SwiftCon China 2016

www.swiftconchina.com

SwiftCon × Think

理解 MONAD

唐巧@小猿搜题
http://weibo.com/tangqiaoboy

基础知识一：数组的MAP

```
let arr = [1, 2, 4]
// arr = [1, 2, 4]

let brr = arr.map {
    "No." + String($0)
}
// brr = ["No.1", "No.2", "No.4"]
```

基础知识二：数组的FLATMAP

```
let arr = [[1, 2, 3], [6, 5, 4]]
let brr = arr.flatMap {
    $0
}
// brr = [1, 2, 3, 6, 5, 4]
```

基础知识二：数组的FLATMAP

```
let arr: [Int?] = [1, 2, nil, 4, nil, 5]
let brr = arr.flatMap { $0 }
// brr = [1, 2, 4, 5]
```

基础知识三：OPTIONAL的MAP

```
let a1: Int? = 3
let b1 = a1.map{ $0 * 2 }
// b1 = 6

let a2: Int? = nil
let b2 = a2.map{ $0 * 2 }
// b2 = nil
```

基础知识四：OPTIONAL的FLATMAP

```
let s: String? = "abc"
let v = s.flatMap { (a: String) -> Int? in
    return Int(a)
}
```

基础知识五：类型转换

```swift
let s2: String? = nil
let s1: String? = "abc"
```

基础知识五：类型转换

```
public enum Optional<Wrapped> :
    _Reflectable, NilLiteralConvertible {
    case None
    case Some(Wrapped)

    @available(*, unavailable, renamed="Wrapped")
    public typealias T = Wrapped

    /// Construct a `nil` instance.
    @_transparent
    public init() { self = .None }

    /// Construct a non-`nil` instance that stores `some`.
    @_transparent
    public init(_ some: Wrapped) { self = .Some(some) }

}
```

MONAD是什么?

MONAD是什么？

链式调用的编程范式

链式调用的编程范式

数组的链式调用

```
let arr = [1, 3, 2]

let brr = arr.map {
    $0 * 2
} .map {
    "this is " + String($0)
} .map {
    $0.uppercaseString
}
```

Optional 的链式调用

```swift
let tq: Int? = 1
let b = tq.map {
    $0 * 2
}.map {
    "abc" + String($0)
}
```

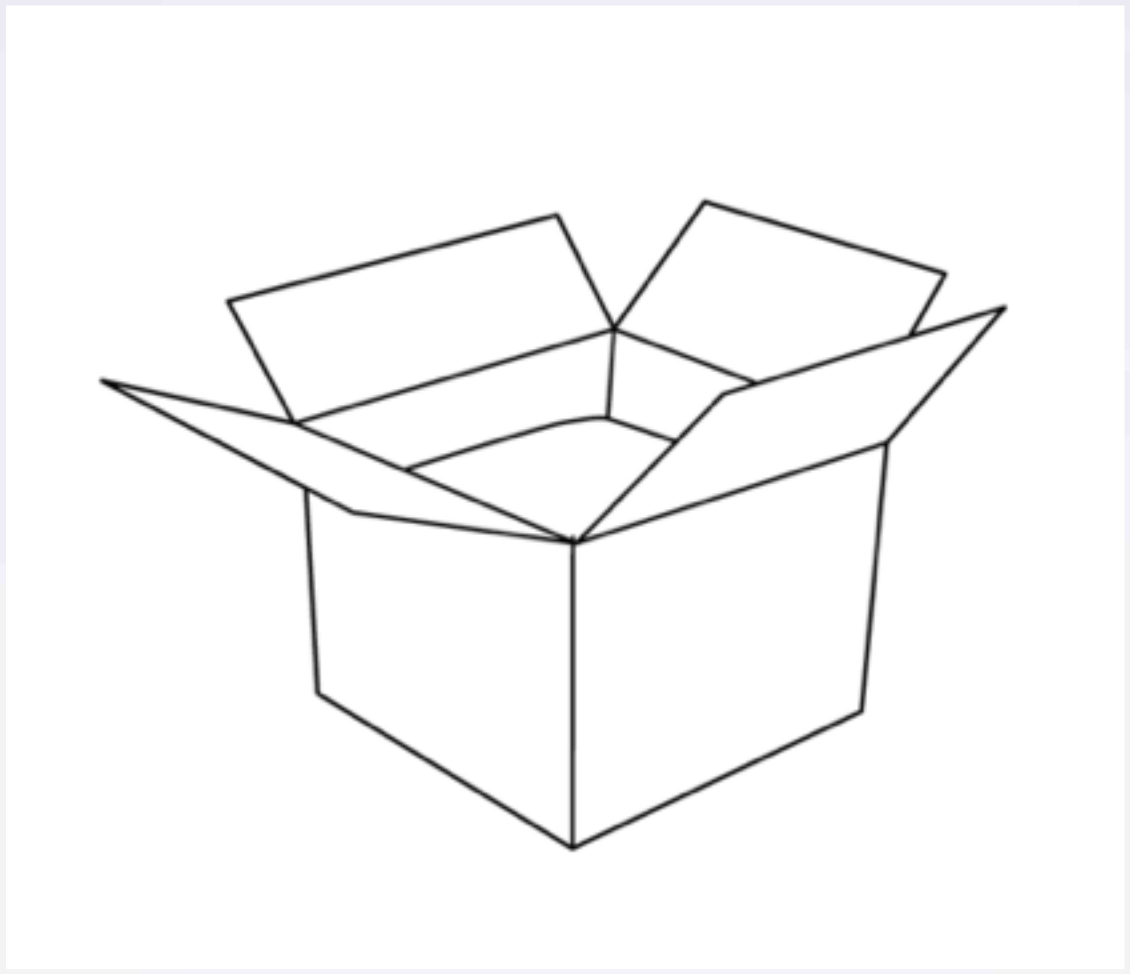
链式调用有什么好处?

```
    TTRequest *req1 = [TTRequest requestWithUrlString:@"url1"];
    [req1 startWithCompletionBlockWithSuccess:^(__kindof YTKBaseRequest *request) {
        TTRequest *req2 = [TTRequest requestWithUrlString:[NSString stringWithFormat:@"%@", req1.result]];
        [req2 startWithCompletionBlockWithSuccess:^(__kindof YTKBaseRequest *request) {
            TTRequest *req3 = [TTRequest requestWithUrlString:[NSString stringWithFormat:@"%@",
req2.result]];
            [req3 startWithCompletionBlockWithSuccess:^(__kindof YTKBaseRequest *request) {
                ;
            } failure:^(__kindof YTKBaseRequest *request) {
                [TTAlertUtils showAutoHideHint:@"网络错误" inView:self.view];
            }];
        } failure:^(__kindof YTKBaseRequest *request) {
            [TTAlertUtils showAutoHideHint:@"网络错误" inView:self.view];
        }];
    } failure:^(__kindof YTKBaseRequest *request) {
        [TTAlertUtils showAutoHideHint:@"网络错误" inView:self.view];
    }];
```

