Fabric

2019 年 07 月 09 日

目次

このサイトでは Fabric の利用法と API のドキュメントをカバーしています。公開されている変更履歴とこのプロジェクトがどのようにメンテされているかを含んだ、Fabric とは何かとうことに関する基本的な情報は メインのプロジェクトウェブサイト をご覧ください。

第 1 章

チュートリアル

新しいユーザーや Fabric の基本的な機能の概要を知りたい方は **概要とチュートリアル** を参照してください。このドキュメントの残りの部分では、内容に関して少なくとも一時的にはよく知っていることと仮定します。

1.1 概要とチュートリアル

Fabric へようこそ！

このドキュメントは Fabric の機能を紹介する駆け足のツアーであり、使い方のクイックガイドでもあります。さらに詳しいドキュメント (全体にリンクされています) は **使用方法** にあります。ぜひご覧になってください。

1.1.1 Fabric とは？

`README` によると:

> Fabric は、アプリケーションのデプロイやシステム管理のタスクのために SSH の利用を簡素化するための Python(2.5-2.7) のライブラリとコマンドラインのツールです。

もっと具体的に言うと、Fabric とは:

- **コマンドライン** 経由で **任意の Python 関数** を実行するツールです。
- (低レベルライブラリの上に構築された) サブルーチンのライブラリで、SSH 経由で **簡単に** かつ **Python 風に** シェルコマンドを実行します。

当然、たいていのユーザーはこの 2 つを組み合わせます。Fabric を使って Python の関数もしくは **タスク** を作成し、実行し、リモートサーバとのやりとりを自動化します。ではちょっと見てみましょう。

1.1.2 Hello, `fab`

"いつもの"がないと正しいチュートリアルではないですよね:

```
def hello():
    print("Hello world!")
```

上のコードを `fabfile.py` という名前の Python モジュールファイルとしてカレントのワーキングディレクトリに置くと、 `fab` ツール (Fabric のパーツとしてインストールされています) で `hello` 関数を実行することができ、期待した通りに動きます:

```
$ fab hello
Hello world!

Done.
```

どうってことはありませんね。この機能性により、自身の API を何もインポートしなくても (とても) ベーシックなビルドツールとして Fabric を利用することができます。

注釈: `fab` ツールは単にあなたの fabfile をインポートしてその指示にしたがい、ひとつもしくは複数の関数を実行します。何かマジックがあるわけではありません。通常の Python スクリプトで可能なすべてのことが fabfile 内でも可能なのです！

参考:

実行ストラテジー, **タスクの定義**, *fab* **オプションと引数**

1.1.3 タスク引数

Fabric では実行時引数をタスクに渡せるので便利なことも多いです。ちょうと通常の Python プログラミングのようなものです。Fabric はこの基本的なサポートを持っていて、シェル互換ノーテーションを使っています: `<task name>:<arg>,<kwarg>=<value>,...` 不自然な感じがするかもしれませんが、上の例を拡張してあなたに say hello と言うようにしてみましょう:

```
def hello(name="world"):
    print("Hello %s!" % name)
```

デフォルトでは、 `fab hello` を呼び出しても以前と同じ動きをします。今度はこれをパーソナライズしてみましょう:

```
$ fab hello:name=Jeff
Hello Jeff!
```

(次のページに続く)

(前のページからの続き)

```
Done.
```

Python プログラミングに慣れた方なら、この呼び出しでもまったく同じ挙動をすることが想像できると思います:

```
$ fab hello:Jeff
Hello Jeff!

Done.
```

差し当たりは、引数の値は常に文字列として Python に現れ、リストなどの複雑な型では少し文字列操作が必要になります。将来のバージョンではこれをより簡単にするため、型キャストシステムが追加されるかもしれません。

参考:

Per-task **引数**

1.1.4 ローカルコマンド

上の例では、`fab` は `if __name__ == "__main__"` の定型文の何行かを省略できるに過ぎません。たいていは Fabric の API と利用するためにデザインされます。API にはシェルコマンドの実行、ファイルの転送などの関数 (もしくは **操作**) が含まれます。

では、仮定のウェブアプリケーションの fabfile を作ってみましょう。この例のシナリオは次のようなものです: このウェブアプリケーションはリモートホスト `vcshost` 上に Git 経由で管理されています。 `localhost` 上ではこのウェブアプリケーションのローカルクローンがあります。 `vcshost` に変更をプッシュすると、すぐに、そして自動的にリモートホスト `my_server` に変更を反映させたいと思います。これを、ローカルとリモートの Git コマンドを自動化することによって実施させてみましょう。

通常は、fabfile はプロジェクトのルートに置くといいでしょう:

```
.
|-- __init__.py
|-- app.wsgi
|-- fabfile.py <-- our fabfile!
|-- manage.py
`-- my_app
    |-- __init__.py
    |-- models.py
    |-- templates
    |   `-- index.html
    |-- tests.py
    |-- urls.py
    `-- views.py
```

注釈: ここでは Django アプリケーションを使用していますが、単に例として用いているだけです。Fabric は、SSH ライブラリは別として、どんな外部のコードベースにもひも付けられていません。

まず第一にこのテストを実行し、VCS にコミットしてみましょう。そしてデプロイを準備をします:

```
from fabric.api import local

def prepare_deploy():
    local("./manage.py test my_app")
    local("git add -p && git commit")
    local("git push")
```

出力はだいたい次のようになるでしょう:

```
$ fab prepare_deploy
[localhost] run: ./manage.py test my_app
Creating test database...
Creating tables
Creating indexes
.......................................
----------------------------------------------------------------------
Ran 42 tests in 9.138s

OK
Destroying test database...

[localhost] run: git add -p && git commit

<interactive Git add / git commit edit message session>

[localhost] run: git push

<git push session, possibly merging conflicts interactively>

Done.
```

このコード自身は単純です。Fabric の API 関数 *local* をインポートし、それを利用してローカルのシェルコマンドを実行し、やりとりを行います。他の Fabric の API も似ていて、すべてただの Python です。

参考:

オペレーション, *fabfile* **の探索**

1.1.5 好きなように構造化する

Fabric は"ただの Python"なので、fabfile は好きなように自由に構造化できます。例えば、サブタスクに分けることから始めると便利でしょう:

```
from fabric.api import local

def test():
    local("./manage.py test my_app")

def commit():
    local("git add -p && git commit")

def push():
    local("git push")

def prepare_deploy():
    test()
    commit()
    push()
```

prepare_deploy タスクは以前と同じように呼び出すことができますが、今回は必要であればサブタスクの一つとしてより粒度を細かくして呼び出しをすることができます。

1.1.6 失敗

基本的な動きは問題ないですが、もしテストに失敗したらどうなるでしょうか? デプロイの前にブレーキをかけて修正する機会があります。

Fabric は操作経由で呼び出されたプログラムの返り値をチェックして、正常に終了しなかった場合には停止します。テストのひとつがエラーに出くわしたときにどうなるか見てみましょう:

```
$ fab prepare_deploy
[localhost] run: ./manage.py test my_app
Creating test database...
Creating tables
Creating indexes
.............E.........................
======================================================================
ERROR: testSomething (my_project.my_app.tests.MainTests)
----------------------------------------------------------------------
Traceback (most recent call last):
[...]

----------------------------------------------------------------------
Ran 42 tests in 9.138s
```

(次のページに続く)

(前のページからの続き)

```
FAILED (errors=1)
Destroying test database...

Fatal error: local() encountered an error (return code 2) while executing './manage.py␣
↪test my_app'

Aborting.
```

素晴らしい! 私たち自身では何もする必要がありませんでした。Fabric が失敗を検知して停止し、`commit` タスクは決して実行されることはありません。

参考:

Failure handling (usage documentation)

失敗の扱い

さて、これを柔軟にしてユーザーに選択をさせるにはどうすれいいでしょう? *warn_only* と呼ばれる設定 (もしくは **environment variable**、通常は短く **env var**) が停止を警告に変え、柔軟なエラーの扱いを可能にします。

`test` 関数でこの設定を有効にして、*`local`* 呼び出しの結果を調べて見ましょう:

```
from __future__ import with_statement
from fabric.api import local, settings, abort
from fabric.contrib.console import confirm

def test():
    with settings(warn_only=True):
        result = local('./manage.py test my_app', capture=True)
    if result.failed and not confirm("Tests failed. Continue anyway?"):
        abort("Aborting at user request.")

[...]
```

この新しい機能を追加するにあたり、新しいことをたくさん導入しました:

- Python 2.5 では `with:` を使うために `__future__` のインポートが必要です。
- Fabric の `contrib.console` サブモジュールは `confirm` 関数を含んでいて、簡単なイエス/ノープロンプトに使われます。
- *`settings`* コンテキストマネージャーはコードの特定のブロックに設定を適用するのに使われます。
- *`local`* のようなコマンドランニング操作は、その結果 (`.failed` や `.return_code` など) に関する情報を含むオブジェクトを返すことができます。

- そして *abort* 関数は手動で停止を実行するために使われます。

とは言え、この追加的な複雑性を別にすれば、理解するのは依然としてとても簡単で、さらに柔軟になりました。

参考:

コンテキストマネージャー, *env* **変数の全リスト**

1.1.7 接続する

では今度は、肝心な部分を入れて fabfile を仕上げましょう。 `deploy` タスクは一つもしくは複数のリモートサーバーで実行され、コードが確実に最新になるようにします:

```
def deploy():
    code_dir = '/srv/django/myproject'
    with cd(code_dir):
        run("git pull")
        run("touch app.wsgi")
```

今回もまた、たくさんの新しいコンセプトが導入されています:

- Fabric はただの Python です。したがって、変数や文字列の操作などの通常の Python コードの概念を自由に利用することができます。
- *cd* はコマンドに ‘‘cd /どこ/かの/ディレクトリ ‘‘ 呼び出しを追加する簡単な方法です。これは同じことをローカルで実行する *lcd* と似ています。
- *run* は *local* に似ていますが、ローカルではなく **リモートで** 動作します。

また、ファイルの一番上で新しい関数を確実にインポートするようにします:

```
from __future__ import with_statement
from fabric.api import local, settings, abort, run, cd
from fabric.contrib.console import confirm
```

これらを変更したら、デプロイしてみましょう:

```
$ fab deploy
No hosts found. Please specify (single) host string for connection: my_server
[my_server] run: git pull
[my_server] out: Already up-to-date.
[my_server] out:
[my_server] run: touch app.wsgi

Done.
```

この fabfile では接続情報は指定していません。したがって、Fabric はどのホスト (複数可) でこのリモートコマン

ドが実行されるべきなのかが分かりません。このようなとき、Fabric は起動時に入力を促します。接続定義は SSH のような "ホスト文字列" (例えば `user@host:port`) を使い、デフォルトではローカルのユーザー名が使われます。そのため、この例では単にホスト名 `my_server` だけを指定しています。

リモートとの双方向性

チェックアウトしたソースコードがすでにあるのなら `git pull` で問題ないでしょう。しかし最初のデプロイだったらどうでしょう? そうしたケースも扱えて、最初の `git clone` も実行するようにするといいでしょう:

```
def deploy():
    code_dir = '/srv/django/myproject'
    with settings(warn_only=True):
        if run("test -d %s" % code_dir).failed:
            run("git clone user@vcshost:/path/to/repo/.git %s" % code_dir)
    with cd(code_dir):
        run("git pull")
        run("touch app.wsgi")
```

上の *`local`* との場合と同じように *`run`* もまた、シェルコマンドの実行をベースにきれいな Python レベルのロジックを組み立てることができます。しかし、ここでの興味深い部分は `git clone` 呼び出しで、Git サーバ上のリポジトリへのアクセスに Git の SSH メソッドを利用します。つまりリモートの *`run`* 呼び出しは、自身の認証を必要とするのです。

Fabric の以前のバージョン (と、同じようなハイレベルな SSH ライブラリ) では、リモートプログラムのの実行は中途半端な状態で、ローカル側からは触れませんでした。これはパスワードの入力が本当に必要な場合やリモートプログラムとの情報のやりとりが必要な場合に解決が難しい問題でした。

Fabric 1.0 以降ではこの問題を解決し、リモート側と常にやりとりできることを確保しています。では、Git チェックアウトがないときに新しいサーバー上でアップデートした `deploy` タスクを実行したときに何が起こるか見てみましょう:

```
$ fab deploy
No hosts found. Please specify (single) host string for connection: my_server
[my_server] run: test -d /srv/django/myproject

Warning: run() encountered an error (return code 1) while executing 'test -d /srv/
↪django/myproject'

[my_server] run: git clone user@vcshost:/path/to/repo/.git /srv/django/myproject
[my_server] out: Cloning into /srv/django/myproject...
[my_server] out: Password: <enter password>
[my_server] out: remote: Counting objects: 6698, done.
[my_server] out: remote: Compressing objects: 100% (2237/2237), done.
[my_server] out: remote: Total 6698 (delta 4633), reused 6414 (delta 4412)
[my_server] out: Receiving objects: 100% (6698/6698), 1.28 MiB, done.
[my_server] out: Resolving deltas: 100% (4633/4633), done.
```

(次のページに続く)

(前のページからの続き)

```
[my_server] out:
[my_server] run: git pull
[my_server] out: Already up-to-date.
[my_server] out:
[my_server] run: touch app.wsgi

Done.
```

`Password:` プロンプトは、ウェブサーバ上のリモートの `git` 呼び出しで、Git サーバーへのパスワードへの問い合わせであることに留意してください。パスワードをここで入力することができ、クローンは通常のように継続されます。

参考:

リモートプログラムとのやりとり

予め接続を定義する

起動時に接続情報を指定するのはすぐにうんざりしてくると思います。そのため Fabric では、fabfile 内やコマンドライン上でこれを行うためのたくさんの手段を提供しています。ここではすべてをカバーしませんが、もっともよくある手段、グローバルなホストリストの設定 *env.hosts* をお見せしましょう。

env は Fabric のたくさんの設定を操作するグローバルな辞書のようなオブジェクトで、さらに属性とともに書くことも可能です。(実際のところ、上にみられるように *settings* はこれの単なるラッパーです) したがって、モジュールレベルで、自分の fabfile の一番上に近いところで次のように変更が可能です:

```
from __future__ import with_statement
from fabric.api import *
from fabric.contrib.console import confirm

env.hosts = ['my_server']

def test():
    do_test_stuff()
```

`fab` が fabfile を読み込むとき、今回変更した `env` が実行され、設定の変更を格納します。その結果は上の通りになり、`deploy` タスクが `my_server` に対して実行されます。

また、このようにして、Fabric に対して一度に複数のリモートシステム上で実行させることもできます。`env.hosts` はリストなので `fab` はこのリストを順に処理し、各接続に対して与えられたタスクを呼び出します。

参考:

環境辞書、 *env*, ホストリストがどのように作られるか

1.1.8 まとめ

完成した fabfile は、それでもかなり短いものです。全体では以下になります:

```
from __future__ import with_statement
from fabric.api import *
from fabric.contrib.console import confirm

env.hosts = ['my_server']

def test():
    with settings(warn_only=True):
        result = local('./manage.py test my_app', capture=True)
    if result.failed and not confirm("Tests failed. Continue anyway?"):
        abort("Aborting at user request.")

def commit():
    local("git add -p && git commit")

def push():
    local("git push")

def prepare_deploy():
    test()
    commit()
    push()

def deploy():
    code_dir = '/srv/django/myproject'
    with settings(warn_only=True):
        if run("test -d %s" % code_dir).failed:
            run("git clone user@vcshost:/path/to/repo/.git %s" % code_dir)
    with cd(code_dir):
        run("git pull")
        run("touch app.wsgi")
```

この fabfile は Fabric の機能セットのうちのかなりの部分を利用しています:

- fabfile のタスクを定義し、それを *fab* で実行
- *local* でローカルのシェルコマンドを呼び出し
- *settings* で env 変数を変更
- コマンド失敗の扱い、ユーザーにプロンプト表示、手動アボート
- ホストリストの定義と *run* のリモートコマンド実行

とは言え、ここではカバーしていないこともまだたくさんあります！ぜひさまざまな "see also" リンクをたどってみてください。また、 *the main index page* のドキュメントの目次もチェックしてみてください。

読んでくれて、ありがとうございます！

第 2 章

使い方のドキュメント

次のリストは Fabric の API ではないドキュメントの主な項目をすべて含んでいます。 **概要とチュートリアル** のコンセプト概要から発展させて、上級者向けのトピックもカバーしています。

2.1 環境辞書、 env

Fabric のシンプルだけれども不可欠な側面は "(環境)environment" として知られているものです。これは Python の辞書サブクラスで、設定レジストリの組み合わせとして利用され、タスク間のデータ空間で共有されます。

この環境辞書はグローバルなシングルトン、 `fabric.state.env` として現在は実装されていて、便利なように `fabric.api` 内に含まれています。 `env` のキーは "env variables(環境変数)" として言及さることがあります。

2.1.1 設定としての環境

Fabric のほとんどの挙動は `env` 変数、例えば **チュートリアル** で見られる `env.hosts` などを変更することによってコントロールされます。よく変更されて利用される `env` 変数は以下のものがあります:

- `user`: SSH 接続を実行するときの Fabric のデフォルトの値はご自分のローカルユーザー名で、必要なら `env.user` を使ってオーバーライドできます。また、ドキュメントの **実行モデル** にはホストごとにユーザー名を指定するための情報が記載されています。
- `password`: 必要に応じてデフォルト接続や sudo のパスワードを設定するために使用します。Fabric は、この値が設定されていない場合や正しくない場合に必要になった時にプロンプトを表示します。
- `warn_only`: リモート側でエラーを検知したときに Fabric が終了するかどうかを決めるブール設定です。この挙動に関する詳細は **実行モデル** をご覧ください。

他にもたくさんの env 変数が有ります。完全な一覧はこのドキュメントの下、 *env* **変数の全リスト** をご覧ください。

settings のコンテキストマネージャー

多くのケースで、一時的に env 変数の変更ができて、与えられる設定変更があるコードブロックだけに適用される方が便利なことあります。Fabric は *settings* コンテキストマネージャを提供しています。これは任意の数のキー/バリューのペアを取ることができ、それでラップされたブロック内の env を変更するのに利用できます。

例えば、 warn_only (下を参照) 設定が便利な状況はたくさんあると思います。これを何行かのコードに適用するには contrib exists 関数にあるように、 settings(warn_only=True) を使います。:

```
from fabric.api import settings, run

def exists(path):
    with settings(warn_only=True):
        return run('test -e %s' % path)
```

settings とその他類似のツールに関する詳細は **コンテキストマネージャー** API ドキュメントをご覧ください。

2.1.2 共有された状態としての環境

前述のとおり、 env オブジェクトは単純な辞書サブクラスで、自分の fabfile コードにも保管することができます。これは単一の実行内に複数のタスク間で状態を保持するのに便利なことが有ります。

注釈: env のこの側面は多分に歴史的なものです。以前は、fabfile は純粋な Python ではありませんでしたので、この「環境」がタスク間でやりとりするための唯一の方法でした。現在では、他のタスクやサブルーチンをダイレクトに呼び出すことができ、必要ならモジュールレベルでの共有された状態を保持することさえできます。

将来のバージョンでは、Fabric はスレッドセーフになり、その時点で env はグローバルな状態を保持する唯一の簡単で安全な方法になるでしょう。

2.1.3 その他の留意事項

これは dict のサブクラスである一方、Fabric の env は変更されていて、これまでの例等のいくつかに見られるよう、属性アクセスの方法でその値を読み書きできます。言い換えると、 env.host_string と env['host_string'] は機能的に同一です。属性アクセスはタイピングを少しだけですが節約できることがよくありますし、コードをより読みやすくします。なので、 env とのやりとりにはこの方法をおすすめします。

これが辞書であるということは、Python の **辞書** ベースの文字列挿入などの他の場合に有益になりえます。これは特に、単一の文字列に複数の env 変数を挿入する必要があるときにとても便利です。"通常の" 文字列挿入の利用は次のようなものです:

```
print("Executing on %s as %s" % (env.host, env.user))
```

辞書スタイルの挿入を利用するとより読みやすく、少しだけ短くなります:

```
print("Executing on %(host)s as %(user)s" % env)
```

2.1.4 env 変数の全リスト

以下は定義済みの (もしくは実行中に Fabric 自身によって定義される) すべての環境変数のリストです。この内の多くは直接操作できますが、通常は *settings* 経由もしくは *cd* などの特定のコンテキストマネージャ経由で *context_managers* を使うほうが多くの場合はベストでしょう。

これらの多くが `fab` のコマンドラインスイッチ経由で設定可能であることに留意してください。このコマンドラインスイッチの詳細は *fab* **オプションと引数** をご覧ください。クロスリファレンスが適宜提供されています。

参考:

--set

abort_exception

デフォルト: `None`

Fabric は通常は、標準エラー出力にエラーメッセージを表示し ‘‘sys.exit(1)‘‘ を呼び出すことによって中止を扱います。この設定により、この挙動 (`env.abort_exception` が `None` の時に起こること) をオーバーライドできます。

文字列 (表示されるであろうエラーメッセージ) を取り、例外インスタンスを返す呼び出し可能なものを与えます。`SystemExit` (`sys.exit` が行うこと) の代わりにこの例外オブジェクトが引き起こされます。

大抵の場合、上記説明 (呼び出し可能で、文字列を取り、例外インスタンスを返す) に完全に合わせるように、単にこれを例外クラスにセットするとよいでしょう。例えば、 `env.abort_exception = MyExceptionClass` です。

abort_on_prompts

デフォルト: `False`

`True` の時、Fabric は非インタラクティブモードで動作し、いつでも *abort* を呼び出すとユーザーに入力を促すプロンプトを表示します (パスワードの入力プロンプト、"どのホストに接続しますか?" プロンプト、 *prompt* の fabfile 実行などなど)。これにより、予期せぬ状況が発生した時にユーザーの入力を永遠にブロックする代わりに、ユーザーは Fabric のセッションを常にきれいに終了させることができます。

バージョン 1.1 で追加.

参考:

--abort-on-prompts

all_hosts

デフォルト: `[]`

実行しているコマンドのために全ホストのリストが `fab` によって設定されます。情報目的のみです。

参考:

実行モデル

always_use_pty

デフォルト: `True`

`False` にセットされると *run*/*sudo* が `pty=False` 付きで呼び出された時のように振る舞います。

参考:

--no-pty

バージョン 1.0 で追加.

colorize_errors

デフォルト `False`

`True` にセットされると、見分けやすいようにターミナルへのエラー出力を赤で、警告をマジェンタで表示します。

バージョン 1.7 で追加.

combine_stderr

デフォルト: `True`

SSH 層でリモートプログラムの標準出力と標準エラー出力のストリームをマージして、表示するときに見難くならないようにします。これが必要な理由とその効果についての詳細はを **標準出力と標準エラー出力の結合** ご覧ください。

バージョン 1.0 で追加.

command

デフォルト: `None`

fab によって実行中のコマンド名がセットされます (例えば、 `$ fab task1 task2` として実行された場合、`env.command` は `task1` の 実行中は `"task1"` にセットされ、次に `"task2"` にセットされます)。情報目的のみです。

参考:

実行モデル

command_prefixes

デフォルト: `[]`

prefix によって変更され、 *run*/*sudo* によって実行されるコマンドの先頭に追加されます。

バージョン 1.0 で追加.

command_timeout

デフォルト: `None`

リモートコマンドのタイムアウト、秒で指定。

バージョン 1.6 で追加.

参考:

--command-timeout

connection_attempts

デフォルト: `1`

新しいサーバーに接続するときに Fabric が接続を試行する回数です。後方互換性のため、デフォルトでは 1 回だけの試行になっています。

バージョン 1.4 で追加.

