



一度作ったものは二度と作らない。
効率的なプログラミングをおこなうための技術

JCGS 痴山紘史

- プロダクションにパイプラインシステムを提供
- オフィスは設けず、全てオンラインで開発
- 成果物を複数社に提供
- フィードバックを得ながら改良を繰り返す
- システムの開発を始めて大体4年

コードが腐るとモロに影響を受ける

それはさておき

資料は丸っと全て公開中



今年、スライドを撮影する人多くないですか？

そんなあなたに!!

GitHub にて資料を丸っと公開中です

<https://github.com/JCGS/CEDEC2014>

ということで本題

- コード品質の維持
- デバッグ
- 処理の最適化

- コード品質の維持
- デバッグ
- 処理の最適化

データコンバートあるある



- テクスチャのパスをかきかえる
- キャッシュのパスをかきかえる
- レンダリング先を指定する

やっていることは同じ”パスの設定をする”
でも、それぞれ微妙に方法が違う

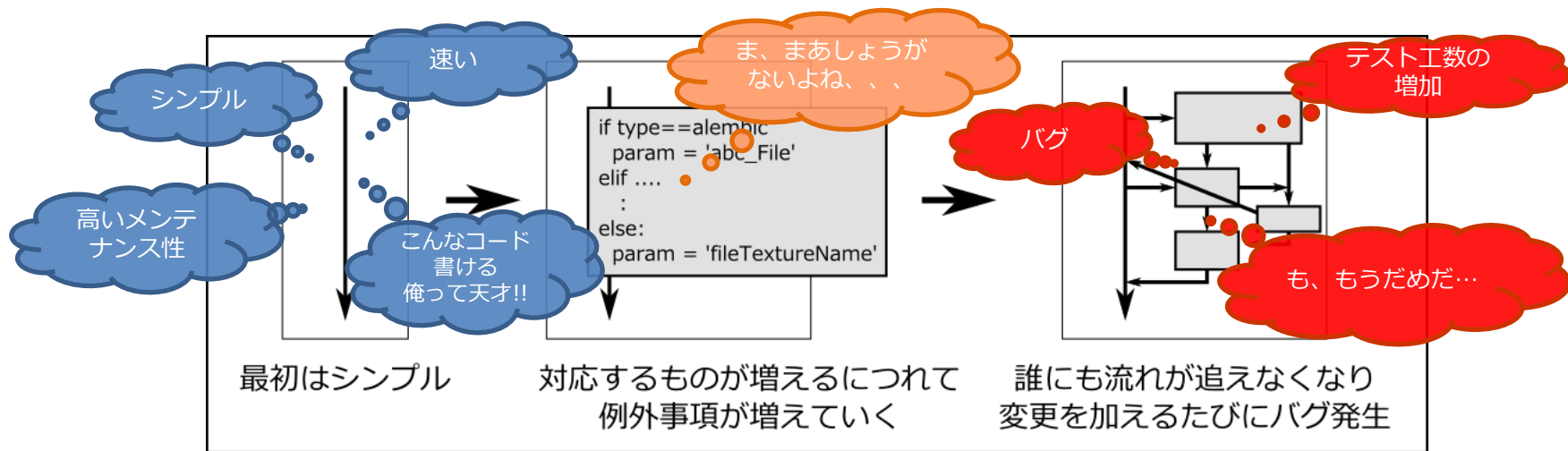
似たような処理の実装



```
cmds.setAttr(node + '.fileTextureName', path, type='string')  
cmds.setAttr(node + '.Path', path.replace(os.sep, '/'), type='string')  
cmds.setAttr(node + '.cacheFileName', path, type='string')  
cmds.setAttr('defaultRenderGlobals.imageFilePrefix', prefix, type='string')
```

- みんなそれぞれ微妙にやり方が違う
- 対応する形式が増えるとパターンが増える
- 複雑な挙動のプラグインとか
 - FumeFXなんて、モードがあるんですよ？