链式调用的编程范式

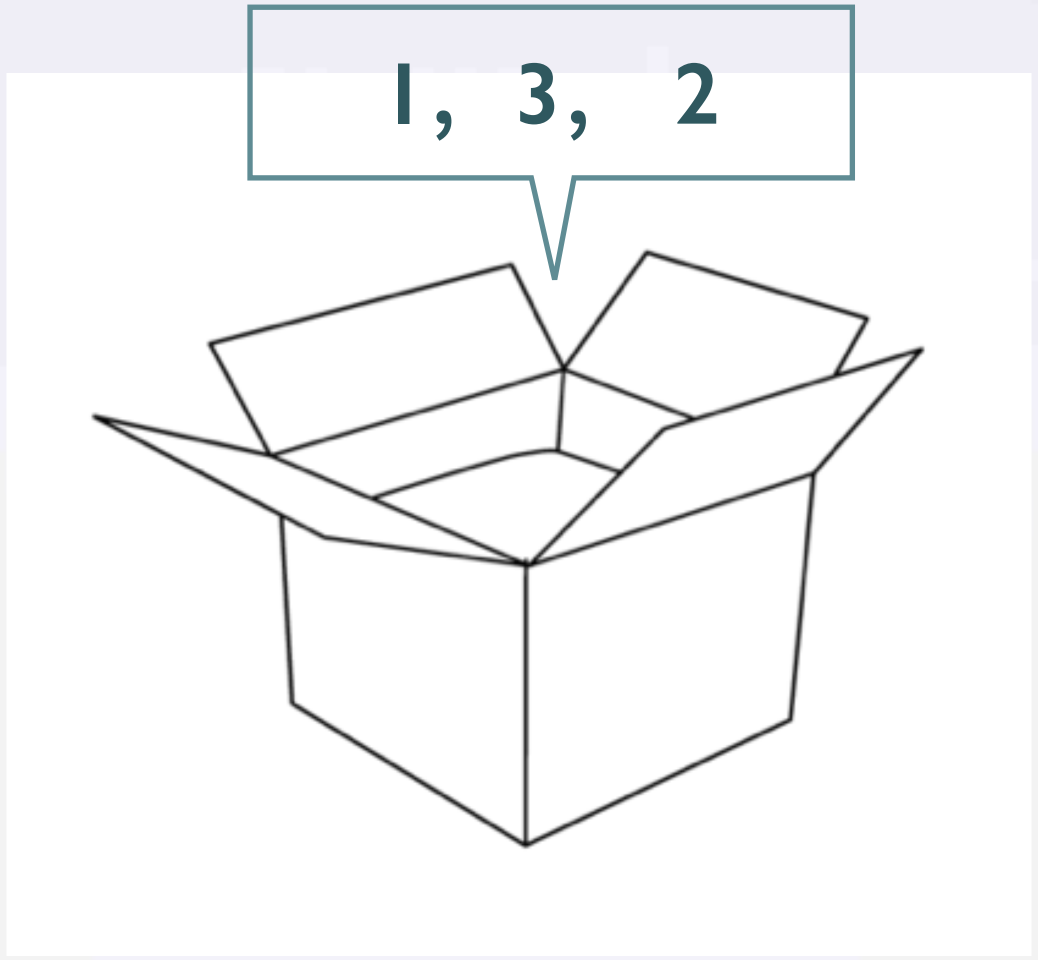
一种更 General 的设计模式

盒子：封装的数据



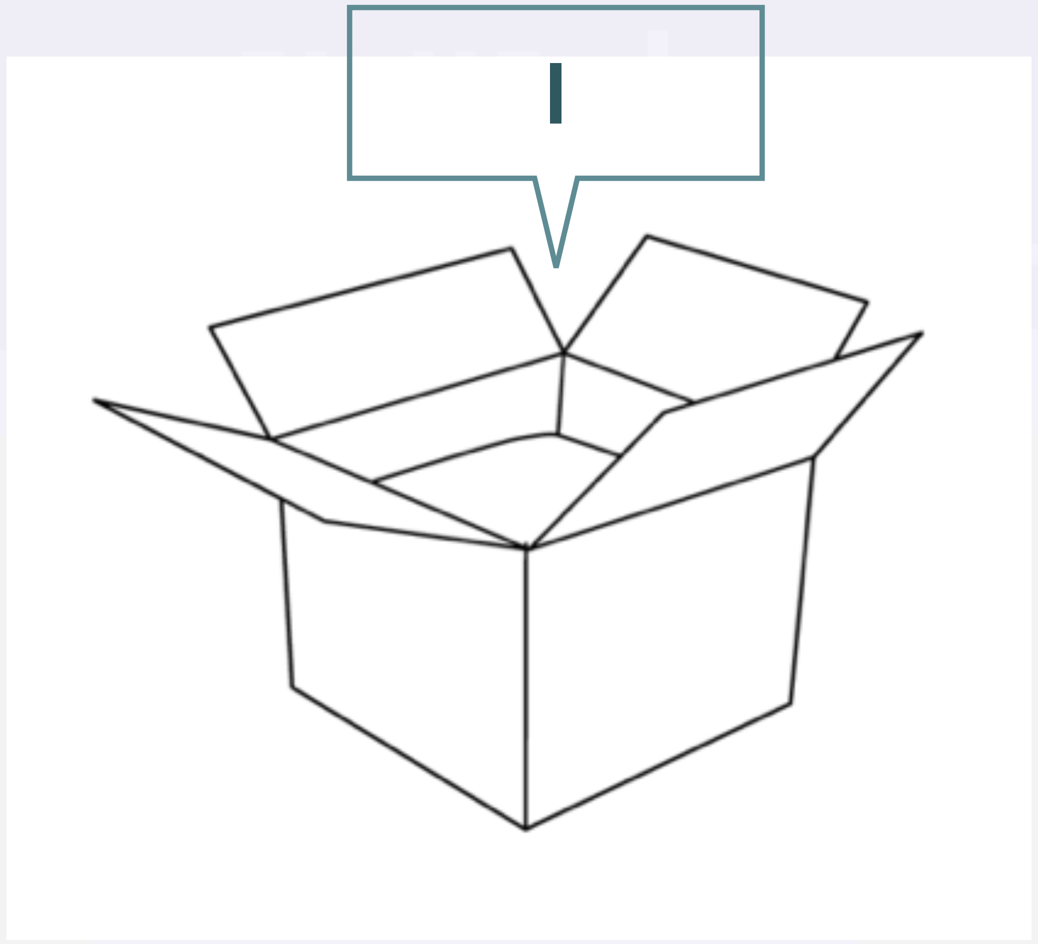
数组形式的盒子

```
let arr = [1, 3, 2]
```



OPTIONAL形式的盒子

```
let tq: Int? = 1
```