参考:

--connection-attempts, *timeout*

cwd

デフォルト: `''`

カレントのワーキングディレクトリ。*cd* コンテキストマネージャ用に状態保持のために利用されます。

dedupe_hosts

デフォルト: `True`

マージされたホストリストで重複を取り除きます。そのため、与えられたすべてのホスト文字列は 1 度しか使用されません (例えば、 `@hosts` + `@roles` もしくは `-H` と `-R` の組み合わせ使用時)。

`False` にセットされると、このオプションは重複を排除しなくなります。そうすると、明示的に同一ホストに対して一つのタスクを複数回実行したい場合 (例えば、シリアルに動作するけれども並列で実行させたい場合) に、そのように実行できるようになります。

バージョン 1.5 で追加.

disable_known_hosts

デフォルト: `False`

`True` の場合、SSH 層はユーザーの known-hosts ファイルの読み込みをスキップします。ホストキーが実際には有効なのに "既知のホスト" が変わった場合 (例えば、EC2 などのクラウドサーバなど) に発生する例外を避けるのに便利です。

参考:

`--disable-known-hosts`, *SSH* **の動作**

eagerly_disconnect

デフォルト: `False`

`True` にすると `fab` は実行の最後ではなく各個別のタスク実行後に接続を切ります。これにより、一般的な未使用セッションのパイルアップや、プロセスごとに開けるファイル制限やネットワークのハードウェアに伴う問題の発生を避けることができます。

注釈: アクティブの場合、この設定は、最後ではなく出力全体を通して接続解除のメッセージを表示します。将来のリリースでは改善されるかもしれません。

---

effective_roles

デフォルト: `[]`

実行中のコマンドのロールのリストが `fab` によって設定されます。情報目的のみです。

バージョン 1.9 で追加.

参考:

実行モデル

exclude_hosts

デフォルト: `[]`

`fab` 実行中に **スキップする** ホスト文字列のリストを指定します。通常は *`--exclude-hosts/-x`* 経由でセットされます。

バージョン 1.1 で追加.

fabfile

デフォルト: `fabfile.py`

`fab` が fabfile を読み込むときに探すファイル名パターンです。特定のファイルを指定するにはそのファイルへのフルパスを用います。これを fabfile 内に指定するのはもちろん意味がありませんが、 `.fabricrc` 内やコマンドライン上で指定することができます。

参考:

`--fabfile`, *fab* **オプションと引数**

gateway

デフォルト: `None`

指示されたホストを通した SSH ドリブンなゲートウェイ接続を可能にします。値は *env.host_string* で用いられるような通常の Fabric のホスト文字列でなければいけません。この値がセットされると、新規に作成される接続はそのリモート SSH デーモンを通して SSH のトラフィックが最終目的地に送られるようにセットされます。

バージョン 1.5 で追加.

参考:

`--gateway`

host_string

デフォルト: `None`

`run`、 *`put`* などの実行時に Fabric が接続するカレントのユーザー/ホスト/ポートを定義します。前回セットされたホストリストを順に処理するときに `fab` によってセットされます。また、Fabric をライブラリとして使用するときに手動でセットすることもできます。

参考:

実行モデル

forward_agent

デフォルト: False

True にセットされると、ローカルの SSH エージェントをリモート側に転送できるようになります。

バージョン 1.4 で追加.

参考:

--forward-agent

host

デフォルト: None

fab によって env.host_string のホスト名部分にセットされます。情報目的のみです。

hosts

デフォルト: []

タスクごとのホストリストの作成時に利用されるグローバルなホストリストです。

参考:

--hosts, 実行モデル

keepalive

デフォルト: 0 (例: keepalive 無し)

SSH のキープアライブ間隔に使用するために指定する整数値です。基本的に SSH のコンフィグオプションの ClientAliveInterval にマップされます。ネットワークハードウェアの問題等により接続がタイムアウトするときに便利です。

参考:

--keepalive

バージョン 1.1 で追加.

key

デフォルト: None

接続認証中に使用される SSH キーを含む文字列もしくはファイル等のオブジェクトです。

注釈: SSH キーを利用するためのもっとも一般的な方法は *key_filename* をセットすることです。

バージョン 1.7 で追加.

key_filename

デフォルト: `None`

接続試行時の SSH キーファイルへのファイルパスを参照する文字列もしくは文字列のリストです。SSH 層に直接渡されます。 `-i` でセット/追加できます。

参考:

SSHClient.connect() に関する Paramiko のドキュメント

linewise

デフォルト: `False`

一般的には並列モード実行時に、文字/バイトごとではなく行ごとにバッファするようにします。 `--linewise` 経由で有効化できます。このオプションは *env.parallel* によって暗黙に定義されます。 `linewise` が False でも `parallel` が True なら linewise の挙動をとります。

参考:

出力 行単位 対 バイト単位 *(Linewise vs bytewise)*

バージョン 1.3 で追加.

local_user

ローカルのシステムユーザー名を含む読み込み専用の値です。これは *user* の初期値と同じ値ですが、 *user* は CLI 引数、Python のコードもしくは特定のホスト文字列によって変更可能で、*local_user* は常に同じ値を含みます。

no_agent

デフォルト: `False`

`True` にすると、鍵ベースの認証時に SSH 層が動作中の SSH エージェントを探さないようにします。

バージョン 0.9.1 で追加.

参考:

--no_agent

no_keys

デフォルト: `False`

`True` にすると、SSH 層が `$HOME/.ssh/` フォルダーから秘密鍵ファイルを読み込まないようにします (もちろん、 `fab -i` 経由で明示的に読み込まれた鍵ファイルは利用されます)。

バージョン 0.9.1 で追加.

参考:

-k

parallel

デフォルト: `False`

`True` の時、強制的にすべてのタスクを並列に実行します。 *env.linewise* に影響します。

バージョン 1.3 で追加.

参考:

--parallel, **並列実行**

password

デフォルト: `None`

リモートホスト接続時および/もしくは *sudo* のプロンプトの答える時に SSH 層が使うデフォルトのパスワードです。

参考:

--initial-password-prompt, *env.passwords*, **パスワード管理**

passwords

デフォルト: {}

この辞書は主に内部で利用され、ホスト文字列ごとのパスワードキャッシュとして自動的に埋められます。キーは完全な **ホスト文字列** で、値はパスワード (文字列) です。

警告: この辞書を手動で修正もしくは生成したのなら、ユーザーとポートの値をともなった **条件を完全に満たしたホストストリングを使用しなくてはなりません**。ホストストリング API についての詳細は上のリンクを参照してください。

参考:

パスワード管理

バージョン 1.0 で追加.

path

デフォルト: `''`

run/*sudo*/*local* のコマンド実行時の `$PATH` シェル環境変数のセットに利用されます。この値の管理には、直接のセットではなく *path* コンテキストマネージャの利用をおすすめします。

バージョン 1.0 で追加.

pool_size

デフォルト: `0`

タスクの並列実行時に利用する並列プロセスの数をセットします。

バージョン 1.3 で追加.

参考:

--pool-size, **並列実行**

prompts

デフォルト: `{}`

`prompts` 辞書はユーザーが対話式のプロンプトをコントロールできるようにします。この辞書内のキーがコマンドの標準出力ストリーム内に見つかれば、Fabric は自動的に対応する辞書の値を応答します。

バージョン 1.9 で追加.

port

デフォルト: `None`

ホストリストを順次処理するときに fab によって env.host_string のポート部分にセットされます。デフォルトポートを指定するときにお利用されます。

real_fabfile

デフォルト: None

読み込んだ fabfile へのパスが fab によってセットされます。情報取得のみです。

参考:

fab **オプションと引数**

remote_interrupt

デフォルト: None

Ctrl-C がリモートでの中断をもたらすかローカルでのキャプチャにするかをコントロールします。設定は:

- None (デフォルト): *open_shell* がリモートの割り込み挙動を表示するだけで、 *run*/*sudo* がローカルの割り込みをキャプチャします。
- False: *open_shell* がローカルでキャプチャします。
- True: すべての機能が割り込みをリモート側に送ります。

バージョン 1.6 で追加.

rcfile

デフォルト: $HOME/.fabricrc

Fabric のローカルの設定ファイル読み込み時に利用されるパスです。

参考:

--config, *fab* **オプションと引数**

reject_unknown_hosts

デフォルト: False

True の場合、ユーザーの既知のホストファイルに載っていないホストへの接続時に SSH 層が例外を発生させます。

参考:

--reject-unknown-hosts, *SSH* **の動作**

system_known_hosts

デフォルト: None

セットするには、 known_hosts ファイルへのパスをセットします。SSH 層はユーザーの既知のホストファイルを読む前にこのファイルを読みます。

参考:

SSH の動作

roledefs

デフォルト: {}

ホストリストへマッピングするロールを定義する辞書です。

参考:

実行モデル

roles

デフォルト: []

タスクごとのホストリスト作成時に使用されるグローバルなロースリストです。

参考:

--roles, 実行モデル

shell

デフォルト: /bin/bash -l -c

例えば *run* などとともにコマンドを実行する時のシェルラッパーに使われる値です。 <env.shell> "<command goes here>" 形式で存在する必要があります。例えば、デフォルトで使用する Bash の -c オプション (その値をコマンド文字列として扱う) です。

参考:

--shell, FAQ - デフォルトシェルとしての bash, 実行モデル

skip_bad_hosts

デフォルト: False

True の時、 fab (もしくは *execute* を使った fab 以外の利用) が接続できないホストをスキップします。

バージョン 1.4 で追加.

参考:

--skip-bad-hosts, **特定のホストの除外**, **実行モデル**

skip_unknown_tasks

デフォルト: `False`

`True` の時、 `fab` (もしくは *execute* を使った `fab` 以外の利用) が見つからなかったタスクを中止せずにスキップします。

参考:

--skip-unknown-tasks

ssh_config_path

デフォルト: `$HOME/.ssh/config`

代替の SSH 設定ファイルパスの指定できます。

バージョン 1.4 で追加.

参考:

--ssh-config-path, **ネイティブの *SSH config* ファイルの活用**

ok_ret_codes

デフォルト: `[0]`

このリストのリターンコードが *run*/*sudo*/*sudo* への呼び出しを成功したとみなすか否かを決定するために利用されます。

バージョン 1.6 で追加.

sudo_prefix

デフォルト: `"sudo -S -p '%(sudo_prompt)s' " % env`

sudo 呼び出しのコマンド文字列の先頭に追加された実際の `sudo` コマンド。デフォルトのリモート `$PATH` の `sudo` 権限を持っていないユーザーやその他の変更 (パスワード無し sudo 実施時に `-p` を取り除いたり) を必要とするユーザーにとってはこの変更は便利です。

参考:

sudo オペレーション; *env.sudo_prompt*

sudo_prompt

デフォルト: "sudo password:"

リモートシステム上の sudo プログラムに渡されることにより、Fabric がそのパスワードプロンプトを正しく認識できます。

参考:

sudo オペレーション; *env.sudo_prefix*

sudo_user

デフォルト: None

sudo の user 値が与えられたなった時のフォールバック値として使われます。*settings* との組み合わせが便利です。

参考:

sudo

tasks

デフォルト: []

fab によって、実行中のコマンドのために実行すべきタスクの完全な一覧がセットされます。情報取得目的のみです。

参考:

実行モデル

timeout

デフォルト: 10

ネットワーク接続のタイムアウトを秒で。

バージョン 1.4 で追加.

参考:

--timeout, *connection_attempts*

use_shell

デフォルト: `True`

run/*sudo* への `shell` 引数のように振る舞うグローバルな設定です。`False` にセットされるとオペレーションは `env.shell` 内の実行されたコマンドをラップしません。

use_ssh_config

デフォルト: `False`

`True` にセットすると、Fabric はローカルの SSH コンフィグファイルを読み込みます。

バージョン 1.4 で追加.

参考:

ネイティブの *SSH config* ファイルの活用

user

デフォルト: ユーザーのローカルユーザー名

リモートホスト接続時に SSH 層によって利用されるユーザー名です。グローバルにセットすることも可能で、ホスト文字列に明示的にセットされていない場合にも利用されます。しかし、そのように明示された場合、この変数は一時的に現行値で上書きされます。例えば、その時に接続されているユーザーとして常に表示されます。

これを説明するには、次の fabfile を:

```
from fabric.api import env, run

env.user = 'implicit_user'
env.hosts = ['host1', 'explicit_user@host2', 'host3']

def print_user():
    with hide('running'):
        run('echo "%(user)s"' % env)
```

そしてこれを実行してみます:

```
$ fab print_user

[host1] out: implicit_user
[explicit_user@host2] out: explicit_user
[host3] out: implicit_user

Done.
```

(次のページに続く)

(前のページからの続き)

```
Disconnecting from host1... done.
Disconnecting from host2... done.
Disconnecting from host3... done.
```

ご覧のとおり、host2 に対して実行中は env.user は "explicit_user" にセットされましたが、その後、前の値 ("implicit_user") に戻されています。

注釈: env.user はいまのところ少し紛らわしい (設定**および**情報取得目的に用いられるので) ので、将来的には変更されると考えてください。情報取得の部分は別の env 変数に分けられると思います。

参考:

実行モデル, *--user*

version

デフォルト: Fabric の現行バージョン文字列

主に情報取得が目的です。変更は、おそらく何も壊さないですがおすすめしません。

参考:

--version

warn_only

デフォルト: False

run/*sudo*/*local* がエラー状態に遭遇した時に中止の代わりに警告を発するかどうかを指定します。

参考:

--warn-only, **実行モデル**

2.2 実行モデル

概要とチュートリアル をすでに読んでいるのなら、Fabric が基本的なケース (単一のホストに対する単一のタスク) でどのように動作するのかお分かりかと思います。しかし、多くの場合、複数のタスクおよび/または複数のホストに対する実行を望むことでしょう。大きなタスクを小さくて再利用可能なパーツに分けたり、一群のサーバで古いユーザを削除したりすることを望むかと思います。そのようなシナリオでは、いつ、どのようにタスクを実行するかの特定のルールが必要になってきます。

このドキュメントでは Fabric の実行モデルを説明します。メインの実行ループ、ホストリストの定義方法、どのように接続が行われるかなどが含まれています。

2.2.1 実行ストラテジー

デフォルトでは、Fabric は単一でシリアルな実行メソッドです。ただし、Fabric 1.3 からはパラレルモードも利用できるようになっています (**並列実行** 参照)。このデフォルトの挙動は次のようになっています:

- タスクの一覧が作成されます。この時点ではこのリストは単に *fab* に与えられた引数で、与えられた順番も保持します。
- さまざまなソースから、各タスクごとにタスク用のホストリストが生成されます (詳細は **ホストリストがどのように作られるか** を参照)。
- タスクリストが順番に実行され、各タスクはホストリストの各ホストごとに一度づつ実行されます。
- ホストリストにホストのないタスクはローカルのみと判断され、常に一度のみしか実行されません。

したがって、次の fabfile が実行されると:

```
from fabric.api import run, env

env.hosts = ['host1', 'host2']

def taskA():
    run('ls')

def taskB():
    run('whoami')
```

次のように起動されます:

```
$ fab taskA taskB
```

そして、次のように Fabric が実行します:

- `taskA` を `host1` に対して実行
- `taskA` を `host2` に対して実行
- `taskB` を `host1` に対して実行
- `taskB` を `host2` に対して実行

このアプローチはとても単純なものですが、タスク機能の分かりやすい構成を可能にし、(マルチホスト機能を個々の関数呼び出しに落としこむような他のツールとは地 g って) 出力を内省したり、与えられたコマンドのコードを返して、次に何をするか決定したりできるシェルスクリプトのようなロジックを可能にします。

2.2.2 タスクの定義

Fabric タスクの構成や構造化についての詳細はを **タスクの定義** ご覧ください。

2.2.3 ホストリストの定義

Fabric を単一のビルドシステムとして使うのでなければ (可能ですが、主なユースケースではありません)、特定のリモートホストに対してタスクを実行できなければ、役立たずでしょう。Fabric では、ホストを指定する方法がたくさんあります。グローバルからタスクごとまでのスコープで、また、必要に応じて組み合わせたり、マッチさせたりすることもできます。

ホスト

ここでは、ホストとは "ホスト文字列" で、ユーザー名、ホスト名、ポートを `username@hostname:port` の形式で組み合わせた Python の文字列のことを指します。ユーザーおよび/またはポート (と関連付けられた `@` もしくは `:`) は省略可能で、その場合は実行ユーザーのローカルのユーザー名および/もしくはポート 22 がそれぞれ利用されます。したがって、`admin@foo.com:222`、`deploy@website`、`nameserver1` はいずれも有効なホスト文字列です。

IPv6 アドレスのノーテーション、例えば `::1`、`[::1]:1222`, `user@2001:db8::1`、`user@[2001:db8::1]:1222` もサポートしています。角括弧はアドレスとポート番号を別にするときだけ必要です。ポート番号を使用しないのなら、角括弧は任意です。また、コマンドラインの引数経由でホスト文字列を指定でき、その場合は、シェルに寄ってはカッコをエスケープする必要が出てくるかもしれません。

注釈: ユーザー/ホスト名は最後に見つかった `@` で分割されます。したがって、メールアドレスでのユーザー名も有効で、正しくパースされます。

実行中、Fabric は与えられたホスト文字列を正規化し、各部分 (ユーザー名/ホスト名/ポート) を環境辞書に保存し、必要なときにタスクが参照します。詳細は **環境辞書、** *env* を参照してください。

ロール

ホスト文字列は単一のホストにマップされますが、ホストをグループで用意したほうが都合のいい場合もあるでしょう。例えば、ロードバランサーの後ろにたくさんの Web サーバがあって、それをすべてアップデートしたい場合や "クライアントのすべてのサーバ" にタスクを実行したい場合です。ロールは、ホスト文字列のリストに対応した文字列を定義できる手段を提供します。これにより、毎回ホストリスト全体を書き出す代わりに、この文字列を指定することができます。

このマッピングは辞書 `env.roledefs` として定義され、利用するためには fabfile 内で指定する必要があります。例えば:

```
from fabric.api import env

env.roledefs['webservers'] = ['www1', 'www2', 'www3']
```

env.roledefs は当然ながらデフォルトでは空なので、どんな情報も失うおそれなく再割当てすることができます (もちろんこれを指定している他の fabfile を読み込まなければ):

```
from fabric.api import env

env.roledefs = {
    'web': ['www1', 'www2', 'www3'],
    'dns': ['ns1', 'ns2']
}
```

ロール定義にはホストのみの設定が必要というわけではなく、任意のロール固有の設定を持たせることも可能です。これはディクショナリとしてのロールと hosts キー配下のホストストリングで設定することができます。

```
from fabric.api import env

env.roledefs = {
    'web': {
        'hosts': ['www1', 'www2', 'www3'],
        'foo': 'bar'
    },
    'dns': {
        'hosts': ['ns1', 'ns2'],
        'foo': 'baz'
    }
}
```

リスト/反復可能なオブジェクトタイプに加えて、 env.roledefs の値 (もしくは辞書スタイル定義内の hosts キーの値) は呼び出し可能で、そのため、モジュールの読み込み時の代わりにタスク実行時にルックアップされたときに呼び出されます。(例えば、ロール定義を得るためにリモートサーバに接続することが可能で、fab --list などの呼び出し時に fabfile の読み込み時間の遅れの発生を心配する必要はありません)

ロールの使用は必須ではありません。サーバーの一般的なグループ化をするときに便利なようにしているだけです。

バージョン 0.9.2 で変更: roledefs の値として呼び出し可能な機能を追加。

ホストリストがどのように作られるか

ホストリストを指定する方法は全体であれ、タスクごとであれ、たくさんあります。またたいていの場合、これらの方法はマージするのではなくそれぞれの方法をオーバーライドできます (将来のリリースでは変更されるかもしれませんが)。これらの各方法は主に 2 つの部分、ホスト用とロール用に分けられます。

env 経由でグローバルに

ホストもしくはロールを設定するいちばん一般的な方法は環境辞書 *env*: `hosts` と `roles` に 2 つのキーバリューペアを設定する方法です。これらの変数の値は起動時、各タスクのホストリスト作成中にチェックされます。

したがって、これらの値はモジュールレベルで設定され、fabfile のインポート時に有効になります:

```
from fabric.api import env, run

env.hosts = ['host1', 'host2']

def mytask():
    run('ls /var/www')
```

単に `fab mytask` として動作させるこのような fabfile の場合、`mytask` が `host1` に対して、続いて `host2` に対して実行されます。

env 変数は **各タスク** でチェックされるので、必要に応じてタスク内で `env` を変更することができ、その変更はその後に続くタスクにも反映されます。

```
from fabric.api import env, run

def set_hosts():
    env.hosts = ['host1', 'host2']

def mytask():
    run('ls /var/www')
```

`fab set_hosts mytask` として実行すると、`set_hosts` はホストリストがからのため "local" タスクとなりますが、`mytask` は与えられた 2 つのホストに対して再び実行されます。

注釈: この手法は見せかけの "roles" の作成方法としてよく利用されていましたが、ロール機能が完全に実装されたので今は必要性が少なくなりました。とは言え、場合によっては今でも便利な方法です。

`env.hosts` と同じように、`env.roles` (`env.roledefs` と間違えにように!) が与えられると、`env.roledefs` 内を探すためにロール名のリストとして扱われます。

コマンドライン経由でグローバルに

モジュールレベルでの `env.hosts`、`env.roles`、`env.exclude_hosts` の設定に加え、コンマで分けた文字列引数としてコマンドラインのスイッチ *`--hosts/-H`* と *`--roles/-R`* に渡すことでもこれらの設定が可能です。:

```
$ fab -H host1,host2 mytask
```

こうした実行は env.hosts = ['host1', 'host2'] と同等で、引数パーサーはこれらの引数を探し、パース時に env を設定します。

注釈: これらのスイッチを単一のホストやロールを設定するためだけに利用するのは、可能ですし、たしかに一般的です。Fabric は、与えられた文字列に対して string.split(',') を単に呼び出しているだけで、コンマがない文字列は単一のアイテムリストとして扱われます。

これらのコマンドラインスイッチは、fabfile が **読み込まれる前** に解釈されるということは重要なので、これに留意してください。つまり、fabfile 内の env.hosts もしくは env.roles で再割当てされた値がこれらのスイッチを上書きするのです。

コマンドラインのホストと fabfile で指定されたホストの非破壊的なマージをしたい場合は、fabfile で env.hosts.extend() を使うようにしてください:

```
from fabric.api import env, run

env.hosts.extend(['host3', 'host4'])

def mytask():
    run('ls /var/www')
```

この fabfile を fab -H host1,host2 mytask として起動すると、 mytask の実行時に ['host1', 'host2', 'host3', 'host4'] が env.hosts に含まれます。

注釈: env.hosts は単なる Python のリストオブジェクトなので env.hosts.append() やその他のメソッドも使うことができます。

コマンドライン経由でタスクごとに

いつも同じホストリストに対してすべてのタスクを実行したいのなら、グローバルにホストリストを設定するといいでしょう。しかし常にそうとは限らないので、Fabric はより粒度の細かい方法をいくつか提供していて、単一のタスクのみに適用されるホストリストを指定することができます。最初の方法はタスク引数を利用する方法です。

fab **オプションと引数** でも説明したように、特別なコマンドラインシンタックスによってタスクごとに引数を指定できます。タスク機能に実際に引数を設定できるのに加え、host、 hosts、 role、 roles "引数" をセットするのに使われます。これはホストリストの組み立て時に Fabric によって解釈されます (そして、タスクに渡された引数から取り除かれます)。

注釈: コンマはすでにタスク引数の分割に使われているので、各ホスト文字列やロール名の設定にはそれぞれの `hosts` もしくは `roles` 引数でセミコロンを使う必要があります。さらに、シェルがセミコロンを解釈しないように引数はクォートされていなければなりません。

以下の fabfile を見てみると、これまで使っていたものと同じですが、ホストの情報はまったく設定されていません:

```
from fabric.api import run

def mytask():
    run('ls /var/www')
```

`mytast` 用のタスクごとのホストを指定するには、以下のように実行します:

```
$ fab mytask:hosts="host1;host2"
```