コードのカオス化



- 例外事項への対応ですぐにコードはカオス化する
- 誰もメンテナンスできない→一から書き直そうという不毛なループへ

どうやって対応する？



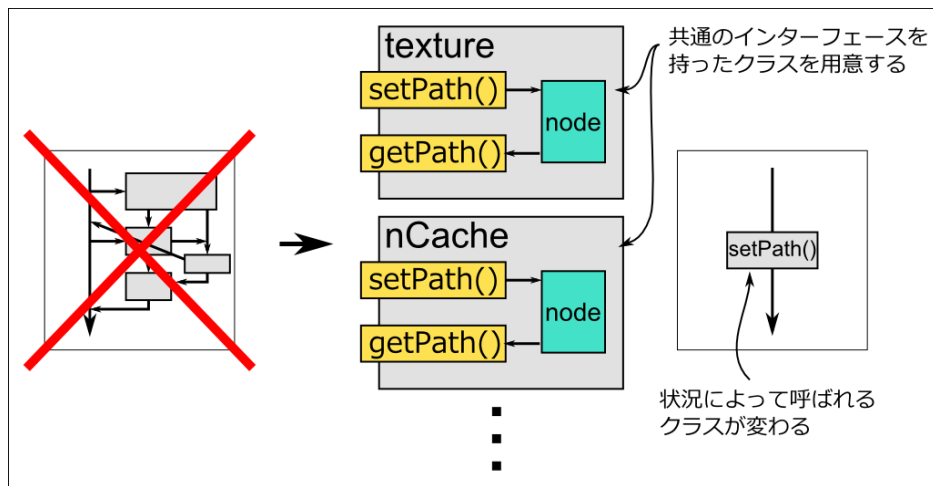
- 問題を簡単で小さな単位におしこめる
- コードの依存関係を極力排除する
- 複雑さが一定水準に保たれるようにする

そんなことを言っても。。。。



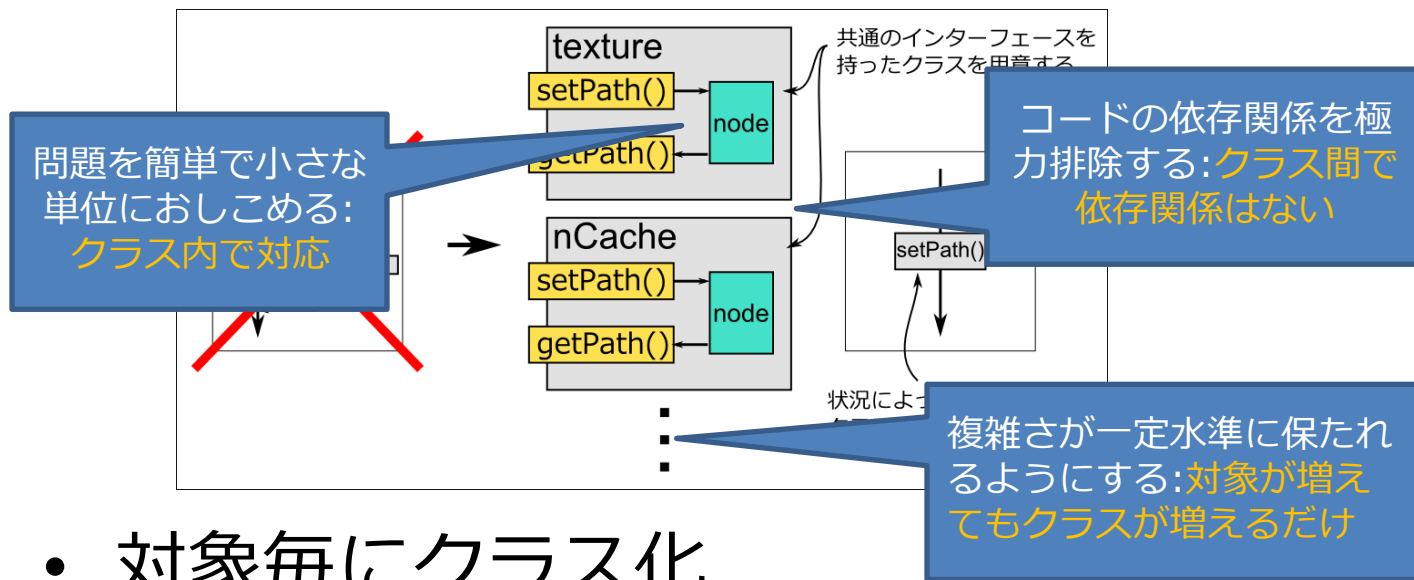
- 全部微妙に違うし。。。。
- 対応しないといけないものは山盛り
- 新しいプラグインが出たらどうするの？
- バグを踏んだら例外処理しないと

