错误模型

原文：*The Error Model*

翻译: @dontpanic

版本: 20220127 (4053495)

在线阅读：*https://tuesday.dontpanic.blog*

1 错误模型简介

译注：

原作者 Joe Duffy 曾在微软参与开发一款操作系统 Midori，这是一款研究型/孵化型项目。这款操作系统主要由一种 C# 的变种语言（有人称作 M# 语言）编写。

下文中所有的 “我”、“我们” 均指代原作者 Joe Duffy 或其所在团队。作者总结了目前主流编程语言中常见的错误模型的优缺点，同时分别给出了自己的针对不可恢复错误（Unrecoverable Error）和可恢复错误（Recoverable Error）的处理方案。由于本文有一些惯用语和专业词语，受限于我的水平，可能出现翻译错误、或措辞与主流方案不同等问题，欢迎指出以便修正。本文较长，请做好长时间阅读的准备。

原文链接：http://joeduffyblog.com/2016/02/07/the-error-model/

Midori 使用了一种基于 C# 的、支持 AOT 编译、类型安全的语言。除了我们的微内核，整个系统都是使用这种语言编写的，包括驱动程序、域内核（Domain Kernel），以及全部的用户代码。我在这段时间里收获了很多，现在是时候总结一下了。整个语言涵盖的东西太多了，我会分成几篇文章来阐述，就先从错误模型开始。错误（Errors）的传递与处理在任何编程语言中都是非常基础的部分，而对于用来编写高可靠操作系统的语言来说更是如此。就像 Midori 项目的其他部分一样，任何修改一部分都应该从全局的角度来考量，并进行不断地迭代。我经常从原来的同事那里听到说，错误模型是他们在 Midori 上开发程序时最怀念的部分。我也很怀念这部分。那么废话少说，我们这就开始。

1.1 错误模型简介

错误模型需要回答的最基本的问题是：“错误” 应该如何传达给程序员和用户？这问题似乎很简单。

要回答这个问题，最大的阻碍之一是：如何定义什么是 “错误”。很多语言把 Bug 和可恢复错误归为一类，用同样的机制来处理，比如把空指针引用错误、数组访问越界错误与网络连接错误、语法分析错误混为一谈。这种一致性初看起来很美好，但它有着深层次的问题——这种一致性会造成很大误解，并且通常会导致程序员写出不可靠的代码。

总的来说，我们的解决方法是同时提供两套错误模型。一方面，对于程序的 Bug，我们提供了快速失败（Fail-Fast）模型，在 Midori 中我们称其为放弃（Abandonment）；另一方面，对于可恢复的错误，我们也提供了静态受检查异常（Statically Checked Exception）。这两种错误模型从编程模式到背后的机制都截然不同。放弃会无条件地立即终止整个进程，不会再运行任何用户代码（需要说明的是：一个典型的 Midori 程序会由很多个小的、轻量级的进程组成）；异常则允许恢复用户代码的执行，在这个过程中类型系统会起到重要的检查和验证的作用。

我们的旅程漫长而曲折。为了讲好这段故事，这篇文章分为以下 6 个主要部分：

- 野心和经验
- Bugs 不是可恢复错误！
- 可靠性、容错性和隔离性
- Bugs：放弃（Abandonment）、断言（Assertions）和合约（Contracts）
- 可恢复错误：类型导向的异常回顾与总结

现在看来，有些结论似乎很明显，尤其是在更现代的系统语言（比如 Go 和 Rust）出现之后。但是一些结论还是让我们很惊讶。我会尽量省略废话，但也会提供足够的背景故事。我们走过很多弯路，但我觉得这些弯路甚至要比最终的结论更有趣。

2 野心和经验

这是一篇译文。原文链接：http://joeduffyblog.com/2016/02/07/the-error-model/

让我们首先来看看现有的系统，总结一下我们需要哪些东西，并从中学习一些架构方面的经验。

2.1 准则

首先，我们需要定义什么是一个好的错误模型：

- 易用性：在开发人员面对程序错误的时候，他们必须能够非常方便、甚至不加思考地做出“正确的事”。一个朋友（也是同事）说这就叫做“好坑（The Pit of Success）”。错误模型不能为编码工作增添大量的额外负担。正常情况下，开发人员应该对它非常熟悉。
- 可靠性：错误模型是整个系统可靠性的基础。毕竟我们正在编写一个操作系统，可靠性是最重要的。有人可能会觉得我们太过于重视这一点了。我们提倡“构建正确”（Correct by Construction），很多编程模型的开发都基于这个理念。
- 性能：在大多数情况下，我们都希望代码能够足够快。也就是说，在没有发生错误的时候，我们需要尽可能地减少性能损失。而对于发生错误的时候，任何额外的性能开销必须是“按需付费”。与很多愿意在发生错误时过度损失性能的现代系统不同，我们有一些对性能非常敏感的组件，当发生错误时它们也需要足够快才行。
- 并发：我们的整个系统都是高并发、分布式的。对于其他的错误模型来说，这通常是事后才考虑的事情，但我们需要提前正视这个问题。
- 诊断：不论是交互式调试还是事后调试，都需要简单而高效。
- 协调：从根本上讲，错误模型是一种编程语言的功能，是开发人员表达代码逻辑的重要部分。因此，错误模型必须与系统中的其他功能协调一致。不同组件之间的集成必须是自然的、可靠的，结果是可预测的。

这些要求也许有些严格，但我认为最终我们在所有的方面都取得了成功。

2.2 经验

现有的错误模型并没有满足我们上面提到的要求，至少没有全部满足。通常某个错误模型可以在一个方面做的很好，但在另一个方面却很差。比如，错误码可以有很高可靠性，但很多程序员发现错误码在使用时很可能会出错。具体来说，它很容易导致程序员做出错误的事，比如忘记检查错误码。这明显违背了“好坑”原则。

考虑到我们寻求的是最高等级的可靠性，大多数模型都不能满足我们的需求也不足为奇了。

相比于可靠性，如果易用性是更优先的目标（例如你在使用一门脚本语言），那么结论可能会完全不同。Java 和 C## 之类的语言的纠结之处在于，它们的使用场景可能很复杂——有时候用于系统编程，有时候用于应用编程。总的来说，它们的错误模型很不满足我们的需求。

最后需要说明的是，我们的故事开始于十几年前，那时还没有 Go、Rust、Swift 等语言供我们参考。这三门语言在错误模型方面取得了很大的进步。

2.2.1 错误码

错误码大概可以说是最简单的错误模型了，它背后的原理非常基础，甚至不需要语言或运行时的支持。函数返回一个值，通常来说是一个整数，来表明成功或失败。

```
int foo() {
    // 执行一些语句
    if (failed) {
        return 1;
    }
    return 0;
}
```

这就是很一种典型的模式，返回 0 表示成功，非零值表示失败。调用时必须检查返回值：

```
int err = foo();
if (err) {
    // 出错了！赶紧搞定
}
```

大多数系统支持使用常量来表示一组错误码，而不是直接使用具体的数字。也可能会有一些函数用于获取最近一次错误的详细信息（比如 C 语言中的 errno，以及 Win32 中的 GetLastError）。返回一个错误码确实不是什么特殊的事——就是返回一个值。

C 语言一直在使用错误码。因此，大多数基于 C 语言的生态系统也用错误码。很多底层系统的代码也使用了返回错误码的方法，Linux 如此，不计其数的关键任务系统、实时系统也如此。公平地说，错误码模型有着深远的历史和广泛的影响。

在 Windows 上，`HRESULT` 也是一样，它就只是一个整数“句柄”，还附带了一堆常量和宏，例如 `S_OK`、`E_FAULT`、`SUCCEEDED()`，定义在 `winerror.h` 里面。这些常量和宏用来创建或检查返回值。Windows 最核心的部分都使用了返回错误码的方式。内核里面不会使用异常，至少不会刻意去使用。

在一些需要手动管理内存的环境下，发生错误时释放内存尤其困难。考虑到这种情况，返回错误码的方式相对可以接受。C++ 使用 RAII 来自动解决这个问题，但是除非你非常认同整个 C++ 的编程模型（很多系统程序员并不），在 C 语言中没什么好的方法来添加上 RAII 的功能。

Go 语言选择了使用错误码。虽然 Go 的方式与 C 很像，但它提供了更加现代化的语法和库的支持。

很多函数式编程语言也使用返回码——它们可能被掩盖在 Monad、`Option<T>`、`Maybe<T>`、或是 `Error<T>` 里面。它们与数据流、模式匹配相结合，使用起来更加自然。这种方法解决了错

误码的很多弊端（我们会在下文提到），尤其是与 C 相对比更是如此。Rust 大范围地采用了这个模型，同时也提供了令很多系统程序员非常欣赏的新功能。

尽管错误码具有非常简洁的优势，它也会带来一些包袱，比如：

- 性能可能会受到影响。易
- 用性可能会很差。
- 最重要的一点：你可能会忘记检查错误码。

我们会结合上面提到的那些语言逐条讨论。

2.2.1.1 性能 错误码并不满足 “对于大部分场景提供性能零负担，对于小部分场景按需’ 付费’” 的标准：

1. 对调用约定的影响。现在，对于非 void 返回值类型的函数来说，你需要返回两个值：一个是真正的返回值，一个是可能发生的错误。这就消耗了更多的寄存器和/或栈空间，让函数调用变得不高效。当然，对于可以内联的函数来说，内联可以帮忙解决一些问题。
2. 当被调用方可能产生错误的时候，调用方需要有很多的分支来进行处理。我把这种开销叫做 “花生酱”，因为分支检查随意分布在代码中，很难直接测量它的影响。在 Midori 中，我们曾做过实验，并且确认了这些分支确实会带来性能损失。分支太多还会带来另一方面的影响——编译器中的优化器可能会被这么多的分支搞蒙。

也许对很多人来说这个结论有点让人惊讶，毕竟每个人都一定听说过 “异常很慢”。事实证明，未必如此。而且，在使用正确的情况下，异常会让错误处理代码和数据与热路径（Hot Path）分开，这就增加了 L 缓存和 TLB 的性能。而在上述错误码的模型中，L 缓存和 TLB 的性能会明显受到影响。

你可能会觉得我在吹毛求疵，对这种模型太过苛刻，毕竟很多高性能系统都在使用错误码模型。然而，下面我们将看到错误码模型的更严重的问题。

2.2.1.2 忘记检查错误码 开发人员非常容易忘记检查函数返回的错误码。比如，有这样一个函数：

```
int foo() { ... }
```

在调用时，如果我们默默地忽略返回值并且继续执行呢?

```
foo();
// 继续执行 ——但是 foo 可能已经失败了!
```

在这个时候，你已经掩盖了程序中的一个潜在关键错误。这是错误码最烦人、也是后果最严重的问题。后面我会提到，一些函数式语言使用了 “Option” 类型来解决这一点不足。但是对于基于 C 的语言，甚至对于具备现代语法的 Go 来说，这是一个确实存在的问题。

我不是在纸上谈兵。我遇到过无数的 Bug，都是由于忽略了返回值引起的，我相信你也遇到过。事实上在使用错误码模型时，我的团队同事遇到了一些很有趣的 Bug。比如，当时我们在往 Midori 上移植微软的语音服务（Microsoft’s Speech Server），结果发现 80% 的繁体中文请求会失败——不是那种立刻就抛出来的错误，而是表现为客户端收到了一些垃圾数据。一开始我们以

为是我们自己的问题，但最终我们发现是原始代码中忽略了 `HRESULT` 返回值。当移植到 Midori 上时，这个 Bug 就突然出现了。这一段经历佐证了我们关于错误码的观点。

对于 Go 语言，我有点小惊讶。Go 语言认为未使用的 `import` 是一种错误，但居然在返回码这种关键得多的地方不做要求。太可惜了！

当然，你可以使用静态分析器，或者在编译时给出一个“未使用的返回值”的警告（很多商业 C++ 编译器都有支持）。但是如果它不是一门语言的核心功能、作为一项强制要求，由于代码分析的噪声存在，这些方法都不会起到根本性的作用。

由于这些原因，在我们的语言里，没有使用的返回值是一种编译错误。你需要显示地忽略他们。一开始我们我们使用了 API 来搞定这个问题，但是最终我们还是设计了专门的语法——等同于 `>/dev/null`：

```
ignore foo();
```

虽然我们不使用错误码模型，但无法隐式忽略一个函数的返回值对于提高系统的整体可靠性来说非常重要。你有多少次调试之后才发现，根本问题只是在于忘记使用了返回值？当然，让程序员必须使用 ignore 并没有真正解决问题，他们还是可能会做错事。但是这至少需要明确地说出来，并且便于代码审查。

2.2.1.3 编程模型的易用性 在基于 C 的、使用错误码的语言中，你会发现在调用函数之后需要很多 `if` 检查语句。C 语言程序中，由于申请内存失败也是采用了返回错误码来表示，因此很多函数都可能会返回错误信息，这让写那些 if 变得更没意思。而如果需要返回多个值，也会让代码变得更臃肿。

提示：这是我个人主观的想法。其实有很多方法，可以让你优雅地使用错误码。不过你所能用的只是一些非常简单的结构——整数，`return`，`if` 分支，它们也会在其他地方使用。以我的拙见，错误处理的重要性足以让它在编程语言特性当中占有一席之地，编程与语言应当辅助你完成这些工作。

Go 提供了一种语法捷径，能够是标准的错误码检查 *稍微地* 优雅一些：

```
if err := foo(); err != nil {
    // 错误处理
}
```

我们在一行代码中实现了调用 foo、并检查错误是否非 `nil`。非常简洁。

然而，错误码在易用性方面并不止存在上述这些问题。

很常见的是，一个函数通常会共享很多错误恢复和补救的逻辑。很多 C 语言程序员使用标签和 goto 来组织这些代码。比如：

```
int error;

// ...

error = step1();
if (error) {
```

```
    goto Error;
}

// ...

error = step2();
if (error) {
    goto Error;
}

// ...

// 函数正常退出
return 0;

// ...
Error:
// 根据`error`值进行一些额外处理
return error;
```

