简介

Hypothesis是一个用于创建单元测试的Python库，使用这个库可以使单元测试编写起来更简单，运行时功能更强大，并且可以在代码中查找一些不易察觉的边界情况。使用这个库编写的单元测试是稳定的、强大的，并且很容易添加到任何现有的测试套件中。

Hypothesis 的工作原理是让您编写**断言某些东西应该适用于所有情况，而不仅仅是你能够想到的那些情况的测试**。

可以将普通的单元测试想象成如下情况：

1. 设置一些数据。
2. 对数据进行一些操作。
3. 对结果进行断言。

Hypothesis 可以让你写出这样的测试用例：

1. 对于所有匹配某些规范的数据。
2. 对这些数据进行一些操作。
3. 对结果进行断言。

这种测试通常被称为基于特性的测试（property-based testing），简称 PBT，并因Haskell库 Quickcheck 而流行起来。

基于特性的测试的一般流程为：

1. 生成符合特定规范的任意数据，称为 "示例"；
2. 然后使用生成的这些示例运行测试，同时检查测试代码中的断言是否仍然成功；
3. 如果找到了使得断言失败的示例，则将该示例缩小并简化，直至找到能够使得断言失败，并且小得多的示例。
4. 然后将该示例保存下来，以便以后发现代码中的问题时不会忘记该示例。

编写这种形式的测试通常应该确定代码应该做出的保证——无论遇到什么情况，这些特性都应该始终正确。这种保证的例子可能是:

- 代码不应该抛出异常，或仅应抛出特定类型的异常（如果有大量内部断言，则效果很好）；
- 如果删除了一个对象，则该对象将不再可见。
- 如果先序列化一个值，然后再将得到的这个值反序列化，则最终得到的值应该与最初的那个值相同。

现在您应该已经对 Hypothesis 的基础知识有了一定了解，本文档的其余部分将带您了解如何以及为什么要使用 Hypothesis。

快速入门指南

本节将介绍开始使用 Hypothesis 所需的一切。

示例

假设我们编写了一个 行程长度压缩算法 系统，并且想要对其进行测试。

以下代码来自 Rosetta Code Wiki，（仅删除了一些注释掉的代码并修复了格式，但是没有功能上的修改）：

```
def encode(input_string):
    count = 1
    prev = ""
    lst = []
    for character in input_string:
        if character != prev:
            if prev:
                entry = (prev, count)
                lst.append(entry)
            count = 1
            prev = character
        else:
            count += 1
    entry = (character, count)
    lst.append(entry)
    return lst


def decode(lst):
    q = ""
    for character, count in lst:
        q += character * count
    return q
```

我们要为上述代码编写测试，以检查这些函数的某些不变量。

当进行这种 编码/解码 时，往往会尝试使用不变式，即如果先对某些内容进行编码然后再解码，那么将得回相同的值。 让我们看看如何通过 Hypothesis 来做：

```
from hypothesis import given
from hypothesis.strategies import text


@given(text())
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```

（在此示例中，我们仅让pytest发现并运行测试。我们稍后将介绍运行该测试的其他方式）。

`text()` 函数返回 Hypothesis 调用的搜索策略。搜索策略是一个具有描述如何生成和简化某些值的方法的对象。 然后， `@given` 装饰器将测试函数转换为一个参数化的函数，当调用该参数化的函数时，将使用 `text()` 返回的搜索策略中的各种匹配数据运行测试函数。

无论如何，此测试会立即在代码中发现一个错误：

```
Falsifying example: test_decode_inverts_encode(s='')

UnboundLocalError: local variable 'character' referenced before assignment
```

Hypothesis 正确地指出，如果输入为一个空字符串，则测试失败。

如果将下面的代码添加到 `encode()` 函数的开头，则 Hypothesis 将会告知我们测试通过（不用执行任何附加操作）。

```
if not input_string:
    return []
```

如果想确保始终都对这种用例进行检查，则可以使用 `@example` 装饰器将其显式添加：

```
from hypothesis import given, example
from hypothesis.strategies import text


@given(text())
@example("")
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```

这对于向其他开发人员（或将来的自己）显示什么样的数据是有效输入非常有用，或者可以确保每次都测试特定的边缘情况（例如 `""`）。这对于回归测试也非常有用，因为尽管 Hypothesis 会记住失败的用例，但我们不建议您分发该数据库。

还需要注意的是，`example` 和 `given` 都支持关键字参数以及位置参数。上述代码的等价写法如下：

```
@given(s=text())
@example(s="")
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```

假设有一个更有趣的bug，即每次都忘记重置计数。将 `encode()` 函数中的第十行代码注释掉：

```
def encode(input_string):
    count = 1
    prev = ""
    lst = []
    for character in input_string:
        if character != prev:
            if prev:
                entry = (prev, count)
                lst.append(entry)
            # count = 1  # 缺失重置操作
            prev = character
        else:
            count += 1
    entry = (character, count)
    lst.append(entry)
    return lst
```

则 Hypothesis 将很快报告下面这样一条导致测试失败的示例：

```
Falsifying example: test_decode_inverts_encode(s='001')
```

请注意，提供的示例确实非常简单。Hypothesis 不仅可以找到测试的反例，而且还知道如何将其简化，从而生成更小更易理解的示例。在上面的示例中，输入字符串 `s` 中相邻的两个相同的字符足以将 `count` 设置为 1 以外的数字，然后另一个不同于前两个相同字符的字符应该导致 `count` 被重置为 1，但在这种情况下 `count` 并没有被重置。

Hypothesis 提供的示例是可以运行的有效Python代码。 您在调用函数时显式提供的任何参数都不是由Hypothesis生成的，如果您显式提供所有参数，则Hypothesis只会调用一次底层函数，而不是对其进行多次调用。

安装

可以使用如下命令安装：

```
pip install hypothesis
```

可以使用下面的命令为 其他拓展 安装依赖，例如：

```
pip install hypothesis[pandas,django]
```

运行测试

在上面的示例中，我们只是让pytest发现并运行测试，但是也可以显式运行测试：

```
if __name__ == "__main__":
    test_decode_inverts_encode()
```

我们也可以将其作为 `unittest.TestCase` 来运行：

```
import unittest


class TestEncoding(unittest.TestCase):
    @given(text())
    def test_decode_inverts_encode(self, s):
        self.assertEqual(decode(encode(s)), s)


if __name__ == "__main__":
    unittest.main()
```

> 技术细节

> 之所以可行，是因为 Hypothesis 忽略了未要求提供的任何参数（位置参数从最右边开始匹配），因此测试方法的 `self` 参数被简单地忽略了并且可以正常使用。 这也意味着 Hypothesis 可以与参数化测试的其他方式很好地配合。 例如，如果某些参数用于pytest夹具而其他参数用于 Hypothesis，则可以很好地工作。

编写测试

Hypothesis 中的测试由两部分组成：

- 一个看起来像您选择的测试框架中的常规测试函数，但带有一些其他参数；
- 以及用于指定如何提供这些参数的 `@given` 装饰器。

以下是一些使用示例：

```python
from hypothesis import given
import hypothesis.strategies as st


@given(st.integers(), st.integers())
def test_ints_are_commutative(x, y):
    assert x + y == y + x


@given(x=st.integers(), y=st.integers())
def test_ints_cancel(x, y):
    assert (x + y) - y == x


@given(st.lists(st.integers()))
def test_reversing_twice_gives_same_list(xs):
    # 将会生成具有任意长度(通常为 0 到 100)的整数列表.
    ys = list(xs)
    ys.reverse()
    ys.reverse()
    assert xs == ys


@given(st.tuples(st.booleans(), st.text()))
def test_look_tuples_work_too(t):
    # 根据提供的类型生成一个元组，并且对应位置上的元素具有对应的类型
    assert len(t) == 2
    assert isinstance(t[0], bool)
    assert isinstance(t[1], str)
```

如上例所述，可以将参数作为位置参数或者关键字参数传递 `@given` 。

如何开始使用 Hypothesis?

您现在应该已经对基础知识有了足够的了解，可以尝试使用Hypothesis为代码编写一些测试。

最好的学习方法是边做边尝试。 如果您对如何在代码中使用这种测试有想法，这里有一些不错的资料：

1. 尝试仅使用适当的任意数据调用函数，看看函数是否崩溃。 您可能会惊讶于这种方法的工作频率。 例如，请注意，我们在编码示例中发现的第一个错误甚至没有触发我们的断言：那个函数之所以崩溃，是因为无法处理我们提供的数据，而不是因为做错了事情。
2. 寻找测试代码中的冗余代码。有没有用不同的测试用例来测试相同的场景的情况？如果有，你能利用Hypothesis将其归纳为单个测试吗？
3. 这篇文章是为 F# 实现所设计的，但仍然是非常好的建议，您可能会发现这些建议有助于为您使用Hypothesis提供帮助。

如果您在入门时遇到任何麻烦，请不要害怕 寻求帮助。

细节和高级特性

本节主要介绍 Hypothesis 的一些不太常用的特性。虽然你可能不会用到，但是有时却很有用。

附加的测试输出

一般情况下，失败的测试的输出看起来是下面这样的：

```
Falsifying example: test_a_thing(x=1, y="foo")
```

其中包含每一个关键字参数的 `repr()` 。

有时，仅有这些是不够的，或者因为有些值的 `__repr__` 方法不是非常具有描述性，或者因为您需要查看测试的一些中间步骤的输出。 这时， `note()` 函数就能排上用场了：

```
hypothesis.note(value)
```

该函数用于在最后的执行中打印 `value` 。

```
>>> from hypothesis import given, note, strategies as st
>>> @given(st.lists(st.integers()), st.randoms())
... def test_shuffle_is_noop(ls, r):
...     ls2 = list(ls)
...     r.shuffle(ls2)
```

```
...     note("Shuffle: %r" % (ls2))
...     assert ls == ls2
...
>>> try:
...     test_shuffle_is_noop()
... except AssertionError:
...     print('ls != ls2')
Falsifying example: test_shuffle_is_noop(ls=[0, 1], r=RandomWithSeed(1))
Shuffle: [1, 0]
ls != ls2
```

`note()` 将在测试的最后运行中打印，以包括您在测试中可能需要的任何其他信息。

测试统计数据

如果您使用的是pytest，则可以通过传递命令行参数 `--hypothesis-show-statistics` 来查看有关已执行测试的大量统计信息。 其中包括有关测试的一些常规统计信息：

例如，如果使用 `--hypothesis-show-statistics` 参数运行下面的代码：

```
from hypothesis import given, strategies as st


@given(st.integers())
def test_integers(i):
    pass
```

则将会看到：

```
- during generate phase (0.06 seconds):
    - Typical runtimes: < 1ms, ~ 47% in data generation
    - 100 passing examples, 0 failing examples, 0 invalid examples
- Stopped because settings.max_examples=100
```

最后的 "Stopped because" 这一行特别要注意：其中的 `settings.max_examples=100` 决定了测试何时应该停止尝试新的示例。这对于理解测试的行为是很有用的。理想情况下，你总是希望这个值是 `max_examples` 。

在某些情况下（如经过过滤的策略和递归策略），你会看到提到的事件描述了数据生成的某些方面：

```
from hypothesis import given, strategies as st


@given(st.integers().filter(lambda x: x % 2 == 0))
def test_even_integers(i):
    pass
```

将会看来类似下面的输出：

```
test_even_integers:

  - during generate phase (0.08 seconds):
      - Typical runtimes: < 1ms, ~ 57% in data generation
      - 100 passing examples, 0 failing examples, 12 invalid examples
      - Events:
        * 51.79%, Retried draw from integers().filter(lambda x: x % 2 == 0) to satisfy filter
        * 10.71%, Aborted test because unable to satisfy integers().filter(lambda x: x % 2 == 0)
  - Stopped because settings.max_examples=100
```

标记自定义事件

```
hypothesis.event(value)
```

该函数用于记录测试中发生的事件。 如果在统计报告模式下运行 Hypothesis，那么关于每个事件的测试运行数量的统计数据将在最后报告。

事件应该是字符串或可转换为字符串的对象。

```
from hypothesis import given, event, strategies as st


@given(st.integers().filter(lambda x: x % 2 == 0))
def test_even_integers(i):
    event("i mod 3 = %d" % (i % 3,))
```

将会看到类似下面这样的输出：

```
test_even_integers:

  - during generate phase (0.09 seconds):
      - Typical runtimes: < 1ms, ~ 59% in data generation
      - 100 passing examples, 0 failing examples, 32 invalid examples
      - Events:
        * 54.55%, Retried draw from integers().filter(lambda x: x % 2 == 0) to satisfy filter
        * 31.06%, i mod 3 = 2
        * 28.79%, i mod 3 = 0
        * 24.24%, Aborted test because unable to satisfy integers().filter(lambda x: x % 2 == 0)
        * 15.91%, i mod 3 = 1
  - Stopped because settings.max_examples=100
```

`event()` 函数的参数可以是任意可哈希类型，但是如果两个事件在用 `str` 转换为字符串后是相同的，则它们将被视为相同事件。

作出假设

有时，Hypothesis 并没有提供你想要的准确类型的数据——虽然大部分的示例是有效的，但是某些示例是无效的，您也不想关心它们。可以通过提前中止测试来忽略此问题，而这会带来意外测试的危险比您想像的要少得多。 另外，花更少的时间在不好的例子上也是一件好事——如果每个测试运行100个例子（默认），结果发现其中70个例子不符合您的需求，那会浪费很多时间。

```
hypothesis.assume(condition)
```

`assume()` 与 `assert` 类似，但是仅会将示例标记为不好的，而不是使测试失败。

这使您可以指定 *假定* 为真的特性，并让 Hypothesis 在将来避免类似的示例。

假如有下面这样的测试：

```
@given(floats())
def test_negation_is_self_inverse(x):
    assert x == -(-x)
```

运行上面的测试将会得到下面的输出：

```
Falsifying example: test_negation_is_self_inverse(x=float('nan'))
AssertionError
```

这很烦人！我们知道 NaN，并且不太在意这个值，但是一旦Hypothesis找到 `NaN` 的例子，Hypothesis就会被 `NaN` 分散注意力并报告 `NaN` 导致测试失败。 但是我们希望测试能够通过。

因此，让我们阻止这个特定的示例：

```
from math import isnan


@given(floats())
def test_negation_is_self_inverse_for_non_nan(x):
    assume(not isnan(x))
    assert x == -(-x)
```

这次测试将顺利通过。

为了避免 Hypothesis 做出比你预想的多得多的假设，当Hypothesis无法找到足够的例子使得假设成立，Hypothesis就会让测试失败。

如果我们写了下面这样的代码：

```
@given(floats())
def test_negation_is_self_inverse_for_non_nan(x):
    assume(False)
    assert x == -(-x)
```

那么在运行的时候，我们将会得到下面的异常：

```
Unsatisfiable: Unable to satisfy assumptions of hypothesis
test_negation_is_self_inverse_for_non_nan. Only 0 examples considered satisfied
assumptions
```

假设足够好吗?

Hypothesis有一种适应性的探索策略，试图避免那些证伪假设的例子，这通常会导致Hypothesis能够在很艰难的情况下找到例子。

假如我们有下面这样的代码：

```
@given(lists(integers()))
def test_sum_is_positive(xs):
    assert sum(xs) > 0
```

不出所料，这个测试失败了，并给出了反例 `[]` 。

向测试函数中添加 `assume(xs)` 将会避免测试在空列表的情况下失败，但仍然会在 `[0]` 失败。

向测试函数中添加 `assume(all(x > 0 for x in xs))` ，则测试将通过：正整数列表的元素之和为正。

之所以令人惊讶，并不是因为没有找到反例，而是找到了足够多的例子。

为了确保正在发生一些有趣的事情，假设我们想对较长的列表进行尝试。 例如，假设我们在其中添加了 `assume(len(xs) > 10)` 。 在这种情况下，基本上是找不到例子的：这个糟糕的策略将发现少于千分之一的例子，因为如果列表中的每个元素都是负数的概率为二分之一，那么要让一个列表恰好包含十个正整数的概率则为 $\frac{1}{2^{10}}$。 在默认配置中，Hypothesis 早在尝试1000个示例之前就放弃了（默认情况下尝试200个示例）。

如果运行下面的代码将会发生什么?

