一个Go开发者的Rust体验

> JUL 21, 2018 #rust #go

一直很关注Rust这门语言的发展，不过没有实际使用过。最近Rust准备以2018 Rust的名义发布可以用作生产环境的稳定版本，又赶上有兴趣写点东西，所以把一个基础模块同时用Rust和Go实现了一下。本文就是这次实现的一些结果。

我自己有很长的Go使用经历，所以本文对Go的看法会相对比较准确。Rust虽然关注了很长时间，但代码基本上是最近一个星期左右的成果，可能看法有偏颇。

这次对比的内容是实现一个针对短字符串优化的字符串。在反序列化时，由于字符串长度未知，导致反序列化时需要根据接收到的字符串创建一段内存来容纳内容。不过一般情况下，反序列化时的字符串长度都较小，由此导致反复申请释放内存，会消耗很多性能。短字符串的优化是，针对某个长度的字符串，提前分配好内存，如果反序列化的长度不足这个长度，直接使用分配好的内存存储内容，不再申请内存，也就没有了释放过程，从而提高效率。

由于我对Go更加熟悉，所以首先使用Go实现了这个功能：

```
const prealloc_size = 20


type String struct {
  data [prealloc_size]byte
  buf  []byte
}


func (s *String) fill(r io.Reader, n uint) error {
  if n > prealloc_size {
```

```
    s.buf = make([]byte, n)
  } else {
    s.buf = s.data[:n]
  }
  _, err := io.ReadFull(r, s.buf)
  return err
}

func (s *String) String() string {
  return *(*string)(unsafe.Pointer(&s.buf))
}
```

这段代码相当直白，不做过多介绍。唯一值得一提的是`String.String()`方法，利用`unsafe`来将`buf`直接当做字符串，避免再次分配内存。具体可以参考官方库的`strings.Builder.String()`方法。

既然要测试性能，必然要有相关的性能测试代码。为了简洁，这里就不展示单元测试的内容了：

```
const (
        shortStr = "short str"
        longStr  = "looooooooooonnnnnnnnnngggggggg string"
)

func BenchmarkShortString(b *testing.B) {
        b.StopTimer()
        var str String
        data := []byte(shortStr)
        r := bytes.NewReader(data)
        l := uint(len(data))
        b.StartTimer()
```

```
        for i := 0; i < b.N; i++ {
                r.Reset(data)
                str.fill(r, l)
                str.String()
        }
}

func BenchmarkLongString(b *testing.B) {
        b.StopTimer()
        var str String
        data := []byte(longStr)
        r := bytes.NewReader(data)
        l := uint(len(data))
        b.StartTimer()

        for i := 0; i < b.N; i++ {
                r.Reset(data)
                str.fill(r, l)
                str.String()
        }
}
```

测试结果如下：

```
$ go test -bench . -benchmem
goos: darwin
goarch: amd64
pkg: github.com/googollee/rtnx/rtmp/amf
BenchmarkShortString-4          50000000                25.1 ns/op
0 B/op         0 allocs/op
BenchmarkLongString-4           20000000                67.0 ns/op
48 B/op        1 allocs/op
```

```
PASS
ok      github.com/googollee/rtnx/rtmp/amf      3.189s
```

可以看到，短字符串在测试中确实没有再次分配内存，并且相比长字符串，性能有了显著提高。

之后，先是对这段代码直接翻译为Rust。但在翻译过程中遇到了不少问题。

首先遇到的问题，当然是所有权的问题！Go结构里的`buf`实际有两种引用：要么引用内部`data`的一段内存，要么引用一段分配的内存。而Rust，基本上没办法让一个struct在内部互相引用。Rust强制所有引用必须以树状组织所有权，而内部引用这种情况，相当于一个节点内引用自己，没办法形成树状所有权。所以需要将这个域拆成两个，一个表示在引用`data`时，需要引用多长的地址，另一个表示如果有必要，引用到一段分配的内存。第一次尝试如下：

```
pub struct String {
    data: [u8; 20],
    heap: Option<Box<[u8]>>,
    len: usize,
}
```

> *本来是想利用Rust的模版，将预先分配的内存长度参数化。结果试了很久，发现Rust还不支持常数模版，所以这里20的长度直接硬编码。好在后面的实现不会直接依赖20这个数，所以没有做进一步改动。*
>
> *Rust社区使用类似`String<[u8; 20]>`的方式来实现常数模版参数。这是一个workaround。目前Rust官方已经有常数模版的提案。*