不用说，这样的代码天下间只有母亲才可能会喜欢。

在一些语言中，例如 D、C#、Java，`finally` 代码块可以用来直接实现这种 “在作用域结束之前执行” 的模式。类似的还有微软的 C++ 私有扩展 `__finally`，即便你不认同 RAII 和异常，也可以单独使用这个扩展。D 语言提供了 `scope`，Go 语言提供了 `defer`。所有的这些都是在帮助从根本上去除 `goto Error` 模式。

接下来考虑一下，如果我的函数想要同时返回一个值以及可能发生的错误，怎么办？我们已经占用了返回值，所以下面是两种显而易见的方案：

1. 我们可以使用返回值返回两个值当中的一个（通常是错误码），而另一个（通常是真正的返回值）则通过其他方式——例如指针参数——进行返回。这在 C 语言中非常常见。
2. 我们可以返回一个数据结构，这种结构既可能携带真正的返回值，也可能携带错误码。但是像是在 C 语言、甚至是 Go 语言中，由于没有参数多态，返回值会丢失类型信息，因此这种做法不常见。C++ 有模板，因此理论来说它是可以做到的，但由于 C++ 支持异常，很少有围绕返回错误码构建的生态。

结合上文中提到的性能问题，想一想这两种方法对你的程序编译后产生的汇编代码有什么影响。

2.2.1.4 从旁路返回的返回值

第一种方案，用 C 语言来实现的话，就像这样：

```
int foo(int* out) {
    // 执行一些操作
    if (failed) {
        return 1;
    }
```

```
    *out = 42;
    return 0;
}
```

返回值只能从旁路返回，让调用的代码显得很笨：

```
int value;
int ret = foo(&value);
if (ret) {
    // 出错了!
}
else {
    // 使用 value...
}
```

除此之外，这种模式还会对编译器的确定性赋值分析造成干扰，例如可能会影响编译器无法给出“使用未初始化的变量”之类的警告。

针对这个问题，Go 同样提供了很漂亮的语法，得益于多返回值：

```
func foo() (int, error) {
    if failed {
        return 0, errors.New("Bad things happened")
    }
    return 42, nil
}
```

调用方也因此变得简洁很多。与上面提到的单行 if 检查将结合的话——虽然有点奇怪，因为初看下面的代码可能会以为 value 不在作用域中，但它确实在——错误检查也变得更优雅：

```
if value, err := foo(); err != nil {
    // Error!  Deal with it.
} else {
    // Use value ...
}
```

值得注意的是，这同样有助于提醒你检查错误码。当然这也没有从根本上解决这个问题，因为函数可能会只返回错误码，这时就和 C 语言一样，很容易忘记检查。

就像上面提到的一样，一些人可能会在易用性这一点上反对我。我猜尤其是 Go 语言的设计师们。其实，Go 语言使用错误码这一点非常吸引我，它就像是对当今世界过于复杂的语言的一种反叛。我们已经丢失了 C 语言的优雅性——通常看到一行 C 语言代码，我们就能够猜出它被翻译成机器码后是什么样子的。我不会反对这些观点。事实上，相比于不受检查的异常和 Java 典型的受检查异常，我更喜欢 Go 语言的模型。最近我也写了很多 Go 代码，在我写下这篇文章的时候，想起 Go 的简洁性，我甚至在反思：Midori 是不是在 `try` 和 `require` 的道路上走的太远了？我不知道。Go 的错误模型似乎是这门语言中最具争议的一部分，也许很大程度上是因为你不能像在其他语言里那样草率地对待错误。但是，Midori 也不允许程序员草率地对待错误，可是程序员们确实很喜欢在 Midori 上编程。所以我很难做出比较。我只能确信这两种方式都能够写出可靠的代码。

2.2.1.5 使用数据结构包裹返回值 使用数据结构返回函数值函数式编程语言通常这样来解决易用性的问题：一个数据结构，既可能包含了函数返回值，也可能包含了错误码。在调用者使用返回值的时候，你必须把这个数据结构拆开，检查是否有错误，才能继续使用函数的返回值。这对多亏了数据流的编程风格。这样就很容易地避免了忘记检查错误这类严重的问题易用

举一个现代语言的例子，你可以看看 Scala 的 `Option` 类型。不过很遗憾的是，一些语言（例如 ML 家族，以及 Scala —— 这得归咎于 JVM）把这种优雅的模型跟不受检查的异常混合在了一起。这破坏了使用数据结构返回数据的优雅性。

Haskell 甚至更酷，它使用了错误值和局部控制流来给人一种 “异常处理” 的假象：

> C++ 程序员在异常或错误返回代码谁对谁错的问题上存在着一个古老的争论。Niklaus Wirth 认为异常就是另一种 GOTO，因此在他的语言中不支持异常。Haskell 用一种折中的方式解决了这个问题：函数返回错误码，但是错误码的处理不会使代码变丑。

（译注：Niklaus Wirth，Pascal 之父，1984 年图灵奖获得者。）

这里的技巧在于，Haskell 仍然支持熟悉的 `throw` 和 `catch` 模式，但是使用 Monad 而不是控制流来实现。

虽然 Rust 也使用错误码，但它提供了一种函数式的错误类型。例如，在 Go 语言中，我们有一个 bar 函数：我们在其中调用 `foo` 函数，并且简单地把 foo 的错误传播回调用者：

```
func bar() error {
    if value, err := foo(); err != nil {
        return err
    } else {
        // Use value ...
    }
}
```

在 Rust 中，如果想写的比较长，那么与 Go 相比并没有更加简洁。下面的代码对于 C 语言程序员来说可能像是在看外语，因为我们使用了模式匹配的语法（这是个问题，但不那么严重）。如果你熟悉函数式编程，那么你应该马上就能理解这段代码，它会时刻提醒你需要处理错误：

```
fn bar() -> Result<(), Error> {
    match foo() {
        Ok(value) => /* Use value ... */,
        Err(err) => return Err(err)
    }
}
```

不过 Rust 做的更好之处在于：它提供了一个 `try!` 宏，减少了不必要的繁文缛节，上面那一堆代码就变成了一句：

```
fn bar() -> Result<(), Error> {
    let value = try!(foo);
    // Use value ...
}
```

这简直就像是个世外桃源。不过确实，这种方法仍然存在我们提到的性能问题，但它在所有其他的方面都做得非常好。此外，只有这一种模型还不够——我们还需要快速失败（Fail-Fast），即“放弃”（Abandonment）——不过就像我们马上会介绍的，Rust 的方式比其他任何广泛应用的、基于异常的模型都要好。

2.2.2 异常

异常的历史有点意思。在这段旅途上，我花费了不计其数的时间跟随工业界的步伐重新走了一遍，包括读了一些古老的论文——比如 1975 年的经典：Exception Handling: Issues and a Proposed Notation。除此之外我还参考了几种其他语言的实现方式：Ada、Eiffel、Modula-2 和 3、ML、以及最受启发的，CLU。很多论文对这段漫长而艰辛的历史的总结要比我讲的好，所以我就不在这里多说了。我会主要讲述为了构建高可靠的系统，哪些方法好用、哪些方法不行。

在开发错误模型的时候，可靠性是上面列举的几个需求当中最重要的。如果你不能合理地处理故障（Failure），根据定义，你的系统就称不上可靠。正常来说，操作系统需要可靠，然而最常见的模型——不受检查的异常——在这一方面差无可差，不能再差。

由于这些原因，大多数高可靠的系统使用返回码来取代异常。这使得调用者能够进行局部分析并根据不同情况分别应对发生的错误。

2.2.2.1 不受检查的异常（Unchecked Exceptions）

快速回顾一下。在一个不受检查的异常模型中，你可以 throw 或是 catch 异常，而异常并不是类型系统或函数签名的一部分。例如：

```
// Foo 会抛出一个未处理的异常：
void Foo() {
    throw new Exception(...);
}

// Bar 调用 Foo，并且处理了那个异常：
void Bar() {
    try {
        Foo();
    }
    catch (Exception e) {
        // Handle the error.
    }
}

// Baz 也调用了 Foo，但是没有处理那个异常：
void Baz() {
    Foo(); // 异常会继续抛给调用者
}
```

在这个模型里，任何函数调用——有时甚至是任何 *语句* ——都可以抛出一个异常，把控制流导向了不知道什么地方。没有任何标注、或是类型系统的数据能够辅助分析。所以结果是，在抛出异常的时候，任何人都很难分析出程序的状态，也很难说清在异常向调用栈上方抛出的时候程

序的状态如何改变（在一个并发程序中，这可能会跨越线程之间的鸿沟），同样很难说清异常是否会被 catch，以及它被 catch 时的状态。

不过，分析程序状态还是可以一试的。这需要去读 API 的文档、做一些代码审计、严重依赖于 Code Review，并且还需要一些运气。编程语言不会给你提供丝毫帮助。由于故障很少发生，这种模型似乎并不像它听起来那么烂。我的结论是：这就是为什么在工业界很多人认为不受检查的异常 “足够好”，因为这些东西在执行正常的情况下不会映入你的眼帘，而且由于大多数人在非系统程序里不会编写非常健壮的错误处理程序，这使得 通常情况下，直接抛一个异常能够使你最快地摆脱困境。Catch 住异常然后继续执行的话，程序往往也能正常工作。无害就不受罚。从统计上来讲，程序能 “工作”。

也许统计意义上的 “正确” 对于脚本语言来说是 OK 的，但对于最底层的操作系统来说，或者对于任何关键任务程序和应用来说，它不是一个合适的方案。我相信这没什么争议。

由于异步异常的存在，.NET 的处境非常糟糕。C++ 也有着所谓的 “异步异常”：这些故障是由硬件错误出发的，例如非法访问。然而在.NET 里，这一部分肮脏不堪：任何一个线程都可以让代码中的任何地方发生故障（Failure），甚至可以在一个赋值语句的左右操作数之间！所以在源代码中看起来像是原子操作的地方，其实不是。关于这个话题，我在 10 年前阐述过，但如今它仍然存在。不过这个问题已经被不那么严重了，因为.NET 终于逐渐认识到线程终止是有问题的。新的 CoreCLR 甚至不再有 AppDomain，新的 ASP.NET Core 1.0 也不会在像以前那样终止一个线程了。但是相关的 API 仍然存在。

C## 的主设计师 Anders Hejlsberg 有一段非常有名的采访，叫做 The Trouble with Checked Exceptions。从一个系统程序员的角度来看，大部分的论断都会让你抓耳挠腮。下面这一段话足以证明 C## 的目标群体是 RAD 开发人员了：

> Bill Venners：但是即便在一个使用不受检查的异常模型的语言里，你不是也同样 break 了他们的代码吗？如果 foo 的新版本抛出了一个新的异常，使用者应该知道他们需要来处理这个新异常。他们的代码不也是被在写代码时没有纳入考量的异常破坏了吗？
>
> Anders Hejlsberg：不。因为在大多数情况下，人们不在乎。他们不会处理任何异常。在消息循环的最底层，有一段错误处理程序，这段程序会弹出一个框框说 “XX 出错了” 然后继续执行。程序员们总是随意地写上 try finally，所以当异常发生后还是可以保持程序的正确性，但他们真的对处理异常不敢兴趣。

这让我想起了 Visual Basic 里面的 `On Error Resume Next`，以及 Windows Forms 自动捕捉并忽略应用程序的异常然后尝试继续执行。我不是在指责 Anders 的观点；相反，鉴于 C## 的广泛流行，我确信在那个时间、那个环境，这是一个正确的决定。但是它显然不能用来编写一个操作系统。

C++ 至少还尝试过通过 `throw` 异常规范（throw exception specifications）来改善不受检查的异常。不幸的是，这个功能依赖于动态执行，这就直接给这个功能敲响了丧钟。

如果我有一个函数 `void f() throw(SomeError)`，`f` 的函数体仍然可以调用抛出其他异常的函数。同样，如果我使用 `void f() throw()` 声明了 `f` 不会抛出异常，他仍然有可能调用会抛出异常的函数。为了实现声称的合约，编译器和运行时必须确保：如果这种情况发生，需要调用 `std::unexpected` 以确保进程终止。

我不是唯一一个发现这个设计是个错误的人。throw 已经被抛弃了。WG21 文件 Deprecating

Exception Specifications 详细描述了 C++ 是如何在这条路上停下来的。下面是这份文件的开篇陈述：

> 异常规范已经被证明了在实践中几乎毫无用处，同时还为程序增加了可见的负担。

作者列举了 3 条放弃支持 `throw` 的原因，其中两条都与动态执行有关：运行时检查（以及与其相关的不透明的失败模式）和运行时的性能负担。第三条原因是，缺乏与普通代码的协调性，这应该通过合适的类型系统来处理（当然需要付出一定的代价）。

然而，解决这个问题的手段却依赖于另一个动态构建的概念—— `noexcept` 标识符——而它在我眼里，跟上一个问题一样糟糕。

“异常安全” 是一个在 C++ 社区经常讨论的话题。它清晰地从调用者的角度根据调用会产生的影响对函数进行划分，包括是否会失败、程序状态的转移以及内存管理。函数调用分为四种类型：*no-throw* 意味着之前的程序状态不会改变，而且不会有异常发生；*strong safety* 意味着状态的修改是原子性的，失败不会造成状态部分修改或是破坏不变量（Invariant）；*basic safety* 意味着虽然这个函数可能会部分地修改程序状态，但不变量还是会保证正确，同时也不会发生内存泄漏；最后，*no safety* 意味着一切皆有可能。

（译注：不变量（Invaraint）指的是某些程序状态，它们应当始终满足约定的条件。例如，如果约定了栈顶指针始终指向栈顶，no-throw、strong safety、basic safety 函数调用都应当能够保证不论发生什么情况，栈顶指针仍然指向栈顶。下文还会有相关内容。）