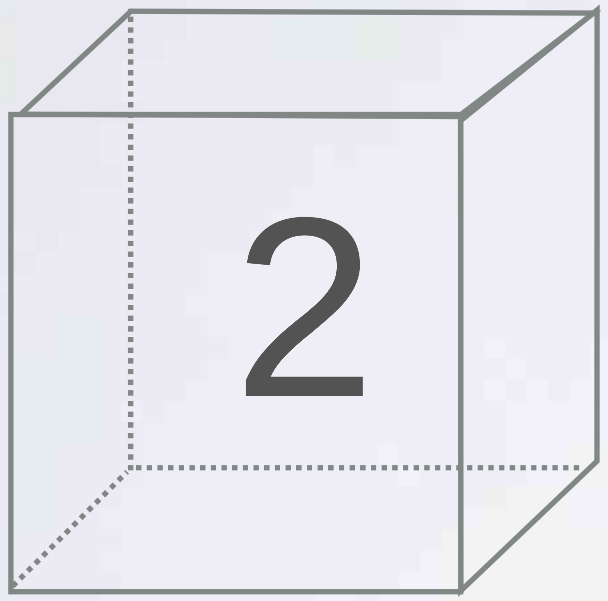


```
let arr = [1, 3, 2]
let tq: Int? = 1
```

```
enum Result<T> {
    case Success(T)
    case Failure(ErrorType)
}
```

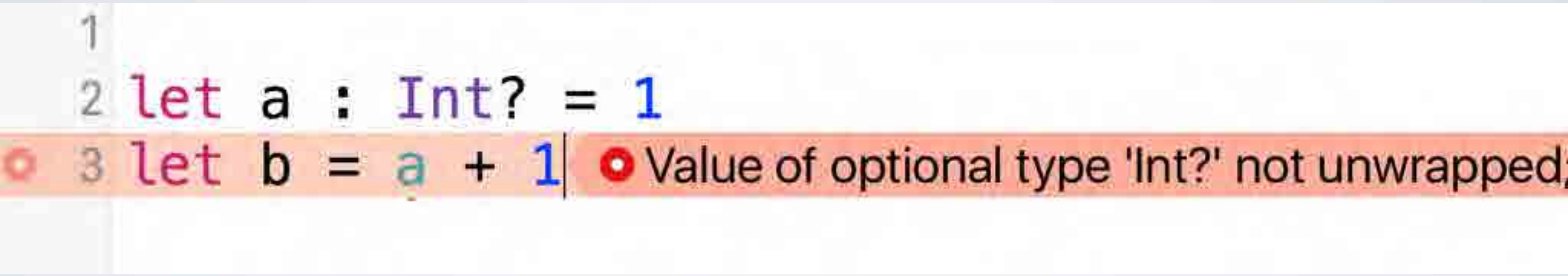
所有可以被“打开”的数据

困境：封装的数据不能直接计算



+ 1 = ?

