プログラミング言語Go
メモリモデル入門

柴田　芳樹　著

扉ページ（差し替え）

目次

付録 A

Go メモリモデル仕様

この付録は、Go 言語のメモリモデル仕様である「The Go Memory Model」（Version of May 31, 2014）を訳したものです。

A.1 はじめに

「Go メモリモデル」は、一つのゴルーチンによる変数の読み出しが、別のゴルーチンによって同じ変数へ書き込まれた値を観察する[*1] ことを保証する条件を定めています。

A.2 忠告

複数のゴルーチンにより同時にアクセスされようとしているデータを修正するプログラムは、そのデータへのアクセスを直列化（*serialize*）しなければなりません。

アクセスを直列化するには、データをチャネル操作あるいは`sync`パッケージや`sync/atomic`パッケージの同期基本操作でもって保護します。

あなたのプログラムの挙動を理解するために、このドキュメントの残りの部分を読まなければならないなら、あなたは賢すぎます。

賢くならないでください。

A.3 事前発生

単一のゴルーチン内での読み書きは、プログラムで書かれた順序で実行されたかのごとく動作しなければなりません[*2]。言い換えると、コンパイラとプロセッサ（CPU）は、単一のゴルーチン内で実行される読み書きの順序を変更して構いません。ただし、その順序の変更が、そのゴルーチン内での振る舞いを変更せずに、言語仕様で定義されている通りであればです。この順序の変更により、一つのゴルーチンで観測される実行順序は、別のゴルーチンから観察される実行

[*1] 訳注：「観察（*observer*）する」とは、変数を読み出したら、書き込まれた値が読み出されることを指します。

[*2] 訳注：「**逐次的に一貫**（*sequentially consistent*）」すると言います。

順序とは異なってもよいです。たとえば、一つのゴルーチンが `a = 1; b = 2;` を実行したとして、別のゴルーチンは `a` の更新された値より前に `b` の更新された値を観察するかもしれません[*3]。

読み書きの要件を定義するために、Go プログラムにおけるメモリ操作の実行に関する半順序（*partial order*）である**事前発生**（*happens before*）を定義します。イベント e_1 がイベント e_2 より前に発生したならば、e_2 は e_1 より後に発生したと言います。また、e_1 が e_2 より前に発生しておらず、かつ、e_2 より後にも発生していなければ、e_1 と e_2 は並行（*concurrently*）に発生していると言います。

規則：単一ゴルーチン内では、事前発生順序（*happens-before order*）とは、プログラムで表現されている順序です。

次の両方の条件が成り立てば、変数 `v` の読み出し r は、`v` への書き込み w を観察することが**可能です**。

- r は、w より前に発生していない。
- w より後だが r より前に `v` への他の書き込み w' が発生していない。

変数 `v` の読み出し r が、`v` への特定の書き込み w を観察することを保証するには、r により観察することが許される唯一の書き込みが w であることを保証することです。すなわち、次の二つの条件が成り立てば、r は `w` を観察することが**保証されます**。

- w が r より前に発生している。
- 共有された変数 `v` への他の書き込みが、w より前か、r より後に発生している

この二つの条件は、最初の二つの条件よりも制約が強いです。すなわち、w あるいは r と並行に他の書き込みが発生していないことを要求しています。

単一のゴルーチン内では、並行性はありません。したがって、二つの定義は同等です。読み出し r は、`v` への最も最近の書き込み w により書き込まれた値を観察します。**複数のゴルーチンが共有された変数 `v` へアクセスした場合、期待通りの書き込みが読み出しによって観察されることを保証するために、それらのゴルーチンは事前発生条件を確立するためにイベント[*4] を同期しなければなりません。**

