Go Tool Belt

Everyday tools used at Crowdstrike

30 June 2016

Sean Berry
Principal Engineer, Crowdstrike

The Go Tool Belt

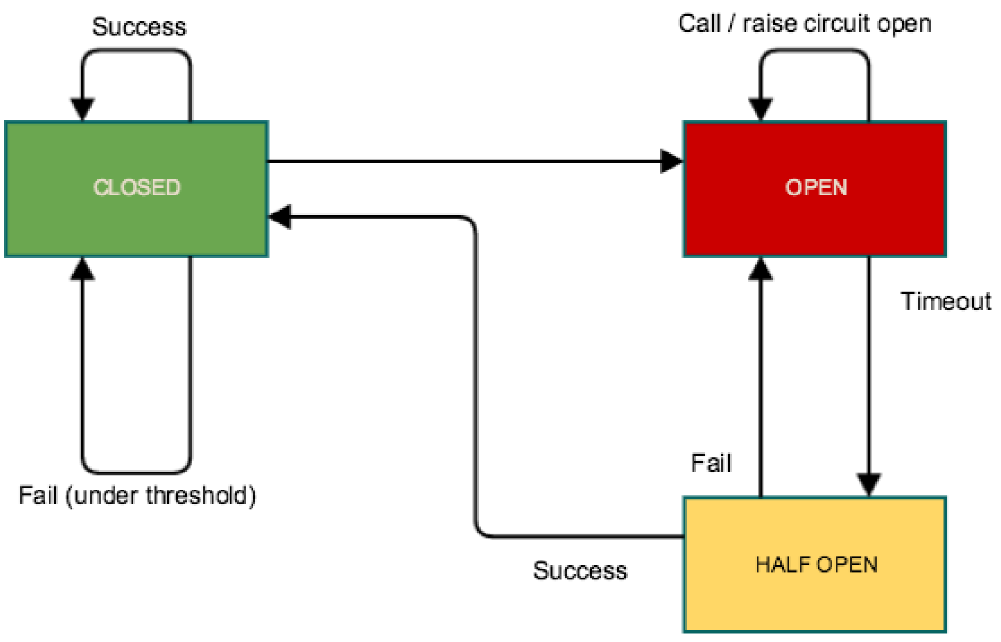
Tools we will cover

- Circuit Breakers
- Retriers
- Deadlines
- Caches
- context.Context
- Rate Limiting

Circuit Breaker

Circuit Breaker

github.com/sony/gobreaker (http://github.com/sony/gobreaker)



Circuit Breaker

- Circuit starts in a closed state
- When the error threshold is reached the circuit opens
- After a configurable amount of time the circuit goes half-open
- A request is made in the half-open state
- If the request succeeds the cicuit closes
- If it fails we reset our timer and go back to open

Circuit Breaker Setup

```
    // START SETUP
    breakerSettings := gobreaker.Settings{
        Name:    "Request local resource",
        Timeout: 5 * time.Second,
        OnStateChange: func(name string, from gobreaker.State, to gobreaker.State)
            fmt.Printf("State Change %s --> %s\n", state(from), state(to))
        },
    }
    breakerSettings.ReadyToTrip = func(counts gobreaker.Counts) bool {
        failureRatio := float64(counts.TotalFailures) / float64(counts.Requests)
        return (counts.Requests > 5 && failureRatio > 0.4) || counts.ConsecutiveFa
    }
    breakerSettings.MaxRequests = 2
    breaker := gobreaker.NewCircuitBreaker(breakerSettings)
    // END SETUP
```

Circuit Breaker Running

```
// START CODE
body, err := breaker.Execute(func() (interface{}, error) {

    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()
    body, _ := ioutil.ReadAll(resp.Body)
    if resp.StatusCode == 400 {
        return nil, errFailedResponseCode
    }
    return body, nil
})
// END CODE
```

