 |
| 伦敦银行同业拆借利率 | 2 | 0 |
| 滥用频道 | 1 | 0 |
| **全部的** | 20 | 10 |

**表10。修复非阻塞错误的策略。子脚本s代表同步。**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **计时**  **添加移***动* | | | **指示**  **旁路** | **数据**  **私有的** | **其他。** |
| **共享内存传统等待组匿名**  利布 | 27  3  5  1 | 4  2  2  2 | 5  1  0  1 | 10  0  4  0 | 0  0  0  2 |
| **消息传递**  chan库 | 6  0 | 7  0 | 3  0 | 0  0 | 0  1 |
| **全部的** | 42 | 17 | 10 | 14 | 3 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mutex信道原子等待组条件其他。没有一个** | | | | | | | |
| **共享内存传统等待组匿名**  利布 | 24  2  3  0 | 3  0  2  2 | 6  0  3  1 | 0  4  0  1 | 0  3  0  0 | 0  0  0  1 | 13  0  3  2 |
| **消息传递**  chan库 | 3  0 | 11  1 | 0  0 | 2  0 | 1  0 | 2  0 | 1  0 |
| **全部的** | 32 | 19 | 10 | 7 | 4 | 3 | 19 |

## 表11。非阻塞bug补丁中的同步原语。

*Go特定的语义和库是这些非阻塞错误发生的原因。*

**含义6：如果正确使用，消息传***递比共享内存处理更不容易出现非阻塞错误。然而，在一种语言中传递消息的复杂设计可能会导致这些bug在与其他特定于语言的功能结合时特别难以发现。*

## 非阻塞错误的修复

与我们对阻塞bug修复的分析类似，我们首先通过非阻塞bug的策略分析它们的修复。表10对我们研究的Go非阻塞bug的修复策略进行了分类，其方式与之前C/C++中非阻塞bug修复的分类类似[43]。

大约69%的非阻塞错误是通过重新限制时间来修复的，要么通过添加同步原语（如Mutex），要么通过移动现有原语（如图9中的Add）。通过消除访问共享变量的指令或绕过指令，修复了10个非阻塞错误（例如，图10）。通过创建共享变量的私有副本修复了14个错误（例如，图8），这些错误都是共享内存错误。

为了更好地理解非阻塞性错误修复及其与错误原因的关系，我们进一步检查补丁中使用了哪些原语。表11根据补丁中使用的原语类型列出了修复。与先前对C/C++中的货币错误补丁的研究结果类似[43]，互斥锁是最广泛使用的原语，用于强制互斥和修复非阻塞错误。除了传统的bug之外，互斥锁还用于修复匿名函数和WaitGrou[p和](#_bookmark28)

以替换误用的通道。

作为一种新的原语，通道是第二种使用最广泛的。利用通道在两个通道之间传递值

**表12。数据竞赛检测器的基准和评估结果。我们将在100次运行中检测到的错误视为检测到的缺陷。**

gorroutine和替换共享变量以修复数据竞争。它还用于在不同的goroutine中强制执行两个操作之间的顺序。还有一些错误，通道没有正确使用，并且在补丁中被修复（例如，图10）。有趣的是，通道不仅用于修复消息传递错误，而且还用于修复传统共享内存同步导致的错误。我们怀疑这是因为一些Go程序员认为消息传递是执行线程间通信的更可靠或更容易编程的方式-

通信而不是共享内存同步。

最后，其他并发原语修复了24个错误，19个错误在不使用任何并发原语的情况下被修复（例如，图8）。

与第5.2节中的提升分析类似，*我们计算了*非阻塞bug的原因和修复策略之间以及原因[和修复原语](#_bookmark18)之间的提升。在超过10个错*误的错误*类别中，原因误用通道和修复原始通道之间的相关性最强，提升值为2.7。原因匿名函数和修复策略数据私有具有第二高的提升值2.23。其次，误用通道与提升值为2.21的移动高度相关。