これはどんなホストリストも上書きし、`mytask` は常にこの 2 つのホストに対して実行されます。

デコレーター経由でタスクごとに

与えられたタスクが常に事前に定義されたホストリストに対して実行される場合は、fabfile 内でのこのリストの指定を望むことでしょう。これは `hosts` もしくは `roles` デコレータでタスク関数をデコレートすることで可能です。これらのデコレータは変数引数リストをとります。例えば:

```
from fabric.api import hosts, run

@hosts('host1', 'host2')
def mytask():
    run('ls /var/www')
```

これは、繰り返し可能な単一の引数を取ることもできます。例えば:

```
my_hosts = ('host1', 'host2')
@hosts(my_hosts)
def mytask():
    # ...
```

これが利用されると、このデコレータはこの特定のタスクのホストリスト用の `env` のチェックをオーバーライドします (`env` が変更されるわけではありません。単に無視されます)。そして、たとえ上記 fabfile で `env.hosts` が指定されていたり *fab* が *`--hosts/-H`* を使っていても `mytask` は `['host1', 'host2']` のホストリストに対して実行されます。

とは言え、デコレータのホストリストは、上のセクションで説明したタスクごとのコマンドラインを **上書きすることはありません**。

優先順位

ここまで、ホストリスト設定のどの方法が他の方法よりも優先するかについて説明してきました。もっと明確にするため、以下に簡単にまとめます:

- タスクごとのコマンドラインホストリスト (`fab mytask:host=host1`) は他のすべてを完全にオーバーライドする。
- タスクごとのデコレータで指定されたホストリスト (`@hosts('host1')`) は `env` 変数をオーバーライドする。
- fabfile でグローバルに指定されたホストリスト (`env.hosts = ['host1']`) はコマンドラインでのホストリストをオーバーライド **できる** が、それは自分が注意していない場合 (もしくは意図的に行った場合) のみ。
- コマンドラインでグローバルに設定されたホストリストは (`--hosts=host1`) `env` 変数を初期化するが、それだけしかしない。

この優先順位はより一貫性を持たせるために将来的には少し変更されるかもしれません (例えば、コマンドラインのタスクごとのリストがコード内のリストに優先されるのと同じように *`--hosts`* が `env.hosts` より優先される) が、それは後方互換性リリースの時だけです。

ホストリストの結合

ホストリストがどのように作られるか で言及している様々なソース間のホストを "結合" する方法はありません。もし `env.hosts` が `['host1', 'host2', 'host3']` に設定されていて、関数ごと (例えば `hosts` 経由) のホストリストが `['host2', 'host3']` と設定されている場合、この関数は `host1` に対しては **実行されません** 。タスクごとのデコレータホストリストがの方が優先されるからです。

とは言え、与えられた各ソースでは、もしロールとホストの **両方** が設定された場合、両方ともひとつのホストリストにマージされます。例えば、以下のように両方のデコレータが使われている fabfile を見てみましょう:

```
from fabric.api import env, hosts, roles, run

env.roledefs = {'role1': ['b', 'c']}

@hosts('a', 'b')
@roles('role1')
def mytask():
    run('ls /var/www')
```

`mytask` 実行時にはコマンドラインのホストやロールが与えられていないとすると、この fabfile は `role1` と `hosts` 呼び出しの中身の結合されたホストリスト `['a', 'b', 'c']` に対して `mytask` を実行します。

ホストリストの重複

デフォルトでは、 **ホストリストの結合** をサポートするため Fabric は最終的なホストリストから重複を取り除くので、与えられるどのホスト文字列も一度だけしか対象になりません。とは言え、これでは有益なこともある同じターゲットホストに対して明示的/意図的な複数回タスクの実行ができません。

重複除去の機能を無効にするには *env.dedupe_hosts* を `False` にします。

特定のホストの除外

時には、ひとつもしくは複数の特定のホストを除外すると便利な時もあります。例えば、あるロールや自動的に生成されたホストリストから引き出されるいくつかの必要のないホストをオーバーライドする場合などです。

注釈: Fabric 1.4 からは接続できなかったホストをスキップする *skip_bad_hosts* を使うこともできます。

グローバルでは *--exclude-hosts/-x* でホストの除外ができます:

```
$ fab -R myrole -x host2,host5 mytask
```

`myrole` が `['host1', 'host2', ..., 'host15']` として定義されている場合、上のように実行すると、有効なホストリスト `['host1', 'host3', 'host4', 'host6', ..., 'host15']` となります。

> **注釈:** このオプションを利用しても `env.hosts` は修正されません。メインの実行ループがリクエストされたホストをスキップするだけです。

除外は、付加的な `exclude_hosts` 引数を利用することでタスクごとに指定することもできます。これは、上記で言及したタスクごとの `hosts` と `roles` 引数と同じように実装されていて、実際のタスク実行にストリップされます。以下の例では、上記のグローバル除外と同じ結果になります:

```
$ fab mytask:roles=myrole,exclude_hosts="host2;host5"
```

ホストリストはタスクごと引数の `hosts` と同じようにセミコロンで分けられます。

除外の結合

ホスト除外リストは、ホストリスト自身と同じように、宣言されている "levels" が違うリスト間ではマージされません。例えば、グローバルな `-x` オプションは、デコレータやキーワード引数でセットされたタスクごとのホストリストに影響しません。また、タスクごとの `exclude_hosts` キーワード引数もグローバルな `-H` リストに影響しません。

このルールにはひとつだけ小さな例外があります。`@hosts` もしくは `@roles` 経由でのホストリストの分析時に、CLI レベルのキーワード引数 (`mytask:exclude_hosts=x,y`) が取り入れられます。したがって、 `@hosts('host1', 'host2')` でデコレートされているタスク関数が `fab taskname:exclude_hosts=host2` として実行されると、 `host1` だけに対してのみ実行されます。

ホストリストのマージに関しては、現行では機能は限定的 (実装をシンプルに保つためでもあります) で、将来のリリースでは拡張されるかもしれません。

2.2.4 `execute` での賢いタスクの実行

バージョン 1.3 で追加.

ここの情報のほとんどは、最初の例で `fab taskA taskB` を呼び出したように、*fab* 経由で実行される "トップレベル" のタスクに作用します。とは言え、以下の "meta" タスクのような複数タスクの実行をまとめたものも便利な時があります。

Fabric 1.3 以前は **ライブラリの利用** で書かれていたように手動で行う必要がありました。Fabric のデザインは魔法的な挙動を避けているので、単純にタスクを **呼び出しても** `roles` のようなデコレータは考慮 **しません** 。

Fabric 1.3 では新しく *`execute`* ヘルパー関数が追加されました。これは最初の引数としてタスクオブジェクトもしくはタスク名を取ります。コマンドラインから与えられたタスクを呼び出すのと同じくらい効率的に利用できます。上の **ホストリストがどのように作られるか** で与えられたすべてのルールが適用されます。(*`execute`* への `hosts` と `roles` キーワード引数は、他のすべてのホスト/ロール設定方法をオーバーライドする ***CLI* のタスクごとの引数** に類似しています)

例として、ウェブアプリケーションをデプロイする 2 つの別個に定義された fabfile をあげます。

```
from fabric.api import run, roles

env.roledefs = {
    'db': ['db1', 'db2'],
    'web': ['web1', 'web2', 'web3'],
}

@roles('db')
def migrate():
    # Database stuff here.
    pass

@roles('web')
def update():
    # Code updates here.
    pass
```

Fabric <=1.2 では、`migrate` を DB サーバに対して確実に実行し、`update` を Web サーバに対して確実に実行する唯一の方法 (`env.host_string` 操作の短いマニュアル) は両方をトップレベルのタスクとして呼び出す方

法しかありませんでした:

```
$ fab migrate update
```

Fabric >=1.3 ではメタタスクのセットアップに *execute* が使えます。 import の行を以下のようにします:

```
from fabric.api import run, roles, execute
```

そして、ファイルの最後に次を追加します:

```
def deploy():
    execute(migrate)
    execute(update)
```

これだけです; roles デコレータが期待通りに履行し、以下の実行シーケンスの結果になります:

- db1 に対して migrate
- db2 に対して migrate
- web1 に対して update
- web2 に対して update
- web3 に対して update

警告: このテクニックは、ホストリストを自分では持たないタスク (これにはグローバルなホストリスト設定も含まれます) は一度しか実行されないので動作します。もし、複数ホストに対して実行される "通常の" タスク内で利用された場合、*execute* への呼び出しは複数回実行され、結果としてサブタスク呼び出しの倍数分実行されるので、お気をつけて!

自分の execute 呼び出しを 1 度だけの呼び出しにするには runs_once デコレータを使います。

参考:

execute, runs_once

マルチホストの結果へのアクセスに execute を活用する

Fabric の実行がひと仕事ある場合、特に並列にある場合、最後にたくさんあるホストごとの結果の値を、例えばテーブルサマリーの表示や計算の実行をするためなど、ひとまとめにしたいことがあると思います。

Fabric のデフォルトの "ナイーブ" モード (あなたが頼りにしている、ホストリストに対するあなたのために行う Fabric の繰り返し処理のモードです) ではこれはできませんが、 *execute* を使うととても簡単です。単純に実際

の分割したタスクの呼び出しから *execute* との実行をコントロールする "meta" タスクの呼び出しにスイッチします。

```
from fabric.api import task, execute, run, runs_once

@task
def workhorse():
    return run("get my infos")

@task
@runs_once
def go():
    results = execute(workhorse)
    print results
```

上の例では、 workhorse は Fabric で可能なこと、文字通り古い "naive" なタスクはすべて可能です。なにか有益なことを返す必要がある時を除いてですが。

go は新しいエントリーポイント (fab go などとして実行されます) で、その仕事は *execute* 呼び出しから results 辞書を取り出し、それに対して必要なことをなんでもすることです。返り値の構造についての詳細は API ドキュメントをご覧ください。

ホストリストの動的セットとの execute の利用

実行時にターゲットホストリストの参照を並行で行うのは Fabric の中級から上級のよくあるユースケースです (**ロール** の利用では十分ではない場合)。 execute は以下のようにこれをとても簡単に実現できます:

```
from fabric.api import run, execute, task

# For example, code talking to an HTTP API, or a database, or ...
from mylib import external_datastore

# This is the actual algorithm involved. It does not care about host
# lists at all.
def do_work():
    run("something interesting on a host")

# This is the user-facing task invoked on the command line.
@task
def deploy(lookup_param):
    # This is the magic you don't get with @hosts or @roles.
    # Even lazy-loading roles require you to declare available roles
    # beforehand. Here, the sky is the limit.
    host_list = external_datastore.query(lookup_param)
    # Put this dynamically generated host list together with the work to be
    # done.
    execute(do_work, hosts=host_list)
```

例えば、 external_datastore が単純な "データベース内をタグでホストをルックアップ" するサービスなら、そして自分のアプリケーション スタックに関連するタグが付けられたすべてのホストに対してタスクを実行するのなら、上記を以下のように呼び出すことができます:

```
$ fab deploy:app
```

ちょっと待って! DB サーバ上のデータのマイグレーションがどっかに行ってしまいました。ソースリポジトリのマイグレーション用のコードを修正して、DB ボックスだけに再度どプロイしてみましょう:

```
$ fab deploy:db
```

このユースケースはロールに似ていますが、もっと潜在力があり、決してひとつの引数だけに限定されるものでもありません。どのようにもタスクを定義でき、必要に応じてどのようにも外部のデータストアにクエリーを行うことができます。結局のところ、ただの Python なのです。

別のアプローチ

上記と似ているけれども、execute 呼び出しを明示する代わりに fab の機能を利用して連続で複数タスクを呼び出すには、ホストリストの参照タスクでの *env.hosts* を変化させ、同じセッションで do_work を呼び出します:

```
from fabric.api import run, task

from mylib import external_datastore

# Marked as a publicly visible task, but otherwise unchanged: still just
# "do the work, let somebody else worry about what hosts to run on".
@task
def do_work():
    run("something interesting on a host")

@task
def set_hosts(lookup_param):
    # Update env.hosts instead of calling execute()
    env.hosts = external_datastore.query(lookup_param)
```

そして次のように実行されます:

```
$ fab set_hosts:app do_work
```

その前のアプローチと比べた時のこちらのアプローチの利点は do_work をどんな "workhorse" タスクとも入れ替え可能ということです:

```
$ fab set_hosts:db snapshot
$ fab set_hosts:cassandra,cluster2 repair_ring
$ fab set_hosts:redis,environ=prod status
```

2.2.5 失敗の扱い

タスクリストが構築されると、Fabric は **実行ストラテジー** で説明したようにこのタスクの実行を開始し、そのホストリスト全体にすべてのタスクが実行されます。とは言え、Fabric はデフォルトでは "fail-fast" の挙動パターンになっていて、もし何かが失敗した場合、例えばリモートプログラムがノンゼロ返り値を返したり、自分の fabfile の Python コードが例外に遭遇したりした場合、すぐに停止します。

これは通常は望ましい挙動ですが、このルールにはたくさんの例外があり、そのため、Fabric はブール設定の `env.warn_only` を提供しています。これはデフォルトでは `False` になっていて、エラー状態はただちにそのプログラムの停止を意味します。しかし、もし失敗時に – *`settings`* コンテキストマネージャーなどで – `env.warn_only` が `True` に設定されていると、Fabric は警告メッセージを発しますがプログラムの実行は継続します。

2.2.6 接続

実は `fab` 自身ではリモートホストへの接続は行っていません。その代わり、それぞれのホストごとに対して一つのタスクをそれぞれ個別に実行するようにすることと、env 変数 `env.host_string` に正しい値がセットされていることを単純に保証しています。Fabric をライブラリとして活用したいユーザーは、手動で行うことにより同じような動作を達成することができます (とは言え、Fabric 1.3 では *`execute`* の利用をのほうが好ましく、より強力です)。

`env.host_string` は、(その名称がほのめかしているように) "カレントの" ホスト文字列で、ネットワークを利用する関数が実行されるときに、どの接続を行うか (もしくは再利用するか) を Fabric が決定するために利用されます。 *`run`* や *`put`* のようなオペレーションは、ホスト文字列を SSH 接続オブジェクトにマップしている共有辞書内の参照キーとして `env.host_string` を利用します。

注釈: この接続用の辞書 (今のところ `fabric.state.connections` にあります) はキャッシュとして振る舞い、オーバーヘッドを減らすために可能なら前回作成された接続を返そうとし、そうした接続がなければ新たに作成します。

レイジーな接続

接続は各オペレーションによって駆動されるので、Fabric は実際に必要になるまで接続を行いません。以下の例を見てください。このタスクはリモートサーバとのやりとりの前にローカルでハウスキーピング処理を行います:

```
from fabric.api import *

@hosts('host1')
def clean_and_upload():
    local('find assets/ -name "*.DS_Store" -exec rm '{}' \;')
```

(次のページに続く)

(前のページからの続き)

```
    local('tar czf /tmp/assets.tgz assets/')
    put('/tmp/assets.tgz', '/tmp/assets.tgz')
    with cd('/var/www/myapp/'):
        run('tar xzf /tmp/assets.tgz')
```

接続という観点からどのようなことが起こっているのか順に見て行きましょう:

1. 2 つの *local* 呼び出しがどんなものであれ接続をまったく行わないで実行されます
2. *put* が `host1` へ接続するための接続キャッシュを要求します
3. 接続キャッシュは該当のホスト文字列用の既存の接続を見つけられなかったので、新しい SSH 接続を作成し、その接続を *put* に返します
4. *put* がこの接続を通じてファイルをアップロードします
5. 最後に、 *run* 呼び出しが同じホスト文字列への接続のためのキャッシュを要求し、既存のキャッシュされた接続を自身の利用のために与えます

以上を基に推察すると、ネットワーク関連の操作を伴わないタスクは実際にどのような接続も始めないことがお分かりになるでしょう (ただし、もしあればですが、ホストリスト内の各ホストに一度実行されます)。

接続の解除

Fabric の接続キャッシュは接続自身を閉じることはありません。どのように使われていてもそのままにしておきます。 *fab* ツールがその状態を保持し、すべての開いている接続に対して繰り返し処理を行い、プログラムから抜け出る直前に (タスクの成功、不成功に関わらず) それらの接続を閉じます。

ライブラリのユーザーは、自分のプログラムから抜け出る前にすべての開いている接続を確実に閉じるようにする必要が有ります。これは、自分のスクリプトの最後で *disconnect_all* を呼び出すことによって実施可能です。

注釈: *disconnect_all* は将来的にはよりパブリックな場所に移動されるかもしれません。わたしたちは、Fabric のライブラリとしての側面をより堅固に、より整理されたものへしようと作業をしています。

複数回接続の試みとうまくいかないホストのスキップ

Fabric 1.4 では、エラーを伴った中止の前に、リモートサーバーへの接続が複数回試行されるかもしれません。Fabric はあきらめる前に *env.connection_attempts* での回数分、毎回 *env.timeout* で指定された秒数のタイムアウトまで接続をトライします (これらの値は、現行では以前の挙動に合わせるためデフォルトで 1 回と 10 秒になっていますが、必要に応じて安全に変更かのうです)。

さらに、サーバへの接続の完全な失敗が完全にハード的な停止ではない場合でも set *env.skip_bad_hosts* を `True` にすれば大抵の場合 (通常は初期接続) で Fabric は中止をする代わりに単に警告を発し、タスクの実行を続けます。

バージョン 1.4 で追加.

2.2.7 パスワード管理

Fabric はメモリー上に 2 層のパスワードキャッシュを保持し、特定の状況でのログインと sudo のパスワードを記憶します。これにより、複数システムで同じパスワードを共有しているとき[*1] やリモートシステムの `sudo` 設定が自身のパスワードをキャッシュしない時に退屈な再入力を避けるのに役立ちます。

最初の層は単純なデフォルトもしくはフォールバックのパスワードのキャッシュ、 *env.password* です (これは `--password` もしくは `--initial-password-prompt` 経由のコマンドラインでも設定可能です)。この env 変数は (空でない場合に) 特定のホストのキャッシュ (下を参照) がカレントの **ホスト文字列** 用のエントリを持っていない時に試される一つのパスワードを保持します。

env.passwords (複数形!) はユーザーごと/ホストごとのキャッシュとして利用され、ユーザー/ホスト/ポートの各組み合わせごとにもっとも最近入力されたパスワードを保持します (もしこの構造を手動で修正する場合は **この 3 つすべての値** を必ず含める必要があるということに **留意** してください)。このキャッシュのおかげで同一セッションでの複数の異なるユーザーおよび/またはホストへの接続で、それぞれ一度のパスワード入力だけで済ますことができます。(Fabric の以前のバージョンでは単一のデフォルトパスワードのキャッシュのみしか利用できなかったため、その前に入力されたパスワードは毎回無効になり、パスワードの再入力が必要になっていました)

設定やセッションが接続するホストの数にもよりますが、この env 変数のどちらかもしくは両方を設定すると便利でしょう。とは言え、Fabric は必要に応じて追加の設定なしでも自動的にこれらを入力します。

特に、ユーザーにパスワードプロンプトが表示されるたびに、入力された値は単一のデフォルトパスワードキャッシュと `env.host_string` のカレントの値のためのキャッシュの値の両方のアップデートに使われます。

2.2.8 ネイティブの SSH config ファイルの活用

コマンドラインの SSH クライアント (OpenSSH によって提供されているものなど) は、通常は `ssh_config` として知られる特定の設定フォーマットを利用し、プラットフォーム特有の場所の `$HOME/.ssh/config` (もしくは `--ssh-config-path`/*env.ssh_config_path* に与えられる任意のパス) にあるファイルからそれを読み込みます。このファイルはデフォルトもしくはホストごとのユーザ名、ホスト名のエイリアス、その他設定の切り替え (**エージェントフォワーディング** を利用するか否かなど) など、さまざまな SSH オプションの設定を可能にします。

Fabric の SSH 実装では、実際に SSH config ファイルがあればそこからこれらのオプションのサブセットを読み込むことが可能です。この挙動は後方互換性のためにデフォルトでは有効になっていませんが、お使いの fabfile の一番上で *env.use_ssh_config* を `True` にすることによって有効にすることができます。

これを有効にすると、次の SSH config 指示が読み込まれ、Fabric によって履行されます:

- `User` と `Port` は、次の方法で他に指定されない限り、適切な接続パラメータに利用されます:

[*1] 同一のパスワード設定に頼るよりも SSH の 鍵ベースのアクセス の利用を強くおすすめします。こちらのほうがかなり安全です。

  - グローバルに指定された User/Port は、該当の env 変数がセットされていなければ、カレントの初期値 (それぞれローカルのユーザー名と 22) の代わりに利用されます。
  - しかし、 *env.user*/*env.port* が **セットされていれば** 、グローバルの User/Port の値をオーバーライドします。
  - ホスト文字列自身の User/port の値 (例えば ‘‘hostname:222‘‘) は ssh_config の値を含むすべてをオーバーライドします。
- 通常の ssh と同じように HostName は与えられたホスト名で置き換えられます。 HostName example.com を指定している Host foo のエントリーは、Fabric にホスト名 'foo' を与えることができ、接続時に 'example.com' に展開されます。
- IdentityFile は *env.key_filename* を (置き換えるのではなく) 拡張します。
- ForwardAgent は "OR(論理和)" 方式で *env.forward_agent* を補完します。どちらかが真の値にセットされていれば、エージェントフォワーディングは有効になります。
- ProxyCommand は通常の ssh と同じようにホスト接続でのプロキシコマンドを動作させます。

注釈: もし SSH のトラフィックをゲートウェイに送るだけなら、ゲートウェイとして ProxyCommand を使う通常の ssh gatewayhost nc %h %p 方式よりも *env.gateway* の方がより効率的な接続方法です。

注釈: もし SSH config ファイルが ProxyCommand を含んでいて **なおかつ** *env.gateway* が None 以外の値にセットされている場合、env.gateway が優先され ProxyCommand は無視されます。

もしすでに SSH config ファイルが作成されていれば、conf ファイルの内容全体で対処するよりも env.gateway (例えば *settings* 経由で) を修正するほうが容易でしょう。

2.3 fab オプションと引数

Fabric の最も普通の使い方はコマンドライン・ツールとしてです。fab は Fabric がインストールされた時にシェルの実行パスに置かれているはずです。 fab は Unix の良い住民になろうと努力しています。標準のコマンドライン・スイッチやヘルプ出力など。

2.3.1 基本的な使い方

最もシンプルなものでは fab は全くオプションなしで、あるいは 1 つか 2 つのタスク名を引数として呼ばれます。例えば:

```
$ fab task1 task2
```

概要とチュートリアル と **実行モデル** で詳しく説明されていますが、これは `task1` に続き `task2` を実行します。ただし Fabric がこれらの名前の Python 関数を含んだ fabfile を見つけられればですが。

しかしオプションやそれぞれのタスクに渡される引数を使ってさらにフレキシブルに拡張することができます。

2.3.2 任意のリモートシェル・コマンド

バージョン 0.9.2 で追加.

Fabric はあまり知られていないコマンドラインの記述で次のように呼ぶこともできます:

```
$ fab [options] -- [shell command]
```

`--` の後はすべて一時的な *run* コールとなり `fab` オプションとは解釈しません。モジュールレベルやコマンドラインでホストリストを定義すれば、これは一行の anonymous タスクのように使えます。

例えばシステムのカーネル情報をまとめて取得するにはこの様にします:

```
$ fab -H system1,system2,system3 -- uname -a
```

これは以下の fabfile と同じです:

```
from fabric.api import run

def anonymous():
    run("uname -a")
```

まるでこれが実行されたように:

```
$ fab -H system1,system2,system3 anonymous
```

ほとんどの場合タスクは fabfile に書くことになるでしょう。(一度使ったものは、もう一度使うことになるでしょうから) しかしこの機能は fabfile の接続設定を利用している時に手軽ですばやく SSH-borne コマンドを実行できます。

2.3.3 コマンドライン・オプション

利用可能なコマンドライン・オプションの概要は `fab --help` で表示できます。オプションの詳細は以下です。

注釈: `fab` は Python の optparse ライブラリを使います。これは Linux や GNU スタイルのショートやロングオプションが使えるということです。また自由にオプションと引数を使えます。例えば `fab task1 -H`

`hostname task2 -i path/to/keyfile` はもっとわかりやすく `fab -H hostname -i path/to/keyfile task1 task2` と書けます。

-a, --no_agent

env.no_agent を `True` にセット。SSH レイヤーにプライベートキーをアンロックするときに SSH エージェントと接続しないように強制します。

バージョン 0.9.1 で追加.

