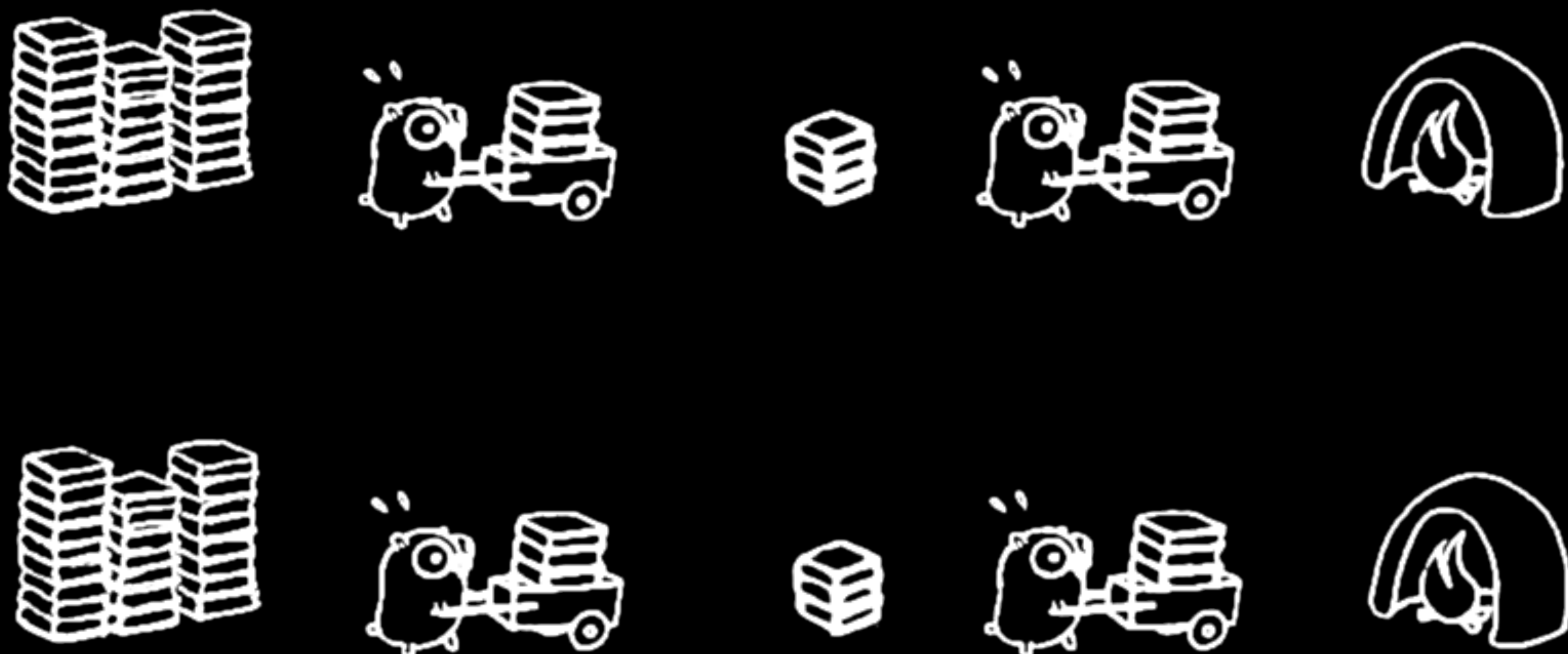
Principles of designing Go APIs with channels