共通のインターフェースを定義



- 対象毎にクラス化
- 外からは共通のインターフェース
- 内部の処理を対象毎に実装する

共通のインターフェースを定義



- 対象毎にクラス化
- 外からは共通のインターフェース
- 内部の処理を対象毎に実装する

コード例



```
class nodeBase(object):
    def __init__(self, node, prop=None):
        self.node = node

    def getPath(self):
        return None

    def setPath(self, path):
        return path
```

```
class texture(nodeBase):
    def getPath(self):
        path = cmds.getAttr(self.node + '.fileTextureName')
        if path == None:
            return ""

        return path.replace('/', os.sep)

    def setPath(self, path):
        cmds.setAttr(self.node + '.fileTextureName',
                     path.replace(os.sep, '/'), type='string')

        return self.getPath()
```

```
class nCache(nodeBase):
    def getPath(self):
        cacheName = cmds.getAttr(self.node + r'.cacheName')
        path = cmds.getAttr(self.node + r'.cachePath')
        if path == None:
            path = os.path.join(getCurrentWorkspace(), 'data',
                                cacheName)

        return os.path.join(path, cacheName+'.xml')

    def setPath(self, path):
        d,f = os.path.split(path)
        cacheName, ext = os.path.splitext(f)

        cmds.setAttr(self.node + '.cachePath',
                     d.replace(os.sep, '/'), type='string')
        cmds.setAttr(self.node + '.cacheName', cacheName,
                     type='string')
```

- 大したことをしているわけではない
- ちょっとした工夫でコードを健康な状態に保つことができるという例

コードの品質を維持するために



- 処理をそのままベタ書きしない
- プログラミング言語の力を利用する
- やりすぎるとメンテナンスできないコードになるのでやりすぎはダメ

- コード品質の維持
- デバッグ
- 処理の最適化

バグが出た時の対応



- バグの原因の特定
- printf デバッグに頼っていませんか?
- 勘で適当に print を埋め込んで実行
- 変数を表示しまくって山のようなログが流れるとか

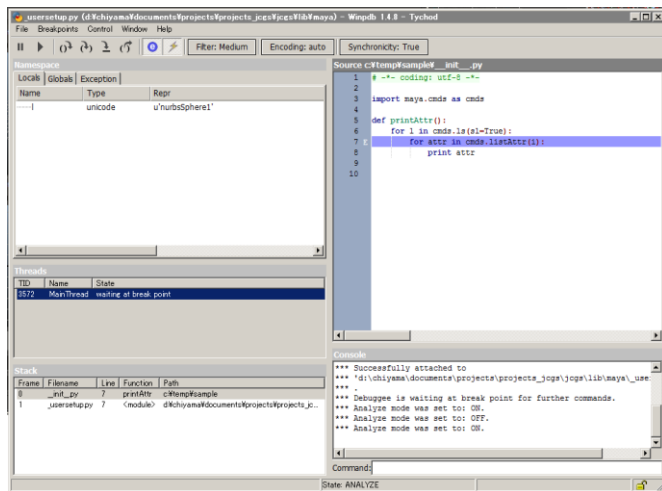
デバッガの活用



- デバッガを使いましょう
- とりあえず殺してその時の状態を見る
- Python だったら pdb
- GUI が必要なら WinPDB とか Eclipse とかいろいろあります
- 今回は WinPDB の紹介