其中`data`和之前一样，是预先分配的一段内存。`len`用来记录，在短字符串的情况下，使用了多长的`data`的内容。由于去掉了引用关系，所以

不会违反Rust的所有权。

但下一个问题，`heap`应该如何定义？Rust规定，所有的引用都必须指向相应的引用对象，不能出现没有引用的引用变量，因此`& T`是不能为空的。而我希望`heap`在正常情况下需要为空。因此这里使用`Option<_>`来表示一个要么为`None`，要么有值的类型。

接下来的`Box<_>`表示持有一个在堆上分配的类型。与Go不同，Rust非常明确的区分堆变量和栈变量，而Go通过编译器的逃逸分析，会自己决定一个变量是否需要放到堆上并在之后由GC回收。这体现了两种语言完全不同的思路：Go试图降低程序员需要了解的实现细节，提供尽量少但足够的基础设施，保证上手容易的同时保证性能；而Rust则暴露了大量的细节给程序员，并强迫程序员遵守约束，并使用这些约束保证程序正确。由于Rust暴露了实现细节，有经验的用户可以根据这些细节做更激进的优化。而Go很多时候受限编译器的能力无法实现（很多时候是不知道可以进行）这类优化。比如Go在读取链接时常会开一段临时内存进行存储：

```
func Handle(r io.Reader) {
  // ...
  var data [20]byte
  r.Read(data[:])
  // ...
}
```

其中`data`就是临时分配的内存。理论上，这段内存如果只在`Handle`中使用，只在栈上就可以，随着`Handle`退出自动释放。但是由于逃逸分析无法获知是否在`r.Read()`时持有了`data[:]`这个引用，这里会认为`data`有逃逸，所以会将其分配在堆上，并导致后面gc的参与。

回到Rust，最终，`heap`被定义为`Option<Box<[u8]>>`，表示一个可以为`None`，也可以是一个分配在堆上的`[u8]`。`[u8]`这个类型与Go的slice从

概念到实现都很类似，是对数组的一种引用。值得注意的是，`[u8]`由于无法在编译时预知引用的长度，所以无法直接在栈上创建一个`[u8]`的变量。

接下来是`fill()`方法：

```
pub fn fill(&mut self, r: &mut impl io::Read, n: usize) ->
io::Result<()> {
    let buf = if n > self.data.len() {
        let mut v = Vec::with_capacity(n);
        unsafe { v.set_len(n) };
        self.heap = Some(v.into_boxed_slice());
        match self.heap.as_mut() {
            Some(b) => b,
            None => panic!("should not here"),
        }
    } else {
        self.len = n;
        &mut self.data[0..n]
    };

    return read_full(r, buf);
}
```

Rust默认保证变量使用时必须做过初始化，而这里`Vec::with_capacity()`分配内存后，并不需要确定数据的内容，随后会填入具体数据，所以使用`unsafe { v.set_len(n) }`来强制使用该段未初始化的内存。值得注意的是，当使用`v.into_boxed_slice()`将这段内存的所有权交给`self.heap`后，需要有一种方法再取回这段内存的引用，并赋给`buf`变量。`self.heap`的类型是`Option<Box<[u8]>>`，按照习惯，需要按照`Option<Box<[u8]>> -> Box<[u8]> -> [u8] -> &mut [u8]`的顺序取得引用。但是这里有个问题，如果变成了`Box<[u8]>`，证明这是一个自己拥有生命周期的变量类型，而不

是一个引用。所以这里的取引用的顺序是`Option<Box<[u8]>> -> Option<&mut Box<[u8]>> -> &mut Box<[u8]> -> &mut [u8]`。而第一步就是通过`self.heap.as_mut()`来取得引用。

~~由于Rust没有官方的`io.ReadFull()`，所以这里做了自己的实现：~~经@upsuper提醒，Rust官方有类似的实现`std::io::Read::read_exact`。不过这里依旧使用自己的实现：

```
fn read_full(r: &mut impl io::Read, b: &mut [u8]) -> io::Result<()>
{
    let mut i = 0;
    while i < b.len() {
        match r.read(&mut b[i..]) {
            Err(b) => return Err(b),
            Ok(n) => {
                i += n;
            }
        }
    }
    Ok(())
}
```

