Go の隠されたオプションでビルド動作を可視化する

December 21, 2020

この記事は Go Advent Calendar 2020 の 21 日目の記事です。

20 日目は @soichisumi さんの [ ]byte フィールドをbase64変換しない json.Marshal/Unmarshalライブラリを作った でした。
21 日目は @KthS さんです。

はじめに

Go の本体のコードを眺めていたら -debug-actiongraph、 -debug-trace という面白そうなオプションを見つけました。 この記事ではこれらのオプションがどういうものなのかを解説します。

注意として、このオプションは公式でドキュメント化されていません。 安定しておらず互換性が壊れる可能性があり、この記事の情報も古くなっている可能性があります。 ^1

-debug-actiongraph は以前より存在したようですが、-debug-trace オプションはまだ正式にリリースはされておらず、beta 版として最近公開された Go 1.16beta1 に含まれています。 実際に試すには Go 1.16beta1 をインストールするか、最新の Go のソースコードを取得して自分でビルドする必要があります。 Go コンパイラをソースコードからビルドするには、公式のドキュメント もしくは Go コンパイラをソースコードからビルドする を参照してください。

以下では Go は go1.16beta1 を利用しています。

```
$ go version
go version go1.16beta1 darwin/amd64
```

-debug-trace オプションとは？

このIssue で提案されて追加されたオプションです。

-debug-trace オプションはビルドの動作を可視化することが目的のようです。
Go のビルドは Action (code)という単位で実行されているようです。

もともと、この Action の流れのグラフを出力するための -debug-actiongraph というデバッグオプションがあったのですが、 「Action グラフが構築する前の情報も見たいよね。」ということで、追加されたのが -debug-trace オプションのようです。

-debug-actiongraph で表示される情報

そもそも -debug-actiongraph ではどのような情報が取得されていたのでしょうか。

実際にアプリケーションを作成して、-debug-actiongraph を利用してビルドしてみます。

グラフを出力する

今回は次のような「なにもしない」をするアプリケーションを用意しました。

```
package main

func main() {
}
```

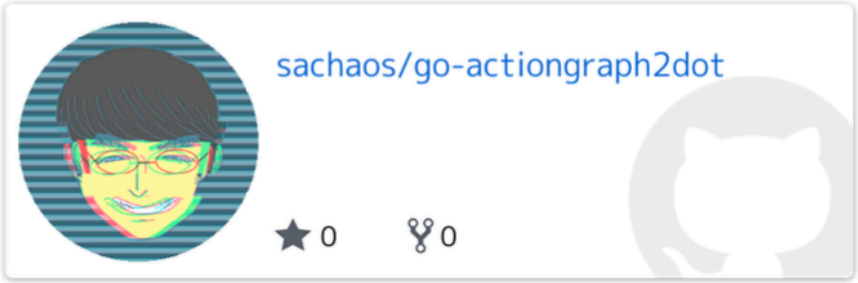
これをビルドしてみます。

```
$ go build -debug-actiongraph=graph.json main.go
```