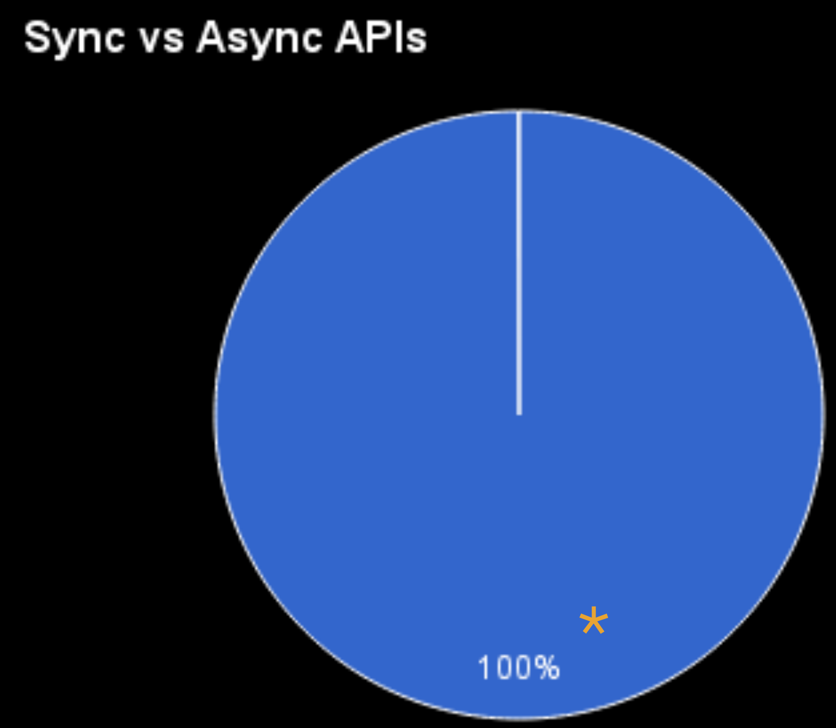
Alan Shreve
Keen IO
@inconshreveable

First-class concurrency support in Go



- APIs in stdlib without `chan`: ~ 30,000
- APIs in stdlib with `chan:` ~10

APIs in the Standard Library



*rounded up

io.Reader

```
type Reader {
    Read([]byte) (int, error)
}
```

Where is io.AysncReader?

```
type ReadResult struct {
    n    int
    err error
}

type AsyncReader {
    ReadAsync([]byte) <-chan ReadResult
}
```

APIs should be synchronous (blocking).

Principle 0

- Concurrency in Go is easy
- Leave concurrency decisions to the caller
  - Easier for you
  - More flexible for the caller
- Reduces API surface area (no additional Async APIs)
- *Your API still needs to be thread-safe.*
- **Most important takeaway of this talk**

```
n, err = conn.Read(buf)
```

```
reads := make(chan readResult)
go func() {
    n, err = conn.Read(buf)
    reads <- readResult{n, err}
}()
```

So - when is it appropriate to have a channel in your API?

let's come back to this later

An API should declare the directionality of its channels.

Principle 1

```
func After(d Duration) <-chan Time
```

```
func Notify(c chan<- os.Signal,
    sig ...os.Signal)
```

“The optional <- operator specifies the channel direction, send or receive. If no direction is given, the channel is bidirectional.”

– *The Go Programming Language Specification*

- Proper API usage enforced by the compiler
- Type signature elucidates API data flow and proper usage

This will not compile

```
t := time.After(time.Second)

t <- time.Now()
```

```
send to receive-only type <-chan Time
```

An API that sends an **unbounded** stream of values into a channel **must** document how it behaves for slow consumers.

Principle 2

```
// NewTicker returns a new Ticker containing
// a channel that will send the
// time with a period specified by the
// duration argument.
// It adjusts the intervals or drops ticks
// to make up for slow receivers.
// ...
func NewTicker(d Duration) *Ticker {
    ...
}
```

```go
// Notify causes package signal to relay
// incoming signals to c.
// ...
// Package signal will not block sending to c
// ...
func Notify(c chan<- os.Signal, sig ...os.Signal) {
	...
}
```

```
// OpenChannel tries to open an channel.
// ...
// On success it returns the SSH Channel
// and a Go channel for incoming, out-of-band
// requests. The Go channel must be serviced, or
// the connection will hang.
OpenChannel(name string, data []byte) (Channel, <-
chan *Request, error)
```

- Implementation has a choice when sending values into a channel that is full:
  - Block
  - Don’t block

    ```
    select {
    case c<-val:
    default:
    }
    ```

- No language annotation, must be documented.

An API that sends a **bounded** set of values into a channel it *accepted as an argument* **must** document how it behaves for slow consumers.

