go并发编程

对 https://songlh.github.io/paper/go-study.pdf 论文的翻译和译注，未完待续，待继续补充论文所有的图片，表格，并优化排版等……

摘要

Go 是一种静态类型的编程语言，旨在提供一种简单、高效和安全的方式来构建多线程软件。 自 2009 年创建以来，Go 已经日趋成熟，并在生产环境和开源软件广泛使用。 Go 提倡使用消息传递作为线程间通信的手段，并提供了几个新的并发原语和标准库来简化多线程编程。就程序错误和bug而言，了解这些新提案的含义以及对比消息传递和共享内存同步是非常重要的。 很遗憾的是，据我们所知目前还没有关于 Go 的并发 bug 的研究。

在本论文中，首先我们对真实世界的go并发错误进行系统研究。我们研究6种非常流行的go软件，包括：docker,k8s, gRPC等；我们共分析了 171 个并发 bug，其中超过一半是由非传统的、go语言特性导致的（Go-specific，go特定的）问题。除了这些错误的根本原因，我们还研究了它们的修复方案，bug复现实验，并使用两个公开可用的 Go 错误检测器对其进行评估。总的来说，我们的研究既可帮助大家更好理解的go并发模型，也可以指导未来的研究人员和从业者编写更好、更可靠的 Go 软件、或开发 Go 调试和诊断工具。

1 简介

golang 是谷歌公司与2009年设计推出的静态类型编程语言。在过去几年中，他获得了极大的关注并在生产环境下被很多类型的软件广泛采用。这些go软件应用范围从基础库、高级应用到云基础设施如：容器系统，简直数据库等。

Go 的一个主要设计目标是改进传统的多线程编程语言，并使并发编程更简单且更少出错。为此，Go以多线程设计为中心，并围绕两个原则：

- 1、使线程（称为 goroutines）更轻量级且容易创建；
- 2、使用显式消息传递（称为channel）进行线程间通信。

为了这些设计原则，Go 不仅提出了一套新的原语、新标准库；同时提供了现有语义的新实现。

了解 Go 的新并发原语、并发机制如何影响并发 bug 非常重要，这类错误（指并发编程bugs）在传统多线程编程语言中最难调试、并且也被最广泛研究。不幸的是，目前还没有有关 Go 并发编程错误的研究。导致迄今为止，与传统语言相比，我们无法明确（go提供的）这些并发机制是否真的使得 Go 更易于编程并且更不易出并发错误。

在本论文中，首先我们使用六个开源的、生产级的开源软件开展有关go并发错误的研究。这六个软件包括：

- docker，k8s，两个数据中心容器系统；
- etcd，分布式键值存储系统；
- gRPC，RPC库；
- CockroachDB，BoltDB，两个数据库系统。

我们总共研究了这些应用程序中的 171 个并发错误。 我们分析了这些bug的根本原因，执行bug复现实验，并检查它们的修复补丁。 最后，我们用两个现有的 Go 并发错误检测器（唯二公开可用的）进行测试。

我们的研究聚焦于并发编程中一个长期存在且非常基础的问题：在消息传递和共享内存之间，哪种线程间通信机制更不易出错。Go 是研究这个问题的完美语言，因为它提供了同时支持共享内存和消息传输的框架。但是，它更鼓励使用通过而不是共享内存，因为他信仰显式消息传递更少出错。

为了了解 Go 并发错误并比较在消息传递和共享内存，我们建议沿着两个正交维度对并发错误进行分类：错误原因及其行为。在错误原因维度上，我们把bug分类共享内存误用和消息传递误用导致的bug。在行为纬度上，我们也把bug分为两类：

- 1、涉及（任意数量的）goroutine无法继续执行推进的bug；我们称之为阻塞型bug。
- 2、不涉及任何阻塞的bug，称之为非阻塞型bug。

令人惊讶的是，我们的研究表明，消息传递和共享内存一样容易产生并发错误，有时甚至更多。例如，大约 58% 的阻塞型bug是由消息传递引起的。除了违反 golang chan 的使用规则（如：在一个没有发送数据或关闭的chan上等待获取数据），许多并发错误是由于混使消息传递和其他Go中的新语义和新程序库导致的，这些错误很容易被忽视，但很难排查。

为了演示消息传递导致的buug，我们使用图 1 中 Kubernetes 的阻塞型错误。finishReq 函数使用在第 4 行的匿名函数创建一个子 goroutine 来处理请求——这是go服务程序的一个常见做法。
子 goroutine 执行 fn 函数，并通过 ch chan 将结果发送回父 goroutine（在第 6 行）。子goroutine在第六行将阻塞，直到父goroutine消费 ch (第九行)。同时，父goroutine在执行select语句时也将阻塞直到子goroutine发送数据到ch或者超时。如果超时发生得更早或者 两个select case都同时满足时，go 运行时选择执行第二个case(第11行)，那么父 goroutine 将在第12行返回。那么将没有任何goroutine可以消费ch，导致子 goroutine 永久阻塞。解决方法是将ch 从无缓冲chan 更改为 有缓冲chan，这样即使父级退出子goroutine 就可以也始终发送结果。（译注：指子goroutine成功发送结果到ch并不被阻塞，从而也正常退出）

这个 bug 表明了在 Go 中使用新特性的复杂性以及编写正确的 Go 程序的难度。程序员为此需要对以下特性有清晰的认识：使用匿名函数创建 goroutine（go提倡的用来简化goroutine创建的特性），有缓冲chan和无缓冲chan的使用；使用select语句等待多个chan操作的不确定性；和特定的有关time的标准库。尽管这些特性中的每一个都旨在简化多线程编程，但实际上，用它们编写正确的 Go 程序也很困难。