困境：封装的数据不能直接计算

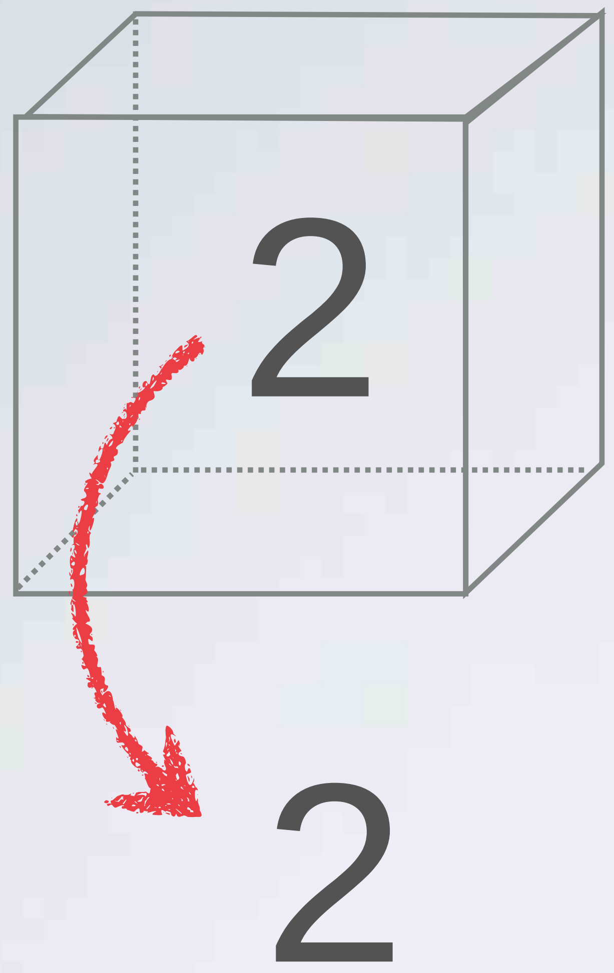


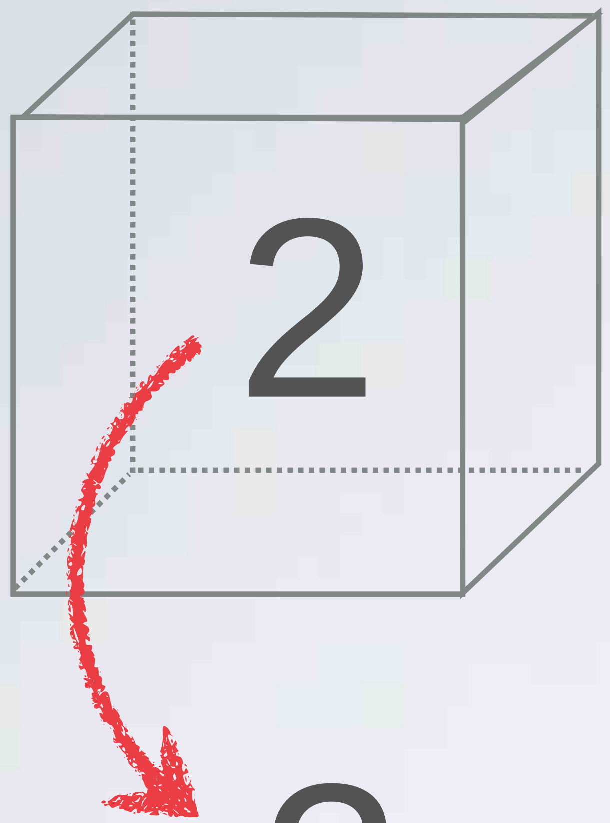
```

let a : Int? = 1
let b = a + 1  Value of optional type 'Int?' not unwrapped
```

困境：封装的数据不能直接计算

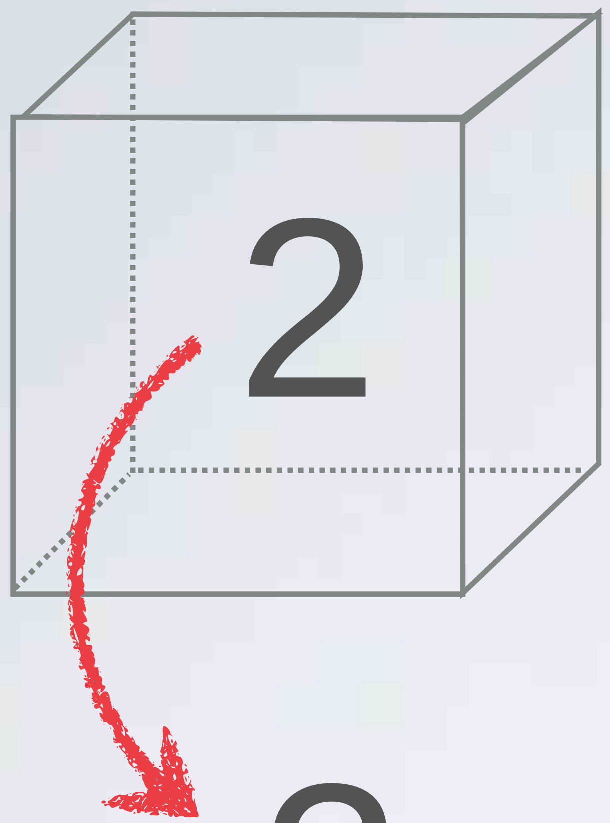
```
let a : Int? = 1
var b: Int?
if let a = a {
    b = a + 1
} else {
    b = nil
}
```