-A, --forward-agent

env.forward_agent を `True` にセット。エージェントのフォワードを有効にします。

バージョン 1.4 で追加.

--abort-on-prompts

env.abort_on_prompts を `True` にセット。入力を要求されたら Fabric を中断するよう強制します。

バージョン 1.1 で追加.

-c RCFILE, **--config**=RCFILE

env.rcfile を指定のファイルにセット。Fabric はこれを起動時に読み込み環境変数を更新します。

-d COMMAND, **--display**=COMMAND

指定したタスクのドキュメント文字列をすべて表示します。タスクの関数のシグネチャーは表示しません。 ドキュメント文字列を説明に使うのは良いアイデアです。(They're *always* a good idea, of course – just moreso here.)

--connection-attempts=M, **-n** M

接続を試みる回数を指定。*env.connection_attempts* をセットします。

参考:

env.connection_attempts, *env.timeout*

バージョン 1.4 で追加.

-D, --disable-known-hosts

env.disable_known_hosts を `True` にセット。Fabric はユーザーの SSH `known_hosts` をロードしません。

-f FABFILE, **--fabfile**=FABFILE

fabfile を検索するパターン (デフォルトは `fabfile.py`)、あるいは fabfile としてロードする任意のファイルパス (例えば `/path/to/my/fabfile.py`)

参考:

fabfile **の構成と使い方**

-F LIST_FORMAT, **--list-format**=LIST_FORMAT
: *--list* の出力フォーマットを制御します。. short は *--shortlist* と同じ、 normal は単にこのオプションの省略と同じです (すなわちデフォルト)、そして nested はネストされた名前空間のツリーを出力します。

  バージョン 1.1 で追加.

  参考:

  --shortlist, *--list*

-g HOST, **--gateway**=HOST
: *env.gateway* を HOST ホスト文字列にセット。

  バージョン 1.5 で追加.

-h, **--help**
: すべてのオプションと簡単な説明があるヘルプメッセージを表示して終了します。

--hide=LEVELS
: デフォルトで隠すコンマで区切った *output levels* のリスト。

-H HOSTS, **--hosts**=HOSTS
: *env.hosts* をコンマで区切ったホスト文字列のリストにセット。

-x HOSTS, **--exclude-hosts**=HOSTS
: 最終ホストリストの他に *env.exclude_hosts* に与えられたコンまで区切ったホスト文字列のリストをセットします。

  バージョン 1.1 で追加.

-i KEY_FILENAME
: ファイルのパスをセットすると SSH 認証ファイルとしてロードします。(通常はプライベート鍵) このオプションは何度も繰り返されるかもしれません。*env.key_filename* にセット (あるいは追加) します。

-I, **--initial-password-prompt**
: pre-fill *env.password* のためにセッション開始時にパスワードを要求するように強制します。(fabfile をロードしてオプションを解析してタスクを実行する直前)。

  これはパスワードを *--password* や fabfile の *env.password* で設定したくない時に (特に実行時に入力が不可能な並行セッションで) 走らせて放置 (fire-and-forget) できるので便利です。

  注釈: ここでの入力はモジュールレベルの *env.password* や *--password* を **上書き** します。

  参考:

  パスワード管理

-k

env.no_keys を `True` にセット。SSH レイヤーはホームのプライベート・キーファイルを探しません。

バージョン 0.9.1 で追加.

--keepalive=KEEPALIVE

env.keepalive を指定した整数にセット。SSH キープアライブの間隔を指定します。

バージョン 1.1 で追加.

--linewise

強制的に出力バッファをバイトごとでなく行ごとにします。*parallel execution* に有効でありまた必要になります。

バージョン 1.3 で追加.

-l, --list

通常通り fabfile をインポートして見つかったタスクのリストを表示して終了します。またタスクのドキュメント文字列の最初の行を表示します。必要なら切り詰めます。

バージョン 0.9.1 で変更: 出力にドキュメンテーション文字列 (docstring) を追加。

参考:

--shortlist, *--list-format*

-p PASSWORD, **--password**=PASSWORD

env.password を指定した文字列にセット。SSH 接続や `sudo` プログラムで使うデフォルトのパスワードです。

参考:

--initial-password-prompt

-P, --parallel

env.parallel を `True` にセット。タスクを並行処理します。

バージョン 1.3 で追加.

参考:

並列実行

--no-pty

env.always_use_pty を `False` にセット。すべての *run*/*sudo* の呼び出しは `pty=False` の指定と同じになります。

バージョン 1.0 で追加.

-r, --reject-unknown-hosts

env.reject_unknown_hosts を `True` にセット。Fabric はユーザーの SSH `known_hosts` で見つからないホストへの接続を中止します。

-R `ROLES`, **--roles**=`ROLES`

env.roles をコンマで区切られた role 名のリストにセット。

--set `KEY=VALUE,...`

デフォルトの y Fabric の env の値を任意の値にセット。しかしこの方法の優先度は低くなります。 – コマンドラインで指定された env の値を上書きできません。例えば:

```
fab --set password=foo --password=bar
```

これは `env.password = 'bar'` となります。

複数の `KEY=VALUE` はカンマで区切ります。例えば `fab --set var1=val1,var2=val2` のように。

基本的な文字列の他に `=VALUE` を省略することで env は True になります。(例えば `fab --set KEY`), and you may set values to また イコールは使って ‘‘VALUE‘‘ を省略することで空の文字列 (そして False と等価) にセットできます。(例えば `fab --set KEY=`)

バージョン 1.4 で追加.

-s `SHELL`, **--shell**=`SHELL`

env.shell を指定した文字列にセット。リモートコマンドの実行に使用するデフォルト・シェルを変更します。

--shortlist

--list に似ているがどんな修飾もなくインデンテーションやドキュメント文字列もない改行で区切られたタスク名。

バージョン 0.9.2 で追加.

参考:

--list

--show=`LEVELS`

デフォルトで表示されるコンマで区切られた *output levels* のリスト。

参考:

run, *sudo*

--ssh-config-path

env.ssh_config_path をセット。

バージョン 1.4 で追加.

参考:

ネイティブの *SSH config* ファイルの活用

--skip-bad-hosts

env.skip_bad_hosts をセット。Fabric は利用できないホストをスキップします。

バージョン 1.4 で追加.

--skip-unknown-tasks

env.skip_unknown_tasks をセットすると、見つからないタスクを Fabric がスキップします。

参考:

env.skip_unknown_tasks

--timeout=N, **-t** N

接続のタイムアウトを秒で指定します。 *env.timeout* をセット。

参考:

env.timeout, *env.connection_attempts*

バージョン 1.4 で追加.

--command-timeout=N, **-T** N

リモートコマンドのタイムアウトを秒で指定します。*env.command_timeout*

参考:

env.command_timeout,

バージョン 1.6 で追加.

-u USER, **--user**=USER

env.user を指定の文字列にセット。SSH 接続を行なうときにデフォルトのユーザー名として使われます。

-V, --version

Fabric のバージョンを表示して終了します。

-w, --warn-only

env.warn_only を `True` にセット。Fabric はコマンドがエラーを起こしても実行を続けます。

-z, --pool-size

env.pool_size をセット。並行処理で同時に実行するプロセスの数を指定します。

バージョン 1.3 で追加.

参考:

並列実行

2.3.4 Per-task 引数

コマンドライン・オプション で与えられたオプションは `fab` 呼び出しの全体に適用されます。順番によらずオプションは等しくタスクに与えられます。加えてタスクは Python 関数なので引数は実行時に渡されるのが望ましいでしょう。

この両方のニーズに答えるのが "per-task 引数" のコンセプトです。これはタスク名に添える特殊な構文です:

- 引数とタスク名を区別するにはコロン (`:`) を使います;
- コンマ (`,`) を引数を区別するために使います。(バックスラッシュでエスケープします。例えば `\,`);
- イコール (=) はキーワード引数、あるいは位置パラメーターの省略に使います。バックスラッシュでエスケープします。

このプロセスは文字列解析を呼び出します。すべての値は Python 文字列になります。(直感的な構文が見つかれば将来のバージョンの Fabric で改良します。)

例えば、 "create a new user" タスクはこの様に定義できます。(簡単にするために実際のロジックは省略):

```
def new_user(username, admin='no', comment="No comment provided"):
    print("New User (%s): %s" % (username, comment))
    pass
```

ユーザー名だけを指定できます:

```
$ fab new_user:myusername
```

あるいは任意のキーワード引数として扱います。

```
$ fab new_user:username=myusername
```

もし両方が与えられたら、再び位置パラメーターとして:

```
$ fab new_user:myusername,yes
```

あるいは Python のように mix and match で:

```
$ fab new_user:myusername,admin=yes
```

上記の `print` の呼び出しはコンマのエスケープの説明に有効です。このように:

```
$ fab new_user:myusername,admin=no,comment='Gary\, new developer (starts Monday)'
```

注釈: コンマはバックスラッシュでエスケープします。そうしないとシンタックスエラーになります。クォートも

また引数にスペースのようなシェルの特殊文字を含むときには必要になります。

上記のすべては期待される Python 関数の呼び出しに変換されます。例えば最後の呼び出しはこのように::(訳注:admin の値がおかしい)

```
>>> new_user('myusername', admin='yes', comment='Gary, new developer (starts Monday)')
```

Roles と hosts

the section on task execution で言及したようにいくつかの per-task キーワード引数があります。(`host`, `hosts`, `role` ,‘‘roles‘‘) これは実際にはタスク関数にマッピされませんが per-task host や role リストの設定に使われます。

これらの特殊な kwargs は args/kwargs から **取り除かれて** タスク関数に送られます。これはタスクが問題の kwargs を定義していないなら TypeErrors になりません。doesn’t define the kwargs in question. (これはまたこれらの名前の引数を定義して **いた** らこの方法で指定することはできません – 残念なことに必要な犠牲。)

注釈: kwargs に plural と singular の両方が与えられたら、plural の値が優先され singular は棄てられます。

これらの引数に plural を使うときセミコロン (`;`) を使います。コンマは引数で使います。さらにシェルがセミコロンを特殊文字として扱わないようにホストリストにはクォートを使うほうがよいでしょう。例えば:

```
$ fab new_user:myusername,hosts="host1;host2"
```

`new_user` タスク関数に送られるとき `hosts` kwarg は引数リストから除かれます。実際、Python の呼び出しは `new_user('myusername')` で, `['host1', 'host2']` のホストリストで関数が実行されます。

2.3.5 設定ファイル

Fabric は現在シンプルなユーザー設定ファイル、`fabricrc` (`fab` のための `bashrc` のような) を使います。これには一つ以上の キー/値 のペアの行があります。書式は `string.split('=')` に従います。現在は特定の文字列の設定に使われます。このようなキー/値のペアは `fab` が実行されるときに *env* の更新に使われます。これは fabfile が読み込まれる前にロードされます。

デフォルトで Fabric は `~/.fabricrc` 探しますが、これは `fab` のオプション *-c* で変更できます。

例えばワークステーションのユーザー名と SSH ログインのユーザー名が違うときにプロジェクトで使う fabfile の `env.user` を変更したくはないでしょう。(恐らく他の人も使うでしょうから) `fabricrc` ファイルをこの様にします:

```
user = ssh_user_name
```

これで fab を実行するときに fabfile の env.user は 'ssh_user_name' になります。他のユーザーの fabfile も同様にすると fabfile でデフォルトのユーザー名を気にしなくて済みます。

2.4 fabfile の構成と使い方

このドキュメントは fabfile に関する雑多なセクションを含んでいます。優れた fabfile の作成方法と作成後の利用の仕方の両方も含みます。

2.4.1 fabfile の探索

Fabric は Python のモジュール (例えば fabfile.py) やパッケージ (例えば __init__.py を含んでいる fabfile/ ディレクトリ) を読み込むことができます。デフォルトでは、(Python のインポート機構にしたがって) fabfile と名付けられた fabfile/ もしくは fabfile.py を探します。

fabfile の探索アルゴリズムは、起動しているユーザーのカレントワーキングディレクトリやその親ディレクトリを探します。したがって、"プロジェクト" ユース周り指向で、例えばコードツリーのルートに fabfile.py を保持しておきます。こうした fabfile は、ユーザーが fab を呼び出すツリー内であればどこであれ見つけられます。

探索される特定の名称は *-f* オプション付きのコマンドラインや fabfile の値をセットする *fabricrc* 行を追加することでオーバーライドできます。例えば、fabfile を fab_tasks.py と名づけたい場合、そのファイル名でファイルを作成し、 fab -f fab_tasks.py <task name> という具合に呼び出すか、~/.fabricrc に fabfile = fab_tasks.py を追加します。

与えられた fabfile 名にファイル名ではなくパス要素が含まれる場合 (例えば、../fabfile.py や /dir1/dir2/custom_fabfile)、それはファイルパスとして扱われ、その存在の確認がどのような種類の探索もなしに直に行われます。このモード時、チルダは展開されて適用されますので、例えば、~/personal_fabfile.py なども参照可能です。

注釈: Fabric はそのコンテンツにアクセスするために fabfile の通常の import (実際は __import__) を行います。 **評価** もしくはそれに類似する動作は行いません。そのため、Fabric は一時的に見つけた fabfile を含むフォルダーを Python の読み込みパスに追加します (そして後ですぐに取り除きます)。

バージョン 0.9.2 で変更: fabfile のパッケージを読み込む機能。

2.4.2 Fabric のインポート

Fabric はただの Python なので、そのコンポーネントを好きなようにインポート **可能** です。とは言え、カプセル化と利便性のため (そして Fabric のパッケージスクリプトの仕事をやりやすくするため)、 fabric.api モジュール内に Fabric のパブリック API がメンテナンスされています。

Fabric の **オペレーション**、 **コンテキストマネージャー**、 **デコレーター**、 **ユーティリティ** のすべてが、単一でフラットな名前空間としてこのモジュールに含まれています。これにより fabfile 内の Fabric に対するとても単純で一貫性のあるインターフェイスが可能になっています:

```
from fabric.api import *

# call run(), sudo(), etc etc
```

これは (多くの理由により) 技術的なベストプラクティスではありませんし、Fab API の呼び出しを 2,3 利用するだけならおそらく `from fabric.api import env, run` などのように明示したほうが **いい** でしょう。とは言え、たいていの単純ではない fabfile ではすべてもしくはほとんどの API を利用するでしょうから、スター (アスタリスク) のインポートを使うといいでしょう:

```
from fabric.api import *
```

上記は、以下を読み書きするよりははるかに容易でしょう:

```
from fabric.api import abort, cd, env, get, hide, hosts, local, prompt, \
    put, require, roles, run, runs_once, settings, show, sudo, warn
```

なので、このケースではベストプラクティスよりも実用主義の方がより優位に感じます。

2.4.3 タスクの定義とコールバックのインポート

fabfile を読み込んだ時に Fabric が正確には何をタスクとみなすのかについて、そして他のコードをインポートする良い方法についての重要な情報が **実行モデル** ドキュメンテーションの **タスクの定義** にありますので、参考にしてください。

2.5 リモートプログラムとのやりとり

Fabric の主な操作、 *run* と *sudo* では、ある意味 `ssh` とほとんど同じようにローカルの入力をリモート側に送ることができます。例えば、パスワードのプロンプトを表示するプログラム (例えば、データベースのダンプユーティリティやユーザーパスワードの変更など) は直接やり取りをしているように振る舞います。

とは言え、 `ssh` と同様に、Fabric のこの機能の実装も直感的とは限らないいくつかの制限に影響されます。このドキュメントではこうした問題について詳述します。

注釈: Unix の標準出力や標準エラー出力のパイプとターミナルデバイスに基礎に馴染みのない読者は Unix パイプ や 擬似端末 を参照するとよいでしょう。

2.5.1 標準出力と標準エラー出力の結合

最初に気づく問題は標準出力と標準エラー出力のストリームで、なぜ両者が随時分離したり結合したりするのかということです。

バッファリング

Fabric 0.9.x 以前では、また Python 自身でも、バッファーの出力は原則的に行ごとで、新しい文字が現れるまでテキストはユーザーに表示されませんでした。これは大抵の場合は問題ありませんが、プロンプトのように一部分の行の出力を扱う必要がある場合には問題となります。

注釈: 行でバッファされた出力はプログラムを理由なく中断させたりフリーズさせたりするように見えます。プロンプトは新しい行なしでテキストを表示し、ユーザーが入力してリターンを押すのを待ちます。

新しい Fabric のバージョンは入力と出力の両方ともを文字ごとにバッファし、プロンプトとのやりとりを可能にします。これは、 "呪われた" ライブラリを活用する複雑なプログラムとのやりとりを可能にします。さもなければスクリーンを再描画します (`top` を考えてみてください)。

ストリームの交差

残念ながら、標準エラー出力と標準出力へ同時に表示させるということは (多くのプログラムで行われていますが)、この 2 つのストリームが無関係に 1 バイトずつ表示されるという意味であり、文字化けしたり、混ざって表示されたりすることがあります。これはストリームの一方を行でバッファすることで緩和させることはできますが、重要な問題であることには変わりありません。

この問題を解決するため、Fabric では低レベルでこの 2 つをマージする SSH 層の設定を利用し、出力をより自然に表示するようにしています。この設定は Fabric では *combine_stderr* env 変数とキーワード引数として表され、デフォルトでは ‘‘True‘‘ が設定されています。

このデフォルトの設定のおかげで出力が正しく表示されますが、*`run`*/*`sudo`* の返り値に空の `.stderr` 属性という代償を払っていて、すべての出力が標準出力に表示されてしまいます。

逆に、Python レベルでの標準エラー出力のストリームを区別した形で必要とするユーザーやユーザーが目にする出力が文字化けしていても構わないユーザー (あるいは、問題となる標準出力や標準エラー出力をコマンドから隠しているユーザー) は、必要に応じてこの値を `False` にセットすることもできます。

2.5.2 擬似ターミナル

ユーザーに対話式プロンプトを表示するときに考慮すべきもう一つの大きな問題は、ユーザー自身の入力のエコーです。

エコー

一般的なターミナルアプリケーションや本物のテキストターミナル (例えば、GUI なしで Unix システムを使用している時) では tty もしくは pty (pseudo-terminal - 擬似ターミナル) と呼ばれるターミナルデバイスとともにプログラムが表示されます。タイプした文字を見ずにやりとりするのは困難なため、これらでは、タイプされたすべてのテキストが (標準出力経由で) ユーザーに自動的にエコーされます。ターミナルデバイスでもまた、条件付きでエコーを向こうにでき、パスワードのプロンプトを安全にできます。

とは言え、tty や pty をまったく表示させなくてもプログラムは実行可能 (例えば cron のジョブを考えてみてください) で、このプログラムに注ぎ込まれるどんな標準入力のデータもエコーされません。これは人間が周りにいなくても実行されるプログラムにとっては望ましい挙動で、Fabric の以前のデフォルトの操作モードでした。

Fabric のアプローチ

残念ながら、Fabric 経由でコマンドを実行するということは、ユーザーの標準入力をエコーするための pty がないときに Fabric がそれをエコーしなくてはならないということです。これは多くのアプリケーションでは問題ありませんが、パスワードプロンプトにとっては問題であり、安全ではありません。

セキュリティ上の理由と驚かすことは最小にするべきという原則に沿うため (ユーザーが一般的にはターミナルエミュレータで動作しているときと同じような挙動を期待している限りにおいては)、Fabric 1.0 以上ではデフォルトで pty を強制します。pty が有効のとき、Fabric は単にリモート側でエコーを扱うようにするか、もしくは標準入力を隠してまったく何もエコーしないようにします。

注釈: 通常のエコーの挙動を許可することに加えて、ターミナルデバイスにアタッチされた時に別の挙動を示すプログラムは、その挙動を示します。例えば、ターミナル上ではカラーで出力し、バックグラウンドではカラーで出力しないプログラムは、pty はカラーで出力します。`run` もしくは `sudo` の返り値を調べる場合は注意してください!