WinPDB

- シンプル
- デバッガに特化
- リモートデバッグにも対応
- DCCTool上のスクリプトも手軽にデバッグできる
- 機能的にはちょっと足りないところも



- アプリケーションとデバッガ間で通信して動作を確認する
- DCCTool でも使用可能
- デバッグを開始したい場所に以下のコードを埋め込んで実行

```
import rpdb2  
rpdb2.start_embedded_debugger('password')
```

リモートデバッグの例

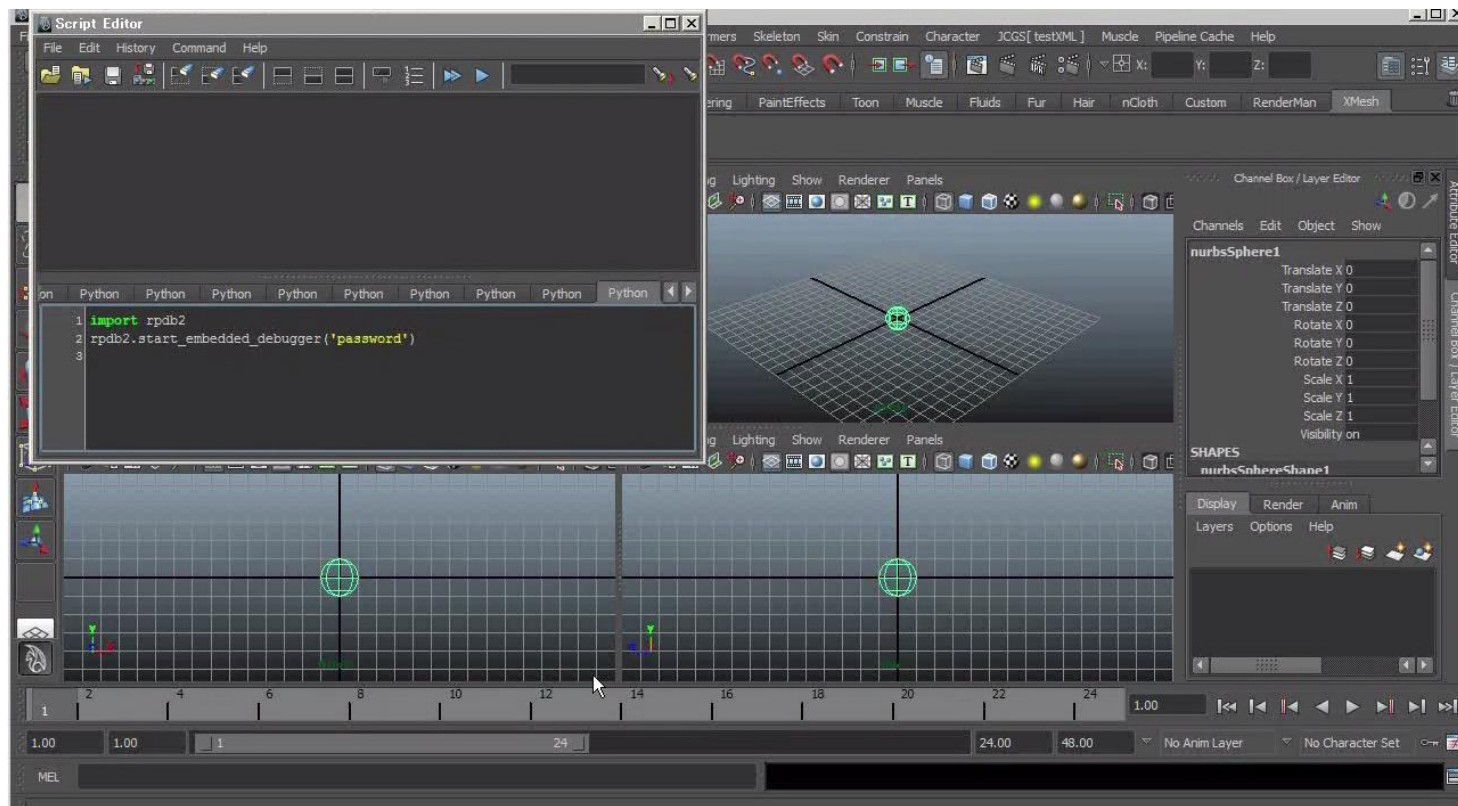


- Maya上で動作するスクリプトをデバッグする
- 例:選択しているオブジェクトのアトリビュートを一覧する

```
import maya.cmds as cmds

def printAttr():
    for l in cmds.ls(sl=True):
        for attr in cmds.listAttr(l):
            print attr
```