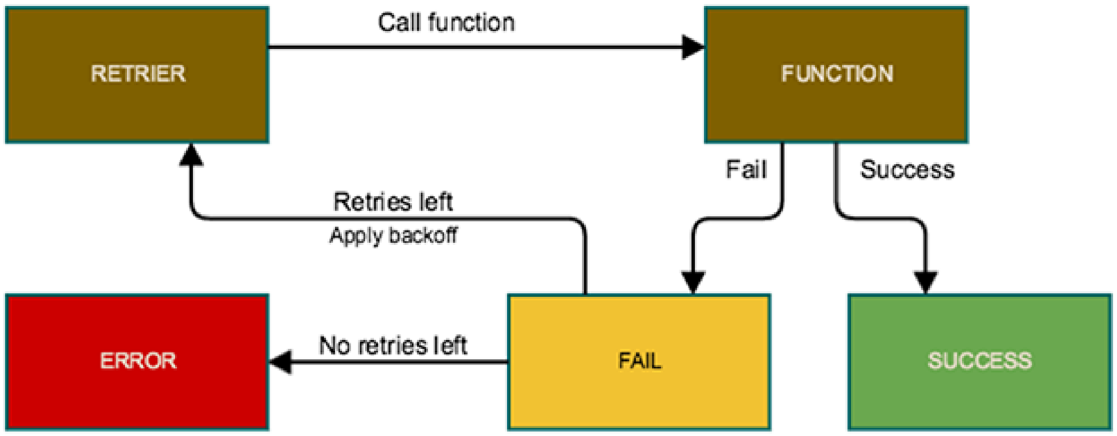
Circuit Breaker Use Cases

- You need to keep throughput high
- External service may be unreliable
- You have an alternative method during open circuit
- Avoid putting more pressure on a struggling system
- Avoid waiting for network timeouts

Retriers

Retriers

github.com/eapache/go-resiliency (https://github.com/eapache/go-resiliency)



Retriers

- Determine a backoff strategy for failure
- Determine a max number of times we will try
- Determine **whitelist** errors

Some examples of errors you may want to whitelist:

- Decoding / Unmarshalling erros when the source won't change
- Permission Violations
- Rate limiting error

Retriers Setup

```
    // START SETUP
    retry := retrier.New(retrier.ConstantBackoff(2, 10*time.Millisecond), nil)
    // END SETUP
```

Options for backoff strategy:

- ConstantBackoff will do retries at N, 2N, 3N, 4N, ... XN times
- ExponentialBackoff will do retries at N, 2N. 4N, 8N, ... 2^(X-1)N times

Retriers Running

- Hard Failures - failed twice in a row
- Failures - failed first call
- Success - succeeded within two calls

```
        // START CODE
        var body []byte
        reqErr := retry.Run(func() error {
            resp, err := http.Get(url)
            if err != nil {
                return err
            }
            defer resp.Body.Close()
            body, _ = ioutil.ReadAll(resp.Body)
            if resp.StatusCode == 400 {
                tempFailures++
                return errFailedResponseCode
            }
            return nil
        })
        // END CODE
```

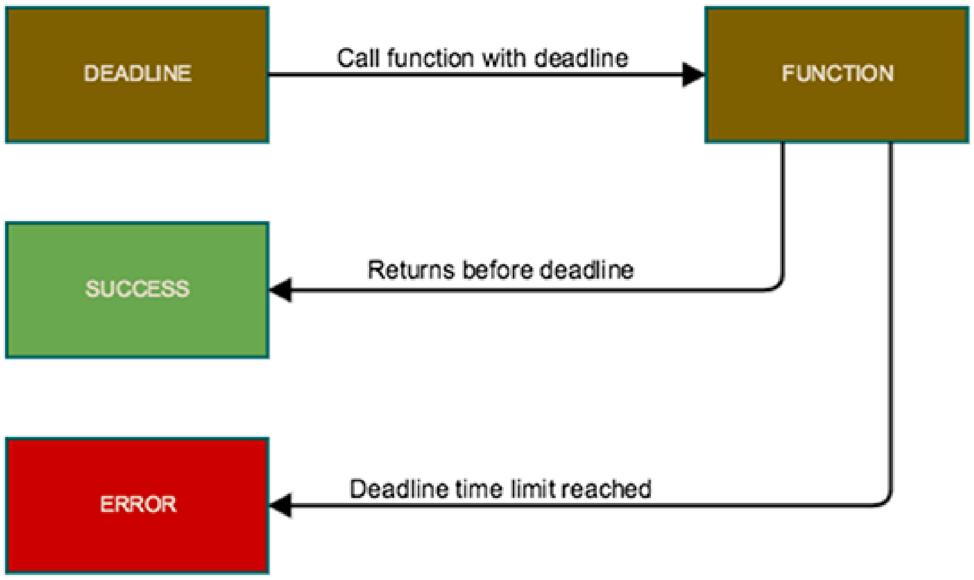
Retriers Use Cases

- Downstream service may fail
- Downstream service implements rate limiting
- Success of request is paramount
- Still good to have alternative on failure

Deadlines

Deadlines

github.com/eapache/go-resiliency (https://github.com/eapache/go-resiliency)



Deadline

- Function call is required to compelete before timeout
- If response comes before timeout, all good
- Otherwise a deadline.ErrTimedOut is received
- Can use closures