这种分类方法非常好，如果你希望严格要求一个函数在发生错误时的行为，我建议你使用这种或是类似的方法对函数进行分类。就算你在使用错误码模型，这种分类也是有好处的。不过问题在于，当一个系统使用了不受检查的异常模型时，几乎不太可能遵循上述的准则。只有当一个函数处在叶子节点的位置，只调用了一些小的、便于审计的函数时，这些准则才可行。想想就知道了：为了能在所有地方都确保 strong safety，你需要考虑所有的函数调用抛出异常的可能性，然后把这段代码安全地保护起来。这通常意味着编程时需要小心谨慎、信任英语文档（也就是说计算机无法检查）、只调用 noexcept 函数，或是心中默默祈祷。RAII（以及更现代的智能指针）能够帮助我们解决内存泄漏问题，因此实现 basic safety 可以相对轻松一点。然而，避免破坏不变量仍然是一个麻烦事。这篇文章 Exception Handling: A False Sense of Security 很好地总结了这这些问题。

对于 C++ 来说，真正的解决方案非常容易想到，而且也很直接：对健壮的系统程序来说，不要使用异常。这是 Embedded C++ 所采用的方案，也是无数实时系统和关键任务系统的 C++ 准则，例如 NASA 的喷气推进实验室所也采用这种方案。C++ on Mars sure ain’t using exceptions anytime soon。

如果这样的话，假设你可以安全地避免使用异常，并且像 C 语言一样只返回错误码，有什么不好吗？

整个 C++ 的生态都在使用异常。为了遵循上面提到的方案，你必须避免使用异常——这是这门语言中非常重要的部分，也就导致了无法使用 C++ 生态环境中的很多库。想使用 STL？不行，它用了异常。想用 Boost？不行，它用了异常。你的内存分配器也很可能会抛出 bad_alloc 异常，等等。这导致了很多人都会 fork 一套已有的库，然后魔改去掉异常。例如 Windows 内核就有一套它自己的不使用异常的 STL。生态系统的这种分歧既让人难受，又难以为继。

这简直就是一团乱麻，相当多的语言都使用了不受检查的异常模型。结论很明显，这些语言不适合编写底层、高可靠的系统代码。（我这么坦率肯定会得罪很多 C++ 程序员。）在 Midori 上写

了很多年代码之后，再回过头来使用不受检查的异常模型来写程序，对我简直就是一种折磨。不过值得“庆幸”的是，还有 Java 这种使用受检查异常模型的语言，也许我们可以从中学到点什么？

2.2.2.2 受检查的异常（Checked Exceptions） 啊哈，受检查的异常。它就像一个碎布娃娃，每个 Java 程序员、甚至非 Java 程序员都想凑上去揍他一顿！不过在我看来，这不太公平，毕竟还有个不受检查的异常在那垫底呢。

在 Java 中，你能知道一个方法所能抛出的大部分异常，因为它必须显示地声明出来：

```
void foo() throws FooException, BarException {
    ...
}
```

这样的话，`foo` 的调用者就能够知道它可能会抛出 FooException 或者 BarException。程序员在调用它时必须选择：1）将这些异常原封不动地再抛给上层；2）catch 住这些异常然后处理；3）把这些异常转换为其他异常然后再抛出（很有可能会“不小心”抹掉异常的类型）。比如：

```
// 1) 原封不动地抛给上层:
void bar() throws FooException, BarException {
    foo();
}

// 2) Catch 住然后处理
void bar() {
    try {
        foo();
    }
    catch (FooException e) {
        // Deal with the FooException error conditions.
    }
    catch (BarException e) {
        // Deal with the BarException error conditions.
    }
}

// 3) 把异常的类型转换一下然后抛出:
void bar() throws Exception {
    foo();
}
```

这与我们所期望的模型很接近了。但是，Java 的模型仍然在某些方面有所不足：

1. 异常也用来传递不可恢复的 bug，比如空指针解引用、零除错误，等等。
2. 其实你没办法真正知道一个方法能抛出的所有异常。这得归功于我们的 RuntimeException 小朋友。由于 Java 用异常来处理所有的错误（包括 Bug），语言的设计者们已经察觉到了大家会对这些异常的显式说明感到恼怒。所以他们就引入了一类不受检查的异常，也就是说，一个方法可以不加说明地抛出这些异常，调用者调用这个方法时也可以不做任何修改。

3. 尽管异常类型是函数签名的一部分，在方法调用时却没什么东西来指明这个方法是否会抛出异常。

4. 大家都烦它。

最后一点很有意思。在后面介绍 Midori 的错误模型时，我们会再回来回顾这一点。简单来说，人们对 Java 的受检查异常模型的厌恶感来自于前面那 3 点。最终的结果是，这个异常模型似乎在可靠性和易用性两个方面都是最差的。它并没有对写出高可靠的代码提供了什么帮助，同时还非常难用。在代码里你只能写下一大堆啰嗦的语句，而得到的好处却寥寥无几。而且需要对接口进行版本控制简直蛋疼。我们后面会看到，这个模型还有改进的空间。

“版本控制” 这一点值得多说两句。如果你只抛出一种类型的异常，那么它跟错误码方案没什么区别。一个函数要么失败，要么成功。如果你的 API 的第一个版本不会失败，而第二个版本可能会失败，那么这两个版本就是不兼容的。我认为这种思路其实是正确的。API 的失败模型是 API 设计最重要的部份之一，理应与调用者说清楚，这就好像你不会悄悄地、不跟调用者说明白就把一个 API 的返回值类型改了一样。后面会有更多关于这个话题的讨论。

Barbara Liskov 在她 1979 年的论文 Exception Handling in CLU 中描述了 CLU 使用的一种很有意思的方法。他们很关注 “语言学”——换句话说他们想要做出一门大家都喜欢的语言。在 CLU 中，调用方检查以及传播错误的方式跟错误码方案很像，同时他们的编程模型还一点声明式的味道，也就是我们现在所熟知的异常。最重要的是，`signal`（即 `throw`）是受检查的。程序如果抛出了一个没有写明的 `signal`，他们还提供了一种很方便的方式来结束程序。

2.2.2.3 异常的通病 大多数异常系统都有几点很重要的事情做错了，不论是受检查的还是不受检查的异常。

首先，抛出一个异常总是很慢，不可思议的慢。这通常与收集栈信息有关。在一些托管平台上，收集栈信息需要遍历元数据，以便能够创建函数符号名。如果一个错误被抓住之后被合理地处理了，我们就根本不需要这些信息！诊断信息最好在与日志和诊断相关的基础设施里来实现，而不是在异常系统里。这两点问题是同时存在的。不过，要真正能够满足上面提到的诊断需求，还是需要有人能够恢复栈信息的——永远不要小看 printf 调试大法，以及栈信息对它的重要性。

其次，异常会极大地损害代码质量。我在最近的一篇文章中提到过这一点，另外还有很多论文从 C++ 的角度讨论过这个问题。缺少静态类型信息，意味着编译器很难对控制流进行建模，进而导致编译器更加倾向于保守地优化代码。

异常系统的另一个问题在于它鼓励粗粒度地处理错误。很多人喜欢返回错误码的方案，就是因为调用一个函数后就必须进行错误处理。（我也喜欢这一点。）而在异常处理系统中，人们经常用一个的 try/catch 块包住一大坨代码，而不是小心地去应对每一处可能出现的错误。这样的代码很脆弱，而且几乎都是错的；就算现在看起来没问题，将来等代码一重构，问题就会暴露出来。这个问题很大程度上是因为语言没有提供合适的语法导致的。

最后，`throw` 的控制流通常都不可见。就算是在 Java 里，异常是方法签名的一部分，也不可能通过分析函数体就能准确知道异常是从哪抛出来的。不明显的控制流带来的问题就跟 `goto`、`setjump/longjmp` 一样大，增加了编写可靠的代码的难度。

2.3 小结

在继续到下一部分之前，我们先来总结一下吧：

优点 缺点 不是很好的地方

| 错误码 |

所有可能失败的函数都会显式标明

在调用时显式处理所有的错误

| 你可能会忘记检查错误码不发生错误时代码的性能会有所影响 | 易用性不太好 | | 所有的异常系统 | 语言原生支持 |

性能差，其实本可以做的更好一点

错误处理通常都不是局部处理，就像 goto 一样，关于错误的信息比较少

|
不受检查的异常 | 它有助于快速开发（Rapid Development），即错误处理不是最重要的事情 | 任何东西都可能会失败，编译器也不会有警告 | 可靠性很差 |

| 受检查的异常 | 所有可能会失败的函数都会显式标注 |

在函数调用处，函数是否会失败、错误如何传播并不明确

系统允许某些异常不受检查（即不是所有的错误都是显式声明的）

| 遭人恨（至少 Java 是这样）|

如果我们能集成所有优点、再扔掉所有缺点，岂不美哉?

这是我们向前迈出的一大步，不过这还不够。我们第一次觉得思路清晰起来，能把事情都想清楚。对于某一类的错误，这些方法一个合适的都没有!

3 Bug 不是可恢复错误!

关于可恢复错误和 Bug，我们之前提到过二者之间明确的界限：

- 可恢复错误通常都是程序化数据验证的结果。有时候程序会检查这个世界的状态，但认为当前的状态不能接受——比如解析一些标签语言、解析用户在网站上的输入、或是网络不稳定等等。在这些情况下，程序应该恢复执行。写程序的开发人员必须提前安排好要怎么处理这些情况，因为不论我们怎么做，都避免不了这些状况。程序给出的响应可能是给用户返回一些信息、重试或者直接放弃这个操作。虽然它们也被称作 “错误”，但这些错误都是可预测的，而且对于这些状况，程序通常是有准备的。
- Bug 是一类程序员无法预测的错误。用户输入没有被正确地验证，或是逻辑写错了，等等。这类问题一旦发生，就会极大地损害程序状态；然而它们有时候很隐蔽，甚至要等到它们让别处的代码出现了问题后，才会被间接发现。由于程序员并没有想到会发生这些情况，我们就前功尽弃了。那段有问题的代码所能修改的所有的数据结构都可能已经损坏了，同时因

> 为这些问题没有被直接检测到，很大可能所有的东西都已经出了问题。根据你使用的语言所提供的隔离性，也许整个进程都已经被污染了。

这些区别至关重要。令人惊讶的是，很多系统完全不区分它们，至少没有以一种原则性的方式来区分！正如我们所见，Java、C# 以及很多动态语言使用异常、而 C 和 Go 使用错误码，来同时应对这两种错误。C++ 则使用了一种混合的方式，这取决于使用者如何选择，但常见的情况是一个项目只选择一种方法使用。通常来说，没什么语言建议使用两种不同的方法来处理错误。

考虑到 Bug 本身是不可恢复的，我们不需要尝试去恢复执行。在运行时检测到的所有 Bug 需要触发 "*放弃*"，这是 Midori 中的术语，用来指代 "Fail-Fast"。

上面提到的所有系统都提供了类似于放弃的机制。C# 提供了 `Environment.FailFast`；C++ 提供了 `std::terminate`；Go 提供了 `panic`，Rust 也有 `panic!`，等等。它们都能让你迅速脱离当前的上下文。上下文的范围取决于系统——比如 C# 和 C++ 会结束进程，Go 会结束当前的 Goroutine，Rust 会结束当前的线程，还可选附带一个错误处理程序来拯救整个进程。

虽然我们对放弃机制的使用比一般语言更多，但我们显然不是第一个注意到这种方案的。这篇 Haskell 文章就讲的非常好：

> 我曾经参与过编写一个 C++ 库。有一个开发人员对我说，开发者们分成了两派：一派喜欢使用异常，一派喜欢使用返回值。在我看来，返回值派赢得了这场战争的胜利。然而，我却觉得他们所讨论的论点就是错误的：异常和返回值在表达能力上是等价的，它们不应该用来表示错误。很多返回值包含了这样的常量定义：ARRAY_INDEX_OUT_OF_RANGE，但我很不解：如果在调用其他函数时遇到了一个这样的错误，我的函数究竟能做点什么呢？需要发一封邮件给我吗？当然，这个错误码还可以返回给更上层的函数，但是更上层的函数也会不知所措。更糟糕的是，由于我不清楚所有的细节，我只能假设我所调用的所有函数都会返回数组越界的错误。我的结论是：数据越界是一种（编程）错误，在运行时它无法被处理或是修复，只能被开发人员手工修正。因此它不应该使用返回值来表示，而应该使用断言（asserts）。

我有点怀疑只放弃细粒度的可变共享内存（例如 Goroutine、线程之类的东西）的正确性，除非你的系统对潜在损害的范围有所保证。不管怎么说，有这些机制就是件好事！这说明了在这些语言里实践 "放弃" 原则是可行的。

然而，如果想让这种方法在大规模的程序上也可行，我们还需要一点架构设计上的思考。我敢肯定你现在在想："如果在我的 C# 程序里，每遇到一次空指针错误就结束掉整个进程，我肯定会被客户怼的"，或者 "这根本就不叫可靠！"。可靠性，也许并不是你想的那样。

4 可靠性、容错性和隔离性

在继续下一个话题之前，我们需要明确的是：~~猫咪总是四脚朝地~~ 故障（Failure）总是会发生的。

4.1 如何构建一个可靠的系统

大家通常认为，一个可靠的系统就是能够系统性地证明故障不可能发生。直觉上看，似乎很对。但它存在一个问题：在有限的条件下，这是不可能的。如果你能在可靠性这一点上花上上亿

美刀，情况也许会有所改观；此外，也可以使用像 SPARK（基于合约（Contracts）的 Ada 扩展）之类的语言来形式化地证明每行代码的正确性。然而，经验表明这种方法也不是万无一失。

我们选择了接受这种现状。当然，我们必须尽可能地减少故障的发生，错误模型必须能够让故障透明化，并且易于处理。但是更重要的是，你需要在系统设计上下功夫，使得就算某个部分出现了问题，整个系统也还能够工作；同时还要让系统能够优雅地恢复出错的部分。这其实在分布式系统中是常识，有什么特别之处吗?