总的来说，我们的研究揭示了go并发编程的新特性和新问题，它揭示了有关消息传递与共享内存访问争论的答案。我们的发现有利于人们了对 Go 并发的理解，并且可以为未来的工具设计提供有价值的指导。本论文具有以下关键贡献：

- 1、有我们首次，对六个真实世界的生产级 Go 应用程序进行了 Go 并发错误的实证研究。
- 2、我们对 Go 并发错误的原因、修复和检测进行了九个高级关键考察；他们对 Go 程序员的参考很有用。进一步的，我们对我们研究的意义提出八项见解，这些见解能指导未来有关golang的开发，测试和bug检测。

- 3、我们提出了依据错误原因和行为两个维度对并发bug进行分类的新方法。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、修复的关联性。我们相信其他bug研究也可以同样借鉴类似的分类方法。

我们所有的研究结果和提交日志可参见： https://github.com/system-pclub/go-concurrency-bugs。

2、背景和应用

Go 是一种静态类型的编程语言，从一开始就被设计用于并发编程。几乎所有主要的 Go 修订版都包括对其并发模块包的改进。本节简要介绍关于 Go 并发机制的背景，包括它的线程模型，线程间通信方法和线程同步机制。我们还介绍了我们为这项研究选择的六个 Go 应用程序。

2.1 goroutine

Go 使用一个叫做 goroutine 的概念作为它的并发单元。Goroutines 是轻量级的用户级线程，Go运行时库管理他们并以 M : N 的方式映射到内核级线程。创建 goroutine 可以通过很简单的方式：在函数调用之前添加关键字 go即可。为了使 goroutines 易于创建，Go 还支持使用匿名函数创建一个新的 goroutine（函数定义没有标识符或“函数名称”）。所有在匿名函数可访问之前声明的局部变量，匿名函数可直接访问。并且这些局部变量也极有可能在父协程和子协程直接是共享的，当然这会导致数据竞争（第 6 节）。

2.2 共享内存同步

Go 支持goroutine间传统的共享内存访问。它支持各种传统的同步原语，如加锁/解锁(Mutex)、读写锁 (RWMutex)、条件变量 (Cond) 和原子读写 (atomic)。 Go 对 RWMutex 的实现与 C 中的 pthread_rwlock_t 不同。 Go 中的写锁请求比读锁请求具有更高的优先级。

作为 Go 引入的新原语，sync.Once 被设计用于保证一个函数只执行一次。它有一个Do方法，以函数 f 作为参数。即使Once.Do(f)被多次调用，只有第一次调用，f 才会被执行。Once被广泛用于确保共享变量只会由多个 goroutine 初始化一次。

与 C 中的 pthread_join 类似，Go 使用 WaitGroup 来允许多个 goroutine 在等待 goroutine 之前完成它们的共享变量访问。 Goroutines 通过调用 Add 被添加到 WaitGroup。 WaitGroup 中的 Goroutine 使用 Done 来通知它们的等待的条件已经完成，一个 goroutine 调用 Wait 来等待 WaitGroup 中所有 goroutine 的完成通知。滥用 WaitGroup 会导致阻塞错误（第 5 节）和非阻塞错误（第 6 节）。

2.3 消息传递同步

chan是 Go 引入的新并发原语，它可以用于跨 goroutine 发送数据和状态并可以实现更复杂的功能。Go 支持两种类型 chan：缓冲和无缓冲。将数据发送到一个无缓冲的通道（或从中接收数据）将阻塞 goroutine，直到另一个 goroutine 从chan读取数据（或发送数据到chan）。发送数据到到缓冲chan，只有缓冲区已满时才会阻塞。使用chan有几个潜在的规则，违反这些规则可能导致并发bug。比如：

- 1、chan只能初始化之后才能使用；
- 2、向nil chan发送数据或读取数据将永远阻塞当前 goroutines；
- 3、发送数据到一个关闭的chan或关闭一个已经关闭的chan可以触发运行时恐慌(panic)。

select 语句允许 goroutine 等待多个chan操作。 select语句将阻塞，直到它的一个case分支满足或可执行default分支。 当一个 select 中有多个 case 有效时，Go 会随机选择一个来执行。 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跨多个 goroutine 的交互。例如：为了方便通过生成一组协同工作的 goroutines 来处理服务用户请求的编程模型，Go 引入了跨goroutines上下文来携带特定请求的数据或元数据。再举一个例子，Pipe 旨在在 Reader 和 Writer 之间传输数据。 context 和 pipe 都是传递消息的新形式，滥用它们会产生新类型的并发错误（第 5 节）。

2.4 Go应用

近年来， Go 语言人气迅速上升并被广泛采用。 Go 位列2017 年 GitHub 最受欢迎语言第 9 名 。截至撰写本文时，有 187K GitHub 项目是用 Go 编写的。在这项研究中，我们选择了六个具有代表性的真实世界
用 Go 编写的软件，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（CockroachDB 和 BoltDB）和一个 RPC程序库（gRPC-go1）（表格1）。 这些应用程序是在数据中心获得广泛使用的开源项目环境。 例如，Docker 和 Kubernetes 是在 GitHub 上用 Go 编写的最受欢迎的2个应用程序，分别有 48.9K 和 36.5K 星（etcd 是第 10 个，其余都排名前100）。 我们选择的应用程序都至少有三年的开发历史，目前由开发者积极维护。 我们所有选定的应用程序都为大中型项目，代码行数从 9千到两百万以上不等。 在六个应用程序中，
Kubernetes 和 gRPC 是最初由谷歌开发。

3 Go 并发使用模式

在研究 Go 并发 bug 之前，首先了解现实世界中的 Go 并发程序到底怎样是非常重要的。本节介绍我们选择了六个应用程序有关goroutine 用法和 Go 并发原语用法的静态和动态分析结果。

