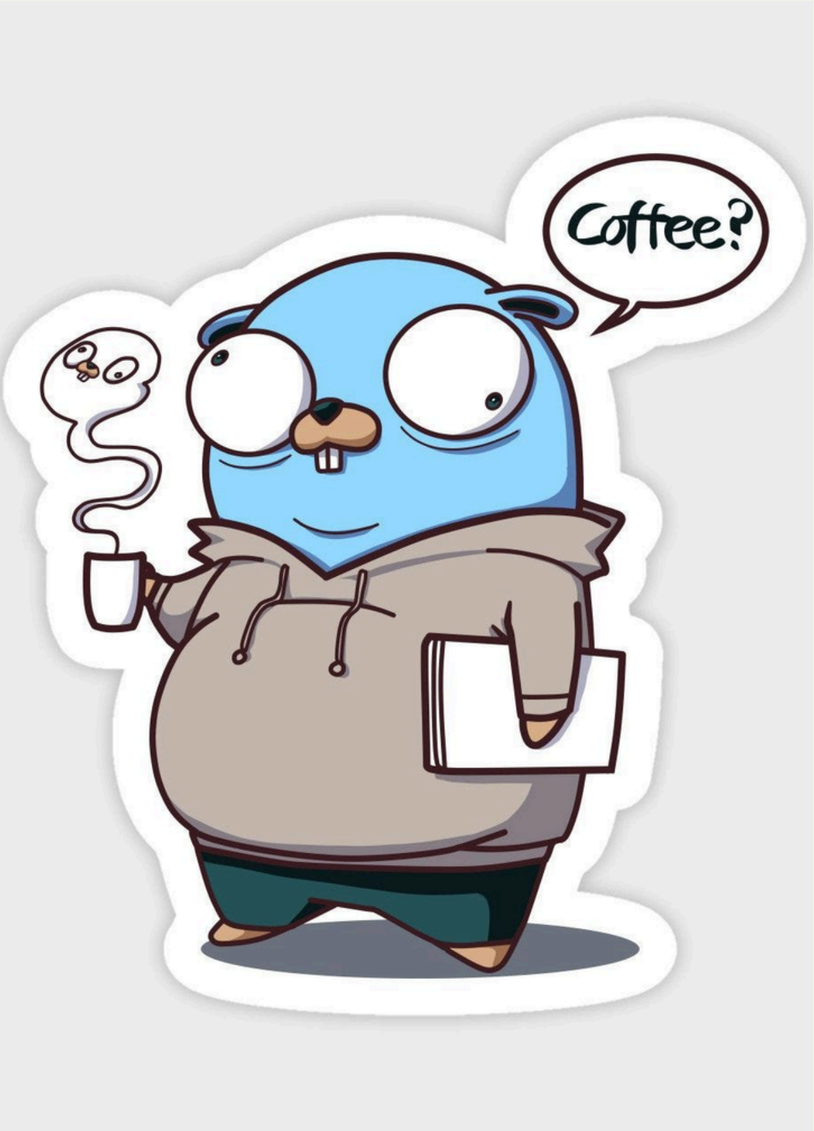
Tracing in Go

BECAUSE PRINT DEBUGGING IS A CRIME

TOWHID KHAN

Software Engineer (Cloud Team) @ pgEdge

WHO IS THIS GUY?



- 🚀 Absolute Go Fanatic ("Concurrency is my love language")
- ☁️ Software Engineer @ pgEdge Cloud Team ("Making Postgres distributed")
- 🎮 Ex-competitive Gamer (Now just stress-testing ranked matches in my free time)
- 🔍 Breaking infra just to fix it again—for science."

CONNECT WITH ME:

- 📧 Email: hello@yourtowhid.com
- 🌐 LinkedIn: linkedin.com/in/yourtowhid
- 🐦 Twitter (X): x.com/yourtowhid
- 📸 Instagram: instagram.com/yourtowhid
- 📘 Facebook: facebook.com/knockoutez
- 💻 GitHub: github.com/KnockOutEZ

A LITTLE ABOUT PGEDGE [PGEDGE.COM]

We don’t just work with Postgres—we supercharge it.
pgEdge powers global, mission-critical systems. Fast, reliable, always available—exactly where your data needs to be. (If your database could dream, it’d want to be on pgEdge.)

WHAT IS TRACE?

- A trace is like a detailed timeline of everything that happens in your program, with exact timestamps for each event.

- Events can be anything interesting that happens while your program runs - from function calls to memory usage to goroutine creation.

- Examples of events we'll look at:
  - Function calls and returns (what code is running?)
  - Memory allocation/deallocation (how is memory being used?)
  - Goroutine creation/completion (how are concurrent tasks managed?)
  - System calls (how does your program interact with the OS?)
  - Garbage collection cycles (when is memory being cleaned up?)

Summary: your program's "black box recorder" that helps you understand what your code is doing, when it's doing it, how long it takes, and where it might be having problems.

Is This Go Specific?

Not at all. Even though we're using Go for our examples today, this is a universal concept. Modern software environments often include a diverse set of tools and services, even in organizations that have standardized on a common stack. Each language ecosystem has developed its own tracing tools - VisualVM for Java, gprof for C++, pprof for Go, and so on. While these tools are powerful, they can be inconsistent, incomplete, and often unsafe to run on systems serving live traffic. They're great tools when used by people who deeply understand their applications and the risks involved, but can be inappropriate or even dangerous when used by those who don't fully understand the services or applications they're working with. What we're learning today are fundamental concepts about understanding program behavior that apply regardless of your technology stack - we're just using Go's tooling as our practical example.

SAMPLE PERFORMANCE PROBLEM - LET'S INVESTIGATE!

Let me show you a real example of how tracing can help us understand and improve program performance. I have a simple program that:

- Reads XML files containing news articles
- Searches for specific terms in the articles
- Counts how many times these terms appear

This looks like a straightforward program, but it's hitting some performance limits. Instead of guessing what's wrong, let's use tracing to actually see what's happening inside the program while it runs.

We'll:

- 1. Run the program with tracing enabled
- 2. Look at what the trace tells us
- 3. Use that information to make improvements
- 4. Compare the results

This hands-on example will show you exactly why tracing is such a powerful tool for understanding program behavior.

Source code can be found in: **github.com/KnockOutEZ/tracingingo**

WHAT DID WE LEARN TODAY?

Tracing gave us concrete data about our program's behavior

We identified bottlenecks without guessing

We could verify our optimizations actually worked

Sequential I/O operations were killing our performance

Q&A Session

THANK YOU FOR LISTENING!

SOCIALS:

- 📧 Email: hello@yourtowhid.com
- 🌐 LinkedIn: linkedin.com/in/yourtowhid
- 🐦 Twitter (X): x.com/yourtowhid
- 📸 Instagram: instagram.com/yourtowhid
- 📘 Facebook: facebook.com/knockoutez
- 💻 GitHub: github.com/KnockOutEZ