デモ



- コード品質の維持
- デバッグ
- 処理の最適化

パフォーマンスチェック



- ツールが遅いという報告が上がった場合
どうしていますか？
- 無暗にコードを改変していませんか？
 - ファイルアクセスが遅い→メモリにキャッシュ
 - 小手先のテクニックでコードの最適化
- その改変、本当に意味ありますか？
 - 勘でとりあえずエイヤツと変更

その改変、効果ありますか？



- ボトルネックを把握しましょう
- どうやって？
- プロファイラを使用する
 - プログラムの実行状況を計測する
 - プログラムのどの部分が何回呼ばれているか
 - プログラムのどこで時間がかかっているか

- Python では cProfile を使用する

```
import cProfile
cProfile.run('someHeavyOperation()') #処理時間を計測
```

- someHeavyOperation() を行った際にどの関数でどれだけ処理に時間がかかったかを把握することができる

プロファイリング結果



```
D:\¥chiyama>python testDirectoryDefs.py
11190417 function calls (11057225 primitive calls) in 28.010 CPU seconds
```

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.000	0.000	<string>:1(<module>)
84	0.003	0.000	27.492	0.327	DirectoryDefs.py:1033(filterNames)
36	0.000	0.000	0.000	0.000	DirectoryDefs.py:1066(Get)
261	0.002	0.000	0.006	0.000	DirectoryDefs.py:121(getChild)
855	0.004	0.000	0.004	0.000	DirectoryDefs.py:132(_getSomeChildren)
285	0.000	0.000	0.002	0.000	DirectoryDefs.py:157(getRequiredChildren)
285	0.000	0.000	0.001	0.000	DirectoryDefs.py:161(getOptionalChildren)
285	0.000	0.000	0.001	0.000	DirectoryDefs.py:165(getUnknownChildren)
12	0.000	0.000	0.001	0.000	DirectoryDefs.py:169(getChildren)
261	0.001	0.000	0.001	0.000	DirectoryDefs.py:206(_addElement)
24	0.004	0.000	27.544	1.148	DirectoryDefs.py:219(addChildren)
(中略)					
744	0.003	0.000	25.210	0.034	Project.py:349(getAbsPathByNode)
732	0.024	0.000	24.842	0.034	Project.py:353(getAbsPath)
98256/16376	1.432	0.000	22.201	0.001	Project.py:90(getInfo)
32796	0.278	0.000	0.278	0.000	ProjectInfoManager.py:132(Get)
32796	0.599	0.000	0.600	0.000	ProjectInfoManager.py:64(getProject)

プロファイリング結果の分析



```
D:\¥chiyama>python testDirectoryDefs.py
11190417 function calls (11057225 primitive calls) in 28.010 CPU seconds
```