3.1 Goroutine 的使用

要理解 Go 中的并发，我们首先应该了解 goroutines 在现实世界的 Go 程序中是如何使用的。

Go 的设计理念之一是使得 goroutine 轻量级且易于使用。 因此，我们可能会有疑问：“真正的 Go 程序员是否倾向于使用许多 goroutines 编写他们的代码？（静止的）？”，“真正的 Go 应用程序在运行时（动态）创建了很多 goroutines 吗？”为了回答第一个问题，我们统计了创建 goroutine 的数量（即：创建goroutine的源码行数。 表 2 表明了结果。 总的来说，六个应用程序都使用了大量的 goroutine。每千个源代码行的平均创建goroutine数为 0.18 到 0.83。我们进一步将创建goroutine的方式进行区分：使用普通函数来创建 goroutines 和使用匿名函数创建。 除了 Kubernetes 和 BoltDB ，所有应用程序更多的使用匿名函数。

为了了解 Go 与传统语言的区别，我们还分析了另一个gRPC实现：用C/C++实现的gRPC-C。 gRPC-C 包含 140K 行代码，也由 Google 的gRPC 团队维护。 与 gRPC-Go 相比，gRPC-C 线程创建数量出人意料的少（平均每千行代码只有0.03个线程创建点）。

我们进一步研究了 运行时创建的goroutine 。 我们跑了 gRPC-Go 和 gRPC-C 来执行三种性能压测，它被设计用于比较用不同的编程语言编写的 gRPC 版本性能。这些基准gRPC测试配置包括：不同的消息格式、不同的连接数以及同步与异步 RPC 请求等。由于 gRPC-C 比 gRPC-Go 快，我们让 gRPC-C 和 gRPC-Go 允许处理相同数量的 RPC 请求，而不是运行相同的时间。

表 3 显示了在运行三种工作负载时，在 gRPC-Go 中创建的 goroutine 数量与在 gRPC-C 中创建的线程数量的比率。不同工作负载下，客户端和服务端都创建了更多的goroutines。表 3 还展示了我们有关 goroutine 运行持续时间的研究结果，并将它们与 gRPC-C 线程运行持续时间进行比较。由于 gRPC-Go 和 gRPC-C 的总执行时间不同，比较 goroutine/thread 绝对持续时间没有意义，我们报告和比较 gRPC-Go或gRPC-C的 goroutine或线程持续时间和总运行时间的相对值。具体来说，我们利用所有 goroutines/threads 的执行时间计算平均值并对其使用程序的总执行时间进行规范化。我们发现所有gRPC-C 中的线程从开始执行，一直到整个程序结束执行（即 100%），因此只包括gRPC-Go 的结果见表 3（注：这里没有看懂，不知如何翻译，见谅）。 对于所有工作负载，规范化的执行时goroutine 比线程短。

- **结论 1：**Goroutines 比 C 执行时间更短，但创建频率更高（静态和运行时）。

3.2 并发原语使用

在对 goroutine 在真实世界Go 程序中的用法有了基本了解之后，我们接下来研究 在这些程序中 goroutines 如何通信和同步的。具体来说，我们计算了在六个应用程序中，不同类型并发原语的使用情况。表 4 显示了总数（原始使用的绝对数量）和每种类型原语的使用比例。共享内存同步操作比消息传递使用的更频繁，所有应用程序中Mutex互斥锁是最广泛使用的原语。对于消息传递原语， chan 是使用频率最高的一种，占比从 18.48% 到 42.99% 不等。

我们进一步比较了在 gRPC-C 和 gRPC-Go 中并发原语的使用。 gRPC-C 只使用锁，而且它是在 746 个地方使用（每千行代码 有 5.3 个并发原语使用）。 gRPC-Go在 786 个地方使用了八种不同类型的原语（每千行代码14.8个并发原语使用）。显然，相比 gRPC-C，gRPC-Go 使用了更多数量和更多种类的并发原语。

接下来，我们研究一下并发原语使用随着时间推移的变化。 图 2 和图 3 显示了从 2015 年 2 月到 2018 年 5 月六个应用程序中共享内存消息传递原语使用情况。总体而言，随着时间的推移使用量基本保持稳定，这也意味着我们的研究结果对未来的 Go 程序员很有价值。

- **结论2：**虽然传统的共享内存线程通信和同步方式仍被大量使用，Go 程序员还使用大量的消息传递原语。
- **推论1：**随着 goroutines 和 新类型并发原语的大量使用，Go 程序有可能引入更多并发bug。

4 bug研究方法论

本节讨论我们如何收集、归类和复现本论文中研究的bug。收集并发bug，为了收集并发bug，我们通过搜索和并发关键字有关的提交日志，来过滤以上六个应用的github历史提交日志。搜索关键字包括：数据竞争，死锁，同步，并发，锁，互斥锁，原子性，上下文，Once，协程泄漏等（“race”，“deadlock”，“synchronization”，“concurrency”，“lock”，“mutex”，“atomic”，“compete”，“context”，“once”，and “goroutine leak”）。 其中一些关键字在其他的工作中也被用于收集其他语言中的并发bug。其中一些是Go引入的新并发原语或相关标准库，例如context库和once(译注：sync.Once库)。 其中，“goroutine 泄漏”，与 Go 中的一个特殊问题有关。我们总共发现 3211 个符合我们搜索条件的不同提交记录。

然后我们随机抽样过滤后的commit，确认属于修复并发bug的commit，并手动研究他们。 很多与bug相关的commit 日志也提到了相应的bug报告，我们为了bug分析也研究了这些报告。 我们总共研究了 171 个并发错误。

错误分类法

我们提出了一种根据两个正交维度对 Go 的并发bug进行分类的新方法。第一个维度基于错误的行为。如果一个或多个 goroutine 执行中过程无意间卡住并且无法继续推进，我们称这类并发问题为阻塞性bug。反之，如果所有的 goroutine 都可以完成他们的任务但他们的行为不符合预期，我们称这类问题为非阻塞bug。