```
@given(lists(integers()))
def test_sum_is_positive(xs):
    assume(len(xs) > 1)
    assume(all(x > 0 for x in xs))
    print(xs)
    assert sum(xs) > 0
```

`test_sum_is_positive()` 中的 `print(xs)` 将打印下面的内容：

```
[17, 12, 7, 13, 11, 3, 6, 9, 8, 11, 47, 27, 1, 31, 1]
[6, 2, 29, 30, 25, 34, 19, 15, 50, 16, 10, 3, 16]
[25, 17, 9, 19, 15, 2, 2, 4, 22, 10, 10, 27, 3, 1, 14, 17, 13, 8, 16, 9, 2, ...]
[17, 65, 78, 1, 8, 29, 2, 79, 28, 18, 39]
[13, 26, 8, 3, 4, 76, 6, 14, 20, 27, 21, 32, 14, 42, 9, 24, 33, 9, 5, 15, ...]
[2, 1, 2, 2, 3, 10, 12, 11, 21, 11, 1, 16]
```

如您所见，Hypothesis 无法找出 *很多* 例子，但仍然可以找到一些。 通常，如果您可以根据测试制定更好的策略，例如 `integers(1, 1000)` 比 `assume(1 <= x <= 1000)` 好很多，但是如果你不能自己定制策略，则 Hypothesis 将为您代劳。

定义策略

`SearchStrategy` 对象用于探索提供给测试函数的示例。 可以使用 `hypothesis.strategies` 模块中公开的函数创建 `SearchStrategy` 对象。

这些策略中的许多都公开了可用于自定义生成的各种参数。 例如，对于整数，可以指定所需整数的 `min` 和 `max` 。 如果您想确切了解策略生成的示例，可以使用 `.example()` 方法：

```
>>> integers(min_value=0, max_value=10).example()
1
```

许多策略是建立在其他策略之上的。 例如，如果要定义一个元组，则需要给出每个元素的类型：

```
>>> from hypothesis.strategies import tuples
>>> tuples(integers(), integers()).example()
(-24597, 12566)
```

更多细节，请参考 自定义策略。

@given 装饰器参数详解

`given()` 函数的签名如下：

```
Hypothesis.given(*_given_arguments, **_given_kwargs)
```

用于将接受参数的测试函数转换为随机测试的装饰器。

该函数是 Hypothesis的主入口点。

`@given` 装饰器可用于指定应对函数的哪些参数进行参数化设置。 您可以使用位置参数或关键字参数，但不能同时使用两者。 例如，以下所有都是有效用法：

```
@given(integers(), integers())
def a(x, y):
    pass
```

```
4
5
6   @given(integers())
7   def b(x, y):
8       pass
9
10
11  @given(y=integers())
12  def c(x, y):
13      pass
14
15
16  @given(x=integers())
17  def d(x, y):
18      pass
19
20
21  @given(x=integers(), y=integers())
22  def e(x, **kwargs):
23      pass
24
25
26  @given(x=integers(), y=integers())
27  def f(x, *args, **kwargs):
28      pass
29
30
31  class SomeTest(TestCase):
32      @given(integers())
33      def test_a_thing(self, x):
34          pass
```

下面的用法则是不正确的：

```
1   @given(integers(), integers(), integers())
2   def g(x, y):
3       pass
4
5
6   @given(integers())
7   def h(x, *args):
8       pass
9
10
11  @given(integers(), x=integers())
```

```
def i(x, y):
    pass


@given()
def j(x, y):
    pass
```

可以使用如下规则来确定 `given` 的使用是否正确：

1. 可以将任何关键字参数传递给 `given`；
2. `given` 的位置参数等效于测试函数最右边的命名参数；
3. 如果底层的测试函数具有可变长位置参数、任意关键字参数或仅关键字参数，则不能使用位置参数。
4. 使用 `given` 测试的函数可能没有任何默认值。

使用 "最右边的的命名参数" 的原因是使 `@given` 可以与实例方法一起工作：`self` 将被正常传递给函数，而不会被参数化。

`@given` 返回的函数的参数，等于原始测试函数的参数减去 `@given` 所填充的参数。

查看 有关框架兼容性的说明，以了解其如何影响您可能正在使用的其他测试库。

生成有针对性的示例

有针对性的基于特性的测试（T-PBT）结合了基于搜索和基于特性的测试的优点。T-PBT并非完全随机，而是使用基于搜索的组件将输入生成引导到更有可能证伪特性的值。与随机PBT相比，T-PBT可以更有效地探索输入空间，并且需要较少的测试来发现错误或在被测系统中获得较高的可信度。 （Löscher and Sagonas）

这并不总是一个好主意——例如，计算搜索指标可能需要花费更多时间来运行更均匀随机（uniformly-random）的测试用例——但是，Hypothesis 对 T-PBT 提供了**实验性**支持。

```
hypothesis.target(observation, *, label='')
```

除了所有常见的启发式方法之外，使用 `int` 或 `float` 类型的 `observation` 参数调用该函数将获得反馈，以指导我们搜索所有会导致错误的输入，以及所有常见的启发式方法。 `observation` 必须始终是有限的。

Hypothesis将通过几个示例来尝试使观察值最大化。只要递增指标是有意义的，那么就可以使用几乎任何指标。例如， `-abs(error)` 是一个随着 `error` 接近于零而增加的指标。

示例指标：

- 集合中的元素数量或队列中的任务数量
- 任务的平均运行时间或最大运行时间（或如果使用 *label*，则两者都可用）

- 数据压缩率（可能是每个算法或每个级别）
- 状态机接受的步骤数

可选的 `label` 参数可用于区分并分别优化不同的观测值，例如数据集的均值和标准差。 每个测试用例多次调用具有任何标签的 `target()` 是错误的。

> **注意**
>
> 运行的示例越多，该技术就越有效。
>
> 根据经验，当 **max_examples=1000** 以上时，定位效果非常明显，并且立即在测试使用的 *每个标签* 上显示约一万个示例。

> **注意**
>
> `hypothesis.target` 是实验性的，在未来的版本中可能会发生根本性的更改甚至删除。如果您觉得它有用，请告诉我们，以便我们可以分享，并再接再厉。

测试统计信息包括每个标签的最佳分数，当最小示例收缩到失败阈值（问题 #2180）时，这可以帮助避免阈值问题。

我们建议用户浏览介绍 T-PBT 的论文；这些论文来自 ISSTA 2017 和 ICST 2018。

Hypothesis 的最初实现通过变异的模糊器进行爬山搜索，并使用一些受模拟退火启发的策略来避免陷入困境以及避免不断变异局部最大值。

自定义函数执行

Hypothesis 提供了一个用于控制其运行示例的方式的钩子。

这样，您就可以为每个示例设置 setup 和 teardown、在子进程中运行示例、将协程测试转换为普通测试等。例如，Django的 `TransactionTestCase` 在单独的数据库事务中运行每个示例。

通过引入执行器（executor）的概念，这种方式得以实现。执行器本质上是一个接受一段代码并运行这段代码的函数。 默认执行器是：

```
def default_executor(function):
    return function()
```

可以通过在类上定义 `execute_example` 方法来定义执行器。该类上使用 `@given` 的任何测试方法都将使用 `self.execute_example` 作为执行器来执行测试。例如，以下执行器将其所有代码运行两次：

```
from unittest import TestCase


class TestTryReallyHard(TestCase):
    @given(integers())
    def test_something(self, i):
        perform_some_unreliable_operation(i)

    def execute_example(self, f):
        f()
        return f()
```

注意：您在 `map` 等方法中使用的函数将在执行器内部运行。也就是说，在您调用传递给 `execute_example` 的函数之前，它们不会被调用。

执行器必须能够处理传递返回 `None` 的函数的情况，否则执行器将无法运行正常的测试用例。 因此，例如以下执行器是无效的：

```
from unittest import TestCase


class TestRunTwice(TestCase):
    def execute_example(self, f):
        return f()()
```

并且必须被重写为：

```
from unittest import TestCase


class TestRunTwice(TestCase):
    def execute_example(self, f):
        result = f()
        if callable(result):
            result = result()
        return result
```

Hypothesis 还提供了可供无法使用 `execute_emaple` 方法的测试运行器（runner）扩展（例如 pytest-trio）使用的替代钩子。不建议最终用户使用这个钩子——最好直接编写一个完整的测试函数，也许可以在应用 `@given` 之前使用装饰器执行相同的转换。

```
@given(x=integers())
@pytest.mark.trio
async def test(x):
    ...


# pytest-trio 插件中的示例代码
test.hypothesis.inner_test = lambda x: trio.run(test, x)
```

但是，对于测试运行器的开发者来说，分配给测试的 `hypothesis` 属性的 `inner_test` 属性将取代内部测试。

> **主意**
>
> 新的 `inner_test` 必须接受并传递原始测试所期望的所有 `*args` 和 `**kwargs` 。

如果最终用户还使用 `execute_example` 方法指定了自定义执行器，则该自定义的执行器以及其他运行时逻辑将应用于测试运行器分配的新的内部测试。

使随机代码具有确定性

尽管Hypothesis的示例生成可用于非确定性测试，但调试具有不确定性的任何东西通常都是非常令人沮丧。 更糟糕的是，我们的示例收缩依赖于每次导致相同失败的相同输入——尽管我们会显示未收缩的失败和一个合适的错误消息（如果未收缩）。

默认情况下，Hypothesis 会为您处理全局 `random` 和 `numpy.random` 随机数生成器，但您也可以注册其他生成器：

```
hypothesis.register_random(r)
```

该函数用于注册由 Hypothesis 管理的 `Random` 实例。

您可以将 `random.Random` 实例（或具有 `seed` 、 `getstate` 和 `setstate` 方法的其他对象）传递到 `register_random(r)` ，以与来自 `random` 和 `numpy.random` 模块中的全局PRNG相同的方式播种和恢复其状态。

所有全局PRNG，例如，模拟器或调度框架，都应该注册，以防止不稳定的测试。 Hypothesis将确保PRNG状态在所有测试运行中均保持一致，或者如果您选择使用 `random_module()` 策略，则可再现地变化。

推断策略

在某些情况下，Hypothesis可以确定您省略参数时的处理方式。 这是基于内省而不是魔术，因此具有明确的限制。

`builds()` 将检查 `target` 的签名（使用 `inspect.getfullargspec()` ）。 如果具有带有类型注释的必需参数，并且未将任何策略传递给 `builds()` ，则使用 `from_type()` 进行填充。您还可以将特殊值 `hypothesis.infer` 作为关键字参数传递，以对带有默认值的参数强制执行此推断。

```
>>> def func(a: int, b: str):
...     return [a, b]
...
>>> builds(func).example()
[-6993, '']
```

`@given` 不会为必需参数执行任何隐式推断，因为这会破坏与 pytest 夹具的兼容性。可以将 `infer` 用作关键字参数，以显式地从其类型注释中填充参数。

```
@given(a=infer)
def test(a: int):
    pass


# is equivalent to
@given(a=integers())
def test(a):
    pass
```

限制

Hypothesis 不会在运行时审查 PEP 484 类型注释。尽管 `from_type` 将照常工作，但仅当您手动创建 `__annotations__` 属性（例如使用 `@annotations(...)` 和 `@returns(...)` ）时， `builds` 和 `@given` 中的推断才起作用。

`typing` 模块是临时的，在Python 3.5.0和3.6.1之间具有许多内部更改，包括次要版本。我们在尽最大努力对其提供支持，但使用旧版本的模块时可能会遇到问题。如果发现问题，请向我们报告，并考虑更新到一个更新的Python版本以解决此问题。

Hypothesis 中的类型注解

如果你安装 Hypothesis 并使用 mypy 0.590+ 或另一个 PEP 561 兼容工具，类型检查器应该自动获取类型提示。

> **注意**
>
> Hypothesis的类型提示可能会在次要发行版之间产生重大变化。
>
> 有关类型提示的上游工具和约定仍在不断变化中——例如，`typing` 模块本身是临时的，而Mypy尚未达到1.0版——我们计划支持该生态系统的最新版本以及可行的旧版本。
>
> 我们可能还会发现更精确的方式来描述各种接口的类型，或者以一种向后兼容的方式一起更改它们的类型和运行时行为。 我们通常会省略弃用的函数或参数的类型提示，以作为其他形式的警告。

在推断 `deferred()` 、 `recursive()` 、 `one_of()` 、 `dictionaries()` 和 `fixed_dictionaries()` 生成的示例的类型时有一些已知的问题。我们将修复这些问题，并随着生态系统的改进，要求相应地使用Mypy的较新版本进行类型提示。

编写下游的类型提示

提供Hypothesis策略 并使用类型提示的项目也可能希望对其策略进行注释。这是受支持的情况，也是尽最大努力在临时项目的基础上提供的。例如：

```
def foo_strategy() -> SearchStrategy[Foo]:
    ...
```

class.**hypothesis.strategies.SearchStragety**

`SearchStrategy` 是所有策略对象的类型。它是一个泛型类型，并且在创建的示例类型中是协变的。 例如：

- `integers()` 是 `SearchStrategy[int]` 类型。
- `lists(integers())` 是 `SearchStragety[List[int]]` 类型。
- 如果 `Dog` 是 `Animal` 的子类型，则 `SearchStrategy[Dog]` 是 `SearchStragety[Animal]` 的子类型。

> ⚠️
>
> `SearchStrategy` **应仅用于类型提示**。请不要从其继承、与其进行比较或将其用于类型提示之外的场景中。构造此类型对象的唯一受支持的方法是使用 `hypothesis.strategies` 模块提供的函数！

Hypothesis 的 pytest 插件

Hypothesis包括一个微型插件，用于改善与pytest的集成，该插件默认情况下处于激活状态（但不会影响其他测试运行器）。它旨在通过提供额外的信息并方便地访问配置选项来改善Hypothesis和Pytest之间的集成。

- `pytest --hypothesis-show-statistic` 可用于 显示测试和数据生成统计数据。
- `pytest --hypothesis-profile=<profile name>` 可用于 加载配置文件。
  `pytest --hypothesis-verbosity=<level name` 可用于 覆盖当前的详细级别。
- `pytest --hypothesis-seed=<an int>` 可用于 使用特定的种子重现失败。

最后，所有使用 Hypothesis定义的测试都将自动应用 `@pytest.mark.hypothesis`。有关使用标记的信息，请点击 这里。

> **注意**
>
> 如果安装了Hypothesis，Pytest将自动加载插件。您完全不需要做任何事情就可以使用。

与外部模糊器配合使用

> **提示**
>
> 想要为团队的本地测试、CI和持续模糊测试提供集成工作流吗?
>
> 使用 HypoFuzz 对整个测试套件进行模糊测试，无需更多测试即可发现更多错误!

有时，您可能希望将传统的模糊器（例如 python-afl，pythonfuzz 或Google的atheris（针对Python和原生扩展））指向您的代码。如果可以将任何 `@given` 测试用作模糊目标而不是手动将字节字符串转换为对象，那不是很好吗?

```
@given(st.text())
def test_foo(s):
    ...


