目次

第 I 部　構造化と性能の間を Golang で攻める技術 5

第 1 章　本書の目的 6

1.1 性能と構造化の間 6
1.2 本書の構成 8
1.3 実験環境 8
1.4 本書のクラス図について 8

第 2 章　1M op/s で何ができるか？　Go のベンチマークを使う 10

2.1 Go のベンチマーク及び解析の方法 10
2.2 シングルスレッドによる 1M op/s 13
実験 2.1 1M op/s で実行できる圧縮 13
実験 2.2 1M op/s で実行できるソート 14
実験 2.3 毎回メモリアロケーションを行う場合 16
まとめ 17
2.3 マルチスレッドによる 1M op/s 17
実験 2.4 マルチコアでの圧縮 17
まとめ 20

第 3 章　インターフェイスと関数型による構造化 21

3.1 他の言語とのインターフェイスの仕様の違い 21
3.2 構造化: 3 層レイヤー構造について 22
3.3 構造化: GoF デザインパターンについて 23
Strategy パターン 23
3.4 インターフェイスの実態と操作 25
3.5 関数型 27
実験 3.1 関数型を用いた Strategy パターンのコスト 28
3.6 インターフェイスのコスト 29
実験 3.2 多重にインターフェイスを呼び出すコスト 29

実験 3.3 インターフェイスを用いた Strategy パターンのコスト 34
3.7 インターフェイスを戻り値にするコスト 35
実験 3.4 インターフェイスを戻り値にするコスト 35
3.8 まとめ 38

第 4 章 Go ルーチンによる構造化 39

4.1 処理がパラレルではなく、コンカレントであるとはどういうことか 39
4.2 Go ルーチンは軽量なのか 39
実験 4.1 Go ルーチン、Linux OS のスケジューリング 40
4.3 Go ルーチンのコンテキストスイッチのコスト 41
実験 4.2 Go ルーチンのコンテキストスイッチのコスト 41
4.4 コンカレンシーの構造化 44
4.5 Go ルーチンを使い捨てにできるか 44
実験 4.3 Go ルーチンを使い捨てるコスト 44
4.6 チャネル 51
チャネルの仕組み 51
実験 4.4 Go ルーチン間のチャネルによるデータの受け渡しコスト 52
4.7 呼び出し元へ非同期に結果を戻すチャネルを返す 56
実験 4.5 非同期実行の管理コスト 57
4.8 非同期での参照の終了にコンテキストを使う 59
実験 4.6 context.Context によるリソースの解放 61
4.9 まとめ 62

第 5 章 Sync による構造化 63

5.1 sync.Once で Singleton な生成を行う 63
実験 5.1 sync.Once を用いた初期化 63
5.2 sync.Mutex で Singleton なオブジェクトを管理する 65
実験 5.2 sync.Mutex を用いた排他制御のコスト 65
実験 5.3 sync.RWMutex を用いた排他制御のコスト 67
5.3 sync.Pool で少ないリソースを節約する 69
実験 5.4 sync.Pool でリソースを管理する 70
5.4 sync.Cond でオブザーバーパターンを作る 72
実験 5.5 sync.Cond を用いたオブザーバパターン 75
5.5 まとめ 75

第 6 章 マイクロサービス化 77

6.1 マイクロサービスの定義 77
6.2 シリアライズされたデータのパース 78
実験 6.1 パース 78

6.3 同一コンピュータ内でのプロセス間通信 79
サーバの Go ルーチンの実装について 79
実験 6.2 サーバの Go ルーチンの実装 80
プロセス間通信のプロトコル 87
実験 6.3 プロセス間通信 コールの度に新しい接続を確立するケース 87
実験 6.4 プロセス間通信 1 度確立した接続を利用して、継続して相互通信を行うケース 88
6.4 同一データセンター内での通信 89
実験 6.5 同一データセンター内での通信 90
6.5 まとめ 90
6.6 最後に 91

