SPARTAN VIM 3.0

MURAOKA Taro
(KoRoN, @kaoriya)

スパルタン Vim 3.0

はじめに

今回のスパルタン Vim は 実際にプログラムの一部を書きながら Vim をどのように操作しているのか解説し、その背景にある Vim の哲学を読み取ってもらおうという趣旨です。これから書くプログラムのお題は「艦これ」です。擬似的に「艦これ」サーバ内の処理を Go 言語(以下 golang)で書きます。Vim と艦これと golang が一度に楽しめる、(筆者的に)お得なミニ冊子になっています。

なお本文中で登場するコード片は本物の艦これとは一切関係ありませんのであしからず。

スパルタン的コード編集

書くのは艦これの中でも特に楽しい戦闘処理です。戦闘処理に必要なインターフェース、実際のロジック(関数)を記述し、インターフェースのスケルトンを書いてみます。最終的にそれらを使って昼間の戦闘っぽいものを書いてみましょう。

インターフェース定義

艦これの戦闘処理と行っても膨大ですから、砲撃処理だけを考えてみます。砲撃には艦船(=BattleShip)が 2 隻必要ですね。ではその BattleShip インターフェースを決めてしまいましょう。

```
type BattleShip interface {
    HP() float32
    SetHP(v float32)
    Fire() float32
    Shield() float32
}
```

このくらいのものなら 40 秒(で支度しな!)以内でタイプできなければ失格です。そのためには 2 つめ以降の `float32` の入力に補完を使いましょう。

```
SetHP(v f
```

までタイプしたところで `<C-P>` として、すぐ上の `float32` から補完ができます。また `type` や `interface` といった golang 由来のキーワードは、別バッファで適当な golang のソースコードを開いておけば、それぞれ `ty<C-P>`, `inte<C-P>` で手軽に入力できるはずです。

ここで `t<C-P>` や `i<C-P>` としないのは、`true` や `int` などの別のキーワードを拾ってしまう可能性を避けるためです。このように補完メニューからいちいち選ぶのではなく、先行文字を多めに入力することで予め候補を絞り込んでしまい `<C-P>` 一発で目的の単語を選ぶのはスパルタンとしては基本技能です。

さらに付け加えればコンテキストに応じて先行文字を調整すべきです。たとえば golang において type は続けて使われることが多いでしょう。

```
type BattleShip interface {
    HP() float32
    SetHP(v float32)
    Fire() float32
    Shield() float32
}

type KanMusume struct {
    // TODO: implement BattleShip interface
}
```

このような時には 2 つ目となる `type KanMusume` の `type` は `t<C-P>` で入力します。すなわち、カーソルのある位置からみてどのような単語がどのような順番で存在しているか、それをどれだけ把握しているかがスパルタンにおける入力効率の差として顕著に現れます。把握すべき対象は `<C-P>` で辿れる上方向に限らず `<C-N>` で辿れる下方向にも、さらに言えば別バッファにまで及びます。

1. 次に入力する単語はこのファイルにないな
2. あ～既に開いた別のバッファにあったはず
3. あのバッファ、上と下どっちから探したほうが良いかな?
4. 下から探すと別のキーワードが邪魔だな
5. よし上から探そう `<C-N>`

スパルタンはこのようなことを、その単語を入力し始めるずっと前に考え **終わって** います。

砲撃処理

さて、では戦闘処理の本体を実装していきましょう。まずは昼間の砲撃によるダメージ処理を書いてみます。

```
func Bombard(attacker, defender BattleShip) float32 {
    // Calculate damege
    damege := attacker.Fire() * rand.Float32()
    damege -= defender.Shield() * rand.Float32()
    if damege < 0.0 {
        damege = 0.0
    }
    // Update defender's HP
    remain := defender.HP() - damege
    if remain < 0.0 {
        remain = 0.0
    }
    defender.SetHP(remain)
    return damege
}
```

上記には命中率やクリティカル判定もなく、悪名高い「アルテリオス方式」に少しランダム要素を加えてます。こんなものでも実際にプレイした感覚には結構近いでしょう。

おっと! ダメージ値を表す変数名は本来 `damage` でなければなりませんが、`damege` と typo してますね。直しましょう。

```
/dam<CR>
*
```

```
:%s//damage/g
```

最初に `/dam<CR>` で検索しているのは `*` で間違った変数名 `damege` を検索&捕捉するためです。`*` で検索するとカーソル下の単語に単語境界を示すメタ文字が自動的に追加され検索されます。つまり `damege` は `\<damege\>` というパターンで検索レジスタに記録(=捕捉)できます。捕捉したパターンは `:%s//damage/g` でパターン省略した際に利用します。