# This is a traditional fuzz target - call it with a bytestring,
# or a binary IO object, and it runs the test once.
fuzz_target = test_foo.hypothesis.fuzz_one_input

# For example:
fuzz_target(b"\x00\x00\x00\x00\x00\x00\x00\x00")
fuzz_target(io.BytesIO(...))
```

根据对 `fuzz_one_input` 的输入，将会发生下面的三件事情：

- 如果字节字符串无效（例如，由于字节字符串太短或过滤器失败，或者因为 `assume()` 太多次），则 `fuzz_one_input` 返回 `None`。
- 如果字节串有效且测试通过，则 `fuzz_one_input` 返回规范化并修剪的缓冲区，该缓冲区将重播（replay）该测试用例。 提供此选项是为了提高变异模糊器（mutating fuzzer）的性能，但可以放心地将其忽略。
- 如果测试失败，例如，引发了一个异常，则 `fuzz_one_input` 会将修剪后的缓冲区添加到Hypothesis示例数据库中，然后重新引发该异常。 要重现、最小化和消除通过模糊检测发现的所有故障，您需要做的所有工作就是运行测试套件！

请注意，输入字节字符串和输出字节字符串的解释都特定于您所使用的Hypothesis的确切版本以及为测试提供的策略，就像 示例数据库 和 `@reproduce_failure` 装饰器一样。

与设置的交互

`fuzz_one_input` 仅使用足够的Hypothesis内部构件来使用模糊器提供的字节字符串驱动测试函数，因此大多数设置在此模式下均无效。 建议在进行模糊测试之前先以通常的方式运行常规测试，以获取运行状况检查的好处，然后重放、缩小、删除重复数据并报告发现的任何错误。

- `database` 设置用于模糊测试模式——将故障添加到要在下次运行测试时重播的数据库是首选报告机制和对 "模糊驯服（fuzzer taming）" 问题 的解决办法。。
- `verbosity` 和 `stateful_step_count` 设置照常工作。

`deadline`、`derandomize`、`max_examples`、`phases`、`print_blob`、`report_multiple_bugs` 和 `suppress_health_check` 设置不影响模糊测试模式。

线程安全策略

正如 问题 #2719 中所讨论的，Hypothesis 不是真正的线程安全的，并且将来不太可能改变。因此，此策略描述了如果您将 Hypothesis 与多个线程一起使用，将会发生什么。

在多个进程中运行测试，例如使用 `pytest -n auto`，是完全支持的，并且我们会定期在CI中对其进行测试——由于进程隔离，我们只需要确保 `DirectoryBasedExampleDatabase` 不会太过自行其道即可。如果您在此处发现错误，我们将尽快修复。

在多个线程中运行单独的测试不是我们设计或测试的目的，并且不受正式支持。也就是说，它确实发挥了很大作用，我们希望它能继续发挥作用——我们接受合理的补丁程序和低优先级的错误报告。这里的主要风险是全局状态、共享缓存和缓存策略。

在一个测试中使用多个线程，或者在多个线程中同时运行一个测试，很容易触发内部错误。除非可读性或单线程性能受到影响，否则我们通常会接受针对此类问题的补丁程序。

Hypothesis假设测试是单线程的，或者很好地假装是单线程的。在内部使用帮助程序线程的测试应该没问题，但是用户必须小心以确保测试结果仍然是确定的。特别是，如果帮助程序线程定时使用 `data()` 改变了动态生成的顺序，则测试结果被视为不确定的。

与帮助程序线程中的任何Hypothesis API进行交互可能会产生 奇怪/不好 的事情，因此也请避免这种情况——我们在某些地方依赖线程局部变量，并且未明确 测试/审核 它们如何响应跨线程 API 调用。尽管 `data()` 和等效项是最明显的危险，但其他API也会受到轻微影响。

设置

Hypothesis 试图为其行为设定良好的默认值，但有时这还不够，您需要对其进行调整。

执行此操作的机制是 `settings` 对象。 您可以使用 `settings` 装饰器将基于 `@given` 的测试设置为使用此设置：

`@given` 调用如下：

```
from hypothesis import given, settings


@given(integers())
@settings(max_examples=500)
def test_this_thoroughly(x):
    pass
```

上面代码中使用了一个 `settings` 对象，该对象导致测试接收的示例集比正常情况大得多。

可以在 `@given` 之前或之后应用 `settings`，并且结果相同。 下面的代码与上面的代码具有完全相同的作用：

```
rom hypothesis import given, settings


@settings(max_examples=500)
@given(integers())
def test_this_thoroughly(x):
    pass
```

可用的设置

class **hypothesis.settings**(*parent=None*, ***, *max_examples=not_set*, *derandomize=not_set*, *database=not_set*, *verbosity=not_set*, *phases=not_set*, *stateful_step_count=not_set*, *report_multiple_bugs=not_set*, *suppress_health_check=not_set*, *deadline=not_set*, *print_blob=not_set*)

设置对象控制伪造中使用的各种参数。这些参数可以控制伪造策略和所生成数据的细节。

默认值从 `settings.default` 对象中获取，并将在新创建的 `settings` 中获取在默认值中所做的更改。

- `database`

  `ExampleDatabase` 的一个实例，该实例用于保存示例，并从中加载以前的示例。可能为 `None`，在这种情况下将不使用任何存储空间。

  有关内置示例数据库实现的列表以及如何定义自定义实现，请参见 示例数据库文档。

  默认值：（动态计算）

- `deadline`

  如果设置，则测试中的每个单独的示例（即每次调用的测试函数，而不是整个被装饰的测试函数）的持续时间（以 `timedelta` 或 整数\浮点数 的毫秒数表示）将不允许超过该值。 持续时间超过该值的耗时较长的测试可能会转换为错误（但如果接近该值，则不一定会出现这种情况，以允许测试运行时间有所变化）。

  将此参数设置为 `None` 可完全禁用此行为。

  默认值：`timedelta(milliseconds=200)`

- `derandomize`

  如果设置为 `True`，则使用测试函数的哈希值作为Hypothesis的随机数生成器的种子，以便每次运行都将测试同一组示例，直到您更新Hypothesis、Python或测试函数。

  这使您可以使用 单独的设置配置文件 来 检查回归并查找错误 ——例如，在每次提交时运行快速确定性测试，以及在夜间进行较长时间的非确定性测试。

  默认值：`False`

- `max_examples`

  一旦运行了这么多令人满意的例子而没有找到任何反例，则伪造将终止。

  选择默认值以适配工作流，在该工作流中，测试将成为在本地或CI服务器上定期执行的套件的一部分，从而在总运行时间与遗漏错误的机会之间取得平衡。

  如果您正在编写一次性测试，则运行数以万计的示例是非常合理的，因为Hypothesis可能会丢失默认设置下的罕见错误。对于非常复杂的代码，我们观察到Hypothesis在测试SymPy时经过数百万个示例后发现了新的错误。 如果您要运行10万个以上的示例，请考虑集成以覆盖为指导的模糊测试——在运行几分钟或几小时的情况下，它确实会发挥很大的作用。

  默认值：`100`

- `phases`

  控制应运行哪些阶段。 请参阅完整的文档以获取更多详细信息。

  默认值：`(Phase.explicit, Phase.reuse, Phase.generate,Phase.target, Phase.shrink)`

- `print_blob`

  如果设置为 `True`，则 Hypothesis 将打印失败示例的代码，该代码可与 `@reproduce_failure` 一起使用以重现失败的示例。如果设置了 `CI` 或 `TF_BUILD` 环境变量，则默认值为 `True`，否则为 `False`。

  默认值：（动态计算）

- `report_multiple_bugs`

由于Hypothesis多次运行测试，因此有时可以在一次运行中发现多个错误。一次报告所有这些错误通常非常有用，但是替换异常有时可能会与调试器发生冲突。如果禁用，则只会引发带有最小示例的异常。

默认值：`True`

- `stateful_step_count`

  在放弃有状态程序之前运行有状态程序的步骤数。

- `suppress_health_check`

  要禁用的 `HealthCheck` 项的列表。

  默认值：`()`

- `verbosity`

  控制 Hypothesis 消息的详细级别。

  默认值：`Verbosity.mormal`

控制要运行哪些示例

假设将测试分为五个不同的逻辑阶段：

1. 运行 由 `@example` 装饰器提供的 显式示例。
2. 重新运行先前失败的示例，以重现先前看到的错误。
3. 生成新的示例。
4. 用于 T-PBT 的变异示例。
5. 试图缩小在先前阶段中找到的示例（阶段1除外——不能缩小显式示例）。这将潜在的大而复杂的示例变成难以阅读的小而简单的例子。

可以使用 `settings` 对象的 *phase* 参数对要运行哪些阶段进行更细粒度的控制，其中每个阶段都对应于 `Phase` 枚举的一个成员：

1. `Phase.explicit`：控制显式示例是否运行。
2. `Phase.reuse`：控制是否重新运行之前的示例。
3. `Phase.generate`：控制是否生成新的示例。
4. `Phase.target`：控制是否将示例进行突变以用于定位。
5. `Phase.shrink`：控制示例是否被缩小。

phase 参数接受 `Phase` 枚举成员的任何子集。例如 `settings(phases=[Phase.generate, Phase.shrink])` 将会生成新的示例并将其缩小，但不会运行显式的示例或重新运行之前的失败示例，而 `settings(phase=[Phase.explicit])` 将仅运行显式的示例。

查看中间结果

要查看Hypothesis运行测试时的情况，您可以打开详细度设置。

```
>>> from hypothesis import find, settings, Verbosity
>>> from hypothesis.strategies import lists, integers
>>> @given(lists(integers()))
```

```
... @settings(verbosity=Verbosity.verbose)
... def f(x):
...     assert not any(x)
... f()
Trying example: []
Falsifying example: [-1198601713, -67, 116, -29578]
Shrunk example to [-1198601713]
Shrunk example to [-1198601600]
Shrunk example to [-1191228800]
Shrunk example to [-8421504]
Shrunk example to [-32896]
Shrunk example to [-128]
Shrunk example to [64]
Shrunk example to [32]
Shrunk example to [16]
Shrunk example to [8]
Shrunk example to [4]
Shrunk example to [3]
Shrunk example to [2]
Shrunk example to [1]
[1]
```

四个级别分别是：`quite`、`normal`、`verbose` 和 `debug`。默认值为 `normal`；在 `quite` 模式下，Hypothesis 将不打印任何输出，即使最后的伪造示例也不会打印。`debug` 模式基本上与 `verbose` 相同，但更复杂一些。

如果您使用的是pytest，则可能还需要禁用通过的测试的输出捕获。

构建 settings 对象

可以通过使用任何可用设置值调用 `settings` 来创建设置。 任何未提供的参数都将被设置为默认值：

```
>>> from hypothesis import settings
>>> settings().max_examples
100
>>> settings(max_examples=10).max_examples
10
```

也可以将一个 "父" 设置对象作为第一个参数传递，并且任何未作为关键字参数传递的设置都将会从该 "父" 设置对象中复制。

默认设置

无论何时，都可以通过 `settings.default` 访问当前的默认设置。除了作为设置对象之外，所有未明确基于其他设置对象创建的新的设置对象均基于该默认设置，因此将从该默认设置继承未显示设置的值。

您可以使用配置文件来更改默认值。

设置配置文件

根据您的环境，您可能需要不同的默认设置。 例如：在开发过程中，您可能希望减少示例数量以加快测试速度。但是，在CI环境中，您可能需要更多示例，因此您更有可能发现错误。

Hypothesis允许您定义不同的设置配置文件。 这些配置文件可以随时加载。

- *static* **settings.register_profile**(*name*, *parent=None*, ***kwargs*)

  注册一个可作为设置配置的值的集合。

  设置配置文件可以按名称加载——例如，您可以创建一个运行较少示例的 "快速" 配置文件，保留 "默认" 配置文件，并创建一个 "CI" 配置文件以增加示例数量并使用其他数据库存储失败的示例。

  此方法的参数与 `settings` 完全相同：可选的 `parent` 设置，以及将与父设置不同的每个设置的关键字参数（如果 `parent` 为 `None`，则为 `settings.default`）。

- *static* **settings.get_profile**(*name*)

  返回具有给定名称的配置设置。

- *static* **settings.load_profile**(*name*)

  加载在提供的配置文件中定义的设置。

  如果配置文件不存在，将引发 `InvalidArgument` 异常。 在配置文件中未定义的任何设置将是该设置的库定义默认值。

加载配置文件会更改默认设置，但不会更改显式更改设置的测试的行为。

```
>>> from hypothesis import settings
>>> settings.register_profile("ci", max_examples=1000)
>>> settings().max_examples
100
>>> settings.load_profile("ci")
>>> settings().max_examples
1000
```

您可以加载特定测试的配置文件，而不是加载配置文件并覆盖默认值。

```
>>> settings.get_profile("ci").max_examples
1000
```

您可以可选地通过定义环境变量来加载配置文件。 这是在CI上运行测试的建议模式。下面的代码应在 `conftest.py` 或测试套件的任何 设置/初始化 部分中运行。 如果未定义此变量，则将加载Hypothesis定义的默认值。

```
>>> import os
>>> from hypothesis import settings, Verbosity
>>> settings.register_profile("ci", max_examples=1000)
>>> settings.register_profile("dev", max_examples=10)
>>> settings.register_profile("debug", max_examples=10, verbosity=Verbosity.verbose)
>>> settings.load_profile(os.getenv(u"HYPOTHESIS_PROFILE", "default"))
```

如果您使用的是Hypothesis 的 pytest插件，并且配置文件已由 `conftest.py` 注册，则可以使用命令行选项 `--hypothesis-profile` 加载配置文件。