---

pty の挙動の無効が求められる場合は、 `--no-pty` のコマンドライン引数と *always_use_pty* env 変数が利用可能です。

2.5.3 この 2 つの統合

最後の注意点として、擬似ターミナルの利用は事実上、標準出力と標準エラー出力を統合するということです。これは *combine_stderr* の設定が行うのとまったく同じ方法です。これはターミナルデバイスが当然ながら標準出力と標準エラー出力の両方を同じ場所、ユーザーのディスプレイに送るためで、したがってこれにより両者の分化が不可能になっています

とは言え、Fabric レベルではこの 2 つの設定グループはそれぞれ明確に区別され、様々な方法で統合することができます。デフォルトでは、両方とも `True` にセットされています。他の組み合わせは以下になります。

- `run("cmd", pty=False, combine_stderr=True)`: Fabric はパスワードも含め、すべての標準入力をエコーします。また、`cmd` の挙動が変更される可能性もあります。pty で動作していてパスワードのプロンプトについて心配していない時に `cmd` の挙動が好ましくない場合に便利です。
- `run("cmd", pty=False, combine_stderr=False)`: 両方の設定が `False` の時、Fabric は標準入力をエコーし、pty を発行しません。そしてこれは、もっともシンプルなコマンド以外のすべてで好ましくない挙動をもたらす可能性が高くなります。とは言え、区別された標準エラー出力にアクセスする唯一の方法ですし、そうした場合には便利です。
- `run("cmd", pty=True, combine_stderr=False)`: 有効ですが、`pty=True` はそれでも結果としてはストリームにマージされるため、実際には大した違いはありません。`combine_stderr` では問題があるまれなケースを避けるには便利かもしれません (まだ知られたものはありませんが)。

2.6 ライブラリの利用

Fabric の主なユースケースでは fabfile と *fab* ツール経由で、このドキュメントのほとんどではそのことについて書かれています。とは言え、Fabric の中は `fab` や fabfile をまったく使わなくても簡単に使えるように書かれています。このドキュメントではその方法を紹介します。

fabfile を書くことや実行時に `fab` を使うことと比較した場合に、いくつか念頭に置いてほしいことが有ります。どのように接続されているのかとどのように切断されるかということです。

2.6.1 接続

Fabric が実際にどのようにホストに接続するかについては以前に文書化していますが、いまは *execution docs* 全体のどこかに埋もれてしまっています。特に **接続** セクションに行ってざっと読んでみるといいかもしれません (必須ではありませんが、一度はこのドキュメントの全体をざっと読んだほうがよいでしょう)。

このセクションでも言及しているように、重要なのは *env.host_string* に接続時に *`run`*、*`sudo`* その他の操作がひとつの場所を見に行っているだけだということです。ホストをセットするその他のすべてのメカニズムは実行時に `fab` ツールによって解釈され、ライブラリとして実行される場合は問題にはなりません。

とは言え、与えられたタスク `X` と与えられたホストのリスト `Y` を結合したいという大抵のユースケースでは、Fabric 1.3 の時点で、`execute(X, hosts=Y)` 経由の *`execute`* 関数で扱うことができます。詳細は *`execute`* のドキュメントをご覧ください。手動でのホスト文字列の操作はほとんど必要ないはずです。

2.6.2 接続解除

他に `fab` が行うおもなことは、セッションの終わりにすべてのホストからの接続を解除することです。そうしないと、Python はそれらのネットワークリソースが解放されるまで居座り続けるでしょう。

Fabric 0.9.4 以降ではこれを簡単に行うために利用できる関数、*disconnect_all* が有ります。終了時 (たいていは `try: finally` ステートメントの外の `finally` 節で、何かの拍子に接続解除を妨げるエラーをださないよう) に、単にこれを自分のコードで呼び出すようにします。これでうまく動作するはずです。

Fabric 0.9.3 以上なら、単にこれだけです (`disconnect_all` はこのロジックにちょっと整った出力を加えているだけです):

```
from fabric.state import connections

for key in connections.keys():
    connections[key].close()
    del connections[key]
```

2.6.3 最後に

このドキュメントは初期の草稿段階で、 `fab` の利用とライブラリの利用との違いをすべてカバーしてはいません。とは言え、上記はもっともつまずきやすい箇所にハイライトを当てています。疑問に思った時は Fabric のソースコード内のメモに `fab` によって実行される追加の動作の大半が含まれていますので参照してください。

2.7 出力の管理

`fab` ツールはデフォルトで非常に冗長です。リモート側の標準出力、標準エラー出力、実行されたコマンドなど可能な限りほぼすべてを出力します。 何が起こったか知るために多くの場合これが必要になりますが実行中の Fabric のタスクを追いかけるのは難しくなります。

2.7.1 出力レベル

タスクの出力を整理するために Fabric の出力はいくつかの重複しないレベルやグループに分けられます。そしてそれぞれ独立してオン、オフできます。これでユーザーに表示される内容を柔軟にコントロールします。

注釈: `debug` を除いてすべてのレベルでデフォルトはオンです。

標準出力レベル

標準のアトミック出力レベル/グループは次の通りです:

- **status**: 状況メッセージ、例えば Fabric の実行後にはなし、ユーザーがキーボードインターラプトを使ったか、サーバーが切断されたか、などです。これらのメッセージはほぼ適切でめったに冗長でありません。

- **aborts**: アボート (中止) メッセージ。状況メッセージのように Fabric をライブラリとして使用するときのみオフにすべきです。本当はすべきでありませんが。注意：この出力グループをオフにしてアボートが発生したら – なぜ Fabric がアボートしたか、どんな出力もありません!
- **warnings**: 警告メッセージ。ファイルの中のテキストを `grep` するときのような失敗が予想される動作ではしばしばオフにされます。 `env.warn_only` の設定を True で一緒に使えばリモートプログラムが失敗した時の警告が完全に沈黙します。 `aborts` と同様にこの設定は実際の警告の動作をコントロールするわけではありません。ただ警告を表示するかしないかだけです。
- **running**: 実行したコマンドや転送したファイルの出力。例えば `[myserver] run: ls /var/www`。実行したタスクの出力もコントロールします。例えば `[myserver] Executing task 'foo'`。
- **stdout**: ローカルやリモートの標準出力、例えばコマンドからのエラーでない出力。
- **stderr**: ローカルやリモートのエラー出力、例えばコマンドからのエラーに関する出力。
- **user**: ユーザーが生成する出力、例えば *`fastprint`* や *`puts`* 関数を使った fabric コードで出力されるローカル出力。

バージョン 0.9.2 で変更: "Executing task"を `running` 出力レベルの追加。

バージョン 0.9.2 で変更: `user` 出力レベルを追加。

デバッグ出力

最後のアトミック出力レベル、 `debug` これは他とはやや異なる動作をします:

- **debug**: 現在デバッグをオンは (デフォルトでオフ) 主に実行中の "完全な" コマンドを調べるのに使います。例えばこの *`run`* 呼び出し:

```
run('ls "/home/username/Folder Name With Spaces/"')
```

通常 `running` 行は *`run`* に何が渡されるか正確に表示します。このように like so:

```
[hostname] run: ls "/home/username/Folder Name With Spaces/"
```

`debug` がオンで *`shell`* が `True` ならリモートサーバーに渡される文字通りすべての文字列が表示されます:

```
[hostname] run: /bin/bash -l -c "ls \"/home/username/Folder Name With Spaces\""
```

`debug` 出力が有効ならアボート中のすべての Python トレースバックを表示します。

注釈: 出力の部分的変更 (上の例では 'running' 行にシェルとエスケープ文字を表示する変更) は他の設定に優先します。もし `running` が False で `debug` が True ならデバッグフォームで 'running' 行が表示されます。

バージョン 1.0 で変更: 現在のデバッグ出力は、アボート中の完全な Phython トレースバックを含みます。

出力レベルのエイリアス

上記の アトミック/スタンドアロン レベルに加えて Fabric は複数のレベルを組み合わせたいくつかの便利なエイリアスを提供します。組み合わされているすべてのレベルを効果的に切り替えることができます。

- **output**: `stdout` と `stderr` の両方。これは 'running' 行と出力状況に注目するときに便利です (それと警告)。
- **everything**: `warnings`, `running`, `user` そして `output` を含む (上記)。`everything` をオフにすると自分自身の出力と最小の出力 (もしオンなら `status` と `debug`) になります。
- **commands**: `stdout` と `running` を含む。標準エラー出力を表示してエラーでない出力全体を隠すのに便利です。

バージョン 1.4 で変更: `commands` 出力エイリアスを追加。

2.7.2 出力レベルの表示/非表示

Fabric の出力レベルの切り替え方法はいくつかあります; それぞれの箇条書きの API ドキュメントへのリンクを参照してください:

- **直接 fabric.state.output を編集する**: `fabric.state.output` はディクショナリーサブクラスで (*env* のような) key は出力レベル名、そして value は True (指定したタイプの出力を表示) か False (隠す) のどちらかです。

  `fabric.state.output` これは出力レベルの最も低いレベルの実装で Fabric が出力を表示するかどうかの内部参照です。

- **コンテキストマネージャー**: *`hide`* と *`show`* は文字列としてひとつ以上の出力レベル名を持つコンテキストマネージャーです。 hide か show どちらかのラップしたブロックがあります。Fabric の他のコンテキストマネージャーのように、ブロックが存在すれば以前の値が保持されています。

  参考:

  `settings` は内部で *`hide`* および/もしくは *`show`* への呼び出しをネストできます。

- **コマンドライン引数**: *`--hide`* と *`--show`* が使えます。 *fab* **オプションと引数** これは同じ名前のコンテキストマネージャー (しかし当然グローバルに適用されます) と同じ動作で入力にコンマで区切られた文字列を使います。

2.8 並列実行

バージョン 1.3 で追加.

Fabric はデフォルトですべてのタスクを **連続** で実行します。(参照 **実行ストラテジー**) このドキュメントは per-task デコレーター、コマンドラインスイッチを使って複数のホストでタスクを **並行** で実行する Fabric のオプションを説明します。

2.8.1 どうなるか

Fabric 1.x は完全なスレッドセーフではありません。(一般的な使い方ではタスク関数は他に影響しません) この機能は Python の multiprocessing モジュールで実装されています。これはそれぞれのホストとタスクコンビネーションで新しいプロセスを作り同時に多くのプロセスが走るのを避けるために任意 (設定可能) のスライディングウィンドウを使います。

例えばいくつかのウェブサーバーでウェブアプリケーションのコードを更新してコードが配信されたら (コードの更新が失敗したら簡単にロールバックできる) サーバーをリロードする状況を想像してください。次の Fabfile で実行できます:

```
from fabric.api import *

def update():
    with cd("/srv/django/myapp"):
        run("git pull")

def reload():
    sudo("service apache2 reload")
```

そして 3 つのサーバーで連続して実行するには:

```
$ fab -H web1,web2,web3 update reload
```

並列実行のオプションを有効にしなければ通常 Fabric はこのように実行します:

1. `update` on `web1`
2. `update` on `web2`
3. `update` on `web3`
4. `reload` on `web1`
5. `reload` on `web2`
6. `reload` on `web3`

並列実行を有効 (*-P* – 下記参照) にした場合はこのように:

1. `update` on `web1`, `web2`, and `web3`
2. `reload` on `web1`, `web2`, and `web3`

うまくいけば利益は明らかです – もし `update` に 5 秒かかって `reload` に 2 秒かかるとすると連続タスクでは (5+2)*3 = 21 秒。しかし並列実行では 1/3 平均で (5+2) = 7 秒。

2.8.2 使い方

デコレーター

並列実行の最小 "単位" はタスクなのでこの機能はタスク単位で有効、無効を指定できます。`parallel` と `serial` デコレーターを使います。例えばこの fabfile:

```
from fabric.api import *

@parallel
def runs_in_parallel():
    pass

def runs_serially():
    pass
```

この方法で実行すると:

```
$ fab -H host1,host2,host3 runs_in_parallel runs_serially
```

次のシーケンスで実行します:

1. `runs_in_parallel` on `host1`, `host2`, and `host3`
2. `runs_serially` on `host1`
3. `runs_serially` on `host2`
4. `runs_serially` on `host3`

コマンドライン・フラグ

コマンドラインフラグ *-P* や環境変数 *env.parallel* を使ってすべてのタスクに並行処理を強制することもできます。しかし、どんなタスクも `serial` を指定すると設定を無視してシリアルで実行します。

例えば、次の fabfile は上記の実行シーケンスと同じです:

```
from fabric.api import *
```

(次のページに続く)

(前のページからの続き)

```
def runs_in_parallel():
    pass

@serial
def runs_serially():
    pass
```

このようにすると:

```
$ fab -H host1,host2,host3 -P runs_in_parallel runs_serially
```

前述のように `runs_in_parallel` はパラレルで `runs_serially` はシーケンスで実行されます。

2.8.3 バブルサイズ

巨大なホストリストではローカルマシンが同時に実行されるあまりの多くの Fabric プロセスのために高負荷になります。このため同時に走る Fabric のアクティブプロセスの数を制限するムービング・バブルアプローチを選ぶこともできます。

デフォルトではバブルは使わずすべてのホストでひとつの pool で実行します。この per-task レベルは `parallel` にキーワード引数 `pool_size` を与えることや全体的には `-z` でオーバーライドできます。

例えば、同時に 5 つのホストで走らせるには:

```
from fabric.api import *

@parallel(pool_size=5)
def heavy_task():
    # lots of heavy local lifting or lots of IO here
```

あるいは kwarg `pool_size` をスキップして:

```
$ fab -P -z 5 heavy_task
```

2.8.4 出力 行単位 対 バイト単位 (Linewise vs bytewise)

Fabric のデフォルトの端末への出力モードはバイト単位です。これは **リモートプログラムとのやりとり** のサポートのためです。これはパラレルモードでは悲惨なことになります。複数のプロセスが端末の標準出力に同時に書き込むかもしれません。

これを避けるためにパラレルが有効なときには自動的に Fabric の linewise output オプションが有効になります。これは上記リンクの Fabric のリモートインターラクティブ機能の便利さの大部分を失うことになりますが、パラレル呼び出しではうまく配置できません。一般的にはフェアな取引きです。

行単位の出力でも複数のプロセスの混乱を避けることはできません。しかしプレフィックス the host-string で少なくとも見分けることができます。

注釈: 将来のバージョンでは並行処理のトラブルシューティングを簡単にするためにロギングサポートが改良されるでしょう。

2.9 SSH の動作

Fabric では、接続の管理にピュア Python の SSH 再実装を利用しています。これはつまり、このライブラリーの機能に制限される箇所があるということを意味します。以下は `ssh` コマンドラインプログラムの動作とは同等ではないか、それほどには柔軟性がない、Fabric が示す動作の領域です。

2.9.1 未知のホスト

SSH のホストキートラッキングメカニズムは、接続を試行したすべてのホストを把握し、識別子 (IP アドレスや、ときにはホスト名も) と SSH キーをマッピングして `~/.ssh/known_hosts` で維持管理します。(この動作の詳細は OpenSSH documentation を御覧ください。)

`paramiko` ライブラリは `known_hosts` ファイルを読み込むことができ、このマッピングとともに、接続するすべてのホストを比較します。未知のホスト (ユーザー名や IP が `known_hosts` に見当たらないホスト名) に遭遇したときにどのように動作させるかを決定する設定があります:

- **Reject(拒否)**: そのホストキーが拒否され、接続は行われません。結果として Python の例外を発生させ、未知のホストであるというメッセージとともに Fabric のセッションを終了させます。
- **Add(追加)**: 新しいホストキーが既知のホストのインメモリーリストに追加され、接続が行われ、通常通り動作が継続されます。これによってあなたのディスク上の `known_hosts` ファイルが変更されることは **ありません**！
- **Ask(尋ねる)**: Fabric のレベルではまだ実装されていません。これは `paramiko` ライブラリのオプションで、未知のキーについてそれを受け入れるかどうかユーザーに尋ねます。

上記のホストの拒否もしくは追加は Fabric の *env.reject_unknown_hosts* オプション経由でコントロールされ、利便性のため、デフォルトでは False に設定されています。これは利便性とセキュリティとの間の妥当なトレードオフだと私たちは感じています。そのようには感じない方は簡単にモジュールレベルで fabfile を変更し、`env.reject_unknown_hosts = True` にセットすることができます。

2.9.2 変更されたキーの既知のホスト

SSH のキー/フィンガープリントのポイントは中間者攻撃を検知できることです。もしあなたの SSH トラフィックを攻撃者が自分のコントロール下にあるコンピュータにリダイレクトさせ、本物の目的サーバに見せかけようとしても、ホストキーがマッチしません。そして、SSH(とその Python 実装) のデフォルトの挙動では、`known_hosts` に以前に記録されているホストが突然違ったホストキーの送信を始めると、その接続は直ちに切断されます。

EC2 でのデプロイメントなどのいくつかのエッジケースでは、この潜在的な問題を無視するとよいでしょう。私たちの SSH 層では、これを書いている時点では、この挙動を正確にコントロールすることはできません。ただし、`known_hosts` の読み込みを単にスキップすることでこれを回避することができます。比較するこのホストリストが空なら問題は発生しない、ということです。この挙動にするには *env.disable_known_hosts* を True にセットします。デフォルトでは、SSH のデフォルト挙動を維持するため、False になっています。

警告: *env.disable_known_hosts* を有効にすると中間者攻撃に対して無防備になります! 注意して利用してください。

2.10 タスクの定義

Fabric 1.1 からは、fabfile の中でどのオブジェクトをタスクとして示すかを定義するために利用できる、2 つのはっきりと異なった方法があります:

- 1.1 からスタートした "新しい" 方法は *`Task`* もしくはそのサブクラスのインスタンスを考慮し、また、ネストされた名前空間を構築できるようにするためにインポートされたモジュールに落とし込まれています。
- "クラシックな" 方法は 1.0 以前からのもので、すべてのパブリックな呼び出し可能なオブジェクト (関数やクラスなど) を考慮し、インポートされたモジュールへの再帰的には処理せず、その fabfile のオブジェクトのみ考慮します。

注釈: これらの 2 つの方法は **相互に排他的です** : もし Fabric が fabfile 内やインポートしたモジュール内に **どんな** 新しいスタイルのタスクオブジェクトでも見つければ、タスク宣言のこのメッソドをコミットしたとしてみなされ、*`Task`* 以外の呼び出しは考慮されません。新しいスタイルのタスクが **なければ** クラシックな挙動に戻ります。

このドキュメントではこの 2 つのメソッドを詳細に説明します。

注釈: fabfile でどのタスクが `fab` 経由で実行されうるかを厳密に確認するには *`fab --list`* を使います。

2.10.1 新しいスタイルのタスク

Fabric 1.1 では新しい機能を促進し、プログラミングのベストプラクティスを可能にするために *Task* クラスを導入しました。特に:

- **オブジェクト指向のタスク**。継承とそれに伴うすべては、単純に関数オブジェクトを渡し回るより、より実用的なコードの最利用を可能のします。タスク宣言のクラシックなスタイルは完全に排除されているわけではありませんでしたが、ことをとても簡単にするわけでもありませんでした。
- **名前空間**。タスクの宣言を明確なメソッドにすることによって、例えば、(クラシックな手順のもとで有効な "タスク" として表示される)Python `os` モジュールのコンテンツでタスクリストを汚すことなく、再帰的な名前空間のセットアップが容易になります。

Task の前置きとして、新しいタスクをセットアップするには 2 つの方法があります:

- `@task` で通常のモジュールレベルの関数をデコレートします。これは *Task* サブクラス内の関数を透過的にラップします。実行時には関数名がタスク名として使われます。
- *Task* サブクラス (*Task* 自身は抽象的であることを意図されています) は `run` メソッドを定義し、モジュールレベルであなたのサブクラスをインスタンス化します。インスタンスの `name` 属性がタスク名として使われます。省略した場合には、その代わりにインスタンスの変数名が使われます。

新しいスタイルのタスクの利用はまた、 **名前空間** のセットアップも可能にします。

@task デコレータ

新しいスタイルのタスク機能を利用するもっとも簡単な方法は基本のタスク関数を `@task`: で囲ってしまう方法です:

```
from fabric.api import task, run

@task
def mytask():
    run("a command")
```

このデコレータが使われると、このデコレータで囲われた関数 ***のみ*** が有効なタスクとして読み込まれることを Fabric に伝えます。(表示されない場合は、 **クラシックスタイルの** 挙動で動作しています)

引数

`@task` はまた、引数とともに呼び出してその挙動をカスタマイズすることもできます。以下に記述されていない引数は利用中の `task_class` のコンストラクタにその最初の関数自身を最初の引数として渡されます。(詳細は *@task* **とのカスタムサブクラスの使用** をご覧ください)

- task_class: *Task* のサブクラスで、デコレートされた関数をラップするために使用されます。デフォルトは *WrappedCallableTask* です。
- aliases: ラップされた関数用のエイリアスとして繰り返し使える文字列名。詳細は **エイリアス** をご覧ください。
- alias: aliases と似ていますが、繰り返しできない単一の文字列引数を取ります。もし alias と aliases の両方が設定されている場合、 aliases の方が優先されます。
- default: デコレートされたタスクが、タスク名としてそれが含むモジュールの代役を務めるかどうかを決めるための真偽値。**デフォルトのタスク** を参照してください。
- name: そのタスクがコマンドラインインターフェースに表示されるときの名称を設定する文字列です。Python のビルトインを別の方法でシャドーイングするタスク名で便利です (これは技術的には可能ですが、好まれませんし、バグの温床にもなります)。

エイリアス

以下は、人間が読める長めのタスク名とすばやくタイプするための短めのタスク名の両方の利用を手助けするための alias キーワード引数の簡単な利用例です:

```
from fabric.api import task

@task(alias='dwm')
def deploy_with_migrations():
    pass
```

この fabfile で *--list* を呼び出すとオリジナルの ‘‘deploy_with_migrations‘‘ とエイリアスの dwm: を表示します:

```
$ fab --list
Available commands:

    deploy_with_migrations
    dwm
```

同じ関数に複数のエイリアスが必要な場合は、単に aliases キーワード引数をスワップします。これにより、単一の文字列の代わりに繰り返し利用可能な文字列が取られます。

デフォルトのタスク

aliases と同じような方法で、モジュール内の与えられたタスクを "default" タスクとして指定するときに便利な場合があり、そのモジュール名を **単に** 言及することで呼び出すこともできます。これによりタイピングが省略できたり、一つの "メイン" タスクとたくさんの関連タスクもしくはサブモジュールがある場合のより整理された構成が可能になります。

例えば、 deploy サブモジュールが新しいサーバのプロビジョニング、コードのプッシュ、データベースの移行などのタスクを含んでいるとして、デフォルトの "just deploy" アクションとしてタスクを強調できるととても便利でしょう。そうした deploy.py モジュールは次のようになります:

```
from fabric.api import task

@task
def migrate():
    pass

@task
def push():
    pass

@task
def provision():
    pass

@task
def full_deploy():
    if not provisioned:
        provision()
    push()
    migrate()
```

タスクリストは以下のようになります (単に deploy を読み込んでいるだけの簡単なトップレベルの fabfile.py であると仮定します):

```
$ fab --list
Available commands:

    deploy.full_deploy
    deploy.migrate
    deploy.provision
    deploy.push
```

デプロイのたびに deploy.full_deploy を呼び出すのはちょっと古めかしいですし、チームに加わった新らしい方にとってはこれが実行する正しいタスクなのか迷うことでしょう。

@task への default キーワード引数を利用することにより、例えば、デフォルトのタスクとして full_deploy をタグ付けすることができます:

```
@task(default=True)
def full_deploy():
    pass
```

このようにアップデートするとタスクリストは以下のようになります:

```
$ fab --list
Available commands:

    deploy
    deploy.full_deploy
    deploy.migrate
    deploy.provision
    deploy.push
```

full_deploy は明示的なタスクとしてそのままあることに留意してください。そして、 full_deploy のある種トップレベルのエイリアスとして deploy が表示されます。

もし一つのモジュール内に複数の default=True がセットされている場合は、最後に読み込まれたもの (通常はファイルの最も下にあるもの) が優先されます。

トップレベルのデフォルトタスク

トップレベルの fabfile で @task(default=True) を使用すると、ユーザーがタスク名なしで fab を呼び出した際にそのタスクを実行します (例えば、 make に似ています)。このショートカットの使用時にはタスク自身に引数を指定することはできません。引数が必要な場合は通常のタスク呼び出しを実行してください。

Task サブクラス

クラシックスタイルのタスク に慣れている場合は、 *Task* はその run メソッドがクラシックなタスクとそのまま同等であると考えると分かりやすいでしょう。その引数が (self 以外の) タスクの引数で、そのボディが実行される内容となります。

例えば、この新しいスタイルのタスクは:

```
class MyTask(Task):
    name = "deploy"
    def run(self, environment, domain="whatever.com"):
        run("git clone foo")
        sudo("service apache2 restart")

instance = MyTask()
```

この関数ベースのタスクとまったく同等です:

```
@task
def deploy(environment, domain="whatever.com"):
    run("git clone foo")
    sudo("service apache2 restart")
```

クラスのインスタンスをどのように作成しているかに留意してください。これは実際に動作する単純な通常の

Python オブジェクト指向プログラミングです。この時点では小さなひな形に過ぎませんが、例えば、Fabric はインスタンス作成時に与える名前は気にせず、そのインスタンスの **名前** 属性のみを気にします。クラスの力 (ちから) を利用可能にすることの恩恵は検討に値するでしょう。

将来的にはこの API を拡張して、このエクスペリエンスをより洗練させていく予定です。

`@task` とのカスタムサブクラスの使用

カスタムな *`Task`* サブクラスと `@task` を結合させることも可能です。これはコアの実行ロジックがクラス/オブジェクト指向ではないけれどもクラスのメタプログラミングやそれと似たテクニックを活用したい場合に有用です。

特に、呼び出し可能なものとして最初のコンストラクタ引数を取るように設計されているすべての *`Task`* サブクラスは (ビルトインの *`WrappedCallableTask`* と同様に)、 `@task` への `task_class` 引数として指定することができます。

Fabric は与えられたクラスのコピーを自動的にインスタンス化し、最初の引数としてラップされた関数に渡されます。デコレーターに渡されるすべての他の引数/キーワード引数 (**引数** に記述されている "特別な" 引数以外) はその後に追加されます。

これを明確にするための簡単でいくらか工夫されている例をお見せします:

```
from fabric.api import task
from fabric.tasks import Task

class CustomTask(Task):
    def __init__(self, func, myarg, *args, **kwargs):
        super(CustomTask, self).__init__(*args, **kwargs)
        self.func = func
        self.myarg = myarg

    def run(self, *args, **kwargs):
        return self.func(*args, **kwargs)

@task(task_class=CustomTask, myarg='value', alias='at')
def actual_task():
    pass
```

この fabfile が読み込まれた時、`CustomTask` のコピーがインスタンス化され、事実上は以下を呼び出しています:

```
task_obj = CustomTask(actual_task, myarg='value')
```

`alias` キーワード引数がデコレーター自身によってどのように取り除かれるか、そしてクラスのインスタンス化には到達しないことに留意してください。これは **コマンドラインタスクの引数** の動作と機能的に同一です。