`io::Result<()>`是很展现Rust特色的类型。这个类型是表示一个要么是`()`的类型值，要么是`io::Error<_>`的类型值。`()`类型是Rust里的某种元类型，某种程度上可以看作是`nil`类型或者表示不需要关心的返回值。`io::Error<_>`就很简单，是表示某种io的错误，具体错误类型由模版类型`_`决定。对比Go的类型：

```
func ReadFull(r io.Reader, n int) (int, error)
```

显然，Go的类型无法在语法上保证`ReadFull()`同时返回`int`和`error`。而同时返回两个值会造成调用者困惑：到底这是个正常返回，还是个错误？不要以为Go里常用`error`后置的约定，就不会出现这种情况。实际上Go标准库都无法防止这种情况出现：

```
When Read encounters an error or end-of-file condition after
successfully reading n > 0 bytes, it returns the number of bytes
read. It may return the (non-nil) error from the same call or
return the error (and n == 0) from a subsequent call. An instance
of this general case is that a Reader returning a non-zero number
of bytes at the end of the input stream may return either err ==
EOF or err == nil. The next Read should return 0, EOF.
```

而对这种类型错误的处理，简单处理可以直接`unwrap()`（后面会有展示），这里使用`match`是更精细的处理。由于`match`是个表达式而不是语句，所以可以使用下面的方法处理错误：

```
let ret = match function() {
  Err(err) => {
    // handle err
    return Err(err);
  },
  Ok(r) => r,
}
```

与Go的错误处理对比：

```
ret, err := function()
if err != nil {
  // handle err
```

```
  return err
}
```

Rust错误处理的优点是不会出现`err`变量四处定义的现象，不过Go代码量更少。不过，借助Rust的宏，如果不需要特别处理返回错误err，而是直接返回的话，Rust可以简写为：

```
let ret = function()?
```

这就显得比Go好看多了。即便是预期中Go2的错误处理都并不比这个好。

接下来是性能测试部分。我不知道Rust对于*工程*这件事情是怎么理解的，反正在我实验时，Cargo项目在stable分支支持`bench`命令，但却依赖nightly的一个testing包才能用。目前的Rust项目里充满了这种奇怪的，号称stable提供但却依赖nightly的特性，导致构建完整工程时十分痛苦。与Go自带非常稳定的`testing.B`特性相比，Rust这方面感觉一点都不像一个有着合理规划的号称稳定版三年之久的项目。

最终，使用Criterion项目完成整个性能测试，代价是性能测试代码与`String`定义代码分在不同目录，且没办法引用`String`类型的私有方法。这也是上面将`fill()`定义为`pub`的原因。测试代码如下：

```
#[macro_use]
extern crate criterion;
extern crate rtnx;

use criterion::Criterion;
use rtnx::rtmp::*;

fn string(c: &mut Criterion) {
    c.bench_function("short str", |b| {
```

```
        let mut str = String::new();
        let data: &'static str = "short str";

        b.iter(|| {
            str.fill(&mut data.as_bytes(), data.len()).unwrap();
            str.string();
        })
    });

    c.bench_function("long str", |b| {
        let mut str = String::new();
        let data: &'static str = "looooooooooooooooooonnnnnnnnnnnng
str";

        b.iter(|| {
            str.fill(&mut data.as_bytes(), data.len()).unwrap();
            str.string();
        })
    });
}
```

Rust本身为`&[u8]`类型实现了`Read`，所以可以直接将`data.as_bytes()`作为一个`Read`使用。由于`fill()`需求的是`&mut Read`，这里会使用`&mut data.as_bytes()`。而`'static`表示静态生命周期，也就是常量字符串。

测试结果：

```
$ cargo bench
    Finished release [optimized] target(s) in 0.27s
     Running target/release/deps/rtnx-b3fb2c92c52ff0ff

running 1 test
test rtmp::amf::string_test::test_string ... ignored
```

```
test result: ok. 0 passed; 0 failed; 1 ignored; 0 measured; 0
filtered out

     Running target/release/deps/rtnx-b073d7725e43e2cb

running 1 test
test rtmp::amf::string_test::test_string ... ignored

test result: ok. 0 passed; 0 failed; 1 ignored; 0 measured; 0

filtered out

     Running target/release/deps/string-e1abdea77166d91b
Gnuplot not found, disabling plotting
short str               time:   [8.8030 ns 8.8762 ns 8.9604 ns]
                        change: [-2.0804% +0.0430% +1.9924%] (p =
0.96 > 0.05)
                        No change in performance detected.
Found 12 outliers among 100 measurements (12.00%)
  8 (8.00%) high mild
  4 (4.00%) high severe

long str                time:   [29.893 ns 30.005 ns 30.156 ns]
                        change: [-0.1391% +0.6297% +1.4001%] (p =
0.12 > 0.05)
                        No change in performance detected.
Found 6 outliers among 100 measurements (6.00%)
  3 (3.00%) high mild
  3 (3.00%) high severe

Gnuplot not found, disabling plotting
```