[*3] 訳注：コンパイラは、命令の実行順序を `b = 2; a = 1;` と変更しても、それを実行しているゴルーチン内では、二つの文の実行後には、`a` に `1` が、`b` に `2` が入っているので振る舞いを変更していません。さらに、コンパイラが実行順序を変更しなくても、今日の *out-of-order* CPU では、書き込みはラインキャッシュにさえも書き込まれずに、*reorder buffer* と呼ばれるレジスタに両方の値が書き込まれてから、後で、実際のメモリへ書き込まれます。その場合、メモリへ先に書き込まれるのが `a` であることは保証されません。

[*4] 訳注：変数の読み出しや書き込みなどのプログラムの実行中のある種のイベントを指します。

変数 v をその型に対するゼロ値で初期化することは、メモリモデルにおける書き込みとして動作します。

単一のマシンワード（*machine word*）よりも大きな値の読み出しと書き込みは、順序が決まっていない複数のマシンワード単位の操作として動作します[*5]

A.4　同期

A.4.1　初期化

プログラムの初期化は、単一ゴルーチンで実行されますが、そのゴルーチンが並行に動作する他のゴルーチンを生成しても構いません。

規則：パッケージ *p* がパッケージ *q* をインポートしている場合、*q* のすべての `init` 関数の完了は、`p` のどの `init` 関数の開始より前に発生します。

規則：関数 `main.main` の開始は、すべての `init` 関数が完了した後に発生します。

A.4.2　ゴルーチンの生成

規則：新たなゴルーチンを開始する `go` 文は、その新たなゴルーチンの開始より前に発生します。

たとえば、次のプログラムを見てください。

```
var a string

func f() {
    print(a)
}

func hello() {
    a = "hello, world"
    go f()
}
```

[*5] 訳注：たとえば、32 ビット CPU では、`int64` や `float64` などの 64 ビット長のデータは、二回のメモリアクセスでメモリへの読み書きが行われます。

hello の呼び出しは、将来のある時点（おそらく hello が戻ってしまった後）に"hello, world"を表示します[*6]。

A.4.3　ゴルーチンの終了

ゴルーチンの終了は、プログラム内でのどのイベントよりも前に発生することは保証されていません。たとえば、次のプログラムを見てください。

```
var a string

func hello() {
    go func() { a = "hello" }()
    print(a)
}
```

a への代入の後に同期イベントはありません。したがって、他のゴルーチンがその代入の結果を観察できる保証はありません。実際、積極的なコンパイラは、go 文全体を削除するかもしれません。

あるゴルーチンの影響が他のゴルーチンから観察されなければならないなら、相対的な順序を確立するためにロック（*lock*）やチャネル（*channel*）通信といった同期機構を使わなければなりません。

A.4.4　チャネル通信

チャネル通信は、ゴルーチン間での同期を行う主な手段です。特定のチャネルへの個々の送信は、通常別のゴルーチンで行われる、そのチャネルからの対応する受信と一対になります。

規則：チャネルへの送信は、そのチャネルからの対応する受信が完了する前に発生します。

次のプログラムを見てください。

```
var c = make(chan int, 10)
var a string

func f() {
    a = "hello, world"
    c <- 0
}
```

[*6] 訳注：hello 関数内で変数 a に"hello, world"を代入していますが、その書き込みは、go f() で生成されたゴルーチンからは観察されることが保証されているということです。

```
func main() {
    go f()
    <-c
    print(a)
}
```

このプログラムは、"hello, world"を表示することが保証されています。a への書き込みは、c への送信より前に発生し、その送信は c からの対応する受信が完了する前に発生しており、受信の完了は print より前に発生しています。

規則：チャネルのクローズは、クローズによりゼロ値を返す受信より前に発生します。

前述のプログラム例で、c <- 0 を close(c) で置き換えても、同じ動作が保証されたプログラムとなります。

規則：バッファなしチャネルからの受信は、そのチャネルへの書き込みが完了する前に発生します。

次のプログラム（上記のプログラムと似ていますが、送信文と受信文が入れ替えられており、バッファなしチャネルを使っています）を見てください。

```
var c = make(chan int)
var a string

func f() {
    a = "hello, world"
    <-c
}

func main() {
    go f()
    c <- 0
    print(a)
}
```