名前空間

クラシックなタスク では、複数の fabfile は単一でフラットなタスク名のセットに制限され、それらを体系化する本格的な方法はありません。Fabric 1.1 以降では、タスクを新しいやり方 (`@task` もしくは ご自分の *`Task`* サブクラスのインスタンス経由) で宣言すれば **名前空間** を活用することができます:

- fabfile にインポートされたどのモジュールオブジェクトも再帰的に処理され、追加のタスクオブジェクトを探します。
- サブモジュール内では、Python 標準の `__all__` モジュールレベル変数名を使ってどのオブジェクトが "exported" されるかをコントロールすることができます (有効な新しいタスクオブジェクトでなければなりませんが)。
- これらのタスクは、Python 自身のインポートシンタックスと似た、それを含んだモジュールをベースにした新しいドットノーテーション名が与えられます。

単純な複雑なものまで fabfile のパッケージを組み立てて、どのように動作するか見てみましょう。

基本

まずはいくつかのタスクを含む一つの `__init__.py` (簡潔性のため Fabric の API は省略します) から始めてみましょう:

```
@task
def deploy():
    ...

@task
def compress():
    ...
```

`fab --list` の出力は次のようになるでしょう:

```
deploy
compress
```

ここでは名前空間は一つだけで、 "root" もしくはグローバルな名前空間です。今は単純に見えますが、実世界の fabfile ではたくさんのタスクがあり、管理が難しくなり得ます。

サブモジュールのインポート

前述のとおり、モジュールが Python のインポートパス上のどこにあるかには関係なく、Fabric はインポートされたモジュールオブジェクトのタスクを調べます。今のところは、とりあえず "手近にある" 自前のタスクを含めたいと思いますので、そうですね、ロードバランサを扱うためのパッケージに新しいサブモジュール `lb.py` を作ってみましょう:

```
@task
def add_backend():
    ...
```

そして __init__.py の一番上にこれを追加します:

```
import lb
```

さて、これで fab --list は次のようになります:

```
deploy
compress
lb.add_backend
```

モジュールに一つだけのタスクではある種他愛のないもののように見えますが、その恩恵はかなり明白だと思います。

さらに奥深くへ

名前空間化は単にひとつのレベルに制限されることはありません。より大きなセットアップを持ち、データベース関連のタスクのための名前空間が必要になっていて、その中に追加の別のタスクがあるとしましょう。 db/ と名前を付けられたサブパッケージを作り、その中に migrations.py モジュールを起きます:

```
@task
def list():
    ...

@task
def run():
    ...
```

このモジュールは db をインポートしているすべてから見えるようにする必要があるので、このサブパッケージの __init__.py に以下を追加します:

```
import migrations
```

最後のステップとして、ルートレベルの __init__.py にサブパッケージをインポートします。これで最初の何行かは以下のようになります:

```
import lb
import db
```

そして、ファイルツリーは以下のようになります:

```
.
├── __init__.py
├── db
│ ^^c2^^a0^^c2^^a0 ├── __init__.py
│ ^^c2^^a0^^c2^^a0 └── migrations.py
└── lb.py
```

`fab --list` は次のようになります:

```
deploy
compress
lb.add_backend
db.migrations.list
db.migrations.run
```

また、タスクを `db/__init__.py` 内に直接設定 (もしくはインポート) することも可能で、ご想像どおり `db.<` **なんちゃら**> として表示されます。

`__all__` での制限

インポートされたモジュールを Fabric が分析するときに、モジュールレベル `__all__` 変数 (変数名のリスト) の Python の決まり事を利用することによって、Fabric が "見る" ものを制限することができます。もし何かの理由によりデフォルトでは `db.migrations.run` タスクを表示させたくない場合、 `db/migrations.py` の一番上に以下を追加することができます:

```
__all__ = ['list']
```

ここには `'run'` がないことに留意してください。もし必要なら `run` をこの階層のどこかに直接インポートすることも可能ですが、そうでなければ、隠れれたままになります。

ひとつ上へ

これまで、fabfile パッケージを片付いた状態に維持し、直接的な方法でインポートしてきましたが、ファイルシステムのレイアウトはここでは実際には考慮していません。Fabric のすべてのローダーが気にするのは、インポートされた時のモジュールに与えられた名称です。

例えば、ルートの `__init__.py` の市場うえを次のように変更すると:

```
import db as database
```

タスクリストは次のように変わります:

```
deploy
compress
```

(次のページに続く)

(前のページからの続き)

```
lb.add_backend
database.migrations.list
database.migrations.run
```

これは他のどのインポートにも適用されます。サードパーティのモジュールを自分のタスク階層にインポートしたり、深くネストされたモジュールを取ってきてトップレベル近くに置くことも可能です。

ネストされたリスト出力

最後に、このセクションではデフォルトの Fabric *--list* 出力を使用してきました。これにより実際のタスク名は何なのかがより明確になります。とは言え、*--list-format* オプションに `nested` を渡すと、よりネストされていたりツリーライクな表示を得られます:

```
$ fab --list-format=nested --list
Available commands (remember to call as module.[...].task):

    deploy
    compress
    lb:
        add_backend
    database:
        migrations:
            list
            run
```

"実際の" タスク名が少し分かりにくくなりますが、この表示は大規模な名前空間でのタスク構成を把握する簡単な方法を提供します。

2.10.2 クラシックなタスク

新しいスタイルの *Task* ベースのタスクが見つからない場合、Fabric は fabfile 内の呼び出し可能などんなオブジェクトでも検討します。 **ただし、** 次を除きます:

- アンダースコア (_) で始まる名称の呼び出し可能なオブジェクト。言い換えると、ここでは Python の通常の "プライベート" 規則が適用されます。
- Fabric 自身内で定義されている呼び出し可能なオブジェクト。 *run* と *sudo* などの Fabric 自身の関数はタスクリストには表示されません。

インポート

Python の `import` ステートメントは事実上、あなたのモジュール名前空間にあるインポートされたオブジェクトを含みます。Fabric の fabfile は単なる Python モジュールなので、インポートもまた、fabfile 自身に定義されてい

るすべてと一緒に出来る限りクラシックスタイルのタスクとみなされます。

> **注釈:** これはインポートされた **呼び出し可能なオブジェクト** のみに適用され、モジュールには適用されません。インポートされたモジュールは **新しいスタイルのタスク** を含む場合のみ実行され、その時点でこのセクションは適用されません。

このため、私達としては後ろに `module.callable()` が続くインポートの `import module` 形式を利用するよう強くおすすめします。これにより、結果として `from module import callable` を行うよりもよりきれいな fabfile API になります。

ウェブサービスから何らかのデータを取り出すために `urllib.urlopen` を使用しているサンプルの fabfile を例に上げましょう:

```
from urllib import urlopen

from fabric.api import run

def webservice_read():
    objects = urlopen('http://my/web/service/?foo=bar').read().split()
    print(objects)
```

これはかなり単純でエラー無しで実行できます。しかし、この fabfile で *`fab --list`* を実行するとどうなるか見てみましょう:

```
$ fab --list
Available commands:

  webservice_read   List some directories.
  urlopen           urlopen(url [, data]) -> open file-like object
```

1 つのタスクしかない fabfile で 2 つの "タスク" が表示されています。これはよくありませんし、疑うことを知らないユーザーがうっかりと `fab urlopen` を呼びだそうとするかもしれませんし、たぶんまともには動作しないでしょう。実世界の fabfile を想像してみてください。かなり複雑になることが多いでしょうし、すぐに乱雑になってしまうことがわかってもらえると思います。

参考のため、以下がおすすめの方法になります:

```
import urllib

from fabric.api import run

def webservice_read():
    objects = urllib.urlopen('http://my/web/service/?foo=bar').read().split()
    print(objects)
```

簡単な変更ですが、この fabfile を使う方は誰でもこれにより少しは幸せになるでしょう。

第 3 章

API ドキュメント

Fabric は 2 つの API ドキュメントをメンテナンスしています。これはソースコードの docstrings から自動生成されるもので、通常はとても詳細なものです。

3.1 コア API

コア API は、Fabric の基礎的な構成要素 (*run* や *sudo* など) を形成する関数、クラス、メソッドとしておおまかに定義されています。そしてこの上に他のすべて (以下の "contrib" セクションやユーザーの fabfile) が作られています。

3.1.1 色出力関数

バージョン 0.9.2 で追加.

ANSI 色コードで文字列をラッピングするための関数です。

このモジュール内の各関数は、適切な色のための ANSI 色コードでラップされた入力文字列 **テキスト** を返します。

例えば、サポートしているターミナルでテキストを緑で表示するには:

```
from fabric.colors import green

print(green("This text is green!"))
```

これらの関数は単純に変更された文字列を返すだけなので、これらをネストすることもできます:

```
from fabric.colors import red, green

print(red("This sentence is red, except for " +           green("these words, which␣
↪are green") + "."))
```

bold が True にセットされていると、ボールドにするための ANSI フラグが特定の実行でオンになり、通常はたいていのターミナルではボールドとして、もしくは元の色よりも明るく表示されます。

fabric.colors.**blue**(*text*, *bold=False*)

fabric.colors.**cyan**(*text*, *bold=False*)

fabric.colors.**green**(*text*, *bold=False*)

fabric.colors.**magenta**(*text*, *bold=False*)

fabric.colors.**red**(*text*, *bold=False*)

fabric.colors.**white**(*text*, *bold=False*)

fabric.colors.**yellow**(*text*, *bold=False*)

3.1.2 コンテキストマネージャー

コンテキストマネージャーは with とともに使用します。

注釈: Python 2.5 の利用時、 with ステートメントを利用するためには fabfile を from __future__ import with_statement から始める必要があります (Python 2.6 以上では通常の非 non __future__ 機能です)。

注釈: ダイレクトにネストされている複数の with ステートメントを使用する場合、1 つの with ステートメント内に複数のコンテキスト表現を用いたほうが便利なことがあります。次のように書く代わりに:

```
with cd('/path/to/app'):
    with prefix('workon myvenv'):
        run('./manage.py syncdb')
        run('./manage.py loaddata myfixture')
```

次のように書くことができます:

```
with cd('/path/to/app'), prefix('workon myvenv'):
    run('./manage.py syncdb')
    run('./manage.py loaddata myfixture')
```

このようにするには Python 2.7 以上が必要です。Python 2.5 もしくは 2.6 では次のようにします:

```
from contextlib import nested

with nested(cd('/path/to/app'), prefix('workon myvenv')):
    ...
```

最後に *settings* は nested 自身を実装していることに留意してください。詳しくはこの API ドキュメントをご覧ください。

---

fabric.context_managers.**cd**(*path*)

コンテキストマネージャーはリモート操作を呼び出すときにはディレクトリの状態を維持します。

これでラップされたブロック内の run、 sudo、 get、 put へのどんな呼び出しも、実際に状態が維持されているという意味を持たせるために "cd <path> && " がプリフィックスされたのと似た文字列を暗に含みます。

注釈: *cd* は **リモート** のパスでのみ作用します。 **ローカル** パスを変更するには *lcd* を使います。

---

cd の利用はこうしたすべての実行に影響を与えるので、contrib セクションの多くなど、うした操作を行うコードもまた *cd* の利用に影響を受けます。

実際のシェルビルトインの *cd* のように、 *cd* は相対パスで呼び出すことができ (最初のディレクトリのデフォルトはリモートユーザーの $HOME であることに気をつけてください)、ネストすることも可能です。

下の例はシェルの *cd* を使った "ノーマルな" 試みで、 run もしくは sudo の実行と実行の間の状態が維持**されない** というシェルなし SSH 接続の実装のため動作しません。

```
run('cd /var/www')
run('ls')
```

上の例は /var/www ではなく、リモートユーザーの $HOME のコンテンツの一覧を表示します。*cd* とともに使うと期待通りに動作します:

```
with cd('/var/www'):
    run('ls') # Turns into "cd /var/www && ls"
```

最後に、ネストの例 (インラインのコメントをご覧ください) をお見せします:

```
with cd('/var/www'):
    run('ls') # cd /var/www && ls
    with cd('website1'):
        run('ls') # cd /var/www/website1 && ls
```

注釈: このコンテキストマネージャーは環境変数のカレントの値、 env.cwd に追加することによって実装されています (そして、いつものようにその後もとの状態にもどします)。とは言え、この実装は将来的には変更されるかもしれませんので、手動で env.cwd に変更することはおすすめしません。後方互換性を保証するのは *cd* の **挙動** のみです。

注釈: 空白文字を含むディレクトリ名を扱うため、空白文字は自動的にエスケープされます。

バージョン 1.0 で変更: コマンドでの操作に加え、 `get` と `put` にも適用されます。

参考:

lcd

fabric.context_managers.**char_buffered**(**args*, ***kwds*)

ローカルターミナルの **パイプ** を行ごとではなく文字ごとに強制的にバッファします。

ユニックスベースのシステムのみの適用されます。Windows では操作できません。

fabric.context_managers.**hide**(**args*, ***kwds*)

与えられた出力 `groups` を False にセットするためにコンテキストマネージャー。

`groups` は `output` で定義された出力グループを命名している 1 つ以上の文字列でなければなりません。与えられたグループは、囲まれたブロックでの継続性のため False にセットされ、その後は前の状態に戻ります。

例えば、 "[hostname] run:" のステータスラインを隠し、同じように標準出力と標準エラー出力を表示させないようにするには、次のように *hide* を利用することができます:

```
def my_task():
    with hide('running', 'stdout', 'stderr'):
        run('ls /var/www')
```

fabric.context_managers.**lcd**(*path*)

ローカルのカレントワーキングディレクトリを更新するためのコンテキストマネージャー。

このコンテキストマネージャーは *cd* と同じですが、別の環境変数 (`cwd` ではなく `lcwd`) を変更するので、*local* の実行と *get*/*put* へのローカルの引数のみに影響を与えます。

相対パスの引数はローカルユーザーのカレントのワーキングディレクトリに対する相対になり、Fabric(もしくは Fabric を使うコード) がどこで実行されるかによって違ってきます。これは os.getcwd で確認できます。fabfile が利用されている場所からの相対に固定すると便利かもしれません。それは *env.real_fabfile* で確認できます。

バージョン 1.0 で追加.

fabric.context_managers.**path**(*path*, *behavior='append'*)

ラップされたコマンド実行するために使われるパスに与えられた `path` を追加します。

これで囲まれたブロック内の `run` もしくは `sudo` に対するどんな呼び出しも、与えられたコマンドの前に `"PATH=$PATH:<path> "` に似た文字列を暗黙のうちに追加されます。

次のように、オプションの behavior キーワード引数を指定することによって *path* の挙動をカスタマイズできます:

- 'append': カレントの $PATH に与えられたパスを追加します。例えば PATH=$PATH:<path> です。これはデフォルトの挙動になります。
- 'prepend': カレントの $PATH に与えられたパスを前に追加します。例えば PATH=<path>:$PATH です。
- 'replace': $PATH の前の値を完全に無視します。例えば PATH=<path> となります。

注釈: このコンテキストマネージャーは、環境変数 env.path と env.path_behavior のカレントの値を変更する (そしていつものようにその後に元に戻す) ように今は実装されています。しかし、この実装は将来的には変更されるかもしれないので直接手動で変更することはおすすめしません。

バージョン 1.0 で追加.

fabric.context_managers.**prefix**(*command*)

囲まれたすべての run/sudo コマンドの前に、与えられたコマンドと && を追加します。

これは cd とほとんど同一ですが、1 つの文字列を変更する代わりにコマンド文字列のリストに、ネストされた呼び出しが追加されます。

たいていの場合、シェル環境変数をエクスポートしたり変更するような、シェルの状態を変えるシェルスクリプトと一緒に使うといいでしょう。

例えば、このツールのもっとも一般的な利用方法の 1 つは virtualenvwrapper からの workon コマンドと使うものです:

```
with prefix('workon myvenv'):
    run('./manage.py syncdb')
```

上記の例では、実際のシェルコマンドの実行はこのようになります:

```
$ workon myvenv && ./manage.py syncdb
```

このコンテキストマネージャーは *cd* と互換性があるので、もし自分の virtualenv がその postactivate スクリプトで cd できないときは次ようにもできます:

```
with cd('/path/to/app'):
    with prefix('workon myvenv'):
        run('./manage.py syncdb')
        run('./manage.py loaddata myfixture')
```

これは、実行時には次のような結果になります:

```
$ cd /path/to/app && workon myvenv && ./manage.py syncdb
$ cd /path/to/app && workon myvenv && ./manage.py loaddata myfixture
```

最後に、冒頭近くで示唆したように、*prefix* はお好みのようにネストできます。例えば:

```
with prefix('workon myenv'):
    run('ls')
    with prefix('source /some/script'):
        run('touch a_file')
```

結果:

```
$ workon myenv && ls
$ workon myenv && source /some/script && touch a_file
```

不自然かもしれませんが、説明的だと思います。

fabric.context_managers.**quiet**()

settings(hide('everything'), warn_only=True) へのエイリアス。

時折失敗が予期されるかつ/もしくは何も表示させたくない疑問形のリモートコマンドをラップするときに便利です。

例:

```
with quiet():
    have_build_dir = run("test -e /tmp/build").succeeded
```

タスク内で使用されると、上記スニペットは `run: test -e /tmp/build` の行を生成せず、また、標準出力/標準エラー出力のまったく表示させず、コマンドの失敗は無視されます。

参考:

env.warn_only、 *settings*、 *hide*

バージョン 1.5 で追加.

fabric.context_managers.**remote_tunnel**(**args*, ***kwds*)

ローカルの開いているポートからリモートのターゲットにトンネルフォワーディングを作成します。

例えば、クライアントホストにインストールしてあるデータベースへリモートからアクセスさせます:

```
# Map localhost:6379 on the server to localhost:6379 on the client,
# so that the remote 'redis-cli' program ends up speaking to the local
# redis-server.
with remote_tunnel(6379):
    run("redis-cli -i")
```

データベースは、クライアントホストからのみ接続可能なクライアント上にインストールされているとします (クライアント自身の **上に** あるのとは対照的に):

```
# Map localhost:6379 on the server to redis.internal:6379 on the client
with remote_tunnel(6379, local_host="redis.internal")
    run("redis-cli -i")
```

`remote_tunnel` は 4 個までの引き数を取ります:

- `remote_port` (必須) リッスンするリモートホストのポート。
- `local_port` (オプション) は接続するローカルのポートです。デフォルトはリモートのポートと同じポートです。
- `local_host` (オプション) は接続するローカルで到達可能なコンピュータ (DNS 名か IP アドレス) です。デフォルトは `localhost` です (これはつまり、Fabric が動いているのと同じコンピュータのことです)。
- `remote_bind_address` (オプション) は、カレントターゲット上のリッスンするためにバインドするためのリモートの IP アドレスです。これはターゲット上のインターフェースに割り当てられた IP である必要があります (もしくはそうした IP をリゾルブする DNS 名)。すべてのインターフェースにバインドするには "0.0.0.0" を使います。

注釈: デフォルトでは、たいていの SSH サーバはリモートトンネルにローカルホストのインターフェース (127.0.0.1) をリッスンすることしか許可していません。こうしたケースでは `remote_bind_address` はサーバーによって無視され、トンネルは 127.0.0.1 だけをリッスンします。

`fabric.context_managers.`**`settings`**(**args*, ***kwargs*)

コンテキストマネージャーをネストおよび/もしくは `env` 変数を上書きします。

`settings` には 2 つの目的があります:

- もっとも便利なのが、 `env` を与えられるどんなキーワード引き数、例えば `with settings(user='foo'):` などで一時的に上書き更新できることです。元の値は、もしあればですが、`with` ブロックが閉じられると戻されます。
  - キーワード引き数 `clean_revert` は `settings` 自身にとっては特別な意味を持ち (下記参照)、実行前に取り除かれます。
- 加えて、 contextlib.nested を使うと、与えられたどんな非キーワード引き数でもネストできます。これは他のコンテキストマネージャーである必要がありますが、例えば `with settings(hide('stderr'), show('stdout')):` などです。

これらの挙動は望めば同時に指定することも可能です。これがなぜ便利であるのかは例でわかりやすく説明します:

```
def my_task():
    with settings(
        hide('warnings', 'running', 'stdout', 'stderr'),
        warn_only=True
    ):
        if run('ls /etc/lsb-release'):
            return 'Ubuntu'
        elif run('ls /etc/redhat-release'):
            return 'RedHat'
```

上のタスクは run 命令文を実行しますが、もし ls が失敗した場合に中断するのではなく警告を出し、その警告自身を含めたすべての出力はユーザーには表示されません。その結果としてこのシナリオでは、通常であれば発生するたくさんの出力を発生させずに、リモートホストがどのシステムのタイプなのかを呼び出しが判別することができる完全に静かなタスクとなります。

したがって *settings* は、特定のレベルの出力を隠す (もしくは表示する) のとともに、もしくはコンテキストマネージャーとして実装された他の Fabric の機能のひとつとともに、環境変数とのどんな組み合わせでもセットするために利用可能です。

clean_revert が True にセットされると、settings はネストされたブロック内で変更されたキーは **元には戻しません** 。その代わり、与えられたときと同じ値を持つキーだけを元に戻します。さらに例をご覧になるとより理解できるでしょう。以下は settings の通常の動作です:

```
# Before the block, env.parallel defaults to False, host_string to None
with settings(parallel=True, host_string='myhost'):
    # env.parallel is True
    # env.host_string is 'myhost'
    env.host_string = 'otherhost'
    # env.host_string is now 'otherhost'
# Outside the block:
# * env.parallel is False again
# * env.host_string is None again
```

env.host_string の内部での変更は取り消されていますが、これが常に希望の状態であるとは限りません。ここで clean_revert の登場です:

```
# Before the block, env.parallel defaults to False, host_string to None
with settings(parallel=True, host_string='myhost', clean_revert=True):
    # env.parallel is True
    # env.host_string is 'myhost'
    env.host_string = 'otherhost'
    # env.host_string is now 'otherhost'
# Outside the block:
# * env.parallel is False again
# * env.host_string remains 'otherhost'
```

clean_revert 有効時には settings の利用前に env に存在しなかった新しいキーも保存されます。

False の時はこうしたキーはブロック終了時に取り除かれます。

バージョン 1.4.1 で追加: clean_revert キーワード引数。

fabric.context_managers.**shell_env**(**kw)

ラップされたコマンドのためのシェル環境変数をセットします。

例えば、以下は Python の ZMQ ライブラリをインストールするときに関連する環境変数をどのようにセットするのかをお見せしています:

```
with shell_env(ZMQ_DIR='/home/user/local'):
    run('pip install pyzmq')
```

prefix と同様に、これは実質的に run コマンドを以下に変更します:

```
$ export ZMQ_DIR='/home/user/local' && pip install pyzmq
```

複数のキーバリューペアを同時に与えることができます。

注釈: Windows のローカルホストから実行したときに *local* の挙動に営業が出る場合、この機能の実装には SET コマンドが使われます。

---

fabric.context_managers.**show**(*args, **kwds)

与えられた出力 groups を True にセットするためにコンテキストマネージャー。

groups は output で定義された出力グループを命名している 1 つ以上の文字列でなければなりません。与えられたグループは、囲まれたブロックでの継続性のため True にセットされ、その後は前の状態に戻ります。

例えば、デバッグ出力を有効にします (デフォルトではたいていオフになっています):

```
def my_task():
    with show('debug'):
        run('ls /var/www')
```

ほぼすべての出力グループがデフォルトでは表示されますが、通常は非表示になっている debug グループをオンにしたり、ご自分のコードを呼び出すコードが *hide* で出力を隠そうとしていることを知っていたり疑っていたりする場合に *show* はとても有用です。

fabric.context_managers.**warn_only**()

settings(warn_only=True) のエイリアスです。

参考:

env.warn_only、 *settings*、 *quiet*

3.1.3 デコレーター

3.1.4 ドキュメントヘルパー

fabric.docs.**unwrap_tasks**(*module*, *hide_nontasks=False*)