一切的前提是，我们要把一个操作系统看作是各个协作进程组成的分布式网络，就像由微服务组成的分布式集群、或是互联网本身一样。主要的差别包括延迟、信任级别、关于位置、身份的各种假设，等等。在一个高度异步的、分布式的、IO 密集的系统中，故障是必然发生的。我的印象是，在很大程度上，由于宏内核的成功，我们还没有实现 “操作系统作为一个分布式系统” 的飞跃。然而，一旦你这样看待一个操作系统，很多设计准则都会发生改变。

在大多数分布式系统中，系统架构都会认为一个进程发生故障是不可避免的。从而我们需要花大力气来避免级联故障，定期记录日志，并且将程序和服务设计为允许重新启动的。

一旦你接受了这样的设定，很多事情都变得不一样了。

举个例子，隔离性就变得极端重要。Midori 的进程模型鼓励轻量级、细粒度的隔离。结果就是，程序、甚至是现代操作系统中的 “线程” 之间都是独立分隔的实体。相比于在同一个地址空间中共享可修改的状态，这样的设计可以更容易地在某个实体发生故障时提供保护。

高隔离性同样有利于实现简洁性。Butler Lampson 的经典论文 Hints on Computer System Design 探索了这一方面。我非常喜欢 Hoare 的这句话：

> 可靠性不可避免的代价就是简单化
> C. Hoare
>
> （译注：C. A. R. Hoare (或 Tony Hoare)，发明了快速排序、霍尔逻辑形式验证等算法和系统）

通过把程序分成更小的部分，每一个部分都允许成功或失败，他们的状态管理可以变得更加简单。结果就是，从故障中恢复也变得更加容易。在我们的语言中，可能发生故障的地方都会显式说明，这进一步保证了内部状态机的正确性，同时能够点明哪些地方会与混乱的外部世界发生联系。在这个世界上，局部故障的代价并没有那么可怕。我并没有过分地重视这一点——架构方面提供的隔离性，是我后面描述的所有语言功能的基础。

Erlang 已经非常成功地把这种属性加入到了语言的基础部分。就像 Midori 一样，它使用了轻量级进程, 通过消息传递互相连接, 并且鼓励容错架构设计。一个通用的模式叫做 “Supervisor”，有一些进程会负责监控和重启其他发生故障的进程。这篇文章非常好地阐述了这种理念——“Let it crash”，同时还推荐了一些用来构建可靠的 Erlang 程序的实用技术。

关键之处并非是如何防止故障，而是知道如何应对故障。一旦这样的架构建立了起来，你就会相信它的确能够正常工作。对于我们来说，我们进行了长达一周的压力测试，以确保我们的系统在整体上足够稳定，即便在这期间某些进程可能会因为故障而崩溃、重启。这让我想起了类似于 Netflix 的 Chaos Monkey 系统，它会随机杀掉集群中的某些机器来确保服务的运行状况良好。

随着分布式计算越来越流行，我很期待能够有更多人能够采用这样的思想。比如，在一个微服务的集群中，在单一容器上发生的故障通常能够被外部的集群管理软件无缝地处理（例如 Kubernetes、Amazon EC2 Container Service、Docker Swarm 等等）。所以我这篇博客所描

述的内容可能对写出可靠的 Java/Node.js/Javascript/Python/Ruby 服务有所帮助，但不幸的是你很可能需要跟你使用的语言作一番斗争。为了在出现问题时能够继续勉强工作，你可能需要写上一大堆的代码。

4.2 放弃

就算进程很轻量、隔离得很好、重启很简单，仍然会有人认为遇到 bug 就直接结束进程的做法太过激进。这也是能够理解的。我就来试着说服你吧。

如果你想要构建一个健壮的系统，当遇到 bug 时选择继续执行是很危险的。如果一个程序员没有考虑过某些状况的发生，谁都不知道那些代码会不会继续正常工作。重要的数据也许已经被破坏成不正确的状态了。举一个极端（也许有点傻）的例子：某段程序本来是要对你的银行存款向下取整，现在却开始向上取整了。

也许你认为放弃的粒度应该更小，这就有点复杂了。举例来说，假如你的进程遇到了 bug 并且出现了故障，这个 bug 可能是由于某些静态变量的值出错导致的。尽管其他的线程看起来没问题，你也没办法笃定它们一定不会被影响。除非你的系统支持某些特性——比如语言提供的隔离性、各个独立线程中可访问的顶层对象的隔离性、等等，否则最安全的假设就是只能扔掉整个地址空间，其他的操作都是有风险、不可靠的。由于 Midori 中的进程非常轻量，放弃一个 Modiri 进程就像在其他的系统中放弃一个线程一样。我们的隔离模型能够确保这样做的可靠性。

我必须得承认，“放弃” 范围的界定可能产生滑坡谬误。可能在世界上所有的数据都已经坏掉了，你怎么知道放弃这个进程就够了呢？这里有一个很明显的区别：进程的状态是提前设计好的、非持久化的。在一个设计良好的系统中，进程可以随时被丢弃然后重新创建。的确，一个 bug 可以破坏外部持久化的状态，但如果这种更严重的问题发生了的话，你就需要使用不同的方法来处理了。

我们可以看一看容错系统的设计来了解更多的背景。放弃（快速失败）在容错系统中已经是一个非常常见的技术了，我们也可以把这一领域的大多数经验应用到普通的程序和进程中。也许最重要的一点就是定期记录日志、定期在检查点记录宝贵的持久化状态。1985 年 Jim Gray 的发表了一篇论文 Why Do Computers Stop and What Can Be Done About It? 详细地描述了这种观点。随着程序不断地迁入云服务、并且激进地划分为更小的、互相独立的服务，这种明确区分非持久化和持久化的状态越发显得重要起来。这种潮流也影响了软件的开发方式，如今 “放弃” 在现代架构中也变得更加容易实现。同时，放弃也有助于防止数据的损坏，因为 bug 在下一个检查点到来之前就会被检测到，程序也不会在错误状态下继续执行。

对于 Midori 内核中的 Bug，我们的处理方式有所不同。例如，微内核的 bug 相比于用户态进程的 bug 来说，破坏性更大，最安全的方法是放弃整个 “域（Domain）”（即地址空间）。幸运的是，你所了解的大多数典型的 “内核” 功能——调度器、内存管理、文件管理、网络栈、甚至设备驱动——都在用户态以独立进程的方式执行。这些模块中的错误可以用上文中提到的通常方法解决。

5 Bugs：放弃、断言和合约

在 Midori 中，很多类型的 bug 都可能会触发放弃：

- 不正确的类型转换

- 尝试对 null 指针解引用
- 尝试越界访问数组
- 零除错误
- 算数计算发生意外的上溢/下溢
- 内存不足栈溢出
- 显式放弃
- 合约失败
- 断言失败

我们最根本想法是，这里每一种错误都是不可恢复的。让我们逐一进行讨论。

5.1 普通的老 Bug

上面那些情况中，有一些是公认的程序 Bug。

很显然，不合法的类型转换、尝试对 null 解引用、数组越界访问、零除错误都是程序的逻辑错误，这些都是非法操作。我们在下文会看到，这些操作也有可能被认为是合法操作（比如当一个数除以零时，也许你想要把结果表示为 NaN）；不过我们先默认假设这些就是 Bug。

很多程序员都会接受这个结论。使用 “放弃” 来应对这些 Bug，使得这些 Bug 能够在开发阶段就被快速发现、快速修正。放弃确实有助于提高写代码的生产力。一开始我还有点惊讶，但确实很有道理。

而另一方面，上面其他的情况是否属于 Bug，则是个主观问题。我们必须决定当出现这些状况时程序的默认行为、可能还需要提供程序化的控制方法。我们当时还为此争论不休。

5.1.1 算数计算上溢/下溢

算数计算发生了意外的上溢/下溢究竟算不算一种 Bug，这就是一个有争议的问题。在不安全的系统里，这样的错误经常会导致安全漏洞。我建议你看看国家漏洞数据库，就能知道有多少漏洞是由这种问题导致的。

Windows TrueType 字体的语法分析器就曾经在过去的几年里不断地受到这类问题的影响（我们把它 port 到了 Midori，还有性能提升）。语法分析器似乎是这种安全漏洞的重灾区。SafeInt 因此而生，它本质上就是不让你使用语言本身提供的算术计算操作，转而使用这个库提供的受检查的计算操作。

当然，在这些漏洞中，大多数都跟访问不安全的内存有关。所以你可能会反驳说，溢出在一门安全的语言中没什么害处，应该被允许。然而，基于我们在安全方面的经验，通常当一个程序遇到了意外的上溢/下溢时，继续执行也不会得到正确的结果——简单来说，开发者经常会忽略可能发生的溢出，因此程序执行的结果也就无法预料。这很符合 bug 的定义，而 “放弃” 就是用来发现这些问题的。压死骆驼的最后一根稻草则是一个哲学问题：当我们面对任何有关正确性的问题时，我们更倾向于站在显式意图这一边。

因此，所有无标注的上溢、下溢都会被视为 Bug 并触发放弃。这与 C# 的 `/checked` 编译选项很类似，只有一点不同：我们的编译器会激进地优化掉冗余的检查。（对于 C# 来说，由于很少有人用到这个开关，代码生成器不会那么激进地删掉插入的检查。）多亏了语言和编译器的协同开

发，生成的代码比大多数使用 SafeInt 的 C++ 编译器产生的结果要好。跟 C# 一样，`unchecked` 作用域结构可以用来显式指明允许上溢或下溢。

大多数 C# 和 C++ 程序员第一次听到这个决定时都持反对态度。但尽管如此，我们发现，由算术溢出所触发的放弃，十次有九次都是程序 Bug。剩下的一次是在我们进行了 72 个小时的压力测试之后发生的——我们不断地使用浏览器、媒体播放器等软件来折磨我们的系统——然后，一些没什么危害的计数器溢出了。我觉得相比于通过压力测试来使得程序更加成熟稳定（比如死锁和竞态条件），直接提前发现问题然后修了它要有意思得多了。

5.1.2 内存不足和栈溢出

内存不足（Out-of-Memory，OOM）的情况有点复杂。我们的方法在这当然也是有争议的。

在内存需要手动管理的情况下，错误码风格是最常见的检查方式：

```
X* x = (X*)malloc(...);
if (!x) {
    // 处理内存申请失败
}
```

这种方式有一个很微妙的好处：内存分配很痛苦、需要思考，因此使用这种技术的程序通常更加节俭、谨慎地使用内存。但这种方式有一个巨大的不足：它极易出错、同时会产生大量的没有经过测试的代码路径（code path）。如果一条代码路径没有被测试过，那么它很容易出问题。

当内存快要不够的时候，开发人员所采取的措施通常都不够妥当。以我在 Windows 和.Net Framework 上的经验来看，这就是犯下严重错误的地方，衍生出了极其复杂的编程模型，比如.Net 中所谓的受限执行区域（Constrained Execution Regions）。一个程序一瘸一拐，连一小块内存都拿不到，很快就会成为可靠性的敌人。Chris Brumme 的一篇非常赞的文章讲述了与其有关的故事。

我们的系统中有一些部分比较难搞，比如在内核的底层，放弃的范围肯定要比单一进程更大。不过我们在尽可能地控制这种代码的规模。系统中的其他部分呢？没错，你肯定猜到了——放弃。优雅而又简洁。

令人惊讶的是，这种方法似乎并没有带来太多的麻烦，我把大部分原因归功于隔离模型。实际上，根据资源管理的策略的设置，我们也可以有意地让使一个进程 “内存不足” 进而触发放弃，以此来证明整个系统的健壮性和可恢复性。

当某一次内存申请失败时，你也可以选择不触发放弃。这其实不太常见，但我们也有机制来支持这种选择。可能最合适的例子是：假如你的程序这一次想要申请 1MB 的内存。这跟平时小于 1KB 的对象内存申请不太一样，开发人员可能会提前考虑到内存不够的情况并作出充分的准备。比如：

```
var bb = try new byte[1024*1024] else catch;
if (bb.Failed) {
    // 处理内存申请失败
}
```

栈溢出是这个思路的简单延伸。实际上，由于我们的 “异步链接栈”（asynchronous linked stacks）模型，栈内存不足在物理上就等同于堆内存不足，因此处理方法与 OOM 一致也并不令

人惊讶。如今很多系统都用这样的方式来处理栈溢出。

5.2 断言

断言是指在代码中手动检查某些条件必须为真，否则就会触发放弃。与很多系统一样，我们的系统中断言也分为两种：仅在 Debug 模式下的断言和 Release 模式下的断言。然而不同的是，我们使用 Release 断言的时候更多。实际上，断言在我们的代码里随处可见，大多数的方法都有不止一个断言。

这种实践符合这样的哲学：当运行时遇到 bug 时，把它暴露出来要比继续执行更好。当然，我们的编译器后端知道如何激进地对这些断言进行优化。我们的代码中断言的密度很高，与很多高可靠系统所建议的很类似，比如在 NASA 的论文 The Power of Ten -Rules for Developing Safety Critical Code 中：

> 规则：代码中断言的密度至少为每个函数 2 个。断言用来检查在真正执行时不应该发生的反常现象。断言必须没有副作用，应当定义为布尔值检测。
> 原因：根据工业界编码工作的统计数据，通常每编写 10 行到 100 行代码，单元测试就能发现一个缺陷。发现缺陷的机率随着断言密度的增加而增长。断言也通常作为强防御编码策略（strong defensive coding strategy）所建议的一部分。

如果想要插入断言，只需要简单地调用 `Debug.Assert` 或是 `Release.Assert`：

```
void Foo() {
    Debug.Assert(something); // Debug-only assert.
    Release.Assert(something); // Always-checked assert.
}
```

我们同样实现了与 C++ 中 `__FILE__` 和 `__LINE__` 宏类似的功能，外加 `__EXPR__`，代表了断言中谓语表达式的文本。这样的话由于失败断言而导致的放弃将会包含一些有用的信息。

