PyCon China 2020

PyConChina 2020
PYTHON 中国开发者大会 2020

Python For Good

使用Rust助力加速Python

流云坠海 | Chuigda

大纲

1. 分析Python的优缺点
2. 分析Rust的优缺点
3. 安利Ruaaast
4. Python是如何与Rust交互的
5. 总结

Python: Pros&Cons

Pros

众所周知，Python是一个灵活的编程语言

简单易学——容易学习和教学

它拥有极强的元编程能力

热加载能力

以及比大部分主流语言强的开发效率和表达力

```
1 import importlib
2 module = importlib.import_module("name")

10    print("hello world")
```

Python: Pros&Cons

Cons

Python老生常谈的性能问题

- 解释器开销，变量查表开销，运行时类型检查开销
- 不加限制的Dynamic操作造成的优化困难

Python的安全性

- Python的静态分析和验证难做
- Python缺乏可靠（强制）的类型系统
  - 通过类型验证的代码未必完全符合定义行为
  - 一些人为确定的，可靠的代码，却无法通过类型检查

相比于Rust，我为什么不用C拓展?

Rust: Pros&Cons

Pros

动 力 强 劲

- 接近于C的强大性能
- 零成本GC以及它的几乎无停顿垃圾回收

对复杂结构的处理能力

同样也很灵活的元编程能力——宏与过程宏

安全性!

- 通过类型系统保证其内存安全和线程安全
- 通过类型系统来**证明**代码正确性

低碳

Rust: 44 knots speed

Python For Good
PyCon China 2020 | PYTHON 中国开发者大会 2020

The Computer Language Benchmarks Game

Rust versus C gcc fastest

vs C | vs Clang | vs C++

Always look at the source code.

These are only the fastest programs. Lo
programs. They may seem more-like a
you.

binary-trees

source	secs	mem	gz	busy
Rust	**1.21**	200,476	721	4.33
C gcc	1.79	168,760	809	5.35

k-nucleotide

source	secs	mem	gz	busy
Rust	**2.76**	159,224	1691	10.20
C gcc	3.81	130,112	1506	12.18

mandelbrot

source	secs	mem	gz	busy
Rust	**0.93**	32,788	763	3.69
C gcc	1.27	31,692	1135	5.08

n-body

source	secs	mem	gz	busy
Rust	**3.31**	836	1767	3.36
C gcc	4.30	8	1391	4.32

reverse-complement

source	secs	mem	gz	busy
Rust	**0.76**	995,032	1330	1.36
C gcc	0.86	698,264	820	1.27

fannkuch-redux

source	secs	mem	gz	busy	cpu load
Rust	**7.28**	1,168	1016	28.82	100% 97% 100% 100%
C gcc	7.52	836	910	29.37	99% 100% 93% 99%

spectral-norm

source	secs	mem	gz	busy	cpu load
Rust	**0.71**	2,696	1055	2.86	100% 100% 100% 100%
C gcc	0.72	1,100	569	2.86	100% 99% 100% 100%

pidigits

source	secs	mem	gz	busy	cpu load
Rust	0.74	2,888	1420	0.76	0% 0% 100% 3%
C gcc	0.73	2,848	416	0.75	1% 100% 1% 0%