以前的大多数并发bug研究将错误分类为死锁型bug和非死锁型bug，死锁型bug包括存在多个线程等待循环的情形。我们对阻塞的定义是比死锁更广泛，包括存在以下情况的场景：没有循环等待，但是有一个（或多个）goroutines 等待其他 goroutine 永远都不会提供的资源。正如我们将在第 5 节中展示的那样，相当多的 Go 并发错误都属于这个类型。我们相信随着新编程习惯的适应和有类似go语义的新语言的出现，人们应该会对这些非死锁阻塞型bug给予更多关注，并扩展传统的并发buug分类机制。

第二个维度是依据导致并发bug的原因。 当多个线程尝试通信时会导致并发bug，或者在这种通信过程中发生错误。因此，我们的想法是通过以下不同的 goroutine 通信方式对并发bug的原因进行分类：通过共享内存访问或通过消息传递。 这种分类方法可以帮助程序员和研究人员选择更好的线程间通信方法，同时可以检测和避免执行此类线程间通信时的潜在错误。

根据我们的分类方法，总共有85 个阻塞性bug和 86 个非阻塞性bug；还有105个误用共享内存保护导致的bug和 66 个误用消息传递导致的bug。表 5 显示了在每个应用程序中bug类别的详细细分。

我们进一步分析了我们研究的bug的生命周期，即：从添加（提交）错误代码开始的时间到软件中修复的时间（提交了错误修复补丁）。如图4所示，我们研究的大部分bug（包括共享内存型和消息传递型）的生命周期都很长。我们还研究了这些bug被报告的时间到bug被修复而关闭的时间。结果表明：我们研究的大多数bug都不太容易触发或被检测到，但是一旦触发或被检测到，它们也很快就得到修复。 因此，我们相信这些bug都很意义并且值得仔细推敲。

并发bug复现

为了评估Go 内置的死锁和数据竞争检测技术，我们复现了 21 个阻塞型bug和 20 个非阻塞型bug。 为了复现错误，我们将应用程序回滚到有bug的版本，构建有bug的版本，并使用GitHub bug report中描述的错误触发输入运行所构建的程序。我们利用了bug report中提到的bug表现来判定我们是否成功复现了一个bug。由于它们的非确定性，并发bug很难复现。有时，我们需要运行有bug的程序很多次或需往有bug的程序手动添加sleep语句等。对于没有复现的bug，要么是因为我们没有相应的依赖库，或者是因为我们未能观察到所描述的bug表现。

效度威胁

对我们研究有效性的威胁来自很多方面。 我们选择了六个具有代表性的 Go 应用程序， 还有许多用Go 实现其他应用，它们可能不会面临相同的并发问题。 我们只研究了已修复的并发错误， 可能还有其他并发错误，他们很少被复现并且从未被开发人员修复。对于一些已修复的并发bug，提供的信息太少使得难以理解。我们的研究中不包括这些错误。 尽管有这些限制，我们仍尽最大努力收集现实世界中的Go 并发 bug 和并进行全面、公正的研究。 我们相信我们的发现足以激励和指导未来的Go并发错误研究。

5 阻塞型bug

本节介绍我们在阻塞性错误方面的研究结果，包括它们的根本原因、修复和Go 内置的运行时死锁检测器在检测死锁阻塞情形的有效性。

5.1 阻塞错误的根本原因

当一个或多个 goroutine 执行等待资源的操作时，就会出现阻塞错误，而这些资源永远不可用。 为了检测和避免阻塞错误，了解它们的根本原因很重要。 我们通过检查哪个操作阻塞了一个 goroutine 以及为什么该操作没有被其他 goroutine 解除阻塞来研究阻塞错误的根本原因。 使用错误分类的第二个维度，我们将阻塞错误分为由旨在保护共享内存访问的阻塞操作引起的错误和由消息传递操作引起的错误。 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比消息传递原语（第 3.2 节）更频繁地使用，消息传递操作更可能导致阻塞错误。

- **结论 3：**与消息传递不太容易出错的普遍看法相反，我们研究的 Go 应用程序中，更多的阻塞错误是由消息传递误用引起的，而不是由共享内存保护误用引起的。

5.1.1 共享内存保护

众所周知，共享内存访问很难正确编程，并且一直是死锁研究的主要焦点之一 [35, 51, 54]。 它们继续在 Go 中导致阻塞错误，既有传统模式，也有新的、特定于 Go 的原因。

Mutex

28 个阻塞错误是由误用锁（Mutex）引起的，包括重复加锁、以冲突的顺序获取锁、忘记解锁等。 所有此类别错误都是传统错误，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错误。

RWMutex

如 2.2 节所述，Go 的写锁请求比读锁请求具有更高的优先级。 这种独特的锁实现可能会导致阻塞错误：比如，当一个 goroutine (称之为 th-A) 通过两次读锁获取一个 RWMutex 时，并且这两个读锁操作和另一个 goroutine (称之为th-B) 的写锁操作交叉。当 th-A 的第一次读锁操作成功时，它将阻塞 th-B 的写锁操作（因为写锁是互斥的）。 但是，th-B 的写锁操作也会阻塞 th-A 的第二次读锁操作，因为写锁请求在 Go 的实现中具有更高的优先级。 最终 th-A 和 th-B 都无法继续执行。

译注：这种情形倒是第一次听说，长姿势了，后面写个程序验证下。

五个阻塞错误是由这个原因引起的。 请注意，相同的交叉锁定模式不会导致 C 中 pthread_rwlock_t 的阻塞错误，因为 pthread_rwlock_t 在默认设置下优先读取锁定请求。 RWMutex 的阻塞错误类型意味着即使 Go 使用与传统语言相同的并发原语，由于 Go 对原语的新实现方式，仍然可能导致新类型的错误。