Deadline Setup

```
    // START SETUP
    dl := deadline.New(1 * time.Second)
    // END SETUP
```

Deadline Running

```
        err := dl.Run(func(stopper <-chan struct{}) error {
            resp, err := http.Get(url)
            if err != nil {
                return err
            }
            defer resp.Body.Close()
            body, _ = ioutil.ReadAll(resp.Body)
            if resp.StatusCode == 400 {
                return errFailedResponseCode
            }
            return nil
        })

        delta := time.Since(tStart).Nanoseconds() / 1e6
        switch err {
        case deadline.ErrTimedOut:
            fmt.Printf("Timeout error: %d ms\n", delta)
        case nil:
            fmt.Printf("Request response: %s, %d ms\n", string(body), delta)
        default:
            fmt.Printf("Some other error: %s, %d ms\n", err, delta)
        }
```

Deadline Use Cases

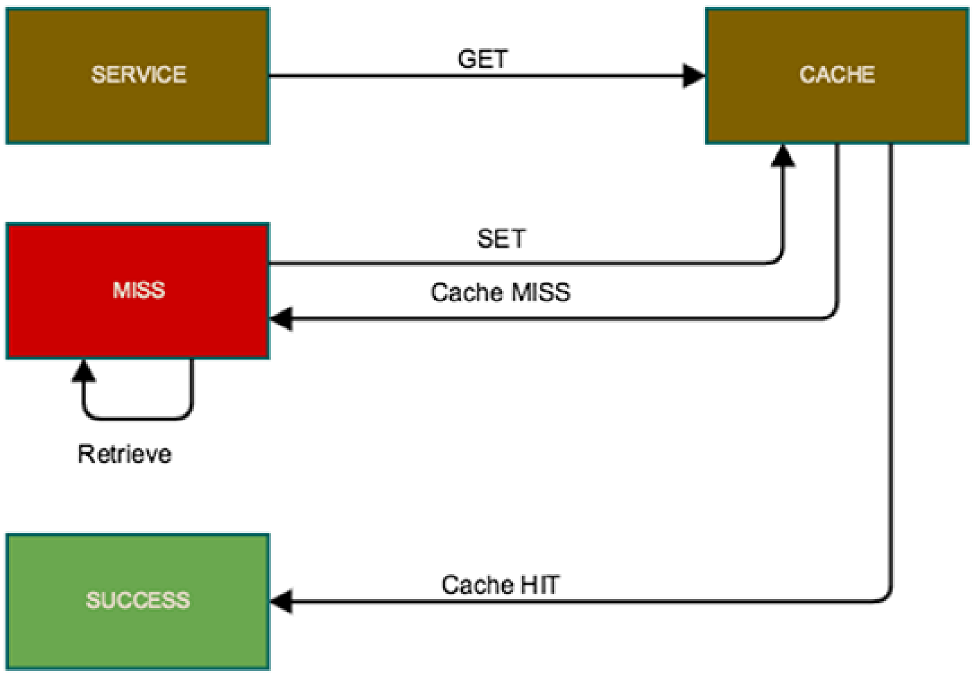
- SLA
- Complete processing time needs to be shorter than X ms
- Individual request times need to be shorter than Y ms

Caches

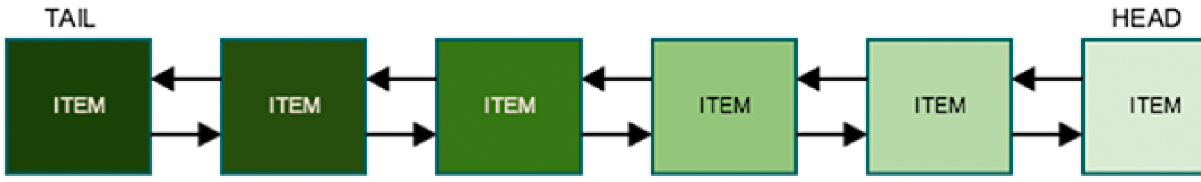
github.com/hashicorp/golang-lru (https://github.com/hashicorp/golang-lru)

github.com/patrickmn/go-cache (http://github.com/patrickmn/go-cache)