第 II 部 WebWorker 活用技術 92

第 7 章 Web Worker を利用した処理性能の向上 93

7.1 前置き 93
目的 93
方法 93
対象読者 93
検証環境 94
登場人物 94
Web Worker の使用方法 95
この節のまとめ 102
7.2 プログラムの実装と検証 103
流れ 103
クラス設計 103
各種意思決定 104
検証と課題の洗い出し、解決策の検討 110
7.3 結論とまとめ 116

著者紹介 118

74th 118
こばたく 118

第 I 部

構造化と性能の間を Golang で攻める技術

- コンテキストスイッチに、スタックサイズや Go ルーチンの数は関係がない。

4.4 コンカレンシーの構造化

4.3 節の実験により Go ルーチンのコンテキストスイッチのコストは十分に小さいことがわかった。Go ルーチンに仕事を割り振るには、以下の 2 つが考えられる。

- 仕事を割り振るたびに Go ルーチンを起動する。
- チャネルを仲介し、Go ルーチンにチャネルを受信させる。

また、仕事を別の Go ルーチンで行ったとしても、結果を依頼元の Go ルーチンに返すことや、依頼元に仕事が終わったことを通知する必要がある。それには、以下の 3 つの方法が考えられる。

- 依頼元の Go ルーチンに、チャネルを返し、そのチャネルを介してデータを戻す。
- sync.WaitGroup を依頼元に返し、依頼元はそれを用いて待ち状態になる。依頼先が終了したときに Done() を実行し、依頼元の待ち状態を解く。
- 別の Go ルーチンに対して、仕事が続いていることを示すコンテキスト context.Context を依頼元に戻す。

本章ではチャネル、コンテキストについて扱い、sync については次章で扱う。

4.5 Go ルーチンを使い捨てにできるか

Go ルーチンの多数を実行しなければ複数の CPU コアを使い切る事はできない。その一方で、一度 Go ルーチンを開始すると、その Go ルーチンを正しく終了させるまで Go ルーチンは生き続け、メモリを圧迫する。可能であれば、実行ごとに新しい Go ルーチンを実行し、一つの処理が終わるごとにその Go ルーチンを終了させ、それにより Go ルーチンの開始と停止に気を使うことがなくなる。Go ルーチンの生成のコストを求め、Go ルーチンを使い捨てにできるかを考える。