カーソルの位置によっては入力文字を省略して `/da<CR>` で検索してもよいでしょう。また `?dam<CR>` のように方向を意識するのも大事です。特に `'incsearch'` をオンにしておくと目的の単語を拾えるか最短で確認できて便利なのですが、その反面で画面が大きく書き換わって自分が何をしていたのか不明瞭になってしまうことがあります。それを避けるためにも検索方向は常に意識したほうがよいでしょう。

このくらい短い単語であればいきなり `:%s/damege/damage/g` してしまうという手も考えられます。ただしこれは `damege` を含む正しい単語(造語?)を書き換えてしまう可能性があります。それを避けるために単語境界メタ文字を加えるわけですが `\<damege\>` はタイプするにはやや複雑過ぎます。もちろん記号もばっちりタッチタイプで入力できるスパルタンにとっては 大した問題ではありませんが、さらなる typo を防ぐという観点や修正結果を目視で確認しなくて済むという観点からも `*` で捕捉しそれを置換に活用するのが最善手と言えるでしょう。

いずれにせよ「`damege` の `e` だけを `a` に書き換える」のではなく「`damege` を `damage` に書き換える」という考え方が重要です。日本人は日本語入力のコストの高さのせいか、間違えている文字だけを修正したくなりがちです。しかし Vim を使う上では間違った単語を正しい単語に書き換えるほうが、何かと都合が良いのです。

スケルトンの生成

次に KanMusume に BattleShip インターフェースの要件を満たさせるべく、各メソッド関数のスケルトンを書いてしまいましょう。

```
func (*KanMusume) HP() float32 {
    // TODO: implement me.
    return 0.0
}
```

1 つ目のメソッド関数は普通にタイプしてしまいます。ほぼ全部の単語に補完が聞くので 10 秒もかからないでしょう。関数の前に空行があるのに気をつけてください。

次にこの関数をヤンクします。ヤンク方法は関数の先頭行までカーソルを持って行って、Vjjjy でも 4yy でも良いでしょう。タイプ数以外には大した違いはありません。どちらのほうがミスが少ないとか、修正後の確認作業が要らないとか、そういうことが無い限り本質的な違いはありませんから、どちらでやっても良いのです。

ヤンクできたら同じものを 3 つペースト 3P します。ヤンク終了時点で関数の先頭行にカーソルがあります。その状態で 3P するとカーソルの上に 3 つ同じものが印刷され次のようになるでしょう。

```
func (*KanMusume) HP() float32 {
    // TODO: implement me.
    return 0.0
}
func (*KanMusume) HP() float32 {
```

```
    // TODO: implement me.
    return 0.0
}
func (*KanMusume) HP() float32 {
    // TODO: implement me.
    return 0.0
}
func (*KanMusume) HP() float32 { // この行にカーソルがある
    // TODO: implement me.
    return 0.0
}
```

この状態から以下のように操作します。

1. `[` で空白行に移動
2. `/H<CR>` で最初の `HP()` 関数の名前に移動
3. `*` で `\<HP\>` を捕捉しつつ 2 つ目の `HP()` へ移動
4. `cwSe<C-P><ESC>` で `HP()` を `SetHP()` に書き換え
5. `n` で次の `HP()` へ移動して `cwF<C-P><ESC>` で `Fire()` へ書き換え
6. `n` で次の `HP()` へ移動して `cwSh<C-P><ESC>` で `Shield()` へ書き換え (`S<C-P>` だと `SetHP` になってしまうので良くない)

するとこうなります。

```
func (*KanMusume) HP() float32 {
    // TODO: implement me.
    return 0.0
}
func (*KanMusume) SetHP() float32 {
    // TODO: implement me.
    return 0.0
}
func (*KanMusume) Fire() float32 {
    // TODO: implement me.
```

```
    return 0.0
}
func (*KanMusume) Shield() float32 { // またこの行にカーソルがある
    // TODO: implement me.
    return 0.0
}
```

あとは `SetHP` が少し間違っているので、これを以下のように直します。

1. `?Se<CR>` で `SetHP` の行へ移動
2. `%` で `SetHP()` の `)` へ移動
3. `if<C-P><ESC>` で `SetHP(float32) float32 {` に修正 (カーソルは 1 つめの `float32` の `2` の位置)
4. `Wdw` で 2 つめの `float32` を削除
5. `jjdd` で `return` 文を削除