```
$ pytest tests --hypothesis-profile <profile-name>
```

数据生成

大多数事物应该易于生成，并且一切皆有可能。

为了支持该原则，Hypothesis为大多数内置类型提供了具有限制或调整输出的参数策略，以及可以组合生成更复杂类型的高阶策略。

本节是有关可用于生成数据的策略以及如何构建它们的指南。 策略具有各种其他重要的内部功能，例如如何简化，但是可以生成的数据是其API的唯一公开部分。

核心策略

用于构建策略的函数均位于 `hypothesis.strategies` 模块。其中一部分函数用于构建 Python 中的内置类型，另一部分用于基于这些 Python 内置类型构建特定的复合类型，下面我们来分别介绍。

用于构建 Python 内置类型的策略

- **binary**(**, min_size=0, max_size=None*)

  生成 `bytes` 。

  生成的字节的最小长度为 *min_size*，最大长度为 *max_size*；如果 *max_size* 为 `None` ，则长度无上限。

  这个策略的示例会收缩到更小的字符串和更低的字节值。

- **booleans**()

  返回生成 `bool` 实例的策略。

这个策略的例子将会向 `False` 收缩（即收缩将在可能的情况下用 `False` 代替 `True`）。

- **characters**(**, whitelist_categories=None, blacklist_categories=None, blacklist_characters=None, min_codepoint=None, max_codepoint=None, whitelist_characters=None*)

  生成遵循特定过滤器规则的长度为 1 的 `str`。

  - 如果没有指定过滤器规则，则生成任意的字符。
  - 如果指定了 `min_codepint` 或 `max_codepoint`，则只会生成代码点在该范围中的字符。
  - 如果指定了 `whitelist_categories`，则只会生成这些 Unicode 类别内的字符。这是进一步的限制，字符还必须满足 `min_codepoint` 和 `max_codepoint`。
  - 如果指定了 `blacklist_categories`，则不会生成这些 Unicode 类别内的字符。`whitelist_categories` 和 `blacklist_categories` 之间的任何重叠部分均会引发一个异常，因为每一个字符只能属于一个 Unicode 类别。
  - 如果指定了 `whitelist_characters`，则还会生成该列表中的字符。
  - 如果指定了 `blacklist_characters`，则不会生成该列表内的字符。`whitelick_characters` 和 `blacklist_characters` 之间的任何重叠部分均会引发一个异常。

  `_codepoint` 参数必须是 0 和 `sys.maxunicode` 之间的整数。`_characters` 参数必须是长度为1的 unicode 字符串的集合，例如一个unicode 字符串。

  `_categories` 参数必须用于指定一个 "one-letter" Unicode主要类别或 "two-letter" Unicode常规类别。 例如，`('Nd', 'Lu')` 表示 "Number, decimal digit" 和 "Letter, uppercase"。 可以给出一个字母（"主要类别"）以匹配所有相应的类别，例如，对于任何标点符号类别中的字符，均使用 `'P'` 表示。

  该策略的示例向代码点为 `'0'` 的字符收缩，如果不包括 `'0'`，则收缩到其后的第一个允许的代码点。

- **complex_numbers**(**, min_magnitude=0, max_magnitude=None*)

  返回生成复数的策略。

- **dates**(*min_value=datetime.date.min, max_value=datetime.date.max*)

  用于生成 `min_value` 和 `max_value` 之间的 `date` 的策略。

  该策略的示例向 "2001年1月1 日" 收缩。

- **datetimes**(*min_value=datetime.datetime.min, max_value=datetime.datetime.max, *, timezones=none(), allow_imaginary=True*)

  用于生成可能是时区敏感（timezone-aware）的 `datetime` 的策略。

  此策略通过在 `min_value` 和 `max_value` 之间生成原始（naive） `datetime` 来工作，其中 `max_value` 和 `min_value` 必须均为原始（naive） `datetime`（没有时区）。

  `timezones` 必须是生成 `None`（用于原始 `datetime`）或 `tzinfo` 对象（用于时区敏感的 `datetime`）的策略。 尽管我们建议使用 dateutil 包和 `hypothesis.extra.dateutil.timezones()` 策略，但您也可以构建自己的策略，并提供 `hypothesis.extra.pytz.timezones()`。

  可以通过传递 `allow_imaginary=False` 来过滤由于夏时制、润秒、时区和日历调整等原因而没有出现（或不会出现）的 "虚构的" `datetime`。默认情况下，"虚构的" `datetime` 是允许的，因为格式错误的时间戳是常见的错误的来源。 请注意，由于 pytz 早于 PEP 495，因此这对于使用负DST偏移量的时区（例如 `"Europe/Dublin"`）无法正常使用。

  该策略的示例向本地时间（local time）2000年1月1日午夜收缩。

- **decimals**(*min_value=None, max_value=None, *allow_nan=None, allow_infinity=None, palces=None*)

生成 `decimal.Decimal` 实例，其可能是：

- `min_value` 和 `max_value` 之间的有限有理数。
- 非数字，如果 `allow_nan` 为 `True`。`None` 表示 "除非 `min_value` 和 `max_value` 不为 `None`，否则允许 `NaN`"。
- 正无穷或负无穷，如果 `max_value` 和 `min_value` 分别为 `None`，而 `allow_infinity` 不为 `False`。`None` 表示 "除非被最小值和最大值排除，否则允许无穷大"。

请注意，当浮点数的值为 `NaN` 时，它可能是下面中的一个：

- `Decimal('NaN')`
- `Decimal('-NaN')`
- `Decimal(sNaN)`
- `Decimal(-sNaN)`

有关特殊值的更多信息，请参见 `decimal` 模块文档。

如果 `places` 不为 `None`，则从该策略得出的所有有限值的小数点后的位数均为 `places` 位。

此策略的示例没有明确定义的收缩顺序，但是在收缩时尝试使人类可读性最大化。

- **dictionaryies**(*keys*, *values*, ***, *dict_class=<class 'dict'>*, *min_size=0*, *max_size=None*)
- **emails**()
- **fixed_dictionaries**(*mapping*, ***, *optional=None*)
- **floats**(*min_value=None*, *max_value=None*, ***, *allow_nan=None*, *allow_infinity=None*, *width=64*, *exclude_min=False*, *exclude_max=False*)
- **fractions**(*min_value=None*, *max_value=None*, ***, *max_denominator=None*)
- **from_regex**(*regex*, ***, *fullmatch=False*)
- **from_type**(*thing*)
- **frozensets**(*elements*, ***, *min_size=0*, *max_size=None*)
- **functions**(***, *like=lambda: ...*, *returns=none()*, *pure=False*)
- **integers**(*min_value=None*, *max_value=None*)
- **ip_addresses**(***, *v=None*, *network=None*)
- **iterables**(*elements*, ***, *min_size=0*, *max_size=None*, *unique_by=None*, *unique=False*)
- **just**(*value*)
- **none**()
- **nothing**()
- **one_of**(**args*)
- **random_module**()
- **randoms**(***, *note_method_calls=False*, *use_true_random=False*)
- **recursive**(*base*, *extend*, ***, *max_leaves=100*)
- **register_type_strategy**(*custom_type*, *strategy*)
- **runner**(***, *default=not_set*)
- **sampled_from**(*elements*)

- **sets**(*elements, *, min_size=0, max_size=None*)
- **shared**(*base, *, key=None*)
- **slice**(*size*)
- **text**(*alphabet=characters(blacklist_categories=('Cs')), *, min_size=0, max_size=None*)
- **timedeltas**(*min_value=datetime.timedelta.min, max_value=datetime.timedelta.max*)
- **times**(*min_value=datetime.time.min, max_value=datetime.time.max, *, timezones=none()*)
- **timezone_keys**(*, allow_prefix=True*)
- **timezones**(*, no_cache=False*)
- **tuples**(**args*)
- **uuid**(*, version=None*)

其他策略

- **builds**(*target, /, *args, **kwargs*)

  通过从 `args` 和 `kwarg` 生成并将它们传递到可调用对象 `target` （作为第一个位置参数提供）来生成值。

  例， `builds(target, integers(), flag=booleans())` 将生成一个整数 `i` 和一个布尔值 `b` 并调用 `target(i, flag=b)` 。

  如果可调用对象具有类型注释，则将使用它们来为未传递到 `builds` 的必需参数推断策略。您还可以通过将特殊值 `hypothesis.infer` 作为关键字参数传递给 `builds` ，而不是将该参数的策略传递给可调用对象来告诉 `builds` 推断可选参数的策略。

  如果可调用对象是使用 attrs 定义的类，则尽量从属性中推断出缺少的必需参数，例如，通过检查 attrs标准验证器。 `dataclass` 通过类型提示的推断来本地处理。

  该策略的示例通过收缩可调用对象的参数值来收缩。

- **composite**(*f*)

  定义一个从可能任意多个其他策略中构建的策略。

  它旨在用作装饰器。有关如何使用此函数的更多详细信息，请参见 复合策略。

  通过收缩每个 `draw` 调用的输出来收缩该策略的示例。

- **data**()

  这实际上不是正常的策略，而是一个可用于交互式地从其他策略中提取数据的对象。

  请参阅文档的 在测试中交互地生成，以获取更完整的信息。

  此策略的示例不会收缩（因为只有一个），但是每次调用 `draw()` 的结果都会像往常一样收缩。

- *class* **DataObject**(*data*)

  这种类型仅用于使用 `data()` 策略为测试编写类型提示。 不要直接使用该对象。

- **deferred**(*definition*)

  延迟策略用于编写引用其他尚未定义的策略的策略。这使得可以轻松定义递归策略和相互递归策略。

  definition 参数应该是一个返回策略的零参数函数。 首次使用该策略生成示例时，将对其进行计算。

  使用示例：

```
>>> import hypothesis.strategies as st
>>> x = st.deferred(lambda: st.booleans() | st.tuples(x, x))
>>> x.example()
(((False, (True, True)), (False, True)), (True, True))
>>> x.example()
True
```

也能定义相互递归的策略：

```
>>> a = st.deferred(lambda: st.booleans() | b)
>>> b = st.deferred(lambda: st.tuples(a, a))
>>> a.example()
True
>>> b.example()
(False, (False, ((False, True), False)))
```

该策略的示例会像 *definition* 返回的策略中的示例一样正常地收缩。

- **from_regex**(*regex, * fullmatch=False*)
- **from_type**(*thing*)

  为给定的类型查找合适的搜索策略。

  `from_type` 在内部用于填充 `builds()` 缺少的参数，并且可以交互地用于探索可用的策略或调试类型解析。

  可以使用 `register_type_strategy()` 处理自定义类型，或全局地重新定义某些策略——例如，从浮点数中排除 `NaN`，或者使用时区敏感而不是时区无关的时间和日期时间策略。

  解析逻辑可能会在将来的版本中更改，但目前尝试使用以下五个选项：

  1. 如果 `thing` 在默认的查找映射或用户注册的查找映射中，返回对应的策略。默认查找涵盖所有具有假设策略的类型，包括可能的其他类型。
  2. 如果 `thing` 来自 `typing` 模块，则返回对应的策略（特殊逻辑）。
  3. 如果 `thing` 在合并的查询中具有一个或多个子类型，则为那些不是查询中其他元素的子类型的类型返回策略的并集。
  4. 最后，如果 `thing` 的所有必需参数都有类型注解并且 `thing` 不是抽象类，则其通过 `builds` 解析。
  5. 因为抽象类不能实例化，因此抽象类型被视为其具体子类的并集。请注意，这种查找是通过继承，而不是通过 `abc.ABCMeta.register`，因此，你仍然需要使用 `register_type_strategy()`。

  `from_type()` 特别有价值的作用是生成指定类型之外的值。例如：

```
def everything_except(excluded_types):
    return (
        from_type(type)
        .flatmap(from_type)
        .filter(lambda x: not isinstance(x, excluded_types))
    )
```

比如，`everything_except(int)` 返回一个策略，该策略可以生成除 `int` 实例外的 `from_type()` 能生成的任何值，以及通过 `register_type_strategy()` 添加的类型的实例。

当编写检查无效输入被以特定方式拒绝的测试时，这很有用。

- **just**(*value*)

  返回仅会生成 `value` 的策略。

  注意：`value` 不被复制。使用可变对象时应谨慎。

  如果 `value` 是一个可调用对象的结果，可以使用 `builds(callable)` 代替 `just(callable())` 来每次都生成新值。

  该策略生成的示例不会收缩（因为仅有一个值）。

- **nothing**()

  此策略永远不会成功生成值，并且在尝试生成时将引发异常。

  该策略生成的示例不会收缩（因为根本就没有）。

- **one_of**(**args*)

  one_of(*iterable*)

  返回 `args` 指定的多个策略中的任何一个策略。

  也可以使用元素为策略的可调用对象来调用该方法，在这种情况下，`one_of(x)` 和 `one_of(*x)` 是等效的。

  通常，此策略的示例将收缩为列表中较早策略的示例，然后根据产生这些策略的策略的行为进行缩小。为了获得良好的收缩行为，请将较简单的策略放在前面。例如 `one_of(none(), text())` 比 `one_of(text(), none())` 要好。

  当使用递归策略时，这尤为重要，例如：

  `x = st.deferred(lambda: st.none() | st.tuples(x, x))` 将会很好的收缩，但是 `x = st.deferred(lambda: st.tuples(x, x) | st.none())` 则不会很好的收缩。

- **permutations**(*values*)

  返回一个返回有序集合值的排列的策略。

  返回的策略的示例向 `values` 的原始顺序收缩。

- **random_module**()

  Hypothesis 引擎在内部处理标准库和Numpy随机模块的伪随机数生成器（PRNG）的状态，始终将它们的种子设定为零，并在测试后恢复以前的状态。

如果使用固定的种子会削弱测试，并且不能使用 `randoms()` 提供的 `random.Random` 实例，则此策略将使用任意的整数调用 `random.seed()` 并传递一个不透明的对象，该对象的 `repr` 显示用于调试的种子值。如果 `numpy.random` 可用，则其状态也将被管理。

这些策略的示例收缩到接近零的种子。

- **randoms**(**, note_method_calls=False, use_true_random=False*)

  生成 `random.Random` 实例。生成的 `Random` 实例属于一个特殊的 `HypothesisRandom` 子类。

  - 如果 `note_method_calls` 设置为 `True`，Hypothesis 将会在证伪的测试用例中打印随机生成的值。这对于调试随机算法的行为很有帮助。
  - 如果 `use_true_random` 设置为 `True`，则将从其常规分布中提取值，否则它们实际上是Hypothesis生成的值（并且将在任何失败的测试用例中相应地收缩）。设置 `use_true_random=False` 将倾向于暴露在设置为 `True` 时发生概率极低的错误，并且仅当代码依赖于正确性的值分布时，才应将此标志设置为 `True`。

- **recursive**(*base, extend, *, max_leaves=100*)

  - `base`：起始策略。
  - `extend`：接受一个策略并返回新的策略的函数。
  - 在给定运行中从 `base` 生成的最大元素数。

  返回策略 `S`，以使 `S = extend(base | S)`。也就是说，值可以从 `base` 中生成，也可以从混合使用 `|` 和 `extend` 的任何策略中生成。

  使用一个例子来说明：`recursive(booleans(), lists)` 将返回一个策略，该策略可以返回任意嵌套和混合的布尔值列表。所以 `False`、`[True]`、`[False, []]` 和 `[[[[[True]]]]]` 都是从该策略中得出的有效值。

  这种策略中的示例通过试图减少递归量和根据 `base` 的收缩行为和 `extend` 的结果来收缩。

- **register_type_strategy**(*custom_type, strategy*)

  在全局类型到策略（type-to-strategy）的映射中添加一个条目。

  该映射用于 `builds()` 和 `@given`。

  `builds()` 将自动用于在 `__init__` 上带有类型注释的类，因此，只有在一个或多个参数需要比其基于类型的默认值更严格地定义时，或者要为具有默认值的参数提供策略时，才需要注册策略。

  `strategy` 可以是搜索策略，也可以是接受类型并返回策略的函数（对泛型类型有用）。

  请注意，可能无法直接注册参数化的泛型类型（例如 `MyCollection[int]`），因为解析逻辑无法正确处理这种情况。不过，可以为 `MyCollection` 注册一个函数并检查该函数中的类型参数。

- **sampled_from**(*elements*)

  返回一个生成 `elements` 中出现的任何值的策略。

  注意，与 `just()` 一样，值不会被拷贝，因此，应谨慎使用可变数据。

  `sampled_from` 支持有序集合，以及 `Enum` 对象。`Flag` 对象还可以生成其成员的任何组合。

  该策略的示例通过使用列表中较早的值进行替换来收缩，因此，例如，`sampled_from([10, 1])` 通过尝试使用 `10` 替换 `1` 来收缩，而 `sampled_from([1, 10])` 通过尝试使用 `1` 替换 `10` 来收缩。

  从空序列中进行抽样是一个错误，因为返回 `nothing()` 太容易静默地删除复合策略的某些部分。

- **shared**(*base, *, key=None*)

返回一个每次运行均从 `base` 生成一个共享值的策略。具有相同 `key` 的任何两个共享实例将共享相同的值，否则将使用此策略的标识。也就是说：

```
>>> s = integers()  # or any other strategy
>>> x = shared(s)
>>> y = shared(s)
```

上面的 `x` 和 `y` 可能生成不同的（或相同的）值。而下面的则生成相同的值：

```
>>> x = shared(s, key="hi")
>>> y = shared(s, key="hi")
```

该策略的示例会像它们的每个 `base` 策略一样收缩。

临时策略

该模块包含各种临时API和策略。

它们仅供内部使用，以简化代码重用，并且不稳定。 发行要点可能随时移动或破坏内容！

Internet 策略应符合 RFC 3986 或其链接的权威定义。如果不符合，请报告错误！

- **hypothesis.provisonal.domains**(**, max_length=255, max_element_length=63*)

  生成符合 RFC 1035 的完全限定域名。
- **hypothesis.provisional.urls**()

  用于 RFC3986 的策略，生成 http/https URL。

收缩（Shrinking）

在使用策略时，数据如何**收缩**，是值得考虑的事情 。收缩是Hypothesis在发现故障时试图通过导致该故障的示例制作出易于理解的示例的过程：Hypothesis接受一个复杂的示例并将其变成一个简单的示例。

每种策略都定义了收缩的顺序——通常不需要太在意，但是值得一提的是，收缩的顺序会影响编写自定义策略的最佳方式。

确切的收缩行为不是API所保证的一部分，但是并不会经常更改；如果发生更改，通常是因为我们认为新方法可以生成更好的示例。

可能要注意的最重要的一个策略是 `one_of()`，它偏爱参数列表中较早的策略所产生的值。其他大多数策略在很大程度上都会 "做正确的事"，而无需过多考虑。

适应性策略

通常情况下，一种策略并不能完全产生想要的结果，而是需要对其进行调整。 有时可以在测试中执行此操作，但会损害重用性，因为随后必须在每个测试中重复进行调整。

Hypothesis提供了从其他具有给定数据转换函数的策略中构建策略的方法。

map

`map` 可能是这些函数中最简单和最有用的一个。如果有一个策略 `s` 和一个函数 `f`，则示例 `s.map(f).example()` 等同于 `f(s.example())`，例如，我们从 `s` 生成一个示例，然后将 `f` 应用到该示例：

例如：

```
>>> lists(integers()).map(sorted).example()
[-25527, -24245, -23118, -93, -70, -7, 0, 39, 40, 65, 88, 112, 6189, 9480, 19469,
27256, 32526, 1566924430]
```

请注意，许多使用 `map` 实现的功能也能通过 `builds()` 实现。

filter

可以通过 `filter` 过滤掉一些示例。 `s.filter(f).example()` 是 `s` 中使得 `f(example)` 为真的示例。

```
>>> integers().filter(lambda x: x > 11).example()
26126
>>> integers().filter(lambda x: x > 11).example()
23324
```

请务必注意，如果条件难以满足，则 `filter` 可能会失败：

```
>>> integers().filter(lambda x: False).example()
Traceback (most recent call last):
    ...
hypothesis.errors.Unsatisfiable: Could not find any valid examples in 20 tries
```

通常，应该只使用过滤器来避免不必要的极端情况，而不是尝试切掉大部分搜索空间。

通常比较有效的一种技术是先使用 `map` 转换数据，然后使用过滤器过滤掉不满足条件的数据。因此，例如，如果想生成成对的整数 `(x, y)` ，并且 `x <y` ，则可以执行以下操作：