**观察9：传统的共享内存同步技***术仍然是Go中非阻塞错误的主要修复程序，而Channel被广泛用于修复不仅与通道相关的错误，而且与共享内存错误有关的错误。*

**含义7：尽管Go程序员继续使***用传统的共享内存保护机制来修复非阻塞性错误，但在某些情况下，他们更喜欢使用消息传递作为修复，这可能是因为他们认为消息传递是跨线程通信的一种更安全的方式。*

## 非阻塞错误的检测

Go提供了一个数据竞争检测器，它使用与ThreadAnitizer相同的先发生后发生算法[53]。它可以通过使用“race”标志构建程序来启用。在程序执行期间，竞争检测器为每个内存对象创建最多四[个影](#_bookmark80)子字，以存储对象的历史访问。它将每个新访问与存储的阴影字值进行比较，以检测可能的种族。

我们使用我们的20个复制的非阻塞错误来评估检测器可以检测到多少错误。我们在打开种族检测器的情况下运行每个bug程序100次。表12总结了在每个根本原因类别下检测到的bug数量。检测器没有误报。

数据竞争检测器成功检测到7/13个传统错误和3/4个由匿名函数引起的错误。对于其中的六次成功，数据竞赛检测器在每次运行中都报告了错误，而对于其余的四次，在检测器报告错误之前，大约需要运行100次。

数据竞赛检测器未能报告许多非阻塞错误的原因有三种。首先，并非所有的非阻塞bug都是数据竞赛；种族检测器不是为检测这些其他类型而设计的。其次，底层算法的有效性取决于并发goroutine的交织。最后，由于每个内存对象只有四个影子字，检测器无法保持长时间的历史，并且可能错过数据竞赛。

**含义8：简单的传统数据竞赛检***测器无法有效检测所有类型的Go非阻塞错误。未来的研究可以利用我们的错误分析来开发更多信息，Go特定的非阻塞错误检测器。*

# 讨论和未来工作

Go提倡让线程创建变得简单和轻量级，并使用共享内存上的消息传递进行线程间通信。事实上，我们看到在Go程序中创建的gor例程比传统线程多，Go通道和其他mes sage传递机制有着重要的用途。然而，我们的研究表明，如果不正确使用，这两种编程实践可能会导致并发错误。

**共享内存与消息传递。我们的研究发现，消息传递并不一定会使多线程程序比共**享内存更容易出错。事实上，消息传递是阻塞错误的主要原因。更糟糕的是，当与传统的同步原语或其他新的语言特性和库相结合时，消息传递可能会导致难以检测的阻塞错误。与共享内存同步相比，消息传递导致的非阻塞性错误更少，而且令人惊讶的是，它甚至被用于修复错误共享内存同步导致的错误。我们相信消息传递提供了一种干净的线程间通信形式，在传递数据和信号时非常有用。但它们只有在正确使用时才有用，这要求程序员不仅要很好地理解消息传递机制，还要理解Go的其他同步机制。

**对错误检测的影响。我们的研究揭示了许多错误代码模式，这些模式**可以用来进行货币错误检测。作为初步工作，我们构建了一个针对匿名函数导致的非阻塞错误的检测器（例如图8）。我们的检测器已经发现了一些新的bug，其中一个已经被真正的应用程序开发人员证实[12]。

更一般地说，我们认为静态分析加上先前的死锁检测算法在检测由共享内存同步错误导致的大多数Go阻塞错误时仍然有用。静态技术也有帮助

在检测由通道和锁的组合导致的错误时，如图7所示。

误用Go库会导致阻塞和非阻塞错误。在我们的研究中，我们总结了关于错误使用Go库的几种模式。检测器可以利用我们学习到的模式来揭示以前未知的错误。

我们的研究还发现，违反Go使用其并发原语强制执行的规则是导致货币错误的一个主要原因。一种新的动态技术可以尝试强制执行这些规则，并在运行时检测违规行为。

# 相关工程

**研究真实世界的臭虫。有许多关于真实世界bug的实证研**究[9，24，25，29，40，44，45]。这些研究成功地指导了各种防bug技术的设计。据我们所知，我[们](#_bookmark37)的工[作是](#_bookmark52)第一[次专](#_bookmark53)注于[Go](#_bookmark57)中并[发错](#_bookmark67)误的[研究](#_bookmark71)，也[是第](#_bookmark72)一次比较访问共享内存时的错误和传递消息时的错误导致的错误。