実際に取得された `graph.json` は Gist にあげています。

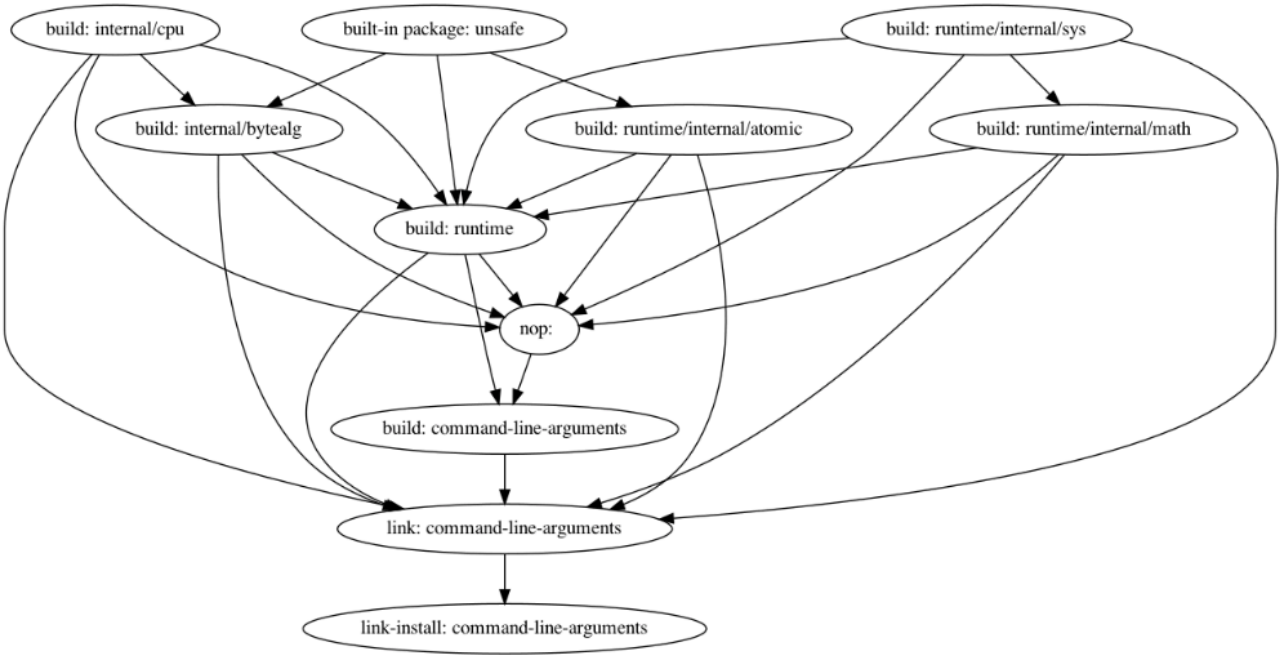
グラフを可視化する

ちょっとそのままではわかりにくいので可視化できるように `Dot` 言語に変換するツールを作ってみました。



これを使って `Dot` 言語に変換することで、以下のようなグラフを描画できます。

```
$ cat graph.json | go-actiongraph2dot | dot -Tpng > graph.png
```



これで build や link などの各アクションにどのような依存関係が存在するのか、というのが一目瞭然でわかるようになりました。

-debug-trace を使ってみる

トレースを出力する

先程の「なにもしない」をするアプリケーションを -debug-trace 付きでビルドします。

```
$ go build -debug-trace=trace.json main.go
```

出力は JSON が出てきます。

```
$ cat trace.json | head -n 5
[
{"name":"Running build command","ph":"B","ts":1608120675704571
,{"name":"load.PackagesAndErrors","ph":"B","ts":16081206757048
,{"name":"load.PackagesAndErrors","ph":"E","ts":16081206757873
,{"name":"exec.Builder.Do (link-install main)","ph":"B","ts":1
```

trace.json は こちらのGist に上げています。

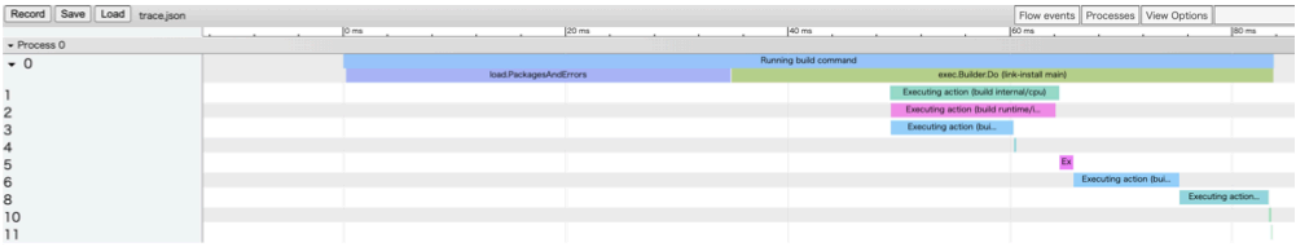
トレースを可視化する

トレースは Trace Event Format という形式で出力されているようです。 このJSONの出力結果はGoogle の Trace-viewer を使うことで、可視化することができます。