```
>>> tuples(integers(), integers()).map(sorted).filter(lambda x: x[0] <
x[1]).example()
[-8543729478746591815, 3760495307320535691]
```

flatmap

`flatmap` 首先生成一个示例，然后将该示例转换为一个策略，最后，再根据该策略生成一个示例。

一开始，您可能不太清楚为什么需要这样做，但结果证明这是非常有用的，因为通过这种方式可以生成彼此之间存在某种关系的不同类型的数据。

例如，我们想生成一个元素为列表的列表，其中每个子列表的长度都相同：

```
>>> rectangle_lists = integers(min_value=0, max_value=10).flatmap(
...     lambda n: lists(lists(integers(), min_size=n, max_size=n))
... )
>>> rectangle_lists.example()
[]
>>> rectangle_lists.filter(lambda x: len(x) >= 10).example()
[[], [], [], [], [], [], [], [], [], []]
>>> rectangle_lists.filter(lambda t: len(t) >= 3 and len(t[0]) >= 3).example()
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> rectangle_lists.filter(lambda t: sum(len(s) for s in t) >= 10).example()
[[0], [0], [0], [0], [0], [0], [0], [0], [0], [0]]
```

在上面的例子中，首先为元组指定一个长度 *n*，然后构建一个策略，该策略生成包含长度为 *n* 的列表的列表。输出显示了生成的示例。

大多数情况下，您可能不希望使用 `flatmap` ，但与 `filter` 和 `map` 仅提供实现某些功能的便捷方式不同的是， `flatmap` 允许您真正地生成新的数据，而这是原本无法轻松完成的。

递归数据

有时，要生成的数据具有递归定义。 例如，如果要生成JSON数据，而有效的JSON数据是：

1. 任意浮点数、任意布尔值、任意 Unicode 字符串。
2. 有效 JSON 数据的任意列表。
3. 将 Unicode 字符串映射到有效 JSON 数据的任意字典。

问题在于，不能期待在递归地调用策略的情况下不会耗尽所有的内存。这里的另一个问题是，并非所有的Unicode字符串都能在不同的计算机上一致地显示，因此我们将仅在doctest中进行演示。

Hypothesis 使用 `recursive()` 策略处理此类问题。首先将基础策略和为特定数据类型提供策略的函数传递给 `recursive()` 策略，然后返回新的策略。 因此，例如：

```
>>> from string import printable
... from pprint import pprint
>>> json = recursive(
...     none() | booleans() | floats() | text(printable),
...     lambda children: lists(children, min_size=1)
...     | dictionaries(text(printable), children, min_size=1),
... )
>>> pprint(json.example())
[[1.175494351e-38, ']', 1.9, True, False, '.M}Xl', ''], True]
>>> pprint(json.example())
{'de(l': None,
 'nK': {'(Rt)': None,
        '+hoZh1YU]gy8': True,
        '8z]EIFA06^li^': 'LFE{Q',
        '9,': 'l{cA=/'}}
```

也就是说，从叶子数据开始，然后通过生成可以作为JSON数据的任意列表和字典来进行扩充。

通过对从基础策略中提取的值的最大数量进行限制，从而控制生成的递归数据的规模。 因此，例如，如果我们只想生成很小的JSON，则可以这样做：

```
>>> small_lists = recursive(booleans(), lists, max_leaves=5)
>>> small_lists.example()
True
>>> small_lists.example()
[False]
```

复合策略

`composite` 装饰器用于以任意方式组合其他策略。这可能是定义复杂的自定义策略的主要方式。

`composite` 装饰器的工作原理是将一个返回示例的函数转换为返回产生此类示例的策略的函数——可以将被装饰的函数传递给 `@given` 、使用 `.map` 或 `.filter` 进行修改或将其像他任何策略一样使用。

`composite` 将被其装饰的函数的第一个参数作为一个特殊的函数提供给你，你可以在测试中将这个特殊的函数像 `data()` 策略的对应方法一样使用。实际上，`data()` 和 `composite()` 的实现方式几乎是相同的——但是使用 `composite` 定义策略可以使代码更易重用，并且通常可以改善失败示例的显示。

例如，下面的示例给出了一个列表和该列表中的任意一个索引：

```
>>> @composite
... def list_and_index(draw, elements=integers()):
...     xs = draw(lists(elements, min_size=1))
...     i = draw(integers(min_value=0, max_value=len(xs) - 1))
...     return (xs, i)
...
```

`draw(s)` 应被视为返回 `s.example()` 的函数，但是结果是可重现的，并且将正确地收缩。被装饰的函数的第一个参数已从参数列表中删除，但被装饰的函数将按预期顺序接受所有其他参数，并且默认值将被保留。

```
>>> list_and_index()
list_and_index()
>>> list_and_index().example()
([15949, -35, 21764, 8167, 1607867656, -41, 104, 19, -90, 520116744169390387,
7107438879249457973], 0)

>>> list_and_index(booleans())
list_and_index(elements=booleans())
>>> list_and_index(booleans()).example()
([True, False], 0)
```

请注意，被 `composite` 装饰的函数的对象表示形式（repr）与所有内置策略的对象表示形式完全相似：一个可以被调用并返回策略的函数，并且仅当提供的参数值与参数的默认值不匹配时，提供的这个参数值才会显示。

可以在 `composite` 函数中使用 `assume`：

```
@composite
def distinct_strings_with_common_characters(draw):
    x = draw(text(min_size=1))
    y = draw(text(alphabet=x))
    assume(x != y)
    return (x, y)
```

`assume` 会像往常一样正常工作，过滤掉使得传入的参数计算为 `False` 的所有示例。

请注意，被 `composite` 装饰的函数可以应付对抗性生成（cope with adversarial draws），或者使用 `.filter()` 方法或 `assume()` 显式过滤掉一些示例——突变和收缩逻辑可能会做一些奇怪的事情，并且糟糕的实现可能会导致严重的性能问题。 例如：

```
@composite
def reimplementing_sets_strategy(draw, elements=st.integers(), size=5):
    # 不好的方式：如果 Hypothesis 一直生成同一个值，
    # 那么循环将持续很长的时间
    result = set()
    while len(result) < size:
        result.add(draw(elements))
    # 好的方式：使用过滤器，这样 Hypothesis 就知道什么数据是无效的！
    for _ in range(size):
        result.add(draw(elements.filter(lambda x: x not in result)))
    return result
```

如果使用 `composite` 装饰一个实例方法或者类方法，那么 `draw` 参数应该在 `self` 或 `cls` 之前。因此，尽管我们建议将策略编写为独立的函数，并使用 `register_type_strategy()` 函数将它们与类相关联，但也支持使用 `composite` 装饰实例方法和类方法，并且 `@composite` 装饰器可以在 `@classmethod` 或 `@staticmethod` 之上或之下。 请参阅 问题 #2578 和 pull request #2634，以获取更多详细信息。

在测试中交互地生成

可以通过 `data` 交互地使用策略。这样就没有必要在 `@given` 中预先指定所有内容，而是可以在测试函数定义体中通过策略生成示例。

`data()` 与 `@composite` 类似，但功能更强大，因为 `data()` 允许您将示例生成混入测试代码。`data()` 的缺点是与显式的 `@example(...)` 不兼容，并且当出现错误时，混合的代码通常更难调试。

如果需要受之前的生成影响，但不依赖于测试执行的值，请使用更简单的 `@composite`。

```
@given(data())
def test_draw_sequentially(data):
    x = data.draw(integers())
    y = data.draw(integers(min_value=x))
    assert x < y
```

如果测试失败，将会打印带有反例的每一次生成。例如，上面的生成将会出错（边界条件错误），因此，将会打印：

```
Falsifying example: test_draw_sequentially(data=data(...))
Draw 1: 0
Draw 2: 0
```

如您所见，以这种方式生成的数据像往常一样被简化。

您也可以提供标签，以标识每次调用 `data.draw()` 所生成的值。 这些标签可用于标识输出的反例中的值。

例如：

```
@given(data())
def test_draw_sequentially(data):
    x = data.draw(integers(), label="First number")
    y = data.draw(integers(min_value=x), label="Second number")
    assert x < y
```

将会产生下面的输出：

```
Falsifying example: test_draw_sequentially(data=data(...))
Draw 1 (First number): 0
Draw 2 (Second number): 0
```

内置拓展

Hypothesis具有最小的依赖，可以最大程度地提高兼容性，并使Hypothesis的安装尽可能容易。

因此，与特定包的集成是由 `extra` 模块提供的，并且与特定包的需要安装各自的依赖才能正常工作。可以使用 `setuptools` 的附加功能来安装这些依赖项，例如， `pip install hypothesis[django]` 。这将检查兼容版本的安装。

也可以通过上述方法将 Hypothesis 安装到项目中，忽略版本约束，并希望达到最佳效果。

通常，"Hypothesis与哪个版本兼容？" 这是一个很难回答的问题，甚至更难以定期测试。Hypothesis 始终针对最新的兼容版本进行测试，并且每个包都会注明预期的兼容范围。如果遇到以上任何一个错误，请指定依赖版本。

针对Django用户的 Hypothesis 和 针对科学堆栈的 Hypothesis 则有单独的页面。

hypothesis[cli]

```
$ hypothesis --help
Usage: hypothesis [OPTIONS] COMMAND [ARGS]...

Options:
  --version   Show the version and exit.
  -h, --help  Show this message and exit.

Commands:
  codemod  `hypothesis codemod` refactors deprecated or inefficient code.
  fuzz     [hypofuzz] runs tests with an adaptive coverage-guided fuzzer.
  write    `hypothesis write` writes property-based tests for you!
```

该模块需要 `click` 包，并且提供 Hypothesis 的命令行接口。例如，通过终端使用 `ghostwriting tests` 。这也是 HypoFuzz 添加 `hypothesis fuzz` 命令的地方（更多信息，参见 HypoFuzz 文档）。

hypothesis[codemods]

该模块提供基于 LibCST 库的 codemod，该模块可以检测并自动修复使用 Hypothesis 的代码的问题，包括从弃用的功能升级到推荐的样式。

可以通过CLI运行codemods：

```
$ hypothesis codemod --help
Usage: hypothesis codemod [OPTIONS] PATH...

  `hypothesis codemod` refactors deprecated or inefficient code.

  It adapts `python -m libcst.tool`, removing many features and config
  options which are rarely relevant for this purpose.  If you need more
  control, we encourage you to use the libcst CLI directly; if not this one
  is easier.

  PATH is the file(s) or directories of files to format in place, or "-" to
  read from stdin and write to stdout.

Options:
  -h, --help  Show this message and exit.
```

或者，可以使用 `python -m libcst.tool`，它以附加配置（在 `.libcst.codemod.yaml` 中的 `modules` 列表中添加 `'hypothesis.extra'`）为代价提供了更多控制，并且 在Windows上存在一些问题。

- **hypothesis.extra.codemods.refactor**(*code*)

  将源字符串从弃用的Hypothesis API更新为最新的。

  这可能无法解决代码中的所有弃用警告，但是我们相信，这比手工完成要容易得多。

  我们建议使用CLI，但是如果要使用Python函数，则可以使用该函数。

hypothesis[dpcontracts]

该模块提供了与 dpcontracts 库结合使用的工具，因为将合约和基于特性的测试结合起来确实非常有效。

该模块需要 `dpcontracts >= 0.4`。

> 提示
>
> 对于新项目，我们建议使用 deal 或 icontract 和 icontract-hypothesis，而不是 pcontracts。通常，前者是用于按合约设计（design-by-contract）编程的功能更强大的工具，并且还具有更好的 Hypothesis 集成！

- **hypothesis.extra.pcontracts.fulfill**(*contract_func*)

  装饰 `contract_func` 以拒绝违反预置条件的调用，然后使用不同的参数重试。

  这是一个方便的函数，用于测试使用 dpcontracts 的内部代码，以在触发合约错误之前自动过滤掉公开接口会拒绝的参数。

  这可以用作 `builds(fulfill(func), ...)` 或在测试主体中使用，例如，`assert fulfill(func)(*args)`。

hypothesis[lark]

使用Lark解析器库，可以使用此模块生成与任何上下文无关语法匹配的字符串。

目前仅支持Lark的本机EBNF语法，但我们计划将其扩展为支持其他常见语法，例如ANTLR和RFC 5234 ABNF。Lark已经支持从nearley.js加载语法，因此您可能根本不必编写自己的语法。

请注意，由于Lark的版本为0.x，因此如果不支持最新版本的Lark，则该模块可能会破坏次要版本中的API兼容性。如果有人自愿出资资助或进行维护，我们也可能会大胆尝试降低Lark的最低版本。

- **hypothesis.extra.lark.from_lark**(*grammar, *, start=None, explicit=None*)

  用于生成给定上下文无关语法接受的字符串策略。

  `grammar` 必须是包装EBNF规范的 `Lark` 对象。Lark EBNF语法参考可以在这里找到。

`from_lark` 将自动生成与语法中非终结 `start` 符号匹配的作为 `Lark` 类的参数提供的字符串。要生成与包括终结符在内的其他符号匹配的字符串，可以将 `start` 参数传递给 `from_lark` 。请注意，在编译语法时，Lark可能会删除不可达的代码，因此您可能应该将相同的 `start` 值传递给这两个。

当前 `from_lark` 不支持需要自定义词法化的语法。任何词法分析器都将被忽略，并且任何使用 `%declare` 的未定义终结符将导致生成错误。要为此类终结符定义策略，请传递将其名称映射到相应策略的字典作为 `explicit` 参数。

hypothesmith 项目包含基于语法和妥善的后期处理的用于生成Python源码的策略。

可以在Lark存储库和第三方集合中找到为测试提供有用的起点的示例语法。

hypothesis[pytz]

该模块提供 `pytz` 时区。

可以使用该策略使 `hypothesis.strategies.datetimes()` 和 `hypothesis.strategies.times()` 生成时区相关的值。

- **hypothesis.extra.pytz.timezones()**

  Olsen数据库中的任何时区，作为pytz `tzinfo` 对象。

  此策略收缩到UTC或最小的可能固定偏移量，并且被设计为与 `hypothesis.strategies.datetimes()` 一起使用。

hypothesis[dateutil]

该模块提供 `dateutil` 时区。

可以使用该策略使 `hypothesis.strategies.datetimes()` 和 `hypothesis.strategies.times()` 生成时区相关的值。

- **hypothesis.extra.dateutil.timezones()**

  来自 `dateutil` 的时区。

  此策略收缩到UTC或与 UTC 时间 2000-01-01 有最小偏移量的时区，并且被设计为与 `datetimes()` 策略一起使用。

  请注意，该策略生成的时区可能会根据计算机的配置而有所不同。 有关更多信息，请参见 `dateutil` 文档。

更多示例

这是一些有关如何以有趣的方式使用Hypothesis的示例的集合。 该集合目前很小，但会随着时间的推移而增长。

所有这些示例都被设计为在 `pytest` 下运行，但是在 `nose` 下应该也能运行。

如何不按部分顺序排序?

以下是从 Hypothesis 的早期版本中发生的实际错误中提取并简化的示例。真正的错误很难找到。

假设我们有如下类型:

```
class Node:
    def __init__(self, label, value):
        self.label = label
        self.value = tuple(value)

    def __repr__(self):
        return f"Node({self.label!r}, {self.value!r})"

    def sorts_before(self, other):
        if len(self.value) >= len(other.value):
            return False
        return other.value[: len(self.value)] == self.value
```

每个节点都有一个标签和一个包含数据的元组，以及判断节点之间是否具有 `sorts_before` 关系的方法。如果两个节点之间具有 `sorts_before` 关系，则表示左侧节点的数据是右侧节点的数据的初始段。 所以，值为 `[1, 2]` 的节点将排在值为 `[1, 2, 3]` 的节点之前，但是值为 `[1, 2]` 的节点和值为 `[1, 3]` 的节点之间不存在这种关系。

我们有一个节点列表，并且我们希望根据上述顺序对它们进行拓扑排序。也就是说，我们想要对该排列进行列表，以便当 `x.sorts_before(y)` 时， `x` 在列表中的出现早于 `y` 。我们天真地认为，执行此操作的最简单方法是通过任意方式决胜，以将上面定义的部分顺序扩展为总顺序，从而可以使用普通的排序算法。因此，我们定义以下代码:

```
from functools import total_ordering


@total_ordering
class TopoKey:
    def __init__(self, node):
        self.value = node

    def __lt__(self, other):
        if self.value.sorts_before(other.value):
            return True
        if other.value.sorts_before(self.value):
            return False

        return self.value.label < other.value.label
```

```
16
17
18  def sort_nodes(xs):
19      xs.sort(key=TopoKey)
```

`TopKey` 接受 `sorts_before` 定义的顺序并通过比较节点标签的方式进行决胜从而将其拓展为总顺序。

但现在我们想测试它是否有效。

首先，我们编写一个函数来验证我们想要的结果：

```
1  def is_prefix_sorted(xs):
2      for i in range(len(xs)):
3          for j in range(i + 1, len(xs)):
4              if xs[j].sorts_before(xs[i]):
5                  return False
6      return True
```

如果找到顺序错误的节点对，则返回 `False`，否则返回 `True`。

有了这个函数，我们想要使用 Hypothesis 做的事情就是，对于所有节点序列，断言对其调用 `sort_nodes` 的结果都是经过排序的。

首先，我们需要为 `Node` 定义策略：

```
1  import hypothesis.strategies as st
2
3  NodeStrategy = st.builds(Node, st.integers(), st.lists(st.booleans(), max_size=10))
```

我们希望生成短的值列表，以便有很大的机会将一个值作为另一个值的前缀（这也是为什么选择布尔值作为元素的原因）。然后，我们定义一种策略，该策略将根据整数和这些布尔值的短列表之一构建一个节点。

现在，我们可以写一个测试：

```
from hypothesis import given


@given(st.lists(NodeStrategy))
def test_sorting_nodes_is_prefix_sorted(xs):
    sort_nodes(xs)
    assert is_prefix_sorted(xs)
```

下面的示例导致上面的测试失败了

```
[Node(0, (False, True)), Node(0, (True,)), Node(0, (False,))]
```

之所以会失败，是因为 `False` 不是 `(True, True)` 的前缀，反之亦然，前两个节点之所以排序为相等的，是因为它们具有相同的标签。这使得整个顺序不具传递性，并产生了基本无意义的结果。

但是，这是令人非常不满意的。因为仅当两个节点具有相同的标签时才起作用。也许我们实际上希望标签具有唯一性。让我们更改测试以确保这一点。

```
def deduplicate_nodes_by_label(nodes):
    table = {node.label: node for node in nodes}
    return list(table.values())
```

我们定义了一个通过标签对 *重复的节点（具有相同标签的节点）* 进行去重的函数，现在可以将这个函数映射到生成节点列表的策略上，从而提供了会生成具有唯一标签的节点列表的策略：

```
@given(st.lists(NodeStrategy).map(deduplicate_nodes_by_label))
def test_sorting_nodes_is_prefix_sorted(xs):
    sort_nodes(xs)
    assert is_prefix_sorted(xs)
```

但是 Hypothesis 仍然很快给出了一个导致测试失败的示例：

```
[Node(0, (False,)), Node(-1, (True,)), Node(-2, (False, False))]
```

这是一个更有趣的示例。在这个示例中，没有节点会排序为相等的。其中，第一个节点严格小于最后一个节点，因为 `(False,)` 是 `(False, False)` 的前缀。而最后一个节点又严格小于第二个节点，因为两者都不是另一个的前缀并且 `-2 < -1`。然后，中间节点小于第一个节点，因为 `-1 < 0`。

因此可以确信之前的实现是糟糕的。现在来编写了一个更好的实现：

```
def sort_nodes(xs):
    for i in range(1, len(xs)):
        j = i - 1
        while j >= 0:
            if xs[j].sorts_before(xs[j + 1]):
                break
            xs[j], xs[j + 1] = xs[j + 1], xs[j]
            j -= 1
```

这只是一个略经修改的插入排序——我们向后交换节点，直到进一步交换会违反顺序约束。之所以起作用，是因为节点之间的顺序已经是部分顺序（对于一般的拓扑排序，这不会产生有效的结果，如果要产生有效的结果，那么顺序需要具有传递性）。

现在，我们再次运行测试，然后测试通过了，这表明已经成功地对一些节点进行了排序。

时区算术

下面是一些 pytz 测试的示例，这些测试可检查各种时区转换的行为是否符合预期。这些测试都应该通过，并且主要演示了使用 Hypothesis 进行测试时的一些有用的东西，以及 `datetimes()` 策略如何工作。

```
from datetime import timedelta

# 默认情况下，datetimes 策略是时区无关的，因此需要告诉它使用 timezones
aware_datetimes = st.datetimes(timezones=st.timezones())


@given(aware_datetimes, st.timezones(), st.timezones())
def test_convert_via_intermediary(dt, tz1, tz2):
    """测试时区之间的转换不受绕过的其他时区的影响。.
    """
    assert dt.astimezone(tz1).astimezone(tz2) == dt.astimezone(tz2)


@given(aware_datetimes, st.timezones())
def test_convert_to_and_fro(dt, tz2):
    """如果转换到一个新的时区，然后再转换回旧的时区，那么结果将保持不变。
    """
    tz1 = dt.tzinfo
    assert dt == dt.astimezone(tz2).astimezone(tz1)


```

```
@given(aware_datetimes, st.timezones())
def test_adding_an_hour_commutes(dt, tz):
    """在时区之间进行转换时，在转换前增加一个小时与在转换后增加一个小时的效果相同。
    """
    an_hour = timedelta(hours=1)
    assert (dt + an_hour).astimezone(tz) == dt.astimezone(tz) + an_hour


@given(aware_datetimes, st.timezones())
def test_adding_a_day_commutes(dt, tz):
    """在时区之间进行转换时，在转换前增加一天与在转换后增加一天的效果相同。
    """
    a_day = timedelta(days=1)
    assert (dt + a_day).astimezone(tz) == dt.astimezone(tz) + a_day
```

投票悖论

投票理论中的一个经典悖论，称为Condorcet悖论，即大多数人的偏好不具有传递性。比如，有一群人和三个候选项 A、B 和 C，其中大多数人喜欢 A 胜过喜欢 B，喜欢 B 胜过喜欢 C，喜欢 C 胜过喜欢 A。

何不使用Hypothesis来提供一个示例呢?

关键点是如何表示选举结果——在本示例中，我们将使用一个 "投票" 列表，其中每个投票都是按照投票人的偏好列出的候选项列表。下面是代码：

```
from collections import Counter

from hypothesis import given
from hypothesis.strategies import lists, permutations


# 需要至少三个候选项和至少三个投票人才能出现悖论；
# 否则，只能导致胜出，或者在最糟糕的情况下，导致平局。
@given(lists(permutations(["A", "B", "C"]), min_size=3))
def test_elections_are_transitive(election):
    all_candidates = {"A", "B", "C"}

    # 首先，计算出有多少人更喜欢某一对候选人
    counts = Counter()
    for vote in election:
        for i in range(len(vote)):
            for j in range(i + 1, len(vote)):
                counts[(vote[i], vote[j])] += 1

```

```
    # 现在，看看哪一个候选人对中的一方比另一方占优势，并将其存储起来。
    graph = {}
    for i in all_candidates:
        for j in all_candidates:
            if counts[(i, j)] > counts[(j, i)]:
                graph.setdefault(i, set()).add(j)

    # 现在，对于每个三元组，断言其是可传递的。
    for x in all_candidates:
        for y in graph.get(x, ()):
            for z in graph.get(y, ()):
                assert x not in graph.get(z, ())
```

下面是 Hypothesis 在我运行上面的测试时给出的示例（在你的电脑上行运行时显示的示例可能会有所不同）：

```
[["A", "B", "C"], ["B", "C", "A"], ["C", "A", "B"]]
```

确实可以做到这一点：多数人（投票0和1）喜欢 B 胜过喜欢 C，（投票0和2）喜欢 A 胜过喜欢 B，（投票1和2）喜欢 C 胜过喜欢 A。上面Hypothesis 给出的示例基本上是投票悖论的典型示例。

对 HTTP API 进行模糊测试

```
import math
import os
import random
import time
import unittest
from collections import namedtuple

import requests

from hypothesis import assume, given, strategies as st

Goal = namedtuple("Goal", ("slug",))


# 通过环境变量传入 API 凭证.
waspfinder_token = os.getenv("WASPFINDER_TOKEN")
waspfinder_user = os.getenv("WASPFINDER_USER")
assert waspfinder_token is not None
assert waspfinder_user is not None

```

```
GoalData = st.fixed_dictionaries(
    {
        "title": st.text(),
        "goal_type": st.sampled_from(
            ["hustler", "biker", "gainer", "fatloser", "inboxer", "drinker",
"custom"]
        ),
        "goaldate": st.one_of(st.none(), st.floats()),
        "goalval": st.one_of(st.none(), st.floats()),
        "rate": st.one_of(st.none(), st.floats()),
        "initval": st.floats(),
        "panic": st.floats(),
        "secret": st.booleans(),
        "datapublic": st.booleans(),
    }
)


needs2 = ["goaldate", "goalval", "rate"]


class WaspfinderTest(unittest.TestCase):
    @given(GoalData)
    def test_create_goal_dry_run(self, data):
        # 我们希望对于每个运行而言，slug 都是唯一的，以便多个测试运行不会相互干扰。
        # 如果由于某些原因，一些 slug 触发了错误，而另一些 slug 没有触发错误，
        # 我们将得到一个flaky错误，但这没关系。
        slug = hex(random.getrandbits(32))[2:]

        # 使用 assume 来指导我们进行已知的验证，否则将花费大量时间来生成无聊的示例。

        # 标题不得为空
        assume(data["title"])

        # 这些值中必须恰好有两个值不为 None. 其他的将会由 API 推断.
        assume(len([1 for k in needs2 if data[k] is not None]) == 2)

        for v in data.values():
            if isinstance(v, float):
                assume(not math.isnan(v))
        data["slug"] = slug

        # 该API很好地支持使用空运行选项，这意味着我们不必担心用户帐户被大量伪造的目标所淹没，
        # 否则我们必须确保在运行测试后对自身进行清理。
        data["dryrun"] = True
        data["auth_token"] = waspfinder_token
```

```
        for d, v in data.items():
            if v is None:
                data[d] = "null"
            else:
                data[d] = str(v)
        result = requests.post(
            "https://waspfinder.example.com/api/v1/users/"
            "%s/goals.json" % (waspfinder_user,),
            data=data,
        )

        # 不要过于频繁的向API发请求，这会使测试运行的比以前更慢
        time.sleep(1.0)

        # 目前，我们正在测试的API不会产生内部错误。如果不使用空运行选项，
        # 则可以尝试对结果进行更多操作，但这是一个好的开始。
        self.assertNotEqual(result.status_code, 500)


if __name__ == "__main__":
    unittest.main()
```

代写测试

使用 Hypothesis 编写测试可以使您从决定和编写特定的测试输入的繁琐工作中解放出来。 现在，`hypothesis.extra.ghostwriter` 模块也可以为您编写测试函数！

其理念是**提供一种简便的方法来开始**基于属性的测试，并**无缝过渡**到更复杂的测试代码——因为代写的测试是您可以自己编写的源代码。

因此，只需选择一个您要测试的函数，然后将其提供给以下函数之一即可。它们遵循导入，使用但不需要类型注释，并且通常尽最大努力为您编写有用的测试。 您还可以使用命令行接口：

```
$ hypothesis write --help
Usage: hypothesis write [OPTIONS] FUNC...

  `hypothesis write` writes property-based tests for you!

  Type annotations are helpful but not required for our advanced
  introspection and templating logic.  Try running the examples below to see
  how it works:

      hypothesis write gzip
```

```
        hypothesis write numpy.matmul
        hypothesis write re.compile --except re.error
        hypothesis write --equivalent ast.literal_eval eval
        hypothesis write --roundtrip json.dumps json.loads
        hypothesis write --style=unittest --idempotent sorted
        hypothesis write --binary-op operator.add

Options:
  --roundtrip               start by testing write/read or encode/decode!
  --equivalent              very useful when optimising or refactoring code
  --idempotent
  --binary-op
  --style [pytest|unittest]  pytest-style function, or unittest-style method?
  -e, --except OBJ_NAME      dotted name of exception(s) to ignore
  -h, --help                 Show this message and exit.
```

注意

Ghostwriter 需要 black，但是生成的代码仅需要 Hypothesis。

注意

有法律问题吗？ 尽管ghostwriter的代码片段和逻辑与Hypothesis 的其他代码片段和逻辑一样，都在 MPL-2.0 许可下，但ghostwriter的输出是在 Creative Commons Zero（CC0） 公共领域专用协议下提供的，因此您可以不受限制地使用。

- **hypothesis.extra.ghostwriter.magic**(**modules_or_functions, except_=(), style='pytest'*)

  对于一个模块或函数集合，猜测要使用哪个ghostwriter。

  对于所有ghostwriter， `exception_` 参数应为 `Exception` 或异常的元组，并且 `style` 可以是将生成测试函数的 `"pytest"` 或生成测试方法和 `TestCase` 子类的 `"unittest"` 。

  在找到附加到任何模块的公共函数之后， `magic` ghostwriter 将会查找要传递给 `roundtrip()` 的成对的函数，然后检查 `binary_operation()` 和 `ufunc()` 函数，并将其他函数传递给 `fuzz()` 。

  例如，在命令行上尝试 `hypothesis write gzip` 。

- **hypothesis.extra.ghostwriter.fuzz**(*func, *, except_=(), style='pytest'*)

  编写 `func` 的基于特性的测试的源代码。

  生成的测试将会检查有效输入仅会导致预期的异常。 例如，对于：

```
from re import compile, error

from hypothesis.extra import ghostwriter

ghostwriter.fuzz(compile, except_=error)
```

将生成如下的测试：

```
# This test code was written by the `hypothesis.extra.ghostwriter` module
# and is provided under the Creative Commons Zero public domain dedication.
import re

from hypothesis import given, reject, strategies as st

# TODO: replace st.nothing() with an appropriate strategy


@given(pattern=st.nothing(), flags=st.just(0))
def test_fuzz_compile(pattern, flags):
    try:
        re.compile(pattern=pattern, flags=flags)
    except re.error:
        reject()
```

请注意，它包括所有必需的导入。由于 `pattern` 参数没有注解或默认参数，因此您需要指定一种策略——例如 `text()` 或 `binary()` 。之后，您将得到一个可运行的测试！

- **hypothesis.extra.ghostwriter.idempotent**(*func, *, except_=(), style='pytest'*)

  编写 `func` 的基于特性的测试的源代码。

  生成的测试将会检查对 `func` 的调用结果再次调用 `func` 后，结果是否保持不变。 例如，对于：

```
from typing import Sequence

from hypothesis.extra import ghostwriter


def timsort(seq: Sequence[int]) -> Sequence[int]:
    return sorted(seq)


ghostwriter.idempotent(timsort)
```

将会生成如下的测试：

```
# This test code was written by the `hypothesis.extra.ghostwriter` module
# and is provided under the Creative Commons Zero public domain dedication.

from hypothesis import given, strategies as st


@given(seq=st.one_of(st.binary(), st.binary().map(bytearray),
st.lists(st.integers())))
def test_idempotent_timsort(seq):
    result = timsort(seq=seq)
    repeat = timsort(seq=result)
    assert result == repeat, (result, repeat)
```

- **hypothesis.extra.ghostwriter.roundtrip**(**funcs, except_=(), style='pytest'*)

  编写 `funcs` 的基于特性的测试的源代码。

  生成的测试将会检查最终的结果是否等于首个输入参数。其中，最后的结果按照如下方式得出：调用第一个参数，将调用结果传给第二个函数，以此类推，得出最后的结果。

  这是一个非常强大的可以进行测试的特性，尤其是当配置选项与要往返的对象一起变化时。例如，尝试为 `json.dumps()` 代写测试——您会想到所有这些吗?