Wait

三个阻塞错误是由于Wait 操作无法继续推进导致。 与 Mutex 和 RWMutex 相关的错误不同，它们不涉及循环等待。 （这三个错误中的）其中两个错误是因为：使用 Cond 来保护共享内存访问并且一个 goroutine 调用 Cond.Wait() ，但在此之后没有其他 goroutine 调用 Cond.Signal()（或 Cond.Broadcast()）。

第三个错误，Docker#25384， 在使用类型为 WaitGroup 的共享变量时导致，如图 5 所示。第 7 行的 Wait() 只能被第 5 行的 Done() 被调用 len (pm .plugins) 次才解除阻塞，因为 len(pm.plugins) 在第 2 行被用作调用 Add() 的参数。 然而，Wait() 在循环内被调用，因此它在下一次的循环迭代中阻塞了第 4 行的 goroutine 创建，同时也阻塞了在每个创建的 goroutine 中调用 Done()。 此错误的修复是将 Wait() 的调用从循环中移出。

虽然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犯这些并发错误的原因之一。 例如，与 pthread_join 是一个显式等待线程完成的函数调用不同，WaitGroup 是一个可以跨 goroutine 共享的变量，它的 Wait 函数隐式地等待 Done 函数。

虽然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犯这些并发错误的原因之一。 例如，与 pthread_join 是一个显式等待（命名）线程完成的函数调用不同，WaitGroup 是一个可以跨 goroutine 共享的变量，它的 Wait 函数隐式地等待 Done 函数。

- **结论 4：**与传统语言一样，大多数由共享内存同步引起的阻塞错误具有相同的原因和相同的修复方法。 然而，由于 Go 对现有原语的新实现或其新的编程语义，其中一些（由共享内存引发的阻塞型错误）与传统语言不同。

5.1.2 消息传递误用

我们现在讨论由消息传递误用引起的阻塞型bug，与通常的看法相反，这是我们研究的应用程序中阻塞错误的主要类型。

chan

在 goroutine 间误用chan传递消息的错误导致 29 个阻塞型bug。 许多与通道相关的阻塞型bug是由于缺少向chan发送数据（或从chan接收）或没有关闭chan导致的，这将导致等待从chan接收（或发送数据到chan）的 goroutine 阻塞。 图 1就是一个这样的例子。

结合 Go 特定库的使用，chan的创建和 goroutine 阻塞可能会隐藏在库调用中。 如图 6 所示，在第 1 行创建了一个新的上下文对象 hcancel的同时，也创建了一个新的 goroutine，可以通过 hcancel 变量的 channel 字段向新的 goroutine 发送消息。 如果第 4 行 timeout 大于 0，则在第 5 行创建另一个上下文对象，并且 hcancel 指向该新对象。 之后，就无法向旧hcancel对象关联的 goroutine 发送消息或关闭该 goroutine。 该补丁是为了避免在timeout大于 0 时创建额外的上下文对象。

chan和其他阻塞原语

对于 16 个阻塞型bug，一个是 goroutine 在chan操作时被阻塞，另一个 goroutine 在锁操作或等待时被阻塞。 例如图 7 所示，goroutine1 在向通道 ch 发送请求时被阻塞，而 goroutine2 在 m.Lock() 时被阻塞。 修复方法是为 goroutine1 添加一个带有默认分支的select语句，使得 ch 不再阻塞。

消息传输标准库

Go 提供了几个库来传输数据或消息，比如 Pipe。 如果使用不当，这些特殊的库调用也会导致阻塞错误。 例如，与 channel 类似，如果 Pipe 没有关闭，goroutine 在尝试向未关闭的 Pipe 发送数据或从未关闭的 Pipe 中读取数据时可能会被阻塞。 我们收集了 4 个由特定的 Go 消息传输库调用引起的阻塞型bug。

- **结论 5**：所有由消息传输引起的阻塞错误与 Go 的新消息传递语义有关：比如chan。它们很难以检测，特别是当消息传输操作与其他同步机制一起使用时。
- **推论 2**：与通常看法相反，消息传递比共享内存会导致更多的阻塞型bug。 因此我们呼吁关注使用消息传输进行编程的潜在危险，并提出该领域的错误检测研究问题。

5.2 阻塞错误的修复

在了解了 Go 中阻塞 bug 的原因之后，我们现在来分析 Go 程序员在现实世界中是如何修复这些 bug 的。

消除 goroutine挂起导致的阻塞可以解除阻塞，这是修复阻塞错误的通常方法。 为了达成这一目标，Go 开发人员经常调整同步操作，包括添加缺失的、移动或更改错位/误用的以及删除多余的。 表 7 总结了这些修复方式。

大多数因误用保护共享内存访问而导致的阻塞错误都通过类似于修复传统死锁的方法进行了修复。 例如，在 33 个 Mutex 或 RWMutex 相关的错误中，有 8 个是通过添加缺失的解锁来修复的； 9 通过移动锁定或解锁操作到适当的位置来修复； 和 11 个是通过删除额外的锁操作来修复的。

11 个由错误消息传递引起的阻塞错误通过在不同的 goroutine 上添加丢失的消息或对chan执行关闭操作（并且有两次，关闭Pipe）来修复。8 个阻塞错误通过往select语句添加default选项（例如，图 7）或不同chan的case分之操作进行修复。另一个与通道相关阻塞错误的常见修复是用缓冲chan替换无缓冲通道（例如图 1所示）。 其他与通道相关的阻塞错误可以用如下策略来修复：例如将chan操作移出临界区并用共享变量替换chan。