Principle 3

```go
func (client *Client) Go(serviceMethod string,
                         args interface{},
                         reply interface{},
                         done chan *Call
                         ) *Call
```

- Only sends one value into the channel
- Channel was an argument, could still be full
- No documented behavior

```
func (call *Call) done() {
	select {
	case call.Done <- call:
		// ok
	default:
		// We don't want to block here.
		// It is the caller's responsibility to make
		// sure the channel has enough buffer space.
		// See comment in Go().
		if debugLog {
			log.Println("rpc: discarding Call reply due to insufficient Done chan capacity")
		}
	}
}
```

An API that sends a **bounded** set of values may do so safely by returning an appropriately buffered channel.

Principle 4

```go
type CloseNotifier interface {
        // CloseNotify returns a channel that
        // receives a single value
        // when the client connection has gone
        // away.
        CloseNotify() <-chan bool
}
```

```go
func After(d Duration) <-chan Time
```

- Returned channel can be buffered to maximum number of values to send
- Sending on the channel guaranteed to never block

Act II

Tradeoffs

An API sending an **unbounded** stream of values must trade off between *accepting the channel as an argument* and *returning a new channel*.

Tradeoff 0

```go
func Notify(c chan<- os.Signal, sig ...os.Signal)
```

VS

```go
func Notify(sig ...os.Signal) <-chan os.Signal
```

- why doesn't `signal.Notify` allocate the channel for you?
- would have to choose channel buffer size
  - memory tradeoff
  - tradeoff tolerance of missed signals
- why not this then:
  - `func Notify(sig ...os.Signal, size int) <-chan os.Signal`
  - caller loses ability to use the same channel for handling (one goroutine)

```
sigs := make(chan os.Signal, 10)
go handleSigs(sigs)
signal.Notify(sigs, os.Interrupt)

// later …
signal.Notify(sigs, os.Kill)
```

but on the other hand . . .

```go
func NewClientConn(c net.Conn,
                   addr string,
                   config *ClientConfig)
(Conn, <-chan NewChannel, <-chan *Request, error)
```

VS

```go
func NewClientConn(c net.Conn,
                   addr string,
                   config *ClientConfig,
                   newChans chan<- NewChannel,
                   newReqs chan<- *Request)
                   (Conn, error)
```

- Returned channel makes it easy to know when an SSH connection is finished
    - The channels close - need another mechanism when channel is an argument
- Returned channel grants compiler protection against send on closed channel

```
var c <-chan
close(c)
```

    - close(c) (cannot close receive-only channel)

So - when is it appropriate to have a channel in your API?

What do these APIs all have in common?

```
func After(d Duration) <-chan Time
```

```
func Notify(c chan<- os.Signal,
        sig ...os.Signal)
```

```
type CloseNotifier interface {
        CloseNotify() <-chan bool
}
```

```
func (client *Client) Go(serviceMethod string,
                         args interface{},
                         reply interface{},
                         done chan *Call
                         ) *Call
```

- When a timer fires
- When the process receives a signal
- When a client closes the HTTP connection
- When a response is received to an RPC call
- When a new SSH channel is opened

Asynchronous notification of events

A challenger appears

```
func ListenAndServe(addr string,
                    handler Handler) error
```

Callbacks, in my Go code?

It’s more likely than you think.

Any API written with a channel could use a callback instead.

Tradeoff 1

Mirror Universe

Mirror Universe

```
func ListenAndServe(addr string)
     (chan *http.ReqResp, error)
```

```
func Notify(handler func(os.Signal),
            sig ...os.Signal)
```

Tradeoffs

- Callback overhead is a function call; channel has synchronization overhead
- Callbacks can be wrapped (recover from panic, behavior after completion)
- Trivial to make a callback that sends into a channel
- Interface callback make for easy composition

Tradeoffs

- Channels are more idiomatic than callbacks
- Channels make code-flow easier to reason about
- I just left Node.js . . .

Takeaways

- Always prefer synchronous APIs if possible
- Declare the directionality of your channels
- Document the behavior of your APIs in the presence of slow consumers

Takeaways

- Weigh tradeoff between channel as an argument vs channel as a return value
- Consider using callbacks instead of channels for notifying your caller of async events

Questions

Alan Shreve
Keen IO
@inconshreveable