fasta

PyCon China 2020

Rust: Pros&Cons

Cons

陡峭的学习曲线 一大半锅都得给官方文档背

挫败感 和编译器斗智斗勇时的挫败感

开发效率

当然，你可以无脑用Arc（Atomic Reference Count）+Clone

那你的性能也别想要了

生态问题

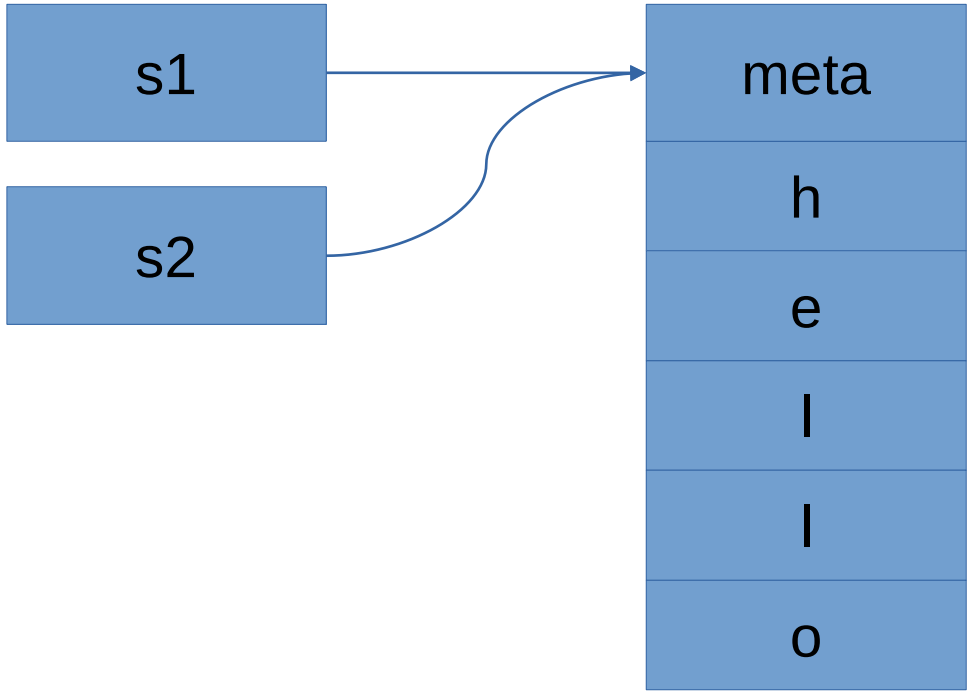
筛选

ADT（代数数据类型）
Pattern Matching（模式匹配）
Ownership & Borrrow Checking（所有权和借用检查）
Trait & dyn Trait（特征）
Macro & Process Macro（宏与过程宏）
……

Python For Good
PyCon China 2020 | PYTHON 中国开发者大会 2020

Python & String

```
s1 = "hello"
s2 = s1
print(s1)
print(s2)
```



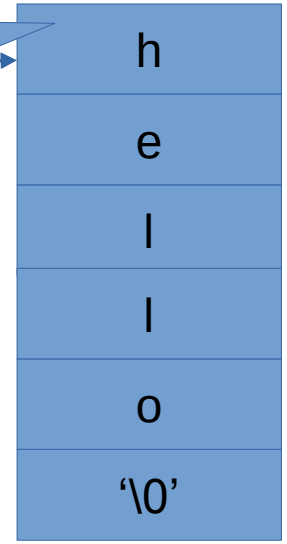
PyCon China 2020

Why Rust amazing: ownership

C & String

```
char *s = (char*)malloc(6 *
strcpy(s, "hello");
char *s2 = s;
puts(s);
puts(s2);
```

Memory leak!



“Stack” “Heap”

Why Rust amazing: ownership

Malloc & free: the bad point

```c
char *s1 = /* malloc */;
strcpy(s1, "hello");
char *s2 = s1;

free(s1);
free(s2); /* double free */
```

Why Rust amazing: ownership

Malloc & free: the bad point

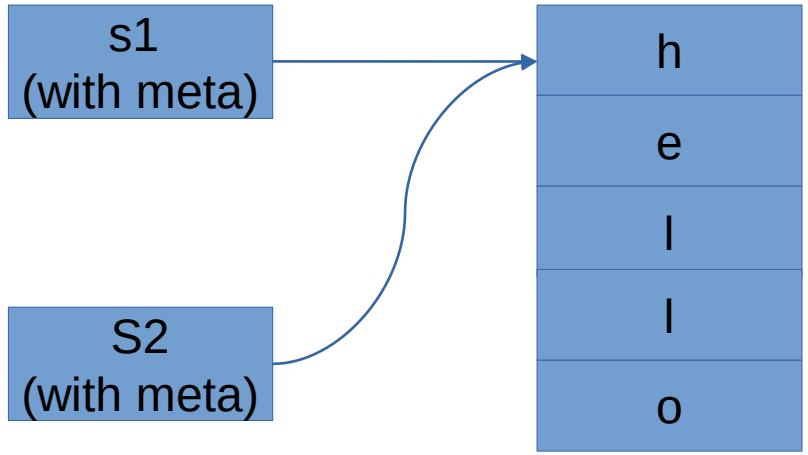
```c
char *getString(void) {
    char *ret = /* malloc */;
    /* strcpy */
    return ret;
}

int main() {
    char *s = getString();
    /* ... */
    free(s);
}
```

```c
/* imported from some lib */
char *getString(void);

int main() {
    char *s = getString();

    /* should I free s? */
}
```