**打击堵塞漏洞。作为一个传统问题，有许多研究工**作在C和Java中解决死锁[7，28，33–35，51，54，55，58，59]。虽然有用，但我们的研究表明，围棋中存在许多非死锁阻塞错误，这些错误不是这些技术的目标。提[出](#_bookmark35)了一[些技](#_bookmark56)术来[检测](#_bookmark60)误[用信](#_bookmark61)道导[致的](#_bookmark78)阻塞[错误](#_bookmark81)[3[8，](#_bookmark82)39[，4](#_bookmark85)9，[56](#_bookmark86)]。然而，阻塞错误可能由其他原语引起。我们的研究揭示了许多用于阻止错误的代码模式，这些模式可以作为未来阻止错误检测技术的基础。

**打击非阻塞漏洞。许多先前的研究工作都是为了检测、诊断和修复**由于未能同步共享内存进程而导致的非死锁错误[4、5、8、14、16、17、30–32、43、46、47、52、62–64]。他们

有望应用于围棋比赛中的bug。然而，我们的研究发现，由于消息传递过程中的错误而导致的非阻塞性错误中有一部分是不可忽略的，而这些错误在以前的工作中并没有涵盖。我们的研究强调需要新技术来对抗消息传递过程中的错误。

# 结论

作为一种为并发设计的编程语言，Go提供了轻量级的goroutine和goroutine之间基于通道的消息传递。面对Go在各种类型应用程序中日益增多的使用，本文从两个正交维度对171个真实世界的Go并发bug进行了首次全面的实证研究。我们的研究提供了许多有趣的发现和启示。我们希望我们的研究能够加深对Go并发bug的理解，并引起对Go并发Bug的更多关注。