Chrome ではすでにこちらを搭載しているので chrome://tracing にアクセスすることで、 Trace-viewer を利用することができます。

出力された JSON を Trace-viewer にドラッグ&ドロップすることで読み込ませて、可視化することができます。

先ほどのトレース結果を可視化すると次のようになりました。



少し詳しくみていきます。

まずは `load package` によってソースコードが読み込まれます。

そして `exec.Builder` によって依存しているパッケージがビルドされます。 今回は何も `import` していないプログラムですが、 `runtime` が必要になるため、 以下のような `runtime` から依存されている `package` がビルドされているようです。

- `internal/cpu`
- `runtime/internal/atomic`
- `runtime/internal/sys`
- `runtime/internal/math`
- `internal/bytealg`

そして `command-line-arguments` (`build` コマンドの引数で渡された `package`) がビルドされ、リンクされます。

`Hello World` を 可視化してみる

次に `fmt package` に依存した `Hello World` するアプリケーションをビルドしてみます。

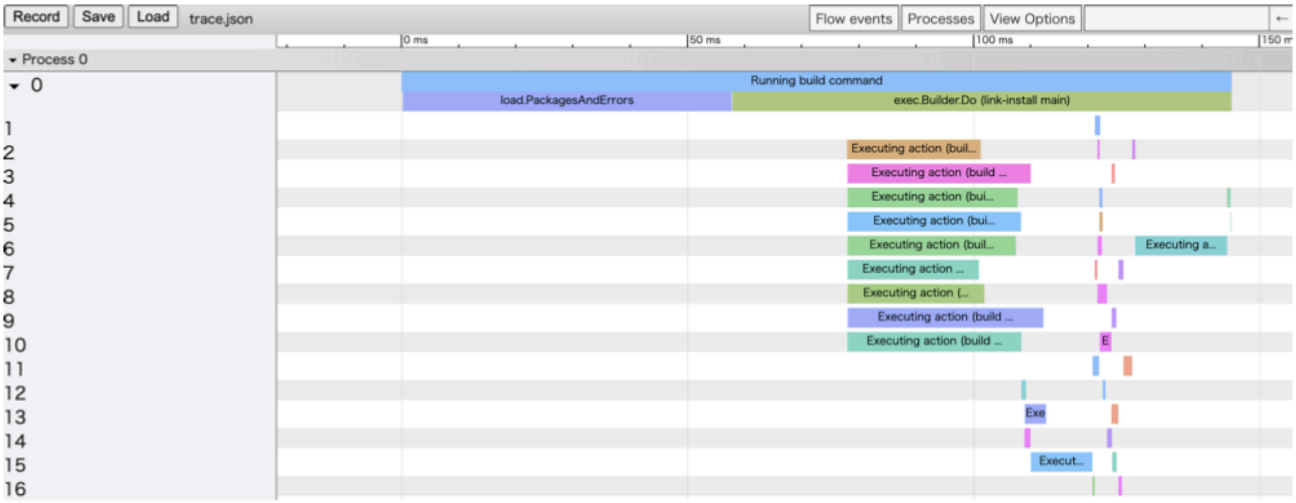
```
package main

import "fmt"

func main() {
```

```
        fmt.Println("Hello World")
}
```

これをビルドしてトレースを可視化すると以下のようになりました。



trace.json は こちらのGist に上げています。

先ほどの、無のアプリケーションに比べ多くのビルドプロセスが走っています。

runtime と並行して、unicode package という如何にも fmt package が依存していそうな package のビルドが走っています。 実際にこうして可視化してみると、ビルドが並列に行われてビルド時間が短縮されている様子がわかりやすいですね。

まとめ

この記事では、

- -debug-actiongraph を使って Action Graph を出力する方法を説明しました。
- また、出力された Action Graph を可視化する自作ツールを紹介しました。
- -debug-trace を使って Go のビルド中のトレースを得る方法を説明しました。
- また、得られたトレースを可視化する方法を紹介しました。

アプリケーションが巨大化してきて、ビルドに時間がかかるようになってきます。
そういう時にトレースを取得すると、何がボトルネックになっているのかわかるようになるかもしれませんね。

GitHub: @sachaos
Twitter: @sachaos