为了理解阻塞错误的原因与其修复之间的关系，我们应用了一个称为lift的统计指标，并遵循之前对现实世界错误的实证研究。

lift计算公式：lift(A，B)=P(AB)/P(A)P(B)；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。 当提升值等于 1 时，A 根本原因与 B 修复策略无关 . 当lift值大于1时，A和B是正相关的，也就是说如果是A引起的阻塞，则更可能被B修复。当lift小于1时，A和B是负相关的。

在所有具有超过 10 个阻塞错误的错误类别中（我们省略少于 10 个错误的类别，因为它们在统计上无意义），互斥锁是与移动加解锁位置修复方式具有最强的相关性——它的lift 值为 1.52。 Chan 和 Adds 之间的相关性是第二高的，lift值为 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，表明没有强相关性。

我们还根据阻塞型错误修复补丁中使用的并发原语类型进行分析。 正如预期的那样，大多数因为与某种类型的原语误用有关的bug也通过调整该原语得到了修复。 例如，所有与 Mutex 相关的错误通过调整 Mutex 原语得到修复。

造成的bug原因与用于修复它们的原语和策略的高度相关性，以及 Go 中有限类型的同步原语，意味着在研究 Go 中阻塞型错误的自动修复方面可取得卓有成效的成果。 我们进一步发现，我们研究的阻塞型错误的fix补丁大小很小，平均为 6.8 行代码。 我们研究的大约 90%的 阻塞型错误都是通过调整同步原语来修复的。

- **结论 6：**我们研究的大多数阻塞型错误（包括传统的共享内存错误和消息传递错误）可以通过简单的解决方案来修复，并且许多修复方案与错误原因相关。
- **推论 3：**Go 中阻塞型错误的原因和修复方案之间的高度相关性，以及修复补丁的简单性表明开发全自动或半自动化工具来修复 Go 中的阻塞型错误是有希望的。

5.3 阻塞型错误检测

Go 提供了一个内置的死锁检测器，它在 goroutine 调度器中实现。 该检测器始终在 Go 运行时中启用，假如当正在运行的进程中没有 goroutine 可以取得继续执行推进时，它就会报告死锁。 我们使用 Go 内置的死锁检测器测试了所有被复现的阻塞行错误，以评估它可以发现哪些错误。 对于每个被测试的bug，阻塞可以在每次运行中确定性地触发。 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。内置的死锁检测器只能检测两个阻塞错误，BoltDB#392 和 BoltDB#240，在所有其他情况下都失败（尽管检测器没有报告任何误报 [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 goroutine 正在运行时，它不会将被监控的系统视为阻塞。其次，它只检查 goroutines 是否在 Go 并发原语处被阻塞，而不考虑等待其他系统资源的 goroutines。这两个限制主要是因为内置检测器的设计目标——最小的运行时开销。当在运行时调度程序中实现（检测器）时，检测器很难在不牺牲性能的情况下有效识别复杂的阻塞错误。

- **推论 4：**Go中简单的运行时死锁检测器在检测阻塞型错误方面不太高效。未来的研究应侧重于构建新兴的阻塞型错误检测技术，例如，将静态和动态阻塞模式检测相结合。

6 非阻塞错误

本节介绍我们对非阻塞错误的研究。 与我们在第 5 节中所做的的类似，我们研究了非阻塞错误的根本原因和修复，并评估了 Go 的内置竞争（data-race）检测器。

6.1 非阻塞型bug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：由于未能有效保护共享内存而导致的bug；和消息传递出错（表 9）。

6.1.1 未能有效保护共享内存

之前的研究工作表明，不对共享内存访问进行（加锁）保护或这种保护执行中的错误是导致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% 的非阻塞错误是由于未保护共享内存访问或错误的保护共享内存访问造成的。 然而，并非所有阻塞型错误都与传统语言中的非阻塞错误原因相同。

传统 bug

我们收集的非阻塞错误中，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，包括：没有遵循原子性，没有遵循正确的顺序， 和数据竞争。 这个结果表明，不同语言的开发人员会犯同样的错误。 也同样表明使用现有的并发错误检测算法来检测 Go 中的bug是有希望的。

有趣的是，我们发现了七个非阻塞错误，其根本原因和传统问题一样，但主要是由于对新的 Go 特性缺乏清晰的理解造成的。例如，Docker#22985 和 CockroachDB#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chan在 goroutine 之间传递。

匿名函数

Go 设计者使 goroutine 声明类似于常规的函数调用（甚至不需要有“函数名称”），以简化 goroutine 的创建。 在 Go 匿名函数之前声明的所有局部变量都可以被匿名函数访问。 不幸的是，当使用匿名函数创建 goroutine 时，这种简化代码开发的方式会增加数据竞争错误的机会，因为开发人员可能没有充分意识到需要保护这些共享的局部变量。

我们发现了 11 个此类错误，其中 9 个是由使用匿名函数创建的子 goroutine 和父 goroutine 之间的数据竞争引起的。另外两个是由两个子 goroutine 之间的数据竞争引起的。 如图 8 所示：一个Docker 的例子。局部变量 i 在父 goroutine 和它在第 2 行创建的 goroutine 之间共享。开发人员意图每个子 goroutine 使用不同的 i 值来初始化第 4 行的字符串 apiVersion。然而，在有问题的程序中 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apiVersion的值都等于'v1.21'。只有当每个子 goroutine 在其创建后并且在 i 被分配给新值之前立即初始化字符串 apiVersion ，该有问题的程序才会产生预期的结果。 Docker 开发人员通过在每次循环迭代时复制共享变量 i 并将复制的值传递给新的 goroutine 来修复此错误。

误用WaitGroup

使用WaitGroup 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 显示了 etcd 中的一个类似错误，其中不能保证 func1 第 8 行的 Add 在 func2 第 5 行的 Wait 之前执行。 修复方法是将 Add 操作移到临界区，以确保 Add 要么在 Wait 之前执行，要么不执行。