在早些时候，我们设计了与现在不同的断言“等级”。一共三级，`Contract.Strong.Assert`、`Contract.Assert`、`Contract.Weak.Assert`。最强的等级表示“始终检查”，中间的等级表示“编译器说了算”、最弱的等级表示“仅在 Debug 模式下检查”。我做了一个很有争议的决定——抛弃这种模型。实际上，我非常肯定，我们组里 49.99% 的人都讨厌我对术语的选择（`Debug.Assert` 和 `Release.Assert`），但我始终都觉得这两个词很恰当，它们完美地表明了这两种断言究竟在做什么。旧的分类方法的问题在于，没有人确切地知道什么时候应该检查断言；在我看来，由于好的断言准则对于程序的可靠性极其重要，而断言的使用方法居然会令人困惑，这是我无法接受的。

在我们把合约（contracts）添加到语言中之后（下文很快就会介绍），我们也尝试过把 `assert` 设计为一个关键字。不过，最终我们还是继续使用了 API 的方式。主要的原因是，与合约不同，断言不是 API 签名（signature）的一部分；同时由于断言很容易以库的形式实现，我们也不确定把它加入到语言当中有什么好处；还有一点，“Debug 模式下的检查”和“Release 模式下的检查”之类的策略，看着就不像一门编程语言应该有的东西。我得承认，就算这么多年过去了，我还是不太确定哪种方式更好一些。

5.3 合约

合约是在 Midori 中发现 Bug 的最核心的机制，没有之一。我们是从 Singularity 开始做起的，它使用的是 Sing#，一门 Spec# 的变种语言。不过我们很快就换成使用正规的 C# 了，只能把我们想要的东西重新发明一遍。最终我们实现的模型跟最初的模型已经相去甚远。

多亏了我们的语言对不变性和副作用的理解，所有的合约和断言都被证明了是没有副作用的。这大概是这门语言最大的创新点，我肯定会再写一篇博客讲讲它的。

跟其他方面一样，我们也受到了很多其它系统的影响。Spec# 是最明显的一个。Eiffel 也对我们有很大的影响，尤其是有很多公开发表的研究。另外还有一些其他的研究工作也对我们有所帮助，例如基于 Ada 的 SPARK、一些实时与嵌入式系统的提议等等。更深入地讲，像霍尔公理语义这样的编程逻辑为所有的一切提供了基础。然而对我来说，最重要的哲学启发来自于 CLU、以及之后的 Argus 中提供的错误处理方法。

5.3.1 前置条件和后置条件（Preconditions and Postconditions）

合约最基本的形式就是方法的前置条件，它声明了要调用这个方法所必须满足的条件。通常来说先决条件会用来验证参数。它有时也会用来验证目标对象的状态，不过这非常少见，因为对于程序员来说考虑清楚模态（modality）是一件很困难的事。前置条件基本上就是调用者对被调用者做出的承诺。

在我们最终的模型中，前置条件使用 `requires` 关键字声明：

```
void Register(string name)
    requires !string.IsEmpty(name) {
    // 继续执行，字符串一定不为空
}
```

一种稍微不太常见的合约就是方法的后置条件。它声明了在这个方法调用之后应有的状态。这是被调用者对调用者做出的承诺。

在我们最终的模型中，后置条件使用 `ensures` 关键字声明：

```
void Clear()
    ensures Count == 0 {
    // 继续执行，调用者明确知道在方法返回时 Count 一定 0
}
```

在后置条件中，同样可以通过特殊的名字 `return` 来引用返回值。旧值——比如在后置条件中需要引用输入的时候——可以通过 `old(...)` 来得到。例如：

```
int AddOne(int value)
    ensures return == old(value)+1 {
    ...
}
```

当然，前置条件和后置条件可以一起混用。例如，在 Midori 内核中的环形缓冲区的实现里：

```
public bool PublishPosition()
    requires RemainingSize == 0
```

```
    ensures UnpublishedSize == 0 {
    ...
}
```

这个方法的函数体在执行时可以确信 RemainingSize 一定为 0，调用者在调用之后也可以确信 UnpublishedSize 也一定为 0。

在运行时，任何一个合约没有被满足时，都将会触发放弃。

在这个领域，我们的方法跟其他人的工作有点不同。最近，合约在一些高级证明技术中的程序逻辑中作为一种表达式越来越流行，这些工具工厂使用全局分析来证明声明的的合约是否为真。我们也采用了类似的方法，但合约默认是在运行时检查的。如果编译器在编译期能够证明真假，那么它也可以不生成运行时检查的代码，或是抛出一个编译错误。

现代的编译器都有基于约束的分析，在这方面做得很好，例如我上篇博客中提到的范围分析。它们通过传递事实信息（facts）来对代码进行优化，例如消除冗余的检查：既包括显式声明的合约，也包括普通的程序逻辑。它们也能在合理的时间内完成这些分析，这样程序员才不会觉得太慢转而使用别的编译器。定理证明技术所支持的代码规模满足不了我们的需要。最好的定理证明分析框架花了整整一天来分析我们的核心系统模块！

进一步来说，方法所声明的合约是它签名的一部分。这就意味着这些合约会自动显示在文档里、IDE 的代码提示里，等等。合约就跟方法的参数类型、返回类型一样重要。合约其实就是类型系统的扩展，使用语言所能表达的任何逻辑来对类型交换（exchange types）加以控制。因此，所有常见的对子类型的要求都适用于合约。同样，这也有助于模块化局部分析，使用标准优化的编译器技术能够在几秒钟之内完成。

.NET 和 Java 中大约 90% 的异常都能够使用前置条件替代,包括所有的 ArgumentNullException、ArgumentOutOfRangeException 以及相关的类型。更重要的是，我们不再需要手动检查之后再 throw 了。如今很多 C# 代码中四处可见这样的检查，仅在.NET 的 CoreFX 仓库中就有上千个。例如，下面是 System.IO.TextReader 中的 Read 方法：

```
/// <summary>
/// ...
/// </summary>
/// <exception cref="ArgumentNullException">Thrown if buffer is null.</exception>
/// <exception cref="ArgumentOutOfRangeException">Thrown if index is less than zero.</e
/// <exception cref="ArgumentOutOfRangeException">Thrown if count is less than zero.</e
/// <exception cref="ArgumentException">Thrown if index and count are outside of buffe
public virtual int Read(char[] buffer, int index, int count) {
    if (buffer == null) {
        throw new ArgumentNullException("buffer");
    }
    if (index < 0) {
        throw new ArgumentOutOfRangeException("index");
    }
    if (count < 0) {
        throw new ArgumentOutOfRangeException("count");
    }
```

```
    if (buffer.Length - index < count) {
        throw new ArgumentException();
    }
    ...
}
```

这样的写法不太好，原因有很多。显而易见的是它太冗长了，除此之外似乎没什么不好——然而，我们需要在文档中提及所有可能抛出异常，这似乎在提醒开发人员，这些异常需要被捕获。但这些异常真的不应该被捕获，相反，它们应该在开发阶段就发现这些 bug 然后修掉。

另一方面，如果我们使用 Midori 风格的合约，则这些代码可以简化为：

```
/// <summary>
/// ...
/// </summary>
public virtual int Read(char[] buffer, int index, int count)
    requires buffer != null
    requires index >= 0
    requires count >= 0
    requires buffer.Length - index >= count {
    ...
}
```

这段代码有很多地方值得一说。首先，它更简洁，更重要的是，它以一种文档化的方式描述了这个 API 的合约，调用者也更容易理解。相比于使用用于来描述这些错误条件，调用者还可以直接查看表达式，相关工具也更容易理解和利用这些条件。最后，当合约检查失败时，它会触发放弃。

同样值得一说的是，我们提供了非常多的合约帮助方法，以便能够更方便地实现一些常见的前置条件。上述的显式范围检查同样非常啰嗦，容易出错。我们可以改写为：

```
public virtual int Read(char[] buffer, int index, int count)
    requires buffer != null
    requires Range.IsValid(index, count, buffer.Length) {
    ...
}
```

说句有点偏题的，如果结合两个高级功能——数组切片和非空类型——我们能够进一步简化代码，同时提供了相同的承诺：

```
public virtual int Read(char[] buffer) {
    ...
}
```

还是让我们从头开始讲吧...

5.4 谦卑的开始

虽然我们最终的语法很像 Eiffel 和 Spec#，但就像我前面提到的，我们一开始并没有想改变语言本身。因此我们从 API 开始：

```
public bool PublishPosition() {
    Contract.Requires(RemainingSize == 0);
    Contract.Ensures(UnpublishedSize == 0);
    ...
}
```

这种方法有很多问题，就像 .NET Code Contracts 所遇到的一样。

首先，这种方式实现的合约属于 API *实现* 的一部分，然而我们希望它成为 API *签名* 的一部分。这似乎只是理论上的问题，但其实不是。我们希望生成的程序包含一些内置的元数据，从而像 IDE、调试器之类的工具能够在函数调用时把合约显示出来。我们也希望文档工具能够根据合约自动生成文档。把合约埋在实现当中无法实现这些功能，除非你知道如何反汇编这些函数然后再把这些信息提取出来（然而这是一种 hack）。这也让合约与编译器后端的整合更加困难，生成高性能的代码需要这些信息。

其次，你可能会注意到对 `Contract.Ensures` 的调用。`Ensures` 需要在函数的每一个返回点调用，我们如何只使用 API 来实现呢？答案是：确实不能。一种方法是在编译器生成 MSIL 后，对 MSIL 进行重写，但这很麻烦。看到这你也许回想，为什么我们不能简单地承认这是一个语言的表达能力和予以的问题，然后加上专门的语法呢？

另一方面，一个长期举棋不定的问题是合约是否允许根据编译模式的不同而不同。很多经典的系统中，Debug 模式会检查合约，但完全优化的版本不会。在很长一段时间里，我们有三种不同级别的合约，就像我们之前提到的断言一样：

- Weak，声明为 `Contract.Weak.*`，表示仅在 Debug 模式开启
- Normal，声明为 `Contract.*`，表示由具体实现来决定什么时候检查合约
- Strong，声明为 `Contract.Strong.*`，表示始终检查。

我得承认，一开始我觉得这种方案非常优雅。然而不幸的是，始终都有人对 “Normal” 合约存在疑惑，它们会在 Debug 还是 Release 模式下开启？开始两者都会开启？（也因此大家也都用错了 Weak 和 Strong。）不管怎样，我们开始把这种合约模式整合进语言，以及编译器后端工具链。而此时，问题开始不断出现。我们不得不做出一些妥协。

首先就是，如果你简单的把 `Contract.Weak.Requires` 翻译为 `weak requires`、把 `Contract.Strong.Requires` 翻译为 `strong requires`，你会的到一种相当笨拙而又不够普适的语法，过多可选的策略也让我不舒服。这就直接引出了 `weak/strong` 策略的参数化和可替换性。

其次，这种方法引入了一种新的条件编译模式，对我而言有点不爽。换句话说，如果你想要只在 Debug 模式下检查，已经有语法来实现了：

```
#if DEBUG
    requires X
#endif
```

最后——对我而言这是压死骆驼的最后一根稻草——合约应该是 API 签名的一部分。条件编译一个合约是几个意思？相关工具应该怎么看待 API 签名？需要为 Debug 和 Release 模式产生不同的文档吗？更进一步，如果我们真的采取了这种方式，你就失去了严格的保证——即如果前置条件不满足，你的代码就不会运行。

最后，我们修改了整个条件编译的方案。

我们最终只采用了一种合约：它是 API 签名的一部分，并且始终检查。如果一个编译器在编译阶段就能够证明合约成立（我们花了很大的精力在这个方面），那么可以省略这些检查。我们始终都能够保证前置条件不满足的话就不执行代码。任何需要条件检查的情况，都可以使用上面提到的断言系统。

在部署了这个新的模型之后，我跟高兴曾经误用了 “weak” 和 “strong” 的人都不再困惑了。强迫开发人员做出决定，有助于提高代码的质量。

5.5 未来的方向

当我们的项目结束的时候，很多领域都已经不同程度地有所成熟。

5.5.1 不变量（Invariants）

我们对不变量做了大量的尝试。每次我跟精通合约设计的人讨论问题的时候，他们对我们没有从一开始就使用不变量感到惊讶。老师说，我们的设计的确一开始就包括了这些内容，但我们没能有足够的时间来实现和部署这一功能。部分原因是工程人力的不足，但同时也是由于存在一些难以解决的问题。同时，团队几乎对前置条件、后置条件和断言非常满意。我甚至怀疑，随着时间的推移，我们已经完整实现了不变量，但如今仍有一些问题困扰着我。我得再多看一看它在实践中的应用。

我们的设计所提供的方法是，`invaraint` 是类型内部的一个成员。例如：

```
public class List<T> {
    private T[] array;
    private int count;
    private invariant index >= 0 && index < array.Length;
...
}
```

需要注意的是 `invariant` 标记为了 `private`。一个不变量的访问控制符表明了这个不变量需要检查的成员。例如，一个 `public invaraint` 只需要对 `public` 函数的入口和出口做出限制。这使得一些 `private` 函数能够临时的破坏不变量，只要其在 `public` 的入口和出口处仍然成立即可，这也是一种常见的模式。当然，就像上面的例子一样，一个类也可以声明一个 `private invariant`，这就要求所有的函数在进入和退出时都要满足不变量的条件。

我非常喜欢这种设计，我也认为它能够实现。一个主要的忧虑是我们在所有的地方都引入了一个隐式的检查。就算在今天，这也让我有点不安。例如在 `List<T>` 的例子中，这个类中的每一个函数的开始和结束的地方都需要插入 `index >= 0 && index < array.Length` 的检查。我们的编译器最终能够很好的识别并合并冗余的合约检查，并且在很多情况下合约都能够 *显著* 提高代

码的质量。然而，在上面给出的极端例子中，我确信性能上会有所影响。这迫使我们考虑换一种不变量的检查策略，而这可能会使得整个合约模型变得复杂。

我真的希望我们能够有更多的时间，来更深入地探索不变量。我们的团队似乎并没有对这个功能非常期待——至少我们没有听到过抱怨没有不变量的声音（也许是因为整个团队特别看重性能问题）——但我确定的是，不变量肯定会为合约系统增色不少。