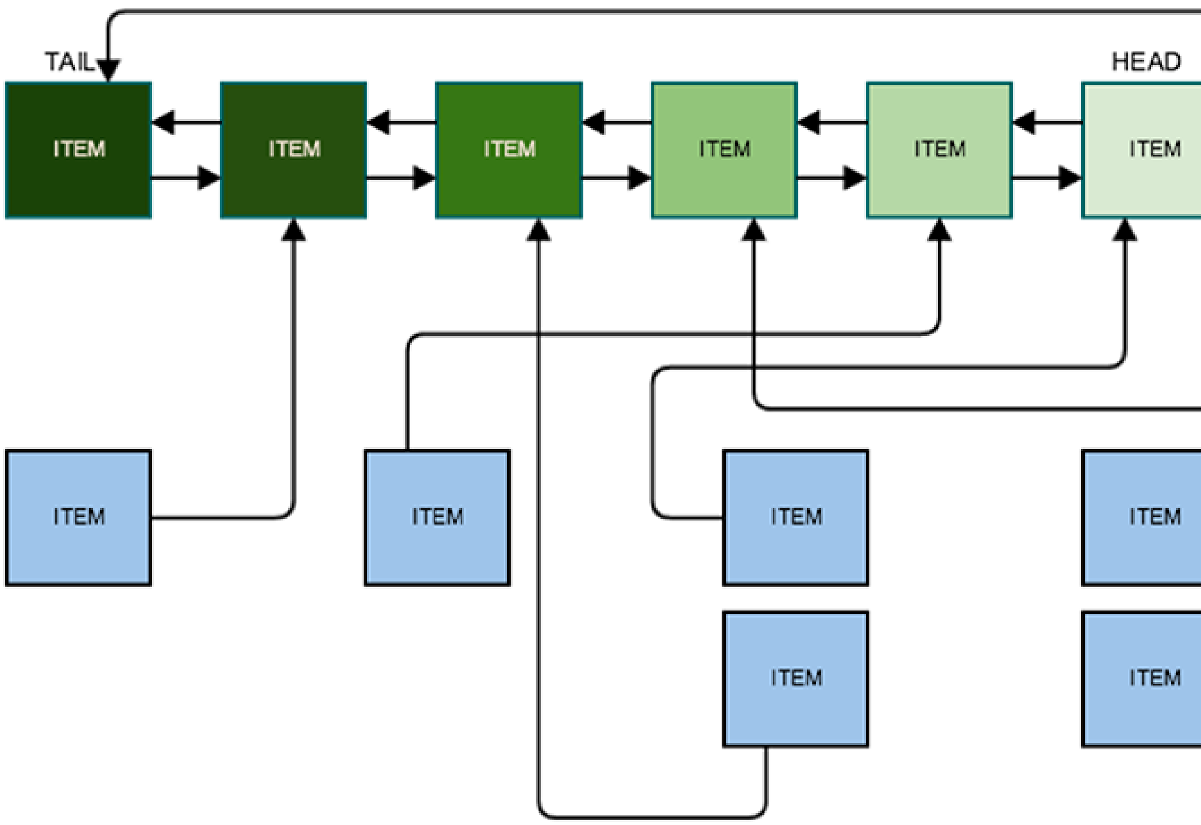
Caches


SERVICE
GET
CACHE
SET
MISS
Cache MISS
Retrieve
SUCCESS
Cache HIT

Caches - LRU



Caches - LRU


TAIL
HEAD
ITEM
ITEM
ITEM
ITEM
ITEM
ITEM
ITEM
ITEM
ITEM
ITEM
ITEM
ITEM

Caches - LRU

- Standard LRU
- ARC Cache
- Q2 Cache

Caches - LRU

```
func main() {
    cache, _ := lru.New(5)
    for i, key := 0, 0; i < 100; i, key = i+1, i%5 {
        if res, ok := cache.Get(key); ok {
            fmt.Printf("Got item %d from cache\n", res)
            continue
        }
        item := getSlowThing(key)
        fmt.Printf("Adding %d to cache\n", item)
        cache.Add(key, item)

    }
    fmt.Printf("Cache size: %d\n", cache.Len())
    time.Sleep(1 * time.Second)
}
```

context.Context

context.Context

- github.com/golang/net/tree/master/context(https://github.com/golang/net/tree/master/context)