Ordered by: standard name

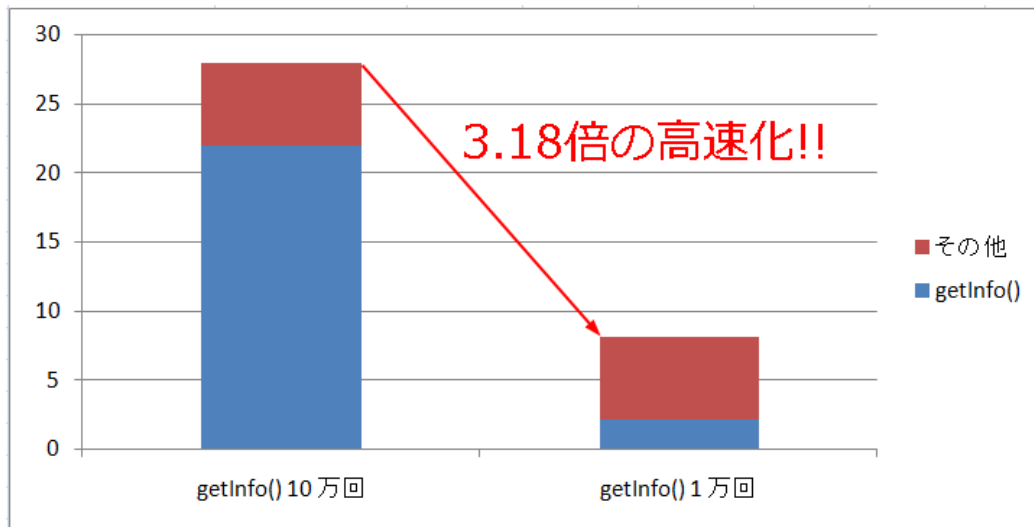
ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.000	0.000	<string>:1(<module>)
84	0.003	0.000	27.492	0.327	DirectoryDefs.py:1033(filterNames)
36	0.000	0.000	0.000	0.000	DirectoryDefs.py:1066(Get)
261	0.002	0.000	0.006	0.000	DirectoryDefs.py:121(getChild)
855	0.004	0.000	0.004	0.000	DirectoryDefs.py:132(_getSomeChildren)
285	0.000	0.000	0.002	0.000	DirectoryDefs.py:157(getRequiredChildren)
285	0.000	0.000	0.001	0.000	DirectoryDefs.py:161(getOptionalChildren)
285	0.000	0.000	0.001	0.000	DirectoryDefs.py:165(getUnknownChildren)
1	0.000	0.000	0.000	0.000	DirectoryDefs.py:169(getChildren)
1	0.000	0.000	0.000	0.000	DirectoryDefs.py:206(_addElement)
1	0.000	0.000	1.148	1.148	DirectoryDefs.py:219(addChildren)
1	0.000	0.000	0.000	0.000	Project.py:349(getAbsolutePath)
1	0.003	0.000	25.210	0.034	Project.py:349(getAbsolutePath)
1	0.024	0.000	24.842	0.034	Project.py:353(getAbsPath)
98256/16376	1.432	0.000	22.201	0.001	Project.py:90(getInfo)
32796	0.278	0.000	0.278	0.000	ProjectInfoManager.py:132(Get)
32796	0.599	0.000	0.600	0.000	ProjectInfoManager.py:64(getProject)