5.5.2 高级类型系统

我喜欢这种说法：合约是类型系统的补充。类型系统让你能够使用类型来对变量的属性进行定义，例如类型可以限制一个变量的值的范围。合约同样可以检查变量的值的范围。那么区别在哪呢？在编译时，严格的、可组合的归纳规则确保了类型的正确性。在检查局部的函数时，这些规则通常不会太耗时，同时还会有开发人员提供的标注作为帮助（不过也可能没有）。而合约则是：尽可能的在编译期证明合约的正确性，如果证明不了的话，就在运行时检查。因此，合约不提供严格的规范，允许使用语言本身提供的任意逻辑进行声明。

类型总是首选，因为它们能够在编译期就提供保证，而且保证能够做到快速地检查。这种为开发人员提供的强有力的保证，有助于提高开发人员整体的生产力。

然而，类型系统不可避免地有一些限制；类型系统需要留出一定的回旋的余地，否则它们就会变得非常难用。极端的情况下，类型系统甚至会退化至位/字节级别的规定。即便如此，有两点“回旋的余地”始终让我很不爽，使得我们必须使用合约来解决：

1. 是否可为空
2. 数值范围

大约 90% 的合约会落入这两点当中。结果就是，我们认真探索了更加复杂的类型系统，以便能够使用类型系统（而不是合约）来判别是否为空、以及变量值的范围。

具体来说，下面这段代码使用了合约：

```
public virtual int Read(char[] buffer, int index, int count)
    requires buffer != null
    requires index >= 0
    requires count >= 0
    requires buffer.Length - index < count {
    ...
}
```

而下面这段代码则不需要，只利用编译期的静态检查就能够得到同样的保证：

```
public virtual int Read(char[] buffer) {
    ...
}
```

将这些属性放入类型系统能够大幅度减轻错误检查的负担。举个例子，对于某个状态来说，它有 1 个生产者、10 个消费者。与其在 10 个消费者那里对状态进行检查，我们可以做到把检查放到生产者那里——比如加上一句强制的类型断言，或者更好的做法是一开始就把值存进正确的类型中。

5.5.3 非空类型

第一点其实有点难办：静态地保证一个变量不会为 `null` 值。这就是 Tony Hoare（译注：就是上文提到的 C. Hoare）所说的著名的 “billion dollar mistake”。对任何语言来说，想办法解决这个问题都是一个非常合理的目标。现在很多新语言的设计师都决定直面这个问题，我非常高兴。

一门语言的很多方面都会阻碍我们实现这一特性，泛型、零值初始化、构造函数，等等。把非空类型添加进一门现有的语言真的很难！

5.5.3.1 类型系统 简单来说，非空类型可以简单归纳为类型系统的一些规则：

1. 所有不加装饰的类型 `T` 默认都是非空的
2. 所有添加了 `?` 的类型，例如 `T?`，都是可空的
3. 对于非空类型来说，`null` 是不合法的值
4. `T` 能够隐式转换为 `T?`。换句话说，`T` 是 `T?` 的一个子类型（尽管不完全对）
5. 存在从 `T?` 到 `T` 的显式转换，运行时需要检查是否为空，如果为空则触发放弃

这其中的大部分似乎都 “显而易见”，我们没有太多的选择。根据类型名称的不同，类型系统能够知道 `null` 的所有路径。具体来说，一个 `null` 永远不能偷偷地变成一个非空类型 `T`；这就意味着我们需要零值初始化，也许这才是所有问题当中最难的一个。

5.5.3.2 语法 我们提供了几种将 `T?` 转换为 `T` 的语法，用来实现第 5 条规则。当然，我们不建议做这种转换，而是更希望你能够尽可能地使用非空的类型。但有时候确实是需要做转换的。多阶段初始化很常见，尤其是容器类的数据结构，因此还是需要支持这种转换。

考虑一下，假如我们有一个 map：

```
Map<int, Customer> customers = ...;
```

这一句构造语句告诉我们：

1. Map 自身不为 `null`
2. Map 当中的 `int` 键不会为 `null`
3. Map 当中的 `Customer` 值也不会为 `null`

假设我们在索引函数中，使用 `null` 来表示某个键不存在的情况：

```
public TValue? this[TKey key] {
    get { ... }
}
```

那么现在我们需要在调用处检查一下索引是否成功。我们讨论了很多种语法。

最简单的方法是加上一个检查来做保护：

```
Customer? customer = customers[id];
if (customer != null) {
    // 此处的 `customer` 变为非空类型 `Customer`
}
```

我承认，我总是对神奇的隐式转换持怀疑态度。当出现问题的时候，很难发现到底是哪里不对。例如，当你试图将 c（译注：应为 customer）跟一个值为 null 的变量进行比较的时候，就会出现问题。不过，这种语法很好记，而且通常情况下都没问题。

当值确实为 null 时，这种语法就会执行另一条分支。但有时你只是像简单地做一次断言，如果值为空则触发放弃。显式类型断言操作符能够实现这样的操作：

```
Customer? maybeCustomer = customers[id];
Customer customer = notnull(maybeCustomer);
```

notnull 操作符将任意的 T? 类型的表达式转换为 T 类型。

5.5.3.3 泛型

泛型有点复杂，因为有多级的可空性需要考虑。考虑下面的代码：

```
class C {
    public T M<T>();
    public T? N<T>();
}

var a = C.M<object>();
var b = C.M<object?>();
var c = C.N<object>();
var d = C.N<object?>();
```

一个很基本的问题是：a、b、c 和 d 的类型是什么？

一开始我们走了点弯路，主要是因为我们当时在模仿 C# 中的 nullable，但其实 C# 的 nullable 有点特别。好消息是，我们最终找到了自己的方法，虽然花了点时间。举个例子来说明我想表达的意思，关于这个问题的答案，现在有两大阵营：

- .NET 阵营：a 是 object；b、c 和 d 是 object?
- 函数式语言阵营：a 是 object；b 和 c 是 object?；d 是 object??

换句话说，.NET 阵营认为你应该把所有的 ? 压缩成一个。而理解了数学组合之美的函数式阵营，则选择不使用任何神奇的操作，让这个世界该怎么样就怎么样。我们最终意识到，.NET 的方法出奇的复杂，同时还需要运行时支持。

函数式语言的方法确实会让人一时摸不着头脑。例如，前面举的 map 的例子：

```
Map<int, Customer?> customers = ...;
Customer?? customer = customers[id];
if (customer != null) {
    // 请注意，`customer` 在此处为 `Customer?`，仍然可能为 `null`！
}
```

在这个模型中，每次只能剥掉一次 ?。但说实话，如果停下来想想，就会发现这其实很有道理。它更加透明，准确地表达了究竟发生了什么事。我们最好不要逆势而为。

此外，在实现上还有问题需要考虑。最简单的实现方法是把 T? 扩展为某些 “wrapper” 类型，比如 Maybe<T>，然后插入合适的 wrap 和 unwrap 操作。确实，这似乎是一个合理的实现方法。

但有这个模型存在两点问题。

第一点问题是，对于引用类型 T 来说，T? 不能浪费不必要的空间——一个指针的运行时表示已经能够包括 null 了。作为一门系统语言，我们希望能把 T? 和 T 实现得一样高效。这可以简单地通过特化泛型的实例化来解决。但这意味着非空类型不再只是简单的前端问题，变为需要编译器后端支持。（需要注意的是，这种解决方案难以扩展至 T??）

第二点是，由于有可变型（Mutability）的标注，Midori 支持安全协变（safe covariant）数组。如果 T 和 T? 有着不同的物理表示，那么从 T[] 到 T?[] 就不是一种转换操作了。这其实只是个小瑕疵，特别是当你堵住了协变数组的安全漏洞之后，协变数组就不再那么有用了。

不管怎样，我们最后放弃了.NET Nullable<T> 这条路，转而使用了组合性更强的多个 ? 的设计。

5.5.3.4 零值初始化

零值初始化是一个真正的难题。它需要：

- 一个类的所有非空域（field）都需要在构造时初始化
- 一个非空类型的数组中的所有成员需要在构造时全部初始化

但这还不是最糟糕的。在.NET 中，所有的值类型（value types）都是隐式零值初始化的。因此最基本的规则就是：

一个 struct 的所有域必须是可空的

绝望的气息。这几乎把整个系统都污染了。我的假设是，只有当可空类型不常用时（比如 20% 的使用率），不可空性才会真正有用。这条规则立刻就毁了一切。

所以我们开始了去除自动零值初始化的工作，工作量非常大。（C# 6 则选择了允许 struct 提供自己的零值初始化构造函数，但由于对整个生态系统的影响太大而不得不放弃。）非空类型本来是能够成功的，但我们有点偏离了跑道、分了心，进而引出了别的问题。如果我能再重来一次，我只需要将 C# 中的值类型和引用类型统一起来就够了。在我下一篇文章——垃圾回收的斗争之路中，这方面的原因将会有更清楚的解释。

5.5.3.5 非空类型的命运

我们有坚实的设计、实现了几个原型，但从来没有在整个操作系统层面部署这个功能。原因与我们所期望的 C# 兼容性有关。公平地说，我在这件事上犹豫了很久，最终就做出了这样的决定。在 Midori 的早期，我们希望人们一见到它就觉得很熟悉；后来，我们甚至在考虑，是不是所有的功能都能够作为 C# 的附加扩展来实现。正是这种想法阻碍了非空类型的实现。如今我相信，仅仅添加一些标注是无法成功的——Spec# 已经这么尝试过了！而且空和非空的默认选择应该倒过来——非空值必须得是默认实现，它才能够达到我们想要的效果。

我最大的遗憾之一就是没能尽早着手非空类型的工作。我们在开始大规模应用合约之后才开始非空类型的探索——因为我们发现代码中充斥着成千上万个 requires x != null。虽然它很复杂、代价很高，但如果同时把去除值类型也实现的话，它们俩绝对是个超赞的组合。真的是要活到老学到老啊！

如果我们把我们的语言从 C# 独立出来单独发布，我相信我们能做的更好。

5.5.4 范围类型

我们设计过向 C# 中添加范围类型，但它的复杂度总是刚好高过我能接受的程度。

基本的想法是，每一个数值类型都可以接受一个下界类型参数和一个上界类型参数。例如，你能够表达出一个整型只能存放 0 到 1000000 之间的整数。例如声明为 `int<0..1000000>`。当然，这其实代表你应该使用一个 `uint`，编译器应该能够给出警告。实际上，所有的整型都可以根据这个概念进行表示：

```
typedef byte number<0..256>;
typedef sbyte number<-128..128>;
typedef short number<-32768..32768>;
typedef ushort number<0..65536>;
typedef int number<-2147483648..2147483648>;
typedef uint number<0..4294967295>;
// ...
```

真正帅爆了（但同时极度复杂）的是，我们可以进而使用依赖类型来允许符号范围参数。例如，我有一个数组，并且希望传入的索引能够保证在某个范围之内。通常来说我会写：

```
T Get(T[] array, int index)
        requires index >= 0 && index < array.Length {
    return array[index];
}
```

或者我可以使用 `uint` 来替代前半部分的检查：

```
T Get(T[] array, uint index)
        index < array.Length {
    return array[index];
}
```

如果有了范围类型，我就可以把索引的上界跟数组的长度直接绑定起来：

```
T Get(T[] array, number<0, array.Length> index) {
    return array[index];
}
```

当然，如果你破坏了编译器的别名检查，边界检查可能就会在运行时进行了。但我们希望至少它能跟合约检查的代价相当。必须要承认，这种方法就是对类型系统中的信息进行更直接的操作。

不管怎样，我还是觉得这个主意太酷了。不过它仍然是“有它更好，没有也不强求”的那一类功能。“不强求”的原因在于，在我们的类型系统中，切片（slice）是一等公民。我打赌，在三分之二甚至更多的情况下，切片是比范围检查更合适的写法。我认为大多数人还是习惯于使用范围检查，因此他们还是写出了标准的 C# 写法，而不是使用切片。我会在之后的文章中介绍切片，在大多数情况下，它确实能够替代范围检查。

6 可恢复错误：类型导向的异常

我们当然不是只需要“放弃”就够了。在很多情况 i 西安，程序员能够把程序从发生的错误中恢复过来。例如：

- 文件 I/O
- 网络 I/O
- 解析数据（例如编译器的语法分析器）
- 验证用户数据（例如用户提交的网页表单）

在这些情况下，遇到错误时你通常不会想要触发放弃。相反，你的程序应该做好了随时遇见这些错误的准备，并且能够合理地解决它们。通常来说是向某些人传递一些信息：例如在网页上输入数据的用户、系统的管理员、开发人员等等。当然，如果合适的话，放弃仍然可以作为一种选择，但通常来说这有点太夸张了。尤其是对于 IO 操作来说，这会让整个系统太过脆弱。想象一下：每当你的网络丢了一个包时，你的程序就决定从此消失！

6.1 使用异常

我们使用异常来应对可恢复错误。不是不受检查的那种，跟 Java 的受检查异常也不太一样。

必须要说明的是：虽然 Midori 使用异常，但一个没有标记为 `throws` 的方法永远不允许抛出异常。Java 中提供的鬼鬼祟祟的 `RuntimeException` 在这是不存在的。我们其实也不需要它，因为在 Java 中使用 `RuntimeException` 的那些情况，在 Midori 中会使用放弃来进行处理。

在最终的系统中，我们神奇的发现，大约有 90% 的函数不会抛出异常！或者说，它们根本不允许抛出异常。这与 C++ 之类的系统形成了鲜明对比——在那些系统中，你必须时刻尽力地避免碰上异常，还需要使用 `noexcept` 来标记。由于存在放弃机制，API 调用仍然可能会失败，但只有调用者没有满足声明的合约时在会发生——类似于传入的参数类型错误。

我们选择使用异常，但这从一开始就有争议。命令式、过程式、面向对象和函数式语言的方法，在我们组里全都有人支持。C 程序员想要用错误码，担心我们会重新发明 Java 的异常设计，甚至更糟糕—— C# 的设计。函数式的观点认为我们应该使用数据流来应对所有错误，异常是一种面向控制流的方法。最后，我认为我们选择的是一种集所有优点于一身的组合模型。一会我们就会看到，我们的确提供了一种机制来将错误看作是一等公民，并且正如开发人员想要的一样，使用数据流风格的编程方法来处理这些偶发情况。