# 工具书类

1. 使用channels.URL设计Go API的原则：https://inconshreveable.co m/07-08-2014/principles-of-designing-go-带有通道/的api。
2. Go博客：通过交流分享记忆。网址：https://blog.golang.org/share-me mory-by-communicating.
3. Ajmani先生。高级Go并发模式。网址：https://talks.golang.org/2013/advconc.slide.
4. Joy Arulraj、Po Chun Chang、Guo Liang Jin和Shan Lu。通过硬件性能计数器进行生产运行软件故障诊断。2013年3月，美国德克萨斯州休斯顿，第18届编程语言和操作系统体系结构支持国际会议（ASPLOS*’13）论文集。*
5. Joy Arulraj、Guoliang Jin和Shan Lu。利用硬件的短期内存诊断生产运行软件故障。2014年3月，美国犹他州盐湖城，第19届编程语言和操作系统体系结构支持国际会议（ASPLOS‘14）论文集。
6. 螺栓db。Go的嵌入式键/值数据库。网址：https://github.com/boltdb/bolt.
7. 蔡燕和陈伟凯。Magiclock：大规模多线程程序中潜在死锁的可扩展检测。IEEE软件工程学报，40（3）：266-2812014。
8. 李周和李大卫。Kivati：快速检测和预防原子性违规。《第五届欧洲计算机系统会议论文集》（EuroSys’10），法国巴黎，2010年4月。
9. 周安迪、杨俊峰、本杰明·切尔夫、赛斯·哈勒姆和道森·恩格勒。操作系统错误的实证研究。2001年10月，在加拿大阿尔伯塔省班夫举行的第18届ACM操作系统原理研讨会（SOSP'01）上。
10. 蟑螂。CockrochDB是一个云原生SQL数据库，用于构建能够抵御灾难的全球可扩展云服务。网址：https://github.com/cockroachdb/cockroach.
11. Russ Cox.贝尔 实验室和CSP威胁.URL：http://swtch.com/rsc /thr[ead/。](http://swtch.com/)
12. Graphql开发人员。将错误附加到Result.errors中的线程安全方法。URL:https://github.com/graphql-go/graphql/pull/434.
13. Docker公司。Docker-随时随地构建、发布和运行任何应用程序。网址：https://www.docker.co[m/.](http://www.docker.com/)
14. 约翰·埃里克森、马丹拉尔·穆苏瓦蒂、塞巴斯蒂安·伯克哈特和柯克·奥林尼克。有效的数据-内核的竞争检测。在2010年10月于加拿大BC温哥华举行的第九届Usenix操作系统设计和实现会议记录中。
15. 等等。分布式、可靠的键值存储，用于存储分布式系统中最关键的数据。网址：https://github.com/coreos/etcd.
16. Cormac Flanagan和Stephen N Freund。原子化器：用于多线程程序的动态原子性检查器。2004年1月，在意大利威尼斯举行的第31届ACM SIGPLAN-SIGCT编程语言原则研讨会（POP*L'04）上发表论文。*
17. 齐高、张文彬、陈哲哲、郑迈和秦峰。第二个打击：显示隐藏的并发类型状态错误。2011年3月，在美国加利福尼亚州纽波特海滩举行的第16届编程语言和操作系统架构支持国际会议（ASPLOS‘11）上发表论文。
18. GitHub。GitHub上15种最流行的语言。URL：https://octoverse.github.com/.
19. 谷歌。一个高性能、开源、通用的RPC框架，将移动和HTTP/2放在首位。网址：https://github.com/grpc/grpc-go.
20. 谷歌。有效围棋。网址：https://golang.org/doc/effective\_go.html.
21. Google. 有效Go：并发.URL：https://golang.org/doc /effective\_go.html#concurrency.
22. Google. RPC基准测试URL：https://grpc .io/docs/guides/benchmarking.html.
23. 谷歌。Go编程语言–发布历史。网址：https://golang.org/doc/devel/release.html.
24. Rui Gu、Guoliang Jin、Linhai Song、Linjie Zhu和Shan Lu。什么样的变化历史告诉我们线程同步。2015年8月，意大利贝加莫，2015年第十届软件工程基础联席会议论文集。
25. Haryadi S.Gunawi、Mingzhe Hao、Tanakorn Leesatapornwongsa、Tiratat Patana anake、Thanh Do、Jeffry Adityatama、Kurnia J.Eliazar、Agung Laksono、Jeffrey F.Lukman、Vincentius Martin和Anang D.Satria。云上有什么虫子？对云系统中3000多个问题的研究。2014年11月，美国华盛顿州西雅图，ACM云计算研讨会（SOCC'14）论文集。
26. 公顷。轻量级SMTP客户端，以go编写。网址：https://github.com/hectane.
27. C、 A.R.霍尔。通信顺序过程。ACM通讯，21（8）：666-6771978。
28. Omar Inverso、Truc L.Nguyen、Bernd Fischer、Salvatore La Torre和Gennaro Parlato。Lazy cseq：用于多线程c程序的上下文限制模型检查工具。2015年11月，在美国内布拉斯加州林肯举行的第30届IEEE/ACM自动化软件工程国际会议（ASE’*15）上。*