このプログラムも"hello, world"を表示することが保証されています。a への書き込みは、c からの受信の前に発生し、c からの受信は c への対応する送信が完了する前に発生し、送信の完了は print より前に発生しています。

チャネルがバッファあり（たとえば、c = make(chan int, 1)）であれば、プログラムは"hello, world"を表示する保証はないでしょう（空の文字列を表示したり、クラッシュしたり、何か別のことを行うかもしれません）。

規則：容量 C を持つチャネルに対する k 番目の受信は、そのチャネルへの $k+C$ 番目の送信が完了する前に発生します。

この規則は、前述の規則をバッファありチャネルに対して一般化しています。この規則により、バッファありチャネルによって計数セマフォ（*counting semaphore*）をモデル化できます。すなわち、チャネル内の項目数はアクティブな利用数に相当し、チャネルの容量は同時に行える利用の最大数に相当し、項目の送信はセマフォを獲得し、項目の受信はセマフォを解放します。これは、並行性を制限するための一般的なイデオムです。

次のプログラムはワークリスト内の個々のエントリに対してゴルーチンを開始しますが、ゴルーチンは、同時に高々三つの処理が行われていることを保証するために、`limit` チャネルを使って協調しています。

```
var limit = make(chan int, 3)

func main() {
    for _, w := range work {
        go func(w func()) {
            limit <- 1
            w()
            <-limit
        }(w)
    }
    select{}
}
```

A.4.5　ロック

`sync` パッケージは二つのロックデータ型を実装しており、`sync.Mutex` と `sync.RWMutex` です。

規則：`sync.Mutex` あるいは `sync.RWMutex` の変数 `l`、および `n < m` の関係に対して、`n` 回目の `l.Unlock()` の呼び出しは、`m` 回目の `l.Lock()` の呼び出しが戻ってくる前に発生します。

次のプログラムを見てください。

```
var l sync.Mutex
var a string

func f() {
```

```
    a = "hello, world"
    l.Unlock()
}

func main() {
    l.Lock()
    go f()
    l.Lock()
    print(a)
}
```

このプログラムが、"hello, world"を表示することは保証されています。l.Unlock()に対する最初の呼び出し（f 内）は、l.Lock() に対する二番目の呼び出し（main 内）が戻る前に発生し、l.Lock() に対する二番目の呼び出しは print より前に発生します。

規則：sync.RWMutex の変数 l に対する l.RLock のすべての呼び出しに対して、l.RLock が l.Unlock の *n* 番目の呼び出しの後に発生し（戻り）、対応する l.RUnlock が *n*+1 番目の l.Lock の呼び出しの前に発生するような *n* が存在します。

A.4.6　Once

sync パッケージは、Once 型を使って、複数のゴルーチンが存在する中での初期化のために安全な機構を提供しています。特定の f に対して複数のスレッドが once.Do(f) を実行できますが[*7]、一つのゴルーチンだけが f() を実行し、他の呼び出しは f() が戻ってくるまで待たされます。

規則：once.Do(f) からの f() の単一の呼び出しは、once.Do(f) のすべての呼び出しが戻る前に発生します（戻ります）。

次のプログラムを見てください。

```
var a string
var once sync.Once

func setup() {
    a = "hello, world"
}
```

[*7] 訳注：Go のランタイムでは、複数のゴルーチンが一つの OS スレッド（*thread*）上で実行されます。ここでは、別々の OS スレッド上で実行されている二つ以上のゴルーチンが once.Do(f) を実行するという意味です。

```
func doprint() {
    once.Do(setup)
    print(a)
}

func twoprint() {
    go doprint()
    go doprint()
}
```

twoprint の呼び出しは、setup を正確に一度だけ呼び出します。setup 関数は、print のどちらの呼び出しよりも前に完了します。その結果として、"hello, world"は二回表示されます。