context.Context

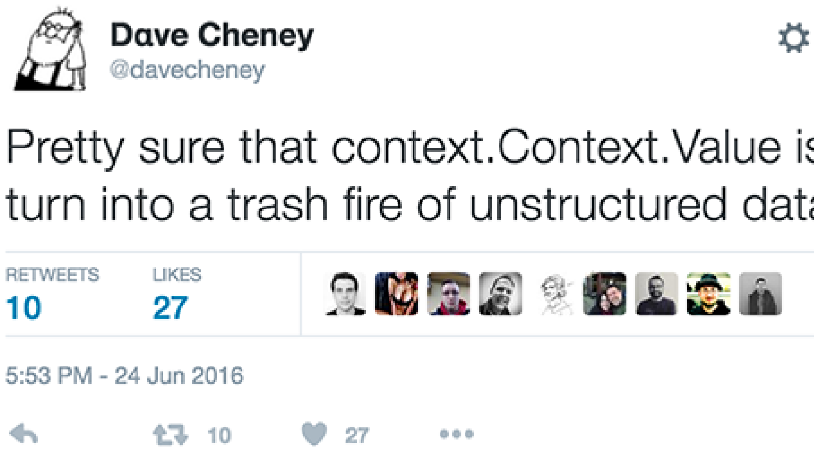
Create a context at the start of a request and propogate througout the re
lifetime

```
func WithValue(parent Context, key interface{}, val interface{}) Context {
  return &valueCtx{parent, key, val}
}
```

Start with the base Context, context.Background()

```
func (c *valueCtx) Value(key interface{}) interface{} {
  if c.key == key {
    return c.val
  }
  return c.Context.Value(key)
}
```

context.Context



Dave Cheney
@davecheney

Pretty sure that context.Context.Value is
turn into a trash fire of unstructured dat

context.Context

```go
func main() {
    rand.Seed(time.Now().UTC().UnixNano())
    ctx := context.Background()
    ctx, cancel := context.WithTimeout(ctx, 200*time.Millisecond)
    defer cancel()
    result := make(chan int, 2)
    wg.Add(1)
    go doSomething(ctx, result)
    select {
    case <-ctx.Done():
        fmt.Println("We give up")
    case c := <-result:
        fmt.Println("Work complete.  Answer is", c)
    }
    wg.Wait()
    time.Sleep(10 * time.Millisecond)
}
```

context.Context

```
func main() {
    rand.Seed(time.Now().UTC().UnixNano())
    ctx := context.Background()
    ctx, cancel := context.WithTimeout(ctx, 200*time.Millisecond)
    defer cancel()
    result := make(chan int, 2)
    wg.Add(1)
    go doSomething(ctx, result)
    select {
    case <-ctx.Done():
        fmt.Println("We give up")
    case c := <-result:
        fmt.Println("Work complete.  Answer is", c)
    }
    wg.Wait()
    time.Sleep(10 * time.Millisecond)
}
```

Rate Limiting

github.com/CrowdStrike/ratelimiter (https://github.com/CrowdStrike/ratelimiter)

Rate Limiting Setup

```
maxCapacity := 1000
ratePeriod := 10 * time.Second
rl, err := ratelimiter.New(maxCapacity, ratePeriod)
if err != nil {
    fmt.Printf("Unable to create cache")
}
```

Rate Limiting Running

```
func main() {
    maxCapacity := 1000
    ratePeriod := 10 * time.Second
    rl, err := ratelimiter.New(maxCapacity, ratePeriod)
    if err != nil {
        fmt.Printf("Unable to create cache")
    }
    userKey := "sean"
    maxCount := 100 // the maximum number of items I want from this user in ten se

    for {
        if cnt, underRateLimit := rl.Incr(userKey, maxCount); underRateLimit {
            fmt.Printf("%s is making request. %d requests made\n", userKey, cnt)
            time.Sleep(50 * time.Millisecond)
        } else {
            fmt.Printf("%s is over rate limit, current count [%d]\n", userKey, cnt
            time.Sleep(1 * time.Second)
        }
    }

}
```

Rate Limiting Use Cases

- API Access
- Downstream service DOS protection
- Resource protection

Rate Limiting - Really just an LRU

Our rate limiter is based entirely off of the Hashicorp LRU library

- Each unique identifier for rate-limiting is a cache item
- The value of the cached item contains a counter and time
- Inspect the counter and check vs the max number within a time perio
- Note: need to have > max users of cache or you are still open to DOS

Honorable Mentions

Facebook RPool (https://github.com/facebookgo/rpool)

Ginkgo (https://github.com/onsi/ginkgo)

Gomega (https://github.com/onsi/gomega)

Go-Restful (https://github.com/emicklei/go-restful)

Sarama (https://github.com/Shopify/sarama)

HttpControl (https://github.com/facebookgo/httpcontrol)

Errors (https://github.com/pkg/errors)

Thank you

Sean Berry
Principal Engineer, Crowdstrike
sean@crowdstrike.com (mailto:sean@crowdstrike.com)
http://crowdstrike.com/ (http://crowdstrike.com/)
@schleprachaun (http://twitter.com/schleprachaun)