2

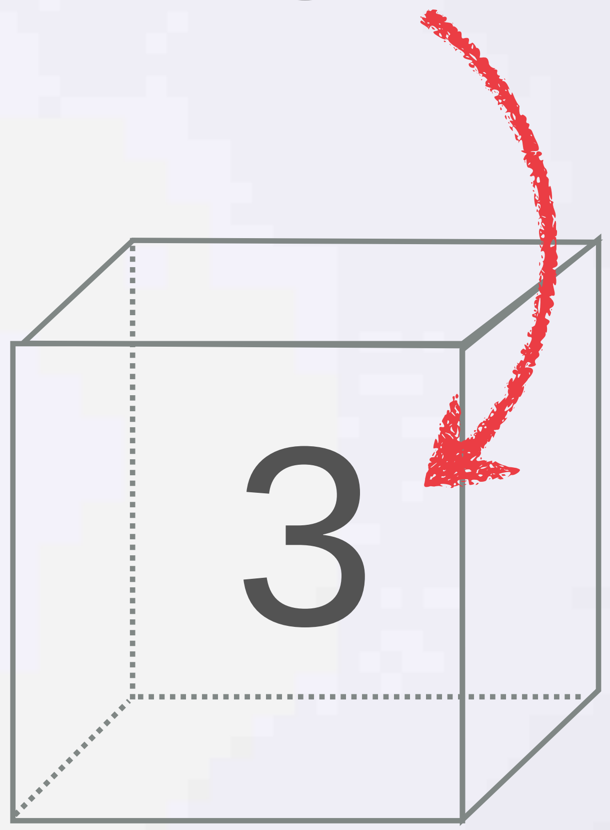

2
2

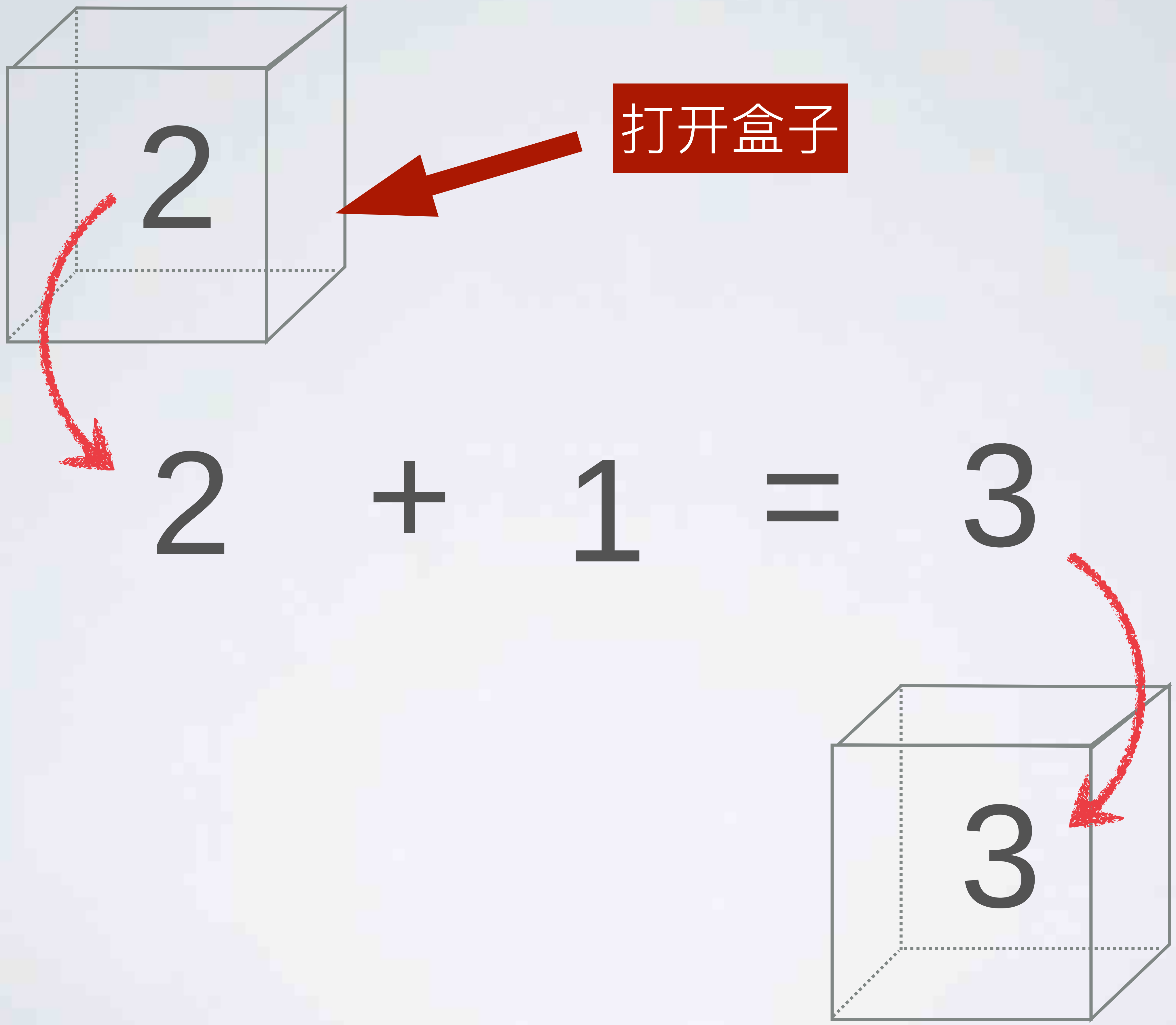


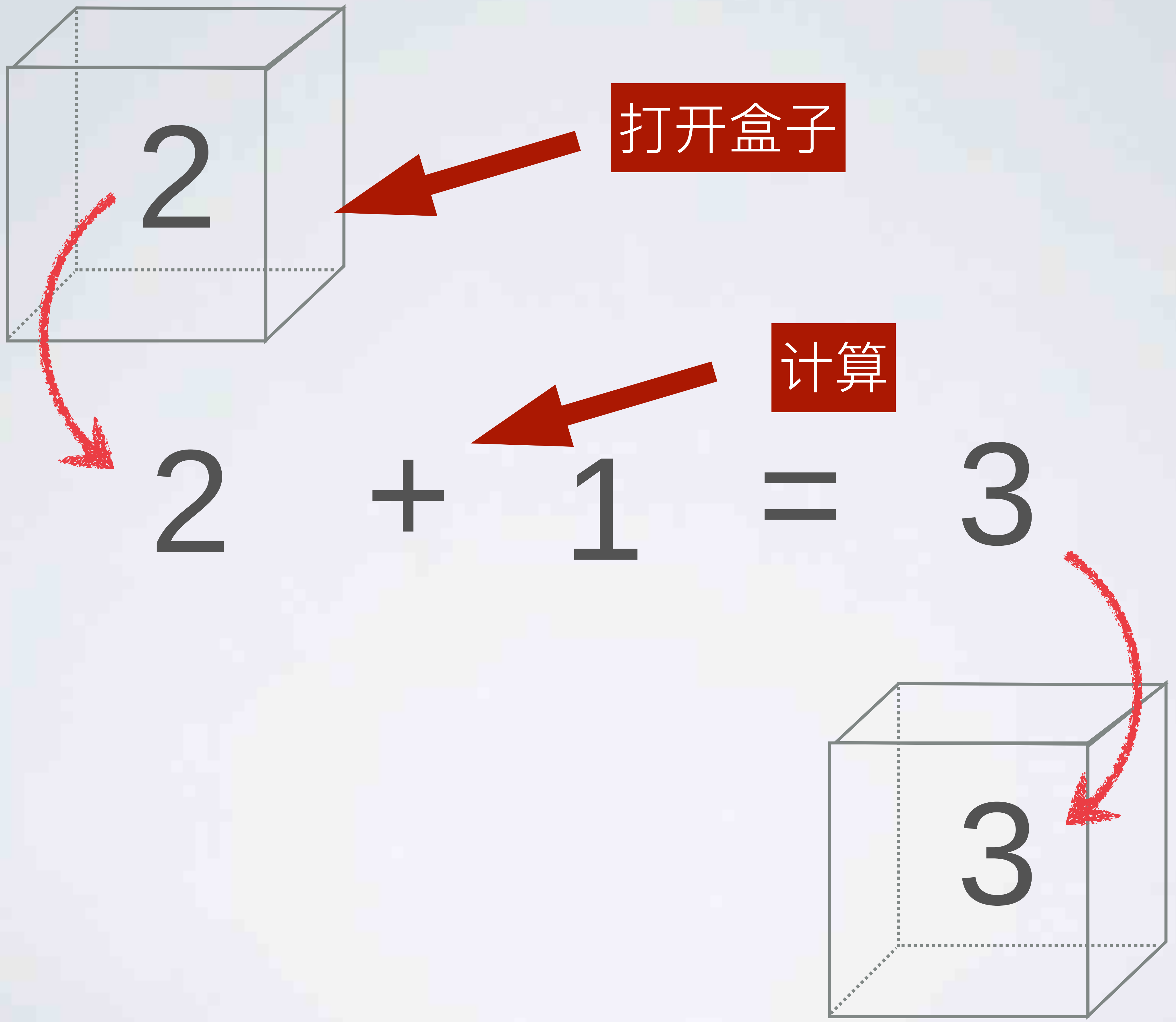
$$2 + 1 = 3$$

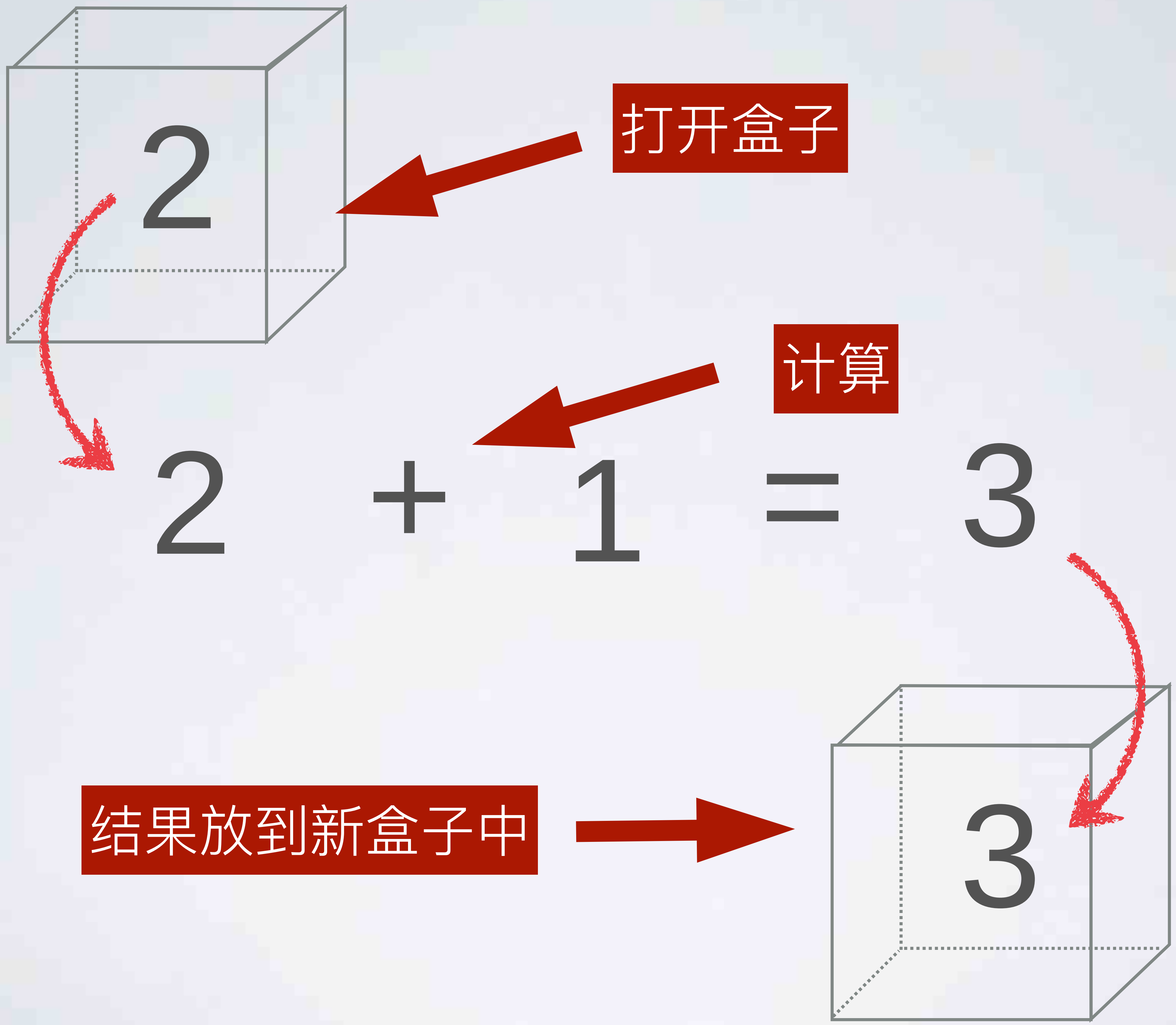


$$2 + 1 = 3$$




2
打开盒子
2 + 1 = 3
3


打开盒子
2
计算
2 + 1 = 3
3


打开盒子
2
计算
2 + 1 = 3
结果放到新盒子中
3

- 计算前需要打开盒子
- 计算之后再封装盒子

- 计算之前的打开能不能是自动的?
- 计算之后的封装能不能是自动的?

这就是MAP

```
let arr = [1, 3, 2]

let brr = arr.map {
    (element: Int) -> Int in
    return element * 2
}
```

这就是MAP

```
let arr = [1, 3, 2]

let brr = arr.map {
    (element: Int) -> Int in
    return element * 2
}
```

这就是MAP

```
let arr = [1, 3, 2]

let brr = arr.map {
    (element: Int) -> Int in
    return element * 2
}
```

自动将数组中的数据取出来，
算完之后再放到新数组中去

OPTIONAL 的 MAP

```
let a1: Int? = 3
let b1 = a1.map{ (e: Int) -> Int in
    return e * 2
}
```

```
let a1: Int? = 3
let b1 = a1.map{ (e: Int) -> Int in
    return e * 2
}
```

回顾

- 什么是盒子?
- 什么是 map ?