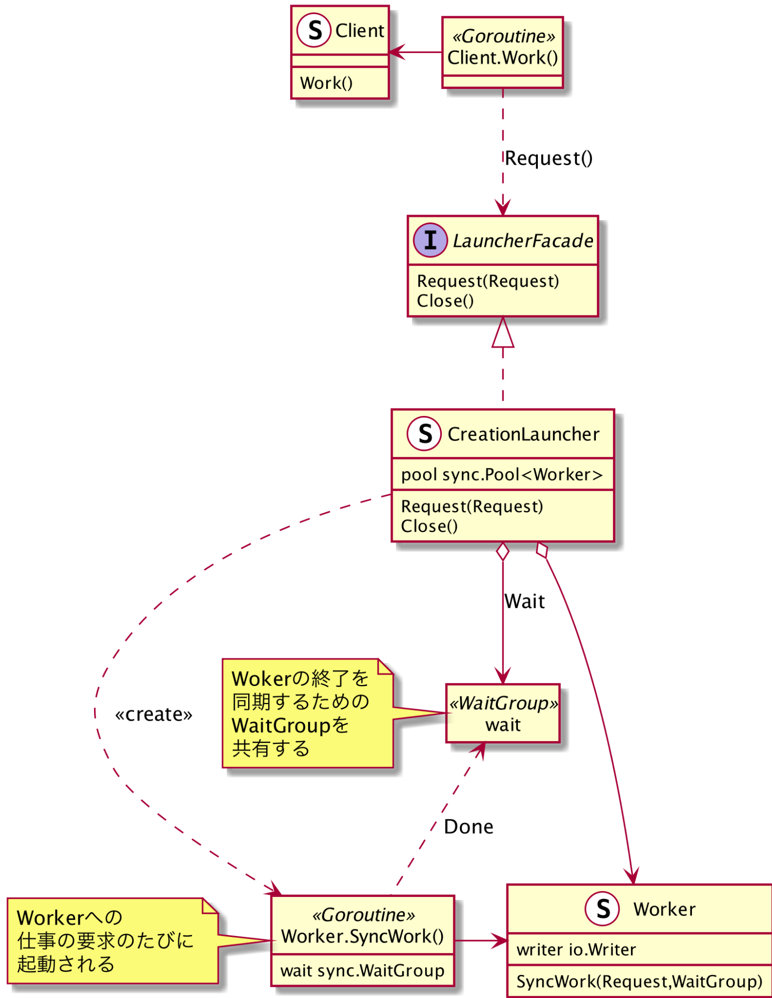
実験 4.3 Go ルーチンを使い捨てるコスト

実験目的

- Go ルーチンを新しく実行するコストを求め、1 つの仕事ごとに Go ルーチンを使い捨てにできるか判定する。

実験方法

仕事の依頼 Request() をするたびに Go ルーチンを起動し仕事を行うモジュールを作成した。このクラス図を図 4.2 に、コードを以下に示す。



▲図 4.2　Go ルーチン使い捨てる実験のクラス図

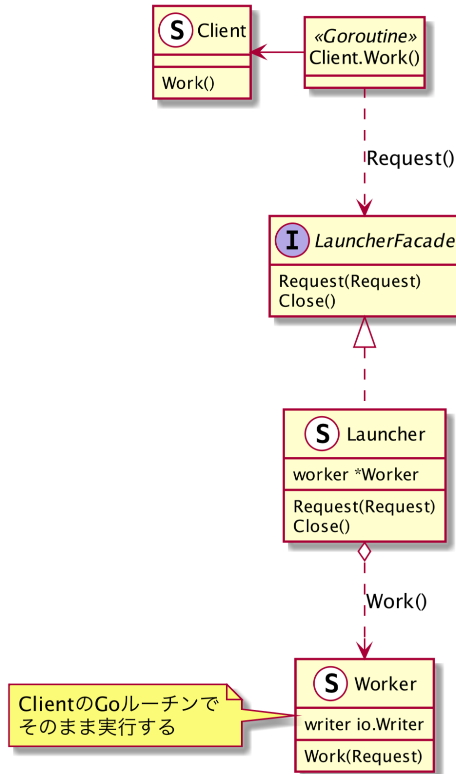
▼リスト 4.2　リクエストの度に Go ルーチンを起動する CreationLauncher

```
// LauncherFacade 仕事の要求を受け付ける
type LauncherFacade interface {
  Request(Request)
  Wait()
}
```

```
// CreationLauncher リクエストの度に Go ルーチンを起動するラウンチャー
type CreationLauncher struct {
  pool sync.Pool
  wait sync.WaitGroup
}

// NewCreationLauncher リクエストの度に Go ルーチンのワーカーを起動するランチャーを起動
する
func NewCreationLauncher(ctx context.Context, newWorker func() *Worker)
    LauncherFacade {
  l := new(CreationLauncher)
  l.pool.New = func() interface{} {
    return newWorker()
  }
  for i := 0; i < 16; i++ {
    // 事前に 16 ワーカーを作成しておく
    l.pool.Put(l.pool.New())
  }
  return l
}

// Request 仕事の要求
func (l *CreationLauncher) Request(req Request) {
  w := l.pool.Get().(*Worker)
  l.wait.Add(1)
  // 1 つのリクエストごとに Go ルーチンを起動する
  go func(w *Worker) {
    w.work(req, &l.wait)
    l.pool.Put(w)
  }(w)
}

// Wait すべてのワーカが終了するまで待つ
func (l *CreationLauncher) Wait() {
  l.wait.Wait()
}
```

Go ルーチンを起動後に、仕事が終わったことを確認するため wait を渡している。また、一度使った Woker を再利用するため、sync.Pool を利用している。

比較として、Go ルーチンの生成を行わず Launcher.Request() を実行した Go ルーチンと同じ Go ルーチンで仕事を行うモジュール Launcher を作成した。このクラス図を図 4.3 に、コードを以下に示す。