モジュール 上のタスクオブジェクトをラップされた関数に置き換えます。

特に *WrappedCallableTask* のインスタンスを探し、その .wrapped 属性 (元のデコレートされた関数) と置き換えます。

これは Sphinx の autodoc ツールとともに使用するよう意図されていて、プロジェクトの conf.py の最後のほうで実行されます。autodoc 拡張機能が関数シグネチャなどにおける "実際の" 関数に対して完全なアクセスを確実に持つようにします。 unwrap_tasks 利用なしでは autodoc は関数シグネチャにアクセスできません (__doc__ などで見ることはできますが)。

例えば、conf.py の最後の方で:

```
from fabric.docs import unwrap_tasks
import my_package.my_fabfile
unwrap_tasks(my_package.my_fabfile)
```

それ以上のことも可能です。hide_nontasks=True と指定することによってタスク以外のすべての関数を明示的に **隠す** ことができます。これによりすべてのオブジェクトで "これはタスク?" のチェックに失敗するようになり、そのためプライベートのように見え、その結果 autodoc はこれらをスキップします。

したがって、hide_nontasks は、fabfile に実際のタスクを持つサブルーチンが入っていて、本当のタスク **だけ** をドキュメント化したいときに便利です。

もし実際のセッションを使用している Fabric コード内で (Sphinx の conf.py 内ではなく) これを実行するのなら、すぐに病院で診てもらってください。

参考:

WrappedCallableTask, task

3.1.5 ネットワーク

ネットワーク接続や関連するトピックを扱うクラスやサブルーチンです。

fabric.network.**disconnect_all**()

その時点のすべての接続されているサーバーから接続を解除します。

fab のメインループの最後で使われ、またライブラリーのユーザーによって利用されることを想定しています。

class fabric.network.**HostConnectionCache**

ホスト接続/クライアントのキャッシュを許可する辞書サブクラス。

このサブクラスは、キーがリクエストされたときに理にかなった新しいクライアント接続を作成するか、もしくはその前に作成された接続を返します。

また、ゲートウェイ接続と ProxyCommand の実装が求められたときに新しいソケットライクなオブジェクトを作成し、それらを内部接続メソッドに渡す処理をします。

キーバリューは Fabric 全体のホスト指定子と同じです。オプションとしてユーザー名 + @、必須ホスト名、オプションの : + ポート番号です。

- example.com - よくあるインターネットのホストアドレス。
- firewall -特殊な、しかし有効なローカルのホストアドレスです。
- user@example.com - 特定のユーザー名が付加されています。
- bob@smith.org:222 - 特定の非標準なポートをともなっています。

ユーザー名が与えられない場合は env.user が使われます。 env.user のデフォルトは起動時のカレント実行中ユーザーですが、ユーザーのコードもしくはコマンドラインフラグでの指定によりオーバライドできます。

同一のホスト名で別のユーザー名が指定された場合、複数のクライアント接続が形成されます。例えば、user1@example.com は user1 としてログインされた example.com への接続を作成しますが、その後で ‘‘user2@example.com‘‘ を指定すると user2 として新しく 2 番めの接続が作られます。

同じことがポートにも当てはまります。同じホストへ 2 つの違うポートを指定すると、2 つの別々の接続が作成されます。ポートが指定されなければ、22 が適用されます。つまり example.com は example.com:22 と同等になります。

__contains__(*k*) → True if D has a key k, else False

__delitem__(*key*)

x.__delitem__(y) <==> del x[y]

__getitem__(*key*)

自動接続 + 戻り値接続オブジェクト

__setitem__(*key*, *value*)

x.__setitem__(i, y) <==> x[i]=y

__weakref__

そのオブジェクトへの弱い参照 (定義されていれば)

connect(*key*)

強制的に key ホスト文字列へ新しく接続。

fabric.network.**connect**(*user*, *host*, *port*, *cache*, *seek_gateway=True*)

与えられたホストへ接続された新しい SSHClient インスタンスを作成し、返します。

パラメータ

- **user** – 接続するときのユーザー名。
- **host** – ネットワークのホスト名。
- **port** – SSH デーモンのポート。
- **cache** – ゲートウェイが有効なときにホストをキャッシュ/保存するために利用される HostConnectionCache インスタンス。
- **seek_gateway** – その接続でゲートウェイソケットのセットアップをトライするかどうか。実際のゲートウェイ接続が反復を防ぐことを可能にするために利用されます。

fabric.network.**denormalize**(*host_string*)

与えられたホスト文字列の既定値をストリップします。

ユーザーの部分がデフォルトのユーザーなら取り除かれます。ポートが 22 なら、これも取り除かれます。

fabric.network.**disconnect_all**()

その時点のすべての接続されているサーバーから接続を解除します。

fab のメインループの最後で使われ、またライブラリーのユーザーによって利用されることを想定しています。

fabric.network.**get_gateway**(*host*, *port*, *cache*, *replace=False*)

必要な場合にゲートウェイソケットを作成し、返します。

この関数はゲートウェイもしくはプロキシコマンド設定のための env をチェックし、最終的なホスト接続によって利用されるための必要なソケットライクなオブジェクトを返します。

パラメータ

- **host** – ターゲットサーバーのホスト名。
- **port** – ターゲットサーバーへ接続するためのポート。
- **cache** – HostConnectionCache オブジェクトで、どのゲートウェイ SSHClient オブジェクトで取得/キャッシュされるかを指定します。
- **replace** – キャッシュされたゲートウェイクライアントオブジェクトを強制的に置き換えるかどうかを指定します。

戻り値 socket.socket ライクのオブジェクト、もしくはなにも作成されていなければ None 。

fabric.network.**join_host_strings**(*user*, *host*, *port=None*)

ユーザー/ホスト/ポート文字列を user@host:port の統合された文字列にします。

この関数はユーザー/ポート文字列がない場合の対処はしません。その場合は normalize 関数をご覧ください。

host が IPv6 のようであれば角括弧で囲みます。

port が省略されている場合は、返される文字列は user@host の形式になります。

fabric.network.**key_filenames**()
使用中の env.host_string のための SSH キーのファイル名リストを返します。

リストへの正規化も含め、 ssh_config と env.key_filename を取り入れます。また、すべてのキーファイル名に対する os.path.expanduser 拡張を実行します。

fabric.network.**key_from_env**(*passphrase=None*)
プライベートキーのテキスト文字列から paramiko 対応のキーを返します。

fabric.network.**needs_host**(*func*)
env.host_string が空の時、 env.host_string の値を入力するためのプロンプトを表示します。

このデコレータは基本的にあれやこれやでホスト/ホストリストの指定を忘れてしまうぼーっとしたユーザーのためのセーフティネットです。ネットワーク接続を必要とする操作をラップするために使用します。

コマンドは main() のホストごとに実行されるため、この時点で複数のホストを指定することが不可能です。したがって、単一のホストのみの入力に限られます。

このデコレータは env.host_string をセットするので、コマンドごとにプロンプトを一回 (一回のみ) 表示させます。コマンドとコマンドの間でも main() が env.host_string をクリアするので、結局はこのデコレータがコマンドごとにユーザーにプロンプトを表示します (もちろん複数コマンドにホストがセットされていない場合です)。

fabric.network.**normalize**(*host_string*, *omit_port=False*)
与えられたホスト文字列を正規化し、明示的なホスト、ユーザー、ポートを返します。

omit_port が与えられてそれが True の場合、ホストをユーザーのみが返されます。

この関数は、Fabric がそのように設定されている場合に SSH コンフィグファイルを処理し、いくつかの初期値を埋めたりホスト名エイリアスをスワップするために利用されます。

fabric.network.**normalize_to_string**(*host_string*)
normalize() はタプルを返しますが、これは別の有効なホスト名を返します。

fabric.network.**prompt_for_password**(*prompt=None*, *no_colon=False*, *stream=None*)
必要時に新しいパスワードのためのプロンプトを表示し、それを返します。それ以外は何も返しません。

no_colon が True でない限り最後にコロンが付加されます。

もしユーザーが何もパスワードを入力しなければ、空ではないパスワードを入力するまでプロンプトがそのユーザーに表示されます。

prompt_for_password はその時点で接続しているホストに応じてユーザープロンプトを自動生成します。この挙動をオーバーライドするには prompt に文字列の値を指定します。

stream はプロンプトが書き込まれる先のストリームです。何も指定されなければ、デフォルトの sys.stderr になります。

fabric.network.**ssh_config**(*host_string=None*)

カレントの env.host_string ホストの値のための ssh 設定辞書を返します。

読み込んだ SSH コンフィグファイルをメモ化しますが、ホストごとの結果に限定されたものではありません。

この関数は必要な "SSH コンフィグが有効か?" チェックを行い、有効でなければ単に空の辞書を返します。SSH コンフィグが **有効** で env.ssh_config_path の値が有効なファイルではない場合、中断します。

host_string として特定のホスト文字列を与えることもできます。

3.1.6 オペレーション

fabfile 内とその他のコア以外のコードで使われる run()/sudo() などの関数です。

fabric.operations.**get**(**args*, ***kwargs*)

リモートホストから一つもしくは複数のファイルをダウンロードします。

get はダウンロードしたすべてのローカルファイルへの絶対パスを含む反復可能オブジェクトを返します。これはもし local_path が StringIO オブジェクトの場合には空になります (StringIO オブジェクト利用時についての詳細は下の方を参照してください)。このオブジェクトはまた、ダウンロードに失敗したすべてのリモートファイルのパスを含む .failed 属性と not .failed と同等の .succeeded 属性を提示します。

remote_path はダウンロードするリモートファイルもしくはリモートディレクトリのパスで、例えば "/var/log/apache2/*.log" などのシェル glog シンタックスを含むことができ、チルダはリモートのホームディレクトリに置き換えます。相対パスはリモートのユーザーのホームディレクトリからの相対位置として、もしくは *cd* によってコントロールされたリモートのカレントワーキングディレクトリからの相対位置として扱われます。もしリモートパスにディレクトリが指定されている時、そのディレクトリが再帰的にダウンロードされます。

local_path はダウンロードしたリモートファイルもしくはファイルが保存される場所へのパスです。もし相対パスの場合、*lcd* によってコントロールされるローカルのワーキングディレクトリを受け取ります。これは Python 標準の辞書ベースの挿入によって、次の変数を伴って挿入されうことがあります:

- host: env.host_string の値で、 myhostname もしくは user@myhostname-222 などです (ホスト名とポートの間のコロンはファイルシステムの互換性を最大限にするためダッシュに置き換えられます)。

- dirname: src/projectname/utils.py の src/projectname などのリモートファイルパスのディレクトリ部分です。
- basename: src/projectname/utils.py の utils.py などのリモートファイルパスのファイル名部分です。
- path: src/projectname/utils.py などのリモートのフルパスです。

SFTP プロトコル (*get* が利用します) には接続ユーザーによって所有されていない場所からファイルをダウンロードする直接的な方法はありませんが、use_sudo=True と指定することによりこれが可能になります。これがセットされたとき、この設定により *get* は (sudo を使って) リモートファイルをリモート側の一時フィアル保管場所 (デフォルトではリモートユーザーの $HOME ですが、temp_dir によって上書き可能です) にコピーし、それからそのファイルを local_path ダウンロードします。

注釈: remote_path がディレクトリの絶対パスのとき、内部ディレクトリだけがローカルに作られて上記の変数に渡されます。例えば、get('/var/log', '%(path)s') は、ローカルのワーキングディレクトリで apache2/access.log、postgresql/8.4/postgresql.log などのように出力されます。var/log/apache2/access.log などのようには **出力されません**。

さらに、単一のファイルをダウンロードするときは %(dirname)s と %(path)s は当然意味をなしませんので、空となりそれぞれ %(basename)s と同等になります。したがって get('/var/log/apache2/access.log', '%(path)s') のような呼び出しは、var/log/apache2/access.log ではなくローカルのファイル名 access.log として保存されます。

この挙動はコマンドラインの scp と一致するようにするためです。

空のままの場合、複数ホスト実行での安全性のため local_path はデフォルトの "%(host)s/%(path)s" になります。

警告: local_path 引き数が %(host)s を含んでいなくて *get* の呼び出しが複数のホストに対して実行される場合、ローカルのファイルはそれぞれ実行が成功したもので上書きされます!

local_path が上記の変数 (例えば、単純な場合では指定されたファイルパス) を使用していない場合、scp や cp と似た動作をします。必要に応じて既存のファイルを上書きし、ディレクトリが与えられた場合はそのディレクトリへのダウンロードされます (例えば、get('/path/to/remote_file.txt', 'local_directory') は local_directory/remote_file.txt を作成します)。

もしくは local_path は、open('path', 'w') の結果や StringIO のインスタンスなど、ファイルライクなオブジェクトであることも可能です。

注釈: ディレクトリをファイルライクなオブジェクトに *get* して入れようとするのは無効で、エラーにな

ります。

注釈: この関数は seek と tell を使ってファイルライクなオブジェクトのコンテンツ全体を上書きします。これは *put* (ファイル全体も考慮します) の挙動と一致させるためです。とはいえ、 *put* とは違い、このファイルポインターは前のロケーションには戻されません。これは意味を成さないですし、不可能でもあります。

注釈: StringIO が name 属性を持つ場合、デフォルトの <file obj> の代わりにそれが Fabric の出力に使われます。

バージョン 1.0 で変更: リモートのワーキングディレクトリを *cd* で操作されたものとしてみなし、ローカルのワーキングディレクトリを *lcd* で操作されたものとしてみなします。

バージョン 1.0 で変更: local_path 引き数にファイルライクなオブジェクトが使えます。

バージョン 1.0 で変更: local_path はパスとホスト関連変数が挿入されたものを含むことができます。

バージョン 1.0 で変更: ディレクトリは remote_path 引き数で明示することができ、再帰的なダウンロードを実行すます。

バージョン 1.0 で変更: 返り値はダウンロードされたローカルファイルのパスの反復可能オブジェクトで、.failed と .succeeded 属性を提示します。

バージョン 1.5 で変更: ログ出力にファイルライクなオブジェクトへの name 属性を許可するようになりました。

fabric.operations.**local**(*command*, *capture=False*, *shell=None*)

ローカルシステム上でコマンドを実行します。

local は shell=True が有効化されている Python ビルトインの subprocess モジュールの便利なラッパーです。なにか特別なことをする必要がある場合は subprocess モジュールを直接使うことを検討してください。

shell はダイレクトに subprocess.Popen の execute 引き数 (これが利用するローカルのシェルを決めます) に渡されます。リンク先のドキュメントにあるように、Unix ではデフォルトでは /bin/sh を使います。したがって、例えばこの値を /bin/bash などに設定したい場合に便利です。

local はいまのところ *run*/*sudo* のように出力を同時にプリントしたりキャプチャしたりすることはできません。 capture キーワード引き数によって、必要に応じてプリントとキャプチャを切り替えることができ、デフォルトは False になっています。

capture=False の時、ローカルのサブプロセスの標準出力と標準エラー出力のストリームはターミナル

に直接繋がれます。これはグローバルの *output controls* output.stdout と output.stderr を使って片方もしくは両方を必要に応じて隠すことができます。このモードでは返り値の標準出力/標準エラー出力は常に空になります。

capture=True の時には、ターミナルにはサブプロセスからのどんな出力も表示されませんが、返り値にはキャプチャされた標準出力/標準エラー出力が含まれます。

どちらにせよ、*run* と *sudo* と同様にこの返り値は return_code、stderr、failed、succeeded、command、 real_command 属性を提示します。詳細は *run* をご覧ください。

local は *lcd* コンテキストマネージャーを優先し、これによりリモート側とは切り離して (これは *cd* を優先) カレントのワーキングディレクトリをコントロールできるようにします。

バージョン 1.0 で変更: succeeded と stderr 属性を追加しました。

バージョン 1.0 で変更: *lcd* コンテキストマネージャーを優先するようになりました。

バージョン 1.0 で変更: capture のデフォルト値を True から False に変更しました。

バージョン 1.9 で追加: 返り値属性の .command と .real_command。

fabric.operations.**open_shell**(**args*, ***kwargs*)

リモート側で完全な対話式シェルを起動します。

command が与えられると、起動ユーザーにコントロールが渡される前にそれがパイプに送り込まれます。

この機能は、大量のシェルベースのコマンドや、デバッギング時やリモート側プログラムの障害に要する完全にインタラクティブなリカバリー作業時などの一連のコマンドとのやりとりが必要なときにとても役に立ちます。

Fabric スクリプトの途中にインタラクティブなシェルセッションを組み入れるのは簡単な方法とみなすべきで、*run* のドロップインの代替ではありません。また、リモート側とのやりとりが可能で (与えられたコマンドが実行されているときだけですが)、エラー処理や標準出力/標準エラー出力のキャプチャのようなとても強いプログラム能力を持ちます。

厳密には、*open_shell* は *run* よりも良いインタラクティブな体験をもたらします。しかし完全なリモートシェルの利用は、シェル内でのプログラムの起動に失敗したのかどうかを Fabric が判別するのを妨げ、ログインバナー、プロンプト、エコーされた標準入力などのシェルの出力で標準出力/標準エラー出力が判別しにくくなります。

そのため、この関数は返り値を持たず、どんなリモートプログラムがエラーで終わっても Fabric の失敗ハンドリングを引き起こしません。

バージョン 1.0 で追加.

fabric.operations.**prompt**(*text*, *key=None*, *default=''*, *validate=None*)

text でユーザーに入力を促し、(raw_input のように) その入力を返します。

利便性のため空白文字が一つだけ付加されます。したがって、プロンプトテキストはクエスチョンマークやコロンで終えるようにするといいでしょう (例えば `prompt("What hostname?")`)。

`key` が与えられた場合、そのユーザーの入力は *prompt* によって返されるものに加えて `env.<key>` として保存されます。`env` にそのキーがすでに存在する場合はその値が上書きされ、そのユーザーには警告が表示されます。

`default` が与えられた場合、それが角かっこ内に表示され、ユーザーが何も (つまり何もテキストを入力せず Enter を押下) を入力しなかった場合にそれが使用されます。`default` のデフォルトは空の文字列です。空の文字列以外ならスペースが一つ追加されます。したがって、 `prompt("What hostname?", default="foo")` のような呼び出しは (`[foo]` の後ろにスペースをともなった) `What hostname? [foo]` のプロンプトとして表示されます。

オプションのキーワード引き数 `validate` は呼び出し可能なもの、もしくは文字列です:

- 呼び出し可能なものの場合、ユーザーの入力とともに呼び出され、成功時に格納される値を返します。失敗時には例外メッセージとともに例外を発生させ、ユーザーに表示されます。
- 文字列の場合、`validate` に渡される値は正規表現として使用されます。そのため、この場合は生の文字列を使用することをおすすめします。正規表現に関する注意点ですが、(`^` と `$` で区切られた) 完全マッチではない場合は、そうなるようにします。つまり、入力は正規表現に完全にマッチさせます。

どちらにせよ、バリデーションに通るまで (もしくはユーザーが `Ctrl-C` を押すまで) *prompt* は再度入力を待ちます。

注釈: *prompt* は *env.abort_on_prompts* を優先し、このフラグが `True` にセットされている場合はプロンプトを表示する代わりに *abort* を呼び出します。もそこれにかかわらずユーザーの入力をブロックしたい場合は、 *settings* で囲ってみてください。

例:

```
# Simplest form:
environment = prompt('Please specify target environment: ')

# With default, and storing as env.dish:
prompt('Specify favorite dish: ', 'dish', default='spam & eggs')

# With validation, i.e. requiring integer input:
prompt('Please specify process nice level: ', key='nice', validate=int)

# With validation against a regular expression:
release = prompt('Please supply a release name',
        validate=r'^\w+-\d+(\.\d+)?$')

# Prompt regardless of the global abort-on-prompts setting:
```

(次のページに続く)

(前のページからの続き)

```
with settings(abort_on_prompts=False):
    prompt('I seriously need an answer on this! ')
```

fabric.operations.**put**(**args*, ***kwargs*)

リモートホストへ一つもしくは複数のファイルをアップロードします。

put はアップロードしたすべてのリモートファイルの完全パスを含む反復可能オブジェクトを返します。この反復可能オブジェクトはまた、アップロードに失敗したすべてのローカルファイル (そしておそらく真偽テストとして使われたもの) のパスを含む .failed 属性を提示します。

local_path はローカルのファイルもしくはディレクトリの相対もしくは絶対パスで、Python の glob モジュールによって理解されるよう (この挙動を無効にしたい場合は use_glob=False としてください)、シェルスタイルのワイルドカードを含むことがあります。(os.path.expanduser によって実装されている) チルダの展開も実行されます。

local_path は、 open('path') の結果や StringIO のインスタンスなど、ファイルライクなオブジェクトであることも可能です。

注釈: この場合、*put* は seek を使ってリワインドすることによりファイル形式オブジェクトのコンテンツ全体を読み取ろうとします (そして前回のファイル位置を保存するためにその後に tell を使います)。

---

remote_path は相対的もしくは絶対的な場所で、リモートホストに適用されるものです。相対パスはリモートユーザーのホームディレクトリに対する相対で、必要に応じてチルダ展開 (例えば ~/.ssh/) が実施されます。

どちらのパス引き数も空の文字列の場合、適切な側のカレントワーキングディレクトリによって置き換えられます。

SFTP プロトコル (*put* が利用します) には接続ユーザーによって所有されていない場所へファイルをアップロードする直接的な方法はありませんが、use_sudo=True と指定することによりこれが可能になります。これがセットされたとき、この設定により *put* はローカルファイルをリモート側の一時フィアル保管場所 (デフォルトではリモートユーザーの $HOME ですが、 temp_dir によって上書き可能です) にアップロードし、それから *sudo* を利用してそれらのファイルを remote_path に移動します。

場合によっては、新しくアップロードしたファイルのモードを強制的にローカル側のものと一致させたほうが望ましいこともあります (実行可能ファイルをアップロードした時など)。これを行うには mirror_local_mode=True を指定します。

代わりに、os.chmod もしくは Unix の chmod コマンドと同様に mode キーワード引き数を使って同じモードを指定することもできます。

put は *cd* を優先するので、もしあれば、remote_path の相対値がリモートのカレントワーキングディレ

クトリによって追加されます。したがって、以下の例では ~/files/test.txt ではなく /tmp/files/test.txt にアップロードしようとします:

```
with cd('/tmp'):
    put('/path/to/local/test.txt', 'files')
```

同じように *lcd* の利用は local_path に影響を与えます。

例:

```
put('bin/project.zip', '/tmp/project.zip')
put('*.py', 'cgi-bin/')
put('index.html', 'index.html', mode=0755)
```

注釈: StringIO が name 属性を持つ場合、デフォルトの <file obj> の代わりにそれが Fabric の出力に使われます。

バージョン 1.0 で変更: リモートのワーキングディレクトリを *cd* で操作されたものとしてみなし、ローカルのワーキングディレクトリを *lcd* で操作されたものとしてみなします。

バージョン 1.0 で変更: local_path 引き数にファイルライクなオブジェクトが使えます。

バージョン 1.0 で変更: ディレクトリは local_path 引き数で明示することができ、再帰的なアップロードを実行します。

バージョン 1.0 で変更: 返り値はアップロードされたローカルファイルのパスの反復可能オブジェクトで、.failed と .succeeded 属性を提示します。

バージョン 1.5 で変更: ログ出力にファイルライクなオブジェクトへの name 属性を許可するようになりました。

バージョン 1.7 で変更: glob を無効にできるように use_glob オプションを追加。

fabric.operations.**reboot**(**args*, ***kwargs*)

リモートシステムを再起動します。

これは一時的に Fabric の再接続設定 (*timeout* と *connection_attempts*) を調整し、少なくとも wait 秒は再接続を止めないようにします。

注釈: Fabric 1.4 以降では一つのセッション途中での再接続の機能は内部 API の利用を必要とはしません。この機能を公式には非推奨とはしませんが、これにさらに機能を追加することは優先度としては低いです。

より細かく制御したいユーザーは、この関数のソースコード (6 行で、よくコメントしてあります) をチェックし、異なるタイムアウト/試行値または追加のロジックを使用して独自の変更を書くことを奨励します。

バージョン 0.9.2 で追加.