最重要的是，我们使用这种模型写了一堆的代码，对我们来说这个模型工作的很好，甚至得到了经常使用函数式语言的朋友的认可。由于我们也从返回码模型中汲取了一些经验，这让 C 程序员们也很开心。

6.2 语言和类型系统

当时，我做了一个有争议的决定。正如当我们改变了函数的返回类型时，一定会产生兼容性的影响，你也不应该认为改变了函数的异常类型时就没什么兼容性影响。换句话说，*异常和错误码一样，都是返回值的一种！*

这一点在之前介绍受检查的异常时提到过，它是受检查异常的反对者的论据之一。我的回答有点老套，但很简单：反对得不对。你正在使用的是一门静态类型编程语言，而异常的动态特性

才是它们难用的原因。我们想要去解决这些问题，因此我们接受了“异常是一种返回值”的观点，并辅之以强类型。我们从没想过要回头。错误码和异常之间就这样建立起了一座桥梁。

一个函数所抛出的异常是它的签名的一部分，就像参数和返回值一样。请记住，由于异常本来就很少见，这种做法没有你想象的那么痛苦。进而，很多直观的想法就自然而然地出现了。

首先要说的是里斯科夫替换原则（Liskov Substitutin Priciple）。为了避免类似于 C++ 的糟糕局面，所有“检查”都在编译期发生。因此，在 WG21 中所提到的所有性能问题，对我们来说就都不是问题了。类型系统必须坚不可摧，禁止存在任何后门。之所以类型安全与这个问题息息相关，是因为我们的优化器需要依赖于 `throws` 标注才能解决这些性能问题。

我们尝试过使用不同的语法。在我们对语言进行修改之前，我们已经尝试过了各种使用 C# 属性（attribute）和静态分析的方法。用户体验不是非常好，也难以做出一个真正的类型系统。这种方法也存在太多的限制。我们曾经尝试过 Redhawk 项目所采取的方法——Redhawk 最终变为了.NET Native 和 CoreRT——然而，他们的方法同样没有借助语言本身，而是使用了静态分析。不过他们跟我们的最终方案还是有着很多相似之处的。

最终的语法大概是这样，一个方法后面只跟着一个 `throw`：

```
void Foo() throws {
    ...
}
```

（有很长一段时间，我们把 `throws` 放在了函数的最前面。但那种方式读起来有问题。）

这时，可替代性的问题就非常简单了。一个 `throws` 函数不能够替换一个非 `throws` 函数（不合法的加强）。另一方面，非 `throws` 函数能够替换 `throws` 函数（合法减弱）。显然，虚重载、接口实现和 Lambda 会受到这些规则的影响。

我们也实现了预想的协变、反协变替换。例如，如果 `Foo` 是个虚方法，而你的重载实现不会抛出异常，那么你就不需要声明 `throws`。当然，只有直接调用它时，非 `throw` 的性质才会体现，通过虚方法调用它的话则不会。

例如，这是合法的：

```
class Base {
    public virtual void Foo() throws {...}
}

class Derived : Base {
    // 某个特定的实现可能不会抛出异常
    public override void Foo() {...}
}
```

在直接调用 Drived 的方法时，可以从非 `throws` 这一点获益。然而，下面的写法完全不合法：

```
class Base {
    public virtual void Foo () {...}
}

class Derived : Base {
```

```
    public override void Foo() throws {...}
}
```

鼓励使用“单一失败模式”解放了我们——Java 的受检查异常所带来的大量的复杂性瞬间就消失了。如果你看一下 Midori 中可能会失败的 API，大多数都采取了单一失败模式（曾经所有的失败模式都是利用放弃完成的），不论是 IO 失败还是解析失败，等等。例如在处理 IO 操作时，开发人员所准备的恢复操作大多与 *究竟发生了什么错误无关*。（有一些是有关的，而对于那些情况，看守人模式（the keeper pattern）则更加适合。下面很快会讨论到这一点。）其实大多数现代的异常中所携带的信息，从编程角度来说，都没什么用；相反，它们是为了更方便的调试和查错。

大约有两到三年，我们都始终坚持这种“单一错误模式”。最终我又做了一个有争议的决定来支持多失败模式。这种情况不算常见，但经常从团队成员那里听到关于这一点的合理需求，使用的场景看起来很确实是合法的、有用的。这的确会增加类型系统的复杂度，但是跟处理其他的子类型差不多。更复杂的场景——例如终止（Abort）（下面会讨论）——需要我们提供这个功能。

语法看起来像这样：

```
int Foo() throws FooException, BarException {
    ...
}
```

或者说，单一的 `throws` 是 `throws Exception` 的简写。

如果你不关心，把这些显式错误类型“忘掉”也很简单。例如，也许你想把 `Foo` 包进一个 Lambda，但又不想让用户关心 `FooException` 和 `BarException`，那么就可以只写一个 `throws`。这似乎是一个很常见的模式：在系统内部，对于内部的控制流和错误处理来说，使用这种带类型的异常；而在 API 边界处，如果不需要具体的信息，就可以把这些异常都变为简单的 `throws`。

所有的这些标注做法给可恢复错误带来了强大的能力。不过，如果合约与异常之间的比例是 10:1 的话，抛出单一的 `throws` 异常的函数与多异常模式的函数之比就又是 10:1。

到这里，你也许会想，这与 Java 的受检查异常究竟有什么不同？事实上，

1. 大部分的错误都使用了放弃来处理，因此大多数 API 根本不会抛出异常。
2. 我们更建议使用单一错误模式，它让整个系统变得更加简洁。更进一步，多模式和单一模式之间的转换同样非常简单。

强大的类型系统关于弱化和强化的支持在这里同样有所帮助，我们还做了一些别的事情来填补返回码和异常之间的鸿沟，提升了代码的可维护性，等等...

6.3 易于审核的调用现场

目前，我们还没能实现错误码方案中完全显式的语法。函数的声明提到了它们可能会失败 ()，但调用这些函数的地方的控制流仍然是隐式的 ()。

我们的异常模型中，我始终非常喜欢的就是这一点——调用方需要写明 try：

```
int value = try Foo();
```

这将会调用函数 `Foo`，如果发生了错误的话就继续向上传播，否则就把返回值赋给 `value`。

这有一个巨大的好处：程序中所有的控制流都是显式的。你可以把 try 看作是一种条件 return（或者是条件 throw 也行）。在做 Code Review 时，我简直爱死这个语法了！例如，想象一下你有一个很长的函数，里面有一些 try，这些显式的标记能让我们很容易地注意到可能失败的地方，进而能够了解整个控制流。这就跟理解一个 return 语句一样简单：

```
void doSomething() throws {
    blah();
    var x = blah_blah(blah());
    var y = try blah(); // <-- 可能会失败哟!
    blahdiblahdiblahdiblahdi();
    blahblahblahblah(try blahblah()); // <-- 另一处可能会失败的地方!
    and_so_on(...);
}
```

如果你的编辑器提供语法高亮，try 会被加粗、变成蓝色，这就更给力了。这种方案吸收了错误码的很多长处，同时又丢掉了所有的包袱。

（Rust 和 Swift 都提供了类似的语法。我必须得承认，我真的很遗憾，几年前我们没能把这个系统公布与众。它们的实现有所不同，但这绝对是它们语法中非常吸引人的地方。）

如果你在一个函数里使用了 try 来调用一个函数，那么将会发生两种可能的情况：

- 异常继续向上层传播
- try/catch 块处理了这个错误

如果是第一种情况，你也必须把你的函数声明为 throws。不过，是否要把类型信息也向上传播则取决于你。

如果是第二种情况，因为我们能够了解到所有的类型信息，所以如果你试图 catch 没有声明的异常，编译器就会给出错误来避免不可达的代码。这是另外一个与经典的异常系统所不一样的、有争议的地方。我始终对下面这一点耿耿于怀：catch (FooException) 本质上隐藏了动态类型测试。你会允许一个人调用一个类型为 object 的 API，然后自动地把返回值赋值给一个其他类型的变量吗？当然不能！异常这里也是一样。

在异常处理这个方面，CLU 同样对我们有所影响。里斯科夫在 A History of CLU 中讨论了这个话题：

> CLU 对待未处理异常的机制有点不同寻常。大多数机制都把这些异常再次抛出：如果一个调用者没有处理被调用方的异常，那么就把这个异常再抛给上面的调用者。我们不同意使用这种方法，因为它跟我们的模块化程序构建相违背。我们希望知道一个函数的说明就能够去调用它，而不是需要知道它是如何实现的。如果一个异常被自动传播出去，那么一个过程就有可能会抛出它声明中没有指明的异常。

虽然我们不鼓励大范围的 try 块，但在概念上它跟传播错误码是一样的。举个例子，想象一下再使用错误码的系统里，例如 Go，你可能会写：

```
if err := doSomething(); err != nil {
    return err
}
```

在我们的系统里，你可以这样写：

```
try doSomething();
```

但你可能会说，我们用的是异常啊！这完全不一样啊！当然，运行时的系统有所不同。但从语言的“语义”角度来说，它们是一样的。我们鼓励大家从错误码的角度来看待这种方法，而不是它们熟知和喜爱的异常。这看起来有点高效：那我们为什么不直接使用错误码呢？下面我会介绍它们为什么确实是一样的，并会向你解释我们的选择。

6.3.1 语法糖

我们为错误处理提供了一些语法糖。try/catch 块的构建有点啰嗦，同时我们还建议要尽可能细粒度地处理错误，这就让每次都写 try/catch 块显得更加啰嗦。如果你把异常看作是一种错误码的话，它甚至还有点 goto 的味道。这使得我们提供了一种 Result<T> 的类型，简单说它里面要么存放的是 T，要么是一个 Exception。

它在控制流和数据流的世界之间建立起了一座桥梁，对于某些场景来说后者更加自然。两者都有其用武之地，尽管大多数开发人员都对控制流的语法更加习惯。

举个例子，假设你想要在向上抛出异常之前记录下所有的错误。尽管这是一种常见的模式，使用 try/catch 块的控制流看起来太重了：

```
int v;
try {
    v = try Foo();
    // 其他操作...
}
catch (Exception e) {
    Log(e);
    rethrow;
}
// 使用变量 `v`...
```

在“其他操作”那里，你可能会加上一些本不该放在 try 块中的语句。对比一下使用 Result<T> 的方法，这更有返回值的感觉，处理更加局部化：

```
Result<int> value = try Foo() else catch;
if (value.IsFailure) {
    Log(value.Exception);
    throw value.Exception;
}
// 使用 `value.Value`...
```

try ... else 结构还允许使用自己定义的值来替代 catch，也可以触发放弃：

```
int value1 = try Foo() else 42;
int value2 = try Foo() else Release.Fail();
```

通过将 T 的成员映射至 Result<T>，我们还支持了 NaN 形式的数据流错误传播。例如，假如我们有两个 Result<int> 希望进行相加操作：

```
Result<int> x = ...;
Result<int> y = ...;
Result<int> z = x + y;
```

请注意第三行，我们将两个 Result<int> 加了起来，正确地产生了第三个 Result<T>。这就是 NaN 风格的数据流传播，与 C# 中新的 .? 功能类似。

这种方法是对异常、返回错误码和数据流错误传播的一种优雅的混合。

6.3.2 实现

我上面所描述的模型不一定非要用异常来实现。它足够抽象，使用异常或是返回码进行实现都很合理。这不是纸上谈兵——我们确实做过尝试。而我们选择使用异常而不是错误码来实现，是出于性能考虑。

下面来说明一下返回码的实现是如何工作的。我们来做一些简单的转换：

```
int foo() throws {
    if (...p...) {
        throw new Exception();
    }
    return 42;
}
```

将变成：

```
Result<int> foo() {
    if (...p...) {
        return new Result<int>(new Exception());
    }
    return new Result<int>(42);
}
```

以及这样的代码：

```
int x = try foo();
```

将变成：

```
int x;
Result<int> tmp = foo();
if (tmp.Failed) {
    throw tmp.Exception;
}
x = tmp.Value;
```

编译器的优化器可以通过消除不必要的拷贝、或是将函数内联，而使得这些逻辑更加高效。

如果你使用这种方式来建模 try/catch/finally 代码块（大概需要使用 goto），你就会知道为什么编译器难以优化那些使用了不受检查的异常的代码。大量的隐式控制流在作祟！

不管怎样，这个例子非常生动地展示了返回错误码方式的缺点。这些错误处理的代码其实很少有机会被执行（当然，我们需要假设故障发生的频率没那么高），但它们仍然在热路径（Hot Path）上，会影响程序在主要场景下的性能。这违背了我们最重要的准则之一。

我在上一篇博客中描述了我们对两种模式的实验。简单来说，使用异常来实现的话，我们做到了平均减小 7% 的程序体积、提高 4% 的执行速度。原因在于：

- 调用约定没有受到影响。
- 调用函数时不需要做分支检查。
- 在类型系统中，所有会抛出异常的函数都是已知的，因此能够实现更灵活的代码移动。
- 在类型系统中，所有会抛出异常的函数都是已知的，因此能够让我们实现一些更新颖的错误处理优化，例如当 `try` 块中没有异常会抛出时，可以把 `try/finally` 块展平。

还有其他原因使得异常更加高效。我已经提到了，我们不会像大多数异常系统一样遍历调用栈、收集元数据。我们把诊断功能留给了我们的诊断子系统。另一个常见的模式在这同样有所帮助：将异常缓存为不可变的对象，使得 `throw` 时不会发生内存申请：

```
const Exception retryLayout = new Exception();
...
throw retryLayout;
```

对于会频繁抛出和捕获异常的系统来说——例如我们的语法分析器、FRP UI 框架（译注：Functional Reactive Programming，函数式反应式编程范式）等等——这种方法对于提高性能非常重要。这也很好地说明了，我们最好不要把 “异常很慢” 当作真理来对待。