```
$ hypothesis write --roundtrip json.dumps json.loads
```

- **hypothesis.extra.ghostwriter.equivalent**(**funcs, except_=(), style='pytest'*)

  编写 `funcs` 的基于特性的测试的源代码。

  生成的测试将会检查每个函数的调用结果是否相同。这可以用作经典的 "oracle"，例如针对内置的 `sorted()` 函数测试快速排序算法，或用于差异测试，其中，所有被比较的函数都不是完全可信的，但是任何差异都表明存在错误（例如，在不同数量的线程上运行一个函数，或者只是多次运行运行一个函数）。

  这些函数应具有相当相似的签名，因为只有公共形参才会传递相同的实参——而其他形参则可以不同。

- **hypothesis.extra.ghostwriter.binary_operation**(*func, *, associative=True, commutative=True, identity=infer, distributes_over=None, except_=(), style='pytest'*)

  为二进制操作 `func` 编写基于属性的测试。

  尽管二进制操作并不是特别常见，但是它们具有非常好的特性可以进行测试，如果不使用 ghostwriter 来演示它们，似乎很遗憾。 对于运算符 *f*，进行以下测试：

  - 如果 `associative` 为 `True`，则 `f(a, f(b, c)) == f(f(a, b), c)`
  - 如果 `commutative` 为 `True`，则 `f(a, b) == f(b, a)`
  - 如果 `indentity` 不是 `None`，则 `f(a, identity) == a`
  - 如果 `distributes_over` 是 `+`，则 `f(a, b) + f(a, c) == f(a, b+c)`

例如：

```
ghostwriter.binary_operation(
    operator.mul,
    identity=1,
    distributes_over=operator.add,
    style="unittest",
)
```

- **hypothesis.extra.ghostwriter.ufunc**(*func, *, except_=(), style='pytest'*)

  为 array ufunc `ufunc` 编写基于属性的测试。

  生成的测试将检查您的 `ufunc` 或 `gufunc` 是否具有预期的广播和dtype强制转换行为。您可能想要添加额外的断言，但是与其他ghostwriter一样，这为您提供了一个不错的起点。

```
$ hypothesis write numpy.matmul
```

健康检查

Hypothesis 试图通过一系列 "健康检查" 来检测常见的错误以及在运行时会引起困难的事情。

其中包括检测和警告：

- 数据生成速度非常慢的策略
- 过滤过多的策略
- 分支策略过多的递归策略
- 在合理的时间内不太可能完成的测试。

如果检测到这些情况中的任何一种，则Hypothesis将发出有关这些情况的警告。

这些健康检查的一般目标是警告你，你正在做的事情可能看起来有效，但会导致Hypothesis无法正常工作或性能下降。

要有选择地禁用健康检查，请使用 `prevent_health_check` 设置。 此参数的值是一个列表，其中包含从 `HealthCheck` 类的任何类级别属性中生成的元素。使用 `HealthCheck.all()` 的值将禁用所有健康检查。

class **hypothesis.HealthCheck**(*value*)

用于 `suppress_health_check` 的参数。这个枚举的每一个成员都是一个要禁用的健康检查的类型。

- `data_too_large=1`

检查生成的示例的典型大小是否经常超过最大允许大小。

- `filter_too_much=2`

  检查测试何时过滤掉过多的示例，或者通过使用 `assume()` 或 `filter()` 或出于 Hypothesis 内部原因而进行检查。

- `too_slow=3`

  检查您的数据生成何时非常缓慢并可能损害测试。

- `return_value=5`

  检查测试是否返回非 `None` 值（该值将被忽略，并且不太可能执行您想要的操作）。

- `large_base_example=7`

  检查要缩小的自然示例是否非常大。

- `not_a_test_method=8`

  检查 `@given` 是否已应用于 `unittest.TestCase` 定义的方法（例如，不是一个测试）。

- `function_scoped_fixture=9`

  检查 `@given` 是否已应用于具有pytest函数范围夹具的测试。函数范围的夹具针对整个函数运行一次，而不是每个示例运行一次，通常这不是您想要的。

  由于此限制，需要为每个示例设置或重置状态的测试通常需要使用适当的上下文管理器在测试内手动进行此操作。

  仅在极少数情况下需要禁用此健康检查，例如，使用了不需要在各个示例之间重置的函数范围的夹具，但由于某些原因，您不能使用更广泛的夹具范围（例如，会话范围，模块范围，类范围））。

  此检查需要 Hypothesis pytest 插件，当在pytest中运行Hypothesis时，该插件将默认启用。

弃用

我们还使用了一系列自定义异常和警告类型，这样您就可以确切地看到错误来自何处——或者只将警告转换为错误。

class **hypothesis.errors.HypothesisDeprecationWarning**

Hypothesis 发出的弃用警告。

实际上，它继承自 `FutureWarning`，因为默认警告过滤器隐藏了 `DeprecationWarning`。

您可以配置Python警告，以与其他警告不同的方式处理这些警告，既可以将它们转变为错误，也可以将其完全抑制。显然，我们希望使用前者！

弃用特性将会在至少6个月内继续发出警告，然后在接下来的主要版本中被删除。但是请注意，并非所有警告均受此宽限期的约束；有时，我们会通过添加警告来加强验证，并且这些警告在主要版本中可能会立即变成错误。

示例数据库

Hypothesis发现错误后，会在数据库中存储足够的信息以重现该错误。这使您能够拥有一个经典的测试工作流，即查找错误、修复错误、并确信正在做正确的事情，因为Hypothesis首先会使用之前导致测试失败的示例运行测试，然后再使用新的示例运行测试。

限制

最好将示例数据库视为不会失效的缓存：升级Hypothesis版本或更改测试时，信息可能会丢失，因此，不应该依靠示例数据库来确保正确性——如果要确保在每次运行测试时都会使用某些特定的示例，最好将这些示例显式的添加到测试函数中——但是通过确保重现刚刚找到的导致测试失败的示例，可以极大地帮助开发工作流。

示例数据库还记录了使用较少被使用的代码部分的示例，因此即使未找到失败的示例，该数据库也可能会更新。

升级 Hypothesis 并更改测试

Hypothesis数据库的设计使您可以将任意数据放入数据库中，而不会出现错误的行为。升级Hypothesis时，旧数据可能会失效，但这应该透明地发生。 例如，更改生成参数的策略会为提供来自旧策略的数据。

ExampleDatabase 实现

Hypothesis的默认 `database` 设置会在当前工作目录下的 `.hypothesis/examples` 中创建一个 `DirectoryBasedExampleDatabase` 。如果此位置不可用，例如，因为没有读取或写入权限，则Hypothesis将发出警告并回退到 `InMemoryExampleDatabase` 。

Hypothesis 提供了下面一些 `ExampleDatabase` 实现。

- *class* **hypothesis.database.InMemoryExampleDatabase**

  非持久化的示例数据库。根据集合的字典来实现。

  如果在单个会话中多次调用测试函数，或者用于测试其他数据库实现，则此功能很有用，但是由于它在两次运行之间不持久，因此建议不要将其普遍使用。

- *class* **hypothesis.database.DirectoryBasedExampleDatabase**(*path*)

  使用目录将Hypothesis示例存储为文件。

  每个测试对应一个目录，每个示例对应于该目录中的文件。尽管内容相当不透明，但是可以通过将目录检入版本控制中来共享 `DirectoryBasedExampleDatabase` ，例如，使用以下 `.gitignore` ：

  ```
  # Ignore files cached by Hypothesis...
  .hypothesis/*
  # except for the examples directory
  !.hypothesis/examples/
  ```

  但是请注意，这仅在固定到Hypothesis的确切版本时才有意义，并且通常建议使用网络数据存储区实现共享数据库——请参见 `ExampleDatabase` 和 `MultiplexedDatabase` 的帮助文档。

- *class* **hypothesis.database.ReadOnlyDatabase**(*db*)

将给定数据库变为只读的包装器。

该实现允许 `fetch`，但是将 `save`、`delete` 和 `move` 转变为无操作。

请注意，该实现会禁用Hypothesis自动丢弃陈旧示例的功能。该实现旨在允许本地计算机访问共享数据库（例如从CI服务器），而无需从本地或开发中分支传播更改。

- *class* **hypothesis.database.MultiplexedDatabase**(**dbs*)

  多个示例数据库的包装器。

  每一个 `save`、`fetch`、`move` 或 `delete` 操作将会对被包装的所有数据库执行一次。`fetch` 操作不会产出重复的值，即使相同的值出现在两个或两个以上的被包装数据库中。

  该实现可以与 `ReadOnlyuDatabase` 很好的结合。例如：

  ```
  local = DirectoryBasedExampleDatabase("/tmp/hypothesis/examples/")
  shared = CustomNetworkDatabase()

  settings.register_profile("ci", database=shared)
  settings.register_profile(
      "dev", database=MultiplexedDatabase(local, ReadOnlyDatabase(shared))
  )
  settings.load_profile("ci" if os.environ.get("CI") else "dev")
  ```

  因此，CI系统或模糊测试运行可以填充中央共享数据库。虽然开发机上的本地运行可以从CI重现任何故障，但只会在本地缓存其自身的故障，而无法从共享数据库中删除示例。

- *class* **hypothesis.extra.redis.RedisExampleDatabase**(*redis, *, expire_after=datetime.timedelta(days=8), key_prefix=b'hypothesis-example:'*)

  将Hypothesis示例作为集合存储在给定的 `redis.Redis` 数据存储中。

  这对于共享数据库特别有用，正如 `MultiplexedDatabase` 的使用示例所演示的那样。

  > **注意**
  >
  > 如果在 `expire_after` 所指定的时间内未运行测试，则这些示例将被允许过期。默认的TTL在每周运行之间保留示例。

自定义示例数据库实现

可以自定义 `ExampleDatabase` 的实现。只需要覆盖几个方法即可：

- *class* **hypothesis.database.ExampleDatabase**(**args, **kwargs*)

  用于以Hypothesis的内部格式存储示例的示例数据库的抽象基类。

  `ExampleDatabase` 将每个字节键映射到许多不同的字节值，例如，`Mapping[bytes, AbstractSet[bytes]]`。

  - *abstract* **save**(*key*, *value*)

将 `value` 存储到 `key` 下。

如果该 `value` 已出现在 `key` 下，则什么也不做。

- *abstract* **fetch**(*key*)

  返回匹配给定 `key` 的所有值的可迭代对象。

- *abstract* **delete**(*key, value*)

  从 `key` 中删除 `value` 。

  如果 `value` 没有在 `key` 中，则什么也不做。

- *abstract* **move**(*src, dest, value*)

  将 `value` 从键 `src` 移动到键 `dest` 。等同于先执行 `delete(src, value)` 然后执行 `save(dest, value)` ，但是效率更高。

  请注意，无论 `value` 是否出现在 `src` ，它都会被插入 `dest` 。

状态测试

使用 `@given` 的测试仍然是您自己编写的测试，不同之处是 Hypothesis 提供了一些数据。使用 Hypothesis 的状态测试（stateful testing），Hypothesis 不仅尝试生成数据，还尝试生成整个测试。通过指定许多可以组合在一起的原始动作，Hypothesis 会尝试查找导致失败的那些动作的序列。

> **提示**
>
> 在阅读本参考文档之前，我们建议您按顺序阅读 How not to Die Hard with Hypothesis 和 基于规则的状态测试简介。 当在实践中使用之后，并知道为什么每个方法或装饰器可用时，再了解实现细节就更有意义了。

> **注意**
>
> 这种类型的测试通常称为**基于模型的测试**，但在 Hypothesis 中被称为状态测试（主要是出于历史原因——Hypothesis 中该想法的最初实现更接近于 ScalaCheck的状态测试）。这两个名称都有一定的误导性：实际上不需要任何形式的代码模型来使用这种测试，并且这种测试对于不涉及任何状态的纯API和有状态API一样有用。
>
> 最好不要太在意这种测试的名称。无论叫什么名字，这种测试都是一种强大的对于大多数非平凡的API都是有用的测试形式。

可能不需要状态机

状态测试的基本思想是让 Hypothesis 为测试选择操作和值，而状态机是实现这一目的的一种很好的声明方式。

但是对于更简单的情况，可能根本不需要状态机——带有 `@given` 的标准测试可能就足够了，因为可以在分支或循环中使用 `data()` 。事实上，这就是状态机资源管理器内部工作的方式。但是，对于更复杂的工作负载，需要使用更高级别的API，请继续阅读！

基于规则的状态机

class **hypothesis.stateful.RuleBasedStateMachine**

`RuleBasedStateMachine` 提供了一种定义状态机的结构化方法。

其思想是，状态机携带一组划分多个绑定（bundle）的类型数据，并具有一组可以从绑定 （或普通策略）中读取数据和将数据推送到绑定的规则。在任何给定点，将会执行随机的适用规则。

基于 `@given` 的测试会接受从策略生成的值，然后将值传递给用户定义的测试函数。而规则与基于常规 `@given` 的测试非常相似。关键区别在于，基于 `@given` 的测试必须是独立的，而规则可以链接在一起——单个测试运行可能涉及可以以各种方式进行交互的多个规则调用。

规则可以将常规策略作为参数，也可以将特定类型的策略称为绑定。绑定是可以由测试中的其他操作重用的生成的值的命名集合。绑定由规则的结果填充，并且可以用作规则的参数，从而允许数据从一个规则流转到另一个规则，并且规则可以处理先前的计算或操作的结果。

可以将添加到任何绑定中的每个值看做是分配给新变量的。从绑定策略中获取一个值意味着选择一个相应的变量并使用该值，并将 `consums()` 作为该变量的 `del` 语句。如果可以将绑定的使用替换为类的实例属性，那么将会更简单，但严格来说，绑定的功能通常更为强大。

以下基于规则的状态机示例是 Hypothesis 的示例数据库的实现的测试的简化版本。一个示例数据库将键映射到一组值，在此测试中，我们将它的一种实现与它的行为的简化内存模型进行比较，该模型只是将相同的值存储在Python字典中。然后，测试将针对实际数据库及其内存表示形式执行操作，并查找它们行为上的差异。

```
import shutil
import tempfile
from collections import defaultdict

import hypothesis.strategies as st
from hypothesis.database import DirectoryBasedExampleDatabase
from hypothesis.stateful import Bundle, RuleBasedStateMachine, rule


class DatabaseComparison(RuleBasedStateMachine):
    def __init__(self):
        super().__init__()
        self.tempd = tempfile.mkdtemp()
```

```
            self.database = DirectoryBasedExampleDatabase(self.tempd)
            self.model = defaultdict(set)

        keys = Bundle("keys")
        values = Bundle("values")

        @rule(target=keys, k=st.binary())
        def add_key(self, k):
            return k

        @rule(target=values, v=st.binary())
        def add_value(self, v):
            return v

        @rule(k=keys, v=values)
        def save(self, k, v):
            self.model[k].add(v)
            self.database.save(k, v)

        @rule(k=keys, v=values)
        def delete(self, k, v):
            self.model[k].discard(v)
            self.database.delete(k, v)

        @rule(k=keys)
        def values_agree(self, k):
            assert set(self.database.fetch(k)) == self.model[k]

        def teardown(self):
            shutil.rmtree(self.tempd)


    TestDBComparison = DatabaseComparison.TestCase
```

在上面的代码中，声明了两个绑定——一个用于键，一个用于值；两个将被数据（`k` 和 `v`）填充的平凡规则，以及三个非平凡规则：`save` 将值保存到键下，而 `delete` 从键下删除值，并且这两个规则还将更新数据库中实际存储的模型。最后，由 `values_agree` 检查数据库的内容是否与特定键的模型一致。

然后，可以通过从 `DatabaseComparison` 获取 `unittest.TestCase` 来将其集成到现有的测试套件中：

```
TestTrees = DatabaseComparison.TestCase

# Or just run with pytest's unittest support
if __name__ == "__main__":
    unittest.main()
```

上面的测试当前是可以通过的，但是如果将调用 `self.model[k].discard[v]` 的语句注释掉，那么使用 pytest 运行则会打印下面这样的输出：

```
AssertionError: assert set() == {b''}

------------ Hypothesis ------------

state = DatabaseComparison()
var1 = state.add_key(k=b'')
var2 = state.add_value(v=var1)
state.save(k=var1, v=var2)
state.delete(k=var1, v=var2)
state.values_agree(k=var1)
state.teardown()
```

请注意，它是如何打印出一个简短的程序来演示该问题的。基于规则的状态机的输出通常应该与Python代码非常接近——如果具有不返回有效Python代码的自定义 `repr` 实现，则情况会有所不同，但是在大多数情况下，都应该能够将代码复制并粘贴到测试中，从而使问题重现。

可以使用 `TestCase上` 的 `settings` 对象（该对象是正常的 Hypothesis `settings` 对象，使用 `TestCase` 类第一次被引用时的默认值）来控制详细的行为。 例如，如果想使用更大的程序运行较少的示例，则可以将 `settings` 更改为：