バージョン 1.4 で変更: wait キーワード引き数を任意に変更し、新しい再接続機能を活用するようにリファクタリング。実際には、再接続前に wait 秒待つ必要はないかもしれない。

fabric.operations.**require**(**keys*, ***kwargs*)

共有環境辞書に与えられているキーをチェックし、なければ中止します。

位置指定引数は文字列で、どの環境変数をチェックするのかを指定します。与えられた引き数がどれでも存在しない場合は、Fabric は実行を中止し、見つからなかったキーの名称を出力します。

位置指定引数の used_for は文字列であることも可能で、その場所でそれがなぜ必須なのかをユーザーに知らせるためのエラーに出力されます。 used_for は次の文字列に似た文字列の一部として出力されます:

```
"Th(is|ese) variable(s) (are|is) used for %s"
```

そしてこれを適切にフォーマットします。

位置指定引数の provided_by は、単一もしくは複数のキーをセットするためにユーザーが実行することのできる複数の関数または複数の関数名、あるいはひとつの関数またはひとつの関数名のリストとして渡すことができます。もし必須事項を満たしていなければ、エラーメッセージの中にそれが含められます。

キーワード引き数はグループとしてすべての与えられたキーに適用されるとうことを前提としていることに留意してください。 used_for を複数指定する必要を感じた場合、例えば、ロジックを require() への複数の呼び出しに分割するといいでしょう。

バージョン 1.1 で変更: 単一の値ではなく、反復可能オブジェクト provided_by の値を可能にしました。

fabric.operations.**run**(**args*, ***kwargs*)

リモートホストに対してシェルコマンドを実行します。

shell が True (デフォルト) の場合、 *run* はシェルインタープリター経由で渡されるコマンドを実行します。この値は env.shell (デフォルトは /bin/bash -l -c "<command>" と同様なものです) の設定によって制御可能です。shell が True のとき、コマンド内の二重引用符 (") またはドルマーク ($) 文字は自動的にエスケープされます。

run は単一の文字列 (おそらくは複数行) としてリモートプログラムの標準出力の結果を返します。この文字列はそのコマンドが成功したか失敗したかを明記するため failed と succeeded の真偽値属性を提示し、 return_code 属性としてのリターンコードも含みます。さらに、それぞれ .command と .real_command として、リクエストされ、実際に実行されたコマンド文字列のコピーも含んでいます。

ローカルのターミナルに入力したすべてのテキストは、それが実行されるたびにリモートに送られます。つまり、パスワードやその他のプロンプトと自然な感じでやりとりできるようにします。この挙動の詳細は **リモートプログラムとのやりとり** をご覧ください。

該当のコマンドにとってその存在が問題になる場合は、リモート側の擬似ターミナルの生成に先立って pty=False を渡すこともできます。とは言え、そうするとそのコマンドの実行中はタイプした入力をパス

ワードも含めてすべてエコーするよう Fabric に強制します。(pty=True であれば、リモートの擬似ターミナルはエコーするものの、パスワードスタイルのプロンプトは賢く扱います) 詳細は **擬似ターミナル** をご覧ください。

同じように、リモートプログラムの標準エラー出力ストリーム (この関数の返り値に stderr として明示されます) をプログラム的に調べる必要がある場合は、 combine_stderr=False をセットすることができます。これを設定すると高い確率で自分のターミナルでの出力を文字化けさせてしまいます (*run* によって返される結果文字列は適切に分離されていますが) 詳細は **標準出力と標準エラー出力の結合** をご覧ください。

ゼロ以外のリターンコードを無視するには warn_only=True を指定します。ゼロ以外のコードの無視とコマンドのサイレント実行の強制の両方を行うには quiet=True を指定します。

リモート側の標準出力および/もしくは標準エラー出力の表示にどのローカルのストリームを使用するかをオーバーライドするには stdout もしくは stderr を指定します。

例えば、 run("command", stderr=sys.stdout) はリモートの標準エラー出力をローカルの標準出力に表示しますが、返り値の自身の識別属性は保持されます (上述の通り)。あるいは、例えば myout = StringIO(); run("command", stdout=myout) のように、独自のストリームオブジェクトもしくはロガーを提供することも可能です。

リモートプログラムの実行に時間がかかりすぎているときに例外を発生させたい場合は timeout=N を指定します。 N は秒の整数値で、これより時間がかかるとタイムアウトさせます。これは run に CommandTimeout の例外を発生させます。

引用符やドルマークなどに対する Fabric の自動エスケープを無効にしたい場合は shell_escape=False を指定します。

例:

```
run("ls /var/www/")
run("ls /home/myuser", shell=False)
output = run('ls /var/www/site1')
run("take_a_long_time", timeout=5)
```

バージョン 1.0 で追加: succeeded と stderr の返り値の属性、combine_stderr キーワード引き数、インタラクティブな挙動。

バージョン 1.0 で変更: pty のデフォルト値を True に変更しました。

バージョン 1.0.2 で変更: combine_stderr のデフォルト値が True ではなく None になりました。しかし、デフォルトの **挙動** に変更はなく、グローバルなセッティングは True のままです。

バージョン 1.5 で追加: quiet、warn_only、stdout、stderr キーワード引き数。

バージョン 1.5 で追加: 返り値属性の .command と .real_command。

バージョン 1.6 で追加: timeout 引き数。

バージョン 1.7 で追加: shell_escape 引き数。

fabric.operations.**sudo**(**args*, ***kwargs*)

リモートホストに対してスーパーユーザー権限でシェルコマンドを実行します。

sudo はスーパーユーザー権限を提供するために与えられた command を sudo プログラムへの呼び出し内にラップする以外は、あらゆる点で *run* と同一です。

sudo は追加で user と group の引き数を取ります。これは sudo に渡され、root 以外の user や group として実行できるようにします。たいていのシステムでは、sudo プログラムは username/group の文字列もしくは userid/groupid (uid/gid) の整数値をとることができます。user と group は同じように文字列もしくは整数値となります。

モジュールレベル、もしくは同じ user の値を持つ複数の sudo 呼び出しを行う場合は *settings* 経由で *env.sudo_user* をセットすることが可能です。もちろん、明示的な user 引き数はこのグローバル設定をオーバーライドします。

例:

```
sudo("~/install_script.py")
sudo("mkdir /var/www/new_docroot", user="www-data")
sudo("ls /home/jdoe", user=1001)
result = sudo("ls /tmp/")
with settings(sudo_user='mysql'):
    sudo("whoami") # prints 'mysql'
```

バージョン 1.0 で変更: *run* の変更および追加点をご覧ください。

バージョン 1.5 で変更: *env.sudo_user* を優先するようになりました。

バージョン 1.5 で追加: quiet、warn_only、stdout、stderr キーワード引き数。

バージョン 1.5 で追加: 返り値属性の .command と .real_command。

バージョン 1.7 で追加: shell_escape 引き数。

3.1.7 タスク

class fabric.tasks.**Task**(*alias=None*, *aliases=None*, *default=False*, *name=None*, **args*, ***kwargs*)

Fabric のタスクとしてピックアップされることを要求するオブジェクトのための抽象基底クラスです。

fab ツールによって読み込まれる fabfile 内にサブクラスのインスタンスが存在するときに有効なタスクとして扱われます。

Task サブクラスの実装と利用の方法の詳細は *new-style tasks* の利用方法のドキュメントをご覧ください。

バージョン 1.1 で追加.

__init__(*alias=None*, *aliases=None*, *default=False*, *name=None*, **args*, ***kwargs*)
x.__init__(...) initializes x; see help(type(x)) for signature

__weakref__
そのオブジェクトへの弱い参照 (定義されていれば)

get_hosts_and_effective_roles(*arg_hosts*, *arg_roles*, *arg_exclude_hosts*, *env=None*)
与えられたタスクが利用するホストリストと利用されているロールを含んだタプルを返します。

ホストリストがどのようにセットされるのかについての詳細はドキュメントの **ホストリストがどのように作られるか** をご覧ください。

バージョン 1.9 で変更.

class fabric.tasks.**WrappedCallableTask**(*callable*, **args*, ***kwargs*)
与えられた呼び出し可能オブジェクトを透過的にラップし、有効なタスクであるとマークします。

通常は直接ではなく `task` 経由で利用されます。

バージョン 1.1 で追加.

参考:

unwrap_tasks、 `task`

__call__(*...*) <==> *x(...)*

__init__(*callable*, **args*, ***kwargs*)
x.__init__(...) initializes x; see help(type(x)) for signature

fabric.tasks.**execute**(*task*, **args*, ***kwargs*)
ホスト/ロールデコレーターなどを履行し、 `task` (呼び出し可能なオブジェクトもしくはタスク名) を実行します。

`task` は実際の呼び出し可能なオブジェクトもしくは登録されたタスク名で、まるでその名称が (*namespaced tasks* も含み、例えば `"deploy.migrate"` など) コマンドラインに与えられたかのように呼び出し可能なオブジェクトを調べるために利用されます。

このタスクはホストリストにもとづき各ホストに対して一度ずつ実行されます。ホストリストは、CLI で指定されたタスクと同じように (再び) 生成されたもので、 *-H*、 *env.hosts*、 `hosts` もしくは `roles` デコレーターなどから取り出されます。

`host`、 `hosts`、 `role`、 `roles`、 `exclude_hosts` キーワード引き数は最終的な呼び出しでは取り除かれ、例えば `fab taskname:host=hostname` のようにまるでコマンドラインで指定されたかのように、そのタスクのホストリストのセットに利用されます。

その他のすべての引き数やキーワード引き数は呼び出されたときにそれぞれ `task` (関数自身で、関数をラップする `@task` デコレーターではありません!) に渡されます。そして `execute(mytask, 'arg1', kwarg1='value')` が (各ホストごとに) `mytask('arg1', kwarg1='value')` を実行します。

戻り値 そのホストに対して実行した与えられたタスクの返り値にホスト文字列をマッピングした辞書です。例えば `execute(foo, hosts=['a', 'b'])` は、 `foo` がホスト `a` に対して何も返していなくてホスト `b` に対しては `'bar'` を返している時、 `{'a': None, 'b': 'bar'}` を返します。もし全体的な進行は中止されないけれども与えれれたホストに対してタスク実行に失敗した場合 (*env.skip_bad_hosts* が True のときなど)、そのホストのための返り値はエラーオブジェクトかエラーメッセージになります。

参考:

詳しい説明とさらなる例は *The execute usage docs* を参照してください。

バージョン 1.3 で追加.

バージョン 1.4 で変更: 返り値のマッピングを追加。以前はこの関数は定義された返り値は持っていませんでした。

`fabric.tasks.`**`requires_parallel`**(*task*)

与えられた `task` が並列モードで動作すべきなら True を返します。

具体的には:

- `@parallel` で明示的にマークされているか:
- `@serial` で明示的にマーク **されていない かつ** グローバルの並列オプション (`env.parallel`) が `True` にセットされている。

3.1.8 ユーティリティ

内部のサブルーチンで、例えば、エラーメッセージをともなった実行の中止や複数行でのインデントの実施などです。

class `fabric.utils.`**`RingBuffer`**(*value*, *maxlen*)

`__init__`(*value*, *maxlen*)

x.__init__(...) initializes x; see help(type(x)) for signature

`__setitem__`(*key*, *value*)

x.__setitem__(i, y) <==> x[i]=y

`__setslice__`(*i*, *j*, *sequence*)

x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

`__weakref__`

list of weak references to the object (if defined)

append(*value*)

L.append(object) – append object to end

extend(*values*)

L.extend(iterable) – extend list by appending elements from the iterable

insert(*index*, *value*)

L.insert(index, object) – insert object before index

fabric.utils.**abort**(*msg*)

実行の中止で、標準エラー出力に msg をプリントしエラーステータス (1) で終了します。

この機能はいまのところ sys.exit を利用していて、これにより SystemExit を発生させます。したがって、except SystemExit もしくは類似のものを利用して *abort* への内部の呼び出しから検知、回復することが可能です。

fabric.utils.**error**(*message*, *func=None*, *exception=None*, *stdout=None*, *stderr=None*)

与えられたエラー message とともに func を呼び出します。

func が None (デフォルト) のとき、env.warn_only の値が abort もしくは warn のどちらを呼び出すかを決定します。

exception が与えられた場合、文字列メッセージを得るために点検され、ユーザーが生成した message とともに表示されます。

stdout および/もしくは stderr が与えられた場合、それらは表示するための文字列であるとみなされます。

fabric.utils.**fastprint**(*text*, *show_prefix=False*, *end=''*, *flush=True*)

text を直ちに、どのようなプリフィックスや行末もなしで表示します。

この機能は異なるデフォルト値を持つ *puts* の単純なエイリアスで、例えば text は一切の装飾なしで表示され、直ちにフラッシュされます。

表示するときに Python の出力バッファリングによってバッファされてしまうかもしれないテキストを表示させたいときに役に立ちます (プロセッサに集中する for ループなど)。そのようなユースケースではたいていは行末なしを要求されるので (進捗を示すドットの連なりなど)、デフォルトでは伝統的な改行を含めません。

注釈: *fastprint* は *puts* を呼び出すので、同様に user *output level* に依存します。

バージョン 0.9.2 で追加.

参考:

puts

fabric.utils.**indent**(*text*, *spaces=4*, *strip=False*)

与えられたスペースの数でインデントされた text を返します。

テキストが文字列ではない場合、行のリストとしてみなされ、インデントの前に \n によって結合されます。

strip が True の時、与えられた文字列の左側から最小限の空白スペースが取り除かれます (そのため、相対的なインデントは維持されますが、その他は左側が取り除かれます)。これにより、入力によっては、それまでのインデントを実質的にすべて "正規化" されます。

fabric.utils.**isatty**(*stream*)

Check if a stream is a tty.

Not all file-like objects implement the *isatty* method.

fabric.utils.**puts**(*text*, *show_prefix=None*, *end='\n'*, *flush=False*)

print のエイリアスで、その出力は Fabric の出力コントロールによって管理されます。

つまり、この関数は単に ‘‘ sys.stdout‘‘ に出力しますが、その ‘‘ user‘‘ の *output level* が False に設定されている場合、その出力を非表示にします。

‘‘sâĂŃâĂŃhow_prefix = False‘‘ の場合、 *puts* はデフォルトで付加される先頭の [hostname] を省略します。(env.host_string が空の場合にもこの接頭を省略します)

改行は end に空の文字列 ('') をセットすることで無効にすることができます。(これは意図的に Python の 3 の print 構文を反映させてものです)

flush=True をセットすることで強制的に出力をフラッシュすることができます (例えば出力のバッファリングをバイパスするため)。

バージョン 0.9.2 で追加.

参考:

fastprint

fabric.utils.**warn**(*msg*)

警告メッセージを出力しますが、実行は中止されません。

この機能は Fabric の *output controls* を優先し、与えられた msg を標準エラー出力に出力され、 warnings 出力レベル (デフォルトで有効) がオンで提供されます、

3.2 Contrib API

Fabric の **contrib** パッケージにはユーザー I/O、リモートファイルの修正などのタスクようにに一般的で便利なツールが含まれています (たいていはユーザーの fabfile からマージされています)。core の API が小さくサイズに保たれ比較的長期間にわたって変更されないのに対し、この contrib セクションはユーザーケースが解決されて追加されるごとに成長し、(後方互換性は保つようにしてはいますが) 進化します。

3.2.1 コンソール出力ユーティリティ

3.2.2 Django との統合

バージョン 0.9.2 で追加.

これらの関数は Django のモジュール環境変数の初期化プロセスを簡素化します。ひとたび行われると、`manage.py` プラグインの利用を要求されることなく、もしくは自分の fabfile を使うごとに環境変数を設定することなく、fabfile が Django プロジェクトもしくは Django 自身からインポートします。

いまのところこれらの関数は、Django インストールのご自分のローカルの fabfile とのやり取りを Fabric ができるようにしているだけです。これはそれほど限定的なものではありません。例えば、ローカルでの利用と同じようにリモートの"build" ツールとして Fabric を利用することも可能です。次の fabfile をご覧ください:

```
from fabric.api import run, local, hosts, cd
from fabric.contrib import django

django.project('myproject')
from myproject.myapp.models import MyModel

def print_instances():
    for instance in MyModel.objects.all():
        print(instance)

@hosts('production-server')
def print_production_instances():
    with cd('/path/to/myproject'):
        run('fab print_instances')
```

Fabric がローカルとリモートの両方にインストールされていれば、`print_production_instances` をローカルで実行すると本番サーバー上の `print_instances` を実行します。そして、これは本番の Django データベースとやり取りを行います。

別の例として、ローカルとリモートのセッティングが似ている場合、これを利用して、例えばデータベースのセッティングを取得し、それをリモート (非 Fabric の) コマンド実行時に利用することも可能です。こうすれば、Fabric がローカルにしかインストールされていなくてもある程度の自由を得ることができます。

```
from fabric.api import run
from fabric.contrib import django

django.settings_module('myproject.settings')
from django.conf import settings

def dump_production_database():
    run('mysqldump -u %s -p=%s %s > /tmp/prod-db.sql' % (
        settings.DATABASE_USER,
```

(次のページに続く)

(前のページからの続き)

```
        settings.DATABASE_PASSWORD,
        settings.DATABASE_NAME
    ))
```

上のスニペットは、ローカルの開発環境からの起動で動作し、データベースの接続情報に関して提供されたローカルの `settings.py` がリモートの設定を反映します。

`fabric.contrib.django.`**`project`**(*name*)

`DJANGO_SETTINGS_MODULE` を `'<name>.settings'` にセットします。

この機能は、設定ファイルやその場所に Django のデフォルト命名規約を利用している場合に、よくあるケースの便利なショートカットを提供します。

`settings_module` を利用します – なぜ、そしてどのようにこの機能を利用するのかについての詳細はそのドキュメントをご覧ください。

`fabric.contrib.django.`**`settings_module`**(*module*)

`DJANGO_SETTINGS_MODULE` のシェル環境変数を `module` にセットします。

シェル環境変数 `DJANGO_SETTINGS_MODULE` を正しく設定しない限り、Django の仕組みにより、Django もしくは Django プロジェクトからのインポートは、失敗します (the Django settings docs を参照してください)。

この関数はそのようにするショートカットを提供します。fabfile もしくは Fabric を利用しているコードの一番うえのほうで呼び出すと、それ以降のどこでも Django のインポートは正しく動作します。

注釈: この関数は **シェル** 環境変数を (`os.environ` 経由で) セットしますが、Fabric 自身の内部 "env" 変数とは関連しません。

3.2.3 ファイルとディレクトリ管理

3.2.4 プロジェクトツール

コア以外の便利な機能。例えば複数オペレーションを構成する機能。

`fabric.contrib.project.`**`rsync_project`**(**args*, ***kwargs*)

rsync 経由でリモートディレクトリとカレントのプロジェクトディレクトリを同期します。

`upload_project()` は実行されるたびに `scp` を利用してプロジェクト全体をコピーしますが、`rsync_project()` は `rsync` を利用し、リモート側のファイルよりも新しいファイルのみを転送します。

rsync_project() は rsync の単純なラッパーです。 rsync の動作に関する詳細はその man ページをご覧ください。この操作が正しく動作するためにはローカルとリモートシステムの両方に rsync がインストールされている必要があります。

この機能は Fabric の local() オペレーションを利用し、その関数呼び出しの出力を返します。つまり、(もしあれば) rsync の呼び出し結果の標準出力が返ってきます。

rsync_project() は次のパラメータを取ります:

- remote_dir: 唯一の必須のパラメータで、リモートサーバー上のディレクトリへのパスです。 rsync がそのように実装されているため、その挙動は local_dir の値に依存します:
  - local_dir が末尾のスラッシュで終わっている場合、ファイルは remote_dir 内にアップロードされます。例えば、 rsync_project("/home/username/project/", "foldername/") は /home/username/project 内の foldername のコンテンツとなります。
  - local_dir が **スラッシュで終わっていない** 場合 (これにはデフォルトのシナリオの local_dir が指定されていない場合も含まれます)、remote_dir は実質的に "親" ディレクトリとなり、その中に local_dir の名称の新しいディレクトリが作成されます。したがって、rsync_project("/home/username", "foldername") は (必要であれば) 新しいディレクトリ /home/username/foldername を作り、ファイルをそこにコピーします。
- local_dir: デフォルトでは、rsync_project はカレントのワーキングディレクトリをソースディレクトリとして利用します。これは local_dir を指定することでオーバーライドすることができます。 local_dir は rsync にそのまま渡される文字列で、単一のディレクトリ ("my_directory") もしくは複数のディレクトリ ("dir1 dir2") とすることが可能です。詳細は rsync のドキュメントをご覧ください。
- exclude: これはオプションで、単一の文字列、もしくは文字列の繰り返し可能オブジェクトとすることが可能で、単一もしくは複数の --exclude オプションとして rsync に渡されるために利用されます。
- delete: rsync の --delete オプションを使用するかどうかをコントロールする真偽値です。True なら、 rsync に指示してローカルには存在しなくなったファイルをリモート側でも削除します。デフォルトでは False になっています。
- extra_opts: オプションで、カスタムな引き数やオプションを rsync に渡すための任意の文字列です。
- ssh_opts: extra_opts と似ていますが、SSH のオプション文字列 (rsync の --rsh フラグ) 用です。
- capture: *local* 呼び出しの内部に直接送られます。
- upload: ファイル同期の実行をアップストリームで行うのかダウンストリームで行うのかをコントロールする真偽値です。デフォルトはアップストリームです。

- default_opts: デフォルトの rsync のオプションの -pthrvz で、必要に応じてオーバライドします (詳細表示 (verbosity) をとりのぞくなど)。

さらに、この関数は Fabric のポートと SSH キー設定を透過的に優先します。カレントホストの文字列に非標準のポートが含まれている場合や env.key_filename が空ではない場合にこの関数を呼び出すと、その指定したポート及び/もしくは SSH 鍵ファイル名 (複数可) を使用します。

参考までに、この関数によって構成される rsync コマンドラインの概要は次のようになっています:

```
rsync [--delete] [--exclude exclude[0][, --exclude[1][, ...]]] \
    [default_opts] [extra_opts] <local_dir> <host_string>:<remote_dir>
```

バージョン 1.4.0 で追加: ssh_opts キーワード引き数。

バージョン 1.4.1 で追加: capture キーワード引き数。

バージョン 1.8.0 で追加: default_opts キーワード引き数。

fabric.contrib.project.**upload_project**(*local_dir=None*, *remote_dir=''*, *use_sudo=False*)

カレントのプロジェクトを tar/gzip 経由でリモートシステムにアップロードします。

local_dir はアップロードするローカルのプロジェクトディレクトリを指定します。デフォルトはカレントのワーキングディレクトリです。

remote_dir アップロード先のターゲットディレクトリを指定します (つまり、remote_dir のサブディレクトリとして local_dir のコピーができます)。デフォルトはリモートユーザーのホームディレクトリです。

use_sudo はリモートでコマンドを実行するさいの使用するメソッドを指定します。use_sudo が True のときは sudo が使用され、その他は run が使用されます。

この関数は tar と gzip のプログラム/ライブラリを利用します。そのため、Win32 システムでは Cygwin もしくは類似のものを使用しない限りうまく動作しないでしょう。実行後、たとえ失敗に終わっても、ローカルとリモートの tar ファイルの削除を試みます。

バージョン 1.1 で変更: local_dir と remote_dir キーワード引き数を追加しました。

バージョン 1.7 で変更: use_sudo キーワード引き数を追加しました。

Python モジュール索引

f

索引