6.4 模式（Pattern）

在我们的语言和类库中，我们添加了很多有用的模式。

6.4.1 并发

回到 2007 年，我写了一篇有关并发和异常的博客文章。这篇文章主要是从并行、共享内存计算的角度来写的，但那些难点是所有的并发调度模式所共有的。我们遇到的基本问题在于：异常是基于单一连续的栈、以及单一失败模式来实现的。在一个并发系统中，你可能有多个栈、多种失败模式，很多情况都有可能 “同时” 发生。

Midori 所做的一个简单的改进是，它会保证所有与 `Exception` 相关的基础设施都能够处理多个内层错误。至少当有多个错误信息同时存在时，程序员不用考虑究竟要留下哪一个。更重要的是，调度和栈爬取设施从根本上掌握着这些 “仙人掌风格” 的栈的信息，也知道如何作出处理。这多亏了我们的异步模型。

一开始，我们不支持异常跨越出异步执行的边界。然而最后我们还是扩展了异常系统，使得异常能够跨越异步边界。这就为异步角色编程模型（Asynchronous Actors Programming Model）引入了一个强大的、有类型支持的编程模型，就像是一种自然的扩展。我们从 CLU 的继任者 Argus 那里学习了一个。

我们的诊断设施在其 “栈视图” 里提供了一种全面的、跨进程因果关系的调试体验。在高度并发的系统中，栈不仅仅是 “仙人掌” 式的，它们还经常出现在进程之间的消息传递边界上。支持这种形式的调试，为开发人员节省了大量的时间。

6.4.2 中止

有时，某个子系统需要“快速脱离苦海”。当遇到 Bug 时，放弃是一种选择。在一个进程中没什么东西能阻止放弃的执行。不过，在确认条件允许的情况下，我们能不能把调用栈恢复到某个时间点、进行恢复、然后在同一个进程中继续执行？

异常比较接近我们的需求。但不幸的是，在调用栈上的代码能够捕获异常，并将其屏蔽掉。我们想要一个不能被屏蔽的东西。

这个“东西”就是中止（Abort）。我们发明中止，主要是因为我们的 UI 框架使用了函数式反应式编程（FRP），这种编程范式在其他一些地方也有应用。当 FRP 重计算发生时，某些事件、或是某些新的发现会令当前的计算失效——例如，当某些计算的调用栈中同时包含了用户代码和系统代码的时候。如果这种情况发生了，FRP 引擎就需要返回栈顶，从而能够安全地开启新一轮计算。由于我们的用户代码是纯函数式的，在其间中止执行非常简单，也不会发生错误的副作用。同时，由于使用了类型化的异常，我们的引擎代码能够被彻底审查并加固，以此来保证满足所有的不变量的条件。

中止的设计借鉴于 capability playbook。首先，我们引入了一个基类，叫做 `AbortException`。它可以直接使用，也可以派生出子类。有一点比较特别：没人能够捕获这个异常然后忽略它。如果有任何 `catch` 试图捕获它，它就会自动地 `catch` 块的结尾再次抛出。我们把这种异常叫做 *不可屏蔽的（undeniable）*。

但总是要有人能够捕获中止的。整个思路就是退出某个上下文，而不是放弃整个进程。我们将会看到如何获得捕获中止的能力。下面是 `AbortException` 的样子：

```
public immutable class AbortException : Exception {
    public AbortException(immutable object token);
    public void Reset(immutable object token);
    // 省略了其他不重要的成员...
}
```

需要注意的是，构造函数接收了一个不可变的 `token`；如果不想再继续向上层抛出，就需要调用 `Reset` 方法，同时必须传入一个相同的 `token`。如果 `token` 不匹配的话，就触发放弃。这里的思路是，抛出和想要捕获中止的模块通常是同一个，或者二者之间至少有协同关系，能够彼此安全地共享 `token`。这是一个典型的对象具有不可伪造特性的例子。

当然，调用栈上的任何一段代码都可以触发放弃，但只要简单地对 `null` 解引用就可以实现了。这种技术禁止在中止上下文中执行，因为上下文可能还没有作出足够的准备。

一些其他的框架也有类似的模式。.NET Framework 中有 `ThreadAbortException`，这同样是一种不可屏蔽的异常，只有调用 `Thread.ResetAbort` 才可以将其捕获。不过，由于这种异常不是基于能力的（译注：参见 Compatibility-based Security），因此它需要安全注释（Security Annotations）和宿主 API 的同时努力，才能够防止中止被意外地屏蔽。老话重提，这种异常也是不受检查的。

由于异常是不可变的，上文提到的 `token` 也是不可变的，一种常见的做法是把这些东西利用单例缓存到静态变量中。例如：

```
class MyComponent {
    const object abortToken = new object();
```

```
    const AbortException abortException = new AbortException(abortToken);

    void Abort() throws AbortException {
        throw abortException;
    }

    void TopOfTheStack() {
        while (true) {
            // 调用一些其他函数
            // 调用栈的某处可能会中止，我们将其捕获并重置：
            let result = try ... else catch<AbortException>;
            if (result.IsFailed) {
                result.Exception.Reset(abortToken);
            }
        }
    }
}
```

这种模式使得中止非常的高效。平均来说一次 FRP 计算会中止多次。要记得，FRP 是系统中所有 UI 的基石，因此不能因为使用了异常就影响性能——申请一个异常对象可能都会不幸地触发 GC。

6.4.3 可选的 “Try” API

上文提到过，很多操作失败时都会触发放弃，例如申请内存、算术计算溢出或是零除，等等。在某些情况下中，有一些失败更适合动态的错误传播和恢复，而不是直接触发放弃。当然，放弃在大多数情况下都是更好的选择。

这也变成了一种模式。不是很常用，但我们也提供。我们提供了一系列的算数 API，它们使用了数据流风格的传播（Propagation）来处理溢出、NaN 或是其他的情况。

我在前面已经提到了这项功能的具体例子——我们使用 `try new` 来申请内存，这时内存不足将产生一个可恢复错误，而不是触发放弃。这极其少见，但有时还是很有用的，例如当你想申请一大块缓存来做一些多媒体操作的时候。

6.4.4 看守人

这里介绍的最后一种模式，叫做 *看守人模式（the keeper pattern）*。

在很多方面，可恢复异常都是 “由内而外” 地进行处理。一堆代码执行了，向下传递了一些参数，知道某段代码发现自己的状态有点问题。然后，在异常模型中，控制流重新沿着调用栈向上传播，进行栈展开，直到某段代码能够处理这个错误。如果此时决定需要重试的话，就再次执行一系列的操作。

一种可选的替代模式是使用看守人。看守人是一个对象，它知道如何 “就地” 从错误当中恢复，因此调用栈就不需要展开了。相反，之前需要抛出异常的代码，现在可以咨询看守人，而它就可以对如何继续执行做出指示。看守人的好处之一是，如果它作为一项配置能力时（configured

capability），周围的代码甚至不需要知道它们的存在。这就与异常不同，在我们的系统中，异常是类型系统的一部分。看守人的另一个好处是，它们非常简单，代价也不高。

Midori 中的看守人可以用来做一些提示的操作，不过更常见的是用来跨越异步的边界。

看守人典型的使用场景是对文件系统操作进行保护。访问文件系统中的文件和目录通常会由于这些原因失败：

- 不合法的路径
- 文件没找到
- 目录没找到
- 文件正在使用
- 权限不足
- 设备已满
- 设备写保护

一种方案是，对每一个文件系统 API 都标注为 `throws`。或者像 Java 一样，创建一套 `IOException` 的体系，把它们作为子类。而另一种方案就是使用看守人。这保证了应用程序整体上不需要知道、或是不需要关心 IO 错误，能够让错误恢复逻辑集中在一起。这个看守人的接口可能会是这样：

```
async interface IFileSystemKeeper {
    async string InvalidPathSpecification(string path) throws;
    async string FileNotFound(string path) throws;
    async string DirectoryNotFound(string path) throws;
    async string FileInUse(string path) throws;
    async Credentials InsufficientPrivileges(Credentials creds, string path) throws;
    async string MediaFull(string path) throws;
    async string MediaWriteProtected(string path) throws;
}
```

这里的想法是，对于每种情况，当错误发生时，相关的输入都会提供给看守人。看守人随后会执行一些操作——也许是异步的——来进行恢复。很多时候，看守人可以选择返回一个新的参数来执行这项操作。例如，`InsufficientPrivileges` 可以返回一个替代的 `Credentials` 来使用。（程序可以向用户弹出一个提示框，让用户切换到有写权限的账户。）对于上面列举的每一种情况，如果看守人不想处理的话，也可以抛出异常，不过这个功能不是看守人模式必须提供的。

最后，我应该说明的是，Windows 的结构化异常处理（Structured Exception Handling, SEH）支持“可继续的”异常，它们从概念上讲是一样的。SEH 允许让某段代码来决定如何重新执行失败的计算。不过遗憾的是，SEH 使用了调用栈上的环境处理函数，而不是语言中的一等公民对象，这让它看起来不那么美观，而且更容易出错。

6.5 后续的方向：Effect Typing

很多人都问过我们，在类型系统中加上 `async` 和 `throws` 会不会让库代码变得更加臃肿。答案是“不会”。不过在高度多态的库代码中，这确实是件令人头痛的事。

最让人不安的是那些组合子，例如 map，filter，sort，等等。在这些情况下，你通常会希望

那些带有 async 和 throws 的函数能够透明地“流过”。

我们用来解决这个问题的方案是，允许你使用效果（Effect）来参数化类型。例如，这是一个通用的映射函数 Map，它把它的 func 参数上的 async 或是 throws 效果进行传播：

```
U[] Map<T, U, effect E>(T[] ts, Func<T, U, E> func) E {
    U[] us = new U[ts.Length];
    for (int i = 0; i < ts.Length; i++) {
        us[i] = effect(E) func(ts[i]);
    }
    return us;
}
```

需要注意的是，我们使用了一个常规的泛型类型 E，只是在它前面多了一个关键字 effect。而后我们就可以用 E 来符号化地用在 Map 的效果列表里，同时还在“传播”时使用了它——通过 effect(E) 来传播调用 func 时产生的效果。我们可以简单地把 E 替换为 throws、effect(E) 替换为 try，来看一看背后的逻辑转换。

对 Map 的合法调用如下：

```
int[] xs = ...;
string[] ys = try Map<int, string, throws>(xs, x => ...);
```

由于 throws 会“流过”内部操作，我们因此可以传入一个会抛出异常的回调函数。

总而言之，我们进一步讨论了这个想法，并允许程序员声明任意多的效果。我之前就设想过这样的类型系统。不过我们还是有些顾虑，无论这种高阶编程多么强大，它们也许都只是小聪明，而且还难以理解。上面这个简单的模型也许看起来不错，如果在多给我几个月的时间，我们也许能够把它实现出来。

7 回顾和总结

这趟旅程终于快要结束了。就像我一开始提到的一样，这趟旅程也许没什么惊喜，平淡无奇。但我希望所有的这些背景历程能够帮助你在错误模型方面取得更多的进展。

总结一下，我们最终的模型：

- 具备一个良好的架构，能够细粒度地提供隔离，并在失败时提供恢复能力。
- 区分 Bug 和可恢复错误。
- 对于所有的 Bug，使用合约、断言，以及放弃来处理。
- 对于可恢复错误，使用精简的受检查异常模型来处理，它还有强大的类型系统和语言语法的支持。
- 汲取了返回错误码的部分优点（比如局部检查）来提高可靠性。

虽然这是一个持续了多年的旅程，但一直到我们的项目被砍掉时，我们还是在不断地做出改进。有些改进没有写入上面的列表中，因为我们还没有足够的经验来表明它们一定能成功。多希望我们能够走的更远、把那些功能都真正发布出来啊！特别是，我真的很想把下面这一点放入我们的最终总结当中：

- 默认使用非空类型，从而消除大量的不可空标注。

放弃，以及我们大量使用放弃的程度，在我看来是我们在错误模型方面最成功的赌注。我们经常能够及早发现 Bug，在它们还很容易调试的时候修掉。基于放弃的错误和可恢复错误大约比例为 10:1，这使得受检查的异常很少见，开发人员也能够接受使用受检查的异常。

尽管我们没能成功的发布这套系统，还是有一些经验可以带到其他的项目中去的。

例如，在把 Internet Explorer 重写为 Microsoft Edge 时，我们在一些地方采用了放弃。最关键的一处就是内存不足，这是由一个 Midori 工程师实现的。就像我上文中提到的那样，老的代码在内存不足时还会尝试继续一瘸一拐地走下去，而此时它通常都是在做错误的事情。我的理解是，放弃能够发现相当多的潜在 Bug，我们在移植现有代码到 Midori 上时经常会遇到。另一个很棒的地方在于，放弃更像是一种架构层面的准则，可以在不同语言的现有代码中使用。

细粒度隔离的体系基础非常关键，然而许多系统对这种架构都只有非正式的概念。在浏览器中，内存不足时采用放弃工作的很好，原因之一就是大多数浏览器都已经把单独的标签页放在独立的进程中了。浏览器在很多方面都在模拟一个操作系统，这也是其中之一。

最近，我们还在探索把这些准则——包括合约——带入 C++。同样，对于 C#，我们也有某些功能带入其中的具体提议。我们还在非常积极地改进一份把非空检查带入 C# 的提议。我希望这些提议一切顺利，但是需要承认的是，这些语言并不是从一开始就遵循同样的错误处理准则，因此这些特性未必能够做到完美无缺。同样需要注意的是，整个隔离和并发模型对于放弃来说同样至关重要。

我希望，我持续地分享这些知识，能够让更多人了解、采用这些想法。

当然，正如我提到的那样，Go、Rust 和 Swift 为我们提供了非常棒的、适合系统开发的错误模型。我也许还是有一点小挑剔，但事实是它们要比 Midori 刚启动时工业界的所有模型都要好。这真是一个系统程序员的好时代！

下次我会谈一谈我们使用的语言。尤其是，我们会看到，Midori 是如何使用包括架构、语言支持和库在内的神奇药水来驯服垃圾收集器的。期待与你下次再见！