以上の工程を経ると `SetHP()` はこう変わっています。

```
func (*KanMusume) SetHP(float32) {
    // TODO: implement me.
}
```

これで `KanMusume` のスケルトンはできました。細かいスタイリングは `:Fmt` に任せるか、実装時にやれば良いので今は無視です。

同様に敵側となる ShinkaiSeikan もスケルトンを作ってしまうと良いですね。手順の詳細は省略しますが `KanMusume` を `*` で捕捉した上で、KanMusume の定義全体をビジュアルモードでコピペして

```
:'<,'>s//ShinkaiSeikan/g
```

すると良いんじゃないでしょうか。`'<,'>` にはコピー元となった範囲がペースト後も正しく保持されていますから 上手く `ShinkaiSeikan` に置き換わり定

義ができるという寸法です。

昼間の戦闘処理

仕上げとして、ここまでに定義したメソッドや関数群を使って昼間の戦闘処理を書いてみましょう。

```
func CombatDaytime(friends []*KanMusume, enemies []*ShinkaiSeikan) {
    for _, ship := range friends {
        Bombard(ship, enemies[rand.Intn(len(enemies))])
    }
    for _, ship := range enemies {
        Bombard(ship, friends[rand.Intn(len(friends))])
    }
}
```

ゲームとしては行動順序とか轟沈判定とかもう何もかもメチャクチャですが、味方側敵側ともに 1 回ずつ砲戦を行うという意味においては間違っていません。

このようなものを書くときの手順はこうなります。まずは 1 つ目の for ループだけで関数を書いてしまいます。その際、キーワード補完の活用は忘れないように。

```
func CombatDaytime(friends []*KanMusume, enemies []*ShinkaiSeikan) {
    for _, ship := range friends {
        Bombard(ship, enemies[rand.Intn(len(enemies))])
    }
} // カーソルはここにある
```

ここから k でカーソルを 1 つ上に v% で for ループ全体を選択してしまいます。次に yP するとこうなります。

```
func CombatDaytime(friends []*KanMusume, enemies []*ShinkaiSeikan) {
    for _, ship := range friends { // カーソルはここ
        Bombard(ship, enemies[rand.Intn(len(enemies))])
    }
    for _, ship := range friends {
        Bombard(ship, enemies[rand.Intn(len(enemies))])
    }
}
```

ここからの操作手順は以下のようになります。

1. `/fr<CR>*` で `friends` を捕捉し同時に 2 つ目のループへ移動
2. `cwen<C-P><ESC>` で `friends` を `enemies` に書き換え
3. `*n` で `enemies` を捕捉しつつ次の `enemies` へ移動
4. `cwfr<C-P>` で `enemies` を `friends` に書き換え
5. `n.` でもう 1 つの `enemies` を `friends` に書き換える

ループのなかが複雑であればもっと違ったコマンドになりますが、このくらいシンプルならば以上のような操作で、簡単な間違いも避けられ十分だといえるでしょう。

まとめ

以上、手短ではありましたが Vim でコードを書く際の具体的な操作を解説しました。ポイントとなる操作は * による捕捉と <C-P> によるキーワード補完、それに各種の繰り返し操作と言って良いとは思いますが、より本質的にはヒューマンエラーをいかに排除するかという試みです。

本文で紹介した各操作にはよりタイプ数の少ない別バリエーションがいくらでもあります。しかしそれらは重要な箇所を人間が入力したり、すべての修正箇所を人間が目視で確認しないといけないものだったりすることが多いのです。

それに対して本文が採用した各操作には多少無駄な部分もあるものの、修正結果が 100%正しいことを保証するように考えられたものでその結果を目視で確認する必要はなく、タイプしている時間で次の編集内容とその方法を考察するという スパルタン Vim 1.0 で紹介した脳の使い方を体現しています。

その観点から言えば本書にあげた操作よりも、もっと良い操作を考えられるかもしれません。ぜひ皆さんももう一度読み直し考えてみてください。

あとがき

衝撃の前作スパルタン Vim 2.0 - Vim ニンジャスレイヤーから 1 年半、別の意味で衝撃の初代スパルタン Vim (1.0)から丸 2 年、真面目な方のスパルタン Vim が帰ってきました。とは言え半日で書き上げたミニ冊子ですが、例によって私が最近はまってるものを織り込んだ、スパルタン Vim らしいカオスな仕上がりになったかと考えます。少しでも楽しんでもらえればと願う次第です。

2013/12/31 村岡 太郎 (KoRoN, @kaoriya)