**Defer, Panic & Recover**

Go has a special statement called defer which schedules a function call to be run after the function completes. Consider the following example:

package main

import "fmt"

func first() {

fmt.Println("1st")

}

func second() {

fmt.Println("2nd")

}

func main() {

defer second()

first()

}

This program prints 1st followed by 2nd. Basically defer moves the call to second to the end of the function:

func main() {

first()

second()

}

defer is often used when resources need to be freed in some way. For example when we open a file we need to make sure to close it later. With defer:

f, \_ := os.Open(filename)

defer f.Close()

This has 3 advantages: (1) it keeps our Close call near our Open call so it's easier to understand, (2) if our function had multiple return statements (perhaps one in an if and one in an else) Close will happen before both of them and (3) deferred functions are run even if a run-time panic occurs.

**Panic & Recover**

Earlier we created a function that called the panic function to cause a run time error. We can handle a run-time panic with the built-in recover function. recover stops the panic and returns the value that was passed to the call to panic. We might be tempted to use it like this:

package main

import "fmt"

func main() {

panic("PANIC")

str := recover()

fmt.Println(str)

}

But the call to recover will never happen in this case because the call to panic immediately stops execution of the function. Instead we have to pair it with defer:

package main

import "fmt"

func main() {

defer func() {

str := recover()

fmt.Println(str)

}()

panic("PANIC")

}

A panic generally indicates a programmer error (for example attempting to access an index of an array that's out of bounds, forgetting to initialize a map, etc.) or an exceptional condition that there's no easy way to recover from. (Hence the name “panic”)