标准库

Go 提供了许多新的标准库，其中一些使用由多个 goroutine 隐式共享的对象。 如果使用不当，可能就会发生数据竞争。 例如，context上下文对象类型被设计用来由关联到该context对象的多个 goroutine 访问。 etcd#7816 是由多个 goroutine 访问context 对象的字符串字段引起的数据竞争错误。

另一个例子是旨在支持自动化测试的testing包。 一个测试函数（函数名以“Test”开头来标识）只接受一个 testing.T 类型的参数，用于传递测试状态：例如错误日志等。 三个数据竞争错误是由于从运行测试函数的 goroutine 和在测试函数内部创建的其他 goroutine 访问 testing.T 变量引起的。

- **结论 7：**大约三分之二的共享内存非阻塞错误是由传统原因引起的。 剩下的三分之一由Go 新的多线程语义和新库导致。
- **推论 5：**Go 为简化多线程编程而引入的新编程模型和新标准库本身可能是导致更多并发错误的原因。

6.1.2 消息传递错误

消息传递过程中的错误也可能导致非阻塞错误，它们在我们收集的非阻塞错误中占比20% 左右。

chan误用

正如我们在第 2 节中讨论的，在使用 channel 时需遵循几个规则，违反这些规则除了阻塞错误之外还会导致非阻塞错误。 有 16 个非阻塞错误是由误用chan引起的。

例如图 10 所示的 Docker#24007 bug，是由于违反了chan只能关闭一次的规则造成的。 当多个 goroutine 执行这段代码时，其中不止一个可以执行 default 子句并尝试关闭第 5 行的通道，从而导致 Go 中的运行时恐慌。 修复方法是使用sync.Once包来强制chan只关闭一次。

同时使用 channel 和 select 时会发生另一种并发错误。 在 Go 中，当一个 select 收到多条消息时，不能保证先处理哪一条。 select 的这种不确定性实现导致了 3 个错误。 图 11 显示了一个这样的例子。第 2 行的循环，在每次定时器滴答时（第12行，case语句2）执行第8行比较耗时的函数 f()，并在第 10 行（case 语句1）从通道 stopCh 接收消息时停止执行。如果在收到来自 stopCh 消息的同时定机器也滴答了，则无法保证 select 会选择哪一个case语句。 如果 select 选择 case 2，f() 将不必要地再执行一次。 解决方法是在循环开始处添加另一个select语句来处理来自 stopCh 的未处理信号。

标准库

Go 的一些特定的标准库以一种微妙的方式使用chan，这也会导致非阻塞错误。 图 12 显示了一个与time包相关的的错误，time包被设计用来计量时间。在这里，在第 1 行创建了一个超时时间为 0 的计时器。在创建 Timer 对象时，Go 运行时（隐式）启动了一个标准库内置的 goroutine，它开始计时器倒计时。计时器在第 4 行设置了一个超时值 dur。开发人员这里的意图是：仅在 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，库内置的 goroutine 会在计时器创建后立即向 timer.C chan发送信号，导致函数过早返回（第 8 行）。 解决方法是避免在第 1 行创建 Timer。

- **结论 8：**与共享内存访问相比，由消息传递引起的非阻塞错误要少得多。 chan规则和使用chan与其他 Go 特定语义和标准库的复杂性是这些非阻塞错误发生的原因。
- **推论 6：**如果使用得当，消息传递比共享内存访问更不容易出现非阻塞错误。 然而，当与其他特定于语言的特性结合时，编程语言中消息传递的复杂设计会导致这些错误特别难以发现。

6.2 修复非阻塞型错误

与我们对阻塞型错误修复的分析类似，我们首先通过策略分析非阻塞错误的修复。 表 10 对我们研究的 Go 非阻塞错误的修复策略进行了分类，其方式与之前在 C/C++ 中对非阻塞错误修复的分类类似。

大约 69% 的非阻塞错误是通过限制时间线（没太懂什么是限制时间线，原文为fixed by restricting timing）来修复的：比如通过添加同步原语（如互斥锁），或通过移动现有原语（如图 9 中的移动Add操作）。通过移除访问共享变量的指令或绕过指令（例如，图 10）修复了 10 个非阻塞错误。 通过创建共享变量的私有副本（如图 8）修复了 14 个错误，这些错误都是共享内存错误。

为了更好地了解非阻塞错误修复补丁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根据使用的原语类型 列出了修复补丁（数）。

与之前对 C/C++ 中并发错误修复补丁的研究结果类似，Mutex是最广泛使用的原语，用于强制互斥访问和修复非阻塞错误。 除了传统的错误，互斥锁也被用于修复由匿名函数和 WaitGroup 引起的数据竞争，以及用于替换误用的chan。

作为一个新的原语，chan是第二被最广泛使用的。chan 被用来在两个 goroutines 之间传递值并替换共享变量来修复数据竞争。 它还用于强制不同 goroutine 中两个操作之间的执行顺序。 还有一些错误，如chan没被正确使用并已在补丁中修复（如图 10）。
有趣的是，channel 不仅用于修复消息传递错误，还用于修复由传统共享内存同步引起的错误。 我们怀疑这是因为：一些 Go 程序员将消息传递视为比共享内存同步更可靠或更易于编程的执行线程间通信的方式。

最后，通过其他并发原语修复了 24 个错误，另外 19 个错误在不使用任何并发原语的情况被修复。（如图 8）。

类似于我们在 5.2 节中的lift公式分析，我们计算了非阻塞错误型原因和修复策略之间的lift值；以及非阻塞错误原因和修复原语之间的lift值。在超过 10 个 bug 的 bug 类别中，相关性最强的是误用chan和使用chan原语进行修复之间的相关性（译注：听起来似乎有点拗口，但是英文表达意思就是如此），提升值为 2.7。 匿名函数导致的（非阻塞型错误）原因和数据私有化的修复策略具有第二高的提升值，提升值为2.23。 其次，误用chan与 Move 修复方式（译注：指通过chan发送或者接收操作的位置）也密切相关，提升值为2.21。