与之前Go的测试结果对比：

语言	短字符串	长字符串
Rust	8.8762ns	30.005ns
Go	25.1ns	67.0ns

Rust在两种情况都只用了Go不到一半的时间。Rust在性能方面确实完胜Go。

一些上面没有提到的事情。

首先是并发。Go自带并发，可以简单使用`go/chan`来做到高效且资源消耗小的并发。Rust语言层面没有这个支持，但是凭借灵活的类型系统，通过Future类型为并发提供了基础设施。在此之上，tokio实现了类似goroutine的一套并发机制。由于Go的类型系统过于简单，是没办法以库的方式实现并发的。这一点，Rust做的非常漂亮。

关于包，与Go基于目录的package相比，Rust就充满了没有必要的复杂性。Rust里通过`mod name`可以引入另一个编译单元name，这个单元可以是同目录下的文件`name.rs`，也可以是子目录下的文件`name/mod.rs`。而`mod`本身还会生成一个`name`的命名空间，如果要引用包里的符号，要么通过`use`引入符号，要么带着`name::`作为符号前缀给出命名空间。这导致想将一个包里的代码分到不同文件时，变得异常复杂。基本上，想要做到拆分文件又能互相引用，需要写类似下面的代码：

```
mod part1;
mod part2;


pub use self::part1::*;
pub use self::part2::*;
```

而这还会导致`part1`和`part2`里的私有符号互相不可见。`mod/use`这一特性看似灵活，实际最终会强制一个包的内容全部写到一个文件里。与Go自然的“一个包一个目录”的形式相比，理解和使用都会困难很多。而Rust 2018也试图改进这个问题，细节可以查看Path clarity。

Rust对编译单元的控制，是依靠宏指令来完成的。比如如果说一部分代码只在测试时编译，就需要写：

```
#[cfg(test)]
mod tests {
  // ...
}
```

这样`mod tests`内部的代码就只会在`test`时才做编译。而Go使用文件后缀来做这种区分。比如`xxx_test.go`默认只在测试是编译。这里我认为Go的方式更自然。一个包里有哪些编译控制，哪个文件会在哪个状态用到，直接看文件名就能知道。类似`xxx_unix.go/yyy_windows.go`也为维护提供了方便。Rust如果没有合适的工具的话，就只能一个一个文件查看，才能知道编译规则。

说道工具，Rust工具链的质量和Go相比，可以用惨不忍睹来形容。上文提到的性能测试就是一例。Rust大量工具目前处于preview或者只提供nightly的地步（好像最近clippy刚刚进入到stable的preview状态）。Rust社区希望2018能够提供一个稳定生产力的版本Rust 2018，不过现在2018都已经过半，还有一部分语法工作没有合并到stable。~~这不得不想起当年C++ 98拖到C++ 00再拖到C++ 0x。希望Rust能在发布语言版本的时候，考虑一部分核心工具链的同时发布。~~Rust社区将2018版本的发布时间定在10月，目前各个项目与预期进度相符。希望在今年底能看到更具生产力的稳定版本，不仅仅是语言的特性，还包括关键工具链。

至于工程项目，cargo做的很好。Go目前的`GOPATH`模式，大概只有Google自己觉得好。好在Go modules已经箭在弦上，马上就不用忍受这么难用的功能了。

最后的总结。

Rust和Go是完完全全两个思路下的语言。Go会尽力优化开发流程，减少暴露给程序员的概念，并有足够稳定的接口。而Rust则将所有特性暴露给程序员做选择，语言社区只负责核心编译器，将最佳实践固化为语言特性，工具链交由社区进行完善。

我建议所有的Go程序员去尝试一下Rust，体会一下编译器如何强制要求理解程序的细节。这些细节是Go试图隐藏的复杂性，但是理解这些细节对编写程序，哪怕是Go程序，都是很有帮助的。另一方面，对于公司，如果是为了建立一个“铁打的营盘”，组建一个不同技能等级的团队，充分利用已有的社区力量，简化工程复杂性的话，Go是目前更好的选择。

Googol Lee

多年生软件工程师，信仰开源

Munich, Germany *http://air.googol.im*