処理全体で28秒かかっている

再帰呼び出しも含めて
10 万回呼ばれている

getInfo()で22秒(78%)
かかっている

最適化戦略を立てる



- getInfo() の呼び出しを1/10 に減らすことができればこの部分は 2.2 秒となる
- 全体では 28 秒→ 8.8秒と、3.18倍の高速化

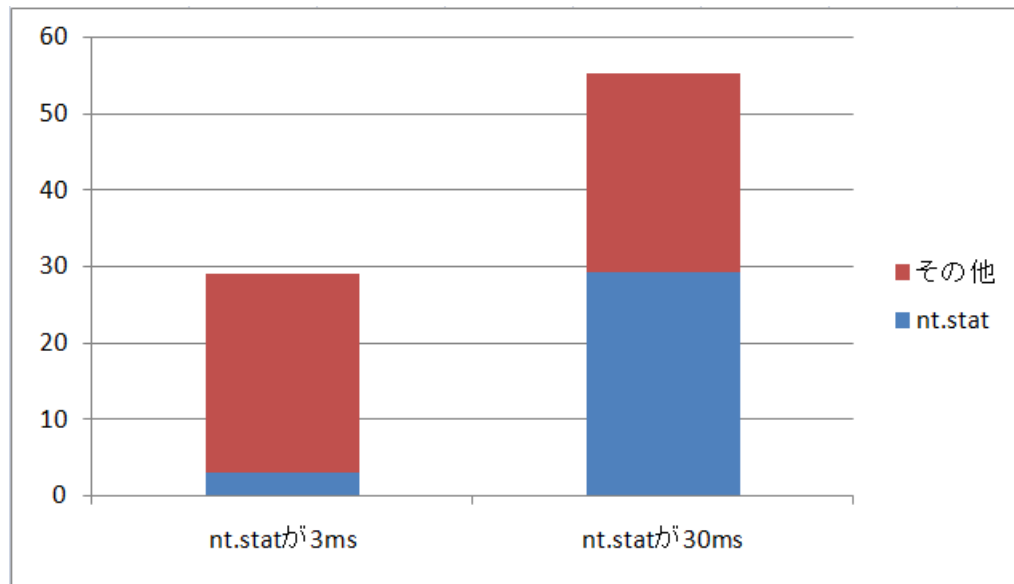
修正前後で比較する



- 修正後もきちんとプロファイリングをおこなう
- 効果が出ていれば成功
- ユーザーに対しても“XX倍速くなりましたよ(ドヤ)”と言える←**これ、結構大事**
- 効果が出ていなかったら単なるおまじないを唱えたただけなので失敗

- ボトルネックは以下の傾向がある
 - 一回の呼び出しに時間がかかっている
 - 一回の呼び出しは大したコストではないが、大量に呼び出されている
 - ディスクやネットワークの I/O
- こういった部分を見つけて、優先的に最適化を行っていくことが大事

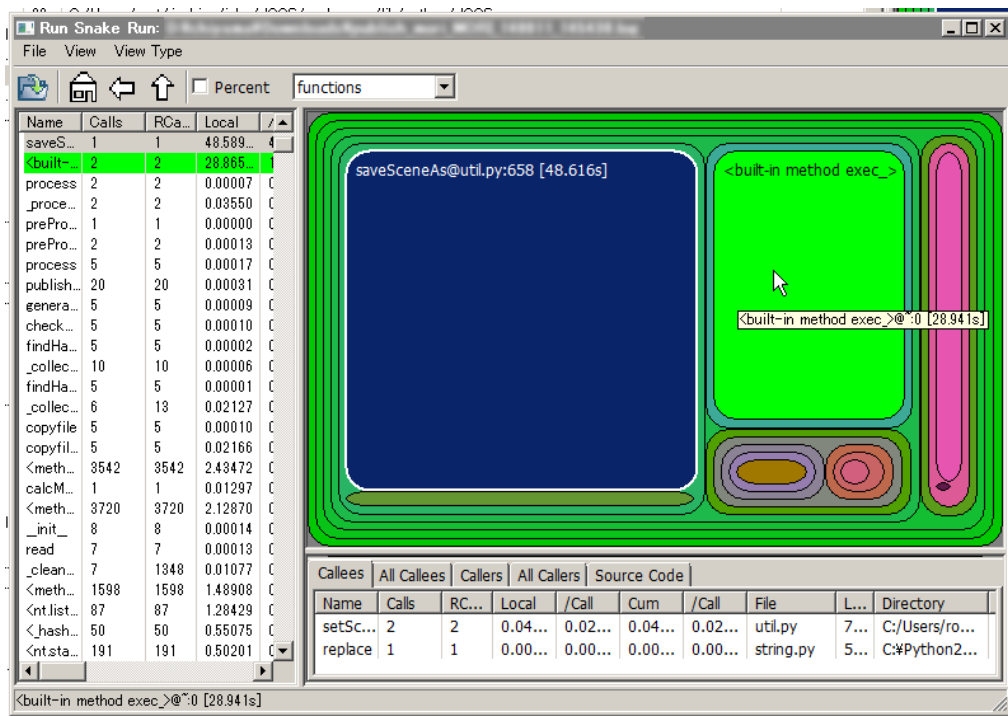
環境によるボトルネックの変化



- ボトルネックは、環境によっても変化する
- 実際の環境できちんと計測することが大事

**コードをこねくり回す前に
アルゴリズムはきちんと精査しましょう**

プロファイラログ確認ツール



Run Snake Run

- プロファイラとデバッガを一緒に使うとエラーになるので注意
- 最初、何が起きているのかわからなかった

混ぜるな危険

以上、三つ



- コード品質の維持
- デバッグ
- 処理の最適化

ご静聴ありがとうございました



Q&A