```
DatabaseComparison.TestCase.settings = settings(
    max_examples=50, stateful_step_count=100
)
```

这将使每个程序运行的步骤数翻倍，将运行的测试用例数减半。

规则

如前所述，规则是 `RuleBasedStateMachine` 中最常用的功能。通过在函数上应用 `rule()` 装饰器来定义规则。 请注意， `RuleBasedStateMachine` 必须至少定义一个规则，并且不能使用单个函数来定义多个规则（这是为了避免多个规则执行相同的操作）。由于状态执行方法，规则通常不能从其他源（例如夹具或 `pytest.mark.parametrize` ）获取参数——请考虑通过诸如 `sampled_from()` 之类的策略提供参数。

- **hypothesis.stateful.rule**(**, targets=(), target=None, **kwargs*)

  用于定义 `RuleBasedStateMachine` 中的规则的装饰器。 `targets` 或 `target` 中存在的任何名称都将定义用于保存此方法的最终结果的变量。如果 `targets` 和 `target` 都为空，则最终结果将被丢弃。

  `target` 必须是 `Bundle` 对象，或者如果结果应该进入多个绑定，则可以将绑定的元组作为 `target` 参数传递。对单个规则使用两个参数是无效的。如果结果应准确的保存到多个绑定中的一个，则应该为每种情况定义一个单独的规则。

  `kwargs` 用于定义将传递给方法调用的参数。如果 `kwargs` 的值是 `Bundle` 或 `consums(b)` （其中 `b` 是 `Bundle` ），则将提供先前为该绑定生成的值。如果使用了 `comsume` ，则该值也将从绑定中删除。

  任何其他关键字参数都应该是策略，并且来自这些策略的值将会被提供给被装饰的方法。

- **hypothesis.stateful.consumes**(*bundlle*)

  当在 `RuleBasedStateMachine` 中引入规则时，此函数可用于标记应删除给定规则的步骤中使用的每个值的绑定。此函数返回可以像其他任何对象一样进行操纵和组合的策略对象。

  例如，每次执行时，以下规则将消费来自绑定 `b2` 的一个值和来自绑定 `b3` 的多个值，以填充 `value2` 和 `value3` ：

  ```
  @rule(value1=b1, value2=consumes(b2), value3=lists(consumes(b3)))
  ```

- **hypothesis.stateful.multiple**(**args*)

  此函数可用于将多个结果传递给规则的 `target(s)` 。只需在规则中使用 `return multiple(result1, result2, ...)` 。

  也可以使用不带参数的 `return multiple()` 来结束规则而不传递任何结果。

初始化

初始化是规则的一种特殊情况，可以保证在运行开始时（即在调用任何常规规则之前）最多运行一次。请注意，如果定义了多个初始化规则，则可以按任何顺序进行调用，并且每次运行的顺序大不相同。

初始化通常对填充绑定很有用：

- **hypothesis.stateful.initialize**(**, targets=(), target=None, **kwargs*)

  用于定义 `RuleBasedStateMachine` 中的初始化规则的装饰器。

该装饰器的行为与 `rule` 的行为类似，但是所有被 `initialize` 装饰的方法将在所有被 `rule` 装饰的方法之前以任意顺序执行。每个被 `initialize` 装饰的方法在每次运行时仅精确的执行一次，除非其中的一个引发了异常——在这种情况下，将仅运行 `.teardown()` 方法。被 `initialize` 装饰的方法可能没有预置条件。

```
import hypothesis.strategies as st
from hypothesis.stateful import Bundle, RuleBasedStateMachine, initialize, rule

name_strategy = st.text(min_size=1).filter(lambda x: "/" not in x)


class NumberModifier(RuleBasedStateMachine):

    folders = Bundle("folders")
    files = Bundle("files")

    @initialize(target=folders)
    def init_folders(self):
        return "/"

    @rule(target=folders, name=name_strategy)
    def create_folder(self, parent, name):
        return f"{parent}/{name}"

    @rule(target=files, name=name_strategy)
    def create_file(self, parent, name):
        return f"{parent}/{name}"
```

预置条件

尽管在 `RuleBasedStateMachine` 规则中可以使用 `assume()`，但如果仅在少数几个规则中使用，则可能会很快遇到很少或根本没有一个通过假设的情况。因此，假设提供了一个 `precondition()` 装饰器来避免此问题。 `precondition()` 装饰器用于被 `rule` 装饰的方法，并且必须为其提供一个基于 `RuleBasedStateMachine` 实例返回 `True` 或 `False` 的函数。

- **hypothesis.stateful.precondition**(*precond*)

  用于为 `RuleBasedStateMachine` 中的规则设置预置条件的装饰器。该装饰器指定将规则视为状态机中有效步骤的预置条件，这比在规则中使用 `assume()` 更有效。将使用 `RuleBasedStateMachine` 实例调用 `precond` 函数，并应返回 `True` 或 `False` 。通常， `precond` 需要查看该实例上的属性。

  例如：

```
class MyTestMachine(RuleBasedStateMachine):
    state = 1

    @precondition(lambda self: self.state != 0)
    @rule(numerator=integers())
    def divide_with(self, numerator):
        self.state = numerator / self.state
```

如果为单个规则设置了多个预置条件，只有当所有预置条件返回 `True` 时才认为该规则是有效的步骤。预置条件可以应用于不变量以及规则。

```
from hypothesis.stateful import RuleBasedStateMachine, precondition, rule


class NumberModifier(RuleBasedStateMachine):

    num = 0

    @rule()
    def add_one(self):
        self.num += 1

    @precondition(lambda self: self.num != 0)
    @rule()
    def divide_with_one(self):
        self.num = 1 / self.num
```

通过使用 `precondition` 而不是 `assume()`，Hypothesis 可以在运行规则之前过滤掉不兼容的规则。这使得很有可能会生成一系列有用的步骤。

请注意，当前 `precondition` 无法访问绑定；如果需要使用 `precondition`，则应将相关数据存储在实例上。

不变量

通常，希望确保在过程中的每个步骤之后都满足不变量。可以将不变量添加为正在运行的规则，但是这些不变量将在其他规则之间运行零次或多次。假设提供了一个用于标记要在每个步骤之后运行的函数的装饰器。

- **hypothesis.stateful.invariant**(**, check_during_init=False*)

  为 `RuleBasedStateMachine` 中的规则设置不变量的装饰器。被装饰的函数将在每个规则之后运行，并且可能引发异常以指示失败的不变量。

  例如：

```
class MyTestMachine(RuleBasedStateMachine):
    state = 1

    @invariant()
    def is_nonzero(self):
        assert self.state != 0
```

默认情况下，仅在运行所有被 `initialize()` 装饰的规则后才检查不变量。传递 `check_during_init=True` 以指示可以在初始化期间进行检查。

```
from hypothesis.stateful import RuleBasedStateMachine, invariant, rule


class NumberModifier(RuleBasedStateMachine):

    num = 0

    @rule()
    def add_two(self):
        self.num += 2
        if self.num > 50:
            self.num += 1

    @invariant()
    def divide_with_one(self):
        assert self.num % 2 == 0


NumberTest = NumberModifier.TestCase
```

也可以为不变量设置预置条件，在这种情况下，仅当预置条件函数返回 `True` 才检查该不变量。

请注意，不变量不能访问绑定；如果需要使用不变量，则应将相关数据存储在实例上。

更细粒度的控制

如果要绕过 `TestCase` 基础结构进行手动调用，则可以使用 `hypothesis.stateful` 模块公开的 `run_state_machine_as_test` 函数，该函数接受返回 `RuleBasedStateMachine` 的任意函数和可选地 `settings` 参数，并且与基于类的 `runTest` 所提供的功能相同。

不建议这样做，因为该函数绕过了一些重要的内部函数，包括报告诸如运行时和 `event()` 调用的统计信息的函数。该函数最初是为了支持自定义 `__init__` 方法而添加的，但现在可以使用 `initialize()` 规则来代替自定义的 `__init__` 方法。

兼容性

Hypothesis竭尽全力与其他可能会一起使用的库保持兼容。一般来说，您应该首先进行尝试以检查是否兼容。如果不兼容，请查看本文档以了解详细信息。

Hypothesis 版本

向后兼容性比支持修复要好，因此我们使用语义版本控制，并且仅支持最新版本的Hypothesis。 有关更多信息，请参见 帮助和支持。

除了主要版本之间的冲突外，已记录的API不会发生中断性更改。除非明确指出是临时的，否则本文档中提到的所有API都是公开的，在这种情况下，可能会在次要版本中对其进行更改。未记录的属性、模块和行为可能包括修补程序版本中的重大更改。

Python 版本

Hypothesis在CPython 3.6+上得到了支持和测试，例如，所有具有上游支持的CPython版本。

Hypothesis 还支持适用于Python 3.6的最新PyPy。 尽管目前仅在Windows上进行了测试，但32位CPython版本也可以使用。

通常，除了受支持的任何Python版本的最新修补程序版本以外，其他Python版本都不受正式支持。较早的版本应该可以工作，并且如果报告了错误，这些错误将得到修复，但是由于没有在CI中对较早的版本进行测试，因此无法保证兼容性。

测试框架

一般而言，Hypothesis需要花费大量的精力来生成看起来像普通Python测试函数的函数，并尽可能地使这些函数的行为与原始函数的行为相接近，因此对于每个测试框架，Hypothesis都应该可以开箱即用。

如果测试依赖于执行其他操作而不是调用函数并查看是否引发异常，则可能无法开箱即用。特别是像返回生成器并希望对生成器进行某些操作的测试（例如，nose 的基于 `yield` 的测试）是无法正常工作的。使用装饰器或类似的东西来包装测试函数以便采用 Hypothesis 支持的形式，或者要求框架维护者支持 Hypothesis 的钩子，以便稍后能够插入这样的包装器。

已知的受支持的框架;

- Hypothesis 可以与pytest和unittest 完美集成，并且作为CI的一部分进行了验证。
- 对于以 `@given` 装饰器装饰的测试函数，pytest 夹具可以照常工作，唯一需要注意的是不要为由夹具提供值的参数提供策略。然而，每个夹具将为每个测试函数而不是每个示例运行一次。使用 `@given` 装饰夹具

函数是没有意义的。

- `unittest.mock.patch()` 装饰器可以与 `@given` 一起使用，但我们建议将其用作被装饰的测试函数中的上下文管理器，以确保 mock 是针对每个测试用例的，并避免与Pytest夹具的不良交互。
- Nose 可以与 Hypothesis 一起使用，并且是作为 CI 的一部分进行了测试的。基于 `yield` 的测试无法与 Hypothesis 一起使用。
- 与 Django 测试的集成需要使用 Django 用户的 Hypothesis 包。问题在于，在Django测试的正常执行模式下，每个测试均会重置一次数据库，而不是每个示例重置一次数据库，这不是我们期待的行为。
- Coverage 可以与 Hypothesis 完美集成。Hypothesis 的测试套件具有100%的分支覆盖率。

可选包

文档中列出了 `hypothesis.extra` 中策略的可选软件包的受支持版本。我们的总体目标是支持上游支持的所有版本。

常规验证

我们使用GitHub Actions在每次提交时检查上面明确支持的所有东西。我们的持续交付渠道会在发布每个版本之前运行所有这些检查，因此，文档中列出的受支持的库必然能够被支持。

重现失败的测试

对于使用随机测试的人来说，经常要考虑的问题之一是如何重现失败的测试用例。

> 注意
>
> 最好将 Hypothesis生成的数据是看作是 *任意的*，而不是 *随机的*。Hypothesis会故意生成任何可能会导致错误的有效数据，因此不应依赖生成的数据的任何预期分布或它们之间的关系。如果您有兴趣，可以阅读有关 "swarm testing" 和 "覆盖率为导向的模糊测试" 的信息，因为您无需了解假设！

幸运的是，Hypothesis具有许多功能来支持重现测试失败。示例数据库 是在本地开发时最常使用的用来重现失败的测试的功能，这意味着完全不必考虑本地使用的问题——测试失败会自动重现，而无需执行任何操作。

示例数据库非常适合在机器之间共享，但是目前还没有很好的工作流，因此，Hypothesis 提供了多种方法，可以通过将示例添加到测试的源代码中来使其具有可重现性。例如，当您尝试重新运行在CI上失败的示例，或者在计算机之间共享导致失败的示例时，示例数据库是非常有用的。

提供显式的示例

重现失败测试的最简单方法是让Hypothesis运行所打印的失败示例。 例如，如果打印了证伪的示例：`test(n=1)`，则可以使用 `@example(n=1)` 装饰测试函数。

`@example` 还可用于确保始终将特定示例作为回归测试执行或覆盖某些边缘情况——将Hypothesis测试和传统的参数化测试结合在一起。

- **hypothesis.example**(**args, **kwargs*)

  用于确保特定的示例总是被测试的装饰器。

Hypothesis 会首先运行所有显式提供的示例。如果显式提供的任何示例失败了，则不会进一步运行更多的示例。

`@example` 装饰器和 `@given` 装饰器的顺序无关紧要。

请注意，`@example` 可以基于位置参数或关键字参数。如果使用位置参数，则在调用时将从右到左进行填充，因此以下两种样式均可按预期工作：

```
@given(text())
@example("Hello world")
@example(x="Some very long string")
def test_some_code(x):
    assert True


from unittest import TestCase


class TestThings(TestCase):
    @given(text())
    @example("Hello world")
    @example(x="Some very long string")
    def test_some_code(self, x):
        assert True
```

与 `@given` 一样，不允许单个 `example` 混合使用位置参数和关键字参数。

使用 @seed 重现特定的测试运行

- **hypothesis.seed**(*seed*)

  从特定的种子启动测试执行。

  `seed` 可以是任何可哈希对象。`seed` 除了提供固定的种子值以使 Hypothesis 尝试相同的操作（只要它能给出非确定性的外部源，如定时和哈希随机化）外，没有任何其他含义。

  覆盖非随机化设置，该设置旨在启用确定性构建，而不是重现观察到的故障。

当测试意外失败（通常是由于运行状况检查失败）时，如果测试尚未使用固定种子运行，则Hypothesis会打印出导致该失败的种子。然后，可以使用 `@seed` 装饰器或（如果正在运行pytest）使用 `--hypothesis-seed` 重新创建该故障。 例如，多亏了 `@seed()`，以下测试函数和 `RuleBasedStateMachine` 每次执行时都将检查相同的示例：

```
@seed(1234)
@given(x=...)
def test(x):
    ...


@seed(6789)
class MyModel(RuleBasedStateMachine):
    ...
```

使用 @reproduce_failure 重现一个示例

Hypothesis具有不透明的二进制表示形式，可用于生成的所有示例。此表示形式不打算在版本之间或测试中的变化方面保持稳定，而是可以用于通过 `@reproduce_failure` 装饰器重现失败。

- **hypothesis.reproduce_failure**(*version, blob*)

  运行与此数据 `blob` 对应的示例，以重现故障。

  使用此装饰器进行的测试始终仅运行一个示例，并且始终失败。如果提供的示例没有导致失败，或者在某种程度上对该测试无效，则其将失败，并显示 `DidNotReproduce` 错误。

  此装饰器不是要永久添加到测试套件中。只需添加一些代码，即可在无法访问测试数据库的情况下简化问题重现。因此，无法在不同版本的 Hypothesis 之间保证兼容性——其API可能会在版本之间任意更改。

目的是永远不要手工编写此装饰器，而是由Hypothesis提供。当测试失败并带有证伪的示例时，Hypothesis 可能会打印出在测试中使用 `@reproduce_failure` 来重新创建问题的建议，如下所示：

```
>>> from hypothesis import settings, given, PrintSettings
>>> import hypothesis.strategies as st
>>> @given(st.floats())
... @settings(print_blob=True)
... def test(f):
...     assert f == f
...
>>> try:
...     test()
... except AssertionError:
...     pass
...
Falsifying example: test(f=nan)

You can reproduce this example by temporarily adding @reproduce_failure(...,
b'AAAA//AAAAAAAAEA') as a decorator on your test case
```

将建议的装饰器添加到测试中应该会重现失败（只要其他所有条件都相同——更改Python版本或涉及的任何其他内容，当然可能会影响测试的行为！请注意，更改Hypothesis的版本将导致不同的错误——每个 `@reproduce_failure` 调用都特定于一个Hypothesis版本）。

默认情况下，不打印这些消息。 如果要查看这些内容，必须将 `print_blob` 设置设置为 `True`。