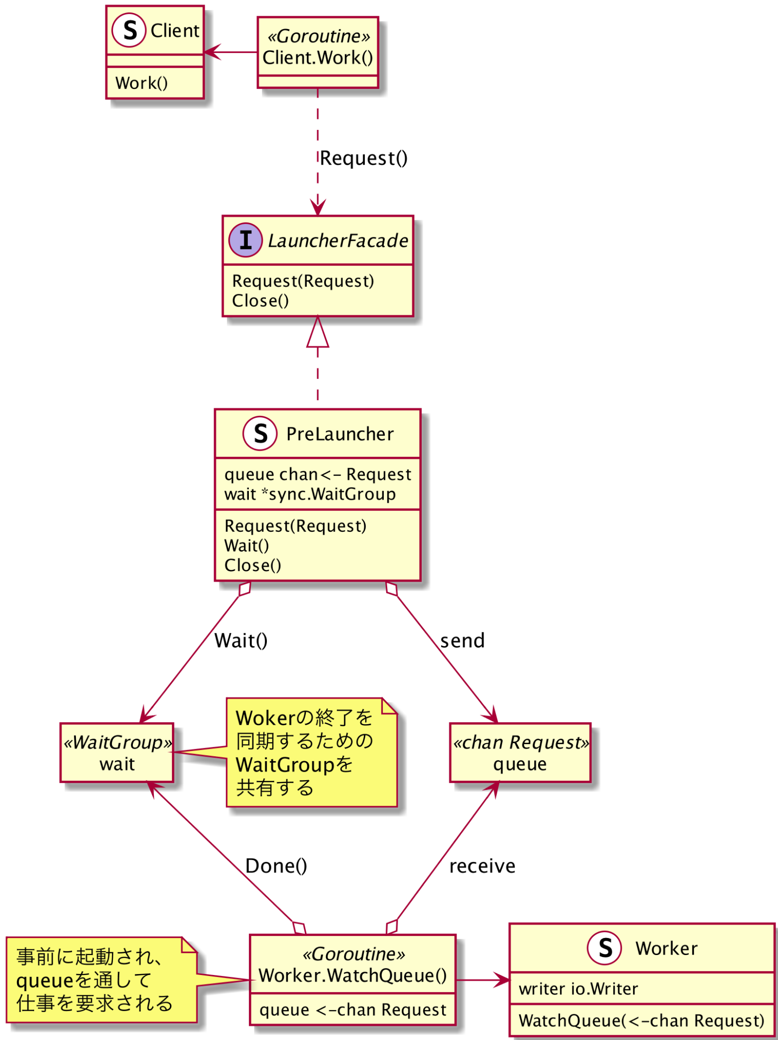
▲図 4.3　同じ Go ルーチンで仕事をする Launcher のクラス図

▼リスト 4.3　同じ Go ルーチンで仕事をする Launcher

```
// Launcher 同じ Go ルーチンでワーカを実行するラウンチャー
type Launcher struct {
  worker *Worker
}

// NewLauncher 同じスレッドでワーカーを実行するランチャーを起動する
func NewLauncher(ctx context.Context, newWorker func() *Worker)
    LauncherFacade {
  l := new(Launcher)
  l.worker = newWorker()
  return l
}

// Request リクエスト
func (l *Launcher) Request(req Request) {
```

```
    // 元の Go ルーチンで実行する
    l.worker.work(req)
}
```

更に比較として、事前に Go ルーチンを起動し、次節で触れるチャネルを使って Go ルーチンに仕事を渡すモジュール PreLauncher を作成した。このクラス図を図 4.4 に、コードを以下に示す。