Why Rust amazing: ownership

Rust & String

```
   Compiling playground v0.0.1 (/playground)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:4:20
  |
2 |     let s1 = "hello".to_string();
  |         -- move occurs because `s1` has type `String`
3 |     let s2 = s1;
  |              -- value moved here
4 |     println!("{}", s1);
  |                    ^^ value borrowed here after move
```



Why Rust amazing: ownership


```rust
fn get_string() -> String {
    // any implementation
    // but returning a String always requires a move
    // caller always has the ownership of the returned value
}

fn main() {
    let s = get_string();
    // s automatically dropped
}
```

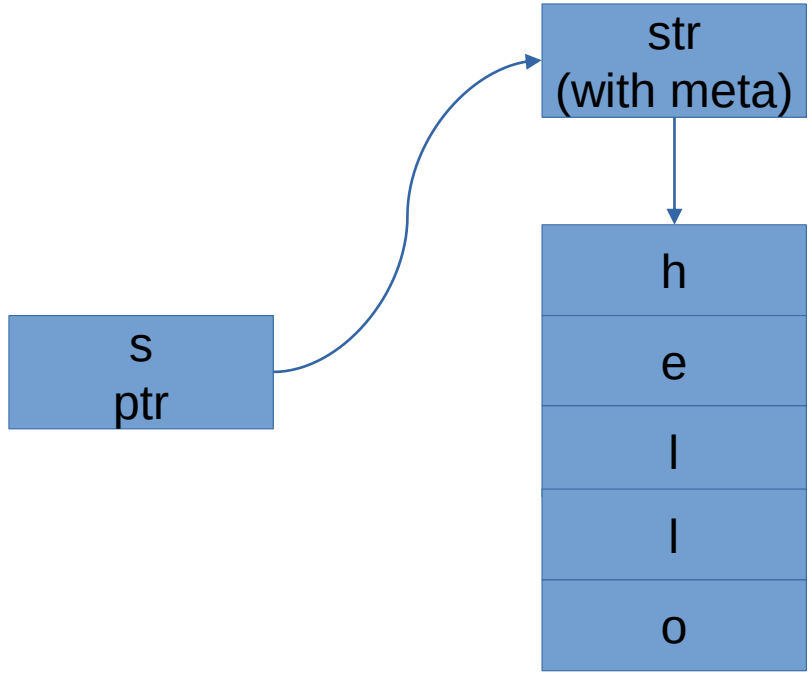
But……

Why Rust amazing: borrow checker


```
1 fn get_str_length(s: String) → (String, usize) {
2     return (s, s.len());
3 }
```

Why Rust amazing: borrow checker


```rust
fn calculate_length(s: &String) -> usize {
    s.len()
}
```



Why Rust amazing: borrow checker

只有可变值才可被可变的借用

```
1 let a = String::from("hello world");
2 let b = &mut a;
3 println!("a: {}", b);
```

```
error[E0596]: cannot borrow `a` as mutable, as it is not declared as mutable
 --> src/main.rs:5:13
  |
4 |     let a = String::from("hello world");
  |         - help: consider changing this to be mutable: `mut a`
5 |     let b = &mut a;
  |             ^^^^^^ cannot borrow as mutable
```

```
1 let mut a = String::from("hello world");
2 let b = &mut a;
3 println!("b: {}", b);
```

```
b: hello world
```

Why Rust amazing: borrow checker


```rust
fn add_end(s: &String) {
    s.push_str("end");
}
```

只有可变借用才能被修改

```
error[E0596]: cannot borrow `*s` as mutable, as it is behind a `&` reference
 --> src/main.rs:4:5
  |
3 | fn add_end(s: &String) {
  |               ------- help: consider changing this to be a mutable reference: `&mut String`
4 |     s.push_str("end");
  |     ^ `s` is a `&` reference, so the data it refers to cannot be borrowed as mutable
```

Why Rust amazing: borrow checker


```rust
let a = String::from("hello world");
let b = &a;
let c = &a;
println!("a: {}", a);
```

```rust
let mut a = String::from("hello world");
let b = &mut a;
let c = &mut a;
b.push_str("114");
c.push_str("514");
```

可变不共享，共享不可变

将这个原则推广到更多地方……

```
error[E0499]: cannot borrow `a` as mutable more than once at a time
 --> src/main.rs:6:13
  |
5 |     let b = &mut a;
  |             ------ first mutable borrow occurs here
6 |     let c = &mut a;
  |             ^^^^^^ second mutable borrow occurs here
7 |     b.push_str("114");
  |     - first borrow later used here
```