Talk is cheap. Show me the code.

— Linus Torvalds

数组的MAP源码

```swift
public func map<T>(@noescape transform:
    (Generator.Element) throws -> T)
    rethrows -> [T] {
        let count: Int = numericCast(self.count)
        if count == 0 {
            return []
        }

        var result = ContiguousArray<T>()
        result.reserveCapacity(count)

        var i = self.startIndex

        for _ in 0..<count {
            result.append(try transform(self[i]))
            i = i.successor()
        }

        _expectEnd(i, self)
        return Array(result)
}
```

```swift
public func map<T>(@noescape transform:
     (Generator.Element) throws -> T)
     rethrows -> [T] {
          let count: Int = numericCast(self.count)
          if count == 0 {
               return []
          }

          var result = ContiguousArray<T>()
          result.reserveCapacity(count)

          var i = self.startIndex

          for _ in 0..<count {
               result.append(try transform(self[i]))
               i = i.successor()
          }

          _expectEnd(i, self)
          return Array(result)
}
```

打开盒子

```
public func map<T>(@noescape transform:
    (Generator.Element) throws -> T)
    rethrows -> [T] {
        let count: Int = numericCast(self.count)
        if count == 0 {
            return []
        }

        var result = ContiguousArray<T>()
        result.reserveCapacity(count)

        var i = self.startIndex

        for _ in 0..<count {
            result.append(try transform(self[i]))
            i = i.successor()
        }

        _expectEnd(i, self)
        return Array(result)
}
```

打开盒子

结果放到新盒子中

OPTIONAL的MAP源码

```
public func map<U>(@noescape f: (Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

打开盒子

```
public func map<U>(@noescape f: (Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