A.5　誤った同期

読み出し *r* は、それと並行に発生した書き込み w により書き込まれた値を観察するかもしれません。たとえ観察したとしても、*r* の後に発生した読み出しが *w* の前に発生した書き込みを観察することを意味していません。

次のプログラムを見てください。

```
var a, b int

func f() {
    a = 1
    b = 2
}

func g() {
    print(b)
    print(a)
}

func main() {
    go f()
    g()
}
```

g が 2 を表示してから 0 を表示することは起こり得ます[*8]。

[*8] 訳注：f() を実行するゴルーチンと g() を実行するゴルーチンが、別々の OS スレッド上で実行された場合、コンパイラによる実行命令の入れ替え、あるいは *out-of-order* CPU の *reorder buffer* の影響で g() 関数内で a をメモリから読み出したときに、f() 関数内での a = 1 の書き込みはメモリに反映されていない可能性があります。

この事実は、いくつかの一般的なイデオムを無効にします。

二重チェックロック（*double-checked locking*）は、同期のオーバーヘッドを避けようとするものです。たとえば、twoprint プログラムは、次のように誤って記述されるかもしれません。

```
var a string
var done bool

func setup() {
    a = "hello, world"
    done = true
}

func doprint() {
    if !done {
        once.Do(setup)
    }
    print(a)
}

func twoprint() {
    go doprint()
    go doprint()
}
```

しかし、doprint で、done への書き込みを観察することが、a への書き込みを観察することは保証されていません。このプログラムは、"hello, world"の代わりに（誤って）空文字列を表示するかもしれません[*9]。

別の誤ったイデオムは、次のようにある値に対してビジーウェイト（*busy wait*）することです。

```
var a string
var done bool

func setup() {
    a = "hello, world"
    done = true
```

[*9] 訳注：doprint() 関数を実行する二つのゴルーチンが別々の OS スレッド上で実行された場合、doprint() 内で done のゼロ値（false）を二つのゴルーチンが観察すると、どちらも once.Do(setup) を呼び出すので、期待通りの出力となります。

一つのゴルーチンが先に once.Do(setup) を実行して done = true が実行された後に、もう一つのゴルーチンが遅れて done を観察した場合、ゼロ値である false を観察する場合と true を観察する場合の二通りが起こり得ます。true を観察すると、one.Do(setup) の呼び出しは行わずに print(a) を実行します。その場合、コンパイラによる実行命令の入れ替え、あるいは *out-of-order* CPU の *reorder buffer* の影響で a へ代入される値は、メモリに反映されていない可能性があります。その結果、空文字列を表示することになります。

```
}

func main() {
    go setup()
    for !done {
    }
    print(a)
}
```

前の例と同様に、main 内で、done への書き込みを観察することは、a への書き込みを観察することを意味しません。したがって、このプログラムも空文字列を表示するかもしれません。もっと悪いことに、done への書き込みが main から観察されることは保証されません。なぜなら、二つのスレッド間でイベントの同期がないからです[*10]。main のループは終了することも保証されていません。

この問題には、次のプログラムのように、さらに微妙な変形があります。

```
type T struct {
    msg string
}

var g *T

func setup() {
    t := new(T)
    t.msg = "hello, world"
    g = t
}

func main() {
    go setup()
    for g == nil {
    }
    print(g.msg)
}
```

main が g != nil を観察してそのループを抜けたとしても、g.msg に対する初期化された値を観察することは保証されません。

ここでのすべての例では、解決方法は同じです。すなわち、明示的な同期を使うことです。

[*10] 訳注：このプログラムでは、main() 関数を実行しているゴルーチンと setup() 関数を実行しているゴルーチンの二つが存在しますが、それぞれが別々の OS スレッド上で実行される場合です。一つの OS スレッド上で二つのゴルーチンが実行されるのか、別々の OS スレッドが使われるかは、スケジューラ次第であり、プログラマが指定できません。