小结

肉眼检查、文档交接、代码规范

编译器类型系统的强制检查

PyO3: Rust 与 Python 沟通的桥梁

将Rust编译为Python模块
将Python嵌入到Rust中使其作为脚本执行

为 Python添加 Rust Module


```rust
fn module1(py: Python, m: &PyModule) -> PyResult<()> {
    unimplemented!()
}
```

```rust
#[pymodule]
fn module1(py: Python, m: &PyModule) -> PyResult<()> {
    unimplemented!()
}
```

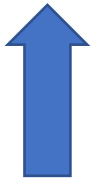
PyO3：对象的映射

Python	Rust	Rust (Python-native)
`object`	-	`&PyAny`
`str`	`String`, `Cow<str>`, `&str`	`&PyUnicode`
`bytes`	`Vec<u8>`, `&[u8]`	`&PyBytes`
`bool`	`bool`	`&PyBool`
`int`	Any integer type (`i32`, `u32`, `usize`, etc)	`&PyLong`
`float`	`f32`, `f64`	`&PyFloat`
`complex`	`num_complex::Complex`[1]	`&PyComplex`
`list[T]`	`Vec<T>`	`&PyList`
`dict[K, V]`	`HashMap<K, V>`, `BTreeMap<K, V>`	`&PyDict`
`tuple[T, U]`	`(T, U)`, `Vec<T>`	`&PyTuple`
`set[T]`	`HashSet<T>`, `BTreeSet<T>`	`&PySet`
`frozenset[T]`	`HashSet<T>`, `BTreeSet<T>`	`&PyFrozenSet`
`bytearray`	`Vec<u8>`	`&PyByteArray`
`slice`	-	`&PySlice`
`type`	-	`&PyType`
`module`	-	`&PyModule`
`datetime.datetime`	-	`&PyDateTime`
`datetime.date`	-	`&PyDate`
`datetime.time`	-	`&PyTime`
`datetime.tzinfo`	-	`&PyTzInfo`
`datetime.timedelta`	-	`&PyDelta`
`typing.Optional[T]`	`Option<T>`	-
`typing.Sequence[T]`	`Vec<T>`	`&PySequence`
`typing.Iterator[Any]`	-	`&PyIterator`

为 Module 添加 Function


```rust
fn add(a: usize, b: usize) -> usize {
    a + b
}
```

```rust
#[pyfunction]
fn add(a: usize, b: usize) -> usize {
    a + b
}
```



```rust
#[pymodule]
fn module1(py: Python, m: &PyModule) -> PyResult<()> {
    #[pyfn(m, "add")]
    fn add(a: usize, b: usize) -> usize {
        a + b
    }
    Ok(())
}
```

```rust
#[pymodule]
fn module1(py: Python, m: &PyModule) -> PyResult<()> {
    m.add_wrapped(wrapper: wrap_pyfunction!(add))?;
    Ok(())
}
```

PyO3带来的加速效果


```python
def factorial(i: int) -> int:
    if i == 0:
        return 0
    else:
        return i * factorial(i-1)
```

```rust
#[pyfn(m, "factorial")]
fn factorial(i: usize) -> usize {
    if i == 0 {
        1
    } else {
        i*factorial(i-1)
    }
}
```

PyO3带来的加速效果

Rust

```
In [34]: %time _ = [qhsm.factorial(50) for _ in range(100_000)]
Wall time: 32 ms
```

```
In [37]: %time _ = [qhsm.factorial(50) for _ in range(1_000_000)]
Wall time: 167 ms
```

```
In [39]: %time _ = [qhsm.factorial(50) for _ in range(10_000_000)]
Wall time: 1.69 s
```

Python

```
In [36]: %time _ = [factorial(50) for _ in range(100_000)]
Wall time: 475 ms
```

```
In [38]: %time _ = [factorial(50) for _ in range(1_000_000)]
Wall time: 4.74 s
```

```
In [41]: %time _ = [factorial(50) for _ in range(10_000_000)]
Wall time: 49.9 s
```

总结

- Rust是一门安全的，高效的编程语言，我们可以用它来加速Python的硬CPU计算
- Rust的类型系统是Rust安全性的基石
- 抛开健壮性。在AI和科学计算领域，Rust并不能代替生态极强的Python

AND MORE

PyCon China 2020

Python For Good

PyConChina 2020 | PYTHON 中国开发者大会 2020

THANK YOU