29. 金国亮、宋林海、石晓明、乔尔·谢尔佩兹和陆山。了解和检测真实世界的性能错误。2012年6月，在中国北京举行的第33届ACM SIGPLAN编程语言设计与实现会议（PLDI'12）上。
30. 金国梁、宋林海、张伟、陆山和本·利卜利特。自动原子性违规修复。2011年6月在美国加利福尼亚州圣何塞举行的第32届ACM SIGPLAN编程语言设计与实现会议（PLDI'11）上发表。
31. Jin Gu亮亮，Aditya V.Thakur，Ben Liblit和Shan Lu。协作并发bug隔离的仪器和采样策略。《面向对象编程系统语言和应用ACM国际会议论文集》（OOPSLA’10），美国内华达州雷诺/-塔霍，2010年10月。
32. 金国亮、张伟、邓冬冬、本·利布莱特和卢山。自动并发错误修复。2012年10月在美国加利福尼亚州好莱坞举行的第十届USENIX操作系统设计与实现会议（OSDI'12）上发表。
33. Pallavi Joshi、Chang Seo Park、Koushik Sen和Naik市长。一种用于检测实际死锁的运行驻留动态程序分析技术。2009年6月，爱尔兰都柏林，第30届ACM SIGPLAN程序语言设计与实施会议（PLDI'09）论文集*。*
34. Horatiu Jula、Daniel Tralamazza、Cristian Zamfir和George Candea。死锁免疫：使系统能够防御死锁。2008年12月在美国加利福尼亚州圣地亚哥举行的第八届USENIX操作系统设计和实现会议（OSDI'0*8）上发表。*
35. 丹尼尔·克罗宁、丹尼尔·波茨尔、彼得·施拉姆和比约恩·韦希特。c/pthreads的静态死锁分析。2016年9月，在新加坡举行的第31届IEEE/ACM自动化软件工程国际会议（ASE’16）上。
36. Kubernets。生产级容器编排。网址：https://kubernetes.io/.
37. 莱斯利·兰波特。同时阅读和写作。ACM通讯，20（11）:806-811977。
38. Julien Lange、Nicholas Ng、Bernardo Toninho和Nobuko Yoshida。围栏关闭：基于频道的节目的生动性和安全性。2017年1月，法国巴黎，第44届ACM SIGPLAN编程语言原则研讨会（POPL‘17）论*文集。*
39. Julien Lange、Nicholas Ng、Bernardo Toninho和Nobuko Yoshida。使用行为类型传递消息的静态验证框架。2018年6月，在瑞典哥德堡举行的IEEE/ACM第40届国际软件工程会议（ICSE’18）上。
40. Tanakorn Leesatapornwongsa、Jeffrey F.Lukman、Shan Lu和Haryadi S.Gunawi。Taxdc：数据中心分布式系统中非确定性一致性错误的分类。2016年4月，美国佐治亚州亚特兰大，第21届编程语言和操作系统体系结构支持国际会议（ASPLOS‘16）论文集。
41. 李振民、谭林、王宣辉、陆山、周媛媛和翟成祥。现在情况变了吗？现代开源软件中bug特性的实证研究。2006年10月，在美国加利福尼亚州圣何塞举行的“提高软件可靠性的体系结构和系统支持”（ASID'06）第一次研讨会论文集。
42. 林子怡、达尔科·马里诺夫、钟浩、陈玉婷和赵建军。Jacobebe：一套真实的java并发bug的基准测试套件。2015年11月，在美国内布拉斯加州林肯举行的第30届IEEE/ACM自动化软件工程国际会议（ASE’15）上。
43. 刘昊鹏、陈玉玺和陆山。了解并生成高质量的竞赛bug补丁。2016年11月，美国华盛顿州西雅图，2016年第24届ACM SIGSOFT软件工程基础国际研讨会（FSE’16）论文集。
44. 陆兰岳，Andrea C.Arpaci Dusseau，Remzi H.Arpaci Dusseau和Shan Lu。linux文件系统演化研究。2013年2月，美国加利福尼亚州圣何塞，第11届USENIX文件和存*储技术会议（FAST’13）论文集。*
45. Shan Lu、Soyeon Park、Eunsoo Seo和Yuanyuan Zhou。从错误中学习——对现实世界并发bug特性的全面研究。2008年3月在美国华盛顿州西雅图举行的第十三届编程语言和操作系统体系结构支持国际会议（ASPLOS'08）上发表。
46. Shan Lu、Joseph Tucek、冯勤和周媛媛。Avio：通过访问交织不变量检测原子性违规。2006年10月在美国加利福尼亚州圣何塞举行的第12届国际程序设计语言和操作系统体系结构支持会议（ASPLOS'06）上发表。
47. 布兰登·露西亚和路易斯·塞泽。使用上下文感知通信图查找并发错误。2009年12月，美国纽约，第42届IEEE/ACM国际微体系结构研讨会（MICRO’09）论文集。
48. Kedar S.Namjoshi。更容易编写的并发程序也更容易检查吗？在高效和正确利用并发性研讨会上，2008年。
49. Nicholas Ng和Nobuko Yoshida。通过全局会话图合成实现静态死锁检测。2016年3月，在西班牙巴塞罗那举行的第25届编译器构造国际会议（CC’16）上发表论文。
50. Rob Pike。Go C oncurrency Patterns.URL：https: //talks.golang.org/2012/concurrency.slide.
51. 道森·R·恩格勒和肯·阿什克拉夫特。Racerx：有效、静态地检测竞争条件和死锁。《第19届ACM操作系统原理研讨会论文集》（SOSP'03），美国纽约，Bolton Landing，2003年*10月。*
52. Stefan Savage、Michael Burrows、Greg Nelson、Patrick Sobalvarro和Thomas Anderson。橡皮擦：动态数据竞争检测器