▲図 4.4　Go ルーチンを事前に生成する PreLauncher のクラス図

▼リスト 4.4　Go ルーチンを事前に生成する PreLauncher

```
// PreLauncher 事前に Go ルーチンを動かすワーカーのラウンチャー
type PreLauncher struct {
  queue chan<- Request
```

```
	wait  sync.WaitGroup
}

// NewPreLauncher ワーカーを起動するランチャーを起動する
func NewPreLauncher(ctx context.Context, nWorker int, newWorker func() *Worker)
	LauncherFacade {
	l := new(PreLauncher)
	queue := make(chan Request)
	l.queue = queue

	l.wait.Add(nWorker)

	for i := 0; i < nWorker; i++ {
		w := newWorker()
		go w.WatchQueue(queue, &l.wait)
	}
	go func() {
		<-ctx.Done()
		close(l.queue)
	}()

	return l
}

// Request 要求
func (l *PreLauncher) Request(req Request) {
	l.queue <- req
}

// Wait すべてのワーカが終了するまで待つ
func (l *PreLauncher) Wait() {
	l.wait.Wait()
}

// WatchQueue キューを見て仕事をする
func (w *Worker) WatchQueue(queue <-chan Request, wait *sync.WaitGroup) {
	for {
		req, ok := <-queue
		if !ok {
			break
		}
		w.work(req)
	}
	wait.Done()
}
```

この時対象とするデータのサイズを 1kB とし、Snappy 圧縮の仕事を行う場合と、なにもしない場合の両方を比較する。

PreLauncher においては、以下のパラメータを設定する。

- 事前に起動する Go ルーチンの数: 8
- チャネルのサイズ: 128

実験結果

実験結果は表 4.4 のようになった。

第 II 部

WebWorker 活用技術

  - Transferable、SharedArrayBuffer による転送はそれぞれ「Transferable インターフェース」、「SharedArrayBuffer」で述べた通りコピーコストがない。それに比べ方法 1 は Structured Clone アルゴリズムにより 2 度のデータコピーが存在し、圧倒的に不利なはず
- 1 転送コスト (初期化から転送の合計) の良い順では、方法 2 > 方法 1 > 方法 3 となる。
- 全転送コスト (分割転送をすべておこなう場合の計算) の良い順では、方法 3 > 方法 2 > 方法 1 となる。
  - 方法 3 については初期化時のコストに相当するコスト 3-1 が分割転送の中でも 1 度しかかからないため、

これを踏まえ、実際に計測を行った結果が以下。

1オペレーション当たりの時間

単位:ms

元文字列数	分割数	cost1_1	cost1_2and3	cost2_1	cost2_2	cost3_1	cost3_2and3
100,000	2	0.99	17.50	2.37	15.25	3.57	15.50
↑	8	1.69	17.40	1.00	17.40	5.39	14.40
1,000,000	2	8.34	17.60	13.73	12.60	18.88	14.00
↑	8	6.71	14.00	7.47	17.20	24.75	14.40
10,000,000	2	46.50	18.60	102.55	14.80	181.96	22.80
↑	8	49.77	15.40	29.87	15.00	203.73	17.40

▲図 7.11　Web Worker への転送コスト (Chrome)

また、1 転送コストを [cost1+cost1__2and3, cost2__1+cost2__2, cost3__1+cost3__2and3]、全転送コストを [cost1*分割数 +cost1__2and3*分割数, cost2__1*分割数 +cost2__2*分割数, cost3__1+cost3__2and3*分割数] と計算したものが以下。

コストの合計値

単位:ms

		1転送当たりのコスト			全転送コスト		
		方法1	方法2	方法3	方法1	方法2	方法3
100,000	2	18.49	17.62	19.07	36.97	35.24	34.57
↑	8	19.09	18.40	19.79	152.73	147.20	120.59
1,000,000	2	25.94	26.33	32.88	51.89	52.65	46.88
↑	8	20.71	24.67	39.15	165.66	197.37	139.95
10,000,000	2	65.10	117.35	204.76	130.20	234.71	227.56
↑	8	65.17	44.87	221.13	521.34	358.94	342.93

▲図 7.12　転送コストの合計計算値

結果を見てみると、いくつか予想と異なっていた。

- [予想] 初期化時のコストは方法 1 が最も有利 (コスト 1-1 > コスト 2-1 > コスト 3-1)