打开盒子

```
public func map<U>(@noescape f: (Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

结果放到盒子中

为什么MAP不能解决所有问题?

为什么MAP不能解决所有问题?

计算之后的封装不一定能自动。

自动封装的问题

```swift
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

自动封装的问题

自动转换

```
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

对比源码

打开盒子

```
public func map<U>(@noescape f: (Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

结果放到盒子中

```swift
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

```swift
public func map<U>(@noescape f:
(Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

```
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

self 为 Some(1)

```
public func map<U>(@noescape f:
(Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

```
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

```
public func map<U>(@noescape f:
(Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

self 为 Some(1)

self 有值, y 为 1

```
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

self 为 Some(1)

self 有值，y 为 1

调用闭包f，得到：
Optional<Int>.None

```
public func map<U>(@noescape f:
(Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

```
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

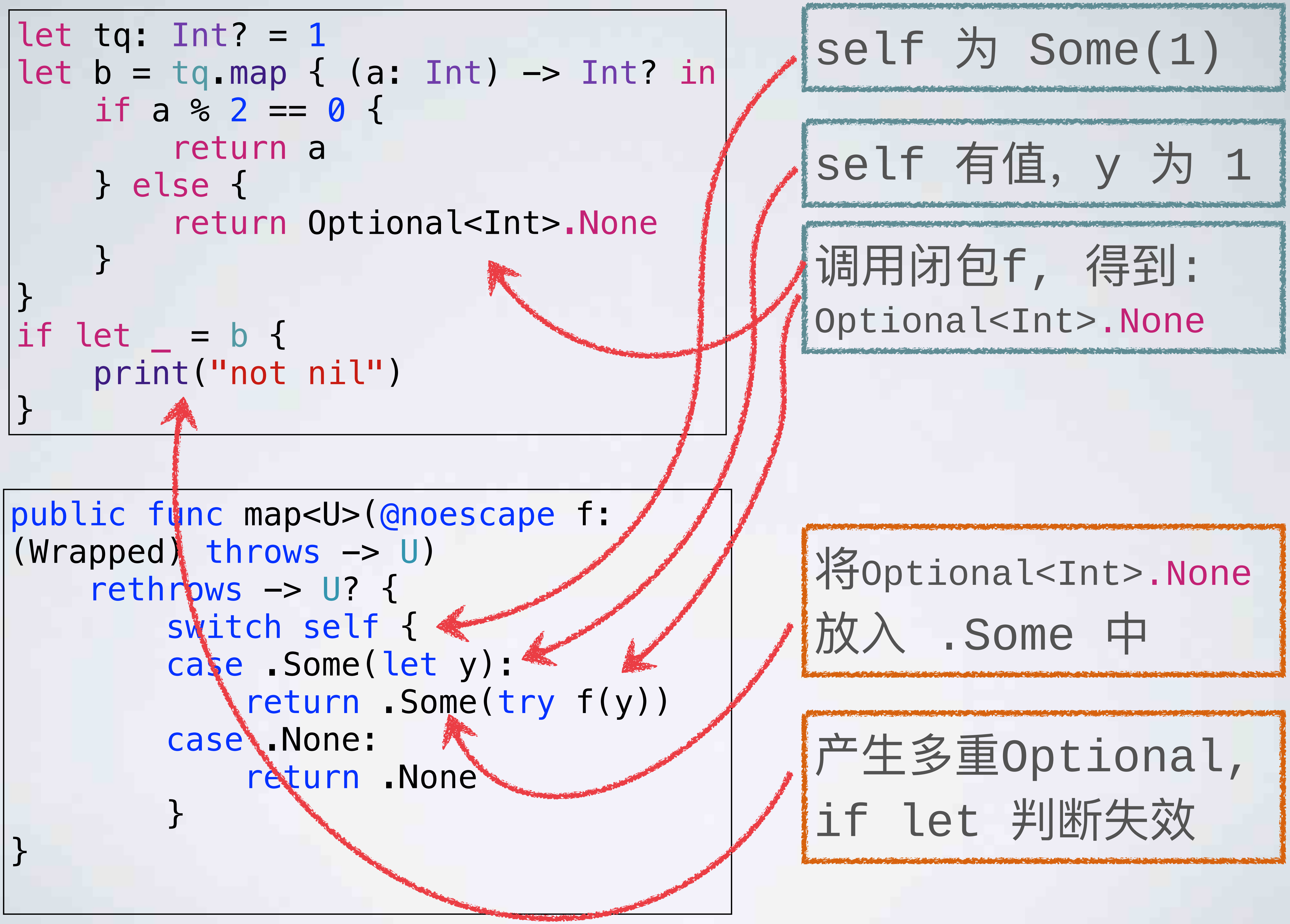
self 为 Some(1)

self 有值, y 为 1

调用闭包f, 得到:
Optional<Int>.None

```
public func map<U>(@noescape f:
(Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

将Optional<Int>.None
放入 .Some 中



```
let tq: Int? = 1
let b = tq.map { (a: Int) -> Int? in
    if a % 2 == 0 {
        return a
    } else {
        return Optional<Int>.None
    }
}
if let _ = b {
    print("not nil")
}
```

```
public func map<U>(@noescape f:
(Wrapped) throws -> U)
    rethrows -> U? {
        switch self {
        case .Some(let y):
            return .Some(try f(y))
        case .None:
            return .None
        }
}
```

self 为 Some(1)

self 有值, y 为 1

调用闭包f, 得到:
Optional<Int>.None

将Optional<Int>.None
放入 .Some 中

产生多重Optional,
if let 判断失效