多线程程序。ACM计算机系统学报，15（4）：*391-4111997。*

1. 康斯坦丁·塞雷布雷尼和铁木尔·伊斯霍季恰诺夫。Threadsanitizer：实际中的数据竞争检测。《二进制仪器和应用研讨会论文集》（WBIA’09），美国纽约，2009年12月。
2. Vivek K Shanbhag。使用静态分析在java库中检测死锁。2008年12月，在中国北京举行的第15届亚太软件工程会议（APSEC’08）上。
3. 弗朗切斯科·索伦蒂诺。Picklock：一种死锁预测方法

嵌套锁定。《第22届模型检查软件国际*研讨会论文集》（SPIN'15），南非Stellenbosch，2015年8月。*

1. Kai Stadtmüller、Martin Sulzmann和Peter“Thiemann。同步迷你围棋的基于静态跟踪的死锁分析。第14届亚洲编程语言和系统符号（APLAS’16），越南河内，2016年11月。
2. 王杰、窦文胜、于高、高楚书、冯勤、康寅、魏军。对node.js中真实世界并发错误的全面研究。《第32届IEEE/ACM自动化软件工程国际会议（ASE’17）论文集》，美国伊利诺伊州厄巴纳-香槟，2017年10月。
3. 王茵、特伦斯·凯利、曼朱纳斯·库德卢尔、斯特凡·拉福和斯科特·A·马赫克。Gadara：多线程程序的动态死锁避免。2008年12月在美国加利福尼亚州圣地亚哥举行的第八届USENIX操作系统设计和实现会议（OSDI'08）上发表。
4. 尹旺、斯特凡·拉福、特伦斯·凯利、曼朱纳斯·库德卢尔和斯科特·A·马赫克。通过离散控制避免死锁的理论。2009年1月，在美国佐治亚州萨凡纳举行的第36届ACM SIGPLAN-SIGCT编程语言原则研讨会（POPL'09）上发表论文。
5. Wikipedia. Go（编程语言）。网址：https://en.wikiped ia.org/wiki/Go\_（编程语言）。
6. 熊薇薇、朴素妍、张佳琪、周媛媛和马志强。临时同步被认为是有害的。2010年10月，在加拿大不列颠哥伦比亚省温哥华举行的第九届USENIX操作系统设计和实现会议（OSDI'10）上。
7. Jie Yu和Satish Narayanasamy。交织约束共享内存多处理器的情况。2009年6月，美国德克萨斯州奥斯汀，第36届计算机架构国际研讨会（ISCA'09）论文集。
8. Yuan Yu、Tom Rodehefer和Wei Chen。赛道：通过自适应跟踪有效检测数据竞赛条件。2005年10月，在英国布莱顿举行的第20届ACM操作系统原理研讨会（SOSP'05）上发表论文。
9. 张伟、孙崇和陆山。康门：通过面向效果的方法检测严重的并发错误。2010年3月，在美国宾夕法尼亚州匹兹堡举行的第15届程序设计语言和操作系统体系结构支持国际会议（ASPLOS’10）上发表论文。