- **结论 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方式，而chan同时被广泛用于修复与产相关的错误以及共享内存错误。
- **推论 7：**虽然 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错误，在某些情况下，他们更喜欢使用消息传递作为修复方式，可能是因为他们认为消息传递是一种更安全的跨线程通信方式。

6.3 非阻塞错误检测

Go 提供了一个数据竞争检测器，它使用与 ThreadSanitizer [53] 相同的happen before算法（译注：很抱歉，对golang 运行时不是很了解，不知

道这是个什么算法，也不清楚怎么翻译合适）。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子字（shadow words）来存储对象的历史访问。 它将每次新访问与存储的影子字值进行比较以检测可能的竞争。

我们使用 20 个可复现的非阻塞错误来评估检测器可以检测到多少个。 我们在打开竞争检测器的情况下运行每个有问题的程序 100 次。 表 12 总结了在每种根本原因类别下检测到的错误数量。 竞争检测器没有误报。

数据竞争检测器成功检测到 7/13 的传统错误和 3/4的 由匿名函数引起的错误。 对于其中（检测）成功的6个，数据竞争检测器在每次运行时都会报告，而对于其余4个，在需要运行大约 100 次检测器才会报告bug。

数据竞争检测器未能报告许多非阻塞错误的原因可能有以下三种。 首先，并非所有非阻塞错误都有数据竞争； 检测器并非设计用于检测这些其他类型（译注：意思是数据竞争检测器不是被设计用于检测非数据竞争bug）。其次，底层happen-before算法的有效性取决于并发goroutines的交错执行。 最后，每个内存对象只有四个影子字，检测器不能保持很长的历史记录，可能会错过数据竞争。

- 推论8：简单的传统数据竞争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丰富（原文：more informative）、 Go特定 的非阻塞错误检测器。

7 讨论和展望（Discussion and Future Work)

Go 提倡线程轻量级和线程创建简单化，并提倡消息传递线程通信优于共享内存进程通信。 事实上，我们看到在 Go 程序中创建的 goroutines 比传统线程更多，并且 Go chan和其他消息传递机制也有重要用途。 但是，我们的研究表明，如果使用不当，这两种编程实践可能会导致并发错误。

共享内存vs消息传递

我们的研究发现，多线程程序中，消息传递并不一定会比共享内存更不容易出错（更少出错）。 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标准库相结合时，消息传递可能会导致很难被检测的阻塞错误。与共享内存同步相比，消息传递导致的非阻塞错误更少，而且令人惊讶的是，它甚至可用于修复由共享内存同步误用引起的错误。 我们相信消息传递提供了一种干净利落的线程间通信形式，并且在传递数据和信号方面也很有用。 但它们只有在正确使用才可发挥作用，这就要求程序员不仅要了解消息传递机制，还要了解 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可用于进行并发错误检测的bug代码模式。 作为初步工作，我们针对匿名函数引起的非阻塞错误构建了一个检测器（如图 8）。 我们的检测器已经发现了一些新的错误，其中一个已经被真实的应用程序开发人员确认。

更一般地说，我们相信静态代码分析，加上以前的死锁检测算法，对检测大多数由共享内存同步错误引起的 Go 阻塞错误方面仍然有用。 静态技术还可以帮助检测由chan和锁结合引起的错误，图 7 就是一个例子。

误用 Go 标准库会导致阻塞和非阻塞错误。 在我们的研究中，我们总结了误用 Go 标准库的几种模式。 检测器可以利用我们学到的模式来揭示以前未知的bug。

我们的研究还发现，违反 Go 的并发原语强制性规则是并发错误的主要原因之一。 我们可以尝试一种新颖的动态技术强制执行此类规则，并在运行时检测违规行为。

8 相关工作

关于现实世界的bug研究

有许多关于现实世界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bug对抗技术的设计。 据我们所知，我们的工作是第一个专注于 Go 并发bug的研究，也是第一个将访问共享内存导致的bug和传递消息导致的bug进行对比的研究。

阻塞型bug对抗研究

作为一个传统问题，在 C 和 Java 中有很多研究工作来解决死锁问题 [7, 28, 33–35, 51, 54, 55, 58, 59]。 我们的研究表明： 上述研究也对Go 中许多非死锁阻塞错误有用，但这不是这些技术的目标。 我们提出了一些技术来检测由误用chan引起的阻塞错误 [38, 39, 49, 56]。 但是，阻塞错误也可能是由其他原语引起的。 我们的研究揭示了许多阻塞型bug的代码模式，可以为未来的阻塞型bug检测技术提供基础。

非阻塞型bug对抗研究

许多之前的研究工作是为了检测、诊断和修复由同步共享内存访问失败而导致的非死锁错误 [4, 5, 8, 14, 16, 17, 30–32, 43, 46, 47, 52, 62–64]。 它们页可能适用于 Go 并发错误。 然而，我们的研究发现，消息传递错误导致的非阻塞型bug占比不可忽视，而这些错误并没有被以前的研究所涵盖。我们的研究强调：需要新技术来对抗消息传递过程中的bug。

9 结论

作为一种为并发而设计的编程语言，Go 提供了轻量级的 goroutines 和 goroutines 之间基于chan的消息传递机制。 随着 Go在越来越多的各类应用中使用，本轮文首次从两个正交维度对 171 个真实世界的 Go 并发错误进行了全面的实证研究。 我们的研究有许多有趣的发现和影响力。 我们希望：我们的研究能够加深对 Go 并发 bug 的理解，并让人们对 Go 并发 bug给予更多关注。